Multi-task learning for Atari games using Deep Neuroevolution

Deyan Dyankov
(Student ID Number: 24885531)

Submitted in partial fulfillment of the requirements of the
University of Reading
for the degree of
Master of Science in Advanced Computer Science

Supervisor:
Dr Giuseppe Nicosia
Department of Computer Science

September 2018

Abstract

Today, autonomous vehicles, drones, robots and various types of expert systems rely on Reinforcement Learning to an ever growing extent. The stabilization and democratization of any of the technologies just mentioned is likely to affect the way one lives and, perhaps more importantly, the way we perceive the world. Unfortunately, even though RL borrows concepts from evolution, neuroscience and animal behaviour, its efficiency is far from optimal. To learn any given task, artificial agents need to perform orders of magnitude more iterations of trial and error than their biological counterparts. If Reinforcement Learning is to live up to its full potential, these inefficiencies need to be resolved.

Aims: The objective of the current research is to use multi-task learning to train an agent to play two similar Atari 2600 games simultaneously, *using a single model*, rather than training two different agents to play the same two games using two separate models.

Methods: Paradigms from Reinforcement Learning and Multi-Task Learning have been combined for the implementation of a Multi-Task Evolution Strategies model. Five experiments have been devised and conducted in order to compare the introduced model against previous state-of-the-art benchmarks.

Results: The new model reduced the time required to learn both games by more than 45%, ultimately achieving better performance on one of the games and adequate performance on the other game. It is able to better generalize common features of the two environments. Furthermore, the experiments made clear that models can sometimes benefit from stripping certain actions from an agent's repertoire - a notion which, at first, contradicts conventional wisdom.

Conclusion: Multi-Task Evolution Strategies is a viable alternative to other state-of-the-art models when a substantial reduction of learning time is required and potentially suboptimal task performance can be tolerated.

palindrome, *n.*:

a word, phrase, verse, or sentence that reads the same backward or forward

Word count: 10101

Acknowledgements

I wish to express my deep gratitude to my supervisor Dr. Giuseppe Nicosia for his support, guidance and constructive criticism.

I would also like to thank the dissertation committee and all of University of Reading's Computer Science department staff for their continuous efforts and hard work.

1. Introduction

Research in Artificial Intelligence (AI) and in particular in Machine Learning (ML) has increased significantly during the 21st century. It seems that more and more areas of our lives are being affected by machine intelligence and autonomous agents. Although most of the ML success stories and awareness are the result of supervised learning, Reinforcement Learning (RL) may have an even greater impact on our lives and our society as a whole.

Today, autonomous vehicles, drones, robots and various types of expert systems rely on RL to an ever growing extent. The stabilization and democratization of any of the technologies just mentioned is likely to affect the way one lives and, perhaps more importantly, the way we perceive the world. With RL having such great potential, it must come as no surprise that many research teams, often well funded, are focused on improving the state-of-the-art in RL.

Unfortunately, even though RL borrows concepts from evolution, neuroscience and animal behaviour, its efficiency is far from optimal. To learn any given task, artificial agents need to perform orders of magnitude more iterations of trial and error than their biological counterparts. If RL is to live up to its full potential, these inefficiencies need to be resolved.

The current research represents an effort to combine two existing paradigms, Reinforcement Learning and Multi-Task Learning, in an attempt to address some of these inefficiencies.

Objective

The objective of the current research is to use multi-task learning to train an agent to play two similar Atari 2600 games simultaneously, *using a single model*, rather than training two different agents to play the same two games using two separate models.

We hypothesise that training such a model will result in an overall increase of feature generalisation and decrease in learning time. The game performance of an agent trained using this model will be comparable to or higher than the performance achieved by two agents that have been trained separately.

A successful outcome of the above objective consists of a single neural network that can be used by a single agent to play two different games, along with the code to reproduce the results.

Outline of this document

The outline of this document is as follows.

- Section 1. Introduction opens with a description of the problem at hand along with motivation on why it requires researching.
- Section 2. Literature Review is an account of the existing research to date that is related to similar problems.
- Section 3. Overview of Deep Reinforcement Learning Implementations describes the design of the most recent methods and algorithms used to tackle similar problems.

- Section 4. Investigation describes the work done as part of the current study with an emphasis on the solution design and its motivation.
- Section 5. Experiment design is a summary of the experiments performed as part of the current study.
- Section 6. Analysis of results presents the outcomes of the experiments and discusses pros and cons of using the new model compared to using previous models.
- Section 7. Known issues highlights inherent shortcomings of the proposed model.
- Section 8. Conclusion is a summary of the results of the study along with suggestions on how the proposed model could be improved and extended in the future.

2. Literature review

This section focuses on Reinforcement Learning (RL) and Multi-Task Learning (MTL). These two branches of machine learning, coupled with Deep Neural Networks (DNNs), will provide the tools to tackle the objective of this research.

Reinforcement Learning

Reinforcement Learning, also called Adaptive Dynamic Programming (ADP) can be used to solve complex sequential decision-making problems. In general, RL algorithms train agents to perform actions in particular environments so as to maximize a specific reward. A very comprehensive overview of RL methods at the end of the 20th century can be found in “Reinforcement learning: An introduction” [1].

Classical methods

Historically, RL has had three main threads that developed independently during the second half of the twentieth century, until they eventually converged. Those three threads focus on learning by optimal control, trial and error, and temporal-difference methods.

Optimal control aims to find a control law for a given system such that a certain optimality criterion is achieved. In the late 1950s, the method of Dynamic Programming (DP) was introduced by Richard Bellman in [6] as one approach to optimal control. In [7], Bellman also introduced Markovian Decision Processes (MDPs), which are a discrete stochastic version of the optimal control problem. The only widely considered feasible way to solve general stochastic optimal control problems is DP. However, the computational complexity of the solution grows exponentially with introducing more dimensions to the data - a phenomenon labelled by Bellman as “the curse of dimensionality”. Still, it is far more efficient than any other general method.

Trial-and-error learning has its roots in psychology and the first to express its essence was Edward Thorndike [8]. It is the idea that actions result in outcomes that can be either positive or negative, and the tendency to select a particular action is altered based on the distribution of those outcomes. Thorndike called this “Law of Effect” - a principle which is currently widely regarded to underlie much of animal behaviour. Law of Effect allows us to develop algorithms that combine *search* and *memory*: trying out and selecting among particular actions in each situation, and associating those actions with the situations they produced the highest rewards. Perhaps the earliest investigations of trial-and-error from a computational perspective were done in 1954 by Minsky [9] and by Farley and Clark [10], who developed computer models based on principles found in animal psychology. Over the next few decades, researchers tended to focus more on supervised learning rather than reinforcement learning but there have nevertheless been some influential developments in trial-and-error. One such development is MENACE (Matchbox Educable Naughts and Crosses Engine) by Donald Michie, which is a trial-and-error learning system that plays tic-tac-toe. Michie and Chambers [11] developed another tic-tac-toe reinforcement learner called GLEE (Game Learning Expectimaxing Engine) and a controller called BOXES that they applied to the task of balancing a pole hinged to a movable cart. This pole balancing problem is now widely used as a benchmark task for many RL algorithms - it is the first benchmark for a learning task under conditions of incomplete knowledge.

Temporal-difference (TD) learning methods build on trial-and-error learning by taking into account the variance of the reward caused by an action, when this action is performed successively. The first implementation of a TD learning method was by Arthur Samuel [12] who used it to teach an agent to play the game of checkers. In 1972, Harry Klopf [13] linked TD learning with trial-and-error learning by developing the idea of "generalized reinforcement" - a change in perspective where every input to a model was treated in reinforcement terms, i.e. excitatory inputs (rewards) and inhibitory inputs (punishments). This allowed researchers to build on the solid empirical knowledge of animal psychology and test out new methods that mimic animal behaviour computationally. Following this thread, Sutton and Barto [14] developed a computational equivalent of the psychological model of classical conditioning. In the next several years, a number of ideas from neuroscience were borrowed to further the development of temporal difference and trial-and-error learning [15, 16, 17].

Finally, the three threads that have so far been discussed (optimal control, trial-and-error, temporal-difference learning) have been brought together by Chris Watkins' Q-Learning in 1989 [18]. The goal is to learn a policy which informs an agent what action to take under what circumstances. The output of a Q-Learning model is a Q-Learning table of states and actions which are initialized with zeros and updated on-line during the process of training. The actual Q-Learning algorithm varies and is tailored to the task at hand, i.e. environment and action space. A lot of the modern methods that will be discussed in the next section are ultimately different means of deriving and expressing Q-Learning tables.

Modern Methods

The exponentially increasing computational capacity over time [19] coupled with the decreasing costs of digital storage and memory at the beginning of the 21st century has allowed us to generate, store and process ever increasing volumes of data. As a consequence of these increased capabilities, research in AI intensified substantially. Computationally demanding methods such as Neural Networks (NNs) which were first introduced by Warren McCulloch and Walter Pitts in 1943 [20] became more feasible and attractive following optimisations such as back-propagation [21] and developments such as Convolutional Neural Networks (CNN) [22]. The applicability and success of RL algorithms in autonomous vehicle navigation [23, 24] and various board-games [25, 26] have prompted the development of several platforms for RL research that provide environments which agents can seamlessly integrate with and learn to navigate. Such environments include MuJoCo [27], a physics engine that aims to facilitate research and development in robotics and other areas where simulation is needed, and the Arcade Learning Environment (ALE) [28] and OpenAI Gym [29] which are environments that simulate Atari 2600 video games and other control theory benchmark problems.

Following the increases in processing capacity and improvements in methods discussed above, many areas of ML have been greatly impacted by Deep Learning (DL) [30] which are neural network architectures comprising of many hidden layers. Most importantly, DL reduces the effect of the previously mentioned "curse of dimensionality" by automatically discovering low-dimensional representations of high-dimensional data. DL has similarly impacted RL, defining the field of Deep Reinforcement Learning (DRL) [2].

In [2], the researchers from DeepMind set out to create a single algorithm that would be able to perform well in varied types of environments. Their Deep Q-Network (DQN) is trained on a set of Atari 2600 games and several challenges simulated in MuJoCo [27]. DQN integrates a Q-table into a deep Convoluted Neural Network (CNN) [22] in order to take advantage of CNN's inherent hierarchical structure. In the case of the Atari 2600 games, the agent's inputs are raw pixel data from the game's subsequent screen frames, and the outputs are the available actions the agent can perform in this particular game.

