International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 8, Number 2, pages 408–433, July 2018

GUNREAL: GPU-accelerated UNsupervised REinforcement and Auxiliary Learning

Koichi Shirahata, Youri Coppens[1], Takuya Fukagai, Yasumoto Tomita, and Atsushi Ike
FUJITSU LABORATORIES LTD., 4-1-1, Kamikodanaka, Nakahara-ku
Kawasaki, Kanagawa, 211-8588, Japan

Received: February 1, 2018
Revised: April 27, 2018
Accepted: June 6, 2018
Communicated by Yasuaki Ito

Abstract

Recent state-of-the-art deep reinforcement learning algorithms, such as A3C and UNREAL, are designed to train on a single device with only CPU's. Using GPU acceleration for these algorithms results in low GPU utilization, which means the full performance of the GPU is not reached. Motivated by the architecture changes made by the GA3C algorithm, which gave A3C better GPU acceleration, together with the high learning efficiency of the UNREAL algorithm, this paper extends GA3C with the auxiliary tasks from UNREAL to create a deep reinforcement learning algorithm, GUNREAL, with higher learning efficiency and also benefiting from GPU acceleration. We show that our GUNREAL system achieves 3.8 to 9.5 times faster training speed compared to UNREAL and 73% more training efficiency compared to GA3C.

Keywords: Deep Reinforcement Learning, GPU, Auxiliary Tasks, Deep Learning

1 Introduction

Deep Learning considers the training of deep neural networks (DNNs), which are computational models capable of representing high-dimensional mathematical functions. These networks consist of many linked layers of non-linear operation units and parameters. Training a DNN generalizes the input-output relation of a dataset by updating the parameters (often called weights) via gradient descent, which has led to tremendous advances in image recognition. Deep Learning benefits from GPU acceleration, since the matrix computations of the DNN units can be easily parallelized over the many arithmetic cores of the GPU, resulting in a higher throughput for the DNN and hence reducing the training time.

Reinforcement learning is a Machine Learning methodology which teaches the action policies of agents through interaction with the environment in which they should operate [25]. Humans and other animals also need to learn how they can survive in their environment. Neuroscience provided evidence on the exitistence of neurons which respond to the predicted rewards and punishments [15–17, 23]. The responses of these neurons indicate that humans might also be using the strategy of reinforcement learning.

During the last years, Deep Learning enabled reinforcement learning to be applied on more advanced and complex environments such as robotics and HVAC (heating, ventilation, and air conditioning) control [28] for data centers. Training neural networks to represent the decision model in a reinforcement learning context used to be slow and unstable, but since the introduction of the

[1]He is currently a Ph.D. candidate at the Artificial Intelligence Laboratory of the Vrije Universiteit Brussel.

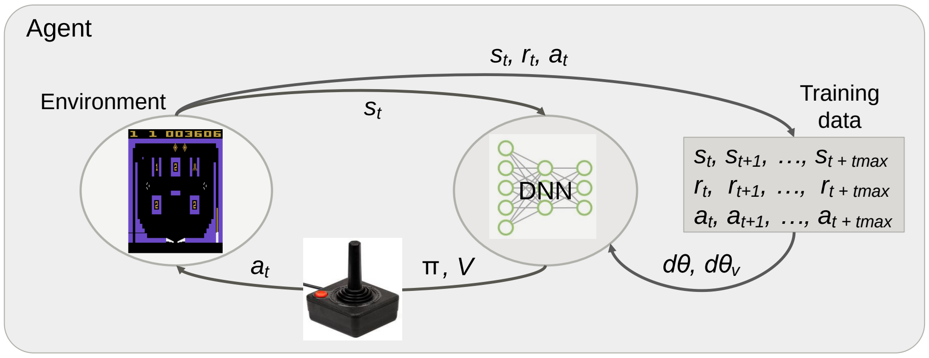

Agent
s_t, r_t, a_t
Environment
s_t
Training data
s_t, s_{t+1}, …, s_{t+tmax}
r_t, r_{t+1}, …, r_{t+tmax}
a_t, a_{t+1}, …, a_{t+tmax}
DNN
a_t
π, V
$d\theta$, $d\theta_v$

Figure 1: Deep reinforcement learning architecture (policy-based, model-free method). $d\theta$ is policy gradients and $d\theta_v$ is state-value gradients.

Deep Q-learning Network (DQN) algorithm [19], deep reinforcement learning has been advancing the state-of-the-art results. DQN stabilized the training procedure for DNNs by introducing an experience replay memory buffer in order to avoid divergence.

Latest state-of-the-art deep reinforcement learning algorithms focused on concurrent simulations, allowing to use multiple agents simultaneously to train a DNN faster. Asynchronous Advantage Actor-Critic (A3C) [18] has been proposed in 2016. The design of A3C targets single machines that only use CPU's. Therefore, when running the algorithm with GPU acceleration, A3C does not utilize the full performance of the GPU. The reason behind this is the lack of an experience replay memory buffer, resulting in small batches of data used for training the DNN. CPU-GPU communication is therefore a bottleneck which limits the GPU acceleration capabilities for A3C.

The next year, DeepMind extended the A3C algorithm with auxiliary learning tasks, which they named UNREAL [14]. The auxiliary task models share neural network layers with the main task model, so training these auxiliary tasks guides a part of the main model. The algorithm achieves higher learning efficiency, meaning that less actions have to be performed in total to learn the main task.

In order to reduce the CPU-GPU communication bottleneck of A3C, GA3C [2] was introduced in the same year as UNREAL, which re-designed the architecture to benefit better from GPU acceleration. GA3C sends larger batches of data at once to the GPU and has lightweight agents, which allows the system to increase the number of concurrent simulations on the CPU and hence increasing the GPU utilization and learning speed.

In this paper, we propose an extension to GA3C with the auxiliary tasks from UNREAL to achieve a learning efficient deep reinforcement learning algorithm, GUNREAL, that also benefits from GPU acceleration. We evaluated and optimized the throughput of our system by adding a memory constraining feature to the monitoring process and changing the data structure of our DNN input. compared GUNREAL with GA3C and an open-source replication of UNREAL to verify the learning performance over time and in terms of steps, the total number of performed actions by the agents. We used Atari games in OpenAI Gym [5] as a 2D simulator and 3D games in DeepMind Lab [4] as a 3D simulator for the experiments. The results show that GUNREAL gained higher learning efficiency, which allows the algorithm to learn complex tasks in less steps than GA3C. Next to this, GUNREAL also trains faster than UNREAL, hence proving that our algorithm gains both benefits of accelerated training time and learning more efficiently. Although GUNREAL is evaluated using games but GUNREAL is not designed for the specific application and the target applications are not limited to a specific application. GUNREAL is applicable to other applications with some modifications into the GUNREAL architecture. We discuss more about applicability of GUNREAL to other applications in section 5.

Our work makes the following major contributions:

- We propose a GPU-accelerated architecture of UNREAL, a state-of-the-art deep reinforcement learning algorithm, by extending GA3C for learning the auxiliary tasks.

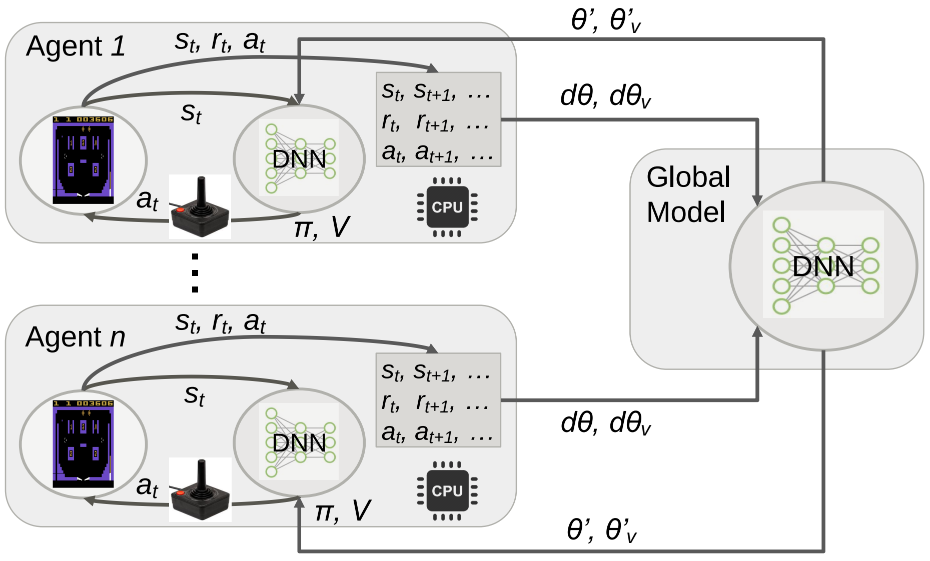


Figure 2: A3C architecture. θ' is updated policy parameters and θ'_v is updated state-value parameters.

- We evaluate the necessity of a preprocessing phase and improve the throughput on the GPU by changing the data format.
- We conduct experiments on both a 2D simulator (OpenAI Gym) and a 3D simulator (DeepMind Lab), and show our proposed architecture achieves 3.8 to 9.5 times faster training speed compared to UNREAL and 73% more training efficiency (fewer training steps) compared to GA3C.

2 Prior Reinforcement Learning Algorithms

Reinforcement learning provides a general framework for learning decision-making tasks. An agent has to operate in an environment and learn by trial-and-error. In each time step t, the agent observes the state of the environment s_t. Based on that observation, it has to decide which action a_t to perform. The decision model guiding the agent's action selection process is the policy π, which maps observations to a probability distribution over the set of available actions. After performing the selected action, the agent observes the next state of the environment s_{t+1}. Next to this, a feedback signal is received in the form of a reward r. This process is repeated until the agent ends up in a terminal state. After that, the environment is reset and the decision-making cycle is repeated. The agent's learning goal is to maximize the expected future reward.

Deep reinforcement learning architecture of a policy-based, model-free method is shown in Figure 1. Policy-based, model-free reinforcement learning methods learn by updating the parameters θ of a function approximation model to estimate the policy as a conditional probability distribution over the action set $\pi(a_t|s_t;\theta)$. Updating these parameters is done by applying gradient ascent on the expected future reward $\mathbb{E}[R_t]$, with $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$, representing the step-wise accumulated reward, discounted by a factor $\gamma \in (0, 1]$.

The policy gradient $\nabla_\theta \log \pi\left(a_t|s_t;\theta\right) R_t$ is usually estimated to update the parameters of the policy model. In order to reduce the variance, a baseline function is subtracted. A regularly used baseline is the state-value function $V^\pi(s_t) = \mathbb{E}[R_t|s_t]$, the expected future reward by starting from state s_t and following the policy π. Hence, the used gradient becomes $\nabla_\theta \log \pi\left(a_t|s_t;\theta\right)\left(R_t - V^\pi(s_t)\right)$. The usage of a value-based function in policy methods received the name *actor-critic methods*. Actor-critic methods have to estimate both the state-value function and the policy iteratively.

