CS 5600/6600: F21: Intelligent Systems
Open AI Gym Mountain Car README

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 04, 2021

Introduction

To follow this README, you'll need to isntall tensorflow and tflearn.org. You'll also need to install the Open AI Gym that we talked about in Lecture 09 (PDF is in Canvas).

This document gives you a very brief tutorial in TensorFlow and documents the steps I followed to train NNs to play the Mountain Car, an Open AI Gym evnironment.

Training NNs to play computer games is also bulldozing, but it's bulldozing of a different kind: instead of going through training, testing, and validation datasets to build classification models, we bulldoze through game iterations/epochs to train decision making models.

This README provides, in a nutshell, a framework for training AI systems to take over human operators (e.g., drivers, pilots, computer game players, chess/go/checkers players, journalists, stock traders, and, yes, us – programmers, etc.). In the first stage, we place as many real or virtual sensors in a target environment (e.g., a car, a plane, a desktop, an office, etc.) to collect decision making frames. A decision making frame is a vector of numbers, some of which represent the operator's actions (brake, turn the wheel x° left, turn the wheel x° right, push the altitude lever up/down, type "def" in the editor window, etc.)

When enough of these frames are collected (how much is enough is a context-sensitive question that we'll leave outside of the scope of this document), stage 2 (i.e., the "game playing") starts. At this stage, we define a decision making model (e.g., a ConvNet) and a reward policy (e.g., +10 if a truck's on the road, +5 if a truck stops at a red light, +10 if a truck slows down because of a pedestrian crossing the road, +100 if a generated article on a boxing match has 1000 likes on some social network, +15 if a defined Python function produces correct output, etc.) and use NN training combined with reinforcement learning (i.e., Deep Q Learning) to train and evaluate the model. Stage 3 is deployment on a real hardware/software platform.

Open AI Gym provides a plethora of game platforms and virtual environments. Don't get addicted though! The first time I went into the gym while chasing a reference from a technical blog, I spent 7 straight hours there. It's a ton of fun, but I also have a painful memory of staying up till 3:00 a.m. on that day to meet my other mandatory deadlines that have a tendency to pile up.

TensorFlow Coding Lab

I've included in the zip `tf01.py`, `tf02.py`, `tf03.py`, and `tf04.py`. These are small programs I wrote when I was teaching myself tensorflow by going through their online documentation. If you haven't used tensorflow extensively before, I suggest that you go through this quick coding lab to

become more comfortable with the library. If you know tensorflow (e.g., you use it at work or in your research project), skip it.

Let's go over the code in `tf01.py`. You can think of a tensor as a node in a graph and of a tensorflow as a graph of tensors.

```
0. import tensorflow as tf
1. CONST_1 = tf.constant(1.0, name='CONST_1')
2. A = tf.Variable(5.0, name='A')
3. B = tf.Variable(3.0, name='B')
```

On line 1, we define a tensorflow constant `CONST_1` (aka a constant sensor) and set its value to 1. When queried for value, this tensor always produces 1.0. On lines 2 and 3, we define two variables, `A` and `B`, respectively, and set their values to 5.0 and 3.0, respectively. Unless we change the values, these variable tensors, when queried for values, will produce 5.0 and 3.0, respectively.

Let's define a few tensor operations to define our tensorflow.

```
1. C = tf.add(A, B, name='C')
2. D = tf.add(B, CONST_1, name='D')
3. E = tf.multiply(C, D, name='E')
4. init_op = tf.global_variables_initializer()
```

On line 1, tensor `C` is computed by adding tensors `A` and `B`. On line 2, tensor `D` is computed by adding tensor `B` and `CONST_1` (a constant tensor defined above). On line 3, `E` is computed by multiplying `C` and `D`. On line 4 the standard tensorflow operation is defined to initialize a tensorflow session. One can think of a session as a separate process with its own scope.

We're ready to define a tensorflow session.

```
1. with tf.Session() as sess:
2.     sess.run(init_op)
3.     EV = sess.run(E)
4.     print('Variable E = {}'.format(EV))
```

Line 1 creates a tensorflow session and binds it to the variable `sess` and can be referenced within the scope of the `with`-macro. Line 2 initializes the session. These two lines are boilerplate in tensorflow code, and you'll see/do them over and over again if you start using tensorflow on a regular basis.

