Group 19

Ömer Faruk Pekten
22002383

Ahmet Bera Özbolat
21902777

EEE 485 Statistical Learning and Data Analytics

Term Project Phase 1 Report

Handwritten Digit Recognition

1. Introduction

The primary objective of this project is to implement three machine learning algorithms from scratch for the task of image classification. The focus of the classification task is to recognize handwritten digits and categorize them into distinct classes by analyzing their features. To achieve this, we will utilize the MNIST dataset, a widely used benchmark dataset in the machine learning community [1]. Dataset consists of pixel grayscale images of handwritten digits (0 to 9), and each is associated with a label indicating the corresponding digit. Also, as it is mensioned in the project description, three distinct learning algorithms will be used to decide which algorithm is most convenient.

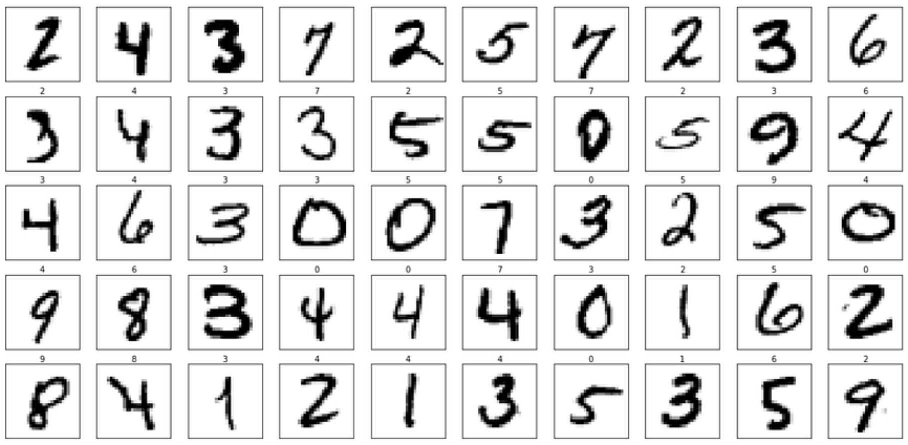


Fig. 1 Handwritten digits [8]

2. Motivation

Handwritten digit recognition is a fundamental problem in the field of computer vision and pattern recognition, with widespread applications in areas such as postal automation, check processing, and document verification [2]. By implementing machine learning algorithms tailored to this specific task and leveraging the MNIST dataset, we aim to explore the intricacies of digit recognition and gain insights into the underlying principles of image classification

3. Expected Contributions of each group member

Expected Contributions of each group member is stated below:

Ahmet Bera Özbolat: Implementation of FNN and KNN systems.

Ömer Faruk Pekten: Implementatıon of random forest and implementation of PCA and K-fold cross-validation.

Other then stated contribitions, each group member will have an idea of each task of the project and work together and help each other when it is neccesary.

4. Methodology

4.1. MNIST Dataset

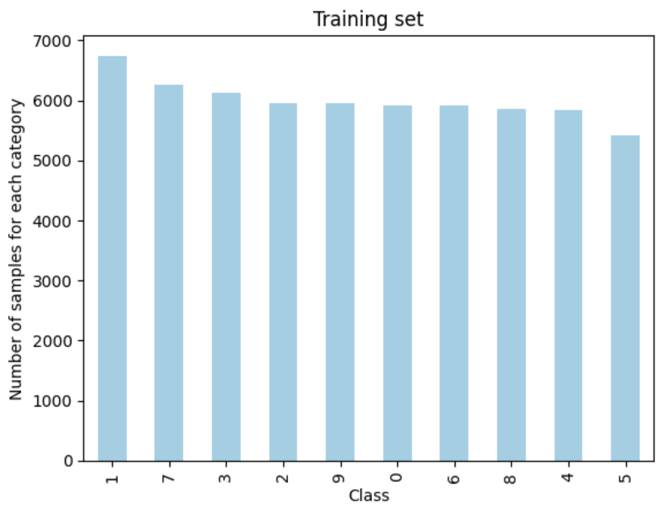


Fig. 2 MNIST Dataset distribution

The MNIST dataset consists of 28x28 pixel grayscale images of handwritten digits (0 to 9), making it ideal for digit recognition tasks. Each image in the dataset is associated with a label indicating the corresponding digit. It has 60000 samples in it and as can be seen in Fig 2., it can be said that it is evenly distributed. Which will contribute to reliability of the results.

4.2. Data Manipulation

Data manipulation is crucial for the effectiveness of machine learning algorithms. The distribution of data within the dataset, as well as its dimensionality and size, can introduce unwanted errors during training and increase implementation costs. Therefore, necessity of implementing preprocessing methods to address these challenges is recognized. By normalizing the data, ensuring uniformity in scale across features, aim is to stabilize the training process and prevent issues stemming from varying ranges of feature values [3]. For this purpose in the system, it is decided to use implemented Principal Component Analysis (PCA) and K-fold cross-validation as techniques for reducing the dimensionality of our dataset.