Building on DeepMind's paper [2], researchers from OpenAI have explored [3] the use of Evolution Strategies (ES) as an alternative to popular MDP, Q-Learning and policy gradient approaches. ES [32] are black-box optimization algorithms that imitate natural evolution and have first originated in Germany during the 1960s. OpenAI's experiments show that ES is a viable solution strategy that scales extremely well with the number of CPUs available to the training process. Moreover, the ES optimization technique is invariant to delayed rewards and tolerant of extremely long horizons, which proves very beneficial in the domain of Atari games.

Following DeepMind's [2] and OpenAI's [3] research, agent performance on specific Atari games has been further improved by researchers from Uber in [4] and [5]. Instead of using, as is common, gradient-based learning algorithms such as back-propagation [21], the researchers set out to demonstrate that evolving the neural network weights using Genetic Algorithms (GA) is a viable alternative that performs well on hard DRL problems. More importantly, this research makes available the multitude of neuroevolution techniques that improve performance and demonstrates that Novelty Search (NS), which encourages exploration on the agent's part, can be beneficial for tasks in which reward-maximizing algorithms fail.

Multi-task learning

Multi-Task Learning (MTL) is a learning paradigm in ML where the aim is to solve multiple learning tasks at the same time. MTL can result in improved learning efficiency and prediction accuracy compared to when separate models are trained for separate tasks. It improves generalization by leveraging domain-specific information from training signals in related tasks. Historically, MTL was called "learning from hints" and was initially described by Yaser Abu-Mostafa in [34]. A 1997 paper [35] published by Rich Caruana demonstrates that the benefit of using additional tasks can be substantial and discusses a number of mechanisms that explain how MTL improves generalization.

Several methods exist that support MTL in the context of DNNs - the two most common such methods are described below.

Hard parameter sharing

Hard Parameter SHaring, as proposed in [36], is perhaps the single most commonly used approach for MTL. Generally, it is applied by creating a single DNN for multiple tasks and sharing its hidden layers between tasks, while having multiple task-specific output layers.

In hard parameter sharing, the risk of overfitting specific tasks is greatly reduced as demonstrated by Jonathan Baxter in [37]. Hard Parameter Sharing MTL models are optimizing for similarities in all of the tasks they learns on, and the noise from these tasks is more likely to be cancelled out.

Soft parameter sharing

In Soft Parameter Sharing, on the other hand, the tasks are still learned in a single model, but each task has its own separate set of inputs and different sets of hidden layers correspond to different learning tasks. The tasks' inputs are stacked together to combine a single input layer, however a particular task input layer section links to its task-specific hidden layers. The task-specific hidden layers remain separate, however they are intertwined by hidden shared features. Finally, each task has its own separate output layer.

3. Overview of Deep Reinforcement Learning implementations

As previously mentioned, Uber [4] have extended DeepMind's [2] and OpenAI's research [3], therefore what follows are overviews of the different approaches found in those key papers.

DeepMind (Deep Q-Network)

In [2], the researchers from DeepMind have developed a code base that allows them to train a model by making an agent interact with a particular Atari game or a particular MuJoCo [27] environment. The current document focuses on Atari games only and therefore MuJoCo will not be discussed any further. In the Atari environment, the goal of the agent is to maximize its game score. The inputs to the model are consecutive frames from the game and the outputs are the actions that the agent must take, after observing a particular environment stage, in order to maximize the reward.

Source code available at https://sites.google.com/a/deepmind.com/dqn

Preprocessing

Since Atari 2600 games' frames are 210x160 pixel images with a 128-colour palette, researchers apply a basic preprocessing step which aims to reduce the input dimensions for the model. From the input RGB image, the Y channel (also known as luminance) is extracted and rescaled to 84x84 pixels. The output of $m=4$ different frames is stacked to produce the input for the neural network, and therefore m is a hyperparameter to the model that can be adjusted accordingly.

Model architecture

The input layer of the DNN is the 84x84x4 image produced by the preprocessing step. Following are three hidden layers that convolve 32 filters of 8x8 neurons with stride=4, 64 filters of 4x4 neurons with stride=2 and 64 filters of 3x4 neurons with stride=1 respectively. The last hidden layer consists of 512 rectifier units and is fully connected. Finally, the output layer consists of a set of nodes for each valid action of the particular game. Different games have different action spaces expressed with growing integer values and the no-op action is denoted as 0. The games tested on have action spaces ranging from 4 to 18 actions. This means that the actual model architecture is generated on the fly, at the beginning of the training process, and the size of its output layer is game-dependent.

Training details

A different network is trained on each game, however all networks have been trained using the same learning algorithm, have the same architecture, and use the same values for their hyperparameters. There is one important alteration regarding the rewards - since different games have different reward structures, positive rewards are translated to 1 and negative rewards are translated to -1 during training only. This allows for the use of the same learning rate in all games but may negatively affect game performance because the agent cannot differentiate between rewards of different magnitudes. Finally, following previous approaches [31], frame skipping is in place. This means that the agent only acts on every *k* frame and repeats the selected action for another *k* frames (*k=4* by default). Frame-skipping is an optimisation technique that allows the agent to play roughly *k* more games for the same amount of wall time.

Algorithm

The algorithm derived in [2] is called Deep Q-Learning with Experience Delay and is thoroughly described in the paper. The key points are that the agent does not have information on the emulator's internal state, it only uses vectorized processed images (frames) from the game that represent the current screen, and that the sequence of the screens already observed from the particular episode are stored in memory along with sequences of actions already taken.

OpenAI (Evolution Strategies)

In [3], the researchers from OpenAI take as a benchmark the problems tackled by DeepMind in [2], only this time they use ES as an optimisation technique instead of back-propagation. As a consequence, the preprocessing of the inputs, the neural network architecture and the training details are exactly the same as outlined in the previous section. A very interesting aspect of the paper is that the researchers have come up with a novel communication strategy for their ES implementation based on scalars of seeds used for random number generation. This novel communication strategy allows the learning process to scale to thousands of workers.

Source code available at https://github.com/openai/evolution-strategies-starter

Algorithm

The implementation of ES in the paper follows a class of Neural Evolution Strategies (NES) and is closely related to [33]. It consists of heuristic search procedures which perturb a set of parameter vectors on each iteration and evaluate their objective function. This procedure is repeated until the objective is fully optimized.

A very important feature of the ES implementation described above is that every game episode that the agent is training on can be distributed to a separate worker. This allows the researchers to use a master-slave architecture during training, where one process is the master and slaves are separate processes that only need to be coordinated by the master and send its results to it. All of these processes can run on different servers - the master requires little resources whereas the slaves maximise the use of RAM and CPU cores that are available on the server they operate in.

The whole process is further optimised by the novel communication strategy described earlier. The algorithm makes use of shared random seeds to reduce the bandwidth required for communication between the workers. Instead of all workers having to transfer whole gradients between themselves, they only send each other a single scalar value to agree on a parameter update.

Setup

The code base is developed in the Python3 programming language as opposed to Lua which was used by DeepMind in [2]. The experiments can be scheduled to run on a single server with one process being the master and other processes operating as slaves, or this setup could be extended to multiple servers where one server runs the master process and a set of slave servers executing up to several worker processes each. This setup makes it extremely easy for researchers to use commercially available cloud service providers such as Amazon Web Services, Microsoft Azure or Google Cloud Engine.

Uber (Genetic Algorithms)

In [4], the researchers from Uber take the Python3 code published by OpenAI as part of [3] and extend it to support Novelty Search (NS) and Genetic Algorithms (GA). They set out to demonstrate that non-gradient-based evolutionary algorithms, such as NS and GA, can work at DNN scales. As hypothesized by the researchers, NS indeed performs well on both Atari 2600 and MuJoCo benchmarks.

Perhaps as importantly, Uber have recently published an updated version of the code in [5] which makes use of a Graphics Processing Unit (GPU) and maximises CPU utilisation. More specifically, the NN-related computations are performed on the GPU whereas the actual training and testing, which consists of making the agent play the game, is carried out by the CPU. The latest version of the code enables one to train an agent on a modern GPU-enabled desktop computer instead of a cluster of several hundred servers that only have access to CPUs.

When training the agent to play the Atari games, the preprocessing of the inputs, the NN architecture and the training details are exactly the same as outlined in the DeepMind section. In addition to GA and NS, the researchers implement another task which they call Image Hard Maze. This task is meant to demonstrate the problem of local optima, also known as deception, in reinforcement learning. This happens when the agent tends towards exploitation rather than exploration by greedily following a particular sequence of actions that initially lead to high rewards but eventually lead to a dead end. As hypothesized in their paper, algorithms that spend more time exploring the environment, like GA and NS, are able to overcome this trap.

Source code available at https://github.com/uber-research/deep-neuroevolution

Wall-clock benchmarks

Performance-wise, Uber's code is significantly faster than any of the solutions that have been discussed previously. All hyperparameters and targets being the same, OpenAI's code can be used to train an agent in 1 hour, using a total 720 CPU cores. After Uber implemented the algorithms to run on a GPU, the same task takes 4 hours *on a single modern desktop* with 4 GPUs and 48 CPU cores. This is the reason why the current research is based on Uber's code published as part of [5].

4. Investigation

The current section outlines the work that has been done as part of this research after considering the current literature and state-of-the-art approaches to DRL and MTL.

Limitations and core principles

Several limitations and core principles have been adhered to and followed when approaching the objective of this research.

After reviewing the current implementations of similar work it is evident that DRL research is computationally expensive. The hardware resources and time it takes to complete a single training iteration, or an experiment, coupled with the deadline of the project have influenced the solution design and approach taken. Because of this, the scope has been limited to a proof of concept rather than an extensive investigation.

As a core principle, the path of least resistance in terms of source code changes and development has been taken on purpose. A minimal amount of code has been altered and introduced in order to achieve the objective of the research. As a consequence, these changes are easy to review, document and build upon. The experiments are easy to reproduce and extend.

Approach

From the overview of the current DRL implementation by DeepMind, OpenAI and Uber, it is obvious that the GPU-enabled, distributed code base published by Uber as part of [5] is orders of magnitude faster in terms of execution time. Moreover, both OpenAI and Uber have an implementation of the ES algorithm, only difference being that the older code produced by OpenAI requires hours to run on a number of servers whereas Uber's implementation produces the same results much quicker while using a single GPU-enabled server. Therefore, for the purposes of a faster development and testing cycle, a decision was made to use Uber's GPU-enabled codebase from [5] as a starting point.

The selected codebase features different GPU-enabled learning algorithms: GA, NS and ES. Adhering to the core principles outlined in the previous section, a decision was made to focus on a single algorithm - ES. The objective was therefore reduced to making the code changes necessary to support MTL in the context of a DNN that has been optimised using ES.

