dlf23-hw1-prob5

September 21, 2023

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.

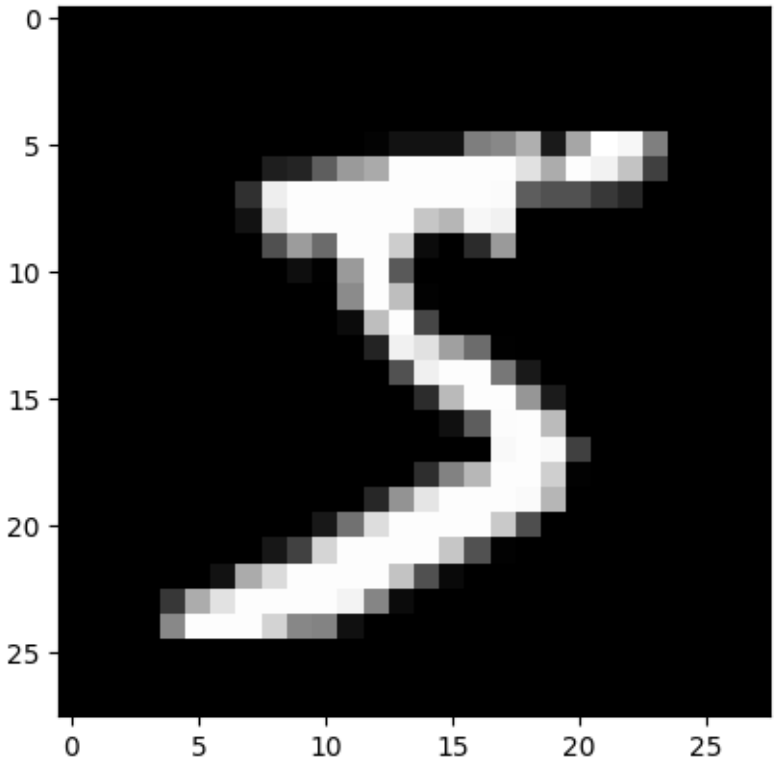
For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

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.

[7]:
```
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") #Loading dataset

plt.imshow(x_train[0],cmap='gray'); #One of the data points
```



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.

I understand what is happening here but I found this demo helped a lot: https://www.youtube.com/watch?v=w8yWXqWQYmU

Everything I referenced for help outside of this course is from that video.

[8]:
```
import numpy as np

def sigmoid(x): #For hidden layers
  # 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): #For output layer
  # 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): #Loss function
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max): #Categorical to numerical
  # 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 $1/\max(n_{in}, n_{out})$.

[9]:
```
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 = []
```

```
l1w=rng.normal(loc=0,scale=1/math.sqrt(784),size=(784,32)) #784 input and 32
 ↪output
l1b=np.zeros(32) #A bunch of zeroes
l2w=rng.normal(loc=0,scale=1/math.sqrt(32),size=(32,32))
l2b=np.zeros(32)
l3w=rng.normal(loc=0,scale=1/math.sqrt(32),size=(32,10)) #10 outputs
l3b=np.zeros(10)

weights.append(l1w)
weights.append(l2w)
weights.append(l3w)
biases.append(l1b)
biases.append(l2b)
biases.append(l3b)
```

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).

[10]:
```
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.
  # ...
  dummy=sample
  for i in range(len(weights)): #Each weight set
    z=np.matmul(dummy,weights[i])+biases[i] #dataset times weights plus biases
    if i==(len(weights)-1): #last weight set, output layer
        activation=softmax(z)
        output=activation
    else:
        activation=sigmoid(z)
    dummy=activation #Updates input for loop

  one_hot=integer_to_one_hot(y,10) #Make categorical data numerical
  loss=cross_entropy_loss(one_hot,output) #Loss function
  two_hot=integer_to_one_hot(np.argmax(output),10)
```

```
  return loss, two_hot


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]): #Makes images into vectors
      dummy=x[i].reshape(-1)
      label=y[i]
      loss,one_hot=feed_forward_sample(dummy,label)
      losses[i]=loss
      one_hot_guesses[i]=one_hot

  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): 880.0 / 10000 ( 8.80 %)
```

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.

[11]:
```
def train_one_sample(sample, y, learning_rate=0.003):
  a = sample.flatten()

  # 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.
  # ...
  weight_gradients=[None]*3 #3 of each
  bias_gradients=[None]*3
  # Forward pass
  dummy=a
  for i in range(len(weights)):
    dummy2=np.matmul(dummy,weights[i])+biases[i] #Same as before just with␣
 ↪activations
    if i==(len(weights)-1):
        activation=softmax(dummy2)
        output=activation
    else:
        activation=sigmoid(dummy2)
    dummy=activation
    activations.append(dummy)

# Backward pass
  one_hot=integer_to_one_hot(y,10)

  for index in range(len(weights)): #Getting the gradients, derivatives are␣
 ↪written out
      i=len(weights)-index-1
      if index==0:
          dl_dz=activations[i]-one_hot #For output layer
      else:
          dl_dac=np.matmul(weights[i+1],dl_dz.T).reshape(-1) #Hidden layers
          dl_dz=dl_dac*(activations[i]*(1-activations[i]))


      if i==0: #To the end
          dl_dw=np.matmul(a[:,np.newaxis],dl_dz[np.newaxis,:]) #First layer
      else:
          dl_dw=np.matmul(activations[i-1][:,np.newaxis],dl_dz[np.newaxis,:])␣
 ↪#Others
```

```
        dl_db=dl_dz
        weight_gradients[i]=dl_dw
        bias_gradients[i]=dl_db


    # 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].flatten() * learning_rate
```

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

Q4. Train your model for 3 epochs.

[12]:
```
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 j in range(x_train.shape[0]): #3 epochs
        train_one_sample(x_train[j],y_train[j],learning_rate=0.003)
    feed_forward_training_data()

    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): 880.0 / 10000 ( 8.80 %)

Training for one epoch over the training dataset…
```

```
Feeding forward all training data…

Average loss: 1.04
Accuracy (# of correct guesses): 39068.0 / 60000 ( 65.11 %)

Finished training.

Feeding forward all test data…

Average loss: 1.04
Accuracy (# of correct guesses): 6505.0 / 10000 ( 65.05 %)

Training for one epoch over the training dataset…
Feeding forward all training data…

Average loss: 0.84
Accuracy (# of correct guesses): 43749.0 / 60000 ( 72.91 %)

Finished training.

Feeding forward all test data…

Average loss: 0.83
Accuracy (# of correct guesses): 7330.0 / 10000 ( 73.30 %)

Training for one epoch over the training dataset…
Feeding forward all training data…

Average loss: 0.9
Accuracy (# of correct guesses): 41467.0 / 60000 ( 69.11 %)

Finished training.

Feeding forward all test data…

Average loss: 0.88
Accuracy (# of correct guesses): 6994.0 / 10000 ( 69.94 %)
```

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.

[ ]: