Faculty of Engineering & Technology

Electrical & Computer Engineering Department

COMPUTER VISION– ENCS5343

Project Report

Prepared by:

Name: Mohammad Abu Shams **ID:** 1200549

Name: Joud Hijaz **ID:** 1200342

Instructor: Dr. Aziz Qaroush

Section: 2

Date: 25-1-2025

BIRZEIT

Table of Contents

List of Figures

List of Tables

1. Introduction

This project studied how deep learning can be used to classify Arabic handwriting. This is a hard problem because handwriting styles are very different and complex. The dataset had grayscale images of handwritten text from many users, where each user was one class. The goal was to build and test deep learning models for good and fast classification.

The project was split into four tasks. Each task helped make the model better and tested different methods. Task 1 built a simple Convolutional Neural Network (CNN) from scratch as a starting point. In Task 2, data augmentation was used to add more variety to the dataset and improve the model's performance. Task 3 trained `AlexNet`, a better CNN model, to compare it with the custom CNN. Task 4 used transfer learning with a pre-trained `ResNet50` model to bring knowledge from big datasets and improve the results even more.

This step-by-step method showed how architecture design, data augmentation, and transfer learning changed the accuracy of the models. The report explained the setup, results, and performance of each task, showing how the models improved over time.

2. Experimental Setup and Results

2.1 Task1: Build and Train a Custom CNN Network

2.1.1 Import Libraries

The libraries were imported for important tasks to build and train a custom CNN network. The `zipfile` module was used to open the dataset if it was compressed. The `os` and `glob` libraries were used to work with files and folders, like finding and opening image files. The `random` and `numpy` libraries were used for random tasks and math calculations. Images were shown by `matplotlib.pyplot`, and `cv2` from `OpenCV` was used to process images, like changing size and making them grayscale. The `h5py` library was used to save and load model weights in HDF5 format. `PyTorch`, a deep learning library, was used a lot. The `torch` library was used for tensor work, `torch.nn` was used for making neural networks, and `torch.optim` was used for optimization. Functions from `torch.nn.functional` were used for activation and loss calculations. The `torch.utils.data` library was used for making datasets, splitting data, and creating data loaders for training and validation.

2.1.2 Custom Dataset

The `ArabicHandwritingDataset` class was created as a custom version of `PyTorch's Dataset` module. It was made to load and prepare Arabic handwriting data for training deep learning models. When the class was started, it was given the main folder of the dataset (`root_dir`), optional transformations (`transform`), and the wanted size of the images (`width` and `height`). It was expected that each folder inside the main folder represented one user (or class). Each user was given a unique label using a dictionary (`user_to_label`). The class went through every user's folder, collected all image file paths, and matched them with their labels. This way, the dataset was organized into two lists: `image_paths` for images and `labels` for their classes.

The `__getitem__` method was used to load and prepare one image at a time. For a given index (`idx`), the method found the image path and its label. The image was opened in grayscale using `OpenCV`, resized to 256x128 pixels, and changed into a `PyTorch` tensor with the shape `[1, H, W]`. The pixel values were divided by 255 to make them between 0 and 1. If extra transformations (like data augmentation) were given, they were applied here. In the end, the processed image tensor and its label were returned. This class was designed to work with `PyTorch's DataLoader`, making it easy to process data in batches and train models faster. It was made flexible, scalable, and ready for deep learning workflows.

2.1.3 Utility to Get DataLoaders

The `get_data_loaders` function was made to help load and split the dataset into training and validation sets. It was prepared for easy batch processing using `PyTorch's DataLoader`. The function was given some inputs, like the dataset folder (`data_dir`), batch size (`batch_size`), validation set ratio (`val_ratio`), and image size (`width, height`). First, the `ArabicHandwritingDataset` class was used to load the dataset and find the total number of classes by checking the unique subfolders. Then, the dataset was split into training and

validation sets with the `random_split` function. A fixed random seed was used to make sure the split stayed the same every time. Finally, the training and validation sets were wrapped into `DataLoader` objects. This allowed the data to be processed in batches, shuffled, and loaded during training and validation. This function made it easier to handle datasets and prepared the data for deep learning tasks.