Algorithm 1 Asynchronous advantage actor-critic - pseudocode for each actor-learner agent thread [18].

```
 1: // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T \leftarrow 0$
 2: // Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
 3: Initialize thread step counter $t \leftarrow 1$
 4: repeat
 5:     Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
 6:     Synchronize thread-specific parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
 7:     $t_{start} \leftarrow t$
 8:     Get state $s_t$
 9:     repeat
10:         Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
11:         Receive reward $r_t$ and new state $s_{t+1}$
12:         $t \leftarrow t+1$
13:         $T \leftarrow T+1$
14:     until terminal $s_t$ or $t - t_{start} == t_{max}$
15:     $R \leftarrow \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t;\theta'_v) & \text{for non-terminal } s_t \text{ // Bootstrap from last state} \end{cases}$
16:     for $i \in \{t-1,\ldots,t_{start}\}$ do
17:         $R \leftarrow r_i + \gamma R$
18:         Accumulate gradients wrt $\theta'$ : $d\theta \leftarrow d\theta + \nabla_{\theta'}[\log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v)) + \beta H(\pi(s_t;\theta'))]$
19:         Accumulate gradients wrt $\theta'_v$ : $d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v}(R - V(s_i;\theta'_v))^2$
20:     end for
21:     Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$
22: until $T > T_{max}$
```

2.1 Asynchronous Advantage Actor-Critic (A3C)

A3C [18] is an actor-critic method which uses a DNN to estimate both the state-value function and the policy in one model. The A3C architecture is shown in Figure 2. The algorithm runs multiple agents concurrently to train a global DNN model. Each agent contains a local copy of the model to perform steps in its environment. Every t_{max} steps, or earlier when a terminal state is observed, the agent calculates gradients for policy $d\theta$ and gradients for state-value $d\theta_v$, based on the local model and sends these gradients asynchronously to the global DNN model. The global DNN model is updated using the received gradients. Afterwards, the agent synchronizes its local model parameters with the updated global network parameters θ' and θ'_v.

A3C uses two cost functions to update the policy and state-value function respectively with gradient ascent (for the policy) and gradient descent (for the value function). For the policy, we have $f_\pi(\theta) = \log\pi(a_t|s_t;\theta)(R_t - V(s_t;\theta_t)) + \beta H(\pi(s_t;\theta))$, where we notice an additional term representing the policy's entropy, regularized by a factor β. In this function, the return R_t is estimated by the bounded n-step return $R_t = \gamma^n V(s_{t+n};\theta_t) + \sum_{i=0}^{n-1}\gamma^i r_{t+i}$, where n is upper-bounded by t_{max}. For the value function, the loss is the square error of the n-step return: $f_v(\theta) = (R_t - V(s_t;\theta))^2$. The pseudocode for an actor-learner agent thread of A3C is described in Algorithm 1 [18].

The authors of A3C presented two DNN models. The first model, A3C-FF, is a feed-forward DNN which consists of two convolution layers followed by a fully connected layer which all contain a rectifier. From the latter layer, a Softmax layer approximates the policy and a linear layer (a fully connected layer without a rectifier) estimates the state-value function. The second model, A3C-LSTM, is similar to the first model with long short-term memory (LSTM) units of recurrent neural network (RNN). A LSTM unit contains a cell, in which values over arbitrary time intervals are memorized for learning tasks where time series information such as time lags and duration between events may help. This model also has two convolution layers followed by a fully connected layer, but afterwards an LSTM is inserted in the network. From this LSTM, a Softmax layer then approximates the policy and a linear layer estimates the state-value function.

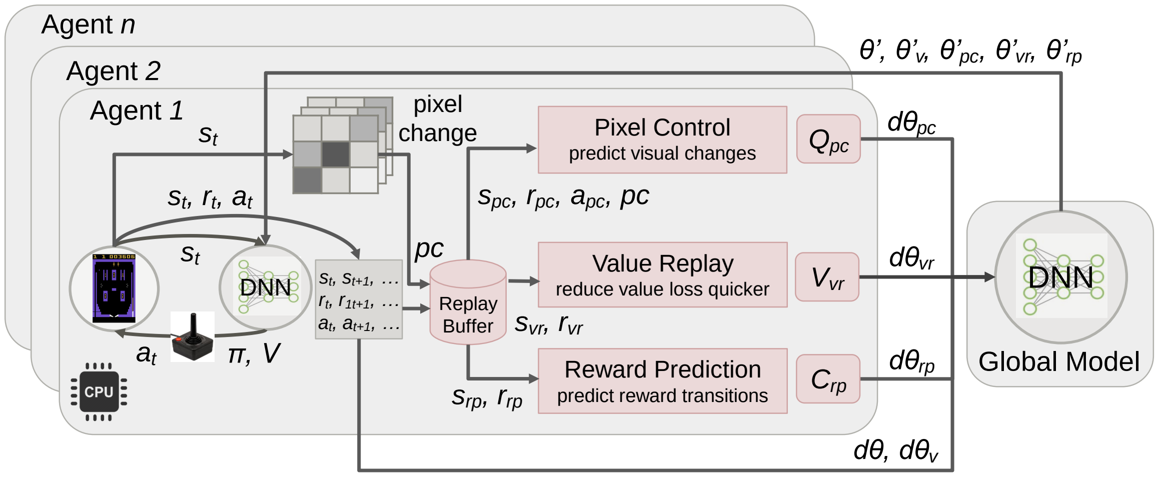


Figure 3: UNREAL architecture. Red blocks are the extensions to A3C with the auxiliary tasks. pc is pixel change, calculated every step.

2.2 UNsupervised REinforcement and Auxiliary Learning (UNREAL)

UNREAL [14] has extended A3C with three additional learning tasks, consisting of *pixel control* (PC), *value function replay* (VR), and *reward prediction* (RP), to increase the learning efficiency. The architecture of UNREAL is shown in Figure 3. The auxiliary tasks are trained by sampling frame sequences from a re-introduced experience replay memory buffer per agent which buffers most recent frames of observations (states), actions, and rewards for the latest 2,000 steps it performed. The UNREAL algorithm runs multiple agents concurrently in the same way as A3C. Once the replay buffer is generated, the algorithm runs A3C and each agent calculates gradients $d\theta$ and $d\theta_v$ for every t_{max} steps or earlier when a terminal state is observed.

Since DQN and successors mostly learn games by observing a screen, the pixel control task was introduced to learn the effect of actions on the environment by predicting average changes in the screen. Pixel change pc_t is generated every step for input of the pixel control task, by calculating the absolute difference per pixel for cropped 80 by 80 pixels and averaging each 4 by 4 pixels. When the gradients of A3C are calculated, the pixel control task is executed using $t_{max}+1$ sampled experience frame sequence. The last one frame in addition to the t_{max} frames is used for calculating last next state. Gradients of pixel control $d\theta_{pc}$ are calculated using rewards R_{pc} from the output of pixel control $max_a Q_{pc}(s_{t_{pc}}, a; \theta_{pc})$.

Value function replay updates the value function from the actor-critic model a second time to reduce the value loss faster. As the pixel control task, the value function replay task is executed when the gradients of A3C are calculated, using $t_{max}+1$ sampled experience frame sequence. The value function is calculated using the same DNN structure with A3C and gradients $d\theta_{vr}$ are calculated using rewards R_{vr} from the output V_{vr}.

The reward prediction task learns to predict from a sequence of observations whether the next step will result in a positive, negative or zero reward. As the other auxiliary tasks, the reward prediction task is executed when the gradients of A3C are calculated, using $t_{max}+1$ sampled experience frame sequence, but the frame sequence is selected in such a way that with 50% probability the value of reward at the end step is zero (or non-zero). The reward prediction task predicts classification of reward C_{rp} in the next step and calculate gradients $d\theta_{rp}$ based on the classification error.

After the gradients of A3C and the auxiliary tasks are calculated, the agent sends the gradients to the global DNN model and the global DNN model is updated using the received gradients. Afterwards, the agent synchronizes its local model parameters with the updated global network parameters. The pseudocode for an agent thread of UNREAL is described in Algorithm 2.

The pixel control and value function replay tasks require additional DNN models. These models re-use layers from the main actor-critic model and thereby guide the learning of the main task.

Algorithm 2 UNREAL - pseudocode for each actor-learner agent thread.

```
1: // Assume global shared parameter vectors $\theta$, $\theta_v$, $\theta_{pc}$, $\theta_{vr}$, $\theta_{rp}$ and global shared counter $T \leftarrow 0$
2: // Assume thread-specific parameter vectors $\theta'$, $\theta'_v$, $\theta'_{pc}$, $\theta'_{vr}$, $\theta'_{rp}$
3: // Assume experience replay buffer $RB$ with a user defined buffer size
4: Initialize thread step counter $t \leftarrow 1$
5: repeat
6:     if $RB$ is not full then
7:         Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
8:         Receive reward $r_t$ and new state $s_{t+1}$
9:         Calculate pixel change $pc_t$
10:        Add a frame $\{s_t, r_t, a_t, terminal\_or\_non\text{-}terminal, pc_t\}$ to $RB$
11:        continue
12:    end if
13:    Reset gradients: $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$, $d\theta_{pc} \leftarrow 0$, $d\theta_{vr} \leftarrow 0$, $d\theta_{rp} \leftarrow 0$
14:    Synchronize thread-specific parameters $\theta' \leftarrow \theta$, $\theta'_v \leftarrow \theta_v$, $\theta'_{pc} \leftarrow \theta_{pc}$, $\theta'_{vr} \leftarrow \theta_{vr}$, $\theta'_{rp} \leftarrow \theta_{rp}$
15:    Process A3C in Algorithm 1, line 7 - 20
16:    /* Process pixel control */
17:    Sample $t_{max}+1$ frame sequence from $RB_{t_{start\_pc}}$, $t_{start\_pc}$ is randomly selected
18:    $t_{pc} \leftarrow t_{start\_pc} + t_{max}$
19:    $R_{pc} \leftarrow \begin{cases} 0 & \text{for terminal } s_{t_{pc}-1} \\ max_a Q_{pc}(s_{t_{pc}}, a; \theta'_{pc}) & \text{for non-terminal } s_{t_{pc}-1} \text{ // Bootstrap from last state} \end{cases}$
20:    for $i \in t_{pc}-1, \ldots, t_{start\_pc}$ do
21:        $R_{pc} \leftarrow pc_i + \gamma_{pc} R_{pc}$
22:        Accumulate gradients wrt $\theta'_{pc} : d\theta_{pc} \leftarrow d\theta_{pc} + \nabla_{\theta'_{pc}}(R_{pc} - max_a Q_{pc}(s_i, a_i; \theta'_{pc}))^2$
23:    end for
24:    /* Process value function replay */
25:    Sample $t_{max}+1$ frame sequence from $RB_{t_{start\_vr}}$, $t_{start\_vr}$ is randomly selected
26:    $t_{vr} \leftarrow t_{start\_vr} + t_{max}$
27:    $R_{vr} \leftarrow \begin{cases} 0 & \text{for terminal } s_{t_{vr}-1} \\ V(s_{t_{vr}}, \theta'_v) & \text{for non-terminal } s_{t_{vr}-1} \text{ // Bootstrap from last state} \end{cases}$
28:    for $i \in t_{vr}-1, \ldots, t_{start\_vr}$ do
29:        $R_{vr} \leftarrow r_i + \gamma R_{vr}$
30:        Accumulate gradients wrt $\theta'_{vr} : d\theta_{vr} \leftarrow d\theta_{vr} + \nabla_{\theta'_{vr}}(R_{vr} - V(s_i; \theta'_{vr}))^2$
31:    end for
32:    /* Process reward prediction */
33:    Sample $t_{max\_rp}+1$ frame sequence from $RB_{t_{start\_rp}}$, with 50% probability sample zero reward
       (or non-zero reward) at the end step
34:    $t_{rp} \leftarrow t_{start\_rp} + t_{max\_rp}$
35:    $C_{rp} \leftarrow \begin{cases} 0 & \text{if } r_{t_{rp}} == 0 \\ 1 & \text{if } r_{t_{rp}} > 0 \quad \text{// } C_{rp} \text{ is the class index label in reward prediction} \\ -1 & \text{if } r_{t_{rp}} < 0 \end{cases}$
36:    for $i \in t_{rp}-1, \ldots, t_{start\_rp}$ do
37:        Accumulate gradients wrt $\theta'_{rp} : d\theta_{rp} \leftarrow d\theta_{rp} + \nabla_{\theta'_{rp}}(E(s_i, C_{rp}; \theta_{rp}))$ // $E$: classification
       error
38:    end for
39:    Perform asynchronous update of $\theta$ using $d\theta$, of $\theta_v$ using $d\theta_v$, of $\theta_{pc}$ using $d\theta_{pc}$, of $\theta_{vr}$ using
       $d\theta_{vr}$, and of $\theta_{rp}$ using $d\theta_{rp}$
40: until $T > T_{max}$
```

