Purdue University

OPT4DL

HW 4

October 3, 2024

33678229 John Yeary

Contents

1 Question 1 **1**

2 Question 2 **2**

3 Question 3 **3**

3.1 Part A 3

3.1.1 Deriving Hessian 3

3.1.2 Proving Convex and L Smooth 4

3.2 Part B 4

4 Question 4 **6**

4.1 Explanation 6

4.2 Code 6

1 Question 1

This paper presents a benchmark of various zero order methods for fine tuning a large language model (LLM). This is in stark contrast to first order methods, which requires a backpropagation step. These steps are memory and computationally efficient. As models increase in size, memory requirements scale exponentially. To remedy this, ZO methods offer the opportunity to estimate the gradient without requiring backpropagation. These methods have the potential to reduce memory requirements and computational overhead.

One major finding of the paper was the importance of task alignment during fine tuning. During pre-training, the model is fed massive amounts of data, and training is performed using next token prediction, or similar methods. This informs the model's parameters to be within a 'ballpark' of a final solution. Once the fine tuning step begins, it is important to use a similar format to the pre-training data that was fed to the model. In doing so, the model starts at a reasonable place within its parameter space, so fine tuning is more straightforward.

This starting point is important since zero order methods will be sensitive to starting location. Since these methods are approximating the gradient with finite differences in values, small changes in function value can cause the model's convergence to vary widely. By identifying a better starting point, convergence becomes easier with a ZO model. This may struggle with generalization as the model will become specialized for certain data types.

The paper suggests that there is a tradeoff in memory efficiency and model accuracy. There are a few suggestions from the paper as well. The authors suggest using block ZO, hybrid FO and ZO, and gradient pruning to improve ZO performance.

Block-wise ZO solves the issue of high gradient estimation variance typical of ZO models. Typically the gradient for the entire model is estimated at once, leading to reduced accuracy across the model. By breaking the model into smaller blocks and calculating a gradient approximation for each block, you can limit the variance of each block's variance, effectively improving the training for that block. Since we aren't requiring a backpropagation, each block can be self contained. One drawback to this approach is that it requires multiple forward passes per iteration, since each block will require a forward pass.

Hybrid approaches aim to combine the accuracy of FO methods with the memory and computational efficiency of ZO methods.The idea is to use FO methods for some layers, and ZO for the rest. Typically, FO methods are used for deeper model layers, and critical layers within the model. The remaining layers apply ZO approximations to reduce computational requirements. The difficulty in this approach is identifying which layers to treat as FO and which to treat as ZO. This is likely application specific.

Some ZO methods, such as random gradient estimation, attempt to estimate the gradient by randomly perturbing the model's parameters. This can introduce variance to the gradient estimate, especially with large numbers of parameters. To resolve this issue, gradient pruning selects a subset of parameters to perturb in order to approximate the gradient. The selection criteria for which parameters to include/remove from this perturbation can have a large impact on model accuracy.

There are clear memory advantages of using a zero order approximation, but there are still some questions about performance stability and generalization when applying these methods. In general the performance gap between ZO and FO methods increases with task complexity.

The paper does a good job of providing an overview and benchmark for ZO methods. The authors provide a solid starting point for future investigation into ZO methods.

Future work should investigate the scalability issues with ZO methods. This would likely involve some adaptive methods to reduce variance of the gradient estimate.

More work could also focus on establishing a more theoretical reasoning for why the proposed methods work. This would involve evaluating conditions for when and why to apply each method. This could tie into some of the previous reading, where a decision metric was used to identify when it was fine to use 'easier' samples instead of 'harder' samples for training. Similarly, this would apply to deciding to use FO or ZO methods depending on some external criteria.

2 Question 2

A backdoor attack is performed by adding faulty data with a hidden trigger to the training data. This will activate some malicious function within the model. Once trained, the model will behave normally until the trigger appears, causing the model to misbehave and output whatever the malicious functionality is.

These attacks have typically been easy to spot, since any time the trigger appears, the model will behave differently. These were fairly simple to catch since the trigger remains static.

The paper proposes an input aware attack, where multiple triggers are hidden across multiple different training samples. The faulty behavior would then only occur when the unique data and trigger are presented. This would allow the trigger to be unique to each image, and prevent the trigger from occuring on other samples.

