Report – P1 Navigation @Unity ML Agent Toolkit

Krishna Sankar
10/2/18

1. Project & General Notes

This is a very interesting problem – to train an agent to navigate (and collect bananas!) in a large, square world. The environment is the Banana Collector environment using the Unity ML-Agents toolkit.

The environment has four actions viz Move Forward, Move Backward, Turn Left and Turn Right.

The agent will encounter two types of bananas – blue and yellow. A reward of +1 for collecting Yellow Bananas and -1 for collecting blue bananas. The goal, of course, is to collect as much yellow bananas and avoiding blue bananas.

The environment returns a state space of 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction.

The environment terminates after 300 steps.

In order to solve the environment, an agent has to get a minimum score of 13 averaged over a window of 100 consecutive episodes.

2. Algorithm

This is a classical Markov Discrete Process and can be solved using the Deep Q Network algorithm.

2.1. Reinforcement Learning Model

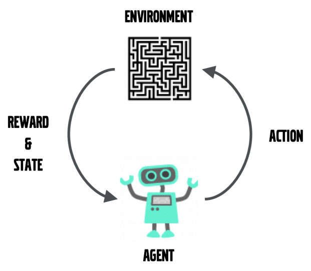
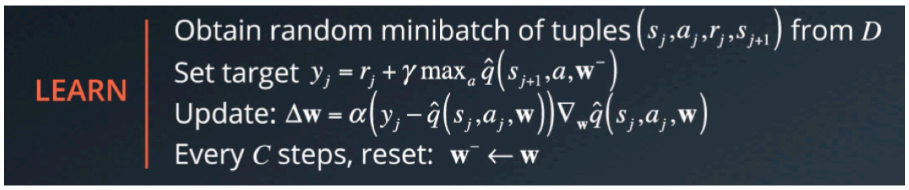


Figure 1

In the classical Reinforcement Learning scenario, an agent interacts with an environment by taking actions. The environment returns a reward as well as an observation snapshot of the state space. The state space observation encapsulates all the cumulative information of all the previous actions. Hence the current state space observation is enough for the agent to take next action. The agent's goal is to select actions that maximizes cumulative future reward. This fits very well with the current problem.

2.2. Model Architecture

The agent has 3 main components viz:

1. The DQN Orchestrator which interacts with the environment by taking actions and then unpacking the returned package to rewards, state space et al.
2. It also has to do housekeeping like tracking scores, store high performant models and check when the problem is solved
3. The 3rd component is the most interesting one, which gives the agent the capability to select right actions that maximize long term rewards
    a. The fundamental unit is the Q value, which is the total expected reward, discounted for the future actions
    b. The Q value is represented as Q(s,a) i.e. q values for all states for a state. In our case the dimensionality of the state is 37 and 4 actions are possible.
    c. One method is to store all the Q-values in a dictionary with the state and action as key.
    d. Another important concept is the policy i.e. the strategy to choose an action at a given state. We follow the ε-greedy policy i.e. for a probability of ε, we choose a random action and choose the action that maximizes the total reward otherwise.
    e. So, if we have all the Q values in a table, we can search the actions for every state and then choose the action based on the ε-greedy policy. But as the state space becomes larger or even continuous, this table becomes unmanageable and resource consuming even impossible.
    f. Function Approximator – Instead of the Q Table, we can train a function approximator which then can give us the q-value for a given state-action pair.
    g. The function approximator in the DQN algorithm is a neural network, which gives a state returns the q-values for all the actions. This approach is better because, otherwise we need to run the inferencing for each of the action at every state
    h. Now the challenge is how to train the network. There a few points to explore.
        - First is data. Usually the Reinforcement Learning problems involve a sequence of actions and so the Q values are correlated. So then learning run samples from a cache of State-Action-Reward tuple called the experience replay. This gives the network uncorrelated data to learn from. The experience replay buffer is a circular cache
        - Second is the training algorithm – we use the Fully conventional Neural networks and train them using SGD (Stochastic Gradient Descent). The actual implementation uses the Adam optimizer. The training algorithm is succinctly captured in the following figure from Udacity.
        - 

