ABSTRACT

Title of Thesis: **EFFECTIVENESS OF PROXIMAL POLICY OPTIMIZATION METHODS FOR NEURAL PROGRAM INDUCTION**

Runxing Lin,
Master of Computer Science, 2020

Thesis Directed By: Professor James A. Reggia
Department of Computer Science

The Neural Virtual Machine (NVM) is a novel neurocomputational architecture designed to emulate the functionality of a traditional computer. A version of the NVM called NVM-RL supports reinforcement learning based on standard policy gradient methods as a mechanism for performing neural program induction. In this thesis, I modified NVM-RL using one of the most popular reinforcement learning algorithms, proximal policy optimization (PPO). Surprisingly, using PPO with the existing all-or-nothing reward function did not improve its effectiveness. However, I found that PPO did improve the performance of the existing NVM-RL if one instead used a reward function that grants partial credit for incorrect outputs based on how much those incorrect outputs differ from the correct targets. I conclude that, in some situations, PPO can improve the performance of reinforcement learning during program induction, but that this improvement is dependent on the quality of the reward function that is used.

EFFECTIVENESS OF PROXIMAL POLICY OPTIMIZATION METHODS FOR NEURAL PROGRAM INDUCTION

by

Runxing Lin

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Computer Science
2020

Advisory Committee:
Professor James A. Reggia, Chair
Professor Dana Nau
Professor Garrett E. Katz

Acknowledgements

I express my gratitude to Professor James A. Reggia, my advisor, for helping me so much in my research and my master thesis.

I want to thank Professor Garrett E. Katz, for giving me suggestion about my research.

I want to thank Professor Dana Nau. He is the first professor who bring me into the area of Artificial Intelligent.

I want to thank Professor Yee Chiew. He is my advisor when I am in chemical engineering. It was Dr. Chiew who brought me into the area of scientific research.

I have been experienced so much warmth and love in University of Maryland, Computer Science Department. I want to say thanks to my colleagues and my friends.

Table of Contents

Table of Contents

List of Figures

1. Introduction

Neural program induction (NPI) has become a popular research area in recent years. NPI is the problem of training a neural network that can solve algorithmic problems, such as sorting or reversing a list, usually by providing a very large number of labeled input examples as training data. This has proven to be a very challenging task, but one that is very important for extending the range of abilities possessed by neural networks. It has been suggested that NPI can help to solve decision-making problems in robotics (Xu, et al., 2018). For example, planning a complex task with multiple actions under different conditions can be transferred to the problem of learning a policy to sort a sequence of actions regarding different states of situations. NPI can also help in visual navigation by breaking the visual navigation rules into state transitions (Zheng, et al., 2019), and by providing complex question-answering over knowledge bases by decomposing the task into a sequence of atomic actions (Ansari, 2019). In general, NPI can be a conducive paradigm for decomposing high-level tasks into program executions.

Training a neural network architecture to perform algorithmic problems is not as simple as mapping input to outputs in supervised learning. It requires the networks to have long-term memories about the complex relationship among all character symbols as well as fast and precise executions for each output. Most of the existing neural architectures designed to perform algorithmic tasks involve a *local representation* paradigm such as having symbols or variables represented by individual neurons (Abdelbar, et al., 2003), data structures represented by single time-

step neural activities rather than a temporal sequence (Plate, 1995), or requiring different architectures in different programming task (Dehaene & Changeux, 1997). Those neural architectures involving local representations are less neuro-biologically faithful and lack flexibility from an engineering perspective (Katz, et al., 2019). The Neural Virtual Machine (NVM) is a novel neural architecture that supports all of the functionalities of a traditional computer architecture (Katz, et al., 2019). The NVM is able to interpret and execute computer programs written in assembly-like language, using solely neural computational methods without local representation. This novel neurocomputational architecture shows great potential for performing NPI since its architecture can remain unchanged in different algorithmic problems or with different numbers of symbols to process.

Many of the potential applications of NPI, such as action planning, are more related to reinforcement-learning-based problems than supervised-learning-based problems. In spite of this, most current NPI approaches use a supervised learning paradigm. Recently, a reinforcement-based version of the neural virtual machine, referred to here as the NVM-RL, was designed to perform NPI using a reinforcement-learning paradigm (Katz, et al., 2020). The existing NVM-RL uses *standard policy gradient methods* with an all-or-nothing reward function, and this past work showed that the NVM-RL can perform NPI for some simple algorithmic problems such as selecting the maximum value of a numeric list or filtering out certain given values from a list. However, the NVM-RL failed to have consistent success in more complex problems like reversing a list.

Past results with the NVM-RL with NPI is still encouraging despite the inconsistent performance of NVM-RL in complex problems. One possible way to improve the NVM-RL is applying a more modern reinforcement learning method. The *standard policy gradient* (SPG) algorithm used in NVM-RL only allows one gradient update for each set of training data samples while sampling training data can be very computationally expensive. Proximal policy optimization (PPO) methods (Schulman, et al., 2017) are designed to address this issue by increasing the data efficiency of standard policy gradients, which allows a policy function to be updated multiple times under one set of sampled data. Here I explore the hypothesis that replacing SPG with PPO will help to improve the performance of NVM-RL, either by allowing NVM-RL to converge faster or by enabling it to solve larger or more complex problems.

In this work, I applied PPO to train the NVM-RL and compared the experimental results of PPO and SPG to each other. Surprisingly, I found that PPO did not have a substantial impact on improving the performance of NVM-RL using the existing all-or-nothing reward function. However, under a reward function that gives partial credits for partially correct outputs, the NVM using PPO was able to outperform SPG in complex problems like reversing or sorting a list. These results, while limited in scope, suggest that PPO combined with a more guided reward function has the potential to improve the effectiveness of reinforcement learning of neurocomputationally-supported programs. This work shows that using a modern reinforcement learning method does have a positive impact on NVM-RL,

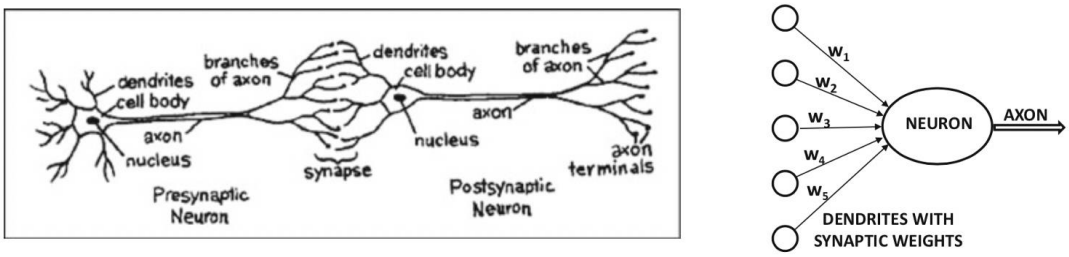
encouraging further research to improve the NVM-RL to achieve more complex NPI tasks by seeking some more appropriate reinforcement learning method.

2. Background

In this section, I briefly review past related work on neural networks, reinforcement learning, and the neural virtual machine, to provide important contextual information for the research done for this thesis.

2.1 Artificial Neural Networks

Artificial neural networks are computational algorithms that are inspired by biological neurons, and arguably simulate the information processing mechanisms of human brains (Aggarwal, 2018). The basic unit of an artificial neural networks is a simplified model of a neuron in a biological nervous system. In the human nervous system, neurons are connected by axons, dendrites and synapses, while in artificial neural networks, connections between units are much simpler and represented as matrices (or tensors) of numeric synaptic weights. Information flows in the human nervous system from neurons to neurons through electrochemical transmission, while the output and input of neurons in artificial neural networks are usually numerical values, which are scaled by the weight matrices involved. In the following description, I use the term “neural network” to refer to artificial neural networks unless explicitly noted otherwise.



(a) Biological neural networks *(b) A perceptron*

Figure 2.1 Biological neural networks and a single layer artificial neural network. (Aggarwal, 2018)

2.1.1 Feedforward networks

The perceptron (Rosenblatt, 1961), or single-layer neural networks in general, can be considered as the simplest neural networks. Elementary perceptron contains only one input layer of nodes that are assigned values and one output node, showed as Figure 2.1b. The mechanism of a perceptron can be represented as formula:

$$y = \phi(w \cdot x + b) \tag{2.1}$$

where x is the input vector, w is the weight vector, b is the bias, and ϕ is the activation function. There can be different kinds of activation functions, the most common being a linear threshold unit in the form of a 0/1 step function, but other common ones are shown here:

Sign $$\Phi(in_x) = sign(in_x) \tag{2.2}$$

ReLu $$\Phi(in_x) = max(in_x, 0) \tag{2.3}$$

Sigmoid $$\Phi(in_x) = \frac{1}{1 + e^{-in_x}} \tag{2.4}$$

Hyperbolic Tangent $$\Phi(in_x) = \frac{e^{in_x} - e^{-in_x}}{e^{in_x} + e^{-in_x}} \tag{2.5}$$

More generally, a feedforward network (Haykin, 2007) contains more than one layer of neurons. It consists of a set of input neurons (not counted as a layer), one or more hidden layers, and an output layer. Each neuron in the hidden layers and output layer uses an activation function. An example of the architecture of a feedforward network is shown in figure 2.2, where in general the output layer consists of more than one node.

The mechanisms for computing the output for each neuron in feedforward networks are similar to those of Equations 2.1 – 2.4. The transformation from the input layer to the output layer can be expressed as the following composed equations:

Input layer to hidden layer $$h_1 = \Phi(W_1 x) \tag{2.5}$$

Hidden layer to hidden layer $$h_{p+1} = \Phi\left(W_{p+1} h_p\right) \tag{2.6}$$

Hidden layer to output layer $$o = \Phi(W_{n+1} h_n) \tag{2.7}$$

where h refers to the activation of hidden layers, o refers to the activation of the output layer, and W refers to the weight matrix for each layer. A softmax function can be applied on the output layers, normalizing the output of each node to be between 0 and 1 and the sum of all nodes in the output layer to be 1. This can be used to represent the likelihood or each node in a classification problem:

Softmax $$\Phi(v)_i = \frac{e^{v_i}}{\sum_{j=1}^{K} e^{v_j}} \tag{2.8}$$

where, in the softmax activation function, the input is a vector v with K dimensions.

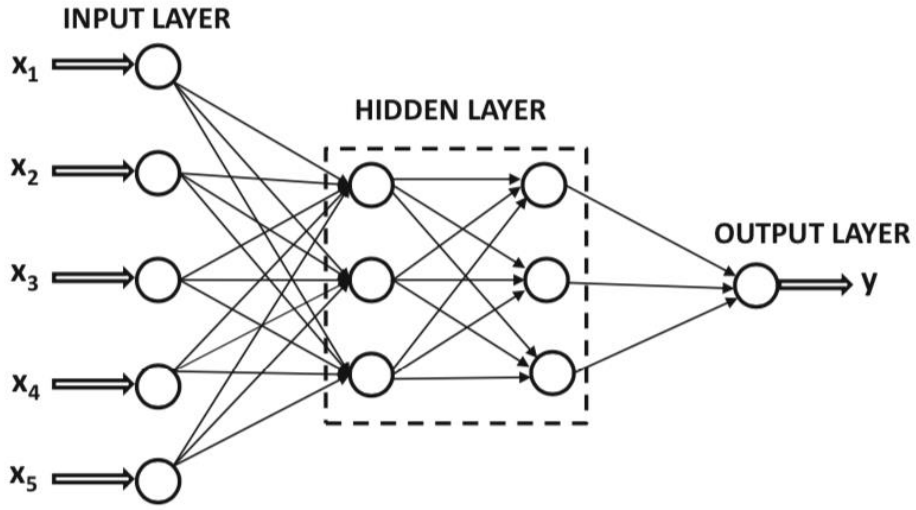


Figure 2.2 An example of a feedforward network (Aggarwal, 2018)

2.1.2 Gradient Descent and Error Backpropagation

In supervised learning, the task of training a neural network usually refers to minimizing the differences between target vectors in the training data and the network outputs for this same data, which is called the *error* or *loss*. We usually define a loss function to represent the error of the neural networks so that the training task is to minimize the loss function or objective function, ultimately over test data that has been held out and not used during training. Two commonly used loss functions *L* are given here:

Mean Square Error

$$L = \sum_{i=1}^{n} \frac{(\theta(x_i) - t)^2}{n} \tag{2.9}$$

Cross Entropy Loss

$$L = -\Big(t_i log\big(\theta(x_i)\big) + (1 - t_i) log\big(1 - \theta(x_i)\big)\Big) \tag{2.10}$$

where t_i refers to the target output and $\theta(x_i)$ refers to the actual output of the neural network; n refers to the total number of training examples. The mean square error

(MSE) function is usually used for regression problems while cross entropy loss is often used on classification problems.

To minimize the objective function, we usually use a stochastic gradient descent method (Ruder, 2016). Gradient descent is a function optimization algorithm to find the local minimum of a function. In gradient descent, for each step we move the variables negatively proportional to the gradient of the function until we reach the local minimum. Mathematically this can be written as:

$$W = W - \eta \nabla \mathrm{L}(W) \tag{2.11}$$

Here W refers to the weights of the neural network, η refers to the learning rate, and $\nabla \mathrm{L}(W)$ refers to the gradient of the loss function in the weight space, which can be written as:

$$\nabla L(W) = \begin{bmatrix} \vdots \\ \dfrac{\partial L}{\partial w_{ij}} \\ \vdots \end{bmatrix} \tag{2.12}$$

where w_{ij} refers to each individual weight of the neural network, specifically from node j to node i. Therefore, *Equation 2.11* can be expressed as:

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{2.13}$$

$$\Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}} \tag{2.14}$$

Applying the chain rule to the derivation, we get:

$$\Delta w_{ij} = \eta \delta_i a_j \tag{2.15}$$

where a_j refers to the activation level of node j, δ_i refers to the error signal at node i.

