Designing Interpretable Chess Engines Using Logical Neural Networks

January 31, 2022

Abstract

The problem of interpretability in DNNs is classically a hard one, as the parameterisation of network layers is hard to intuit. This paper uses findings in Logical Neural Network architectures to construct an interpretable model for use cases which are easily modelled by logical statements, and applies this architecture to the problem of learning Chess.

Contents

Chapter 1

Interpretability

1.1 Introduction

Research into problems in Machine Learning over the past two decades has focused largely into using Neural Network (NN) models to solve an increasingly large breadth of problems. NNs, much like many other models, define a hypothesis class of functions which differ only in their parameterisation, and uses Stochastic Gradient Descent (SGD) or some derivative thereof, to optimise said parameters given a loss function. Classical Multi-Layer Perceptrons (MLPs) consist of a series of linear layers separated by some non-linearity, (e.g. ReLU, Sigmoid functions). Given certain conditions, it is known that MLPs can learn any continuous function to arbitrary precision, by the Universal Approximation Theorem (UAT). The effectiveness of SGD methods allows us to learn arbitrary functions tractably, which has resulted in a widespread adoption of the architecture in practical settings.

A common criticism of NNs, however, is that they are considered "black box" functions. NNs are difficult to interpret, making it even more difficult to diagnose issues that may arise in production. A particular node in the network may be considered as capturing a single "concept", which can further be used to determine a metric for the presence of other concepts. It is difficult, however, to intuit how these concepts are generated - taking linear combinations of features and then applying a non-linear map to the result is not a terribly human line of thinking when it comes to pattern recognition. In this way, when comparing NNs to real-world neural processes, the description of NNs capturing general human intuition, rather than any kind of conscious reasoning, is most apt.

The "black box" nature of NNs has resulted in a hesitancy for it's adoption in particular settings, most notably in the medical industry, where even small risks of misdiagnosis cannot be tolerated. One would assume that an architecture that is so widely used in so many sensitive settings should be easily examinable, but this isn't that case.

From this, the notion of "interpretability" as an important concept in ML

was introduced. An ideal interpretable model would allow the user to know precisely what the model is achieving by arriving at a particular optimal parameterisation. Interpretability is not a quantifiable metric - the user may gain an understanding through a mixture of the learnt parameters and an existing intuition over the architecture itself. This allows for a wide breadth in methods which may be used to gain an understanding of a model, and also hopefully the underlying problem.

1.2 Interpretation Methods

There are many ways we may attempt to approach the problem of interpretability. Commonly used methods take arbitrary NN architectures, and attempt to gauge how relevant a particular feature of a given input is in the overall output of the model. These are known as *Variable Importance Methods* (VIMs). Gradient-based attribution methods [1] are the classical example, where the gradient of the model with respect to it's input features are used as the measure of feature importance. This makes intuitive sense, as if the output of the model is subject to large changes with small deviations in an input feature, it must naturally be fairly important. The most commonly seen setting for these methods is in image classification, where the importance of a particular pixel measures how much of an influence said pixel has on determining the category of an image. Plotting the importance of all the pixels hopefully shows the user precisely which portions of the image contain the relevant object to classification. E.g., distinguishing between cats and dogs would largely rely on examining particular features of the face shape, so one would expect these features to be the most important by this metric.

These methods are very versatile, as they are *model-agnostic*. There are many flaws, however - what if the classification of an image relies on a combination of features, rather than just a single one? We can capture this notion by instead using VIMs over all nodes in the network, i.e. the input layers and all hidden layers, but we run into the same problem - if we determine that a node in a hidden layer is important, how do we begin to understand what this node is doing? This method captures relevance, but does not capture what concepts these features may represent - we can leave this again up to the intuition of the user, or we can apply VIMs recursively between the input features and the hidden feature. This eventually becomes somewhat unwieldy.

Another issue is that VIMs are *local* interpretation methods, as they do not describe the model as a whole - only the model given a set input. This does not give us a good understanding as to why the model's solution to a problem is best.