MTL techniques that can be applied in a DNN context were then reviewed. Both Hard Parameter Sharing and Soft Parameter Sharing (as described in the Literature review section) require changes to the DNN model's architecture. In Soft Parameter Sharing, the input layer of the resulting DNN is a concatenation of the input vectors of the separate tasks. Due to the nature of the Atari games and the preprocessing involved, different games are reduced to an input with a common format - a grayed out 84x84 pixel image as described in the Preprocessing section of the Overview of the Deep Reinforcement Learning implementations. This meant that the resulting DNN could have a single input layer which consists of a 84x84 = 7056 element vector that can be used to accommodate inputs from two distinct games - that is, the input layer of the original DNN architecture proposed by DeepMind and used by

OpenAI and Uber need not change. Furthermore, Soft Parameter Sharing also requires changes to the hidden layers and the output layer of the DNN. In contrast, Hard Parameter Sharing only alters the output layer of a DNN - it does not require alterations to the hidden layers and, as argued above, extending the input layer is not needed in the case of Atari games. It appeared that Hard Parameter Sharing can be easily implementing by extending the output layer of the DNN to accommodate the actions of the two particular games. Therefore, a decision was made to use Hard Parameter Sharing for the implementation since it was estimated to fit more naturally in the codebase and require less time to develop.

As already pointed out in the Literature review section, when considering MTL, higher model performance is to be expected given that the tasks which the model is attempting to learn are related. In the context of the Atari games, this means finding a pair of games that are visually similar and have common action spaces. A common feature of the Atari games is that all actions are discretized, for example action=0 is the “do nothing” action, or no-op. The rest of the game actions are positive integers and are game-dependent. This meant that, perhaps, in the set of Atari games, there could be at least two games that are sufficiently similar from a visual perspective, allow the agent to perform similar actions, and have similar objectives that the agent is meant to achieve in order to maximize its reward (game score). This raises the question: are there any such two games? This question prompted a review of the Atari games available to the OpenAI Gym emulator.

Given that two sufficiently related Atari games exist, and are supported by the OpenAI gym emulator, would simplify the implementation of MTL’s Hard Parameter Sharing significantly. Not only the input and hidden layers of the DNN can remain the same, but the output layer need not change either.

The problem of implementing MTL is therefore reduced to loading two games simultaneously during training phase (rather than one), using the same ES algorithm implementation, and obtaining the same DNN, from an architectural standpoint, as the DNNs obtained by running the same process twice for two separate games. The only difference would be that this DNN would allow an agent to separately play the two games that it has been trained on.

Games

As described previously, the two games chosen for the experiments must be sufficiently similar for MTL to prove beneficial. The code base uses OpenAI Gym [29] for emulating them, therefore the pool of available Atari 2600 games to choose from is already restricted to what is supported by OpenAI Gym. At the time of writing, there are 63 such games and they are very diverse in terms of their environments, action spaces and objectives. After a thorough examination, however, two games stood out as potential candidates.

Space Invaders

Space Invaders is a classic two-dimensional arcade shooter game in which the player (agent) controls a laser cannon by moving it horizontally across the bottom of the screen and firing at descending enemy ships. The objective of the game is to defeat a wave of enemy ships that move horizontally back and forth across the screen as they advance toward the bottom of the screen, closing in on the laser cannon. The player earns points (rewards) by shooting invading ships with the laser cannon. As more enemy ships are shot down, their speed is increased and defeating an entire wave of ships produces the next such wave. This loop continues until the agent is eventually destroyed, or until the current wave reaches the bottom of the screen. A special type of ship occasionally appears and moves at the top of the screen and provides additional bonus points (reward) if destroyed before it disappears. The laser cannon is partially protected by several stationary defense bunkers that are gradually destroyed by numerous blasts from the aliens or player.

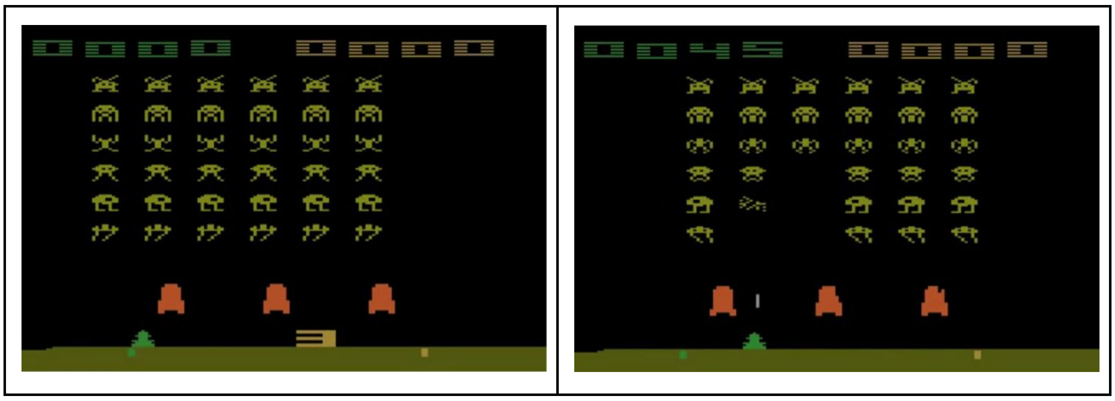
Figure 4.1.: Example frames from the game of Space Invaders.

Phoenix

Phoenix is another classic two-dimensional arcade shooter game in which the player (agent) controls a spaceship that moves horizontally at the bottom of the screen, firing upward. Enemies, typically one of two types of birds, appear on the screen above the player's ship, shooting at it and periodically, and randomly, dive towards it in an attempt to crash into it. The agent is awarded points (rewards) upon destroying a bird. The agent is equipped with a shield that can be used to destroy any of the birds that attempt to crash into it. The agent's movements are restricted while the shield is active and it must wait a few seconds before using it again. The game has separate levels that are advanced sequentially and the speed and unpredictability of the birds increases with those levels. The game ends when the agent's ship is eventually destroyed.

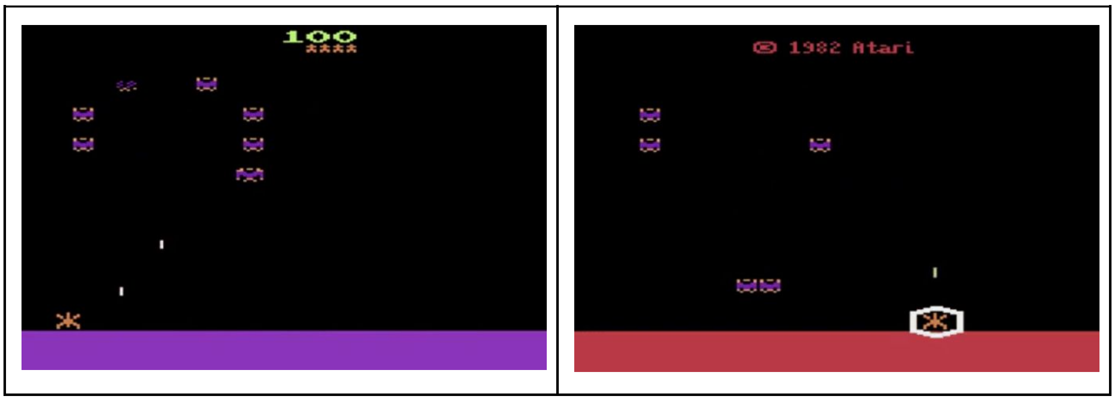

Figure 4.2.: Example frames from the game of Phoenix. Notice the agent's shield on the right.

Reward structure and episode specifics

Different games have different objectives, different reward structures and different termination conditions. In the case of Space Invaders and Phoenix, there are some similarities between those, but there are also differences. For example, shooting an alien ship in Space Invaders results in increasing the agent's score by 5 points whereas shooting a bird in Phoenix increases the agent's score by 20-80 points, depending on the type of bird. The agent in Phoenix is additionally rewarded when progressing from one level to another. In general, even though the score is measured in points in both games, their scale is different.

Bridging RL and Atari games terminology, the act of an agent playing one full game iteration is called an *episode*. Here, the game termination conditions between Space Invaders and Phoenix also differ. In Space Invaders, the agent starts with 3 lives whereas in Phoenix it starts with 5. The agent being "killed" in the game translates to a negative reward, or a penalty. When the agent has no more lives left and is killed, the game terminates. As a consequence, the time an agent spends playing the different games is, on average, different.

The details described above have implications to the reinforcement learning algorithms. Naturally, most algorithms will tend to seek higher rewards gained during episodes of shorter length. Consider how such game specifics are likely to affect a MTL model - it is reasonable to expect that the model will be biased toward one of the games - the one that ultimately results in a higher score (reward).

In order to minimize the effects of these differences, during training, all positive rewards (i.e. shooting an alien ship or a bird) are translated to 1 and all negative rewards (i.e. losing a life) are translated to -1. Furthermore, a hard limit of 5,000 timesteps (frames) is set for each episode. If the current game has not finished within 5,000 timesteps, it is terminated and the current score and state/action sequence are fed back to the model, just as if the game had indeed terminated due to specific game rules. These measures are in line with the original design and implementation by DeepMind in [2].

Action spaces

Space Invaders and Phoenix are sufficiently related in terms of environment and objective. That is, there is an agent (ship or cannon) that tries to shoot enemy vessels while simultaneously trying to not be shot down. One problem remained: those two games allow the agent to perform slightly different actions that allow it to achieve its objective. Below is a table that shows the available actions for the two games. Note how they are partly aligned, but still differ.

ACTION	**Space Invaders**	**Phoenix**
0	Do nothing	Do nothing
1	Fire	Fire
2	Move right	Move right
3	Move left	Move left

4	Fire and move right	Shield
5	Fire and move left	N/A

Table 4.1.: List of possible actions available to agents playing each game

Going back to MTL techniques, a decision had already been made to use Hard Parameter Sharing because its implementation was deemed simpler than the one of Soft Parameter Sharing. Implementing Hard Parameter Sharing in the Space Invaders and Phoenix case entails modifying the already existing DNN output layer to contain the concatenation of the two games' action spaces (see Figure 4.3. below).

