NEURAL PROGRAM LATTICES

Chengtao Li [*]
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
ctli@mit.edu

Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman
Microsoft Research
Cambridge, CB1 2FB, UK
{dtarlow,algaunt,mabrocks,nkushman}@microsoft.com

ABSTRACT

We propose the Neural Program Lattice (NPL), a neural network that learns to perform complex tasks by composing low-level programs to express high-level programs. Our starting point is the recent work on Neural Programmer-Interpreters (NPI), which can only learn from strong supervision that contains the whole hierarchy of low-level and high-level programs. NPLs remove this limitation by providing the ability to learn from weak supervision consisting only of sequences of low-level operations. We demonstrate the capability of NPL to learn to perform long-hand addition and arrange blocks in a grid-world environment. Experiments show that it performs on par with NPI while using weak supervision in place of most of the strong supervision, thus indicating its ability to infer the high-level program structure from examples containing only the low-level operations.

1 INTRODUCTION

A critical component of learning to act in a changing and varied world is learning higher-level abstractions of sequences of elementary tasks. Without such abstractions we would be forced to reason at the level of individual muscle contractions, making everyday tasks such as getting ready for work and making dinner almost impossible. Instead, as humans, we learn a hierarchy of skills starting with basic limb movements and eventually getting to the level of tasks such as *get ready for work* or *drive to the airport*. These abstractions have many different names. For example, in computer programming they are called functions or subroutines and in reinforcement learning they are called options or temporally extended actions. They facilitate learning in two important ways. First, they enable us to learn faster, i.e. with lower sample complexity. Second, they enable us to strongly generalize from our prior experience so that we can, for example, drive to a new location once we have learned how to drive to a few other locations.

A primary mechanism used for learning is watching others perform a task. During such demonstrations, one typically observes the *elementary operations* performed, such as the movements of individual limbs or the mouse clicks in a computer interface. In some cases, the demonstrations can also provide supervision of the *abstract operations* (i.e., the abstraction hierarchy) that generated the elementary operations, either through a formal annotation process or through informal natural language descriptions. Recent work on Neural Programmer-Interpreters, NPI (Reed & de Freitas, 2016), has shown that when the training data includes both elementary and abstract operations, learning the abstractions results in strong generalization capabilities. This enables, for example, the ability to add very large numbers when trained only on the addition of relatively small numbers.

Providing supervision of the abstract operations during a demonstration requires significant additional effort, however, and so in typical real-world scenarios we will observe only the elementary operations. For example, we can see a person's limbs move (elementary operations), but we cannot see the mental states that led to these movements (abstract operations). In the same vein, we

[*]Work done primarily while author was an intern at Microsoft Research.

can easily capture a user's clicks in an online application or their real-world movements using a skeletal tracking depth camera (Microsoft Corp. Redmond WA). NPI cannot directly be applied on data like this, however, because the data does not contain the abstraction hierarchy. This motivates the desire for a model which can learn an abstraction hierarchy from only sequences of elementary operations, but this is an ill-posed problem that requires either additional modeling assumptions or some strongly supervised data. In this work, we take a first step by assuming access to a small number of strongly supervised samples that provide the components of the abstraction hierarchy and disambiguate which of infinitely many abstraction hierarchies are preferred. While we currently only consider domains without noise, we believe our work provides a starting point for future research on adding additional modeling assumptions that could remove the need for strong supervision altogether.

There are several technical issues that arise in developing NPL, which are addressed in this paper. In section 2, we reformulate the NPI model to explicitly include a program call stack, which is necessary for the later modeling developments. Next we need to formulate a training objective for weakly supervised data instances. Ideally we could treat the abstract operations as latent quantities and optimize the marginalized log probability that arises from summing out the abstract operations. However, there are exponentially many such abstraction hierarchies, and so this is computationally intractable. To overcome this challenge, we compute an approximate dynamic program by building on two ideas from the literature. First, we draw inspiration from Connectionist Temporal Classification, CTC (Graves et al., 2006), observing that it provides a method for learning with latent alignments. In section 3.1 we reformulate the CTC objective into a feedforward process that executes a dynamic program. Applying this to our problem, however, requires handling the program call stack. In section 3.2 we do this through an approximation analogous to that of Stack-Augmented Recurrent Nets, StackRNNs (Joulin & Mikolov, 2015), resulting in a fully-differentiable feedforward process that executes a dynamic program to approximately compute the marginalized log probability that we desire. Finally, we observe in section 3.3 that there are alternative dynamic programs for approximating the desired marginalized log probability and present one that uses more computation to more closely resemble the exact (exponentially expensive) dynamic program while remaining tractable.

Our key contributions can be summarized as follows:

- We show how ideas from CTC and StackRNNs can be adapted and extended to enable the training of NPI-like models from only flat sequences of elementary operations and world states.
- We introduce a method to compute a more accurate approximation of marginalized log probabilities in such models.
- On the long-hand addition task from Reed & de Freitas (2016) and a new task involving arranging blocks in a grid-world, we demonstrate empirically that using NPL to train with elementary operation sequences combined with only a few training samples with full program traces can achieve similar performance to NPI but with weaker supervision.

2 Model Background

The NPI model is based on a Recurrent Neural Network (RNN) which, at each step, either calls an abstract program, performs an elementary operation, or returns from the current program. To make this decision, each step of the RNN takes as input: (1) a learnable embedding of program to execute, (2) embedded arguments for this program, and (3) an embedding of the current world state. Calling an abstract program resets the LSTM hidden state to zero and updates the program and arguments provided as input to the following steps. Returning from an abstract program inverts this process, restoring the hidden state and input program and arguments to those from before the program was called. Performing an elementary operation updates the world state, but leaves the current program and arguments in place, and performs the standard LSTM update of the hidden state.

Rather than present the details of the NPI model as in Reed & de Freitas (2016), we will cast it in the formulation that we will use throughout the paper. The main difference is that our presentation will explicitly maintain a call stack, which we will refer to as Stack-based NPI. Morally this does not change the model, but it will enable the extension to weaker supervision described in section 3.

The basic structure of the reformulated model can be seen in Figure 1. The model learns a library of *programs*, $\mathcal{G}$, and *arguments*, $\mathcal{R}$, to these programs, where each program $g \in \mathbb{R}^n$ and each argument

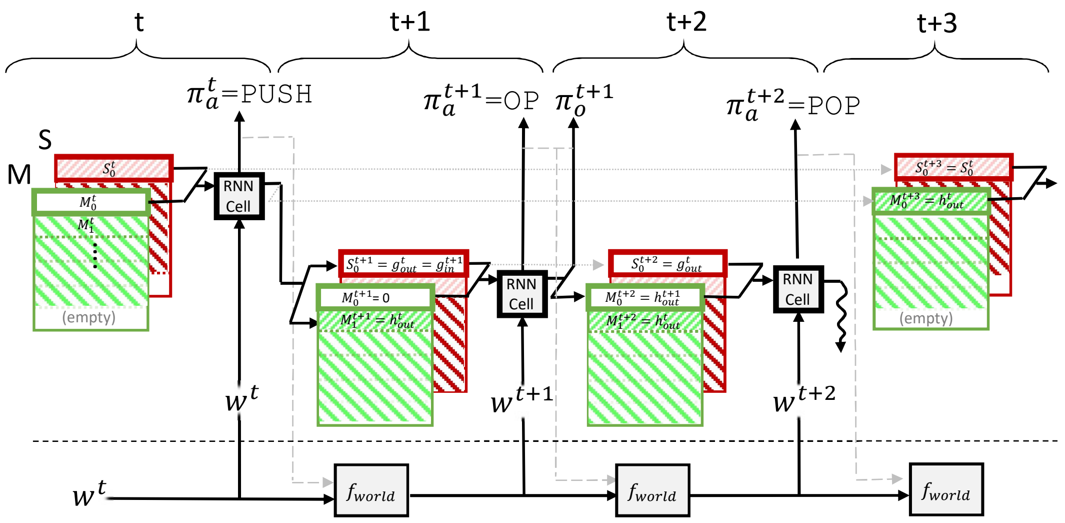


Figure 1: **Stack-based NPI:** Four time steps from the execution of the stack-based NPI model. Each color/hash pattern represents a unique set of unchanging data values which, over time, move up and down (and in and out of) the stack. Operations below the dotted line to calculate the new world state are executed only at test time, since we do not have access to f_{world} at training time, and the training data contains the correct sequence of world states.

