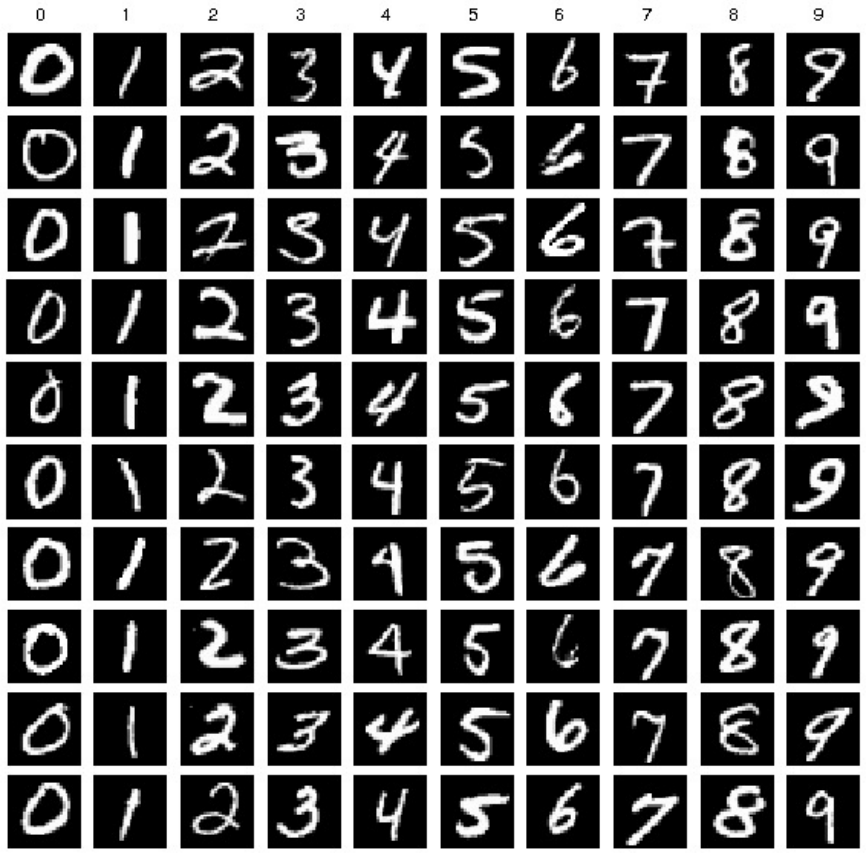
CS777 – Term Project

Image Classification using Neural Network *(from Scratch)*

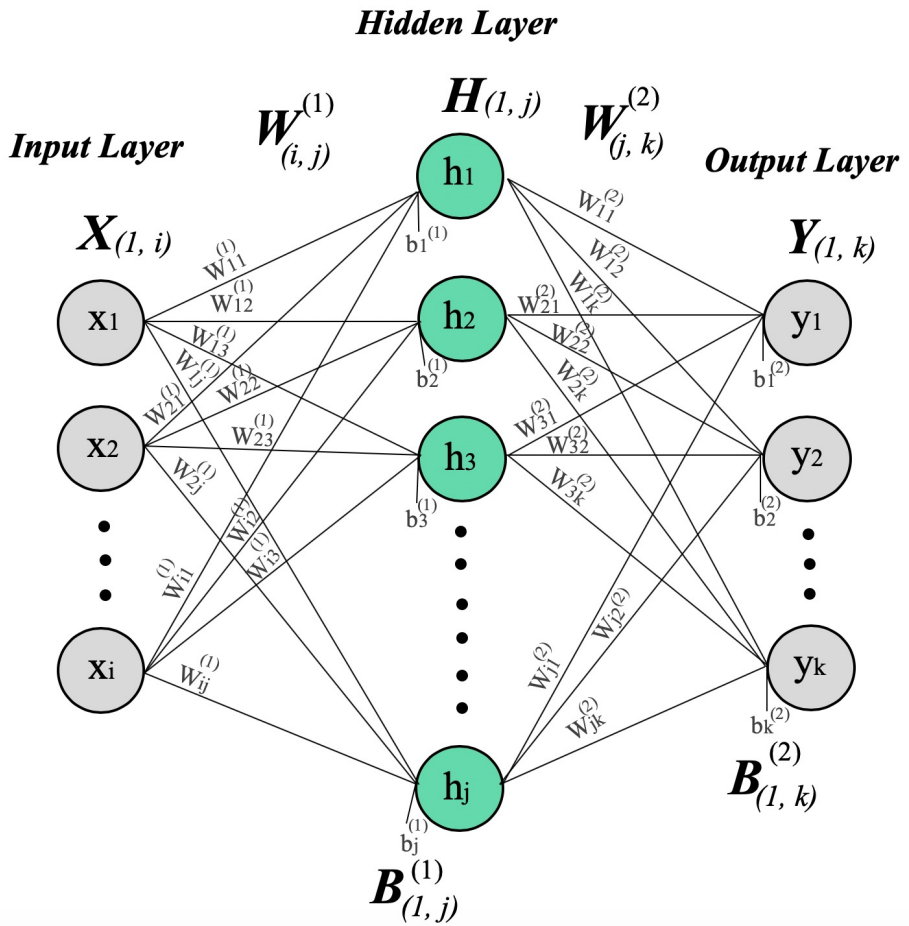