A solution to the problem of not capturing feature relationships is in developing *model-specific* methods of interpretability. We can design an architecture which allows for novel ways of visualising model behaviour, often by restricting the expressiveness of the model in a manner which allows the remaining hypothesis class to be easily distinguishable.

One example of such an architecture are Neural Additive Models (NAMs) [2]. Neural Additive Models are a generalisation of General Additive Models (GAMs) in that they are fully described by the equation

$$M(\mathbf{x}) = \sigma\left(\sum_i M_i(x_i)\right)$$

Where the $M_i : \mathbb{R} \mapsto \mathbb{R}$ represent univariate NNs, and σ is the *link function*.

In backpropagation, we learn the parameters of each subnetwork M_i simultaneously. Given that each subnetwork is a map $\mathbb{R} \to \mathbb{R}$, we can capture the behaviour of the model not only locally through VIMs, but globally, as we can easily plot the value of M_i over the entire domain. Simply observing this graph allows the user to speculate as to what the model has learnt.

This model, while very interpretable, is not very expressive - we cannot capture any relationships between variables that aren't described by the link function σ, as is the nature of GAMs. This is the very problem we intended to solve by discussing NAMs - we want to be able to capture not only the relevance of input features, but of learnt concepts over those features.

Again, new model architectures have been introduced to resolve this. The aptly named Explainable Neural Network (xNN) [3] is an architecture which extends NAMs with a single linear "projection layer". These are equivalent to learning a GAM over *linear combinations* of input features. They are therefore fully described by the equations

$$\mathbf{a} = \mathbf{W}\mathbf{x} + \mathbf{b}$$
$$M(\mathbf{x}) = \sigma\left(\sum_i M_i(a_i)\right)$$

Where $\mathbf{W}, \mathbf{b}$ are learnable parameters as standard. We can interpret this model very similarly, by again plotting the univariate subnetworks M_i, and simply interpreting what a concept a_i represents by directly observing the linear combination. The UAT tells us that this single added layer is enough to model any function to an arbitrary precision, but in practice this hidden layer would need to be quite wide for anything other than the most trivial problems, and the features introduced in a large hidden layer may not be terribly intuitive to understand, and may even introduce a large bias.

We could extend this further, by adding more perceptron layers to capture more complicated relations between input features, but this naturally comes at the expense of interpretability. An ideal solution to this problem would allow for the model to be extended with more layers without sacrifice.

This motivates a new architecture for layers in our NN, as perceptron layers are the main obstruction to interpreting our models.

1.3 Inspiration from Non-NN Architectures

AI research is not a field that began with the development of NNs. Early research focused mainly on the nature of logical statements - put simply, if we assume facts $P_1, \ldots, P_n$, what further facts can we derive? A simple example is given by; if we know A and $A \to B$ to be true, then B immediately follows.

Given a set of background knowledge facts $P = P_1 \wedge \cdots \wedge P_n$, sets of *positive examples* $E^+ = E_1^+ \wedge \cdots$, and *negative examples* $E^- = \neg E_1^- \wedge \cdots$, it would be useful to determine a hypothesis H such that;

$$\begin{aligned}
\text{(Necessity)}\quad & P \not\Longrightarrow E^+ \\
\text{(Sufficiency)}\quad & P \wedge H \Longrightarrow E^+ \\
\text{(Weak Consistency)}\quad & P \wedge H \not\Longrightarrow \text{False} \\
\text{(Strong Consistency)}\quad & P \wedge H \wedge E^- \not\Longrightarrow \text{False}
\end{aligned}$$

Intuitively, the necessity and sufficiency conditions ensure that we can verifiably prove E^+ with H, but not without. Weak consistency forces H to not result in a contradiction (as this would entail every logical statement), and strong consistency requires H not to prove anything in E^- that we assert not to be true.

