Reinforcement-based Program Induction in a Neural Virtual Machine

1st Garrett E. Katz
Elec. Engr. & Comp. Sci.
Syracuse University
Syracuse, NY, USA
gkatz01@syr.edu

2nd Khushboo Gupta
Biological Sciences Divison
Pacific Northwest National Laboratory
Richland, WA, USA
khushboo.gupta@pnnl.gov

3rd James A. Reggia
Computer Science
University of Maryland
College Park, MD, USA
reggia@cs.umd.edu

***Abstract*—We present a *neural virtual machine* that can be trained to perform algorithmic tasks. Rather than combining a neural controller with non-neural memory storage as has been done in the past, this architecture is purely neural and emulates tape-based memory via fast associative weights (one-step learning). Here we formally define the architecture, and then extend the system to learn programs using *recurrent policy gradient reinforcement learning* based on examples of program inputs labeled with corresponding output targets, which are compared against actual output to generate a sparse reward signal. We describe the policy gradient training procedure used, and report its empirical performance on a number of small-scale list processing tasks, such as finding the maximum list element, filtering out certain elements, and reversing the order of the elements. These results show that program induction via reinforcement learning is possible using sparse rewards and solely neural computations.**

***Index Terms*—Neural Networks, Policy Gradient, Program Induction, Fast Weights**

I. Introduction

Neural program induction (NPI) is the problem of training a neural network to perform an algorithmic task (like sorting a list), based on examples of program input (an unsorted list) and target output (the sorted list) [1]–[5]. Recent approaches to NPI often use a hybrid approach where neural controllers are coupled with non-neural memory. Humans also leverage non-neural memory (e.g., pen and paper) when solving algorithmic tasks, but the brain's working memory plays an equally important role. The most realistic AI models of working memory are those based on Hebb's rule [6], [7]. In the deep learning context these models are said to use "fast weights" in contrast with gradient descent, since Hebbian learning is very fast and in fact can be part of the *forward pass* rather than the gradient descent step [8], [9]. However, the use of fast weights specifically for NPI is underexplored. Moreover, NPI approaches (with fast weights or not) often use a supervised learning paradigm, which dictates the specific time-step at which each element of an output sequence must be produced.

In this work, we explore the alternative of a reinforcement learning (RL) paradigm [10] for NPI using solely neural computations (working memory but no external memory). From an engineering perspective, sparse reward signals (i.e., a single non-zero reward at the end of an episode), as opposed to timed supervision, may afford NPI with more flexibility in determining an effective algorithm. From a scientific perspective, our motivation is to advance NPI a small step towards greater biological relevance. For this purpose we modify a recently proposed fast-weight model called the "Neural Virtual Machine" (NVM) [11]. The NVM contains a set of *working memory* layers, which use a fast-weight associative learning rule that combines elements of Hebbian and anti-Hebbian learning. The NVM also includes a recurrent *controller* sub-network, that decides when fast learning and/or activation should occur in working memory layers, via multiplicative gating [12], [13]. Neural activity patterns in working memory layers are used to emulate various components of a typical computer architecture such as "register contents" and "memory addresses." The controller manipulates those patterns over time to emulate execution of a traditional computer program. In other words, the NVM is a purely neural model that can represent and emulate computer programs.

In [11], the NVM was "programmed by hand:" explicitly provided, human-authored source code was converted into a purely neural representation using a specialized algorithm. In this paper, we instead train a modified NVM model without explicit source code, using examples of program input and target output, and compare target and actual output to generate a reward signal. We redesign the NVM controller as an RL agent, and maximize the reward signal using recurrent policy gradient ascent [14]. We refer to this architecture as NVM-RL. The next section formally defines the NVM-RL architecture and training process, and subsequent sections report empirical performance on a number of algorithmic list processing tasks. The code is open-source and freely available online.[1]

II. The NVM-RL Architecture

The NVM-RL contains a set $\mathcal{L}$ of working memory layers and a set $\mathcal{P}$ of associative pathways (i.e., fast-weight matrices) between them. As detailed below, layers in $\mathcal{L}$ represent computer registers and tape-based memory addresses, while pathways in $\mathcal{P}$ are used to emulate operations like register

This work was supported by DARPA award HR00111890044 and ONR award N00014-19-1-2044. K. Gupta's contribution was work done while at Syracuse University.

[1] https://github.com/garrettkatz/ghu

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

moves and tape shifts. We use $\mathtt{p}$ to denote a pathway, and $\mathtt{q}$ and $\mathtt{r}$ to denote layers. The activity vector in layer $\mathtt{q}$ at time t, which can represent current register contents or tape positions, is denoted $\mathbf{v}_t^{\mathtt{q}}$, and the weight matrix of pathway $\mathtt{p}$ at time t is denoted $W_t^{\mathtt{p}}$. The NVM-RL also includes a simple recurrent "controller" network $\mathcal{C}_\theta$, with trainable parameters θ, which determines when associative learning and recall occur in any given pathway. $\mathcal{C}_\theta$ has a single hidden layer whose activity at time t is denoted $\mathbf{h}_t$. Its inputs are the previous hidden state $\mathbf{h}_{t-1}$ as well as the working memory vectors $\{\mathbf{v}_t^{\mathtt{q}}\}_{\mathtt{q}\in\mathcal{L}}$. Its outputs are scalar 0/1 multiplicative gates $\{\ell_t^{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ and $\{u_t^{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$. The gate value $\ell_t^{\mathtt{p}}$ modulates associative learning in pathway $\mathtt{p}$ at time t, while $u_t^{\mathtt{p}}$ modulates associative recall in pathway $\mathtt{p}$ at time t, as given in the equations below. The NVM-RL interacts with an environment $\mathcal{E}$ which has an unobservable state ψ_t at time t and can alter the patterns in the working memory layers. The NVM-RL's forward pass for one time-step consists of four sub-steps: (1) the controller observes current working memory and decides where to perform associative learning and recall; (2) associative recall occurs in the chosen pathways, (3) associative learning occurs in the chosen pathways, and (4) the environment provides external input by overwriting zero or more working memory layers with new activity vectors. The controller's "decision" to perform recall or learning in a given pathway is implemented by the corresponding multiplicative gates in its output layer. Mathematically, the sub-steps are:

$$\{u_t^{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}, \{\ell_t^{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}, \mathbf{h}_t = \mathcal{C}_\theta(\{\mathbf{v}_t^{\mathtt{q}}\}_{\mathtt{q}\in\mathcal{L}}, \mathbf{h}_{t-1}) \tag{1}$$

$$\hat{\mathbf{v}}_{t+1}^{\mathtt{q}} = \tanh\left(\sum_{\mathtt{p}\in\mathcal{P}_{\mathtt{q}}} W_t^{\mathtt{p}} \mathbf{v}_t^{\mathtt{r}_{\mathtt{p}}} u_t^{\mathtt{p}}\right) \tag{2}$$

