# Neural Program Lattices

## Learning through Weak Supervision

**James D. Rampersad**

Department of Engineering

University of Cambridge

This dissertation is submitted for the degree of

*MPhil in Machine Learning,*

*Speech and Language Technology*

St Edmund's College August 2017

# Declaration

I, James David Rampersad of St Edmund's College, being a candidate for the M.Phil in Machine Learning, Speech and Language Technology, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.
Total word count: 7483

Signed: James Rampersad

James D. Rampersad
August 2017

# Acknowledgements

I would firstly like to thank my two supervisors for their help throughout the research and write-up of this thesis: Professor Bill Byrne of Cambridge University and Dr Nate Kushman of Microsoft Research. I'd also like to acknowledge more generally the excellent teaching I've experienced throughout the course of my studies this year at the Cambridge University Engineering Department.

# Abstract

Learning useful abstractions from superficially complex data has been a long-standing problem in Machine Learning. The *Neural Program Lattice* [7] is an important step forwards, but still requires a few hand-crafted abstractions to facilitate learning. In this thesis we investigate the possibility of removing the remaining supervision entirely, enabling inference of latent functional hierarchies from purely elementary operations.

Our preliminary work highlights an aspect of the existing objective function that prevents learning in the absence of at least some strong supervision and we devise a new, marginal loss that facilitates training when provided with only weak supervision.

While learning can occur under this new loss, we show that the model acts essentially as a sequence to sequence learner, failing to exploit the advantages of the program hierarchy. We attribute this to the presence of a set of a dominant local minima in weight space and devise a new environment, *Bitflip* to deduce whether certain incentives can facilitate basic functional inference. Our results indicate that the NPL struggles to encapsulate behaviour without pathological constraints. We conclude with a discussion as to possible reformulations of the NPL framework to more naturally approach the inference problem.

MOVEMENTS OF LEG
PROBABLE AREA OF THE HIGHER MENTAL FACULTIES
OF ARM
OF HAND
OF FACE
OF TONGUE
AREA OF SENSATIONS OF TOUCH PAIN AND TEMPERATURE
AREA OF SIGHT
SPEECH AREA
AREA OF HEARING
AREA OF SMELL AND TASTE
CEREBELLUM
PONS
MEDULLA
SPINAL CORD

# Table of contents

**List of figures** xi

**List of tables** xv

**Nomenclature** xvii

**1 Introduction** 1
- 1.1 Motivation . . . 1
- 1.2 Hierarchies of Invariance . . . 2
  - 1.2.1 Spatial Invariance - Convolutional Filter . . . 2
  - 1.2.2 Temporal Invariance - Recurrence . . . 3
  - 1.2.3 Logical Invariance - Neural Program Interpreter . . . 3

**2 The Neural Program Lattice** 5
- 2.1 Model Framework . . . 5
- 2.2 Inference and Learning . . . 8
  - 2.2.1 Strong Supervision . . . 8
  - 2.2.2 Weak Supervision . . . 8

**3 A Marginal Loss Function** 11
- 3.1 Analysing Lattice Flow . . . 11
  - 3.1.1 Weight-Space Hypothesis . . . 11
  - 3.1.2 Visualising Probability Propagation . . . 12
- 3.2 A New, Local Objective . . . 13
  - 3.2.1 Marginal Loss over Elementary Operations . . . 14
  - 3.2.2 Marginal Loss over Sub-Sequences of Elementary Operations . . . 15
- 3.3 Nanocraft . . . 17
  - 3.3.1 Experiments . . . 17
- 3.4 Improving Training Stability . . . 21

**4 Encouraging Hierarchy Use** 23
- 4.1 Recording and Constraining Hierarchy Use . . . 23
- 4.2 Bitflip . . . 26
- 4.3 Bitflip Results . . . 27

4.3.1 Learnt Functions . . . . . . . . . . 27

**5 Related Work** **29**

**6 Discussion** **31**

6.1 Conclusion . . . . . . . . . . 31

6.2 Future Work . . . . . . . . . . 32

**References** **33**

**Appendix A Recording Hierarchy Utilisation** **35**

**Appendix B Further Training Curves** **37**

# List of figures

1.1 A simple sequential hierarchy, the lowest level discrete sequence can be efficiently represented by two levels of abstraction. Once the hierarchy is learnt, reasoning/prediction at the top level reduces complexity by a factor of 6. . . . . 1

1.2 Features learnt by convolutional filters in a facial recognition task. Image by Honglak Lee and colleagues (2011) as published in "Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks". . . . . 3

2.1 An example of one algorithmic task from Neural Programmer-Interpreters (2016) Reed and De Freitas. The controller is given a desired angle and elevation and must control the camera's azimuthal angle and height so as to achieve the correct viewpoint. In the execution trace above, `GOTO`, `HGOTO` and `LGOTO` are *functions* while `ACT` is the name given to elementary operations that change the world-state (the visual representation of the car). Each of these `ACT` calls also contain *arguments*, that specify the angle to turn or height to rise. . . . . . . . . . . . . 6

2.2 The LSTM controller, shown in context with environmental encoders $f_{enc}$ and actuators $f_{action}$, $f_{op}$, $f_{prog}$. The input and output program embeddings are labelled $g^t_{in}$and $g^t_{out}$ respectively. (This image is from Neural Program Lattices [7]) . . . . 7

2.3 The lattice of execution paths. Each point in the grid represents a grouping by elementary operation $i$ and call depth $l$. The recursive variable $y_i^{t,l}$ is computed across all these nodes sequentially, denoting the probability that at a time $t$ the controller has correctly called $i$ elementary operations and exists at a depth $l$. This image is taken from [7] . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.1 The weight-space problem, characterised by strongly attractive local minima that prevents the randomly initialised trajectories reaching the global minima. . . . . 12

3.2 A time-merged plot of $y_i^{t,l}$ probability flow through the lattice under fully supervised pre-training and the original loss function after 10,000 iterations. Colour intensity corresponds to higher values of $y_i^{t,l}$ (summed across $t$). Training appears to have converged as there is only a single path, that successfully encapsulates all its actions within functions -shown by the majority of calls originating from the second level of the hierarchy. The path `POP`s to the top level before calling a new function with `PUSH`, returning to the second level. This is the behaviour we would hope to see inferred in the absence of strong supervision. This example is taken from the *Nanocraft* domain. . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Probability flow through the lattice under the original loss function in the absence of strong supervision. Note there is no horizontal flow of $y$ at all, with all probability mass lost through incorrect action calls or `POPPING` too soon. . . . . 13
3.4 The time-merged lattice under the new, marginal loss function. Probability now propagates throughout the whole of the lattice, with latent paths making a function call at approximately the point where the nature of the task changes significantly (as will be seen in Section 3.3). This was not observed however during the greedy-decoding runs.. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.5 The time-merged lattice under the marginal sub-sequence loss. Probability flows throughout the lattice, with a portion of the latent paths calling functions and moving into the second level of the hierarchy. In greedy decoding, the controller LSTM calls only functions of a single action in length, corresponding to placing a block in the *Nanocraft* domain. As will be discussed in Section 3.3. . . . . . . . 15
3.6 a) The evolution of clipped gradient during a training run in which fully supervised samples were present. Red corresponds to the average gradient of weakly supervised samples **under the original loss function**. Blue shows the mean gradient of the strongly supervised samples (trained with the loss function from NPI [9]). Note that gradient for the weak samples is zero at first before rising steadily as more probability mass reaches the terminal node due to the guidance from full supervision. b) The clipped gradient across three runs using entirely weak supervision **with the new marginal loss function**. Note that there is now a workable gradient immediately which slowly decreases as the model fits to the training data. 16
3.7 The *Nanocraft* task. The agent receives input from from three locations, the grid world embedding (from a convolutional encoder), a natural language embedding (specifying the type of house, size and location to build) and an embedding from the currently running program. Some blocks of the house are already placed in the grid. The agent must avoid placing blocks twice in the same location so needs to monitor the world state at all times. . . . . . . . . . . . . . . . . . . . . . . . 17

3.8 The mean training curves, for 16, 32, 64, 128 and 512 weakly supervised samples. As indicated by the standard-deviation, training is highly volatile under gradient descent. There is a general trend that loss decreases most rapidly for those with the fewest numbers of samples as the network can rapidly adapt its parameters to fit the sequences. A minimum of two trials are performed in each case, with 512 and 128 repeated in greater number due to higher variation in results. Training loss here is the negative log-probability of calling the correct elementary operations independently and then calling `POP` to terminate the algorithm. . . . . . . . . . . 18
3.9 The individual training runs performed with 512 weakly supervised samples. Note the fact that curves tend to follow one of three tracks possibly a result of different valleys in weight space. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.10 The training runs for the model with 16 Samples. Note that only one sample appears to fully explain the training data. . . . . . . . . . . . . . . . . . . . . . . 19
3.11 The average progression of models trained under each number of samples through the *Nanocraft* program. There is a clear trend for the progression to increase with the number of samples, as one would expect. The anomalous result for 128 samples reflects the volatility of the training process, with none of the performed trials completely fitting to the training samples in the allowed iterations. . . . . . 20
3.12 A matrix plot of $Q_i$ values averaged over successive groups of 1000 iterations. $Q_i$ is the marginal probability of generating the $i$th elementary operation correctly, conditional on having generated the preceding $i-1$ elementary operations either correctly or incorrectly (Equation 3.3). . . . . . . . . . . . . . . . . . . . . . . . . 21
3.13 Trials using the scheduling procedure are found to be less stable - especially towards the end of training. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

4.1 When $\beta_0$ is used alone, the model avoids proper use of function calls by simply calling several zero-length functions before modelling the actions as a sequence. 24
4.2 When constrained through $\beta_0$ and $\beta_1$ the controller avoids having to learn the correct program embeddings by calling the correct number of `PUSH` and `POP` operations but in such a way as to still model the sequence as usual. . . . . . . 25
4.3 When $\beta_0$, $\beta_1$ and $\beta_2$ are used the desired result is exhibited with functions of approximately constant length being called. . . . . . . . . . . . . . . . . . . . . 25
4.4 The *world independent* Bitflip environment. To generate the synthetic data, a set of latent functions (sequences) are generated (far left). These functions are then concatenated together repeatedly in a random order and this order is shown to the NPL at the start of program execution. The simplest way to solve the task is assigning a program embedding to model each of the latent functions and then calling these programs in the order shown originally. . . . . . . . . . . . . . . . 26

4.5 *World Dependant* Bitflip the action sequence depends both on latent functions and the current state of the world. Random functions are again generated however in this case they represent *desired behaviour* rather than fixed patterns of actions. The currently running function dictates how the agent interacts with the value observed in the world grid. For example the random function - marked '0' - is generated in the bottom of this diagram and indicates that when running, the action 2 should be called when a 0 is seen and 3 when a 1 is seen. A positional marker would also be present in the world grid to indicate where the agent is in the array of bits. . . . . 27
4.6 We monitor the average number of each Function called during the greedy decoded runs performed on held-out data at the end of each epoch (1000 iterations). The controller rapidly learns to use a single function, minimising the three lattice penalties before adjusting slightly to call the second function more frequently. Training terminates when the loss decreases to zero for a set of consecutive iterations. 28

A.1 The *Nanocraft* task. An example execution trace with elementary operations shown adjacent to corresponding *strong supervision*. The hand-crafted functions would not be present under the case of purely weak supervision. . . . . . . . . . 36

B.1 Individual trials with 32 Weak training samples. . . . . . . . . . . . . . . . . . . . . 37
B.2 Individual trials with 64 Weak training samples. . . . . . . . . . . . . . . . . . . . . 38
B.3 Individual trials with 128 Weak training samples. . . . . . . . . . . . . . . . . . . . 38

# List of tables

4.1 The actions performed when Functions 1 and 2 were called, alongside their frequency during the greedily decoded held-out data. These are the final form of the functions, before training terminated (Epoch 66) corresponding to Iteration 66000 in Figure 4.6. The true generative functions were **2233** and **2323**. . . . . . 28

# Nomenclature

## Roman Symbols

| | |
|---|---|
| $w_t$ | World State |
| $y_i^{l,t}$ | Probability of generating $i$ elementary operations at time $t$ in a state at lattice depth $l$. |

## Greek Symbols

| | |
|---|---|
| $\lambda$ | Weakly supervised action sequences |
| $\pi$ | Fully supervised execution sequences |

## Terminology

Abstract Operation (Function)

A function call that does not effect the world state but consumes a program-embedding from the hierarchy, moving the agent into a different mode of operation. In the *Nanocraft* environment this could be a function to build one of the walls which itself consists of several elementary operations involving placing blocks and moving. Since these actions do not effect the world state they are unobservable.

Elementary Operation

An action taken by the controller that is observable and modifies the world state $w_t$. This could be for example placing a block in the *Nanocraft* task or moving the carry-bit in the long addition task.

Strong Supervision

A form of learning in which training data consists of full execution traces, with both elementary and abstract operations. This form of supervision is rich in that it conveys the structure of the hierarchy but harder to obtain as it is essentially hand-crafted. We denote a trace of fully supervised operations $\pi$.

Weak Supervision

A form of learning in which the training data consists purely of elementary operations. Higher level, abstract operations are omitted. These sequences are easy to obtain as they consist of only observable actions. We denote a trace of elementary operations $\lambda$.

## Acronyms / Abbreviations

| | |
|---|---|
| HMM | Hidden Markov Model |
| LSTM | Long Short Term Memory |
| MLP | Multilayer Perceptron. |
| NPI | Neural Programmer Interpreter |
| NPL | Neural Program Lattice |
| NTM | Neural Turing Machine |
| `OP` | Perform an elementary operation that effects the world state. |
| `POP` | Recurse from the currently executing program. |
| `PUSH` | Execute a new program from within the current one. |

# Chapter 1

# Introduction

## 1.1 Motivation

Everyday life necessitates the human brain both perform and learn sequences of actions that on a superficial level appear extremely noisy or complex such as walking, driving or dancing. This can be done efficiently in the *pre-frontal cortex* by decomposing sequences into a **hierarchy** of increasingly abstract patterns of motion [1]. At the lowest echelon of this hierarchy are the limb-level muscle contractions (**elementary operations**) while higher levels contain **abstractions** such as 'spin', 'turn' or 'reverse'. These abstractions ease computation in the brain as reasoning no longer need occur at the most fundamental level, reducing sample complexity (Figure 1.1). In addition, the modularity of learnt abstractions facilitates strong **generalisation** abilities as they naturally lend themselves to forming distinct new motions from the pre-existing lower-levels. Analogous compositional structure can be seen in computer programming functions, and is ubiquitous within algorithmic tasks such as sorting and long-addition.