          LEARN

          Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D

          Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$

          Update: $\Delta \mathbf{w} = \alpha\left(y_j - \hat{q}(s_j, a_j, \mathbf{w})\right)\nabla_{\mathbf{w}}\hat{q}(s_j, a_j, \mathbf{w})$

          Every C steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$
        -

- In the above diagram, we have no way of getting the target y_j. Interestingly this value is obtained from a target network that is exactly same as the expected value network, but updated less frequently. In the actual implementation, it is updated using the exponential decaying technique with the decay constant $\boldsymbol{\tau}$ = 0.001.

2.3. Implementation

Once we have the components design, the implementation is relatively straightforward. We have the agent, the replay buffer and the Q network as separate classes and then the DQN as a function that brings everything together to implement the algorithm. I did experimentation with the network architecture and the hyper parameters as discussed in the hyperparameters sub section under Results below.

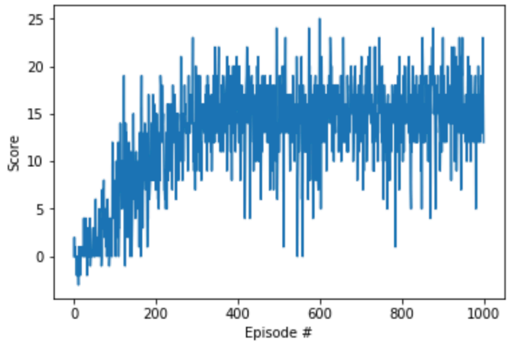
1. After some hyperparameter search, the optimum network is a fully connected network with 16 input units, one hidden layer with 8 units and the output layer with 4 units.
   - *As a quick note, the network would have been a couple of convolutional networks plus one or two fully connected layers, if the input had been images, as in the optional project*
2. ε-greedy policy - The DQN takes three values an initial value, a minimum value and the ε-decay. For each episode, the ε starts from the initial value, decays for each episode until the min value is reached and then keeps that value. The values used are initial = 1.0, min=0.005 and decay=0.85.

3. Results

The environment always terminated after 300 steps. I tried max_t > 1K. I didn't see any complex adaptive temporal behavior.
The environment was declared solved if an agent can get a score of > 13 (averaged over 100 consecutive episodes) An additional challenge was to solve it in less than 1800 episodes. My algorithm was able to solve it by 209 episodes and was able to reach a max average score of 15.42. With a larger (and deeper) network, I was able to get a max average score of 17.05. I have captured the run results in the Jupyter notebook itself – that way all the runs and results are in the same place.

A formal capture of the results as well as run stats line percentile, variance, absolute max score and a score plot is below:

```
Episode : 100   Average Score :  1.78   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 200   Average Score :  8.04   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 300   Average Score : 12.58   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 309   Average Score : 13.01   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Environment solved in 209 episodes!     Average Score: 13.01
Episode : 400   Average Score : 15.05   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 500   Average Score : 14.55   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 600   Average Score : 14.38   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 700   Average Score : 15.32   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 800   Average Score : 14.83   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 900   Average Score : 15.23   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Episode : 1000  Average Score : 15.42   Max_steps : 300 eps : 0.005     Max.Score : 0.000
Elapsed : 0:24:41.908608
2018-10-01 21:49:22.482448
```



