Neural Autonomous Computing Agent for Deep Model-free Reinforcement Learning

Victor Kolev

Scientific Advisers:

Rafael Rafailov Svetlin Penkov

Abstract

Memory-augmented neural networks have shown great potential in solving algorithmic and simple tasks, yet they have not been thoroughly tested on classic reinforcement learning problems, such as ATARI games, notwithstanding the abundance of similarities between them and a von Neumann architecture, or even the human hippocampus and cerebral cortex. In this paper, we show preliminary results that, when incorporated in RL algorithms, the Differentiable Neural Computer, the paramount memory-augmented machine learning architecture, achieves outstanding performance both in terms of training efficiency and in robustness to noise. Furthermore, we believe that this model is key for surpassing human-level performance in computational tasks, such as the ATARI game Montezuma's Revenge. Moreover, the DNC has potential to be able to train directly from ATARI RAM state, rather than pixel inputs, which is currently an unsolved problem. Finally, the external memory implementation in the model allows for extensive analysis of the data structures stored there, which enables us to gain invaluable insight into the processes inside of the black box that is a neural network.

Contents

1 **Introduction** **3**

2 **Preliminaries** **4**
- 2.1 Reinforcement Learning . . . 4
- 2.2 Recurrent Neural Networks . . . 5
- 2.3 Turing Machines . . . 6

3 **Proximal Policy Optimization** **8**
- 3.1 Motivation for choosing PPO . . . 8
  - 3.1.1 Policy Optimization . . . 8
  - 3.1.2 Q-learning . . . 9
- 3.2 The PPO algorithm . . . 9

4 **Differential Neural Computer** **11**
- 4.1 Controller . . . 12
- 4.2 Interfacing with the memory . . . 13
- 4.3 Read and Write Heads . . . 14

5 **Experiments** **17**

6 **Results** **19**

7 **Conclusion and Future Development** **20**

1 Introduction

While neural networks have been quite successful at function approximation, they have, for the most part, overlooked two of the main principles of computing [19] - control flow (branching) and external memory, instead relying solely on arithmetic operations. While sufficient for some tasks, this limitation is what causes poor learning in some RL problems such as the ATARI game Montezuma's Revenge [12], which are computational in nature, rather than arithmetic.

A direct implementation of branching would introduce a point of non-differentiablility, due to the fact that the condition itself is indifferentiable, which impedes the backpropagation pass and makes the use of gradient descent highly impractical. Nevertheless, integrating dynamic external memory has been achieved - the Differentiable Neural Computer (DNC) architecture [4] has managed to implement a memory matrix, akin to random-access memory in a traditional computer. Furthermore, as a result of a content-based lookup operation, DNCs can emulate a conditional statement. Moreover, all of this is attained whilst maintaining full end-to-end differentiablility, hence it can be easily and efficiently trained using backpropagation.

Taking into account that the Recurrent Neural Networks used in the DNC are Turing complete, this means that the architecture can be regarded as a general problem solver, a von Neumann architecture with learning capability. Additionally, the DNC decouples the learning process and the memory operations, unlike RNNs such as Long-Short Term Memory networks, which enables it to solve problems of exponentially greater complexity.

In a sense, the DNC bears resemblance to the human hippocampus and cerebral cortex - the DNC controller acts as the hippocampus, since it is responsible for decision making and interacting with the memory, which is, in turn, the cerebral cortex. This analogy between the two provoked this research into the capabilities of the architecture to learn by interaction in an environment, just as a human would. In addition, it is shown that the rewards in reinforcement learning are functionally comparable to dopamine [2], further strengthening the resemblance.

This application of the DNC in reinforcement learning has been briefly examined by the original authors, yet they only touch on hand-coded, simple problems, which by no means exhaust the vast potential of the architecture. To the best of our knowledge, the DNC has yet not been tested on classic autonomous agent RL tasks.

In this paper we show initial results that the Differentiable Neural Computer exhibits great performance on benchmark reinforcement learning tasks, such as classic control problems and ATARI games, training with outstanding speed. The principal objective of our work is to gain insight into the procedures that the neural network learns and stores in memory and develop a stable and robust RL architecture that is capable of deriving not only arithmetic routines, but also computational operations.

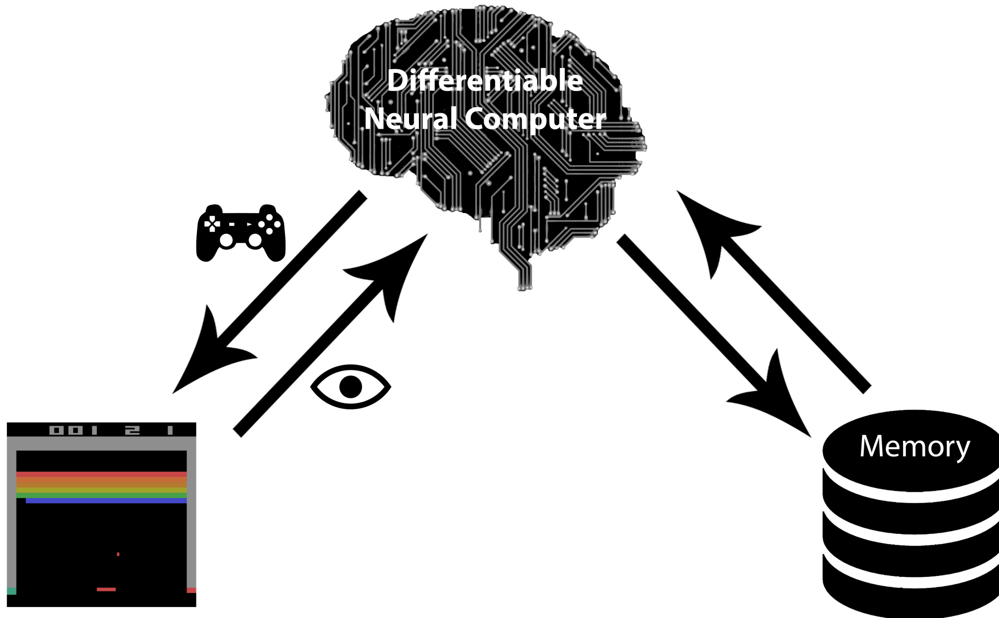


Figure 1: Illustration of the our memory-augmented RL architecture.

2 Preliminaries

2.1 Reinforcement Learning

We consider a standard reinforcement learning setting - an agent interacts with an environment, aiming to maximize total discounted reward. Formally, these interactions are modelled as a Markov Decision Process, defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r)$, where $\mathcal{S}$ is the state space; $\mathcal{A}$ is the set of actions; $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}^+$ is a the probability function, mapping a state s and action a to next state s'; $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward that the agent receives at each timestep of the episode. The agent's actions are determined by a policy π, which we model as a neural network, parameterized by a vector θ.