$r \in \mathbb{R}^m$ is represented as an embedding, with n and m as the embedding dimensions. When a program is *called* with a list of arguments it performs a sequence of actions, where each action is one of: OP, PUSH, or POP. OP performs an elementary operation, e.g. move one step. PUSH calls to another program. POP returns from the current program back to the parent program.

An LSTM-based controller, shown in Figure 2, is used to generate the sequence of actions, deciding the action at timestep t based on the currently running program and arguments, g_{in}^t, the LSTM's internal state h_{in}^t and an observation of the current world state, w^t. To support calls to and returns from subprograms, the controller state contains two call stacks, one for the internal RNN state, which we denote as M (green in Figure 1), and one for the program and arguments, which we denote as S (red in Figure 1). M_d^t and S_d^t refer to the elements at depth-d of the stacks at timestep t.

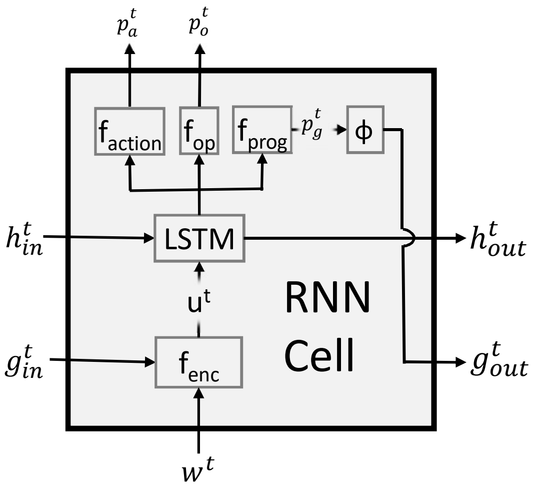


Figure 2: **RNN Cell:** A zoomed in view of the internals of an RNN cell from Figure 1.

The training data for NPI requires full execution traces. We use π to denote all the observations recorded in a single full exectution trace. Specifically, for timestep t in the execution we define π_w^t to be the input world state, and π_a^t to be the decision of which of the following actions to take:

OP: perform elementary operation π_o^t

PUSH: push the currently running program and LSTM internal state onto the call stack and call program π_g^t

POP: return to the parent program by popping the parent's program state off the top of the call stack

Note that, as with the original NPI model, we also include arguments for both the operation and program calls, but for notational simplicity we subsume those into π_o^t and π_g^t respectively.

The stack updates are formally defined as:

$$M_d^{t+1} = \begin{cases} [\![\pi_a^t = \texttt{POP}]\!] M_1^t + [\![\pi_a^t = \texttt{OP}]\!] h_{out}^t + [\![\pi_a^t = \texttt{PUSH}]\!] 0, & d = 0 \\ [\![\pi_a^t = \texttt{POP}]\!] M_2^t + [\![\pi_a^t = \texttt{OP}]\!] M_1^t + [\![\pi_a^t = \texttt{PUSH}]\!] h_{out}^t, & d = 1 \\ [\![\pi_a^t = \texttt{POP}]\!] M_{d+1}^t + [\![\pi_a^t = \texttt{OP}]\!] M_d^t + [\![\pi_a^t = \texttt{PUSH}]\!] M_{d-1}^t, & d > 1 \end{cases} \tag{2.1}$$

$$S_d^{t+1} = \begin{cases} [\![\pi_a^t = \texttt{POP}]\!] S_1^t + [\![\pi_a^t = \texttt{OP}]\!] S_0^t + [\![\pi_a^t = \texttt{PUSH}]\!] g_{out}^t, & d = 0 \\ [\![\pi_a^t = \texttt{POP}]\!] S_{d+1}^t + [\![\pi_a^t = \texttt{OP}]\!] S_d^t + [\![\pi_a^t = \texttt{PUSH}]\!] S_{d-1}^t, & d > 0 \end{cases}$$

The conditions in the Iverson brackets choose which type of update should be performed based on the action type. `POP`ing from the stack moves all items up one location in the stack. Performing an elementary `OP`, updates the top element of stack M to contain the new RNN hidden state but otherwise leaves the stacks unchanged. `PUSH`ing onto the stack pushes the new program and arguments, g_{out}^{t-1}, onto stack S, pushes a default (zero) hidden state onto stack M, and moves all of the other elements in the stacks down one location.

The RNN cell inputs are:

$$\begin{aligned} h_{in}^t = M_0^t = & \text{ the current LSTM internal state,} \\ g_{in}^t = S_0^t = & \text{ the current program and arguments,} \\ w^t = \pi_w^t = & \text{ the current world state.} \end{aligned}$$

Inside the RNN cell, as shown in Figure 2, g_{in}^t and w^t are passed through a task specific encoder network, f_{enc} to generate a combined embedding u_t which is passed directly into an LSTM (Hochreiter & Schmidhuber, 1997). Formally,

$$u^t = f_{enc}(w^t, g_{in}^t), \quad h_{out}^t = f_{lstm}(u^t, h_{in}^t); \tag{2.2}$$

The LSTM output is passed in parallel through four different decoder networks to generate the following probability distributions:

$p_a^t =$	the action	$p_r^t =$	the arguments for the program or operation
$p_g^t =$	the program to be called	$p_o^t =$	the elementary operation to be performed

At test time, we use a greedy decoder that makes the decision with the highest probability for each choice. Formally:

$$g_{out}^t = \phi(p_g^t) = \text{argmax}_{\gamma \in \mathcal{G}}\, p_g^t(\gamma)$$

At training time our objective is to find neural network parameters θ which maximize the following (log) likelihood function:

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

$$p(\pi) = \prod_{t=1}^{T} p(\pi^t), \qquad \mathcal{L}(\theta) = \log p(\pi) \tag{2.3}$$

3 Neural Program Lattices

In this section we introduce our core contribution, a new framework for training NPI-like models when the training data contains only sequences of elementary actions instead of full program abstractions. The basis of our framework is the Neural Program Lattice, which approximately computes marginal probabilities using an end-to-end differentiable neural network.

In this section, the training data is an elementary operation trace λ, which includes a sequence of elementary steps, λ_o, and a corresponding sequence of world states, λ_w. For each elementary step, λ^i, the elementary operation performed is λ_o^i and the input world state is λ_w^i. We define $\mathcal{O}$ as a many-to-one map from a full execution trace π to it's elementary operation trace λ. With these definitions and $p(\pi)$ as defined in equation 2.3, our desired (log) marginal likelihood for a single example becomes

$$\mathcal{L}(\theta) = \log \sum_{\pi \in \mathcal{O}^{-1}(\lambda)} p(\pi). \tag{3.1}$$

Computing this quantity is intractable because the number of possible executions $|\mathcal{O}^{-1}(\lambda)|$ is exponential in the maximum length of π and each execution may have unique stack states. In the following sections, we describe how to approximately compute this quantity so as to enable learning from weak supervision. To also learn from strong supervision, we simply add $\log p(\pi)$ terms to the objective for each strongly supervised example π.

