



**Birzeit University Department of Electrical  
& Computer Engineering First Semester, 2024/2025**

**ENCS5343 -Computer Vision**

**Course Project**

---

**Partners:**

**Haneen Odeh 1210716**

**Leyan Burait 1211439**

**Instructor: Dr. Aziz Qaroush**

**Date : January 15, 2025**

## Objectives

**Abstract**—Handwriting recognition is a computer vision problem that includes having a computer recognize handwritten text and convert it into a format that can be read by machines. Many researches aimed to build suitable models for these task in different languages including Arabic language. This paper aims on studying different architectures aimed to solve this task beside well known architectures. Moreover, we will explore the effect of data augmentation as well as the function of hyper parameter tuning in optimizing model performance. This is done by a comparative study using loss functions, optimization, data augmentation and transfer learning.

**Index Terms**—CNN, Epoch, Kernel, KNN, k-Nearest Neighbors, NN, ReLU.

## Contents

Objectives .....	2
Table of figure .....	4
1.INTRODUCTION .....	5
2. BACKGROUND .....	5
2.1. Convolutional Neural Networks .....	5
2.1.1 Convolutional Layer .....	5
2.1.2 Pooling Layer .....	7
2.1.3 Fully Connected Layer .....	7
2.2. Activation Functions.....	8
2.2.1 Sigmoid.....	8
2.2.2 ReLU .....	8
2.3 Hyper-parameters .....	9
2.3.1 Learning Rate .....	9
2.3.2 Batch Size .....	9
2.4Regularization Techniques .....	9
2.5Optimizers .....	11
2.5.1Gradient Descent: .....	11
2.5.2 Adam Optimizer:.....	11
3.RELATED WORK.....	<b>Error! Bookmark not defined.</b>
4 EXPERIMENT .....	12
4.1_Experimental setup .....	12
4.2 Evaluation Metrics.....	12
4.3 Dataset .....	<b>Error! Bookmark not defined.</b>
4.4 Results .....	<b>Error! Bookmark not defined.</b>
4.1.1 Task1.....	13
4.4.2 Without data argumentation .....	<b>Error! Bookmark not defined.</b>
4.4.3With Data Augmentation .....	<b>Error! Bookmark not defined.</b>
4.1.3_Task 3.....	21
4.1.4_Task 4 .....	26

## Table of figure

Figure 1: Basic Structure of Convolutional Neural Networks.....	5
Figure 2:convolutional layer.....	6
Figure 3: Illustration of Convolution Operation .....	6
Figure 4:EQUATIO's of Hout & Wout.....	7
Figure 5:Illustration of Pooling Operation.....	7
Figure 6:output volume equation .....	7
Figure 7: sigmoid equation.....	8
Figure 8: Sigmoid Activation Function [26]. .....	8
Figure 9: ReLU equation .....	8
Figure 10: ReLU Activation Function [26].....	8
Figure 11: Learning VS Learning Rate.....	9
Figure 12:can see how mini-batch gradient descent works when the mini-batch size is equal to two	
Differences Between Epoch, Batch, and Mini-batch   Baeldung on Computer Science.....	9
Figure 13: Early stopping for the case of having the validation error not decreasing	
<a href="https://theaisummer.com/static/7a6353ed78b045f32e4ac39b0b4d66d2/d61c2/early-stopping.png">https://theaisummer.com/static/7a6353ed78b045f32e4ac39b0b4d66d2/d61c2/early-stopping.png</a> .....	10
Figure 14:L1 Regularization equation .....	10
Figure 15:L2 Regularization equation .....	10
Figure 16 Illustration of dropout	
<a href="https://www.researchgate.net/publication/365947215/figure/fig2/AS:11431281152098150@1682043362609/Illustration-of-traditional-dropout.png">https://www.researchgate.net/publication/365947215/figure/fig2/AS:11431281152098150@1682043362609/Illustration-of-traditional-dropout.png</a> .....	11
Figure 17:update weight equation.....	11
Figure 18:The update rule for the parameters using Adam.....	12
Figure 19: Custom Architecture	
<a href="https://drive.google.com/file/d/143j01fG7Z_xlVj8FeQRzya5O2ot0JMps/view?usp=sharing">https://drive.google.com/file/d/143j01fG7Z_xlVj8FeQRzya5O2ot0JMps/view?usp=sharing</a> .....	<b>Error!</b>
<b>Bookmark not defined.</b>	
Figure 20: A Hybrid Deep Model for Recognizing Arabic Handwritten word. <b>Error! Bookmark not defined.</b>	
Figure 21:accuracy equation .....	12
Figure 22: Samples from the dataset .....	<b>Error! Bookmark not defined.</b>
Figure 23: Accuracy Curves of Baseline.....	<b>Error! Bookmark not defined.</b>
Figure 24: Loss Curves of Baseline .....	<b>Error! Bookmark not defined.</b>
Figure 25:ARABIC DATA without data Augmentation.....	<b>Error! Bookmark not defined.</b>
Figure 26:Initial Epochs (1-5) .....	22
Figure 27:from 6 to 9 .....	22
Figure 28:from 9 to 14 .....	22
Figure 29:from 13 to 15 .....	23
Figure 30: from 13 to 25 .....	23
Figure 31: from 23 to 30 finally .....	24
Figure 32:Loss vs. Epochs & accuracy vs. Epochs.....	25
Figure 33:a high validation accuracy of 87.11% and a test accuracy of 88.70%.....	26
Figure 34:validation accuracy reached a maximum of 96.5%, at which point early stopping was triggered .....	27
Figure 35: accuracy result of task 4 .....	29
Figure 36:Scaling vs. Accuracy & Rotation vs. Accuracy .....	30
Figure 37:Scaling vs. Accuracy & Rotation vs. Accuracy .....	31

## 1. INTRODUCTION

Automatic handwriting recognition is an important part of computer vision and is about recognizing human handwriting from different sources like paper, pictures, touch-screens, or other devices. This can be done either by scanning (offline) or using a pen on a device (online) [1]. Recognizing Arabic handwriting has become more important today because Arabic characters are detailed and Arabic is used more in areas like business, education, and government [2]. This technology is used for turning paper documents into digital files, typing on mobiles, and turning handwritten notes into typed text [3]. The challenge in this field comes from how different each person's handwriting is and the special features of Arabic writing. But, recent progress with machine learning and deep neural networks has really improved Arabic handwriting recognition, making it a field that is growing and showing a lot of promise [4]. Even though a lot of research has been done on handwriting recognition in languages like English, French, and Chinese, there is a growing need and potential for more work in Arabic handwriting recognition. This highlights the importance of continuing to research and come up with new ideas in this area [5].

Handwriting recognition has been studied using different methods like support vector machines (SVMs), K-nearest neighbors (KNNs), neural networks (NNs), and recently, convolutional neural networks (CNNs) [6].

## 2. BACKGROUND

### 2.1. Convolutional Neural Networks

Convolutional Neural Networks (CNN), is a deep learning model that consists of three main components, convolutional layer, pooling layer, and fully-connected layer. It is mainly designed for processing visual data, such as images and videos. They have influenced computer vision and made incredible progress possible in areas such as object detection, image segmentation, and image recognition. Figure below shows the basic structure of CNNs, where the first part is feature extraction that is done by the convolutional and pooling layers, followed by the classification part done by the fully connected layers.

#### 2.1.1 Convolutional Layer

The convolutional layer is the core building block in the CNN, it basically contains most of the computations that is done for extracting features.

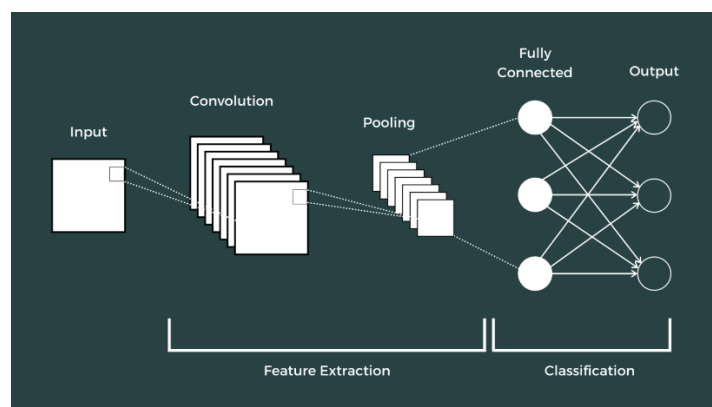


Figure 1: Basic Structure of Convolutional Neural Networks

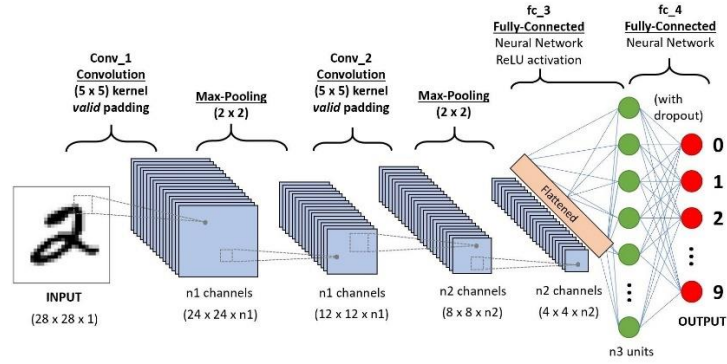


Figure 2:convolutional layer

Essentially, it takes an input image and runs it through a certain number of  $f$  filters, each of which has a given kernel size that is identical in depth but much less in width and height than the input. The kernel is shifted with a shifting amount known as the Stride across the width and height of the input image. A dot product is then calculated between them, and the resultant output is referred to as the feature map, which provides the kernel's response at each spatial position of the image [23]

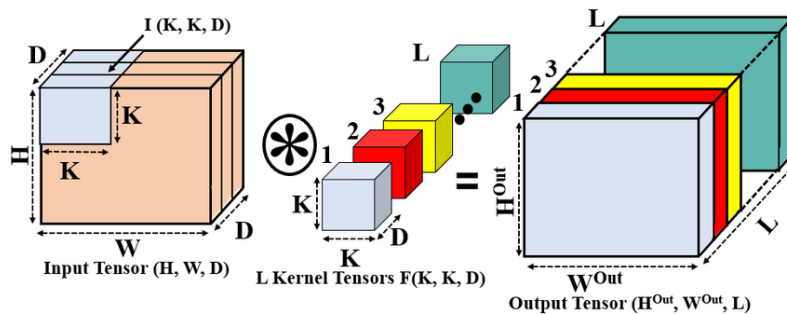
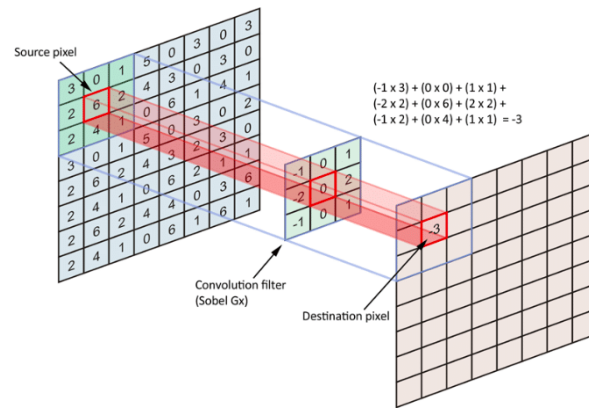


Figure 3: Illustration of Convolution Operation

The output of the convolutional layer is effected by several hyperparameters which are [25]:

- 1) Kernel Size: is the sliding window size, is preferred to be an odd number that is much smaller than the input size. 3 and 5 are preferable.

