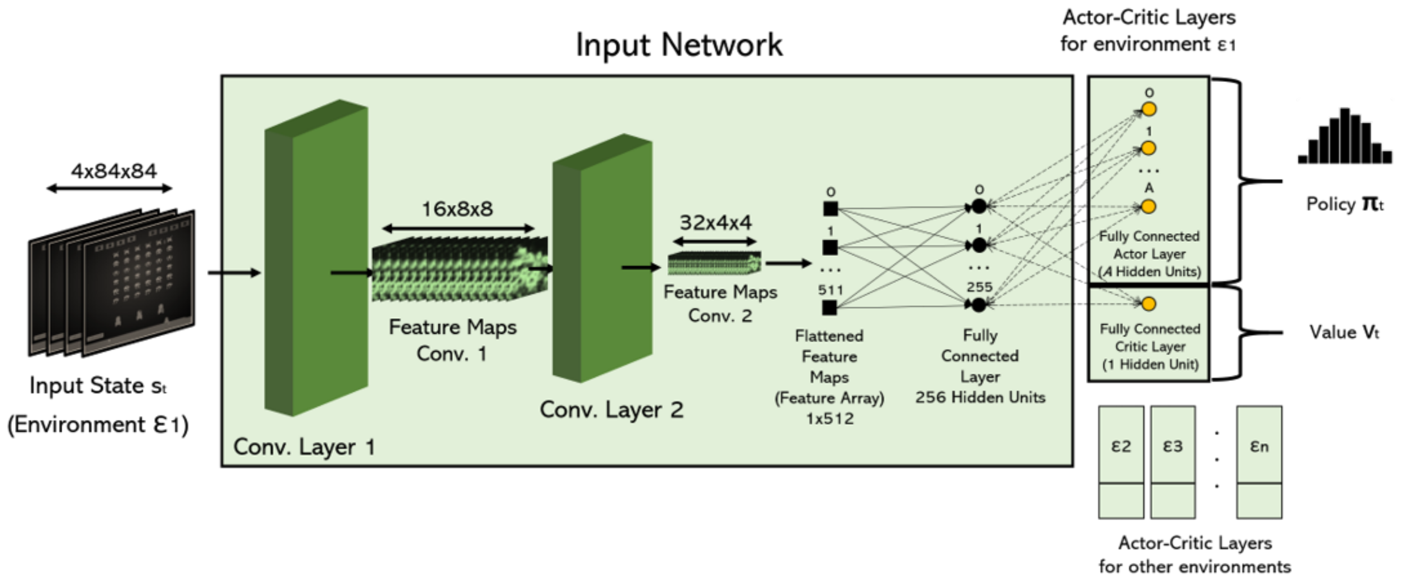
TÉCNICO LISBOA



Using Deep Learning to create a Universal Game Player

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

João Manuel Godinho Ribeiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Francisco António Chaves Saraiva de Melo
Prof. João Miguel De Sousa de Assis Dias

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. Francisco António Chaves Saraiva de Melo
Member of the Committee: Prof. Manuel Fernando Cabido Peres Lopes

October 2018

Acknowledgments

This dissertation is dedicated to my parents and family, who supported me not only throughout my academic career, but throughout all the difficulties I've encountered in life. To my two advisers, Francisco and João, for not only providing all the necessary support, patience and availability, throughout the entire process, but especially for this wonderful opportunity to work in the scientific area for which I am most passionate about. To my beautiful girlfriend Margarida, for always believing in me in times of great doubt and adversity, and without whom the entire process would have been much tougher. Thank you for your strength. Finally, to my friends and colleagues which accompanied me throughout the entire process and without whom life wouldn't have the same meaning. Thank you all for always being there for me.

Abstract

This dissertation introduces a general-purpose architecture that allows a learning agent to (i) transfer 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. The proposed architecture modifies the asynchronous advantage actor-critic, on GPU (GA3C, enabling multi-task learning and augments it with the Elastic Weight Consolidation algorithm, alleviating catastrophic forgetting. With our obtained agent, named the Universal Game Player (UGP), we show that by learning multiple tasks it is possible to improve the learning efficiency for a new one that is now required to learned 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; Neural Networks; Deep Reinforcement Learning; Multi-Task Learning; Transfer Learning; Catastrophic Forgetting;

Resumo

Nesta dissertação introduzimos uma nova arquitectura de agentes inteligentes que permite (i) transferir conhecimento de tarefas previamente aprendidas para uma nova tarefa à qual tem como objectivo aprender e (ii) lembrar-se de como exercer as tarefas previamente aprendidas enquanto aprende a nova. A arquitectura proposta modifica a recente arquitectura actor-crítico assíncrona, em GPU (GA3C), permitindo-a ser aplicada a aprendizagem multi-tarefa, e adiciona-lhe o algoritmo "Elastic Weight Consolidation" (EWC) de modo a aliviar o esquecimento catastrófico. Com o agente obtido, ao qual chamamos de "Universal Game Player" (UGP), mostramos que (i) aprender várias tarefas em simultâneo facilita a aprendizagem de uma nova tarefa, semelhante às anteriores e (ii) adicionando o algoritmo EWC ao GA3C, é possível evitar uma quantidade substancial de esquecimento catastrófico. Tornamos público todo o código fonte em https://github.com/jmribeiro/UGP.

Palavras Chave

Agentes Inteligentes; Redes Neuronais; Aprendizagem por Reforço Profunda; Aprendizagem Multi-Tarefa; Aprendizagem por Transferência; Esquecimento Catastrófico;

Contents

1 Introduction 1
1.1 The Problem . . . 3
1.2 Contribution . . . 5
1.3 Outline . . . 5
2 Background 7
2.1 Neural Networks . . . 9
2.2 Multi-Task Learning . . . 9
2.3 Catastrophic Forgetting in Neural Networks . . . 10
2.4 Deep Learning . . . 10
2.5 Convolutional Neural Networks . . . 11
2.6 Reinforcement Learning . . . 13
2.7 Tensorflow & Computational Graphs . . . 14
3 Related Work 17
3.1 Deep Reinforcement Learning . . . 19
3.1.1 The Deep Q-Network . . . 19
3.1.2 The Asynchronous Advantage Actor-Critic . . . 21
3.1.3 The Asynchronous Advantage Actor-Critic on a GPU . . . 23
3.1.4 AlphaGO . . . 24
3.2 Catastrophic Forgetting . . . 25
3.2.1 Elastic Weight Consolidation . . . 26
3.3 Transfer Learning . . . 26
3.3.1 Mid-level Feature Transfer . . . 27
3.3.2 Hybrid A3C . . . 28
3.4 Deep Reinforcement Learning Test Scenarios . . . 29
3.4.1 OpenAI Gym . . . 29
3.5 Discussion . . . 30

4 Solution 31
4.1 Requirements . . . 33
4.2 Approach . . . 33
4.3 Architecture . . . 34
4.3.1 The Environments . . . 35
4.3.2 The Actor-Learners and the Interaction Process . . . 35
4.3.3 The Prediction Process . . . 37
4.3.4 The Training Process . . . 38
4.3.5 The Actor-Critic Network . . . 41
4.3.5.A Loss Functions and Backpropagation . . . 43
4.3.5.B Elastic Weight Consolidation . . . 44

5 Evaluation 47
5.1 Approach . . . 49
5.2 The Environments as Tasks . . . 49
5.3 The First Hypothesis - Transfer Learning . . . 52
5.4 The Second Hypothesis - Catastrophic Forgetting . . . 60

6 Conclusion 65
6.1 Conclusions . . . 67
6.2 Future Work . . . 68
6.2.1 Stronger Claim for the First Hypothesis . . . 68
6.2.2 More Platforms . . . 68
6.2.3 Continuous Action Spaces . . . 68

List of Figures

1.1 Space Invaders (left) and Assault (right). Two games from the Atari2600 platform which share a lot similar features. 4

2.1 Filter trained to detect the player character feature in Space Invaders being activated. Since the trained filter has matched with the player character's location, there is an high activation that produces an output stored in the feature map. 12

2.2 Filter trained to detect the player character feature in Space Invaders not being activated. Since there is no player character at the filter's location, there will be a low activation or no activation at all. Nothing is stored in the feature map. 12

2.3 Computational graph for a simple dot product. 15

2.4 Feeding a computational graph with input and obtaining an output. 15

3.1 DeepMind's Deep Q-Network (DQN)'s Architecture - A CNN that approximates the Q-values for each action, when given a state represented by the screen's pixels. [1]. 20

3.2 DQN playing Atari Breakout, where it excelled with superhuman performance [1] 21

3.3 Asynchronous Advantage Actor-Critic (A3C)'s Architecture. Several independent instances here called "workers" run asynchronously on several independent instances of the task's environment. [2] 22

3.4 The game DOOM [2] 23

3.5 An overview of the Asynchronous Advantage Actor-Critic on a GPU (GA3C) architecture [3]. The GA3C contains a single global network. The actor-learners (agents) use the Predictor threads to retrieve predictions (policies) and the Trainer threads use the actor-learner's experiences to update the network. 24

3.6 The transfer learning process. Knowledge from source tasks is used in order to assist the learning process for a target task. From [4] 27

3.7 Transferring the internal layers (and their parameters) of a convolutional neural network trained for a source task, where data is abundant, into a target task, where data is scarce. The new network's output layers are adapted to fit the target task's output space. (From [5]) 28

4.1 The Universal Game Player. Architecture based upon the GA3C [3] (Fig. 3.5). 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. 34

4.2 The actor-learner interacting with its environment instance through an environment handler. 36

4.3 The prediction process. The actor-learner places a prediction request on a global prediction queue. The Predictor threads process the prediction requests, taking them from the queue (first in, first out) and stacking them in a batch. This batch is forward propagated through the actor-critic network, which returns the corresponding predictions for each single state in the batch as input. The Predictor threads, now holding the predictions for the given states, return them to the actor-learners. 38

4.4 The training process 40

4.5 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 π (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). 42

5.1 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 50

5.2 The target task - Space Invaders. OpenAI Gym environment SpaceInvadersDeterministic-v4 50

5.3 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. 52

5.4 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. 53

5.5 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. 53

5.6 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. 54

5.7 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. 54

5.8 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. . . . 55

5.9 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. 56

5.10 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. 56

5.11 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. . . 57

5.12 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). 58

5.13 Space Invaders learning curves for the HybridNet-200000 and the Universal Game Player (UGP) instances with different λ. The higher the λ hyper parameter, the less altered the input network's parameters p will be in order to learn the new environment. Using a λ of 0, 0.5 and 1 provides similar results. As expected, the higher the λ, the lower worst the learning curve. 61
5.14 Performance variations for the original source tasks, given different lambda values. With λ=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. 63

List of Tables

5.1 Hyper parameters and configuration choices used when training the five agents for 150000 episodes. 51

5.2 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. 57

5.3 The five trained agents playing (exploitation mode) for 100 episodes on each environment, after training for 50000 additional episodes on Space Invaders. 58

5.4 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 5.2 and 5.3. 60

5.5 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+Elastic Weight Consolidation Algorithm (EWC), where different λ values were experimented, trained. The HybridNet-200000, even though has no EWC algorithm implemented, can be seen as a EWC augmentation with $\lambda = 0$. 62

5.6 Results from Table 5.5, but relative to the HybridNet-150000's score. Both UGP λ=50 and λ=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. 62

List of Algorithms

1 Reward Accumulation 39

Acronyms

CPU Central Processing Unit

GPU Graphics Processing Unit

UGP Universal Game Player

MDP Markov Decision Process

DQN DeepMind's Deep Q-Network

A3C Asynchronous Advantage Actor-Critic

GA3C Asynchronous Advantage Actor-Critic on a GPU

EWC Elastic Weight Consolidation Algorithm

1

Introduction

Contents

1.1 The Problem 3
1.2 Contribution 5
1.3 Outline 5

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 [6–12]. Recent advances can be credited to the success of the so called deep learning techniques [13, 14]. 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 [15] (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 [1, 16], 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 [17,18], 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 [5]. 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 [19,20]). 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 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) [19], 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. [19], *"The ability to learn tasks in a sequential fashion is crucial to the development of artificial intelligence"*.

As a concrete example, consider the two following games - Space Invaders and Assault (Fig. 1.1), both from the Atari2600 platform [15].

Figure 1.1: Space Invaders (left) and Assault (right). Two games from the Atari2600 platform which share a lot similar features.

Even though these are two different games, their mechanics are very similar. Any human player that has mastered the game of Space Invaders can easily apply the same game play mechanics to Assault. Most existing agents, however, are not only unable to generalize the mechanics from both games, but also forget how to play Assault after mastering Space Invaders.

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 [19], 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 [21], build from the Atari2600 platform [15], 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.

1.3 Outline

This work is is organized as follows: Chapter 1 introduces the dissertation, exposes the hypothesis it tests and highlights its unique contribution.

Chapter 2 provides a background overview of neural networks, deep learning and reinforcement learning.

Chapter 3 provides both a n overview of the relevant available literature regarding work on deep reinforcement learning and how problems (such as catastrophic forgetting) are currently being tackled, whilst highlighting the main algorithms and techniques that laid the foundation for the creation of the UGP

Chapter 4 presents, describes and discusses the solution to the problem, in terms of architecture and functionality.

Chapter 5 lays an evaluation approach to testing both hypothesis, explains how the agent was concretely tested and presents the obtained results, providing an answer to both hypothesis.

Chapter 6 concludes the dissertation by providing an overall balance and highlighting possible enhancements that can be made in the future in order to improve the UGP.

2

Background

Contents

2.1 Neural Networks 9
2.2 Multi-Task Learning 9
2.3 Catastrophic Forgetting in Neural Networks 10
2.4 Deep Learning 10
2.5 Convolutional Neural Networks 11
2.6 Reinforcement Learning 13
2.7 Tensorflow & Computational Graphs 14

This section provides an overview of the required background knowledge related to the architecture and implementation of the introduced solution.

2.1 Neural Networks

Neural networks are machine learning models capable of learning both linear and non-linear functions. Unlike other machine learning models that could only learn linear functions (such linear regression), neural networks are capable of approximating non-linear functions in a high-dimensional feature space.

Architecturally, neural networks can be visualized as sequences of layers, where each layer contains units called neurons. Each neuron has a set of parameters that can be changed in order to fit the training data. The combined set of parameters from all the neurons in the network is usually denoted as θ.

Neural networks are traditionally used in a supervised learning context. In a supervised learning context, the networks are given several input-output pairs and by iterating over them several times, they are able to find patterns and correlations within the data. This approach is called supervised because the training data points are labeled, i.e., they contain the correct output for the given input.

The training process of a neural network in a supervised learning context consists of giving the input to the network, that then forward propagates it through its neurons and obtains an output value. The whole network can therefore be represented by a large composite function. The network's output is then compared with the correct output to compute the error. The error is then used in a Loss function $L(\theta)$. Training a network to fit a given data set has the objective of minimizing the Loss function $L(\theta)$. The Loss function $L(\theta)$ is minimized by adjusting the network parameters θ. Adjusting the network parameters θ in order to minimize a function $L(\theta)$ can be done by computing the gradient of $L(\theta)$ with respect to θ and using it to update θ.

2.2 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 [1, 16]. 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 [17]. 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 [19]. Since the required amount of memory space is proportional to the number of tasks, this approach impractical when learning large numbers of tasks [19]. 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.

2.3 Catastrophic Forgetting in Neural Networks

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 θ 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 θ_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 θ_a to θ_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.