3.1 CTC as a Feed-forward Network

In formulating a loss function which approximates the exponential sum in equation 3.1, the first challenge is aligning the elementary steps, λ_i, in the training data, to the timesteps, t, of the model. Specifically, when the model calls into a program or returns from a program in a given timestep, it does not perform any elementary operation in that timestep. As a result, the alignment between elementary steps in the data and the timesteps of the model depends crucially on the choice of high-level abstraction. To overcome this challenge, we draw inspiration from CTC (Graves et al., 2006).

CTC is an RNN-based neural network architecture used in speech recognition to handle the analogous problem of aligning audio sequence inputs to word sequence outputs. It can be seen as a combination of an RNN and a graphical model. The RNN computes a distribution over possible outputs for each timestep, while the graphical model consumes those distributions and uses a dynamic program to compute the marginal distribution over possible label sequences. A crucial assumption is that the RNN outputs at each timestep are conditionally independent, i.e. no feedback connections exist from the output layer back into the rest of the network. This assumption is incompatible with the NPI model because action decisions from timestep t determine the world state, hidden state, and program input for the next timestep. In section 3.2 we will adapt the CTC idea to work in the NPI setting. In this section we prepare by reformulating CTC into a feed forward neural network that can be trained with standard back propagation.

The main challenge solved by CTC is finding the alignment between the elementary steps, i, observed in the training data and the timesteps, t, of the model. To facilitate alignment discovery, the output layer in a CTC network is a softmax layer with a unit for each elementary operation in O, the set of elementary operations, as well as one additional unit for a `BLANK` output where no elementary operation is performed because (in our case) the model calls into a new program or returns from the current program. Define $\beta \in O'^T$ as an output sequence over the alphabet $O' = O \cup \texttt{BLANK}$. Additionally, define the many-to-one map $\mathcal{B}$ from an output sequence β to λ_o the sequence of elementary operations created by removing all of the `BLANK` outputs from β. As discussed above, the CTC model assumes that the RNN inputs at time t are independent of the decisions made by the model, π. Thus for purposes of this subsection, we will assume both that $h^t_{in} = h^{t-1}_{out}$, and that $w = (w^1, \ldots, w^T)$ and $g_{in} = (g^1_{in}, \ldots, g^T_{in})$ are provided as inputs and are thus independent of the output decisions. We can then formally define

$$p^t(\beta^t|w, g_{in}) = \begin{cases} p^t_a(\texttt{POP}|w, g_{in}) + p^t_a(\texttt{PUSH}|w, g_{in}), & \beta^t = \texttt{BLANK} \\ p^t_a(\texttt{OP}|w, g_{in}) p^t_o(\beta^t|w, g_{in}), & \text{otherwise} \end{cases}$$

$$p(\beta|w, g_{in}) = \prod_{t=1}^{|w|} p^t(\beta^t|w, g_{in})$$

$$\mathcal{L}(\theta|\lambda_o, w, g_{in}) = \log p(\lambda_o|w, g_{in}) = \log \sum_{\beta \in \mathcal{B}^{-1}(\lambda_o)} p(\beta|w, g_{in}).$$

The dynamic program used by CTC to compute this likelihood is based on y^t_i, the total probability that as of timestep t in the model we have generated $\lambda^{1:i}_o$, the first i elementary actions in λ_o. y^t_i is calculated from $w^{1:t}$ and $g_{in}^{1:t}$, the first t elements in w and g_{in} respectively. Formally,

$$y_i^t = \sum_{\beta \in \mathcal{B}^{-1}(\lambda_o^{1:i})} p(\beta | w^{1:t}, g_{in}^{1:t})$$

$$\mathcal{L}(\theta|\lambda_o, w, g_{in}) = \log y_{|\lambda_o|}^{|w|}.$$

We can compute this value recursively as $y_0^0 = 1$ and

$$y_i^t = p^t(\lambda_o^i | w^{1:t}, g_{in}^{1:t}) y_{i-1}^{t-1} + p^t(\mathtt{BLANK} | w^{1:t}, g_{in}^{1:t}) y_i^{t-1}.$$

This formulation allows the likelihood to be computed in a feed-forward manner and the gradients of θ to be computed using standard back propagation through time. Note that if there were feedback connections in the model, then it would not be sufficient to only use y_i^t as the dynamic programming state; we would need to keep track of all the different possible stack states after having produced the sequence prefix, which is what leads to the intractability.

3.2 DIFFERENTIABLE CALL STACK

In the last section we assumed that the RNN inputs w, and g_{in} were defined independently of the decisions π made by the model and that $h_{in}^t = h_{out}^{t-1}$. In this section we show how to relax these assumptions to handle the full Stack-based NPI model described in section 2. The key idea is that rather than propagating forward all possible stack states, which leads to a combinatorial explosion, we will propagate forward a single stack state which is a weighted average of all possible stack states, where the weights are computed based on local probabilities of actions at each timestep. This operation is analogous to that used in StackRNNs (Joulin & Mikolov, 2015). The result is a tractable and differentiable forward execution process that no longer exactly computes the desired marginal likelihood. However, we will show experimentally that learning with this model for weakly supervised examples leads to the behavior that we would hope for if we were learning from the true marginal log likelihood. That is, we can share model parameters while training on strongly and weakly labeled examples, and adding the weakly labeled data improves generalization performance.

In more detail, we estimate all quantities specified in π but not in λ using a soft-argmax function that computes deterministic functions of the previously observed or estimated quantities. These estimated quantities are π_a, π_g, and implicitly π_w. Both π_w and π_g can be directly replaced with a soft-argmax as follows:

$$w^t = \sum_{i \in I} y_i^{t-1} \lambda_w^i \tag{3.2}$$

$$g_{out}^t = \phi_{soft}(p_g^t) = \sum_{\gamma \in \mathcal{G}} p_g^t(\gamma)\gamma \tag{3.3}$$

Replacing decision π_a^t with a soft-argmax changes the stack updates from equation 2.1 into differentiable stack updates similar to those used in Joulin & Mikolov (2015). Formally,

$$y^t = \sum_{i \in I} y_i^t$$

$$\alpha^t(a) = \begin{cases} \sum_{i \in I} (y_i^t / y^{t+1}) p_a^t(a) p_o^t(\lambda_o^i), & a = \mathtt{OP} \\ (y^t / y^{t+1}) p_a^t(a), & a \neq \mathtt{OP} \end{cases}$$

$$M_d^{t+1} = \begin{cases} \alpha^t(\mathtt{POP}) M_1^t + \alpha^t(\mathtt{OP}) h_{out}^t + \alpha^t(\mathtt{PUSH}) 0, & d = 0 \\ \alpha^t(\mathtt{POP}) M_2^t + \alpha^t(\mathtt{OP}) M_1^t + \alpha^t(\mathtt{PUSH}) h_{out}^t, & d = 1 \\ \alpha^t(\mathtt{POP}) M_{d+1}^t + \alpha^t(\mathtt{OP}) M_d^t + \alpha^t(\mathtt{PUSH}) M_{d-1}^t, & d > 1 \end{cases}$$

$$S_d^{t+1} = \begin{cases} \alpha^t(\mathtt{POP}) S_1^t + \alpha^t(\mathtt{OP}) S_0^t + \alpha^t(\mathtt{PUSH}) g_{out}^t, & d = 0 \\ \alpha^t(\mathtt{POP}) S_{d+1}^t + \alpha^t(\mathtt{OP}) S_d^t + \alpha^t(\mathtt{PUSH}) S_{d-1}^t, & d > 0 \end{cases}$$

with α introduced for notational simplicity. This change enables h_{in}^t and g_{in}^t to now depend on the distribution over output decisions at time $t-1$ via the stack, as $g_{in}^t = S_0^t$ and $h_{in}^t = M_0^t$, where S_0^t (resp. M_0^t) are computed from S_0^{t-1} and S_1^{t-1} (resp. M_0^{t-1} and the LSTM cell's output at timestep t).

