REINFORCEMENT LEARNING PROJECT

Ms. Pac-Man with DQN and DDQN

EMAI

2023-2024

Authors:

Nikolay Kormushev

Cristian Bassotto

Course Coordinator:

Gergely Neu

22^{th} January 2024

Contents

1 Abstract

Reinforcement learning (RL) has witnessed significant advancements, especially with the introduction of Deep Q-Networks (DQN), a class of algorithms that combines deep neural networks with Q-learning. In this project, we explore the application of Double Deep Q-Networks (Double DQN) in the challenging Ms.Pac-Man environment, a classic arcade game known for its complexity and dynamic nature.

Our implementation leverages a convolutional neural network (CNN) architecture to process game frames and approximate the Q-values for different actions. The training process incorporates experience replay and a target network, enhancing the stability and efficiency of learning. Additionally, we integrate the Double Q-learning approach to mitigate overestimation bias and potentially improve the overall performance by decoupling action selection and evaluation.

The experimental setup involves testing DQN's, Double DQN's and a third approach called Clipped double Q-learning, to optimize performance in the Ms.Pac-Man domain. We analyze the impact of these choices on the learning process and the agent's ability to navigate the complex game environment.

The outcomes of our experiments are assessed through metrics such as average episode rewards and scores. We discuss challenges encountered during training and potential avenues for further improvement.

The aim of this project is to test these algorithms, tune them to see what results we can get and as a result gain a deeper understanding.

2 Introduction

Deep reinforcement learning has demonstrated remarkable success in tackling complex decision-making tasks across various domains. One prominent algorithm in this field is the Deep Q-Network (DQN), which utilizes deep neural networks to approximate the Q-values, representing the expected cumulative rewards for different actions in a given state. While DQN has shown great promise, it is not without its challenges, particularly in scenarios involving overestimation of action values.

In this project, we focus on addressing these challenges by incorporating the Double Q-learning approach into the DQN framework. This extension aims to mitigate the overestimation bias by decoupling the action selection and evaluation processes. To assess the effectiveness of our approach, we apply it to the Ms. Pac-Man environment, a classic and complex arcade game.

The Ms. Pac-Man environment presents a dynamic and visually rich setting, making it an ideal testbed for evaluating the robustness and adaptability of our Deep Double Q-Networks (Double DQN). Our implementation involves the use of a Convolutional neural networks(CNNs) to process game frames and learn optimal action policies.

3 Preprocessing and CNN architecture

3.1 Preprocessing

In the realm of Atari-playing Deep Q Networks (DQNs), where agents are tasked with making sense of the screen to excel at games like Ms. Pac-Man, preprocessing plays an important role in shaping the input data before it reaches the neural network. Unlike humans, who can effortlessly interpret the game screen, the DQN relies on a convolutional neural network (CNN) to derive meaningful insights from the pixel values.

The use of a CNN involves employing a series of convolutional layers that aim to discern relevant game information from the chaotic array of pixel values. However, due to computational constraints, the raw game screen is not directly fed into the network. Instead, a preprocessing pipeline is implemented:

- **Noop reset:** This step introduces variability in the starting states by performing a random number of no-op (no operation) actions. This helps prevent overfitting and ensures that the agent encounters different initial conditions during training.

- **Frame skipping (4):** Given that many consecutive frames may not provide useful information, the pipeline skips every 4th frame. This reduces the computational load and focuses the learning process on frames that are more likely to convey relevant game dynamics.

- **Max-pooling (most recent two observations):** From the frames skipped in the previous step, the pipeline selects the one with the highest pixel values. This max-pooling approach is designed to mitigate the sprite-flicker effect, where ghosts fade in and out. By prioritizing frames with higher pixel values, the model aims to enhance stability in observing the game environment.

- **Termination signal when a life is lost:** This step introduces a termination signal when a life is lost in the game. This termination signal aids the model in understanding and responding to changes in the game state associated with the loss of a life.

- **Resize to a square image (84 x 84):** The game screen is resized to a square image with dimensions 84 x 84 pixels. While this operation may result in the removal of some pellets, the model relies on the chance that Ms. Pac-Man will inadvertently consume them as she

moves through their positions. This resizing contributes to more manageable input sizes for the neural network.

- **Grayscale observation:** The colors are removed from the image, resulting in a grayscale observation. While this eliminates color-related information, such as distinguishing between different colored ghosts, it aids in reducing input dimensionality, making the learning process more efficient.

- **Clip reward to -1, 0, 1:** The rewards obtained during gameplay are clipped to the range -1, 0, 1. This normalization ensures that the rewards do not become excessively large, facilitating more stable learning during the training process.

- **Frame Stacking (4 frames):** The pipeline stacks the most recent 4 frames. This stacking enables the model to discern the direction in which both Ms. Pac-Man and the ghosts are moving, providing valuable temporal information for decision-making.

This comprehensive preprocessing pipeline is designed to strike a balance between reducing input dimensionality, enhancing computational efficiency, and providing the neural network with meaningful information to learn the complexities of Ms. Pac-Man gameplay.

