Deep Learning and Reasoning

by Joseph Marcus Tungate
B913717

23COD290
Loughborough University
Supervisor: Dr. Andrea Soltoggio
Date Submitted: 27th August 2024

Abstract

There is a growing sentiment within the artificial intelligence community that methods which combine the machine learning and symbolic disciplines ought to be further investigated. It is believed that such methods may provide solutions to some of the issues currently faced by AI, such as a lack of explainability and a reliance on very large datasets.

This project proposes and implements Fuzzy Tensor Logic, a method for constructing neuro-symbolic agents. Fuzzy Tensor Logic is a first-order, fuzzy, and differentiable logic over the set of real-valued tensors. The logic allows for the construction of agents which combine deep learning and knowledge-based reasoning in a flexible and explainable way. A Python 3 implementation of Fuzzy Tensor Logic is developed which is used to demonstrate the method on classification and regression tasks.

Acknowledgements

I would like to give thanks to Dr. Andrea Soltoggio for his support, patience, and excellent lectures which inspired me to learn more about artificial intelligence. I would like to give additional thanks to the staff of Loughborough University for their continued help and understanding throughout my studies. Finally, I would like to give special thanks to my family and friends whose endless support and encouragement made completing this project possible.

Contents

Chapter 1

Introduction

There is a growing sentiment within artificial intelligence research that believes efforts should be directed towards combining the symbolic and connectionist schools of thought. So called neuro-symbolic systems look to jointly overcome the problems faced by either discipline alone, exhibit higher levels of performance, and address some of the issues surrounding the leading wave of AI systems such as a lack of explainability, overly large training sets, and high environmental impact.

This project aims to determine effective methods of integrating deep learning with symbolic knowledge-based approaches to construct neuro-symbolic AI systems. A class of neuro-symbolic agents is proposed and tools for producing them are developed. Example systems are then presented to demonstrate the effectiveness of these methods.

This introduction describes the motivating factors behind this project by addressing the current state of AI, comparing the differences between deep learning and knowledge-based systems, and describing the potential benefits of neuro-symbolic systems. Following this, project aims, objectives, and outcomes are proposed.

1.1 Motivation

1.1.1 The Importance of AI

The past decade has seen an explosion in the economic and cultural impact of artificial intelligence across the world. The European Parliamentary Research Service (2024) has valued the global AI market at over €130 billion, with estimates predicting growth to nearly €1.9 trillion by 2030. The United States, who have been the leaders in private AI investment for over a decade, have seen private investment grow from under \$15 million in 2013 to over \$65 million in 2023 (Stanford University Institute for Human-Centered AI, 2024). The rest of the world also sees high levels of private investment, with China receiving €7.3 billion, and the EU and UK collectively receiving €9 billion throughout 2023. Public investment is also growing, with programs such as the EU's Digital Europe programme aiming to provide €2.1 billion of investment throughout the 2020s.

This increase in investment has seen the rapid emergence of many novel AI technologies into both the working and private lives of people world over. Most recently, focus has been directed towards large language, multi-modal, and generative models (Stanford University Institute for Human-Centered AI, 2024), many of which aim for varying degrees of general intelligence, such as Google's Gemini, Open AI's GPT-4, and Microsoft's Copilot. Agents are being employed in a wide range of applications from real-time business data analytics and medical diagnosis to personal assistants and AI art generation.

As AI becomes increasingly pervasive, so has its public awareness. However, public opinion on AI has proven mixed. Ipsos (2023) surveys show that 52% of globally surveyed people express nervousness towards AI. Additionally, only 54% of those surveyed believe products using artificial intelligence provide more benefits than drawbacks. The surveys show a clear lack of trust in AI companies to protect data, respect privacy, and produce models which do not display bias towards protected characteristics. There also persist fears over the impact of AI on the job market and the potential misuse of AI for nefarious purposes.

It's clear that AI plays a very important role in the daily lives of businesses and individuals alike. As time progresses, this importance is only going to increase as agents become increasingly sophisticated and prevalent. It is therefore crucial that AI progresses towards producing transparent, trustworthy, and sustainable systems which compliment, rather than compete with, the aims and beliefs of the majority of the population. It is not clear, however, whether the current approach to AI is likely to meet these goals. As will be shown, agents are becoming increasingly difficult to understand, trust, and produce sustainably.

1.1.2 Deep Learning

The principles behind the latest and most successful AI agents are deeply rooted in machine learning, particularly those of deep learning. Simply put, deep learning models consist of a long (or deep) chain of parameterised tensor operations. When given an input, the model passes the data through this chain of operations to produce a corresponding output. To train a deep learning model to perform a desired task, the developer first defines a loss metric, which is a measure of how 'wrong' an agent's output is with respect to its input. Then, the loss is computed on a large set of examples and the model's parameters are updated slightly to lower the loss. This process is repeated until the loss is satisfactorily low. The process of lowering the loss is done by gradient descent. This is made possible by the model's design supporting the differentiation of its parameters with respect to the loss. The back-propagation algorithm is a popular method for implementing gradient descent (Goodfellow, Bengio, and Courville, 2017).

While advances in deep learning are to thank for the sophistication of current agents, they also prove to be the source of many of the technical and cultural issues they face. One of the major problems with deep learning, which is of concern to both practitioners and the public, is that deep learning models aren't easily explainable (Dingli and Farrugia, 2023). In other words, most deep learning algorithms produce black-box agents whose responses prove very difficult to both explain and justify. As the size and scale of agents increases, this problem is exacerbated making it particularly relevant for the cutting edge models being developed and deployed across a wide range of applications. Although there do exist some methods to aid in a better understanding of certain kinds of deep learning models (Mohamed, Sirlantzis, and Howells, 2022), most practical deep learning agents remain incredibly difficult to explain even within the research community let alone to the general public.

The unexplainable nature of deep learning models has far reaching implications. For one, it proves very difficult to guarantee constraint satisfaction of a machine learning model. Although some companies apply methods to refine a model's behaviour to an acceptable standard, such as OpenAI employing continual human feedback (Wu et al., 2023), such methods are expensive and far less rigorous than the formal methods used to prove constraint satisfaction. It is therefore difficult to justify the use of such agents in any sort of safety critical environment where constraint satisfaction is very important. This is a vital issue given many of the proposed use cases of AI are safety critical, such as self-driving cars. Unfortunately, deep learning models are already being deployed haphazardly in environments which affect the daily lives of many, such as in therapy chat-bots (Tidy, 2024).

Besides formal constraint satisfaction, it often proves difficult to provide any meaningful explanation of deep learning model behaviour. Unfortunately, the high performance of deep learning models is prone to misleading both researchers and the public alike into believing models are more intelligent than they really are. Goodfellow, Shlens, and Szegedy (2014) presented an excellent example showing how the introduction of almost undetectable noise into an image of a panda caused the then cutting-edge GoogLeNet to misclassify the image as a gibbon with high confidence. Such behaviours are clearly at odds with the most basic properties commonly attributed to intelligence. Unfortunately, it proves difficult to both predict and explain such behaviours.

The tendency for deep learning models to emulate intelligence by learning spurious statistical correlations in their training data, rather than developing a logical understanding of the problem, is so widespread that it has led researchers to humorously analogise models with the case of Clever Hans (Lapuschkin et al., 2019). Unfortunately, deep learning models have a tendency to break down in bizarre and seemingly unexplainable ways when faced with even marginally novel inputs. This brings into question their ability to generalise effectively beyond their training distributions, let alone form logical understandings of their tasks.

Another major problem with deep learning is that it's very resource intensive. State of the art deep learning models require exceptional amounts of data, computing power, and capital to produce. For example, the Stanford University Institute for Human-Centered AI (2024) estimates the training costs for OpenAI's GPT-4 and Google's Gemini Ultra at \$78 million and \$191 million respectively. Training these LLMs typically takes months of computing power from supercomputers composed of large arrays of dedicated training hardware such as GPUs and TPUs. Naturally, powering these supercomputers requires substantial amounts of energy. Although some companies claim to make an effort to reduce the carbon footprint associated with training, training large models has a clear environmental cost. For example, estimates place the training of GPT-3 at more than five-hundred tonnes of CO_2 (Heikkilä, 2022).

Training large models also requires exceptionally large amounts of data. As an example, OpenAI's GPT-3 was trained on forty-five terabytes of data (Halpern, 2023). This wealth of data comes from both publicly available datasets (such as the internet) and from third-party suppliers. Using public data from the internet to train models has proven a controversial practice. The general public have shown noticeable concern about the protection of their private data within AI practices (Ipsos, 2023), and artists have expressed deep concern about copyright infringement within generative AI methods (Appel, Neelbauer, and Schweidel, 2023). Worryingly, models trained on public datasets have also been shown to adopt political and racial biases (Peters, 2022).

Governments and regulatory bodies are acting to impose stricter laws and regulations over the use of data for training AI (European Parliament, 2024). It's likely this will pressure AI companies to find alternative methods of training which are less data intensive. This may prove very difficult given how reliant the performance of current deep learning models is on the amount of training data. Clearly, efforts need to be focused in finding ways to minimise the data and energy costs associated with training current generation models.

1.1.3 Symbolic AI

Despite its overwhelming domination of the current AI landscape, deep learning is only one school of AI practice. Prior to the boom in machine learning, symbolic AI proved to be the principle domain of AI research and development (Kautz, 2020). As with machine learning, symbolic AI encompasses a wide range of approaches and methods, but one area of particular importance is knowledge-based agents (also referred to as logical agents or reasoning agents). At the heart of a typical knowledge-based agent sits a knowledge base, which contains facts about the domain of interest, and a reasoning engine which computes queries over the knowledge base (Russel and Norvig, 2021). Facts can be added to the knowledge base in response to agent inputs, and queries can be used to facilitate a wide-range of intelligent behaviours. Together, the knowledge base and reasoning engine are based on a formally defined logic, such as first-order logic.

A crucial distinction between knowledge-based agents and contemporary machine learning models, is that knowledge-based agents are explainable. Since knowledge-based agents implement some form of formal logic to perform reasoning, the mathematical properties of the logic can be used to explain, justify, and prove properties of the agent. In a typical knowledge-based agent, entire proof trees can be given to justify the response of a query (Russel and Norvig, 2021).

Consider the difference in determining how GPT-4 and a knowledge-based agent arrived at their outputs. In the former, one can only confidently speak about a long and unintuitive chain of tensor operations parameterised by billions of numbers chosen through a complex optimisation process. In the latter, one can observe an entire semantically correct proof provided in a well-studied formal logic. Clearly, it is significantly easier to answer the question 'why did the agent do this?' for the knowledge-based agent.

Despite the many ways logical reasoning agents prove themselves preferable to machine learning agents, the focus of AI shifted from the latter to the former with good reason. The many attempts at implementing practical applications of logical reasoning systems over the decades have highlighted significant issues with the approach. The failure of the knowledge-based expert systems, for example, is cited as one of the leading causes of the second AI winter (Kautz, 2020).

Firstly, while some types of knowledge lend themselves to formal description (such as mathematics), a great deal of human knowledge proves difficult to formalise. Many concepts which humans reason with are fuzzy, incomplete, and based upon intuition and past experience. Attempts to formalise these concepts result in varying degrees of abstraction which may fail to properly capture the concept and may additionally encode the biases of the knowledge base engineers (Dingli and Farrugia, 2023). Consider, for example, the difficulty in formally defining even a simple object such as a chair. Chairs come in all different forms and even the seemingly defining property of a chair, that it can be sat upon, does not exclude it from the wide range of non-chair objects which can also be sat upon. Therefore, in practice it can prove very difficult and time-consuming to develop a comprehensive and consistent knowledge-base about a domain of interest.

In contrast, machine learning methods have proven very effective in tasks which evade formal description, such as image classification. For nearly a decade, the top performing agents in the popular image classification task ImageNet have all been deep learning models (Papers With Code, 2024).

Secondly, it is inevitable that knowledge about a domain evolves with time. This is truer now than ever with the current breakneck pace of research and advancement across myriad disciplines. Therefore, the knowledge within knowledge-based systems becomes outdated. Formally, this idea is referred to as 'concept drift' (Dingli and Farrugia, 2023). Given the monotonic nature of many knowledge-based systems, a great deal of work is required to remove incorrect knowledge and bring the whole system up to date. This significantly limits the scalability and adaptability of knowledge-based systems. It's worth noting that concept drift does affect machine learning agents, but these agents often prove more adjustable to the shift in knowledge due to their ability to learn from new data.

1.1.4 Neuro-symbolic AI

Despite a drastic shift in the focus of mainstream AI from symbolism to deep learning, the symbolic school of thought is far from redundant. It is clear from the previous discussion that symbolic agents possess desirable properties that machine learning agents lack, making them preferable to machine learning agents in certain tasks. However, rather than seeing symbolism and machine-learning as two competing fields, it appears it may be useful to consider ways in which the two fields can be combined to build hybrid agents which exploit the advantages of each.

In theory, a neuro-symbolic agent could benefit from the versatile learning capabilities of machine learning agents to model a wide range of phenomena but also exhibit the explainable behaviour of symbolic agents. Additionally, such an agent could use the information inside a knowledge base to reduce the amount of data required for training and improve generalisation beyond the training dataset. The learning capabilities of the agent could allow its knowledge to remain current, even as information about the domain of interest changes. This project aims to explore the possibility of such systems.

1.2 Project Objectives

1.2.1 Overview of Aims

This project aims to design and and develop effective methods for combining deep learning with knowledge-based reasoning to produce neuro-symbolic agents. These methods will be developed by adapting and extending approaches found during research. The result of method development will be their implementation in a library. The effectiveness of these methods will then be demonstrated with example systems.

This project hopes to serve several purposes. Firstly, developing effective neuro-symbolic methods hopes to demonstrate the validity and benefits of adopting a neuro-symbolic approach to AI. Secondly, the production of methods and tools hopes to make future development of neuro-symbolic agents more accessible and effective. Thirdly, the research, development, and evaluation of neuro-symbolic AI methods hopes to further the student's personal knowledge of AI, skills in using the associated tools, and ability to carry out long form AI research projects.

1.2.2 Objectives

The objectives of this project can be divided into two distinct categories, research and development, the objectives for which are as follows.

Research

1. The discovery and evaluation of existing neuro-symbolic AI systems and methods.
2. The identification of existing systems and methods which are appropriate to adapt and extend to meet the aims of this project.
3. Understanding the theory and mathematics behind deep learning, symbolic reasoning, and neuro-symbolic agents to ensure proposed methods are logically sound.
4. The identification of suitable tasks and benchmarks upon which to evaluate neuro-symbolic agents produced by the proposed methods.

Development

1. The securing and prepossessing of data necessary to evaluate models on the selected tasks and benchmarks.
2. The iterative development of effective neuro-symbolic methods which can be used to produce effective neuro-symbolic agents. These methods will be based upon those found during research, but will be extended and modified to make them novel.
3. The iterative development of tools (such as libraries) which efficiently implement the proposed methods.

4. The development of several example systems which demonstrate the effectiveness of the proposed methods.
5. The evaluation of developed neuro-symbolic AI systems and tools to demonstrate usefulness.
6. The completion of a report which describes, explains, and justifies the work carried out during the course of the project.

1.2.3 Outcomes

By the end of the project, the following deliverables will have been produced.

1. A set of methods for producing effective neuro-symbolic agents.
2. A library which implements these methods.
3. A set of neuro-symbolic agents which demonstrate the effectiveness of these methods.
4. A report detailing the research, development, and evaluation conducted during the course of the project.

Chapter 2

Literature Review

This chapter provides a discussion of the current approaches to neuro-symbolic AI. In addition, arguments are made to justify the direction of this project in light of the existing research.

2.1 The Neuro-Symbolic Landscape

Neuro-symbolic AI is a relatively new field of study. The question of how to best integrate symbolism and machine learning remains open, and research presents a wide array of contrasting arguments about which approaches appear most promising. Due to the newness of the field, the amount of research is relatively modest compared to other areas of AI. Additionally most research focuses on presenting experimental solutions to toy problems rather than providing theoretical foundations. Given the early and experimental nature of the field, many approaches appear valid and worthy of further study.

In his 2020 Engelmore Memorial Award lecture, Kautz (2020) first proposed a taxonomy for classifying the existing approaches to neuro-symbolism. During research, this taxonomy proved useful for conceptualising the landscape of neuro-symbolism and is therefore employed to discuss the current state of the field. Kautz classifies neuro-symbolic agents into one of six categories, which are as follows:

- Type 1: Symbolic Neuro Symbolic - Symbolic inputs are mapped to tensor-valued embeddings. These embeddings are then passed through a neural network whose output is mapped back to a symbolic representation. Although considered neuro-symbolic in the taxonomy, this arrangement is common practice in many deep learning applications where the inputs and outputs are expected to be symbolic data, such as in natural language processing (DeepLearningAI, 2023).

- Type 2: Symbolic[Neuro] - A primarily symbolic problem solver contains a neural pattern-identification subroutine. This routine is employed to improve the performance of the overall symbolic system by using it to solve sub-problems better suited to neural systems.

- Type 3: Neuro ; Symbolic - Inputs are processed by neural networks and mapped to symbolic outputs. The symbolic outputs are then processed by a symbolic reasoner. The symbolic reasoner is designed to be differentiable to allow the neural components to be trained via back-propagation.

- Type 4: Neuro: Symbolic $\rightarrow$ Neuro - A set of example applications of symbolic rules are used to train a neural network such that the network 'compiles away' or models the rules. The network can then be used alone to solve tasks typically reserved for symbolic systems.
- Type 5: Neuro$_{\text{symbolic}}$ - Symbolic rules are used as templates to produce structures within a neural network.
- Type 6: Neuro[Symbolic] - A primarily neural system contains a symbolic reasoning entity which is used to solve tasks better suited to symbolic agents.

2.1.1 Type 1: Symbolic Neuro Symbolic

Type 1 neuro-symbolic systems encompass many methods which are considered standard practice within current deep learning. Type 1 systems are primarily neural systems which involve components to translate network inputs and outputs to and from symbols. Kautz (2020) provides the simple example of an agent which performs sentiment analysis on a document. The system receives a document as input, maps the contents of the document to tensors (using a system such as word2vec (Mikolov et al., 2013)), processes these tensors using a neural network to perform the task of sentiment analysis, and then maps the network's tensor output back into a symbolic form (such as natural language).

The symbolic component of these systems is mostly used for basic pre-and-post processing of inputs and outputs by mapping between symbols and tensors. As a result, the degree of integration between neural and symbolic components is rather limited and the possible benefits of proper neuro-symbolic integration aren't observed.

Given that this category of systems mostly encompasses methods already considered standard deep learning practice, they aren't given much discussion within this project so that resources could be spent exploring more novel neuro-symbolic architectures.

2.1.2 Type 2: Symbolic[Neuro]

Type 2 systems are primarily symbolic systems which contain neural components to handle tasks which lend themselves well to machine learning approaches. Kautz (2020) provides the example of current generation self-driving cars. These systems are primarily symbolic reasoning agents, but they contain neural components to handle tasks such as vision, sensing, and state estimation.

Deep learning methods have been shown to far exceed the performance of purely symbolic methods in perceptual tasks such as computer vision (Voulodimos et al., 2018). It therefore proves beneficial to use these neural components within primarily symbolic systems to handle such tasks. Keeping the overall design of the system symbolic allows for the many benefits offered by the approach, such as explainability and transparency, as explained in section 1.1.3.

An interesting example of a type 2 system is DeepMind's AlphaGo (Silver et al., 2016). Go is a problem that has proven challenging to solve by symbolic means due to its exceptionally large search space. The size of the search space makes producing value estimations for game states particularly challenging. AlphaGo applies neural methods to the problem of state estimation to address the problem of finding a valuation function for Go.

The overall structure of AlphaGo is symbolic, using a Monte Carlo tree search method in order to play the game of Go. However, instead of using standard symbolic play-out and selection policies to evaluate states, AlphaGo uses neural networks. Through training a series neural networks using supervision and reinforcement, DeepMind developed a neural network capable of estimating the value of any given Go state. The result was a Go playing agent which was significantly more successful than purely symbolic Monte Carlo methods. Additionally, evaluating neural networks to generate policy and value predictions proved to be more time-efficient than comparable symbolic Monte Carlo tree searches.