The authors proposed a system for generating such an attack: First, they propose creating a trigger generator network. This will encode a unique trigger pattern for each image. They proposed training the model with clean mode, attack mode, and cross trigger mode. Clean mode is simply training, attack mode is training the network to output a predefined 'attack' label when a trigger corresponding to the input is present, and cross trigger mode will train the network to output the correct result even if a trigger for a different image is present.

The authors then tested their attack using various datasets. They tested its effectiveness against multiple forms of defense, and found their attack was able to circumnavigate all of the defenses they tried. They showed that the regions in the input image the network focused on when faced with a trigger was more diverse than in standard backdoor attacks, making it more difficult to identify problematic data.

This paper makes great strides at introducing a key vulnerability to many networks. This exposes vulnerabilities that were previously undocumented or not theorized.

One weakness in the paper is that it focuses solely on image classification tasks. Another shortcoming briefly mentioned by the authors was that the resulting trigger patterns could be easily recognized by a human.

Future work could include creating trigger patterns that appear more realistic, or to expand the domain into other data types. In general the paper itself is a call to researchers to find ways to remove this vulnerability.

3 Question 3

3.1 Part A

3.1.1 Deriving Hessian

Start with the loss function:

$$\mathcal{L}(w) = \frac{1}{N}\sum_{i=1}^{N} y_i log(1+e^{-<x_i,w>}) + (1-y_i)(<x_i,w>) \tag{1}$$

Let:

$$X = \begin{bmatrix} x_{11} & x_{12} & ... & ... & x_{1d} \\ x_{21} & . & & & x_{2d} \\ ... & & . & & \\ x_{N1} & & & . & x_{Nd} \end{bmatrix} \tag{2}$$

Where N is the number of training samples, and d is the number of features. Let:

$$W = (w_1, w_2, ..., w_d) \tag{3}$$

And let $j \in [1, d]$ Now, the partial derivative of $\mathcal{L}(w)$ with respect to w_j is:

$$\begin{aligned} \frac{\partial \mathcal{L}(w)}{\partial w_j} &= \frac{1}{N}\frac{\partial}{\partial w_j}\sum_{i=1}^{N} y_i log(1+e^{-<x_i,w>}) + (1-y_i)(<x_i,w>) \\ &= \frac{1}{N}\sum_{i=1}^{N} y_i \frac{\partial}{\partial w_j} log(1+e^{-<x_i,w>}) + \frac{\partial}{\partial w_j}(1-y_i)(<x_i,w>) \\ &= \frac{1}{N}\sum_{i=1}^{N} -x_{ij}\frac{1}{1+e^{W^T x_i}} y_i + (1-y_i)x_{ij} \end{aligned} \tag{4}$$

Next, take the derivative w.r.t w_k, where $k \in [1, d]$

$$\begin{aligned} \frac{\partial^2 \mathcal{L}(w)}{\partial w_j \partial w_k} &= \frac{1}{N}\sum_{i=1}^{N} -x_{ij}\frac{\partial}{\partial w_k}(\frac{1}{1+e^{W^T x_i}})y_i + 0 \\ &= \frac{1}{N}\sum_{i=1}^{N} x_{ij}x_{ik}\frac{e^{W^T x_i}}{(1+e^{W^T x_i})^2} y_i \end{aligned} \tag{5}$$

Let:

$$t_i = \frac{e^{W^T x_i} y_i}{(1+e^{W^T x_i})^2} \tag{6}$$

Note that since $y \in [0, 1]$, $t_i >= 0$ So:

$$T = \begin{bmatrix} t_1 & 0 & 0 & \cdots & 0 \\ 0 & t_2 & 0 & \cdots & 0 \\ 0 & 0 & t_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & t_n \end{bmatrix} \tag{7}$$

And:

$$q_i = (x_{1i}, x_{2i} \cdots x_{ni})^T \tag{8}$$

For any $i \in [0, d]$. These are the columns of the X matrix.