A program that can compute answers to the above problem is known as an Inductive Logic Programming (ILP) system. ILP systems are very useful - suppose we know of a particular game concept which is beneficial to the player but for which we have no logical representation. However, we may have E^+ as positive examples, and E^- as negative examples. If we can determine an H which satisfies the above conditions, that is, H is a logical statement that is consistent with our set of examples, then we can use this learnt H to describe the concept as a whole, and extrapolate our findings to game states we have not yet seen. We can therefore incorporate H into a heuristic we may use in evaluating a game state, which can go on to be used in more sophisticated algorithms such as the breadth of variants of Min-Max tree search.

A further benefit of ILP systems is that they are by nature interpretable. The hypothesis H is a single logical statement which can easily be read and verified by humans. Therefore if we do manage to devise an ILP system, and corresponding heuristic, we could learn to play a game in a highly interpretable manner. The goal of this paper is to come up with a system similar in nature to this.

Symbolic inference, like the ILP systems described above, formed the bulk of research into AI and Machine Learning from it's early inception in the 1950s up until the 1990s. This is notably no longer the case - ever since the 2000s, Machine Learning as a field has become incredibly publicly prominent through the development of statistical ML methods, and most notably NNs. NNs proved to be much more adept at learning behaviours given smaller amounts of data, where in practice, symbolic machine learning requires an incredibly large amount of data to derive anything meaningful. This lead to famous criticisms, such as

those levelled by Hubert Dreyfus, as to whether the field of symbolic machine learning would be suitable for anything other than simple toy problems.

If we want to leverage the learning power of NNs, and the interpretability of ILP systems, we need to find a way of compromising the two approaches. That is, we want to create an ILP system which learns in the same way as NNs, through backpropagation. There is immediately an obvious issue - logical hypotheses created by ILP systems are *boolean functions*, that is they are mappings $\{0,1\}^n \to \{0,1\}$. Backpropagation relies on computing derivatives of a model described as a function over a continuous domain - which $\{0,1\}^n$ is distinctly not. If we want to begin devising methods of backpropagation over boolean functions, we have to solve this issue.

Chapter 2

Logical Neural Networks

2.1 Introduction

The field of research which focuses on deriving methods of backpropagation over boolean functions is referred to as *Neurosymbolic Machine Learning*. The models that are created to solve such a problem are known as *Logical Neural Networks*[1] (LNNs)[2].

The natural solution to solving the problem of $\{0, 1\}$ being a discrete space, is by embedding values **T**, **F** in a continuous space (or more formally, a topological manifold), and computing derivatives in this space instead. This poses an obvious problem - what if we find that the optimal parameters are values that do not equal **T** or **F**? The ways we may begin to solve this problem differ depending on the particular architecture we decide to use. This paper discusses two different classes of architecture, *Fuzzy Logic Neural Networks* and *bool2vec*.

From now on, we will begin using the symbol $\mathbb{B}$ when referring to the space of boolean representations for a given architecture. Where two modes of representation are being compared, they will be differentiated with a subscript (e.g. $\mathbb{B}_2$, $\mathbb{B}_\mathrm{f}$, etc.)

We will explicitly refer to the set $\{0, 1\}$ used in classical logic with notation $\mathbb{B}_2$.

2.2 General Observations

We will revisit the particular implementations of LNNs later. Initially, it is important to discuss some general ideas.

As mentioned previously, the UAT shows us that it is possible to model any boolean function within the framework of MLPs, but we want to restrict the architecture of MLPs such that we could begin to interpret the learnt model as a logical formula, while maintaining full expressiveness. It is well known that

[1] also *Neural Logic Networks*
[2] also NLNs

any boolean function can be represented in Disjunctive Normal Form (DNF). That is, a formula that takes the form;

$$\begin{aligned}\phi(x_1, \ldots, x_N) = &(a_{11} \wedge a_{12} \wedge \cdots \wedge a_{1n_1}) \\ &\vee (a_{21} \wedge a_{22} \wedge \cdots \wedge a_{2n_2}) \\ &\vee \cdots \\ &\vee (a_{m1} \wedge a_{m2} \wedge \cdots \wedge a_{mn_m})\end{aligned}$$

where the $a_{ij} \in \{x_1, \ldots, x_N, \neg x_1, \ldots, \neg x_N\}$.