3.2 CNN architecture

In the context of our Ms. Pac-Man Deep Q Network (DQN), the convolutional neural network (CNN) architecture serves as a component for estimating Q values, which represent the expected future rewards associated with each possible action. The CNN processes the preprocessed input to generate them, and the entire network is trained.

The architecture, depicted in Figure 1, consists of several layers designed to extract essential features from the game frame. The architecture is outlined as follows:

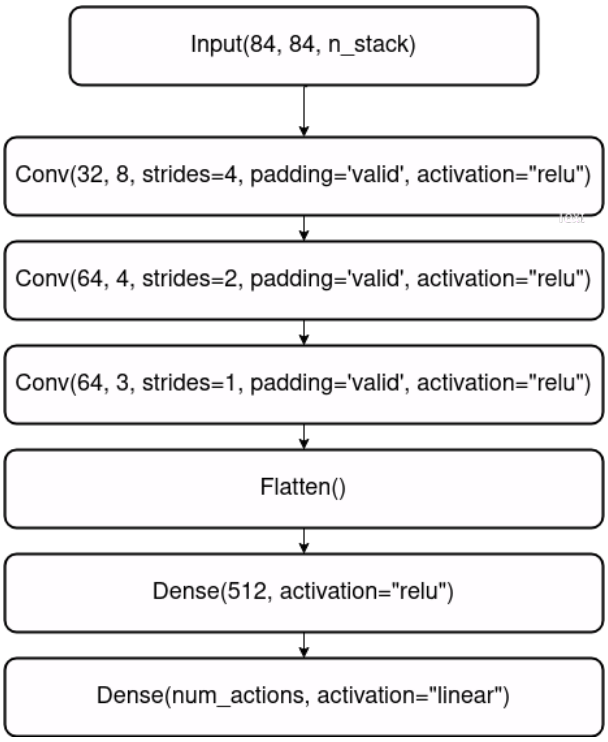

Figure 1: *CNN architecture*

- **Convolutional Layers:** The convolutional layers progressively employ different filters and sizes to extract and refine features from the input data.
- **Flatten Layer:** After the convolutional layers, the output is flattened, converting the multi-dimensional tensor into a flat vector.
- **Dense Layers:** Now the flow goes through two fully connected layers: `Dense1` captures complex, abstract features, while the final layer, `dense2`, contains output neurons corresponding to the possible actions in the Ms. Pac-Man game.

4 Deep Q Network (DQN) algorithms

4.1 DQN

Deep Q-Network (DQN) is a reinforcement learning algorithm that combines deep learning with Q-learning to address the challenges of learning optimal policies in high-dimensional state spaces, like explained in the paper of Mnih et al., 2013. In this section we will describe the algorithm and explain how it works

---

DQN

---

1. for $k = 1, 2, ...K$
2. $\pi_k = \sum -greedy(Q_{\theta_k})$
3. Generate data $\tau_k = (X_{1,k}, A_{1,k}, R_{1,k}, ..., X_{n,k}, A_{n,k}, R_{n,k})$
4. $B_k = (\widetilde{X_1}, \widetilde{A_1}, \widetilde{R_1}, ..., \widetilde{X_b}, \widetilde{A_b}, \widetilde{R_b}) \in \tau_1 \cup \tau_2 \cup ... \cup \tau_k$ - sample minibatch from experience replay buffer
5. $y_{target} = \widetilde{R_i} + y * max_{a'}(Q_{\theta'_k}(\widetilde{X'_i}, a'))$
6. $g_k = \nabla_{\theta_k} \frac{1}{b} \sum_{i=1}^{b} (y_{target} - Q_{\theta_k}(\widetilde{X_i}, \widetilde{A_i}))^2$
7. $\theta_{k+1} = \theta_k + \alpha_k + g_k$
8. Every M steps $\theta'_k = \theta_k$

The algorithm itself uses a for loop on two neural networks that estimate Q values. These networks are called an online network Q_{θ_k} and an offline (or target) network $Q_{\theta'_k}$. At the start both networks are initialized with random weights θ_k and θ'_k. We fix the weights of the offline network and we use it as a target for training as seen on lines 5, 6 and 7 which show how we do a single iteration of SGD. We do this to keep stability during training and to reduce variance.

The y_{target} is a bit more accurate then our estimations because it is a combination of the immediate reward (which we know) and the best reward we can get in our future state (which is calculated by maximizing the offline Q value estimate in that next state $\widetilde{X'_i}$). Using this we train the online network to estimate the Q values in our current state $\widetilde{X_i}$ when taking the action $\widetilde{A_i}$.

