VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING INTEGRATION PROJECT

Assignment

HANDWRITTEN DIGIT RECOGNITION

Instructors/Lecturers: Nguyen Tien Thinh

Authors: Nguyen Manh Dan - 2052932

HO CHI MINH CITY, 12/2022

Contents

1 Motivation

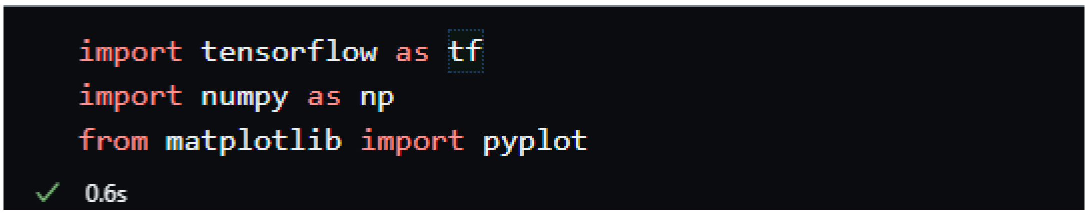
A common challenge in machine learning is the hand-written digit recognition problem based on the MNIST dataset, which contains 60000 images for training and 10000 images for testing.

Researchers working on this problem have achieved ”near-human performance” on the MNIST database, using a committee of neural networks, with error rates of less than 0.1.

In my attempt to solve this problem, I aim to implement from scratch a convolutional neural network with 1 convolution layer, 1 hidden layer using ReLU activation function, and an output layer using the cross entropy softmax activation function.

2 Implementation

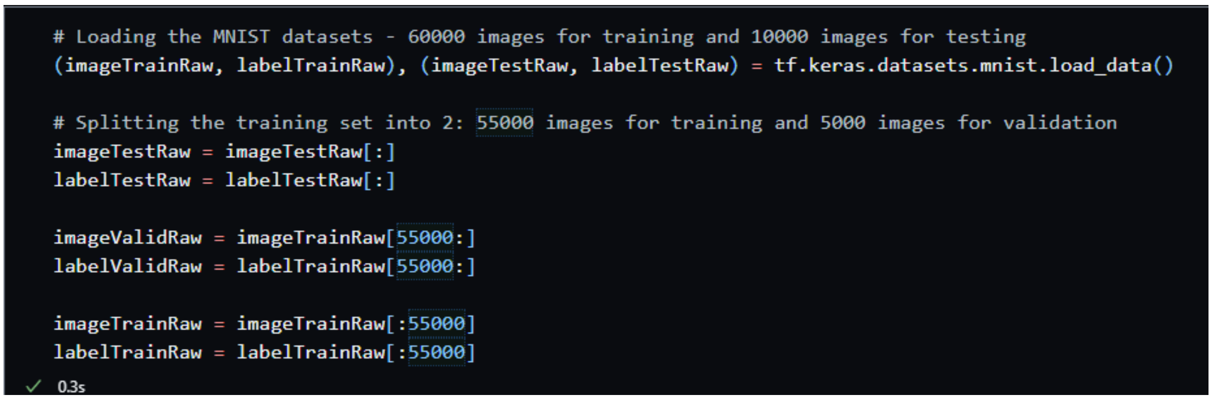
To begin, we will import a few modules needed for this implementation.



```
import tensorflow as tf
import numpy as np
from matplotlib import pyplot
```

```
✓ 0.6s
```

- NumPy is used for matrix calculations.
- Matplotlib is used to draw images and plots.
- Tensorflow is only used to load the MNIST dataset and perform 1-hot encoding.



```
# Loading the MNIST datasets - 60000 images for training and 10000 images for testing
(imageTrainRaw, labelTrainRaw), (imageTestRaw, labelTestRaw) = tf.keras.datasets.mnist.load_data()

# Splitting the training set into 2: 55000 images for training and 5000 images for validation
imageTestRaw = imageTestRaw[:]
labelTestRaw = labelTestRaw[:]

imageValidRaw = imageTrainRaw[55000:]
labelValidRaw = labelTrainRaw[55000:]

imageTrainRaw = imageTrainRaw[:55000]
labelTrainRaw = labelTrainRaw[:55000]
```