This is very convenient, as it means that any architecture we choose to build can learn this highly regular form, instead of some arbitrarily deep tree of logical operators. To transform this into the language of NNs, we want to somehow decompose the above form into a series of "logical layers".

There is a natural homomorphism between subsets $S \subseteq \{1, \cdots, N\}$ and conjunctions $C : \{0,1\}^n \to \{0,1\}$. That is, any conjunction C is fully defined by some set S, where

$$C(x_1, \ldots, x_n) = \bigwedge_{i \in S} x_i$$

If we want to learn the conjunction C, it is therefore equivalent to learn the membership of the set S. Let $w_i = \mathbb{1}(i \in S) \in \mathbb{B}$. Then we need only learn the values of the parameters w_i. It is important to note that, given the w_i, if we force $\mathbb{B} = \mathbb{B}_2 = \{0,1\}$, we can represent conjunctions instead by

$$C(x_1, \ldots, x_n) = \bigwedge_{i=1}^{N} (1 \cdot (1 - w_i) + x_i \cdot w_i)$$

This form becomes important when we begin to extend the domain of boolean values to continuous spaces, as it can accommodate *fuzzy set membership*, the meaning of which we will detail in the next section.

The above only captures an output in one dimension, but to behave like a NN layer, we need to have inputs of arbitrarily many dimensions, each with different parameterisations of w_i. We can capture this like so.

$$\begin{aligned}C(\mathbf{x}; \mathbf{w}) &= \mathbf{y}, \text{ where} \\ y_j &= \bigwedge_{i=1}^{N} (1 \cdot (1 - w_{ij}) + x_i \cdot w_{ij})\end{aligned}$$

A similar process can be used to define a "disjunctive layer" much like the "conjunctive layer" above.

$$\begin{aligned}D(\mathbf{x}; \mathbf{W}) &= \mathbf{y}, \text{ where} \\ y_j &= \bigvee_{i=1}^{N} (0 \cdot (1 - w_{ij}) + x_i \cdot w_{ij})\end{aligned}$$

Here we use 0 instead of 1, as these are each operator's respective identity values. To have full expressiveness over boolean functions, it is therefore enough to have the following architecture;

$$M(\mathbf{x}; \mathbf{w_D}, \mathbf{W_C}) = D(C(\mathbf{x} \oplus \neg\mathbf{x}; \mathbf{W_C}); \mathbf{w_D})$$

In this particular context, $\oplus$ refers to vector concatenation - this is not the case later on in this document. $\mathbf{w_D}$ refers to a vector, rather than a matrix, as there is only a single disjunction.

Suppose we are given feature valuations $x_1, \ldots, x_N$, and we are also given whether said features satisfy some set predicate $P(x_1, \ldots, x_N)$. To create a functioning ILP system, it is enough to find some logical formula ϕ which is consistent with this predicate, meaning that for all inputs $\mathbf{x}$ we have received, $\phi(\mathbf{x}) = P(\mathbf{x})$. The purpose of LNNs is to use backpropagation to find an approximation to such a solution. By nature, NNs are "forgetful" in that if the distribution and output value of incoming data changes, they are able to modify their parameterisation accordingly. They therefore aren't guaranteed to be consistent with all input data, but this actually becomes a benefit, rather than a hindrance, when we begin to embed LNN systems within classical NN architectures.

This embedding is important to note, as the aim of the architecture presented in this paper is not to find *correct* concepts, but *useful* ones. We do not necessarily aim to find hypotheses that perfectly represent known concepts, but concepts that when incorporated into our decision making, allow the player of a game to perform effectively. In that sense, we are using LNNs in a way that is similar in construction, but different in intention to ILPs. This is an important distinction to make when comparing this architecture to existing ones in the space of Neurosymbolic ML, which we will discuss later.

