Autonomous and Adaptive Systems M Project with Keras and Procgen

Krystian Koss
Department of Computer Engineering
University of Bologna
Viale del Risorgimento, 2, 40136 Bologna BO
koss.krystian@gmail.com

Abstract

The project consisted in the training of a model to make it able play several games generated by Procgen[1]. For the training the framework Tensorflow[2] with Keras[3] library have been used.

1 Introduction

This document is organized into 4 main sections, each focusing on a specific minigame of Procgen. The first one is longer than the others because it required to get in touch with the environment, so several attempts has been made in order to make the first network able to learn in a reasonable time. In this project DQN was the first algorithm exploited, it worked well with coinrun, but moving to other minigames(bossfight,fruitbot,starpilot,chaser,maze,heist,leaper) the learning was very hard. Hyperparameters and network architecture variations were not sufficient, so I decided to move to the more sophisticated algorithm ActorCritic. Here coinrun was still learned without too difficulty, but this algorithm was extremely sensitive to hyperparameters modifications. Seeing no big improvements and high training instability I got back on DQN, accepting its performance. The games were selected to be distinct from one another: coinrun, leaper, bossfight, starpilot. All training was made on the first 20 levels of each game starting with no background, then adding difficulty in proportion to the learning capabilities. Model evaluations were done only on DQN networks, Actor critic ones were not considered as they not showed particular better abilities in gaming.

2 Coinrun

2.1 DQN

The learninig of this game consisted in starting from playing 20 levels with these Hyperparameters: alpha=0.9 , gamma=0.95 , episodes=500 , learningrate=0.001 , startEpsilon=0.7 , discountEpsilon=0.0003 , lowerBoundEpsilon=0.3 and DQN as algorithm. The tests were made on 500 episodes at a time, saving the weights of the network at the end of the learning in order to run other rounds on the weights just learned. The epsilon started from a value of 0.7 and with a discount of 0.0003 each episode until it reaches lower bound of 0.3. Lower values were tried but the player was used to get stuck in some places, unable to escape local minimums. At the very beginning, also the network architecture had to be decided, so a very simple network with few layers (3-5) has been adopted. These results were bringing nowhere with no apparent learning, but the addition of a convolutional network showed great improvements. So the final network looked like this:

```
model_Q = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(15)
])
```

The CNN architecture was taken from Tensorflow main CNN tutorial[4]. The Dropout was added to make the learning more stable and avoid overfitting. The input shape consisted of a colorized image of 64x64 pixels, and the possible outputs the environment of Gym[5] accepted were numbers of actions from 0 to 14. Not every action was connected to a real action of the player, but this was not a problem as it was expected the network was able to learn this. During the training the main variations that had a strong impact on the learning were about the credit assignment problem: three strategies were introduced to find the best way of managing the score distributions.

2.1.1 DQN with Monte Carlo method, step penalty and replay buffer

In this case I adopted a Monte Carlo update of the network at the end of each episode on a batch of 500 steps on a replay buffer with dimension 10000, just to make the updates less correlated. The rewards were: +10 if the player managed to reach the coin, -10 if the game ended both for timeout and death, -0.05 as step penalty. The introduction of step penalty must always be deeply analyzed, as could be situations were an imminent death brings more score than walking in loop waiting the timeout. We can say that using step penalty makes the player fall in local minimums more frequently.

At the end of each episode all the steps were re-elaborated, setting all the rewards to the same value, calculated summing all the rewards of that episode. The figure shows the training graph of the third round of 500 episodes(first two were made without background and easy mode, while the last one with max difficulty).

Figure 1 shows the final training of a network that understood how to play coinrun pretty well, but was not able to avoid with high probability circular saws and moving enemies. This is probably because the threats that cause death of the player are small compared to the whole image, so more rounds of training should be made, as the network is not only learning how to play, but it is also learning how to see.

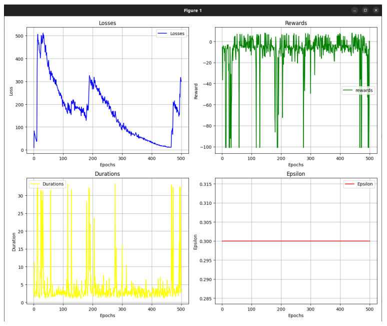


Figure 1: DQN with Monte Carlo method, step penalty, replay buffer. Third round of training

Just from this graph we can't say much about the real performance, we need to test it counting how many wins it makes and comparing them with a random player, or it is sufficient to look at the player and see if it behaves in a way it looks "smart". To better understanding, the evaluation strategy adopted during training was the second one, with this method it is also possible to see some misbehaviour of the agent which helps us to introduce more aimed solutions, rather than just knowing the victories or defeats compared with a random player.