4.2.1. Principal Component Analysis (PCA)

PCA allows reduction in dimension of the data while still capturing the most important patterns and structures. The principal components of the dataset, which are orthogonal vectors that represent the directions of maximum variance, were found in order to accomplish this reduction. It successfully reduced the dimensionality of the dataset while preserving a sizable portion of its information by choosing the top 143 principal components of 784, which account for the greatest variance in the data. In addition to aiding in the mitigation of the dimensionality curse, this dimensionality reduction allowed for faster computation and enhanced the effectiveness of later machine learning tasks in systems.

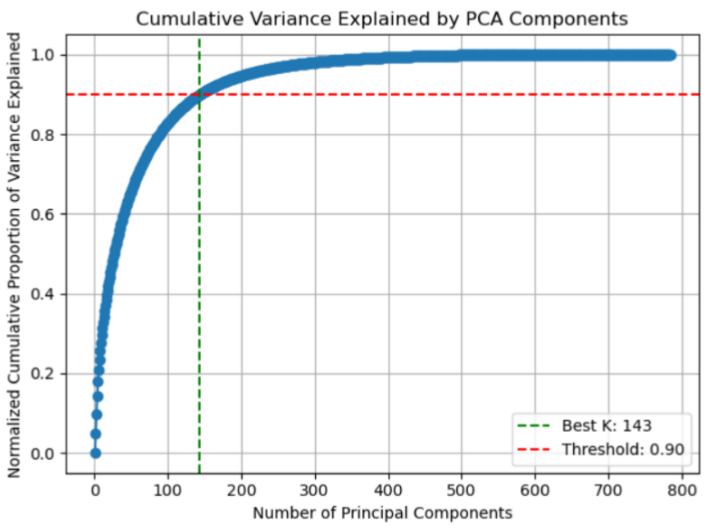


Fig. 3 Cumulative variance and Chosen number of principal components

4.2.2. K-Fold Cross-Validation

The dataset is systematically divided into K equally-sized folds using K-fold cross-validation. The model undergoes training and assessment K times, with each iteration utilizing a different fold as the validation set and the remaining folds as the training set. This method is implemented to mitigate the possibility of overfitting or underfitting to any particular training-validation split, thereby enabling a comprehensive evaluation of the model's performance across multiple data subsets. A more representative estimate of the model's generalization capacity is derived by averaging the evaluation metrics obtained from the K iterations. Additionally, K-fold cross-validation provides insightful information about the model's variance and stability over various data subsets. This enhances confidence in the model's predictive power and facilitates informed decisions regarding model selection and hyperparameter tweaking [4].

4.3. Algorithms

Selecting apprproite maching learning algorithms is crucical for the success of our mission. After the investigation of the dataset by looking at various factors such as dataset charateristics and implementation cost, performance requirements, feasilbility we have choosen to implement Feedforward Neural Network (FNN), Random Forest, and K-Nearest Neighbors (KNN). By leveraging these diverse algorithms, we aim to explore different modeling approaches and potentially achieve improved performance on the digit recognition task.

4.3.1. Feedforward Neural Networks (FNN)

The operation of a Feedforward Neural Networks (FNN) is sequential. FNNs are made up of layers of networked nodes, or neurons, that resemble the composition and operation of the human brain. FNN’s neurons gather input signals, give each one a weight, and then produce an output signal that is sent to the neurons in layers below it. This process keeps going until the network's prediction or classification is represented in a final output. A set of labeled examples is given to the FNN during the training phase, which enables it to modify the weights of connections between neurons to reduce the differences between predicted and actual outputs. FNNs offer a powerful framework for learning complex patterns and representations from data, leveraging their multiple layers of interconnected neurons to automatically extract hierarchical features from the input images [6].

4.3.2. Random Forest

Random Forest, functioning as an ensemble tree-based algorithm, aggregates decisions from multiple decision trees to determine the class of input objects. Furthermore, each decision tree is tasked with classifying data utilizing only a subset of the available features. Random Forest stands out for its versatility and robustness in handling high-dimensional data, making it well-suited for the diverse range of handwriting styles present in the MNIST dataset [5].