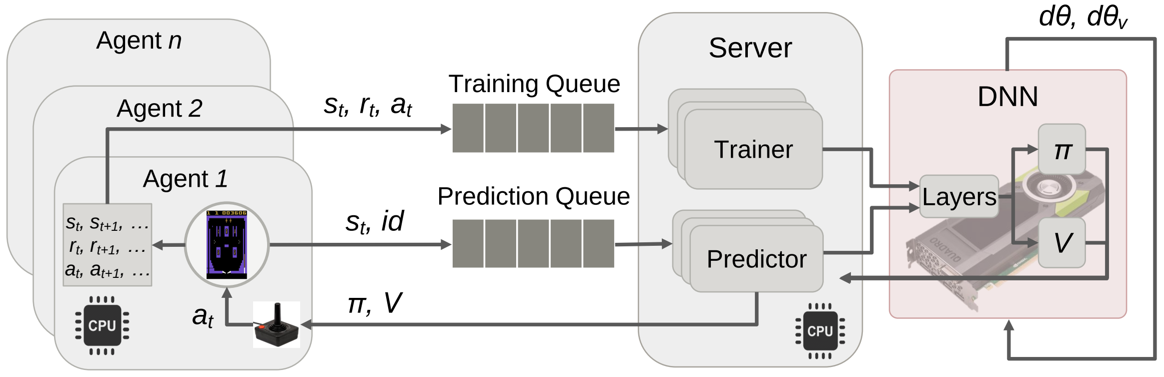


Figure 4: GA3C architecture. DNN computation is run on GPU. *id* is an agent id.

UNREAL's actor-critic model uses a slightly altered version of the A3C-LSTM model, which provides extra input to the LSTM. The DNNs are trained as a whole by reducing the weighted sum of losses for the main and auxiliary tasks.

Note that UNREAL is not designed for a specific task such as gaming though it is evaluated by 2D and 3D games in the paper. Especially the VR task and RP task are applicable to any applications which can work with the main A3C task. On the other hand, the PC task is specifically designed for processing image data, therefore PC is applicable only to applications using image data as input to a DNN of UNREAL.

2.3 GPU-accelerated Asynchronous Advantage Actor-Critic (GA3C)

GA3C changed the architecture of A3C to benefit better from GPU acceleration. The architecture is visualized in Figure 4. Unlike A3C, agents do not have a local model anymore. There is only a global DNN residing on the GPU. As actor-critic model, GA3C uses the feed-forward A3C model (A3C-FF). In order to decide on the action to be performed in the environment, agents must now consult the global DNN. Due to the lack of local models, agents do not calculate gradients anymore and simply send their experience to the global network so that the GPU can calculate gradients and update the network.

Communication with the GPU is achieved via a multi-producer-consumer architecture. Whenever agents have to step in the environment, they put pairs of their observation and the agent id on a queue, which is served by a set of predictor threads, managed by a server process. The predictors bundle observations into a large batch which is then sent to the GPU to calculate the network output for each observation. Afterwards, the predictors send the results back to the corresponding agents based on the received agent ids who can then decide on the action to apply in their environments. After at most t_{max} steps, the agent will have to send a batch of training data to update the network. These training batches with at most t_{max} samples, are put on a different queue, which is served by a set of trainer threads that is also managed by the server process. These threads bundle the received batches into a larger batch, which is then sent to the GPU to update the DNN. By using server threads to bundle data, the overall number of CPU-GPU round-trips is reduced while giving the GPU more data at once to process, which increases the GPU utilization and hence improves acceleration. The pseudocode of GA3C is described in Algorithm 3 for an actor agent thread, in Algorithm 4 for a predictor thread, and in Algorithm 5 for a trainer thread respectively.

It must be noted that GA3C suffers from a phenomenon called *policy lag*, which makes the learning procedure of GA3C noisier than A3C. This can be reduced by increasing the training batch size sent to the GPU, but doing so might decrease the speed with which learning progresses. A batch size of 40 was according to GA3C's authors an optimal value [11].

Algorithm 3 GA3C - pseudocode for each actor agent thread.

```
1: // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T \leftarrow 0$
2: Initialize thread step counter $t \leftarrow 1$
3: repeat
4:     $t_{start} \leftarrow t$
5:     Get state $s_t$
6:     repeat
7:         Put $\{agent\_id, s_t\}$ to prediction_queue
8:         Wait $\{\pi(a_t|s_t;\theta), V(s_t;\theta_v)\}$ from predictor_thread for the prediction to come back
9:         Perform $a_t$ according to policy $\pi(a_t|s_t;\theta)$
10:        Receive reward $r_t$ and new state $s_{t+1}$
11:        $t \leftarrow t + 1$
12:        $T \leftarrow T + 1$
13:    until terminal $s_t$ or $t - t_{start} == t_{max}$
14:    $R \leftarrow \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t;\theta_v) & \text{for non-terminal } s_t \text{ // Bootstrap from last state} \end{cases}$
15:    for $i \in \{t-1, \ldots, t_{start}\}$ do
16:        $R \leftarrow r_i + \gamma R$
17:    end for
18:    Put frames $\{s, a, R\}$ to training_queue
19: until $T > T_{max}$
```

Algorithm 4 GA3C - pseudocode for each predictor thread.

```
1: // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T \leftarrow 0$
2: // Assume thread-specific constant prediction_batch_size
3: repeat
4:     batch_size $\leftarrow 0$
5:     repeat
6:         Get frames $\{agent\_id, s_{t_{agent\_id}}\}$ from prediction_queue
7:         batch_size $\leftarrow$ batch_size + the_number_of_frames
8:     until batch_size $\geq$ prediction_batch_size or prediction_queue is empty
9:     Calculate $\pi(a_t|s_t;\theta)$ and $V(s_t;\theta_v)$ // on GPU
10:    for $i$ in $0, \ldots,$ batch_size $- 1$ do
11:        Send $\{\pi(a_i|s_i;\theta), V(s_i;\theta_v)\}$ to $agent\_thread_{agent\_id_i}$
12:    end for
13: until $T > T_{max}$
```

Algorithm 5 GA3C - pseudocode for each trainer thread.

```
1: // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T \leftarrow 0$
2: // Assume thread-specific constant training_min_batch_size
3: repeat
4:     Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
5:     batch_size $\leftarrow 0$
6:     repeat
7:         Get frames $\{s, a, R\}$ from training_queue
8:         batch_size $\leftarrow$ batch_size + the_number_of_frames
9:     until batch_size $>$ training_min_batch_size
10:    Calculate gradients wrt $\theta : d\theta \leftarrow \nabla_\theta[\log \pi(a|s;\theta)(R - V(s;\theta_v)) + \beta H(\pi(s;\theta))]$ // on GPU
11:    Calculate gradients wrt $\theta_v : d\theta_v \leftarrow \nabla_{\theta_v}(R - V(s;\theta_v))^2$ // on GPU
12:    Perform update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$ // on GPU
13: until $T > T_{max}$
```

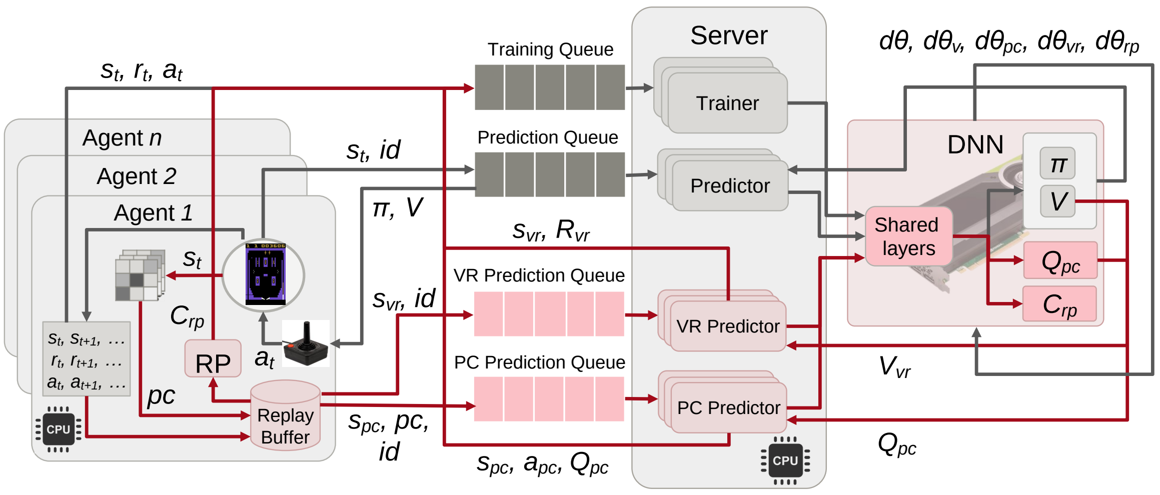


Figure 5: GUNREAL architecture. Red arrows and blocks are the extensions to GA3C with the auxiliary tasks from UNREAL. *id* is an agent id.

3 GUNREAL architecture

In an attempt to achieve higher learning efficiency for GA3C, we propose GUNREAL (GPU-accelerated UNREAL) [8], an extension to GA3C with the auxiliary tasks from UNREAL. We implemented the auxiliary tasks from UNREAL into the GA3C architecture. The resulting extension is shown in Figure 5. Agents have a replay memory buffer to perform experience replay to train the auxiliary tasks. While stepping through the environment, the agent will save the observed state together with the performed action and the received reward in the buffer. When the agent has performed at most $t_{max} = 20$ steps, the GA3C training batch will be generated. After that, the training batches for the auxiliary tasks will also be sampled from the replay memory buffer.

The entire training batch for the reward prediction task can be generated based on the information available in the memory buffer, since the training output for this task is simply the sign of received rewards. However, pixel control task and the value function replay require a n-step return to be estimated for the training output. To do so, the output of the DNN model linked to the respective task has to be consulted. For the PC task, we need the output of the PC DNN model and for the VR task, we need the state-value outputted by the main actor-critic model. To enable consulting these respective network outputs at the requested time, two extra predictions lines are inserted into the communication architecture. For each of the two tasks, a queue is added to the system together with their respective set of server threads, which we call PC and VR predictors. These threads serve the requests put on the queues by several agents and bundle them into batches that will be sent to the GPU to calculate the respective network outputs. Afterwards, the threads will send the results back to the corresponding agent. When all the training batches are generated, the agent puts them all on the training queue in order to update GUNREAL's DNN as a whole. The pseudocode of GUNREAL training is described in Algorithm 6 for an actor agent thread, in Algorithm 7 for a PC predictor thread, in Algorithm 8 for a VR predictor thread, and in Algorithm 9 for a trainer thread respectively. The pseudocode for a base A3C predictor thread is described in Algorithm 4, the same as the pseudocode for a predictor thread in GA3C. A PC predictor thread and a VR predictor thread perform in the same way as the base A3C predictor thread, except parameters θ_{pc} and θ_{vr} are used instead of θ, and functions $max_a Q_{pc}(s_{t_{pc}}, a; \theta_{pc})$ and $V(s_{t_{vr}}; \theta_v)$ are used instead of $\pi(a_t|s_t; \theta)$ and $V(s_t; \theta_v)$.