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

$$\{\mathbf{v}_{t+1}^{\mathtt{q}}\}_{\mathtt{q}\in\mathcal{L}}, \psi_{t+1} = \mathcal{E}(\{\hat{\mathbf{v}}_{t+1}^{\mathtt{q}}\}_{\mathtt{q}\in\mathcal{L}}, \psi_t) \tag{4}$$

where $\mathcal{P}_{\mathtt{q}}$ is the set of incoming pathways to layer $\mathtt{q}$; $\mathtt{r}_{\mathtt{p}}$ is the source layer for pathway $\mathtt{p}$; $\hat{\mathbf{v}}_{t+1}^{\mathtt{q}}$ is the new neural activity before any modifications by the environment; and $\Delta W_t^{\mathtt{p}}$ is a fast weight update to pathway $\mathtt{p}$ at time t, described shortly.

The NVM is reframed for RL by viewing the controller as an agent, the controller's gating outputs as its actions, the working memory layers as its state observations, and $\mathcal{E}$ as the underlying environment's state transition function. To perform NPI with this system, certain working memory layers are designated for representing program inputs and outputs. The environment provides program inputs and consumes program outputs by accessing and modifying the designated layers.

For the purposes of this study we assume that program input/output symbols are elements of a finite "alphabet" $\mathcal{A}$, each represented by patterns of neural activity. A program manipulates these symbols by moving them between registers or reading/writing them from/to tape-based memory, similarly to a Turing machine [15]. Different registers, as well as the tape memory's read-write head, are represented by different working memory layers. The symbols currently stored in a register, as well as the current head position, are represented by the current activity patterns in the corresponding layers. The pattern in layer $\mathtt{q}$ representing symbol $\mathtt{a}\in\mathcal{A}$ is denoted $\mathbf{v}^{\mathtt{q}}[\mathtt{a}]$. This assigned pattern is fixed before training and may or may not equal the actual pattern $\mathbf{v}_t^{\mathtt{q}}$ occurring in layer $\mathtt{q}$ at any given time t. The patterns are randomly chosen from $\{-\rho,\rho\}^{N_{\mathtt{q}}}$, where $N_{\mathtt{q}}$ is the number of neurons in layer $\mathtt{q}$, and $\rho\in(0,1)$ is a fixed hyperparameter chosen by the user (values near 1 work best in practice; this study uses $\rho=0.99$). The set of all pattern assignments is $\Omega=\{\mathbf{v}^{\mathtt{q}}[\mathtt{a}]\}_{\mathtt{q}\in\mathcal{L},\mathtt{a}\in\mathcal{A}}$, and in principle could also be treated as part of the trainable parameters.

The fast weight update in the NVM-RL is a combination of Hebbian learning and unlearning. Formally, we define the update function $\mathcal{H}(\cdot,\cdot,\cdot)$ by

$$\mathcal{H}(W,\mathbf{x},\mathbf{y}) = \left(\tanh^{-1}(\mathbf{y}) - W\mathbf{x}\right)\mathbf{x}^\top/(N_{\mathbf{x}}\rho^2), \tag{5}$$

where W is the current weight matrix, $\mathbf{x}$ and $\mathbf{y}$ are activity patterns being associated, and $N_{\mathbf{x}}$ is the number of neurons in pattern $\mathbf{x}$. The weight update formula is then given by $\Delta W_t^{\mathtt{p}} = \mathcal{H}(W_t^{\mathtt{p}}, \mathbf{v}_t^{\mathtt{r}_{\mathtt{p}}}, \mathbf{v}_t^{\mathtt{q}_{\mathtt{p}}})$, where $\mathtt{q}_{\mathtt{p}}$ is the destination layer for pathway $\mathtt{p}$. The normalization term $N_{\mathbf{x}}\rho^2$ is chosen in this way so that associative recall from input patterns with $\pm\rho$ entries will tend to produce output patterns also with $\pm\rho$ entries. In one-time step, this rule adjusts the associative weights W so that the output pattern $\mathbf{y}$ can be recalled from the input pattern $\mathbf{x}$. Moreover, any previous association of $\mathbf{x}$ with some other pattern $\tilde{\mathbf{y}}$ will be erased, which is useful for emulating "over-writing" a tape position. [11] proved that $\mathcal{H}$ does indeed have this behavior - with certainty when the symbol encodings in Ω are pair-wise orthogonal, and in expectation when they are sampled uniformly from $\{-\rho,+\rho\}^{N_{\mathtt{x}}}$. In this work we use orthogonal encodings due to their superior reliability. We generate sets of orthogonal pattern vectors using the randomized Hadamard matrix technique from [11].