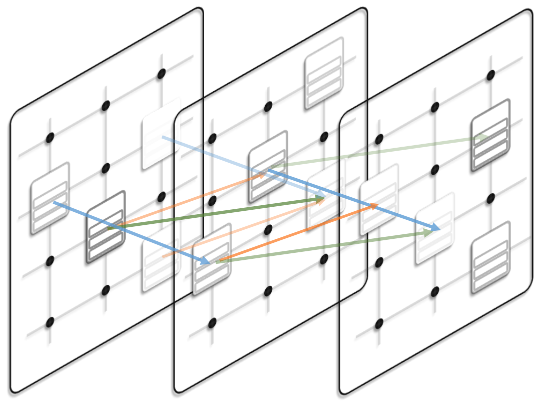


Figure 3: **NPL lattice:** Each slice corresponds to one timestep, and each node in a timestep corresponds to a given call depth, l, and elementary operation index, i. A subset of the lattice transitions are shown with blue arrows for `PUSH` transitions, green for `OP` and orange for `POP`.

	Blurred Stack	Blurred World	All Paths Return	Computational Cost	Gradient Accuracy
Execute All Paths	False	False	True	Highest	Exact
NPL	True	False	True	Medium	Medium
CTC+StackRNN	True	True	False	Lowest	Lowest

Table 1: Outlines the tradeoff between representational accuracy and computational cost for two extreme solutions and NPL.

The last remaining complexity is that λ does not indicate the necessary number of model timesteps. Thus the likelihood function must sum over all possible execution lengths up to some maximum T and ensure that the final action is a return, i.e. `POP`. If we define $I = |\lambda_o|$ then formally,

$$\mathcal{L}(\theta) = \log \sum_{t<T} p_a^t(\texttt{POP}) y_I^t$$

This gives a fully differentiable model for approximately maximizing the marginal probability of λ.

3.3 Lattice of Program States

Although the model we have defined so far is fully differentiable, the difficultly in training smoothed models of this form has been highlighted in the original Neural Turing Machine work (Graves et al., 2014) as well as much of the follow on work (Gaunt et al., 2016; Kurach et al., 2016; Graves et al., 2016; Neelakantan et al., 2016; Joulin & Mikolov, 2015). To help alleviate this difficulty, we introduce in this section the *neural lattice* structure after which Neural Program Lattices are named.

To motivate the need for this lattice, consider the set of possible program execution paths as a tree with a branch point for each timestep in the execution and a probability assigned to each path. Exact gradients could be computed by executing every path in the tree, calculating the gradient for each path, and then taking an average of the gradients weighted by the path probabilities. This solution is impractical however since it requires computation and memory that scales exponentially with the number of timesteps. To avoid this problem, the NTM and related techniques perform a single forward execution which is meant to approximately represent the simultaneous execution of all of the paths in the tree. To avoid the exponential explosion, the state at each timestep, i.e. tree depth, is approximated using a fixed-sized, representation. The approximation representation chosen by both NTM and Joulin & Mikolov (2015) is a soft-argmax of the states generated by performing each of the possible actions on the previous approximate state.

We observe that these two choices are really extreme points on what is a continuous spectrum of options. Instead of choosing to maintain a separate state representation for every path, or to group together all paths into a single representation, we can group together subsets of the paths and maintain an approximate state representation for each subset. This allows us to move along this spectrum, by trading higher memory and computational requirements for a hopefully closer approximation of the marginal probability.

In our implementation we group together execution paths at each timestep by call depth, $l \in L$, and number of elementary operations performed so far, $i \in I$, and maintain at each timestep a separate embedded state representation for each group of execution paths. Thus the unrolled linear architecture shown in Figure 1 becomes instead a lattice, as shown in Figure 3, with a grid of approximate program states at each timestep. Each node in this lattice represents the state of all paths that are at depth l and elementary operation i when they reach timestep t. Each node contains a soft-argmax of the stack states in M and S and an RNN cell identical to that in Figure 2[1]. For each node we must also compute $y_i^{t,l}$, the probability that at timestep t the execution is at depth l and at elementary operation i and has output the elementary operation sequence $\lambda_{1:i}$. As before we can compute this recursively as:

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

Similarly, the averaged call stack values are computed recursively as follows:

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

$$M_{d,i}^{t+1,l} = \begin{cases} \alpha_{i,i}^{t,l+1,l}(\texttt{POP})M_{1,i}^{t,l+1} + \alpha_{i-1,i}^{t,l,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)h_{out,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})0, & d=0 \\ \alpha_{i,i}^{t,l+1,l}(\texttt{POP})M_{2,i}^{t,l+1} + \alpha_{i-1,i}^{t,l,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)M_{1,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})h_{out,i-1}^{t,l-1}, & d=1 \\ \alpha_{i,i}^{t,l+1,l}(\texttt{POP})M_{d+1,i}^{t,l+1} + \alpha_{i-1,i}^{t,l,l}(\texttt{OP})p_{o,i-1}^{t,l}(\lambda_o^i)M_{d,i-1}^{t,l} + \alpha_{i,i}^{t,l-1,l}(\texttt{PUSH})M_{d-1,i}^{t,l-1}, & d>1 \end{cases}$$

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

We have left out the boundary conditions from the above updates for readability, the details of these are discussed in Appendix A.4.

Finally, the likelihood function approximately maximizes the probability of paths which at any timestep have correctly generated all elementary operations in λ, are currently at depth 0 and are returning from the current program. Formally,

$$\mathcal{L}(\theta) = \log \sum_{t \in T} p_{a,I}^{t,0}(\texttt{POP})y_I^{t,0}. \tag{3.4}$$

Remark: The specific choice to group by elementary operation index, and call depth was motivated by the representational advantages each provides. Specifically:

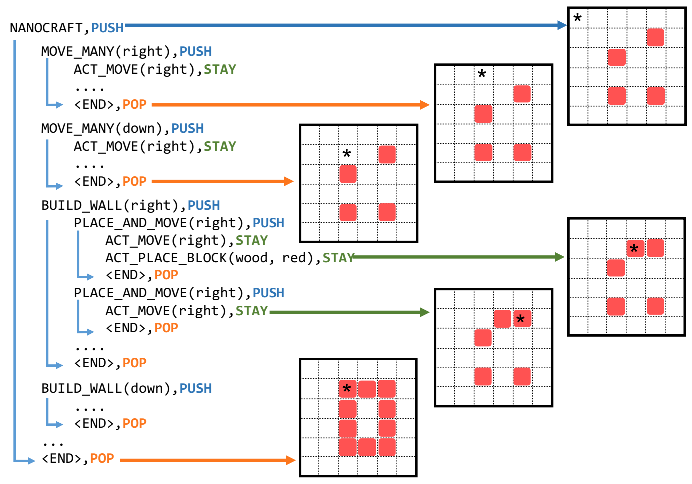
- **Grouping by elementary operation index:** allows the model to represent the input world state exactly instead of resorting to the fuzzy world state representation from equation 3.2.
- **Grouping by call depth:** allows the representation to place probability only on execution paths that return from all subprograms they execute, and return only once from the top level program as specified in equation 3.4.

