Karan Vora (kv2154)

ECE-GY 7143 Introduction to Deep Learning, Assignment 1, Question 5

Disclosure:

I have discussed this particular problem with Rithviik Srinivasan (rs8385), Ratik Vig (rv2292) and Charmee Mehta (cm6389)

I have also used online resources like StackOverflow, GitHub repos, Kaggle Competition entries, ChatGPT, YouTube video on back-propagation by prof. Chinmay Hegde and official pytorch, numpy, matplotlib documentation.

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.

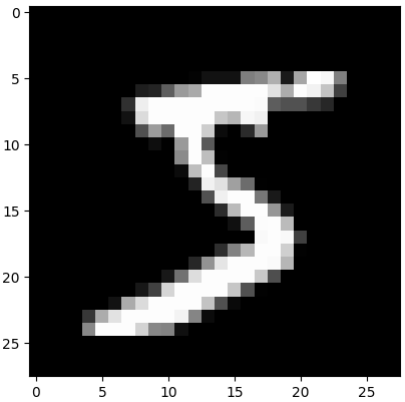
In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations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

In [ ]:
```
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')
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f0ef26f0640>
```



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.

In [ ]:
```
import numpy as np
from math import exp

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 $1/\max(n_{in}, n_{out})$.

In [ ]:
```
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.
# ...

# weights =
# biases =
InputLayerSize = 784
FirstHiddenLayer = 32
SecondHiddenLayer = 32
OutputLayer = 10

Weight1 = np.random.randn(InputLayerSize, FirstHiddenLayer) * np.sqrt(1 / max(InputLayerSize, FirstHiddenLayer))
Bias1 = np.zeros((1, FirstHiddenLayer))
Weight2 = np.random.randn(FirstHiddenLayer, SecondHiddenLayer) * np.sqrt(1 / max(FirstHiddenLayer, SecondHiddenLayer))
Bias2 = np.zeros((1, SecondHiddenLayer))
Weight3 = np.random.rand(SecondHiddenLayer, OutputLayer) * np.sqrt(1 / max(SecondHiddenLayer, OutputLayer))
Bias3 = np.zeros((1, OutputLayer))

# max_n_m = max(InputLayerSize, FirstHiddenLayer)
# Weight1 = rng.normal(loc=0, scale=1/math.sqrt(max_n_m), size=(InputLayerSize, FirstHiddenLayer))
# Bias1 = np.zeros(FirstHiddenLayer)

# max_n_m = max(FirstHiddenLayer, SecondHiddenLayer)
# Weight2 = rng.normal(loc=0, scale=1/math.sqrt(max_n_m), size=(FirstHiddenLayer, SecondHiddenLayer))
# Bias2 = np.zeros(SecondHiddenLayer)

# max_n_m = max(SecondHiddenLayer, OutputLayer)
# Weight3 = rng.normal(loc=0, scale=1/math.sqrt(max_n_m), size=(SecondHiddenLayer, OutputLayer))
# Bias3 = np.zeros(OutputLayer)

weights = [Weight1, Weight2, Weight3]
biases = [Bias1, Bias2, Bias3]
```

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).

In [ ]:

```
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.
  # ...

  sample = [sample]

  for i in range(len(weights)):
    z = np.dot(sample[-1].flatten().T, weights[i])
    z = z + biases[i]
    if(i < len(weights) - 1):
      a = sigmoid(z)
      sample.append(a)
    else:
      a = softmax(z)
      sample.append(a)

  # Compute cross entropy loss
  y_one_hot = np.zeros((1, 10))
  y_one_hot[0, y] = 1

  output = sample[-1]
  loss = cross_entropy_loss(output, y_one_hot)

  # Compute most likely class
  one_hot_guess = np.zeros((1, 10))
  one_hot_guess[0, np.argmax(output)] = 1

  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 = x[i, :]
    label = y[i]

    loss, one_hot_guess = feed_forward_sample(sample, label)

    losses[i] = loss
    one_hot_guesses[i, :] = one_hot_guess


  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...
/tmp/ipykernel_9962/1909522419.py:26: RuntimeWarning: divide by zero encountered in log
  return -np.sum(y * np.log(yHat))
Average loss: inf
Accuracy (# of correct guesses): 1025.0 / 10000 ( 10.25 %)
```

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.

In [ ]:
```
# def train_one_sample(sample, y, learning_rate=0.003):
#   a = sample.flatten()

#   # We will store each layer's activations to calculate gradient
#   activations = []
#   #activations = np.array(a)

#   # Forward pass
#   for i in range(len(weights)):
#     z = np.dot(a, weights[i])
#     z = z + biases[i]
#     if(i < len(weights) - 1):
#       a = sigmoid(z)
#       activations.append(a)
#     else:
#       a = softmax(z)
#       activations.append(a)

