Comparing Deep Reinforcement Learning Techniques on Cartpole Control

Waleed Bin Khalid, Logan Bowers, Keyang Lu, Seungmi Lee
Georgia Institute of Technology
Atlanta, USA
{wkhalid6,lbowers9,klu303,slee3490}@gatech.edu

Abstract

The CartPole problem serves as a classic control problem to test reinforcement learning methods. A common deep RL method is Deep Q-Learning, which seeks to optimize the expected next-state value of picking an action in a given state. Proximal Policy Optimization is another method that optimizes a policy (probability of choosing each possible action given a state) by using an Actor-Critic architecture. In this work, we tackle the classical Cartpole control challenge using deep reinforcement learning techniques. This report serves as an experimental comparison between several Deep Q-Learning Methods: including DQN, Double DQN, Dueling DQN, and CNN-based DQN, as well as Proximal Policy Optimization (PPO). The report describes the methods used in detail using both a theoretical and implementation lens. Each algorithm's performance is evaluated using loss, accuracy, reward, score, and Q-values. We also show the testing results via videos of the cartpole environment and calculate average testing scores for each method across multiple episodes. Our results after tuning include perfect scores for PPO as well as DDQN.

1. Introduction

Cartpole control is a classical control problem traditionally tackled through model-based control methods. In contrast, Reinforcement learning (RL) offers an end-to-end solution without the need for a mathematical model of the system. RL helps an agent learn optimal action strategies for a given task. For the given problem, an agent learns to balance a pole on a moving cart. This project aims to develop a survey paper that compares different algorithms for the Cartpole problem. The two algorithms that would be studied with their variations are the Deep Q-learning algorithm and the Proximal Policy Optimization algorithm.

The motivation behind the project is that we seek to provide a comprehensive report on RL methods tested in a controlled environment, and by doing so motivate future explorations into promising results, whether that be in the form of applications to other environments where reinforcement learning is appropriate or research into optimizing RL methods further. The goal was to not just solve a control problem, but also create a coding pipeline to study these fundamental methods in comparison.

2. Related Works

A paper on Deep Q-Learning trained several deep neural networks to estimate the Q-value functions and can play seven Atari 2600 games [2]. The model takes in inputs of images of the game and uses a simple, small-scale convolution neural network (CNN) with 4 layers to approximate Q-values for each available control. Algorithm 1 of the paper details the experience replay process with the explore-exploit strategy, which we use variations of in all of our DQN experiments. The proximal Policy Optimization method was introduced and demonstrated on the continuous domain of humanoid running and steering [3]. The paper includes comparisons to other policy network methods applied to the Atari environment of the previous paper, such as A2C and ACER.

We hope to combine the knowledge of these two papers to compare algorithms that optimize policy with the Q-learning algorithms that optimize reward functions. [6] compares Deep Q Networks and Policy Gradient methods on a 3D environment known as vizdoom, which is a simulation of the first-person shooter video game Doom. Experimentation of methods includes graphs of the reward functions as well as simulation-specific measures. We look to focus on environment measures as well such as score and reward, while putting more of a focus on comparing different DQN modifications as described in [1].

3. Methodology

This project is developed using the OpenAI Gym, Keras, and Tensorflow libraries (version 2.14.0) in Python 3.10.12. The training happens on Google Colab and testing happens

offline. Figure 1 shows the CartPole.

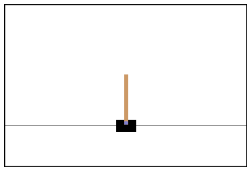


Figure 1. CartPole-v1 from OpenAI gym.

3.1. Dataset

The dataset for this project comes from the experiences the agent gains from interactions with the environments. The exact details of how the experiences are recorded are described for each method. Table 1 describes the state space. The action space comprises two values (0: push left, 1: push right). A reward of +1 is given if an action does not cause termination, otherwise, a reward of -100 is given for DQN methods and 0 is given for PPO methods. Termination occurs if the pole angle is outside of ±12°, the cart position is outside of ±2.4, or number of steps taken exceeds 500.

