EEE 485 Term Project, Final Report

Beste Aydemir, Ege Ozan Özyedek
{beste.aydemir, ozan.ozyedek} @ ug.bilkent.edu.tr

I. Introduction

A Reinforcement Learning (RL) problem includes an agent and an environment. The agent's action can be chosen with one of the reinforcement learning algorithms. For this project, we applied Episodic Semi Gradient SARSA, Deep Q-Learning and Deep Deterministic Policy Gradient (DDPG) algorithms. Since these are RL algorithms, there is no dataset. The algorithms are tested on Open AI Gym [1]. We chose the SARSA as an introduction to the RL problems. Deep Q-Learning is more suitable for larger state spaces. For those cases, action value function is estimated with a neural network, as a non-linear approximator. Finally, we decided to implement last algorithm (DDPG) because it can operate on continuous action spaces. All of the implementation is done on Python and Numpy, without utilizing machine learning libraries.

Methods (Section 2), Results (Section 3) and the Conclusion (Section 4) are given below. Methods and Results sections are divided into four subsections: Episodic Semi Gradient SARSA, Neural Network, DQN and DDPG. Although it is not one of our RL algorithms, Neural Network section is given as to explain it in detail and present its modular test results. The algorithm pseudo codes and the Python code are given in Appendix.

II. Methods

A. Episodic Semi Gradient SARSA

As our first RL algorithm implementation, we chose State-Action-Reward-State-Action (SARSA). There are several reasons behind this choice, for the most part we wanted to ascend in complexity of the algorithms we chose to implement and bare-bones SARSA is the least complex out of the three that we chose. Below, the description of the theory behind SARSA, the implementation details, our current progress and finally further goals can be found.

1) Description: SARSA is an RL algorithm which has he goal of finding the Q-value function, $q_*(S,A)$, where S represents the state space and A represents the action space. It learns on-policy, meaning for a policy given for action selection the learning also depends on that policy. This is different from off-policy methods (such as Q-Learning which is also reviewed in this report) which learn only using the action which maximizes the Q value [2]. Below, the update rule for the discrete version of the algorithm can be found.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)] \tag{1}$$

Unfortunately, most problems do not occur in a discrete state space. Hence this version of the SARSA algorithm deems useless in environments such as Mountain Car and Cart Pole (both of which are used to test the presented

versions of SARSA). Instead of finding the Q-value function, we can approximate it. This version of SARSA is called Semi-Gradient SARSA, and it approximates the q-function by the following linear equation.

$$\hat{q}(s,a,w) = w^T x(s,a) \tag{2}$$

Where the vector w is a weight vector that gets updated at each step, and $x(s,a)$ is a feature vector that is constructed according to the state and action given. The weight update is as follows, where ∇ defines the gradient.

$$w_{t+1} \leftarrow w_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1},w_t) - \hat{q}(S_t,A_t,w_t)]\nabla\hat{q}(S_t,A_t,w_t) \tag{3}$$

However, since $\hat{q}$ is linearly defined, the gradient (w.r.t the weight vector) is nothing but the feature vector $x(s,a)$. This simplifies the implementation.

2) Implementation Details: For the implementation of Episodic Semi-Gradient SARSA, we followed the algorithm provided in Sutton's book [2]. Our implementation of the algorithm can be observed in Algorithm 1 (in the Appendix). The update function written in the algorithm below updates the model weights as explained in the previous section. For the policy, we used the $\varepsilon-$greedy policy, this can be observed in Algorithm 2 (in the Appendix) [3].

An important note on the linear relation that defines $\hat{q}$ between the feature vector x and weight vector w, since the feature construction only depends on the states, we change the elements in Equation 2 slightly, instead of having a weight vector, we create a weight matrix that has dimension $(d,|A|)$. This way we have separate weight vectors for each action that maps the feature vector to the corresponding $\hat{q}$ of that action.

As mentioned, a feature construction method is required for the algorithm to learn and converge to an optimal q_* [2]. Sutton's book provides multiple different methods for such a constructor, these include polynomial basis, Fourier basis, two different coding methods (coarse coding and tile coding) and finally the feature constructor that we used, a radial basis function. The general goal of these methods is to create different relations between the state features that are observed from the environment so that the model can chose its next action more accurately.

Radial basis functions (RBF) create features following a Gaussian equation, the i_{th} element of the feature vector $x(s)$ is defined as shown below.

$$x_i(s) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{||c_i - s||}{2\sigma^2}} \tag{4}$$

In this equation, s defines the state vector that is obtained from the environment, c_i indicates the center (mean) of the Gaussian and σ is the standard deviation. To choose the centers and variances of RBFs for our problem, we followed a paper that reviews different feature construction methods for RL. It was advised to use n evenly distributed centers for each dimension d of the state vector s, and to take the variance as $\frac{2}{n-1}$ for all centers [4]. This in the end results in n^d RBFs.

The idea of RBF is that, the point that represents the state vector gets different values based on its distance to each center (hence each RBF). This way each state value can be related with different RBFs and the weights can be updated accordingly, resulting in better learning of the problem.

B. Multi-Layer Perceptron (Neural Network)

1) Description: Although not the focal part of the project, we implemented a neural network for the Deep Q-Network and DDPG models. Below, a short description, implementation details, results and goals can be found.

Neural networks are predictive models that are inspired by the way the human brain works. They are non-linear thanks to the activation function that is at the end of each layer. This way the model fits data and predicts in a more complex way, resulting in a more accurate model [5]. We have only implemented a fully connected layer (a Perceptron), hence this section will follow the Multi Layer Perceptron model.

2) Implementation Details: Our implementation of a neural network consists of two parts. The first one is the class that creates the network structure, called Network. The other, are the layers that are in the network. In this section, we will review the fundamental mathematical equations that are implemented so that the network learns and updates weights accordingly.

Starting with the forward pass, this is where the model predicts the input of the next layer, for the i_th layer this can be shown as below, ϕ is the nonlinear activation function, W_i is the weight matrix, b is the bias vector, X_i is the input of the layer.

$$X_{i+1} = \phi(X_i W_i + b) \tag{5}$$

Next is the backward pass, after the forward pass is complete, gradient descent takes affect. All neural networks have a cost function that determines whether the prediction was successful or not (mean squared error and cross entropy error are two examples). Gradient descent enables the network to update its weights according to this function. After the forward pass, for each layer weight, we want to find its gradient with respect to this function. Below are two equations that sum this up. The residual term is the residual error gradient that is updated at each layer to find the gradient of the next layer weights/bias. Back propagation exploits the chain rule in derivation. Assume the equations below are for the i_th layer. ϕ' is the derivative of the activation function and y_i is the output of the forward pass of that layer.

$$\frac{\partial E}{\partial W_i} = \text{residual} \cdot \phi_i' \cdot y_{i-1} \tag{6}$$

$$\frac{\partial E}{\partial b_i} = \vec{1} \cdot \text{residual} \tag{7}$$

Where the residual updates as follows.

$$\text{residual} \leftarrow \text{residual} \cdot \phi_i' \cdot W_i \tag{8}$$

The final part is the update of the network weights, it uses the gradients we have found above.

$$(W_i)_{t+1} \leftarrow (W_i)_t - \alpha \frac{\partial E}{\partial W_i} \tag{9}$$

C. Deep Q-Learning

1) Description: Tabular methods are not appropriate when the state space is too large. Deep Q-Networks approximate the Q function using a neural network. The goal is to find the Q function as the output of a neural network when the state is the input [6].

However, this method can have unstable learning since it has the three elements of a deadly triad issue [2]: function approximation, bootstrapping, and off-policy training. Deep Q methods approximate functions (with large state spaces) using bootstrap (using the estimation of the reward instead of observing at the end of the episode at temporal difference (TD) methods). Also, Q-Learning is off-policy training as the policy is learned independently from the agent's actions [7]. To overcome this unstable learning issue, two methods have been proposed: target network and replay memory (See Implementation details) [8].

In most of the applications of Deep Q Networks, the state is given as an image of the current state (e.g. one frame of Atari games, such as Breakout) [9]. So, in order to process the images better, usually a Convolutional Neural Network (CNN) is used. However, another version of state input can be used, where RAM of the Atari machine is the input. But for this project, we only use a regular feed-forward neural network for nonlinear Q-value function estimation. Algorithm 3 is updated by omitting the necessary steps for handling states as images as they are not necessary for RAM input implementation.

2) Implementation Details: **Target network** freezes the network parameters for a few iterations and updates its weights after the iterations. It is implemented by maintaining another network for generating targets y_j and copying the main network to this target network every C steps (See hyperparameters). This reduces the risk of divergence and makes the overall algorithm more stable because it introduces a delay between when the main target is updated and when that update changes y_j's [10].

Replay memory keeps (s, a, r, s') tuples for the last N time steps as experience. During the updates, random samples are used from these tuples, resulting in less correlation, since the sampled experiences are not adjacent in time to each other. The consecutive steps can contain the same or similar actions [10]. In addition, a time step is used multiple times for updating the network, which is more data efficient [10].

D. Deep Deterministic Policy Gradient (DDPG)

1) Description: Actor Critic methods approximate both the Q-function and also additionally the policy which controls action selection. Policy is denoted by a parameter vector $\theta^{\mu} \in \mathbb{R}^d$ and allows the calculation of the probability of choosing that action in a given state [2]. In addition to the policy, a value function parameterized by $\theta^{Q} \in \mathbb{R}^{d'}$ can be constructed [2], [11]. These approximators are called "actor" for the policy approximator and "critic" for the value function approximator.

For this project, we chose the Deep Deterministic Policy Gradient (DDPG) algorithm to implement. For the previous methods, the policy function is a probability over possible actions and stochastic. But this deterministic model constructs the policy as a deterministic decision, the network outputs a single action [12]. In order to introduce exploration to this algorithm, an exploration noise is added while selecting an action.

Per its definition, DDPG can solve problems with continuous action spaces. For environments such as MuJoCo, Bipedal Walker and Pendulum such models are required [13]. Because of its use of a replay buffer and target networks, it can be defined as an extension of DQN for continuous action spaces [11]. Algorithm 4 as proposed is given in the Appendix [14].

2) Implementation Details: The implementation of DDPG follows the implementation in its paper, and for our case it can be divided into three parts: Actor, Critic and combining the two to create the DDPG model.

The implementation of the actor is an extension of the neural network that we have presented in the previous sections. Since DDPG uses target networks for the actor, target layers are defined. The target update is as given in the algorithm. One important difference of the actor network is its update rule. The update term is shown in the algorithm, although this can be a little confusing. It uses the chain rule to take the derivative of the policy error, which is defined as the sum of Q values for the given mini batch.

$$\frac{\partial Q(s,\mu(s))}{\partial \theta^{\mu}} = \frac{\partial Q(s,\mu(s))}{\partial \mu(s)} \frac{\partial \mu(s))}{\partial \theta^{\mu}} \tag{10}$$

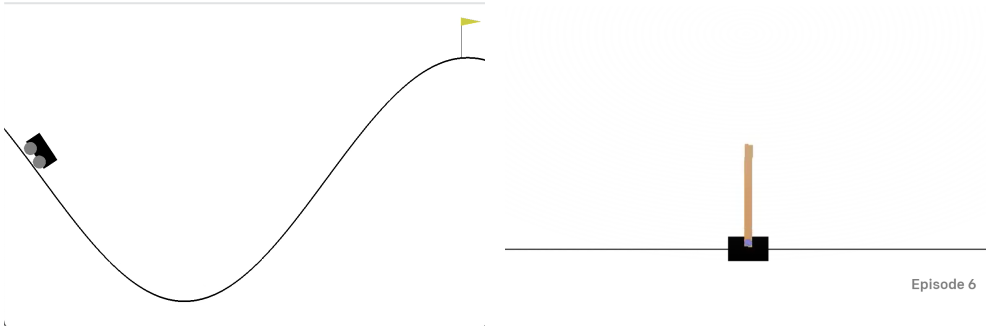
The first derivative is obtained from the critic network, while the second derivative follows the backpropagation algorithm. The critic network, similar to the actor network, has a set of target layers that have the same initial values as critic weights and follows the update rule in the algorithm. One challenging step for the implementation of critic was the fact that it has two inputs, the state vector and the action vector. To solve this, three different sets of layers were defined, one taking the state vector as input, the second taking the actions as it input. The output of these layers are concatenated and then fed into the third set of layers, which defines the Q-value as its output. The backpropagation of the concatenation layer is simple, the residual term is separated according to the output size of the state and action layers. The derivative $\frac{\partial Q(s,\mu(s))}{\partial \mu(s)}$ simply corresponds to the residual at the end of the action layer and is passed onto the update of the actor.

