Algorithm: Double DQN

The Deep Q Network

The deep Q-network (DQN), combines reinforcement learning with deep neural networks, and can be used at circumstance that requires continuous state space. The particular algorithm it use is Sarsamax, aka Q-Learning, a subclass method of Temporal-Difference learning.

The widely used Q-Learning is an efficient and proved to be successful algorithm. But the original form of Q-Learning with Q-table can only be applied to discrete state and action space.When dealing with the continuous space, such as the environment in this Navigation project, or video games, we have to find a way to generalize reinforcement learning algorithms to this kind of problems.

There are several approaches of doing that, and the one DQN use is function approximation. To be more precisely, it use deep neural networks to be a nonlinear function approximation. The neural network can serve as a parameterized approximated function $\mathrm{Q}(\mathrm{s},\mathrm{a},\theta)$ to estimate the true value function $q_\pi(S_t, A_t)$. The loss function can be established with mean squared error (MSE) between the true value function and estimated q function produced by the network:

$$\mathrm{J}(\theta) = E_\pi\ ((q_\pi(s,a) - q'(s,a,\theta)^2)$$

in which $q'(s,a,\theta)$ is the estimated value function while $\mathrm{q}_\pi(\mathrm{s},\mathrm{a})$ is the true value function ,and θ is the weights(parameters) of the neural network. Like the common neural network, we can use Gradient Descent to train the network the update θ until it converges.

The network architecture I use in this project is multiple layers perceptions (MLP), I didn't use convolutional neural net because the input of my model will be just real numbers (the 37-dimension state data of the Navigation environment), and thus I don't have to deal with pixel-leveled data. The MLP network I build has 3 full-connection layers (including the output layer), and the two hidden layers both have 64 nodes. During training, I use the Adam optimizer with a learning rate of 0.001. At each time step, the MLP neural network can produce an output of 4-dimensional vector at given state, corresponding with the estimated state-action values Q(s,a), and the agent can then use the Epsilon-Greedy policy to choose an action.

And there are two crucial design of the DQN algorithm in the original paper [1], which I used for my implementation.

Experience Replay

In the original method of Q-Learning, the Q-table is updated every time the agent takes a new action. But when comes to the neural network, this kind of strategy is not

suitable anymore. Instead, we store experiences including state transitions, rewards and actions inside an internal memory of the agent, and sample them into mini-batches at regular intervals to update the network. Because the sampling is random, using experience replay can reduces correlation between experiences in updating DNN. It can also increases learning speed and reuses past transitions to avoid catastrophic forgetting.

Fixed Q target

When updating the Q network, we need to calculate the TD error, the difference between true value function and the estimated one. The problem here is we don't know the true value function so we have to using some approximation to replace it. The update formulation of network parameters using gradient descent is as follows:

$$\Delta w = \alpha\left[\left(R + \gamma max_a Q(s', a, w)\right) - Q(s, a, w)\right]\nabla_w Q(s, a, w)$$

In which $R + \gamma max_a Q(s', a, w)$ is the approximation for true Q values, it's called the Q target. But if we update a guess with a guess, it can potentially lead to harmful correlations, and maybe some severe oscillation, which will eventually cause an unstable learning progress.

The solution is to use fixed Q-targets which are introduced by DeepMind. To be specific , we can use a separate network with a fixed parameter (let's call it w-) for estimating the TD target, and using a soft update strategy slowly and gradually update target network.

Double DQN

For the original DQN algorithm, there is a problem that can't be neglect. Which is the overestimating problem when estimate the q values. This systematic overestimation introduces a maximization bias in learning [2]. To address this, the double DQN [3] [4] architecture is introduced, which using two separate Q-value estimators to update each other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates.

Work and Results

In my work, I use the double DQN learning algorithm with a 3-layer MLP neural network. My implementation combines the double DQN with fixed Q targets, there are two neural networks with the same structure, local net and target net. Local net is responsible of learning, at fixed period of time (every 4 time step), it samples mini-batch (size of 64) of experiences from the memory (buffer size: 105) and using them for training and update its parameters. While the target net provides a relative Q

target value for the loss function of local net, it doesn't get trained, only soft updated (the soft update parameter TAU is set to be 0.001).