Now:

$$\begin{aligned} \frac{\partial^2 \mathcal{L}(w)}{\partial w_j \partial w_k} &= \frac{1}{N}\sum_{i=1}^{N} q_j^T T q_k \\ \nabla_W^2 \mathcal{L}(w) &= \frac{1}{N} X^T T X \end{aligned} \tag{9}$$

3.1.2 Proving Convex and L Smooth

Since $t_i >= 0$ (due to $y_i = [0, 1]$, the eigenvalues for T must also be ≥ 0. Let $P \in \mathbf{R}^d$:

$$\begin{aligned} P^T \nabla \mathcal{L}_w(w)^2 P &= P^T(\frac{1}{N} X^T T X) P \\ &= \frac{1}{N} P^T X T (P^T X)^T \\ &= \frac{1}{N} ||P^T T X||^2 \geq 0 \end{aligned} \tag{10}$$

For any P This is positive semi-definite, meaning it is convex, but not strongly.

Next, to prove L smoothness, show that the maximum and minimum values for T are bounded:

$$\begin{aligned} \lim_{W^T x^i \to +\infty} &\approx \frac{e^{W^T x_i}}{e^{2W^T x_i}} \approx 0 \\ \lim_{W^T x^i \to -\infty} &\approx \frac{e^{W^T x_i}}{1} \approx 0 \end{aligned} \tag{11}$$

Therefore the slope must be bounded; i.e.

$$\nabla_w \mathcal{L}(w)^2 \leq L I_d \tag{12}$$

So the loss is L-Smooth, and convex.

3.2 Part B

```
# -*- coding: utf-8 -*-
"""OPT4DL_Hw4_Problem3

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1ITnkKN_lKsvAPFeTOVntvNvVBx_uxN8Q
"""

import numpy as np                    # advanced math library
import matplotlib.pyplot as plt       # MATLAB like plotting routines
import random                         # for generating random numbers

from keras.datasets import mnist      # MNIST dataset is included in Keras
from keras.models import Sequential   # Model type to be used

from keras.layers import Dense, Dropout, Activation # Types of layers to be used in our
    model

# The MNIST data is split between 60,000 28 x 28 pixel training images and 10,000 28 x 28
    pixel images
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 784) # reshape 60,000 28 x 28 matrices into 60,000 784-
    length vectors.
```

```
y_train = np.atleast_2d(y_train.T)      # 2d saves headache when multiplying
y_train[y_train>=5] = 1                # binary encoding
y_train[y_train<5] = 0

X_test = X_test.reshape(10000, 784)    # reshape 10,000 28 x 28 matrices into 10,000 784-length vectors.
y_test = np.atleast_2d(y_test.T)        # 2d saves headache when multiplying
y_test[y_test>=5] = 1                  # binary encoding
y_test[y_test<5] = 0

X_train = X_train.astype('float32')    # change integers to 32-bit floating point numbers
X_test = X_test.astype('float32')

X_train /= 255                         # normalize each value for each pixel for the entire vector for each input
X_test /= 255

print("Training matrix shape", X_train.shape)
print("Testing matrix shape", X_test.shape)
print("Label Matrix Shape:", y_train.shape)

# Constants and initialization
B = 32       # batch size
T = 100      # number of epochs
eta = 0.0001 # placeholder
losses = np.zeros((T,1)) # loss for epoch t
w = np.random.normal(0,1.5,size = [784,1])
idx = np.arange(60000)

def sigmoid(x):
  return 1/(1+np.exp(-x))

def lossFn(x,w,y):
  prod = x@w
  prod = np.clip(prod,0,1)
  return (y.T)@np.log(1+np.exp(-prod)) +(1-y).T@prod

for t in range(T):
  #first, select the batch for this iteration:
  batchIdx = np.random.choice(X_train.shape[0],size = X_train.shape[0],replace=False)
  print(f'Epoch ={t}')
  epochLoss = 0
  for b in range(batchIdx.shape[0]):
    thisBatch = batchIdx[b*B:(b+1)*B]
    X_batch = X_train[thisBatch]
    y_batch = y_train[thisBatch]
    # gradient of loss function w.r.t w
    gradL = X_batch.T@(1+y_batch*sigmoid(X_batch@w)-2*y_batch)
    w = w-eta*gradL #update rule
    epochLoss += lossFn(X_batch,w,y_batch)
  losses[t] = epochLoss/X_train.shape[0] #avg loss/sample

plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title(f'Loss vs Epoch: eta = {eta}')
plt.show()

pred = sigmoid(X_test @ w)
pred_labels = (pred >=0.5).astype(int)
accuracy = np.mean(pred_labels == y_test)
print(f'Accuracy: {accuracy}')
```