Error backpropagation (Kelley, 1960) is usually the learning rule used to train a neural network, which is based on a combination of gradient descent and using the chain rule. The error of the output node is straightforward to calculate. However, to calculate the error of the hidden layer, we need to propagate the error back from the output layers. Figure 2.3 shows the basic concept of backpropagation.

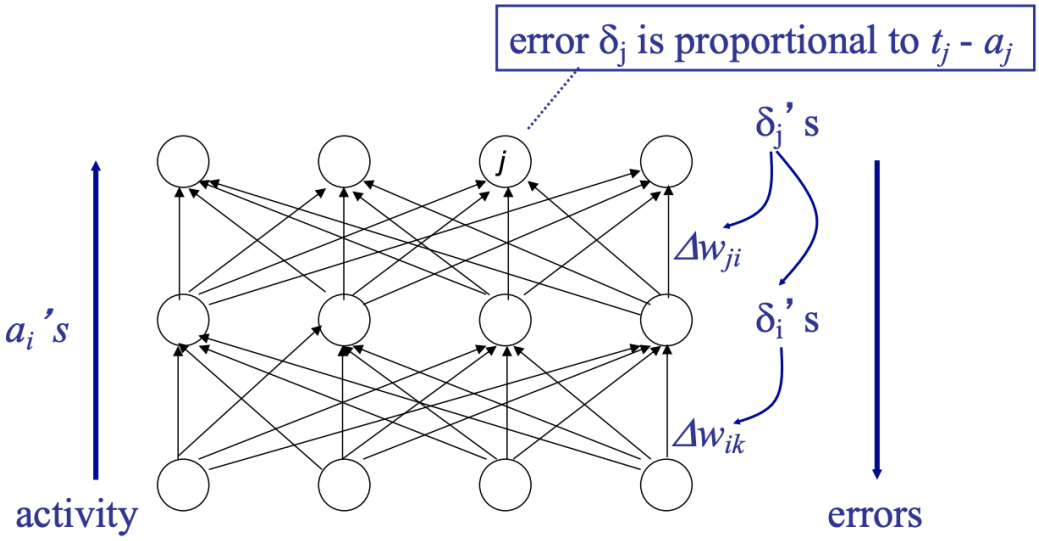


Figure 2.3 Error Backpropagation

Using the concept of backpropagation, we can compute the δ_i values for nodes in the hidden and output layer as:

For output nodes $$\delta_i = (t_i - a_i) f'(a_i) \tag{2.16}$$

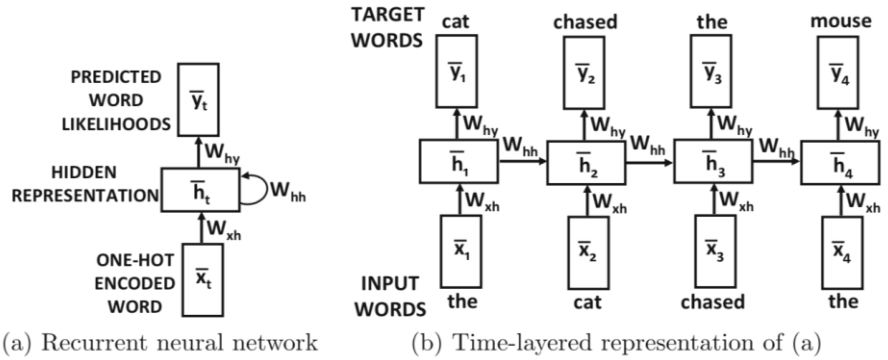
For hidden nodes $$\delta_i = \sum w_{ji} \delta_j f'(a_i) \tag{2.17}$$

Here δ_j refers to the error of the nodes that node i is directed to.

2.1.3 Recurrent Neural Networks

The neural virtual machine used in this work makes use of not only feedforward networks, but also recurrent neural networks (RNN), so we briefly consider them here. In reinforcement learning with the neural virtual machine, RNNs are used as an agent's policy generator.

The inputs of a feedforward network are generally a fixed-length vector. Therefore, feedforward networks are not well suited for analyzing time-series data or sets or sentences having different lengths. Such temporal data is usually the domain of RNNs which have recurrent connections (loops, cycles) and allow sequential inputs and outputs. The current output or an RNN is derived not only from the current input but also from a hidden state which contains the memory of previous states. Figure 2.4a shows an abstract representation of a simple RNN and Figure 2.4b shows the RNN unfolded in time.



(a) Recurrent neural network (b) Time-layered representation of (a)

Figure 2.4 Recurrent Neural Networks. (a) Network architecture. (b) The network unfolded in time. (Sutton & Barto, 2018)

Taking a sequence as an input to an RNN, the position of each element of the input sequence can refer to the time-step at which that element serves as input to the network. The mechanism of the RNN in each state or time-step can be expressed as:

$$h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1}) \tag{2.18}$$

$$y_t = W_{ht}h_t \tag{2.19}$$

where h_t refers to the hidden state of the current step and h_{t-1} refers to the hidden state of the previous step. The activation function $tanh$ can be replaced by other activation functions such logistic functions.

To train an RNN, we often use backpropagation through time (BPTT) (Werbos, 1990), which is based on propagating the errors back to the previous time steps. Mathematically this can be written as:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{xh}^{(t)}} \tag{2.20}$$

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hh}^{(t)}} \tag{2.21}$$

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hy}^{(t)}} \tag{2.22}$$

2.2 Reinforcement Learning

Reinforcement learning is a machine learning area focused on training an agent to take actions in different situations so that the cumulative reward signals collected from the environment are maximized. Figure 2.5 shows a typical agent-environment-interaction problem in reinforcement learning. In each time step, an

agent takes an *action* towards the environment and observes a new *state* as well as receives a *reward* signal from the environment. The agent then determines which action to take for next time step based on the rewards and state signal it has received. Here we just consider reinforcement learning problems that can be viewed as a Markov Decision Process (MDP) (Wei, et al., n.d.), which is a framework in which sequential states depend just on the preceding state and action taken by the agent.

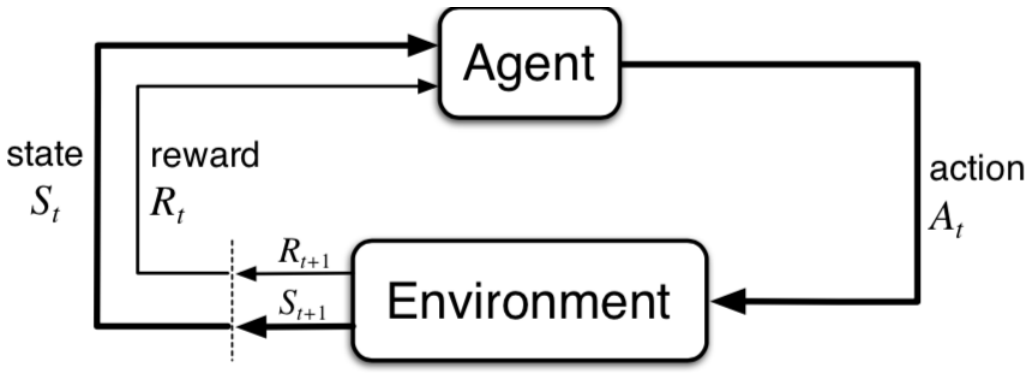


Figure 2.5 The traditional agent-environment interaction used in reinforcement learning. (Sutton & Barto, 2018)

In addition to agent and environment, reinforcement learning usually includes another four sub-elements: a *policy* $\pi(s)$, determining which action to take at each time step; a *reward signal* R_t, essentially defining the goal of the reinforcement learning problem and usually sent from the environment to the agent; a *value function* $V(s)$, calculating the value of a state, usually by predicting the future rewards that can be obtained by the agent from the state; and (in model-based RL) finally a *model of the environment*, which mimics the true environment and is used by the agent to predict what states it can possibly get to after taking an action.

The methods used to solve reinforcement learning problems can be divided into two main parts: Tabular Solution Methods and Approximate Solution Methods.

Tabular solution methods are usually used in small problems with finite action spaces and finite state spaces. Approximate solution methods are mostly used in problems with large action or state spaces and involve training a function estimator that replaces the state-action table to evaluate the V or Q-values or to represent a policy.

2.2.1 Key concepts in reinforcement learning

The *model* of the environment should contain at least two elements: the state-transition function T and the reward function R,

$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] \tag{2.23}$$

$$R(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{2.24}$$

The transition function calculates the probability that the agent will go to state s' after taking action a at state s. The reward function calculates the expectation of the reward of taking action a in state s.

Not every reinforcement learning algorithm requires the agent to have a built-in model of the environment. However, the transition function and reward function are also implicitly learned in algorithms without a model of the environment, such as with policy gradient methods.

A *policy* of an agent determines how the agent takes actions in different states. A policy can be either deterministic or stochastic; mathematically it can be written as:

Deterministic $$\pi(s) = a \tag{2.25}$$

Stochastic $$\pi(a_i|s_i) = \mathbb{P}[A = a_i|S = s_i] \tag{2.26}$$

Here, $\mathbb{P}[A = a_i|S = s_i]$ refers to the probability that the agent takes action a_i when in state s_i.

A *value function* of an agent evaluates the goodness of a state, usually by calculating/estimating the future reward that the agent can get starting in that state and subsequently following its policy. The future reward, also refers to the *return*, after time-step *t* can be define as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.27}$$

where γ is the discount factor used to determine the weight of future rewards. Based on the return G_t, the state-value function of a policy π is the expectation of the future reward of the current state:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.28}$$

Similarly, the action value function, also referred to as the *Q-value*, calculates the expectation of the return at state s if the agent takes action a:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.29}$$

Based on equation 2.26 and 2.29 the state value function can then be defined as:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s, a)\, \pi(a|s) \tag{2.30}$$

and the *advantage function* can then be defined as:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{2.31}$$

which refers to the advantage of taking action a, at state s.

When the state space or action space is continuous, $Q_\pi(s,a)$, $V_\pi(s)$, and $A_\pi(s,a)$, are each usually represented as an approximate function that can be trained by sampling large amounts of data through practicing via agent-environment interactions.

2.2.2 Policy Gradient Methods

2.2.2a Standard Policy Gradient

A *policy gradient method* is an algorithm in reinforcement learning that learns the policy π directly without learning the state-value function $V_\pi(s)$ or the action-value function $Q_\pi(s,a)$ (Sutton, et al., 2000). With policy gradients, the policy is a parameterized function with parameters θ, written as $\pi(a_i|s_i,\theta)$. Policy $\pi(a_i|s_i,\theta)$ should usually be differentiable with respect to θ. The NVM-RL currently uses a policy gradient method for program induction, so we briefly review this concept here.

With a policy gradient method, we usually measure the performance of the policy by an objective function $J(\theta)$. Then we perform *gradient ascent* on the function $J(\theta)$ similar to with Equation 2.11:

$$\theta_{t+1} = \theta_t + \eta\nabla J(\theta) \tag{2.32}$$

The objective function $J(\theta)$ can be defined as the state value function of the beginning state:

$$J(\theta) = V_\pi(s_0) \tag{2.33}$$

By the policy gradient theorem (Sutton & Barto, 2018), $\nabla J(\theta)$ can be calculated as:

$$\nabla J(\theta) = \mathbb{E}_{\pi}[\nabla ln(\pi(a|s,\theta))Q_{\pi}(s,a)] \tag{2.34}$$

The objective function $J(\theta)$ and its gradient can vary in different ways by switching the Q-value function with the advantage function or other functions or values that represents the change of total rewards by taking an action:

$$\nabla J(\theta) = \mathbb{E}_{\pi}\big[\nabla ln\big(\pi(a|s,\theta)\big)A_{\pi}(s,a)\big] \tag{2.35}$$

$$\nabla J(\theta) = \mathbb{E}_{\pi}\nabla ln\big(\pi(a|s,\theta)\big)\sum_{t=t\prime}^{T} R_t \tag{2.36}$$

In equation 2.36, $\sum_{t=t\prime}^{T} R_t$ refers to the method "reward-to-go," which stands for the state-action value at a current time-step $t^{'}$ and is represented by the cumulative rewards from t' to the terminal time-step T (Peters & Schaal, 2006).

It has been shown that policy gradient methods generally work better by comparing action values with a baseline value $b(s)$, where $b(s)$ can be any arbitrary value:

$$\nabla J(\theta) = \mathbb{E}_{\pi}[\nabla ln\big(\pi(a|s,\theta)\big)\big(Q_{\pi}(s,a) - b(s)\big)] \tag{2.37}$$

2.2.2b Proximal Policy Optimization (PPO)

In this thesis work, I compare the use of proximal policy optimization (PPO) for the first time versus the standard policy gradient method described above in using the NVM for program induction during reinforcement learning.

A major limitation of the standard policy gradient method of Equation 2.32 is that it can only be executed one time per set of data sampled by policy π_θ. This is because once Equation 2.32 is executed, the parameter of the policy θ becomes θ_{new} and we have to resample training data from the viewpoint of the new policy $\pi_{\theta_{\text{new}}}$. Sampling training data from agent-environment interactions is fairly computationally expensive. Investigators have tried various ways to enable the policy gradient method to update θ_{new} with the old data sampled from the old policy $\pi_{\theta_{\text{old}}}$, and proximal policy optimization (PPO) methods provide a simple implementation of this with great experimental results.