This way, the input layer will stay the same, because the inputs from both games are the same from a structural perspective (both are 7056 element vectors). The hidden layers can also remain unmodified while at the same time provide Parameter Sharing for the MTL part. So a small change to the DNN architecture is required – in the output layer. Implementing the change in the output layer of the DNN, however, meant that the action labels for the second game, whichever is chosen to be second, need to change and be offset by the number of actions available in the first game. For example:

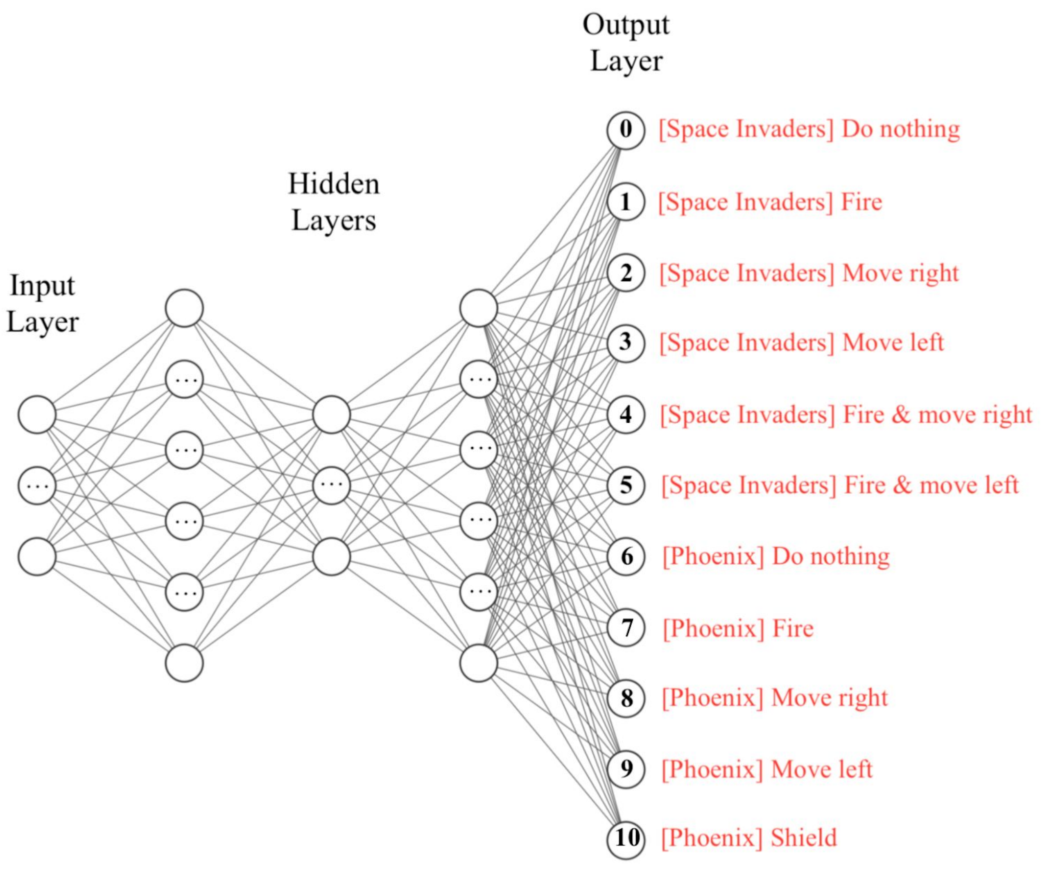

Figure 4.3.: Example of a Hard Parameter Sharing layout output layout

This also meant modifying the agent's actions on-line as they are performed during training and testing. Instead of playing each game directly after training, the agent would need a translation layer to re-map its actions accordingly. The implications of this would be a slowed down learning process and a complicated implementation.

This raises the question: can such action re-mapping be avoided in the context of MTL for Atari games?

The answer is yes, in the special case of Space Invaders and Phoenix, but it involves limiting the agent's capabilities in both games. It is reasonable to expect that imposing such limitations on the agent will reduce its overall game score - this will be investigated in the Analysis of results section. For the remaining part of this document, limiting the agent's capabilities will be referred to as "capping".

In line with the core principles described earlier, a decision was made to cap and align the action spaces of both games in such a way that they match completely. That is, during training and testing phase, the agent is oblivious to particular actions. Since actions 0..3 are the same in both games, these have been kept intact, whereas actions 4 and 5 have been forbidden. This decision simplifies the task and the DNN architecture substantially. Now, the output layer if the DNN need only have four neurons, 0-3, the input layer stays the same and the hidden layers are shared. In a sense, the output layer is also shared between both games.

An example architecture of the capped multi-task DNN looks as follows:

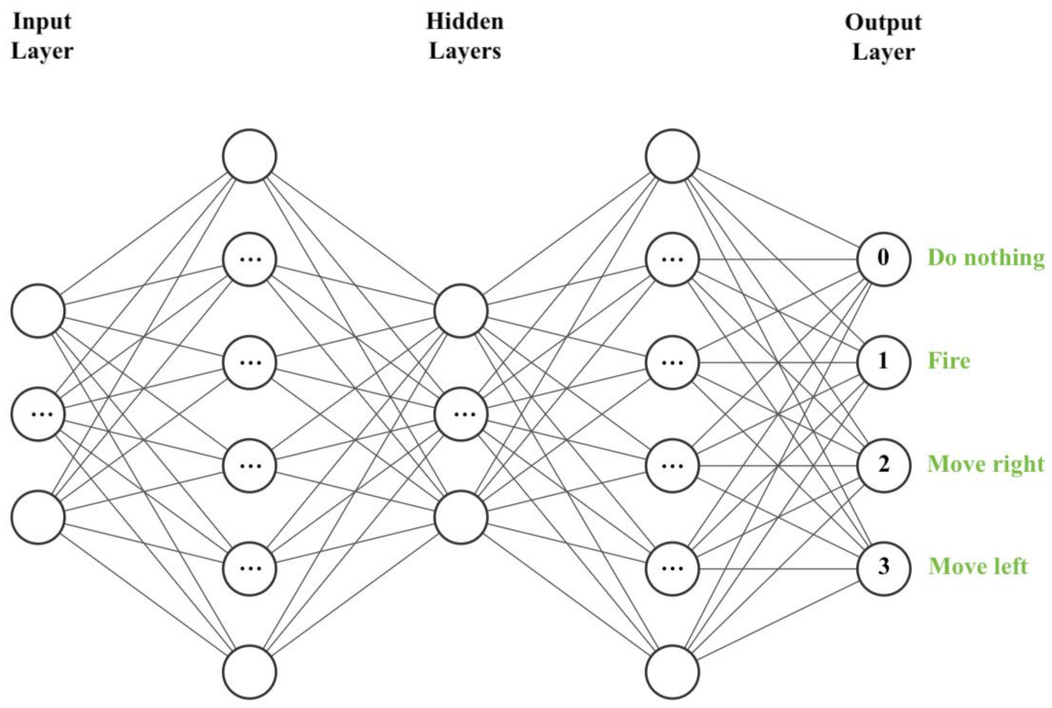

Figure 4.4.: An alternative output layer combining actions from both games

The architecture displayed above is the one used in the implementation that is part of the current document.

Implementation - Uber, single-task

It is useful to discuss Uber's GPU ES implementation before going into the changes to the code that enabled MTL.

Libraries

The core libraries used by Uber are *tensorflow*, *numpy*, *openai_gym, threading* and *queue*. The researchers from Uber have built some additional tooling on top of these libraries to support for GA and ES models.

Tensorflow [38] is a system for large scale ML that uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. The key features of Tensorflow are that those graphs can then be processed on CPUs, GPUs and TPUs, as well as on standalone servers and large clusters with many servers - without changes to code. Tensorflow is a very low level library that essentially translates Python routines to C code, for performance and compatibility purposes. As such, it is very often used in conjunction with NumPy.

NumPy [39] is a Python numerical library that provides a floating point array data structure along with a number of routines that handle array, set, matrix and other operations. Like Tensorflow, NumPy is a Python wrapper around C code. When using Python, large-scale matrix manipulations are normally performed using NumPy arrays for performance reasons.

OpenAI Gym [29] has been discussed earlier in this document. It is a Python wrapper around Atari games and other environments, such as MuJoCo [27], and its aim is to provide a platform that aids the development and testing of RL algorithms.

Python3's *threading* module provides thread, lock, semaphore, event, and other types of objects that allow the development of programs that run asynchronously. The *threading* module provides an essential piece of functionality necessary for the asynchronous nature of the agent training process.

Python3's *queue* module implements multi-producer, multi-consumer queues and is especially useful in conjunction with the *threading* module when information must be exchanged between different threads. The use of queues is also central to Uber's code base since the agent is trained by means of asynchronous workers, and the pool of these workers is a queue structure.

Program flow

What follows is a brief overview of the code found in *es.py* from Uber's code published as part of [5]. This is the main script that trains an ES model and also the skeleton for the *mtes.py* script that was developed as part of the current research in order to implement the new MTL model.

The program begins by loading its configuration which contains the name of the game that an agent will be trained to play, along with training-specific parameters and hyperparameters. The DNN model is then instantiated using Tensorflow and NumPy, as well as the random shared noise table that had been discussed previously. Finally, before training can begin, a queue of 64 concurrent worker processes are created with each of them loading the particular game. Initial performance of the agent is evaluated - this is Iteration 0 which is derived from the random shared noise table.

The program then enters an infinite loop that is broken out of on two conditions: the number of maximum training timesteps have been reached, or there had been an error the program could not recover from. Every step in this loop is treated as an *Iteration*. In every iteration, each worker process plays the game independently, using its own offspring of the policy obtained from the model evolved by the previous iteration. After a sufficient number of *Episodes* (5000 by default) have been played in total by all agents, their results are evaluated in a number of test games (200 by default) and the model is updated, ending the current *Iteration*.

After the end of each *Iteration*, its resulting model is exported to a separate file that can later be loaded by an agent to play the game.

Configuration

The configuration loaded by the program is stored in a JSON [40] file under the *configurations* directory. There are just a handful of configuration parameters and a few of them are worth outlining here:

- **game**: the game that the agent is to be trained on. I.e., "space_invaders" or "phoenix". This name must correspond to a game supported by Atari Gym.
- **model**: the DNN model that will be used - this is defined as a Python3 class within the project. Default value: ModelVirtualBN
- **population_size**: how many games the workers learn on, in total, in each *Iteration*. Default value: 5000
- **num_test_episodes**: how many test games are played for evaluating population elite. Default value: 200
- **episode_cutoff_mode:** how many timesteps is a worker allowed to play a game for during training - once this limit is reached, the game is terminated and the score is recorded. Default value: 5000
- **timesteps**: training finishes when the workers play this many timesteps in total. Default value: 250e6

Output

The program produces two kinds of outputs: the trained model, after every iteration, and a continuous stream of console output with runtime information and statistical data about the iterations.

Once the program starts, it creates a temporary directory in which it stores the models as *pickle* files. Pickle files are files created by Python3's *pickle* module and they are a binary representation of serialized python objects. Since the DNN that contains the model is simply a python object of a particular class that represents the model's structure and contents, it can be safely serialized into a binary *pickle* file. A new *pickle* file is created in the designated log directory after each *Iteration* ends and updates the model. Every such pickle file can then be loaded by a separate application that can make an agent use this particular module to play the game.