Observation	Min	Max
Cart Position	-4.8	4.8
Cart Velocity	$-\infty$	∞
Pole Angle	$-24°$	$24°$
Pole Angular Velocity	$-\infty$	∞

Table 1. Ranges for State-space parameters.

3.2. Deep Q-Learning (DQN)

DQN is an extension of the Q-learning method. A neural network is used to model the Q-value function which estimates the expected future rewards for a state-action pair (s, a). To train the model, we first make the cartpole take random actions in the environment to gather some memory. The memory stores data in the form of tuples: (s_t: current state,a_t: action, r_{t+1}: reward, s_{t+1}: new state, done: Termination Flag). The target Q-values for DQN are found using Equation 1:

$$Y_t = R_{t+1} + \gamma \max_a Q(s_{t+1}, a|\theta) \quad (1)$$

for non-terminal states. For terminal states, the Q-value is simply R_{t+1}. γ refers to the discount factor, which is used to maintain a balance between immediate and future rewards. The action selection is governed by an epsilon-greedy policy. At time t, a random action a_t is chosen with probability ϵ. An action = $\mathrm{argmax}\, Q\,(\mathrm{s_t}, \mathrm{a_t}|\theta)$ with probability 1 - ϵ. Once the memory is created, ϵ is decayed by a factor of ε_{decay}, to increase the probability of choosing the action predicted by the model.

3.3. Maximization Bias and Double-DQN (DDQN)

As another baseline implementation, we used the DDQN method [4]. A common pitfall with DQN is maximization bias. In the vanilla DQN, the target uses the Q network to both choose the "best" action and evaluate the learning target. As a result, the model tends to overestimate the future reward of certain actions, resulting in a positive bias. DDQN combats this bias by using two Q networks: one for action selection and another for action evaluation. We can rewrite the target to factor in selection and evaluation as shown in Equation 2.

$$Y_t = R_{t+1} + \gamma Q'(S_{t+1}, \mathrm{argmax}_a Q(S_{t+1}, a|\theta_t)|\theta'_t) \quad (2)$$

Here, we use the main learned network (Q, θ_t) for action selection and a target network (Q', θ'_t) for evaluation. The main network uses the evaluated target to update its weights, whereas the target network is updated less frequently at specified timesteps. In our implementation, we update the target network after each episode. We introduce a hyperparameter τ, which controls the extent to which the main network influences the target network on each update. The update function shown in Equation 3 uses linear interpolation.

$$\theta'_i = \theta'_{i-1} * (1 - \tau) + \theta_i * \tau \quad (3)$$

where τ is the interpolation parameter between the target network and the current network.

3.4. Dueling-DDQN

Another modification is Dueling DDQN [5]. This adds the concept of action advantage shown in Equation 4.

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \quad (4)$$

An action a has an advantage in state s represented by the Q value of choosing that action compared to the overall value of s independent of the next action. In the standard DQN architecture we've been using, the final hidden layer is a flat layer that outputs the scores of each action (in this case 2 actions corresponding to moving the cartpole left and right). For the dueling architecture, we replace this final layer with two separate layers: one estimating state value (V) and another estimating the action advantage of A, and then we use an aggregation layer to combine the two.

3.5. CNN-DQN

CNN-DQN is another approach that we targeted, where the state is the input image of size (400,600,3). In our implementation, we render the image through OpenAI Gymnasium's CartPole environment. To significantly reduce the input shape, we cut the top 160 pixels of the image, which

contains no relevant information, and downsized both dimensions of the image by a factor of 2. This gives us an input space of 120 pixels by 300 pixels. In addition, we normalize the image by setting all non-white pixels to black. This significantly reduces our state space to $\{0, 1\}^{120\times300}$.