PPO (Schulman, et al., 2017) defines a quantitative measurement for measuring the difference between two policies π_θ and $\pi_{\theta_{\text{old}}}$, which is the probability ratio between the two policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{2.38}$$

Given this, the objective function for PPO is defined as:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{2.39}$$

where A_t is the function estimator for the *advantage function* at time step t, ϵ is an arbitrary hyperparameter, and the output of the function $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ is:

$$\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) = \begin{cases} 1+\epsilon & \text{if } r_t(\theta) > 1+\epsilon \\ r_t(\theta) & if\ (1-\epsilon) < r_t(\theta) < (1+\epsilon) \\ 1-\epsilon & \text{if } r_t(\theta) < 1-\epsilon \end{cases} \tag{2.40}$$

The function $\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$ then selects the minimum of the two objects. The ratio is then bounded by $(1+\epsilon)$ if $A_t > 0$ and $(1-\epsilon)$ if $A_t < 0$.

Under this objective function, the policy π_θ can be updated using the data sampled by $\pi_{\theta_{old}}$. Each time the parameters θ can be updated as was done with Equation 2.32:

$$\theta_{new,t+1} = \theta_{new,t} + \eta \nabla L^{CLIP}(\theta) \tag{2.41}$$

where θ_{new} refers to the parameters in the new policy $\pi_{\theta_{new}}$. Equation 2.41 can be executed several times without sampling a new advantage function estimator A_t by the new policy, which largely improve the data efficiency.

The original paper describing PPO (Schulman, et al., 2017) ran several computational experiments to compare PPO with other existing reinforcement learning methods. The results showed that PPO outperformed other methods within a 1 million time-steps framework. Their results are shown in Figure 2.6.

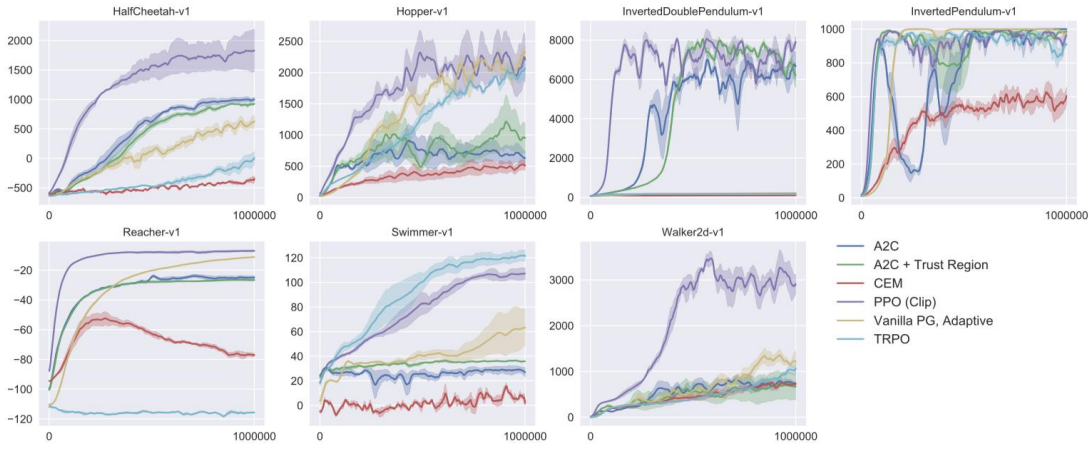


Figure 2.6 The comparison between PPO and other methods over 1 million time-steps (Schulman, et al., 2017). The vertical axes represent the rewards each algorithm achieved in a specific game while the horizontal axes represent the number of time-steps. The purple curve shows the performance of

PPO, which shows better results than the other algorithms in multiple tasks. A2C refers to an actor-critic algorithm; CEM refers to cross entropy method; TRPO refers to Trust Region Policy Optimizations (Schulman, et al., 2015); Vanilla PG refers to Vanilla Policy Gradient. (Mnih, et al., 2016)

2.3 The Neural Virtual Machine

The NVN is a purely neural architecture that can emulate the functionality of a traditional Harvard architecture of a computer (Katz, et al., 2019). The NVM uses neural layers to emulate the registers of traditional computer where register values are represented as distributed activity patterns over the corresponding neural layers. Values transferred through registers and tape-based memory are emulated via associative recall and associative learning through the pathways between the neural layers. Multiplicative weight vectors determine whether or not associative learning or associative recall happens in those pathways (Mehaffey, et al., 2005) (Salinias & Sejnowski, 2001). Figure 2.7 shows a simplified rendition of the internal neural architecture of the part of the NVM that serves as a working memory for NVM-RL (Katz, et al., 2020).

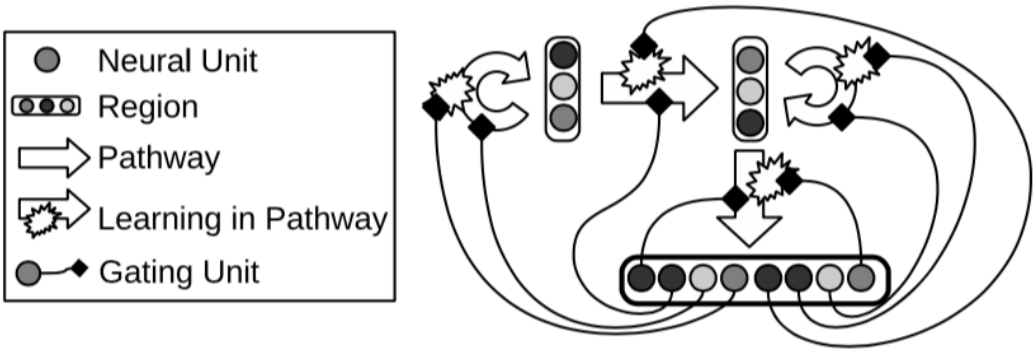


Figure 2.7 Architecture of the NVM's working memory. Each region represents a register or the tape-based memory of a traditional computer. The bottom region represents a gated vector to control if associative recall or associative learning happens in each pathway. (Katz, et al., 2019)

NVM-RL can be divided into two parts: a portion representing working memory $\mathcal{L}$ and a controller $\mathcal{C}_\theta$ where θ represents the parameters in the controller. $\mathcal{L}$ consists of multiple neural activity layers, and those layers are fully connected to each other by a set of pathways $\mathcal{P}$. An environmental function $\mathcal{E}$ is used to manipulate the inputs and outputs of the working memory. The controller outputs multiplicative gate vectors u_t^p and ℓ_t^p to determine whether associative learning and recall should happen in any corresponding pathway p based on a hidden vector h_t and the activity patterns $\{v_t^p\}_{q\in\mathcal{L}}$ received back from the working memory. Figure 2.9 shows the overall structure of NVM-RL.

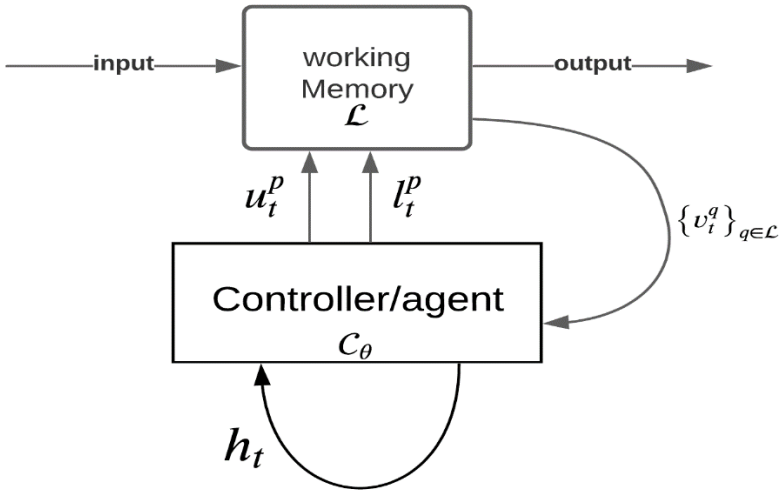


Figure 2.8 Overall structure of NVM-RL. A working memory serves as registers in a traditional computer architecture and a controller is trained about how to manage the register actions that solve the algorithmic problems to be learned.

2.3.1 Working Memory of NVM-RL

The working memory consists of multiple neural activity layers. Those layers can emulate the "registers" and "memory address" of a typical computer architecture. Neural activity layers are connected to each other through pathways.

At each time step, v_t^q can be updated by *associative recall* from neural layer r_p and the corresponding pathway p_r^q, which can emulate the movement of register contents from r_p to r_q, Mathematically,

Associative recall:
$$\hat{v}_{t+1}^q = tanh\left(\sum_{p\in\mathcal{P}_q} W_t^p \, v_t^{r_p} u_t^p\right) \tag{2.42}$$

where $\mathcal{P}_q$ is the set of pathways connecting to the layer q, $v_t^{r_p}$ denotes the activity vector at pathway r_p, which denotes the source layer of the pathway p; u_t^p denotes the

multiplicative gates, which is a vector of 0 and 1 elements with one position per pathway connected to layer q and only the position of the chosen pathway to proceed with associative recall is a 1.

W_t^p is the weight matrix for each pathway p, which can also be updated at a given time step. The changes during this step define the associative learning on pathway p and involve a novel *store-erase learning rule* first introduced in the NVM that simultaneously stores one association while erasing another, as follows:

Associative learning:

$$W_{t+1}^p = W_t^p + \Delta W_t^p \ell_t^p \tag{2.43}$$

$$\Delta W_t^p(x, y) = \left(tanh^{-1}(y) - W_t^p x\right)x^{\intercal}/(N_x \rho^2) \tag{2.44}$$

where N_x is the number of neurons in layer x, and the ΔW_t^p of a pathway from layer r_p to layer q_p is calculated by equation 2.44 as $\Delta W_t^p(r_p, q_p)$. Here ℓ_t^p is a vector of multiplicative gates like u_t^p, choosing the pathway that will have its weights updated as the one with $\ell_t^p = 1$.

The NVM-RL expects that its inputs and outputs will be fixed-length sequences and that each element in a sequence is a character symbol a in a finite space $\mathcal{A}$. Those symbols are represented as distributed neural patterns that are stored in the neural activity layers. As noted earlier, $v^q[a]$ denotes a pattern a stored in layer q and v_t^q denotes activity vector at time step t in layer q. Moving a pattern $v^{r_p}[a]$ from register r_p to register q can be simply obtained by associative recall as in equation 2.42 if W_t^p has been updated by associative learning with $\Delta W_t^p(a, a)$.

Therefore, before we train the NVM-RL, each pathway between registers will be initialized with Algorithm 2.1:

Algorithm 2.1 Initial associative learning for all pathways

For $\boldsymbol{p} \in \boldsymbol{\mathcal{P}}$ do:
    Initialize $\boldsymbol{W_0^p}$ with all zero entries,
    For $\boldsymbol{a} \in \boldsymbol{\mathcal{A}}$**:**
        Update $\boldsymbol{W_0^p = W_0^p + \Delta W_0^p(a, a)}$
    End for
End for

A neural layer can also be used as a tape-based memory, denoted as v^m, which is usually set up with two recurrent pathways: inc_m and dec_m, which are used to increase or decrease the current position in the memory. A memory position can also be represented by a pattern k, denoted as $v^m[k]$. The head address of memory $v^m[k]$ can be updated from $v^m[k]$ to $v^m[k+1]$ through pathway inc_m if W^{inc_m} has been updated by associative learning with $\Delta W_0^{inc_m}(k, k+1)$. We use the input pattern to serve as a memory address pattern. The recurrent pathways inc_m and dec_m are then initialized with Algorithm 2.2.

Algorithm 2.2 Initialize tape-based memory pathway

Initialize $\boldsymbol{W_0^{inc_m}}$ and $\boldsymbol{W_0^{dec_m}}$ with all zero entries
For $\boldsymbol{k} \in \boldsymbol{\mathcal{A}}$ do**:**
    Update $\boldsymbol{W_0^{inc_m} = W_0^{inc_m} + \Delta W_0^{inc_m}(k, k+1)}$
    Update $\boldsymbol{W_0^{dec_m} = W_0^{dec_m} + \Delta W_0^{dec_m}(k, k-1)}$
End for

2.3.2 The controller network

The controller $\mathcal{C}_\theta$ is a simple recurrent neural network. For each time step, the controller is updated by:

$$h_t = tanh\left(\sum_{q\in\mathcal{L}} W^{h,q} v_t^q + W^{h,h} h_{t-1} + b^h\right) \tag{2.45}$$

$$l_t = \sigma(W^{l,h} h_t + b^l) \tag{2.46}$$

$$u_t^{\mathcal{P}_q} = \mu(W^{q,h} h_t + b^q) \tag{2.47}$$

where σ is a hyperbolic tangent activation function and μ is a softmax activation function. Here l_t is an activity pattern with one neuron per pathway p, and the value at each of the neurons denotes the probability that weight updating occurs at each corresponding pathway. Also, $u_t^{\mathcal{P}_q}$ is the probability that associative recall will happen in the pathways $\mathcal{P}_q$ that connect to layer q. The multiplicative gates can then be sampled by:

$$\ell_t^p \sim \mathcal{B}(l_t) \tag{2.48}$$

$$p_t^q \sim \mathcal{M}\left(u_t^{\mathcal{P}_q}\right) \tag{2.49}$$

$$u_t^p = \begin{cases} 1 & if\ \exists q: p = p_t^q \\ 0 & else \end{cases} \tag{2.50}$$

where $\mathcal{B}$ and $\mathcal{M}$ denote Bernoulli and multinomial distributions, respectively. The sampling makes each layer q be updated by only one associative pathway at a certain time-step *t*.