4.3.3. K-Nearest Neighbors (KNN)

The commonly utilized classification algorithm, K-Nearest Neighbor (KNN), operates non-linearly and requires the determination of an optimal value for "k" to effectively perform the classification task. Through analysis, it has been observed that Eq. 1 is used commonly, with "N" representing the dataset size. KNN provides a simple yet effective approach to classification, leveraging the similarity between images to make accurate predictions, particularly beneficial for smaller subsets of data or when computational resources are limited. However, KNN faces challenges with high-dimensional data due to the curse of dimensionality. To address this, Principal Component Analysis (PCA) will be employed to reduce the number of features utilized in the algorithm. Additionally, while the algorithm typically utilizes Euclidean distance, alternative distance metrics such as Kullback-Leibler divergence can also be considered [7].

$$k = \frac{\sqrt[2]{N}}{2}$$

$(Eq.\ 1)$

5. Implementations

5.1. FNN Implementation

It was first necessary to come up with a way to get data out of the database to start training the network. After the appropriate dataset was obtained via research made on the internet, it is imported into the Python interpreter by means of the "MNIST" module. Following that, the 60,000-sample training dataset and the 10,000-sample testing dataset were successfully imported with their associated labels. Once the data was ready for processing, the focus shifted to using Python to build the neural network structure. Due to the requirement of producing multiple neural networks with different parameters, an object-oriented methodology was chosen. We have chosen ReLu as activation function for hidden layers and sigmoid for output layer furthermore we have decided to use MSE as a Loss function in our algorithm. For learning rates and size of the hidden layers we have decided to use 3 different learning rates and size of the number of neurons in the hidden layer as 300. Following this, the focus shifted towards advancing the forward propagation process.

$$v_1 = {W_1}^T \cdot x \quad (Eq.2)$$

$$o_1 = \text{activation}(v_1) \quad (Eq.3)$$

$$v_2 = {W_2}^T \cdot o_1 \quad (Eq.4)$$

$$o_2 = \text{activation}(v_2) \quad (Eq.5)$$

In this scenario, where x represents the input vector, W_1 and W_2 denote the weights for the hidden layer and the second layer, respectively, the forward propagation process was implemented in code, as demonstrated in the appendix.

After the forward propagation's accuracy was established, focus was directed towards the backpropagation stage. This algorithm is essential for optimizing the weights in the network because it minimizes the difference between the expected and actual outputs. As shown below, backpropagation computes the gradients of the error with respect to the model's parameters in each iteration and modifies these parameters using gradient descent. *Eq.* 6 had to be constructed in a matrix form for each of the layers in order to carry out this operation in an algorithm with NumPy's assistance in an effective way. The chain rule is used to determine the output layer's gradient.

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

$(Eq. 6)$ Gradients of error

$$\frac{\partial E(n)}{\partial w_{ji}} = \frac{\partial E(n)}{\partial e_j}\frac{\partial e_j}{\partial o_j}\frac{\partial o_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} = e_j(n)(-1)f'\left(v_j(n)\right)y_i$$

$(Eq. 7)$ Gradient of last layer

$$\delta(n) = \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_R \end{bmatrix} \quad e(n) = \begin{bmatrix} e_1 \\ \vdots \\ e_R \end{bmatrix} \quad T'^{(n)} = \begin{bmatrix} f'(v_1) & 0 & \cdots & 0 \\ \vdots & f'(v_2) & \cdots & \vdots \\ 0 & 0 & \cdots & f'(v_R) \end{bmatrix}$$

$$\delta(n) = T'(n)e(n)$$

$(Eq. 8)$ Matrix Derivation of the gradient

$$W_e \leftarrow W_e + \eta\delta(n)y_e^T$$

$(Eq. 9)$ Weight update by the gradient

Using the previously given equations, the derivation for the last layer's backpropagation was finished. This derivation was then converted into code to carry out the last layer's backpropagation procedure. The same process had to be followed for the first layer in order to finish the back propagation. The equations below were used to derive the gradient for the hidden layer.

$$\frac{\partial E(n)}{\partial \widehat{w}_{ji}} = \frac{\partial E(n)}{\partial e_k}\frac{\partial e_k}{\partial o_k}\frac{\partial o_k}{\partial v_k}\frac{\partial v_k}{\partial y_j}\frac{\partial y_j}{\partial \hat{v}_j}\frac{\partial \hat{v}_j}{\partial \widehat{w}_{ji}} = \sum_{k=1}^{R} e_k(-1)f'(v_k)w_{kj}f'(\hat{v}_j)z_i$$