AlphaGo serves as an excellent example of using neural components within a symbolic system to estimate solutions to hard symbolic problems. The overall symbolic design of the system takes advantage of the accuracy of existing game-theoretic approaches to game playing (i.e. Monte Carlo tree search), while the inclusion of neural components addresses the complexities of navigating a large search space by providing an effective heuristic. Additionally, the overall symbolic design of the system makes it somewhat explainable, as it is easy to trace the decisions made by the algorithm through its search tree.

2.1.3 Type 3: Neuro ; Symbolic

Type 3 systems attempt to cascade the output of a neural network into a symbolic reasoner. The reasoner is designed in such a way to allow back-propagation so that the neural components can be trained.

An example of such a system is the neuro-symbolic concept learner (Mao et al., 2019). The neuro-symbolic concept learner is able to answer questions about objects in a scene. It consists of a neural-perception module mapping images to latent representations, a semantic parser mapping natural language questions to executable programs, and a symbolic reasoner which evaluates the program on the latent representation of the scene. Interestingly, training is performed end to end using only images and question answer pairs. The authors describe the training process as curriculum learning, which begins by training the system on simple scenes and with object-based questions, and ends with complex scenes and with questions about object relations and scene composition. Visual and language concepts are disentangled allowing generalisation beyond the training set. The system reached a then state-of-the-art performance on the CLEVR dataset (Johnson et al., 2017). Experiments showed that performance was far better with smaller datasets than other models.

The neuro-symbolic concept learner directly demonstrates some of the potential benefits of neuro-symbolism. The inclusion of a symbolic reasoner increases the explainability of the decisions made by the agent by allowing one to follow its reasoning. Moreover, the system demonstrates the usefulness of neuro-symbolic approaches in reducing the amount of data required to train an agent. One particularly interesting property is the disentanglement of learned concepts. The structure of the agent forces concepts such as shape and colour to be learned individually, increasing both the modularity and interpretability of its neural components. The authors demonstrate how this enabled the system to generalise more effectively by demonstrating its performance on different problems and datasets.

A possible weakness of type 3 systems is their rigid structure. Neural computation is done first and then fed into a symbolic component. When applied to complicated problems, it may prove useful to take a more flexible approach where both neural and symbolic components are interwoven. For example, this would allow symbolic computation to produce logical scene representations from earlier neural perceptions in order to aid later neural computation.

2.1.4 Type 4: Neuro: Symbolic $\rightarrow$ Neuro

Type 4 systems aim to use symbolic rules themselves as training data for a neural network so that the symbolic knowledge is modeled network. The result is a system which can approximate symbolic computation.

Deep Learning for Symbolic Mathematics (Lample and Charton, 2019) explores the idea of teaching neural networks how to perform non-trivial mathematical operations. The authors train transformer seq2seq models to perform the tasks of symbolic integration and solving differential equations. Mathematical expressions are treated as trees, which are processed by the transformer, and the output is decided by beam search. Using this method, the authors produce models whose performance significantly exceeds those of existing symbolic mathematics applications, such as Mathematica and Matlab. Additionally, the transformer models were shown to generalise beyond expressions producible by the algorithm which created the training datasets.

Deep learning for symbolic mathematics is particularly interesting given that mathematical problems naturally lend themselves very well to symbolic formulation, and it doesn't appear obvious that switching to deep learning approaches would offer many advantages. Remarkably, deep learning for symbolic mathematics demonstrates that transformers are surprisingly competent at difficult mathematics. They necessarily require a large training set but this isn't such a drawback given both the difficulty in specifying a complete symbolic solution, and the data being inherently exempt from the usual problems of privacy, collection, copyright, etc.

For all their promise, there is an unavoidable downside to type 4 systems. Namely, they have none of the explainability nor provability of equivalent symbolic approaches. The system in Deep learning for Symbolic Mathematics may be effective, even more so than symbolic approaches, but it is a black box with absolutely no guarantees of correctness. It cannot demonstrate why a solution is correct, or even present in an understandable manner how it arrived at a solution.

Given the increase in agent explainability being a major theme in neurosymbolism, it seems difficult to argue in favour of systems which directly reduce explainability for performance gains.

2.1.5 Type 5: Neuro$_{\text{symbolic}}$

Type 5 systems use symbolic knowledge in order to produce structures within neural networks. The result is a primarily neural agent with an architecture which is considerably more interpretable having been built from a logical specification.

Logic Tensor Networks (Serafini and Garcez, 2016) explore the idea of producing neural networks which are specified in a differentiable first-order logic called 'Real Logic'. By specifying a system in Real Logic, the neural components are trained to maximise the satisfaction of the specification through gradient descent. Logic Tensor Networks also support reasoning, querying, and a limited form of inference. A similar idea is explored in Differentiable Fuzzy Logic (Krieken, Acar, and Harmelen, 2021) which uses a different language than Logic tensor networks.

At first glance, type 5 systems may seem similar to type 4 systems, but there are significant differences. In a type 4 system, networks are trained under supervision to model symbolic rules. A data set of input-output examples of the rule is given, and training proceeds as in standard deep learning. The desired result is a network which directly implements the rule. In a type 5 system, the network is trained to satisfy (rather than model) a set of symbolic rules. The rules themselves aren't necessarily captured by the network, but instead become constraints which the network aims to satisfy.

Type 5 systems seem particularly interesting because they offer a generic yet tight integration of both neural and symbolic components. They allow for the training of much more explainable networks because the very structure of the system is defined in logic. Logic tensor networks demonstrate that this can decrease the amount of training data necessary for a system to learn effectively. Their focus on producing networks which satisfy knowledge, rather than directly model it, allow for existing symbolic knowledge to be easily injected into the training process.

2.1.6 Type 6: Neuro[Symbolic]

Type 6 systems are primarily neural engines which contain a symbolic reasoner. Kautz (2020) describes the interplay of the two systems as similar to the relationship between the two types of thinking popularised by Thinking, Fast and Slow (Kahneman, 2015). In Kahneman's model of the mind, there exist two distinct modes of reasoning. Type one reasoning is fast, approximate, and reactive reasoning, whereas type two is slow, exact, and deliberate. In a Neuro[Symbolic] system, a neural engine behaves much like a type one reasoner, which can call upon a symbolic subsystem to behave like a type two style reasoner. An internal model of the system's state of attention is used to determine when control is passed from the neural engine to the symbolic subsystem.

Despite the interesting idea, little published research has been done in this area. From comments in his Engelmore Memorial lecture, it seems Kautz (2020) proposed this category in preparation for his and his colleagues' yet to be published research into these systems. It is therefore difficult to discuss this category of systems.

One can say that the idea of using neural components to perform Kahneman's type 1 reasoning and using symbolic components to perform type 2 reasoning is a theme found across many of the other neuro-symbolic systems discussed in this report, if only implicitly. Neural components have been used to perform tasks such as perception and state estimation while symbolic components have been used to reason over neural outputs (section 2.1.3) and determine the overall logic of the agent (section 2.1.2). However, the idea of passing control between neural and symbolic systems using an internal model of system attention is vague and difficult to speculate upon.

2.2 Summary

Available research reveals a range of different approaches to neuro-symbolic agents. Kautz' taxonomy (2020) provides a helpful lens through which to group and analyse existing systems.

Type 1 and Type 4 systems are predominantly neural. Type 1 systems encompass a lot of which can be considered standard deep learning practice and will therefore not be given much attention in this project. Type 4 systems aim to use neural networks to implement heavily symbolic operations, such as mathematics, by training networks on input-output examples of symbolic rule applications. Although successful type 4 systems have been developed, their predominantly neural nature makes them black boxes which limits their explainability and transparency.

Type 2 systems are predominantly symbolic, typically consisting of an overall symbolic program which calls upon neural networks to perform tasks where symbolic methods fall short, such as perception. Their focus on symbolism allows them to take advantage of existing symbolic knowledge about a problem. Additionally, their results are inherently more explainable than fully neural systems. Their primarily symbolic nature comes at the cost of requiring networks to be trained before insertion into the system which both limits the agent's ability to learn as a unit and does not reduce the time, energy, and data costs associated with training the neural components.

Type 6 systems propose an interesting balance between their neural and symbolic components. They consist of a neural engine with an embedded symbolic reasoner. Processing is primarily neural with the reasoner invoked when assistance is needed. The idea is heavily inspired by Kahneman's (2015) model of human reasoning. Unfortunately, type 6 systems remain largely hypothetical with no clear examples found during research.

Type 3 and type 5 systems strike a relatively even balance between their neural and symbolic components. Type 3 systems use networks to interface with and perceive their environment, and this information is forwarded into a symbolic reasoner. The reasoner is designed in such a way to enable neural training through back-propagation. Type 3 systems are reasonably explainable and have been demonstrated to generalise effectively on small datasets. However, the rigid structure of such agents may limit their ability to scale to more complex problems which require interweaving both symbolic and neural computation.

Type 5 systems directly use symbolic rules to produce structures within networks. A specification of neural behaviour is given in logic and the network is trained to maximise the satisfaction of the specification. The result is more explainable networks which are able to make direct use of already existing symbolic knowledge about a problem, and thus reduce the amount of data required for training.

The field of neuro-symbolic AI is young and there are seemingly many viable routes by which to produce effective neuro-symbolic agents. Based on the research carried out in this project, type 5 systems seem to offer a particularly promising approach to combining knowledge-based methods with neural networks. Such systems demonstrate the ability to integrate both symbolic knowledge and neural learning in a complimentary, data efficient, and explainable manner. Additionally, their flexible structure may allow them to scale more effectively to complex problems than other approaches.

Chapter 3

Methodology

This chapter discusses the methods and tools which were employed during the development phase of this project.

3.1 Software Methodology

Developing deep learning models is inherently an iterative and experimental affair. This is especially true when exploring new methods and architecture choices, such as when integrating symbolic elements. Therefore, an Agile software methodology was followed throughout the development phase of this project. Prototypes of neuro-symbolic methods, tools, and models were created and, informed by their usability and performance on various tasks, were refined through an iterative process.

In order to properly track progress, Kanban boards were used to divide the work into sprints. Early sprints focused on designing and developing neuro-symbolic methods and their implementation as a library, while later sprints focused on utilising this library to develop example models. Models were built to explore the effectiveness of the neuro-symbolic framework at solving problems. Evaluation of the model's performance and usability informed changes to the framework. Manual testing was performed throughout to verify the code base.

This method was employed with the hope that it would satisfy the project objectives of the development of effective neuro-symbolic methods and example systems. The outcome is discussed in chapter 8.

3.2 Software Tools

Research carried out during this project has shown that there are very few, if any, high level tools for developing neuro-symbolic systems. This is a natural result of the newness of the field and the lack of standardised neuro-symbolic approaches. Therefore, many of the tools used to build neuro-symbolic methods and agents were developed as part of this project.

Development used Python 3 as the primary programming language. In order to support efficient development of the deep learning components of the neuro-symbolic systems, Tensorflow 2 and Keras 3 were widely used. Other Python 3 libraries such as Numpy and Matplotlib were also used to facilitate development.

Development took place in Visual Studio Code and Jupyter Notebooks were employed when training and evaluating neuro-symbolic models. Git and GitHub were used for version control. Overall progress was tracked using Jira. This section further describes and provides justification for these tools.

Python 3

Due to the availability of many sophisticated and free machine learning libraries, Python 3 is frequently used as the primary programming language for deep learning projects. Besides the numerous libraries on offer, Python 3 itself supports a wealth of easy to use high level data structures and operations which make Python particularly suitable for developing symbolic AI systems. When beginning this project, the student had both general and machine learning experience in Python 3, making it a helpful tool for reducing development time associated with learning new languages and libraries.

Tensorflow 2 and Keras 3

One of the aforementioned deep learning libraries which sees extensive use in Python led deep learning development is Google's Tensorflow 2. Tensorflow 2 offers low-level tensor-based data structures and operations which are able to utilise GPU and TPU hardware to maximise efficiency. Additionally, Tensorflow 2 offers tools to perform efficient automatic symbolic differentiation on tensors, which is frequently used to quickly train deep learning models through back-propagation. Tensorflow 2 also supports deployment on a wide range of devices, ranging from distributed super computers to smartphones, which massively increases the usability of systems built on top of it.

Keras 3 is a high-level deep learning library built on top of Tensorflow 2. It offers high-level data structures and algorithms which allow for rapid development of deep learning models without the developer needing to concern themselves with many of the complexities involved in writing models purely in Tensorflow 2. Simple use cases are quick and easy to develop, but complex and novel use cases are equally accessible.

When beginning this project, the student had experience developing basic classification and regression deep learning models in Keras. Although he had limited experience in using Tensorflow 2, he wanted to take the opportunity to develop his skills in using it. He felt doing so would better prepare him for further AI projects and work in the AI industry.

There are alternative libraries to Tensorflow and Keras which see frequent use in developing deep learning models, such as PyTorch. However, after spending some time reviewing these alternatives, it was decided that they did not offer substantial enough benefits over Tensorflow and Keras to justify the additional time required to learn them. Given the above reasons, Tensorflow 2 and Keras 3 were used in this project to develop the neuro side of the proposed neuro-symbolic methods.

Other Python 3 Libraries

Besides Tensorflow 2 and Keras 3, other Python libraries were used to facilitate the development of the project. These are as follows,

- NumPy - NumPy allows for the efficient handling of multi-dimensional arrays, such as tensors. It is common for machine learning data to be imported and pre-processed as NumPy arrays, making it a suitable choice for this project. Additionally, the student had limited experience with NumPy and wanted to use this project as an opportunity to advance his NumPy skills.
- Matplotlib - Matplotlib facilitates straightforward creation of graphs and data visualisations. Since visualisations play an important role throughout the process of developing deep learning models, Matplotlib is a useful tool for this project. Additionally, much like NumPy, the student had limited experience with Matplotlib and wanted to use this project as an opportunity to advance his knowledge.

Jupyter Notebooks

Jupyter notebooks offer a development environment allowing one to easily write code and provide supporting markup text. Notebooks support Python 3 development enabling one to produce readable Python projects. Jupyter notebooks have seen extensive use throughout machine learning practice for this reason. Additionally, the student had extensive experience using Jupyter Notebooks and they stood as his preferred environment for deep learning development. Jupyter Notebooks were therefore used to train and test neuro-symbolic models during this project.

Visual Studio Code

Microsoft's Visual Studio Code is a popular light-weight IDE. It provides a wide range of useful development and debugging features and offers support for all of the other software tools discussed in this section. Visual Studio Code was also the student's preferred IDE. Therefore, Visual Studio Code was used as the primary IDE throughout this project.

Git and GitHub

Git is a widely used version control system allowing users to efficiently track and manage the development of their software projects. GitHub is a free web platform for hosting remote Git repositories. Both of these tools proved incredibly useful in managing the development stage of this project. Much like the other tools which were used, the student wanted to develop his existing knowledge of both Git and Github given their industry wide use.

Jira

Jira is a web-based project management tool which allows for effective project planning and tracking. Jira offers a variety of Agile management tools, such as Kanban boards, which were used in this project to ensure steady progress was made over the weeks.

3.3 Hardware Tools

Training neural networks is computationally expensive and therefore it is common to employ specialised hardware to speed up development time. In most cases, training is performed on GPUs and TPUs as they prove more efficient than CPUs at performing the necessary computations. The student had access to a PC equipped with an Intel i5 processor and an NVIDEA 1060TI GPU. Although underpowered compared to cutting-edge deep learning hardware, the PC had proven effective at training small neural networks in the past. It was therefore employed to develop basic neuro-symbolic models in this project.

Chapter 4

Design and Method Overview

The aim of this project is to develop and demonstrate effective neuro-symbolic methods which combine deep learning and knowledge-based reasoning. Research revealed a wide variety of approaches to this problem, however type 5 systems proved to be of particular interest. This section details the design of a class of type 5 neuro-symbolic systems, called Fuzzy Tensor Logic systems. A formal definition of Fuzzy Tensor Logic is given in chapter 5. Implementation details of Fuzzy Tensor Logic methods are given in chapter 6. The design and implementation of example systems is given in chapter 7.

4.1 Fuzzy Tensor Logic Systems

Fuzzy Tensor Logic (FTL) systems are a class of type 5 neuro-symbolic agent. These systems support both learning from data and logical reasoning. At the heart of an FTL system is a knowledge base, which contains a set of formulas expressing all symbolic knowledge known to the agent. These formulas are written in a purpose built logic called Fuzzy Tensor Logic. An agent additionally contains an interpretation, which is a learnable mapping used to evaluate the truthfulness of the formulas in its knowledge base. The interpretation can contain neural networks to implement the behaviour of the functors and predicates reasoned over in the knowledge base. An agent learns by searching for an interpretation which maximises the truthfulness of its knowledge base.

An FTL system is able to use its symbolic knowledge to constrain and direct what its neural components learn. In this way, FTL systems can be used to train neural networks more efficiently, by directly injecting symbolic knowledge into the learning process and ensuring its outputs satisfy logical constraints. The system is also inherently modular, allowing for the extraction of its neural components for use in other applications. Alternatively, one can insert existing neural networks into an FTL system to fine tune their behaviour to meet logical constraints.

Furthermore, an FTL system is able to use its neural knowledge to assist in reasoning. The agent's interpretation can contain neural networks to implement the components which appear in its knowledge base. Neural networks can therefore be used to perform tasks which are difficult for purely symbolic agents, such as image classification and natural language processing, within the context of logical reasoning.

Fuzzy Tensor Logic is, as its name implies, fuzzy, which enables the agent to reason using the notion of partial truth. This allows for efficient reasoning over a wide range of tasks which prove difficult to formally define in traditional binary logic.

Furthermore, FTL systems allow users to select the operators used to implement its fuzzy semantics. There exist a wide range of proposed fuzzy operators, each of which define very different semantics for evaluating formulas. Some operators are better suited for learning and reasoning across different problem domains. Users are free to select from a predefined set of existing fuzzy operators or define their own.

FTL systems demonstrate a tight integration of both their symbolic and neural components allowing for explainable learning and efficient reasoning over a wide variety of domains. To summarise, a Fuzzy Tensor Logic system consists of the following components:

- A knowledge base - A set of formulas written in FTL which express the symbolic knowledge known to the agent.
- An interpretation - A learnable mapping which assigns the predicates, functors, constants, and variables which appear in the knowledge base to actual objects. These are either tensors (in the case of constants and variables) or operations over tensors (in the case of predicates and functors). Importantly, predicates and functors can be mapped to neural networks.
- A fuzzy operator set - A set of differentiable fuzzy logic operators which define the semantics of the junctors and quantifiers which appear in the knowledge base.

Additionally, Fuzzy Tensor Logic systems allow for the following operations:

- Learning - The neural components of FTL systems can be trained in order to maximise the satisfaction of its knowledge base.
- Querying - FTL systems can answer logical queries using information in its knowledge base and interpretation.

The remainder of this chapter presents the design of these aspects of FTL systems in more detail.

4.2 Fuzzy Tensor Logic

The knowledge contained in FTL systems is written in a purpose built language called Fuzzy Tensor Logic. Fuzzy Tensor Logic is a many-valued fuzzy first-order logic which supports the differentiation of predicates and functors with respect to the truth values of formulas. A formal definition of Fuzzy Tensor Logic is given in chapter 5. This section provides an overview of the features of Fuzzy Tensor Logic necessary to understand the design of FTL systems. In essence, the distinguishing features of Fuzzy Tensor Logic are the following:

- First-order logic over tensors - Fuzzy Tensor Logic is a first-order logic which limits the universe to real-valued tensors. This allows one to naturally implement predicates and functors as functions on tensors, such as neural networks.
- Fuzzy semantics - Fuzzy Tensor Logic is a fuzzy logic. The truthfulness of an expression is any real number in the interval $[0, 1]$. This allows for the notion of partial truth. Logical junctors take on fuzzy semantics to allow fuzzy truth values to propagate through the system.
- Differentiable semantics - The truthfulness of a Fuzzy Tensor Logic formula is differentiable with respect to the functors and predicates it contains. This allows one to train neural functors and predicates via backpropagation.

4.2.1 First Order Logic Over Tensors

First order logic is a well studied branch of logic which sees wide application across knowledge-based reasoning agents (Russel and Norvig, 2021). Simply put, formulas in first order logic express knowledge about relationships between objects within a universe.

As a very brief reminder, first order logic starts with a signature, which is a set of constant, functor, and predicate symbols. Then, one defines a structure over the signature which gives meaning to the signature by mapping its symbols to constants, functors, and predicates within and over a universe of objects. Constants denote individual elements of a universe, functors denote mappings between elements of the universe, and predicates denote relations over the elements of the universe. Equipped with these symbols, one can write formulas to express relational knowledge about the universe. Formulas can be combined using logical junctors (also called connectives) to express conjunction, disjunction, negation, and implication. Additionally, quantifiers can be used to express knowledge of the form "for all x..." and "there exists a y such that..." (Freydenberger, 2020).