The objective of the agent is to maximize his expected reward for the episode

$$\sum_{t=1}^{T} E_{(s_t,a_t)\sim p(s_t,a_t)}[r(s_t, a_t)]$$

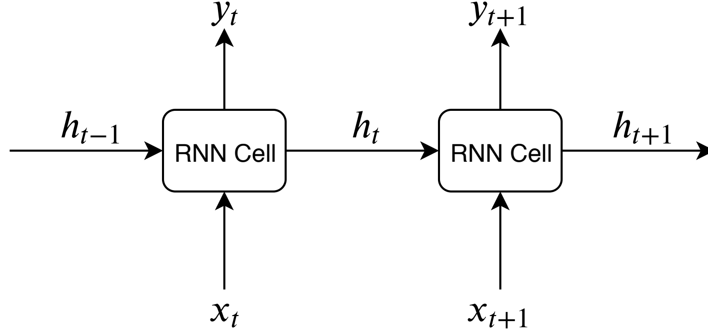
We define a trajectory τ as a sequence $(s_1, a_1, s_2, a_2, \dots s_T, a_T)$ and the objective can be rewritten as

$$\max_{\theta} : E_{\tau\sim p(\tau)}[r(\tau)]$$

Therefore, by interacting with the environment and modifying its policy accordingly, the agent learns an optimal policy π_{θ^*}.

2.2 Recurrent Neural Networks

Recurrent neural networks are a class of models, where connections between nodes (RNN cells) can be represented as a directed graph along a temporal sequence. In their basic form, RNNs are a recursive application of a function, approximated by a neural network, taking an input at each timestep of the sequence along with a recurrent hidden state, determined by the previous timestep. In this paper, we utilize extensively LSTM networks [6], which have been developed to capture long-term temporal dependence, a task that is virtually impossible to accomplish with traditional RNNs in practice.



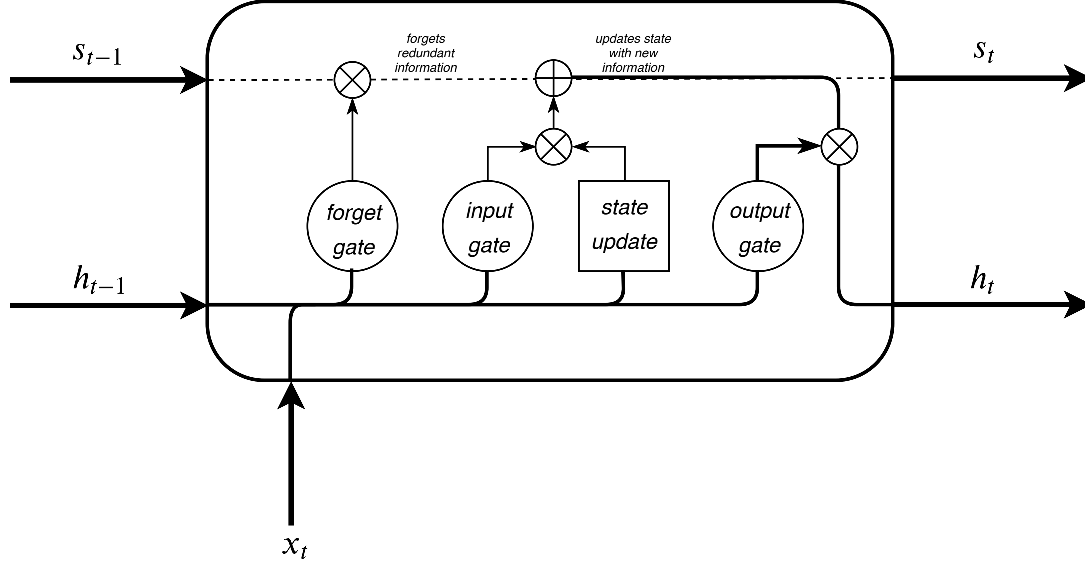
A standard recurrent neural network model.

Each layer of a LSTM cell consists of an input, forget, output and state gates, with an additional internal cell state that is passed along all timesteps. The input gate determines what part of the vector will be added to the cell state, while the forget gate is responsible for nullifying redundant and unnecessary information in the cell state of the previous timestep. A third state gate transforms the input into a practical and convenient for the cell state form. The three gates explained so far govern the cell state update of the LSTM. The last gate, the output gate, governs the transition from the newly derived cell state to the layer output.

The model can be formalized as follows:

$$
\begin{aligned}
i_t^l &= \sigma(W_i[x, h_t^{l-1}, h_{t-1}^l] + b_i) \\
f_t^l &= \sigma(W_f[x, h_t^{l-1}, h_{t-1}^l] + b_f) \\
s_t^l &= f_t^l s_{t-1}^l + i_t^l tanh(W_s[x, h_t^{l-1}, h_{t-1}^l] + b_s) \\
o_t^l &= \sigma(W_o[x, h_t^{l-1}, h_{t-1}^l] + b_o) \\
h_t^l &= o_t^l tanh(s_t^l)
\end{aligned}
$$

where i_l^t is the input gate at timestep t and layer l; f_l^t is the forget gate; s_l^t is the LSTM state; o_l^t is the output gate; h_t^l is the LSTM output; σ is the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$; $tanh$ is the hyperbolic tangent function $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.



The structure of a LSTM cell.

At first glance, utilizing a RNN model in RL would seem unnecessary, given the Markov property which states that knowing the current state is sufficient for proceeding forward in the process, therefore past states can be forgotten. However, this holds true solely for first order MDPs and full observability, which is often not the case. Furthermore, even when these conditions are satisfied, system parameters are still unknown and have to be derived though observation; upon the use of a RNN, we would get a cleaner signal and more precise dependencies in the data.

2.3 Turing Machines

A Turing machine is a mathematical model of computation that defines an abstract machine, which manipulates symbols on a strip of tape according to a set of rules, thereby simulating the logic of an arbitrary algorithm through the rule table. In essence, it is equivalent to a modern computer, insofar as it can solve any problem that it can, however greatly simplified operations-wise, as to accommodate mathematical analysis and proofs.

More formally, a Turing machine is defined as a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where

- Q is the (finite) set of states that the turing machine can operate in
- Σ is a (finite) set of possible input symbols
- Γ is the complete set of symbols, $\Sigma \subset \Gamma$
- δ is the transition function; if defined, it takes the form of:

  $$\delta(q, X) = (p, Y, D)$$

  q is the current state, $X \in \Sigma$ is the input symbol, $Y \in \Gamma$ is the symbol being written in place of X and D is the direction, in which the head will move, either left or right.
- $q_0 \in Q$ is the initial state of the Turing machine
- B is the blank symbol, $B \in \Gamma, B \notin \Sigma$, and is the only member of Γ that can occur infinitely often on the tape
- F is the set of termination states

In much the same way, RNNs receive an input at each timestep, which they manipulate given their internal state and weight matrices, in which the rule table is encoded. It is shown by Siegelmann et al. [17] that RNN models are indeed Turing complete, meaning that they can, provided carefully chosen weights, simulate any computer program. This is however a purely theoretical postulation and can only occur when the hidden state of the network is split into two stacks and the precise interaction between them is encoded in the weights, thereby simulating a queue data structure. In practice, however, this is hardly feasible and becomes exponentially more difficult as task complexity increases.

