City University London MSc in Artificial Intelligence

Project Report

2020

THE ROLE OF GAME DIFFICULTY, IN TRAINING DEEP REINFORCEMENT LEANRNING AGENTS FOR CLASSIC ARCADE GAMES

Student name: Vasilis Kotsos

Student number: 190033787

Supervised by Salako Kizito

December/18/2020

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed: VASILIS KOTSOS

Abstract

This piece of work investigates the role of difficulty, for training artificial agents in the Atari games setting. More specifically this piece of work tackles the question of whether an agent trained on a higher difficulty of a game, can perform well on the easy setting of the same game. Two games are considered for experimentation. The first is 'Pong' and the second is 'Demon Attack'. Several, agent architectures are tested, all of which using the Deep Q-learning algorithm. The results of the experimentation show that there is not definitive answer to this question, with the results of Demon Attack supporting the intuitive idea that agents trained on the harder setting should perform better on the easy setting than the ones trained solely on the easy setting, whereas the results of Pong contradict this statement. The special case of curriculum learning is also examined, and its effects are studied throughout. This special training routine shows that agents can benefit from a mixed curriculum that uses both easy and hard settings during training.

Keywords: Reinforcement Learning, Deep Reinforcement learning, Deep Learning, Atari Benchmark, Q-learning, Deep Q-learning.

Contents

Chapter 1. Introduction and Objectives

1.2 Introduction to the project purpose

The explosion of deep learning over the past decade has resulted in several advancements in areas that computers had previously performed very poorly in. Computer vision, natural language processing are two areas that have been experiencing a great deal of advancement. The fuel that is driving advancements in these fields is Deep Learning (DL). Using the power of Deep Neural Networks (DNNs), new high scores are set yearly on computer visions and natural language understanding benchmarks. Over the past decade, the Reinforcement Learning (R) field has seen dramatic improvements as well. The mix of Deep Learning and Reinforcement Learning has led to stunning breakthroughs to optimal control tasks. Some of these breakthroughs include the creation of artificial agents that rival human results in Atari games[1], and perhaps the most famous breakthrough being the creation of artificial agents that achieved super-human level performance in the game of GO[2]. As a field, RL concerns itself with the creation of agents that make optimal decisions in sequential decision-making scenarios[3]. Unlike Supervised Learning (SL), which requires full supervision of an agent, by providing it with predefined labels, and unlike Unsupervised Learning (UL) which employs no supervision and provides no labels for data, RL lies somewhere in between. Instead of ground truth labels, RL employs reward signals that are used to modify the behavior of the agent during the training process. The mix between RL and DL has given birth to a new subfield called Deep Reinforcement Learning (DRL), which concerns itself with the creation of agents, that employ DNNs for function approximation. Theoretically DNNs can approximate any function, which is what

makes them such a great tool for RL. Deep Reinforcement Learning is now a vibrant field with many open topics to explore.

1.3 Project aims and objectives

The main proving ground for DRL has been the Atari arcade game suite[4], which was until recently considered impossible to achieve good performance on due to the massive complexity of the games included. The entire suite of games is now considered "solved" [5], however the role that game difficulty plays in training DRL agents is an underexplored topic. This dissertation focuses on exploring the role of game difficulty in training agents for Atari arcade games.

The main aims of the project as set out in the project proposal document are:

- Investigate the role of game difficulty in training AI for classic arcade games.
- Experiment with different training strategies and schedules and explore whether the use of harder difficulty settings during training can prove to be beneficial.
- Explore whether agents trained on high difficulty settings have an edge, compared to their counterparts trained under easier difficulty settings.

The main objectives of this dissertation are:

- Identify suitable Deep RL learning algorithms for Classic Arcade Games
- Identify suitable training strategies for the learning algorithms.
- Design and develop at least 2 learning algorithms that can exhibit good performance in at least 3 games.
- Compare and contrast RL algorithms in terms of performance and discuss the best results.
- Compare and contrast different training routines and investigate what impact the use of high difficulty setting has if any.

This dissertation explores whether an agent trained under a harder difficulty training setting can perform well when tested on the easier setting of the game. Moreover, the performance of such an agent is compared to that of an agent which was trained on the easier setting.

Beneficiaries of this piece of work include researchers in the RL field and especially Deep RL. It could prove beneficial to researchers that study how agents can generalize to a different target setting than the one they were trained on. Moreover, researchers that are interested in the Atari platform as a benchmark could find the methods used in this piece of work interesting. Lastly, of course any researcher interested in how difficulty affects the performance of an agent on Atari games, is going to benefit from this piece of work.

The starting point was to create a self-learning agent that can achieve good performance on the easy setting of the game "Pong". This was a good starting point as the game is one of the easier titles in the Atari game suite. The most challenging part of this first step is creation of a suitable training routine. Subsequently the training routine was tested on other games and higher difficulties. Multiple changes were made to the training routines, and more specifically the hyper-parameters that define the training routines. This was done so that the agents could learn a good enough policy to play the games to a high level.

No major changes to the goals or methods took place during the duration of the project. However, the scope of the project was scaled down due to the significant computational complexity that the Atari benchmark presents. The dissertation presents results on two games, instead of three.

1.4 Report structure

Chapter 2 provides the context of the dissertation. Firstly it explores the basics of RL and Q-learning, which is the main algorithm used to develop the self-learning agents, and then proceeds to introduce and motivate the use of Deep Q-learning for the creation of agents that can learn to play Atari games. Furthermore, it also covers enhancements to the Deep

Q-learning algorithm which have been used in the experiments. Lastly it introduces the Curriculum Learning (CL) methodology to training agents for RL.

Chapter 3 firstly introduces the Gym toolkit which provides the necessary interface for the ALE and a principled way of running RL experiments. The chapter then covers the training and evaluation methodology of the self-learning agents is described in detail. The results of the experiments are presented along with critical commentary.

Chapter 4 covers further evaluation of the results in a more wholistic approach which considers the goals the were set in the project proposal document. Furthermore, validity and generalizability of the results is discussed here.

Chapter 5 discusses conclusions, reflection, lessons learned from undertaking this work and proposes future directions.

Chapter 2. Context

2.1 Introduction to RL concepts.

Reinforcement Learning uses trial and error to help an agent learn a sequence of actions that maximizes the total reward. By using a system of rewards and punishment an agent learns how to behave under a given environment. This area is inspired by behavioral psychology, as the methods it uses, mimic the human experience and animal experience of learning, in other words RL is inspired by the way humans and animals learn to accomplish a task. For example, animals can be trained to choose actions that are rewarding and avoid actions that lead to a punishing outcome. The feedback they receive for their actions shapes their behavior. The same learning mechanism can be found in human, as they also learn through trial and error in their early years,

In the RL setting, an artificial agent takes an action, denoted as a, under a specific state of the environment s, which can cause the environment to change its state. When an agent takes an action, feedback is produced by the environment and is sent to the agent in the

form of a reward signal, denoted by r, along with its new state s'.The signal takes the form of a scalar value. This value can be positive, negative, or zero. The reward signal helps to update the agent's policy which changes the behavior of the agent, as it learns more about the environment. The agent interacts with the environment until it reaches a terminal state. A series of such interactions with the environment constitutes an episode, also known as trajectory.

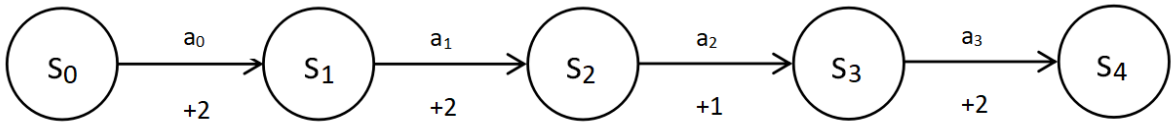


Figure 1. Example of a trajectory/episode. The environment is in a state s and when an action a is taken by the agent, it produces a reward signal and changes state.
Ravichandiran, S., 2020. Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL and more with OpenAI Gym and Tensorflow. Packt Publishing Ltd.

An agent can update its behavior either during a training episode or at the end of one. The agents' goal is to maximize the amount of reward that it gets from the environment. Many training episodes are needed for the agent to find a good policy, which means that the agent requires a lot of interaction with the environment to learn how to behave optimally in it.

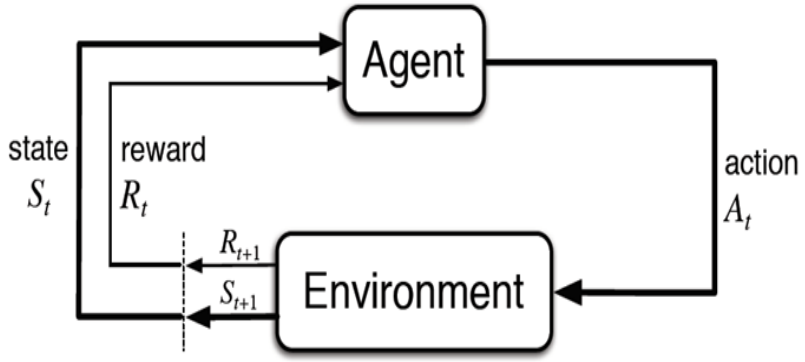


Figure 2. Interaction between agent and environment. Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*.

Most of the time the dynamics of the environment are unknown, meaning that an agent does not know what the resulting state of the environment will be, after it takes an action. Environment dynamics define the probability of an agent ending up in state a state s' by taking some action a from the preceding state s. Mathematically it can be written as:

$$P(s'|s,a)$$

Equation 1. State transition function. Defines the environment dynamics

This is known as the state transition function or transition probability.

RL problems are formulated as Markov Decision Processes (MDPs) which can be solved using a variety of methods such as Dynamic Programming (DP) and Monte Carlo (MC). Solving an MDP means finding a policy that maximizes the rewards. The main property of an MDP is that any state of the environment, which is observed by the agent, depends only on the preceding state and not on any other past states.

This can be written as:

$$P(s'|s) = P(s'|s_1, s_2, s_3, \dots, s_n)$$

Equation 2. The Markovian property. Each state solely depends on the preceding state

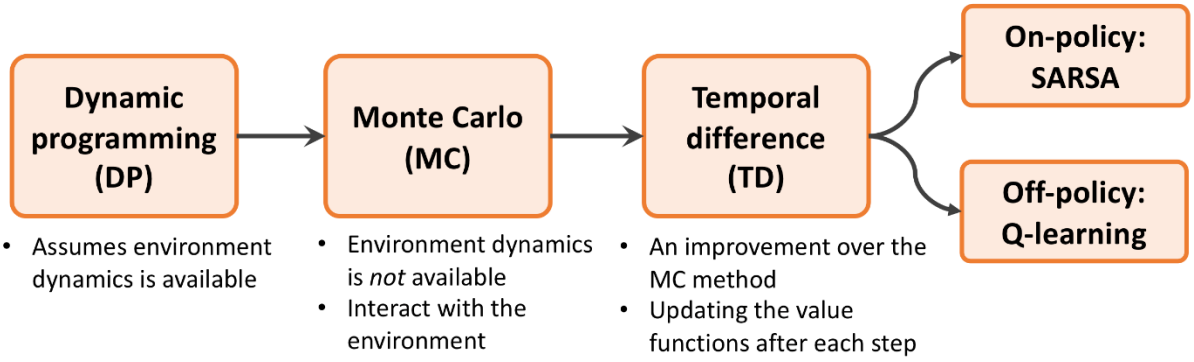


Figure 3. Methods of solving Markov Decision Processes and how they relate to each other. Raschka, S., 2015. *Python machine learning*. Packt publishing ltd.

For DP to be used, the model of the environment must be known, which is inconvenient for a lot of settings. MC methods do not require the model to be known, however the policy of the agent can only be updated at the end of a training episode. A mixture of the two approaches is Temporal Difference (TD) learning which does not require a model of the environment, which makes it model-free, nor the completion of an episode for an update to the agent to take place. The two main TD methods are State Action Reward State Action (SARSA) and Q-learning. The latter is the focus of this dissertation.

2.2 The Q-learning algorithm

The Q-learning algorithm aims to estimate the optimal state-action function $Q(s, \alpha)$ that assigns the optimal value to a state-action pair. It is an algorithm that maximizes the sum of expected rewards that the agent can extract from the environment. The value assigned to

States/Actions	"NOOP"	"FIRE"	"RIGHT"	"LEFT"	"RIGHTFIRE"	"LEFTFIRE"
S1	201	. .. ..	. .. ..	...	....	...
S2	.. .	523	456	. . .	....	...
S3	. . .	. ..	. .. .	789	....	...

Figure 4 Example of look-up Q-table, with several states and the actions of an Atari game. Using the Q-learning update rule, these values should converge to the optimal ones.

each action by the function, is a proxy for the reward that the agent expects to receive from the environment by taking that action and following a specific policy thereafter. By estimating this function an agent can decide which action is optimal for each state of the environment. In the tabular setting, the Q-learning algorithm is implemented using a lookup table called a Q-table, to store the values of state-action pairs in memory. The values are updated periodically with the use of the of the Bellman optimality equation, modified for

the Q-learning algorithm. In the beginning of training the agent performs mostly random actions, this is done in order to learn a good Q function, whereas closer to the end, the agent looks up values of actions on the Q table and takes the ones with the highest value. This means that the policy of an agent starts out stochastic and develops into a deterministic one overtime. The Q value function is recursive in nature meaning that the value of a state-action pair is related to the Q value of the next state-action pair, more specifically the biggest Q-value over all possible actions.

$$Q(s,a) = r + \gamma \, max\{a'\} \, Q(s',a')$$

Equation 3. Bellman equation for discounted future rewards

The max operator considers that maximum Q value out of all values assigned to the actions of the next state s'. The parameter γ is used as a discount factor and represents the uncertainty the agent has about future rewards. Usually set to 0.99 since the agent can never be 100% sure about the reward it is going to receive by taking a series of actions. The value r represents the reward the agent receives for its action. Q-learning is an off-policy TD control algorithm, and it is the RL method used in the entirety of the dissertation. It a control method because its goal is to find the optimal Q function, which leads to the optimal policy. It is an off-policy method because it learns from a different policy from the one that it is following during training. This method, while following the epsilon-greedy exploration

policy learns from a strictly greedy policy. The estimates of the values of state-action pairs are updated using the following update rule.

$$Q(s,a) \Leftarrow Q(s,a) + \eta\,[r + \gamma\, max\{a'\}Q(s',a') - Q(s,a)]$$

Equation 4. Q-learning update rule. Also known as one-step Q-learning

Where η is the learning rate, a parameter which dictates how fast learning takes place, and a' is the action taken in the next state s'. The quantity between the brackets is called the TD error. Essentially it is the difference between the value that the agent thinks an action has and the value the Bellman equation assigns to it, to steer it towards optimality.

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 5. An example of a Q-table changing overtime during training. The starting values can either be random or zero, and by using the update rule, they are steered towards their optimal values. Source: Satwik Kansal, S. and Martin, B., n.d. *Reinforcement Q-Learning From Scratch In Python With Openai Gym*. [online] Learndatasci.com. Available at: <https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/> [Accessed 15 December 2020].

2.2 Introducing Deep Q-learning and motivating its use.

The focus of this dissertation is the use of Q-learning, and more specifically the use of NNs as non-linear approximators that do away with look up tables from the tabular RL setting. Deep Q-learning has become massively popular in the last decade with its introduction in 2013[6]. Tabular methods simply cannot cope with the huge state space that even simple Atari games present, as they need to hold in memory a value for every possible state-action pair of the environment. The Atari platform has a resolution of 210x160 pixels, and each pixel can take one of 128 colors [7]. The number of screens that are possible is 12,833,600, however, the overwhelming majority of these will never be encountered. This is the main reason why Deep RL algorithms are used to approximate the Q learning function. Instead of looking at a Q table to find which action is optimal for a specific state, a function can serve the purpose of that table, by outputting a value for each state action pair. Moreover, this method of approximating the Q function does not require that every state be visited, as it is able to generalize in unseen states.

When it comes to Deep Q-Learning, there exist two main approaches to using a NN as the Q function approximator. The first method is to configure a DQN to accept the state as an

input and provide a value for each action that can be taken from that state, which signifies the value of each action. The second method is to configure the DQN to take as an input, a state-action pair and output one single value. The approach used in this dissertation is the former, as the latter approach is computationally more expensive since a separate forward pass in needed to compute the Q-value of each action for the same state, resulting in a cost that scales linearly with the number of actions. The two architectures aim to accomplish the same goal of approximating the Q function, while using slightly different approaches.

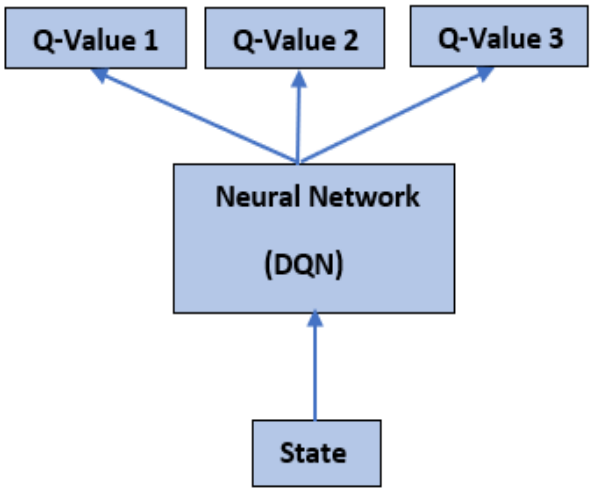


Figure 7 DQN - Action-out architecture.

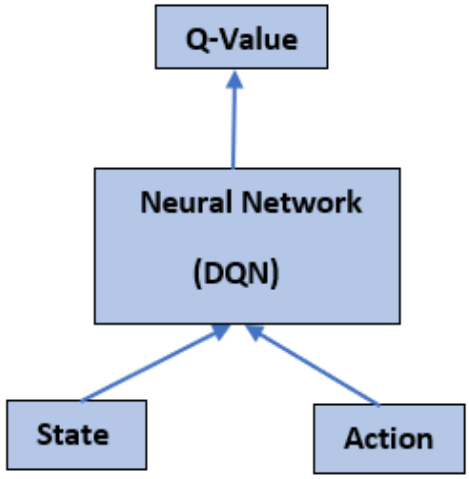


Figure 6 DQN – Action-in architecture

2.3 Overview of feed forward NNs and CNNs.

Neural Networks (NNs) can be categorized into families, with each one excelling at different tasks and processing different types of input data [8][9]. The simplest version of an NN is the Multilayer Perceptron (MLPs), which consists of only fully connected layers. In this network type every node in a layer is connected to every node in the next layer by a weight parameter, with non-linear activations applied to the outputs of each layer. The non-linearity of the activation functions make is possible for NNs to approximate highly complex functions, which makes them a good fit for approximating value functions and state-action functions of RL. The use of MLPs in the Atari setting is infeasible since the number of weights grows very quickly when working with image data. When working with MLPs an image needs to be flattened to a 1-dimensional vector to be used as an input, thus

a greyscale image of dimensions 210x160 would need to be flattened to a 1D array of 33600 elements, each one representing a single pixel value between 0 and 255. With the size of the input layer being 33600 and the next layer with a size of 512 neurons, it would result in 33600 x 512 = 17,203,200 parameters

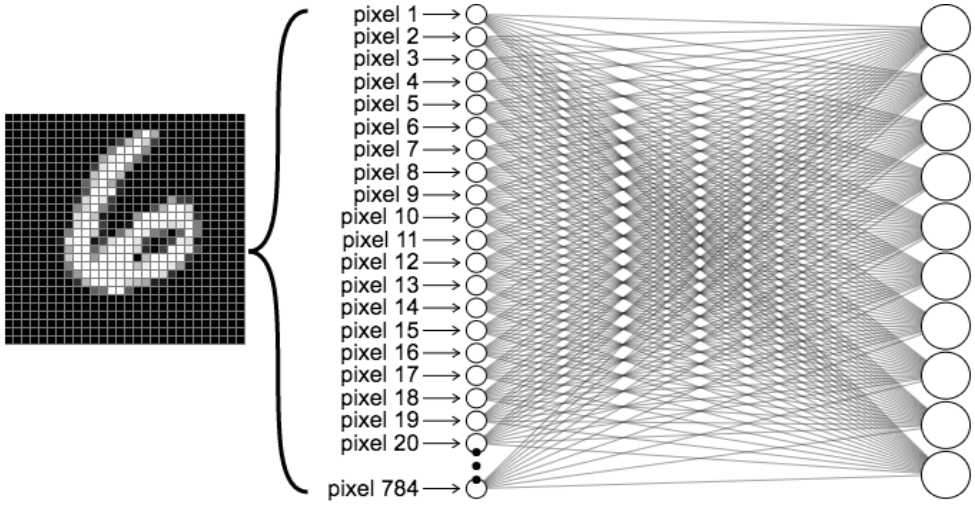


Figure 8. Example of an MLP that requires the input image to be flattened. Image is a sample from the MNIST dataset. Images of 28x28 dimensions, must be flattened to a 784 1-dimensional vector. Source: <https://ml4a.github.io/ml4a/looking_inside_neural_nets/> [Accessed 15 December 2020].

just for the first set of weights. Optimizing such a huge number of parameters is impractical, furthermore MLPs ignore structure, which is inherent to 2D images, by flattening an image to 1 dimensional representation. Convolutional neural networks have skyrocketed in popularity over the past 10 years, and right now are being used for diverse tasks such as object detection, image generation and autonomous driving. CNNs excel at learning from images, with their biggest strength being their ability to exploit the special structure of image data because of the use of convolution kernels, in the early layers. The early layers of a CNN can be thought of as feature extractors, with the resulting features being propagated to fully connected layers as in the original DQN. Furthermore, CNNs employ kernels, which is an effective way of reducing the number of trainable weight parameters and keeping it within reasonable bounds, this form of parameter sharing is very

useful. All agents developed for this project follow the architecture which was used in the Nature publication by DeepMind.

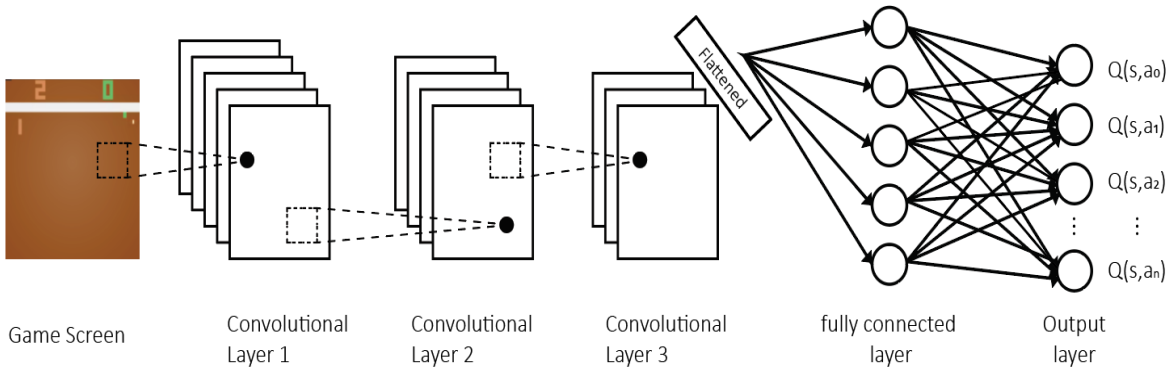


Figure 9. CNN architecture example. The convolution operations keep the number of weights small compared to MLPs. This is an example of a DQN, employing a CNN architecture. A similar approach is followed in the experimentation. Ravichandiran, S., 2020. *Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL and more with OpenAI Gym and Tensorflow.* Packt Publishing Ltd.

2.4 The use of Deep Q learning in creating agents that play Atari games.

The most popular way of training DQNs for Atari games is to provide raw frames from the games without creating any hand-crafted features. The input to the DQN is an image that captures the state of the game at a specific time step. Since the input is a frame the DQN employs a Convolutional Neural Network (CNN), that extracts features from the input image. These features are propagated forward to Fully Connected Layers which in turn pass the signal along to the output layer, which provides a single scalar value for each possible action. The output node with the highest activation corresponds to the action that the agent has decided to take for the input state. This is how the agent interacts with the Atari game environment.

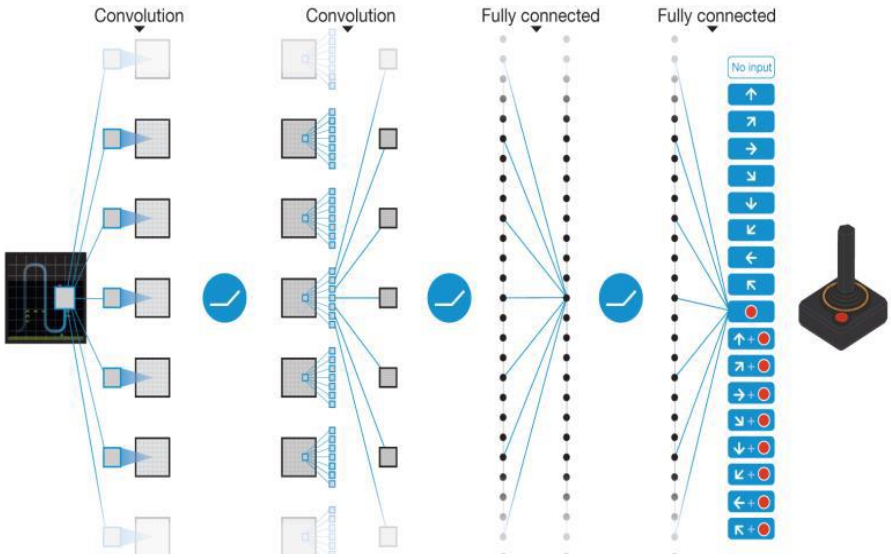


Figure 10. Original DQN architecture. V. Mhin et al. *Nature* 518, 529-533

The action could have a positive, negative or a neutral outcome, this is determined by the game and its reward system. The output of the DQN is a continuous value that represents how "good" a particular state-action pair is, or how good a particular action is for a specific state. The DQN learns to play the game by adjusting its outputs while training, this is achieved by Gradient Descent (GD), a popular optimization technique for NNs. During training, the weights of the DQN are adjusted by GD and by the end the DQN has converges to a policy that selects the optimal action for any state.

2.5 DQN Training and optimization routine and loss function.

DQNs of the kind described here require target values that tell the DQN how to change its Q-value estimates. These target values are constructed using the Q-learning algorithm[10]. Since the output of the DQN and the target value are both continuous numerical values, a popular loss function used is the Mean Squared Error (MSE). This error has the form:

$$\text{MSE} = \frac{1}{k}\Sigma_i^k (y_i - \hat{y}_i)^2$$

Equation 5. General form of Mean squared error, where the number of samples is K.

Where y_i is the target value and $\hat{y}_i$ is the output of the DQN.

For the DQN setting, the loss function takes the form of:

$$l(\theta) = \frac{1}{k}\Sigma_i^k \left(y_i - Q_\theta(s,a)\right)^2$$

Equation 6. MSE Loss function for DQN optimization.

The DQN is represented by Q_θ, which stands for the weight vector that parameterizes the DQN. The loss must be minimized by computing the gradient of the function and updating the weight vector θ. Usually, a stochastic version of GD is used, which means that the network is trained on batches of data. The individual loss of samples is calculated as:

$$\text{Loss} = \left(r + \gamma\, max\{a'\} Q_\theta(s',a') - Q_\theta(s,a)\right)^2$$

Equation 7. Prediction - Target pair loss for the DQN, using the Bellman equation for computing the target.

For each sample in the dataset. Where r is the immediate reward resulting from action α being taken in state s and γ is a discount factor usually set to 0.99. This approach uses Q-learning to update the parameters of the NN, so that overtime the Q function approximation gets better.

2.6 Getting SGD to work with the use of Experience Replay.

The final important piece of the puzzle to get the Deep RL approach to work is the optimization method. Deep neural networks are parameterized by weights that are updated periodically during training, this is what constitutes 'learning' in NNs. In the Deep RL setting, a DQN that has been trained appropriately has converged to a good approximation of the optimal Q function. The loss function and optimization method (Stochastic Gradient Descent (SGD), Adam, etc…) require that prediction-target pairs be Independent and Identically Distributed (IID). For the samples to have this property, a replay buffer is constructed to hold transitions, then the samples are sampled uniformly. The replay buffer is implemented as a fixed size que data structure, with new experiences pushing out old ones as they are generated. This means that as the agent interacts with the environment, the transitions between states are stored in the queue, along with the actions taken and the rewards received. These transitions are called experiences and usually take the form of <SARS> tuples (State, Action, Reward, State).

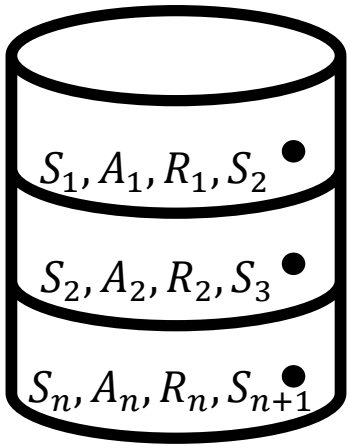


Figure 11. Experience replay buffer example. This can be thought of as the dataset of the RL setting.

Because subsequent transitions come from the same episodes, it means that the samples are highly correlated with one another, which is why they are sampled uniformly for GD, usually in mini batches of 32 or 64 transitions. Once the buffer has enough experience, the

agent can begin learning from these experiences by gradient descent. Lastly to make training more stable a target network is used in the Bellman update to calculate the $Q(s', a')$ value. This network is a copy of the main network, with its weights frozen during training. This network is synched periodically with the main network to provide stability during training.

The MSE loss takes the form of:

$$l(\theta) = \frac{1}{k}\Sigma_i^k\left(r_i + \gamma \max\{a'\} Q_{\theta'}(s_i', a') - Q_\theta(s_i, a_i)\right)^2$$

Equation 8. DQN MSE loss with target network for non-terminal transitions

If $s'$is a non-terminal state, or:

$$l(\theta) = \frac{1}{k}\Sigma_i^k\left(r_i - Q_\theta(s_i, a_i)\right)^2$$

Equation 9. DQN MSE Loss for terminal transitions

For terminal transitions where there is no next state. Only the reward is used to update the Q-function value approximation. Where Q_θ is the main network and $Q_{\theta'}$ is the target network, which is parameterized by a different weight vector. As the DQN is trained on more experiences, its approximation of the optimal Q value function becomes better. One huge barrier to get DQNs to work is that since the targets are not given before training as in SL, the agent needs to interact with the environment to produce its training data. The Appendix provides more details about how the network is optimized using Pytorch. The full code is also attached in the Appendix.

2.7 Markov Formulation and state space description for using Deep Q-learning on Atari games.

To get Deep Q-learning to work, it is crucial that the problem can be formulated as a Markov Decision Problem (MDP). This must apply to the Arcade game setting as well[11]. The most fundamental assumption that is made, when using RL methods to control Atari games, is that the agent has all the information it requires to act optimally in any state. In other

words, the state should, no matter how it is constructed, hold all the information necessary for the agent to make optimal decisions. When it comes to Atari, a single state in a game can be represented as a single frame. However, for practical and theoretical reasons several frames are combined to form a complete state.

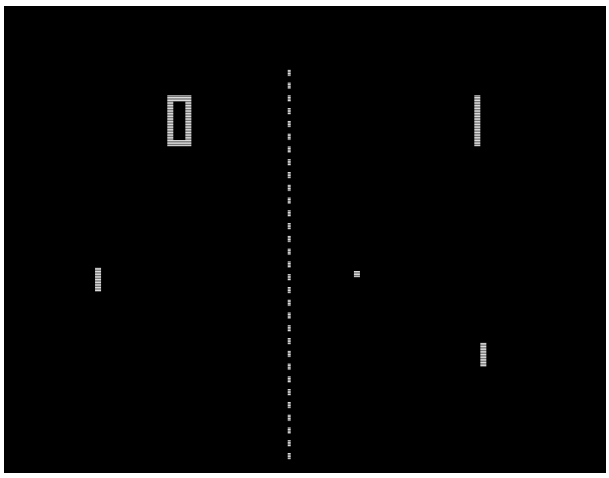


Figure 12. A single frame of Pong. Several of these are stacked together to create the input to the DQN.