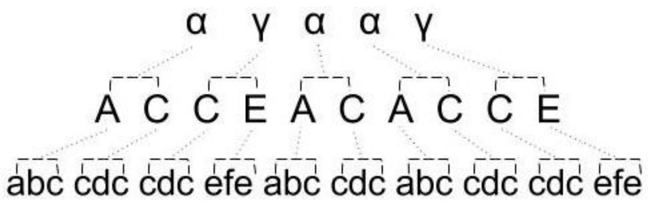


Fig. 1.1 A simple sequential hierarchy, the lowest level discrete sequence can be efficiently represented by two levels of abstraction. Once the hierarchy is learnt, reasoning/prediction at the top level reduces complexity by a factor of 6.

Recurrent Neural Networks (RNNs) do not exploit this intrinsic hierarchical structure, nor do they have the capacity to sequentially learn new tasks, suffering from *catastrophic forgetting* [5]. The *Neural Program Lattice* (NPL) offers elegant solutions to these problems by augmenting a standard LSTM with an external hierarchy of callable functions. The NPL is capable of learning algorithmic tasks from training data consisting mostly of fundamental action sequences, with a few samples containing hand-crafted functional abstractions. In this thesis, we attempt to remove the need for these hand-crafted abstractions entirely. We show that reformulation of the loss-function allows learning to occur in their absence, albeit without efficient exploitation of the program hierarchy. We relate this to the presence of a set of dominant local minima in weight-space and devise a new environment to further analyse functional inference, finding that program induction is difficult without the use of pathological constraints. Finally we conclude with a discussion of future modifications that might enable program induction more naturally.

In this Chapter we motivate the need for program induction, arguing that if Deep Learning techniques are to enjoy the same successes in the domain of *Reasoning* as they have in that of *Perception*, a robust method for inferring a basis of logically invariant features from unsupervised data is essential[1] [11].

## 1.2 Hierarchies of Invariance

Deep Neural Networks (DNNs) in the form of LSTMs and CNNs have achieved remarkable success in domains such as Speech Recognition, Computer Vision and Neural-Machine translation [4], [6], [10]. When discussing the strength of these networks, parallels are frequently drawn with the hierarchical, recurrent structure of the *neocortex* in the human brain [5]. The notion of a hierarchical model for superficially complex data has intuitive advantages and lends itself naturally to representation by a network of biological neurons. Indeed, when viewed from a hierarchical perspective, CNNs and RNNs seem analogous in that their base features remain invariant to spatial and temporal translations respectively. In the space of algorithms and logical programs, it seems natural that the corresponding ubiquitous feature take the form of a *function* - a unit of **logical invariance** with respect to changes in input, performing a fixed set of deterministic actions.

### 1.2.1 Spatial Invariance - Convolutional Filter

The use of *convolutional filters* in CNNs revolutionised the field of computer vision by allowing a single neuron to specialise in the recognition of an image feature. Sharing weights spatially, reduces total parameters and hence the amount of training data required for good generalisation. When layers of convolutions are stacked, higher level neurons share the activations of lower levels to compose more complex features (Figure 1.2).

[1]By *Perception* we refer to tasks related to *seeing* and *hearing* that Deep learning approaches have typically been very good at, while tasks relating to *Reasoning* such as logical induction or planning have not had the same success. [11]

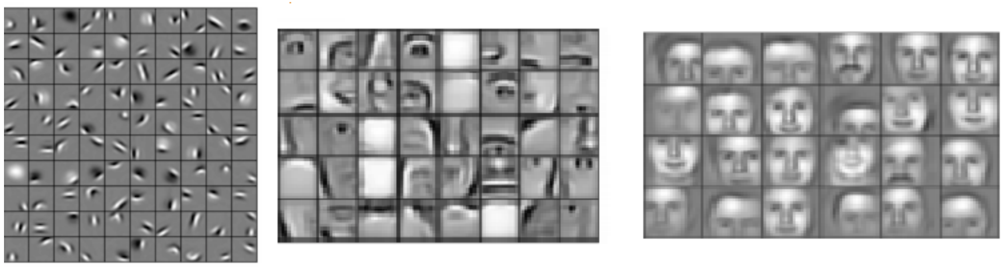


Fig. 1.2 Features learnt by convolutional filters in a facial recognition task. Image by Honglak Lee and colleagues (2011) as published in "Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks".

### 1.2.2 Temporal Invariance - Recurrence

Applications of Deep Learning to dynamical systems began to accelerate with development of the Recurrent Neural Network and Back-Propagation through Time. The recurrent property of the RNN core enables the input to hidden layer of the network to act as a temporally invariant non-linear filter with *weights shared throughout time*. In the same way as the convolutional filter this reduces parametric complexity while facilitating powerful pattern recognition abilities.

### 1.2.3 Logical Invariance - Neural Program Interpreter

It seems intuitive that Deep Learning could have similar success in *reasoning* tasks - if a robust method for analogous weight sharing between logical blocks (i.e *functions*) could be defined. These functions would encapsulate a set of logical operations, performed on some *input x* such that the *output y* may vary with *x* - but the core logical rules of the function remain invariant. From the most basic of logical operations, sophisticated reasoning processes could then be constructed, leading to the mastery of tasks requiring abilities such as causal inference or planning [11].

To illustrate this we envisage two learning cases in which a standard sequence to sequence LSTM would struggle without the capacity to identify **logical invariance**. One such task is addition where in fact it has been shown that complex LSTMs are incapable of modelling summation of numbers even with less than 10 bits in length [5].

1. **Differing Functional Arguments**: Consider sequences of the form:

   `ABACBCABACBCCBCDBDDBDABACBC`

   Naive modelling under a Deep RNN or LSTM would require significant parametric complexity to explain the sequence as a function of preceding observations. The models would likely over-fit and generalise poorly to new sequences. In contrast, a model capable of logical inference may identify function (1.1):

$$f(x) = \texttt{xBx} \tag{1.1}$$

Leading to a significantly simpler form of hypothesis in which the observation is a repeated call of $f(\mathtt{x})$ with differing arguments. Following Occam's Razor, this much simpler explanation is better justified and more likely to generalise well.

2. **Nested Function Calls**: Individually basic functions can produce extremely varied output when nested recursively. Consider the three functions:

$$f(x) = \mathtt{xBx} \qquad g(x) = \mathtt{xxx} \qquad h(x) = \mathtt{AxA} \tag{1.2}$$

Then consider two possible alternate mappings generated by different nesting orders:

$$fgh(\mathtt{C}) = \mathtt{ACAACAACABACAACAACA} \tag{1.3}$$

$$gfh(\mathtt{C}) = \mathtt{ACABACAACABACAACABACA} \tag{1.4}$$

These output sequences are entirely different, after just a single change to the left hand side of 1.3. This would require an immensely complex set of Network parameters for accurate modelling and the result would be highly inflexible when extended further e.g $fghfghfgh(\mathtt{x})$ whereas a network capable of program induction could identify three distinct components of logical invariance (functions) and model the observations as a chain of these functions.

# Chapter 2

# The Neural Program Lattice

In this Chapter we review the framework underpinning the NPL before highlighting extensions to its predecessor the Neural Program Interpreter (NPI) [9], that enable learning from weakly supervised samples. We discuss the role that strong supervision plays currently and the theoretical difficulties of removing this entirely. At its core, the NPL consists of a modified LSTM network that acts as a central controller. Ideally, we would like this controller to entirely master some form of algorithmic task having been given a small number of correct execution traces.

We discriminate between the two types of training data from which inference and program induction can occur, namely **weak** and **strong**. The NPI is only able to learn from strong supervision, in which execution traces contain hand-crafted abstract functions. Modifications introduced by the NPL allow learning from mostly weak supervision but still require a few samples to be strongly supervised.

## 2.1 Model Framework

The central LSTM controller dictates the course of action taken by the model. At any given time $t$, the controller interacts with its environment known as the *world state* $w_t$ by taking one of three actions: it can perform an **elementary operation** - directly affecting the *world state*, call a new program or return from its currently running program. The latter two choices have no direct effect on the *world* but change how the model will interact in future. Figure 2.1 shows an example in which $w_t$ takes the form of an image embedding with elementary operations consisting of rotation and elevation changes.

Calling a function feeds an input vector consisting of the program and argument embeddings $p_g^t$ as input to the LSTM controller in the subsequent time-step. The hidden-state of the recurrent controller is wiped to zero and saved to a stack. While the program is running, the controller could perform a series of elementary operations, or decide to descend further into the hierarchy by calling another program. When the model decides to terminate the program by calling `POP`, the hidden state is entirely reset to its state before the particular program was called. The parent program embedding is similarly reset. Resetting these components to the state prior to program execution is a key part of functional encapsulation, as the central controller maintains no memory of the actions that took place within the function but observes only an updated *world state*.

This framework permits very precise and hierarchical sequences of computation to be performed, as the controller only need learn actions dependant on the current program $p_g^t$ rather than potentially very long dependencies between observations. Programs are returned from and called according to a probability distribution $p_a^t$ over the three possible decisions `PUSH`, `POP` and `OP` summarised below. The decision is made based on three inputs to the controller at time $t$: an observation of the world state $w_t$, the current program embedding $p_g^t$ and any argument embeddings produced at the previous time-step[1].

1. `PUSH`. Execute a program. The relevant program embedding is fed as input to the controller in the next time-step and the hidden units of the LSTM are **pushed** onto the state stack $S$ while being wiped to zero in the controller itself.

2. `POP`. Return from the currently executing program. Resets the controller's hidden state to before calling the program by **popping** from the state's stack $S$. If at the top of the hierarchy, this becomes the probability of terminating the currently running algorithm.

3. `OP`. Perform an **elementary operation**. Affects and updates the current world state $w_t$, for example performing a 'swap' operation in a sorting algorithm or placing a block in '*Nanocraft*'.

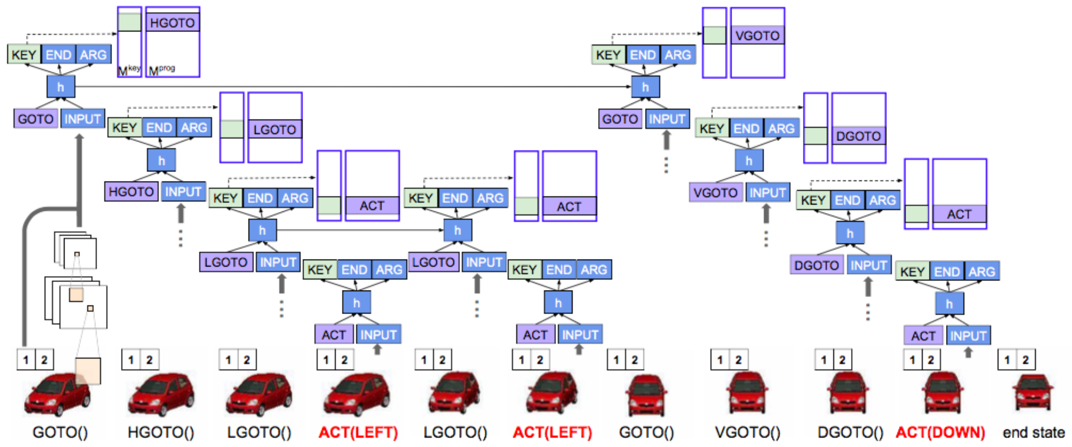


Fig. 2.1 An example of one algorithmic task from Neural Programmer-Interpreters (2016) Reed and De Freitas. The controller is given a desired angle and elevation and must control the camera's azimuthal angle and height so as to achieve the correct viewpoint. In the execution trace above, `GOTO`, `HGOTO` and `LGOTO` are *functions* while `ACT` is the name given to elementary operations that change the world-state (the visual representation of the car). Each of these `ACT` calls also contain *arguments*, that specify the angle to turn or height to rise.

[1] As well as any information that might be contained in the controller LSTM's hidden state.

### Stacks

Two stacks are maintained to record POPPED and PUSHED model states and programs. Whenever a program is called by the controller, its embedding and arguments are pushed onto the $S$ stack. Whenever POP is called, the top program is popped from the stack to become the currently running program. In a similar manner, the controller's internal states are pushed or popped to the stack $M$ such that the top of the stack is the current state of the controller.

### Learnable Encoders

The NPL has three learnable components: the parameters of the central LSTM controller, a hash-table of abstract program embeddings and a set of domain specific encoders. These encoders provide an interface between the controller and its operational environment, allowing the LSTM to remain a stage detached, facilitating operation between domains. Encoders are used for both input and output as illustrated in Figure 2.2:

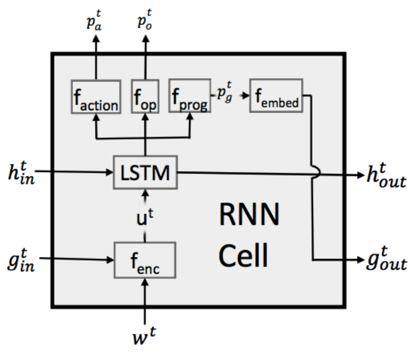


Fig. 2.2 The LSTM controller, shown in context with environmental encoders $f_{enc}$ and actuators $f_{action}$, $f_{op}$, $f_{prog}$. The input and output program embeddings are labelled $g_{in}^t$ and $g_{out}^t$ respectively. (This image is from Neural Program Lattices [7])

.

1. **Input Encoder** $u_t = f_{enc}(g_{in}^t + w^t)$
   The current program embedding $g_{in}^t$ and world state $w^t$ are concatenated and mapped through an encoder $f_{enc}$ that takes the form of a two layer Multilayer Perceptron (MLP).

2. **Action Decoder** $P_a^t = f_{act}(h_{out}^t)$
   The output of the LSTM is mapped onto a one-hot vector pertaining to the probabilities of PUSH, POP and OP.

