Final Project - MAT4373

Tanner Giddings

2024-04-10

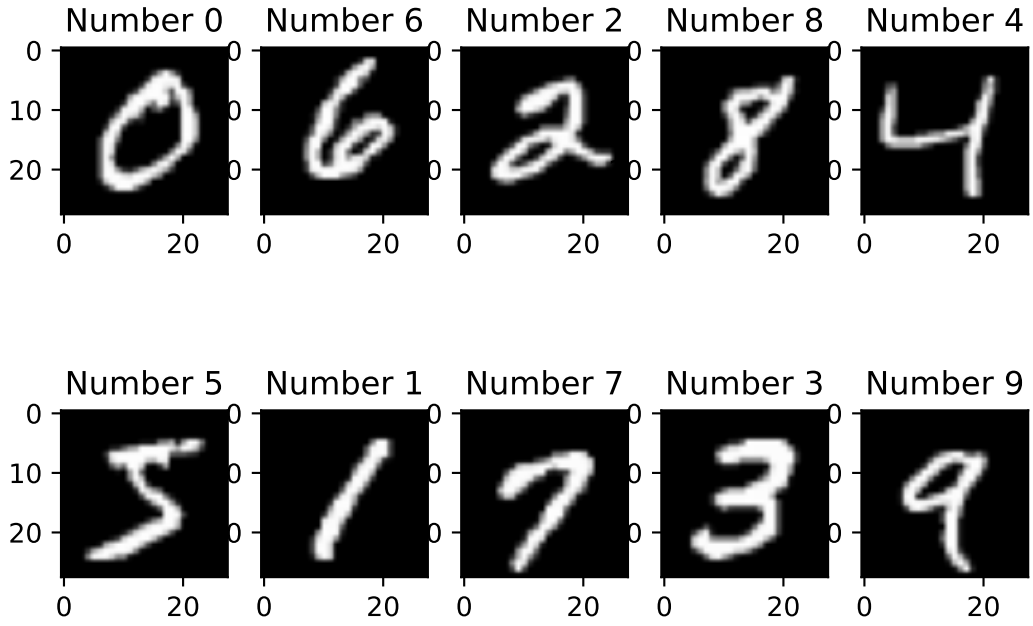
```
#!pip install matplotlib
#!pip install pandas
#!pip install tqdm
```

Question 1

```
import matplotlib.pyplot as plt
from scipy.io import loadmat
import pandas as pd

data = loadmat('mnist_all.mat')
data.keys()
```

```
## dict_keys(['__header__', '__version__', '__globals__', 'train0', 'test0', 'train1', 'test1', 'train2
```

```
fig, ax = plt.subplots(2, 5)
for i in range(10):
  ax[i % 2][i % 5].imshow(data[f"train{i}"][0].reshape((28,28)), cmap='gray')
  ax[i % 2][i % 5].set_title(f"Number {i}")
plt.show()
```




```
#Cleaning the data
#Taking too long to run
"""
import pandas as pd

def clean(D):
  df = []
  for i in range(0,10):
    dict = {"pixel" + str(j) : [D[f"train{i}"][k][j] for k in range(len(D[f"train{i}"]))] for j in rang
    dict['Y'] = [i] * len(dict['pixel0'])
    df.append(pd.DataFrame(dict))
  return pd.concat(df)

data_clean = clean(data)
data_clean
"""
```

```
## '\nimport pandas as pd\n\ndef clean(D):\n  df = []\n  for i in range(0,10):\n    dict = {"pixel" + s
```

```
def clean(D):
  d = {}
  for i in range(0, 10):
    d["train" + str(i)] = [elem / 255 for elem in D["train" + str(i)] for i in range(0,10)]
  return d
data_clean = clean(data)
```

