Navigation project

Project in Deep reinforcement learning as part of a Nano degree.

26/11/2019
Jonas le Fevre Sejersen
Jonas.le.fevre@gmail.com

1 The scope of the project

For this project, we will train an agent to navigate (and collect bananas!) in a large, square world. The rules are quite simple, a reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

- **0** - move forward (or do nothing).
- **1** - move backward.
- **2** - turn left
- **3** - turn right

The task is episodic, and in order to solve the environment, our agent must get an average score of +13 over 100 consecutive episodes (spoiler it will, but only barely).

In this project, I chose to implement the rainbow model to give myself a challenge and to learn the different aspects of the improved DQN. In the next couple of section, we will go briefly through what rainbow model is and how it tries to improve some of the pitfalls of DQN.

Important note is that I have somewhere in the code (that i cannot locate) that i have a true random variable, even though i tried to seed all of them for the purpose of comparison. So each training might have a slightly different result.

2 Rainbow

The Rainbow DQN model takes the standard Deep Q-Network and combines a lot of different improvements that together forms a more robust and faster model. The below is a listing of the papers used to form the rainbow (it is links to the paper):

1. Double Q-learning
2. Prioritised Experience Replay buffer
3. Dueling Network
4. Noisy Net for Exploration
5. N-step Q-learning
6. Categorical DQN

In the next couple of section we will look into some of the key features of each improvement, starting with Double Q-learning.

2.1 Double Q-learning

Deep Q-learning has this tendency to overestimate actions values, since what it essentially trying to do is update a guess value based on another guess value, and this can potentially lead to unwanted correlations. We can lower this tendency, we can introduce a section network that are not changed during the learning steps. This way we are trying to base our guess on a more static value which leads to a less overestimation in the action value. The main idea is to use two separate networks with identical architectures. Lets call it target Q-Network and primary Q-Network, where the target Q-Network is updated less often to have a stable (fixed) target $\hat{q}(S', a, w^-)$. To compute the Q value we can do as follows:

$$q(s,a) = r(s,a) + \gamma \hat{q}\big(s', argmax_a q(s',a)\big)$$

Therefore, Double DQN helps us reduce the overestimation of q values and, as a consequence, helps us train faster and have more stable learning.

2.2 Prioritised Experience Replay buffer

Another common improvement is gathering experiences in a replay buffer and uniformly selecting a batch of these experiences to train the model. What is does, is that it separates the exploration aspect of the network from the updating functionality, this way improving the stability of the DQN during training.

Prioritised Experience Replay (here by refereed as PER) build upon the same functionality by storing the experiences potential influence on the q value as a priority value. Now instead of sampling batches uniformly, we compute a probability of being chosen based on how high the priority is.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

To not always pick the highest priority replays, we introduces the hyper-parameter α where $0 \leq \alpha \leq 1$, which reintroduces some randomness. What we gain from using PER is that rare experiences are chosen more often makes the model more resistance to cases we do not see as often.

2.3 Dueling Network

Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values **for each action**. Q-values correspond to how good it is to be at that state and taking an action at that state Q(s,a). However by decompose Q(s,a) as the sum of:

- **V(s)**: the value of being at that state
- **A(s,a)**: the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state).

$$Q(s,a) = V(s) + A(s,a)$$

By doing so we can assess the value of each state, without having to learn the effect of each action.

We do this by splitting the network into two streams, one for the state value and one for the advantage value for each action. At the output layer for of our network we use an aggregation layer: $Q(s,a) = value + (advantage - mean(advantage))$

This architecture helps us accelerate the training. We can calculate the value of a state without calculating the Q(s,a) for each action at that state. And it can help us find much more reliable Q values for each action by decoupling the estimation between two streams.

2.4 Noisy Network

The normal DQN uses an epsilon greedy approach, where epsilon dictates how often the agent chooses a random action. As training goes on, the epsilon value will decrease to lower exploration and stay truer to the policy. The noisy network paper is proposing another way of balancing exploration and acting accordingly to the current policy. Instead of using epsilon to dictate randomness, we are incorporating noise into the weights and biases of our value and action streams. This introduces exploration through noisy outputs in the early training stages, but as the training goes on, the agent will learn to minimise the added noise when the correct action becomes clear and hereby act accordingly to the policy.