A last interesting note to mention: given this successful training, a learning with same conditions but starting directly with background and max difficulty has been tried, resulting in not learning much. This is probably due to an exponential increase of the search space. At the same time, starting the third round with hard mode and background after the network learned the game without them, did not introduced too much instability to the network. This observations can be justified saying that: the background is not so relevant for this exact game, and once the player learned the game without it, the network learned to ignore it, so introducing it after some training it did not affected the learning too much.

2.1.2 DQN with Monte Carlo method, no step penalty and replay buffer

This variation is the same of the previous one, without the step penalty, meaning that the rewards of all steps at the end of each episodes were all +10 or -10. Figure 2 shows how the third round of training performed:

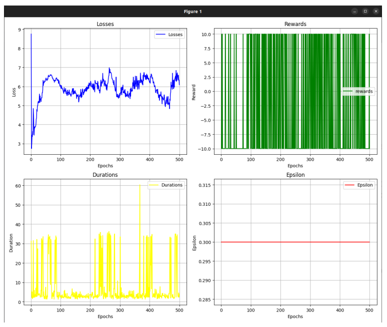


Figure 2: DQN with Monte Carlo method, no step penalty, replay buffer. Third round of training

Talking about performance, this solution to credit assignment problem seemed the best one, playing slightly better than the version with step penalty. Here we have a more stable learning, but if we look a the duration of each episode(yellow line) we see a "ladder pattern", this was probably caused by a to high batch size(500).

A small note we can make: in the duration graph(yellow) there is a line that is higher than all the others. This is because using time as a evaluation parameter is not a wise choice, in fact that episode lasted longer because my computer freezed for a while, infecting the test results. It was better to choose step numbers.

2.1.3 DQN with TD(0), no step penalty and replay buffer

With this variation we have a one step temporal difference method update, so each action received only the single reward it gained in the game. Rewards types were: +10 if the agent gets a coin and -10 if it dies or timeouts. Figure 3 shows the third round of training with lower epsilon discount rate:

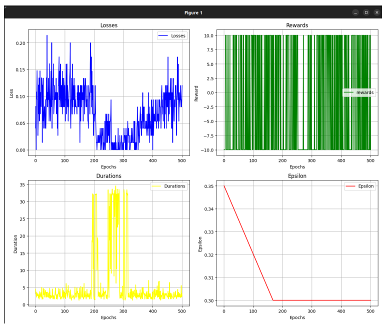


Figure 3: DQN with TD(0), no step penalty, replay buffer. Third round of training

Its performance was a disaster, the random player played far better.

After many attempts of using DQN with other games didn't get great results i decided to move to a more sophisticated algorithm called Actor Critic, starting again with coinrun and then extending it to other games.

2.2 Actor Critic

Here the hyperparameters were identical to DQN ones, and both learning rates of the two networks had same value 0.001. The rewards were +10 if coin reached, -10 if match lost. The first attempts with a replay buffer brought to a situation called "saturation of the network", where an action reached probability close to 1.0 after few training episodes. The cause of this problem was the learning rate too high, but after setting it to 0.00001 it solved the problem. An other issue was that all output probabilities became all NaN, due to the a batch size too big(100). A size of 32 was the solution. Despite the positive troubleshooting the network seemed not learning much.

After removing the replay buffer that does works better with DQN, clipping the rewards in the range +1/-1, starting the training with only one level, setting the learning rate to 0.000001, normalizing the inputs[6] and simplifying the network architecture improved the learning. The final architecture used was as follows:

```
oi = tf.keras.initializers.Orthogonal()

model_Actor = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, kernel_size=8, strides=4, activation='relu',
            input_shape=(64,64,3)),
            tf.keras.layers.Conv2D(64, kernel_size=4, strides=2, activation='relu'),
            tf.keras.layers.Conv2D(64, kernel_size=3, strides=1, activation='relu'),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(512, activation='relu'),
            tf.keras.layers.Dense(15, activation='softmax')
        ])

model_Critic = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, kernel_size=8, strides=4, activation='relu',
            input_shape=(64,64,3)),
            tf.keras.layers.Conv2D(64, kernel_size=4, strides=2, activation='relu'),
            tf.keras.layers.Conv2D(64, kernel_size=3, strides=1, activation='relu'),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(512, activation='relu'),
            tf.keras.layers.Dense(1)
        ])
```

The CNN architecture was taken from "Human-level control through deep reinforcement learning" paper[7], and orthogonal feature was introduced to start each training without any initial bias. Figure 4 shows the fourth round of training with 100 episodes each round and dividing the learning rate by 10 at each new round, otherwise the stability would have been affected.

