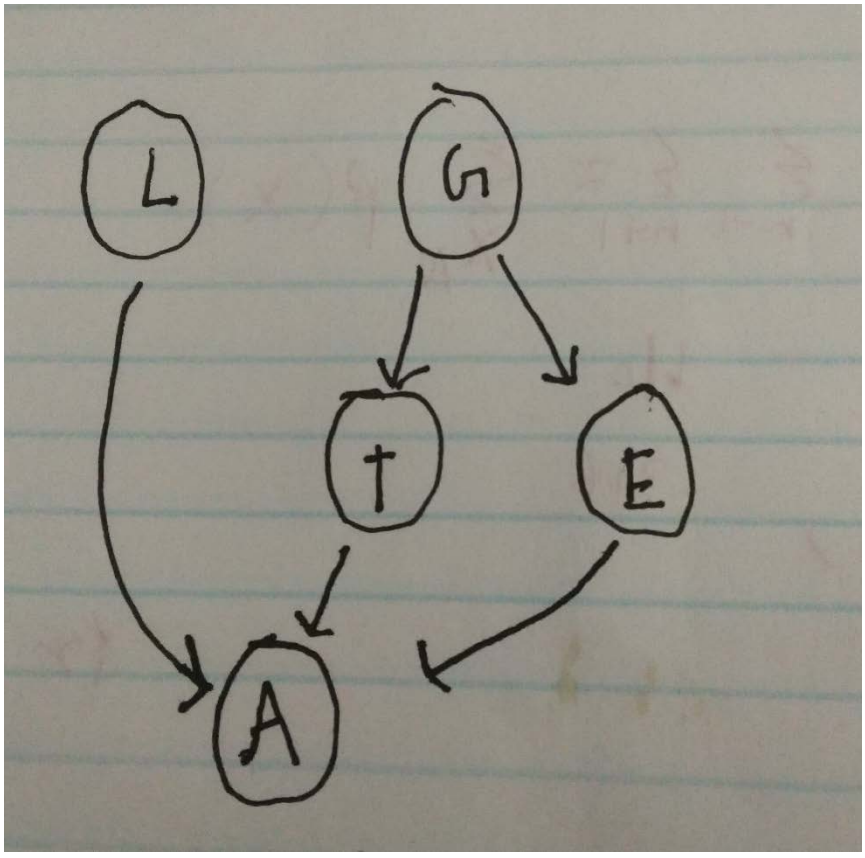
Khizr Ali Pardhan | 301314376 |March 24th 2020

CMPT 419 – Assignment 2

1 Graphical Models

1)

According to my model, government impacts both tuition and economy, both of which, along with parents' education level will impact if someone goes to SFU.
Economy related to how much money a student has available. Tuition is related to how much a student has to pay. So they both connect to node A



2)

$$p(z_1, \dots, z_k) = \prod_{k=1}^{K} p(x_k | pa_k)$$

$$p(A, L, G, E, T) = p(L) * p(G) * p(E\ |G) * p(T\ |G) * p(A|L)\ * p(A|E)\ * p(A|T)$$

3)

$$P(L) = \ p(o = 0.8) + p(u = 0.2)$$

Discrete random variable

$$P(G) = \ \mathrm{p(l\ =\ 0.5)} + p(d = 0.5)$$

Discrete random variable. Assuming even chance of election

$$P(E|G) = N(\ 245, 5)$$

Continuous normally distributed variable. Expressed in billions of Canadian dollars.

$$P(T|G)\ will\ be\ \dots$$
$$If\ Liberal\ party, N(250{,}10)$$
$$If\ NDP\ party, N(240{,}10)$$

Continuous normally distributed variable. Expressed in billions of Canadian dollars.

$$P(A|L,\ E,\ T)$$

Discrete random variable, from 1 discrete random variable and 2 continuous variables.

To obtain, we can take either a weight linear combination with a round function. or use the sigmoid function.

I will elect to use the sigmoid function since PRML states “using a logistic sigmoid function acting on a linear combination of the parent variables”. When going from a continuous parent node, to discrete child node we need either of the following functions.

$$P(A|L,\ E,\ T) = \ SIGMOID\ (W^t\ * X) = \frac{1}{1 + e^{-W^t X}}$$

Also, the SoftMax could be used.

$$P(G = i\ |E = x) = \frac{\exp(w_i x + \ b_i)}{\sum_j \exp\ (w_j x + b_j)}$$

According to this source, decay populated areas tend to get less than 65% voter turnout, and knowing that mathematically, the votes of 1-2 parents has a minimal effect on the election, for which the population is about 5 million or larger. For parameter reduction, p(goverment|education of parents) is assumed to be p(government)

4.