By augmenting a recurrent neural network with fully differentiable external memory with content-based search, the resulting model is much more akin to the von Neumann architecture and so, not only is it capable of simulating any real computer and it's capabilities, but it is also much more efficient, since it segregates the learning task from the memory interactions, thereby greatly decreasing the complexity of tuning the weights.

3 Proximal Policy Optimization

3.1 Motivation for choosing PPO

In order to achieve the RL objective, the agent optimizes a policy function, an artificial neural network with parameters θ. There are two ways to achieve this without explicitly knowing the system dynamics (model-free algorithms) - policy optimization and Q-learning.

Before elaborating on the two methods, let us first introduce some necessary nomenclature:

- Value function:

  $$V(s_t) = \sum_{t'=t}^{T} E_{\pi_\theta}[\gamma^{t'-t} r(s_{t'}, a_{t'})|s_t]$$

  where π_θ is the current policy; T is the time horizon (when considering infinite time horizon problems $\lim_{t' \to \infty} \gamma^{t'-t} = 0$ for $\gamma \in (0, 1)$); γ is a discount factor.

  The value function can be interpreted as the average reward left after the current state before the termination of the episode. Naturally then, states with a higher value function are more advantageous for the agent.

- Q-function:

  $$Q(s_t, a_t) = \sum_{t'=t}^{T} E_{\pi_\theta}[\gamma^{t'-t} r(s_{t'}, a_{t'})|s_t, a_t]$$

  The Q-function differs from $V(s_t)$ in that, rather than computing the average reward left given the current state, it provides the rewards left after being at a given state and taking a certain action. We can therefore derive the following two equations:

  $$V(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)}[Q(s_t, a_t)]$$
  $$Q(s_t, a_t) = r(s_t, a_t) + \gamma V(s_{t+1})$$

- Advantage function:

  $$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

  The advantage function signifies how much an action is better (or worse) than the average action.

3.1.1 Policy Optimization

Policy optimization algorithms model the policy as a function mapping from states and actions to a probability distribution and they optimize a neural network to approximate this function.

Therefore, the fundamental objective of any policy gradient algorithm is to derive

$$\theta^* = \underset{\theta}{\text{argmax}} : E_{\tau \sim \pi_\theta}[r(\tau)]$$

Prevalent policy optimization algorithms include Policy Gradients, actor-critic methods (A2C, A3C)[11], Trust Region Policy Optimization [14] and Proximal Policy Optimization [16].

3.1.2 Q-learning

Q-learning derives the optimal policy indirectly by selecting the action that maximizes the Q-function at the present state.

$$\pi(a_t|s_t) = \begin{cases} 1 - \varepsilon & a_t = \underset{a'_t}{\text{argmax}}\, Q(s_t, a'_t) \\ \varepsilon & \forall a_t \end{cases}$$

where ε is a small number, which allows for exploration by selecting a random action $a_t \sim \mathcal{U}(\mathcal{A})$ with probability ε .

The seminal and most widely used Q-learning algorithm is Deep Q-Network (DQN) [12], which in its classic form approximates the Q-function with a deep neural network, trained in a supervised manner through fitted Q-iteration.

While Q-learning algorithms have shown outstanding results, they are more prone to instability than policy optimization, since they learn a Bellman equilibrium, which is much more arduous than a state-action mapping. For that reason, we have opted against using them.

3.2 The PPO algorithm

The Proximal Policy Optimization algorithm (PPO) [16] performs incredibly well and has good sample efficiency, all whilst maintaining relative simplicity, making it easy to tune. It is due to those characteristics that it has become an industry standard and benchmark for any new algorithm, therefore it is a natural choice for bedrock of our memory-augmented algorithm.

PPO originates from the Monotonic Improvement Theory, which is created to address some innate problems with Policy Gradient methods, namely the inequivalence between gradient descent steps taken in parameter space and how they impact the policy. Due to the on-policy nature of Policy Gradients, one cannot reuse trajectories from old policies and one needs to generate samples after each update. Whilst this can be mitigated through importance sampling, the weight term that is added is incredibly unstable and tends to either vanish or explode (for details, view Appendix A). PPO solves this by deriving an objective, in which a single sample for the importance weight at each timestep is used.

Rather than the traditional policy optimization objective (maximize discounted rewards), PPO utilizes a surrogate objective, which, on a high level, seeks to maximize the improvement of the policy, all whilst keeping within a certain predetermined distance from the old policy, the samples from which are used to evaluate the objective. This bound on the difference between the policies can be achieved by computing their KL-divergence, however a much computationally cheaper and just as effective in practice operation is to clip the ratios of the probabilities at each timestep. In this manner, the steps taken reflect distance in policy space, instead of parameter space, which greatly aids convergence success.

The PPO surrogate objective is

$$\mathcal{L}_{\theta_k}(\theta) = E_{\tau \sim \pi_{\theta_k}} \left[\sum_{t=0}^{T} \min\left[R_t(\theta)\hat{A}_t^{\pi_{\theta_k}}, clip(R_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t^{\pi_{\theta_k}}\right] \right]$$

where

- $R_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ are the probability ratios;
- θ_k is the last policy, from which samples have been generated;
- $clip(k, a, b) = \text{argmin}_{c \in [a,b]} : |c - k|$ gives the closest number to k within the interval $[a, b]$;
- ε is the clip parameter;
- $\hat{A}_t^{\pi_{\theta_k}}$ are the estimated advantages of the trajectories from θ_k. We employ a critic network with parameters φ to compute the value functions (updated using standard mean squared error loss; targets are Monte Carlo estimates), from which we derive the advantages by utilizing Generalized Advantage Estimation [15]

  $$\hat{A}_t = \sum_{k=0}^{T-t-1} (\gamma\lambda)^k \left(r_{t+k} + \gamma V^{\varphi}(s_{t+k+1}) - V^{\varphi}(s_{t+k})\right)$$

  where γ is the discount factor and λ is a parameter controlling the bias/variance trade-off in the estimation of the advantages.

  It is worth noting that although the advantages are evaluated on trajectories from π_{θ_k}, their estimates are computed with the parameters of the current network φ.

The policy is optimized by taking gradient descent steps via the Adam algorithm [9] for K epochs before updating the target policy and generating new trajectories.