Marvin Martin – Spring 2021

Dataset - MNIST



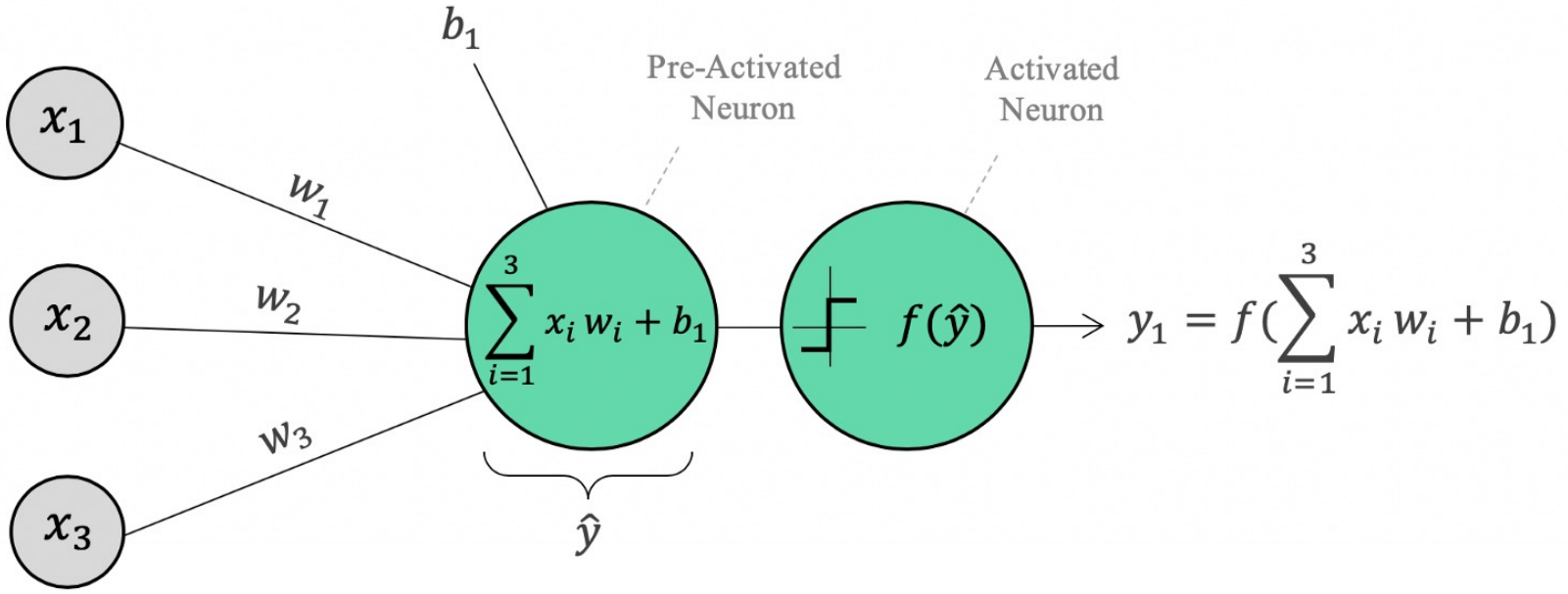
- Training set of 60,000 examples
- Test set of 10,000 examples
- 28 x 28 pixels
- Grey Scale

