2021 Spring STAT 413 : Machine Learning

Jimmy Le
Department of Statistics
University of California, Los Angeles

jimml12@g.ucla.edu

Abstract

In this course, you will learn multiple algorithms in machine learning. In each homework, you are required to write the corresponding codes. I will provide a structure of code and a detailed tutorial online. Your job is (1) to deploy and understand the code. (2) Run the code and output reasonable results. (3) Make ablation study and parameter comparison. (4) Write the report with algorithms, experiment results and conclusions.

You will use this latex template on every homework. Each homework is a section of this report. you should comment out the section which is not for this homework. e.g. comment out homework 1-4, 6 when submitting your homework 5. At the end of the quarter, you will submit the full version report.

Homework submission forms

You have to submit both code and reports. For the code, zip everything into a single zip file. For the report, use this latex template. When submitting, no need to submit the tex file. Submit one single pdf file only.

1 Final: ???

1.1 DQN-Learning

1.1.1 Introduction

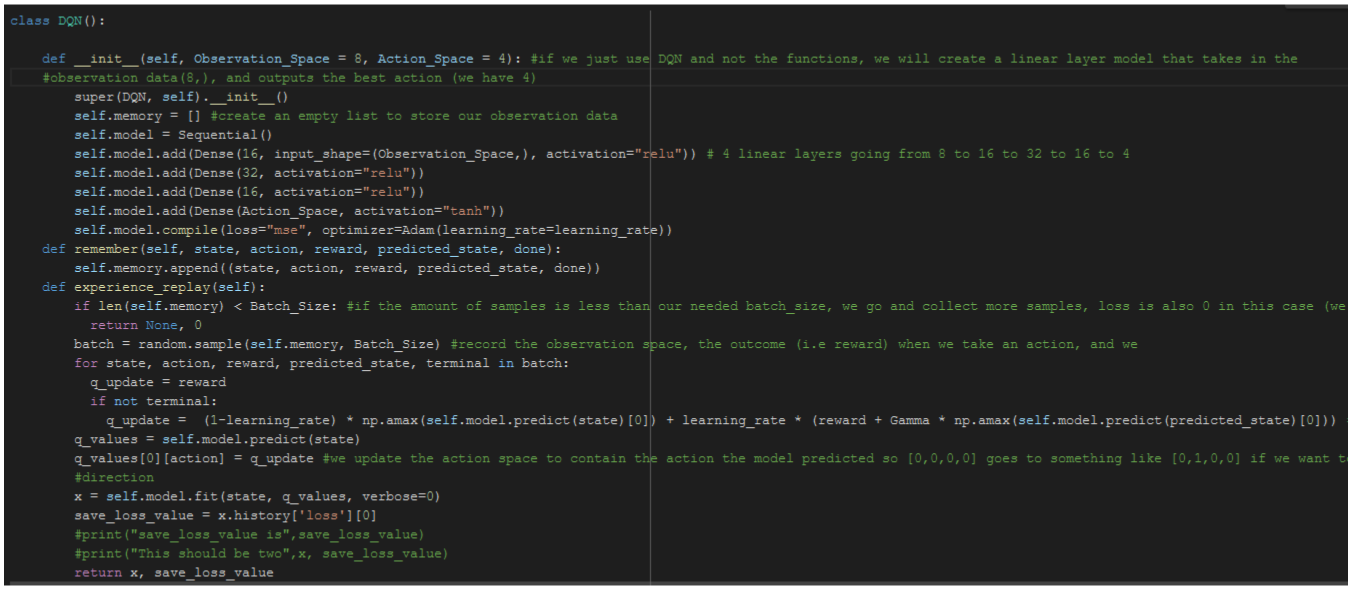
Deep Q Network Learning, is a reinforcement learning algorithm that takes in the observation space and tries to figure out the best action that maximizes the reward (Q). Chess algorithms use this type of learning method. We take in information of past actions and the consequences of the action in terms of reward and the observed state, and try to see if there's another action that creates better reward and/or observed state.

1.1.2 Key Equation

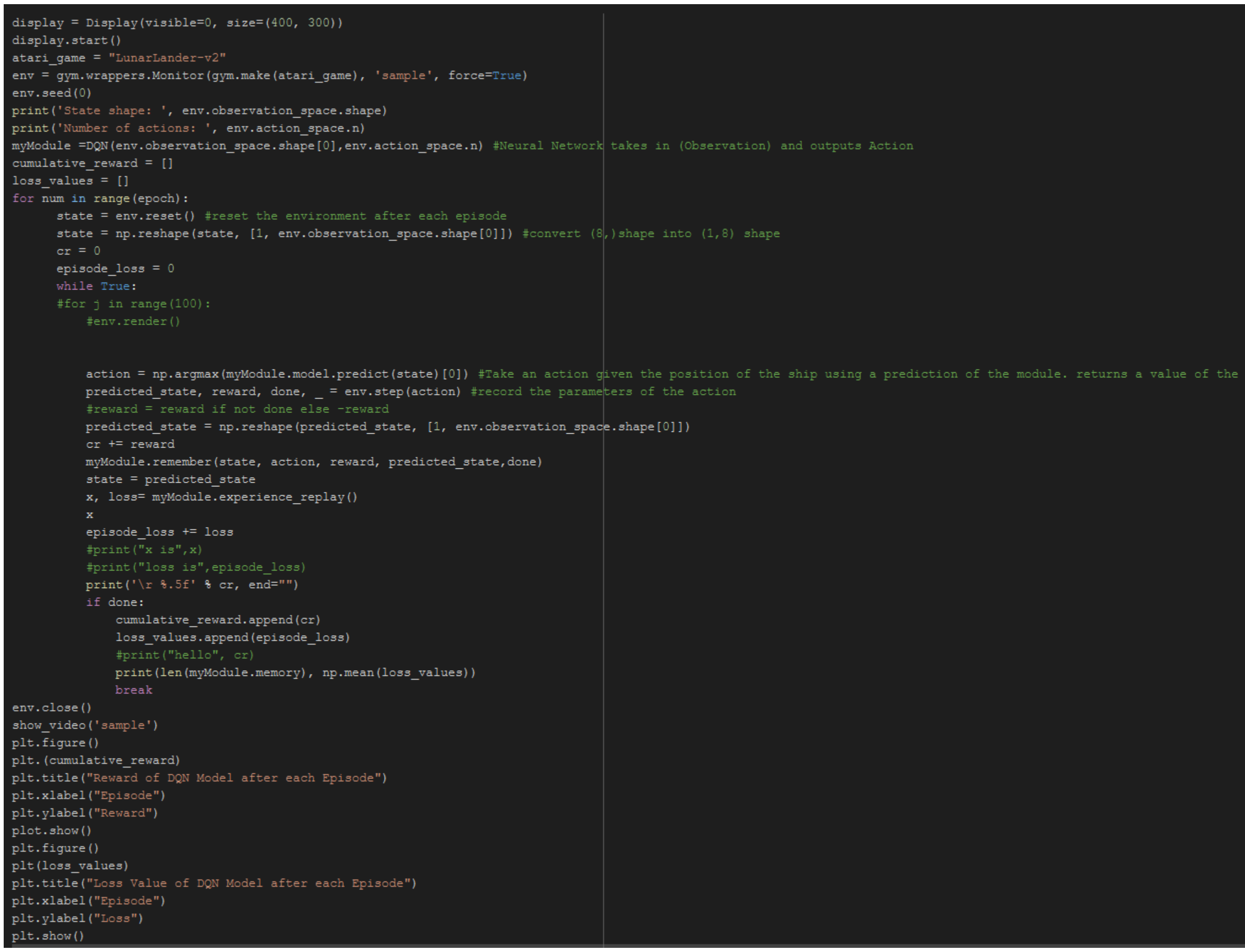
$$Q^{new}(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha(reward_t + \gamma * maxQ(s_{t+1}, a_t)) \quad (1)$$

What the equation means is that we shall calculate the expected reward at state t and with action t with the expected reward at state t+1 multiplied by a factor of gamma. Learning rate in this equation simply means how confident we are to use the new information over the old information.

1.1.3 Code



```
class DQN():

    def __init__(self, Observation_Space = 8, Action_Space = 4): #if we just use DQN and not the functions, we will create a linear layer model that takes in the
    #observation data(8,), and outputs the best action (we have 4)
        super(DQN, self).__init__()
        self.memory = [] #create an empty list to store our observation data
        self.model = Sequential()
        self.model.add(Dense(16, input_shape=(Observation_Space,), activation="relu")) # 4 linear layers going from 8 to 16 to 32 to 16 to 4
        self.model.add(Dense(32, activation="relu"))
        self.model.add(Dense(16, activation="relu"))
        self.model.add(Dense(Action_Space, activation="tanh"))
        self.model.compile(loss="mse", optimizer=Adam(learning_rate=learning_rate))
    def remember(self, state, action, reward, predicted_state, done):
        self.memory.append((state, action, reward, predicted_state, done))
    def experience_replay(self):
        if len(self.memory) < Batch_Size: #if the amount of samples is less than our needed batch_size, we go and collect more samples, loss is also 0 in this case (we
          return None, 0
        batch = random.sample(self.memory, Batch_Size) #record the observation space, the outcome (i.e reward) when we take an action, and we
        for state, action, reward, predicted_state, terminal in batch:
          q_update = reward
          if not terminal:
            q_update =  (1-learning_rate) * np.amax(self.model.predict(state)[0]) + learning_rate * (reward + Gamma * np.amax(self.model.predict(predicted_state)[0]))
        q_values = self.model.predict(state)
        q_values[0][action] = q_update #we update the action space to contain the action the model predicted so [0,0,0,0] goes to something like [0,1,0,0] if we want t
        #direction
        x = self.model.fit(state, q_values, verbose=0)
        save_loss_value = x.history['loss'][0]
        #print("save_loss_value is",save_loss_value)
        #print("This should be two",x, save_loss_value)
        return x, save_loss_value
```



```
display = Display(visible=0, size=(400, 300))
display.start()
atari_game = "LunarLander-v2"
env = gym.wrappers.Monitor(gym.make(atari_game), 'sample', force=True)
env.seed(0)
print('State shape: ', env.observation_space.shape)
print('Number of actions: ', env.action_space.n)
myModule =DQN(env.observation_space.shape[0],env.action_space.n) #Neural Network takes in (Observation) and outputs Action
cumulative_reward = []
loss_values = []
for num in range(epoch):
      state = env.reset() #reset the environment after each episode
      state = np.reshape(state, [1, env.observation_space.shape[0]]) #convert (8,)shape into (1,8) shape
      cr = 0
      episode_loss = 0
      while True:
      #for j in range(100):
          #env.render()


          action = np.argmax(myModule.model.predict(state)[0]) #Take an action given the position of the ship using a prediction of the module. returns a value of the
          predicted_state, reward, done, _ = env.step(action) #record the parameters of the action
          #reward = reward if not done else -reward
          predicted_state = np.reshape(predicted_state, [1, env.observation_space.shape[0]])
          cr += reward
          myModule.remember(state, action, reward, predicted_state,done)
          state = predicted_state
          x, loss= myModule.experience_replay()
          x
          episode_loss += loss
          #print("x is",x)
          #print("loss is",episode_loss)
          print('\r %.5f' % cr, end="")
          if done:
              cumulative_reward.append(cr)
              loss_values.append(episode_loss)
              #print("hello", cr)
              print(len(myModule.memory), np.mean(loss_values))
              break
env.close()
show_video('sample')
plt.figure()
plt.(cumulative_reward)
plt.title("Reward of DQN Model after each Episode")
plt.xlabel("Episode")
plt.ylabel("Reward")
plot.show()
plt.figure()
plt(loss_values)
plt.title("Loss Value of DQN Model after each Episode")
plt.xlabel("Episode")
plt.ylabel("Loss")
plt.show()
```

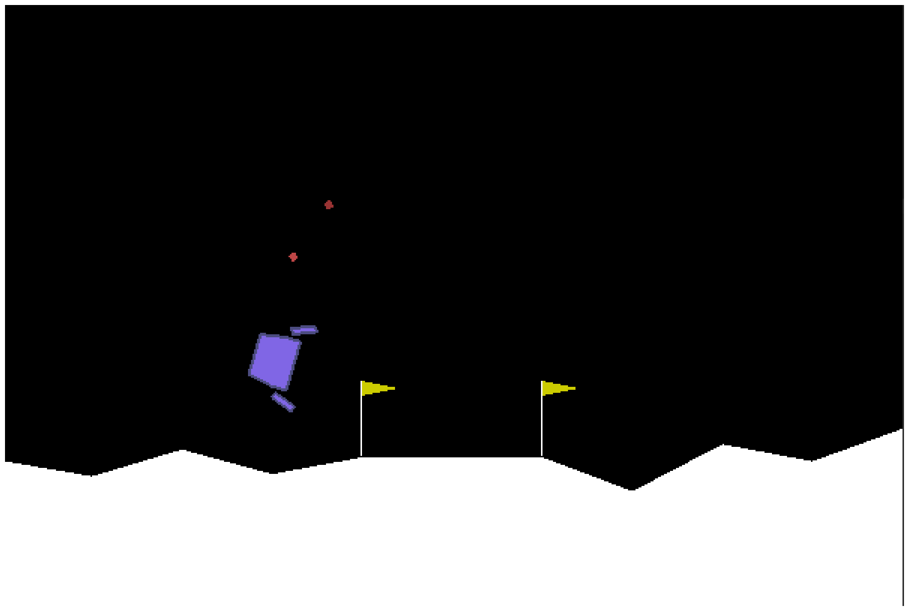


Figure 1: Model keeps predicting the same action over and over, however the model is being updated so not sure what's going on. Also the model likes to dive-bomb the ship into the goal so it means it's not respecting the reward value of velocity.

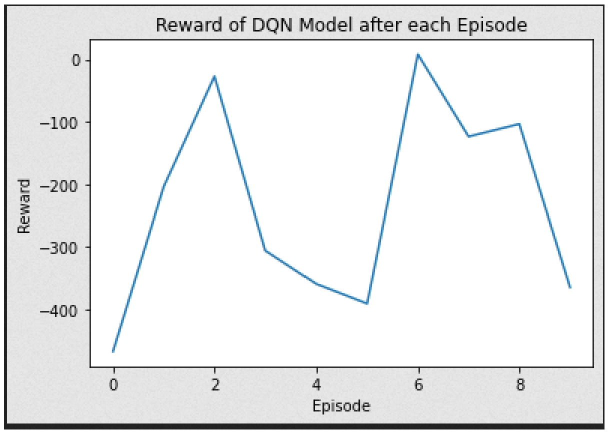


Figure 2: Reward of DQN at Gamma = 0.5 and learning rate = 0.001

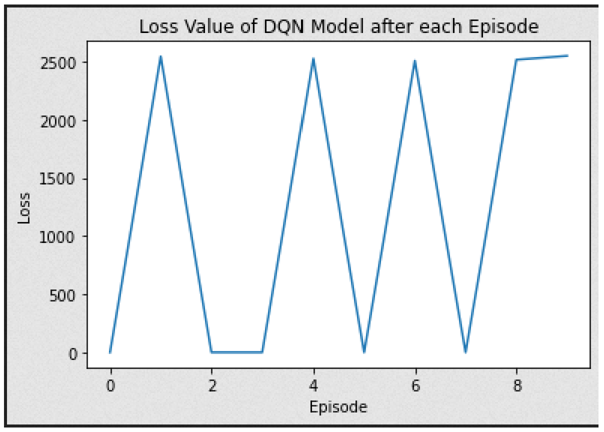


Figure 3: Loss of DQN at Gamma = 0.5 and learning rate = 0.001

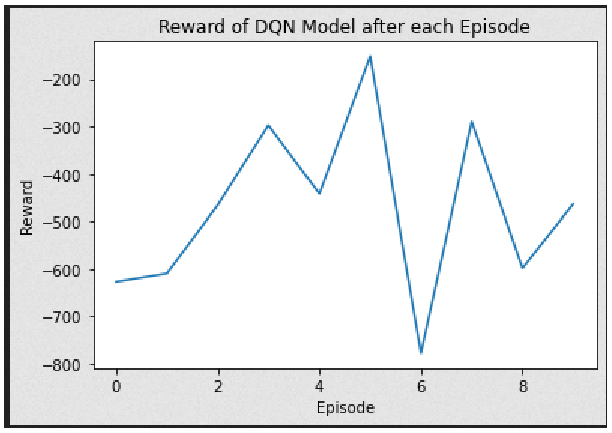


Figure 4: Loss of DQN at Gamma = 0.8 and learning rate = 0.01

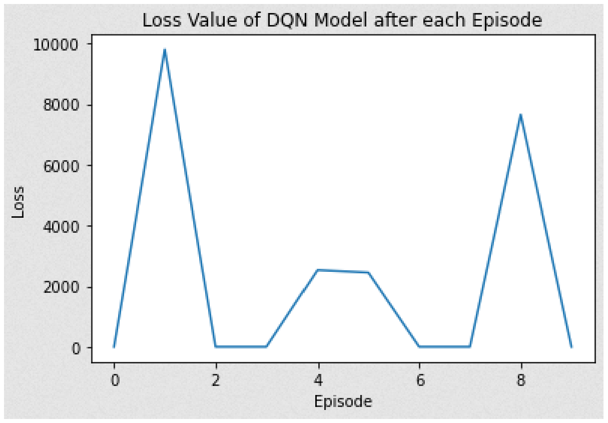


Figure 5: Loss of DQN at Gamma = 0.8 and learning rate = 0.01

1.2 Policy Gradient

1.2.1 Introduction

Policy Gradient is another reinforcement learning algorithm that directly optimizes the policy in the policy space. This policy space is created via a probability distribution. The advantage of Policy

Gradient over DQN-Learning is that Policy Gradient doesn't rely on value-based equations. Also Policy Gradient algorithms don't require a discrete action space, that is to say that in terms of the Lunar Landing game, we don't need to only push left right or up but rather any combination and at any "strength".