2.5 N-step Q-learning

N-step Q-learning aims to improve the temporal awareness of the agent. This is done by storing the experiences as $< s_t, a, R, s_t + n, done >$, where the reward R is the discounted sum of rewards over the next N sequential experiences. The $s_t + n$ is the state we end in after the N-steps we took. What this does is giving the agent a broader understanding how actions and rewards are tired together and how a sequential of states can be coupled together as a positive experience despite of getting some negative rewards on the way.

2.6 Categorical DQN

The categorical or distributional DQN is trying to learn the value distribution over all possible outcomes: the distribution of the random return received by the agent, instead of the common approach of modelling the expectation of the value. The main motivation for using the categorical model is that distributions may have several peaks. Just averaging them in one expected value may be inappropriate and lead to inadequate results. What they did is to replace the output of DQN with probabilities over categorical variables representing different reward values ranges (from v_min to v_max). The number of bins chosen is called atoms and is essentially how many reward values we are projecting in too. If a reward is larger than v_max or lower than v_min they are projected un too these outer bins.

3 Hyper-parameters

This section start to leads up to the result section by explaining some of the hyper-parameters used to produce the final result. We will go through general, model, agent, n-step PER and training information. Here starting with the general info.

General info

The general info is info about the game and what seed have been used:

```
game: Banana.exe
seed: 0
state_size: 37
action_size: 4
```

Not much to note here since it is the basic game data and the only real changeable parameter is the seed.

Model info

The model info is the hyper-parameters for the model structure. I have chosen to only make the initial sigma value: σ_0 a changeable parameter and let the structure of the model (the number of layers and neuron) be constant. I use 512 neurons and 1 hidden layer for the value stream and advantage stream. The hyper-parameter is set as follows:

```
std_init: 0.2
```

They use 0.5 in the paper, but after testing various values I found the 0.2 to be giving better result in fewer episodes.

Agent info

The agent info is the hyper-parameters used to configure the rainbow agent.

```
GAMMA: 0.95                         # Discount value
TAU: 0.001                          # Amount we update the target model each update
LR: 5e-5                            # The learning rate
opt_eps: 1.5e-4                     # Adam epsilon
UPDATE_MODEL_EVERY: 10              # The number of steps between model updates
UPDATE_TARGET_EVERY: 8000           # The number of steps between target updates
use_soft_update: True               # Whether to hard or soft update target model
priority_method: reward             # How to initialise priorities for PER
atom_size: 51                       # Number of atoms (steps used in support)
v_max: 1                            # Max value represented in network output
v_min: - v_max                      # Min value represented in network output
```

I found that using a lower discount value (0.95) and a higher n-step (20) gave a good result since we prioritise more steps higher if it ends in a reward, but by setting the discount value lower, we still value the steps closer to reward better than the steps far from a reward. Since we use a soft update method for our target model, each update call will also update the target model's weights but only slightly (Tau amount). Another observation is the UPDATE_TARGET_EVERY parameter becomes not use.

When looking at the categorical DQN hyper-parameter, I have used a very small value and they are ranging from -1 to 1. I have no real explanation of why these are good for this task, but this is tested to be a fair range to use.

N-step PER info

The N-step PER info is both the information to configure the PER implementation and the N-step functionality.

```
BUFFER_SIZE: 2**20              # The space we use to store Experiences.
BATCH_SIZE:512                  # Amount of replays we train on each update.
PER_e: 0.01                     # Epsilon : Small value we add to prevent zero priority.
PER_a: 0.6                      # Alpha : .
PER_b: 0.4                      # Beta init : .
PER_bi: 1e-05                   # Beta increase : choosing more prioritised exp.
PER_aeu: 1                      # Absolute error upper is the max priority a replay can have.
PER_learn_start 0               # Used to populated the sumtree with replays.
n_step 20                       # For choosing how many sequential replays to use.
```

As previous stated, we are using a big n_step parameter, which decide how many sequential replays is used to compute a reward. Since PER_learn_start is 0, we do not populate the replay buffer before training, we only wait for it to have at least batch_size amount of replays.