Fuzzy Tensor Logic is a first order logic which limits its universe to real-valued tensors. Structures map constants to tensors, functors to tensor-valued functions, and predicates to functions which accept tensors and return a truth value. This allows one to naturally implement predicates and functors as neural networks if they wish. For example, one can implement the predicate isCat as a neural network which returns a truth value indicating whether or not is input is an image of a cat. Fuzzy Tensor Logic therefore provides a rich and expressive language to write knowledge about the behaviour of neural networks.

Although functors and predicates can be neural networks, this doesn't have to be the case. Functors and predicates can take on any internal structure so long as they accept appropriate inputs and produce appropriate outputs. Therefore, functors and predicates can be purely symbolic computations if desired. Neural and symbolic components can sit side-by-side in the system and seamlessly interact with one another.

4.2.2 Many-Valued Fuzzy Semantics

A many-valued logic is a logic which supports more than the standard truth values of true and false (Stanford Encyclopedia of Philosophy, 2015). There is a wealth of proposed many-valued logics. One such example is Kleene's three valued logic which supports truth values of true, false, and unknown. Another example is the infinitely-valued probability logic, whose truth values represent the probability of a statement being true. Fuzzy Tensor Logic is a fuzzy logic, whose truth values are all real numbers in the interval $[0, 1]$ to capture the idea of partial truth (Novák, Perfilieva, and Močkoř, 1999).

For example, a fuzzy logic might evaluate the proposition "All cars have four wheels" to the value 0.9. It is certainly true that the overwhelming majority of cars have four wheels, but there do exist exceptions so the proposition is not strictly true. Another example would be evaluating the implication "if it is raining, it is not sunny". A fuzzy logic might evaluate this statement to a truth value of 0.7. After all, it is very often not sunny when it rains, but there are exceptions and as a result the truthfulness of the implication is weakened.

Because fuzzy logic is many-valued, the semantics of logical junctors (also known as connectives) have to be defined to reflect this. In other words, conjunction, disjunction, implication, negation, and the quantifiers have to be defined to produce many-valued outputs, rather than the standard binary outputs true and false. There are a number of well studied fuzzy junctors available for this task. Furthermore, Fuzzy Tensor Logic limits the semantics to differentiable fuzzy junctors. This is necessary to facilitate training via back-propagation. The user of Fuzzy Tensor Logic is free to decide which operators are used in an agent, and they can be switched during the agent's lifetime.

4.2.3 Differentiable Semantics

The semantics of Fuzzy Tensor Logic ensures that the truth value of formulas are differentiable with respect to the predicates and functors they contain. Since predicates and functors can be neural networks, this enables their training via back-propagation.

Training an FTL system amounts to finding parameters for its predicates and functors which maximise the satisfaction of its knowledge base. This method of training allows one to directly inject both domain knowledge into and enforce constraints upon the neural elements of the system.

4.3 Knowledge Bases

All symbolic knowledge known to an FTL agent is contained within its knowledge base. Simply put, a knowledge base is a collection of formulas written in Fuzzy Tensor Logic. FTL systems store formulas in the form of parse trees, which can be generated by a dedicated parser sub-system. The parse trees can then be evaluated under the agent's interpretation to produce truth values.

4.3.1 Formulas and Parse Trees

The most natural representation of formulas for users is as text. However, it proves computationally inefficient to reason over text when evaluating formulas.

Therefore, formulas are stored internally as parse trees. In order to convert between textual and parse tree representations of formulas, the system provides a parser sub-system. Every FTL formula has a corresponding parse tree.

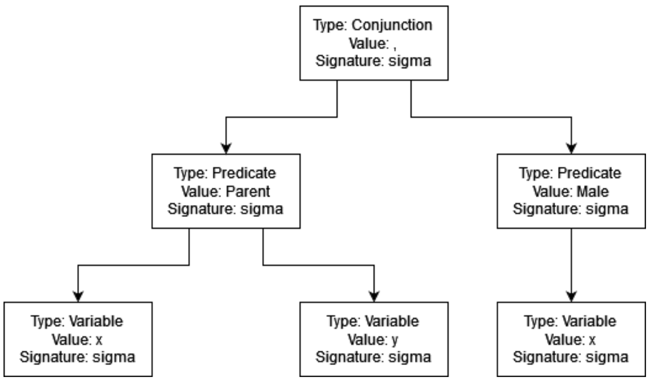


Figure 4.1: A parse tree for the formula $\mathsf{Parent}(x, y) \wedge \mathsf{Male}(x)$. Intuitively, the formula asserts that x is the father of y.

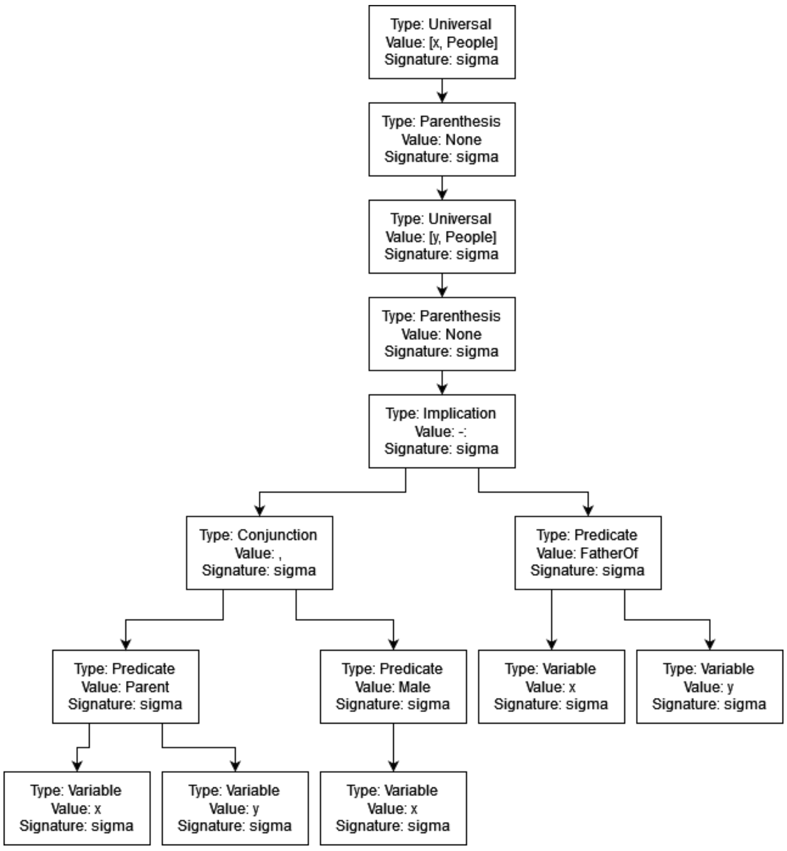


Figure 4.2: A parse tree for the formula which intuitively defines fatherhood. $\forall x \in \mathsf{People} : (\forall y \in \mathsf{People} : \mathsf{Parent}(x, y) \wedge \mathsf{Male}(x) \rightarrow \mathsf{FatherOf}(x, y))$

A parse tree is much like an ordinary tree data structure. It consists of a non-empty set of nodes, where each node can have one or more child nodes. Every parse tree has a root node which is the only node without a parent. Furthermore, each node contains information about the type of syntactic object it represents, as well as values appropriate for that type. Since formulas are written in reference to a signature, the signature is also stored within the nodes of a parse tree. For the sake of brevity, examples will be given in place of a formal definition in figures 4.1 and 4.2. For more information, the complete implementation details of parse trees are given in section 6.4.

Converting from a textual representation to a parse tree formula is handled by a dedicated parsing algorithm. The algorithm recursively builds a parse tree by attempting to match its input string with a type of expression in reverse order of precedence (e.g. the algorithm first attempts to match the expression as an implication, then a disjunction, then a conjunction, and so on until it reaches parentheses). When a match is found, a corresponding parse tree node is made and the algorithm proceeds to parse the sub-expressions to produce child nodes. If a match isn't made, then the expression is an invalid FTL expression, and an error is raised. Constants and variables do not have children and, since every FTL formula is made up of operations over constants and variables (see chapter 5), the recursion is guaranteed to end. Finally, FTL formulas are always written over signatures, therefore the algorithm ensures that all constant, predicate, and functor symbols belong to the signature. Pseudo-code for the algorithm is given in listing 4.1.

```
#Algorithm: Parse - Generates a parse tree for expression.
#Inputs
#   expression - A string representation of an FTL formula.
#   signature  - A signature containing constant, predicate,
#                and functor symbols.
#   precedence - A list of syntactic FTL elements
#                in order of precedence.
#Output
#   A parse tree representation of expression.

function parse(expression, signature, precedence):
    for type in reverse(precedence):
        if matches(expression, type):
            childNodes = []
            for subexp in subexpressions(expression, type):
                childNodes.append(
                    parse(subexp, signature, precedence))
            return new Node(
                signature = signature,
                type = type,
                value = value(expression),
                children = childNodes
            )
    error("Invalid expression")
```

Listing 4.1: The parsing algorithm with converts text representations of FTL formulas to parse trees.

4.4 Interpretations

A knowledge base provides the agent with statements about the objects in the domain of interest, but it does not specify what exactly these objects are. The purpose of an interpretation is to map all of the elements which appear in FTL formulas to the actual objects. An interpretation consists of the following:

- A structure - A structure is defined over a signature. A signature provides an agent with a vocabulary of constant, functor, and predicate symbols. The purpose of a structure is to map the elements to actual objects. To be precise,
  - Constants are mapped to real valued tensors.
  - Functors are mapped to functions which accept real valued tensors and return real valued tensors. These can be neural networks.
  - Predicates are mapped to functions which accept real valued tensors and return a fuzzy truth value in the interval $[0, 1]$. These too can be neural networks.
- A variable assignment - A mapping of variables to real-valued tensors.
- A domain assignment - A mapping of domains to sets of real-valued tensors. Domains appear in quantifications to limit the set of objects over which quantification is performed.

4.4.1 Evaluating Formulas

Equipped with an interpretation, it is possible to evaluate the truth value of formulas which use its elements. Different interpretations will cause formulas to evaluate to different truth values. The higher the truth value, the closer the interpretation behaves according to the knowledge contained within the formula.

The algorithm for evaluating the truth value of a formula (in parse tree form) under an interpretation is given in listing 4.2. The algorithm computes the truth value of a formula recursively. The evaluation of constants and variables simply returns their corresponding tensors. The evaluation of predicates and functors returns the result of applying them to their child values. The evaluation of junctors returns the result of applying their associated fuzzy operation to their child values. Finally, the evaluation of quantifiers returns the result of applying their aggregator to the extended interpretations of their children as limited by the domain.

It is important that the evaluation algorithm is implemented in such a way that ensures it is possible to compute the gradients of truth values with respect to a formula's predicates and functors. The implementation of the evaluation algorithm given in section 6.6.2 ensures this by implementing all necessary operations in TensorFlow 2.

```
#Algorithm: Evaluate - Determines the truth value parse tree
#           under the given interpretation and fuzzy operators.
#Inputs
#   pt             - A parse tree to be evaluated.
#   interpretation - An appropriate interpretation.
#   junctors       - A set of fuzzy operators.
#Output
```

```
#   The truthfulness of the pt under the interpretation.

function evaluate(pt, interpretation, junctors):
    if pt.type == CONSTANT or pt.type == VARIABLE:
        # return intepretation's mapping of constant
        # or variable.
        return interpretation(pt.value)

    elif pt.type == FUNCTOR or pt.type == PREDICATE:
        childValues = [] # empty list.
        for child in pt.children:
            childValues.append(evaluate(
                child, interpretation, junctors))
        # return the result of applying the function to children.
        return interpretation(pt.value)(childValues)

    elif pt.type == BRACKET:
        return evaluate(pt.children[0], interpretation, junctors)

    elif pt.type == NEGATION:
        operator = junctors.negation
        # Return fuzzy negation of truth value of child.
        return operator(evaluate(
            pt.children[0], interpretation, junctors))

    elif pt.type == CONJUNCTION OR  pt.type == DISJUNCTION OR
         pt.type == IMPLICATION:
        if pt.type == CONJUNCTION:
            operator = junctors.tnorm
        elif pt.type == DISJUNCTION:
            operator = junctors.tconorm
        else
            operator = junctors.implication
        # Return fuzzy operator applied to of child truth values.
        return operator(
            evaluate(pt.children[0], interpretation, junctors),
            evaluate(pt.children[1], interpretation, junctors))

    elif pt.type == UNIVERSAL or pt.type == EXISTENTIAL:
        boundVariable = pt.value[0]
        domainSymbol = pt.value[1]

        # Collect the truth values of formula under all ext. interp.
        results = []
        domain = interpretation(domain)
        for i in len(domain):
            variableValue = domain[i]
            # Define a variable substitution and create a resulting
            # extended interpretation.
            substitution = {boundVariable: variableValue}
            extInterpretation = interpretation.extend(substitution)
            results.append(
                evaluate(pt.children[0], extInterpretation, junctors))

        if pt.type == UNIVERSAL:
            aggregator = junctors.universal
        elif pt.type == EXISTENTIAL:
            aggregator = junctors.existential

        #Apply relevant aggregator to results.
        return aggregator(results)

    else:
        error("unsupported node type.")
```

Listing 4.2: The evaluation algorithm which determines the truth value of parse trees under an interpretation.

4.4.2 Evaluating Knowledge Bases

It is necessary to evaluate not just the truth value of a single formula, but the truthfulness of an entire knowledge base. The truthfulness of a knowledge base under an interpretation is referred to as its satisfaction.

The satisfaction of a knowledge base is simply an aggregation of the truth values of its formulas. This aggregation is performed by a given satisfaction aggregator, which is any differentiable function which maps arbitrary length lists of fuzzy truth values to a single fuzzy truth value. The choice of satisfaction aggregator naturally determines the computed satisfaction of the knowledge base. Users are free to choose a satisfaction aggregator which suits their needs.

The algorithm for determining the satisfaction of a knowledge base under an interpretation is given in listing 4.3. Simply put, the algorithm iterates over the formulas in the knowledge base, evaluates them using the algorithm in listing 4.2, and applies the given satisfaction aggregator the result.

Much like the evaluation algorithm, it is important that the knowledge base satisfaction algorithm is implemented in a way to guarantee differentiability of its output with respect to the truth values of the knowledge base formulas. The implementation in section 6.6.2 ensures this by again using differentiable TensorFlow 2 operations.

```
#Algorithm: EvaluateKB - Determines the satisfaction of a
#           knowledge base w.r.t. an interpretation.
#Inputs
#   kb               - A knowledge base.
#   interpretation   - An appropriate interpretation.
#   junctors         - A set of fuzzy operators.
#   sagg             - A satisfaction aggregator.
#Output
#   The satisfaction of kb w.r.t intepretation

function evaluateKB(kb, interpretation, junctors, sagg):
    results = [] # Empty list
    for pt in kb.parseTrees:
        results.append(evaluate(pt, interpretation, junctors))

    return sagg(results)
```

Listing 4.3: The knowledge base evaluation algorithm.

4.5 Fuzzy Operator Sets

In order to evaluate FTL formulas, differentiable fuzzy operators need to be supplied to implement the behaviour of junctors and quantifiers. In particular, operators need to be supplied for the following:

- Negation

- Conjunction
- Disjunction
- Implication
- Existential quantification
- Universal quantification

There exists a wide range of proposed fuzzy operators, some of which are already differentiable or can be easily modified to be differentiable. Different operators result in formulas evaluating to different truth values. Additionally, the gradients produced by operators vary wildly. Therefore, the choice of fuzzy operators is an important one which significantly affects agent behaviour and training. The implementation of FTL comes with several built in operator sets, but also allows the user to implement their own if they wish.

It is beyond the scope of this project to provide extensive details of fuzzy logic junctors, but a brief overview of a popular approach will be given. The following ideas and definitions are summarised from Krieken, Acar, and Harmelen (2021). One common way to define fuzzy junctors, which is used by FTL, is to start with t-norms. T-norms are binary functions which map inputs in the interval $[0, 1]$ to outputs in the interval $[0, 1]$. T-norms also satisfy other mathematical properties which will not be discussed for the sake of brevity. In practice, t-norms capture many of the behaviours one desires in fuzzy conjunction. Examples include the Godel t-norm $T_{godel}(a, b) = \mathsf{min}(a, b)$ and the product t-norm $T_{product}(a, b) = ab$.

Given a t-norm and a fuzzy definition for negation, one can derive a corresponding t-conorm (also known as an s-norm) which suitably models fuzzy disjunction. From fuzzy conjunction, disjunction, and negation, one can define implication using the standard logical identity that $a \to b = \neg a \vee b$. There also exist various definitions for existential and universal quantifiers which perform aggregation over lists of truth values.

4.5.1 Symmetric Operator Sets

In FTL, a symmetric operator set is a complete set of fuzzy operators derived from a t-norm and the standard fuzzy negation. The standard fuzzy negation N is defined as $N(x) = 1 - x$. N is an intuitive, simple, and differentiable operation which provides the properties one typically expects from negation. A symmetric operator set for a given t-norm T has the following properties:

- Negation is the standard fuzzy negation N.
- Conjunction is the t-norm T.
- Disjunction is the t-conorm (or s-norm) S, where $S(a, b) = 1 - T(1 - a, 1 - b)$. This derivation is based on De Morgan's Law $a \wedge b = \neg(\neg a \vee \neg b)$.
- Implication is the S-implication I_S on the t-norm T, or the R-implication I_R on the s-norm S. S-implication is defined such that $I_S(a, b) = S(N(a), b)$, which satisfies the logical identity $a \to b = \neg a \vee b$. R-implication is defined such that $I_R(a, b) = \mathsf{sup}\{c \in [0, 1] | T(a, c) \leq b\}$ where sup is the supremum.

- Existential quantification is the extended t-norm (i.e. a variant of T which accepts an arbitrary number of inputs).
- Universal quantification is the extended s-norm (i.e. a variant of S which accepts an arbitrary number of inputs)

When choosing fuzzy operators for an FTL system, it is helpful to use a symmetric operator set. This is because the operators interact in predictable ways which satisfy many of the long established logical equivalencies. FTL provides several standard symmetric operator sets to the user, including

- The Godel set, which is based on the Godel T-norm $T_g(a, b) = \mathsf{min}(a, b)$.
- The product set, which is based on the product T-norm $T_p(a, b) = ab$.
- The Łukasiewicz set, based on the Łukasiewicz T-norm $T_l(a, b) = \mathsf{max}(0, a + b - 1)$.
- The Drastic set, based on the Drastic T-norm T_d,

$$T_d(a, b) = \begin{cases} b & \text{if } a = 1, \\ a & \text{if } b = 1, \\ 0 & \text{otherwise.} \end{cases}$$

- The Nilpotent Set, based on the Nilpotent minmum T-norm T_n,

$$T_n(a, b) = \begin{cases} \mathsf{min}(a, b) & \text{if } a + b > 1, \\ 0 & \text{otherwise.} \end{cases}$$

Both S-implications and R-implications are available for each set.

Although symmetric sets produce predicable semantics which satisfy many long established logical identities, it is not always practical to choose a symmetric set. This is because some of its operators have undesirable properties which effect training performance. Therefore, users are free to mix and match operators from different sets, or even produce their own operators, to improve training performance. Users can then switch back to a symmetric set during querying to improve querying performance.

4.5.2 Vanishing and Single Passing Operators

In Analysing Fuzzy Differentiable Operators (Krieken, Acar, and Harmelen, 2021), the authors investigate the suitability of fuzzy operators when performing back-propagation. Their results show that their are two properties which one must be mindful of when selecting operators for training: operator vanishing and operator single passing.

Intuitively, an operator is vanishing if there is an interval in its domain over which the operator outputs zero. Over this interval, a vanishing operator loses its ability to effectively propagate gradients and therefore disrupts the learning process. It is therefore important to avoid vanishing operators during agent training.

An operator is single passing if it has non-zero derivatives on at most one input argument. This too disrupts the learning process as gradients can only be propagated through at most one input at a time. Single passing operators are particularly ill-suited to quantification, since such operators are expected to accept an arbitrarily large number of inputs. Training using single passing operators is less disruptive than vanishing ones, however they result in significantly less efficient training.