1.3 Key Equation

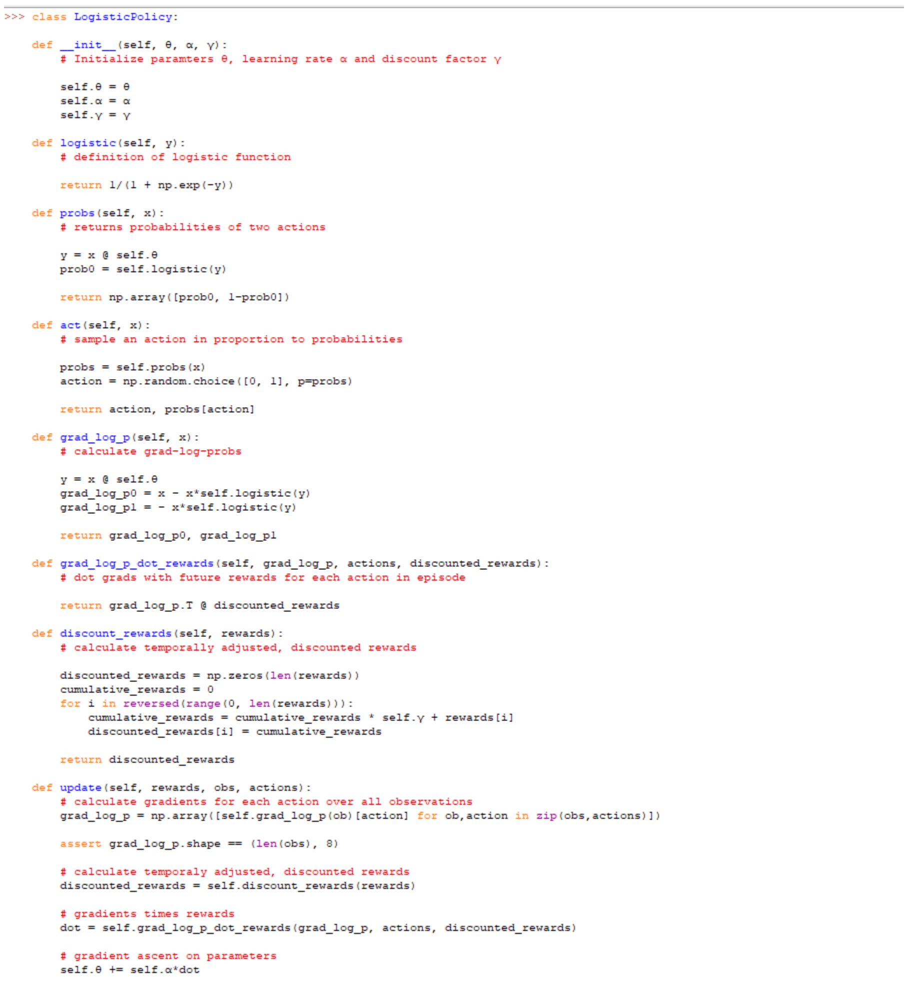
The policy gradient descent is:

$$\theta_{k+1} = \theta k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{2}$$

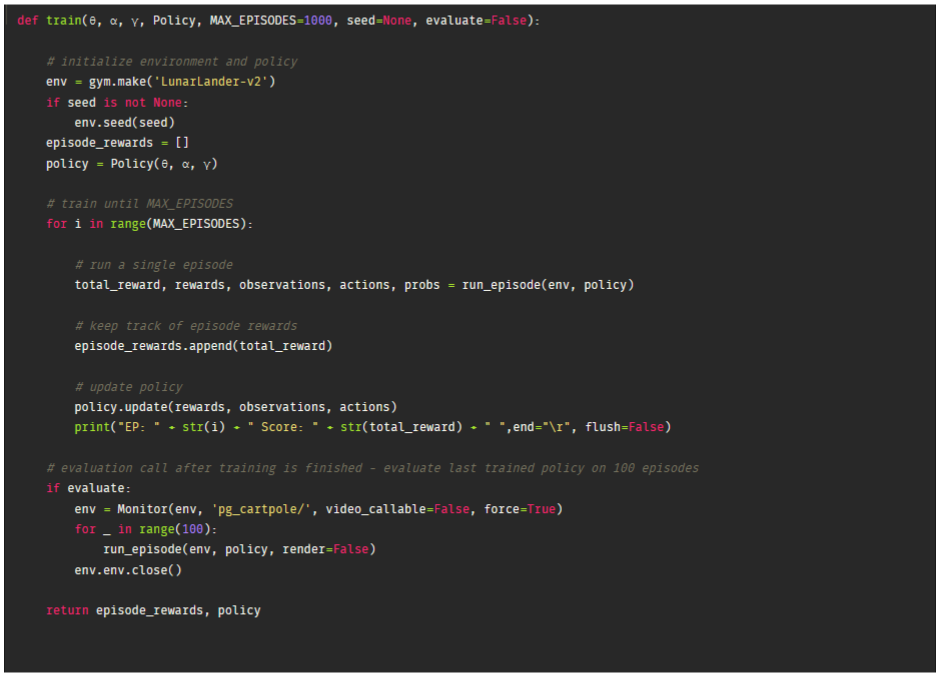
∇ term can be calculated using any policy gradient algorithms such as Vanilia Policy Gradient and TRPO. We will use the simplest policy gradient Which is as follows:

$$\nabla_\theta J(\pi_\theta) = E_{\pi_\theta}(\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(\alpha_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) \tag{3}$$

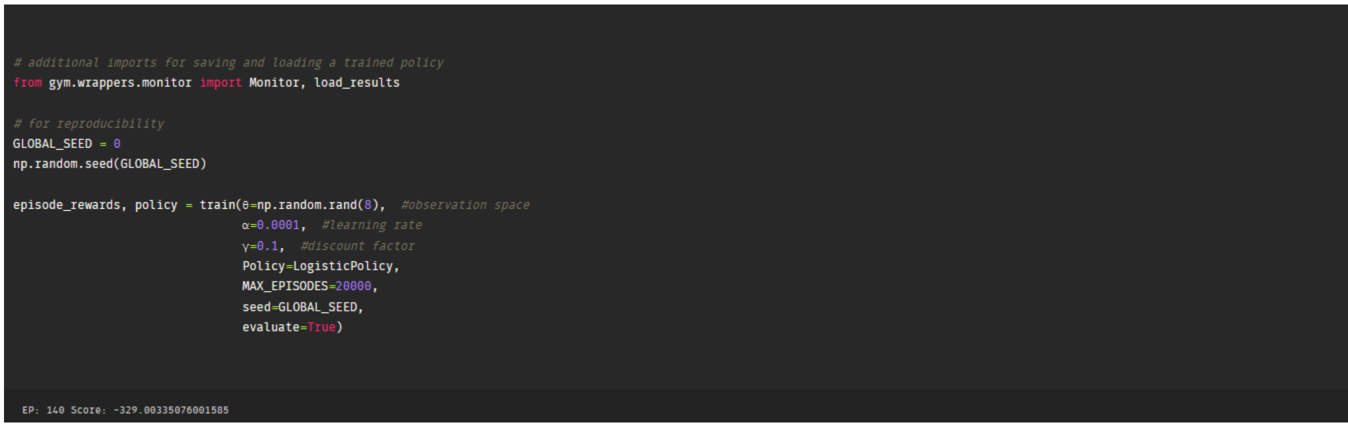
1.3.1 Code



```
>>> class LogisticPolicy:

    def __init__(self, θ, α, γ):
        # Initialize paramters θ, learning rate α and discount factor γ

        self.θ = θ
        self.α = α
        self.γ = γ

    def logistic(self, y):
        # definition of logistic function

        return 1/(1 + np.exp(-y))

    def probs(self, x):
        # returns probabilities of two actions

        y = x @ self.θ
        prob0 = self.logistic(y)

        return np.array([prob0, 1-prob0])

    def act(self, x):
        # sample an action in proportion to probabilities

        probs = self.probs(x)
        action = np.random.choice([0, 1], p=probs)

        return action, probs[action]

    def grad_log_p(self, x):
        # calculate grad-log-probs

        y = x @ self.θ
        grad_log_p0 = x - x*self.logistic(y)
        grad_log_p1 = - x*self.logistic(y)

        return grad_log_p0, grad_log_p1

    def grad_log_p_dot_rewards(self, grad_log_p, actions, discounted_rewards):
        # dot grads with future rewards for each action in episode

        return grad_log_p.T @ discounted_rewards

    def discount_rewards(self, rewards):
        # calculate temporally adjusted, discounted rewards

        discounted_rewards = np.zeros(len(rewards))
        cumulative_rewards = 0
        for i in reversed(range(0, len(rewards))):
            cumulative_rewards = cumulative_rewards * self.γ + rewards[i]
            discounted_rewards[i] = cumulative_rewards

        return discounted_rewards

    def update(self, rewards, obs, actions):
        # calculate gradients for each action over all observations
        grad_log_p = np.array([self.grad_log_p(ob)[action] for ob,action in zip(obs,actions)])

        assert grad_log_p.shape == (len(obs), 8)

        # calculate temporaly adjusted, discounted rewards
        discounted_rewards = self.discount_rewards(rewards)

        # gradients times rewards
        dot = self.grad_log_p_dot_rewards(grad_log_p, actions, discounted_rewards)

        # gradient ascent on parameters
        self.θ += self.α*dot
```



```python
def train(θ, α, γ, Policy, MAX_EPISODES=1000, seed=None, evaluate=False):

    # initialize environment and policy
    env = gym.make('LunarLander-v2')
    if seed is not None:
        env.seed(seed)
    episode_rewards = []
    policy = Policy(θ, α, γ)

    # train until MAX_EPISODES
    for i in range(MAX_EPISODES):

        # run a single episode
        total_reward, rewards, observations, actions, probs = run_episode(env, policy)

        # keep track of episode rewards
        episode_rewards.append(total_reward)

        # update policy
        policy.update(rewards, observations, actions)
        print("EP: " + str(i) + " Score: " + str(total_reward) + " ",end="\r", flush=False)

    # evaluation call after training is finished - evaluate last trained policy on 100 episodes
    if evaluate:
        env = Monitor(env, 'pg_cartpole/', video_callable=False, force=True)
        for _ in range(100):
            run_episode(env, policy, render=False)
        env.env.close()

    return episode_rewards, policy
```



```python
# additional imports for saving and loading a trained policy
from gym.wrappers.monitor import Monitor, load_results

# for reproducibility
GLOBAL_SEED = 0
np.random.seed(GLOBAL_SEED)

episode_rewards, policy = train(θ=np.random.rand(8),  #observation space
                                α=0.0001,  #learning rate
                                γ=0.1,  #discount factor
                                Policy=LogisticPolicy,
                                MAX_EPISODES=20000,
                                seed=GLOBAL_SEED,
                                evaluate=True)
```

```
EP: 140 Score: -329.00335076001585
```

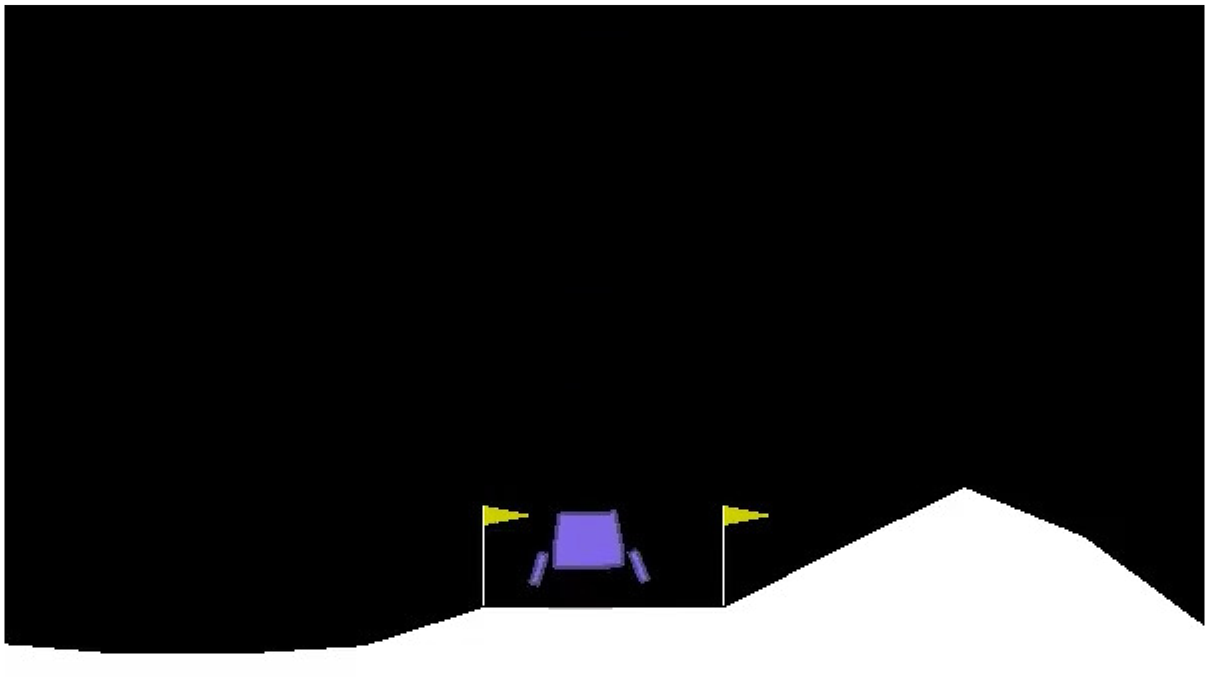


Figure 6: The problem here is that it keeps predicting the same action (always left, or always nothing)so maybe the model doesn't update, but looking at the code it should be updating

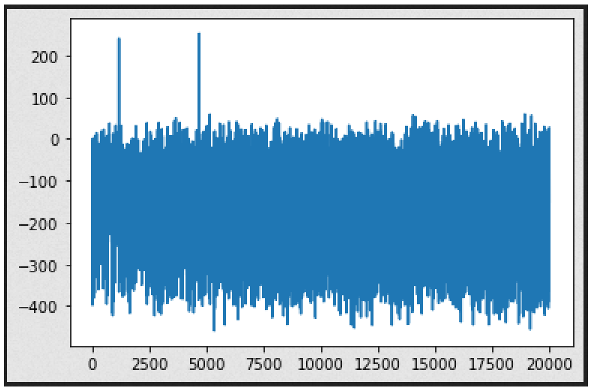


Figure 7: Policy Learning module with Learning Rate = 0.001 and Gamma = 0.5

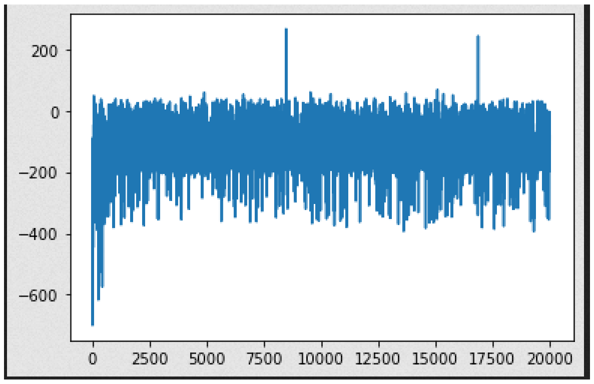


Figure 8: Policy Learning module with Learning Rate = 0.01 and Gamma = 0.95

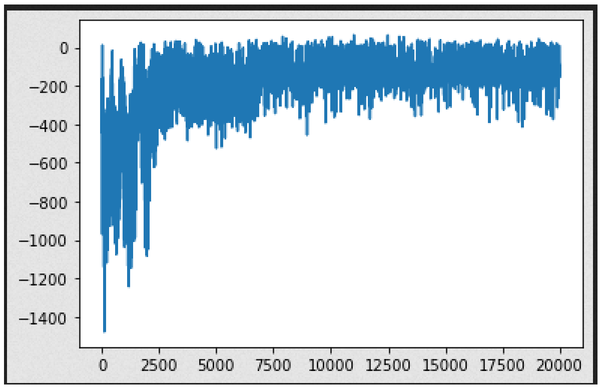


Figure 9: Policy Learning module with Learning Rate = 0.05 and Gamma = 0.999

1.3.2 Conclusion

The models doesn't seem to respect velocity in updating itself, it keeps trying to go as fast as possible in the zone, which leads to a crash. I'm not sure what's going on with the prediction, but it doesn't seem to value slowing down at all.

2 Homework 1: Neural Networks: Fully connected, CNN and ResNet

The following is an example of the homework.

2.1 Fully Connected Layer

Introduce the algorithm, no more than 100 words. Answer the following questions: What is this algorithm? What is it used for? e.g. classification, generation, translation, etc. What is the relation compare to the previous method? (e.g. in FC layer, answer the difference to linear regression).

The fully connected layer is when the neural network has converged to a single vector using the function softmax. Softmax translates the matrix in the previous layer into probabilities of the category. It is generated by re-scaling the input into a range [0,1] with the sum of the inputs equaling to 1. Linear regression still maintains features of the original input. The fully connect layer contains only probabilities of the input being in n category.

2.1.1 Kernel Formular

Write kernal formulars. e.g. In the FC layer, write how to calculate forward pass, how to calculate the gradient, and how to do updates. Typically 3-5 equations for each algorithm is fine.

First obtain the input layer in matrix form and how many inputs. (4,3,28,28) in the example represented 4 inputs, 3 channels (RGB) with each input being a 28 x 28 pixel image. Then optionally perform max pooling. Then change the dimension of the input layer by sub-sampling the matrix via linear or convolution layers. After the final sub-sample layer, change the matrix such that the dimensions are collapsed to a single vector. In the (4,3,28,28) example, we should collapse it to (4,x) where x is the sub-sample dimensions. We don't collapse 4 because 4 is the batch size. E.g if (4,3,10,10) was the final sub-sample layer, we should collapse this layer to (4,300). Perform non-linear transformation on this FC layer, such as ReLU, TanH, Leaky ReLu. For the gradient, we go backwards from the output layer towards the input layer. We first calculate

$$Cost = (predicted - observed)^2 \tag{4}$$

We take the partial derivative of the output layer w.r.t each element/weight for a single training example. In order to calculate the partial derivative of the 2nd to last layer however, you'd need to find the partial derivative of the 3rd to last layer and so on. This is why it's called back propagation. We create a for loop to repeat this gradient descent over n amount of times, conventionally it is called epoch.

2.1.2 Algorithm flow

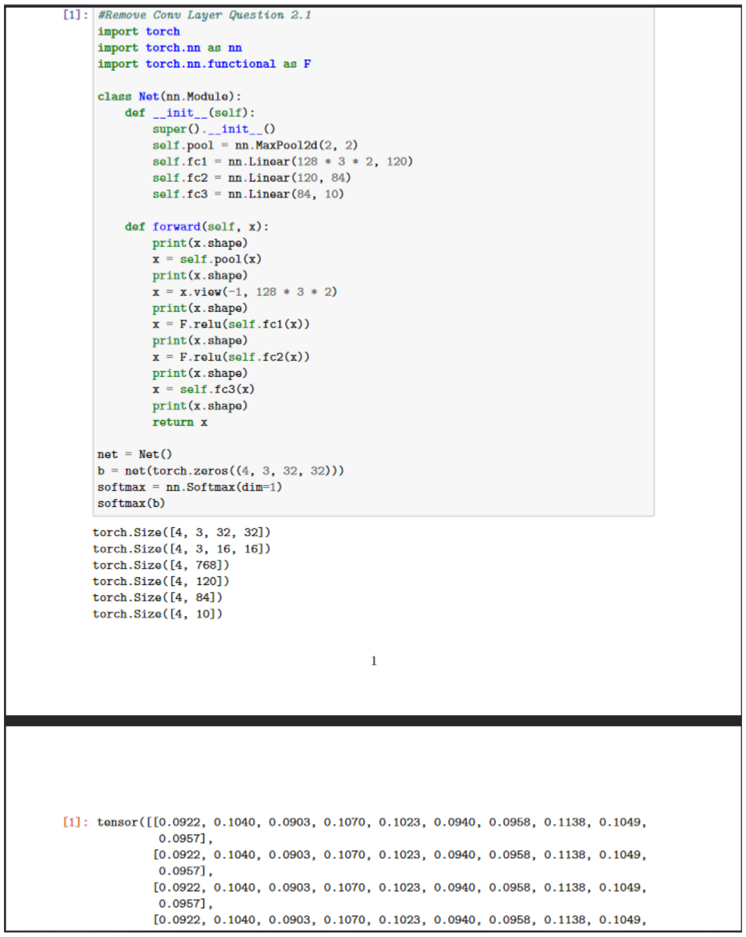
Write the algorithm flow of the algorithm in the following form. Feel free to copy-paste the existing one on the original paper. Typically 10 lines are good.

Algorithm 1: How to write algorithms

```
Result: Item(max) is the predicted category
initialization;
while Sub-sampling do
    Take first matrix m x n, and multiply by n x a;
    if last sub-sample layer then
        net = Net(); Store neural network into a module that you can call functions from.
        softmax = nn.Softmax(dim=1); Squishes sub-sample layer to positive values that sum up
          to 1
        softmax(input);
        torch.max(softmax(input);
    else
        Multiply matrix n x a by a x b Perform non-linear transformation of the multiplied matrix
          Next sub-sample layer = Relu(a x b) Matrix a x b in this case;
    end
end
```

2.1.3 Experiments

How to run Write the specific command on how to train your code and how to test your code. If jupyter notebook is used, specify which block is the entry for train and test.