2.3 Interpreting a Neurosymbolic Layer

We have constructed layers which represent learnable conjunctions and disjunctions, but how do we go about converting this into a human interpretable format? So far we have discussed an architecture which learns any boolean function in DNF, but is this an ideal way of communicating concepts to humans?

Let's consider what a formula in DNF is actually communicating. Each disjunction is naturally going to represent a single concept, and each conjunction within can be interpreted as being instances of said concept. Using the example of Chess, if we want to learn what it means for a bishop to attack a king, we want to iterate over all instances where a bishop is in a directly diagonal position to a king, with no obstruction. Determining each one of these instances can be done with a conjunction (e.g. Bishop on A1, empty spaces in B2, C3, ..., King on E5), and determining whether any of these conjunctions has occurred is handled by the disjunction.

In this manner, the disjunction is a set of instances of a particular concept, and the conjunctions are the elements of this set.

Is this the best architecture for interpretability? One may consider an architecture with two DNF layers - that is, concepts building on top of further concepts. This can be incredibly useful - suppose we have both a bishop and a knight attacking the king. The presence of both of these concepts, one can imagine, is greater than the sum of it's parts, so it may be useful to consider this a joint concept. We don't necessarily require a second DNF layer to discover this, as we have shown that a single layer is sufficient to fully express all boolean functions, but for the sake of interpretability, it may be advantageous to add this second layer. This is in stark comparison to conventional NNs, where adding layers generally sacrifices interpretability for the sake of expressiveness. This consideration will become important when comparing different implementations of neurosymbolic architectures in practice.

We can now begin to discuss different ways of actually implementing such an architecture.

2.4 Fuzzy Logic

An intuitive, and well researched approach to extending the space of valid boolean values is to consider truth to be "vague", in the sense that something can be "somewhat" true or "somewhat" false. To quantify this, we take boolean values in the closed interval $[0, 1]$, rather than simply $\{0, 1\}$. This approach is known as *fuzzy logic*.

It is important to note that while this emulates the definition of a probability measure, it is distinctly not. $\mathbb{P}(x) = \frac{1}{2}$ states that x is true with a probability of $\frac{1}{2}$, which can mean either that in $\frac{1}{2}$ of cases, x appears true (the frequentist interpretation), or that we believe that x is true with $\frac{1}{2}$ certainty (the Bayesian interpretation). In fuzzy logic, $x = \frac{1}{2}$ instead states that x is "half-true", with 100% certainty. We are only using fuzzy logic to approximate classical, discrete logic, so it is not important to dwell on the philosophical implications of this. Extending the domain of boolean values means that we have to revisit the definitions of simple logical operators $\neg, \wedge$ and $\vee$.

2.4.1 T-Norms

We will begin with a generalisation for $\wedge$, as all further definitions follow from this. We want to find a function $\otimes : \mathbb{B}_{\mathrm{f}}^2 \to \mathbb{B}_{\mathrm{f}}$ which has the same value as $\wedge$ for $\mathbb{B}_2$, and maintains some natural properties of $\wedge$ also. Suppose we assume the following axioms for $\otimes$;

$$\begin{aligned}
\text{(Associativity)}\ & (a \otimes b) \otimes c = a \otimes (b \otimes c) \\
\text{(Commutativity)}\ & a \otimes b = b \otimes a \\
\text{(Monotonicity)}\ & a \leq b, c \leq d \implies a \otimes c \leq b \otimes d \\
\text{(Identity)}\ & \forall a, a \otimes 1 = a
\end{aligned}$$

The axioms are actually already enough to ensure that $\otimes|_{\mathbb{B}_2} = \wedge$. However, they are not enough to ensure $\otimes$ takes one particular value in the set $\mathbb{B}_{\mathrm{f}}^2 \to \mathbb{B}_{\mathrm{f}}$,

in fact there are an infinite family of possible functions $\otimes$. The above axioms are known as the *t-norm axioms*, and the functions which satisfy it are known as *t-norms*. Some examples of t-norms are;

$$\begin{aligned}
\text{(Product t-norm)}\quad & a \otimes b \coloneqq ab \\
\text{(Minimum t-norm)}\quad & a \otimes b \coloneqq \min\{a, b\} \\
\text{(Łukasiewicz t-norm)}\quad & a \otimes b \coloneqq \max\{a + b - 1, 0\}
\end{aligned}$$