$(Eq. 10)$ Gradient of hidden layer

$$\hat{\delta}_j(n) = f'(\hat{v}_j)\left[\sum_{k=1}^{R} \delta_k(n)\, w_{kj}\right]$$

$$\hat{\delta}(n) = T'(\hat{v})W^T\delta(n)$$

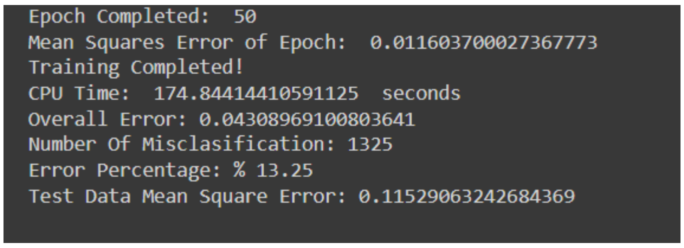
$(Eq. 11)$ Matrix Derivation of the gradient

$$\widehat{W}_e \leftarrow \widehat{W}_e + \eta\hat{\delta}(n)z_e^T$$

$(Eq. 12)$ Weight update by the gradient

A similar process was followed as with the last layer when the necessary derivation for the backpropagation of the hidden layer was completed. The network was considered ready for training after the backpropagation function was completed. As such, a function for controlling both forward and backpropagation was developed. The network is trained using a generic batch size equal to the size of the input dataset in another function that was designed to run training for 50 epochs. Also, this function captures system performance metrics including the training sets mean square error.

Upon completing the implementation of the code, the system's performance and errors were documented by evaluating its performance using the testing dataset. Consequently, the process of gathering accuracy data commenced. To illustrate how this data was collected, below is the output from a run:



```
Epoch Completed:  50
Mean Squares Error of Epoch:  0.011603700027367773
Training Completed!
CPU Time:  174.84414410591125  seconds
Overall Error: 0.04308969100803641
Number Of Misclasification: 1325
Error Percentage: % 13.25
Test Data Mean Square Error: 0.11529063242684369
```

Fig. 4 Systems Accuracy output

Usually, it takes two to three hours to train each neural network using the complete dataset, which consists of 60,000 images. As such, it would take a full day to train all combinations using all of the training data. A reduced portion of the dataset, comprising 1,250 images, was used in order to speed up the procedure and obtain accurate data for every combination. Faster training and result comparison are possible with this method, which is similar to applying stochastic gradient descent.

Table 1: Data with activation function is ReLu and sigmoid (without PCA)

CASE 1 (NO PCA)	Epoch	Training MSE	Test MSE	# of Misclas.	Accuracy	CPU Time
N = 300 , η = 0.01	50	0.126	0.105	1220	87.80 %	98.347 sec
N = 300 , η = 0.05	50	0.044	0.116	1304	86.96 %	90.652 sec
N = 300 , η = 0.09	50	0.036	0.112	1284	87.17 %	91.353 sec

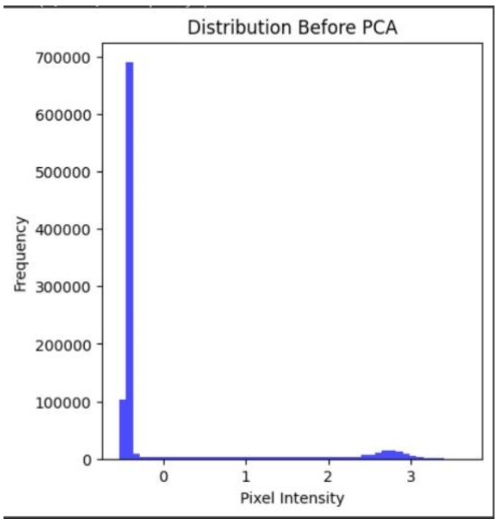


Fig. 5 Distribution before PCA

Table 2: Data with activation function is ReLu and sigmoid (with PCA)

CASE 2 (PCA)	Epoch	Training MSE	Test MSE	# of Misclas.	Accuracy	CPU Time
N = 300 , η = 0.01	50	1.24	0.155	1096	89.04%	43.648 sec
N = 300 , η = 0.05	50	1.23	0.149	1129	88.71 %	42.44 sec
N = 300 , η = 0.09	50	1.19	0.123	1132	88.68 %	77.026 sec

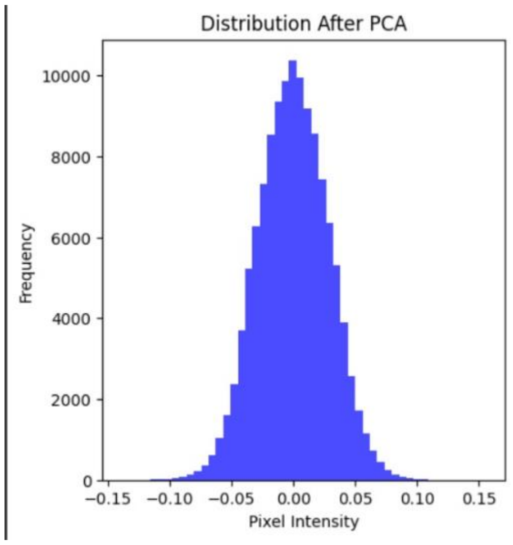


Fig. 6 Distribution after PCA

As it can be seen usage of PCA reduces complexity of dataset by removing unneccesary features leading to reduce in implementation cost and increasing accuracy.