The interesting stuff starts on line 3. The method `run()` queries tensor `E` for value and computation starts flowing backward. Specifically, tensor `E` must multiply tensors `C` and `D` so it queries `C` and `D` for values. Tensor `C`, in turn, queries `A` and `B` for values. Tensors `A` and `B` are variable sensors with defined values, so they produce 5.0 and 3.0, which `C` adds to produce 8.0. In the meantime and parallel to the computation of `C`, tensor `D` queries tensors `CONST_1` and `B`. The latter two produce 1.0 and 3.0, respectively, which `D` adds to produce 4.0. Finally, tensor `E` takes the value produced by `C` (i.e., 8.0) and the value produced by `D` (i.e., 4.0), multiplies them to produce 32.0, which is saved in `EV` and printed out. Here's the output.

```
>>> python tf01.py
Variable E = 32.0
```

OK. So, what's the big deal? The big deal is that each path in a tensorflow can run on a separate thread/CPU, which makes these computations higly parallelizable and, therefore, efficient. Another

thing is the each tensor in a tensorflow can be a complex multi-dimensional matrix or a separate tensorflow. To put it differently, one can combine tensorflows into more complex tensorflows.

Let's quickly go over `tf02.py`, which illustrates how we can feed numpy arrays into tensoflows with tensorflow placeholders.

```
1. const = tf.constant(2.0, name='const')
2. b = tf.placeholder(tf.float32, [None, 1], name='b')
3. c = tf.Variable(1.0, name='c')
```

Tensor `b` on line 2 is defined as a placehodler for 1D arrays of `float32` values. Note that its dimensions are defined as `[None, 1]`. The value of `None` means that we don't know a priori how many `float32` values there'll be. There can be 5, 10, or 1 000 000.

Let's define a few tensor operations.

```
1. d = tf.add(b, c, name='d')
2. e = tf.add(c, const, name='e')2.
3. a = tf.multiply(d, e, name='a')
```

On line 1 in the above code segment, we define tensor `d` as the output of adding tensor `b` (i.e., an array) and tensor `c` (i.e., a variable). What's `d` going to be? It'll be a tensor array obtained by adding the value of `c` to each element in `b`. On line 3, each element of `d` is multiplied by the value of `e`.

The question is, how can we construct a numpy array to feed into this tensorflow where computation flows back from tensor `a` to `b` and `c`. This is done through feed dictionaries. A feed dictionary is a dictionary of placeholders and their values that are looked up and fed into the placeholders when appropriate. Here's an example.

```
>>> d = {b: np.arange(0, 5)[:,np.newaxis]}
>>> d
{<tf.Tensor 'b:0' shape=(?, 1) dtype=float32>: array([[0],
       [1],
       [2],
       [3],
       [4]])}
```

What we did in the code segment above was to construct a dictionary `d` and bound the key `b` (that's the placedholder defined above) to a 5-element numpy array converted to a column vector. Let's define a session.

```
1. init_op = tf.global_variables_initializer()
2. with tf.Session() as sess:
3.     sess.run(init_op)
4.     a_out = sess.run(a,
5                       feed_dict={b: np.arange(0, 10)[:, np.newaxis]})
6.     print('Variable a is {}'.format(a_out))
```

Everthing is boilerplate, except for lines 4 and 5. The method `run()` takes the output sensor and the feed dictionary defined with the keyword parameter `feed_dict`. When, at any point in the tensorflow computation, the value of a tensor is needed, it'll be looked up in the feed dictionary under an appropriate key. In particular, if at any point in the defined tensorflow, the value of `b` will be needed, it'll be looked up in the feed dictionary.

We're ready to run the tensorflow in `tf02.py` and figure out what's computed. Let's do it.

```
>>> python tf02.py
Variable a is [[ 3.]
 [ 6.]
 [ 9.]
 [12.]
 [15.]
 [18.]
 [21.]
 [24.]
 [27.]
 [30.]]
```

Tensor `a` multiplies `d` and `e` so it queries both for values. Tensor `d` adds `b` and `c`. Since `b` is a placehoder, it doesn't have a predefined value, so the value is looked up in the feed dictionary and found to be a 1D column numpy array of floats from 0 to 9. Tensor `c` is queried for value and readily produces 1.0, because its a variable. Now tensor `d` produces the value by adding 1 to each element in the 1D column numpy array, which produces a 1D column numpy array of floats from 1 to 10.