#   # Q3. This should be the same as what you did in feed_forward_sample above.
#   # ...

#   # Backward pass
#   delta = -(y - activations[-1]) * dsigmoid(activations[-1])
#   #print(delta.shape)
#   weight_gradients = []
#   bias_gradients = []
#   for i in reversed(range(len(weights))):
#     # print("Shape of activation: {}".format(activations[i].reshape(-1, 1).shape))
#     # print("Shape of Delta: {}".format(delta.reshape(1, -1).shape))
#     weight_gradients.insert(0, np.outer(delta, activations[i]))
#     bias_gradients.insert(0, delta)
#     if i > 0:
#       delta = (np.dot(weights[i], delta.T)) * (dsigmoid(activations[i]))

#   # 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.
#   # ...

#   # Update weights & biases based on your calculated gradient

#   for i in range(len(weights)):
#     weights[i] -= weight_gradients[i] * learning_rate
#     biases[i] -= bias_gradients[i] * learning_rate
```

In [ ]:
```
# def train_one_sample(sample, y, learning_rate=0.003):
#     a = sample.flatten()

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

#     # Forward pass
#     for i in range(len(weights)):
#         z = np.dot(activations[-1], weights[i]) + biases[i]
#         if i == len(weights) - 1:
#             a = softmax(z)
#         else:
#             a = sigmoid(z)
#         activations.append(a)

#     # Q3. This should be the same as what you did in feed_forward_sample above.
#     # ...

#     # Backward pass
#     weight_gradients = []
#     bias_gradients = []
#     for i in reversed(range(len(weights))):
#         if i == len(weights) - 1:
#             delta *= dsigmoid(activations[i])
#         else:
#             delta = np.dot(delta, weights[i+1].T) * dsigmoid(activations[i])
#         weight_gradients.insert(0, np.dot(activations[i].reshape(-1, 1), delta))
#         bias_gradients.insert(0, delta)

#     # 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.
#     # ...

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

In [ ]:
```
def train_one_sample(sample, y, learning_rate=0.003):
    # a = sample.flatten()
    a = np.reshape(sample,(1,28*28))

    # We will store each layer's activations to calculate gradient
    activations = []
    #activations = np.array(a)

    # Forward pass
    FirstLayer = np.dot(a, weights[0]) + biases[0]
    FirstLayerActivation = sigmoid(FirstLayer)
    SecondLayer = np.dot(FirstLayerActivation, weights[1]) + biases[1]
    SecondLayerActivation = sigmoid(SecondLayer)
```

```
    ThirdLayer = np.dot(SecondLayerActivation, weights[2]) + biases[2]
    ThirdLayerActivation = softmax(ThirdLayer)

    # Q3. This should be the same as what you did in feed_forward_sample above.
    # ...
    prediction = np.argmax(ThirdLayerActivation)
    one_hot_guess = integer_to_one_hot(prediction,10)
    loss = cross_entropy_loss(integer_to_one_hot(y, 10), ThirdLayerActivation)

    # 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.
    # ...

    LastLayer = ThirdLayerActivation - integer_to_one_hot(y,10)
    LastLayerWeights = SecondLayerActivation.T.dot(LastLayer)
    LastLayerBiases = LastLayer

    SecondLastLayer = LastLayer.dot(weights[2].T)
    SecondLastLayerUpdate = np.multiply(SecondLastLayer, dsigmoid(SecondLayer))
    SecondLastLayerWeights = FirstLayerActivation.T.dot(SecondLastLayerUpdate)
    SecondLayerBiases = SecondLastLayer

    FirstLastLayer = SecondLastLayer.dot(weights[1].T)
    FirstLayerUpdate = np.multiply(FirstLastLayer, dsigmoid(FirstLayer))
    FirstLayerWeights = a.T.dot(FirstLayerUpdate)
    FirstLayerBiases = FirstLastLayer

    weight_gradients = [FirstLayerWeights, SecondLastLayerWeights, LastLayerWeights]
    bias_gradients = [FirstLayerBiases, SecondLayerBiases, LastLayerBiases]

    for i in range(3):
        weights[i] = weights[i] - learning_rate * weight_gradients[i]
        biases[i] = biases[i] - learning_rate * bias_gradients[i]

    return loss
```

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

Q4. Train your model for 3 epochs.

In [ ]:
```
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 values in range(x_train.shape[0]):
    train_one_sample(x_train[values], y_train[values], 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...
/tmp/ipykernel_9962/1909522419.py:26: RuntimeWarning: divide by zero encountered in log
  return -np.sum(y * np.log(yHat))
Average loss: inf
Accuracy (# of correct guesses): 1025.0 / 10000 ( 10.25 %)

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

Feeding forward all test data...

Average loss: inf
Accuracy (# of correct guesses): 4175.0 / 10000 ( 41.75 %)

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

Feeding forward all test data...

Average loss: inf
Accuracy (# of correct guesses): 4934.0 / 10000 ( 49.34 %)

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

Feeding forward all test data...

Average loss: inf
Accuracy (# of correct guesses): 5664.0 / 10000 ( 56.64 %)
```

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.