LAYERWISE SYMBOLIC KNOWLEDGE EXTRACTION FROM DEEP NEURAL NETWORKS

by
Simon Odense

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

City, University of London
School of Mathematics, Computer Science & Engineering

Nov 2019

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I'd like to thank my supervisor, Prof A. Garcez. He has guided me throughout this process with important feedback, thought-provoking discussion, and general support. His passion and dedication to the field of neural-symbolic computing has continually inspired me to learn everything I could. Additional thanks to Dr. T. Weyde and Dr. T. Besold for useful feedback on my work and everyone else in the neural-symbolic community for their dedication to the subject.

Thanks also to the computer science department at City for all the interesting discussions and various forms of support. In particular, the machine learning group with special thanks to Benedikt Wagner and Adam White for discussion and feedback on all the various ideas that were bouncing around our heads at the time.

I'd also like to thank my Family for the encouragement they've given me throughout this process. Finally, I'd like to thank Teddie Brock, for her help proofreading and editing as well as her constant love and support.

Abstract

We examine the feasibility of rule extraction as a method of explanation for neural networks with an emphasis on deep neural networks. This is done by establishing a framework for neural-symbolic computing which gives precise meaning to notions such as fidelity, neural encoding, and rule extraction. Using this framework, we establish semantic and syntactic relationships between different classes of neural networks and different logical systems. This shows that there is nothing inherently different about the computations done by deep neural networks and logical systems. We use this to argue that complexity is the primary difference between neural and symbolic approaches. We develop a measure of complexity and two different rule extraction algorithms using *M-of-N* rules. The first extraction algorithm is a fast decompositional algorithm for Deep Belief Networks that builds on the optimal confidence extraction algorithm. The second algorithm is a parallel search for optimal *M-of-N* rules that implements a hyperparameter that controls the complexity of the extracted rules. We apply this algorithm to a variety of deep networks and find that although differences in architecture, dataset, and learning algorithm influence the complexity of extracted rules, generally only the final softmax layer can be represented simply and accurately with *M-of-N* rules. We conclude by experimenting with the combination of rule extraction from the final layer and importance methods to visualize the inputs to the final layer.

Part I

NEURAL-SYMBOLIC COMPUTING

Chapter 1

Introduction

1.1 Explainable AI

As AI systems become more sophisticated in their capacity to approach a wider variety of sociotechnical problems, an increase in their integration across almost all facets of society appears more likely, if not inevitable. One of the key issues this raises is the explainability of these AI systems. Many people are concerned about the use of AI to make crucial decisions in which there are potential social impacts or even safety issues. This concern is only compounded in the case that the AI functions as a black-box. In many circumstances it is not adequate that an AI simply produces an answer. Instead, we may want further indication of the reasoning used to produce the answer. Accuracy on a test set may provide an initial metric to evaluate an AI system but the system may still learn reasoning procedures which are generally invalid. This may be due to a lack of contextual information in the dataset, which limits the applicability of the learned reasoning. A striking example of this can be found in medical predictions. In one instance, a machine learning algorithm was trained to predict the survival rates of hospital patients based on a combination of symptoms. The predictions were meant to determine whether or not a patient should be discharged from the hospital. When inspecting the decisions made by the AI, it was found that a particularly deadly combination of symptoms was given the green light for discharge. This was because when patients presented these symptoms, they were immediately rushed to the emergency room, boosting their survival rate significantly. The AI system, however, only saw that this combination of symptoms resulted in a higher survival rate and had no means of understanding the underlying reasons behind this increase Caruana et al. [2015]. There are many other instances in which AI systems

develop faulty reasoning because of a lack of contextual information in the dataset.

For these reasons, before deploying AI systems into roles with issues of safety or ethics, we want a guarantee that decisions are not being made based on fallacious reasoning. The simplest criterion we can impose on decisions made by AI is the same one imposed on decisions made by humans: an explanation for the action taken. If an AI can provide not only an answer to a query, but an explanation of the reasoning used to produce the answer, then a human will be able to verify that the reasoning is both sound and just. This prevents not only incorrect reasoning, but potentially malicious use of AI systems in which someone trains an AI with an explicit agenda.

Explainable AI has several other advantages as well. In having an explanation of an AI's reasoning, we can improve our ability to prevent fallacious reasoning and thus develop more robust AI systems. For example, it is possible that an AI can detect features that are statistically relevant in a dataset but have no causal efficacy. Ultimately, this renders a system incapable of generalizing accurately. This concern has been given greater importance with the development of adversarial attacks. Adversarial attacks are techniques used to fool a neural network by modifying input in an imperceptible way that nevertheless causes the network to make a false prediction Goodfellow et al. [2015]. The success of adversarial attacks has effected the confidence that people have in many state of the art AI systems. Explainable AI helps to prevent this by illuminating any faulty decision making in a network that could potentially be exploited by an adversarial attack. Furthermore, explainable AI has potential to help us formulate scientific hypotheses from data that experts may be unable to identify due to the massive amounts of available data. Finally, the potential for explainable AI to provide insight into core cognitive processes cannot be understated. For example, an AI that passes the Turing test but is incomprehensible may have practical implications although it tells us very little about how the mind works.

The issue of explainable AI seems, at first glance, to be primarily a practical one. However, in attempting to properly address the question, one cannot avoid serious philosophical questions about the mind. At the heart of the issue is the distinction between abstract and intuitive reasoning. Whereas abstract

reasoning relates conceptual information via universally agreed upon deductive rules, intuitive reasoning has no concrete explanation but rather relies on a multitude of subtle influences to make a conclusion. The initial ideas about AI were born out of a computational theory of the mind in which mental processes were thought of in an abstract computational framework Russell and Norvig [2010]. This lead to the development of symbolic systems in which reasoning processes were made explicit and encoded into systems that could make predictions based on current knowledge. Over the years, the symbolic approach has declined in popularity while more adaptable methods which do not attempt to model abstract reasoning have grown in popularity. Currently, connectionist systems are among the most commonly used AI systems. Although they are more flexible than the earlier symbolic systems, they lack the inherent explainability of the previous systems. One method of addressing this problem is the use of rule extraction algorithms; Rule extraction aims to identify interpretable symbolic rules whose behaviour corresponds with the neural network. Generally speaking, the set of symbolic rules that can be used as explanations for a neural network is exponential and makes exhaustive searches intractable. In addition to computational difficulty, rule extraction is complicated by the presence of competing objectives, namely accuracy and complexity. As we will see, for most significant practical applications the difference between a symbolic and connectionist system is mainly representational. This means that in most cases we can represent any neural network with a corresponding symbolic system to an arbitrary degree of accuracy. However, the symbolic system itself may be no more interpretable than the network. The goal, then, is to find symbolic systems which faithfully capture the behaviour of the network while being of limited complexity. The existence of such rules is by no means guaranteed and in many cases it seems reasonable to assume that a network cannot be represented by an interpretable set of rules.

In the domain of deep learning, neural networks are arranged hierarchically with lower layers feeding into higher layers. The use of rule extraction as a method of explanation for deep networks has been limited. This is perhaps mainly due to the fact that a full symbolic account of a deep network would have to itself be hierarchical, with rules explaining lower levels of a network feeding into rules explaining higher levels. Errors generated in one layer of the symbolic system are propagated upwards and compounded potentially causing a dramatic drop in overall fidelity. This issue, combined with the large number

of hidden units in a deep network has meant that layer-wise rule extraction as a method of explanation for deep networks has not often been attempted. More often, model-agnostic methods which ignore the underlying structure of the model are employed. Although these have been effective to some degree, ignoring the structure used by a model makes evaluation of explanations more difficult. How can we be certain that the explanations given are an accurate representation of the reasoning employed by the model?

Although the hierarchical nature of deep networks provides a barrier to the application of layer-wise rule extraction as an effective method of explanation, it also provides an opportunity for its use in a more limited scope. By applying rule extraction to higher layers we can explain the reasoning of the model in terms of the input to those layers. By treating the inputs to higher layers as atomic concepts to be explained with model-agnostic methods we can explain more of the structural reasoning of the network through rule extraction than we could by using model agnostic methods on their own.

In order for this to work we must first examine the effectiveness of rule extraction for providing interpretable explanations of different layers of a deep network before deciding how to apply it in conjunction with other methods. We begin by reviewing the difference between symbolic and connectionist AI and the methods that have been develop to integrate the two before analyzing previous rule extraction algorithms. From there we develop the necessary preliminaries and present two new *M-of-N* extraction algorithms; one of which is fast but relies on a weaker heuristic and the other which is slower but carries out a more thorough search. The first algorithm can be applied to large networks in which more thorough searches are not feasible. The second algorithm can be applied to small and medium sized networks and will mainly be used to evaluate the layer-wise explainability of deep networks.

1.2 Neural-Symbolic Computing

Traditionally, there have been two approaches to AI: the symbolic approach and connectionism. Symbolic AI is based on a computational theory of the mind, that is one in which the mind is viewed as a computational model operating on abstract symbols in a syntactic way. The goal of symbolic AI is to come up with computation models that resemble cognitive processes. In this view,

cognitive processes relevant to AI operate at the level of abstract concepts rather than low-level sensory information; hence in symbolic AI the semantic meaning of a symbol is not as relevant as its syntactic relationship to other symbols. Over the years many symbolic systems have been described. Often these systems fall into the category of logic programming. Logic programming, in which the central object of study is the *logic program*, is an attempt to model computation in a logical framework. A logic program is made up of a set of sentences called rules which are typically written as $A \leftarrow X_1, ..., X_n$. Here the truth of the right hand side (called the body or antecedent) implies the truth of the left hand side (called the head or consequence). Logic programs can be seen as a computational model operating on a set of concepts by considering the implications of the concepts an agent holds true at a given time. In many cases, it is desirable to have a guarantee that an initial set of beliefs will settle on a stable set of unchanging beliefs. This is guaranteed in one important restriction of logic programming in which a rule is restricted so that there are no negations in the body. Clauses of this form are called Horn clauses and the convenience of their semantics has led to them being extensively studied. many other variants of logic programming also exist, some of which have stable model semantics and some of which do not.

Connectionism takes a more bottom-up approach to AI. Instead of abstract symbols and rules, Artificial Neural Networks (ANNs) are the main object of study in connectionism. ANNs are biologically inspired models that consist of a large number of simple computational units called neurons (also referred to as units or nodes) which interact with each other via weighted connections. By adjusting these weights in response to observed data, ANNs can be trained to perform a large number of tasks often with little or no hard-coded background knowledge (although the design of the network and training procedures may be influenced by the desired task). This flexibility has lead to ANNs being deployed in a large number of industrial and academic settings. Despite symbolic AI being more popular in the 1960s and 1970s, the success of ANNs has lead to connectionism becoming the dominant force in contemporary AI. Philosophically, the connectionist position is distinct from symbolic AI in that it views many cognitive processes as taking place at the *subsymbolic* level instead of the symbolic level. The argument is that many thoughts or ideas cannot necessarily be explained solely in terms of other previously held abstract ideas. Rather, they can only be explained by looking at the level of

neurons. Even if an idea can then be verified through symbolic systems, the source of the idea is not the result of a symbolic system but of a subsymbolic system. The subsymbolic position was fully elucidated during the early development of connectionist systems making it a distinct philosophical position in cognitive science rather than a simply practical approach to problem solving in AI Smolensky [1988]. Much discussion on the subsymbolic vs. symbolic debate was generated including a rebuttal highlighting some of the challenges connectionism must overcome in order to be viewed as a general cognitive model McCarthy [1988].

Many experts in AI have long observed that many of the strengths and weaknesses of connectionism and symbolic AI are complementary. The flexibility of connectionism is lacking in symbolic AI whereas the transferability and interpretability of symbolic AI is lacking in connectionist systems. This has lead to the development of neural-symbolic systems, systems looking to integrate the connectionist and symbolic approach in a way that can capitalize on the benefits of both systems without the disadvantages of either. The neural-symbolic approach can be divided into three categories: The implementation of symbolic systems with connectionist systems, the extraction of logical systems from connectionist systems, and the creation of new systems with both connectionist and symbolic components. The first category, the encoding of symbolic knowledge in connectionist systems, has been given much academic attention Garcez et al. [2008]. Not only does this area have practical applications, but the fact that propositional modal logic can be encoded in a neural network at least partially addresses the "propositional fixation" identified as one of the epistemological challenges of connectionism Garcez [2005]. The foundation of this encoding is the CILP (Connectionist Inductive Logic Programming) Garcez and Zaverucha [1999]. At its most basic, CILP implements the immediate consequence operator of a Horn Logic Program. Running the CILP system for enough iterations will result in the network entering a stable configuration corresponding to the least-fixed point of the immediate consequence operator. This means that CILP effectively defines the semantics of the corresponding Horn Logic Program. Modal propositional logic programs can be encoded using an ensemble of CILP networks. Prior to CILP, a similar method of encoding propositional knowledge into ANNs was given by Knowledge Based Artificial Neural Networks(KBANN) Towell and Shavlik [1994]. As we will see, the encoding and extraction of symbolic rules to/from neural net-

works is almost always – at least in theory – possible. However, in doing so we often lose the very advantages that make one system appealing over another. In order to properly understand these difficulties we must first formalize the relationship between connectionist systems and neural networks.

1.3 Outline

With the need for explainability in AI becoming more apparent, the question of rule extraction has attracted renewed interest. Despite decades of work, rule extraction remains unable to be used as a general-purpose explainability technique. Some researchers point to a fundamental incompatibility between neural-networks and symbolic systems as a likely culprit, but the formal distinction between the two approaches is often unclear, making it difficult to distinguish between what is the result of a fundamental limitation and what is merely the result of practical difficulties. In this thesis, we attempt to provide an answer to that question using two different approaches; the first theoretical and the second empirical. The first approach is the development of a concrete theoretical framework in which to understand the relationship between neural networks and symbolic systems. Within this formalism, we elucidate the difference between semantic and syntactic relationships and outline theoretical as well as practical limitations to the different approaches. This allows us to recontextualize existing neural-symbolic algorithms in such a way that makes clear exactly how different classes of neural networks can be expressed as symbolic systems and vice-versa. We add to this literature by providing several fundamental baseline relationships between propositional logic and various feed-forward network architectures. With the theoretical equivalences clarified, we develop a rigorous notion of fidelity that generalizes various other notions of fidelity that have been used in the past. Turning our attention to rule extraction specifically, we make the argument that complexity is the sole theoretical factor that distinguishes connectionist and symbolic models, at least from the perspective of explainability.

The second approach we take is empirical. We develop two rule extraction algorithms based on the *M-of-N* framework. The first being a fast decompositional algorithm for RBMs (Restricted Boltzmann Machines) that can be applied iteratively to explain DBNs (Deep belief Networks). The second algorithm searches over a space of *M-of-N* rules ordered by weight and uses a

complexity measure along with an approximation of fidelity as a regularized loss function. By weighting the complexity loss with a hyperparameter we can vary the trade off between fidelity and complexity that our extraction algorithm produces. This allows us to empirically address the question of whether or not there exist sufficiently simple rules that can accurately explain a deep network. We test this question with deep CNNs (Convolutional Neural Networks) trained on multiple datasets, using different transfer functions, and with different learning algorithms. Applying our search in a layerwise fashion we find that the internal layers of a deep networks are in general too complex to be considered adequately explainable with rule extraction. However, under most circumstances the final softmax layer is remarkably explainable. We use this result to justify further experiments in which rule extraction is applied to the final layer and other explainability methods are used to visualize the inputs to the final layer. With these experiments we give a proof of concept for a new method of modular explainability. Rather than use a single approach to provide an explanation we use rule extraction in conjunction with other methods to explain features of a deep network at different levels. Output labels are described in terms of a logical rule system whose inputs are explained visually. These modular explanations mirror the distinction between symbolic and subsymbolic computing making them an extremely appealing approach to explainability. We also provide some initial experiments on the effect of final layer rule extraction on robustness, indicating that final layer rule extraction provides a small improvement to robustness against numerous adversarial attacks. We conclude by discussing how the modular approach could be improved and extended.

In chapters 2 and 3 we give a brief overview of neural networks and logical systems, reviewing the required preliminaries and clarifying notation. In chapter 4 we develop our framework for neural-symbolic integration. This includes all relevant definitions as well as several theorems relating feed-forward networks to propositional logic along with key observations and identities. We also discuss computational issues relating to recurrent networks including the potential importance of hypercomputation. In chapter 5 we review previous extraction and encoding techniques used in neural-symbolic computing. We describe the encoding techniques inside our formalism allowing us to more clearly map out the existing relationships and those that still need to be established. Moving on to the empirical portion of the thesis we discuss the

approach of layerwise rule extraction in chapter 6. This involves the description of a fast *M-of-N* extraction algorithm along with experiments applying it to RBMs. In chapter 7 we describe our slow extraction algorithm in detail. chapter 8 contains the results of all experiments conducted using the slower *M-of-N* extraction algorithm algorithm. This includes both the characterization of the fidelity/complexity relationship along with our experiments in modular explainability approaches. In the concluding chapters we review our theoretical and experimental results and discuss future directions for the research.

1.4 Objective and Contributions

The objective of this thesis is to examine the limitations and potential of rule extraction as a method of explainability for neural networks and, in particular, deep networks. Although plenty of rule extraction algorithms exist, they are unable to explain neural networks in general. Furthermore, many have criticised the approach of rule extraction based both on practical and philosophical grounds. The practical argument maintains that the distributed nature of neural networks along with the large number of parameters makes rule extraction unlikely, if not impossible. The philosophical argument views neural networks as operating in a fundamentally different way than symbolic systems, making the extraction of rules from a neural network futile. We contribute to this discussion by systematically addressing this question. We develop an analytic framework help formalize the arguments against rule extraction into a precise quantitative question. We develop multiple extraction algorithms to give empirical evidence for the relative explainability of a deep network. The contributions are summarized as follows

- The development of a theoretical framework for neural-symbolic integration
- The clarification of several issues facing neural-symbolic integration using this framework. These include the distinction between semantic and syntactic encodings as well as the reduction of the arguments against rule extraction to a precise issue of complexity
- The formalization of several equivalences between neural networks and propositional logic that had yet to be elucidated as well as the contextualization of previous neural-symbolic results into the framework

- The development of a fast *M-of-N* extraction algorithm based on confidence rules that can be applied to deep networks
- Several results outlining the limitations of the decompositional extraction of confidence rules from DBNs
- The development of a slower *M-of-N* extraction algorithm that can explicitly relate complexity and accuracy
- The experimental application of *M-of-N* rule extraction to a variety of deep neural networks layer-by-layer, identifying the final layer of CNNs as a good target for rule extraction as well as providing evidence for the claims against full end-to-end rule-based explanation methods
- Proof of concept experiments for a new, modular approach to explainability that combines visualization methods with final-layer rule extraction
- Preliminary experiments on the effect of final-layer rule extraction on robustness

The following publications contain material in this thesis

- Simon Odense, Artur Garcez, *Confidence Values and Compact Rule Extraction From Probabilistic Neural Networks*, NeSy 2017
- Simon Odense, Artur Garcez, *Extracting M of N Rules from Restricted Boltzmann Machines*, ICANN 2017

Both publications relate to the extraction of *M-of-N* rules from RBMs. The first publication deals with the issues inherent to using conjunctive confidence rules covered in sections 6.1 and 6.2. The second publication contains the details of algorithm 2 in section 6.4 along with experimental results.

Chapter 2

Artificial Neural Networks

2.1 The Idealized Neuron

In order to study neural computation we begin by reviewing its basic definitions. The central object of study in neural computation is the *artificial neural network* (ANN). A neural network is a highly parallel model consisting of simple computational units called neurons which communicate to each other via a set of connections called weights. Artificial neural networks were developed as a computational model meant to replicate the behaviour of the biological neural networks found in the nervous system McCulloch and Pitts [1943]. Although substantial differences exist between the neurons found in the nervous system and the idealized neurons used in artificial neural networks, the central idea is preserved. A neuron receives input from the neurons to which it is connected. If the total input crosses a certain threshold the neuron will fire, sending the signal to all the other neurons it is connected to Hodgkin and Huxley [1952]. To be explicit, we denote a neuron by a variable x_i, the output of the neuron is denoted $O(x_i)$ and its input $I(x_i)$. In the basic conception of an ANN the input of x_i is determined by a weighted sum of the outputs of the neurons connected to x_i, in other words $I(x_i) = \sum_j w_{j,i} O(x_j) + b_i$. Here $w_{j,i} \in \mathbb{R}$ is the connection from neuron x_j to x_i and $b_i \in \mathbb{R}$ is an additional parameter called the bias. The output is then defined by the simple step function defined by $f(x) = 1$ if $x \geq 0$ and $f(x) = 0$ otherwise. This gives us $O(x_i) = 1$ if $I(x_i) \geq 0$ and 0 otherwise.

The power of neural networks comes from their ability to adapt via a *learning algorithm*. Given some test input/output examples (supervised) or input ex-

amples (unsupervised), the neural network adapts by adjusting its weights in order to mimic the desired behaviour. The ability of neural networks to learn has made them an invaluable tool for AI.

The basic neuron described above can be generalized considerably by allowing for various transfer functions (also called an *activation function*). A transfer function is a function $g : \mathbb{R} \to \mathbb{R}$ which maps the input of a neuron to its output. We can further generalize the artificial neuron by allowing the output to take multiple or continuous values. The output of a neuron is then given by $O(x_i) = g(I(x_i))$. Transfer functions for neurons taking continuous values are generally continuous. Popular transfer functions include $tanh$, the logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$, and the rectified linear function,

$$Relu(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

The output equation is then written as $x_i = g(\sum_j w_{j,i}x_j + b_i)$. A neural network, N, defines a function on its state space by $N(x_1, ..., x_n) = (O(x_1), ..., O(x_n))$. We will use N^t to denote the function resulting from the application of N repeatedly t times. From this point on, we will generally suppress the notation for input output and transfer function and simply use x_i to refer to the output value of neuron x_i. The application of a neural network is largely dependent on the connectivity pattern of the weights, the choice of transfer function, and the specific learning algorithm. The remaining sections in this chapter are dedicated to reviewing the popular variations of neural networks that are most relevant for neural-symbolic computing.

2.2 Feed-Forward vs. Recurrent Neural Networks

The behaviour of a neural network is constrained by its architecture. The most significant distinction is between feed-forward and recurrent networks which we will discuss here. In order to understand the architecture of a neural network we discuss how the weights of a network can be converted to a graph before going over the most pertinent examples of feed-forward and recurrent networks.

The weights of a neural network define a connectivity graph between neurons.

From a neural network, we can extract a graph with nodes corresponding to the neurons and edges connecting nodes which have a non-zero weight. To be precise, given nodes i, j corresponding to neurons x_i, x_j, there is an edge $e_{i,j}$ if and only if $w_{i,j} \neq 0$. A neural network is called feed-forward if its graph has no cycles and recurrent otherwise. Feed-forward networks can be thought of as functional approximators representing a function from a set of input neurons to a set of output neurons. The most basic feed forward network is the perceptron Rosenblatt [1957] which consists of a single layer of input neurons fully connected to a single layer of output neurons. The neurons are assumed to take binary values while the transfer function is a simple step function. Equipped with a learning algorithm called the perceptron learning algorithm, a perceptron can iteratively update its weights based on example input/output pairs. The perceptron algorithm works as follows: given examples from a training set, S, and a initial set of weights, w, if the perceptron predicts 1 when the correct output is 0, update the weight by $w \leftarrow w + x$. If the perceptron predicts 0 when the correct output is 1, update the weight by $w \leftarrow w - x$. Repeat until convergence or for a fixed number of iterations. By taking examples from a training set, the perceptron will hopefully learn an input/output relationship that can generalize correctly to unseen examples. The generalization capabilities of a neural network are evaluated using a test set which is a set of examples that are not used in the learning algorithm but reserved to evaluate the accuracy of the final model. The perceptron learning algorithm is an example of a supervised learning algorithm since the target output is given to the algorithm to calculate the weight update.

Although a perceptron is adequate for many classification tasks, it can only represent functions which are linearly separable. That is, functions whose output classes can be separated with a linear decision boundary. Functions whose output classes cannot be separated by a linear decision boundary, for example the XOR function, can not be expressed with a perceptron. This limitation can be overcome by adding a hidden layer of neurons between the input and output layers with a nonlinear activation function. This modification turns the simple perceptron into a multi-layer perceptron (MLP). Unlike simple perceptrons, the MLP is a universal functional approximator Hornik et al. [1989]. This means that any continuous function on a compact set can be represented by an MLP. Rather than use the perceptron learning algorithm, MLPs learn by using gradient descent. In gradient descent an error function is minimized

by calculating the gradient of the error function with respect to the model parameters and updating the parameters to move down the gradient. Ideally, over time a minimum of the error function will be reached. This can be complicated by the existence of local minima in the error function leading to the adoption of many different strategies to combat this. The most common error function is the mean squared error defined by $E = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$ where Y_i is the output label associated to example i, $\hat{Y}_i$ is the output label predicted by the network. Given an example, or set of examples, each parameter of the network, w_i, is updated using the update rule $w_i \leftarrow w_i + \alpha\frac{dE}{dw_i}$ where α is a hyperparameter in the range $[0, 1]$. In order to calculate the gradient, MLPs use a technique called *backpropagation* in which the error at the output layer is propagated backwards in order to calculate the error accumulated in the previous layer. Rather than use the chain rule to compute $\frac{dE}{dw_i}$ for each w_i at each layer, backpropagation calculates a quantity δ^l that can be used to calculate the gradient for the parameters in each layer l. δ^l is defined recursively. Starting with the final layer L, we have $\delta^L = (g^L)'\nabla_{a^L}E$ where $(g^L)'$ is the derivative of the activation function of layer L evaluated at the inputs to layer L and $\nabla_{a^L}E$ is the gradient of the error function with respect to the activation values of the neurons in layer L for the test example. Moving on we have $\delta^{l-1} = (g^{l-1})' \cdot (W^{l-1})^T \cdot \delta^l$ where $(g^{l-1})'$ is the derivative of the transfer function at layer $l-1$ evaluated at the inputs to layer $l-1$, and $(W^{l-1})^T$ is the transpose of the weights at layer $l-1$. The name backpropagation comes from the recursive calculation of δ. The actual gradient with respect to a weight in layer l is given by $\delta^l \cdot a^{l-1}$ where a^{l-1} is the activation values of the neurons at layer $l-1$. Given an input example, backpropagation uses a forward pass to calculate the input values at each layer as well as the activation values at each layer. It then does a backward pass calculating δ at each layer and uses this to calculate the gradient of the error function with respect to the weights at that layer. Backpropagation has become an essential tool in the training of neural networks Rumelhart et al. [1986]. Further refinements to the supervised learning algorithm involve the use of regularization terms on the error function, additional terms taking into account the second derivative for the weight update, the inclusion of momentum in the weight update, and many more. Bengio [2012] gives a practical overview of the various techniques for optimizing the learning process and their application.

Feed-forward neural networks are most commonly treated as functions. Given

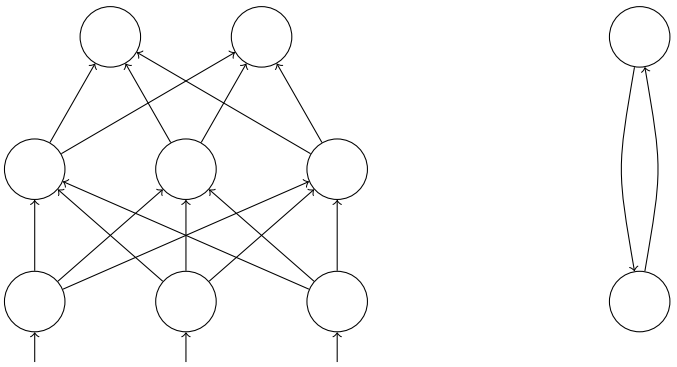


Figure 2.1: Example architecture for a feed-forward (left) and recurrent (right) network

an input pattern, they produce an output by calculating the activations of each successive layer until the end of the network is reached. This means that the activations of neurons in layers not currently being calculated are generally ignored. For practical purposes this is not important, however it adds a technical quirk to our theoretical analysis. As we define them, neural networks start in an initial state and update the activations of every neuron at each time step. Although there are frameworks where activations are updated asynchronously, for simplicity, we generally limit our discussion to networks which update synchronously. One issue with this definition is that the input neurons in a feed forward network, having no connections or bias, will always revert back to the same state regardless of the input configuration. In some instances this is desirable (for instance, computing the minimal model of a Horn logic program, as we will see in chapter 5). Unfortunately, it fails to capture the way in which feed-forward networks are usually used. Luckily, we can rectify this by giving the input units self-connections of 1, a bias of 0, and a linear activation. This ensures that the input neurons will remain in their initial state. When giving an initial state, we must also choose one for the hidden layers. Because we do not care what the initial activation of the hidden layers is, we will consider all initial states equivalent if they differ only by the value of hidden units which have not been calculated by previous layers. We will formalize this when we discuss proposition encodings of feed-forward networks. Although technically this modification makes the network recurrent (due to the self connections), we will consider feed-forward networks as either those which are feed-forward by definition or those described above. Recurrent neural networks are most often used to model time series. Recurrent neural networks most often have a visible and hidden layer similar to MLPs. The difference between recurrent networks and feed-forward networks is due to the recurrent connections within

the hidden and/or visible layer which allows the network to model time dependency. Recurrent networks are most often used for unsupervised learning tasks where the goal is predict the next m steps of a time series given the previous n steps. Learning in recurrent networks can be difficult, often requiring the use of backpropagation through time which is vulnerable to problems such as vanishing gradients in addition to being computationally difficult for long time series. Despite this, there are numerous powerful recurrent networks such as the Long-Short Term Memory and Recurrent Restricted Boltzmann Machine which have successfully been applied to an number of tasks Hochreiter and Schmidhuber [1997], Sutskever et al. [2008]. Although recurrent networks in general are capable of more complicated dynamic behaviour than feed-forward networks, some recurrent networks share the property that, given any starting state, eventually the network will settle to a stable state. Neural networks with this property are an important class of networks for machine learning; formally defined as follows

Definition 2.2.1. *A neural network, N, is stable if for all initial states, x_0, there exists $t > 0$ such that for all $t' \geq t$, $N^{t'}(x_0) = N^t(x_0)$*

Next we examine what is perhaps the most well-known class of stable recurrent networks.

2.3 Symmetrically Connected Networks

One of the most important kinds of recurrent neural networks is the Symmetrically Connected Network (SCN). A symmetrically connected network is a recurrent neural network which satisfies two properties, the first is that for all i, $w_{i,i} = 0$, the second is that for all i, j, $w_{i,j} = w_{j,i}$ hence the name symmetrically connected networks. The standard example of an SCN is the *Hopfield network*. The Hopfield network is an SCN in which, like the simple perceptron, the neurons take binary values and the transfer function is given by the standard step function Hopfield [1982]. The other common variant of the Hopfield network is the *Boltzmann Machine*, Ackley et al. [1985]. The Boltzmann machine is a probabilistic analog of the Hopfield network in which the logistic function is used as a transfer function. Because the Boltzmann machine is probabilistic, the transfer function calculates the *probability* that a neuron will be activated rather than its activation value. For instance, in a 3 neuron network with weights $w_{1,2} = 1, w_{1,3} = -11, w_{3,2} = 2$ and biases uniformly equal

to 0, $P(x_3 = 1|x_1 = 1, x_2 = 1) = \sigma(w_{1,3}x_1 + w_{2,3}x_2 + b_3) = \sigma(0) = 0.5$. When running an SCN, the neurons are usually updated asynchronously, that is they updated one at a time rather than all at once.

The dynamics of an SCN are governed by the following energy function $E(x) = \frac{1}{2}\sum_{i,j} x_i x_j w_{i,j} + \sum_i b_i$. Although SCNs are recurrent networks, (because $w_{i,j}$ and $w_{j,i}$ form a path from i to itself for all i) they are different from many recurrent networks because they are guaranteed to settle to a stable state (or a stable distribution in the case of Boltzmann machines). This is because each time the network is updated it will move to a state of either the same, or lower energy. Because there are only a finite number of states the network much eventually reach a stable state. For this reason, Hopfield networks are thought of as associative memories in which the stable states are the stored memories and the starting configurations are the noisy memories meant for retrieval. Boltzmann machines, on the other hand, are used to model probability distributions.

SCNs are trained using Hebbian learning. In Hebbian learning weights are updated based on the mutual firing patterns of the two connected neurons Hebb [1949]. We can store a single pattern of activation by setting each weight $w_{i,j} = x_i x_j$. For multiple patterns we simply take the sum of the pairwise activations over every pattern. In Boltzmann machines the corresponding update rule is $E[x_i x_j]_0 - E[x_i x_j]_{\text{inf}}$ where the first term is the expected value in the desired distribution and the second term is the expected value in the Boltzmann distribution of the network. Since calculating the expected value in the Boltzmann distribution involves summing an exponential number of terms this becomes quite difficult.

Often SCNs are partitioned into visible and hidden units. The network dynamics are unchanged but we are only concerned with learning patterns of the visible units. The hidden units exist solely to provide computational power. For example, we may wish to store a number of m-dimensional patterns in a Hopfield network with m visible neurons and k additional hidden neurons. The capacity of this network will be larger than one without the hidden units.

SCNs have played an important role in the modern development of deep learning LeCun et al. [2015]. In order to help train deep networks, a particular variant of the Boltzmann machine known as the Restricted Boltzmann Ma-

chine (RBM) was used. An RBM is a Boltzmann machine which is partitioned into a set of visible and hidden units. The RBM is further restricted by disallowing connections with the visible and hidden layer. In order to use RBMs to pretrain a deep network, first an RBM is trained on the input distribution. This is done using an algorithm called contrastive divergence, which uses Gibbs sampling to approximate the log-likelihood of a pattern in the Boltzmann distribution Hinton [2002]. Once the first RBM is trained, a second RBM is placed on top of the first and trained on the hidden distribution of the first RBM conditioned on the visible distribution. This process is iterated until the desired number of layers is achieved. A softmax layer is then placed on top of the final RBM and the whole network is trained using backpropagation in order to fine-tune the weights. By pre-training with RBMs, we are essentially initializing a deep network so that it will be closer to the desired solution.

The success of unsupervised pre-training in deep learning lead to an explosion in the application of deep networks. Today other methods of deep learning are more popular but the use of RBMs in deep networks remains an important breakthrough in the field of deep learning.

2.4 Deep Neural Networks

The explosion in neural networks is in large part due to the effectiveness of deep neural networks. These are our central object of concern for rule extraction and so we will review the basic definitions as well as some of the tentative theory that has been developed to explain their effectiveness.

Deep neural networks are neural networks which have multiple (often many) hidden layers. Deep networks can be feed-forward or recurrent but the networks we will focus on are deep feed-forward neural networks. Although the universal approximation theorem for neural networks shows that any measureable function on a compact set can be expressed as a feed forward network with a single hidden layer Hornik et al. [1989], deep networks are often thought to be able to represent functions more efficiently. This has some theoretical support assuming certain conditions Delalleau and Bengio [2011], Mhaskar et al. [2016], but it is not likely true that a deep architecture will be more efficient (in terms of number of neurons, parameters, or training time) for every conceivable problem. This can be seen as an instance of the 'no free lunch' theorem in

machine learning which states that over the entire problem domain, no learning algorithm is better on average Wolpert and Macready [1997]. The immediate implication of the no free lunch theorem is that in order to achieve exemplary performance on some set of tasks, it may be necessary to equip the learning algorithm with an appropriate inductive bias. Furthermore, although this inductive bias will be beneficial for a certain subset of tasks, it will likewise make our algorithm perform worse on a different set of tasks. A universal inductive bias, one which is beneficial for all learning tasks, does exist, and it has been argued that the no free lunch theorem is thus not a real constraint on machine learning Lattimore and Hutter [2011]. However, the universal inductive bias is uncomputable and whether approximations to this bias can prove useful to machine learning is an ongoing research problem. Luckily, most kinds of tasks that we are interested in modelling with AI share similar properties which we can exploit with neural architecture and/or inductive biases. In particular, many datasets in AI can be represented with a hierarchical structure. This argument has been extended to physics where it has been argued that the prevalence of low-dimensional Hamiltonians in nature makes hierarchical representations more efficient for most tasks to which deep learning is applied Lin et al. [2017].

The standard method of learning in deep networks is gradient descent through back-propagation. The computational difficulty of this task prevented the widespread use of deep networks until a major breakthrough showed that a deep-network could be effectively pre-trained by greedily training RBMs stacked on top of one another Hinton et al. [2006]. Doing so results in a Deep Belief Network (DBN), a type of neural network (or generative graphical model depending on your view) consisting of layers of latent variables with connections running between one layer and then next but with no connections within a layer. The procedure for doing this is as follows: begin by training an RBM on the dataset using contrastive divergence. Next, train another RBM on the conditional distribution of the hidden units of the first RBM given the dataset. Repeat this process for as many layers as desired. Each time we stack another RBM on top of our network, we are adding another layer to the deep network by training it to learn a hidden representation of the previous layers hidden units. In doing this, ideally the network will learn a representation of the input data that makes supervised learning converge to an accurate solution faster. Following the unsupervised pre-training, we enter a 'fine-tuning'

phase where back-propagation is applied in the standard way. If the network was able to find a suitable representation of the input data, then the network we begin back-propagation with will be much closer to a solution than one that was initialized randomly. A common alternative to RBMs is to use autoencoders in the pre-training phase. Autoencoders are feed-forward networks with a single hidden layer which are trained using back-propagation but with the output layer made identical to the input layer. In other words, it learns an encoding of the data in the hidden layer which it then decodes to produce the original input. In either case the idea is the same - to produce a network which, internally, encodes a faithful representation of the data which one can then use for supervised learning. Although these techniques have proven to be effective, the theory behind deep learning has not developed at a similar pace. The intuitive justification for hierarchical representations has yet to be formalized. Some have attempted to use the *information bottleneck* framework to understand the success of deep learning Tishby and Zaslavsky [2015] but counter examples have shown that some deep networks are effective despite not conforming to the optimality conditions of the information bottleneck Saxe et al. [2018]. Aside from the rudimentary theory and intuition we outlined above, the finer details of the efficacy of deep learning remain mysterious.

Following the initial success of unsupervised pre-training, many smaller adjustments to the learning process have been developed to improve deep learning. Examples include changing the activation functions from smooth sigmoidal functions to the piece wise linear Relu units and the use of drop-out. LeCun et al. [2015]. With all of these improvements to technique along with hardware optimization, modern deep learning no longer relies on unsupervised pre-training but it does remain an important tool in the extensive machine learning package that was developed in the subsequent decade. Perhaps the most important tool in that package is the use of convolutional weights which is the topic of the next section.

2.5 Convolutional Neural Networks

The success of deep learning highlights the importance of network architecture in machine learning. Although a single hidden layer in both recurrent and feed-forward networks is sufficient to represent any function, it does not mean that, given a learning algorithm and dataset, a single layer network will

be effective in finding a good solution to the problem. The architecture of a network forces the solution to be represented in a certain form that allows the learning algorithm to have a better chance of converging to the correct general solution. The reasons why deep learning are effective were briefly discussed in the previous section; however, the hypothesis presented there remain largely conjectural. Another structural bias that has become of great importance is the use of convolutional weights. Convolutional weights exploit symmetry in the dataset by restricting the connections of a hidden neuron to a small segment of the total number of input neurons. For each segment, a copy of this neuron is made with identical weight values but connecting to a different segment. Collectively, the neurons with a specific set of weights are known as a *filter*. Convolutional neural networks have the dual advantages of a reduced number of parameters and a natural invariance to translation, a property that is fundamental to natural image data. This is another example of structural inductive bias. Just as deep architectures have naturally model the hierarchical structure of a dataset, the inclusion of convolutions in neural layers injects knowledge about the statistical properties of natural images and other locally invariant datasets directly into its structure. The inclusion of this bias can be mathematically justified assuming that the dataset has a compositional structure Mhaskar et al. [2016]. This puts convolutions on theoretically better footing than deep networks as the use of convolutions in a network can be analytically shown to reduce the number of parameters required to represent a compositional dataset.

Many datasets are compositional, the most obvious of these is image data. For this reason, Convolutional Neural Networks (CNNs) have become a fundamental tool in computer vision. CNNs build the locality of image data directly into the network. To do this, the input image is divided into patches. For example, if the input is a 28×28 image and we want to use 3×3 image patches, we associate a filter to the collection of $28 \times 28 = 784$ image patches each centered at one of the input features. If a patch extends beyond the boundaries of the image then we ignore the portion of the patch outside the image although other padding techniques are sometimes used. Each filter represents a collection of neurons with identical weights but connected to a different patch of the image. Because the neurons in a filter have identical weights, they all represent the same feature. The result of this is that translating a feature in an image will not effect behaviour of the network because there will always be

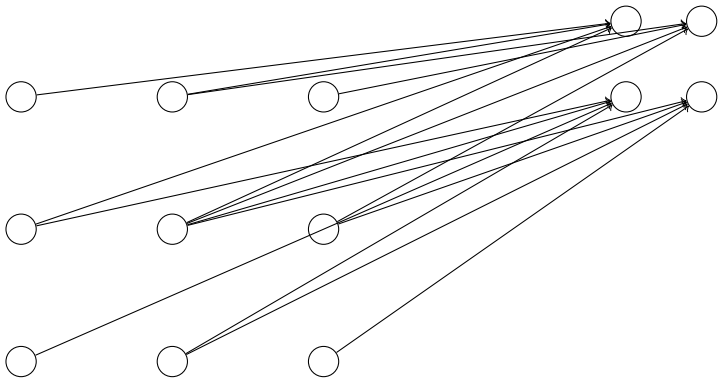


Figure 2.2: A 2×2 filter on 3×3 input.

a neuron in the filter that detects the feature in its image patch.