Table 1 summarizes these advantages and the computational trade-offs discussed earlier.

Finally, in practice we find that values of the y's quickly underflow, and so we renormalize them at each timestep, as discussed in Appendix A.3.

4 Experiments

In this section, we demonstrate the capability of NPL to learn on both the long-hand addition task (ADDITION) from Reed & de Freitas (2016) and a newly introduced task involving arranging blocks in a grid-world (NANOCRAFT). We show that using the NPL to train with mostly the weak supervision of elementary operation traces, and very few full program traces, our technique significantly outperforms traditional sequence-to-sequence models, and performs comparably to NPI models trained entirely with the strong supervision provided by full program traces. Details of the experimental settings are discussed in Appendix A.5.



```
NANOCRAFT,PUSH
    MOVE_MANY(right),PUSH
        ACT_MOVE(right),STAY
        ....
        <END>,POP
    MOVE_MANY(down),PUSH
        ACT_MOVE(right),STAY
        ....
        <END>,POP
    BUILD_WALL(right),PUSH
        PLACE_AND_MOVE(right),PUSH
            ACT_MOVE(right),STAY
            ACT_PLACE_BLOCK(wood, red),STAY
            <END>,POP
        PLACE_AND_MOVE(right),PUSH
            ACT_MOVE(right),STAY
            <END>,POP
        ....
        <END>,POP
    BUILD_WALL(down),PUSH
        ....
        <END>,POP
    ...
    <END>,POP
```

Figure 4: **NanoCraft:** An illustrative example program, where the agent (denoted as "*") is required to build 3×4 rectangular red wooden building at a certain location in a 6×6 grid world. We can see that some of the blocks are already in place in the initial world-state. To build the building, the agent (program) first makes two calls to `MOVE_MANY` to move into place in the X and Y dimensions, and then calls `BUILD_WALL` four times to build the four walls of the building.

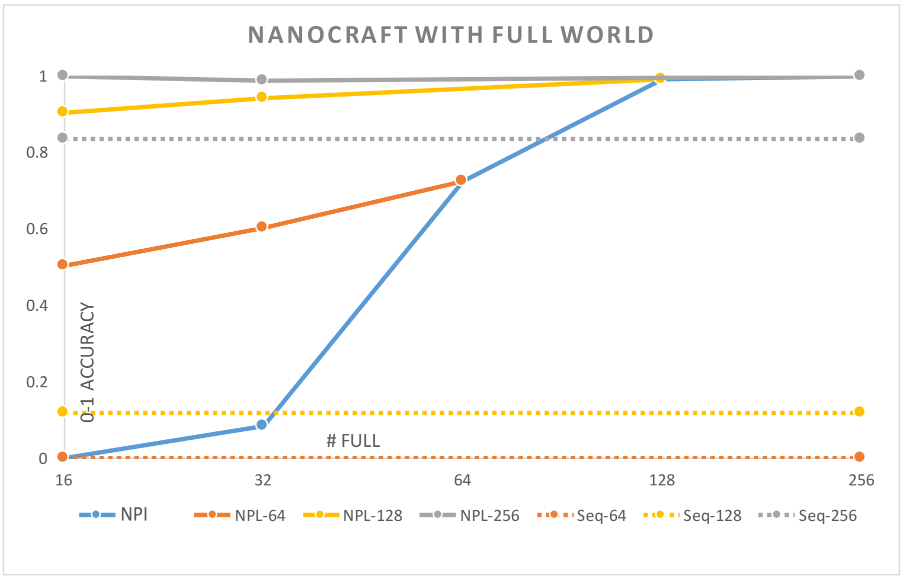
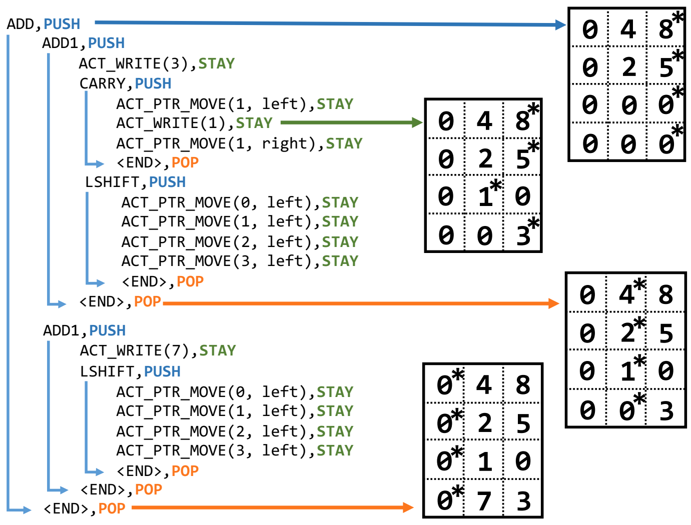


Figure 5: **NanoCraft Sample Complexity:** The x-axis varies the number of samples containing full program abstractions, while the y-axis shows the accuracy. *NPL-{64,128,256}* shows the accuracy of our model when trained with 64/128/256 training samples. *NPI* shows the accuracy of NPI, which can utilize only the samples containing full program abstractions. Finally, *Seq-{64,128,256}* shows the accuracy of a seq2seq baseline when trained on 64/128/256 samples. It's performance does not change as we vary the number of samples with full program abstractions since it cannot utilize the additional supervision they provide.

4.1 Sample Complexity

Task: We study the sample complexity using a task we call NanoCraft. In this task we consider an environment similar to those utilized in the reinforcement learning literature. The perceptual input comes from a 2-D grid world where each grid cell can be either empty or contain a block with both color and material attributes. The task is to move around the grid world and place blocks in the appropriate grid cells to form a rectangular building. The resulting building must have a set of provided attributes: (1) color, (2) material, (3) location, and sizes in the (4) X and (5) Y dimensions. As shown in the example in Figure 4, at each step the agent can take one of two primitive actions, *place a block* at the current grid cell with a specific color and material, or *move* in one of the four

[1] with additional indexes for i and l on all of the inputs and outputs.



```
ADD,PUSH
   ADD1,PUSH
      ACT_WRITE(3),STAY
      CARRY,PUSH
         ACT_PTR_MOVE(1, left),STAY
         ACT_WRITE(1),STAY
         ACT_PTR_MOVE(1, right),STAY
         <END>,POP
      LSHIFT,PUSH
         ACT_PTR_MOVE(0, left),STAY
         ACT_PTR_MOVE(1, left),STAY
         ACT_PTR_MOVE(2, left),STAY
         ACT_PTR_MOVE(3, left),STAY
         <END>,POP
      <END>,POP
   ADD1,PUSH
      ACT_WRITE(7),STAY
      LSHIFT,PUSH
         ACT_PTR_MOVE(0, left),STAY
         ACT_PTR_MOVE(1, left),STAY
         ACT_PTR_MOVE(2, left),STAY
         ACT_PTR_MOVE(3, left),STAY
         <END>,POP
      <END>,POP
   <END>,POP
```

Figure 6: **ADDITION:** An illustrative example program of the addition of 25 to 48. We have four pointers (denoted as "*"), one for each row of the scratch pad. We repeatedly call `ADD1` until we hit the left most entry in the scratch pad. Each call to `ADD1` we call `ACT_WRITE` to write the result, `CARRY` to write the carry digit (if necessary) and `LSHIFT` to shift all four pointers to the left to work on the next digit. The digit sequence on the fourth row of scratch pad is the result of the addition.

cardinal directions. We explored both a fully observable setting, and a partially observable setting. In the fully observable setting, the world is presented as a stack of 3 grids, one indicating the material of the block at each location (or empty), a similar one for color and a final one-hot grid indicating the agent's location. In the partially observable setting, the agent is provided only two integers, indicating the color and material of the block (if any) at the current location. Finally, in both settings the world input state contains an auxiliary vector specifying the five attributes of the building to be built. In each sample, a random subset of the necessary blocks have already been placed in the world, and the agent must walk right over these locations without placing a block.

