# Tackling Transfer Learning and Catastrophic Forgetting Using Asynchronous Methods for Deep Reinforcement Learning

Extended Abstract

Paper #1
João Ribeiro
Instituto Superior Técnico
joao.g.ribeiro@tecnico.ulisboa.pt

## ABSTRACT

In this paper we evaluate how asynchronous methods for deep reinforcement learning can be augmented in order to handle transfer learning and overcome catastrophic forgetting. We introduce an hybrid version of the Asynchronous Advantage Actor-Critic on GPU (GA3C), capable of (i) transferring knowledge from a previously learned task to a new one that is now required to learned and (ii) remember how to perform the previously learned tasks as it learns the new one. We show that by learning multiple tasks it is possible to improve the learning efficiency for a new, similar task, and that by augmenting the GA3C's with the Elastic Weight Consolidation (EWC) algorithm, it is possible to overcome a substantial amount of catastrophic forgetting. We make our source code publicly available at https://github.com/jmribeiro/UGP.

## KEYWORDS

Intelligent Agents; Deep Reinforcement Learning; Multi-Task Learning; Transfer Learning; Catastrophic Forgetting; Asynchronous Advantage Actor-Critic

## 1 INTRODUCTION

More and more, there has been an increasingly larger success on machine learning when it comes to intelligent agents and their applicability to different tasks. Seeing as traditional agents were usually unable to generalize as well as current agents, they were usually created to solve specific tasks [3, 6, 9, 14, 23, 27, 28]. Recent advances can be credited to the success of the so called deep learning techniques [16, 17]. These techniques made possible for the agents to directly sense input from within their environment and from it extract which features are relevant and which features are not (given, of course, the current task's objective). Traditional agents, which did not have these techniques, required their developers to manually extract the features from the raw sensorial input, and therefore introducing a possibly large human bias into the agent. This can be counter productive for tasks that share the same environment, such games from the Atari2600 platform [4] (where the goal is to achieve the highest score possible). These games all share the same interface - the game screen. Newer agents, such as Deep Q-Network [21, 22], have proven successful for learning different tasks, by being only given raw input (without any features manually extracted). They suffer, however from a few limitations. Most of the time, these agents require multiple instances, each learning a separate game individually. Even when using a single instance, these agents, when faced with a new task after learning a previous one, would have to learn the second task again from scratch (even if both were very similar). Even though there have been some agents successfull at handling multiple tasks [5, 8], this inability to transfer knowledge between tasks, is something that has only been done in learning contexts where the algorithms have the entire task's dataset available at their disposal [24]. Since agents often operate in environments and acquire data in real time (something known as online learning), they do not have the entire dataset available at their disposal. Another limitation, even more severe than the first one, is the agents' inability to remember previous tasks, when learning new ones (a problem known in machine learning as catastrophic forgetting [12, 13]). Due to these limitations, existing deep learning techniques have proven unsuccessful at creating a single agent capable of learning in such a generalized manner on a single instance of itself.

### 1.1 The Problem

*A father of three wants his sons to learn three sports - football, basketball and volleyball. He enrols his first son on football practice (training four of the five days of the week), the second son on basketball (also training four of the five days of the week) and the third son on both sports (two days for each sport). After a few months, the first boy learned to play football very well but performs badly on basketball. The second boy learned to play basketball very well but performs badly on football. The third boy, as expected, learned both sports, however not playing as well on each of them as his brother that trained for that specific sport. If the father now enrolls all three on volleyball practice (four days per week), will the third boy be able to learn the sport more efficiently than his two brothers? Will any of the boys forget how to play their previous sport?*

In this work we tested these hypothesis with intelligent agents. We introduce a single general-purpose agent that simultaneously learns a large number of tasks, capable of sharing knowledge between similar ones and remembering how to perform previously learnt tasks after learning new ones. These hypothesis we tested were:

- Compared to *single-task learning*, does *multi-task learning*, i.e., learning multiple tasks simultaneously, improve the agent's efficiency when learning a new, similar task? Is the agent able to transfer learnt knowledge from previous tasks

*Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. Andre, S. Koenig (eds.), July 2018, Stockholm, Sweden*

https://doi.org/doi

into the new one? What we mean by "efficiency", is, is it capable of achieving the same performance as its single-task counterpart, with fewer training iterations?

- Does applying Elastic Weight Consolidation Algorithm (EWC) [13], an algorithm that tackles catastrophic forgetting, allow the multi-task agent to maintain the acquired knowledge from previous tasks after learning new ones? As said by Kirkpatrick et al.[13], *"The ability to learn tasks in a sequential fashion is crucial to the development of artificial intelligence"*.

### 1.2 Contribution

We introduce a novel, state-of-the-art, deep reinforcement learning agent named the Universal Game Player (UGP). With the UGP, we explore how recent asynchronous methods for reinforcement learning can be used for multiple tasks, tackling transfer learning, and augmented with the EWC algorithm [13], which alleviates the problem of catastrophic forgetting in neural networks. We create a general-purpose scalable agent that, using no other information from its environment other than high-dimensional sensorial data (in this case, the game screen, in addition to the current score for reinforcement), and tested our approach to see if it could:

- Learn multiple tasks on a single instance of itself, instead of having multiple instances, each learning a different task.
- Transfer learnt knowledge between tasks that are very similar. The agent, trained on four first tasks, efficiently learns a fifth new task, better than four individually trained agents for each of the first four tasks.
- Learn multiple tasks in a *continuous learning* context. The UGP adapts to new experience, without forgetting previous one.

We name this new agent the UGP due to its capability of being applied to different tasks. Even though the agent was only tested on five total tasks, consisting on virtual environments from the OpenAI Gym Toolkit [7], build from the Atari2600 platform [4], we provide a module called an environment handler, which resized the original game screen into a normalized shape, therefore allowing the UGP to be applicable to different platforms. We do not explore different platforms in this dissertation, other than the Atari2600, which is used as an industry standard testing platform for reinforcement learning agents.

## 2 BACKGROUND

### 2.1 Multi-Task Learning

*Multi-task learning* is a learning context where an agent must learn multiple tasks at the time. In the opposite (and most common) approach, *single-task learning*, when the need to perform multiple tasks appears, a separate agent is usually instantiated for each of the individual tasks, and learns every single one individually [21, 22]. The goal in multi-task learning context is for the agent to be as general-purpose and versatile as possible. The agent is expected to have a better generalization performance than several independent agents learning every single task independently [8]. One approach to multi-task learning is to store datasets from all tasks (if available) in an agent's memory system and then using them for training later on [13]. Since the required amount of memory space is proportional to the number of tasks, this approach impractical when learning large numbers of tasks [13]. The agent had to store the data points for all tasks in its memory, before starting their learning process. The solution is therefore to learn tasks sequentially, in a continuous learning context. When learning tasks in a continuous, learning context, a problem arises, known as catastrophic forgetting.

### 2.2 Catastrophic Forgetting

*Catastrophic forgetting* occurs when a neural network "forgets" how to perform previously learnt tasks in order to learn new ones. When first training for a task $T_a$, the goal is to update the neural network's parameters $\theta$ in order to minimize the task's loss function $L_a(\theta)$, and therefore maximizing its performance measure for $T_a$. After training, the network is able to obtain a set of parameters $\theta_a$ which maximizes the performance for $T_a$. However when sequentially learning a new task, $T_b$, the goal is now to minimizing $T_b$'s loss function $L_a(\theta)$ in order to maximize its performance on $T_b$. The network's parameters will therefore be updated accordingly, shifting from $\theta_a$ to $\theta_b$. These new values may not maximize the agent's performance on $T_a$, so in other words, the neural network "catastrophically" forgot how to perform the first task.