The final step of the implementation is simply the assembly of the actor and critic networks. The training is done according to the algorithms. Similar to DQN, a replay buffer is used. To compensate for the lack of exploration, random Gaussian noise is added to the predicted action at the start of each step. The rate of this is controlled with a variable, ε, similar to DQN and Sarsa.

III. Results

A. Episodic Semi Gradient SARSA

To train and evaluate our implementation of SARSA, we used the Open AI Gym library, and more specifically the mountain car and cartpole environments.



(a) The Mountain Car Environment. (b) The Cart Pole Environment.

Figure 1. Test Environments for Sarsa.

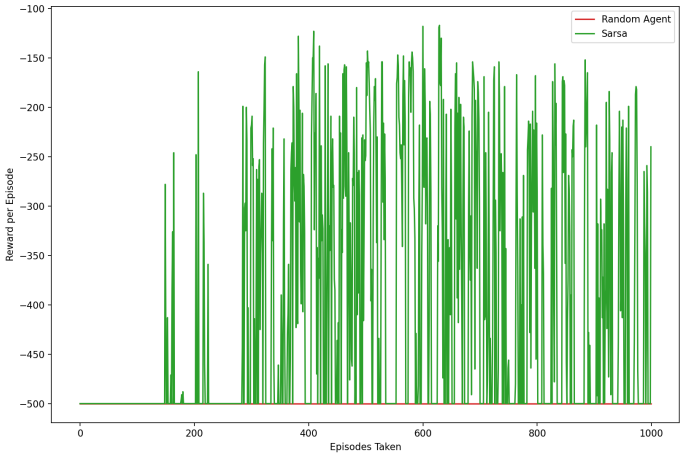
Table I
HYPER-PARAMETERS FOR EPISODIC SEMI GRADIENT SARSA

Environment	Episodes	Learning Rate (α)	Discount Rate (γ)	Exploration (ε)	RBF Order (n)
MountainCar-v0	1000	0.05	1	$1 \rightarrow 0.1$	6
CartPole-v1	400	0.05	1	$1 \rightarrow 0.1$	6

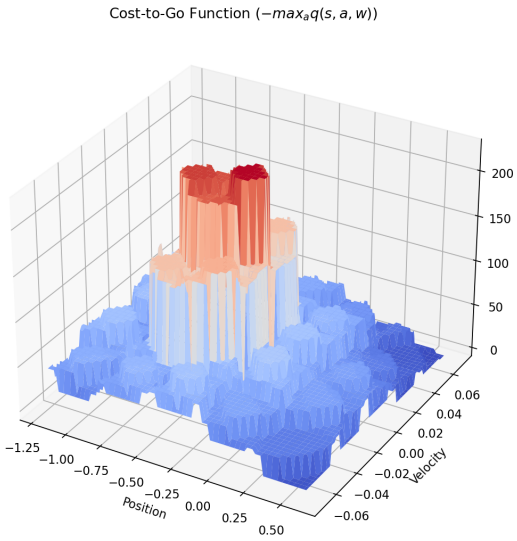
1) Mountain Car: Mountain Car is a classic control problem, and one that we can compare to other implementations easily, since the book that we followed for the implementation also tested their model on this environment. Mountain Car rewards -1 for each step where the problem is not solved. So the goal of the algorithm is to get its total reward as. close to zero as possible. Additionally, its easily visualized compared to other methods since its state vector only has 2 features, position and velocity [15].

The optimized hyper-parameter list can be reviewed in Table I. To increase faster learning of different outcomes, we use decaying ε, which motivates the model to explore first and as it converges the exploration decays to 0 and it only exploits its findings to chose the next action. We also decay the learning rate ever so slightly so that the model can converge to the optimal function properly.

Below two different figures that give information on the success of the algorithm can be observed. Observing Figure 2.a we can understand that the algorithm reaches to higher rewards steps per episode after 400 episodes. It is constantly unstable however; and this might be caused by a phenomenon called catastrophic forgetting [16]. This is a common problem that is faced in many learning algorithms, but is most prominent in neural network based models. To prevent this, as mentioned previously, decaying the learning rate is useful. This way the algorithms learning gets less impactful as time goes on, leading to more stable results.



(a) Reward per Episode for Sarsa and Random Agent.



(b) Cost-to-Go function of Episodic Semi Gradient SARSA.

Figure 2. Results of Episodic Semi Gradient Sarsa with Mountain Car Environment.

Another important visualization that we can make, given that our state space is 2 dimensional, is visualizing the cost-to-go function. This can be viewed in Figure 2.b . To clearly understand this visualization, lets first review the Mountain Car problem in summary. The car has to move in the opposite way of the goal in order to gain acceleration and make it to the top. Looking at the figure we can see that the starting position with low velocity

has the most cost value, which motivates the model to act to change its position. On the contrary, the blue spaces encircle this high-cost area, which makes sense considering the fact that as long as it gains acceleration its position doesn't really matter.

2) Cart Pole: The Cart Pole problem is a bit more complex, since its state vector has 4 features. The environment rewards the agent 1 for each step where the agent balances the pole. The agents goal is again to maximize the reward. The Cart Pole problem is "solved" when the agent gets an average of at least 195 reward or for 100 consecutive episodes [17].

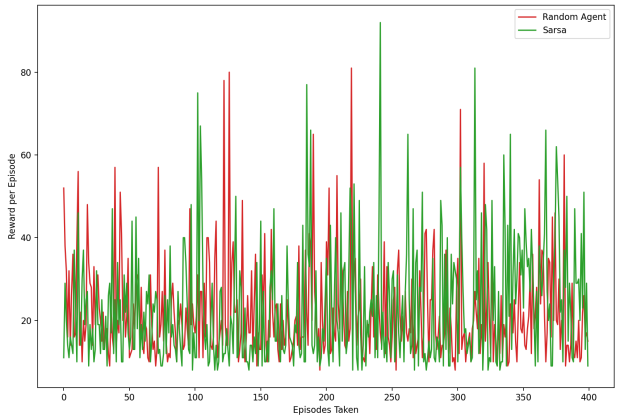


Figure 3. Reward per Episode for Sarsa and Random Agent with Cart Pole Environment.

A major setback of SARSA, and also using RBF as the feature constructed is that it cannot find a good approximation on higher dimensional state spaces [18]. This setback is prominent in Cart Pole, and the results can be seen in Figure 3. Another problem that SARSA faces is it's incredibly slow in higher dimensions, for RBF with $n = 6$ it takes around 5 minutes for SARSA to go over 1000 episodes. RBF is way too computationally consuming for us to chose its order high enough for good approximations, and Episodic Semi Gradient SARSA is not a complex enough model to fit problems such as Atari games. SARSA acts similarly to the random agent, which means that it's not learning the Cart Pole problem.

3) Comparison and Insights: SARSA positions itself as an introductory model to RL rather than being one of the pinnacle algorithms that are widely used today. Although on-policy methods generally find better solutions, because they can't use replay buffers in definition (since the action predictions need to follow the current policy), the SARSA model becomes insufficient to solve the given environments [19]. Additionally, the use of RBF's slow down the training process. Some solutions can be proposed such as Tile Coding which constructs features similar to RBF's but without computation [2]. An additional benefit of this is that it creates sparse vector, which again adds to the efficiency.

B. Neural Network

As a quick sanity check and to test whether the network implemented was working before using it in DQN and DDPG algorithms, we used a simple dataset called Pima Indians Diabetes Dataset with 769 samples to see if the network would converge to a low loss value [20]. We used MSE as our cost function, $\alpha = 0.4$ as the learning rate, 2 hidden layers with 5 neurons each and activation function $tanh$, and a sigmoid output layer with a single neuron (which we round to find the classification of prediction). We trained on 700 samples and tested the predictions in remaining samples. As it can be observed in the loss plot of the network in Figure 4, the network converges to around $MSE = 0.21$. Additionally, this dataset is deemed solved when the prediction accuracy is between $65-77\%$, and we got a test accuracy of 69%. Considering the fact that we are doing batch and not using mini-batch learning, this is a good result and shows us that the network implementation works as intended.

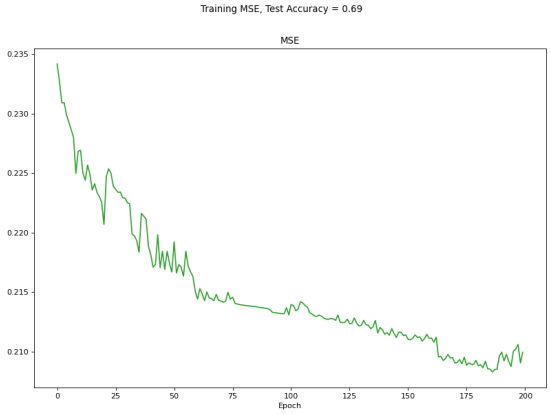
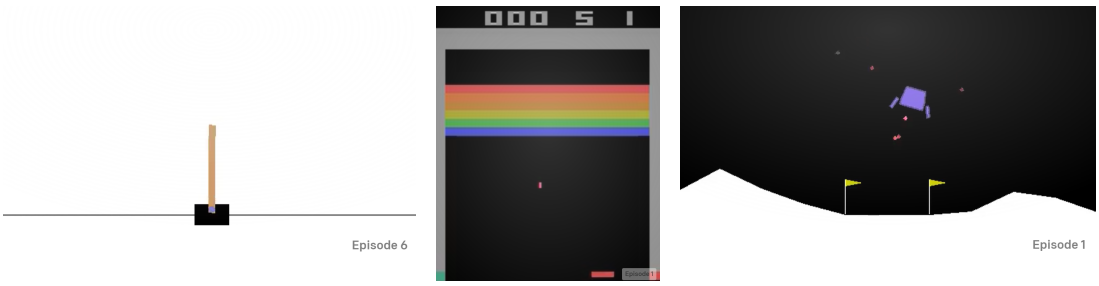


Figure 4. Loss plot of network, test accuracy of 69%.

C. Deep Q-Learning



(a) Cart Pole. (b) Breakout. (c) Lunar Lander.

Figure 5. Environments used for DQN.

Table II
HYPER-PARAMETERS FOR DQN

Environment	Episodes	α	γ	ε	Batch Size	Target Update (C)
CartPole-v0	400	0.002	0.95	$1 \rightarrow 0.1$	32	4
LunarLander-v2	500	0.001	0.95	$1 \rightarrow 0.01$	64	100
Breakout-ram-v0	1500	0.0002	0.99	$1 \rightarrow^{L} 0.01$	128	100

1) Cart Pole: For the Cart Pole environment, the hyper parameters are given on the hyper parameter table. For understanding the data structures to implement the algorithm on [21] efficiently and get a starting point on the search for hyper parameters, we examined the implementation on [22] loosely as an example to guide us.

The reward plots are given on the Figure 6. As it can be observed, DQN reward clearly differs from the random agent's rewards, meaning that the agent is learning. The average reward for the random agent is found to be 22 with our random agent, whereas it is 174 for DQN between episodes 250-350 (average). For this environment, the maximum reward is 200. After nearly 250 episodes of training, we can say that the agent can balance the cart pole and succeeds in the environment. From the leader board of the Open AI Gym [17], the best scores indicates learning after 2-3 episodes with more sophisticated algorithms. However, the leader board ranges from bests to 382 episodes of training, so we can say that our results are comparable and the learning is not too slow either.

2) Lunar Lander: There is not a maximum reward on Lunar Lander, however solving is defined as a reward of 200. We can see from the plot that our algorithm learned the environment after roughly 215 episodes. The average

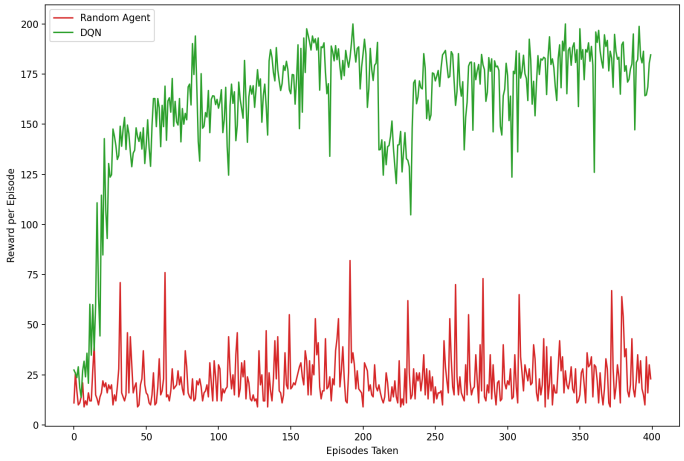


Figure 6. Reward per Episode for DQN and Random Agent on the Cart Pole Environment

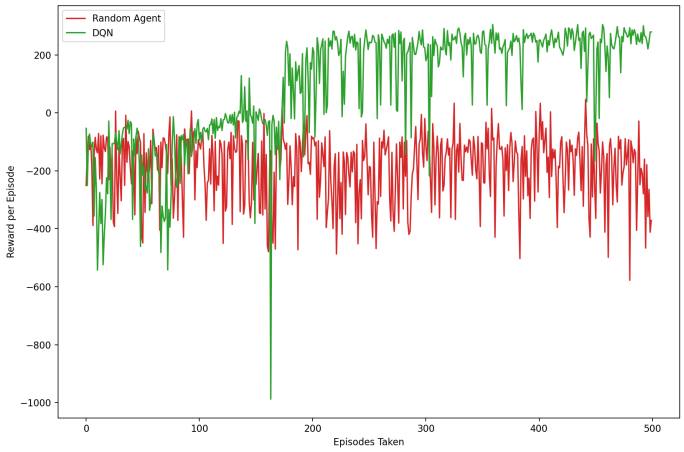


Figure 7. Reward per Episode for DQN and Random Agent on the Lunar Lander Environment.

reward is clearly different from the random agent's average of -186, our average between episodes 215 to 315 is 201. The leader board [17] indicated that for this environment, the fastest learning is 16 episodes. However, again our results are comparable to other learning algorithms as the leader board ranges from 16 to degrees of 1000.