This simple method allows the games to be solved as MDPs. This means that states can be distinguished from one another and that all information that is needed to take an optimal action in each state, is part of the state representation. For example, in Pong a single frame cannot be used as a state, since it does not contain any information about the direction of the ball, so in this example the agent does not have the full information to make an informed decision as to if it should go up, down or take no action. More information must be used from subsequent frames to capture this dynamic. In Arcade games usually 4 frames are combined to make up the state. While there may exist other longer-term dependencies in the environment, this method works well for simple arcade games. The biggest outlier being the game "Montezuma's Revenge" which was notoriously hard to beat in the early days of Deep Q-learning.

2.8 Exploring of the environment and exploiting the accumulated experience.

A crucial component to building a self-learning agent is the exploration strategies that it follows. A common theme that presents itself is the exploration-exploitation dilemma,

which is one of the many open questions in the field of RL. Fundamentally a good exploration strategy will have a nice tradeoff with the agent exploring the environment some of the time, while also exploiting what it has learned so that it can learn from the feedback signal it receives when is takes an action.

Some of the most used exploration strategies [12] in RL are:

- Epsilon-greedy

  Using this strategy, the agent performs a random action with probability e and the rest of the time, the agent decides on which action to take by using its Q-table or Q-approximation. The variable e controls whether the agent will choose to explore or exploit what it has learned. Usually, e starts out with a value of 1.0, which means that all actions are chosen randomly, and is slowly decayed to a small value such as 0.05 or 0.01, which means that the agent takes a random action only %1 percent of the time. Using epsilon-greedy helps the agent to explore the environment during the beginning of training while also forcing the agent to stick to a coherent policy close to the end of training.
- Boltzmann exploration

  Instead of picking an action at random with uniform probability, the Boltzmann exploration approach assigns a probability to each available action. When using the e-greedy all actions have equal probability of being chosen, in other words they are sampled uniformly. The Boltzmann method, however, assigns a probability to each action based on the average reward that the agent collects from each. This method also uses the SoftMax function to force the sum of the probabilities assigned to each action to be equal to 1.
- Finally, others exploration methods like Thompson sampling and the Upper confidence bound, which are more complex to implement are used heavily in RL and other statistical settings like operations research.

2.9 Advanced forms of Deep Q-networks.

Since the introduction of Deep Q-learning several advancements have been made to the core algorithm which offer better performance and faster convergence. Two of these add-ons are discussed here as they have been used in the experiments. These methods account for the advanced DQN types that were talked about in the proposal for this dissertation.

2.9.1 Double Deep Q-Learning and tackling overestimation bias.

The simple DQN algorithm is known to suffer from an overestimation bias[13], due to the max operator when computing the target values for GD. This is known to hinder agent learning as the agent gets overconfident about the amount of reward it is going to get from taking a specific action.

In the case where the agent has just started learning and the Q function approximation is bad, if it performs an action and receives a reward, the agent will on subsequent episodes tend to choose the same set of actions and this could create a loop where the agent is stuck making sub-optimal decisions because of the initial update.

This bias can be fixed by tweaking the way that the target value is computed. Instead of simply using the target network to calculate the Q-values of the next state using the target network and taking the action with the highest value, a more stable approach is to use the main network to choose the action of the next state and take its Q-value from the target network.

[Original DQN approach]:

$$\text{Target} = r + \gamma \, max\{a'\} Q_{\theta'}(s', a')$$

Equation 10. DQN Target value computation. Using the target network

[Double DQN approach]:

$$\text{Target} = r + \gamma \, Q_{\theta'}\big(s', argmax\{a'\} Q_{\theta}(s', a')\big)$$

Equation 11. Double DQN target value computation using both the main and target network

The $arg\,max()$ expression returns the action with the biggest Q-value, as decided by the main network, and the target network $Q_{\theta'}$ is used to calculate its Q-value. This change to the calculation of the target has been shown to fix the overestimation of the simple DQN algorithm. The application of Double Q-learning in Deep RL not only reduced the overestimation bias but also lead to better performance on several games of the Atari benchmark when it was introduced. This method is since known as Double Deep Q-learning.

2.9.2 Dueling DQN architecture and modelling the advantage function.

The Dueling DQN architecture[14] is a big change from the simple DQN architecture that only has one defined path from the convolutional input layers to the output layer.

The main concept that the Dueling architecture is trying to capture, is that of the 'advantage' of taking an action over all others in a specific state. Instead of simply computing the $Q(s,a)$ pairs, this architecture breaks down this term into two other terms, that of $V(s)$ which represents the value of a state, and $A(s,a)$ which represents the advantage that a

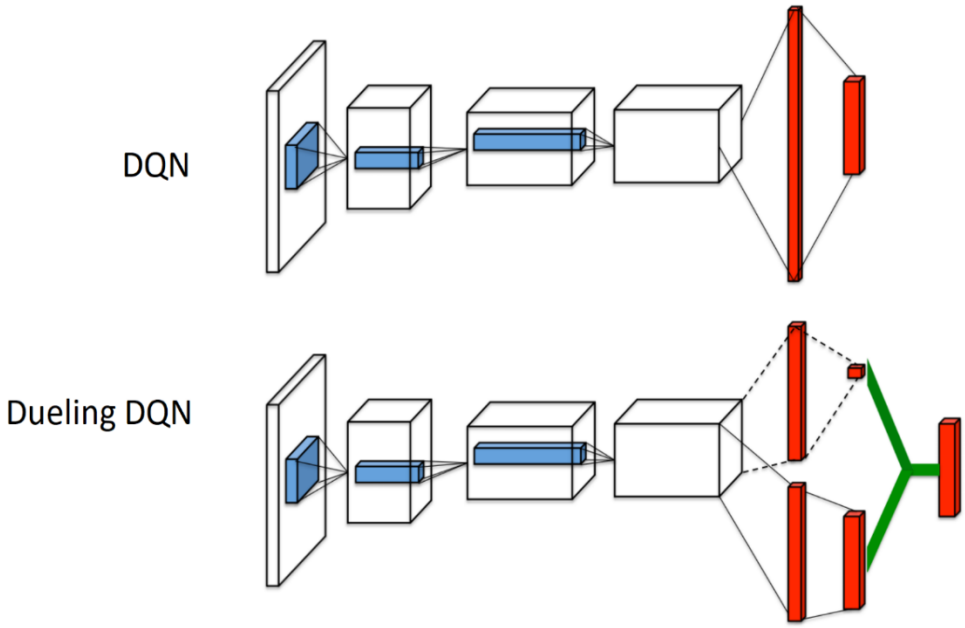


Figure 13. Dueling DQN architecture. Compared to the simple DQN architecture, the Dueling DQN has two defined paths. Both versions output Q-values for each action. Wang, Z et al

specific action has over all others in a specific state. To compute these two scalar values, the architecture of the DQN is split into 2 paths, modelling each function separately. This architecture decomposes the Q function into the advantage and state-value functions as:

$$Q(s,a) = V(s) + A(s,a)$$

Equation 12. Q function as the sum of the advantage and state-value functions

The Dueling architecture calculates a scalar value for the value of the state on the top path, while also producing a scalar value for each action on the bottom path. The final output is still the $Q(s,a)$ value and it is computed in the following manner:

$$Q(s,a) = V(s) + [A(s,a) - \frac{1}{|A|} \sum_{a'} A(s,a')]$$

Equation 13. Q-values computation in the output layer of the Dueling DQN architecture

Where |A| stands for the length of the action space.

By subtracting the mean advantage value from each Q value, the final vector is computed, which holds a Q value for each action. Another valid quantity to subtract is be the maximum advantage value, however the mean value has been reported to result in more stable performance. This operation also helps the DQN to not overestimate the Q values. In a state where the any action leads to negative reward, it is more beneficial to compute the value of that state ($V(s)$), independent of the actions. This way the agent can more easily learn that it is a bad state that it should avoid. Moreover, by computing the advantages values $A(s,a)$for each action, instead of directly computing the $Q(s,a)$values, it is clear whether an action is substantially mode beneficial than the rest, and it is not 'the least bad' action to take. The introduction of the dueling DQN architecture brought better training stability faster convergence and better results on the Atari benchmark. Both improvements to the vanilla DQN covered above are used in the experimentation.

2.10 Curriculum Learning

Curriculum Learning (CL) mimics the human academic experience, by slowly exposing the agent to more difficult situations the agent should exhibit constant learning and improvement by building on past experiences, just like humans progress an academic environment. CL has proven to be beneficial in the supervised setting, in diverse tasks such as computer vision and natural language processing [15]. In the RL setting, CL has been explored and applied in numerous ways[16]. The simplest approach to CL is to manually design a curriculum that slowly increases the difficulty of the task, so that the agent can get better incrementally. This is the approach that is followed in the experiments. This setup is tricky however since a balance must be kept between exploring the easy and hard environments. Moreover, since NNs are known to suffer from catastrophic forgetting, it is crucial that the manual curriculum is constructed in a way that it presents the agent with easy examples, even during the later stages of training. This means that simply increasing the difficulty of the environment overtime will result in the agent 'forgetting' how to behave on the easier setting. Other more advanced approaches to CL applied to RL exist, such as the student Teacher-Student Curriculum Learning (TSCL)[17], a framework of automatic curriculum construction which consists of a teacher agent choosing the tasks that the student agent should train on, usually based on some heuristic, such as the rate of progress the Student shows on a task. This framework has been shown to outperform even the results of manually crafted curricula in some settings. This approach is framed as a Partially Observable Markov Decision Process (POMDP) and the approaches of Deep Q-learning presented here are not sufficient to solve it, as they constitute a harder class of MDP problems.

The approach to curriculum learning taken in this dissertation, is the creation of a manual curriculum based on the number of steps that an agent is allowed to take in a specific environment setting. The two most important requirements are followed however, with an agent starting under the 'Easy' setting, advancing to the 'Hard' setting, and taking the last

steps on ‘Easy’. This means that difficulty is increased incrementally, and that the agent is exposed to the easier setting later in the training so that it does not forget how to behave properly under it. CL approaches can lead to improved generalization and faster convergence, which from a mathematical point of view means that better local minima can be found in the landscape of the loss function. The issue of generalization is of great importance since real world applications of RL agents (with the most popular one being autonomous robots) would require the agent to be able to generalize to a variety of deployed settings.

Chapter 3. Methods and results

3.1 OpenAI gym overview.

The dissertation takes advantage of the Gym library[18] provided by OpenAI and its plethora of game difficulty levels and modes for Atari game playing. The Gym library makes use of the famous Arcade Learning Environment (ALE)[19], while also providing other diverse environments, such as classic control environments. Gym provides many implementations of the same game, each of which has a slightly different internal structure. Environment types for Pong include:

- Pong-v4
- PongNoFrameskip-v4
- PongDeterministic-v4
- Pong-ram-v4
- Pong-ramDeterministic-v4
- Pong-ramNoFrameskip-v4

The dissertation focuses on 2 games. Specifically, “Pong” and “Demon Attack” as these games required less computational resources to achieve good results on. Most research literature uses the ‘NoFrameskip’ environment type, since in this version most of the pre-

processing can be controlled by the user, which provides great flexibility. The data preprocessing is written in environment wrappers. The OpenAI baselines GitHub repository is a great reference for environment wrappers. This dissertation follows the trend of using the 'NoFrameskip' version of the games, along with the OpenAI developed environment wrappers for the crucial preprocessing. The entirety of the NN code was developed using Pytorch, it is provided in the appendices. The main benefit of using Pytorch is that the computation of the gradients of NN weights, with respect to the loss function, is automatic.

3.2 A note on computational requirements for training Atari agents.

Training RL agents to play Atari games at a high level requires substantial computational power, even after 7 years from the original achievement. Papers and publications tend to measure the performance of an agent by tracking the average reward that the agent receives over millions of played frames. While some games are easier than others, such as Pong in contrast to Montezuma's revenge, the Atari benchmark remains a challenging target to this day since it requires vast amount of computation to achieve results on the entire game suite, and is therefore out of reach for individual researchers. This highlights the major issue Deep RL presents, namely the problem of sample inefficiency. The training routines presented in this paper have been designed to work on specific games, and not the entire suite. This dissertation focuses on easy games "Pong" and "Demon Attack", which do not require agents to be trained for millions of frames before starting to exhibit good performance. This limits the generalizability of the results presented here, since not all games can be "solved" using the training routines detailed here.

3.4 "Pong" Experimentation

3.4.1 Game description and reward system.

The first game used for experimentation is Pong. The goal of the game is to use the paddle to hit the ball at the right time and launch it towards the opponent's goal, past their paddle. Each episode is divided into 21 rounds with each round ending with a reward of +1 or -1

for the agent. The case worst scenario is that the agent losses all 21 rounds of the episode and ends up with a reward of -21, the best case scenario is that the agent wins all rounds of the episode and accumulates a reward of +21. A reward of -1 or +1 is given to the agent when they either lose or win a round, on every other occasion the reward is 0. This intermediate reward is the signal feedback that the agents will be trained to maximize.

Available actions for 'PongNoFrameskip-v4' given by AIgym:

- "Noop" (No action taken)
- "FIRE"
- "RIGHT"
- "LEFT"
- "RIGHTFIRE"
- "LEFTFIRE"

The agent will have to pick and choose among these actions at each timestep. By training over a period of hundreds of games it can learn how to win.

3.4.2 Difficulty settings.

The OpenAI gym version of Pong includes 2 difficulty settings, easy and hard. Examples of the 'Hard' and 'Easy' setting are provided below. The paddle on the right is controlled by the artificial agent, while the enemy CPU controls the left.

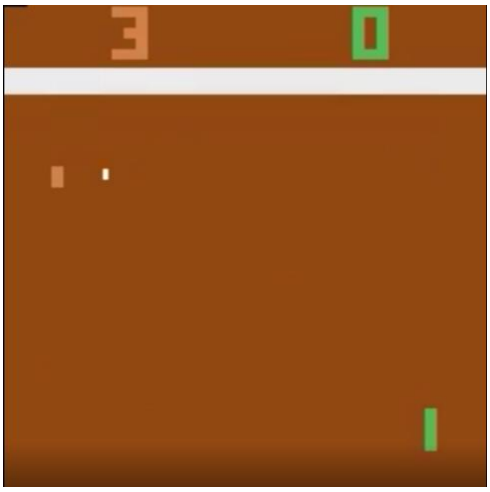


Figure 15. Pong 'Easy' setting environment frame

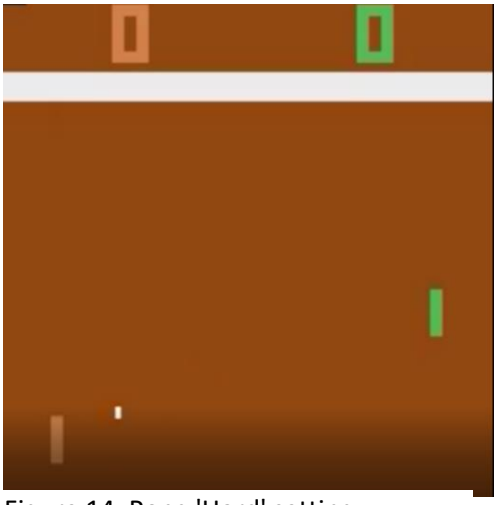


Figure 14. Pong 'Hard' setting environment frame

Under the ‘Easy’ setting the agent has control of a bigger paddle than the CPU. This gives the agent a clear advantage, whereas under the ‘Hard’ setting, it controls a paddle that is the same size as the CPU. The Gym version of Pong includes only these 2 difficulty modes. The reward system does not change between the 2 settings.

3.4.3 Exploration approach

A crucial component to building a self-learning agent, is setting an appropriate exploration method. At the beginning of training the Q-function approximation provided by the NN is bad, since the weights have been initialized randomly, so the agent should act randomly to build up experience and update its Q-function approximation. As the training progresses and the agent becomes more experienced, acting randomly becomes inefficient, so the Q approximation should be used to for the agent to decide on the action to take. A method that facilitates this type of exploration is the epsilon-greedy approach, this method is used in all experiments. All agents are trained using an epsilon-greedy policy starting from 1.0 and decaying to 0.01 over the first 100k frames. Meaning that at the very beginning the agents take random actions 100% of the time, but by the time they have reached 100k frames, they take random actions only 1% of the time for the rest of the training.

3.4.4 Experiment set up and evaluation methodology.

The DQN agents are first trained on the 'Easy' environment for 500k steps. This allows them to become proficient at the game. Subsequently they are tested on the 'easy' setting for 50 test episodes.

The same agent types are trained on the 'Hard' environment for 500k steps. They are subsequently tested on the 'Easy' setting for 50 episodes.

Lastly the same agent types are trained using a manually created curriculum. They are also tested on the 'Easy' setting for 50 episodes.

Training parity: The same hyper-parameters are used to train all agents so that there is a level playing field among the agents. A small exception is made for the curriculum routine, where epsilon decays to value of 0.01 over the first 300k frames.

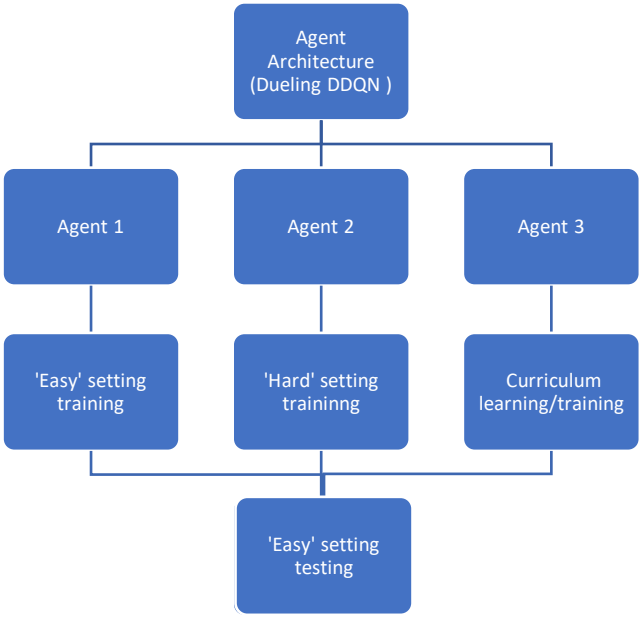


Figure 17 Graphical representation of Agent training and evaluation

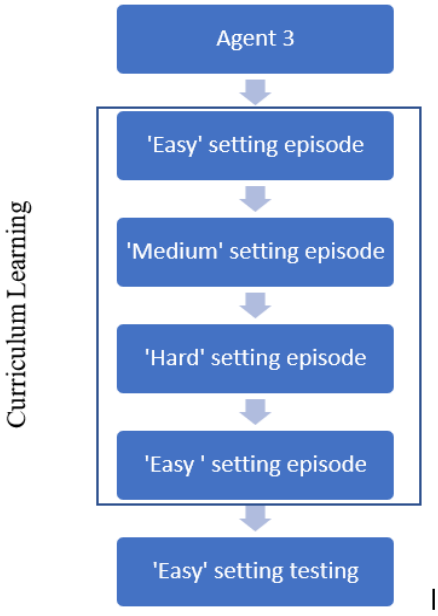


Figure 16 Graphical example of manual curriculum training and evaluation

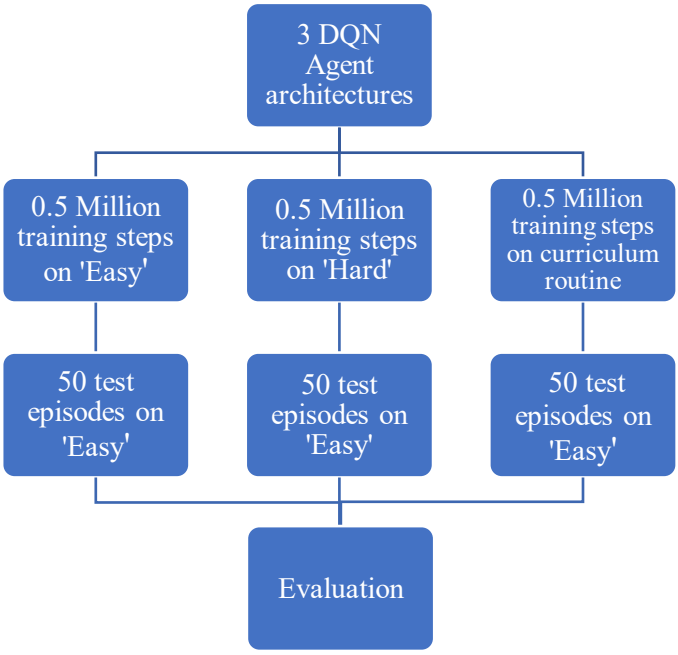


Figure 19. Training and evaluation method for agents trained on pong

Figure 18. Curriculum training routine used in the experiments

3.4.5 Preprocessing of states and observation wrappers.

The environment outputs raw frames from the ALE console that must be processed before they are fed into the DQNs. The original Atari frames have dimensions of 210x160 with a 128-colour palette. To reduce the dimensionality of the setting and for reasons of computational efficiency each frame is processed by a wrapper which greyscales and resizes the frame to 84x84 dimensions. Moreover, in order to ensure that the problem is an MDP, frame stacking is used, which means that the input to the NN is 2 frames stacked together to capture the dynamics of the environment such as the direction that the ball is moving in, and the speed at which it is moving, this is slightly different to the popular method approach of stacking 4 frames, however it provides significant speed up. Another key preprocessing method is to make the agent take a decision about an action every 4th frame, since the difference between subsequent frames is negligible, the latest action taken by the agent is

repeated over the 4 skipped frames, this also speeds up the training process. All these pre-processing tricks that make the algorithm work, are implemented as Gym wrappers and have been adapted from the OpenAI baselines repository. These preprocessing steps are crucial to make the use of Deep RL agent practical for games.

The repo can be found at:
https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py

3.4.6 Model architectures

The model architectures used are: Deep-Q Network, Double-Deep-Q Network and Dueling-Double-Deep-Q Network. The structure of the neural networks is detailed below.

Number of layers	Optimization method	Activation function	Loss function optimized	Dueling architecture
3 Conv Layers + 2 Fully Connected Layers	Mini-batch Stochastic Gradient Descent	RELU, on every layer except the output layer.	Mean Squared Error	3 Conv Layers+ 2 Paths of 2 Fully Connected layers

Table 1. Neural network architecture and configuration

Both the Deep-Q Network and the Double Deep-Q Network share the same NN architecture. The Dueling architecture used by the third agent is slightly different. Instead of just 2 FC layers, 2 separate paths are defined with 2 FC layers each, and their outputs are combined at the end, using the process detailed in 2.9.2. The first hidden layer is a convolution of 32 filters, with a kernel size of 8x8 and stride 4. The first hidden layer applies a rectified nonlinearity. The second hidden layer is a convolution of 64 filters with a kernel size of 4x4 and a stride of 2 with a ReLu activation. The third layer is also a convolution of 64 filters, but with a kernel size of 3x3 and a stride of 1. The fourth hidden layer is a FC layer with a ReLu activation with 512 nodes. The final layer does not have an activation

function, and the number of output nodes is 6, with each one representing a distinct action. The Dueling architecture shears the same convolution layers but has half the number of nodes in the FC layers. This is done to keep the number of parameters the same.

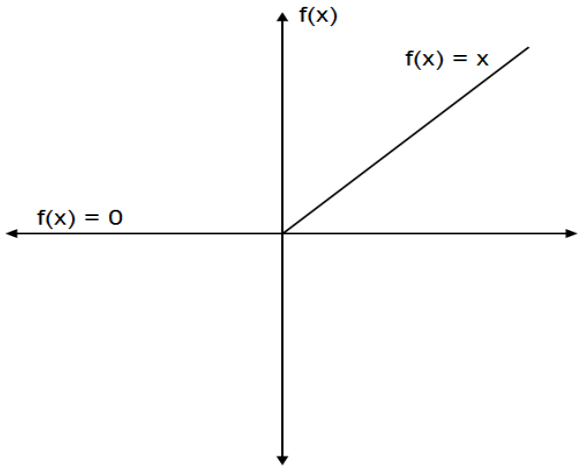


Figure 20. Rectified Linear unit activation.

3.4.7 Training Hyper-parameter values:

Below the training parameters are given for the 'Hard' and 'Easy' training routines.

Parameter	Value	Description
Minibatch size	**32**	Number of training cases over which each stochastic gradient descent update is computed.
Experience replay size	**100K**	The size of the object that holds the transition (State, Action, Reward, Next state) objects.
Target network update frequency	**1k**	The frequency with which the target network is synched with the main network.
Discount factor	**0.99**	The Gamma value used in the Q-Learning update.

Initial / Final exploration	**1.0/0.01**	Initial/final value of ε for the ε-greedy policy.
Final exploration frame	**100K**	The number of frames over which the initial value of ε is linearly annealed to its final value.
Replay memory start size	**10k**	The number of transitions that must exist inside the buffer, before SGD updates can start to take place.
Stopping criterion: Number of steps	**0.5M**	The maximum number of steps an agent can take to achieve the target reward. If an agent reaches this number, the training is stopped.
Stopping criterion: Mean Reward achieved	**+18**	The target mean reward to achieve on average over 100 consecutive episodes. The maximum reward that can be achieved in an episode of Pong is +21, with the minimum being -21.

Table 2. Training hyper-parameters

3.4.8 Evaluation of training results.

All results can be run through Tensorboard, for the purposes of illustrating the results, the values were extracted, and MS Excel was used to create the graphs. All excel files can be found on the supporting material. Two types of graphs are shown here. The individual rewards and a 100-episode moving average. An example of the Tensorboard interface is provided in the Appendix.

“Easy” training routine evaluation

[Visualizing individual episode rewards]

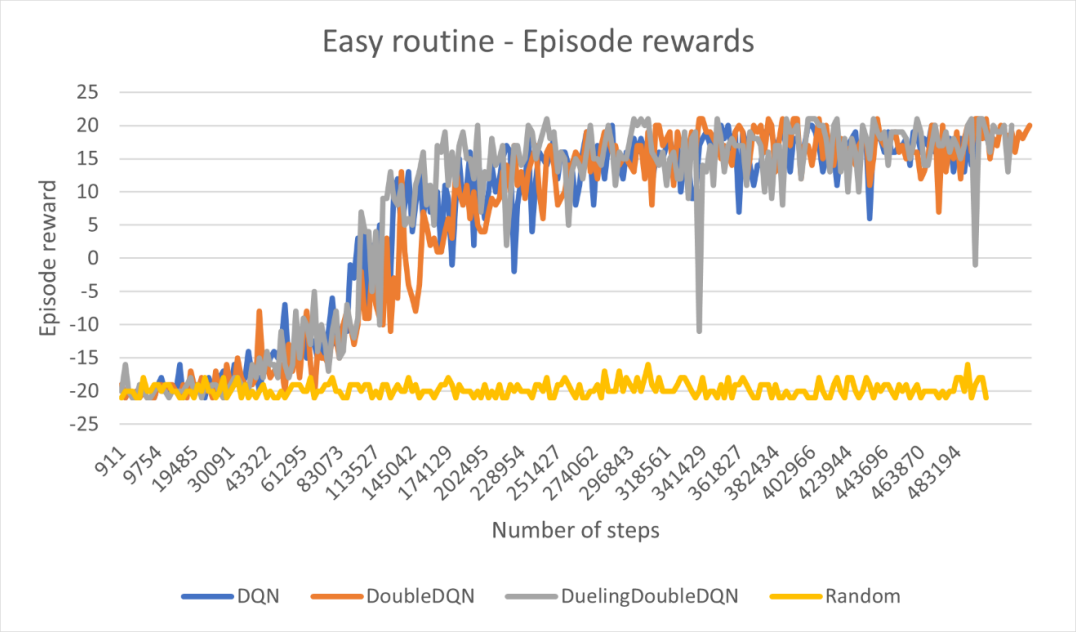


Figure 21. Easy training routine results. Individual episode rewards over 500k steps are plotted here. This equated to roughly 250 episodes. Small deviations were observed for each agent.

The training results show that all agents found a winning strategy on the ‘Easy’ setting of Pong. The random agent of course never learns anything and continues to get close to the minimum reward on every episode played. During the first 100k steps all agents mostly explored the environment. It is evident from the graph that all agents learn to play the game better as they accumulate more experience. As the exploration nears to zero the agents get to explore the environment less and must use what they have learned to play better and achieve a high score. In fact, most of the learning takes place between steps 100k and 200k, where all agents go from a reward of -10 for an episode, to a reward of +15. This representation of agent learning can be quite noisy, since in one episode the reward might be close to perfect, while in the following, the reward might be a lot smaller due to random actions taken, this results in sharp drops that are evident in the graph. A more convenient and illuminating approach is to graph the mean reward over 100 episodes, this results in a

much more readable graph and demonstrates the steady progress agents have in learning to play the game.

[Visualizing mean rewards over the last 100 episodes]

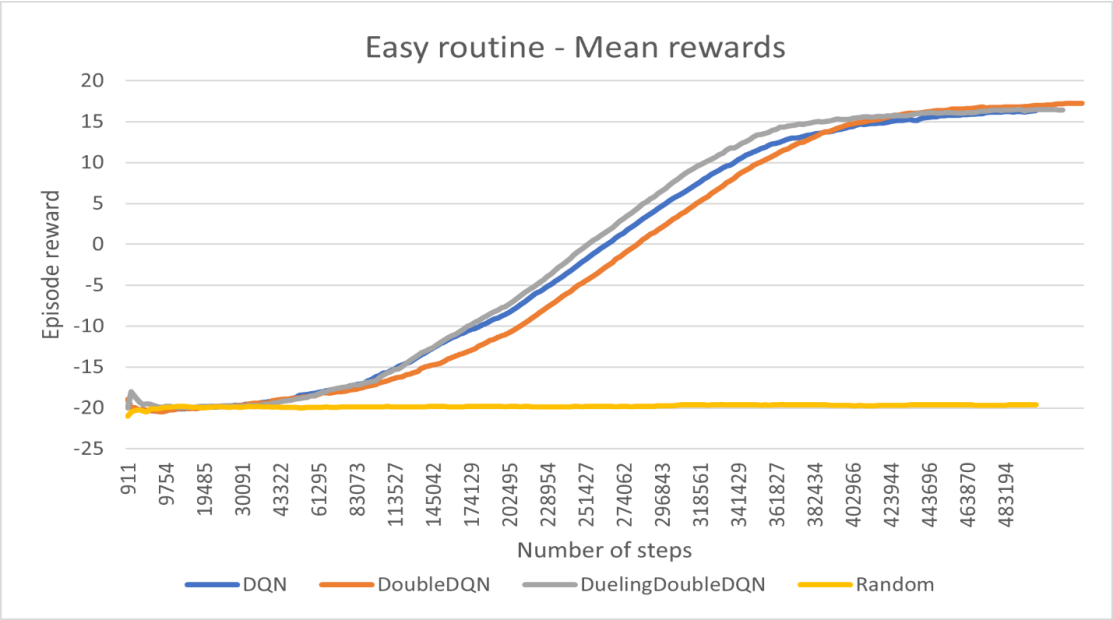


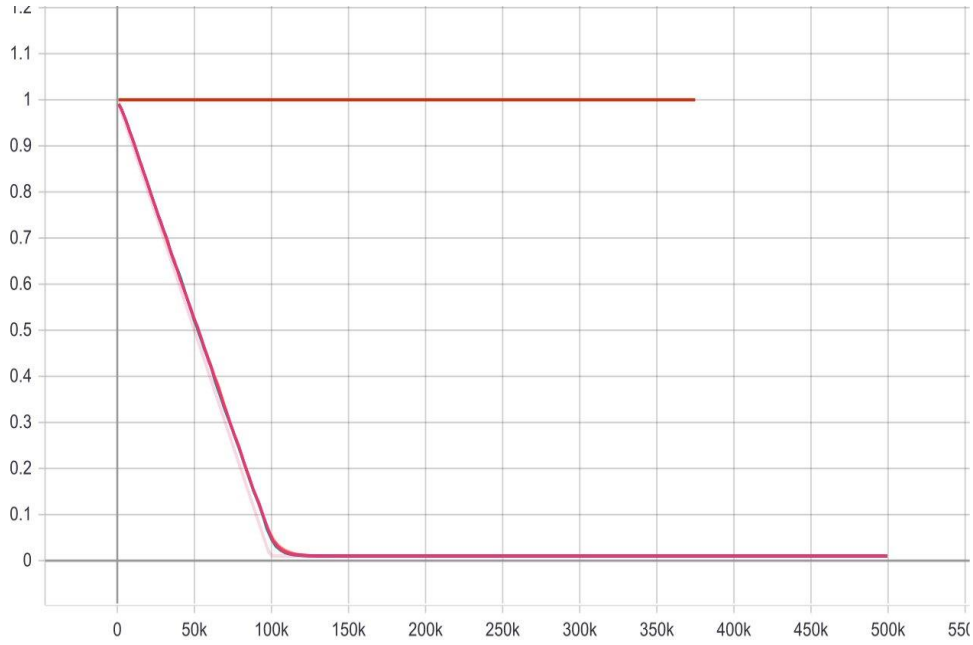
Figure 22. Easy training routine results. The mean reward over 100 episodes is shown here. None of the agents achieve the target reward.