Next tensor `e` is queried for value. It adds tensors `c` and `const` to produce 3. Now tensor `a` has the necessary tensor values to do its computation, i.e., to multiply `d` and `e`, which produces a 1D column array of values from 3 to 30, and that's, indeed, what prints out.

The files `tf03.py` and `tf04.py` supply more examples of defining sessions, tensorflows, feeding arrays into tensorflows, and putting multiple entries into feed dictionaries.

Mountain Car from Open AI Gym

Recall the mountain car game we discussed in Lecture 09. It's one of the simpler games in the Open AI Gym. The zip contains the `mountain_car` folder with three files: `MCar.py`, `Memory.py`, and `Model.py`. If the Open AI Gym is installed, you can run this Deep Q learner right away as follows.

```
>>> python MCar.py
```

You should see a small car bouncing left and right on the screen, as I showed you in class at the end of Lecture 09.

The objective of the car is to learn to hit the yellowish flag on the right hill as many times as possible. A state of the car is defined by its position and velocity (See Slide 25 in `CS5600_6600_F21_RLNN_Part02.pdf` in the Lecture 09 module in Canvas). The actions available to the car are: push left, no push, push right. A correct strategy to learn is to push as far back onto the left hill and use the momentum to carry you through to the top of the right hill. As the car trains, the lines you'll see in your command window should look like these.

```
Episode 1 of 10
Step 1, Total reward: -1.0, Eps: 0.999901004949835
Step 2, Total reward: -2.0, Eps: 0.9998020197986801
Step 3, Total reward: -3.0, Eps: 0.9997030445455454
Step 4, Total reward: -4.0, Eps: 0.999604079189441
Step 5, Total reward: -5.0, Eps: 0.9995051237293776
Step 6, Total reward: -6.0, Eps: 0.9994061781643654
Step 7, Total reward: -7.0, Eps: 0.9993072424934148
Step 8, Total reward: -8.0, Eps: 0.9992083167155369
Step 9, Total reward: -9.0, Eps: 0.9991094008297421
...
```

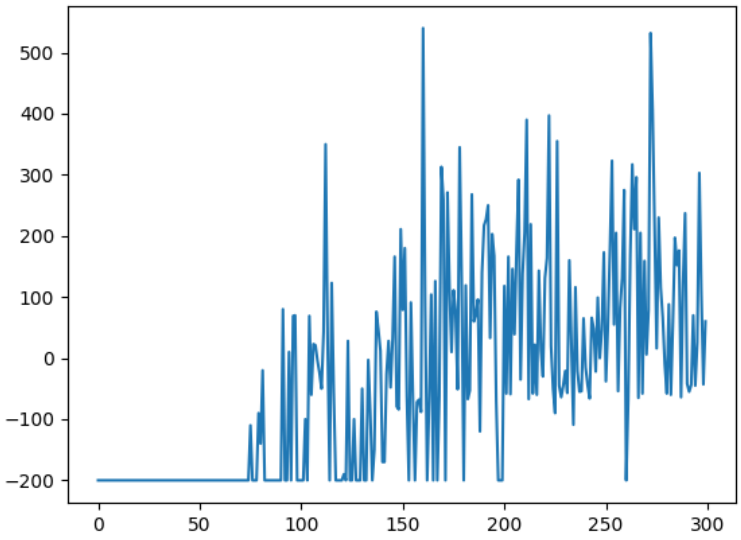

Figure 1: Mountain car's rewards of 2x50x50x3 NN after 300 episodes.