3.6. Proximal Policy Optimization (PPO)

DQN methods, despite being a fundamental advancement in reinforcement learning, are subject to some issues. Firstly, they are restricted to discrete action spaces only. They have memory dependency to train. So essentially, without a long enough memory, DQN methods cannot be trained. It also has unstable behaviors at times. This can be because of the use of the same networks to select and evaluate actions and overestimating Q-values due to the max operator. We consulted the use of policy gradient methods for our control problem. Primarily, policy gradient methods update the parameters of a policy by gradient descent by using the gradient of the expected reward with respect to the policy parameters. A key algorithm in this domain is Proximal Policy Optimization (PPO) [3]

PPO works by maximizing an objective function while making sure that the new policy stays close to the original policy and does not diverge away heavily. It works by estimating the gradients of the expected returns with respect to the policy parameters and then updates these parameters in the direction that increases the returns. To avoid large policy updates it uses a clipped surrogate objective function which is shown in Equation 5.

$$L^{CLIP}(\theta) = E_t\left[\min\left(r_t(\theta)A_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)A_t\right)\right] \tag{5}$$

Where the ratio $r_t(\theta)$ is defined by Equation 6.

$$r_t(\theta) = \frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{old}}\left(a_t \mid s_t\right)} \tag{6}$$

$r_t(\theta)$ is essentially representing a probability ratio, that represents the likelihood of the new policy π_θ taking action a_t given state s_t as compared to the old policy $\pi_{\theta_{old}}$. The value of ϵ is a hyperparameter that is tuned to define the clipped range. By clipping the ratio, PPO achieves its purpose of not letting the policy deviate away so much. If the new policy becomes too different, then clipping it helps in staying close to the policy at the previous step. Clipping also achieves exploration and exploitation in PPO. It tries new actions (exploration) while also sticking to the old actions when needed (exploitation) which ensures stability. A_t is the advantage function at time t, which tells that the goodness of a specific action compared to the average actions taken. We calculate it using Generalized Advantage Estimation as shown in Equation 7. This helps in reducing the bias-variance trade-off and improves performance. γ is the discount factor and λ is a hyperparameter (weight) for adjusting δ_t.

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{7}$$

where δ_t is found using Equation 8,

$$\delta_t = r_t + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \tag{8}$$

where $V\left(s_t\right)$ is the output of the value network and represents the value function. PPO is also viewed from the lens of the actor-critic method. The actor in PPO represents the policy of the agent and is implemented as a neural network. The goal of the actor is to learn a policy that maximizes the expected rewards over time. It takes a state as input and outputs a probability distribution from which the actions are sampled. The Critic in PPO estimates the value of the states that the agent encounters. It evaluates the actor's actions by estimating how good an action taken in a particular state is. It takes the state as input and returns a single value representing the expected returns in the future. A buffer class is used for PPO to store the trajectory information that includes state information, actions, rewards, and termination checks. This buffer is created during each epoch. Then after the creation, it is used to train the models.

4. Experiments

For our initial experiments with the DQN models, we used the same neural network architecture. The input is the state vector of length 4. It is passed through hidden layers of 512, 256, and 64 neurons. Each hidden layer's weights are initialized through 'he_uniform' and the activation function used for each hidden layer is ReLU. The final layer is of size 2 and uses a linear action function with the same initializer for weights. All the algorithms were run till 100 episodes. We used a discount rate (γ) of 0.95 and a ϵ of 1.0. The value for τ for double DQN is set as 1, which means that we use a hard update for the target network. Memory is kept at a maximum length of 2000 and training starts when the memory reaches a size of 1000. The minimum value ϵ is kept at 0.01 and ϵ_{decay} is set as 0.999. The batch size is kept at 32. The loss function used is shown in Equation 9. The optimizer used is RMSprop with a $lr = 0.00025$, $\rho = 0.95$, and $\epsilon = 0.01$. The metric used is 'accuracy' which determines the accuracy between the target Q-values and the predicted Q-values. However, another metric called 'score' is used to study the progress of the model. The score is simply the number of steps that a cartpole balances itself before falling in an episode.

$$L(\theta) = \mathbb{E}\left[\left(y_t - Q\left(s_t, a_t|\theta\right)\right)^2\right] \tag{9}$$

Then for DQN, we tried a new architecture consisting of densely connected hidden layers of 512, 256, 128, 64,

and 32 nodes with ReLU activation and 'He_uniform' initialization. The loss function was kept the same but Adam optimizer was used with a learning rate of 0.00025 and an ϵ of 0.01. This time we trained for 1000 episodes.