2.3.3 Overall Mechanism of the NVM-RL

Combining the mechanisms of the working memory and the controller, the operation of the NVM-RL at one time-step can be characterized by four sub-steps:

Step 1: $$\{u_t^p\}_{p\in\mathcal{P}}, \{\ell_t^p\}_{p\in\mathcal{P}}, h_t = \mathcal{C}_\theta(\{v_t^p\}_{q\in\mathcal{L}}, h_{t-1}) \tag{2.51}$$

Step 2: $$\hat{v}_{t+1}^q = tanh\left(\sum_{p\in\mathcal{P}} W_t^p \, v_t^{r_p} u_t^p\right) \tag{2.52}$$

Step 3: $$W_{t+1}^p = W_t^p + \Delta W_t^p \ell_t^p \tag{2.53}$$

Step 4: $$\{v_{t+1}^p\}_{q\in\mathcal{L}}, \psi_{t+1} = \mathcal{E}(\{\hat{v}_{t+1}^p\}_{q\in\mathcal{L}}, \psi_t) \tag{2.54}$$

In step 1, the controller assigns the multiplicative gates values u_t^p and ℓ_t^p based on the activity vectors of the memory layers and the hidden vector of the controller. In step 2, the chosen memory layers are updated by associative recalls, where $\hat{v}_{t+1}^q$ denotes the activity vector at layer q before it is modified by the environment function $\mathcal{E}$ at time step $t+1$, and $v_t^{r_p}$ denotes the activity vector at layer r_p which is the source of the pathway p. In step 3, weight matrixes of the associative pathways are updated. Finally, in step 4, the environment function modifies the activity vector for each memory layer based on external inputs.

2.3.4 Policy Gradient in NVM-RL

The NVM-RL uses standard policy optimization for training the controller to have desired target outputs. The NVM-RL expects inputs and outputs to be fixed length sequences. The environment function modifies an input sequence to feed into a designated input layer. The NVM-RL is supposed to produce a correct sequence at a

designated output layer. If NVM-RL is expected to have a target output list $['1','2','0','3']$ from timestep t to t+3, the designated output layer is supposed to have $v_t^{out} = v_t^{out}[1], v_{t+1}^{out} = v_{t+1}^{out}[2], v_{t+2}^{out} = v_{t+2}^{out}[0], v_{t+3}^{out} = v_{t+3}^{out}[3]$. In the training process, the output will be compared with the target and receive a reward. The NVM-RL will only receive a reward r_t at the final time-step.

Under the reinforcement learning paradigm, the controller is regarded as an agent, the gating decision at each time-step t is regarded as an action:

$$a_t = \{u_t^p\}_{p\in\mathcal{P}} \cup \{\ell_t^p\}_{p\in\mathcal{P}} \tag{2.55}$$

The set of activity vectors in working memory is regarded as the state observation at time t,

$$s_t = \{v_t^p\}_{q\in\mathcal{L}} \tag{2.56}$$

The hidden states of the controller $\mathcal{C}_\theta$ encode the history of all state observations and can calculate the probability distribution of the action for each state by:

$$Pr\left(\ell_t^p \middle| s_0, s_1, \dots s_t\right) = Pr\left(l_t^p \middle| h_t\right) = \begin{cases} l_t^p & if\ \ell_t^p = 1 \\ 1 - l_t^p & if\ \ell_t^p = 0 \end{cases} \tag{2.57}$$

$$Pr\left(p_t^q \middle| s_0, s_1, \dots s_t\right) = Pr\left(p_t^q \middle| h_t\right) = u_t^{p_t^q} \tag{2.58}$$

where $u_t^{p_t^q}$ is the value of a neuron in $u_t^{\mathcal{P}_q}$, $u_t^{p_t^q}$ represents the probability that associative recall will happen in the pathways $p \in \mathcal{P}_q$ to a layer q. Because the likelihoods of each gating output are independent of one another, the policy formula of NVM-RL can be written as:

$$\pi(a_t|s_0, s_1, \dots s_t) = \prod_{g \in a_t} Pr(g|h_t) \tag{2.59}$$

For each training set of data, NVM-RL is trained with E number of episodes. Each episode represents one sequence of input and output that the NVM-RL is supposed to execute. The objective is to maximize the rewards obtained during each episode. For each episode, the NVM-RL uses a reward-to-go method to estimate the state-action value at each time step:

$$R_{t,e} = \sum_{t=t'}^{T} r_{t,e} \tag{2.60}$$

The objective function approximation for a training epoch is defined as:

$$\mathbb{E}[R_0] \approx \frac{1}{E} \sum_{e}^{E} \sum_{t}^{T} log(\pi_\theta(a_{t,e}|s_{0,e}, \dots s_{t,e}))(R_{t,e} - b_t) \tag{2.61}$$

where E is the total number of episodes in the epoch; $R_{t,e}$ refers to the reward-to-go at time step t at training episode e; b_t is the average reward for all episodes at a time step t: $b_t = \frac{1}{E}\sum_e R_{t,e}$; T is the total time steps that NVM-RL has run for the episode (referred to as the *episode duration*), which is also equal to the length of the output sequence.

2.3.1 Experimental Results using the NVM-RL

All experiments in the original NVM-RL paper (Katz, et al., 2020) use an all-or-nothing reward function, which gives a reward 1 if the NVM-RL predict a correct result and gives a reward 0 otherwise. The reward is only given at the last time-step of an episode. For each experiment, the NVM-RL is trained for N epochs with E

episodes per epoch. For each episode, the NVM-RL is fed an input sequence with fixed length L and runs for T time-steps. The NVM-RL only reads in one input element each time-step. The character symbol $'0'$ serves as a special "delimiter" that can be padded by other meaningful symbol generated at the previous or later timestep, which means an output list $['1','0','2','3']$ is equivalent to the list $['1','2','0','3']$.

The experiments can be classified into two types: Experiments on problems that the NVM only needs 3 registers to accomplish, and experiments on problems that require the NVM to have a tape-based memory. Each experiment for a specific problem has 30 trials. In the following we consider three experiments (Max, filter and reverse) initially done using the NVM-RL.

Experiment *Max* addresses the problem of selecting the maximum value of a list. This experiment uses an input length $L = 5$ and episode duration $T = L$, which means the NVM-RL is expected to print out the correct answer right after reading all elements in the input list. The NVM-RL is expected to print the maximum value of the given list at the final time step (see Fig. 2.9). For example, the correct output of an input list $['5','2','0','4','3']$ can be any list with symbol $'5'$ at the final step.

Experiment *filter* trains the NVM-RL not to output some certain elements in the output list. In the experiments for Figure 2.9 (middle, bottom panels), the goal is to filter out the elements that are lower than $'4'$ in the input list. For example, a correct output list for input list $['9','2','0','5','3']$ can be a variety of lists with 9 and 5 in the correct order, such as $['9','0','0','5','0']$ or $['0','9','0','5','0']$, since the symbol $'0'$ is just padding.

The working memory of both *Max* and *filter* experiments is set up with 3 registers, $'rinp'$, $'rout'$ and $'rtemp'$. Each element of the input list is first written into $'rinp'$ and the output element for each time step is expected at $'rout'$. Register $'rtemp'$ serve as a register that can hold a symbol at each time step. In both experiments, the NVM-RL has the correct output in most episodes after a certain number of epochs, as shown in Figure 2.9.

In the experiment *reverse*, a correct output of NVM-RL would be a reversed list of the input list. This experiment use input length $L = 4$ and episode duration $T = 8$. That means the NVM-RL needs to print the correct output at the same length of the input list after reading all elements in the input list. This experiment requires a tape-based memory layer m instead of a register for temporary value $'rtemp'$ because the working memory needs to save more than one pattern at a time.

The NVM-RL did not consistently reach success (reward ≈ 1) with standard policy gradients in experiment *reverse*. As shown in Figure 2.10, despite the NVM-RL having a reward value that trends to one in some of the trials, the average reward over all 30 trials remains at about 0.6. The average reward of one epoch is trapped in a local minimum for many of the trials, which means the NVM-RL can only output some certain cases correctly but fails to learn the full mechanism to reverse a list.

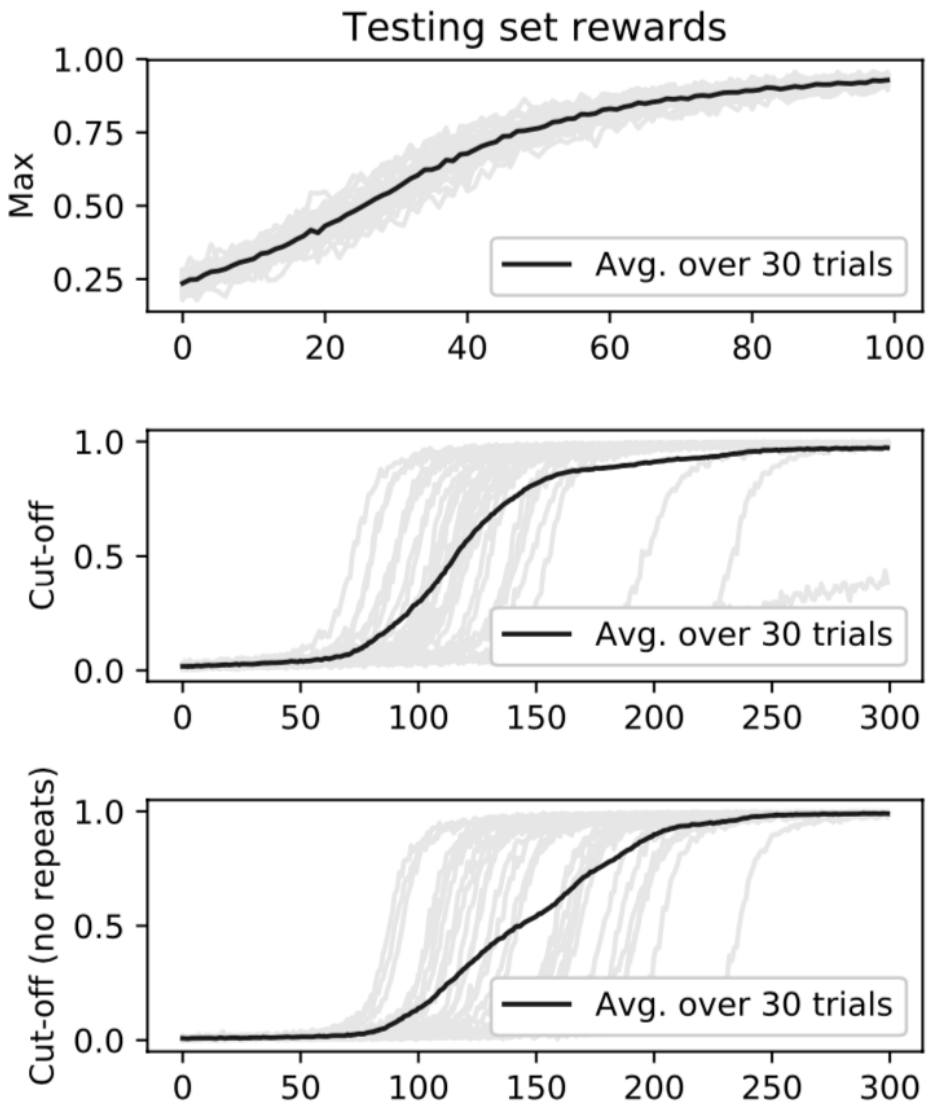


Figure 2.9 Experimental results for Max and filter. Grey lines show individual runs. The top panel is the result of experiment Max, the middle panel is the result of experiment filter without repeat elements in the list. The bottom panel is the result of experiment filter with repeat elements in the list. The horizontal axis is the number of epochs and the vertical axis is the reward received. (Katz, et al., 2020)

