Capstone Project Proposal

David Forester

Machine Learning Engineer Nanodegree

20 May 2017

Domain Background

Reinforcement learning (RL) methods have been successfully employed to learn rules, called policies, to solve tasks such as navigate a maze, play poker, auto-pilot an RC helicopter, and even play video games better than humans[1]. This last feat was accomplished in 2013 by DeepMind who were quickly bought by Google. Their paper "Playing Atari with Deep Reinforcement Learning", describes how a computer can learn to play video games by taking the screen pixels and associated user actions (such as "move left", "move right", "fire laser", etc.) as input and and receiving a reward when the game score increased.

Deep reinforcement learning combines a Markovian decision process with a deep neural network to learn a policy. To better understand Deep RL and particularly the theory of Q-learning and its uses with deep neural networks to form "Deep Q Networks" I read Tambet Matiisen's brilliant Guest Post on the Nervana Systems website[2]. I say brilliant because even I could begin to understand the beauty and the elegance of the method. In looking for a Python implementation of this method that I could study, I found Eder Santana's post "Keras plays catch"[3]. This post describes his solution to the task of learning a policy to catch a falling piece of fruit (represented by a single pixel) with a basket (represented by three adjacent pixels) that can be moved either left or right to get underneath. In his post, he provides a link to the code. It was that post and code which most inspired this proposed project.

1. Playing Atari with Deep Reinforcement Learning, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al., https://arxiv.org/abs/1312.5602
2. https://www.nervanasys.com/demystifying-deep-reinforcement-learning
3. http://edersantana.github.io/articles/keras_rl

Problem Statement

This project's goal is to use a reinforcement learning method called Q-learning to arrive at a policy that an autonomous driving agent can use to cross a simulated busy intersection without incident and in the minimum time possible. That is to say that our agent should always wait until a large enough gap in traffic appears before driving through it, and it

should never miss such an opportunity when presented. Our agent should be neither too timid nor too brave.

Training Environment

The training environment must accurately reflect the problem to be solved as described above in the Problem Statement, after all this is how the training data are produced. Since Q-learning will be employed to learn the correct intersection crossing policy based on traffic, it is certain that a numerical representation of the intersection is required, as well as cars which move in front of the agent waiting to cross. Furthermore, since training and testing trials should be sometimes viewed, the environment state will be represented with a 2-dimensional numerical array. For simplicity, we will let different integer values represent pieces of different objects or environmental features:

0 - empty space (no car may drive here)

1 - road (this is only for visualization)

2 - cars passing in front of the agent

3 - the agent

The environment above has been developed and implemented as a Python class. The environment world is 21 pixels high by 21 pixels wide. All cars, including the agent, are of length 3 pixels. At each time step, each car moves forward one pixel; the agent upwards, and the traffic leftwards. Figure 1 below shows two sequential frames of the agent approaching the intersection.