3) Breakout: Again, there is not a maximum reward for this environment. Another hurdle with this environment is that the best algorithms utilize CNN and more complicated algorithms while we are using a RAM based implementation on a vanilla DQN. As it can be observed from the reward plot, DQN starts to learn the environment and performs better than the random agent, but the learning is too slow for this algorithm to be used for practical purposes. We acknowledge that the poor performance of this algorithm is due to not using more sophisticated methods and not using CNNs where spatial information is more highly utilized. Also, for 1500 episodes, the algorithm took more than 7 hours. This can be linked to our lack of processing power, not utilizing a GPU and the fact that we have not used highly optimized and fast libraries and platforms such as TensorFlow or PyTorch.

4) Comparison and Insights: DQN's learning time (or episodes until the environment is learned), highly depends on its action and state space. For our environments the smallest action and the state space belongs to the Cart Pole and the learning is the most rapid for that case. Lunar Lander learns after around 200 episodes. But for the Breakout environment, the learning would take even more than 1500 episodes even if we hadn't terminated the algorithm. This is due to the fact that the state space is 128 dimensional and there are 4 possible actions, which is larger than

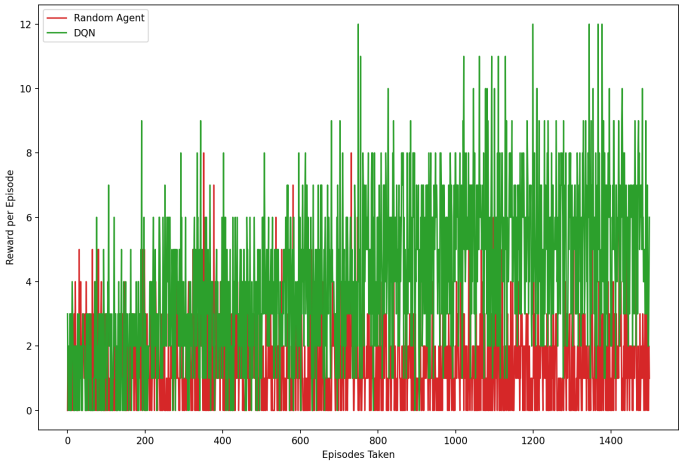


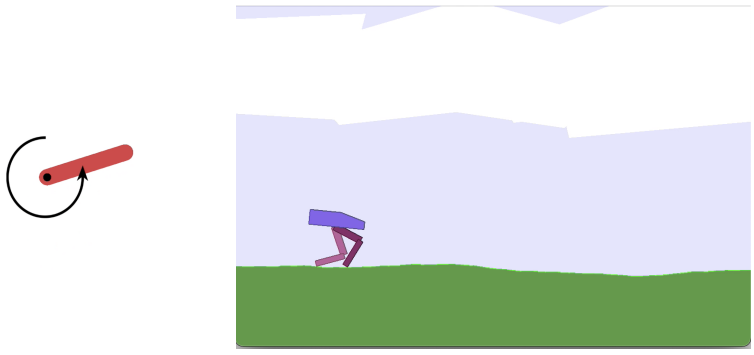
Figure 8. Reward per Episode for DQN and Random Agent on the Breakout Environment.

the previous two environments.

DQN can be compared with SARSA on CartPole environment. Figure 3, Reward per Episode for Sarsa and Random Agent with Cart Pole Environment, shows that SARSA is not applicable for CartPole problem and indicated the necessity of the DQN, a more complicated non-linear function approximator for the Q value function.

Another insight we obtained was during the annealing of the hyper parameters ε and α. Decreasing the learning and the exploration rate through the course of the algorithm highly benefits the algorithm. Another thing that we concluded was that the exploratory experiences in the beginning of the algorithm benefit the performance on the long run, since they are used again and again via experience replay.

D. Deep Deterministic Policy Gradient (DDPG)



(a) Pendulum. (b) Bipedal Walker.

Figure 9. Environments used for DDPG.

The hyperparameters chosen for test runs of DDPG are given in Table III. To correctly choose a starting point for hyperparameters and the network structures for both actor and critic networks different implementations such as [23], [24], [25], [26] were reviewed.

1) Swinging Pendulum: The swinging pendulum environment has 3 features on its state space and a single feature on its action space. The action space is continuous. The goal for the agent is to balance the pendulum that starts

Table III
HYPER-PARAMETERS FOR DDPG

DDPG	Episodes	Critic α	Actor α	γ	ε	Batch Size	τ
Pendulum-v0	120	0.001	0.0001	0.99	$1 \rightarrow 0.01$	64	0.001
BipedalWalker-v3	2000	0.001	0.0001	0.99	$1 \rightarrow^{L} 0.01$	128	0.001

at the bottom. The max reward per episode would be 0 in the perfect case. As of yet, there is no definite solution to the pendulum problem, but the best average reward per episode values are between -123 to -152 according to OpenAI Gym's leaderboards [17].

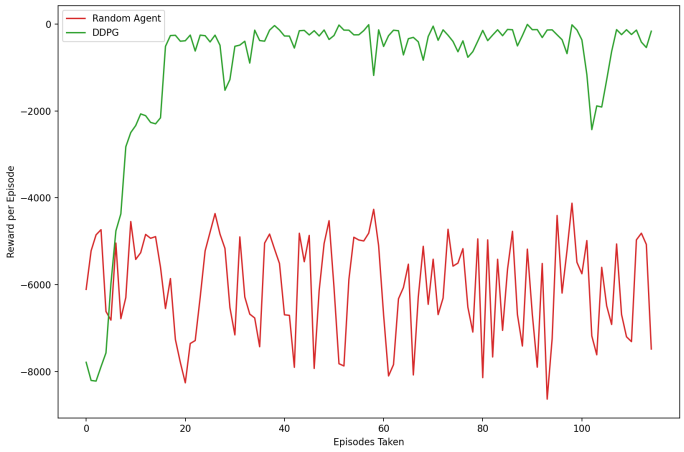


Figure 10. Reward per Episode for DDPG and Random Agent on the Pendulum Environment.

Our implementation of DDPG successfully managed to learn the pendulum environment. The random agent has -6235 reward average over 100 episodes while our algorithm has -384. Although we cannot discuss whether it solved the environment, as it mentioned OpenAI Gym does not define a solution, we can compare our results with the leaderboards. The worst reward is given as -153 in the leaderboards, our result is not as good as this but we obviously have some caveats [17]. Since we are using solely Numpy, including our networks, we cannot fully optimize our results compared to ones that use state of the art algorithms. However, this is still a good indication that our algorithm can successfully learn, and is correct in its implementation.

2) Bipedal Walker: The bipedal walker environment has 24 features on its state space and 4 features on its action space. This action space is also continuous. The agent tries to learn how to walk on the uneven environment. The solution to this environment is given as an average of 300 points over 100 episodes [17]. The episode to reach this amount goes up to 5000 episodes, which should be noted for the upcoming discussion.

As it can be observed, our implementation of DDPG fails to solve the problem in the given constraints and is similar to the random agent. Our algorithms has -104 reward average over 100 episodes and random agent has -100. There are several reasons that this might be occurring.In general, DDPG algorithms are trained over very long periods of time, for simple examples this is around 5000 episodes, for bigger state and action spaces this value increases to 28000 [17], [27]. We simply do not have the resources for very long episodes. These resources are state of the art network libraries and GPU's, which significantly decrease training time. Our implementation of the Adam optimization algorithm was one solution to decrease the training time. Although it achieved this goal, it wasn't extremely noticeable. Our algorithm trained 2000 episodes in 12 hours. Perhaps one solution to this problem would be to use methods such as batch normalization to try to decrease the training time, which is one of the recommended methods to apply in the DDPG paper [14]. Although we couldn't get great results, we were able to capture the agent somewhat walking in a few cases.

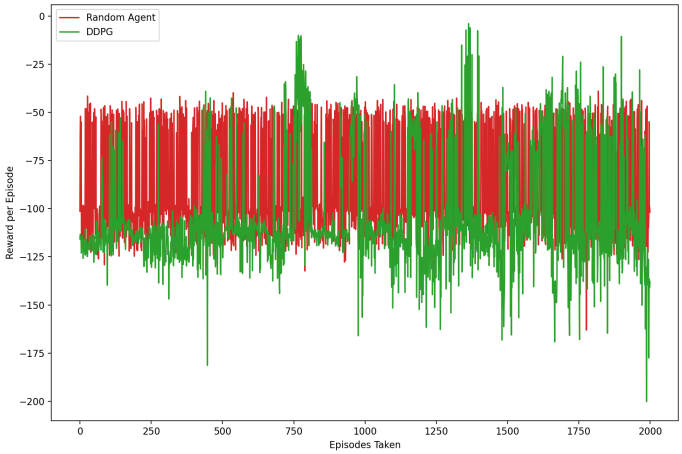


Figure 11. Reward per Episode for DDPG and Random Agent on the Bipedal Walker Environment.

3) Comparison and Insights: DDPG is one of the first RL algorithms to challenge problems containing continuous action spaces. It still is very efficient, although as it was discussed it requires optimal networks to achieve the best results. For environments that have small state and action spaces such as the swinging pendulum environment, our DDPG implementation works very well, but for more complex problems it falls short. It seems that DDPG, even with state of the art implementations, can fail to find an optimal solution and gets stuck in a poor solution [28]. This is something that we encountered while working with the bipedal walker environment, where the walker would attempt to do the same action at each episode at times, which we were able to observe by rendering the environment.

IV. Conclusion

For this project, we explored various different reinforcement learning algorithms and evaluated their performances on different game environments provided by Open AI Gym.

First, we implemented Episodic Semi Gradient SARSA as an introduction to reinforcement learning, which allowed us to understand Bellman Equation better and how learning can happen. In addition to SARSA algorithm itself, in order to use continuous action spaces, we feature construction methods (RBF).

For Deep Q-Learning we first implemented the neural network using only Numpy library and tested it in modular fashion. In addition to to using the neural network in the algorithm which utilized experience replay and the target network, we did more to increase the performance of the network. Implementing Adam optimizer, Dropout regularization (later omitted), momentum technique lead us to understand the neural network architecture better.

For Deep Deterministic Policy Gradient, we tried to understand a sophisticated algorithm that can be used for continuous action spaces, which was not possible for SARSA and DQN. The information we gained on experience replay and target network during DQN helped us in this part. Also, implementing complicated loss functions for this part let us see how the algorithms can be modified to give better results.

Finally, using different algorithms for one learning algorithm allowed us to see how its performance and requirements differ when the state and action space change. In addition, using the same environment for different algorithms (Cartpole for SARSA and DQN) made us compare the algorithms. Overall, thanks to this project we were able to work on different reinforcement learning algorithms and understand them.

REFERENCES

[1] "Gym," https://gym.openai.com/, (Accessed on 04/02/2021).

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book.html

[3] V. Kumar, "Reinforcement learning: Temporal-difference, sarsa, q-learning amp; expected sarsa on python," Oct 2019. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e

[4] G. Konidaris, S. Osentoski, and P. Thomas, "Value function approximation in reinforcement learning using the fourier basis," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, ser. AAAI'11. AAAI Press, 2011, p. 380–385.

[5] J. Chen, "Neural network definition," Dec 2020. [Online]. Available: https://www.investopedia.com/terms/n/neuralnetwork.asp

[6] "Deep q-learning tutorial: mindqn. a practical guide to deep q-networks | by mike wang | towards data science," https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc, (Accessed on 04/02/2021).