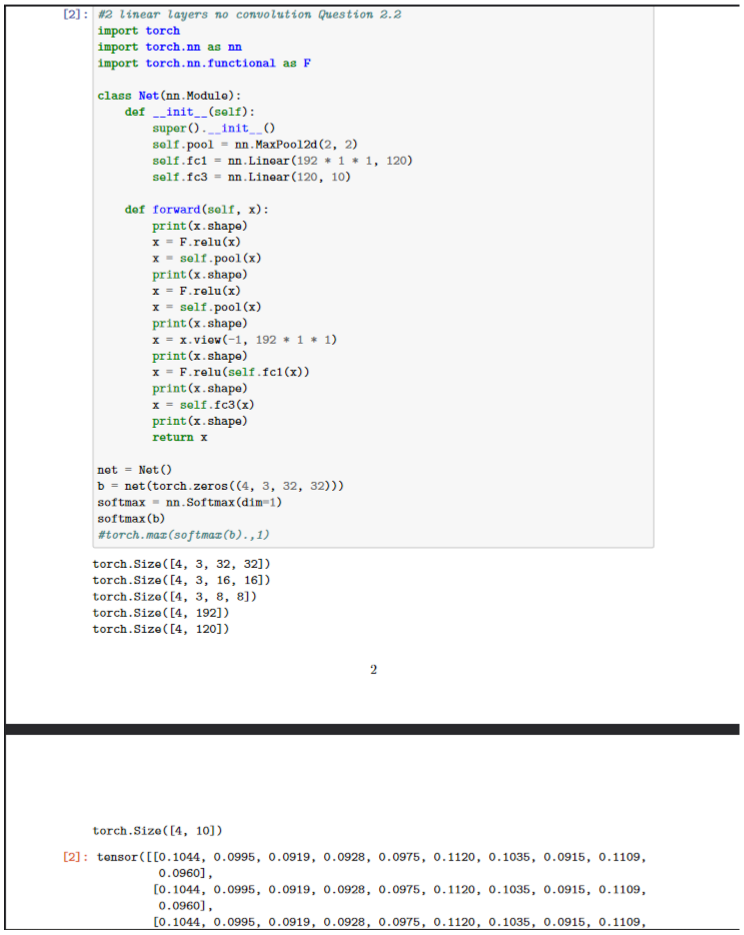
```
[1]: #Remove Conv Layer Question 2.1
     import torch
     import torch.nn as nn
     import torch.nn.functional as F

     class Net(nn.Module):
         def __init__(self):
             super().__init__()
             self.pool = nn.MaxPool2d(2, 2)
             self.fc1 = nn.Linear(128 * 3 * 2, 120)
             self.fc2 = nn.Linear(120, 84)
             self.fc3 = nn.Linear(84, 10)

         def forward(self, x):
             print(x.shape)
             x = self.pool(x)
             print(x.shape)
             x = x.view(-1, 128 * 3 * 2)
             print(x.shape)
             x = F.relu(self.fc1(x))
             print(x.shape)
             x = F.relu(self.fc2(x))
             print(x.shape)
             x = self.fc3(x)
             print(x.shape)
             return x

     net = Net()
     b = net(torch.zeros((4, 3, 32, 32)))
     softmax = nn.Softmax(dim=1)
     softmax(b)
```

```
torch.Size([4, 3, 32, 32])
torch.Size([4, 3, 16, 16])
torch.Size([4, 768])
torch.Size([4, 120])
torch.Size([4, 84])
torch.Size([4, 10])
```

1

```
[1]: tensor([[0.0922, 0.1040, 0.0903, 0.1070, 0.1023, 0.0940, 0.0958, 0.1138, 0.1049,
              0.0957],
             [0.0922, 0.1040, 0.0903, 0.1070, 0.1023, 0.0940, 0.0958, 0.1138, 0.1049,
              0.0957],
             [0.0922, 0.1040, 0.0903, 0.1070, 0.1023, 0.0940, 0.0958, 0.1138, 0.1049,
              0.0957],
             [0.0922, 0.1040, 0.0903, 0.1070, 0.1023, 0.0940, 0.0958, 0.1138, 0.1049,
```

Figure 10: Question 2.1



```
[2]: #2 linear layers no convolution Question 2.2
     import torch
     import torch.nn as nn
     import torch.nn.functional as F

     class Net(nn.Module):
         def __init__(self):
             super().__init__()
             self.pool = nn.MaxPool2d(2, 2)
             self.fc1 = nn.Linear(192 * 1 * 1, 120)
             self.fc3 = nn.Linear(120, 10)

         def forward(self, x):
             print(x.shape)
             x = F.relu(x)
             x = self.pool(x)
             print(x.shape)
             x = F.relu(x)
             x = self.pool(x)
             print(x.shape)
             x = x.view(-1, 192 * 1 * 1)
             print(x.shape)
             x = F.relu(self.fc1(x))
             print(x.shape)
             x = self.fc3(x)
             print(x.shape)
             return x

     net = Net()
     b = net(torch.zeros((4, 3, 32, 32)))
     softmax = nn.Softmax(dim=1)
     softmax(b)
     #torch.max(softmax(b).,1)
```

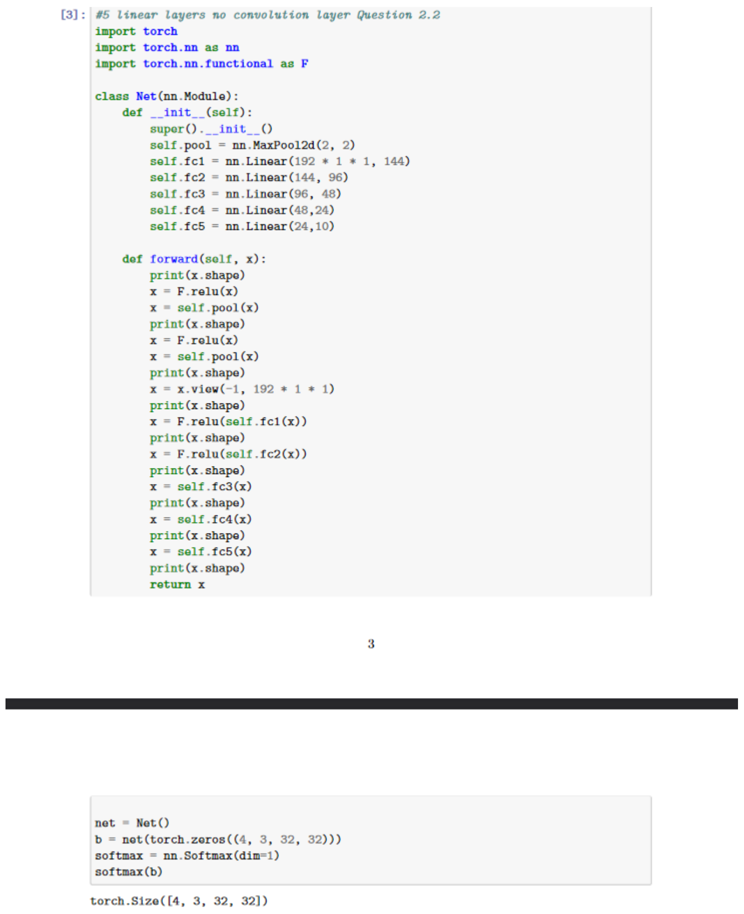
```
torch.Size([4, 3, 32, 32])
torch.Size([4, 3, 16, 16])
torch.Size([4, 3, 8, 8])
torch.Size([4, 192])
torch.Size([4, 120])
```

2

```
torch.Size([4, 10])
```

```
[2]: tensor([[0.1044, 0.0995, 0.0919, 0.0928, 0.0975, 0.1120, 0.1035, 0.0915, 0.1109,
              0.0960],
             [0.1044, 0.0995, 0.0919, 0.0928, 0.0975, 0.1120, 0.1035, 0.0915, 0.1109,
              0.0960],
             [0.1044, 0.0995, 0.0919, 0.0928, 0.0975, 0.1120, 0.1035, 0.0915, 0.1109,
```

Figure 11: Question 2.2



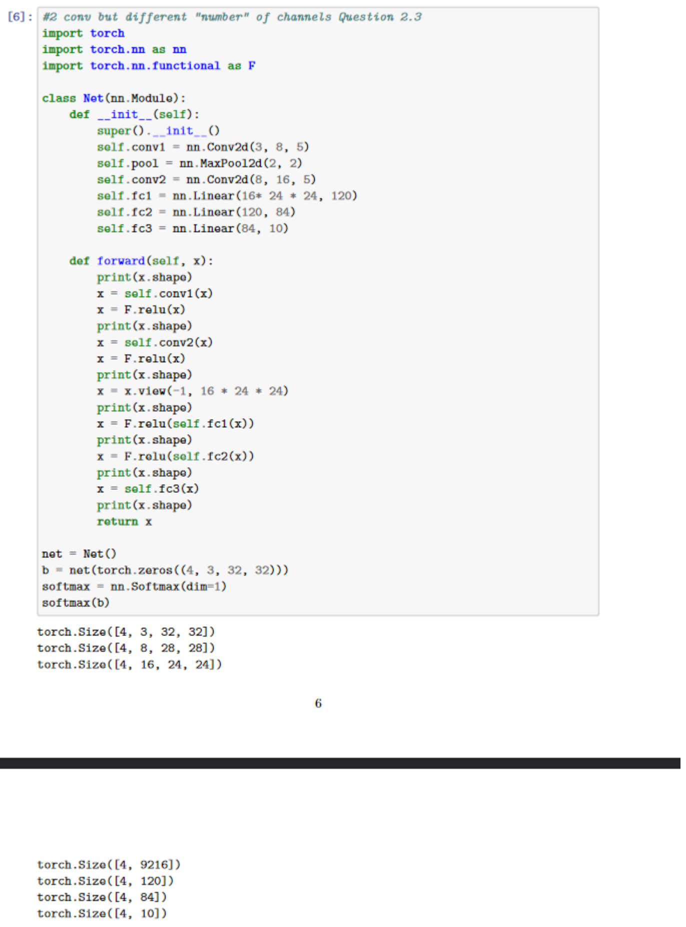
```
[3]: #5 linear layers no convolution layer Question 2.2
     import torch
     import torch.nn as nn
     import torch.nn.functional as F

     class Net(nn.Module):
         def __init__(self):
             super().__init__()
             self.pool = nn.MaxPool2d(2, 2)
             self.fc1 = nn.Linear(192 * 1 * 1, 144)
             self.fc2 = nn.Linear(144, 96)
             self.fc3 = nn.Linear(96, 48)
             self.fc4 = nn.Linear(48,24)
             self.fc5 = nn.Linear(24,10)

         def forward(self, x):
             print(x.shape)
             x = F.relu(x)
             x = self.pool(x)
             print(x.shape)
             x = F.relu(x)
             x = self.pool(x)
             print(x.shape)
             x = x.view(-1, 192 * 1 * 1)
             print(x.shape)
             x = F.relu(self.fc1(x))
             print(x.shape)
             x = F.relu(self.fc2(x))
             print(x.shape)
             x = self.fc3(x)
             print(x.shape)
             x = self.fc4(x)
             print(x.shape)
             x = self.fc5(x)
             print(x.shape)
             return x
```

3

```
net = Net()
b = net(torch.zeros((4, 3, 32, 32)))
softmax = nn.Softmax(dim=1)
softmax(b)
```

```
torch.Size([4, 3, 32, 32])
```

Figure 12: Question 2.2 5 Layers



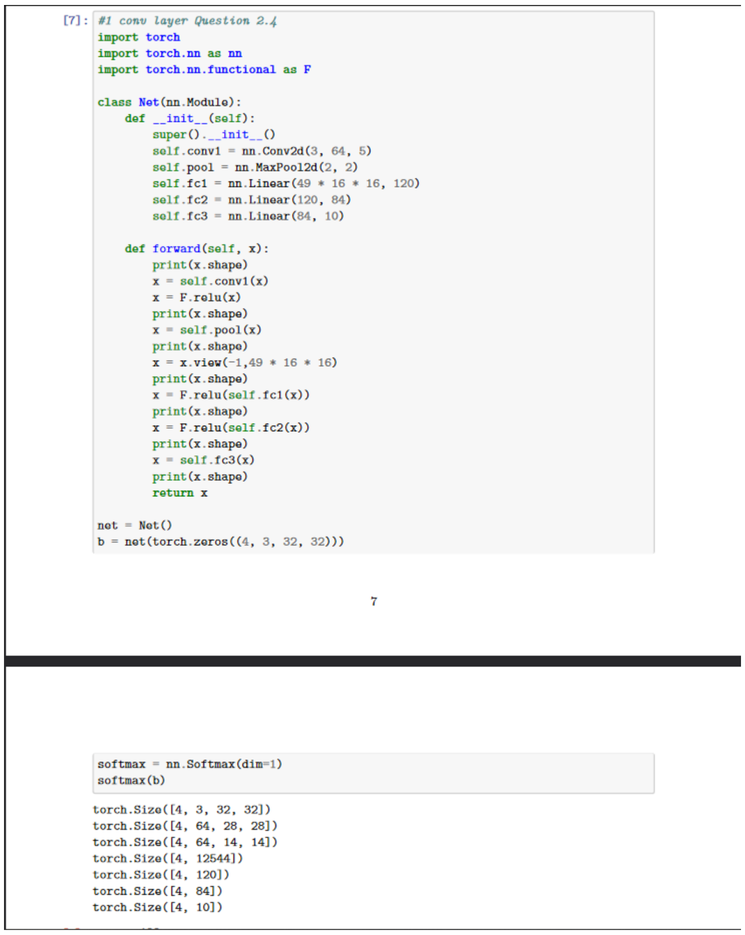
```
[6]: #2 conv but different "number" of channels Question 2.3
     import torch
     import torch.nn as nn
     import torch.nn.functional as F

     class Net(nn.Module):
         def __init__(self):
             super().__init__()
             self.conv1 = nn.Conv2d(3, 8, 5)
             self.pool = nn.MaxPool2d(2, 2)
             self.conv2 = nn.Conv2d(8, 16, 5)
             self.fc1 = nn.Linear(16* 24 * 24, 120)
             self.fc2 = nn.Linear(120, 84)
             self.fc3 = nn.Linear(84, 10)

         def forward(self, x):
             print(x.shape)
             x = self.conv1(x)
             x = F.relu(x)
             print(x.shape)
             x = self.conv2(x)
             x = F.relu(x)
             print(x.shape)
             x = x.view(-1, 16 * 24 * 24)
             print(x.shape)
             x = F.relu(self.fc1(x))
             print(x.shape)
             x = F.relu(self.fc2(x))
             print(x.shape)
             x = self.fc3(x)
             print(x.shape)
             return x

     net = Net()
     b = net(torch.zeros((4, 3, 32, 32)))
     softmax = nn.Softmax(dim=1)
     softmax(b)
```

```
torch.Size([4, 3, 32, 32])
torch.Size([4, 8, 28, 28])
torch.Size([4, 16, 24, 24])
```

6

```
torch.Size([4, 9216])
torch.Size([4, 120])
torch.Size([4, 84])
torch.Size([4, 10])
```

Figure 13: Question 2.3



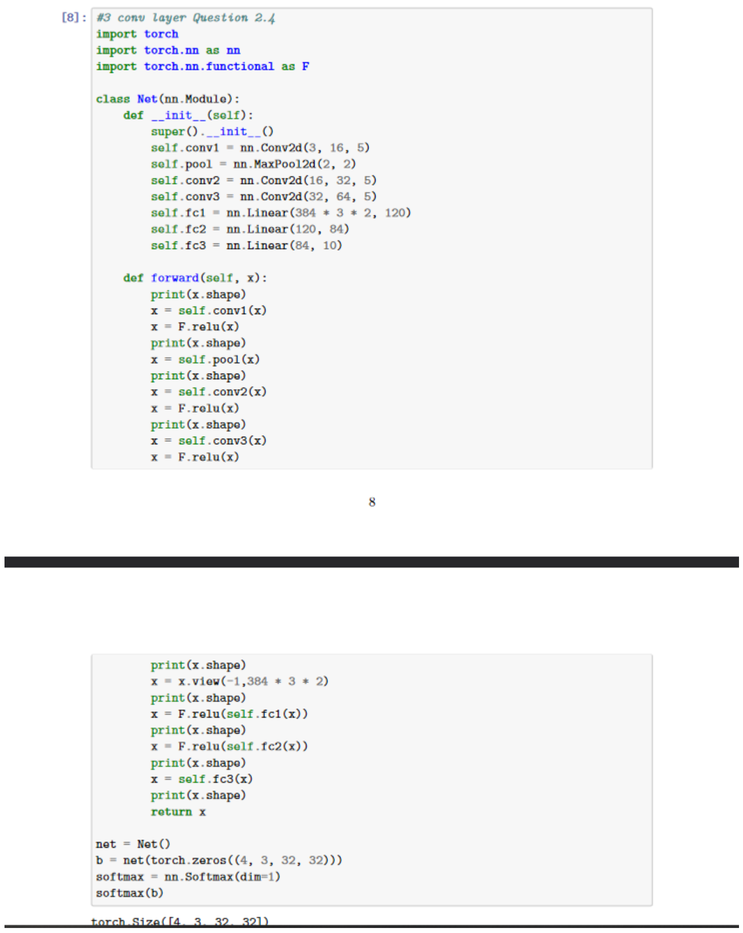
```
[7]: #1 conv layer Question 2.4
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(49 * 16 * 16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        print(x.shape)
        x = self.conv1(x)
        x = F.relu(x)
        print(x.shape)
        x = self.pool(x)
        print(x.shape)
        x = x.view(-1,49 * 16 * 16)
        print(x.shape)
        x = F.relu(self.fc1(x))
        print(x.shape)
        x = F.relu(self.fc2(x))
        print(x.shape)
        x = self.fc3(x)
        print(x.shape)
        return x

net = Net()
b = net(torch.zeros((4, 3, 32, 32)))
```

7

```
softmax = nn.Softmax(dim=1)
softmax(b)
```

```
torch.Size([4, 3, 32, 32])
torch.Size([4, 64, 28, 28])
torch.Size([4, 64, 14, 14])
torch.Size([4, 12544])
torch.Size([4, 120])
torch.Size([4, 84])
torch.Size([4, 10])
```

Figure 14: Question 2.4 1 Layer



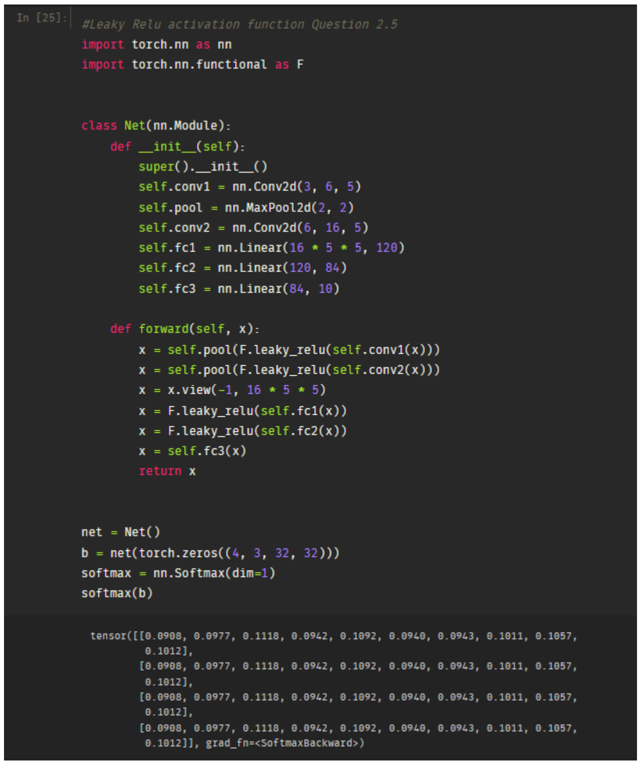
```
[8]: #3 conv layer Question 2.4
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.conv3 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(384 * 3 * 2, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        print(x.shape)
        x = self.conv1(x)
        x = F.relu(x)
        print(x.shape)
        x = self.pool(x)
        print(x.shape)
        x = self.conv2(x)
        x = F.relu(x)
        print(x.shape)
        x = self.conv3(x)
        x = F.relu(x)
```

8

```
        print(x.shape)
        x = x.view(-1,384 * 3 * 2)
        print(x.shape)
        x = F.relu(self.fc1(x))
        print(x.shape)
        x = F.relu(self.fc2(x))
        print(x.shape)
        x = self.fc3(x)
        print(x.shape)
        return x

net = Net()
b = net(torch.zeros((4, 3, 32, 32)))
softmax = nn.Softmax(dim=1)
softmax(b)
```

```
torch.Size([4, 3, 32, 32])
```

