Project Navigation Report

Udacity Deep Reinforcement Learning Nanodegree.

15th October 2018. Simon Birrell.

Introduction

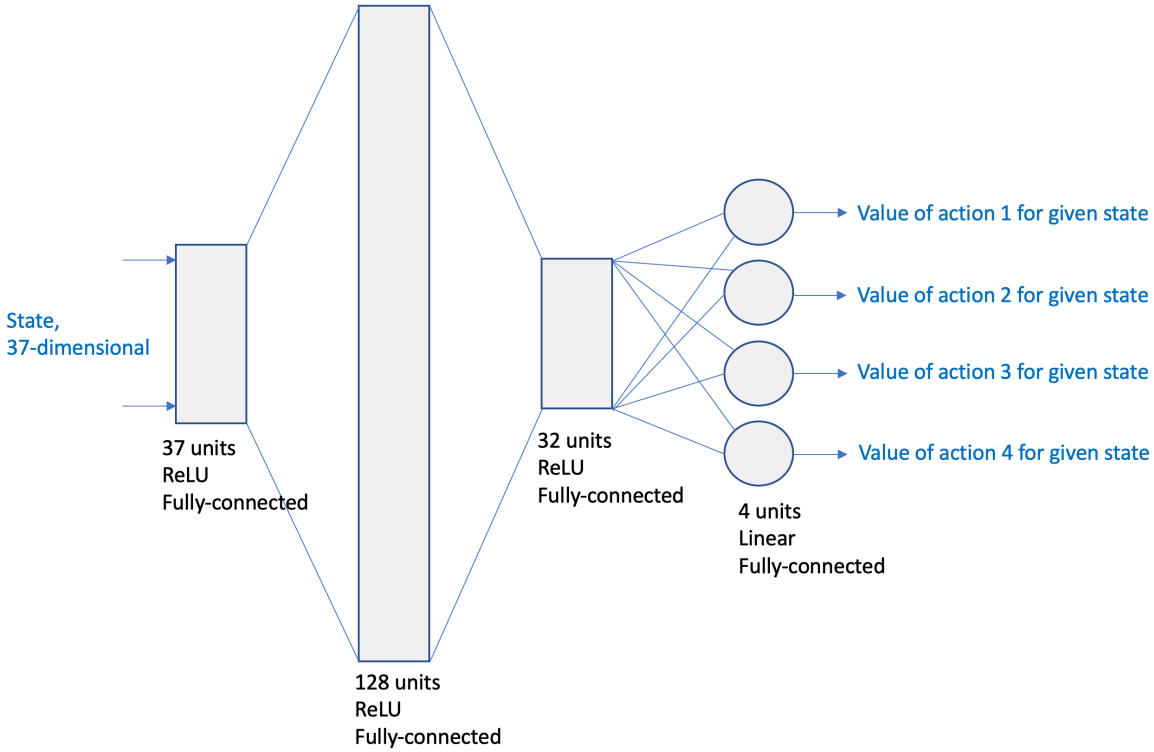
This project aimed to solve a Unity reinforcement learning test environment where the agent has to navigate around a world, eating yellow bananas and avoiding blue ones. The code uses a Deep Q Network that arrives at a solution in 135-159 episodes.

The Code

The Python 3.6 code is packaged within a Jupyter notebook. Full instructions are in the repository at https://github.com/SimonBirrell/dqn-navigation-project/

The Agent

The agent is controlled by a neural network that accepts as input the 37-dimensional continuous state provided by the Unity environment and outputs the predicted "value" of performing each of the 4 possible actions for that state. The action with the highest value is the one that the policy instantiated by the network believes should be chosen to maximize the overall reward in the episode. When running the agent in test mode the recommended action can be executed greedily.



The network comprises four fully-connected linear layers, the first three having RELU activation functions and the final one being purely linear.

The weights of the network are available in the file checkpoint.pth.

Training the Agent

The architecture of the network is extremely simple; the complexity lies in the training process that leads to the weights that constitute a well-performing policy.

Reinforcement Learning algorithms attempt to find an optimal policy π_* that maximizes reward for an agent over an entire episode, not just over the immediate step. One way of doing this is to wait for each episode to complete before applying learning updates as the algorithm can at that point be certain of the overall reward and can project back the value of each action taken at each step in the episode. This value is known as the Q value of an action a taken at state s. Another, faster way is to make a best guess of this value during the episode itself and use the guess to apply learning updates. This is known as temporal difference (TD) reinforcement learning, and deep Q-networks are one example of TD algorithms.