GUNREAL extends GA3C's DNN model with auxiliary layers in a similar fashion as was done in UNREAL. The DNN structure of GUNREAL is summarized in Figure 6. The main actor-critic model is the same as in GA3C, namely the A3C-FF model. For the pixel control task, we re-use the two convolution layers and the first fully connected (FC) layer of this main model. Another FC layer follows with a split to two deconvolution layers, which are afterwards recombined according to the principle of the dueling network architecture for DQN [27]. The pixel control task is trained via n-step Q-learning [18]. The output of this network represents a 20 by 20 grid for each action. The cells represent changes in density of that particular region of the input state, which would be caused when the action would be performed. Because of this, the generation of training data requires the agent to save the changes caused by its performed actions in the replay memory buffer while stepping through the environment. The reward prediction network only re-uses the convolution layers of the main model and adds two fully connected layers and a Softmax operation to output a classification of 3 sequential states from the replay memory buffer that predicts whether the state following this sequence will provide the agent positive, negative or zero reward. Since value function replay simply re-uses the state-value from the main actor-critic model, no extra additions to the DNN are needed to perform this auxiliary task.

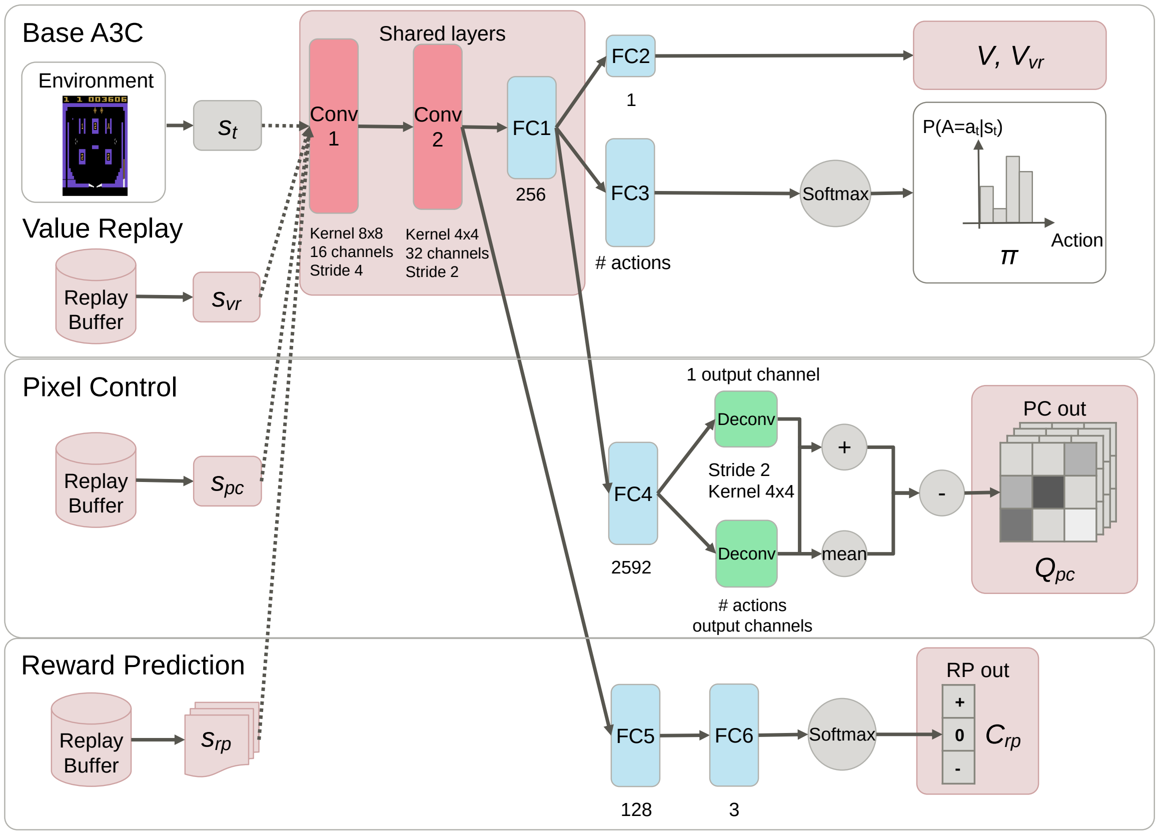


Figure 6: GUNREAL DNN structure

3.1 Preprocessing the environment observations

Originally, (G)A3C has a preprocessing phase, collecting four sequential 84 by 84 gray-scale frames. The result is then used as a single environment observation for the learning process, so that the observations contain extra information such as motion, which makes learning faster. On the other hand, preprocessing the environment observations was omitted in UNREAL and single RGB frames were used instead. A clear reason for this omission was not mentioned in the UNREAL paper, but is likely due to the usage of an LSTM in the DNN. GUNREAL's DNN does not contain an LSTM, since it is an extension of GA3C. We tested GUNREAL with and without preprocessing the environment observations. Figure 7 shows a comparison of learning curves of GUNREAL on the DeepMind Lab game, *nav_maze_static_01*, when single RGB frames or preprocessed observations

Algorithm 6 GUNREAL - pseudocode for each actor agent thread.

```
// Assume global shared parameter vectors $\theta$, $\theta_v$, $\theta_{pc}$, $\theta_{vr}$, $\theta_{rp}$ and global shared counter $T \leftarrow 0$
// Assume experience replay buffer RB with a user defined buffer size
Initialize thread step counter $t \leftarrow 1$
repeat
    /* Process base A3C */
    $t_{start} \leftarrow t$
    Get state $s_t$
    repeat
        Put $\{agent\_id, s_t\}$ to prediction_queue
        Wait $\{\pi(a_t|s_t;\theta), V(s_t;\theta_v)\}$ from predictor_thread for the prediction to come back
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta)$
        Receive reward $r_t$ and new state $s_{t+1}$
        Calculate pixel change $pc_t$
        Add a frame $\{s_t, r_t, a_t, terminal\_or\_non\text{-}terminal, pc_t\}$ to RB
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R \leftarrow \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t;\theta_v) & \text{for non-terminal } s_t \text{ // Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
    end for
    if RB is full then
        /* Process pixel control */
        Sample $t_{max} + 1$ frame sequence from $RB_{t_{start\_pc}}$, $t_{start\_pc}$ is randomly selected
        $t_{pc} \leftarrow t_{start\_pc} + t_{max}$
        Put $\{agent\_id, s_t\}$ to pc_prediction_queue
        Wait $max_a Q_{pc}(s_{t_{pc}}, a; \theta_{pc})$ from pc_predictor_thread for the prediction to come back
        $R_{pc} \leftarrow max_a Q_{pc}(s_{t_{pc}}, a; \theta_{pc})$
        for $i \in t_{pc} - 1, \ldots, t_{start\_pc}$ do
            $R_{pc} \leftarrow pc_i + \gamma_{pc} R_{pc}$
        end for
        /* Process value function replay */
        Sample $t_{max} + 1$ frame sequence from $RB_{t_{start\_vr}}$, $t_{start\_vr}$ is randomly selected
        $t_{vr} \leftarrow t_{start\_vr} + t_{max}$
        Put $\{agent\_id, s_t\}$ to vr_prediction_queue
        Wait $V(s_{t_{vr}}, \theta_v)$ from vr_predictor_thread for the prediction to come back
        $R_{vr} \leftarrow V(s_{t_{vr}}, \theta_v)$
        for $i \in t_{vr} - 1, \ldots, t_{start\_vr}$ do
            $R_{vr} \leftarrow r_i + \gamma R_{vr}$
        end for
        /* Process reward prediction */
        Sample $t_{max\_rp} + 1$ frame sequence from $RB_{t_{start\_rp}}$, with 50% probability sample zero
            reward (or non-zero reward) at the end step
        $t_{rp} \leftarrow t_{start\_rp} + t_{max\_rp}$
        $C_{rp} \leftarrow \begin{cases} 0 & \text{if } r_{t_{rp}} == 0 \\ 1 & \text{if } r_{t_{rp}} > 0 \quad \text{// } C_{rp} \text{ is the class index label in reward prediction} \\ -1 & \text{if } r_{t_{rp}} < 0 \end{cases}$
    end if
    Put frames $\{s, a, R\}$ of base A3C, and if exist $\{s, a, R_{pc}\}$ of pixel control, $\{s, R_{vr}\}$ of value
        function replay, and $\{s, C_{rp}\}$ of reward prediction, to training_queue
until $T > T_{max}$
```

Algorithm 7 GUNREAL - pseudocode for each pixel control predictor thread.

```
1: // Assume global shared parameter vectors $\theta_{pc}$ and global shared counter $T \leftarrow 0$
2: // Assume thread-specific constant prediction_batch_size
3: repeat
4:     $batch\_size \leftarrow 0$
5:     repeat
6:         Get frames $\{agent\_id, s_{t_{agent\_id}}\}$ from prediction_queue
7:         $batch\_size \leftarrow batch\_size + the\_number\_of\_frames$
8:     until $batch\_size \geq prediction\_batch\_size$ or pc_prediction_queue is empty
9:     Calculate $max_a Q_{pc}(s_{t_{pc}}, a; \theta_{pc})$ // on GPU
10:    for $i$ in $0, \ldots, batch\_size - 1$ do
11:        Send $max_a Q_{pc}(s_{t_{pc}}, a; \theta_{pc})$ to $agent\_thread_{agent\_id_i}$
12:    end for
13: until $T > T_{max}$
```

Algorithm 8 GUNREAL - pseudocode for each value function replay predictor thread.

```
1: // Assume global shared parameter vectors $\theta_{vr}$ and global shared counter $T \leftarrow 0$
2: // Assume thread-specific constant prediction_batch_size
3: repeat
4:     $batch\_size \leftarrow 0$
5:     repeat
6:         Get frames $\{agent\_id, s_{t_{agent\_id}}\}$ from prediction_queue
7:         $batch\_size \leftarrow batch\_size + the\_number\_of\_frames$
8:     until $batch\_size \geq prediction\_batch\_size$ or vr_prediction_queue is empty
9:     Calculate $V(s_{t_{vr}}; \theta_v)$ // on GPU
10:    for $i$ in $0, \ldots, batch\_size - 1$ do
11:        Send $V(s_{t_{vr}}; \theta_v)$ to $agent\_thread_{agent\_id_i}$
12:    end for
13: until $T > T_{max}$
```

Algorithm 9 GUNREAL - pseudocode for each trainer thread.

```
1: // Assume global shared parameter vectors $\theta$, $\theta_v$, $\theta_{pc}$, $\theta_{vr}$, $\theta_{rp}$ and global shared counter $T \leftarrow 0$
2: // Assume thread-specific constant training_batch_size
3: repeat
4:     Reset gradients: $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$, $d\theta_{pc} \leftarrow 0$, $d\theta_{vr} \leftarrow 0$, $d\theta_{rp} \leftarrow 0$
5:     $batch\_size \leftarrow 0$
6:     repeat
7:         Get frames$\{s, a, R\}$ of base A3C, and if exist $\{s, a, R_{pc}\}$ of pixel control, $\{s, R_{vr}\}$ of value function replay, and $\{s, C_{rp}\}$ of reward prediction, from training_queue
8:         $batch\_size \leftarrow batch\_size +$ the_number_of_frames
9:     until $batch\_size \geq training\_batch\_size$
10:    Calculate gradients wrt $\theta : d\theta \leftarrow \nabla_\theta[\log \pi(a|s;\theta)(R - V(s;\theta_v)) + \beta H(\pi(s;\theta))]$ // on GPU
11:    Calculate gradients wrt $\theta_v : d\theta_v \leftarrow \nabla_{\theta_v}(R - V(s;\theta_v))^2$ // on GPU
12:    if frames of auxiliary tasks exist then
13:        Calculate gradients wrt $\theta_{pc} : d\theta_{pc} \leftarrow \nabla_{\theta_{pc}}(R_{pc} - max_a Q_{pc}(s, a; \theta_{pc}))^2$ // on GPU
14:        Calculate gradients wrt $\theta_{vr} : d\theta_{vr} \leftarrow \nabla_{\theta_{vr}}(R_{vr} - V(s;\theta_{vr}))^2$ // on GPU
15:        Calculate gradients wrt $\theta_{rp} : d\theta_{rp} \leftarrow \nabla_{\theta_{rp}}(E(s, C_{rp}; \theta_{rp}))$ // on GPU, $E$: classification error
16:    end if
17:    Perform update of $\theta$ using $d\theta$, of $\theta_v$ using $d\theta_v$, of $\theta_{pc}$ using $d\theta_{pc}$, of $\theta_{vr}$ using $d\theta_{vr}$, and of $\theta_{rp}$ using $d\theta_{rp}$ // on GPU
18: until $T > T_{max}$
```

were used as network input. It is clear that the preprocessed observations result in quicker learning for GUNREAL, as the state observation from single RGB frames did not give the desired learning abilities for GUNREAL. Therefore, we decided to keep the original preprocessing phase from GA3C in GUNREAL.