We want

$$L(\Theta: D) = P(D|\Theta) = \Pi\, P(X_n: \Theta)$$

$$p(D|\mu) = \prod_{n=1}^{N} \mu^{x_n}(1-\mu)^{1-x_n}$$

Where *Theta* or MU are the parameters we want to learn

Taking the log, which is monotonically increasing, preserves the relative shape while making an expensive multiplication in the cheaper summation. This takes the form,

$$\frac{1}{N}\sum_{n=1}^{N} logp(X|Y)$$

To determine

$$P(A|pa_A)$$

We must learn MU, which follows from its parent nodes. All elements from the needed, unless a element is in no way linked (which is not that case here). This is done with the factorization formula (8.5)

$$l(X|\,\Theta) \;=\; \Pi\_k\; \Pi\_n\; P(A_n^k|\; pa_n^k, \Theta)$$

Next, we take the argmax of Θ, which is a maximization operation. This is which we can leverage factorization and the log function. Then we take a gradient. we have arrived at a relatively simple optimization problem. Note that the optimization won't necessarily be convex, I call it simple because it's just a few lines of code.

2 Reinforcement Learning

1)

My ANN with two “two fully hidden layers”. The second fully hidden layer is un-activated

```
self.layer_1 = tf.layers.dense(self.inputs, h1_size,
                                                activation=tf.nn.relu,
                                                kernel_initializer=tf.contrib.layers.xavier_initializer())
                                                # "ideal"

self.hidden_layer_2 = tf.layers.dense(self.layer_1, h2_size,
                                                activation=tf.nn.relu,
                                                kernel_initializer=tf.contrib.layers.xavier_initializer())

self.hidden_layer_3 = tf.layers.dense(self.hidden_layer_2, a_size,
                                                activation=None,
                                                kernel_initializer=tf.contrib.layers.xavier_initializer())
```

2)

My ANN with an un-activated final layer, in this case is feed into a SoftMax activation function.

```
#// use with `action_distribution = sess.run(agent.output_layer ,feed_dict={agent.inputs:[state]})`
# softmax ONLY FOR action_distribution in main loop
self.output_layer = tf.nn.softmax(self.hidden_layer_3)
```

3)

My ANN with an un-activated final layer, in this case is feed into a SoftMax cross entropy with logits function, this function wants an un-activated layer. This is multiplied by the advantage vector

I called reduce mean on the loss

```
### -- Part e) Define loss function for policy improvement procedure --
# uses hidden_layer_3, which has no activation
self.loss_temp = self.advantage * tf.nn.sparse_softmax_cross_entropy_with_logits(logits=self.hidden_layer_3, labels=self.actions)
self.loss = tf.reduce_mean(self.loss_temp)
```

4)

In the code I included my reference.

I noticed some recommend normalization of ‘advantage’

```
reward = r
reward_list = []
episode_list = []
eq_reward = []
episode=[]
for index, el in enumerate(j):
    if index != el:
       print(index == el)
    if index == 0:
        episode.append(el)
        eq_reward.append(reward[index])
    elif el == 0:
        episode_list.append(episode)
        episode = [0] #re init
        reward_list.append(eq_reward)
        eq_reward = [reward[index]] #re init
    elif index == len(j)-1:
        episode.append(el)
        episode_list.append(episode)
        eq_reward.append(reward[index])
        reward_list.append(eq_reward)
    else:
        episode.append(el)
        eq_reward.append(reward[index])
episode, eq_reward = None, None
advantage = []
for i, epi in enumerate(episode_list):
    b=0
    discount = 1
    for t in range(len(epi)):
        b += discount * reward_list[i][t]
        discount *= gamma
    for t in epi:
        discounted_reward = 0
        discount = 1
        for idx in range(t,len(epi)):
            discounted_reward += discount * reward_list[i][idx]
            discount *=gamma
            advantage_t = discounted_reward - b
        advantage.append(advantage_t)
return advantage
```

5)

```
### --- Part g) create the RL agent ---
agent = agent(0.001, 4, 2, 8, 8)
```

I created an agent with learning rate, states = 4, actions = 2, layer_1 = 8, layer_2 = 8

```
### ------------ Part g) -------------------
### Probabilistically pick an action given policy network outputs.
action_distribution = sess.run(agent.output_layer ,feed_dict={agent.inputs:[state]})
action = np.random.choice(action_distribution[0],p=action_distribution[0])
action = np.argmax(action_distribution == action)
### ---------------------------------------
```

Here I called part 2, which is an ANN with an unactivated final layer, in this case which is feed into a SoftMax activation function.

