Contents

Abstract 0

Introduction 1.0

- Reinforcement learning background & theory 1.1
- Historic approaches to state of the art 1.2
- Proposed solution and outline 1.3

Objectives & Goals 2

Proposition and motivation 3

Methodology and implementational details 4

Results 5

Conclusion 6

0. Abstract

In recent years we have seen a resurgence in artificial intelligence (AI) owing to a combination of availability to large amounts of compute, large amounts of data and neural networks. Reinforcement learning is a subsection of artificial intelligence and it too has benefited greatly, notable examples being, beating the world champion of the game Go [0] and beating the best chess and shogi engines[1]. All with little to no input from humans.

These accomplishments however have required large amounts of compute and data, mainly from simulators, with poor performance when the agent is required to perform in unfamiliar situations. For AI and reinforcement learning to become more useful in solving complex problems where large amounts of data or compute may not be available, it is important that we address these issues. In this work we attempt to do this by first looking at reinforcement learning theory and doing a literature search over current approaches to propose modifications to current state of the art. Specifically we modify the well-known DQN algorithm by introducing non-linear parametric function approximators, with convolutional auto-encoders . We compare this modified approach with the original to see if these problems are addressed.

1. Introduction

1.1 – Reinforcement learning background & theory

Reinforcement learning is a subset of machine learning and is used for sequential decision making. Intuitively, it can be seen as an agent that observes the environment and performs some action after which it receives a reward as a scaler value, this repeats at each timestep till infinity or till the agent is terminated. The objective of the agent is to learn the optimal action in each state, what optimal is, we will define later.

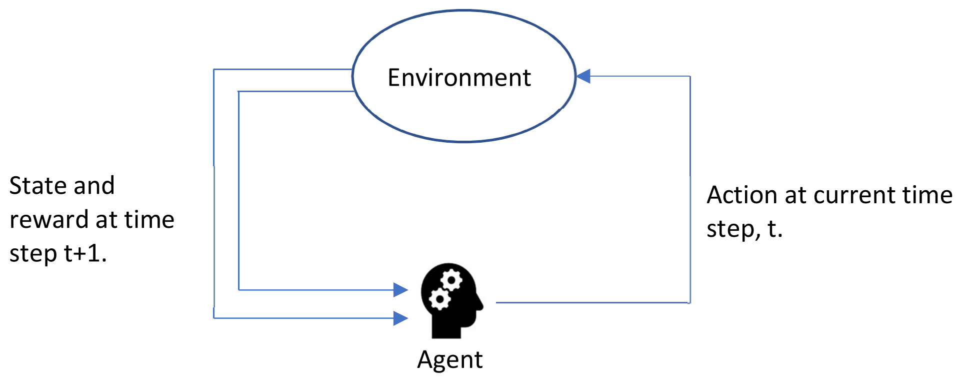

Environment
State and reward at time step t+1.
Action at current time step, t.
Agent

Figure 1 – Diagram outline of usual reinforcement learning flow.

Reinforcement learning is commonly modelled by a Markov decision process (MDP) defined by the tuple $\langle S, A, R, P, \gamma \rangle$

Element	Description
States / S	The set of all possible states the agent can be in, for example every possible position of a robot in a maze. For our purposes we will let it be finite but extensions to infinite state spaces is possible.
Actions / A	The set of all possible action, for example in a grid a robot may have an action space of {up, down, left, right}. For our purposes we will let it be finite but extensions to infinite action spaces is possible.
Reward function / R	A function $E[r_{t+1} \| s_t, a_t]$ which gives the expected reward at the next time step given a state and action at the current time step.
Transition probability / P	A function $P(s_{t+1} \| s_t, a_t)$which gives the probability of reaching the next state s_{t+1} given a state s_t and action a_t. In the real world we can think of this as the physics determining how the world evolves.
Discount factor / γ	This is a value between 0 and 1 that specifies the importance placed on future rewards over more recent rewards, 0 means only consider the reward at the next time step when considering an action, and 1 means consider all rewards till termination. It also prevents numerical instability due to infinite sums of rewards when the agent does not 'terminate'.

Figure 2 – Fundamental components of a reinforcement learning problem.

Using this we can begin to define what it means to act optimally, or in reinforcement learning what is known as finding the optimal policy. A policy $\pi(a|s)$ is a mapping from $S \rightarrow A$, so that for any state we can make an action, it can also be probabilistic. To find the optimal policy we need to be able to compare them, this is done by defining what is know as the value of a state. The better policy has a higher value for each state in the state space. The value of a state is defined as the sum of discounted reward following a given policy from that state till termination or infinity. This leads us to the definition of the value function,

$$V^{\pi}(s_t) = E^{\pi}[r_t + \Upsilon r_{t+1} + \Upsilon^2 r_{t+2} + \dots] = E^{\pi}[r_t + \Upsilon V^{\pi}(s_{t+1})].$$

The superscript π represents that to define a value function we need a policy, the discount factor Υ assures that the series converges for infinite time steps.

Now that an optimal policy can be defined, the question of how to find it arises. In reinforcement learning literature there are 3 main approaches value based, policy based, and actor-critic based. We will focus on value based as that is the focus of this project. In value-based methods for finding the optimal policy a function called the action value function is defined,

$$Q^{\pi}(s_t, a) = E^{a}[r_t] + \Upsilon\, V^{\pi}(s_{t+1}).$$

Qualitatively this tells us the expected reward we will receive if we choose an action *a* then follow our policy after. The goal of value-based methods is to find $Q^{\pi^*}(s, a)$ the action value function corresponding to the optimal policy, once we have this we know how to act optimally by simply doing $\underset{a}{\operatorname{argmax}}\, Q^{\pi^*}(s_t, a)$.

1.2 – History of approaches to state of the art.