The combination of deep architectures and convolutional layers has resulted in state of the art image recognition. Deep convolutional networks will be the focus of our rule extraction experiments. Now that we have described the most important neural networks we move on to a description of symbolic systems.

Chapter 3

Symbolic Systems

3.1 Symbolic Reasoning and Logical Systems

In this chapter we review the symbolic systems that are most relevant to neural-symbolic integration. Although it is straightforward to define neural networks, what does and does not count as a symbolic system is more ambiguous. Symbolic systems are intuitively understood as systems which operate by reasoning with or manipulating abstract symbolic data. In chapter 4 we will discuss this in more detail along with a comparison to the intuition underlying connectionism, but for now we will recall the important concepts underlying the kind of symbolic systems we will focus on: logical systems. We will then review those systems that have the strongest connection to neural-symbolic integration.

A logical system consists of a formal language that is interpreted via some kind of semantics along with a collection of deductive rules. The semantics are defined in terms of models. A model is a way of interpreting the abstract entities described by the language that allows one to determine whether or not a sentence is true under that interpretation. Deductive rules are used to derive sentences that are logically follow from another set of sentences. A set of sentences in a logical system is known as a knowledge base. Given a knowledge base, L, we say that L entails l or $L \models l$ if l is true for every model in which all sentences in L are true. We say $L \vdash l$ if there is a deductive rule which derives l from L. A sound deductive system is one in which $L \vdash l$ implies $L \models l$ and a complete deductive system is one in which $L \models l$ implies $L \vdash l$. A sound and complete deductive system is one that exactly describes the semantic relationship between statements in the knowledge base.

These concepts are most easily demonstrated with the prototypical logical system: propositional logic. The language of propositional logic consists of a countable set of propositional variables, $X_1, X_2, ...$ (also known as atoms) and the set of connectives $\{\neg, \rightarrow, \wedge, \vee\}$. A well-formed formula in propositional logic is either an atom, or derived from other well-formed formulas, ϕ, ψ via one of the following rules: $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi$. A truth assignment is a an assignment of either true or false to each of the atoms. A truth assignment on atoms extends to a truth assignment on sentences by the following rules:

$$\neg\phi \text{ is True iff } \phi \text{ is not True}$$
$$\phi \wedge \psi \text{ is True iff } \phi \text{ and } \psi \text{ are True}$$
$$\phi \vee \psi \text{ is True iff } \phi \text{ or } \psi \text{ are True}$$
$$\phi \rightarrow \psi \text{ is True iff } \neg\phi \vee \psi \text{ is True}$$

The symbol $\neg$ is known as *not* or a *negation*, $\wedge$ is known as *and* or a *conjunction*, $\vee$ is known as *or* or a *disjunction*, and $\rightarrow$ is known as an implication. Implications are usually written left to right but sometimes also right to left. The arrow in an implication points from the *antecedent* and to the *consequence*. A double arrow, $\phi \leftrightarrow \psi$, is used as a shorthand for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. A truth-assignment is a model of a sentence if it assigns it a value of True. For convenience, truth assignments in propositional logic are often represented by the set of atoms that are assigned true. For example, given a propositional system with the atoms X_1, X_2, X_3, the truth assignment that assigns true to X_1 and false to X_2 and X_3 is represented as $\{X_1\}$. This notation will be used for logical systems that share the syntax and method for evaluating truth with propositional logic; specifically, logic programming and penalty logic.

Propositional logic is equipped with a set of inference rules that allow you to deduce the truth of a sentence from the truth of other sentences. We will not spend much time discussing inference rules in this thesis; however, several propositional inference rules of propositional logic will be used in our development of a syntactic encoding of feed-forward neural networks into propositional logic. For this reason we will discuss three important inference rules. The first, and perhaps the most famous, is known as *modus ponens*. Given propositional sentences ϕ and ψ, if a knowledge base contains $\{\phi \rightarrow \psi, \phi\}$ then *modus ponens* allows us to deduce ψ. The other rules that will be implicitly referenced are simplification and conjunction. Simplification tells us that from $\{\phi \wedge \psi\}$

we can deduce ϕ and ψ. Conjunction tells us that from $\{\phi, \psi\}$ we can deduce $\{\phi \wedge \psi\}$. Propositional logic will be the foundation for the majority of logical systems we discuss. But first-order logic also plays a central role in logic programming as well as providing an example for a system of semantics not based solely on truth assignments of atomic variables. For this reason we give the basic definitions here.

The symbols used in the construction of first-order sentences can be divided into logical and non-logical symbols. The logical symbols consist of the set of connectives found in propositional logic along with the universal and existential quantifier, $\forall$ and $\exists$ respectively. The non-logical symbols are made up of variables, $z_1, z_2, ...$, constant symbols, $c_1, c_2, ...$, functions $f_1, f_2, ...$ and relations, $P_1, P_2, ...$ Each function and relation symbol has an associated natural number known as the arity which defines the number of arguments taken by the function or relation. Terms in first-order logic are either variables, constants, or a function with each required argument being assigned a term. For example, if f_1 has arity 1 and f_2 has arity 2 then $f_2(f_1(z_3), z_6)$ is a term. Terms are used in conjunction with relation symbols to define atomic formulas. An atomic formula is a relation symbol in which each argument is assigned a term. Atomic formulas can loosely be thought of as the first-order analog of atomic variables in propositional logic and act as the most basic well-formed formulas in first-order logic. From atomic formulas, well-formed formulas are constructed using the logical symbols. The logical connectives carried over from propositional logic are used in the same way to construct well-formed formulas with atomic formulas replacing atomic variables. Quantifiers are used to construct well-formed formulas as follows. If ϕ is a well-formed formula and x is a variable then $\forall x\phi$ and $\exists x\phi$ are well-formed formulas. Here ϕ is known as the *scope* of the quantifier. For example, take the well-formed formula $\exists x(\forall y(P_1(x, y) \rightarrow P_1(y, x)))$ (where P_1 is a relation symbol of arity 2), The scope of the existential quantifier here is $(\forall y(P_1(x, y) \rightarrow P_1(y, x)))$ and the scope of the universal quantifier is $(P_1(x, y) \rightarrow P_1(y, x)))$. If a variable is in the scope of a quantifier it is *bound* otherwise it is *free*. The sentences of first-order logic are the well-formed formulas which contain no free-variables.

The semantics of first order logic are defined in terms of first-order structures. A first-order structure gives an interpretation of the symbols in the language that allows one to determine the truth of any first-order sentence. As with

propositional logic, the set of structures that make a sentence true are known as the models of the sentence. Before we define first-order structures, a small note on terminology. A model is usually reserved for a structure that makes a sentences true. As we've seen, models in first-order logic and propositional logic are first-order structures and truth assignments which make a sentence true. In the standard use of the term, a model is a concept that only makes sense relative to some sentence or set of sentences. Unfortunately, when discussing logical systems in general, there are various types of structures that may be candidate models in the logical system. In first-order logic these are first-order structures and in propositional logic these are truth assignments. When discussing a logical system abstractly, we need a term to refer to the set of objects from which models are taken. We will use the term model in both this context and the traditional context. When we refer to an object as a model without reference to any knowledge-base, we simply mean a structure that interprets the language and may or may not be a model for any particular sentence or set of sentences. When discussing models *of* a knowledge base, then we are referring only to those structures which are a model of the knowledge base. This allows us to discuss the semantics of an arbitrary logical system without making reference to the particular kinds of structures used in the semantics.

Now we are ready to define the semantics of first-order logic. A first-order structure is a set together with a set of functions, relations, and constants for each function, relation, and constant symbol in the language. For each constant symbol, the first-order structure assigns an element of its associated set. For each function symbol of arity n, the first-order structure assigns a function on the set of arity n, and for each relation of arity n, the first-order structure assigns a relation on the set of arity n. In order to see how a first-order structure defines the semantics we must first go over the notion of a grounded formula. A *grounded* atomic formula is a formula in which the arguments of the relation are not variables. In other words, each argument is a constant or a function of constants. A grounded atomic formula has arguments which refer to specific elements in a first-order structure. A grounded atomic formula is true if the corresponding relation in the first-order structure relates the corresponding objects. For example, if $Follows$ is interpreted as the relationship in the natural numbers that is true if the first argument is number immediately following the second argument and the constants c_2 and

c_1 are interpreted as the numbers 2 and 1 respectively, then $Follows(c_2, c_1)$ is true but $Follows(c_4, c_2)$ is not. Truth from the logical connectives is defined in the same way as it is in propositional logic. A universally quantified sentence is true in a first-order structure if the scope of the quantifier is true in the structure when the quantified variable is replaced by any element of the set of objects in the structure. An existentially quantified sentence is true if the quantified variable can be replaced by some element in the set of objects to make the scope true. Note that elements in the set of objects may not correspond to constant symbols. In this case truth from an atomic formula is determined by the corresponding relation in the structure using arguments found in the set. For example, suppose we have a first-order language with 2-ary relation symbol, $Follows$, 1-ary function symbol, $Successor$, and constant symbols c_0, c_1. Suppose also that we have a first-order structure whose set is the natural numbers that assigns $Follows$ to the same relationship in the above example, $Successor$ to the map addition by one, and c_0 and c_1 to be the numbers 0 and 1. Then the sentence $\forall x(Follows(Successor(x), x))$ is true because no matter what number is assigned to x, $Follows(Successor(x), x)$ is always true under the structures interpretation of $Follows$.

First-order logic also has a corresponding set of inference rules that expand on those from propositional logic. All of the inference rules involving the propositional logical connectives remain unchanged with first-order sentences taking the place of propositional well-formed formulas. Additional rules for the introduction and elimination of quantifiers are also included. As we will not make use of first-order deduction rules in our discussion of neural-symbolic integration, we will not go into further detail. The semantics of first-order logic are important in our discussion because they highlight the fact that the way in which truth values are determined by models is highly specific to the logical system. Other logical systems such as modal logic use even more complicated structures to assign truth values to sentences. In order to remain completely general when we give our definition of logical systems, the set of models will be defined simply as a set along with some function that assigns a truth value to each sentence in the language. The nature of this function will be left ambiguous. Now that we have a basic understanding of logical systems along with the two most prominent examples, we move onto the logical system that is most closely associated with neural-symbolic computing: Logic Programming.

3.2 Logic Programming

Logic programming is a symbolic system that adapts the basic principles of logical reasoning in a way that can be easily used in a computational framework. Although logic programming is generally used as a programming language, in many cases it can also be seen as its own logical system that uses the language of propositional or first-order logic but modifies the semantics and deductive system. The logic programming languages we discuss are primarily propositional and these will be the focus of our discussion, but we will also give a brief discussion of first-order logic programming. The language of a propositional logic program consists of propositional sentences that are either a single atom, a conjunction of atoms, or contain a single implication whose consequence (known as the head) is a single atom and whose antecedent (known as the body) is a conjunction of literals (an atom or its negation). A sentence containing an implication in logic programming will be known as a rule and the body will consist of a *clause*. We will use clause to refer to either a conjunction of literals or, later, an *M-of-N* collection of literals. Unlike propositional logic, logic programming writes implications right to left and implications with an empty body are allowed (these are seen as equivalent to simply writing the atom at the head of the implication). With the most basic semantic interpretation, truth assignments assign true or false to propositional atoms (represented as 1 and 0 respectively) with the truth of conjunctions and implications being defined in the same way as propositional logic. From a programming perspective, a logic program represents a set of **IF** ... **THEN** rules which operate on truth assignments of the atoms by repeatedly applying *modus ponens* with the *closed-world assumption*. The closed-world assumption states that if an atom cannot be deduced from a knowledge base it is assumed to be false. For example, consider the following logic program

$$\begin{aligned} A &\leftarrow \\ B &\leftarrow \\ C &\leftarrow A \wedge B \end{aligned}$$

Starting with the values $A = B = C = 0$ we first deduce $A = 1, B = 1, C = 0$ and then deduce $A = 1, B = 1, C = 1$, which is stable under *modus ponens*. Now consider the following logic program

$$B \leftarrow A$$

Starting from the values $A = 1, B = 0$, since there is nothing in the program to prove A, we first deduce $A = 0, B = 1$. Then we deduce $A = 0, B = 0$ which is stable.

The most basic form of logic program is one which contains no negations. An implication of this form is known as a *Horn Clauses*. Horn clauses have the desirable property that iteration of *modus ponens* as demonstrated above is guaranteed to converge to a minimal model. This isn't necessarily the case for general logic programs. To understand this property and what is meant by minimal model, we first define a partial order on the set of truth assignments using subset inclusion. In other words, given two truth assignments M_1 and M_2, we define $M_1 \leq M_2$ if and only if $\{X_i : M_1(X_i) = 1\} \subset \{X_i : M_2(X_i) = 1\}$. This means that every atom that is true under M_1 is also true under M_2. A minimal model is a model which is minimal under this partial order. In other words, M_1 is minimal if and only if $M_2 \leq M_1 \implies M_2 = M_1$.

A logic program, L, defines an operator, called the *Least Fixed Point Operator* on truth assignments defined as $T_L(M) = \{X_i$: There exists a rule, $X_i \leftarrow X_{i_1} \wedge X_{i_2} \wedge ... \wedge X_{i_k} \in L$ with $X_{i_j} \in M$ for all i_j $\}$. A stable model is a model which is a fixed point of T_L. In Horn Logic programming, T_L will always converge to a truth assignment which is minimal and a model of the propositional sentences making up the logic program, furthermore this will be the unique minimal stable truth assignment for L. The semantics of Horn clauses are easily defined with the single model of L being the unique fixed-point of T_L.

Horn clauses can be extended to *general* logic programs through the introduction of negated literals in the body. Introducing negation to logic programming can be a semantic challenge. The standard paradigm is *negation-as-failure* in which $\neg A$ is derived from a failure to derive A. Unfortunately, when negation is introduced, T_L is no longer guaranteed to converge to a unique fixed point. Take for example the program $P \leftarrow \neg P$, for which T_L has cyclic behaviour. Defining the semantics in this case can be more difficult, but for propositional logic programming we can define the semantics in terms of the *completion* of a logic program Clark [1978]. The completion of a propositional logic program is a propositional knowledge base consisting of the following: all the implications of the original logic program with each implication replaced by a double implication (if there is more than one rule with the same head then the body of

the double implication replaced by the disjunction of the bodies of each rule), each propositional atom that is at the head of an implication with an empty body, the negation of each propositional atom in the language that is not at the head of any implication. For example,

$$\begin{aligned} A \leftarrow \\ C \leftarrow A \wedge \neg B \end{aligned}$$

Assuming the language consists of the propositional atoms A, B, C, the completion is

$$\begin{aligned} A \\ \neg B \\ C \leftrightarrow A \wedge \neg B \end{aligned}$$

The models for the logic program are the propositional models of its completion. In several specific cases, the semantics of general logic programs can also be defined in terms of the fixed-points of T_L, for instance when the implications can be placed in a hierarchy in such a way that an atom at the head of an implication only appears in the body of implications 'above' it in the hierarchy. The set of general logic programs is sometimes restricted to a set of 'acceptable' logic programs for which T_L has some kind of desirable behaviour that allows for a simple definition of the semantics. In the most general case, multi-valued semantics and multiple operators may be necessary to define a meaningful semantics for general logic programming. Fitting [2002] Gives a thorough survey of fixed-point semantics for general logic programs.

In our discussion of logic programming we have only considered *propositional* logic programming. In practice, first-order logic programming is commonly used. In first-order logic programming, atomic variables are replaced by relations. For example we might have the following logic program.

$$\begin{aligned} P(x, y) \leftarrow \\ Q(x, y) \leftarrow \\ R(x, z) \leftarrow P(x, y) \wedge Q(y, z) \end{aligned}$$

Rules such as $P(x, y) \rightarrow$ are considered shorthand for $\forall x \forall y P(x, y) \rightarrow$. In other words, we omit the universal quantifiers. This means that for any assignment of the variables x, y, z, the above will hold. We generally restrict the semantics by only considering models whose domain is the *Herbrand universe* of the

language. The Herbrand universe is defined as the closure under application of all function symbols in the language of the set containing all constant symbols of the language. We understand a logic program as being shorthand for the (possibly infinite) logic program in which each sentence is grounded by each possible variable assignment.

Commonly, we restrict our language by removing function symbols. When this is done, assuming a finite number of constant symbols, the Herbrand universe is finite and thus so is our grounded logic program. Under these conditions, first-order logic programs are merely a shorthand for propositional logic programs since every relation with a specific grounding can be thought of as its own atomic propositional variable. Although the field of logic programming is extensive, for our purposes (namely rule extraction from neural networks), it is sufficient to consider propositional logic programs with stable model semantics.

Along with a semantic interpretation, a straightforward set of inference rules can be given to logic programming. In practice, computational rules on queries replace deductive inference and thus remove the need for well-defined deductive rules on logic programs. However, according to the definitions we use, a logical system must contain deductive rules. For this reason we merely note that acceptable logic programs can be equipped with the deductive rules of conjunction, simplification, and *modus ponens* along with an additional rule which deduces $\neg A$ from $\neg\phi_1 \wedge \neg\phi_2 \wedge ... \wedge \neg\phi_k$ where $\{\phi_1, ..., \phi_k\}$ is the set of the bodies of all rules with A as the head.

3.3 *M-of-N* Rules

In the previous section we discussed logic programs. Our rule extraction algorithms will not make use of full logic programming, but a specific kind of logic program that restricts the clauses to a form known as *M-of-N*. We introduce *M-of-N* rules here.

Propositional logic programs are made up of propositional rules of the following form, $A \leftarrow B \wedge C \wedge D$. The body of the clause, $B \wedge D \wedge C$ is a conjunction, that is, it is true if and only if each one of B C and D are true. If we want A to be true in other cases, for example if E and B are true, then we need to introduce an additional rule $A \leftarrow B \wedge E$. As we will see, for the

purposes of rule extraction, having a large number of rules to describe a single variable can result in a logic program that is uninterpretable. Luckily there exists an alternative which is better suited for rule extraction from neural networks. These are the *M-of-N* rules.

An *M-of-N* rule is a standard logic programming rule with the conjunctive clause in the body of the rule replaced by an *M-of-N* clause. *M-of-N* clauses generalize conjunctive clauses by relaxing the condition that the clause is true if and only if *all* of the literals in the body are true. Instead, an *M-of-N* clause only requires M of its N literals to be true (hence the name). For example 2-of-$\{B, C, D\}$ is true if and only if 2 or more of B,C, or D are true. We can express this in disjunctive normal form (DNF) by $(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$. It is easy to see that by writing the *M-of-N* clause in DNF an *M-of-N* rule is logically equivalent to a standard logic program by replacing the single *M-of-N* rule with multiple conjunctive rules. This raises the question, what is the advantage of using *M-of-N* rules over simple conjunctive ones? The answer lies in compactness. Using the previous example, if we wanted to express the rule $A \leftarrow$ 2-of-$\{B, C, D\}$ using only conjunctive clauses we would need three separate rules, $A \leftarrow B, C$, $A \leftarrow B, D$, and $A \leftarrow C, D$. That's three rules each containing three literals as opposed to one rule with 4 literals and a single integer. *M-of-N* rules are a more compact representation of the corresponding conjunctive rules. In the example given here the difference is small, but examples in following sections will highlight the significance of this. In circuit theory, an analog to *M-of-N* rules known as linear threshold circuits have been studied in detail where many results comparing their compactness to traditional circuits have been made Håstad and Goldmann [1991], Kautz [1961]. Compactness has the benefit of requiring fewer parameters to specify a set of rules and, as a result, can correspond to a smaller physical implementation or less space required in memory.

Although we have discussed compactness, we have not discussed interpretability. This is a much more difficult problem and many people have pointed out that the interpretability of *M-of-N* rules can be poor, even though the representation is compact. A 37-of-106 rule is unlikely to be easily understood by a human. For this reason, when using *M-of-N* rules for rule extraction we will include explicit measures of complexity in order to better understand the relationship between the complexity and accuracy of rules extracted from

neural networks. Our measure of complexity will be based on the length of the DNF of an *M-of-N* rule. This gives a good relative measure of complexity against other *M-of-N* rules but it should be noted that an *M-of-N* rule may be considered simpler than the equivalent set of rules expressed in DNF due to the existence of a shorter description. An *M-of-N* rule will usually be denoted by M-of-$\{X_1, ..., X_N\}$ but occasionally it will be convenient to refer to an *M-of-N* rule by $\binom{X_1,...,X_N}{M}$. This notation is more compact and also highlights the fact that there are M choose N conjunctive clauses which will satisfy an *M-of-N* rule

In addition to being more compact, *M-of-N* rules are an obvious choice for rule extraction due to their similarity to neural networks. In fact, an *M-of-N* rule can simply be thought of as a 'weightless perceptron.' Both can be thought of as types of threshold circuits in which the output is calculated based on a weighted sum of the input. In an *M-of-N* rule each weight for the input is 1. As with perceptrons, *M-of-N* rules can be extended to hierarchical *M-of-N* rules by using the atoms at the heads of one set of *M-of-N* rules to be the inputs to the next set of *M-of-N* rules. Although feedback can exist in a set of *M-of-N* rules, we will only be concerned with hierarchical *M-of-N* rules. Note that not every propositional function can be represented by a single *M-of-N* rule. For example, the XOR function, whose DNF is $(X_1 \wedge \neg X_2) \vee (\neg X_1 \wedge \neg X_2)$ cannot be expressed as an *M-of-N* rule. However, all propositional functions can be expressed by hierarchical *M-of-N* rules. This can be seen by first observing that any conjunctive clause $X_1 \wedge X_2, ..., \wedge X_N$ can be expressed by $\binom{X_1,X_2,...,X_N}{N}$, in particular any literal X_i can be expressed as $\binom{X_i}{1}$. We also have that any disjunction, $X_1 \vee X_2 \vee ... \vee X_N$ can be expressed with $\binom{X_1,X_2,...,X_N}{1}$. Then for any propositional sentence, by writing it in DNF, $\alpha_1 \vee \alpha_2 \vee, ..., \vee \alpha_N$ (here each α_i is a conjunctive term), we can see that this is expressed by the *M-of-N* rule $\binom{\alpha_1,\alpha_2,...,\alpha_N}{1}$ where each α_i is represented by a dummy variable at the head of an *M-of-N* rule representing the corresponding conjunction. Therefore, hierarchical *M-of-N* rules simply amount to a compact representation of propositional logic. Finally it should be noted that although *M-of-N* rules are usually defined only for $0 < M \leq N$, rules with $M = 0$ can be considered another way of writing $\top$, the clause that is always true, and rules with $M > N$ can be considered another way of writing $\bot$, the clause that is always false. These can be intuitively justified by noting that there are always at least 0 clauses in an *M-of-N* rule that are true and there are never more than N clauses in

an *M-of-N* rule that are true.

3.4 Decision Trees and Decision Sets

Although a large portion of deterministic symbolic systems can be boiled down to rule sets, when it comes to interpretability, the way in which a set of rules is represented can be important. For this reason, many popular explainability methods choose to use either *decision trees* or *decision sets*. Both can be written as a set of propositional formulas, but their specific form is generally thought to be more easily understandable than a simple set of propositional rules.

A decision tree is a graphical tree in which each node represents a propositional statement. The idea is, given an input example, we begin at the root and traverse the tree until reaching one of its leaves, which will be the classification. At each node, we decide which branch to follow based on the result of the propositional formula applied to our input example.

Every decision tree can be expressed as a set of propositional sentences in which each path from the root to a leaf is represented as a propositional rule whose head is the classification decision and whose body is the conjunction of the decisions taken at each node along the path. Thus we can easily convert a decision tree into a set of propositional rules. As previously mentioned, however, it is generally easier to comprehend these rules when presented in a decision tree format, despite the meaning being unchanged. One popular way of building decision trees is based on *information gain*, Quinlan [1986]. Information gain is the change in average entropy when a set is split in two. Given a probability distribution, P, over a finite set X, the entropy is a measure of the uncertainty of P defined by $H(P) = -\sum_{x \in X} P(x) \log(P(x))$. Given a set of input examples, each with a corresponding label, we can form a probability distribution of the labels using their frequency. This probability distribution will have an associated entropy. Splitting this set in two will result in a probability distribution over the resulting subsets, each with an associated entropy. We can calculate the average of the two entropies by weighting by the size of the respective sets normalized by the size of the original set. In

other words if X is split into X_1 and X_2 then the average entropy of the split is $\frac{|X_1|}{|X|}H(X_1) + \frac{|X_2|}{|X|}H(X_2)$. The *information gain* of this split is simply the difference between the original entropy and the new average entropy. We can use information gain to build decision trees iteratively. Given a set of input examples, choose a split which maximizes the information gain. This split becomes the root node. For each subsequent node repeat the same procedure for the set of examples which reach that node given the queries from higher nodes. This can be repeated until the only elements left in the set all have the same label or until some entropy threshold is reached. At this point a leaf node is added which classifies the example based on the label of the majority of the input examples that reach this leaf.

Similar to decision trees, a decision set is an ordered rule list in which the truth of one rule determines whether or not we check for the truth of the next rule. Each rule in a rule list makes a class prediction for an input. If the body of the rule is satisfied, then the head of the rule predicts the class label. However, if the body of a rule is not satisfied, then we move on to the next rule. A rule list will end with a 'default' prediction. If the input satisfies none of the rule-bodies, then we simply make the default prediction. Like decision trees, rule lists can be converted to propositional formulas in DNF. For each class label, consider each rule in the list with the class label at its head. We will check this rule only if every previous rule is not satisfied. Thus we will predict the class label with this rule if and only if the conjunction of the rule body with the negation of each previous rule body is satisfied. This converts a rule in a rule list into an equivalent propositional formula. The disjunction of each of these formulas will be equivalent to the class label. We can see that the equivalent DNF is much more complicated than the rule list itself which is why rule lists are considered to be a more easily understandable representation of propositional logic.

3.5 Penalty Logic

Although propositional logic is considerably useful for abstract reasoning, it is too rigid to model the nuances of everyday reasoning. This is because during our day to day life, we usually only have imperfect information about the world around us. This means that we may make observations which are inconsistent with our previously held beliefs. If we are using propositional logic, we will

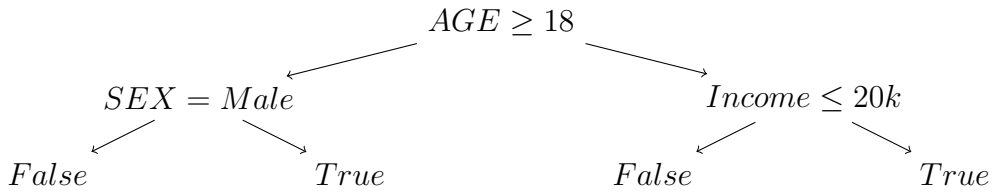


Figure 3.1: An example decision tree. Given an input, start at the root and follow the path right if the input satisfies a node and left if it does not until you reach a leaf

be unable to make any logical deductions once we observe some piece of information that conflicts with our previously held beliefs. By contrast, neural networks are much more tolerant of ambiguity. If we consider each neuron as representing some amount of evidence for a fact, then an output neuron may receive evidence both in support of and against it. Only by integrating all of this evidence it comes to a 'conclusion' which we can consider a degree of belief in a fact given various evidence for or against it. In an attempt to soften the restrictiveness of propositional logic and, as a result, create a logical system more like neural networks, *penalty logic* was developed Pinkas [1995]. Penalty logic is a an important logical system in neural-symbolic computing because of its close relationship to SCNs. It also serves as the inspiration for rule extraction techniques using confidence values and so we will cover it in detail here.

Penalty logic generalizes propositional logic by weighting propositional sentences by a positive integer called a *penalty*. The penalty is meant to represent our degree of belief in the sentence. In other words it is the 'consistency penalty' we pay if the sentence is not true in a truth assignment. Given a set of sentences with associated penalties, we can define a penalty associated to each propositional truth assignment by computing the sum of the penalties of each sentence that is violated. For instance, given the following set of sentences

$$\begin{aligned} 10 &: A \wedge B \\ 5 &: \neg A \vee \neg B \\ 2 &: B \end{aligned}$$

consider the truth assignment $\{A\}$. In propositional logic, this is a model of the second sentence, but not the first or third, giving it a penalty of 12. Notice

that the above set of sentences has *no* propositional models. A traditional propositional model is one in which every sentence is satisfied and hence has penalty 0. However, in penalty logic we do not wish to be that restrictive. Instead of requiring a 0 penalty, we consider the set of truth assignments with *minimum* penalty. The models of a knowledge base in penalty logic are the truth assignments with minimum associated penalty. Penalty logic also has a sound and complete deductive system to go along with the minimum-penalty semantics Pinkas [1995].

Penalty logic is a complete logical system similar to propositional logic in that it shares the same language (excluding the penalty values) and has a sound and complete deductive system. However, propositional logic is *monotonic* whereas penalty logic is not. A monotonic logic is one in which additional assumptions do not negate any previously derived conclusions. So if $L_1 \vdash l$, then $L_1 \cup L_2 \vdash l$. This does not hold in non-monotonic logics. To see that penalty logic is non-monotonic, consider the following example, $L_1 = \{1 : P, 2 : P \rightarrow Q\}$ and $L_2 = \{3 : \neg P\}$. The single model for L_1 is $\{P, Q\}$ so $L_1 \vdash Q$. On the other hand, the models for $L_1 \cup L_2$ are $\{P, Q\}, \{Q\}$, and $\{\}$ so $L_1 \cup L_2 \nvdash Q$. Logic programs with negation as failure are another example of a non-monotonic logic.

Considering that penalty logic was developed with the explicit purpose of developing a logical system more like SCNs, it comes as no surprise that there is a strong equivalence between penalty logic and SCNs. This makes penalty logic an important example for neural-symbolic integration and it will be referenced both when we discuss semantic equivalence and when we develop new rule extraction algorithms. Penalty logic is also useful because it generalizes propositional logic. A knowledge base in penalty logic in which the weights on all statements are uniform is identical to the corresponding knowledge base in propositional consisting of the same statements but with all weights removed. The deductive apparatus and model theory in this case is identical. We can therefore regard propositional logic as a specific subset of penalty logic meaning anything that can be proved for all knowledge bases in penalty logic can also be proved for all knowledge bases in propositional logic. This is a fact we will use when discussing neural-symbolic integration.

Although we have discussed a number of different network architectures and

logical systems, there are countless others that we have omitted. There are many different modifications and expansions of all of the systems that have been discussed so far. The models presented here are those that are relevant to our discussion on neural-symbolic integration as well as playing a role in our own experiments. Now that the basic systems we will be studying have been described we will begin to answer the question of how they can be integrated.

Chapter 4

A Framework for Neural-Symbolic Integration

4.1 Neural Networks vs Symbolic Systems

Neural networks have become popular due to their flexibility and general-purpose learning algorithms. Both of these features have traditionally been difficult for symbolic systems. Conversely, symbolic systems use comprehensible abstract reasoning which can be verified by a human. Furthermore, the use of abstract features in symbolic systems ensures that it reasons about situations in sufficient generality. If a symbolic system encodes the concept 'dogs like to play fetch' then we do not have to worry that it will only apply this reasoning to some dogs or dogs that look a certain way. Neural Networks, on the other hand, do not have this guarantee. Although it may predict that dogs play fetch in all cases in a test set, we cannot be sure that the network actually encodes the concept of 'dogs', 'fetch', or the relationship between them. Indeed, the development of adversarial attacks seems to indicate that neural networks might not be learning the concepts necessary for truly general solutions akin to those a human uses. The complementary strengths and weaknesses of symbolic systems and neural networks was the initial inspiration for neural-symbolic integration. However, if we are to fully reach the potential of neural-symbolic integration, the exact formal differences between symbolic systems and neural networks must be made explicit.

Much has been made of the differences in how symbolic and connectionist systems view and approach the AI problem, but these differences are more often than not left ambiguous. This might lead one to assume that neural net-

works and symbolic systems are fundamentally different in nature. However, the underlying relationship between neural networks and symbolic systems is a more subtle one. One issue here is the lack of clarity in what constitutes a 'symbolic system'. Intuitively, symbolic systems operate at the 'symbolic level.' Reasoning at the symbolic level is reasoning with abstract entities. To illustrate, consider the famous example, 'all men are mortal', 'Socrates is a man', therefore 'Socrates is mortal.' In this example we use the abstract concepts of being a man and mortality. The reasoning process uses the logical rule *modus ponens*. The key feature here is that the abstract entities can be meaningfully identified by a person. In a symbolic system we are attempting to model the same kind of reasoning that humans do in their everyday life. We may see that it is raining out and decide to get an umbrella. This kind of deduction is done in terms of abstract concepts. Umbrellas can come in all shapes, sizes, and colours but the exact nature of a particular umbrella is not important in the process of our reasoning. Neural networks, on the other hand, operate at the 'sub-symbolic' level. Individual neurons do not necessarily represent any easily identifiable concept. Instead they may model weak statistical regularities in a dataset with a network owing its predictive capabilities to raw computational power rather than abstract reasoning. For example, a single layer neural network analyzing image data may use features such as the colour and brightness of pixels. Because there might be many pixels, the colour of a single pixel is not considered an abstract feature when trying to determine what objects are contained in the image. This distinction is intuitive but generally not formalized in a meaningful way. The colour value of a pixel can be represented in logic easily and the same reasoning process applied to abstract concepts can likewise applies to the colour values.

When looking at symbolic systems and neural networks formally, the distinction between abstract and non-abstract concepts is lost. There is nothing inherent to symbolic systems that requires the atomic variables to represent meaningful concepts. Furthermore, the notion of what is and what isn't a meaningful and transferable concept is difficult to define. The issue is further complicated by the fact that there exist many different types of neural networks and symbolic systems which can often have vastly different properties. We have already seen that symbolic systems can be adapted to reason about things in a softer and more human-like manner with penalty logic, something thought to be more characteristic of neural networks. It is also clear that

a simple perceptron can implement various logic gates in a way identical to propositional logic. Given the rich variety of neural networks and symbolic systems, along with the implicit Turing-equivalence [1], it seems likely that the difference between neural networks and symbolic systems is more of an issue of *representation* rather than something *fundamental*. In this chapter, we develop a framework for neural-symbolic integration that clarifies the relationship between neural networks and logical system by defining the ways in which neural networks and logical systems can be identified. We focus specifically on symbolic systems that can be thought of as logical systems because the majority of neural-symbolic integration has involved logical systems. By proving the equivalence between basic network architectures and various logical systems, along with collecting several important equivalences previously developed in the field of neural-symbolic integration, we show that the difference between neural networks and logical systems is primarily a representational one. The definitions also allow us to make a precise definition of fidelity that generalizes both the probabilistic and deterministic cases. Using these results we reduce both the practical and philosophical arguments against rule extraction to one of complexity and use this to test the claims empirically.

4.2 Neural and Symbolic Encodings

In this section, we will formally define the possible relationships between a neural network and a symbolic system. The mapping of a symbolic system into a neural network is known as a neural encoding whereas the reverse is a symbolic encoding. The presence of both encodings is an equivalence. Encodings will be defined both syntactically and semantically with the implications of each discussed in detail. Finally, we build on this by giving a precise definition of fidelity.

We begin with a definition of a symbolic system. A neural network can be easily understood as a dynamical system of the form presented in chapter 2. A symbolic system is more difficult to define. Because the focus of neural-symbolic integration has been on logical systems and automata, we will limit

[1] It is necessarily true that any symbolic system and any neural network implemented on a finite-precision machine can both be simulated by a Turing machine, in the case of arbitrary precision this is no longer true for some neural networks. We will discuss this issue in this chapter

our discussion of symbolic systems to these. Most neural-symbolic integration has related neural networks to logical systems with an emphasis on logic programming. One exception to this is the use of finite automata to describe recurrent neural networks. We will discuss the neural symbolic work done on the relationship between automata and neural networks, but the focus will be on logical systems. Informally, a logical system is a deductive system over a formal language with some notion of semantics. The semantics are a set of interpretations for the language that determine the truth of each sentence in the language. In order to formalize this, we follow the standard metalogical definition with some minor variation Hunter [2008]. For our purposes, we define a logical system as such.

Definition 4.2.1. *A logical system, $\mathcal{S}$, is a triple, $(\mathcal{L}, \vdash_{\mathcal{S}}, \mathcal{M})$ where $\mathcal{L}$ is a computable language, $\vdash_{\mathcal{S}}$ is a set of deductive rules (defined as a computable relation on $2^{\mathcal{L}}$) and $\mathcal{M}$ is a set of models. A model, M, is a set together with a truth function $f_M : \mathcal{L} \to \{0, 1\}$.*

A logical system has two components. The first is the deductive system, which represents the abstract reasoning in a language. In other words, given that some sentences are true, which other sentences are true? An inference from a set of sentences (hereafter referred to as a knowledge base), L, to some other set of sentences, L', is considered valid in a logical system if $(L, L') \in \vdash_{\mathcal{S}}$. The second component of a logical system is the semantics. The semantics provide the means for evaluating the truth of a sentence given a specific interpretation. Following the standard Tarskian definition of truth, we view the models of a logical system as interpretations of the language in which the sentences are either true of false. The idea is that each model is a 'possible world' in which the truth of a sentence can be evaluated. In practice the truth function of a model is usually built by assigning elements in M to atomic units in the language and recursively defining the value of f_M based on connectives in the language. The definition we give is too general in the sense that we can always define a system with trivial semantics in which every model interprets every sentence as true. However, every logical system with meaningful semantics will be of this form making a more precise definition needlessly technical for our purposes. We now recall the basic notions of logical entailment. Given a set of sentences $L \subset \mathcal{L}$, we say $M \in \mathcal{M}$ is a model of L if for all $l \in L$, $f_M(l) = 1$. Given a sentence l_0, we say $L \models_{\mathcal{S}} l_0$ if every model of L is also a model of l_0. We write $L \vdash_{\mathcal{S}} l_0$ if $(L, l_0) \in \vdash_{\mathcal{S}}$. For every $L, L' \in \mathcal{L}$ we say $L \vdash_{\mathcal{S}} L'$ if for

all $l' \in L'$, $L \vdash_{\mathcal{S}} l'$. A logical system is *sound* if for every $L \subset \mathcal{L}$, $L \vdash_{\mathcal{S}} l_0$ implies $L \vDash_{\mathcal{S}} l_0$. This is just to say that the deductive system respects the semantics. Conversely a logical system is *complete* if for every $L \subset \mathcal{L}$, $L \vDash_{\mathcal{S}} l_0$ implies $L \vdash_{\mathcal{S}} l_0$. This definition gives two possible avenues for neural symbolic integration, syntactic and semantic. Using semantics, neural networks are used to determine the models of a logical system. To do this, we start by choosing a set of neurons along with a neural network defined on these neurons and an injective mapping from states of the neurons into the models of $\mathcal{S}$.

Definition 4.2.2. *Given a Neural Network, N, with a state space, X, and an injective map $i : X \to \mathcal{M}$ where $\mathcal{M}$ is the set of models for a logical system, $\mathcal{S} = (\mathcal{L}, \vdash_{\mathcal{S}}, \mathcal{M})$, we say $x \in X$ is a model for $L \subset \mathcal{L}$ if $i(x)$ is a model of for L.*

This gives us the following definition of a neural model

Definition 4.2.3. *A network N, with an initial state x_0 is a model for L if there exists $t_0 > 0$ such that for all $t > t_0$, $x = N^t(x_0)$ is a model for L. A network is called a model for L if it is a model for L given any initial state.*

In other words, starting at initial state x_0, there is some time after which every subsequent state of the network is a model for L. It is possible that there are states of the network that are models for a knowledge base, but appear only temporarily in the dynamics of the network. When using neural networks to define the semantics of a logical system, we only want to count those models that appear infinitely often. For this reason we give the following definition.

Definition 4.2.4. *Given a knowledge base L, a state, x, of a neural network N is an L-model of N if x is a model of L and there exists an initial state x_0 such that for all $t > 0$, $\exists t' > t$ such that $N^{t'}(x_0) = x$.*

The L-models of N are those states which model L and appear infinitely many times for at least one initial state. By looking at the set of L-models of a network, we can define a notion of neural entailment by restricting the semantics of a logical system.

Definition 4.2.5. *If N is a neural-model of L we say that $L \vDash_N l_0$ if l_0 is true in every L-model of N.*

The inference process is the same except that only L-models are considered when making a judgement. Importantly, if the set of L-models of a network

is equal to the set of models of a knowledge base, then the semantics are identical. Semantic encodings have most commonly been applied to stable neural networks. When the network is stable, it is much easier to define a neural model. This is because the only states of stable networks that appear infinitely often are the stable states meaning that they are the only states that need to be considered. This is summarized with the following proposition

Proposition 4.2.1. *Given a stable network N, if x is an L-model then x is a stable state. Furthermore N is a neural model of L if and only if each of its stable states are models of L.*

Proof. If x is not a stable state, then for all x_0, because N is stable, there exists t_0 such that $N^{t_0}(x_0)$ is stable so for all $t > t_0$, $x \neq N^t(x_0)$ and thus x is not an L-model. If every stable state of N is a model of L then because N is stable it is a model for all x_0 and thus by definition a model for L. Furthermore, if N is a model for L then if there exists a stable state, x', which was not a model for L, then N with initial state x' would not be a model for L and again by definition N would not be a model for L □

The use of stable networks and their corresponding stable states to model logical systems has sometimes been referred to as *stable-state semantics* but here it can be seen as a specific case of a more general neural semantics. Stable-state semantics will be our focus when we investigate various neural and symbolic encodings in detail in future sections.