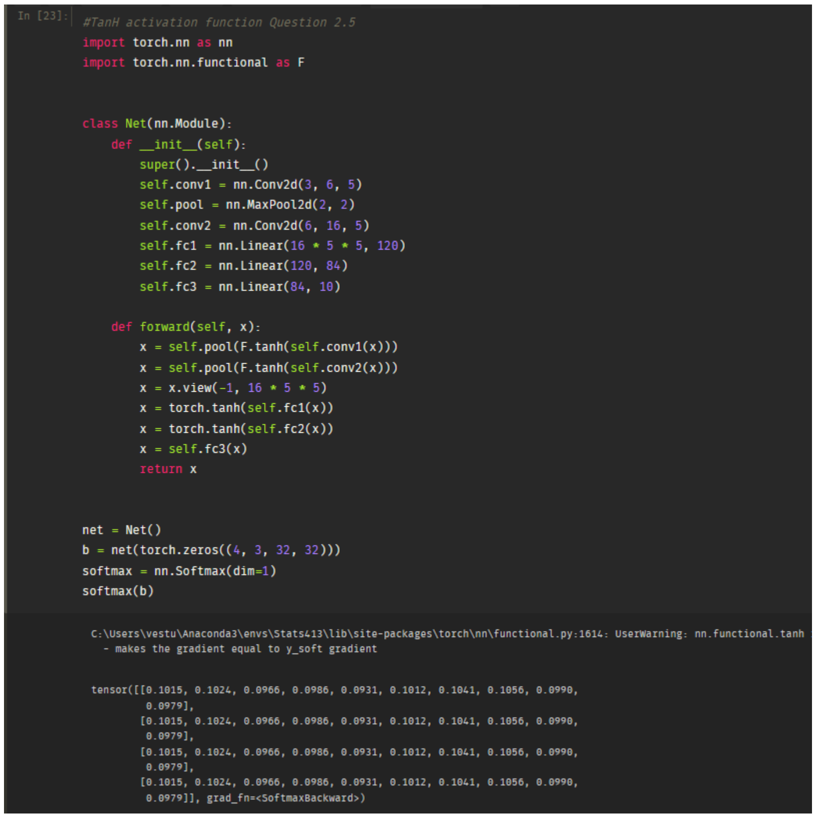
Figure 15: Question 2.4 3 Layers



```
In [25]: #Leaky Relu activation function Question 2.5
         import torch.nn as nn
         import torch.nn.functional as F


         class Net(nn.Module):
             def __init__(self):
                 super().__init__()
                 self.conv1 = nn.Conv2d(3, 6, 5)
                 self.pool = nn.MaxPool2d(2, 2)
                 self.conv2 = nn.Conv2d(6, 16, 5)
                 self.fc1 = nn.Linear(16 * 5 * 5, 120)
                 self.fc2 = nn.Linear(120, 84)
                 self.fc3 = nn.Linear(84, 10)

             def forward(self, x):
                 x = self.pool(F.leaky_relu(self.conv1(x)))
                 x = self.pool(F.leaky_relu(self.conv2(x)))
                 x = x.view(-1, 16 * 5 * 5)
                 x = F.leaky_relu(self.fc1(x))
                 x = F.leaky_relu(self.fc2(x))
                 x = self.fc3(x)
                 return x


         net = Net()
         b = net(torch.zeros((4, 3, 32, 32)))
         softmax = nn.Softmax(dim=1)
         softmax(b)
```

```
tensor([[0.0908, 0.0977, 0.1118, 0.0942, 0.1092, 0.0940, 0.0943, 0.1011, 0.1057,
         0.1012],
        [0.0908, 0.0977, 0.1118, 0.0942, 0.1092, 0.0940, 0.0943, 0.1011, 0.1057,
         0.1012],
        [0.0908, 0.0977, 0.1118, 0.0942, 0.1092, 0.0940, 0.0943, 0.1011, 0.1057,
         0.1012],
        [0.0908, 0.0977, 0.1118, 0.0942, 0.1092, 0.0940, 0.0943, 0.1011, 0.1057,
         0.1012]], grad_fn=<SoftmaxBackward>)
```

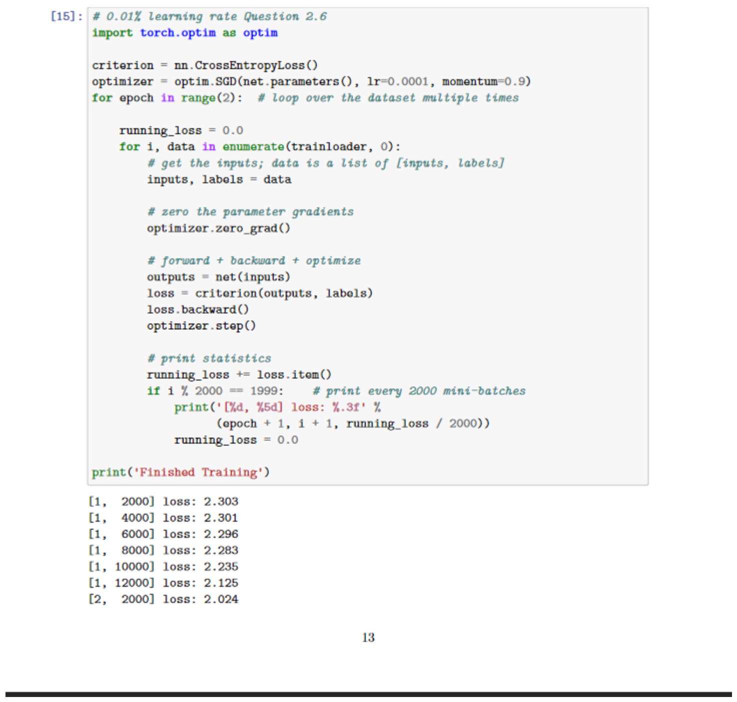
Figure 16: Question 2.5 Leaky ReLu



```
In [23]:
#TanH activation function Question 2.5
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.tanh(self.conv1(x)))
        x = self.pool(F.tanh(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
b = net(torch.zeros((4, 3, 32, 32)))
softmax = nn.Softmax(dim=1)
softmax(b)
```

```
C:\Users\vestu\Anaconda3\envs\Stats413\lib\site-packages\torch\nn\functional.py:1614: UserWarning: nn.functional.tanh
  - makes the gradient equal to y_soft gradient


tensor([[0.1015, 0.1024, 0.0966, 0.0986, 0.0931, 0.1012, 0.1041, 0.1056, 0.0990,
         0.0979],
        [0.1015, 0.1024, 0.0966, 0.0986, 0.0931, 0.1012, 0.1041, 0.1056, 0.0990,
         0.0979],
        [0.1015, 0.1024, 0.0966, 0.0986, 0.0931, 0.1012, 0.1041, 0.1056, 0.0990,
         0.0979],
        [0.1015, 0.1024, 0.0966, 0.0986, 0.0931, 0.1012, 0.1041, 0.1056, 0.0990,
         0.0979]], grad_fn=<SoftmaxBackward>)
```

Figure 17: Question 2.5 TanH

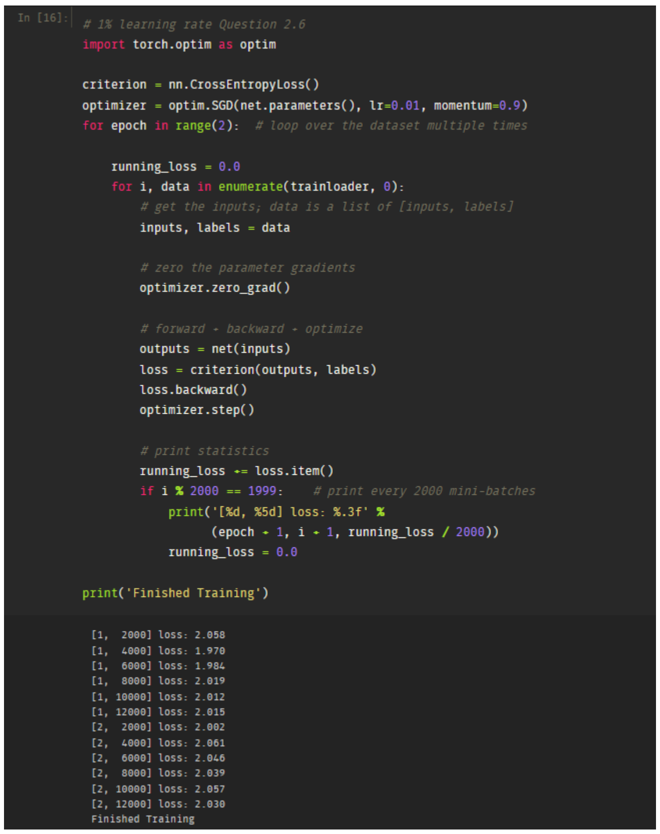


```
[15]: # 0.01% learning rate Question 2.6
      import torch.optim as optim

      criterion = nn.CrossEntropyLoss()
      optimizer = optim.SGD(net.parameters(), lr=0.0001, momentum=0.9)
      for epoch in range(2):  # loop over the dataset multiple times

          running_loss = 0.0
          for i, data in enumerate(trainloader, 0):
              # get the inputs; data is a list of [inputs, labels]
              inputs, labels = data

              # zero the parameter gradients
              optimizer.zero_grad()

              # forward + backward + optimize
              outputs = net(inputs)
              loss = criterion(outputs, labels)
              loss.backward()
              optimizer.step()

              # print statistics
              running_loss += loss.item()
              if i % 2000 == 1999:    # print every 2000 mini-batches
                  print('[%d, %5d] loss: %.3f' %
                        (epoch + 1, i + 1, running_loss / 2000))
                  running_loss = 0.0

      print('Finished Training')
```

```
[1,  2000] loss: 2.303
[1,  4000] loss: 2.301
[1,  6000] loss: 2.296
[1,  8000] loss: 2.283
[1, 10000] loss: 2.235
[1, 12000] loss: 2.125
[2,  2000] loss: 2.024
```

13

Figure 18: Question 2.6 Learning Rate = 0.0001



```
In [16]: # 1% learning rate Question 2.6
         import torch.optim as optim

         criterion = nn.CrossEntropyLoss()
         optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
         for epoch in range(2):  # loop over the dataset multiple times

             running_loss = 0.0
             for i, data in enumerate(trainloader, 0):
                 # get the inputs; data is a list of [inputs, labels]
                 inputs, labels = data

                 # zero the parameter gradients
                 optimizer.zero_grad()

                 # forward + backward + optimize
                 outputs = net(inputs)
                 loss = criterion(outputs, labels)
                 loss.backward()
                 optimizer.step()

                 # print statistics
                 running_loss += loss.item()
                 if i % 2000 == 1999:    # print every 2000 mini-batches
                     print('[%d, %5d] loss: %.3f' %
                           (epoch + 1, i + 1, running_loss / 2000))
                     running_loss = 0.0

         print('Finished Training')
```

```
[1,  2000] loss: 2.058
[1,  4000] loss: 1.970
[1,  6000] loss: 1.984
[1,  8000] loss: 2.019
[1, 10000] loss: 2.012
[1, 12000] loss: 2.015
[2,  2000] loss: 2.002
[2,  4000] loss: 2.061
[2,  6000] loss: 2.046
[2,  8000] loss: 2.039
[2, 10000] loss: 2.057
[2, 12000] loss: 2.030
Finished Training
```

Figure 19: Question 2.6 with Learning Rate 1 Percent

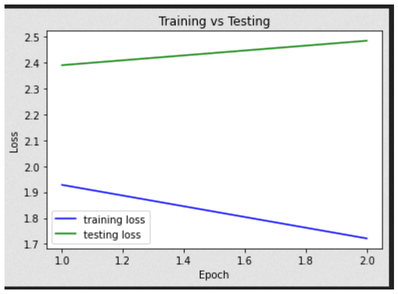


Figure 20: Question 3.1

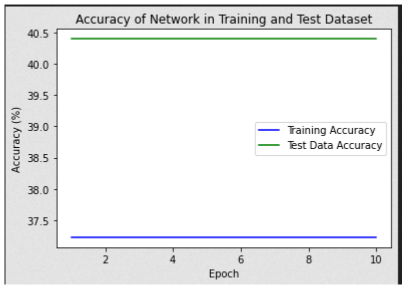
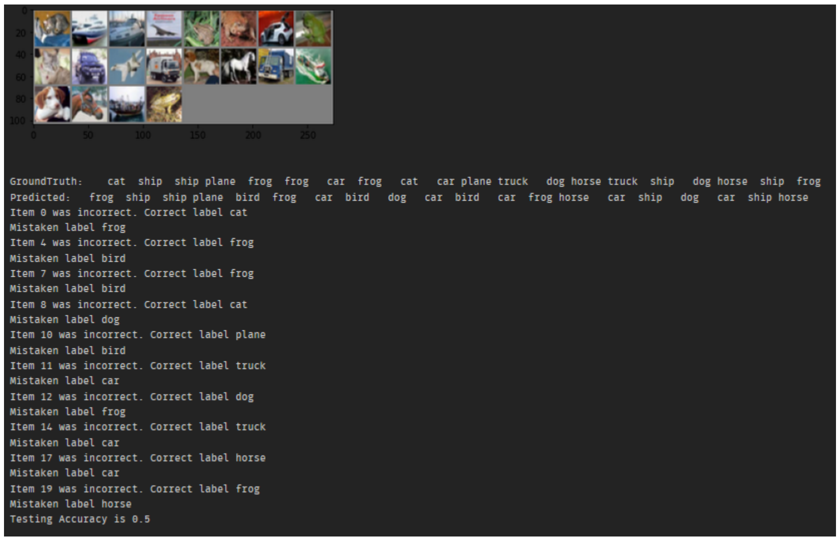


Figure 21: Question 3.2



```
GroundTruth:    cat  ship  ship plane  frog  frog   car  frog   cat   car plane truck   dog horse truck  ship   dog horse  ship  frog
Predicted:   frog  ship  ship plane  bird  frog   car  bird   dog   car  bird   car  frog horse   car  ship   dog   car  ship horse
Item 0 was incorrect. Correct label cat
Mistaken label frog
Item 4 was incorrect. Correct label frog
Mistaken label bird
Item 7 was incorrect. Correct label frog
Mistaken label bird
Item 8 was incorrect. Correct label cat
Mistaken label dog
Item 10 was incorrect. Correct label plane
Mistaken label bird
Item 11 was incorrect. Correct label truck
Mistaken label car
Item 12 was incorrect. Correct label dog
Mistaken label frog
Item 14 was incorrect. Correct label truck
Mistaken label car
Item 17 was incorrect. Correct label horse
Mistaken label car
Item 19 was incorrect. Correct label frog
Mistaken label horse
Testing Accuracy is 0.5
```

Figure 22: Question 3.3

Block 12 contains how the data was split into training and testing sets, and it is modifiable. Block 14 is how you would change parameters or change the amount of layers in the neural network. For block 17, you can adjust the learning rate, how to calculate the gradient of the loss function, and the number of epochs used in testing the training set and test set.

Best results Show the result including but not limited to: (1) training loss/testing loss curve (2) accuracy (3) parameter/network structure you select. (4) Analysis for about 50-100 words.

An example of the network sturcture:

```
latent z (dim = 2) -->
deconv (OutputChannel = 512; Kernel = 4*4; Stride = 2) -->
Batch Normalization -->
```

```
Leaky Relu -->
deconv (OutputChannel = 256; Kernel = 5*5; Stride = 2) -->
Batch Normalization -->
Leaky Relu -->
deconv (OutputChannel = 128; Kernel = 5*5; Stride = 2) -->
Batch Normalization -->
Leaky Relu -->
deconv (OutputChannel = 64; Kernel = 5*5; Stride = 2) -->
Batch Normalization -->
Leaky Relu -->
deconv (OutputChannel = 3; Kernel = 5*5; Stride = 2) -->
Tanh  -->
Output Image (64*64)
```

Include images like this:

Hyper parameter Comparison e.g. change number of layer, change the batch size, change learning rate.

Typically at least 5 different settings. Detailed requirements will be provided on each HW.

Comment on the difference for about 50-200 words.

The lower the amount of layers the bigger in size each remaining layer becomes. A change in test batch size didn't change the accuracy rating which makes sense since the model was trained on the training batch size. Learning rate changes varied greatly in performance, a higher learning rate performed the worst but the lowest learning rate wasn't the best either.

2.1.4 conclusion

Express your experience in running this homework. e.g. It is slow. It requires GPU. It is easy to run / hard to have a good result. No more than 50 words.

The computation was a lot slower than I expected, my CPU is very good, but I wasn't expecting the run time to last 30+ seconds. I am curious as to how to optimize matrix sizes and number of layers because just guessing and checking would be tedious and not efficient.

2.2 Convolution Neural Network (CNN)

Convolution layers are the layers before the fully connected layer and it composes of the average of "nearby" grouping of elements in the matrix. Each convoluted layer is composed of an average of a group of nearby elements in the previous layer. The amount or size of the grouping of elements is called the kernel size. A bigger kernel size would technically dilute the quality of the sub-sample layer because each element in the layer loses it's unique value due to the elements being averaged.

Algorithm 2: How to write algorithms

```
Result: Item(max) is the predicted category
initialization;
while Convolution layers do
    Input Layer (a x m) matrix =Conv2d(m,n,kernel size) This changes the initial matrix to an m
      x n matrix with kernel size elements affecting the input image. The image dimensions are
      also reduced by kernel size.;
    Input Layer = Relu(Input Layer) Apply a non-linear transformation before passing the matrix
      to another layer/output layer.;
    if Last Layer then
        net = Net(); Store neural network into a module that you can call functions from.
        softmax = nn.Softmax(dim=1); Squishes sub-sample layer to positive values that sum up
          to 1
        softmax(input);
        torch.max(softmax(input);
        ;
    else
end
```

3 Homework 2: Residual Network (ResNet) RNN/GRU/LSTM

3.1 ResNet

Introduction Residual Neural Network is a network that takes in shortcuts as part of the input in the hidden layers. This shortcut can just be the identity which means just the input layer or it can be a projection shortcut when moving from layers of different dimensions. ResNet was created to lower the degradation phenomena that occurs with Neural Networks that have many layers. The more layers a Neural Network has, the more error on the training data. To alleviate the higher training error rate, but also maintain the benefits of having a deep neural network, a ResNet takes the input from a shallow layer and either duplicates it (i.e identity shortcut) or adds a matrix that follows the output dimensions. If an output layer takes in identity mappings of shallower layers, the training error should not be greater than the shallow layer. This would increase the amount of layers but theoretically maintain the training error.

Key Equation

$$\mathcal{H}(X) = \mathcal{F}(X) + X \tag{5}$$

Let F (X) represent the residual function H(X) represent the target mapping to be fit and X , Y to be the input and output vectors of the layers respectfully. The projection shortcut is defined as,

$$y = \mathcal{F}(x, \{W_i\}) + W_s x \tag{6}$$

with $W_s = 1$ if the shortcut is the identity shortcut. Since H(X) is our target mapping, if we subtract X, which is the input layer, we are left with the "residual" values needed to achieve the target mapping. We define this as F(X). When backpropagating, we would like our residuals from the deep layers to be close to the shallower layer.