### 2.3 Reinforcement Learning

*Reinforcement Learning* is a sub field of machine learning that studies how an agent can be programmed to learn a task by a process of trial-and-error. In a reinforcement learning context, an agent is usually placed directly in the task's environment. A reinforcement learning task can be modeled by resorting to the Markov Decision Process (MDP) framework. An MDP is defined as a *system under control of a decision maker (agent) which is observable as it evolves through time.* An MDP can be represented by quadruple **(S, A, $\mathbf{P}_a, a \in A$, R)**, where **S** represents the set of all possible states where the system can be in any given instant in time, **A** represents the set of all possible actions that the agent can execute in any given state, $\mathbf{P}_a, a \in A$ represents the transition probabilities for all states, for a given action and **R** represents the rewards the agent can obtain by executing an action $a$ in a state $s$. The agent $\alpha$, interacts with an environment $\epsilon$ over several *episodes.* Each episode consists on a discrete sequence of time steps $t$, where $\epsilon$ can be in a given state $s_t$. In each time step $t$, the agent $\alpha$ receives $s_t$ from the set of states **S** abd chooses an action $a_t$ from the set of possible actions **A** (according to the agent's policy $\pi$. A policy function $\pi(s)$ is a function that for each state $s$ returns the probability of executing each action on that state. It can be either a continuous or discrete probability distribution, depending on the action's space **A**. The agent $\alpha$ then executes $a_t$ in $s_t$ and $\epsilon$ transitions in time into the next state $s_{t+1}$. $\alpha$ is then rewarded accordingly to how well it did by executing $a_t$ in $s_t$ by receiving a reward value $r_t$. This sequence is repeated until either the episode ends (i.e., when the agent either reaches a terminal state) or the environment resets itself. A reinforcement learning task's data point can therefore by represented by the tuple $(s_t, a_t, r_t, s_{t+1})$. The better an agent performs by executing an action on a certain environment's state, the higher the reward it receives. The goal for an agent in a reinforcement learning context is therefore to learn a control strategy, called a policy $\pi$, that represents the agent's decision making process and maximizes the accumulated reward over time. Two other important concepts in reinforcement learning are *value functions* and *Q-value functions*. A value function $V(s)$ is a function that for each state $s$ returns the state's value. A state's value $V(s)$ represents how good it is for the agent $\alpha$ to currently be in. A Q-function $Q(s, a)$ is a function that for each state-action pair ($s$,$a$) returns its value. A state-action pair's value $Q(s, a)$ represents how good it is for the agent $\alpha$ to execute the action $a$ in the state $s$. Both $V(s)$ and $Q(s, a)$ provide *long-term information* regarding how good a state s (in the case of $V(s)$) or a state-action pair ($s$,$a$) (in the case of $Q(s, a)$) are to the agent. For neural networks in a reinforcement learning context, the data points are represented by tuple ($s_t$, $a_t$, $r_t$, $s_{t+1}$). Unlike in a supervised learning approach, the network is embedded on an agent interacting with an environment. The agent takes in as input the environment's current state $s_t$, gives it as input to the network, the network forward propagates it through its neurons and obtains an output $a_t$, representing the action that the agent executes on the environment's current state $s_t$. After executing the action $a_t$ on the environment's state $s_t$, receives a reward value $r_t$ and the environment evolves into a new state $s_{t+1}$. The reward value is then used in the network's Loss function $L(\theta)$ that, like in the supervised learning approach, has the objective of being minimized by adjusting the network parameters $\theta$. The parameter adjustment process in a reinforcement learning context is therefore the same as in a supervised learning context.

## 3 RELATED WORK

### 3.1 The Asynchronous Advantage Actor-Critic

DeepMind's newer attempt at a general-purpose agent, the Asynchronous Advantage Actor-Critic (A3C) [20], proved to learn tasks much faster than their previous agent, DeepMind's Deep Q-Network (DQN) [22]. The reason for this speed increase is its asynchronous execution of multiple agents, called "actor-learners", on multiple instances a single task's environment. The name A3C stands for *Asynchronous, Advantage, Actor-Critic*. *Asynchronous* means that the algorithm has multiple independent agents running asynchronously on several instances of the task's environment. Each of these agents has a local copy of the global network. Their local network is used to act on the environments. The consequences of the actions from each actor-learner are then accumulated and used to update the global network. *Actor-Critic* means the A3C uses both a policy function $\pi$ (the Actor), trained using policy gradient method [18], and a value function $V$ (the Critic) in two separate fully-connected layers of the global network. The A3C, proved effective in tasks with both discrete and continuous action spaces (control tasks from the MuJoCo physics simulator [29], 2D games from the Atari2600 platform [4] and also 3D games such as TORCS 3D Car Racing [30] and Labyrinth [20]). The A3C was also able to learn faster and perform the tasks better than the DQN. It performed the tasks better than DQN because the global experience became more diverse, due to several independent agents being used to obtain it.

### 3.2 The Asynchronous Advantage Actor-Critic on a GPU

Mnih et al. [20] believe the success of A3C, both in 2D and 3D environments, makes it *"the most general and successful reinforcement learning agent to date"*. There were, however, newer contributions already made over its architecture [2, 5]. One of them is the Asynchronous Advantage Actor-Critic on a GPU (GA3C). Proposed by Babaeizadeh et al. [2], the GA3C is a version of the A3C algorithm that uses the Graphics Processing Unit (GPU) in order to speedup training. It has the following differences:

(1) The GA3C holds only a single global network, *"centralizing predictions and training updates and removing the need for synchronization"*, while the A3C holds a global network and several local networks on each actor-learner.
(2) The GA3C has its actor-learners placing *prediction requests* on a global queue, which an auxiliary thread called *Predictor* then collects a batch of and forward-propagates through the global network in order to obtain the predictions. The A3C had several local networks running forward-propagations of their agent's requests. Since with the GA3C there's a single network running forward-propagation with a batch instead of a single datapoint, the GPU can be used to obtain a speedup.
(3) The GA3C, like with the predictions, has its actor-learners placing *training requests* on a global queue, which an auxiliary thread called *Trainer* then collects a batch of, computes the gradients and updates the global network's parameters. In contrast, the A3C's actor-learners computed and accumulated the gradients themselves and then sent them to the global network, which asynchronously performed the update. Like with the predictions, by collecting a larger batch, this gradient computation is done using the GPU, and therefore obtaining a speedup.

### 3.3 Elastic Weight Consolidation Algorithm

The EWC, introduced by Kirkpatrick et al. [13], is an algorithm that aims to prevent catastrophic forgetting in Neural Networks when an agent is sequentially trained for tasks. According to Kemker et al. [12], catastrophic forgetting is a problem that has not yet been solved. Given two tasks, $T_a$ and $T_b$ with their respective Loss functions, $L_a(\theta)$ and $L_b(\theta)$, and a neural network, optimally trained for $T_a$, with internal parameters $\theta = \theta_a^*$ that minimize $L_a(\theta)$, catastrophic forgetting occurs when the network trains for $T_b$ and, by trying to minimize $L_b(\theta)$, loses $\theta_a^*$ and obtains $\theta_b^*$. The EWC tackles this problem by using a Loss function that combines both $L_b(\theta)$ with $\theta_a^*$, trying to protect $\theta_a^*$. The authors claim in their work that the EWC is both scalable and effective. The do so by evaluating the EWC on several classification tasks created from the MNIST dataset and several Reinforcement Learning tasks consisting of video games from the Atari2600 platform. In order to play the Atari2600 video games, they augmented the DQN with the EWC. The EWC algorithm was only tested, however, in ten of the 49 games from the Atari 2600 platform. The augmented DQN agent with EWC was given another algorithm to infer which task it was given. The combined agent did not forget how to play previous games. However it did not achieve such good performances as the single DQN trained individually for each game.

## 3.4 The Mid-level Feature Transfer Approach

In [24], Oquab et al. propose a solution regarding the transfer of learnt knowledge in convolutional neural networks. Given a source task similar to the target task, where unlike the target task, training data is abundant, the convolutional neural network can be first trained for the source task and the internal layers of the network (with their parameters) reused for the training of the target task. The authors' approach was as following.

(1) Train a convolutional neural network $N_a$ for object recognition in images from the ImageNet 2012 Large Scale Visual Recognition Challenge (ILSVRC-2012 [26]). The dataset contains 1.2 million images and 1000 classes. Oquab et al. use the ImageNet [15] architecture in their work.
(2) Transfer the internal, trained layers, from $N_a$ into a new instance of the same convolutional neural network $N_b$.
(3) Train the convolutional neural network $N_b$ for object recognition in images from the PASCAL Visual Object Classes Challenge 2007 (PASCAL VOC 2007 [11]). The dataset contains only 9,963 images and 20 classes.

This approach demonstrated enhancements regarding average precision (in %) when compared to both the PASCAL VOC 2007 [11] winners (INRIA [19]) by 18.3% and more recent approaches (NUS-PSL [10]) by 7.2%. This technique is used on the UGP, by reusing the internal layers from the convolutional neural network (used to extract the features from the game screens) and changing only the output layers to fit the different task's action spaces.

## 3.5 The Hybrid A3C

Neither the A3C nor the GA3C were originally designed to handle multi-task learning. Birck et. al [5] modify the A3C and give each of the actor-learners different instances of different environments (instead of giving all of them different instances of the same environment). With this approach, they seek to test if by learning two tasks simultaneously, the resulting agent is able to obtain a better policy than two single-task A3C instances, individually trained on each task. Two instances of their resulting agent, named the Hybrid A3C, are ran on two pairs of games - Pong/ Breakout and Space Invaders/Demon Attack. The pairs were chosen accordingly to the similarities between games. Both agents outperform single-task A3C instances which trained on each game individually. In this work we test a different hypothesis - if learning multiple tasks allows the agent to learn a new task better than a version which learned a single-task.

## 3.6 Discussion

By taking all the existing work and relevant literature into account, we contribute with a novel agent which was created not only solve specific tasks, but tasks with the same high-dimensional sensorial input, but was also able to:

(1) Learn multiple tasks at the same time.
(2) Transfer already learnt knowledge from previous tasks to newer, similar tasks.
(3) Remember how to perform previously learnt tasks after learning new tasks.

We used the GA3C [2], algorithm as the foundation for the UGP, given the success of the algorithm it was itself based on, the A3C [20], and the proven speedup of using GPU. However, since the GA3C algorithm has not been used in multi-task context nor designed for transfer learning, the algorithm was extended the same way the A3C was by Birck et. al [5], in order to address these novel challenges. We then added the EWC algorithm by changing the loss function for the actor-critic network in order to tackle catastrophic forgetting.

## 4 PROPOSED SOLUTION

We propose and test a novel solution, that combines elements from already existing work discussed in the related work section. These are the GA3C[2], the Hybrid A3C[5], the EWC[13] and the mid-level feature transfer approach for convolutional neural networks from[24]. We implemented our agent in Python 3 and setup the model using the Tensorflow[1] library, running all computational graph forward propagations and backward propagations using GPU. We resorted to the OpenAI gym toolkit[7], where available environments from the Atari2600 platform[4] were used as tasks. When interacting with an environment, the agent only has access to the game screen (pixels) and current score.