2.4 Deep Learning

Most traditional machine learning models (e.g. neural networks), when given data as input, require these inputs to be described by values known as features. How to automatically learn good features from the data is a difficult problem in traditional machine learning. In order to achieve good results, traditional techniques heavily rely on the quality of these features. An agent's input must therefore have features that provided good internal representations of the environment in which it operates.

Instead of trying to figure out how an agent can process raw data and automatically find out which features are relevant and which are irrelevant, most developers simply design hand-crafted feature extractors that return, for the inputs, the features with their respective values. This is not only very task-specific (and therefore unpractical when one goal includes generalization as much as possible) but also requires expert knowledge (sometimes not available and always introducing some kind of human bias into the model, since what is considered an important feature is left to the developer's judgment).

These features must not only capture the task's relevant details but also be invariant to its irrelevant details. If the agent's input is to be abstracted from the current task, then the agent must be able to automatically learn which features are relevant to the current task's learning process. Different tasks may even share the same features, for example, different object recognition tasks in images. Examples of such irrelevant details in object recognition are the object's pose, scale, illumination, and other types of noise, in the image. Since these details do not represent any relevant information for the classification

process, the agent must learn how to ignore them.

Representation learning methods are machine-learning methods that allow a given agent to be directly fed raw input data (without passing it through a feature extractor). Representation learning methods automatically find out the data's features for the learning process.

Deep Learning methods consist of representation learning methods that contain several layers of internal representations. Each of these layers is responsible for extracting its own features. As information flows through each layer, they output their relevant information according to the their level of abstraction. From an architectural point of view, this is done by creating several modules and stacking them in an hierarchical manner. The depth of a deep learning model corresponds to the number of modules it has. Each module captures its own internal representation and has its own level of abstraction. Modules are stacked from the least abstract to the most abstract. The first modules contain a low level of abstraction since they deal with the raw input. As the information flows upwards through the modules, the level of abstraction in the representations themselves grows.

2.5 Convolutional Neural Networks

With the introduction of deep learning techniques, one technique that probably stands out the most is the feature extraction through convolutional neural networks [13, 14, 22]. Convolutional neural networks are neural networks which are able to learn how to automatically extract features from multi-dimensional arrays. Images (pixels) and sound (.wav files) are examples of high-dimensional ”raw” data. An RGB image is stored as a 3-dimensional array of pixels, and a stereo sound file is stored as a 2-dimensional array of sound samples. From these arrays, convolutional neural networks learn what is relevant for the task at hand and what is not. A convolutional layer receives as input a high-dimensional array, ex. image, and returns several 2-dimensional arrays called feature maps. Feature maps are produced by filtering the image, passing over it with 2-dimensional filters. Filters scan the image and will ”activate” whenever they are matched with a feature. Training a convolutional layer consists on updating the values on the filters, and they can be seen as weights being multiplied by the values on the input image (and added with a bias and passed through an activation function). Therefore training a neural network consists on updating the filter parameters in order to learn features and minimize the error for the given task, and the same minimization process from traditional neural networks can be used. Figure 2.1 provides an example of a filter, already trained to detect the player character in Space Invaders, activating when passing over the actual player character in the image. Figure 2.2 shows the same filter passing over a dark location on the image, and since there is no player character there, the filter will produce a low/no activating at all.

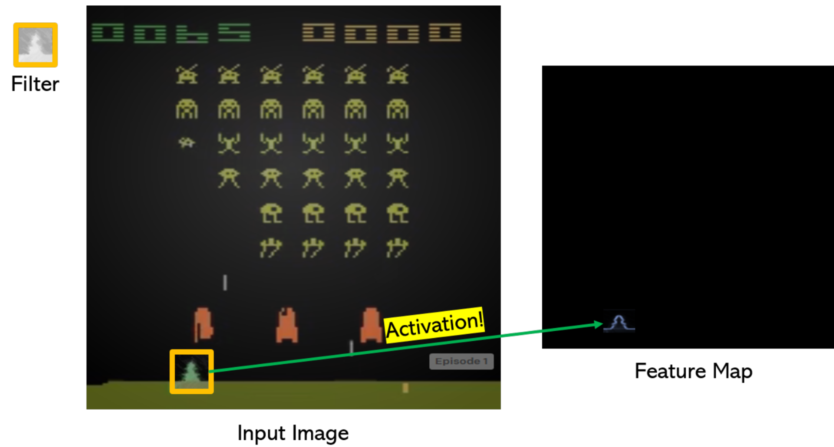


Figure 2.1: Filter trained to detect the player character feature in Space Invaders being activated. Since the trained filter has matched with the player character's location, there is an high activation that produces an output stored in the feature map.

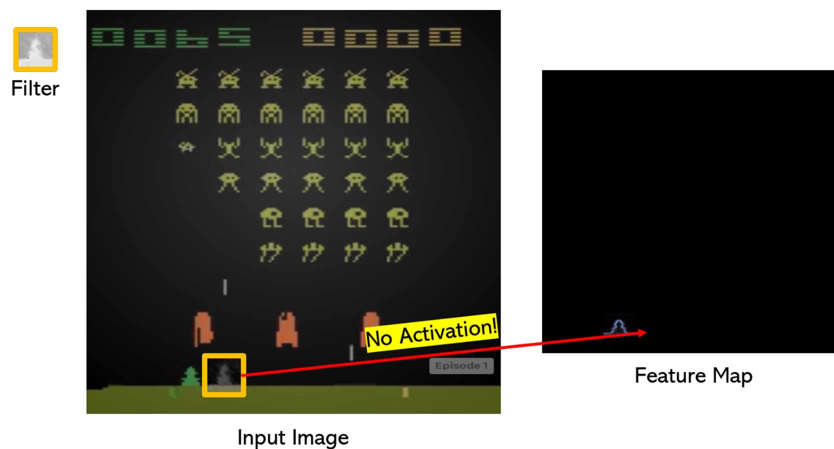


Figure 2.2: Filter trained to detect the player character feature in Space Invaders not being activated. Since there is no player character at the filter's location, there will be a low activation or no activation at all. Nothing is stored in the feature map.

These outputs are stored in the feature map, each responsible for storing the extracted features from the original image. This process is done resorting to the convolution operation (hence the name). Convolution is represented by the $*$ operator consists on the multiplication of each value on the 2-dimensional filter by the corresponding value on the input image filter location. All the values are then added with a bias, and the result passed through an activation function, like with traditional neural networks.

By giving stacking several convolutional layers on top of one another, we obtain a convolutional neural network. The output features maps produces by one convolutional layer can be given as input to another. The deeper a layer is within the network, the more abstract (and less concrete) the information contained within its feature maps. When arrays flow through several layers they increase in depth but

reduce their dimensions, so while the first layers produce few large feature maps, the last/deepest layer will produce a lots of small feature maps. The values from the last convolutional layer's feature maps are then flattened into a single 1-dimensional array. This ”feature array” is then given as input to a traditional neural network (Sec. 2.1). Features have been automatically extracted without human intervention.

2.6 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 α, interacts with an environment ϵ over several episodes. Each episode consists on a discrete sequence of time steps t, where ϵ can be in a given state s_t.

The control sequence over an episode is as it follows.

In each time step t:

1. α receives s_t from the set of states **S**.
2. α chooses an action a_t from the set of possible actions **A**. This decision is done according to the agent's policy π. 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**.
3. α executes a_t in s_t and ϵ transitions in time into the next state s_{t+1}.
4. α is 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 π, 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 α 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 α 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 θ. The parameter adjustment process in a reinforcement learning context is therefore the same as in a supervised learning context (Sec. 2.1).

2.7 Tensorflow & Computational Graphs

Tensorflow [23] is an open-source library created numerical computations. It allows the developer to setup a computational graph using an high-level API in Python (available in other languages), which is then built and compiled in optimized C++ or CUDA [24] code, which can both run on Central Processing Unit (CPU) and Graphics Processing Unit (GPU). Computational graphs are composed by operation nodes, where values flow through, being applied the respective operation. Figure 2.3 provides an example of a simple dot product, equivalent to a perceptron (neural network with one hidden unit) without the activation function, setup as a computational graph.

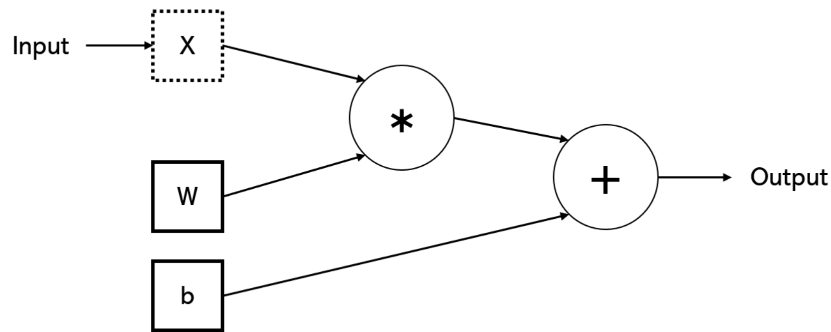


Figure 2.3: Computational graph for a simple dot product.

$\vec{X}$ is called a placeholder variable. When fed a certain input tensor (N-dimensional array), the values will be forward propagated to the first node it is connected to - the matrix multiplication node ($*$). $\vec{W}$ is called a variable which contains another tensor (N-dimensional array). Since it already contains values, it cannot be fed new values as input. However, these values can be changed. The tensors fed to $\vec{X}$ flow into the matrix multiplication node where each is multiplied by the tensor $\vec{W}$. The output values then flow into the addition node ($+$), where they are added with the value from another variable b. This result is then returned by the graph. Figure 2.4 provides an example of the workflow.

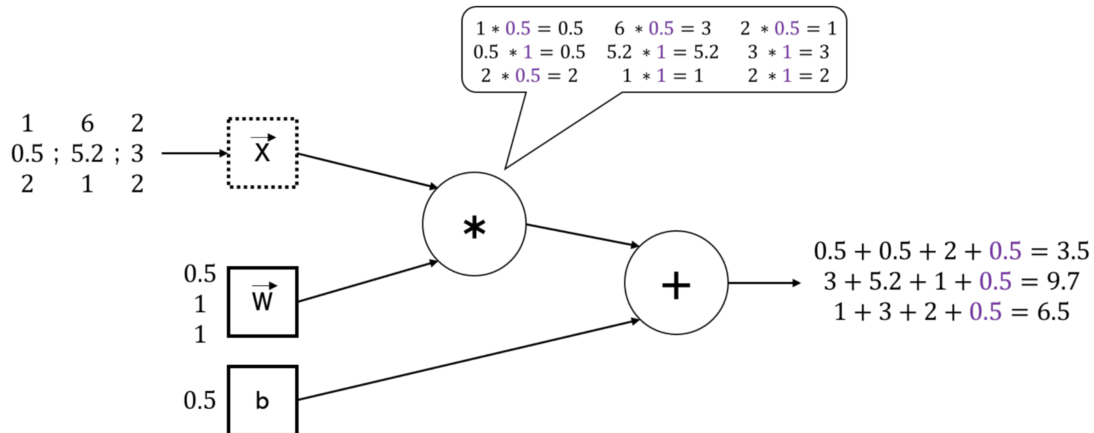


Figure 2.4: Feeding a computational graph with input and obtaining an output.

When minimizing a given Loss function L, an optimizer will update the variable values $\vec{W}$ and b in order to find a minimum for L. When computing the gradient for the update a given variable value p, in order to minimize the loss, the optimizer uses the chain rule, computing a single derivative at each node, until it reaches the node which depends directly on p. Now holding the gradient, different gradient descent algorithms [25] and rules can be used to update the new value for p.

One advantage of using computational libraries, such as Tensorflow, is that the GPU can be used for faster matrix multiplication, and different cores can even run part of the graph in parallel.

3

Related Work

Contents

3.1 Deep Reinforcement Learning 19
3.2 Catastrophic Forgetting 25
3.3 Transfer Learning 26
3.4 Deep Reinforcement Learning Test Scenarios 29
3.5 Discussion 30

To create the UGP, state-of-the-art deep reinforcement learning techniques were used, augmented with the necessary modifications to handle the agent's requirements (Sec. 1.2). The goal of this section is to provide both a global overview of the available literature regarding deep reinforcement learning and describe how problems such as catastrophic forgetting are currently being tackled, whilst highlighting the main algorithms that laid the foundation for the UGP.

3.1 Deep Reinforcement Learning

Deep reinforcement learning, as the name suggests, consists of the combination of reinforcement learning techniques (Sec. 2.6) with deep learning techniques (Sec. 2.4).

According to Arulkumaran et al. [26], a key problem with reinforcement learning techniques is the lack of scalability (meaning the agents could not be scaled up to a larger number of tasks). This lack of scalability stems from the way the reinforcement learning techniques take the task's input into account. The inputs were mostly task-specific and had low-dimensional feature representations that required manual extraction. An agent created for a single task could not take in data from another task as input.

With recent advances in deep learning [13], new techniques, such as convolutional neural networks (Sec. 2.5 [13, 14]) have proven to be able to extract features directly from high-dimensional sensorial input [1, 6, 14, 16, 22]. Automatic feature extraction alleviates the problem of lack of scalability. The agents are now able to be applied in a broader number of tasks in a reinforcement learning context. The only requirement for the tasks is that they need to provide an abstracted interface common between all of them. One example of tasks sharing the same interface are the video games from the Atari2600 platform [15], where all the games share the same screen pixels.

Existing work in deep reinforcement learning therefore provides the foundations for both the core concept and architecture of the UGP.

3.1.1 The Deep Q-Network

The DeepMind's Deep Q-Network (DQN) [1, 16] is, perhaps, one of the first notorious attempts on an general-purpose agent. Created to solve multiple tasks taking only high-dimensional sensorial input, the DQN is an agent that succeeded in playing video games from the Atari2600 platform [15], outperforming the best human players on a large set of its games.

The motivation for the DQN was, according to Mnih et al. [1], the creation of a single agent that could learn any given task from a varied range, and, in their words, *"a central goal of general artificial intelligence"*. However, unlike the UGP, it is not meant to learn on neither a multi-task or sequential learning context. The DQN learns one task (video game) at the time and when given a new video game from the Atari2600 platform, a new untrained instance of itself is then trained from scratch.

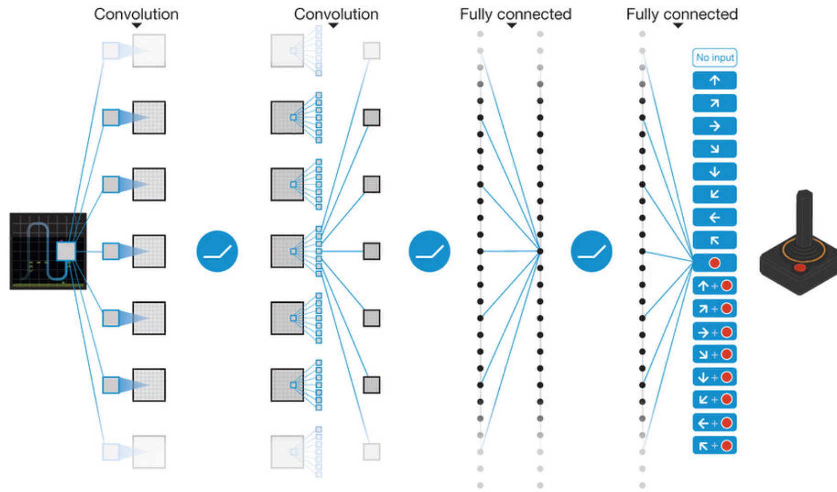


Figure 3.1: DQN's Architecture - A CNN that approximates the Q-values for each action, when given a state represented by the screen's pixels. [1].