A perfect score of +21 is very difficult to achieve, but all agents come close to the target reward of 18, although none of them achieve the target reward within the allowed number of steps. The DuelingDoubleDQN agent while achieving great results earlier than other agents, came in second, while the DoubleDQN came in first, and the simple DQN last. It is crucial to point out that the final difference is not big, however the add-ons to the simple DQN have proven beneficial for this game. The final mean rewards achieved are given below.

Table 3. Pong -Easy training routine results. Final mean reward for all agent types

Agent Type	Final Mean Reward
DQN	**16.36**
Double DQN	**17.26**
Dueling Double DQN	**16.40**

[Epsilon decay example]



Red shows the steady epsilon of the random agent

Purple represents the decay of all agents

Figure 23. Epsilon parameter linear decay example. The parameter e is decaved over the first 100k steps of training.

An example of epsilon decay is offered here. The parameter starts at 1.0 and by the 100kth step it reaches its minimum value of 0.01 at which point it remains steady until the end of the training routine. This linear decay schedule is helpful to the agents and allows them to converge on good results.

The above results show that the agents can successfully learn to play Pong, at a high level on the 'Easy' setting. The result that we are interested in however is the performance of these agents during testing on the 'Easy' setting.

"Hard" training procedure evaluation

The individual rewards for each episode for the 'Hard' training routine are given below.

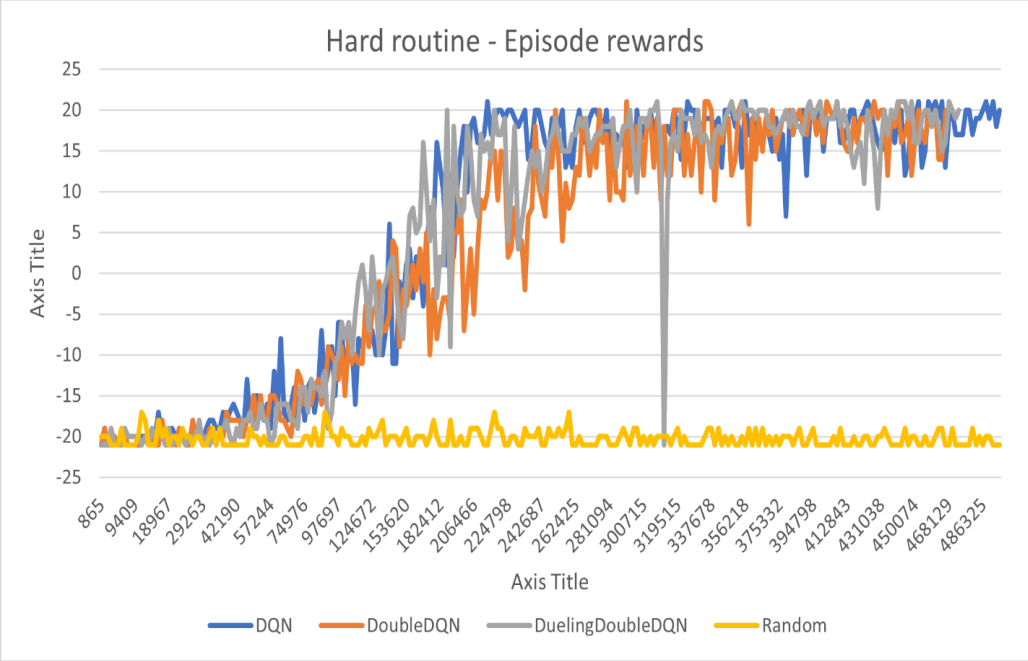


Figure 24. Hard training routine results. Iindividual episode rewards. The agents can find a winning policy even under the harder setting

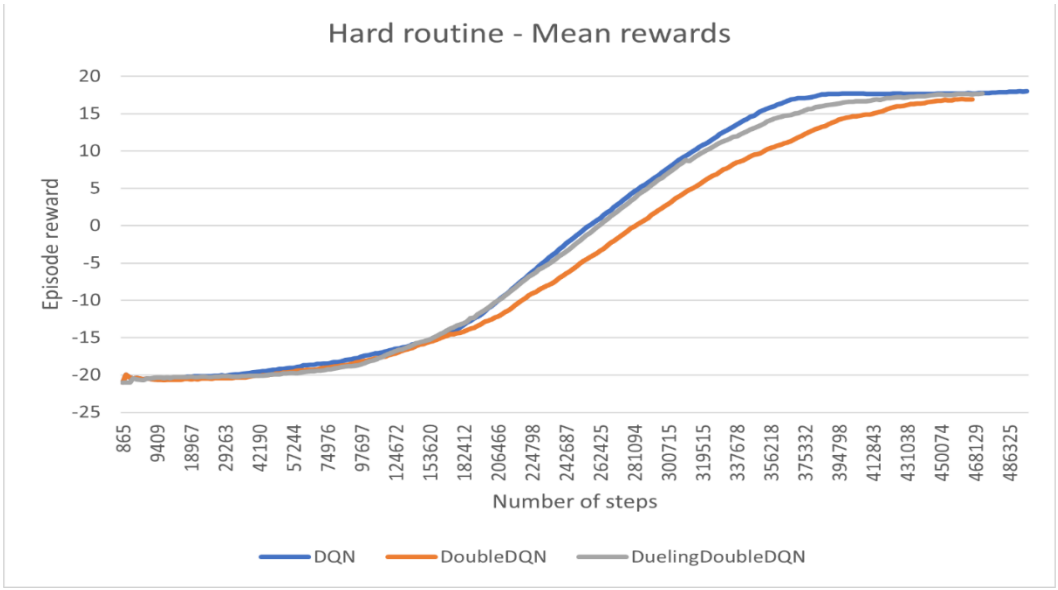


Figure 25. Hard training routine results. mean rewards over 100 episodes. All agents exhibit learning

Like the 'Easy' setting, all agents have found a winning strategy. In this setting the simple DQN has an advantage over the other agents. The harder difficulty setting did not have a negative impact on the performance of the agents, in fact, as is evident from the results the agents achieved results comparable to the 'Easy' routine. This was achieved with the same training parameters, so at least for Pong, the difficulty setting does not have an impact on agent learning. The simpler DQN manages to perform better under the higher difficulty setting. All three agents converge close to the same mean reward. The DuelingDoubleDQN comes in 2nd outperforming the DoubleDQN, which is the reverse of what happened under the 'Easy' training routine. It is notable however that only the simple DQN has achieved the target mean reward of +18, whereas the others exhausted their number of allowable training steps. The full results are given below.

Agent Type	Final mean reward
DQN	**18**
Double DQN	**16.92**
Dueling Double DQN	**17.74**

Table 4. Pong. Hard routine training results. Final mean reward of all agents.

Curriculum training routine evaluation

The curriculum learning approach is the hardest training routine to set up, as there exist many variations that are interesting to explore. The most important parameter that needs tweaking is epsilon. In this set up the below parameters were tweaked, and new ones are defined to control the number of steps that the agent takes in each version of the environment.

Parameter	Value
Easy number of steps	200k
Hard number of steps	200k
Final easy steps	100k
Final exploration frame	300k

Table 5. Curriculum training bespoke parameters to allow for exploration of all settings

The curriculum training routine starts off with 200k steps taken on the 'Easy' environment, then the agents are trained for 200k steps on the 'Hard' setting, and finally the agents take 100k steps in the 'Easy' setting. The epsilon is linearly decayed over the first 300k steps, this is done so that the agents have time to explore the 'Hard' environment. The final 100k steps taken in the 'Easy' environment are taken with epsilon set its minimum value of 0.01. This curriculum fulfils the two most important criteria of CL. The difficulty of the samples increases during training and the agent is given easy examples after it has been exposed to the difficult ones. All 3 agents have found a winning policy, however the switches from the 'Easy' to the 'Hard' environments are evident during the learning process. The results illustrate this in a clear way.

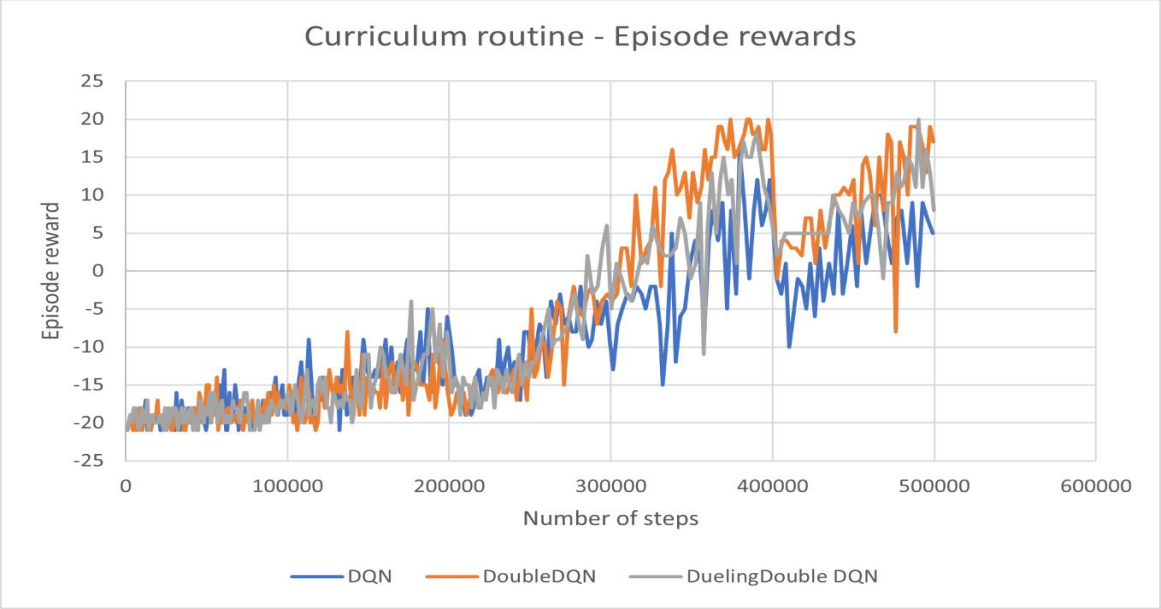


Figure 26.Curriculum training routine episode results. the reward drops appear when agents are switch to a different difficulty setting.

During the first 200k steps the agents explore and learn to behave in the 'Easy' environment, around step 200k the agents are put into the 'Hard' environment, this results in a drop in the reward that they accumulate. This is made obvious by the graphs. For the next 200k steps the agents are trained on the 'Hard' and they demonstrate great learning with all of them managing to achieve a positive reward. By the 300kth step the epsilon parameter has decayed to 0.01 which means that the agents overwhelmingly choose to exploit what they have learned, rather than explore the environment. On the 400kth step the agents are again put on the 'Easy' environment, this also comes with a drop in performance. Just like in the first switch, all 3 agents exhibit reduced performance, as they try to adapt to the new setting, the sharpest decrease is seen on the DQN agent. This time the agents do not get to explore the environment as the epsilon parameter is kept at 0.01, instead the agents have to use what they have learned in the 'Hard' environment to perform well on the 'Easy' environment and have to adjust their policy accordingly. Remarkably, all 3 agents manage to adapt to this new environment without needing to explore. It took the simple DQN agent less than 50k

steps to start achieving positive scores on episodes. The advanced agents are more resilient to this environment change, as they do not exhibit as big of a drop in performance, they are able to adapt faster and achieve rewards comparable to the last rewards they achieved on the 'Hard' environment.

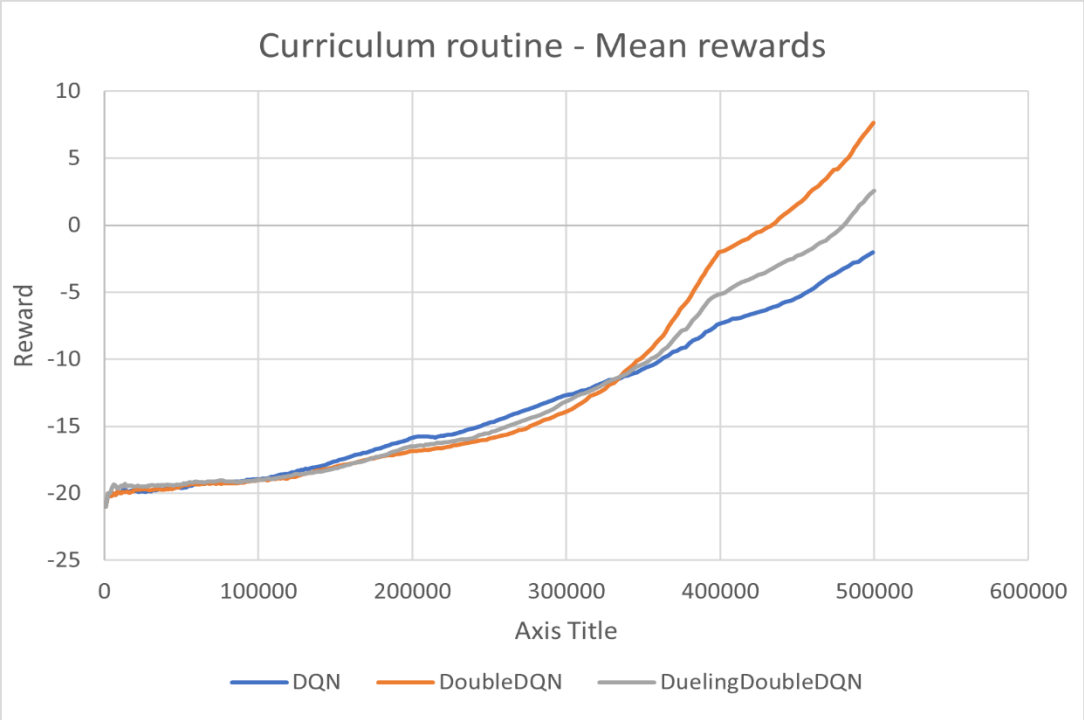


Figure 27. Curriculum routine - mean rewards. Fluctuations are present when the agents change environments.

The simple DQN is the worst performer of the three and it is clear that the enhancements to the simple DQN architecture, present in the other two agents, were beneficial in this curriculum routine. The best performer is the DoubleDQN with the DuelingDoubleDQN following second. The DoubleDQN agent achieved the highest mean reward of +7, while the DuelingDoubleDQN managed to achieve a reward of +2. The simple DQN agent did not manage to achieve a positive reward by the end of the training routine and ended up with a reward of -2. The switches from 'Easy' to 'Hard' and from 'Hard' to 'Easy' are evident on the 200kth and 400kth step, respectively. The reward lines start ascending, and on the 200kth step a dip starts to form, by the 400kth all lines are trending upwards, and once

again a dip starts to form and by this point the reward outcomes have started to diverge. For the next 100k steps it is made clear that the more complex agents have an advantage over the simpler DQN agent and can generalize to the 'Easy' setting a lot better. This curriculum training approach has shown that the 'Hard' setting proves to be beneficial to the agents if they are allowed to tune their policy a bit on the easy setting. Lastly, these agents, having been trained on both difficulty settings, should perform well on both difficulties, when tested. The outcome that we are interested in however is the performance they can achieve when they are tested on the 'Easy' difficulty setting.

Agent Type	Final mean reward
DQN	**-2.04**
Double DQN	**+7.64**
Dueling Double DQN	**+2.58**

Table 6. Pong. Curriculum training results. Final mean reward

3.4.9 Evaluation of testing performance and evaluation of results.

The final part of the proposed evaluation technique is the testing of all agents on the 'Easy' setting of the Pong environment. The trained agents are deployed on the 'Easy' setting, for 50 episodes. Over 50 test episodes, they accumulate rewards, which can be compared, and a breakdown of greater detail can take place, by considering the type of the agent and the training routine it was trained on. The testing routine is run with exploration parameter e set to 0.02, this is a good set up for determining whether an agent can generalize to different difficulties. The results presented are broken down by the agent type. This allows us to analyze the impact of the different training routines for each agent type.

DQN agents testing performance

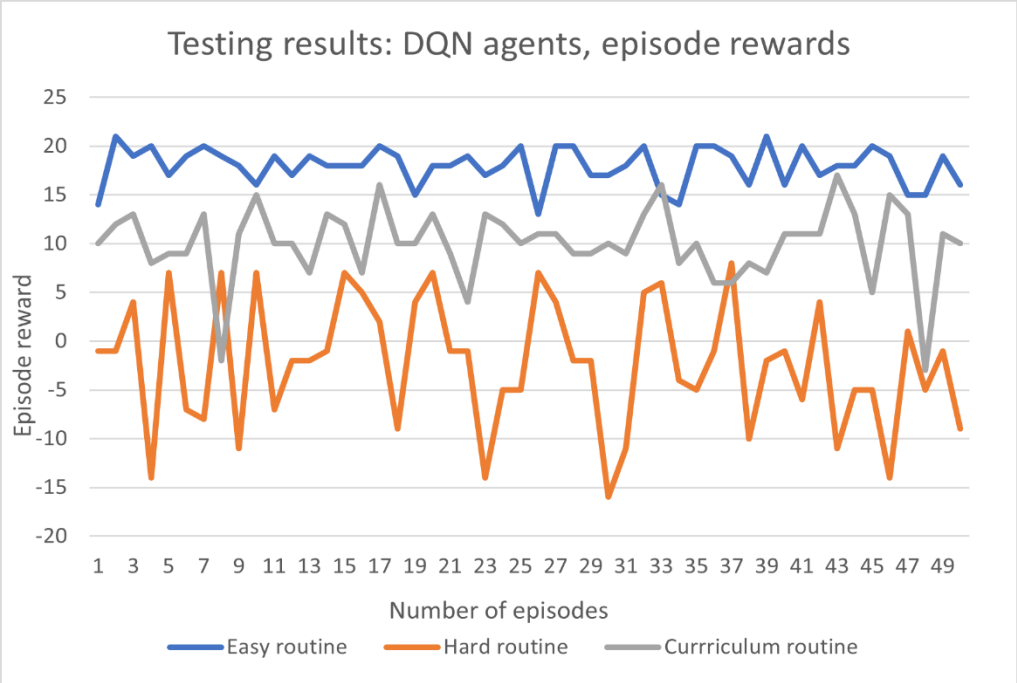


Figure 28. Pong. Testing results. DQN agents. Episode rewards

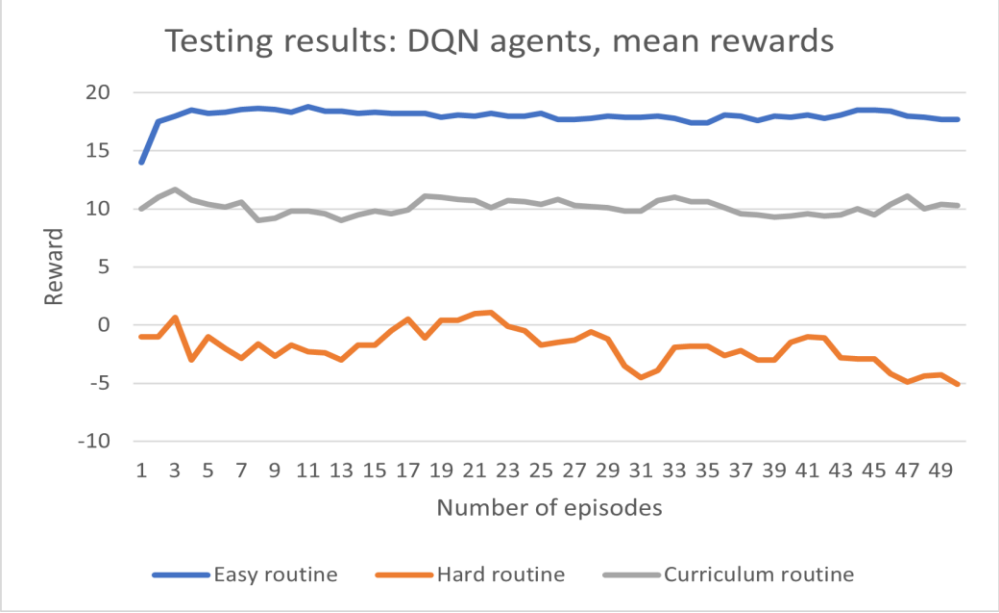


Figure 29. Pong. Testing results. DQN agents. Mean rewards

The results show that while the 'Hard' training curriculum was able to produce a simple DQN agent that can perform well on that setting, the agent struggles to generalize to the easier setting effectively. The agent usually gets between +2 and -10 points in each episode, which results in an overall negative mean reward. This goes against the intuitive notion that an agent trained on a harder setting, should perform well on an easier one.

The DQN agent that was trained on the 'Easy' routine, achieves a high reward throughout testing. This was expected, as during the later steps of the training reward the same agent was achieving close to the optimal reward of +21. The mean reward hovers around +18, which matches the reward the agent was achieving during close to the end of the training routine.

The DQN agent that was trained on the curriculum performs substantially better than the agent that was strictly trained on the 'Hard' setting, indeed its performance is more comparable to that of the agent that was trained on the 'Easy' setting. The results show that the simple DQN agent has benefited greatly from this type of curriculum learning. While the average reward does not come close to a perfect score, it is still able to get on average a respectable +10 points.

These results show that while training an agent only on the 'Hard' difficulty does not produce good results, mixing the training routine with hard and easy settings can beneficial. Furthermore, while the agent trained on the 'Hard' difficulty is the worst performer, it is still able to achieve a positive reward on a few occasions, which shows that the agent not completely ineffective on the easier setting. The compiled mean results are presented below:

Agent Type	**Final mean reward**
Easy routine	**+17.7**
Hard routine	**-5.1**
Curriculum routine	**+10.3**

Table 7. Pong. DQNs testing results. Final mean rewards

DoubleDQN agents testing performance

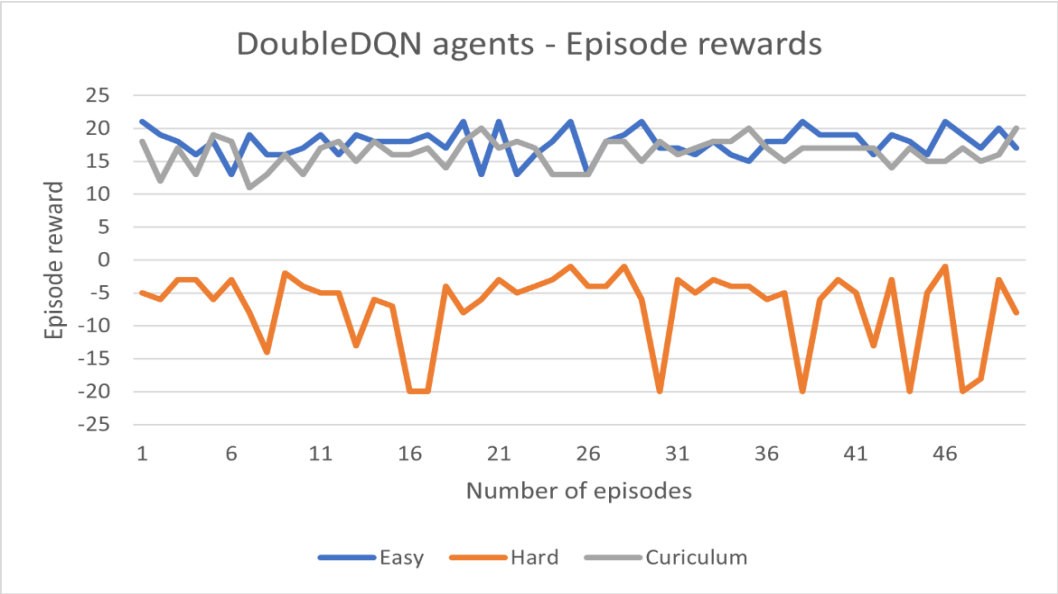


Figure 30. Pong-Testing results. DoubleDQN agents, episode rewards

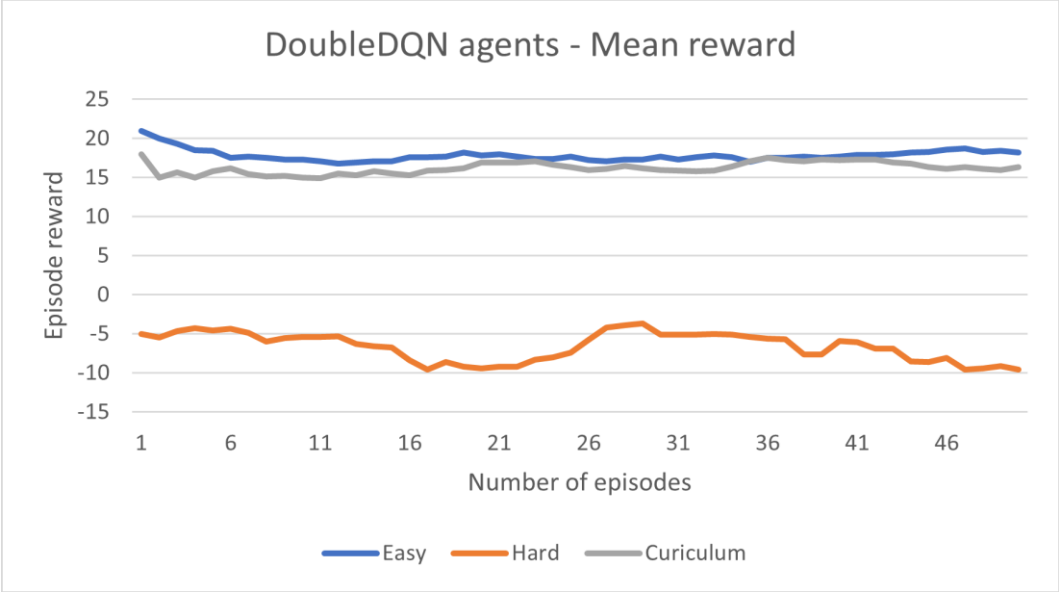


Figure 31. Pong-Testing results. DoubleDQN agents, mean rewards

The DoubleDQN agents follow the same pattern as their simpler counterparts, the best performer is the agent that was trained strictly on 'Easy', while the curriculum learner comes in second, and the agent trained on 'Hard' comes in last in terms of performance.

This time the agent trained on the curriculum routine manages to achieve even better results with a mean reward around +15. This makes the helpful effect of Double Q-learning clear and demonstrates that the curriculum routine is best used with more complex DQN agents.

The DoubleDQN agent that was trained on the 'Hard' setting performs worse than the simple DQN that was trained using the same training routine, this means that training any agent on the 'Hard' routine will hamper the ability of the agent to perform well in the hard setting. This is further evidence that an agent trained on the higher difficulty setting, does not perform well on the easier one. The individual episode rewards show even more extreme dips in the reward received compared to the simple DQN counterpart.

The compiled mean reward results are presented below:

Agent Type	Final mean reward
Easy routine	**+18.2**
Hard routine	**-9.6**
Curriculum routine	**+16.3**

Table 8. Pong. DoubleDQNs testing results. Final mean rewards.

DuelingDoubleDQN agents testing performance

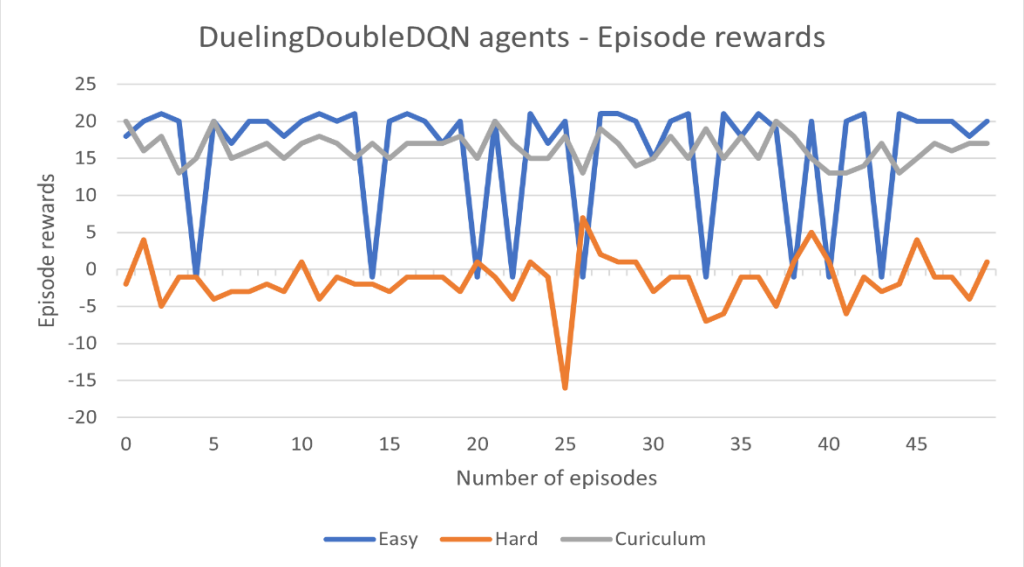


Figure 32. Pong. DuelingDoubleDQNs individual episode rewards

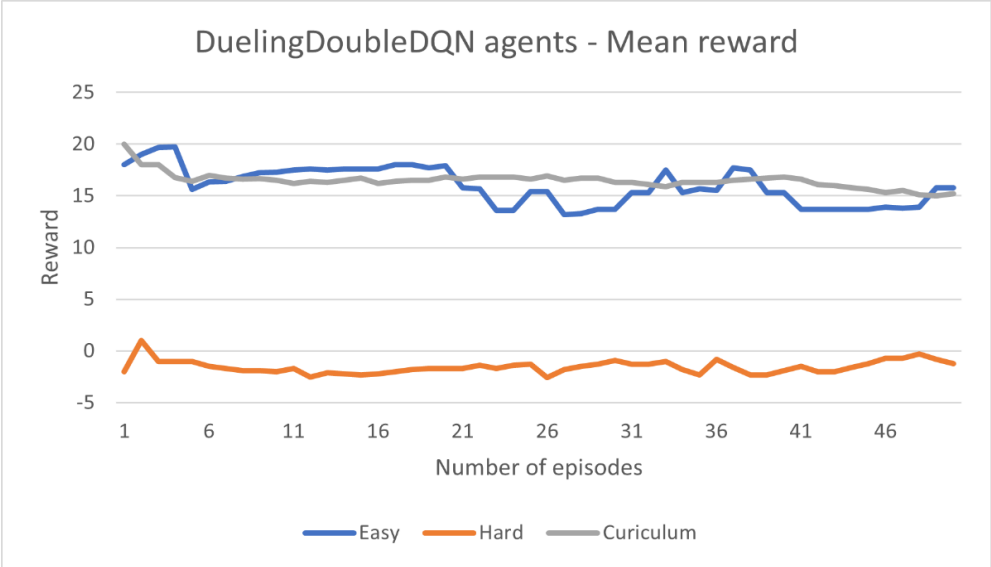


Figure 33.Pong. DuelingDoubleDQNs testing mean rewards

The most advanced group of agents follows the same established pattern. With the 'Easy' and curriculum routines achieving high rewards while the agent trained on the 'Hard' routine achieving a mean reward close to 0.

The curriculum training routine essentially matches the results of the "Easy" routine. This is the only time that this happened during testing and again shows that the curriculum training routine is best paired with a more complex agent.

Agent Type	Final mean reward
Easy routine	**+15.8**
Hard routine	**-1.2**
Curriculum routine	**+15.2**

Table 9. Pong. DuelingDoubleDQNs testing results. Mean reward

Complied testing results on Pong

A detailed table of the results is given below. The major filters that define an agent are:

- The type of the agent

- The training routine it was used on

Overall, there are 9 trained agents. The results are given below. The cumulative reward is given, as well as the mean reward over the 10-episode moving average.

Agent Type	Training Routine	Final Cumulative reward	Final mean reward
DQN	EASY	+899	+17.7
DoubleDQN	EASY	+887	+18.2
DuelingDoubleDQN	EASY	+800	+15.8
DQN	HARD	-114	-5.1
DoubleDQN	HARD	-354	-9.6
DuelingDoubleDQN	HARD	-77	-1.2
DQN	CURRICULUM	+501	+10.3
DoubleDQN	CURRICULUM	+814	+16.3
DuelingDoubleDQN	CURRICULUM	+816	+15.2

Table 10. Pong. complied test results.