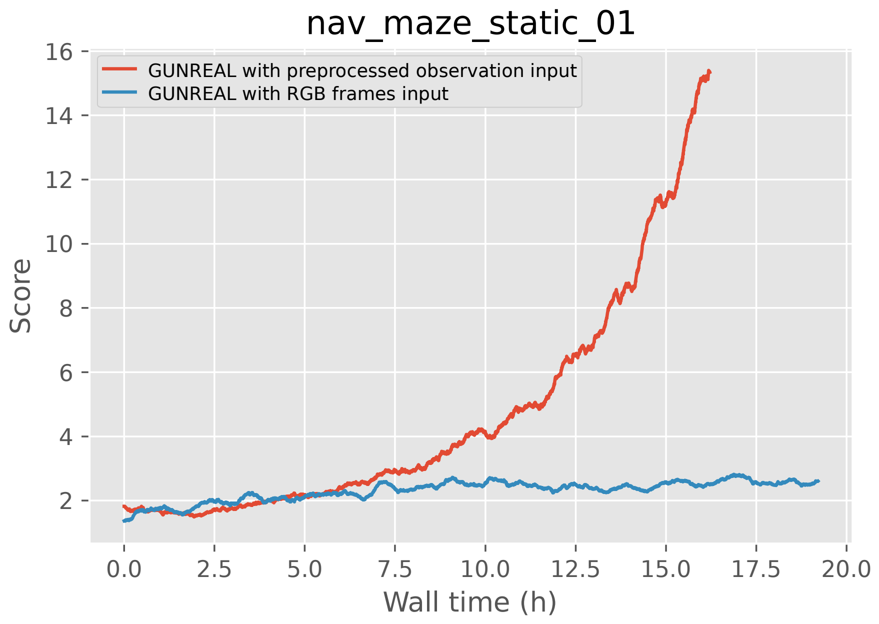


Figure 7: Training results of GUNREAL using different network input

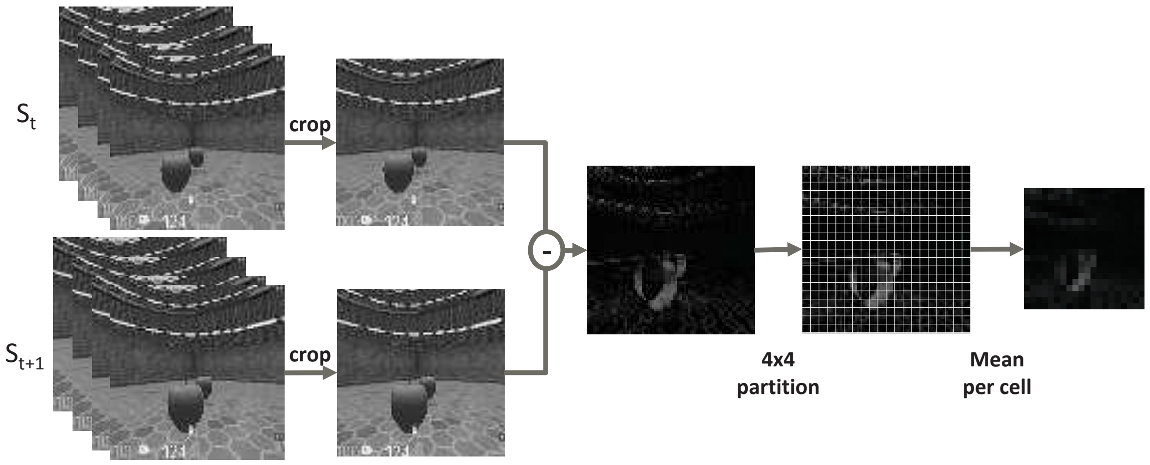


Figure 8: Pixel control training output calculation process for GUNREAL

Preprocessing the environment's observations affects the calculation process needed to save pixel changes for the pixel control task. In UNREAL, where the agent observes RGB frames, this was done by taking the absolute difference from the 80 by 80 center crop of two 84 by 84 RGB frames, for which the average is then calculated over the channel axis, resulting in a gray frame containing the differences in density per pixel. Then, a partition of the frame was made into groups of 4 by 4 adjacent pixels. From this partition, the average density per group is then outputted, resulting in a 20 by 20 grid. Since the agent's observations in GUNREAL consist of four gray-scale frames, calculating the pixel changes between two observations is determined by taking the absolute difference per pixel from the final frame in both states. After this, a partition is made by grouping 4 by 4 adjacent pixels and the average value per group is outputted, resulting in a 20 by 20 grid summarizing the visual changes between the two states. This process is illustrated in Figure 8.

3.2 Dynamics

Similar to GA3C, GUNREAL possesses the ability to alter the number of running agents and server threads dynamically at runtime. The system monitors the achieved predictions per second by the GPU and alters the number of the components in regular time intervals in order to reach a maximal throughput. Since the agents of GUNREAL use a replay memory buffer, we decided to put an upper limit to the number of components that can be created by monitoring the overall memory usage of the system. Whenever the device reaches a RAM consumption higher than 95%, the number of agents and server threads currently present are set as an upper limit.

4 Experiments

We evaluate the performance of our GUNREAL algorithm. Similar to GA3C, GUNREAL has been developed using the Deep Learning framework TensorFlow [1]. In order to achieve better throughput on the GPU, we investigated the tensor structure in the DNN. GUNREAL has been tested on games from the Atari environment of OpenAI Gym [6] and the more complex 3D simulator, DeepMind Lab [3]. To assess the learning performance, we provide learning curves, showing the achieved game scores by the algorithm in a step scale, in order to present the learning efficiency, and a runtime scale to demonstrate the real-time learning speed. We compare GUNREAL's performance with GA3C, which the authors have made publicly available[2], and also compare with an open-source replication of the UNREAL algorithm[3]. For the experiments, UNREAL has been enabled with GPU usage for DNN computations using TensorFlow.

The system environment configuration is shown in Table 1. For the OpenAI Gym experiments, we used Python 3.6.1 as interpreter, while the DeepMind Lab experiments have been run with Python 2.7, due to compatibility issues with Python 3. The parameter configuration of the algorithms is shown in Table 2. We opted to use similar values for the parameters present in multiple algorithms so that we focus on the main differences in architectural designs between the algorithms.

For GA3C and GUNREAL, we start with 15 agents and 5 sever threads of each kind. Both algorithms use dynamic monitoring to converge to an optimal throughput. UNREAL uses the total number of CPU threads as number of agents. For both UNREAL and GUNREAL, the pixel control auxiliary task uses a separate discount factor of 0.9.

Table 1: System environment configuration

Environment	OpenAI Gym 0.8.0	DeepMind Lab
Processor	Intel Xeon E5-1650v3 (12 threads, 3.50 GHz)	Intel Xeon E3-1245v5 (8 threads, 3.50 GHz)
GPU (NVIDIA)	GeForce GTX Titan X	Quadro M4000
Python Interpreter	CPython 3.6.1	CPython 2.7.13
CUDA Version	CUDA 8 + cuDNN 6	CUDA 8 + cuDNN 5.1
Framework	TensorFlow 1.2	

4.1 Tensor structure

In general, two data formats are used to process image batches, NHWC and NCHW. The first dimension (N) represents the number of images in the batch. C represents the channel values of

[2]`https://github.com/NVlabs/GA3C`
[3]`https://github.com/miyosuda/unreal`

Table 2: Parameter configuration of the learning algorithms

Parameter	**UNREAL**	**GA3C**	**GUNREAL**
Learning rate (α)	0.0003		
Entropy rate (β)	0.01		
Discount factor (γ)	0.99		
t_{max}	20		
Training batch size	***N.A.***	40	
Replay buffer size	2000 steps	***N.A.***	2000 steps
Pixel control discount	0.9	***N.A.***	0.9

the images and the other two remaining dimensions respectively represent the size of the images in terms of their height (H) and width (W). By default, TensorFlow processes image data structured as NHWC tensors to optimize for CPU computations. However, it is known that NCHW tensor structures are more suitable for GPU acceleration because of the usage of the cuDNN libraries. We compared the default NHWC tensors with the NCHW tensors by running GUNREAL on GPU to train the Atari game Pong. Comparing the two runs, we find that using NCHW tensors increases the number of predictions and trainings per second that could be processed by the GPU. As the dynamics of GUNREAL converge to the optimal throughput, it can be seen on Figure 9a that we reach about 1550 predictions per second (PPS) when using NCHW tensors, whilst only reaching 1477 PPS when NHWC tensors were used. This change in tensor format resulted in an increase of 5% in PPS. Correlated, the number of trainings per seconds (TPS) that could be reached by GUNREAL increased from 36 to 38 when using NCHW tensors, which is shown in Figure 9b.

Based on the increase in speed using NCHW tensors, we decided to standardly use NCHW tensors in GUNREAL. GA3C's DNN is originally configured to process the default NHWC tensor format and we did not alter this format for GA3C in the results of the following sections.

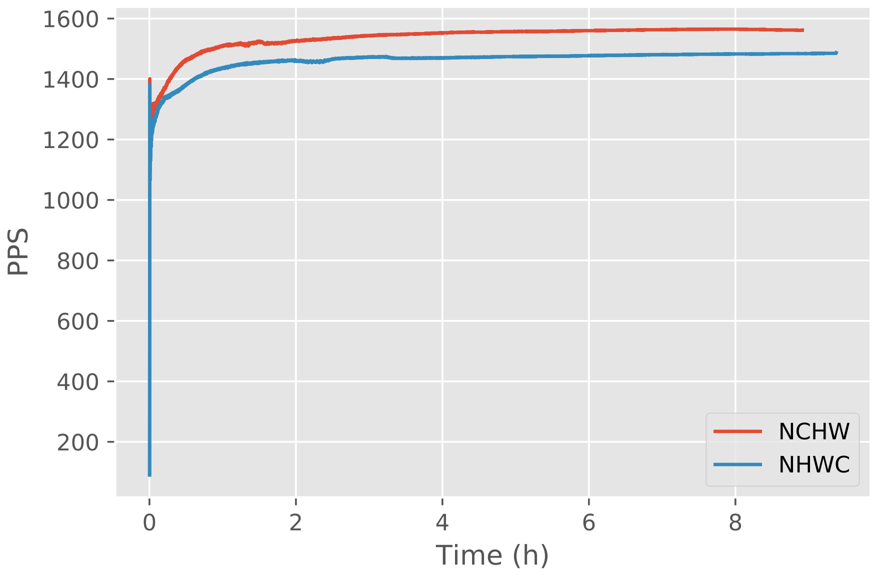
4.2 OpenAI Gym training results

We ran GUNREAL on three games from the Atari environment of OpenAI Gym and used the *Deterministic-v3* configuration of each game. The results are shown in Figure 10 for *Breakout*, in Figure 11 for *Pong*, and in Figure 12 for *Space Invaders*. In terms of steps, GUNREAL reached higher scores in fewer steps compared to GA3C: for *Breakout*, a score of 150 was reached in 6.7% fewer steps, and for *Space Invaders*, a score of 550 was reached in 26.0% fewer steps. In terms of runtime, GUNREAL performed similar to GA3C, and finished training faster than UNREAL in all cases: 3.8 times faster for *Breakout*, 9.5 times faster for *Pong*, and 4.0 times faster for *Space Invaders*.