Advances in Deep Learning techniques such as convolutional neural networks [13] made possible the extraction of features directly from high-dimensional sensorial input. Due to these advances, the DQN architecture consists on a deep CNN that automatically extracts features from high-dimensional sensorial input and learns a successful policy for each state by a process of trial-and-error. The DQN approximates the Q-value function in a reinforcement learning context (Fig. 3.1). In the Atari2600's reinforcement learning context, the game states, consisting of high-dimensional sensorial input, correspond to the Atari2600 platform's screen pixels (84x84 pixel images). Mnih et al. did, however, preprocess the raw screen pixels by downscaling them from 210x160 to 84x84 pixel images. The reason was that *"working directly with raw Atari2600 frames (...) can be demanding in terms of computation and memory requirements"* [1]. The game score was used on the DQN loss function as the reward value.

The DQN was tested in 49 games of the Atari2600 platform [15]. In 29 out of the 49 games, it obtained a performance equal or superior to human players, like in Atari Breakout, where it achieved a superhuman performance (Fig. 3.2).

In 43 out of the 49 games, it performed above the best reinforcement learning algorithms that where created for each one of the specific games. Given these results, we conclude that the DQN has proven capable of solving several problems by perceiving only a high-dimensional input.

It has, however, some limitations.

The first limitation, according to Lillicrap et al. [27], is that it *"can only handle discrete and low-dimensional action spaces"*. This means that the DQN cannot be applied to tasks with a large (discrete) or continuous action space. The reason for this problem is that in order to pick the action that is executed, the DQN requires, in each time step t, an iteration over the Q-values of each action, for a given state. Mnih et al. ended up developing a newer agent [28] capable of working in both continuous and discrete

Figure 3.2: DQN playing Atari Breakout, where it excelled with superhuman performance [1]

action spaces.

The second limitation, according to Roderick et al. [29] is the inability to succeed in domains with a long horizon and sparse rewards, something shown by the low performances on games such as Montezuma's Revenge and Venture. This problem is known as the temporal "credit assignment" [30]. These are domains where the agent does not obtain the rewards immediately after executing the action, therefore disrupting the learning process, since the action's consequences do not appear until later on in time.

The third limitation, according to Kirkpatrick et al. [19], is that when trained sequentially for two different games, it catastrophically forgets how to perform the first game. Kirkpatrick et al. [19] tackled this limitation by augmenting the DQN with their own algorithm.

The DQN primary goal is to show that an agent can learn a large number of tasks using high-dimensional sensorial input. With the UGP, we contribute with a novel agent that, much like DQN, is capable of learning any game using high-dimensional sensorial input, but also sharing knowledge between similar games, by learning them both sequential and asynchronous (parallel) contexts, and in the sequential context, continuously learning without forgetting previously learnt games.

3.1.2 The Asynchronous Advantage Actor-Critic

DeepMind's newer attempt at a general-purpose agent, the Asynchronous Advantage Actor-Critic (A3C) [28], proved to learn tasks much faster than the DQN. 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 π (the Actor), trained using policy gradient method, and a value function V (the Critic) in two separate fully-connected layers of the global network. It is the same method used with the UGP and we will therefore detail it in Chap. 4.

Figure 3.3 provides an overview of A3C's architecture.

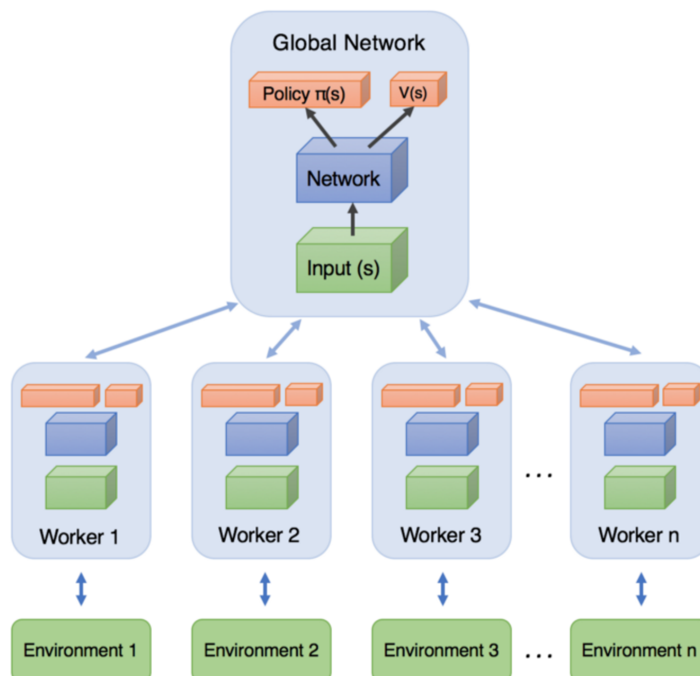


Figure 3.3: A3C's Architecture. Several independent instances here called "workers" run asynchronously on several independent instances of the task's environment. [2]

The A3C therefore consists on several actor-learners, each holding a local copy of a global neural network. Each actor-learner uses its independent copy of the network. After a fixed number of interactions, the agents use the local experience and calculate gradients. These gradients are then used to asynchronously update the global network. After another specified number of iterations, the now updated global network's parameters are copied into each local network.

The A3C, proved effective in tasks with both discrete and continuous action spaces (control tasks from the MuJoCo physics simulator [31], 2D games from the Atari2600 platform [15] and also 3D games such as TORCS 3D Car Racing [32] and Labyrinth [28]). 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.

In [2], the author's implementation learned how to play the tasks created from the game of DOOM,

for the VizDoom challenge (`http://vizdoom.cs.put.edu.pl/`) (Fig. 3.4). It is debatable, however, if DOOM can considered a 3D game (something relevant to the discussion of the A3C effectiveness on 3D games).

Figure 3.4: The game DOOM [2]

Given the requirements of the UGP, the base implementation of the A3C has a problem - it is designed so that all actor-learners act upon different instances of the same environment, which consists on single-task learning. There has already been a contribution [18] which augmented the A3C with multi-task learning capabilities. This was the same approach used with the UGP.

3.1.3 The Asynchronous Advantage Actor-Critic on a GPU

Mnih et al. [28] believe the success of A3C (3.1.2), 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 [3, 18]. One of them is the Asynchronous Advantage Actor-Critic on a GPU (GA3C). Proposed by Babaeizadeh et al. [3], the GA3C is a version of the A3C algorithm that uses the GPU in order to speedup training. It has the following differences:

- 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.
- 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.
- 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.

Figure 3.5 provides an overview of the GA3C's architecture.

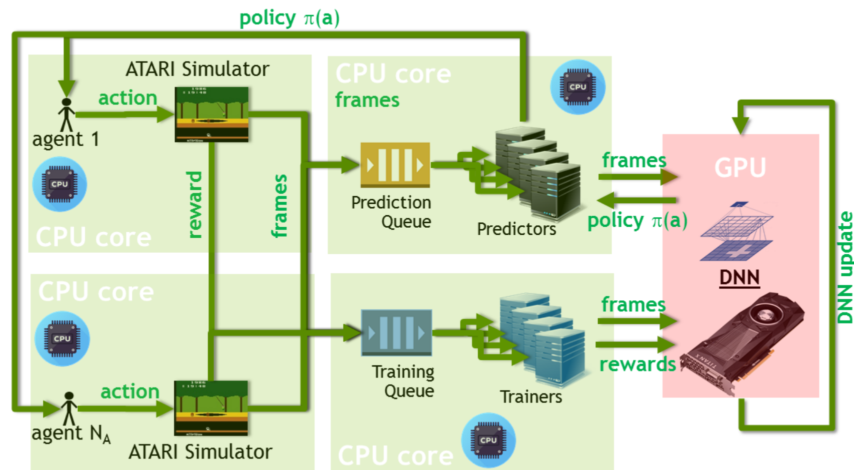


Figure 3.5: An overview of the GA3C architecture [3]. The GA3C contains a single global network. The actor-learners (agents) use the Predictor threads to retrieve predictions (policies) and the Trainer threads use the actor-learner's experiences to update the network.

The GA3C architecture served as the foundation for the UGP. Since the neither the A3C nor the GA3C were originally designed to handle multi-task learning, its architecture was modified so that the actor-learners could be interacting on different instances of different environments. This multi-task augmentation has also already been done recently by Birck et al. [18] with their agent, the Hybrid A3C (Sec. 3.3.2).

3.1.4 AlphaGO

AlphaGO [6] is an agent created with a single purpose, to play the game of GO at superhuman levels. It was considered a great accomplishment in the field of Artificial Intelligence, like its predecessor DeepBlue (IBM) [33].

In October 2015, it won by 5-0 against the European champion Fan Hui and more recently in March 2016, it won 4-1 against the World champion Lee Sedol.

AlphaGO also obtained a 99.8 win rate against the best agents created for the game of GO (ex. Pachi [11], an agent that according to Silver et al. [6] was considered *"a sophisticated Monte Carlo [12] search program"*).

Unlike previous agents that contributed to its creation [7, 10, 11], AlphaGO's training had three different stages. The first stage was to make AlphaGO play as well as an expert human player. A Policy Network $P_\sigma(a|s)$ that for each game board s returns the probability for each move a was trained, using a supervised learning approach with state-action pairs recorded from professional human players.

The second stage was to make AlphaGO play better than any human player. A Policy Network $P_\rho(a|s)$ similar to the $P_\sigma(a|s)$ was trained, this time using a reinforcement learning approach, by playing over and over against an instance of itself.

The third stage was to allow AlphaGO to efficiently evaluate a game board. A Value Network $V_\theta(s)$ that for each game board s returns the board's Value (the higher the value, the higher the probability of AlphaGO winning) was trained. This network was created to assist the already existing Policy Network $P_\rho(a|s)$ with the rewarding process, by quickly evaluating the states.

Even though considered one of the most successful deep reinforcement learning agents created for games, AlphaGO is an agent created with a single purpose, to excel at the game of GO. UGP on the other hand, is an agent required to be able to play several games. AlphaGO's input is done resorting to convolutional neural networks that provide automatic feature extraction from the game board and they are created to parse the GO game's board, which is a 19x19 matrix. This can be considered low-dimensional input, and means AlphaGO cannot be applied to other tasks with higher-dimensional sensorial input, like the DQN [1, 16].

3.2 Catastrophic Forgetting

In this section, we discuss the problem of catastrophic forgetting and some approaches that seek to address it. As explained in Sec. 2.3, catastrophic forgetting is a phenomenon that occurs when an agent learns two different tasks sequentially. This phenomenon can be viewed as the *"tendency for knowledge of the previously learned tasks to be abruptly lost as information relevant to the current task is incorporated"* [19].

Agents like DQN [1,16] or the A3C [28] have proven successful in solving deep reinforcement learning tasks. In the example of the DQN, it was created with the intention of being able to learn any given task, however it could only do so by learning one at the time. According to Kirkpatrick et al. [19] an agent's ability to *"learn tasks in a sequential fashion is crucial to the development of artificial intelligence"*. According to Kemker et al. [20], catastrophic forgetting is a problem that has not yet been solved. The UGP tackles the problem of catastrophic forgetting, by using existing techniques studied in this section.

3.2.1 Elastic Weight Consolidation

The EWC, introduced by Kirkpatrick et al. [19], is an algorithm that aims to prevent catastrophic forgetting in Neural Networks when an agent is sequentially trained for tasks.

Inspiration for this work came from the brain of mammals, where it was suggested that catastrophic forgetting was prevented by protecting acquired knowledge. *"When a mouse acquires a new skill, a proportion of excitatory synapses are strengthened"* [19].

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 $taskA$, with internal parameters $\theta = \theta_a^*$ that minimize $L_a(\theta)$, catastrophic forgetting occurs when the network trains for $taskB$ and, by trying to minimize $L_b(\theta)$, loses θ_a^* and obtains θ_b^*.

The EWC tackles this problem by using a Loss function that combines both $L_b(\theta)$ with θ_a^*, trying to protect θ_a^*. We will detail the process in Chap. 4.

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 `http://yann.lecun.com/exdb/mnist/` 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.

The EWC algorithm provides a good foundation for the UGP catastrophic forgetting avoidance. It does not, however, allow the transfer of knowledge between tasks. The UGP approaches this issue by learning several similar tasks simultaneously, where knowledge may be transferable between them.

3.3 Transfer Learning

Transfer Learning techniques are techniques which aim to assist an agent's learning process when learning a task, by resorting to the agent's already acquired knowledge on previously learnt tasks.

"(...) we may find that learning to recognize apples might help to recognize pears. Similarly, learning to play the electronic organ may help facilitate learning the piano." [4].

The motivation behind transfer learning approaches is to allow agents to learn a task where training data is scarce by transferring the already acquired knowledge from a similar task where training data is abundant [4]. Therefore, the goal of transferring learnt knowledge from a source task into a target task is to assist the target task's learning process [4] (Fig. 3.6). Unlike multi-task learning, transfer learning does not focus on learning the source and target tasks simultaneously. Transfer learning focuses more

on learning the target task by using already acquired knowledge from the source task.

Transfer learning is the most efficient way for a neural network to acquire experience [34], since it can resort to knowledge it has already acquired instead of needing to learn it *"from scratch for each task"*.

Transfer learning techniques are applicable when the tasks are, in some way related and have the same low-level features (e.g. two image classification tasks, where the input has the same low-level features, such as object corners and edges, but the classification labels are different).

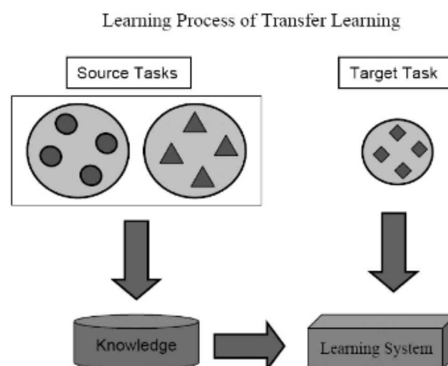


Figure 3.6: The transfer learning process. Knowledge from source tasks is used in order to assist the learning process for a target task. From [4]

3.3.1 Mid-level Feature Transfer

In [5], Oquab et al. propose a solution regarding the transfer of learnt knowledge in convolutional neural networks. According to the authors, the problem was the following.

Given a single convolutional neural network with more than 60 million parameters and a target task, where training data is scarce, *"directly learning so many parameters from only a few thousand training images is problematic"*. 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 (Fig. 3.7).

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 [35]). The dataset contains 1.2 million images and 1000 classes. Oquab et al. use the ImageNet [22] 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.