## 4.1 Requirements

The requirements for the UGP were the following:

**General-purpose, scalable agent:** The UGP must be applicable to a variety of different tasks, using nothing but high-dimensional sensorial input and a reward value for reinforcement. There can be no manual feature extraction which is task-specific.
**Multi-task learning:** The UGP must be capable of learning multiple tasks in a single instance of itself.
**Transfer learning:** The UGP must be capable of transferring already learnt knowledge from previous tasks to newer, similar tasks, and therefore allowing a more efficient training.
**Overcoming catastrophic forgetting:** The UGP must be capable of remembering how to perform previously learnt tasks after learning new tasks.

*4.1.1 Multi-Task Learning.* Even though the GA3C uses multiple agents running asynchronously, each has the purpose of running independently on environments from the same task. This cannot be considered multi-task learning, since the asynchronous agents are all learning the same task. Therefore, by augmenting the GA3C's architecture and giving the actor-learners different combinations of environments (the same way it was done to the A3C's architecture by Birck et. al[5]), it was possible to apply the GA3C to a multi-task learning context

*4.1.2 Transfer Learning.* In order to tackle the knowledge transfer between similar tasks, the mid-level feature transfer learning approach from[24] will be used. This approach consists on keeping the parameters on the internal layers of the convolutional neural network when switching between tasks, modifying only the output layers in order to fit the network to each task's output space.

#### 4.1.3 *Catastrophic Forgetting.*

In order to tackle catastrophic forgetting, the Hybrid GA3C architecture was augmented with the EWC algorithm, by changing its default loss function.

## 4.2 Architecture

The UGP consists on a modified version of the GA3C both capable of handling multi-task learning (through mid-level feature transfer [24]) and overcoming catastrophic forgetting (being augmented with the EWC algorithm [13]). Figure1 provides an overview of the UGP's architecture. There are five main components:

**The Actor-Learners Agents:** Independent processes interacting with individual instances of environments. They can all be interacting with different instances of the same environment (single-task), different instances of multiple environments (multi-task) and combinations of both, where some agents interact with different instances of the same environment and other with different instances of another environment. They query the predictor threads with the current state for its policy and feed the trainer threads with a batch of recent experiences and rewards.

**The Environment Handlers:** Interfaces between the actor-learners and the environments. Provide an abstraction for the agent and can therefore made for any environment that returns the screen's pixels and the reward, and therefore not only the Atari2600 platform.

**The Predictor Threads:** Threads that forward propagate a batch of input states, called prediction requests, through the actor-critic network. They then send the policies to the requesting actors-learners.

**The Trainer Threads:** Threads that back propagate a batch of datapoints through the actor-critic network, therefore updating its internal parameters.

**The Actor-Critic Network:** A deep convolutional neural network followed by actor-critic layers, which returns a policy and a value (respectively) for a given input state. A new actor-critic layer is created for every new environment that the network is required to learn.

#### 4.2.1 ***The Actor-Learners and the Interaction Process.***

An *actor-learner agent* is a single process that interacts with its own instance of the environment and communicates with the actor-critic network through the predictor and trainer threads, requesting policies through prediction requests and feeding the network with experiences batches, respectively. In a given timestep $t$, the actor-learner starts by requesting the current state $s_t$ and reward $r_t$ from its environment handler. The environment handler stacks the current preprocessed screen observation (frame) $o_t$ with the previous preprocessed three observation $o_{t-1}$, $o_{t-2}$ and $o_{t-3}$. We use the same pre-processing method used by Babaeizadeh et al. with the GA3C[2], which was based on Mnih et al. pre-processing method with the DQN). An overview of the interaction between an actor-learner and its environment handler is described in2. After obtaining the current state $s_t$ from the environment handler, the actor-learner requests a prediction from the actor-critic network. A *prediction* is a tuple containing the policy $\pi(s_t)$ and value $V(s_t)$ for a given state $s_t$. From the policy $\pi(s_t)$, which is a distribution containing for each action, the probability of the action being selected $\mathbf{P}_a, a \in A$, the actor-learner selects the action $a_t$. When the UGP is training, the actor-learners are in exploration mode. In exploration mode, the actor-learner randomly select an action from the policy, non-uniformly, given the actions probabilities. Actions with higher probability values naturally have a higher probability of being selected. On the other hand, when the UGP is playing, the actor-learners are in exploitation mode. In exploitation mode, the actor-learner first selects the action with the highest probability from the policy $\pi(s_t)$, and check which other action probabilities are at a given distance (we use 0.10 but define this exploitation distance as an hyper parameter). The actor-learner then randomly selects between the action with the highest probability and the actions within the exploitation distance of that probability. This random selection is done uniformly, meaning that all these actions now have the same probability of being selected. Now that the actor-learned has acquired the action $a_t$ to execute upon the given state $s_t$, it sends it to the environment handler, which updates the environment and returns the next state $s_{t+1}$ and reward $r_t$, then given to the actor-learner.

#### 4.2.2 ***The Prediction Process.***

When the actor-learners require an action $a_t$ to execute upon the current state $s_t$, they create a *prediction request* - a tuple containing their own unique id, the name of the environment they're acting upon, and the current state $s_t$. This is the same prediction process. The Predictor threads now process the prediction requests, taking them from the queue and stacking them in a batch. The size of this batch is configurable and we used the same size as Babaeizadeh et al. with the GA3C[2], which was 128. This number represents the maximum batch size for which the Predictor threads process forward propagate through the actor-critic network, which returns the corresponding predictions for each state in found the batch, as input. The reason for running a full batch through the actor-critic network is the speedup achieved by using the GPU to perform faster matrix multiplication[2]. The Predictor threads, now holding the predictions for the given states, return them to the actor-learners. The actor-learner then uses the policy $\pi(s_t)$ to select an action $a_t$. The actor-learner then executes the action on the environment, through the environment handler, and obtains the next state $s_{t+1}$, reward $r_t$ and a boolean telling if $s_{t+1}$ is a terminal state. This recent experience is now going to be used to update the actor-critic network. The actor-learner therefore creates a *datapoint*, a tuple containing the the current state ($s_t$), the executed action ($a_t$), the obtained policy ($\pi(s_t)$), the obtained value for the state ($V(s_t)$), the obtained reward from executing $a_t$ on $s_t$ ($r_t$), and a boolean telling if the next state $s_{t+1}$ is a terminal state or not. It does not need to contain the next state $s_{t+1}$. The datapoint is then stored in the actor-learner's recent experiences memory, now used to update the actor-critic network in the training process.

#### 4.2.3 ***The Training Process.***

An actor-learner, which has been collecting datapoints until it has finished an episode (by reaching a terminal state) or has been running for a certain number of timesteps, will prepare a batch for training the actor-critic network, called a recent experiences batch. The recent experiences batch contains datapoints $d_t$, for each timestep $t$, from a given timestep $T_{start}$, until another timestep $T_{end}$, This "certain number of timesteps" is a configurable parameter called $T_{max}$, representing an interval at which the actor-learners send the experiences for training. We used $T_{max} = 5$, the same value used by Babaeizadeh et al. with the GA3C[2]. Having collected datapoints and stacked them in the recent experiences batch, the actor-learner accumulates the rewards from the batch, using a discount factor. The reward accumulation process is the same used by Babaeizadeh et al. with the GA3C[2] and Mnih et al. with the A3C[20]. We used the same discount factor as Babaeizadeh et al.[2], 0.99. After running in reverse order through every timestep in the batch, accumulating the rewards, the newly updated batch with the discounted rewards is now returned. The actor-learner, now holding the recent experiences batch, with the discounted cumulative rewards, places the batch in a global training queue, accessible to all actor-learners and trainer threads. The trainer threads fetch datapoints from the training queue until they hold a training batch of a certain size, ready to be used for backpropagation. We use a training batch size of 128. Unlike the GA3C, where the entire training batch was composed by 128 datapoints from the same environment, given by multiple actor-learners, the UGP backpropagates a training batch composed by 128 datapoints from multiple environments, given by multiple actor-learners. Updating the actor-critic network with experiences from multiple environments at the same time was also done by Birck et al. with the Hybrid A3C[5]. When it comes to the backpropagation, Birck et al.[5] decided to change the optimizer from the RMSProp, used in the A3C, to Adam. With the UGP we sticked with the RMSProp optimizer [1], because it had also been used in the GA3C by Babaeizadeh et al.[2] and proven to work really well for single-task learning. The trainer threads then backpropagates a multi-task 128 datapoints batch, updating the network's parameters. Once again, like the Predictor threads, the reason for running a full batch through the actor-critic network is the speedup achieved by using the GPU to perform faster matrix multiplication[2]. An overview of the training process can be seen in Fig. 3.