[7] "What is the difference between off-policy and on-policy learning? - cross validated," https://stats.stackexchange.com/questions/184657/what-is-the-difference-between-off-policy-and-on-policy-learning#:~:text=%22An%20off%2Dpolicy%20learner%20learns,agent%20including%20the%20exploration%20steps.%22, (Accessed on 05/07/2021).

[8] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *CoRR*, vol. abs/1811.12560, 2018. [Online]. Available: http://arxiv.org/abs/1811.12560

[9] "Deep q-network (dqn)-i. openai gym pong and wrappers | by jordi torres.ai | towards data science," https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af, (Accessed on 04/02/2021).

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[11] "Mit 6.s091: Introduction to deep reinforcement learning (deep rl) - youtube," https://www.youtube.com/watch?v=zR11FLZ-O9M, (Accessed on 04/02/2021).

[12] L. Weng, "Policy gradient algorithms," *lilianweng.github.io/lil-log*, 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

[13] "Gym," https://gym.openai.com/envs/#mujoco, (Accessed on 04/02/2021).

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016, cite arxiv:1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540

[16] C. Atkinson, B. McCane, L. Szymanski, and A. V. Robins, "Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting," *CoRR*, vol. abs/1812.02464, 2018. [Online]. Available: http://arxiv.org/abs/1812.02464

[17] "Leaderboard · openai/gym wiki · github," https://github.com/openai/gym/wiki/Leaderboard, (Accessed on 05/07/2021).

[18] G. J. Gordon, "Stable function approximation in dynamic programming," in *Machine Learning Proceedings 1995*, A. Prieditis and S. Russell, Eds. San Francisco (CA): Morgan Kaufmann, 1995, pp. 261–268. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558603776500402

[19] R. Sutton, "Introduction to reinforcement learning with function approximation," in *Tutorial at the Conference on Neural Information Processing Systems*, 2015, p. 33.

[20] J. Brownlee, "10 standard datasets for practicing applied machine learning," May 2020. [Online]. Available: https://machinelearningmastery.com/standard-machine-learning-datasets/

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[22] "mindqn/mindqn.py at main · mswang12/mindqn," https://github.com/mswang12/minDQN/blob/main/minDQN.py, (Accessed on 05/07/2021).

[23] "Deep deterministic policy gradient (ddpg)," https://keras.io/examples/rl/ddpg_pendulum/, (Accessed on 05/07/2021).

[24] "Ddpg_numpy_only/ddpg_numpy.py at 127feb93179d9bf1b836cf252b70ab965a0ce891 · kcg2015/ddpg_numpy_only · github," https://github.com/kcg2015/DDPG_numpy_only/blob/127feb93179d9bf1b836cf252b70ab965a0ce891/ddpg_numpy.py, (Accessed on 05/07/2021).

[25] "policy-value-methods/ddpg at master · qasimwani/policy-value-methods · github," https://github.com/QasimWani/policy-value-methods/tree/master/DDPG, (Accessed on 05/07/2021).

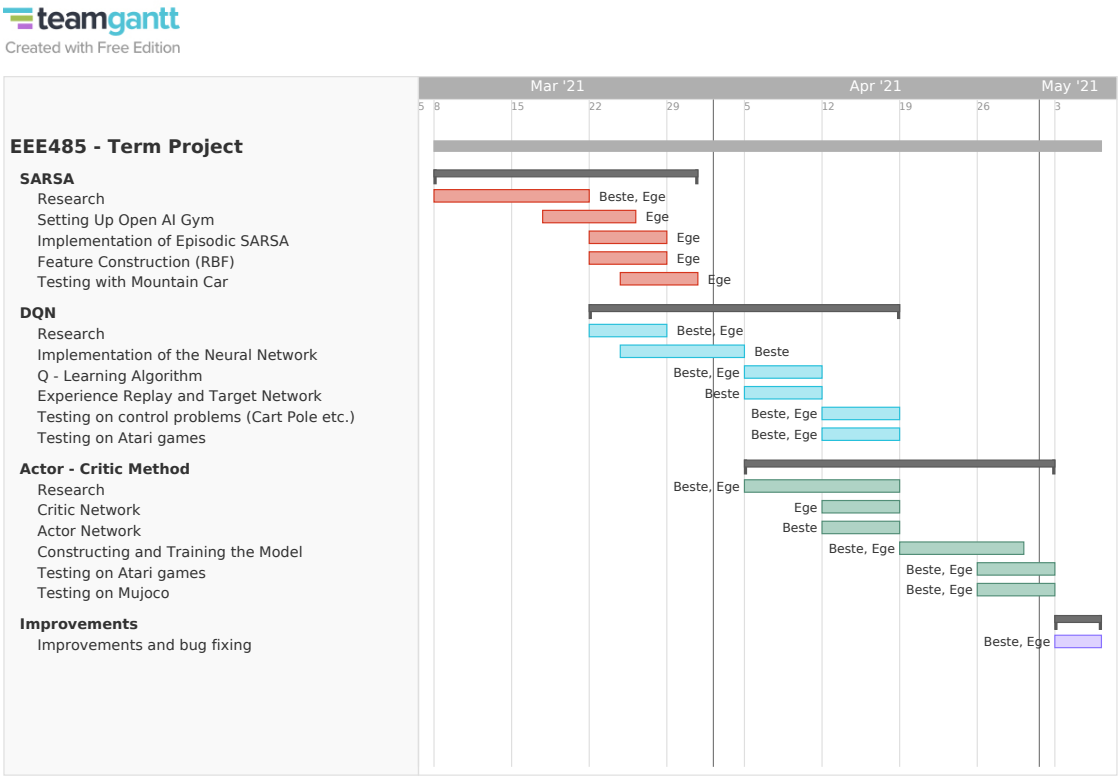
[26] "Teach your ai how to walk | solving bipedalwalker | openaigym | by shiva verma | towards data science," https://towardsdatascience.com/teach-your-ai-how-to-walk-5ad55fce8bca, (Accessed on 05/07/2021).

[27] A. Kumar, N. Paul, and S. N. Omkar, "Bipedal walking robot using deep deterministic policy gradient," 2018.

[28] G. Matheron, N. Perrin, and O. Sigaud, "The problem with ddpg: understanding failures in deterministic environments with sparse rewards," 2019.

V. Appendix

A. Gantt Chart



B. Algorithms

Algorithm 1: Episodic Semi Gradient SARSA update $\hat{q} \approx q_*$

```
Set weights w ← 0;
for ep ← 0 to final episode do
    Reset environment;
    Observe S and choose A using policy and observed state;
    while True do
        Take step: Commit to action A, observe next state S′, reward R and termination;
        if Termination then
            update(S, A, R);
            break;
        else
            Get next action A′ following policy;
            update(S, A, R, S′, A′);
            S ← S′;
            A ← A′;
        end
    end
end
```

Algorithm 2: $\varepsilon-$greedy policy, action selection algorithm

Input: Q function $\hat{q}$, state S and exploration rate ε;
$r \leftarrow$ random from [0, 1];
if $r < \varepsilon$ **then**
    $A \leftarrow$ random action;
else
    $A \leftarrow$choose action corresponding to $max_a \hat{q}(S, ; w)$;
end

Algorithm 3: Deep Q-learning with Experience Replay [21]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for *episode = 1 **to** M* **do**
  for $t = 1$ ***to*** T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t in emulator and observe reward r_t and state s_{t+1}
    Set $s_{t+1} = s_t, a_t, s_{t+1}$
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D

$$y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$$

    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
  end
end

Algorithm 4: DDPG Algorithm [14]

Randomly initialize critic network $Q(s, a \mid \theta^Q)$ and actor $\mu(s \mid \theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for *ep = 0 **to** M* **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $t = 1$ ***to*** T **do**
    Select action $a_t = \mu(s_t \mid \theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'\left(s_{i+1}, \mu'\left(s_{i+1} \mid \theta^{\mu'}\right) \mid \theta^{Q'}\right)$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i \left(y_i - Q\left(s_i, a_i \mid \theta^Q\right)\right)^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q\left(s, a \mid \theta^Q\right) \Big|_{s=si, a=\mu(s_i)} \nabla_{\theta^\mu} \mu\left(s \mid \theta^\mu\right) \Big|_{si}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end
end

C. Sarsa

```python
import numpy as np
import pickle

# todo: we should implement the tiling thing in the book for continuous action spaces

class Sarsa:

    def __init__(self, state_dim, action_dim, basis_function):
        """
        :param state_dim: Dimension of state vector
        :param action_dim: The number of actions available in environment
        :param basis_function: An object, RBF
        """

        self.basis_function = basis_function
        self.W = np.zeros((self.basis_function.order ** state_dim, action_dim))  # weight vector, see Ch.10 or 9,



    def update(self, state, action, reward, next_state=None, next_action=None, learning_rate=0.5, discount=0.5, terminate=False):
        """
        The SARSA update rule, updates the weight of corresponding action using S, A, R, S', A'
        :param state: current state
        :param action: current action
        :param reward: reward obtained after committing current action
        :param next_state: state observed after committing current action
        :param next_action: next action chosen on-policy
        :param learning_rate: alpha, the learning rate
        :param discount: gamma, generally 1
        :param terminate: whether the environment is at termination
        """


        q_approx_grad = self.transform(state)  # wrote it like this so that its more understandable

        if terminate:
            update_target = reward - self.q_approx(state, action)
        else:
            update_target = reward + discount * self.q_approx(next_state, next_action) - self.q_approx(state, action)

        self.W[:, action] += learning_rate * update_target * q_approx_grad



    def q_approx(self, state, action=None):
        """
        Approximate Q function, defined by the linear equation W.T @ x
        :param state: state vector
        :param action: discrete action
        :return: the approximated q-value
        """

        x = self.transform(state)

        # print(self.W)
        if action is None:
            return self.W.T @ x
```

```
        else:
            return self.W[:, action].T @ x


    def transform(self, state):
        """
        Constructs features following the given basis function (RBF for now)
        :param state: the state vector
        :return: the constructed features
        """

        return self.basis_function.transform(state)




    def save_weights(self, dir="../weights/Sarsa"):
        """
        A function to save weights of SARSA
        :param dir: dierctory
        """
        np.save(f"{dir}/W", self.W)
```

D. Radial Basis Function

```python
import numpy as np


class RBF:

    def __init__(self, order, state_dim, low, high):
        """
        Radial Basis Function for feature construction
        :param order: n divisions for each dimension of the state vector
        :param state_dim: dimension of state vector
        :param low: min values of state features, for normalization
        :param high: max values of state features, for normalization
        """

        self.order = order
        self.state_dim = state_dim
        self.low = low.astype('float64')
        self.high = high.astype('float64')
        self.centers = None
        if order == 1:
            self.variance = 1
        else:
            self.variance = 2 / (order - 1)
        self.create_centers()


    def create_centers(self):
        """
        Creates centers of each RBF, centers are evenly distributed
        """

        partition = np.linspace(0, 1, self.order)

        grid = np.meshgrid(*([partition]*self.state_dim))
        centers = np.vstack(map(np.ravel, grid)).T

        self.centers = centers.tolist()


    def transform(self, state):
        """
        Creates features
        :param state: the state vector from which the features will be created
        :return: constructed features
        """

        state = self.normalize(state)

        features = np.empty(self.order ** self.state_dim)  # todo: 1d vector, possible problem creator

        for index, c in enumerate(self.centers):
            norm = np.linalg.norm(state - c)**2
            features[index] = np.exp(-norm/(2 * self.variance))


        features = np.where(features > 0.95, 1, 0)

        return features


```

```
61
62      def normalize(self, state):
63          """
64          Normalizes given input
65          :param state: the state vector from which the features will be created
66          :return: normalized state vector
67          """
68          # return np.exp(state) / (1 + np.exp(state))
69          return (state - self.low)/(self.high - self.low)
```