Question 2

```
import numpy as np
from math import log

def softmax(x):
  e_x = np.exp(x)
  return e_x / e_x.sum()

def ReLU(x):
  return [max(0, elem) for elem in x]

def forward(X, w, b):
  return np.matmul(X, w) + b

def predict(X, w, b):
  return softmax(forward(X, w, b))

def cross_entropy(y, p):
  return sum([-y[i] * log(p[i]) - (1-y[i]) * log(1-p[i])])

w1 = np.random.rand(28*28, 9)
b1 = np.random.rand(9)

print(predict(np.reshape(data_clean['train0'][0], (28*28)), w1, b1), "\nThis is the predicted outputs fo
```

```
## [1.38424090e-02 2.29639338e-04 8.79357618e-01 7.34235469e-02
##  2.75275702e-04 2.98060643e-04 1.54456760e-03 2.96705601e-02
##  1.35832284e-03]
## This is the predicted outputs for an untrained model
```

Part 3

Let $L(\overrightarrow{y}, \overrightarrow{p}) = -\sum_{i=0}^{9}(y_i log(p_i) + (1-y_i) log(1-p_i))$ be the loss function at the end of the network for one sample, the variable i here represents each class, and y_i represents the one-hot encoded value of the class, e.g. $y_i = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ represents 2. Also, p_i is the output of the softmax function.

$$\frac{\delta}{\delta p_i} L(\overrightarrow{y}, \overrightarrow{p}) = \frac{\delta}{\delta p_i}(-\sum_{i=0}^{9}(y_i log(p_i) + (1-y_i)log(1-p_i)))$$

$$= -\sum_{i=0}^{9}(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i})$$

$$= -\sum_{i=0}^{9}(\frac{y_i - y_i p_i - p_i + y_i p_i}{p_i(1-p_i)})$$

$$= -\sum_{i=0}^{9}\frac{y_i - p_i}{p_i(1-p_i)}$$

Suppose the class of the sample is k, so $y_k = 1$ and $y_i = 0\ \forall i \neq k$

$$= \sum_{i=0, i\neq k}^{9}\frac{1}{1-p_i} - \frac{1}{p_k}$$

Since p_i is the output of the softmax function, $p_i = \frac{e^{z_i}}{\sum_{j=0}^{9} e^{zj}}$. Here we find the derivative of p_i in relation to $\overrightarrow{z} = \sigma(w\overrightarrow{x}_i + \overrightarrow{b})$, with $x_i \in \mathbb{R}^n$, $w \in \mathbb{R}^{9\times n}$, $b \in \mathbb{R}^9$ and $\sigma(x) = \max\{0, x\}$:

$$\frac{\delta}{\delta z_i} p_i = \frac{\delta}{\delta z_i}\frac{e^{z_i}}{\sum_{j=0}^{9} e^{zj}}$$

$$= \frac{e^{z_i}\sum_{j=0}^{9} e^{zj} - (e^{z_i})^2}{(\sum_{j=0}^{9} e^{zj})^2}$$

