Imperial College London

Independent Study Option

Report

Neural-Symbolic Learning

George Alexander

May 1, 2019

1 Introduction

1.1 Why think about combining Neural and Symbolic Reasoning?

Gradient descent through backpropagation in Neural networks has allowed deep models of complicated procedures consisting of multiple intermediate steps to be learned with data. Tasks that require complex models and reasoning procedures that have in the past been considered too hard for machines can now be learned effectively. This success motivates researchers to attempt to reframe additional problem areas by creating neural networks that can express those tasks in the hope that similar improvements can be made.

For the symbolic domains, one difficult problem is *logical induction* or *rule learning*, which is where we try to learn logical models with data, with a key difficulty of current approaches being that it is challenging to learn complex models that require the invention of new rules or predicates. The complexity of the models learned by neural networks suggests that the learning approaches within that area could potentially be useful in logical induction, if the gradient descent through backpropagation approach could be suitably adapted. This project will discuss neural-symbolic learning in general, but will also explore its application to this problem in detail, and offer some new modifications of existing approaches to the topic.

Logic is important within artificial intelligence as it can express a variety of functions in a succinct way as well as being able to represent more varied relational information. Knowledge bases are often more complex and inter-related than the row-column form many machine learning tasks take. Since there are often situations where there are both complex relationships suitable for modelling with logic, as well as huge amounts of data, neural networks which excel with huge amounts of data that could be adapted for such relationships would be ideal for these tasks.

Current purely symbolic logic learning is capable of both being expressive and interpretable see Cropper and Muggleton 2016, Muggleton et al. 2013. Deep learning methods fail to be interpretable as the output artefact of the learning process is a multi-dimension array of perhaps millions of weights, even the simplest models in machine learning such as decision trees, ensembles, or regression can still have a very large number of parameters and so fail to either be interpretable or generate additional understanding that can augment human knowledge beyond just their predictions. A deep learning system whose produced artefact was logic could be expressive enough to be able to explain many things with highly compact, potentially interpretable models.

The extent to which current neural networks perform the sort of difficult reasoning which equates to these complex logical models is difficult to determine. As the blog post by Joel Grus 'FizzBuzz in Tensorflow' (www.joelgrus.com/2016/05/23/fizz-buzz-in-tensorflow/) shows with some humour, more difficult reasoning seems to be beyond standard neural networks. One possibility is that the current approaches reason by discovering a huge number of relatively simple patterns and heuristics rather than by creating complex rules. Many lauded Deep Learning systems that operate in areas where complex reasoning is seen as necessary for humans to succeed, such as Silver et al. 2016, use massive search algorithms augmented by neural networks as a heuristic tool to drive their performance, these systems are bypassing the learning and understanding of complex rules mainly through such a search. This is perturbing for the approach of applying neural networks to logic, however there are multiple reasons to explore the area despite this. Some neural models have been designed to explicitly model computer systems that perform complex computational processes Graves et al. 2014, and there exist neural models that play complex games without large search like the recent OpenAi Five model using proximal policy optimization deep reinforcement learning and just repeated evaluation of an LSTM Schulman et al. 2017 (Alphago Zero Silver et al. 2016 can also achieve less good but still high level results in chess without search). Additionally, it is the case that models whose logical decisions may be limited but are still complex in other ways, such as convolutional neural networks, whose structure represents the result of a logically expressible higher order model might be of interest for learning with meta-rules. Additionally, even though it may perhaps be difficult, the domain of logic allows researchers to directly target the creation of systems that learn these difficult complex models as opposed to finding a huge number of small patterns that may not generalise or relying on a search that may not scale, which is in itself an important goal for Artificial Intelligence and deep learning.

1.2 Features of Potential Neural Symbolic Methods

The basic objective of performing a task without being explicitly programmed to do so by instead learning from experience, is common to neural and symbolic methods. However, these methods often differ greatly in the setting and execution. Neural Symbolic learning might attempt to combine the features of both to create a more general learning system. The settings compared in the table is that of inductive logic programming ***ILP*** (where logic programs are a kind of logical theory well-suited for computing) and supervised learning with neural networks.

Logic and symbolic approaches	Neural Networks	hypothetical neural symbolic system
Mostly restricted to small data	Needs Big data	Any amount of data
Relations/ knowledge	Tables, Sequences, Images	A blend of all types of data
Mostly Knowledge - Databases	Big data	A blend of sources
Small logic programs	Large networks	Logic programs with differentiable components
Symbolic search	gradient descent	Mostly gradient descent
High expressivity	model resetricted expressivity	High expressivity
Potentially high explainability	Black boxes	potentially high explainability

Table 1: A comparison of learning styles

The limited hypothesis space, inclusion of background knowledge and strong learning bias within symbolic models can allow learning on small amounts of data – the compression and generalisation as well as the encodable bias of logic allows the learning with small data, these are roughly what one might think of as the practical human tools for learning over small amounts of data. Gradient descent allows large amounts of data to be considered within learning whereas this can be difficult with the search and derivation approaches used in symbolic fomains, these two elements are not mutually exclusive and so a learning system could potentially be able to incorporate both and thus learn well with multiple styles of data. A similar argument can be made with regards to the kind of data the two methods are suited for. The sources for the data for these sources are different as different sources tend to produce data in different preferred formats and modalities, For example, noisy numerical sources for neural networks, and complex relations and scientific or expert knowledge for symbolic systems, a potential advantage of neural symbolic learning would be the synthesis of these sources.

The Logic in neural symbolic learning can be of varied types, just as many approaches to logic have been utilised within artificial intelligence. Fuzzy logic, real logic, relevant logics, as well as full first order logic, and datalog programs as can be seen in Ceri et al. 2012 and Garcez et al. 2002. The most basic logic intuitively familiar to anyone familiar with programming is boolean propositional logic, where ANDs, ORs and NOTs combine to create statements that can be evaluated for a truth value, 1/True or 0/False, if enough of the truth values of the parts are known. First Order Logic is the basis for much of computational logic, and offers greatly increased expressibility through the use predicates applied to variables and constants, and thus of universal statements that are true or false for an entire domain. Where neural networks have the universal approximation theorem Hornik 1991, classical logic has the Church-Turing conjecture that functions as a universal representation thesis for first order logic. Full first order logic is usually too difficult for use in learning systems, and so modified versions are used that are a sub-system of first order logic, and sometimes additions are made that are not within first order logic. In practice a logic program consists of a series of *if-then* rules where a program can test the truth of a statement by finding any *if-then* statement where the statement being tested corresponds to the *then* part of the rule, and then moving on to test the statements that make up its *if* part. Simple facts are represented by *then* statements without corresponding *if*s, and a query terminates if no more necessary statements must be proved that are not present as a fact i.e. success or True is returned, or there is some statement that must be checked but has no *then* corresponding to it i.e. failure or False. In this approach disjunctions appearing in the *if* part -also known as the body- are represented by having to separate *if-then* rules each with the same *then* (*then* is also known as the head), but each with a different disjunct as its body. Classical negation is a challenge in this setting and often preferred is negation as failure, where if a negation appears in a rule body a sub process is executed to attempt to check to see if that negated statement can be proved in the same way as an ordinary query – if it can then the negation as failure returns false, if that proof fails then it returns true. Rules represent universal rules through the use of variables in the bodies and heads of rules and finding rules with a possible substitution to be made. This proof process is the one used in Prolog resolution. Some of the consequences of a logic program can be discovered by collecting a set of known facts and applying rules that have the bodies satisfied by this set and adding their heads to our new set of known facts, and repeating indefinitely.

2 Literature Review

2.1 Tensor Product representations in Neural Networks and symbolic functions from neural computation

An important perspective in neural-symbolic learning that comes from a linguistic natural-language processing perspective is that of a tensor product representation as distributed symbolic processing within neural networks. An early criticism of Neural networks and machine learning was their inability to perform tasks that required symbolic representations Smolensky 2012, especially in the area of symbolic natural language processing. Theories of neural-symbolic computing describing just how neural networks can compute important classes of function requiring symbolic calculation. Smolensky et al. 2014,Smolensky 2012 explain how the tensor product representation allows for symbolic operations on simple data structures. The tensor product represetntation TPR is the outer product of a series of vectors representing roles and a series of vectors representing fillers. Roles specify locations within a symbolic structure e.g. position in an array, while features represent the contents present at each location, where the same filler can be reused at multiple locations. One hot vectors for fillers and roles would simply result in a data structure where each entry in the matrix representation represented a unique filler/role binding, Tensor Products can be combined using addition for compositionality. Recursive representations where a filler for a given role can itself be a tensor product representation of some sub-structure give tensor product representations their expressive power and are made constructed through the tensor product. Unbinding vectors derived from the embeddings of the roles can return their respective fillers and a combination of these operations forms the basis for a class of functions called "primitive-recursive" that are represented by linear transformations of these tensor product representations.

Recurrent Neural Networks, Neural Attention and Convolutional Neural Networks Lai et al. 2018 have all been able to perform well in tasks such as question answering and translation where this sort of symbolic processing was claimed to be necessary. the extent to which these kinds of networks utilise tensor product representations is not yet known, although McCoy et al. 2018 explores this question in some detail, discovering that certain tensor product representations can capture very well the implied structure within the learned representations of some recurrent neural network architectures. The dimension of a tensor product representation is simply the product of the dimensions of its role and filler embeddings, and so these require exponentially larger amounts of space with the depth of the recursion. Smolensky's works includes thoughts on compressions both by replacing tensor products by circular convolution creating Holographic Reduced Representations, or by using simply elementwise products of the representations (i.e. just the diagonal) although these do not have the same theoretical guarantees as the uncompressed representations. Neural networks could be expected to reasonably approximate these representations in an efficient way constrained to neither the requirements of fully uncompressed tensor product representations or the initial attempts at compression as Word2Vec Mikolov et al. 2013 and similar contemporary approaches trained on natural language do seem to learn complex compositional representations, the example of $king - man = queen - woman$, and use the features of Tensor Product representations. The representations learned in these tasks, although showing some compositionality need not be overly recursive and so we would still be unclear as to the ability of neural networks to efficiently compress more difficult tensor product representation structured, the paper Fernandez et al. 2018 quite nicely explores this by teaching neural networks to operate complex queries over tensor product representations, allowing the neural network to learn its own internal representations for the required operations to succeed. Thus demonstrating the ability of neural networks to learn approximate compressed embeddings that precisely replicate Tensor Product representation's structure. If one learns symbolic structure by learning a weighting over each possible combination of sub-structures, then what is learned is essentially a TPR that is fully recursive with only one role. If one learns symbolic structure by learning by giving a weight to each sub-structure individually, then what is learned is essentially a TPR with one-hot embeddings of fillers within an equal number of roles.

This work is an extremely strong argument for the viability of purely neural computing and the ability of these networks to operate on symbolic compositional tasks, although since it can primarily be seen as providing theoretical explanations of the capabilities of neural networks in general the usefulness of tensor product representations requires specialist architecture to bring it out clearly and make it a foundation for neural-symbolic systems.

Also, the class of functions computed, whilst interesting does not represent the full power of logic that we would like to see, for an extension of this framework towards logical calculations with tensor representations,Grefenstette 2013

A key application of the theory of Tensor Product Representations is that analysis of learned Tensor Product representation and approximate Tensor product representations in networks that are designed to create them allows an additional avenue to interpretability for those networks as in Palangi et al. 2018 with interpretability with respect to grammatical features. One can attempt to observe and recognise patterns with the assignments to fillers and try to understand what semantic features each role represents, this significantly greys the black box as it allows further understanding of the internal understanding that a neural network has created of the problem area, however this does not render those neural networks behaviour fully explainable, with the functions performed on those representations as well as the exact meanings of individual fillers and roles remaining mysterious.

This sort of interpretability and much clearer semantic features could potentially be very powerful when designing extensions to these neural systems. For instance, it allows for the careful adding in of additional analogical reasoning – where similar roles and similar fillers having meaningful similarity metrics independently. It is hard to speculate about the potential uses of separating components within, or querying these approximate compositional symbolic representations, certainly the ability for the designers of models to create scenarios where components of their networks directly interact with such a representation in multiple could encourage increased interpretability and compositional structure throughout a model allowing more deliberate semantic manipulation than would otherwise be possible, for example in computer vision semantic image interpretation as Donadello et al. 2017 imposes a level of semantic meaning similar to what a Tensor Product Representation might provide.

Future directions for this approach relevant to the sorts of neural-symbolic computing explored elsewhere in this project would be building upon models where more concrete Tensor Product Representations are used, and these representations are considered to be interesting products of the learning process themselves, and extractable and useable for future neural-symbolic reasoning. A revealed role system describing the features of a problem and their relations and internal structures, a set of atomic fillers extracted from a problem would not only be suitable for analysis in the form of rough interpretability, but could be treated as contributions to a knowledge base. This would allow an integration between the forms of reasoning within neural networks and those within other symbolic or neural-symbolic systems where selected outputs from one could be used seamlessly as inputs for the next.

2.2 Learning Complex Logic Tasks Within Neural Networks

Another approach to using neural networks to solve symbolic problems comes with Dong et al. 2019 *Neural Logic Machines* or NLM are a network architecture designed to be capable of computing a probabilistic first order logic with horn clauses. For this network, each step of forward chaining inference is completed by a set of parallel neural computation blocks, each block performing computation upon the predicates of a specific arity. A single step of reasoning, with predicates of arity null to two would require three blocks.

The bulk of the logical computation is performed by a multi-layer perceptron within each block. It uses a sigmoid activation function to calculate probabilistic Boolean functions of the form:

$$expression(x_1, x_2, ..., x_n) \rightarrow p(x_1, x_2, ..., x_n) \tag{1}$$

where expression, can be any Boolean expression over the just the variables x_1 to x_n. The remaining parts of the neural architecture are then concerned with creating operations that represent logical quantifiers. This is done by modifying the dimension of the input tensor to represent the introduction of a new variable/quantifier, or the elimination of an existing variable/quantifier.

Each block for calculations with a specific arity receives its input from the existing predicates outputted from the last step of its arity, along with predicates from the block in the previous step of one greater and one lower arity. These predicates with a different arity to the block's own are converted to the correct arity by the expand and reduce operations.

The expand operation is used to represent generalisations of the form:

$$p(x_1, x_2, ..., x_n) \rightarrow \forall(x)q(x_1, x_2, ..., x_n, x) \tag{2}$$

theoretically a rule of the form $p(x), q(y)\forall r(x, y)$ can be described in this system as :

$$p(x) \rightarrow \forall(z)pX(x, z) \tag{3}$$

$$q(y) \rightarrow \forall(z)qX(y, z) \tag{4}$$

$$pX(x, y), qX(y, x) \rightarrow r(x, y) \tag{5}$$

where the first two rules are expand rules, and the third requires no expansion and is computed as an expression. The reduce operation represents the reverse:

$$\forall(x)p(x_1, x_2, .., x_n, x) \rightarrow q(x_1, x_2, ..., x_n). \tag{6}$$

Expand adds a new dimension to the input tensor, repeating the components in the original tensor along the added dimension. Reduce implements existential quantification by producing a new tensor that takes the max value along the dimension corresponding to the variable removed, whilst universal quantification takes the minimum. The inputs to the MLP in a specific block are permuted so that every ordering of variables is given to the MLP. This introduces factorial scaling with the number of variables and could very likely prove a problem for expressions of higher arity. Each layer of blocks represents one step of computation, and so the depth of the network is the number of possible steps of reasoning. The networks also have residual connections so that the inputs to each block are subsequently concatenated to its outputs.

This model is thus a deep feedforward network, with sigmoid activation, residual connections, and with unique additional block separation within a layer and interconnectivity between nearby blocks between layers, such that tensors of different dimensions are evaluated simultaneously at each layer. A drawback of this method is that even if this network performs logical computation, there is no way to extract the logical rules used from the network. While making this network unusable for any instance where exact rules are desired, this approach could still be highly interesting if it provided significant improvements over similar networks from which you also cannot realistically extract rules either. The network does appear to be able to compute the sort of logical operations it was designed for, due to the lack of extraction it is hard to confirm that this is due to it unique architecture, or a similarly trained network with some parameter sharing, some sigmoid layers, residual connections etc.. would not also perform similarly.

The NLM was trained with a curriculum learning approach Bengio et al. 2009, and was tested and compared to two highly different models MemNN (the memory networks from Sukhbaatar et al. 2015) and δILP (Differentiable Inductive logic programming from Evans and Grefenstette 2017 will be discussed in detail later – it not a state of the art Inductive Logic programming model but able to perform well on some tasks through neural-symbolic learning). All three models were tested on family tree and graph reasoning datasets. NLM appears to outperform the MemNN as the NLM achieves 100% accuracy, the MemNN does not, while δILP also achieves 100% accuracy but cannot scale to large examples. Several toy further toy examples are given, with a comparison to solely MemNN, where the NLM once again performs with 100% accuracy in every task, while the MemNN fails with 0% on one, 12% on another (although it does achieve 90% performance on a integer sorting task). These results appear highly promising for NLM, however do only apply to toy datasets. The current state of the art in neural-symbolic inductive logic programming - where exact rules can be extracted - is also limited to toy datasets due to the early state in the development of that approach. However, an approach that does not allow extraction of learned rules and gives a comparison to similar approaches that also do not, such as the MemNN, is one where I would have hoped for application to more difficult and more realistic problems. It is difficult to evaluate the performance of an approach when the only examples given are those for which the network achieves 100% accuracy and no details on training times are given, thus the scalability and general trainability of this approach is highly unclear – especially when the authors admit that it is very difficult to train even upon these toy examples.

If this is an approach that can neither scale nor extract exact rules, then, while showing promising performance and ability to reason on toy examples, it would combine the problematic aspects of both the more statistical approaches to neural symbolic computing where rules cannot be extracted, as well as more symbolic approaches such as the papers Evans and Grefenstette 2017, and Campero et al. 2018, which, to a greater or lesser degree, have significant issues with scaling.

2.3 Learned Embeddings for logic

Having explored neural models that attempt to replicate some symbolic processing, two further trends are discussed. One area involves a focus on knowledge base completion, databases ,and limited rules for operations over those settings. The other general area focuses on using techniques from neural networks to solve the inductive logic programming problem, where the direction of research is towards learning difficult classes of function from a purely logical point of view. It is the second general area that draws direct comparison to core machine learning algorithms and deep learning through the fact that it can be seen as a general form of function approximation, and although at this stage both trends will be discussed as there are of course many similarities and links between them, it is the second that will be explored in greater depth in order to answer the questions with regards to logical learning that motivate this project as a whole.

2.4 Knowledge Bases and Neural Symbolic Learning

Given a knowledge base with a set of predicates, and constants, by treating giving those constants embeddings within a vector space, similarity and semantic hierarchy can be captured, and learned through training, as discussed when looking at the tensor at tensor product representations, there can also be compositionality to these representations, and relations among certain types of predicates, facilitating question answering over knowledge bases.

More complex logical relationships can be found within knowledge bases, and models have been developed to combine this approach of learning embeddings with logical reasoning over knowledge bases, I will discuss the Serafini and d'Avila Garcez 2016 and the Rocktäschel and Riedel 2017 approach. These logical relationships can be used as regularisation for the embeddings of predicates, or these rules can themselves be learned more directly.

2.5 Logic Tensor Networks

There are many approaches to the problem of knowledge completion and many of them involve approaches both similar to those found in logic, as well as those found in neural networks, the area of statistical relational learning explores this topic in depth Getoor and Taskar 2007. For a neural-symbolic approach, Logic tensor networks Serafini and d'Avila Garcez 2016 combine Deep learning and logical reasoning from data and knowledge Is a neural-symbolic solution to the problem of knowledge completion.

The task is filling in facts that are not given within a logical database but should nevertheless be true. This paper contributes a differentiable fuzzy form of first order logic suited to deductive inference called Real logic. In Real logic constants are represented as vectors, functions are functions of tuples of vectors to vectors, and predicates map tuples of vectors to reals between zero and one.

This approach treats learning as satisfiability – a theory is satisfiable if there is a grounding which extends the theory in such a way that for any pair of confidence interval and closed clause required, this extended grounding will with the confidence required entail the clause.

The goal of this system is to find an extended grounding to a knowledge base that satisfies as many as possible of the required interval-clause pairs (as measured by a satisfiability error) for best approximate satisfiability. What this means in a knowledge base is discovering the likely truth values for unknown facts where the truth or falsity of those facts is unknown but general rules regarding the facts are given, i.e. Allowing gradient descent and fuzzy deduction to combine to allow abductive inference.

A system in which atoms are given a value of value between zero and one is necessary for loss evaluation that allows gradient descent, this fuzzy approach also allows full first order logic, which other approaches do not. Rules with a single positive atom in the head form the basis of logic programming, logic programming allows efficient resolution proof searches. Induction with logic programs is often a combination of a search over horn clauses, followed by a resolution phase that calculates the consequences of the chosen combination or purely a resolution over higher order predicates. Neural-symbolic induction is free to move away from these approaches, although some approaches use neural networks to approximate resolution Rocktäschel and Riedel 2017. Approaches that do move away from logic programming resolution can thus potentially have a more complete representation of first order logic given that alternative calculation of logical consequences permits it.

Donadello et al. 2017 extends this to a practical application of semantic image interpretation where taking object recognition bounding boxes a semantic map is created linking the detections with relations such as “part of”. In this task the bounding boxes are produced by a fast-RCNN (Girshick 2015) The logical tensor network allows semantic constraints in the form mereological constraints translated to first order logic. This demonstrates well the potential integration of neural and symbolic computation to general problems within machine learning, as in this case the system that uses a symbolic component performs better on the given task than one that only uses neural components.

2.6 Neural Theorem Proving

The Logic tensor networks seen so far fully ground first order logic rules, an approach to logic knowledge completion that allows the use of some function terms, a similar approach for neural-symbolic knowledge completion through the use of neural representations of can be found in Rocktäschel and Riedel 2017 that will now be discussed where rules are not ground but a resolution procedure to generate proof networks that produce a differentiable proof network is used.

The construction of a proof network consists of repeated applications of a series of templates (OR, AND and UNIFY) to grow a neural network in a tree structure that represents backward chaining. Backward chaining translates queries into sub queries via rules, attempting for all rules in the Knowledge Base via a depth first search, as in prolog.

The relaxation of backward chaining here constructs these exact same trees, but instead of each rule application having a binary success or failure leading to either the growth of the tree at that leaf, or the end of exploration within that leaf, the application of a rule affects a continuous proof score by how well the embeddings for its substitutions match the required embeddings within the rule and current variable substitutions, when the proof is finished no more rule body atoms left to prove each path can generate a proof score, which is a function of the radial basis kernel applied to all the matchings it required. Even with a maximum proof depth this is still an evaluation over a huge number of possibilities, a proposed solution to this problem was the use of an approximate nearest neighbour search to explore only the most plausible substitutions. This method is differentiable with respect to the embeddings of predicates within rules, thus sit can be used for inductive logic programming as well, given desired proof-success scores, although the size of the backward chaining trees will be extremely large for a comprehensive search of the space of possible proofs.

2.7 Neural Inductive Logic Programming

Having now looked over multiple types of neural-symbolic learning, and having evaluated and assessed some approaches in greater detail, it is possible to orient ourselves with regards to the topics within neural-symbolic learning that most closely align to the purpose of this project. methods that combine the modalities of symbolic computing and neural networks are important and interesting, as they allow better use of data, and extend the potential real-world scenarios for neural networks to be applied, these methods usually involve to some degree the replication of some of the parts of logic within systems that can use the learning strengths of neural networks. The other approaches that do not focus on the challenges of introducing symbolic data to neural networks, tend to be those that focus on the abilities of logic to be a powerful representation tool for general functions and thus those approaches deal even more closely with systems that use learning strengths of neural networks on logic. The ability to learn within that context, the context of learning logic using neural networks (whether that complex logic is merely implicit within nerual nets as with TPRs or NLMs or whether it is explicit in a more specialised model), will prove to be an ability that determines the effectiveness of most other neural-symbolic approaches. The remainder of this project will thus focus on the learning of logic programs, *inductive logic programming*, using the gradient descent techniques used in training neural networks. There are two main papers that focus primarily on the learnability of rules in this way, and both will be discussed.

2.7.1 Learning Clause Pairs

The first paper to focus on inductive logic programming and learning difficult logical rules through backpropagation comes from Evans and Grefenstette 2017 and its method has been named *Differentiable Inductive Logic Programming.* Any inductive logic system must have a way of using logical rules to discover the consequences of its choices given the current background knowledge, in a neural symbolic system this process should be fully differentiable with regards to the representation to be learned. There are two main approaches to this task, feed-forward and backward-chaining. Backward chaining as used in Rocktäschel and Riedel 2017 constructs proofs calculates consequences by attempting to construct proofs for each candidate fact by unifying the heads of logical rules with those facts, and then recursively proving body atoms. Feed-forward approaches stores a large set of proven ground atoms, starting from just the background knowledge, and then adds new facts to this set when rules can be found where the head atom unifies with an unproven fact, and the body atoms are already in the set of proven facts.

The feed-forward approach can require retaining a large number of ground facts, and when facts are not merely true or false, but are given a continuous value between zero and one to represent degree of proof, potentially every possible ground fact must be retained as they all may have some small score. Additionally given that rules embeddings are also continuous, all predicates will be at least partially matched and the process of checking for whether new facts will be able to generate some new matching, even if of low value, will require the application of every rule to every possible ground fact, and the subsequent body atoms in the rule will need to be checked against every pair of possible ground facts where there is a variable matching. For a large number of rules or for problems with a large number of potential ground facts, this will not scale.

The backward chaining approach will attempt to prove the same intermediate fact in a proof each time the backward chaining search is performed whereas forward chaining only tries each once. Although backward chaining does not necessarily require a new execution of the search each time the learned embeddings are adjusted, the size of the differentiable proof trees means that it is not practically possible to retain every possible proof for every possible fact. In situations where rules are learned from scratch and there is little or no pre-existing information within the predicate or constant embeddings it would be difficult to select the correct proof paths to evaluate and so there would be no work-around to the explosion of trees. This leaves forward-chaining, even with its requirement of listing ground facts as the more plausible option for the general case.

The method in this paper uses is forward chaining, where the weights of every possible pair of clauses that could define a rule consistent with a given program template are learned. This method could be thought of as learning a one-hot embedding of a predicate where each place in the vector represents the joint appearance of two clauses. This paper does not use the specific rule templates that reduce the hypothesis space significantly that other inductive logic programming approaches tend to use, and additionally given the large number of possible clauses, there are generally a huge number of possible predicate definitions represented as pairs of clauses.

The first phase of this algorithm is the generation of these clause pairs, each clause pair is transformed into table with entries for each ground fact with the indices of other ground facts that with this clause pair would result in the ground fact of the entry being proven - that can be used to perform an operation that takes a set of all current predicate valuations, and returns an updated set of all predicate valuations after they have been applied to said clause pair.

Given weights corresponding to each clause pair and an initial valuation for each ground fact, forward chaining is the process of applying every clause, to create a new validation after a first step of reasoning, then repeating for the number of steps desired. The final valuations will be differentiable with respect to the weights and can thus be learned by stochastic gradient descent.

This method is assessed on a range of toy learning tasks, and is compared to the purely symbolic Metagol systems. On these tasks Differentiable Inductive Logic Programming successfully learns the correct rule in every case. Metagol is reported as not learning the correct rule in a few cases, but it is unclear the settings used for Metagol, and it seems this may have been a result of an implementation mistake, and I would not draw too many conclusions about their relative capabilities from this. This system also works with noisy data, in the sense of some percentage of incorrectly classified examples within the training data.

The requirement to generate and evaluate every single pair of clauses that can define each predicate, and then compute its interaction with all ground facts is a severe problem, as with a large numbers of ground facts or rules this creates a huge amount of required computation. There does not seem to be an available implementation of this approach and it is possible that this process can be done efficiently on sufficiently strong GPUs. I attempted an partial implementation of this approach within torch that can be found on my GitHub (www.github.com/ggeorgea/Differentiable-ILP-Implementation-WIP), as expected this approach does not seem to scale, and even with the simplest problems (I implemented the learning of the $is\_even()$ predicate), the pre-processing required was prohibitive, and after that pre-processing the forward chaining was fairly slow. This led me to prefer the next approach to neural-symbolic learning as it is an order of magnitude faster with a similar PyTorch implementation.

2.7.2 Learning Logical Atoms

"Logical Rule induction and theory learning using neural theorem proving" Campero et al. 2018 tackles differentiable inductive logic programming, as well as theory learning. The main insight comes in the representation of predicates in a logic program as an R^d embedding. This allows computation of a gradient of the loss function of the learning task, with regards to this predicate representation, allowing the optimisation of predicate choice through gradient descent.

Inductive logic programming uses the learning of logical rules (usually horn clauses), to generate facts through forward chaining inference. In this paper facts are represented as a tuple (θ_p, s, o, v), where s and o are terms, θ_p is a R^d embedding of the predicate p and v is the value of the fact, 1 being true and 0 false, with intermediate values being acceptable.

The earlier paper Evans and Grefenstette 2017 tackles a similar problem, but does not relax the predicates in this way, instead it generates every possible atomic fact that could be used within the body of a clause, and uses a SoftMax weight over the resultant matrix to decide which are the preferred two atomic facts for the body of some given clause. In that paper the ground atoms that satisfy each rule must be pre-computed. This paper however only needs to learn the three embeddings for the rule's head predicate, and its two body predicates, here rules are represented as $((\theta_h, v_1, v_2), (\theta_{b1}, v_3, v_4), (\theta_{b2}, v_5, v_6))$, with v_1 *to* v_6 being the terms and θ_h *to* θ_b being the head and body embeddings respectively. The steps of forward chaining consist in applying each pair of ground facts currently held to not be false to each rule. For each pair applied to each rule, multiple output facts are generated, one potential new fact for each predicate in the language. Fact pairs and rules are first compared to ensure that the required variable substitution is possible, then an output fact is produced for each possible predicate, this fact $(\theta_p, s, o, v_{out})$ is created from the possible predicate, the required substitution for s and o, and a new value that calculates the extent to which the predicates in the head and body of the rule are similar in embedding to the predicates of the ground facts, and the confidence in the ground facts:

$$v_{out} \;=\; cos(\theta_h, \theta_p) * cos(\theta_{\mathbf{b}1}, \theta_{\mathbf{f}1}) * cos(\theta_{\mathbf{b}2}, \theta_{\mathbf{f}2}) * v_{f1} * tv_{f2} \tag{7}$$

An OR is then performed with the produced fact $(\theta_p, s, o, v_{out})$ and any previously existing fact with the same predicate and terms, keeping the higherv_{out}.

This calculation is repeated for each predicate/rule/ground facts combination where a substitution works, and then the whole process is repeated t times, where t is the desired number of forward chaining steps. Initially the only ground facts that are held are the input or background knowledge. If one then applies the rules exhaustively, this collection grows to encompass every ground fact that could be generated. For problems with a large number of constants this creates a massive number of possible facts and is a serious issue with the scalability of this approach.

Once forward chaining is complete the facts that were created, and the target facts are compared with them. The exact loss function for this model is problem dependant (e.g. cross-entropy for ILP), but once it has been calculated, the gradient of each rules' parameter embedding can be computed, and the representations optimised.

In this paper the authors comment that given a one-hot embedding of θ their method is very similar to that of Evans and Grefenstette 2017. To make this comparison more explicit I would note that to describe a rule with one-hot embeddings across the domain of possible predicates, three arrays of length n where n is the number of predicates, are needed for description, this contrasts with one $n{*}n$ tensor for Evans and Grefenstette 20178. It was explained in Evans and Grefenstette 2017 that their choice to go with the larger tensor was due to the fact that, experimentally, they had trouble training two separate arrays. In Evans and Grefenstette 2017 each trained rule must be explicitly assigned a predicate beforehand, and thus for a single rule with m possible predicates, $m*n*n$ weights are learned (actually m individual rules are learned but if there is uncertainty about the correct head predicate the only option is to use them all). This does cause scalability issues, but it would be interesting, although not obvious how, to see the approach in this paper attempt a similar, more high dimensional embedding. This would mean embedding all three predicates within a single tensor, comparisons and embeddings of this sort of structure with existing predicates could lend themselves to tensor product representation (Schlag and Schmidhuber 2018 or Huang et al. 2018), which might be able to leverage both the computation of similarity and analogical reasoning, as well as being able to learn on the joint effects of the individual atoms.

A drawback of both this approach and the Evans and Grefenstette 2017 approach is that storing all, or a large proportion of all possible facts is soon required. Although this approach does not need to generate facts where there is no possible substitution, I am unclear to what extent this realistically reduces the size of the library of facts held, and thus the facts which must be substituted, and have gradients tracked at each step.

This paper applies their approach to inductive logic programming tasks as well as theory acquisition tasks. When evaluating rule learning ILP they compare their approach to that of Evans and Grefenstette 2017 on a range of examples, on the majority of tasks both approaches achieve 100% accuracy, on those where DILP does not, this approach mostly still achieves 100% accuracy. However, this approach does fail completely at the graph colouring task and the *two_childen* task, the authors claim that this is likely due to the fact that the global optima have sharp neighbourhoods and their approach would require lucky random initialisation. The fact that DILP was able to solve these problems suggests once again the expressive power of a representation that learns at the level of whole rules instead of compositionally from atoms. The algorithm is also able to learn fairly well in learning theory tasks, which shows an impressive level of generalisation to this forward chaining approach.

3 An Experimentally Driven Exploration of Neural Symbolic Inductive Logic Methods

3.1 Implementation Information

The implementation of the adjustments proposed has been based on the original implementation of Campero et al. 2018 that can be found online (www.github.com/ACampero/diff-Theory-ILP.git). The proof-of-concept and experimentation code adapted from that repository has been added to this report in an appendix. The implementation of the newly created problem, the calculation of consequences for the greater than and choice rules, as well as the implementation of each of the algorithmic modifications discussed below is new. As discussed earlier, there is also a partial implementation of the Evans and Grefenstette 2017 that can be found on my GitHub (www.github.com/ggeorgea/Differentiable-ILP-Implementation-WIP)

3.2 Notes on Practical Constraints

It must be noted that due to the practical speed and scaling constraints upon these algorithms there are likely to be significant Hardware-dependant elements to performance that were not explored, all experiments were however run on identical hardware. The various frameworks used for this sort of backpropagating gradient descent approach i.e. Torch, Tensorflow etc... may also have significant effects on the speed of these algorithms. PyTorch was used for these experiments, many of these methods create computations that are not the ideal case for efficient automatic differentiation and do not therefore have too significant of a speedup when performed on a GPU using Nvidia CUDA. This difficulty arises in computing the feed-forward consequences of a logic program, with the large and deep for-loops not being a structure that these frameworks are overly suited to. Some effort was made to have these procedures be written in such a way that would make efficient computation possible (such as list comprehensions for Evans and Grefenstette 2017) but these efforts did not appear to yield any significant speedups. I expect that even if implementing on different hardware or within a different framework i.e. using Tensorflow automatic differentiation and computational graphs instead of Torch's could yield speedups. However, the relative speeds of the different neural-symbolic ILP approaches would remain constant as it is the same style of nested for loop that causes the most significant computational bottleneck in all these approaches, and it is just the depth of this for-loop that differs. A positive side of this is that if there are ways of speeding up this bottleneck whether by more efficient implementation or modified algorithms, the benefits should generalise to multiple models.

3.3 Purpose of the Exploration

In this section I now wish to further explore the questions raised in the previous section through experimental modification and adaptation of existing methods. Specifically, expanding upon and exploring the situations in which feed-forward neural-symbolic ILP finds it difficult to learn, and in so doing, explore the limits and future pathways for such approaches. The Campero et al. 2018 approach appears to have a disadvantage in that it is unable to use gradient descent to navigate through the solution space effectively when there are local optima and a sharp hard-to-find global optima. These neural-symbolic ILP approaches are also limited to simple datalog programs, which while expressive and capable of learning some very complex reasoning is still a very limited area. Therefore, in this section problems are explored that are specifically designed to test potential extensions to the types of programs currently learned. The Evans and Grefenstette 2017 approach would appear to have scaling issues, which is explored through adjusting the Campero et al. 2018 approach to act in a similar way.

3.4 The Specific Questions To Be Answered

This exploration will be divided into two main parts, one on exploring the limits and potential modifications of the existing datalog setting for Neural-Symbolic ILP and a second on exploring a modification of the setting itself through new types of rule templates, and approaches based on that modification.

The toy problems that are constructed in this section are still only representative of very few problems as far as machine learning or AI as a whole is concerned, or as far as applicability to the real-world is concerned. However, there are real world machine learning models that are fairly close to some of these toy problems (the learning of simple decision trees for example), and some of these toy problems are based on real-world but merely extremely simple examples (e.g. family relations). I would claim that simple real world machine learning problems are possible with these methods, and applying these methods to such problems as those found in the UCI ML repository Dua and Graff 2017 would be an immediate next step given further opportunity to develop these methods.

The Campero et al. 2018 approach of atomic level feed forward learning struggles with some learning tasks due to an appparently unpleasant solution space. The particular problem that illustrates this is the two child problem, where given a set of relations describing a family tree, the task is two learn a predicate that is true if and only if its subject is someone with two children, the model is given knowledge of the child predicate and also knowledge of a not-equal predicate, the correct solution is:

$$has\_two\_children(X) \leftarrow has\_Child(X,Y), auxiliary\_predicate(X,Y)$$
$$auxiliary\_predicate(X,Y) \leftarrow has\_Child(X,Z) and not\_equal\_to(Z,Y)$$

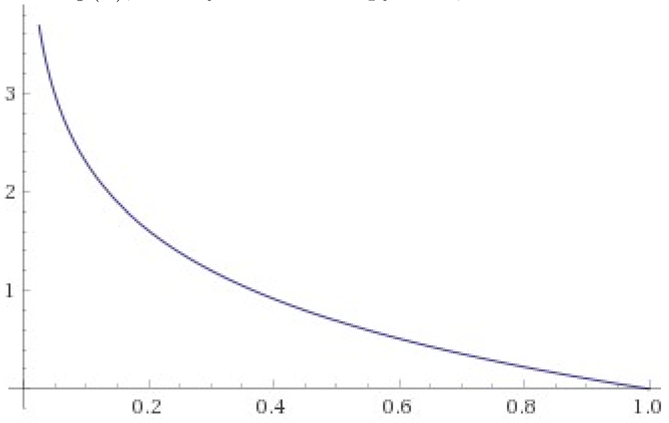
The requirement for two rules each with two body predicates,as well as the inclusion of two background predicates, makes this not too simple of a rule to learn. Using just the has_child predicate, gives a good partial solution that will reduce some error, and that could be one reason for difficulty. The training examples used also matter, Campero et al. 2018 possibly performs a gradient update after evaluating for each family example, however not all family examples necessarily have examples that show the required edge scenarios and so this could result in unstable gradient descent. Additionally, the required insight that goes further than using the has_child predicate once or twice, is that if it is used twice, we must further exclude cases where the variables are bound to the same individual.

3.5 Modifications and Adjustments

3.5.1 Initialisation

State of the art neural networks use more thought out initialisation schemes than simply applying standard distributions with standard variances, instead the weights are drawn from such a distribution then scaled by factors determined by the architecture of the networks, with more inputs to a layer generally reducing the size of the initial weights Glorot and Bengio 2010. I would suggest that bad initialisation is a danger, perhaps a greater danger, in Neural-Symbolic ILP networks as well. Neural Networks consist of many structurally identical parts and a single logical statement could be implemented in a vast number of different places within such a network, thus if initialised in such a way that the representation of some necessary logical statement is unlikely to appear in a certain location, it may well simply appear somewhere else, the fact that trained neural networks within the same task, and the same dataset can have extremely different weights when comparing identical locations illustrates this. Neural-Symbolic ILP systems do not work in this way, and usually a necessary logical

Figure 1: $-log(x)$, Binary Cross Entropy Loss, when the true value is 1



statement can and must appear in one and only one place within a system. Thus, initialisation which is likely to rule out the use of certain parts of a network for certain statements is highly undesirable. Both Evans and Grefenstette 2017 and Campero et al. 2018 reports that on harder tasks and with harder settings (e.g. with noise), even when their methods do learn, they often learn only for a certain percentage of executions, due to the possibility of unfavourable initialisation. In hard problems where a number of complex components must be learned simultaneously the danger of this bad initialisation increases, and if we assume that initialisation creates at least a small independent chance of each component being unlearnable, a large number of components that all must be learnable could make this problem very difficult. Initialising with very low values could help as although this may make the initial convergence slower, there will be fewer situations in which there is such a large initial distance to a particular solution relative to others such that it is unlikely to ever be learned.

3.5.2 Asymmetric Loss

Cross entropy loss is a standard way of evaluating classification problems in machine learning, it penalises errors to a greater extent when the predictions are further from the true value, and in the binary setting a prediction close to 0, where 1 is the true value tend to infinity, and vice versa for predictions close to 0 where 1 is the true value. From a probabilistic point of view, and for training logistic regression or neural networks this loss function makes theoretical sense. However, it will be suggested that in this setting unmodified cross entropy loss is problematic.

The Neural-Symbolic ILP model used in this setting, when initialised in a random way will most likely not have many consequences. This is because it is the case that for the head of a rule to significantly increase our belief in the probability of the statement represented within that head, the embedding for the head must match the embedding of the desired fact, also the embeddings for both bodies must match embeddings predicates where there is a body, and if there are body facts that are matched, those facts must themselves be true. If any one of these things is not true then no new fact will be generated, with any reasonable number of rules these conditions are unlikely to hold and new facts will not be generated at initialisation (and this is especially the case when embeddings are deliberately initialised to be not close to any predicates at all). What this means is that the predicted vales for all target facts will be approximately zero, thus the error initially will be extremely high.

As a result of this extremely high error driven by the extremely low valuation of facts that should be true, for the early part of the gradient descent search, the algorithm will attempt to raise value of all facts as quickly as possible, significantly prioritising this over avoiding the increased value of facts that should not be true. If, as is likely where complex reasoning is required, there is an initial locally optimum solution that would raise all target facts to have a higher value in a small number of reasoning steps, this will be reached over having a more precise solution. Gradient descent over this highly sharp loss space (sharpness entailed by the heavily discrete and non-continuous nature of the approximated logical solution space) will thus initially consistently over-estimate target truth values in order to avoid these highly disliked initializations. The gradient descent search that follows may be constrained by this initial local optimum, and thus appears unable to learn more precise but complex functions that may require multiple reasoning steps.

An adjustment to temper this effect could be to replace the loss function with another that does not have such extreme behaviour with regards to incorrect predictions.

Although I did not attempt to fully remove cross entropy loss from these methods, with Neural-Symbolic ILP this would make some theoretical sense as cross-entropy loss is inappropriate as an error metric once we have converted our logical solution back into the form of a logic program. Any errors in this fully logical program produce an infinite loss according to cross entropy. Instead I suggest that the asymmetry of the initial classifications of truth and falsity easily motivates the use of an asymmetric binary cross entropy loss:

$$y * \gamma * -log(\hat{y}) + (1 - y) * -log(1 - \hat{y}) \tag{8}$$

Such a loss will have the same behaviour in the way in which it assigns extremely high loss to extremely wrong classifications, but the speed at which it does this, and thus the degree to which the gradient descent will rush to unsuitable local optima, can be reduced. Over longer periods for standard classification tasks an asymmetric loss function will lead towards a bias in favour of either under or over-estimation, there are two things that are of interest when we consider this within the logical setting.

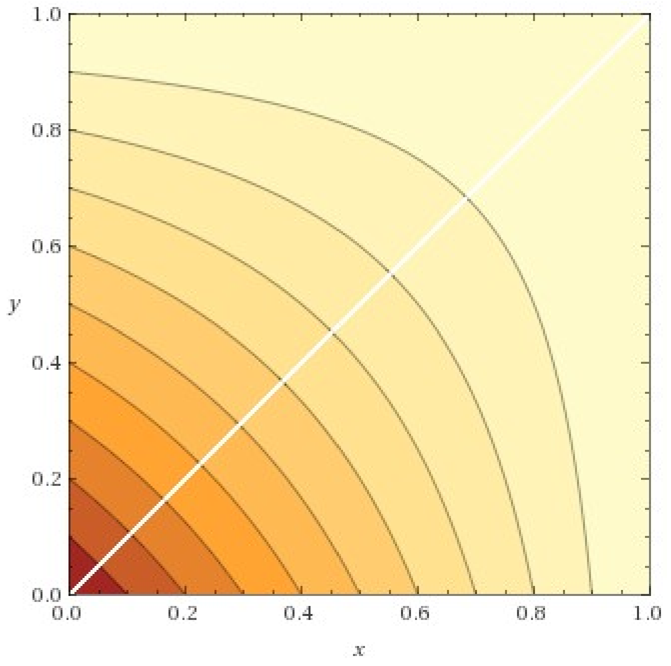
Firstly, the logical models that overestimate or underestimate only by degree within a relaxed model and not across a classification boundary are of no consequence and will not appear to over or under-estimate within the extracted logical model. In the toy problems being tested here there is always a perfect solution, and since models with a perfect classification boundary are always preferred even with an asymmetric cross entropy loss function, there is no resultant bias in the model given that the model is able to find a perfect logical model.

Secondly, there are wider considerations brought forwards by the potential success of an asymmetric loss function of this sort. Given that at the beginning of this learning process we had no logical model, and thus could only derive falsehoods anyway, an incomplete theory that used an asymmetric cross entropy loss function would not prove any new incorrect things. In inductive programming it can be desirable to chain inference tasks in series Muggleton et al. 2015, with each new task able to use the new predicates learned by the previous, and thus enabling the learning of more complex logical models for difficult predicates, consisting of additional invented intermediate predicates than would otherwise be possible. it is unclear how this could be adapted to allow the learning of deeper logical models on a single task. In the Neural-Symbolic ILP we could simply learn a theory that explains data as best we can, add our predicates to the background knowledge and repeat. The issue is however that a more complex model would still be committed to the compromises of the original model, as if something can be proved one way, it will remain provable at a later time, even if we do not wish it to be so. For instance, a logic program that initially either is correct or overestimates all facts would not be improvable by adding further predicates unless we are able to remove or modify previous predicates. To create a system that could remove or modify previous predicates would be a complex task, however an asymmetric loss function could solve this issue to some extent as a logic program that is instead either correct or underestimates target facts can be improved upon, and a series of strict asymmetric loss function-based inductions would be able to improve upon each other without needing to rewrite previous rules, only adding. In summary, underestimation can be easily improved in simple logic programs, whereas overestimation cannot.

3.5.3 Amalgamation

One important difference between the approaches in Evans and Grefenstette 2017 and Campero et al. 2018 is how facts that have been generated by learned rules are re-incorporated into the current set of accepted facts and their scores. In Logic Programming there is no need to consider this question too carefully, as all proofs prove something to the same degree. In Neural-Symbolic ILP alternative proofs of the same thing can provide different scores for the resultant fact. What is thus required is an OR function between new and old rule applications. DBLP:journals/corr/abs-1809-02193 uses the simplest method where facts values are updated whenever and only a new greater value can be generated, with that new value. An alternative is the probabilistic sum used $a + b - a * b$.

Figure 2: $x + y - (x * y)$

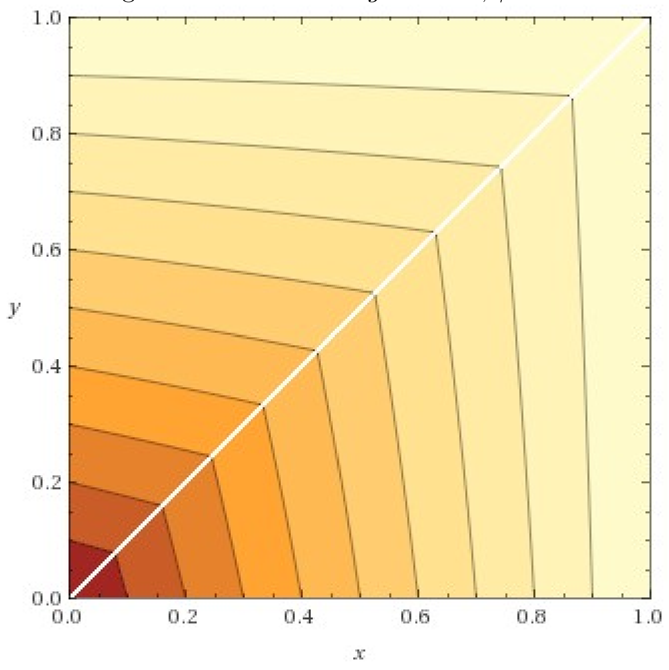


Simply amalgamating new information with a max operation is efficient and simple, and Campero et al. 2018 shows that a system that uses max in this way can learn a variety of problems, however using a simple max comes with a huge cost. This fact is noted in Evans and Grefenstette 2017 gradients are only defined for the maximum of a max operation, and information is lost about all other potentially important or interesting ways of proving a given fact. Unfortunately, there are also problems with the simple probabilistic sum. As can be seen in the plot , the probabilistic sum is only a close approximation of the max operation for values close to 1 and has the property that repeated application to low values outputs much higher values. For Neural Symbolic ILP this means that if we are likely to see a large number of not very good ways of proving a fact, the fact will nevertheless be given a high value in the same way as if there was a good way of proving the given fact. This makes the probabilistic sum an unfaithful relaxation of logic programs when too many alternative ways to prove a fact are considered and will result in models that may still find a good classification boundary in the relaxed setting, but will not converge to a representation of a real logic program. An effective compromise could be a mix of the max operation and a probabilistic sum:

$$mixed\_amalgamation(X, Y) = max(x, y) + \gamma(min(x, y) - x * y) \tag{9}$$

This function shares the desirable properties of the probabilistic sum in its bounds and the gradient flows, but through the adjustment of gamma can be a

Figure 3: $Mixed\_Amalgamation, \gamma = 0.3$



much more faithful approximation of a max, and remain a good approximation even after multiple applications.

If there is concern over a gamma that is too high not allowing large enough effects to flow to the lesser of the inputs then gamma can be varied during training.

3.6 Modifications of the Setting for Basic Numerical Computation

The most general form of datalog rule used in the neural-symbolic systems is:

$$Predicate(var, var) \leftarrow Predicate(var, var), Predicate(var, var) \quad (10)$$

where each predicate and variable can be any predicate or variable symbol, for example:

$$P(X, Y) \leftarrow P(X, Z), Q(Z, Y) \quad (11)$$

or

$$P(X) \leftarrow R(X, Z) \quad (12)$$

Generally the arrangement of variables is specified beforehand but this is not necessarily the case. Even not considering the need to learn the pattern in which variables appear or the number and arity of predicates there are limitations to this setting. Function symbols do not appear (as is the case in much of logic programming), and these approaches can not have models that add, multiply, or act numerically in the way that current machine learning does. This restriction is not a fatal flaw as there are many interesting areas in which these problems do not occur, however neural symbolic approaches do have the potential due to end-to-end training to incorporate numerical features, and interesting problems may well have a mix of categorical, numerical and relational data whilst also incorporating background knowledge, potentially a scenario in which end to end neural symbolic methods could be highly effective.

The feed forward approach can be modified by incorporating a mix of flexible and pre-set numerical operations and functions, in similar ways to Manhaeve et al. 2018 although with more numerical operations being specified and constructed within the ILP system itself. End to end learning means that these operations can be optimised by gradient descent over parameters, or can be selected over using SoftMax weights as ordinary logical statements are in Evans and Grefenstette 2017.

The general form of this numerical ILP rule would be:

$$Predicate(variable, variable) \leftarrow \tag{13}$$

Such models no longer represent logic programs as they are used in some areas as a resolution procedure must test every possible option. Efficient querying can still be attained for programs in this setting; however a mix of forward and backward chaining could be needed. Models constructed from such statements can still be severely limited in what sorts of things they can represent, however.

In the unmodified setting the pattern of variables and arity and number of predicates is usually pre-set, allowing a more focused search that can still learn interesting classes of logic. Thus, in the name of simplicity, and also in the same way as ILP operates usually, a simpler problem and type of rule is considered to explore the potential of this extended setting.
The rules considered within the restricted model are:

$$1)Predicate(Variable) \leftarrow \tag{14}$$

$$2)Predicate(Variable) \leftarrow \tag{15}$$

Or alternatively If we wish to learn the threshold as a parameter instead of using a range of given possible thresholds:

$$2)Predicate(Variable) \leftarrow \tag{16}$$

Also of note is the way in which we deal with the now tabular form of data, each row is given an id when it is placed within a minibatch for learning, and these ids are processed as additional arguments to the predicates within the rules. These restricted rules can represent simple decision rules, and it should be obvious how one could implement simple models such as decision trees using similar types of rules.

The problem presented to this model was one of creating a simple AND decision rule that required two independent thresholds to be chosen. For this purpose, I allowed two predicates of type 2 and one of type 1. Each example consisted of three features and a target predicate, where the three features were numbers between one and five, the correct rule for determining the truth of the target predicate would simply be that the numbers corresponding to the first two features for each example be greater than or equal to two. For example:

feature one	feature two	feature three	target
1	1	1	false
2	2	1	true
1	2	1	false
3	2	2	true

While extremely simple for a human or most machine learning algorithms, this problem can be ill-conditioned for these neural symbolic ILP systems, especially when there is a small amount of data. There are opportunities to overestimate as well as opportunities to find imperfect but good solutions from which we are required to simultaneously move multiple embeddings in ways which would independently be harmful to our solution unless co-ordinated.

If for instance it is learned initially to have a single threshold of one feature, to add another in order to create an AND rule would require changing three embeddings simultaneously to merely not reduce accuracy – the embedding of the head of that rule and the embedding of a body atom in a type 1 rule, and the head of that type 1 rule to the embedding of the target predicate. If we pre-set our embeddings to favour such AND rules, some of these issues are removed and the learning can occur easily. However, this problem was chosen as discussed earlier, precisely because it demonstrates what was hard about the two-children case from the previous section, and attempts to magnify these issues in order to test the limits of Neural-Symbolic ILP systems as opposed to merely demonstrating their strengths. Given this simplified but still new task, the Evans and Grefenstette 2017 method appears to take an extremely long time to construct and consider the potential clauses, and could be considered more or less equivalent to a simple exhaustive symbolic search, and on the other hand the Campero et al. 2018 method without modifications completely fails to overcome these difficulties and fails to learn the correct decision rule.

In addition to the adaptations described in the previous section some further changes to the neural symbolic ILP approaches were considered:

3.6.1 SoftMax Scaling

With SoftMax choices over thresholds and feature selections we are confronted by the fact that for the low weights that were being experienced, single rules tend not to dominate. Scaling the initialisation of these weights is one option, another is to scale the inputs to the SoftMax function, which has a similar effect.

It was the case that scaling the inputs to the SoftMax function had the better impact on learning, perhaps due to the fact that this magnified the speed at which the model moved towards preferred choices.

3.6.2 Similarity Penalties

In feed forward neural networks repetition is avoided within parts of a network by having a high variety of initial conditions. For reasons discussed earlier this was difficult within these neural symbolic ILP systems, when multiple rule templates of the same type were introduced whilst simultaneouosly having small initialisation values it would indeed seem that the problem of both rules being learned identically does in fact arise. One approach to solving this issue was with an explicit similarity penalty to ensure rules are learned differently. Such a similarity penalty forms an auxiliary part of the model's overall loss and is calculated by combining some embedding similarity measure such that if a single embedding within a rule differs between rules, the two rules are considered totally different.

$$embedding\_score(x, y) = 1 - abs(x - y) \tag{17}$$

$$Total\_similarity\_score = \prod_{i=0}^{n} embedding\_score(i, i) \tag{18}$$

Applying similarity penalties to specific parts within a rule can act as a bias towards certain rule types that are not merely avoiding pointless repeats. For example, a similarity penalty for the embeddings of two body atoms within a single rule can discourage certain rules and encourage others.

3.6.3 Parameterised Sigmoid Thresholds

There are two ways of learning the correct point to cut for the purposes of decision thresholds, having a SoftMax-choice over alternative pre-set thresholds, and having a boundary that is learned through gradient descent. As this somewhat artificial problem is designed to create an intentionally difficult challenge for neural-symbolic ILP, I focused mostly on learning a choice over pre-set thresholds as learning discrete choices well is the main task that has been focused on up to this point. However, the alternative was also explored where thresholds were learned directly by gradient descent, this did still involve a SoftMax style choice over features to apply the threshold to, but each new choice to make is a large increase in the size of the hypothesis space and so this does represent a large simplification of the search problem. Conclusions about the difficulty of more complicated cases involving this sort of secondary learned operations have not been drawn, although the relative ease or difficulty of learning within a system that uses these thresholds in comparison to one that has pre-defined thresholds may be indicative.

In order to create an approximation of a hard boundary suitable for a gradient descent approach a Sigmoid functions was used.

$$Sigmoid(x) = \frac{1}{(1 + e^{-\gamma x})} \tag{19}$$

Where higher values of gamma create a sharper function and thus a more faithful approximation of the step function that represents a greater than operation for a fixed threshold. If a sigmoid is too sharp then the gradients will be very close to zero far from the threshold, and extremely high close to it, a smooth sigmoid is therefore more suitable for gradient descent. The mean of two sigmoids, one with a much higher gamma is what was used to create a function with a good approximation of a hard threshold, but with useful gradients throughout.

Figure 4: sigmoid function, large γ

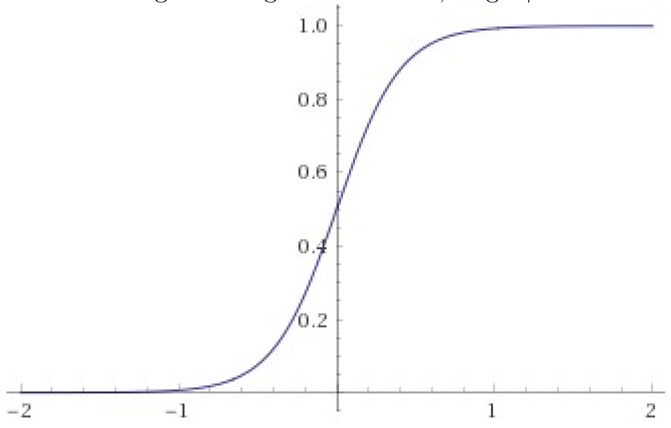


Figure 5: sigmoid function, small γ

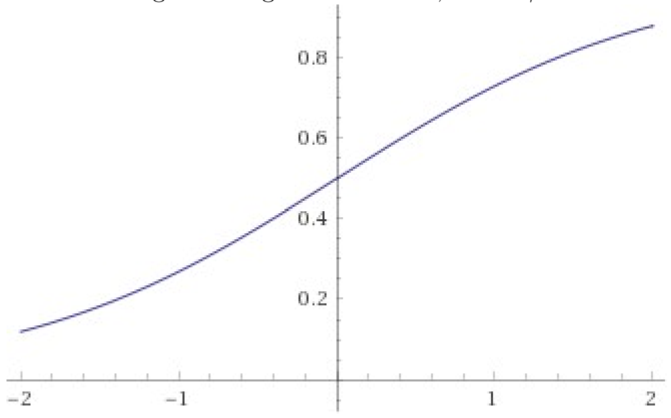
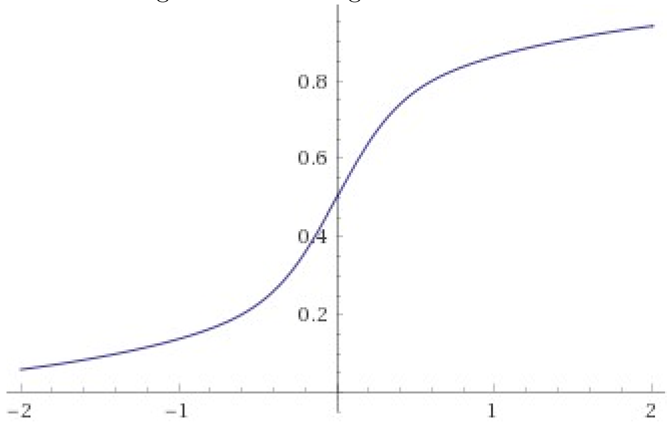
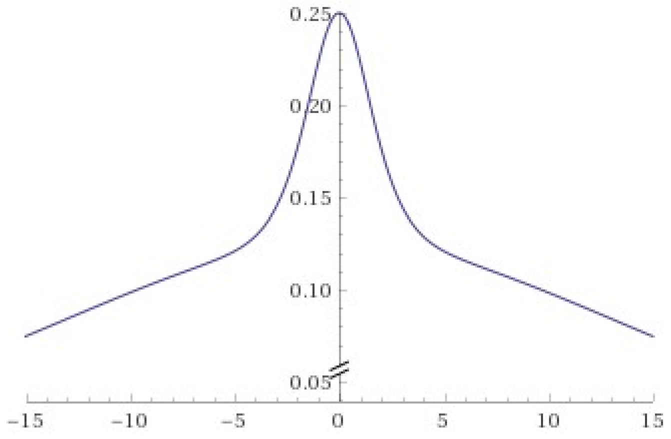


Figure 6: mix of sigmoid functions



As can be seen in the figure below, the derivative of this mixed sigmoid is higher in the region around the threshold than would be the case for a sigmoid with a generally high derivative far from the threshold, and higher at the threshold than a sigmoid with generally high derivatives far from the threshold. At least to a greater extent than a sigmoid that merely had its γ parameter set to create a threshold that was merely somewhat hard.

Table 2: derivative of a mix of sigmoid functions



3.6.4 Learning Level Changing with Deeper Representations

As discussed earlier when directly comparing the two approaches to feed-forward learning, the type of representations learned by a gradient descent model area of huge importance for both scalability and for the ability to learn complex sets of rules. Representations that optimise embeddings of individual predicate symbols are learning at the level of atoms within rules, whereas learning predicate definition weights as in Evans and Grefenstette 2017 is learning at the level of whole pairs of rules. Learning at a level higher than individual atoms e.g. the level of pairs of rules should allow the learning of more complex functions as more information is gained on the joint effects of the rules within the pairs. This additional information should be present at any increased level of learning, with the extreme of learning at the level of logic programs being similar to a simple combinatorial search through the hypothesis space, a situation in which gradient descent is extraneous. An interesting question is whether the lowest levels of representation can still generate enough information for learning of complex rules to occur, and whether the observed limits to current Neural-Symbolic ILP systems are a result of a lack of joint effect information of this sort.

It is plausible that some problems do not require understanding of these joint effects and thus are easier to learn, whereas some might require a great deal of information of this sort, additionally it might be hard to tell exactly what parts of a program should be learned jointly. The Evans and Grefenstette 2017 approach of learning at the level of pairs of rules is not the only option for higher level learning, with the main constraint being the scaling difficulties introduced. One approach to explore is that a learning system can learn with both an efficient low level representation and a high level one within the different parts of the same algorithm. Low level representations can be extracted from high level representations as a large soft-max weight matrix defining a choice of two parts of a rule can be converted into two separate weight vectors by summing the rows and columns, high level representations can be extracted in a similar way through the outer product. (It is not the space taken by the high level representation of the weights that is at issue however, and performing the sum operations does not significantly affect run-times, so it is possible to simply leave the weights in their higher level form throughout, which allows the representations to be agnostic to the feed forward evaluation learning level). The problem of selecting thresholds for a decision problem is attempted with a method where the learning is done partially with low level, and partially with higher level representations.

3.7 Results

Given the three tasks that have been discussed, the learning of the *two_children* predicate, the learning of correct choices in the numerical cut scenario amongst a selection of pre-existing cuts and alternatively learning cuts by gradient descent, three different systems, all based on the Campero et al. 2018 approaches emerged, each using at least some of the additions discussed above.

learning the *two_children* predicate proved the easiest task with all that was required was adjusted initialization to avoid an overly specific start point, a low learning rate to ensure careful exploration and the use of mixed amalgamation to ensure gradient flows. Reasons why each of these things should help were

	two_children	cut choices A	cut choices B	learned cuts
probabilistic sum	fails	required	fails	required
mixed amalgamation	required	not required	required	not required
adjusted initilization	required	required	required	required
asymetric loss	helps significantly	required	not required	required
softmax scaling	not required	required	required	required
similarity penalty	not required	required	not required	required
learning rate	0.001	0.1	0.003	0.1
other			*learning level change*	*sigmoid mix*

Table 3: A table showing the features of the three systems built

given above, and it is important to confirm that this simple problem is not the limit of these learning systems.

As expected, the hardest problem was the learning of correct cut choice amongst a pre-chosen selection. There were two successful approaches to this problem that were found. The first created was $Cut_Choices_B$ which uses a learning level change to learn instead of at the level of atoms, to learn at the level of atoms at most points, but for the choices of feature and cut point within the two rules that contained them, these would be learned jointly. This led to a massive slow-down in the run-time of the program, but as predicted did allow additional learning. minibatch learning was successfully used in this case to give a ceiling to the additional computation time. Alongside this significant change to the algorithm, the initialization was adjusted and SoftMax scaling was used. Full probabilistic sums failed in this case due to the increased number of evaluations made, as was discussed above this can lead to unwanted inflation to the valuations, to remedy this a mixed amalgamation was used, this further adjustment proved successful.

The alternative approach to this problem, $Cut_Choices_A$, did not utilise a learning level change and thus also did not require mixed amalgamation, instead the joint effects that were discovered through this level change were encouraged through the use of a similarity penalty on the two predicates involving cut choices. Additionally, a final required adjustment was the use of an asymmetric loss function, incidentally an asymmetric loss function also dramatically sped up the learning in the *two_children* case. This approach learned much faster than the previous that utilised level changing, although the use of similarity penalties is not likely to generalise to solve all difficult choice problems over a large number of choices as it was in this case. This approach used all of the applicable additions bar mixed amalgamation, although that does not appear to hurt the learnability in this case either. The final problem of parameterised cut points learned by gradient descent is also solvable by this third approach, given the use of a mix of sigmoids to represent the cut points themselves.

As this third approach that uses all but the level change proved the most successful learner (the *two_children*) that solves all of these three problems selected for difficulty or novelty with a fast run-time, I would suggest that its methods are an excellent starting point for any more developed neural-symbolic inductive logic programming system. Although the usefulness of learning level changes should be remembered, especially when it seems that aspects of the cut choices A style model like the similarity penalty may fail to generalise across settings.

3.8 Future Directions for These Methods

A question that can be asked of all these systems for neural inductive logic programming, is whether they really are any better than the similar purely symbolic approaches. Before the addition of these numerical operations to the logical rules I do not think that there was any concrete area in which current logic-learning systems (such asCropper and Muggleton 2016) or modern Answer Set Programming solvers (such as Gebser et al. 2012) were at a clear disadvantage to neural-symbolic methods. The addition of noise to a dataset can be dealt with within ASP, and probabilistic inductive logic programming is a highly active research area, with inductive programming contributing Bayesian approaches such as Muggleton et al. 2013.The potential of adding more differentiable mathematical operations as subsystems within a rule-learning system could allow an extremely general machine learning system that if able to learn efficiently would be very useful for both problems where computational logic is currently used, and machine learning tasks that are currently considered unsuitable for such symbolic methods For example, classification over large datasets with a mix of categorical and numerical data. I remain open-minded as to whether the straightforward but extremely hard task of learning a set of very complex rules to solve a classification task would be better solved by neural-symbolic or purely symbolic approaches, although there is sufficient promise to Neural-symbolic Inductive Logic Programming approaches that they should be developed towards this goal.

If I were to explore the topics within this project further, there are many larger modifications that could improve or expand upon these approaches. A method for the efficient use of rules within the knowledge base that are not to be learned but whose consequences should be added, the pre-computation approach of Evans and Grefenstette 2017, although scaling badly for learnt rules, could be precisely what is required for rules already needed in the knowledge base. This would allow multiple learning operations on the same goal in sequence, which can lead to logical models with significantly more invented predicates. An expansion upon the already numerically expanded setting is also possible, a general form for more expanded rule templates should be designed to allow additional numerical classification models to be represented.

4 Conclusion

This project gave an overview of the different approaches to neural-symbolic learning, focusing on those where the learning of logical rules was the main aim, whether within neural networks themselves, or using neural network like techniques. As a result of this, the inductive logic programming task was targeted, with the methods found in Evans and Grefenstette 2017 and Campero et al. 2018 discussed in detail. The limits to these methods, when it comes to expressibility, scaling and difficulty with learning complex rules was explored. These models are limited to datalog, and this creates severe limits to expressivity, the Evans and Grefenstette 2017 approach scales in a significantly worse way to Campero et al. 2018, and so the later was taken to be the basis for further exploration. Difficult cases and themes of problems for this model were identified, with the *two_children* faliure case, and the requirement for linked choices singled out. A new problem was then formulated that tested the potential of these approaches to adapt to a setting with additional expressivity and difficult problems to learn. The *two_children* problem was used, as was a decision problem which required non-independent cut off points to be learned through gradient descent over multiple features for multiple examples that represented an advancement in expressivity of these models, also used was a problem where a selection of pre-chosen cut points had been made, over which a model had to choose in order to create a difficult situation where there was an increasingly sharp loss landscape. By considering and testing a number of modifications to the Campero et al. 2018 approach, which included adding some elements of the higher level learning approach of Evans and Grefenstette 2017, some selection of these modifications was able to learn and overcome each task. This success suggests that the limits to neural-symbolic models in general, and feed-forward ILP in particular have not yet been reached. This success also provides motivation for further exploration of the modifications used, and their application to a wider range of problems.

References

Bengio, Y., J. Louradour, R. Collobert, and J. Weston
2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, Pp. 41–48, New York, NY, USA. ACM.

Campero, A., A. Pareja, T. Klinger, J. Tenenbaum, and S. Riedel
2018. Logical rule induction and theory learning using neural theorem proving. *CoRR*, abs/1809.02193.

Ceri, S., G. Gottlob, and L. Tanca
2012. *Logic Programming and Databases*, 1st edition. Springer Publishing Company, Incorporated.

Cropper, A. and S. Muggleton
2016. Learning higher-order logic programs through abstraction and invention. In *IJCAI*.

Donadello, I., L. Serafini, and A. S. d'Avila Garcez
2017. Logic tensor networks for semantic image interpretation. *CoRR*, abs/1705.08968.

Dong, H., J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou
2019. Neural logic machines.

Dua, D. and C. Graff
2017. UCI machine learning repository.

Evans, R. and E. Grefenstette
2017. Learning explanatory rules from noisy data. *CoRR*, abs/1711.04574.

Fernandez, R., A. Çelikyilmaz, P. Smolensky, and R. Singh
2018. Learning and analyzing vector encoding of symbolic representation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*.

Garcez, A. S. d., D. M. Gabbay, and K. B. Broda
2002. *Neural-Symbolic Learning System: Foundations and Applications*. Berlin, Heidelberg: Springer-Verlag.

Gebser, M., B. Kaufmann, and T. Schaub
2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89.

Getoor, L. and B. Taskar
2007. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Girshick, R. B.
2015. Fast R-CNN. *CoRR*, abs/1504.08083.

Glorot, X. and Y. Bengio
2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, eds., volume 9 of *Proceedings of Machine Learning Research*, Pp. 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Graves, A., G. Wayne, and I. Danihelka
2014. Neural turing machines. *CoRR*, abs/1410.5401.

Grefenstette, E.
2013. Towards a formal distributional semantics: Simulating logical calculi

with tensors. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*, Pp. 1–10, Atlanta, Georgia, USA. Association for Computational Linguistics.

Hornik, K.
1991. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257.

Huang, Q., P. Smolensky, X. He, L. Deng, and D. O. Wu
2018. Tensor product generation networks for deep NLP modeling. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, Pp. 1263–1273.

Lai, T. M., T. Bui, and S. Li
2018. A review on deep learning techniques applied to answer selection. In *Proceedings of the 27th International Conference on Computational Linguistics*, Pp. 2132–2144, Santa Fe, New Mexico, USA. Association for Computational Linguistics.

Manhaeve, R., S. Dumancic, A. Kimmig, T. Demeester, and L. D. Raedt
2018. Deepproblog: Neural probabilistic logic programming. *CoRR*, abs/1805.10872.

McCoy, R. T., T. Linzen, E. Dunbar, and P. Smolensky
2018. Rnns implicitly implement tensor product representations. *CoRR*, abs/1812.08718.

Mikolov, T., K. Chen, G. S. Corrado, and J. Dean
2013. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

Muggleton, S., D. Lin, J. Chen, and A. Tamaddoni-Nezhad
2013. Metabayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In *ILP*.

Muggleton, S., D. Lin, and A. Tamaddoni-Nezhad
2015. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100:49–73.

Palangi, H., P. Smolensky, X. He, and L. Deng
2018. Question-answering with grammatically-interpretable representations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Pp. 5350–5357.

Rocktäschel, T. and S. Riedel
2017. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds., Pp. 3788–3800. Curran Associates, Inc.

Schlag, I. and J. Schmidhuber
2018. Learning to reason with third order tensor products. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., Pp. 9981–9993. Curran Associates, Inc.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov
2017. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Serafini, L. and A. S. d'Avila Garcez
2016. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *CoRR*, abs/1606.04422.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis
2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Smolensky, P.
2012. Symbolic functions from neural computation. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 370 1971:3543–69.

Smolensky, P., M. Goldrick, and D. Mathis
2014. Optimization and quantization in gradient symbol systems: A framework for integrating the continuous and the discrete in cognition. *Cognitive Science*, 38(6):1102–1138.

Sukhbaatar, S., A. Szlam, J. Weston, and R. Fergus
2015. End-to-end memory networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, Pp. 2440–2448, Cambridge, MA, USA. MIT Press.

5 Appendix

Adapted from www.github.com/ACampero/diff-Theory-ILP.git :

5.1 *two_children* Solution

```
import numpy
import torchtext
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F

torch.manual_seed(0)

#################### two children
##-----------DATA-------------
background_predicates =[0,1]
intensional_predicates=[2,3]

##Background Knowledge
num_constants = 5

neq_extension = torch.ones(num_constants, num_constants)
for i in range(num_constants):
    neq_extension[i,i] = 0

edge_extension = torch.zeros(num_constants, num_constants)
edge_extension[0,1] = 1
edge_extension[0,2] = 1
edge_extension[1,3] = 1
edge_extension[2,3] = 1
edge_extension[2,4] = 1
edge_extension[3,4] = 1

num_constants_2 = 5
edge_extension_2 = torch.zeros(num_constants_2, num_constants_2)
edge_extension_2[0,1] = 1
edge_extension_2[1,2] = 1
edge_extension_2[1,3] = 1
edge_extension_2[2,2] = 1
edge_extension_2[2,4] = 1
edge_extension_2[3,4] = 1

#Intensional Predicates
target_extension = torch.zeros(num_constants, num_constants)
aux_extension = torch.zeros(num_constants,num_constants)

valuation_init = [Variable(neq_extension), Variable(edge_extension)
    , Variable(aux_extension), Variable(target_extension)]

##Target
target = Variable(torch.zeros(num_constants,num_constants))
target[0,0] = 1
target[2,2] = 1

```

```python
#Intensional Predicates
target_extension_2 = torch.zeros(num_constants_2, num_constants_2)
aux_extension_2 = torch.zeros(num_constants_2, num_constants_2)

valuation_init_2 = [Variable(neq_extension), Variable(edge_extension_2), Variable(aux_extension_2), Variable(target_extension_2)]

##Target
target_2 = Variable(torch.zeros(num_constants_2, num_constants_2))
target_2[1,1] = 1
target_2[2,2] = 1

steps = 2

num_rules = 2
rules_str = [15,3]

## 3 F(x,y)<--- F(x,Z),F(Z,Y)

## 15 F(X,X) <--- F(X,Z), F(X,Z)

num_predicates= len(valuation_init)
num_intensional_predicates = len(intensional_predicates)
num_feat = num_predicates

##---------FORWARD CHAINING---------
def decoder_efficient(valuation, step, num_constants):
    ##Unifications
    rules_aux = torch.cat((noisy_rules[:,:num_feat],noisy_rules[:,num_feat:2*num_feat],noisy_rules[:,2*num_feat:3*num_feat]),0)
    rules_aux = rules_aux.repeat(num_predicates,1)
    embeddings_aux = noisy_embeddings.repeat(1,num_rules*3).view(-1,num_feat)
    unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(num_predicates,-1)

    ##Get_Valuations
    valuation_new = [valuation[i].clone() for i in background_predicates] + \
                    [Variable(torch.zeros(valuation[i].size())) for i in intensional_predicates]

    for predicate in intensional_predicates:
        for s in range(num_constants):
            for o in range(num_constants):
                max_score = Variable(torch.Tensor([0]))
                for rule in range(num_rules):
                    if rules_str[rule] == 3:
                        for body1 in range(num_predicates):
                            if valuation[body1].size()[1] > 1:
                                for body2 in range(num_predicates):
                                    if valuation[body2].size()[1] > 1:
                                        unif = unifs[predicate][rule]*unifs[body1][num_rules+rule]*unifs[body2][2*num_rules+rule]
                                        num = torch.min(valuation[body1][s,:], valuation[body2][:,o])
                                        num = torch.max(num)
                                        score_rule = unif*num
                                        max_score = amalgamate2(max_score, score_rule)
                    elif rules_str[rule] == 15 and s==o:
                        for body1 in range(num_predicates):
                            if valuation[body1].size()[1] > 1:
                                for body2 in range(num_predicates):
                                    if valuation[body2].size()[1] > 1:
                                        unif = unifs[predicate][rule]*unifs[body1][num_rules+rule]*unifs[body2][2*num_rules+rule]
                                        num = torch.min(valuation[body1][s,:], valuation[body2][s,:])
                                        num = torch.max(num)
                                        score_rule = unif*num
                                        max_score = amalgamate2(max_score, score_rule)
                valuation_new[predicate][s,o] = amalgamate(valuation[predicate][s,o], max_score)
    return valuation_new

def soft_amalgamate(x,y):
    return x + y - x*y
def hard_amalgamate(x,y):
    return torch.max(x,y)
def mix_amalgamate(x,y):
    return torch.max(x,y)+0.8*(torch.min(x,y)-x*y)
def mix_amalgamate2(x,y):
    return torch.max(x,y)+0.5*(torch.min(x,y)-x*y)
amalgamate2 = mix_amalgamate
amalgamate = mix_amalgamate2
def AsymCrossEntropy(yHat, y):
    return y*0.5*-torch.log(yHat+0.000001)  + (1-y)*-torch.log(1-yHat+0.000001)

##--------SETUP--------
num_iters = 700
learning_rate = .001

embeddings = Variable(torch.eye(num_feat), requires_grad=True)

rules = Variable(torch.rand(num_rules, num_feat*3)/40, requires_grad=True)
# rule1 = torch.Tensor([0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,1,0]).view(1,-1)
# rule2 = torch.Tensor([0,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,0]).view(1,-1)
# rule3 = torch.Tensor([0,1,0,0,1,0,0,1,0]).view(1,-1)
# rules = Variable(torch.cat((rule1,rule2),0), requires_grad=True)

optimizer = torch.optim.Adam([
             embeddings,
             rules], lr=learning_rate)
```

```
criterion = torch.nn.BCELoss(size_average=False)

##---------TRAINING---------
loss = 0
for epoch in range(num_iters):
    #noise was not used
    hyper_epsilon_r = 0.0
    epsilon = torch.randn(rules.size())
    noisy_rules = rules + hyper_epsilon_r*epsilon
    noisy_rules = noisy_rules.detach().clamp_(min=0.0,max=1.0)
    noisy_rules = noisy_rules - rules.detach() + rules
    noisy_embeddings = embeddings


    for par in optimizer.param_groups[:]:
        for param in par['params']:
            param.data.clamp_(min=0.,max=1.)

    optimizer.zero_grad()
    fi = False
    if epoch%2==0:
        valuation = valuation_init
        const_aux = num_constants
        target_aux = target
        fi = True

    else:
        valuation = valuation_init_2
        const_aux = num_constants_2
        target_aux = target_2
        fi = False

    for step in range(steps):
        valuation = decoder_efficient(valuation,step,const_aux)
    loss = loss+torch.sum(AsymCrossEntropy(valuation[-1],target_aux
    ))
    #each iteration calculates for a seperate example, we do not
    update untill we have calculated for all examples
    if fi:
        print(epoch,'losssss',loss.item())
    if epoch<num_iters-1 and fi :
        loss.backward()
        optimizer.step()
        loss = 0.0

# ##---------PRINT RESULTS---------
# print('embeddings', embeddings)
# print( 'rules',rules)
# rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*
    num_feat],rules[:,2*num_feat:3*num_feat]),0)
# rules_aux = rules_aux.repeat(num_predicates,1)
# embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,
    num_feat)
# unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(
    num_predicates,-1)
# print('unifications',unifs)
# #=====
# valuationa = valuation_init
# const_auxa = num_constants
# target_auxa = target
# #====
# valuationb = valuation_init_2
# const_auxb = num_constants_2
# target_auxb = target_2
# for step in range(steps):
#         #pdb.set_trace()
#     valuationa = decoder_efficient(valuationa,step,const_aux)
# accu = torch.sum(torch.round(valuationa[-1])==target).item()#data
    [0]
# print('accuracy',accu, '/', target.nelement())

# for step in range(steps):
#         #pdb.set_trace()
#     valuationb = decoder_efficient(valuationb,step,const_auxb)
# accu = torch.sum(torch.round(valuationb[-1])==target_2).item()#
    data[0]
# print('accuracy',accu, '/', target.nelement())
# #======
# print(target)
# print(valuationa[-1])
# print(target_2)
# print(valuationb[-1])
# print(torch.round(rules))
```

5.2 *cut_choices_A* Solution

```
import numpy as np
import torchtext
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import time

torch.manual_seed(0)
np.random.seed(0)
steps = 2
num_constants =14

background_predicates =[]
intensional_predicates=[0,1,2]

#Intensional Predicates
aux_extension = torch.zeros(num_constants,1)
aux_extension2 = torch.zeros(num_constants,1)
target_extension = torch.zeros(num_constants,1)

num_rules = 3
rules_str = [5,5,4]

##Valuation
valuation_init = [ Variable(aux_extension),Variable(aux_extension2)
    , Variable(target_extension)]
num_predicates= len(valuation_init)
```

```python
num_intensional_predicates = len(intensional_predicates)
num_feat = num_predicates

target =  torch.tensor([1.0,1.0,1.0,1.0,1.0,1.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]).view(-1,1)
f_data = torch.tensor([[2.0,2,1],[2,3,2],[3,2,3],[4,4,3],[2,2,1],[2,3,2],[3,2,1],[3,0,3],[1,3,3],[1,4,2],[1,2,3],[1,2,1],[2,1,3],[3,1,2]])


num_ex = f_data.shape[0]
id_int_predicates = [0,1,2]

#similarity penalty
def sim_penalty(params,limit = -1,second_limit = -1):
    sp = 0
    for p in range(0,len(params)):
        if second_limit > 0 and p>=limit:
            break
        for q in range(p+1,len(params)):
            if second_limit >0 and q>=limit:
                break
            pair_cost = 1
            for r in range(0,len(params[p].view(-1))):
                if limit>0 and r>=limit:
                    break
                pair_cost = pair_cost*(1-(params[p].view(-1)[r]-params[q].view(-1)[r]).abs())
            sp = sp + pair_cost
    return sp


##-------FORWARD CHAINING-------
def decoder_efficient(valuation, step, valperm, am,ep,training = False):
    #calculating scaling and amalgamation, as these are allowed to change over learning
    sm_mult  = sm_mult2*(inc_rate**ep)
    mul = minweight * (disc**ep)
    # extracting the rules from their weights and scaling them by the softmax scaling factor sm_mult
    rulesh sm = F.softmax((noisy_rulesh*sm_mult).view(-1),dim = 0).view(noisy_rulesh.shape)#!!!
    urulesh = torch.stack([rulesh sm.sum(dim = 2).sum(dim = 1),rulesh sm.sum(dim = 2).sum(dim = 0),rulesh sm.sum(dim = 0).sum(dim = 0)])
    rulesblocksm = F.softmax((noisy_rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(noisy_rulesblock.shape)
    rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm.sum(dim=1)),dim = 1)
    rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*num_feat],rules[:,2*num_feat:3*num_feat]),0)
    rules_aux = rules_aux.repeat(num_predicates,1)
    embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,num_feat)
    #performs unification to calculate distances
    unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(
    num_predicates,-1)
    #G - unifs the defines the degree of compatibility of each learned predicate embedding (dim1) and each learned predicate choice for each rule (dim2 is #rules*3)
    #G - where the first three values for some pred in unif are compatibility with the heads of each of the three predicates (as they stand)
    valuation_new = [valuation[i].clone() for i in background_predicates] + \
                    [Variable(torch.zeros(valuation[i].size())) for i in intensional_predicates]
    scweights2 = scweights
    scweightsm =  (F.softmax((scweights2*sm_mult).view(-1),dim = 0)).view(scweights2.shape)
    scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
    for id_pred in id_int_predicates:
        for id in range(0,len(valuation[-1])):
            # (if propositional)
            if valuation[id_pred].size()[1] == 1:
                max_score = Variable(torch.Tensor([0]))
                for rule in range(num_rules):
                    #F(id)<-F(id),F(id)
                    if rules_str[rule] == 4:
                        for body1 in id_int_predicates:
                            if valuation[body1].size()[1] == 1:
                                for body2 in id_int_predicates:
                                    if valuation[body2].size()[1] == 1:
                                        unif = unifs[id_pred][rule]*unifs[body1][num_rules+rule]*unifs[body2][2*num_rules+rule]
                                        num = valuation[body1][id,0]*valuation[body2][id,0]
                                        score_rule = unif*num
                                        max_score = amalgamate(max_score,score_rule,mul)
                    if rules_str[rule] == 5:
                        for sel_f in range(f_data.shape[1]):
                            sel = f_data[valperm[id]][sel_f]

                            comps = torch.tensor([sel>0,sel>1,sel>2,sel>3])
                            for  cw_i in range(0,4):
                                score_rule = unifs[id_pred][rule]*comps[cw_i]*scws[rule][sel_f][cw_i]
                                max_score = amalgamate(max_score,score_rule,mul)
                valuation_new[id_pred][id,0] = amalgamate(valuation[id_pred][id,0], max_score,mul)
    return valuation_new

##------SETUP------
num_iters = 50
#softmax scaling
sm_mult2 = 13.0
inc_rate = 1.0

```

```python
#amalgamtion mixing
minweight = 1.0
disc = 1.0
embeddings = Variable(torch.eye(3), requires_grad=True)

def amalgamate(x,y,mul):
    return torch.max(x,y)+mul*(torch.min(x,y)-x*y)

def AsymCrossEntropy(yHat, y):
    return y*0.3*-torch.log(yHat)  + (1-y)*-torch.log(1-yHat)

scweights = Variable(torch.rand(f_data.shape[1],4,f_data.shape[1],4)/100,requires_grad=True)
rulesh  = Variable(torch.rand(num_feat, num_feat,num_feat)/100, requires_grad=True)
rulesblock  = Variable(torch.rand(num_rules, num_feat,num_feat)/100,requires_grad=True)
optimizer = torch.optim.Adam([
            rulesblock,
            scweights,
             rulesh],lr = 0.1)
criterion = torch.nn.BCELoss(size_average=False)


##---------TRAINING--------
start = time.time()
for epoch in range(num_iters):
    for par in optimizer.param_groups[:]:
        for param in par['params']:
            param.data.clamp_(min=0.,max=1.)
    optimizer.zero_grad()
    valuation = valuation_init

    # allows minibatch learning
    #==================================
    batch_size =14
    val_permuted = []
    valperm = np.random.permutation(num_constants)
    for v in range(0,len(valuation_init)):
        val_permuted.append(valuation[v][valperm][0:batch_size])
    target_permuted = target[valperm][0:batch_size]
    #==================================
    valuation = val_permuted

    #noise is not used
    noisy_rulesblock = rulesblock
    noisy_rulesblock = noisy_rulesblock.detach().clamp_(min=0.0,max=1.0)
    noisy_rulesblock = noisy_rulesblock - rulesblock.detach() + rulesblock
    noisy_rulesh = rulesh
    noisy_scweights = scweights



    for step in range(steps):
        if step <= step*-1:
            valuation = decoder_efficient(valuation,step,valperm,True,epoch,training = True)
        else:
            valuation = decoder_efficient(valuation,step,valperm,False,epoch,training = True)

    #calculates similarity penalty
    ep = epoch
    sm_mult  = sm_mult2*(inc_rate**ep)
    mul = minweight * (disc**ep)
    scweightsm =  (F.softmax((scweights*sm_mult).view(-1),dim = 0)).view(scweights.shape)
    scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
    a = sim_penalty(scws)
    aux_loss = a

    loss = 0.2*aux_loss + torch.sum(AsymCrossEntropy(valuation[-1],target_permuted))
    print(1.0*epoch,'lossssssssssssssssssssssssssss',loss.item()/batch_size)#.data[0])
    if epoch<num_iters-1:
        loss.backward()
        optimizer.step()
    end = time.time()
    print("ep took:",end-start)
    start = time.time()


# ##————PRINT RESULTS————
# sm_mult  = sm_mult2*(inc_rate**num_iters)
# valuation = valuation_init
# for step in range(steps):
#     valuation = decoder_efficient(valuation,step,np.arange(num_constants),True,num_iters,training = False)
# urulesh = torch.stack([rulesh.sum(dim = 2).sum(dim = 1),rulesh.sum(dim = 2).sum(dim = 0),rulesh.sum(dim = 0).sum(dim = 0)])
# rules = torch.cat((urulesh,rulesblock.sum(dim = 2),rulesblock.sum(dim=1)),dim = 1)
# print('rules\n',rules)
# ruleshsm = F.softmax((rulesh*sm_mult).view(-1),dim = 0).view(rulesh.shape)#!!!
# urulesh = torch.stack([ruleshsm.sum(dim = 2).sum(dim = 1),ruleshsm.sum(dim = 2).sum(dim = 0),ruleshsm.sum(dim = 0).sum(dim = 0)])
# rulesblocksm = F.softmax((rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(rulesblock.shape)
# rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm.sum(dim=1)),dim = 1)
# print("==============================")
# print('extra information')
# print('rules')
# print(rules)
# print("sc")
# scweightsm =  F.softmax((scweights*sm_mult).view(-1),dim = 0).view(scweights.shape)
# scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),
    torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
# print(scws)
# print("==================================")
# rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*
    num_feat],rules[:,2*num_feat:3*num_feat]),0)
# rules_aux = rules_aux.repeat(num_predicates,1)
# embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,
    num_feat)
# unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(
    num_predicates,-1)
# print('unifications')
# print(unifs)
# print(target)
# print('val')
# print(valuation[-1][:7])
# print(valuation[-1][7:])
# accu = torch.sum(torch.round(valuation[-1])==target).item()
# print('accuracy',accu, '/', target.nelement())
```

5.3 *cut_choices_B* Solution

```
import numpy as np
import torchtext
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
from copy import deepcopy
import time

torch.manual_seed(0)
np.random.seed(0)
steps = 2
num_constants =14

background_predicates =[]
intensional_predicates=[0,1,2]

#Intensional Predicates
aux_extension = torch.zeros(num_constants,1)
aux_extension2 = torch.zeros(num_constants,1)
target_extension = torch.zeros(num_constants,1)

num_rules = 3
rules_str = [5,5,4]

##Valuation
valuation_init = [ Variable(aux_extension),Variable(aux_extension2)
    , Variable(target_extension)]
num_predicates= len(valuation_init)
num_intensional_predicates = len(intensional_predicates)
num_feat = num_predicates

##Data
target =  torch.tensor([1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    0.0,0.0,0.0,0.0,0.0,0.0,0.0]).view(-1,1)
f_data = torch.tensor
```

```
    ([[2.0,2,1],[2,3,2],[2,3,1],[3,2,3],[2,2,1],[2,2,2],[4,2,4],
    [3,1,3],[1,3,3],[1,4,2],[1,1,3],[1,2,1],[2,1,3],[3,1,1]])


num_ex = f_data.shape[0]
id_int_predicates = [0,1,2]


##------FORWARD CHAINING------
def decoder_efficient2(valuation, step, valperm, am,ep):
    #calculating scaling and amalgamation, as these are allowed to change over learning
    sm_mult  = bsm_mult2*(binc_rate**ep)
    mul = bminweight * (bdisc**ep)
    # extracting the rules from their weights and scaling them by the softmax scaling factor sm_mult
    ruleshsm = F.softmax((rulesh*sm_mult).view(-1),dim = 0).view(rulesh.shape)
    urulesh = torch.stack([ruleshsm.sum(dim = 2).sum(dim = 1),ruleshsm.sum(dim = 2).sum(dim = 0),ruleshsm.sum(dim = 0).sum(dim = 0)])
    rulesblocksm = F.softmax((rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(rulesblock.shape)
    rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm.sum(dim=1)),dim = 1)
    rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*num_feat],rules[:,2*num_feat:3*num_feat]),0)
    rules_aux = rules_aux.repeat(num_predicates,1)
    embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,num_feat)
    #performs unification to calculate distances

    unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(num_predicates,-1)
    #G - unifs the defines the degree of compatibility of each learned predicate embedding (dim1) and each learned predicate choice for each rule (dim2 is #rules*3)
    #G - where the first three values for some pred in unif are compatibility with the heads of each of the three predicates (as they stand)
    valuation_new = [valuation[i].clone() for i in background_predicates] + \
                    [Variable(torch.zeros(valuation[i].size())) for i in intensional_predicates]
    scweightsm =  (F.softmax((scweights*sm_mult).view(-1),dim = 0)).view(scweights.shape)
    scws = scweightsm
    for id_pred in id_int_predicates:
        for id in range(0,len(valuation[-1])):
            # (if propositional)
            if valuation[id_pred].size()[1] == 1:
                max_score = Variable(torch.Tensor([0]))
                for rule in range(num_rules):
                    #F(id)<-F(id),F(id)
                    if rules_str[rule] == 4:
                        for body1 in id_int_predicates:
                            if valuation[body1].size()[1] == 1:
                                for body2 in id_int_predicates:
                                    if valuation[body2].size()[1] == 1:
                                        unif = unifs[id_pred][rule]*unifs[body1][num_rules+rule]*unifs[body2][2*num_rules+rule]
                                        num = valuation[body1][id,0]*valuation[body2][id,0]
                                        score_rule = unif*num
                                        max_score = amalgamate(max_score,score_rule,mul)
                valuation_new[id_pred][id,0] = amalgamate(valuation[id_pred][id,0], max_score,mul)
    for id_pred in id_int_predicates:
      for id_pred2 in id_int_predicates:
            for id in range(0,len(valuation[-1])):
                # (if propositional)
                if valuation[id_pred].size()[1] == 1:
                    if valuation[id_pred2].size()[1] == 1:
                        max_score = Variable(torch.Tensor([0]))
                        max_score2= Variable(torch.Tensor([0]))
                        rule = 0
                        rule2 = 1
                        for sel_f in range(f_data.shape[1]):
                            for sel_f2 in range(f_data.shape[1]):
                                sel = f_data[valperm[id]][sel_f]
                                sel2 = f_data[valperm[id]][sel_f2]
                                comps = torch.tensor([sel>0,sel>1,sel>2,sel>3])
                                for  cw_i in range(0,4):
                                    for  cw_i2 in range(0,4):
                                        weight = scws[sel_f][cw_i][sel_f2][cw_i2]
                                        score_rule = unifs[id_pred][rule]*comps[cw_i]*weight
                                        score_rule2 = unifs[id_pred2][rule2]*comps[cw_i2]*weight
                                        max_score = amalgamate(max_score,score_rule,mul)
                                        max_score2 = amalgamate(max_score2,score_rule2,mul)
                        valuation_new[id_pred][id,0] = amalgamate(valuation[id_pred][id,0], max_score,mul)
                        valuation_new[id_pred2][id,0] = amalgamate(valuation[id_pred2][id,0], max_score2,mul)
    return valuation_new


def decoder_efficient(valuation, step, valperm, am,ep):
    sm_mult  = sm_mult2*(inc_rate**ep)
    mul = minweight * (disc**ep)
    ruleshsm = F.softmax((rulesh*sm_mult).view(-1),dim = 0).view(rulesh.shape)#!!!
    urulesh = torch.stack([ruleshsm.sum(dim = 2).sum(dim = 1), ruleshsm.sum(dim = 2).sum(dim = 0),ruleshsm.sum(dim = 0).sum(dim = 0)])
    rulesblocksm = F.softmax((rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(rulesblock.shape)
```

```python
    rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm.sum(dim=1)),dim = 1)
    rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*num_feat],rules[:,2*num_feat:3*num_feat]),0)
    rules_aux = rules_aux.repeat(num_predicates,1)
    embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,num_feat)
    unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(num_predicates,-1)
    #G - unifs the defines the degree of compatibility of each learned predicate embedding (dim1) and each learned predicate choice for each rule (dim2 is #rules*3)
    #G - where the first three values for some pred in unif are compatibility with the heads of each of the three predicates (as they stand)
    valuation_new = [valuation[i].clone() for i in background_predicates] + \
                    [Variable(torch.zeros(valuation[i].size())) for i in intensional_predicates]
    scweightsm =  (F.softmax((scweights*sm_mult).view(-1),dim = 0)).view(scweights.shape)
    scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
    for id_pred in id_int_predicates:
        for id in range(0,len(valuation[-1])):
            if valuation[id_pred].size()[1] == 1:
                max_score = Variable(torch.Tensor([0]))
                for rule in range(num_rules):
                    if rules_str[rule] == 4:
                        for body1 in id_int_predicates:
                            if valuation[body1].size()[1] == 1:
                                for body2 in id_int_predicates:
                                    if valuation[body2].size()[1] == 1:
                                        unif = unifs[id_pred][rule]*unifs[body1][num_rules+rule]*unifs[body2][2*num_rules+rule]
                                        num = valuation[body1][id,0]*valuation[body2][id,0]
                                        score_rule = unif*num
                                        max_score = amalgamate(max_score,score_rule,mul)
                    if rules_str[rule] == 5:
                        for sel_f in range(f_data.shape[1]):
                            sel = f_data[valperm[id]][sel_f]
                            comps = torch.tensor([sel>0,sel>1,sel>2,sel>3])#,sel>4,sel>5])
                            for  cw_i in range(0,4):
                                score_rule = unifs[id_pred][rule]*comps[cw_i]*scws[rule][sel_f][cw_i]
                                max_score = amalgamate(max_score,score_rule,mul)
                valuation_new[id_pred][id,0] = amalgamate(valuation[id_pred][id,0], max_score,mul)
    return valuation_new


##--------SETUP--------
```

```python
num_iters = 250
tr = 90
tr2 = 200
#softmax scaling
sm_mult2 = 13.0
inc_rate = 1.001
#amalgamtion mixing
minweight = 1.0
disc = 1.0
bsm_mult2 = sm_mult2*inc_rate**tr
binc_rate = 1.0
bminweight = minweight*disc**tr
bdisc = 1.0
embeddings = Variable(torch.eye(3), requires_grad=True)

def amalgamate(x,y,mul):
    return torch.max(x,y)+mul*(torch.min(x,y)-x*y)

scweights = Variable(torch.rand(f_data.shape[1],4,f_data.shape[1],4)/100,requires_grad=True)
rulesh  = Variable(torch.rand(num_feat, num_feat,num_feat)/100,requires_grad=True)
rulesblock  = Variable(torch.rand(num_rules, num_feat,num_feat)/100,requires_grad=True)
optimizer = torch.optim.Adam([
            rulesblock,
            scweights,
             rulesh],lr = 0.003)
criterion = torch.nn.BCELoss(size_average=False)

##----------TRAINING----------
start = time.time()

for epoch in range(num_iters):
    for par in optimizer.param_groups[:]:
        for param in par['params']:
            param.data.clamp_(min=0.,max=1.)
    optimizer.zero_grad()
    valuation = valuation_init

    #this uses minibatch learning to for a modest speed gain
    #======================================
    batch_size =8
    val_permuted = []
    valperm = np.random.permutation(num_constants)
    for v in range(0,len(valuation_init)):
        val_permuted.append(valuation[v][valperm][0:batch_size])
    target_permuted = target[valperm][0:batch_size]
    #======================================
    valuation = val_permuted
    for step in range(steps):
        if epoch <= tr:
            valuation = decoder_efficient(valuation,step,valperm,True,epoch)
        else:
            if epoch<=tr2:
                rulesh.requires_grad=False
                valuation = decoder_efficient2(valuation,step,valperm,True,epoch)
            else:
                rulesh.requires_grad=True
                scweights.requires_grad=False
                valuation = decoder_efficient(valuation,step,valperm,True,epoch)
    loss = criterion(valuation[-1],target_permuted)
    print(epoch,'losssssssssssssssssssssssssssss',loss.item()/batch_size)
    if epoch<num_iters-1:
        loss.backward()
        optimizer.step()
    end = time.time()
    print("ep took:",end-start)
    start = time.time()




# ##--------PRINT RESULTS--------
# sm_mult = bsm_mult2
# valuation = valuation_init
# for step in range(steps):
#     valuation = decoder_efficient(valuation,step,np.arange(num_constants),True,num_iters)
# urulesh = torch.stack([rulesh.sum(dim = 2).sum(dim = 1),rulesh.sum(dim = 2).sum(dim = 0),rulesh.sum(dim = 0).sum(dim = 0)])
# rules = torch.cat((urulesh,rulesblock.sum(dim = 2),rulesblock.sum(dim=1)),dim = 1)
# print('rules\n',rules)
# ruleshsm = F.softmax((rulesh*sm_mult).view(-1),dim = 0).view(rulesh.shape)#!!!
# urulesh = torch.stack([ruleshsm.sum(dim = 2).sum(dim = 1),ruleshsm.sum(dim = 2).sum(dim = 0),ruleshsm.sum(dim = 0).sum(dim = 0)])
# rulesblocksm = F.softmax((rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(rulesblock.shape)
# rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm.sum(dim=1)),dim = 1)
# print("==================================")
# print('extra information')
# print('rules')
# print(rules)
# print("sc")
# scweightsm =  F.softmax((scweights*sm_mult).view(-1),dim = 0).view(scweights.shape)
# scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
# print(scws)
# print("==================================")
# rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*num_feat],rules[:,2*num_feat:3*num_feat]),0)
# rules_aux = rules_aux.repeat(num_predicates,1)
# embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,num_feat)
# unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(
    num_predicates,-1)
# print('unifications')
# print(unifs)
# print(target)
# print('val')
# print(valuation[-1][:7])
# print(valuation[-1][7:])
# accu = torch.sum(torch.round(valuation[-1])==target).item()#.data
    [0]
# print('accuracy',accu, '/', target.nelement())
```