Unfortunately, many operators are in-fact vanishing on at least part of their domain. Serafini and Garcez (2016), attempt to reduce the effects of vanishing operators by wrapping them in projections. Projections ever so slightly adjust the output of operators to move them away from troublesome values. FTL systems adopt a form of optional projections on operators.

4.5.3 Standard Product Set

During development, different operator sets were investigated. As a result, FTL recommends a specific non-symmetric fuzzy operator set when training called the Standard Product Set. A similar set of operations has been shown to work well during back-propagation in (Serafini and Garcez, 2016). The Standard Product Set is primarily built around use of the product t-norm, which is vanishing on only a very small part of its domain. Additionally, the t-norm is not single passing. The elements of this set are as follows:

- Negation - Standard fuzzy negation $N(a) = 1 - a$.
- Conjunction - Product t-norm $T(a, b) = ab$.
- Disjunction - The product s-norm $S(a, b) = a + b - ab$.
- Implication - The product s-implication $I_s(a, b) = 1 - a + ab$.
- Existential Quantification - An aggregator E based on the mean,

$$E(x_1, ..., x_n) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- Universal Quantification - An aggregator U based on the root mean squared error,

$$U(x_1, ..., x_n) = 1 - \left(\frac{1}{n}\sum_{i=1}^{n}(1 - x_i)^2\right)^{\frac{1}{2}}$$

The mean and root mean squared error are used in place of the extended s-norm and t-norm is because the latter do not scale well to many arguments. The extended t-norm is very sensitive to outliers. A single zero truth value causes the entire quantification to become zero, which makes the extended t-norm vanishing on a large section of its domain. Additionally, the repeated multiplication of many small truth values can lead to numerical underflow. Therefore, the use of averaging aggregators proves more useful.

4.5.4 Satisfaction Aggregators

In order to evaluate the satisfaction of knowledge bases, a satisfaction aggregator needs to be specified. The most intuitive choice is an operator set's universal quantification operator which captures the meaning of satisfaction aggregation very well and offers interpretable semantics when paired with its operator set. Users are free to choose whichever satisfaction aggregator they want, but the recommended choice is the universal quantification operator.

4.6 Learning

A Fuzzy Tensor Logic system is able to learn from data. A formal definition of learning in FTL is given in section 5.4.1, therefore this section will instead present an intuitive overview of the process. Simply put, learning in an FTL agent is the process of maximising the satisfaction of its knowledge base. To do this, the differentiable properties of FTL are used to perform back-propagation.

Firstly, a forward pass is computed to achieve a measure of the current satisfaction of the knowledge base. This is done by using the algorithm in listing 4.3. Next, the satisfaction is converted into an appropriate loss. Then, the parameters of neural predicates and functors are adjusted slightly the lower the loss through back-propagation. The process is repeated for a set number of epochs. A pseudo-code implementation is given in listing 4.4.

```
# Algorithm: Train - Trains the FTL system via backpropagation.
# Inputs:
#   agent - FTL agent to train.
#   epochs - Number of epochs to train for.
#   sagg - A satisfaction aggregator.
# Output:
# Agent after training for the given number of epochs.

function train(agent, epochs):
    for i = 0 < epochs:
        satisfaction = evaluateKB(
            agent.knowledgeBase,
            agent.interpretation,
            agent.junctors,
            sagg
        )
        loss = calculateLoss(satisfaction)
        # calculate gradients of loss w.r.t agent's trainable
        # parameters.
        gradients = calculateGradients(loss, agent.parameters)
        agent.updateParameters(gradients)
```

Listing 4.4: Learning algorithm for a Fuzzy Tensor Logic system.

4.7 Querying

In first-order knowledge-based agents, querying is the task of determining whether a formula is logically entailed from a knowledge base. A formula ψ is logically entailed by a knowledge base KB, written $\mathsf{KB} \models \psi$, if all interpretations which satisfy KB also satisfy ψ. In many knowledge-based agents, the process of determining logical entailment is carried out by unification algorithms. Unification algorithms carry out substitutions on the contents of a knowledge base to determine whether the query is entailed by Modus Ponens (Russel and Norvig, 2021). Unification is therefore a purely symbolic algorithm based on well established logical equivalences between first-order formulas.

The situation in fuzzy logic is more complicated given the infinite possible truth values and wide range of possible semantics. Therefore, FTL systems adopt a much weaker concept of querying which still proves useful in practice. In FTL, querying an agent with an FTL formula simply returns the truth value of that formula as evaluated under the agent's learned interpretation. This straightforward idea is demonstrated as pseudo-code in listing 4.5.

```
# Algorithm: Query - Queries the FTL system.
# Inputs:
#   agent - An FTL agent to query.
#   pt - A parse tree with which to query the agent.
# Output:
# Truth value of pt under agent's interpretation.

function query(agent, pt):
    return evaluate(
        pt,
        agent.interpretation,
        agent.junctors)
```

Listing 4.5: Querying algorithm for a Fuzzy Tensor Logic system.

Chapter 5

Fuzzy Tensor Logic

This chapter defines Fuzzy Tensor Logic, which forms the basis of the neuro-symbolic systems developed during this project. Firstly, the syntax and semantics of Fuzzy Tensor Logic are given. Following that are discussions of knowledge bases and Fuzzy Tensor Logic agents. Finally, a definition of agent learning and querying are given. The syntactic and semantic definitions in this chapter use the first order logic definitions by Freydenberger (2020) as a starting point.

5.1 Syntax

The definition begins with the syntax of Fuzzy Tensor Logic.

Definition 1. A *signature* σ is a set of predicate symbols, functor symbols, and constant symbols. Every predicate symbol $\hat{P} \in \sigma$ and functor symbol $\hat{f} \in \sigma$ has an *arity* $\mathsf{ar}(\hat{P}) \in \mathbb{N}_1$ or $\mathsf{ar}(\hat{f}) \in \mathbb{N}_1$, respectively.

Fuzzy Tensor Logic signatures are like the signatures in other first order languages. Intuitively, they provide the vocabulary of symbols one wishes to reason over. In addition to this vocabulary, it is useful to have *variables.*

Definition 2. The set of *variables* VAR is defined as

$$\mathsf{VAR} := \{x_i \mid i \in \mathbb{N}_1\}$$

Although it aids the clarity of later definitions to define variables in this strict way, in practice any non-ambiguous and non-conflicting name can be used for a variable.

Notation 1. We permit the use of any non-ambiguous and non-conflicting symbol as a variable, e.g. `a`, `img_1`, `cat3`.

In addition to variables, Fuzzy Tensor Logic also introduces the concept of *domains.*

Definition 3. The set of *domains* DOM is defined as

$$\mathsf{DOM} := \{D_i \mid i \in \mathbb{N}_1\}$$

As will be made clear later, a domain is a set which groups together related elements of the universe (e.g. training images, digits, etc.). The purpose of domains is to allow the construction of quantification formulas which can be efficiently computed by only considering a subset of the universe. Much like variables, the strict definition of domains is to aid in later definitions, but in practice any non-ambiguous and non-conflicting name can be used for a domain.

Notation 2. We permit the use of any non-ambigious and non-conflicting symbol as a domain, e.g. `Y`, `Images`, `Train_set`.

Equipped with signatures and variables, it is possible to form *terms*.

Definition 4. For each signature σ, the set of *σ-terms* $\mathsf{Terms}(\sigma)$ is defined recursively as follows:

- For every constant symbol $\hat{c} \in \sigma$, we have $\hat{c} \in \mathsf{Terms}(\sigma)$.
- For every variable $x \in \mathsf{VAR}$, we have $x \in \mathsf{Terms}(\sigma)$.
- For every functor symbol $\hat{f} \in \sigma$ with $k = \mathsf{ar}(\hat{f})$, we have:

$$\text{if } t_1, ..., t_k \in \mathsf{Terms}(\sigma), \text{ then } \hat{f}(t_1, ..., t_k) \in \mathsf{Terms}(\sigma).$$

Just as with standard first order logic, a term is an expression which refers to some object within the universe. With terms defined, it is possible to proceed with the definition of *formulas*.

Definition 5. For each signature σ, the set $\mathsf{FT}[\sigma]$ of all *Fuzzy Tensor Logic formulas* over σ is defined recursively as follows:

- For every predicate symbol $\hat{P} \in \sigma$ with $k = \mathsf{ar}(\hat{P})$ and $t_1, ..., t_k \in \mathsf{Terms}(\sigma)$, we have $\hat{P}(t_1, ..., t_k) \in \mathsf{FT}[\sigma]$.

  Additionally, all formulas formed by this rule are called *atomic formulas* or simply *atoms*.
- If $\psi \in \mathsf{FT}[\sigma]$, then $(\psi) \in \mathsf{FT}[\sigma]$.
- If $\psi \in \mathsf{FT}[\sigma]$, then $\neg\psi \in \mathsf{FT}[\sigma]$.
- If $\psi, \phi \in \mathsf{FT}[\sigma]$, then
  - $\psi \wedge \phi \in \mathsf{FT}[\sigma]$,
  - $\psi \vee \phi \in \mathsf{FT}[\sigma]$,
  - $\psi \rightarrow \phi \in \mathsf{FT}[\sigma]$.
- If $\psi \in \mathsf{FT}[\sigma]$, $x \in \mathsf{VAR}$, and $D \in \mathsf{DOM}$, then
  - $\exists x \in D : \psi \in \mathsf{FT}[\sigma]$,
  - $\forall x \in D : \psi \in \mathsf{FT}[\sigma]$.

There are several aspects of this definition which clearly diverge from standard first-order logic syntax. Firstly, equality between terms is not included as an atomic formula (i.e. there are no formulas of the form $t_1 = t_2$). Secondly, quantifiers are limited to a domain. Although justification of these choices requires an understanding of Fuzzy Tensor Logic semantics and applications, they are briefly explained here.

Firstly, terms refer to objects of the universe, which in FTL are real valued tensors, and therefore evaluating equality between terms is evaluating the equality between tensors. By the standard definition of equality, this is not a differentiable operation, however formulas in FTL ought to be differentiable. Therefore, FTL leaves it to the user to implement their preferred form of differentiable equality (which is likely to be some differentiable similarity operation) as a predicate.

Secondly, the existential and universal quantifiers are limited to quantifying over domains. Evaluating quantifiers over large universes is computationally very expensive and therefore limiting quantification over an explicit domain limits this cost[1]. Additionally, users are likely to only want to quantify over particular sets of tensors (e.g. training sets), rather than all possible tensors in a universe, and limiting quantification to a domain achieves this in a clear and computationally efficient manner.

The syntax concludes with several definitions about the nature of variables in formulas and terms.

Definition 6. For a term $t \in \mathsf{Terms}[\sigma]$, the set of *term variables* $\mathsf{var}(t)$, contains exactly all variables which occur in t.

Definition 7. For a formula $\psi \in \mathsf{FT}[\sigma]$, the set of *formula variables* $\mathsf{var}(\psi)$, contains exactly all variables which occur in ψ.

Definition 8. For a formula $\psi \in \mathsf{FT}[\sigma]$, the set of *free variables* $\mathsf{free}(\psi)$, contains exactly all variables which occur unbounded to a quantifier in ψ.

Definition 9. For a formula $\psi \in \mathsf{FT}[\sigma]$, the set of *bound variables* $\mathsf{bound}(\psi)$, contains exactly all variables which occur bounded to a quantifier in ψ.

Definition 10. For a formula $\psi \in \mathsf{FT}[\sigma]$, the set of *domains* $\mathsf{doms}(\psi)$, contains exactly all domains which occur in ψ.

Notice that $\mathsf{free}(\psi)$ and $\mathsf{bound}(\psi)$ are not necessarily disjoint. Although it is bad practice, a variable can appear bound to a quantifier in one part of a formula and unbound in another. Finally, it's worth noting that $\mathsf{var}(\psi) = \mathsf{free}(\psi) \cup \mathsf{bound}(\psi)$.

To summarise, signatures provide a vocabulary of predicates, functors, and constants over which to reason. Variables represent individual members of the universe, while domains represent subsets of members from the universe. Using constants, variables, and functors, one can form terms which represent members of the universe. Formulas are formed using the usual first-order rules with two exceptions: there are no atomic formulas for equality between terms, and all quantifications are limited to members of a domain.

[1]For a universe of size n, the runtime of evaluating a quantifier is $O(n)$. If quantifiers are stacked k deep, then evaluating them incurs a runtime of $O(n^k)$. Clearly, limiting the set over which quantification occurs is important.

5.2 Semantics

The semantics of Fuzzy Tensor Logic is where it most noticeably departs from standard first order logic. The semantics of Fuzzy Tensor Logic begins with the definition of fuzzy operator sets.

Definition 11. A *fuzzy operator set* is a set FOP containing the following elements,

- A differentiable fuzzy negation N,
- A differentiable t-norm T,
- A differentiable t-conorm S,
- A differentiable fuzzy implication J,
- A differentiable fuzzy universal aggregator U,
- A differentiable fuzzy existential aggregator E.

In Fuzzy Tensor Logic, differentiable fuzzy operators are used to implement junctors. Therefore, fuzzy operator sets define exactly which fuzzy operators implement which junctors. The definition of semantics continues with *structures.*

Definition 12. A *structure* over the signature σ is a triple $\mathcal{A} := (A, \xi_\theta, \Theta)$ that contains

- A non-empty set of real-valued tensors A called the *universe*,
- A function ξ_θ, parameterised by $\theta \in \Theta$, called the *interpretation function* that maps
  - each predicate symbol $\hat{P} \in \sigma$ to a function $\xi_\theta(\hat{P}) : A^{\mathsf{ar}(\hat{P})} \mapsto [0, 1]$,
  - each functor symbol $\hat{f} \in \sigma$ to a function $\xi_\theta(\hat{f}) : A^{\mathsf{ar}(\hat{f})} \mapsto A$,
  - each constant symbol $\hat{c} \in \sigma$ to a real-valued tensor $\xi_\theta(\hat{c}) \in A$.
- A set Θ of possible parameters for ξ, called the *hypothesis space.*

Much like in standard first-order logic, a Fuzzy Tensor Logic structure provides meaning to the elements of a signature by mapping them to predicates, functions, and constants over and in a universe of objects. However, there are some very important differences. It is here that the coming together of logic and deep learning emerges.

Firstly, the universe is limited to only real valued tensors. Since Fuzzy Tensor Logic is a logic evaluated only over tensors, this must be the case. Although the contents of the universe can be arbitrary tensors, in practice it consists only of tensors which have specific meaning to the problem instance. For example, these tensors might represents images of digits, natural language sentences, or any other feature vector.

Secondly, rather than the interpretation function mapping predicates to relations over the universe, it instead maps predicates to fuzzy truth-valued functions over tensors. This allows for the mapping of predicates to objects such as neural networks. For example, a predicate $\hat{\mathsf{isDog}}$ could be mapped to a convolutional neural network which accepts images and outputs the probability whether the image is of a dog.

Additionally, the interpretation function is explicitly parameterised by a set of parameters θ drawn from a hypothesis space Θ. The practical idea here is that θ represents the parameters of the deep learning models which implement the predicates and functors. This idea will prove to be essential when learning is discussed. Crucially, Fuzzy Tensor Logic has been designed in such a way to allow the truth value of a formula to be differentiable with respect to θ. This property can be used to train the models which implement predicates and functors via back-propagation.

In order to improve readability in further definitions, an alternative notation for applying the interpretation function to a symbol is given.

Notation 3. For a given σ-structure $\mathcal{A} = (A, \xi_\theta, \Theta)$,

- if $\hat{P} \in \sigma$ is a predicate symbol, then $\hat{P}^{\mathcal{A}} := \xi_\theta(\hat{P})$,
- if $\hat{f} \in \sigma$ is a functor symbol, then $\hat{f}^{\mathcal{A}} := \xi_\theta(\hat{f})$,
- if $\hat{c} \in \sigma$ is a constant symbol, then $\hat{c}^{\mathcal{A}} := \xi_\theta(\hat{c})$.

The semantic definition proceeds to build to the concept of an *interpretation.*

Definition 13. A *variable assignment* for a σ-structure $\mathcal{A} = (A, \xi_\theta, \Theta)$ is a partial function $\alpha : \mathsf{VAR} \mapsto A$.

Definition 14. A *domain assignment* for a σ-structure $\mathcal{A} = (A, \xi_\theta, \Theta)$ is a partial function $\beta : \mathsf{DOM} \mapsto \mathcal{P}(A) - \emptyset$, where $\mathcal{P}(A)$ denotes the power set of A.

Definition 15. A *σ-interpretation* is a triple $I = (\mathcal{A}, \alpha, \beta)$ where $\mathcal{A}$ is a σ-structure, α is a variable assignment for $\mathcal{A}$, and β is a domain assignment for $\mathcal{A}$.

Equipped with an interpretation, the values of terms can be decided.

Notation 4. For a function $f : X \mapsto Y$, let $\mathsf{domain}(f) = X$.

Definition 16. A σ-interpretation $I = (\mathcal{A}, \alpha, \beta)$ is an *interpretation for a σ-term t* if $\mathsf{domain}(\alpha) \supseteq \mathsf{var}(t)$.

In other words, an interpretation is an *interpretation for a term* if the interpretation's variable assignment maps every variable which appears in the term.

Definition 17. Given a signature σ, a σ-term $t \in \mathsf{Terms}(\sigma)$, and a σ-interpretation $I = (\mathcal{A}, \alpha, \beta)$ for t, we define the value $[\![t]\!]^I$ of t under I recursively as follows:

- For all $x \in \mathsf{VAR}$, we define $[\![x]\!]^I := \alpha(x)$.
- For all constant symbols $\hat{c} \in \sigma$, we define $[\![\hat{c}]\!]^I := \hat{c}^{\mathcal{A}}$.
- For all functor symbols $\hat{f} \in \sigma$ with $k = \mathsf{ar}(\hat{f})$ and for all σ-terms $t_1, ..., t_k \in \mathsf{Terms}(\sigma)$, we define

$$[\![\hat{f}(t_1, ..., t_k)]\!]^I := \hat{f}^{\mathcal{A}}([\![t_1]\!]^I, ..., [\![t_k]\!]^I)$$

Put simply, evaluating variables is done by applying the variable assignment function unto them, evaluating constants is done by applying the signature's interpretation function unto them, and evaluating functor application is done by applying the signature's interpretation of the functor to its arguments. As one would expect, the values of terms are all elements of the structure's universe. The definition for semantics concludes with the evaluation of formulas.

Definition 18. A σ-interpretation $I = (\mathcal{A}, \alpha, \beta)$ is an *interpretation for a formula* ψ if $\mathsf{domain}(\alpha) \supseteq \mathsf{free}(\psi)$ and $\mathsf{domain}(\beta) \supseteq \mathsf{doms}(\psi)$.

In other words, an interpretation is an interpretation for a formula if the variable assignment maps every free variable which appears in the formula and the domain assignment maps every domain which appears in the formula.

Definition 19. For an assignment function α, we define the *extended assignment substituting x for y*, denoted $\alpha_{x \to y}$, as

$$\alpha_{x \to y}(n) = \begin{cases} \alpha(n) & \text{if } n \neq x, \\ y & \text{if } n = x. \end{cases}$$

Definition 20. For a σ-interpretation $I = (\mathcal{A}, \alpha, \beta)$, we define the *extended interpretation substituting x for y*, as $I_{x \to y} = (\mathcal{A}, \alpha_{x \to y}, \beta)$.