Monte Carlo control[2]

The first one we will look at which is also the simplest is called Monte carlo control and is relatively simple. We initialise any random $Q^{\pi}(s, a)$ as a lookup table, then collect data from the environment by letting the agent act according to the policy $\underset{a}{\operatorname{argmax}}\, Q^{\pi}(s_t, a)$. The number of times a state is visited, and an action is taken is recorded in $N(s, a)$ as well as the total reward received (G_t) for following our policy. The difference between the prediction of the expected sum of rewards and the actual received sum of reward (the target) is taken, resulting in the update as follows:

$$Q^{\pi}(s_t, a_t) \;\rightarrow\; Q^{\pi}(s_t, a_t) + \frac{1}{N(s_t, a_t)} * [\,\underbrace{G_t}_{\text{Target}} - \underbrace{Q^{\pi}(s_t, a_t)}_{\text{Prediction}}\,]$$

Repeating this infinitely for every state action pair or alternatively exploring the environments infinitely results in converging to the optimal $Q^{\pi^*}(s_t, a)$. In practice exploration is done by initiating an epsilon greedy policy, this is the same as the policy $\underset{a}{\operatorname{argmax}}\, Q^{\pi}(s_t, a)$ but we have some chance ε of taking a random action, this starts at 1 which is completely random and linearly decreases to a value close to 0, usually 0.1, which guarantees in the limit as the time steps approaches infinity we will visit every state an infinite number of times. While this exploration strategy can be substituted for other types it is generally used for its simplicity and because it is not so easy to beat, surprisingly.

SARSA[2]

This is identical to monte carlo control but instead the update is as follows:

$$Q^{\pi}(s_t, a) \;\rightarrow\; Q^{\pi}(s_t, a) + \;\alpha * [\,\underbrace{r_t \;+\; \gamma Q^{\pi}(s_{t+1}, a_{t+1})}_{\text{Target}} - \;\underbrace{Q^{\pi}(s_t, a_t)}_{\text{Prediction}}\,]$$

The difference is that instead of G_t as our target which is the true sum of rewards received, we substitute it for $r_t \;+\; \gamma Q^{\pi}(s_{t+1}, a_{t+1})$ which is our prediction for the sum of rewards received. The next state s_{t+1} is derived from the same epsilon greedy policy we mentioned earlier, and the alpha is a constant analogous to the learning rate in something like gradient descent. Again, if we perform this update infinitely for every state action pair we are guaranteed to converge to the optimal $Q^{\pi^*}(s_t, a)$.

Q Learning[3]

The methods so far have been all been on-policy meaning that to learn the best policy we needed data (G_t for Monte Carlo control and r and $s_{t+1}$for SARSA) that has been generated using the agents own policy, meaning it cannot use data generated from other agents' 'behaviour' policies. It is important to be able to do this since we may have data generated from an expert that we want our agent to learn like a teacher and student, or data from the past under a different policy that we might want to make use of. Q learning allows for this by modifying the update like so:

$$Q^{\pi}(s_t, a) \rightarrow Q^{\pi}(s_t, a) + \alpha * [\underbrace{r_t + \gamma \max_{a_{t+1}} Q^{\pi}(s_{t+1}, a_{t+1})}_{\text{Target}} - \underbrace{Q^{\pi}(s_t, a_t)}_{\text{Prediction}}]$$

The key difference is that now in the target we must choose an action that maximises the action value function from where the behaviour policy took us, instead of where in SARSA where we used action value of the state-action pair that we arrived in due to our policy. Like the previous approaches, this is guaranteed to converge to the optimal action-value function provided we explore each state-action pair infinitely many times.

Function approximation and DQN

The approaches that we have been looking are guaranteed to converge for look up tables where we simply store the Q value for every state-action pair, this might work well for small action and state spaces like in a small gird world, where we can store this in memory and reasonably explore all states. However, most interesting problems have state-action spaces that are too large to be stored in memory, for this reason the action value function is commonly represented as a function approximator, $Q(s_t\, , a_t\, ; w)$ where w are the function approximators parameters. This makes it so that we do not have to explicitly store every single state-action pair.

For this project we focus on two types of function approximators, linear function approximators and neural network function approximators. Linear function approximators approximate the value function by using a weighted sum of basis functions often referred to as features.

$$Q^{\pi}(s_t, a) = \theta(s_t, a) \cdot \omega$$

The features are a function of the input state and are each weighted by a weight. The Q value is then a dot product of the vector of basis functions and vector of weights. Many basis already exist in the literature e.g. RBF [4], PVF [5], Fourier [6].

The second type of function approximator is the neural network, neural network function approximators are a parametric nonlinear approach to function approximation. They have recently shown state of the art results in some challenging tasks with very large state spaces [7], [8], [9] most of these approaches are modifications of the DQN architecture [10].

We can extend function approximators naturally to the approaches discussed before, where now the updates from before are now extended to a loss function which is commonly the squared loss between the prediction and target. An example is the loss function for Q learning is as follows:

$$L = \left(\underbrace{r + \gamma \max_{a_{t+1}} Q(s_{t+1}\, , a_{t+1}; w\,)}_{\text{Target}} - \underbrace{Q(s_t\, , a_t\, ; w)}_{\text{Prediction}}\right)^2$$

Which if we used gradient descent to minimise results in updating the function parameters as follows:

$$\Delta w = \alpha\left(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; w) - Q(s_t, a_t; w)\right)\nabla_w Q(s_t, a_t; w)$$

DQN which is a state-of-the-art approach uses these exact updates together with Q learning and a neural network function approximator with two key modifications.

1) We have two neural network function approximators, one to generate the targets, the target network and the initial one for acting in the environment. We update the initial network every step as we do in normal Q learning, and every C steps we update the target network parameters to be the same as the initial network parameters. This stops the network from diverging towards infinite values by stopping the network from bootstrapping off bad predictions. This changes the gradient descent update to this:

$$\Delta w = \alpha\left(r_t + \gamma \max_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}; w_{target}\right) - Q(s_t, a_t; w_{action})\right)\nabla_w Q(s_t, a_t; w_{action})$$

2) Instead of using the data the agent receives immediately (s_t , a_t, r_{t+1}, s_{t+1}) , the data is stored in memory and we randomly sample batches of a particular size (usually 32 or 64) and update the parameters using this data. This decorrelates the samples used to train sequentially and decreases variance, making training more efficient.

Together with these modifications state of the art results is achieved on high dimensional problems, such as with the Atari games, gaining super human performance through simply playing the game, showing these algorithms can extend to high dimensional states (7056 dimensional for Atari for 84x84 input image) with a state space of size 256^{84} if each pixel is a grey scale 8-bit value.

1.3 – Upcoming sections outline.