A. The controller network

We implement the controller $\mathcal{C}_\theta$ with a simple recurrent network, linear read-outs, and sigmoid and softmax activation functions to compute a stochastic policy. The binary learning/recall gates are then sampled from the policy. More formally, the controller network update is given by:

$$\mathbf{h}_t = \tanh\left(\sum_{q\in\mathcal{L}} W^{\mathtt{h},\mathtt{q}}\mathbf{v}_t^{\mathtt{q}} + W^{\mathtt{h},\mathtt{h}}\mathbf{h}_{t-1} + \mathbf{b}^{\mathtt{h}}\right) \tag{6}$$

$$\mathbf{l}_t = \sigma(W^{\mathtt{l},\mathtt{h}}\mathbf{h}_t + \mathbf{b}^{\mathtt{l}}) \tag{7}$$

$$\mathbf{u}_t^{\mathcal{P}_{\mathtt{q}}} = \mu(W^{\mathtt{q},\mathtt{h}}\mathbf{h}_t + \mathbf{b}^{\mathtt{q}}) \tag{8}$$

where σ denotes logistic sigmoid, applied element-wise, and μ denotes softmax. Optionally one can use relu in place of $\tanh$ for the hidden layer update. $\mathbf{l}_t$ is a pattern with one neuron per pathway in $\mathcal{P}$, whose value is the probability that fast learning will occur in that pathway at time t. $\mathbf{u}_t^{\mathcal{P}_{\mathtt{q}}}$ is a pattern with one neuron per pathway in $\mathcal{P}_{\mathtt{q}}$, whose values sum to one and which collectively represents a multinomial distribution over the incoming pathways to layer $\mathtt{q}$: i.e., for each destination layer $\mathtt{q}$, associative recall is only allowed in

one of its incoming pathways at a time. The weights and biases in (6)-(8) comprise the trainable parameters of $\mathcal{C}_\theta$:

$$\theta = \{W^{\mathtt{h},\mathtt{h}}, \mathbf{b}^{\mathtt{h}}, W^{\mathtt{l},\mathtt{h}}, \mathbf{b}^{\mathtt{l}}\} \cup \bigcup_{\mathtt{q}\in\mathcal{L}} \{W^{\mathtt{h},\mathtt{q}}, W^{\mathtt{q},\mathtt{h}}, \mathbf{b}^{\mathtt{q}}\}$$

Lastly, the gates are sampled from the output probabilities:

$$\ell_t^{\mathtt{p}} \sim \mathcal{B}(\mathbf{l}_t^{\mathtt{p}}) \tag{9}$$

$$p_t^{\mathtt{q}} \sim \mathcal{M}(\mathbf{u}_t^{\mathcal{P}_{\mathtt{q}}}) \tag{10}$$

$$u_t^{\mathtt{p}} = \begin{cases} 1 & \text{if } \exists \mathtt{q} : \mathtt{p} = p_t^{\mathtt{q}} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where $\mathbf{l}_t^{\mathtt{p}}$ is the particular neuron in $\mathbf{l}_t$ corresponding to pathway $\mathtt{p}$, $p_t^{\mathtt{q}}$ is the incoming pathway to layer $\mathtt{q}$ chosen for associative recall, and $\mathcal{B}$ and $\mathcal{M}$ denote Bernoulli and multinomial distributions, respectively. As described above, each layer $\mathtt{q}$ only allows associative recall in one incoming pathway at a time, so the set of multiplicative gates $\{u_t^{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}_{\mathtt{q}}}$ will be "one-hot" for each $\mathtt{q}$, as formalized in (11).

B. Training Episodes

The NVM-RL can manipulate its working memory layers over the course of a training episode to emulate execution of a computer program. We assume program inputs and outputs take the form of sequences, where each sequence element is a symbol in the alphabet $\mathcal{A}$. Without loss of generality we can denote the symbols `0`, `1`,..., `K-1`, with $\mathtt{K} = |\mathcal{A}|$. We designate `0` as a special "delimiter" or "padding" symbol that can surround meaningful symbols in the input and output.

As a concrete example, consider the task of swapping two elements in a list. Given an input sequence `1,2`, the NVM-RL should produce a target output sequence `2,1`. However, we allow that one or more `0` symbols can be interspersed throughout the sequences while the NVM-RL is performing its computations. For example, the input sequence for the first five time-steps could be $[1, 2, 0, 0, 0]$. This input is represented by the corresponding patterns in a designated input layer, `in`:

$$\mathbf{v}_0^{\mathtt{in}} = \mathbf{v}^{\mathtt{in}}[1], \mathbf{v}_1^{\mathtt{in}} = \mathbf{v}^{\mathtt{in}}[2], \mathbf{v}_2^{\mathtt{in}} = \mathbf{v}^{\mathtt{in}}[0], ...$$

We use the environment $\mathcal{E}$ to stream in these values for $\mathbf{v}_t^{\mathtt{in}}$ in (4), overwriting any intermediate activity $\hat{\mathbf{v}}_t^{\mathtt{in}}$ produced internally by (2). Two possible correct output sequences are $[0, 0, 0, 2, 1]$ and $[0, 0, 2, 0, 1]$, while an incorrect example output is $[0, 2, 0, 3, 1]$. The NVM-RL is responsible for producing a correct output sequence in another designated layer `out`. This layer gets set by the NVM-RL dynamics and is left unchanged by the environment, i.e., $\mathbf{v}_t^{\mathtt{out}} = \hat{\mathbf{v}}_t^{\mathtt{out}}$. The actual output can then be compared with the target output by first removing all occurrences of the delimiter symbol `0`, and then measuring sequence similarity. In this work we use a simple, sparse "all or nothing" reward. The rewards at each time-step, denoted r_t, are all zero except possibly for the last time-step. If the actual and target outputs match perfectly, then the reward at the final time-step is 1; otherwise it is also 0.

A typical solution to the swap problem above would use an additional register. Similarly, the NVM-RL can solve this problem if it has another layer, say `tmp`, to serve as a general-purpose register for temporary information storage. If an association between $\mathbf{v}^{\mathtt{q}}[\mathtt{a}]$ and $\mathbf{v}^{\mathtt{r}}[\mathtt{a}]$ has already been learned for each symbol `a` and each pair of layers `q` and `r`, then a "register move" operation can be emulated by nothing more than associative recall in the corresponding pathway. To that end, we initialize these associations in each NVM-RL pathway prior to training as shown in Algorithm 1. The set of pathways can include auto-associative recurrent pathways from each layer to itself, which are used to maintain the current symbol in a layer as long as it is needed there. In other words, $\mathtt{r}_{\mathtt{p}} = \mathtt{q}_{\mathtt{p}}$ is possible for some $\mathtt{p}$.