The console output of the program contains information that can be used for monitoring the process, debugging potential problems and obtaining performance metrics of the model after every iteration. For a list of all such metrics output after each iteration, consider the table in List of iteration metrics as part of Appendix II. Additional information.

The metrics from these tables are used for the purposes of analysing the results of the performed experiments as part of the current research.

Implementation - Multi-Task Evolution Strategies

The implementation of a Multi-Task Evolution Strategies (MTES) algorithm involved minor changes to the configuration, program flow, and output.

Changes to configuration

A new configuration file, *mtes.json*, has been introduced. Two changes have been made there:

- The **game** parameter has been changed to **games** and contains a list of OpenAI Gym supported Atari 2600 games. It's value is the following: "games": ["space_invaders", "phoenix"].
- The **timesteps** parameter's value has been increased to 20e8 in order to lengthen the training time during experiments.

Theoretically, one can do *m*-task MTES by simply adding more than two games in the list of **games** stored in this configuration file.

Changes to program flow

The changes to the program flow that enable MTES are two-fold:

- half the workers load the first game, the other half load the second game. Queue size has been increased from 64 to 200. Each worker has a label that contains the index of the game it has loaded (0 or 1). This index corresponds to the position of the loaded game in the **games** parameter from the configuration.
- when sending metrics to console output, provide separate metrics for the two sets of workers that have loaded the two games.

In order to provide backward compatibility, a new file *mtes.py* has been created based on *es.py* and a new class called *MTConcurrentWorker* has been created that inherits all of its functionality, other than initialisation, from *ConcurrentWorker.* Two other changes involve capping the action space of the games: one is capping it on model initialisation, and another is capping it during game play so that the agent is oblivious to the actions that have been capped.

The small number and simplicity of these changes aligns well with the core principle of following the path of least resistance. Their drawbacks and limitations will be discussed in the sections that follow.

Consider the additional benefits of such a simple design. Not changing the original DNN model allows for the simultaneous training of more than two games, without any source code changes. However, the games still need to be similar and their action spaces must be aligned.

Changes to output

Having a MTL-enabled process requires the output of several additional metrics if its performance and accuracy is to be analysed. The following metrics have been preserved as part of the Iteration console log: *TestRewMean, TestRewMedian* and *TestEpCount.*

However, these are aggregated using data from both games and can therefore only be used for analysing the overall results of the agents. They do not support any breakdowns of the agent's performance on the two different games it is learning to play. In order to allow for such breakdowns, the following split of metrics has been introduced:

- Game**0**TestRewMean
- Game**0**TestRewMedian
- Game**0**TestEpCount
- Game**1**TestRewMean
- Game**1**TestRewMedian
- Game**1**TestEpCount

These metrics break down *TestRewMean, TestRewMedian* and *TestEpCount* into two categories: for game 0 and for game 1, respectively. These are calculated and gathered based on the worker game label that results are pooled from.

In order to allow for slightly more informed investigation, the following metrics have also been introduced:

- TestRewMin
- TestRewMax
- Game**0**TestRewMin
- Game**0**TestRewMax
- Game**1**TestRewMin
- Game**1**TestRewMax

These metrics will be elaborated upon in the section that focuses on analysing the results.

5. Experiment design

In order to assess the performance of the new MTES model and compare it with single-task ES, different experiments had to be carried out. The same conditions were preserved during experiment time. This involved using the same hardware and comparable configurations for the different experiments.

Five experiments were conducted in total and their conditions are outlined below. In every experiment, training continued for full 200 Iterations. The same server was used for each and every experiment. The experiments were carried out in sequence, one after the other, with no overlap between them. With the exception of standard Operating System processes, no other programs were running on the server during the time when the experiments were being carried out.

The table below outlines the major parameters of the experiments. Their results will be analysed in the section that follows.

Experiment #	**Algorithm**	**Game(s)**	**Action Space**	**Iterations**
1	ES	'space_invaders'	normal	200
2	ES	'space_invaders'	capped	200
3	ES	'phoenix'	normal	200
4	ES	'phoenix'	capped	200
5	MTES	['space_invaders', 'phoenix']	capped	200

Table 5.1.: Experiments performed as part of the current study

6. Analysis of results

The experiments produced policy files in the form of trained DNN models on successful completion of each iteration, as well as console logs with iteration-related metrics as described in the previous section. As part of the current research, an R [41] script was developed to consume the console logs from the experiment and to parse the iteration metrics into a tabular format. The charts and analysis that follow are based on this tabular data. Information on how to reproduce these charts, along with the experiments themselves, is available in Appendix I.

Outline

In the subsections that follow, an attempt is made to analyse and rationalise the results of the experiments both in isolation and in conjunction with each other. To preserve consistency in the charts, black lines were chosen to denote results from baseline experiments with normal action spaces (experiments #1 and #3), red lines - capped action spaces (experiments #2 and #4), and blue lines are the result of MTES (experiment #5).

Differences between Space Invaders and Phoenix

Experiments #1 and #3 provide a baseline for experiments #2, #4 and #5. In #1 and #3, agents learn to play Space Invaders and Phoenix throughout the course of 200 iterations. These experiments can be run using Uber's *es.py* script without doing any modifications other than using configuration parameters that allow the learning to continue for enough number of timesteps that would produce 200 or more iterations of the model.

The most obvious chart to begin with shows the agent's test score achieved during a test phase (200 episodes) after each training phase (5000 episodes).

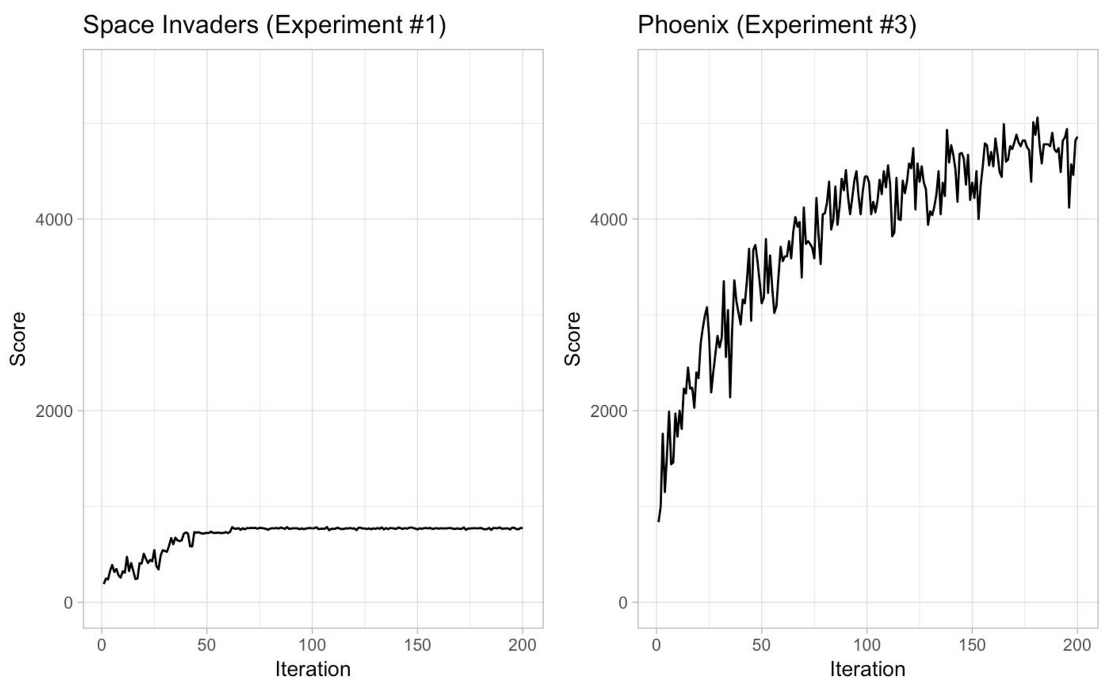

Figure. 6.1.: Scores achieved on each game from the baseline experiments

One thing that is immediately apparent is that in Space Invaders (Experiment #1), performance plateaus after around 60-70 iterations whereas in Phoenix (Experiment #3), performance continues to increase throughout all of the 200 iterations. It is unclear whether additional iterations will improve agent performance in either game. Answering this question requires performing additional experiments.

The scores achieved by the agents in both games differ substantially. Space Invaders' top score across all iterations is 837 whereas the top score reached in Phoenix is 5060. Those are the top *mean* scores in an iteration of 200 test games.

The next plot allows for a sense of how much time it takes to train any particular iteration in both games. In total, Experiment #1 took 69.05 hours to train and Experiment #3 took 84.37 hours.

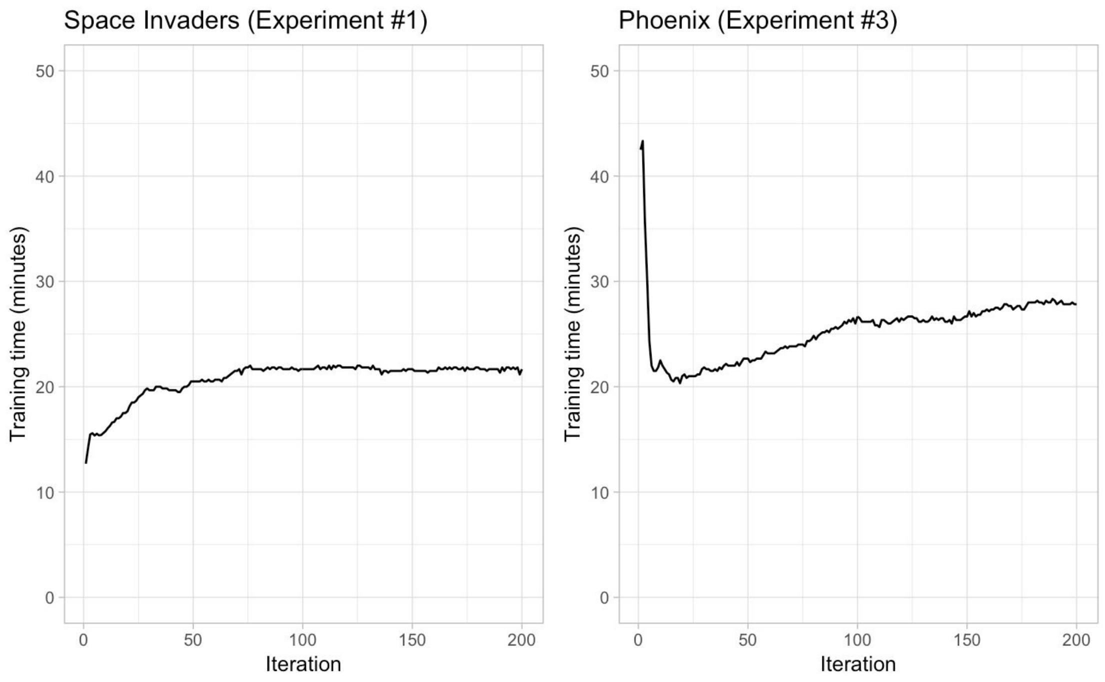

Figure 6.2: Minutes required to complete an Iteration from the baseline experiments

A notable difference is visible here as well. While Space Invaders requires an average of 16.02 minutes of training per iteration for the first 20 iterations before it settles on 21.23 minutes on average for the remaining iterations, Phoenix takes 3 times longer (85min vs. 26min) for the first couple of iterations - and still much more time per iteration on average throughout the rest of the learning process. This information points to an important difference between Space Invaders and Phoenix that will eventually affect the performance of the newly introduced MTS model.