While this does well it has its problems, such as requiring large amounts of data (roughly 10 million frames or 46 hours' worth of real world time for simple games), not transferring to situations that have not been experienced during training and the need for large amounts of compute and memory. It is clear we need to modify the current approach to address the issues mentioned, which is what we will attempt in this report.

The coming sections will be as follows:

Section 2 - Describe the objectives & goals and how they changed over time.

Section 3 - Describe how the proposal to meet the goals was arrived at.

Section 4 – Methodology, implementational details and problems encountered.

Section 5 – Results.

Section 5 – Conclusion.

2. Objective and goals

The objectives and goals stem from knowing the disadvantages of DQN and trying to remedy them. Namely the large data required, the low performance on states that have not been observed much (transfer learning), and the requirement of large amounts of compute and memory .

Initially, through the first term, by looking through the reinforcement learning literature, the main goal was to improve on the ability to perform better in unfamiliar scenarios while maintaining performance and see if the 'end to end' (described in figure 3) approach of DQN is ultimately superior to that of ours. During the second term, which was mainly focused with the implementation of the algorithms, due to the lengthy time required to get results as well implementation difficulties it was decided to leave that for future work. Thus, the final goals were as follows:

1) Try a different approach to current state of the art methods, that addresses current short comings and is motivated by literature.
2) Implement the approach.
3) Compare the two approaches.
   -Which one performs better?
   -Which one is more data efficient?
   -Is the end to end approach of DQN superior to ours?

3. Proposal and motivation

The proposal that was made was to move from the end to end approach of DQN to one of feature extractor with function approximator. The function approximator used is a convolutional-autoencoder[11] this is a neural network that learns lower dimensional embeddings off the data, by trying to reconstruct the data from the lower dimension. It consists of an encoder, which projects the input data into a lower dimensional space, and a decoder which tries to reconstruct the data embedded in the low dimensional space. Through this training we can use the encoder by itself for dimensionality reduction.

We use a convolutional autoencoder and not a regular one that consists of fully connected layers because the convolutional layers play a big role in the success of DQN, we can see this from looking at [12] which is a work prior to DQN, that used regular neural networks with fully connected layers instead with not as much success as DQN suggesting the convolutional layers played a large role in the success of DQN.

This makes sense since convolutional autoencoders are extremely good at extracting features from images shown extensively in the literature through state-of-the-art performance in computer vision tasks[13][14]. And the high dimensional state environments that DQN performs state of the art in are image based. While we will focus on image based now for testing there are aims to generalise convolutional neural networks and convolutional auto-encoders to non-image data i.e. convolutions on graphs[15], so our approach can be extended to non-image data. Since a convolutional auto-encoder is essentially two convolutional neural networks facing each other we will use the same architecture as used in the DQN paper for the encoder, and simply reverse this for the decoder.

The choice of the function approximator was a simple linear function approximator and a single layer neural network and was more difficult to decide on as there was no clear-cut way of choosing as with the convolutional auto-encoder which is decidedly state of the art in its role. The choice was made to allow us to test the performance of a simple function approximator versus a more complex one, to see if we really need the extra expressive power provided by the fully connect network. We could choose others but due to the lengthy time it takes to run experiments we only consider these two.

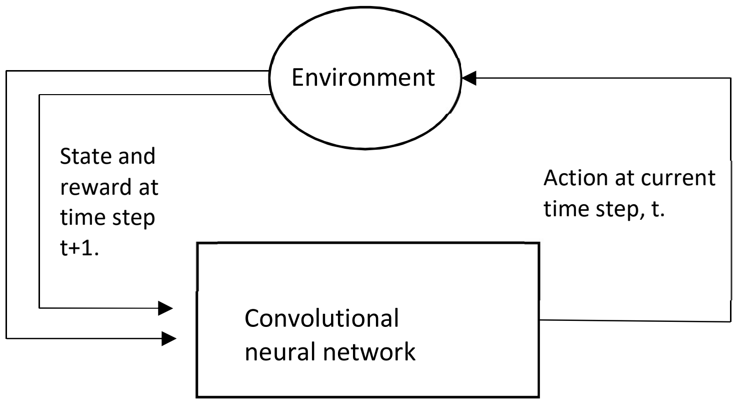

Environment
State and reward at time step t+1.
Action at current time step, t.
Convolutional neural network

Figure 3 - DQN Architecture, end to end approach action out state in, all in the same network.

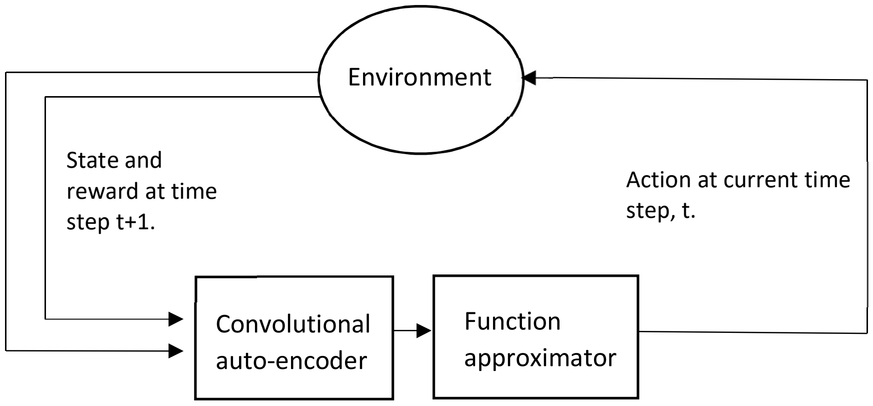

Environment
State and reward at time step t+1.
Action at current time step, t.
Convolutional auto-encoder
Function approximator

Figure 4 - Proposed architecture, feature extraction + function approximator.

The initial intuition behind the approach was to address the weak ability to transfer to unseen environments in DQN, in which the convolutional neural network learns the value function through the sequence of rewards that it gains when the agent is active in the environment. The representations learned by the neural network will then be ones that are dependent on state and the reward received at that state. This decreases transfer learning ability since when it is in a different environment or a state that has never been seen, relationships between reward and states will be different and so the representation that it has built will need to be relearnt.

The difference in our approach is that we learn a representation that is independent of the reward through simply reconstructing the state and then learn a function approximation using these representations. The motivation is that the representation learnt will only be dependent on the state and so in a different domain that shares states that are similar, the function approximator that is learnt will exhibit transfer learning since the function is based on features constructed from only the information about states.