Over 50 test episodes, the maximum cumulative score is 50x21=1050 points, or on average an agent should receive +21 points if it has learned a good enough policy. The results how that the agent that comes the closest to this perfect score is the DoubleDQN agent trained on the 'Easy' routine, achieving a mean reward of +18.2, with the simpler DQN achieving the second highest value of +17.7. The curriculum training approach saw good performance overall with the DoubleDQN achieving a respectable mean reward of +16.2, which rivals the agents that were trained strictly under the 'Easy' setting, in fact this agent achieves a higher mean reward than the more complex DuelingDoubleDQN agent which managed to achieve a mean reward of +15.8. Moreover, this specific curriculum shows it benefits more complex DQN architectures, with the DuelingDoubleDQN (+15.2) and the DoubleDQN (16.2) outperforming the simpler DQN agent (10.3). Lastly these curriculum-learned agents should also be able to perform quite well when test on the 'Hard' environment setting. It stands to reason that agents trained on a curriculum of mixed 'Hard' and 'Easy' episodes, can achieve relatively good results on both settings. It turns out that the success of the agents trained on a 'Hard' curriculum is limited when deployed on the 'Easy' setting, with the best

performer, the DuelingDoubleDQN, being close to achieving a mean positive reward, but not quite making it (-1.2). The results show that under this training routine the most complex algorithm can achieve better results. Presumably, these agents can perform a lot better on the 'Hard' setting.

To sum up, the results show that the agents trained on the harder difficulty setting, no matter the type, cannot compete with the agents trained on the 'Easy' routine. The results do show however that the agents trained on a curriculum can produce results comparable to the ones produced by the agents trained on the 'Easy' routine. This is a valuable piece of information, as it implies that the agents trained on the 'Hard' routine could achieve a lot higher rewards if they were to be allowed to evolve their policy by exploring the easier setting for a few episodes.

3.5 "Demon Attack" experimentation and results.

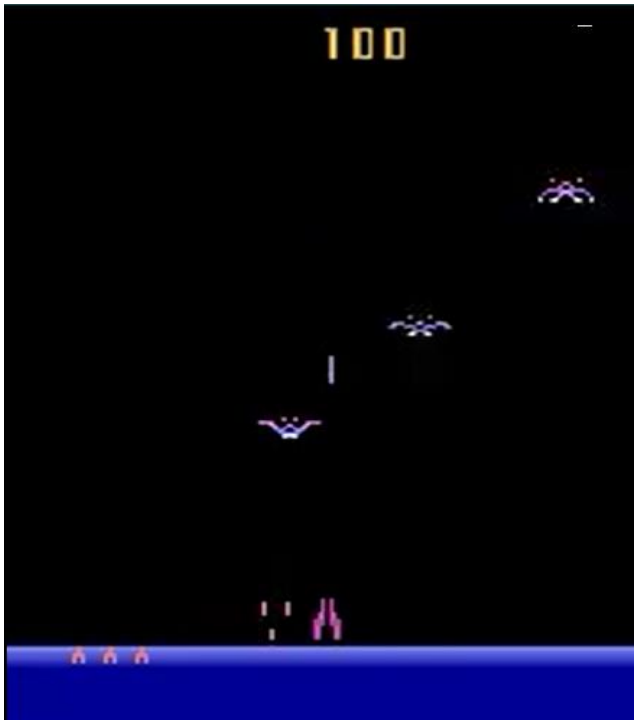


Figure 34. Demon Attack example frame. The agent controls the gun on the bottom and enemies shoot from above

3.5.1 Game description and reward structure.

The second game tested is “Demon Attack”. Demons appear in waves like other space-themed shooters, but individually combine from the sides of the screen to the area above the agent’s cannon. The agent starts with 3 lives and the game ends once all lives are lost. Enemies come in waves, and the difficulty is increased after each one, as new enemies are introduced with more complex attack patterns. The agent achieves higher rewards for taking out an enemy from the 5th wave, than an enemy from the 1st wave. The smallest reward that can be achieved is +10, for taking out a single enemy from the 1st wave, since the loss of a life is not accompanied by a negative reward, and if not a single enemy can be killed during an episode, then the reward is 0. Enemies in later waves are worth a lot more points if they are taken out. If an agent clears a wave of enemies, it is given an additional life. This game does not have an end state, instead the goal of the game is to achieve the highest score possible. If need be, enemies are recycled for later waves.

3.5.2 Available actions:

This version of the game has the same action space as ‘PongNoFrameskip-v4’. A low number of actions reduces the complexity of a game significantly. In other Atari games such as “Tennis” number of actions is much bigger (18 Actions), and the agents need to be trained for millions of frames to find a winning policy.

3.5.3 Difficulty settings.

The OpenAI gym version of Demon Attack includes 2 difficulty settings, easy and hard. The gym environment used is ‘DemonAttackNoFrameskip-v4’. The environments do not change drastically between difficulty settings. Under the hard setting the enemies on screen spawn faster and are more aggressive. In both difficulty settings, as an agent progresses in the game, more powerful enemies are presented. This is a different set up to that of Pong where the opponent is permanently changed for the entirety of the game. For comparison reasons random agents are run on both settings to establish a baseline performance.

3.5.4 Exploration approach

The approach used is identical to the one employed on 'PongNoFrameskip-v4'. The same exploration parameters are used here.

3.5.5 Experiment set up and evaluation methodology.

The training routines are the same as the 'Pong' solution. The testing now has a duration of 100 episodes instead of 50.

3.5.6 Preprocessing and observation wrappers.

The preprocessing steps remain the same with only one addition made for 'Demon Attack'. An additional observation wrapper is used. The wrapper is used on games that use lives. Each loss of life is declared as the end to the episode, however the environment is not reset until all lives have been exhausted. This is supposed to help agent learning, by forcing the agent treat all lives equally.

3.5.7 Model architectures

The same architectures are reused here.

3.5.8 Training Hyper-parameter values:

Same training hyperparameters as Pong are used here, with a minor exception. This time there is no target reward, as there is no end state, and the goal of the game is to gather as many points as possible and maximize the score. The agents simply exhaust the number of steps it can take.

3.5.9 Evaluation of training results.

Random agent runs.

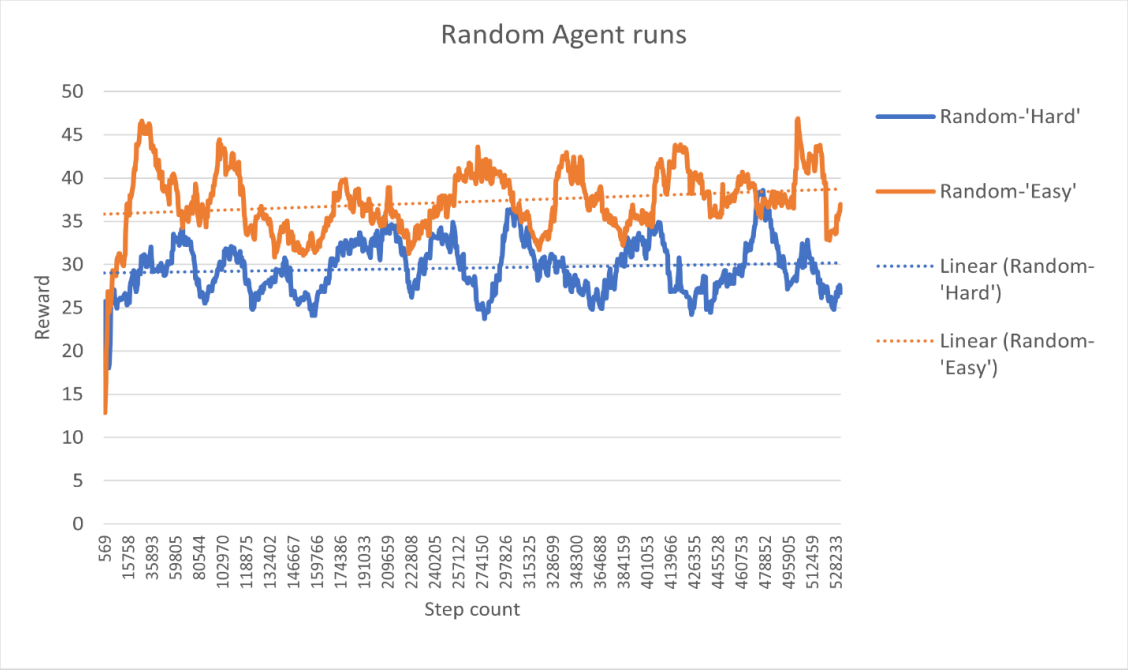


Figure 35. Demon attack. Random agent runs

Random agent results:

On average a random agent playing on 'Easy' difficulty achieves +35 reward. This equates to defeating the first 3 or 4 enemies of the game, with each of them giving a +10 points in reward. On the 'Hard' difficulty similar results are encountered with agent achieving +30 reward on average. The run collects the mean reward over the last 100 episodes over a period of 512k steps. These results establish that at best an average run includes the random agent defeating the first wave of enemies if it starts from the beginning of the game. This is done to establish a baseline to measure agent learning against for both difficulty settings. Multiple runs were carried out to establish which setting should be labeled 'Easy' and which should be labelled 'Hard' as the AIgym toolkit does not make it clear, since the difference between the two difficulty settings is not immediately obvious like with Pong, where the opponents paddle changes size.

"Easy" training routine evaluation

All the graphs presented here show the mean reward over a 100-episode moving window, since individual episode rewards are extremely noisy.

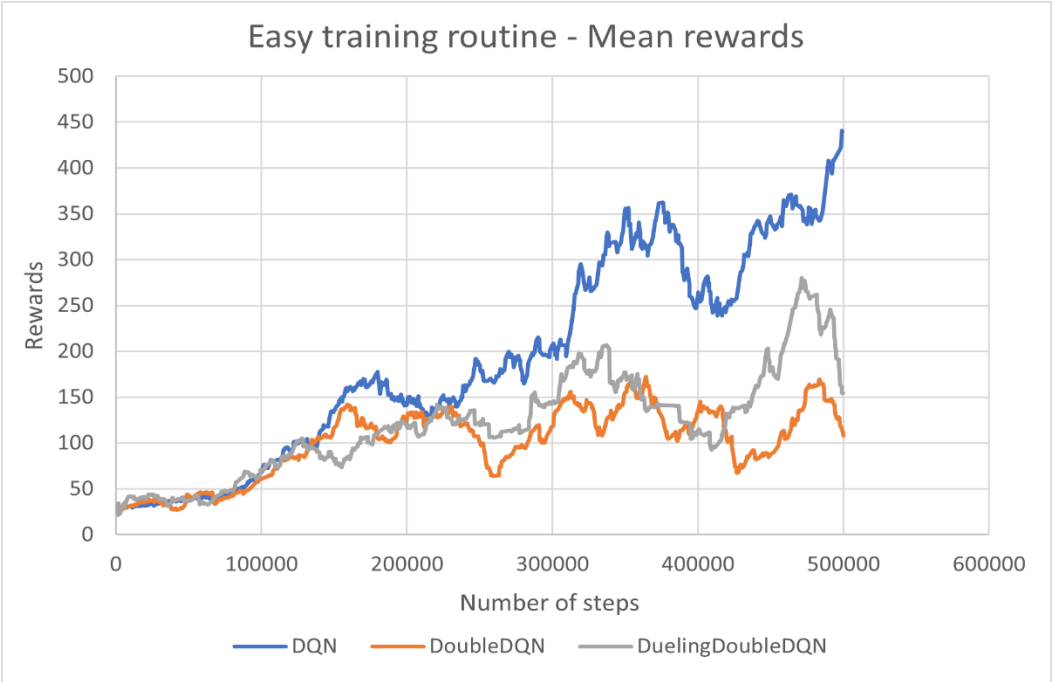


Figure 36. 'Demon attack' -Easy training routine results - mean rewards

The first routine is 'Easy', and it is clear from the graph that the simplest DQN agent has collected the most reward during training. Furthermore, it is evident that the agents have not found a policy close to the optimal one, as peaks and dips appear throughout training, as opposed to the training results of 'Pong' were the best average reward was increasing smoothly. The agents show signs of improvement overtime and manage to surpass a random agent; they are limited however in their performance. It important to point out that the DoubleDQN and the DuelingDoubleDQN have concluded their training with a smaller average reward than the peaks they achieved. The DoubleDQN achieved a high reward of 172 but finished with 108. This is another sign that the training routine is limiting the learning potential. The drops can be attributed to the agents reaching a completely new state of the game that they have not seen before and thus cannot act well in it. For this game, a new state is a new wave of enemies, that the agents must adapt to, in order to continue

earning rewards. So periodically the agents show reduced performance as they adapt to the new states. The highest reward peaks are also presented in a table. The simplest DQN approach is the best for this game and setting.

Agent type	Highest mean reward
DQN	440.85
Double DQN	172
Dueling Double DQN	280

Table 11. Demon attack - Highest reward achieved by each agent in the easy training routine

Agent type	Final Mean reward
DQN	439.4
Double DQN	108
Dueling Double DQN	154.4

Table 12. Demon attack - Final Mean reward achieved in the easy training routine

"Hard" training routine evaluation

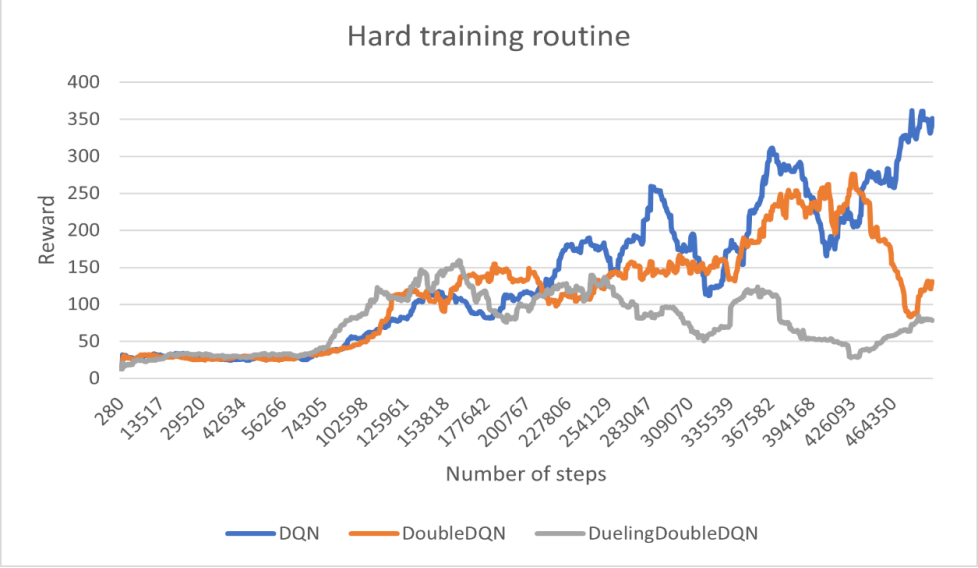


Figure 37. Demon Attack - Hard training routine results - mean rewards

Agent type	Final Mean reward
DQN	**351.4**
Double DQN	**130.9**
Dueling Double DQN	**78.35**

Table 13. Demon Attack - Hard routine final mean rewards

The training results of the ‘Hard’ routine show similarities to the ‘Easy’ routine results. Here, again the simple DQN achieves the highest mean reward out of all agents, with the more complex agents finishing the training routine with less than half of the reward of the simple agent and failing to cope well with the changes in the environment, as a result of new waves of enemies being introduced. The DoubleDQN agent was coping well up until the 450kth step, but its performance took a noticeable hit after that and was not able to recover. The simple DQN shows performance hits during training, however it was able to cope and increase its performance in later steps. Compared to the ‘Easy’ setting, the DQN agent achieved 100 points less, this was expected as this is a higher difficulty setting. These results again illustrate that the training parameters (i.e. max_steps, epsilon_decay, etc.) are not optimal to this game. In fact, for agents to find a good winning policy must be trained on the environment for a longer amount of time. Lastly, the results up to now show that the more complex approaches of Double Q-learning and the Dueling architecture do not offer any performance benefits, which could’ve been the case if the agents were allowed to run for millions of steps.

Curriculum training routine evaluation

The curriculum training setup for Demon Attack is identical to that of Pong.

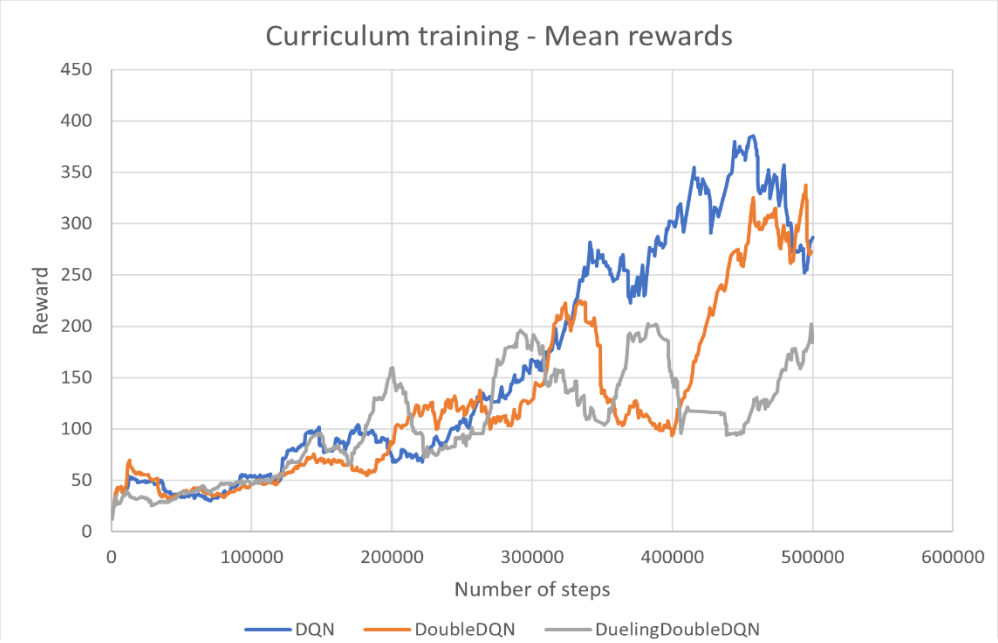


Figure 38. Demon Attack - Curriculum training routine - mean reward results

Agent type	Final Mean reward
DQN	**286.6**
Double DQN	**272.5**
Dueling Double DQN	**184.6**

Table 14. Demon attack - Curriculum training routine final mean rewards

The curriculum training routine once again provides the most interesting results out of all training routines. The curriculum seems to have a different impact on each agent. All agents can explore the environments a lot more, with the change to the epsilon-decay parameter and by the 400kth step, the simple DQN agent exhibits good performance, however that changes closer to the end of training routine with the agent exhibiting a hit in performance, possibly due to a new wave of enemies being introduced. Perhaps the agent would do a lot better and set a new reward high if it were allowed to continue. The DoubleDQN agent has taken advantage of what it has learned from the 'Hard' setting between steps 200k and 400k, since the performance increases drastically as it enters the 'Easy' setting again between

steps 400k and 500k. This is an example of an agent using what it has learned in the harder setting, to make good decisions on the easier setting. The DoubleDQN agent reached its highest reward with this mixed curriculum. The DuelingDoubleDQN agent has benefited from this mixed curriculum too since it achieved a higher reward here compared to the other training routines.

3.5.8 Evaluation of testing results and outcomes of the experiments.

The final evaluation test to determine the role of difficulty in training the agents is to run 100 test episodes on the 'Easy' environment. This time the exploration parameter is kept constant at 0.05, which means that the agents can take random actions 5% of the time. This was done to combat weird behavior of some agents, in particular the DoubleDQN agents can take anywhere from 1k steps to 98k steps to complete a single episode, this happens because the agent-controlled ship gets stuck on the left side of the screen with nothing to shoot at, as the enemies gather on the right side of the screen. This is further evidence that the agents all need more time to explore the environment. Two types of graphs are presented here. The episode rewards and the cumulative reward over the 100 test episodes.

DQN agents testing performance

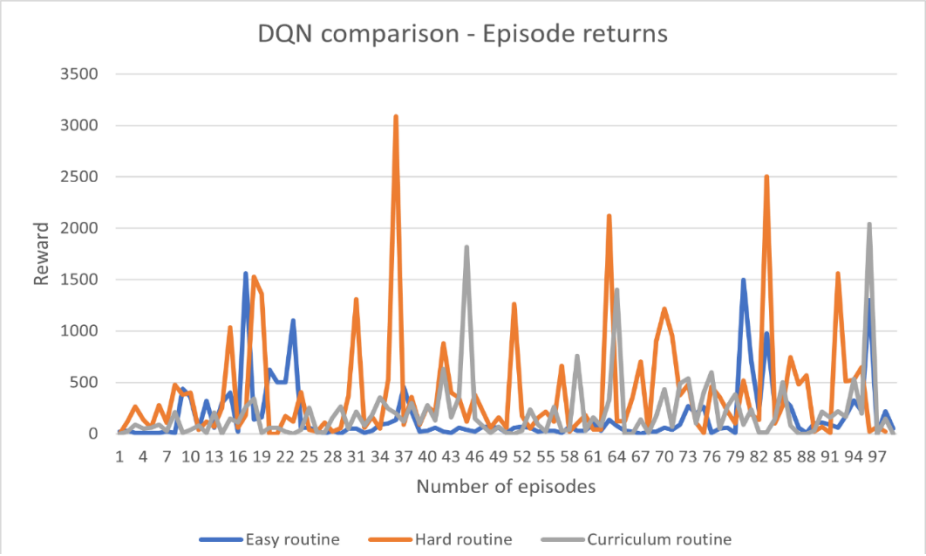


Figure 39. Demon Attack - DQN testing results - individual episode reward

The individual episode rewards can vary drastically as is evident from the graph. An episode reward can range from a 0 reward to an infinite reward as long as the agent does not lose a life. This explains the drastic difference in rewards. To illustrate this further, the DQN agent trained on the 'Hard' routine was able to achieve a score of over 3000 on the 36th episode, this means that either the agent managed to kill high-level enemies that appear in later waves in the game, or the agent started from the beginning of the game and without losing a single life was able to clear several waves of enemies it was hit, which resulted in the loss of a life. This metric, even though it is noisy, shows for the first time, that an agent trained on the 'Hard' routine, can outperform an agent of the same type that is trained on the 'Easy' routine, and the peaks it reaches are far higher than the other agents. A better representation of agent learning is the cumulative reward the agents receive throughout testing.

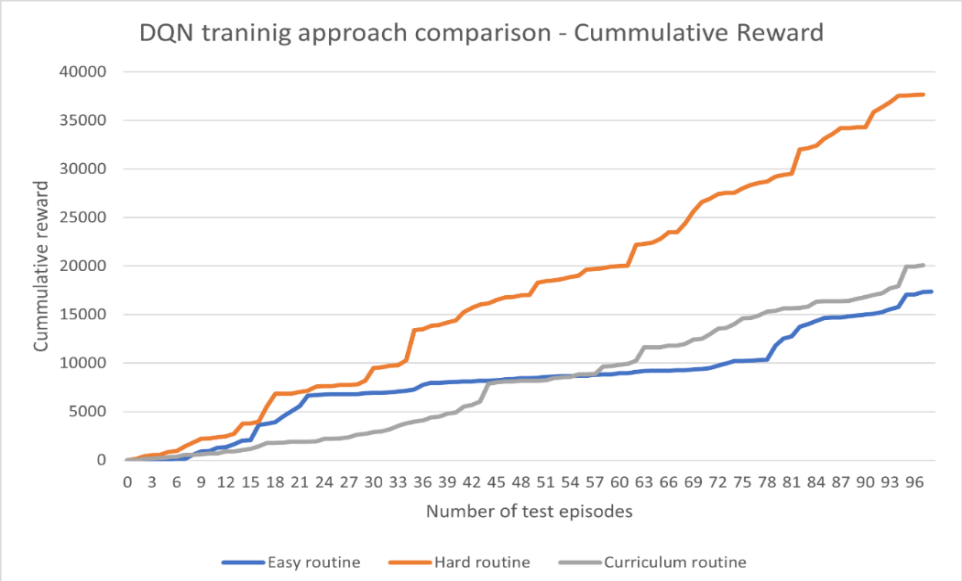


Figure 40. Demon Attack – DQN testing results – cumulative reward.

Training routine	Final Cumulative reward
Easy	**17,355**
Hard	**37,660**
Curriculum	**20,010**

Table 15. Demon Attack - Cumulative reward achieved by DQNs

The results show that the simple DQN agent performs better on the easy setting when it is trained on the harder setting. This is a first in all the results presented up to now and it is the opposite of what transpired in the Pong environment. A reason for this could be the fact that the state space for Demon Attack does not change permanently as in Pong, where the opponent's paddle gets smaller. The same exact enemies appear; however, they are more dangerous and aggressive which has prompted the agent to learn a better strategy to take out these enemies by moving around a lot more. This contrasts with the agent trained on the 'Easy' routine which stays on the left side of the screen and limits the number of points it can accumulate. This agent follows a suboptimal optimal policy that is to avoid the enemies while at the same time missing out on potential reward. This is most evident during episodes between 40 and 70 where the individual episode reward is extremely small.

The curriculum routine seems to alleviate this issue slightly with the DQN agent being able to achieve a slightly bigger cumulative reward. In this instance the curriculum training routine showed that it can help with overfitting problems.

DoubleDQN agents testing performance

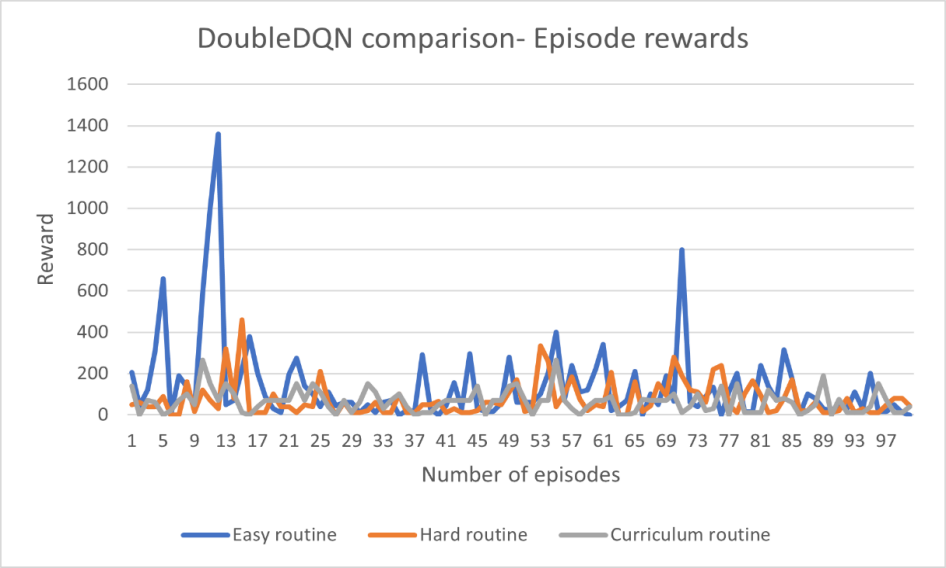


Figure 41.Demon attack - DoubleDQN testing results, individual episode rewards.

Like the simple DQN results, the individual episode rewards can vary greatly for the DoubleDQN agents. The DoubleDQN agent that was trained on the 'Easy' routine reaches the highest peaks, with the agent trained on 'Hard' following second and the curriculum agent finishing third. This is a reversal of the simple DQN results.

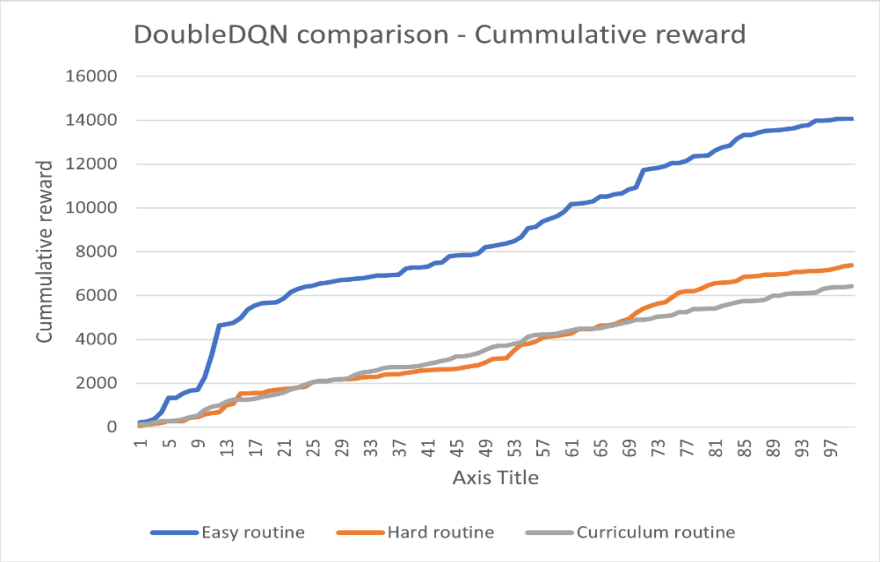


Figure 42.Demon attack - DoubleDQN testing results, cumulative reward.

None of the DoubleDQN agents managed to achieve a higher reward than their simpler DQN counterparts under the same training routine. This time, the agent trained on the 'Easy' curriculum outperforms the other agents substantially, in fact it achieves nearly double the reward of the agent that was trained on the 'Hard' setting. These results indicate that the use of Double Q-learning did not result in a better policy on this game and that the harder setting is not beneficial to a DoubleDQN agent. The curriculum routine is the worst performing out of the 3 training routines.

Training Routine	Cumulative reward
Easy	14,065
Hard	7,385
Curriculum	6,430

Table 16. Demon attack – Cumulative reward achieved by DoubleDQNs.

DuelingDoubleDQN agents testing performance.

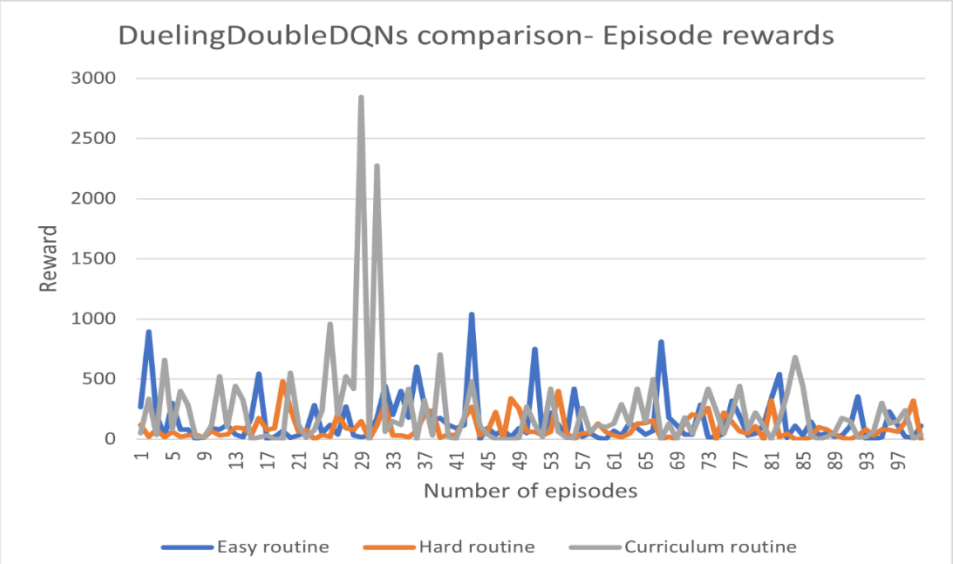


Figure 43. Demon attack - DuelingDoubleDQN testing results, individual episode reward.

The results show that the DuelingDoubleDQN agent benefits substantially from the mixed training curriculum, in fact, the highest single episode reward was achieved by this agent. The agent trained on the 'Easy' routine once again managed to accumulate more reward than the agent trained on the 'Hard' routine.