Network Structure

ResNet((conv1): Conv2d(3, 64, kernel$_size = (3,3), stride = (1,1), padding = (1,1), bias = False)$
$(bn1) : BatchNorm2d(64, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$(maxpool) : MaxPool2d(kernel_size = 3, stride = 2, padding = 1, dilation = 1, ceil_mode = False)$
$(layer1) : Sequential($
$(0) : Bottleneck($
$(conv1) : Conv2d(64, 64, kernel_size = (1,1), stride = (1,1), bias = False)$

$(bn1) : BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv2) : Conv2d(64, 64, kernel_{s}ize = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv3) : Conv2d(64, 256, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(relu) : ReLU(inplace = True)$
$(downsample) : Sequential($
$(0) : Conv2d(64, 256, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(1) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$)$
$)$
$(1) : Bottleneck($
$(conv1) : Conv2d(256, 64, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv2) : Conv2d(64, 64, kernel_{s}ize = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv3) : Conv2d(64, 256, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(relu) : ReLU(inplace = True)$
$)(2) : Bottleneck($
$(conv1) : Conv2d(256, 64, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv2) : Conv2d(64, 64, kernel_{s}ize = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(64, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv3) : Conv2d(64, 256, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$)$
$(layer2) : Sequential($
$(0) : Bottleneck($
$(conv1) : Conv2d(256, 128, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(128, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv2) : Conv2d(128, 128, kernel_{s}ize = (3, 3), stride = (2, 2), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(128, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv3) : Conv2d(128, 512, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(relu) : ReLU(inplace = True)$
$(downsample) : Sequential($
$(0) : Conv2d(256, 512, kernel_{s}ize = (1, 1), stride = (2, 2), bias = False)$
$(1) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$)$
$)$
$(1) : Bottleneck($
$(conv1) : Conv2d(512, 128, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(128, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv2) : Conv2d(128, 128, kernel_{s}ize = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(128, eps = 1e - 05, momentum = 0.1, affine = True, track_{r}unning_{s}tats = True)$
$(conv3) : Conv2d(128, 512, kernel_{s}ize = (1, 1), stride = (1, 1), bias = False)$

$(bn3) : BatchNorm2d(512, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$(2) : Bottleneck($
$(conv1) : Conv2d(512, 128, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(128, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(128, 128, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(128, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(128, 512, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(512, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$(3) : Bottleneck($
$(conv1) : Conv2d(512, 128, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(128, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(128, 128, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(128, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(128, 512, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(512, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$)$
$(layer3) : Sequential($
$(0) : Bottleneck($
$(conv1) : Conv2d(512, 256, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(256, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(256, 256, kernel_size = (3, 3), stride = (2, 2), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(256, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(256, 1024, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$(downsample) : Sequential($
$(0) : Conv2d(512, 1024, kernel_size = (1, 1), stride = (2, 2), bias = False)$
$(1) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$)$
$)$
$(1) : Bottleneck($
$(conv1) : Conv2d(1024, 256, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(256, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(256, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(256, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(256, 1024, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$(2) : Bottleneck($
$(conv1) : Conv2d(1024, 256, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(256, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(256, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(256, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats =$

$True)$
$(conv3) : Conv2d(256, 1024, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)(3) : Bottleneck($
$(conv1) : Conv2d(1024, 256, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(256, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(256, 1024, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)(4) : Bottleneck($
$(conv1) : Conv2d(1024, 256, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(256, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(256, 1024, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)(5) : Bottleneck((conv1) : Conv2d(1024, 256, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(256, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(256, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(256, 1024, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(1024, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$)$
$(layer4) : Sequential($
$(0) : Bottleneck($
$(conv1) : Conv2d(1024, 512, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(512, 512, kernel_size = (3, 3), stride = (2, 2), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(512, 2048, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(2048, eps = 1e-05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$(downsample) : Sequential($
$(0) : Conv2d(1024, 2048, kernel_size = (1, 1), stride = (2, 2), bias = False)$
$(1) : BatchNorm2d(2048, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$)$
$)$
$(1) : Bottleneck($
$(conv1) : Conv2d(2048, 512, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(512, 512, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(512, 2048, kernel_size = (1, 1), stride = (1, 1), bias = False)$

$(bn3) : BatchNorm2d(2048, eps = 1e{-}05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$(2) : Bottleneck($
$(conv1) : Conv2d(2048, 512, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn1) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv2) : Conv2d(512, 512, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False)$
$(bn2) : BatchNorm2d(512, eps = 1e - 05, momentum = 0.1, affine = True, track_running_stats = True)$
$(conv3) : Conv2d(512, 2048, kernel_size = (1, 1), stride = (1, 1), bias = False)$
$(bn3) : BatchNorm2d(2048, eps = 1e{-}05, momentum = 0.1, affine = True, track_running_stats = True)$
$(relu) : ReLU(inplace = True)$
$)$
$)$
$(avgpool) : AdaptiveAvgPool2d(output_size = (1, 1))$
$(fc) : Linear(in_features = 2048, out_features = 10, bias = True)$
$)$
(7)

The above equation essentially means this RNN contains 4 identical layers that each perform a 8-step layer function n amount of times. This function has 3 convolution layers, 1x1 then, 3x3, then 1x1. The order goes 1x1 -> batch normalization -> ReLu -> 3x3 -> batch normalization -> ReLu -> 1x1 -> batch normalization -> next layer. The 4 big identical layers are to repeat this 8-step layer function n times, with the first layer repeating 3 times, 2nd = 4 times, 3rd = 6 times, 4th = 3 times. At the start of each big layer, downsampling is done to allow the module to add the identity shortcuts into the output values of the convolution layers. Another ReLu is performed during downsampling. The dimensions of the input change at every layer. Then at the end it collapses by taking the average pool, and with the 512 features condenses to 10 which are the labels of the image.

3.2 RNN/GRU/LSTM experiments

Introduction Recurrent Neural Network, Gate Recurrent Unit, and Long-Short Term Memory are all Neural Networks that can take in varying amounts of input at each iteration. This allows for more flexible structures of the network such as single input multiple outputs, multiple inputs single output, etc. All of these networks in their forward pass uses a 0- matrix as the initialization of the hidden matrix. This hidden matrix is then used in tandem with the input matrix to produce the output matrix and is reused for subsequent inputs in that iteration. LSTM has 2 hidden layers whereas RNN and GRU only have 1. GRU has "gates" that act as matrices that tell the input and/or hidden layer what features in the matrix is useful or not. This means that the hidden layer in GRU while it is the same values for all inputs at the same iteration, produces different outputs with the input layer because the gate layer is included in the calculation. GRU's "gates" acts as both the short term and long term layers that LSTM has as a single layer.

Key Equation

$$h_t, O_t = O_1 * O_2 \tag{8}$$

This is both the equation for $h_t and y_t = O_t$
$C_t = C_{t-1} * f + i_{input}(9)$

$$O_1 = \sigma(W_{output_1} * (H_{t-1}, x_t) + bias_{output_1}) \tag{10}$$

$$O_2 = tanh(W_{output_2} * C_t + bias_{output_2}) \tag{11}$$

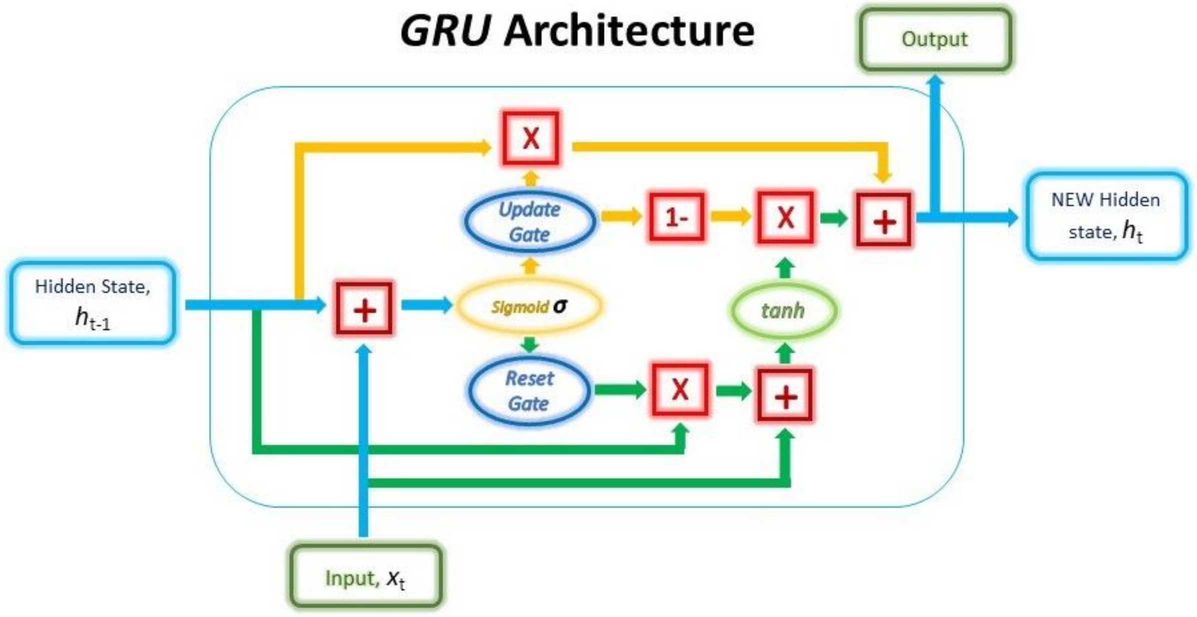


Figure 23: Gate Recurrent Unit Architecture
Input as well as hidden state updates both the gates. These gates then alter which features are accepted in the hidden state and the input state.

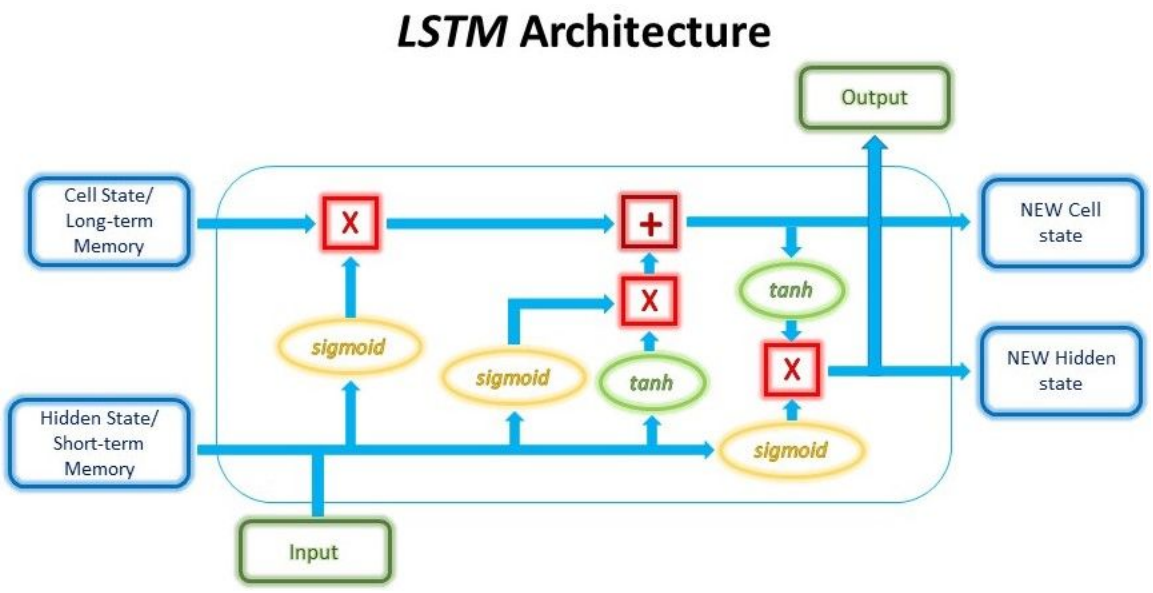


Figure 24: Long-Short Term Memory Architecture
Input and hidden state updates the input gate. The input gate decides which features will be stored in the long-term memory layer it takes in only current input and hidden layer. Forget gate takes current long-term memory and is multiplied by a vector that has short-term and current input. Forget gate decides which features in long-term memory should be kept or discarded.

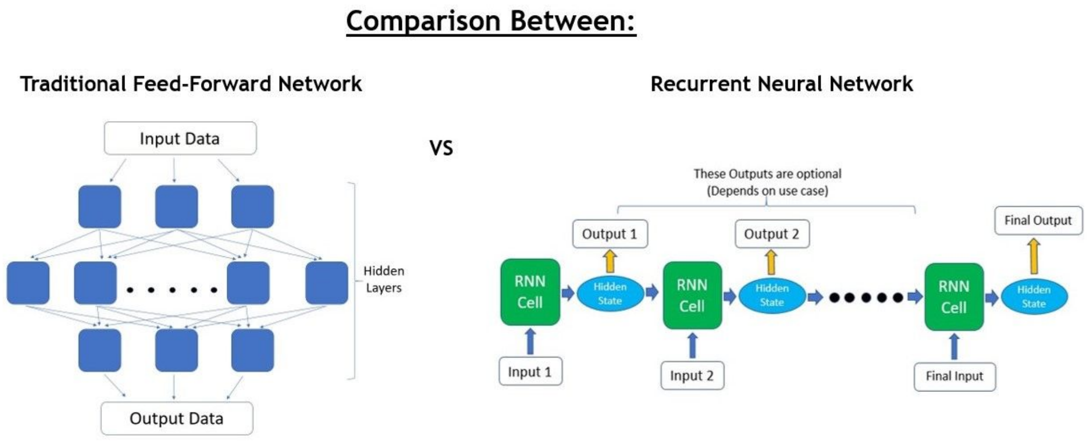


Figure 25: Recurrent Neural Network Architecture
Recurrent Neural Network reuses the same hidden layer for multiple inputs at the same iteration. It takes an initial zero matrix as the first hidden layer and through back-propagation, changes the values of this matrix to optimize accuracy. Like all other RNN, the benefit of this network is that it can take in a varying amount of input and output.

3.3 Experiment

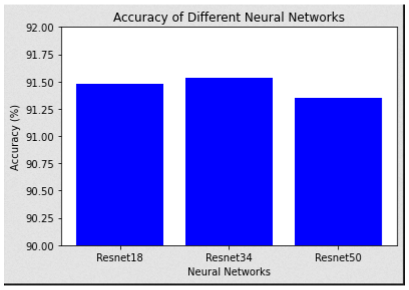


Figure 26: ResNet18, ResNet34, ResNet50 Accuracy Depictions

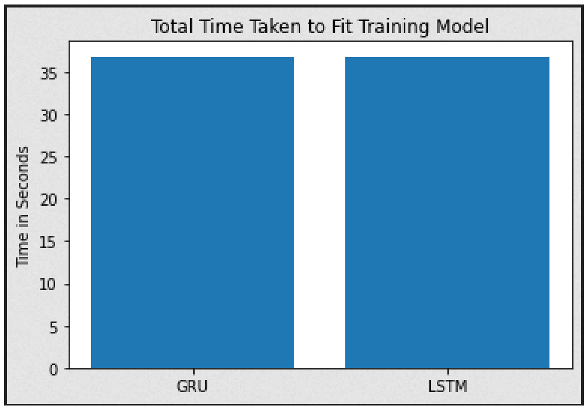


Figure 27: Time taken to fit Training Model

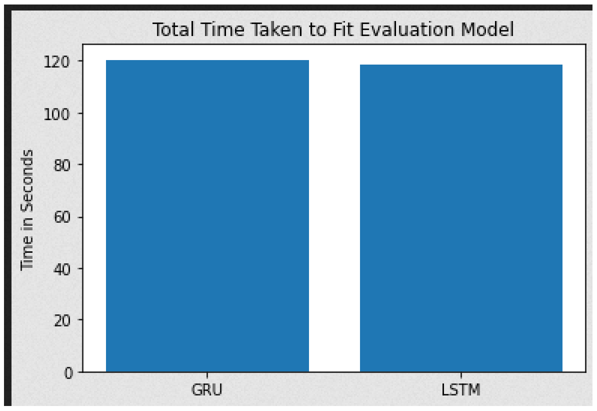


Figure 28: Time taken to fit Evaluation Model

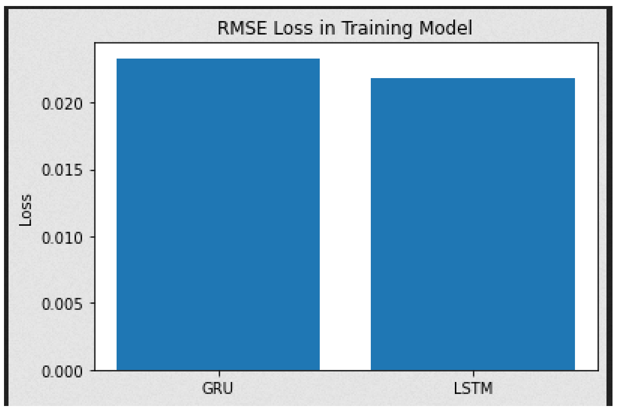


Figure 29: Using Root Mean Squared Error as Loss Function for Training Model

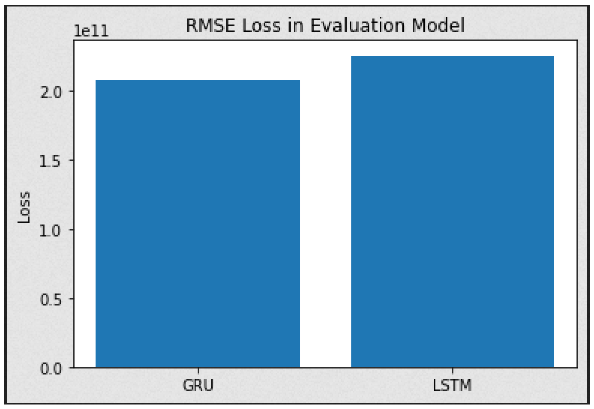


Figure 30: Using Root Mean Squared Error as Loss Function for Evaluation Model

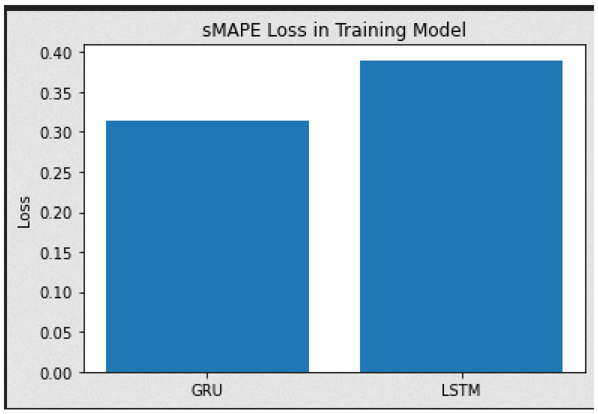


Figure 31: Using Symmetric Mean Absolute Percentage Error as Loss Function for Training Model

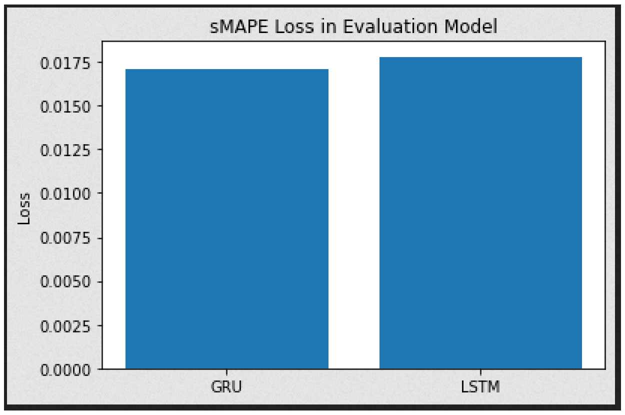


Figure 32: Using Symmetric Mean Absolute Percentage Error as Loss Function for Evaluation Model

4 Homework 3: RNN and Transformers (BERT)

Introduction Bidirectional Encoder Representations from Transformers (BERT) is a neural network focused on Neural Language Processing (NLP). It uses transformer encoder blocks to predict the words in a sentence. The uniqueness of BERT from other models on NLP is that BERT takes in each word of the sentence and draws context before and after the word, hence why it's called Bidirectional. It is also the first unsupervised learning model in NLP.

Key Equations and Structures

4.1 Embedding

Embedding occurs in the input layer to the first encoder layer. It is simply the words transformed into a numerical matrix that encapsulates the uniqueness of the word. In the pretrained model, the dimensions of the embedding is 30,522 times 768. This is the number of words in the vocabulary times hidden size(arbitrary). Each word is embedded as a 1 by 768 vector. The X^{th} row in the vector can be chosen to not be learned in the training process. This is called the "padding" vector and in the BERT model is used to designate the start of a sentence.