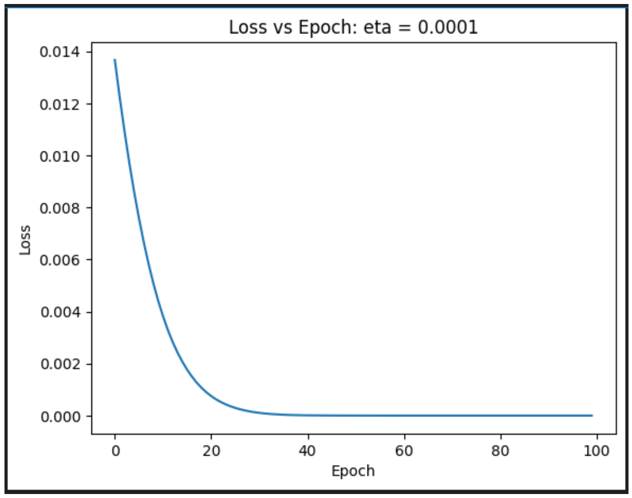

Figure 1: Loss

4 Question 4

4.1 Explanation

Adding more hidden units to the hidden layer will improve the fitting and allow the network to learn more complex patterns. This comes at the risk of overfitting to the training data and not generalizing well. During training, this would be observed by comparing the training loss/accuracy to the validation loss/accuracy. If the training loss continues to improve while the validation loss stagnates or increases, overfitting has occurred. More neurons may also require more training data/epochs to effectively fit the data.

Decreasing the hidden units to 128 has the opposite problem. The model is more simplistic so struggles to learn complex patterns. This may cause the accuracy to be quite low as the model may never be able to approximate a complex pattern. If the model is underfitting, training and test loss will be high, and accuracy will be low. Few neurons typically require less data/epochs to effectively fit the data, assuming the distribution is simple enough for the reduced set to learn it.

By adding a dropout layer, you effectively combat over fitting by regularizing the network. This prevents a single neuron/ path from dominating the model. This typically results in improved test loss/accuracy since the model is less prone to overfitting. This can reduce training performance, since it will effectively limit the learning capacity of the model. What you gain from this limitation is a more robust model.

4.2 Code

My code is really split into two classes: Layer and MLP. The layer class holds members for weights, activation functions, biases, masks (for dropout), and activation input / output for backpropagation. The function 'forward' will handle a single forward pass from this layer, returning the activation output from this layer. The function 'backward' will handle the backpropagation step, which will perform a chain rule and apply the weight and bias update for this layer.

The MLP class holds all of the requested layer objects, and sets up a framework for forward passes and backward passes between layers. This class will also run small batch SGD, calculate cross entropy loss and accuracy for each epoch.

With those classes established, the rest of the code/implementation for each question just relies on instantiating an MLP object and adding layers to it.

```python
# -*- coding: utf-8 -*-
"""OPT4DL_HW4_Problem4

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1h9jtgs6dBFbqmKMHQmoa8jXsNORLKP5x
"""

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pdb
np.random.seed(42)
#Load fasion MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)

x_train = x_train.reshape(60000,784) # flatten images
x_test = x_test.reshape(10000,784)
y_train = np.atleast_2d(y_train.T)      # 2d saves headache when multiplying
y_test = np.atleast_2d(y_test.T)        # 2d saves headache when multiplying

x_train = x_train.astype('float32')   # change integers to 32-bit floating point numbers
x_test = x_test.astype('float32')

x_train /= 255                        # normalize each value for each pixel for the entire
    vector for each input
x_test /= 255


print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)

# Parameters
B = 256 #batch size
num_epochs = 10
learning_rate = 0.001

# Make single layer perceptron
class Layer:

  def __init__(self, input_size, output_size, activation = 'relu',dropout_prob=None):
    #self.weights = np.random.normal(size=(input_size,output_size),loc = 0, scale = 0.01)
    self.weights = np.random.randn(input_size,output_size)*np.sqrt(2./input_size)
    self.bias = np.zeros(shape = (1,output_size))
    self.activation = activation
    self.activation_output = None
    self.mask = None
    self.dropout_prob = dropout_prob
    self.layer_input = None
    self.activation_input = None

  def dropout(self,x,p):
    #hold the mask for backprop
    self.mask = np.random.binomial(1,1-p,size=x.shape)
    return x*self.mask / (1-p)

  def relu(self,x):
    return np.maximum(0,x)