One obvious limitation to the semantic approach is that the number of models a network can represent is limited. In first-order logic the Lowenheim-Skolem theorems state that every knowledge base with an infinite model has models of arbitrarily large cardinality Marker [2002]. This means that the state space of even an uncountable number of neurons with continuous activations will not be large enough to map onto every model of some knowledge bases because the collection of all cardinalities is not a set. Although the number of elementary equivalence classes of models of a countable first-order language is at most uncountable, making a semantic neural encoding technically possible, the practical difficulties in defining such an encoding make the applicability of it dubious. Furthermore, when we move to multi-valued logical systems there will be systems with a large enough set of inequivalent models that a semantic neural encoding will be genuinely impossible. An early criticism of neural networks was a perceived 'propositional fixation' which saw the tendency to

model neurons as propositional atoms as a fundamental shortcoming of neural networks that must be overcome in order for neural networks to be a truly general purpose tool in AI McCarthy [1988]. We observe that the general focus of neural-symbolic integration on semantic encodings may have contributed to this perception. In order for neural-networks to completely overcome the propositional fixation they cannot be related to logical systems via semantics, but must represent logical systems with *syntactic encodings*

In a syntactic encoding the states of a neural network are mapped to the sentences in the languages themselves rather than models. The dynamics of a network are required to respect the deduction rules. In order to formalize this, we first develop a notion of entailment for neural networks whose states represent knowledge bases. Given a network, N, with an injective map from its state space to the set of knowledge bases in a logical system, $\mathcal{S}$. If a state x_0 is mapped to the knowledge base L_0, we say $L_0 \vdash_N l$ if there exists $t_0 > 0$ such that $N^{t_0}(x_0) = x$ and x maps to a knowledge base L such that $l \in L$. All this is doing is translating the dynamics of the network N into a relation on knowledge bases in $\mathcal{S}$. The definition of a syntactic model is merely a neural network whose dynamics do not violate the deductive framework of a logical system. In other words

Definition 4.2.6. *A syntactic model of a logical system, $\mathcal{S}$, is a neural network, N, with an injective mapping from its state space into $2^{\mathcal{L}}$ such that if $L \vdash_N l_0$, then $L \vdash_{\mathcal{S}} l_0$.*

In a syntactic encoding the neural network at least partially models the deductive process. Each state of the network represents a knowledge base and the transition from one state to another represents a deduction from one knowledge base to another. Most neural-symbolic integration has been semantic in nature; however, A general purpose syntactic encoding for neural networks has been developed Smolensky and Legendre [2006]. This encoding decomposes sentences in a formal language into roles and fillers. A role is the position of a symbol in the language and a filler is the symbol filling that role. The rolls and fillers are both represented as linearly independent vectors in a vector space and the tensor product of the role and filler vector is taken to represent the filler/role association. The encoding is extended to full sentences by summation of each filler/role vector contained within the sentence. Because neural networks are Turing complete Siegelmann and Sontag [1995], this representation can, in principle, simulate any deductive rule in a logical system. Using a

distributed representation for logical systems may have other advantages, too, which we will discuss when we cover the representational differences between neural networks and logical systems.

In our definitions of neural models we require the mapping from states of the network to models or knowledge bases to be injective. This is to exclude trivial mappings in which we simply map every element in the state space to a single model or knowledge base. This is especially important for syntactic encodings. Suppose we mapped every state to the same knowledge base, then because the deduction $L \vdash_{\mathcal{S}} L$ is always true (we don't put this requirement in our definition explicitly but it is valid in every logical system used) we would have a syntactic model of $\mathcal{S}$. This trivial mapping would imply that any neural network can be symbolically encoded into any logical system with an encoding that gave us no information about the dynamics of the network. The objective here is to give a complete description of the neural network as a deductive apparatus. However, this comes with one small caveat. In many cases, the dynamics of certain neurons are not considered relevant outside of some intermediate calculation steps. The dynamical properties of the network in which we are interested pertain to a certain subset of the nodes. Returning to our umbrella analogy, the function of an umbrella is defined by the state transition between closed and open. A complete logical description of an umbrella from a functional perspective may be that, when it rains, someone will change the state of an umbrella from closed to open only to be closed again when the rain lets up. The exact mechanisms facilitating this change in state, while important to those who design umbrellas, are not relevant to the abstract functionality of an umbrella. For this reason we do not always need a complete description of the neural state when deciding which model or language base it encodes. Although we have defined our maps into logical systems on the *entire* state space, in many cases we are only interested in the dynamics on some equivalence class of state spaces. For example, many neural networks have hidden units which are used only to boost a network's computational power and whose internal state does not necessarily represent anything meaningful to a human observer. Another instance is the state of hidden layers in a feed-forward network before the network has propagated up an input. In our definition, the initial state of a neural network must define a starting activation for *every* neuron in the network; however, in the case of feed-forward networks, there is no need to assign the hidden layers any

particular value until it has been calculated from the input. Although this might seem to be a minor detail, it turns out to be a problematic technicality. For this reason we define the mappings to logical systems only on an equivalence class of states in which the equivalence classes contains all states of the network that are identical up to variation in the subset of states which are deemed irrelevant. As an example, consider a three neuron network x_1, x_2, x_3 in which we have decided that the neuron x_3 is irrelevant to our encoding. In this case the states $(0, 1, 0)$ and $(0, 1, 1)$ map to the same model or knowledge base. Which states are defined as equivalent depends highly on the networks and logical system under investigation. A general guideline is that as long as two states of a neural network are thought to have a meaningful representational difference then they should not be identified. With that cleared up, we move on to notions of equivalence between logical systems and neural networks.

The relationship between neural networks and logical systems in bi-directional. From a semantic point of view, We can map a set of neural networks into a set of knowledge bases in a logical system (known as rule extraction) as well as map a set of knowledge bases into a set of neural networks. In practice there is an acceptable amount of error when mapping neural networks into logical systems. We don't expect the extracted symbolic sentences to always agree with the network. For this reason we reserve the terms neural encoding and symbolic encoding for the exact identification of logical systems and neural networks and vice-versa. Starting with the semantic approach, a neural encoding is a choice of neural model for a knowledge base which captures its semantics while, conversely, a symbolic encoding is a choice of knowledge base for a neural network such that the network captures the semantics of the knowledge base. This gives us the following definitions.

Definition 4.2.7. *Given a knowledge base, L_0, in a logical system, $\mathcal{S}$, a neural encoding of L_0 is a neural model of L_0, N, such that $L_0 \vDash_N L \Rightarrow L_0 \vDash_{\mathcal{S}} L$. Given a network, N, a symbolic encoding is a choice of knowledge base L such that N is a neural encoding of L.*

Note that because a neural model is a restriction of the models of $\mathcal{S}$, we always have $L_0 \vDash_{\mathcal{S}} L \Rightarrow L_0 \vDash_N L$ meaning that given an neural encoding of L, $L_0 \vDash_N L \iff L_0 \vDash_{\mathcal{S}} L$. Moving on to syntactic encodings, we want a syntactic neural encoding to model the entire deductive process of a logical system. In other words, given any knowledge base and any deduction in the logical

system, there should be some syntactic neural model that has a corresponding initial state and sequence of states that arrives at the same conclusion as the deduction. For a logical system to be a (syntactic) symbolic encoding of a (syntactic) neural model we require that the neural model is a neural encoding for each knowledge base it maps to. In other words,

Definition 4.2.8. *Given a logical system, $\mathcal{S}$, with a syntactic model N, we say that N is a (syntactic) neural encoding of a knowledge base L if there exists a state of the network x that maps to L and $L \vdash_{\mathcal{S}} l \Rightarrow L \vdash_N l$. N is a neural encoding of a set of knowledge bases if it is a neural encoding of each knowledge base in the set. We say that a set of knowledge bases of $\mathcal{S}$ is a (syntactic) symbolic encoding of N if N is a (syntactic) neural encoding of each knowledge base mapped to by a state in the state space of N.*

With both the syntactic and semantic methods of encoding defined we can formally state what it means for a set of neural networks to be equivalent to a logical system.

Definition 4.2.9. *A set of neural networks is (semantically/syntactically) equivalent to $\mathcal{S}$ if every knowledge base in $\mathcal{S}$ can be (semantically/syntactically) neurally encoded into a network in the set of neural networks and every neural network in this set can be (semantically/syntactically) symbolically encoded into $\mathcal{S}$.*

In future work, it may be useful to require a stricter definition of equivalence in which the correspondence between knowledge bases and neural networks is one-to-one. This would allow one to think of neural and symbolic encodings as invertible mappings between neural networks and knowledge bases (or collections of knowledge bases) in a logical system. For the time being we merely require that we can always represent one in the other somehow. In both semantic and syntactic equivalences, the maps are defined in terms of knowledge bases. In a syntactic equivalence, a knowledge base has a neural encoding if we can directly represent the knowledge base as a state of a neural network in which the dynamics represent every possible deduction from that knowledge base. Semantically, a neural encoding represents the models of a knowledge base as states of the network and the dynamics merely drive a convergence to these states. Semantic symbolic encodings are simply knowledge bases that exactly capture the semantic implications of a neural model. For syntactic symbolic encodings, we do not map to a single knowledge base but an entire

set of knowledge bases thus fully describing the states of a network. From a syntactic point of view, a symbolic encoding is a complete description of the network and its dynamics whereas from a semantic point of view a symbolic encoding is simply a knowledge base whose models are completely described by the neural network.

in most cases a precise neural or symbolic encoding is not feasible. This is especially true when translating neural networks into logical systems. When this is the case, rule extraction techniques will be used. Like a symbolic encoding, a rule extraction technique associates a knowledge base in a logical system to a state of a neural network. Usually this knowledge base will contain sentences that describe the whole network along with additional information describing the state. The difference between rule extraction and symbolic encoding is that a rule extraction technique does not require a network to be a model (either semantic or syntactic) of the associated knowledge base. When this is the case, it is important to have some kind of measure for how closely the neural network models the knowledge base. The most obvious measure is the percentage of initial states of a network which are models of the knowledge base. This measure is known as the *fidelity* of a rule extraction technique and it is given the following formal definition

Definition 4.2.10. *The fidelity of a knowledge base, L, given a neural network, N, is the quantity*

$$\frac{1}{Vol(I)} \int_I \chi_{N,L}(\hat{x}) d\hat{x}$$

where I is the set of states of the neural network and $\chi_{N,L}$ is an indicator function returning 1 if N with initial state $\hat{x}$ is a model of the knowledge base L and 0 otherwise. The integral is the standard Lebesgue integral (which in the finite case will be a summation) and $Vol(I) = \int_I d\hat{x}$. A syntactic definition of fidelity is identical with the indicator function returning 1 if $\hat{x}$ is a *syntactic* neural model of its knowledge base and 0 otherwise. Although technically fidelity can be measured as long as one gives a logical system and a neural model, syntactic or otherwise, in virtually all cases fidelity will be used when trying to find approximate symbolic encodings of neural networks (in other words, rule extraction). We will revisit this later but to summarise, we are not generally interested in approximate neural encodings. A neural encoding is only considered significant if it can exactly model a knowledge base in a

logical system, whereas a complete symbolic encoding of a neural network generated by rule extraction will generally be highly complicated and thus have limited to no practical use. For this reason, approximate solutions are often preferred making the fidelity an important metric for rule extraction.

Next, we will examine the relationship between propositional logic and stable neural networks with either binary, discrete, or continuous activations.

4.3 Stable Binary Neural Networks and Propositional Logic

The first neural-symbolic relationship we will discuss is that of propositional logic and binary neural networks.

Theorem 4.3.1. *Propositional logic is semantically equivalent to stable neural networks whose neurons take binary activation values, hereafter referred to as binary neural networks.*

Proof. Given a binary neural network with a finite number of neurons, we associate a propositional variable X to each neuron x. We can then associate states of the neural network with a propositional model by including a propositional variable in the model if the corresponding neuron is activated. In other words, with each state of the network we associate the model $\{X : x = 1\}$. With a finite number of variables, we can translate models into sentences by associating a model with a conjunction that includes all variables in the model and the negation of all variables absent from the model. For example, given the propositional variables X_1, X_2, X_3 we map the model $\{X_1\}$ to the sentence $X_1 \wedge \neg X_2 \wedge \neg X_3$. With this in mind, we show that every stable neural network has a symbolic encoding. We do this by taking the set of stable configurations of the network and translating them into propositional sentences. For example, the configuration $x_1 = 1, x_2 = 1, ..., x_k = 0, ..., x_n = 0$ is translated to the sentence $X_1 \wedge X_2 \wedge ... \wedge \neg X_k \wedge ... \wedge \neg X_n$. The disjunction of each of these sentences will be a symbolic encoding of the network. To see that every knowledge base has a neural encoding we refer to the proof that every knowledge base in penalty logic has a neural encoding (See next chapter). Since propositional logic can be extracted from penalty logic by setting each penalty to 1, and the encoding method between neurons and propositional models is the same, it follows that each propositional knowledge base has a neural encoding □

The symbolic encoding of stable binary neural networks is somewhat trivial for the case of propositional logic. This is because models for propositional logic can be exactly represented by a propositional sentence. This reduces the problem to simply writing the sentence corresponding to the stable states of a network. Not all models in all logical systems have this property. First order logic proves more difficult as there are models which cannot be unambiguously described by a knowledge base (famously, standard arithmetic cannot be described by a recursively enumerable knowledge base — you will always end up with non-standard models). Moving on, we prove that all feed-forward binary neural networks can be syntactically symbolically encoded into propositional logic.

Theorem 4.3.2. *Binary feed forward networks can be syntactically symbolically encoded in propositional logic.*

Proof. First we develop an encoding method. Again we start with a neural network whose neurons each correspond to an atomic variable in propositional logic. This time, we directly identify a configuration of neurons with a propositional conjunction. We translate a configuration of n binary neurons, $x_{i_1} = 1, x_{i_2} = 1, ..., x_{i_k} = 0, ..., x_{i_n} = 0$, into the propositional sentence $X_{i_1} \wedge X_{i_2} \wedge ... \wedge \neg X_{i_k} \wedge ... \wedge \neg X_{i_n}$. For a configuration $\hat{x}$ call this $c(\hat{x})$. Define the knowledge base $L_{\hat{x}}$ as the knowledge base including $c(\hat{x})$ and every sentence which can be deduced from $c(\hat{x})$. In other words $L_{\hat{x}}$ is the closure under implication of $c(\hat{x})$. Now take a binary neural network N and start with an empty knowledge base L. For each neuron, say h, with inputs $x_1, ..., x_n$, enumerate all input/output relationships and start with an empty knowledge base L. If $\hat{x}$ is a configuration of the input neurons which results in an output of 1, add the sentence $H \leftarrow c(\hat{x})$ to the knowledge base along with all its implications, namely $H \vee \neg c(\hat{x})$. If $\hat{x}$ is a configuration of the input neurons that results in an output of 0, add the sentence $\neg H \leftarrow c(\hat{x})$ and its consequences. If h has no input neurons then after time $t = 1$, h is either always on or always off. If it is always on add H to the knowledge base and if it is always off add $\neg H$. We will write propositional statements consisting of a single atomic variable, X, as $X \leftarrow$ to distinguish the statement from the variable itself. With this notation we add either $H \leftarrow$ or $\neg H \leftarrow$ if H has no inputs. Now we repeat this for every neuron. The result of this is that every input/output relationship between neurons in the network is represented with a propositional statement in L. Next we must develop the notion of a *supported* neuron. We define this

recursively. If there is a rule $X \leftarrow$ in L then if $x = 1$, x is supported. If there is a rule $\neg X \leftarrow$ then if $x = 0$, x is supported. Now for any x_i, if there is a rule in L such that the head of the rule matches the value of x_i (ie a rule with X_i at the head if $x_i = 1$ and $\neg X_i$ at the head if $x_i = 0$) whose body contains only supported neurons then x_i is supported. For example, if we have a configuration $x_1 = 0, x_2 = 1, x_3 = 1$ with rules $\neg X_1 \leftarrow X_3$, $X_2 \leftarrow X_1 \wedge X_3$, and $X_3 \leftarrow$, then x_1 and x_3 are supported and x_2 is not. Given any configuration of neurons, let $\hat{x}$ be the configuration of the supported neurons, we define a set of equivalence classes of configurations with the relation $x \sim x'$ iff the configuration of the supported neurons in x is identical to the configuration of the supported neurons in x'. Define $L_{\hat{x}}$ on this equivalence class of states to be the conjunction of the literals corresponding to the configuration of the supported neurons. Now we map each state, x, (up to the previously defined equivalence class) of the neural network to the knowledge base $\hat{L}_{\hat{x}}$ where $\hat{L}_{\hat{x}}$ consists of $L \cup L_{\hat{x}}$ and all consequences of the propositions that do not involve *modus ponens*. We know this is injective because $\hat{L}_{\hat{x}}$ is not closed under modus ponens meaning that $\hat{L}_{\hat{x}}$ only contains information about the current state and thus no inferences about neurons which are unsupported in $\hat{x}$ can be made. Therefore if $\hat{L}_{\hat{x_1}} = \hat{L}_{\hat{x_2}}$ then $\hat{x_1} = \hat{x_2}$ (up to a difference in state of unsupported neurons).

If a neuron is supported, its state does not change. This can easily be seen by induction. The input neurons are only supported when they reach their stable state. Then if a neuron has inputs which are supported, by our induction hypothesis the inputs neurons do not change and thus the output neuron does not either.

In order to prove that this is a symbolic encoding we must show that for all configurations of supported neurons $\hat{L}_{\hat{x}} \vdash_{\mathcal{S}} l$ iff $\hat{L}_{\hat{x}} \vdash_N l$.
'$\Rightarrow$'
Suppose $\hat{L}_{\hat{x}} \vdash_{\mathcal{S}} l$. This implies that there is a deductive sequence, $L_1, L_2, L_3, ..., L_k$ with $l \in L_k$ such that $\hat{L}_{\hat{x}} \vdash_{\mathcal{S}} L_1, L_1 \vdash_{\mathcal{S}} L_2, L_2 \vdash_{\mathcal{S}} L_3, ..., L_{k-1} \vdash_{\mathcal{S}} L_k$ where each deduction represents a set of inference rules in propositional logic. We will show that at each step, for all $l_i \in L_i$, $\hat{L}_{\hat{x}} \vdash_N l_i$. Recall that $\hat{L}_{\hat{x}} \vdash_N l$ implies that there exists $t > 0$ such that $l \in \hat{L}_{N^t(\hat{x})}$. We will use induction to prove that $\hat{L}_{N^i(\hat{x})} \supset L_i$ for all $1 \leq i \leq k$. Because the sequence starts with $\hat{L}_{\hat{x}}$ the base case is trivial, now consider some L_i and assume $\hat{L}_{N^{i-1}(\hat{x})} \supset L_{i-1}$. Take

any $l \in L_i$, by assumption, $L_{i-1} \vdash_{\mathcal{S}} l$ via a one step deduction. By construction of $\hat{L}_{N^{i-1}(\hat{x})}$, l is either an application of *modus ponens* using a rule in L of the form $Y \leftarrow X_1 \wedge ... \wedge \neg X_k \wedge ... \wedge \neg X_n$ whose body is satisfied by a sentence in $\hat{L}_{N^{i-1}(\hat{x})}$, or some other consequence of $\hat{L}_{N^{i-1}(\hat{x})}$. In the first case, the configuration of $x_1, x_2, ..., x_n$ will be unchanged in $N^i(\hat{x})$ because each neuron is supported. By construction of L, y will be supported in $N^i(\hat{x})$ and thus $l \in \hat{L}_{N^i(\hat{x})}$. In the second case, because $\hat{L}_{N^{i-1}(\hat{x})}$ is closed under all other deductions by construction, $l \in \hat{L}_{N^i(\hat{x})}$. By the previous argument, monotonicity of propositional logic, and the inductive hypothesis, we have that for all $l \in L_i$, $L_{i-1} \vdash_{\mathcal{S}} l \implies \hat{L}_{N^{i-1}(\hat{x})} \vdash_{\mathcal{S}} l \implies l \in \hat{L}_{N^i(\hat{x})}$. Which gives us $\hat{L}_{N^i(\hat{x})} \vdash_N L_i$.
'$\Leftarrow$'
Applying the previous induction but on time steps gives the same result. Given a state, $\hat{x}$ and a proposition, $l \in \hat{L}_{\hat{N(x)}}$, either l is a conjunction atoms representing supported neurons, or it is a consequence of the conjunction of supported neurons and L. In the latter case $\hat{L}_{\hat{N(x)}} \vdash_{\mathcal{S}} l$ by definition. In the former case, every atom, X_i, in the conjunction must correspond to a supported neuron in $\hat{N(x)}$. This means that either x_i is supported in $\hat{x}$ or that X_i is at the head of a rule in L with all the atoms in the body corresponding to neurons supported in $\hat{x}$. In both cases $\hat{L}_{\hat{N(x)}} \vdash_{\mathcal{S}} X_i$ or $\hat{L}_{\hat{N(x)}} \vdash_{\mathcal{S}} \neg X_i$ depending on whether or not X_i is negated in the head of the corresponding rule in L. This implies that $\hat{L}_{\hat{N(x)}} \vdash_{\mathcal{S}} l$ as l is the conjunction of each literal. Putting this into the induction, we have that if $\hat{L}_{\hat{x}} \vdash_N l$, then this can be expressed as $\hat{L}_{\hat{x}} \vdash_N \hat{L}_{\hat{N(x)}} \vdash_N ... \vdash_N \hat{L}_{\hat{N^k(x)}}$ and at each step we have proved that $\hat{L}_{\hat{N(i-1)(x)}} \vdash_{\mathcal{S}} \hat{L}_{\hat{N^i(x)}}$ and thus by induction we have that if $\hat{L}_{\hat{x}} \vdash_N l$ then $\hat{L}_{\hat{x}} \vdash_{\mathcal{S}} l$ □

The previous example shows that feed forward networks can be described exactly as deductions in a propositional system in which the knowledge bases are sets of acyclic implications and all of their consequences. Such a system is effectively a type of logic programming. An exact semantic equivalence with various kinds of logic programming will be described when we review several important neural encodings. If we restrict propositional logic to knowledge bases generated from a set of acyclic implications, we can also show that these knowledge bases have not only semantic but syntactic neural encodings by feed-forward networks; however, full unrestricted propositional logic may contain cyclic implications which makes their translation to feed-forward neural networks impossible. This might be surprising as feed-forward networks can be thought of as equivalent to propositional statements in that they can en-

code any binary function. The possibility of feedback means that feed-forward neural networks, when conceived as a stable dynamical system, can no longer represent the deductive process of propositional logic. For example, it is a property of propositional logic that for an inconsistent knowledge base, L, $L \vdash_{\mathcal{S}} l$ for all propositional sentences l. In order to encode any inconsistent knowledge base of propositional logic into a stable neural network the stable states much correspond to the entire language. This may be possible but it is probably not particularly enlightening and we will limit ourselves to the encoding of feed-forward neural networks into propositional logic.

The previous theorem, although correct in identifying the dynamics of feed-forward networks with applications of *modus ponens*, does not quite match how feed-forward networks are used in practice. As discussed in the section on feed-forward networks, in the definition of a feed-forward network we have given, input neurons are part of the network and subject to the same dynamics of the rest of the neurons. In practice this means that no matter what initial state you give the input neurons will immediately revert back to the single stable state determined by their biases. Although this is useful when making the identification with propositional logic programming, in practice the input neurons in feed-forward networks are often treated as a representation of the input to a function and hence they are not subject to the neural dynamics. In other words, we choose an input configuration and then propagate upwards to determine the output, keeping the inputs fixed at whatever value we gave for input. Luckily, a few simple modifications to our encoding reestablishes the relationship between feed-forward neural networks and propositional logic. First we give all input neurons 0 bias, a single self-connection of 1, and a linear function for the transfer function. With these additions, the input neurons remain in their initial state for all time. In order to now rectify our symbolic encoding, we simply define any configuration of input neurons to be supported and remove from L all instances of rules whose head is an atomic variable corresponding to an input neuron. These will exactly be the rules whose body is empty. It is not difficult to see that the same proof shows that this gives a symbolic encoding for these modified feed-forward networks. Although the self connections of the input technically disqualify these networks from being feed-forward they are still stable and this minor nit-pick does not contain any substance. In the following chapter, when we refer to feed-forward networks in the context of neural-encodings of logic programs we will be referring to those

in which the inputs are not fixed, but when we discuss rule extraction from deep networks in part 2 we will assume that these feed-forward networks have fixed inputs.

To summarize, stable binary networks are semantically equivalent to propositional logic and feed-forward networks can be syntactically symbolically encoded into propositional logic. This suggests that feed-forward neural networks can be exactly described by propositional logic. Any feed-forward neural network can be described in propositional logic semantically by a set of knowledge bases or syntactically by a single knowledge base and its consequences. What we have established here is that any solution to a problem given by a feed-forward neural network could have equally been given a solution using propositional logic. Binary neural networks may still have representational differences to propositional logic including an easily computable learning algorithm but there is no fundamental difference in solutions to problems represented by feed-forward neural networks and the solutions represented by propositional knowledge bases. Binary neural networks; however, only represent a small part of the neural networks used in practice. For this reason we move on to look at the relationship between propositional logic and stable networks with finite, discrete, and continuous activation values. As we will see, the move from stable binary neural networks to stable multi-valued networks retains the same equivalence to propositional logic, but moving to infinite-valued and continuous networks changes the relationship somewhat. With infinite valued networks, despite remaining the same most often in practice, the trivial semantic equivalence is no longer valid. Furthermore, the syntactic relationship is true only in an approximate sense. From a theoretical perspective continuous stable networks don't share the same relationship to propositional logic as finite-valued ones but in reality, given that all neural networks are implemented on finite-precision machines, the relationship is still a valid one.

4.4 Stable Finite Valued Neural Networks

Now that we have established the relationship between binary networks and propositional logic we expand our class of networks and reexamine the relationship. What if our neural networks are not binary, but discrete? Can we still relate these networks to propositional logic in the same way? Because finite-valued neural networks extend binary neural networks, all neural en-

codings described in the previous section are still valid. The question, then, turns to the symbolic encodings. We can again rely on the fact that models of propositional logic can themselves be represented by sentences consisting of a conjunction of propositional variables. The trick now is to find a sensible way of encoding the states of a finite-valued network into propositional variables. Luckily there is a fairly obvious way. Suppose each neuron takes values in $\{a_1, ..., a_k\}$. Then for each neuron x_i we associate propositional variables $X^i_{a_1}, X^i_{a_2}, ..., X^i_{a_n}$. If $x_i = a_j$ then, for a semantic encoding, we want the associated model, M, to contain $X^i_{a_j}$ and to *not* contain any other propositional variable associated with x_i. So M should have the property that for $l \neq a_j$, $X^i_l \notin M$. As an example, suppose we have neural network with 3 neurons that each take values $\{1, 2, 3\}$. Given the state, $x_1 = 1, x_2 = 1, x_3 = 1$, the corresponding model is $\{X^1_1, X^2_1, X^3_1\}$. In a finite but non binary valued network a neuron gets associated to n distinct propositional variables where n is the number of values that the neuron takes. Given a configuration of the states, the corresponding model contains the propositional variables corresponding to the values of the neurons in the configuration.

With this identification, we can tackle the problem of symbolic encodings. In the semantic case, we can use the same trivial symbolic encoding from the previous section by enumerating the stable states of the network, translating each stable state to a conjunction, and mapping the network to the knowledge base consisting of the disjunction of the conjunctions associated with each stable state. Replicating the syntactic results from the previous section is similarly easy. Because the bulk of the proof is nearly identical to 4.3.2, we will simply refer to 4.3.2 when making arguments that differ only in the number of implications or other superficial details.

Theorem 4.4.1. *There is a syntactic symbolic encoding for finite-valued feed-forward neural networks into propositional logic*

Proof. Again, associate each neuron with a set of propositional variables corresponding to each of its possible activation values. To construct a knowledge base, enumerate the input/output relationships between each neuron to derive a set of implications of the form $H_{a_i} \leftarrow X^1_{a_{j_1}}, X^2_{a_{j_2}}, ..., X^k_{a_{j_k}}$ where the body corresponds to a configuration of the inputs neurons of h that maps h to the value a_i. The finiteness of the number of neurons and values they take ensures that this list will be finite. construct L by adding each of these implications

along with, for each input neuron, x, X_{a_i} if $x = a_i$ for all times $t > 1$. Call this knowledge base L. We define a similar notion of supported neurons to that found in 4.3.2. An input neuron is supported if its value corresponds to the atom in L. All other neurons are supported if their value corresponds to a an atom which is at the head of a rule whose body contains only atoms which themselves correspond to supported neurons. For example, suppose we have a neuron, y, with value a, if $Y_a \leftarrow X^1_{a_{j_1}} \wedge ... \wedge X^k_{a_{j_k}}$ is a rule in L and $x_1, ..., x_k$ are supported then y is supported if $y = a$. Again we observe that supported neurons are static in value according to the same inductive argument presented in 4.3.2. Given a state $\hat{x}$, Definite $L_{\hat{x}}$ as the conjunction of the atoms corresponding to the values taken on by the supported neuron in $\hat{x}$ along with all consequences of the conjunction. Finally, form $\hat{L}_{\hat{x}}$ as the closure under deductions not involving *modus ponens* of $L \cup L_{\hat{x}}$. The remainder of the proof is identical to 4.3.2 □

What this shows is that by allowing for stable networks to have multiple but finite values, despite the significantly larger number of implications to consider, we can still think of these networks as knowledge bases in propositional logic in which each update of the network carries out *modus ponens* along with all the resulting consequences of conjunction and simplification. Another way we could have gone about this is by converting finite-valued neural networks into binary ones in a way that preserves the dynamics. The procedure for this would be similar to the one used to convert multi-valued neurons into propositional variables. It would begin by translating a single multi-valued neuron into a set of binary ones in which each binary neuron represents a particular value of the translated neuron in such a way that the dynamics of the finite-valued network are equivalent to the dynamics of the binary network. To accomplish this, each pair of connected neurons in the multi-valued network would have to translate the weight value in a way that preserves the same input/out relation in the binary network. One way to do this could be with strong inhibitory connections between all binary neurons associated with the same finite-valued neuron. Such a procedure translates the dynamics of a finite-valued neural network into those of a much larger binary neural network. This suggests a kind of relationship between different neural networks and their equivalence to logical systems. Namely, if we can map the state space of one neural network onto another surjectively in a way that preserves the dynamics, then if there is a symbolic encoding of the target network in a logical system, there is also a symbolic encoding of the initial network in the same logical system.

Theorem 4.4.2. *Given a neural networks N_1 and N_2, if there is a bijective map, f, from the state space of N_1 to the state space of N_2 such that $f(N_1(x)) = N_2(f(x))$, then for a logical system $\mathcal{S}$*

- *if N_2 can be symbolically encoded into $\mathcal{S}$ then N_1 can be symbolically encoded into $\mathcal{S}$*
- *if $\mathcal{S}$ can be neurally encoded into N_2 then $\mathcal{S}$ can be neurally encoded into N_1*

Proof. See Appendix □

This theorem is valid for both semantic and syntactic encodings, but for stable networks we can weaken the condition on f substantially to $N_1(x) = x$ iff $N_2(f(x)) = f(x)$ and retain the result for the semantic case. In other words, in a stable network, as long as f preserves stable states then the semantic relationships remain unchanged moving from one network to the next. What this is beginning to look like is a *categorical equivalence*. That is, we can define maps between neural networks in a way that turns the set of neural networks into a category, similarly we can define maps between logical systems in a way that turns the set of logical systems into a category. When this is done, neural and symbolic encodings are simply functors between the two categories. The previous result then states that isomorphic objects in the category of networks must map to isomorphic logical systems. Further development of this idea is beyond the scope of this thesis so we merely make a note of it and remark that developing this theory could potentially provide valuable insights for neural-symbolic integration in the future.

We have seen that moving to finite-valued networks does not change anything about the relationship between stable neural networks and propositional logic. Because all currently existing physical computing architecture has finite-precision, this means that any neural network implemented in real life, even if ostensibly continuous, will still have the same relationship to propositional logic from a theoretical standpoint. The very high level of precision available to computers, however, means that any translation between a 'continuous' network and propositional logic using the method above would lead to monumentally large knowledge bases and thus have little practical value. Any propositional system containing potentially millions of variables and at least as many rules can no longer claim to be operating on a conceptual level, at

least not one comprehensible to humans. This problem lies at the heart of rule extraction and will be the focus of much of our subsequent discussion. Despite the technical equivalence, for the reasons outlined above, it is useful to discuss the relationship between infinite and continuous-valued neural networks and logical systems.

4.5 Continuous-Valued Neural Networks

In this section we once again broaden the set of neural networks under consideration and establish syntactic and semantic relationships to important logical systems. In this case, the networks being discussed are those with continuous values. This is perhaps the most important case to consider as almost all neural networks used in practice have neurons with continuous-values, at least in the hidden layers.

Many neural networks are continuous-valued. Many are also continuous in time (most notably biological neural networks), but we will limit our discussion to continuous-valued networks operating in discrete time (which we will refer to as continuous networks for convenience). Because there is no bijective map between the state space of a continuous network and a finite one, we clearly cannot call on 4.4.2 to provide the same results for continuous networks that we had for finite ones because the injectivity of our map between state spaces is only required to make the encodings valid. We have the option of considering surjective mappings from the state space of the continuous network to the state space of a finite-valued network and identifying states with the same image as equivalent. Under this equivalence relation we would have a bijective map from the state space of continuous network to a finite-valued one and we could then attempt to apply 4.4.2. However, we may not be able to find a map like this that will preserve the dynamics of the target network. Although for some networks we may be able to find such a map, in other networks this may not be possible. Let us discuss this in more detail. Suppose we have a surjective function between the state space of a continuous network, N_1 and the state space of a finite network, N_2 such that $f(N_1(x)) = N_2(f(x))$. We can use f to define an equivalence relation on the state space of N_1 by $x_1 \sim x_2$ iff $f(x_1) = f(x_2)$. f will be bijective and 4.4.2 will apply. Note that N_1 must be well defined on this equivalence relation because it can be shown that if $f(N_1(x)) = N_2(f(x))$, then $f(x_1) = f(x_2) \implies f(N_1(x_1)) = f(N_1(x_2))$. The

contrapositive of this is that if $f(N_1(x_1)) \neq f(N_1(x_2))$ then we can't have $f(N_1(x)) = N_2(f(x))$. These observations underlie the general procedure for extracting rules from continuous networks. We want to find a partition of the state space that is well defined under the neural dynamics. In other words, if x_1 and x_2 are in the same partition, then so are $N(x_1)$ and $N(x_2)$. When this is satisfied, it is simple to construct a finite-valued network and surjective mapping between the state spaces that preserves the neural dynamics. The target network, by 4.4.1, is equivalent to a set of propositional rules relating the partitions as atomic variables and so by 4.4.2 the continuous network can be symbolically encoded into propositional logic with the given partition of the state space. When this condition is not satisfied, then it will be impossible, at least when encoding the partitions using the same method as 4.4.1.

In general, this property does not hold for an arbitrary partition of a continuous dynamical system. We know it always holds for *some* partition, because it is always true of the trivial partition which maps every state to the same thing. However, what we really want is to be able to make the partitions arbitrarily fine as to be able to describe the network exactly with arbitrary amount of precision. Given a successively finer sequence of partitions, although each individual partition might not satisfy $f(x_1) = f(x_2) \implies f(N_1(x_1)) = f(N_1(x_2))$, it might hold in an approximate sense so that the fidelity of the extracted rules goes to 100% as the partitions get finer and finer. To be precise, we want to find a sequence of partitions, P_i, of the state space, X, each with an associated rule extraction algorithm that has fidelity $1 - \epsilon_i$ such that $\lim_{i\to\infty} P_i = X$ $\lim_{i\to\infty} \epsilon_i = 0$. Where our abuse of notion is meant to indicate that the partition, P_i is a collection of subsets of X that, in the limit consists of every singleton subset of X and that the fidelity of the sequence of rule extraction algorithms is 100% in the limit. This property gives a characterization of an approximate symbolic encoding.

So our old syntactic encoding will no longer work for continuous feed-forward networks, what about our semantic equivalence? Unfortunately, the possibility of infinite stable states renders our obvious encoding invalid. Consider, for example a continuous network with a single neuron that has no bias and a self connection of 1. Every state is a stable state and thus the propositional sentence describing this stable state contains an uncountably infinite number of variables making the sentence not only invalid in propositional logic

but unwritable. Luckily, it is easy to see that the strategy of finite-partitions can recover an approximate version of our previous results. Take some finite-partition of the state space. Then use this to form an equivalence class on the state space as before. Because there are a finite number of partitions, there are a finite number of stable states so the trivial symbolic encoding into propositional logic from 4.3.1 holds. As for semantic neural encodings of propositional knowledge bases, by 4.4.2 it is sufficient to note that every stable binary network is the discretization (transformation of a continuous network into a finite-valued network) of some stable continuous network. Thus, given an arbitrary discretization, stable continuous networks are semantically equivalent to propositional logic under finite partition. Because this is true for any partition, any sequence of partitions that, in the limit, splits the state space into the entire collection of singleton subsets, represents an approximate semantic symbolic encoding because each partition has a rule extraction algorithm with 100% fidelity as shown above.

Now we will show that continuous feed-forward neural networks can be approximately syntactically symbolically encoded into propositional logic. To do this, we will develop a sequence of partitions that vanishes in the limit along with a general rule extraction procedure whose fidelity is 1 in the limit. The most obvious way to discretize the state space of a continuous network is to partition the state space of each of its neurons. By this we mean that for each neuron $x \in [a, b]$, we partition the interval with values $a < c_1 < c_2 < ... < c_k < b$ which results in the partition $[a, b] = [a, c_1) \cup [c_1, c_2) \cup ... \cup [c_k, b]$. For simplicity we assume that the activations of each neuron are bounded and lie in the same interval. Extending this to the more general case does not change any of the arguments but it does make the notation much more complex. Now that we have discretized the activations of x, we can associate them with propositional variables using the same method employed in the previous section. Namely, for each partition, $[c_i, c_{i+1})$ we associate a propositional variable $X_{c_i,c_{i+1}}$. This gives a mapping from states of a continuous network to models of propositional logic. If we have the configuration $x_1 = y_1, x_2 = y_2, ..., x_n = y_n$ such that for all i, $c_{x_i} \leq y_i < c_{x_i+1}$, we map this configuration to the model $\{X^1_{c_{x_1},c_{x_2}}, X^2_{c_{x_2},c_{x_3}}, ..., X^n_{c_{x_n},c_{x_n+1}}\}$.

Assuming we have a feed-forward network, we define a map from states of the neural network into knowledge bases. Each partition determines a set

of variables which in turn determines a knowledge base. This necessarily introduces a degree of error as the rules are defined on variables associated to partitions rather than states. An output neuron may take values from different partitions given two different inputs in the same partition. In some cases we may still have perfect fidelity but this is not generally true. For this reason, the *decision boundaries* of the output neurons become important. If all configurations of an input partition produce an output that is contained between two decision boundaries, then we have a exact neural model for the knowledge base consisting of the corresponding rule. Of course if there are states within a partition that produce outputs on *different* sides of a decision boundary, then there must be some states of the network inside that partition that are not neural encodings of the knowledge base. This is because no matter which rule is assigned to the input and output partition, some states will invalidate the extracted rule.

The inability of a discrete logical system to exactly describe a continuous network is not surprising as some amount of information loss is to be expected when compressing a continuous signal into a finite one. For this reason, in the next chapter we will identify the loss in fidelity induced by mapping a continuous state space onto a discrete one as the *compression error*. In order to produce an approximate encoding, we want to consistently be able to refine our partition so that we can reduce the compression error from rule to an arbitrarily small amount.

In order to make precise the rule extraction method we are using, we will go over in detail how rules are extracted from a feed-forward network using a finite partition of the state space. We must first give some assumptions for our network. We assume that the transfer function is continuous, and that the activation values of the neurons are bounded. The second assumption may seem harsh as it neglects important networks that use transfer functions such as Relu but the range of activation values can be made arbitrarily large and the fact that the input neurons are generally bounded means that activations are essentially always bounded in practice. The first step in our approximate encoding is to discretize the network according to the method above. We again enumerate the input/output relationships between discretized input and output neurons; however, as discussed above, a single discretized input configuration may result in multiple discretized output configurations. In this case,

we add the rule which holds for the greatest percentage of the inputs. By this we mean: assuming a uniform distribution over the networks configurations within the discretized input configuration, select the value for the discretized output neuron as the one that is generated by the largest area within the input configuration of any of the output values. This is why we required continuity and bounded activations, in order to define the relative volume of the regions corresponding to the different output values, we need these regions to be bounded and measurable in the usual sense. Once this is complete, we have produced a finite set of rules in such a way that by using 4.4.1 to add the necessary state information to each knowledge base we have mapped states of our continuous network to a propositional knowledge base in a way that encodes the neural dynamics with a certain fidelity determined by what percentage of the state space violates the condition $x_1 \sim x_2 \implies N(x_1) \sim N(x_2)$.

To give a more detailed illustration of the discretization process, take a simple feed-forward network with three neurons, x_1, x_2, x_3, each with activations in $[0, 1]$, and 0 bias. The weights are $w_{1,3} = 1, w_{2,3} = -1$, and the output neuron has a Relu activation function. Suppose we have the partition $\{\{0.2, 0.6\}, \{0.5\}, \{0.1\}\}$. We associate the continuous variables with discrete ones, namely $x_1 \leq 0.2$, $0.2 < x_1 \leq 0.6$, $0.6 < x_1$, $x_2 \leq 0.5$, $0.5 < x_2$ and similarly for the output variable. Now we have turned our continuous network into a set of discrete variables. This allows us to define functions from the discrete input variables to the discrete output variables. However, these functions cannot be completely accurate because there exist configurations of the input variables for which there are activations in the continuous network which result in outputs satisfying one discrete output variable and activations which result in outputs satisfying a different output variable. For example, take the activations $x_1 = 0.8, x_2 = 0.9$, and $x_1 = 1, x_2 = 0.7$. Both activations correspond to $0.6 < x_1$ and $0.5 < x_2$ but the outputs correspond to 0 and 0.3 respectively which in turn correspond to the variables $x_3 \leq 0.1$ and $0.1 < x_3$. This means that any function on the discrete variables will never always be accurate.