Algorithm 1 Associations for register moves

```
for p ∈ P do
    Initialize W_0^p with all zero entries
    for a ∈ A do
        W_0^p ← W_0^p + H(W_0^p, v^{r_p}[a], v^{q_p}[a])
    end for
end for
```

A generalization of swap beyond length 2 is reversing the elements of a list, but this calls for a different approach. For longer lists, it is preferable to emulate tape memory, in which a read-write head can be incremented and decremented, and contents can be written to or read from the head location. To this end, we replace `tmp` with a new working memory layer `m` that represents the current position of the read-write head. In addition to the same auto-associative pathway as before, there are two additional pathways from `m` to itself, named `inc` and `dec`, each with a distinct weight matrix, used to increment or decrement the head position, respectively. Prior to training, these pathways learn associations between consecutive head positions as appropriate, as shown in Algorithm 2 (since $\mathcal{A}$ is finite, we use arithmetic modulo $|\mathcal{A}|$ for $\mathtt{k} \pm 1$). As a result, head increments and decrements are nothing more than associative recall in the corresponding pathways. With this setup, if the current symbol in layer `q` is `c` (i.e., $\mathbf{v}_t^{\mathtt{q}} = \mathbf{v}^{\mathtt{q}}[\mathtt{c}]$), and the current head location is `k` (i.e., $\mathbf{v}_t^{\mathtt{m}} = \mathbf{v}^{\mathtt{m}}[\mathtt{k}]$), then `c` can be "written" to tape address `k` via associative learning in the pathway `p` from `m` to `q`:

$$W_{t+1}^{\mathtt{p}} = W_t^{\mathtt{p}} + \mathcal{H}(W_t^{\mathtt{p}}, \mathbf{v}_t^{\mathtt{q}}, \mathbf{v}_t^{\mathtt{m}}) \tag{12}$$

Note that this need not happen prior to training: the controller can "decide" to perform this operation *during an episode* via (3), by emitting the appropriate gate value. Later in the episode, if the head location is restored to position `k`, the symbol `c` stored there can be retrieved by associative recall in pathway `p`. Due to the anti-Hebbian term in $\mathcal{H}$, this process can be performed multiple times per episode (a tape location can be overwritten with new contents multiple times if necessary).

C. Policy Gradient Ascent

The NVM-RL can be trained to maximize rewards using recurrent policy gradients [14]. The state observation at time

Algorithm 2 Associations for tape-head movements

Initialize $W_0^{\mathtt{p}}$ with all zero entries, for $\mathtt{p} \in \{\mathtt{inc}, \mathtt{dec}\}$
for $\mathtt{k} \in \mathcal{A}$ **do**
  $W_0^{\mathtt{inc}} \leftarrow W_0^{\mathtt{inc}} + \mathcal{H}(W_0^{\mathtt{inc}}, \mathbf{v}^{\mathtt{m}}[\mathtt{k}], \mathbf{v}^{\mathtt{m}}[\mathtt{k}+1])$
  $W_0^{\mathtt{dec}} \leftarrow W_0^{\mathtt{dec}} + \mathcal{H}(W_0^{\mathtt{dec}}, \mathbf{v}^{\mathtt{m}}[\mathtt{k}], \mathbf{v}^{\mathtt{m}}[\mathtt{k}-1])$
end for

t is the set of working memory activities:

$$s_t = \{\mathbf{v}_t^{\mathtt{q}}\}_{\mathtt{q} \in \mathcal{L}} \tag{13}$$

The action at time t is the collection of gating decisions:

$$a_t = \{p_t^{\mathtt{q}}\}_{\mathtt{q} \in \mathcal{L}} \cup \{\ell_t^{\mathtt{p}}\}_{\mathtt{p} \in \mathcal{P}} \tag{14}$$

The probability of action a_t given all observations so far is the stochastic policy $\pi(a_t \,|\, s_0, ..., s_t)$, which depends on the probability distributions output by $\mathcal{C}_\theta$. The hidden state $\mathbf{h}_t$ encodes the history of state observations $s_0, ..., s_t$. Likelihoods of individual gate values are computed from the output of $\mathcal{C}_\theta$:

$$\Pr(\ell_t^{\mathtt{p}} \,|\, \mathbf{h}_t) = \begin{cases} \mathbf{l}_t^{\mathtt{p}} & \text{if } \ell_t^{\mathtt{p}} = 1 \\ 1 - \mathbf{l}_t^{\mathtt{p}} & \text{if } \ell_t^{\mathtt{p}} = 0 \end{cases}$$
$$\Pr(p_t^{\mathtt{q}} \,|\, \mathbf{h}_t) = \mathbf{u}_t^{p_t^{\mathtt{q}}}$$

Here $\mathbf{u}_t^{p_t^{\mathtt{q}}}$ denotes the neuron in softmax layer $\mathbf{u}_t^{\mathcal{P}_{\mathtt{q}}}$ corresponding to the selected recall pathway $p_t^{\mathtt{q}} \in \mathcal{P}_{\mathtt{q}}$. We generate each gate value independently of the others, conditional on $\mathbf{h}_t$, so the stochastic policy has the formula

$$\pi(a_t \,|\, s_0, ..., s_t) = \prod_{g \in a_t} \Pr(g \,|\, \mathbf{h}_t), \tag{15}$$

where a_t is viewed as the set in (14). The objective is to maximize the expected net reward over all possible episodes:

$$\max_\theta \mathbb{E}\left[\sum_t r_t\right] \tag{16}$$

which is optimized via gradient ascent. The gradient is estimated from a sample of episodes. To reduce the variance of the estimate, we use baselines and rewards-to-go [16]. The reward-to-go at time t in episode e is $R_{t,e} = \sum_{t'=t}^{T} r_{t',e}$, where T is the duration of the episode and $r_{t,e}$ is the value of r_t in episode e. We use the average over all episodes as a time-dependent baseline: $b_t = \sum_e R_{t,e}/E$, where E is the total number of episodes. The resulting policy gradient estimate is:

$$\nabla_\theta \mathbb{E}[R_0] \approx \frac{1}{E} \sum_{t,e} \nabla_\theta \log \pi(a_{t,e} \,|\, s_{0,e}, ..., s_{t,e})(R_{t,e} - b_t) \tag{17}$$

where $a_{t,e}$ and $s_{t,e}$ are the actions and states at time t in episode e, respectively. Note that R_0 is the reward-to-go from time 0, i.e., the net reward for an episode.[2]

[2]In this study, since we use a sparse reward where r_t is zero except possibly at the last time-step, the rewards-to-go $R_{t,e}$ do not actually vary over time. We use the more general formulation to cover both sparse and dense rewards.