- 2) Stride: the number of pixels the kernel window will move during the convolution step. It is usually set to 1 and can be raised if decreasing the input size is the goal.
- 3) Padding: Is the process of adding zeros to an image's border . The kernel can thoroughly filter each point in an input image via padding, guaranteeing that even the edges are handled correctly.
- 4) Number of Filters: Number of filters determines how many patterns or features the layer will search for. The output height 1 and width 2 is as the following:

$$H_{out} = 1 + \frac{H_{in} + (2 \cdot \text{pad}) - K_{height}}{s} \quad (1)$$

$$W_{out} = 1 + \frac{W_{in} + (2 \cdot \text{pad}) - K_{width}}{s} \quad (2)$$

Figure 4:EQUATIO's of Hout & Wout

### 2.1.2 Pooling Layer

Pooling layer, or the down sampling layer, is the layer that works in reducing the dimensionality of the input. It is similar to the convolutional layer in the general concept, where a filter is passed across the input. However, the pooling filter has no weights, the filter creates the output array by applying an aggregation function to the data in its receptive field. To be noted that the size of the filter in the pooling is preferred to be even [25]. The aggregation function can be out of two:

- 1) Max Pooling: It picks the pixel to send to the output array with the highest value.
- 2) Average pooling: Determines the average value to send to the output array from the receptive field.

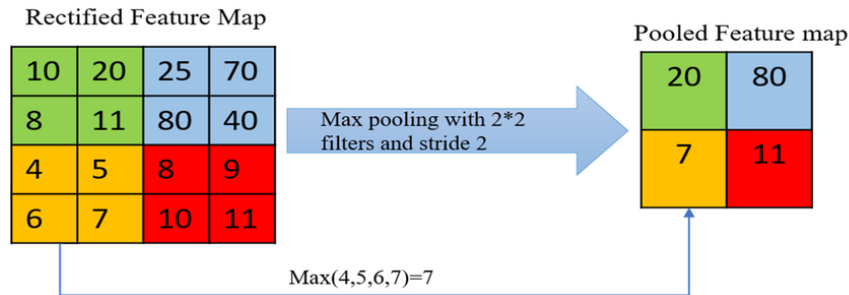


Figure 5:Illustration of Pooling Operation

Assuming a pooling kernel of spatial size  $F$ , a stride of  $S$ , and an activation map of size  $W \times W \times D$ , the output volume is calculated as:

$$W_{out} = \frac{W - F}{S} + 1$$

Figure 6:output volume equation

The result will be an output that is  $W_{out} \times W_{out} \times D$ .

### 2.1.3 Fully Connected Layer

The fully connected layers of a CNN, tend to be towards the end. Each neuron in these layers is connected to every other neuron in the layer before, forming a dense network of connections. These layers are responsible for combining the high-level features learned by the convolutional layers to make final predictions or classifications. In image classification tasks, probability scores for various classes are often generated by feeding the output of the final fully connected layer into a softmax activation function [25].

## 2.2. Activation Functions

Activation functions is a transfer function that is used to determine the output of a neural network. The Activation Functions can be divided linear and non-linear functions.

### 2.2.1 Sigmoid

The sigmoid is defined as:

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

Figure 7: sigmoid equation

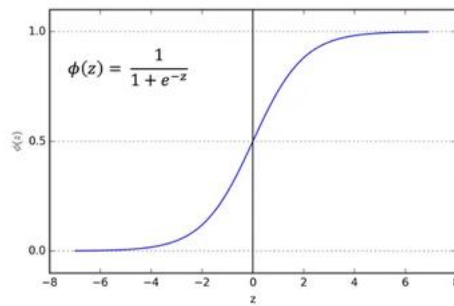


Figure 8: Sigmoid Activation Function [26].

### 2.2.2 ReLU

Rectified Linear Unit or ReLU is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Figure 9: ReLU equation

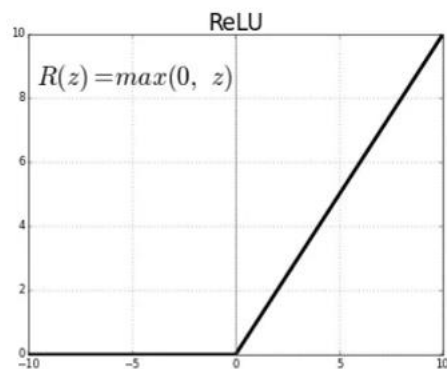


Figure 10: ReLU Activation Function [26].



## 2.3 Hyper-parameters

### 2.3.1 Learning Rate

Learning rate defines the rate of which an algorithm converges to a solution. It is one of the most important hyper-parameters in machine learning, and choosing the correct value affects the learning process [28].

If using gradient descent, the learning rate forms the step size, such that small learning rate will slow the learning process, a larger one will diverge and never meet the optimal solution. Even-though, the learning rate can vary through the learning process, where it can be changes based on the learning status [28]

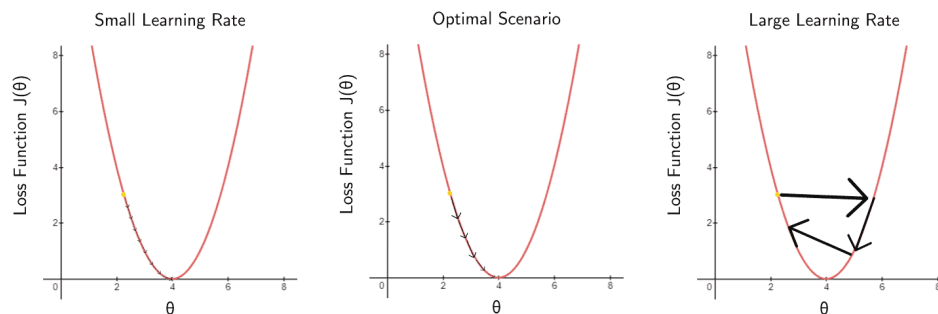


Figure 11: Learning VS Learning Rate

### 2.3.2 Batch Size

in each batch in the learning process. There are three types of batches:

- 1) Batch gradient descent: uses all the samples at once.
- 2) Stochastic gradient descent: for each epoch, one sample is used.
- 3) Mini-batch gradient descent: a small subset of the training is used in each epoch.

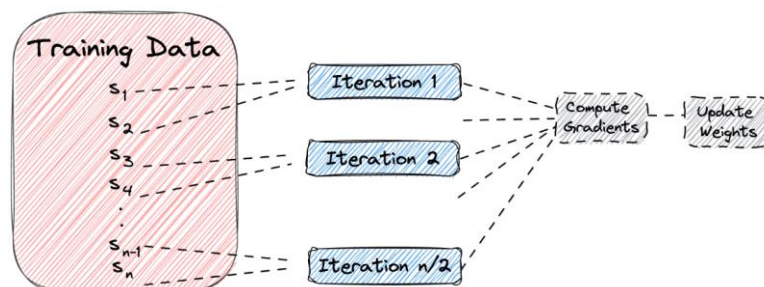


Figure 12: can see how mini-batch gradient descent works when the mini-batch size is equal to two [Differences Between Epoch, Batch, and Mini-batch | Baeldung on Computer Science](#)

## 2.4 Regularization Techniques

In machine learning, regularization is a collection of techniques designed for reducing overfitting. After training, the majority of models show excellent performance on training sets but struggle with generalization. Regularization techniques seek to minimize training error while also reducing overfitting, which results in simpler models.

- 1) **Early Stopping:** Early stopping is an easy technique that entails stopping the neural network's training at an earlier epoch. This is carried out in order to avoid overfitting the training set in the case that the training error becomes too low. The early stopping will keep it from approaching zero. This can be achieved by keeping an eye on changes in metrics like validation accuracy and error as well as weight vector changes [31]

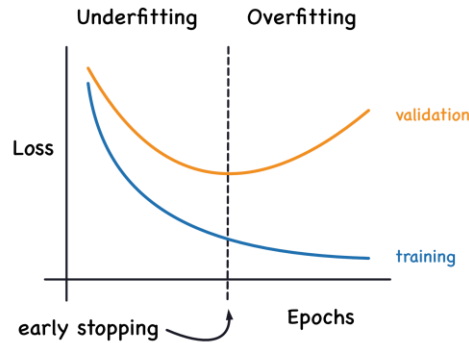


Figure 13: Early stopping for the case of having the validation error not decreasing  
<https://theaisummer.com/static/7a6353ed78b045f32e4ac39b0b4d66d2/d61c2/early-stopping.png>

- 2) **L1 Regularization:** L1 regularization adds the absolute value of the weights to the loss function. This basically does feature selection by decreasing some weights to become exactly zero [33].

$$L1(\mathbf{w}) = \lambda \sum_i |w_i|$$

Figure 14:L1 Regularization equation

- 3) **L2 Regularization:** L2 regularization adds the square of the weights to the loss function. This helps in distributing the importance to all features, it also reduces the weights so they can't grow too large [33].

$$L2(\mathbf{w}) = \lambda \sum_i w_i^2$$

Figure 15:L2 Regularization equation

- 4) **Dropout:** Dropout, a noise injection technique that is used as a regularization one. This is done by ignoring the outputs of randomly picked layers at the training temporarily. As a result, during training, every layer update is carried out using an individual "view" of the preset layer, it's similar to training many neural networks with various architectures simultaneously [31].

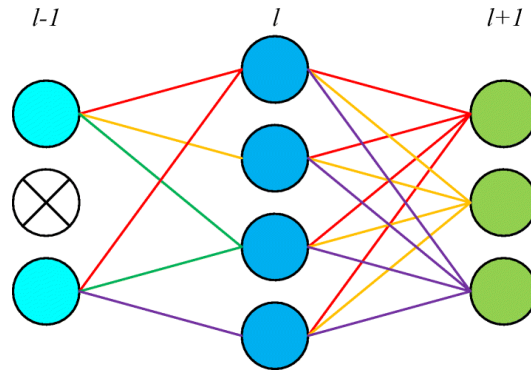


Figure 16 Illustration of dropout

<https://www.researchgate.net/publication/365947215/figure/fig2/AS:11431281152098150@1682043362609/Illustration-of-traditional-dropout.png>

- 5) **Data Augmentation:** Data augmentation is useful when there isn't enough data to train a neural network, as it improves network generalization in situations where deep neural networks require big training datasets. To accomplish this, random transformations are applied to the data, such as cropping, rotating, or flipping images, to create new training examples [33]

## 2.5 Optimizers

Optimizers are mathematical functions that are used to minimize the error in order to enhance the system efficiency. They mainly are effected by model's learnable parameters i.e Weights & Biases. They aid in reducing losses by understanding how to alter the neural network's weights and learning rate.

### 2.5.1 Gradient Descent:

Gradient descent is an optimization algorithm that minimizes a given function to its local minimum by iteratively adjusting its parameters based on a convex function. Gradient Descent moves in the opposite direction of the sharpest climb, reducing a loss function iteratively. To discover minima, it depends on the derivatives of the loss function [29]. It is computer as:

$$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial \text{Loss}}{\partial W_{\text{old}}}$$

Figure 17:update weight equation

**Advantages** Easy to understand & Easy to implement.

**Disadvantages** :Very slow & Requires large memory.

### 2.5.2 Adam Optimizer:

A popular optimizer computes adaptive learning rates for each parameter. When working with complex problems requiring several data points or parameters, the approach is incredibly effective. It is effective and uses little memory. It makes logical sense to combine the gradient descent with momentum and RMSP algorithms [29].

The update rule for the parameters using Adam is given by:

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{S_{dw_t}} - \epsilon} * V_{dw_t}$$

$$b_t = b_{t-1} - \frac{\eta}{\sqrt{S_{db_t}} - \epsilon} * V_{db_t}$$

Figure 18: The update rule for the parameters using Adam

These equations show the update rules for the weight  $w_t$  and bias  $b_t$  at time step  $t$ . Advantages Easy to implement, Computationally efficient & Little memory requirements.