Training info

Lastly is the training info, which is simply how many episodes we want to train for and how often we want to evaluate the model.

```
episodes: 1000
evaluation_interval: 100
```

In the next section we take a look at the result of both training phase of the model and the evaluation steps done through training.

4 Results

Every model trained is saved to the folder called *saved/* and then a sub-folder is created called test and a number generated, where a **model_test.md** is placed containing all the hyper-parameters and the result of that run. A plot is also generated called **plot.png** (if the code is executed from jupyter notebook, the plot will then not be saved correctly only a empty plot is there). The plot is showing a blue line being the mean value of the last 5 episodes and a olive coloured line representing the rolling average over 30 episodes. These are created this way to keep the plot easy to read, since due to exploration each episode vary a lot in score. Lastly the folder contains the trained model weights in called **rainbow_checkpoint.pth**, this is the file you would load to use the trained model. In this section we are going through the result of *test24* that will be located in the saved folder, since i have a problem with my seeding i cannot recreate the same result each time I train so this is the training run i decided to cover.

Training

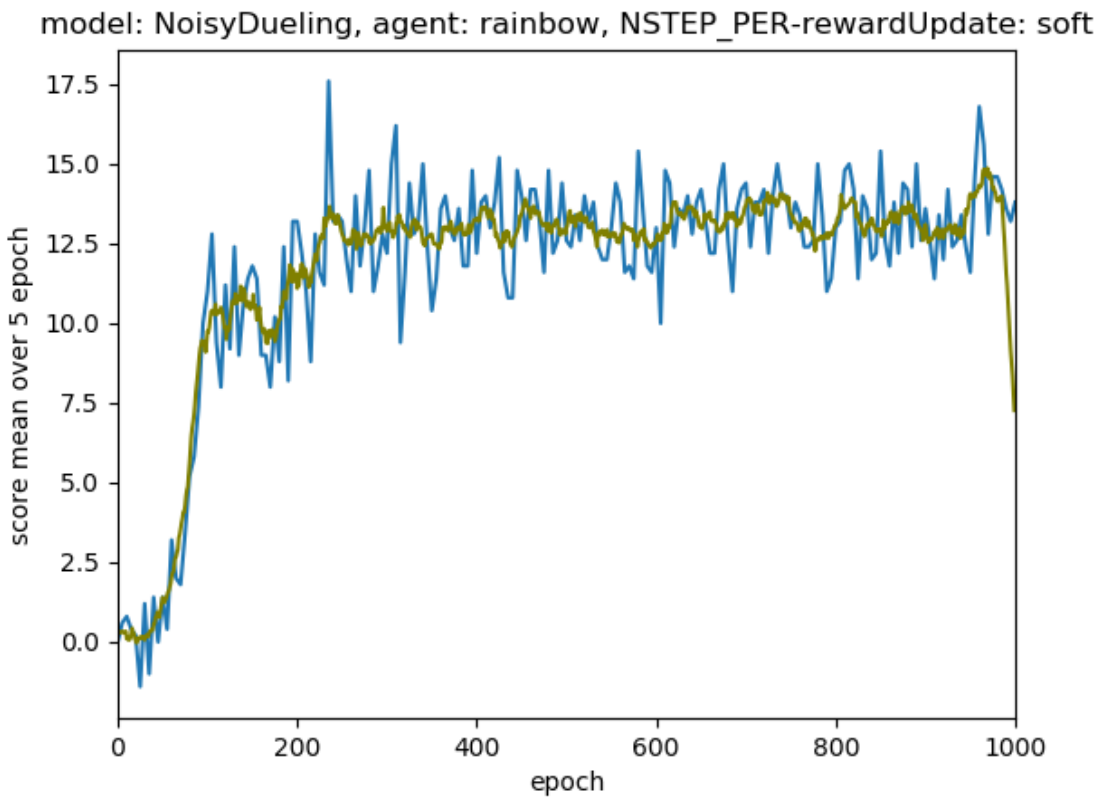

Fig. 1: The y-axis is the mean score of the last 5 episodes and the x-axis is the associated episode