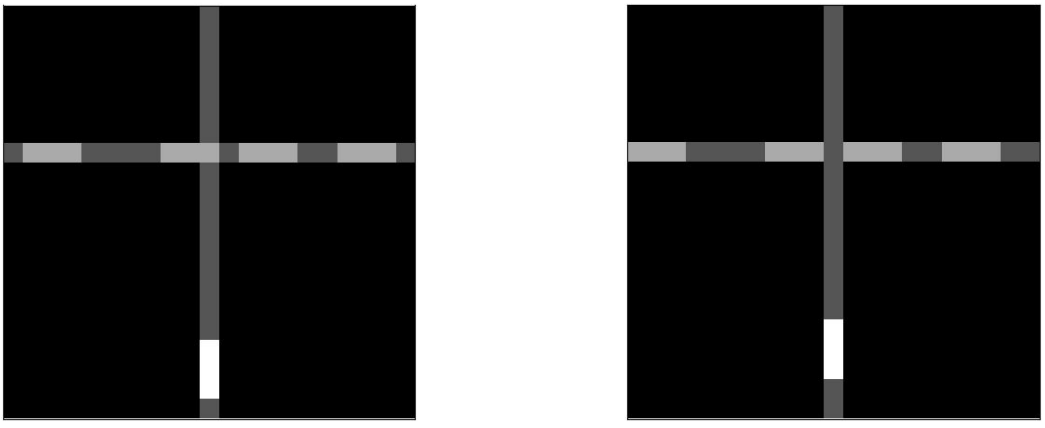

Figure 1. Two sequential frames produced by the environment. In the second frame (right) the agent has advanced upwards and the traffic has advanced leftwards by one pixel.

In the environment, the agent is not required to stop if there happens to exist a gap in the traffic large enough, otherwise it must come to a stop until such a gap presents itself (that's this agent's *raison d'être*). Since vehicles advance by one pixel per time step, we can say that a gap in traffic must be at least 3 pixels wide in order for the agent to "squeeze through".

Solution Statement

Three major components of the proposed project are here described. First, methods in the environment class are outlined, then the artificial neural network used to learn the Q values is described, and finally the Experience Replay implementation, a key feature of Q-learning, is mentioned.

The environment is instantiated in the following manner:

```
env = Drive(grid_dims)
```

where grid_dims is a tuple such as (21,21) representing the pixel height and width of the simulated world. Next the environment is set to initial conditions with:

```
env.reset()
```

Traffic is placed on the road with randomly chosen gap lengths. Gaps must be at least one pixel wide. In practice they are usually between one and four, though they can sometimes be several pixels wider. The following call moves the traffic leftward one pixel:

```
env.propagate_horz()
```

The agent then determines if it is at the intersection with the call:

```
env.at_intersection()
```

If it is not at the intersection, then the only valid action is "forward". Otherwise, the agent must decide whether to go forward or remain. When the action is determined, the agent action is implemented by:

```
env.propagate_vert(action)
```

The action input to this method is what we must learn.

In Q-learning, deep neural networks are used to learn relevant features of the environment. Instead of determining through trial-and-error which of the many features should be used to most efficiently describe the state of the environment at each time step, the RL practitioner may simply input the whole 2-dimensional image to the network. After all, for our use-case, the image with the associated actions take completely describes the state of the system.

The network output must be two numbers representing the probability to perform the two agent actions “forward” and “stay”. The deep learning library Keras was used to build a model that the final implementation is expected to resemble. The model.summary() method shows the proposed deep learning architecture.

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 441)               194922
dense_2 (Dense)              (None, 441)               194922
dropout_1 (Dropout)          (None, 441)               0
dense_3 (Dense)              (None, 441)               194922
dense_4 (Dense)              (None, 2)                 884
=================================================================
Total params: 585,650.0
Trainable params: 585,650.0
Non-trainable params: 0.0
```

The output shapes of all layers but the final are of size 21x21 = 441. The output layer is of size two. A trained model given an image of our environment as an input would output two numbers representing the decision to move forward or to stay put. For any situation that the agent encounters, i.e. any possible state of the environment, one of these numbers should be much larger than the other.

The Experience Replay functionality, and the reasons for using it is explained in reference 2 in this way:

“When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm.”

The experience replay implementation proposed here is almost identical to that Python class used in reference 3. Only one minor change is necessary to make it work with the output of the network described above.

The following software have been used: Python 3.6, Numpy, Scipy, Matplotlib, Keras, TensorFlow.

Benchmark Model

In the next section, evaluation metrics will be discussed by which the performance of the learned model can be quantified, but how should these results be interpreted. Would it be sufficient, for example, if we determined that the learned model performs better than

random chance? Should it perform better at crossing the simulated intersection than an average human driver crossing an actual busy intersection? The comparison between a human driver and the simulated road is not useful for the following reason: our computer agent moves exactly one unit forward in one discrete time step, even when moving from a stop, whereas a car must often accelerate from a stop, building up speed over continuously flowing time. This could be approximated with a much more sophisticated simulation perhaps, but there are doubtless many other differences which make such a comparison problematic. Let us say then that there are no benchmarks against which to compare the results, but that this agent should be so trained so that it only rarely fails out of thousands of trials.

Evaluation Metrics

To evaluate the learned model and the Q-value implementation, a python program was written which instantiates an environment and loads the trained model. This testing program presents the agent with thousands of randomly generated environments and records the following two evaluation metrics:

1. Percentage of successful crossings
2. Number of missed crossing opportunities

The first metric has already been implemented and tested. The second metric is still to be implemented. The best way to do this is probably through defining a callable method in the Drive class which returns True when a gap of three or greater in length first appears in front of the vehicle that has reached the intersection.

Project Design

At a minimum, the project would consist of the following files “Training.py”and “Testing.py”. Pseudocode for the training file is as follows:

```
for e in epochs:
    instantiate environment as env
    while trial not terminated:
        initialize env
        check if agent at intersection
        if at intersection:
            move forward
```

```
        else:
            if random int < epsilon:
                explore by choosing an action at random
            else:
                chose the policy model
        experience replay to get previous experiences
        update the model based on experience replay
```

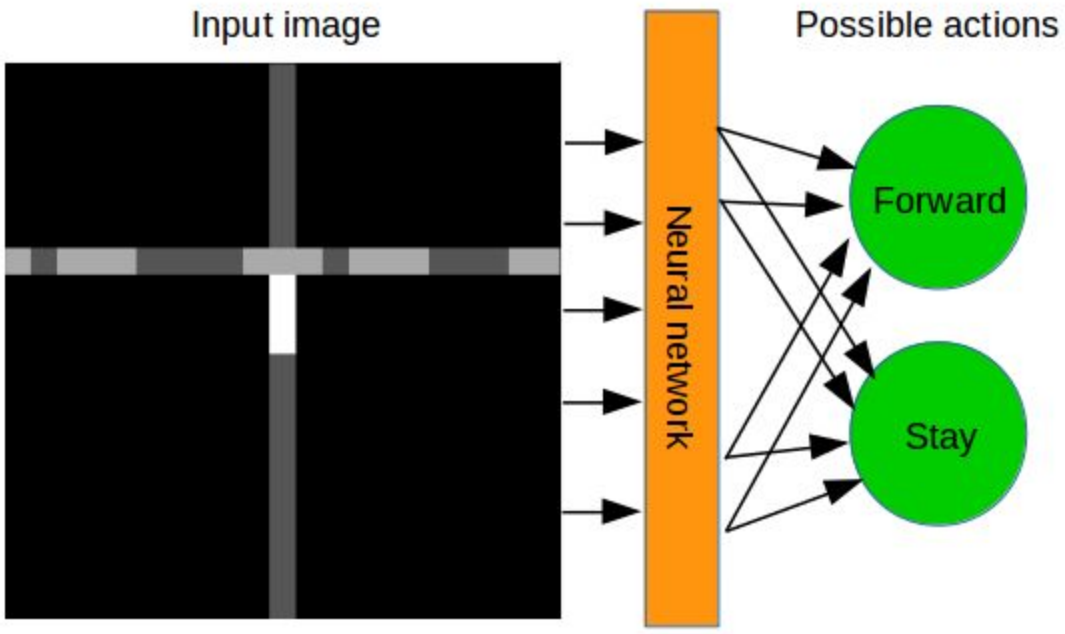

Figure 2. A simplified representation of deep reinforcement learning

Pseudocode for the training file is:

```
load pre-trained model
for t in trials:
    instantiate environment
    while trial not terminated:
        initialize environment
```

```
check if agent at intersection
if at intersection:
    use q-values from model to determine action
else:
    move agent forward
keep count of values used for evaluation metrics
```