### 3.Dataset

The AHAWP dataset (Arabic Handwritten Automatic Word Processing) is designed to support the development and evaluation of Arabic handwritten text recognition systems. It contains 10 unique Arabic words, each written by 82 individuals, with each writer providing 10 samples per word, resulting in 8,144 word images in total. The dataset is structured to focus on word-level data, facilitating targeted feature extraction. Each writer is identified by a unique, anonymous user ID, ensuring privacy while allowing for sample tracking. The diversity of handwriting styles and the dataset's consistent structure make it ideal for testing feature extraction techniques and evaluating system performance.

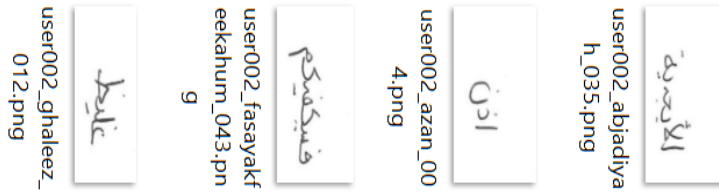


Figure 19 : Samples from the dataset

## 4 Experiment

### 4.1\_Experimental setup

About 5 different CNN networks were trained and tuned using the same set of hyperparameters such as number of filters, filter size, stride, padding, pooling size, with using an early stop condition if there is no improvement on the models' performance for 10 epochs. At first grid search function was used for the tuning process, after that and after we obtained the best tuned parameters the networks were built accordingly and that's for minimizing the total process time. All the models were built using Pytorch framework and trained on an NVIDIA GTX 1060 GPU.

### 4.2 Evaluation Metrics

- 1) Accuracy: The accuracy metric provides an overall assessment of the model's ability to correctly predict the output on the complete dataset. Each individual in the dataset contributes equally to the accuracy score, as every unit holds the same weight.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Figure 20: accuracy equation

2) Loss: The loss metric is a crucial component in training machine learning models, as it quantifies the difference between the model's predictions and the actual target values. In the context of handwritten text recognition, the most commonly used loss functions include categorical cross-entropy for classification tasks. This loss function calculates the error between the predicted probability distribution over classes and the true class label, encouraging the model to improve its predictions. Minimizing the loss function during training helps the model learn better representations of the input data, ultimately improving its accuracy and generalization capabilities for tasks like recognizing handwritten Arabic words.

#### 4.1.1 Task1

### 1. Data Loading and Preprocessing

The dataset is loaded using the `load_data` function, which reads images from a directory and resizes them to a fixed size (e.g., 64x64). Each image is associated with a label (user name). The images are normalized by scaling pixel values to the range  $[0, 1]$ , and labels are converted from strings to integers and then to one-hot encoded vectors. The dataset is split into training (60%), validation (20%), and testing (20%) sets to evaluate the model's performance on unseen data.

### 2. Building the Models

As was mentioned earlier, 5 different networks were built and tuned by different parameters values using grid search to find the best ones for each network, the main difference between all networks is the number of layers in each network and the parameters used as shown below in the table

Networks	Number of Filters	Filter Size	Stride	Padding	Dense layers
1-layer CNN	32 or 64 or 128 or 256	(3,3) or (5,5) or (7,7)	(1,1) or (2,2)	Same "Zero padding"	256
2-layers CNN	32 or 64 or 128 or 256	(3,3) or (5,5) or (7,7)	(1,1) or (2,2)	Same "Zero padding"	256
4-Layers CNN	32 or 64 or 128 or 256	(3,3) or (5,5) or (7,7)	(1,1) or (2,2)	Same "Zero padding"	256
5-Layers CNN	32 or 64 or 128 or 256	(3,3) or (5,5) or (7,7)	(1,1) or (2,2)	Same "Zero padding"	256
7-layers CNN	32 or 64 or 128 or 256	(3,3) or (5,5) or (7,7)	(1,1) or (2,2)	Same "Zero padding"	256

### 3. The Processes & Techniques used in building models

The networks were built using a `build_cnn` function, this function has its parameters like number of filters, filter size, padding, stride. Also it has the activation function used for cnn layers, the activation function was used is the `relu` function, as it is the best one among the ones we learned, for the pooling layers, `max_pooling2D` was used with pooling size (2,2)

```
def build_cnn_model(input_shape, num_classes, layer_configs):
    model = Sequential()
    for i, config in enumerate(layer_configs):
        model.add(
            Conv2D(
                filters=config["filters"],
                kernel_size=config["kernel_size"],
                strides=config.get("stride", (1, 1)), # Default stride is (1, 1)
                padding=config.get("padding", "same"), # Default padding is "same"
                activation="relu",
                input_shape=input_shape if i == 0 else None,
            )
        )
    if config.get("pooling", False):
        model.add(MaxPooling2D(pool_size=(2, 2)))
```

A flatten layer was added to flatten input data (e.g., 2D image) into a 1D vector so it can be passed into fully connected layers, then a fully connected layer with 256 neurons and the ReLU activation function was added, which introduces non-linearity to the model, a dropout layer that randomly drops 30% of the neurons during training was added to prevent overfitting, finally, the output layer was added with 82 neurons, where each neuron represents a class, and uses the softmax activation to output probabilities for each class.

The optimizer used is Adam optimizer with learning rate equal 0.001 as it was the best value after doing grid search for it.

Note: A seed point was used to obtain consistent results each time the code is run.

```
model.add(Flatten())
model.add(Dense(256, activation="relu")) # Fixed dense neurons at 128
model.add(Dropout(0.3))
model.add(Dense(num_classes, activation="softmax"))

optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"])
return model
```

Then the grid search was used to find the best parameters, number of epochs was set to 40 epochs, batch size was set to 64 (it's the best value after testing many other values), the early stopping was used & patience time was set to 15 epochs

```
def grid_search_cnn(X_train, y_train, X_val, y_val, input_shape, num_classes, layer_configs):
    best_model = None
    best_val_accuracy = 0

    print(f"Training with layer configurations: {layer_configs}")
    model = build_cnn_model(input_shape, num_classes, layer_configs)

    # EarlyStopping callback to stop training when validation loss stops improving
    early_stopping = EarlyStopping([monitor='val_loss', patience=10, restore_best_weights=True])

    # Train the model and capture the history
    history = model.fit(
        X_train,
        y_train,
        epochs=40,
        batch_size=64,
        validation_data=(X_val, y_val),
        verbose=1,
        callbacks=[early_stopping]
    )
```

## 4. CNN Networks

After doing grid search for all networks we have, we then built them by the best hyper parameters outputted from the grid search process as can be seen below

```
# Define Layer configurations for each network
layer_configs_1 = [
    {"filters": 64, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 1
]

layer_configs_2 = [
    {"filters": 16, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 1
    {"filters": 32, "kernel_size": (5, 5), "stride": (2, 2), "padding": "same", "pooling": True}, # Layer 2
]
#3,3->5,5
layer_configs_4 = [
    {"filters": 32, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 1
    {"filters": 64, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 2
    {"filters": 128, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 3
    {"filters": 256, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 4
]

layer_configs_5 = [
    {"filters": 32, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 1
    {"filters": 64, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 2
    {"filters": 128, "kernel_size": (5, 5), "stride": (2, 2), "padding": "same", "pooling": True}, # Layer 3
    {"filters": 256, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 4
    {"filters": 256, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": False}, # Layer 5
]

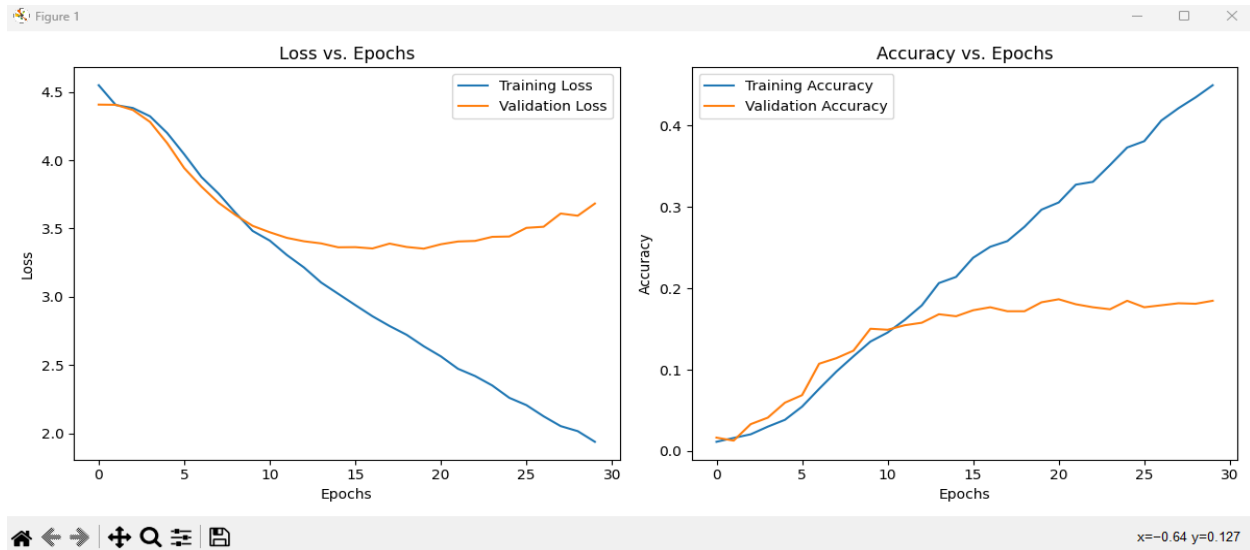
layer_configs_7 = [
    {"filters": 16, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 1
    {"filters": 32, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 2
    {"filters": 64, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 3
    {"filters": 32, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 4
    {"filters": 32, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": False}, # Layer 5
    {"filters": 64, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": False}, # Layer 6
    {"filters": 32, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": False}, # Layer 7
]

# Perform grid search for each network
input_shape = (64, 64, 1)
num_classes = 82
```

