Háskólinn í Reykjavík

Introduction to Machine Learning

Project 2

Haukur Sveinsson
haukurs@ru.is

Kristófer Fannar Björnsson
kristoferb21@ru.is

Logi Sigurðarson
logis21@ru.is

supervised by
Stephan Schiffel

November 6, 2023

1 The problem

This section covers the nature of the task ahead of the group, what data is available from the environment, comparisons of models, and which challenges arise from each option.

1.1 Environment

The problem and its environment is very convenient for training a model on. Flappy Bird is a very straightforward game, and can be learnt quite well with just very few data points.

The data points representing the game's state (with height=512 and width=288) are:

- player_y: integer ranging from 0 to 381, inclusive. ($0.79 \cdot height - 24$, where 24 is the bird's image height).
- player_vel: integer ranging from -8 to 10, inclusive.
- next_pipe_dist_to_player: integer ranging from 0 to 143, inclusive. (exception to this is before the first pipe, where the player gets an offset of -75, which didn't have an effect)
- next_pipe_top_y: integer ranging from 25 to 192, inclusive. (self.pipe_height / 4 = 25, and $0.79 \cdot 0.6 \cdot height - 100/2$, where 100 is the pipe gap)
- next_pipe_bottom_y: integer ranging from 125 to 292, inclusive. (add 100 (pipe gap) to the previous)
- next_next_pipe_dist_to_player: integer ranging from 144 to 287, inclusive.
- next_next_pipe_top_y: integer ranging from 25 to 192, inclusive.
- next_next_pipe_bottom_y: integer ranging from 125 to 292, inclusive.

For each state, there are only two available actions:

1. Flap wings
2. Do nothing

Finally, a state can only be observed for every frame, and so we have a countably finite time space as well. The combination of these facts tells us that we have a discrete environment.

Within the set of states that the environment has, every action will lead to a known next state. We can confirm this by going over all of the data points making up the state.

- player_y: is dependant on the current velocity each time, as can be seen in BirdPlayer.update() method.
- player_vel: changes based on the current action taken. If the action is flap, the velocity is incremented by $-1 \cdot self.FLAP\_POWER$, which for this project is defaulted to 9. If the action is no flap, the velocity will drop by $self.GRAVITY = 1$, until max drop speed, 10, is reached.

The only arguably unknown factor is where the first non-generated pipe will be, both its height and distance, as it has some randomness to it. By looking at the code for how pipes are generated, they seem to have a constant distance between them, at half the screen's width. And whenever a pipe disappears behind the player, it is provided a new gap height randomly, with the constraints mentioned above. Still, as the newly generated pipe is not present nor relevant within the game's current or next state, the environment is classified as deterministic.

In determining whether the environment is episodic or sequential, it's fair to mention that the environment holds traits of both. The environment is episodic in the sense that it has terminal states, where it repeats an episode afterwards with the prior not affecting the newer unless with the changed policy. On the other hand, the environment is sequential in the sense that if played perfectly, an episode will not reach a terminal state, as there is no limit to how long a single flappy bird game can last. Still, as the environment is inherently episodic, and does have terminal states, it is classified as being episodic.

With all this considered, the environment can safely be classified as being a finite Markov Decision Process, as it follows the markov property, with the only information needed to estimate the best action for a state is only that current state, and no prior states in the episode. Furthermore, the states and actions are finite, and an action in a specific state will have a deterministic next state, with a reward for travelling to states.

1.2 Models

Multiple reinforcement learning tactics exist, but each has its own purpose for a specific problem. It's necessary to find out exactly what kind of model fits this problem, and its environment best, so that a good model can be learned.

As our state space is very large, a model-based approach becomes impractical. Therefore, we can safely disregard dynamic programming methods, as sweeping the entire state set. We could consider asynchronous dynamic programming, which avoids the hopelessly long sweeps, but as it is still rather computationally heavy, we consider other options.

For model-free implementations, we can consider Monte Carlo methods, as that way we can nicely reach an optimal policy. As the bird starts in a random position for every run, we're fulfilling exploring starts, as we have a probability of starting in (just about) every state, with a non zero probability of either flapping or not flapping.
Still, Monte Carlo methods don't bootstrap, and so to find a value for a single state, the entire episode (which can have an unlimited amount of actions), must be carried out, which can cause large variations between actions and can also be heavy on memory.

The solution to that problem is introduced with Temporal difference models, which update each state after every step, and therefore aren't as computationally heavy in the worst case, as well as being more memory efficient. With low enough α, a temporal difference model will also find an optimal policy, just like Monte Carlo, but because of the benefits described above, along with the fact that the state space is quite large, it is better suited for this problem. As a bonus, TD models estimate a lower error, and therefore converge more quickly than MC (if we are correct in assuming that we're dealing with a markov process).