So for each continuous feed-forward network, we can choose a partition of the state space to produce a finite-valued network which can be encoded into propositional logic. The existence of rules whose conclusion does not always correspond to the conclusion derived from the network means that, in general,

the propositional system will not have 100% fidelity with the original network. However, by selecting finer and finer partitions, the number of states in the finite-valued network with ambiguous outcomes will decrease and in the limit go to 0. This is formalized in Theorem 4.5.1. The idea of this proof is intuitive while certain formal details are quite technical. For this reason we will sketch the idea of the proof here before presenting the theorem and full proof. The idea of the proof is that given an output neuron with a partition and a set of inputs. Choose some initial partition of the input space by partitioning the activation values of each input neuron. Given an input configuration, the configuration must land inside a box defined by the partitions of the input variables. Given that the transfer function is continuous, the partition that the output neuron will end up in is determined by a set of decision boundaries in the input space. If the decision boundary does not intersect an input box, then every configuration inside that box will always give an output value in the same output partition meaning the corresponding discrete rule agrees with the network all the time. If one or more decision boundaries intersect the interiors of a box, then the fidelity measured over that box must be less than 100%. However, we can further refine the partition of the input space so that the partition does not contain any points of the decision boundary in the limit. Because the decision boundary has measure 0, in the limit, the overall fidelity will be 100%. Now suppose we have a feed-forward network. The error in a single layer is a function of the error generated by the state space partition in the current layer and the accumulated errors of previous layers. Because we can reduce the error to 0 in each layer, we can reduce the error to 0 of the whole network by starting at the bottom and reducing the error to be arbitrarily close to 0 and doing the same for the next layer until we get to the top. So given any partition of the output neurons there is a partition of the remaining neurons along with a rule extraction method that gives fidelity $1 - \epsilon$ for any $\epsilon > 0$. We get an approximate syntactic symbolic encoding by choosing a sequence of output partitions with corresponding input partitions in a way so that the partitions converge to the whole state space and the fidelity converges to 1. We give the exact statement and proof of this as follows.

Theorem 4.5.1. *In a continuous feed-forward network, for any $0 < \epsilon \leq 1$ we can always find a partition of the state space such that there is a syntactic rule extraction algorithm defined from the state space of the network to propositional logic with fidelity greater than $1 - \epsilon$.*

Proof. We begin by defining partitions of $\mathbb{R}^n$ and proving some of their prop-

erties. A *box* in $\mathbb{R}^n$ is a compact subset of the form $\{x : a_{x_1} \leq x_1 \leq b_{x_1}, a_{x_2} \leq x_2 \leq b_{x_2}, ..., a_{x_n} \leq x_n \leq b_{x_n}\}$. A box partition of $\mathbb{R}^n$ is a set of boxes whose union is $\mathbb{R}^n$. We can define a finite box partition of $\mathbb{R}^n$ by choosing points $-\infty < a_1 \leq a_2 \leq ... \leq a_k < \infty$ for each x_i. A box in this partition is defined by pairs of points for each x_i so that $a_{i_j} \leq x_i \leq a_{i_{j+1}}$. Each choice of pairs of points clearly defines a unique box and the union of all such boxes is obviously $\mathbb{R}^n$. Call box partitions of this form *interval* partitions. Although not every box partition is an interval partition, it is easy to see that for every box partition, there is an interval partition that is finer than the box partition. This is done by choosing the collection of all endpoints of x_i used by the boxes in the partition as points for the interval corresponding to x_i. Doing this for each x_i gives an interval partition whose boxes are strictly contained in the boxes of the original partition.

Now suppose we have a neural network consisting of n input neurons, x_i, and a single output neuron, h. Suppose h is partitioned by $a_1, a_2, ..., a_k$. Assuming that h has a continuous transfer function, f, $f^{-1}((a_i, a_{i+1}))$ will all be open sets including the unbounded intervals. It is a fact from point-set topology that any open set in $\mathbb{R}^n$ can be expressed as a countable union of boxes. Take the first m such boxes for each $f^{-1}((a_j, a_{j+1}))$ and take their union. Because each of these sets are disjoint and they cover the entire input space, this will give us a box partition of the input space. From these boxes we can define an interval partition finer than the union of all of these boxes. Call this partition p_m.

Now we define a mapping from states of the neural network to a propositional knowledge base. Given our partition, p_m, the activations for each neuron are divided into intervals. For example, given neuron x_i, p_m defines intervals $x_i \leq a_1^i, a_1^i < x_i \leq a_2^i, ..., a_k^i < x_i$. Associate with each of these intervals a propositional variable, $A_{j,l}^i$ where i indicates the neuron that the variable corresponds to and j, l correspond to the endpoints of the interval. Note j may be $-\infty$ and l may be ∞ if the activations of the neuron are unbounded. This is done for the output variable h as well giving us propositional variables of the form $H_{j,l}$. Each box is thus associated with a conjunction of propositional variables. For each box we associate a rule whose body consists of the associated conjunction and whose head is the output variable corresponding to the output partition mapped to by the majority of the input states within a box. Majority is defined as percentage of volume where volume is defined in

the usual way with the Lebesgue measure. If a box has infinite volume then if every state in the box maps to the same output partition assign the variable for that partition to the head of the rule, otherwise choose an arbitrary variable to assign to the head of the rule. Define a knowledge base L as the collection of each of these rules. To each configuration of the network we associate the knowledge base L along with the conjunction of supported atoms and their consequences under propositional inference excluding *modus ponens*. Supported atoms are defined as in 4.4.1. Configurations of input neurons are always supported and other neurons are supported only if their value is in a partition that corresponds to a variable at the head of a rule whose body only contains literals that themselves represent supported neurons.

If a box is contained in $f^{-1}((a_j, a_{j+1}))$ for some values, a_j, a_{j+1}, in the partition of an output neuron, h, then every input configuration inside the box results in a value of h that corresponds to the variable $H_{a_j,a_{j+1}}$. Thus the rule $H_{a_j,a_{j+1}} \leftarrow p_{m,l}$ (where $p_{m,l}$ is the conjunction of all atomic variables corresponding to the box) is an exact syntactic neural encoding for states that are inside the box as the rule is satisfied for all initial states of the network. When there are states inside a box on either side of a decision boundary, then the associated rule will have less than 100% fidelity depending on ratio of the volumes of the regions inside the box but on either side of the decision boundary. If every box of input neurons was contained strictly inside the decision boundaries of the output neurons, then following the same reasoning as 4.4.1 we could construct an syntactic symbolic encoding of the network using the partition to define the atomic variables. The fidelity of a knowledge base on the input space, I, with partition p_m can be decomposed into $Fidelity(I) = \frac{Vol(\overline{I_{p_m}})}{Vol(I)} Fidelity(\overline{I_{p_m}}) + \frac{Vol(I^\circ_{p_m})}{Vol(I)} Fidelity(I^\circ_{p_m})$. Where $\overline{I_{p_m}}$ is the set of boxes containing points on either side of a decision boundary and $I^\circ_{p_m}$ is the set of boxes whose points are all on the same side of every decision boundary. As shown above, we can extract rules so that the fidelity on $I^\circ_{p_m}$ is 100%. Since each $f^{-1}((a_j, a_{j+1}))$ is exactly the union of an infinite sequence of boxes, for any input state, x, that is not on the decision boundary we can choose an m such that in the partition p_m, x is contained inside a box strictly contained inside of some set $f^{-1}((a_j, a_{j+1}))$. Thus in the limit, $\overline{I_{p_m}}$ is equal to the union of every decision boundary. Because the each decision boundary is an $n-1$ dimensional subspace of $\mathbb{R}^n$ and the set of all decision boundaries for the output partitions is the countable union of each individual

decision boundary, the set of states on a decision boundary has measure 0 and thus in the limit as $m \to \infty$, we have $\frac{Vol(\overline{I_{pm}})}{Vol(I)} = 0$ and thus the fidelity is 100%.

For multiple output neurons it suffices to note that for every pair of partitions there exists a partition finer than each. Thus, given the set of partition sequences corresponding to the output neurons we can form another sequence of partitions such that the fidelity of the extracted rules for every output neuron is 100% in the limit. Note that if the activations are unbounded then we cannot make sense of the fidelity on unbounded boxes as the volume is infinite. However, the same sequence of partitions will converge to a partition in which the boxes with infinite volume are strictly contained within the decision boundaries and can thus simply be assigned a fidelity of 1.

Moving on to multi-layer networks, we fix an output partition and use the previous step to find a partition with fidelity $1 - \epsilon$. Repeat this layerwise so that the rules extracted from each layer have fidelity $1 - \epsilon$ over their respective input spaces when considered as independent layers. Because we can refine the partition of each layer to obtain arbitrarily high fidelity, starting with any partition of the output neurons, we can find a partition of the input neurons that gives a certain fidelity to the output neurons. Repeating this process by moving down each layer, because the number of errors accumulated at each layer is at worst additive and there are a finite number of layers, we can always choose a partition so that the overall fidelity is $1 - \epsilon$ for arbitrary $\epsilon > 0$ $\square$

We have shown that feed-forward networks can always be syntactically encoded into propositional logic, at least in the limiting case. On a formal level, this establishes that the difference between logical systems and feed-forward neural networks is a representational one. However, our proofs often involved adding a massive number of rules to a knowledge base. This theoretical correspondence is therefore not a practical one. As we will see in the next chapter when we review different rule extraction techniques, we consistently end up sacrificing a degree of fidelity in exchange for reduced complexity. As we will also see, semantic encodings of neural networks have usually focused on other kinds of logic considered more aligned with dynamics of neural networks. Before we review the major neural encoding and rule extraction techniques, there are some loose ends to tie up.

4.6 Unstable Networks

So far we've discussed stable networks, that is, networks which always settle down into a stable state. Stable state dynamics is convenient for neural-symbolic integration since it allows us to model the semantics of a logical system in terms of the stable states. As we will see in a future section, stable-state semantics have not just been applied to propositional logic, but also other forms of logic due to the simple nature of their dynamics. Stable networks are very common in practice, mostly due to the prevalence of feed-forward networks, but unstable networks are also common. Unfortunately, for unstable networks the question of neural-symbolic integration, at least in the current formulation, has some issues. We will explore these issues now.

To begin, we first note that all unstable networks are recurrent. Of course, there are many examples of recurrent networks which *are* stable, including SCNs as well as other kinds of networks which we will revisit in our discussion of the neural encoding of propositional logic programming, but recurrence is a necessary condition for a network to be unstable. The next thing we note is that every unstable recurrent network with finite activation values will settle into a cycle. Although this type of network may have stable points, for most initial states, we will eventually run into a cycle. This is because, assuming a finite number of neurons, the state space is finite. The implication is that eventually we will have to return to a previously visited state. Because the network is deterministic this means it must revisit the same states in the same order it did previously until it loops back again. When the network has continuous activations we can have very complicated dynamics which poses a problem for any finite symbolic rule extraction techniques.

One common explainability approach for recurrent networks is to associate them with a finite automata. This is done by partitioning the state space of a continuous network and using the networks dynamics to define a transition function on the set of states defined by the partition. Many different approaches for this have been developed Jacobsson [2005]. This approach doesn't technically fit into our definition of a symbolic encoding because it uses finite automata rather than logical systems. Rule extraction from recurrent networks is an important example of neural-symbolic integration with symbolic systems that are not logical systems. If the recurrent networks have

finite values, then they can always be transcribed exactly into a finite automata. Like before, this can be done by simply enumerating the states of the networks and defining the transition function. When the networks are continuous, we run into serious theoretical limitations. The first is that continuous recurrent networks are capable of *hypercomputation*, Siegelmann [1995]. Hypercomputation is the ability of a network to compute non-Turing computable functions. The ability of a system to do this depends, crucially on the use of continuum. A neural network capable of hypercomputation can never be simulated by a finite-automata due to the fact that it is in a different computability class. Furthermore, since the symbolic systems we have been concerned with are discrete, the behaviour of continuous networks capable of hypercomputation simply cannot be captured by finite symbolic systems. It is still possible, perhaps even likely, that symbolic systems which themselves use continuous variables are capable of encoding unstable neural networks either semantically or syntactically. This question is left open for future work.

Although the existence of recurrent networks capable of hypercomputation is a concern for rule extraction, the vast majority of continuous neural networks are *not* capable of hypercomputation. In fact, the practical implications of hypercomputation has seen heavy criticism. It is unclear whether or not hypercomputation is physically realizable at all Davis [2006]. The relevance of recurrent networks capable of hypercomputation to effective rule extraction is dubious at best. Despite this, there is another fundamental issue we run into when attempting to do rule extraction on continuous recurrent neural networks. Different partitions of the state space may lead to output sequences which belong to grammars of different computational classes Kolen [1994]. This observation has lead to the criticism that rule extraction for recurrent continuous networks may be a dead end.

To summarize, using a finite symbolic system to emulate a continuous one will, in general, result in a degree of error (which in the future we will call the compression error). For feed-forward networks we can always reduce this compression error by choosing a finer partition. This introduces new variables to the symbolic system and results in a more complex description of the network. In other words, we can always increase fidelity by increasing complexity. For recurrent networks we see that there are, at least in certain cases, fundamental computational differences which suggest no discrete symbolic system could

be considered an adequate description of the network. It should be noted, however, that in practice truly continuous machines would require infinite (or at least arbitrarily high) precision. In particular, since every neural network today is implemented on a Turing-complete machine they, too, are Turing complete and the computational considerations described above do not apply to them. Considering we will be focusing our attention on stable networks, the limitations on the effectiveness of rule extraction have to do with the required complexity of the rules rather than some fundamental difference in the two computational models.

Although the majority of neural-symbolic integration with recurrent networks has used finite automata, integration with logical systems has been developed for certain kinds of recurrent networks de Penning et al. [2014]. What distinguishes this encoding from the ones discussed previously is that the neural network in question is probabilistic. In the next section we will discuss how our definitions of neural-symbolic integration can be extended into multi-valued logic and probabilistic networks.

4.7 Neural-Symbolic Integration with Other Logical Systems and Neural Networks

The framework developed in section 4.2 gives a precise account of neural-symbolic integration between deterministic networks and logical systems with two truth values. However, this excludes many important classes of neural networks and logical systems. Luckily, our definitions can be extended to include these cases. These extensions are the focus of this section.

We start with a generalized definition of a logical system which includes multi-valued logic.

Definition 4.7.1. *A multi-valued logical system is a quadruple,* $\mathcal{S} = (\mathcal{L}, \vdash_{\mathcal{S}}, \mathcal{M}, T)$. *Where* $\mathcal{L}$ *is a recursive language,* $\vdash_{\mathcal{S}}$ *is a computable relation on* $2^{\mathcal{L}\times T}$, $\mathcal{M}$ *is a set of models, and* $T \subset \mathbb{R}$ *is a set of truth values.*

A model is a set, M, along with a truth function $f_M : \mathcal{L} \to T$. The difference between this definition and the definition for a boolean logical system is that we must specify a particular truth value for each sentence in a knowledge base in order to apply an inference procedure to it. Likewise a model does

not necessarily assign a value of true or false to a sentence, but rather one of many, possibly infinite truth values. We can see that this definition generalizes the old one by setting $T = \{0, 1\}$ and only defining the deductive relation on subsets which assign a value of 1 to every element of the knowledge base. By extending our definition in this way, we can formalize neural-symbolic integration for a whole family of logics which have played an important role in AI. Most notably among these is fuzzy logic. We would like to also note that the central barrier to neural-symbolic integration with recurrent continuous valued networks is now not an issue, because there are continuous-valued logics which would not be required to partition the state space of a continuous network in order to define a map from the states of a network to models of the logic. In this sense, the theoretical problems of neural-symbolic integration with recurrent networks are not an issue with symbolic systems themselves but merely an issue with representing a continuous state space with a discrete one.

Like multi-valued logics, we can incorporate probabilistic networks into our framework of neural-symbolic integration by generalizing the definitions of a semantic encoding. Given an initial state x_0, a probabilistic network defines a probability distribution on its state space at every time t. We write a configuration of the state space at time t as $x^{(t)}$. If P is the probability distribution over all sequences defined by the network, the probability of a configuration at time t given initial state x_0 is $P(x^{(t)}|x^{(0)} = x_0)$. Assuming we have a map from configurations of the state space to models of a logical system as before, then we have the following definition for a neural model

Definition 4.7.2. *A neural network is a model for a knowledge base L if, given any initial state x_0, if x is not a model of L then* $\lim_{t\to\infty} P(x^{(t)} = x|x^{(0)} = x_0) = 0$

Recall that for deterministic networks, the L-models of N are those models which appear infinitely often given some initial state. We extend this to probabilistic models by defining an L-model of N as a model, x, of L such that there exists an initial state x_0 such that for all t, there exists $t' > t$ with $P(x^{(t')} = x|x^{(0)} = x_0) > 0$. Again this definition simply discounts models which only appear temporarily in the dynamics of the network. With our definition of L-models we can define semantic neural and symbolic encodings in the same fashion as with deterministic networks. Although it is possible to extend the definition of syntactic encodings, to our knowledge there have not been any attempts at probabilistic syntactic encoding so we defer this definition to later work.

In the deterministic case, stable neural networks play an important role for semantic encodings. The probabilistic analog of a stable state is a *stable distribution*. A stable distribution is one in which $P(x^{(t)})$ does not change as t changes. Many important stochastic processes settle to a stable distribution. In the context of neural networks the best example of this is the Boltzmann machine which, given any initial state, converges to a stable Boltzmann distribution. In the case that a probabilistic network is stable, we only need to check whether configurations which are not models have a 0 probability in the stable distributions in order to verify that a network is a model for a knowledge base.

The final thing to consider is the fidelity of a rule extraction algorithm. For deterministic networks we defined the fidelity as the percentage of initial states which were models of a knowledge base. We may be tempted to use the same definition for probabilistic networks; however, this is inadequate. To see this, consider the Boltzmann machine which converges to the same distribution for every initial state and therefore any Boltzmann machine which is not a model will have 0% fidelity even if there is a 99% chance of a state in the stable distribution being a model. A more sensible definition in this case would be the probability in the stable distribution that a state is a model. However, not all stochastic processes are stable. What we want is some time t at which the probability of being in a model state is never lower than in the future than it was at time t. In other words, at time t the probability of being in a model state is always at least what it is now. Furthermore, we want the time with the maximum such probability. Thus, for a given initial state, x_0, the fidelity with respect to x_0 is defined as follows, let M_N

Definition 4.7.3. *The fidelity with respect to* x_0 *is* $\max_t P(x^{(t)}$ *is a model* $|x^{(0)} = x_0)$ *where* t *satisfies the property that* $\forall t' > t, P(x^{(t')}$ *is a model* $|x^{(0)} = x_0) \geq P(x^{(t)}$ *is a model* $|x^{(0)} = x_0)$

This definition may seem difficult to work with, but in the case we have a stable distribution it reduces to the probability of a state being a model in the stable distribution. We will use this definition when calculating the fidelity of rule extraction techniques when applied to Boltzmann machines. The fidelity of the whole network is defined as the average fidelity over all initial states. All of these definitions can be seen as generalizing the ones we gave for deterministic networks. This is because a deterministic network can be seen as

a probabilistic network with the probabilities for any state transition to be either 1 or 0. When formulated this way, the definitions of models and fidelity become equivalent to the definitions we gave for deterministic networks. Now that we've given a thorough definition for the kinds of relationships different logical systems and neural networks may have, we will discuss some of the practical and representational differences between the two approaches before giving a review of the various neural encoding and rule extraction methods developed over the years.

4.8 Learning and Representation

As we have seen, for many important cases, neural networks can either be thought of as defining the semantics of a logical system, or as implementing the deductive apparatus. This type of formal equivalence is in addition to the more general computational limitations obeyed by both with the only exception arising when the neural networks are continuous (either in time or the state space). However, since in almost all practical scenarios neural networks are implemented on a discrete computer, the networks will be limited by Turing completeness. Given this kind of formal equivalence, what, then, is the difference between symbolic systems and neural networks? This question is examined further in this section.

There are two possible angles to look at: The first of which is the learning procedures. Much of the success of neural networks can be attributed to the use of continuous optimization algorithms such as gradient descent LeCun et al. [2015]. Although symbolic learning algorithms exist, the power of gradient descent along with its very general applicability give it an advantage over the learning algorithms of symbolic systems. As discussed before, this is one of the main motivating factors for the translation of symbolic systems into neural networks Garcez et al. [2008]

The second area of difference is a more subtle one: representation. It has been argued that one of the key features of neural networks allowing them to have such good generalization ability is that concepts are encoded using *distributed representations*, Smolensky [1988], LeCun et al. [2015]. In a distributed representation, the abstract concepts relevant for reasoning are not represented with a single variable or neuron, but rather from patterns of ac-

tivity over many neurons Smolensky [1988]. At the surface, the difference between a local and distributed representation in a neural network is negligible as any distributed representation is isomorphic to a local one via linear transformation. However, distributed representations have been shown to exhibit *graceful saturation*, that is, as more states of a system are encoded into a finite network the errors accumulate at a reasonable rate rather than resulting in a catastrophic failing of the network Smolensky and Legendre [2006].

Another important feature of distributed representations is a notion of distance between concepts. In a distributed representation, we can measure the distance between points in the vector space representing completely different concepts. In a purely symbolic system, relations such as *Paris:France* and *Rome:Italy* have no meaningful similarity even though conceptually they represent a similar relationship. All similarity has to be encoded axiomatically, whereas in a distributed representation we are given a metric by default. This has been proposed as a mechanism for the generalization capabilities of neural networks. Even if an input is completely new it will be spatially close to similar inputs which may be familiar. In this view, distributed representations equip a concept space with a geometry that can be used for strict logical inference as well as encoding fuzzier relations between concepts. The idea of using the geometry of embedded distributions as a model theory for symbolic systems has been partially formalized Guha [2014] and further neural symbolic integration using the geometry of embedded distributions has been pursued in Serafini and d'Avila Garcez [2016].

According to the previous analysis, the benefit of neural networks lies in their use of many *subsymbolic* variables to represent symbolic structures. This allows neural networks to implement the type of intuitive reasoning required for generalization that symbolic systems are unable to capture. This observation seems to be in contrast to the formal neural-symbolic equivalence that was discussed previously. Since, for all practical purposes, both systems are computationally equivalent, it should in theory be possible to define a symbolic system which is able to reason like a neural network. The key difference between the symbolic systems in use and the kind that would be required to implement general neural networks is that we require our symbolic systems to be relatively compact, with variables and atoms that represent very abstract concepts. A symbolic system may reason about an object like *dog* or

cat whereas a neural network will reason over a large set of variables each one of which may not represent anything particularly meaningful. This leads us to what is perhaps the fundamental difference between symbolic and connectionist systems: compactness. The question, then, isn't whether or not there is any fundamental difference between neural networks and symbolic systems but rather to what degree can a dataset be compressed into a set of variables that can still make accurate predictions about the label. The representational issues outlined above boil down to how much information is lost when transitioning from a high dimensional geometric space to a low dimensional symbolic one.

Chapter 5

Encoding and Extraction Techniques

5.1 Other Neural-Symbolic Relationships

In the previous chapter, we explicitly proved some very general relationships between logical systems and propositional logic. The only features we considered were the set of values taken on by the neurons and whether or not the network had a recurrent or feed-forward architecture. From this information alone we can establish some important relationships, but many other, often closer, relationships have been established by relating logical systems other than propositional logic to networks with a more restrictive architecture. Also notably absent from the previous chapter is any discussion of specific rule extraction techniques. This is because rule extraction is more concerned with establishing an approximate relationship between a neural network and a logical system in terms of fidelity rather than an exact encoding.

In this chapter we will review some of the most important equivalences that have been established in neural-symbolic integration as well as the most notable rule extraction techniques. We will describe the equivalences in terms of semantic and syntactic encodings. In several cases we will either extend an encoding to an equivalence or comment on the class of networks represented by its image. This has two purposes, the first is to provide a more detailed image of how various logical systems and neural networks relate to each other. The second, is to validate our neural-symbolic framework by showing that previous work in neural-symbolic integration fits into the chosen definitions. The discussion of rule extraction will be more focused on technique as motivation for,

as well as a point of comparison to, the extraction algorithms developed in Part II. We conclude the chapter by summarizing the neural-symbolic relationships that are established in chapters 4 and 5.

5.2 Other Neural Encodings

So far we have shown that stable networks are, at least in a limiting sense, semantically equivalent to propositional logic. However, neural encodings of stable networks have been developed for other logical systems as well. In this section we will discuss some of these encoding methods.

Most relevant to the development of one of our own algorithms is the semantic equivalence of penalty logic and SCNs. Recall that Penalty logic was explicitly developed in an attempt to capture the reasoning properties of SCNs (see section 3.5). The relationship between SCNs and penalty logic turns out to be a semantic one. First we must show that there is a neural encoding of penalty logic. To do this, we associate each knowledge base in penalty logic to an energy function Pinkas [1995]. To construct this energy function, first we translate a knowledge base into a function over the propositional variables (interpreted as binary variables) such that the minima of the function are exactly the preferred models of the knowledge base. After doing this, we construct an energy function for SCNs that is equal to the previous function up to some constant difference c. This function will have the same minima as the previously defined function. This gives a one-to-one correspondence between minima in the energy function and preferred models of the knowledge base. Since SCNs are stable, the neural models of this encoding are exactly the minima of the energy function, which are exactly the preferred models of the knowledge base. Therefore we have a neural encoding. To show that every SCN has a semantic encoding we merely have to repeat the construction in reverse. For every energy function of an SCN, we show that there is some function over a set of binary variables whose minima correspond to the preferred models of a knowledge base that is equal to our energy function up to a constant difference c. By the same reasoning as before this is a symbolic encoding of the knowledge base.

One minor difficulty with this procedure is the existence of high order terms. The energy function we construct may have terms of the form $x_1x_2x_3$. This can be interpreted as a multi-weight, a connection between three neurons. In

most definitions of neural networks, a connection exists only between two neurons. Luckily, the addition of hidden units solves this problem. A connection between three neurons can be replaced by three two-valued connections all to a single hidden neuron. In other words, we replace the term $x_1x_2x_3$ with three terms hx_1,hx_2, and hx_3.

Before we continue we should make a minor note on monotonicity. We have now shown that SCNs are equivalent to both propositional logic and penalty logic. This first seems to pose a contradiction: while propositional logic is monotonic, penalty logic is not. Indeed, one can construct a map from penalty logic to propositional logic by first mapping a knowledge base to an equivalent network, and then mapping that network to an equivalent knowledge base in propositional logic. Because this is a semantic encoding, the link here is that they share the same set of models. However, if we add a sentence to the knowledge base in penalty logic and map it to a neural network, the knowledge base in propositional logic we get from mapping the neural network back into propositional logic will not necessarily consist of the previous knowledge base with an additional sentence. Indeed, if any connections in our network are changed, or added, many of the sentences we had in our old propositional knowledge base will not be there. All our map has really shown here is that for every knowledge base in penalty logic, there is a knowledge base in propositional logic with the same models, but this could be seen trivially anyways.

Next we will look at neural encodings for logic programming. Logic programming has been a common target for neural-symbolic integration due to its common use in AI. Many variants of logic programming have been given neural encodings making the integration of logic programming and neural computation one of the greatest successes of neural-symbolic integration. However, encodings are generally of propositional logic programming, or, if they are encodings for first-order logic programming, then they deal with a reduced language and model-space that allows them to be reduced to what is essentially a propositional language. An explicit neural encoding of fully-fledged first-order logic programming remains one of the challenges for neural-symbolic integration. As we have discussed Chapter 4, this challenge must be overcome using syntactic, rather than semantic encodings.

One of the first neural encodings of logic programming was Knowledge-Based

Artificial Neural Network (knowledge based artificial neural networks) Towell and Shavlik [1994]. Similar to the previous encodings we have described, KBANN encodes acyclic Horn clauses into feed-forward neural networks by assigning atomic variables in the logic program to neurons in the network. This is done by viewing the logic program as a hierarchy. To see an acyclic logic program as a hierarchy, start by taking any rule in the logic program. If any of the variables in the body of this rule are themselves the head of another rule, go 'down a level' and consider that rule. Repeat this procedure until you reach a rule for which none of the variables in its body are the heads of any rules. The 'level' of the variable at the head of a rule is the longest path to a rule for which there are no variables in the body that are the heads of any rule. in KBANN, the level of a variable determines which layer in the network it will be placed in. The first step of KBANN is to rewrite the logic program to eliminate disjuncts of more than one variable, ie multiple clauses with the same head whose body contains multiple variables. This is done by adding new variables corresponding to each disjunct. For example, if we have the disjuncts $A \leftarrow B \wedge C$ and $A \leftarrow C \wedge D$, then we rewrite this as $A' \leftarrow B \wedge C$, $A'' \leftarrow C \wedge D$, $A \leftarrow A'$, $A \leftarrow A''$. This is done because neural networks are threshold units. If any combination of variables reaches the threshold then it will fire regardless of which particular combination of variables that happens to be. Now translate the logic program into a neural network by assigning each variable a neuron and for each rule set the weights so that the head is only true if each neuron in the body is true. Insert appropriate biases for neurons which are the head of a rule with no variables. Notice that this network, regardless of its input, will settle at a single stationary configuration, namely the one in which all input units with biases are on and all other are off. This is because, as discussed in the section on logic programming, the semantics of Horn clauses are determined by their minimal model. Thus this is a neural encoding. In its original formulation, KBANN perturbs the weights and adds additional neurons meant to represent additional facts that may be relevant to the problem. The intention of KBANN was to create a network that can be trained to learn new facts related to the problem. For our purposes, we are only interested in the ability of KBANN to encode logic programs and for this reason we do not discuss these aspects of it here. What about the other direction? can KBANN always be encoded in a logic program? The important thing to ask here is: what is the class of neural networks that KBANN represents? Because it doesn't deal with negation, the encoding we have given never uses negative weights. Thus

we should ask whether or not every feed-forward network with positive weights can be expressed as a logic program. The answer is yes. We merely use the same encoding of rules as we did for propositional logic, but only add those rules for which the head is not negated. In other words, we only add the rules which turn on the output neuron. Additionally, we drop any negated terms from the body of each rule. It is not difficult to check that the single stable state of our network will be the minimal model of our extracted Horn clause. With the converse established, we can state that Horn clauses are semantically equivalent to feed-forward networks with positive weights. We can also adapt the propositional syntactic encoding used in the previous chapter to this case. Equipping Horn clauses with the deductive rules discussed at the end of section 3.2 excluding the inference rule for negated literals allows us to use the same symbolic encoding as we did for propositional logic to encode binary feed-forward networks with positive weights into Horn logic programming. The corresponding neural encoding now also holds using the encoding developed above with the mapping from neural states to truth assignments replaced by an appropriate map to conjunctive clauses along with the required rules and all sentences inferred from conjunction and simplification of the supported neurons. Thus feed-forward networks with positive weights are syntactically and semantically equivalent to Horn logic programming (for continuous values this becomes approximately true in the same sense as 4.5.1).

The neural encoding techniques of KBANN were extended to general logic programs with the Connectionist Inductive Logic Programming (CILP) algorithm Garcez and Zaverucha [1999]. CILP uses a similar encoding technique to KBANN, but condenses the network by using only a single hidden layer. Furthermore, CILP adds recurrent connections from the output neurons to the input neurons. Given a general logic program, CILP implements the least-fixed point operator of the logic program. This implies that CILP associates a neural model to every *acceptable* logic program. Acceptable logic programs are a broader class of logic programs that are not necessarily Horn clauses but for which the least-fixed point operator always converges to a stable minimal model. Like Horn clauses, the semantics of an acceptable logic program is defined entirely by the stable models of the least-fixed point operator and, given enough time, the least fixed-point operator of the logic program is guaranteed to converge to the unique stable model. Because CILP implements the least-fixed point operator, given any input, the CILP network associated to

an acceptable logic program will eventually converge to a stable state corresponding to a minimal model of the logic program. Because the models of an acceptable general logic program are the fixed points of the least-fixed point operator and every such fixed point will be a fixed point of CILP with some initial state, CILP is a semantic neural encoding of acceptable logic programs. Note that the converse, however, is not true. Given a feed-forward network with recurrent connections from the final layer to the first layer, there isn't necessarily a symbolic encoding of it with acceptable logic programs. Take, for example, the CILP network corresponding to the program $\{A \leftarrow \neg B, B \leftarrow A\}$. This program does not have a unique minimal model and the least-fixed point operator will cycle through every configuration of A and B including the states $(A, B) = (0, 0)$ and $(A, B) = (0, 1)$ which are not models of the program. This implies that the CILP network corresponding to this logic program is not a neural model of it. The main techniques of CILP have been extended to provide neural encodings for other classes of logic programming including modal, temporal, and epistemic logic programs Garcez et al. [2008].

Moving from propositional logic programming to first-order logic programming, Markov Logic Networks (MLNs) give a probabilistic semantic encoding of a restricted set of first-order logic programs Richardson and Domingos [2006]. MLNs restrict first-order logic programs by removing all function symbols and constructing models from the Herbrand base rather than the more general model theory of first-order logic. MLNs associate a Markov network to each grounding of a knowledge base so that every model of the logic program has a non-zero probability in the stationary distribution and every state which is not a model of the knowledge base has 0 probability in the stationary distribution. In other words, MLNs give a probabilistic semantic neural encoding of a knowledge base. Conversely it is shown that every probability distribution can be represented as a MLN. Although Markov Networks are technically not themselves neural networks, they can be represented to arbitrary precision by Bolztmann machines by a universal approximation theorem Le Roux and Bengio [2008] making this a semantic equivalence between Boltzmann machines and (restricted) first-order logic programming.

This is just a brief overview of the most relevant neural encodings that have been developed over the years. As one can tell from this overview, the focus of neural encoding techniques has been on semantic encodings of logic pro-

gramming, despite there having been syntactic encoding methods developed as well. Next we will turn our attention to the other side of neural-symbolic integration, rule extraction.

5.3 Extracting Knowledge from Neural Networks

Neural encoding is one of the major tasks of neural computing. Combined with rule extraction, they form the two main halves in the neural-symbolic feedback process. Two thirds if you include hybrid systems, and progressively smaller fractions depending on how far you want to stretch the definition of neural-symbolic computing [1]. One obvious difference between rule extraction methods and the neural encoding techniques that we've examined is every translation from a logical system to a neural network that we've discussed is a full blown neural encoding. For semantic encodings this means that we define a network which captures the semantics of our target logical system exactly. Conversely, many of these networks are themselves symbolic encodings of the same logical system making them semantically equivalent. Rule extraction has historically been done with the aim of explainability. This means that a perfect symbolic encoding of a neural network, even if it is possible, may not be desirable as the resulting knowledge base may be just as complicated as the network itself. Encoding methods, on the other hand, are not concerned with the complexity of the network and thus only exact neural encodings were considered. The result is that we have a wealth of rule extraction methods but there are no approximate neural encoding techniques. Reflecting on chapter 4, it is clear that this isn't the result of fundamental incompatibilities between symbolic systems and neural networks so much as it is the result of different goals for each task. The purpose of this section is to do a short review of the rule extraction techniques developed over the decades. We note that every rule extraction technique presented here is semantic in nature, it works by mapping models of a logical system to states of a network.

We begin by examining the problems preventing us from extracting rules which can describe neural networks perfectly. As discussed in chapter 4, for most neural networks we can, in principle, find a rule set which describes them accurately to an arbitrary desired precision. However, in practice, finding an

[1] Encoding, Extraction, and Hybridization can be see as very broad categories that can further be refined into a larger set of more precise tasks

adequate set of rules can be difficult due to the large search space. This forces to reduce the search space in order make rule extraction. By reducing our search space we will often not be able to guarantee that a rule extraction method can find a solution with the desired accuracy. With this in mind we identify all possible sources of error for rule extraction methods and the different contexts for which they become problems.

The most immediate difficulty for rule-extraction algorithms of any kind is the time complexity. A simple perceptron with a single output neuron and n input neurons has 2^n input configurations which results in a maximum of 2^n conjunctive rules that can potentially describe the network. Even for very small binary network this makes an exhaustive search computationally costly. In networks containing neurons that take multiple or even continuous values, exhaustive searches become impossible. In order to overcome this, rule extraction algorithms will either employ heuristic searches or simply restrict the search space to something more manageable.

Even if a particularly clever algorithm is able to find a set of rules with 100% fidelity efficiently, the rules themselves may still not be acceptable due to their high complexity. To illustrate, consider a finite-valued feed forward network. We know from 4.4.1 that for any finite-valued feed forward network, we can find a propositional knowledge base that exactly encodes the dynamics of the network into a set of propositional rules. The procedure used to prove this involved enumerating every network configuration and recording all the input/output relationships; effectively turning the network into a giant lookup table. Besides being computationally difficult for the previously described reason, the motivation behind such an act is questionable. Ideally we would want to find a different symbolic encoding that uses far fewer variables and rules while. Whether or not this is possible is a property of the network in question.

The final problem facing rule extraction occurs when trying to find symbolic encodings of feed-forward continuous networks into discrete systems. As discussed in section 4.5, any continuous network must be discritized in some way before it can be symbolically encoded into a discrete network. Although in some cases, there may be a way of discritizing a continuous state space that admits a syntactic symbolic encoding, in general, no matter the discritization, a continuous network will have no symbolic encoding. 4.5.1 does guarantee

that with a fine enough partition we will be able to find a set of rules with arbitrarily high fidelity but this comes at the cost of adding additional variables potentially increasing the complexity of the extracted rules.

Even though we have shown that (up to arbitrary approximation for continuous networks) stable networks are semantically equivalent to propositional logic, and feed-forward networks can be symbolically encoded into propositional logic, this does not mean we can easily translate a neural network into a logical system. Even though there are theoretical equivalences, the practical limitations on the search algorithm along with the desired complexity properties of a solution mean that it might be impossible to produce a desirable solution using rule extraction. The sources of error that we have identified above are broken down into the following.

Compression Error: The error induced by mapping a continuous state space onto a lower-dimensional or discrete one.

Complexity Error: The error induced by restricting the possible complexity of a solution.

Structural Error: The error induced from the algorithm itself, either from a further restriction of the search space or from nonoptimality of the algorithm.

It should be noted that compression error does not apply to binary or discrete neural networks and one can reduce the compression error with finer and finer partitions of the state space, however, this adds more variables for a rule to consider, boosting the complexity error. Complexity error and structural error can in some cases be thought of as the same thing since both simply exclude a set of possible solutions based on various criteria. We make the distinction because in practice these will be two separate conditions. For example, when we search through a space of *M-of-N* rules we induce structural error by ignoring possible solutions which are not in an *M-of-N* form. At the same time we may restrict the complexity so that only sufficiently simple *M-of-N* be required. In the search procedure we will use in later sections we will penalize the complexity so that all possible *M-of-N* rules will be considered but their complexity will be taken into account when choosing our solution. From the point of view of knowledge extraction, complexity error is the issue of most

interest. Whereas compression error and structural error are properties of a chosen knowledge representation and algorithm, complexity error is a property of the network itself.

Now we will review some important rule extraction methods. Note that some of these methods will produce decision trees and decision sets instead of rule lists. Since both decision trees and decision sets are logically equivalent to rule lists we will include them here. All rule extraction algorithms can be classified by a taxonomy according to various dimensions of accuracy and explainability as well as distinguishing methods which use the behaviour of the network (Pedagogical) and the structure of the network (Decompositional) for the rule extraction process Andrews et al. [1995]. Pedagogical techniques often have the advantage of being *model-agnostic*. Because they treat the model to explain as a black box, pedagogical techniques can be applied to any model, not just a specific class of models such as neural networks. This makes pedagogical algorithms more broadly applicable but this often comes at the expense of a more detailed description of the internal reasoning used by the model. This can be a problem when using fidelity to evaluate the accuracy of a rule-extraction algorithm. Fidelity, as described in the previous section, gives a way of estimating how accurate an explanation is. However, fidelity can be misleading as it is possible for a set of rules to arrive at the same conclusion as a model without using the same reasoning. This may not be a problem depending on the application, but in many instances, for example legal cases, the actual reasoning used to come to a conclusion is as important as the accuracy of the conclusion itself. In this area decompositional techniques have an advantage over pedagogical ones as they examine the internal processes of the model in a more fine-grained manner.

The original decompositional techniques involve looking at sets of positive weights and sets of negative weights and comparing their sum to the threshold of the corresponding neuron. An early example of an algorithm using this technique is KT LiMin Fu [1994]. These early rule extraction algorithms extracted Horn clauses from a network. Horn clauses are simple in nature but often aren't flexible enough to capture the behaviour of a neural network. As described in section 3.3, *M-of-N* rules generalize Horn clauses, allowing them to more accurately represent a network with fewer parameters. Despite being able to represent a larger set of functions than Horn clauses, *M-of-N* rules still

are unable to represent every function that a single-layer neural network can. Despite this, *M-of-N* rules have been a popular choice for rule extraction due to their similarity to neural networks as well as their ease of interpretability. The use of *M-of-N* rules for rule extraction began in the context of the KBANN model Towell and Shavlik [1993]. The method used here consisted of clustering the weights and looking for combinations which exceeded the threshold of the output neuron. An early example of a Pedagogical technique is that of VIA Thrun [1994]. In VIA activation values of neurons are split into intervals which are modified for consistency.