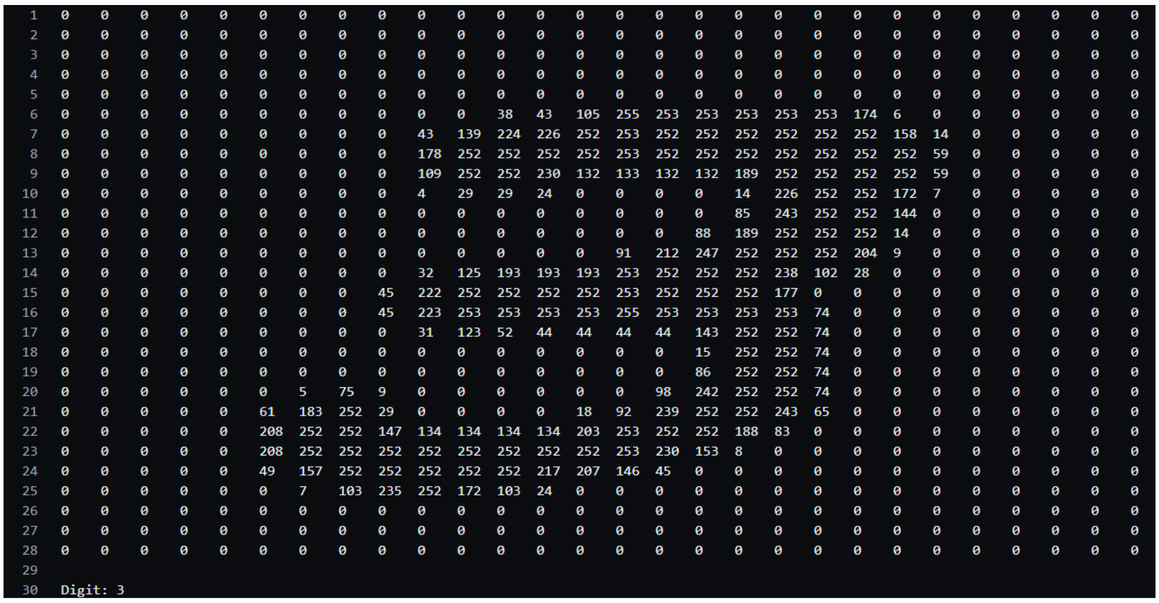
```
✓ 0.3s
```

We will use Tensorflow to load the MNIST dataset which contains 60000 images for training and 10000 images for testing. Then we will split 5000 images from the training set to make a validation set.

Each element in an image array is a 2D 28x28 matrix. Each image array is accompanied by a label array which indicates which digit the 2D matrix should be.

For example, a matrix that represents the digit 3 looks like the below:

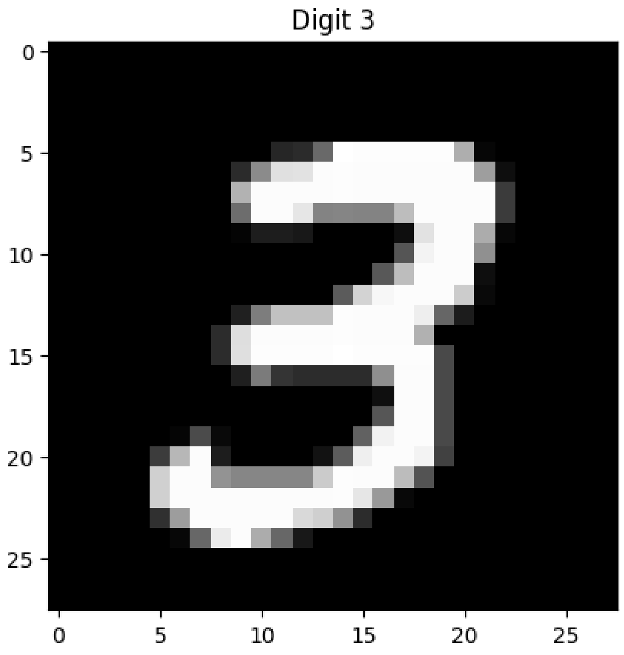
As we can see, this matrix has 784 elements, each with a value ranging from 0 to 255. Looking