We have two main types of temporal difference models; Sarsa and Q-learning. In reality, they act in a very similar way, with the only difference being how the value of the

next state transition is calculated, with Q-learning selecting the absolute highest valued state transition, while sarsa chooses a more safer path, using the same value function for the next state as is used for the current. However, as when actually playing flappy bird, we have full control of what we do, and don't have to account for being extra safe to avoid suffering punishments for randomness, we should rather select Q-learning as it learn the better policy with that in mind.

2 Q-learning

2.1 Q-learning Base Model

Implementing and running Q-Learning on the Flappy Bird task using a tabular approach with the defined state components led to interesting observations during the training process. The plot displays the progress of training with 20.000 episodes and then 100 episodes of testing. The highest training score was 120 and the highest test score was 745 which is pretty good.

Upon initial examination of the learning curve, it is apparent that the agent experiences a phase of learning where the scores are low and highly variable. This is expected as the agent begins to understand the consequences of its actions within the environment. Over time, there's a small but noticeable improvement, suggesting that the agent is indeed learning from its interactions with the game. The scores increase, indicating a better grasp of the task and more successful navigation through the pipes.

During the middle phase of training, the curve shows considerable fluctuations in performance but maintains an upward trajectory. These fluctuations could be attributed to the ϵ-greedy policy where ϵ is set to 0.1, allowing the agent to explore alternative strategies 10 percent of the time. While exploration is essential for learning, it can also lead to suboptimal immediate outcomes, hence the observed variability in scores.

As training progresses, the increase in scores becomes less steep, and a pattern of peaks and troughs emerges. However, the general trend does not display a clear plateau within the 20,000 episodes, suggesting that while the agent is improving, there is still room for further learning or that the agent has not yet fully converged to an optimal policy. The most notable peaks, which show exceptional performance, could represent instances where the agent's policy coincidentally aligned well with the game's immediate demands, or it may have learned specific parts of the state space more effectively.

With a constant learning rate of 0.1, the agent is continually adjusting its policy at a consistent pace, which seems to work well given the complexity of the task.
The choice of state representation, discretizing the continuous state space into 15 intervals for each relevant component (except velocity), appears to allow the agent to differentiate between different states adequately. However, it is possible that finer discretization could lead to better policy learning, at the cost of increasing the state space and possibly requiring more training episodes for convergence.

Considering the lack of a discount factor, the agent is heavily influenced by past rewards. This reward structure, with a positive reward for passing a pipe and a negative one for crashing, incentives the agent to avoid crashing.

In conclusion, the algorithm with these parameters and state representation shows promise in learning the task, as evidenced by the generally upward trend in the learning curve. Nonetheless, there is potential for improvement. Adjustments such as refining the state space discretization, experimenting with a dynamic learning rate or exploration rate, or even introducing future reward discounting could be explored to potentially enhance the performance of the Q-Learning agent on this task.

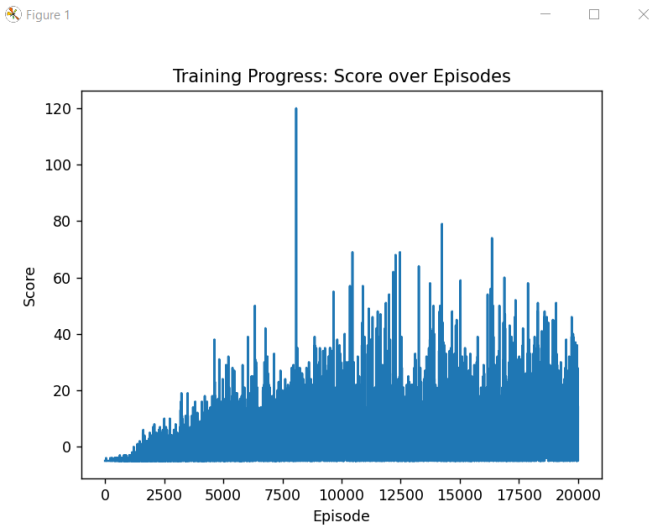

Figure 1: Model score with Q-learning

3 Deep-Q-Learning

We decided to implement the Deep-Q-Learning algorithm in Pytorch for greater flexibility and configuration. This proved to be some challenge but with the help of a Deep-Q-Learning tutorial and with some help from our dear friend ChatGPT we managed to get it working after some trial and error. Adhering to the API given in the flappyAgent proved to be doable and in the end quite nice since no other configuration of the agent was required. For comfort we added the tqdm package to monitor how many episodes of training and testing were done. This small addition proved to be extremely useful for boosting our moral and giving us the illusion of progress.
For the model architecture we used a combination of three or four linear hidden layers of 128-256 neurons with which gave us many different results. The data was normalized to be between 0 and 1 since we decided to use the RELU activation function when training the model. This was done for efficiency and shorter training time. Using the sigmoid activation function like was suggested would have increased the training time, and most likely given the same results. In fact we tried this a couple of times and this was exactly what we observed.