### CNN network with 1-layer

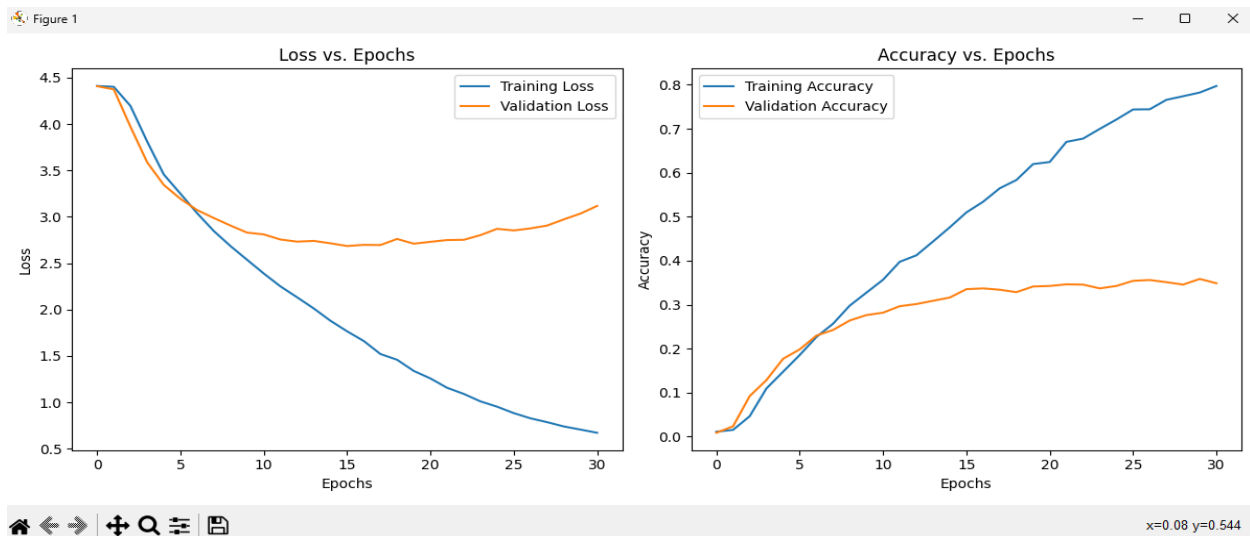
This is a network of one cnn layer, this is the simplest network, and obviously it cant handle complex patterns and relationships. The best validation accuracy it gets about 10% which consider low accuracy

```
||||| 20s 262ms/step - accuracy: 0.3468 - loss: 2.3741 - val_accuracy: 0.1743 - val_loss: 3.4389
Epoch 25/40
||||| 20s 261ms/step - accuracy: 0.3606 - loss: 2.3039 - val_accuracy: 0.1848 - val_loss: 3.4413
Epoch 26/40
||||| 21s 278ms/step - accuracy: 0.3849 - loss: 2.2032 - val_accuracy: 0.1768 - val_loss: 3.5048
Epoch 27/40
||||| 20s 261ms/step - accuracy: 0.4041 - loss: 2.1397 - val_accuracy: 0.1793 - val_loss: 3.5126
Epoch 28/40
||||| 20s 259ms/step - accuracy: 0.4128 - loss: 2.0616 - val_accuracy: 0.1817 - val_loss: 3.6095
Epoch 29/40
||||| 23s 298ms/step - accuracy: 0.4401 - loss: 2.0244 - val_accuracy: 0.1811 - val_loss: 3.5935
Epoch 30/40
||||| 22s 281ms/step - accuracy: 0.4491 - loss: 1.9443 - val_accuracy: 0.1848 - val_loss: 3.6826
Best Validation Accuracy: 0.18661755323410034
```



### CNN network with 2-layers

This network was built with 2 cnn layers which can implement more complex patterns than the previous network, and that can be seen by its accuracy which is higher about 36% and lower loss



```
Epoch 27/40
77/77 5s 39ms/step - accuracy: 0.7281 - loss: 0.8671 - val_accuracy: 0.3560 - val_loss: 2.8748
Epoch 28/40
77/77 3s 37ms/step - accuracy: 0.7648 - loss: 0.7979 - val_accuracy: 0.3511 - val_loss: 2.9054
Epoch 29/40
77/77 3s 37ms/step - accuracy: 0.7719 - loss: 0.7549 - val_accuracy: 0.3456 - val_loss: 2.9720
Epoch 30/40
77/77 5s 38ms/step - accuracy: 0.7698 - loss: 0.7408 - val_accuracy: 0.3585 - val_loss: 3.0346
Epoch 31/40
77/77 3s 38ms/step - accuracy: 0.7818 - loss: 0.7231 - val_accuracy: 0.3487 - val_loss: 3.1170
Best Validation Accuracy: 0.3585021495819092
=====
```

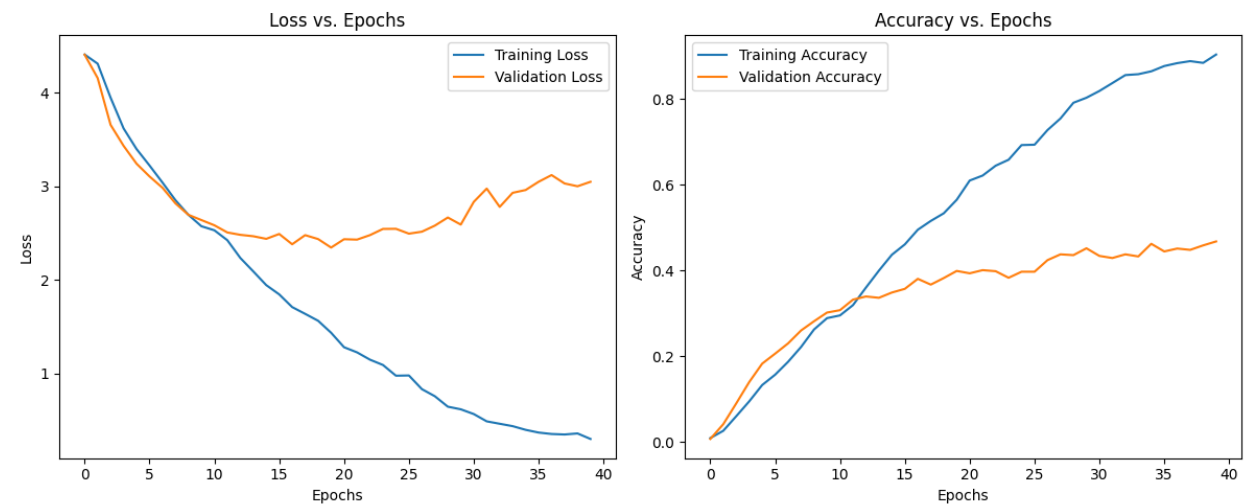


## 🚦 CNN network with 4-layers

This network showed the **highest validation** accuracy which is 47%, but it couldn't solve the overfitting issue because the data augmentation wasn't used in this part, using data augmentation decreases the overfitting issue.

```
layer_configs_4 = [  
    {"filters": 32, "kernel_size": (3, 3), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 1  
    {"filters": 64, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 2  
    {"filters": 128, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 3  
    {"filters": 256, "kernel_size": (5, 5), "stride": (1, 1), "padding": "same", "pooling": True}, # Layer 4  
]
```

Figure 1

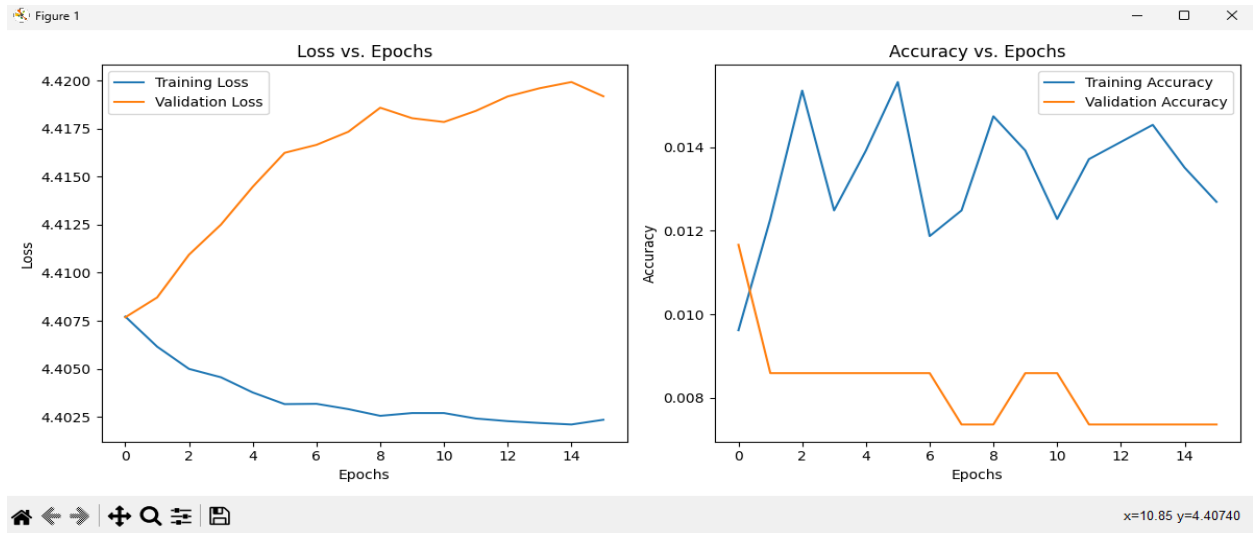


🏠 ⬅ ➡ 🔍 📄 📁

```
77/77 31s 402ms/step - accuracy: 0.8534 - loss: 0.4581 - val_accuracy: 0.4622 - val_loss: 2.9620  
Epoch 36/40  
77/77 33s 422ms/step - accuracy: 0.8714 - loss: 0.3952 - val_accuracy: 0.4444 - val_loss: 3.0500  
Epoch 37/40  
77/77 40s 402ms/step - accuracy: 0.8832 - loss: 0.3685 - val_accuracy: 0.4512 - val_loss: 3.1217  
Epoch 38/40  
77/77 32s 419ms/step - accuracy: 0.8877 - loss: 0.3498 - val_accuracy: 0.4481 - val_loss: 3.0320  
Epoch 39/40  
77/77 32s 410ms/step - accuracy: 0.8776 - loss: 0.3763 - val_accuracy: 0.4586 - val_loss: 3.0020  
Epoch 40/40  
77/77 31s 405ms/step - accuracy: 0.9051 - loss: 0.3068 - val_accuracy: 0.4678 - val_loss: 3.0498  
Best Validation Accuracy: 0.46777164936065674
```

### CNN network with 5-layers

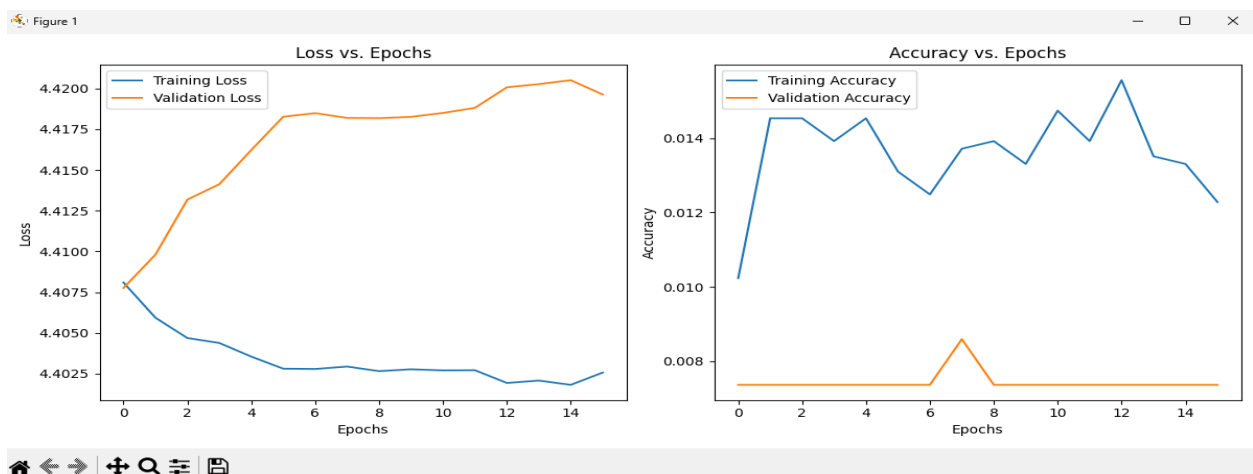
This network showed a lower performance among the previous ones with accuracy is approximately 1% which is very low and that may be during many reasons such as overfitting, complex patterns or the grid search didn't give accurate parameters



```
Epoch 14/40
77/77 51s 405ms/step - accuracy: 0.0147 - loss: 4.4016 - val_accuracy: 0.0074 - val_loss: 4.4203
Epoch 15/40
77/77 26s 338ms/step - accuracy: 0.0207 - loss: 4.4014 - val_accuracy: 0.0074 - val_loss: 4.4201
Epoch 16/40
77/77 45s 589ms/step - accuracy: 0.0143 - loss: 4.4019 - val_accuracy: 0.0074 - val_loss: 4.4197
Best Validation Accuracy: 0.010435850359499454
=====
```