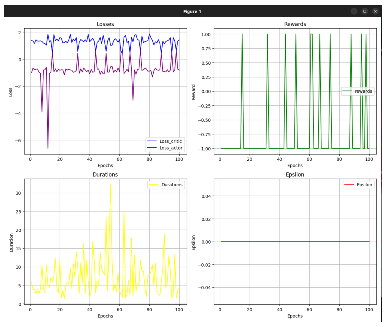


Figure 4: Actor Critic learning. Fourth round on single level

After these experiments we can say the player seemed to be able learning and that the Actor Critic algorithm is very sensitive, if you set wrong the parameters the learning is heavily influenced.

General observation: DQN coinrun network seems to choose always the action jump-right regardless the state, this is because a similar behaviour brings to a good score. With Actor critic the player looked "smarter" by not choosing always the same action, changing also distribution probabilities of the policy whenever there was a change of scenario in the frame(it is possible to see the video in the github repo).

2.3 Model evaluation

Tests were made on DQN with Monte Carlo method, no step penalty and replay buffer; they got the following results:

- Real player has completed the games with an average of: 56.35 steps, 6.24 rewards, 312 wins
- Random player has completed the games with an average of: 623.468 steps, 2.62 rewards, 131 wins

3 Leaper

This minigame is different from coinrun, but reward policy was similar (only final reward of +10 if the agent reached the end). Hyperparameters remained the same as in previous coinrun training, grayscale was used and rewards were: +10 if winning and -10 if losing, applying a reward distribution with Monte Carlo approach.

3.1 DQN with Monte Carlo Methods and Replay buffer

Also here both TD(0) and Monte Carlo has been exploited, resulting in a victory of the second one. During the learning epsilon value was kept high (0.9) for first round, then with discount of 0.0003 each episode until it reached 0. When this condition was reached the network was playing worse than random, and in the levels where there was no obstacles but just the finish line, the agent strangely was not able to finish the game. To solve this problem was sufficient to train another round on a epsilon of constant 0, focusing the network on exploitation rather than exploration. Figure 5 shows the last round of training:

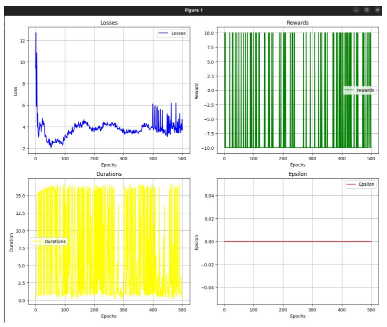


Figure 5: DQN with Monte Carlo and replay buffer. Seventh round of training

3.2 Model evaluation

With 500 episodes par player(network and random), no background, all levels starting from 50 with easy mode the results are the following:

- Real player has completed the games with an average of: 96.856 steps, 2.6 rewards, 130 wins
- Random player has completed the games with an average of: 113.024 steps, 2.22 rewards, 111 wins

4 Bossfight

This training has been divided in two parts: first rounds were made upon Monte Carlo, then with TD(0) updates.

4.1 DQN with Monte Carlo method

This minigame is pretty different from coinrun: rewards are scarce, there is an enemy to defeat which constantly shoots and at the beginning of a play the first 8 seconds the enemy's ship has a shield that makes it immortal. For this reason, could be a good idea to add, togheter to standard reward policy, a small reward (+0,1) for each step that keeps the player alive, as a player that knows how to avoid enemy's bullets has much more probability to win the game. This led the player hide in the corners, lasting more steps than random player, but without giving too importance on defeating the enemy.

4.2 DQN with TD(0) method

At this point, I wanted to make the agent understand that shooting was profitable by adopting a TD(0) update with rewards +40 if ship destroyed and -20 for end of game. Those exact values has been chosen because in previous Monte Carlo training every step got on average from 15 to 30 points, so a higher reward had to be added to tell the network we give more importance to shooting actions. TD(0) approach gave us the ability of rewarding the exact steps that led to victory, while using an epsilon of 0 in order to focus on exploitation.