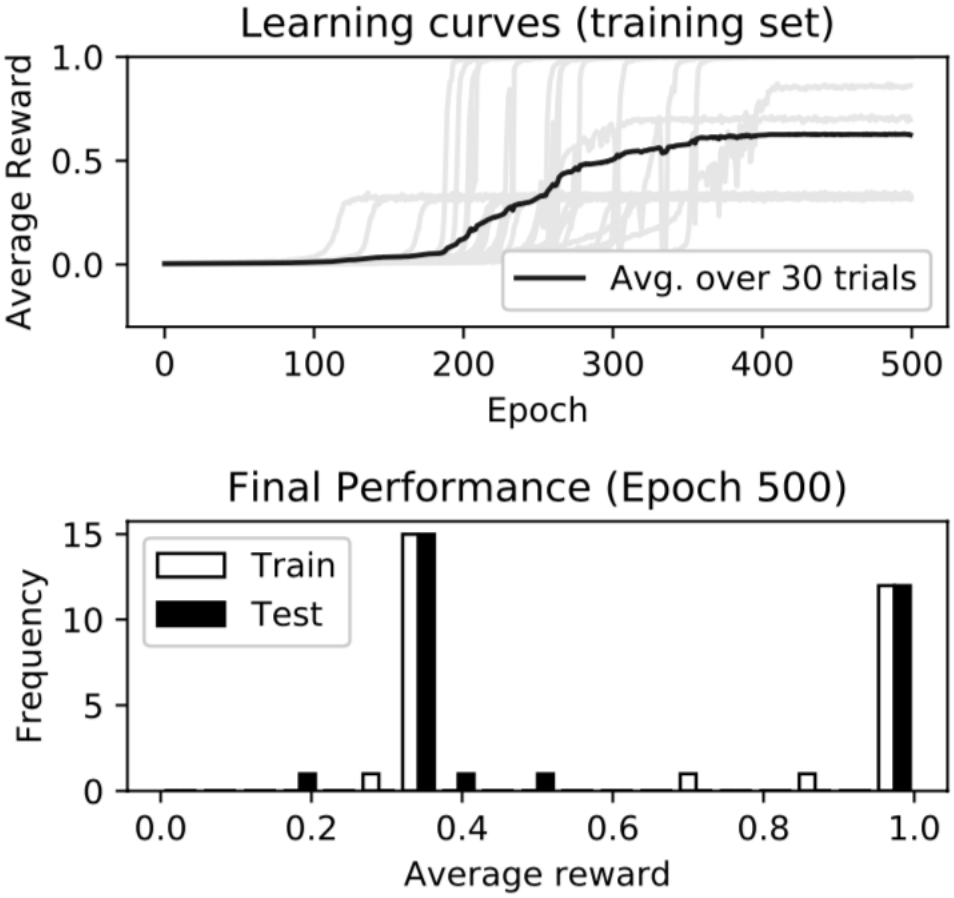


Figure 2.10 Result of experiment Reverse. The top panel shows the performance of NVM-RL through 500 epochs. The grey curve shows the average reward of each trial, the black curve shows the average result of all trials. The bottom figure shows the frequency of the final results after training. (Katz, et al., 2020)

3. Methodology

Experimental results have shown a large advantage of PPO over other reinforcement learning algorithms in many application problems (Schulman, et al., 2017). Therefore, I examined the hypothesis that training the NVM-RL with PPO might improve NVM-RL to have stable success with experiment *Reverse* or even enable NVM-RL to achieve success on more complex tasks. In this section, I introduce the methodology I used to apply PPO in NVM-RL and in the experiments comparing PPO and the SPG used in the original NVM-RL.

In PPO, we want to update the new policy π_θ with the data sampled from the old policy $\pi_{\theta_{\text{old}}}$. From equation 2.59, we calculate the stochastic policy as the likelihoods of gating outputs given the hidden states of the controller, since the hidden states encode the history of observation states from time-step 0 to time-step t.

$$\pi_{\theta_{\text{old}}}(a_t|s_0, \dots s_t) = \prod_{g \in a_t} Pr_{\theta_{old}}(g|h_t) \tag{2.59}$$

where $Pr_{\theta_{old}}(g|h_t)$ refers to the likelihoods of each gating decision from the controller with parameters θ_{old}. Following this equation, our new policy can be calculated by the using the same hidden states in the controller with new parameter θ_{new}:

$$\pi_{\theta_{new}}(a_t|s_0, \dots s_t) = \prod_{g \in a_t} Pr_{\theta_{new}}(g|h_t) \tag{3.1}$$

and the state-action value can still be estimated using rewards-to-go:

$$R_{t,e} = \sum_{t=t\prime}^{T} r_{t,e} \tag{2.60}$$

The ratio of 2 policies is just the same as 3.16:

$$r_t(\theta) = \frac{\pi_{\theta_{\text{new}}}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{3.2}$$

Further, we define our objective function by replacing the advantage estimator of equation 2.36 with the reward-to-go:

$$R_0{}^{CLIP}(\theta) \approx \frac{1}{E}\sum_{e}^{E}\sum_{t}^{T} \min\Big(r_t(\theta)\big(R_{t,e} - b_t\big), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\big(R_{t,e} - b_t\big)\Big) \tag{3.3}$$

The algorithm we use to train the NVM-RL is as follows:

Algorithm 3.1 Train NVM-RL with PPO

For epochs 1,2,3…do:

Run NVM-RL with a batch of training samples for $\boldsymbol{E}$ episodes,

Compute Reward-to-go, save hidden states $\boldsymbol{h_t}$ for all episodes.

For iteration 1,2,3…do:

Update $\boldsymbol{\theta_{new}}$ with objective function $\boldsymbol{R_0}^{\boldsymbol{CLIP}}(\boldsymbol{\theta})$

End for

End for

The NVM-RL is then modified with Algorithm **3.1**, denoted as NVM-PPO. The original NVM-RL trained with standard policy gradients is then denoted as NVM-SPG. Algorithm NVM-PPO is tested in experiments *Max*, *Filter*, *Reverse* and a more complex new task called *Sorting.*

The experiments *Max, Filter* and *Reverse* are set up with the same configuration of the NVM-SPG described in Section 2.3. Two experiments of a sorting task are performed, with input length $L = 3$, and input length $L = 4$ being tested. The episode duration is set as $E = 2L$ for both experiments. The sorting task can be considered as a more complex task than *reverse* because the NVM-RL is supposed to learn both the positional information and the numerical order for each pattern in the list. The NVM's working memory set up for experiment *Sorting* is the same as for experiment *Reverse.* The configuration of experiments on each task is shown in Table 1.

I tested the NVM-PPO on the experiments described above while using two different reward functions separately: the *all-or-nothing reward function* used in the original NVM-RL, and a *partial-credit reward function*. The partial-credit reward function is designed to give partial rewards to incorrect output lists of the NVM-RL. A correct output list will receive a reward 0 with the partial-credit reward function while every positional mistake in an output list will be added to form a negative reward *k*. In the experiments using the partial-credit reward function, I measured the performance of NVM-RL with the fraction of episodes that reach a reward 0, which is an equivalent measurement of the average reward in all-or-nothing function. The original NVM-SPG is also tested with the partial-credit reward function in order to compare its performance with NVM-PPO.

Table 1: CONFIGURATION FOR EACH EXPERIMENT

	L	T	E	$\mathcal{A}$
Max	5	5	500	10
Filter	5	5	1000	10
Reverse	4	8	5000	5
sorting	3-4	6-8	5000	5

4. Experimental Results

In this section, I describe the results of the experiments done in testing the performance of NVM-PPO versus the original NVM-SPG.

4.1 Results with All-or-Nothing Reward Function

The NVM-PPO with an all-or-nothing reward function was tested on experiments *Max, filter and Reverse.* Figure 4.1a shows the result of NVM-PPO in experiment *Max.* As with the results of NVM-SPG shown in Figure 2.9 top, NVM-PPO stably reach an average reward approximately equal to 1 in every individual trial of the experiments. Figure 4.1b shows the comparison between NVM-SPG and NVM-PPO. The original NVM-RL paper (Katz, et al., 2020) showed the results of experiment *Max* with a learning rate 0.1. I reproduced the experiment *Max* of NVM-SPG, changing the learning rate to find the best performance of NVM-SPG to see if the NVM-PPO outperforms NVM-SPG by approaching average reward equal to 1 with a smaller number of epochs. The results in Figure 4.1b show that the best performance of NVM-SPG can reach an average reward within a similar number of epochs as the NVM-PPO can do that. Therefore, NVM-PPO and NVM-SPG have essentially the same performance in experiment *Max,* with both of them can having consistent success.

Figure 4.2 shows that the NVM-PPO can also achieve consistent success in the *filter* task. Comparing with Figure 2.9 middle, we can conclude that both NVM-PPO and NVM-SPG can stably reach an average reward approximating 1. In other

words, for simple tasks *Max* and *filter* which only involve 3 registers, the NVM-PPO can achieve stable success just as NVM-SPG can do.

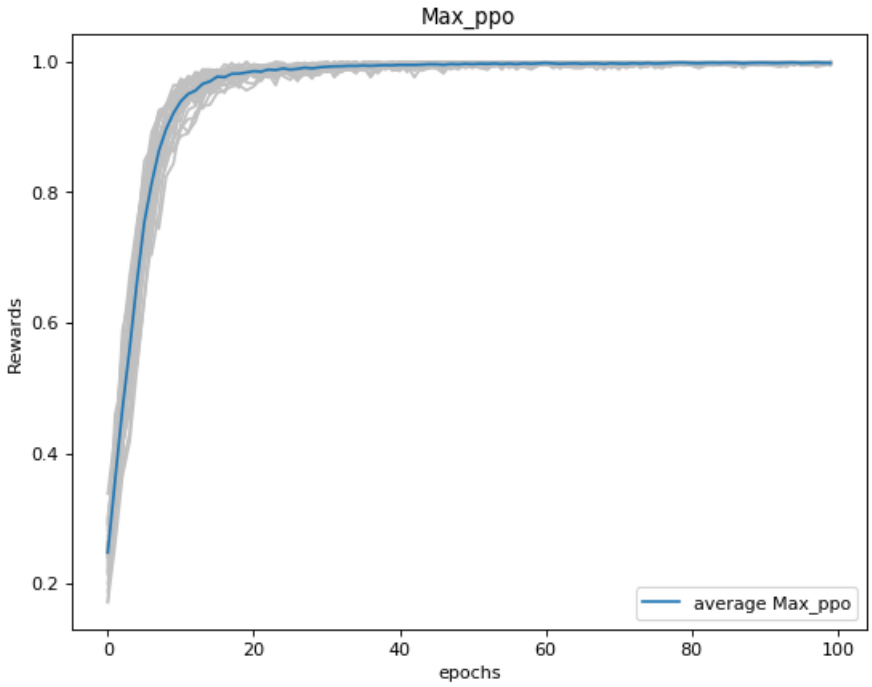


Figure 4.1a Result of Experiment Max in NVM-PPO. The grey curves show the result of each trial the blue line shows the average of all 30 trials. We see that every trial is able to reach an average reward equal to 1.

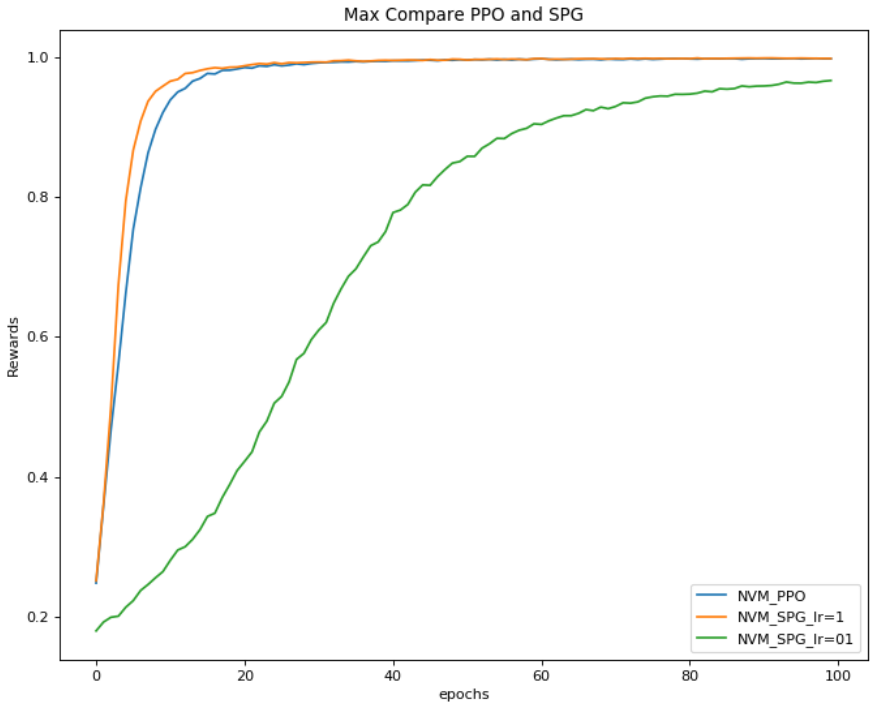


Figure 4.1b Comparing PPO and SPG in experiment Max. The blue line is the average reward over 30 trials with NVM-PPO. The green line is the average reward of NVM-SPG reproducing the result of the original NVM-RL paper, the blue line is the average reward of NVM using a different learning rate.

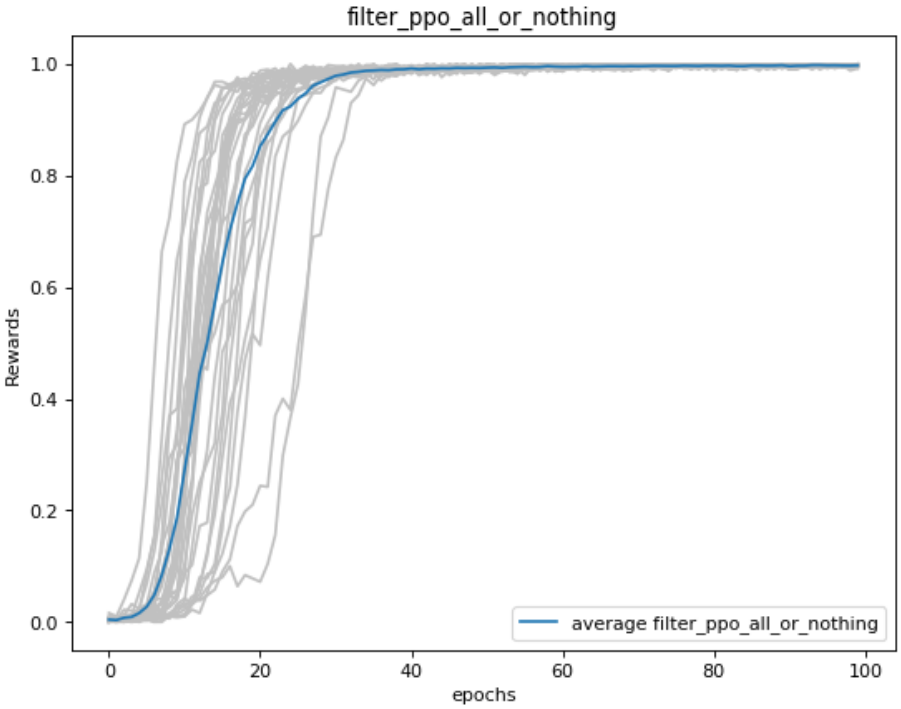


Figure 4.2 Experimental results for PPO with experiment Filter using all-or-nothing function. The grey curves represent the average reward of each individual trial. The blue curve represents the average reward of all 30 trials. Every trial is able to reach reward 1.