3. **Elementary Operation Decoder** $P_o^t = f_{op}(h_{out}^t)$
   The decoder produces a distribution over the possible elementary operations, during greedy decoding this would only be used assuming the *action decoder* above calls OP.

4. **Program Embeddings** $g_{out}^t = f_{embed}(p_g^t)$
   The decoder $f_{prog}$ outputs a distribution over programs, $p_g^t$ which are encoded by the $f_{embed}$ network to produce program embeddings $g_{out}^t$ for input to the LSTM in the next time step.

## 2.2 Inference and Learning

### 2.2.1 Strong Supervision

**Strongly** supervised data contains the full program execution trace - including the identity of abstract programs. This data is very rich as it contains the optimal program hierarchy, but difficult to obtain in practice as it requires the labelling of abstract concepts that may not be easy to identify (depending on the domain). Currently, the NPL uses mostly weak with a small amount of strong training data. This strong supervision primes the NPL controller to use the optimal hierarchy. Weak supervision can then be used for tuning program embeddings and improving generalisation. The model does not, however perform program inference itself, as the strong supervision entirely conveys this information [7]. Learning from the full execution trace $\pi$ therefore, is relatively straight-forward with the objective to find Neural Network parameters $\theta$ that maximise the log-likelihood of the true sequence (Eq. 2.1). Gradients can be obtained by differentiating the log likelihood (Eq.2.3) of the true execution sequences with respect to the model parameters.

$$p(\pi^t) = [\![\pi_a^t = \texttt{OP}]\!] p_a^t(\texttt{OP}) p_o^t(\pi_o^t) + [\![\pi_a^t = \texttt{PUSH}]\!] p_a^t(\texttt{PUSH}) p_g^t(\pi_g^t) + [\![\pi_a^t = \texttt{POP}]\!] p_a^t(\texttt{POP}) \tag{2.1}$$

$$p(\pi) = \prod_{t=1}^{T} p(\pi^t) \tag{2.2}$$

$$\mathscr{L}(\theta) = log(p(\pi)) \tag{2.3}$$

### 2.2.2 Weak Supervision

**Weak** supervision is naturally abundant and easy to obtain, pertaining to sequences of **elementary operations** only. Ideally, we would marginalise over all latent, full sequences $\pi$ that could have generated the observable actions $\lambda$ and then maximise the corresponding likelihood [7]. If we define, $\mathscr{O}$ as the many to one mapping from the elementary operations to full execution traces, the desired log-likelihood becomes:

$$\mathscr{L}(\theta) = log \sum_{\pi \in \mathscr{O}_\lambda^{-1}} p(\pi) \tag{2.4}$$

This however, is computationally intractable as the number of latent sequences that could have generated the observed actions grows exponentially with the maximum length of $\pi$ considered. This becomes intuitively clear if the execution trace is imagined as a graphical tree. Branches form upon each decision between executable programs. Calculation of the true log-likelihood means keeping track of program and hidden stacks at every branch point and calculating the marginal likelihood of *every* path through the graph.

In the analogous work of [5], the authors overcome this intractability by approximating a merged state of all possible actions and paths. Stacks and program embeddings are averaged with a weighting given by the probability of occurrence $p_a^t$. This produces a tractable and differentiable feed-forward process to obtain an approximation to the true marginal log-likelihood. While computationally efficient - foregoing the exponential explosion in paths for a single one in the branching analogy - the obtained gradients make training difficult as highlighted in [2].

The Neural Program Lattice introduces a technique to more closely approximate the marginal likelihood by constructing a **lattice**. Rather than keeping track of every path or just a single one with a fuzzy approximation, the NPL groups possible states at each time step by **elementary operation index** $i$ and **hierarchy depth** $l$ creating a grid of merged states that are propagated through time to form a lattice (Figure 2.3). This intermediate case between the two extremes increases accuracy at the cost of computational efficiency. The training objective can be defined from the lattice, seeking to maximise the probability of paths that correctly generate all the elementary operations and return from the top level. Gradients can then be obtained from this approximation to the log-likelihood and used in parameter updates.

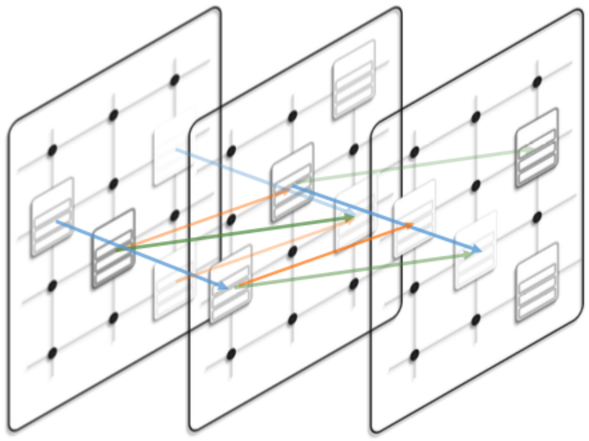


Fig. 2.3 The lattice of execution paths. Each point in the grid represents a grouping by elementary operation $i$ and call depth $l$. The recursive variable $y_i^{t,l}$ is computed across all these nodes sequentially, denoting the probability that at a time $t$ the controller has correctly called $i$ elementary operations and exists at a depth $l$. This image is taken from [7]

.

To weight paths according to their likelihood, Li et al. keep track of the recursive quantity $y_i^{t,l}$ , the probability that at time $t$, depth $l$ the controller has correctly produced elementary operations up to index $i$. A value of $y$ is created at every point in the lattice, alongside the merged state representations.

$$y_i^{t+1,l} = p(\texttt{OP})_{a,i-1}^{t,l} p_{o,i-1}^{t,l}(\lambda_o^i) y_{i-1}^{t,l} + p(\texttt{POP})_{a,i}^{t,l+1} y_i^{t,l+1} + p(\texttt{PUSH})_{a,i}^{t,l-1} y_i^{t,l-1} \tag{2.5}$$

Soft-stack updates to the state-stack $S$ and program-stack $M$ can then be defined as follows. Note the weightings $\alpha$ given to contributions from PUSH, POP and OP depend both on the type of action and magnitudes of current and proceeding $y$.

$$M_{d,i}^{t+1,l} = \begin{cases} \alpha_{i,i}^{t,l+1,l}(\texttt{POP})M_{1,i}^{t,l+1} + \alpha_{i-1,i}^{t,l,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)h_{out,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})0 & d=0 \\ \alpha_{i,i}^{t,l+1,l}(\texttt{POP})M_{d+1,i}^{t,l+1} + \alpha_{i-1,i}^{t,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)M_{d,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})M_{d-1,i}^{t,l-1} & d>0 \end{cases}$$

$$S_{d,i}^{t+1,l} = \begin{cases} \alpha_{i,i}^{t,l+1,l}(\texttt{POP})S_{1,i}^{t,l+1} + \alpha_{i-1,i}^{t,l,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)S_{0,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})g_{out,i}^{t,l-1} & d=0 \\ \alpha_{i,i}^{t,l+1,l}(\texttt{POP})S_{d+1,i}^{t,l+1} + \alpha_{i-1,i}^{t,l,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)S_{d,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})S_{d-1,i}^{t,l-1} & d>0 \end{cases}$$

With $\alpha$ defined for notational brevity:

$$\alpha_{i_1,i_2}^{t,l_1,l_2}(a) = \frac{y_{i_1}^{t,l_1}}{y_{i_2}^{t+1,l_2}} \times p_{a,i_1}^{t,l_1}(a)$$

Finally, since the programs have no set number of time-steps $t$ to finish in, marginalise over all possible lengths up to a limit $T$ (Eq.2.6). The final marginal log-likelihood maximises the probability of paths returning from the top level of the program lattice having correctly generated all elementary operations up to $I$.

$$\mathscr{L}(\theta) = log(\sum_{t<T} p_{a,I}^{t,0}(\texttt{POP})y_I^{t,0}) \tag{2.6}$$

# Chapter 3

# A Marginal Loss Function

Our preliminary analysis focusses on developing a thorough understanding of why the NPL fails to learn under weak supervision alone. An initial hypothesis attributed this failure to insufficient use of the program hierarchy, due to a strong trajectory attractor within weight space. Visualisations of lattice flow reveal however, that the foremost reason is in fact a dissipating gradient caused by the form of loss function. We devise a new, marginal objective and perform a set of experiments with purely weak supervision in the *Nanocraft* domain.

## 3.1 Analysing Lattice Flow

### 3.1.1 Weight-Space Hypothesis

In *Nanocraft* and *Addition* domains, learning to encapsulate repeated behaviour within function-calls is not just advantageous but essential to mastery of the task given only a handful of training samples. The challenge of program induction directly from elementary operations however is particularly difficult given the vast numbers of possible generative hierarchies. Indeed, learning to use programs effectively necessitates a period in which the programs are called but are unhelpful, lacking the proper form.

We therefore postulate that the weight space consists of two minima:

1. The most dominant corresponds to the region of weight-space in which the LSTM **models the action sequence directly**. This is a strong trajectory attractor where loss decreases rapidly, but fails to fully master the task as no function calls are made.

2. The **global minima** is more elusive, occupying a region of weight-space in which configurations of LSTM parameters have **learnt to call the functions effectively** and encapsulate behaviour. Since there are potentially more than one effective hierarchy, there could be multiple similar minima.

The space residing between these minima has a high loss, where the NPL simultaneously is learning when to call functions and what their form should be. We theorise that the random initialisation of LSTM parameters lies within the support basin of the strongly dominant local minima, preventing the functions from being learnt under weak supervision alone. Fully supervised pre-training, primes the NPL to learn from weak supervision by moving the LSTM to a point that lies within the support-basin of the global minima (See Figure 3.1 for a one-dimensional analogy).

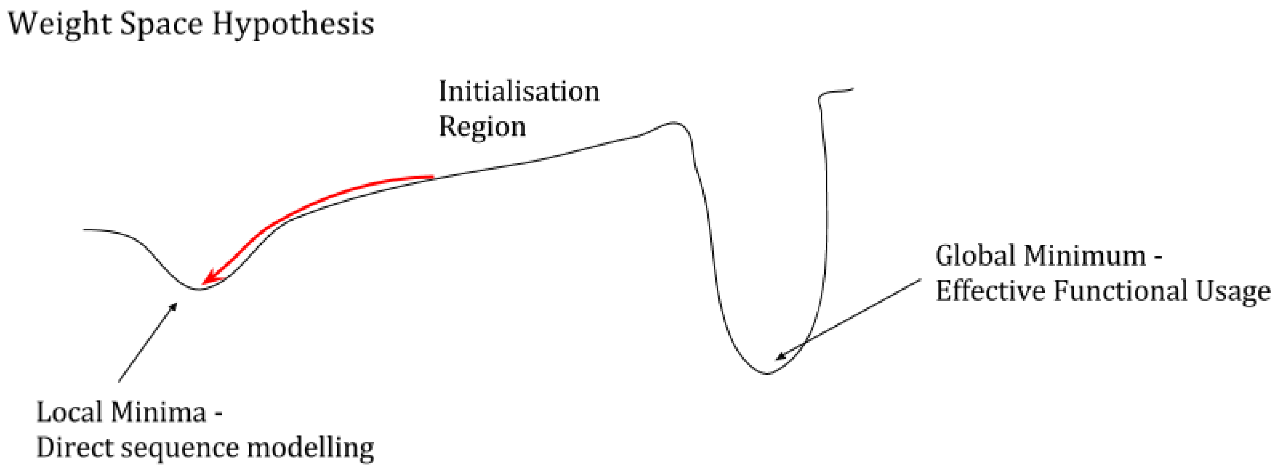

Weight Space Hypothesis
Initialisation Region
Global Minimum - Effective Functional Usage
Local Minima - Direct sequence modelling

Fig. 3.1 The weight-space problem, characterised by strongly attractive local minima that prevents the randomly initialised trajectories reaching the global minima.

### 3.1.2 Visualising Probability Propagation

To confirm or invalidate our belief we create animations of probability flow in the lattice by generating a matrix plot of $y_i^{t,l}$ yielding a frame for each time $t$. The recursive variable $y_i^{t,l}$ quantifies the likelihood at each time step that the sum of latent execution paths has performed $i$ elementary operations correctly and is currently at depth $l$ within the lattice. To convert our animations to an image, we sum along the time index, leading to Figure 3.2, in the presence of strong supervision. The probability flow is monitored at different points in training, both for trials using strong supervision and without. As expected in the fully supervised case, the probability flows through the lattice from left to right, dropping down into the second level as functions are called. In the absence of this strong supervision however, it is found that probability does not propagate further than the first elementary operation at any point in training, with only a sequence of `POP` or `PUSH` operations observed (Figure 3.3). This is unexpected - as our weight-space hypothesis would suggest that probability mass flows through the lattice horizontally, but simply fails to call functions.

#### Flaws of the existing loss function

The observed phenomena has its origins in the loss function. The objective maximises the marginal probability of paths through the lattice that correctly perform all $I$ elementary operations before calling `POP` from the top level of the lattice. However, in order for this loss to be finite, a non-negligible degree of probability mass, $y_i^{t,l}$ must reach the terminal node and subsequently call `POP`. Due to random weight initialisation, the first training batch is extremely volatile with probability mass lost from paths that POP too early, call the wrong functions or PUSH too deeply into the hierarchy. The result is that no measurable probability mass reaches the terminal node index $I$ yielding an infinite loss and a meaningless negligible gradient. This issue is notably absent in the presence of even a small amount of strong supervision as the pre-training biases the controller into a behaviour that is able to generate correct actions under subsequent weak supervision.