Taking one of these definitions for $\otimes$, we can define a *fuzzy conjunction* in the same way we defined classical conjunctions before.

$$C(x_1, \ldots, x_N) = \bigotimes_{i=1}^{N} (1 \cdot (1 - w_i) + x_i \cdot w_i)$$

Where the $w_i = (i \in S) \in B_{\mathrm{f}}$. That is, the i's can now be "partially" in the set S. This is what is meant by *fuzzy set membership*. Intuitively, we can interpret this as linearly interpolating between values x_i and identity 1, based on "how much" i is a member of S.

T-norms are an important first step, but if we hope to do any kind of learning at all, we need to be able to generalise some other operators too.

2.4.2 A Full Fuzzy Operator Scheme

If we can generalise either $\neg$ or $\vee$, then we can generalise both. A seemingly obvious way to generalise $\neg$ is by simply declaring that $\neg x = (1 - x)$. This immediately satisfies the definition of $\neg$ in classical logic, and comes with some useful properties;

$$\begin{aligned}
\text{(Self-Invertibility)}\quad & \neg(\neg a) = a \\
\text{(Monotonicity)}\quad & a \leq b \implies \neg a \geq \neg b
\end{aligned}$$

From this, we can fully define generalisations for $\vee$, which we call *t-conorms*. In classical logic, we can appeal to *De Morgan's laws*, in that $a \vee b = \neg(\neg a \wedge \neg b)$. Similarly, we can say that $a \oplus b = \neg(\neg a \otimes \neg b) = 1 - (1 - a) \otimes (1 - b)$, given a particular choice of t-norm $\otimes$. We therefore have;

$$\begin{aligned}
\text{(Product t-conorm)}\quad a \oplus b &\coloneqq 1 - (1 - a)(1 - b) \\
&= a + b - ab \\
\text{(Minimum t-conorm)}\quad a \oplus b &\coloneqq 1 - \min\{1 - a, 1 - b\} \\
&= \max\{a, b\} \\
\text{(Łukasiewicz t-conorm)}\quad a \oplus b &\coloneqq 1 - \max\{(1 - a) + (1 - b) - 1, 0\} \\
&= \min\{a + b, 1\}
\end{aligned}$$

Given the t-norm axioms, we immediately have some convenient properties of t-conorms also.

$$\begin{aligned}
\text{(Associativity)}\ & (a \oplus b) \oplus c = a \oplus (b \oplus c) \\
\text{(Commutativity)}\ & a \oplus b = b \oplus a \\
\text{(Monotonicity)}\ & a \leq b, c \leq d \implies a \oplus c \leq b \oplus d \\
\text{(Identity)}\ & \forall a, a \oplus 0 = a
\end{aligned}$$

The only difference here being that the identity has value 0, rather than 1. From this, we can define every logical formula using fuzzy logic operators, allowing us to motivate an architecture for a fuzzy NN.

In the field of fuzzy logic, the above method of constructing fuzzy operators is not actually the preferred way of doing so. Fuzzy logicians instead choose to define the *residuum* $\Rightarrow$ in terms of fuzzy conjunction $\otimes$. The residuum, as the symbol suggests, is meant to generalise the implication operator $(a \Rightarrow b) = \neg(a \wedge \neg b)$. The reason this can be done only relying on the existence of the t-norm (rather than t-norm + fuzzy negation) is because it can be proven that the residuum is the *only* function that satisfies the conditions

$$a \otimes b \leq c \iff a \leq (b \Rightarrow c)$$

If we can find this unique definition for $\Rightarrow$, we can go on to define all other logical operators;

$$\begin{aligned}
\neg a &= (a \Rightarrow 0) \\
a \oplus b &= (\neg a \Rightarrow b)
\end{aligned}$$

Again, we will not dwell on this, as we are aiming to construct fuzzy operators specifically to approximate classical logical formulae in DNF. The definitions for negation, conjunction and disjunction given above are more than enough to begin doing so.