4.2 Transformers

Self-attention is the process of using the other words in the sentence as clues to figuring out what the word it's trying to figure out is. An example would be the word "it" in "The animal didn't cross the street because it was too tired." in the BERT model would take all the other words, "The" "Animal" "didn't" "cross", etc. to see if these other words represent "it". The way it does this is by creating 3 sub-vectors of the input vector (the first self-attention layer has the embedded words as the input vector) and each sub-vector(called weights now) is multiplied by the input vector to create 3 different values. These 3 values are called "Query" "Keys" and "Values". The "Query" vector multiplies with the "Keys" vector of every word in the sentence divided by the square root of the dimension size of these sub-vectors. This vector is the score of how much emphasis is placed on other parts of the sentence given this word. Then it goes through a softmax function. This is then multiplied by the "Values" matrix then summed over all the words in the sentence. This vector is then the embedding vector of the word. BERT actually does all the words in parallel meaning that for each word it calculates the softmax values of every other word, sums them up and spits out a matrix containing all of the embedding vectors in the sentence. The first row of this matrix represents the first word, the second row the second word, etc. This is just one self-attention layer with 1 attention head. BERT has 12 self-attention layers and 12 attention heads. Attention head simply means how many multiples of the 3 sub-vectors are there. It's used to try to draw out aspects of the word other than itself. What you do with the concatenated matrix is multiply by another weight matrix designed to condense this matrix back to the size suitable for input in the feed-forward neural network layer above it.

4.3 Output

The output of the last encoding layer is fed to a fully connected neural network that produces a logits vector that has the dimension size of the vocabulary. From here we apply the softmax function then take the highest value in the vector and that is the word that best corresponds to the position in the sentence.

Find Components in the Source Code

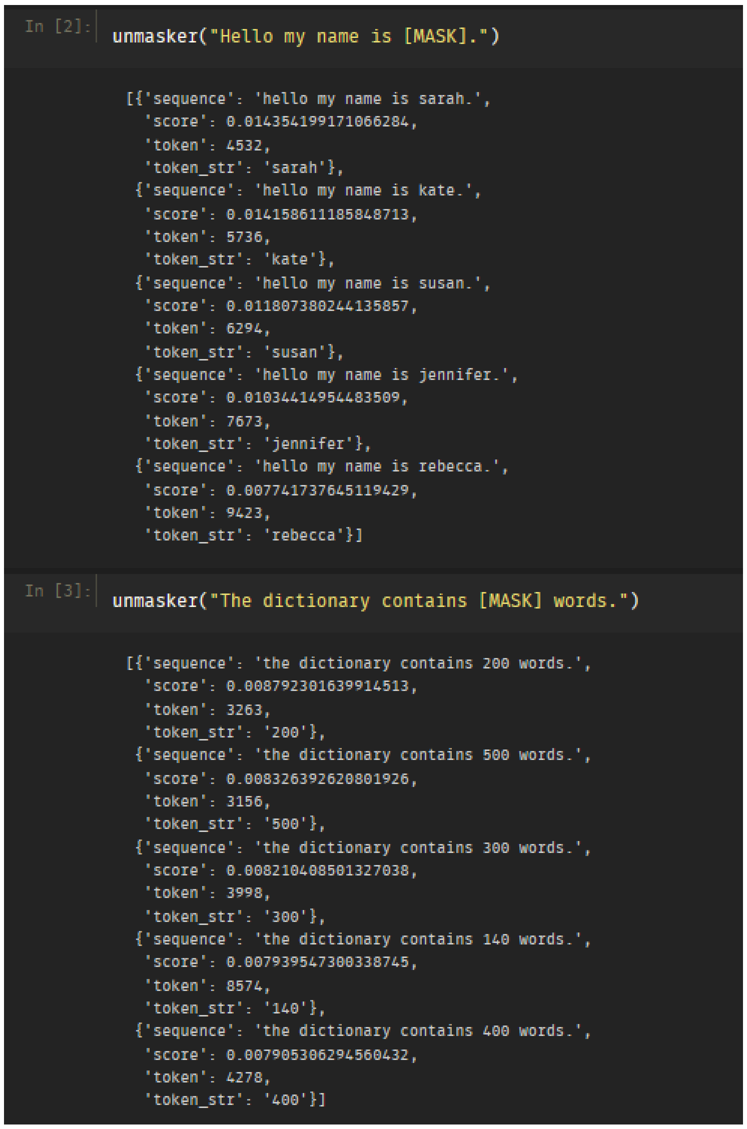
4.4 Embedding

Lines 166-210, the actual number of vocab size and hidden size in line 171 comes from a different python file called "configuration_bert.py" on lines 122,123 respectfully.

4.5 Self-attention

Lines 213-335. Attention head size is found on line 223 with the actual number 12 being defined on the config bert python file on line 125. Multiplying the "Query" vector with all the other "Key" vector is found on line 291. The square rooting of the sub-vector size is found on line 309. Conversion of the scores via softmax is done on line 315. Multiplying the value layer with these softmax scores is done on line 325. Line 325 is essentially the output. This output is fed to the feed forward neural network where its output is either the next self-attention layer or if it's the classification neural network will output the word that best fits that position in the sentence.

4.6 Experiments



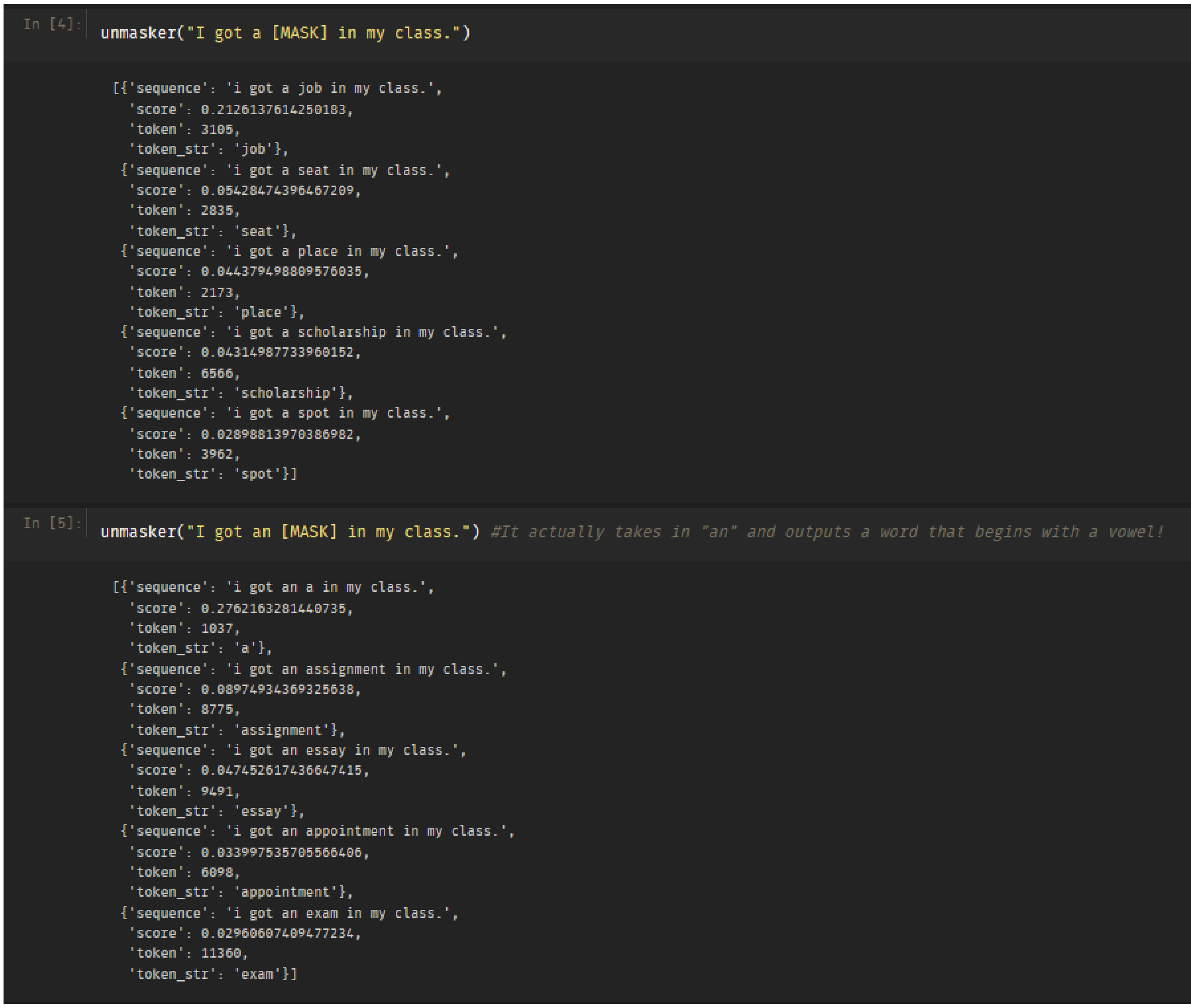
```
In [2]: unmasker("Hello my name is [MASK].")
```

```
[{'sequence': 'hello my name is sarah.',
  'score': 0.014354199171066284,
  'token': 4532,
  'token_str': 'sarah'},
 {'sequence': 'hello my name is kate.',
  'score': 0.014158611185848713,
  'token': 5736,
  'token_str': 'kate'},
 {'sequence': 'hello my name is susan.',
  'score': 0.011807380244135857,
  'token': 6294,
  'token_str': 'susan'},
 {'sequence': 'hello my name is jennifer.',
  'score': 0.010344149544483509,
  'token': 7673,
  'token_str': 'jennifer'},
 {'sequence': 'hello my name is rebecca.',
  'score': 0.007741737645119429,
  'token': 9423,
  'token_str': 'rebecca'}]
```

```
In [3]: unmasker("The dictionary contains [MASK] words.")
```

```
[{'sequence': 'the dictionary contains 200 words.',
  'score': 0.008792301639914513,
  'token': 3263,
  'token_str': '200'},
 {'sequence': 'the dictionary contains 500 words.',
  'score': 0.008326392620801926,
  'token': 3156,
  'token_str': '500'},
 {'sequence': 'the dictionary contains 300 words.',
  'score': 0.008210408501327038,
  'token': 3998,
  'token_str': '300'},
 {'sequence': 'the dictionary contains 140 words.',
  'score': 0.007939547300338745,
  'token': 8574,
  'token_str': '140'},
 {'sequence': 'the dictionary contains 400 words.',
  'score': 0.007905306294560432,
  'token': 4278,
  'token_str': '400'}]
```

Figure 33: Question 2.1



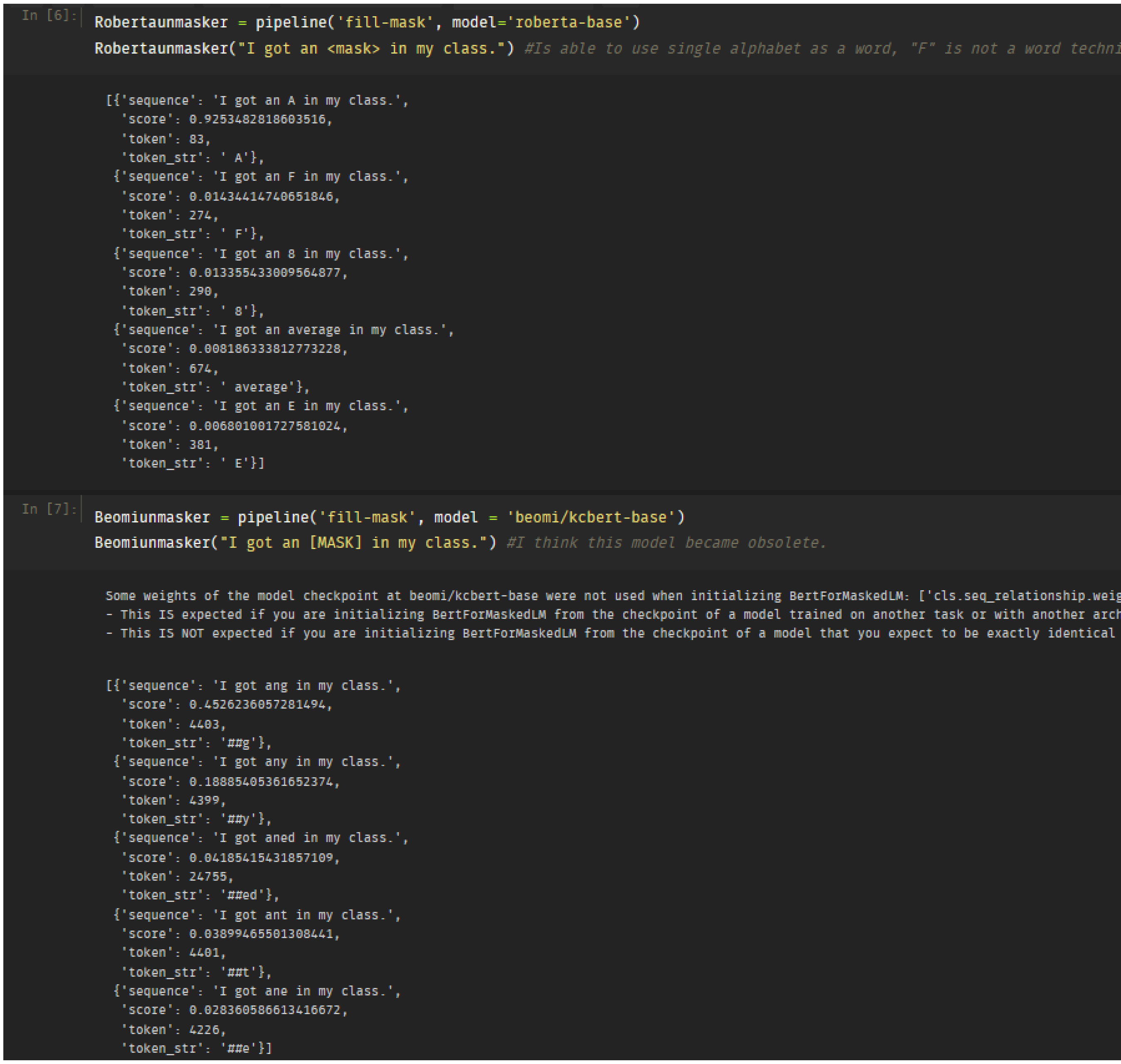
```
In [4]: unmasker("I got a [MASK] in my class.")
```

```
[{'sequence': 'i got a job in my class.',
  'score': 0.2126137614250183,
  'token': 3105,
  'token_str': 'job'},
 {'sequence': 'i got a seat in my class.',
  'score': 0.05428474396467209,
  'token': 2835,
  'token_str': 'seat'},
 {'sequence': 'i got a place in my class.',
  'score': 0.044379498809576035,
  'token': 2173,
  'token_str': 'place'},
 {'sequence': 'i got a scholarship in my class.',
  'score': 0.04314987733960152,
  'token': 6566,
  'token_str': 'scholarship'},
 {'sequence': 'i got a spot in my class.',
  'score': 0.02898813970386982,
  'token': 3962,
  'token_str': 'spot'}]
```

```
In [5]: unmasker("I got an [MASK] in my class.") #It actually takes in "an" and outputs a word that begins with a vowel!
```

```
[{'sequence': 'i got an a in my class.',
  'score': 0.2762163281440735,
  'token': 1037,
  'token_str': 'a'},
 {'sequence': 'i got an assignment in my class.',
  'score': 0.08974934369325638,
  'token': 8775,
  'token_str': 'assignment'},
 {'sequence': 'i got an essay in my class.',
  'score': 0.047452617436647415,
  'token': 9491,
  'token_str': 'essay'},
 {'sequence': 'i got an appointment in my class.',
  'score': 0.033997535705566406,
  'token': 6098,
  'token_str': 'appointment'},
 {'sequence': 'i got an exam in my class.',
  'score': 0.02960607409477234,
  'token': 11360,
  'token_str': 'exam'}]
```

Figure 34: Question 2.1 Continued



```
In [6]: Robertaunmasker = pipeline('fill-mask', model='roberta-base')
        Robertaunmasker("I got an <mask> in my class.") #Is able to use single alphabet as a word, "F" is not a word techni

[{'sequence': 'I got an A in my class.',
  'score': 0.9253482818603516,
  'token': 83,
  'token_str': ' A'},
 {'sequence': 'I got an F in my class.',
  'score': 0.01434414740651846,
  'token': 274,
  'token_str': ' F'},
 {'sequence': 'I got an 8 in my class.',
  'score': 0.013355433009564877,
  'token': 290,
  'token_str': ' 8'},
 {'sequence': 'I got an average in my class.',
  'score': 0.008186333812773228,
  'token': 674,
  'token_str': ' average'},
 {'sequence': 'I got an E in my class.',
  'score': 0.006801001727581024,
  'token': 381,
  'token_str': ' E'}]
```

```
In [7]: Beomiunmasker = pipeline('fill-mask', model = 'beomi/kcbert-base')
        Beomiunmasker("I got an [MASK] in my class.") #I think this model became obsolete.

Some weights of the model checkpoint at beomi/kcbert-base were not used when initializing BertForMaskedLM: ['cls.seq_relationship.weig
- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another arch
- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical


[{'sequence': 'I got ang in my class.',
  'score': 0.4526236057281494,
  'token': 4403,
  'token_str': '##g'},
 {'sequence': 'I got any in my class.',
  'score': 0.18885405361652374,
  'token': 4399,
  'token_str': '##y'},
 {'sequence': 'I got aned in my class.',
  'score': 0.04185415431857109,
  'token': 24755,
  'token_str': '##ed'},
 {'sequence': 'I got ant in my class.',
  'score': 0.03899465501308441,
  'token': 4401,
  'token_str': '##t'},
 {'sequence': 'I got ane in my class.',
  'score': 0.028360586613416672,
  'token': 4226,
  'token_str': '##e'}]
```

Figure 35: Question 2.2

GPT-2 Generative Pre-trained Transformer 2(GPT-2) is another NLP neural network. GPT is a transformer decoder block. Like traditional NLP models, it outputs a single token(i.e word) every time. Each output is then used as part of the input of the next word. This is similar to how RNN works. A key difference to models that encode such as BERT is that the later parts of the sentence are cut off from using as inputs to the model. BERT uses before and after the word in a sentence (hence bidirection), GPT-2 only uses the present and back.

4.7 Predicting the next word based on previous words

This is done by reusing the "Query" "Key" and "Values" vectors in previous words but removing the "mask" of the scores up to the current position. For each word and attention head, GPT generates 3 subvectors, "Query" "Key" and "Values", calculating scores is very similar to BERT but before softmax is being done on the raw scores, a "mask" multiplies the score matrix on positions the word should not have encountered and therefore use in determining the attention distribution of said word. An example would be if the third word of a sentence is being calculated, the attention of this word should not be to the 4th and onwards word in sentence. Another difference from encoding self-attention process is that the "Key" and "Values" vectors of previous words are reused as the input "Key" and "Values" of the current words for the previous words. Another key difference is that GPT has a dedicated fully connected neural network layer as the last layer of the model. The output of the FCNN is then multiplied by the embedding matrix to give a logit matrix representing the probabilities that the specific row (word) is the word that fits the position, we take the best one (sometimes top-k).