After multiple training iterations, the online neural network becomes proficient at estimating target values, representing expected future rewards. We then update the offline network with the learned weights from the online network. This synchronization enhances the accuracy of future reward estimates. The training process restarts with the updated offline network, continuing the

iterative cycle of learning and refinement. This alternating update strategy ensures the neural network converges to more accurate representations over time. This update is done every M iterations where M is around 10,000 - 100,000 so that we continue learning. (See line 8)

The policy we use to generate data is epsilon greedy. This means with some probability based on an epsilon hyperparameter we choose a random action or the best action we can take based on the Q value functions. The epsilon starts big and is reduced from around 1 which means we only do random actions to around 0.1 which means we do random actions only 10% of the time. This period is called the discovery period because we want to discover new actions so we do a lot of random ones. After it ends we keep a 10% chance of doing a random action so we can still continue to discover when we end up in new positions. The generation of data using the policy is seen on lines 2 and 3

The algorithm uses also an experience replay buffer which is used to sample for batches of data as seen on line 4. In the buffer the previous observations or so called experiences are added so we can use them to learn.

On line 6 MSE is used for the loss function but in our implementation we use the Huber loss. Our reasoning for this is that we saw that stable baselines3 uses the smooth l1 loss and the closest thing to that that Keras offers is the Huber loss. We read that the main benefit of this is that high variance is not penalised as much in this case compared to with MSE.

We also want to mention that we did a vectorized implementation that can run multiple games in parallel and learn from all of them. This uses the Vec methods from stable_baselines3 for preprocessing.

4.2 Double DQN

Double DQN is similar to DQN but with a different target value for calculating the gradient:

$$y_{target} = R_i + y * Q_{\theta'_k}(X'_i, argmax_{a'}(Q_{\theta_k}(X'_i, a')) \tag{1}$$

This approach is implemented with the objective of minimizing overestimation. In the y_{target} equation in line 5 of DQN, the offline $Q_{\theta'}$ network is employed for both selecting the action and estimating its value. This practice introduces a risk of overestimation, as noted in Van Hasselt et al., 2016, potentially leading to a preference for suboptimal actions associated with inflated Q values.

To address this, we separate the network that makes the choice of action from the network that gives us its Q value. In equation (1) we use the online network to choose the action and the offline one to choose its value. This means that we will not always overestimate actions because the action chosen by Q_{online} might have a lower Q value given by $Q_{offline}$.

When the overestimation is reduced, large performance improvements can be seen for many games in Van Hasselt et al., 2016. 'Ms Pacman' does not benefit that much but it is a easy to implement improvement over DQN we decided to add and look how it compares.

4.3 Clipped Double Q-learning

This section is composed of two parts. The first should be a quick exploitation through the first paper of Clipped Double Q-learning paper Fujimoto et al., 2018and the newer one Jiang et al., 2022. Instead, the second part, should cover our assumptions of a deep reinforcement learning with Clipped DDQN.

4.3.1 Clipped Double Q-learning review

Clipped Double Q-learning is designed within the framework of actor-critic methods. In this context, we introduce an additional neural network known as π_ϕ, which serves as the policy network. Alongside π_ϕ, we utilize two distinct Q-value estimation networks, denoted as Q_{θ_1} and Q_{θ_2}. Each of these neural networks is equipped with its own dedicated fixed target or offline network. They chose the target for the Q networks as such:

$$y_{target} = r + \gamma min_{i=1,2} Q_{\theta'_i}(s', \widetilde{a}) \tag{2}$$

The term 'clipped' in Clipped Double Q-learning arises from the interplay between these two policy networks, where they essentially act as upper bounds for each other. Specifically, only using this target configuration is considered as employing the Clipped Double Q-learning approach.

Furthermore, the algorithm described in the paper incorporates additional features to enhance its performance. It adopts delayed policy and target updates, contributing to the stability of the training process. Additionally, the algorithm incorporates target policy smoothing regularization, which ensures that similar actions result in similar Q values. The comprehensive integration of these techniques leads to the final algorithm described in the paper, known as

TD3 (Twin Delayed DDPG).

The paper suggests updating the policy at less frequent intervals, occurring every M steps, along with making adjustments to target networks. The way they tweak the policy involves using a deterministic policy gradient, described here:

$$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s,a)\Big|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s) \tag{3}$$

The reason for infrequent policy updates is to focus on minimizing errors in Q-value functions before making simultaneous changes to both the policy and Q-values. This is done to avoid introducing too much variance, which could lead to divergent behavior if the policy is updated too frequently. This concern is particularly pertinent to DDQN, where each Q-value function update leads to changes in the policy and increased variance. In DDQN, the use of fixed target networks also contributes to stability by reducing noise and variance.

The preference for TD3 over DDQN is based on the expected benefits of reduced variance and overestimation. However, due to time constraints, a thorough evaluation comparing these advantages in our specific context hasn't been carried out yet, leaving room for further exploration.

4.3.2 Our assumptions and implementations

Now, our **first assumption** is made from the article Yoon, 2019. It is stated that Clipped Double Q-learning uses the following target:

$$y_{target} = r + \gamma * min_{i=1,2}(max_{a'} Q_{\theta_i}(s', a')). \tag{4}$$

which results in us considering the minimum of the maximum Q-values from both networks $Q_{offline}$ and Q_{online} in a given state s'. In simple terms, it ensures that we're not overly optimistic by picking the smaller of the maximum Q-values from either network. This cautious approach in estimating the target value aims to make our learning process more reliable and robust. However, as described in the article, our training approach involved both networks. Unfortunately, this led us to an unbalanced and improperly clipped version that failed to learn anything meaningful from the game.

