Deep Learning laboratory Report

Assignment 4

Mohamed AbouHussein
Louay Abdelgawad

Department of Machine Learning
University of Freiburg
November 2016

Summary

This report is related to the work of implementing a deep Q-learning neural network using `tensorflow as tf` and `keras` package for motion planning. The report is divided into 5 sections as follows: section 1 is the solution for exercise 1 in the sheet, section 2 represent a simple explanation of the deep Q-learning network implementation for solving the problem in exercise 2, section 3 represents the designed deep Q-learning neural network, section 4 represents the results and discussion and section 5 mentions several possible improvements.

Report

1 Q-Learning Exercise 1

This section represents the solution for the questions in part 1 of the sheet.

1.1

Initially the Q-value for the goal terminating state is zero and having a reward of 0 which is higher than the reward of the rest of the non terminal grid locations Q-value. Therefore, it will have a higher Q-value. When the terminal state is reached, the algorithm starts from a random initial state (i.e, starting a new episode) and then it keeps repeating until the values converges or there is no more change in the policy or the number of training episodes is reached.

1.2

Assigning action values as:
Left: 0
Up: 1
Right: 2
Down: 3
And also assigning the Grid values to start with 0 on the top left and 9 on the bottom right. The updated Q values would be:
$Q[0, 3] = 0 + 1 * (-1 + 0.5(0) - 0) = -1$
$Q[3, 2] = 0 + 1 * (-1 + 0.5(0) - 0) = -1$
$Q[4, 1] = 0 + 1 * (-1 + 0.5(0) - 0) = -1$
$Q[4, 2] = 0 + 1 * (-1 + 0.5(0) - 0) = -1$
$Q[5, 1] = 0 + 1 * (0) = 0$

An improvement to this would be by gradually reducing the learning rate after some initial episodes to include the learned Q-values and also making a reward of 1 instead of 0 for the goal state.

2 Convolutional Neural Network Implementation

In this algorithm we do not load any data. This is because in deep Q-learning networks, the agent/solver react through exploring the space and getting rewards as explained later. In the `train_agent.py` the `Simulator` and the `TransitionTable`. The functions of both objects will be used later. We begin by initializing the `model` using `keras` package that acts over `tensorflow`. The functions we utilized from `keras` are `Convolutional2D`, `Activation`, `MaxPooling2D`, `Flatten`, `Dense` and `Dropout`. After that, we initialize the number of learning steps to 1 million iterations. The algorithm begins by randomly defining actions and updating the `TransitionTable`. There's two variables, `explore` and `observe` that define when to generate random actions, when to generate action through the network and when to train. through time, the algorithm reduce the rate of random actions and increase the rate of network actions and starts iteration after a certain amount of population the `TransitionTable`. Afterwards, a `batch` of 32 samples are selected from the `TransitionTable`. Q-values from the network are generated from the network for the batch. From the `TransitionTable` we retrieve `state_batch`, `action_batch`, `next_state_batch`, `reward_batch` and `terminal_batch`. We compute the `next_batch_actions`. The loss function is computed and feed to the optimizer based on the mean square error function

$$L = \frac{1}{2}[r + max_{action\_next}Q(state\_next, action_next) - Q(state, action)]^2 \quad (1)$$

T

he exact CNN architecture will be illustrated in the next section.

3 Designing Deep Q-learning Networks

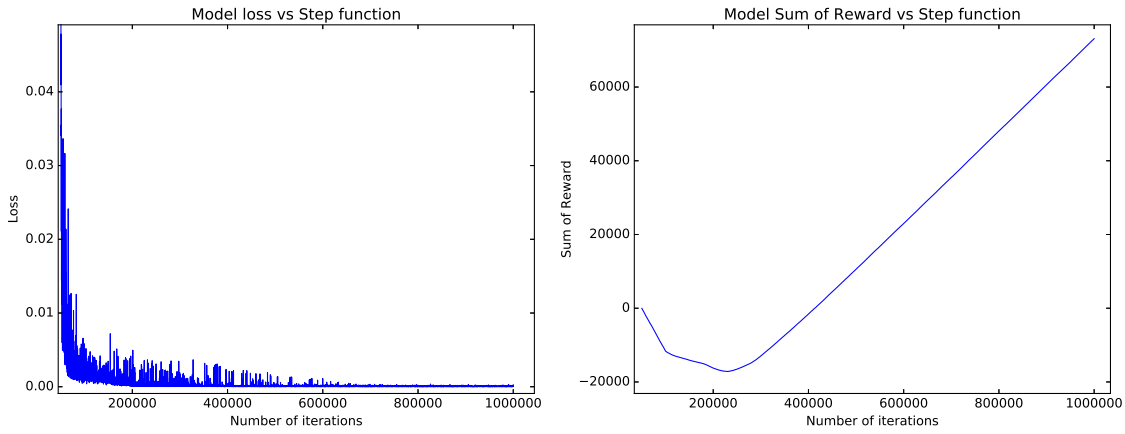
For the design of the Deep Q-learning network, the procedure utilized was to create a single convolutional layer at a time. The layer parameters was randomly tried out from a range of values. The best estimated random values was the one implemented. Afterwards, another layer is added and the same procedure is tried out. In addition, the hypermaraters is also set in similar manner. The previous procedure has resulted in the following: The first conv layer included 32 `Covolutional2D()` filters, 8x8 `filter_size`. The activation function for the first layer was `relu`. Another `Convolutional2D()` layer is added that has the same specs and activation function but of size 64 instead, after that a third one was added with 64 filters and 3x3 `filter_size`. Finally, the data is `Flatten()`ed and then it was reduced to 32 units using `Dense()`, afterwards it was reduced to the number of actions. Moreover, `relu` function is used on the output layer and Mean squared error `mse` is selected using Adam optimizer as the optimization technique for loss correction. The `batch_size` selected was 32. The results of the DQN are discussed in section 3.