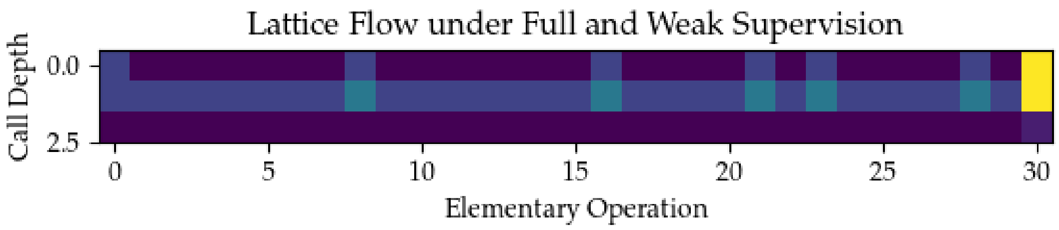


Fig. 3.2 A time-merged plot of $y_i^{t,l}$ probability flow through the lattice under fully supervised pre-training and the original loss function after 10,000 iterations. Colour intensity corresponds to higher values of $y_i^{t,l}$ (summed across $t$). Training appears to have converged as there is only a single path, that successfully encapsulates all its actions within functions -shown by the majority of calls originating from the second level of the hierarchy. The path POPs to the top level before calling a new function with PUSH, returning to the second level. This is the behaviour we would hope to see inferred in the absence of strong supervision. This example is taken from the *Nanocraft* domain.

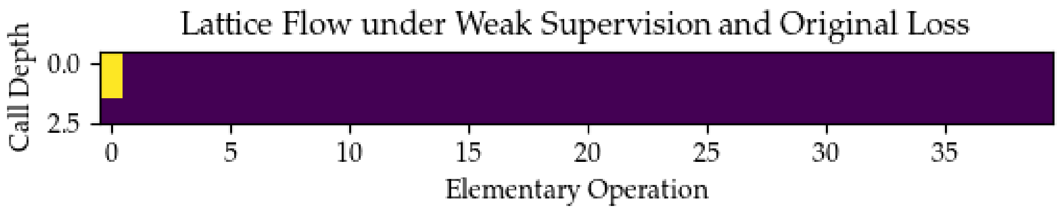


Fig. 3.3 Probability flow through the lattice under the original loss function in the absence of strong supervision. Note there is no horizontal flow of $y$ at all, with all probability mass lost through incorrect action calls or POPPING too soon.

## 3.2 A New, Local Objective

Evidently, the initialisation problem necessitates a *local* form of loss to remain valid in the case of partially or even entirely incomplete action sequences. In deciding between candidate functions it is crucial that they:

1. Produce finite loss and usable gradients immediately, regardless as to the degree of probability mass reaching the terminal node.
2. Encourage POP actions once all $I$ of the elementary operations have been called.

Two alternatives to maximising the probability of an entire sequence - as done originally - would be to marginalise over elementary operations $i$ or variable length sub-sequences beginning from the initial index $i = 0$. These are investigated in turn, both producing workable gradients, with the former leading to more stable training.

### 3.2.1 Marginal Loss over Elementary Operations

The log likelihood can be written in marginal form:

$$log(P(\lambda_0, \lambda_1 ... \lambda_I)) = \sum_{i=0}^{I} log(P(\lambda_i)) \tag{3.1}$$

To facilitate computation of this marginal loss, we redefine the recursive variable $y_i^{t,l}$ as the probability of generating $i$ elementary operations either correctly or incorrectly (Eq. 3.2). Intuitively, this means that probability mass will flow through the lattice regardless as to whether or not the correct operation is called (as long as some operation 'OP' is called at all). See Figure 3.4

$$y_i^{t+1,l} = p(\texttt{OP})_{a,i-1}^{t,l} y_{i-1}^{t,l} + p(\texttt{POP})_{a,i}^{t,l+1} y_i^{t,l+1} + p(\texttt{PUSH})_{a,i}^{t,l-1} y_i^{t,l-1} \tag{3.2}$$

We define a variable $Q_i$, based on this new definition of $y_i^{t,l}$ as the probability of generating the $i$th elementary operation correctly, conditional on having generated the preceding $i-1$ elementary operations (either correctly or incorrectly). Philosophically, the lattice now learns to maximise the probability of each action independently, rather than the joint probability of the entire sequence. To encourage POP operations at the correct point in the lattice the original form of loss function is also included in the product for the final likelihood term. We refer to this reformulation as the marginal loss over elementary operations.

$$Q_i = \sum_{l,t} y_{i,l}^{t} * p_{a,i}^{t,l}(\texttt{OP}) * p_i^{t,l}(\lambda_o) \tag{3.3}$$

$$\mathscr{L} = log \prod_i Q_i + log(\sum_{t<T} p_{a,I}^{t,0}(\texttt{POP}) y_I^{t,0}) \tag{3.4}$$

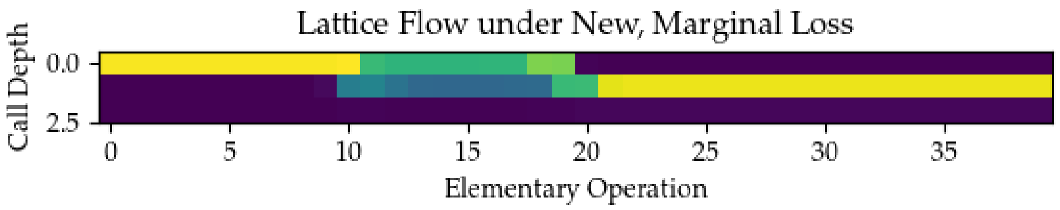


Fig. 3.4 The time-merged lattice under the new, marginal loss function. Probability now propagates throughout the whole of the lattice, with latent paths making a function call at approximately the point where the nature of the task changes significantly (as will be seen in Section 3.3). This was not observed however during the greedy-decoding runs..

### 3.2.2 Marginal Loss over Sub-Sequences of Elementary Operations

An alternate formulation of the loss is to marginalise over sub-sequences of correct elementary operations:

$$\sum_{j=0}^{I} log(P(\lambda_0 ... \lambda_j)) \tag{3.5}$$

Maximising the probability of longer chains of operations more closely reflects the original loss function. At test time a zero-one loss is used, meaning sequences of operations need be entirely correct to receive zero loss. In this formulation $y$ is left as before, the probability of *correctly* generating the preceding $i$ elementary operations (Equation 3.6).

$$y_i^{t+1,l} = p(\mathtt{OP})_{a,i-1}^{t,l} p_{o,i-1}^{t,l}(\lambda_o) y_{i-1}^{t,l} + p(\mathtt{POP})_{a,i}^{t,l+1} y_i^{t,l+1} + p(\mathtt{PUSH})_{a,i}^{t,l-1} y_i^{t,l-1} \tag{3.6}$$

The variable $Q_i$ remains identical to Equation 3.3 albeit with the original form of $y$. We find that this form of loss leads to relatively unstable training curves, perhaps due to the over representation of initial elementary operations, present in each marginal term. Visualising lattice flow reveals that some functions of a single action are used, with the remainder called from the top level of the hierarchy (Figure 3.5).

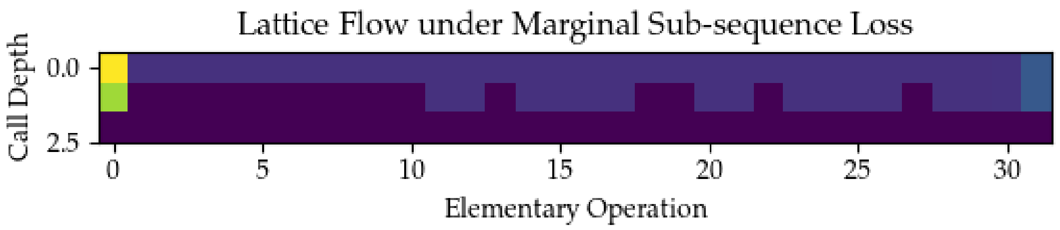
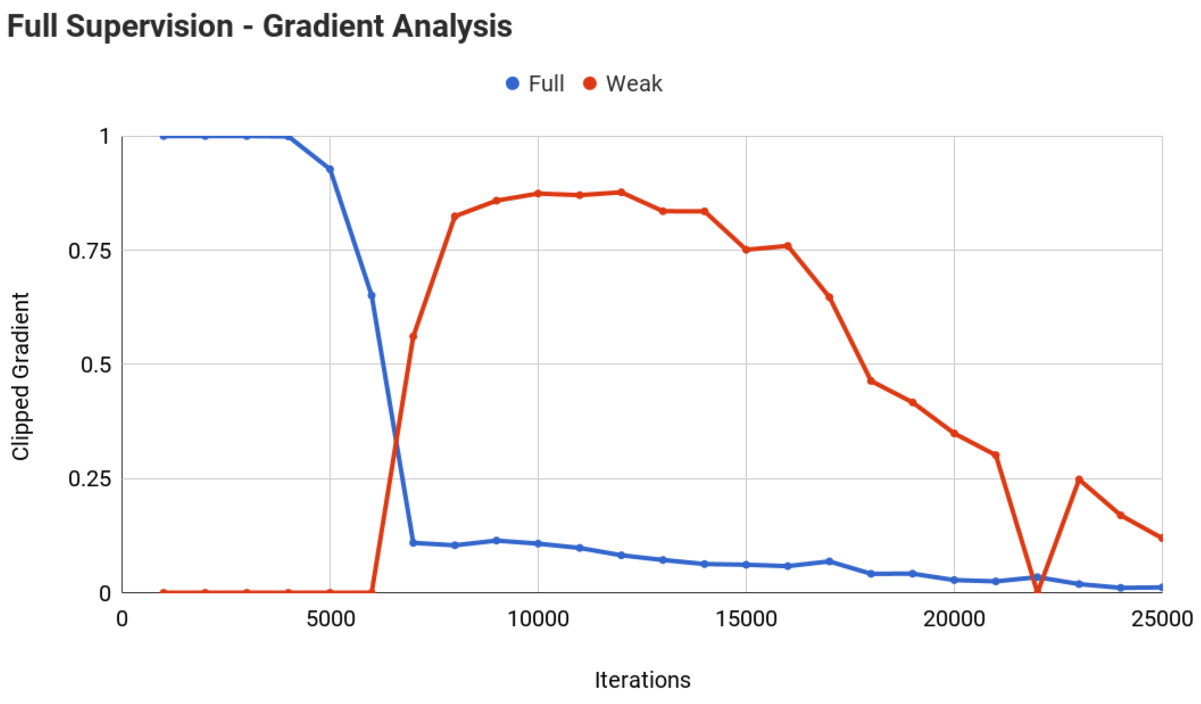


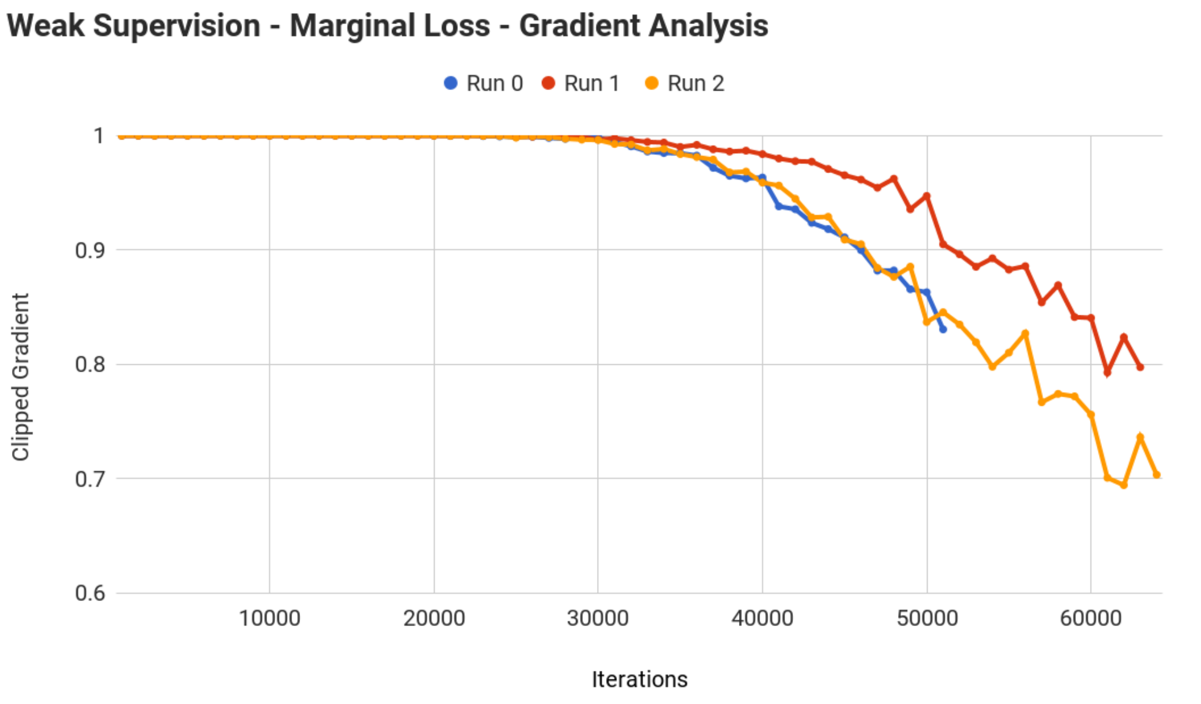
Fig. 3.5 The time-merged lattice under the marginal sub-sequence loss. Probability flows throughout the lattice, with a portion of the latent paths calling functions and moving into the second level of the hierarchy. In greedy decoding, the controller LSTM calls only functions of a single action in length, corresponding to placing a block in the *Nanocraft* domain. As will be discussed in Section 3.3.

#### A Workable Gradient

Experiments from this point forwards are carried out using the marginal loss over elementary operations - defined with $y_i^{t,l}$ of the form in Equation 3.2. When trained under this new loss, probability is able to flow immediately through the lattice, with a workable gradient absent from initialisation problems of the original loss (Figure 3.6a).



(a)



(b)

Fig. 3.6 a) The evolution of clipped gradient during a training run in which fully supervised samples were present. Red corresponds to the average gradient of weakly supervised samples **under the original loss function**. Blue shows the mean gradient of the strongly supervised samples (trained with the loss function from NPI [9]). Note that gradient for the weak samples is zero at first before rising steadily as more probability mass reaches the terminal node due to the guidance from full supervision. b) The clipped gradient across three runs using entirely weak supervision **with the new marginal loss function**. Note that there is now a workable gradient immediately which slowly decreases as the model fits to the training data.