Decision trees have also been a popular choice of structure for representing neural networks with rule-extraction. A series of several pedagogical methods for producing decision trees built on each other to create one of the most enduring and popular rule extraction algorithms. The first of which, C4.5, forms a decision tree by treating the model as a black box with input and output features and uses the information gain to calculate splitting values on the input features. These splitting values then become the nodes in a decision tree. Quinlan [1993]. Shortly after, *M-of-N* rules were used in conjunction with decision trees in ID2-of-3 and then TREPAN Murphy and Pazzani [1991], Craven [1996]. Here, information gain was once again used to construct a decision tree, but now the nodes consisted of *M-of-N* rules. *M-of-N* rules were constructed using a hill climbing search (a greedy search considering $M+1$ of N and $M+1$ of $N+1$ at each step) To find the *M-of-N* rule which maximized the information gain. The combination of *M-of-N* rules with decision trees proved to be a powerful way of representing a model, however, experiments have shown that the interpretability of the explanation is often lacking Percy et al. [2016]. More recently Two level decision sets have been used to generate both local explanations Lakkaraju et al. [2016] and global explanations Lakkaraju et al. [2017], Angelino et al. [2018] but have only been done in a model-agnostic way with no attempt to explain the internal variables of a model such as the hidden neurons in a deep network. These recent examples differ from many of the eariler techniques in that they have explicit paramters controlling the complexity of the extracted rules. CORELS Angelino et al. [2018], for instance, gives a verifiably optimal set of rules for a given set of parameters controlling complexity. It does this by performing a linear search through the set of possible rule lists (an equivalent formulation of decision sets). This linear search adds a new rule to the list at each step based on several metrics developed to measure

the optimality of the rule with regard to the given complexity parameters. Several analytic results allow the search to end once certain conditions are satisfied, meaning that the average complexity does not involve searching through the whole list of possible rules. We will discuss CORELS in more detail in a future section when we use it to benchmark our own rule extraction algorithms.

With the increased popularity of deep networks, rule extraction techniques specifically designed with deep architectures in mind are becoming of interest. The main difficulty of this task is the interpretation of the hidden layers. One way to extended simple rule extraction techniques to deep networks is to simply apply the extraction algorithm layer by layer. This was done with DIMLP, an eclectic extraction technique which restricts the weight space of a trained perceptron in order to facilitate simple rule extraction Bologna and Hayashi [2016]. DeepRED extended a previous decompositional rule extraction method, CRED Sato and Tsukimoto [2001] which used C4.5 to extract decision trees with splits on the hidden units before extracting decision trees corresponding to each of the splits resulting in a series of if-then rules relating the input and output variables. DeepRED applies this technique to deep networks by starting from the output layer and creating decision trees similar to CRED before doing the same to the preceding layer Zilke et al. [2016]. Finally, an adaption of the relationship between penalty logic and SCNs was utilized for a layerwise rule extraction technique for DBNs Tran and Garcez [2016]. This final technique will be explored in more detail when we develop our own rule extraction methods. This overview is by no means an exhaustive list of rule-extraction techniques, Many other techniques have been developed as well as almost as many review papers. Complicating this further, Rule extraction is just one possible approach to explaining neural networks, more recently *importance methods* have become popular methods of explanation along with other techniques not based on knowledge-extraction. We will give a brief overview of these alternative techniques in the following section.

5.4 Summary of Neural-Symbolic Relationships

We have now established the relationship between neural networks and logical systems as well as reviewed a number of the neural encoding and rule extraction methods that have been developed. We have shown that neural networks can be encoded into logical systems and vice-versa semantically and syntacti-

cally. Whether or not we can translate between a neural network and logical system depends on the particular network and system in question. Specifically, the capacity of a neural network to represent a logical system is determined primarily by its state space and architecture. We have also seen that classes of neural networks may be able to represent many different kinds of logic. The encodings we have described in this thesis are simply a portion of the possible encodings that have been developed or could be developed in the future. One important question in characterizing the relationship between neural networks and logical systems moving forward is to determine the *strongest* logic that a class of neural networks can represent. Although not all logical systems can be arranged in a hierarchy, some logical systems are stronger than others in the sense that one logic is representable in another. For example, first-order logic is stronger than propositional logic because you can map the knowledge bases of propositional logic to knowledge bases in first-order logic in a way that respects the deductive system. Finding hard limits to the constraints that the architecture of a class of neural networks places on its capacity to represent symbolic systems is a fundamental problem of neural-symbolic integration. Although we have not answered this question here, we catalog the encodings that have been established in this chapter and the previous one in Table 5.1. Table 5.1 contains only those relationships that have been either directly proven or referenced in the previous sections. Many other neural encodings have been developed which we have not covered in our review. Often these encodings relate a restricted form of first-order logic programming to a neural network with a very specific architecture. This makes it difficult to classify the relationship based on the topology of the network. Furthermore, the fundamental relationship between stable networks and propositional logic programming has been established making the numerous other encodings of less theoretical importance. It is likely possible to establish stronger relationships in some cases. For example, finite stable networks are probably also syntactically equivalent to penalty logic and propositional logic, continuous feed-forward networks are also presumably approximately equivalent to Horn logic programming both semantically and syntactically. Rather than directly proving every equivalence, future work should proceed by exploiting Theorem 4.4.2 and developing other theorems that can fill in the gaps.

One might notice that so far our discussion of *M-of-N* rules has been conspicuously absent from our comparison of neural networks and symbolic sys-

tems. Being that *M-of-N* rules are essentially a restriction of logic programming, most of the same theoretical relationships with feed forward networks are identical. Any acyclic logic program consisting of *M-of-N* rules *is* an acyclic logic program and any acyclic logic program can be translated into an equivalent acyclic *M-of-N* logic program by. *M-of-N* rules particularly interesting is their close relationship to neural networks, in particular feed-forward neural networks given that they both broadly fall under the category of threshold circuits. In terms of semantic or symbolic equivalence, when restricting to simple *M-of-N* rules (recall these are those rules in which N is a set of atomic propositional variables), any *M-of-N* rule can be represented by a perceptron in which the output neuron represents the head of the rule, and visible neurons represent the body of the rule. In order to encode an *M-of-N* rule in a neural network, set the bias of the output neuron to M and the weights of each input neuron to 1 or -1 for neurons corresponding to positive or negative literals, respectively. This also shows that although every *M-of-N* rule can be expressed as a propositional rule, the converse is not true as XOR and other functions cannot be implemented with a simple perceptron. One may ask whether *M-of-N* rules express *exactly* the same set of propositional formulas that a perceptron does. This, however, is not true. Consider a simple perceptron with 3 input neurons, x_1, x_2, x_3 each with 0 bias and a single output neuron, y, with bias -1. If the weights of the perceptron are $w_{1,y} = 2$, $w_{2,y} = 0.6$, and $w_{3,y} = 0.6$ then the perceptron calculates exactly the sentence $x_1 \vee (x_2 \wedge x_3)$ which cannot be expressed as an *M-of-N* rule. Thus the set of propositional sentences can be given the following hierarchy.

$$All \supset perceptrons \supset M - of - N \supset Conjunctions$$

So although *M-of-N* rules are similar to single layers of neural networks, there are functions that a single-layer neural network can compute that *M-of-N* rules cannot. We will empirically examine the impact of this discrepency in a future section by comparing extracted *M-of-N* rules to a more general set of rules produced by CORELS.

Now that we have clarified the foundations of neural-symbolic integration, we move on to the question of extracting rules from deep networks for the purpose of explaining them.

Neural Network	**Relationships to Symbolic Systems**
Binary Feed Forward with Positive Weights	Semantic and Syntactic Equivalence with Horn Logic Programming Towell and Shavlik [1994]
Finite Feed Forward	Semantic and Syntactic encodings into propositional logic 4.4
Finite Hamiltonian Feed Forward	Semantic Neural Encodings for acceptable logic programs including general, extended, modal, temporal, and epistemic logic programs d'Avila Garcez et al. [2001]
Finite Stable	Semantically equivalent to propositional logic and Penalty Logic 4.4, Pinkas [1995]
Finite Unstable	Computationally equivalent to finite automata 4.6
Continuous Feed Forward	Approximate syntactic and semantic encoding into propositional logic 4.5
Continuous Stable	Approximate semantic equivalence with propositional logic 4.5
Continuous Unstable	Not generally describable by Turing-complete symbolic systems Siegelmann [1995]
Boltzmann Machines (via MLNs)	Semantically equivalent to (restricted) first order logic programming Richardson and Domingos [2006]

Table 5.1: The relationships between different classes of neural networks and symbolic systems established in this thesis (given by reference to section number) or established previously (given by reference to relevant paper or book)

Part II

RULE EXTRACTION FROM DEEP NETWORKS

Chapter 6

Hierarchical Rule Extraction From Deep Networks

6.1 Explaining Latent Variables with Rule Extraction

In Part I we discussed some of the rule extraction techniques used to explain neural networks as well as the difficulties encountered by such methods. These difficulties have only been exacerbated by the growing use of deep neural networks LeCun et al. [2015]. Deep neural networks contain multiple layers meaning a complete rule-based explanation of a network must be hierarchical which introduces the problem of errors propagating through stages in the hierarchical rule list. That combined with the fact that neural networks generally have a large number of hidden units along with the fact that information is thought to be represented in a distributed way LeCun et al. [2015] makes the proposition of using rule extraction to fully explain a deep network dubious. For this reason, many view rule extraction methods as being inadequate for explaining deep networks Frosst and Hinton [2017]. However, the explainability of a deep network may vary according to other factors like the specific network architecture, the properties of the training set, and the learning algorithm. Of particular interest is the variability of explainability *between* the layers. Intuitively, higher layers are thought to represent more abstract concepts than lower layers which may make them more amenable to rule extraction. There is some theoretical backing for the notion that higher layers encode more abstract, and thus more compact, features of the training set. For instance, it has been pointed out that many aspects of our physical world can be described

by a compact hierarchy which allows one to reason accurately without having to observe enormous numbers of variables. This could explain the apparent effectiveness of deep learning Lin et al. [2017]. More generally the efficiency of deep architectures have been shown in circuit theory and for neural networks in the context of certain problem domains Håstad and Goldmann [1991], Eldan and Shamir [2016]. These results have been characterized as *no flattening* theorems. We know that a single layer neural network can represent any continuous function on a compact set so the question becomes can deep networks represent a function using *fewer parameters* than shallow networks. A no flattening theorem states that a shallow network will require exponentially more parameters to represent a function. What a no flattening theorem implies is that the hierarchical structure of a deep network is particularly well suited to represent a given problem. Presumably this is because the dataset itself can be represented hierarchically. Although it seems obvious that the efficiency of deep networks on a hierarchical dataset is due to the ability of deep networks to represent the inherent abstract features of the dataset in its higher layers, a direct link has yet to be made. Furthermore, the existence of adversarial attacks casts some doubt on the representational efficiency of deep networks Goodfellow et al. [2015], Szegedy et al. [2014], Su et al. [2019].

Rather than simply look at representational efficieny on a dataset, the compactness of features in a deep network have been investigated in information bottleneck framework Tishby et al. [1999]. Some have gone further and claim that the good generalization capabilities of deep net can be attributed to compression in the information plane during training, however further experiments have demonstrated that good generalization in deep networks cannot be explained by compression in every case Tishby and Zaslavsky [2015], Saxe et al. [2018]. With the case being made for more compact representations being used in deep learning for certain datasets we return to the question of rule extraction as a method of explanation. If rule extraction cannot be used as a general method for explainability in deep networks it may still be a useful tool when applied selectively. If the choice of dataset, transfer function, network architecture, and learning method can effect how much compression a network is doing then perhaps certain networks can be explained with more comprehensible rules. Even within a deep network different layers may be learning hidden representations which are explainable with rule extraction. These possibilities suggest that we should consider carefully the context in which we apply rule

extraction methods. Ideally, we want to identify the circumstances in which rule extraction methods are effective and apply them accordingly.

In order to investigate the potential of rule extraction for deep networks, we develop two *M-of-N* rule extraction algorithms. The first is a quick decompositional algorithm based on confidence rules which can be applied hierarchically to explain deep belief networks. We follow this by developing a more thorough eclectic search that allows us to empirically evaluate the explainability of hidden neurons in a given network. Using the slower thorough search on medium sized networks we identify contexts which are promising for rule extraction and then apply our decompositional algorithm in conjunction with other explainability techniques on larger networks to provide a detailed explanation of them.

6.2 Confidence Rules Revisited

The development of our fast *M-of-N* extraction algorithm will depend crucially on a previous algorithm relying on confidence rules. Confidence rules are an adaptation of penalty logic to rule extraction. In this section we give a detailed description of confidence rules and discuss several issues and limitations concerning their extraction from RBMs.

Recall that a set of confidence rules is a set of propositional rules in which each rule is assigned a positive integer called its confidence. Given an arbitrary assignment of truth values to the propositional atoms, we can use the confidence values to compute a penalty value for a set of confidence rules by calculating the sum of the confidence values for each rule which is not satisfied by the truth assignment. We then define the models of a set of confidence rules to be the truth assignments with minimal penalty values. The purpose of penalty logic is to create a logical system whose semantics are more like those of neural networks. As we have seen, in the case of symmetrically connected networks, there is a semantic equivalence between a neural network and a knowledge base in penalty logic. Pinkas [1995]

When it comes to rule extraction, the correspondence between SCNs and penalty logic makes confidence rules an attractive choice. The fact that deep networks are often pretrained layerwise with RBMs makes confidence rules

an obvious choice for rule extraction. Because RBMs are probabilistic and stable, according to 4.7.3 we can measure the fidelity of a rule by the probability that a state in the stable distribution is a model. In other words, we measure the fidelity by the probability that the state of the network satisfies the extracted rules. Unfortunately, if we restrict our extraction domain to conjunctive clauses then we can no longer guarantee that a hidden unit in an RBM will be semantically equivalent to some conjunctive rule. Furthermore, it turns out that a decompositional algorithm cannot guarantee that confidence values preserve fidelity (which is, in this case, the probability). The distinction between probability and confidence can be seen at the level of penalty logic in the following example

$$\begin{aligned} &100 : A \to \neg B \wedge \neg C \\ &3 : A \\ &2 : B \\ &2 : C \end{aligned}$$

Here although A has a higher confidence than B and C, the minimal model is $\{A \to \neg B \wedge \neg C, B, C\}$. A quick check shows that B and C are both more probable in the corresponding distribution. In this example we can think of B and C as 'red flags'. Each is evidence that A is not true but we are not confident in them enough alone to disregard A. However, when we observe both of them we conclude that A is false.

To examine the effect of this discrepancy on rule extraction, recall that in stable probabilistic networks fidelity is measured by the probability of a rule being true in the stable distribution. In other words, the fidelity of a rule is the probability that the network is in a state which the rule evaluates as true. For example given a rule $h \leftarrow x_1, x_2$ and an RBM with distribution P, $P(h \leftarrow x_1, x_2) = 1 - P(h = 0, x_1 = 1, x_2 = 1)$. Note that the probability of a biconditional rule being true corresponds exactly to the expected error of the predictions made by that rule against the network. Since rule extraction is trying to optimize the fidelity of the extracted rules to the network, the question that naturally arises with biconditional confidence rules is whether or not the confidence value associated to them corresponds to the probability of the rule. That is, does a higher confidence imply a higher probability of the rule being true? First, assuming that when we compute the expected value of the error we use the visible distribution as defined by the RBM, the

answer is no by the universal approximation theorem of RBMs Le Roux and Bengio [2008]. This is because a decompositional extraction algorithm relies only on local information, a rule involving a hidden unit h only depends on the weights of h. However since RBMs are universal approximators we can always define a new RBM with exactly the same parameters for h but with an arbitrarily chosen distribution on the visible units. This means that we can change $P(h \leftrightarrow x_1, ..., x_k, \neg x_{k+1}, ..., \neg x_n)$ a great deal without changing the confidence of the rule. This shows that *any* assignment of confidence values to RBMs that depends only on local information will not preserve the order of the probabilities. This fact is summarized in the following proposition.

Proposition 6.2.1. *Given an RBM, N, any local decompositional rule extraction algorithm cannot assign confidence values, c, to rules in a way such that for rules R_1 and R_2, $c_{R_1} < c_{R_2} \implies P(R_1) < P(R_2)$.*

Proof. See appendix □

The relationship between RBMs, penalty logic, and rule extraction can be complicated so we will summarize it here. Given an SCN there is a corresponding knowledge base in penalty logic In a deterministic network (Hopfield network) the minimal models of this knowledge base are the stable states and in a probabilistic network (Boltzmann Machine) they are the states with highest probability. It is easy to see that a statement in the knowledge base with a high confidence will not necessarily have a high probability in the Boltzmann Machine because it may be contradicted by many other statements of low confidence. When applying rule extraction we search for a set of conjunctive clauses which maximizes the fidelity of the corresponding logic program to the network. The set of clauses we extract, however, will not necessarily be the statements in the corresponding knowledge base because we will only look for rules rather than arbitrary propositional statements. Furthermore, if we are using a decompositional method, then no matter how we assign confidence it will tell us nothing about the probability and hence the fidelity of the extracted rule.

Now that we've clarified the relationship between confidence and fidelity let's examine in detail a rule extraction algorithm for RBMs which uses confidence values.

6.3 Extracting Confidence Rules from RBMs

Here we will outline a method for extracting confidence rules from RBMs. The method is decompositional so in theory it can be applied to any neural network but it was specifically developed for RBMs Tran and Garcez [2016]. We will refer to this method as the *optimal confidence algorithm*.

The algorithm will be applied to explain the hidden units of an RBM. This means that in the extracted rules the head of each rule corresponds to the a hidden unit and the literals in the body of the rule correspond to the visible units. Since the neurons in an RBM take on binary values, we define the atomic variable X_i by $X_i = \top$ iff $x_i = 1$ and $X_i = \bot$ iff $x_i = 0$. Here x_i is a neuron in the RBM. Since the neurons in an RBM are partitioned into hidden and visible we will use the variables $\{h_j\}_{j=1}^k$ to denote the hidden neurons and the variables $\{x_i\}_{i=1}^k$ to denote the visible neurons. Our conjunctive rules will then be of the form

$$h_j \leftrightarrow x_{i_1} \wedge x_{i_2} \wedge ... \wedge \neg x_{i_k} \wedge ... \wedge \neg x_{i_l}$$

Since the RBM is a symmetric network, whatever effect a visible unit has on a hidden unit must be equal to the effect of the hidden unit on the visible unit. For this reason the rules that are extracted are considered biconditionals rather than one-way implications. That means if the body of the extracted rule for a literal H is not satisfied then we predict $H = \bot$. In other words, if we fail to prove H with our extracted rule we assume H is false. Because the rules extracted are hierarchical, the semantic difficulties of negation in logic programming are not an issue.

Given an RBM with n visible units and m hidden units we can represent a set of conjunctive rules modeling the hidden units with a matrix $S \in \{0, 1\}^{n \times m}$ defined by $S(i, j) = 1$ if X_i is in the body of the rule for H_j, $S(i, j) = -1$ if $\neg X_i$ is in the body of the rule corresponding to H_j, and $S(i, j) = 0$ otherwise. In order to generate a rule for each hidden unit, the optimal confidence algorithm finds what is essentially an optimal cluster of visible weights by minimizing the following loss function

$$I_{loss} = \sum_{i,j} \frac{1}{2} |w_{i,j} - c_j S(i, j)|^2 \tag{6.1}$$

The algorithm works iteratively, first, given a conjunctive rule, we calculate the optimal value of the confidence that minimizes the loss function. Then we

Algorithm 1 Extracting conjunctive confidence rules from an RBM

Input: An RBM, $\mathcal{N}$

for Each hidden neuron $h_j \in \mathcal{N}$ **do**

for all input neurons x_i to h_j, set $S(i,j) = sign(w_{i,j})$

$c_j = \frac{\sum_i w_{i,j} S(i,j)}{\sum_i S(i,j)^2}$

$I_{loss}^{new} = \sum_{i,j} \frac{1}{2} |w_{i,j} - c_j S(i,j)|^2$

$I_{loss} = \infty$

while $I_{loss}^{new} < I_{loss}$ **do**

$I_{loss} = I_{loss}^{new}$

for all input neurons x_i to h_j **do**

if $c_j \geq 2|w_{i,j}|$ **then**

$S(i,j) = 0$

end if

end for

$c_j = \frac{\sum_i w_{i,j} S(i,j)}{\sum_i S(i,j)^2}$

Set $I_{loss}^{new} = \sum_{i,j} \frac{1}{2} |w_{i,j} - c_j S(i,j)|^2$

end while

end for

return Matrix S representing a set of rules

prune literals from the rule if doing so will lower the loss function. In detail, the algorithm begins by setting $S(i,j) = 1$ if $w_{i,j} \geq 0$ and $S(i,j) = -1$ if $w_{i,j} < 0$. Then we set each confidence to the value that minimizes I_{loss}. It is not hard to see that this is

$$c_j = \frac{\sum_i w_{i,j} S(i,j)}{\sum_i S(i,j)^2} \tag{6.2}$$

From there we want to set $S(i,j) = 0$ if it will decrease (7.1), this is true iff $c_j \geq 2|w_{i,j}|$. We then repeat this process until convergence. As an example, imagine a neuron, x_4, with inputs x_1, x_2, x_3 and associated weights $w_{1,4} = 2, w_{2,4} = -1, w_{3,4} = -0.5$. S is initialized to $S(1,4) = 1$, $S(2,4) = S(3,4) = -1$. Using 6.2 we calculate $c_4 = \frac{3.5}{3}$. Because $c_4 \geq 2|w_{2,4}|$, $S(3,4)$ becomes 0 while $S(1,4)$ and $S(2,4)$ are unchanged. Recalculating 6.2 we get $c_4 = \frac{3}{2}$. Because atoms are only removed if doing so reduces the information loss, we know that there is at least one more iteration to do. Now we compare c_4 with $2|w_{i,4}|$ and observer that it is smaller for both $i = 1$ and $i = 2$. $S(i,4)$

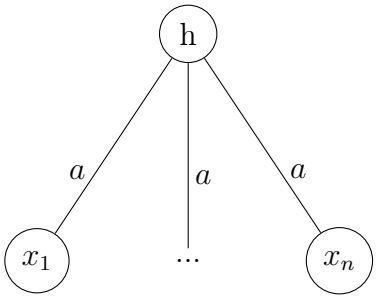


Figure 6.1: An RBM with a single hidden unit and n identical visible units

remains unchanged and thus the algorithm has converged giving us the rule $X_4 \leftrightarrow X_1 \wedge \neg X_2$.

In the previous section we saw that any decompositional knowledge extraction algorithm cannot produce confidence rules which will always correspond to probability. This observation came from the fact that the visible distribution could be changed arbitrarily without changing the confidence rule produced by the algorithm. What if we clamp the visible distribution? For example, in DBNs, when performing inference we draw from a given input distribution and propagate up through the network. Does the optimal confidence algorithm preserve probabilities in this case? A simple example shows that it doesn't. Take a network with one hidden unit and n visible units all connected to the hidden unit with weight a. All biases are 0 as shown in figure 6.1. The optimal confidence algorithm will extract the rule $a : h \leftrightarrow x_1, ..., x_n$. But as a increases to infinity, the probability of any configuration of the visible units other than all 0s activating h goes to 1. But when $h = 1$ the rule is only true if all the visible units are activated. Since we're assuming an independent and uniform distribution for the visible units this implies that by taking $n \to \infty$ the probability of the biconditional goes to 0 regardless of a. In fact we can make a arbitrarily large and still get the same result showing that we can have arbitrarily large confidence values corresponding to biconditionals with arbitrarily low probability. In other words, we have the following

Proposition 6.3.1. *For all $0 < \epsilon < 1$ and all $c > 0$, there exists an RBM such that the optimal confidence algorithm extracts a rule R with $P(R) < \epsilon$ and $c_R > c$*

Proof. See appendix □

This proposition shows that conjunctive rules are too restrictive. In particular,

rules with single conjunctions can be prone to false negatives. That is, if we extract a rule $h \leftarrow x_1, x_2$ then we will predict $h = 1$ only when $x_1 = x_2 = 1$. However there may be many other configurations in which we want to predict $h = 1$. We can either add more single implication rules to cover additional cases or we can hope to find a biconditional rule that accurately captures the behaviour of the network. The above proposition shows that the latter approach can result in rules with arbitrarily low fidelity. This motivates the use of a less restrictive paradigm. This is where we bring in *M-of-N* rules. *M-of-N* rules are not as prone to false negatives as conjunctive clauses but they do not require us to search through (potentially large) arbitrary sets of rules. We will use the optimal confidence algorithm to define a new algorithm which extracts *M-of-N* rules instead of purely conjunctive ones.

6.4 Extracting *M-of-N* Confidence Rules

Now we are finally prepared to define our first *M-of-N* extraction rule. To do this, first note that a conjunctive rule can be turned into an *M-of-N* rule simply by choosing a value for M. For example, a confidence rule $h \leftrightarrow x_1, x_2, \neg x_3$ can be turned into a confidence rule $h \leftrightarrow 1 - of - \{x_1, x_2, \neg x_3\}$. With this in mind, we develop a decompositional algorithm for finding *M-of-N* rules by first using the optimal confidence algorithm to find a set of confidence rules, and then converting these rules into *M-of-N* rules by choosing a suitable value of M for each rule. All that is required to do this is a suitable way of choosing M. In the original *M-of-N* extraction algorithm, M was chosen so that the input coming into the neuron from M of the literals is greater than the neurons threshold Towell [1991]. We use a similar method for converting confidence rules to *M-of-N* rules.

For each confidence rule $c_j : h_j \leftrightarrow x_1, ..., x_l, \neg x_{l+1}, ..., \neg x_k$, we choose a value M_j by setting M_j to be the minimum value so that $M_j \cdot c_j \geq T_j$ where T_j is some threshold. Consider the minimum possible input to h_j, this is just $I_{j,min} := b_j + \sum_{i:w_{i,j}<0} w_{i,j}$. In general, given some arbitrary threshold, T, we set M_j to be the minimum value such that $M_j \cdot c_j + I_{j,min} \geq T$ which gives us that $M_j \cdot c_j \geq T - I_{J,min}$ so $T_j = T - I_{J,min}$. In the case of the RBM for example, since $P(h_j = 1|x) > 0.5$ if the total input to h_j is greater than 0 we should predict 1 if $M_j \cdot c_j + I_{j,min} \geq 0$. In other words, $T_j = -I_{j,min}$ so we set M_j to the minimum value with $M_j \cdot c_j \geq -I_{j,min}$. For the case that no value of M_j

Algorithm 2 Extracting *M-of-N* rules from a network using a set of conjunctive rules with confidence values as a starting point

Input: An RBM, $\mathcal{N}$

Run Algorithm 1 on $\mathcal{N}$ to get an initial set of rules R represented by matrix S

for Each hidden neuron $h_j \in \mathcal{N}$ **do**

$M_j = N + 1$

$W_{j,to\_add} = \{w_{i,j} : S(i,j) = 0\}$

while $W_{j,to\_add} \neq \emptyset$ **and**$\{M : M \cdot c_j \geq -b_j - \sum_{i:w_{i,j}<0} w_{i,j}\} = \emptyset$ **do**

$k =$arg max$\{|w_{\cdot,j}| : S(i,j) = 0\}$

$S(k,j) = sign(w_{k,j})$

$W_{j,to\_add} = W_{j,to\_add} \setminus w_{k,j}$

Update c_j according to 6.2

end while

if $\{M : M \cdot c_j \geq -b_j - \sum_{i:w_{i,j}<0} w_{i,j}\} \neq \emptyset$ **then**

$M_j = \min\{M : M \cdot c_j \geq -b_j - \sum_{i:w_{i,j}<0} w_{i,j}\}$;

end if

end for

return Set of rules R, represented by matrix S and integers M_j

can exceed T_j we proceed by attempting to add a literal to the rule and recalculating c_j according to equation 6.2, we choose the literal to add to be the one corresponding to the neuron with the highest absolute weight in the set of literals not in the rule. If there is nothing to add and we still cannot exceed T_j then we output the rule $N+1$ of N, in other words, the rule which always outputs 0. To illustrate we consider a simple example. Take an RBM with a single hidden unit and two visible units with weights $w_{1,h} = 1, w_{2,h} = -1, h_b = 0$. Then minimizing 6.1 gives us the rule $h \leftarrow x_1 \wedge \neg x_2$ with confidence $c = 1$. Then we set the threshold $T = 1$ and since $1 \cdot 1 \geq 1$ we set $M = 1$ giving us the rule 1 of $(x_1, \neg x_2)$. The procedure is summarized above.

Intuitively, the algorithm begins by using the optimal confidence algorithm to generate confidence rules. From there, we choose M to be the smallest value so that if M of the literals in the confidence rule are satisfied then the minimum possible input to the hidden neuron is positive. In some cases even if all of the literals in the confidence rule are below 0 the minimum input to the

hidden neuron is below 0. When this happens we try to amend the confidence rule by adding the next most relevant literal, that is the one corresponding to the neuron whose weight has the greatest magnitude, recalculating the confidence value, and trying again. We proceed until we either find an *M-of-N* rule which guarantees a non-negative input or we run out of literals to add. In the second case we output the 'impossible to satisfy' rule, $N+1$ of N indicating that our rule should always predict that the hidden unit outputs 0 since in this case, regardless of the input configuration, there is a higher probability of the output being 0.

To illustrate, take the example from the previous section: x_1, x_2, x_3, x_4 with $w_{1,4} = 2, w_{2,4} = -1, w_{3,4} = -0.5$. Suppose x_4 has 0 bias. Running the optimal confidence algorithm produces the rule $X_4 \leftrightarrow X_1 \wedge \neg X_2$ with confidence $\frac{3}{4}$. The minimum possible input to x_4 is -1.5. $1 \cdot \frac{3}{4} \geq 1.5$ but $0 \cdot \frac{3}{4} < 1.5$ so $M = 1$ is the minimum value of M that satisfies the necessary condition. The resulting rule is thus $x_4 \leftrightarrow 1 - of - \{X_1, \neg X_2\}$.

The algorithm, given a set of literals, chooses M in what one might call a pessimistic way. The relevant information for the algorithm is if the rule is satisfied, is the smallest possible input to the hidden node positive or negative? If the answer is negative then the rule being satisfied is not good enough to ensure we have a greater than 50% chance of being correct. The algorithm chooses the minimum value of M so that in the worst case scenario we still have a greater than 50% chance or being right. The optimistic approach would assume that all positive literals are on. In this case, unless the bias of the hidden node is negative and larger than all positive weights the 1 of N (or even 0 of N) rule would be sufficient so this is not a useful threshold. We could also have used an average case scenario where we choose some combination of positive and negative weights but this didn't seem to work as well experimentally although further trials could be done. The worst-case approach we use is essentially trying to decrease the chance of false positives by ensuring that the rule *only* outputs true if it is guaranteed that there is an over 50% chance of the corresponding hidden unit being on. The full conjunctive rule will minimize the chance of false positives but at the expense of increasing the chance of false negatives as shown in the proof of Proposition 6.3.1. By choosing the *minimum* M that achieves this we lower the chance of false negatives while still keeping the chance of false positives low. To summarize, the current

approach to choosing M is to conservatively minimize false positives so that we don't increase false negatives too much.

Note that we are technically not looking at the worst case scenario by considering $M \cdot c_j$ as the positive input from the rule. c_j is the average weight of the literals in the rule but in the worst case scenario the literals with the smallest weights would be the ones satisfied. For example take a rule $h_j \leftrightarrow 2 - of - \{x_1, x_2, x_3\}$ with $w_{1,j} = 2, w_{2,j} = 3, w_{3,j} = 1$. Then $c_j = 2$ so our algorithm would use $I_{j,min} + 2 \cdot 2 = I_{j,min} + 4$ as the minimum input but in reality the configuration $x_1 = x_3 = 1, x_2 = 0$ satisfies the rule and the minimum input is $I_{min,j} + 1 + 2 = I_{j,min} + 3$. Modifying the algorithm to use the actual minimum instead of the approximate minimum is possible but in this case every M has to be tested whereas we can explicitly solve for M using $M \cdot c_j$ as an approximation. It is possible to implement this approach without significant time penalty by simply keeping a list of indices for weight magnitudes of the literals in the rule. Since there are N possible literals and we can check the minimum for each M with a single operation this addition is asymptotically linear. Despite this we continue to use the confidence value approximation for the time being due to its convenience.

In order to evaluate this algorithm, we compared it to the original optimal confidence algorithm as well as a Hillclimbing procedure. Hillclimbing starts with an initial rule $1 - of - \{x_i\}$ where x_i is the visible unit with the strongest connection to h_j. If x_i has a negative connection to h_j then the initial rule is $1 - of - \{\neg x_i\}$. From there the algorithm increases M and N by either adding the literal with the next highest weight (positive or negative depending on the sign of the weight), adding the literal with the next highest weight and increasing M by one, or remaining with the same rule. It does this by considering the expected error, whichever option has the lowest expected error is chosen. If the current rule is the best option then the algorithm outputs that rule and moves on to generate the next rule. Hillclimbing is optimal for certain tasks, in otherwords it outputs the rule with the smallest error, as we will see, however, this is not the case for generating *M-of-N* rules. Unfortunately, for larger tasks the expected error is intractable and hillclimbing becomes essentially impossible.

To evaluate the fidelity we calculated the expected value of the error of the rules

with respect to the network. Recall that an RBM converges to a stationary distribution and in that case the fidelity can be measured by the probability of the rule being true in the stationary distribution. Because we extract biconditionals, given some input x, the probability of a rule being false is $P(h_i = 1|x)|1 - r(x)| + P(h_i = 0|x)$ where P is the Boltzmann distribution of the Network. For simplicity we assume a clamped visible distribution, then if there are 2^l inputs we have $P(x) = \frac{1}{2^l}$ which implies that the probability of a rule being false is

$$\frac{1}{2^l} \sum_{x \in X} (p(h_i = 1|x)|1 - r(x)| + P(h_i = 0|x)|r(x)|) \tag{6.3}$$

Now given a single rule for each hidden unit, the probability of not satisfying the entire set of rules would be the product of each conditional probability. However, in many cases rule extraction may be applied selectively, that is we only want to extract rules for a few hidden units. For this reason, we simply give the average error over each hidden unit which gives us a final error measure of

$$\frac{1}{2^l} \frac{1}{m} \sum_i \sum_{x \in X} (p(h_i = 1|x)|1 - r(x)| + P(h_i = 0|x)|r(x)|) \tag{6.4}$$

We tested the three different rule extraction algorithms on 12 different RBMs each trained on a different small dataset generated using logical functions. The datasets were generated by taking every possible combination of the first $n - 1$ visible units and setting the n^{th} visible unit to the output of the first $n - 1$ visible units with the function. For example the XOR (3 vis) dataset has the first two variables free and the third calculated as x_1 XOR x_2 giving us $(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0)$. Since the datasets are small, we are able to use the expected error as mentioned in previous sections. This is just the expected value of the difference between the output of our rules and the output of the network for a single hidden unit, averaged over all hidden units. So in a network with l visible units and m hidden units the error is

$$\frac{1}{2^l} \frac{1}{m} \sum_i \sum_{x \in X} (p(h_i = 1|x)|1 - r(x)| + P(h_i = 0|x)|r(x)|) \tag{6.5}$$

Here X is the space of visible configurations, $r(x)$ is the output of our rule on input x. The maximum error possible is 1 and the minimum is 0. Again note that we are comparing deterministic rules to a probabilistic network which is why we look at the error as a random variable dependent on the output of the network, also note that the *M-of-N* rules generated also have associated

confidence values which we can take into consideration when looking over the set of generated rules. We also report the average value of M and N in order to gain some insight into how interpretable these rules might be, if N scales linearly with the size of the input data then large datasets will give us totally incomprehensible rules.

Here we tested three different rule extraction algorithms on small RBMs trained on datasets constructed from Boolean functions. We tested the original optimal confidence algorithm along with the *M-of-N* optimal confidence algorithm and with the expected error.

Each Network was trained for 10000 iteration using contrastive divergence with 10 Gibbs steps. It is clear that of the three methods tested, overall the *M-of-N* optimal confidence algorithm gives the best trade off for time and accuracy. As expected the most accurate method is the hill climb using the expected error but this sums over all visible states and is thus exponential. We see that the time it takes grows quite quickly whereas the time it takes the *M-of-N* optimal confidence algorithm to finish grows relatively slowly. In addition the expected error for the *M-of-N* optimal confidence algorithm is more or less in line with hill climbing, in some cases even beating it, which implies that hill climbing is not optimal for finding the minimum expected error.

Speed wise the optimal confidence algorithm is the fastest with the *M-of-N* optimal confidence algorithm being only slightly slower. Also interesting things to note is that for all algorithms the expected error is relatively consistent as the dataset increasing in size. This is encouraging. Finally notice that M and N appear to increase slower than the dimension of the test set in all cases although they grow slowest with hill climbing. For larger datasets the expected error will be intractable and instead we will have to rely on test error over a sample set to evaluate the accuracy.

	hcee			optmial confidence		
Network	**M/N**	**Error**	**Time(s)**	**M/N**	**Error**	**Time(s)**
XOR(3 vis 10 hid)	2.1/2.7	0.0991	0.322	2.7/2.7	0.2279	0.09
XOR(6 vis 20 hid)	2.65/3.4	0.1095	7.819	4.55/4.55	0.3344	0.014
XOR(9 vis 30 hid)	3.36/4.03	0.1035	89.158	5.8/5.8	0.2766	0.036
NAND(3 vis 10 hid)	1.4/1.4	0.1277	0.222	2.6/2.6	0.2302	0.007
NAND(6 vis 20 hid)	2.35/2.65	0.1206	5.248	3.6/3.6	0.2540	0.022
NAND(9 vis 30 hid)	2.93/3.3	0.1227	74.511	5.4/5.4	0.2723	0.034
OR(3 vis 10 hid)	1.6/2.1	0.1424	0.095	2.4/2.4	0.2328	0.004
OR(6 vis 20 hid)	2.4/2.8	0.1132	5.429	3.6/3.6	0.2665	0.021
OR(9 vis 30 hid)	2.53/3.26	0.1369	74.209	5.13/5.13	0.3614	0.036
AND(3 vis 10 hid)	2.0/2.2	0.0884	0.249	2.4/2.4	0.1735	0.007
AND(6 vis 20 hid)	2.5/2.85	0.1068	3.584	3.5/3.5	0.2439	0.021
AND(9 vis 30 hid)	3.26/3.93	0.1052	87.367	5.4/5.4	0.2819	0.035

MofN optimal confidence

Network	**M/N**	**Error**	**Time(s)**
XOR(3 vis 10 hid)	2.4/2.8	0.1149	0.009
XOR(6 vis 20 hid)	3.35/4.75	0.1067	0.023
XOR(9 vis 30 hid)	4.86/6.06	0.1234	0.043
NAND(3 vis 10 hid)	2.3/2.8	0.1562	0.011
NAND(6 vis 20 hid)	3.3/3.9	0.1312	0.026
NAND(9 vis 20 hid)	4.53/5.56	0.1300	0.039
OR(3 vis 10 hid)	1.8/2.5	0.1614	0.004
OR(6 vis 20 hid)	3.1/3.75	0.1223	0.024
OR(9 vis 30 hid)	3.83/5.2	0.1440	0.038
AND(3 vis 10 hid)	2.2/2.4	0.1042	0.007
AND(6 vis 20 hid)	3.4/4.05	0.1125	0.029
AND(9 vis 30 hid)	4.6/5.6	0.1240	0.040

Table 6.1: Comparing hill climbing with expected error(hcee) to the optimal confidence algorithm presented in Tran and Garcez [2016] and the *M-of-N* version of the optimal confidence algorithm. Error is the average error of the extracted rules, M/N is the average value of M and N of the extracted rules, and time is the time for the extraction algorithm to complete.

Chapter 7

Characterizing the Accuracy/Interpretability Landscape for *M-of-N* Explanations of Latent Variables

7.1 The Accuracy/Interpretability Tradeoff

As discussed previously, knowledge extraction methods must balance between accuracy and interpretability. Although some trade off may be necessary when attempting to distill a network into a comprehensible knowledge base, the extent to which accuracy is constrained by interpretability in real world neural networks has not been studied in detail. Indeed, in most cases, given a rule produced by some extraction algorithm, we do not know how much the accuracy of the rule was limited by the structural error of the algorithm versus how much of the accuracy was limited by the complexity of the problem. In other words, it is impossible, or at least difficult, to tell whether or not we could find a better rule using a different algorithm. In some cases, a rule extraction algorithm will produce rules which are both satisfactorily accurate and interpretable. However, in other cases extraction algorithms will produce rules which are either inaccurate or too complex to be interpretable. The lack of a satisfactory rule based explanation may be a failure of the particular algorithm or it could be that one doesn't exist at all.

Even when satisfactory rules are produced it is unclear whether or not they

are *optimal* in terms of their accuracy/interpretability trade off. We are often left with the question, can we find a simpler explanation for this model? To answer this question, we consider a brute force approach. By testing every possible rule in a set of candidate rules, we can be sure that the resulting rule is a good indicator of the complexity inherent in the network rather than a product of the specific search procedure. The reason a brute force approach has been avoided in the past is because of its computational difficulty. The total number of possible rules that can be assigned to a neuron grows exponentially with the number of inputs making a complete search infeasible for any but the smallest networks. To get around this, we do not attempt a full brute force search of the solution space, instead we make several simplifying assumptions that reduce the search space to a manageable size. The first assumption is the restriction to *M-of-N* rules. The intuitive reasons for using *M-of-N* rules to describe neural networks has been explained in detail, but the fact remains that a single layer in a neural network can implement a larger class of functions than an *M-of-N* rule. For this reason we will still need to empirically justify the restriction to *M-of-N* rules. The second assumption is a specific ordering of the literals. In particular a weight ordering. What we mean by this is that any *M-of-N* rule that we consider will contain only the first N literals in our ordering. If X_1 is a literal that comes before X_2 in our ordering then any rule which contains X_2 as a literal but not X_1 will be excluded from our search. This restriction makes our search space polynomial in size which allows us to conduct experiments on medium sized networks. This restriction is analytically justifiable in some cases but remains a heuristic approach in the most general case.