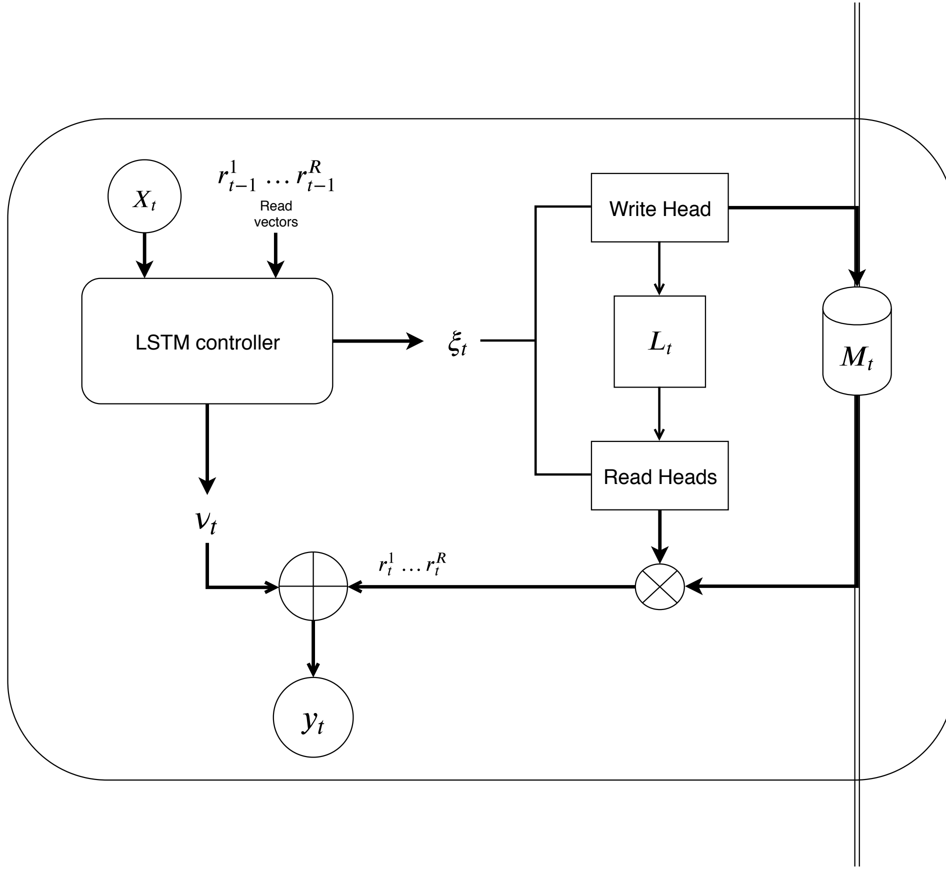
Algorithm 1: The PPO algorithm

Initialize θ, φ, θ_k;
for *m in 1:Iterations* **do**
  generate (partial) trajectories τ on policy π_{θ_k};
  for *K in 1:Epochs* **do**
    estimate advantages $\hat{A}^{\pi_{\theta_k}}$ with parameters φ;
    evaluate $\mathcal{L}_{\theta_k}(\theta)$;
    compute $L^V(\varphi) = \frac{1}{2N}\sum_{i=1}^{N}(V_i^{\varphi} - y_i)^2$;
    calculate loss $L = -\mathcal{L}_{\theta_k}(\theta) + L^V(\varphi)$;
    take gradient descent step via Adam:
      $\theta \leftarrow \theta - \alpha\nabla\theta$;
      $\varphi \leftarrow \varphi - \alpha\nabla\varphi$;
  end
  Update policy parameters: $\theta_k \leftarrow \theta$;
end

4 Differential Neural Computer

Here we describe the DNC architecture in detail.

At the heart of the DNC model is a controller network, for which we use a standard deep LSTM architecture. It is responsible for processing the input data and managing interactions with the memory matrix. Read operations are handled by R read heads, each of which computes a read weighting $w_t^{r,i}$, given which data is read from the memory matrix $M_t \in \mathbb{R}^{N\times W}$, according to either a content-based lookup mechanism or based on the order information was written in. A single write head controls the writing procedure thought the write weighting w_t^w, utilizing dynamic memory allocation, which determines the locations where information can be overwritten.



The DNC model structure

4.1 Controller

The controller is what is responsible for both managing read/write operations and deriving the solution of the task. At each timestep t, it receives R read vectors of length W, one from each read head, along with the input vector X_t, which are all concatenated to form the LSTM input.

$$\chi_t = [X_t, r_{t-1}^1, r_{t-1}^2, \dots r_{t-1}^R]$$
$$X_t \in \mathbb{R}^X, r_{t-1}^i \in \mathbb{R}^W \quad i \in [1:R]$$

The input is then passed to all layers of the LSTM as follows:

$$
\begin{aligned}
i_t^l &= \sigma(W_i[\chi_t, h_t^{l-1}, h_{t-1}^l] + b_i) \\
f_t^l &= \sigma(W_f[\chi_t, h_t^{l-1}, h_{t-1}^l] + b_f) \\
s_t^l &= f_t^l s_{t-1}^l + i_t^l tanh(W_s[\chi_t, h_t^{l-1}, h_{t-1}^l] + b_s) \\
o_t^l &= \sigma(W_o[\chi_t, h_t^{l-1}, h_{t-1}^l] + b_o) \\
h_t^l &= o_t^l tanh(s_t^l)
\end{aligned}
$$

The DNC may use either the output of all layers $[h_t^1, h_t^2, \ldots, h_t^L]$ or only that of the last layer h_t^L. We assume all outputs are used.

The network then diverges to two linear layers, akin to the Dueling Networks architecture [20] - one layer transforms the LSTM output to the DNC output size $\nu_t = W_y[h_t^1, h_t^2, \ldots, h_t^L]$, $\nu_t \in \mathbb{R}^y$ and the other layer returns an interface vector $\xi_t = W_\xi[h_t^1, h_t^2, \ldots, h_t^L]$, $\xi_t \in \mathbb{R}^{2W \times R + 4W + 7R + 3}$, on which we elaborate later.

