Project report

Learning algorithm

The learning algorithm used is Deep Q Learning as described in original paper with few modifications

First modification is using **Double Deep Q learning**, algorithm presented in following paper: https://arxiv.org/abs/1509.06461

General idea of this improvement is connected with calculation of following part:

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-) .$$

In original Deep Q learning algorithm both max action number and max action value was taken from **target network** (network that was updated with delay of C steps/frames) with weights $\boldsymbol{\theta}_t^-$

In double Q learning calculating max action number and max action value is separated into two steps:

- max action number is calculated using **local network** (network we update every time) with weights $\boldsymbol{\theta}_t$
- action value is calculated with target network.

This separation allows us to minimize effect of taken very big number by mistake from local network (agent get som mistaken reward etc.). Instead we are using value of given state/action form target network (delayed and more stable one), what allows us to take more optimized value, usually smaller than in local network that is constantly changing.

Given this we have following equation implemented in agent:

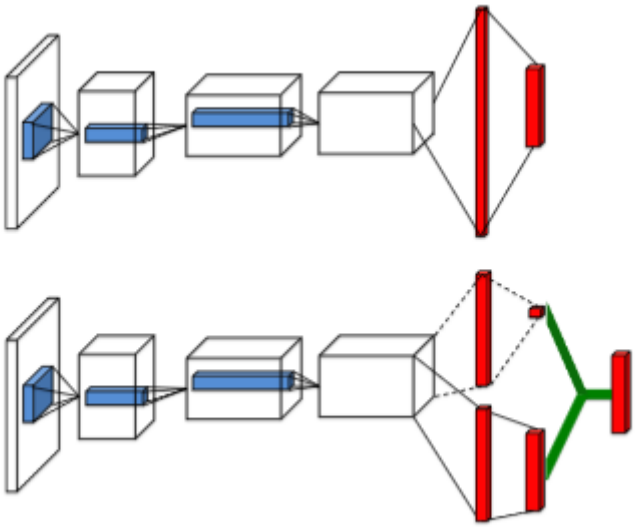
$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax} Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t')$$

Secon improvement is usage of **Dueling Deep Q learning alghoritm** presented in https://arxiv.org/abs/1511.06581. Main idea of this improvement is that action taken is not so important in all states, just in few one(should we go right to pick yellow banana or should we go left to pick blue one?). Basing on this assumption there is slight modification made in neural network used in given algorithm. After last activation function we have separation into two layers :

- layer that calculates action-state values (**advantage values**)
- layer that calculate value-state value (**state value**)

after separation both values are summed to one layer that gives us action-state value(action values).

Below image is taken from mentioned paper and presents case of CNN but the same idea is in no-CNN networks(below one is for dueling DQlearning):



Third improvement is usage of **prioritized experience Replay Buffer** taken from https://arxiv.org/abs/1511.05952. Instead of double queue **we are using binary tree**, sum tree to be specific, where each node is sum of nodes that are below. In this class there are two buffers that are holding data and priority values for given data (both buffers are correlated). Usage of this structure gives us search time O(logN) and insertion time o O(1). Main algorithm of buffer is that when we take each batch we're splitting our buffer into equal parts and take one sample from each part, so we can have representative of each priority segment available when we pass this information into neural network. Before using of this technique there is need to pre-populate buffer with some random values, what is done. This was the hardest one to implement due to complexity of new data structure and probability calculation etc.

Neural Network is simple one (no CNN/RNN …). This type of network was chosen because we have vector of data, not some image with few layers , so there is no need to use CNN. Secondly with normal NN there is much less parameters to calculate during back propagation, then when CNN is used, and as I mentioned before we have vector input.

Because input is size 37, there is need to use bigger first hidden layer, so I use 128 neurons there. Second layer have 64 neurons, what was chosen with trials and errors(128 neurons had worse results than this one).