2.4.3 Defuzzification and Interpretation

Suppose we have embedded a fuzzy LNN layer into our larger neural network. How do we go about interpreting the parameterisation of the function? We have already discussed how we may approach doing so for classical DNFs, but to achieve anything at all, we would need to be able to do so for fuzzy DNFs as well.

We can avoid this issue by ensuring that we learn parameters that are as close to $\mathbb{B}_2$ as possible. We refer to the (total / average) distance between the current parameters of the model and the ideal 0,1 as the "crispness" of the model.

To do so, we may consider applying some regularisation term to our loss function, which negatively weights parameter values that are far from 0 and 1. We may choose some even function f which increases with x, and take the joint distance metric

$$d(x) = \min\{f(x-0), f(x-1)\}$$

A reasonable choice of f could be $f(x) = x^2$, taking a sort of "best squared error". We could then add a regularisation term $\lambda \sum_w d(w)$ over all parameters w, and use cross-validation to find an optimal λ for learning.

In practice, this hinders learning quite a bit. Much of the strength of fuzzy logic architectures are a direct result of maintaining vagueness - if we are close to values 0, 1, it can take a lot of data to modify this.

A more effective approach in practice seems much more naive - that is to simply clamp the value of the parameters w into the range $[0, 1]$ at every optimisation step. This allows the model to maintain a level of vagueness where it cannot learn much information, i.e. not "jumping to conclusions". When it actually can decide the nature of a concept, it can freely do so. We may then choose to terminate our learning when the parameters are sufficiently close to values $\{0, 1\}$.

2.5 Weighted Non-Linear Logic

I'll return to this section once I've actually implemented an example
https://arxiv.org/pdf/2006.13155.pdf

2.6 Bool2vec

So far we have met Fuzzy Logic, which fundamentally reinvents the common perceptron architecture seen in most NNs. What if we want to maintain an architecture which is more conventional in it's approach, while still allowing for boolean interpretations? This seems difficult at first - although we can approximate any boolean function, determining the nature of this function is difficult in conventional architectures, as we have seen.

Rather than modifying the architecture of the network, we can instead regularise an existing network architecture to act like a boolean function, while making sure that the regularisation somehow allows us to easily interpret the network's parameters. This sounds rather involved - how might we begin to approach this?

2.6.1 Boolean Embeddings

We have discussed the idea of embedding values representing **T** and **F** into a continuous space $\mathbb{B}$, and then differentiating over the space in backpropagation. In fuzzy logic, we specifically fix the set $\mathbb{B}_\mathrm{f} \coloneqq [0, 1]$ to act as our continuous space, and exploit this by using continuous functions which are guaranteed to behave like logical operators. What if we instead allow for the full range of parameters $\mathbb{R}$, how may we begin to interpret this as a boolean function?

We can take inspiration from other architectures where feature embeddings are common. In natural language processing, word embeddings are maps which take words in a dictionary, and map them into a finite-dimensional vector space.

The position the word is mapped to in this vector space can be used to characterise the meaning of the word, and capture relations between words. We could likewise map the boolean values $\mathbf{T}$, $\mathbf{F}$ into a vector space $\mathbb{B}$, and measure the "crispness" of a learnt function by how similar certain values are to the boolean embeddings in this vector space.

We can define some sort of "distance metric" over $\mathbb{B}$, a convex, commutative function $d : \mathbb{B}^2 \to \mathbb{R}$ such that $d(\mathbf{a}, \mathbf{a}) = 0$ for all $\mathbf{a} \in \mathbb{B}$. This can be a formal distance metric in the topological sense, but it need not be. With this, we can say that the more distant a value $\mathbf{x} \in \mathbb{B}$ is from both $\mathbf{T}$ and $\mathbf{F}$, the less crisp. A particular example which is effective in practice is given later in this section.