Experiment Setup: We assume that data with full programmatic abstractions is much more difficult to obtain than data containing only flat operation sequences,[2] so we study the sample complexity in terms of the number of such samples. All experiments were run with 10 different random seeds, and the best model was chosen using a separate validation set which is one-quarter the size of the training set.

Results: Figure 5 shows the sample complexity for the NANOCRAFT task in the fully observable setting. We can see that *NPL* significantly outperforms the NPI baseline (*NPI*) when only a subset the total training samples have full abstractions. *NPL* similarly outperforms a sequence-to-sequence baseline (*Seq-**) trained on all of the available data. We also performed preliminary experiments for the partially observable setting, and obtained similar results.

4.2 GENERALIZATION ABILITY

Task: We study generalization ability using the ADDITION task from Reed & de Freitas (2016). The objective of this task is to read in two numbers represented as digit sequences and compute the digit sequence resulting from the summation of these two numbers. The goal is to let the model learn the basic procedure of long-hand addition: repeatedly add two one-digit numbers, write down the result (and the carry bit if necessary) and move to the left until the beginning of the numbers is reached. The whole procedure is represented using a four-row scratch pad, where the first and second rows are input digit sequences, the third row is the carry digit and the forth row the result. The model is provided a world-state observation which only provides a partial view into the full scratchpad state. Specifically, it is provided the integers at the location of four different pointers, each in one row of the scratchpad. The model has two possible elementary operations, either move a pointer left or right, or write a single digit into one of the four pointer locations. All four pointers start at the rightmost location (the least significant digit), and are gradually moved to the left by the

[2]Operation sequences can be obtained by observing a human demonstrating a task, whereas full abstractions require additional effort to annotate such traces.

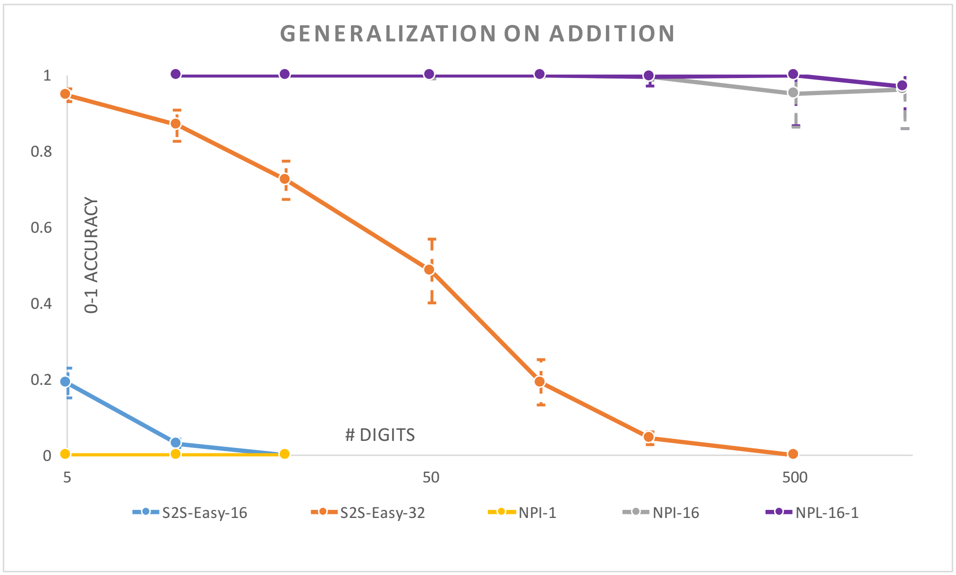


Figure 7: **ADDITION Generalization Performance:** The x-axis varies the number of input digits for the samples in the test set, while the y-axis shows the accuracy. All models are trained on addition programs with inputs of 1 to 10 digits. *NPL-16-1* shows the accuracy of our model when trained with 16 total samples (per number of digits), of which *1* sample (per number of digits) includes full program abstractions. *NPI-1* and NPI-16 show the accuracy of the NPI model when trained with 1 total samples and 16 total samples respectively (per number of digits), all containing full program abstractions. *S2S-Easy-16* and S2S-Easy-32 show the performance of the *S2S-Easy* baseline when trained with 16 and 32 samples respectively (per number of digits).

program throughout the execution. Figure 6 gives an example of a full program trace as well as state of the scratch pad at a particular timestep.

Experiment Setup: A primary advantage of learning programmatic abstractions over sequences is an increased generalization capability. To evaluate this, we train our model on samples ranging from 1 to 10 input digits . The training data contains an equal number of samples of each length (number of digits), and includes full program abstractions for ***only one*** randomly chosen sample for each length such that $|\texttt{FULL}| = 10$. We then test NPL using samples containing a much larger number of digits, ranging up to 1,000. On this task we found that both our model and the original NPI model were somewhat sensitive to the choice of initial seed, so we sample many different seeds and report both the mean and standard deviation, using a bootstrapping setup (Efron & Tibshirani (1994)) which is detailed in Appendix A.6.2.

Compared Models: We originally compared to a standard flat LSTM sequence model. However, we found that even with 32 samples per digit such a model was not able to fit even the training data for samples with more than 4 or 5 digits, so we did not present these results.[3]. Instead, we compare to a model called *S2S-Easy*, which is the strongest baseline for this task from (Reed & de Freitas, 2016). This model is custom-designed for learning addition and so it represents a very strong baseline. We discuss the model details in Appendix A.6.1. For completeness we also compare to a reimplementation of NPI in two different training regimes.

Results: Figure 7 shows the generalization capabilities of our model on the ADDITION task. Our model with "one-shot" strong supervision (*NPL-16-1*) significantly outperforms the *S2S-Easy* baseline even when the baseline is provided twice as many training samples (*S2S-Easy-32*). This is particularly notable given that the *S2S-Easy* model is specifically designed for the addition task. This result highlights the generalization capabilities our model brings by learning the latent structures which generate the observed sequences of elementary operations. Furthermore, we can see that

[3]This is consistent with the findings of Reed & de Freitas (2016)

these latent structures are learned mostly from the unlabeled sequences, since the vanilla NPI model trained with only 1 sample per digit (*NPI-1*) cannot generalize beyond the 10-digit data on which it was trained. Finally, we can see that just a single fully supervised sample is sufficient since it enables our model to perform comparably with a vanilla NPI model trained with `FULL` supervision for all samples (*NPI-16*).

5 Related Work

We have already discussed the most relevant past work upon which we directly build: CTC (Graves et al., 2006), StackRNNs (Joulin & Mikolov, 2015) and NPI (Reed & de Freitas, 2016).

Neural Programs Training neural networks to perform algorithmic tasks has been the focus of much recent research. This work falls into two main categories: weakly supervised methods that learn from input-output examples, and strongly supervised methods that additionally have access to the sequence of elementary actions performed to generate the output.