5.2. KNN Implementation

To start the KNN implementation, several libraries such as NumPy and Pandas are leveraged. Again, MNIST dataset is introduced to the system. When it comes to choosing an initial k value, ‘3’ is chosen as default.

Later, the essential objective was to choose a proper distance calculation method. Between the distance calculation methods, Euclidean distance is one of the most widely used distance metrics in KNN, which is used to quantify the proximity between data points. In essence, the length of the line segment that connects two points is measured as the Euclidean distance, which is the straight-line distance between them in a Euclidean space. Hence, Euclidian distance is chosen for the desired KNN approach.

$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

$(Eq. 13)$ Euclidian distance formula

Chosen method calculates the distance between train data point and test data point in the training set in a KNN implementation using Euclidean distance. Next, using these distances as a guide, it chooses the K closest neighbors and gives the query point the majority class label among them.

For the efficiency comparison, dataset is divided into different sizes in sense of train and test. Also, it is tested with and without PCA. Results can be seen below.

5.3. Random Forest Implementation

First, the number of estimators ‘n_estimators’ and the maximum depth of each tree ‘max_depth’ are used to instantiate the Random Forest classifier. The decision trees are the crucial component for Random Forest. Recursively dividing the feature space according to the information gained by splitting at each node creates each decision tree. The decision trees in this implementation are built using the same methodology as in the DecisionTree class, with the development of the trees being controlled by parameters like ‘max_depth’ and ‘max_features.’

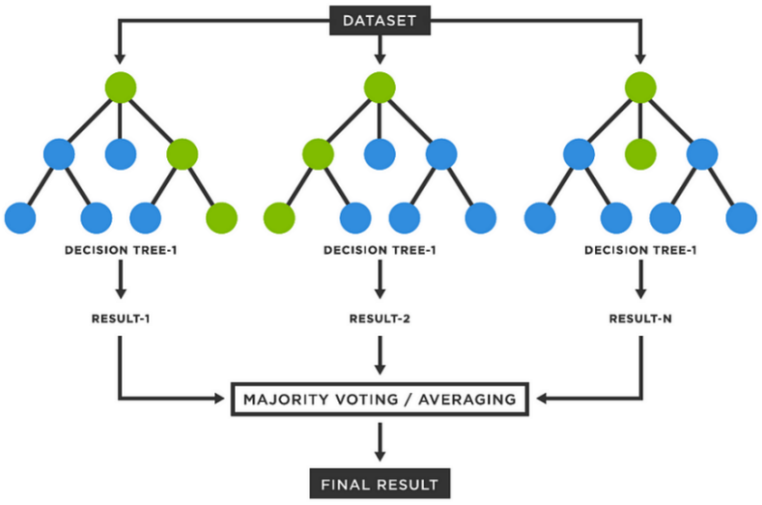


Fig. 7 Random Forest schematic [9]

To ensure variety in the ensemble, a certain number of decision trees are trained on arbitrary subsets of the training data throughout the fitting phase. Using a technique called bootstrapping, samples are chosen at random from the training set, and a decision tree is trained on each sampled subset. The forecasts from each individual decision tree are combined to create predictions once the Random Forest has been trained. The ultimate prediction is the class label that appears most frequently in all of the trees' predictions. This ensemble-based method aids in enhancing generalization performance and decreasing overfitting.

A training and test set of different sizes are created from the MNIST dataset to assess the Random Forest classifier's performance. In addition, tests are carried out to evaluate the effect of dimensionality reduction on model performance, both with and without Principal Component Analysis (PCA). These trials yielded findings that shed light on how well the Random Forest method handles the MNIST dataset.

Conclusion:

To sum up, the project's initial phase used the MNIST dataset to apply three machine learning algorithms for classifying handwritten digits. For the first phase purpose was to fully understand and implement Feedforward Neural Network (FNN). Even though there were obstacles to overcome, especially when it came to incorporating Principal Component Analysis (PCA) into the neural system, a great deal of progress was accomplished, ultimately leading to the successful deployment of the FNN. With the sigmoid function at the output layer and the ReLU function at the hidden layer, the FNN demonstrated encouraging results in digit recognition, setting a strong foundation for the project's later stages. Even though the Random Forest and K-Nearest Neighbors (KNN) algorithms have yet to be implemented, their tenacity and devotion guarantee a steady dedication to conquering challenges and accomplishing the project's goals.

References:

[1] MNIST Dataset. (2019, January 8). Kaggle. https://www.kaggle.com/datasets/hojjatk/mnist-dataset

[2] Shamim, S., Miah, M., Sarker, A., Rana, M., & Jobair, A. (2018). Handwritten Digit Recognition Using Machine Learning Algorithms. Indonesian Journal of Science and Technology, 3(1), 29-39. https://doi.org/10.17509/ijost.v3i1.10795

[3] Srivatsavaya, P. (2023, October 4). Flatten Layer — Implementation, Advantage and Disadvantages. Medium. https://medium.com/@prudhviraju.srivatsavaya/flatten-layer-implementation-advantageand-disadvantages0f8c4ecf5ac5#:~:text=Overall%2C%20the%20Flatten%20layer%20is,other%20components%20of% 20the%20model

[4] K, D. (2021, December 27). Advantages and Disadvantages of K fold cross-validation. Medium. https://dhirajkumarblog.medium.com/advantages-and-disadvantages-of-k-fold-cross-validation5e833009ddb1

[5] Bernard, S., Heutte, L., & Adam, S. (2007, October 23). Using Random Forests for Handwritten Digit Recognition. Proceedings of the International Conference on Document Analysis and Recognition, ICDAR, 2, 1043-1047. https://doi.org/10.1109/ICDAR.2007.4377074

[6] Kalra, K. (2023, July 13). Neural Networks - KHWAB KALRA - Medium. Medium. https://medium.com/@khwabkalra1/neural-networks-a559876c6505

[7] Amato, G., Falchi, F. (2011, January 28). Local Feature based Image Similarity Functions for kNN Classification. ICAART 2011 - Proceedings of the 3rd International Conference on Agents and Artificial Intelligence, 1, 157-166. https://doi.org/10.5220/0003185401570166

[8] L. Cares, "Handwritten Digit Recognition using Convolutional Neural Network (CNN) with Tensorflow," *Medium*, Nov. 30, 2023. https://learner-cares.medium.com/handwritten-digit-recognition-using-convolutional-neural-network-cnn-with-tensorflow-2f444e6c4c31

[9] D. Gunay, "Random Forest - Deniz Gunay - Medium," *Medium*, Sep. 14, 2023. https://medium.com/@denizgunay/random-forest-af5bde5d7e1e

APPENDIX

APPENDIX A

Download link of the .ipynb file: https://we.tl/t-Hn5Z5wlSLL (Wetransfer)