Realizing our initial approach wasn't working, we decided to switch things up, leading to our **second assumption**. We came across the two papers mentioned earlier, and by tweaking

the equation, we managed to achieve better results. Here's the formula we found to be more effective:

$$min(R_i + y * Q_{\theta'_k}(X'_i, argmax_{a'}(Q_{\theta_k}(X'_i, a')), R_i + y * max_{a'}(Q_{\theta'_k}(X'_i, a')) \quad (5)$$

We didn't see a huge leap forward, but there was some improvement in learning. The initial puzzle we didn't catch was that this equation turned out to be essentially the same as Double DQN, as the minimum value consistently originates from the first part of the equation.

In our **most recent version**, we took the equation (4), but with a different approach. We decided to train solely on the $Q_{offline}$, treating it like a regular DQN model. What's more, we introduced a soft update with a different set of parameters. This adjustment led to some very positive outcomes.

5 Experimental setup and results

5.1 Training Process

5.1.1 Initial Simple Implementation

Our initial implementation [1] leverages a straightforward Python script to apply the Deep Q-Network (DQN) algorithm for training an agent in the Ms. Pac-Man environment. The key components of this implementation include a Convolutional Neural Network (CNN) architecture created using the Sequential model from TensorFlow, a basic experience replay class to manage the buffer of past experiences, and various functions for creating, training, updating, and saving the network.

After conducting further research, we opted to implement preprocessing techniques as explained in Chapter 3. We utilized AtariWrapper from the Stable Baseline 3 wrappers and wrapped the Atari environment in a VecFrameStack to stack a variable number of frames.

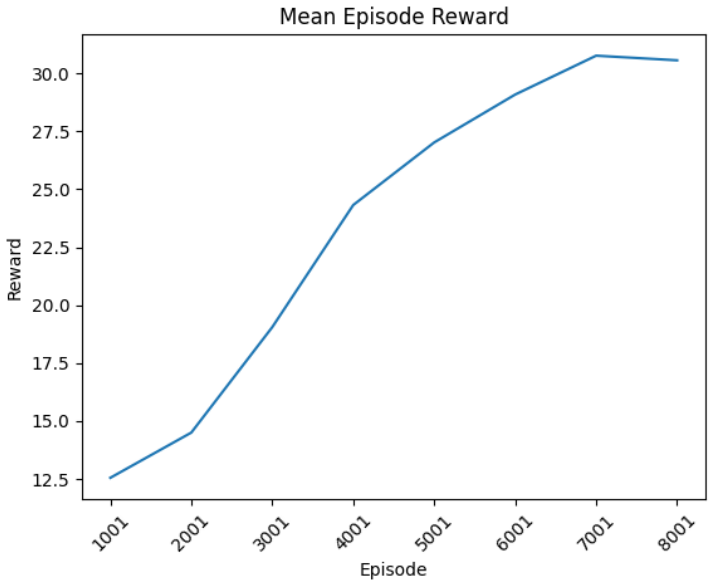

Figure 2: Mean episode clipped reward for the initial DQN Model

In the initial implementation, we chose to train by episodes and evaluated the model based on the mean reward of each episode, determined by the clip reward at the end of each step. Here, one episode corresponds to the loss of one life, not the entire game. A game reset occurs after three lost lives or when the time limit is reached. For subsequent implementations, we shifted to training by the number of steps, considering three sub-episodes as one game/full episode.

[1] https://github.com/nickormushev/PacmanDQN/tree/main/initial_DQN

Despite this, the initial DQN showcased a noteworthy increase in reward, as illustrated in the graph in Figure 2.

5.1.2 DQNmodel Class

Our final implementation [2] , encapsulated within the `DQNmodel` class, is a versatile tool for training and evaluating Deep Q-Network (DQN) models. This class offers extensive flexibility through numerous parameters, enabling users to finely tune the training process. Going beyond the capabilities of the DQN class in Stable Baselines 3, our implementation introduces features such as creating checkpoints for both models and optimizers, as well as logging mechanisms for future evaluations of the training process.

One notable aspect of our implementation is its support for various DQN variants, including Vanilla, Double, and Clipped Double DQN. Users can easily select these variants by specifying the desired `dqn_type` parameter during class instantiation. This design promotes extensibility, allowing for the straightforward implementation of additional models within the same class. Moreover, our class facilitates training in a multi-environment setup while utilizing the same underlying environment, providing a seamless exploration of different DQN configurations.

A distinctive feature of our implementation is its ability to record and save gameplays, enhancing the capacity for analysis and review of the agent's behavior during training and evaluation. This feature enables users to gain deeper insights into the learning process and make informed decisions about model improvements.

Here's an overview of its main functionalities:

- **Training:** The `train` method is used to train the DQN model. It takes the total number of training steps as an argument and handles the training process, including warm-up, experience replay, and model updates.
- **Prediction:** The `predict` method is utilized for making predictions with the trained model. It takes an observation as input and returns the predicted actions.
- **Evaluation:** The `evaluate` method evaluates the model over a specified number of episodes. It returns a `LogData` object containing relevant statistics, including scores, rewards, exploration rates, lengths, and times.