Definition 21. Given a signature σ, an operator set $\mathsf{FOP} = \{N, T, S, J, U, E\}$, a formula $\psi \in \mathsf{FT}[\sigma]$, and a σ-interpretation $I = (\mathcal{A}, \alpha, \beta)$ for ψ, we define $[\![\psi]\!]^I$, the *truth value of ψ under I w.r.t* FOP, recursively as follows:

- For every predicate symbol $\hat{P}$ with $k = \mathsf{ar}(\hat{P})$, and for all σ-terms $t_1, ..., t_k \in \mathsf{Terms}(\sigma)$, we define

$$[\![\hat{P}(t_1, ..., t_k)]\!]^I := \hat{P}^{\mathcal{A}}([\![t_1]\!]^I, ..., [\![t_k]\!]^I).$$

- $[\![(\psi)]\!]^I := [\![\psi]\!]^I$.
- $[\![\neg\psi]\!]^I := N([\![\psi]\!])$.
- For all $\psi, \phi \in \mathsf{FT}[\sigma]$,
  - $[\![\psi \wedge \phi]\!]^I := T([\![\psi]\!]^I, [\![\phi]\!]^I)$,
  - $[\![\psi \vee \phi]\!]^I := S([\![\psi]\!]^I, [\![\phi]\!]^I)$,
  - $[\![\psi \to \phi]\!]^I := J([\![\psi]\!]^I, [\![\phi]\!]^I)$,
- For each $D \in \mathsf{DOM}$, let $\beta(D)_i$ denote the *ith* element in $\beta(D)$, and let $k = |\beta(D)|$, then
  - $[\![\forall x \in D : \psi]\!]^I := U([\![\psi]\!]^{I_{x \to \beta(D)_1}}, ..., [\![\psi]\!]^{I_{x \to \beta(D)_k}})$
  - $[\![\exists x \in D : \psi]\!]^I := E([\![\psi]\!]^{I_{x \to \beta(D)_1}}, ..., [\![\psi]\!]^{I_{x \to \beta(D)_k}})$

Evaluating the truth value of a formula is straightforward. Predicates are evaluated by applying their associated function to the evaluations of their terms. Formulas involving junctors are evaluated by applying their associated fuzzy operation to the evaluations of their sub-formulas. With quantifiers, the sub-formula is evaluated under all extended interpretations substituting the variable for each element in the domain, and the results are aggregated using the fuzzy aggregator.

Finally, we define the standard precedence of the operators which appear in Fuzzy Tensor Logic formulas.

Definition 22. The standard order of operator precedence is the following, in order from highest to lowest:

- Parentheses
- Negation
- Quantification
- Conjunction
- Disjunction
- Implication

5.3 Knowledge Bases

Fuzzy Tensor Logic allows one to write formulas to express symbolic knowledge about a problem. The purpose of a *knowledge base* is to collect formulas so that they can be reasoned over as a whole.

Definition 23. A knowledge base over a signature σ is a set $\mathsf{KB}_\sigma \subseteq \mathsf{FT}[\sigma]$.

The previous definition for evaluating the truth value of a formula can be naturally extended to evaluating the truth value of an entire knowledge base. To evaluate the satisfaction (or truthfulness) of a knowledge base, each of its formulas are individually evaluated and the results are aggregated by a suitable *satisfaction aggregator.*

Definition 24. A satisfaction aggregator is a differentiable function $\mathsf{SAG} : [0,1]^* \mapsto [0,1]$.

Definition 25. Given a knowledge base KB_σ, a σ-interpretation I, and satisfaction aggregator SAG, we define the *satisfaction of* KB *under* I as

$$[\![\mathsf{KB}_\sigma]\!]^I = \mathsf{SAG}_{\psi \in \mathsf{KB}_\sigma}([\![\psi]\!]^I)$$

What makes a suitable satisfaction aggregator depends on the fuzzy operators in FOP. A simple choice is the universal quantifier. In symmetric operator sets, the universal quantifier is just conjunction extended to an arbitrary number of arguments.

A natural question for a knowledge base is the following: which interpretation maximises the satisfaction of the knowledge base? Solving this problem is central to the process learning with Fuzzy Tensor Logic agents, as will be discussed shortly. A formal definition of this problem follows.

Definition 26. Given a knowledge base KB_σ, a fuzzy operator set FOP, a satisfaction aggregator SAG, and a σ-interpretation $I = (\mathcal{A}, \alpha, \beta)$ with $\mathcal{A} = (A, \xi_\theta, \Theta)$, the *maximum knowledge base satisfaction problem* is finding a solution θ^* to the following

$$\theta^* = argmax_{\theta \in \Theta}[\![\mathsf{KB}_\sigma]\!]^I$$

In particular, the maximum knowledge base satisfaction problem is about finding a maximally satisfying value for θ. Since θ is the parameter for the interpretation function, this in practice makes θ the parameters for the neural networks which implement the functors and predicates. Therefore, one can interpret the maximum knowledge base satisfaction problem as trying to find parameters for the functor and predicate neural networks which make the knowledge base 'most true'.

5.4 Agents

With Fuzzy Tensor Logic and knowledge bases fully defined, it is finally time to put everything together to define Fuzzy Tensor Logic *agents*.

Definition 27. An *agent* (also called a model) is a triple $\mathsf{Agent} = (\mathsf{KB}_\sigma, I, \mathsf{FOP})$ where KB_σ is a knowledge base over σ, I is a σ-interpretation, and FOP is a fuzzy operator set.

The agent's knowledge base KB_σ is a collection of formulas which express all available symbolic knowledge relevant to the agent's task. These formulas can range from axiomatic definitions about predicates to quantifications over data. The agent's interpretation I provides implementations for all constants, functors, and predicates in addition to mappings for variables and domains. Importantly, the interpretation is parameterised, which allows for functors and predicates to be learned. Finally, the fuzzy operator set FOP provides the differentiable fuzzy logic operators which enable the evaluation σ-formulas, like those in KB_σ.

5.4.1 Learning

The concept of an agent undergoing learning in Fuzzy Tensor Logic is straightforward. Simply put, learning is the process of solving the maximum knowledge base satisfaction problem. As with logical satisfaction problems, the search space of solutions is incredibly large which makes naive or general search methods impractical. However, agents are able to side-step this difficulty by taking advantage of the differentiability of Fuzzy Tensor Logic formulas to solve the problem by gradient descent.

In particular, gradient descent is performed by a chosen back-propagation algorithm. Since it is standard practice for gradient descent to minimise a loss function (rather than maximise a utility function), solving the maximum knowledge base satisfaction problem is re-framed as minimising a loss function. In addition, to aid with the effectiveness of training neural networks, a regularisation term is included in the loss.

Definition 28. For an agent $\mathsf{Agent} = (\mathsf{KB}_\sigma, I, \mathsf{FOP})$ and a regularisation function R, the *knowledge base satisfaction loss* L is defined as

$$L(\mathsf{Agent}) = (1 - [\![\mathsf{KB}_\sigma]\!]^I) - R(\mathsf{Agent})$$

Although a formal proof will not be given, it should be clear to see that L is differentiable with respect to the parameters θ of I so long as the functors and predicates which use θ are themselves differentiable. Thankfully, this is the case if the predicates and functors are implemented by any standard deep learning architecture.

Hopefully, it is now possible to understand Fuzzy Tensor Logic agents from a deep learning perspective. In particular, one can view the entire Fuzzy Tensor Logic system as offering a way to inject symbolic domain knowledge into, and impose constraints upon, the deep learning training process. By treating deep learning models as predicates and functors within a first order language, one is able to build a knowledge base defining desired model behaviour. Models are then directly trained on this knowledge base to implement the desired behaviour as well as they can. The process of training is carried out by the usual methods of gradient descent via back-propagation.

5.4.2 Querying

The concept of agent *querying* is very straightforward. Simply put, querying an agent is equivalent to determining the truth value of the query under the agent's interpretation.

Definition 29. For an agent $\mathsf{Agent} = (\mathsf{KB}_\sigma, I, \mathsf{FOP})$ and a σ-formula $\psi \in \mathsf{FO}[\sigma]$, the *response of* Agent *to the query* ψ, denoted $\mathsf{Agent}(\psi)$, is defined as

$$\mathsf{Agent}(\psi) = [\![\psi]\!]^I$$

Chapter 6

Fuzzy Tensor Logic Implementation

This chapter documents the implementation of Fuzzy Tensor Logic as a Python 3 package.

6.1 Package Overview

The Fuzzy Tensor Logic package `ftlogic` implements all of the basic components necessary to build and use Fuzzy Tensor Logic agents. The package is composed of two sub-packages, namely `core` and `fuzzyops`. The `core` sub-package consists of modules which can be used to define, train, and use a Fuzzy Tensor Logic system, while the `fuzzyops` sub-package contains a broad set of differentiable fuzzy logic operators.

The following details the modules within `core`:

- `interpretation.py` - Contains an `Interpretation` class enabling users to create Fuzzy Tensor Logic interpretations. Additionally contains methods for evaluating the truthfulness of formulas and knowledge bases.
- `knowledgebase.py` - Contains a `KnowledgeBase` class enabling users to compile Fuzzy Tensor Logic formulas into knowledge bases for training and querying.
- `model.py` - Contains a `Model` class enabling users to build, train, and query Fuzzy Tensor Logic agents.
- `parser.py` - Contains a `ParseTree` class which is used to store parse trees of Fuzzy Tensor Logic expressions. Additionally contains a `parse` method for parsing formulas into `ParseTree` objects.
- `signature.py` - Contains a `Signature` class, enabling users to create Fuzzy Tensor Logic signatures.
- `structure.py` - Contains a `Structure` class, enabling users to create Fuzzy Tensor Logic structures over a signature.

The fuzzyops package contains a single module, operators, which contains a wide variety of fully differentiable fuzzy logic operators including negations, t-norms, t-conorms, fuzzy implications, and existential and universal aggregators. These operators are implemented as Python functions which make use of TensorFlow 2 methods to ensure differentiability. Additionally, the module contains an OperatorSet class which allows users to compile a selection of fuzzy operators for use in evaluating formulas. For the user's convenience, the module contains a OperatorSet object implementing the recommended Standard Product Set.

Please note that in order to use the package included with this report, it needs to be properly installed. Installation instructions are included in the readme.txt file.

6.1.1 Tensors and Differentiability

Fuzzy Tensor Logic is a first order logic over tensors. Therefore, tensors and tensor operations form the basis of most operations in FTL. Internally, FTL systems use TensorFlow 2 Tensor objects and their associated functions to implement tensor-based operations, including differentiation. FTL systems expect all user-defined tensors to be either Tensor objects or objects of a type which can be converted to a Tensor object by TensorFlow, such as the NumPy ndarray.

Tensors come in all shapes and sizes and are frequently manipulated in batches. In order to reduce ambiguities when working with tensors, the implementation of FTL expects all tensors to be provided in batches, even if the resulting batch size is one. Enforcing this constraint allows the implementation to make assumptions about the shapes of tensors, which in turn increases readability and reduces the amount of unnecessary tensor reshaping. Additionally, many TensorFlow operations expect their inputs to be batched and extending this expectation to FTL allows for easy integration of TensorFlow functions into the system.

The computing of gradients during the FTL learning operation makes use of TensorFlow's GradientTape object. Therefore, any functors or predicates which the user wishes to be trained during the process of learning must be capable of producing gradients within a GradientTape scope. This is naturally the case for properly designed neural networks implemented in TensorFlow or Keras. All FTL algorithms which are involved in the learning process make appropriate use of TensorFlow operations to ensure the necessary values can be differentiated. This extends to all fuzzy operators included in the fuzzyops package. The differentiability of the FTL implementation has been verified through extensive testing.

6.1.2 Development

The implementation of ftlogic followed an iterative process. Components were developed and tested, and were later refactored based on performance and usability. Initially, the implementation of most modules and classes was more complex, including extensive type checks, wide use of getter and setter methods, and many wrapper methods. As development unfolded, the decision was made to reduce the complexity of the implementation until the package reached its current state.

Many of the since removed class methods proved unnecessary and less usable than directly working with object properties. It is widely considered good Python design to avoid excessive 'bloat' within class designs, especially when the extra functionality only serves to prevent the user from doing something they know they shouldn't do. Since many of the classes in the implementation of FTL are essentially light-weight bundles of data, extensive accessor functions and wrapper methods were removed and their attributes were made public. Although this runs against the best practices of object oriented languages like Java and C++, it is in accordance with the Python PEP-8 style guide (Rossum, Warsaw, and Coghlan, 2013). Furthermore, this design philosophy is widely observed in Python packages such as Keras.

Where accessor-like behaviour is necessary, Python properties have been used instead of explicit getter and setter functions. Additionally, many class methods have been reworked as overridden Python special functions such as `__eq__` and `__contains__`. The overall effect is simpler and more readable code. Incorrect use of objects is caught where relevant using Python exceptions. Python docstrings are used extensively to clarify the behaviour and expected usage of public module elements.

6.2 Signature Module

Signatures provide a vocabulary of symbols which can be used to construct Fuzzy Tensor Logic formulas. The `Signature` Module contains the `Signature` class which allows for easy storage and retrieval of symbols and their arities.

6.2.1 Signature Class

The `Signature` class provides a straightforward implementation of Fuzzy Tensor Logic signatures.

Signature Class Properties

The `Signature` class contains the following properties:

- `predicates` - A dictionary containing the signature's predicates. Each entry maps a predicate to its associated arity.
- `functors` - A dictionary containing the signature's functors. Each entry maps a functor to its associated arity.
- `constants` - A list containing the signature's constants.

Signature Class Methods

The Signature class contains the following methods:

- `__init__(predicates, functors, constants)` - The class constructor. Accepts optional dictionaries of predicates, functors, and a list of constants to initialise the object's respective properties.
- `__contains__(symbol)` - Returns true if the `Signature` contains the symbol as a functor, predicate, or constant. Otherwise, returns false.

- `__eq__(other)` - Returns true if `other` is of type `Signature` with equal `predicates`, `functors`, and `constants` properties. Otherwise, returns false.

6.3 Structure Module

Structures map the elements of a signature to tangible objects. Constants are mapped to elements of the universe, functors are mapped to functions over the universe, and predicates are mapped to functions from the universe to a fuzzy truth value. The `Structure` module provides the `Structure` class to implement structures in a straight-forward manner.

6.3.1 Structure Class

The `Structure` class provides a way to store and retrieve mappings of elements in a `Signature` object.

Structure Class Properties

The `Signature` class contains the following list of properties:

- `signature` - A `Signature` object over which the structure is defined.
- `mappings` - A dictionary which maps elements in `Signature` to appropriate objects. Constants are mapped to tensors (e.g. TensorFlow Tensor objects), functors are mapped to functions which accept and return tensors, and predicates are mapped to functions which accept tensors and return fuzzy truth valued tensors. Predicates and functors can be mapped to neural networks.
- `complete` - True if `mappings` contains an entry for every symbol in the object's `signature`. False otherwise.

Recall that in definition 5.2, a structure over a signature consists of a universe, interpretation function, and hypothesis space. The `Structure` class does contain an explicit representation of the universe as this is both unnecessary and computationally unfeasible. Instead, the universe is defined implicitly by the possible values that functor applications, constants, variables, and domains can take. In practice, it is not necessary for FTL systems to know explicitly what the universe actually contains.

The `Structure` class implements the interpretation function as the `mappings` dictionary. The change of name is to avoid confusion with `Interpretation` objects and their methods. Additionally, the hypothesis space is implicitly defined by the hypothesis spaces of the neural networks to which functors and predicates are mapped. Similarly, the current parameter of the interpretation function is implicitly defined by the current parameters of the neural networks. Again, an explicit representation of the hypothesis space and choice of parameters is unnecessary to compute with FTL systems.

It's worth noting that `complete` is not an attribute in the strict sense, but rather a Python property. In other words, it is a function which dynamically computes the value of `complete` when called, as shown in listing 6.1. The complete property is necessary to check whether a given Signature can be safely used to evaluate a parse tree over a signature.

```
@property
def complete(self):
    for p in self.signature.predicates.keys():
        if not p in self.mappings:
            return False

    for f in self.signature.functors.keys():
        if not f in self.mappings:
            return False

    for c in self.signature.constants:
        if not c in self.mappings:
            return False

    return True
```

Listing 6.1: The complete property of Structure.

Structure Class Methods

The Structure class contains the following methods:

- __init__(signature, mappings) - The class constructor. Accepts arguments to populate the object's corresponding properties.
- __contains__(symbol) - Returns true if the Structure contains an entry in mappings for symbol. Otherwise, returns false.
- __call__(symbol) - Returns the entry in mappings for symbol.
- __eq__(other) - Returns true if other is of type Structure with equal signature and mappings properties. Otherwise, returns false.

6.4 Parser Module

In order to manipulate FTL formulas, the implementation requires a representation which is both computationally efficient and unambiguous. As explained in section 4.3.1, the chosen solution is parse trees. The Parser module provides a parse method and ParseTree class to parse and represent parse trees, respectively. Additionally, a NodeType enumeration is made available which contains an entry for each possible syntactic element in a parse tree.

The syntax of FTL chooses to use classical logic notation to denote quantifiers and junctors. However, such symbols are difficult to type and therefore the implementation supports alternative notations. Users are free to provide a set of symbols to represent junctors, but the default is called the Standard Junctor Symbol Set and uses the following:

- negation ($\neg$) is represented by !
- conjunction ($\wedge$) is represented by ,
- disjunction ($\vee$) is represented by ;

- implication ($\rightarrow$) is represented by `-:`

Additionally, existential quantification ($\exists x \in D$) is represented by `Ex~D`. Finally, universal quantification ($\forall x \in D$) is represented by `Ax~D`.

6.4.1 NodeType Enum and ParseTree Class

The `NodeType` enumeration contains values to represent the various syntactic elements of FTL formulas. `NodeType` values can then be used, as the name suggests, to assign types to nodes in parse trees. The `NodeType` enumeration implementation is given in listing 6.2.

```
class NodeType(Enum):
    UNIVERSAL = 1,
    EXISTENTIAL = 2,
    IMPLICATION = 3,
    DISJUNCTION = 4,
    CONJUNCTION = 5,
    NEGATION = 6,
    PREDICATE = 7,
    VARIABLE = 8,
    CONSTANT = 9,
    BRACKET = 10,
    FUNCTOR = 11,
```

Listing 6.2: NodeType enum implementation

The `ParseTree` class provides a straightforward implementation of parse trees. Every node within a parse tree is represented by a `ParseTree` object which stores its type, value, signature, and references to its children. Additionally, a `ParseTree` object contains methods to give further information about its syntactic properties, such as its variables and whether it is an FTL term or formula.

ParseTree Class Properties

The `ParseTree` class contains the following properties:

- `value` - In short, value contains any information necessary to understand the node which is not represented by the other three properties. In the case of predicates, functors, constants, and variables, it is their associated symbol. For quantifiers, it is the symbol of the quantifier's domain. For junctors and brackets, value is not required, but for the sake of readability the value can take a descriptive string.
- `type` - An element of `NodeType` representing the type of the node, e.g. constant, implication, universal quantification, etc.
- `children` - A list of child `ParseTree` objects. Can be empty.
- `signature` - The `Signature` object over which the formula was written.

ParseTree Class Methods

The `ParseTree` class contains the following methods:

- `__init__(value, type, children, signature)` - The class constructor. Accepts arguments to initialise the objects' corresponding properties.
- `__eq__(other)` - Returns true if `other` is of type `ParseTree` with equal `value`, `type`, `children`, and `signature` properties. Otherwise, returns false.
- `__str__()` - Returns a string representation of the parse tree. The string consists of a separate line for each level of the parse tree. Each node on a line is rendered in the form `<value>[<type>](<no. of children>)`. An example of the formatting is given in 6.3.
- `isFormula()` - Returns true if the parse tree represents a formula over its signature. Otherwise, returns false.
- `isTerm()` - Returns true if the parse tree represents a term over its signature. Otherwise, returns false.
- `isAtom()` - Returns true if the parse tree represents an atomic formula over its signature. Otherwise, returns false.
- `getVariables()` - Returns a list of all variables which appear in the parse tree.
- `getFree()` - Returns a list of all variables which appear as free in the parse tree.
- `getBound()` - Returns a list of all variables which appear as bound in the parse tree.
- `getDomains()` - Returns a list of all domains which appear in the parse tree.

```
parse tree for Ex~D:!P(c, f(x))
depth 0:  Ex~D[EXISTENTIAL](1)
depth 1:  ![NEGATION](1)
depth 2:  P[PREDICATE](2)
depth 3:  c[CONSTANT](0) f[FUNCTOR](1)
depth 4:  x[VARIABLE](0)
```

Listing 6.3: The string representation of a parse tree.

The methods `isFormula`, `isTerm`, and `isAtom` work by simply checking the root node's `type`, as shown in listing 6.4. Comparing these methods with the definitions of formulas, terms, and atoms in chapter 5 will demonstrate their validity.

```
def isFormula(self):
    return self.type == NodeType.CONJUNCTION or \
           self.type == NodeType.DISJUNCTION or \
           self.type == NodeType.EXISTENTIAL or \
           self.type == NodeType.IMPLICATION or \
           self.type == NodeType.UNIVERSAL or \
           self.type == NodeType.BRACKET or \
           self.type == NodeType.NEGATION or \
           self.type == NodeType.PREDICATE

```

```
def isTerm(self):
    return self.type == NodeType.CONSTANT or \
           self.type == NodeType.VARIABLE or \
           self.type == NodeType.FUNCTOR

def isAtom(self):
    return self.type == NodeType.PREDICATE
```

Listing 6.4: ParseTree property checking methods.