E. Neural Network

```
import numpy as np
from src.NeuralNetwork.Layers.nn_toolkit import error_map


class Network(object):

    def __init__(self, layers, input_size, error_function=None, optimizer="sgd"):
        """
        Initializes important network parameters
        :param layers: Network layers
        :param input_size: input size of first layer and also the network
        :param error_function: MSE or CE
        """

        self.layers = layers
        self.input_size = input_size

        if error_function:
            self.error_function = error_map[error_function]

        next_size = self.input_size

        for layer in self.layers:
            next_size = layer.initialize(next_size, optimizer)



    def fit(self, X, Y, epoch, mini_batch_size, learning_rate):
        """
        Update network weights by forward and backward passes
        :param mini_batch_size:
        :param X: input
        :param Y: labels
        :param epoch: trials
        :param learning_rate: learning rate
        :return: a list of losses at each epoch
        """


        sample_size = X.shape[0]
        iter_per_epoch = int(sample_size / mini_batch_size)
        loss_list = []

        for i in range(epoch):

            start = 0
            end = mini_batch_size


            p = np.random.permutation(X.shape[0])
            shuffledX = X[p]
            shuffledY = Y[p]


            loss_sum = 0


            for it in range(iter_per_epoch):

                batchX = shuffledX[start:end]
                batchY = shuffledY[start:end]
```

```
                pred, loss = self._fit_instance(batchX, batchY, learning_rate)

                loss_sum += loss

                start = end
                end += mini_batch_size


            loss_list.append(loss_sum/iter_per_epoch)

        return pred, loss_list


    def predict(self, X):
        """
        Predict for given input using network weights
        :param X: input
        :return: the prediction
        """

        next_X = X

        for layer in self.layers:
            next_X = layer.predict(next_X)

        return next_X


    def _fit_instance(self, X, Y, learning_rate, config=None):

        prediction = self._call_forward(X)
        loss, residual = self._calculate_error(prediction, Y)
        self._call_backward(residual)
        self._call_update(learning_rate, config=config)

        return prediction, loss


    def _call_forward(self, X):

        next_X = X

        for layer in self.layers:
            next_X = layer.forward_pass(next_X)
        return next_X



    def _call_backward(self, residual):

        next_residual = residual

        for layer in reversed(self.layers):
            next_residual = layer.backward_pass(next_residual)


    def _call_update(self, learning_rate, config=None):

        for layer in self.layers:
            layer.update(learning_rate, config)


```

```
125
126     def _calculate_error(self, pred, actual):
127         error, residual = self.error_function(pred, actual)
128
129         return error, residual
```

F. Fully Connected Layer

```python
import numpy as np
from src.NeuralNetwork.Layers.nn_toolkit import activation_map


class FullyConnected(object):


    def __init__(self, nodes, activation):
        """
        Initializes important variables
        self.opv: private member of class, opv being an abbreviation for one pass variables

        :param nodes: output of layer
        :param activation: the activation function of layer
        """


        self.weight = None
        self.bias = None

        self.opv = {"X": None, "dW": None, "db": None, "dA": None, "mW": 0, "mb": 0, "sW": 0, "sb": 0}

        self.nodes = nodes
        self.activation = activation_map[activation]
        self.optimizer = None
        self.tanh = activation
        self.iteration = 1



    def initialize(self, input_size, optimizer):
        """
        Xavier Initialization for layer parameters
        :param input_size: input size of this layer
        :return: output size of this layer / layer nodes
        """

        self.optimizer = optimizer
        if self.tanh == "tanh":
            init = 3e-3
        else:
            init = np.sqrt(6/(input_size + self.nodes))
        self.weight = np.random.uniform(-init, init, size=(input_size, self.nodes))
        self.bias = np.zeros((1, self.nodes))


        return self.nodes



    def forward_pass(self, X):
        """
        Forward pass of layer
        :param X:
        :return:
        """
        # print(X.shape, self.weight.shape, self.bias.shape)
        if X.ndim is 1:
            X = X.reshape(1, -1)

```

```
        self.opv["X"] = X
        potential = X @ self.weight + self.bias
        phi, self.opv["dA"] = self.activation(potential)


        return phi


    def backward_pass(self, residual):
        """
        Backward pass of layer, gradient descent
        :param residual: The residual error gradient from the earlier layer
        :return: the next iteration residual
        """

        # print(self.nodes, residual.shape, self.opv["dA"].shape )
        # print(residual.shape, self.opv["X"].shape, self.weight.shape, self.opv["dA"])

        residual = residual * self.opv["dA"]  # updater residual term
        self.opv["dW"] = self.opv["X"].T @ residual
        self.opv["db"] = residual.sum(axis=0, keepdims=True)

        return residual @ self.weight.T




    def update(self, learning_rate, config):
        """

        Updates layer weight and bias

        dW: gradient of weight(t)
        db: gradient of bias(t)

        mW: momentum of weight(t)
        mb: momentum(t) of bias(t)


        :param learning_rate: learning rate for update
        :param momentum_rate: momentum_rate for updatte
        """

        # print("Weight:", self.weight.shape, self.weight.min(), self.weight.max(), "Bias:",
    self.bias.shape, self.bias.min(), self.bias.max())


        if self.optimizer == "sgd":

            if config is None:
                momentum_rate = 0.9
            else:
                momentum_rate = config

            self.opv["mW"] = learning_rate * self.opv["dW"] + momentum_rate * self.opv["mW"]
            self.opv["mb"] = learning_rate * self.opv["db"] + momentum_rate * self.opv["mb"]

            self.weight -= self.opv["mW"]
            self.bias -= self.opv["mb"]

        if self.optimizer == "adam":

```

```
            if config is None:
                beta1 = 0.9
                beta2 = 0.999
                epsilon = 1e-8
            else:
                beta1, beta2, epsilon = config

            self.opv["mW"] = (1 - beta1) * self.opv["dW"] + beta1 * self.opv["mW"]
            self.opv["mb"] = (1 - beta1) * self.opv["db"] + beta1 * self.opv["mb"]

            self.opv["sW"] = (1 - beta2) * np.square(self.opv["dW"]) + beta2 * self.opv["sW"]
            self.opv["sb"] = (1 - beta2) * np.square(self.opv["db"]) + beta2 * self.opv["sb"]

            # print((1 - beta1 ** self.iteration), (1 - beta2 ** self.iteration))
            #
            # self.opv["mW"] = self.opv["mW"] / (1 - beta1 ** self.iteration)
            # self.opv["mb"] = self.opv["mb"] / (1 - beta1 ** self.iteration)
            #
            # self.opv["sW"] = self.opv["sW"] / (1 - beta2 ** self.iteration)
            # self.opv["sb"] = self.opv["sb"] / (1 - beta2 ** self.iteration)


            self.weight -= learning_rate * self.opv["mW"]/(np.sqrt(self.opv["sW"]) + epsilon)
            self.bias -= learning_rate * self.opv["mb"]/(np.sqrt(self.opv["sb"]) + epsilon)

            self.iteration += 1



    def predict(self, X):
        """
        Predicts given output using layer weight and bias
        :param X: Input vector/matrix
        :return: the predicted value
        """

        potential = X @ self.weight + self.bias
        return self.activation(potential)[0]
```

G. Dropout Layer

```
import numpy as np
from src.NeuralNetwork.Layers.nn_toolkit import activation_map




class Dropout(object):


    def __init__(self, nodes, activation, p = 1):
        """
        Initializes important variables
        self.__opv: private member of class, opv being an abbreviation for one pass variables

        :param nodes: output of layer
        :param activation: the activation function of layer
        """


        self.weight = None
        self.bias = None

        self.__opv = {"X": None, "dW": None, "db": None, "dA": None, "mW": 0, "mb": 0, "sW": 0,
     "sb": 0}

        self.nodes = nodes
        self.activation = activation_map[activation]
        self.optimizer = None
        self.iteration = 1

        self.p = p


    def initialize(self, input_size, optimizer):
        """
        Xavier Initialization for layer parameters
        :param input_size: input size of this layer
        :return: output size of this layer / layer nodes
        """

        self.optimizer = optimizer
        init = np.sqrt(6/(input_size + self.nodes))
        self.weight = np.random.uniform(-init, init, size=(input_size, self.nodes))
        self.bias = np.zeros((1, self.nodes))

        # self.weight = np.zeros((input_size, self.nodes))
        # self.bias = np.zeros((1, self.nodes))

        return self.nodes



    def forward_pass(self, X):
        """
        Forward pass of layer
        :param X:
        :return:
        """

        if X.ndim is 1:
            X = X.reshape(1, -1)
```

```
61
62          self.__opv["X"] = X
63          potential = X @ self.weight + self.bias
64          phi, self.__opv["dA"] = self.activation(potential)
65
66          drop = (np.random.rand(phi.shape[0], phi.shape[1]) < self.p)*1.0
67          phi *= drop / self.p
68          self.__opv["dA"] = self.__opv["dA"] * drop / self.p
69
70          return phi
71
72
73      def backward_pass(self, residual):
74          """
75          Backward pass of layer, gradient descent
76          :param residual: The residual error gradient from the earlier layer
77          :return: the next iteration residual
78          """
79
80          # print(self.nodes, residual.shape, self.__opv["dA"].shape )
81
82          residual *= self.__opv["dA"]  # updater residual term
83          self.__opv["dW"] = self.__opv["X"].T @ residual
84          self.__opv["db"] = residual.sum(axis=0, keepdims=True)
85
86          return residual @ self.weight.T
87
88
89
90
91      def update(self, learning_rate, config):
92          """
93
94          Updates layer weight and bias
95
96          dW: gradient of weight(t)
97          db: gradient of bias(t)
98
99          mW: momentum of weight(t)
100         mb: momentum(t) of bias(t)
101
102
103         :param learning_rate: learning rate for update
104         :param momentum_rate: momentum_rate for updatte
105         """
106
107
108         if self.optimizer == "sgd":
109
110             if config is None:
111                 momentum_rate = 0.1
112             else:
113                 momentum_rate = config
114
115             self.__opv["mW"] = learning_rate * self.__opv["dW"] + momentum_rate * self.__opv["
    mW"]
116             self.__opv["mb"] = learning_rate * self.__opv["db"] + momentum_rate * self.__opv["
    mb"]
117
118             self.weight -= self.__opv["mW"]
119             self.bias -= self.__opv["mb"]
120
121         if self.optimizer == "adam":
```

```
122
123             if config is None:
124                 beta1 = 0.9
125                 beta2 = 0.999
126                 epsilon = 1e-8
127             else:
128                 beta1, beta2, epsilon = config
129
130             self.__opv["mW"] = (1 - beta1) * self.__opv["dW"] + beta1 * self.__opv["mW"]
131             self.__opv["mb"] = (1 - beta1) * self.__opv["db"] + beta1 * self.__opv["mb"]
132
133             self.__opv["sW"] = (1 - beta2) * np.square(self.__opv["dW"]) + beta2 * self.__opv["
    sW"]
134             self.__opv["sb"] = (1 - beta2) * np.square(self.__opv["db"]) + beta2 * self.__opv["
    sb"]
135
136
137
138
139             # self.__opv["mW"] /= 1 - beta1 ** self.iteration
140             # self.__opv["mb"] /= 1 - beta1 ** self.iteration
141             #
142             # self.__opv["sW"] /= 1 - beta2 ** self.iteration
143             # self.__opv["sb"] /= 1 - beta2 ** self.iteration
144
145
146             self.weight -= learning_rate * self.__opv["mW"]/(np.sqrt(self.__opv["sW"]) +
    epsilon)
147             self.bias -= learning_rate * self.__opv["mb"]/(np.sqrt(self.__opv["sb"]) + epsilon)
148
149             self.iteration += 1
150
151
152
153     def predict(self, X):
154         """
155         Predicts given output using layer weight and bias
156         :param X: Input vector/matrix
157         :return: the predicted value
158         """
159
160         potential = X @ self.weight + self.bias
161         return self.activation(potential)[0]
```

H. Activation Functions and Error Functions

```
import numpy as np

"""
"""

def relu(X):

    phi = X * (X > 0)
    grad = 1 * (X > 0)

    return phi, grad


def silu(X):

    sigX = sigmoid(X)[0]

    phi = X * sigX
    grad = sigX * (1 + X * (1 - sigX))

    return phi, grad


def tanh(X):

    phi = np.tanh(X)
    grad = 1 - phi ** 2

    return phi, grad


def sigmoid(X):

    phi = 1 / (1 + np.exp(-X))
    grad = phi * (1 - phi)

    return phi, grad


def linear(X):

    phi = X
    grad = 1

    return phi, grad


def cross_entropy(pred, actual):


    error = np.sum(- actual * np.log(pred)) / len(actual)

    residual = pred
    residual[actual == 1] -= 1
    residual /= len(actual)

    return error, residual



def mse(pred, actual):
```

```
62
63      # print("Pred:", pred.shape, pred.min(), pred.max(), "Actual:", actual.shape, actual.min(),
        actual.max())
64
65      if isinstance(actual, float):
66          size = 1
67      else:
68          size = len(actual)
69
70      error = ((actual - pred)**2).mean()
71
72      residual = (pred - actual)/size
73
74
75      return error, residual
76
77
78
79
80  activation_map = {"relu": relu, "silu": silu , "tanh": tanh, "sigmoid": sigmoid, "linear":
        linear}
81  error_map = {"MSE": mse, "CE": cross_entropy}
```