2.1.4 CNN Architectures with Adaptive Pooling

The `SimpleCNN` was designed as a small and fast model for easy classification tasks. It was made with two convolutional layers (`conv1` and `conv2`) to take features from input images. Each convolutional layer used 3x3 filters with padding to keep the image size the same. The number of features increased from 16 to 32 in the second layer to learn more complex patterns. After the convolutional layers, an adaptive max-pooling layer was used to make the image size smaller. It always made the size 4x4, no matter the input size, so different image sizes could be handled. The feature maps were flattened and sent to two fully connected layers. The first layer changed the size to 128 neurons, and the second layer gave the final predictions for 82 classes. A dropout layer with 50% chance was added before the last fully connected layer. This was done to stop the model from overfitting. This architecture was made simple and worked well for taking features and classifying them in tasks where speed and less computation were important.

The `DeeperCNN` was designed as a more advanced model for hard tasks that needed high accuracy. It was made with three convolutional layers (`conv1`, `conv2`, and `conv3`). The number of filters became bigger in each layer: 32, 64, and 128. These deeper layers were used to take more detailed features from the input image. After each convolutional layer, a max-pooling layer was added. These pooling layers made the image smaller and focused on the most important features. Then, an adaptive average pooling layer was used to make the size 4x4. This step helped to connect with fully connected layers and allowed different input sizes. The feature maps were flattened and sent to two fully connected layers. The first layer expanded the size to 256 neurons, and the second layer gave the final class predictions. A dropout layer with 20% probability was used to stop overfitting. This model had more layers and pooling steps, so it was stronger and better for hard datasets. It gave better feature learning and generalization than the simple models.

2.1.5 Training and Evaluation

The `train_model` function was created to train and check a neural network model over many epochs. It was made flexible, so different optimizers like Adam, SGD, or RMSprop could be chosen. The learning rate and weight decay for regularization were also adjusted. In training, data batches were taken from `train_loader`. Predictions were calculated by forward propagation, and the loss was found using the `CrossEntropyLoss` function. Backpropagation was used to update the model's weights. During training, the total loss and accuracy for each epoch were recorded. After training, the model was checked on validation data from `val_loader`. This step was done in a separate loop with gradients turned off to make it faster. It helped to see how well the model worked on new data and to find problems like overfitting or underfitting.

The losses and accuracies for training and validation were saved for every epoch. The best validation accuracy was tracked. If better accuracy was found, the model's weights were saved, so the best version of the model could be restored after training. The `plot_curves` function was used to show the training and validation loss and accuracy trends in side-by-side graphs. These graphs helped to understand how the model learned. Problems like overfitting (when validation loss went up, but training loss went down) or underfitting (when both losses stayed high) were seen easily. Together, these functions gave an organized way to train, check, and understand how a neural network worked.

2.1.6 Save the Model

The `save_model_h5` function was used to save the trained model's weights in HDF5 file format. This format was good for keeping and sharing big numerical data. The function took the model's state dictionary (`state_dict`), which had all the trainable parameters like weights and biases. The `h5py` library was used to make an HDF5 file with the given file name. Inside the file, each layer's name and its weights were saved as separate datasets. By saving the weights this way, the model's parameters could be loaded later. This was helpful for more training, testing, or using the model without training it again. This function made sure the model's state was kept in a small and easy-to-share file.

2.1.7 Main Execution

The main execution block was used to manage the whole training and testing process for `SimpleCNN` and `DeeperCNN` models. A fixed seed value of 42 was set for Python's random module, `NumPy`, and `PyTorch`. This was done to make sure the results stayed the same every time the script was run. Things like dataset shuffling, weight initialization, and data augmentation gave the same outputs, so experiments could be compared easily. The script also checked the computing device, like CPU or GPU, and printed the GPU name if it was available. Using a GPU was done to make training faster, especially for deep learning models.