4 Results and Performance Evaluation

4.1 Training Single Q-model

This section illustrates our first Q-learning model trial. A single Q-model was implemented and was updated in every step of the training. The Q-values of the `batch_state` and the `batch_state_next` was computed from the same model. In addition, weights of the model was updated in every training step using the loss function equation 1. The result of this model was not satisfactory at 600k, 800k and 1 million iteration runs. Though the loss function converged, yet the agent when tested was hovering in place and was not moving to target at all. A modified improved concept is described in the next subsection.

4.2 Training Q-model and Target-model

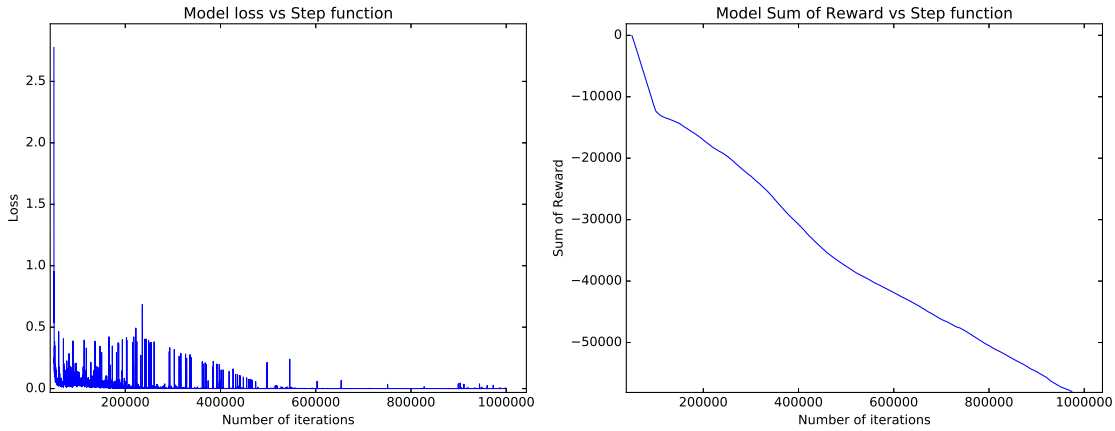


(a) loss vs steps

(b) Sum_Reward vs steps

Figure 1: Network descirbed in the Report

In this improved method, we initiate two different models for the `state_batch` and `state_batch_next`. They are named `model_q` and `model_target` respectively. The first is used to generate Q-values to the `state_batch` while the latter is used to generate Q-values for `state_batch_next`. The loss function equation in 1 is computed using the difference of the Q-values output from these models. Furthermore, every 10,000 iterations, the values of the weights of the Q-model is copied to the values of the Target-model. With respect to the loss function, the loss value converged. In addition, the model was tested at 300k, 600k, 800k and 1 million iterations. As for the 300k, the agent exhibited a similar behavior to that of the single Q-model. However, the 600k, 800k and the 1 million training iterations weighed models were optimal. They all solved the problem with minimum distance every time from all the random initial locations. It can be seen in graph 1a, which is the graph including the trained Network discussed in this report, that the loss starts low and decreases gradually. Moreover, in graph 1b the `reward_sum`, which is the sum of



(a) loss vs steps

(b) Sum_Reward vs steps

Figure 2: One conv2d Removed and with dense output of 512

total rewards according to the action taken, first starts decreasing at a very high rate as the actions taken were random, until reaching 100,000 steps, were the steepness decreases gradually as the actions were partly random and partly from the network. However, starting from around 230,000 steps, it can be seen that the `reward_sum` is increasing, which is an indication that the network is well trained and is gaining more positive rewards, or in other words, less negative rewards, and reaching the goal more, more training steps afterwards are needed so that it will reach the goal optimally without hovering around. Moreover, graph 2a was after a slight change in the network, after removing one of the Convolution2D layers, and making the Dense layer of output 512 instead of 32, it can be seen that the loss decreases also but not with the same rate as graph 1a. However, the `reward_sum` in 2b doesn't increase, it keeps decreasing which means the agent takes more negative values. Moreover, many networks were tried and compared by this method, by comparing the `reward_sum` and it was seen that the current network is the best of the tried ones.