Finally, ν_t is combined with the newly read vectors to form the DNC output.

$$y_t = \nu_t + W_r[r_t^1, r_t^2, \ldots, r_t^R]$$

4.2 Interfacing with the memory

The aforementioned interface vector ξ_t carries the necessary instructions for the read and write heads. It is split as follows:

$$
\begin{aligned}
\xi_t = &[\mathbf{k}_t^{r,1}, \ldots, \mathbf{k}_t^{r,R}, \hat{\beta}_t^{r,1}, \ldots, \hat{\beta}_t^{r,R}, \hat{\mathbf{m}}_t^{r,1}, \ldots, \hat{\mathbf{m}}_t^{r,R}, \hat{\mu}_t^1, \ldots, \hat{\mu}_t^R, s_t^{\phi,1}, \ldots, s_t^{\phi,R}, s_t^{b,1}, \ldots, s_t^{b,R}, \\
&\mathbf{k}_t^w, \hat{\beta}_t^w, \hat{\mathbf{m}}_t^w, \hat{e}_t, v_t, \hat{f}_t^1, \ldots, \hat{f}_t^R, \hat{g}_t^a, \hat{g}_t^w]
\end{aligned}
$$

where

- $\mathbf{k}_t^{r,i} \in \mathbb{R}^W$ is the lookup key for read head i; lookup keys act as search terms and are used in the content-based lookup mechanism
- $\beta_t^{r,i} = oneplus(\hat{\beta}_t^{r,i}) \in \mathbb{R}^1$ is the key strength of read head i; keys are assumed to have different impact and this is reflected in their key strength
- $\mathbf{m}_t^{r,R} = \sigma(\hat{\mathbf{m}}_t^{r,R}) \in \mathbb{R}^W$ is the search mask for read head i; the lookup key might only include a partial information to seek in the rows of the memory matrix, similar to keywords. Subsequently, it is imperative that the rest of the key is masked, as to prevent noise in the lookup operation.

[1] $oneplus(x) = 1 + ln(1 + e^x)$

- $\mu_t^i = softmax(\hat{\mu}_t^i) \in \mathbb{R}^3$ [2] is a distribution of the 3 modes that the read head can utilize, namely content-based reading, forward temporal sequence reading and backward temporal reading.
- $s_t^{\phi,i}$ is the sharpness parameter for the forward temporal sequence reading operation for read head i
- $s_t^{b,i}$ is the sharpness parameter for the backward temporal sequence reading operation for read head i
- $\mathbf{k}_t^w \in \mathbb{R}^W$ is the lookup key for the write head
- $\beta_t^w = oneplus(\hat{\beta}_t^w) \in \mathbb{R}$ is the key strength of the write head
- $\mathbf{m}_t^w = \sigma(\hat{\mathbf{m}}_t^w) \in \mathbb{R}^W$ is the search mask for the write head
- $e_t = \sigma(\hat{e}_t) \in \mathbb{R}^W$ is the erase vector, which governs how information is erased
- v_t is the write vector, containing information that is to be written
- $f_t^i = \sigma(\hat{f}_t^i) \in \mathbb{R}$ is the free gate of read head i, which determines whether the most recently read-from location by the read head can be freed or not
- $g_t^a = \sigma(\hat{g}_t^a) \in \mathbb{R}$ is the allocation gate, which controls how much will be written based on the allocation manager's output and how much based on the content lookup mechanism
- $g_t^w = \sigma(\hat{g}_t^w) \in \mathbb{R}$ is the write gate, which regulates how strong the write weighting is.

4.3 Read and Write Heads

Reading from Memory

The purpose of a read head is to efficiently obtain the demanded from the controller information from the memory matrix. To do so, each read head outputs a read weighting $w_t^{r,i} \in \Delta_N$, where $\Delta_N := \{\alpha \in \mathbb{R}^N | \sum_{i=1}^N \alpha_i \leq 1\}$ is the N-dimensional unit simplex. This weighting determines the locations from which the head will read data from. Therefore, the read vector of the head i is:

$$r_t^i = M_t^T w_t^{r,i}$$

[2] $softmax(\mathbf{x})[i] = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$ for $i \in 1:n$

Writing in Memory

In much the same way, the write head computes a write weighting $w_t^w \in \Delta_N$, which encodes the locations to be overwritten. The write head receives from the controller a write vector v_t and an erase vector e_t, which are used to update the matrix. In addition, based on the formerly read from location and free gates, dictating whether that data is still necessary, the write head constructs the vector $\psi_t \in \mathbb{R}^N$, which deallocates unneeded information from memory.

$$M_t = M_{t-1} * \psi_t \mathbf{1}^T * (\mathbf{E} - w_t^w e_t^T) + w_t^w v_t^T$$

where "$*$" is element-wise multiplication and $\mathbf{E} \in \mathbb{R}^{N \times W}$ is a matrix of ones.

Content-based Lookup

In order to be able to efficiently search in memory, a mechanism must be in place that compares stored data to a search key. This operation is the content-based lookup, which uses a key $\mathbf{k} \in \mathbb{R}^W$, a key mask $\mathbf{m}$ that prevents noise from impacting the search process and, finally, a key strength β, which signifies the importance of the keyword.

We define the content-based lookup operation as:

$$\mathcal{C}(M, \mathbf{k}, \mathbf{m}, \beta) = softmax\Big[\mathcal{D}(\mathbf{k} * \mathbf{m}, M * \mathbf{1m})\beta\Big]$$

where $\mathcal{D}(u, v)$ is the cosine similarity

$$\mathcal{D}(u, v) = \frac{u^T v}{|u||v| + \epsilon} = \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2}\sqrt{\sum_{i=1}^n v_i^2} + \epsilon}$$

$u, v \in \mathbb{R}^n$ are vectors and ϵ is added for numerical stability in computation.

Dynamic Memory Allocation

Just like computers with Random-access memory, the DNC utilizes a dynamic memory allocation mechanism, which dictates where new information should be written given the current state.

Firstly, the allocation manager receives the read weightings $w_{t-1}^{r,i}$, $i \in 1 : R$ from the previous timestep, as well as R free gates $f_t^i \in [0, 1]$ from the interface vector and computes the vector ψ_t, which shows how much of the most recently read-from location will remain after the memory update.

$$\psi_t = \prod_{i=1}^{R} 1 - w_{t-1}^{r,i} f_t^i \qquad \psi_t \in [0, 1]^N$$

The next step for the allocation manager is to update its usage vector u_{t-1}, which stores how much of each memory location (row) is used. Since the last update, additional memory has been written at locations determined by w^w_{t-1} and some data that has been read can be discarded. Therefore, the update takes the following form:

$$u_t = (u_{t-1} + w^w_{t-1} - u_{t-1} * w^w_{t-1}) * \psi_t$$

After that, an index vector ς_t is constructed that contains the memory locations in ascending order of usage ($\varsigma_t[1] = \text{argmin}_i\, u_t[i]$), so that the rows that are least used get more data written. Finally, the allocation vector a_t, signifying where new data should be written, is calculated as:

$$a_t\big[\varsigma_t[j]\big] = \Big(1 - u_t\big[\varsigma_t[j]\big]\Big) \prod_{i=1}^{j-1} u_t\big[\varsigma_t[i]\big]$$

Write Weighting

Along with the allocation vector, the write head can also choose to write based on content and so, we also have a content write weighting

$$c^w_t = \mathcal{C}(M_{t-1}, \mathbf{k}^w_t, \mathbf{m}^w_t, \beta^w_t)$$

In order to interpolate between the content weighting and the allocation vector, the allocation gate g^a_t, outputted by the controller, is utilized. Furthermore, other than writing based on usage or content, the head might choose not to write at all, therefore the gate g^w_t is the final component in the computation of the write weighting.

$$w^w_t = g^w_t \Big(g^a_t a_t + (1 - g^a_t) c^w_t\Big)$$

Temporal Memory Linkage

In some operations, the order data was written in might be important and, for that reason, a record of this information is kept in the Temporal Link Matrix $L_t \in [0,1]^{N \times N}$, with $L_t[i,j]$ showing to what degree location i was written to after location j.