4.2.4 ***The Actor-Critic Network.*** The actor-critic network is a deep convolutional neural network that follows an actor-critic approach. An overview of the model can be seen in Fig. 6. The actor-critic network therefore consists on single input network, shared between all environments, followed by several actor-critic layers, one per environment. We use introduce the mid-level feature transfer[24] approach, consisting on sharing the input layers and setting up individual output layers, in order to tackle multi-task learning. Even though the action spaces are the same between environments (since all share the Atari2600 platform), actions from different environments have different meanings (the same way two output layers that classify images may have the same number of output units, but the classes hold different meanings and therefore require multiple output layers[24]). The input network is made of two convolutional layers and one fully connected hidden layer. The convolutional layer are responsible for extracting the features from the high-dimensional sensorial data (game pixels), i.e., they take as input the current state $s_t$, which is a 4x84x84 tensor (four dimensional matrix), and return 32, 4x4, feature maps. Feature maps are 2-dimensional arrays that represent the features that may be relevant for selecting an action, given the environment's objective. These 32 feature maps are then flattened into 512 single values, which now serve as input extracted feature values, given to the fully connected hidden layer. This hidden layer contains 256 hidden units and its output is now redirected both the actor and critic layers. *The actor layer* is a fully connected hidden layer with $A$ hidden units (where $A$ is the number of actions for the environment). The actor's output estimates a policy function $\pi$ that consists on a distribution for a state, that for each action value, returns the probability of the action maximizing the expected cumulative reward for that state. It is through the actor actor that the agent selects an action to execute. *The critic layer* is a fully connected hidden layer, with a single unit, that estimates a value function $V$. The value of the current state represents how good it is for the agent to be on the state. The critic $V$ is primarily used to evaluate and update the actor function $\pi$, a method known as policy-gradient, from which the loss function is derived[18].

4.2.5 ***Loss Functions and Backpropagation.*** Training the actor-critic network consists on backpropagating the batches, which contain datapoints from multiple environments through the network's layers. For each different environment $\epsilon$ a loss function $L_\epsilon(\theta)$ is setup. This loss function is given to the RMSProp algorithm[25], which when given datapoints, computes the gradients and updates the network's parameters. When backpropagating using a multi-environment batch, the optimizer runs for each environment with its respective datapoints, on environment at the time. This is done using the environment's specific loss function. Even though they have different loss functions, there are common parameters that updated by all the loss functions. We can divide the parameters (weights and biases) into four separate sets - $\theta_{input}$, $\theta_\pi$ and $\theta_V$. The first set, $\theta_{input}$, represents the shared parameters from the input network. These parameters are shared by all environments and are updated when backpropagating datapoints from all environments. The second set $\theta_\pi$, represents the parameters from the actor layer of an environment, and are only updated by backpropagating datapoints from that specific environment. The third set $\theta_V$ represents the parameters from the critic layer of an environment, and like $\theta_\pi$, these are only updated when backpropagating datapoints from that specific environment. A single environment's loss function $L_\epsilon(\theta)$ therefore updates the environment's individual actor-critic parameters $\theta_\pi$ and $\theta_V$, and then shared parameters $\theta_{input}$. This loss function is a combination of two, one for the actor - $L_\pi(\theta)$, and one for the critic - $L_V(\theta)$. When minimizing $L_\pi(\theta)$, the parameters $\theta_{input}$ and $\theta_\pi$ are updated. When minimizing $L_V(\theta)$, the parameters $\theta_{input}$ and $\theta_V$ are updated. Like with the GA3C[2], these two loss functions are first setup and then combined into a single one - $L_\epsilon(\theta)$. Having $L_\pi(\theta)$ and $L_V(\theta)$ setup, these are combined into a single loss function $L_\epsilon(\theta)$. $L_\epsilon(\theta)$ is then given to the optimizer which now computes the gradients and updates the parameters in way that the loss is minimized. For each environment, the final, combined, loss function is $L_\epsilon(\theta) = L_\pi(\theta) + 0.5L_V(\theta)$. Like Babaeizadeh et al. with the GA3C[2] and Mnih et al. with the A3C[20], we reduce the critic loss by half in order to make policy learning faster than value learning, i.e., the agent will learn the best ways to act on a given state, faster than it learns how good it is for him to be on a given state. We then implemented the EWC algorithm in order to tackle catastrophic forgetting. An input network that has learned multiple environments $\epsilon_1, \epsilon_2, ... \epsilon_n$ and obtained parameters $\theta^*_{input}$, if given a new environment $\epsilon_{n+1}$ to learn, then with training, the input network's parameters will be updated from $\theta^*_{input}$ into $\theta^{**}_{input}$,

[1]http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

catastrophically forgetting how to perform on the previous environments. The EWC algorithm tackles this by modifying the $L_{\epsilon_{n+1}}(\theta)$ before starting the learning process. First we stored the stored the optimal parameters $\theta^*_{input}$. Using a samples from environments $\epsilon_1, \epsilon_2, ... \epsilon_n$ and the optimal parameters $\theta^*_{input}$, we computed the fisher information matrix $F$. The EWC algorithm has a parameter $\lambda$ which allows to specify how important are the old parameters, when learning a new task. An higher $\lambda$ means the network will not forget as much, by sacrificing performance on the new task. A lower $\lambda$ means the network is more willing to forget the old task in order to learn the new one. The final loss function $L_{\epsilon_{n+1}}(\theta)$ is now given to the optimizer, and the training can now begin.

## 5 EVALUATION

We had two hypothesis to test, one regarding transfer learning and another regarding catastrophic forgetting:

(1) Does multi-task learning improve the agent's efficiency when learning a new, similar task? Is the agent able to transfer learnt knowledge from previous tasks into the new one, being capable of achieving the same performance as its single-task counterpart, with fewer training iterations?
(2) Does applying EWC [13], an algorithm that tackles catastrophic forgetting, allow the multi-task agent to maintain the acquired knowledge from previous tasks after learning new ones?.

In order to test both our hypothesis we setup five different environments consisting on "bottom-up" shooting games, which we considered shared a lot of features. Both hypothesis require one source tasks and one target task. However, the first hypothesis requires multiple source tasks, so we instantiated four environments as source tasks and one environment as a target task. The OpenAI Gym environments used as source tasks (all from the Atari2600 platform) were *Assault* (Fig. 4, top left), *Phoenix* (Fig. 4, top right), *Carnival* (Fig. 4, bottom left) and *Demon Attack* (Fig. 4, bottom right). The environment used as target task (also from the Atari2600 platform), was *Space Invaders* (Fig. 5). For testing purposes, and to speed up training we resorted to the deterministic versions of these environments. In the deterministic version, episode always starts the same way, however, depending on the actions executed by the agent, it evolves accordingly. We then created five agents, using the implemented, modified GA3C architecture (without the EWC algorithm) and trained them each a total of 150000 episodes:

**AssaultNet:** A GA3C equivalent agent that learned how to play Assault. 16 actor-learners trained on Assault for a total of 150000 episodes.

**PhoenixNet:** A GA3C equivalent agent that learned how to play Phoenix for 150000 episodes. 16 actor-learners trained on Phoenix for a total of 150000 episodes.

**CarnivalNet:** A GA3C equivalent agent that learned how to play Carnival for 150000 episodes. 16 actor-learners trained on Carnival for a total of 150000 episodes.

**DemonNet:** A GA3C equivalent agent that learned how to play Demon Attack for 150000 episodes. 16 actor-learners trained on Demon Attack for a total of 150000 episodes.

**HybridNet:** An Hybrid GA3C, equivalent to the Hybrid A3C [5], but on GPU, that learned how to play all four tasks simultaneously. 16 actor-learners trained on all four tasks for a total of 150000 episodes. 4 actor-learners trained on Assault for 45230 total episodes, 4 actor-learners trained on Phoenix for 31932 total episodes, 4 actor-learners trained on Carnival for 44682 total episodes and 4 actor-learners trained on Demon Attack for 28156 total episodes. We require that the total amount of episodes from the combined environments matches 150000. We did not require each task to have the same number of episodes as the others, since episodes from some tasks were quicker than episodes from others.

Table 1 shows configuration choices and hyper parameters.

### 5.1 The First Hypothesis

*"Compared to single-task learning, does learning multiple tasks simultaneously improve the agent's performance when learning a new, similar task to the ones it has already learned?*