In figure 1 a plot over the training period representing the scores on the y-axis and the episodes on the x-asis, we can make the observation that during the first 90 episodes the agent is exploring the state space and try to couple the state of running over a yellow banana give good rewards and run into blue bananas gives bad rewards. After the first 90 episodes it is clear to see that this link is created and we see a massive improvement in the scores. If we look at the training data listed below, we can see that the training starts to have an average score of around 13 after 400 training episodes and is slowly increasing further as it keeps training.

```
Episode 100     Average Score: 2.69
Episode 200     Average Score: 10.46
Episode 300     Average Score: 12.47
Episode 400     Average Score: 12.96
Episode 500     Average Score: 13.21
Episode 600     Average Score: 12.86
Episode 700     Average Score: 13.30
Episode 800     Average Score: 13.24
Episode 900     Average Score: 13.42
Episode 1000    Average Score: 13.49
```

Evaluation

Every 100 training episodes an evaluation is done to test the current progress of the model.

```
train_episode: 100      Average Score over 100 episodes: 9.86
train_episode: 200      Average Score over 100 episodes: 9.72
train_episode: 300      Average Score over 100 episodes: 13.16
train_episode: 400      Average Score over 100 episodes: 12.6
train_episode: 500      Average Score over 100 episodes: 12.69
train_episode: 600      Average Score over 100 episodes: 13.32
train_episode: 700      Average Score over 100 episodes: 13.34
train_episode: 800      Average Score over 100 episodes: 13.55
train_episode: 900      Average Score over 100 episodes: 13.2
train_episode: 1000     Average Score over 100 episodes: 13.22
```

We can see that the model is performing better than the 13 average score over 100 episodes after 300 episodes of training. It is then staying just around the 13 for the remaining of the evaluation rounds.

The score of 13 is not impressive results and not given the time it took to get to the state. This is why we have some ways we might be able to improve the solutions.

5 Future improvement

While we did manage to get solve the project (barely), there should be a lot of improvements to be made. An observation I have made is adding the complexity of Categorical DQN actually made the model train slower and needs more episodes before completion. Without Categorical DQN, I solved the project in 161 episodes and after 200 episodes the average score was stable between 14 to 15. What this tells us is that either my implementation of Categorical DQN (C51) is wrong somewhere or my hyper-parameters are completely off. I have tested a various settings of the hyper-parameters but none of them seem to improve the model much. I hope that this review can help me find the flaw in my implementation or at least guide me some of the way.

In the next section we will look at a few question i have for the reviewers.

6 Questions for reviewers

This section is for questions to the reviewers. I have chosen to upload an implementation of the rainbow model, despite that it didn't perform that well. On the way of implementing the rainbow agent, I have had much better-looking results, but I wanted to know why this model isn't performing well. I want you to take a very good look at the code and be very critically on every little detail so I can learn to improve myself. Here are some of my Questions:

1. It seems that even though the model learns the noisy parameters, the scores vary a lot. In the epsilon greedy we control this by lowering epsilon over time, but here it seems that it keeps exploring too much to increase the performance. Is this a true observation or just a rubbish theory?

2. Since the Categorical DQN implementation didn't improve my model, I might have missed the point of why the distribution should be better than just computing the expected value.

3. Does my hyper-parameters look completely off?

4. The v_min and v_max parameter is the minimal and maximum value that will be represented in the network output for predicting the Q value. These are used to create the support vector together with the number of atoms (steps in the support). Which we use to project the distribution on to. They state (section 5.1 in the paper) 'From the data, it is clear that using too few atoms can lead to poor behaviour, and that more always increases performance;'. The more atoms we have the slower the training time, that is true since the model is getting more notes we have to train. Here is the question: Since my model perform badly am I using too few atoms? (I have not tested this)

5. Here is a question about the noisy network. I have observed that the std_init value works alright okay for me at 0.2, but why does it matter what value it is? The noise is static, so the network has to learn to minimise the noise no matter what init value I chose?

6. Another noisy network question. When training the network, the longer the training session i use the more the results starts to variate? Is there a way to tune down the amount of exploration, and why is this happening after the model has started to stabelise?

What I really need is someone to look through the code and sanity check that it is not wrong what I am doing since hyper-parameter optimisation is not that important as long as I know my code is correct.