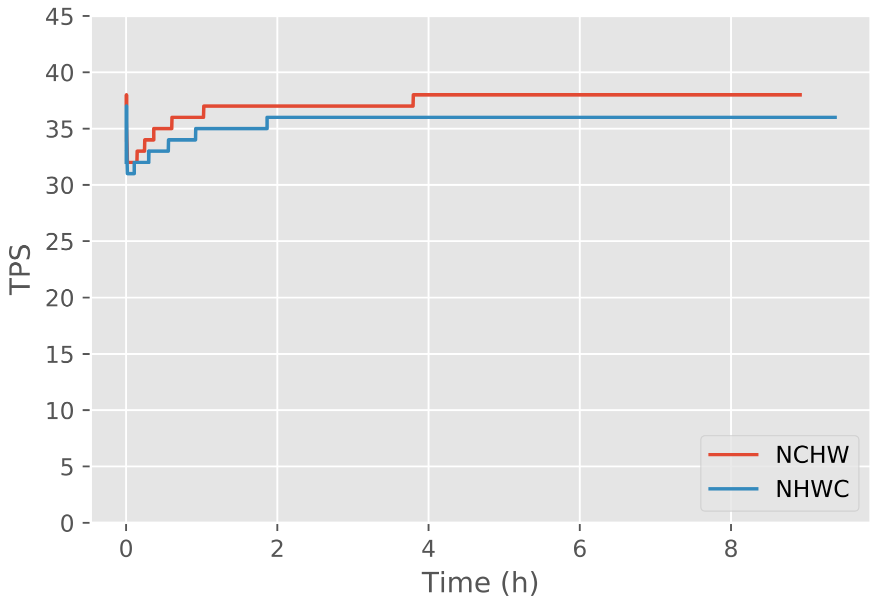
Discussing the runtime of the algorithm, GA3C lacks auxiliary tasks, giving it less computation per step and less data communication over its training queue than GUNREAL does. On the other hand, the learning efficiency introduced by the auxiliary tasks in GUNREAL needs less steps and correlated less time to reach higher scores than GA3C. As a result, GA3C and GUNREAL show similar runtime performance on simple applications such as Pong and Breakout.

4.3 DeepMind Lab training results

We also verified GUNREAL's performance on DeepMind Lab, for which is known that UNREAL performed better than A3C. Two games of DeepMind Lab were tested, namely *nav_maze_static_01*, where the agent has to navigate through a maze with a static layout towards a goal, and *seekavoid_arena_01*, where the task is to collect melons while avoiding collecting lemons. We defined 6 discrete actions for each game, being the four walking directions (forward, backward, left and right) and two pivoting actions to turn the agent around (left and right). We again show learning curves of GUNREAL, GA3C and UNREAL in terms of steps and runtime in Figure 13 for *nav_maze_static_01*, and in

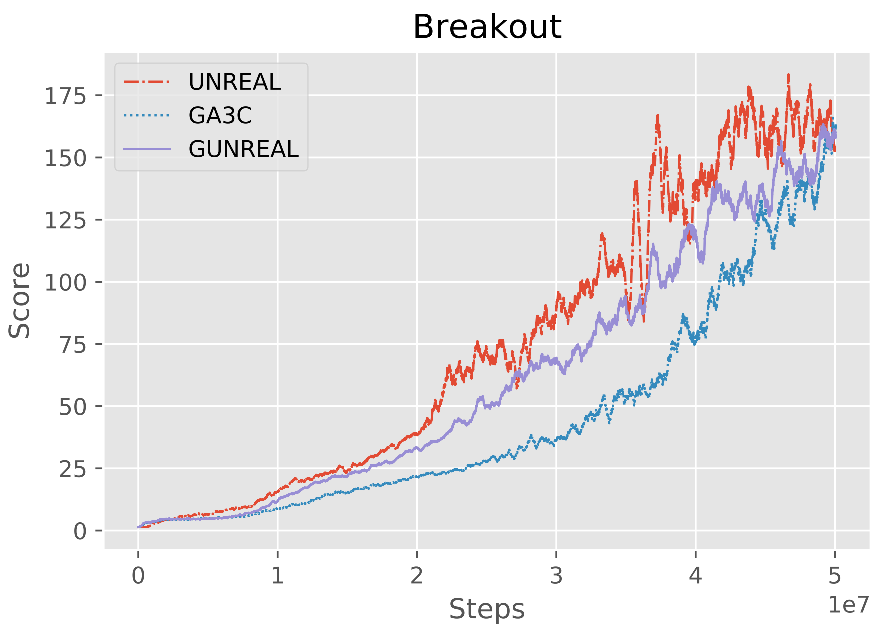


(a) Predictions per second (PPS)

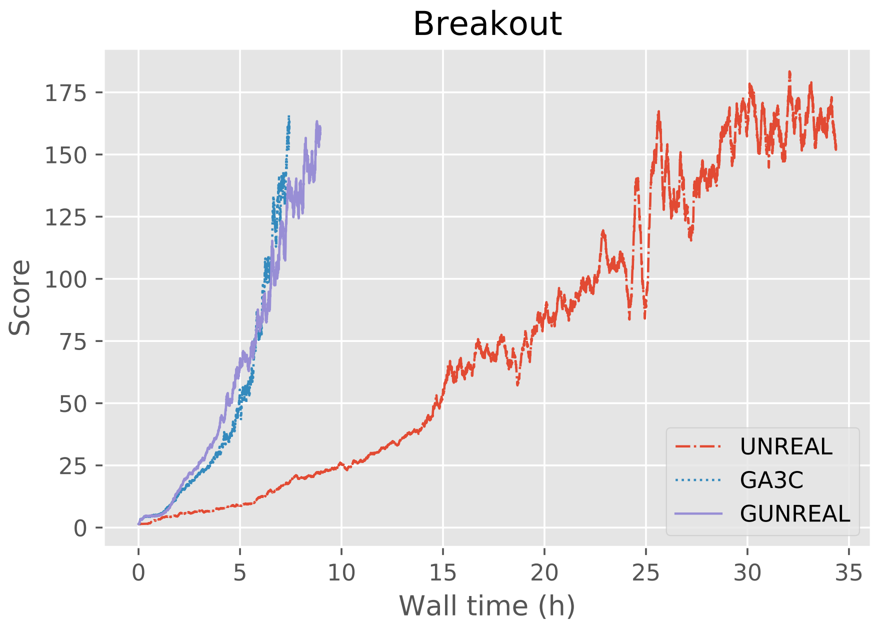


(b) Trainings per second (TPS)

Figure 9: Throughput comparison of GUNREAL using different tensor structures

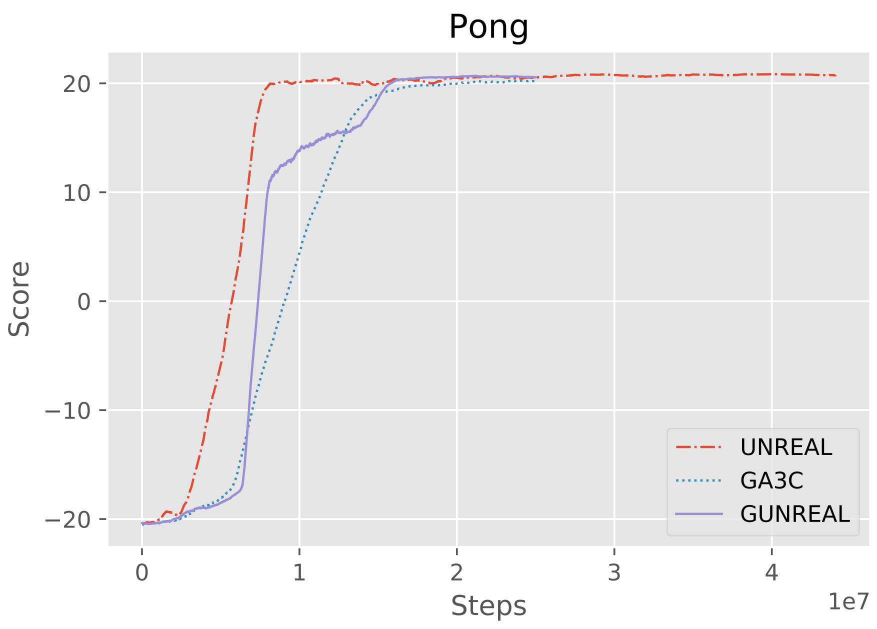


(a) Learning over steps

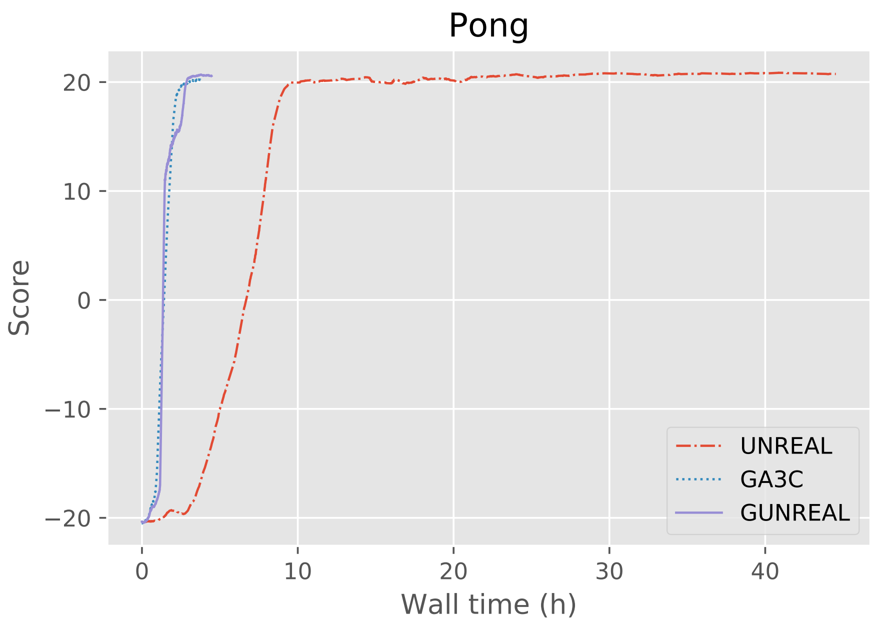


(b) Learning over time

Figure 10: Training results on OpenAI Gym (Breakout)