To test this hypothesis, we test if the HybridNet, which trained simultaneously for the four source tasks, is capable of achieving a better Avg Score / Ep. after training for an additional 50000 episodes on Space Invaders. In every episode $e \in [T_{start}, T_{end}]$, we recorded the episode score, the average score (Avg Score / Ep.) and the standard deviation. We display a learning curve by plotting the Avg Score / Ep. for each $e \in [T_{start}, T_{end}]$. We start by evaluating the learning of all five agents, comparing their learning curves on their respective environments. Since different environments have different score spaces, we normalized all values between 0 and 1, where 0 represents the lowest Avg Score / Ep. and 1 represents the highest Avg Score / Ep., relative to each environment. Fig. 7 displays the curves. As expected, by having to learn four environments simultaneously, the HybridNet's performance was worst than the other four agents. This makes sense, given the less amount of training data it receives compared with its single task counterparts. Taking these results into account, one could argue that if this comparison is done respectively to the number of training episodes the HybridNet actually trained in each environment (all adding up to a total of 150000), then the HybridNet was able to achieve a better Avg. Score / Ep. than the single-task agents, with fewer training episodes for the environments. We then compared the HybridNet with the other single-task agents given the exact number of training episodes for each environment (Figs. 12, 13, 14 and 15). Even though we do not consider this a fair comparison, given that, at the last episode for each environment (ep. 45230 for Assault, ep. 31932 for Phoenix, ep. 44682 for Carnival and ep. 28156 for Demon Attack), the HybridNet had trained almost a total of 150000 multi-task episodes, we can still argue that, with less training episodes on each environment, the HybridNet achieves a better Avg. Score / Ep. than the single-task agents, proving that when learning multiple tasks, the learning processes complement one another [5]. Naturally, it does not mean that the HybridNet agent, which trained on four environments adding up to a total of 150000 episodes, was capable of beating the single-task agents which trained on one single environment for the same amount of episodes (which makes perfect sense). We now consider an additional situation, our hypothesis - is the HybridNet is able to learn Space Invaders, better than other agents, by achieving a higher Avg. Score / Ep. after 50000 additional training episodes? We first evaluate all agents in all five tasks for a total of 100 episodes. For these measurements, all agents are ran in *exploitation mode*, where the actor-critic network's parameters aren't updated when interacting with the environments and the action selection is done using our *exploit* function. The exploit function, takes as input the policy $\pi$, which contains a distribution with the probabilities for all actions, and starts by selecting the action with the highest probability as pivot, adding it to a selection list. It then checks the probabilities from all other actions, and the ones which are at a certain *exploitation distance* from the pivot, are added to the selection list. It then uniformly returns an action from the selection list. We used an *exploitation distance* of 0.10. The obtained results are found in table 2. The HybridNet was the agent who better performed (without training) in Space Invaders. This result was as expected, given that the HybridNet's had a more diverse multi-task learning experience, it is very plausible that it is capable of a better generalization. We now train the five networks on Space Invaders for a total of 50000 additional episodes. Fig. 16 displays the resulting learning curves. The results show that after 50000 additional episodes, the HybridNet had the best learning curve, achieving at episode 50000, an Avg Score / Ep. of 440.44 points. However, this is not such a significant difference from the other agent's results, and therefore we decided to run the agents on another 100 episodes, in exploitation mode, to see if the HybridNet did in fact obtain substantially better results. Table 3 displays the results. The five agents now have a similar performance on Space Invaders, with the AssaultNet achieving the best results, with an Avg. Score / Ep. of 562.90 points). The HybridNet did not outperform the other single-task agents, as we were expecting. Taking all these results into account, we can conclude, although not very strongly, that the HybridNet did learn the environment a little bit better, as seen by its learning curve. Learning multiple source tasks simultaneously did provide a slight advantage when learning the new task (Fig. 16), but there was no strong advantage when performing the new task (Table 3). This discrepancy may be due to the difference between different learning curves being relatively small, showing that, even if the HybridNet's learning curve is above the AssaultNet's learning curve, the differences are negligible. In conclusion, it is plausible that learning multiple tasks simultaneously may in fact allow for an improved training of a new, similar task, by transferring learnt knowledge from old ones, but in order to provide a stronger claim, the play results in exploitation mode should have had the HybridNet displaying better results.

## 6 THE SECOND HYPOTHESIS

*"Does applying EWC, an algorithm that tackles catastrophic forgetting, allow the agent to maintain the acquired knowledge from previous tasks after learning new ones?"*. As it was expected, after training on Space Invaders for 50000 episodes, all agents catastrophically forgot how to perform their previous task (on in the case of the HybridNet, tasks). For disambiguation purposes, whenever we refer to an agent from now on, we will append the number of episodes it trained, this way it is possible to compare the agents after training on the original tasks (150000 episodes) and after training an additional 50000 episodes on Space Invaders (200000). Table 4 displays the score differences between the agent instances which trained a total of 200000 episodes and their instances which only trained 150000 episodes, highlighting which agents suffered more from catastrophic forgetting. We can see that the original environments for which each agent trained, for 150000 episodes, were where the agents suffered from heavier catastrophic forgetting. The single-task agents even obtained a small increase in performance on some of the original four environments, however insignificant enough to say that they learned them. To overcome catastrophic forgetting, we decided to augment the HybridNet-150000 with the EWC algorithm. We want to test if it is possible to sacrifice performance on Space Invaders in order to maintain performance on the other environments. The EWC requires the star values $\theta^*_{input}$ from the HybridNet-150000's input network, which were stored after training the HybridNet-150000 on all four environments, and the Fisher Information Matrix $F$, which tells, for each parameter $p \in \theta_{input}$, how much the input network's output would be likely to change, given a change to $p$. $F$ is created by computing the parameters' gradients on a large sample of multi-task datapoints (i.e., datapoints from all four environments). The HybridNet-150000 ran for 20 episodes, collecting a total of 720 datapoints from all four environments. From these datapoints, gradients with respect to each input network's parameter are computed and stored in $F$. We called this new agent, the combination of the HybridNet-150000+EWC augmentation, the Universal Game Player (UGP). In order to test our hypothesis we trained five instances of the UGP, each with a different EWC $\lambda$ hyperparameter value. The used $\lambda$ values were $\lambda$=0.5, $\lambda$=1.0, $\lambda$=10.0 and $\lambda$=50.0. The lower $\lambda$, the more the parameters $\theta^*_{input}$ will be changed in order to learn the new environment (and subsequently forgetting the old one). The HybridNet-200000 (which has no EWC) is therefore equivalent to an UGP with $\lambda$ = 0.00. Figure 17 displays the obtained learning curves. As expected, the higher the $\lambda$, the worst the learning curve. However, the same happens as with the first hypothesis - the differences are not much significant, and even though there is a visible downwards trend as the $\lambda$ increases, if we run each agent in exploitation mode for 100 episodes on each task, a discrepancy between the learning curves and performance is detected. We ran 100 episodes in exploitation mode in order to measured the Avg. Score / Ep. on the all environments, and catastrophic forgetting on the original four. Since all the UGP instances are forks from the HybridNet-150000 trained on 150000 episodes on all environments, we now compare both the HybridNet-200000 and the UGP instances with the HybridNet-150000. Table 5 displays the results after running each agent on each environment, in exploitation mode, for 100 episodes. As expected, we can see that as $\lambda$ increases, the less catastrophic forgetting occurs on the original four environments. Table 6 shows the the performance variations (in %), relative to the HybridNet-150000's performance and Fig. 18 plots the performance variations (in %) for the other four tasks, as the $\lambda$ value increases. Taking these results into account, we can see that as the $\lambda$ parameter increases, both the amount of catastrophic forgetting in the original four tasks and the performance on Space Invaders decrease (as expected). Even though there is a clear downwards trend for the Space Invader's performance (Fig. 18), three of the UGP instances were able to outperform the HybridNet-200000 ($\lambda$=0.5, $\lambda$=1 and $\lambda$=50). We were not expecting this to happen. It is plausible that these differences are given to stochastic events

that occur during training, since the training process is not deterministic. These differences are not that substantial, however, they once more contradict the results show in the learning process (as it happened with the first hypothesis), where the HybridNet obtained a better learning curve and the higher the $\lambda$, the lower the Avg. Score / Ep. at episode 200000. UGP $\lambda = 50$ was able to outperform the HybridNet-200000 on the older tasks, being able to maintain the best percentage of performance compared to all the other agents. It maintained 53.13% of performance on Assault, 65.01% of performance on Phoenix, 58.46% of performance on Carnival and 23.20% of performance on Demon Attack. UGP $\lambda = 100$ on the other hand, even though it wasn't able to outperform the HybridNet-200000 on the older tasks, was able to overcome a substantial amount of catastrophic forgetting. It was able to maintain 83.46% of performance on Assault, 89.86% of performance on Phoenix, 84.24% of performance on Carnival and 62.83% of performance on Demon Attack, while still getting a significant performance increase in Space Invaders (219.73%). We can therefore consider an interval for the $\lambda$ parameter, which provides a good trade-off between lost performance on Space Invaders and lost performance on the original environments, to be $\lambda \in [50, 100]$. Even though we do not have a trained UGP instance with $\lambda > 100$, it is logical that it would prevent even more catastrophic forgetting, while sacrificing even more performance on Space Invaders. We can conclude that by augmenting the Hybrid GA3C with the EWC, the resulting agent is able to overcome a substantial amount of catastrophic forgetting, while still being able to learn the new task very well. By setting the $\lambda$ hyper parameter to a value between 50 and 100, the resulting agent (UGP) is able to both perform similar on Space Invaders, when compared to its counterpart agent that did not have the EWC algorithm (the HybridNet-200000) and outperform it by not forgetting as much how to perform the old tasks.

## 7 CONCLUSION

We tested two novel hypothesis which had never been explored before. In the first hypothesis our results were somewhat inconclusive. On one hand, our proposed multi-task GA3C architecture, the HybridNet, was able to both achieve a slightly better performance on Space Invaders without training and a slightly better learning curve than the other single task GA3C agents. On the other hand, it obtained a similar policy to its counterpart single-task agents when performing 100 episodes for each task. This discrepancy was not explained by the use of the exploitation mode, since we ran all the agents for 100 episodes with the non-uniform action selection directly from the policy and still observed the same results. We think this difference may have to do with the fact that the learning curve differences are not that substantial in order to consider the HybridNet agent relatively better than the others. Nevertheless, taking the results into account, we argue that learning multiple tasks may allow for a more efficient training of a new task, but does not necessarily mean that it will provide an overall better policy in the end of training. In the second hypothesis, however, the obtained results provided a very strong conclusion. Our proposed multi-task architecture GA3C + EWC, the UGP, had the goal of remembering how to perform the original tasks by sacrificing as little performance as it could when learning the new task. The UGP, with a $\lambda$ parameter value set between 50 and 100, is not only able to achieve a similar performance in Space Invaders as the HybridNet, which has no EWC algorithm, but is also able to overcome a substantial amount of catastrophic forgetting.