For Double DQN, We tested soft update (τ) values of 0.9 and 0.99. Additionally, we compared RMSProp and Adam for model optimizers, including the AMSGrad variant of Adam. For RMSProp, we used $lr = 0.00025$, $\rho = 0.95$, and $momentum = 0$. For Adam, we used $lr = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We achieved the best results by using the Adam optimizer without AMSGrad and using a hard weight update in addition to the fixed parameters already mentioned.

We tried many different architectures for CNN-DQN but did not get satisfactory results. The model that worked within the computational limits consisted of a convolutional layer of 64 filters, with a kernel size of 3 by 3 and a ReLU activation with a stride of 1. This was followed by a pooling layer with a stride of 2 and kernel size of 2 by 2. Then a convolutional layer of 128 filters, kernel size of 3 by 3, stride of 1 with ReLU activation was used again, followed by a similar pooling layer. After that, the output is flattened and fully connected layers of 512 and 64 are used with ReLU activation and the output layer is of size 2 with linear activation. The loss function used is MSE and the optimizer is the RMS prop optimizer described earlier.

For proximal policy optimization, we observed the best results with a neural network architecture consisting of two 64-sized hidden layers followed by tanh activation functions. For the actor-network (policy network) the output consists of 2 elements and for the critic network (value network) the output is a single value representing the value of the given state. The loss function used for the policy network is the negative of the clipped objective function described in Equation 5. The goal is the maximize the objective not minimize and the optimization framework in the libraries used tends to minimize the loss function. So to implement this, we add a negative sign to the clipped objective function to use it as a loss function. So when the loss function is minimized, the clipped objective is maximized. For the value network, we used the mean square error loss as shown in Equation 10.

$$L_{\text{value}} = \text{MSE}\left(V_\theta(s), R_t\right) \tag{10}$$

where $V(\theta_s)$ is the predicted value for a state and the R_t represents the actual return from the state. The optimizer used for both networks is Adam with the policy network using a learning rate of $3e-4$ and the value network using a learning rate of $1e-3$. These learning rates were set after surveying common choices of learning rates for PPO. The learning rate for the policy network is kept lower than the value network, for faster updates. The parameters that resulted in the best performance included steps per epoch number of 400, epochs/episodes of 100, γ of 0.99, λ of 0.97, and clip ratio of 0.2. During each epoch, after the buffer creation to train the policies, both policies are trained for 80 iterations. Early stopping using KL divergence is also used in the training of the policy network. If the KL divergence between the distribution of the new policy and old policy exceeds by 1.5*target$_{kl}$ then the policy training stops. The value for target$_{kl}$ is set as 0.01. The metrics used for the PPO analysis are the score, reward, policy loss, value loss, and the values obtained over time.

5. Results

For testing, we run the policy for 100 episodes. During each episode, the cartpole continues to move until termination happens. The score of this episode is recorded and at the end, the score is averaged over the 100 episodes. For the initial trials, the DQN algorithm's average test score was 113.54, for DDQN it was 102.6, and for, Dueling DDQN it was 120.45. For the CNN-DQN model, the performance was the worst as an average score of 9.31 was obtained across 100 episodes.

For the second model structure used for DQN, an average score of 137.41 was obtained. The second experiment with DDQN achieved an optimal score of 500 after 93 episodes of training. During testing, it achieved a perfect average score of 500 across 100 episodes. For PPO, the performed experiment resulted in the best performance and maintained an average score of 500 across 100 episodes. PPO control of the cartpole is faster as compared tpo the DDQN control of cartpole. The termination for DQN and Dueling DDQN happens due to cart position going out of bounds and for CNN-DQN it happens due to cartpole angle going out of bounds. The visual results for all the models can be viewed **here.** Figures 2, 3, 4, 5, 6 show the loss, score, value, reward and accuracy graphs respectively, for the best experiment performed for each model, except Dueling DDQN, for which the results for the initial experiment as shown.