```

```python

  def forward(self,x,training=True):
    z = np.dot(x,self.weights) + self.bias

    self.activation_input = z
    self.layer_input = x

    if self.activation == 'relu':
      self.activation_output = self.relu(z)
    elif self.activation == 'dropout' and training:
      self.activation_output = self.dropout(z,self.dropout_prob)
    else:
      self.activation_output = z
    print(f'Number of dead neurons: {np.sum(self.activation_output == 0)}')

    return self.activation_output

  def backward(self,grad_output,learning_rate,training=True):

    if self.activation == 'relu':
      dz = grad_output * (self.activation_input > 0).astype(float)
    elif self.activation == 'dropout' and training:
      dz = grad_output * self.mask / (1-self.dropout_prob)
    else:
      dz = grad_output
    #compute gradient wrt weights and biases
    dw = 1/self.layer_input.shape[0]*np.dot(self.layer_input.T,dz)
    db = np.mean(dz,axis=0,keepdims=True)

    print(f'dw max: {np.max(dw)}')
    print(f'dw min: {np.min(dw)}')
    print(f'db max: {np.max(db)}')
    print(f'db min: {np.min(db)}')
    #compute gradient wrt input to layer
    self.weights -= learning_rate * dw
    self.bias -= learning_rate * db
    return np.dot(dz,self.weights.T) # Gradient w.r.t. input to the layer

# Combine single layers into MLP

class MLP:

  def __init__(self, input_size, output_size, dropout_prob = None):
    self.layers = []
    self.input_size = input_size
    self.output_size = output_size
    self.dropout_prob = dropout_prob
    self.epochTrainingLoss = []   #list to store epoch losses for SGD
    self.epochTrainingAccuracy = [] #list to store epoch accuracies for SGD
    self.epochValidationLoss = [] # list to store validation losses
    self.epochValidationAccuracy = [] #list to store validation accuracies for SGD

  def add_layer(self,layer):
    self.layers.append(layer)

  def softmax(self, x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))  # for numerical stability
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

  def forward(self,x,training = False):
    for layer in self.layers:
      x = layer.forward(x,training)
      #print(f'x.shape{x.shape}')
    return x

  def cross_entropy_loss(self,y_true,y_pred):
    y_pred = np.clip(y_pred,1e-7,1-1e-7) #avoid zero
    return -np.sum(y_true*np.log(y_pred))/y_true.shape[0]
```

```
  def one_hot_encoding(self,y):
    y = y.reshape(-1)
    one_hot = np.zeros((y.shape[0],self.output_size))
    one_hot[np.arange(y.shape[0]),y] = 1
    return one_hot

  def compute_loss_gradient(self, y_pred, y_true):
      # Compute the gradient of the loss with respect to the predicted probabilities
      return (y_pred - y_true)

  def backward(self, loss_grad,learning_rate):
      for layer in reversed(self.layers):
        loss_grad = layer.backward(loss_grad, learning_rate)

  def process_batch(self, x_batch, y_batch, learning_rate):

    # Forward pass

    logits = self.forward(x_batch, training=True)
    y_pred = self.softmax(logits)
    print(f'prediction:{np.argmax(y_pred,axis=1)}')
    print(f'true:{np.argmax(y_batch,axis=1)}')
    # Compute loss for this batch
    loss = self.cross_entropy_loss(y_batch, y_pred)
    # Backward pass
    loss_grad = self.compute_loss_gradient(y_pred, y_batch)
    self.backward(loss_grad, learning_rate)

    return loss


  def evaluateLoss(self,y_val, y_pred):
    ## '''
    ## Calculate the loss on the validation set.
    ## Pass in validation data and predicted values and evaluate loss.
    ## '''
    return self.cross_entropy_loss(y_val,y_pred) #note this returns non-normalized, so
      average over data

  def calcAccuracy(self,y,y_pred):
    ## '''
    ## Calculate model accuracy. Take in y validation data, and the prediction
    ## from running self.forward(x).
    ## '''
    y_pred = np.argmax(y_pred,axis=1)
    y = np.argmax(y,axis=1)

    return np.mean(y_pred == y)

  def sgd(self,x_train,y_train,x_val,y_val, learning_rate=0.001,epochs = 10, batch_size =
     32):
    ## '''
    ## Run the actual training. This will evaluate loss and accuracy each epoch.
    ## '''
    num_samples = x_train.shape[0]