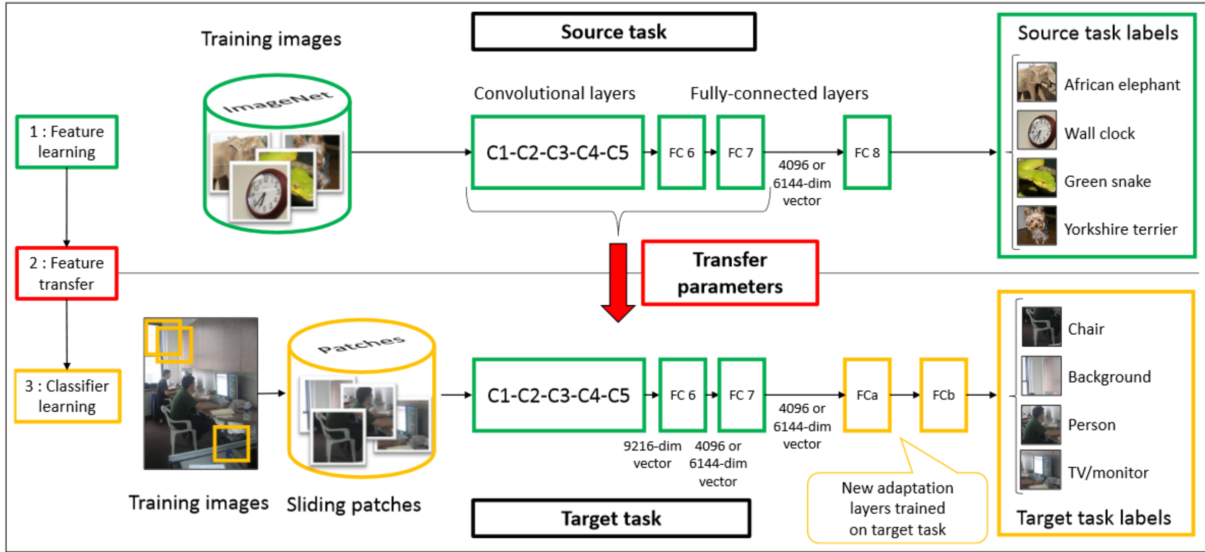


Figure 3.7: Transferring the internal layers (and their parameters) of a convolutional neural network trained for a source task, where data is abundant, into a target task, where data is scarce. The new network's output layers are adapted to fit the target task's output space. (From [5])

3. Train the convolutional neural network N_b for object recognition in images from the PASCAL Visual Object Classes Challenge 2007 (PASCAL VOC 2007 [36]). 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 [36] winners (INRIA [37]) by 18.3% and more recent approaches (NUS-PSL [38]) 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 (detailed in section 4).

3.3.2 Hybrid A3C

Birck et. al [18] wanted to test an hypothesis - Can the A3C handle multi-task learning? More specifically, does learning two different tasks at the same time enhance the training efficiency for both? Does it learn faster? Does it obtain an overall better policy? Since the A3C algorithm is targeted at single-task learning, its architecture was modified in order to handle multi-task learning. This modification was exactly the same we made to the GA3C's architecture [3], in order to create the UGP. It consists on giving each of the actor-learners different instances of different environments (instead of giving all of them different instances of the same environment).

The Hybrid A3C was tested on two different pairs of games from the Atari2600 platform [15], Pong

- Breakout and Space Invaders - Demon Attack, using the OpenAI Gym toolkit [21]. The pairs were chosen accordingly to the similarities between games. The testing procedure then went as following:

1. Train two instances of the A3C on each of the two games (single-task learning).
2. Train a single hybrid instance of the A3C on both games (multi-task learning).
3. Compare the learning curves over the same number of play episodes.

Both pairs test cases were successfull, i.e., the Hybrid A3C obtained a better policy than the A3C. However, in the first case (Pong - Breakout), it was at first performing worse in Pong than its single task learning counterpart (but better in Breakout). Later it converged into a better policy, outperforming both its single task learning counterparts. In the second test case (Space Invaders - Demon Attack), its policy was always performing better than both single task learning counterparts.

Even though the Hybrid A3C and the UGP tackle multi-task learning, they test different hypothesis. The Hybrid A3C tests if learning multiple tasks simultaneously improves the training for all of them, while with the UGP we test if learning multiple tasks simultaneously improves the training for a new, similar task, sequentially learned after them. A similar testing procedure was used in this work with the UGP, however with some different nuances, explained in both Sec. 4 and Sec. 5.

3.4 Deep Reinforcement Learning Test Scenarios

In order to test the scalability of general-purpose agents, test scenarios with a large number of tasks are required. The tasks in these scenarios should also provide a standardized interface, common among all tasks, and high-dimensional sensorial input for the agents. This was where video game platforms, such as the Atari2600 platform [15], OpenAI Gym Toolkit [21] or the more recent DeepMind Lab [1] have shown great popularity for testing newer methods [1, 3, 16, 18, 19, 27]. In the words of Arulkumaran et al. [26] *"as video games can vary greatly, but still present interesting and challenging objectives for humans, they provide an excellent testbed for RL agents"*. These virtual scenarios provide large number of tasks in controlled environments, that could be abstracted to high-dimensional sensorial input such as the game's screen pixels (non task-specific).

3.4.1 OpenAI Gym

OpenAI Gym Toolkit provides a collection of tasks described by virtual controlled scenarios, where all share a common interface. This allows for simple testing of agents, since the way they perceive these virtual environments is abstracted as simple "observations" and the rewards directly obtained from acting

[1] https://deepmind.com/blog/open-sourcing-deepmind-lab/

on a certain state. For most of the tasks (games), the observations consist on the game's screen pixels, as in with the Atari2600 platform. We will return in detail to the OpenAI Gym toolkit in both chapters 4 and 5.

3.5 Discussion

By taking all the existing work and relevant literature into account, we contribute with a novel agent which was created not only to 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 [3], algorithm as the foundation for the UGP, given the success of the algorithm it was itself based on, the A3C [28], 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 [18], 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

Solution

Contents

4.1 Requirements 33
4.2 Approach 33
4.3 Architecture 34

In this section we describe our solution to the problem, in terms of requirements, approach and architecture. We will describe our modified GA3C (Sec. 3.1.3) architecture, capable of handling multi-task learning, and the incorporation of the EWC algorithm within the actor-critic network's loss function. All the source code is publicly available at https://github.com/jmribeiro/UGP.

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.

4.2 Approach

We propose and test a novel solution, that combines elements from already existing work discussed in the related work section. These are the GA3C [3], the Hybrid A3C [18], the EWC [19] and the mid-level feature transfer approach for convolutional neural networks from [5]. We implemented our agent in Python 3 and setup the model using the Tensorflow [23] library, running all computational graph forward propagations and backpropagations using GPU.

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 [18]), it was possible to apply the GA3C to a multi-task learning context.

In order to tackle the knowledge transfer between similar tasks, the mid-level feature transfer learning approach from [5] 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.

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

4.3 Architecture

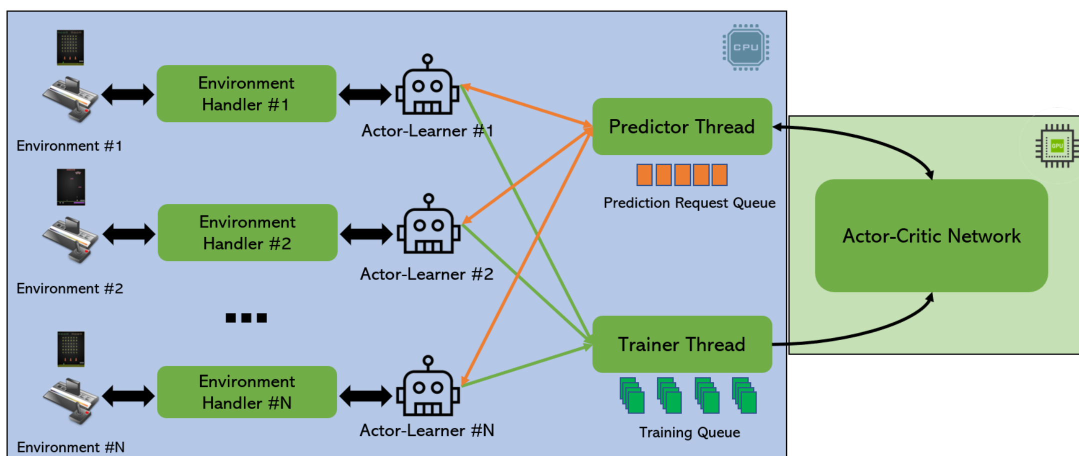


Figure 4.1: The Universal Game Player. Architecture based upon the GA3C [3] (Fig. 3.5). Several actor-learners interact asynchronously upon different instances of environments. They submit prediction requests to the global actor-critic network using Predictor threads, which return the policy for the 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.

The UGP consists on a version of the GA3C with some modifications. Figure 4.1 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.

4.3.1 The Environments

We resorted to the OpenAI gym toolkit [21], where available environments from the Atari2600 platform [15] were used as tasks. When interacting with an environment, the agent only has access to the game screen (pixels) and current score. When learning an environment, the agent's interaction can be split down into several episodes. An episode represented by the discrete-time sequence of states and executed actions. The episode can be discretized into timesteps t, that go from T_{start}, when the episode starts, until T_{end}, when the episode ends. The main goal for the agent is to execute actions upon the environment in a way that maximizes the average score per episode. The toolkit is provided in Python3 and its control sequence for a created virtual environment is as following.

First, the environment is reset. This provides the an observation. The observation consists the current game screen's pixels, i.e., a RGB image 210 pixels high and 160 pixels wide (260x160x3). Second, an action is executed on the environment, which then evolves, providing the next observation and the reward from executing the action on the environment's current state. By repeating this process until the environment returns the boolean value *done* as *True*, the environment has ran on a full episode.

Our actor-learners do not directly interact with the environment. They do so through an interface called an environment handler.

4.3.2 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. The interaction with the environment interaction is abstracted through an object called an environment handler. An overview of the interaction with its environment handler is described in Fig. 4.2.

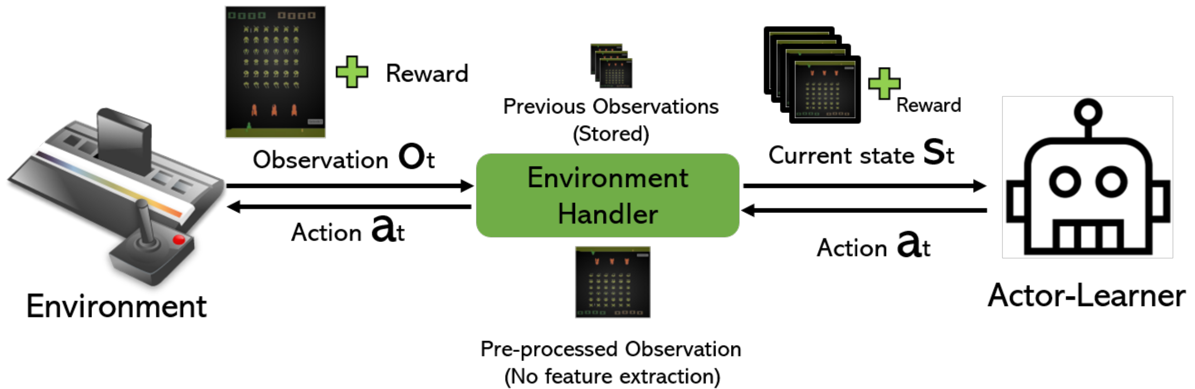


Figure 4.2: The actor-learner interacting with its environment instance through an environment handler.

In a given timestep t, the actor-learner starts by requesting the current state s_t and reward r_t from its environment handler. Before returning the current state s_t, the environment handler needs to build it. A state s_t consists on four stacked observations (frames) - o_t, o_{t-1}, o_{t-2} and o_{t-3} (the current one and the previous three). By stacking the current observation and its previous three it is possible to capture the temporal context of the current state of the game. This is required given that all environments used were history-dependent (i.e. the current observation may have different meaning depending on which was the last). A way to understand the concept of history-dependence in environments is to imagine a photograph of a ball thrown upon the air. Intuitively, there is no way to tell if the ball is rising or falling, however if we hold a previous photograph, which shows the ball a few decimeters below position of the ball on the current photograph, may allow us to infer that the ball is in fact rising through the air.

The environment handler first fetches the current game screen, the observation o_t, with dimensions 210x160x3 (for the Atari2600). The observation o_t is now resized into a grey-scale 84x84 image (same pre-processing method used by Babaeizadeh et al. with the GA3C [3], which was based on Mnih et al. pre-processing method with the DQN). It is also very important to emphasize that no feature extraction is made during this step. All the pre-processed observations still represent high-dimensional sensorial input. The environment handler therefore had to keep, in a memory queue, the previous three pre-processed observations - o_{t-1}, o_{t-2} and o_{t-3}. The environment handler now stacks them with the new observation o_t, creating a 4-dimensional tensor with shape 4x84x84 which corresponds to the current state s_t and sends it to the actor-learner.

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.

4.3.3 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 prediction request is a tuple which contains their own unique id, the name of the environment they're acting upon, and the current state s_t. Figure 4.3 provides an overview of the prediction process. At each timestep, each actor-learner places a single prediction request on a global prediction queue. The size of this prediction queue can be specified in the configuration file. We use a maximum size of one hundred prediction requests, the same used by Babaeizadeh et al. with the GA3C [3].

The Predictor threads now process the prediction requests, taking them from the queue (first in, first out) 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 [3], 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 [3]. 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 (process already explained in above in 4.3.2). 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.

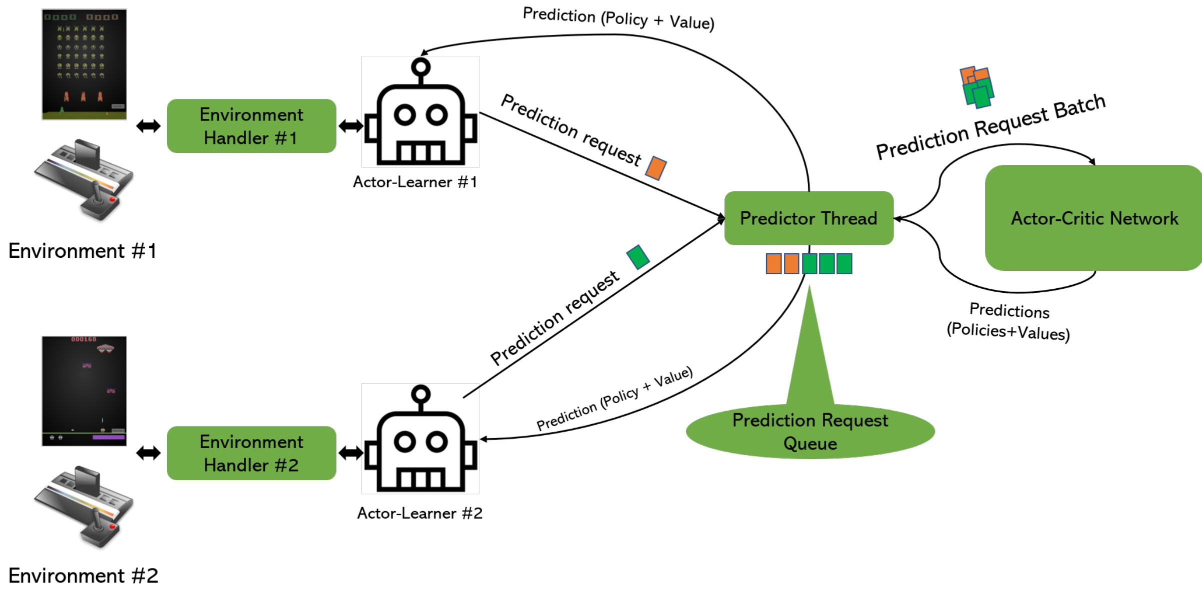


Figure 4.3: The prediction process. The actor-learner places a prediction request on a global prediction queue. The Predictor threads process the prediction requests, taking them from the queue (first in, first out) and stacking them in a batch. This batch is forward propagated through the actor-critic network, which returns the corresponding predictions for each single state in the batch as input. The Predictor threads, now holding the predictions for the given states, return them to the actor-learners.

4.3.4 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 [3]. 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 fundamentally the same used by Babaeizadeh et al. with the GA3C [3] and Mnih et al. with the A3C [28]. The algorithm is described in Algorithm 1.