Model – Fully Connect Neural Network



- Input Layer i=784
- Hidden Layer j=64
- Output Layer k=10
- Parameters:
  - W1 *shape (784, 64)*
  - B1 *shape (1, 64)*
  - W2 *shape (64, 10)*
  - B2 *shape (1, 10)*

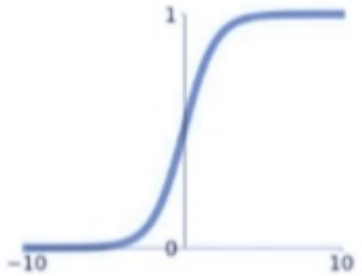
Neuron


b_1
Pre-Activated Neuron
Activated Neuron
x_1
x_2
x_3
w_1
w_2
w_3
$\sum_{i=1}^{3} x_i w_i + b_1$
$f(\hat{y})$
$\hat{y}$
$y_1 = f(\sum_{i=1}^{3} x_i w_i + b_1)$

Activation

Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



tanh

$$\tanh(x)$$

Forward Propagation

$$\hat{H} = X \cdot W^{(1)} + B^{(1)}$$

$$\underset{(l,j)}{\hat{H}} = \underset{(l,i)}{\begin{bmatrix} x_1 & x_2 & \cdots & x_i \end{bmatrix}} \cdot \underset{(i,j)}{\begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \cdots & w_{1j}^{(1)} \\ w_{21}^{(1)} & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ w_{i1}^{(1)} & \cdot & \cdots & w_{ij}^{(1)} \end{bmatrix}} + \underset{(l,j)}{\begin{bmatrix} b_1^{(1)} & b_2^{(1)} & \cdots & b_j^{(1)} \end{bmatrix}} \quad (1)$$

$$\hat{Y} = H \cdot W^{(2)} + B^{(2)}$$

$$\underset{(l,k)}{\hat{Y}} = \underset{(l,j)}{\begin{bmatrix} h_1 & h_2 & \cdots & h_j \end{bmatrix}} \cdot \underset{(j,k)}{\begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & \cdots & w_{1k}^{(2)} \\ w_{21}^{(2)} & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ w_{j1}^{(2)} & \cdot & \cdots & w_{jk}^{(2)} \end{bmatrix}} + \underset{(l,k)}{\begin{bmatrix} b_1^{(2)} & b_2^{(2)} & \cdots & b_k^{(2)} \end{bmatrix}} \quad (2)$$

$$\hat{H} \Rightarrow Activation \Rightarrow H$$

$$\hat{Y} \Rightarrow Activation \Rightarrow Y$$

$$\underset{(l,j)}{H} = Activation(\underset{(l,j)}{\hat{H}})$$

$$\underset{(l,j)}{H} = \begin{cases} h_1 = f(x_1 w_{11}^{(1)} + x_2 w_{21}^{(1)} + \cdots + x_i w_{i1}^{(1)} + b_1^{(1)}) \\ \cdot \\ \cdot \\ h_j = f(x_1 w_{1j}^{(1)} + x_2 w_{2j}^{(1)} + \cdots + x_i w_{ij}^{(1)} + b_j^{(1)}) \end{cases}$$

$$\underset{(l,k)}{Y} = Activation(\underset{(l,k)}{\hat{Y}})$$

$$\underset{(l,k)}{Y} = \begin{cases} y_1 = f(h_1 w_{11}^{(2)} + h_2 w_{21}^{(2)} + \cdots + h_j w_{j1}^{(2)} + b_1^{(2)}) \\ \cdot \\ \cdot \\ y_k = f(h_1 w_{1k}^{(2)} + h_2 w_{2k}^{(2)} + \cdots + h_j w_{jk}^{(2)} + b_k^{(2)}) \end{cases}$$