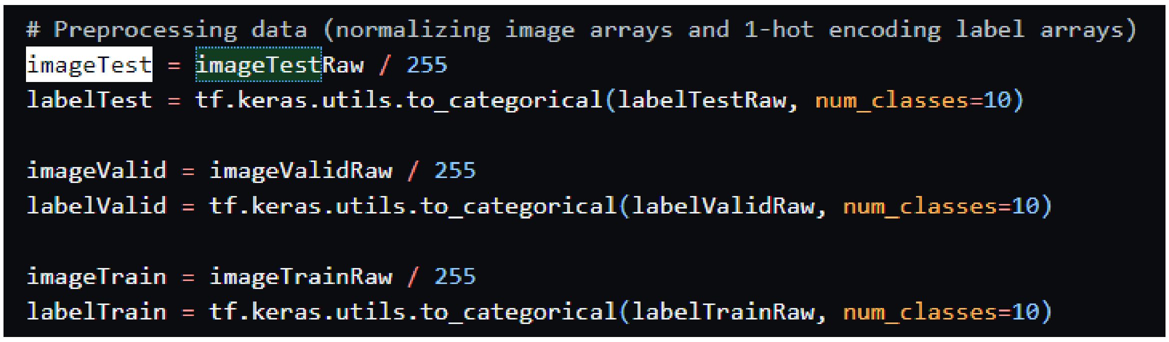
```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   38  43  105 255 253 253 253 253 253 174 6   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   43  139 224 226 252 253 252 252 252 252 252 252 158 14  0   0   0   0   0
0   0   0   0   0   0   0   0   0   178 252 252 252 252 253 252 252 252 252 252 252 252 59  0   0   0   0   0
0   0   0   0   0   0   0   0   0   109 252 252 230 132 133 132 132 189 252 252 252 252 59  0   0   0   0   0
0   0   0   0   0   0   0   0   0   4   29  29  24  0   0   0   0   14  226 252 252 172 7   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   85  243 252 252 144 0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   88  189 252 252 252 14  0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   91  212 247 252 252 252 204 9   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   32  125 193 193 193 253 252 252 252 238 102 28  0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   45  222 252 252 252 252 253 252 252 252 177 0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   45  223 253 253 253 253 255 253 253 253 253 74  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   31  123 52  44  44  44  44  143 252 252 74  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   15  252 252 74  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   86  252 252 74  0   0   0   0   0   0   0   0
0   0   0   0   0   0   5   75  9   0   0   0   0   0   0   98  242 252 252 74  0   0   0   0   0   0   0   0
0   0   0   0   0   61  183 252 29  0   0   0   0   18  92  239 252 252 243 65  0   0   0   0   0   0   0   0
0   0   0   0   0   208 252 252 147 134 134 134 134 203 253 252 252 188 83  0   0   0   0   0   0   0   0   0
0   0   0   0   0   208 252 252 252 252 252 252 252 252 253 230 153 8   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   49  157 252 252 252 252 252 217 207 146 45  0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   7   103 235 252 172 103 24  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

Digit: 3
```

at the layout of this matrix, we can see that it resembles the digit 3.

Indeed, if we plot this matrix using Matplotlib.pyplot, we will see a grey image showing digit 3.



Next, we will preprocess the datasets.



```
# Preprocessing data (normalizing image arrays and 1-hot encoding label arrays)
imageTest = imageTestRaw / 255
labelTest = tf.keras.utils.to_categorical(labelTestRaw, num_classes=10)

imageValid = imageValidRaw / 255
labelValid = tf.keras.utils.to_categorical(labelValidRaw, num_classes=10)

imageTrain = imageTrainRaw / 255
labelTrain = tf.keras.utils.to_categorical(labelTrainRaw, num_classes=10)
```

For the image arrays, we will perform normalization to make sure that there will not be overflowing calculation problems later on.

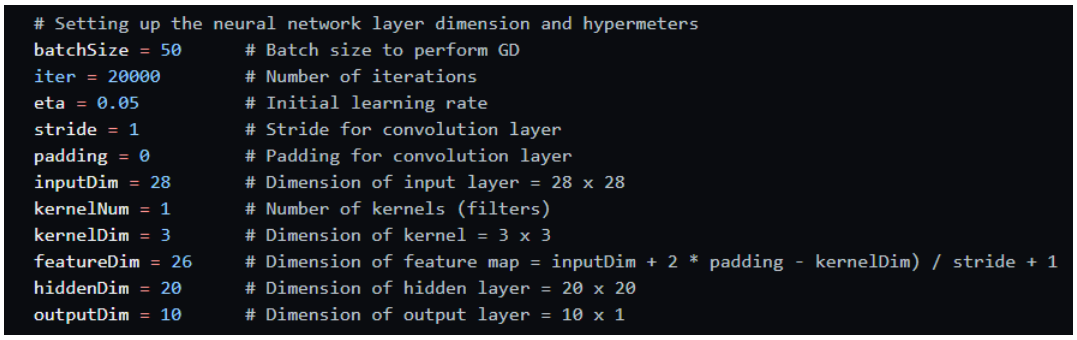
The normalization process is based on this formula

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

In each matrix, the min value is 0 and the max value is 255, therefore, we only need to divide each matrix by 255 to get the normalized matrix.

For the label arrays, we will perform 1-hot encoding using a Tensorflow function. This function will take in a list of indices i, with i ranging from 0 to 9, and generate an array of length 10 consisting of 9 zeros and 1 1 at the i-th position.

The next step is to set up the neural network layer dimension and hypermeters.