By reducing the search space and testing every solution in the space we can guarantee that we find an optimal rule among the candidate rules, but we cannot be certain that the truly optimal rule is even among the candidate rules. The partial analytic justification for the weight-ordering may give us some comfort, but the choice of *M-of-N* rules has no analytic justification in any context. In order for us to be confident that our search procedure is indeed finding rules that are at least close to optimal, we will need to test it empirically against a search procedure that is optimal. Luckily, CORELS is such an algorithm. CORELS is a sequential extraction process that produces optimal rules (see 5.3 for a more thorough description). Unsurprisingly, CORELS is too slow to apply to most networks, but by comparing our slow *M-of-N* approach

to CORELS on small networks we can verify that rules in our simplified test set are at least close to optimal. This will allow us to interpret experimental results from our algorithm as evidence, although not absolute proof, of the necessary complexity of a rule to describe a network.

Even with these simplifications, our approach can be slow. While medium-sized networks may have hundreds or thousands of neurons, many state of the art networks in domains such as image recognition have hundreds of thousands of neurons. When this is the case, our algorithm will be computationally infeasible and faster algorithms that rely on more heuristics must be turned to. The goal of our experiments will not be to develop a general purpose extraction algorithm that can be used to explain any network. Rather, we wish to see to what degree rule extraction is limited by the inherent complexity of neural networks. Recalling our discussion in 6.1, the complexity of features used by a network is likely the result of several factors and the complexity between layer of a deep network may vary as well. Applying the search to different layers of number of different networks which differ in crucial ways (dataset, size, learning algorithm, etc.), we can identify conditions in which simple and accurate rules may be extracted. When looking to explain larger networks for which our approach method is not possible, we can use the information about the inherent complexity of extracted rules to decide whether or not rule extraction is worth pursuing at all. When we have reason to believe that a large network may be explainable with rule extraction (either entirely or in part), then we can employ fast decompositional techniques such as the *M-of-N* optimal confidence algorithm developed in the previous section. These techniques may not be optimal, but we know there is at least the possibility of finding an interpretable rule.

When we describe rules as optimal, it is important to define what it means for a rule to be optimal. Complexity and accuracy are competing objectives. One rule may be highly accurate but highly complex while another rule is simple but inaccurate. Which, then, is more optimal? Optimality in this context depends crucially on a desired tradeoff. We can quantify this desired tradeoff with a parameter $\beta \in \mathbb{R}^+$ which weights the relative importance of accuracy and interpretability. Given a measures of accuracy, A, and interpretability, I, we can measure how well a rule, R, explains a model with the equation

$$A(R) + \beta I(R) \tag{7.1}$$

Given a set of rules $\mathbf{R}$, a rule, $R_0 \in \mathbf{R}$ is optimal with respect to a given β if it maximizes the previous equation taken over $\mathbf{R}$. If we set $\mathbf{R}$ to be the set of every possible rule then the inherent complexity of the model is captured by the relationship between β and $A(R_0)$ for an optimal R_0. For the special case of $\beta = 0$ we search for the most accurate rule in our set of candidate rules $\mathbf{R}$. In the previous sections we established that, in a feed forward network, if $\mathbf{R}$ is the set of all possible rules then the structural error of an optimal extraction algorithm over $\mathbf{R}$ is 0, although there will still be some degree of compression error if the network is continuous. As β increases the accuracy of our optimal rule decreases. In a model that is explainable with rule extraction, the accuracy should decrease only a little for relatively large values of β. In an unexplainable model accuracy will decrease very rapidly at relatively small values of β.

In addition to looking at specific accuracy and interpretability values, the shape of graph of the relation between interpretability and accuracy is of interest. By plotting $A(R_0)$ against $I(R_0)$ starting with $\beta = 0$ and increasing β we can determine the exact relationship between accuracy and interpretability. If the graph is relatively linear, then picking a desired tradeoff becomes difficult as you can always sacrifice a little bit of interpretability for a little bit of accuracy or vice-versa. If, on the other hand, the graph is exponential, then we have a 'natural' choice for our rules. It is still possible that our model is unexplainable as the 'natural' choice may have a low interpretability.

Before we get into the details of our search method, we must first define ways of measuring accuracy and interpretability. We have already developed a way of measuring the accuracy, namely the fidelity (see 4.2.10). Unfortunately measuring the fidelity over the entire input space can be difficult even in the simplest case. In the more general case, the input comes from some unknown manifold making an explicit measure of the fidelity impossible. Rather, we will use an approximation of the fidelity based on a number of selected samples. Furthermore, for convenience we will use the *error* rather than the fidelity which is simply defined as $1 - fidelity$. We will discuss explicitly how the error is calculated in the next section but intuitively it can be thought of as the percentage of disagreements a rule set has with the network over some set of test inputs. In order to measure complexity we will use the notion of minimum description length. The minimum description length is a robust measure

of complexity that, as the name implies, judges complexity by the length of sequences minimum description. We will fix the language as propositional formulas expressed in DNF and measure the complexity based on the length of this formula. Although fidelity and error are obvious choices for measuring accuracy, the relationship between complexity and interpretability is more subtle. It is true that very complex rules are not interpretable, but it isn't always true that simple rules are. The interpretability of a rule is highly dependent on the context and the audience. Although complexity provides a rough estimation of interpretability, whether or not a set of rules is understandable for a specific audience ultimately has to be empirically justified for a given context. Before elucidating the technical details of error and complexity, we should note that these measures are inverses of accuracy and interpretability. The lower the complexity the higher the interpretability and the lower the error the higher the accuracy. The only thing this changes in practice is that we search for rules which minimize equation 7.1 instead of rules which maximize it.

7.2 Measuring Accuracy and Interpretability

Now that we have discussed the important metrics for rule extraction, we are ready to define them formally. The two metrics we are concerned with in rule extraction are accuracy and interpretability. For our experiments these will be measured with error and complexity respectively. In order to understand how the error is measured, recall that according to our definition of fidelity, the error is the percentage of initial states of a network that are *not* models for the extracted rules. In our case, we are extracting semantic symbolic encodings from feed forward networks. In other words, from a neuron h with inputs x_i, we extract a rule $H \leftarrow X_{r_1} \wedge X_{r_2} \wedge ... \wedge \neg X_{r_k} \wedge ... \wedge \neg X_{r_l}$. Because the network is feed forward, an initial state, $x_{r_1}, ..., x_{r_l}$ will be a model of the rule if either the corresponding $X_{r_1} \wedge X_{r_2} \wedge ... \wedge \neg X_{r_k} \wedge ... \wedge \neg X_{r_l}$ is false or, in the next time step, the value of h maps H to true. Strictly speaking then, the error of this single rule is percentage of inputs over the input space that result in a false positive. These are inputs for which the rule predicts H is true but the dynamics of the network do not reflect this. In practice, we wish to count the false negatives as well. A rule whose body is never satisfied will make no false positive predictions but is of little use for describing the network. Therefore, rather than extract unidirectional implications, all of our rules will be bidirectional. When the rule is bidirectional, if the body

is not satisfied but the head is, then the configuration is not a model of the rule meaning error measures both false negatives and false positives. In the discrete case, the number of input examples is exponential in the number of input neurons, making the exact calculation of error intractable. In the continuous case, although we may be able to integrate over the input space, we are often dealing with inputs taken from an unknown manifold (consider, for example, normalized 32×32 grayscale images, the total input space is $\mathbb{R}^{992}$ but the natural images we are interested in only consist of a small fraction of these). For this reason we approximate the error by sampling the input space and calculating the error on this set of samples. Putting this all together, the process for calculating the error of a rule is as follows. Given a neuron h in a neural network with input neurons x_i, we use the network to compute the state of h from the state of the input neurons which then determines the truth of the propositional variable H. Thus we can use the network to determine the truth of H, call this $N(x)$. Furthermore, if we have some rule R relating variables H and X_i, we can use the state of the input x to determine the value of the variables X_i, and then use R to determine the value of H, call this $R(x)$. Given a test set of input configurations I we can measure the discrepancy between the output of the rules and the network as

$$E(R) := \frac{1}{|I|} \sum_{x \in I} [R(x), N(x)] \tag{7.2}$$

where $[R(x), N(x)]$ is a function returning 0 if $R(x) = N(x)$ and 1 otherwise. Because we only ever deal with binary variables, if we quantify the truth values of H to 1 and 0 corresponding to $\top$ and $\bot$ respectively then this term becomes $|R(x) - N(x)|$. This provides an approximation of the error for a single rule, what about an entire rule set? If we extract a rule for every neuron in a deep network, then by our definition of fidelity, if a single rule makes an incorrect prediction then the initial state is not a model of our extracted rules. This means that we could conceivably extract highly accurate rules for each neuron, but that, taken as a whole, the extracted rules have very low fidelity because the large number of neurons implies that there is likely to be at least one mistake somewhere in the network. Furthermore, when experimenting with larger networks, we do not always extract rules from every neuron but rather a sample of them. For these reasons we take the more pragmatic approach of evaluating each rule independently and averaging the results to give a sense of how explainable the average neuron is. This will pay off when we develop the notion of modular explainability methods in which rule extraction is only

applied to especially explainable neurons.

Moving on to our interpretability measure we will discuss complexity. Here, we think of complexity in an analogous way to the Kolmogorov complexity which is determined by a minimal description. Using this idea we calculate the complexity of a rule by the length of its body when expressed in a minimal disjunctive normal form (DNF). In analogy to Kolmogorov complexity, DNFs in propositional logic with the standard syntax acts as the reference language with equivalent formulas being different DNF representations of the same model. The choice of restricting our attention to DNFs when calculating the complexity may raise objections as in practice this means some rules could conceivably have simpler representations than other rules that are measured as less complex, however calculating the minimal complexity over an arbitrary number of possible representations is impractical if not infeasible. Furthermore, because *M-of-N* rules all share the same form, any further reduction in complexity would apply to all of them. Using this definition, the complexity of an *M-of-N* rule is $M\binom{N}{M}$ (where $\binom{}{}$ denotes the binomial coefficient). To see this, consider each minimal model satisfying an *M-of-N* rule. This is a configuration in which exactly M of the N literals are true. For a specific collection of M of the N literals, the conjunction of these M literals will have as models all configuration in which at least the selected M literals are true. There will be $\binom{N}{M}$ such configurations. Thus we can express an *M-of-N* rule as a disjunction of $\binom{N}{M}$ terms each of which is a conjunction of M literals giving a total length of $M\binom{N}{M}$. This will capture every possible model of our *M-of-N* rule and no other. Furthermore, since each of these terms will have at least one term that differs from every other term, this is a minimal DNF.

We also measure complexity in a relative manner by normalizing w.r.t. a maximum complexity. Given N possible input variables the maximum complexity of an *M-of-N* rule is $\lceil\frac{N+1}{2}\rceil\binom{N}{\lceil\frac{N+1}{2}\rceil}$, where $\lceil\rceil$ denotes the ceiling function (rounding to the next highest integer). This is because the error is between 0 and 1. In principle it should not matter whether or not we normalize because this is equivalent to scaling β but because the numbers grow quite large it is easier to think of complexity in relative terms. Finally, because the complexity grows very fast in N and M, we control for growth by taking the logarithm

which gives us the following normalized complexity measure.

$$C(R) := \frac{\log(M\binom{N}{M})}{\log(\lceil \frac{N+1}{2} \rceil \binom{N}{\lceil \frac{N+1}{2} \rceil})} \tag{7.3}$$

As an example, suppose we have a simple perceptron whose output unit has a bias of 1 and two binary visible units with weights $w_{1,1} = 1$ and $w_{2,1} = -0.5$. Then consider the rule $h = 1 \iff$ 1-of-$\{x_1 = 1, \neg(x_2 = 1)\}$. Over the entire input space we see that $R(x) \neq N(x)$ only when $x_1 = 0$ and $x_2 = 1$ giving us an error of 0.25. Furthermore, a $1 - of - 2$ rule is the most complex rule possible for 2 variables as it has the longest DNF of any *M-of-N* rule giving us a complexity of 1.

Using Eqs. 7.2 and 7.3 we define a loss function for a rule R as a weighted sum in which a parameter $\beta \in \mathbb{R}^+$ determines the trade-off between error and complexity.

$$L(R) := E(R) + \beta C(R) \tag{7.4}$$

Now that we have defined a loss function that measures both the error and complexity of a rule, we need to develop a search method that optimizes this quantity. The strategy we want to employ is brute force. Given a set of test inputs and a value of β, we will calculate the loss for every rule in a set of *M-of-N* rules and return the rule with the lowest result. A full search over the entire space of *M-of-N* rules already backs away from a true brute force search over the entire space of rules, but even this remains intractable computing. Because of this, we will employ some heuristics in order to reduce the search space to a manageable size. Our search procedure is therefore not a full brute force search over the entire solution space but a complete search of a smaller solution space with justifications for the exclusion of certain potential solutions. By repeating our search for various values of β we are able to explicitly determine the relationship between the allowed complexity of a rule and its maximum accuracy. For $\beta = 0$ the rule with the minimum loss will be the rule with minimum error regardless of complexity, and for β large enough the rule with the minimum loss will be a rule with 0 complexity, either a $1 - of - 1$ rule or one of the trivial rules which either always predicts true or always predicts false (these can be represented as *M-of-N* rules by $0 - of - N$ and $N + 1 - of - N$ respectively). Because all the networks we will test contain continuous-valued neurons, before we can apply our search to them we must find a way of translating continuous valued neurons into binary valued propositional variables. As discussed before, the simplest way

to do this is to choose a splitting value for the neurons with values above the split corresponding to True and values below the split corresponding to False. How do you choose a splitting value though? A naive approach might be to take the mean or median value of a neuron. If the neuron is bounded, one could potentially choose the center-point of its range. However, sometimes the values a neuron takes could be heavily skewed, meaning that whether or not the activation is above or below average tells us very little about the neurons impact on the neurons it feeds into as well as the network dynamics as a whole. In order to choose atoms that more accurately capture relevant neuron activations we will employ the information gain.

7.3 Generating Splits

When our network has continuous activation values, in order to define the atoms to use for rule extraction we must choose a splitting value a for each neuron leading to an atom of the form $h > a$. In order to choose splitting values for continuous neurons we use information gain (see section 3.4). Given a target neuron we wish to explain, h, we generate an atom for the target neuron by selecting a split based on the information gain with respect to the output labels of the network. More precisely, let X be a set of test examples, let $h(x)$ be the activation value of h with input x and $N(x)$ be the output label of the network given input x. X defines a distribution over the labels, $N(X)$, given by the relative frequency. For example, suppose we have a network with binary outputs and a test set $X = \{x_1, x_2, x_3\}$. If $N(x_1) = 0, N(x_2) = 0, N(x_3) = 1$, then $N(X)$ defines the distribution $P(0) = \frac{2}{3}, P(1) = \frac{1}{3}$.

For any real value a, We can partition X into two sets by splitting X into elements that produce an activation value of h larger than a and those that produce an activation value of h smaller than a. In other words, $X^+ := \{x : h(x) \geq a\}, X^- := \{x : h(x) < a\}$. We choose as our splitting value the value of a which has the highest information gain with respect to the network labels. In other words, we choose the value of a which partitions X to give the smallest average entropy of $N(X^+)$ and $N(X^-)$.

This gives us an atom in the form of $H := h \geq a$. Similar to the way a network defines a distribution over its output labels, given a test set, X, H defines a distribution over $\{0, 1\}$ in which the probability of 1 is defined as the

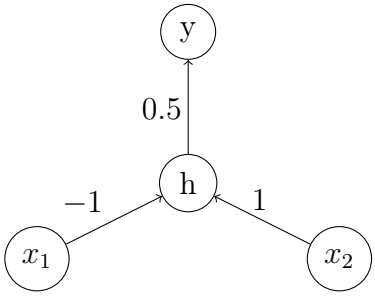


Figure 7.1: A simple perceptron with a single hidden node, a single output node, and two input nodes

fraction of examples $x \in X$ for which $h(x) \geq a$. We generate the input atoms in the same way as before but we maximize the information gain with respect to the distribution defined by the output literal rather than the network output. It is important to note that we generate a different set of splits of the input neurons for each target neuron we explain. This means that even if two target neurons appear to contain the same literals the different splitting values means the corresponding atomic variables are different. The whole process is illustrated in the following simple example

Suppose we have a perceptron with the structure given in figure 7.1. with a linear hidden activation function, a simple step function for the output activation, and 0 biases. Given the test examples $x^0 = (0.5, 1), x^1 = (1, 1), x^2 = (0.5, -1)$, we calculate $h(x^0) = 0.5, h(x^1) = 0, h(x^2) = -1.5$ and $y(x^0) = 1, y(x^1) = 1, y(x^2) = 0$. Then if we choose $a = 0$ to be our split we partition our test set into $X^+ = \{x^0, x^1\}$ and $X^- = \{x^2\}$. Then $y = 1$ with probability 100% on X^+ and $y = 0$ with probability 100% on X^- so the average entropy on these two sets is 0 and $a = 0$ obviously maximizes the information gain. Now let's calculate the output of the literal $H := h \geq 0$. $H(x^0) = 1, H(x^1) = 1, H(X^2) = 0$. Then we can either split x_1 with 0.5 or 1, if we split on 1 then we have $X^+ = \{x^0, x^2\}$ and $X^- = \{x^1\}$. The distribution defined by H on X^+ is the uniform distribution on two elements and thus has 1 bit of entropy. On X^- the entropy is 0 giving an average entropy of $\frac{2}{3} \cdot 1 + \frac{1}{2} \cdot 0 = 0.67$. If we split on 0.5 then $X^+ = \{x^0, x^1, x^2\}$ which gives the distribution $P(1) = \frac{2}{3}, P(0) = \frac{1}{3}$. This has 0.92 bits of entropy. Since X^- is empty this is equal to the average entropy. We see that when splitting on 1 we have less entropy and thus this is the split we choose for x_1 when finding rules to explain h.

In the case that the layer is convolution, each feature map corresponds to a group of neurons each with a different input patch. Rather than test every single neuron in the feature map we only test the one whose optimal split has the maximum information gain with respect to the network output. This gives us a single rule for each feature map rather than a collection of them.

7.4 Search Procedure (Extraction Method)

With the metrics defined, and the generation of splits detailed, we are ready to define the search procedure for our slow *M-of-N* extraction algorithm. Given a neuron h_j with n input neurons x_i, we generate splits for each neuron using the technique just described to produce an atom H_j and a set of atoms $\{X_i\}_i$. Using these we create a set of $\mathcal{O}(n^2)$ *M-of-N* rules whose bodies consists of unnegated variables from $\{X_i\}_i$ that have a positive connection to h_j (in other words $X_i : w_{i,h_j} > 0)$), negated variables from $\{X_i\}_i$ that have a negative connection to $h_j(X_i : w_{i,h_j} < 0)$ and whose head is H_j. We calculate $L(R)$ for each one of these rules and return a rule that minimizes it. In order to create the set of rules to search, we reorder the variables according to the magnitude of the weight connecting x_i to h_j (such that we have $|w_{1,j}| \geq |w_{2,j}| \geq ... \geq |w_{n,j}|$). Then we consider the rule $M - of - \{(\neg)X_1, ..., (\neg)X_N\}$ (where X_i is negated if $w_{i,h_j} < 0$) for each $1 \leq N \leq n$ and each $0 \leq M \leq N + 1$. The search procedure only relies on the ordering of the variables X_i.

A neuron with n input neurons has $\mathcal{O}(2^n)$ possible *M-of-N* rules which makes an exhaustive search intractable. However, here we choose an order for the literals based on the magnitude of the weight of the corresponding neuron. This reduces the search space to $\sum_{i=1}^{n} \sum_{k=1}^{i} \binom{i}{k} + 2 = \mathcal{O}(n^2)$ rules (the extra 2 rules come from the always true rule 0-of-N and the always false rule $N + 1$-of-N). The reduction of an exponential search space to a polynomial one makes our problem tractable although it can still be computationally difficult if the number of neurons and test examples is large. However, because the weight-ordering determines every rule we wish to test beforehand, we can further speed up the search by running it in parallel. This represents a significant advantage in speed over methods which search through the rule space by adding literals one at a time such as the hillclimbing method described in previous sections. The combination of an ordering along with the use of parallel computing allows us to examine the error/complexity landscape of networks which would be intractable using a sequential search. Although restricting

Algorithm 3 Search procedure for finding *M-of-N* rules to explain a hidden feature h

Input: A network $\mathcal{N}$, a set of input examples, I, and a hidden unit to explain, h

Generate a split, s, for h by choosing the value which maximizes the information gain with respect to the network output. Use this to define the atom H

for Each neuron x which is an input of h, **do**

Generate a split for x by choosing the value which maximizes the information gain with respect to H. Use this value to define the literal X if the connection between x and h is positive, and use it to define $\neg X$ otherwise

end for

Order the input literals by the magnitude of their weights

for $N : 1 \leq N \leq$ *number of inputs* **do**

for $M : 1 \leq M \leq N$ **do**

Create an $M - of - N$ rule, R, whose body consists of the first N literals. Then compute $L(R)$;

end for

end for

Compute $L(R)$ for the trivial rules $0 - of - \{\}$ and $1 - of - \{\}$;

return rule with the lowest value of $L(R)$.

the ordering makes our technique feasible for larger networks, it creates the possibility that we miss a truly optimal rule. How do we know that using the weight-ordering guarantees that we find the optimal *M-of-N* rule? Luckily, under certain conditions we can validate the optimality of the weight-ordering with the following theorem.

Theorem 7.4.1. *Given a binary network, N, consisting of n input neurons and a single output neuron. Then, over the test set $\{0,1\}^n$, the most accurate M-of-N rule is the one whose N literals consist of the N literals corresponding to the N neurons with weights of greatest magnitude.*

Proof. See Appendix □

This Theorem tells us that if our input space is binary and homogeneous in the sense that the values of each neuron have identical ranges and are mutually independent, then the literals of the optimal *M-of-N* rule will be those corresponding to the N neurons whose weights have the largest magnitude. The

result follows because of the symmetry of the input space. Another way of putting this is that the average contribution a neuron makes to the activation of the output neuron only depends on its weight. In the continuous case, assuming the same homogeneity condition, if the atoms are generated using the same splitting value the result holds. In general, different choices of splitting value may invalidate this result. For example, in the extreme case, if we choose a splitting value such that the atom is always true or false, it tells us nothing about the output value regardless of its weight. Luckily, by splitting based on the information gain we ensure that the corresponding atom tells us as much as possible about the output. Furthermore, if the input space is homogeneous, then neurons with higher weights must result in splits with higher information gains.

Theorem 7.4.1 tells us that the weight-ordering is optimal if the input space is homogeneous. For this reason, if we sample uniformly over the input space we can be confident that our approximation of error converges to the true minimum error of any *M-of-N* rule over this space as the number of samples increases. Unfortunately, over an arbitrary subset of the input space this is not always true. In practice, neural networks are not trained on datasets which are homogeneous, but rather on manifolds existing in some larger ambient Euclidean space. On these manifolds we expect neurons to be correlated, at least to some degree. When this is the case, adding two highly correlated neurons to a rule will not give a significant increase in fidelity, even when both have weights of high-magnitude. Rather a rule containing a lower-weighted neuron might provide more information about the output, thus invalidating the optimality of our weight-ordering heuristic. How much of an issue this poses depends entirely on the dataset and network, which in turn depends on a learning algorithm. Is it reasonable to assume that if two input neurons are highly correlated that the learning algorithm will assign them both high weights? Or will it assign only one of them a high weight and the other a weight close to 0? These questions aren't immediately clear meaning the impact of this effect on the optimality of our extraction algorithm is not impossible to determine analytically.

Even if our algorithm was able to produce optimal *M-of-N* rules in every case, we would still want to test it empirically as the choice of *M-of-N* rules to begin with is a heuristic, adding a degree of structural error. As pointed out in

Section 5.4, neural networks are capable of representing more than just *M-of-N* rules. Whether or not your typical trained neural network employs reasoning significantly differently than an *M-of-N* rule is something which must be empirically tested. For this reason, in order to be confident that our extracted rules are close to an optimal rule for a given error/complexity tradeoff, we will compare the parallel *M-of-N* extraction algorithm to the verifiably optimal algorithm CORELS on a number of small categorical datasets to see if weight-ordered *M-of-N* rules are a reasonable representation to approximate neural networks without straying too far from optimal. As we will see in section 8.2, *M-of-N* rules with a similar complexity offer only a marginal decrease in fidelity compared to the rules found by CORELS suggesting that restricting your set of explanations to *M-of-N* rules gives you approximately optimal explanations.

The above procedure generates rules for a single neuron. Repeating this throughout the network allows us to create a set of rules for every neuron. By doing so we replace a deep neural network with a set of *M-of-N* rules. Because the outputs of a set of rules extracted from one layer will be the inputs to the set of rules extracted from the next layer, these rules are hierarchical. However, there is a caveat to this. Because splits are chosen independently, the output atoms produced in one layer will not be the same as the input atoms produced in the next. If we want to create a usable set of hierarchical rules, we must develop a consistent splitting technique. This can be done in a number of ways. One approach is to average out the splitting values assigned to a neuron by each iteration of the extraction process. This naive approach will generate a consistent set of splits, but it may result in information loss and as a result: lower fidelity. A more sophisticated technique is the top-down approach. In the top-down approach, splits are generated one layer at a time. Starting with the penultimate layer, splits are generated using information gain with respect to the output label. Moving down a layer, splits are generated using information gain with respect to configurations of the atoms in the penultimate layer. This process can be repeated until splits for the input layer are generated. At each step, a single splitting value is chosen for a neuron. This produces a set of consistent splitting values for the network that can then be used as input for our extraction technique.

Despite methods being available for the generation of hierarchical rules, we do not employ them. This is because errors arising from rules extracted in

one layer of the network can propagate upwards and result in errors in higher layers. In effect, this means that the rules are bottlenecked by their least accurate layer. As we will see in the experimental results, some internal layers of a deep network have low fidelity. Any hierarchical rules generated would be no better than the rules generated for this layer. With this in mind, we can reasonably conclude that from the results that there is no accurate set of hierarchical rules explaining the networks in our experiments and therefore no need to search for one. In the next section we illustrate the process all the way through. Beginning with the generation of splits all the way to the final extraction of rules for various values of β.

7.5 A Demonstration of the Procedure

To demonstrate the procedure we will examine the extraction process for the first hidden feature in a CNN trained on the fashion MNIST data set. First we select 1000 random input examples from the training set and use them to compute the activations of each neuron in the CNN as well as the predicted labels of the network. Since the CNN pads the input with zeros and the input images are 28×28 we have $28 \times 28 = 784$ neurons per feature in the first layer. Each of these neurons correspond to different 5×5 patches of the input. To select a neuron to test, we select the optimal split of each neuron by computing the information gain of each neuron with respect to the predicted labels on the 1000 input examples. For the first feature we find that the neuron with the maximum information gain is neuron 335 which has an information gain of 0.05 when split on the value 0.134. This neuron corresponds to the image patch centered at $(335/28, 335\%28) = (12, 19)$. With this split we define the variable H by $H := 1$ iff $h_{335} \geq 0.134$.

Using this variable we define the input splits by choosing the values which result in the maximum information gain with respect to H. Note our test input consists of the 1000 5×5 image patches centered at $(12, 19)$ taken from the input examples. We then search through the *M-of-N* rules using the input variables defined by the splits to determine an optimal *M-of-N* rule that explains H for various error/complexity tradeoffs. As we increase the complexity penalty we extract four different rules which are summarized in Table 7.1. and visualized in figure 7.2. We can see from figure 7.2 that many of the weights are filtered out by the rules. The most complex rule only contains 6 of the total

Figure 7.2: An example of the various rules extracted for the first feature. The first image represents the weights of the neruon and the following four are rules of decreasing complexity explaining the neuron. Here grey indicates that the input feature is not included in the *M-of-N* rule, white indicates a positive literal, and black indicates a negative literal

25 input literals. Even completely ignoring the majority of the inputs the rules achieve an error of 0.061 or a 93.9% accuracy (See Table 7.1.). Penalizing the complexity extracts rules containing 4 literals and a single input literal with a significant increase in error only occurring when we reach a single input literal.

M-of-N	**Error**	**Complexity**
3-of-6	0.061	0.227
2-of-4	0.077	0.138
2-of-3	0.100	0.099
1-of-1	0.330	0.000

Table 7.1: Summary of the rules extracted to explain feature 1 with various error/complexity tradeoffs

To demonstrate how the rules are used we consider how they behave on an example input. Given an image patch from the test input, we derive the truth of a the corresponding literals by comparing the input values to their corresponding splits. This converts a real valued vector to a binary vector (See Fig 7.3). Using this binary vector we calculate the output of the rules by check to see if enough of the rule conditions are satisfied. In the example case we see that when looking at the $3 - of - 6$ rule, 3 of the positive literals are satisfied and 1 of the negative literals are satisfied meaning 4 of the total 6 literals are satisfied. Since this is a $3 - of - 6$ rule the predicted output for H is 1.

Using the weights and biases of the network we can see that the output of hidden unit 335 on the selected input image patch is 0.474. Since this is greater than the split value of 0.134 the network output for H is 1. In this case the extracted rules agree with the network output. The error is just ratio

Figure 7.3: An image patch used as a test input for the selected feature. On the left is the raw input and the right is the image converted to input literals by comparing the input values to the selected splits for each feature.

of examples for which the output of the extracted rules does not agree with the output of the network.

Chapter 8

Experimental Results

8.1 Outline of Experiments

Now that we are armed with a search procedure, we can begin applying it to deep neural networks. The focus of our experiments is on determining the relationship between the complexity and accuracy of an extracted rule. In order to achieve this, we run our algorithm on a set of neurons with a specific value of β and average the accuracy of each optimal rules found by the algorithm. Repeating this process for different values of β allows us to graph a complexity accuracy curve. By analyzing this curve we can determine just how complex an *M-of-N* rule needs to be on average to explain a neuron. The case argued against rule extraction would predict that, on average, extracted rules are either not accurate enough, or too complex to be viable explanations. Despite this being probably true in the general case, the intuition and theory of information compression behind deep networks leads us to extract rules from every layer separately to determine whether certain layers can be reliably explained with rule extraction.

The first set of experiments we carry out is a simple comparison with CORELS. Because our algorithm uses heuristics to reduce the search space, the rules it extracts do not represent the true optimal accuracy/complexity tradeoff for a given β. Luckily CORELS is verifiably optimal and uses an identical complexity penalty in the loss function. Therefore, the first set of experiments compares our algorithm to CORELS on small networks to determine whether or not the rules it finds are at least close to the optimal and can thus be used to make judgements about the overall explainability of a network.

The second set of experiments is the layerwise rule extraction. Three CNNs with an identical architecture are trained on MNIST and our *M-of-N* extraction algorithm is applied to each layer. The transfer function and learning algorithm are varied between the networks in order to observe the impact on explainability. When the final layer is found to be explainable, the second set of experiments are extended by applying rule extraction to the final layer in a larger number of networks. This is to verify that the good explainability of the final layer found in the previous experiments is a general feature of deep networks and not just an anomaly.

The third set of experiments tries to capitalize on the good explainability in the final layer by combining rule extraction in the final layer with importance methods (LIME, keras-vis) to explain the penultimate layer. The result is a type of hybrid explanation where the behaviour of a network is represented as an abstract rule with variables that can be visualized.

The final set of experiments replaces the final layer of a CNN with a set of extracted *M-of-N* rules to produce a new hybrid network-rule model. The classification accuracy of this model on adversarial examples generated from the original network is measured.

8.2 Benchmarking with CORELS on Categorical Datasets

Before we apply the algorithm outlined in the previous section to larger networks, it is important to examine how close to an optimal solution the search method is getting. In order to do this we will compare the *M-of-N* search method with the slower, but proveably optimal search method CORELS Angelino et al. [2018]. CORELS performs a sequential search over a space of rule lists with a fixed maximum size for the antecedents to find a rule list which provides an optimal complexity/error tradeoff for a given complexity regularization parameter λ. CORELS uses theoretical results to reduce the run time of the algorithm by calculating lower bounds for error and complexity of an optimal rule list at each step and terminating the search once any of the bounds get violated. This has the potential of greatly reducing the run time of the search but it remains too slow to apply systematically to the hidden

neurons of even a medium sized deep neural network. Since our extraction algorithm uses an ordering on the literals, each rule can be evaluated independently so that the search procedure can run in parallel. This greatly speeds up the search compared to CORELS, which requires a sequential search. By making the assumption that *M-of-N* rules provide a good approximation of an optimal rule list describing a single hidden unit we are able to examine the accuracy/complexity tradeoff for rules extracted from much larger networks.

Before making a direct comparison with CORELS we describe under which circumstances a weight-ordered *M-of-N* search will not be optimal. In this case when we describe a search procedure as optimal we mean that it has no structural error as defined in section 5.3. Compression error will be present in any search procedure that searches through a discritized space of rules to evaluate against a continuous model. The compression error may be reduced either by choosing a better compression method or by implementing the compression step during the search procedure. Complexity error is built in to the search procedure and acts as the independent variable in our experiments. Since the set of boolean functions modelled by a single layer in a neural network is strictly larger than the set of *M-of-N* rules, our search procedure will not produce an optimal solution whenever a hidden neuron computes a boolean function that is *not* an *M-of-N* rule. Furthermore, by ordering the literals by weight we may not be able to find an optimal *M-of-N* rule if multiple variables in the input space are highly correlated. It can be easily shown that if the input space is of the form X^n (where $X \subset \mathbb{R}$) then by 7.4.1 the weight ordering will produce an optimal *M-of-N* rule. However, since we wish to evaluate on more complicated input spaces our search procedure may find a non-optimal *M-of-N* rule during our experiments. For these reasons we empirically evaluate the *M-of-N* search procedure against CORELS. In order to compare our search procedure with CORELS as an optimal baseline Angelino et al. [2018], we train a deep neural network with 2 fully connected layers of 16 and 8 hidden neurons, respectively, with a rectified linear (Relu) activation function, on the car evaluation dataset. The car evaluation dataset is a small categorical dataset meant to predict the quality of a used car based on several features. There are 7 features, 4 output classes, and 1728 instances Bohanec and Rajkovic [1988].

Since CORELS has multiple parameters to penalize complexity we run CORELS multiple times with different parameters to generate a set of rules with higher

Method	Comp	Acc	Network	Time
CORELS(1/0.01/0.01)	n/a	n/a	DNA promoter Layer 1	n/a
Parallel *M-of-N* $\beta = 0.3$	0.239	89%	DNA promoter Layer 1	700s
CORELS (1/0.01/0.01)	0.124	93.4%	Cars Layer 1	$1s$
CORELS (2/0.05/0.05)	0.04	87.3%	Cars Layer 1	$1800s$
Parallel *M-of-N* $\beta = 0.2$	0.131	90.3%	Cars Layer 1	$1s$
Parallel *M-of-N* $\beta = 1$	0.031	85.4%	Cars Layer 1	$1s$
CORELS (1/0.01/0.01)	0.053	99.05%	Cars Layer 2	$1s$
CORELS (3/0.02/0.02)	0.079	99.42%	Cars Layer 2	$1s$
Parallel *M-of-N* $\beta = 0.3$	0.057	98.4%	Cars Layer 2	$1s$
Parallel *M-of-N* $\beta = 0.1$	0.069	98.6%	Cars Layer 2	$1s$
CORELS (1/0.01/0.01)	0.165	91.6%	E.COLI Layer 1	$1s$
CORELS (2/0.005/0.001)	0.287	92.6 %	E.COLI Layer 1	$10s$
Parallel *M-of-N* $\beta = 0.2$	0.132	89.4%	E.COLI Layer 1	$1s$
Parallel *M-of-N* $\beta = 0.1$	0.189	90.2%	E.COLI Layer 1	$1s$

Table 8.1: Comparison of similarly complex rules extracted by CORELS and parallel *M-of-N* measures from the layers of three different networks trained on categorical datasets

complexity and one with lower complexity and then compare these rules to rules of similar complexity found by our parallel search. In Table 8.1 we can see that rules found via our *M-of-N* search are only marginally worse than a set of optimal rules with similar complexity found by CORELS and that CORELS can become quite slow when using too broad a search on a dataset with many inputs. Notice also that in this example the second hidden layer is much more explainable than the first, c.f. the large difference in accuracy between layers. Finally, the rate of accuracy decrease vs. complexity of Parallel *M-of-N* seems to be lower than that of CORELS; this deserves further investigation. In summary, the above results show that a parallel *M-of-N* search can provide a good approximation of the complexity/error trade-off for the rules describing the network. Next, we apply Parallel *M-of-N* to much larger networks for which sequential or exhaustive methods become intractable. CORELS was also compared to our *M-of-N* extraction algorithm on a one layer network trained on the E. Coli dataset Horton and Nakai [1996]. This dataset contains 7 features, some categorical and some real, predicting the localization site of a protein on an E. Coli bacteria. There are 8 output labels. The network was a single layer

fully connected network with 100 hidden units and a Relu activation function. In this case, we found that the extracted rules are less directly comparable due to more significant differences in complexity between CORELS and parallel *M-of-N*. Despite this, CORELS is clearly able to find superior rules, although not by a significant margin, with a complexity difference of 0.1 adding only a 2% increase in accuracy. Finally we ran our experiments on the DNA promoter dataset Towell et al. [1990] using a large set of synthetic examples but CORELS was unable to find a solution before exiting due to memory overflow. The DNA promoter dataset is a categorical dataset consisting of 58 features representing a gene sequence with the classifier predicting whether or not the gene is a promoter. The original dataset contained 106 instances but in order to test the extraction algorithms, 10000 instances of synthetic data were generated.

8.3 MNIST

In order to evaluate the capability of compact *M-of-N* rules of explaining hidden features, we now apply the extraction algorithm to the hidden layers of three different networks trained on MNIST and compare results. MNIST is a visual dataset consisting of 28×28 pixel greyscale images representing handwritten digits with an associated class Lecun et al. [1998]. Since applying extraction hierarchically can cause an accumulation of errors from previous layers, we use the network to compute the values of the inputs to the hidden layer that we wish to extract rules from. Hence, the errors from the extracted rules correspond to rule extraction at that layer. This allows us to examine the relative explainability at each layer. In practice, one could extract a hierarchical set of rules by choosing a single splitting value for each neuron.

Our three networks are identical save for the activation function and training procedure. The network architecture consists of two convolutional layers with 16 and 8 hidden units respectively, each with a 3×3 convolutional window and using max pooling. This is followed by a 128-unit densely connected layer with linear activation followed by a softmax layer. The first network uses Relu units in the first two layers and is trained end-to-end. The second network is trained identically to the first but uses the hyperbolic tangent (Tanh) activation function in the first two layers. The third network uses an autoencoder to train the first three layers unsupervised before training the final softmax layer

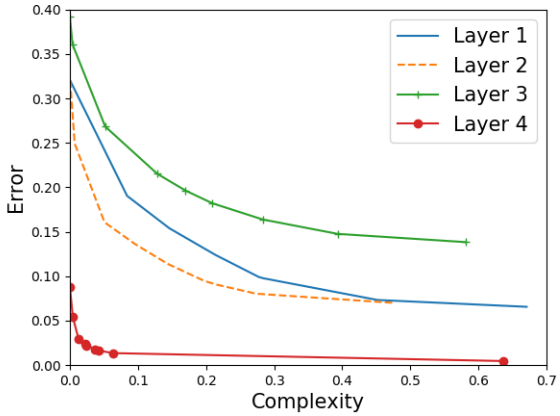
separately.

Comparing the network using Relu to the one using Tanh shows that in both cases the minimum error for each layer remains approximately the same. However, the explainability in the Tanh network is greatly increased in the first three layers, rules extracted from the Tanh network can be made much less complex without significantly increasing the error. This applies not only to the first two layers but also to layer 3 which uses a linear activation in both cases. In both cases the third layer is much less explainable than the first two and the only layer which we are truly able to produce an acceptably accurate and comprehensible explanation is the final one in which we see rules with an average complexity of 0.087 achieving an average error of 0.013%.

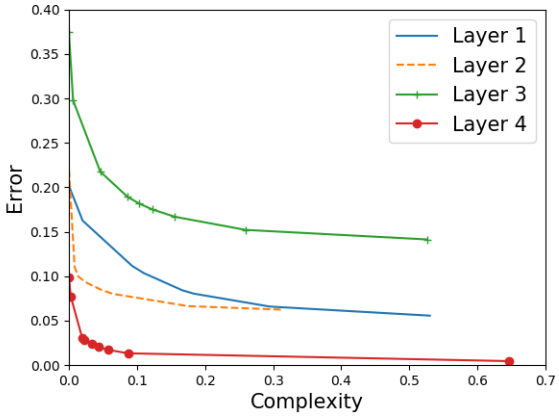
In the third layer we believe that the higher minimum error is mainly the result of the number of input units. Our results suggest that there are a lot of input units which are not relevant enough alone to be included in an *M-of-N* rule, but collectively they add enough noise to have a significant effect on the output. Because our search procedure is heuristic, it is possible that a more thorough search could produce rules which are simpler and more accurate but our results seem to at least tentatively back up the idea that the distributed nature of neural networks makes rule extraction from the hidden layers impractical if not infeasible. We hypothesize that the difference in complexity between rules extracted from the Tanh network and the Relu network is due to the saturating effect of the tanh function. A hidden neuron in the tanh network may have fewer 'marginally relevant' features than in the Relu network. This would explain the steep decline in accuracy found in the Tanh network and the more gradual decline found in the Relu network.