I get the random.choice with provide p values (from the SoftMax), due to my 1st parameter not being the action_space, I must take a argmax to get my action.

```
### --- Part g) Perform policy update ---
feed_dict={agent.advantage:history[:,2],
    agent.actions:history[:,1],agent.inputs:np.vstack(history[:,0])}
sess.run([agent.update_batch], feed_dict=feed_dict)
```

Here I have agent.update_batch to optimize the gradients. In this call I pass in a feed_dict of states, actions, and advantage

6)

Due to file size and coursys limitations, I posted the video on YouTube.

- *4:17 - https://youtu.be/lvzvjtQzvck?t=257*
- *5:40 - https://youtu.be/lvzvjtQzvck?t=343*

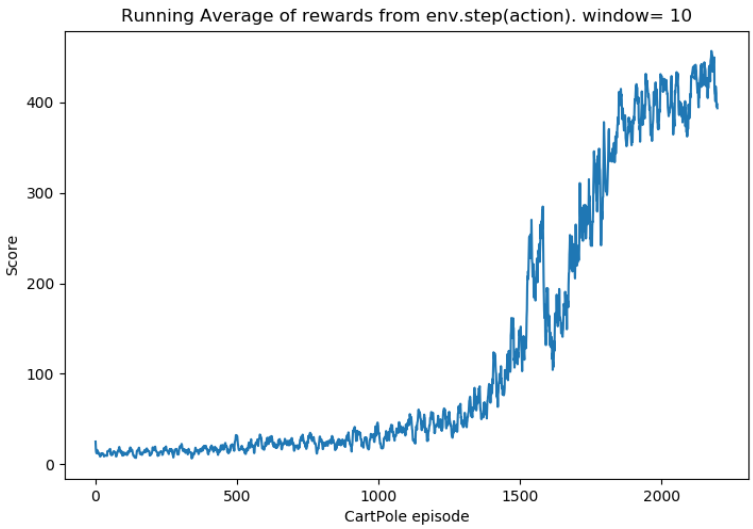
The training ended at a sub optimal level. So in #7 I included “score” (reward) across episode plot.

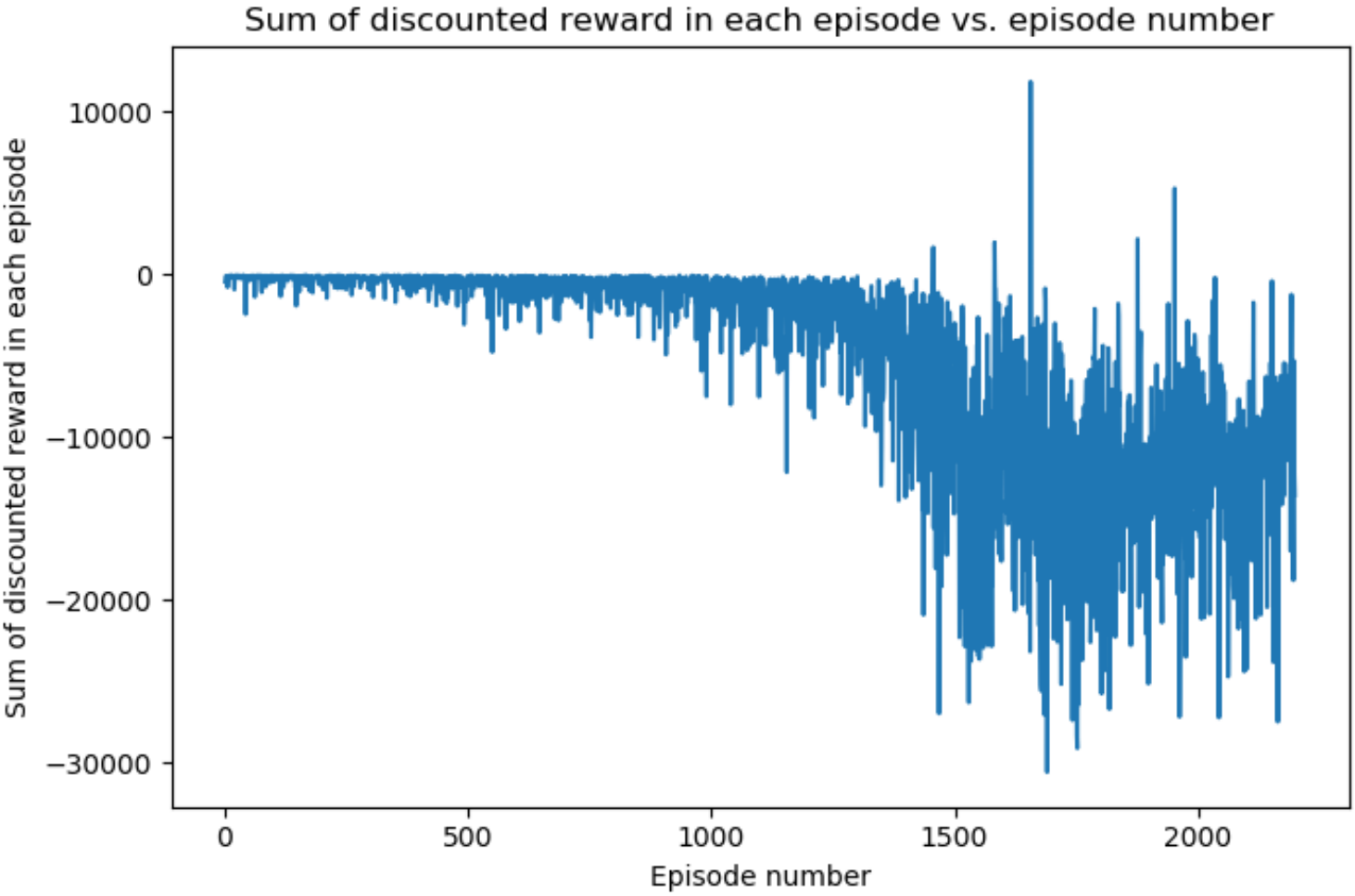
7)

