Reinforcement Learning for Atari 2600

COMP9417 Machine Learning Project

Joshua Derbe Sean Luo Arkie Owen Mariya Shmalko

Introduction

The purpose of this project is to design a deep neural network, incorporate various extensions and modify hyperparameters to learn non-trivial Atari 2600 games. We first evaluate the system to learn Pong, and then utilise the findings to run DQN learning on Breakout.

The method used for learning in this project is a Q-learning algorithm with epsilon greedy exploration. This project implements Vanilla DQN, Double DQN, Dueling DQN, Prioritised Replay Buffer and Canny edge detection as extensions, and modifies the learning rate, target network sync rate, and batch size hyperparameters to discover their impact on rate of convergence and efficiency of the program.

Our team was drawn to this particular project for several reasons. Foremost, the idea of training an AI to play classic Atari games was appealing for the novelty, as our team is composed of several video game fans. However a large attraction of the project was that it allows us to work with highly sophisticated Python packages to construct complicated neural networks with relative ease. We are able to experience the process of experimenting with learning networks and tuning hyperparameters in an advanced supervised system without having to reinvent the wheel and code neural networks by hand.

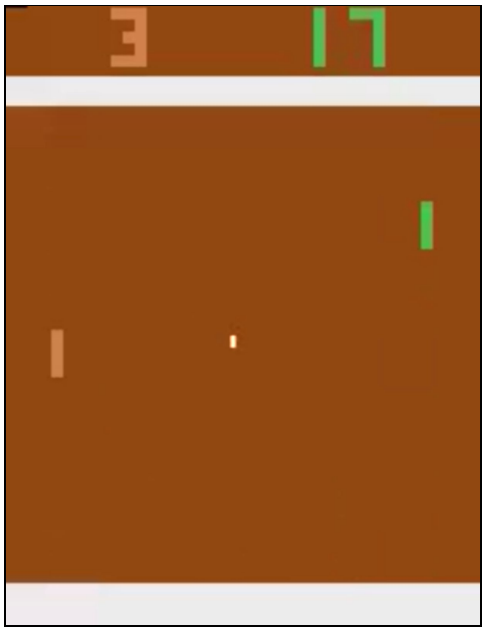
Furthermore, the existence of OpenAI and Gym give us an achievable method of translating Atari game screens to inputs, and control actions to outputs. We were encouraged to take on the challenge by the wide-spanning documentation for this software online.

Implementation

The Atari game environment was simulated using OpenAI gym which allowed us to interact with a simulated environment, with access to all the inputs needed to perform machine learning. The code itself was written in Python 3.8, and uses the Pytorch package for the neural network. TensorBoard is a Python web service which creates an interactive web interface with generated graphs which was used to track important variables in the training process.

The overall method adopted was to first implement DQN and run on Pong, and gradually implement extensions in isolation, comparing their results to the default implementation to see which extensions offer the greatest improvement. Using Pong as a proxy allowed us to experiment with more extensions and hyperparameter settings as it converges in much less frames than Breakout due to its simplicity.

We implemented the DQN algorithm and ran multiple iterations of the program on the Pong environment. These iterations included the tweaking of hyperparameters and extensions to the base vanilla DQN algorithm. We observed whether the program converged to the optimal score (18 - a near-perfect victory) and how many frames this took. Gradually a picture was built of which hyperparameters and extensions are best suited to the task of learning Pong.



After this experimentation process, we applied the findings to Breakout.

Deep Q-Learning:

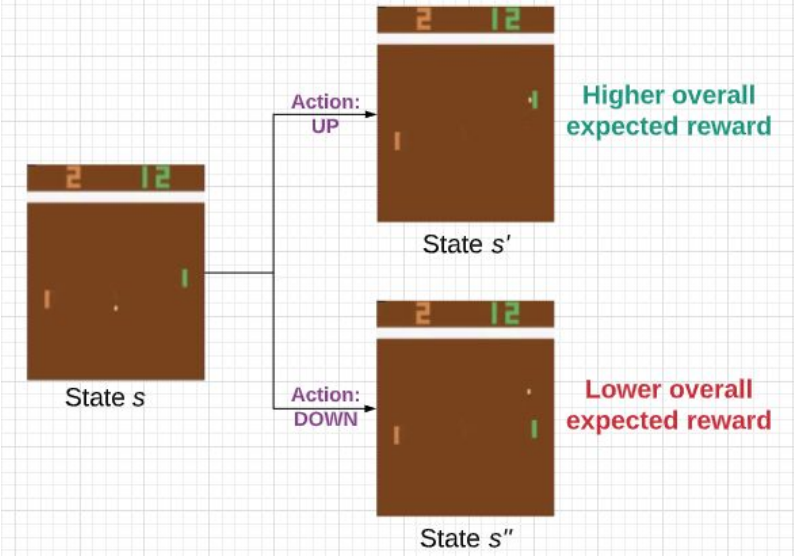
On a fundamental level, the algorithm we selected is based on Q-Learning. This is based on the idea that the program's view of the game can be broken down into a series of states, where each set of the programs input variables can be considered a state. In the case of Pong and Breakout, the program is passed the entire frame of pixels, and so its view of a 'state' would be a collection of frames of the game.

The Q-Learning process involves recording tuples of the form (s, a, r, s'), tuples of $State$, $Action$, $Reward$, $New\ State$. The $Reward$, however, may only come after the result of several actions, so in practice r is

calculated by using the *expected future reward* we can hope to achieve by taking a certain action in a certain state.

$$Q_s(a) = {}_{s'}[r_{s,a} + \gamma V_{s'}]$$

- $Q_s(a)$ is the value of taking action a in state s
- $r_{s,a}$ is the immediate reward of entering state s with action a
- $V_{s'}$ is the value of state s' (which can be calculated recursively)
- γ is a discount hyperparameter



The modelling process keeps a table of these tuples. It works using a neural network which takes the pixels of the screen as input and outputs a Q-value for all possible actions[1]. The neural network performs backpropagation using stochastic gradient descent, minimising loss with respect to the reward of each action taken. Overall this means it tries to learn the probable best actions to take in any given state to maximise its reward (the score of the game).

Neural Network

The vanilla neural network is composed of two components, a convolutional sequence and a linear sequence. The intention of the simple CNN is to extract important features that describe the environment (i.e. character location, obstacles, etc). Such features are then inputted into a sequence of linear layers which represents a function that produces a vector of values each corresponding to the benefits (Q values) of taking an action in a given state. The CNN is composed of three pairs of 2d convolution and ReLU layers. Convolution layers are linear operations that highlight sections inside an image. ReLU activation is used to address the non-linearity in feature extractions and in practice will discard neurons with insignificant weight in the network as means of dimensionality reduction.

Experience Buffer:

In our algorithm, we cannot use gradient descent to train our network after every frame the game plays. This is because our observations are a time series - since we are just passing in subsequent frames from the game, the observations are correlated. They violate the i.i.d. assumption of SGD.

[1] For Pong, this is one of 6 actions in the Gym environment. Two correspond to doing nothing, two move up and two move down.

We use an experience buffer to solve this. The algorithm collects a large amount of data into a replay buffer. It then periodically takes random samples from the buffer in order to provide uncorrelated data to train the neural network. The size of the buffer and frequency of sampling are both hyperparameters of the model.