The autoencoder has hidden features which are in general more explainable than either of the two previous networks. Compared to the Relu network, the error of the extracted rules in the second layer is lower at every level of complexity. Compared to the Tanh network, the autoencoder has more accurate rules at medium levels of complexity (6.1% error at 0.144 complexity vs. 6.6% error at 0.18 complexity). However, as complexity is reduced the extracted rules in the Tanh network remain accurate for longer (9.6% error at 0.053 complexity vs. 8.4% at 0.048 complexity). Interestingly, in the autoencoder the second layer is slightly less explainable than the first. The third layer is

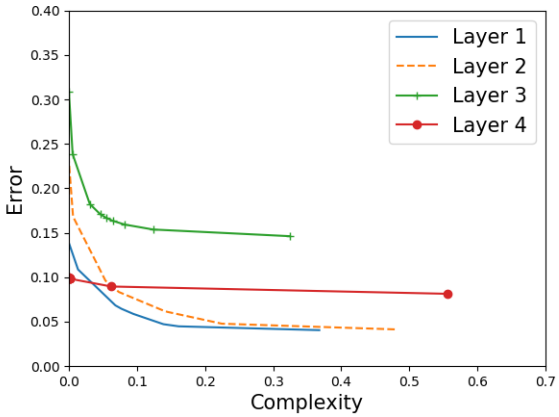
more explainable than it is in the other two networks with significant increases in error only being seen with rules of average complexity less than 0.08.



(a) CNN-Relu



(b) CNN-Tanh



(c) CNN-AE

Figure 8.1: The Complexity/Error relationship for rules extracted from each layer of three different deep networks trained on MNIST. From left to right a CNN with Relu activations trained end-to-end, a CNN with tanh activations trained end-to-end, a CNN with Relu activations trained as an autoencoder.

In the softmax layer trained on top of the autoencoder we see that one cannot extract accurate rules of any complexity. This points to something fundamentally different from the previous two networks in the way that softmax uses the representations from the final layer to predict the output. This is the subject of further investigation.

Our results indicate that, when using binary features and *M-of-N* rules with an assumption of weight-ordering, there are hard limitations to representing hidden units that cannot be overcome with any level of complexity in CNNs. These limitations seem to be the result of the internal representations determined by the training procedure. Whether these limitations can be overcome by refining rule extraction methods or whether they are a fundamental part of the network is to be determined. Discretizing the neurons into a multiple variables rather than binary is likely to improve accuracy at the cost of complexity. Furthermore, although *M-of-N* rules appear to approach the optimal accuracy/complexity tradeoff, it remains possible that expanding the search space could result in simpler and more accurate explanations of the internal features. What we can say is that the final layer of a CNN may be a promising target for rule extraction.

8.4 Extraction from the final layer

Following the observations of the previous section, we extract rules from the final layer of 2 additional CNNs trained on the Olivetti faces dataset and the fashion MNIST dataset. Olivetti faces is a small facial recognition dataset consisting of 64×64 pixel greyscale images of the faces of 40 different people Samaria and Harter [1994]. Fashion MNIST is a replacement for MNIST in which handwritten digits are replaced by images of different articles of clothing. The dataset is set up to be identical to MNIST in terms of input dimension, number of examples, number of classes etc. Xiao et al. [2017]. The networks trained on these datasets are both 4-layer CNNs consisting of two convolutional layers, a fully connected layer, and a softmax. The Relu activation function is used. In all six cases, we observe that units in the softmax layer can be accurately explained by relatively simple rules (See Table 8.2.). The Olivetti faces dataset had the most accurate and interpretable rules of all, this is probably at least partially due to the smaller size of the dataset. In all cases we can see a massive drop off in the complexity with only a penalty of $\beta = 0.1$ with a less

Dataset	**Comp.** ($\beta = 0$)	**Acc.**($\beta = 0$)	**Comp.** ($\beta = 0.1$)	**Acc.**($\beta = 0.1$)
Olivetti Faces	0.03	100%	0.024	99.9%
MNIST	0.7	99.6%	0.06	98.7%
Fashion MNIST	0.28	99.3%	0.06	98.8%
Car Evaluation	0.18	99.9%	0.0	99.7%
DNA promoter	0.9	99.1%	0.06	96.4%
E.Coli	0.145	96.8%	0.06	96.5%

Table 8.2: Comparison of the complexity (Comp), and accuracy (Acc) of rules extracted from the final layer of three CNNs trained on different datasets as well as three fully connected networks trained on different datasets. Repeated for complexity penalties of $\beta = 0$ and $\beta = 0.1$

than 1% decrease in accuracy. This suggests that in the softmax layer, relatively few of the input neurons are being used to determine the output. This opens up the possibility of using rule extraction in the final layer as a method of explanation for deep neural networks. By extracting from the final layer we change the explanation task from explaining the output label to explaining the relevant hidden nodes. As the experiments in the previous section showed, our *M-of-N* rule extraction method is inadequate for this task. Instead, we will employ importance methods to explain hidden neurons and combine this with rule extraction in the final layer to produce an overall explanation of the network. We repeat these experiments on several small datasets trained with small 1 or 2 layer networks. The DNA promoter and Car Evaluation datasets are categorical classification problems whereas the E. Coli dataset consists of 7 continuous features. The E. Coli dataset and the DNA promoter dataset were learned by a single layer network whereas the car evaluation dataset was learned by a small 2 layer network. All activations are Relu for the hidden units and softmax units for the output. Here we find similar results to the previous datasets although not as drastic.

8.5 Visualization Using Importance Methods

The next set of experiments will deal with the combination of rule extraction and importance methods. For this reason, in this section we will briefly go over the notion of importance methods along with several influential examples. Currently, importance methods are the main alternative to rule extrac-

tion. Although techniques exist that build interpretable models from scratch, as discussed in the previous section, these can be thought of as solving a separate problem from rule extraction. For this reason, along with the fact that we will combine several importance methods with rule extraction in future experiments, we discuss importance methods here.

Importance methods generate representations of features or classes by searching for input patterns which in some way can be considered prototypical of the explanation target. For hidden features, a prototypical input is one which maximizes its activation over a given input space. This can also be applied to class labels by choosing the input which most strongly predicts a given class. Importance methods can also be local in which case they determine, given a specific input, the regions or features of the input that most strongly explain the target output. We'll be especially concerned here with the case that the input space is visual in nature. Although all of the methods we discuss here can be applied to any neural network, for networks trained on categorical data, the explanations provided by gradient methods might not make sense. In particular, the regularizers chosen are meant to reflect properties of natural images and not other datasets. This gives us important examples of explainabilility methods in which the nature of the dataset plays a crucial role in our approach to explainability. With that said, let's discuss how gradient methods work.

Gradient methods are a popular and influential approach to explainability that work by finding input patterns which maximize the activation of a hidden neuron. By doing this we hope to find distinct and identifiable features that correspond to a specific hidden neuron. In the first layer of a network this can be easily done by visualizing the weight filter. In higher-layers, however, the non-linear relationship between the input and hidden neuron activation make this problem difficult to solve simply by inspection. Instead we rely on search techniques to find appropriate inputs. One of the first such techniques did this by starting at an initial input image and calculated the gradient of the target activation with respect to the input values. By using gradient ascent we can converge to an input image which produces high activation values in the target neuron Erhan et al. [2009]. One downside to this is that the images produced do not always resemble images found in the input space. This is because the search takes place over all input configurations and it is often possible to find 'hacks' which are input patterns that contain no meaningful

information but nonetheless have a strong influence on the target neuron. This can be overcome to a degree with the introduction of regularization intended to penalize images that do not conform to a natural image prior Simonyan et al. [2014], Mahendran and Vedaldi [2015]. One of the most recent gradient techniques, DeepVis, uses four regularizations which can be tuned with various hyperparameters Yosinski et al. [2015]. These regularization parameters include; L_2 decay, which penalizes pixels in the image with too high an activation, Gaussian blur, which penalizes high-frequency information, small-norm clipping, which removes pixels with small-norm, and finally small contribution clipping, which removes pixels that do not contribute much to the activation of the target neuron. By including these regularization techniques, DeepVis biases the search to produce images that more closely represent things that can be seen in the dataset which, in this case, consists of natural images.

Gradient techniques with regularization can be effective at explaining hidden features in networks trained on natural images, but sometimes more general techniques which attempt to explain individual examples in networks are more desirable. The most popular of these techniques is LIME Ribeiro et al. [2016]. Given an input/output example of the network, LIME generates synthetic data by sampling from the input space in a way that biases examples close to the given input. It then calculates the corresponding network outputs for the sampled input data. Using these input/output pairs, LIME performs a linear regression with the input and the target label to be explained. The input features with the largest regression coefficients are then considered to be the explanation of the label around the given input. When the data is categorical the creation of input examples is straightforward, but when it is continuous -as is the case for visual data- LIME uses a segmentation algorithm to associate a set of binary variables with an image. LIME then generates its synthetic data by randomly choosing segments to replace with a uniform segment in which the value of every input is simply the average of all input values in the original segment. By doing this, LIME generates synthetic visual data that can be represented by a set of binary variables each corresponding to a particular segment of the original image. The regression is done using these variables.

8.6 Combining Importance Methods with rule extraction

The *M-of-N* rule extraction experiments were unable to adequately explain the internal features of a deep network. In many cases we are unable to find a rule that is both accurate and simple enough to be understandable. Although the it is conceivable that more sophisticated rule extraction techniques will be able to find satisfactory explanations by expanding the search space, this runs into the danger of being intractable for real-world networks and there is always the risk that within certain hidden layers any rule simple enough to be of use may necessarily have a low fidelity. This is in addition to the underlying issues of hierarchical rule extraction, namely that the higher up in the deep network a hidden neuron is the less accurate and more complex the hierarchical rules explaining it will be. Despite this, the experiments also show that, at least in the case of the networks trained end-to-end, the final softmax layer can be explained well with simple *M-of-N* rules. Despite having hundreds of inputs, often no more than a dozen input variables are required to produce a rule with very high fidelity. This shows that rule extraction can be a beneficial tool for explainability in deep networks, but that on its own it is insufficient to produce human understandable explanations.

We have seen that in some deep networks rule extraction can produce accurate and interpretable rules for the labels in terms of hidden units but that the hidden units themselves are not easily explainable with rule extraction. The widely held intuition that higher layers represent more abstract features may contribute to this phenomenon, while low-level concepts may rely on highly complicated relationships, neurons in higher layers may represent the kinds of abstract concepts we are used to reasoning with using logical systems. Although, as discussed in section 4.5, all layers of a deep neural network can be described by propositional implications, the number and complexity of these rules may make the translation from a distributed system to a symbolic one fruitless. This might seem disappointing at first, but it also nicely conforms to some of the ideas about intuition and reasoning. Low-level units of perception can be seen as operating on a more intuitive level, finding weak associations between large numbers of patterns which can lead to a conclusion about the corresponding output unit. Intuition is generally accepted to be more or less unexplainable in humans. A number of small details which, alone, can not de-

termine the conclusion but nonetheless are able to, in the aggregate, influence perception. This kind of reasoning is in direct opposition to the more structured 'logical' reasoning. The application of formal rules to abstract concepts in order to derive a conclusion. In this style of reasoning, we can trace an exact sequence of deductions that led us to the conclusion. This is because when we're dealing with a small number of abstract concepts that relate to each other in a more straightforward way, it is easier to follow the exact process that lead you from A to B and there is less ambiguity as to which factors were and were not relevant. In other words, a task like recognising that an object is a circle is not typically something that can be reasoned out with a large number of 'primitive' variables closer to raw sensory data (such as pixel shadings). In contrast, once we have a repertoire of abstract objects we can recognise, making inferences in terms of these objects generally takes the form of less complicated rules. For instance, explaining all the variable dependencies between pixels on a computer screen and the class of circles would be a hopeless task. However, if you already have access to the abstract notion of 'semi-circle' then an explanation of a circle becomes very simple. For this reason, the layered abstraction folk-lore of deep neural networks probably suggests something similar to the results we found. Symbolic reasoning, although on a theoretically level is not any different from a feed-forward network, is ill-suited to explaining the low-level transformations of deep neural networks but much more capable when dealing with a small number of abstract ideas which relate to each other in relatively predictable ways.

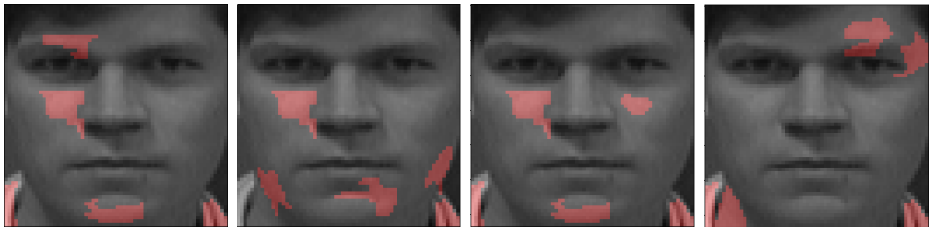
This provides the motivation for a more modular approach to explainability. When pressing a person for an explanation, we require that they frame the argument in terms of 'primitive concepts' for which we require no explanation. We would never ask how a person knew that there was a stop sign in front of them or that there was a bird overhead. We trust that these facts are justified by the persons own perception and intuition. The sensory apparatus of a person can be seen as a map from extremely high dimensional sensory input to a relatively compact set of concepts that can be described by some kind of logical relationship between them.

Moving back to our experiments, although our results for the hidden layers are by no mean the final say for the applicability of rule extraction in internal layers, they do suggest that perhaps the kinds of processes going on in lower

layers are those intuitive process which are the function of too many variables to hope to understand in a comprehensive way. The compactness of rules extraction from the final layer results tells us that the reasoning used by the final layer may indeed be in terms of abstract concepts or objects learned by the internal layers. This just leaves one problem, because we cannot apply rule extraction to explain which concepts are being used by the higher layer, we are forced to look to alternative methods to explain the complicated hidden units. For this we turn to importance methods. Using importance methods we can generate visual explanations for the hidden units and use these in conjunction with rule extraction in the final layer to produce an explanation for the label. By combining rule extraction with importance methods we hope to produce more comprehensive explanations for a network than those produced by using importance methods alone. Model agnostic methods can often produce explanations of a high fidelity but they don't examine any of internal logic of a model which means they can miss important aspects of the reasoning process actually used. For a deep network, layerwise rule extraction will explain the network using some of the same logical dependencies and hidden variables that the network uses. This has the potential to provide a more in-depth understanding of the networks behaviour at the expense of reduced comprehensibility. We will explore two possible ways of combining visualization techniques with rule extraction. The following experiments serve as a proof of concept and in order to realize the full potential of modular explanations much more experimentation must be done.

For our first experiment in combining importance methods with final layer rule extraction we will use the popular model-agnostic importance method LIME (see section 8.5 for a full description of LIME). We run our experiments using a CNN trained on the olivetti faces dataset. The architecture of the CNN is similar to the ones used for the accuracy/interpretability experiments. It consists of 4 layers, 2 convolutional, 1 dense, and one softmax. We use Relu transfer functions for the convolutional layers and a linear function for the dense layer.

We extract *M-of-N* rules from the final layer using the same procedure as before and find that we can achieve rules that have 100% fidelity and very low complexity (see 8.2. For example, label 32 can be explained with 100% fidelity by a $3 - of - 5$ rule despite having 256 input neurons. For this reason, we



(a) Label 32 (b) $\neg H_{228}$ (c) $\neg H_{246}$ (d) H_{71}

Figure 8.2: A comparison of visual explanations produced using LIME to explain label 32 for a given input example and visual explanations for three hidden units which result in the network predicting label 32 according to the extracted rule.

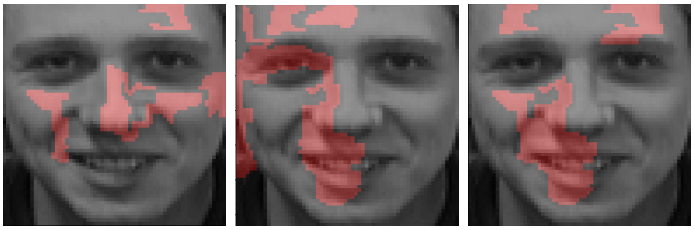
will use label 32 as an example for our combined rule extraction/importance procedure. The high fidelity can at least partially be explained by the fact that the Olivetti dataset is much smaller than MNIST, but the results are still promising. In order to combine LIME with our extracted rules we take each of the literals, $(\neg)H_i$, from our extracted rule and use LIME to explain the classifier defined by $f(X) = 1$ iff $(\neg)H_i = True$. In practice we do this by creating a new network which is identical to our CNN for the first three layers and whose fourth layer consists of a single neuron. If the literal we want to explain is positive (ie we have $\neg H = 1$ iff $h \geq a$ for some hidden neuron h and splitting value a), then we give our single output neuron a bias of $-a$ and connect it to h with a weight of 1 and leave all of its other weights as 0. For a negative literal, we do as before but swap the sign of the bias and weight. We finish by giving the output neuron a sigmoidal transfer function (this is for practical implementation purposes).

Given an input with label we wish to explain we first find the *M-of-N* rule corresponding to that label. We then find the literals from the rule which are true given the input. We produce explanations from M of these literals using LIME by applying it to the corresponding networks described above. If there are not M true literals for a given input then the rule predicts that the label is not true given the input. If this is the case we can use the fact that *M-of-N* rules are closed under negation to produce an explanation for why the label is not true. We demonstrate this by applying it to several examples for label 32 from the network described above. We first choose two examples for which the network outputs label 32. The rule explaining this label

is $label32 \leftrightarrow 3 - of - \{H_{71}, H_{154}, \neg H_{228}, \neg H_{238}, \neg H_{246}\}$, for our first example we find that every one of the input literals is satisfied. We randomly select 3 of them and apply LIME to generate an explanation for each [1]. We also apply LIME to the label itself to compare an explanation for the output of the whole network to the outputs for the target hidden neurons (See figure 8.2). We can see that the explanation for the label includes some features not found in the explanations of the hidden neurons. For example the area above the right eye is selected as being important in determining the label but it is not an important feature for any of the three literals we can use to explain the label. Additionally, the explanations for the literals contain features not found in the explanation of the whole network. Notably parts of the jawline and the area above the left eye. We also find that some features are common to all explanations, namely the base of the neck. This raises an important question, although LIME identified features as relevant to explain the label, it also did not identify those features as relevant to the hidden literals which we know explain the label with 100% fidelity. The visualization of the features combined with the rule allows us to reason counter-factually about the example. We see that the right cheek is important for 2 of the three features. If it is important for 1 of the two features not visualized then changing it could potentially switch 3 of the literals from being true to not true and the rule would no longer be satisfied. The chin, on the other hand, is only important for one of the features. If 1 of the remaining 2 unvisualized features does not depend on the chin, then changing it would probably not effect the truth of at least 3 of the 5 literals and the predicted label would remain unchanged. When explaining the label, LIME uses the regression coefficient of each feature to rank its importance. By using rule extraction and visualizing the features, we have an alternative method of ranking the importance of segments. Namely, how many features is this segment relevant to?

The reason for the discrepancy in the visualization of label 32 and the visualization of the relevant features in the rule predicting label 32 may be in the hyper-parameters of LIME. Given that LIME is a local model and builds a regression using data that is very far away from the typical data a network is trained on, options like the number of features to select may hide important regions for the

[1] *a priori* there is no selection criteria for which of the literals are chosen to explain. Only that they come from the set of literals satisfied in the example. If we wanted, we could visualize all of the literals satisfied as well.



(a) H_{246} (b) $\neg H_{71}$ (c) $\neg H_{154}$

Figure 8.3: Visual explanation for 3 of the literals in a negative example of label 32

regression. We can also apply this method to negative examples. Because the label is given by the rule $3-of-\{H_{71}, H_{154}, \neg H_{228}, \neg H_{238}, \neg H_{246}\}$, the negation of the label can be expressed as $3-of-\{\neg H_{71}, \neg H_{154}, H_{228}, H_{238}, H_{246}\}$. We can use the same method to explain the network output on examples where the network predicts a different label. In figure 8.3 we provide an explanation for why the literals in the rule for label 32 were not satisfied. As we can see, the regions of the image accounting for this are very different than the regions identified in the positive example. The shoulders are not a factor here for any of the literals whereas the mouth and nose are. This is interesting because the shoulders in this example are quite different than in the positive example. The bottom corner patches of the image are quite dark in the negative example whereas in the positive example they are quite light. Given that the shoulders were so important to the decision of the positive examples one might expect the difference in this region to play an important role in the decision. Despite the major differences, some common regions are identified in positive and negative examples, namely the cheeks, indicating that the shape of the cheek is an important feature for discerning between these two labels.

The variation in the features identified between examples indicates that the literals might represent a complex subspace of visual patterns. In order to feel like we've managed to generate an acceptable explanation of a face, we need to understand the hidden units used for the reasoning of the output rules. In our example we noticed unit 32 was a $3-of-5$ rule and gave examples of how LIME can generate explanations for the label itself as well as the hidden units. The downside to LIMEs locality is that it makes definitive reasoning about classes more difficult. In our example we found that the literals that predict label 32 were using information from the right eye region, cheeks, and. shoul-

Figure 8.4: Visual explanations for the literal H_{154} from three different examples

der. In this case it might be tempting to conclude that our classifier primarily looks at these regions to make a judgements; however, in a different example the explanations for the same hidden units may be totally different. In order to address this, we test several more examples with hidden units 154 to determine whether or not the unit can reasonably be said to represent a meaningful concept that could be use to reason about the identity of a person, (maybe something like high cheek bones, or a crooked nose, etc). From figure 8.3, it is apparent that for all visual explanations for label 32 the shoulders seem to provide crucial information. As we see in figure 8.4, what is unique to unit 154 is the importance of jaw and eye regions. This suggests that this literal may represent something specific about a facial structure that the network uses to determine which individual it is looking at. Although the locality of LIME is desirable under certain conditions, it does not allow one to confidently identify global features that a hidden unit may represent. For this reason, we turn to gradient methods to give a more global description of a hidden unit.

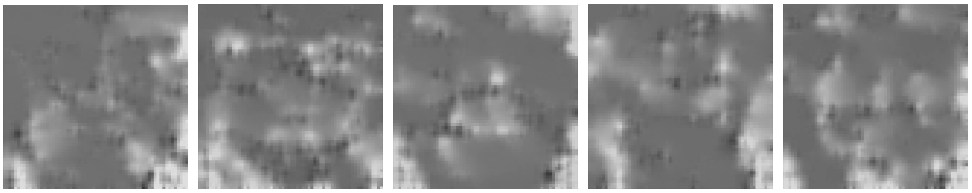
8.7 Combining Gradient Methods with Rule Extraction

Now we turn to gradient methods to provide a visual explanation of the features used by rules extracted from the final layer. To do this we use the keras-vis package which optimizes an input image to produce the highest activation value of a hidden unit subject to two optional regularizers; LP norm and total variation *Kotikalapudiet al.* [2017]. LPNorm penalizes very high or very low activations preventing the image from selecting a few important pixels to maximize (or minimize). Total variation penalizes images with a high variability between adjacent pixels. This is done to produce images which more closely resemble natural images in which the intensity of a pixel varies more smoothly

than in random images. Again we apply this technique to the rule for label 32, $3-of-\{H_{71}, H_{154}, \neg H_{228}, \neg H_{238}, \neg H_{246}\}$. We apply the gradient technique on each of the literals using the same technique we did when explaining literals with LIME. That is, we copy the network and remove the final layer, then add a layer containing a single node and a single connection to the desired literal with a weight of either 1 or -1 depending on whether nor not the literal is negated. This allows us to use gradient techniques for both negative and positive literals. For each image generated we use a variety of weights on the regularizers in order to generate meaningful images. Experimentally we found high weights on LP norm and low weights on the activation maximization and total variation to produce the best images but the exact values differ for many of the images produced.

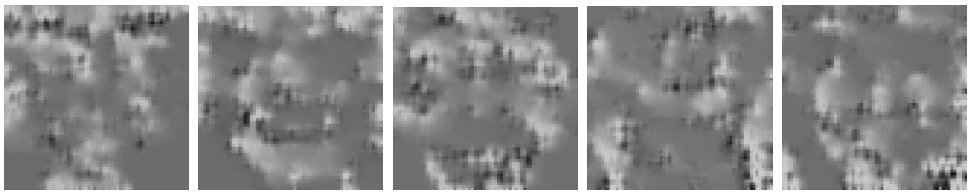
In figure 8.5 we see the input images found to explain each of the relevant literals. Although indistinct, the outline of a facial shape in many of these filters is noticeable. The vagueness of the features makes a precise analysis impossible but some conjectures can be made. To start with, these images provide information on the important regions of the input for the literal. None of the literals are highly localized, each representing multiple regions of the image in different ways. Again we can see that the shoulder area plays an important part for each of these filters although other regions of the image are more or less important for various literals. Comparing the regions identified by LIME as important to unit H_{154} and the image produced to maximize the output of H_{154}, we can see that the regions identified by LIME in each example are all present in the visualization. The shoulders, right section of the chin, and area above the right eye all play an important part in the activation of this hidden unit. Additional areas not found in the LIME examples are also highlighted. For example, the mouth is a distinct region but in none of the examples with LIME did it appear. We expect that for each example given to LIME, the regions identified as important will be among the same relevant regions found by this visualization. We also expect there to be regions that contribute a lot in the global example that aren't necessarily found in every local example. Although tentative, the gradient images seem to support this.

We can also use visualize the negated rule using the same technique used before. In figure 8.6 we can see the results of applying the gradient method to the literals Applying the gradient methods to the literals $\neg H_{71}, \neg H_{154}, H_{228}, H_{238}, H_{246}$.



(a) H_{71} (b) H_{154} (c) $\neg H_{238}$ (d) $\neg H_{228}$ (e) $\neg H_{246}$

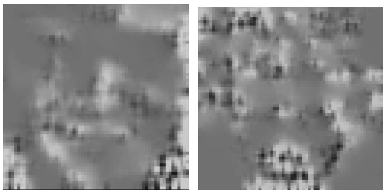
Figure 8.5: Visualizations of the literals in rule 32 using gradient methods



(a) $\neg H_{71}$ (b) $\neg H_{154}$ (c) H_{238} (d) H_{228} (e) H_{246}

Figure 8.6: Visualizations of the literals in the negation of rule 32 using gradient methods

The images produced here are similar to the previous ones in that each literal is not negated by a local region but by areas of the entire image. We see that the regions highlighted are much different than the ones highlighted by rule 32 with the relevant regions usually corresponding to regions which were irrelevant in rule 32. This helps explain the results from LIME. Notice that the shoulder region is not highlighted at all here, indicating that a bright region around the shoulder is evidence for the literals, but that a dark region around the shoulder is not evidence against them. Rather other features which negate the literals are important. Combined with our rule, this tells us that the network looks for features which support the label, and regions which do not support the label, and then weighs them, with a bias towards not supporting the label (because we require 3 of the literals to be satisfied, a clear majority). That is to say, the absence of features which support the label is not considered evidence against the label but rather it is only the presence of other features which provide evidence against the label. Looking at the negative examples for LIME, we can see that the relevant regions identified there correspond to relevant regions in the visualization. The consistency between LIME and visual methods helps assure us that the these are these representations of the hidden units are accurate. Using gradient methods we are able to not only produce images that visualize a single literal, but images



(a) $H_{71} \wedge H_{238} \wedge \neg H_{246}$ (b) $\neg H_{71} \wedge H_{238} \wedge H_{246}$

Figure 8.7: Visualizations for combinations of the literals satisfying the rule and its negation respectively

that maximizes multiple literals simultaneously. This allows us to visualize different input patterns which satisfy a rule. By selecting 3 of the literals to satisfy we can generate images that represent optimized inputs that satisfy the rule with these 3 literals. The results of this are shown in figure 8.7 for both the rule and its negation. What we see is that more facial features are becoming apparent. As the literals are combined, more distinct faces are represented.

Although decompositional rule extraction for deep networks remains a difficult problem, final layer rule extraction provides a simple way of understanding the output as a function of abstract input variables. Visualization methods allow one to visualize relevant regions or features for a hidden unit or input example, but they do not account for how these features are combined to come to a conclusion. By combining final layer rule extraction with gradient methods, we hope to understand how the network reasons about input features to come to a conclusion about the label. Although very preliminary, the experiments combining final layer rule extraction and visualization methods seem promising in that they allow one to reason about the way features are used in the network. Further work in this area could result in comprehensive and intuitively understandable explanations for deep networks.

Part III

Conclusion and Future Work

Chapter 9

The Relationship between Symbolic Systems and Neural Networks

9.1 Formal Equivalences

The field of AI has often been divided into symbolic and connectionist approaches. Although the intuition behind each approach is very different, formally, the distinction between a symbolic system and a connectionist one has more to do with practical differences rather than fundamental ones. In practice, the formal definitions of a connectionist and a symbolic system are often unimportant. Rather than represent distinct classes of models with different properties and abilities, the terms serve more as way of describing different approaches taken to a problem. When studying neural-symbolic computing, however, the exact differences between symbolic and connectionist models becomes important. It is important to know exactly what kind of representational or practical benefits you might gain from translating one kind of model to the other. It is also important to know fundamentally whether you *can* translate from one model to the other without a loss of information.

In order to study this problem formally, we developed a framework for relating logical systems to neural networks. In this framework we identified two distinct forms of equivalence between a logical system and a neural network (semantic and syntactic). This definition allowed us to precisely characterize the relationship between different logical systems and classes of neural networks. We showed that stable networks are semantically equivalent to propo-

sitional logic, meaning that every stable neural network can be thought of as representing the semantics of some propositional knowledge base, and every propositional knowledge base has a stable neural network describing its semantics. We noted that the Lowenheim-Skolem theorems show that no neural network can fully represent the complete semantics of first-order logic making the specific approach to neural-symbolic computing of semantic encoding and extraction fundamentally limited. We also showed that feed-forward networks can be syntactically encoded into propositional logic (at least in a limiting sense) meaning that every feed-forward network can be exactly described up to arbitrary precision by propositional logic.

Within the framework we also developed a formal definition of fidelity as well as extensions to the definitions to account for probabilistic networks and multi-valued logical systems. This allowed us to review much of the literature on neural-symbolic integration and contextualize the results of previous equivalence results in terms of our framework. In particular, the semantic equivalence of stable, recurrent, binary networks and penalty logic; the semantic and syntactic equivalence of Horn logic programming and binary feed-forward networks, the semantic and syntactic equivalence of CILP networks and acceptable logic programs along with all of the extensions to those results; and finally the equivalence between Markov Logic and Boltzmann machines. With our framework we are able to collect and organize much of the previous work on neural-symbolic integration in order to provide a clear account of the exact relationships between neural networks and logical systems. A summary of the relationships between different classes of neural networks and logical systems was given in 5.1.

These allow us to conclude several things. The first is that a semantic approach to neural-symbolic computing will not be possible for many important logical systems. Depending on the application, semantic approaches can still be important, but neural networks are fundamentally limited in this respect by their size. The second take away is that any problem that can be solved by a feed-forward network has an equivalent description in propositional logic. This isn't surprising but it highlights the fact that any kind of neural network classifier can easily be thought of as a complicated propositional system. The third take away is that stable networks cannot be used to represent an arbitrary log-

ical system whereas unstable networks, in theory, can[1]. Although equivalences between Turing machines and neural networks have been developed, first and higher order logics have not been fully described with connectionist models. The final and most important take away is that because feed-forward networks are equivalent to stratified logic programs, the difference in the connectionist and symbolic approach must come from practical and representational aspects. In particular, the perceived differences between neural networks and logical systems can be primarily attributed to the existence of efficient learning algorithms along with the complexity of the internal representations of a network. On the latter point, the use of distributed representations in a network is often pointed to as the primary thing that separates neural networks from logical system, but as we will argue in the next section, this can be thought of entirely as an issue of complexity.

9.2 Complexity and Representation

Despite the formal comparisons we can make between neural networks and logical systems, they are generally used in different ways. This gives rise to important representational differences. Although we can create an abstract propositional knowledge base with millions of distinct atomic variables, this is not done in AI. Rather, we assume that the atomic variables correspond to some abstract concept we wish to model. The same is usually true for the use of first-order logic in AI with the relation symbols, function symbols, and constants usually chosen to represent things or properties that are abstract enough to be generally applicable to multiple contexts. This is not a theoretical property of logical systems, but more one of convention. By contrast, neural networks consist precisely of variables which do not correspond to any predefined concept or notion. Rather we trust that, with a sufficiently large network and dataset, the learning algorithm will be powerful enough to develop a system which behaves the way we want. The resulting network will be formally equivalent to a complicated propositional knowledge base with the neurons identified by atomic variables. The exception to this of course is for unstable continuous networks for which the relationship is more complicated.

Although neural networks can model problems on a test set, the fact that

[1]This is because the class of stable networks is not Turing complete, due to their inability to implement non-terminating computations, while unstable networks are

it is unclear what kind of concepts a single neuron represents means that we are left uncertain as to whether or not a neural network has managed to find a truly general solution to the problem, or whether it is using a more brute force approach. In the later case, the network will fail to generalize to problems from a different, but similar input space. By contrast, logical systems, although technically capable of finding identical solutions to classification problems as neural networks, would require many variables and would be far too complex to design by hand. Learning techniques for logical systems do exist but neural network learning seems to work better for specific kinds of problems such as image recognition. The symbolic approach is to identify a small number of relevant abstract concepts and try to create simple, easy to understand systems in terms of these concepts. The difficulty in applying these systems is that inputs often do not come in the form of abstract entities but rather high-dimensional continuous data. To illustrate the difficulty, consider MNIST digit classification. Although hypothetically we could create a logical system in which individual pixels were atomic variables and all sorts of combinations of rules were included to make the right classification, such a system would be complex and difficult to design. Rather we might want to identify more abstract properties of images that we can reason about such as 'loops' or 'vertical lines'. In order to do this, we would have to automate the recognition of abstract properties which remains difficult to do with logical systems for the reasons described above. Neural Networks, by contrast, implement complicated transformations of the input data and do not necessarily rely on reasoning using abstract features such as loops or lines. The large number of neurons allows them to efficiently capture the relationship between many different variables and use them to make predictions about higher-level features in the data.

In practice Neural Networks operate at a lower level of abstraction than logical systems. Despite this, in order to properly generalize we still want networks to make use of the same abstract reasoning as before but with more flexibility. A network which has learned to reason about a number in terms of different shapes will be able to learn how to reason about letters more simply because the problem involves many of the same abstract concepts. This is the goal of neural-symbolic computing, to use connectionist and symbolic techniques in order to develop systems which can use low level subsymbolic reasoning as well as high level abstract reasoning. Rule extraction, an important part of

this process, is a method of asking the question 'has this network learned any useful abstract symbolic reasoning?' As per the previous discussion, abstract in this context must be thought of in terms of complexity. The more complexity we allow in our rule extraction procedure the more closely we get to merely transcribing the neural network into an equivalent logical system. If we are able to find a small number of atomic variables corresponding to certain activations in a network that can predict the output of a network with a high fidelity then we have found abstract concepts used by the network, although the exact nature of these abstract concepts may still be unknown.

So if these abstract concepts are there, where might they be hiding? One possibility, which was the main research topic of this thesis, is that the hierarchical nature of deep networks puts abstract concepts closer to the top of the network. The intuition behind deep learning has long been that lower layers represent simple low-level concepts and that higher layers represent more complex features [2]. If this is the case, then the higher layers of a deep network are a good place to look for features that are abstract enough to be able to explain the reasoning of a network fairly easily. This is what we investigated and the results will be summarized in the next section. Another possibility is that the abstract concepts correspond to distributed patterns of activation across a variety of neurons. In this case an abstract concept is equivalent to certain complicated combinations of low-level concepts. This idea was not fully explored but we will discuss it when talking about future work. Now we will look at the results of our experiments with layer-by-layer rule extraction to see if there is anything to the idea that we can find abstract concepts residing in higher layers.

9.3 Modular Explainability with Rule Extraction

In our experiments we examined the idea of layer wise rule extraction from deep networks using a new *M-of-N* extraction algorithm. We wanted to see if the intuitive idea that higher layers represented more abstract features translated into less complex rules. As discussed in the previous section, a network which

[2] simple in the colloquial sense, a circle is thought of as simpler than a face yet describing a circle in terms of pixel values of the image is a more complex task than describing a face in terms of its facial features

reasons using abstract concepts should theoretically be explainable with fairly simple rules. We found that there is no clear pattern between the depth of a layer and explainability using our rule extraction algorithm. Instead higher layers may be more or less complex than lower layers. Similarly the size of a layer could not completely predict how relatively complex a symbolic explanation must be. We did, however, find that the final layer could be explained with much simpler rules than the previous layers. This suggests that the neurons in the final layer represent abstract, and potentially more identifiable concepts than the previous layers. The pattern of highly a highly explainable final layer was present across several datasets but not when the training procedure was changed. This suggests that the explainable final layer is primarily the result of the learning algorithm. There were also some differences found when changing the transfer function but the overall pattern of explainability between layers was not altered.

Next we used the previous results to produce modular explanations of deep networks in which the neurons in the final hidden layer are explained via visualization techniques and the network behaviour is explained using a rule extracted from the final layer. We used two different visualization techniques for this, one local (LIME), and one global (keras-vis). Combining these visualizations with rule extraction, we were able to produce explanations of the network that were more detailed than those produced with the visualizations alone, although the visualized features using gradient methods remain indistinct. Seeing how features are combined by the network cannot be done with visualization techniques alone, and rule extraction alone cannot be used to understand the features of a deep network because the explanations will either be too complex or not accurate enough. By combining these two approaches we can overcome the deficiencies of both and create a new kind of explanation which gives a more nuanced view into the inner workings of a neural network.

9.4 Summary

The contributions of the thesis and results are summarized here. The theory, algorithms, and techniques developed are as follows

- Foundations for Neural-Symbolic relations were established with several properties and limitations identified.

- A fast decompositional *M-of-N* extraction algorithm for DBNs was developed.
- A slow *M-of-N* extraction algorithm that is capable of directly controlling the complexity of extracted rules was developed.

The experimental results and theoretical properties of networks derived from the previous algorithms and techniques are as follows

- Relationships between propositional logic and stable networks were established. Including the complete description of feed-forward networks semantically and syntactically with propositional logic up to a arbitrarily small approximation error for continuous-valued networks.
- Previously established neural-symbolic relationships were put into the context of our theory and, in combination with the previous point, the relationship between many classes of neural networks and logical systems were elucidated in table 5.4.
- Using the fast algorithm *M-of-N*, extraction was shown to improve on the accuracy of the previous optimal confidence algorithm in RBMs.
- The slow *M-of-N* algorithm was compared with CORELS in small feed-forward networks on categorical data and found to extract rules of a similar complexity and accuracy.
- The slow *M-of-N* algorithm was applied to feed-forward CNNs trained on image data and found that, like the categorical networks, the final layer of a feed-forward network can be accurately explained with relatively simple *M-of-N* rules.
- The results from the previous experiment also suggested that the internal layers of CNNs could only be adequately explained either by using distributed representations in rule extraction, using finer partitions of the activation values, or with very complex *M-of-N* rules.
- Proof of concept experiments were done with both a global and local visualization method in order to demonstrate how rule extraction from the final layer of a network can be combined with other techniques in order to provide a hybrid end-to-end explanation of the network which is able to explain the network in more detail than either method alone can.

Chapter 10

Future Work

10.1 Rule Extraction For Robustness

We also studied the possibility of rule extraction as a tool for robustness. Despite neural networks being able to achieve very high accuracy on a test set, in many cases it is possible to engineer *adversarial examples*. These are examples which are designed to fool the network while still being easy to classify by humans. Not only does this present possible security threats, but it also reveals that even networks with super-human accuracy on their test sets may not be developing the same concepts that allow humans to reason generally. The existence of adversarial examples once again raises the tension between symbolic and connectionist systems. Recall that logical systems reason in terms of abstract concepts whereas connectionist systems rely on large amounts of data and computational power. While logical systems attempt to model logical reasoning, connectionist ones often model intuitive reasoning. Because the validity of abstract reasoning is invariant under various representations of the same class, symbolic systems are immune to adversarial attacks. Connectionist systems, on the other hand can be influenced, albeit in a small way, by small perturbations even if the variables being perturbed are overall insignificant. The more philosophical question underlying rule extraction is whether or not neural networks, and especially deep neural networks, learn to use abstract features in their reasoning process. Naturally a neural network that represents an abstract feature useful for logical reasoning will do so in a fuzzy way, with several other activation patterns approximating it. Nonetheless, the use of abstract features should endow the neural network with a similar robustness to symbolic systems. As long as perturbations to an input do not change the nature of the abstract features contained within, the neural network's decision

should remain unchanged. However, the vulnerability of neural networks to adversarial attacks suggests that if neural networks are learning abstract features, they are not the same ones used by humans. Although in many cases adversarial attacks can be overcome by adding a regularization function to specifically address the attack in question, the sheer number of attacks seems to indicate that the representations formed by most neural networks suffer from some deeper issues. Fixing a neural network to account for some particular deficiency while it remains vulnerable to others misses the point that human beings are robust against all sorts of adversarial attacks without having specific training for any of them (unless you consider optical illusions adversarial attacks).

With the apparent ability of final layer rule extraction to provide simple and accurate descriptions for deep neural networks, we hypothesized that the more compact *M-of-N* rules we extracted may be more robust to adversarial examples as they force a network to rely on a small set of more abstract features. Even if the majority of the input neurons to the final layer play no significant role on the training set, if pushed in the right direction collectively they can override the influence of the relevant features. Using a similar procedure to the previous section, we extract rules from the final layer of the CNN trained on the Olivetti faces dataset for different values of β. We form a hybrid model by using the network to generate activation values for its final layer given some input before using those values as inputs to our extracted rules to produce a result.