Algorithm 1 Reward Accumulation

Input: batch - The recent experiences batch (datapoints from T_{start} to T_{end})
Input: γ - The discount factor
Output: batch - The recent experiences batch prepared for training
Get $d_{T_{end}}$ from $batch$
Get $s_{T_{end}}$ from $d_{T_{end}}$
$R = \begin{cases} 0, & \text{if } s_{T_{end}+1} \text{ is terminal} \\ V(s_{T_{end}}), & \text{otherwise} \end{cases}$
for $t \in reversed(T_{start}, T_{end})$ **do**
Get d_t from $batch$
Get r_t from d_t
$r_t = \begin{cases} -1, & \text{if } r_t \leq -1 \\ 1, & \text{if } r_t \geq 1 \\ r_t, & \text{if } -1 < r_t < 1 \end{cases}$
$R \leftarrow r_t + \gamma R$
$r_t \leftarrow R$
end for
return $batch$

The actor-learner starts by fetching the last datapoint $d_{T_{end}}$ from the recent experiences batch. $d_{T_{end}}$ contains the current state ($s_{T_{end}}$), the executed action ($a_{T_{end}}$), the output policy ($\pi(s_{T_{end}})$), the output value for the state ($V(s_{T_{end}})$), the obtained reward from executing $a_{T_{end}}$ on $s_{T_{end}}$ ($r_{T_{end}}$), and a boolean telling if the state $s_{T_{end}+1}$ is a terminal state or not (i.e., if executing a_t on s_t terminated the episode. If $s_{T_{end}+1}$ is a terminal state, the reward accumulator R is initialized with the value 0. If $s_{T_{end}+1}$ is not a terminal state, it is initialized with the value $V(s_{T_{end}})$. Then, for every timestep t in the batch (iterating in reverse order, from T_{end} to T_{start}) the reward for the timestep, r_t, is first clipped between -1 and 1, and then overwritten by the accumulated value $r_t + \gamma R$, where γ is the discount factor hyper parameter. We used the same discount factor as Babaeizadeh et al. [3], 0.99. After running through every timestep in the batch, 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. Figure 4.4 provides an overview of the training process.

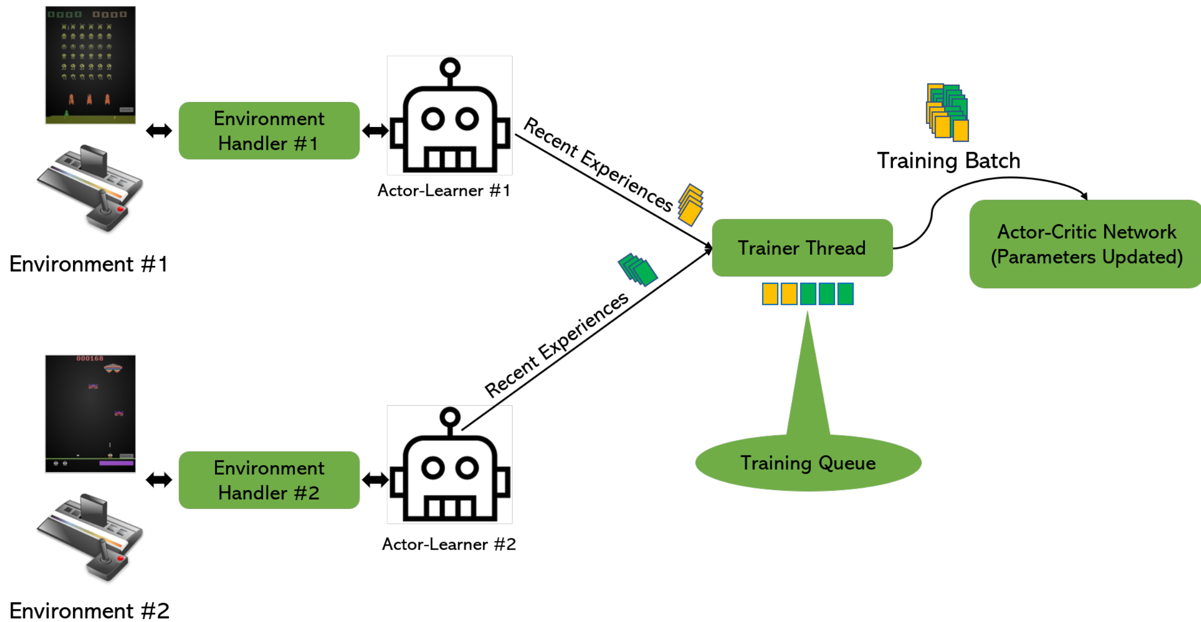


Figure 4.4: The training process

This training queue is processed by the trainer threads in first in, first out order. 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 [18]. When it comes to the backpropagation, Birck et al. [18] decided to change the optimizer from the RMSProp, used in the A3C, to Adam. With the UGP we sticked with the RMSProp optimizer, because it had also been used in the GA3C by Babaeizadeh et al. [3] and proven to work really well for single-task learning. This work will not go into detail on how both the Adam and the RMSProp optimizers work. For more information on gradient descent algorithms check Ruder's paper [25] and Geoff Hinton's Coursera Class Lecture 6e [1], both providing an overview on gradient descent algorithms. 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 [3].

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

4.3.5 The Actor-Critic Network

The actor-critic network is a deep convolutional neural network (Sec. 2.5) that follows an actor-critic approach. 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 [5] 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 [5]). An overview of the network's architecture can be seen in Fig. 4.5.

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 π 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 π, a method known as policy-gradient, from which the loss function is derived [27].

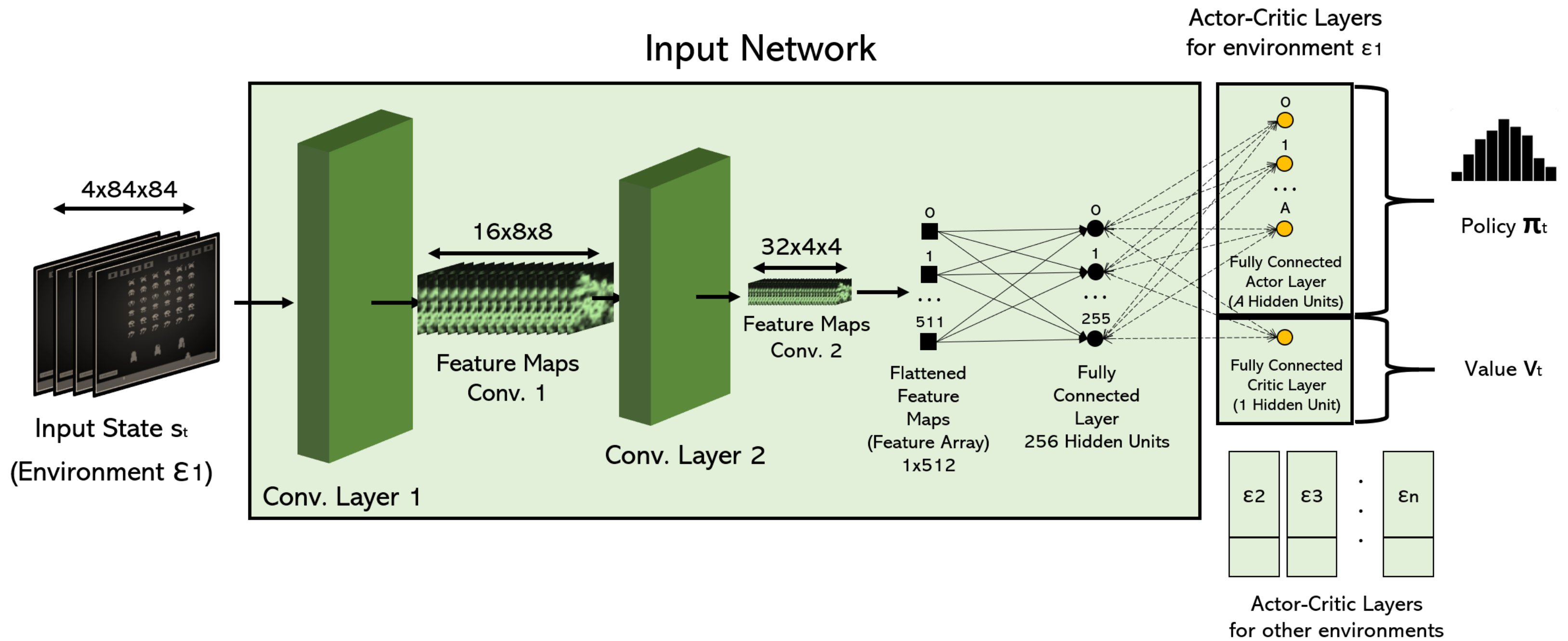


Figure 4.5: The actor-critic network. An input network with two convolutional layers and a fully connected layer. Shared between all environments. Followed by an actor network and a critic network. Both the actor and critic networks contain individual fully connected actor and critic layers (respectively), one per environment. The actor layers return a policy π (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).

4.3.5.A 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 ϵ a loss function $L_\epsilon(\theta)$ is setup. This loss function is given to the RMSProp algorithm [25], which 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, one 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 - θ_{input}, θ_π and θ_V. The first set, θ_{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 θ_π, represents the parameters from the actor layer of an environment, and are only updated by backpropagating datapoints from that specific environment. The third set θ_V represents the parameters from the critic layer of an environment, and like θ_π, 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 θ_π and θ_V, and then shared parameters θ_{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 θ_{input} and θ_π are updated. When minimizing $L_V(\theta)$, the parameters θ_{input} and θ_V are updated. Like with the GA3C [3], these two loss functions are first setup and then combined into a single one - $L_\epsilon(\theta)$.

- $L_\pi(\theta) = -log(\pi(a|s))Adv(s) + \beta H(\pi(a|s))$
- $L_V(\theta) = Adv(s)^2$

where:

- π is the actor, approximated by the actor layer's output (softmax).
- V is the critic, approximated by the critic layer's output.
- $Adv(s)$ is the advantage function approximator. The advantage function approximates the difference between the Q-values for each action for that state (which tell how good it is to execute each action on that state) and the actual value for the state (that tells how good it is to be on that state). Since the actor-critic method does not approximate or compute Q-values, the $Adv(s)$ is given by $R_s - V(s)$, where R is the discounted cumulative reward, computed for each state by the accumulation algorithm (Sec. 4.3.4).

- $H(\pi(a|s))$ is the entropy, a measure that tells how scattered the probabilities of each action maximizing the return for a state are. It was also used by Babaeizadeh et al. [3] in the GA3C, so we decided to keep it. The higher the entropy, the closer to each other these probabilities are. Higher entropy means higher uncertainty when choosing an action (higher chaos). A lower entropy means higher certainty when choosing an action (higher order). By introducing a measure of entropy in the loss function, for backpropagation, where we have access to both the probabilities and the chosen action, we introduce a penalty to exploitation. This encourages exploration, preventing the agent from getting stuck in local minima. Otherwise the first best action, with the higher probability of being chosen, when being chosen would be incrementally encouraged due to its reward, when in fact it could not be much different (or even be worst) than the second best. Entropy is therefore given by $-\sum_{a \in A} \pi(a|s) log(\pi(a|s))$.

The critic loss function could be shared between environments, since neither the discounted cumulative reward R and value V from a datapoint are environment specific. However, according to the mid-level feature transfer approach [5], the output layers should be different, in order to allow for different parameters to be updated. 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 $L_\epsilon(\theta)$ is:

- $L_\epsilon(\theta) = L_\pi(\theta) + 0.5 L_V(\theta)$

Like Babaeizadeh et al. with the GA3C [3] and Mnih et al. with the A3C [28], 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.

When simultaneously learning multiple environments, ϵ_1, ϵ_2, ... ϵ_n, several loss functions $L_{\epsilon_1}(\theta)$, $L_{\epsilon_2}(\theta)$, ..., $L_{\epsilon_n}(\theta)$ are setup. After minimizing them all simultaneously using datapoints from all the environments, the input network's shared parameters θ_{input} are set to a certain value which maximizes performance, θ^*_{input}. Consider that now, when learning a new environment, ϵ_{n+1}, sequentially after learning ϵ_1, ϵ_2, ... ϵ_n, the input network's parameters will be updated from θ^*_{input} into θ^{**}_{input}, in a way that maximizes performance for ϵ_{n+1}. The input network suffered from catastrophic forgetting (Sec. 2.3).

4.3.5.B Elastic Weight Consolidation

We conclude this chapter by introducing the EWC algorithm into our modified Hybrid GA3C architecture. The EWC algorithm [19], introduced in Sec. 3.2.1 is an algorithm that modifies the loss function of a neural network in order to prevent catastrophic forgetting.

When learning a task T_a, the network has to minimize a loss function $L_a(\theta)$. After training for T_a, the set of parameters θ converges to some value θ_a, which hopefully provide good performance for T_a. The problem occurs when the network, after learning T_a, has to sequentially learn a new task T_b, with a different loss function $L_b(\theta)$. After training for T_b, trying to minimize $L_b(\theta)$, the parameters θ will shift from θ_a into θ_b, learning the new task T_b and forgetting the old task T_a. The EWC algorithm tackles this problem by creating a new loss function, which has the goal of minimizing $L_b(\theta)$ while not losing θ_a.

As detailed above (Sec. 4.3.5.A), an input network that has learned multiple environments (our tasks) ϵ_1, ϵ_2, ... ϵ_n and obtained parameters θ^*_{input}, if given a new environment ϵ_{n+1} to learn, then with training, the input network's parameters will be updated from θ^*_{input} into θ^{**}_{input}, 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, and it is as following:

1. Store the optimal parameters θ^*_{input}.

2. Using a sample states $s \in S$ collected from a run on all environments ϵ_1, ϵ_2, ... ϵ_n, and the optimal parameters θ^*_{input}, the fisher information matrix F is computed. F tells, for each parameter $p \in \theta^*_{input}$, how much the output would change, given a modification to p. F is obtained by computing the gradients for each parameter $p \in \theta^*_{input}$, forward propagating each states s in the sample batch S through the input network. Even though we run the actor π, we compute only the gradients with respect to each parameter $p \in \theta^*_{input}$:

$$F_{(p)} = \sum_{s \in S} [\frac{\partial}{\partial p} \log(\pi(s))]^2$$

   And then divide each parameter entry $F_{(p)}$ by the total number of samples $N = len(S)$. For more information regarding the Fisher information matrix F, we recommend Ly et. al's work [39].

3. Having θ_{input} (current values), θ^*_{input} (stored optimal values) and F, $L_{\epsilon_{n+1}}(\theta)$ is modified so that:

$$L_{\epsilon_{n+1}}(\theta) = L_{\epsilon_{n+1}}(\theta) + \sum_{p \in \theta_{input}} [\frac{\lambda}{2} F_{(p)} (\theta_{input(p)} - \theta^*_{input(p)})^2]$$

   Where λ is an hyperparameter which allows to specify how important are the old parameters, when learning a new task. An higher λ means the network will not forget as much, by sacrificing performance on the new task. A lower λ 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

Contents

5.1 Approach 49

5.2 The Environments as Tasks 49

5.3 The First Hypothesis - Transfer Learning 52

5.4 The Second Hypothesis - Catastrophic Forgetting 60

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

1. *"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?*

2. *"Does applying EWC, an algorithm that tackles catastrophic forgetting, allow the agent to maintain the acquired knowledge from previous tasks after learning new ones?"*.