Target Network:

In our algorithm, we cannot update the neural network we are using while simultaneously training the network. Since Q-values are determined based off the reward of 'best actions', and 'best actions' are determined by the neural network learning off Q-values, we have an unstable cycle of inputs correlated with outputs.

We use a target network to solve this. This involves keeping two neural networks, where one of them (the target network) is updated on a lagged interval. This lagged network calculates Q-values, while the normal network determines actions. The target network is synchronised periodically, where the frequency of synchronisation is a hyperparameter.

Epsilon Decay:

Throughout the algorithm, the concept of "exploration vs exploitation" is a concern for our agent's movements. If our agent simply selects the best action that the neural network decides will maximise reward every time, we will have consistent behaviour but we will lack data on expected reward for alternate actions. Intuitively, we want a greater exploration rate at the beginning of the algorithm and lower rate as time goes on, so that our network draws conclusions on sizable data but eventually hones in on an optimal plan.

We achieve this by modulating epsilon, the "randomness" parameter, over the course of the algorithm's runtime. We initialise the value of epsilon to 1.0, and over the first several thousand frames (this is a hyperparameter) linearly decrease epsilon to its final value of 0.1 (also a hyperparameter). Thus our agent begins by fully "exploring" and ends by fully "exploiting" based on its findings.

Training Algorithm

- ❏ Set hyperparameters
- ❏ Initialise all of:
  - OpenAI environment
  - Experience buffer
  - Agent
  - Neural network and target network
- ❏ Loop until mean reward over past games exceeds some game-specific threshold:
  - Decay epsilon
  - Agent chooses an action based on epsilon, the input and neural network
    - The agent takes an action randomly with probability epsilon
    - Otherwise the agent performs the action with the highest Q value prediction from the neural network
  - (s, a, r, s') tuple is added to the buffer
  - ❏ If the buffer is full:
    - If enough frames have passed, sync target network
    - Sample the buffer
    - Perform batch gradient descent on the sample
    - Backpropagation is performed

Hyperparameters

Changes to critical hyperparameters often result in models with significantly different performance. These are the hyperparameters which our program used to control the learning process when performing vanilla DQN[2].

Wrappers

In order to improve performance and stability of our program we implemented several transformations to the Atari platform interaction. Wrappers affected our environment with different functionalities, such as: reducing the amount of frames we needed to process, rescaling our pixel input to a smaller array, etc.[3]

Experimentation

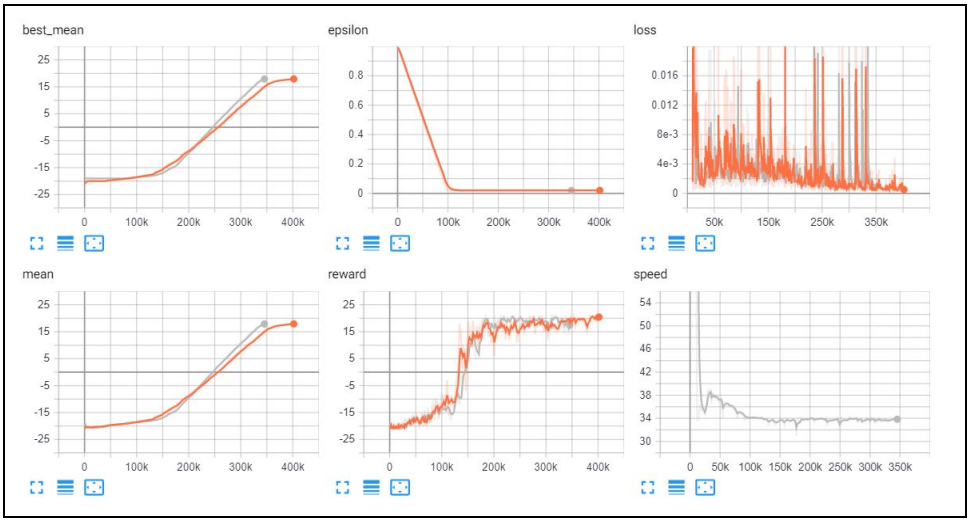
Due to time and resource constraints we were not able to try every combination of extensions and hyperparameters. We did, however, select the combinations which we thought were the most important and informative to the learning process.

Note: In all tensorboards below, the horizontal axis of the graphs in all cases is the total number of frames and the graph labelled ‘speed’ (measures number of frames over time) is an unreliable metric as it will vary depending on the strength of the machine running the code and which background tasks were being performed simultaneously. All graphs show comparisons between tests run on the same machine however some inconsistencies may still occur. Some graphs also were not measuring speed.

Hyperparameter Tuning

Learning rate

Learning rate determines how weights of the network are adjusted with respect to the loss gradient. A learning rate that is too high will not optimise the model as there are chances that the local minima will be skipped over whereas a learning rate which is too low will result in convergence being very slow. With this in mind, the learning rate plays a crucial role in optimisation of a model’s performance as it directly affects how quickly the model can arrive at the best accuracy.



[2] See appendix for full hyperparameter list.
[3] See appendix for full wrapper list.

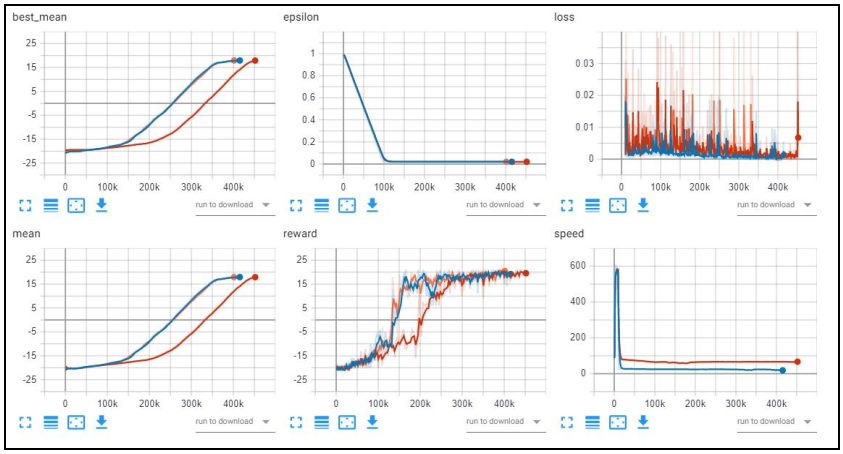
Legend: 0.0001 Learning Rate 0.0002 Learning Rate

In our experimentation with learning rate, we found that a vanilla DQN with huber loss and Kaiming normal initialisation converged to a mean score of 18 in ~400k frames when learning rate was 0.0001 and converged in ~350k frames when the learning rate was 0.0002. Clearly here, a larger learning rate increased convergence speed and no local minimas were skipped.

Batch size

Batch size is the number of observations sampled from the experience buffer at a time and is often limited by a computer's hardware.

Larger batch sizes allows computational gain from the use of parallel GPUs however will lead to poor generalisation. Using smaller batch sizes results in faster convergence rates however does not guarantee that the model will converge to the global optima. Since batch size directly impacts convergence time, it was important for us to tune this hyperparameter to suit our learning problem.