## 3.3 Nanocraft

*Nanocraft* is an algorithmic toy task in which an agent must navigate through a 16x16 size grid-world in order to construct a rectangular 'house' of specified dimension, coordinates and material. The house has some pre-existing blocks but the agent must place the majority by executing a sequence of deterministic elementary operations: `MOVE` and `PLACE` (Figure 3.7). Each action requires arguments for specific directions or types of materials. An example execution trace, showing the abstract functions in bold is shown in the appendix (Figure A.1). Note that the presence of these abstract functions, such as `BUILDWALL` or `MOVEMANY` are not provided during training with weak supervision.

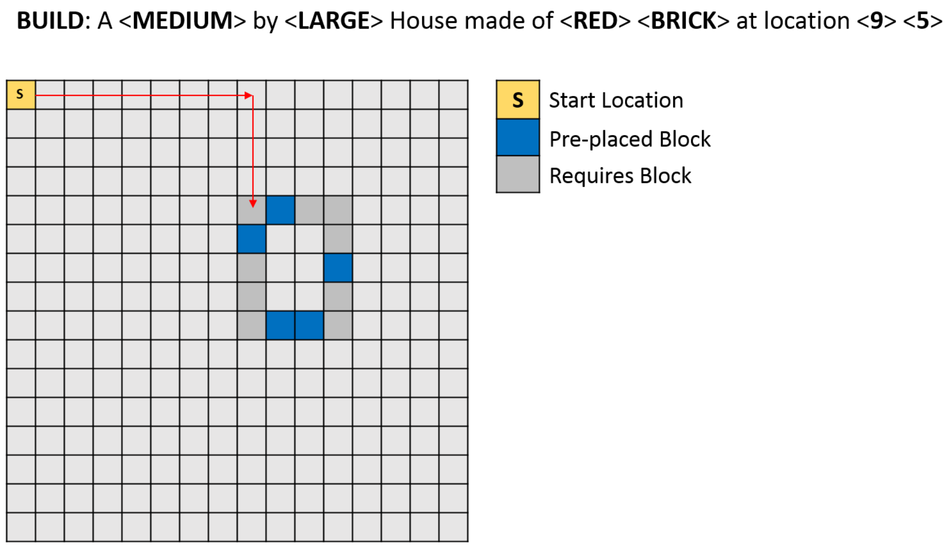

**BUILD**: A <**MEDIUM**> by <**LARGE**> House made of <**RED**> <**BRICK**> at location <**9**> <**5**>
S
S Start Location
Pre-placed Block
Requires Block

Fig. 3.7 The *Nanocraft* task. The agent receives input from from three locations, the grid world embedding (from a convolutional encoder), a natural language embedding (specifying the type of house, size and location to build) and an embedding from the currently running program. Some blocks of the house are already placed in the grid. The agent must avoid placing blocks twice in the same location so needs to monitor the world state at all times.

### 3.3.1 Experiments

We repeat the experiments performed by Li et al. in [7], without any strong supervision. Despite the new marginal objective function - and decrease in training loss - it is found that held-out zero-one test loss does not decrease significantly or comparably with the results reported by Li et al.

#### Results

Marginal training loss is monitored alongside zero-one test loss, computed by performing a greedy decoding run on a batch of 500 held-out test samples and taking the mean of the binary scores. We find that training runs are volatile with differently seeded initialisations showing different convergence speeds and results (Figure 3.8). Curves are often grouped into two or three clusters where loss decreased slowly. Occasionally one of these curves will break from its group, with loss decreasing rapidly until it either reaches the group below or successfully models the training data (registering near zero loss). Setting a limit on iteration number can therefore be difficult as one does not know if permitting training for slightly longer will lead to a sudden decrease in loss. We theorise that the presence of this phenomena is due to valleys in weight space or perhaps the segmented nature of the Nanocraft task, which consists of distinct sections of behaviour.

### Training Curves

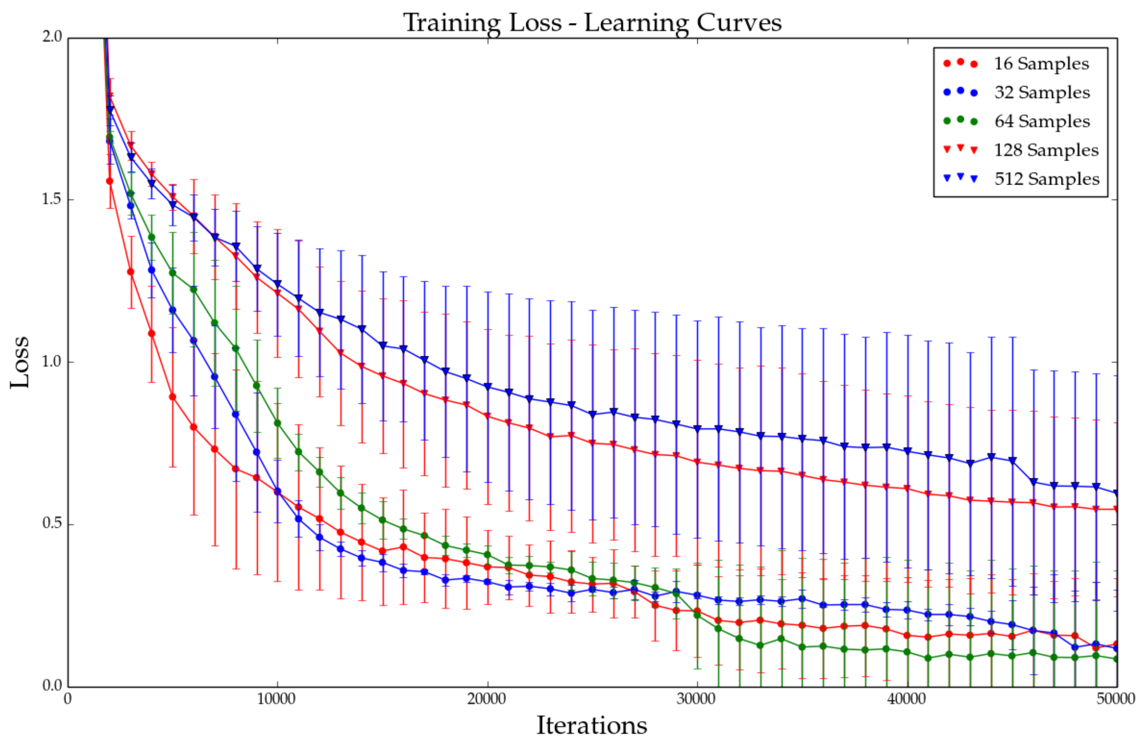


Fig. 3.8 The mean training curves, for 16, 32, 64, 128 and 512 weakly supervised samples. As indicated by the standard-deviation, training is highly volatile under gradient descent. There is a general trend that loss decreases most rapidly for those with the fewest numbers of samples as the network can rapidly adapt its parameters to fit the sequences. A minimum of two trials are performed in each case, with 512 and 128 repeated in greater number due to higher variation in results. Training loss here is the negative log-probability of calling the correct elementary operations independently and then calling `POP` to terminate the algorithm.

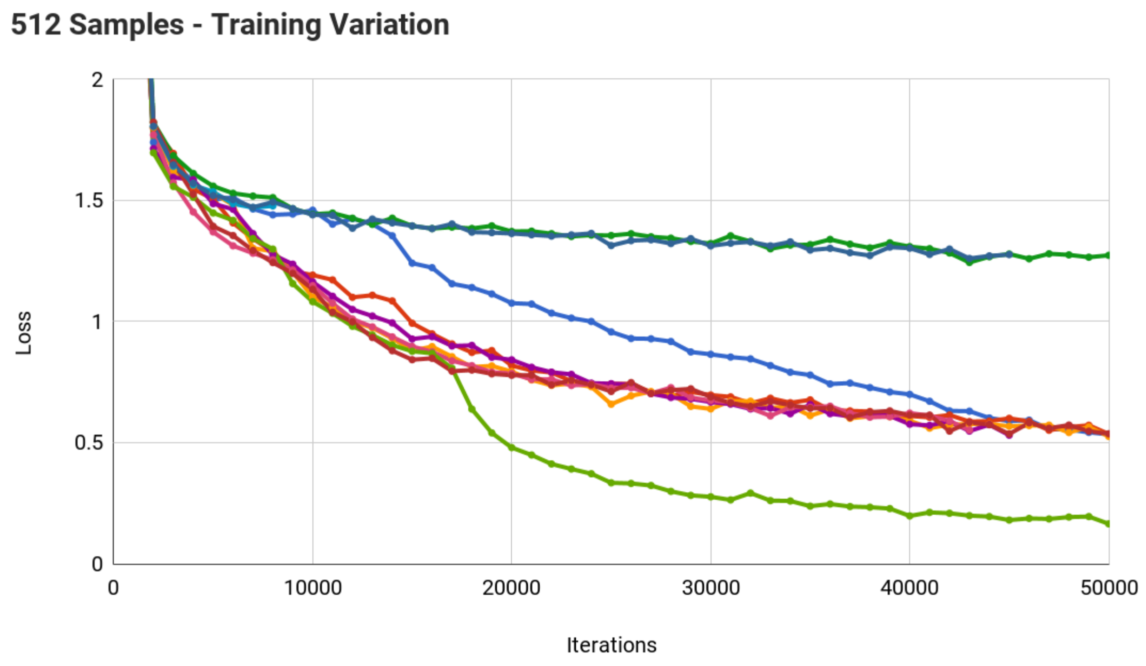


Fig. 3.9 The individual training runs performed with 512 weakly supervised samples. Note the fact that curves tend to follow one of three tracks possibly a result of different valleys in weight space.

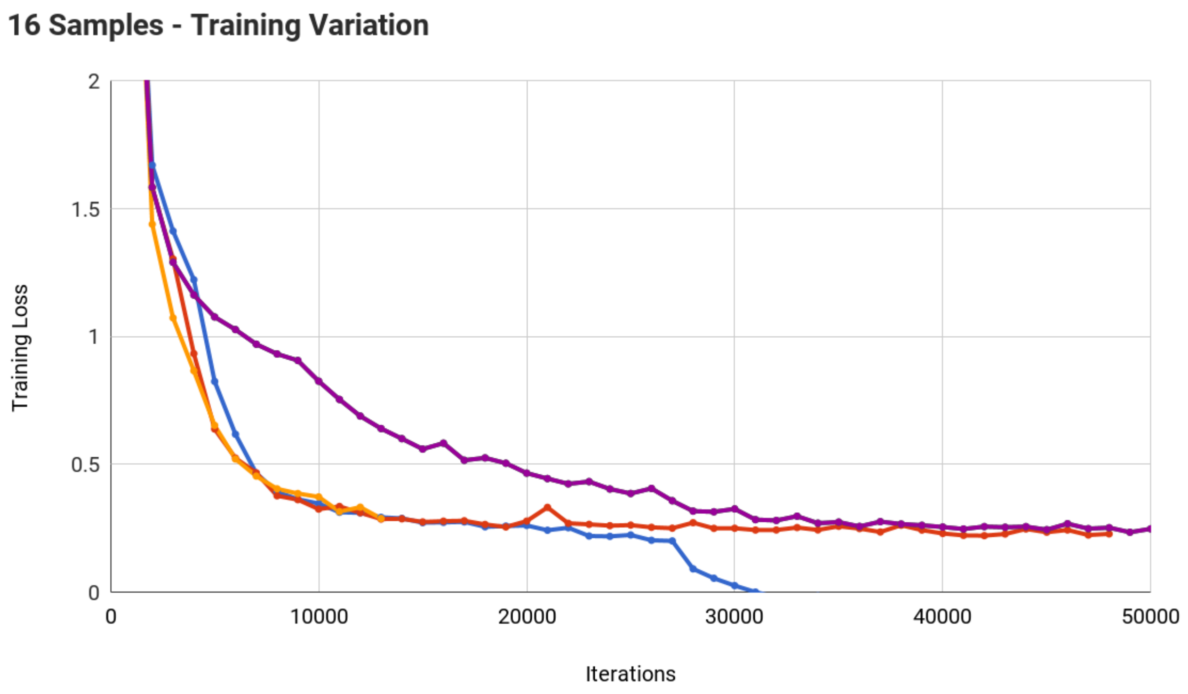


Fig. 3.10 The training runs for the model with 16 Samples. Note that only one sample appears to fully explain the training data.

**Analysis**

The failure to generalise from the training samples stems from the controller modelling the action sequence directly, rather than encapsulating actions within functions. Indeed, only a single function was occasionally learned, in which the controller would place a single block before calling `POP`. The advantage of this function seemingly that the model now has an additional time-step to evaluate whether a block need be placed or indeed if there is a pre-placed block in that location.

The percentage progression through the Nanocraft task was also recorded for each of the held out test samples. As one would expect, this increased with the degree of training data - Figure 3.11. The $Q_i$ values from Equation 3.3 were averaged over each epoch and visualised. Notice that the marginal probability of generating correct elementary operations evolves in line with the learning curves, there are sudden 'jumps' in the proportion of $Q_i$ values that have high probability (Figure 3.12).