3.1 Base model

For the base Deep-Q-Learning model we slightly deviated from the suggested base model because of our decision to use Pytorch instead of Scikit-learn. Below is a table of the hyperparameters for the base model. We decided to use epsilon decay to regulate exploitation's versus exploration. In the beginning the model explores a lot but after around 4000-5000 episodes epsilon reaches its lower limit. We tried the high learning rate suggested (0.01) but the model kept diverging and the results thus not relevant. The base model was trained on 7500 episodes and 100 test episodes.

As can be seen be the graph in figure 2, we see that the learning curve for the Deep-Q-Learning model is erratic and diverges between 4000-5000 episodes but then finds its footing again after that. If we compare figure 2 to figure 1, we see that while the Q-Learning model continuously gets better, the Deep-Q-Learning models learning is a lot more erratic and less predictable. Our hypothesis is that when we use Deep-Q-Learning and it sometimes supposedly randomly diverges, we are really just running into the deadly triad. Deep-Q-Learning relies on bootstrapping, function approximation and off policy

Parameter	Value
Batch size	128
Discount factor (γ)	0.99
Start epsilon (ϵ_{start})	0.9
End epsilon (ϵ_{end})	0.01
Epsilon decay	100000
Soft update factor (τ)	0.005
Learning rate (lr)	$1 \times 10^{-4} \times 5$
Update frequency	100

Table 1: Hyperparameters for Deep Q-Learning

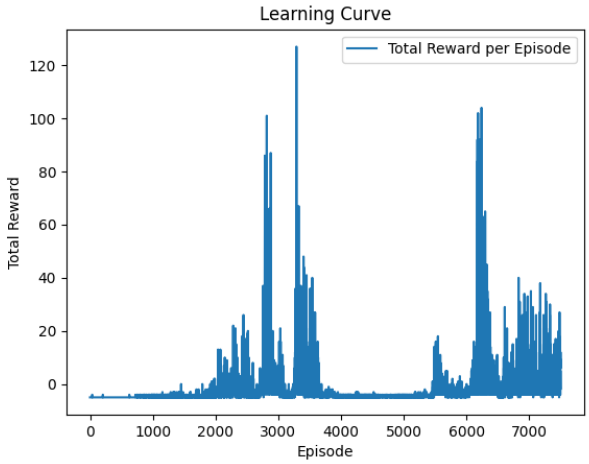

Figure 2: Learning curve of the base model.

learning. This criteria makes up the deadly triad and thus we can expect that the values might diverge for any reasonable γ and α. The highest score for the base model was 41 when testing and 130 for training. The reward structure was kept the same through out all models for consistency (+1 pipe, -5 loss).

4 The best agent

4.1 DQN Agent

Despite the poor results of the Deep-Q-Learning model we were determined to do better. Instead of only taking in four state values, we configured the neural net to take in the whole state dictionary, in total eight observations. Also the architecture for the network was changed and made more complicated. Like mentioned before, through out we decided to use the RELU activation function.

Layer	**Neurons**
Input Layer	8
Layer 1	256
Layer 2	256
Layer 3	32
Output layer	2

Table 2: Network architecture

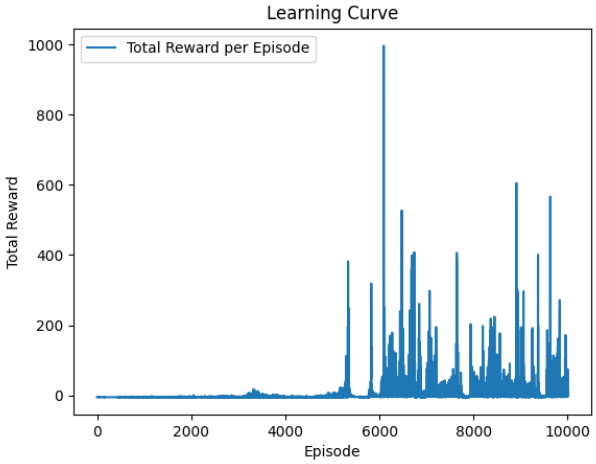

Figure 3: Learning curve of the base model.