Methods for getting variables and domains operate by recursively navigating the tree and compiling all matching elements into a list. The implementation of `getBound` is provided as an example in listing 6.5.

```
def getBound(self):
    if self.isTerm():
        return []
    if (self.type == NodeType.UNIVERSAL
        or self.type == NodeType.EXISTENTIAL):
        vars = self.children[0].getBound()
        vars.extend(self.value[0])
        return vars
    else:
        vars = []
        for child in self.children:
            vars.extend(child.getBound())
        return vars
```

Listing 6.5: ParseTree getBound method implementation.

6.4.2 Parse Method

The `parse` method maps string representations of formulas to their parse tree representation. If successful, the method returns a `ParseTree` object, if unsuccessful the method raises a `ParseError`.

The method has the signature `parse(formula, signature, precedence, junctorSymbols, onlyFormulas)`. It accepts a string representation of an FTL formula, a `Signature` object, a list of `NoteType` elements in order of precedence, a dictionary of junctor symbols, and a boolean indicating whether to raise an exception if `formula` is not an FTL formula (i.e. it is a FTL term).

The `parse` method uses helper functions which are private to the module in order to effectively parse string formulas. These include a `_varSyntaxCheck` method which uses regular expressions to syntax check variable and domain symbols, and a `_findTopLevel` method which finds the top-level occurrence (i.e. occurrence nested in the least number of parentheses) of a junctor. The method responsible for parsing is actually a private helper function `_parse`. The public `parse` method calls this helper function to parse the formula and then performs necessary checks on the resulting `ParseTree` before returning it to the caller.

The implementation of `parse` is considerably more complex than the one proposed in listing 4.1 in order to improve efficiency and provide error checking, but the basic outline remains the same. The implementation also supports customisable junctor symbols. The module provides default junctor symbols and a standard precedence which are used as default arguments for the parse method. To avoid cluttering the report, the full code listing is not given, but listing 6.6 provides the section for parsing predicates and functors.

```python
def _parse(formula, signature, precedence, junctorSymbols):
    for type in reversed(precedence):
        ... # conditionals which parse other syntactic elements.

        #Attempt to parse formula as predicate or functor.
        elif type == NodeType.PREDICATE or type == NodeType.FUNCTOR:
            #Get predicate or functor symbol.
            resultIndex = formula.find("(")
            if resultIndex == -1:
                continue
            symbol = formula[0 : resultIndex]

            #Check symbol is in signature and get its arity.
            arity = 0
            if (type == NodeType.PREDICATE
                and symbol in signature.predicates):
                arity = signature.predicates[symbol]
            elif (type == NodeType.FUNCTOR
                and symbol in signature.functors):
                arity = signature.functors[symbol]
            else:
                continue

            #Process functor/predicate arguments.
            arguments = formula[resultIndex + 1 : -1]
            arguments = _splitAtLevel(arguments, ",")
            if len(arguments) != arity:
                raise ParseError(
                    f"Symbol {symbol} has arity {arity} but" +
                    f"has {len(arguments)} arguments in {formula}.")

            #Attempt to parse subexpressions.
            children = []
            for argument in arguments:
                parsedArgument = _parse(
                    argument, signature, precedence, junctorSymbols)
                #Check children are in fact terms.
                if not parsedArgument.isTerm():
                    raise ParseError(f"Subexpression" +
                    f"{argument} in {formula} is not a term.")
                children.append(parsedArgument)

            return ParseTree(symbol, type, children, signature)

        ... # conditionals which parse other syntactic elements.
```

Listing 6.6: parse method implementation for parsing functors and predicates.

6.5 KnowledgeBase Module

A knowledge base over a signature is simply a collection of formulas written over the signature. The `KnowledgeBase` module provides a `KnowledgeBase` class to implement knowledge bases.

6.5.1 KnowledgeBase Class

The `KnowledgeBase` class provides a straightforward implementation of FTL knowledge bases.

KnowledgeBase Properties

The `KnowledgeBase` class contains the following properties:

- `signature` - A `Signature` object over which all formulas in the knowledge base are written.
- `parseTrees` - A list of `parseTree` objects representing all FTL formulas within the knowledge base.

KnowledgeBase Methods

The `KnowledgeBase` class provides the following methods.

- `__init__(signature, formulas)` - The class constructor. Accepts arguments to initialise the object's corresponding properties. `formulas` is expected to be a list containing `ParseTree` objects and/or strings. Whenever a string is encountered, the constructor parses it before adding it to the `parseTrees` property.
- `__len__()` - Returns the number of elements in `parseTrees`.
- `__contains__(formula)` - Returns true if `formula` is in `parseTrees`. Otherwise, returns false. If `formula` is a string representation of a formula, then it is parsed before checking its inclusion in `parseTrees`.
- `add(formula)` - Adds `formula` to `parseTrees`. If `formula` is of type `ParseTree`, then raises `ValueError` if `formula`'s signature does not match `signature`. If `formula` is not of type `ParseTree`, attempts to parse the formula first.

Although the user is free to add `ParseTree` objects to the `parseTree` property, the `add` method provides a more robust way of doing so by ensuring signatures are consistent across all elements of the knowledge base. Both `add` and the the constructor `__init__` allow users to provide formulas either in parse tree form or as strings to reduce the work required by the user when setting up a knowledge base.

6.6 Interpretation Module

An interpretation over a signature provides mappings for all of the constants, functors, predicates, variables, and domains which can appear in formulas over a signature. Equipped with an interpretation, it is then possible to evaluate the truthfulness formulas.

The `Interpretation` module provides an `Interpretation` class to implement interpretations. The module also provides evaluation methods `evaluate` and `evaluateKB` to evaluate the truthfulness of formulas and knowledge bases, respectively.

6.6.1 Interpretation Class

The `Interpretation` class provides a straightforward way to store the mappings contained within an interpretation. Additionally, the class provides methods to call these mappings and build extended interpretations.

Interpretation Class Properties

The `Interpretation` class contains the following list of properties.

- `structure` - A `Structure` object.
- `variableAssignment` - A dictionary mapping variable names to real valued tensors.
- `domainAssignmnent` - A dictionary mapping domain names to batches of real valued tensors.

Comparing definition 5.2 to the properties of the `Interpretation` class shows that the class offers a direct implementation of the definition.

Interpretation Class Methods

The `Interpretation` class contains the following methods.

- `__init__(structure, variableAssignment, domainAssignment)` - The class constructor. Accepts arguments to initialise the object's corresponding properties.
- `__call__(symbol)` - Returns `interpret(symbol)`.
- `interpet(symbol)` - Returns the mapping of the given symbol. If the `symbol` is a predicate, functor, or constant, then the mapping in `signature` is returned. If `symbol` is a variable, the mapping in `variableAssignment` is returned. If `symbol` is a domain, the mapping in `domainAssignment` is returned. If `symbol` cannot be mapped, raises a `ValueError`.
- `extend(substitution)` - Returns a copy of the `Interpretation` object with its `variableAssignment` dictionary updated to contain values in the given `substitution` dictionary.

The method `interpret` has a very straightforward implementation which simply checks for the existence of the symbol in `structure`, `variableAssignment`, and `domainAssignment` and simply returns the corresponding value in the appropriate dictionary. The method `extend` is used to construct extended interpretations when evaluating quantifiers, as described in chapter 5.

6.6.2 Evaluation Methods

In order to evaluate formulas and knowledge bases, the module provides the `evaluate` and `evaluateKB` methods, respectively. The formula evaluation method has the signature `evaluate(parseTree, interpretation, junctors)` and returns a TensorFlow tensor containing the truth value of `parseTree` evaluated under `interpretation` with respect to the fuzzy operator set `junctors`.

The knowledge base evaluation method has the signature `evaluateKB(knowledgeBase, interpretation, junctors, aggregator)` and returns a TensorFlow tensor containing the truth value of `knowledgeBase` evaluated under `interpretation` with respect to the fuzzy operator set `junctors` and satisfaction aggregator `aggregator`.

The implementation of both functions closely follow the pseudo-code algorithms given in listings 4.2 and 4.3. During testing, both functions were shown capable of producing gradients for neural functors and predicates when called within a TensorFlow `GradientTape` environment so long as the given junctors and satisfaction aggregator were differentiable TensorFlow functions.

6.7 Model Module

An FTL agent (or model) is a neuro-symbolic agent capable of learning and reasoning. The `Model` module provides the `Model` class which implements agents, including their learning and querying operations.

6.7.1 Model Class

The `Model` class provides a straightforward way to create, train, and query FTL agents.

Model Class Properties

The `Model` class contains the following properties:

- `knowledgeBase` - A `KnowledgeBase` object containing all symbolic information known to the agent.
- `interpretation` - An `Interpretation` object which maps all syntactic elements of `knowledgeBase` to appropriate objects.
- `operatorSet` - An `OperatorSet` object mapping all functors and quantifiers to fuzzy differentiable logic operators.

The class definition is straightforward implementation of the FTL agent definition 5.4.

Model Class Methods

The `Model` class contains the following methods:

- `__init__(knowledgeBase, interpretation, operatorSet)` - The class constructor. Accepts arguments to initialise the object's corresponding properties.
- `fit(epochs, optimiser, aggregator, trainables, regulariser)` - Performs back-propagation on `Model` for `epochs` iterations to minimise the knowledge base satisfaction loss (definition 5.4.1) using the given satisfaction aggregator `aggregator`. Only the weights given in `trainables` are modified. Updates are made using the given `optimiser`. An optional `regulariser` argument can be provided to add a regularisation penalty

to the loss. Returns a dictionary containing the loss, knowledge base satisfaction, and individual formula satisfaction at each epoch.

- `query(formula)` - Evaluates the truthfulness of `formula` using the model's `interpretation` and `operatorSet` properties. `formula` can be either a `ParseTree` object or a string representation of the formula. If `formula` has an incompatible signature or otherwise fails to pass, a `ValueError` is raised.

The `fit` method implements the agent learning algorithm proposed in listing 4.4. The method has the signature `fit(epochs, optimiser, aggregator, trainables, regulariser, validationKB)`. It accepts a number of epochs to train over, an Keras style optimiser, a satisfaction aggregator function, a list of trainable weights, an optional regulariser function, and an optional validation knowledge base. The optional validation knowledge base must contain formulas over the same signature as the model's `knowledgeBase` property. The method returns a dictionary containing the knowledge base satisfaction, loss, and optional validation knowledge base satisfaction for each epoch.

The implementation of `fit` is given in listing 6.7. Gradients are computed using the TensorFlow `GradientTape` environment. The `optimiser` can be any function which has an `apply_gradients` method to update the weights in `trainables`. This makes `fit` compatible with all Keras `Optimizer` objects.

```
def fit(self, epochs, optimiser, aggregator, trainables,
        regulariser=None, validationKB=None):
    history = {"loss": [],
               "formula_satisfaction": [
                   [] for _ in self.knowledgeBase],
               "satisfaction": []}
    if validationKB:
        if validationKB.signature != self.knowledgeBase.signature:
            raise ValueError("Knowledge base signatures must match.")
        history["validation_satisfaction"] = [
            [] for _ in validationKB]

    print(f"Training model for {epochs} epochs.")
    for i in range(0, epochs):
        print(f"epoch {i + 1} / {epochs}. ", end=None)

        #Compute the loss.
        with tf.GradientTape() as tape:
            satisfaction = evaluateKB(
                self.knowledgeBase,
                self.interpretation,
                self.operatorSet,
                aggregator
            )
            loss = 1. - satisfaction
            if regulariser:
                loss += regulariser(trainables)

        #Update trainable weights.
        gradients = tape.gradient(loss, trainables)
        optimiser.apply_gradients(zip(gradients, trainables))

        #Update history
        history["loss"].append(loss)
        for i, pt in enumerate(self.knowledgeBase.parseTrees):
            history["formula_satisfaction"][i].append(
                evaluate(pt, self.interpretation, self.operatorSet))
        history["satisfaction"].append(satisfaction)

        if validationKB:
            for i, pt in enumerate(validationKB.parseTrees):
                history["validation_satisfaction"][i].append(
                    evaluate(
                        pt, self.interpretation, self.operatorSet))

        #Print messages.
        print(f"loss: {loss}, ", end=None)
        print(f"satisfaction: {satisfaction}.")

    return history
```

Listing 6.7: Model fit method implementation.

The `query` method is very straightforward and implements the algorithm given in listing 4.5. The implementation is given in listing 6.8.

```
def query(self, formula):
    if not isinstance(formula, ftl.ParseTree):
        try:
            formula = ftl.parse(formula,
                        self.knowledgeBase.signature)
        except Exception as err:
            raise ValueError(f"Could not parse formula {formula}."
                             + f"{err}")

    return evaluate(formula, self.interpretation, self.operatorSet)
```

Listing 6.8: Model query method implementation.

6.8 Operators Module

The `Operators` module is the only module within the `fuzzyops` sub-package. It contains a wide variety of fuzzy logic operators in addition to an `OperatorSet` class for grouping operators. Furthermore, the module contains methods for constructing s-norms, s-implications, and aggregators from a given t-norm. The module provides methods implementing the following t-norms and their symmetric s-norms (all s-norms are based on the strong negation $N(a) = 1 - a$).

- The Godel T-norm $T_g(a, b) = \mathsf{min}(a, b)$.
- The product T-norm $T_p(a, b) = ab$.
- The Łukasiewicz T-norm $T_l(a, b) = \mathsf{max}(0, a + b - 1)$.
- The Drastic T-norm T_d,

$$T_d(a, b) = \begin{cases} b & \text{if } a = 1, \\ a & \text{if } b = 1, \\ 0 & \text{otherwise.} \end{cases}$$

- The Nilpotent minimum T-norm T_n,

$$T_n(a, b) = \begin{cases} \mathsf{min}(a, b) & \text{if } a + b > 1, \\ 0 & \text{otherwise.} \end{cases}$$

Each t-norm and s-norm is implemented using TensorFlow operations. Additionally, the implementations accept inputs as batched tensors allowing them to compute over an arbitrary number of values. As an example, the implementation of the drastic t-norm and s-norm are given in listing 6.9.

```python
import tensorflow as tf
def tnormDrastic(a, b):
    m = tf.minimum(a, b)
    mask = tf.logical_or(tf.equal(a, 1), tf.equal(b, 1))

    return tf.where(mask, m, 0)

def tconormDrastic(a, b):
    m = tf.maximum(a, b)
    mask = tf.logical_or(tf.equal(a, 0), tf.equal(b, 0))

    return tf.where(mask, m, 1)
```

Listing 6.9: Drastic t-norm and s-norm implementation

In addition to the given t-norms and s-norms, the module provides methods for building symmetric s-norms (t-conorms), s-implications, universal aggregators, and existential aggregators from arbitrary operators. These methods are given in listing 6.10. They are based on the definitions of symmetric operators given in section 4.5.1.

```python
def tconorm(tnorm):
    return lambda a, b: strongNegation(tnorm(strongNegation(a),
                                             strongNegation(b)))

def sImplication(tconorm):
    return lambda a, c: tconorm(strongNegation(a), c)

def universalAgg(f):
    return lambda t: tf.foldl(f, t, 1.)

def existentialAgg(f):
    return lambda t: tf.foldl(f, t, 0.)
```

Listing 6.10: Operator builder functions.

There is not a builder function for r-implications given the complexity of their definition. However, the module does provide r-implication methods for each of the included t-norms. Although not strictly necessary given the `tconorm` function, the module provides s-norms for each of the included t-norms. This is because s-norms are a commonly used operation in formulas and it was therefore deemed appropriate to offer an implementation that was quicker to execute than the function returned by the builder method.

In addition to the aggregator builders, the `operators` module provides generalised mean and mean-error aggregators, as used by the Standard Product Set. Their implementation is given in listing 6.11.

```python
def generalisedMeanAgg(t, p):
    return tf.pow(tf.reduce_mean(tf.pow(t, p), keepdims=True), 1./p)

def generalisedMeanErrorAgg(t, p):
    return 1. - tf.pow(tf.reduce_mean(tf.pow(1. - t, p),
                                      keepdims=True), 1./p)
```

Listing 6.11: Generalised mean and mean-error aggregators.

6.8.1 OperatorSet Class

The `OperatorSet` class allows the grouping of fuzzy operators. `OperatorSet` objects therefore include all of the information necessary to implement the behaviour of junctors and quantifiers when evaluating formulas.

OperatorSet Class Properties

The `OperatorSet` class contains the following properties:

- `negation` - A differentiable fuzzy negation.
- `tnorm` - A differentiable fuzzy t-norm for implementing conjunction.
- `tconorm` - A differentiable fuzzy t-conorm (s-norm) for implementing disjunction.
- `implication` - A differentiable fuzzy implication (either an s-implication or r-implication).
- `universal` - A differentiable fuzzy aggregator for implementing universal quantification.
- `existential` - A differentiable fuzzy aggregator for implementing existential quantification.

OperatorSet Class Methods

The `OperatorSet` class contains only one method `__init__` which accepts the following arguments:

- `negation` - Initial value for `negation` property.
- `tnorm` - Initial value for `tnorm` property.
- `tconorm` - Initial value for `tconorm` property.
- `implication` - Initial value for `implication` property.
- `universal` - Initial value for `universal` property.
- `existential` - Initial value for `existential` property.
- `projectionEpsilon` - An optional argument specifying the value by which operator inputs should be moved away from extrema. If used, a small value (≤ 0.01) is recommended.

Some fuzzy operators (such as those based on the product t-norm) have undesirable properties when their inputs approach 0 and/or 1, such as operator vanishing, operator single-passing, and numerical underflow (see section 4.5.2). If `projectionEpsilon` is provided, then inputs passed to the operators are moved away from these values by the specified amount. These projections are only beneficial to some operator sets and therefore the argument is left as optional. Listing 6.12 presents the implementation of the `OperatorSet` class.

```python
class OperatorSet:
    def __init__(self, negation, tnorm, tconorm, implication,
             universal, existential, projectionEpsilon=0):
    if projectionEpsilon < 0:
        raise ValueError(
            f"projection epsilon must be non-negative.")

    if projectionEpsilon:
        incProj = lambda x : ((1 - projectionEpsilon) * x
                                + projectionEpsilon)
        decProj = lambda x : (1 - projectionEpsilon) * x

        self.negation = negation
        self.tnorm = lambda a, b: tnorm(incProj(a), incProj(b))
        self.tconorm = lambda a, b: tconorm(decProj(a), decProj(b))
        self.implication = lambda a, b: implication(incProj(a),
                                                    decProj(b))
        self.universal = lambda t: universal(tf.map_fn(decProj, t))
        self.existential = lambda t: existential(tf.map_fn(
                                                    incProj, t))

    else:
        self.negation = negation
        self.tnorm = tnorm
        self.tconorm = tconorm
        self.implication = implication
        self.universal = universal
        self.existential = existential
```

Listing 6.12: OperatorSet class implementation.

6.8.2 Standard Product Set

As discussed in section 4.5.3, FTL recommends the Standard Product Set as the default choice when training FTL agents. The `operators` module therefore provides an `OperatorSet` implementation of the the standard operator set, as given in listing 6.13.

```python
standardProductSet = OperatorSet(strongNegation,
                        tnormProduct,
                        tconormProduct,
                        sImplication(tconormProduct),
                        lambda t: generalisedMeanErrorAgg(t, 2.),
                        lambda t: generalisedMeanAgg(t, 1.),
                        0.01)
```

Listing 6.13: Standard Product Set implementation included in the operators module.

Chapter 7

Example Systems

In this chapter, example Fuzzy Tensor Logic systems are given to demonstrate FTL in action. Code snippets are given in each example to demonstrate how each system was implemented. Full code listings can be found within the accompanying materials for the report.

7.1 Binary Classifier

In this section, a Fuzzy Tensor Logic implementation of a binary classifier is given. A knowledge base describing the training data is given to the agent which is then used to learn the classifier as a neural predicate. The chosen task is very straightforward as this example primarily aims to demonstrate how Fuzzy Tensor Logic works in practice using the `ftlogic` package described in chapter 6.