In order to test the effect of final layer rule extraction on robustness, we use the foolbox package Rauber et al. [2017] in python to generate adversarial examples for our CNN trained on the Olivettie faces dataset using three different attacks. Using FGSM Goodfellow et al. [2015] and LBFGS Szegedy et al. [2014] we were able to generate 400 adversarial examples each. We also tested for adversarial examples with the one-pixel attack Su et al. [2019], a much weaker attack that attempts to fool the network by only changing a single pixel value. Using this attack we were able to generate an additional 6 adversarial examples. As we can see in Table 10.1, final layer rule extraction seems to provide a small, but consistent resistance to adversarial attacks. Somewhat surprisingly, simpler rules seem to be slightly more vulnerable in general. Although these preliminary experiments don't solve the problem of adversarial attacks, they

Adversarial Attack	$\beta = 0$	$\beta = 0.1$	$\beta = 0.2$	$\beta = 1$
LBFGS	82%	82.25%	83%	85.75%
FGSM	87.5%	89.25%	89.25%	89.75%
One-Pixel	33%	33%	16.7%	50%

Table 10.1: Error percentage on adversarial examples on a CNN trained on the Olivetti faces dataset using final layer rule extraction for various complexity penalties

do seem to indicate that the high dimensionality of neural networks is at least partially responsible for their weakness and that rule extraction can at least provide a modest benefit.

In order to confirm that this trait isn't unique to the CNN trained on Olivetti faces, we also applied FGSM to a CNN trained on MNIST in order to generate 1000 adversarial examples. In Table 10.2, we can see a similar pattern to that

Average Rule Complexity	**Misclassifications**
0.165	62.4%
0.056	66.9%
0.044	68.3%
0.017	68.7%
0.004	80.8%

Table 10.2: Number of misclassifications by rules of various complexities extracted from a CNN on 1000 adversarial examples

of the Olivetti faces CNN with the more complex rules appearing to be more robust than the simpler ones. Furthermore, using this network, final layer rule extraction improves robustness to a much larger degree than in the previous case. From these results it is clear that rule extraction will not provide an answer to adversarial results on its own, however, the fact that the increase in robustness seems to be independent of the attack suggests that the same fundamental flaw underpins many different adversarial attacks and this flaw may be related to the failure of neural networks to capture meaningful abstract data. This observation further solidifies the difficulties of rule extraction but at the same time highlights the importance of neural-symbolic computing to develop truly intelligent computational systems.

10.2 Improving the Optimality of the Extraction Algorithm

The main challenge for a rule extraction algorithm is the fact that the number of possible rules is exponential. For this reason, extraction algorithms always rely on a simplifying assumption to make a search tractable. The downside of this is that you may end up ignoring more optimal rules. CORELS uses theoretical results to produce verifiably optimal rules with only a limited search of the rule space. However, CORELS is still quite slow and becomes impractical for large networks and datasets. Our algorithm makes some simplifying assumptions along with a parallel search which allows it to be applied to much larger networks, but at the cost of not being completely optimal. We demonstrated empirically that the rules found by CORELS are only marginal improvements over our algorithm, but improvements to the *M-of-N* search procedure could be made. In particular, some of the properties of *M-of-N* rules discussed in the next section could potentially be used to further reduce the search space without excluding relevant rules. The fact that an *M-of-N* rule has the same complexity as an *(N-M)-of-N* rule is one example of a property that could be used to reduce the search space. The other big issue is the choice of splits. In our current algorithm we use the information gain to choose splits independently, however, because input variables are not necessarily independent, the choice of an optimal split might change depending on which rule the literal is being added to. CORELS and other extraction algorithms generally assume that splits have already been chosen, but many other data mining techniques can be used to produce a good set of candidate splits. The implementation of these techniques could further improve the rules found by our algorithm. Splitting neurons into multiple atomic variables by choosing more than one splitting value is another option for improving the fidelity of extracted rules but it comes at the cost of increased complexity due to the larger number of variables. Experiments looking at the optimal rules extracting using different numbers of splitting values could provide insight into how finely the state space should be partitioned in order to extract the rules with the best explainability.

Another approach to rule extraction that might make a drastic change in the explainabililty is the consideration of distributed representations in the network. The vast majority of rule extraction algorithms use individual neurons

as prototypes for logical variables. However, as we discussed, there is good reason to think that if a network does use abstract reasoning, then the instantiation of an abstract concept is represented as a pattern of activity across a collection of neurons. If this is the case then some layers within the network might be much more explainable than they were in our experiments. We ran some preliminary tests for this concept by first applying PCA to a hidden layer before applying our rule extraction algorithm using the principal components as the atomic variables. This has the additional advantage in that our extraction method now becomes optimal (see appendix A.4). Our results were intriguing, but inconclusive. Notably, the unexplainable final layer in the deep autoencoder became much more explainable, but the already explainable final layers in the other networks become unexplainable with accuracy-complexity graphs resembling that of the final layer of the autoencoder. PCA essentially switched the explainability graphs of the final layer in the autoencoder and the networks trained with end-to-end backpropagation. The changes to other layers were for the most part insignificant. Other whitening techniques could be applied as well as more general transformations which attempt to find the combinations of different input patterns which best explain the output. One possible approach is the following. Use an extraction algorithm to produce a rule whose atomic variables are defined by the activation values of single input neurons. Then, consider a linear transformation between the input variables and a new set of identical units. Initialize the linear transformation to the identity. Replace the final layer of the network with the extracted rule. Use gradient descent on the new linear transformation to maximize the fidelity of the rules to network. After this, apply the rule extraction algorithm again to produce a new set of rules and repeat the process for a fixed number of steps or until some margin of error is reached. This is one possibility, other techniques may be developed which encode patterns of activity into abstract logical variables.

The potential of distributed representations to play an important role in the abstract reasoning of a network means that our results are not conclusive regarding the use of abstract concepts in deep neural networks. What can be said, is that if deep neural networks use abstract concepts, they are either only present in the final layer, they are represented by distributed patterns of activity in lower layers, or that they simply cannot be represented with *M-of-N* rules and binary partitioning of the state space.

10.3 Extracting Hierarchical Rules

In our experiments we were only be concerned with rule extraction from a single layer in a neural network. However, we will provide a description for how these rules can be combined into a hierarchical explanation of a deep network. When extracting implications from a deep network, the rules are naturally hierarchical. The variables at the head of one set of rules are the variables in the body of the next set and so on. If the rules we produce are implications without any confidence values, the only issue preventing us from chaining together *modus ponens* to reach a conclusion about the output is that the discretization method we used for continuous networks does not ensure that a neuron when treated as and input produces the same atomic variable as it does when treated as an output. In order to make inferences with hierarchical rules, it is essential to make sure that each neuron was discretized the same way in each layer it appears in. For example, if you define a propositional variable X_i by $X_i = \top$ iff $x \geq 0.5$ in one layer, say the layer for which x_i is an output variable, and then in the next layer x_i corresponds to a variable $\hat{X}_i$ for which $\hat{X}_i = \top$ iff $x_i \geq 0.7$ then you cannot simply compose the rules from the second layer with rules from the first. Assuming all variables are consistent, given an assignment of truth values to the atomic variables corresponding to the input neurons, one can repeatedly apply *modus ponens* and the closed world assumption to produce a configuration of the atomic variables corresponding to the neurons in the next layer. This assigns a value of 1 to each variable which is at the head of a body that is satisfied, and 0 to each variable which is not. Repeating this process we end up with a configuration of all atomic variables and thus a full explanation of the network.

When confidence rules are involved this process needs to be adjusted to account for them. Suppose we have a rule $40 : h \leftarrow x_1 \wedge x_2$. If we have $10 : x_1$ and $10 : x_2$, what should the confidence for h be? Luckily we can generalize *Modus Ponens* to confidence rules in a way that propagates confidence values up through the layers of a deep networks Tran and Garcez [2016]. In order to do this, suppose we have a confidence rules, $c : R$, with a given set of input neurons. Suppose further that some of these inputs neurons have assigned confidence values in the range $[min, max]$. First assign the confidence value $\frac{min+max}{2}$ to any input neurons which were not assigned a confidence value. Then, in order to determine a confidence value for the output neuron, add the

confidence values of the positive literals, subtract the confidence values of the negative literals, and multiply by c. Repeating this process layer-by-layer gives us a method of calculating confidence values for the outputs of a hierarchical confidence rule. If our rule is an *M-of-N* rule, we can generalize this by using the sum (and difference) of confidence values corresponding to the M inputs with highest confidence. Other possible schemes exist and many would most likely have to be tried in order to find an effective one.

Whether or not confidence rules are used specifically, hierarchical *M-of-N* rules may be useful in the case that multiple layers of a neural network are explainable with rule extraction. In our experiments, we found that generally only the final layer was explainable. However, it is possible that for some networks, or if the extraction algorithm is improved, the final two or final three layers can be explained with rule extraction. When this is the case, the modular approach to explainability will necessarily call on hierarchical rule extraction to provide an abstract symbolic explanation of the final layers.

When moving from the extraction of basic *M-of-N* rules to hierarchical *M-of-N* rules, simplification techniques become important. Some attempts have been made to develop operations on *M-of-N* rules that would allow one to remove redundancies in order to make rules more compact and hopefully more interpretable.*M-of-N* rules equipped with these operations were called the X-algebra although the research was never carried to fruition Broda and Garcez [2001]. We conclude this chapter with a brief discussion of the X-algebra including some potentially useful operations and identities. In the X-algebra all terms are *M-of-N* rules. The literal X is shorthand for the rule $\binom{\{X\}}{1}$ and $\neg X$ is shorthand for the rule $\binom{\{\neg X\}}{1}$. α_i, and β_i will be used to represent arbitrary *M-of-N* rules. An *M-of-N* rule that contains only literals will be referred to as basic and one that contains other *M-of-N* rules will be referred to as complex.

First, given *M-of-N* rules $\binom{S_1}{M_1}$ and $\binom{S_2}{M_2}$ (here each S_i is a set of literals or other *M-of-N* rules), $\binom{S_1}{M_1} \implies \binom{S_2}{M_2}$ if and only if for every M_1 clauses in S_1, at least M_2 clauses in S_2 are contained in the set of implications of those M_1 clauses. For basic *M-of-N* rules, since for literals X_i and X_j, $X_i \implies X_j$ if and only if $X_i = X_j$ it is not hard to prove that $\binom{S_1}{M_1} \implies \binom{S_2}{M_2}$ if and only if $M_1 - |S_1 \setminus S_2| \geq M_2$.

In addition to conjunction and disjunction, which can be defined using *M-of-N* rules as shown in section 3.3, negation can be defined by $\neg\binom{\alpha_1,\alpha_2,\ldots,\alpha_N}{M} = \binom{\neg\alpha_1,\neg\alpha_2,\ldots,\neg\alpha_N}{N-M+1}$. We also introduce a new operation called *dual*.

Definition 10.3.1. *dual is an operator on complex M-of-N defined recursively by*

1. $dual(X_i) = X_i$
2. $dual(\neg X_i) = \neg X_i$
3. $dual(\binom{\alpha_1,\alpha_2,\ldots,\alpha_N}{M}) = \binom{dual(\alpha_1),dual(\alpha_2),\ldots,dual(\alpha_N)}{N+1-M}$

Where X_i is a literal. For simple *M-of-N* rules, dual simply changes the value of M to $N+1-M$. Extending the definition to complex *M-of-N* rules is done by recursively applying it to its set of clauses. The following properties of the X-algebra are easy to show.

Proposition 10.3.1. *Given M-of-N rules α and β. The dual operator satisfies the following properties*

1. $dual(\alpha \vee \beta) = dual(\alpha) \wedge dual(\beta)$
2. $dual(\alpha \wedge \beta) = dual(\alpha) \vee dual(\beta)$
3. $dual(\top) = \bot$
4. $dual(\bot) = \top$
5. $dual(\neg\alpha) = \neg dual(\alpha)$

The use of the above identities could potentially be incorporated in an extra step in the extraction of hierarchical *M-of-N* rules. After extracting rules from one layer and before moving on to extraction from the next layer, apply identities in a systematic way to reduce redundancy and complexity of the rules. As discussed previously, results suggest that full end-to-end hierarchical rules may be constrained by complexity. For this reason, it may be more fruitful to develop the modular approach to explainability.

10.4 Refining Modular Techniques

The development of modular explainability techniques opens up a huge amount of potential for explainable AI. Although our use of modular techniques was

limited to our own rule extraction algorith, any rule extraction algorithm could just as easily be used. Additionally, many different visualization techniques exist all of which could be used in a modular fashion. Moving on to different, possibly larger networks might mean having to switch to different, faster, extraction algorithms. For example our fast *M-of-N* extraction algorithm could be adapted in order to produce modular explanations for very large networks such as imagenet. The subjective nature of visualization means that all the various techniques and methods could be tested until reasonable visualizations are produced. In some cases, lower layers in a deep network might be more explainable and hierarchical rules for a segment of the network might be extracted. The general rule for modular explanation is that the extracted rules should come after the visualized features, in other words, if we can find accurate and simple rules to explain the final two layers in a network then we can use rule extraction to produce a hierarchical set of rules and use visualization techniques to explain the input features of these rules. On the other hand, if the first layer is explainable but the second layer is not, then rules extracted from the first layer are unlikely to be effectively used in conjunction with visualization.

Appendices

Appendix A

Proofs

A.1 Proof of Theorem 4.4.2

Assume we have two networks, N_1 and N_2, with corresponding state spaces X_1 and X_2, along with a bijective map, f, from the state space of N_1 to the state space of N_2 such that $f(N_1(x)) = N_2(f(x))$. First we deal with the syntactic case

We assume that N_2 is a syntactic model of $\mathcal{S} = (\mathcal{L}, \vdash_{\mathcal{S}}, \mathcal{M})$. This implies that there is an injective map, $i : X_2 \to \mathcal{L}$ such that for all $x \in X_2, i(x) \vdash_{N_2} l_0 \implies i(x) \vdash_{\mathcal{S}} l_0$. Define a mapping $e : X_1 \to 2^{\mathcal{L}}$ given by $e(x) = i(f(x))$. First we show that e makes N_1 a syntactic model of $\mathcal{S}$. By injectivity of i and f, e is injective. Take any state $x \in X_1$. And consider $e(x) = i(f(x))$. Suppose that $e(x) \vdash_{N_1} l_0$, then there exists $t > 0$ such that $l_0 \in e(N_1^t(x))l_0$. By definition this is $i(f(N_1^t(x))$. Which is equivalent to $i(N_2^t(f(x))$ by iterating the assumption on f. Because we have $l_0 \in i(N_2^t(f(x))$ this gives us by definition that $e(x) = i(f(x)) \vdash_{N_2} l_0$ and thus by the fact that i is a symbolic encoding of N_2 we have that $e(x) \vdash_{\mathcal{S}} l_0$. And thus N_1 is a syntactic model of $\mathcal{S}$

Now suppose that i is a syntactic encoding of N_2 into $\mathcal{S}$. By definition this means that for all $x \in X_2$, $i(x) \vdash_{\mathcal{S}} l_0 \implies i(x) \vdash_{N_2} l_0$. We want to show that e also defines a syntactic encoding of N_1 into $\mathcal{S}$. Suppose we have that $e(x) \vdash_{\mathcal{S}} l_0$. The knowledge base $e(x)$ is encoded by the state $f(x) \in X_2$ by i and thus again by the fact that i is a symbolic encoding we have that $i(x) \vdash_{N_2} l_0$. So there exists $t > 0$ such that $l_0 \in i(N_2^t(f(x))$ but again by assumption on f this is equivalent to saying $l_0 \in i(f(N_1^t(x)) = e(N_1^t(x))$ so we have $e(x) \vdash_{N_1} l_0$. Thus e is a symbolic encoding of N_1 into $\mathcal{S}$.

Now suppose that i defines a neural encoding of a set of knowledge bases $Q \subset 2^{\mathcal{L}}$ into N_2. That is for all $L \in Q$, there exists $x \in X_2$ such that $i(x) = L$ and $i(x) = L \vdash_{\mathcal{S}} l_0 \implies L \vdash_{N_2} l_0$. We want to show that e also defines a neural encoding of Q. First, because f is bijective, then if $L = i(x)$ for some $x \in X_2$ we have that $L = e(f^{-1}(x))$. Furthermore, if $L \vdash_{\mathcal{S}} l_0$ then we have $L \vdash_{N_2} l_0$ and thus there exists $t > 0$ with $l_0 \in i(N_2^t(x)) = i(N_2^t(f(f^{-1}(x)))) = i(f(N_1^t(f^{-1}(x)))) = e(N_1^t(f^{-1}(x)))$ so we have $L \vdash_{N_1} l_0$ and thus e defines a neural encoding of Q.

Moving on to semantic encodings, assume that N_2 is now a semantic model of $\mathcal{S}$ via the map i. Again define $e = i \circ f$. By injectivity of i and f, this makes N_1 a semantic model of $\mathcal{S}$. We will show that $L_0 \vDash_{N_1} L$ iff $L_0 \vDash_{N_2} L$. This will imply that if N_2 is a neural encoding of L_0 then so is N_1 because $L_0 \vDash_{N_1} L \to L_0 \vDash_{N_2} L \to L_0 \vDash_{\mathcal{S}} L$ and if L_0 is a symbolic encoding of N_2 then it is also a symbolic encoding of N_1 for the same reason. We do this by showing that if N_2 is a model for L_0 then so is N_1 and that the L_0-models of N_2 are identical to the L_0-models of N_1.

First note that because $e(f^{-1}(x)) = i(x)$, x and $f^{-1}(x)$ map to the same model in $\mathcal{M}$. Suppose that N_2 is a model of a knowledge base $L_0 \in \mathcal{L}$. Take any $x_0 \in X_1$. Because N_2 is a model of L_0, there exists $t_0 > 0$ such that for all $t > t_0$, $N_2^t(f(x_0))$ is a model of L_0. Consider $e(N_1^t(x_0)) = i(f(N_1^t(x_0))) = i(N_2^t(f(x_0)))$ thus $N_1^t(x_0)$ is a model of L_0 for all $t > t_0$ and therefore N_1 is a model of L_0. Now suppose x is an L_0-model of N_2. We have already shown that $f^{-1}(x)$ represents the same model of L_0, now we must show that it is an L_0-model of N_1. By definition of L_0-model, there exists an initial state $x_0 \in X_2$ such that for all $t > 0$, there exists $t' > t$ such that $N_2^{t'}(x_0) = x$. We have $x = N_2^{t'}(x_0) = N_2^{t'}(f(f^{-1}(x_0)) = f(N_1^{t'}(f^{-1}(x_0)))$ meaning $f^{-1}(x) = N_1^{t'}(f^{-1}(x_0))$. Thus for all $t > 0$ there exits $t' > t$ such that $N_1^{t'}(f^{-1}(x_0)) = f^{-1}(x)$ meaning $f^{-1}(x)$ is an L_0-model of N_1 so the L_0-models of N_2 are a subset of the L_0-models of N_1. Now suppose x is an L_0-model of N_1, again we know that $f(x)$ corresponds to the same model of L_0 so we merely need to show that $f(x)$ is an L_0-model of N_2. By definition, there exists $x_0 \in X_1$ such that for all $t > 0$ there exists $t' > t$ with $N_1^{t'}(x_0) = x$. We have $f(x) = f(N_1^{t'}(x_0)) = N_2^{t'}(f(x_0))$ and so for all $t > 0$, there exists $t' > t$ such that $N_2^{t'}(f(x_0)) = f(x)$ meaning $f(x)$ is an L_0-model of N_2. Thus N_1 and

N_2 are models of the same knowledge bases and, for each knowledge base L_0, contain the same L_0 models meaning $L_0 \vDash_{N_1} \iff L_0 \vDash_{N_2} L$ □

A.2 Proof of Proposition 7.2.1

Let's say we have a distribution P derived from an RBM N and a biconditional $h \leftrightarrow ANT$. Where ANT is the antecedent of the rule. We can write the probability as

$$\begin{aligned} P(h \leftrightarrow ANT) &= P(h, ANT) + P(\neg h, \neg ANT) \\ &= \sum_{x \in ANT} P(h, x) + \sum_{x \notin ANT} P(\neg h, x) \\ &= \sum_{x \in ANT} P(h|x)P(x) + \sum_{x \notin ANT} (1 - P(h|x))P(x) \end{aligned}$$

Give an biconditional rule r with head h and body ANT, define $\xi_{r,x} = P(h|x)$ if $x \in ANT$ and $\xi_{r,x} = 1 - P(h|x)$ if $x \notin ANT$. then the previous equation is written simply as

$$P(r) = \sum_x \xi_{r,x} P(x) \tag{A.1}$$

RBMs are universal approximators, furthermore given a network N we can construct another network N' which doesn't alter the parameters for the existing hidden units but changes the visible distribution to be arbitrarily close to any chosen distribution. Effectively this means we can find a network N' with each $\xi_{r,x}$ identical to N but with arbitrary values in $(0, 1)$ chosen for each $P(x)$. With this in mind, let's compare the probabilities of two extracted rules from an algorithm which relies only on local data.

Suppose we extract $c_1 : r_1$ and $c_2 : r_2$ from network N with $c_1 > c_2$. Then if we assume that the extraction algorithm preserves the probabilites we have $P(r_1) > P(r_2)$. referring back to equation A.1 we can see that if $\xi_{r_1,x} > \xi_{r_2,x}$ for all x then we will have $P(r_1) > P(r_2)$ regardless of the visible distribution. However, if there is a single configuration, x_0, with $\xi_{r_1,x_0} < \xi_{r_2,x_0}$, by the universal approximation properties of RBMs, we can find another network N' for which all the values of ξ are unchanged but $P(x_0) \approx 1$ and $P(x \neq x_0) \approx 0$. We can make this approximation to arbitrary precision so we can define the network such that $P(r_1) < P(r_2)$ but since the values of ξ are unchanged we must still have $c_1 > c_2$.

This means that if a local extraction rule is to preserve the probability with the confidence it must only extract confidence rules $c_1 : r_1$ with $c_2 : r_2$ with $c_1 > c_2$ if for all x, $\xi_{r_1,x} > \xi_{r_2,x}$. This is quite a restrictive condition in general. In particular the optimal confidence algorithm described does not satisfy the condition so we can find networks which extract rules that do not preserve the probability.

A.3 Proof of Proposition 7.3.1

given a biconditional $h \leftrightarrow x_1, ..., x_k, \neg x_{k+1}, ..., \neg x_n$ we denote the configuration of the visible units where the antecedent, $x_1, ..., x_k, \neg x_{k+1}, ..., \neg x_n$, is true by ANT and the set of configurations of visible units where the antecedent is not true by $\neg ANT$. Then we have $P(h \leftrightarrow x_1, ..., x_k, \neg x_{k+1}, ..., \neg x_n) = P(h = 1, ANT) + P(h = 0, \neg ANT)$. Now consider the previous example with n nodes of identical weights a. Using some algebra, the probability of the biconditional being true in the network can be written as

$$P(h = 1|ANT)P(ANT) + \sum_{i=0}^{n-1} \binom{n}{i} (1 - P(h = 1|ANT^{\neg n}))P(ANT^{\neg n})$$

Where $P(h = 1|ANT)$ is the probability of the neuron being on when the rule is satisfied and $P(h = 1|ANT^{\neg n})$ is the probability of the neuron being on when exactly n literals of the rule are not satisfied, since all the weights are the same this does not depend on which specific literals are not satisfied. Furthermore we are assuming that this visible units are taken from a uniform distribution so we have $P(ANT) = P(ANT^{\neg n}) = \frac{1}{2^n}$. This gives us

$$\frac{1}{2^n} \left(\sigma(an) + \sum_{i=1}^{n-1} \binom{n-1}{i} (1 - \sigma(ia)) \right)$$

Since a and n are arbitrary we can take them to be as large as possible, in which case the limit of the right term goes to $1 - \sigma(0) = 0.5$ and the left hand term goes to 1 so as $n \to \infty$ the whole thing goes to 0. This shows we can extract rules with arbitrarily high confidence but arbitrarily low probability $\square$

A.4 Proof of Theorem 8.4.1

Suppose our input set is of the form $I = [l, u]^n$ with $0 < l < u$. For simplicity we assume $l > 0$ but the proof carries through with a few minor tweaks if not.

Define $Vol(I)$ as the volume of I using the standard Lesbegue measure, in this case $Vol(I) = (u-l)^n$. Assume also that for each input neuron, x_i, we have a split $s_i \in [a,b]$ defining a propositional variable X_i as described in section 4.4 along with a split s_h defining a propositional variable H for the output neuron in the same way. Given a rule, R, relating the input variables and output variable, The error is given by

$$E(R) := \frac{1}{Vol(I)} \int_I |R(x) - N(x)| dx$$

The binary case is similar except that the propositional variables are directly defined by the binary state of the neuron, the integral is replaced with a summation, and $Vol(I)$ is simply the cardinality of I. In either case I can be partitioned into 4 subsets, I_{tp}, I_{tn}, I_{fp}, I_{fn} representing the true positives and negatives and the false positives and negatives of the rule respectively. In other words $I_{tp} = \{x : R(x) = N(x) = 1\}$, $I_{tn} = \{x : R(x) = N(x) = 0\}$, $I_{fp} = \{x : R(x) = 1, N(x) = 0\}$, $I_{fn} = \{x : R(x) = 0, N(x) = 1\}$. The error can then be decomposed as $E(R) = E_{fp}(R) + E_{fn}(R)$ where $E_{fp}(R) := \frac{Vol(I_{fp})}{Vol(I)}$ and $E_{fn}(R) := \frac{Vol(I_{fn})}{Vol(I)}$.

Consider two *M-of-N* rules which differ only by containing a single different literal. In other words, two rules R_1 and R_2 with $R_1 := \binom{X_l \cup (N-1)}{M}$ and $R_2 := \binom{X_r \cup (N-1)}{M}$ where X_r and X_l are the two different literals and $(N-1)$ is the set of literals common between the two rules. We wish to compare $E(R_1)$ and $E(R_2)$ in the case that $|w_l| > |w_r|$. Note that in order to compare the error of two rules we only need to look at the error on the set of inputs for which the rules have different outputs. We start with the binary case.

Let h_{min} be the minimum possible value of h in other words $h_{min} = \sum_{w_i : w_i < 0} w_i + b$. For a given configuration of input values we can write the total input to h as $h_{min} + \sum_{i=0}^{n} |w_i| s(x_i)$ where $s(x_i) = x_i$ if $w_i > 0$ and $s(x_i) = 1 - x_i$ otherwise.

Consider every possible input configuration. If $s(x_l) = s(x_r)$ then R_1 and R_2 have the same output so the number of false positives and false negatives on this set is equal and thus the error over this set is equal. Consider then the set of input configurations where $s(x_l) \neq s(x_r)$. Take any input sequence in this set. If fewer than $M-1$ literals of the shared literals are satisfied that neither R_1 or R_2 predicts 1. If M are satisfied than both R_1 and R_2 predict 1. The only configurations for which R_1 and R_2 have different predic-

tions are those in which exactly $M-1$ of the shared literals are satisfied and $s(x_l) \neq s(x_r)$. We can pair all of these configurations by mapping a configuration with $s(x_r) \neq s(x_l)$ to the configuration in which all values of x_i are unchanged but the values of $s(x_r)$ and $s(x_l)$ are swapped. The difference between the total input to h in the case where $s(x_l) = 1$ and $s(x_r) = 0$ and the case where $s(x_l) = 0$ and $s(x_r) = 1$ is simply $|w_l| - |w_r|$ which is non-negative since $|w_l| > |w_r|$ by assumption. Thus if the total input for a configuration in which $s(x_r) = 1$ and $s(x_l) = 0$ is greater or equal to the threshold t then the same is true for the paired configuration. Similarly if the total input of a configuration in which $s(x_r) = 0$ and $s(x_l) = 1$ is less than t, the same will true for the paired configuration. This means that for every configuration which is a false positive for R_1 there is a configuration which is a false positive for R_2 and for every configuration which is a false negative of R_1 there is a configuration which is a false negative of R_2 meaning that $E(R_1) \leq E(R_2)$. By induction it is easy to see that this implies that given N literals, the optimal *M-of-N* rule contains the N literals with the highest weight.

The same argument can be adapted for the continuous case assuming the equivalent homogeneity conditions (ie the input space is of the form $[a, b]^n$) if the splits for each input variable are identical. If the splits are different this may no longer hold. What can be said is that if the splits for two variables x_i and x_j are within a certain distance, ϵ, of each other and $|w_i| > |w_j|$ then the rule adding w_i will be more accurate. Getting an exact estimate of ϵ depends highly on the difference between $|w_i|$ and $|w_j|$. What makes a useful estimate of ϵ even more difficult is that it depends highly on the weights and splits of the other variables as well. For this reason it is difficult to produce a useful estimate of ϵ without specifying a particular network and input space.

Glossary

ANN Artificial Neural Network. 6

atom A propositional variable or grounded relational term. 25

CILP Connectionist Inductive Logic Programming. 81

clause A conjunction, disjunction, or *M-of-N* collection of literals. 29

CNN Convolutional Neural Network. 22

DBN Deep Belief Network. 20

discretization The transformation of a continuous network to a finite-valued one via a map between state spaces. Also referred to as a binarization when the target network is binary.. 62

feed-forward A neural network with no cycle in its connectivity graph.. 13

KBANN Knowledge-Based Artificial Neural Network. 79

knowledge base A set of sentences in a logical system. 24

literal An atom or the negation of an atom. 29

MLP Multi-Layer Perceptron. 14

model An object which assigns meaning to the language of a logical system. A model of a sentence or knowledge base is an object which evaluates the sentence/knowledge base as true whereas a model in the general sense is any object which is used to define the semantics of a logical system. 24

RBM Restricted Boltzmann Machine. 18

recurrent In reference to neural networks, a neural network which contains a cycle in its connectivity graph. 13

rule An implication with a conjunctive or *M-of-N* clause in the body and an atom in the head. 29

SCN Symmetrically Connected Network. 17

test set A set of examples not used in the training of a neural network but to evaluate its accuracy post-training.. 14

training set A set of examples used along with a learning algorithm to update the parameters of a neural network with the purpose of making the neural network implement some desired computational behaviour.. 14

transfer function A function that computes the output of a neuron given its inputs. Also referred to as a transfer function.. 13

Bibliography

David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985. doi: 10.1207/s15516709cog0901_7. URL https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog0901_7.

Robert Andrews, Joachim Diederich, and Alan B. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6):373 – 389, 1995.

Elaine Angelino, Nicholas Larus-Stone, Daniel Alabi, Margo Seltzer, and Cynthia Rudin. Learning certifiably optimal rule lists for categorical data. *Journal of Machine Learning Research*, 18(234):1–78, 2018. URL http://jmlr.org/papers/v18/17-716.html.

Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade: Second Edition*, pages 437–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_26. URL https://doi.org/10.1007/978-3-642-35289-8_26.

Marko Bohanec and Vladislav Rajkovic. Knowledge acquisition and explanation for multi-attribute decision. In *8th International Workshop on Expert Systems and Their Applications*, 1988.

Guido Bologna and Yoichi Hayashi. A rule extraction study on a neural network trained by deep learning. In *International Joint Conference on Neural Networks*, 2016. doi: 10.1109/IJCNN.2016.7727264.

Krysia Broda and Artur Garcez. the xalgebra. unpublished manuscript, 2001.

Rich Caruana, Yin Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noemie Elhadad. Intelligible models for healthcare: Predicting pneumo-

nia risk and hospital 30-day readmission. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1721–1730. ACM, 2015. ISBN 978-1-4503-3664-2. doi: 10.1145/2783258.2788613. URL http://doi.acm.org/10.1145/2783258.2788613.

Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Springer US, Boston, MA, 1978. ISBN 978-1-4684-3384-5. doi: 10.1007/978-1-4684-3384-5_11. URL https://doi.org/10.1007/978-1-4684-3384-5_11.

Mark William Craven. *Extracting Comprehensible Models from Trained Neural Networks*. PhD thesis, The University of Wisconsin - Madison, 1996. AAI9700774.

Artur d'Avila Garcez, Krysia Broda, and Dov Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125(1):155 – 207, 2001. ISSN 0004-3702.

Martin Davis. Why there is no such discipline as hypercomputation. *Applied Mathematics and Computation*, 178(1):4 – 7, 2006. ISSN 0096-3003. doi: https://doi.org/10.1016/j.amc.2005.09.066. URL http://www.sciencedirect.com/science/article/pii/S0096300305008295.

Leo de Penning, Artur Garcez, Luís Lamb, and John-jules Meyer. Neural-symbolic cognitive agents: Architecture, theory and application. *13th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2014*, 2, 2014.

Olivier Delalleau and Yoshua Bengio. Shallow vs. deep sum-product networks. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 666–674. Curran Associates, Inc., 2011. URL http://papers.nips.cc/paper/4350-shallow-vs-deep-sum-product-networks.pdf.

Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 907–940. PMLR, 2016. URL http://proceedings.mlr.press/v49/eldan16.html.

Dumitru Erhan, Y. Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *Technical Report, Univeristé de Montréal*, 2009.

Melvin Fitting. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, 278:25–51, 2002. doi: 10.1016/S0304-3975(00)00330-3.

Nicholas Frosst and Geoffrey Hinton. Distilling a neural network into a soft decision tree. In *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML*, pages 879–888, 2017.

Artur Garcez. Fewer epistemological challenges for connectionism. *Lecture Notes in Computer Science*, 3526:289–325, 2005. doi: 10.1007/11494645_18. URL http://openaccess.city.ac.uk/296/.

Artur Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Appl. Intell.*, 11:59–77, 1999. doi: 10.1023/A:1008328630915.

Artur S. d'Avila Garcez, Lus C. Lamb, and Dov M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Springer, 1st edition, 2008. ISBN 3540732454, 9783540732457.

Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

Ramanathan V. Guha. Towards a model theory for distributed representations. *CoRR*, abs/1410.5859, 2014. URL http://arxiv.org/abs/1410.5859.

Johan Håstad and Mikael Goldmann. On the power of small-depth threshold circuits. *computational complexity*, 1(2):113–129, 1991. ISSN 1420-8954. doi: 10.1007/BF01272517. URL https://doi.org/10.1007/BF01272517.

Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Wiley, New York, 1949. ISBN 0-8058-4300-0.

Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002. doi: 10.1162/089976602760128018.

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. doi: 10.1162/neco.2006.18.7.1527. PMID: 16764513.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–80, 1997. doi: 10.1162/neco.1997.9.8.1735.

A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952. doi: 10.1113/jphysiol.1952.sp004764. URL https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1952.sp004764.

J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, 1982. ISSN 0027-8424. URL http://view.ncbi.nlm.nih.gov/pubmed/6953413].

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

Paul Horton and Kenta Nakai. A probabilistic classification system for predicting the cellular localization sites of proteins. In *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 109–115. AAAI Press, 1996. ISBN 1-57735-002-2. URL http://dl.acm.org/citation.cfm?id=645631.662879.

Geoffrey Hunter. Metalogic: An introduction to the metatheory of standard first order logic. *Philosophical Books*, 13:12–14, 2008. doi: 10.1111/j.1468-0149.1972.tb03773.x.

Henrik Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation*, 17(6):1223–1263, 2005. doi: 10.1162/0899766053630350.

William H. Kautz. The realization of symmetric switching functions with linear-input logical elements. *IEEE Transactions on Electronic Computers*, EC-10(3):371 – 378, 1961. doi: 10.1109/TEC.1961.5219224.

John F. Kolen. Fool's gold: Extracting finite state machines from recurrent network dynamics. In *Neural Information Processing Systems*, pages 501–508. Morgan Kaufmann, 1994.

Kotikalapudi, Raghavendra, and contributors. Keras-vis, 2017. URL https://github.com/raghakot/keras-vis.

Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. Interpretable decision sets: A joint framework for description and prediction. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1675–1684. ACM, 2016. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939874. URL http://doi.acm.org/10.1145/2939672.2939874.

Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. Interpretable & explorable approximations of black box models. *CoRR*, abs/1707.01154, 2017. URL http://arxiv.org/abs/1707.01154.

Tor Lattimore and Marcus Hutter. No free lunch versus occam's razor in supervised learning. In *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence*, volume 7070 of *LNCS*. Springer, 2011. doi: 10.1007/978-3-642-44958-1_17.

Nicolas Le Roux and Yoshua Bengio. Representational power of restricted boltzmann machines and deep belief networks. *Neural Computation*, 20(6): 1631–1649, 2008. ISSN 0899-7667. doi: 10.1162/neco.2008.04-07-510.

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN 1558-2256. doi: 10.1109/5.726791.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(2):436—-444, 2015.

LiMin Fu. Rule generation from neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(8):1114–1124, 1994. doi: 10.1109/21.299696.

Henry W. Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017.

A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5188–5196, 2015. doi: 10.1109/CVPR.2015.7299155.

David Marker. *Model Theory: An Introduction*. Graduate texts in Mathematics. Springer, 2002. doi: 10.1007/b98860.

John McCarthy. Epistemological challenges for connectionism. *Behavioural and Brain Sciences*, 11(1):44, 1988.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5 (4):115–133, 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL https://doi.org/10.1007/BF02478259.

Hrushikesh Mhaskar, Qianli Liao, and Tomaso A. Poggio. Learning real and boolean functions: When is deep better than shallow. *CoRR*, abs/1603.00988, 2016. URL http://arxiv.org/abs/1603.00988.

Patrick M. Murphy and Michael J. Pazzani. Id2-of-3: Constructive induction of m-of-n concepts for discriminators in decision trees. In *In Proceedings of the Eighth International Workshop on Machine Learning*, pages 183–187. Morgan Kaufmann, 1991.

Chris Percy, Artur Garcez, Simo Dragicevic, Manoel V. M. França, Greg G. Slabaugh, and Tillman Weyde. The need for knowledge extraction: Understanding harmful gambling behavior with neural networks. *Frontiers in Artificial Intelligence and Applications*, 285:974–981, 2016. doi: 10.3233/978-1-61499-672-9-974. URL http://openaccess.city.ac.uk/id/eprint/16483/.

Gadi Pinkas. Reasoning, nonmonotonicity and learning in connectionist networks that capture propositional knowledge. *Artificial Intelligence*, 77(2):203 – 247, 1995. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(94)00032-V. URL http://www.sciencedirect.com/science/article/pii/000437029400032V.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. ISSN 1573-0565.

J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1558602380.

Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017. URL http://arxiv.org/abs/1707.04131.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144. Association for Computing Machinery, 2016. ISBN 9781450342322. doi: 10.1145/2939672.2939778. URL https://doi.org/10.1145/2939672.2939778.

Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-5833-1. URL https://doi.org/10.1007/s10994-006-5833-1.

Frank Rosenblatt. The perceptron, a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323:533–536, 1986. doi: 10.1038/323533a0.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition, 2010.

F. S. Samaria and A. C. Harter. Parameterisation of a stochastic model for human face identification. In *Proceedings of 1994 IEEE Workshop on Applications of Computer Vision*, pages 138–142, 1994. doi: 10.1109/ACV.1994.341300.

Makoto Sato and Hiroshi Tsukimoto. Rule extraction from neural networks via decision tree induction. In *Proceedings of the 2001 International Joint Conference on Neural Networks*, volume 3, pages 1870 – 1875, 2001. ISBN 0-7803-7044-9. doi: 10.1109/IJCNN.2001.938448.

Andrew Michael Saxe, Yamini Bansal, Joel Dapello, Madhu Advani, Artemy Kolchinsky, Brendan Daniel Tracey, and David Daniel Cox. On the information bottleneck theory of deep learning. In *International Conference on*

Learning Representations, 2018. URL https://openreview.net/forum?id=ry_WPG-A-.

Luciano Serafini and Artur S. d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *CoRR*, abs/1606.04422, 2016. URL http://arxiv.org/abs/1606.04422.

Hava T. Siegelmann. Computation beyond the turing limit. *Science*, 268 (5210):545–548, 1995. ISSN 0036-8075. doi: 10.1126/science.268.5210.545. URL https://science.sciencemag.org/content/268/5210/545.

Hava T. Siegelmann and E.D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132 – 150, 1995. ISSN 0022-0000.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Workshop at International Conference on Learning Representations*, 2014.

Paul Smolensky. The proper treatment of connectionism. *Behavioral and Brain Sciences*, 11, 1988. doi: 10.1017/S0140525X00052432.

Paul Smolensky and Géraldine Legendre. *The Harmonic Mind: From Neural Computation to Optimality-Theoretic GrammarVolume I: Cognitive Architecture (Bradford Books)*. The MIT Press, 2006. ISBN 0262195267.

J. Su, D. V. Vargas, and K. Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019. ISSN 1941-0026. doi: 10.1109/TEVC.2019.2890858.

Ilya Sutskever, Geoffrey Hinton, and Graham Taylor. The recurrent temporal restricted boltzmann machine. In *Advances in Neural Information Processing Systems*, volume 20, pages 1601–1608, 2008.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014,*, 2014. URL http://arxiv.org/abs/1312.6199.

S. B. Thrun. Extracting provably correct rules from artificial neural networks. Technical report, University of Bonn, 1994.

Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *IEEE Information Theory Workshop*, 2015.

Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control and Computing*, pages 368–377, 1999.

Geoffrey G. Towell. *Symbolic Knowledge and Neural Networks: Insertion, Refinement and Extraction*. PhD thesis, University of Wisconsin-Madison, 1991.

Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101, 1993.

Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165, 1994.

Geoffrey G. Towell, Jude W. Shavlik, and Michiel O. Noordewier. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 2*, pages 861–866. AAAI Press, 1990. ISBN 0-262-51057-X. URL http://dl.acm.org/citation.cfm?id=1865609.1865629.

Son Tran and Artur Garcez. Deep logic networks: Inserting and extracting knowledge from deep belief networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(2):246–258, 2016.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

Jason Yosinski, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *CoRR*, abs/1506.06579, 2015. URL http://arxiv.org/abs/1506.06579.

Jan Zilke, Eneldo Mencía, and Frederik Janssen. Deepred – rule extraction from deep neural networks. In *Discovery Science 2016*, LNCS,

pages 457–473. Springer, 2016. ISBN 978-3-319-46306-3. doi: 10.1007/978-3-319-46307-0_29.