Legend: 16 batch size 32 batch size 128 batch size

In our experimentation with batch size, we found that a vanilla DQN with huber loss and Kaiming normal initialisation converged to a mean score of 18 with these statistics:

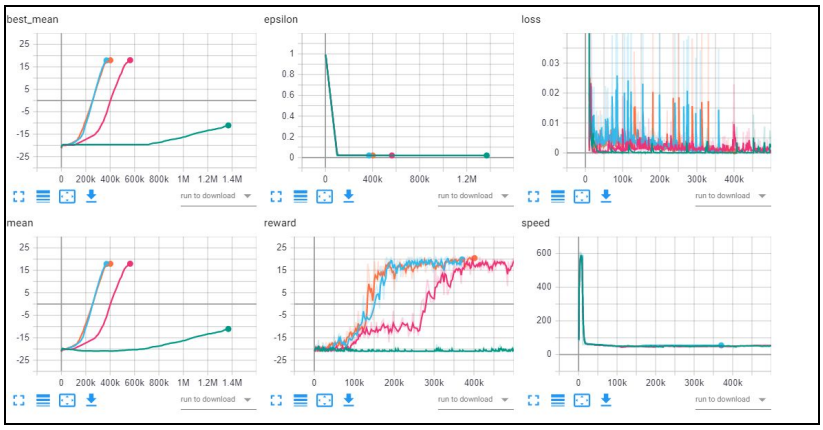
- 16 batch size took ~2 hours and converged to a mean score of 18 in ~450K frames
- 32 batch size took ~2 hours and converged to a mean score of 18 in ~400K frames
- 128 batch size took ~4 hours and converged to a mean score of 18 in ~415K frames

Clearly here a larger batch size of 128 is too big and slows down the program for not much gain. Batch sizes of 16 and 32 were faster to converge. For a simple game like pong, 16 and 32 were about the same however for a more complex game like Breakout, 32 is a more suitable batch size, as 16 may not be giving the neural network enough information to improve quickly.

Frame sync

The rate at which we synchronise our main and target networks has implications for the bias-variance tradeoff in our algorithm. If we synchronise the networks over short periods, we are inhibiting the functionality of the target network and thus our training data becomes more correlated, increasing overall variance. If we synchronise the networks over long periods, we are reducing variance but leaving more time between updating our best estimates of the Q-values of actions, which increases bias.

As a result, experimentation was required to determine an optimal sync for our purposes.



Legend: 100 frames sync 1000 frames sync 10000 frames sync 100000 frames sync

In our experimentation with frame sync , we found that a vanilla DQN with huber loss and Kaiming normal initialisation converged to a mean score of 18 with these statistics:

- 100 frames sync converged to a mean score of 18 in ~370K frames
- 1000 frames sync converged to a mean score of 18 in ~400K frames
- 10000 frames sync converged to a mean score of 18 in ~560K frames
- 100000 frames sync did not converge in a reasonable time and was cancelled

As such, for Pong a smaller sync rate is better as too large a sync rate will result in the Q values not being updated quick enough resulting in slow convergence rates. At the same time, too low a sync rate can result in Q values fluctuating too much. We used a frame sync value of 1000. For Breakout, we chose a larger sync rate of 10,000 as values may fluctuate too much and, again, breakout is more complex than Pong.

Loss function

The traditional loss function used for these kinds of problems is mean squared error (MSE) loss. MSE loss is usually implemented as:

$$L = (actual - predicted)^2$$

Unfortunately, this loss function has the tendency to overestimate the loss of large differences between the actual value and the predicted, and can be vulnerable to outliers and exploding gradients. Exploding gradients is a problem where large error gradients accumulate and result in very large updates to the neural network during training, and can thus be unstable. In an effort to combat this, we experimented between using MSE and Huber loss.

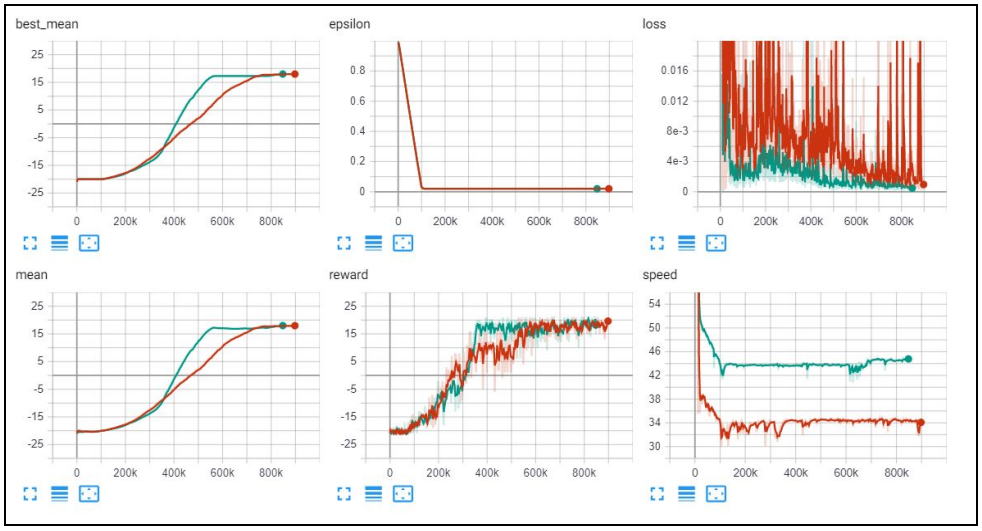
Huber loss is implemented in Pytorch as:

$$\text{loss}(x,y)=\frac{1}{n}\sum_i z_i$$

where z_i is given by:

$$z_i=\begin{cases}0.5(x_i-y_i)^2, & \text{if } |x_i-y_i|<1\\ |x_i-y_i|-0.5, & \text{otherwise}\end{cases}$$

This effectively “clips” the gradient of the loss between -1 and 1, and thus accompanied with the 0.5 factor, prevents loss values from being too high. Running vanilla DQN with both MSE loss and Huber loss gave the following results:



Legend:

Vanilla DQN with MSE loss without Kaiming normal initialisation

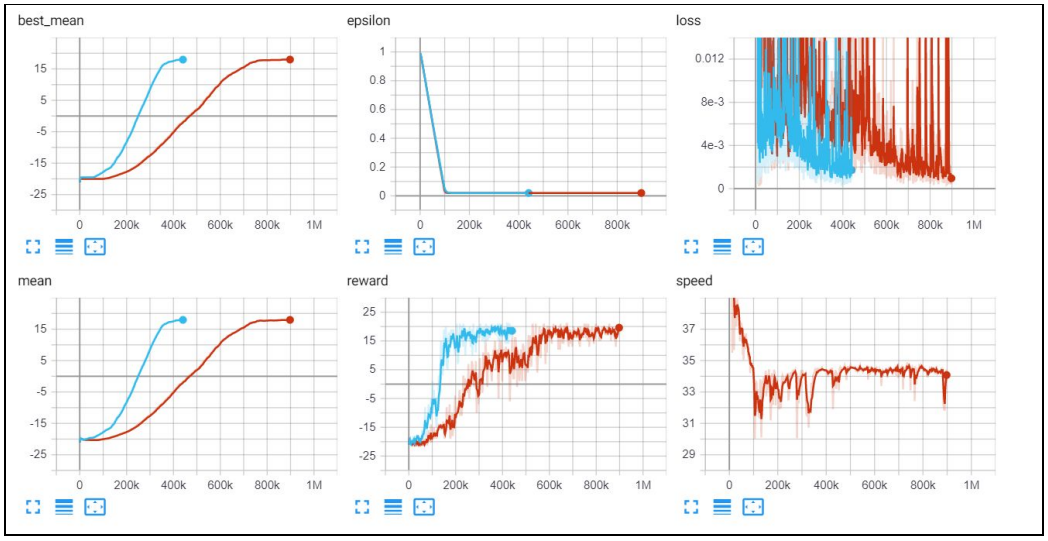
Vanilla DQN with Huber loss without Kaiming normal initialisation

As seen from the graph, DQN with Huber loss converged earlier to a mean of 18 in ~850k frames while DQN with MSE loss converged to the same mean in 900k frames. As Pong is a simple game, it likely doesn’t suffer from exploding gradients and thus Huber should not make a large impact on it. However, we can see in the graphs that Huber loss predicted the loss to be lower than that of MSE. Huber loss also had a steeper increase in reward.