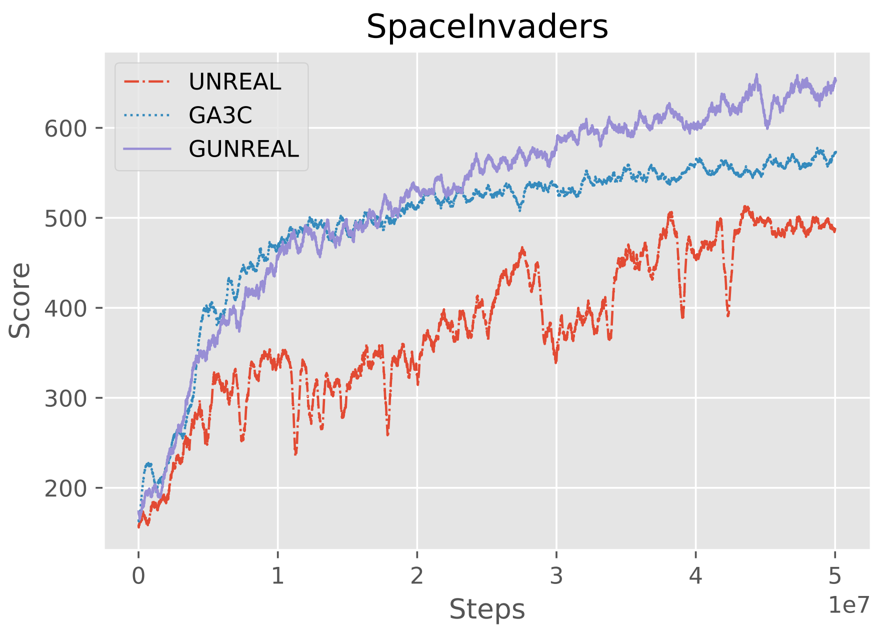


(a) Learning over steps

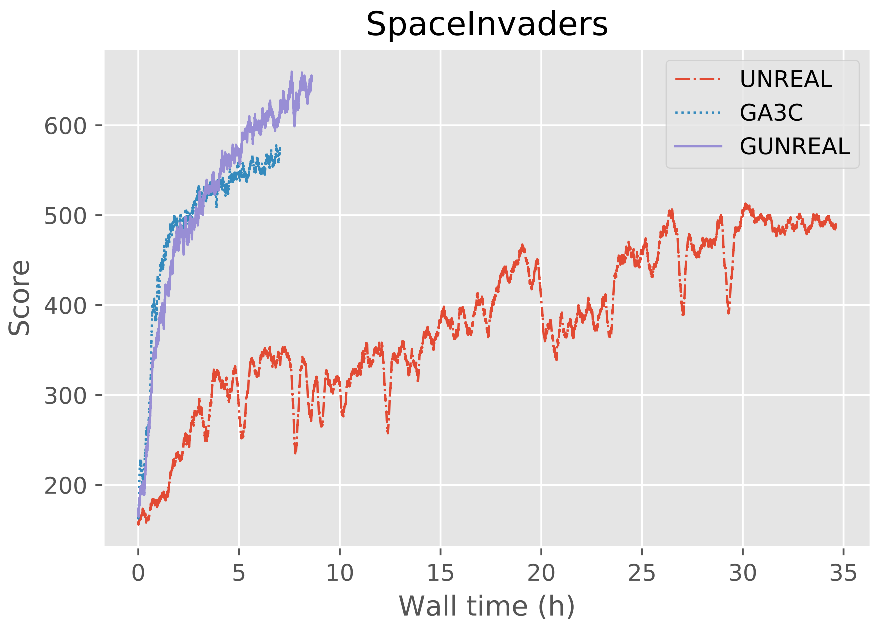


(b) Learning over time

Figure 11: Training results on OpenAI Gym (Pong)



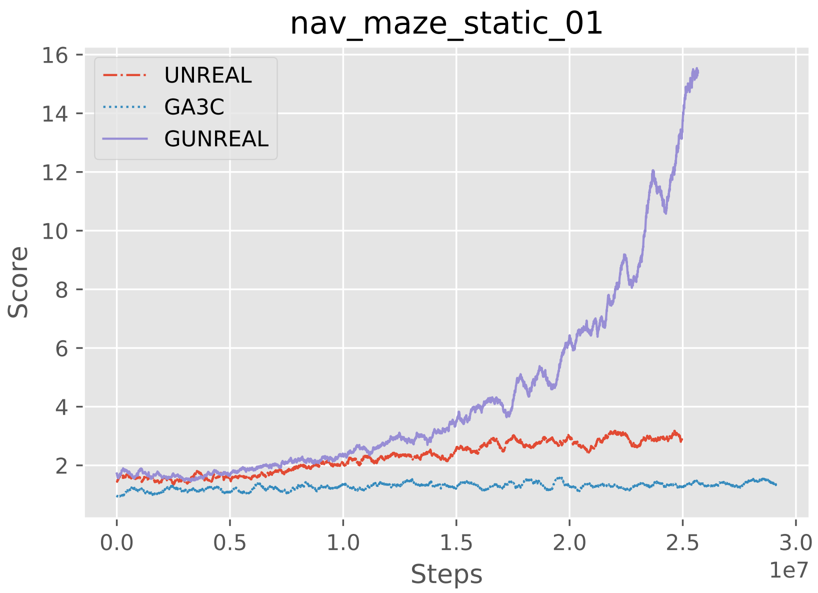
(a) Learning over steps



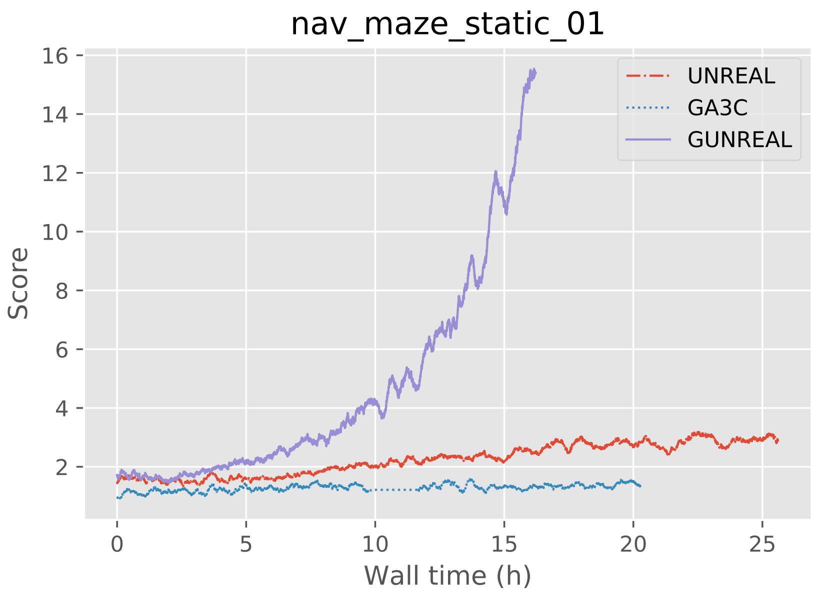
(b) Learning over time

Figure 12: Training results on OpenAI Gym (SpeceInvaders)

Figure 14 for *seekavoid_arena_01*.



(a) Learning over steps

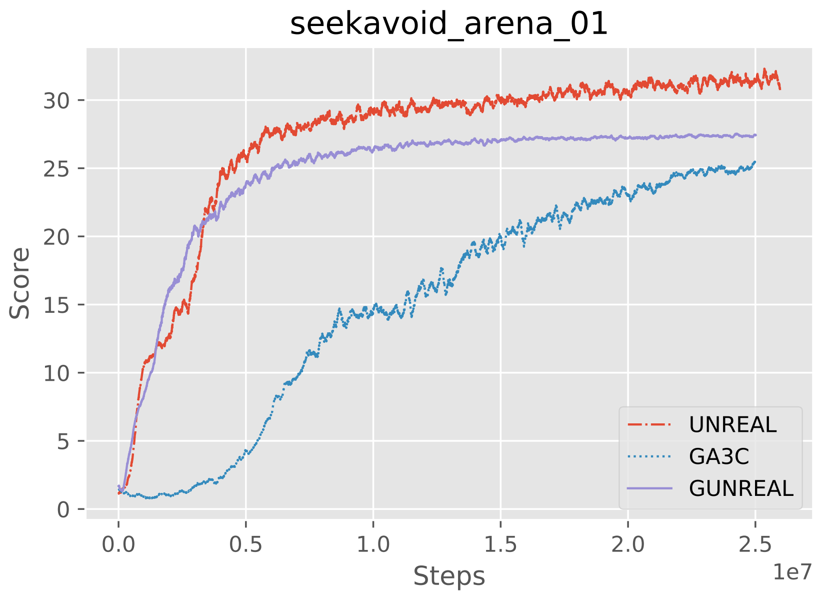


(b) Learning over time

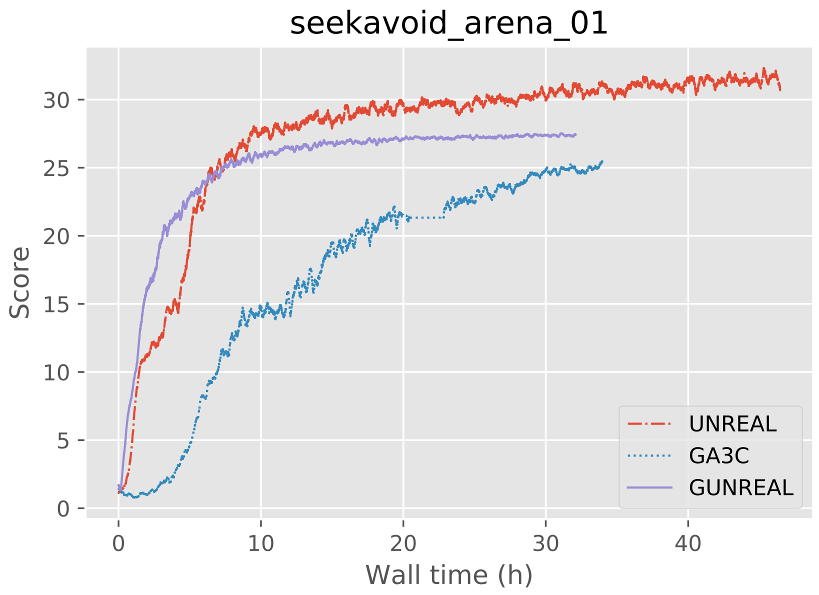
Figure 13: Training results on DeepMind Lab (*nav_maze_static_01*)

For *nav_maze_static_01*, GUNREAL performs clearly better than both UNREAL and GA3C, in terms of steps (Figure 13a), and time (Figure 13b). GUNREAL was able to reach scores of 15, which is 5.0 times more than UNREAL, and also finished training 1.6 times faster. Training results for *seekavoid_arena_01* are shown in Figure 14a and Figure 14b. In the end, GUNREAL reached a score of 27, while UNREAL achieved a higher score of 31 and GA3C approached a score of 25. To reach the score of 25, GUNREAL had to perform 73% fewer steps than GA3C. GUNREAL finished training 1.5 times faster than UNREAL.

Because of the learning efficiency introduced from the auxiliary tasks, GUNREAL achieved higher scores than GA3C in more complex environments such as DeepMind Lab. Especially, for *nav_maze_static_01*, GUNREAL outperforms both UNREAL and GA3C in terms of speed and learn-



(a) Learning over steps



(b) Learning over time

Figure 14: Training results on DeepMind Lab (*seekavoid_arena_01*)

ing efficiency.

4.4 Discussion

Firstly we discuss the reason of large difference that appeared in scores with *nav_maze_static_01*. There are two *events* that give reward in the *nav_maze_static_01* environment. First, there is an intermediate (short-term) reward of +1 whenever you catch an apple while exploring the maze. Second, there is the goal state reward, which gives you a reward of +10, whenever you reach the goal (i.e. end destination) in the maze. The latter reward is further away from the agent, and can therefore be interpreted as a long-term reward. From the learning curves in Figure 13, we can thus say that GUNREAL was able to learn to reach the goal state (i.e. reaching the highly rewarding long-term reward), along with getting some intermediate rewards, while UNREAL and GA3C did not learn to reach the highly rewarding goal state and only achieved collecting short-term rewards.