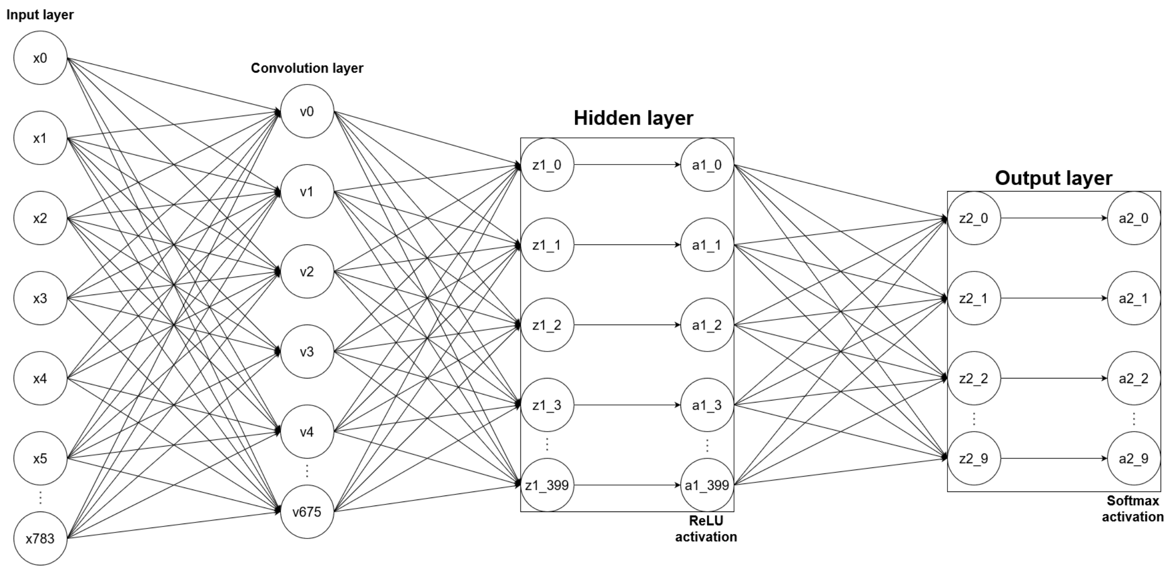


```
# Setting up the neural network layer dimension and hypermeters
batchSize = 50      # Batch size to perform GD
iter = 20000        # Number of iterations
eta = 0.05          # Initial learning rate
stride = 1          # Stride for convolution layer
padding = 0         # Padding for convolution layer
inputDim = 28       # Dimension of input layer = 28 x 28
kernelNum = 1       # Number of kernels (filters)
kernelDim = 3       # Dimension of kernel = 3 x 3
featureDim = 26     # Dimension of feature map = inputDim + 2 * padding - kernelDim) / stride + 1
hiddenDim = 20      # Dimension of hidden layer = 20 x 20
outputDim = 10      # Dimension of output layer = 10 x 1
```

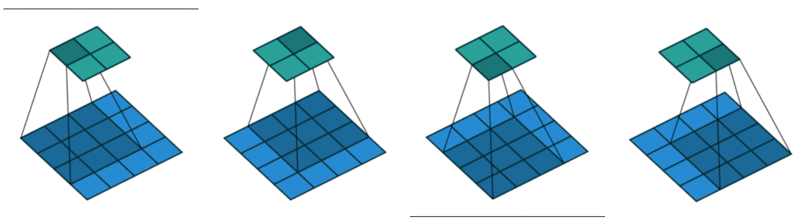
- batchSize is the number of inputs the model will take in in each interation.
- iter is the number of iterations or the number of times the model will train.
- eta is the learning rate used in gradient descent.
- stride, padding are used to calculate the size of the feature map.
- InputDim is the size of the input layer (28 x 28).
- kernelNum is the numbers of kernel we will used to extract features from the input.
- kernelDim the the size of the kernel (3 x 3).
- featureDim is the size of the feature map (26 x 26).
- hiddenDim is the size of the hidden layer (20 x 20).
- outputDim is the size of the output layer (10 x 1).

We can change batchSize, iter and eta later for tuning purpose.

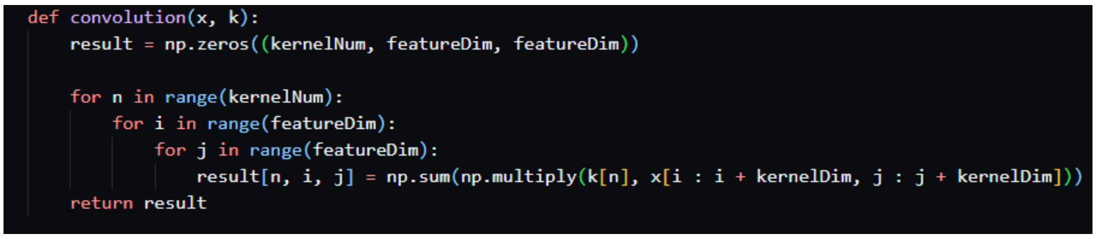
To build a convolutional neural network, first, we will have an overview of how a convolutional neural network works.



From an input of size 28 x 28, we will map it to a matrix of size 26 x 26 by performing 2d convolution with a 3 x 3 filer. This means that we will slide a 3 x 3 filter across the 28 x 28 matrix, multiply each element from the matrix with each element from the filter matrix and sum all the results up to make an element in the 26 x 26 feature map.



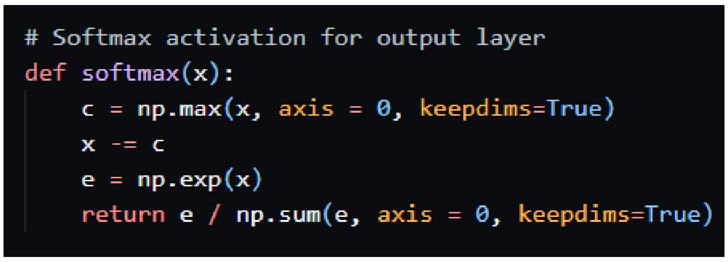
In those pictures, the bottom lighter blue square is our input matrix, the darker blue square on top of it is the kernel or filter, and the green square above is the feature map. Below is the code to perform this function.



```python
def convolution(x, k):
    result = np.zeros((kernelNum, featureDim, featureDim))

    for n in range(kernelNum):
        for i in range(featureDim):
            for j in range(featureDim):
                result[n, i, j] = np.sum(np.multiply(k[n], x[i : i + kernelDim, j : j + kernelDim]))
    return result
```

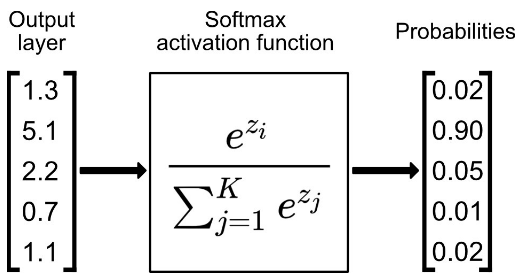
Then we will flatten the feature map from a 26 x 26 matrix to a 676 x 1 matrix and perform a linear transformation using the formula Wx + b = y. In this formula, x has the dimension of 676 x 1, b and y will have the dimension of 400 x 1 and W will have the dimension of 400 x 676.

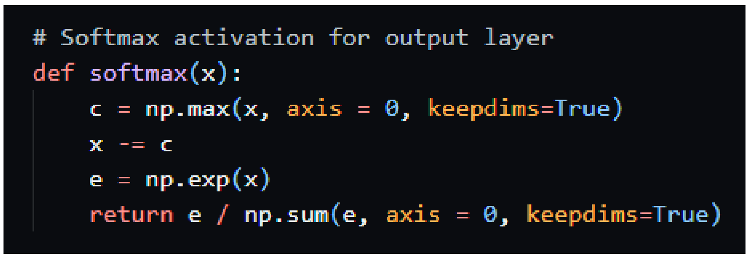
Then an input of size 400 x 1 will go through a ReLU activation that follows the formula y = max(0, x).



```python
# Softmax activation for output layer
def softmax(x):
    c = np.max(x, axis = 0, keepdims=True)
    x -= c
    e = np.exp(x)
    return e / np.sum(e, axis = 0, keepdims=True)
```

We will then linearly transform the 400 x 1 matrix to a 10 x 1 matrix and put it through softmax activation function to get output. The softmax function follows this formula.


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

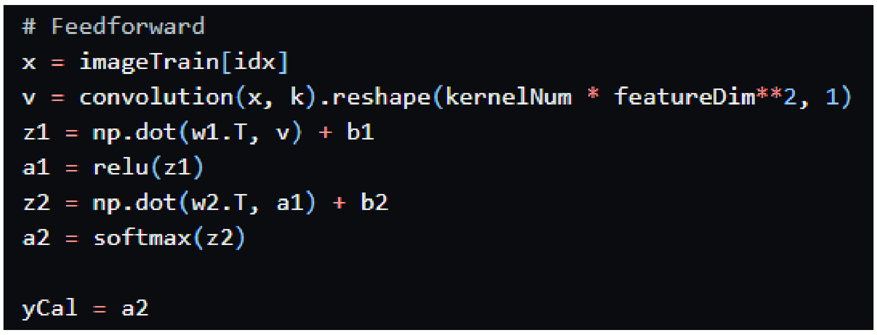


```
# Softmax activation for output layer
def softmax(x):
    c = np.max(x, axis = 0, keepdims=True)
    x -= c
    e = np.exp(x)
    return e / np.sum(e, axis = 0, keepdims=True)
```

We make a small adjustment to the formula by subtracting a c equal to the maximum element of an input matrix from all elements in the matrix to make our function more stable and less prone to overflowing.

All of the above steps we have just done is called belong to a process called feedforward, which essentially means that we just pass our inputs through all layers in our model until it reaches the output layer. The output from this process is an array of size 10 x 1 that contains the possibilities of an image being a certain digit.

For example, if the 5th element of the output array has the value of 0.755 then there is a 75 percent chance that the input matrix will resemble the digit 5.



```
# Feedforward
x = imageTrain[idx]
v = convolution(x, k).reshape(kernelNum * featureDim**2, 1)
z1 = np.dot(w1.T, v) + b1
a1 = relu(z1)
z2 = np.dot(w2.T, a1) + b2
a2 = softmax(z2)

yCal = a2
```

Then we will calculate the loss value. The loss value of a softmax function follows this formula.

$$J(\mathbf{W}) = \sum_{i=1}^{N} ||\mathbf{a}_i - \mathbf{y}_i||_2^2$$

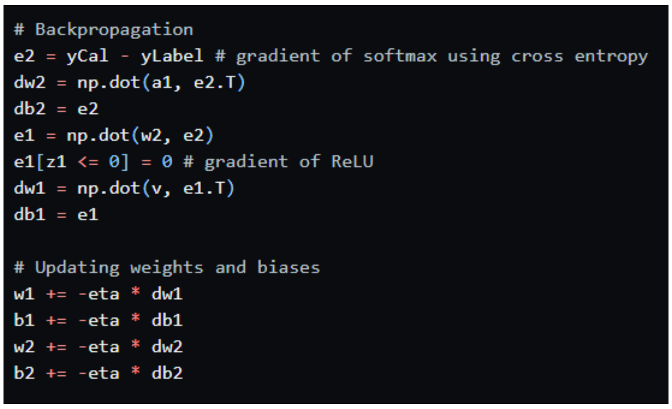
However, there is an alternative way to represent this loss function by using cross entropy, this will make our derivation later on much easier.

$$J(\mathbf{W};\mathbf{x}_i,\mathbf{y}_i) = -\sum_{j=1}^{C} y_{ji}\log(a_{ji})$$

Since we plan to perform stochastic gradient descent, we won't need to divide the loss result by the amount of inputs.

Now we will perform backpropagation to update weights and biases in each layer. However We should note that we won't perform the backpropagation and update on the kernel and this will affect our prediction somewhat.

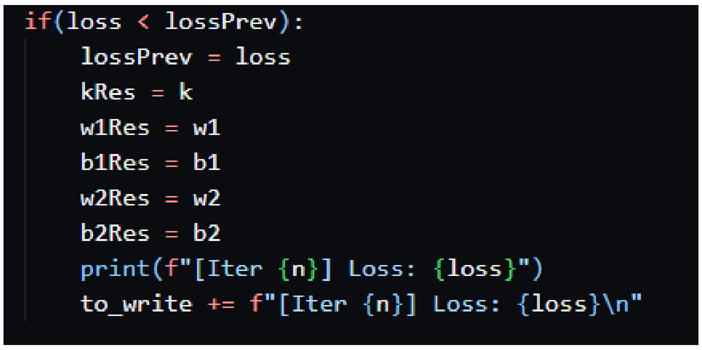
$$\begin{aligned}
\mathbf{E}^{(2)} &= \frac{\partial J}{\partial \mathbf{Z}^{(2)}} = \frac{1}{N}(\hat{\mathbf{Y}} - \mathbf{Y}) \\
\frac{\partial J}{\partial \mathbf{W}^{(2)}} &= \mathbf{A}^{(1)}\mathbf{E}^{(2)T} \\
\frac{\partial J}{\partial \mathbf{b}^{(2)}} &= \sum_{n=1}^{N} \mathbf{e}_n^{(2)} \\
\mathbf{E}^{(1)} &= \left(\mathbf{W}^{(2)}\mathbf{E}^{(2)}\right) \odot f'(\mathbf{Z}^{(1)}) \\
\frac{\partial J}{\partial \mathbf{W}^{(1)}} &= \mathbf{A}^{(0)}\mathbf{E}^{(1)T} = \mathbf{X}\mathbf{E}^{(1)T} \\
\frac{\partial J}{\partial \mathbf{b}^{(1)}} &= \sum_{n=1}^{N} \mathbf{e}_n^{(1)}
\end{aligned}$$



```
# Backpropagation
e2 = yCal - yLabel # gradient of softmax using cross entropy
dw2 = np.dot(a1, e2.T)
db2 = e2
e1 = np.dot(w2, e2)
e1[z1 <= 0] = 0 # gradient of ReLU
dw1 = np.dot(v, e1.T)
db1 = e1

# Updating weights and biases
w1 += -eta * dw1
b1 += -eta * db1
w2 += -eta * dw2
b2 += -eta * db2
```