Backward Propagation

$$E = \frac{2}{n}(Y - Y*)^2$$

$$\frac{\partial E}{\partial B_2} = (Y - Y*) \times f'(\hat{Y})$$

$$\frac{\partial E}{\partial W_2} = H^t \cdot \frac{\partial E}{\partial B_2}$$

$$\frac{\partial E}{\partial B_1} = (\frac{\partial E}{\partial B_2} \cdot W_2^t) \times f'(\hat{H})$$

$$\frac{\partial E}{\partial W_1} = X^t \cdot \frac{\partial E}{\partial B_1}$$

https://medium.com/swlh/mathematics-behind-basic-feed-forward-neural-network-3-layers-python-from-scratch-df88085c8049

Mini Batch Gradient

For epoch in 100 :

For each mini batch of size (~42000):

$$W \leftarrow W - \alpha \frac{\partial E}{\partial W}$$

$$B \leftarrow B - \alpha \frac{\partial E}{\partial B}$$

Code: Data generation

Use *python3 generate_data.py*

```python
import numpy as np
import os

from keras.datasets import mnist
from keras.utils import np_utils

print('Start downloading dataset...')
# load MNIST from server
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# training data : 60000 samples
# reshape and normalize input data
x_train = x_train.reshape(x_train.shape[0], 1, 28*28)
x_train = x_train.astype('float32')
x_train /= 255
# encode output which is a number in range [0,9] into a vector of size 10
# e.g. number 3 will become [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_train = np_utils.to_categorical(y_train)
```

```python
# same for test data : 10000 samples
x_test = x_test.reshape(x_test.shape[0], 1, 28*28)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test)

if not os.path.exists('data/'):
    os.makedirs('data/')


np.savetxt('data/mnist_images_train.csv', x_train.reshape(len(x_train),784).tolist())
np.savetxt('data/mnist_images_test.csv', x_test.reshape(len(x_test),784).tolist())
np.savetxt('data/mnist_labels_train.csv', y_train.tolist())
np.savetxt('data/mnist_labels_test.csv', y_test.tolist())

print('Dataset downloaded.')

print('Data is located here:', os.getcwd() + '\data')
```

Code: Spark Data Loading

```
txt_train_images = sc.textFile(sys.argv[1], 1)
x_train = txt_train_images.map(lambda x : np.fromstring(x, dtype=float, sep=' ')\
                          .reshape(1, 784))\
                          .zipWithIndex()\
                          .map(lambda x: (str(x[1]), x[0]))

txt_train_labels = sc.textFile(sys.argv[2], 1)
y_train = txt_train_labels.map(lambda x : np.fromstring(x, dtype=float, sep=' ')\
                          .reshape(1, 10))\
                          .zipWithIndex()\
                          .map(lambda x: (str(x[1]), x[0]))

txt_test_images = sc.textFile(sys.argv[3], 1)
x_test = txt_test_images.map(lambda x : np.fromstring(x, dtype=float, sep=' ')\
                        .reshape(1, 784))\
                        .zipWithIndex()\
                        .map(lambda x: (str(x[1]), x[0]))

txt_test_labels = sc.textFile(sys.argv[4], 1)
y_test = txt_test_labels.map(lambda x : np.fromstring(x, dtype=float, sep=' ')\
                        .reshape(1, 10))\
                        .zipWithIndex()\
                        .map(lambda x: (str(x[1]), x[0]))

train_rdd = x_train.join(y_train).map(lambda x: x[1])
test_rdd = x_test.join(y_test).map(lambda x: x[1])
train_rdd.cache()
```

One data point:

([0.11, 0.32, …., 0.91], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0])

X -> Shape (1, 784) Y* -> Shape (1, 10)