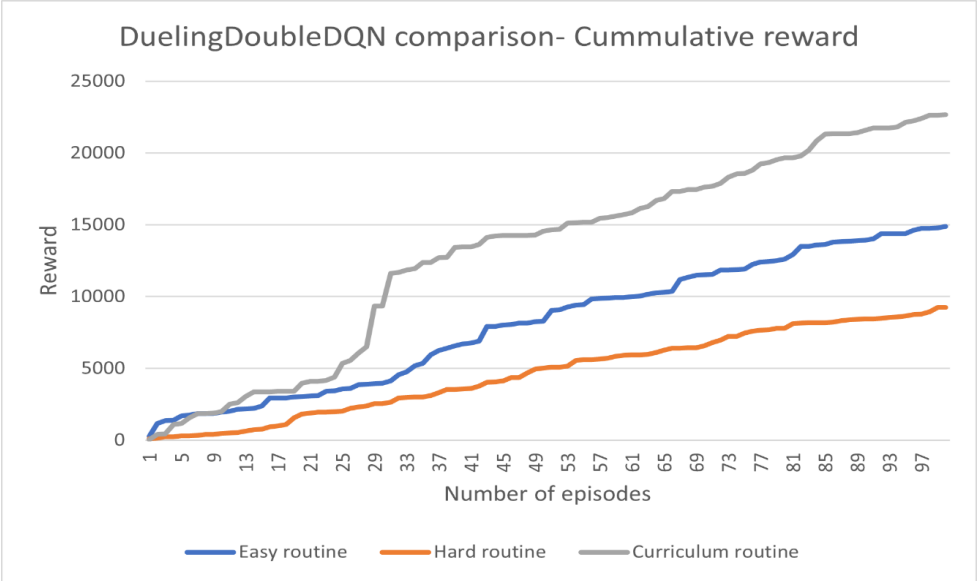


Figure 44. Demon attack - DuelingDoubleDQN testing results, cumulative reward.

The DuelingDoubleDQN agent trained on the curriculum routine can reach rewards comparable to those reached by the simple DQN architecture. In fact, the DuelingDoubleDQN agent trained on the curriculum routine achieves higher reward than the simple DQN agents that were trained on the 'Easy' and the curriculum routines. It is clear the more complex agents benefit greatly from curriculum training.

Training routine	Cumulative reward
Easy	**14,880**
Hard	**9,250**
Curriculum	**22,680**

Table 17. Demon attack – Cumulative reward achieved by DuelingDoubleDQNs

Complied testing results on Demon Attack

Overall, there are 9 trained agents. The results are given below. The final cumulative reward is given for 100 test episodes, along with the final mean reward. Due to the nature of the game, the cumulative reward is a better indicator for an agent performance.

Agent Type	Training Routine	Final Cumulative reward	Final Mean Reward
DQN	EASY	17,355	255
DoubleDQN	EASY	14,065	51
DuelingDoubleDQN	EASY	14,880	96.5
DQN	HARD	37,660	345.5
DoubleDQN	HARD	7,385	41.5
DuelingDoubleDQN	HARD	9,250	81.5
DQN	CURRICULUM	20,010	368
DoubleDQN	CURRICULUM	6,430	42.5
DuelingDoubleDQN	CURRICULUM	22,680	110.5

Table 18. "Demon Attack" Compiled testing results

Overall, the results show that the best performer is a simple DQN trained on the 'Hard' setting. This is the opposite of what transpired in Pong. The second most successful approach is the pairing of a DuelingDoubleDQN with the curriculum routine, and the third most successful approach is the use of the curriculum routine on a simple DQN agent. These

results show the benefits of mixing hard and easy environments during training. One of the ways that the more difficult environment has helped the agents learn a better policy, is to force them to adopt a more aggressive playstyle which is more rewarding.

To sum up, the results achieved in the second experiment are different from the first, as they show that the best performing agent is a simple DQN agent trained on the harder setting. The more complex agents however do not benefit from the 'Hard' training routine. The curriculum approach is once again proven to be beneficial as the agents that follow this curriculum generally benefit from it compared to the 'Easy' training routine. The 'Easy' training routine was found to be a lot more stable in terms of producing well performing agents, as extreme fluctuations of final cumulative rewards are not present as is the case with the other training routines.

Chapter 4. Wholistic discussion of experimentation results.

4.1 Project aims and holistic answer to the main research questions

The aims of the project are also provided here again for clarity.

- Investigate the role of game difficulty in training AI for classic arcade games.
- Experiment with different training strategies and schedules and explore whether the use of harder difficulty settings during training can prove to be beneficial.
- Explore whether agents trained on high difficulty settings have an edge, compared to their counterparts trained under easier difficulty settings.

The original aims as lay out in the proposal have been explored by the analysis carried out on the results of the experiments. The results show that a silver bullet approach does not exist and simply training an agent on the harder setting, does not guarantee good results in the easy setting. Furthermore, the results also show that a mixed training curriculum can prove beneficial to agent training in some settings. This suggests that even though, an agent

trained strictly on a harder setting can't be expected to always perform well under the easy setting, the policy learned on the harder setting can be refined to produce good results. By giving the agent leeway to learn from the easy setting for a while, without having to explore it too much (epsilon parameter set to 0.0), it can learn to quickly adapt to the setting. Lastly since the results vary across games, it is clear that the individual game environments play a crucial role as to weather a 'Hard' training routine can outperform an 'Easy' one. This is a theme that can be found in RL repeatedly, as there is no silver bullet approach for anything, rather carefully considered recipes can get the job done.

4.2 Discussing the success of the objectives

The original project objectives are restated here in the form of a table along with the testable outcomes for each one to be considered met.

Objective	Testable Outcome
Identify suitable Deep RL learning algorithms for Classic Arcade Games	List of RL algorithms that can achieve at least as good performance as a simple DQN
Identify suitable training strategies for the learning algorithms.	List of 3 training strategies that will be used to train the RL algorithms. One of which will employ curriculum learning
Design and develop at least 2 learning algorithms that can exhibit good performance in at least 3 games.	List of RL algorithms that perform well on 3 chosen game. Excluding the vanilla DQN
Compare and contrast RL algorithms in terms of performance and discuss the best results.	Detailed ablation study that provides comparisons broken down by game, training strategy and DQN architecture.

Table 19. Project objectives and testable outcomes, as outlined in the project proposal document.

The objectives laid out in at the beginning of the project have largely been achieved. Enough research was carried out for several advanced Deep Q agents to be designed and implemented. These agents were able to achieve close to a perfect score in one of the games and exhibited good performance on the other. They were also able to achieve good performance on both easy and hard settings of the games. Their performance was analyzed using a detailed ablation study that considered the training routine and the agent architecture. The performance impact that the higher difficulty levels had on the agents was analyzed and evaluated. Due to the computational requirements however, the experiments were not able to be replicated on other games, which could have added to the credibility of the results. In fact, the only other game in which the agents were able to exhibit substantial improvement in a relatively short amount of time is "Tennis", with a slight change in the training hyperparameters. Furthermore, for this piece of work to be considered a thorough effort in studying the effects of difficulty in Atari games, state of the art results should be achieved in the Atari benchmark. That would provide much more confidence in the results.

4.3 Project deliverables

The trained agents were saved and are part of the project deliverables along with the training and testing coding scripts. The training and testing runs that were used to produce the graphs are also part of the deliverables and can be analyzed using Tensorboard. An example of this is given in the appendices. The trained agents can be tested out of the box using the testing scripts. A list of coding environment requirements is also provided in the appendices along with details about the hardware used to run the experiments. Lastly videos that show how agents behave in the game environments are also part of the deliverables of this dissertation.

Chapter 5. Reflection on the work undertaken, conclusions and future directions.

5.1 Conclusions and reflections

The dissertation overall had a successful, but at the same time limited outcome in terms of generalizability of the results. This is mostly due to the training routines being suitable for only a small subset of the Atari benchmark. Much of the time was spent on writing the code for the training and testing routines and finding good training hyper-parameters. A lot of time was spent on trying to find a common set of training hyper-parameters that can produce results in a wide variety of games, however this was deemed unattainable as the training time for a single agent run would require days to complete. Thus, not enough time was spent to exploring different NN architectures and optimization methods that could produce better results, as time was limited. Furthermore, it became obvious at the beginning on the project that the effort required to train agents to perform on a good level on different games was much greater than anticipated. The literature used for this dissertation was enough to provide a good grounding in basic DQN methods, although a lot more interesting approaches to Deep Q-learning exist [20].

The work presented here does not provide conclusive evidence that an agent trained on a higher difficulty setting will outperform an agent trained on the easy setting, however it offers a good start point to think about the games where it would be possible. Furthermore, this piece of work demonstrated that a carefully, manually crafted curriculum learning approach can prove very useful in training RL agents, and on many occasions, it can outperform the simple training routines. This work could be beneficial to a broader audience that tries to make RL agents that can generalize to a slightly different setting than their training environment. The topic of generalizability in RL is hugely popular, especially the off-policy kind of learning. A concrete example would be an artificial robot being able to

walk on the sidewalk, after having been trained to walk only on the rough terrain of a small hill. Would the robot be able to walk on the pavement without any problems?

A major limitation encountered throughout the course of this work, is the high computational requirements to run these agents. Furthermore, setting up the training and testing routines required extra care, as buggy routines could comprise the experiment results. For that reason, extra care was put into the code following test driven development principles. The Google Collab platform was used to run the experiments for "Demon Attack", which helped significantly, while the experiments on 'Pong' were run locally on a Windows machine. The workload was split between the two, which required subtle changes to be made to the code, to make it work for a Linux environment.

5.2 Future direction for the work

Had more time been available, the next step would be to implement another advancement to the DQN algorithm, namely prioritized experience replay[21]. This addition to the DQN algorithm, changes the way transitions are samples from the experience replay buffer. The intuition behind this method is that the agent should train more on situations that it has poorly performed in. So, by training the agent on experiences that proportional to their TD error, the agent is learning to act in situations where it exhibits the biggest deficiencies. Another interesting idea would be to experiment with a Recurrent Neural Network (RNN) architecture, which is designed to deal with processes that happen overtime, in other words, sequences of data. This set up would be interesting to explore as the information of many previous states can be used by the NN to decide what to do in some state, since a game can be thought of as a sequence of states. This could help to deal with partially observable MDPs. Lastly this piece of work only concerned itself with the DQN algorithm, so the natural next step would be to use Policy Gradient algorithms which also provide state of the art results in RL problems.

Glossary

Agent: An entity that learns how to make decisions from interacting with the environment.

Environment: Anything that falls outside of the agent. The environment provides the observations and the reward signal that should be given to the agent.

Reward: The reward signal is the feedback given to the agent in response to an action that it has taken. This is usually a scalar value. It is denoted by r.

Return: The accumulated reward that an agent collects from a series of interactions with the environment, usually at the end of an episode. It is denoted by R.

State: Is a single configuration of the environment at a specific timestep. It is either partially or fully observable by the agent. It is denoted by s.

Action: The mean by which the agent communicates with the environment. The action of an agent results in a different state of the environment. It is denoted by a.

State-Value function: A function that determines the value of a state based on the expected reward that the agent can receive from it under a specific policy. It is denoted by $V(s)$. It usually outputs a scalar value.

Action-Value function: Also known as the Q function. It is a function that indicates how good it is for an agent to take a specific action in a specific state under a specific policy. It is denoted by $Q(s, \alpha)$. It usually outputs a scalar value.

Policy: A mapping between states and actions. The agent uses a policy at each state to decide which action should be taken. Can be either stochastic or deterministic. The optimal policy yields the highest reward. It is denoted by π.

Tabular Learning: A approach which uses a lookup table to store in memory the value of a state-action pair. The table is replaced by a NN in the Deep RL setting.

E-greedy policy: An exploration method that is used by agents to decide whether they should take an action they know will yield a high reward or explore they environment by trying a new action.

Discount factor: A numerical value, usually between 0.9 and 0.7. It serves as a level of confidence that the agent has about the reward it is going to achieve by taking an action. Denoted as γ.

Action space: The set of all possible actions available to the agent. The individual action is usually denoted as a and the action space as A, with $a \in A$**.**

Deterministic policy: A deterministic policy maps one state to one action. If an agent is following a deterministic policy, every time that it observes state s, it will always take the same action.

Stochastic policy: A stochastic policy maps a state to a probability distribution over the action space. This means that if an agent observes the same state more than once, it is able to take an action, with some given probability.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. Human-level control through deep reinforcement learning.Nature, 518(7540), pp.529-533.

[2] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. and Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. *nature*, *529*(7587), pp.484-489.

[3] Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*. MIT press.

[4] 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.

[5] Badia, A.P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D. and Blundell, C., 2020. Agent57: Outperforming the atari human benchmark. *arXiv preprint arXiv:2003.13350*

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

[7] Machado, M.C, Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M. and Bowling, M., 2018. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research, 61, pp.523-562.*

[8] Graesser, L. and Keng, W.L., 2019. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Professional.

[9] Goodfellow, I., Bengio, Y., Courville, A. and Bengio, Y., 2016. *Deep learning* (Vol. 1, No. 2). Cambridge: MIT press.

[10] Raschka, S., 2015. *Python machine learning*. Packt publishing ltd.

[11] Lapan, M., 2018. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd.

[12] Ravichandiran, S., 2020. *Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL and more with OpenAI Gym and Tensorflow.* Packt Publishing Ltd.

[13] Van Hasselt, H., Guez, A. and Silver, D., 2015. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*.

[14] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N., 2016, June. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (pp. 1995-2003). PMLR.

[15] Bengio, Y., Louradour, J., Collobert, R. and Weston, J., 2009, June. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning* (pp. 41-48).

[16] Narvekar S., Stone P., 2019. Learning Curriculum Policies for Reinforcement Learning, *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems(pp. 25-33). International Foundation for Autonomous Agents and Multiagent Systems*

[17] Matiisen, T., Oliver, A., Cohen, T. and Schulman, J., 2019. Teacher-student curriculum learning. *IEEE transactions on neural networks and learning systems*.

[18] Gym.openai.com. 2020. Gym: A Toolkit For Developing And Comparing Reinforcement Learning Algorithms.Gym.openai.[online] Available from: https://gym.openai.com/docs/[Accessed 5 December 2020].

[19] Machado, M.C, Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M. and Bowling, M., 2018. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research, 61, pp.523-562.*

[20] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2017. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*.

[21] Silver D., Antonoglou I.,Schaul T.,Quan T., 2016. Prioritized Experience Replay, ICLR ARXIVarXiv e-prints, arXiv:1511.05952

Appendices

Appendix A. Tools and software used.

Tools used to develop the code. A requirements file that describes the environment is part of the deliverables.

Operating System: WINDOWS

Hardware specs:

RTX 2060 GPU, 6 GB of dedicated memory.

6-Core Intel 9th Gen CPU

Software:

- Pytorch
- Numpy
- Tensorbaord for tracking the experiments, version.
- Python 3.7
- Spyder IDE
- Vscode IDE
- Google Colab for 'Demon Attack' experimentation

Appendix B

B.1 Pytorch gradient computation for the MSE loss function.

Calculating the loss MSE loss and updating the NN weights.

The output of the NN will be a matrix of size (batch_size, action_space). Action space could be "up","down","left",etc. The task is framed as a regression problem using the MSE loss function. This means that the loss takes in single values, instead of tensors. The loss takes in two scaler values

a. The output value given from the NN for the action that was taken
b. The target value, which is constructed by the Bellman equation.

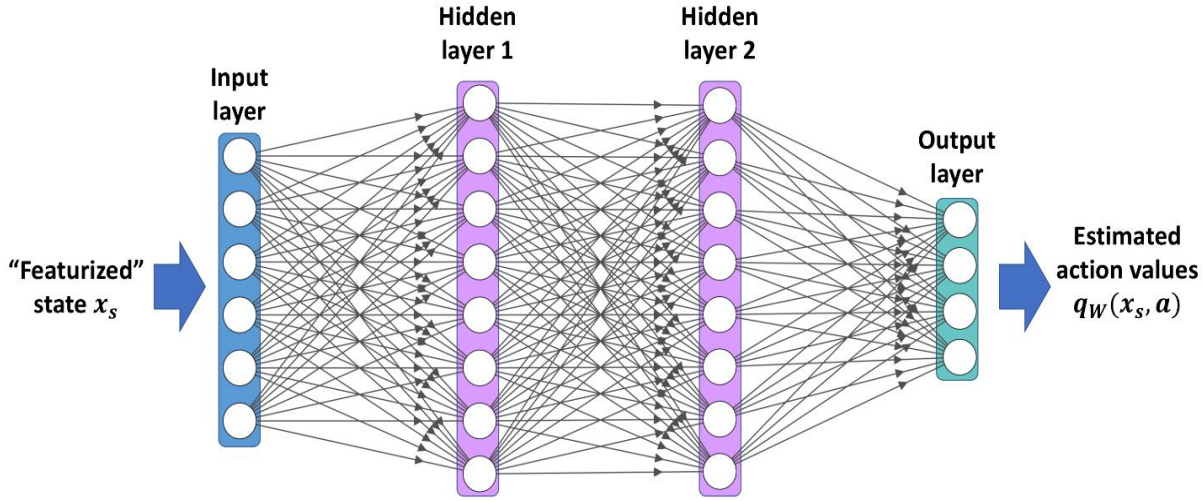


Figure 45 Output of the DQN is a vector, however only the node that represents the action taken is considered for the loss function. Raschka, S., 2015. Python machine learning. Packt publishing ltd.

On back-propagation, the only node that should be used to back-propagate the error is the one representing the action that was taken, this is the only node with a non-zero loss, and thus a non-zero gradient. The only weights to be updated, between the final hidden layer and the output layer, will be the ones projecting onto that node. A visual example is given below along with the code that optimizes the NN. The object variables are illustrated with visual examples.

4.0	2.5	2.9
1.2	1.5	1.9
5.0	5.5	5.9
4.8	2.5	2.9
4.0	2.8	3.9

FIGURE 47. BATCH OUTPUT OF THE DQN. 5 STATES WITH 3 AVAILABLE ACTIONS "TENSOR STATES"

2
1
1
0
1

FIGURE 46. ACTION INDICES, REPRESENT WHICH ACTION WAS TAKEN. PART OF THE <SARS> TUPLE

4.0	2.5	2.9
1.2	1.5	1.9
5.0	5.5	5.9
4.8	2.5	2.9
4.0	2.8	3.9

FIGURE 49. THE Q VALUES THAT WILL BE USED TO CALCULATE THE LOSS

2.9
1.5
5.5
4.8
2.8

FIGURE 48. RESULT OF GATHER() OPERATION, ON THE BATCH OUTPUT. THIS IS WHAT IS FED TO THE LOSS FUNCTION

Loss = (predicted_state_action_values,target_state_action_values)

The loss is calculated between the predicted state action values and the target state action values, which are created using the Bellman equation. Once the loss has been calculated, the gradients can be calculated to update the NN weights.

By using the loss.backward() function the optimizer calculates the partial derivatives of the loss function with respect to the predicted values, and the weights that have to be updated.

1.3
0.63
2.01
3.03
2.02

FIGURE 50. PARTIAL DERIVATIVE CALCULATED WITH RESPECT TO EACH PREDICTED Q-VALUE

0	0	1,3
0	0.63	0
0	2.01	0
3.03	0	0
0	2.02	0

FIGURE 51. PARTIAL DERIVATIVE CALCULATED WITH RESPECT TO EACH PREDICTED Q-VALUE. FLOW BACK TO THE WEIGHTS TO BE UPDATED.

In code this process is done with the following functions:

```
    predicted_state_action_values = main_net(tensor_states).gather(1, tensor_actions
.unsqueeze(-1)).squeeze(-1)
```

The target values are calculated by passing the next states of the <SARS> tuple through the target network and taking the maximum Q-value. If there is no next state because the transition is terminal then there is no Q-value.

```
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_states
).max(1)[0] #only take the values and not the indices
```

The target values are added to the reward received, again part of the <SARS> tuple, and are discounted by gamma. The "target_state_action_values" variable has the same dimensionality as the "predicted_state_action_values".

```
    #target values using Bellman aproximation
    target_state_action_values = tensor_rewards + (gamma * next_state_values )
```

The loss is calculated using the Pytorch defined MSE loss function.

```
criterion = nn.MSELoss()
```

```
return criterion(predicted_state_action_values,target_state_action_values)
```

Small snippet on how Pytorch calculates gradients.

Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

```
>>> import torch
>>> import torch.nn as nn
>>> predictions = torch.tensor([0.235,1.200,1.321], requires_grad = True)
>>> predictions
tensor([0.2350, 1.2000, 1.3210], requires_grad=True)
>>> targets = torch.tensor([2.335,0.890,1.154])
>>> targets
tensor([2.3350, 0.8900, 1.1540])
>>> prediction = predictions[1]
>>> target = targets[1]
>>> prediction
tensor(1.2000, grad_fn=<SelectBackward>)
>>> target
tensor(0.8900)
>>> loss = nn.MSELoss()
>>> output = loss(prediction, target)
>>> output.backward()
>>> prediction.grad
>>> target.grad
>>> predictions.grad
```

tensor([0.0000, 0.6200, 0.0000])

>>> targets.grad

>>> 2 * (prediction - target) [The Derivative of the MSE w.r.t. prediction works out to be 2(prediction-target)]

tensor(0.6200, grad_fn=<MulBackward0>)

Same approach, but using the gather() function. The gather function is used instead of simply indexing.

(base) C:\Users\billy>python

Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

>>> import torch.nn as nn

>>> import torch

>>> predictions = torch.randn(3,requires_grad = True)

>>> predictions

tensor([0.9243, -0.5061, -0.0116], requires_grad=True)

>>> **prediction = torch.gather(predictions,0,torch.tensor([0]))**

>>> prediction

tensor([0.9243], grad_fn=<GatherBackward>)

>>> target = torch.tensor([2.565])

>>> target

tensor([2.565])

>>> prediction.size()

torch.Size([1])

>>> target.size()

torch.Size([1])

>>> loss = nn.MSELoss()

>>> output = loss(prediction,target)

>>> output.backward() # Calculate gradients

>>> predictions.grad

tensor([-3.2815, 0.0000, 0.0000]) # The gradient is calculated and stored in a specific index. The optimizer then uses these gradients to update the values e.g. value += learning_rate * value.grad

>>> target.grad (This is empty as the 'target' is never updated)

>>> 2*(prediction-target) [This again is the partial derivative w.r.t. the MSELoss function.]

tensor([-3.2815], grad_fn=<MulBackward0>)

requires_grad: This member, if true starts tracking all the operation history and forms a backward graph for gradient calculation.

grad: grad holds the value of gradient. If requires_grad is False it will hold a None value. Even if requires_grad is True, it will hold a None value unless .backward() function is called from some other node. For example, if you call out.backward() for some variable ***out*** that involved ***x*** in its calculations then x.grad will hold ***∂*out/*∂*x**.

Backward() simply calculates the gradients

The .backward() function is called on the error!

Full example of the Q-loss function code.

This is the ‘Naïve’ version which loops every sample in a batch.

```
criterion = nn.MSELoss()

def naive_dqn_loss_V2(batch, main_net, target_net, gamma, batch_size = 32, devic
e = "cuda", optimizer = 'opt'):

    tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards,non_fi
nal_mask = batch

    # create predicted (NN output)
    predicted_state_action_values = main_net(tensor_states) # dims = batch_size,nu
mber_of_actions

    next_state_values = torch.zeros((batch_size,predicted_state_action_values.size()[
1]),  device=device)

    #print(next_state_values)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_states
) #
        next_state_values = next_state_values.detach()

    next_state_values = next_state_values * gamma

    batch_loss = 0.0

    for prediction, reward, action,  tensor in zip(predicted_state_action_values,tensor
_rewards,tensor_actions, next_state_values):

        target = tensor.max() + reward


        target = target.detach() # detach, no gradient flow to the ground Truth!!!
        prediction = prediction[action]

        # scalar, NOT vector inputs to the loss function!
        sample_loss = criterion(prediction,target)
```

```
        batch_loss += sample_loss

    return batch_loss / batch_size
```

The batch version of the Loss, using the gather() function on the tensor_states() matrix, with tensor_actions as indices.

```
def simple_dqn_loss(batch, main_net, target_net, gamma, batch_size = 32, device="cuda"):

    tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards,non_final_mask = batch

    predicted_state_action_values = main_net(tensor_states).gather(1, tensor_actions.unsqueeze(-1)).squeeze(-1)

    next_state_values = torch.zeros(batch_size, device=device)

    with torch.no_grad():
        #compute next state values using Target network
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_states).max(1)[0] #only take the values and not the indices
        #detach from pytorch computation graph
        next_state_values = next_state_values.detach()


    #target values using Bellman aproximation
    target_state_action_values = tensor_rewards + (gamma * next_state_values )


    return criterion(predicted_state_action_values,target_state_action_values)
```

B.2 Tensorboard example output.

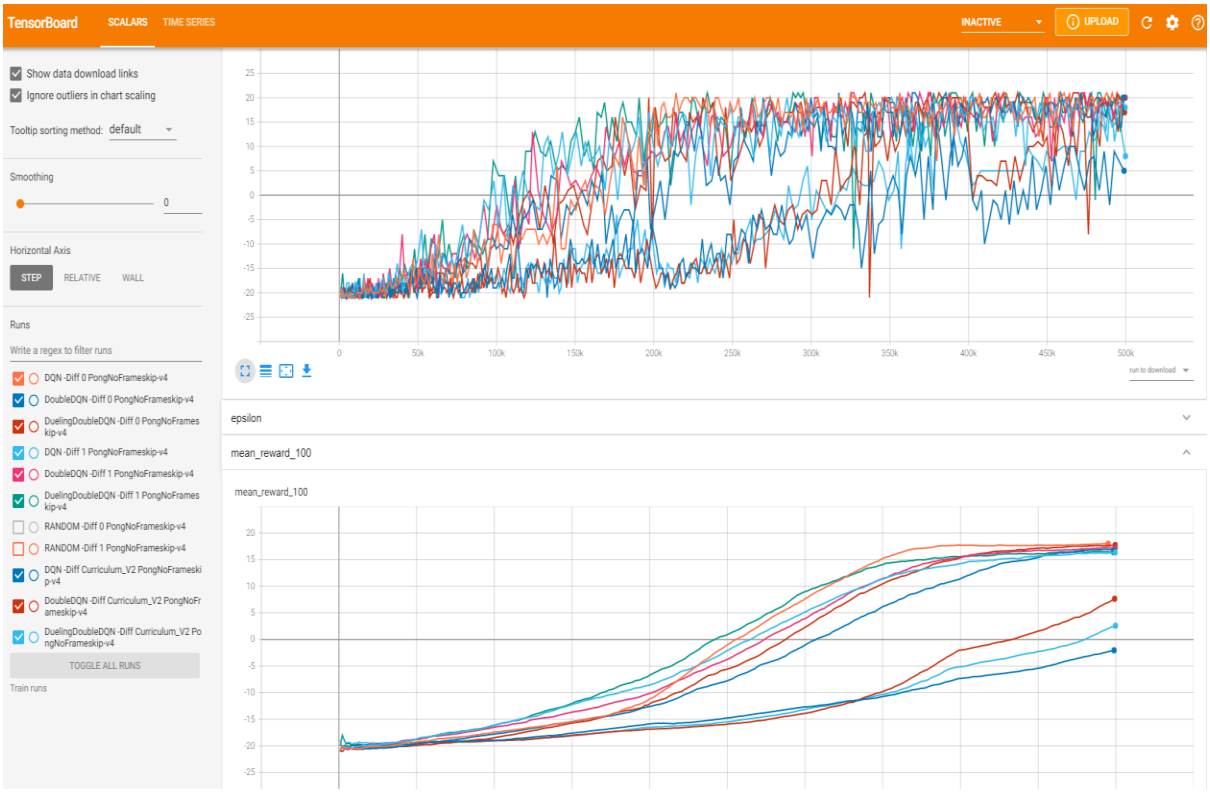


Figure 52. Tensorboard panel. A great way to track the experiments and the training process results. All training and testing runs are included in the deliverables.

All training and testing results can be run through Tensorboard by running the following command:

```
tensorboard --logdir 'name_of_diectory_that_holds_the_runs'
```

Appendix C Project Proposal

The original project proposal is attached here.

"THE ROLE OF GAME DIFFICULTY IN TRAINING DEEP RL AI FOR CLASSIC ARCADE GAMES"

Project Proposal

Course: Artificial Intelligence Masters

Student name: Kotsos Vasilis

Student Number:190033787

Content Page

Figures:

Tables:

Introduction

The explosion of deep learning over the past decade has resulted in several advancements in areas that previously computers performed very poorly in. Computer vision, text processing and speech recognition are a few areas that have been experiencing a great deal of advancement. The common denominator behind all these advancements is Deep Learning, and more importantly the Neural Networks that power it. Reinforcement Learning is a subfield that has seen rapid due to the advent of Deep Learning. The seminal paper published in Nature by Deepmind[1], demonstrated how Deep NNs can be used to achieve human level control on simple Atari games. Deep Reinforcement Learning is now a very hot research area with many open topics to explore. The topic that this proposal will focus on, is the role that game difficulty plays in training deep RL algorithms for arcade games. Although remarkable progress has been made in "solving" these games, the role that game difficulty plays in the training process is an underexplored topic. From a bird's eye view, the objective of the project will be to train agents on varying difficulty levels and compare them against each other in a test environment of low difficulty. This set up will allow for the thorough examination of the intuitive idea that agents which have been trained on harder difficulty, should exhibit better performance than the ones trained on lower difficulty when they are tested under easy difficulty.

Aims of the Project

1. Investigate the role of game difficulty in training AI for classic arcade games.
2. Experiment with different training strategies and schedules and explore whether the use of harder difficulty settings during training can prove to be beneficial.
3. Explore whether agents trained on high difficulty settings appear to have an edge, compared to their counterparts trained under easier difficulty settings.

To fulfil the aim of this project, the following objectives have been set:

1. Identify suitable Deep RL learning algorithms for Classic Arcade Games
2. Identify suitable training strategies for the learning algorithms.
3. Design and develop at least 2 learning algorithms that can exhibit good performance in at least 3 games.
4. Compare and contrast RL algorithms in terms of performance and discuss the best results.
5. Compare and contrast different training procedures and investigate what impact the use of high difficulty setting has if any.

Critical Context

It is of paramount importance that the project idea is surrounded by a well-founded critical context. The context is detailed below.

The Arcade learning environment used by the DQN papers [1][2] was released in 2013 with the purpose of providing a platform for evaluating AI technology using a standardized approach [3]. Prior to the release

of this paper reproducibility of RL experiments was a big issue, since no set of standard environments existed. This version however lacked the option of difficulty levels and as such, all games included in the emulator came with their default difficulty setting. The arcade environment was later revisited and expanded upon in 2017[4]. This revision included the option of several difficulty settings for some of the original games. This project aims to take advantage of those additions. The authors of the paper state that including higher difficulty modes essentially adds new environments that nevertheless have similar underlying state representations to their easier counterparts. This is also a key point to this project, since the main goal is to explore whether an agent can learn a policy on a high difficulty setting that can generalize on easier settings and if not, try to justify the results. A hurdle that might be tackled is the concept of overfitting [5]. Put simply in RL terms, overfitting is exhibited when the agent has learned to behave in a very specific manner in one setting (high difficulty setting) while performing very poorly under a different setting (same game but easier difficulty). Depending on the nature of the games used and whether there is a 2-player component, the concept of Multi-agent learning will be tackled, and thus appropriate training and testing methods will be employed [6]. Finally, the approach of curriculum learning could be used to explore whether a better policy can be found by exposing the agent to different difficulty levels during the training process[[7]. Curriculum learning mimics the human experience, by slowly exposing the agent to more difficult situations the agent should exhibit constant learning and improvement by building on past experiences, just as humans in real life. For the purposes of this project, the experiments will focus on advanced versions of DQN algorithms, as detailed in [8]. Rainbow DQN is a combination of useful add-ons to a simple DQN algorithm. More technical detail is provided in the next section.

Methods and tools for analysis and evaluation

This section will describe in detail the software tools that will be in the project. It also highlights some of the RL DQN algorithms that are under consideration.