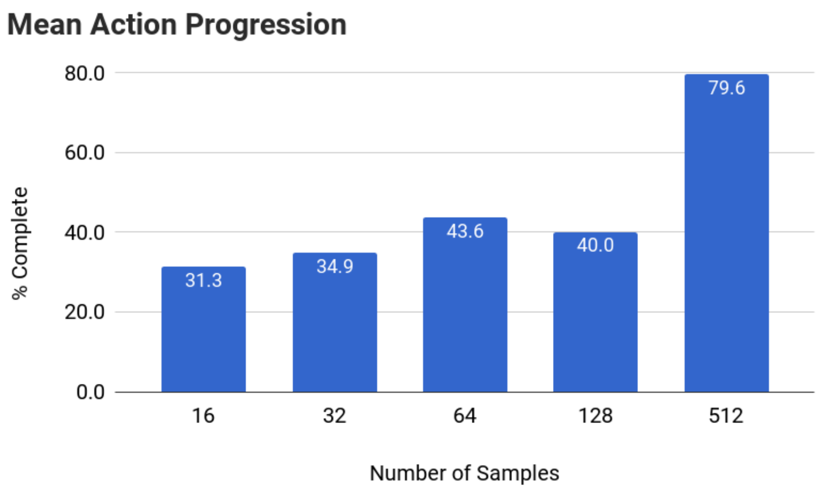
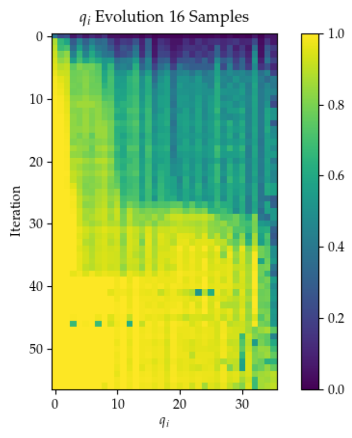
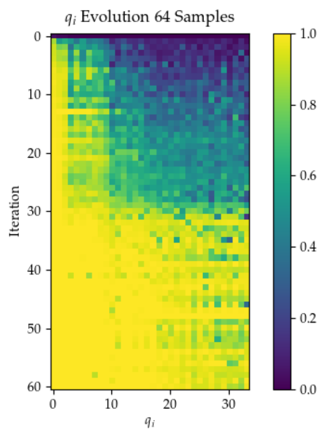


Fig. 3.11 The average progression of models trained under each number of samples through the *Nanocraft* program. There is a clear trend for the progression to increase with the number of samples, as one would expect. The anomalous result for 128 samples reflects the volatility of the training process, with none of the performed trials completely fitting to the training samples in the allowed iterations.

### $Q_i$ Evolution



(a)



(b)

Fig. 3.12 A matrix plot of $Q_i$ values averaged over successive groups of 1000 iterations. $Q_i$ is the marginal probability of generating the $i$th elementary operation correctly, conditional on having generated the preceding $i-1$ elementary operations either correctly or incorrectly (Equation 3.3).

## 3.4 Improving Training Stability

The training process is computationally intensive for a single run, compounded by high volatility with identical hyper-parameters leading to different final results. The same features were exhibited in experiments by [7] and [9]. We looked at different approaches to alleviate this.

### Scheduling Training Samples

We note that the error (and normalised gradient) of some training samples decreases to near zero relatively quickly - while others remain high. Rather than sampling the training data uniformly, we experiment with drawing samples in proportion to the magnitude of their average gradient. Similar approaches taken by [9] were found to decrease the number of iterations required for convergence. Unfortunately in practice, we find this often leads to less stable training curves so was not pursued (Figure 3.13).

### Gradient Noise

Neelakantan et al. found in [8] that addition of Gaussian Noise to the gradients helped program induction, possibly by perturbing trajectories in weight space away from attractive modes. This was investigated but not found to make observable difference in our experiments.

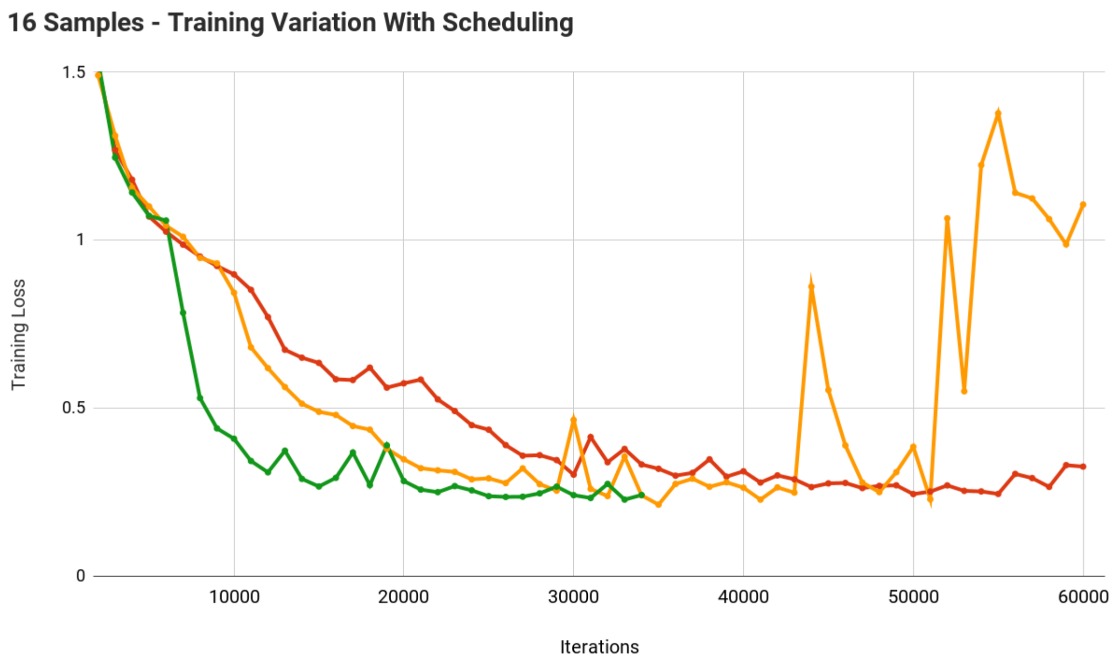


Fig. 3.13 Trials using the scheduling procedure are found to be less stable - especially towards the end of training.

# Chapter 4

# Encouraging Hierarchy Use

Experiments in the previous chapter might suggest that gradient descent alone is insufficient to infer complex functional behaviour from observable actions. We seek to better understand whether this is the case and if the NPL framework could be used to learn functions when there is a very strong necessity to do so. We draw a dichotomy here between two groups of tasks that we refer to as *world dependant* and *world independent*. In both cases we find that the NPL only resorts to learning functions when all other forms of direct sequence to sequence learning are strongly penalised. Indeed, even when the hierarchy is used, functions inferred are not of the optimal form - with the model attempting to encapsulate actions in a single function that should be captured by two or three.

## 4.1 Recording and Constraining Hierarchy Use

One straightforward way to ensure that functions of some form are used would be to penalise long chains of elementary operations called from the same level of the hierarchy. This would account for the controller calling a single function and then modelling the entire process with the LSTM. To avoid the difficulties involved with measuring *expected function length*[1] we instead measure the *expected number of* `PUSH` operations at each time step which we denote $\zeta_t$[2]. To calculate $\zeta_t$ we note that the difference between the elementary operation index $i$ and the current time-step $t$ at any point within the lattice is equal to the number of `PUSH` / `POP` calls made during that path through the lattice. The number of `PUSH`'s called is thus half this number (Equation 4.1).

$$\zeta_t = \sum_{i,l} \frac{t-i}{2} y_i^{t,l} \tag{4.1}$$

[1]See the appendix for a less elegant way to constrain lattice flow

[2]Note that the expected number of `PUSH` operations is identical to the expected number of functions from which *expected sequence length* can be calculated.

### Enforcing Minimum PUSH Calls - $\beta_0$

The first penalty we implement encourages the total expected number of PUSH calls to be equal to the actual number required, namely the number of functions called in generation of the synthetic sequence - $n$. We denote this penalty $\beta_0$. The penalty is computed by taking square of the residual between the desired and actual number of PUSH calls at any point in training (Eq. 4.2). Implementation of $\beta_0$ alone leads to the model simply calling a sequence of PUSH / POP operations initially before modelling the entire sequence within a single program at the top of the hierarchy (Figure 4.1).

$$\beta_0 = (n - \sum_t \zeta_t)^2 \tag{4.2}$$

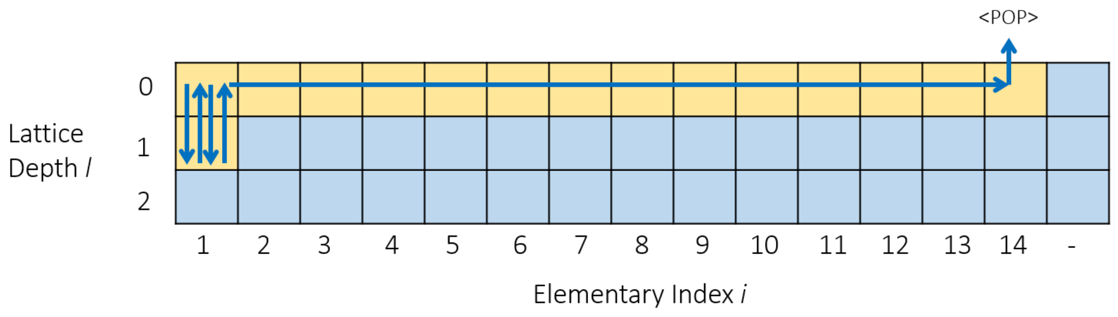


Fig. 4.1 When $\beta_0$ is used alone, the model avoids proper use of function calls by simply calling several zero-length functions before modelling the actions as a sequence.

### Enforcing Minimum Local Variance - $\beta_1$

To prevent repeated length-zero function calls, the *local variance* $v_w$ of $\zeta$ is computed, taking the form of a modified 1D convolutional window that calculates variance - rather than a weighted sum - within each window $w$ (Eq. 4.3). We similarly define $\beta_1$ as the square of the residual between $v_w$ and the true local variance, $\hat{v}_w$. Ideally, the values of $\zeta_t$ would be zero for elementary operations (where the expected number of POP or PUSH calls is zero) and 1 intermittently where the controller enters or returns from a function call. Hence the true value of *local* variance will be high.

$$v_w = \sum_{k \in w} \frac{(\zeta_k - \mu_w)^2}{n-1} \tag{4.3}$$

$$\beta_1 = \sum_w (\hat{v}_w - v_w)^2 \tag{4.4}$$

Under constraints from $\beta_0$ and $\beta_1$, the PUSH / POP calls are made with more appropriate spacing and frequency. However, the controller still avoids using the functions by performing PUSH before POP, moving along the top of the lattice (Figure 4.2).

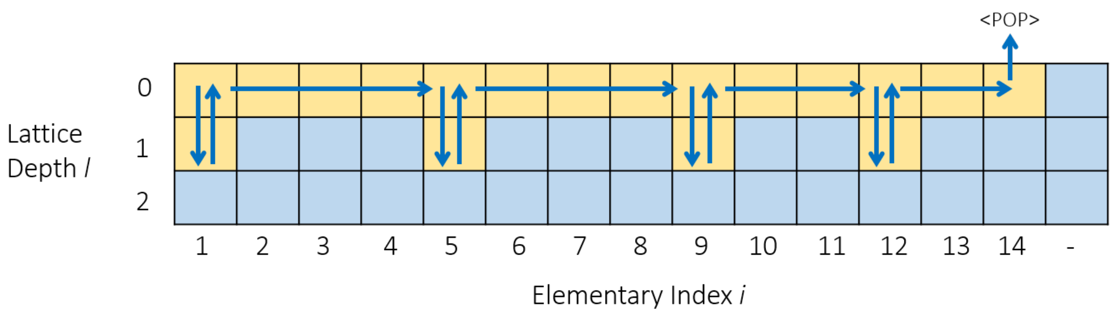


Fig. 4.2 When constrained through $\beta_0$ and $\beta_1$ the controller avoids having to learn the correct program embeddings by calling the correct number of PUSH and POP operations but in such a way as to still model the sequence as usual.

### Enforcing Minimum Depth - $\beta_2$

A final penalty is applied to restrict the expected depth of paths through the lattice. The expected depth, $\delta$ is calculated in (Eq. 4.5) with the target being the actual depth expected if the correct number of functions are used. The loss applied is then the square of the difference between the true and expected depths, denoted $\beta_2$.

$$\delta = \sum_{i,t} l y_i^{t,l} \tag{4.5}$$

$$\beta_2 = (\hat{\delta} - \delta)^2 \tag{4.6}$$

The combination of these three losses forces the NPL to call functions of approximately the right length without specifying exactly when functions must be called and for what purpose. Without any one of these, the functions would not be used.

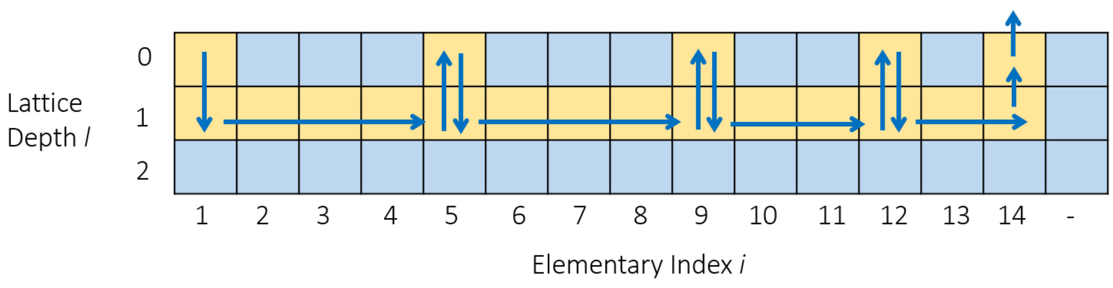


Fig. 4.3 When $\beta_0$, $\beta_1$ and $\beta_2$ are used the desired result is exhibited with functions of approximately constant length being called.

## 4.2 Bitflip

*Bitflip* is the name of a new task designed to analyse functional inference. The aim is to create an environment in which the gap between loss while using the hierarchy and that without is as large as possible. We consider several variations on *Bitflip* with the central theme involving an array of binary digits that must be manipulated according to the actively running program. We distinguish between the two forms of task set, referred to as; **world dependant** in the case where actions depend on the presence of the *world state* $w_t$ and **world independent**, in which actions only depend on the currently executed program. Using gradient descent alone in both cases leads to the top level program calling only elementary operations without use of the other levels. To avoid this we implement the three aforementioned penalties $\beta_0$, $\beta_1$ and $\beta_2$ to force the calling of functions but allow the model flexibility to decide which programs are called and when.

### World Independent Sequences

In the world independent paradigm, a set of patterns is generated (one for each callable function) and a sequence formed by concatenating the patterns in a random order. The order of patterns is shown at first before being obscured, forcing the LSTM to remember the order so as to execute it correctly (Figure 4.4 describes this in detail). The most efficient way to solve this task would be encapsulating each pattern within a single function and then calling in the order shown. Attempting to remember the sequence as a whole would mean larger parametric complexity within the hidden layer of the LSTM.