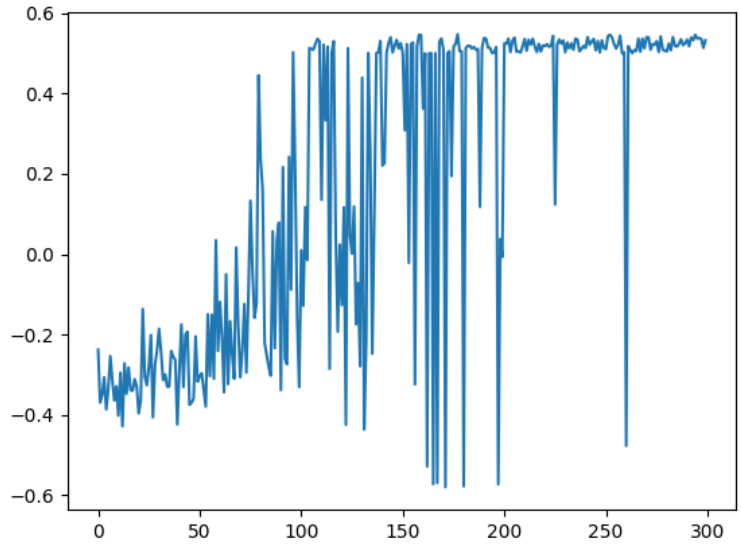

Figure 2: Mountain car's maximum X-position of 2x50x50x3 NN after 300 episodes.

After the RL training stops, the program displays two plots: the rewards plot (episodes vs. rewards), as shown in Fig. 1, and the maximum x-position plot (episodes vs. x-positions), as shown in Fig. 2. As you can see from the reward plot in Fig. 1, it should go up with more episodes. In the max x-position plot, the x-position should approach 0.6.

These plots are constructed from the two array members in the `MCar` class: `self._reward_store` and `self._max_x_store`. You can use these arrays to compute the total award and the average x-position reached by the car over all episodes.

The number of training episodes is defined in the variable `num_episodes` in the function `train_car()`

in `MCar.py`. I generated these plots by training an NN with two 50-neuron fully connected layers for 300 episodes (see the method `_define_model_1(self)` in `Model.py`, which took ≈15 minutes on my laptop.

The code in `MCar.py` implements the monte carlo RL algorithm we discussed on 10/07/20. You don't need to modify this file (other than modifying `num_episodes`) in the function `train_car()`. Obviously, the more episodes you train for, the better the resulting model, but the training will take more time.

The file `Model.py` is where you can define your own NN models that will be trained by RL to drive the car. For example, the method `_define_model_2` builds a 2x50x50x3 NN trained to drive the car. The input to this net is the current state of the mountain car (position and velocity). The output is a 3-element vector of actions: push left, no push, push right.

```
def _define_model_2(self):
    self._states = tf.placeholder(shape=[None, self._num_states],
                                  dtype=tf.float32)
    ## This is the Q(s, a) table.
    ## The number of columns is determined by the input fed to it
    ## The number of actions is 3. So, the size of _q_s_a is ? x 3,
    ## because there are 3 actions in the game.
    self._q_s_a = tf.placeholder(shape=[None, self._num_actions],
                                 dtype=tf.float32)
    # create NN with two fully connected hidden layers
    self._fc1 = tf.layers.dense(self._states, 50, activation=tf.nn.relu)
    self._fc2 = tf.layers.dense(self._fc1, 50, activation=tf.nn.relu)
    self._logits = tf.layers.dense(self._fc2, self._num_actions)
    loss = tf.losses.mean_squared_error(self._q_s_a, self._logits)
    self._optimizer = tf.train.AdamOptimizer().minimize(loss)
    self._var_init = tf.global_variables_initializer()
```

You can follow the example in `_define_model_2()` above. The number of neurons in each hidden layer is up to you. I've left all my comments in the source code as I was getting it to work. I hope they'll prove helpful to make sense of the program's structure. But, you don't really need to modify anything – just add a methods (e.g., `_define_model_3()`) and train the model it defines.

You can use the method `_define_model()` in `Model.py` to specify the current NN model being trained. The default definition is as follows.

```
def _define_model(self):
    self._define_model_2()
```

I recommend that you train each model for at least 300 episodes and generate the reward and max x-position plots for each model to see whether your model is successful.

The only file you need to modify is `Model.py`, where you'll define your models. We'll use `MCar.py` and `Memory.py` as they are given in the zip in our unit tests on just a couple of episodes to make sure that your models can train.

If you want to define your method in `MCar` to compute the total award and the average x-position reached by the car over all episodes for a given model to answer the question which NN gave you the highest total award and the greatest average x postion, go ahead and do it.

Happy Hacking!