[2] https://github.com/nickormushev/PacmanDQN/tree/main/DQN

- **Environment Interaction:** The class provides methods like `env_step`, `restart_env`, `render_env`, and `close` for interacting with the environment, restarting it, rendering frames, and closing the environment.
- **Saving and Loading Models:** The `save_model` method saves the trained model, and the `load_model` method loads a pre-trained model.
- **Logging and Statistics:** The class uses a `LogData` object to store and print training or evaluation statistics. The `print_statistics` method provides a summary of relevant metrics.

In our design choice, we opt to allocate an entire class specifically for handling logging data. The `LogData` class is meticulously crafted to serve as a repository for storing and managing training or evaluation data. It provides dedicated methods for adding episode data, printing statistical summaries, and persisting the data to disk. Notably, the class is optimized to efficiently manage disk space by automatically freeing up storage if the CPU usage exceeds 90%. This optimization ensures a balance between effective data retention and resource utilization during the training or evaluation process.

5.1.3 Training results

As an initial evaluation we decide to analyze the clipped reward as a metric during the training process.

In Figure 3, we illustrate the progress of the Vanilla DQN, commencing from an initial reward of 30 points. Our experimentation involves training five distinct versions, wherein we manipulate the target function update interval (at every 50 and 10,000 steps). Additionally, we fine-tune the exploration rate function, varying the final rate (0.05 for 'final' and 'normal' versions, and 0.2 for others) and exploration fraction (0.1 for 'final' and 'normal' versions, and 0.4 for others). The graph prominently showcases the superior performance of the final version, which updates the target network every 10,000 steps with a lower exploration rate. Notably, excessively frequent network updates can yield suboptimal results, as the online model attempts to adapt to a target model that is evolving and altering its values. It is crucial to strike a balance in the update frequency. All models undergo training for 1 million steps, resulting in varying total numbers of games played, depending on the duration of each game.

Here we also wanted to mention we tried to run the models with different batch sizes. For us a batch size of 32 gave the best results. When using a batch size of 64 the models did not learn anything and in the case of 16 the model was learning very slowly.

For learning rate the best results we got were with 0.00025. For other values the results were either similar or worse.

The results for the batch size and the learning rate tests were done with the vanilla implementation and were not saved to put into graphs. We think this might be a lesson for us to save everything we tested so we can better support our statements in the future.

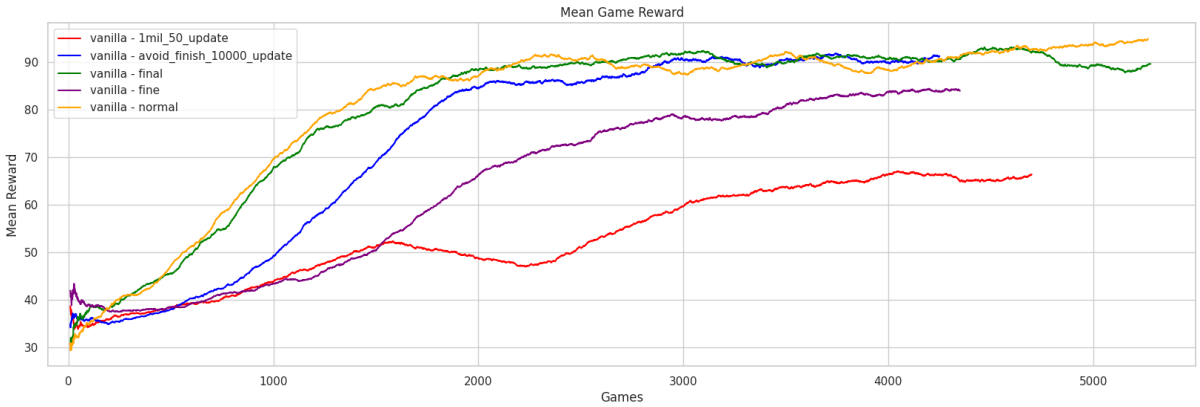

Figure 3: Mean game clipped reward for the Vanilla DQN models

Now, let's have a look into the insights provided by Figure 4, focusing on the Double DQN. Our experimentation involved testing various hyperparameters. τ is the parameter governing soft or hard updating in the formula:

$$Q_{\text{target}}(s,a) = (1-\tau)\cdot Q_{\text{target}}(s,a) + \tau \cdot Q_{\text{online}}(s,a) \tag{6}$$

τ plays a crucial role in determining the extent of target network updates. The graph reveals a comparatively modest improvement, yet intriguingly, higher τ values exhibit superior results. Notably, this analysis yields a peculiar observation – unlike Vanilla DQN, the optimal final model for Double DQN involves more frequent updates. Additionally, the model employing hard updates every 10,000 steps performs well for the first 1,000 games, only to exhibit a nearly linear decline in performance thereafter. Conversely, the model with soft updates performs unexpectedly poorly, even falling below the performance of randomly choosing actions.