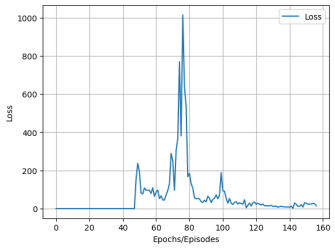
For the second architecture used for DQN, we see that the memory is created around 50 episodes and the loss value takes a rapid increase till around 75 episodes and then begins to decrease with a nice pattern. The training stops at around 160 episodes, as the desired score of 500 is achieved. For double DQN, we observe a similar pattern with the loss value rising after memory creation and then falling down rapidly till the training stops at around 90 episodes. A similar pattern is observed for Dueling DDQN as well. The results for CNN-DQN indicate that the

optimization process is not converging at all, as we only see a spike in the loss value which is around $1e^{-9}$, and then the loss reaches back to 0 without any improvement or pattern. The policy loss for PPO does well as it starts from a high negative value and reaches a low negative value over 100 episodes. The value loss however has an increasing and decreasing trend. Apart from the CNN-DQN method, we observe from the loss functions that the policies are trained toward improvement. But, the learning rate might need to be adjusted for a smoother training.

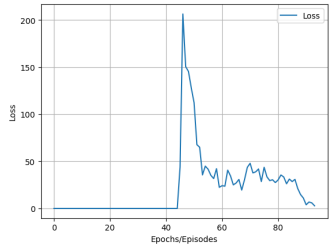
An accuracy of around 80 $\%$ is reached for the Q-value estimation by DQN, Double DQN, and Dueling DDQN. The accuracy levels for CNN-DQN fluctuate a lot between 50 and 100 $\%$, but the final results are not conclusive. We did not use the accuracy metric for PPO. Generally, RL-based methods are not studied through accuracy and we observed this with our initial DQN methods experiments as well. We see that DQN, DDQN, Dueling DDQN, PPO have an increasing score pattern which eventually reaches 500. PPO however, reaches to the 500 score the fastest. This can be owed to the fact the policy changes are not drastic and therefore, during the deployment time, the cart remains stable because the policy does not apply rapidly changing actions, which leads to an increase in the score. The CNN-DQN score values fluctuate a lot again. The value graphs show a fluctuating pattern. For DQN, the value graph eventually stabilizes around a value of 25. For DQN, it reaches a maximum value of 10 and then shuffles between -20 and 10. The value graph for dueling DDQN observes a continuous increase. The value graph for PPO reaches a maximum value of 600 and touches this threshold quite often during training. The value graph of CNN DQN has one major peak and no progress after that, which is similar to its loss graph.

6. Conclusion and Future Works

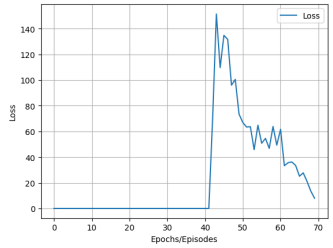
The project was successful in developing a coding pipeline for understanding the fundamental RL algorithms; DQN with its variations and PPO on a significant and classical control problem. The PPO methods and DDQN variation prove to be the best. CNN-DQN methods, however, need further experimentation and exploration, as a lot of times obtaining state through sensors is difficult, and vision-based state estimation is very useful. Possible reasons for CNN-DQN failure included; lack of computation, lack of image processing to get the desired feature inputs, and lack of experimentation. These issues need to be explored in the future. Also, termination of training for DQN methods when the desired score is reached is not good enough, as for PPO the training continues to happen for all episodes, even if the score reaches 500, which potentially is a reason why PPO did so much better than the rest.