Remember that an iteration consists of 5000 training games plus 200 test games. Each game consists of a number of consecutive timesteps. A timestep duration depends on the frameskip parameter which is by default set to 4. That is, a timestep is equal to the time it takes for the agent to receive and process 4 frames of the game. The following chart looks at how many timesteps an agent spends training, per episode, on average, for every iteration that follows.

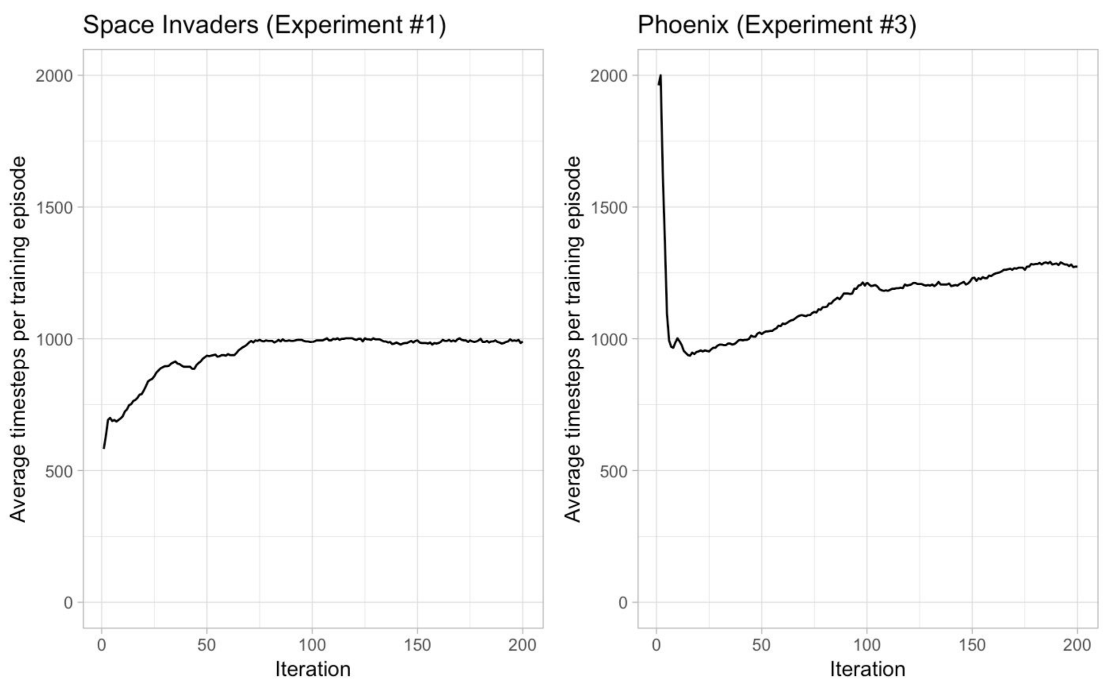

Figure 6.3.: How long gaming episodes last during each Iteration

This chart looks remarkably similar to the previous one, where training time (wall-clock) is visualised. In fact, across all of the experiments and their iterations, there is a positive *r = .998446* correlation between number of training timesteps and number of seconds the iteration took to complete. Those two metrics can therefore be treated equally.

This leads to an important conclusion. On average, the agent spends 22.21% longer playing Phoenix than Space Invaders during training. Now, recall the implementation details of the introduced MTES. Its central feature is a Queue of 100 agents playing Space Invaders and 100 agents playing Phoenix (Queue size is 64 by default but has been increased to 200 for the purposes of the experiments). An iteration begins and 5000 games must be played. 200 threads are spawned and each of them retrieves an agent from the queue: 100 of them acquire an agent that plays Space Invaders and 100 of them, an agent that plays Phoenix. However, those 100 + 100 = 200 agents play 5000 games *in total* during iteration training. Should the average game time of Space Invaders and Phoenix be the same, both groups of agents would play roughly 5000 / 2 = 2500 games *each* before the iteration ends. Knowing that a game of Phenix takes approximately 22% longer to complete than a game of Space Invaders means that, ultimately, Space Invaders is played more times, and Phoenix is played fewer times. This is an obvious source of bias for the MTES model.

Impact of capping the action spaces

The charts in this section illustrate the impact of capping the games' action spaces. Experiments #2 and #4 were designed to match the conditions of experiments #1 and #3, respectively, with the only difference being that the agents can only use actions numbered 0, 1, 2 and 3. In practice, this means that the Space Invaders agent from experiment #2 does not know how to perform actions "fire and move right" and "fire and move left", and the Phoenix agent from experiment #4 does not know how to activate its protective shield. Earlier, a hypothesis was formulated stating that disabling the agents in this manner will prove detrimental to their average score.

Again, the obvious place to start is by comparing average game scores between agents using normal and capped action spaces.

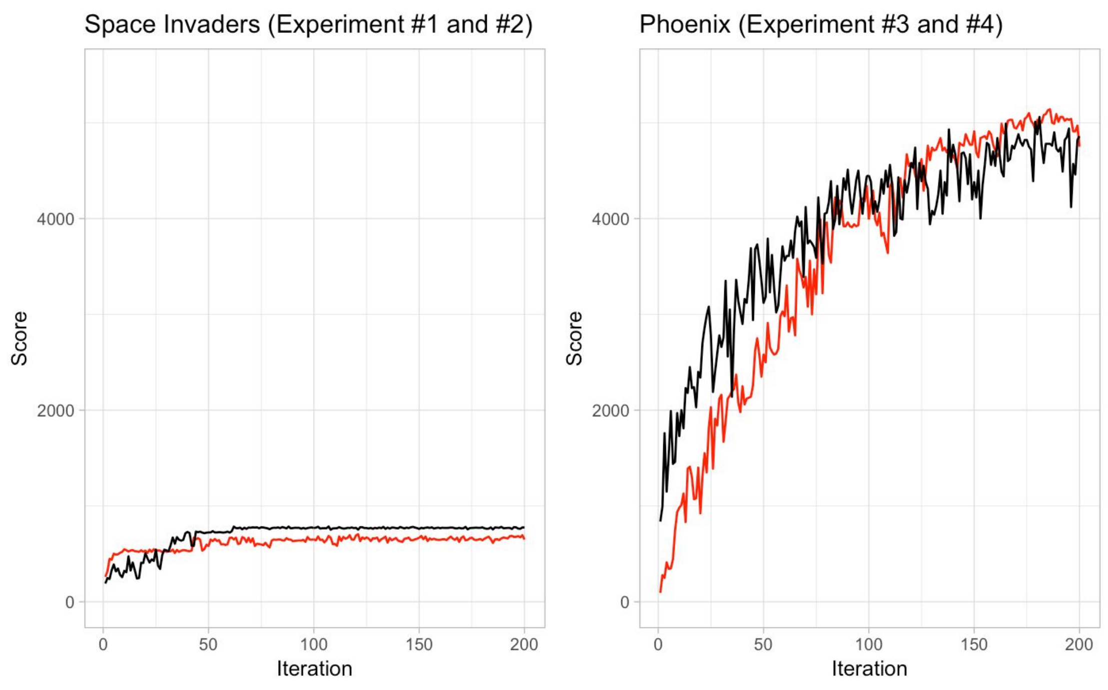

Figure 6.4.: Comparison of game performance between baseline and capped agents

As stated in the beginning of this section, black lines denote normal action spaces and red lines denote capped action spaces.

Interestingly, in the case of Space Invaders, capping the action space proves beneficial early in the learning process (Iterations 1-25) and harmful in later iterations. Conversely, capping the action space in Phoenix has a negative impact on the agent’s score during the first 125 iterations, and a positive impact later on in the learning process. This is counterintuitive given that the agent in Phoenix is deprived of its ability to shield itself. It may be the case that after iteration 125, the agent learns to avoid birds that charge towards it by moving left or right rather than activating its shield, but this assumption has not been validated.

How about training time - does the reduction of available actions have an impact on it?

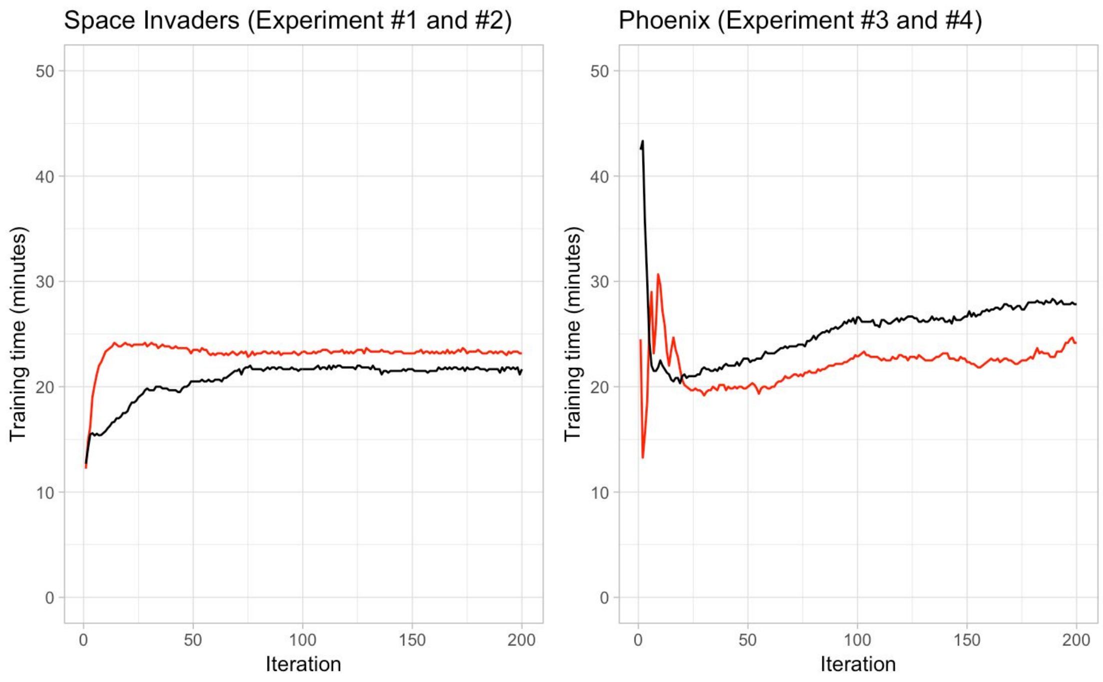

Figure 6.5.: Comparison of training time per iteration between baseline and capped agents

Here, the results pose more questions than they answer. It appears that Space Invaders, when capped, takes more time to train compared to when the agent has access to all actions. Combining this with the knowledge provided by the previous chart, it is clear that longer training times do not necessarily equate to higher scores. Perhaps the agent in Space Invaders manages to survive longer during a game, but lacks the flexibility required to attain higher scores, e.g. by moving left/right and firing at once in order to shoot down an alien. Conversely, in the case of Phoenix, the capped agent takes less time to train (than the normal agent) but manages to attain higher scores.