Now, in reference to the Clipped Double DQN we constructed, elucidated in Chapter 4.3, we implemented two distinct versions of the algorithm. The first closely resembles the DDQN,

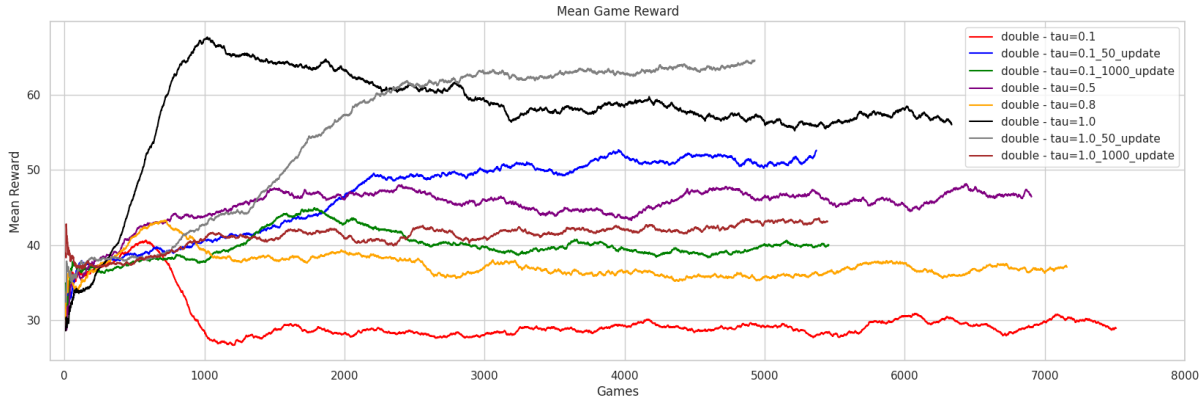

Figure 4: Mean game clipped reward for the Double DQN models

updating both the online and target models, whereas the second exclusively updates the target model, considering the minimum of the Q-values from the online and target networks. The disparity between these implementations is evident in Figure 5. Notably, the initial clipped models exhibit a marginal improvement compared to their more sophisticated counterparts. We conducted training using various updating configurations, manipulating the τ parameter from 1.0 to 0.25, thereby transitioning from a hard to a soft update for the target network. However, in this instance, the update strategy appears to have a limited impact on the model's outcomes, as the results remain relatively consistent. There is a slight improvement observed with higher τ values, but overall, the models perform quite similarly.

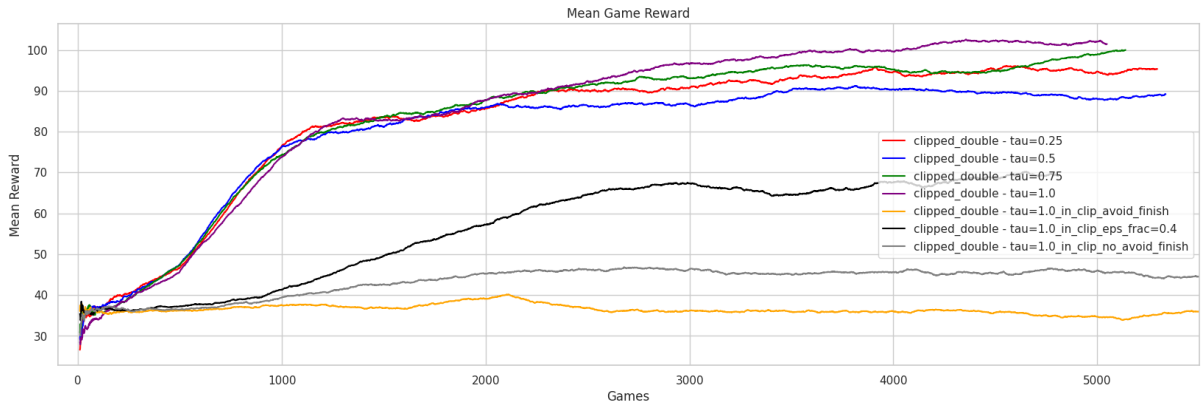

Figure 5: Mean game clipped reward for the Clipped DDQN models

As a conclusion, we can compare the top three models shown in Figure 6. DDQN performs the least effectively, whereas Vanilla and Clipped DDQN appear similar at the beginning. However, as the training progresses, Clipped DDQN surpasses the basic DQN implementation in terms of performance.