## 7.1 Future Work

*7.1.1 Stronger Claim for the First Hypothesis.* One possible way to provide a stronger claim for the first hypothesis, could be to incorporate the evaluation in exploitation mode for the agents while they learn. By running all agents in exploitation mode, for 100 episodes, on all tasks, at a given interval when learning, it is plausible that the learning curve is more in agreement with the obtained performance results. For example, the agents could train for 10000 episodes, then run 100 episodes on each task, train again for additional 10000 episodes and run again for another 100 episodes. We did not follow this approach because it was highly expensive in terms of computation time when training the agents.

*7.1.2 More Platforms.* By implementing environment handlers, we open the possibilities to test the UGP with different OpenAI Gym platforms other than the Atari2600. The only requirement is that the platform provides multi-dimensional array of pixels, which can be resized and preprocessed as desired, and a reinforcement measure, which the OpenAI Gym always provides. Testing the UGP with 3D games would be an interesting approach. Even though we only used five total environments as tasks, all from the Atari2600 platform, we did not explore other platforms for two reasons. First, we already considered the multi-task learning problem with four tasks to be a difficult one. Birck et. al [5], when testing their Hybrid A3C only used two environments at a time, and we wanted to obtain a broader generalization by using more similar tasks, even if by doing so we introduce more differences. The second reason was that using environments from different platforms would not provide any advantages when testing any of the two hypothesis.

*7.1.3 Continuous Action Spaces.* The actor-critic model approximates a policy function $\pi$. A policy consists on a distribution that for each action $a \in A$, returns the probability of $a$ being the action which maximized the discounted cumulative reward $R$ for the input state. This works because have a discrete variable $a \in A$, and the output layer of the actor can contain $A$ output units, which pass through a softmax function in order to obtain the distribution. Another interesting approach would be to explore how well the model can be modified in order to handle continuous action spaces, where $A$ is not a discrete action space, but a continuous one. This could be done by approximating a Gaussian distribution $\mu$, where the actor output layer has two units, one for the mean and one for the variance. Once again, we did not follow this approach because it would not provide any advantages when testing any of the two hypothesis.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. 2016. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256* (2016).

[3] Petr Baudiš and Jean-loup Gailly. 2011. Pachi: State of the art open source Go program. In *Advances in computer games*. Springer, 24–38.

[4] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)* 47 (2013), 253–279.

[5] Marco Birck, Ulisses Corrêa, Pedro Ballester, Virginia Andersson, and Ricardo Araujo. 2017. Multi-Task reinforcement learning: An hybrid A3C domain approach. (01 2017).

[6] Bruno Bouzy and Bernard Helmstetter. 2004. Monte-carlo go developments. In *Advances in computer games*. Springer, 159–174.

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

[8] Rich Caruana. 1998. Multitask learning. In *Learning to learn*. Springer, 95–133.

[9] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-Carlo Tree Search: A New Framework for Game AI.. In *AIIDE*.

[10] Qiang Chen, Zheng Song, Jian Dong, Zhongyang Huang, Yang Hua, and Shuicheng Yan. 2015. Contextualizing object detection and classification. *IEEE transactions on pattern analysis and machine intelligence* 37, 1 (2015), 13–27.

[11] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. [n. d.]. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html. ([n. d.]).

[12] Ronald Kemker, Angelina Abitino, Marc McClure, and Christopher Kanan. 2017. Measuring Catastrophic Forgetting in Neural Networks. *arXiv preprint arXiv:1708.02072* (2017).

[13] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* (2017), 201611835.

[14] Nate Kohl and Peter Stone. 2004. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, Vol. 3. IEEE, 2619–2624.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[17] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. 2010. Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 253–256.

[18] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[19] Marcin Marszalek and Cordelia Schmid. 2007. Semantic hierarchies for visual object recognition. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 1–7.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. 1928–1937.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[23] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. 2006. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*. Springer, 363–372.

[24] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. 2014. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1717–1724.

[25] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[27] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[28] Gerald Tesauro. 1995. Temporal difference learning and TD-Gammon. *Commun. ACM* 38, 3 (1995), 58–68.

[29] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 5026–5033.

[30] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. 2000. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net* (2000).

## A APPENDIX

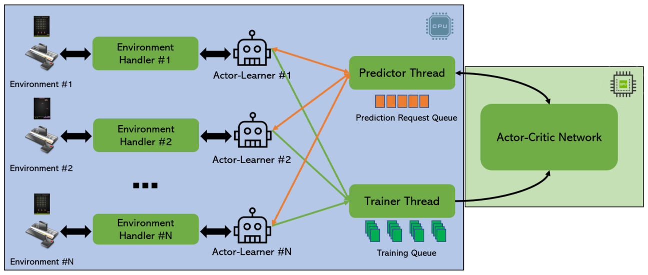

**Figure 1: The Universal Game Player. Architecture based upon the GA3C[2]. Several actor-learners interact asynchronously upon different instances of environments. They submit prediction requests to the global actor-critic network using Predictor threads, which return the policy for the given input state and feed recent experience batches to the global actor-critic network using Trainer threads, which in turn update the network's parameters.**

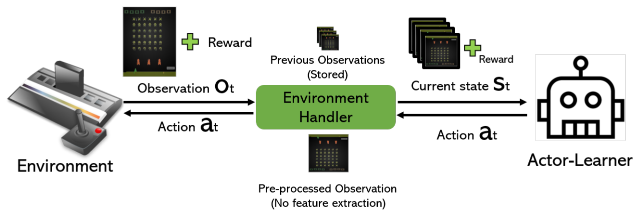

**Figure 2: The actor-learner interacting with its environment instance through an environment handler.**

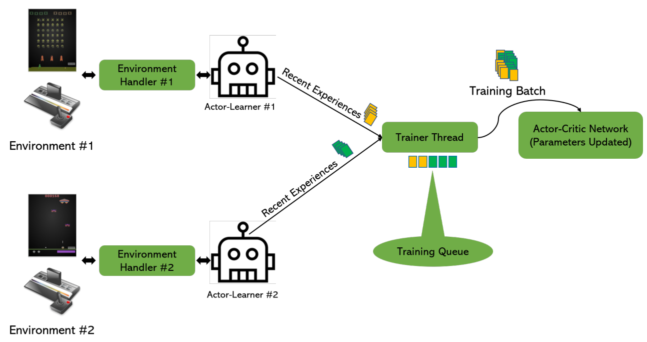

**Figure 3: The training process. Actor-learners place their recent experiences in a training queue, processed by Trainer threads, which stack them in a batch and then use them to update the actor-critic network.**

**Figure 4: The source tasks - Assault (Top Left), Phoenix (Top Right), Carnival (Bottom Left) and Demon Attack (Bottom Right). OpenAI Gym environments AssaultDeterministic-v4, PhoenixDeterministic-v4, CarnivalDeterministic-v4 and DemonAttackDeterministic-v4, respectively**

**Figure 5: The target task - Space Invaders. OpenAI Gym environment SpaceInvadersDeterministic-v4**

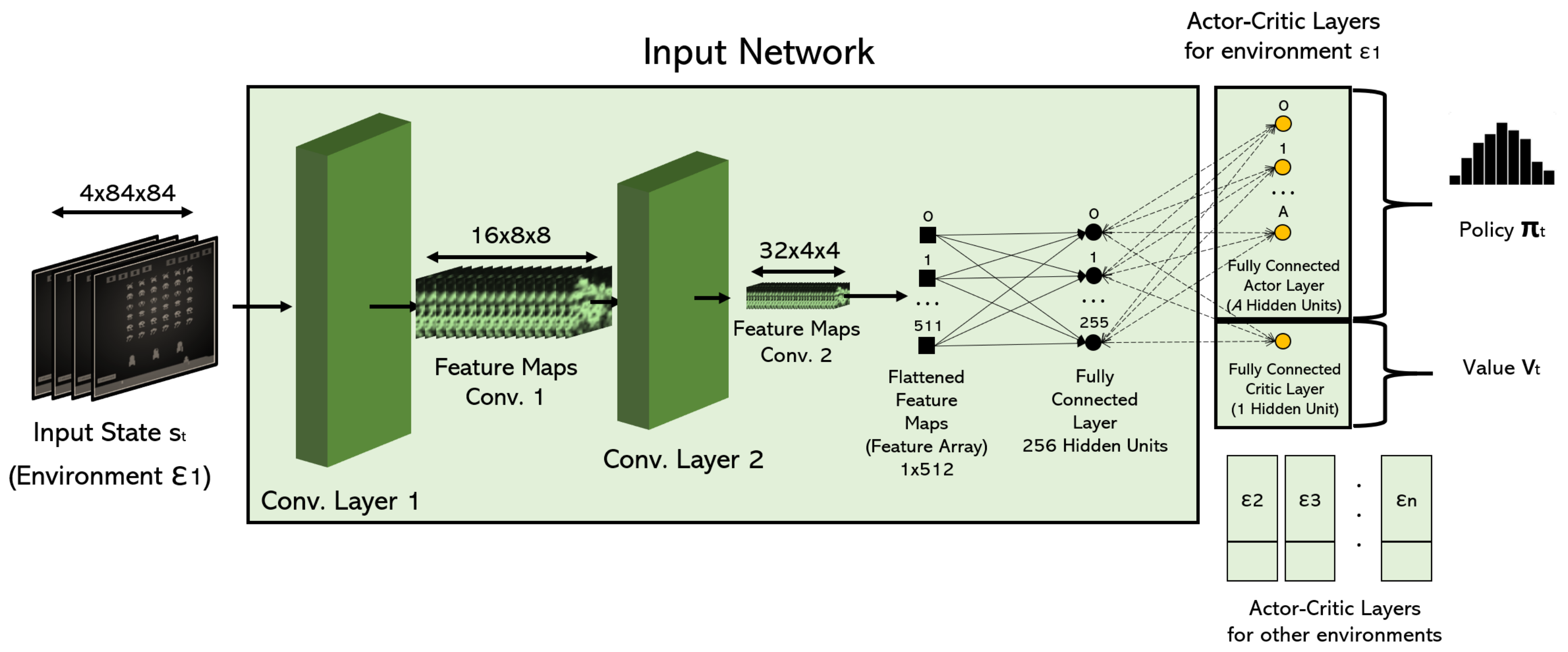
Input Network
Actor-Critic Layers for environment ε1
4x84x84
16x8x8
32x4x4
Input State st (Environment ε1)
Conv. Layer 1
Feature Maps Conv. 1
Conv. Layer 2
Feature Maps Conv. 2
0
1
...
511
Flattened Feature Maps (Feature Array) 1x512
0
1
...
255
Fully Connected Layer 256 Hidden Units
0
1
...
A
Fully Connected Actor Layer (A Hidden Units)
Fully Connected Critic Layer (1 Hidden Unit)
Policy πt
Value Vt
ε2
ε3
εn
Actor-Critic Layers for other environments