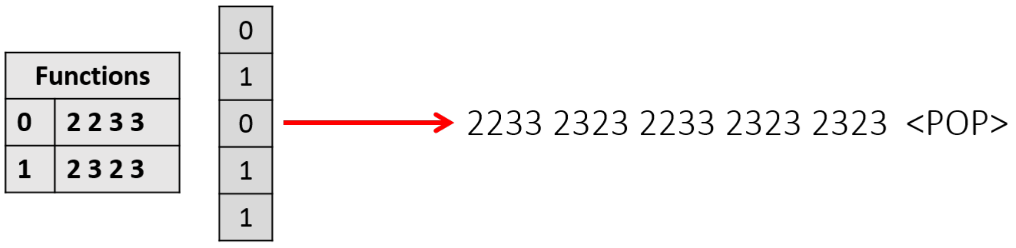
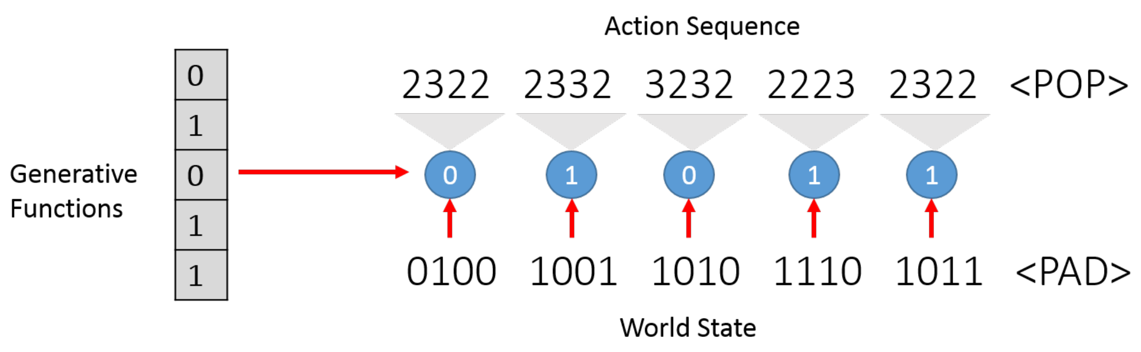


Fig. 4.4 The *world independent* Bitflip environment. To generate the synthetic data, a set of latent functions (sequences) are generated (far left). These functions are then concatenated together repeatedly in a random order and this order is shown to the NPL at the start of program execution. The simplest way to solve the task is assigning a program embedding to model each of the latent functions and then calling these programs in the order shown originally.

### World Dependant Sequences

In the world dependant case there exists an environment $w_t$ that the controller must manipulate according to the currently running function. This takes the form of an array of bits and a positional marker (Figure 4.5). The programs called now encapsulate a *behaviour* (or *policy*) rather than a simple fixed pattern of actions. We posit that this additional complexity could force the model into requiring the use of functions, especially if its LSTM hidden state was sufficiently small to lack the necessary capacity to model the behaviour directly.


Action Sequence
2322 2332 3232 2223 2322 <POP>
Generative Functions
0
1
0
1
1
0 1 0 1 1
0100 1001 1010 1110 1011 <PAD>
World State

| Function | Behaviour |
|---|---|
| 0 | (0 →2) (1→3) |
| 1 | (0 →3) (1→2) |

Fig. 4.5 *World Dependant* Bitflip the action sequence depends both on latent functions and the current state of the world. Random functions are again generated however in this case they represent *desired behaviour* rather than fixed patterns of actions. The currently running function dictates how the agent interacts with the value observed in the world grid. For example the random function - marked '0' - is generated in the bottom of this diagram and indicates that when running, the action 2 should be called when a 0 is seen and 3 when a 1 is seen. A positional marker would also be present in the world grid to indicate where the agent is in the array of bits.

## 4.3 Bitflip Results

### 4.3.1 Learnt Functions

We carry out a set of experiments in both **world independent** and **world dependant** domains. Due to a non-linear rise in computational complexity with sequence length it is unfortunately not possible to consider long sequences where the use of functions would have greatest advantage. As a compromise, we restrict the controller to only 5 hidden units to remove as much capacity for direct sequence modelling as possible. We find that despite this, learning functions does not come naturally to the NPL with the controller strongly favouring use of one function (Figure 4.6). Analysis of the patterns produced by different functions after training has converged shows high variation between the output from a single function call (Table 4.1), suggesting the model is attempting to use one function to fit both patterns.

| Learnt Function 1 | Counts | Learnt Function 2 | Counts |
|---|---|---|---|
| 3 | 10 | 3, 2, 3, 2, 3 | 38 |
| *blank* | 228 | 2, 3, 2, 2, 3 | 48 |
| 2,3,2,2 | 10 | 3, 3, 3, 2, 3, 3 | 10 |
| 2, 3, 2, 3, 2 | 38 | *blank* | 10 |
| 2, 2, 3, 2, 3, 2 | 38 | 2, 2, 3, 3, 2, 3 | 10 |

Table 4.1 The actions performed when Functions 1 and 2 were called, alongside their frequency during the greedily decoded held-out data. These are the final form of the functions, before training terminated (Epoch 66) corresponding to Iteration 66000 in Figure 4.6. The true generative functions were **2233** and **2323**.

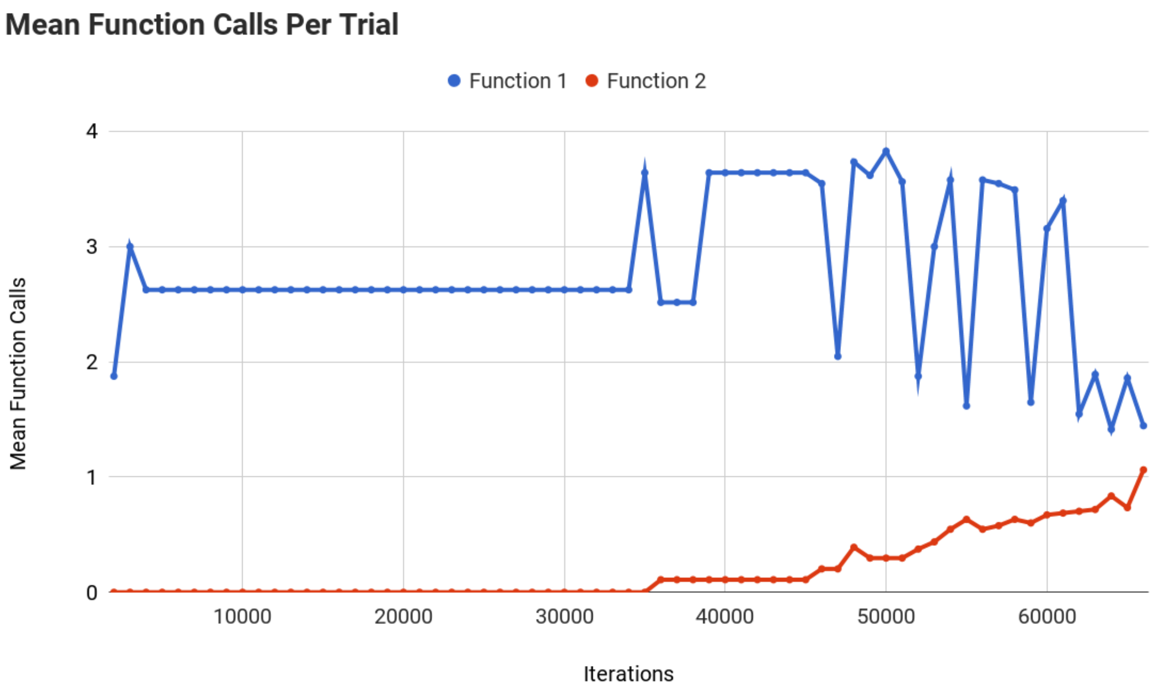


Fig. 4.6 We monitor the average number of each Function called during the greedy decoded runs performed on held-out data at the end of each epoch (1000 iterations). The controller rapidly learns to use a single function, minimising the three lattice penalties before adjusting slightly to call the second function more frequently. Training terminates when the loss decreases to zero for a set of consecutive iterations.

# Chapter 5

# Related Work

In recent years there has been a surge of interest in **Augmented Neural Computation** - the notion of endowing a fuzzy pattern matching neural machine with an external form of persistent manipulable memory [3]. This interest has stemmed primarily from the realisation that standard Recurrent Neural Networks (RNNs), including LSTMs are incapable of modelling certain simple, deterministic sequences [1] and fail to generalise to more than one task, suffering from *catastrophic forgetting* [5] [8] .

### Neural Turing Machine (NTM)

The Neural Turing Machine proposed in [2] first showed that augmenting an LSTM with a manipulable read-write external memory lead to significant improvements in generalisation ability and enabled the machine to generalise to larger sequences than shown in training. The authors provided an LSTM controller with memory in the form of a matrix and attention-based, read/write heads. The focus and position of the heads could be tuned, and read/write operations were soft, making the system end-to-end differentiable and allowing the LSTM to attend to different parts of the memory. Results showed the NTM could infer basic algorithms from a limited number of input-output training examples.

### Stack-Augmented RNN

The Stack-Augmented RNN model in [5] followed a different approach, endowing an LSTM controller with memory stacks to which it could optionally push its hidden state to or read from. It was demonstrated that these stacks enabled the model to perform deterministic sequence modelling tasks with very long dependencies. These tasks were not possible for a standard LSTM as the memory capacity of the hidden units was too small. The soft stack updates performed are similar to those performed at the lattice nodes of the NPL, as described in Chapter 2

[1]Without very large numbers of parameters and training data

### Neural Programmer Interpreter NPI

The *Neural Programmer Interpreter* [9], is the direct predecessor to the *Neural Program Lattice* on which this work is based. The NPI introduced the concept of a hierarchy of programs, that could be called by the LSTM controller. When called, the program's embedding is fed as input to the LSTM in the proceeding time step. In this work, learning could only take place from fully supervised sequences of operations. That is, training data containing the full execution trace of both fundamental operations and abstract programs. The use of a program hierarchy allowed a single LSTM core to learn several distinct algorithmic tasks such as sorting, long addition and canonicalizing 3D models. Encapsulation of repeated behaviour allowed the NPI to achieve perfect generalisation after only 8 samples whereas a sequence-to-sequence model required 256. As discussed in Chapter 2, the Neural Program Lattice built on this work by enabling learning through mostly weak supervision.

### Neural Programmer

Neelakantan et al. augment a neural network with a small set of basic arithmetic and logic operators, trainable with gradient descent. These operations can be called sequentially to compose higher order programs. The training data is unsupervised, avoiding expensive labelling and tasks considered include synthetic table comprehension questions consisting of question answer pairs. The authors note that training under purely weak supervision is challenging but found addition of Gaussian noise to the gradient to improve generalisation [8].

# Chapter 6

# Discussion

## 6.1 Conclusion

We set out to show that the Neural Program Lattice is capable of inferring a generative program hierarchy in the absence of hand-crafted supervision. Modification of the loss function enables learning, albeit without evidence of functional inference. Forcing the use of hierarchy through penalty terms on new, synthetic data shows that the NPL will use as few functions as possible to perform a given task, settling with one when two or three would be preferable. It thus would appear that our original weight-space hypothesis in Section 3.1.1 remains apt, with functional inference possible by the NPL but much less preferable to the strongly attractive region corresponding to sequence to sequence modelling by the LSTM controller in the top level of the hierarchy. Two statements seem fair to conclude with regards to the NPL and its predecessor the NPI:

1. The NPI *learns* and *executes* functions to great effect - enabling low sample complexity and strong generalisation.
2. The NPL is advantageous in that **given a pre-defined hierarchy**, the provision of weakly supervised data enables significant performance increases from the NPI alone.

What is less clear, is whether the NPL can infer the hierarchy from elementary operations alone. Indeed, experiments in the *Bitflip* domain suggest this does not come naturally for the NPL in its current form, without pathological and seemingly over-engineered use of penalties. Cases in which the NPL would have a strong need to use functions, such as with very long sequence lengths are out of scope experimentally due to non-linear increases in computational complexity with the maximum length of sequence.

The core problem with inference in the NPL stems from the use of functions as a voluntary aid to the controller, rather than integral to the modelling process itself. In analogy with CNNs and RNNs where the weight-sharing component is fundamental to learning, the NPL/NPI is an LSTM with the added capacity for functional *execution*. The result is that program induction is not prioritised, with the LSTM controller preferring to stay in the top level of the hierarchy and focus on modelling the sequence directly.

## 6.2 Future Work

Logical encapsulation and functional inference are arguably the next frontier of Deep Learning and form a natural next step to the existing hierarchies present in computer vision and time-series modelling. While it is difficult to suggest concrete re-formulations of the NPL to enable program induction, we offer two guiding principles for future work:

1. Learnt programs must be intrinsic to the modelling process, just as in a CNN the network is forced to compose lower level filters to form higher level features, so this should be the case with functional inference.

2. The inference procedure should have some form of localised learning ability, in the same way that an RNN can sequentially model a long text document, rather than having to observe the entire sequence as a whole.

Without changing the NPL, it would be interesting to see if a HMM or Hierarchical HMM [12] could be used to segment sequences of elementary operations $\lambda$ into clusters of similar behaviour for use as approximate strong supervision. This could potentially bias the NPL towards learning more specific functions with further weak supervision.

# References

[1] Badre, D. and Frank, M. J. (2012). Mechanisms of hierarchical reinforcement learning in cortico–striatal circuits 2: Evidence from fmri. *Cerebral Cortex*, 22(3):527–536.

[2] Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *CoRR*, abs/1410.5401.

[3] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A. P., Hermann, K. M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

[4] Hinton, G., Deng, L., Yu, D., Dahl, G., rahman Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*.

[5] Joulin, A. and Mikolov, T. (2015). Inferring algorithmic patterns with stack-augmented recurrent nets. *CoRR*, abs/1503.01007.

[6] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[7] Li, C., Tarlow, D., Gaunt, A., Brockschmidt, M., and Kushman, N. (2017). Neural program lattices. In *Proceedings of ICLR'17*.

[8] Neelakantan, A., Le, Q. V., and Sutskever, I. (2015). Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834.

[9] Reed, S. E. and de Freitas, N. (2015). Neural programmer-interpreters. *CoRR*, abs/1511.06279.