For more challenging tasks, we found it useful to augment the objective with an additional heuristic term to guide the policy search. Specifically, we note that in many tasks, such as swap, the proper gating sequence should not depend on the actual list contents. In other words, the probability distributions output by $\mathcal{C}_\theta$ should be identical across episodes. We can formalize this by minimizing the empirical variance in those distributions:

$$D = \frac{1}{E}\left(\sum_{t,e} ||\mathbf{l}_{t,e} - \bar{\mathbf{l}}_t||^2 + \sum_{t,e,\mathtt{q}} ||\mathbf{u}_{t,e}^{\mathcal{P}_{\mathtt{q}}} - \overline{\mathbf{u}}_t^{\mathcal{P}_{\mathtt{q}}}||^2\right) \tag{18}$$

where overbars denote averages over all episodes, and again e indexes the episode. Minimizing this term is equivalent to maximizing its negative in the objective function:

$$\max_\theta f(\theta) \tag{19}$$
$$f(\theta) = \mathbb{E}[R_0] - \lambda D \tag{20}$$

where λ is a hyper-parameter balancing expected reward vs. the regularization term. We optimize f starting from random initial parameters $\theta^{(0)}$ and using basic gradient ascent:

$$\theta^{(n+1)} \leftarrow \theta^{(n)} + \eta \nabla_\theta f(\theta^{(n)}) \tag{21}$$

where learning rate η is another hyper-parameter. We call each iteration of (21), which involves E episodes, an "epoch."

III. Computational Experiments

We tested NVM-RL learning on multiple sequence processing tasks of increasing difficulty. In each task, we used three registers, as described above: `in`, `out`, and `tmp` (except `tmp` is replaced with a read-write head layer `m` in the final task). All tasks use an alphabet of size $|\mathcal{A}| = 10$ (except in the final task), a learning rate of $\eta = 0.1$, and sparse "all-or-nothing" reward as described above. We found that this configuration consistently worked well for most tasks we attempted.

Input sequence length L varied as appropriate to the task and ranged from 3 to 6. We also varied E, the number of episodes used to compute each gradient update, and N, the number of neurons in each layer. Heuristically we chose larger E and N for tasks that we perceived to be more challenging. In some tasks we tried non-zero values for λ. We also made task-specific restrictions on the set of pathways where associative learning was allowed, based on prior knowledge of the task.

For training/testing data, we first enumerated the set of all possible input sequences, and then randomly split that set into 80% for training and 20% for testing. In each task we performed 30 independent trials of policy search, each time using a different random train/test split and random initial controller parameters $\theta^{(0)}$. In each epoch of each trial, E input sequences were selected uniformly at random with replacement from the 80% training set and fed through the NVM-RL. The resulting NVM-RL actions and rewards were used to compute the policy gradient update. After each update, we also fed E random samples from the 20% test set through the NVM-RL to track its performance on the test data (i.e.,

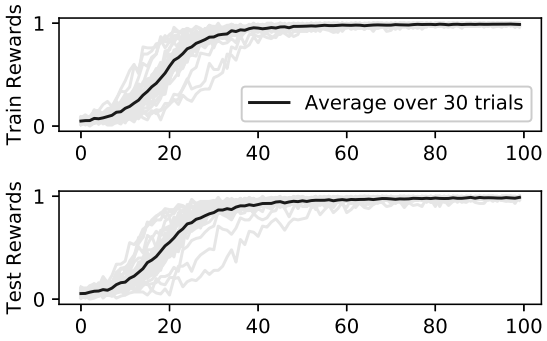

Fig. 1. Rewards at each training epoch for the swap task, averaged over E episodes. Each light grey curve is an independent trial, and the average over all trials is shown in black. *Top:* Rewards on the training data, which were used to compute the policy gradient updates. *Bottom:* Rewards on the test data (measured at every epoch, but not used in the gradient computation).

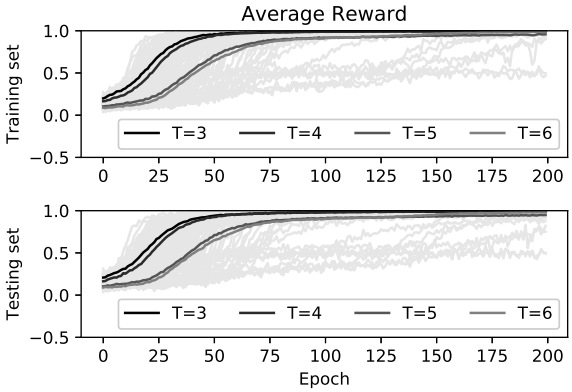

Fig. 2. Learning curves for the echo task. Each light grey curve is a separate trial. Darker grey curves are the average over 30 trials at each episode duration (T), progressing from upper left (T=3) to the right and down (T=6). The top and bottom panels show rewards on training and testing data, respectively. The latter were measured every epoch but excluded from gradient computation.

generalization ability) during the learning process. But no test data was used in the computation of the policy gradient.

We describe each task and its empirical performance in the following sub-sections. Table I summarizes the hyper-parameters used in each task, as well as approximate computing time and space requirements. All experiments were performed on an 8-core Intel i7 CPU with 32GB of RAM. Automatic gradients were calculated with PyTorch [17] but we did not make use of any GPUs. Using GPUs in future work may improve the computation times in Table I.

TABLE I
EXPERIMENT CONFIGURATIONS

Task	L	E	N	λ	Time*	Space*
Swap	3	100	32	0	4s	0.2GB
Echo	1-3	100	32	0	30s	0.2GB
Max	5	500	32	0	30s	0.2GB
Filter (w/ repeats)	5	1000	32	0	1.5m	0.2GB
Filter (w/o repeats)	5	1000	32	0	1m	0.2GB
Key-value lookup	6	5000	64	0.5	27m	2.7GB
Reverse	4	5000	64	0.5	19m	1.7GB

*Approximate time and memory used per trial, averaged over 30 trials.