There is also motivation in the literature where [16] does similar work and a data efficiency of two orders of magnitude is reported. However, the final performance of the approach is only 25% of DQN and the transfer learning capability is not tested. Similar work is presented in [17] where the results showed that when the agent is transferred to a different domain there is a 2.7x median improvement in performance compared to without the feature extraction when no extra training is done. Evidence for greater efficiency in learning is also shown.

In addition to these advantages our proposed architecture is much more flexible, in that the convolutional auto-encoder can be used in many ways. For example, in [18], the decoder section of the auto-encoder is used to generate data by itself with no environment needed and learn from that generated data, the authors say they 'train their agent entirely inside of its own hallucinated dream generated by its world model'. The convolutional auto-encoder has many other areas that is used, one used in industry is for real time ray tracing by Nvidia[19] in their new RTX GPU's. The combination with reinforcement learning could allow automated agents to manoeuvre the environment to help collect data from the game as well as train the auto-encoder at the same time.

4. Methodology and implementational details

We will use python for implementation and testing. We use the PyTorch module [20] for implementing the neural networks and training them. The environment which to test is the Atari breakout game from the OpenAI gym module [21]. This game is chosen as it is one of the games that is commonly used to benchmark RL algorithms.

Some prepossessing is done on the image that is returned by the environment which gives us a 3x110x84 RGB frame at 60fps. We will convert these frames to black and white using the ITU-R 601-2 luma transform: $L = R * \frac{299}{1000} + G * \frac{587}{1000} + B * \frac{114}{1000}$ since the colour does not play a role in getting rewards in the environment and it significantly lowers computational time. The image is also cropped to 84x84 to remove the borders in the image, resulting in a final frame of size 1x84x84. A single state is then defined as four of these frames stacked together, to have dimension 4x84x84. This is done in the original DQN paper and in almost every other approach, the reasoning is that this allows the dynamics and movement in the game to be encoded in the state.

The approach used above is done when we implement DQN, but we add an extra step in our case, after converting the image to grey scale we normalize between 0 to 1 by dividing by 256, since the image is 8 bit, then we round each pixel value by using a threshold, 0.5. This is done simply to allow easier training for the convolutional autoencoder since it is easier to reconstruct black and white images, while not losing any information on the states.

The specific algorithms used are very similar to the Q learning algorithm described earlier, and we summarise each of the algorithm together with figures below.

<u>DQN Algorithm:</u>

1) Initialise the environment to the start state.
2) Initialise 2 convolutional neural networks $Q(s_t , a_t ; w_{action})$ and $Q\big(s_t , a_t ; w_{target}\big)$ where initially they share the same parameters i.e. $w_{action} = w_{target}$.
3) Take a step in the environment according to epsilon greedy policy i.e. $\text{action} = \underset{a}{\text{argmax}}\, Q(s_t , a_t ; w_{action})$ with probability ε else take a random action sampled from a uniform distribution.
4) Add the data from the step to memory, explicitly add: $s_t, a,\ r_t,\ s_{t+1}$.
5) Uniformly sample a batch of data from memory and use this to set the target y: $r_t + \gamma \max_{a} Q\big(s_{t+1} , a; w_{target}\big)$, if s_{t+1} is a terminal state, e.g. the agent lost all lives or finished the game then the target is simply r_t.
6) Optimise the parameters w_{action} with respect to the squared loss between the prediction and target $(\mathrm{y} - Q(s_t , a_t ; w_{action}))^2$
7) Decrease probability ε.
8) Every C steps set w_{target} to w_{action}.
9) Go back to step 3 and repeat for specified number of steps.

The following two figures show the hyperparameters of the algorithm as well as the architecture of the convolutional neural network.

Figure 5 – Table showing the various hyperparameters of the DQN algorithm.

Hyperparameters	Value	Description
Number of training steps	2,000,000	The number of times we update $Q(s_t , a_t ; w_1)$ also the number of steps the agent takes in the environment approximately 9 days of play. A value of 5 – 10 million is used in the DQN paper but the training time taken to use this would be too much 18-45 hours respectively.
Gradient descent optimiser	RMSProp with learning rate = 0.00025	The optimizer used in the DQN paper, which is a variant of gradient descent.
Batch size	32	The number of data points used when calculating the average squared error.
Memory size	250,000	The size of data points of the form $s_t, a,\ r_t,\ s_{t+1}$ to store. In the DQN paper 1 million is used but that much memory was not available to us.
Initial memory start size	12500	Memory is initially filled with random data to initiate training, so the same data points aren't always sampled at the start.
Update target network frequency	Every 10,000 training steps	Update the target network parameters $w_{target} = w_{action}$ every 10,000 steps
Discount factor	0.99	Controlling how far we consider future rewards as described in figure 2.
Initial epsilon, final epsilon, number of steps needed to reach final	1, 0.1, 1,000,000	Initial and final epsilon values and the number of steps required to reach the final value. Epsilon is decreased linearly.

Figure 6 – Table showing the architecture of the convolutional neural network in DQN.

Layer 1	Convolutional layer: input channels = 4, output channels = 32, kernel size = 8, stride = 4, padding = 0, activation function = ReLU
Layer 2	Convolutional layer: input channels = 32, output channels = 64, kernel size = 4, stride = 2, padding = 0, activation function = ReLU
Layer 3	Convolutional layer: input channels = 64, output channels =64, kernel size = 3, stride = 1, padding = 0, activation function = ReLU
Layer 4	Fully connected layer with 512 outputs and ReLU activation function
Layer 5	Fully connected linear layer with 4 outputs 1 for each action