[10] Stahlberg, F., Hasler, E., Waite, A., and Byrne, B. (2016). Syntactically guided neural machine translation. *CoRR*, abs/1605.04569.

[11] Wang, H. and Yeung, D.-Y. (2016). Towards bayesian deep learning: A survey. *CoRR*, abs/1604.01662.

[12] Fine, S., Singer, Y., and Tishby, N. (1998). The Hierarchical Hidden Markov Model: Analysis and Applications

# Appendix A

# Recording Hierarchy Utilisation

If lattice use can be monitored, a penalty can be implemented to encourage paths within the lattice that use lower levels. As such, a second recursive variable was devised that could track hierarchy depth utilisation at every node of the lattice analogous to the existing $y_i^{t,l}$ variable. The variable $z_{i,l'}^{t+1,l}$ was defined as below to track the expected number of elementary operations called at depth $l'$. Intuitively, the $l'$th index of the variable increases when moving horizontally within the lattice at a depth $l'$, weighted by the probability mass $\hat{y}_{i-1}^{t,l}$at the preceding index $(i-1)$.

$$z_{i,l'}^{t+1,l} = (1/\sum_{i,l} \bar{y}_i^{t+1,l}) p(\texttt{OP})_{a,i-1}^{t,l} \hat{y}_{i-1}^{t,l} p_{o,i-1}^{t,l} (\lambda_o^i) \{ z_{i-1,l'}^{t,l} + e_{l'}^{l} \} + p(\texttt{POP})_{a,i}^{t,l+1} \hat{y}_i^{t,l+1} z_{i,l'}^{t,l+1} + p(\texttt{PUSH})_{a,i}^{t,l-1} \hat{y}_i^{t,l-1} z_{i,l'}^{t,l-1}$$

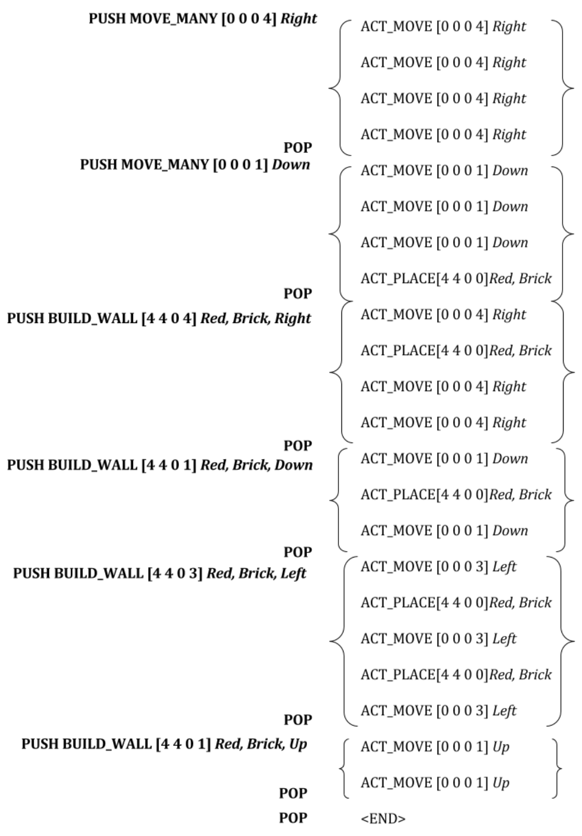
$$e_{l'}^{l} = \begin{cases} 1, & \text{if} \quad l = l' \\ 0, & \text{otherwise} \end{cases}$$

This facilitates the quantitative assessment of hierarchy utilisation as the constituent values of each point in the lattice sum to the total number of elementary operations generated thus far.

$$i = \sum_{l' \leq L} z_{i,l'}^{t,l}$$

A regularisation term $\beta$ was also defined, dependant on a weightings vector $\lambda_{l'}$. Paths calling the entirety of elementary operations from the top level of the lattice would now generate higher losses than those at lower levels.

$$\beta = log \sum_{t \leq T} p_{a,I}^{t,0}(\texttt{POP}) y_I^{t,0} \sum_{l' \leq L} \frac{\lambda_{l'}}{I} z_{I,l'}^{t,0}$$



```
PUSH MOVE_MANY [0 0 0 4] Right           ACT_MOVE [0 0 0 4] Right
                                         ACT_MOVE [0 0 0 4] Right
                                         ACT_MOVE [0 0 0 4] Right
                                         ACT_MOVE [0 0 0 4] Right
POP
PUSH MOVE_MANY [0 0 0 1] Down            ACT_MOVE [0 0 0 1] Down
                                         ACT_MOVE [0 0 0 1] Down
                                         ACT_MOVE [0 0 0 1] Down
                                         ACT_PLACE[4 4 0 0]Red, Brick
POP
PUSH BUILD_WALL [4 4 0 4] Red, Brick, Right   ACT_MOVE [0 0 0 4] Right
                                         ACT_PLACE[4 4 0 0]Red, Brick
                                         ACT_MOVE [0 0 0 4] Right
                                         ACT_MOVE [0 0 0 4] Right
POP
PUSH BUILD_WALL [4 4 0 1] Red, Brick, Down    ACT_MOVE [0 0 0 1] Down
                                         ACT_PLACE[4 4 0 0]Red, Brick
                                         ACT_MOVE [0 0 0 1] Down
POP
PUSH BUILD_WALL [4 4 0 3] Red, Brick, Left    ACT_MOVE [0 0 0 3] Left
                                         ACT_PLACE[4 4 0 0]Red, Brick
                                         ACT_MOVE [0 0 0 3] Left
                                         ACT_PLACE[4 4 0 0]Red, Brick
                                         ACT_MOVE [0 0 0 3] Left
POP
PUSH BUILD_WALL [4 4 0 1] Red, Brick, Up      ACT_MOVE [0 0 0 1] Up
                                         ACT_MOVE [0 0 0 1] Up
POP
POP                                      <END>
```

Fig. A.1 The *Nanocraft* task. An example execution trace with elementary operations shown adjacent to corresponding *strong supervision*. The hand-crafted functions would not be present under the case of purely weak supervision.

*The image of the human brain before the introduction is taken from Cerebral Localization in Thomas' Eclectic Practice of Medicine.*

# Appendix B

# Further Training Curves

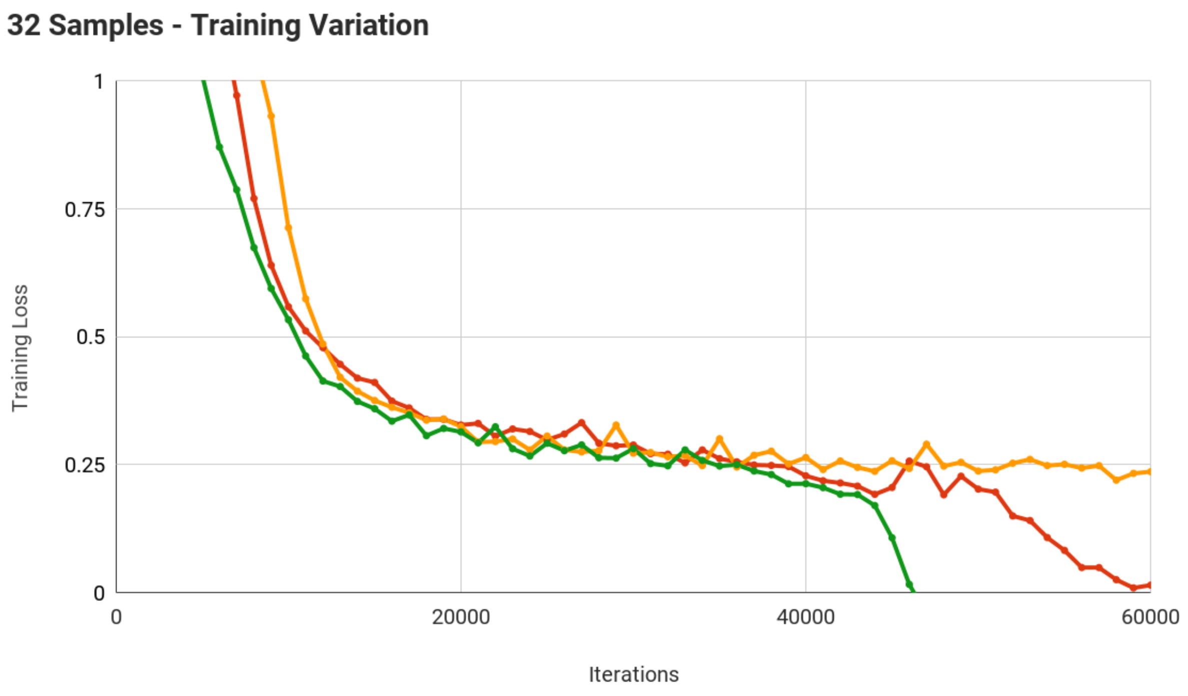


Fig. B.1 Individual trials with 32 Weak training samples.

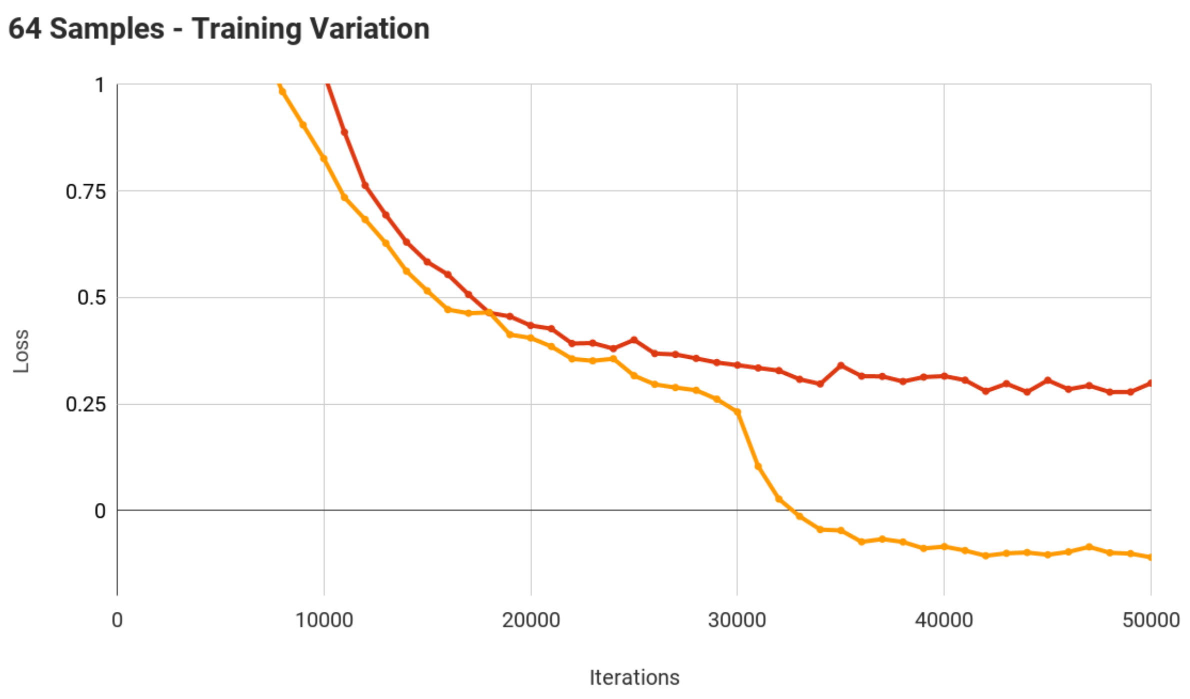


Fig. B.2 Individual trials with 64 Weak training samples.

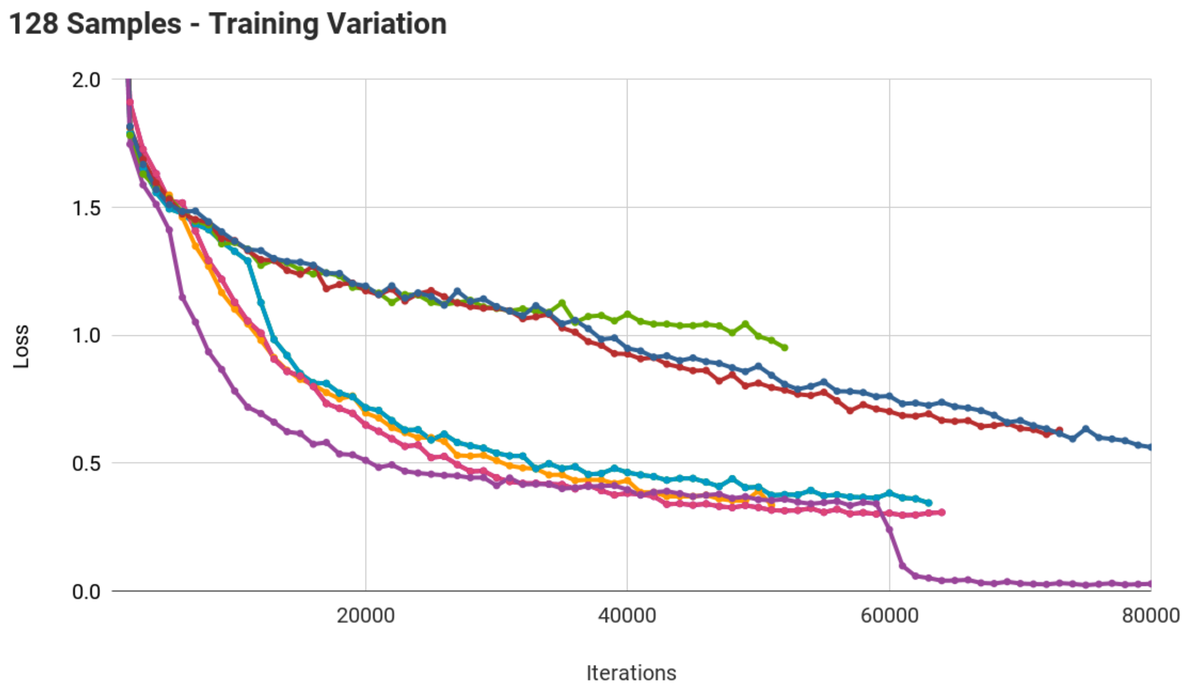


Fig. B.3 Individual trials with 128 Weak training samples.