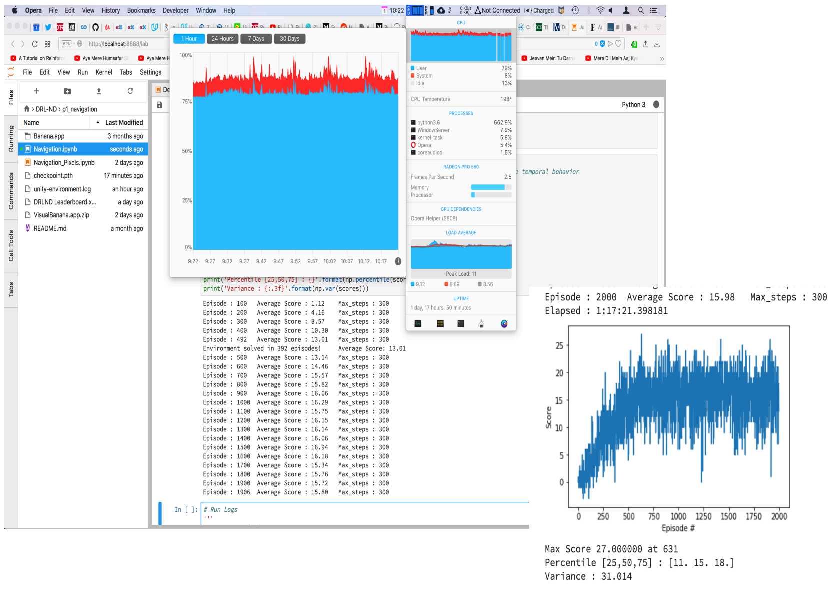
```
QNetwork(
  (fc1): Linear(in_features=37, out_features=16, bias=True)
  (fc2): Linear(in_features=16, out_features=8, bias=True)
  (fc3): Linear(in_features=8, out_features=4, bias=True)
  (fc4): Linear(in_features=4, out_features=4, bias=True)
)
Max Score 25.000000 at 599
Percentile [25,50,75] : [10. 14. 17.]
Variance : 31.916
```

3.1. Network Architecture alternates and Hyperparameters

One of the more interesting part (in addition to writing the code) is the search for optimum architecture and hyperparameters.

I tried a few network architectures – from a minimalist FC32-FC4 to the monstrous atrocity(as it relates to this project – there are huge networks with 100s of layers in the Resnet world) FC64-FC32-FC16-FC8-FC4. I have captured the CPU load as well as various metrics. The metrics are also captured as a summary in the notebook itself for easy reference.

3.1.1. FC64-FC32-FC16-FC8-FC4



```
print('Percentile [25,50,75] : {}'.format(np.percentile(scor
print('Variance : {:.3f}'.format(np.var(scores)))
```

```
Episode : 100   Average Score : 1.12    Max_steps : 300
Episode : 200   Average Score : 4.16    Max_steps : 300
Episode : 300   Average Score : 8.57    Max_steps : 300
Episode : 400   Average Score : 10.30   Max_steps : 300
Episode : 492   Average Score : 13.01   Max_steps : 300
Environment solved in 392 episodes!     Average Score: 13.01
Episode : 500   Average Score : 13.14   Max_steps : 300
Episode : 600   Average Score : 14.46   Max_steps : 300
Episode : 700   Average Score : 15.57   Max_steps : 300
Episode : 800   Average Score : 15.82   Max_steps : 300
Episode : 900   Average Score : 16.06   Max_steps : 300
Episode : 1000  Average Score : 16.29   Max_steps : 300
Episode : 1100  Average Score : 15.75   Max_steps : 300
Episode : 1200  Average Score : 16.15   Max_steps : 300
Episode : 1300  Average Score : 16.14   Max_steps : 300
Episode : 1400  Average Score : 16.06   Max_steps : 300
Episode : 1500  Average Score : 16.94   Max_steps : 300
Episode : 1600  Average Score : 16.18   Max_steps : 300
Episode : 1700  Average Score : 15.34   Max_steps : 300
Episode : 1800  Average Score : 15.76   Max_steps : 300
Episode : 1900  Average Score : 15.72   Max_steps : 300
Episode : 1906  Average Score : 15.80   Max_steps : 300
```