Neural network initialisation

The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. We changed the default initializer for Conv2d and linear layer from Kaiming Uniform to Kaiming Normal. This method initializes each layer’s weight drawn from a normal distribution with the same variance and incremental mean down the sequence. Although the mathematics for the improvement is still a research topic, the difference is clear in the graph. Our hypothesis is

that the image pixels intensities as well as extracted features by nature are more closely resembled spatially with a normal distribution rather than uniform distribution.



Legend:

Vanilla DQN with MSE loss without Kaiming normal initialisation

Vanilla DQN with MSE loss with Kaiming normal initialisation

Vanilla DQN with MSE less with Kaiming initialisation converged to a mean of 18 in ~450k frames while vanilla DQN with MSE without Kaiming initialisation converged in about ~900k frames.

Extensions

Double DQN

A simple yet effective extension to DQN was proposed in the 2015 paper **Deep Reinforcement Learning with Double Q-learning**[4]. The authors of the paper noticed that the basic DQN has a tendency to overestimate values for Q, which can increase convergence time or even decrease the quality of the final policy. In an attempt to solve this problem, the authors proposed modifying the Bellman update. For the basic DQN, the target value for Q is given by:

$$Q(s_t, a_t) = r_t + \gamma max_a(Q'(s_{t+1}, a_{t+1}))$$

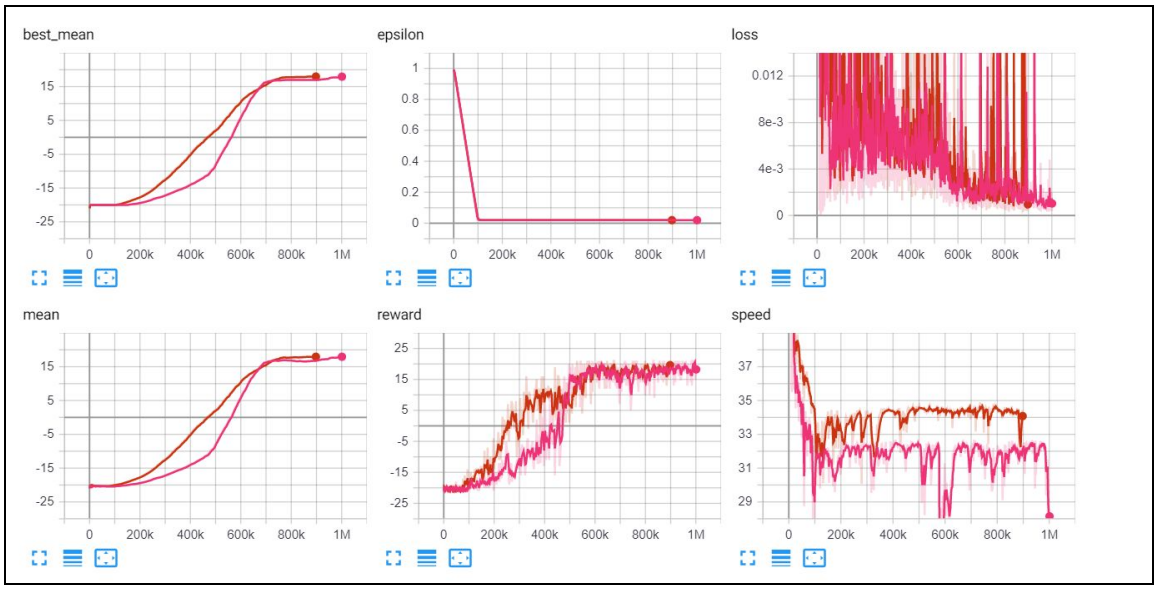
The authors thus proposed to instead choose actions for the next state using the trained network, but taking Q value estimations from the target network. The new expression for Q values is thus:

$$Q(s_t, a_t) = r_t + \gamma max_a(Q'(s_{t+1}, argmax_a(Q(s_{t+1}, a))))$$

This change supposedly reduces overestimation of Q values and thus theoretically increases convergence time and increases the quality of the final policy.

[4] Van Hasselt, H et al. (2015) Google Deepmind

Running vanilla DQN with mean squared error loss and no sophisticated neural network initialiser with and without this double DQN change gave the following results:



Legend:

Vanilla DQN with MSE loss without Kaiming normal initialisation

Double DQN with MSE loss without Kaiming normal initialisation

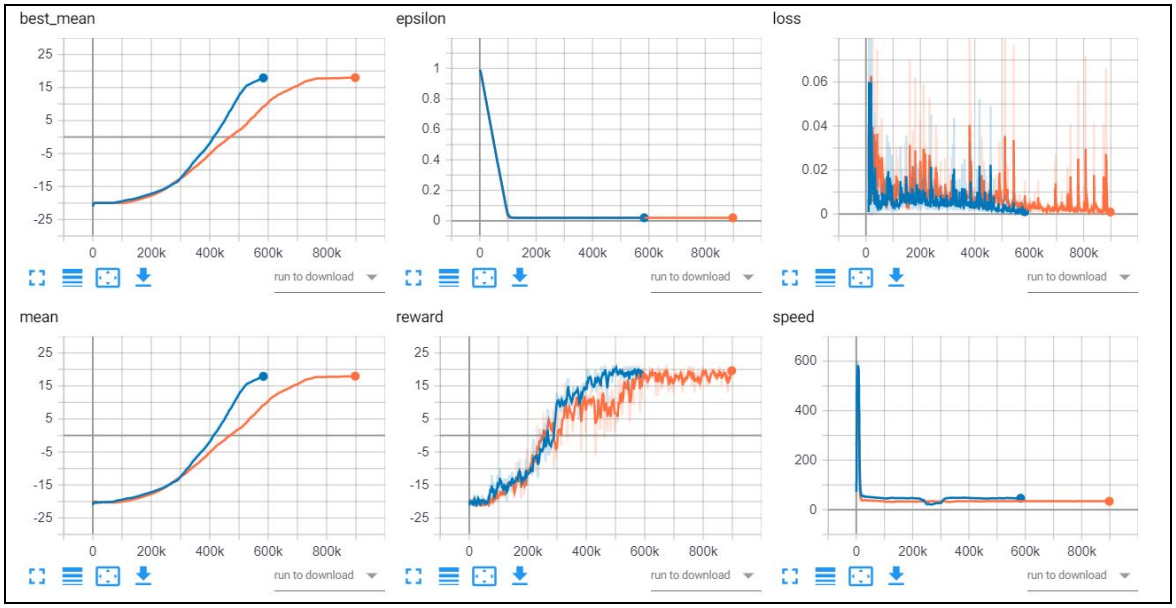
As shown in the graphs, Vanilla DQN reached a mean of 18 in ~900k frames while the double DQN converged to the same mean in ~1 million frames. Double DQN took longer to converge and had a slower fps speed. We believe the reason for this is that double DQN takes too much overhead for a simple game such as Pong and thus we did not see the expected theoretical increase in convergence time in our experiments.

Dueling DQN

The next improvement to DQN was proposed in the 2015 paper **Dueling Network Architectures for Deep Reinforcement Learning**[5]. The key idea behind this extension is to do with the fact that Q values $Q(s,a)$ can be broken down into **value of state** $V(s)$, and **advantage of actions** in this state $A(s,a)$[6].

The main gain from this extension is that in many states, it is unnecessary to estimate the value of each action choice, as no matter what action you choose it has no repercussion on what happens. This should improve convergence as now the dueling architecture can learn which states are valuable without having to learn the effect of each action for each state. The results of using Dueling DQN on Pong are as follows:

[5] Wang Z, Schaul T, Hessel M, van Hasselt, H, Lanctot M, de Freitas N (2015).
[6] See appendix for a more detailed explanation on this extension.



Legend:

Vanilla DQN with MSE loss without Kaiming normal initialisation

Dueling DQN with MSE loss without Kaiming normal initialisation

With dueling the network converged to a mean score of 18 in ~580K frames, while the vanilla DQN converged to a mean score of 18 in ~900K frames. We theorise that, in Pong, when the ball is travelling towards your opponent, the actions you perform while waiting for the ball to bounce back don't have much impact. This may explain why dueling sped up the convergence time.

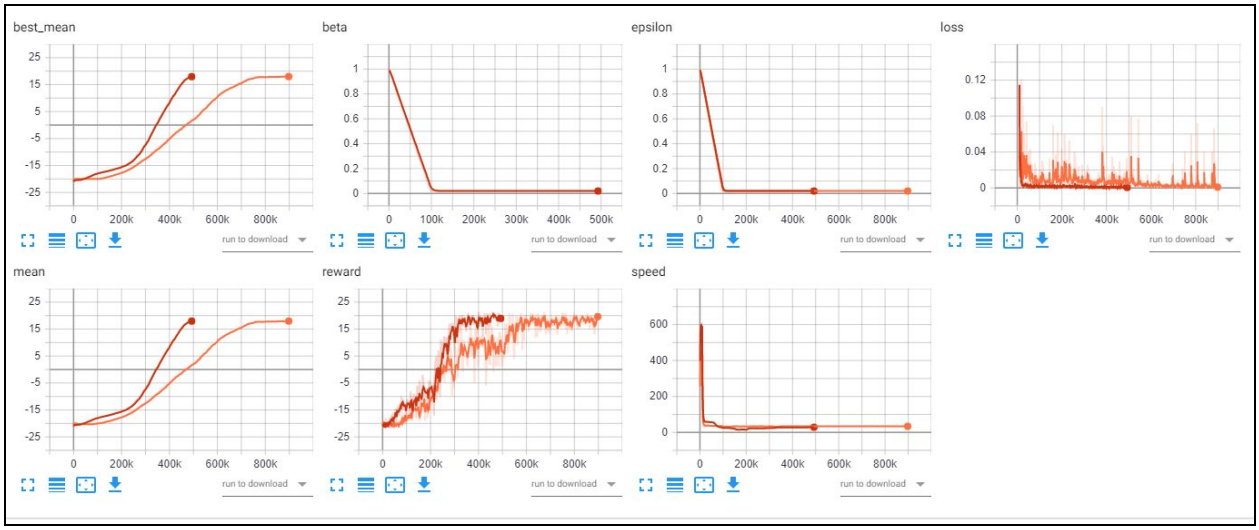
Priority Experience Buffer

The next improvement to DQN was originally proposed in the 2015 paper **Prioritized Experience Replay**[7].

The authors of the paper proposed that assigning priorities to certain experiences according to training loss and sampling the buffer based on those priorities would improve convergence time and increase the quality of the final policy. Essentially, the idea is to sample the transitions from which there is more to learn, and are more interesting and useful than most[8].

For this implementation α was set to 0.6, and β was set to 0.4, increasing linearly until 100000 frames, at which point it would be 1, based on the recommendations in the paper. Running the Pong DQN with only this PEB as an extension gave the following results:

[7] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver (2015).
[8] See appendix for a more detailed explanation on this extension.



Legend:

Vanilla DQN with MSE loss without Kaiming normal initialisation

DQN with Priority Experience Buffer with MSE loss without Kaiming normal initialisation

With the PEB, the network converged to a mean score of 18 in ~500K frames, while the vanilla DQN converged to a mean score of 18 in ~900K frames, so prioritising experiences is a clear improvement.

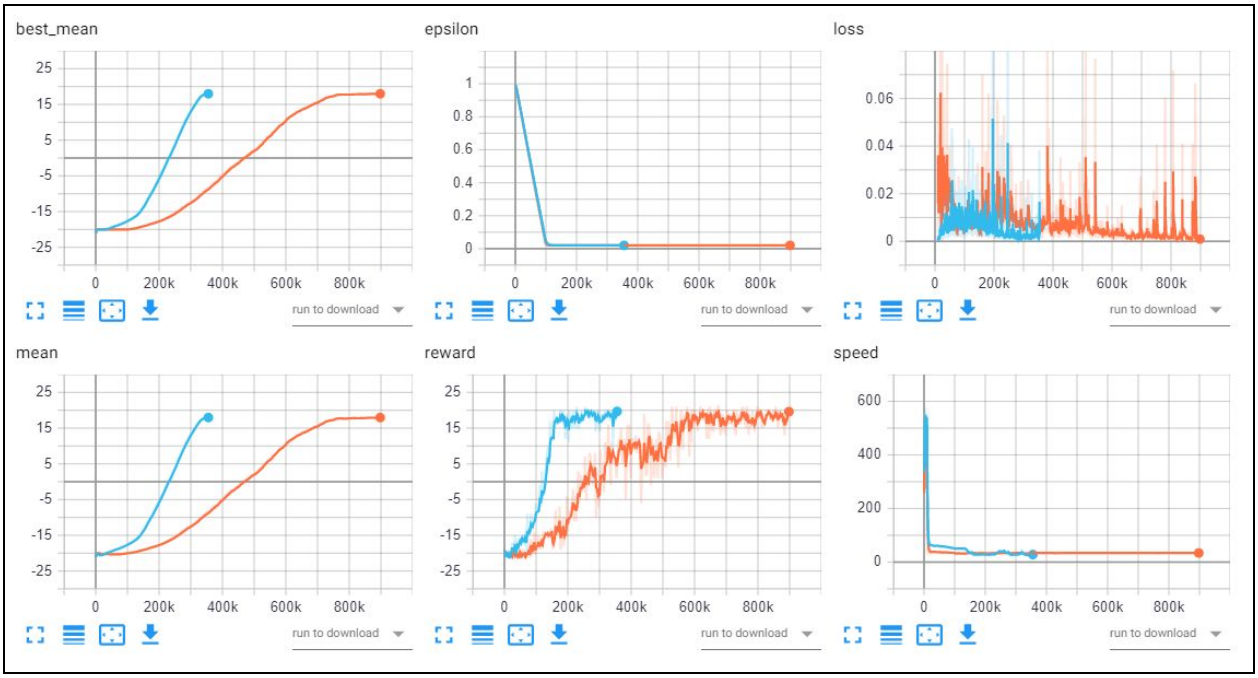
Image Feature Extractor

We used the Canny Edge detector from OpenCV to extract edges from all the frames. The Canny Edge detector finds edges in images by calculating gradient change in pixel intensity in X and Y direction with a preset of cutoff threshold to filter noise. With a vanilla DQN the purpose of CNN is to extract all the useful features from the input frames.