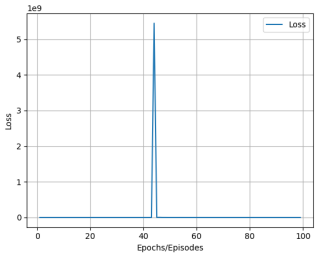
(a) DQN loss



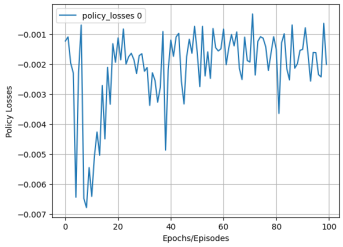
(b) Double DQN loss



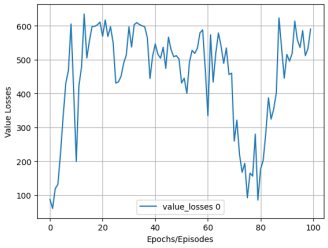
(c) Dueling Double DQN loss



(d) CNN DQN loss

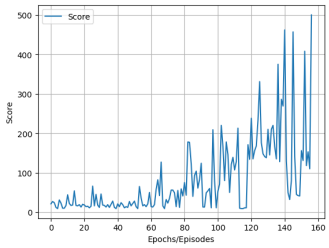


(e) PPO policy loss

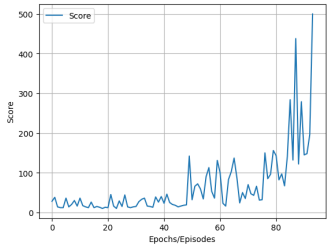


(f) PPO value loss

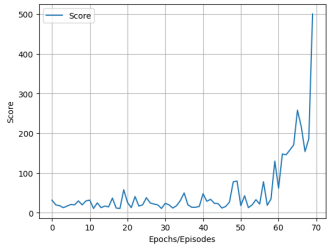
Figure 2. Loss Results



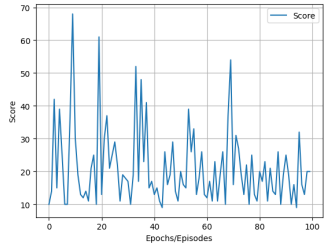
(a) DQN score



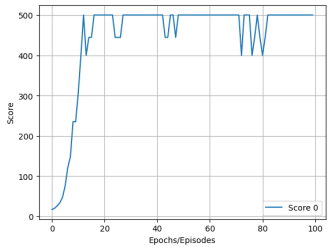
(b) Double DQN score



(c) Dueling Double DQN score

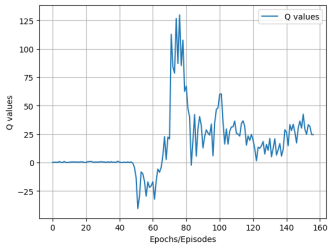


(d) CNN DQN score

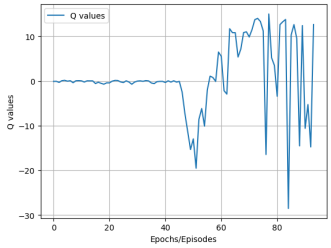


(e) PPO score

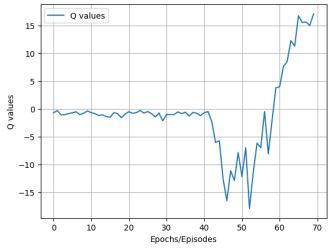
Figure 3. Score Results



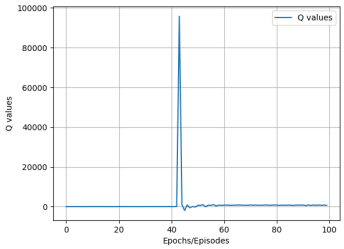
(a) DQN Q-value



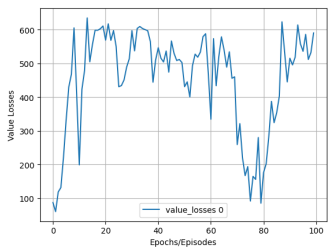
(b) Double DQN Q-value



(c) Dueling Double DQN Q-value

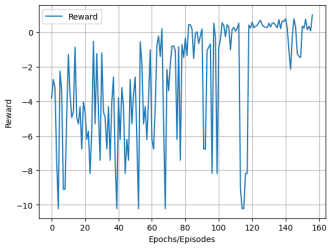


(d) CNN DQN Q-value

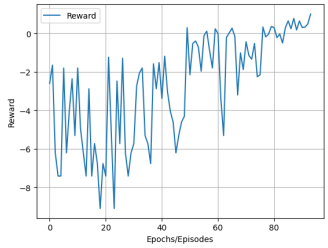


(e) PPO value

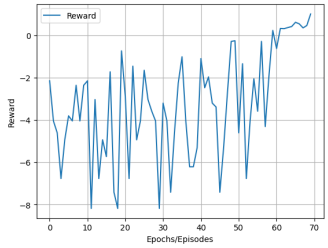
Figure 4. Q Value and Value Results



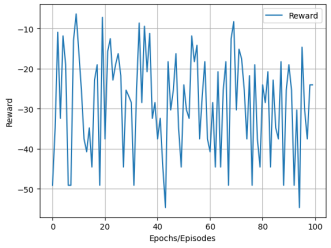
(a) DQN rewards