A. Swap

As a starting point, we tested the NVM-RL on a small swap task, similar to the example above. After the fast-weight initialization prior to training, we disabled associative learning during the episodes, so that the only actions available dealt with associative recall. For this initial task we limited the episode duration to 3 time-steps. Fig. 1 shows the learning curves from 30 independent trials. In all cases, learning converged to essentially perfect success rate on both training and testing data. Test performance is slightly lower than training performance on average, but the difference is negligible.

B. Delayed Echo

Next, we trained the NVM-RL to echo an input symbol after a delay. As with swap, we disabled associative learning in this task after the initial symbol associations were formed. The input sequence was all `0` except for one random time-step in the first half of the episode (the "input window") when a non-delimiter symbol was inserted. The target output was also all `0`, except for echoing the same non-delimiter symbol exactly once in the second half of the episode (the "output window"). The episode duration T is the total number of time-steps for the episode including both input and output windows. We used input windows ranging from 1 to 3 time-steps and corresponding episode durations from 3 to 6 time-steps. Fig. 2 shows the learning curves for all trials and episode durations. On average, convergence to the global optimum is very rapid for shorter episode durations. For longer episodes, convergence was slower, but in all cases the task was solved or nearly solved within 200 epochs. Fig. 3 shows the distribution of rewards on the test data, grouped by episode duration. Early in the training process (epoch 50), generalization performance is clearly better on the shorter duration variants, but by the end (epoch 200), all averages are near 1.0.

C. Filtering

Both swap and echo can be solved with simple algorithms that ignore the input symbols and are only concerned with timing. Here we turn to list filtering tasks, which require the NVM-RL to exhibit branching behavior depending on which input symbols are encountered. For these tasks we used a fixed input length of 5. The first task was to extract the maximum element from a list, where each element was drawn from $\mathcal{A}$ with replacement. A concept of numeric ordering is not explicitly built in to the NVM-RL (each symbol is ultimately represented by a random activity pattern), but we hypothesized that the controller could learn to classify pairwise inequalities drawn from a finite set as "true" or "false" and branch accordingly. For this task an all-or-nothing reward of 1 was given at the last time-step of the episode when the output symbol was indeed the maximum of the input sequence.

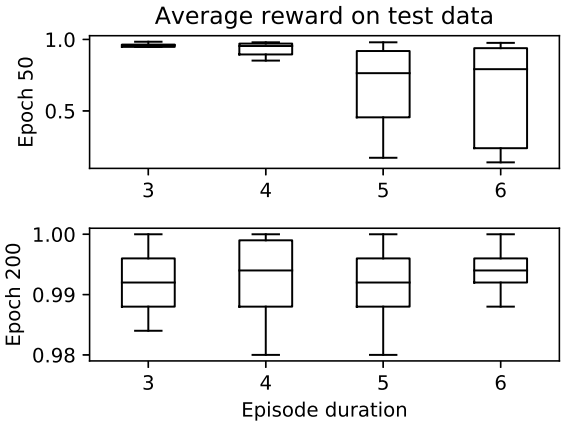

Fig. 3. Box plots of average reward on the echo task testing data, early (top) and late (bottom) in the training process. Note the different y-axis scales.

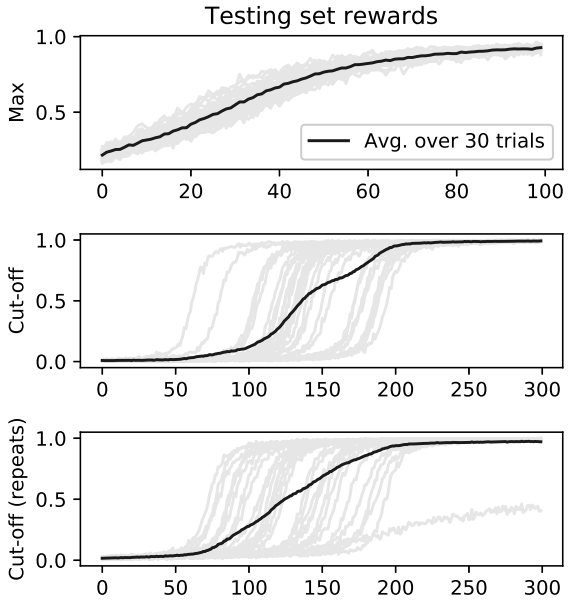

Fig. 4. Generalization on the list filtering tasks. *Top:* Producing the maximum list element. *Middle:* Producing all elements greater than `4` (no repeated input elements). *Bottom*: Like the previous, but with repeats in the input sequence. Note the different x-axis scale in the top panel for the max task.

The second variant of this task was to filter out all list elements greater than a fixed cut-off of `4`. For this second variant we tried list elements drawn both with and without replacement. For reward calculations we used output symbols produced during the entire episode (including before the input sequence was finished). An all-or-nothing reward of 1 was given when the output sequence contained only the elements greater than `4`, in the same order and multiplicity as they appeared in the input. During this output sequence, the delimiter `0` was also allowed as "padding" in between filtered elements without incurring a reward penalty.

Figure 4 shows the test set rewards for each variant of the filter task. In most cases the NVM-RL learns to generalize with near-perfect success rate, although more training epochs are required for the cut-off variants than the max variant, and there are some outlier trials that do not learn to perform effectively within the 300 epoch training window. We also confirm that the learning curves become smoother when more episodes are used at each epoch, as one might expect since the gradient estimate becomes more accurate.