Type: NumPy arrays

Code: Training Loop (1/2)

```
for i in range(num_iteration):


    gradientCostAcc = train_rdd\
                        .sample(False,0.7)\
                        .map(lambda x: (x[0], preforward(x[0], W1, B1), x[1]))\
                        .map(lambda x: (x[0], x[1], activation(x[1], tanh), x[2]))\
                        .map(lambda x: (x[0], x[1], x[2], preforward(x[2], W2, B2), x[3]))\
                        .map(lambda x: (x[0], x[1], x[2], x[3], activation(x[3], sigmoid), x[4]))\
                        .map(lambda x: (x[0], x[1], x[2], sse(x[4], x[5]), derivativeB2(x[4], x[5], x[3], sigmoid_prime), int(np.argmax(x[4]) == np.argmax(x[5]))))
                        .map(lambda x: (x[0], x[1], x[3], x[4],  derivativeW2(x[2], x[4]) ,x[5]))\
                        .map(lambda x: (x[0], x[2], x[3], x[4],  derivativeB1(x[1],  x[3], W2, tanh_prime) ,x[5]))\
                        .map(lambda x: (x[1], x[2], x[3], x[4], derivativeW1(x[0], x[4]) ,x[5], 1)) \
                        .reduce(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2], x[3] + y[3], x[4] + y[4], x[5] + y[5], x[6] + y[6]))

    # Cost and Accuarcy of the mini batch
    n = gradientCostAcc[-1] # number of images in the mini batch
    cost = gradientCostAcc[0]/n # Cost over the mini batch
    acc = gradientCostAcc[5]/n # Accuarcy over the mini batch

     # Add to history
    cost_history.append(cost)
    acc_history.append(acc)
```

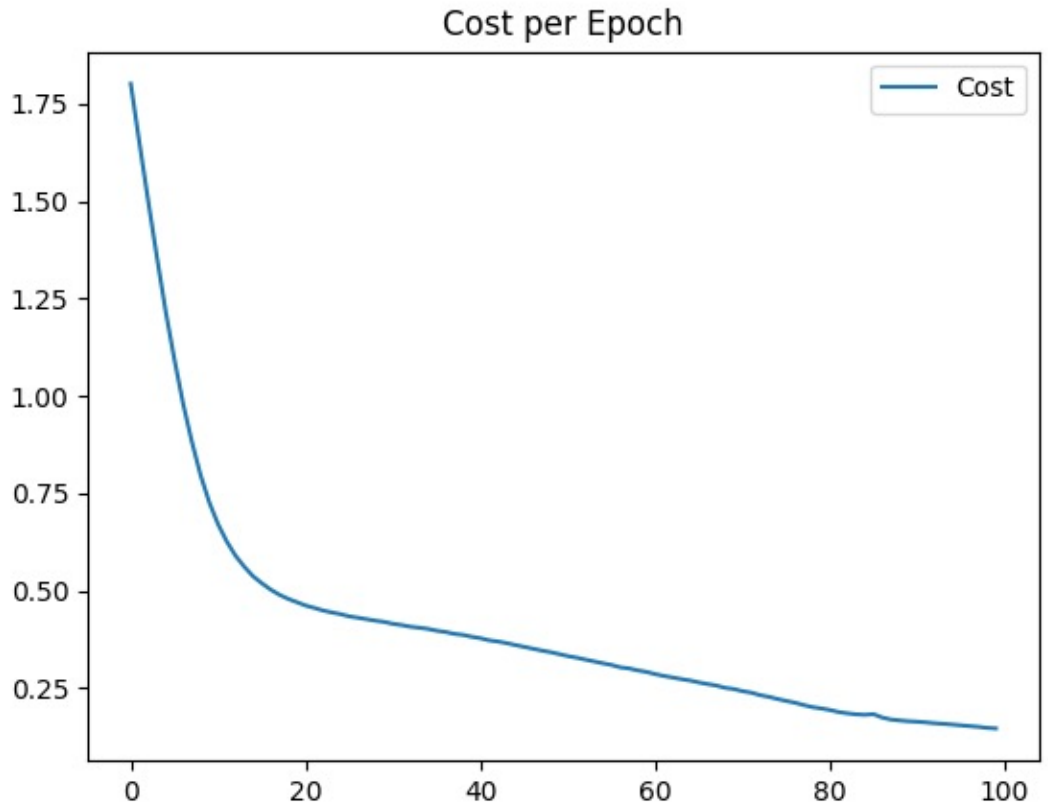
Code: Training Loop (2/2)