7.1.1 Dataset

The dataset for this task was synthetically generated. Two classes were created, positive and negative, each consisting of one thousand data points. The set of positive samples was drawn from a multivariate normal distribution with a mean of $(0, 3)$ and a covariance matrix of $\left(\begin{smallmatrix} 1 & 0.5 \\ 0.5 & 1 \end{smallmatrix}\right)$. The set of negative samples was drawn from a separate multivariate normal distribution with a mean of $(3, 0)$ and the same covariance matrix. The dataset was generated in this way to ensure it was mostly linearly separable. A graph of the dataset is given in figure 7.1.

7.1.2 Model Definition

In this section, the agent is formulated in FTL.

Signature

First, we need to define a suitable signature. Doing so provides us with a set of predicates, functors, and constants with which to reason about the task. Given the simplicity of this problem, the signature only contains a single element. Namely, a predicate which performs binary classification. Let σ be a signature

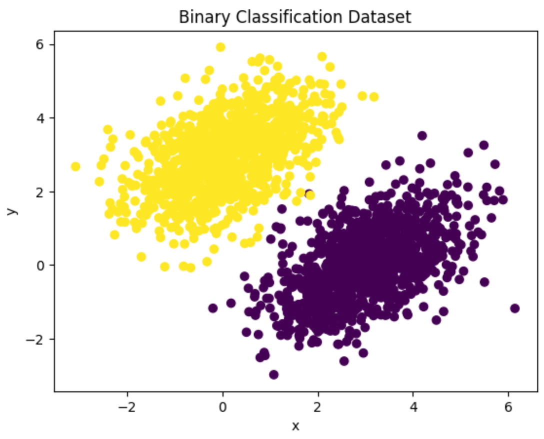


Figure 7.1: Binary classification dataset. Yellow points belong to the positive dataset whereas purple points belong to the negative dataset.

containing only a single predicate symbol Pos with ar(Pos) $= 1$. The code for defining the signature is given in listing 7.1

```
import ftlogic.core as ftl
import ftlogic.fuzzyops as fops

sig = ftl.Signature(
    predicates={"Pos": 1}
)
```

Listing 7.1: Binary classifier signature defintion.

Variables and Domains

In addition to the signature, we need variables and domains to denote elements of the data distribution. We define P to be the domain containing all positive samples in the dataset. Additionally, we define N to be the domain containing all negative samples in the dataset. Finally, we define D to be the domain containing all samples in the dataset.

Knowledge Base

Equipped with a signature and a set of domains, we are able to construct formulas about the task and collect them as a knowledge base. Let $\mathsf{KB}_\sigma = \{\psi_1, \psi_2\}$ be a knowledge base such that

- $\psi_1 = \forall x \in P : \mathsf{Pos}(x)$,
- $\psi_2 = \forall x \in N : \neg\mathsf{Pos}(x)$.

The knowledge base is very simple. It consists of two formulas which define the behaviour of the Pos predicate. ψ_1 asserts that all positive samples make Pos true, whereas ψ_2 asserts that all negative samples make Pos false. The code for defining the knowledge base is given in listing 7.2.

```
kb = ftl.KnowledgeBase(
    signature=sig,
    formulas=[
        "Ax~P:Pos(x)",
        "Ax~N:!Pos(x)",
    ]
)
```

Listing 7.2: Binary classifier knowledge base definition.

Structure

Now that we have a signature and a knowledge base, we must give a meaning to the symbols. We implement the predicate Pos as a neural network. Since the dataset consists of two linearly separable classes, we opt for a very simple architecture consisting of a single densely connected layer with a sigmoid[1] activation function. The purpose of the sigmoid activation function is to constrain the output values to $[0, 1]$, which allows the output to be treated as a fuzzy truth value. Let NN be a neural network defined by the following

$$\mathsf{NN}(x; \theta) = \mathsf{sigmoid}(xW_\theta + b_\theta)$$

where $W \in \mathbb{R}^{2\times 1}$ and $b \in \mathbb{R}$ are the network's weights as parameterised by θ.

We are now able to define a signature. Let $\mathcal{A} = (A, \xi_\theta, \Theta)$ be a σ-structure where,

- The universe A is the set of all tensors in $\mathbb{R}^2$,
- The interpretation function ξ_θ maps Pos to NN.
- The hypothesis space Θ is the hypothesis space of NN.

The code for the signature is given in listing 7.1.

```
import tensorflow.keras as keras
Pos = keras.Sequential([
    keras.Input((2,)),
    keras.layers.Dense(1, activation="sigmoid")
])

struc = ftl.Structure(
    signature=sig,
    mappings={"Pos": Pos}
)
```

Listing 7.3: Binary classifier structure defintion

Interpretation

Next, we must define what the variables and domains mean. Let $I = (\mathcal{A}, \alpha, \beta)$ be a σ-interpretation such that $\beta(P)$ is the set of positive samples, $\beta(N)$ is the set of negative samples, and $\beta(D)$ is the set of all samples. α is an empty variable assignment since there are no free variables in the knowledge base formulas. The code for the interpretation is given in listing 7.4.

[1] $\mathsf{sigmoid}(x) = \frac{1}{1+e^{-x}}$

```
interp = ftl.Interpretation(
    structure=struc,
    domainAssignment={
        "P": xPos,
        "N": xNeg,
        "D": x}
)
```

Listing 7.4: Binary classifier interpretation definition.

Model

Finally, we compile all of this data into a trainable FTL agent. Let $M = (\mathsf{KB}_\sigma, I, \mathsf{SPS})$ where SPS is the Standard Product Set set as defined in section 4.5.3. The code for compiling the model is given in figure 7.5.

```
model = ftl.Model(
    knowledgeBase=kb,
    interpretation=interp,
    operatorSet=fops.standardProductSet
)
```

Listing 7.5: Binary classifier model defintion.

7.1.3 Learning

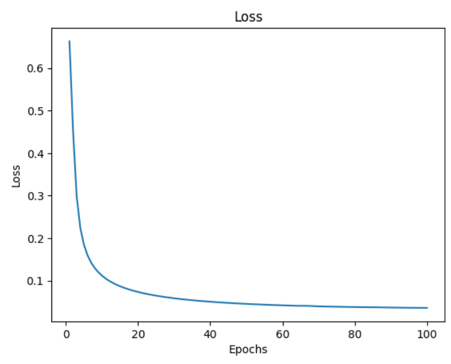
As defined in section 5.4.1, the process of agent learning is one of finding an interpretation which maximises the satisfaction of the knowledge base. In practice, this is done through a gradient-descent method which attempts to minimise an equivalent loss. In this example, learning essentially becomes the problem of training the neural network implementing Pos.

To perform learning, we use Keras' RMS back-propagation optimiser with a learning rate $\epsilon = 0.1$. Additionally we choose the Standard Product Set's universal aggregator as the satisfaction aggregator. We learn over the course of one-hundred epochs. The code which implements training is given in listing 7.6. Training graphs are given in figure 7.2.

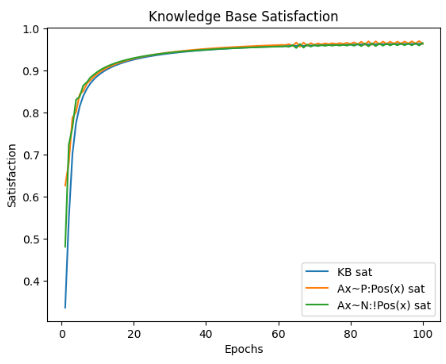
```
history = model.fit(
    epochs=100,
    optimiser=keras.optimizers.RMSprop(learning_rate=0.1),
    aggregator=fops.standardProductSet.universal,
    trainables=Pos.trainable_weights)
```

Listing 7.6: Binary classifier training call.

As figure 7.2a shows, the loss metric decreases as the epoch increases. This is exactly the behaviour one hopes to see during training as it implies that the agent is learning to perform the desired task. In figure 7.2b, one can observe the knowledge base satisfaction increasing as the epochs increase. Maximising knowledge base satisfaction is by definition the aim of learning, and this figure shows how minimising loss via gradient descent is a viable means of performing learning.



(a) Loss



(b) Knowledge base satisfaction

Figure 7.2: Training graphs

7.1.4 Results

The decision boundary learned by the agent is given in figure 7.3. The figure clearly shows that the agent learned a sensible division between the two classes.

By the end of training the agent achieved a loss of 0.0357 and a knowledge base satisfaction of 0.9643. Additionally, the agent achieved a classification accuracy of 0.999. Interestingly, it is possible for the agent to achieve near perfect classification accuracy yet have a substantially lower knowledge base satisfaction. This is due to the satisfaction aggregator being more sensitive to misclassification than the accuracy metric.

With the agent now able to distinguish between the two classes of points, it is now possible to meaningfully query the agent. One such query is the following

$$\phi_1 = \exists x \in P : \neg \mathsf{Pos}(x)$$

In other words, is there a point in the positive class that is classified as negative? The code querying the agent is shown in listing 7.7. The agent responds with a truth value of 0.0057.

```
model.query("Ex~P:!Pos(x)")
```

Listing 7.7: Querying the binary classification agent.

This query demonstrates some interesting properties of the agent. The most desirable interpretation of Pos should naturally evaluate ϕ_1 to false. By definition there are no points in the positive class which are negative. The agent's response to the query closely follows this desired behaviour. However, the decision boundary in figure 7.3 shows that the learned interpretation of Pos does indeed classify a few positive points as negative. Therefore, one would expect the agent to evaluate ϕ_1 with a much higher truth value. Despite imperfections in the learned implementation, the agent shows resilience against outliers which bolsters its ability to generalise.

The reason for this behaviour is primarily due to the choice of fuzzy operators. If one queries the agent again but instead uses the symmetric Godel operator set, the agent responds with a truth value of 0.5842. The code for this query is shown in listing 7.8.

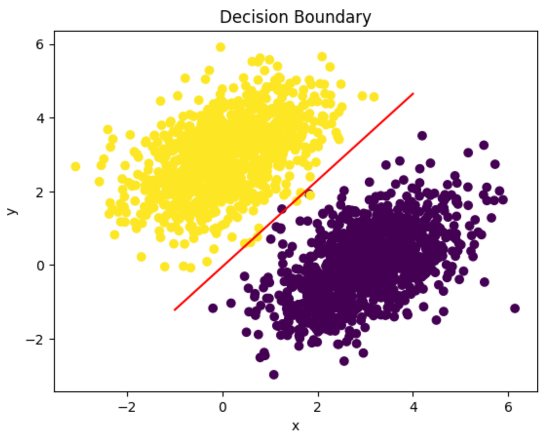


Figure 7.3: The decision boundary of NN. The red line is drawn over the points where the value of NN equals 0.5. Points above the line cause Pos to produce a truth value greater than 0.5, whereas points below produce a truth value less than 0.5.

```
model.operatorSet = fops.OperatorSet(
    fops.strongNegation,
    fops.tnormGodel,
    fops.tconormGodel,
    fops.sImplication(fops.tconormGodel),
    fops.universalAgg(fops.tnormGodel),
    fops.existentialAgg(fops.tconormGodel),
)

model.query("Ex~P:!Pos(x)")
```

Listing 7.8: Querying the binary classification agent using the Godel symmetric set.

In general, the Standard Product Set is remarkably robust against outliers during quantification while the Godel symmetric set is very sensitive. In this case, the sensitivity of the Godel symmetric set comes from its existential aggregation taking the maximum of the extended interpretation evaluations of ϕ_1. This demonstrates the importance of carefully selecting an appropriate operator set when training and querying an agent, as these decisions produce wildly different outcomes. Which outcomes are desirable depends on the intended application.

Hopefully, this example clearly demonstrates the ability of FTL agents to learn and reason. Additionally, the example hopes to have demonstrated the relative simplicity of building and using agents with `ftlogic`.

7.2 Regression

In this section, a Fuzzy Tensor Logic implementation of a regression agent is given. A knowledge base describing the training data is given to the agent which is then used to learn a neural regression model as a functor. The chosen regression task is house price prediction on the California Housing Price dataset as provided by Keras (2024). This task aims to demonstrate how FTL can be used to solve more complex problems than the previously demonstrated binary classification.

7.2.1 Dataset

The dataset for this task is the California Housing Price dataset provided by Keras (2024). This example uses the 'small' version of the dataset to demonstrate how FTL handles a limited number of training samples. The dataset consists of 600 samples. Each sample is an eight dimensional feature vector encoding the following properties:

- MedInc - median income in block group,
- HouseAge - median house age in block group,
- AveRooms - average number of rooms per household,
- AveBedrms - average number of bedrooms per household,
- Population - block group population,
- AveOccup - average number of household members,
- Latitude - block group latitude,
- Longitude- block group longitude.

Each sample has an associated scalar label, representing the house price in dollars. This example splits the dataset into train and test sets with a ratio of 0.8.

In order to enhance performance, feature-wise normalisation is performed. The feature-wise mean of the train set was subtracted from the entire dataset. Following this, the dataset was divided by the feature-wise standard deviation of the training set. The mean and standard deviation were calculated only on the training set to prevent information leaks from the test set. Furthermore, the labels were scaled to be in tens of thousands of dollars to shrink the range of regression targets.

7.2.2 Model Definition

In this section, the regression agent is formulated in FTL.

Signature

The signature σ for this agent includes a predicate Equal with $\mathsf{ar}(\mathsf{Equal}) = 2$. Intuitively, Equal maps house prices to a fuzzy truth value indicating their similarity. The signature σ also includes two functors, price and label with $\mathsf{ar}(\mathsf{price}) = \mathsf{ar}(\mathsf{label}) = 1$. Intuitively, price outputs a predicated price for a house in tens of thousands of dollars and label maps training samples to their regression label. The code defining the signature is given in listing 7.9.

```python
sig = ftl.Signature(
    predicates={"Equal": 2},
    functors={"price": 1, "label": 1}
)
```

Listing 7.9: Regression agent signature definition.

Variables and Domains

We define D to be the set of all samples, P to be the set of training samples, and Q to be the set of test samples.

Knowledge Base

We define the knowledge base for this task as $\mathsf{KB}_\sigma = \{\psi\}$ with

$$\psi = \forall x \in P : \mathsf{equal}(\mathsf{price}(x), \mathsf{label}(x))$$

The knowledge base only contains a single formula which asserts that all price predictions on training set samples by the functor `price` must match the samples' corresponding labels. The code defining the knowledge base is given in listing 7.10.

```python
kb = ftl.KnowledgeBase(
    signature=sig,
    formulas=[
        "Ax~P:Equal(price(x), label(x))"
    ],
)
```

Listing 7.10: Regression knowledge base definition.

Structure

We implement the Equal predicate symbolically. The idea behind the Equal predicate is that it should produce truer outputs the more similar its inputs. To capture this behaviour, we map Equal as follows:

$$\xi_\theta(\mathsf{Equal})(x, y) = \exp\left(-0.05\sqrt{(x-y)^2}\right)$$

Intuitively, the exponent's argument computes a scaled root mean squared error. Raising e to the negative of this value restricts the output of the function to the range $[0, 1]$ while keeping the entire expression differentiable. The implementation is written using TensorFlow operations to ensure gradients can be propagated through the predicate.

We also implement the label functor symbolically. The label functor simply returns the label associated with the given sample. In order to do this efficiently, label uses a hash table $H(x)$ implemented as a Python dictionary. label needn't be differentiable given the role it plays within the system.

Finally, we implement the price functor as a neural network $\mathsf{NN}(x;\theta)$ which accepts eight dimensional feature vectors and returns a one dimensional prediction of the house price. NN consists of three densely connected layers. The first two layers each have sixty-four units and use the rectified linear unit (ReLU) activation function. The final layer consists of a single unit with no activation function. The weights of these layers are parameterised by θ.

We now define our σ-structure $\mathcal{A} = (A, \xi_\theta, \Theta)$ to consist of the following,

- A universe A consisting of eight dimensional feature vectors and one dimensional price vectors.
- An interpretation function ξ_θ where
  - $\xi_\theta(\mathsf{Equal})(x, y) = \exp\left(-0.05\sqrt{(x-y)^2}\right)$,
  - $\xi_\theta(\mathsf{label})(x) = H(x)$,
  - $\xi_\theta(\mathsf{price})(x) = \mathsf{NN}(x; \theta)$,
- A hypothesis space Θ which is the hypothesis space of NN.

The code for the structure is given in listing 7.11.

```
def equal(x, y):
    return tf.math.exp(-0.05 * tf.math.sqrt(tf.math.square(x - y)))

labeldict = dict()
for i, sample in enumerate(x):
    labeldict[str(x[i:i+1])] = y[i:i+1]

def label(x):
    return labeldict[str(x)]

price = keras.Sequential([
    keras.Input((8,)),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(1)
])

struc = ftl.Structure(
    signature=sig,
    mappings={
        "Equal": equal,
        "price": price,
        "label": label}
)
```

Listing 7.11: Regression agent structure definition.

Interpretation

Next, we define what the variables and domains mean. Let $I = (A, \alpha, \beta)$ be a σ-interpretation such that $\beta(P)$ is the set of training samples, $\beta(Q)$ is the set of testing samples, and $\beta(D)$ is the set of all samples. α is an empty variable assignment since there are no free variables in the knowledge base formulas. The code for the interpretation is given in listing 7.12.

```
interp = ftl.Interpretation(
    structure=struc,
    variableAssignment={},
    domainAssignment={
        "D": x,
        "P": xTrain,
        "Q": xTest}
)
```

Listing 7.12: Regression agent interpretation definition.

Model

Finally we compile all of this data into a trainable FTL agent. Let $M = (\mathsf{KB}_\sigma, I, \mathsf{SPS})$ where SPS is the Standard Product Set. The code for compiling the model is given in listing 7.13.

```
model = ftl.Model(
    knowledgeBase=kb,
    interpretation=interp,
    operatorSet=fops.standardProductSet
)
```

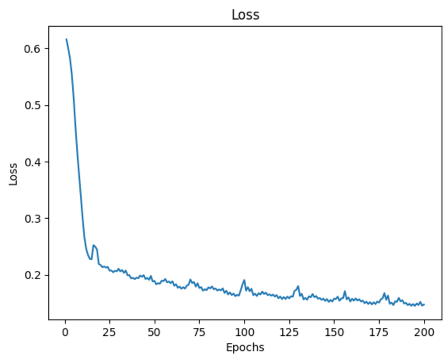
Listing 7.13: Regression model definition

7.2.3 Learning

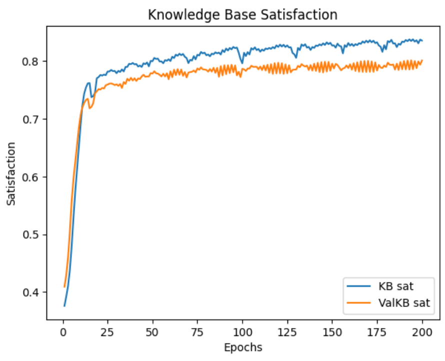
To perform learning, we again use Keras' RMS back-propagation optimiser with a learning rate $\epsilon = 0.01$. Additionally, we choose the Standard Product Set's universal aggregator as the satisfaction aggregator. In this example, we will also use an L2 regularisation function with an L2 value of 0.0001 to regularise the weights of the neural functor. This is to combat model over-fitting on the small training set. Recall that loss values are truth values and therefore we need to select a smaller than usual L2 value to ensure the regularisation output does not dominate the loss. Furthermore, we provide a validation knowledge base to track model performance on the test set during training. This is purely for demonstration purposes. The validation knowledge base contains a single formula $\psi = \forall x \in Q : \mathsf{Equal}(\mathsf{price}(x), \mathsf{label}(x))$. We learn over the course of two-hundred epochs. The code which implements training is given in listing 7.14. Training graphs are given in figure 7.4.

```
def l2reg(trainables):
    reg = keras.regularizers.L2(0.0001)
    errors = tf.Variable(tf.constant(0.))
    for weight in trainables:
        errors.assign_add(reg(weight))

    return errors

history = model.fit(
    epochs=200,
    optimiser=keras.optimizers.RMSprop(learning_rate=0.01),
    aggregator=fops.standardProductSet.universal,
    trainables=price.trainable_weights,
    regulariser=l2reg,
    validationKB = ftl.KnowledgeBase(sig, [
                                "Ax~Q:Equal(price(x), label(x))"])
)
```

Listing 7.14: Regression agent training call.



(a) Loss



(b) Knowledge base satisfaction

Figure 7.4: Training graphs

As figure 7.4a shows, the loss metric decreases as the epoch increases indicating that the agent is properly learning to perform the regression task. Furthermore, in figure 7.4b, one can observe both knowledge base satisfactions increasing as epoch increases. This demonstrates that FTL is not limited to training classifier predicates but can also train scalar valued regression functors.