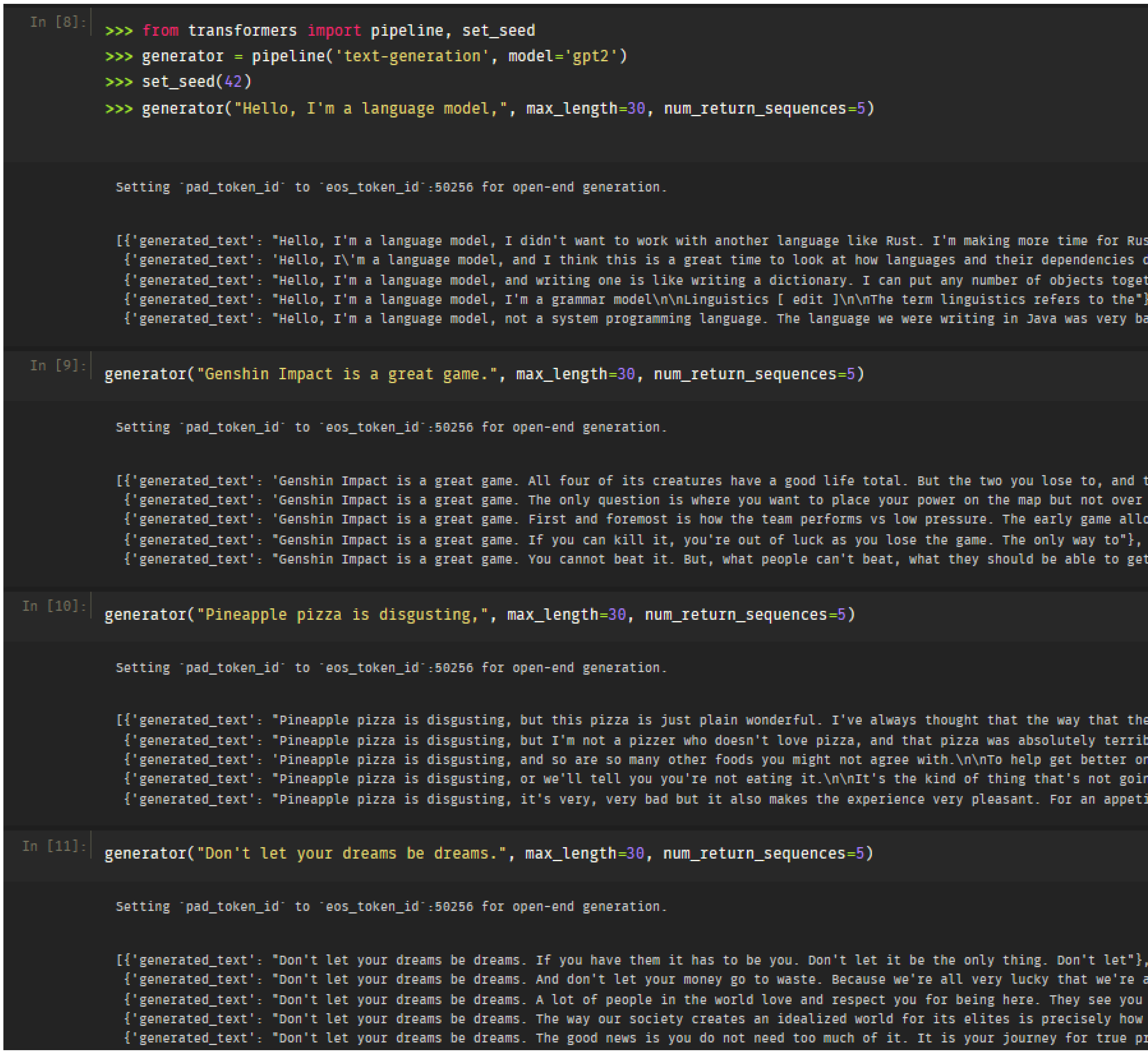
4.8 Masked Self-Attention

Line 139 is number of Attention Heads, Line 178 is the pre-mask attention scores. Line 191 applies the mask onto the scores. Line 193 applies the softmax, Line 200 multiplies softmax with the value matrix and thus gives our output. We also save both the output and the weights of the current word to be used in later words on line 202.

4.9 Attention Heads

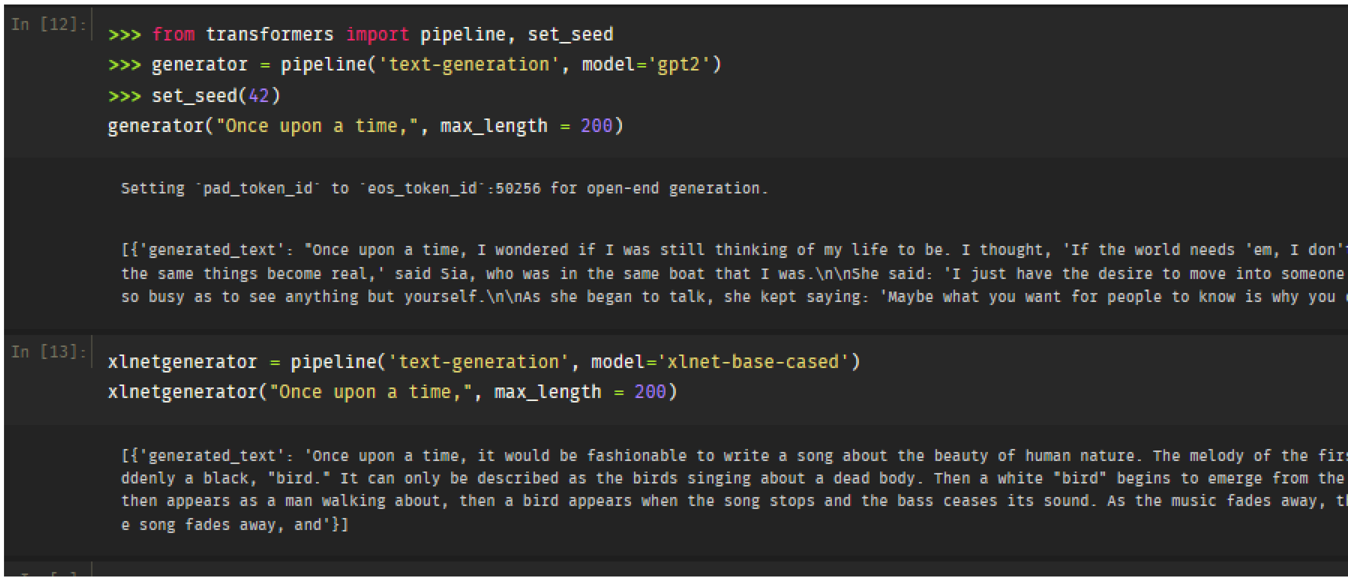
Lines 204-210 is how the model creates Attention heads. It's used to create "Query" "Key" "Values" vectors on lines 244-246. Line 212-218 is how to concatenate the heads. It's used to combine the multiple QKV vectors from the attention heads on line 260. Line 261 converted the concatenated attention heads to the matrix size of the embedded matrix.

4.10 Experiments



```
In [8]: >>> from transformers import pipeline, set_seed
        >>> generator = pipeline('text-generation', model='gpt2')
        >>> set_seed(42)
        >>> generator("Hello, I'm a language model,", max_length=30, num_return_sequences=5)

         Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

         [{'generated_text': "Hello, I'm a language model, I didn't want to work with another language like Rust. I'm making more time for Rus
          {'generated_text': 'Hello, I\'m a language model, and I think this is a great time to look at how languages and their dependencies c
          {'generated_text': "Hello, I'm a language model, and writing one is like writing a dictionary. I can put any number of objects toget
          {'generated_text': "Hello, I'm a language model, I'm a grammar model\n\nLinguistics [ edit ]\n\nThe term linguistics refers to the"}
          {'generated_text': "Hello, I'm a language model, not a system programming language. The language we were writing in Java was very ba

In [9]: generator("Genshin Impact is a great game.", max_length=30, num_return_sequences=5)

         Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

         [{'generated_text': 'Genshin Impact is a great game. All four of its creatures have a good life total. But the two you lose to, and t
          {'generated_text': 'Genshin Impact is a great game. The only question is where you want to place your power on the map but not over
          {'generated_text': 'Genshin Impact is a great game. First and foremost is how the team performs vs low pressure. The early game allo
          {'generated_text': "Genshin Impact is a great game. If you can kill it, you're out of luck as you lose the game. The only way to"},
          {'generated_text': "Genshin Impact is a great game. You cannot beat it. But, what people can't beat, what they should be able to get

In [10]: generator("Pineapple pizza is disgusting,", max_length=30, num_return_sequences=5)

         Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

         [{'generated_text': "Pineapple pizza is disgusting, but this pizza is just plain wonderful. I've always thought that the way that the
          {'generated_text': "Pineapple pizza is disgusting, but I'm not a pizzer who doesn't love pizza, and that pizza was absolutely terrib
          {'generated_text': 'Pineapple pizza is disgusting, and so are so many other foods you might not agree with.\n\nTo help get better or
          {'generated_text': "Pineapple pizza is disgusting, or we'll tell you you're not eating it.\n\nIt's the kind of thing that's not goin
          {'generated_text': "Pineapple pizza is disgusting, it's very, very bad but it also makes the experience very pleasant. For an appeti

In [11]: generator("Don't let your dreams be dreams.", max_length=30, num_return_sequences=5)

         Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

         [{'generated_text': "Don't let your dreams be dreams. If you have them it has to be you. Don't let it be the only thing. Don't let"},
          {'generated_text': "Don't let your dreams be dreams. And don't let your money go to waste. Because we're all very lucky that we're a
          {'generated_text': "Don't let your dreams be dreams. A lot of people in the world love and respect you for being here. They see you
          {'generated_text': "Don't let your dreams be dreams. The way our society creates an idealized world for its elites is precisely how
          {'generated_text': "Don't let your dreams be dreams. The good news is you do not need too much of it. It is your journey for true pr
```

Figure 36: Question 3.1



```
In [12]:  >>> from transformers import pipeline, set_seed
          >>> generator = pipeline('text-generation', model='gpt2')
          >>> set_seed(42)
          generator("Once upon a time,", max_length = 200)

           Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

           [{'generated_text': "Once upon a time, I wondered if I was still thinking of my life to be. I thought, 'If the world needs 'em, I don'
           the same things become real,' said Sia, who was in the same boat that I was.\n\nShe said: 'I just have the desire to move into someone
           so busy as to see anything but yourself.\n\nAs she began to talk, she kept saying: 'Maybe what you want for people to know is why you

In [13]:  xlnetgenerator = pipeline('text-generation', model='xlnet-base-cased')
          xlnetgenerator("Once upon a time,", max_length = 200)

           [{'generated_text': 'Once upon a time, it would be fashionable to write a song about the beauty of human nature. The melody of the fir
           ddenly a black, "bird." It can only be described as the birds singing about a dead body. Then a white "bird" begins to emerge from the
           then appears as a man walking about, then a bird appears when the song stops and the bass ceases its sound. As the music fades away, t
           e song fades away, and'}]
```

Figure 37: Question 3.2

5 Homework 4: Generative Model: VAE & GAN

5.1 Introduction to GAN

Generative Adversarial Network (GAN) is a type of neural network that has 2 models we're trying to optimize. One is called the generator and the other the discriminator. The generator model's goal is to create fake images so well that the discriminator model cannot decide which image is fake and which one is real. The discriminator model classifies which images are fake and which are real.

5.2 Key Equation and Structure

The loss function of GAN is

$$l_n = -[y_n * log(X_n) + (1 - y_n) * log(1 - x_n)] \tag{12}$$

There's two parts to this equation, the both halves are related to the discriminator function. The first half is the probablility that the discriminator sees the image given as from the training set. The second half is the probability that the generator's output is fake. What this does is the discriminator tries to maximize both halves because it wants to correctly classify reals and fakes. However, the generator simply tries to minimize the second half. In actual code, the discriminator has 2 separate loss functions added together and the generator has only 1. Also, we minimize the second half by maximizing X_n. GAN's loss function is known as Binary Cross Entropy, BCEloss.

5.2.1 Structure

GAN can use both linear layers and convolutional layers. GAN that uses convolutional layers is known as DCGAN, Deep Convolutional Generative Adversarial Network. For normal GAN's generator layers, we use linear layers with LeakyReLu as our activation function in between linear layers and finally a TanH activation function at the end of the generator layer. For the discriminator layer, we use LeakyReLu with a dropout of 0.3 in between linear layers, and at the end a Sigmoid activation function. The dimension of GAN's generator layer scales from 100 to our required output image size (28 x 28) by factors of 2 and then downscales from 1024 to 784 at the second to last linear layer. The discriminatory layer scales down from 784 to 1 with the first layer going to 1024 then each subsequent layer halves the dimension with the last layer reducing to 1. We reduce to 1 because the output of a discriminator model is binary (real or fake) so we need only 1 dimension on a scale from [0,1] on the probability that the image is real.

For DCGAN the concept is similar except we use ConvTranspose to upscale our generator dimensions to the required 784 and Conv2d to downscale our discriminator model to 1. We utilize kernal size, strides, and padding to make our dimensions basically halved inbetween every layer just like in GAN. The formula for dimension changing is
For Discriminator or Conv2d:

$$Out = [(In + 2p - k)/s] + 1 \tag{13}$$

where In = input dimension (starts at 28 x 28) p = padding k = kernal size s = stride Out = output dimension.
For Generator or ConvTranspose:

$$Out = (In - 1) * s - 2p + (k - 1) + 1 \tag{14}$$

A big takeaway from DCGAN is that the model heavily utilizes striding and it makes sense since strides half and double the dimensional layers for discriminator and generator model respectively.

5.3 Experiments

In file Stats413DCGAN.ipynb, blocks 5-6 show the Generator model, blocks 7-8 show the Discriminator model, block 9 shows the loss function, block 10 the training loop, block 11 the training loss graph with sum(G+D) included, block 12 the synthesis result.
In file Stats413GAN.ipynb, block 5-6 is the Generator model, 7-8 is the Discriminator model, block 9 is the loss function, block 10 is the training step, block 11 is the training loss, block 12 is the synthesis result, block 15 is the 10 * 10 grid of latent space.

5.4 VAE

5.4.1 Introduction

Variational Autoencoder (VAE) is a type of neural network that utilizes probability into the loss function. VAE has an encoder part and a decoder part of their network. The encoder part shrinks down the input dimensions to a dimension we call the latent dimension. This latent dimension is then used in the decoder section to upscale the matrix back to it's original dimensions. Encoder can also be thought of as probablity of the latent dimension given our input and the decoder as the probablity of the input given a latent dimension.

5.5 Key Equation and Structure

Half of the loss function of VAE is actually the loss function of GAN, which is BCE.The other half requires mu, and log var of a normal distribution.

$$Loss = BCE+KLD = -[y_n*log(X_n)+(1-y_n)*log(1-x_n)]+-0.5sum(1+logvar-mu^2-log.var.exp) \tag{15}$$

The idea is that the further away the latent vector z you got from using the encoder, the less probable the input given is. We want most of our input be able to map to a normal distribution z was created using a normal distribution. We get penalized from not sampling from a normal distribution. The loss function of VAE is called ELBO.

5.5.1 Structure

VAE has 2 parts an encoder and a decoder. These are linear layers that goes from the input dimension of the image (28 x 28) bottlenecking to dimension size 2 then expanding back to the original dimension (28 x 28). The reason why it shrinks down to 2 is because images are 2-dimensional so when we sample from a normal distribution we can already start mimicking an image with a matrix of size 2 rather than multi-dimensional or a scalar. There's multiple functions that can be called from the module. We can start from the latent space z and create an image by calling the decoder part of the module.

5.5.2 Experiments

In file Stats413VAE.ipynb, blocks 2-3 contain the module, block 4 contains the loss function, block 5 contains the training loop, 6 the testing loop, 7 runs the loops for epoch times. block 8 shows the training loss, block 9 is the synthesis result from fixed noise, block 10 is the reconstruction of training data passed through VAE.

6 Homework 5: Boosting and Clustering

6.1 Introduction to Boosting

Boosting is a classifying model that repeatedly adds new estimators to minimize the errors of the previous estimators. Boosting uses many weak classifiers, essentially classifiers that just barely perform better than chance, to build a single strong classifier.

6.2 Key Equations

Boosting uses weights and our loss function updates these weights.

$$\sum_{i=1} w_i * e^{-y_i * \alpha * h_i} \tag{16}$$

Where alpha is

$$\alpha = \frac{1}{2} \ln \frac{(1 - error)}{error} \tag{17}$$

Note that error is always < 0.5 because our criteria of a classifier is to be at least better than random chance. Also once our classifier achieves perfect classification, we'd actually get an error message in our code because we're going to be dividing by 0. We stop the modeling process right before this happens.We use weights to decide whether or not the feature is useful in predicting the outcome label.We initialize the weights to be simply 1/n for n many features. This means initially each feature is equally useful in predicting the outcome label. The sum of the weights of features must add up to 1.
We calculate error by:

$$error = \mathbb{E}(Y_i! = \hat{Y}_i) \tag{18}$$

This equation means that if our predicted classification of y (0,1) is not the same as the true value of y we add a 1 to the total error, then we average out the total error. Error has dimensions equal to the number of samples and the number of features in the model. A big takeaway from boosting models is that to improve the model or rather minimize the error, we only care about the sign of the predicted output because it's a binary classification.

6.2.1 Structure and Adaboost Code reading

Adaboost and its structure can be found in the AdaBoostXGBoost.ipynb file on block 4, it is commented as well.

6.3 Visualization

Block 7 is where I ran the boosting models and created the graph.

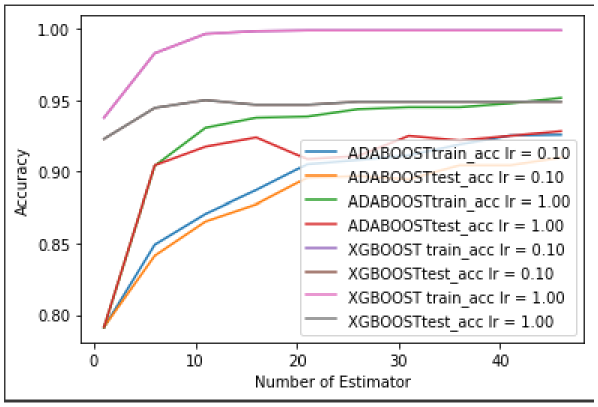


Figure 38: AdaBoost XGBoost Accuracy by different number of Estimators and Learning Rate

6.4 Spectral and K-Means Clustering

6.5 Introduction

Clustering Models uses similarities in Euclidean distance of data points to represent features of the data-set. Essentially, we create features by how close they are to each other. Points that are closer together are considered as part of one group. Different clustering models have different approaches to how to define a group. Spectral Clustering uses eigenvalues rather than absolute location to determine which points belong in which clusters.

6.6 Key Equations and Structure of Spectral Clustering

For Spectral Clustering we need 4 things: A similarity graph, the Laplacian Matrix found through finding the Adjacency and Degree matrix, the eigenvectors of the Laplacian matrix, and train a k-means model using the second smallest eigenvector as input.
The similarity matrix is simply the Euclidean distance of the points between each other point:

$$d(x, y) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2} \tag{19}$$

This matrix is an NxN matrix where N is the number of points. The diagonals equals to 0 (the distance from itself to itself should be 0).
The adjacency matrix is how many nodes are adjacent to a specific node, this can be analagous to the game minesweeper. A property of this matrix is that the diagonals are also 0.
The degree matrix is created by making the elements of the diagonals is equal to the sum of row in the adjacency matrix. The sum of the 1-th row in the adjacency matrix is equal to [i,i] in the degree matrix.
We then subtract the adjacency matrix from the degree matrix and we get the Laplacian matrix.
We calculate the eigenvalues of this Laplacian matrix and find the k lowest ones with k equal to how many clusters you want in the model. One property of the Laplacian matrix is that there will always be a 0 eigenvalue in the matrix. A key note here as well is that there will be n many eigenvalues equal to 0 for n many components in the matrix. A component is a series of points that can be connected to each other. If there's no way for one group of points to travel to another set of points, they are considered different components. We utilize this in Spectral Clustering as they are easy delineates of a unique cluster. So for each k cluster we take the k-lowest eigenvalues and make a cut at that value. Values above and below this cut are considered unique clusters.

6.7 Spectral Clustering Code Reading

Block 3 has the Euclidean Distance function.
Block 4-5 has the Adjacency Matrix function.
Block 6 has the Degree and Laplacian function.
Block 25 contains the similarity matrixes at different delta values as well as the true labels of X.
Block 26 contains the Spectral and KMeans Clustering models at different delta values as well as the mean accuracy results of said models.

6.8 Visualization

As delta increases from 0.1 to 1, the values of the similarity matrix in general increases, but increases moreso when the values are closer to the original point/node. Nodes closest to the original point/node are high regardless of delta. Delta "expands" what is considered close to the point/node.