This model seemed to do better than our base model, but as we can still see the behaviour of the network is hard to predict. This model scored though significantly higher, with a test score of 131 and a training score of 996. This ended up being our best Deep-Q-Learning model, which was disappointing but we could not seem to escape the curse of the deadly triad so we decided to move on to optimizing the Q-Learning model. Other hyperparameter descriptions of the other Deep-Q-Learning models we tried can be found in the code under "./parameters/hyperparameters.txt".

4.2 Optimized Q-Learning Agent

After the Deep-Q-Learning we choose to focus our effort on the Q-learning agents. Starting with adding some parameters for better control. We added a starting epsilon, minimum epsilon, epsilon decay and a learning rate decay. Additionally we increased the partitions from 15 to 25, hopefully for improved performance.

Parameter	**Value**
Discount factor (γ)	0.99
Start epsilon (ϵ_{start})	0.9
End epsilon (ϵ_{end})	0.05
Epsilon decay	100000
Learning rate (α)	0.25
α_{decay}	0.99
α_{end}	0.05
Partitions	25

Table 3: Hyperparameters for Q-Learning

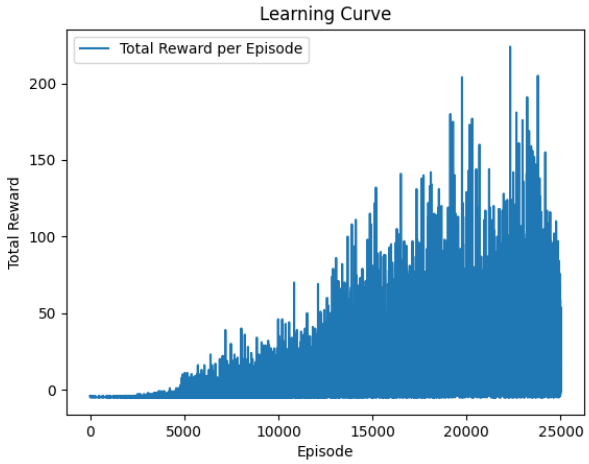

Figure 4: Learning curve of the optimized model.

This model blew our expectations out of the water, scoring 21.606 during testing and had an average score of around 2.800. This model annihilated the Deep-Q-Learning model. Additionally it only took 13 minutes and 24 seconds to train the model on 25 thousand episodes, which is around 10x faster than training the Deep-Q-Learning model.

4.3 Optimized Q-Learning Agent II

We were determined to build on the succes of our previous model. We decided to increase the partitions further and increase the minimum learning rate.

Parameter	Value
Discount factor (γ)	0.99
Start epsilon (ϵ_{start})	0.9
End epsilon (ϵ_{end})	0.05
Epsilon decay	100000
Learning rate (α)	0.25
α_{decay}	0.99
α_{end}	0.1
Partitions	30

Table 4: Hyperparameters for Q-Learning

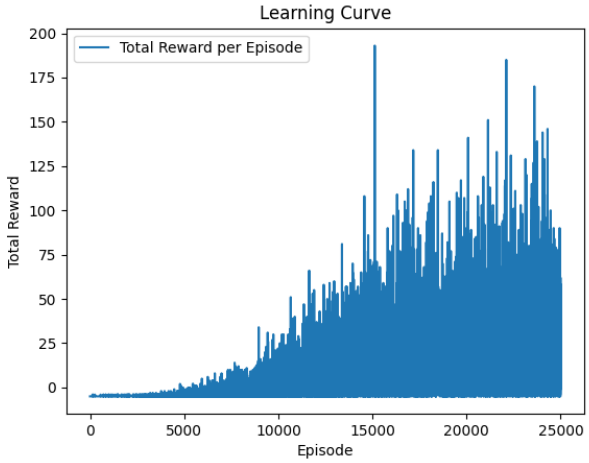

Figure 5: Learning curve of the optimized model II.

This model performed even better with a test score of 32.175 and an average score around 5.800. This will probably be our best model, and luckily, we save the state dict in our models folder, under the name "./model/q_table30000.json". This model can be loaded and tested with the load_model function. All you need to do is comment out the train function and uncomment the load_model call at the bottom of the flappy agent.

5 Conclusion

In the end, it turns out that adding some additional parameters to Q-learning increased its performance greatly, so much that we're confident a model won't easily get much better at Flappy Bird than ours, increasing the partition proved to be a good adjustment and introducing alpha and epsilon decay. We're very satisfied with our resulting model and think it will be a good competitor for the competition. This project has been a great chance to try implementing good, and then finding great models for a specific task.