I. DQN

```python
import numpy as np
from collections import deque
from copy import deepcopy
from src.NeuralNetwork.Network import Network


class DQN:

    def __init__(self, state_dim, action_dim, layers, N):
        """

        :param state_dim:
        :param action_dim:
        :param layers:
        :param batch_size: Batch size for replay memory.
        :param N: Experience replay (pool of stored samples) capacity size (max)
        """

        self.network = Network(layers, state_dim, "MSE", optimizer="adam") # Huber loss, He
    init
        self.target_network = deepcopy(self.network)
        self.replay_memory = deque(maxlen=N)


    def update(self, mini_batch, learning_rate=0.1, discount=1,
               terminate=False):
        """
        The DeepSARSA update rule, updates the weights of the neural network
        :param state: current state
        :param action: current action
        :param reward: reward obtained after committing current action
        :param next_state: state observed after committing current action
        :param next_action: next action chosen on-policy
        :param learning_rate: alpha, the learning rate
        :param discount: gamma, generally 1
        :param terminate: whether the environment is at termination
        """
        # print("mini-batch")
        # print(mini_batch)

        states = np.array([transition[0] for transition in mini_batch])
        next_states = np.array([transition[3] for transition in mini_batch]) # actionsize

        q_values = self.q_approx(states)
        next_q_values = self.q_approx(next_states, target=True)

        rate = 0.9
        #rate = 0.9
        # Find y_i
        X = []
        Y = []
        for index, (state, action, reward, next_state, terminate) in enumerate(mini_batch):
            if not terminate:
                max_future_q = reward + discount * np.max(next_q_values[index])
            else:
                max_future_q = reward

            # Y.append(max_future_q)
            # X.append(state)

            current_qs = q_values[index]
```

```
            current_qs[action] = (1 - rate) * current_qs[action] + rate * max_future_q

            X.append(state)
            Y.append(current_qs)
        x = np.array(X)
        # print(x.shape)
        y = np.array(Y)
        # print(y.shape)
        _, losses = self.network.fit(x, y, epoch=1, mini_batch_size=len(mini_batch),
    learning_rate=learning_rate)

        return losses[0]



    def q_approx(self, state, action=None, target=False):
        """
        Return action.

        :param state: state vector
        :param action: discrete action
        :return: the approximated q-value
        """

        a = self.network.predict(state)


        if target is False:
            if action is None:
                return a
            else:
                return self.network.predict(state)[:,action] #??
        else:
            if action is None:
                return self.target_network.predict(state)
            else:
                return self.target_network.predict(state)[:,action] #??



    def update_target(self):
        self.target_network = deepcopy(self.network)
```

J. DDPG

```
import random

from src.NeuralNetwork.Network import Network
from src.NeuralNetwork.Layers import FullyConnected
from src.Agents.DDPG.Actor import Actor
from src.Agents.DDPG.Critic import Critic
import numpy as np
import copy
from collections import deque


class DDPG(object):


    def __init__(self, state_dim, action_dim, actor_layers, critic_layers, state_layers,
    action_layers, env):

        print(env.action_space.high)
        self.actor = Actor(actor_layers, state_dim, "adam", env.action_space.high)
        self.critic = Critic(critic_layers, state_layers, action_layers, state_dim, action_dim,
     "adam")
        self.env = env
        self.buffer_size = 1000000
        self.batch_size = 128
        self.replay_buffer = deque()
        self.epsilon = 0.9
        self.action_dim = action_dim



    def train(self, episodes, actor_learning_rate, critic_learning_rate, discount_rate, tau,
    max_steps, info_tuple, render=False):

        reward_per_episode, Q_loss, policy_loss = info_tuple

        for ep in range(episodes):
            # get initial state and action (via policy)
            state = self.env.reset()
            i = 1
            rewards_sum = 0
            Q_loss_sum = 0
            policy_loss_sum = 0
            self.epsilon *= 0.993
            actor_learning_rate *= 0.99995
            critic_learning_rate *= 0.99995

            while True:  # loop controlled with termination of state, run until done

                if render is True and ep > 500:  # for visualization
                    self.env.render()

                if np.random.random() < self.epsilon:
                    action = np.clip(self.actor.predict(state.T) + np.random.normal(self.
    action_dim), -1, 1) * self.env.action_space.high
                else:
                    action = self.actor.predict(state.T)


                next_state, reward, terminate, info = self.env.step(action.flatten())  # commit
     to action

```

```
                self.add_replay((state.flatten(), action.flatten(), reward, next_state.flatten(), terminate))

                if len(self.replay_buffer) > self.batch_size:

                    states, actions, rewards, next_states, dones = self.get_replay_batch()

                    next_actions = self.actor.target_predict(next_states)
                    target_Q = self.critic.target_predict((next_states, next_actions))
                    Y = np.empty((self.batch_size, 1))

                    # Y = rewards + discount_rate * target_Q
                    # Y[dones] = rewards[dones]

                    for j in range(len(Y)):
                        if not dones[j]:
                            Y[j] = rewards[j] + discount_rate * target_Q[j]
                        else:
                            Y[j] = rewards[j]

                    # print("Y:", Y.shape, Y.min(), Y.max())


                    critic_loss = self.critic.fit(states, actions, Y, critic_learning_rate)

                    action_prediction = self.actor.predict(states)
                    grad, critic_prediction, actor_loss = self.critic.action_grad(states, action_prediction, self.batch_size)
                    self.actor.update(grad, critic_prediction, actor_learning_rate)

                    self.critic.target_update(tau)
                    self.actor.target_update(tau)

                    Q_loss_sum += critic_loss
                    policy_loss_sum += actor_loss


                # print(f"Step: {i}")
                i += 1
                rewards_sum += reward
                state = next_state

                if terminate or i > max_steps:
                    break


            Q_loss.append(Q_loss_sum/i)
            policy_loss.append(policy_loss_sum/i)
            reward_per_episode.append(rewards_sum)
            print(f"Episode ended: {ep}\nReward total: {rewards_sum}\nSteps: {i}\n")


        # return reward_per_episode, Q_loss, policy_loss


    def add_replay(self, transition):

        self.replay_buffer.append(transition)

        if len(self.replay_buffer) > self.buffer_size:
```

```
            self.replay_buffer.popleft()


    def get_replay_batch(self):
        batch = random.sample(self.replay_buffer, self.batch_size)
        # s, a, r, sn, d = zip(*batch)
        # out = np.asarray(s), np.asarray(a), np.asarray(r).reshape(-1, 1), np.asarray(sn), np.asarray(d).reshape(-1, 1)

        s = np.asarray([sample[0] for sample in batch])
        a = np.asarray([sample[1] for sample in batch])
        r = np.asarray([sample[2] for sample in batch])
        sn = np.asarray([sample[3] for sample in batch])
        d = np.asarray([sample[4] for sample in batch])

        out = s, a, r, sn, d
        return out



    def play(self, episodes, max_steps, render=False):

        reward_per_episode = []

        for ep in range(episodes):
            # get initial state and action (via policy)
            state = self.env.reset()
            i = 1
            rewards_sum = 0


            while True:  # loop controlled with termination of state, run until done

                if render is True:  # for visualization
                    self.env.render()

                action = self.actor.predict(state.T)
                next_state, reward, terminate, info = self.env.step(action.flatten())  # commit to action
                state = next_state
                # print(f"Step: {i}")
                i += 1
                rewards_sum += reward


                if terminate or i > max_steps:
                    break


            reward_per_episode.append(rewards_sum)
            print(f"Episode ended: {ep}\nReward total: {rewards_sum}\nSteps: {i}\n")

        return reward_per_episode
```

K. Actor

```python
from src.NeuralNetwork.Network import Network
import copy


class Actor(Network):

    def __init__(self, layers, input_size, optimizer, action_bound):
        # print(input_size)
        super().__init__(layers, input_size, "MSE", optimizer=optimizer)
        self.target_layers = copy.deepcopy(self.layers)
        self.action_bound = action_bound



    def update(self, grad, state, learning_rate):

        super()._call_backward(self.action_bound * grad) # todo: maybe minus grad
        self._call_update(learning_rate)


    def predict(self, X):
        return super()._call_forward(X) * self.action_bound



    def target_predict(self, X):

        next_X = X

        for layer in self.target_layers:
            next_X = layer.predict(next_X)

        return next_X * self.action_bound

    def target_update(self, tau):
        for target_layer, layer in zip(self.target_layers, self.layers):
            target_layer.weight = (1 - tau) * target_layer.weight + tau * layer.weight
            target_layer.bias = (1 - tau) * target_layer.bias + tau * layer.bias
```

L. Critic

```python
from src.NeuralNetwork.Network import Network
import copy
import numpy as np


class Critic(Network):

    def __init__(self, layers, state_layers, action_layers, state_dim, action_dim, optimizer):



        self.state_layers = state_layers
        state_next_size = state_dim
        for state_layer in self.state_layers:
            state_next_size = state_layer.initialize(state_next_size, optimizer)

        self.action_layers = action_layers
        action_next_size = action_dim
        for action_layer in self.action_layers:
            action_next_size = action_layer.initialize(action_next_size, optimizer)

        self.state_dim = state_next_size
        self.action_dim = action_next_size

        super().__init__(layers, action_next_size + state_next_size , "MSE", optimizer=
    optimizer)


        self.target_layers = copy.deepcopy(self.layers)
        self.target_state_layers = copy.deepcopy(self.state_layers)
        self.target_action_layers = copy.deepcopy(self.action_layers)


    # def fit(self, next_action, replay_batch, learning_rate, discount_rate, tau=0.01):
    #
    #     state, action, reward, next_state, done = replay_batch
    #     reward = np.reshape(reward, (-1, 1))
    #     target_Q = self.target_predict((next_state, next_action))
    #     Y = np.empty_like(action)
    #
    #     for i in range(len(Y)):
    #         if not done[i]:
    #             Y[i] = reward[i] + discount_rate * target_Q[i]
    #         else:
    #             Y[i] = reward[i]
    #
    #     print("Y:", Y.shape, Y.min(), Y.max())


    def fit(self, state, action, Y, learning_rate, a=None):
        # print(state.shape, action.shape, Y.shape)
        _, loss = super()._fit_instance((state, action), Y, learning_rate)

        return loss




    def predict(self, X):
        ns, na = X

```

```
        for state_layer in self.state_layers:
            ns = state_layer.predict(ns)

        for action_layer in self.action_layers:
            na = action_layer.predict(na)

        concat = np.hstack((ns, na))
        # concat = ns + na

        next_X = concat

        for layer in self.layers:
            next_X = layer.predict(next_X)

        return next_X


    def target_predict(self, X):
        ns, na = X

        for state_layer in self.target_state_layers:
            ns = state_layer.predict(ns)

        for action_layer in self.target_action_layers:
            na = action_layer.predict(na)
        #
        concat = np.hstack((ns, na))
        # concat = ns + na

        next_X = concat

        for layer in self.target_layers:
            next_X = layer.predict(next_X)

        return next_X




    def action_grad(self, state, action, N):


        out = self._call_forward((state, action))
        loss = out.sum()

        next_residual = np.ones_like(out)/N

        for layer in reversed(self.layers):
            next_residual = layer.backward_pass(next_residual)

        action_residual = next_residual[:, self.state_dim:]
        # action_residual = next_residual

        for action_layer in reversed(self.action_layers):
            action_residual = action_layer.backward_pass(action_residual)


        return -action_residual, out, loss



    def _call_forward(self, X):
```

```
        ns, na = X

        for state_layer in self.state_layers:
            ns = state_layer.forward_pass(ns)

        for action_layer in self.action_layers:
            na = action_layer.forward_pass(na)


        concat = np.hstack((ns, na))
        # concat = ns + na


        next_X = concat
        for layer in self.layers:
            next_X = layer.forward_pass(next_X)


        return next_X




    def _call_backward(self, residual):

        next_residual = residual
        # print(residual.shape)

        for layer in reversed(self.layers):
            next_residual = layer.backward_pass(next_residual)


        state_residual, action_residual = next_residual[:, :self.state_dim], next_residual[:, self.state_dim:]
        # state_residual = next_residual
        # action_residual = next_residual


        for state_layer in reversed(self.state_layers):
            state_residual = state_layer.backward_pass(state_residual)

        for action_layer in reversed(self.action_layers):
            action_residual = action_layer.backward_pass(action_residual)




    def _call_update(self, learning_rate, config=None):

        for layer in self.layers:
            layer.update(learning_rate, config)

        for layer in self.state_layers:
            layer.update(learning_rate, config)

        for layer in self.action_layers:
            layer.update(learning_rate, config)


    def target_update(self, tau):

```

```
        for target_layer, layer in zip(self.target_layers, self.layers):
            target_layer.weight = (1 - tau) * target_layer.weight + tau * layer.weight
            target_layer.bias = (1 - tau) * target_layer.bias + tau * layer.bias

        for target_layer, layer in zip(self.target_state_layers, self.state_layers):
            target_layer.weight = (1 - tau) * target_layer.weight + tau * layer.weight
            target_layer.bias = (1 - tau) * target_layer.bias + tau * layer.bias

        for target_layer, layer in zip(self.target_action_layers, self.action_layers):
            target_layer.weight = (1 - tau) * target_layer.weight + tau * layer.weight
            target_layer.bias = (1 - tau) * target_layer.bias + tau * layer.bias
```