What needed to be illustrated here is how I calculate the Q-target for loss function. At each state in the sampled data, I first get the action for the maximum estimated Q values (i.e. argmax) from local network, and using this action to select corresponding q values from target net. Then calculate the target Q value with the equation mentioned above, the value for discount factor γ is 0.99. In addition, the agent use epsilon-greedy policy to choose an action at every time step, in order to address the Exploration vs. Exploitation dilemma, I make the supper parameter ε to be changing over time. It has a high value (initial value is 1.0) at the early stages, which given the agent enough freedom to explore over new state. It decreases with time (with a decay rate of 0.995), when the estimated Q function start to converge, ε can become very small (the lower limit is 0.01), indicate that the agent can fully exploit the Q values it has learned.

In this project, I build a double DQN agent using the architecture and technique I talked above and training it at the Navigation environment. After training, the agent can get +13 accumulated reward scores over 100 episodes. I trained it for over 800 episodes and the agent solved the environment after 700 episodes, took about 17 minutes. I plot the rewards per episode for both training and testing, as shown below.

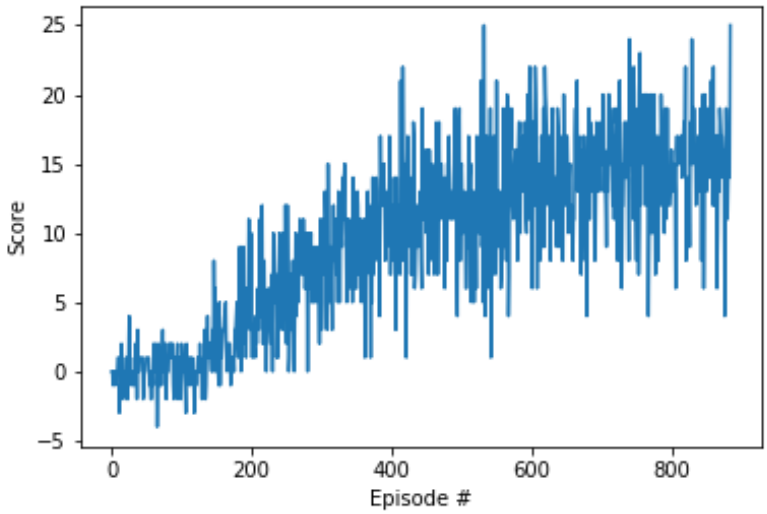

Figure 1. Average reward scores for training

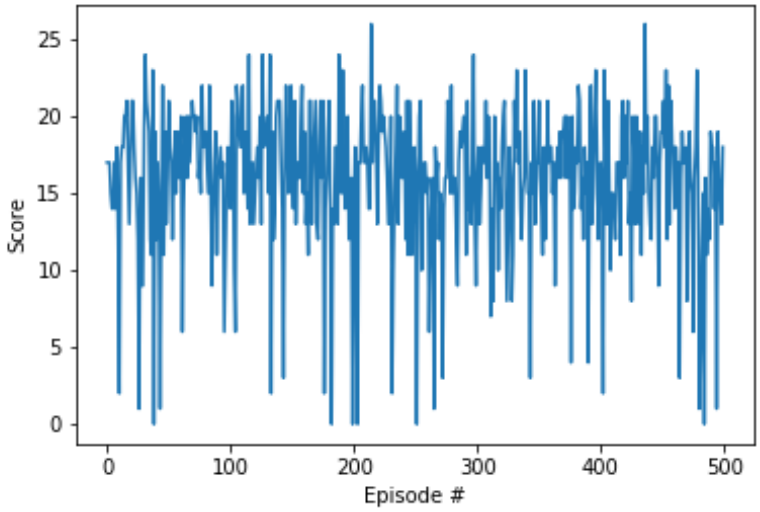

Figure 2. Average reward scores for testing

Plans for Future Work

For this work, I use the DQN algorithm, and applied double DQN structure in order to get better performance. But there are other improvements for the original DQN method, such as Prioritized Experience Replay, Dueling DQN, Distributional DQN, Noisy DQN and so on. I intend to implement several of these methods, especially Prioritized Experience Replay and Dueling DQN, and comparing them with my current work.

References

[1] Human-level control through deep reinforcement Learning (Volodymyr Mnih et al., 2015)
[2] “Issues in Using Function Approximation for Reinforcement Learning” (Thrun and Schwartz, 1993)
[3] Deep Reinforcement Learning with Double Q-learning (Hasselt et al., 2015)
[4] Double Q-learning (Hasselt, 2010)