The policy will map states to state-action values using the neural network described above. To train a neural network we typically need target values, or "labels", and a loss function that is used to nudge the outputs towards these targets. Deep Q-Networks have many variants, but the version we use has a loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\backsim U(D)}\left[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2\right]$$

This looks more complicated than it really is. The current Q value of action a at state s according to the neural network is $Q(s, a; \theta_i)$, where θ_i are the network weights. r is the immediate reward obtained by taking action a at state s. If we define our target T as

$$T = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$$

then the loss function reduces to

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\backsim U(D)}[(T - Q(s, a; \theta_i))^2]$$

which is a standard mean squared error loss function and can be used to optimize the weights using back propagation. The expectation is over a set of tuples (s, a, r, s') drawn from experience, where s' is the subsequent state following action a at state s. These experience tuples are stored in a dataset buffer D and are drawn at random with equal probability, hence the notation $\backsim U(D)$. By drawing randomly from this buffer, the algorithm avoids correlation and feedback effects that occur when learning is immediately performed from the most recent experience (s, a, r, s').

How are the targets derived? Recall that

$$T = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$$

The target is simply the immediate reward from taking action a at state s plus the discounted value of taking the "best" action a' at the subsequent state s'. Our guess for the present state $Q(s, a; \theta_i)$ is modified by our guess for the immediate future. This chasing of one's own tale will gradually improve the policy π instantiated by the neural network.

The final detail to mention is that the target uses an alternative set of weights θ_i^- not θ_i, in other words a separate copy of the neural network. This separate network is updated from the primary network with a delay. This makes the target more stable and less coupled to the immediate changes in the weights θ_i, removing another source of correlations and potentially disruptive feedback from the algorithm. These updates from θ_i to θ_i^- can either be taken

every few steps, or, as in the case in the current code be smoothed in on each step using a weight factor.

All of this translates into the following code:

```
1.          states, actions, rewards, next_states, dones = experiences
2.
3.          self.qnetwork_local.train()
4.          self.qnetwork_target.eval()
5.
6.          # forward pass to get outputs
7.          n = len(states)
8.          # Tensor batch_size x action_size
9.          target_outputs = self.qnetwork_target(next_states)
10.         # Tensor batch_size x 1
11.         targets = cast(
12.             [(rewards[i] + (1.0-
    dones[i])*gamma*torch.max(target_outputs[i]),) for i in range(0,n)]
13.         )
14.
15.         # Tensor batch_size x action_size
16.         local_outputs = self.qnetwork_local(states)
17.
18.         # Get expected Q values from local model
19.         q_values_for_actions_taken = self.qnetwork_local(states).gather(1, actio
    ns)
20.
21.         # Calculate loss using MSE
22.         loss = F.mse_loss(q_values_for_actions_taken, targets)
```