5.1 Approach

As tasks, we used Atari2600 environments available in the OpenAI gym toolkit. An Atari2600 OpenAI Gym environment only provides two perceptions as input for the agent - the game screen (pixels) and the current score (reward). An agent interacts with an environment through episodes. At the end of an episode, the agent has achieved a certain score. The main goal for the agent is to execute actions on the environment in a way that maximizes its Avg Score / Ep. (average score per episode). An episode can be represented by the discrete-time sequence of states, executed actions and obtained rewards (Sec. 4.3.2). For a given timestep t, there is an associated state s_t, built by the environment handler, a reward r_t and a boolean telling if s_t is a terminal state or not. A terminal state represents the end of the episode. At the end of the episode, the sum of all the rewards provides the obtained score.

If an agent A obtains an higher Avg. Score / Ep. than an agent B, where both trained for a environment ϵ during N total episodes, then we consider the agent A to be a better agent for environment ϵ with E training episodes. For example, if A and B were training on ϵ for 100000 episodes, and in episode 100000, agent A has an Avg. Score / Ep. of 2000 points, and agent B has an Avg. Score / Ep. of 1500 points, then we consider agent A to be the better agent on environment ϵ, given 100000 training episodes. This does not invalidate, however, that if both kept training on ϵ for an additional 50000 episodes (adding up to 150000), if agent B surpasses agent A and with 150000 episodes, by achieving an Avg. Score / Ep. of 3000, while A achieved only 2500, that agent A is still a better agent for environment ϵ. Therefore which agents are the best for an environment are relative to the number of training episodes.

5.2 The Environments as Tasks

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. 5.1, top left), Phoenix (Fig. 5.1, top right), Carnival (Fig. 5.1, bottom left) and Demon Attack (Fig. 5.1, bottom right):

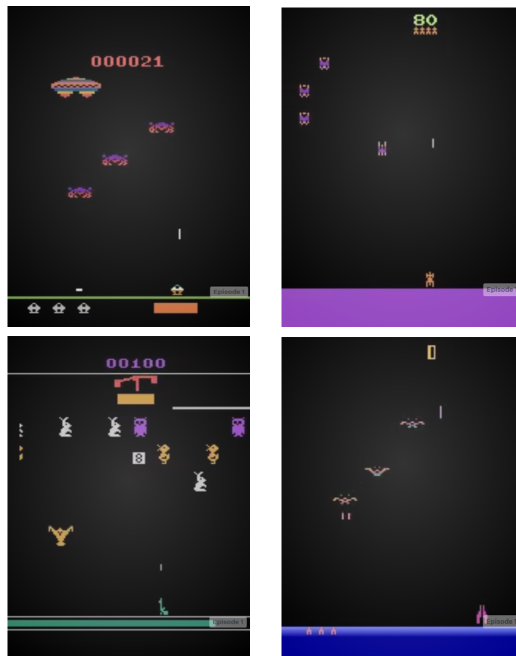


Figure 5.1: The source tasks - Assault (Top Left), Phoenix (Top Right), Carnival (Bottom Left) and Demon Attack (Bottom Right). OpenAI Gym environments AssaultDeterministic-v4, PhoenixDeterministic-v4, CarnivalDeterministic-v4 and DemonAttackDeterministic-v4, respectively

And the environment used as target task (also from the Atari2600 platform), was Space Invaders (Fig. 5.2)

Figure 5.2: The target task - Space Invaders. OpenAI Gym environment SpaceInvadersDeterministic-v4

We chose these environments because they all share noticeable similarities. 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), described in 4, 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 AssaultDeterministic-v4 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 PhoenixDeterministic-v4 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 CarnivalDeterministic-v4 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 DemonAttackDeterministic-v4 for a total of 150000 episodes.

HybridNet: An Hybrid GA3C, equivalent to the Hybrid A3C [18], 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 AssaultDeterministic-v4 for 45230 total episodes, 4 actor-learners trained on PhoenixDeterministic-v4 for 31932 total episodes, 4 actor-learners trained on CarnivalDeterministic-v4 for 44682 total episodes and 4 actor-learners trained on DemonAttackDeterministic-v4 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 5.1 shows configuration choices and hyper parameters.

Hyper Parameter	Value Used	Description
Learning Rate	0.0003	Learning rate for gradient descent steps
Discount Factor	0.99	Discount factor for reward accumulation
Logarithm Noise	0.000001	Minimum value for logarithms during loss function setup
Entropy Beta	0.01	Entropy weight in loss function setup
Reward Minimum Clip	-1	All rewards below this value are set to this value
Reward Maximum Clip	1	All rewards above this value are set to this value
Predictor Threads	1	No. of threads processing prediction requests
Prediction Queue Max Size	100	Maximum prediction requests on queue
Prediction Min Batch	128	No. of predictions processed on GPU at a time
Trainer Threads	2	No. of trainer threads
T_Max	5	No. of timesteps for before training if no terminal state
Training Queue Max Size	100	Maximum datapoints for training on queue
Training Min Batch	128	Number of datapoints processed on GPU at a time

Table 5.1: Hyper parameters and configuration choices used when training the five agents for 150000 episodes.

5.3 The First Hypothesis - Transfer Learning

To test this hypothesis, we test if the HybridNet, which trained simultaneously for Assault, Phoenix, Carnival and Demon Attack, is capable of achieving a better Avg Score / Ep. while training for an additional number of episodes on Space Invaders. In every episode $e \in [T_{start}, T_{end}]$, we recorded the episode score, the (running) average score (Avg Score / Ep.) and the (running) standard deviation. We display a learning curve by plotting the Avg Score / Ep. for each $e \in [T_{start}, T_{end}]$.

Before testing the hypothesis, we start by evaluating the learning of all five agents, by comparing their learning curves on their respective environments, from 0 to 150000 episodes. 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. This way we were able to learning curves for agents that learned different environments (Fig. 5.3).

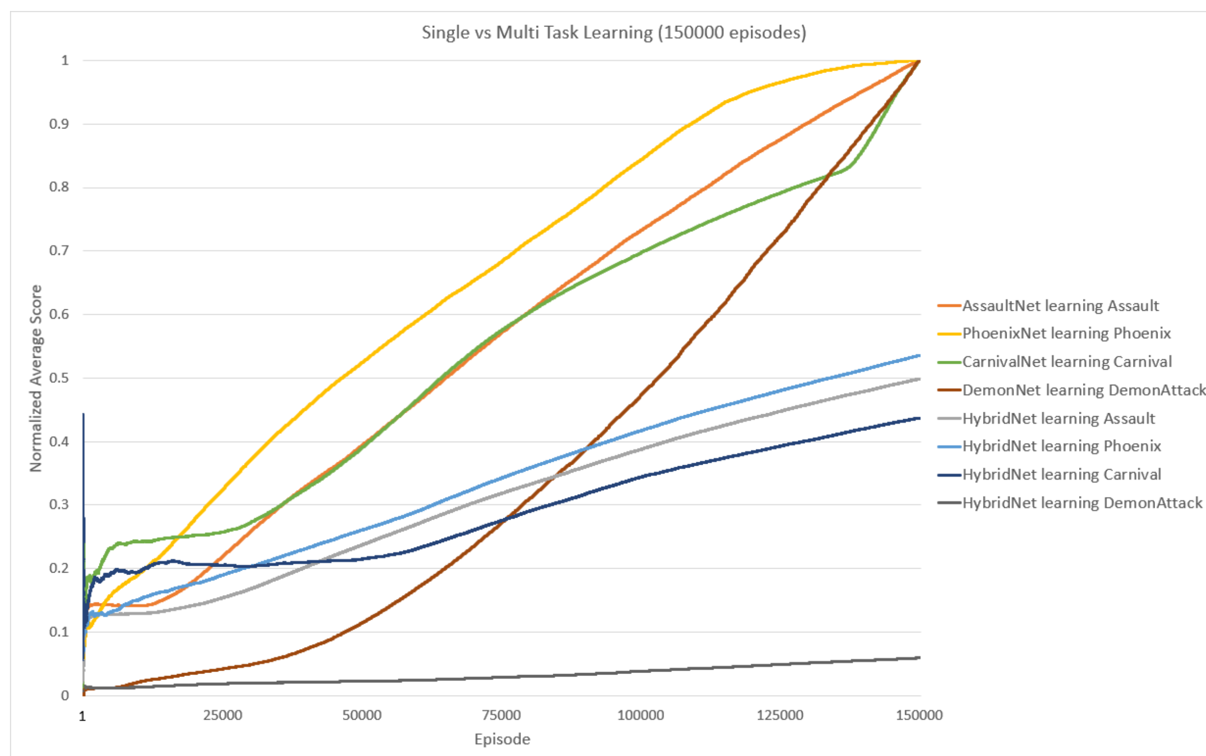


Figure 5.3: Single vs Multi-task Learning. Learning curves for all five agents. Scores are normalized between 0 (lowest Avg Score / Ep.) and 1 (highest Avg Score / Ep.), relative to each environment's score space. The HybridNet has four separate learning curves, one for each environment.

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. We compared the HybridNet's learning curves individually with the other agents on their respective environments. The score spaces are now shown accordingly

to the environment's real scale, i.e., not normalized between 0 and 1 (Figs. 5.4, 5.5, 5.6 and 5.7).

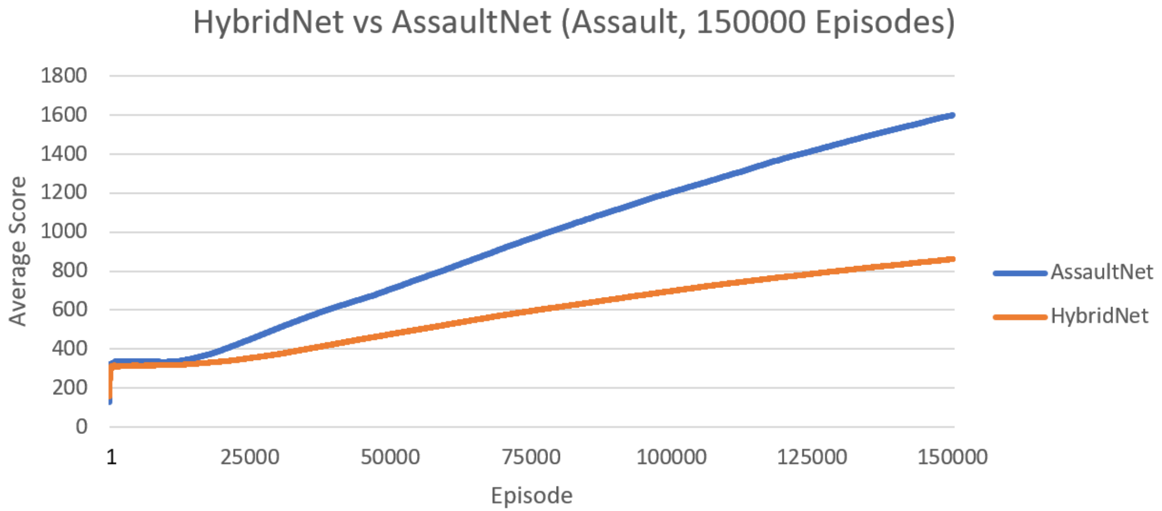


Figure 5.4: AssaultNet and HybridNet learning Assault. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 860.60 points, while the AssaultNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 1599.50 points.

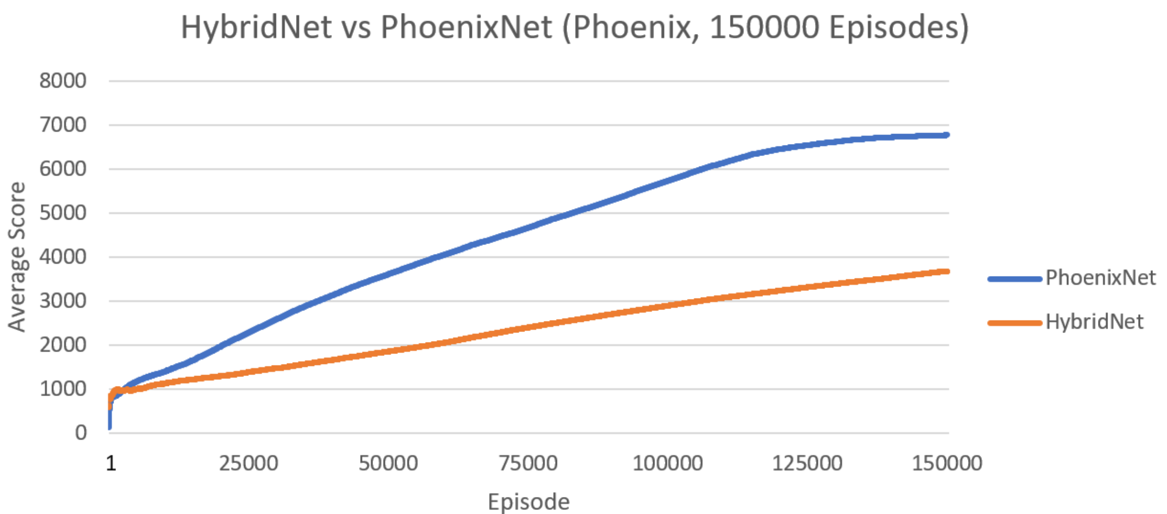


Figure 5.5: PhoenixNet and HybridNet learning Phoenix. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 3685.87 points, while the PhoenixNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 6778.29 points.

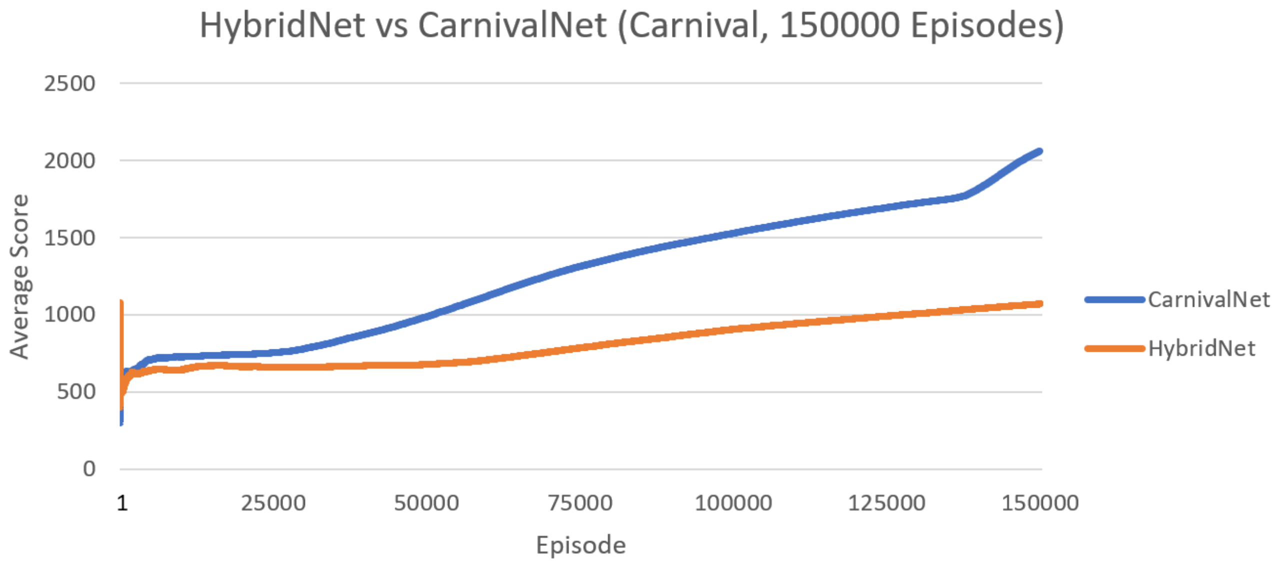


Figure 5.6: CarnivalNet and HybridNet learning Carnival. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 1070.38 points, while the CarnivalNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 2063.18 points.