Non-Normalized advantage function

To my understanding we where not suppose to normalize the advantage function.

However, I would like to mention that there was an improvement in ‘score’ and run time.

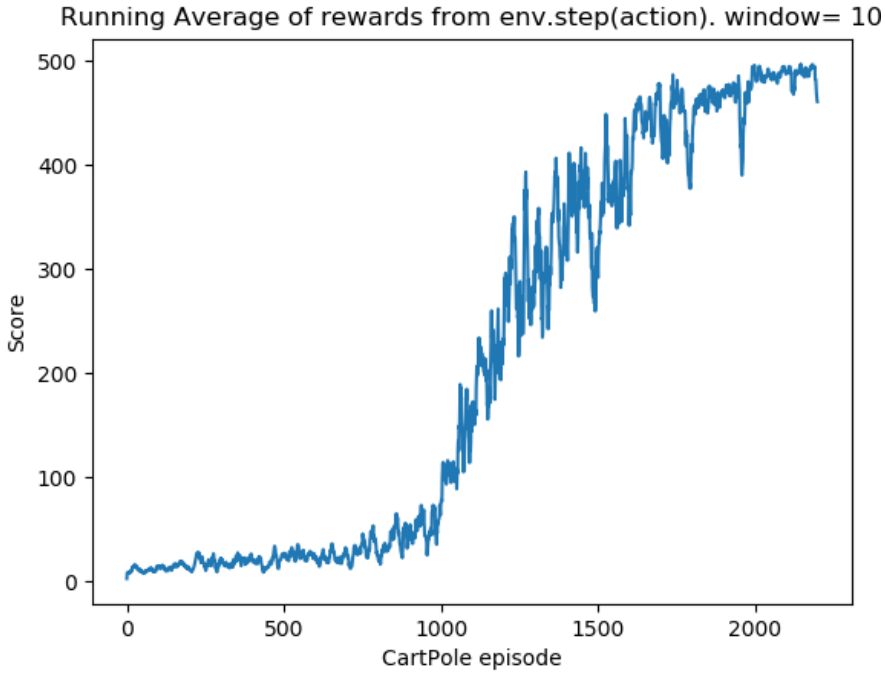


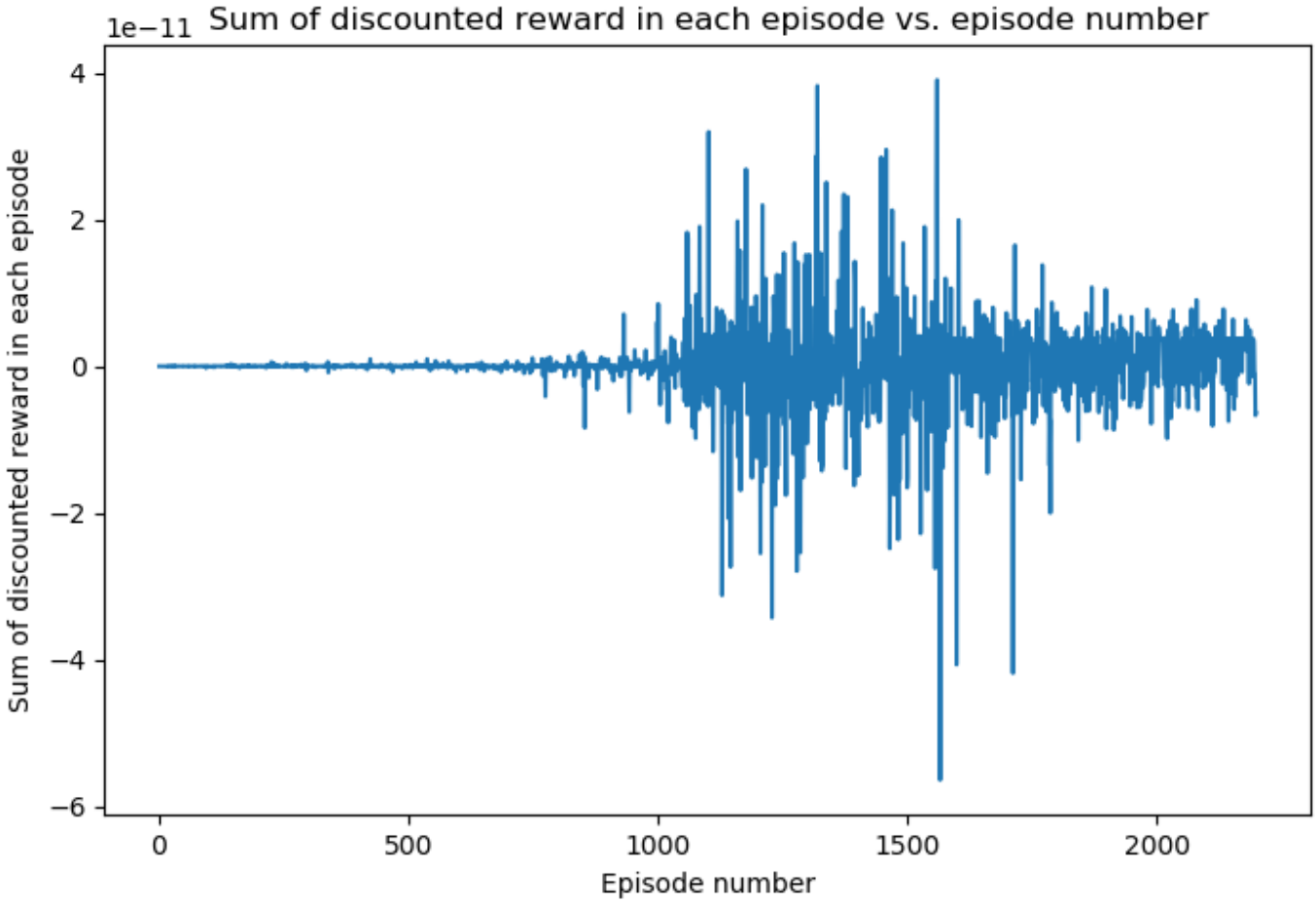


The above graph shows how the sum of discounted rewards increases in magnitude towards the end. This is due to episodes lasting longer.

n) This not in the question, however, I would like to mention that it took a few hundred more episodes of the *Non-Normalized advantage function* case to start showing a rewards (form env.step) of 500.

Normalized advantage function,
appears to be 'ideal'





3 Gated Recurrent Unit

$$reset\ gate: \mathrm{r_j} = sig\big([W_r - x * x]_j + [U_r * h_{t-1}]_j\big)$$

$$\text{update gate: } \mathrm{z_j} = sig\big([W_z - x * x]_j + [U_z * h_{t-1}]_j\big)$$

$$h_j^t = z_j H_j^{t-1} + \big(1 - z_j\big)\boldsymbol{H}_j^t$$

Next potential hidden unit: $\hbar_j^{(t)} = \phi\ \big(\ [Wx]_j + [r \odot h_{(t-1)}]_j\big)$

1)

If z_j, the rest gate, is close to 1, our first sub-term would be 'weighted' closed to 1, while our second sub-term $\big(1 - z_j\big)\boldsymbol{H}_j^t$ would be 'weighted' closed to 0. Our 'new' $h_j$is similar to it it's past state.

For the sake of simplicity, I illustrate z_j as a 'weight' for hidden sate h_t.

2)

Resetting will occur due to the 'weighting' behavior I describe above. The previous hidden state is almost disregarded (as it is weighted close to 0).

our second sub-term $\big(1 - z_j\big)\boldsymbol{H}_j^t$, will cause resetting to take place, by allowing "the hidden state to drop any information that is found to be irrelevant later in the future". The new hidden state will be governed by the current inputs

$$\hbar_j^{(t)} = \phi\ \big(\ [Wx]_j\big)$$