The work on learning neural programs from input-output data was sparked by the surprising effectiveness of the Neural Turing Machine (NTM) (Graves et al., 2014). Similar to NTMs, many of the proposed architectures have used differentiable memory (Kurach et al., 2016; Graves et al., 2016; Weston et al., 2014; Sukhbaatar et al., 2015b; Neelakantan et al., 2016; Gaunt et al., 2016; Feser et al., 2016), while others have used REINFORCE (Williams, 1992) to train neural networks that use sampling-based components to model memory access (Andrychowicz & Kurach, 2016; Zaremba & Sutskever, 2015). Some of this work has considered learning addition from input-output samples, a similar, but more challenging setup than our ADDITION domain. Zaremba & Sutskever (2014) makes use of a few training tricks to enable a standard LSTM to learn to add numbers up to length 9 when training on numbers of the same length. Kalchbrenner et al. (2015) proposes an architecture that is able to learn to add 15-digit numbers when trained on numbers of the same length. The *Neural GPU* model from (Kaiser & Sutskever, 2015) learns to add binary numbers 100 times longer than those seen during training, but requires tens of thousands of training samples and extensive hyperparameter searches. Additionally, using a decimal instead of binary representation with the Neural GPU model (as in our ADDITION task) is also reported to have a significant negative impact on performance.

The work on learning algorithms from sequence data has utilized both related techniques to ours as well as tackled related tasks. The most related techniques have augmented RNNs with various attention and memory architectures. In addition to those we have discussed earlier (Reed & de Freitas, 2016; Joulin & Mikolov, 2015), Grefenstette et al. (2015) proposes an alternative method for augmenting RNNs with a stack. From a task perspective, the most related work has considered variants of the scratchpad model for long-hand addition, similar or our ADDITION domain. This work has focused largely on more standard RNN architectures, starting with Cottrell & Tsung (1993), which showed that the standard RNN architectures at the time (Jordan, 1997; Elman, 1990) could successfully generalize to test samples approximately 5 times as long as those seen during training, if a few longer samples were included in the training set. More recently, Zaremba et al. (2015) showed that an RNN architecture using modern LSTM or GRU controllers can perfectly generalize to inputs 20 times as long as than those seen in the training data when trained in either a supervised or reinforcement learning setting. However this work was focused on trainability rather than data efficiency and so they utilized hundreds of thousands of samples for training.

NPI (Reed & de Freitas, 2016) and NPL distinguish themselves from the above work with the explicit modeling of functional abstractions. These abstractions enable our model, with only 16 samples, to perfectly generalize to data sequences about 100 times as long as those in the training data. Furthermore, concurrent work (Cai, 2016) has shown that an unmodified NPI model can be trained to perform more complex algorithms such as BubbleSort, QuickSort and topological sorting by learning recursive procedures, and we expect that our method can be directly applied to reduce the amount of needed supervision for these tasks as well.

Reinforcement Learning In the reinforcement learning domain the most related work to ours is the options framework, for building abstractions over elementary actions (Sutton et al., 1999). This framework bears many similarities to both our model and to NPI. Specifically, at each time step the agent can choose either a one-step primitive action or a multi-step action policy called an option. As with our procedures, each option defines a policy over actions (either primitive or other options) and terminates according to some function. Much of the work on options has focused on the tabular setting where the set of possible states is small enough to consider them independently. More recent work has developed option discovery algorithms where the agent is encouraged to explore regions that were previously out of reach (Machado & Bowling, 2016) while other work has shown the benefits of manually chosen abstractions in large state spaces (Kulkarni et al., 2016). However, option discovery in large state spaces where non-linear state approximations are required is still an open problem, and our work can be viewed as a method for learning such options from expert trajectories.

Much work in reinforcement learning has also considered domains similar to ours. Specifically, grid-world domains similar to NANOCRAFT are quite standard environments in the reinforcement learning literature. One recent example is Sukhbaatar et al. (2015a), which showed that even the strongest technique they considered struggled to successfully perform many of the tasks. Their results highlight the difficultly of learning complex tasks in a pure reinforcement learning setup. In future work we would like to explore the use of our model in setups which mix supervised learning with reinforcement learning.

6 CONCLUSION

In this paper, we proposed the Neural Program Lattice, a neural network framework that learns a hierarchical program structure based mostly on elementary operation sequences. On the NANOCRAFT and ADDITION tasks, we show that when training with mostly flat operation sequences, NPL is able to extract the latent programmatic structure in the sequences, and achieve state-of-the-art performance with much less supervision than existing models.

REFERENCES

Making neural programming architectures generalize via recursion. 2016. Under submission to ICLR 2017.

Marcin Andrychowicz and Karol Kurach. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*, 2016.

Garrison W Cottrell and Fu-Sheng Tsung. Learning simple arithmetic procedures. *Connection Science*, 5(1):37–58, 1993.

Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. Neural functional programming. *arXiv preprint arXiv:1611.01988*, 2016.

Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pp. 369–376. ACM, 2006.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 (7626):471–476, 2016.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1828–1836, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Michael I Jordan. Serial order: A parallel distributed processing approach. *Advances in psychology*, 121:471–495, 1997.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems*, pp. 190–198, 2015.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

Tejas D Kulkarni, Karthik R Narasimhan, Ardavan Saeedi, and Joshua B Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *NIPS*, 2016.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *ICLR*, 2016.

Marlos C Machado and Michael Bowling. Learning purposeful behaviour in the absence of rewards. *arXiv preprint arXiv:1605.07700*, 2016.

Microsoft Corp. Redmond WA. Kinect for Xbox 360.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *ICLR*, 2016.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. *ICLR*, 2016.

Sainbayar Sukhbaatar, Arthur Szlam, Gabriel Synnaeve, Soumith Chintala, and Rob Fergus. Mazebase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*, 2015a.

Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pp. 2440–2448, 2015b.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. *arXiv preprint arXiv:1511.07275*, 2015.

A Appendix

A.1 Dataset Details

Table 2 lists the set of programs and elementary operations we used to generate the data for ADDITION and NANOCRAFT. The programs and elementary operations for ADDITION are identical to those in Reed & de Freitas (2016). Note that when training with weak supervision the training data contains only the elementary operations and does not contain the programs or arguments.

Programs	**Description**	**Calls**
`ADD`	Multi-digit addition	`ADD1`
`ADD1`	Single-digit addition	`ACT_WRITE`/`CARRY`/`LSHIFT`.
`CARRY`	Write carry digit	`ACT_PTR_MOVE`/`ACT_WRITE`
`LSHIFT`	Shift four pointers left	`ACT_PTR_MOVE`
`ACT_WRITE`	Write result to environment	Elementary Operation
`ACT_PTR_MOVE`	Move pointer to left/right	Elementary Operation
`NANOCRAFT`	Build a rectangular fence	`MOVE_MANY`/`BUILD_WALL`
`MOVE_MANY`	Move multiple steps in one direction	`ACT_MOVE`
`BUILD_WALL`	Build a wall along one direction	`PLACE_AND_MOVE`
`PLACE_AND_MOVE`	Move one step and build a block	`ACT_MOVE`/`ACT_PLACE_BLOCK`
`ACT_MOVE`	Move one step to a direction	Elementary Operation
`ACT_PLACE_BLOCK`	Build a block at current location	Elementary Operation

Table 2: Programs, arguments and elementary operations used for generating training data of ADDITION and NANOCRAFT tasks.

A.2 Implementation Details

Here we describe the implementation details of the various component neural networks inside our implementation of the NPL. Note that the mappings are all the same for both ADDITION and NANOCRAFT except for f_{enc} which is task dependent.