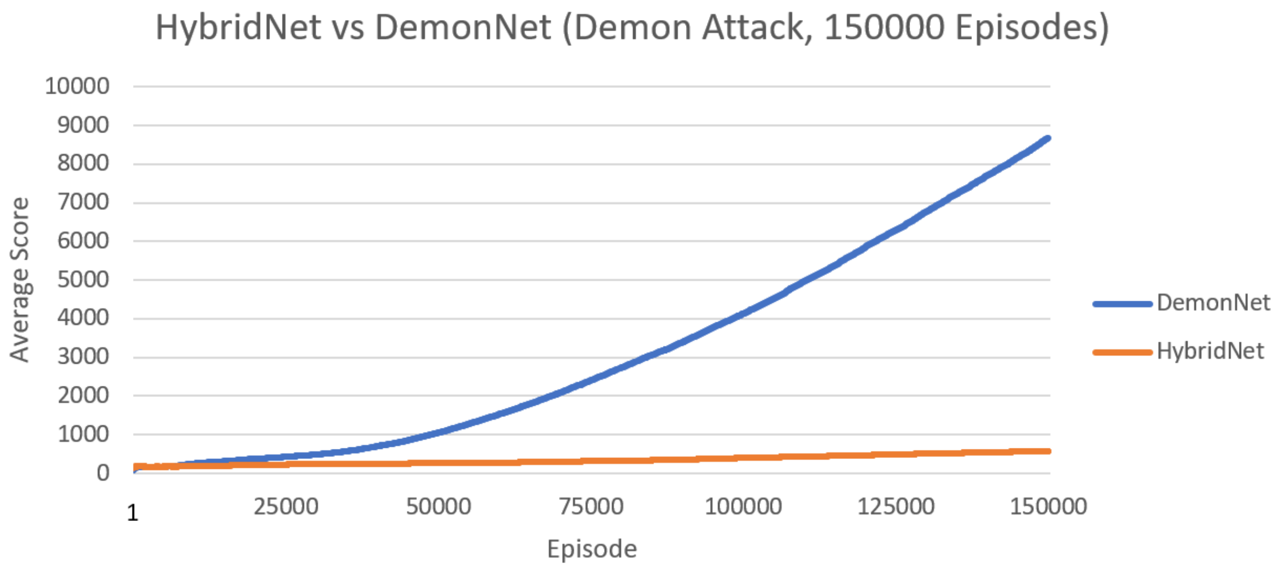


Figure 5.7: DemonNet and HybridNet learning Demon Attack. After 150000 total multi-task episodes, the HybridNet was achieving an Avg Score / Ep. of 577.66 points, while the DemonNet after a total of 150000 single-task episodes was achieving an Avg Score / Ep. of 8675 points.

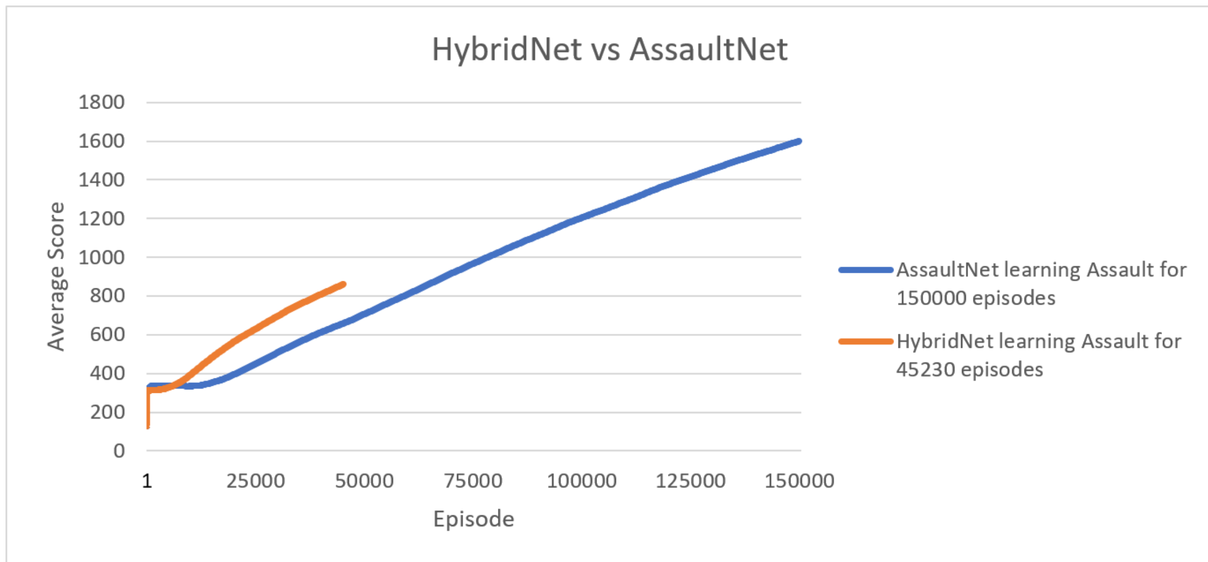


Figure 5.8: AssaultNet and HybridNet learning Assault. With 45230 training episodes, AssaultNet was achieving an Avg Score / Ep. of 658.98 points and the HybridNet was achieving an Avg Score / Ep. of 860.60 points. The AssaultNet, however, kept training on Assault until 150000 episodes where it was able to achieve an Avg Score / Ep. of 1599.50 points.

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. 5.8, 5.9, 5.10 and 5.11).

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. Learning multiple tasks at the same time does in fact enhance the training process for each task. 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?

In order to test our hypothesis, we start by evaluating all agents in all five tasks, meaning Space Invaders is included, in order to see which agent performs the best, given that none have trained a single episode of Space Invaders. For these measurements, all agents are ran in exploitation mode,

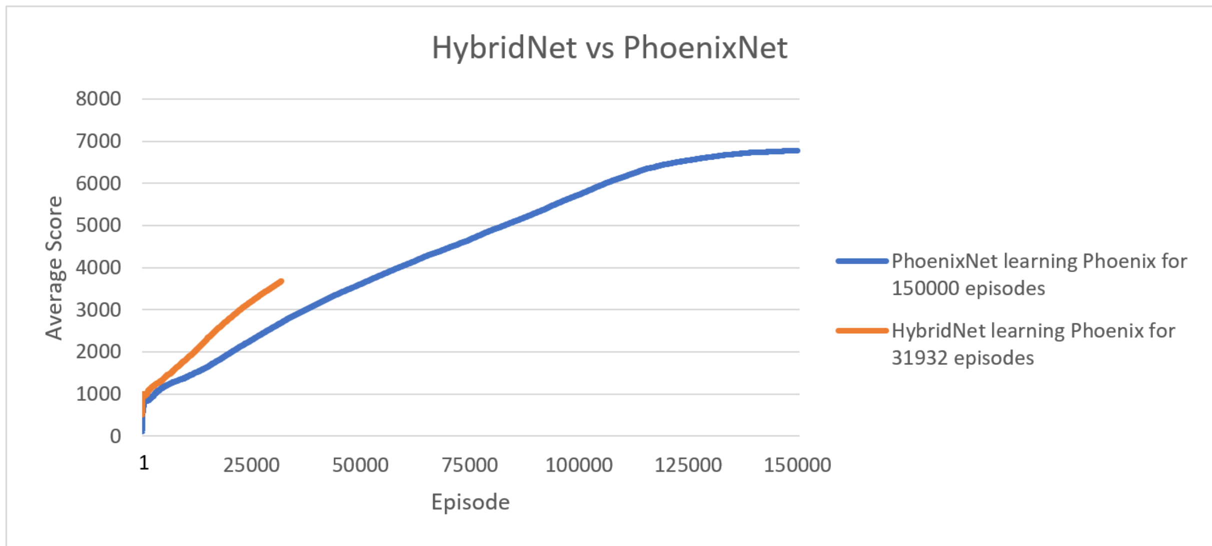


Figure 5.9: PhoenixNet and HybridNet learning Phoenix. With 31932 training episodes, PhoenixNet was achieving an Avg Score / Ep. of 2701.61 and the HybridNet was achieving an Avg Score / Ep. of 3685.87. The PhoenixNet, however, kept training on Phoenix until 150000 episodes where it was able to achieve an Avg Score / Ep. of 6778.29 points.

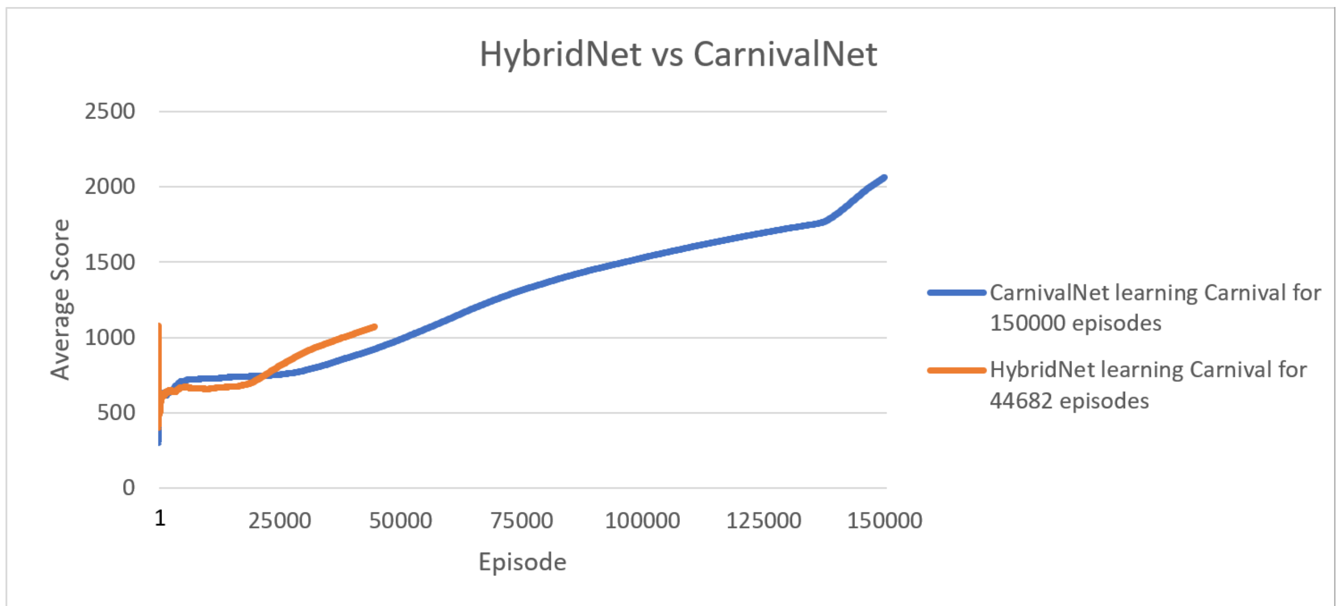


Figure 5.10: CarnivalNet and HybridNet learning Carnival. With 44682 training episodes, CarnivalNet was achieving an Avg Score / Ep. of 923.57 and the HybridNet was achieving an Avg Score / Ep. of 1070.38. The CarnivalNet, however, kept training on Carnival until 150000 episodes where it was able to achieve an Avg Score / Ep. of 2063.18 points.

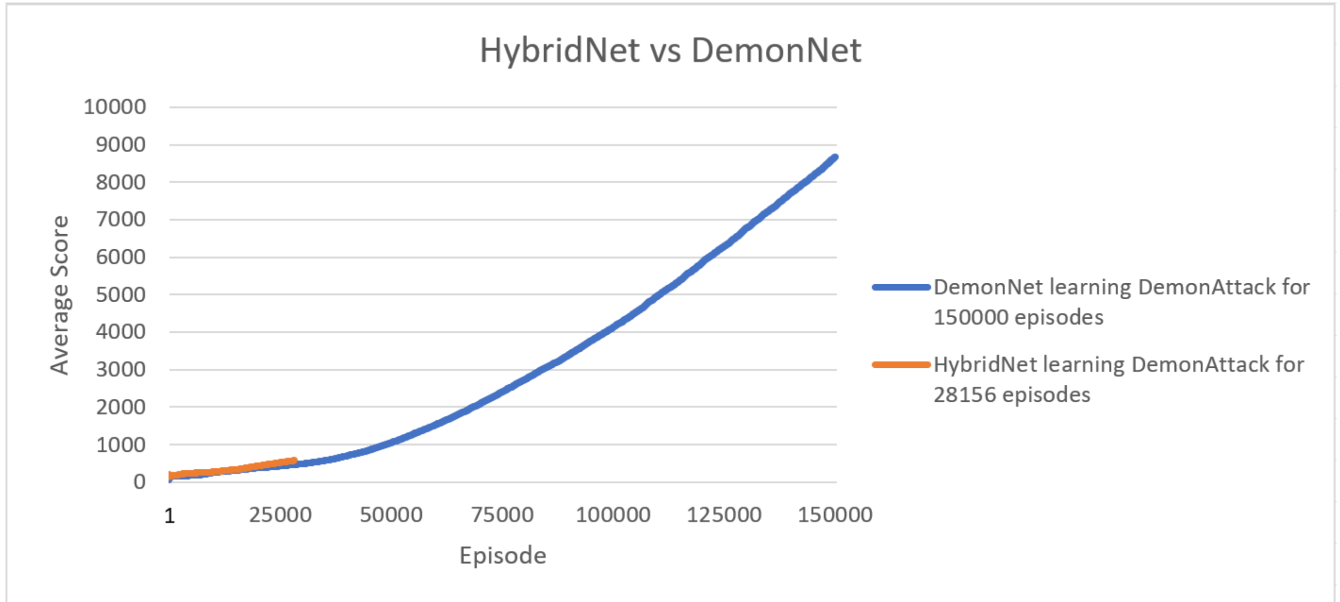


Figure 5.11: DemonNet and HybridNet learning Demon Attack. With 28156 training episodes, DemonNet was achieving an Avg Score / Ep. of 464 and the HybridNet was achieving an Avg Score / Ep. of 577.66. points. The DemonNet, however, kept training on Demon Attack until 150000 episodes where it was able to achieve an Avg Score / Ep. of 8675 points.

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 π, 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, but allow the fine tuning of this value in the configuration file.

In exploitation mode, we ran the five agents, on all five environments, for a total of 100 episodes. The obtained results are found in table 5.2.

	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 5.2: 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.