The project aims to evaluate how algorithms trained on higher difficulties perform on easier settings. To achieve this, several tools and techniques will be used. The learning algorithms will be of Deep-Q Network variety and will ideally be able to achieve a good score across all games. To achieve this, the project will mainly focus on some variety of DQN, approaching Rainbow DQN [8]. The learning algorithms do not have to achieve state-of the art results, but performance should be sufficiently high. The Rainbow DQN algorithm expands upon the simpler DQN version by adding a few tricks such as a Dueling Network and prioritized experience replay, it has demonstrated major improvements in performance when compared to simpler DQN versions, specifically when applied to arcade games. The code will be written in python, more specifically the Pytorch deep learning library will be used to develop the RL algorithms. To develop the more advanced version of the DQN algorithms, reinforcement learning libraries that are built on top of Pytorch could be used, such as SLM Lab and PTAN (Pytorch Agent Net). The potential use of Tensorflow and its high level RL libraries will also be explored. The ideal software platform to use as an Atari game environment emulator is the OpenAI Gym library[10]. The OpenAI Gym is a toolkit that supports the development of RL algorithms by providing several environments that facilitate agent training. The toolkit includes many different types of environments, such as Toy text games and Classic control problems. This project will focus on the Atari game emulator environments that the toolkit provides.

The use of Deep RL in Arcade Games

This section offers details on how RL algorithms work in simple Arcade game environments. It also motives the use of Deep Neural Networks.

Simple algorithms used in RL, such as Q-learning, employ the use of tables to store information about states transitions and rewards these methods are called Tabular methods. Tabular Q-learning and the use of tables is not suited for playing any kind of arcade game, since the state space is prohibitively large to store in memory. As a quick reference, the Atari platform has a resolution of 210x160 pixels, and each pixel can take one of 128 colors [3]. The number of screens that are possible is 128^{33600}. This is the main reason why Deep RL algorithms are used to approximate the Q learning function. Instead of looking at a Q table to find which action is optimal for a specific state, a function can be used to map a state to an action. This learned function takes in states as input and outputs values for each available action. The action with the highest value can be though of as the optimal action to take in that state. A single state in a game can be represented as a single frame, for practical and theoretical reasons however, several frames are combined to form a complete state. This simple method allows the games to be solved as Markov Decision Processes (MDPs), which is a prerequisite for RL problems. Thich means that states can be distinguished from one another and that all information that is needed to take an optimal action in each state, is part of the state representation. For example, in Pong (Figure 1: Pong) a single frame cannot be used as a state, since it does not contain any information about the direction of the ball. More information must be used from subsequent frames to capture the information that is required for the agent to perform optimally (whether to move the paddle up or down). In Arcade games usually 4 sequential frames are combined to make up the state. While there may exist other longer-term dependencies in the environment, this method works well for simple arcade games. The features used to describe each state are extracted from batches of 4 images using Convolutional Neural Networks (CNNs). The use of fully connected networks is this setting is infeasible since as mentioned above a single frame is made up of 210x160 = 33600 pixels. If for arguments sake a fully connected input layer was to be used with 512 input units, the number of weights to be tuned would be 33600 x 512= 17,203,200, just for the first layer, so to keep the number of weights within reasonable bounds CNNs are used.

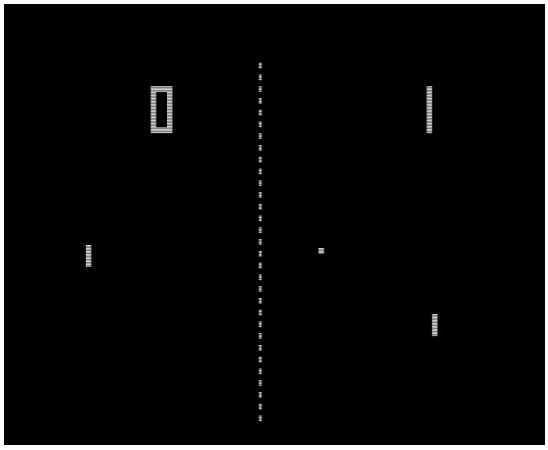


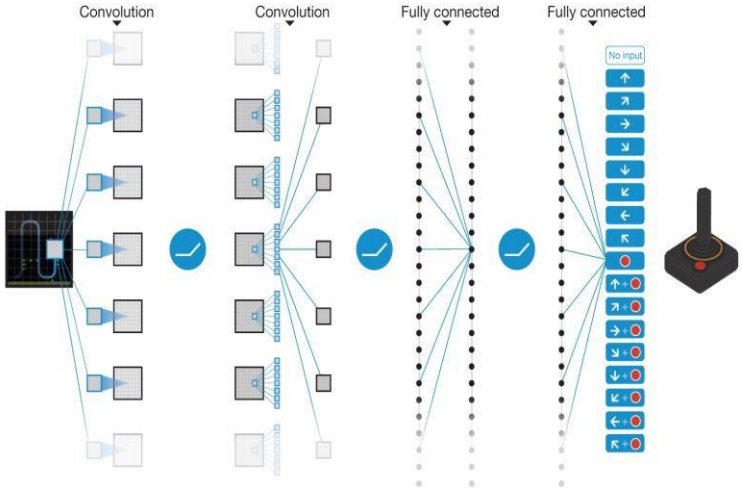
Figure 1: Pong

A good starting point for this project would be to emulate the Neural Network architecture presented in the original Deepmind paper (Figure 2. DQN Architecture V Mnih *et al. Nature* 518, 529533 (2015)). The network used was a CNN, trained with stochastic gradient descent. It also employed the use of an

experience replay buffer to help with creating training data that is independent and identically distributed. This is a requirement that must be met for the use of gradient descent. This version also used a target network to make the training more stable. Essentially a target network is a copy of the main network that is used to evaluate the maximum Q value of the next state. The target network is synchronized with the main network periodically. Finally, a simple e-greedy exploration method was used with a linear decay rate for e. using this method, the neural network will perform a random action or chose to exploit what it has learned. The agent at the start of training will chose to behave randomly most of the time (high e), while it will choose to exploit what it has learned close to the end of training (small e). The network architecture used in the paper is captured in the table (Table 1: Baseline DQN Neural)

Number of Layers	Optimization method	Activation function	Loss function
2 Conv + 2 Fully connected	Stochastic Gradient Descent (SGD)	RELU	Squared error (Regression)

Table 1: Baseline DQN Neural Network parameters



Built on top of this scheme, more advanced approaches can be pursued, such as using an n-step DQN which is simple unrolling of the Bellman equation which replaces one-step transition sequences with longer transitions of n-steps [8]. For a different exploration approach, noisy networks can be used. Noise is added to the fully connected layers of the DQN and overtime the noise is adjusted through back propagation.

Figure 2. DQN Architecture V Mnih *et al. Nature* 518, 529-533
(2015)

The noise added could be Independent or Factorized Gaussian noise. Finally, a significant improvement over a naïve sampling method from the replay buffer is the prioritized method. This method assigns priority to samples in the buffer according to which samples are deemed to be the most useful in learning a good policy [8]. These are some of the advancements that can be used to achieve high performance in playing Arcade Games

Proposed experiment set up and evaluation

This next section highlights how experiments could be set up, as well as how the agents could be evaluated.

Creating learning algorithms that perform well, even in simple Atari games, can be a difficult process. Careful consideration will be given on how the training and testing strategies must be structured, in order to present how game difficulty affects agent performance and more specifically whether or not an agent trained on higher difficulty can outperform an agent trained on easier difficulty, when they are tested on the 'easy' difficulty setting. The following strategies are being considered depending on the nature of the game used, and whether it is a 2-player game. Two games were chosen to illustrate the different training strategy options, as well as evaluation methods.

Game 1. Atari Breakout

Breakout (Figure 3: Atari Breakout) is a game about maximizing your score by moving the paddle at the bottom of the screen. In this simple game, the agent has control over the paddle can move it left and right. The objective is to direct the ball using the paddle to the blocks at the top of the screen. The player is awarded points when the ball hits the blocks. The design strategy for this experiment could be as follows:

For phase 1, two RL algorithms of the same architecture will be trained separately. The first one will be trained on the 'easy' setting of the game, and the second will be trained a harder setting. Both algorithms will be trained until they demonstrate convergence. A threshold score can be set to measure convergence, if the agents can attain the score, then they are thought to have converged. For the second phase the trained algorithms will be tested on the same game, with the difficulty setting set to 'easy' mode. By analyzing the behavior of the second agent we can find out whether it can generalize to the 'easy' setting and ultimately understand whether the training carried out on the harder setting proved to be beneficial. This method is depicted at (Figure 5: Experiment procedure)

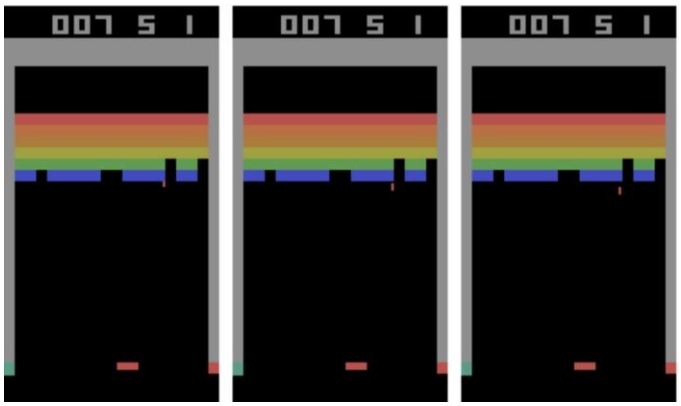
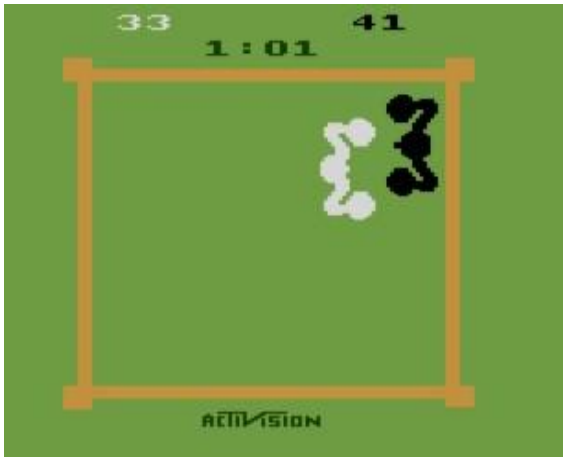


Figure 3: Atari Breakout

Game 2. Atari Boxing

The second game can present a slightly different evaluation methodology as it is a 2-player game. The objective in Boxing (Figure 4: Atari Boxing) is to punch the opponent as many times as possible, while avoiding their punches. The game ends if a player lands 100 punches or the timer has run out. In the game, each successful punch can be interpreted as a reward.



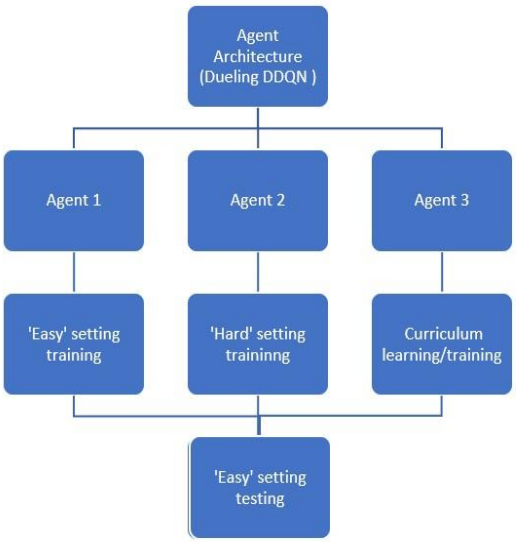


Figure 5: Experiment procedure

If the game completes without a player achieving 100 punches, the player with the highest score is declared the winner. Ties are also possible in this game. The evaluation methodology for a 2-player game of this nature could be as follows. Two agents can be trained in the same manner as the first game. The first agent would be trained on 'easy' difficulty and the second on 'hard' difficulty. As a test the agents would be put in an environment where they had to fight the in-game AI in the easiest difficulty and compare the results. Moreover, the trained agents can be pitted against each other to see which one has leaned a better policy and can generalize in a slightly different setting [6]. The agents would be allowed to play 10 rounds/games against each other and the agent with the most wins would be the winner (Figure 5: Experiment procedure). Finally, an approach that is wholly different can be taken to train an agent, known as curriculum learning [7]. This training approach would be slightly more involved as it would require training the agent on alternated opponents switching between in-game AI that is easy and in-game AI that is hard. The simplest approach that can be taken would be to increase the difficulty slowly and ideally present the agent a hard setting after it has mastered the easy one. This simple approach however has a weakness. As the agent is trained on harder settings, it may forget how to during easy ones. This could manifest as what is known as "catastrophic forgetting". To avoid such a scenario the training strategy might have to present the agent with different opponents, at different stages of the training. If the agent were to be trained on a different opponent after every few games, it could potentially lessen the chances of the agent overfitting and sticking to a policy that is good for a specific opponent, thus, demonstrating better results overall compared to other agents (Figure 6: Curriculum learning example). Overfitting may or may not present a significant challenge and can be tackled in many ways, for example if an agent trained on 'hard' difficulty overfits to that setting, it may exhibit very poor performance when placed on 'easy' difficulty, by changing the training parameters such as decay rate schedule and the architecture of the neural network, the effects of overfitting can be minimized. The report will discuss whether overfitting was encountered, and which steps were taken to minimize its impact. There exist many different schemes to curriculum learning, the goal of this project will be to find a suitable scheme that can offer results comparable to other training strategies

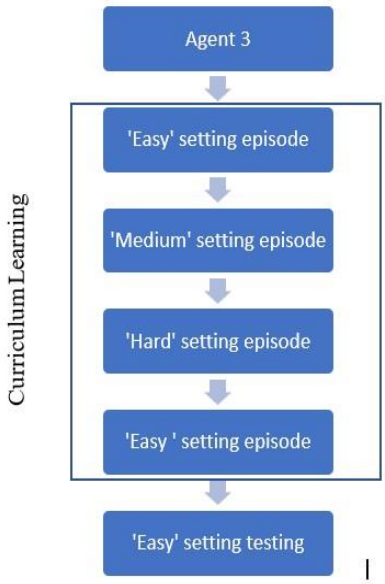


Figure 6: Curriculum learning example

Objectives and Testable Outcomes

This next section explores the objectives in more depth and sets out criteria that must be met for the successfully completing them.

1. It has been established that not all DQN architectures are beneficial for all games. For example, while a Dueling Double Deep Q network might perform extremely well in one game, it may not perform any better or perhaps perform worse than a vanilla DQN on a different game. As such suitable algorithms will be the ones that meet a performance standard across all 3 chosen games. Suitable algorithms will be one that can achieve better results than a vanilla DQN. The testing performance under the easy setting will be the performance metric that will show whether an agent is suitable, as it is possible that an agent can perform not so well during training, but perform very well during testing.

2. The training procedures that will be covered on this project will have to take advantage of the harder setting available for each game. These training strategies can be as simple as only training an agent on hard or easy settings and could be as advanced as employing a curriculum learning strategy. Since curriculum learning has many different training strategies, a strategy that is comparable in terms of results, to the simpler ones, must be found.

3. The results of the project will include two advanced RL algorithms that can perform well in at least 3 different games. This does not include a simple DQN, which will be used only as a baseline and a point of comparison.

4. In order to compare and contrast the algorithms, a detailed ablation study will be structured in the same vein as[8], where depending on the game discussed the agents will be compared on reward metrics or how many rounds they won. These comparisons will determine which agent exhibits optimal behavior in the easy setting. This optimal agent will have a specific DQN architecture and will have followed a specific training strategy. It is important to point out, that a truly 'optimal' policy for any game will not be learned, however the policy that outperforms all other policies will be evident.

5. After the optimal agent has been found, the impact of the training strategy will become clear, along with whether training an agent on hard settings presents an advantage. This of course, as previously stated, may be different depending on the game. For example, a Dueling Double Deep Q network when trained on a hard setting may exhibit better performance on Atari Breakout, but at the same time it may underperform in Atari Boxing, where the best agent might be a network of the same architecture, that was trained on easy settings. If training an agent on high difficulty results in better performance than training the same agent on easy settings, that would mean the high difficulty training setting was beneficial.

Objective	Testable Outcome
Identify suitable Deep RL learning algorithms for Classic Arcade Games	List of RL algorithms that can achieve at least as good performance as a simple DQN

Identify suitable training strategies for the learning algorithms.	List of 3 training strategies that will be used to train the RL algorithms. One of which will employ curriculum learning
Design and develop at least 2 learning algorithms that can exhibit good performance in at least 3 games.	List of RL algorithms that perform well on 3 chosen game. Excluding the vanilla DQN
Compare and contrast RL algorithms in terms of performance and discuss the best results.	Detailed ablation study that provides comparisons broken down by game, training strategy and DQN architecture.

Table 2: Project objectives and testable outcomes

The question to be tackled in this project has no clear answer yet. The outcome of this project is unknown and while it makes intuitive sense that agents which have learned how to behave in a harder setting should outperform the agents that were trained on an easier settings, the outcome remains to be seen. There are several outcome scenarios, the results may in fact go against human intuition and present an outcome where the agent performs badly in the 'easy' setting. This might not be the case for all games though and may not be true for all algorithm architecture, so it is of critical importance that the answer to be given makes those distinctions when they should be made in the form of a detailed ablation study. The reproducibility of these results is also of paramount importance so that can be accepted as reliable, this may be a sticking point that proves to be very difficult to achieve since there is a lot of randomness involved in the kinds of approaches that will be taken (Noisy networks, e-greedy policy, etc..) Multiple runs of the same code should demonstrate trusted results, by using statistics such as mean and standard deviation for reward measurements.

Project Deliverables

The project will offer the following deliverables upon its conclusion:

- A detailed report on the results of the testing carried out. The nature of the report will of course be quantitative, while also including certain remarks and comments about the performance of each agent and training approach.
- The full training and testing code of the agents in the form of .py files. Also, user documentation will be included in how to run the entirety of the code.
- Trained agents in the form of pickle files, that can be tested out of the box, without requiring any further training or tuning.
- A short video presentation summarizing the results achieved and showing agent behavior in the environment.
- A detailed project report that among other things will provide an answer to the central question that the project poses.

The project does not raise any ethical concerns, there will be no data collection from a sample population or any other kind of involvement of individuals. No legal issues or professional issues

should arise from undertaking this project, all third-party contribution will be given appropriate credit when used. An ethics form is provided in the Appendix section of this document.

Project management and work plan

Below the Gantt Chart is given. It presents the several milestones that need to be met for the completion of the project, a detailed timeline is presented.

Gantt Chart	Begin Date	End Date	September	October	November	December
1. Environment setup and configuration	01/09/2020	15/09/2020				
2. Literature Survey research and Background Research						
2.a Algorithm RL survey	01/09/2020	15/09/2020				
2.b Curriculum learning approaches survey	01/09/2020	22/09/2020				
2.c Survey the games environments available and decide on 3 of them	08/09/2020	22/09/2020	*			
3. Small write-up in report						
3.a. Work on report introduction	22/09/2020	29/09/2020				
3.b. Work on report literature review	22/09/2020	29/09/2020				
4. System implementation						
4.a. RL Algorithms implementation - Easy & Hard training strategies	29/09/2020	13/10/2020				
4.b. RL algorithm testings, debugging and fixes	29/09/2020	13/10/2020				
4.c Curriculum learning strategy implementation	13/10/2020	27/10/2020		*		
4.d Work on "Methods used" part of the report	27/10/2020	3/11/2020				
5. System testing and debugging						
5.a. Review all implemented algorithms and implement testing procedures	03/11/2020	17/11/2020			*	
6. Experiments and report write up						
6.a. Gathering of evaluation statistics of agents	17/11/2020	01/12/2020				
6.b. Work on experiment evalaution part of report	17/11/2020	08/12/2020			*	
6.c. Review of results	08/12/2020	15/12/2020				
7. Complete report (Sections such as reflections, future work, etc...)	15/12/2020	21/12/2020				

The project aims to develop several pieces of software. As such, the approach of test-driven development will be followed [11]. Short testing periods will be followed up by quick coding sessions. This method will minimize the occurrence of bugs, and other technical mis happenings. The project will start with the environment set-up of the workstation, at the same time research will be carried out on the types of RL algorithms that can be used, and the different types of curriculum learning. After carrying out the research the beginning sections of the report can be written, more specifically the introduction and literature review (Tasks 3.a, 3.b). A supervisor meeting is proposed close to the end of the month to discuss the progress that has been made on the research [20/09/2020]. The second part of the project starts with the implementation of RL algorithms in the chosen game environments (Tasks 4.a-4.c). The implementation of the algorithms will run in parallel with testing and debugging (this is the approach of test-driven development). During this period, the bulk of the development will take place. The training strategies will be implemented starting with the 'easy' and 'hard' training strategies with curriculum learning following soon after. Overall, 4 weeks are dedicated to developing the core of the software, with one more week added for reviewing the code and implementing the testing procedures (Task 5.a). The second meeting is proposed at the end of the second month, around [20/10/2020] so that the progress in the implementation can be discussed. Moreover, another meeting is proposed during the review of the implementation, around [15/11/2020].

The final part of the project consists of running the experiments several times and gathering statistics (Task 6.a). The most important part of the report will also be written during this period (experiment results and evaluation) (Task 6.b). A meeting is proposed around [8/12/2020], to discuss the report final parts and the project outcomes overall. The project ends with a final review (Task 6.c) and with the completion of the project report (Task 7), and the submission [21/12/2020]

Risks and mitigation strategies.

The project comes with risks that will have to be mitigated as much as possible. The risks below are mostly technically related. They are presented in the form of a table.

Risk Description	Likelihood of Risk	Impact on the Project	Mitigation Action
Getting environment set-up	Medium	High	Start as early as possible on the environment set-up. A virtual machine option will be explored to install a more stable version of AI gym.
Scope creep due to the software coding requirements	High	High	Setting a hard deadline for final software iteration after which the final part of report write-up will take place.
Hardware failures	Low	High	Move experiments to online environments or University premises if the need arises.
Data loss/ Results loss	Low	High	Purchase and use an external SSD as backup. Periodical automatic backups can be put in place.
Implementation of a learning algorithm in pure Pytorch.	Medium	Small	High level RL libraries can be used to implement RL algorithms.

Table 3: Risks and mitigation strategies

Some of the most important and impactful risks have been documented above. The mitigation strategy has also been set out

References

[1]Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540), pp.529-533.

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

[3] 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.

[4] Machado, M.C, Bellemare, M.G., Talvitie, E., Veness, J., Hausknecht, M. and Bowling, M., 2018. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 61, pp.523-562.

[5] Zhang, C., Vinyals, O., Munos, R. and Bengio, S., 2018. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*.

[6] Bhonker N., Rozenberg S., Hubara I., 2016. Playing SNES in the Retro Learning Environment. *ARXIV* arXiv e-prints, *arXiv:1611.02205*

[7] Narvekar S., Stone P., 2019. Learning Curriculum Policies for Reinforcement Learning, *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems* (pp. 25-33). International Foundation for Autonomous Agents and Multiagent Systems

[8] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2018, April. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[9] Silver D., Antonoglou I., Schaul T., Quan T., 2016. Prioritized Experience Replay, *ICLR ARXIV* arXiv e-prints, *arXiv:1511.05952*

[10] Gym.openai.com. 2020. Gym: A Toolkit For Developing And Comparing Reinforcement Learning Algorithms. *Gym.openai*. [online] Available from: https://gym.openai.com/docs/ [Accessed 5 May 2020].

[11] Martin, R.C., (2011). *The clean coder: a code of conduct for professional programmers*. Pearson Education.

Ethics Checklist.

A.1 If you answer YES to any of the questions in this block, you must apply to an appropriate external ethics committee for approval and log this approval as an External Application through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
1.1	Does your research require approval from the National Research Ethics Service (NRES)? *e.g. because you are recruiting current NHS patients or staff?* *If you are unsure try - https://www.hra.nhs.uk/approvals-amendments/what-approvals-do-i-need/*	**NO**
1.2	Will you recruit participants who fall under the auspices of the Mental Capacity Act? *Such research needs to be approved by an external ethics committee such as NRES or the Social Care Research Ethics Committee - http://www.scie.org.uk/research/ethics-committee/*	**NO**

1.3	Will you recruit any participants who are currently under the auspices of the Criminal Justice System, for example, but not limited to, people on remand, prisoners and those on probation? *Such research needs to be authorised by the ethics approval system of the National Offender Management Service.*	**NO**
A.2 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee, you must apply for approval from the Senate Research Ethics Committee (SREC) through Research Ethics Online - https://ethics.city.ac.uk/		*Delete as appropriate*
2.1	Does your research involve participants who are unable to give informed consent? *For example, but not limited to, people who may have a degree of learning disability or mental health problem, that means they are unable to make an informed decision on their own behalf.*	**NO**
2.2	Is there a risk that your research might lead to disclosures from participants concerning their involvement in illegal activities?	**NO**
2.3	Is there a risk that obscene and or illegal material may need to be accessed for your research study (including online content and other material)?	**NO**
2.4	Does your project involve participants disclosing information about special category or sensitive subjects? *For example, but not limited to: racial or ethnic origin; political opinions; religious beliefs; trade union membership; physical or mental health; sexual life; criminal offences and proceedings*	**NO**
2.5	Does your research involve you travelling to another country outside of the UK, where the Foreign & Commonwealth Office has issued a travel warning that affects the area in which you will study? *Please check the latest guidance from the FCO - http://www.fco.gov.uk/en/*	**NO**
2.6	Does your research involve invasive or intrusive procedures? *These may include, but are not limited to, electrical stimulation, heat, cold or bruising.*	**NO**
2.7	Does your research involve animals?	**NO**
2.8	Does your research involve the administration of drugs, placebos or other substances to study participants?	**NO**

A.3 If you answer YES to any of the questions in this block, then unless you are applying to an external ethics committee or the SREC, you must apply for approval from the Computer Science Research Ethics Committee (CSREC) through Research Ethics Online - https://ethics.city.ac.uk/ **Depending on the level of risk associated with your application, it may be referred to the Senate Research Ethics Committee.**		*Delete as appropriate*
3.1	Does your research involve participants who are under the age of 18?	**NO**
3.2	Does your research involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)? *This includes adults with cognitive and / or learning disabilities, adults with physical disabilities and older people.*	**NO**
3.3	Are participants recruited because they are staff or students of City, University of London? *For example, students studying on a particular course or module.* *If yes, then approval is also required from the Head of Department or Programme Director.*	**NO**
3.4	Does your research involve intentional deception of participants?	**NO**
3.5	Does your research involve participants taking part without their informed consent?	**NO**
3.5	Is the risk posed to participants greater than that in normal working life?	**NO**
3.7	Is the risk posed to you, the researcher(s), greater than that in normal working life?	**NO**
A.4 If you answer YES to the following question and your answers to all other questions in sections A1, A2 and A3 are NO, then your project is deemed to be of MINIMAL RISK.		*Delete as appropriate*
If this is the case, then you can apply for approval through your supervisor under PROPORTIONATE REVIEW. You do so by completing PART B of this form. **If you have answered NO to all questions on this form, then your project does not require ethical approval. You should submit and retain this form as evidence of this.**		
4	Does your project involve human participants or their identifiable personal data? *For example, as interviewees, respondents to a survey or participants in testing.*	**NO**

Appendix D Coding Files

The training routine for 'Pong', for both easy and hard settings.

The difficulty variable is changed for changing the environment difficulty.

```
from pong_skeleton_libraries import skeleton_helper_functions
from pong_skeleton_libraries import pong_env_wrappers as env_wrappers
from pong_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter


RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False


agent_type = 'DuelingDoubleDQN'
#print(torch.__version__)
print(agent_type)

"""
ENVIRONMENT AND DIFFICULTY SETTINGS
"""
ENV_NAME = "PongNoFrameskip-v4"
difficulty = 1    # 0,1 for Pong

print ('Difficulty SELECTED!!!!!!  ', difficulty)


env = env_wrappers.make_train_env(ENV_NAME, difficulty, save_every_x_games = 1
0, name_of_game = ENV_NAME,
                                  net_type = agent_type, stacked_frames = 2)
# What game is being played?
print(env.env.game)
# What are the available difficulties?
print(env.ale.getAvailableDifficulties())
```

```
"""
Setting the training parameters.
"""
# Train on CUDA IF POSSIBLE
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Training on: ' + str(device))



############################
####################
NETS_SYNCH_TARGET = 1_000
REPLAY_BUFFER_START_SIZE_FOR_LEARNING = 10_000

LEARNING_RATE = 1e-4
REPLAY_BUFFER_SIZE = 100_000

EPSILON_DECAY_LAST_FRAME = 100_000
EPSILON_START = 1.0
EPSILON_FINAL = 0.01

GAMMA_FACTOR = 0.99
BATCH_SIZE = 32

MAXIMUM_STEPS_ALLOWED = 500_000
TARGET_REWARD = 18


########################################
###################################

print(env.observation_space.shape)


if agent_type == "DQN" or agent_type == "DoubleDQN" :

      net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_space.sh
ape,env.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_s
pace.shape,env.action_space.n).to(device)

elif agent_type == "DuelingDoubleDQN" :

      net = skeleton_dqn_architectures.Dueling_CNN_DQN(env.observation_space.s
hape,env.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.Dueling_CNN_DQN(env.observation_
space.shape,env.action_space.n).to(device)
```

```
# PERFORMANCE TRACKER IN TENSORBOARD
writer = SummaryWriter(log_dir = "runs/Paper runs/" + agent_type +" -
Diff " + str(difficulty) + " " + ENV_NAME)
print("DQN ARCHITECTURE \n", net)

buffer = skeleton_helper_functions.ReplayMemory(REPLAY_BUFFER_SIZE)
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)



'''
TRAINING LOOP !!!!
BELOW !!!!
'''
epsilon = EPSILON_START
state = env.reset()
episode_reward = 0.0
episode_rewards = []
steps_count = 0
max_mean_reward = None


while True:

  steps_count += 1

  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env,  epsilon,
                                                                     buffer, d
evice = device)
  episode_reward += step_reward


  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))
```

```
      if max_mean_reward is None or max_mean_reward < mean_reward_100:
          torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_dif
f_'+str(difficulty)+
                     "-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100


      # the episode is finished so reset the env and the episode reward
      episode_reward = 0.0
      state = env.reset()



  """
  Load the buffer first!!!!! before synchning or back-
propagating and learning on data
  """

  if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
      if steps_count % 1000 == 0:
            print("Populating buffer")
      continue



  if steps_count == MAXIMUM_STEPS_ALLOWED:
        print("Maximum steps allowed reached, training finished.")
        break

  if max_mean_reward == TARGET_REWARD:
        print('Desired reward achieved')
        break


  if steps_count % NETS_SYNCH_TARGET == 0:
      print('Nets synched')
      target_net.load_state_dict(net.state_dict())


  """
  Gradient Descent below

  The simple DQN uses the simple_dqn() TD-LOSS FUNCTION

  The DuelingDouble uses the double_dqn_loss() TD-LOSS FUNCTION
```

```
  """
  optimizer.zero_grad()

  batch = buffer.sample(BATCH_SIZE)

  batch = buffer.to_pytorch_tensors(batch,device)

  batch_loss = None


  if agent_type == "DQN":

        batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                        gamma = GAMMA_FACTOR,
                                                        batch_size = BATCH_SIZE,

                                                        device = device)


  elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

        batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                        gamma = GAMMA_FACTOR,
                                                        batch_size = BATCH_SIZE,

                                                        device = device)
  # Calculate gradients for the weights
  batch_loss.backward()

  # Optimize the weights
  optimizer.step()
```