- f_{enc} for ADDITION: We represent the environment observation, (latent) programs and arguments as one-hot vectors of discrete states. We feed the concatenation of one-hot vectors for environment observation and argument through a linear decoder (with bias) to get a unified arg-env representation. We then embed the programs (via f_{embed}) into an embedding space. Finally we feed the concatenation of arg-env vector and program vector through a 2-layer MLP with rectified linear (ReLU) hidden activation and linear decoder.
- f_{enc} for NANOCRAFT: We represent the environment observation as a grid of discrete states. Here we first embed each entry into an embedding space, and then feed this embedding through two convolutional layers and two MLP layers with ReLU hidden activation and linear decoder. We represent argument again as one-hot vectors and embed programs into an embedding space. Finally we feed the concatenation of argument vectors, convolutional vectors of environment observation and program vector through a 2-layer MLP with ReLU hidden activation and linear decoder.
- f_{lstm}: We employ a two-layer LSTM cell for the mapping. The size of the hidden states is set to 128 for both ADDITION and NANOCRAFT.
- f_{prog}: This mapping will map the LSTM hidden state to a probability distribution over programs. The hidden state output of f_{lstm} is mapped through a linear projection to an 8-dimensional space, and then another linear projection (with bias) with `softmax` generates p_g^t.
- f_{action} and f_{op}: Each of these encoders will output a probability distribution. We feed the top hidden states by f_{lstm} first through a linear projection (with bias) and then a `softmax` function to p_a^t and p_o^t respectively.

A.3 Normalization

When the operation sequence is too long, $y_i^{t,l}$ will become vanishingly small as t grows. To prevent our implementation from underflowing, we follow Graves et al. (2006) by renormalizing $y_i^{t,l}$ at each timestep and storing the normalized values and normalization constant separately. The new update rule becomes:

$$\overline{y}_i^{t+1,l} = [\![l < L]\!] p_{a,i}^{t,l+1}(\texttt{POP})\hat{y}_i^{t,l+1} + [\![0 < i]\!] p_{a,i-1}^{t,l}(\texttt{OP}) p_o^t(\lambda_o^i)\hat{y}_{i-1}^{t,l} + [\![0 < l]\!] p_{a,i}^{t,l-1}(\texttt{PUSH})\hat{y}_i^{t,l-1},$$

and we normalize the values and maintain a log-summation of the normalization constants

$$Y^t = Y^{t-1} + \log(\sum_{i,l} \overline{y}_i^{t,l}), \quad \hat{y}_i^{t,l} = \overline{y}_i^{t,l} / \sum_{i,l} \overline{y}_i^{t,l}.$$

Then the original update for y^{t+1} becomes

$$\log(y^{t+1}) = \texttt{log\_sum\_exp}(\log(y^t), \log(p_I^{t,0}(\texttt{POP})) + \log(\hat{y}_I^{t,0}) + Y^t),$$

the computation of which can be done robustly.

A.4 Update Equations with Boundary Conditions

In Section 3.3 we did not include the boundary conditions in our discussion to improve the readability. Our implementation, however, must account for the bounds on l, and i, as shown in Iverson brackets in the full update equations below:

$$\begin{aligned}
y_i^{t+1,l} =& [\![l < L]\!] p_{a,i}^{t,l+1}(\texttt{POP}) y_i^{t,l+1} + [\![0 < i]\!] p_{a,i-1}^{t,l}(\texttt{OP}) p_{o,i-1}^{t,l}(\lambda_o^i) y_{i-1}^{t,l} \\
& + [\![0 < l]\!] p_{a,i}^{t,l-1}(\texttt{PUSH}) y_i^{t,l-1} \\
\alpha_i^{t,l}(a) =& (y_i^{t,l} / y_i^{t+1,l}) p_{a,i}^{t,l}(a)
\end{aligned}$$

$$M_{d,i}^{t+1,l} = \begin{cases}
[\![l < L]\!]\alpha_i^{t,l+1}(\texttt{POP}) M_{1,i}^{t,l+1} + & [\![0 < i]\!]\alpha_{i-1}^{t,l}(\texttt{OP}) p_{o,i-1}^{t,l}(\lambda_o^i) h_{out,i-1}^{t,l} + \\
& [\![0 < l]\!]\alpha_i^{t,l-1}(\texttt{PUSH}) 0, \qquad d = 0 \\
[\![l < L]\!]\alpha_i^{t,l+1}(\texttt{POP}) M_{2,i}^{t,l+1} + & [\![0 < i]\!]\alpha_{i-1}^{t,l}(\texttt{OP}) p_{o,i-1}^{t,l}(\lambda_o^i) M_{1,i-1}^{t,l} + \\
& [\![0 < l]\!]\alpha_i^{t,l-1}(\texttt{PUSH}) h_{out,i-1}^{t,l-1}, \qquad d = 1 \\
[\![l < L]\!]\alpha_i^{t,l+1}(\texttt{POP}) M_{d+1,i}^{t,l+1} + & [\![0 < i]\!]\alpha_{i-1}^{t,l}(\texttt{OP}) p_{o,i-1}^{t,l}(\lambda_o^i) M_{d,i-1}^{t,l} + \\
& [\![0 < l]\!]\alpha_i^{t,l-1}(\texttt{PUSH}) M_{d-1,i}^{t,l-1}, \qquad d > 1
\end{cases}$$

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

A.5 Experimental Settings

As mentioned before, NPL can be trained jointly with full program abstractions (referred to as `FULL`) as well as elementary operation sequences (referred to as `OP`). When training with `FULL` samples, the training procedure is similar to that for NPI and we use this setting as one of our baselines. For each dataset on which we test NPL, we include mostly `OP` samples with only a small number of `FULL` samples. We pre-train the model solely on `FULL` samples for a few iterations to get a good initialization. After that, in each step we train with a batch of data purely from `FULL` or `OP` based on their proportions in the dataset and generate the parameter update in that step using the corresponding objective. For all tasks, we train the NPL using ADAM (Kingma & Ba, 2015) with base learning rate of 10^{-4} and batch size of 1. We decay the learning rate by a factor of 0.95 every 10,000 iterations. These settings were chosen using a manual search based on performance on the validation data.

A.6 Experimental Details for Addition

A.6.1 *S2S-Easy* Baseline

In our initial seq2seq baseline tests for Addition we represented the data for 90 + 160 = 250 as the sequence: 90X160X250 However, we found that such a model was not able to fit the training data even when trained with 32 samples per number of digits. So we instead compared to the much stronger *S2S-Easy* baseline presented in Reed & de Freitas (2016). This baseline makes it much easier to learn addition through the following two modifications to the model: 1) reverse input digits, and 2) generate reversed output digits immediately at each time step, such that the data sequence looks like: output: 052 input 1: 090 input 2: 061 This model is quite specific to the Addition task (and would not work on the NanoCraft task for instance) and results in a very strong baseline. None-the-less, as we showed in Figure 7, our model still significantly outperforms this baseline.

A.6.2 Bootstrapping

On the Addition task we found that both our model and the original NPI model were somewhat sensitive to the choice of initial seed. To test this sensitivity we ran our experiments for this task using a bootstrapping process (Efron & Tibshirani, 1994). We ran all models using 100 different seeds for each model. We then sampled 25 seed subsets, with replacement. For each subset, we choose the best seed using a validation set which was one-quarter the size of the original dataset, but consisted only of 10-digit samples. We performed this resampling procedure 100 times, and in Figure 7 we report the mean and standard deviation across the resampled seed sets.