Question 5

(3 points) Implementing back-propagation in Python from scratch. Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other Python IDE of your choice) and complete the missing items. In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

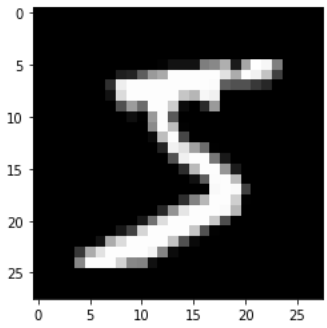
In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )

def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/
  b = x.max()
  y = np.exp(x - b)
```

```
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance 1/ $\max(n_{in}, n_{out})$.

```
import math

# Initialize weights of each layer with a normal distribution of mean 0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with mean
# 0 and standard deviation close to 1 (if biases are initialized as 0, standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...

# np.random.normal(mean, std_dev, size_tuple) as the params.
# We have 2 hidden layer so in total we have 3 weights and 3 biases to initialize.

#three numpy arrays, which represent the weights of the neural network model.
weights = [
 np.random.normal(0, 1/math.sqrt(784), (784, 32)),
 np.random.normal(0, 1/math.sqrt(32), (32, 32)),
 np.random.normal(0, 1/math.sqrt(32), (32, 10))
]
#Initializing bias arrays of zeros for the hidden layer, with the shape of the hidden layer used to determine the size of each ar
biases = [np.zeros((1,32)), np.zeros((1,32)), np.zeros((1,10))]
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

What is happening here

In the Forward Pass, we apply the neural network formula to compute the output of each layer.

We use the dot product of the input with weights, add bias, and apply an activation function to yield the output of each layer.

We use the sigmoid activation function for the hidden layers and the softmax activation function for the output layer.

The expected output is represented by y_v, and we use the cross-entropy loss function to evaluate the model's performance

```
def feed_forward_sample(sample, y):
  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
  """
```

```
  # Q2. Fill code here.
  # ...
  #Reshapes the input sample into a 1D array with 784 elements.
  a = np.reshape(sample,(1,28*28))

  # Forward pass

  z1 = np.dot(a,weights[0]) + biases[0] # dot product
  a1 = sigmoid(z1) # applying sigmoid activation function to z1
  z2 = np.dot(a1,weights[1]) + biases[1]
  a2 =sigmoid(z2)
  z3= np.dot(a2,weights[2], biases[2])
  s = softmax(z3) # applying softmax to z3

  # Calculate loss
  pred_class = np.argmax(s)
  one_hot_guess = integer_to_one_hot(pred_class,10) #Converting the predicted class into a one-hot encoded array
  y_v = integer_to_one_hot(y,10)
  loss = cross_entropy_loss(y_v,s) #Calculating the cross-entropy loss between the true label and the predicted class probabiliti


  return loss, one_hot_guess


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], 10))


  # ...
  # Q2. Fill code here to calculate losses, one_hot_guesses
  # ...
  for i in range(x.shape[0]):
    sample = np.reshape(x[i],(1,28*28))
    losses[i], one_hot_guesses[i] = feed_forward_sample(sample, y[i])

  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")

  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0], "(", correct_guess_percent, "%)")

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
    Feeding forward all test data...

    Average loss: 2.37
    Accuracy (# of correct guesses): 1144.0 / 10000 ( 11.44 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

What is happening here?

We are training a neural network model using the backpropagation algorithm to classify images from the MNIST dataset.

It performs a forward pass to compute the output of the model for a given input and a backward pass to calculate the gradients of the weights and biases with respect to the loss.

These gradients are then used to update the weights and biases to minimize the loss.

The process is repeated for each sample in the dataset to improve the model's accuracy.

```
def train_one_sample(sample, y, learning_rate=0.003):
    a = np.reshape(sample,(1,28*28))

    # We will store each layer's activations to calculate gradient
    activations = []


    # Forward pass

    # Q3. This should be the same as what you did in feed_forward_sample above.
    # ...
    Z1 = np.dot(a,weights[0]) + biases[0]
    a1 = sigmoid(Z1)
    Z2 = np.dot(a1,weights[1]) + biases[1]
    a2 =sigmoid(Z2)
    Z3= np.dot(a2,weights[2], biases[2])
    s = softmax(Z3)
    y_v = integer_to_one_hot(y,10)

    # Backward pass

    # Q3. Implement backpropagation by backward-stepping gradients through each layer.
    # You may need to be careful to make sure your Jacobian matrices are the right shape.
    # At the end, you should get two vectors: weight_gradients and bias_gradients.
    # ...

    #calculating dW1, db1, dW2, db2, dW3, db3.
    #dZ, dW, db are the derivatives of the Cost function wrt Weighted sum, Weights, Bias of the layers
    #output layer
    dZ3 = s - y_v
    dW3 = np.dot(a2.T,dZ3)
    db3 = dZ3

    #second hidden layer
    dZ2 = np.multiply(dZ3.dot(weights[2].T),dsigmoid(Z2))
    dW2 = np.dot(a1.T,dZ2)
    db2 = dZ2

    #first hidden layer
    dZ1 = np.multiply(dZ2.dot(weights[1].T),dsigmoid(Z1))
    dW1 = np.dot(a.T,dZ1)
    db1 = dZ1

    # We will store the derivative weight for each layer. -> vector 1
    weight_gradients = []
    weight_gradients.append(dW1)
    weight_gradients.append(dW2)
    weight_gradients.append(dW3)

    # We will store the derivative bias for each layer. -> vector 2
    bias_gradients = []
    bias_gradients.append(db1)
    bias_gradients.append(db2)
    bias_gradients.append(db3)

    # Update weights & biases based on your calculated gradient
    num_layers = 3
    for i in range(num_layers):
        weights[i] = weights[i] - learning_rate * weight_gradients[i]
        biases[i] = biases[i] - learning_rate * bias_gradients[i]
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  # ...
  for i in range(x_train.shape[0]):
```

```
  for i in range(x_train.shape[0]):
    train_one_sample(x_train[i], y_train[i], learning_rate)
  print("Finished training.\n")


feed_forward_test_data()

def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

for i in range(3):
  test_and_train()
```

```
Feeding forward all test data...

Average loss: 2.37
Accuracy (# of correct guesses): 1144.0 / 10000 ( 11.44 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.93
Accuracy (# of correct guesses): 7001.0 / 10000 ( 70.01 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.96
Accuracy (# of correct guesses): 6539.0 / 10000 ( 65.39 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.91
Accuracy (# of correct guesses): 6807.0 / 10000 ( 68.07 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.

Reference:

1. https://www.geeksforgeeks.org/deep-neural-net-with-forward-and-back-propagation-from-scratch-python/
2. https://github.com/kvgarimella/dl-demos

✓ 2m 11s completed at 15:29