Furthermore, even though the training time per iteration for both capped agents (Space Invaders and Phoenix) look similar, each iteration takes 21.95min and 23.67min, respectively, on average, throughout all of the 200 iterations. This means that Phoenix still takes longer to train than Space Invaders. Moreover, this gap is even larger in the first 25 iterations where the time is 19.32min per iteration on average for Space Invaders and 23.67min per iteration on average for Phoenix.

This difference in training times of the capped agents, even though it is less pronounced, is still significant and is still likely to affect the MTES model performance, as this model uses two capped agents simultaneously.

MTES compared to baselines

It is now time to compare the MTES model introduced with the current research. This model has trained, in a multi-task fashion, a single agent to play both Space Invaders and Phoenix using a capped action space and an ES model.

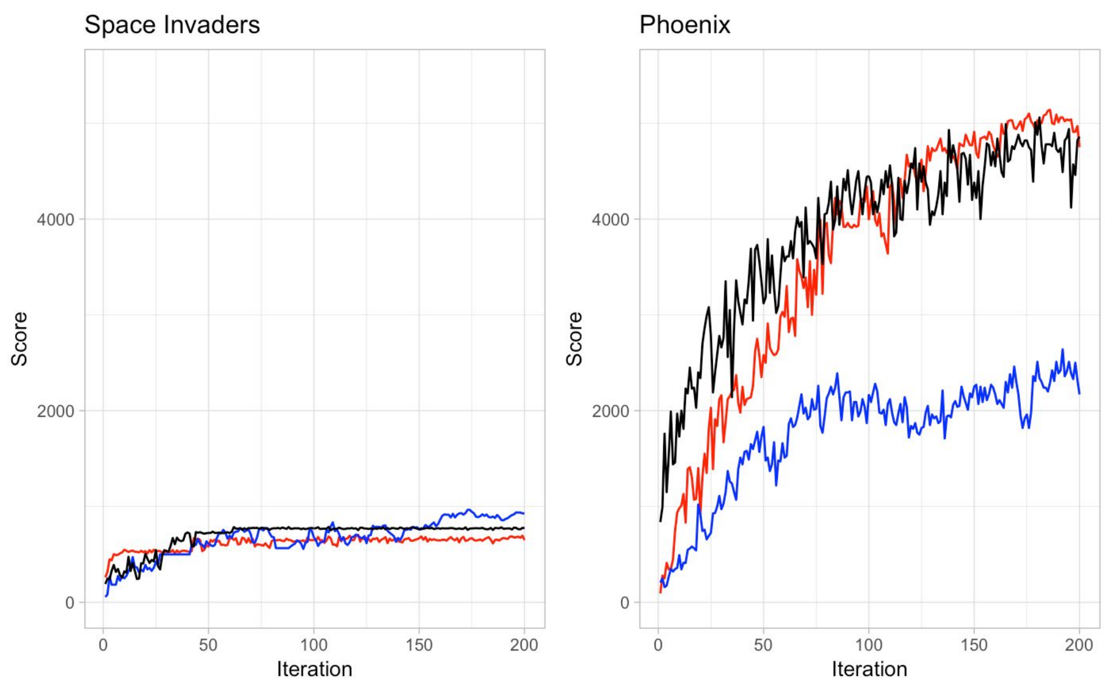

Figure. 6.7.: Comparison of game performance between baseline (black), capped (red) and multi-task (blue) agents

In the case of Space Invaders, the model's performance initially is comparable to that of a single-task non-capped agent. It then struggles to perform as well during the middle of the training process until at iteration 150 it is able to escape the plateau and consistently achieve higher scores for the remaining of the training. Currently, the reasons behind this feat are unclear. It may be that the agent has benefited by generalizing features from the other game, however this is only a speculation that cannot be validated without further research.

The multi-task agent's low performance on Phoenix should not come as a surprise by now. As pointed out earlier, the multi-task agent is likely to have played many more games of Space Invaders than games of Phoenix during its entire training process. It is then natural that its performance will be lower than the one of the previous two models which have had the privilege of making all of their workers play Phoenix.

This imbalance between number of Space Invaders and Phoenix games played is not merely an analytical artifact. As part of the implementation of MTES, two new metrics were added in order to establish how many episodes of each game are played during the testing phase of each iteration. Consider the following chart.

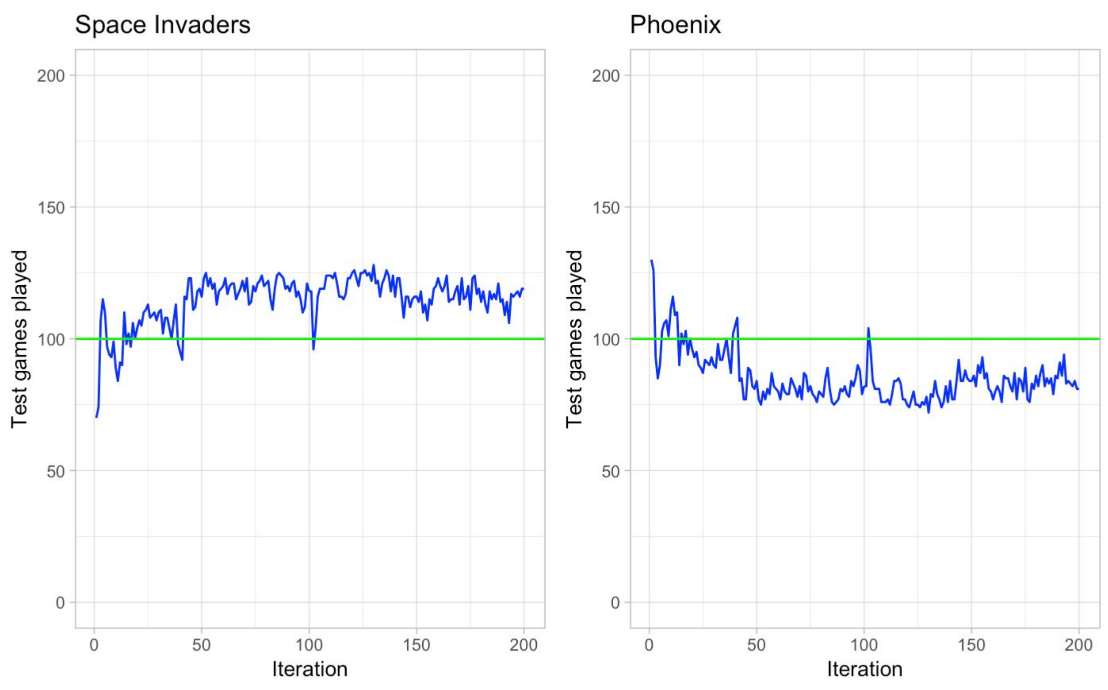

Figure 6.8.: imbalance between number of games played in each iteration during multi-task learning

In a situation of perfect equilibrium, each game would be played 100 times during the test phase of each iteration. The chart therefore validates the assumption made earlier that Phoenix will be played fewer times during MTES.

Lastly, it is worth mentioning that the MTES agent took a total of 70.27 hours to train whereas training Space Invaders and Phoenix separately, using single-task ES and normal action spaces (experiments #1 and #3), took 69.05 and 84.37 hours respectively. MTES has achieved a 45% reduction of training time compared to the total time it takes to train the two single-task ES models.

7. Known issues

This section outlines a number of issues of the MTES implementation that can be resolved by changing model design and project code.

Naive implementation

First and foremost, it must be acknowledged that the current implementation of MTES is very naive. Aligning the games' action spaces by capping them allows for a single-task model to be used in the context of multi-task learning, but for more realistic and cleaner implementation, Hard Parameter Sharing and Soft Parameter sharing model designs are worth testing out.

Separate games sharing single queue

Maintaining a single queue, rather than two separate ones, has had a detrimental effect to the model's overall performance. What is worse, this imposes a bias that the developer has no control over since it is a function of game play time, as discussed in the previous section. A better design would consist of standalone queues for separate games which would allow for equal episodes of the games being played out during training and testing. It is likely that this approach will remove the bias and introduce balance to the model, enabling it to generalise better across multiple games.

Metrics not granular enough

The default performance metrics that are being gathered are enough to support a basic analysis but they must be made more granular. Having only *min, mean, median* and *max* metrics of game scores are insufficient should a researcher decide to plot the distribution of training/test scores of a particular game after a particular iteration. In effect, the metrics should be output on every episode rather than on every iteration. For example, for each training/test episode, the following could be recorded:

- total game score
- total timeframes
- a breakdown of what action is taken by the agent on which timestep
- a breakdown of when an agent loses a life during gameplay (which timeframe it died in)

Having metrics recorded on such a granular level will greatly improve researchers' ability to analyse the training process in depth.

8. Conclusion

The current research aimed towards the development and testing of a naive Multi-Task Learning Evolution Strategies algorithm that can solve the Reinforcement Learning problem of obtaining a single model that enables an agent to play two different Atari games. The resulting model is simple by design which, in turn, allowed for straightforward and non-ambiguous testing. This enabled the three newly developed experiments to be compared with two similar baseline experiments in a clear manner.

After performing the experiments and analysing the results, it is clear that Multi-Task Reinforcement Learning is a viable alternative to single-task learning, when:

- applied to sufficiently similar environments
- trained for a sufficient number of iterations
- reduction in learning time is required
- suboptimal task performance can be tolerated

Overall, the developed algorithm reduced the time required to learn both games by more than 45%, ultimately achieving better performance on one of the games and adequate performance on the other game. The multi-task model is able to better generalize common features of the two environments. Furthermore, the experiments made clear that sometimes models (both single- and multi-task) can benefit from stripping certain actions from the agent's repertoire - a notion which, at first, contradicts conventional wisdom.

