Deep Reinforcement Learning – Project 1: Navigation

NIKHIL TIWARI

The objective of the project is to design an agent to navigate (and collect bananas!) in a large, square world. 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 the agent is to collect as many yellow bananas as possible while avoiding blue bananas. The agent's observation space is 37 dimensional and the agent's action space is 4 dimensional (forward, backward, turn left, and turn right). The task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

First, let's recall our understanding and knowledge about reinforcement learning and its major algorithms.

Reinforcement Learning: Reinforcement learning[1] is training an agent to perform a particular behavior, but the feedback of whether the agent's behavior is correct or not is received after many time steps — i.e. there is delayed feedback (unlike in supervised learning where there is immediate feedback). Reinforcement learning is basically a method of sequential decision making where the environment model is unknown and the agent has to learn the optimal behavior (to get maximum total rewards) in that environment. If the environment's model is known, then it is known as a planning problem. There are many different planning algorithms such as Dynamic Programming, Uniform Cost Search, A*, etc. Strictly speaking, regardless of whether the environment's model is known or not, it is all under the domain of reinforcement learning.

Markov Decision Process (MDP)

The basic framework for a reinforcement learning problem is defined using a Markov Decision Process (MDP). In an MDP, an agent interacts with the environment by taking actions, and in return, the agent gets rewards and the next state information from the environment. And the process continues. The goal of the agent is to maximize its total rewards. More specifically, the goal is to find a policy that tells what actions to take from each state that will result in the maximum amount of total rewards. It is called Markov because the state incorporates all the necessary information about the past (i.e. history of all the visited states) such that the future is independent of the past given the present state.There are two commonly used value-based RL algorithms to solve MDPs: Monte-Carlo and Temporal Difference learning methods.

Monte-Carlo Learning: Intuitively, Monte-Carlo (MC) learning works as follows: we start off with a random policy from a starting state s and taking action a and then follow the policy pi until termination, i.e. reaching the terminal state. The q-value of state s and action a is then updated using the following equation:

qpi(s,a) = qpi(s,a) + alpha(Gt-qpi(s,a)) ….. EQ. 1, where Gt is the total rewards obtained for the episode and alpha is the learning rate.

This same process of updating the q-value is done for all the states encountered in the episode. Then the policy is updated such that for each state (that has been

previously visited), we pick the action with the highest q-value. But given the q-values for the policy pi are only estimates and not the true q-values, this will be a greedy policy that can be suboptimal. To address it, we use an epsilon-greedy policy where with a probability epsilon (eps), we randomly pick an action from state s and with probability 1-eps, we follow the greedy policy. This helps balance exploration-exploitation tradeoff that is so important in reinforcement learning. The process of iteratively updating the eps-greedy policy and the q-values of all the visited states eventually leads to q-value convergence to the optimal policy's q-values. One downside of the MC method is that it only works for episodic tasks, i.e. tasks that terminate. Moreover, it also takes a longer time to learn than other algorithms.

Temporal-Difference Learning: Temporal-Difference (TD) learning is another type of reinforcement learning algorithm that combines the best of Monte-Carlo (MC) learning and Dynamic Programming (DP). Like DP, we bootstrap by updating the q-values after one step instead of waiting until the episode terminates. This allows for the algorithm to converge faster, have less variance, and be computationally efficient. Moreover, like MC, for each state, TD only takes/samples a single action. Which is unlike DP, where we do full action sweep at each step. This further makes it computationally efficient. There are a few different variants of the TD learning algorithm: sarsa, Q-learning, and expected sarsa. And they all differ in how the TD target in their respective update equations is calculated.The q-value update equation for sarsa is given by:

qpi(s,a) = qpi(s,a) + alpha(r(s,a)+qpi(s',a') — qpi(s,a)) ….. EQ. 2, where r(s,a)+qpi(s',a') is the TD target, a' is the action taken according to policy pi from state s', and the other parameters are as defined previously.

And the update equation for q-learning is:

qpi(s,a) = qpi(s,a) + alpha(r(s,a)+max_a{qpi(s',a)} — qpi(s,a)) ….. EQ. 3, where r(s,a)+max_a{qpi(s',a)} is the TD target and it is the maximum q-value over all the actions at state s'.

Once the q-values are updated, the policy is in turn updated in the same way as with the MC method above, i.e. using eps-greedy policy for all the visited states.

Sarsa is an online learning algorithm because for the TD target (i.e. r(s,a)+qpi(s',a')), the action a' is chosen based upon the policy we are trying to learn/improve, i.e. eps-greedy policy pi. Whereas Q-learning is an off-policy algorithm because the TD target (i.e. r(s,a)+max_a{qpi(s',a)}) is chosen based upon the greedy policy, which is not the same as the eps-greedy policy (pi) we are trying to learn. The above equations, however only work for tabular world cases where the state space is finite. In continuous environments, discretizing the statespace can quickly run into Bellman's curse of dimensionality. To address it, we instead use a function approximator to model the q-values.

Using a function approximator, we update the weights of the function approximator and the resulting update equation for Q-learning becomes:

w = w + alpha(r(s,a)+max_a{qpi(s',a,w)} -qpi(s,a,w))*grad_w(qpi(s,a,w)) ….. EQ. 4, where grad_w(qpi(s,a,w)) is the gradient of qpi(s,a,w) with respect to weights w and the other parameters are as defined previously.

The policy update is the same as in the tabular case where for each visited state, with probability eps we select random action and with probability 1-eps we select an action with the maximum q-value (i.e. greedy policy). For linear function approximators, this approach works rather well in practice. That is the learning algorithm doesn't oscillate and instead converges to the optimal policy.

DQN: For nonlinear function approximators like neural networks, the above approach can run into instabilities. To help improve convergence, two modifications can be made and the resulting algorithm is known as Deep Q-Network (DQN) [2].

1. *Fixed Q Targets*: In the above Q-learning algorithm using a function approximator, the TD target is also dependent on the network parameter w that we are trying to learn/update, and this can lead to instabilities. To address it, a separate network with identical architecture but different weights is used. And the weights

of this separate target network are updated every 100 or so steps to be equal to the local network that is continuously being updated.

2. *Experience Replay:* Updating the weights as new states are being visited is fraught with problems. One is that we don't make use of past experiences. An experience is only used once and then discarded. An even worse problem is that there is an inherent correlation in the states being visited that needs to be broken; otherwise, the agent will not learn well. Both of these issues are addressed using experience replay where we have a memory buffer where all the experience tuples (i.e. state, action, reward, and next state) are stored. And to break correlation, at each learning step, we randomly sample experiences from this buffer. This also helps us learn from the same experience multiple times, which is especially useful when encountering rare experiences.

Result:

The agent is trained during 1000 episodes, and we compute the cumulative reward per episode. We consider the environment to be solved when the average reward over the last 100 episodes is at least +13, for that we use the queue (FIFO: First In, First Out) with length 100 using a deque in the collections library of Python to progressively compute the average over the last 100 episodes. When the training is done, we can inspect the plot of rewards per episode:

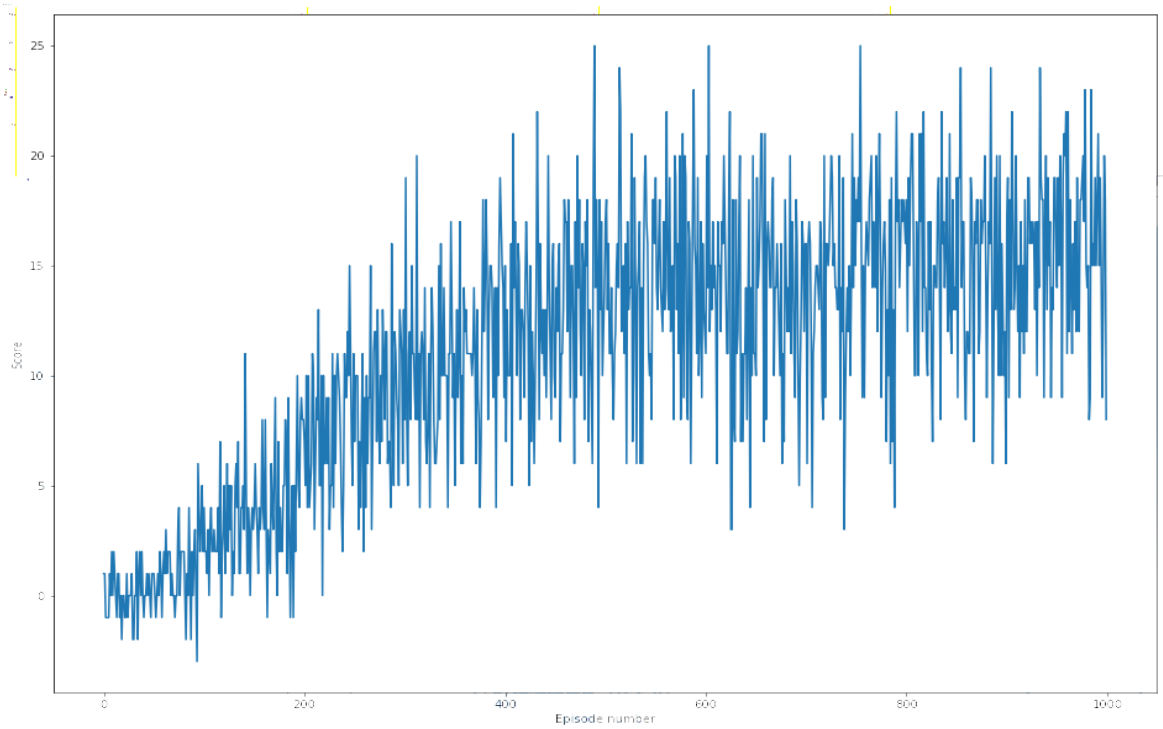

Figure 7: Plot of scores per episode

The environment is solved in 392 episodes.

The help in code is taken from the code provided in the DQN module of the Nanodegree.

Future Ideas:

The Double DQN algorithm which is an improvement to DQN would increase the model performance and agent would perform better. The idea could be very interesting to train the agent directly using raw pixels that the agent "sees." The task could be challenging but would be fun to try using Andrej Karpathy's famous blog "Deep Reinforcement Learning: Pong from Pixels"[3].

References:

[1] Sutton and Barto RL Book: http://incompleteideas.net/book/the-book.html

[2] Deep Mind DQN Paper:

https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[3] Andrej Karpathy's famous blog "Deep Reinforcement

Learning: Pong from Pixels" http://karpathy.github.io/2016/05/31/rl/