The curriculum training routine for 'Pong'

```
from pong_skeleton_libraries import skeleton_helper_functions
from pong_skeleton_libraries import pong_env_wrappers as env_wrappers
from pong_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import torch
import torch.nn as nn
```

```
import torch.optim as optim
from tensorboardX import SummaryWriter


RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False


agent_type = 'DuelingDoubleDQN'
print(torch.__version__)

"""
ENVIRONMENT AND DIFFICULTY SETTINGS
"""
ENV_NAME = "PongNoFrameskip-v4"


env_easy  = env_wrappers.make_train_env_curriculum(ENV_NAME, 1, save_every_x_g
ames = 10,
                                                   name_of_game = ENV_NAME, ne
t_type = agent_type,
                                                   stacked_frames = 2)

env_hard  = env_wrappers.make_train_env_curriculum(ENV_NAME, 0, save_every_x_g
ames = 10,
                                                   name_of_game = ENV_NAME, ne
t_type = agent_type,
                                                   stacked_frames = 2)

env_easy_2  = env_wrappers.make_train_env_curriculum(ENV_NAME, 1, save_every_x
_games = 10,
                                                   name_of_game = ENV_NAME, ne
t_type = agent_type,
                                                   stacked_frames = 2)

# What game is being played?
print(env_easy.env.game)
# What are the available difficulties?
print(env_easy.ale.getAvailableDifficulties())


"""
Setting the training parameters.
"""
# Train on CUDA IF POSSIBLE
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Training on: ' + str(device))

##############################
##########################
####################
NETS_SYNCH_TARGET = 1_000
REPLAY_BUFFER_START_SIZE_FOR_LEARNING = 10_000
LEARNING_RATE = 1e-4
REPLAY_BUFFER_SIZE = 100_000

EPSILON_START = 1.0
EPSILON_FINAL = 0.01
GAMMA_FACTOR = 0.99
BATCH_SIZE = 32

# Must be expnaded to exploret the 'Hard' environment as well.
EPSILON_DECAY_LAST_FRAME = 300_000 # must allow for exploration in both 'easy'
 and 'hard' settings.

#Must add up to 500k
EASY_STEPS = 200_000
HARDER_STEPS = 200_000
FINAL_EASY_STEPS = 100_000

########################################
####################################
print(env_easy.observation_space.shape)


if agent_type == "DQN" or agent_type == "DoubleDQN" :

      net = skeleton_dqn_architectures.NATURE_CNN_DQN(env_easy.observation_spa
ce.shape,env_easy.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.NATURE_CNN_DQN(env_easy.observat
ion_space.shape,env_easy.action_space.n).to(device)

elif agent_type == "DuelingDoubleDQN" :

      net = skeleton_dqn_architectures.Dueling_CNN_DQN(env_easy.observation_sp
ace.shape,env_easy.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.Dueling_CNN_DQN(env_easy.observa
tion_space.shape,env_easy.action_space.n).to(device)


# PERFORMANCE TRACKER IN TENSORBOARD
writer = SummaryWriter(log_dir = "runs/Paper runs/" + agent_type +" -
Diff " + "Curriculum_V2" + " " + ENV_NAME)
print("DQN ARCHITECTURE \n", net)
```

```
buffer = skeleton_helper_functions.ReplayMemory(REPLAY_BUFFER_SIZE)
epsilon = EPSILON_START
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)


'''
TRAINING LOOP !!!!

V2
1) 200K STEPS ON EASY
2) 200K STEPS ON HARD
3) 100K STEPS ON EASY
'''


episode_reward = 0.0
episode_rewards = []
steps_count = 0
max_mean_reward = None



state = env_easy.reset()
for i  in range(EASY_STEPS):

  steps_count += 1
  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env_easy,  epsilon, buffer, device = device)
  episode_reward += step_reward

  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
          torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_dif
f_'+'curriculumV2'+"-best_%.0f.dat" % mean_reward_100)
```

```
            if max_mean_reward is not None:
                print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
            max_mean_reward = mean_reward_100


        # the episode is finished so reset the env and the episode reward
        episode_reward = 0.0
        state = env_easy.reset()



    """
    Load the buffer first!!!!! before synchning or back-
propagating and learning on data
    """

    if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
        print("Populating buffer")
        continue


    if steps_count % NETS_SYNCH_TARGET == 0:
        print('Nets synched')
        target_net.load_state_dict(net.state_dict())



    optimizer.zero_grad()
    batch = buffer.sample(BATCH_SIZE)
    batch = buffer.to_pytorch_tensors(batch,device)
    batch_loss = None

    if agent_type == "DQN":

           batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                   gamma = GAMMA_FACTOR,
                                                   batch_size = BATCH_SIZE,

                                                   device = device)

    elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

           batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                   gamma = GAMMA_FACTOR,
```

```
                                                    batch_size = BATCH_SIZE,

                                                    device = device)
  batch_loss.backward()

  optimizer.step()




print('HARD TRAINING STARTED')
episode_reward = 0.0 # episode reward reset to 0

state = env_hard.reset()
for i  in range(HARDER_STEPS):

  steps_count += 1

  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env_hard,  epsilon, buffer, device = device)
  episode_reward += step_reward


  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
          torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_dif
f_'+'curriculum'+"-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100


      # the episode is finished so reset the env and the episode reward
      episode_reward = 0.0
```

```
      state = env_hard.reset()



  """
  Load the buffer first!!!!! before synchning or back-
propagating and learning on data
  """

  if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
      print("Populating buffer")
      continue


  if steps_count % NETS_SYNCH_TARGET == 0:
      print('Nets synched')
      target_net.load_state_dict(net.state_dict())



  optimizer.zero_grad()

  batch = buffer.sample(BATCH_SIZE)

  batch = buffer.to_pytorch_tensors(batch,device)


  batch_loss = None


  if agent_type == "DQN":

        batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                    gamma = GAMMA_FACTOR,
                                                    batch_size = BATCH_SIZE,

                                                    device = device)


  elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

        batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                    gamma = GAMMA_FACTOR,
                                                    batch_size = BATCH_SIZE,

                                                    device = device)
  batch_loss.backward()
```

```
  optimizer.step()



print('FINAL EASY TRAINING STARTED')
episode_reward = 0.0 # episode reward reset to 0

state = env_easy_2.reset()

for i  in range(FINAL_EASY_STEPS):

  steps_count += 1

  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env_easy_2,  epsilon, buffer, device = device)
  episode_reward += step_reward


  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
          torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_dif
f_'+'curriculum'+"-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100


      # the episode is finished so reset the env and the episode reward
      episode_reward = 0.0
      state = env_easy_2.reset()
```

```
  """
  Load the buffer first!!!!! before synchning or back-
propagating and learning on data
  """

  if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
      print("Populating buffer")
      continue


  if steps_count % NETS_SYNCH_TARGET == 0:
      print('Nets synched')
      target_net.load_state_dict(net.state_dict())



  optimizer.zero_grad()

  batch = buffer.sample(BATCH_SIZE)

  batch = buffer.to_pytorch_tensors(batch,device)


  batch_loss = None


  if agent_type == "DQN":

        batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                    gamma = GAMMA_FACTOR,
                                                    batch_size = BATCH_SIZE,

                                                    device = device)


  elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

        batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                    gamma = GAMMA_FACTOR,
                                                    batch_size = BATCH_SIZE,

                                                    device = device)
  batch_loss.backward()

  optimizer.step()
```

```
print('Training finished')
```

Random agent run script for 'Pong'

```
from pong_skeleton_libraries import skeleton_helper_functions
from pong_skeleton_libraries import pong_env_wrappers as env_wrappers
from pong_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import time # just to have timestampsAgent in the file
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter


agent_type = 'RANDOM' # this changes across files
print(torch.__version__)

"""
ENVIRONMENT AND DIFFICULTY SETTINGS
"""
ENV_NAME = "PongNoFrameskip-v4"
difficulty = 1    # 1, 2, 4 for boxing

print ('Difficulty SELECTED!!!!!!  ', difficulty)


env = env_wrappers.make_train_env(ENV_NAME, difficulty, save_every_x_games = 1
_000, name_of_game = ENV_NAME,
                                  net_type = agent_type, stacked_frames = 2)

# What game is being played?
print(env.env.game)
# What are the available difficulties?
print(env.ale.getAvailableDifficulties())



"""
Setting the training parameters.
"""
# Train on CUDA IF POSSIBLE
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Training on: ' + str(device))


##################################
```

```
MAXIMUM_STEPS_ALLOWED = 500_000


########################################
#####################################
#################################

# SELECT THE DESIRED VERSION OF DQN
net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_space.shape,en
v.action_space.n).to(device)
target_net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_space.s
hape,env.action_space.n).to(device)

# PERFORMANCE TRACKER IN TENSORBOARD
writer = SummaryWriter(log_dir = "runs/Paper runs/" + agent_type +" -
Diff " + str(difficulty) + " " + ENV_NAME)



state = env.reset()
episode_reward = 0.0
episode_rewards = []
steps_count = 0


buffer = None

while True:

  steps_count += 1

  epsilon = 1.0
  action = skeleton_helper_functions.select_action(epsilon,state,env,None,None
)

  state, reward, is_done, _ =  env.step(action)

  episode_reward += reward


  if is_done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])

      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
```

```
                                                    episode_reward,
                                                    len(episode_rewards))

      episode_reward = 0.0
      state  = env.reset()


  if steps_count == MAXIMUM_STEPS_ALLOWED:
        print("Maximum steps allowed reached, training finished.")
        break
```

Testing script of DQN and Double DQN architecture for 'Pong'

```
from pong_skeleton_libraries import skeleton_helper_functions
from pong_skeleton_libraries import pong_env_wrappers as env_wrappers
from pong_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter

import time


FPS = 800  # frames per second count, it is tied to game logic and physics ???
?
device = 'cuda'

env_name  = 'PongNoFrameskip-v4'
difficulty = 1 # Variable and easy to change!!!! {1-> 'Easy', 0->'Hard'}
agent = 'PongNoFrameskip-v4_diff_curriculum-
best_7.dat'  # variable and easy to change


#Models to test:

#DQNs:
# PongNoFrameskip-v4_diff_0-best_18
# PongNoFrameskip-v4_diff_1-best_16
# PongNoFrameskip-v4_diff_curriculum-best_-2


#Double DQNs
# PongNoFrameskip-v4_diff_0-best_17
```

```
# PongNoFrameskip-v4_diff_1-best_17
# PongNoFrameskip-v4_diff_curriculum-best_7



save_10_episodes = 10
env = env_wrappers.make_test_env(env_name, difficulty, save_10_episodes, name_
of_game ='PONG', net_type = 'DQN_'+agent, stacked_frames=2)

net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_space.shape,en
v.action_space.n).to(device)
state = torch.load('models/DoubleDQN/'+agent, map_location=lambda stg, _: stg)
   #Source directory also changes for each agent type
net.load_state_dict(state)


writer = SummaryWriter(log_dir = "runs/Paper runs/Test runs/" +'DoubleDQN_ '+
agent +" -Diff " + str(difficulty))

num_of_eps = 50
episode_reward = 0.0 # Total reward for an episode
episode_rewards = []


e = 0.02
steps_count = 0
episode_steps = 0
for i in range(num_of_eps):
    print('Episode :', i)
    state = env.reset()

    while (True):
            episode_steps += 1
            steps_count +=1

            #Uncomment to render environment
            #env.render()

            if np.random.random() >= e:
                  state = np.array([state], copy=False)
                  state_tensor = torch.tensor(state).to(device)

                  action_v = net(state_tensor).max(1)[1].view(1, 1) #get actio
n with the highest value
                  action_index = int(action_v.item())

            else:
```

```
                action_index = env.action_space.sample() #random action to t
ake


            state, step_reward, done, info = env.step(action_index)
            episode_reward += step_reward

            if done:
                print('Steps taken: ',episode_steps)
                episode_rewards.append(episode_reward)

                #LOG PERFORMANCE with tensorboard
                writer.add_scalar("Rewards", episode_reward, len(episode_rewar
ds)-1)
                writer.add_scalar("Mean Reward", np.mean(episode_rewards[-
10:]),steps_count)
                writer.add_scalar('Cummmulative reward',np.array(episode_rewar
ds).sum(),len(episode_rewards)-1)
                writer.add_scalar('Steps per episode',episode_steps, len(episo
de_rewards))

                print(episode_reward)
                print('\n')

                episode_reward = 0.0
                episode_steps = 0
                break


print(episode_rewards)
print('Cummulative reward: ',np.array(episode_rewards).sum())
```

Testing script for DuelingDoubleDQN architecture for ‘Pong’.

```
from pong_skeleton_libraries import skeleton_helper_functions
from pong_skeleton_libraries import pong_env_wrappers as env_wrappers
from pong_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter

import time
```

```
FPS = 800
device = 'cuda'


env_name  = 'PongNoFrameskip-v4'
difficulty = 1 # Variable and easy to change!!!! {1-> 'Easy', 0->'Hard'}
agent = 'PongNoFrameskip-v4_diff_curriculum-
best_2.dat'  # variable and easy to change


#DuelingDoubleDQNs
#PongNoFrameskip-v4_diff_0-best_18
#PongNoFrameskip-v4_diff_1-best_17
#PongNoFrameskip-v4_diff_curriculum-best_2


save_10_episodes = 10
env = env_wrappers.make_test_env(env_name, difficulty, save_10_episodes, name_
of_game ='PONG', net_type = 'DQN_'+agent, stacked_frames=2)
net = skeleton_dqn_architectures.Dueling_CNN_DQN(env.observation_space.shape,e
nv.action_space.n).to(device)
state = torch.load('models/DuelingDoubleDQN/'+agent, map_location=lambda stg,
_: stg)   #Source directory also changes for each agent type
net.load_state_dict(state)


writer = SummaryWriter(log_dir = "runs/Paper runs/Test runs/" +'DuelingDoubleD
QN_ '+ agent +" -Diff " + str(difficulty))
num_of_eps = 50
episode_reward = 0.0 # Total reward for an episode
episode_rewards = []


e = 0.02
steps_count = 0
episode_steps = 0

for i in range(num_of_eps):
    print('Episode :', i)
    state = env.reset()

    while (True):
            episode_steps += 1
            steps_count +=1

            #Uncomment to render environment
            #env.render()
```

```
            if np.random.random() >= e:
                  state = np.array([state], copy=False)
                  state_tensor = torch.tensor(state).to(device)
                  action_v = net(state_tensor).max(1)[1].view(1, 1)
                  action_index = int(action_v.item())

            else:

                  action_index = env.action_space.sample()


            state, step_reward, done, info = env.step(action_index)
            episode_reward += step_reward

            if done:
                print('Steps taken: ',episode_steps)
                episode_rewards.append(episode_reward)
                #LOG PERFORMANCE with tensorboard
                writer.add_scalar("Rewards", episode_reward, len(episode_rewar
ds)-1)
                writer.add_scalar("Mean Reward", np.mean(episode_rewards[-
10:]),steps_count)
                writer.add_scalar('Cummmulative reward',np.array(episode_rewar
ds).sum(),len(episode_rewards)-1)
                writer.add_scalar('Steps per episode',episode_steps, len(episo
de_rewards))

                print(episode_reward)
                print('\n')

                episode_reward = 0.0
                episode_steps = 0
                break


print(episode_rewards)
print('Cummulative reward: ',np.array(episode_rewards).sum())
```

Environment wrappers for 'Pong'

The code for this file is adapted from the OpenAI baselines repository.

Code can be found at :
https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py

```
"""
These wrappers make the ALE work as intended.
These specific wrappers work with Pong....

Adapted from the AI baselines Repo:
https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py
"""

import numpy as np
from collections import deque
import gym
from gym import spaces
import cv2
import gym
cv2.ocl.setUseOpenCL(False)

RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)



class NoopResetEnv(gym.Wrapper):
    def __init__(self, env, noop_max=30):
        """Sample initial states by taking random number of no-ops on reset.
        No-op is assumed to be action 0.
        """
        gym.Wrapper.__init__(self, env)
        self.noop_max = noop_max
        self.override_num_noops = None
        self.noop_action = 0
        assert env.unwrapped.get_action_meanings()[0] == 'NOOP'

    def reset(self, **kwargs):
        """ Do no-op action for a number of steps in [1, noop_max]."""
        self.env.reset(**kwargs)
        if self.override_num_noops is not None:
            noops = self.override_num_noops
        else:
            noops = self.unwrapped.np_random.randint(1, self.noop_max + 1)  #
pylint: disable=E1101
        assert noops > 0
        obs = None
        for _ in range(noops):
            obs, _, done, _ = self.env.step(self.noop_action)
            if done:
                obs = self.env.reset(**kwargs)
        return obs
```

```
    def step(self, ac):
        return self.env.step(ac)


class FireResetEnv(gym.Wrapper):
    def __init__(self, env):
        """Take action on reset for environments that are fixed until firing."
""
        gym.Wrapper.__init__(self, env)
        assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
        assert len(env.unwrapped.get_action_meanings()) >= 3

    def reset(self, **kwargs):
        self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(1)
        if done:
            self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(2)
        if done:
            self.env.reset(**kwargs)
        return obs

    def step(self, ac):
        return self.env.step(ac)


class EpisodicLifeEnv(gym.Wrapper):
    def __init__(self, env):
        """Make end-of-life == end-of-
episode, but only reset on true game over.
        Done by DeepMind for the DQN and co. since it helps value estimation.
        """
        gym.Wrapper.__init__(self, env)
        self.lives = 0
        self.was_real_done = True

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        self.was_real_done = done
        # check current lives, make loss of life terminal,
        # then update lives to handle bonus lives
        lives = self.env.unwrapped.ale.lives()
        if lives < self.lives and lives > 0:
            # for Qbert sometimes we stay in lives == 0 condtion for a few fra
mes
            # so its important to keep lives > 0, so that we only reset once
            # the environment advertises done.
            done = True
        self.lives = lives
```

```
        return obs, reward, done, info

    def reset(self, **kwargs):
        """Reset only when lives are exhausted.
        This way all states are still reachable even though lives are episodic
,
        and the learner need not know about any of this behind-the-scenes.
        """
        if self.was_real_done:
            obs = self.env.reset(**kwargs)
        else:
            # no-op step to advance from terminal/lost life state
            obs, _, _, _ = self.env.step(0)
        self.lives = self.env.unwrapped.ale.lives()
        return obs


class MaxAndSkipEnv(gym.Wrapper):
    def __init__(self, env, skip=4):
        """Return only every `skip`-th frame"""
        gym.Wrapper.__init__(self, env)
        # most recent raw observations (for max pooling across time steps)
        self._obs_buffer = np.zeros((2,) + env.observation_space.shape, dtype=
np.uint8)
        self._skip = skip

    def reset(self):
        return self.env.reset()

    def step(self, action):
        """Repeat action, sum reward, and max over last observations."""
        total_reward = 0.0
        done = None
        for i in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            if i == self._skip - 2: self._obs_buffer[0] = obs
            if i == self._skip - 1: self._obs_buffer[1] = obs
            total_reward += reward
            if done:
                break
        # Note that the observation on the done=True frame
        # doesn't matter
        max_frame = self._obs_buffer.max(axis=0)

        return max_frame, total_reward, done, info

    def reset(self, **kwargs):
        return self.env.reset(**kwargs)
```

```
#Reward Wrapper
class ClipRewardEnv(gym.RewardWrapper):
    def __init__(self, env):
        gym.RewardWrapper.__init__(self, env)

    def reward(self, reward):
        """Bin reward to {+1, 0, -1} by its sign."""
        return np.sign(reward)

#Observation Wrapper
class WarpFrame(gym.ObservationWrapper):
    def __init__(self, env):
        """Warp frames to 84x84 as done in the Nature paper and later work.
        Expects inputs to be of shape height x width x num_channels
        """
        gym.ObservationWrapper.__init__(self, env)
        self.width = 84
        self.height = 84
        self.observation_space = spaces.Box(low=0, high=255,
                                            shape=(self.height, self.width, 1)
, dtype=np.uint8)

    def observation(self, frame):
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        frame = cv2.resize(frame, (self.width, self.height), interpolation=cv2
.INTER_AREA)
        return frame[:, :, None]


class FrameStack(gym.Wrapper):
    def __init__(self, env, k):
        """Stack k last frames.
        Returns lazy array, which is much more memory efficient.
        Expects inputs to be of shape num_channels x height x width.
        """
        gym.Wrapper.__init__(self, env)
        self.k = k
        self.frames = deque([], maxlen=k)
        shp = env.observation_space.shape
        self.observation_space = spaces.Box(low=0, high=255, shape=(shp[0] * k
, shp[1], shp[2]), dtype=np.uint8)

    def reset(self):
        ob = self.env.reset()
        for _ in range(self.k):
            self.frames.append(ob)
        return self._get_ob()

    def step(self, action):
```

```
        ob, reward, done, info = self.env.step(action)
        self.frames.append(ob)
        return self._get_ob(), reward, done, info

    def _get_ob(self):
        assert len(self.frames) == self.k
        return LazyFrames(list(self.frames))


class LazyFrames(object):
    def __init__(self, frames):
        """This object ensures that common frames between the observations are
 only stored once.
        It exists purely to optimize memory usage which can be huge for DQN's
1M frames replay
        buffers."""
        self._frames = frames

    def __array__(self, dtype=None):
        out = np.concatenate(self._frames, axis=0)
        if dtype is not None:
            out = out.astype(dtype)
        return out

    def __len__(self):
        return len(self._frames)

    def __getitem__(self, i):
        return self._frames[i]

#Observation Wrapper
class PyTorchFrame(gym.ObservationWrapper):
    """Image shape to num_channels x height x width"""

    def __init__(self, env):
        super(PyTorchFrame, self).__init__(env)
        shape = self.observation_space.shape
        #print("Using PytorchFrame wrapper")
        self.observation_space = gym.spaces.Box(low=0.0, high=1.0, shape=(shap
e[-1], shape[0], shape[1]), dtype=np.uint8)

    def observation(self, observation):
        return np.rollaxis(observation, 2)


# Function that creates an Environment with the selected Wrappers....

def make_train_env(env_name, difficulty, save_every_x_games = 20, name_of_game
 = 'NAME THE GAME!!!', net_type ='DQN', stacked_frames = 2):
```

```
    print('OPENAI GYM Version -->',gym.__version__)

    env = gym.make(env_name, difficulty = difficulty)

    env = NoopResetEnv(env)

    env = MaxAndSkipEnv(env)

    env = FireResetEnv(env)

    env = WarpFrame(env)

    env = PyTorchFrame(env) # Converts the frames into Pytorch format by simpl
y reshaping the Matrix always needed THE ORIGINAL TF VERSION IS WxHxCh to Pyto
rch which is CHNANELxWidthxHeight

    env = FrameStack(env, stacked_frames)   # This is always needed as it defi
nes the MDP nature of the problem

    print('Number of stacked frames: ', stacked_frames)

    #Video saving functionality, save the performace of the agent every x trai
ning games
    env = gym.wrappers.Monitor(env, './videos/training-
videos/'+ name_of_game +'- Diff='+ str(difficulty)+' - NN-'+ net_type,
                                    video_callable=lambda episode_id: episode
_id % save_every_x_games == 0,
                                    force=True)

    env.seed(RANDOM_SEED)
    env.action_space.seed(RANDOM_SEED)


    return env




def make_test_env(env_name, difficulty, save_every_x_games = 20, name_of_game
= 'NAME THE GAME!!!', net_type ='DQN', stacked_frames = 2):

    print('OPENAI GYM Version -->',gym.__version__)

    env = gym.make(env_name, difficulty = difficulty)

    env = NoopResetEnv(env)

    env = MaxAndSkipEnv(env)
```

```
    env = FireResetEnv(env)

    env = WarpFrame(env)

    env = PyTorchFrame(env)

    env = FrameStack(env, stacked_frames)

    print('Stacking frames!!!!!!!!!!!!!!: ', stacked_frames)

    #Video saving functionality, save the performace of the agent every x trai
ning games
    env = gym.wrappers.Monitor(env, './videos/testing-
videos/'+ name_of_game +'- Diff='+ str(difficulty)+' - NN-'+ net_type,
                                       video_callable=lambda episode_id: epis
ode_id % save_every_x_games == 0, force=True)

    #env.seed(RANDOM_SEED)
    #env.action_space.seed(RANDOM_SEED)


    return env




def make_train_env_curriculum(env_name, difficulty, save_every_x_games = 20, n
ame_of_game = 'NAME THE GAME!!!', net_type ='DQN', stacked_frames = 2):

    print('OPENAI GYM Version -->',gym.__version__)

    env = gym.make(env_name, difficulty = difficulty)

    """
    NoopReset might be contributing to better performance!!!!..... this is sti
ll ambiguous!!!!
    """
    env = NoopResetEnv(env)

    env = MaxAndSkipEnv(env)

    env = FireResetEnv(env)

    env = WarpFrame(env)

    env = PyTorchFrame(env)
```

```
    env = FrameStack(env, stacked_frames)

    print('Number of stacked frames: ', stacked_frames)

    #Video saving functionality, save the performace of the agent every x trai
ning games
    env = gym.wrappers.Monitor(env, './videos/training-
videos/'+ name_of_game +'- Diff='+ "curriculum"+' - NN-'+ net_type,
                                    video_callable=lambda episode_id: episode
_id % save_every_x_games == 0,
                                    force=True)

    env.seed(RANDOM_SEED)
    env.action_space.seed(RANDOM_SEED)


    return env
```

DQN architectures and calculating the Q loss function.

The `simple_dqn_loss()` function is adapted from the official Pytorch guide example of DQN.

The code can be found at:
https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

```
import numpy as np
from gym import wrappers
from gym import envs
#print(envs.registry.all())
import torch
import torch.nn as nn
import numpy as np
import torch.nn.functional as F


RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False


"""
```

```
Architecture used in the Nature paper: "Human-
level control through deep reinforcement learning".
"""

class  NATURE_CNN_DQN(nn.Module):

    def __init__(self, input_dimensions, n_actions):
        # super constructor?
        super(NATURE_CNN_DQN, self). __init__()

        self.fully_connected_input = 7*7*64 #self.get_convolution_output(input
_dimensions)

        self.conv_trans1 = nn.Conv2d(input_dimensions[0], 32, kernel_size = 8,
 stride = 4)

        self.conv_trans2  = nn.Conv2d(32, 64, kernel_size = 4, stride = 2)

        self.conv_trans3 =  nn.Conv2d(64, 64, kernel_size = 3, stride = 1)

        self.fully_connected_layers = nn.Sequential(
            nn.Linear(self.fully_connected_input, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )

    def forward(self, x):

        x = x.float()

        x = F.relu(self.conv_trans1(x))
        x = F.relu(self.conv_trans2(x))
        x = F.relu(self.conv_trans3(x))

        # flatten vector for the sequential part of the NN
        x = x.view(x.size()[0], -1)

        result = self.fully_connected_layers(x)

        return result



"""
Dueling DQN architecture as introduced in
"Dueling Network Architectures for Deep Reinforcement Learning"
"""

class Dueling_CNN_DQN(nn.Module):
```

```
    def __init__(self,input_dimensions,n_actions):

        super(Dueling_CNN_DQN, self).__init__()

        self.fully_connected_input = 7*7*64

        self.conv_trans1 = nn.Conv2d(input_dimensions[0], 32, kernel_size = 8,
 stride = 4)

        self.conv_trans2  = nn.Conv2d(32, 64, kernel_size = 4, stride = 2)

        self.conv_trans3 =  nn.Conv2d(64, 64, kernel_size = 3, stride = 1)

        self.state_value = nn.Sequential(
            # FCNN PART
            nn.Linear(self.fully_connected_input, 256),
            nn.ReLU(),
            nn.Linear(256, 1) #Value of the state
        )

        self.action_advantages = nn.Sequential(
            # FCNN PART
            nn.Linear(self.fully_connected_input, 256),
            nn.ReLU(),
            nn.Linear(256, n_actions) #Value of the actions
        )




    def forward(self, x):
        #define path through the network.
        x = x.float()

        # First perfrom feature extraction
        conv_features = F.relu(self.conv_trans1(x))
        conv_features = F.relu(self.conv_trans2(conv_features))
        conv_features = F.relu(self.conv_trans3(conv_features))

        conv_features = conv_features.view(conv_features.size(0),-1)

        state_value = self.state_value(conv_features)
        actions_advantages = self.action_advantages(conv_features)

        result = state_value + actions_advantages

        normalized_result = result - actions_advantages.mean(dim = 1, keepdim
= True)  # substract the mean or max of these advantages
```

```
        return  normalized_result




#OPTIMIZE THE MEAN SQUARED ERROR
criterion = nn.MSELoss()

# Naive version of the Q-
loss, by lopping through each sample and collecting the loss
# And summming it into the batch loss, which is then back-propagated

def naive_dqn_loss_V2(batch, main_net, target_net, gamma, batch_size = 32, dev
ice = "cuda", optimizer = 'opt'):

    tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards,n
on_final_mask = batch

    # create predicted (NN output)
    predicted_state_action_values = main_net(tensor_states) # dims = batch_siz
e,number_of_actions

    next_state_values = torch.zeros((batch_size,predicted_state_action_values.
size()[1]),  device=device)

    #print(next_state_values)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_s
tates) #
        next_state_values = next_state_values.detach()

    next_state_values = next_state_values * gamma

    batch_loss = 0.0

    for prediction, reward, action,  tensor in zip(predicted_state_action_valu
es,tensor_rewards,tensor_actions, next_state_values):

        target = tensor.max() + reward

        target = target.detach() # detach, no gradient flow to the ground Trut
h!!!
        prediction = prediction[action]

        sample_loss = criterion(prediction,target)

        batch_loss += sample_loss
```

```
    return batch_loss / batch_size




#  Calculate the loss of the objective function (MSE), using all the parameter
s such as Gamma
def simple_dqn_loss(batch, main_net, target_net, gamma, batch_size = 32, devic
e="cuda"):

    tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards,n
on_final_mask = batch

    predicted_state_action_values = main_net(tensor_states).gather(1, tensor_a
ctions.unsqueeze(-1)).squeeze(-1)

    next_state_values = torch.zeros(batch_size, device=device)

    with torch.no_grad():
        #compute next state values using Target network
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_s
tates).max(1)[0] #only take the values and not the indices
        #detach from pytorch computation graph
        next_state_values = next_state_values.detach()


    #target values using Bellman aproximation
    target_state_action_values = tensor_rewards + (gamma * next_state_values )



    return criterion(predicted_state_action_values,target_state_action_values)




"""
To be used with the Dueling DQN arhicteecture so that the implementation
fits the description of the "Dueling Double DQN "

"""
def double_dqn_loss(batch, main_net, target_net, gamma, batch_size = 32 , devi
ce="cuda"):
```

```
      tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards
,non_final_mask = batch

      #predicted q values unsing the main network
      predicted_state_action_values = main_net(tensor_states).gather(1, tensor
_actions.unsqueeze(-1)) #predicted q values using the main network
      predicted_state_action_values = predicted_state_action_values.squeeze(-
1)

      next_state_values = torch.zeros(batch_size, device=device)

      with torch.no_grad():

            next_state_acts = main_net(tensor_next_non_final_states).max(1)[1]
.view(tensor_next_non_final_states.shape[0], 1)#using the main network, to get
 the next state action

            next_state_values[non_final_mask] = target_net(tensor_next_non_fin
al_states).gather(1, next_state_acts).squeeze(-1) #calculate the q-
value for the next state action to be taken using the target network
            next_state_values = next_state_values.detach()

      #target values using Bellman aproximation
      target_state_action_values =  tensor_rewards + (gamma * next_state_value
s) #target values for our loss function


      return criterion(predicted_state_action_values,target_state_action_value
s)     #Simple MSE loss between target and predicted
```

Helper functions for DQNs

The Replay Memory class is adapted from the official Pytorch Documentation on DQN.