4.3 Model evaluation

The performance of the training over 500 episodes on all levels, with no background and easy mode were:

- Real player has completed the games with an average of: 172.958 steps, 0.316 rewards, 11 wins
- Random player has completed the games with an average of: 47.678 steps, 0.056 rewards, 2 wins

5 Starpilot

5.1 DQN with replay buffer

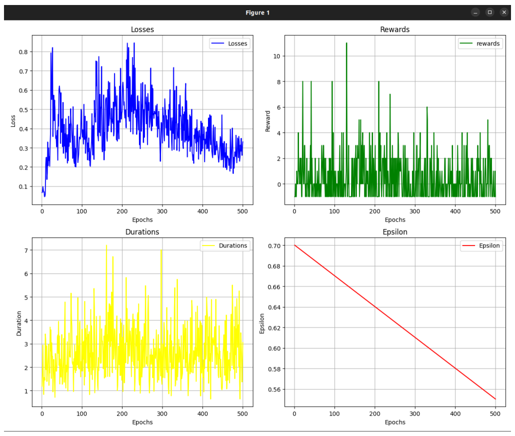
Here together with grayscale wrapper it has been used framestack with the last three frames. The rewards were the same of the default setting, plus a -1 each time the agent dies, trying to instill the behaviour of dodging the bullets.

5.1.1 DQN with TD(0) methods

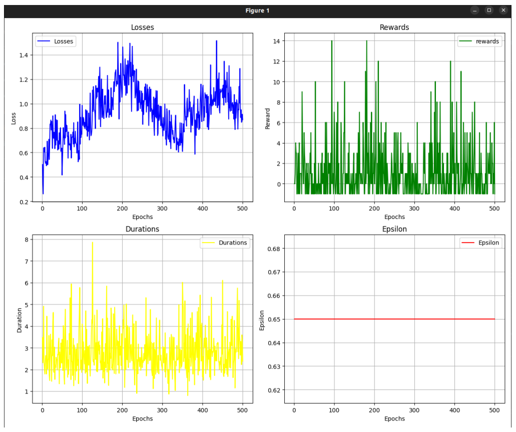
This strategy gave to each action the reward it gained in the game, but no relevant learning showed up. Also a strategy like the one used with bossfight(first Monte Carlo, then TD(0) with epsilon 0) did not introduce any improvement.

5.1.2 DQN with Monte Carlo methods

Giving the total reward of the whole episode to all actions plus the single reward they gained in the game led to improvements, as we can see in Figure 6. This strategy probably gives less emphasis to the negative reward introduced to avoid bullets, thinning the difference of states that caused death and those who didn't.



(a) DQN with Monte Carlo, replay buffer. First round of training.



(b) DQN with Monte Carlo, replay buffer. Fourth round of training.

Figure 6: Comparison of training rounds

We can see little general improvement, but during the real playing what the model does is it mainly select the action "shoot forward" and sometimes it changes position. This is because the states that gained more rewards were the ones that shot, regardless the enemies position, as it is the only way to obtain rewards.

An important observation: in this game during training the epsilon values had to be very high (0.65), the first times I applied the same discount epsilon factor as for previous games but I realized it was too high, probably because with many enemies and many bullets every time in a different position makes the state space exponentially big, requiring a wider exploration.

5.2 Model evaluation

DQN with Monte Carlo methods was the winning network(tested on 500 episodes as always), slightly better than random: in the video provided it performs quite poorly, but in the long run it gathers more points:

- Real player has completed the games with an average of: 73.656 steps, 1.638 rewards, 0 wins
- Random player has completed the games with an average of: 78.594 steps, 1.46 rewards, 0 wins

6 Conclusions

Training a network to perform a task is much harder than I expected. It is true that procgen benchmark is a difficult one, which requires the network to learn both how to see the frames and how to play the game without knowing which is the player or anything about rules. But personally, I thought to achieve far better results than these "slightly above random" performance, at least I expected to see some "intelligent behaviours" in the agent. This expectation had gripped me for some time, reading also around internet of people able to train a network over more difficult games like old version of Doom, I started with DQN trying many times refusing to admit that there was no way to learn well the games, I am probably missing some knowledge that could be helpful to unblock the situation. I moved to Actor Critic hoping that with a more sophisticated algorithm the results would be better, but what I found is that algorithm is extremely sensitive to hyperparameters variation and the learning was not so impressive. For that reason I moved back to DQN and accepted the results I was able to obtain with that more simple and robust algorithm.

References

[1] Procgen, https://github.com/openai/procgen
[2] Tensorflow, https://www.tensorflow.org/
[3] Keras, https://keras.io/
[4] Tensorflow, https://www.tensorflow.org/tutorials/images/cnn
[5] Gymnasium, https://gymnasium.farama.org/
[6] Reinforcement Learning Implementation Tips and Tricks, https://agents.inf.ed.ac.uk/blog/reinforcement-learning-implementation-tricks/
[7] Human-level control through deep reinforcement learning, https://www.nature.com/articles/nature14236