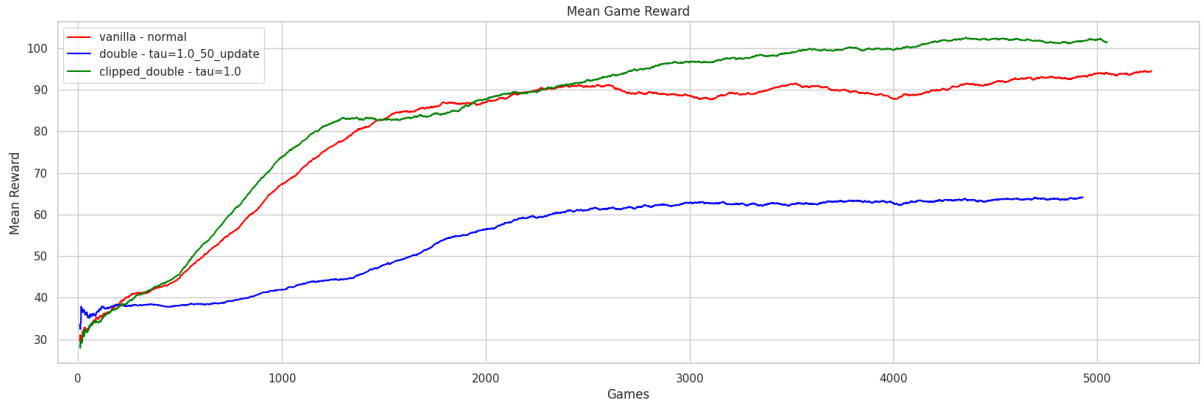

Figure 6: Mean game clipped reward for the best models for each type

5.2 Evaluation process

To ensure a fair and comprehensive evaluation of all models, we employ a standardized evaluation function. This function remains consistent across various DQN types and hyperparameters, with the exception of the final exploration rate, which defaults to 0.05 for the evaluation phase. Subsequently, we assess the models across 1000 episodes, using the mean clipped reward and score per episode as the chosen metrics. While the clipped reward provides insights into the immediate positive or negative feedback, the score offers a broader perspective by considering the overall success or effectiveness of the model across multiple episodes.

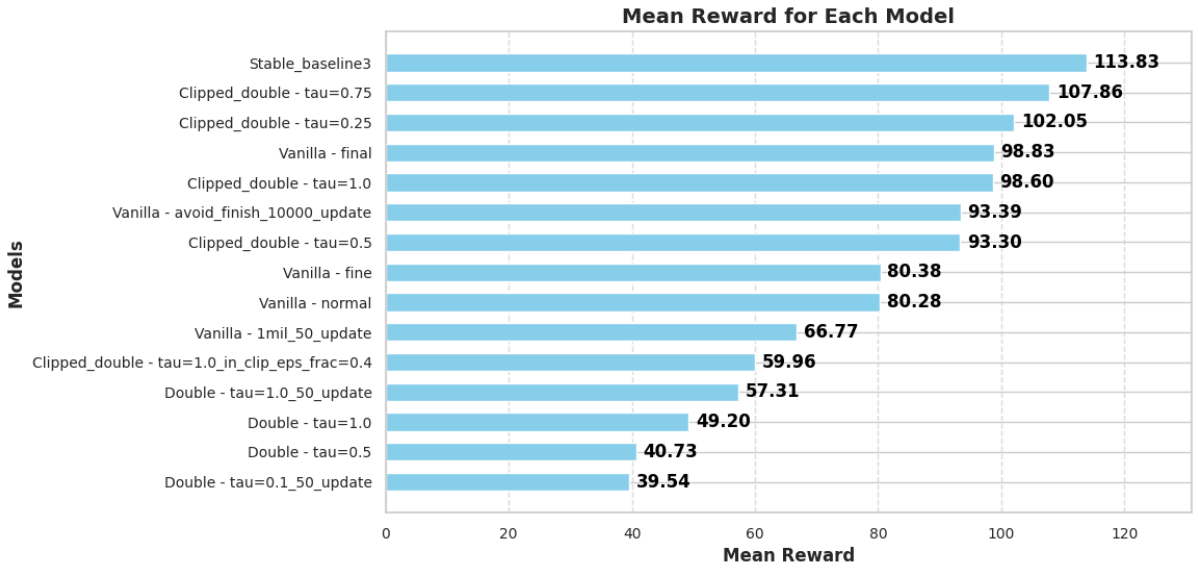

Figure 7: Mean Game Reward for each model during evaluation

The clipped reward focuses on limiting extreme values, providing a more stable measure for loss. However, relying solely on this metric might not fully capture the true progress of the model. Instead, considering the score of a game as our primary metric is a better representative

of the actual improvement. The game score takes into account the cumulative impact of the model's decisions over the entire gameplay, providing a more comprehensive and meaningful measure of its overall performance and progress.