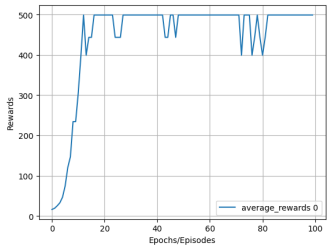
(b) Double DQN rewards



(c) Dueling DDQN rewards

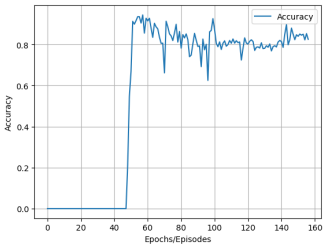


(d) CNN DQN rewards

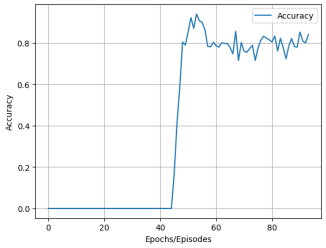


(e) PPO rewards

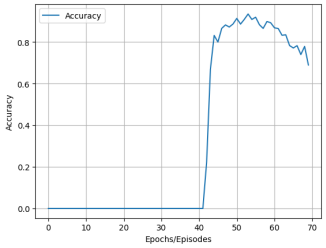
Figure 5. Rewards



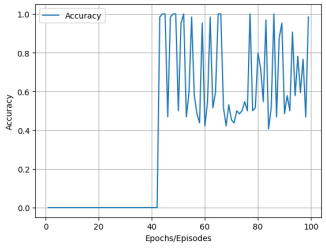
(a) DQN accuracy



(b) Double DQN accuracy



(c) Dueling Double DQN accuracy



(d) CNN-DQN accuracy

Figure 6. Accuracy Results

References

[1] Swagat Kumar. Balancing a cartpole system with reinforcement learning–a tutorial. *arXiv preprint arXiv:2006.04938*, 2020. 1

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 1

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 1, 3

[4] Silver Van Hasselt, Guez. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, 30(1), 2016. 2

[5] Hessel Hasselt Lanctot Freitas Wang, Schaul. Dueling network architectures for deep reinforcement learning, 2016. 2

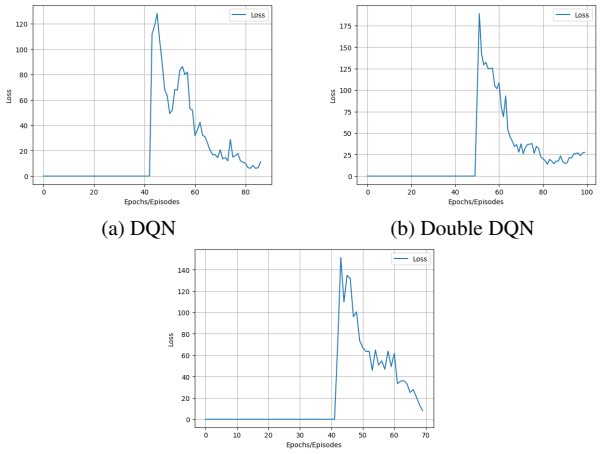
[6] Anton Zakharenkov and Ilya Makarov. Deep reinforcement learning with dqn vs. ppo in vizdoom. In *2021 IEEE 21st International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000131–000136. IEEE, 2021. 1