Figure 6: The actor-critic network. An input network with two convolutional layers and a fully connected layer. Shared between all environments. Followed by an actor network and a critic network. Both the actor and critic networks contain individual fully connected actor and critic layers (respectively), one per environment. The actor layers return a policy $\pi$ (softmax output with probabilities of for each action $a \in A$ maximizing the discounted cumulative reward for the given input state). The critic layers return a value $V$ (linear output value that tells how good it is for the agent to be in the input state).

| Hyper Parameter | Value Used |
|---|---|
| Learning Rate | 0.0003 |
| Discount Factor | 0.99 |
| Logarithm Noise | 0.000001 |
| Entropy Beta | 0.01 |
| Reward Minimum Clip | -1 |
| Reward Maximum Clip | 1 |
| Predictor Threads | 1 |
| Prediction Queue Max Size | 100 |
| Prediction Min Batch | 128 |
| Trainer Threads | 2 |
| T_Max | 5 |
| Training Queue Max Size | 100 |
| Training Min Batch | 128 |

**Table 1: Hyper parameters and configuration choices used when training the five agents for 150000 episodes.**

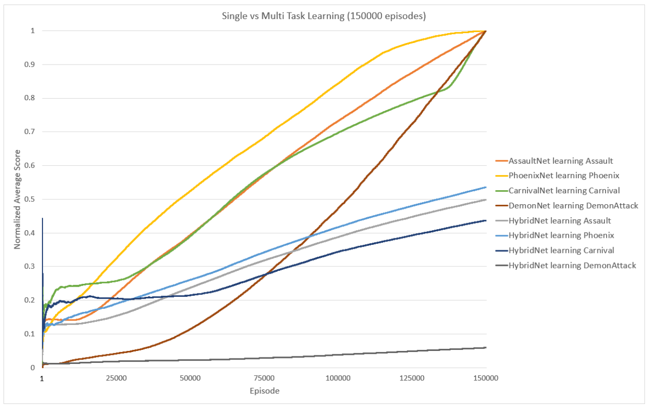

**Figure 7: Single vs Multi-task Learning. Learning curves for all five agents. Scores are normalized between 0 (lowest Avg Score / Ep.) and 1 (highest Avg Score / Ep.), relative to each environment's score space. The HybridNet has four separate learning curves, one for each environment.**

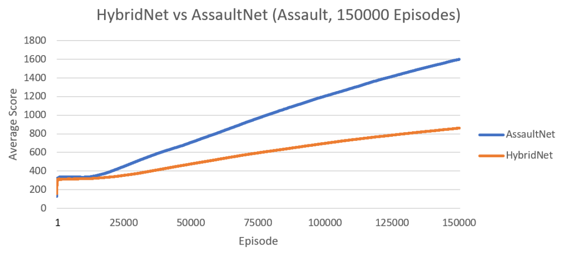

**Figure 8: AssaultNet and HybridNet learning Assault. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 860.60 points, while the AssaultNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 1599.50 points.**

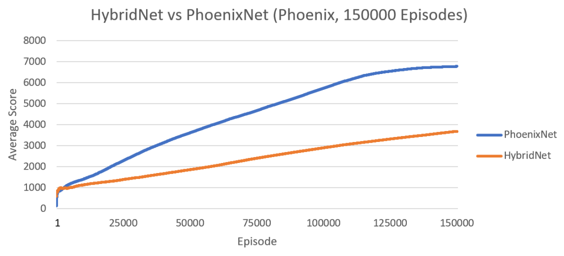

**Figure 9: PhoenixNet and HybridNet learning Phoenix. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 3685.87 points, while the PhoenixNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 6778.29 points.**

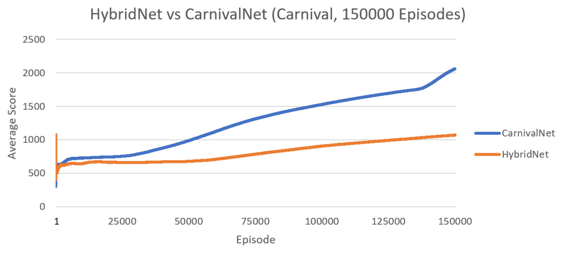

**Figure 10: CarnivalNet and HybridNet learning Carnival. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 1070.38 points, while the CarnivalNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 2063.18 points.**

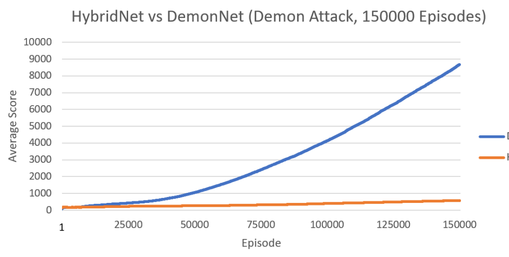

**Figure 11: DemonNet and HybridNet learning Demon Attack. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 577.66 points, while the DemonNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 8675 points.**

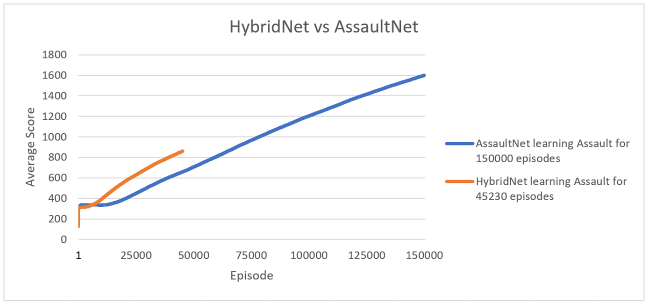

**Figure 12: AssaultNet and HybridNet learning Assault. With 45230 training episodes, AssaultNet was achieving an Avg Score / Ep. of 658.98 points and the HybridNet was achieving an Avg Score / Ep. of 860.60 points. The AssaultNet, however, kept training on Assault until 150000 episodes where it was able to achieve an Avg Score / Ep. of 1599.50 points.**

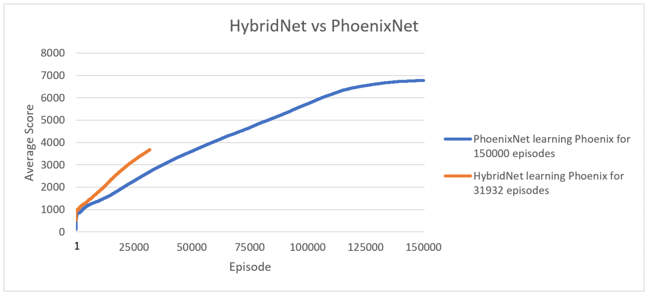

**Figure 13: PhoenixNet and HybridNet learning Phoenix. With 31932 training episodes, PhoenixNet was achieving an Avg Score / Ep. of 2701.61 and the HybridNet was achieving an Avg Score / Ep. of 3685.87. The PhoenixNet, however, kept training on Phoenix until 150000 episodes where it was able to achieve an Avg Score / Ep. of 6778.29 points.**

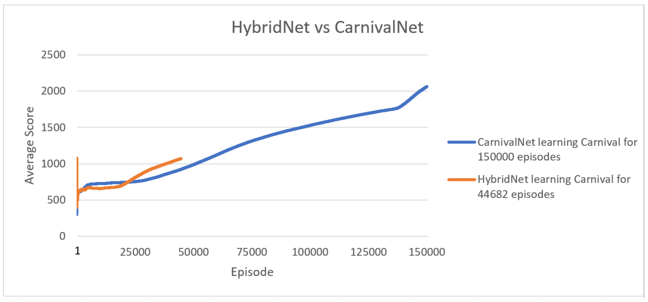

**Figure 14: CarnivalNet and HybridNet learning Carnival. With 44682 training episodes, CarnivalNet was achieving an Avg Score / Ep. of 923.57 and the HybridNet was achieving an Avg Score / Ep. of 1070.38. The CarnivalNet, however, kept training on Carnival until 150000 episodes where it was able to achieve an Avg Score / Ep. of 2063.18 points.**