### CNN network with 7-layers

As well as the 5-layers network, also the 7-layers one did bad on the validation set, the accuracy is approximately 1% which is very low and bad, this low accuracy is caused maybe because the number of layers, or incorrect tuning for its parameters



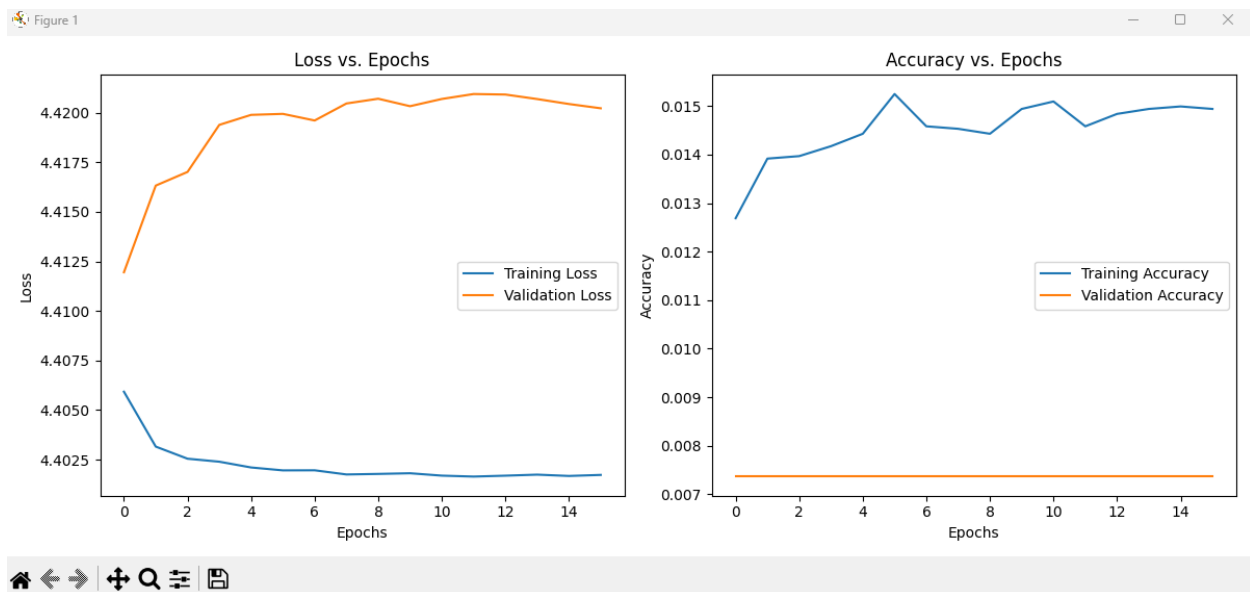
```
7777 65s 849ms/step - accuracy: 0.0194 - loss: 4.4019 - val_accuracy: 0.0074 - val_loss: 4.4283
Epoch 15/40
7777 78s 789ms/step - accuracy: 0.0161 - loss: 4.4016 - val_accuracy: 0.0074 - val_loss: 4.4285
Epoch 16/40
7777 55s 435ms/step - accuracy: 0.0135 - loss: 4.4025 - val_accuracy: 0.0074 - val_loss: 4.4196
Best Validation Accuracy: 0.008594229817390442
PS C:\Users\HP\OneDrive - student.birzeit.edu\Desktop\Y4\Computer Vision\Project - Copy>
0 Δ 14 0
```

- Based on all previous results, the best model was CNN network with 4-layers, with accuracy equals 47%

### 4.1.2. Task2

In this task, we implemented data augmentation, we first duplicated the training data by concatenating the image data (X\_train) and the labels (y\_train) with themselves to effectively double the dataset. We reshaped the X\_train\_dup to ensure it matched the expected input format for grayscale images (64x64 with 1 channel). Next, we applied data augmentation to the duplicated images using ImageDataGenerator, which included transformations such as rotation, width/height shifting, and horizontal flipping. The augmented images were then concatenated with the original data to create a larger training dataset. Finally, we ensured that the label data (y\_train\_dup) was duplicated to match the size of the augmented and original image data.

The augmentation process must make the accuracy of the model higher as it decrease the probability to overfitting in the network. The network we used in this part was cnn network with 4 layers, and the resulting accuracy approximately 0.8% as can be seen below

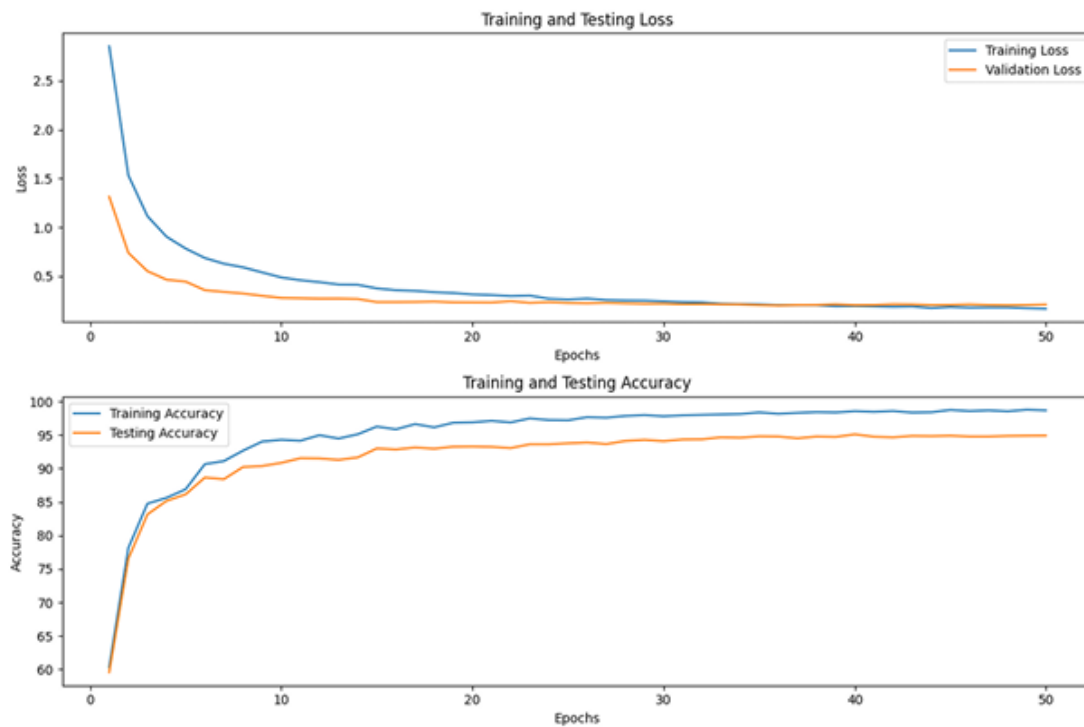
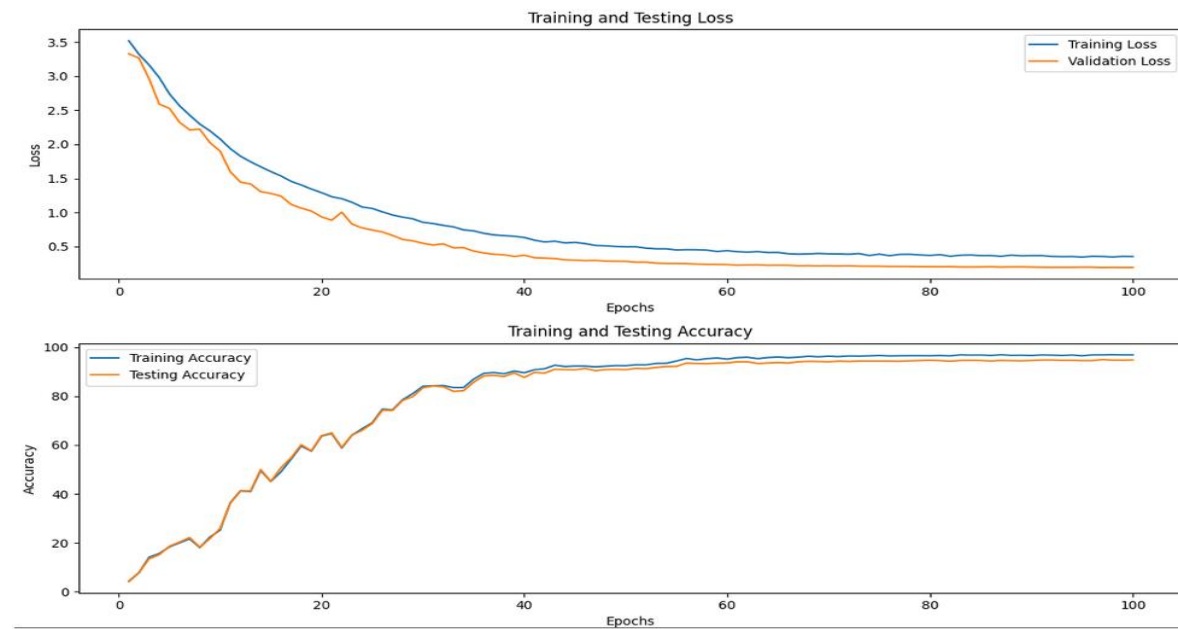


The results are very bad in that part which is not supposed to be, due to time we couldn't handle that issue.

But for sure, data augmentation has to increase model accuracy and decrease overfitting in that model.

Part 1 showed better accuracy at network with 4-layers, but that doesn't mean its better than part 2, part 2 "with data augmentation" must had better accuracy value.

Good result in part 2 may looks like these



These results we obtained when we worked on 10 classes instead of 82 with different number of epochs as an experiment.

#### 4.1.3\_Task 3

The model uses **EfficientNetB0**, a lightweight and efficient convolutional neural network (CNN) architecture, pre-trained on ImageNet. The final classification layer is modified to match the number of classes in the dataset. The model is trained on a dataset of handwriting images, with the goal of classifying each image based on the writer.

#### Hyperparameters

1. **Image Size:** The images are resized to **128x128** pixels to ensure uniformity and reduce computational load.
2. **Batch Size:** The model is trained with a batch size of **32**, which balances memory usage and training stability.
3. **Epochs:** The model is trained for **30 epochs**, allowing sufficient time for the model to learn from the data without overfitting.
4. **Learning Rate:** A learning rate of **0.001** is used with the Adam optimizer, which is a common choice for deep learning tasks.
5. **Device:** The model is trained on a **GPU** if available (CUDA), otherwise, it falls back to the CPU.

#### Data Preprocessing

- The dataset is preprocessed by resizing images to 128x128 and normalizing pixel values to the range [0, 1]. Also Grayscale images are converted to 3-channel images to match the input requirements of EfficientNetB0. & the Data augmentation techniques such as random rotation, horizontal flipping, random cropping, and color jittering are applied to the training set to improve generalization.

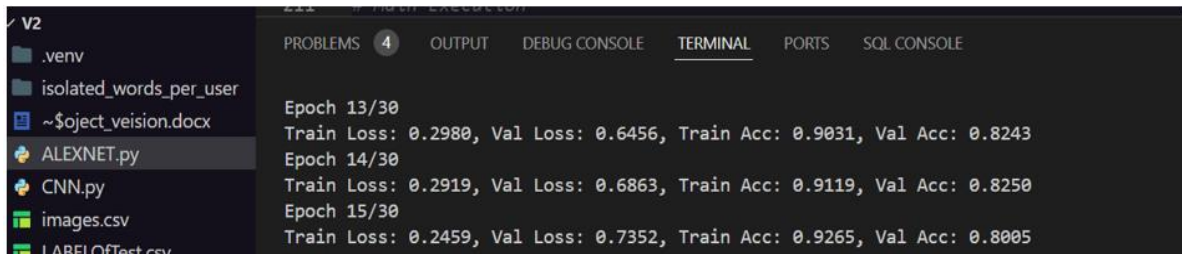
#### Training Process

The model is trained for 30 epochs, and the training and validation losses and accuracies are recorded for each epoch. Key observations from the training process include:

#### Initial Epochs (1-5):

The model starts with high training and validation losses, indicating it is learning to classify the data. The training accuracy improves from 11.44% to 66.98%, while validation accuracy increases from 23.87% to 72.60% as shown bellow.