M. Training

```python
import random

import numpy as np
from src.Agents.SARSA.Sarsa import Sarsa
from src.Agents.SARSA.DeepSARSA import DeepSARSA
from src.Agents.DQN.DQN import DQN

import pickle


class Train:

    def __init__(self, env, model_name, model_config):
        """
        Initializes the model with given configuration to be trained on the given environment
        :param env: Training environment
        :param model_name: Model to be trained
        :param model_config: Configuration for the model
        """

        models = {"Sarsa": Sarsa, "DeepSARSA": DeepSARSA, "DQN": DQN}

        self.env = env

        if model_name is "Sarsa"or "DeepSARSA":
            self.model_type = "SARSA"
        elif model_name is "DQN":
            self.model_type = "DQN"

        print(f"State Space: {self.env.observation_space} \nAction Space: {self.env.action_space}")


        # todo: do some box/discrete checking here later

        self.model = models[model_name](**model_config)

    def play_sarsa(self, episodes, max_steps=None, render=False):

        for ep in range(episodes):

            # get initial state and action (via policy)
            state = self.env.reset()
            i = 1
            rewards_sum = 0

            while True:  # loop controlled with termination of state, run until done

                if render is True:  # for visualization
                    self.env.render()


                action = self.greedy_policy(state, 0)
                next_state, reward, terminate, info = self.env.step(action)  # commit to action
                state = next_state

                # to find average reward for the episode, hold a sum of all rewards and the steps taken
                i += 1
                rewards_sum += reward

```

```
            if terminate is True or i > max_steps:  # in termination, update using only current state-action, and break out of this episode
                break

            # update current state-action



        print(f"Episode ended: {ep}\nReward total: {rewards_sum}\nSteps: {i}\n")


def train(self, episodes, learning_rate, discount, epsilon, max_steps=None, decay=False, render=False):
    """
    Training function, loops for episodes, implemented using Sutton's RL book
    :param episodes: trial amount
    :param learning_rate: learning rate
    :param discount: discount, generally 1
    :param epsilon: the exploration rate, small or decaying
    :param max_steps: max steps per episode
    :param decay: condition on whether to decay exploration and learning rate or not
    :param render: condition which determines whether the game gets rendered or not
    :return: steps/reward per episode
    """

    reward_per_episode = []
    steps_per_episode = []

    for ep in range(episodes):

        # get initial state and action (via policy)
        state = self.env.reset()
        action = self.greedy_policy(state, epsilon)

        i = 1
        rewards_sum = 0

        if decay is True and epsilon > 0.1:  # decay
            epsilon *= 0.995
            # learning_rate *= 0.99995


        while True:  # loop controlled with termination of state, run until done

            if render is True and ep > 100:  # for visualization
                self.env.render()

            next_state, reward, terminate, info = self.env.step(action)  # commit to action

            # to find average reward for the episode, hold a sum of all rewards and the steps taken
            i += 1
            rewards_sum += reward

            if terminate is True or (max_steps is not None and i > max_steps):  # in termination, update using only current state-action, and break out of this episode
                self.model.update(state, action, reward, learning_rate=learning_rate, discount=discount, terminate=True)
                break

            if self.model_type is "SARSA":
                # else get next action using the next state, update following the SARSA rule
```

```
                    next_action = self.greedy_policy(next_state, epsilon)
                self.model.update(state, action, reward, next_state=next_state, next_action=
next_action,
                                  learning_rate=learning_rate, discount=discount)

                # update current state-action

                state = next_state
                action = next_action

            reward_per_episode.append(rewards_sum)
            steps_per_episode.append(i)

            print(f"Episode ended: {ep}\nReward total: {rewards_sum}\nSteps: {i}\n")



        return reward_per_episode, steps_per_episode


    def train_dqn(self, episodes, time_steps, C, min_replay_count, batch_size, learning_rate,
    discount, epsilon, epsilon_decay=0.95, lr_decay = 0.99, duration=30, lr_low=0, max_steps=
    None, decay=False, render=False):
        """
        Training function, loops for episodes, implemented using Sutton's RL book
        :param episodes: trial amount
        :param time_steps: number of time steps
        :param C: target network parameters are updated every C steps
        :param min_replay_count: Experience replay updates happen after this count
        :param batch_size: Size of the mini batch for experience replay update
        :param learning_rate: learning rate
        :param discount: discount, generally 1
        :param epsilon: the exploration rate, small or decaying
        :param max_steps: max steps per episode
        :param decay: condition on whether to decay exploration and learning rate or not
        :param render: condition which determines whether the game gets rendered or not
        :return: steps/reward per episode
        """

        reward_per_episode = []
        steps_per_episode = []
        loss_per_episode = []

        lrinit = learning_rate


        target_update_count = 0
        for ep in range(episodes):

            # get initial state and action (via policy)
            state = self.env.reset()


            i = 1
            rewards_sum = 0

            if decay is True and epsilon > 0.01 and learning_rate > lr_low:  # decay
                #epsilon *= epsilon_decay
                # print("epsilon", epsilon)
                #learning_rate *= lr_decay # landerda yok

                epsilon -= 0.99 / 300
                learning_rate -= 0.00015 / 300
```

```
177
178             loss = 0
179             #print(ep, "ep")
180
181
182             for t in range(time_steps):
183                 #print(t)
184                 target_update_count += 1
185
186                 if render is True and ep > 1450:  # for visualization
187                     self.env.render()
188
189                 # With probability epsilon select a random action a_t, otherwise select at =
    max_a Q(s_t, a;   )
190                 action = self.greedy_policy(state, epsilon)
191                 # print("action")
192                 # print(action)
193
194
195                 # Execute action at in emulator and observe reward r_t and s_t
196                 next_state, reward, terminate, info = self.env.step(action)  # commit to action
197
198
199                 # Store transition in replay memory
200                 self.model.replay_memory.append([state, action, reward, next_state, terminate])
      # s_t, a_t, r_t, s_(t+1)'
201
202                 state = next_state
203
204                 # to find average reward for the episode, hold a sum of all rewards and the
    steps taken
205                 i += 1
206                 rewards_sum += reward
207
208                 if terminate:
209                     break
210
211                 # Sample random mini batch from replay memory
212                 if len(self.model.replay_memory) < min_replay_count:
213                     continue
214
215
216
217                 #print("updating at", ep)
218                 mini_batch = random.sample(self.model.replay_memory, batch_size)
219
220                 loss += self.model.update(mini_batch, learning_rate, discount)
221
222
223                 # Update target network weights
224                 if target_update_count % C == 0:
225                     self.model.update_target()
226
227
228
229             loss_per_episode.append(loss/t)
230
231             reward_per_episode.append(rewards_sum)
232             steps_per_episode.append(i)
233
234             if (ep % 1 == 0):
235                 print(f"Episode ended: {ep}\nReward total: {rewards_sum}\nSteps: {i}\n")
236
```

```
            if (ep % 10 == 0 and ep < 300):
                print("epsilon", epsilon)


        # Save model


        return reward_per_episode, steps_per_episode, loss_per_episode


    def greedy_policy(self, state, epsilon):
        """
        Greedy policy, more explanation on the report
        :param state: state vector
        :param epsilon: exploration rate
        :return: next action
        """

        if np.random.rand() > epsilon:
            # print("state", state)
            # print("self.model.q_approx(state)")
            # print(self.model.q_approx(state))
            approx = self.model.q_approx(state)
            index = np.argmax(approx)
            return index
        else:
            return self.env.action_space.sample()


    def max_action(self, state):
        """
        Returns action that maximizes q function
        :param state: state vector
        :return: next action
        """
        return np.argmax(self.model.q_approx(state))
```

N. Random Agent

```
class RandomAgent(object):

    def __init__(self, env):

        self.env = env


    def play(self, episodes, max_steps, reward_per_episode, render=False):

            for ep in range(episodes):
                # get initial state and action (via policy)
                state = self.env.reset()
                i = 1
                rewards_sum = 0

                while True:  # loop controlled with termination of state, run until done

                    if render is True:  # for visualization
                        self.env.render()

                    action = self.env.action_space.sample()
                    next_state, reward, terminate, info = self.env.step(action)  # commit to
    action

                    i += 1
                    rewards_sum += reward

                    if terminate or i > max_steps:
                        break

                reward_per_episode.append(rewards_sum)
                if ep % 100 == 0:
                    print(f"Episode ended: {ep}\nReward total: {rewards_sum}\nSteps: {i}\n")

            return reward_per_episode
```

O. Testing Sarsa

```
from src.FeatureConstructors.RadialBasis import RBF
from src.NeuralNetwork.Layers import FullyConnected
from src.Train import Train
from matplotlib import pyplot as plt
import gym
import numpy as np
from src.Agents.RandomAgent import RandomAgent


env = gym.make("MountainCar-v0").env

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
state_low = env.observation_space.low
state_high = env.observation_space.high

#
# layers = [FullyConnected(8, "sigmoid"),
#           FullyConnected(action_dim, "silu")]
#
# deepsarsa_config = {"state_dim": state_dim, "layers": layers}

ra = RandomAgent(env)
aa = []
ra.play(1000, 500, aa, False)


rbf_order = 7
basis_function = RBF(rbf_order, state_dim, state_low, state_high)
sarsa_config = {"state_dim": state_dim, "action_dim": action_dim, "basis_function":
    basis_function}


trainer = Train(env, "Sarsa", sarsa_config)

re = []
se = []

for i in range(1):
    reward_per_episode, steps_per_episode = trainer.train(episodes=1000, learning_rate=0.05,
    discount=1, epsilon=1, max_steps=500, decay=True, render=False)
    re.append(reward_per_episode)
    se.append(steps_per_episode)

#
re = np.asarray(re)
se = np.asarray(se)

re = np.mean(re, axis=1)
se = np.mean(se, axis=1)



plt.figure(figsize=(12, 8), dpi=160)
plt.ylabel("Reward per Episode")
plt.xlabel("Episodes Taken")
plt.plot(aa, "C3", label="Random Agent")
plt.plot(reward_per_episode, "C2", label="Sarsa")
plt.legend()
# plt.show()
plt.savefig("../figures/finalreport/sarsa_mountaincar_reward.png",  bbox_inches = 'tight')
```

```
60
61 # trainer.model.save_weights()
```

P. Testing Network

```

import h5py

from src.NeuralNetwork.Network import Network
from src.NeuralNetwork.Layers import FullyConnected
import numpy as np
import matplotlib.pyplot as plt

# arr = np.genfromtxt('../assign2_data1.h5', delimiter=',')
# train = arr[:700]
# test = arr[700:]
# print(arr.shape)
# X = train[:, :-1]
# Y = train[:, -1].reshape(-1, 1)
#
# X_t = test[:, :-1]
# Y_t = test[:, -1].reshape(-1, 1)
#
# print(X.shape, Y.shape)


h5 = h5py.File("../assign2_data1.h5",'r')
trainims = h5['trainims'][()].astype('float64').transpose(0, 2, 1)
trainlbls = h5['trainlbls'][()].astype(int)
testims = h5['testims'][()].astype('float64').transpose(0, 2, 1)
testlbls = h5['testlbls'][()].astype(int)
h5.close()

trainlbls[np.where(trainlbls == 0)] = -1
testlbls[np.where(testlbls == 0)] = -1

X = np.reshape(trainims, (trainims.shape[0], trainims.shape[1] * trainims.shape[2]))
X = 1 * X/np.amax(X)
Y = np.reshape(trainlbls, (trainlbls.shape[0], 1))
X_test = np.reshape(testims, (testims.shape[0], testims.shape[1] * testims.shape[2]))
X_test = 1 * X_test/np.amax(X_test)
Y_test = np.reshape(testlbls, (testlbls.shape[0], 1))



layers = [FullyConnected(20, "tanh"), FullyConnected(5, "tanh"), FullyConnected(1, "tanh")]

net = Network(layers, X.shape[1], "MSE", optimizer="adam")
pred, loss = net.fit(X, Y, epoch=200, mini_batch_size=64, learning_rate=.0005)

test_p = net.predict(X_test)
acc = (np.sign(test_p) == Y_test).mean()

fig = plt.figure(figsize=(12, 8), dpi=80, facecolor='w', edgecolor='k')
fig.suptitle(f"Training MSE, Test Accuracy = {acc:.2f}")
plt.plot(loss, "C2")
plt.title("MSE")
plt.xlabel("Epoch")
# plt.show()
plt.savefig("../figures/report1/NN_testsgd.png",  bbox_inches = 'tight')
```