    # One hot encode the true labels
    y_train = self.one_hot_encoding(y_train)
    y_val = self.one_hot_encoding(y_val)
    # training loop; loop epochs and batches
    for t in range(epochs):
      batchIdx = np.random.choice(num_samples,size = num_samples,replace=False)
      print(f'######## Epoch ={t} ##########')
      num_batches = num_samples // batch_size #number of full batches
      remaining_samples = num_samples % batch_size #remaining samples
      epochTrainingLoss = 0
      for b in range(num_batches):
        #slice batches
```

```
        thisBatch = batchIdx[b*batch_size:(b+1)*batch_size]
        x_batch = x_train[thisBatch,:]
        y_batch = y_train[thisBatch,:]
        loss = self.process_batch(x_batch,y_batch,learning_rate)
        epochTrainingLoss += loss


      if remaining_samples > 0:
        x_batch = x_train[num_batches*batch_size:]
        y_batch = y_train[num_batches*batch_size:]
        loss = self.process_batch(x_batch,y_batch,learning_rate)
        epochTrainingLoss += loss
      self.epochTrainingLoss.append(epochTrainingLoss/num_samples)
      #print(f'epochTrainingLoss{epochTrainingLoss/num_samples}')
      y_predTrain = self.softmax(self.forward(x_train,training=False))

      self.epochTrainingAccuracy.append(self.calcAccuracy(y_train,y_predTrain))
      # evaluate performance
      y_predVal = self.softmax(self.forward(x_val,training=False))
      self.epochValidationLoss.append(self.cross_entropy_loss(y_val,y_predVal)/y_val.shape
          [0])
      self.epochValidationAccuracy.append(self.calcAccuracy(y_val,y_predVal))

# Make MLP
## PART A

inputLayer = Layer(input_size = 784,output_size = 256,activation = 'relu')
outputLayer = Layer(input_size = 256,output_size = 10,activation = 'none')
mlpA = MLP(input_size = 784,output_size = 10)
mlpA.add_layer(inputLayer)
mlpA.add_layer(outputLayer)
#run sgd
mlpA.sgd(x_train, y_train,x_test,y_test, learning_rate, 10, 256)

## PART B
# 512 hidden
mlpB512 = MLP(input_size = 784,output_size = 10)
inputLayer512 = Layer(input_size = 784,output_size = 512,activation = 'relu')
outputLayer512 = Layer(input_size = 512,output_size = 10,activation = 'none')
mlpB512.add_layer(inputLayer512)
mlpB512.add_layer(outputLayer512)
mlpB512.sgd(x_train, y_train,x_test,y_test, learning_rate, num_epochs, 256)

# 128 hidden
inputLayer128 = Layer(input_size = 784,output_size = 128,activation = 'relu')
outputLayer128 = Layer(input_size = 128,output_size = 10,activation = 'none')
mlpB128 = MLP(input_size = 784,output_size = 10)
mlpB128.add_layer(inputLayer128)
mlpB128.add_layer(outputLayer128)

mlpB128.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, 256)

# Part C with each size
mlpC256 = MLP(input_size = 784,output_size = 10)
inputLayer = Layer(input_size = 784,output_size = 256,activation = 'relu')
hiddenLayer = Layer(input_size = 256,output_size = 256,activation = 'relu')
outputLayer = Layer(input_size = 256,output_size = 10,activation = 'none')
mlpC256.add_layer(inputLayer)
mlpC256.add_layer(hiddenLayer)
mlpC256.add_layer(outputLayer)
mlpC256.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, 256)

mlpC512 = MLP(input_size = 784,output_size = 10)
inputLayer = Layer(input_size = 784,output_size = 512,activation = 'relu')
hiddenLayer = Layer(input_size = 512,output_size = 512,activation = 'relu')
outputLayer = Layer(input_size = 512,output_size = 10,activation = 'none')
mlpC512.add_layer(inputLayer)
mlpC512.add_layer(hiddenLayer)
```

```
mlpC512.add_layer(outputLayer)
mlpC512.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, 256)

mlpC128 = MLP(input_size = 784,output_size = 10)
inputLayer = Layer(input_size = 784,output_size = 128,activation = 'relu')
hiddenLayer = Layer(input_size = 128,output_size = 128,activation = 'relu')
outputLayer = Layer(input_size = 128,output_size = 10,activation = 'none')
mlpC128.add_layer(inputLayer)
mlpC128.add_layer(hiddenLayer)
mlpC128.add_layer(outputLayer)
mlpC128.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, 256)

##PART D
# add dropout between in and out
inputLayer256 = Layer(input_size = 784,output_size = 256,activation = 'relu')
dropoutLayer256 = Layer(input_size = 256,output_size = 256,activation = 'dropout',