<u>Our approach</u>

1) Initialise the environment to the start state.
2) Collect data from the environment by using an agent with a random policy.
3) Initialise the convolutional autoencoder H(encoder, decoder).
4) Train the convolutional autoencoders parameters using the reconstruction error specifically with respect to the binary cross entropy loss: $\sum_{i=1}^{84}\sum_{i=1}^{84} y_i * \log\left(y_{i\,prediction}\right) + (1 - y_i) * (1 - y_{i\,prediction})$, the summation is for the loss over each pixel in the image.
5) Initialise the values of the function approximators where initially they share the same parameters i.e. $w_{action} = w_{target}$.
6) Take a step in the environment according to epsilon greedy policy i.e. $\text{action} = \underset{a}{\operatorname{argmax}}\, Q(encoder(s_t)\,, a_t\,; w_{action}\,)$ with probability ε else take a random action sampled from a uniform distribution.
7) Add the data from the step to memory, explicitly add: $encoder(s_t), a,\ r_t,\ encoder(s_{t+1})$.
8) Uniformly sample a batch of data from memory and use this to set the target y: $r_t + \gamma \max_a Q\left(encoder(s_{t+1})\,, a; w_{target}\,\right)$, if s_{t+1} is a terminal state, e.g. the agent lost all lives or finished the game then the target is simply r_t.
9) Optimise the parameters w_1 with respect to the squared loss between the prediction and target $(\mathrm{y} - Q(encoder(s_t)\,, a_t\,; w_{action}))^2$.
10) Decrease probability ε.
11) Every C steps set w_{target} to w_{action}.
12) Every N steps retrain the autoencoder for M samples.
13) Go back to 6 and repeat for specified number of frames.

The following figures show the various neural network architectures in our approach, as well as the hyperparameters for our approach.

Figure 7 – Table showing the architecture of the one hidden layer network we will be testing.

Fully connected neural network 1	Description
Input layer	Input layer of size 400, to match the output of the convolutional auto-encoder.
Hidden layer	Layer of size 512 with ReLU activation function.
Output layer	Four outputs, one for each action.

Figure 8 – Table showing the architecture of the two hidden layer network we will be testing.

Fully connected neural network 2	Description
Input layer	Input layer of size 400, to match the output of the convolutional auto-encoder.
Hidden layer	Layer of size 512 with ReLU activation function.
Hidden layer	Layer of size 128 with ReLU activation function.
Output layer	Four outputs, one for each action.

Figure 9 – Table showing the architecture of the convolutional autoencoder we use in our approach.

Convolutional auto-encoder architecture	
Encoder layer 1	Convolutional layer: input channels = 4, output channels = 32, kernel size = 8, stride = 4, padding = 0, activation function = ReLU
Encoder layer 2	Convolutional layer: input channels = 32, output channels = 64, kernel size = 4, stride = 2, padding = 0, activation function = ReLU
Encoder layer 3	Convolutional layer input channels = 64, output channels =64, kernel size = 3, stride = 1, padding = 0, activation function = ReLU
Encoder layer 4	Convolution layer input channels = 64, output channels =64, kernel size = 3, stride = 1, padding = 0, activation function = ReLU
Decoder layer 1	Transposed convolution input channels =16, output channels =16, kernel size = 3, stride = 1, padding = 0, activation function = ReLU
Decoder layer 2	Transposed convolution input channels =16, output channels =64, kernel size = 3, stride = 1, padding = 0, activation function = ReLU
Decoder layer 3	Transposed convolution input channels = 64, output channels =64, kernel size = 4, stride = 2, padding = 0, activation function = ReLU
Decoder layer 4	Transposed convolution: input channels = 64, output channels =32, kernel size =8, stride = 4, padding = 0, activation function = ReLU
Decoder layer 5	Transposed convolution: input channels =32, output channels =4, kernel size = 1, stride = 1, padding = 0, activation function = Sigmoid

Figure 10 – Table showing the various hyperparameters in our approach.

Hyperparameters	Value	Description
Number of training steps	2,000,000	The number of times we update $Q(s_t\ , a_t\ ; w_1)$ also the number of steps the agent takes in the environment approximately 9 days of play.
Gradient descent optimiser for convolutional auto-encoder	Adam optimizer with learning rate of 0.001	Variation of gradient descent. Useful for training neural networks.
Gradient descent optimiser for linear function approximator	Stochastic gradient with learning rate of 1e-7	Optimization procedure based on going in the opposite direction to the gradient.
Gradient descent optimiser for fully connected neural network function approximator	Adam optimizer with learning rate 0.0000625	Variation of gradient descent. Useful for training neural networks
Batch size	32	The number of data points used when calculating the average squared error.
Memory size	250,000	The size of data points of the form $encoder(s_t), a,\ r_t, encoder(\ s_{t+1})$ to store. Note that since we are storing the encoded states the dimensionality decreases from $84 * 84 * 4 = 28224$ to the output dimension of the encoder which is 16*5*5 = 400, thus we could have a far larger memory size. We choose not to for fair comparison.
Initial memory start size	12500	Memory is initially filled with random data to initiate training, so the same data points aren't always sampled at the start.
Update target network frequency & convolutional auto-encoder training frequency	Every 10,000 training steps	Update the target parameters $w_{target}\ =\ w_{action}$ and train the autoencoder every 10,000 steps.
Size of data sampled from memory to train the convolutional auto-encoder	64,000	Take 64,000 samples from memory.
Discount factor	0.99	Controlling how far we consider future rewards as described in figure 2.
Initial epsilon, final epsilon, number of steps needed to reach final	1, 0.1, 1,000,000	Initial and final epsilon values and the number of steps required to reach the final value. Epsilon is decreased linearly.

5. Results

Obtaining the results was a lengthy process with a single agent taking about 9 to 10 hours to train. Due to this we could not search over hyperparameters which is crucial when training these agents. The results that we show are the average reward per episode, where one episode is the start of an instance of a game to the finish of the game i.e. when the agent dies or completes the game, this is an indication of the performance of the agent. We also show the average action value prediction for a set of random states that have been collected beforehand, this gives some indication of stability for example if the action value prediction looks like it going to diverge to infinity. These values are recorded every 8000 training steps. In the figures we show 1 step on the horizontal axis as 8000 training steps.

The first approach we will show is DQN, the one we are comparing against, the results are as expected and in-line with what would be expected from the paper. We calculate the performance by looking at the average reward for the last 125 steps, or last 1,000,000 training steps since 1 step = 8000 training steps. This is done because as shown in figures 5 & 10 the epsilon value which is used to allow the agent to explore the environment by causing random actions, reaches its minimum at 1,000,000 training steps, in other words this is when the agent is behaving according its 'true' epsilon greedy policy. For DQN an average reward of 2.42 is calculated.