The tensor of `experiences` has been drawn from the replay buffer. The target outputs are defined in lines 10 through 12. The multiplier `(1.0-dones[i])` simply zeroes out the discounted value of the net state at the end of each episode. The Q value estimate based on the current state is in line 19 and the mean squared error loss function is defined in line 22. This is used to optimize the network. Note that the targets use the secondary network `qnetwork_target` not the main one `qnetwork_local`. These are the θ_i^- and θ_i mentioned above. The secondary network is updated each step with

```
self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

where each weight is updated to be

$$\theta_i^- \longleftarrow \tau\theta_i + (1-\tau)\theta_i^-$$

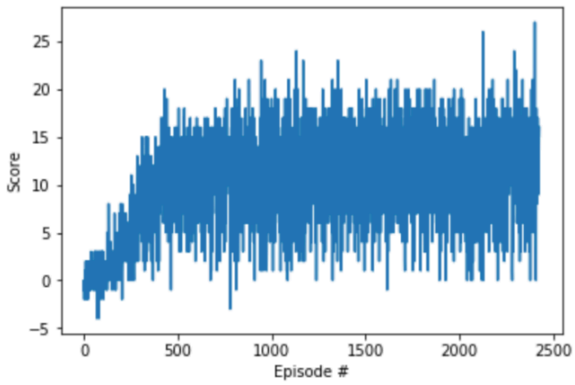
with τ being a small number close to zero.

Finally, when the agent is run in training mode, the action executed should be chosen with an epsilon-greedy algorithm, not greedily. This encourages the agent to explore parts of the solution space that it wouldn't otherwise encounter. As usual, epsilon ε should decay from near 1.0 to near 0.0, in this case 0.01.

Optimization and Hyperparameters

The code was taken from a previous coding assignment, modified to work with the Unity environment and worked almost immediately, solving the environment in 2327 episodes. The process of optimization then began, focusing first on the architecture of the neural network and then on the hyperparameters.

The original architecture had five layers, not four (37, 256, 512, 1024, 4).



I then tried progressively reducing the size of the penultimate layer to see how the time to finding a solution was affected:

Architecture: Neurons per layer	Episodes to solve
37, 256, 512, 1024, 4	2327
37, 256, 512, 512, 4	1343
37, 256, 512, 256, 4	847
37, 256, 512, 128, 4	637
37, 256, 512, 64, 4	631
37, 256, 512, 32, 4	**593 (best)**
37, 256, 512, 16, 4	981

Reducing the size of layer four below 32 stopped improving performance and worsened it. My hypothesis is that the network as it stood before had more learning capacity than was needed and so took longer to train.

Following this logic, I net tried eliminating previous layers:

Architecture: Neurons per layer	Episodes to solve
37, 256, 512, 32, 4	593
37, 256, 32, 4	**350 (best)**
37, 256, 4	452
37, 512, 4	-

Eliminating the third layer improved performance, but eliminating the second was a step too far. I settled for a four layer architecture and turned to the hyperparameters.

The biggest effect came from modifying the rate of decay of epsilon ε. In the code, this is controlled by the hyperparameter eps_decay.

eps_decay on (37, 256, 32, 4)	Episodes to solve
0.995	350
0.985	313
0.999	-
0.975	**234 (best)**
0.965	281
0.970	292
0.980	272

The best value was found to be 0.975. My hypothesis is that reducing the decay rate from 0.995 to 0.975 encouraged more exploration early on in the training.

In a similar way I tried modifying the UPDATE_EVERY parameter but only found the performance to degrade.

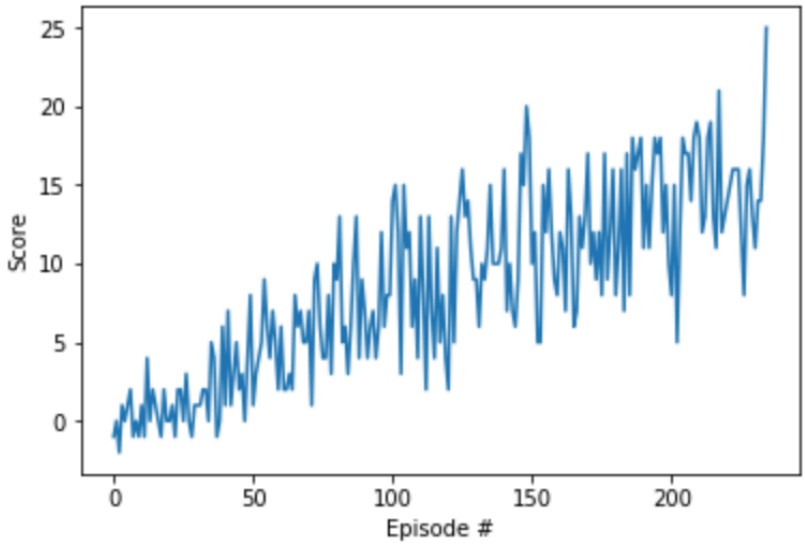
I then returned to modifying the architecture, reducing the units in layer 2:

Architecture (eps_decay = 0.975)	Episodes to solve
37, 256, 32, 4	234
37, 128, 32, 4	**135 (best)**
37, 64, 32, 4	224
37, 96, 32, 4	219
37, 112, 32, 4	235
37, 196, 32, 4	305

I also tried modifying the BATCH_SIZE and GAMMA parameters in the same way, but only degraded the performance.

The (37, 128, 32, 4) architecture with eps_decay=0.975 proved to be the best combination I found:

```
Episode 100     Average Score: 3.68     Epsilon: 0.08
Episode 200     Average Score: 11.24    Epsilon: 0.01
Episode 235     Average Score: 13.10    Epsilon: 0.01
Environment solved in 135 episodes!     Average Score: 13.10
```



It should be noted that this was the best run. Subsequent repetitions gave the following results:

Run (37, 128, 32, 4), eps_decay=0.975	Episodes to solve
1	135
2	159
3	192
4	203
5	159
	Mean = 169.6 var = 607.84

Ideas for Future Work

There are probably further optimizations to be eked out of the architecture and hyperparameters. While doing the "manual grid search" described above I used only the first run of each setup to determine the number of episodes required. As seen above, there is some variance to the numbers, so a "best of 3" at each combination might be best.

Double DQN. This is a technique that avoids the network choosing over-optimistic Q values early on. In this case the optimal action a' for the net state s' is determined using the standard weights θ_i but then the Q value is taken from this action evaluated against the alternative set of weights θ_i^-. This would be simple to add to the above code.

Prioritized Experience Replay. Instead of drawing the experiences uniformly at random from the experience buffer D, it would be useful to prioritize those experiences which result in the greatest learning for the network. To do this, a weighting factor can be added to each experience tuple (s, a, r, s') stored that increases its likelihood of being chosen. One way of calculating this weighting factor is to use the actual difference between target and estimation (the "temporal difference") for the given tuple:

$$T - Q(s, a; \theta_i)$$

Experiences with a high temporal difference will be picked over those with a low one. This can be further tweaked to prevent experiences with a temporal difference of zero never getting picked, at the expense of an additional hyperparamter.

Both of these techniques are reasonably simple to add to the code, but would then require resting variations in architecture and hyperparameters. For reasons of time, this has not been attempted.