    dropout_prob = 0.5)
hiddenLayer256 = Layer(input_size = 256,output_size = 256,activation = 'relu')
outputLayer256 = Layer(input_size = 256,output_size = 10,activation = 'none')
mlp256_Dropout = MLP(input_size = 784,output_size = 10)
mlp256_Dropout.add_layer(inputLayer256)
mlp256_Dropout.add_layer(dropoutLayer256)
mlp256_Dropout.add_layer(outputLayer256)


mlp256_Dropout.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, B)

inputLayer128 = Layer(input_size = 784,output_size = 128,activation = 'relu')
dropoutLayer128 = Layer(input_size = 128,output_size = 128,activation = 'dropout',
    dropout_prob = 0.5)
hiddenLayer128 = Layer(input_size = 128,output_size = 128,activation = 'relu')
outputLayer128 = Layer(input_size = 128,output_size = 10,activation = 'none')
mlp128_Dropout = MLP(input_size = 784,output_size = 10)
mlp128_Dropout.add_layer(inputLayer128)
mlp128_Dropout.add_layer(dropoutLayer128)
mlp128_Dropout.add_layer(outputLayer128)


mlp128_Dropout.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, B)

inputLayer512 = Layer(input_size=784,output_size=512,activation='relu')
dropoutLayer512 = Layer(input_size=512,output_size=512,activation='dropout',dropout_prob
    =0.5)
hiddenLayer512 = Layer(input_size=512,output_size=512,activation='relu')
outputLayer512 = Layer(input_size=512,output_size=10,activation='none')
mlp512_Dropout = MLP(input_size=784,output_size=10)
mlp512_Dropout.add_layer(inputLayer512)
mlp512_Dropout.add_layer(dropoutLayer512)
mlp512_Dropout.add_layer(outputLayer512)

mlp512_Dropout.sgd(x_train, y_train, x_test, y_test, learning_rate, num_epochs, B)

fig,axs = plt.subplots(1,3,figsize = (15,5))
axs[0].plot(mlpA.epochTrainingAccuracy,label = '256 Training Accuracy', linestyle = '-')
axs[0].plot(mlpA.epochValidationAccuracy,label = '256 Validation Accuracy',linestyle = '-')
axs[0].plot(mlpB128.epochTrainingAccuracy,label = '128 Training Accuracy',linestyle = "--")
axs[0].plot(mlpB128.epochValidationAccuracy,label = '128 Validation Accuracy',linestyle = '
    --')
axs[0].plot(mlpB512.epochTrainingAccuracy,label = '512 Training Accuracy',linestyle = ':')
axs[0].plot(mlpB512.epochValidationAccuracy,label = '512 Validation Accuracy',linestyle = ':
    ')
axs[0].set_title(f'Part A & B: Accuracy Vs. Epoch: eta = {learning_rate}')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('Accuracy')
axs[0].legend()

axs[1].plot(mlpC256.epochTrainingAccuracy,label = '256 Training Accuracy', linestyle = '-')
axs[1].plot(mlpC256.epochValidationAccuracy,label = '256 Validation Accuracy',linestyle = '-
```

```
    ')
axs[1].plot(mlpC128.epochTrainingAccuracy,label = '128 Training Accuracy',linestyle = "--")
axs[1].plot(mlpC128.epochValidationAccuracy,label = '128 Validation Accuracy',linestyle = '
    --')
axs[1].plot(mlpC512.epochTrainingAccuracy,label = '512 Training Accuracy',linestyle = ':')
axs[1].plot(mlpC512.epochValidationAccuracy,label = '512 Validation Accuracy',linestyle = ':
    ')
axs[1].set_title(f'Part C: Accuracy Vs. Epoch: eta = {learning_rate}')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('Accuracy')
axs[1].legend()

axs[2].plot(mlp256_Dropout.epochTrainingAccuracy,label = '256 Training Accuracy', linestyle
    = '-')
axs[2].plot(mlp256_Dropout.epochValidationAccuracy,label = '256 Validation Accuracy',
    linestyle = '-')
axs[2].plot(mlp128_Dropout.epochTrainingAccuracy,label = '128 Training Accuracy',linestyle =
    "--")
axs[2].plot(mlp128_Dropout.epochValidationAccuracy,label = '128 Validation Accuracy',
    linestyle = '--')
axs[2].plot(mlp512_Dropout.epochTrainingAccuracy,label = '512 Training Accuracy',linestyle =
    ':')
axs[2].plot(mlp512_Dropout.epochValidationAccuracy,label = '512 Validation Accuracy',
    linestyle = ':')
axs[2].set_title(f'Part D: Accuracy Vs. Epoch: eta = {learning_rate}')
axs[2].set_xlabel('Epoch')
axs[2].set_ylabel('Accuracy')
axs[2].legend()
plt.show()
#plt.legend()
#plt.show()
#plt.plot(mlpB512.epochLoss,label = '512 hidden')
#plt.plot(mlpB128.epochLoss,label = '128 hidden')
#plt.xlabel('Epoch')
#plt.ylabel('Loss')
#plt.title(f'Loss vs Epoch; eta = {learning_rate}')
#plt.legend()
#plt.show()
```