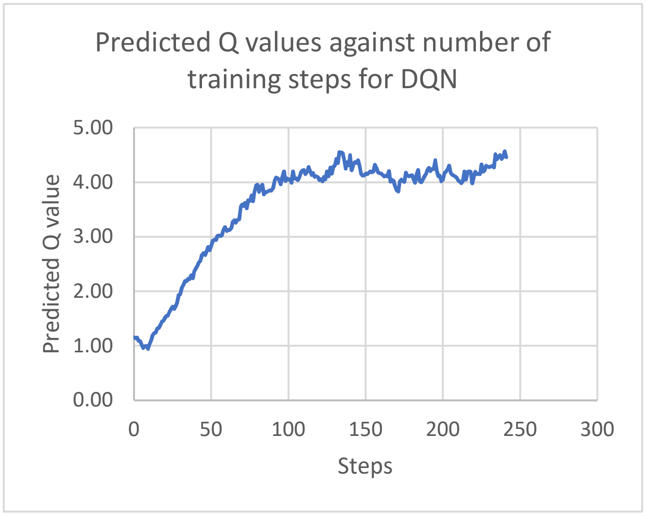


Figure 11 – DQN graph for predicted Q values

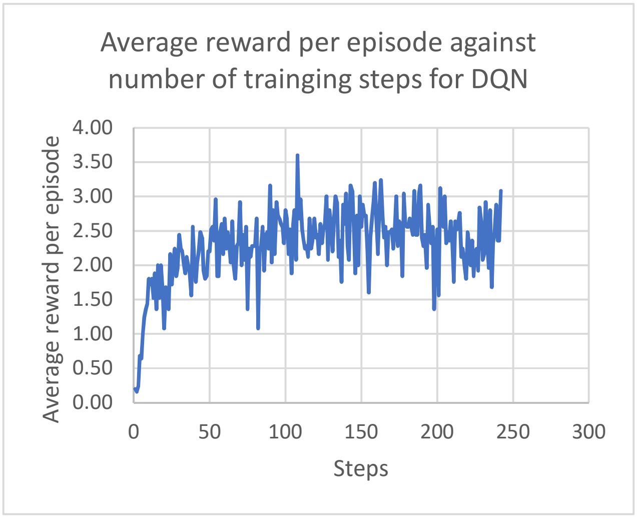


Figure 12 – DQN graph for average reward per episode

The second result that we will see is the case where the function approximator is a linear one. Initially to minimise the loss $(y - Q(encoder(s_t), a_t; w_1))^2$ we simply used matrix inversion via singular valued decomposition (SVD), which returns the w_1 that minimises the error directly. This however led to large numerical instability where w_1 diverged to infinity. This arises due to the very small losses that caused numerical implementations of SVD to diverge. The issue was resolved by instead using gradient descent with the small learning rate in figure 8.

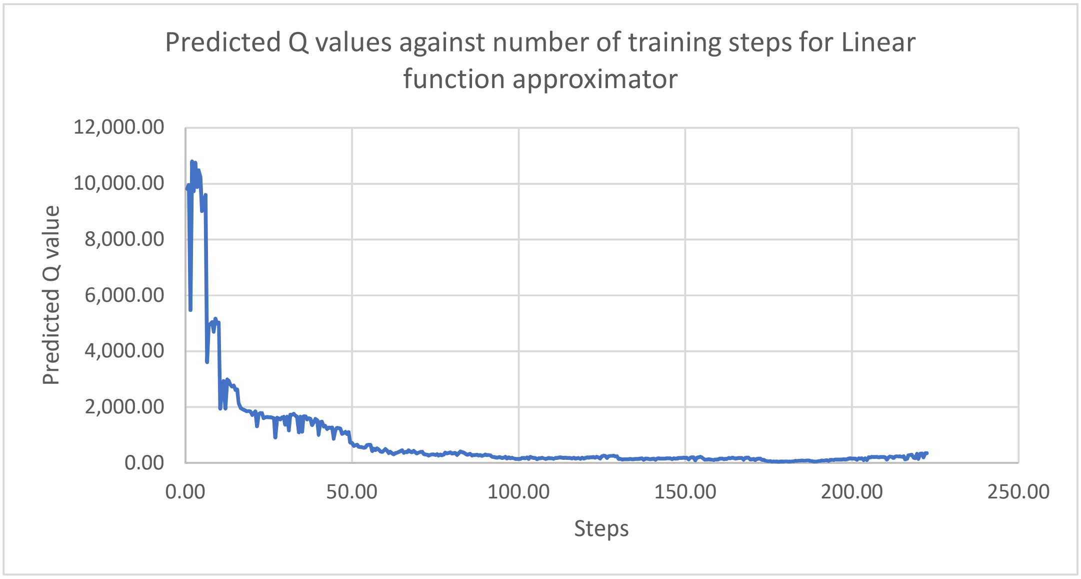


Figure 13 – graph for predicted Q values for our approach with linear function approximator

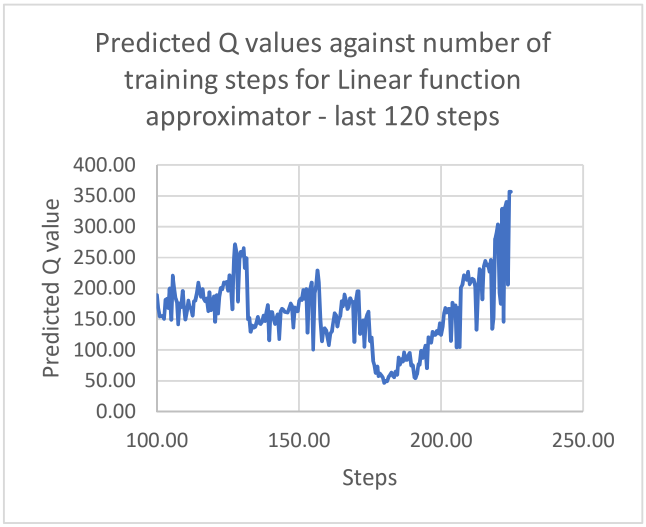


Figure 14 – Graph for predicted Q values for last 120 steps for our approach with linear function approximator