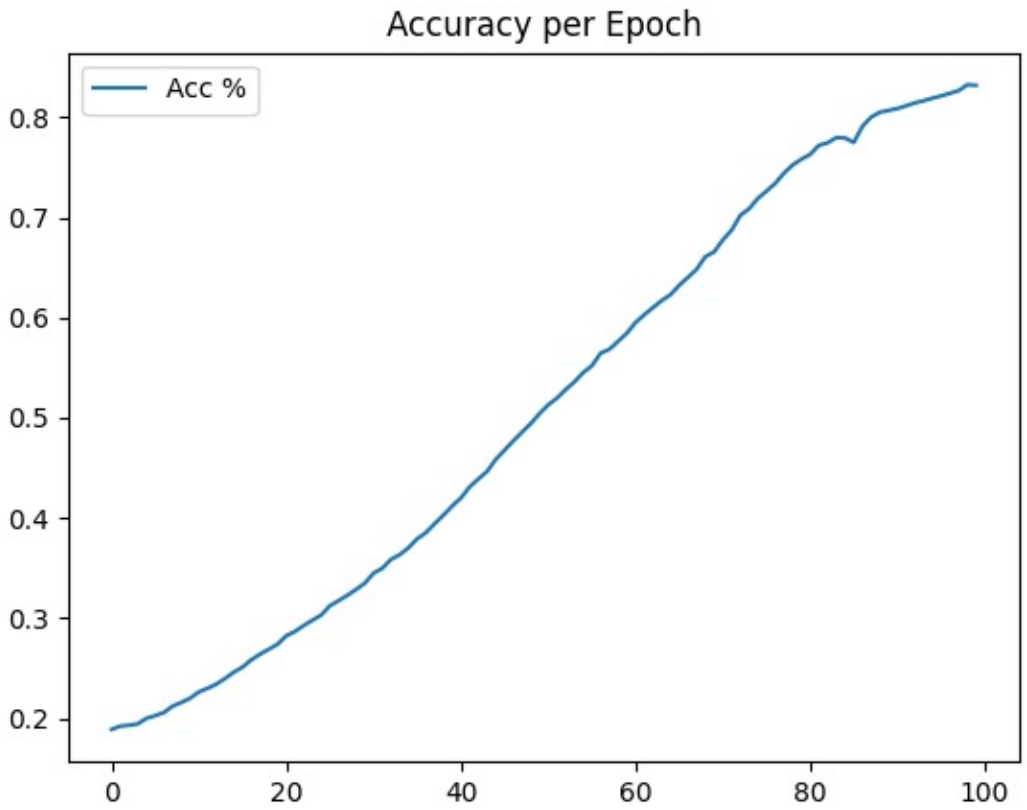
```
    # Bold Driver technique to dynamically change the learning rate
    if len(cost_history) > 1:
        if cost_history[i] < cost_history[i-1]:
            learningRate *= 1.05 #Better than last time
        else:
            learningRate *= 0.5 #Worse than last time

    # Extract gradiends
    DB2 = gradientCostAcc[1]/n
    DW2 = gradientCostAcc[2]/n
    DB1 = gradientCostAcc[3]/n
    DW1 = gradientCostAcc[4]/n

    # Update parameter with new learning rate and gradients using Gradient Descent
    B2 -= learningRate * DB2
    W2 -= learningRate * DW2
    B1 -= learningRate * DB1
    W1 -= learningRate * DW1

    # Display performances
    print(f"   Epoch {i+1}/{num_iteration} | Cost: {cost_history[i]} | Acc: {acc_history[i]*100} | Batchsize:{n}")

print("Training end..")
```