As seen in Table 5.2, 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, expecting the HybridNet to learn better than all the other agents. Figure 5.12 displays the resulting learning curves:

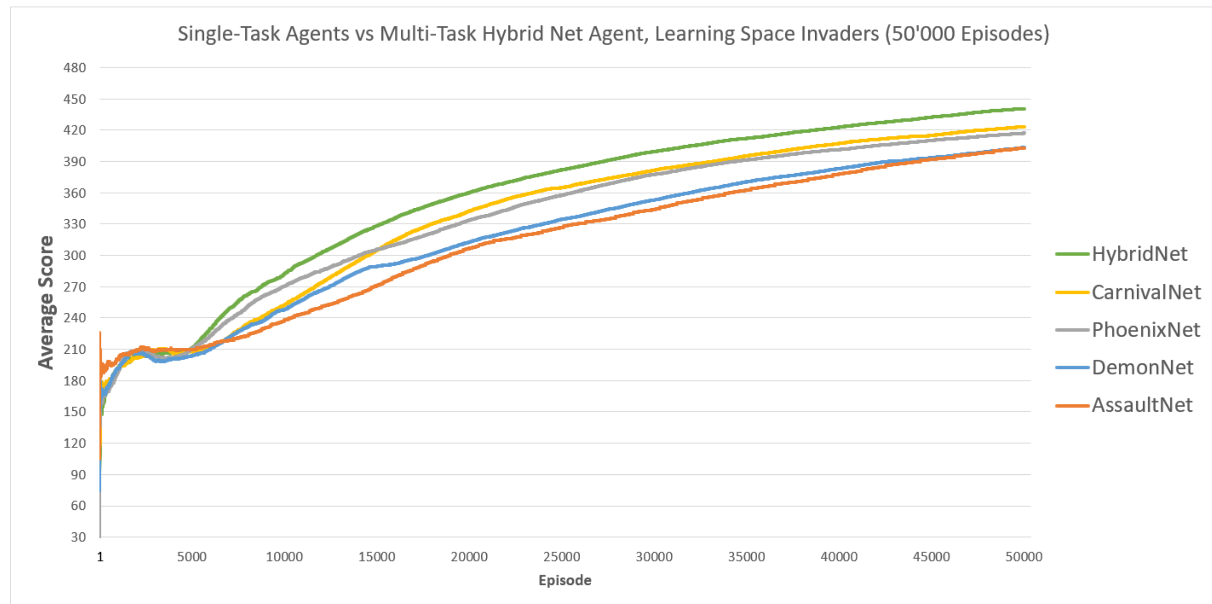


Figure 5.12: All five agents learning Space Invaders for 50000 additional episodes. After the 50000 episodes, the HybridNet was able to achieve an the Avg Score / Ep. of 440.44 points (the best), the CarnivalNet, an Avg Score / Ep. of 423.37 points (second best), the PhoenixNet, an Avg Score / Ep. of 417.05 points (third best), and the AssaultNet, an Avg Score / Ep. of 403.00 (the forth best).

These 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 (ranking 17.07 points above the second best agent, the CarnivalNet). This is not such a significant difference, and therefore we decided to run the five agents on 100 episodes on all environments, in exploitation mode, to see if the HybridNet did in fact obtain substantially better results. Table 5.3 displays the results.

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

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. 5.12), but there was no strong advantage when performing the new task (Table 5.3). 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.

5.4 The Second Hypothesis - Catastrophic Forgetting

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 5.4 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.

	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 5.4: 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 5.2 and 5.3.

We can see that the original environments for which each agent trained, for 150000 episodes, were where the agents suffered from heavier catastrophic forgetting (AssaultNet-200000 on Assault, PhoenixNet-200000 on Phoenix, CarnivalNet-200000 on Carnival, DemonNet-200000 on Demon Attack, and HybridNet-200000 on all four). 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.

Consider the HybridNet-150000, which trained on all four environments simultaneously. After a mixture of 150000 episodes from all environments, it obtained an Avg. Score / Ep. of 860.60 points on Assault, 3685.87 points on Phoenix, 1070.38 points on Carnival and 577.66 points on Demon Attack. The parameters from the input network θ_{input} converged into star values θ^*_{input} which allowed the HybridNet-150000 to achieve this performance. However, when training 50000 additional episodes on Space Invaders, the parameters from the input network θ_{input} converged into new values, losing the old ones. It suffered from catastrophic forgetting. 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 algorithm, detailed in Chap. 4, modifies the loss function for a new environment in order to prevent catastrophic forgetting and losing the parameters for the previous environments. The EWC requires the star values θ^*_{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), sampled from a total of 20 episodes. 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 λ hyper parameter value. The used λ values were λ=0.5, λ=1.0, λ=10.0 and λ=50.0. The lower λ, the more the parameters θ^*_{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 λ = 0.00. Figure 5.13 displays the obtained learning curves.

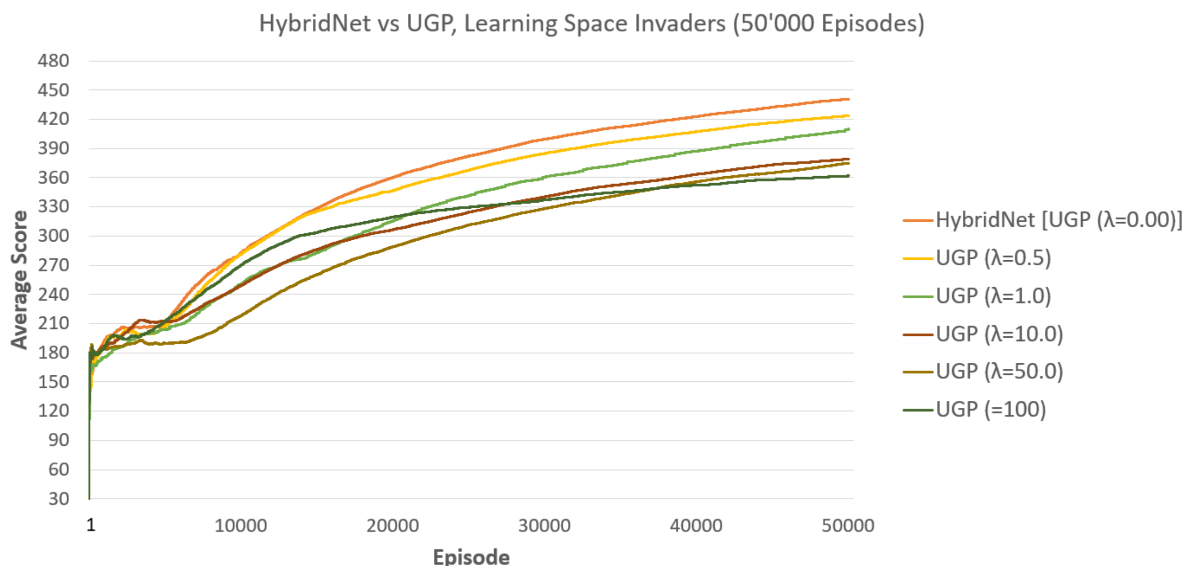


Figure 5.13: Space Invaders learning curves for the HybridNet-200000 and the UGP instances with different λ. The higher the λ hyper parameter, the less altered the input network's parameters p will be in order to learn the new environment. Using a λ of 0, 0.5 and 1 provides similar results. As expected, the higher the λ, the lower worst the learning curve.

As expected, the higher the λ, 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 λ 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 5.5 displays the results after running each agent on each environment, in exploitation mode, for 100 episodes.

	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 λ=0.5	85.05	622.50	439.20	0.00	522.90
UGP λ=1	265.02	679.70	236.00	168.60	576.10
UGP λ=10	95.55	311.40	144.80	200.00	455.35
UGP λ=50	716.92	3547.40	1192.20	240.10	519.40
UGP λ=100	1126.14	4903.70	1717.80	650.35	413.65

Table 5.5: 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 λ values were experimented, trained. The HybridNet-200000, even though has no EWC algorithm implemented, can be seen as a EWC augmentation with λ = 0.

As expected, we can see that as λ increases, the less catastrophic forgetting occurs on the original four environments. Table 5.6 shows the the performance variations (in %), relative to the HybridNet-150000's performance and Fig. 5.14 plots the performance variations (in %) for the other four tasks, as the λ value increases.

	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 λ=0.5	6.30%	11.41%	21.54%	0.00%	277.77%
UGP λ=1	19.64%	12.46%	11.57%	16.29%	306.03%
UGP λ=10	7.08%	5.71%	7.10%	19.32%	241.89%
UGP λ=50	53.13%	65.01%	58.46%	23.20%	275.91%
UGP λ=100	83.46%	89.86%	84.24%	62.83%	219.73%

Table 5.6: Results from Table 5.5, but relative to the HybridNet-150000's score. Both UGP λ=50 and λ=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.

Taking these results into account, we can see that as the λ 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. 5.14), three of the UGP instances were able to outperform the HybridNet-200000 (λ=0.5, λ=1 and λ=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 λ, the lower the Avg. Score / Ep. at episode 200000.

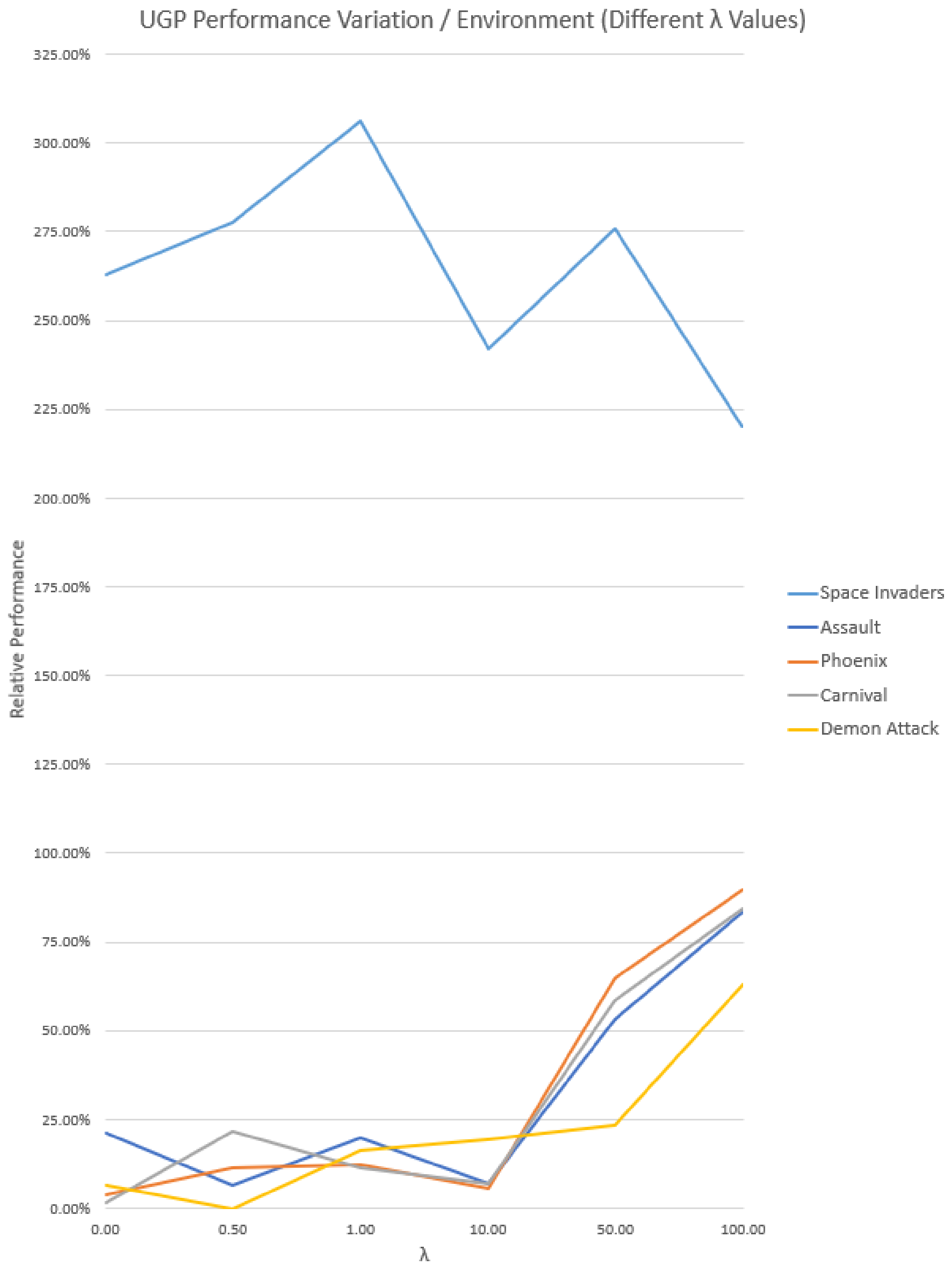


Figure 5.14: Performance variations for the original source tasks, given different lambda values. With λ=100, the UGP is able to overcome a substantial amount of catastrophic forgetting, still being able to achieve a relatively good performance on Space Invaders.

UGP λ = 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 λ = 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 λ 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 λ 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.

6

Conclusion

Contents

6.1 Conclusions 67
6.2 Future Work 68

6.1 Conclusions

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 λ 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.

6.2 Future Work

6.2.1 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.

6.2.2 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 [18], 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.

6.2.3 Continuous Action Spaces

The actor-critic model approximates a policy function π. 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 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 μ, 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.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] A. Juliani, "Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C)," https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2, accessed: 2018-01-03.

[3] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "Reinforcement learning through asynchronous advantage actor-critic on a gpu," *arXiv preprint arXiv:1611.06256*, 2016.

[4] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[5] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1717–1724.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[7] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[8] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 3. IEEE, 2004, pp. 2619–2624.

[9] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, “Autonomous inverted helicopter flight via reinforcement learning,” in *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[10] B. Bouzy and B. Helmstetter, “Monte-carlo go developments,” in *Advances in computer games*. Springer, 2004, pp. 159–174.

[11] P. Baudiš and J.-l. Gailly, “Pachi: State of the art open source go program,” in *Advances in computer games*. Springer, 2011, pp. 24–38.

[12] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, “Monte-carlo tree search: A new framework for game ai.” in *AIIDE*, 2008.

[13] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[14] Y. LeCun, K. Kavukcuoglu, and C. Farabet, “Convolutional networks and applications in vision,” in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 2010, pp. 253–256.

[15] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents.” *J. Artif. Intell. Res.(JAIR)*, vol. 47, pp. 253–279, 2013.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[17] R. Caruana, “Multitask learning,” in *Learning to learn*. Springer, 1998, pp. 95–133.

[18] M. Birck, U. Corrêa, P. Ballester, V. Andersson, and R. Araujo, “Multi-task reinforcement learning: An hybrid a3c domain approach,” 01 2017.

[19] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, “Overcoming catastrophic forgetting in neural networks,” *Proceedings of the National Academy of Sciences*, p. 201611835, 2017.

[20] R. Kemker, A. Abitino, M. McClure, and C. Kanan, “Measuring catastrophic forgetting in neural networks,” *arXiv preprint arXiv:1708.02072*, 2017.

[21] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: a system for large-scale machine learning.” in *OSDI*, vol. 16, 2016, pp. 265–283.

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” in *ACM SIGGRAPH 2008 classes*. ACM, 2008, p. 16.

[25] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747*, 2016.

[26] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “A brief survey of deep reinforcement learning,” *arXiv preprint arXiv:1708.05866*, 2017.

[27] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015.

[28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[29] M. Roderick, C. Grimm, and S. Tellex, “Deep abstract q-networks,” *arXiv preprint arXiv:1710.00459*, 2017.

[30] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[31] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 5026–5033.

[32] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, “Torcs, the open racing car simulator,” *Software available at http://torcs. sourceforge. net*, 2000.

[33] M. Campbell, A. J. Hoane, and F.-h. Hsu, “Deep blue,” *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[34] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra, “Pathnet: Evolution channels gradient descent in super neural networks,” *arXiv preprint arXiv:1701.08734*, 2017.

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

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

[37] M. Marszalek and C. Schmid, "Semantic hierarchies for visual object recognition," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–7.

[38] Q. Chen, Z. Song, J. Dong, Z. Huang, Y. Hua, and S. Yan, "Contextualizing object detection and classification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 1, pp. 13–27, 2015.

[39] A. Ly, M. Marsman, J. Verhagen, R. P. Grasman, and E.-J. Wagenmakers, "A tutorial on fisher information," *Journal of Mathematical Psychology*, vol. 80, pp. 40–55, 2017.