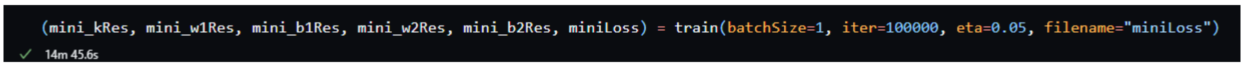
Since we are doing stochastic gradient descent, in each iteration, we will only take 1 image from the dataset and put it through the model instead of using the whole dataset like full batch gradient descent.

This will improve the calculating speed greatly at the cost of loss value not guaranteed to always decrease. Therefore, we will only take the value of w1, b1, w2, and b2 only if they make the loss value of the current iteration lower than that of the previous iteration.

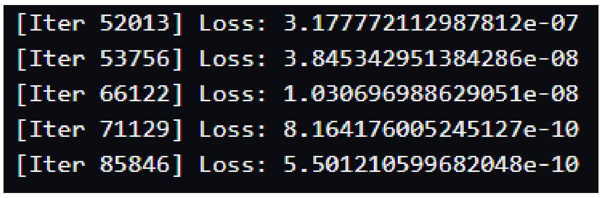


```
if(loss < lossPrev):
    lossPrev = loss
    kRes = k
    w1Res = w1
    b1Res = b1
    w2Res = w2
    b2Res = b2
    print(f"[Iter {n}] Loss: {loss}")
    to_write += f"[Iter {n}] Loss: {loss}\n"
```

3 Results



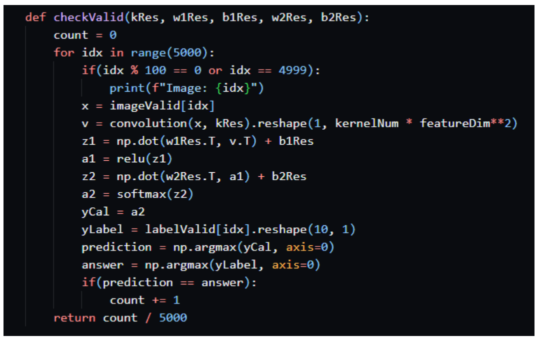
```
(mini_kRes, mini_w1Res, mini_b1Res, mini_w2Res, mini_b2Res, miniLoss) = train(batchSize=1, iter=100000, eta=0.05, filename="miniLoss")
✓ 14m 45.6s
```



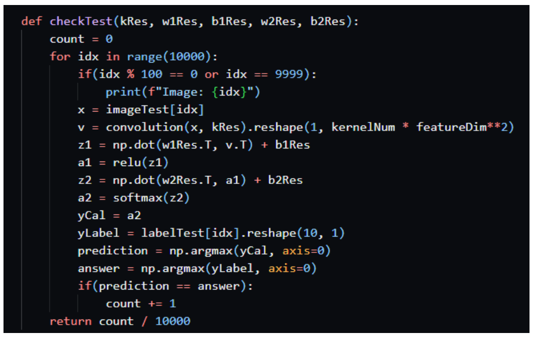
```
[Iter 52013] Loss: 3.177772112987812e-07
[Iter 53756] Loss: 3.845342951384286e-08
[Iter 66122] Loss: 1.030696988629051e-08
[Iter 71129] Loss: 8.164176005245127e-10
[Iter 85846] Loss: 5.501210599682048e-10
```

With a batch size of 1, an eta of 0.06, and 100000 iterations, our model takes around 15 minutes to train and manages to minimize the lost value to 5.5e-10.

This trained model will then get tested with the validation set and the testing set.