D. Key-Value Lookup

All tasks described so far are "online" list processing algorithms that can be solved solely with associative recall and the fast weight initializations *before* the episode begins. We now turn to tasks that require fast weight changes *during* the episode. In the "key-value lookup" task, the NVM-RL must store key-value pairs in its working memory, and later retrieve one of the values when prompted with the corresponding key. For this experiment, in addition to the delimiter `0`, we allocated 5 symbols as possible keys and another 4 as possible values. In each episode, two distinct keys and two distinct values were randomly selected. All symbols are ultimately represented by random activity patterns, so for the sake of clarity and without loss of generality we use `a` and `b` instead of numeric symbols to denote keys in this task. In each episode, the keys were interleaved with their corresponding values in the input sequence, followed by a prompt key and then the delimiter `0`. The value associated with the prompt was the target output. For example, with the input `[b,1,a,2,b,0]`, the NVM-RL should produce an output of `1` at the last step, since `b` was initially paired with `1` and then used as the prompt. An all-or-nothing reward of 1 was given when the output symbol at the last time-step of the episode was correct. Just for this task, we enabled associative learning in one pathway from `tmp` to `out`, so that the NVM-RL could store a key-value pair by moving the key into `tmp`, the value into `out`, and then performing associative learning between them. We also allowed an output window of three time-steps after the prompt had been presented. To receive a reward, the NVM-RL had to output the correct value in any one of those three time-steps, and the delimiter `0` in the other two. This was intended to give the NVM-RL more flexibility in determining its algorithmic strategy. Lastly, to deal with the comparatively long episode duration, we used relu in place of tanh in (6).

Fig. 5 shows the results of training the NVM-RL on key-value lookup. In the interest of time, we only measured generalization performance after training was finished in this experiment. Unlike the previous tasks, we found that this one was more challenging and in general the NVM-RL was not able to solve it perfectly, although performance was usually still much better than if outputs had been selected randomly. On inspection we found that the NVM-RL usually converged to a sub-optimal strategy, such as always echoing the value from the second pair, which will be correct half of the time. We also found that the NVM-RL was more susceptible to overfitting than in previous tasks.

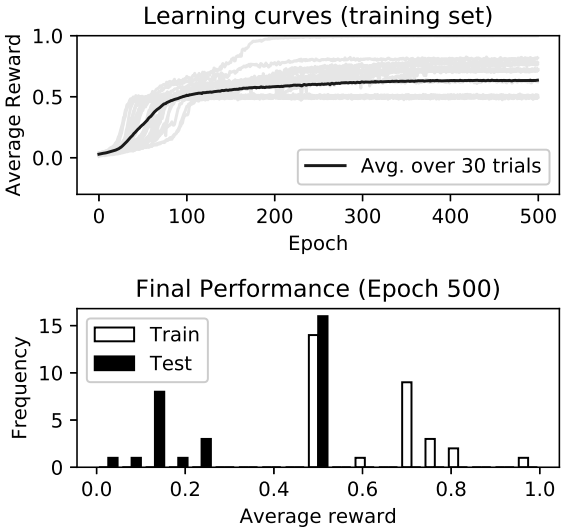

Fig. 5. Performance on key-value lookup. *Top*: Learning curves on the training data during policy ascent. *Bottom*: Histogram of final performance after training was complete on both the training data and testing data.

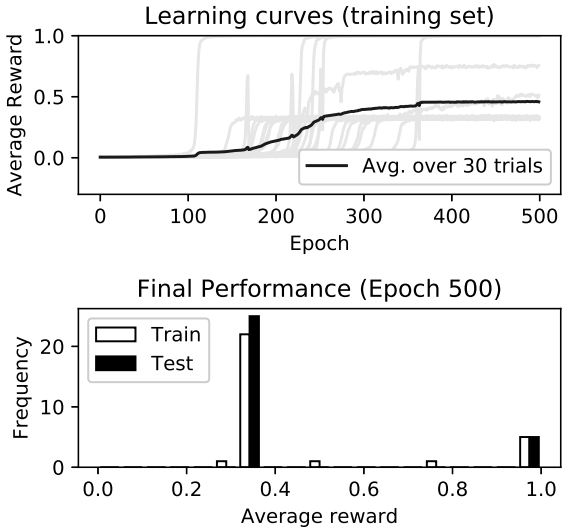

Fig. 6. Performance on list reversal. *Top*: Learning curves on the training data during policy ascent. *Bottom*: Histogram of final performance after training was complete on both the training data and testing data.

E. Reverse

Although key-value lookup involves fast associative learning during an episode, it does not require the tape-based memory emulation described earlier, which is important to make the NVM-RL truly relevant to program induction. To this end, the final task we considered was list reversal. For this task, a `0`-terminated sequence of symbols was presented to the NVM-RL, and after presentation was complete, it was expected to produce a reversed version of that sequence as output. Similar to key-value lookup, we allowed the delimiter `0` to appear in an output window that was one time-step longer than necessary, to provide the NVM-RL more flexibility in its solution. The sparse all-or-nothing reward was given if the NVM-RL output during the output window (after removing occurrences of `0`) contained the entire input list contents in reverse order and nothing else. We used an architectural configuration similar to key-value lookup, but instead of a normal register `tmp`, we used a read-write head layer `m`, with two additional auto-associative pathways `inc` and `dec`, as described earlier. We limited list length to 4.

Since key-value lookup had proved rather challenging for a more generic NVM-RL configuration, we also made certain modifications to constrain the search space based on our prior knowledge of the task, to see if performance improved. Specifically, we limited the size of the address space to equal the list length (i.e., 4), and also limited $|\mathcal{A}|$ to 4 so that the same symbols `0` through `3` could be conceptually "typecasted" as either memory addresses or list contents. The input list contents were drawn with replacement from the symbols `1` through `3`. We also removed some pathways that we knew were not necessary to solve the task (such as the ones from other layers to `inp`), so that the set of choices available to the controller at each time-step was smaller.

Fig. 6 shows the results of 30 independent policy search trials. Although the NVM-RL frequently failed to learn the task perfectly, it sometimes reached perfect success rate on both training and testing data, and was less susceptible to overfitting than key-value lookup.