APPENDIX B

CODE:

```
import numpy as np
import pandas as pd
from scipy.special import expit
import time
```

```python
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
```

```python
def load_mnist_data():
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    return X_train, y_train, X_test, y_test
```

```python
def initialize_weights(hidden_size):
    weights_hidden = np.random.uniform(-0.01, 0.01, size=(hidden_size,
784))
    weights_output = np.random.uniform(-0.01, 0.01, size=(10,
hidden_size))

    return weights_hidden, weights_output
```

```python
def initialize_weights2(hidden_size,PCA):
    weights_hidden = np.random.uniform(-0.01, 0.01, size=(hidden_size,
100))
    weights_output = np.random.uniform(-0.01, 0.01, size=(10,
hidden_size))

    return weights_hidden, weights_output
```

```python
def relu_activation(x):
    return np.maximum(x, 0)
```

```python
def sigmoid_activation(x):
    y = expit(x)
    return y
```

```python
def desired_output(label,notDesiredValue):
    if notDesiredValue == -1:
        d = -np.ones((10,1))
        d[label] = 1
        return d
    else:
        d = np.zeros((10,1))
        d[label] = 1
        return d
```

```python
def desired_output_sigmoid(label):
    desired = np.eye(10)[label]

    return desired
```

```
def find_error(output,desired):
    error = desired - output
    return error
```

```
def activation(x, ActivationFunction):
        return ActivationFunction(x)
```

```
def forward_pass(inputs, weights_input_hidden, weights_hidden_output,
activation_hidden, activation_output):
    hidden_inputs = np.dot(weights_input_hidden, inputs)
    hidden_outputs = activation(hidden_inputs, activation_hidden)
    final_inputs = np.dot(weights_hidden_output, hidden_outputs)
    final_outputs = activation(final_inputs, activation_output)
    return hidden_outputs, final_outputs
```

```
def output_gradient(error,final_output):
    derivative=np.eye(10)*(final_output-final_output*final_output)
    gradient=np.dot(derivative,error)
    output_gradient=np.reshape(gradient,(10,1))
    return output_gradient
```

```
def relu_derivative(x):
    x[x>=0]=1
    x[x<0]=0
    return x
```

```
def
input_gradient(layer_num,first_output,output_gradient,weight_output):
    derivative=np.eye(layer_num)*relu_derivative(first_output)
    gradient=np.dot(np.dot(derivative,np.transpose(weight_output)),outp
ut_gradient)
    input_gradient=np.reshape(gradient,(layer_num,1))
    return input_gradient
```

```
def backpropagation_sigmoid(inputs, labels, weights_hidden_output,
weights_input_hidden, activation_hidden, activation_output,
learning_rate, layer_num):
    hidden_outputs, final_outputs = forward_pass(inputs,
weights_input_hidden, weights_hidden_output, activation_hidden,
activation_output)
    desired = desired_output(labels, 0)
    error = find_error(final_outputs, desired)
    output_gradients = output_gradient(error, final_outputs)
    weights_hidden_output_updated = weights_hidden_output +
learning_rate * np.dot(output_gradients, hidden_outputs.T)
```

```
    input_gradients = input_gradient(layer_num, hidden_outputs,
output_gradients, weights_hidden_output)
    weights_input_hidden_updated = weights_input_hidden + learning_rate
* np.dot(input_gradients, inputs.T)

    return error, weights_hidden_output_updated,
weights_input_hidden_updated
```

```
def backpropagation_sigmoid2(inputs, labels, weights_hidden_output,
weights_input_hidden, activation_hidden, activation_output,
learning_rate, layer_num,lambda_coef):
    hidden_outputs, final_outputs = forward_pass(inputs,
weights_input_hidden, weights_hidden_output, activation_hidden,
activation_output)
    desired = desired_output(labels, 0)
    error = find_error(final_outputs, desired)
    output_gradients = output_gradient(error, final_outputs)
    weights_hidden_output_updated = (1-
learning_rate*lambda_coef)*weights_hidden_output + learning_rate *
np.dot(output_gradients,hidden_outputs.T)
    input_gradients = input_gradient(layer_num, hidden_outputs,
output_gradients, weights_hidden_output)
    weights_input_hidden_updated = (1-
learning_rate*lambda_coef)*weights_input_hidden + learning_rate *
np.dot(input_gradients, inputs.T)

    return error, weights_hidden_output_updated,
weights_input_hidden_updated
```

```
def train2(inputs, labels,test_images
,test_label,weights_hidden_output, weights_input_hidden,
activation_hidden, activation_output, learning_rate, layer_num,
batch_size, epochs):
    trainingError = 0.0
    startTime = time.time()

    for epoch in range(epochs):
        cumulativeError = 0.0

        for i in range(batch_size):
            flattened_data = np.reshape(inputs[i], (784, 1)) / 255
            error, weights_hidden_output, weights_input_hidden =
backpropagation_sigmoid(flattened_data, labels[i],
weights_hidden_output, weights_input_hidden, activation_hidden,
activation_output, learning_rate, layer_num)
            cumulativeError += np.sum(error**2 * 0.5)
        epochErrorMean = cumulativeError / batch_size
        trainingError += epochErrorMean

        print("Epoch Completed: ", epoch + 1, "\nMean Squares Error of 
Epoch: ", epochErrorMean)

    trainingErrorMean = trainingError / epochs
    stopTime = time.time()
    print("Training Completed!\nCPU Time: ", stopTime - startTime, " 
seconds", "\nOverall Error:", trainingErrorMean)

    cumulativeError_2 = 0.0
    numOfMissClassification = 0

    for i in range(10000):