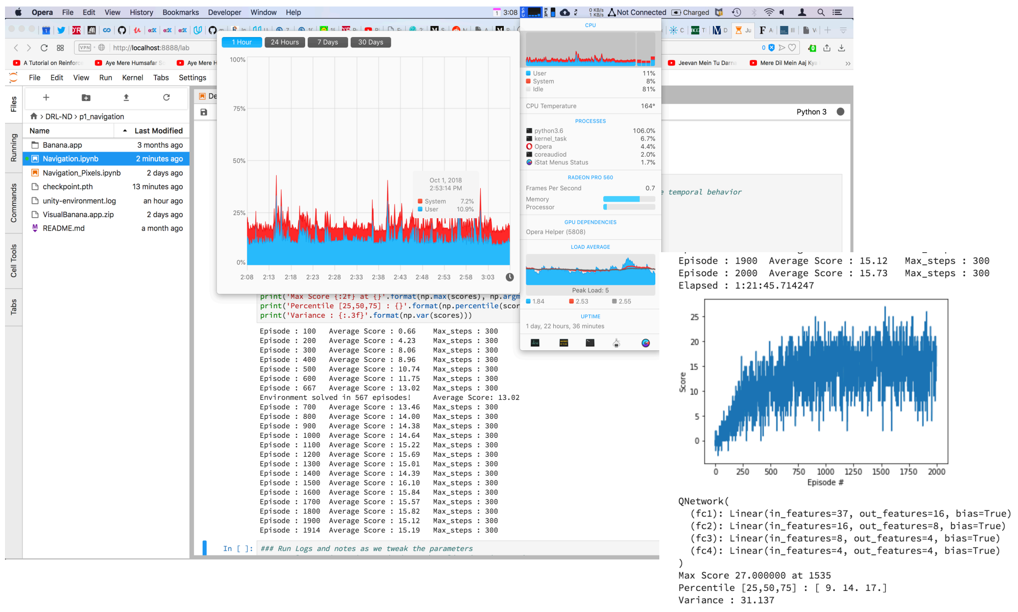
```
In [ ]: # Run Logs
        '''
```

```
Episode : 2000  Average Score : 15.98   Max_steps : 300
Elapsed : 1:17:21.398181
```

```
Max Score 27.000000 at 631
Percentile [25,50,75] : [11. 15. 18.]
Variance : 31.014
```

I have captured the CPU usage and we can see that the CPU usage is very high, actually uses the full 4 threads of the quad core CPU.

3.1.2. FC16-FC8-FC4



```
print('Max Score {:2f} at {}'.format(np.max(scores), np.argm
print('Percentile [25,50,75] : {}'.format(np.percentile(scor
print('Variance : {:.3f}'.format(np.var(scores)))
```

```
Episode : 100   Average Score : 0.66    Max_steps : 300
Episode : 200   Average Score : 4.23    Max_steps : 300
Episode : 300   Average Score : 8.06    Max_steps : 300
Episode : 400   Average Score : 8.96    Max_steps : 300
Episode : 500   Average Score : 10.74   Max_steps : 300
Episode : 600   Average Score : 11.75   Max_steps : 300
Episode : 667   Average Score : 13.02   Max_steps : 300
Environment solved in 567 episodes!     Average Score: 13.02
Episode : 700   Average Score : 13.46   Max_steps : 300
Episode : 800   Average Score : 14.00   Max_steps : 300
Episode : 900   Average Score : 14.38   Max_steps : 300
Episode : 1000  Average Score : 14.64   Max_steps : 300
Episode : 1100  Average Score : 15.22   Max_steps : 300
Episode : 1200  Average Score : 15.69   Max_steps : 300
Episode : 1300  Average Score : 15.01   Max_steps : 300
Episode : 1400  Average Score : 14.39   Max_steps : 300
Episode : 1500  Average Score : 16.10   Max_steps : 300
Episode : 1600  Average Score : 15.84   Max_steps : 300
Episode : 1700  Average Score : 15.57   Max_steps : 300
Episode : 1800  Average Score : 15.82   Max_steps : 300
Episode : 1900  Average Score : 15.12   Max_steps : 300
Episode : 1914  Average Score : 15.19   Max_steps : 300
```

```
In [ ]: ### Run Logs and notes as we tweak the parameters
```

```
Episode : 1900  Average Score : 15.12   Max_steps : 300
Episode : 2000  Average Score : 15.73   Max_steps : 300
Elapsed : 1:21:45.714247
```