Q. Testing DQN - Breakout

```
import pickle

from src.FeatureConstructors.RadialBasis import RBF
from src.NeuralNetwork.Layers import FullyConnected
from src.NeuralNetwork.Layers import Dropout
from src.NeuralNetwork.Network import Network
from src.Train import Train
from matplotlib import pyplot as plt
import gym
import numpy as np
from copy import deepcopy
from utils.calc import window_avg
import atari_py
print(atari_py.list_games())

env = gym.make("Breakout-ram-v0")
#env = gym.make("CartPole-v0").env


state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
state_low = env.observation_space.low
state_high = env.observation_space.high


# Network
layers = [FullyConnected(128, "relu"), FullyConnected(128, "relu"), FullyConnected(128, "relu")
    , FullyConnected(128, "relu"), FullyConnected(action_dim, "linear")]
net = Network(layers, state_dim, "MSE", optimizer="adam") # Huber loss, He init

# Target Network
target_net = deepcopy(net)


# rbf_order = 6
# basis_function = RBF(rbf_order, state_dim, state_low, state_high)
# sarsa_config = {"state_dim": state_dim, "action_dim": action_dim, "basis_function":
    basis_function}

dqn_config = {"state_dim":state_dim, "action_dim":action_dim, "layers":layers, "N": 1000000}


trainer = Train(env, "DQN", dqn_config)

re = []
se = []
le = []

for iter in range(1):
    reward_per_episode, steps_per_episode, loss_per_episode = trainer.train_dqn(episodes=1500,
    time_steps=10000, C=100, min_replay_count=int(32),
                                                                batch_size=int(32), epsilon_decay
    =0.99, learning_rate=0.0002, lr_decay = 0.9972,  discount=0.99, epsilon=1, duration =
    100000, lr_low=0.00005,
                                                                max_steps=None, decay=True,
    render=True)
    re.append(reward_per_episode)
    se.append(steps_per_episode)
    le.append(loss_per_episode)

    if iter == 0:
```

```
        with open('dqn_breakout.pkl', 'wb') as output:
            trainer.model.replay_memory = None
            pickle.dump(trainer.model, output, pickle.HIGHEST_PROTOCOL)


#
np.save( "dqn_breakout_rewards", np.array(re))
np.save("dqn_breakout_steps", np.array(se))
np.save("dqn_breakout_loss", np.array(le))

re = np.asarray(re)
se = np.asarray(se)
le = np.asarray(le)

re = np.mean(re, axis=0)
se = np.mean(se, axis=0)
le = np.mean(le, axis=0)


plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Time Steps Taken")
plt.plot(se, "C3")
plt.show()
plt.savefig("dqn_breakout.png")

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.plot(re, "C4")
plt.show()
plt.savefig("sea_dqn.png")

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Loss")
plt.plot(le, "C5")
plt.show()
plt.savefig("sea_loss.png")

# trainer.model.save_weights()

# # Calculate the window average, to understand when it completes the task
# windowed = window_avg(np.array(re), window_size=100, threshold=195)
# print(windowed)
#
# plt.figure(figsize=(12, 8), dpi=160)
# plt.xlabel("Episode")
# plt.ylabel("Time Steps Taken")
# plt.plot(windowed[0], "C4")
# plt.show()
# plt.savefig("atari_dqn_windowed.png")
```

R. Testing DQN - Lunar Landing

```
import pickle

from src.FeatureConstructors.RadialBasis import RBF
from src.NeuralNetwork.Layers import FullyConnected
from src.NeuralNetwork.Layers import Dropout
from src.NeuralNetwork.Network import Network
from src.Train import Train
from matplotlib import pyplot as plt
import gym
import numpy as np
from copy import deepcopy
from utils.calc import window_avg
import atari_py
print(atari_py.list_games())

env = gym.make("Breakout-ram-v0")
#env = gym.make("CartPole-v0").env


state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
state_low = env.observation_space.low
state_high = env.observation_space.high


# Network
layers = [FullyConnected(128, "relu"), FullyConnected(128, "relu"), FullyConnected(128, "relu")
    , FullyConnected(128, "relu"), FullyConnected(action_dim, "linear")]
net = Network(layers, state_dim, "MSE", optimizer="adam") # Huber loss, He init

# Target Network
target_net = deepcopy(net)


# rbf_order = 6
# basis_function = RBF(rbf_order, state_dim, state_low, state_high)
# sarsa_config = {"state_dim": state_dim, "action_dim": action_dim, "basis_function":
    basis_function}

dqn_config = {"state_dim":state_dim, "action_dim":action_dim, "layers":layers, "N": 1000000}


trainer = Train(env, "DQN", dqn_config)

re = []
se = []
le = []

for iter in range(1):
    reward_per_episode, steps_per_episode, loss_per_episode = trainer.train_dqn(episodes=1500,
    time_steps=10000, C=100, min_replay_count=int(32),
                                                                batch_size=int(32), epsilon_decay
    =0.99, learning_rate=0.0002, lr_decay = 0.9972,  discount=0.99, epsilon=1, duration =
    100000, lr_low=0.00005,
                                                                max_steps=None, decay=True,
    render=True)
    re.append(reward_per_episode)
    se.append(steps_per_episode)
    le.append(loss_per_episode)

    if iter == 0:
```

```
        with open('dqn_breakout.pkl', 'wb') as output:
            trainer.model.replay_memory = None
            pickle.dump(trainer.model, output, pickle.HIGHEST_PROTOCOL)


#
np.save( "dqn_breakout_rewards", np.array(re))
np.save("dqn_breakout_steps", np.array(se))
np.save("dqn_breakout_loss", np.array(le))

re = np.asarray(re)
se = np.asarray(se)
le = np.asarray(le)

re = np.mean(re, axis=0)
se = np.mean(se, axis=0)
le = np.mean(le, axis=0)


plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Time Steps Taken")
plt.plot(se, "C3")
plt.show()
plt.savefig("dqn_breakout.png")

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.plot(re, "C4")
plt.show()
plt.savefig("sea_dqn.png")

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Loss")
plt.plot(le, "C5")
plt.show()
plt.savefig("sea_loss.png")

# trainer.model.save_weights()

# # Calculate the window average, to understand when it completes the task
# windowed = window_avg(np.array(re), window_size=100, threshold=195)
# print(windowed)
#
# plt.figure(figsize=(12, 8), dpi=160)
# plt.xlabel("Episode")
# plt.ylabel("Time Steps Taken")
# plt.plot(windowed[0], "C4")
# plt.show()
# plt.savefig("atari_dqn_windowed.png")
```

S. Testing DQN - Cart Pole

```
import pickle

from src.FeatureConstructors.RadialBasis import RBF
from src.NeuralNetwork.Layers import FullyConnected
from src.NeuralNetwork.Network import Network
from src.Train import Train
from matplotlib import pyplot as plt
import gym
import numpy as np
from copy import deepcopy
from utils.calc import window_avg
import atari_py



env = gym.make("CartPole-v0").env
env.seed(8)
env.observation_space.seed(8)
env.action_space.seed(8)
np.random.seed(7)


state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
state_low = env.observation_space.low
state_high = env.observation_space.high


# Network
layers = [FullyConnected(12, "relu"), FullyConnected(24, "relu"), FullyConnected(action_dim, "
    linear")]
net = Network(layers, state_dim, "MSE", optimizer="adam") # Huber loss, He init

# Target Network
target_net = deepcopy(net)


# rbf_order = 6
# basis_function = RBF(rbf_order, state_dim, state_low, state_high)
# sarsa_config = {"state_dim": state_dim, "action_dim": action_dim, "basis_function":
    basis_function}

dqn_config = {"state_dim":state_dim, "action_dim":action_dim, "layers":layers, "N": 80000}


trainer = Train(env, "DQN", dqn_config)

re = []
se = []
le = []

for iter in range(5):
    print("iter", iter)

    if iter == 2 or iter == 1:
        render = False
    else:
        render = False
    reward_per_episode, steps_per_episode, loss_per_episode = trainer.train_dqn(episodes=400,
    time_steps=200, C=4, min_replay_count=int(32),
```

```
                                                                batch_size=int(32), learning_rate
    =0.002, discount=0.95, epsilon=1, duration = 30, lr_low=0.0005,
                                                                max_steps=None, decay=True,
    render=render)


    if iter == 4:
        with open('dqn_cartpole_save.pkl', 'wb') as output:
            trainer.model.replay_memory = None
            pickle.dump(trainer.model, output, pickle.HIGHEST_PROTOCOL)

    print(np.array(reward_per_episode).mean())
    re.append(reward_per_episode)
    se.append(steps_per_episode)
    le.append(loss_per_episode)

#

np.save( "dqn_cartpole_rewards", np.array(re))
np.save("dqn_cartpole_steps", np.array(se))
np.save("dqn_cartpole_loss", np.array(le))

re = np.asarray(re)
se = np.asarray(se)
le = np.asarray(le)

re = np.mean(re, axis=0)
se = np.mean(se, axis=0)
le = np.mean(le, axis=0)


plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Time Steps Taken")
plt.plot(se, "C3")
plt.show()
plt.savefig("dqnstep.png")

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.plot(re, "C4")
plt.show()
plt.savefig("dqn.png")

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Loss")
plt.plot(le, "C5")
plt.show()
plt.savefig("loss.png")

# trainer.model.save_weights()

# Calculate the window average, to understand when it completes the task
windowed = window_avg(np.array(re), window_size=100, threshold=195)
print(windowed)

plt.figure(figsize=(12, 8), dpi=160)
plt.xlabel("Episode")
plt.ylabel("Time Steps Taken")
plt.plot(windowed[0], "C4")
plt.show()
```

```
120 plt.savefig("dqn_windowed.png")
```

T. Testing DDPG

```
import random

from src.Agents.DDPG.DDPG import DDPG
import gym
from src.NeuralNetwork.Layers.FullyConnected import FullyConnected
import pickle



env = gym.make("LunarLanderContinuous-v2").env
print(env.observation_space.shape, env.action_space.shape)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]



# actor_layers = [FullyConnected(256, "relu"), FullyConnected(256, "relu"), FullyConnected(action_dim, "tanh")]
# critic_layers = [FullyConnected(256, "relu"), FullyConnected(256, "relu"), FullyConnected(1, "relu")]
# state_layers = [FullyConnected(16, "relu"), FullyConnected(32, "relu")]
# action_layers = [FullyConnected(32, "relu")]

# actor_layers = [FullyConnected(600, "relu"), FullyConnected(300, "relu"), FullyConnected(600, "relu"), FullyConnected(action_dim, "tanh")]
# critic_layers = [FullyConnected(600, "relu"), FullyConnected(300, "relu"), FullyConnected(1, "linear")]
# state_layers = [FullyConnected(300, "relu"), FullyConnected(600, "relu"), FullyConnected(300, "linear")]
# action_layers = [FullyConnected(300, "relu"), FullyConnected(300, "linear")]


actor_layers = [FullyConnected(320, "relu"), FullyConnected(640, "relu"), FullyConnected(action_dim, "tanh")]
critic_layers = [FullyConnected(640, "relu"), FullyConnected(1, "linear")]
state_layers = [FullyConnected(320, "relu"), FullyConnected(640, "linear")]
action_layers = [FullyConnected(640, "linear")]

#
# actor_layers = [FullyConnected(300, "relu"), FullyConnected(600, "relu"), FullyConnected(action_dim, "tanh")]
# critic_layers = [FullyConnected(600, "relu"), FullyConnected(1, "linear")]
# state_layers = [FullyConnected(300, "relu"), FullyConnected(600, "linear")]
# action_layers = [FullyConnected(600, "linear")]

# actor_layers = [FullyConnected(64, "relu"),  FullyConnected(action_dim, "linear")]
# critic_layers = [FullyConnected(64, "relu"), FullyConnected(1, "relu")]
# state_layers = [FullyConnected(32, "relu")]
# action_layers = [FullyConnected(32, "relu")]


ddpg = DDPG(state_dim, action_dim, actor_layers, critic_layers, state_layers, action_layers, env)

episodes = 2000
actor_learning_rate = 0.0001
critic_learning_rate = 0.001
tau = 0.001
discount_rate = 0.99
max_steps = 1000

```

```
reward_per_episode = []
Q_loss = []
policy_loss = []

info_tuple = (reward_per_episode, Q_loss, policy_loss)

ddpg.train(episodes, actor_learning_rate, critic_learning_rate, discount_rate, tau, max_steps,
    info_tuple, render=False)

ddpg.replay_buffer = None

with open(f"../src/Agents/DDPG/saved_agents/agent{random.random()}", 'wb') as f:
    pickle.dump((ddpg, reward_per_episode, Q_loss, policy_loss), f)


#
# with open(f"../src/Agents/DDPG/saved_agents/agent47", 'rb') as f:
#     ddpg, _, _, _ = pickle.load(f)
```