Code: Test Evaluation

```
# Use the trained model over the Testset and get Confusion matrix per class
metrics = test_rdd.map(lambda x: get_metrics(np.round(predict(x[0], W1, B1, W2, B2)), x[1]))\
                  .reduce(lambda x, y: x + y)

# For each class give TP, FP, FN, TN and precision, and recall, and F1 score
save_metrics = []
for label, label_metrics in enumerate(metrics):

    print(f"\n---- Digit {label} ------\n")
    tn, fp, fn, tp = label_metrics.ravel()
    print("TP:", tp, "FP:", fp, "FN:", fn, "TN:", tn)

    precision = tp / (tp + fp + 0.000001)
    print(f"\nPrecision : {precision}")

    recall = tp / (tp + fn + 0.000001)
    print(f"Recall: {recall}")

    F1 = 2 * (precision * recall) / (precision + recall + 0.000001)
    print(f"F1 score: {F1}")

    save_metrics.append((precision, recall, F1))
```

Train Results

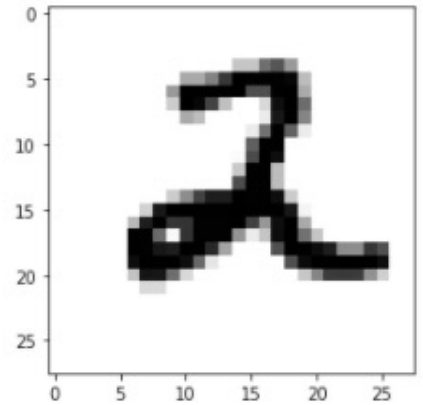




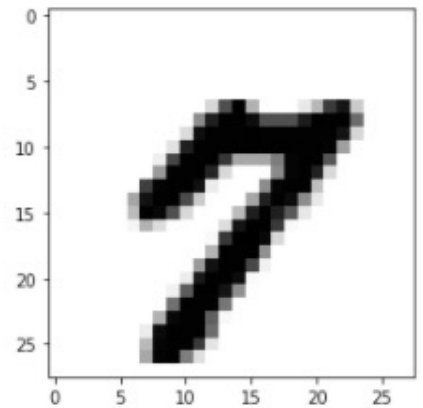
Test Results

```
# Digit 0 (precision, recall, F1) --> GOOD
(0.9675550394350463, 0.8520408154571012, 0.9061308086825957)

# Digit 1 (precision, recall, F1) --> GOOD
(0.6700477322971076, 0.9894273119035881, 0.7990034311500359)

# Digit 2 (precision, recall, F1) --> GOOD
(0.6276816604652722, 0.878875968140624, 0.7323370183555452)

# Digit 3 (precision, recall, F1) --> VERY BAD
(0.0, 0.0, 0.0)

# Digit 4 (precision, recall, F1) --> BAD
(0.21938661289022013, 0.9979633391059437, 0.35969873173559086)

# Digit 5 (precision, recall, F1) --> BAD
(0.1863100634210539, 0.9215246626440307, 0.30995447123030956)

# Digit 6 (precision, recall, F1) --> BAD
(0.10699128879752164, 0.9999999989561587, 0.1933008745760054)

# Digit 7 (precision, recall, F1) --> GOOD
(0.9911816561002087, 0.5466926064720888, 0.7047017352426795)

# Digit 8 (precision, recall, F1) --> BAD
(0.21745316570608175, 0.941478438458441, 0.3533034900409847)

# Digit 9 (precision, recall, F1) --> BAD
(0.99999950000025, 0.001982160553040475, 0.003956474777450049)
```

pred: 2, prob: 0.76 true: 2



pred: 7, prob: 0.73 true: 7



Improvement

To get a Better Model:

- Regularization L1 or/and L2
- Learning rate Decay
- Add another hidden Layer (mathematics are getting complex)
- Implement Dropout

Improve/optimize the spark Implementation:

- Use tree aggregate instead of Reduce
- Use mapPartitionWithIndex followed by a filter instead of sample
- Try to reduce the number of map

Conclusion

This project was a real **challenge** in terms of implementation, as Spark is not optimal for implementing neural networks.

I really enjoyed using my newly acquired **knowledge** from past assignments to apply it to an algorithm that **fascinate** me.

I also like to start building the algorithm from **scratch** to really understand the technology and mathematics behind it.