        testData = np.reshape(test_images[i], (784, 1)) / 255
        layer_output,final_output=forward_pass(testData, 
weights_input_hidden, weights_hidden_output,activation_hidden, 
activation_output)
        desired=desired_output(test_label[i],0)
        error_2=find_error(final_output,desired)
        cumulativeError_2 += np.sum(error_2**2 * 0.5)

        predicted = np.argmax(final_output)

        if test_label[i] != predicted:
            numOfMissClassification += 1

    misclasificationPercentage = numOfMissClassification / 10000 * 100

    testSetErrorMean = cumulativeError_2 / 10000

    print("Number Of 
Misclasification:",numOfMissClassification,"\nError Percentage: 
%",misclasificationPercentage,"\nTest Data Mean Square 
Error:",testSetErrorMean)
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = np.reshape(X_train, (784, 60000)) / 255
X_test = np.reshape(X_test, (784, 10000)) / 255
```

```
#Randomly select 1250 data because of OVERUSE RAM may shutdown the 
system
num_samples_to_keep = 1250
indices_to_keep = np.random.choice(60000, num_samples_to_keep, 
replace=False)
X_train_1250 = X_train[:, indices_to_keep]
```

```
Y_train_1250 = y_train[indices_to_keep]
```

```
num_samples_to_keep = 1250
indices_to_keep2 = np.random.choice(10000, num_samples_to_keep,
replace=False)
X_test_1250 = X_test[:, indices_to_keep2]
Y_test_1250 = y_test[indices_to_keep2]
```

```
X_train_flat = X_train_1250.reshape(X_train.shape[0], -1)#1D ARRAY OF
DATA

X_test_flat = X_test_1250.reshape(X_test.shape[0], -1)
print(X_train_1250.shape)
```

```
Train_mean=np.mean(X_train_flat,axis=1)#Mean vectors
Test_mean=np.mean(X_test_flat,axis=1)
Train_std=np.std(X_train_flat,axis=1)#variance vectors
Test_std=np.std(X_test_flat,axis=1)
Train_std_nonzero = np.nan_to_num(Train_std, nan=1.0)  # Replace NaN
values with 1.0
Test_std_nonzero = np.nan_to_num(Test_std, nan=1.0)  # Replace NaN
values with 1.0
Stand_X_train=(X_train_flat-Train_mean[:,
np.newaxis])/Train_std_nonzero[:, np.newaxis]
Stand_X_test=(X_test_flat-Test_mean[:, np.newaxis])/Test_std_nonzero[:,
np.newaxis]
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.hist(Stand_X_train.flatten(), bins=50, color='blue', alpha=0.7)
plt.title('Distribution Before PCA')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
```

```
Test_cov=np.cov(Stand_X_test,rowvar=  False)
Test_eigen_val,Test_eigen_vec=np.linalg.eig(Test_cov)
Train_cov=np.cov(Stand_X_train,rowvar=  False)
Train_eigen_val,Train_eigen_vec=np.linalg.eig(Train_cov)
```

```
sort_index2=np.argsort(Test_eigen_val)[::-1]
Test_eigen_val_sorted=Train_eigen_val[sort_index2]
Test_eigen_vec_sorted=Test_eigen_vec[:,sort_index2]
Reduced_size=100
```

```
PCA_selectedTest=Test_eigen_vec_sorted[:,:Reduced_size]
PCA_selected3=PCA_selectedTest.real
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.hist(PCA_selected2.flatten(), bins=50, color='blue', alpha=0.7)
plt.title('Distribution After PCA')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
```

```

sort_index=np.argsort(Train_eigen_val)[::-1]
Train_eigen_val_sorted=Train_eigen_val[sort_index]
Train_eigen_vec_sorted=Train_eigen_vec[:,sort_index]
Reduced_size=100
PCA_selected=Train_eigen_vec_sorted[:,:Reduced_size]
PCA_selected2=PCA_selected.real
```

```
def train3(inputs, labels,test_images
,test_label,weights_hidden_output, weights_input_hidden,
activation_hidden, activation_output, learning_rate, layer_num,
batch_size, epochs):
    trainingError = 0.0
    startTime = time.time()

    for epoch in range(epochs):
        cumulativeError = 0.0

        for i in range(batch_size):
            flattened_data = np.reshape(inputs[i], (100, 1))
            error, weights_hidden_output, weights_input_hidden =
backpropagation_sigmoid(flattened_data, labels[i],
weights_hidden_output, weights_input_hidden, activation_hidden,
activation_output, learning_rate, layer_num)
            cumulativeError += np.sum(error**2 * 0.5)

        epochErrorMean = cumulativeError / batch_size
        trainingError += epochErrorMean

        print("Epoch Completed: ", epoch + 1, "\nMean Squares Error of
Epoch: ", epochErrorMean)

    trainingErrorMean = trainingError / epochs
    stopTime = time.time()
    print("Training Completed!\nCPU Time: ", stopTime - startTime, "
seconds", "\nOverall Error:", trainingErrorMean)
```

```
    cumulativeError_2 = 0.0
    numOfMissClassification = 0

    for i in range(1250):
        testData = np.reshape(test_images[i], (100, 1))
        layer_output,final_output=forward_pass(testData,
weights_input_hidden, weights_hidden_output,activation_hidden,
activation_output)
        desired=desired_output(test_label[i],0)
        error_2=find_error(final_output,desired)
        cumulativeError_2 += np.sum(error_2**2 * 0.5)

        predicted = np.argmax(final_output)

        if test_label[i] != predicted:
            numOfMissClassification += 1

    misclasificationPercentage = numOfMissClassification / 1250 * 100

    testSetErrorMean = cumulativeError_2 / 1250

    print("Number Of
Misclasification:",numOfMissClassification,"\nError Percentage:
%",misclasificationPercentage,"\nTest Data Mean Square
Error:",testSetErrorMean)
```

```
weights_hidden2, weights_output2=initialize_weights2(300,100)
learning_rate=0.09
epochs=50
batch_size=1250
layer_num=300
train3(PCA_selected2,Y_train_1250,PCA_selected3
,Y_test_1250,weights_output2, weights_hidden2,activation_hidden_1,
activation_output_1, learning_rate,layer_num,batch_size,epochs)
```