In a pure computer vision supervised learning environment, the role of CNN is clear as it learns to extract important features for categorization. However, in our context what is back propagated through the network is not purely features but Q values as well. Having the DQN learn from the manually extracted features is essentially dimensionality reduction on input data and it improves the learning speed significantly.

Canny Edge Detector on Breakout

Below are results obtained with the canny edge detector.



Legend:

Vanilla DQN with MSE loss without Kaiming normal initialisation

DQN with the Canny edge detector wrapper with MSE loss without Kaiming normal initialisation

With the Canny edge detector wrapper, the program converged to a mean score of 18 in ~350K frames, while vanilla DQN converged to a mean score of 18 in ~900K frames. Clearly this wrapper helped to remove the unnecessary “noise” in the environment and encouraged the network to focus on the important details, resulting in a remarkable improvement in convergence rate.

We also experimented with the Gunnar Farneback optical flow as an attempt to extract better motion information However, due to the heavy computation required, it reduced the training frame rate on vanilla significantly from 40fps to 20fps. Hence, no further development was taken.

ResNet-18

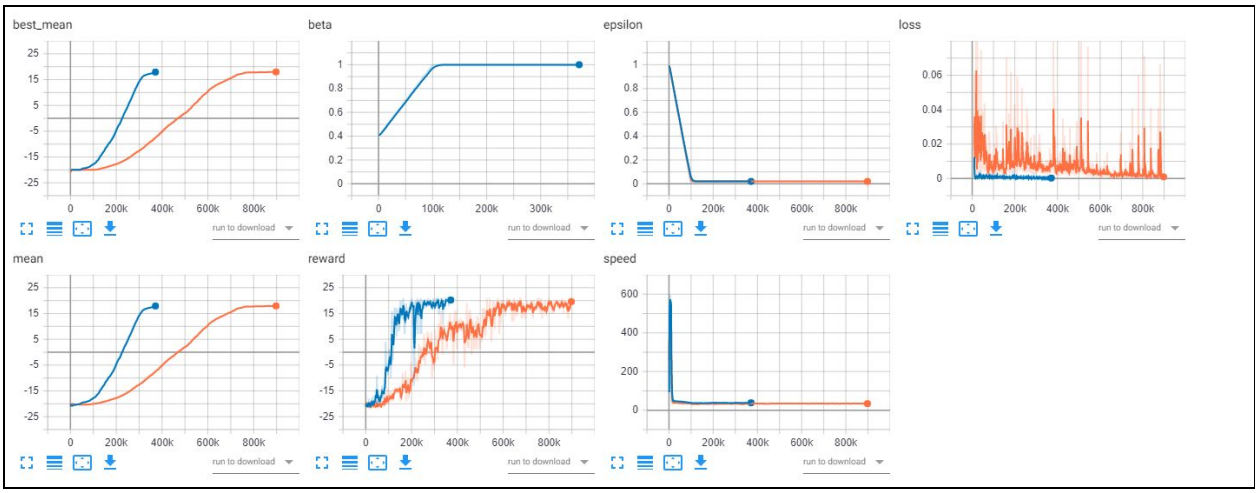
As the initial goal is for the agent to learn with human perception, we considered using a small ResNet as the convolution layer to extract features from the network. However, after some research and testing, it is found that numerous successful implementations did not use such a large network. Further, framerate was dropped to 18fps due to the comparatively large amount of parameters to train for in 18 layers. The idea of using ResNet was thus abandoned and we considered extracting features manually via image processing techniques.

Results

To conclude, we were able to successfully collect results for both Pong and Breakout which are supported by papers and previous research.

After running Pong with different extensions and hyperparameters, it becomes clear that the Pong game converges fastest with all extensions as well as Huber loss and Kaiming normal initialisation. The DQN with all

extensions, Huber loss and with Kaiming normal initialisation converged to a mean score of 18 in ~370k frames while the vanilla DQN with MSE loss and without initialisation converged to the same mean in ~900k frames. This can be seen in the graph below:

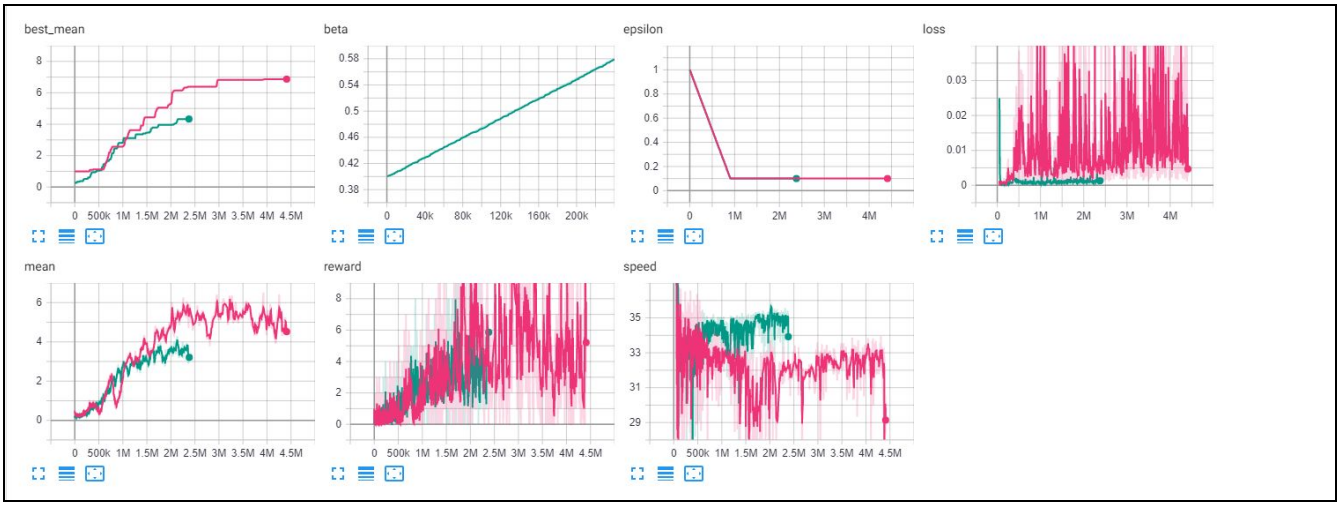


Legend:

Vanilla DQN with MSE loss and without Kaiming normal initialisation

DQN with all extensions with Huber loss and with Kaiming normal initialisation

For Breakout, we were unable to have the network converge as it is quite complex and would require a large amount of frames and time to converge. Deepmind trained their agents for ~200M frames to obtain their results. However, we were able to start the game with and without Pong-proven extensions and observe that, at least initially, vanilla DQN without extensions performed better. This can be seen in the graph below:



Legend:

Vanilla DQN with MSE loss and without Kaiming normal initialisation

DQN with all extensions with Huber loss and with Kaiming normal initialisation

The application of extensions had a directly positive impact on the convergence rate of the network for Pong, however we did not observe much improvement when we applied these to Breakout. We came up with a few hypotheses on why this might be the case.

It may be that the extensions are unhelpful for Breakout and introduce unnecessary overhead. It may also be that adding extensions makes convergence slower at the beginning, but extensions could eventually have a better policy. Further, since we experimented with tuning hyperparameters on Pong and then applied these to Breakout, it may be the case that there is not a 1:1 translation between combinations of parameters and extensions that are useful on both games. Overall we note that successful tuning for one game does not immediately generate success in more complex games, even if we have a prior belief that the two games would be similar to learn.

Conclusion

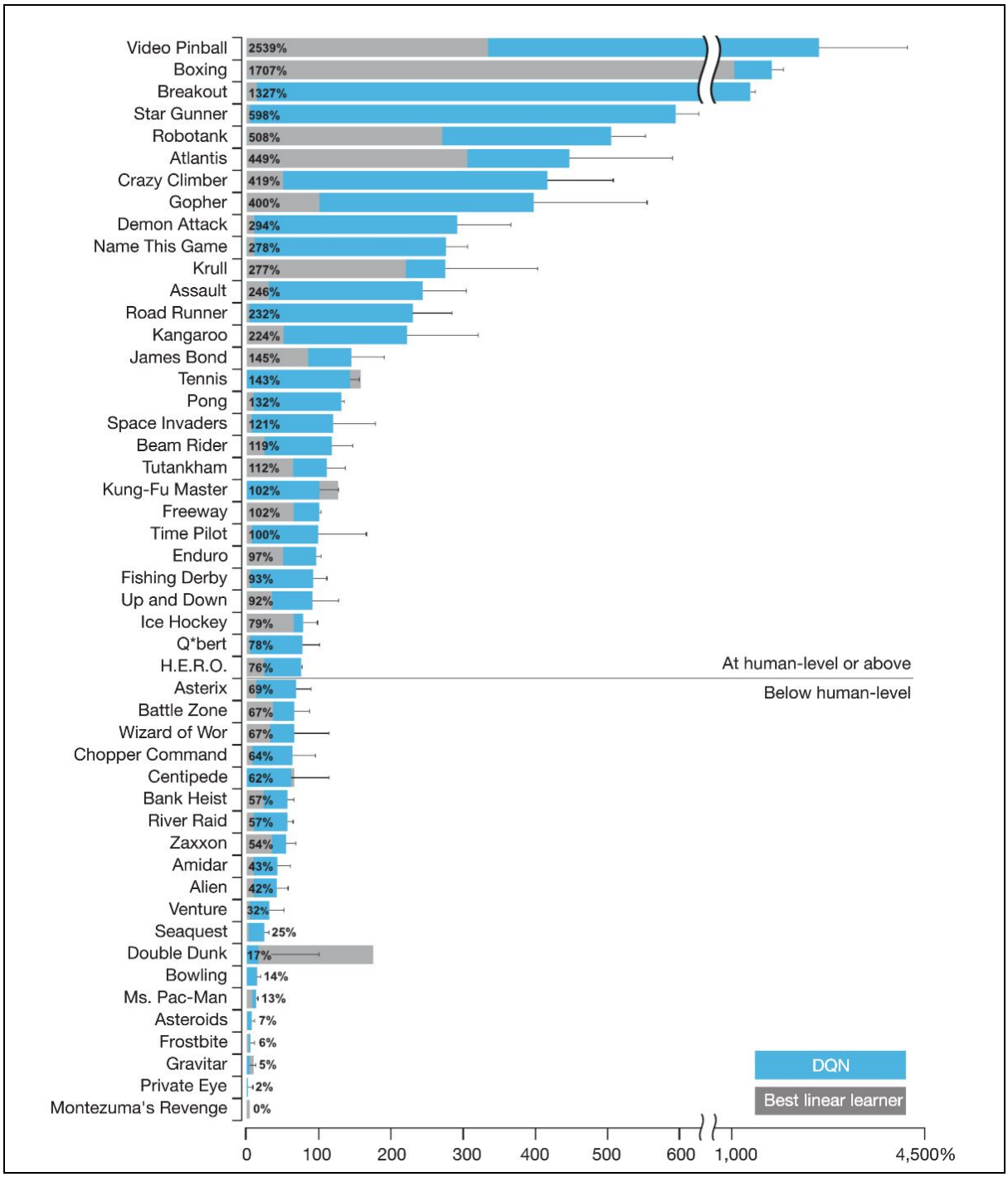
This project was completed in order to determine the benefits of different deep Q-learning extensions and the direct impacts of tuning hyperparameters on the learning of the Pong and Breakout Atari 2600 games. Results were achieved that showed the benefits of extensions in general and the impacts that can be made on convergence time.

While unfortunately we could not significantly improve the convergence rate for Breakout, we believe the success on Pong indicates that the extensions we implemented are likely to improve the convergence rate of other Atari games. We further believe that working specifically to tune our model on just Breakout would yield good results given enough time. This project shows that it is indeed possible to improve the learning for different Atari games and was successful in its findings.

References

- Fortunato, M et al. (2019) Noisy Networks for Exploration, Google DeepMind.
- Lapan, M. (2018) Deep Reinforcement Learning Hands-On, Packt Publishing.
- Marlos, C et al. (2015) Arcade Learning Environment Technical Manual (v.0.5.1), <https://github.com/openai/atari-py/blob/master/doc/manual/manual.pdf>
- Mnih, V., Kavukcuoglu, K., Silver, D. et al (2015) Human-level control through deep reinforcement learning. Nature 518, 529–533. <https://doi.org/10.1038/nature14236>
- Schaul, T et al. (2015) Prioritized Experience Replay. <https://arxiv.org/abs/1511.05952>
- Van Hasselt, H et al. (2015) Deep Reinforcement Learning with Double Q-learning, Google DeepMind.
- Wang, Z et al. (2015) Dueling Network Architectures for Deep Reinforcement Learning, Google DeepMind, London, UK.