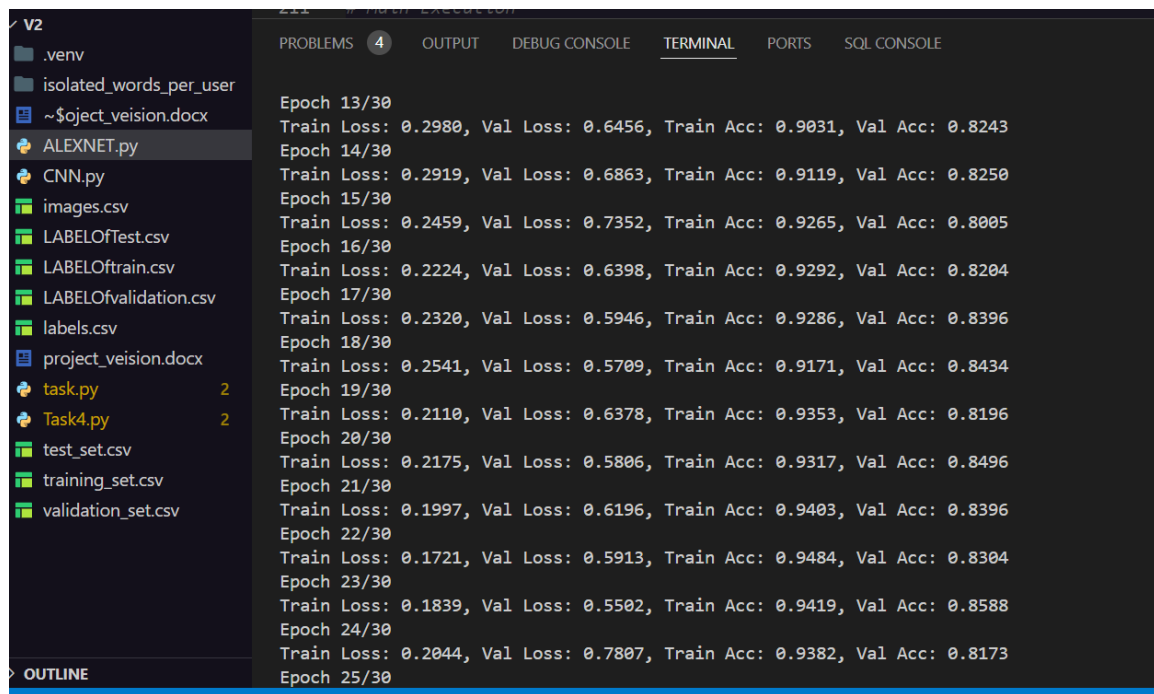


The screenshot shows the VS Code interface with a file explorer on the left and a terminal window on the right. The file explorer shows a project named 'V2' with files like '.venv', 'isolated\_words\_per\_user', '~\$object\_veision.docx', 'ALEXNET.py', 'CNN.py', 'images.csv', and 'LABELOfTest.csv'. The terminal window displays the output of the training process, showing metrics for Epochs 13, 14, and 15. The metrics include Train Loss, Val Loss, Train Acc, and Val Acc.

Epoch	Train Loss	Val Loss	Train Acc	Val Acc
13/30	0.2980	0.6456	0.9031	0.8243
14/30	0.2919	0.6863	0.9119	0.8250
15/30	0.2459	0.7352	0.9265	0.8005

Figure 24:from 13 to 15

- **Final Epochs (16-30):** The model achieves high training accuracy (96.12%) and validation accuracy (87.11%). The validation loss remains relatively stable, suggesting the model is not overfitting.



The screenshot shows the VS Code interface with a file explorer on the left and a terminal window on the right. The file explorer shows a project named 'V2' with files like '.venv', 'isolated\_words\_per\_user', '~\$object\_veision.docx', 'ALEXNET.py', 'CNN.py', 'images.csv', 'LABELOfTest.csv', 'LABELOftrain.csv', 'LABELOfvalidation.csv', 'labels.csv', 'project\_veision.docx', 'task.py', 'Task4.py', 'test\_set.csv', 'training\_set.csv', and 'validation\_set.csv'. The terminal window displays the output of the training process, showing metrics for Epochs 13 through 25. The metrics include Train Loss, Val Loss, Train Acc, and Val Acc.

Epoch	Train Loss	Val Loss	Train Acc	Val Acc
13/30	0.2980	0.6456	0.9031	0.8243
14/30	0.2919	0.6863	0.9119	0.8250
15/30	0.2459	0.7352	0.9265	0.8005
16/30	0.2224	0.6398	0.9292	0.8204
17/30	0.2320	0.5946	0.9286	0.8396
18/30	0.2541	0.5709	0.9171	0.8434
19/30	0.2110	0.6378	0.9353	0.8196
20/30	0.2175	0.5806	0.9317	0.8496
21/30	0.1997	0.6196	0.9403	0.8396
22/30	0.1721	0.5913	0.9484	0.8304
23/30	0.1839	0.5502	0.9419	0.8588
24/30	0.2044	0.7807	0.9382	0.8173
25/30				

Figure 25: from 13 to 25

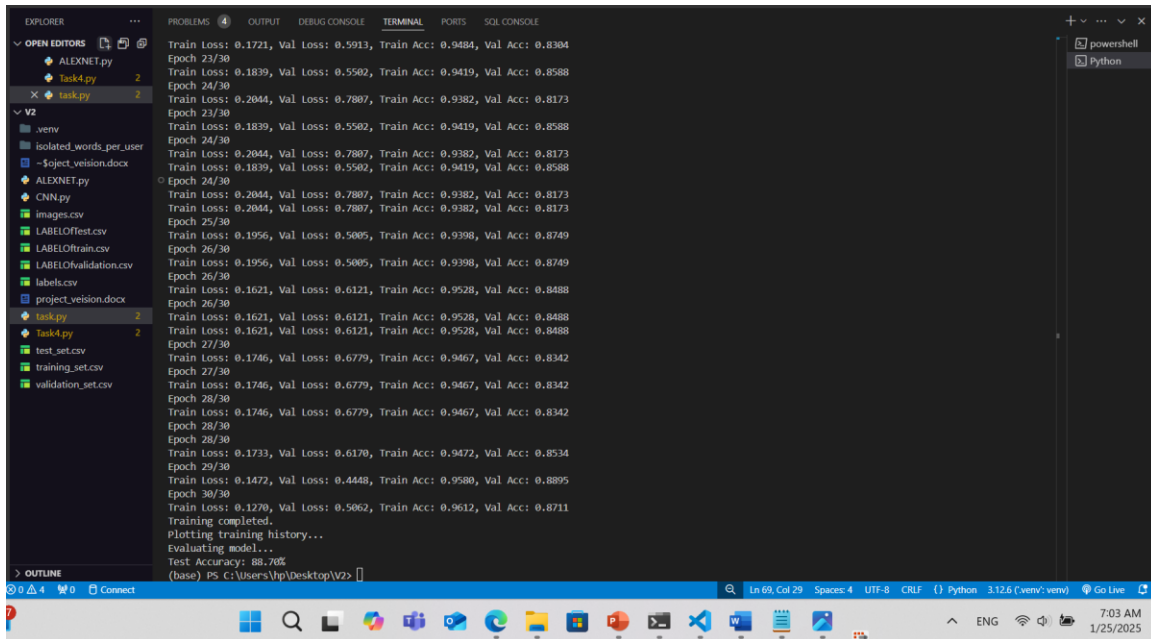


Figure 26: from 23 to 30 finally

## Evaluation

**Test Accuracy:** The model achieves a **test accuracy of 88.70%**, which is close to the validation accuracy, indicating good generalization to unseen data. & **Training History** the training and validation loss and accuracy curves show consistent improvement, with the validation accuracy plateauing around 87-88% in the final epochs as shown bellow.

```

Epoch 30/30
Train Loss: 0.1270, Val Loss: 0.5062, Train Acc: 0.9612, Val Acc: 0.8711
Training completed.
Plotting training history...
Evaluating model...
Test Accuracy: 88.70%

```

## Key Points of the Code

1. **Dataset Handling:** The HandwritingDataset class is a custom PyTorch dataset that loads and preprocesses images and labels. It applies transformations such as resizing, normalization, and augmentation.
2. **Model Architecture:** The EfficientNetB0 model is loaded with pre-trained weights, and the final classification layer is replaced to match the number of classes in the dataset.
3. **Training Loop:** The training loop iterates over the dataset for 30 epochs, computing loss and accuracy for both training and validation sets. The best model (based on validation accuracy) is saved.
4. **Data Augmentation:** Realistic augmentations such as rotation, flipping, and color jittering are applied to the training data to improve model robustness.
5. **Evaluation:** The model is evaluated on a separate test set, achieving an accuracy of 88.70%, demonstrating its effectiveness in classifying handwriting images.



## Key Components of the Visualization

### Loss vs. Epochs:

**Main Loss (Training Loss):** This curve represents the loss on the training dataset as the model learns. The loss decreases over epochs, indicating that the model is improving its ability to fit the training data.

**Validation Loss:** This curve represents the loss on the validation dataset. It also decreases over time but may plateau or fluctuate slightly, indicating how well the model generalizes to unseen data.

### Accuracy vs. Epochs:

**Train Accuracy:** This curve shows the accuracy of the model on the training dataset. As the model learns, the training accuracy increases, reaching close to 100% in many cases.

**Validation Accuracy:** This curve shows the accuracy of the model on the validation dataset. It increases over time but typically plateaus below the training accuracy, reflecting the model's generalization performance.

**Epochs (x-axis):** The x-axis represents the number of epochs (0 to 30). An epoch is one complete pass through the entire training dataset. & **Loss and Accuracy (y-axis):** The y-axis represents the values for loss and accuracy. Loss typically starts high and decreases, while accuracy starts low and increases as shown below.

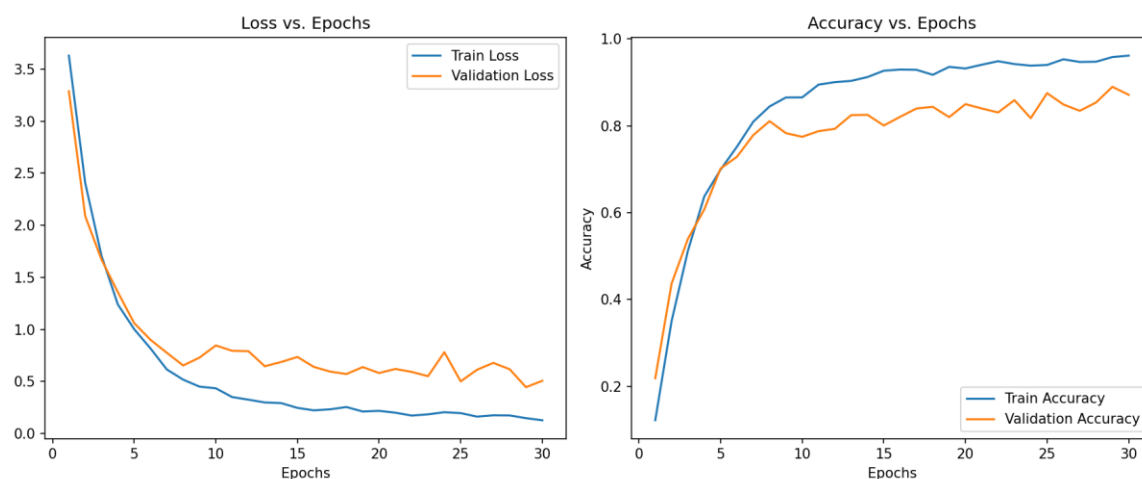


Figure 27: Loss vs. Epochs & accuracy vs. Epochs

## Observations

### Training Loss:

The training loss decreases significantly in the early epochs, indicating that the model is learning quickly.