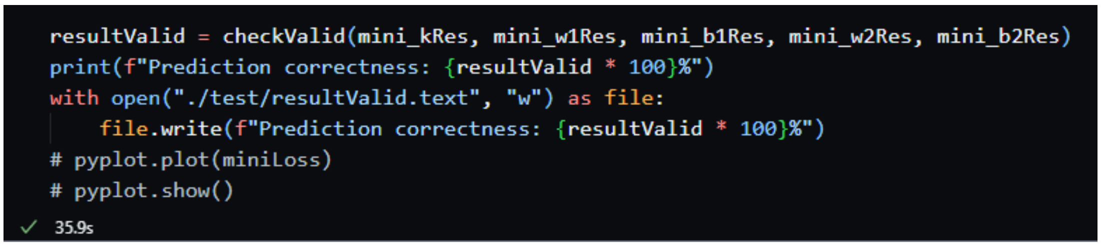


```python
def checkValid(kRes, w1Res, b1Res, w2Res, b2Res):
    count = 0
    for idx in range(5000):
        if(idx % 100 == 0 or idx == 4999):
            print(f"Image: {idx}")
        x = imageValid[idx]
        v = convolution(x, kRes).reshape(1, kernelNum * featureDim**2)
        z1 = np.dot(w1Res.T, v.T) + b1Res
        a1 = relu(z1)
        z2 = np.dot(w2Res.T, a1) + b2Res
        a2 = softmax(z2)
        yCal = a2
        yLabel = labelValid[idx].reshape(10, 1)
        prediction = np.argmax(yCal, axis=0)
        answer = np.argmax(yLabel, axis=0)
        if(prediction == answer):
            count += 1
    return count / 5000
```

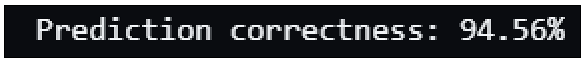


```python
def checkTest(kRes, w1Res, b1Res, w2Res, b2Res):
    count = 0
    for idx in range(10000):
        if(idx % 100 == 0 or idx == 9999):
            print(f"Image: {idx}")
        x = imageTest[idx]
        v = convolution(x, kRes).reshape(1, kernelNum * featureDim**2)
        z1 = np.dot(w1Res.T, v.T) + b1Res
        a1 = relu(z1)
        z2 = np.dot(w2Res.T, a1) + b2Res
        a2 = softmax(z2)
        yCal = a2
        yLabel = labelTest[idx].reshape(10, 1)
        prediction = np.argmax(yCal, axis=0)
        answer = np.argmax(yLabel, axis=0)
        if(prediction == answer):
            count += 1
    return count / 10000
```

For the validation set, the model yields a result of 94.56 percent of correctness in 35.6 seconds.

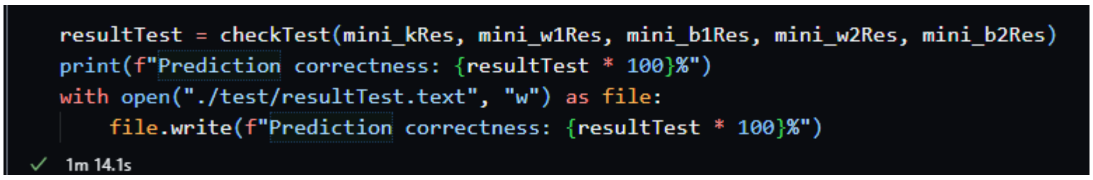


```
resultValid = checkValid(mini_kRes, mini_w1Res, mini_b1Res, mini_w2Res, mini_b2Res)
print(f"Prediction correctness: {resultValid * 100}%")
with open("./test/resultValid.text", "w") as file:
    file.write(f"Prediction correctness: {resultValid * 100}%")
# pyplot.plot(miniLoss)
# pyplot.show()
✓ 35.9s
```

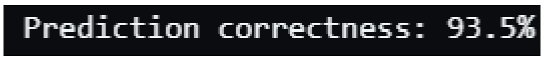


```
Prediction correctness: 94.56%
```

For the validation set, the model yields a result of 93.5 percent of correctness in 1 minute and 14 seconds.



```
resultTest = checkTest(mini_kRes, mini_w1Res, mini_b1Res, mini_w2Res, mini_b2Res)
print(f"Prediction correctness: {resultTest * 100}%")
with open("./test/resultTest.text", "w") as file:
    file.write(f"Prediction correctness: {resultTest * 100}%")
✓ 1m 14.1s
```



```
Prediction correctness: 93.5%
```

4 Improvement

This model only does not offer a lot in terms of improvement on the base model. Mainly because of 2 reasons:

1. I don't perform backpropagation and update on the kernel.
2. I only focus on minimizing the loss function on the training set which will lead to overfitting and makes the model less correct on other unseen datasets.

For the kernel problem, it is indeed possible to perform backpropagation on the kernel, unfortunately, I didn't get the correct results when performing backpropagation so it will be something I need to look into more thoroughly.

For the overfitting problem, perhaps techniques such as early stopping will be useful,

5 References

1. Softmax regression. https://machinelearningcoban.com/2017/02/17/softmax/-cross-entropy

2. Multi-layer Perceptron và Backpropagation. https://machinelearningcoban.com/2017/02/24/mlp/

3. Overfitting. https://machinelearningcoban.com/2017/03/04/overfitting/