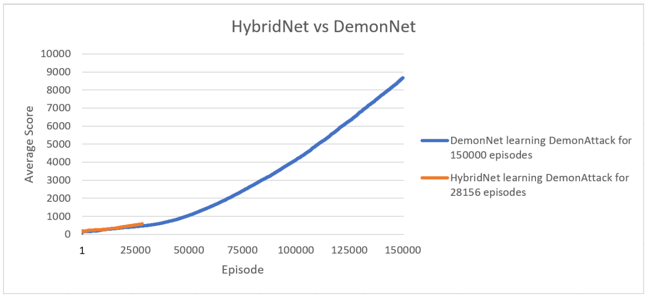

**Figure 15: DemonNet and HybridNet learning Demon Attack. With 28156 training episodes, DemonNet was achieving an Avg Score / Ep. of 464 and the HybridNet was achieving an Avg Score / Ep. of 577.66. points. The DemonNet, however, kept training on Demon Attack until 150000 episodes where it was able to achieve an Avg Score / Ep. of 8675 points.**

| | **Assault** | **Phoenix** | **Carnival** | **Demon Attack** | **Space Invaders** |
|---|---|---|---|---|---|
| **AssaultNet** | 4172.59 | 131.60 | 471.80 | 57.90 | 87.20 |
| **PhoenixNet** | 0.00 | 6867.50 | 446.20 | 0.00 | 0.00 |
| **CarnivalNet** | 217.56 | 879.70 | 5173.40 | 5.90 | 134.55 |
| **Demon Attack** | 134.40 | 270.40 | 0.00 | 20185.00 | 8.55 |
| **HybridNet** | 1349.38 | 5456.90 | 2039.20 | 1035.10 | 188.25 |

**Table 2: The five trained agents playing (in exploitation mode) for 100 episodes on each environment. Regarding the new environment, Space Invaders, the HybridNet was the agent which achieved the best Avg. Score / Ep. after 100 episodes, obtaining 188.25 points. As expected, the HybridNet is not as good as its single-task counterparts on their respective tasks, but is able to achieve fairly good results on each task, where the single-task agents are only able to achieve good results on their specific task.**

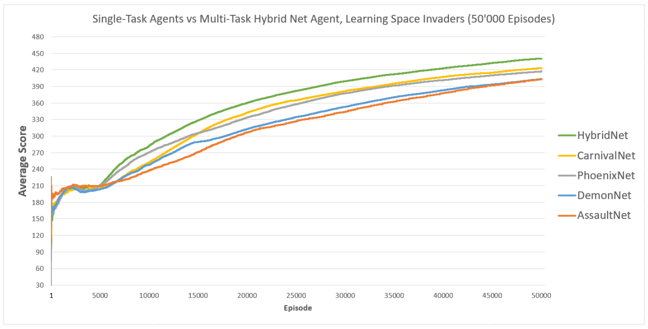

Figure 16: All five agents learning Space Invaders for 50000 additional episodes. After the 50000 episodes, the HybridNet was able to achieve an the Avg Score / Ep. of 440.44 points (the best), the CarnivalNet, an Avg Score / Ep. of 423.37 points (second best), the PhoenixNet, an Avg Score / Ep. of 417.05 points (third best), and the AssaultNet, an Avg Score / Ep. of 403.00 (the forth best).

| | Assault | Phoenix | Carnival | Demon Attack | Space Invaders |
|---|---|---|---|---|---|
| **AssaultNet** | 0.00 | 847.30 | 456.00 | 30.00 | 562.90 |
| **PhoenixNet** | 439.11 | 87.40 | 704.60 | 113.00 | 549.25 |
| **CarnivalNet** | 41.58 | 449.70 | 505.00 | 144.85 | 496.85 |
| **Demon Attack** | 604.20 | 597.60 | 630.60 | 162.55 | 536.30 |
| **HybridNet** | 284.34 | 217.10 | 33.80 | 67.55 | 495.20 |

Table 3: The five trained agents playing (exploitation mode) for 100 episodes on each environment, after training for 50000 additional episodes on Space Invaders.

| | Assault | Phoenix | Carnival | Demon Attack | Space Invaders |
|---|---|---|---|---|---|
| **AssaultNet-200000** | -4172.59 | 715.70 | -15.80 | -27.90 | 475.70 |
| **PhoenixNet-200000** | 439.11 | -6780.10 | 258.40 | 113.00 | 549.25 |
| **CarnivalNet-200000** | -175.98 | -430.00 | -4668.40 | 138.95 | 362.30 |
| **Demon Attack-200000** | 469.80 | 327.20 | 630.60 | -20022.45 | 527.75 |
| **HybridNet-200000** | -1065.04 | -5239.80 | -2005.40 | -967.55 | 306.95 |

Table 4: Avg. Score / Ep. differences for all agents, on all environments, after training 50000 additional episodes on Space Invaders. All agents suffer from catastrophic forgetting on their original tasks. Comparison between Tables 2 and 3.

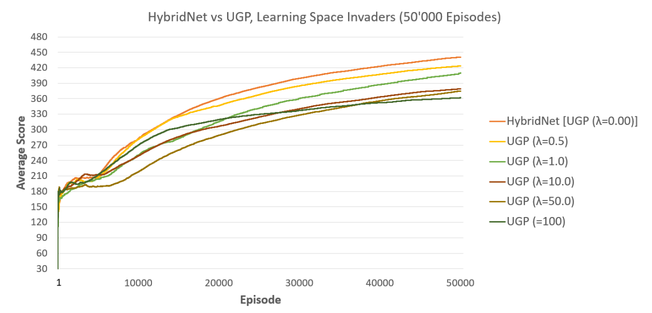

Figure 17: Space Invaders learning curves for the HybridNet-200000 and the UGP instances with different $\lambda$. The higher the $\lambda$ hyper parameter, the less altered the input network's parameters p will be in order to learn the new environment. Using a $\lambda$ of 0, 0.5 and 1 provides similar results. As expected, the higher the $\lambda$, the lower worst the learning curve.

| | Assault | Phoenix | Carnival | Demon Attack | Space Invaders |
|---|---|---|---|---|---|
| **HybridNet-150000** | 1349.38 | 5456.90 | 2039.20 | 1035.10 | 188.25 |
| **HybridNet-200000** | 284.34 | 217.10 | 33.80 | 67.55 | 495.20 |
| **UGP $\lambda$=0.5** | 85.05 | 622.50 | 439.20 | 0.00 | 522.90 |
| **UGP $\lambda$=1** | 265.02 | 679.70 | 236.00 | 168.60 | 576.10 |
| **UGP $\lambda$=10** | 95.55 | 311.40 | 144.80 | 200.00 | 455.35 |
| **UGP $\lambda$=50** | 716.92 | 3547.40 | 1192.20 | 240.10 | 519.40 |
| **UGP $\lambda$=100** | 1126.14 | 4903.70 | 1717.80 | 650.35 | 413.65 |

Table 5: Avg. Score / Ep. for the five agents, playing (exploitation mode) for 100 episodes on each environment. HybridNet-150000 is the agent trained for 150000 episodes on the four source environments. HybridNet-200000 is the HybridNet-150000's instance trained for 50000 additional episodes on Space Invaders. The UGP consists on an HybridNet+EWC, where different $\lambda$ values were experimented, trained. The HybridNet-200000, even though has no EWC algorithm implemented, can be seen as a EWC augmentation with $\lambda = 0$.

| | Assault | Phoenix | Carnival | Demon Attack | Space Invaders |
|---|---|---|---|---|---|
| **HybridNet-150000** | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| **HybridNet-200000** | 21.07% | 3.98% | 1.66% | 6.53% | 263.05% |
| **UGP $\lambda$=0.5** | 6.30% | 11.41% | 21.54% | 0.00% | 277.77% |
| **UGP $\lambda$=1** | 19.64% | 12.46% | 11.57% | 16.29% | 306.03% |
| **UGP $\lambda$=10** | 7.08% | 5.71% | 7.10% | 19.32% | 241.89% |
| **UGP $\lambda$=50** | 53.13% | 65.01% | 58.46% | 23.20% | 275.91% |
| **UGP $\lambda$=100** | 83.46% | 89.86% | 84.24% | 62.83% | 219.73% |

Table 6: Results from Table 5, but relative to the HybridNet-150000's score. Both UGP $\lambda$=50 and $\lambda$=100 were able to obtain a very good increase in Space Invaders (even though it wasn't the best) while also being able to remember how to perform the old tasks better than all the other agents.

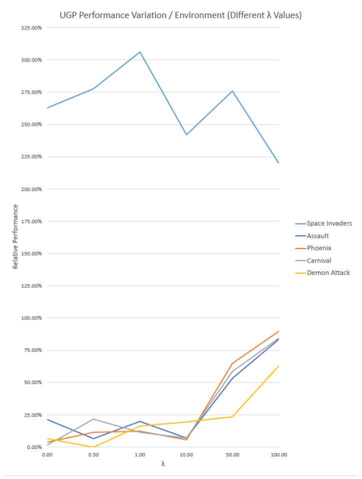

Figure 18: Performance variations for the original source tasks, given different lambda values. With $\lambda$=100, the UGP is able to overcome a substantial amount of catastrophic forgetting, still being able to achieve a relatively good performance on Space Invaders.