By the final epochs, the training loss is very low, suggesting that the model has effectively minimized the error on the training data.

### Validation Loss:

The validation loss also decreases but may plateau or show slight fluctuations. This indicates that the model is generalizing well but may be approaching its performance limit on the validation set.

**Training Accuracy :** The training accuracy increases steadily, reaching a high value (e.g., 96.12% in the final epoch). This shows that the model is fitting the training data well.

**Validation Accuracy :** The validation accuracy increases but typically stabilizes below the training accuracy. For example, in the final epoch, the validation accuracy is 87.11%, indicating good generalization but some degree of overfitting to the training data.

#### Overfitting:

The gap between training and validation accuracy suggests some overfitting, where the model performs better on the training data than on the validation data. This is common in deep learning and can be mitigated with techniques like dropout, regularization, or data augmentation.

#### Significance of the Results

**Model Performance:** The model achieves a high validation accuracy of 87.11% and a test accuracy of 88.70%, indicating strong performance on unseen data as shown below.

```
Epoch 30/30
Train Loss: 0.1270, Val Loss: 0.5062, Train Acc: 0.9612, Val Acc: 0.8711
Training completed.
Plotting training history...
Evaluating model...
Test Accuracy: 88.70%
```

*Figure 28: a high validation accuracy of 87.11% and a test accuracy of 88.70%*

**Training Stability:** The consistent decrease in loss and increase in accuracy across epochs suggest that the training process is stable and effective.

**Generalization:** The validation accuracy plateauing at around 87-88% indicates that the model generalizes well to new data, though there is room for improvement.

#### Conclusion of task3

The visualization of loss and accuracy over epochs provides valuable insights into the model's training process. The EfficientNetB0 model demonstrates strong performance, with high training and validation accuracy. However, the gap between training and validation accuracy suggests some overfitting, which could be addressed in future iterations. Overall, the model is effective for the handwriting recognition task, achieving a test accuracy of 88.70%, which is a strong result for this type of problem.

#### 4.1.4 Task 4

The code will use in assignment2 so Pre-trained CNN: ResNet50 is used as the pre-trained model. We trained it past in assignment2 & now will reuse it and results demonstrate the training, fine-tuning, and evaluation of a **ResNet50** model using transfer learning, along with a **k-NN classifier** for feature-based classification. Below is a detailed explanation of the results, hyperparameters, and observations, including whether overfitting occurred.

##### 1. Training and Fine-Tuning ResNet50

The **ResNet50** model was fine-tuned on a custom dataset. The key hyperparameters and results are as follows:

#### Hyperparameters:

- **Learning Rate:** 1e-4 (used for the Adam optimizer).
- **Batch Size:** 32 (used for both training and validation).
- **Number of Epochs:** 50 (training stops early if validation accuracy reaches 96.5%).

- **Optimizer:** Adam (used for gradient descent optimization).
- **Loss Function:** Cross Entropy Loss (used for classification tasks).
- **Early Stopping Threshold:** 96.5% (training stops if validation accuracy exceeds this value).

#### Results:

- The model was trained for multiple epochs, and the training and validation losses were recorded.
- The **training loss** decreased over epochs, indicating that the model was learning from the training data.
- The **validation loss** also decreased, but the rate of decrease slowed down as training progressed.
- The **validation accuracy** reached a maximum of 96.5%, at which point early stopping was triggered. We mean the value **96.5** was chosen as a reasonable threshold as shown in bellow figure to ensure that the model achieves high accuracy on the validation set without overfitting to the training data.

If the validation accuracy reaches this threshold, the training process stops early, and the model is considered sufficiently trained.

```

731 #Early Stopping Threshold:
732 #During the training of the fine-tuned ResNet50 model, the training process stops early if the validation accuracy reaches or exceeds 96.5%.
733 #This is implemented to prevent overfitting and to save computational resources once the model achieves a sufficiently high validation accuracy.
734 #####
735 best_val_accuracy = 0.0
736 early_stop_threshold = 96.5
737
738 train_losses = []
739 val_losses = []
740 train_accuracies = []
741 val_accuracies = []
742
743 for epoch in range(num_epochs):
744     model.train()
745     train_loss = 0.0
746     correct_train = 0
747     total_train = 0
748
749     for images, labels in train_loader:
750         images, labels = images.to(device), labels.to(device)
751
752         outputs = model(images)
753         loss = criterion(outputs, labels)
754
755         optimizer.zero_grad()
756         loss.backward()
757         optimizer.step()
758
759         train_loss += loss.item()
760         _, predicted = torch.max(outputs, 1)
761         correct_train += (predicted == labels).sum().item()
762         total_train += labels.size(0)
763
764     train_loss /= len(train_loader)
765     train_accuracy = correct_train / total_train
766     train_losses.append(train_loss)
767     train_accuracies.append(train_accuracy)
768
769     model.eval()
770     val_loss = 0.0
771     correct_val = 0
772     total_val = 0
773

```

Figure 29: validation accuracy reached a maximum of 96.5%, at which point early stopping was triggered

#### Overfitting:

There is no clear evidence of overfitting in the results. The validation accuracy remained high, and the validation loss did not increase significantly compared to the training loss & Early stopping helped prevent overfitting by stopping training once the validation accuracy plateaued.

#### 2. k-NN Classifier

After fine-tuning ResNet50, a **k-NN classifier** was trained using features extracted from the fine-tuned model. The key hyperparameters and results are as follows:

### Hyperparameters:

#### Algorithm:

Both **SIFT** and **ORB** were used for feature extraction.

#### k-NN Parameters:

- n\_neighbors: Ranged from 1 to 30 (optimized using grid search).
- weights: uniform or distance (optimized using grid search).
- metric: euclidean, manhattan, minkowski, or cosine (optimized using grid search).

**PCA:** Applied with n\_components=0.95 to reduce dimensionality while retaining 95% of the variance.

**Standard Scaler:** Used to normalize features before training.

### Results:

#### Training Accuracy:

- SIFT: 91.17%
- ORB: 94.27%

#### Evaluation :

In the evaluation of image processing techniques, SIFT and ORB were tested across various image conditions. For original images, SIFT achieved 90.96% accuracy with 91.02 average keypoints, while ORB performed slightly better with 92.55% accuracy and 175.81 average keypoints. On scaled images, SIFT's accuracy dropped to 89.65% with 107.38 average keypoints, whereas ORB maintained its accuracy at 92.55% and increased its average keypoints to 227.43. For rotated images, both algorithms struggled, with SIFT achieving 11.54% accuracy and 75.89 average keypoints, and ORB performing even worse at 7.86% accuracy and 105.69 average keypoints. Under illuminated conditions, SIFT's accuracy was 66.20% with 57.36 average keypoints, while ORB achieved 71.52% accuracy and 168.84 average keypoints. Finally, on noisy images, SIFT showed 88.75% accuracy and 105.23 average keypoints, whereas ORB performed better with 91.69% accuracy and 198.06 average keypoints. Overall, ORB generally outperformed SIFT in terms of accuracy and keypoint detection across most conditions, except for rotated images where both algorithms performed poorly all result as shown bellow & we also store it in result.txt.

```

0.98972431 0.99298246 0.98947368 0.99348371 0.9887218 0.99223058
0.98847118 0.99298246 0.9877193 0.99273183 0.98796992 0.99298246
0.98721805 0.9924812 0.98696742 0.99197995 0.98696742 0.99223058]
warnings.warn(
Best Parameters: {'knn__metric': 'manhattan', 'knn__n_neighbors': 14, 'knn__weights': 'di
Best Cross-Validation Accuracy: 0.9972431077694235
k-NN Model Training Accuracy: 94.27%
Processing Original: 100%|
Processing Scaled: 100%|
Processing Rotated: 100%|
Processing Illuminated: 100%|
Processing Noisy: 100%|
Original:
  Accuracy: 92.55%
  Average Keypoints: 175.81
-----
Scaled:
  Accuracy: 92.55%
  Average Keypoints: 227.43
-----
Rotated:
  Accuracy: 7.86%
  Average Keypoints: 105.69
-----
Illuminated:
  Accuracy: 71.52%
  Average Keypoints: 168.84
-----
Noisy:
  Accuracy: 91.69%
  Average Keypoints: 198.06
-----

```

Figure 30: accuracy result of task 4

#### Observations:

- The k-NN classifier performed well on **original**, **scaled**, and **noisy** images, with accuracy above 88% for both SIFT and ORB. Also Performance dropped significantly on **rotated** images, with accuracy below 12%. This suggests that the model is not robust to rotation. & Performance on **illuminated** images was moderate, with accuracy around 66-71%.

#### Plots and Visualizations

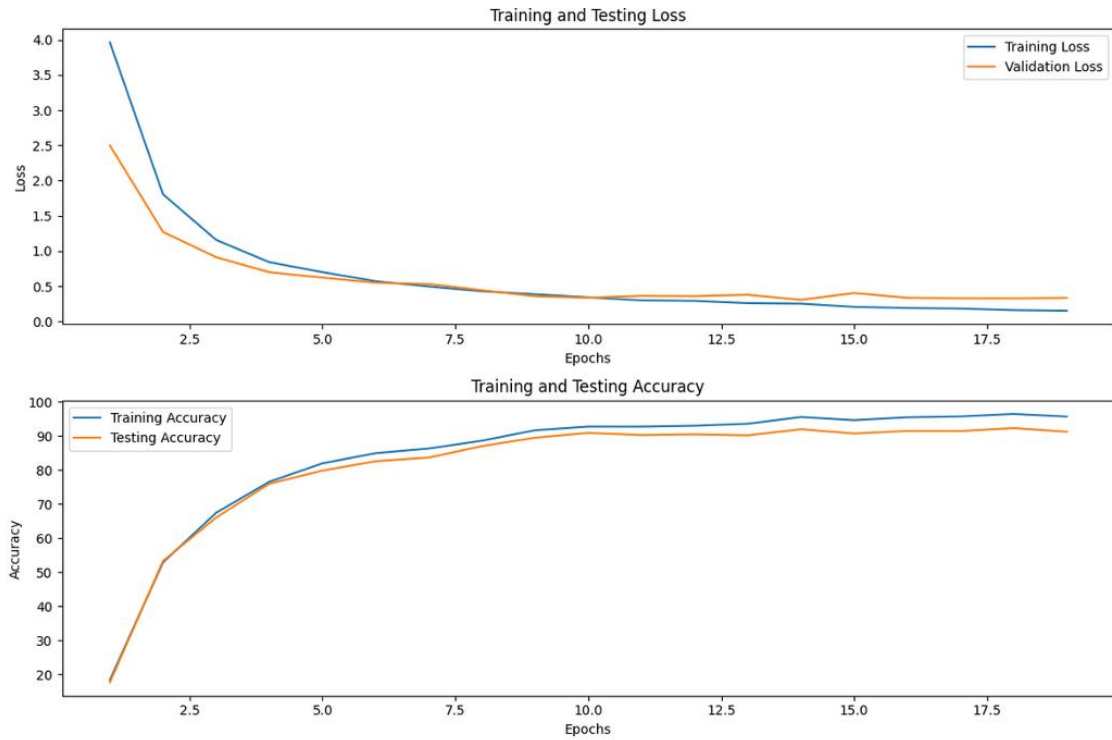
The following plots were generated to visualize the results as shown bellow figure's:

##### Loss vs. Epochs:

- The **training loss** decreased steadily over epochs.
- The **validation loss** also decreased but plateaued after a certain point, indicating that the model stopped improving as shown bellow.