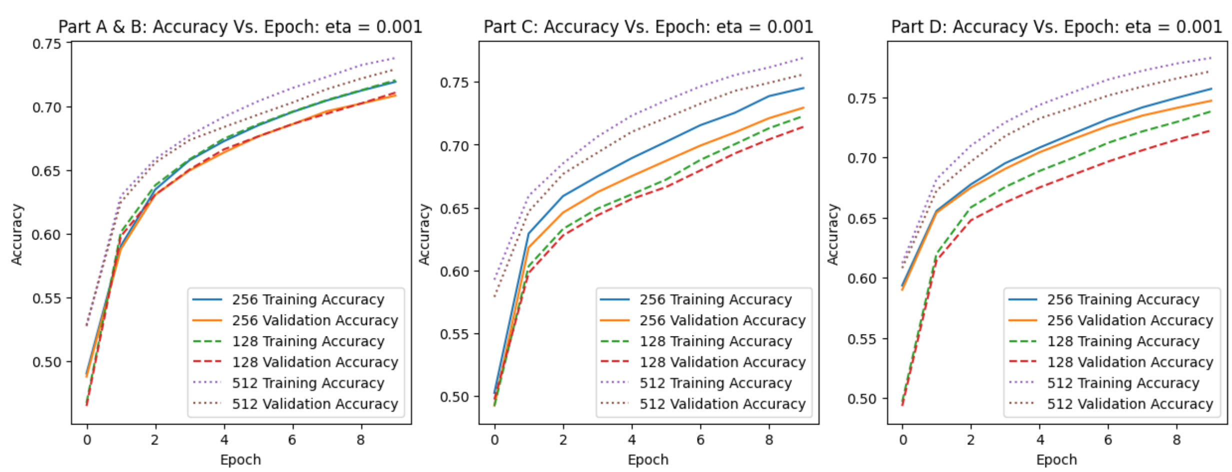

Figure 2: Accuracy Plots