Further work

In order to move this line of research forward, several possibilities can be explored further. It is advisable to first address what was outlined in Known issues before trying out other more sophisticated implementations. The following priorities can be followed:

1. Solve the metric granularity issue. More granular metrics are the foundations for deeper analyses.
2. Create and use dedicated queues for the agents that play separate games. This way, a known source of bias towards one of the tasks will be eliminated.
3. Experiment with Hard Parameter Sharing and Soft Parameter Sharing. It is likely that implementing a Multi-Task learner using these techniques will yield better results.

Once these issues have been resolved, one can potentially fit other algorithms into the same multi-task framework. Promising algorithms to start with would be the ones that have already been studied and reviewed earlier in this document: DQN, GA and NS.

References

1. Sutton, R.S. and Barto, A.G., 1998. *Reinforcement learning: An introduction (Vol. 1, No. 1)*. Cambridge: MIT press.

2. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S. 2015. *Human-level control through deep reinforcement learning*. Nature, 518(7540), p.529.

3. Salimans, T., Ho, J., Chen, X., Sidor, S. and Sutskever, I., 2017. *Evolution strategies as a scalable alternative to reinforcement learning.* arXiv preprint arXiv:1703.03864.

4. Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O. and Clune, J. 2017. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. arXiv preprint arXiv:1712.06567v1.

5. Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O. and Clune, J. 2017. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. arXiv preprint arXiv:1712.06567v3.

6. Bellman, R. 1957. *Dynamic Programming*. Princeton University Press, Princeton, N. J., 1957, MR 0090477

7. Bellman, R., 1957. *A Markovian decision process.* Journal of Mathematics and Mechanics, pp. 679-684.

8. Thorndike, E. L. 1911. *Animal Intelligence*. Hefner, Darien, Conn.

9. Minsky, M. L. 1954. *Theory of Neural-Analog Reinforcement Systems and its Applications to the Brain-Model Problem*. PhD thesis, Princeton University

10. Farley, B. G. and Clark, W. A. 1954. *Simulations of self-organizing systems by digital computer*. IRE Transactions on Information Theory, 4:76-84

11. Michie, D. and Chambers, R. A. 1968. *BOXES: An experiment in adaptive control.* In Dale, E. and Michie, D., editors, Machine Intelligence 2, pages 137-152. Oliver and Boyd.

12. Samuel, A. L. 1959, *Some studies in machine learning using the game of checkers.* IBM Journal on Research and Development, pages 210-229. Reprinted in E. A. Feigenbaum and J. Feldman, editors, Computers and Thought, McGraw-Hill, New York, 1963.

13. Klopf, A. H. 1972. *Brain function and adaptive systems--A heterostatic theory.* Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA. A summary appears in Proceedings of the International Conference on Systems, Man, and Cybernetics, 1974, IEEE Systems, Man, and Cybernetics Society, Dallas, TX.

14. Sutton, R. S. and Barto, A. G. 1981a. *An adaptive network that constructs and uses an internal model of its world.* Cognition and Brain Theory, 3:217-246.

15. Hawkins, R. D. and Kandel, E. R. 1984. *Is there a cell-biological alphabet for simple forms of learning?* Psychological Review, 91:375-391.

16. Tesauro, G. J. 1986. *Simple neural models of classical conditioning.* Biological Cybernetics, 55:187-200.

17. Sutton, R. S. and Barto, A. G. 1987. *A temporal-difference model of classical conditioning.* In Proceedings of the Ninth Annual Conference of the Cognitive Science Society, Hillsdale, NJ. Erlbaum.

18. Watkins, C. J. C. H. 1989 *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England.

19. Koomey, Berard, Sanchez, and Wong 2011. *Implications of Historical Trends in the Electrical Efficiency of Computing.* IEEE Annals of the History of Computing, 33, 3, 46--54.

20. Warren S. McCulloch, Pitts, W. 1943. *A logical calculus of the ideas immanent in nervous activity.* University of Illinois, College of Medicine, Department of Psychiatry at the Illinois Neuropsychiatric Institute, University of Chicago, Chicago, U.S.A.

21. Rumelhart, D. E., Hinton, G. E., and Williams, R. J. 1986. *Learning representations by back-propagating errors.* Nature, 323, 533--536.

22. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. 1998. *Gradient-based learning applied to document recognition*. Proceedings of the IEEE. 86 (11): 2278–2324. doi:10.1109/5.726791.

23. Pomerleau, Dean A. 1991. *Efficient training of artificial neural networks for autonomous navigation.* Neural Computation 3, no. 1 (1991): 88-97.

24. Stafylopatis, A. and Blekas, K. 1998. *Autonomous vehicle navigation using evolutionary reinforcement learning*. European Journal of Operational Research, 108(2), pp.306-318.

25. Tesauro, G. 1995. *Temporal difference learning and TD-Gammon.* Communications of the ACM, 38(3), 58-68.

26. Thrun, S. 1995. *Learning to Play the Game of Chess.* Advances in Neural Information Processing Systems 7.

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

28. Bellemare, M.G., Naddaf, Y., Veness, J. and Bowling, M. 2013. *The arcade learning environment: An evaluation platform for general agents*. Journal of Artificial Intelligence Research, 47, pp.253-279.

29. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. 2016. *Openai gym.* arXiv preprint arXiv:1606.01540.

30. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton 2015. *Deep Learning.* Nature, 521(7553):436–444.

31. Bellemare, M. G., Veness, J. & Bowling, M. 2012. *Investigating contingency awareness using Atari 2600 games.* Proc. Conf. AAAI. Artif. Intell. 864–871

32. Michalewicz, Z., 1996. *Evolution strategies and other methods*. In Genetic Algorithms + Data Structures = Evolution Programs (pp. 159-177). Springer, Berlin, Heidelberg.

33. Sehnke, F., Osendorfer, C., Rückstieß, T., Graves A., Peters J., Schmidhuber, J., 2010. *Parameter-exploring policy gradients*. Neural Networks, 23(4):551–559.

34. Abu-Mostafa, Y.S., 1990. *Learning from hints in neural networks*. J. Complexity, 6(2), pp.192-198.

35. Caruana, R., 1997. *Multitask learning*. Machine learning, 28(1), pp.41-75.

36. Caruana, R. (1993). *Multitask learning: A knowledge-based source of inductive bias.* In Proceedings of the Tenth International Conference on Machine Learning.

37. Baxter, J. (1997). *A Bayesian/information theoretic model of learning to learn via multiple task sampling*. Machine Learning, 28:7–39.

38. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. and Kudlur, M., 2016, November. *Tensorflow: a system for large-scale machine learning.* In OSDI (Vol. 16, pp. 265-283).

39. Walt, S.V.D., Colbert, S.C. and Varoquaux, G., 2011. *The NumPy array: a structure for efficient numerical computation.* Computing in Science & Engineering, 13(2), pp.22-30.

40. Crockford, D., 2006. *The application/json media type for javascript object notation (json)* (No. RFC 4627).

41. Team, R.C., 2013. *R: A language and environment for statistical computing.*

Appendices

I. Replicating the experiments

Replicating the experiments requires specific hardware and software. To replicate the baseline experiments, #1 and #3, one needs to obtain Uber's code from https://github.com/uber-research/deep-neuroevolution and follow the instructions in the README.md file to install the necessary software dependencies like a separate python3 environment, TensorFlow and other required libraries. With this setup performed, the experiments can be initiated by using the following commands from within the *gpu_implementation* directory of the project:

Command	**Experiment**
python es.py configuration/es_spaceinvaders_config.json	#1
python es.py configuration/es_phoenix_config.json	#3

However, the configuration files will be missing from the repository. These are their contents:

es_spaceinvaders_config.json

```
{
  "game": "space_invaders",
  "model": "ModelVirtualBN",
  "num_validation_episodes": 30,
  "num_test_episodes": 200,
  "population_size": 5000,
  "timesteps": 20e8,
  "episode_cutoff_mode": 5000,
  "return_proc_mode": "centered_rank",
  "l2coeff": 0.005,
  "mutation_power": 0.02,
  "optimizer": {
    "args": {
      "stepsize": 0.01
    },
    "type": "adam"
  }
}
```

es_phoenix_config.json

```
{
  "game": "phoenix",
  "model": "ModelVirtualBN",
  "num_validation_episodes": 30,
  "num_test_episodes": 200,
  "population_size": 5000,
  "timesteps": 20e8,
  "episode_cutoff_mode": 5000,
  "return_proc_mode": "centered_rank",
  "l2coeff": 0.005,
  "mutation_power": 0.02,
  "optimizer": {
    "args": {
      "stepsize": 0.01
    },
    "type": "adam"
  }
}
```

In order to replicate the rest of the experiments, one must obtain the code used as part of this research from https://github.com/deyandyankov/deep-neuroevolution/

Then, to run experiments #2, #4 and #5, respectively, one must run:

Command	**Experiment**
python es.py configuration/es_spaceinvaders_config.json	#2
python es.py configuration/es_phoenix_config.json	#4
python mtes.py configuration/mtes.json	#5

In terms of hardware, all of the experiments require at least 4GB of RAM and a GPU available on the server.

In order to avoid any other software interfering with the experiments, it is advisable that they are run in separation and sequentially, while making sure no other programs are hogging the available hardware resources.

Replicating the charts from the Analysis of results section requires running the *gpu_implementation/analysis/explore.Rmd* R notebook file. After installing R, the following packages need to be installed for the notebook to work:

Package	**Installation command**
grid	install.packages(“grid”)
tidyverse	install.packages(“tidyverse”)
ggthemes	install.packages(“ggthemes”)

II. Additional Information

List of iteration metrics

Upon completion of every training iteration, a list of metrics is printed to the standard output. An example is provided below.

Iteration	1
MutationPower	0.02
TimestepLimitPerEpisode	5e+03
PopulationEpRewMax	4.09e+03
PopulationEpRewMean	379
PopulationEpRewMedian	280

PopulationEpCount	5e+03
PopulationTimesteps	8.55e+06
TestRewMean	261
TestRewMedian	180
TestEpCount	200
TestEpLenSum	3.34e+05
InitialRewMax	2.41e+03
InitialRewMean	611
InitialRewMedian	400
TimestepsThisIter	8.55e+06
TimestepsPerSecondThisIter	3.65e+03
TimestepsComputed	8.55e+06
TimestepsSoFar	8.55e+06
TimeElapsedThisIter	2.23e+03
TimeElapsedThisIterTotal	2.34e+03
TimeElapsed	2.23e+03
TimeElapsedTotal	2.76e+03