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=16, bias=True)
  (fc2): Linear(in_features=16, out_features=8, bias=True)
  (fc3): Linear(in_features=8, out_features=4, bias=True)
  (fc4): Linear(in_features=4, out_features=4, bias=True)
)
Max Score 27.000000 at 1535
Percentile [25,50,75] : [ 9. 14. 17.]
Variance : 31.137
```

This network stands out in it's usage of less CPU resources 11% vs 77% ! And achieves very similar results to the larger deeper network. I like this, based on my autonomous car and drone background. So, this architecture was chosen – Occam's razor and law of parsimony applies here. The best idea is to match impedance of the architecture with the appropriate network architecture – for a state space of 37 and 4 actions, a simpler FC16-FC8-FC4 is fine. But for something like pixel banana, a larger and deeper convolutional network would definitely be appropriate.

Note : I tried a few more network architectures and have captured the CPU usage et al. The summary of all the runs are embedded in the notebook. Each run takes about an hour and 20 minutes.

3.1.3. Hyperparameters

I did an informal search on the important hyperparameters – seeking optimum values within a range. It made sense for this problem as the effect of the hyper parameters were not as varied as in other do mains like object detection, behavior cloning for autonomous driving et al.

1. The number of episodes was initially 2000, as the suggestion was that 1800 is a good number. It took 1 hr and 20 min for each run. As I was able to solve the environment within 500 episodes, 1000 episode would have been OK. So that is what I used for hyperparameter search
2. The discount factor was initially 0.99, from the lunar lander environment in the OpenAI framework. After the network architecture was chosen as FC16-FC8-FC4, tried a discount factor of 0.85. This resulted in solving the environment in 852 episodes (as opposed to 567). So, kept discount factor = 9.99.
3. The ε-greedy parameters initially were initial = 1.0, min=0.01 and decay=0.995. I tried a few values and settled on initial = 1.0, min=0.005 and decay=0.85. This solved the environment in 209 episodes!

4. Ideas for future work

There are lots of vectors one can follow. Am listing the most important one here. Have a few ideas how to improve the code.

1. A more automated, systemic and formal way of exploring hyperparameters, like the caret package in R, is needed in the future. The current exploration is fine for this problem, but for a larger complex problem, an automated hyperparameter search is required
2. This program runs on CPU – I do want to add the device statements and run the notebook on a GPU machine and compare the performance
3. Another idea is to run the code in AWS.
4. Currently, the QNetwork class builds the network by hardcoded static code. It would be interesting if it can build a network dynamically based on a list of unit values.
5. As I mention in the program, the save and load model could be more generic. The recommendation, from stackoverflow et al, was that this might be unstable. But with the release of PyTorch 1.0 today at the PyTorch Developer Conference, this might be streamlined.

4.1. Refactor current notebook code

The code works and I even have a systemic way of just running a saved model without training. And if we train, it will save the best model.

4.2. Pixel Banana

This environment returns a discrete state space of dimension 37. There is another version that returns an image and I plan to work on that environment next. As it is optional, I wanted to submit this first.

4.3. Algorithm improvements

The three essential improvements to DQN are the Double DQN, Dueling DQN and prioritized experience replay. Now that I have a method of capturing various metrics it would be informative to see how the improvements affect them.
Of course, I also plan to see how algorithms employing the policy iteration methods perform.

4.4. NIPS Competition!

Plan to participate in the NIPS 2018 competition Ai driving Olympics [https://www.duckietown.org/research/AI-Driving-olympics]. The Fleet management Task [https://www.duckietown.org/research/ai-driving-olympics/challenges] requires interesting Reinforcement Algorithms – plan to use this and the Policy Evaluations plus other techniques there.

5. One More Thing !!

Getting a handle on Reinforcement learning is not easy. One has to write out a few algorithms to get confidence.

"Confidence is recursive ! You have to come thru a few times to be certain and confident that you can beat it !"