First we define $p_t \in \Delta_N$ as a vector, whose i-th element signifies the extent to which location i is the last one written to. The value of p_t at the next timestep is computed as:

$$p_t = \left(1 - \sum_{i=1}^{N} w^w_t[i]\right) p_{t-1} + w^w_t$$

The update of the temporal link matrix should remove old and redundant links and add new ones, therefore

$$L_t[i,j] = \Big(1 - w^w_t[i] - w^w_t[j]\Big) L_{t-1}[i,j] + w^w_t[i] . p_{t-1}[j]$$

Having that information at its disposal, the read head can generate a forward and backward weighting ϕ_t^i and b_t^i, based on the respective direction of the sequence they trace.

$$\phi_t^i = L_t w_{t-1}^{r,i} \qquad b_t^i = L_t^T w_{t-1}^{r,i}$$

However, at each timestep, the temporal link matrix becomes more and more noisy, which hinders performance, consequently, to solve this, an additional sharpening step is integrated.

$$\mathcal{S}(\mathbf{u}, s)[j] = \frac{\mathbf{u}_j^s}{\sum_{i=1}^n \mathbf{u}_i^s}$$

Therefore, using the scalars outputted by the controller network, the sharpened weightings for read head i are

$$\phi_t^i = \mathcal{S}(L_t w_{t-1}^{r,i}, s_t^{\phi,i}) \qquad b_t^i = \mathcal{S}(L_t^T w_{t-1}^{r,i}, s_t^{b,i})$$

Read Weighting

Besides the temporally-determined weightings, the read head further utilizes content-based lookup, searching for a specific keyword. The i-th read head computes its content weighting as

$$c_t^{r,i} = \mathcal{C}(M_t, \mathbf{k}_t^{r,i}, \mathbf{m}_t^{r,i}, \beta_t^{r,i})$$

The final read weighting is interpolated according to the read mode given by the controller

$$w_t^{r,i} = \mu_t^i[1].c_t^i + \mu_t^i[2].b_t^i + \mu_t^i[3].\phi_t^i$$

5 Experiments

We examine and scrutinize the performance of the DNC on classic control and ATARI tasks, starting with Cartpole - a benchmark problem from the OpenAI Gym framework, in which the agent has to balance a pole of variable length (by default: 1m) on a cart, keeping it within a specified angle θ (by default: $12°$) from the upright position.

The baseline with which we compare the results of the DNC is that of a standard LSTM cell. Notwithstanding their generality and successes in other fields such as Natural language processing, LSTM networks are notoriously unstable in model-free reinforcement learning, as a result of which feedforward networks are usually preferred. This volatility in training LSTMs is in large part due to the credit assignment problem [10], consequently of which not only is it arduous to pinpoint the weights that are responsible for a specific outcome (structural assignment), but further the action at which timestep contributed to the final state of the system (temporal assignment).

Since the DNC uses a LSTM controller, it suffers from the same issues, which may even be exacerbated due to the larger network size, attributing to the necessity to learn an efficient reading and writing procedures.

In order to stabilize training and ensure reproducibility of our results, multiple measures and heuristics are applied:

- neural network size is kept to a minimum, respecting the computational difficulty of the task, as to alleviate the structural credit assignment problem;
- duelling networks: rather than employing two separate networks for policy and value estimation, a single controller (LSTM or DNC) is utilized, the output of which is subsequently passed onto linear layers, which are specific for their respective task;
- softmax temperature annealing: in order to assure exploration of state and action space, temperature is added to the softmax function at the initial stages of training, which is then annealed;
- to further encourage exploration, the entropy of the action distribution is subtracted from loss;
- gradient clipping: as a preventative measure against exploding gradients, their $L2$ norm is restricted to a certain value, which, if exceeded, leads to the gradient being clipped;
- value loss clipping: per the aversion of overestimation of the value functions outputted by the network, which would cause action loss to be reduced, a PPO-like clipped objective is utilized

$$L^V_{CLIP} = \frac{1}{2}\sum_{i=1}^{N}(min(\hat{V}_i^{\varphi} - y_i, clip(\hat{V}_i^{\varphi}, \hat{V}_i^{\varphi'} - \varepsilon, \hat{V}_i^{\varphi'} + \varepsilon) - y_i))^2$$

  where y_i are the target values, φ' are the parameters of the old policy, therefore the clipped objective restricts the value network from diverging within more than an ε distance away from the old policy.

In addition to testing for convergence speed, which we postulate would be much faster, owing to the capability to utilize memory and remember important data from the environment, such as the task parameters, we further test for resistance to input noise. Due to the bigger flexibility in training and computational structure of the DNC, we hypothesize that it would be more robust and would be able to solve the task consistently with higher levels of Gaussian noise present in the input state.

We run the DNC and LSTM on the basic `Cartpole-v1` environment from OpenAI Gym, adding Gaussian noise to the input of the network, in order to test for robustness during training. The variance of the noise is derived from the states of the update batch, taking on values

$\sigma^2 = k\text{Var}_{s\sim\pi_{\theta_k}}(s)$ where k is some constant. Additionally, we examine how robust the networks are during test time, after they are successfully trained.

6 Results

In the table below are described the results from the experiments. In all cases during training, the DNC architecture vastly outperforms the traditional LSTM, often improving the convergence speed by more than 60%. This confirms our postulation that the DNC is fundamentally a better and therefore faster learner. In addition, we observed strong correlation between successful learning and high memory usage, which means that a successful solution to the task is highly dependent on effective exploitation of the memory matrix. Furthermore, unlike the LSTM, the DNC, once having gathered enough experience, rapidly learns and converges to an optimal solution, which we attribute to learning those aforementioned efficient memory operation controls.

Noise	LSTM	DNC
0	23543	8860
$\frac{1}{3}\sigma$	31031	11192
σ	26992	11795
2σ	27147	11173
4σ	20967	10875

Figure 2: Average conversion speed in `Cartpole-v1` (more than 10 samples). Gaussian noise ($\mu = 0$, σ is a multiple of the standard deviation of a sample batch) is added to the input state to test for robustness during training.

Similarly, the DNC consistently achieves better results than the LSTM regarding the degradation in performance as noise is added to the input state at test time. This we again ascribe to the fact that the DNC can store structured data in its memory matrix, which aids to cancel the noise.