Despite the expectation that NVM-PPO would outperform NVM-SPG with a complex task like reverse, which requires the NVM to make use of a tape-head memory, the NVM-PPO did not show a better performance than that of NVM-SPG with the all-or-nothing reward function.

Figure 4.3 shows the performance of NVM-PPO on task *Reverse*. The average reward over 30 trials only reaches 0.6, which is at the same level as the result of NVM-SPG shown in Figure 2.10. Figure 4.3b shows that only 15 out of 30 trials have more than 90 percent of episodes producing a completely correct output after training. The results of 12 trials have average rewards trapped within 0.3 to 0.4. Compared with Figure 2.10, which shows that NVM-SPG achieved 12 out of 30 trials with an average reward bigger than 0.9, the NVM-PPO has only 3 trials more, which is not a sufficient lead to conclude that NVM-PPO has better performance. The NVM-PPO neither achieve a significant lead in average reward over all 30 trials nor having a significant number of more success cases than the NVM-SPG.

These results show that the NVM-PPO fails to exhibit consistent success in the task *Reverse,* and the NVM-PPO does not have a significant advantage over the NVM-SPG under the all-or-nothing reward function.

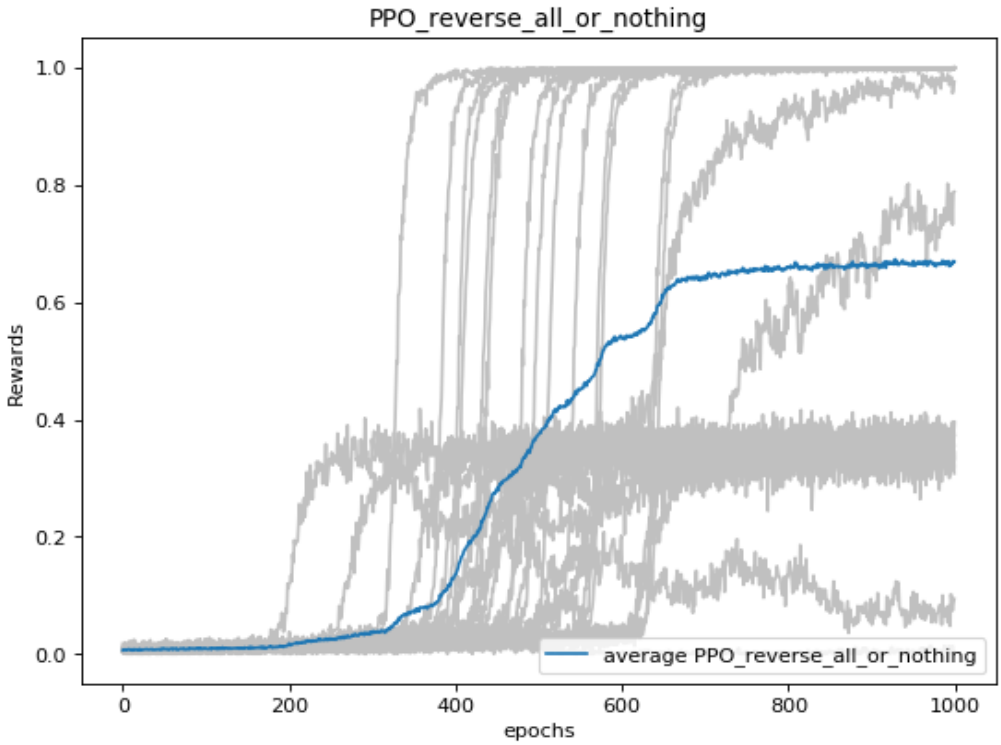


Figure 4.3a PPO experimental results for experiment Reverse using all-or-nothing function. The grey curves show the result of each individual trial, we can see many trials are trapped in an average reward < 0.4. The average reward of all 30 trials after training is about 0.6, shown by the blue line.

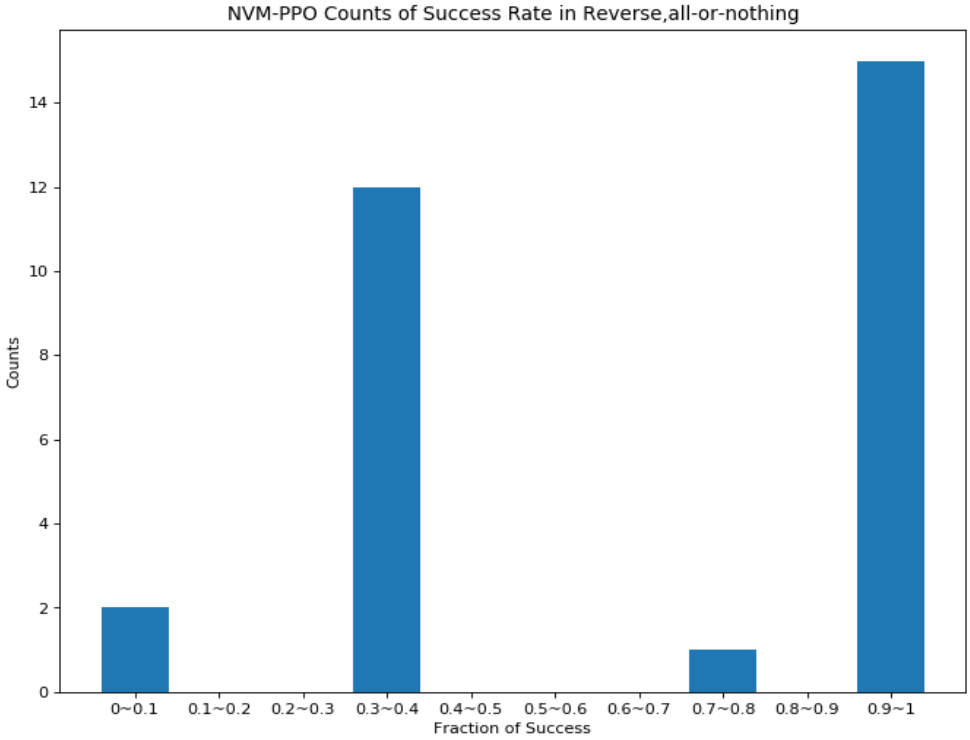


Figure 4.3b PPO experimental results with Reverse. This histogram shows the count of the range of average rewards for each trial. 15 out of 30 trials are able to reach an average reward > 0.9.

4.2 Results When Using a Partial-Credit Reward Function

The NVM-PPO and NVM-SPG with the partial-credit reward function was next tested on tasks *filter, reverse* and *sorting*. Figure 4.4 shows the fraction of correct episode outputs in an epoch of the NVM-PPO and NVM-SPG with the partial-credit function for the *filter* experiment. As when using an all-or-nothing reward function, both NVM-SPG and NVM-PPO are able to reach consistent successes in the experiment *filter.*

Figure 4.5 and Figure 4.6 show the experimental results of *Reverse* for NVM-PPO and NVM-SPG with the partial-credit reward function. None of the trials for NVM-SPG achieved a success fraction higher than 0.4 and the average success fraction over all 30 trials is around 0.2 as shown in Figure 4.5a. Figure 4.5b shows the average success fraction over all trials for NVM-PPO reaches a number around 0.9, which is a large improvement from NVM-SPG with the same partial-credit reward function. Figure 4.5c shows the large advantage of the NVM-PPO over NVM-SPG with the comparison of average success fraction over 30 trials. Figure 4.6b shows 24 out of 30 trials reached a success fraction over 90 percent while Figure 4.6a shows none of the 30 trials can have a success fraction over 40 percent.

That is to say, under a partial-credit reward function, NVM-PPO shows a dominant performance over NVM-SPG. In comparison with the original NVM-SPG with an all-or-nothing reward function in Figure 2.10, NVM-PPO with partial-credit reward function shows a better performance where 100% of trials approach a success fraction equal to 1 (from 12 out of 30 trials to 24 out of 30 trials).

Figure 4.7 shows the experimental results for the new task *Sorting (L=3)* for NVM-PPO and NVM-SPG. As shown in Figure 4.7a, the NVM-SPG only reaches an average success fraction of 0.4 after training while the maximum success fraction out of 30 trials is 0.8. Figure 4.7b shows the NVM-PPO consistently reaches success in task *sorting (L=3)* as the success fraction of every trial asymptotically trends to 1.0 at the end of training. Figure 4.8 shows the result of the NVM-PPO on the task *sorting (L=4),* which shows that the NVM-PPO becomes very unstable and only one trial reaches a success fraction around 1 during the training. In contrast, the NVM-SPG does not show any improvement at all on the task *sorting (L=4)* as the success fraction is always around zero. In conclusion of the sorting experiments with a partial-credit function, NVM-PPO shows a great advantage over NVM-SPG as NVM-PPO consistently performs with a higher success fraction in the experiment.

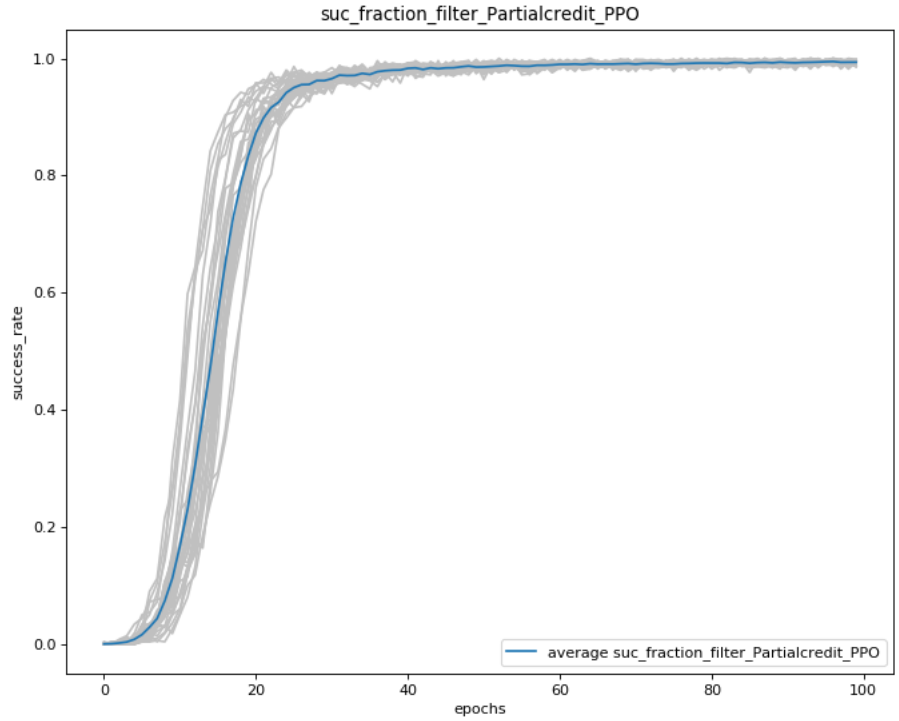


Figure 4.4a Result of experiment filter with NVM-PPO, under partial-credit reward function. The grey curves show every individual trial is able to reach a total success fraction equal to 1.

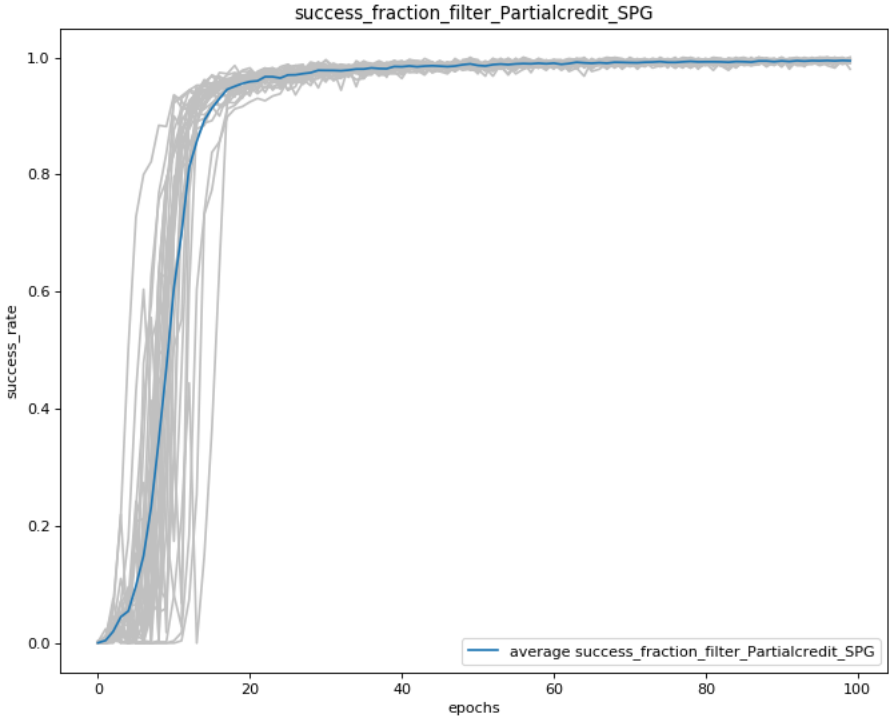


Figure 4.4b Result of experiment filter with NVM-SPG, under partial-credit reward function. Same as NVM-PPO every trial is able to reach success fraction equals 1.