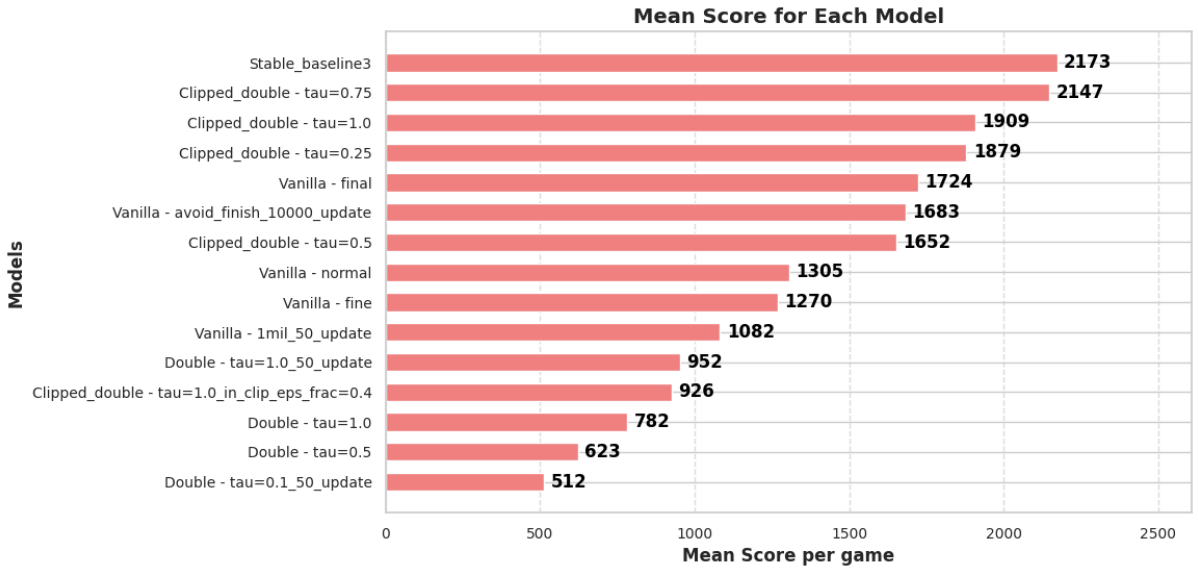

Figure 8: Mean Game Score for each model during evaluation

Figure 7 and 8 represent the final result of the evaluation process. After predicting 1000 episodes, we calculated the mean of the reward and score. We decide to add also the well known implementation of DQN from stable_baseline3 trained with the same hyperparameter as our best vanilla DQN. Even if clipped double with $\tau = 0.75$ wasn't the favourite from the training process, in this final evaluation outperform the best among our implementations, almost reaching the score of the stable_baseline3 class.

Optimal results are likely to be achieved with extended training, ideally surpassing 10 million steps and reaching approximately 40 million steps. Regrettably, we were constrained by limitations in computational power and time, preventing us from conducting such extensive training.

6 Possible Future Developments and Implementations

In this chapter, we discuss potential future developments based on our research and insights from relevant literature. While our main focus has been on implementing Double DQN and Clipped Double DQN models in this project, we came across promising concepts like Prioritized Experience Replay, Noisy Networks, and Dueling Architectures. Due to time constraints, we couldn't implement these interesting strategies, but we highlight them as promising avenues for future improvements.

- Prioritized Experience Replay (PER) (Schaul et al., 2016) - Instead of uniformly sampling experiences, PER assigns priorities to each experience based on its impact on learning. Experiences with higher priorities, such as situations where our Pacman agent was defeated, are more likely to be sampled during training. This prioritization aims to improve the learning process by focusing on more informative experiences.

- Dueling architectures (Wang et al., 2016) - splitting the neural network into two streams to separately estimate the value function (V) and the advantage function (A). The Q-value is then calculated by combining these two estimates. This architecture allows the model to better understand the value and advantage of different actions, potentially improving learning efficiency. By clearly distinguishing between the estimation of value and advantage, the model can capture more detailed information about the state-action pairs, potentially resulting in better-informed decisions.

- Noisy networks (Fortunato et al., 2019)- offer an alternative to traditional epsilon-greedy exploration. Rather than using a fixed exploration probability (epsilon), Noisy Networks introduce stochasticity directly into the parameters of the neural network. The network learns to control the amount of noise applied during exploration. This dynamic approach allows the model to adapt its exploration strategy based on the current environment and learning progress. It is expected that Noisy Networks could offer a more adaptive and efficient exploration method compared to static epsilon-greedy strategies.

7 Conclusion

In this project, we studied advanced reinforcement learning techniques, with a focus on the challenging Ms. Pac-Man game. We used a variant called Double Deep Q-Networks (Double DQN) to address biases in the original Deep Q-Networks (DQN). We applied a convolutional neural network (CNN) and various preprocessing techniques like frame skipping, max-pooling, and frame stacking to extract important information for decision-making.

We experimented with three key algorithms: DQN, Double DQN, and a new approach called Clipped Double Q-learning. While DQN was our baseline, Double DQN aimed to reduce bias, and Clipped Double Q-learning introduced complexity but led to better results. We evaluated the algorithms using metrics like average episode rewards and scores.

Throughout the project, we faced challenges, especially in tuning hyperparameters and selecting methodologies. The results highlighted the performance differences of each algorithm in the Ms. Pac-Man game. Surprisingly, our exploration of Clipped Double Q-learning revealed unexpected improvements, making us reconsider its potential benefits.

References

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., & Legg, S. (2019). Noisy networks for exploration.

Fujimoto, S., Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. *International conference on machine learning*, 1587–1596.

Jiang, H., Xie, J., & Yang, J. (2022). Action candidate driven clipped double q-learning for discrete and continuous action tasks.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized experience replay.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, *30*(1).

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). Dueling network architectures for deep reinforcement learning.

Yoon, C. (2019). *Towards data science* [An article on DQN, DDQN and clipped DDQN]. https://towardsdatascience.com/double-deep-q-networks-905dd8325412