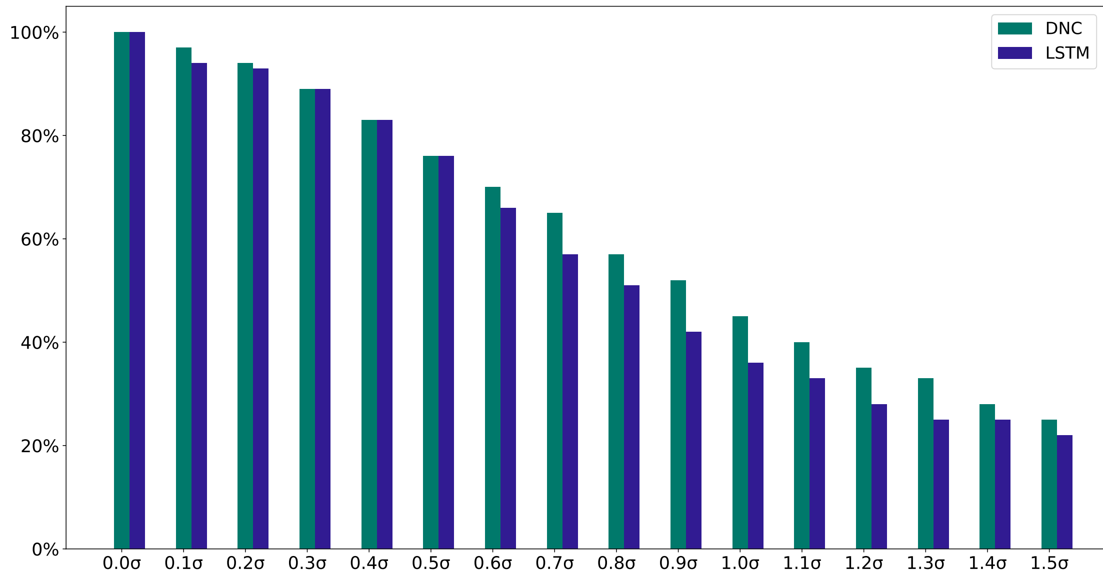


Figure 3: Performance of trained DNC and LSTM networks when noise is added to the state. σ is the standard deviation of the states from 5000 episodes, played with a random policy. 100% is the average reward achieved by the networks without noise, based on 1000 episodes.

7 Conclusion and Future Development

In this paper were discussed the applications of the Differentiable Neural Computer in deep model-free reinforcement learning. We have shown that great potential lies with the use of such memory-augmented architectures, stemming from their superiority in training speed, robustness and their ability to perform computational operations, as well as purely arithmetic.

Further development will be to scrutinize whether the DNC can infer a solution directly from pixels, just as is the case with human players of ATARI games. We shall test our architecture on actual compute tasks, namely Montezuma's Revenge, whose solution includes a conditional statement. We further wish to analyze how the data structures contained in the memory matrix compare with those in the optimal solution of the tasks. Finally, we believe, stemming from the similarities between our architecture and an actual computer and from the results so far, that this model could be able to learn ATARI games from RAM state, which has not been possible so far to the best of our knowledge.

References

[1] Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. Rl2: Fast reinforcement learning via slow reinforcement learning. arxiv, 2016. *arXiv preprint arXiv:1611.02779*.

[2] Glimcher, P. W. Understanding dopamine and reinforcement learning: The dopamine reward prediction error hypothesis. *Proceedings of the National Academy of Sciences 108*, Supplement 3 (2011), 15647–15654.

[3] Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401* (2014).

[4] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature 538*, 7626 (2016), 471.

[5] Ha, D., and Schmidhuber, J. World models. *arXiv preprint arXiv:1803.10122* (2018).

[6] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[7] Ilyas, A., Engstrom, L., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., and Madry, A. Are deep policy gradient algorithms truly policy gradient algorithms? *arXiv preprint arXiv:1811.02553* (2018).

[8] Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144* (2016).

[9] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[10] Minsky, M. Steps toward artificial intelligence. *Proceedings of the IRE 49*, 1 (1961), 8–30.

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

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529.

[13] Narvekar, S., and Stone, P. Learning curriculum policies for reinforcement learning. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent*

Systems (2019), International Foundation for Autonomous Agents and Multiagent Systems, pp. 25–33.

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

[15] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

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

[17] Siegelmann, H. T., and Sontag, E. D. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory* (1992), ACM, pp. 440–449.

[18] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society 2*, 1 (1937), 230–265.

[19] Von Neumann, J. First draft of a report on the edvac. *IEEE Annals of the History of Computing 15*, 4 (1993), 27–75.

[20] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

Appendix A

Advantages of the PPO algorithm

The main advantages of the PPO algorithm are its simplicity, its sample efficiency and robustness.

Sample Efficiency

The on-policy nature of policy gradient methods require them to generate new trajectories each time an update occurs. One way to alleviate this problem is to use importance sampling, meaning that we use imported samples from an old policy. Unfortunately, this doesn't amount to good performance in practice, due to instability in the weight that is applied to the objective.

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) A^\theta(s_t, a_t)\right]$$

This is the gradient of the objective in Policy Gradients. Advantages are evaluated through Monte Carlo sampling in vanilla PG or a separate critic network, which fits the Value function in actor-critic methods, such as A2C.

When using importance sampling the network update becomes:

$$\begin{aligned}
\nabla_\theta J(\theta) &= E_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) A^\theta(s_t, a_t)\right] \\
&= E_{\tau \sim \pi_\theta}[f(\tau)] = \int \nabla_\theta \pi_\theta(\tau) f(\tau) d\tau \\
&= \int \nabla_\theta \frac{\pi_{\theta'}(\tau)}{\pi_{\theta'}(\tau)} \pi_\theta(\tau) f(\tau) d\tau \\
&= \int \nabla_\theta \pi_{\theta'}(\tau) \frac{\pi_\theta(\tau)}{\pi_{\theta'}(\tau)} f(\tau) d\tau \\
&= E_{\tau \sim \pi_{\theta'}}\left[\frac{\pi_\theta(\tau)}{\pi_{\theta'}(\tau)} f(\tau)\right] \\
&= E_{\tau \sim \pi_{\theta'}}\left[\frac{\pi_\theta(\tau)}{\pi_{\theta'}(\tau)} \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) A^\theta(s_t, a_t)\right]
\end{aligned}$$

$$\begin{aligned}
\frac{\pi_\theta(\tau)}{\pi_{\theta'}(\tau)} &= \frac{p(s_1)\prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)}{p(s_1)\prod_{t=1}^{T} \pi_{\theta'}(a_t|s_t) p(s_{t+1}|s_t, a_t)} \\
&= \frac{\prod_{t=1}^{T} \pi_\theta(a_t|s_t)}{\prod_{t=1}^{T} \pi_{\theta'}(a_t|s_t)}
\end{aligned}$$

The cumulative product of ratios is what causes this instability and what makes the use of importance sampling in this manner considerably arduous. PPO solves this by deriving an objective function which uses a single sample for evaluation of the importance weight.