2.6.2 Learning a Boolean Operator

Suppose we want to learn the function $\texttt{XOR}(a, b) = (a \wedge \neg b) \vee (\neg a \wedge b)$. The model we will use to approximate this will be a NN with a single hidden layer, as it is well known that this cannot be done by a simple single-layer perceptron model. The model will therefore map values between three spaces, which we will call $\mathbb{B}^2 \to X \to \mathbb{B}$, for some $\mathbb{B}, X$, which we will not yet specify.

From here, we can proceed as usual and simply backpropagate with correct input-output pairs to learn an appropriate parameterisation for `XOR`. However, we can somewhat cheat, because we know the appropriate behaviours of the function we're trying to learn, we only want to learn it over our arbitrary boolean embedding $\mathbb{B}$. We can exploit known properties of `XOR` to "push along" our learning. For instance, we know that;

$$\begin{aligned}
\text{(Associativity)}\ & \texttt{XOR}(\texttt{XOR}(\mathbf{a}, \mathbf{b}), \mathbf{c}) = \texttt{XOR}(\mathbf{a}, \texttt{XOR}(\mathbf{b}, \mathbf{c})) \\
\text{(Commutativity)}\ & \texttt{XOR}(\mathbf{a}, \mathbf{b}) = \texttt{XOR}(\mathbf{b}, \mathbf{a}) \\
\text{(Identity)}\ & \forall \mathbf{a} \in \mathbb{B}, \texttt{XOR}(\mathbf{a}, \mathbf{F}) = \mathbf{a} \\
\text{(Negation)}\ & \forall \mathbf{a} \in \mathbb{B}, \texttt{XOR}(\mathbf{a}, \mathbf{T}) = \neg \mathbf{a}
\end{aligned}$$

We will define $\neg \mathbf{a}$ in the next section. We will define a "`XOR`-iness loss",

$$\ell_{\texttt{XOR}}(\mathbf{x}, \mathbf{y}) = d(\texttt{XOR}(\mathbf{x}, \mathbf{y}), \texttt{XOR}(\mathbf{y}, \mathbf{x})) + d(\texttt{XOR}(\mathbf{x}, \mathbf{F}), \mathbf{x}) + d(\texttt{XOR}(\mathbf{x}, \mathbf{T}), \neg \mathbf{x})$$

We can see that a perfect parameterisation of `XOR` would minimise this loss. Thus, a complete loss function would look like so,

$$\ell(\mathbf{x}, \mathbf{y}, \mathbf{z}) = d(\texttt{XOR}(\mathbf{x}, \mathbf{y}), \mathbf{z}) + \lambda_{\texttt{XOR}} \ell_{\texttt{XOR}}(\mathbf{x}, \mathbf{y})$$

The parameters of `XOR` are hidden in this representation, as they need not be of any particular architecture.

2.6.3 Boolean Regularisation

2.6.4 Measuring Similarity

2.7 Overview of Different Architectures

Chapter 3

LNNs in Practice

Chapter 4

A Chess Architecture

Chapter 5

Interpreting Chess

Chapter 6

Conclusions

Bibliography

[1] Ancona M., Ceolini E., Öztireli C., Gross M. (2019) Gradient-Based Attribution Methods. *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning.* Lecture Notes in Computer Science, vol 11700. Springer, Cham. https://doi.org/10.1007/978-3-030-28954-6_9

[2] R. Agarwal, L. Melnick, N. Frosst, X. Zhang, B. Lengerich, R. Caruana, G. Hinton. Neural Additive Models: Interpretable Machine Learning with Neural Nets. arXiv:2004.13912

[3] J. Vaughan, A. Sudjianto, E. Brahimi, J. Chen, V. Nair. Explainable Neural Networks based on Additive Index Models. arXiv:1806.01933