Table of contributions

Task	Name
DQN Implementation	Waleed Bin Khalid Seungmi Lee
DQN hyperparameter tuning	Seungmi Lee
DDQN Implementation	Logan Bowers
DDQN hyperparameter tuning	Logan Bowers
Dueling DDQN Implementation	Logan Bowers
PPO implementation	Waleed Bin Khalid
PPO hyperparameter tuning	Waleed Bin Khalid
CNN Implementation	Waleed Bin Khalid Keyang Lu
CNN hyperparameter tuning	Keyang Lu
Metrics Coding and Plotting	Waleed Bin Khalid
Report Writing/Presentation/Videos	All

Appendix

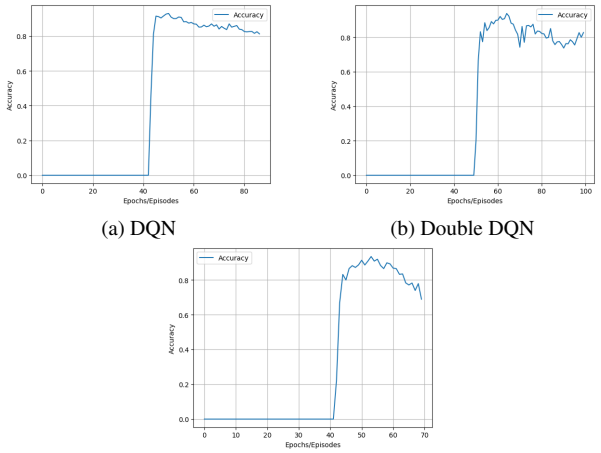
The following results were obtained for the initial hyperparameter settings of DQN, DDQN, and Dueling double DQN.



(a) DQN (b) Double DQN

(c) Dueling Double DQN

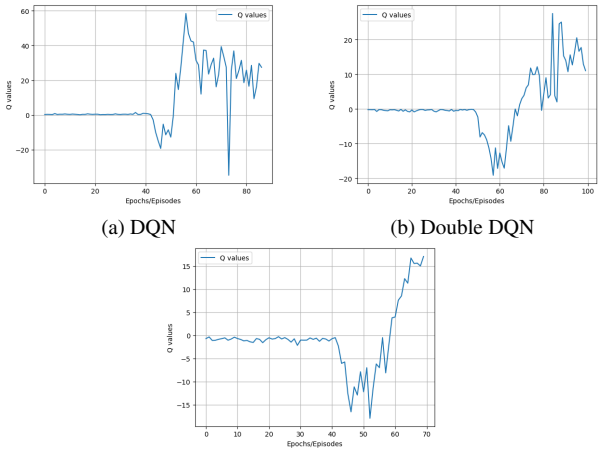
Figure 7. Loss Results



(a) DQN (b) Double DQN

(c) Dueling Double DQN

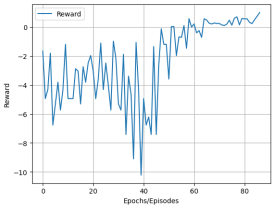
Figure 8. Accuracy Results



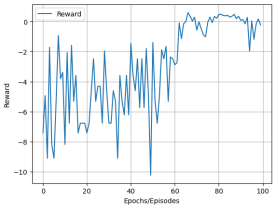
(a) DQN (b) Double DQN

(c) Dueling Double DQN

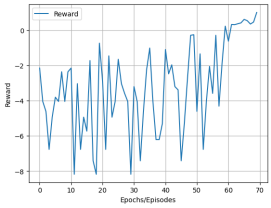
Figure 11. Q values Results



(a) DQN

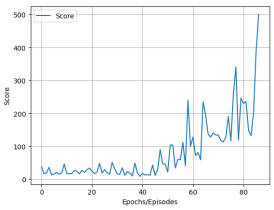


(b) Double DQN

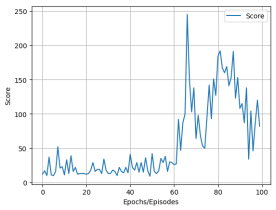


(c) Dueling Double DQN

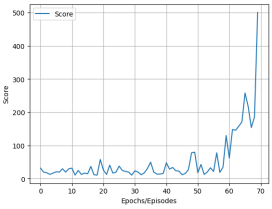
Figure 9. Reward Results



(a) DQN



(b) Double DQN



(c) Dueling Double DQN

Figure 10. Score Results