The code can be found at:
https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

```
import torch
import torch.nn as nn
import numpy as np
import collections
from tensorboardX import SummaryWriter

RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

torch.backends.cudnn.deterministic = True
```

```
torch.backends.cudnn.benchmark = False

Experience = collections.namedtuple('Experience', field_names=['state', 'actio
n', 'reward','done', 'new_state'])


def take_env_step(net, state, env,  epsilon, buffer, device = 'cuda'):

    action = select_action(epsilon, state, env, device, net)

    # do step in the environment, also return some info
    new_state, reward, is_done, _ =  env.step(action)

    #create new Experience transition
    exp = Experience(state, action, reward, is_done, new_state)

    # Add experience to the Buffer
    buffer.append(exp)


    return reward, new_state, is_done




def select_action(e,state,env,device,main_net):
      action=None
      with torch.no_grad():

            if np.random.random() >= e:

                  state = np.array([state], copy=False)
                  state_tensor = torch.tensor(state).to(device) #create pytorc
h tensor for state
                  action_v = main_net(state_tensor).max(1)[1].view(1, 1)
                  action = int(action_v.item())

            else:

                  action = env.action_space.sample() #random action to take

      return action


def print_and_save_logs(writer, steps_count, epsilon, mean_reward_100, reward,
 episodes_completed):

    writer.add_scalar("epsilon", epsilon, steps_count)
    writer.add_scalar("episode reward", reward, steps_count)
```

```
    writer.add_scalar("mean_reward_100", mean_reward_100, steps_count)

    print("Steps taken: %d, Episodes completed: %d , Current epsilon: %f, Mean
 reward achieved: %f " % (steps_count,episodes_completed, epsilon, mean_reward
_100))


class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def append(self, experience):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)

        self.memory[self.position] = experience
        self.position = (self.position + 1) % self.capacity



    def sample(self, batch_size):

        indices = np.random.choice(len(self.memory), batch_size, replace=False
)

        states, actions, rewards, dones, next_states = zip(*[self.memory[index
] for index in indices])

        return np.array(states,dtype=np.float32), np.array(actions), \
               np.array(rewards, dtype=np.float32), \
               np.array(dones, dtype=np.uint8), \
               np.array(next_states)



    def to_pytorch_tensors(self, batch, device):

        states, actions, rewards, dones, next_states = batch
        tensor_states = torch.tensor(states).to(device)
        tensor_next_states = torch.tensor(next_states).to(device)
        tensor_actions = torch.tensor(actions, dtype=torch.long).to(device) #a
ctions must be of type Long for some reason
        tensor_rewards = torch.tensor(rewards).to(device)
        done_flags = torch.BoolTensor(dones).to(device)
```

```
        non_final_mask = ~ done_flags #not operation on the boolean tensor
        non_final_next_states = tensor_next_states[non_final_mask] #tensor of
non final states, to pass through the target net

        return tensor_states,non_final_next_states,tensor_actions,tensor_rewar
ds,non_final_mask


    def __len__(self):
        return len(self.memory)
```

Script for training agents on 'Demon Attack'

```
from demon_skeleton_libraries import skeleton_helper_functions
from demon_skeleton_libraries import demon_env_wrappers as env_wrappers
from demon_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import time # just to have timestampsAgent in the file
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter


RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False


agent_type = 'DoubleDQN'
#print(torch.__version__)
print(agent_type)

"""
ENVIRONMENT AND DIFFICULTY SETTINGS
"""
ENV_NAME = "DemonAttackNoFrameskip-v4"
difficulty = 1
```

```
print ('Difficulty SELECTED!!!!!!  ', difficulty)


env = env_wrappers.make_train_env(ENV_NAME, difficulty, save_every_x_games = 1
0, name_of_game = ENV_NAME,
                                  net_type = agent_type, stacked_frames = 2)
# What game is being played?
print(env.env.game)
# What are the available difficulties?
print(env.ale.getAvailableDifficulties())



"""
Setting the training parameters.
"""
# Train on CUDA IF POSSIBLE
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Training on: ' + str(device))



##########################
####################
NETS_SYNCH_TARGET = 1_000
REPLAY_BUFFER_START_SIZE_FOR_LEARNING = 10_000

LEARNING_RATE = 1e-4
REPLAY_BUFFER_SIZE = 100_000

EPSILON_DECAY_LAST_FRAME = 100_000
EPSILON_START = 1.0
EPSILON_FINAL = 0.01

GAMMA_FACTOR = 0.99
BATCH_SIZE = 32

MAXIMUM_STEPS_ALLOWED = 500_000
TARGET_REWARD = 100_000

########################################
###################################


if agent_type == "DQN" or agent_type == "DoubleDQN" :

      net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_space.sh
ape,env.action_space.n).to(device)
```

```
      target_net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_s
pace.shape,env.action_space.n).to(device)

elif agent_type == "DuelingDoubleDQN" :

      net = skeleton_dqn_architectures.Dueling_CNN_DQN(env.observation_space.s
hape,env.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.Dueling_CNN_DQN(env.observation_
space.shape,env.action_space.n).to(device)



# PERFORMANCE TRACKER IN TENSORBOARD
writer = SummaryWriter(log_dir = "runs/Paper runs/" + agent_type +" -Diff ---
TESTING--" + str(difficulty) + " " + ENV_NAME)
print("DQN ARCHITECTURE \n", net)

buffer = skeleton_helper_functions.ReplayMemory(REPLAY_BUFFER_SIZE)
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)



epsilon = EPSILON_START
state = env.reset()
episode_reward = 0.0
episode_rewards = []
steps_count = 0
max_mean_reward = None


while True:

  steps_count += 1

  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env,  epsilon,
                                                                    buffer, d
evice = device)
  episode_reward += step_reward


  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])
```

```
      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
         # torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_di
ff_-TESTING-'+str(difficulty)+
         #            "-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100


      # the episode is finished so reset the env and the episode reward
      episode_reward = 0.0
      state = env.reset()



  """
  Load the buffer first!!!!! before synchning or back-
propagating and learning on data
  """

  if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
      if steps_count % 1000 == 0:
            print("Populating buffer")
      continue



  if steps_count == MAXIMUM_STEPS_ALLOWED:
        print("Maximum steps allowed reached, training finished.")
        break

  if max_mean_reward == TARGET_REWARD:
        print('Desired reward achieved')
        break


  if steps_count % NETS_SYNCH_TARGET == 0:
      print('Nets synched')
      target_net.load_state_dict(net.state_dict())
```

```
  """
  Gradient Descent below

  The simple DQN uses the simple_dqn() TD-LOSS FUNCTION

  The DuelingDouble uses the double_dqn_loss() TD-LOSS FUNCTION
  """
  optimizer.zero_grad()

  batch = buffer.sample(BATCH_SIZE)

  batch = buffer.to_pytorch_tensors(batch,device)

  batch_loss = None


  if agent_type == "DQN":

        batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                        gamma = GAMMA_FACTOR,
                                                        batch_size = BATCH_SIZE,

                                                        device = device)


  elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

        batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                        gamma = GAMMA_FACTOR,
                                                        batch_size = BATCH_SIZE,

                                                        device = device)
  # Calculate gradients for the weights
  batch_loss.backward()

  # Optimize the weights
  optimizer.step()
```

Testing script for the DuelingDQN architecture for 'Demon Attack'

```
from demon_skeleton_libraries import skeleton_helper_functions
from demon_skeleton_libraries import demon_env_wrappers as env_wrappers
```

```
from demon_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import time
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter

import time


device = 'cuda' #run the testing experiments on cpu


env_name  = 'DemonAttackNoFrameskip-v4'
difficulty = 0 # Variable and easy to change!!!! {0-> 'Easy', 1->'Hard'}
agent = '.dat'  # variable and easy to change


#DuelingDoubleDQNs
#DemonAttackNoFrameskip-v4_diff_0-best-reward=155.45
#DemonAttackNoFrameskip-v4_diff_1-best-reward=78.8
#DemonAttackNoFrameskip-v4_diff_Curriculum_V2-best-reward=184.1


# Must keep an eye on the difficulty setting and the number of frames stacked
(Must be the same number as in the training procedure)
save_10_episodes = 10

#This below must be modified
env = env_wrappers.make_test_env(env_name, difficulty, save_10_episodes, name_
of_game ='Demon', net_type = 'DQN_'+agent, stacked_frames=2)
env = env.unwrapped
print('Max steps',env._max_episode_steps)

# Make sure to load the correct Agent to test!!!!
net = skeleton_dqn_architectures.Dueling_CNN_DQN(env.observation_space.shape,e
nv.action_space.n).to(device)
state = torch.load('models/DuelingDoubleDQN/'+agent, map_location=lambda stg,
_: stg)   #Source directory also changes for each agent type
net.load_state_dict(state)


writer = SummaryWriter(log_dir = "runs/Paper runs/Test runs/" +'DuelingDoubleD
QN_ '+ agent +" -Diff " + str(difficulty))
# Agent makes only non-random moves!!! so the epsilon rate is set to 0!
num_of_eps = 100
episode_reward = 0.0
```

```
episode_rewards = []


e = 0.05
steps_count = 0
episode_steps = 0
for i in range(num_of_eps):
    print('Episode :', i)
    state = env.reset()

    while (True):
            episode_steps += 1
            steps_count +=1

            #Uncomment to render environment
            #env.render()

            if np.random.random() >= e:
                  state = np.array([state], copy=False)
                  state_tensor = torch.tensor(state).to(device) #create pytorc
h tensor for state

                  action_v = net(state_tensor).max(1)[1].view(1, 1) #get actio
n with the highest value
                  action_index = int(action_v.item())

            else:

                  action_index = env.action_space.sample() #random action to t
ake

            state, step_reward, done, info = env.step(action_index)
            episode_reward += step_reward

            if done:
                print('Steps taken: ',episode_steps)
                episode_rewards.append(episode_reward)
                #LOG PERFORMANCE with tensorboard
                writer.add_scalar("Rewards", episode_reward, len(episode_rewar
ds)-1)
                writer.add_scalar("Mean Reward", np.mean(episode_rewards[-
10:]),steps_count)
                writer.add_scalar('Cummmulative reward',np.array(episode_rewar
ds).sum(),len(episode_rewards)-1)
                writer.add_scalar('Steps per episode',episode_steps, len(episo
de_rewards))

                print(episode_reward)
                print('\n')
```

```
                episode_reward = 0.0
                episode_steps = 0
                break


print(episode_rewards)
print('Cummulative reward: ',np.array(episode_rewards).sum())
```

Testing script for DQN and DoubleDQN for 'Demon Attack'

```
from demon_skeleton_libraries import skeleton_helper_functions
from demon_skeleton_libraries import demon_env_wrappers as env_wrappers
from demon_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter


device = 'cuda'
env_name  = 'DemonAttackNoFrameskip-v4'
difficulty = 0 # Variable and easy to change!!!! {0-> 'Easy', 1->'Hard'}
agent = 'DemonAttackNoFrameskip-v4_diff_0-best-
reward=440.85.dat'  # variable and easy to change


#Models to test:

#DQNs:
# DemonAttackNoFrameskip-v4_diff_0-best-reward=440.85
# DemonAttackNoFrameskip-v4_diff_1-best-reward=350.0
# DemonAttackNoFrameskip-v4_diff_Curriculum_V2-best-reward=287.45

#Double DQNs
# DemonAttackNoFrameskip-v4_diff_0-best-reward=107.15
# DemonAttackNoFrameskip-v4_diff_1-best-reward=135.35
# DemonAttackNoFrameskip-v4_diff_Curriculum_V2-best-reward=271.9

save_10_episodes = 10
env = env_wrappers.make_test_env(env_name, difficulty, save_10_episodes, name_
of_game ='Demon', net_type = 'DQN_'+agent, stacked_frames=2)

# Make sure to load the correct Agent to test!!!!
```

```
net = skeleton_dqn_architectures.NATURE_CNN_DQN(env.observation_space.shape,en
v.action_space.n).to(device)
state = torch.load('models/DQN/'+agent, map_location=lambda stg, _: stg)   #So
urce directory also changes for each agent type
net.load_state_dict(state)


writer = SummaryWriter(log_dir = "runs/Test runs/" +'DQN_ '+ agent +" -
Diff " + str(difficulty))
num_of_eps = 100
episode_reward = 0.0 # Total reward for an episode
episode_rewards = []


e = 0.05
steps_count = 0
episode_steps = 0
for i in range(num_of_eps):
    print('Episode :', i)
    state = env.reset()
    #print(state.shape)
    while (True):
            episode_steps += 1
            steps_count +=1

            #Uncomment to render environment
            #env.render()


            if np.random.random() >= e:
                  state = np.array([state], copy=False)
                  state_tensor = torch.tensor(state).to(device) #create pytorc
h tensor for state
                  print(state_tensor.shape)
                  action_v = net(state_tensor).max(1)[1].view(1, 1) #get actio
n with the highest value
                  action_index = int(action_v.item())

            else:

                  action_index = env.action_space.sample() #random action to t
ake



            state, step_reward, done, info = env.step(action_index)
            episode_reward += step_reward

            if done:
                print('Steps taken: ',episode_steps)
```

```
                episode_rewards.append(episode_reward)
                #LOG PERFORMANCE with tensorboard
                writer.add_scalar("Rewards", episode_reward, len(episode_rewar
ds)-1)
                writer.add_scalar("Mean Reward", np.mean(episode_rewards[-
10:]),steps_count)
                writer.add_scalar('Cummmulative reward',np.array(episode_rewar
ds).sum(),len(episode_rewards)-1)
                writer.add_scalar('Steps per episode',episode_steps, len(episo
de_rewards))

                print('Reward',episode_reward)
                print('\n')

                episode_reward = 0.0
                episode_steps = 0
                break


print(episode_rewards)
print('Cummulative reward: ',np.array(episode_rewards).sum())
```

Script for random agent run for 'Demon Attack'

```
from demon_skeleton_libraries import skeleton_helper_functions
from demon_skeleton_libraries import demon_env_wrappers as env_wrappers
from demon_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import time # just to have timestampsAgent in the file
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter


#RANDOM_SEED = 0
#np.random.seed(RANDOM_SEED)
#torch.manual_seed(RANDOM_SEED)

#torch.backends.cudnn.deterministic = True
##torch.backends.cudnn.benchmark = False


agent_type = 'RANDOM_3' # this changes across files
print(torch.__version__)
```

```
"""
ENVIRONMENT AND DIFFICULTY SETTINGS
"""
ENV_NAME = "DemonAttackNoFrameskip-v4"
difficulty = 1

print ('Difficulty SELECTED!!!!!!  ', difficulty)


env = env_wrappers.make_train_env(ENV_NAME, difficulty, save_every_x_games = 1
_000, name_of_game = ENV_NAME,
                                  net_type = agent_type, stacked_frames = 2)

# What game is being played?
print(env.env.game)
# What are the available difficulties?
print(env.ale.getAvailableDifficulties())



"""
Setting the training parameters.
"""
# Train on CUDA IF POSSIBLE
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Training on: ' + str(device))


##################################
MAXIMUM_STEPS_ALLOWED = 512_000
##########################################

# PERFORMANCE TRACKER IN TENSORBOARD
writer = SummaryWriter(log_dir = "runs/Train runs/" + agent_type +" -
Diff " + str(difficulty) + " " + ENV_NAME)


state = env.reset()
episode_reward = 0.0
episode_rewards = []
steps_count = 0


buffer = None

while True:

  steps_count += 1
```

```
  epsilon = 1.0
  action = skeleton_helper_functions.select_action(epsilon,state,env,None,None
)

  state, reward, is_done, _ =  env.step(action)

  episode_reward += reward


  if is_done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])

      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      episode_reward = 0.0
      state  = env.reset()


  if steps_count == MAXIMUM_STEPS_ALLOWED:
        print("Maximum steps allowed reached, training finished.")
        break
```

Training script for curriculum learning for 'Demon Attack'

```
from demon_skeleton_libraries import skeleton_helper_functions
from demon_skeleton_libraries import demon_env_wrappers as env_wrappers
from demon_skeleton_libraries import skeleton_dqn_architectures

import numpy as np
import time # just to have timestampsAgent in the file
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter


RANDOM_SEED = 0
```

```
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False


agent_type = 'DoubleDQN'
print(torch.__version__)

"""
ENVIRONMENT AND DIFFICULTY SETTINGS
"""
ENV_NAME = "DemonAttackNoFrameskip-v4"


env_easy  = env_wrappers.make_train_env_curriculum(ENV_NAME, 0, save_every_x_g
ames = 10,
                                                   name_of_game = ENV_NAME, ne
t_type = agent_type,
                                                   stacked_frames = 2)

env_hard  = env_wrappers.make_train_env_curriculum(ENV_NAME, 1, save_every_x_g
ames = 10,
                                                   name_of_game = ENV_NAME, ne
t_type = agent_type,
                                                   stacked_frames = 2)

env_easy_2  = env_wrappers.make_train_env_curriculum(ENV_NAME, 0, save_every_x
_games = 10,
                                                   name_of_game = ENV_NAME, ne
t_type = agent_type,
                                                   stacked_frames = 2)

# What game is being played?
print(env_easy.env.game)
# What are the available difficulties?
print(env_easy.ale.getAvailableDifficulties())


"""
Setting the training parameters.
"""
# Train on CUDA IF POSSIBLE
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Training on: ' + str(device))


##################################
```

```
##############################
##########################
###################
NETS_SYNCH_TARGET = 1_000
REPLAY_BUFFER_START_SIZE_FOR_LEARNING = 10_000
LEARNING_RATE = 1e-4
REPLAY_BUFFER_SIZE = 100_000

EPSILON_START = 1.0
EPSILON_FINAL = 0.01
GAMMA_FACTOR = 0.99
BATCH_SIZE = 32



# Must be expnaded to exploret the 'Hard' environment as well.
EPSILON_DECAY_LAST_FRAME = 300_000 # must allow for exploration in both 'easy'
 and 'hard' settings.

#Must add up to 500k
EASY_STEPS = 200_000
HARDER_STEPS = 200_000
FINAL_EASY_STEPS = 100_000

########################################
###################################
#################################
########################################

print(env_easy.observation_space.shape)


if agent_type == "DQN" or agent_type == "DoubleDQN" :

      net = skeleton_dqn_architectures.NATURE_CNN_DQN(env_easy.observation_spa
ce.shape,env_easy.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.NATURE_CNN_DQN(env_easy.observat
ion_space.shape,env_easy.action_space.n).to(device)

elif agent_type == "DuelingDoubleDQN" :

      net = skeleton_dqn_architectures.Dueling_CNN_DQN(env_easy.observation_sp
ace.shape,env_easy.action_space.n).to(device)
      target_net = skeleton_dqn_architectures.Dueling_CNN_DQN(env_easy.observa
tion_space.shape,env_easy.action_space.n).to(device)


# PERFORMANCE TRACKER IN TENSORBOARD
```

```
writer = SummaryWriter(log_dir = "runs/Paper runs/" + agent_type +" -
Diff " + "Curriculum_V2" + " " + ENV_NAME)
print("DQN ARCHITECTURE \n", net)



buffer = skeleton_helper_functions.ReplayMemory(REPLAY_BUFFER_SIZE)
epsilon = EPSILON_START
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)

'''
V2
1) 200K STEPS ON EASY
2) 200K STEPS ON HARD
3) 100K STEPS ON EASY
'''


episode_reward = 0.0
episode_rewards = []
steps_count = 0
max_mean_reward = None



state = env_easy.reset()
for i  in range(EASY_STEPS):

  steps_count += 1
  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env_easy,  epsilon, buffer, device = device)
  episode_reward += step_reward

  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
```

```
          #torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_di
ff_'+'curriculumV2'+"-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100


      # the episode is finished so reset the env and the episode reward
      episode_reward = 0.0
      state = env_easy.reset()



  """
  Load the buffer first!!!!! before synchning or back-
propagating and learning on data
  """

  if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
      print("Populating buffer")
      continue


  if steps_count % NETS_SYNCH_TARGET == 0:
      print('Nets synched')
      target_net.load_state_dict(net.state_dict())



  optimizer.zero_grad()
  batch = buffer.sample(BATCH_SIZE)
  batch = buffer.to_pytorch_tensors(batch,device)
  batch_loss = None

  if agent_type == "DQN":

        batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                          gamma = GAMMA_FACTOR,
                                                          batch_size = BATCH_SIZE,

                                                          device = device)

  elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

        batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
```

```
                                                    gamma = GAMMA_FACTOR,
                                                    batch_size = BATCH_SIZE,

                                                    device = device)
  batch_loss.backward()

  optimizer.step()




print('HARD TRAINING STARTED')
episode_reward = 0.0 # episode reward reset to 0

state = env_hard.reset()
for i  in range(HARDER_STEPS):

  steps_count += 1

  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env_hard,  epsilon, buffer, device = device)
  episode_reward += step_reward


  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
          torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_dif
f_'+'curriculum'+"-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100
```

```
        # the episode is finished so reset the env and the episode reward
        episode_reward = 0.0
        state = env_hard.reset()



    """
    Load the buffer first!!!!! before synchning or back-
propagating and learning on data
    """

    if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
        print("Populating buffer")
        continue


    if steps_count % NETS_SYNCH_TARGET == 0:
        print('Nets synched')
        target_net.load_state_dict(net.state_dict())



    optimizer.zero_grad()

    batch = buffer.sample(BATCH_SIZE)

    batch = buffer.to_pytorch_tensors(batch,device)


    batch_loss = None


    if agent_type == "DQN":

           batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                   gamma = GAMMA_FACTOR,
                                                   batch_size = BATCH_SIZE,

                                                   device = device)


    elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

           batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                   gamma = GAMMA_FACTOR,
```

```
                                                    batch_size = BATCH_SIZE,

                                                    device = device)
  batch_loss.backward()

  optimizer.step()



print('FINAL EASY TRAINING STARTED')
episode_reward = 0.0 # episode reward reset to 0

state = env_easy_2.reset()
for i  in range(FINAL_EASY_STEPS):

  steps_count += 1

  epsilon = max(EPSILON_FINAL, EPSILON_START - steps_count / EPSILON_DECAY_LAS
T_FRAME)
  step_reward, state, done = skeleton_helper_functions.take_env_step(net, stat
e, env_easy_2,  epsilon, buffer, device = device)
  episode_reward += step_reward


  if done: # if the episode is done, then log the episode reward

      episode_rewards.append(episode_reward) # holds reward for each game
      mean_reward_100 = np.mean(episode_rewards[-100:])


      skeleton_helper_functions.print_and_save_logs(writer, steps_count, epsil
on,
                                                    mean_reward_100,
                                                    episode_reward,
                                                    len(episode_rewards))

      if max_mean_reward is None or max_mean_reward < mean_reward_100:
          torch.save(net.state_dict(), 'models/'+agent_type+'/'+ENV_NAME+'_dif
f_'+'curriculum'+"-best_%.0f.dat" % mean_reward_100)

          if max_mean_reward is not None:
              print("Best reward updated %.3f -
> %.3f" % (max_mean_reward, mean_reward_100))
          max_mean_reward = mean_reward_100


      # the episode is finished so reset the env and the episode reward
      episode_reward = 0.0
      state = env_easy_2.reset()
```

```
  """
  Load the buffer first!!!!! before synchning or back-
propagating and learning on data
  """

  if len(buffer) < REPLAY_BUFFER_START_SIZE_FOR_LEARNING:
      print("Populating buffer")
      continue


  if steps_count % NETS_SYNCH_TARGET == 0:
      print('Nets synched')
      target_net.load_state_dict(net.state_dict())



  optimizer.zero_grad()

  batch = buffer.sample(BATCH_SIZE)

  batch = buffer.to_pytorch_tensors(batch,device)


  batch_loss = None


  if agent_type == "DQN":

        batch_loss = skeleton_dqn_architectures.simple_dqn_loss(batch, net, ta
rget_net,
                                                          gamma = GAMMA_FACTOR,
                                                          batch_size = BATCH_SIZE,

                                                          device = device)


  elif agent_type == "DoubleDQN" or agent_type == "DuelingDoubleDQN" :

        batch_loss = skeleton_dqn_architectures.double_dqn_loss(batch, net, ta
rget_net,
                                                          gamma = GAMMA_FACTOR,
                                                          batch_size = BATCH_SIZE,

                                                          device = device)
  batch_loss.backward()
```

```
  optimizer.step()



print('Training finished')
```

Script for environment wrappers for ‘’Demon Attack’

These wrappers have been adapted from the OpenAI baselines repository

```
import numpy as np
from collections import deque
import gym
from gym import spaces
import cv2
import gym
cv2.ocl.setUseOpenCL(False)



RANDOM_SEED = 0
np.random.seed(RANDOM_SEED)



class NoopResetEnv(gym.Wrapper):
    def __init__(self, env, noop_max=30):
        """Sample initial states by taking random number of no-ops on reset.
        No-op is assumed to be action 0.
        """
        gym.Wrapper.__init__(self, env)
        self.noop_max = noop_max
        self.override_num_noops = None
        self.noop_action = 0
        assert env.unwrapped.get_action_meanings()[0] == 'NOOP'

    def reset(self, **kwargs):
        """ Do no-op action for a number of steps in [1, noop_max]."""
        self.env.reset(**kwargs)
        if self.override_num_noops is not None:
            noops = self.override_num_noops
        else:
            noops = self.unwrapped.np_random.randint(1, self.noop_max + 1)  #
pylint: disable=E1101
        assert noops > 0
        obs = None
        for _ in range(noops):
```

```
            obs, _, done, _ = self.env.step(self.noop_action)
            if done:
                obs = self.env.reset(**kwargs)
        return obs

    def step(self, ac):
        return self.env.step(ac)


class FireResetEnv(gym.Wrapper):
    def __init__(self, env):
        """Take action on reset for environments that are fixed until firing."
""
        gym.Wrapper.__init__(self, env)
        assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
        assert len(env.unwrapped.get_action_meanings()) >= 3

    def reset(self, **kwargs):
        self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(1)
        if done:
            self.env.reset(**kwargs)
        obs, _, done, _ = self.env.step(2)
        if done:
            self.env.reset(**kwargs)
        return obs

    def step(self, ac):
        return self.env.step(ac)


class EpisodicLifeEnv(gym.Wrapper):
    def __init__(self, env):
        """Make end-of-life == end-of-
episode, but only reset on true game over.
        Done by DeepMind for the DQN and co. since it helps value estimation.
        """
        gym.Wrapper.__init__(self, env)
        self.lives = 0
        self.was_real_done = True

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        self.was_real_done = done
        # check current lives, make loss of life terminal,
        # then update lives to handle bonus lives
        lives = self.env.unwrapped.ale.lives()
        if lives < self.lives and lives > 0:
```

```
            # for Qbert sometimes we stay in lives == 0 condtion for a few fra
mes
            # so its important to keep lives > 0, so that we only reset once
            # the environment advertises done.
            done = True
        self.lives = lives
        return obs, reward, done, info

    def reset(self, **kwargs):
        """Reset only when lives are exhausted.
        This way all states are still reachable even though lives are episodic
,
        and the learner need not know about any of this behind-the-scenes.
        """
        if self.was_real_done:
            obs = self.env.reset(**kwargs)
        else:
            # no-op step to advance from terminal/lost life state
            obs, _, _, _ = self.env.step(0)
        self.lives = self.env.unwrapped.ale.lives()
        return obs


class MaxAndSkipEnv(gym.Wrapper):
    def __init__(self, env, skip=4):
        """Return only every `skip`-th frame"""
        gym.Wrapper.__init__(self, env)
        # most recent raw observations (for max pooling across time steps)
        self._obs_buffer = np.zeros((2,) + env.observation_space.shape, dtype=
np.uint8)
        self._skip = skip

    def reset(self):
        return self.env.reset()

    def step(self, action):
        """Repeat action, sum reward, and max over last observations."""
        total_reward = 0.0
        done = None
        for i in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            if i == self._skip - 2: self._obs_buffer[0] = obs
            if i == self._skip - 1: self._obs_buffer[1] = obs
            total_reward += reward
            if done:
                break
        # Note that the observation on the done=True frame
        # doesn't matter
        max_frame = self._obs_buffer.max(axis=0)
```

```
        return max_frame, total_reward, done, info

    def reset(self, **kwargs):
        return self.env.reset(**kwargs)

#Reward Wrapper
class ClipRewardEnv(gym.RewardWrapper):
    def __init__(self, env):
        gym.RewardWrapper.__init__(self, env)

    def reward(self, reward):
        """Bin reward to {+1, 0, -1} by its sign."""
        return np.sign(reward)

#Observation Wrapper
class WarpFrame(gym.ObservationWrapper):
    def __init__(self, env):
        """Warp frames to 84x84 as done in the Nature paper and later work.
        Expects inputs to be of shape height x width x num_channels
        """
        gym.ObservationWrapper.__init__(self, env)
        self.width = 84
        self.height = 84
        self.observation_space = spaces.Box(low=0, high=255,
                                            shape=(self.height, self.width, 1)
, dtype=np.uint8)

    def observation(self, frame):
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        frame = cv2.resize(frame, (self.width, self.height), interpolation=cv2
.INTER_AREA)
        return frame[:, :, None]


class FrameStack(gym.Wrapper):
    def __init__(self, env, k):
        """Stack k last frames.
        Returns lazy array, which is much more memory efficient.
        Expects inputs to be of shape num_channels x height x width.
        """
        gym.Wrapper.__init__(self, env)
        self.k = k
        self.frames = deque([], maxlen=k)
        shp = env.observation_space.shape
        self.observation_space = spaces.Box(low=0, high=255, shape=(shp[0] * k
, shp[1], shp[2]), dtype=np.uint8)

    def reset(self):
```

```
        ob = self.env.reset()
        for _ in range(self.k):
            self.frames.append(ob)
        return self._get_ob()

    def step(self, action):
        ob, reward, done, info = self.env.step(action)
        self.frames.append(ob)
        return self._get_ob(), reward, done, info

    def _get_ob(self):
        assert len(self.frames) == self.k
        return LazyFrames(list(self.frames))


class LazyFrames(object):
    def __init__(self, frames):
        """This object ensures that common frames between the observations are
 only stored once.
        It exists purely to optimize memory usage which can be huge for DQN's
1M frames replay
        buffers."""
        self._frames = frames

    def __array__(self, dtype=None):
        out = np.concatenate(self._frames, axis=0)
        if dtype is not None:
            out = out.astype(dtype)
        return out

    def __len__(self):
        return len(self._frames)

    def __getitem__(self, i):
        return self._frames[i]

#Observation Wrapper
class PyTorchFrame(gym.ObservationWrapper):
    """Image shape to num_channels x height x width"""

    def __init__(self, env):
        super(PyTorchFrame, self).__init__(env)
        shape = self.observation_space.shape
        #print("Using PytorchFrame wrapper")
        self.observation_space = gym.spaces.Box(low=0.0, high=1.0, shape=(shap
e[-1], shape[0], shape[1]), dtype=np.uint8)

    def observation(self, observation):
        return np.rollaxis(observation, 2)
```

```
# Function that creates an Environment with the selected Wrappers....

def make_train_env(env_name, difficulty, save_every_x_games = 20, name_of_game
 = 'NAME THE GAME!!!', net_type ='DQN', stacked_frames = 2):

    print('OPENAI GYM Version -->',gym.__version__)

    env = gym.make(env_name, difficulty = difficulty)

    env = NoopResetEnv(env)

    env = MaxAndSkipEnv(env)

    env = EpisodicLifeEnv(env)

    env = FireResetEnv(env)

    env = WarpFrame(env)

    env = PyTorchFrame(env)

    env = FrameStack(env, stacked_frames)   # This is always needed as it defi
nes the MDP nature of the problem

    print('Number of stacked frames: ', stacked_frames)

    #Video saving functionality, save the performace of the agent every x trai
ning games
    env = gym.wrappers.Monitor(env, './videos/training-
videos/'+ name_of_game +'- Diff='+ str(difficulty)+' - NN-'+ net_type,
                                    video_callable=lambda episode_id: episode
_id % save_every_x_games == 0,
                                    force=True)

    env.seed(RANDOM_SEED)
    env.action_space.seed(RANDOM_SEED)


    return env



def make_test_env(env_name, difficulty, save_every_x_games = 20, name_of_game
= 'NAME THE GAME!!!', net_type ='DQN', stacked_frames = 2):

    print('OPENAI GYM Version -->',gym.__version__)
```

```
    env = gym.make(env_name, difficulty = difficulty)

    env = NoopResetEnv(env)

    env = MaxAndSkipEnv(env)

    env = EpisodicLifeEnv(env)

    env = FireResetEnv(env)

    env = WarpFrame(env)

    env = PyTorchFrame(env)

    env = FrameStack(env, stacked_frames)

    print('Stacking frames!!!!!!!!!!!!!!: ', stacked_frames)

    #Video saving functionality, save the performace of the agent every x trai
ning games
    env = gym.wrappers.Monitor(env, './videos/testing-
videos/'+ name_of_game +'- Diff='+ str(difficulty)+' - NN-'+ net_type,
                                       video_callable=lambda episode_id: epis
ode_id % save_every_x_games == 0, force=True)

    #env.seed(RANDOM_SEED)
    #env.action_space.seed(RANDOM_SEED)


    return env




def make_train_env_curriculum(env_name, difficulty, save_every_x_games = 20, n
ame_of_game = 'NAME THE GAME!!!', net_type ='DQN', stacked_frames = 2):

    print('OPENAI GYM Version -->',gym.__version__)

    env = gym.make(env_name, difficulty = difficulty)

    env = NoopResetEnv(env)

    env = MaxAndSkipEnv(env)

    env = EpisodicLifeEnv(env)
```

```
    env = FireResetEnv(env)

    env = WarpFrame(env)

    env = PyTorchFrame(env)

    env = FrameStack(env, stacked_frames)   # This is always needed as it defi
nes the MDP nature of the problem

    print('Number of stacked frames: ', stacked_frames)

    #Video saving functionality, save the performace of the agent every x trai
ning games
    env = gym.wrappers.Monitor(env, './videos/training-
videos/'+ name_of_game +'- Diff='+ "curriculum"+' - NN-'+ net_type,
                                    video_callable=lambda episode_id: episode
_id % save_every_x_games == 0,
                                    force=True)

    env.seed(RANDOM_SEED)
    env.action_space.seed(RANDOM_SEED)


    return env
```