5.4 *learned_cuts* Solution

```
import numpy as np
import torchtext
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import time

torch.manual_seed(0)
np.random.seed(0)
steps = 2
num_constants =14

background_predicates =[]
intensional_predicates=[0,1,2]

#Intensional Predicates
aux_extension = torch.zeros(num_constants,1)
aux_extension2 = torch.zeros(num_constants,1)
target_extension = torch.zeros(num_constants,1)

num_rules = 3
rules_str = [5,5,4]

##Valuation
valuation_init = [ Variable(aux_extension),Variable(aux_extension2)
    , Variable(target_extension)]
num_predicates= len(valuation_init)
num_intensional_predicates = len(intensional_predicates)
num_feat = num_predicates

target =  torch.tensor([1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    0.0,0.0,0.0,0.0,0.0,0.0,0.0]).view(-1,1)
f_data = torch.tensor
    ([[2.0,2,1],[2,3,2],[3,2,3],[4,4,3],[2,2,1],[2,3,2],[3,2,1],
    [3,0,3],[1,3,3],[1,4,2],[1,2,3],[1,2,1],[2,1,3],[3,1,2]])


num_ex = f_data.shape[0]
id_int_predicates = [0,1,2]

#similarity penalty, modified to adjust which parts of a rule are
    compared
def sim_penalty(params,limit = -1,second_limit = -1):
```

```python
    sp = 0
    for p in range(0,len(params)):
        if second_limit > 0 and p>=limit:
            break
        for q in range(p+1,len(params)):
            if second_limit >0 and q>=limit:
                break
            # if not p ==q :
            pair_cost = 1
            for r in range(0,len(params[p].view(-1))):
                if limit>0 and r>=limit:
                    break
                pair_cost = pair_cost*(1-(params[p].view(-1)[r]-params[q].view(-1)[r]).abs())
            sp = sp + pair_cost
    return sp

##-----FORWARD CHAINING-----
def decoder_efficient(valuation, step, valperm, am,ep,training = False):
    #calculating scaling and amalgamation, as these are allowed to change over learning
    sm_mult  = sm_mult2*(inc_rate**ep)
    mul = minweight * (disc**ep)
    # extracting the rules from their weights and scaling them by the softmax scaling factor sm_mult
    ruleshsm = F.softmax((noisy_rulesh*sm_mult).view(-1),dim = 0).view(noisy_rulesh.shape)#!!!
    urulesh = torch.stack([ruleshsm.sum(dim = 2).sum(dim = 1),ruleshsm.sum(dim = 2).sum(dim = 0),ruleshsm.sum(dim = 0).sum(dim = 0)])
    rulesblocksm = F.softmax((noisy_rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(noisy_rulesblock.shape)
    rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm.sum(dim=1)),dim = 1)
    rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*num_feat],rules[:,2*num_feat:3*num_feat]),0)
    rules_aux = rules_aux.repeat(num_predicates,1)
    embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,num_feat)
    #performs unification to calculate distances
    unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(num_predicates,-1)
    #G - unifs the defines the degree of compatibility of each learned predicate embedding (dim1) and each learned predicate choice for each rule (dim2 is #rules*3)
    #G - where the first three values for some pred in unif are compatibility with the heads of each of the three predicates (as they stand)
    valuation_new = [valuation[i].clone() for i in background_predicates] + \
                    [Variable(torch.zeros(valuation[i].size())) for i in intensional_predicates]

    scweights2 = scweights
    scweightsm =  (F.softmax((scweights2*sm_mult).view(-1),dim = 0)).view(scweights2.shape)
    scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0), torch.unsqueeze(scweightsm.sum(2).sum(2),0)))

    for id_pred in id_int_predicates:
        for id in range(0,len(valuation[-1])):
            # (if propositional)
            if valuation[id_pred].size()[1] == 1:
                max_score = Variable(torch.Tensor([0]))
                for rule in range(num_rules):
                    #F(id)<-F(id),F(id)
                    if rules_str[rule] == 4:
                        for body1 in id_int_predicates:
                            if valuation[body1].size()[1] == 1:
                                for body2 in id_int_predicates:
                                    if valuation[body2].size()[1] == 1:
                                        unif = unifs[id_pred][rule]*unifs[body1][num_rules+rule]*unifs[body2][2*num_rules+rule]
                                        num = valuation[body1][id,0]*valuation[body2][id,0]
                                        score_rule = unif*num
                                        max_score = amalgamate(max_score,score_rule,mul)
                    if rules_str[rule] == 5:
                        for sel_f in range(f_data.shape[1]):
                            sel = f_data[valperm[id]][sel_f]
                            comps = sig2(sel-noisy_thresholds[rule]*5)
                            score_rule = unifs[id_pred][rule]*comps*scws[rule][sel_f][0]
                            max_score = amalgamate(max_score,score_rule,mul)
                valuation_new[id_pred][id,0] = amalgamate(valuation[id_pred][id,0], max_score,mul)
    return valuation_new

#sigmoid mixture defined
def sig2(x,a1=10,a2=1):
    return (sig(x,a1)+sig(x,a2))/2.0

def sig(x,a=5.0):
    return 1.0/(1.0+torch.exp(-x*a))

##-------SETUP-------
num_iters = 200
#softmax scaling
sm_mult2 = 5.0
inc_rate = 1.0
#amalgamation mixture
minweight = 1.0
#decrease in amalgamation mixture
disc = 1.0
embeddings = Variable(torch.eye(3), requires_grad=True)
def amalgamate(x,y,mul):
    return torch.max(x,y)+mul*(torch.min(x,y)-x*y)

#the assymetric loss function
```

```
def AsymCrossEntropy(yHat, y):
    return y*0.3*-torch.log(yHat)  + (1-y)*-torch.log(1-yHat)

#new adjusted initialisation
scweights = Variable(torch.rand(f_data.shape[1],1,f_data.shape
    [1],1)/100,requires_grad=True)
rulesh  = Variable(torch.rand(num_feat, num_feat,num_feat)/100,
    requires_grad=True)
rulesblock  = Variable(torch.rand(num_rules, num_feat,num_feat)
    /100,requires_grad=True)
thresholds = Variable(torch.ones(num_rules)/5, requires_grad=True)

optimizer = torch.optim.Adam([
            rulesblock,
            scweights,
            thresholds,
             rulesh],lr = 0.1)
criterion = torch.nn.BCELoss(size_average=False)


##---------TRAINING--------
start = time.time()


for epoch in range(num_iters):
    print("current cut points:",thresholds*5)
    thresholds.requires_grad=False
    #this ensures  that potential thresholds do not exceed
    reasonable limits
    for t in range(0,3):
        if thresholds[t]*5>5:
            thresholds[t] = thresholds[t]*np.random.rand()
    thresholds.requires_grad=True
    for par in optimizer.param_groups[:]:
        for param in par['params']:
            param.data.clamp_(min=0.,max=1.)
    optimizer.zero_grad()
    valuation = valuation_init

    #==============================
    #allowing potential minibatch learning
    batch_size =14
    val_permuted = []
    valperm = np.random.permutation(num_constants)
    for v in range(0,len(valuation_init)):
        val_permuted.append(valuation[v][valperm][0:batch_size])
    target_permuted = target[valperm][0:batch_size]
    #==============================
    valuation = val_permuted

    #noise is not used
    noisy_thresholds = thresholds
    noisy_thresholds = noisy_thresholds.detach().clamp_(min=0.0,max
    =1.0)
    noisy_thresholds = noisy_thresholds - thresholds.detach() +
    thresholds
    noisy_rulesh = rulesh
```

```python
    noisy_scweights = scweights
    noisy_rulesblock = rulesblock

    for step in range(steps):
        if step <= step*-1:
        # if step % 2 == 0:
            valuation = decoder_efficient(valuation,step,valperm,True,epoch,training = True)
        else:
            valuation = decoder_efficient(valuation,step,valperm,False,epoch,training = True)

    # calculation of the similarity penalty
    ep = epoch
    sm_mult  = sm_mult2*(inc_rate**ep)
    mul = minweight * (disc**ep)
    scweightsm =  (F.softmax((scweights*sm_mult).view(-1),dim = 0)).view(scweights.shape)
    scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
    a = sim_penalty(scws)
    aux_loss = a



    loss = 0.2*aux_loss + torch.sum(AsymCrossEntropy(valuation[-1],target_permuted))
    print(1.0*epoch,'lossssssssssssssssssssssssssss',loss.item()/batch_size)
    if epoch<num_iters-1:
        loss.backward()
        optimizer.step()
    end = time.time()
    print("ep took:",end-start)
    start = time.time()


# ##--------PRINT RESULTS--------
# sm_mult  = sm_mult2*(inc_rate**num_iters)
# valuation = valuation_init
# for step in range(steps):
#     valuation = decoder_efficient(valuation,step,np.arange(num_constants),True,num_iters,training = False)
# urulesh = torch.stack([rulesh.sum(dim = 2).sum(dim = 1),rulesh.sum(dim = 2).sum(dim = 0),rulesh.sum(dim = 0).sum(dim = 0)])
# rules = torch.cat((urulesh,rulesblock.sum(dim = 2),rulesblock.sum(dim=1)),dim = 1)
# print(thresholds)
# print('rules\n',rules)
# ruleshsm = F.softmax((rulesh*sm_mult).view(-1),dim = 0).view(rulesh.shape)#!!!
# urulesh = torch.stack([ruleshsm.sum(dim = 2).sum(dim = 1),ruleshsm.sum(dim = 2).sum(dim = 0),ruleshsm.sum(dim = 0).sum(dim = 0)])
# rulesblocksm = F.softmax((rulesblock*sm_mult).view(num_rules,-1),dim = 1).view(rulesblock.shape)
# rules = torch.cat((urulesh,rulesblocksm.sum(dim = 2),rulesblocksm
    .sum(dim=1)),dim = 1)
# print("================================")
# print('additional information')
# print('rules')
# print(rules)
# print("sc")
# scweightsm =  F.softmax((scweights*sm_mult).view(-1),dim = 0).
    view(scweights.shape)
# scws = torch.cat((torch.unsqueeze(scweightsm.sum(0).sum(0),0),
    torch.unsqueeze(scweightsm.sum(2).sum(2),0)))
# print(scws)
# print("================================")
# rules_aux = torch.cat((rules[:,:num_feat],rules[:,num_feat:2*
    num_feat],rules[:,2*num_feat:3*num_feat]),0)
# rules_aux = rules_aux.repeat(num_predicates,1)
# embeddings_aux = embeddings.repeat(1,num_rules*3).view(-1,
    num_feat)
# unifs = F.cosine_similarity(embeddings_aux, rules_aux).view(
    num_predicates,-1)
# print('unifications')
# print(unifs)
# print(target)
# print('val')
# print(valuation[-1][:7])
# print(valuation[-1][7:])
# accu = torch.sum(torch.round(valuation[-1])==target).item()
# print('accuracy',accu, '/', target.nelement())
```