Appendix



Performance of DQN algorithm on Atari games as a percentage of human performance
Image credit: [Human-level control through deep reinforcement learning paper]

Wrappers Full List

Pong wrappers used:

- **ProcessFrame84 Wrapper**: We scale every frame down from 210 x 160 with RGB channels into a greyscale 84 x 84 image. We also crop non-relevant parts of the image. This is to reduce passing irrelevant information to the neural net and hence improving convergence rate.
- **ImageToPyTorch wrapper:** We change the shape of our observations from HWC to CHW format as required by PyTorch.
- **BufferWrapper Wrapper:** We create a stack of subsequent frames along the first dimension and return them as an observation. The purpose of this is to give the network an idea about the dynamics of objects in the board, such as their speed and direction of movement.
- **FireResetEnv Wrapper:** We use this wrapper to press the Fire button at the beginning of the game. This is necessary so that the net does not have to learn how to do it itself. If the net was to learn this action it would require more episodes to be played.
- **MaxAndSkipEnv Wrapper:** We skip every 4 frames as well as taking the maximum of two consecutive frames. Since each frame is similar to each other, skipping of every 4 frames does not hinder learning and speeds up the process. The reason that the max of two consecutive frames is taken is to account for flickering graphics in Atari games, this ensures that the objects in the frame are present.
- **ClipRewardEnv Wrapper:** We clip the reward to a value in {+1, 0, -1} by its sign. This is done so that hyperparameters can be generalised for a set of games where the reward bounds are different.
- **ScaledFloatFrame Wrapper:** We scale all observation data from bytes to floats and scale every pixel value to one between 0 and 1. This is the preferred form of representation for a neural network as it makes calculations easier to perform and scale things down.

Breakout wrappers: identical to Pong, with the addition of:

- **EpisodicLifeEnv Wrapper:** This wrapper makes the loss of a life be equivalent to the end of an episode. It helps with value estimation and speeds up the convergence of episodes.
- **NoopResetEnv Wrapper:** This wrapper samples initial states by taking 30 no-ops (do nothing action) on reset. This stabilises training and helps speed up convergence.

Hyperparameters Full List

Hyperparameters used with all vanilla, double and dueling DQN:

- **REWARD_BOUND:** The game-specific reward we try to reach. If the algorithm scores this average over its past games, the game is considered ‘solved’.
- **REWARD_EPISODES:** The number of past games (episodes) over which we average previous scores to determine whether the game is ‘solved’.
- **GAMMA:** The discount factor of reward used to compute expected future reward in the Q-Learning algorithm.
- **BATCH_SIZE:** The number of observations we sample from the experience buffer periodically.
- **REPLAY_SIZE:** The size of the replay buffer.
- **LEARNING_RATE:** Parameter used in gradient descent to periodically update the neural network weights and biases.
- **SYNC_TARGET_FRAMES:** Number of frames which pass between synchronising our main network and target network.
- **REPLAY_START_SIZE:** Amount of frames which pass before the algorithm begins to look at the replay buffer to learn.
- **EPSILON_DECAY_LAST_FRAME:** Amount of frames beyond which epsilon is constant.

- **EPSILON_START:** Starting value of epsilon (generally 1.0).
- **EPSILON_FINAL:** Long run value of epsilon.

Hyperparameters used on the Priority Experience Buffer:

- **ALPHA_PRIORITY:** How much emphasis we give to priority, a value of 0 means uniform sampling
- **BETA_START:** Starting value of beta (generally 0.4)
- **BETA_FRAMES:** Number of frames after which which beta will be 1

Dueling DQN Explanation

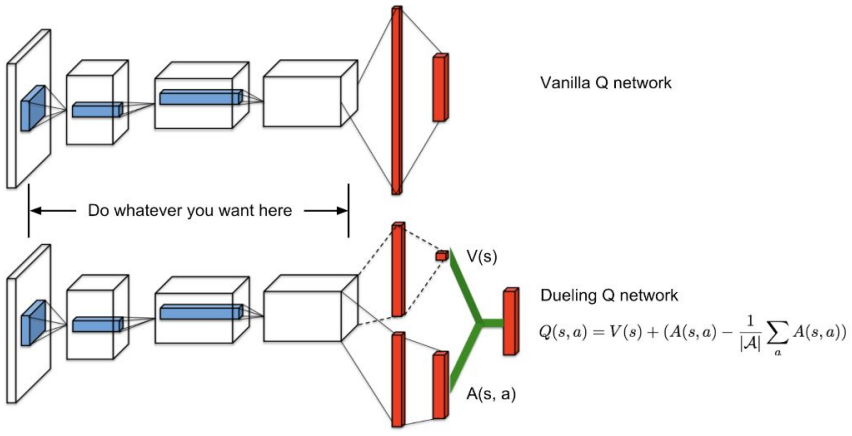
The key idea behind this extension is to do with the fact that Q values $Q(s,a)$ can be broken down into **value of state** $V(s)$, and **advantage of actions** in this state $A(s,a)$.

By definition:

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

And thus the advantage can be seen as the value of how much extra reward an action a can give us when in the state s. Note that advantage can be positive or negative, reflecting whether the action is or isn't beneficial to increasing reward.

While a traditional vanilla DQN takes features from the convolutional layer of the neural network and transforms them directly into Q values for each possible action given a state (pixels on a screen), the paper's idea is to instead process the output of the convolutional layer through two independent paths, one path responsible for the estimation of value of state $V(s)$ (a single number), and another path to predict individual advantage values $A(s,a)$ for each action. The values of state and advantage of actions are then combined to give an estimation for the Q values, which is used and trained as normal.



Original image from Wang, Z et al. Text added by:
https://theaisummer.com/Taking_Deep_Q_Networks_a_step_further/

Now to aggregate these two values, it is tempting to merely sum the two values. However in practice this is not very effective, as it is problematic to assume that $V(s)$ and $A(s,a)$ are good estimates of state value and action advantages, and the sum of the two is unidentifiable in that given a Q value we cannot reconstruct V and A uniquely. Thus, to solve the identifiability issue, the author uses the equation:

$$Q(s,a) = V(s,a) + (A(s,a) - argmax_{a'} A(s,a'))$$

To force the Q value for the maximising action to equal V.

However from the author's experiments, the max operator can be replaced with an average to force the mean to 0 and achieve just as good practical results and increased optimisation stability:

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

Where N is the number of actions, and thus this is the equation we use in our implementation.

Priority Experience Buffer Explanation

In most Atari games, a lot of experiences are highly correlated, as the environment doesn't change much according to our actions. This is problematic as stochastic gradient descent (SGD) relies on the assumption the data has the independent and identically distributed (i.i.d) property. The basic DQN uses a large replay buffer of experiences randomly sampled to try and attain this i.i.d property as much as possible.

The authors of the paper proposed that assigning priorities to certain experiences according to training loss and sampling the buffer based on those priorities would improve convergence time and increase the quality of the final policy. Essentially, the idea is to sample the transitions from which there is more to learn, and are more interesting and useful than most.

The priority of each experience is proportional to the training loss of that experience. The probability of sampling an experience i is given by:

$$P(i) = \frac{{p_i}^{\alpha}}{\sum_k {p_k}^{\alpha}}$$

Where p_i is the priority of experience i, and α is a hyperparameter that is effectively how much emphasis we give to the priority. If α is 0, our sampling would be uniform again.

New samples are also given the maximum priority to ensure they'll be sampled soon, however this introduces a bias in the data distribution that must be accounted for for SGD to work. To solve this, the authors multiplied the individual sample loss by sample weights. The value of weight for each sample is given by:

$$w_i = (N * P(i))^{-\beta}$$

Where N is the size of the buffer, $P(i)$ is the probability of sampling an experience i and β is a hyperparameter that again effectively measures how much prioritisation to apply, and usually starts from 0.4 and increases linearly to 1 overtime due to training usually being very unstable near the beginning.