All neurons are initialized using **Xavier initialization for weights and ZERO initialization for biases**. This is used to make sure that there is no weights initialized with zero value, what could decrease network performance.

As for activation function **RELU** was used, because it's most common one and from my experience it fits image recognition and like most of cases. I've tried with ELU function but I get worse performance.

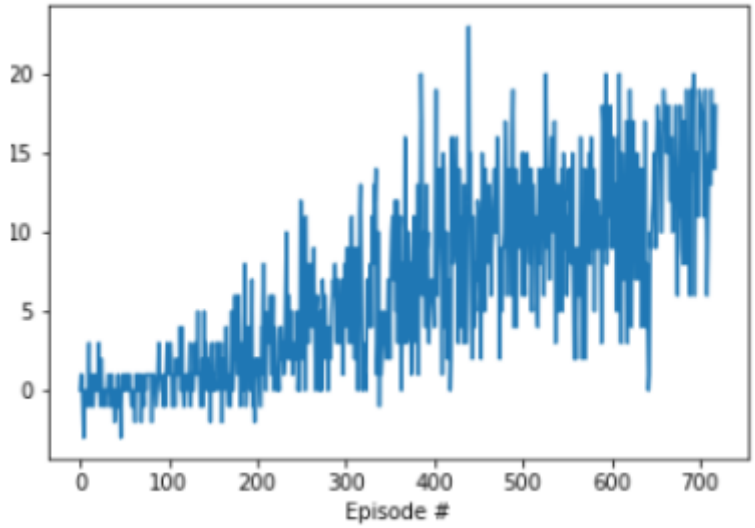
Learning rate is set to 0.001. I've experimented with 0.00025 and results were not much worse (network needed about 20 episodes more to finish task with 0.00025).

Adam parameters were left default , there was no weight decay used or learning decrease methods.

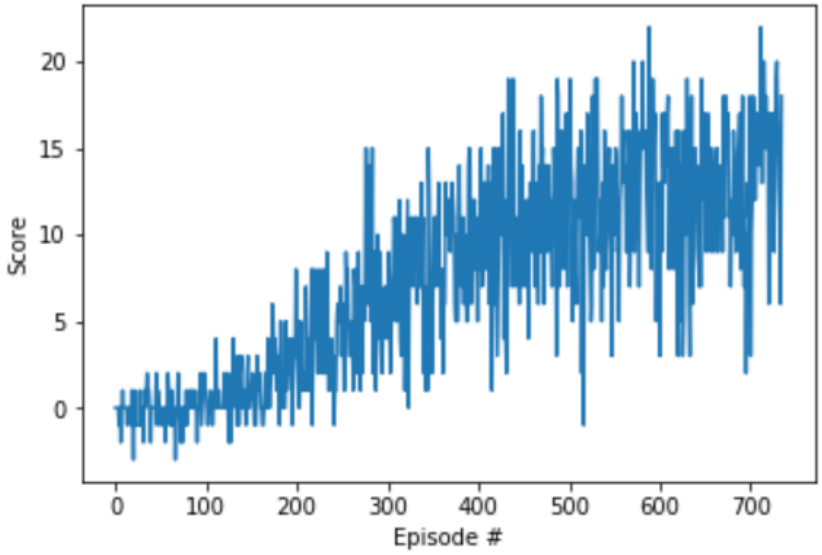
Epsilon values: start, step, stop : 1.0 ; 0.995 ; 0.01

Results

Results for learning rate 0.001 are in file **learning_rate_0_001.pdf** and plot is below:



Results for learning rate 0.00025 are in file **learning_rate_0_00025.pdf** and plot is below:



Improvements

- usage of learning rate decay
- usage of batch normalization after each /first hidden layer (need more hyperparameters tuning)
- usage of Rainbow algorithms
- usage of noisy Q learning
- usage of dropout (with current hyperparameters network is performing much worse. It have to be connected with one more hidden layer plus some hyperparameters tuning)