Robustness

As described in Section 3, PPO uses a surrogate objective function, which aims to maximize the improvement to the old policy, while remaining in relatively close proximity in policy space. We now derive this objective function mathematically.

Let us first evaluate the difference in performance between two policies π and π'.

$$\begin{aligned}
J(\pi) &= E_{\tau\sim\pi}[r(\tau)] \\
\implies J(\pi') - J(\pi) &= E_{\tau\sim\pi'}[r(\tau)] - E_{\tau\sim\pi}[r(\tau)] \\
&= E_{\tau\sim\pi'}[r(\tau)] - V^{\pi}(s_0) = E_{\tau\sim\pi'}[r(\tau)] - E_{\tau\sim\pi'}[V^{\pi}(s_0)] \\
&= E_{\tau\sim\pi'}[r(\tau)] - E_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t V^{\pi}(s_t) - \sum_{t=1}^{\infty}\gamma^t V^{\pi}(s_t)\right] \\
&= E_{\tau\sim\pi'}[r(\tau)] + E_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^{t+1} V^{\pi}(s_{t+1}) - \sum_{t=0}^{\infty}\gamma^t V^{\pi}(s_t)\right] \\
&= E_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t\Big(r(s_t,a_t) + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\Big)\right] \\
&= E_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}(s_t,a_t)\right]
\end{aligned}$$

This is the so called Relative Policy Performance Identity and is the bedrock of the Monotonic Improvement Theory.

We define a discounted future state distribution $d^{\pi}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\pi)$, which gives the probability of ever being in a given state. Using this distribution, the relative performance of the two policies now is

$$\begin{aligned}
J(\pi') - J(\pi) &= E_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}(s_t,a_t)\right] \\
&= \frac{1}{1-\gamma}E_{a\sim\pi',s\sim d^{\pi'}}\left[A^{\pi}(s,a)\right] \\
&= \frac{1}{1-\gamma}E_{a\sim\pi,s\sim d^{\pi'}}\left[\frac{\pi'(a|s)}{\pi(a|s)}A^{\pi}(s,a)\right]
\end{aligned}$$

If π and π' are sufficiently close to each other in policy space, $d^{\pi'} \approx d^{\pi}$

$$\implies J(\pi') - J(\pi) \approx \frac{1}{1-\gamma}E_{a\sim\pi,s\sim d^{\pi}}\left[\frac{\pi'(a|s)}{\pi(a|s)}A^{\pi}(s,a)\right]$$

We denote the RHS with $\mathcal{L}_\pi(\pi')$ which we call a surrogate objective function and the goal of the algorithm now is

$$\max_{\theta'} \mathcal{L}_{\pi_\theta}(\pi_{\theta'}) = \max_{\theta'} E_{\tau \sim \pi_\theta}\Big[\sum_{t=0}^{\infty} \frac{\pi'_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} A_t^{\pi_\theta}(s_t, a_t)\Big]$$

However, this is limited by the condition for proximity of the two policies. One way to formalize this is to use the KL-divergence of the policies, performing constrained optimization or using it as part of the loss function. Notwithstanding their theoretical accuracy, these methods are computationally expensive and, therefore, a heuristic is employed - the ratios $\frac{\pi'(a|s)}{\pi(a|s)}$ are clipped between the values $1-\varepsilon$ and $1+\varepsilon$, thereby preventing the current policy to diverge too far from the old one. Furthermore, an extra minimization term is added, in order to additionally limit and discourage vast differences between policies.

$$\max_{\theta'} \mathcal{L}_{\pi_\theta}(\pi_{\theta'}) = \max_{\theta'} E_{\tau \sim \pi_\theta}\Big[\sum_{t=0}^{\infty} \min\big[\frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}\hat{A}_t^{\pi_\theta}, clip(\frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon)\hat{A}_t^{\pi_\theta}\big]\Big]$$

Appendix B

Architecture Details and Hyperparameters

The agent acts in an OpenAI Gym environment, taking actions and observing states and rewards at each timestep (policy rollout). The policy is represented by a neural network with the following structure:

- it receives as input a vector, representing the current state of the environment;
- this state vector subsequently passes through an affine transformation layer, so that the network can learn to contextually extract the most important information from the input
- the output of that linear layer is then inputted into the DNC (or LSTM)
- after that the output of DNC (LSTM) is passed to two different one-layer artificial neural networks with *tanh* activation functions, one of which is responsible for the value estimation, the other for the action distribution

Below are displayed the architecture and the hyperparameters used (Figure 1, Figure 2).

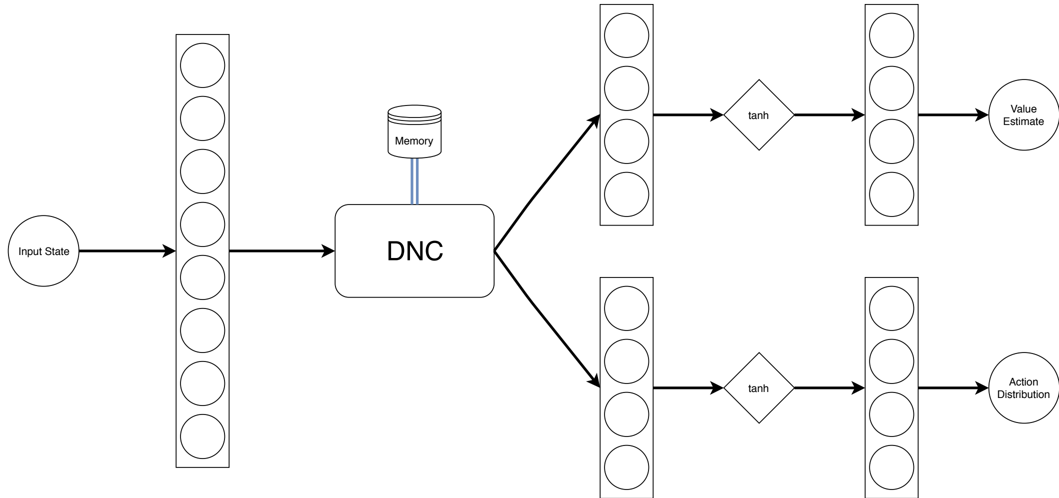


Figure 4: A graphic representation of the architecture.

Parameter	LSTM	DNC
learning rate	5e-3	64e-4
Adam betas	(0.9, 0.999)	(0.9, 0.999)
discount γ	0.98	0.98
GAE λ	0.97	0.97
epochs K	4	4
PPO clip ε	0.2	0.2
gradient clipping	5	5
affine layer size	8	8
network output size	32	16
action layer	16	8
value layer	16	8
row length W	-	4
number of rows N	-	16
controller layer size	-	20
number of read heads R	-	2

Figure 5: Hyperparameter settings