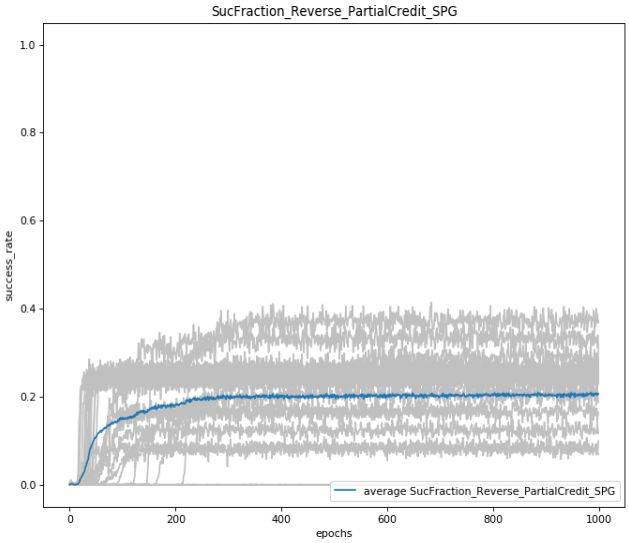


Figure 4.5a NVM-SPG Result of Experiment Reserve under partial-credit reward function. None of the 30 trials is able to reach a success fraction > 0.4.

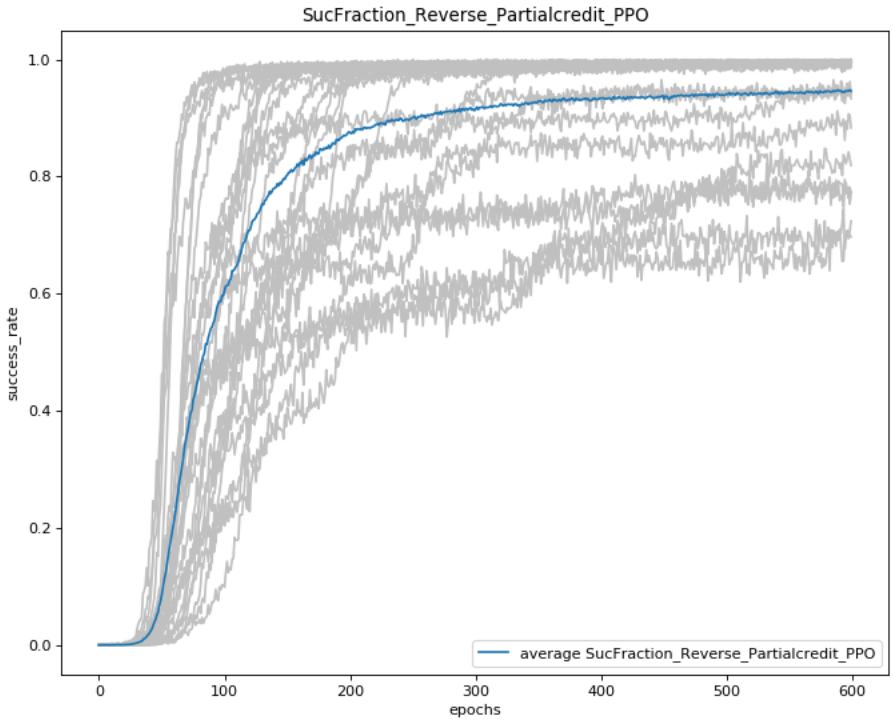


Figure 4.5b NVM-PPO Result of Experiment Reserve under partial-credit reward function. Many but not all of the 30 trials are able to reach the success fraction equals 1.

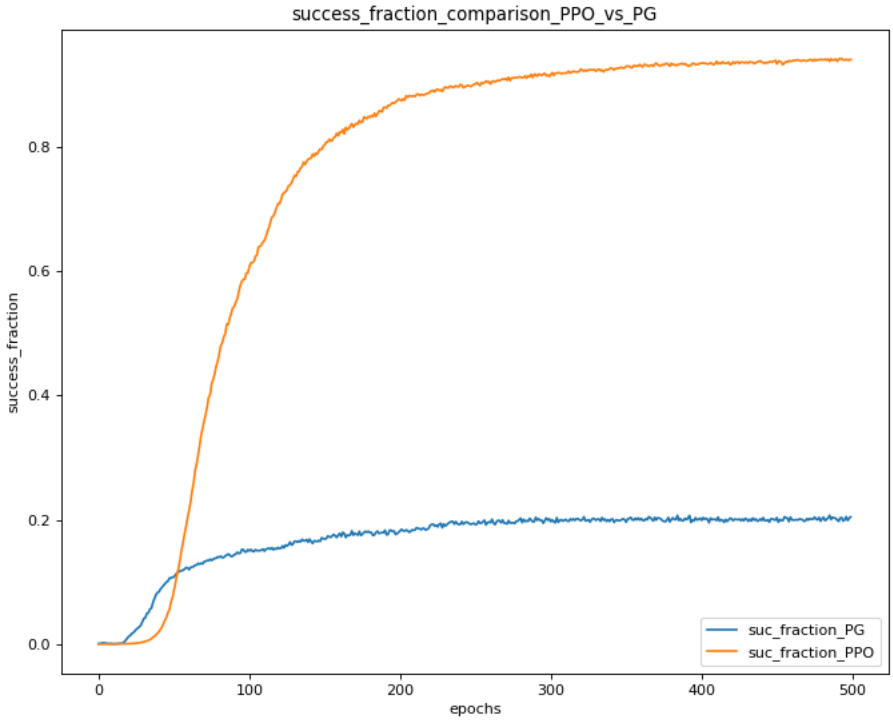


Figure 4.5c PPO vs SPG in Experiment Reverse under partial-credit reward function. The orange curve is the average success fraction of NVM-PPO and the blue curve is the average success fraction of NVM-SPG. NVM-PPO shows a great advantage over NVM-SPG.

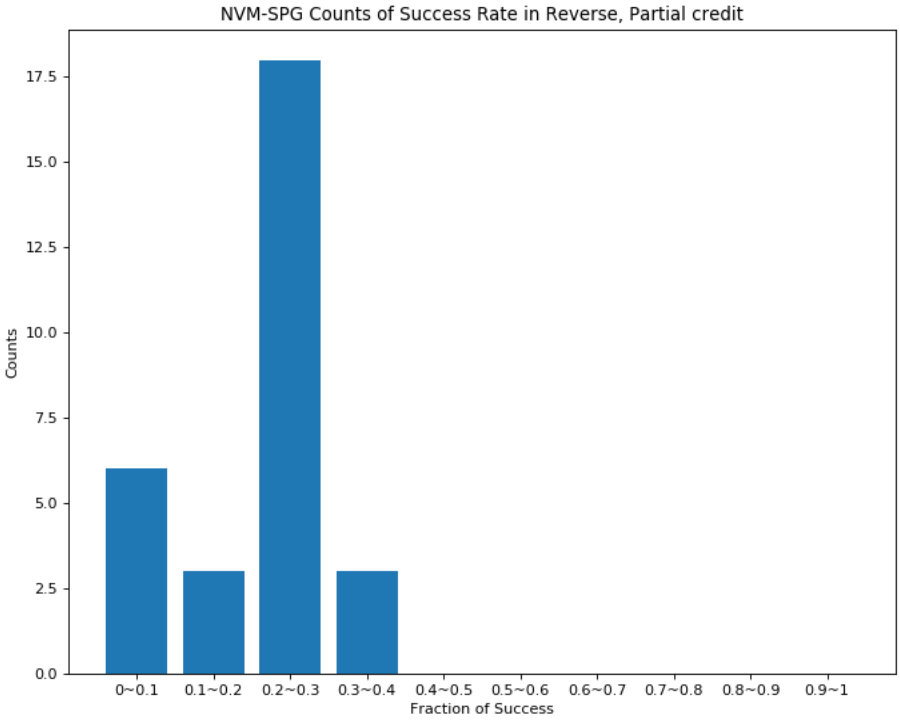


Figure 4.6a NVM-SPG Results for Experiment Reserve, under partial-credit reward function, where the histogram counts success fraction in 30 trials after training. The success fraction of all 30 trials lines within 0~0.4.

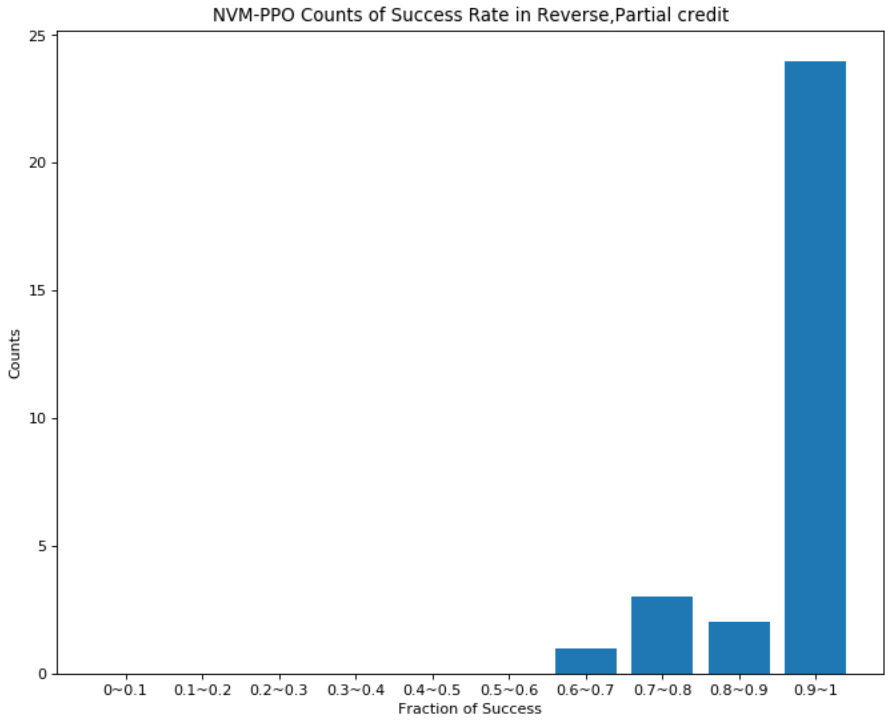


Figure 4.6b NVM-PPO Result of Experiment Reserve, under partial-credit reward function, where the histogram counts success fraction in 30 trials after training. 24 out of 30 trials are able to reach a success fraction > 0.9. All of the trials reach a success fraction > 0.6.

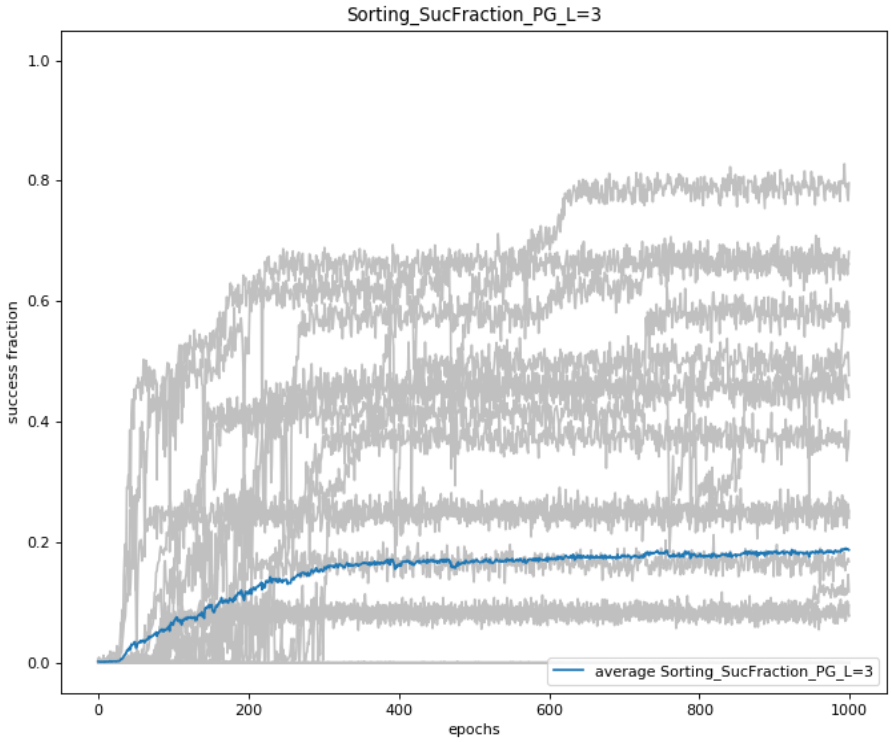


Figure 4.7a NVM-SPG Results of Experiment Sorting (L=3), under partial-credit reward function. The grey curves show the success fraction of every individual trials. None of the trials are able to reach a success fraction = 1. The blue represents the average success fraction of 30 trials, and this did not reach success fraction > 0.2.