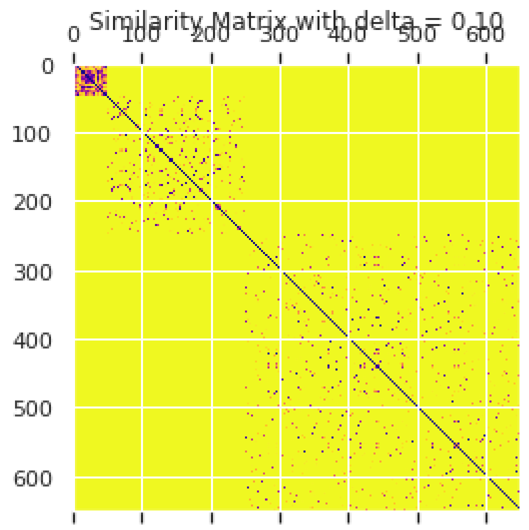


Figure 39: Spectral Clustering Similarity Matrix with Delta = 0.1

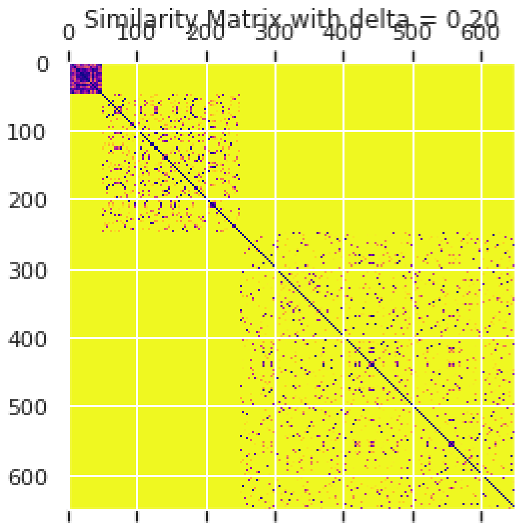


Figure 40: Spectral Clustering Similarity Matrix with Delta = 0.2

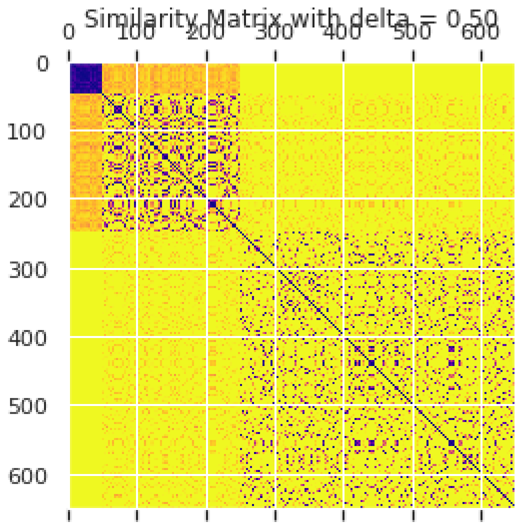


Figure 41: Spectral Clustering Similarity Matrix with Delta = 0.5

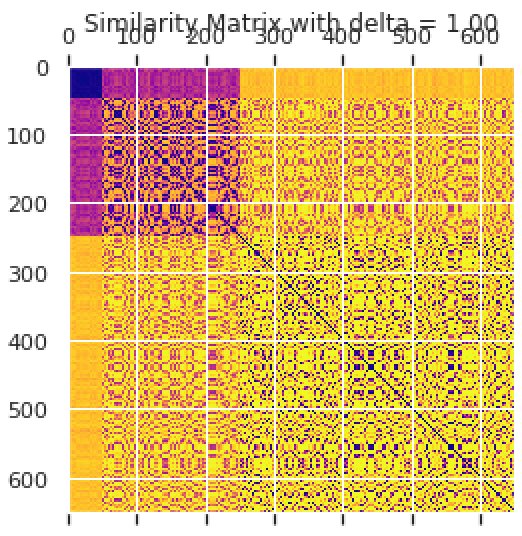


Figure 42: Spectral Clustering Similarity Matrix with Delta = 1

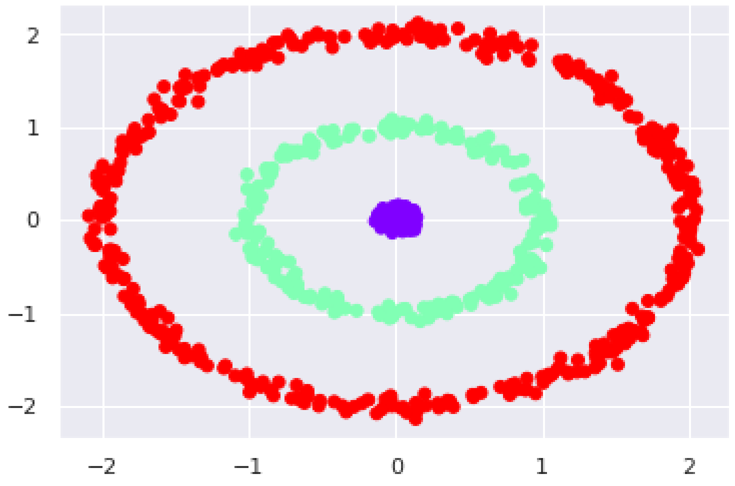


Figure 43: True Labels of X

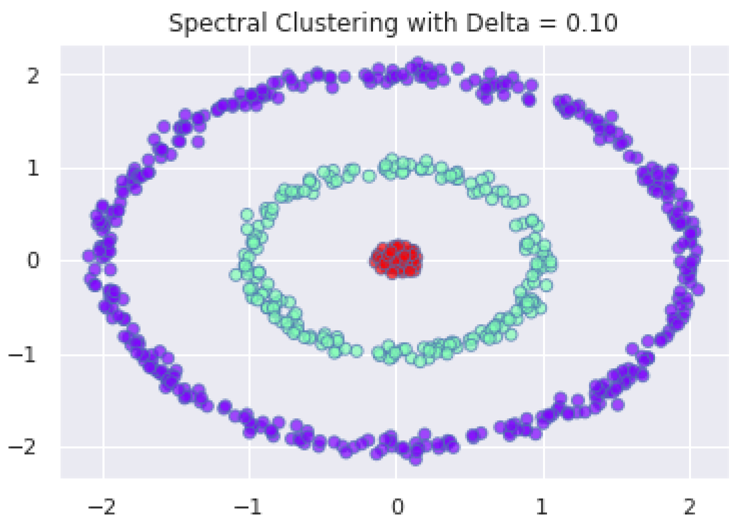


Figure 44: Spectral Clustering Result with Delta = 0.1

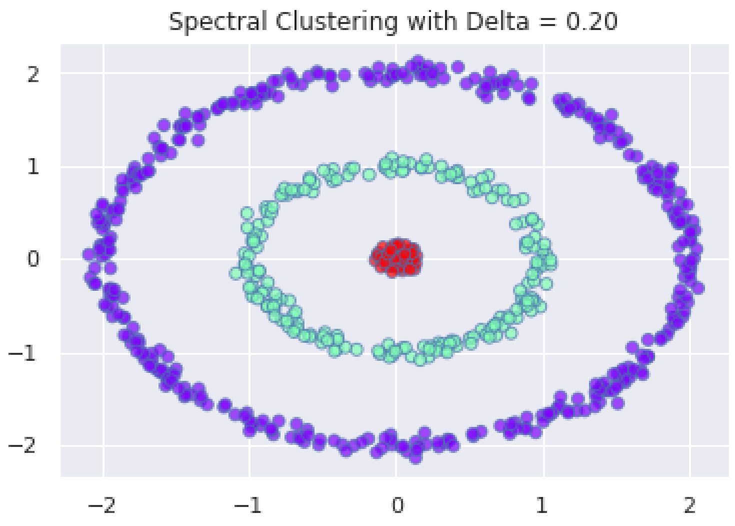


Figure 45: Spectral Clustering Result Matrix with Delta = 0.2

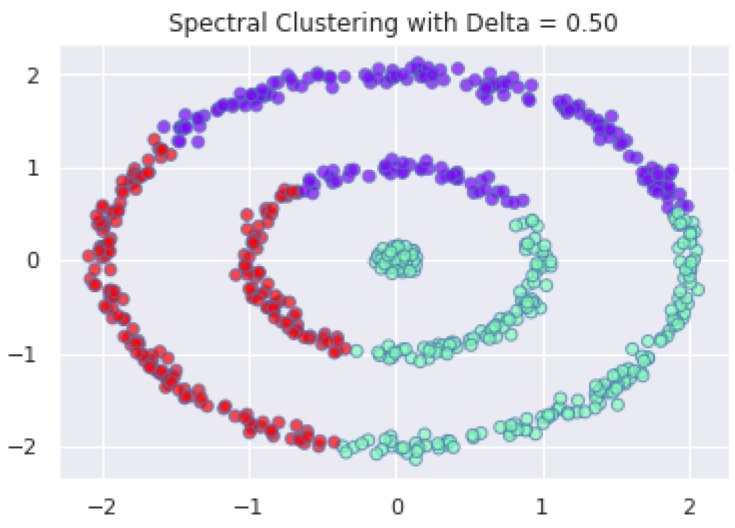


Figure 46: Spectral Clustering Result Matrix with Delta = 0.5

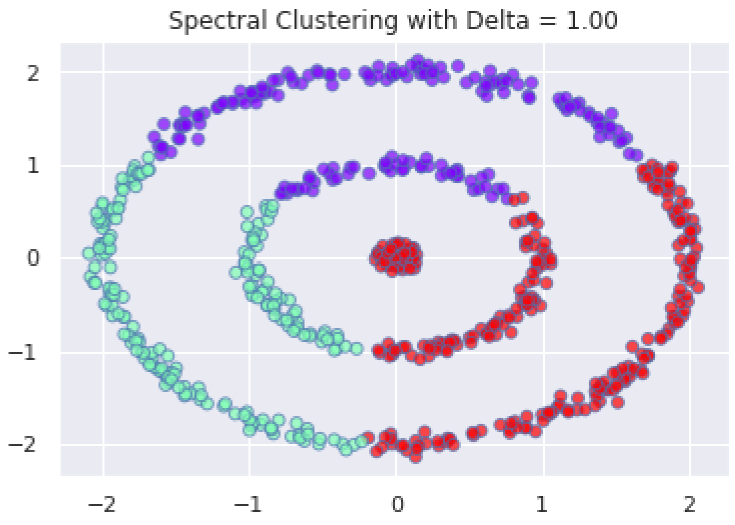


Figure 47: Spectral Clustering Result Matrix with Delta = 1

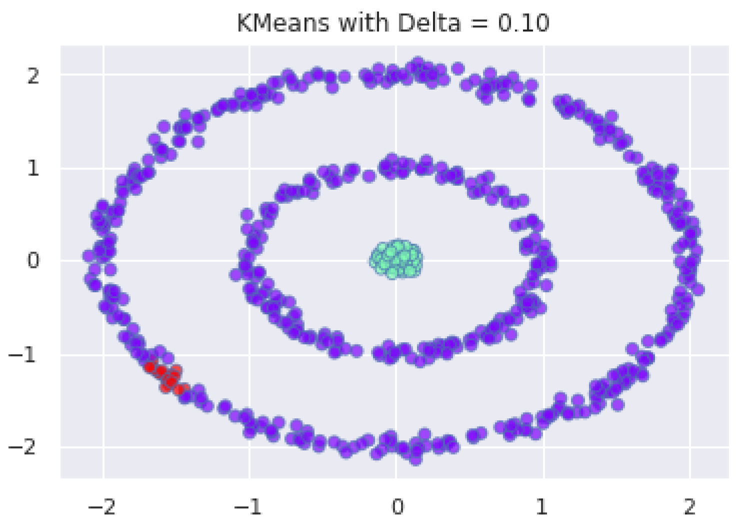


Figure 48: KMeans Result with Delta = 0.1

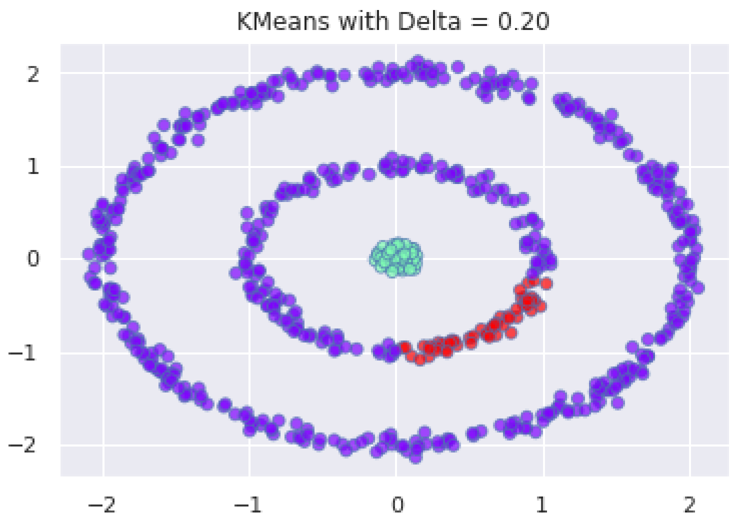


Figure 49: KMeans Result with Delta = 0.2

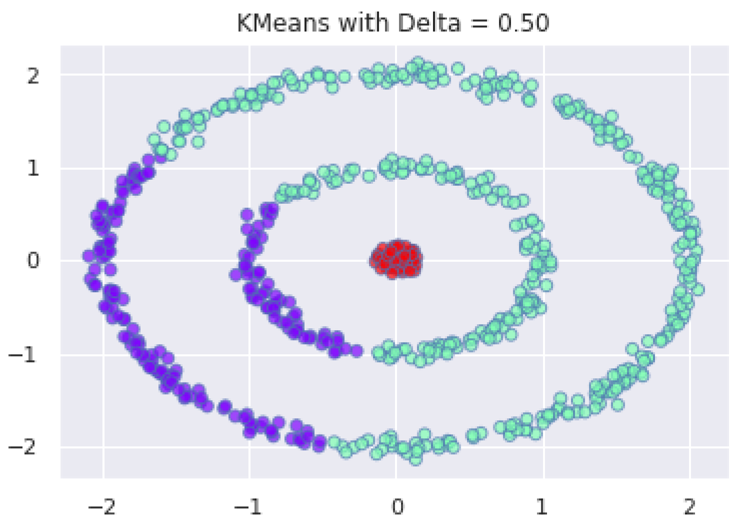


Figure 50: KMeans Result with Delta = 0.5

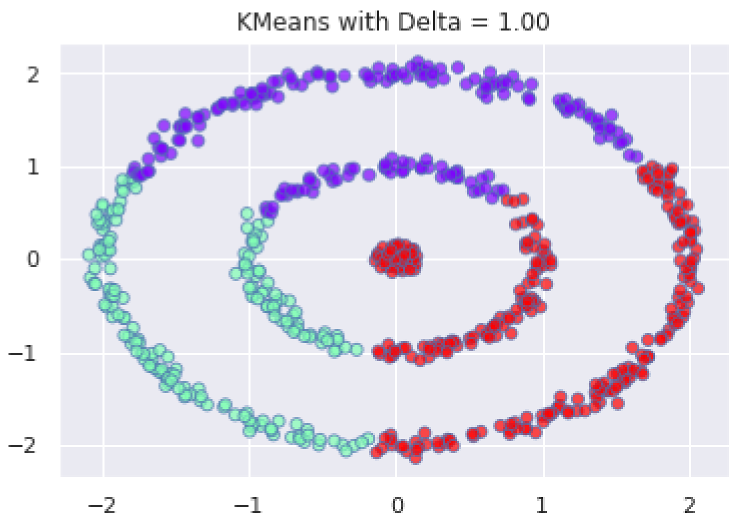


Figure 51: KMeans Result with Delta = 1

6.9 Accuracy of Clustering Models

Spectral Clustering performed better than KMeans on this toy example.
Low delta performed better as well than higher delta.

7 Homework 6: SVM

7.1 Introduction to Support Vector Machine

Support Vector Machine is a classification method that classifies labels by a decision boundary(a line) that maximizes the space between points. It uses kernel functions and a linear equation to define the line that separates the data.

7.2 Key Equations

We use an inequality of a linear regression equation in order to classify the labels.

$$\hat{y_i} = \begin{cases} 1, & if\beta_0 + \beta_1 X_i 1 + \beta_2 X_i 2 + ...\beta_k X_i k \geq 0 \\ -1, & if\beta_0 + \beta_1 X_i 1 + \beta_2 X_i 2 + ...\beta_k X_i k \leq 0 \end{cases} \tag{20}$$

What this means is that from our model, if our model predicts a positive outcome, we say that the point is above the line, and is in classification 1. If it predicts a negative outcome, we say that the point is below the line, and is in classification 2.
The way we update or improve the model is via kernel functions. One kernel function is called Radial Basis Function(rbf) and it's kernel is defined as

$$exp(-\gamma||x - x'||^2) \tag{21}$$

where γ is called Gamma and is synonymous to negative delta in previous homework. As Gamma does up, the importance of datapoints closer to the datapoint relative to points farther away goes up.
There's also a linear kernel function and it's simply

$$(x - x') \tag{22}$$

This means the features of the dataset times itself transposed. It yields a n x n matrix where n is the number of samples in the dataset.
The duel problem of SVM is that we're trying to maximize the minimal distance of datapoints but we're also trying to minimize the maximal distance of the datapoints as well in the primal equation.
the primal equation is

$$min\frac{1}{2}w^T w + C\sum \zeta \tag{23}$$

Where w are the weights, C is an adjustable penalty parameter, and ζ is the distance to the decision boundary.
The dual equation is

$$min\frac{1}{2}\alpha^T Q_\alpha - e^T \alpha \tag{24}$$

Q_α is a function that contains the kernel function we use. α are called the duel coefficients. e is a vector of all ones.

7.3 Figures

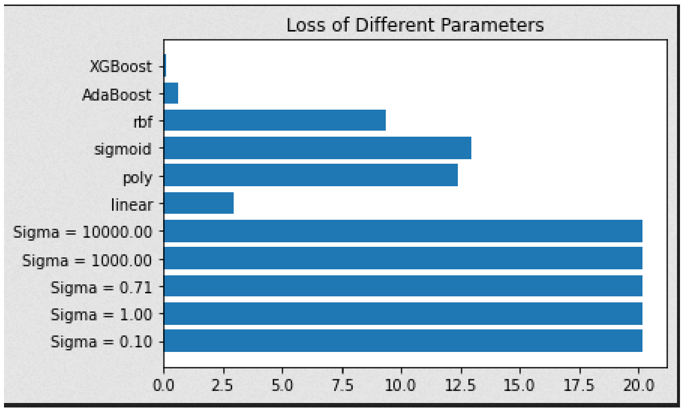


Figure 52: Loss of Different Parameters

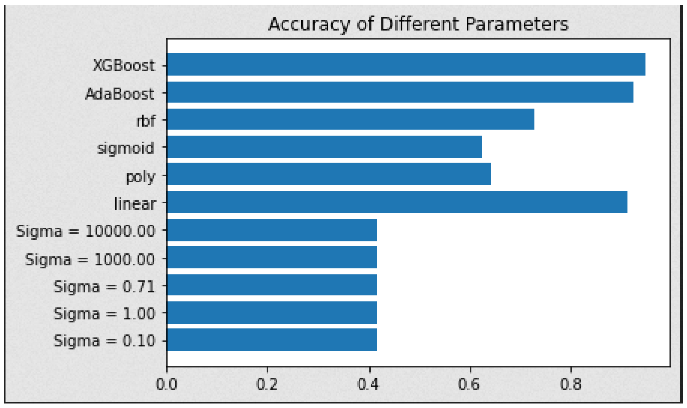


Figure 53: Accuracy of Different Parameters

7.4 Discussion

I think the reason why changes in sigma in the Gaussian Matrix didn't affect performance is because the precomputed function takes in a function of the kernel, so using just the kernel is not sufficient enough for SVC. My guess is that as sigma increases, "Gamma" increases since the Gaussian kernel is a special case of the RBF Kernel. Since "Gamma" increases, datapoints nearby the target datapoint are required to be closer together than if "Gamma" was low. This means that accuracy should be low for low sigmas and be higher at 1000 or 10000. 10000 might be too much so there is a sweet spot, the SVC class defined Gamma as $\frac{1}{features*var(x)}$ which is about 151 in the spam dataset. XGBoost and AdaBoost performed the best out of all the params with the linear kernel function performing the best in the SVC model. It is surprising that a simple kernel function such as the linear kernel function is the best SVC model and is very good at classifying spam.

Acknowledgments

This template is NIPS2017 template downloaded from NIPS2017 website.

Reference

Write a reference here if necessary.