$$= \frac{e^{z_i}}{\sum_{j=0}^{9} e^{zj}} - ()^2$$

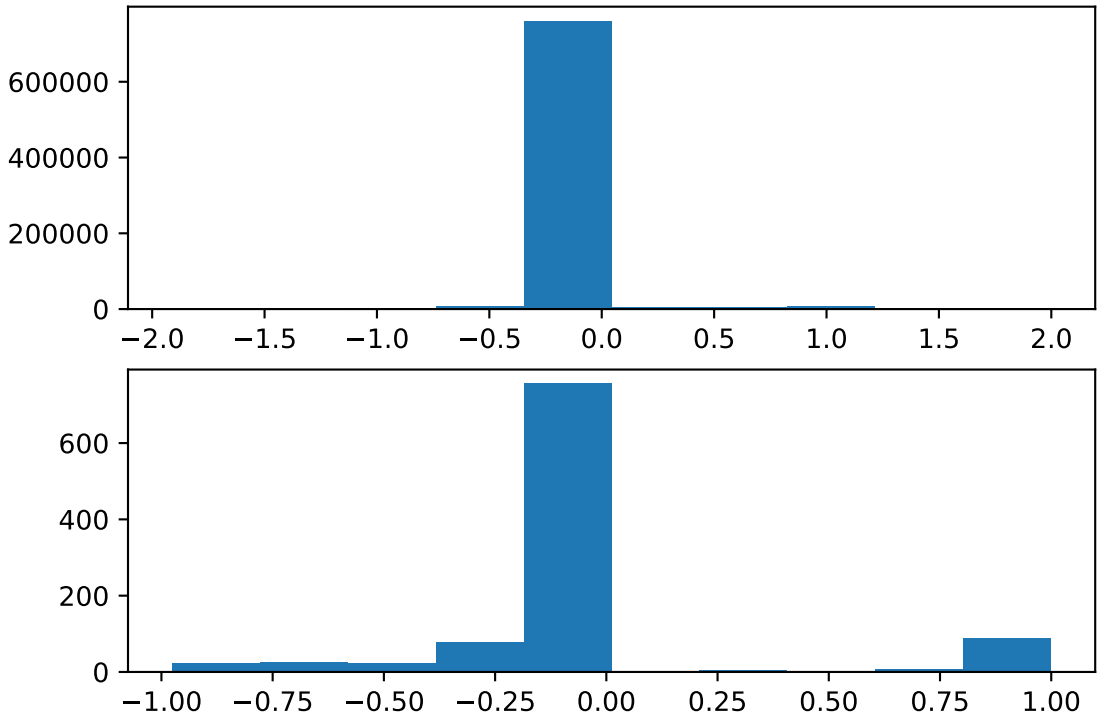
```
#This is the implementation of the gradient of the cost function for b (db) and w (dw)
def backprop(y, p, xi):
  db = y-p
  dw = np.matmul(np.array([xi]).T, np.array([db]))
  return db, dw

def update_parameters(w, b, db, dw, alpha):
  w = w - alpha * dw
  b = b - alpha * db
  return w, b
```

Question 4

```
def create_dataframe(data, name):
    df = []
    for i in range(10):
        cur_df = pd.DataFrame({"pixel" + str(j) : [elem[j] / 255 for elem in data[name+str(i)]] for j in
        cur_df['Y'] = [i for j in range(len(cur_df))]
        df.append(cur_df)
    return pd.concat(df).reset_index(drop=True)

train = create_dataframe(data, 'train')
test = create_dataframe(data, 'test')
```

```
def one_hot(x):
  return np.array([int(i == x) for i in range(10)])

def numerical_approximation_W(x, w, b, y, h):
  a, b = np.shape(w)
  approximations = [[0 for i in range(b)] for i in range(a)]
  for i in range(a):
    for j in range(b):
      h_matrix = np.zeros(np.shape(w))
      h_matrix[i][j] += h
      approximations[i][j] = (cross_entropy(y, predict(x, w + h_matrix, b)) - cross_entropy(y, predict(
  return approximations

def numerical_approximation_b(x, w, b, y, h):
  return (cross_entropy(y, predict(x, w, b + h)) - cross_entropy(y, predict(x, w, b-h))) / (2 * h)

def run(X, Y, h=0.01, iterations = 100):
  w = np.random.rand(28*28, 10)
  b = np.random.rand(10)
  current_dw = 0
  current_db = 0
  batch_counter = 0
  estimation_vs_numerical = {'w' : [], 'b' : []}

  for i in range(iterations):
    x_i = X.loc[i].to_numpy()
    y_pred = predict(x_i, w, b)
    y = one_hot(Y[i])
    db, dw = backprop(y, y_pred, x_i)
    estimation_vs_numerical['w'].append((dw - numerical_approximation_W(x_i, w, b, y, h)).flatten())
    estimation_vs_numerical['b'].append((db - numerical_approximation_b(x_i, w, b, y, h)).flatten())
    w, b = update_parameters(w, b, db, dw, 0.000001)
  return estimation_vs_numerical


X = train.sample(frac=1).reset_index(drop=True)
Y = X['Y']
X = X.drop(columns=['Y'])
estimation_vs_numerical = run(X, Y)
```

```
def combine(L):
  new_L = []
  for elem in [list(elem) for elem in L]:
    new_L += elem
  return new_L

fig, ax = plt.subplots(2)
ax[0].hist(combine(estimation_vs_numerical['w']))
ax[1].hist(combine(estimation_vs_numerical['b']))
plt.show()
```



We can see in the graphs above that the numerical approximation of the gradient approximates closely the approximation of the gradient found with back propagation since the differences are closely distributed around 0.