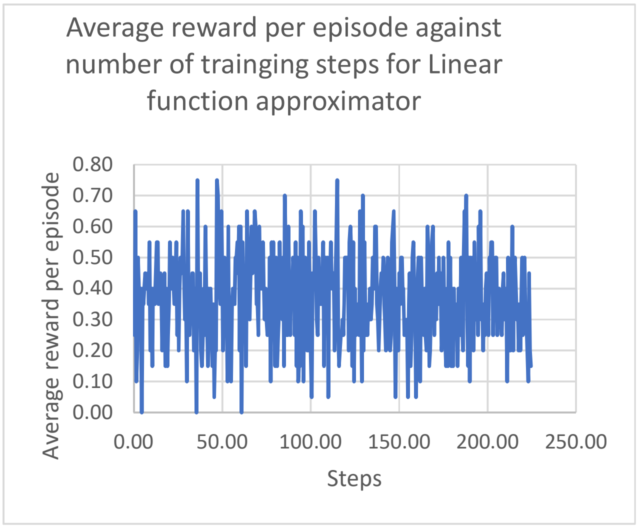
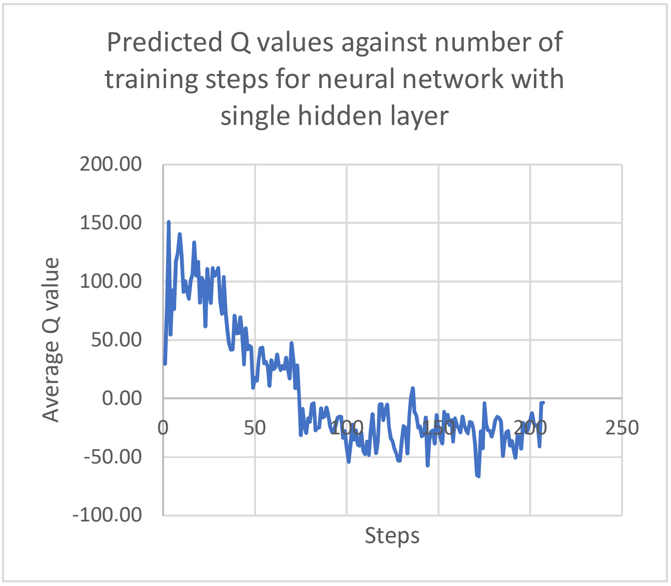


Figure 15 – Graph for average reward per episode for our approach with linear function approximator.

The results obtained initially show the Q value converging, after re-scaling we can see it is very high variance in comparison to DQN. We can calculate the performance like we did with DQN to get a value of 0.363, this is no better than random action. This result shows that linear function approximators are not well equipped for high dimensional reinforcement learning problems, even with the feature extraction from the convolutional autoencoder. This is most likely due to the limited nature of linear functions, in terms of what they can express i.e. they can only implement Q functions within the span of its basis functions. This problem is relieved when using a fully connected neural network with a single hidden layer, since it does not have this same limitation as it can approximate any function, it is a universal function approximator.



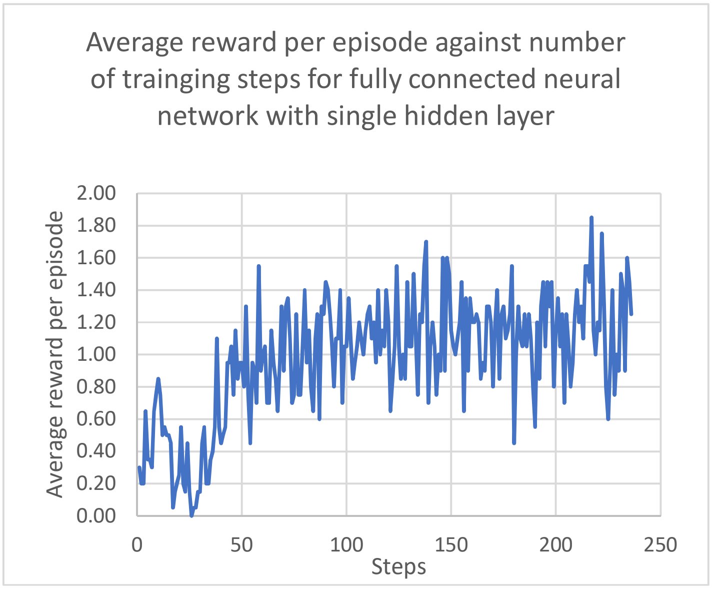
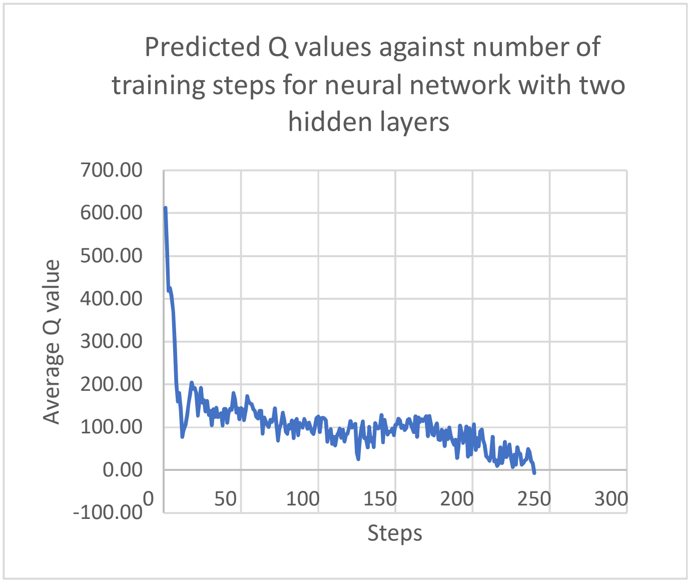


Figure 16 - Graph of predicted Q values for our approach with fully connected neural network, with 1 hidden layers with 512 hidden units followed by a ReLU activation function.

Figure 17 – Graph of average reward per episode for our approach with fully connected neural network, with 1 hidden layers with 512 hidden units followed by a ReLU activation function.

When using the neural network, we can see that average reward is better than that of the linear function approximator also the predicted action values indicate that it is much more stable compared to the linear function approximator. The performance however still falls short of DQN which achieves an average reward of 2.42 per episode compared to ours which is 1.15.

The last experiment that was tried was identical to the ones in figure 14 and 15, however an extra hidden layer was added, for a total of 2 hidden layers the first of size 512 units followed by a ReLU activation function and the second of size 128 units followed by a ReLU activation function. The results are shown below:



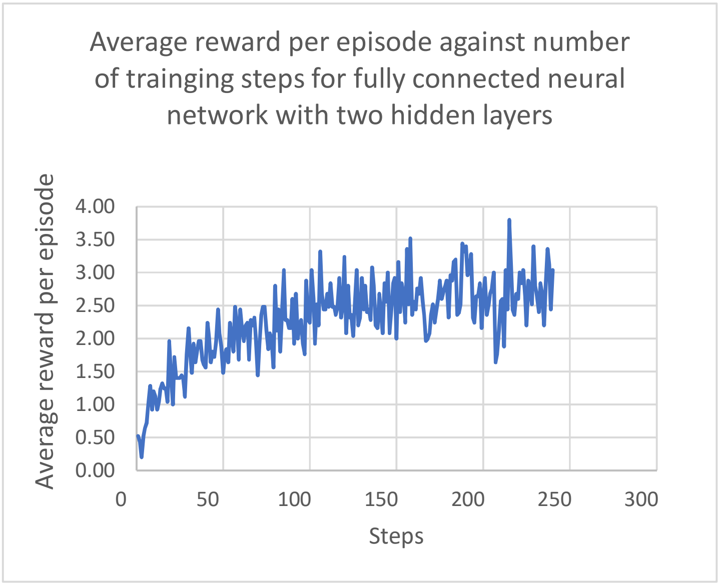


Figure 18 – Graph of predicted Q values for our approach with fully connected neural network, with 2 hidden layers the first with 512 hidden units followed by a ReLU activation function and the second with 128 hidden units followed by a ReLU activation function.

Figure 19 – Graph of average reward per episode for our approach with fully connected neural network, with 2 hidden layers the first with 512 hidden units followed by a ReLU activation function and the second with 128 hidden units followed by a ReLU activation

With this result we see that the predicted Q value appears to have stabilised further with less oscillations. The result to note however is the average reward, the performance which is 2.63 compared to 2.42 of DQN. In addition to getting a better performance, our approach is much more memory efficient as well, comparing the extremely high dimensional state of size 28224 that is stored in the memory against the relatively low dimensionality of the encoded state of size 400 stored in memory.

We can propose the reason behind these results in that perhaps the success of DQN is due to the convolutional layers that excel at extracting features, and not necessarily the end to end approach that has been used. If this is the case it may be more beneficial to transition to an architecture like ours where we employ a feature extractor coupled with a powerful function approximator, due to the better memory efficiency and because, of the advantages mentioned in [16] and [18] i.e. the observed better transfer learning behaviour and the general flexibility of our approach.

5. Conclusion

To conclude, if we list our initial goals:

1) Try a different approach to current state of the art methods, that addresses current short comings and is motivated by literature.
2) Implement the approach.
3) Compare the two approaches.
   -Which one performs better?
   -Which one is more data efficient?
   -Is the end to end approach of DQN superior to ours?

We can see that indeed we have met all of them, we first proposed an approach based on current literature in attempt to address problems with DQN, after which we implemented the approach in python, and tested it against the state-of-the-art, DQN. We obtained results that suggest that our approach reaches a greater performance with the same amount of data implying greater data efficiency as well, on top of being more memory efficient and flexible.

While the results have shown that our approach performs better, we should also note that that since there is some variability due to the random network initialisations and batch sampling. This also implies we need further testing to say for sure our approach is superior, but the results obtained so far add confidence to the belief that this could be the case.

In future work we may aim to perform a hyperparameter search, since due to the long training times we were not able to search for hyperparameters other than those mention in figure 7 and 9, whereas for DQN the hyperparameters were used from the DQN paper where they conducted a hyperparameter search for good selections, together with this it would also be useful to compare the transfer learning ability of the two.

References

- [0] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis, “Mastering the game of go without human knowledge”, Nature 550, pages 354–359 (19 October 2017)
- [1] David silver, *Science* 07 Dec 2018: Vol. 362, Issue 6419, pp. 1140-1144
- [2] Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An introduction
- [3] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine Learning, 8(3- 4):279–292, 1992
- [4] J. Park, I. W. Sandberg, “Universal approximation using radial-basis-function networks”, Neural computation, Volume 3, Issue 2, 246-257 (1991)
- [5] Sridhar Mahadevan, Mauro Maggioni, “Proto-value Functions: A Laplacian Framework for Learning Representation and Control in Markov Decision Processes”, Journal of Machine Learning Research 8 (2007) 2169-2231
- [6] George Konidaris, Sarah Osentoski, Philip Thomas “Value Function Approximation in Reinforcement Learning using the Fourier Basis”,(2011) 25th AAAI Conference on Artificial Intelligence
- [7] Hado van Hasselt “Double Q-learning” at NIPS 2010
- [8] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas “Duelling Network Architectures for Deep Reinforcement Learning” arXiv 2016
- [9] M Hessel, J Modayil, H van Hasselt, T Schaul, G Ostrovski, W Dabney, D Horgan, B Piot, M G Azar, D Silver, “Rainbow: Combining Improvements in Deep Reinforcement Learning”, arXix 2017
- [10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. “Playing atari with deep reinforcement learning” at NIPS Deep Learning Workshop 2013
- [11] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In Computer Vision and Pattern Recognition (CVPR), pages 2528–2535. IEEE, 2010.
- [12] Sascha Lange, Martin Riedmiller, “Deep auto-encoder neural networks in reinforcement learning” at IJCNN 2010
- [13] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, Advances in Neural Information Processing Systems 25 (NIPS 2012)
- [14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep Residual Learning for Image Recognition

- [15] Thomas N. Kipf, Max Welling, “Semi-Supervised Classification with Graph Convolutional Networks” at ICLR 2017
- [16] Daniele Grattarola, “Deep feature extraction for sample efficient reinforcement learning” master’s Thesis
- [17] Irina Higgins, Arka Pal, Andrei A Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, Alexander Lerchner, “DARLA: Improving Zero-Shot Transfer in Reinforcement Learning” at ICML 2017
- [18] World Models David Ha, Jürgen Schmidhuber, arXiv:1803.10122
- [19] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder. ACM Trans. Graph. 36, 4, Article 98 (July 2017)
- [20] https://pytorch.org/docs/stable/index.html
- [21] https://gym.openai.com/envs/Breakout-v0/