The main reasons of the different scores with *nav_maze_static_01* are considered to be the presence of the auxiliary tasks and preprocessed observation input in Figure 7. We can say that the combination of preprocessed observation input, which was kept from GA3C, together with the added auxiliary tasks from UNREAL, resulted in GUNREAL achieving the long-term reward. GA3C had the same preprocessing for the observation input and could not learn to reach the long-term reward due to the lack of training auxiliary tasks. UNREAL has the auxiliary tasks, but does not have a preprocessing for the DNN input, instead it uses single RGB frames as input. Originally, UNREAL compensated the lack of input preprocessing with the usage of an LSTM in the DNN architecture, but the usage of LSTM was omitted for our experiments. Hence, it is the combination of both the preprocessing and the auxiliary tasks that allowed GUNREAL to learn reaching the goal state for this environment in the limited number of steps.

Secondly we discuss the reason of the large differences in aspect of *wall time per step* among benchmarks in Figure 10b, 11b, 12b, 13b, 14b. In general we can say that GA3C and GUNREAL execute faster than UNREAL because GA3C and GUNREAL consume less GPU memory than UNREAL as they only require one global DNN, therefore GA3C and GUNREAL do not lose time on synchronizing weights between the master model and the local copies for each respective agent. However, it is clear that the speed-up is less effective in DeepMind Lab than in OpenAI Gym: GUNREAL achieves 3.8 to 9.5 times shorter wall time per step in the Atari environment of OpenAI Gym in Figure 10b, 11b, 12b, while the difference in DeepMind Lab is 1.5 to 1.6 times in Figure 13b, 14b.

The main reason of the difference between the environments is due to the simulators themselves: the complexity of the task to learn, plus the required resources for simulation. Firstly, *Breakout* is a 2D Atari game from OpenAI Gym. The Atari games from OpenAI Gym runs on CPU and does not perform 3D rendering to generate state observations for the agent, hence the GPU is completely available for the learning agents. Secondly, *nav_maze_static_01* and *seekavoid_arena_01* are DeepMind Lab environments, which are more complex than the Atari games, as these consist of 3D games. DeepMind Lab must therefore perform 3D rendering to generate state input for the agents and thus also requires GPU usage, which results in the GPU being less available for the learning algorithms to compute DNN output. Hence, the GPU acceleration has less effect in terms of wall time, as more of the wall time is spent on the simulator side.

5 Applicability to other applications

GUNREAL can be applied to other applications which use 2D or 3D images, where image frames of time series such as a movie are used as input to a DNN. Although we evaluated GUNREAL on 2D games with Atari of OpenAI Gym and 3D games with DeepMind Lab in section 4, in which RGB images are used as input to a DNN in GUNREAL, GUNREAL is independent of the environments. This is because in GUNREAL a DNN just receives images from an environment and sends selected action from output of the DNN to the environment. Also, a DNN in GUNREAL is generic, except the input image size to the DNN. GUNREAL is particularly more effective for complex tasks where auxiliary tasks help agents learning more efficient, such as robotics and autonomous driving which uses 3D images, and mechanical design and air conditioning in data centers.

Parameter tuning for the DNN and learning is required to optimize to a specific application, such as DNN modification according to image size, batch sizes for prediction and training, experience replay buffer size, the size of t_{max}, learning rate, update function, and loss functions for each task. PC receives images as input and processes to smaller images in which each pixel contains the average value of adjacent pixels on the original image. Although we assume input images to be 84 by 84 RGB in section 3.1 and Figure 8, PC can be used to smaller or larger input images by modifying the DNN. However in order to optimize the effectiveness of Pixel Control to a specific application, parameter tuning is required such as crop size of input image and output pixel size. Since VR simply reuses the same DNN with for the main A3C task, VR can be applied to other applications which can be applied to A3C without modification. RP can be also applied to other applications without modification because RP just calculates and categorizes rewards by sign of the rewards to be positive, negative, or zero.

6 Related work

There have been efforts to parallelize and accelerate deep reinforcement learning algorithms. DQN [19] applied DNN to train an agent in reinforcement learning and used GPU acceleration to train a DNN with a single agent on Atari games. Successors of DQN improved several aspects of the algorithm, such as the sampling method for experience replay [22], the update formula [26] and the neural network architecture [27]. The usage of an experience replay memory buffer made DQN and its successors benefit better from GPU acceleration. DistBelief [9] has been used to train a deep network with billions of parameters using tens of thousands of CPU cores. Its successor, the general reinforcement learning architecture, Gorila [20], distributed DQN across a cluster using CPU cores on each machine. Gorila outperformed the standard DQN in a variety of Atari games. A3C is an on-policy (policy-based) deep reinforcement learning algorithm (also called a policy gradient method) rather than off-policy algorithms such as DQN. A3C takes 4 days of training to solve Atari games using multiple CPU cores on a single machine without the usage of a GPU. GA3C [2] improved the GPU acceleration for A3C, allowing to solve Atari games on a single machine within a day. A novel algorithm agnostic framework for efficient parallelization of deep reinforcement learning called PAAC has been proposed [7], and has been demonstrated with a synchronous version of Advantage Actor-Critic (A2C) implementation on a GPU.

Stooke et al. [24] proposed parallelization methods for accelerating deep reinforcement learning on multi-GPU environments for both policy gradient methods such as A3C and Proximal Policy Optimization (PPO), and Q-value learning methods such as DQN. They extended the existing algorithms to multiple GPUs using a single machine such as NVIDIA DGX-1 (8 GPUs). They improved batched inference to increase GPU utilization, by making batch sizes considerably larger than are standard, without harming sample complexity or final game score. They use multiple simulator instances per process, and also avoid idleness (simulation on CPU and inference on GPU), by forming two alternative groups of simulator processes. They implemented both synchronous and asynchronous versions of the deep reinforcement learning algorithms using multiple GPUs. As a distributed deep reinforcement learning algorithm with a centralized learner with a single GPU, Ape-X [13] focuses on generating more data and selecting from it in distributed environment by introducing distributed version of prioritized experience replay. Unlike the A3C-based agents, in which agents send gradients to the master DNN, the actors focus on generating and accumulating experiences by selecting actions according to a shared neural network, and the learners replays samples of experience and updates the neural network. Ape-X accelerated training and also achieved higher scores than Gorila and prioritized DQN on Atari games. Ape-X DQN using 376 cores and one GPU achieved higher scores on Atari games than A3C, Gorila DQN, and UNREAL. IMPALA [10] extended policy gradient methods to distributed multi-GPU training. They mitigated policy lag that occurs in asynchronous update such as in GA3C, by combining decoupled acting and learning with a novel off-policy correction method called V-trace. Like Ape-X, actors send experiences rather than gradients to a centralized learner and they use a GPU to perform updates on mini-batches of experiences. IMPALA can be used with a learner that is distributed across multiple GPUs to train large networks efficiently, where the parameters are distributed across the learner machines and each actor retrieves the parameters from all the learners in parallel.

Some approaches to improve learning efficiency of deep reinforcement learning have been also proposed. Evolutionary strategies (ES) or gradient evolution, as an alternative to popular Markov Decision Process-based reinforcement learning techniques have been proposed [21]. They apply heuristic search procedures inspired by natural evolution rather than MDP. The algorithm scales with the number of CPUs, by using a novel communication strategy based on common random numbers, where each agent only needs to communicate scalars, making it possible to scale to over a thousand parallel agents. ES solved 3D humanoid walking and achieved competitive results on most Atari games. Harmer et al applied supervised Imitation Learning (IL), learning via guidance from an expert teacher, as initial training of reinforcement learning [12]. Imitation learning provides the agent with prior knowledge about effective strategies for behaving in the world, and is used to speed up training by teaching with examples to resemble how we teach. The algorithm allows multiple actions to be selected at every time-step, to allow complex behaviors to be learnt with higher quality expert data and multi-action policies. They showed training using a 3D first person shooter (FPS) style video game.

Our work focuses on fast and highly efficient deep reinforcement learning, using a single DNN on a machine with multiple CPU cores and a single GPU. We accelerated a state-of-the-art multi-task learning with A3C and auxiliary tasks by extending the GA3C architecture to improve GPU utilization. We also improved learning efficiency by reintroducing preprocessing phase from (G)A3C. Applying the existing techniques for multi-GPU and distributed environments to GUNREAL for further acceleration and investigating to apply other approaches such as ES and IL to GUNREAL for better learning efficiency, are possible directions to future improvement.

7 Conclusion

We presented GUNREAL which extends GA3C with the auxiliary tasks from UNREAL to achieve higher learning efficiency and benefiting from GPU acceleration. GUNREAL adds a memory buffer to each agent for experience replay and two additional prediction lines for the value function replay and pixel control tasks. The DNN model has been extended with additional output layers for the pixel control and reward prediction tasks. In addition, we evaluated the necessity of a preprocessing phase and improved the throughput on the GPU by changing the data format from NHWC to NCHW batches. GUNREAL especially performs better in more complex applications, such as DeepMind Lab, when compared to GA3C. The benefits we gain from using an architecture that is better suited for GPU acceleration allows GUNREAL to learn faster in real-time than UNREAL.

Since the A3C algorithm seemed to perform better with an LSTM in the DNN model, a future improvement could be the integration of an LSTM into GUNREAL's model. Another possibility for future research could be to assess the applicability of GUNREAL in real-world applications. Scalability of GUNREAL to HPC clusters in order to increase the learning speed is also an interesting possibility.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensor-

Flow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. In *International Conference on Learning Representations*, April 24–26, 2017.

[3] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler and Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab, December 2016.

[4] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym, June 2016.

[7] A. V. Clemente, H. N. Castejón, and A. Chandra. Efficient Parallel Methods for Deep Reinforcement Learning. In *3rd Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM '17)*, June 12–14, 2017.

[8] Youri Coppens, Koichi Shirahata, Takuya Fukagai, Yasumoto Tomita, and Atsushi Ike. GUNREAL: GPU-accelerated unsupervised reinforcement and auxiliary learning. In *Proceedings of the Fifth International Symposium on Computing and Networking (CANDAR'17)*, 2017.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[10] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[11] Iuri Frosio. GA3C: A Hybrid CPU/GPU Implementation of A3C for Deep Reinforcement Learning. Talk at the GPU Technology Conference (GTC '17), May 2017.

[12] Jack Harmer, Linus Gisslén, Henrik Holst, Joakim Bergdahl, Tom Olsson, Kristoffer Sjöö, and Magnus Nordin. Imitation learning with concurrent actions in 3d games. *arXiv preprint arXiv:1803.05402*, 2018.

[13] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[14] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *International Conference on Learning Representations*, April 24–26, 2017.

[15] Masayuki Matsumoto and Okihide Hikosaka. Lateral habenula as a source of negative reward signals in dopamine neurons. *Nature*, 447(7148):1111–1115, 2007.

[16] Masayuki Matsumoto and Okihide Hikosaka. Representation of negative motivational value in the primate lateral habenula. *Nature neuroscience*, 12(1):77–84, 2009.

[17] Masayuki Matsumoto and Okihide Hikosaka. Two types of dopamine neuron distinctly convey positive and negative motivational signals. *Nature*, 459(7248):837–841, 2009.

[18] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, June 2016. PMLR.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. Letter.

[20] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. In *ICML Deep Learning workshop*, 2015.

[21] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[22] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations*, Puerto Rico, 2016.

[23] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[24] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[25] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT Press, Second Edition, 1998.

[26] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[27] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[28] Tianshu Wei, Yanzhi Wang, and Qi Zhu. Deep reinforcement learning for building hvac control. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 22:1–22:6, New York, NY, USA, June 18–22, 2017. ACM.