7.2.4 Results

The training graphs in figure 7.4 demonstrate that the agent's ability to learn does in fact generalise to unseen samples, such as those in the test set. As training progressed, the training knowledge base satisfaction did significantly exceed the validation knowledge base satisfaction which indicates over-fitting. Inspecting the graph indicates the optimal number of epochs to be around 100, after which validation satisfaction ceases to increase by any substantial amount.

By the end of training, the agent achieved a loss of 0.1470 and a knowledge base satisfaction of 0.8353. Additionally, the agent achieved a validation knowledge base satisfaction of 0.8008. Furthermore the agent achieved a mean absolute error (MAE) of 3.5587 on the test set, indicating that the agent's predictions were off by an average of \$35,537. While this could certainly be better, the standard deviation of the test set is \$101,522.86 which is considerably higher than the MAE. Depending on the application, this agent could therefore be considered a useful estimator of house prices.

This example has also demonstrated how FTL can be used to construct basic regression agents by treating functors as neural regressors. Furthermore, the results show that FTL agents do learn to effectively generalise to unseen samples even when the available training set is small. Finally, the example has demonstrated `ftlogic`'s support for regularisation and validation metrics.

Chapter 8

Evaluation

This chapter evaluates and summarises the project and its outcomes. A direct evaluation is made of the project against its aims and objectives. Furthermore, a review of project progress is made. Moreover, implications stemming from the results of the project are discussed. Finally, project limitations are analysed and improvements are proposed.

8.1 Review of Project Objectives

The aim of this project was to design and develop methods for combining deep learning with knowledge-based reasoning to produce effective neuro-symbolic agents. In section 1.2, this aim was refined as a set of research and development objectives. Furthermore, project outcomes were defined as a set of deliverables. This section discusses the extent to which those objectives and outcomes were met.

8.1.1 Research Objectives

The first research objective was met, as demonstrated by the literature review in chapter 2. A diverse range of neuro-symbolic systems were surveyed and their comparative strengths were discussed. The discussion made reference to important themes of neuro-symbolism such as explainability and data-efficiency. During research, many other neuro-symbolic systems were identified but the decision was made to focus on evaluating several well-cited systems to attain an overview of some of the major contributions to the field. This was primarily motivated by a need to find a suitable direction for the project's neuro-symbolic method.

The novelty and complexity of existing neuro-symbolic systems made understanding them difficult and time-consuming, which in turn limited the breadth and depth of analysis offered by the literature review. There are certainly approaches to neuro-symbolism which weren't discussed. However, the research conducted was complete enough to inform the design and development of a neuro-symbolic method.

The second research objective was also met, as demonstrated by both the literature review and the successful attempt at developing what Kautz (2020) might classify as a type 5 neuro-symbolic method. The decision to develop a type 5 method was informed by research which demonstrated that combining deep learning and reasoning enabled explainable and data efficient agents, such as in Logic Tensor Networks (Serafini and Garcez, 2016) and Analysing Fuzzy Differentiable Logic Operators (Krieken, Acar, and Harmelen, 2021).

Additionally, the third research objective was met. Throughout the project report explanations of relevant mathematics and logic were given, particularly in chapters 4 and 5. The formal definition of Fuzzy Tensor Logic in chapter 5 demonstrates that the relevant mathematics were understood and applied successfully to produce a neuro-symbolic method which is rooted in formal logic and gradient-based optimisation.

Naturally, a deeper understanding of the relevant mathematics could have been attained. Perhaps this would have enabled development of more extensive querying features. A deeper understanding may have also enabled the inclusion of proofs about FTL properties, which in turn could have been used to refine FTL's design and improve the efficiency of its implementation. As it stands however, the mathematical knowledge acquired was sufficient to successfully design and implement a neuro-symbolic method.

The fourth research objective was only partially met. The classification and regression tasks demonstrated in chapter 7 are limited in their ability to demonstrate the capabilities of FTL systems. The tasks were chosen primarily out of a need for simple problems on which to test FTL during development. It proved difficult to decide on tasks and benchmarks during research due to uncertainty over method direction and a lack of standardised neuro-symbolic benchmarks. Therefore, tasks were instead chosen once the design and implementation of FTL was well underway and the hypothetical capabilities of FTL were already understood.

8.1.2 Development Objectives

The first development objective was met, however the chosen tasks required very little data collection and preprocessing. The data for the classification task was synthetically generated and did not require preprocessing. The data for the regression task was included in Keras and only required basic feature-wise normalisation.

The second development objective was also met. A neuro-symbolic method was developed following an iterative Agile-style software method. FTL was based upon extending existing ideas found in type 5 neuro-symbolic systems identified during research. As far as the student can tell, FTL is unique particularly in its adherence to and extension of formal logic principles in both design and implementation.

The viability and effectiveness of FTL agents was partially demonstrated by the example systems in chapter 7. Although chapters 4 and 5 clearly demonstrated the viability of FTL theory, it is difficult to fully justify the effectiveness of FTL without further example systems. The existing example systems have shown that FTL agents can learn and be queried in a way which combines both logical reasoning and deep learning. However, the examples failed to demonstrate some of the properties one hopes to find in neuro-symbolic agents, such as data efficiency. FTL theory suggests it is capable of providing such properties to agents, but it is yet to be shown in practice.

The third development objective was met. The `ftlogic` package was developed which clearly implements all of the components necessary for building, training, and querying FTL agents. The package was developed following an iterative Agile-style software method. The implementation closely follows the design and definition of FTL methods as given in chapters 4 and 5. By closely following the definitions, the code base proves highly readable, testable, and verifiable.

The fourth development objective was only partially met. As previously discussed, example systems were developed which demonstrated the learning and querying abilities of FTL agents, however these systems were simple and failed to demonstrate some of the proposed properties of effective neuro-symbolic agents. Unfortunately, time constraints prevented the development of more advanced FTL systems which could have better showcased these properties.

The fifth and sixth development objectives were met as evidenced by this report. The report details and justifies the research, design, development, and evaluation conducted during this project.

8.1.3 Outcomes

The proposed outcomes were mostly met. A neuro-symbolic method was developed and its effectiveness was partially demonstrated through example systems. FTL was fully implemented as a Python 3 package. Finally, this report details the research, development, and evaluation conducted throughout the course of this project.

8.2 Review of Project Progress

Included in the project brief was a Gantt chart which proposed a plan by which to carry out this project. Unfortunately, due primarily to external factors, progress was not made according to plan. A small amount of research was carried out pre-Easter break, but most work was completed from the Easter break onwards. Thankfully, the university agreed to a suitable extension which provided the project with enough time to see most of the project aims and objectives met.

External factors aside, research took considerably longer than expected. Neuro-symbolism is a relatively new field of research which has few standardised methods. Therefore, research mainly involved studying a diverse range of example systems in an attempt to evaluate key ideas and methods. There is a distinct lack of accessible resources to bridge the knowledge-gap between the undergraduate curriculum and the surprising variety of concepts discussed across neuro-symbolic research. Therefore, it took a lot of time to research and understand both existing neuro-symbolic systems and the concepts they employ.

One of the primary aims of research was to identify suitable neuro-symbolic methods to extend for the purposes of this project. Therefore, design and development were significantly delayed by the longer than expected research period. This in turn naturally shortened the time available for development and implementation.

Despite a shorter than expected amount of time available, the design and implementation of Fuzzy Tensor Logic went quite well. The relevant tools, such as

TensorFlow, proved relatively straightforward to learn given the extensive documentation available. Furthermore, designing a formal specification for FTL helped clarify ideas and significantly shortened the development time. Many of the definitions translated easily into code making implementation quite straightforward. These definitions also provided a precise set of requirements against which the implementation could be tested and verified.

The area of the project which suffered most from an overrunning research period was the development of example systems. The decision was made to prioritise the development of `ftlogic` over example systems because an incomplete and under-tested `ftlogic` package would have undermined trust in the results of the example systems. Therefore, only two FTL agents could be developed within the remaining time. While they demonstrate the basic features of FTL, they fail to properly demonstrate some of the properties of FTL which theoretically make it an effective approach to neuro-symbolism, such as data efficiency and complex neuro-symbolic reasoning. If more time was available, then it would be best spent implementing systems to demonstrate these properties.

Research and time management therefore proved to be the biggest challenges faced during this project. It seems that this would have been the case even if external factors had not impacted the progress of the project. In future projects, it may prove prudent to not approach neuro-symbolism from such a general perspective but rather identify a set of tasks early on and focus efforts on finding neuro-symbolic solutions. This was the approach most commonly observed in other neuro-symbolic research.

8.3 Implications

Although the lack of sophisticated example systems fails to demonstrate some of the proposed properties of FTL, this project still provides good grounding for further investigation of FTL and type 5 neuro-symbolic systems in general. It has been demonstrated in this report that FTL systems can learn and reason using a flexible combination of deep learning and formal logic. This in itself proves that such approaches are plausible and, given the newness of neuro-symbolism, are therefore worthy of study. The creation of the `ftlogic` package enables further research of such systems.

As demonstrated, FTL allows one to include symbolic knowledge and constraints within the deep learning process. This opens the doorway to neural networks which are trained in more data-efficient ways. This may prove helpful in addressing some of the issues around data and energy-consumption faced by current generation deep learning agents. Furthermore, training models to meet logical constraints may help in producing trustworthy agents which are both better suited to safety critical environments and resilient against biases found in their training data.

FTL also allows symbolic reasoning to take advantage of neural knowledge. Incorporating neural knowledge into the reasoning process may lessen the difficulty associated with reasoning over the broad, fuzzy, and ever changing domains agents experience in the real world. Effective learning in symbolic systems has long been a hard problem, but careful introduction of logically grounded neural elements, such as FTL's neural predicates and functors, may prove useful in solving this problem while preserving the rigour associated with reasoning systems. Perhaps most importantly, a logically grounded yet flexible combination of symbolic reasoning and neural learning may pave the way for versatile agents whose behaviours are justifiable and better understood by everyone.

The theory of FTL certainly implies that it can be used to construct agents more sophisticated than those in chapter 7. It provides a promising set of tools for constructing agents which are more explainable, data-efficient, and capable of combining learning and reasoning in a complementary way. The student is confident that further research with `ftlogic` will prove this possible.

8.4 Limitations and Improvements

The current state of FTL has limitations which can be improved upon in many ways. The most obvious limitation to FTL is the lack of sophisticated example systems which demonstrate data efficiency and complex reasoning. A natural next step in the development of FTL would be to build such systems. The student is confident this is possible with the `ftlogic` package.

There are other important limitations to FTL which were made evident during testing and the implementation of example systems. In general, the `ftlogic` package could be made more efficient. The time efficiency of the learning and querying processes could be significantly improved if the `evaluate` method was better implemented. Two solutions in particular were considered during development. Firstly, it seems possible that the system could be extended to compile the evaluation of a knowledge base into a TensorFlow compute graph, which could then be reused throughout training. This would remove a great deal of the overhead associated with the recursive structure of the `evaluate` method and its frequent use of conditionals. Furthermore, it would open the door to increased parallelism and effective batch training, both of which are standard approaches to improving efficiency within deep learning systems. Secondly, many Python libraries take advantage of Python's interoperability with C to write efficient C implementations of core components. C implementations of basic operations like fuzzy operators and the `evaluate` method could be made to further improve run-time.

Furthermore, the `parse` method could be better implemented. The current implementation works, but is not as readable as other elements of the code base. A more extensive use of regular expressions and helper functions to match formulas against node types would result in more readable and more efficient code with better error messages. Moreover, this would facilitate a more modular design which could be extended to work with a wider range of logical syntax.

Given the simple syntax of first order logic, it may be possible for the `parse` method to build signatures from example formulas, rather than rely on users writing signatures themselves. This would significantly increase the usability of `ftlogic` when building large systems.

Additionally, the current implementation of FTL agents as the `model` class is very straightforward. It lacks some of the features found in libraries like Keras such as customisable metrics, callbacks, and extensive regularisation support. Extending `Agent` in this way was considered too time consuming for this project, but it would be a helpful improvement in the future to enable a more informed and better controlled learning process.

FTL itself could be improved. The limitation of quantification to domains was necessary to make quantification computationally viable, but it also limits the expressive power of FTL in comparison to first-order languages which support quantifications over the entire universe. This is a difficult problem to solve but would likely yield results capable of expanding the representational ability of FTL formulas.

Furthermore, it may prove beneficial to extend FTL agents to support two knowledge bases, one containing general knowledge about the task and the other containing problem specific knowledge such as statements about the training data. As it stands, both forms of knowledge exist within one knowledge base which, while logically sound, can appear cluttered. This change would increase the modularity and scalability of FTL agents.

Finally, it would prove useful to implement more extensive querying options. For example, languages such as Prolog support queries which return query satisfying variable bindings. Additionally, queries about logical entailment should be considered. Given the current design of FTL, it seems difficult to see how this might be done. However, this may prove a necessary limitation to overcome if FTL is to be a viable solution for heavily logic based tasks.

Chapter 9

Conclusion

During this project, existing neuro-symbolic approaches were researched and evaluated. In response, a novel neuro-symbolic method called Fuzzy Tensor Logic was proposed which combines deep learning and knowledge-based reasoning to enable efficient logically-grounded learning and flexible fuzzy reasoning over a wide variety of domains. A formal definition of FTL was given and a Python 3 implementation `ftlogic` was developed. Example systems were developed to demonstrate basic classifier and regressor FTL agents. Finally, the completed work was evaluated against the project aims and improvements were proposed.

Many of the project objectives were fully met, with the remainder being at least partially met. More example systems are required to fully justify FTL's proposed effectiveness. This project opens the door for further exploration of FTL-style neuro-symbolic systems. The tools developed during this project are suitable to aid in such research. Hopefully, such systems will help pave the way for more explainable, adaptable, and efficient AI agents.

9.1 Future Work

A wealth of improvements to both FTL and its implementation in `ftlogic` have been proposed. A natural next step is to develop FTL systems for tasks more complex than those demonstrated in this report. Doing so will aid in developing a deeper understanding of FTL's effectiveness and limitations. Furthermore, it may prove helpful in evaluating FTL against other existing neuro-symbolic frameworks. Beyond FTL, further research into gradient-based optimisation of fuzzy logic systems may open up new and exciting areas of neuro-symbolic design.

Bibliography

Appel, Gil, Juliana Neelbauer, and David. A Schweidel (2023). *Generative AI has an Intellectual Property Problem.* URL: `https://hbr.org/2023/04/generative-ai-has-an-intellectual-property-problem` (visited on 07/28/2024).

DeepLearningAI (2023). *A Complete Guide to Natural Language Processing.* URL: `https://www.deeplearning.ai/resources/natural-language-processing/` (visited on 03/20/2024).

Dingli, Alexiei and David Farrugia (2023). *Neuro-symbolic AI : design transparent and trustworthy systems that understand the world as you do.* Birmingham, UK: Packt Publishing.

European Parliament (2024). *EU AI Act: first regulation on artificial intelligence.* URL: `https://www.europarl.europa.eu/topics/en/article/20230601STO93804/eu-ai-act-first-regulation-on-artificial-intelligence` (visited on 08/01/2024).

European Parliamentary Research Service (2024). *AI investment: EU and global indicators.* URL: `https://www.europarl.europa.eu/RegData/etudes/ATAG/2024/760392/EPRS_ATA(2024)760392_EN.pdf` (visited on 07/25/2024).

Freydenberger, Dominik D. (2020). *Logic for Computer Science: Lecture Notes.*

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2017). *Deep Learning.* Cambridge, MA, US: MIT Press.

Goodfellow, Ian, Jonathon Shlens, and Christian Szegedy (2014). *Explaining and Harnessing Adversarial Examples.* San Diego, CA. DOI: `https://doi.org/10.48550/arXiv.1412.6572`.

Halpern, Sue (2023). *What We Still Don't Know About How A.I. Is Trained.* URL: `https://www.newyorker.com/news/daily-comment/what-we-still-dont-know-about-how-ai-is-trained` (visited on 06/03/2024).

Heikkilä, Melissa (2022). *We're getting a better idea of AI's true carbon footprint.* URL: `https://www.technologyreview.com/2022/11/14/1063192/were-getting-a-better-idea-of-ais-true-carbon-footprint/` (visited on 05/20/2024).

Ipsos (2023). *Global Views On A.I. 2023.* URL: `https://www.ipsos.com/sites/default/files/ct/news/documents/2023-07/Ipsos%20Global%20AI%202023%20Report-WEB_0.pdf` (visited on 07/09/2024).

Johnson, Justin et al. (2017). *CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning.* URL: `https://arxiv.org/pdf/1612.06890` (visited on 05/06/2024).

Kahneman, Daniel (2015). *Thinking, Fast and Slow.* London, UK: Penguin.

Kautz, Henry (2020). *The Third AI Summer, Henry Kautz, AAAI 2020 Robert S. Engelmore Memorial Award Lecture.* URL: `https://www.youtube.com/watch?v=_cQITY0SPiw&ab_channel=HenryKautz` (visited on 03/20/2024).

Keras (2024). *California Housing price regression dataset.* URL: `https://keras.io/api/datasets/california_housing/` (visited on 07/20/2024).

Krieken, Emile van, Erman Acar, and Frank van Harmelen (2021). "Analyzing Differentiable Fuzzy Logic Operators". In: *Artificial Intelligence* 302. DOI: `https://doi.org/10.1016/j.artint.2021.103602`.

Lample, Guillaume and François Charton (2019). *Deep Learning for Symbolic Mathematics.* DOI: `https://doi.org/10.48550/arXiv.1912.01412`.

Lapuschkin, Sebastian et al. (2019). "Unmasking Clever Hans predictors and assessing what machines really learn". In: *Nature Communications* 10 (1). DOI: `10.1038/s41467-019-08987-4`.

Mao, Jiayuan et al. (2019). *The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision.* DOI: `https://doi.org/10.48550/arXiv.1904.12584`.

Mikolov, Tomas et al. (2013). *Efficient Estimation of Word Representations in Vector Space.* DOI: `https://doi.org/10.48550/arXiv.1301.3781`.

Mohamed, Elhassan, Konstantinos Sirlantzis, and Gareth Howells (July 2022). "A review of visualisation-as-explanation techniques for convolutional neural networks and their evaluation". In: *Displays* 73. DOI: `10.1016/j.displa.2022.102239`.

Novák, Vilém, Irina Perfilieva, and Jiří Močkoř (1999). *Mathematical Principles of Fuzzy Logic.* NY, US: Springer.

Papers With Code (2024). *Image Classification on ImageNet.* URL: `https://paperswithcode.com/sota/image-classification-on-imagenet` (visited on 08/13/2024).

Peters, Uwe (2022). "Algorithmic Political Bias in Artificial Intelligence Systems". In: *Philosophy and Technology* 35 (2). DOI: `10.1007/s13347-022-00512-8`.

Rossum, Guido van, Barry Warsaw, and Alyssa Coghlan (2013). *PEP 8 – Style Guide for Python Code.* URL: `https://peps.python.org/pep-0008/#designing-for-inheritance` (visited on 07/07/2024).

Russel, Stuart and Peter Norvig (2021). *Artificial Intelligence: A Modern Approach.* 4th ed. Hoboken, NJ, US: Pearson.

Serafini, Luciano and Artur d'Avila Garcez (2016). *Logic Tensor Networks: Deep Learning and Logical Reasoning from Data and Knowledge.* DOI: `https://doi.org/10.48550/arXiv.1606.04422`.

Silver, David et al. (Jan. 2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529. DOI: `https://doi.org/10.1038/nature16961`.

Stanford Encyclopedia of Philosophy (2015). *Many-Valued Logic.* URL: `https://plato.stanford.edu/entries/logic-manyvalued/` (visited on 04/23/2024).

Stanford University Institute for Human-Centered AI (2024). *Artificial Intelligence Index Report 2024.* URL: `https://aiindex.stanford.edu/report/` (visited on 07/20/2024).

Tidy, Joe (2024). *Character.ai: Young people turning to AI therapist bots.* URL: `https://www.bbc.co.uk/news/technology-67872693` (visited on 04/26/2024).

Voulodimos, Athanasios et al. (2018). *Deep Learning for Computer Vision: A Brief Review.* DOI: `https://doi.org/10.1155/2018/7068349`.

Wu, Tianyu et al. (2023). "A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development". In: *IEEE/CAA Journal of Automatica Sinica* 10 (5), pp. 1122–1136. DOI: 10.1109/JAS.2023.123618.