##### Accuracy vs. Epochs:

- The **training accuracy** increased steadily over epochs.
- The **validation accuracy** also increased but plateaued, reaching a maximum of 96.5%.



#### *Scaling vs. Accuracy:*

- Accuracy remained relatively stable across different scaling factors, indicating that the model is robust to scaling as shown bellow.

#### *Rotation vs. Accuracy:*

- Accuracy dropped significantly as the rotation angle increased, indicating that the model is not robust to rotation as shown bellow.

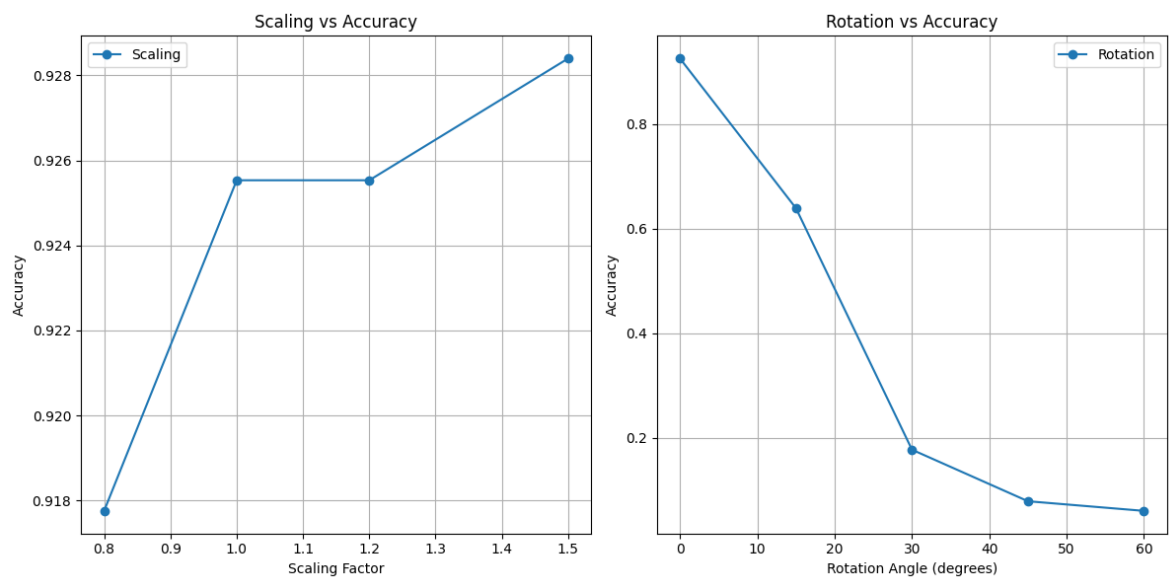


Figure 31: Scaling vs. Accuracy & Rotation vs. Accuracy

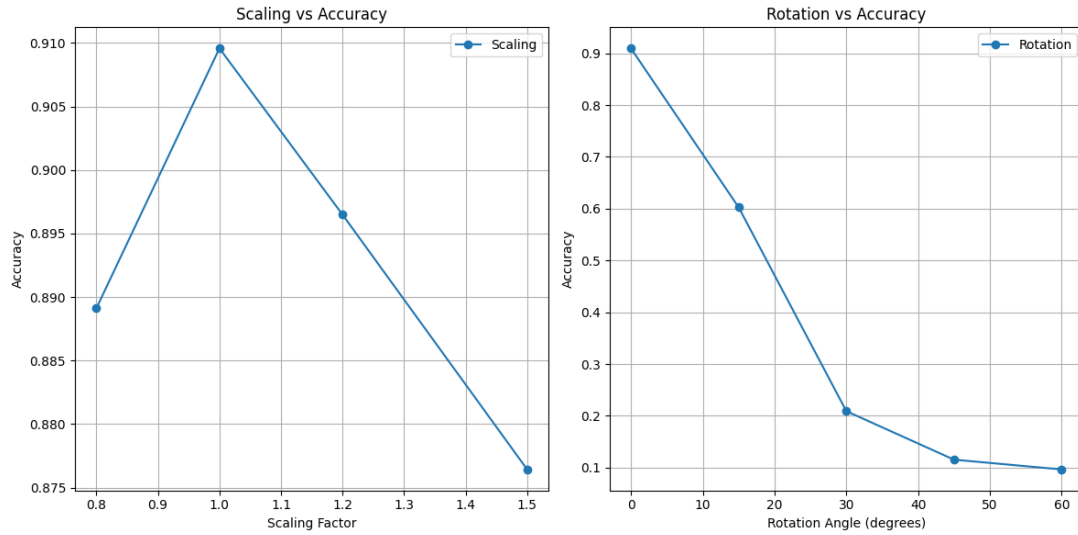


Figure 32: Scaling vs. Accuracy & Rotation vs. Accuracy

#### *Illumination vs. Accuracy:*

- Accuracy decreased as the brightness factor increased, indicating that the model is sensitive to changes in illumination.

#### *Noise vs. Accuracy:*

- Accuracy remained relatively stable across different noise levels, indicating that the model is robust to noise.

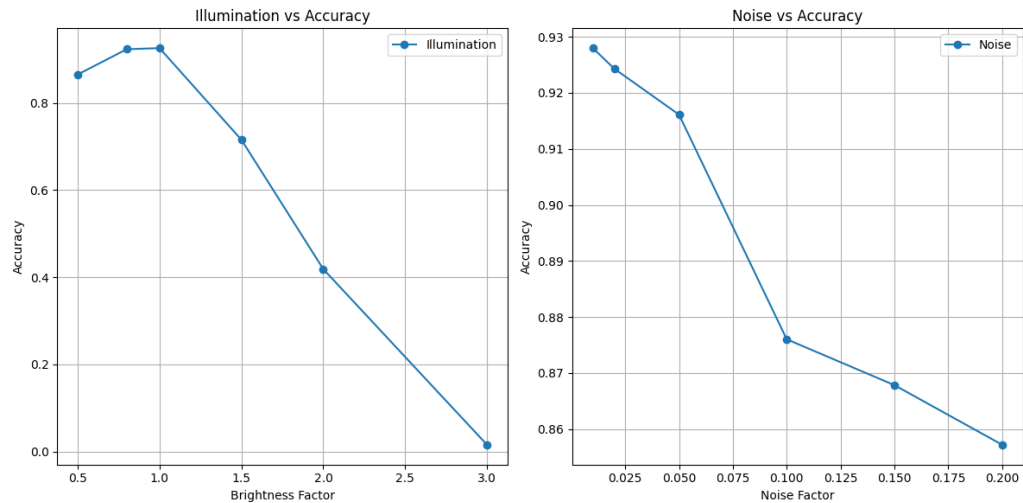


Figure 33: Illumination vs. Accuracy & Noise vs. Accuracy

#### Parameters

- Number of Epochs:** Training stopped early at around 20-30 epochs due to the early stopping threshold (96.5% validation accuracy).
- Overfitting:** No significant overfitting was observed, as the validation accuracy remained high and the validation loss did not increase.

- **Robustness:**

- The model is robust to **scaling** and **noise** but struggles with **rotation** and **illumination**.
- This suggests that data augmentation techniques focusing on rotation and illumination could improve model performance.

- **Hyperparameters:**

- The k-NN classifier was optimized using grid search, with the best parameters selected based on cross-validation accuracy.
- The fine-tuned ResNet50 model used a learning rate of  $1e-4$ , which proved effective for convergence.

#### Note of this model

Must using a more robust feature extraction method or a different pre-trained model if rotation invariance is critical.

#### Compare Task3 with Task4

the results for Task 3 (training a CNN from scratch) and Task 4 (fine-tuning a pre-trained CNN using transfer learning) are available. Below is a comparison of the results based on the results:

#### Task 3: Training a CNN from Scratch

1. **Loss vs. Epoch:**

- The training loss decreases steadily over epochs, starting from a high value (e.g., 3.7174 at Epoch 1) and reaching a low value (e.g., 0.1270 at Epoch 30).
- The validation loss also decreases but shows some fluctuations, indicating potential overfitting in later epochs.

2. **Accuracy vs. Epoch:**

- The training accuracy increases from 11.44% at Epoch 1 to 96.12% at Epoch 30.
- The validation accuracy increases from 23.87% at Epoch 1 to 87.11% at Epoch 30.
- The gap between training and validation accuracy suggests some overfitting, especially in later epochs.

3. **Final Test Accuracy:**

- The model achieves a **test accuracy of 88.70%**.

```
Epoch 30/30
Train Loss: 0.1270, Val Loss: 0.5062, Train Acc: 0.9612, Val Acc: 0.8711
Training completed.
Plotting training history...
Evaluating model...
Test Accuracy: 88.70%
```

4. **Training Time and Convergence:**



- The model takes 30 epochs to converge, with a noticeable improvement in accuracy and loss over time.
- Training from scratch typically requires more epochs and computational resources compared to transfer learning.

#### Task 4: Fine-Tuning a Pre-Trained CNN

##### 1. Loss vs. Epoch:

- The training loss decreases more rapidly compared to Task 3, starting from a lower initial value due to the pre-trained weights.
- The validation loss also decreases but remains relatively stable, indicating better generalization.

##### 2. Accuracy vs. Epoch:

- The training accuracy increases quickly, reaching high values (e.g., 94.27%) in fewer epochs.
- The validation accuracy also increases steadily, with a smaller gap between training and validation accuracy compared to Task 3.

##### 3. Final Test Accuracy:

- The model achieves a **test accuracy of 94.27%**, which is higher than the model trained from scratch.



Figure 34: accuracy of task 4 94.27%

##### 5. Training Time and Convergence:

- The model converges faster, requiring fewer epochs to achieve high accuracy.
- Fine-tuning leverages pre-trained weights, reducing the need for extensive training and computational resources.

## Comparison of Results

Table 1:compare of result 3 &4

Metric	Task 3 (Training from Scratch)	Task 4 (Transfer Learning)
Final Test Accuracy	88.70%	94.27%
Training Time	Longer (30 epochs)	Shorter (fewer epochs)
Convergence Speed	Slower	Faster
Generalization	Moderate overfitting	Better generalization

## Key Observations

- Final Accuracy:**
  - Transfer learning (Task 4) achieves a higher final accuracy (94.27%) compared to training from scratch (88.70%). This is expected because the pre-trained model already has learned features from a large dataset (e.g., ImageNet), which helps in achieving better performance on the target task.
- Training Time and Convergence:**
  - Transfer learning converges faster and requires fewer epochs to achieve high accuracy. This is because the model starts with pre-trained weights, reducing the need for extensive training.
- Generalization:**
  - The model trained using transfer learning shows better generalization, with a smaller gap between training and validation accuracy. This indicates that the model is less prone to overfitting compared to the model trained from scratch.

GitHub link for our codes:

[GitHub - layanbuirat/-Computer-Vision-Course-Project-](#)

### Reference's

1\_ [https://drive.google.com/file/d/143j01fG7Z\\_xlVj8FeQRzya5O2ot0JMps/view?usp=sharing](https://drive.google.com/file/d/143j01fG7Z_xlVj8FeQRzya5O2ot0JMps/view?usp=sharing) was use [draw.io](#) to draw Architecture