We also used this task to explore the effect of regularization term D (see (18) and (20)), i.e. the controller output distribution variance, in more depth. Fig. 7 shows the behavior of D during training for all 30 trials, and compares it with average rewards in three representative examples. In many cases (although not always), rapid transitions in reward from near-zero to near-one were accompanied by rapid transitions in distribution variance down to near-zero. This effect is also visualized in Fig. 8, where we show the per-epoch changes in reward vs. per-epoch changes in D. Small changes in reward exhibit little correlation with changes in D, but large changes are negatively correlated. Based on these results, we hypothesize that including D in the optimization can gradually destabilize intermediate solutions that have settled at sub-optimal reward plateaus, sometimes leading to higher-reward solutions by the end of the training process. However, Fig. 8 contains too few datapoints with large $|\Delta\mathbb{E}[R_0]|$ for a definitive conclusion, and more work is needed to test this hypothesis.

IV. Discussion

We have shown that it is possible to use fast weights and recurrent policy gradients with sparse rewards to train the NVM-RL, a purely neural program induction architecture. We explained how the NVM-RL emulates both register- and tape-based memory, and showed empirically that it can learn to solve a number of algorithmic list processing tasks. The significance of these results is that RL-based NPI in a purely neural architecture may have increased relevance to program

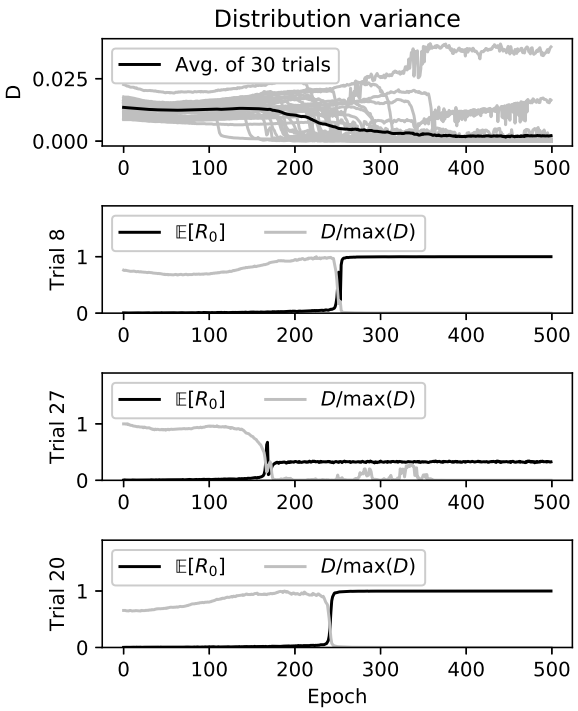

Fig. 7. *Top:* Distribution variance D during list reversal training for each trial (light grey) and averaged over all trials (black). *Others:* Three representative learning curves (average training data reward, an empirical estimate of $\mathbb{E}[R_0]$), superimposed on the corresponding distribution variance (D) at each epoch. For easier comparison with $\mathbb{E}[R_0]$, the D values were rescaled to a [0, 1] range by dividing by the maximum value over the course of training.

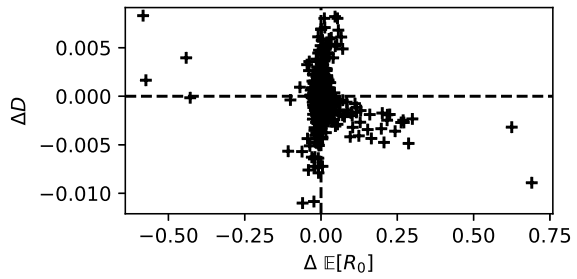

Fig. 8. Changes in empirical average training reward from one epoch to the next ($\Delta\mathbb{E}[R_0]$) versus changes in distribution variance from one epoch to the next (ΔD). The datapoints are collected from all epochs in all trials of list reversal.

induction in biological organisms, and it opens up new possibilities for solving general, high-level cognitive tasks with artificial neural systems. Based on these results, RL may be more competitive than previously thought, compared to the supervised learning paradigm more common in NPI. Using RL for NPI may afford the learning system with more flexibility in determining an algorithmic solution strategy.

Although our results are encouraging, in that learning always worked very effectively for the simplest tasks that we studied, there are many avenues for improvement. Future work should focus on larger problem instances and test whether an NVM-RL trained on one sequence length can generalize to other lengths. In addition, we found that the NVM-RL could not consistently achieve perfect success on the most complex tasks we attempted (key-value lookup and list reversal). Future work should focus on more modern and sophisticated RL methods, such as advantage function approximation and trust-region policy optimization [18], and reduce the reliance on task-specific architectural constraints like those imposed for list reversal. Along those lines, it would be interesting to derive or learn λ from the properties of a dataset rather than setting it heuristically. We are hopeful that future versions of the NVM-RL approach will use less task-specific engineering, scale to larger problem instances, and take further steps towards modeling program induction in biological systems.

References

[1] S. Reed and N. De Freitas, "Neural programmer-interpreters," in *ICLR*, 2016.

[2] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

[3] J. Devlin, R. R. Bunel, R. Singh, M. Hausknecht, and P. Kohli, "Neural program meta-induction," in *Advances in Neural Information Processing Systems*, 2017, pp. 2080–2088.

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

[5] Z. Zheng, X. Wu, and J. Weng, "Emergent neural turing machine and its visual navigation," *Neural Networks*, vol. 110, pp. 116–130, 2019.

[6] D. O. Hebb, *The organization of behavior: A neuropsychological theory*, 1949.

[7] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[8] J. Ba, G. E. Hinton, V. Mnih, J. Z. Leibo, and C. Ionescu, "Using fast weights to attend to the recent past," in *Advances in Neural Information Processing Systems*, 2016, pp. 4331–4339.

[9] I. Schlag and J. Schmidhuber, "Gated fast weights for on-the-fly neural program generation," in *NIPS Metalearning Workshop*, 2017.

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

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

[12] E. Salinas and T. J. Sejnowski, "Gain modulation in the central nervous system: where behavior, neurophysiology, and computation meet," *The Neuroscientist*, vol. 7, no. 5, pp. 430–440, 2001.

[13] W. H. Mehaffey, B. Doiron, L. Maler, and R. W. Turner, "Deterministic multiplicative gain control with active dendrites," *Journal of Neuroscience*, vol. 25, no. 43, pp. 9968–9977, 2005.

[14] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Recurrent policy gradients," *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 620–634, 2010.

[15] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson Addison Wesley, 2007.

[16] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2219–2225.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.