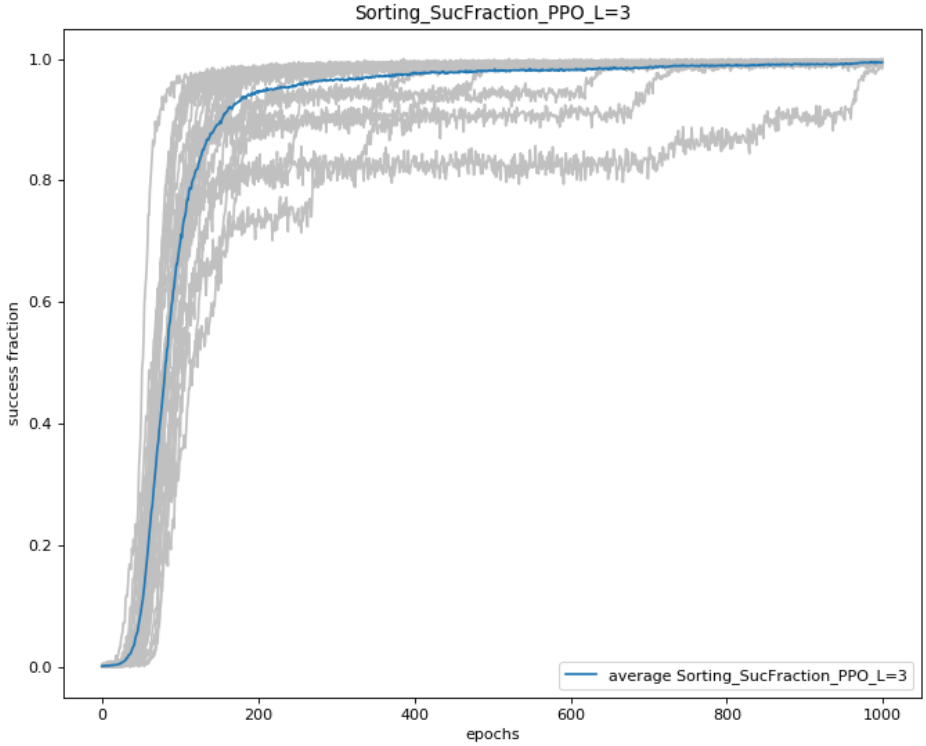


Figure 4.7b NVM-PPO Results of Experiment Sorting (L=3), under partial-credit reward function. All of the 30 trials are able to approach success fraction = 1.

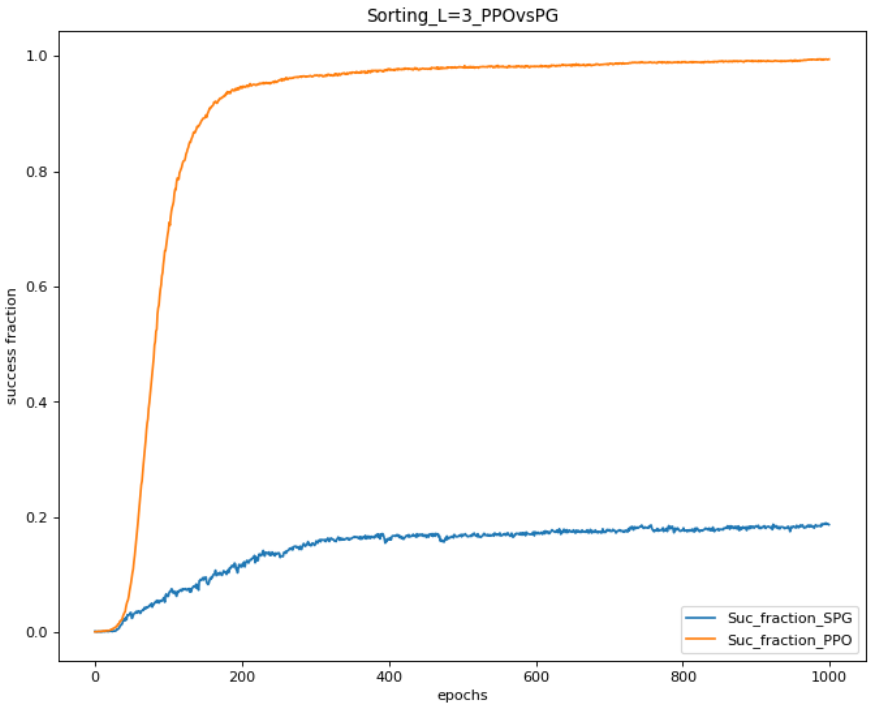


Figure 4.7c PPO vs SPG in Experiment Sorting (L=3), under partial-credit reward function. The orange curve is the average success fraction of NVM-PPO and the blue curve is the average success fraction of NVM-SPG. NVM-PPO shows a great advantage over NVM-SPG.

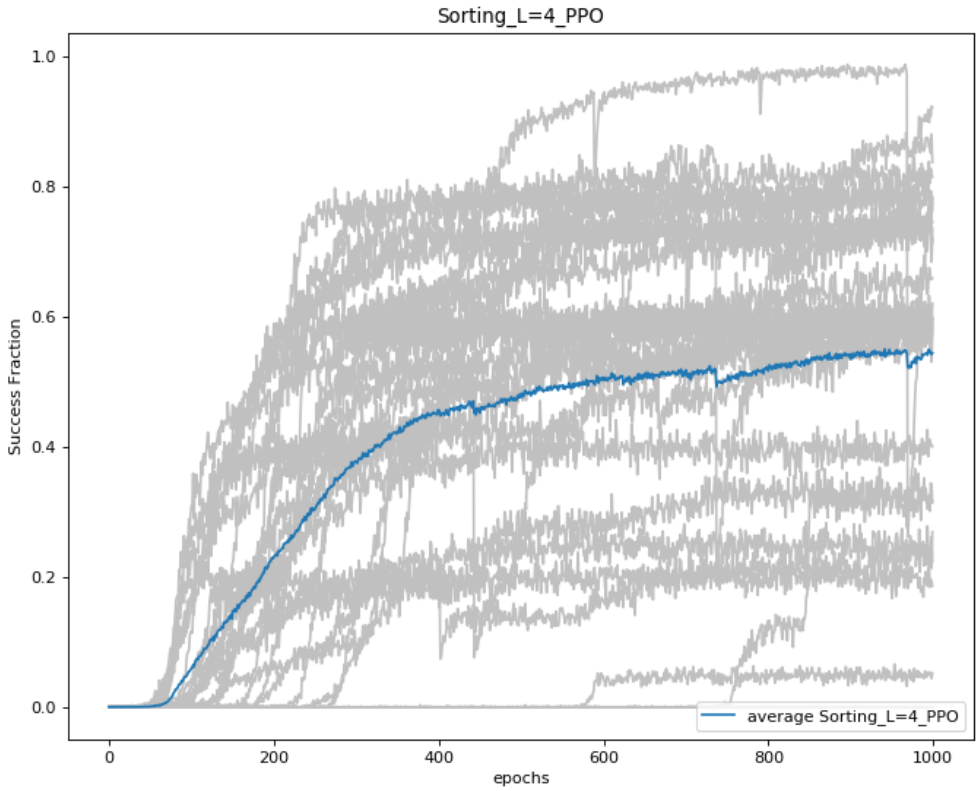


Figure 4.8 NVM-PPO Results with Experiment Sorting (L=4), under partial-credit reward function. The grey curves show the success fraction of every individual trials. None of the trials are able to reach a success fraction = 1. The result of each trial is very unstable. One of the trials is able to approach success fraction equals one during the training process but the performance drops down at some epoch. The average success fraction of 30 trials is around 0.5 after 1000 epochs or training.

5. Conclusion

This thesis aimed to improve the performance of NVM-RL by training it with a different modern reinforcement learning method. To assess this possibility, two versions of the NVM-RL were compared experimentally: NVM-SPG, which was the original NVM-RL, and a modified version NVM-PPO. As one of the most popular reinforcement learning algorithms, PPO was selected because of its simplicity for modifications and its data efficiency. The effects of these methodological variations on the performance of NVM-PPO and NVM-SPG were given in the previous section. These results suggest three conclusions.

First, surprisingly, PPO provided no advantage over SPG when compared to the original published results for NVM-SPG when using the all-or none reward function (Katz, et al., 2020). This clearly refuted my initial hypothesis. In the *Reverse* experiment with all-or-nothing reward, NVM-PPO showed neither a significantly higher average reward over all 30 trials nor had a significant lead in the number of trials that reach $reward > 0.9$ after training. Despite the dominating results of the PPO over other policy gradient methods reported elsewhere (Schulman, et al., 2017), the NVM-PPO did not show an improvement over the NVM-SPG under an all-or-nothing reward function.

Second, it was found that modifying the reward function to be a partial-credit function changed this situation dramatically. By using a partial-credit function, NVM-PPO demonstrated a clear and significant improvement over NVM-SPG. Specifically, NVM-PPO showed a large improvement, by providing more consistent

success in complex tasks such as *Reverse* and *Sorting* that require the use of tape-based memory in NVM-RL. For the more complex task *sorting (L=4),* the NVM-PPO also did reasonably well on a fraction of the episodes while NVM-SPG only rarely produced any correct outputs. In other words, with the change in reward function, the NVM-PPO was consistently successful with some complex problems that NVM-SPG did poorly on, and the NVM-PPO had inconsistent success on a more challenging task for which the NVM-SPG was totally untrainable. This achievement suggests that the further development of NPI with reinforcement learning methods using pure neural architectures might benefit from focusing on PPO rather than SPG.

Third and finally, this work supports not only the potential of NVM-RL to succeed as a new NPI approach, but also provides a basic test of the NVM itself on a specific and challenging application. The NVM is a new and novel neuro-computational architecture that emulates the mechanisms of a traditional computer with a purely neural network distributed representation, and it has undergone only limited testing so far. By building on the basic mechanisms of the NVM, NVM-PPO has convincingly demonstrated that the fundamental underlying concepts of the NVM are both effective and robust when used in practice.

The current main limitation of NVM-PPO that I observed is that its performance on a more complex problem of sorting lists decreases quickly as the number of symbols in the list increased. Although PPO improves the learning efficiency of NVM-RL, the improvement did not overcome the increase of action space and state space sizes when increasing problem size, suggesting that there is still substantial work to be done. For example, future work might be undertaken to

explore whether one could improve the performance of NVM-RL by modifying its internal structure to reduce the increased size of the action space as problem size increases. In addition, one might seek to develop a less task-specific form of NVM-RL. One possible approach to this would be embedding the representation of specific tasks into a vector that could be fed to the NVM controller. One might also explore training an advantage function and applying other reinforcement learning methods such as the actor-critic algorithm in NVM-RL.

Bibliography

Abdelbar, A.M., Andrews, E.A. and Wunsch II, D.C., 2003. Abductive reasoning with recurrent neural networks. *Neural Networks*, *16*(5-6), pp.665-673.

Aggarwal, C.C., 2018. *Neural networks and deep learning*. Springer.

Ansari, G.A., Saha, A., Kumar, V., Bhambhani, M., Sankaranarayanan, K. and Chakrabarti, S., 2019, August. Neural Program Induction for KBQA Without Gold Programs or Query Annotations. In *IJCAI* (pp. 4890-4896).

Dehaene, S. and Changeux, J.P., 1997. A hierarchical neuronal network for planning behavior. *Proceedings of the National Academy of Sciences*, *94*(24), pp.13293-13298.

Katz, G.E., Davis, G.P., Gentili, R.J. and Reggia, J.A., 2019. A programmable neural virtual machine based on a fast store-erase learning rule. *Neural Networks*, *119*, pp.10-30.

Katz, G.E., Gupta, K. and Reggia, J.A., 2020, July. Reinforcement-based Program Induction in a Neural Virtual Machine. In *2020 International Joint Conference on Neural Networks (IJCNN)* (pp. 1-8). IEEE.

Mehaffey, W.H., Doiron, B., Maler, L. and Turner, R.W., 2005. Deterministic multiplicative gain control with active dendrites. *Journal of Neuroscience*, *25*(43), pp.9968-9977.

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, June. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928-1937).

Peters, J. and Schaal, S., 2006, October. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2219-2225). IEEE.

Plate, T.A., 1995. Holographic reduced representations. *IEEE Transactions on Neural networks*, *6*(3), pp.623-641.

Salinias, E. and Sejnowski, T.J., 2001. Gain modulation in the central nervous system: Where behavior. *Neurophysiology, and Computation Meet, Neuroscientist*, *7*, pp.430-440.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P., 2015, June. Trust region policy optimization. In *International conference on machine learning* (pp. 1889-1897).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*. MIT press.

Sutton, R.S., McAllester, D.A., Singh, S.P. and Mansour, Y., 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems* (pp. 1057-1063).

Wei, Z., Xu, J., Lan, Y., Guo, J. and Cheng, X., 2017, August. Reinforcement learning to rank with Markov decision process. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 945-948).

Werbos, P.J., 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), pp.1550-1560.

Xu, D., Nair, S., Zhu, Y., Gao, J., Garg, A., Fei-Fei, L. and Savarese, S., 2018, May. Neural task programming: Learning to generalize across hierarchical tasks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 1-8). IEEE.

Zheng, Z., Wu, X. and Weng, J., 2019. Emergent neural turing machine and its visual navigation. *Neural Networks*, *110*, pp.116-130.

Rosenblatt, F., 1961. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms* (No. VG-1196-G-8). Cornell Aeronautical Lab Inc Buffalo NY.

Kelley, H.J., 1960. Gradient theory of optimal flight paths. *Ars Journal*, *30*(10), pp.947-954.

Ruder, S., 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Haykin, S., 2007. *Neural networks: a comprehensive foundation*. Prentice-Hall, Inc..