The main script tested different hyperparameter settings to see how they changed the performance of `SimpleCNN` and `DeeperCNN`. Three hyperparameter sets were used:

- A learning rate (lr) of 0.001, batch size of 32, 30 epochs, and the Adam optimizer with no weight decay (0.0).
- A smaller learning rate of 0.0005, batch size of 32, 30 epochs, and the Adam optimizer with small weight decay (1e-4) for regularization.
- A bigger learning rate of 0.01, batch size of 32, 30 epochs, and the SGD optimizer with no weight decay.

These configurations were used to test how different learning rates, optimizers, and regularization methods changed the model's performance. For each setting, the training and validation loss and accuracy were recorded and shown in graphs to see the learning process. The best model, found by the highest validation accuracy, was saved. This was done to keep the most effective setup. This careful testing gave a full check of the models and helped to find the best setup for the task.

The training and validation curves for `SimpleCNN` in Experiment 1, with hyperparameters `{'lr': 0.001, 'batch_size': 32, 'num_epochs': 30, 'optimizer_name': 'Adam', 'weight_decay': 0.0}`, showed steady performance improvement over 30 epochs. Training loss was decreased to 2.2197, and training accuracy reached 35.21%. This showed the model learned the dataset well. At the same time, validation loss was reduced to 2.0664, and validation accuracy improved to 43.06%. This showed the model worked good on new data. The Adam optimizer with a learning rate of 0.001 helped with stable and fast learning. No weight decay was used, so the model was not over-regularized. The training and validation metrics stayed close, showing a balanced training with little overfitting.

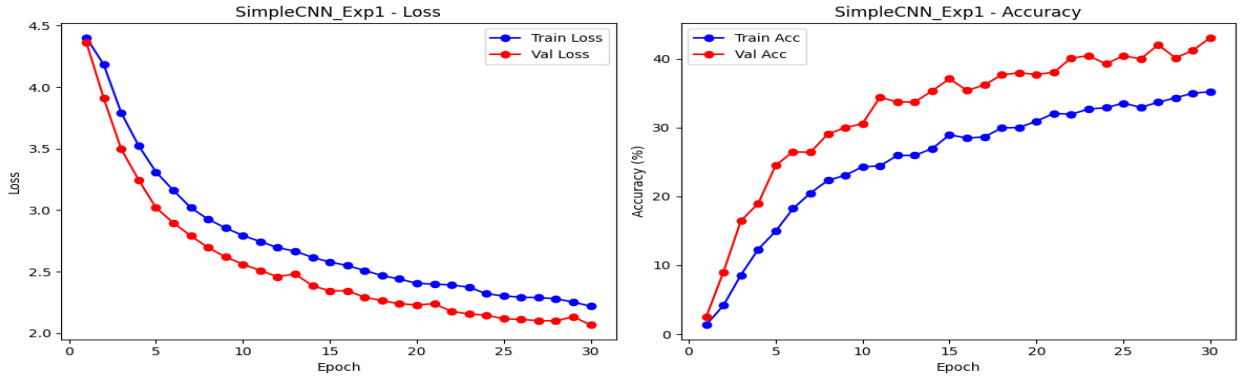

Figure 1: Training and Validation Curves for SimpleCNN Experiment1

The training and validation performance for SimpleCNN in Experiment 2, with hyperparameters `{'lr': 0.0005, 'batch_size': 32, 'num_epochs': 30, 'optimizer_name': 'Adam', 'weight_decay': 0.0001}`, showed steady improvement over 30 epochs. Training loss was reduced to 2.2849, and training accuracy reached 34.42%. Validation loss decreased to 2.0848, and validation accuracy improved to 40.91%. These results showed the model learned well and worked good on new data. Training and validation curves stayed close, which means the model did not overfit. The lower learning rate and small weight decay were used to make updates stable and helped the model generalize better for unseen data. These metrics showed that the experiment balanced learning and generalization well, but the convergence was slower than Experiment 1 because the learning rate was reduced.

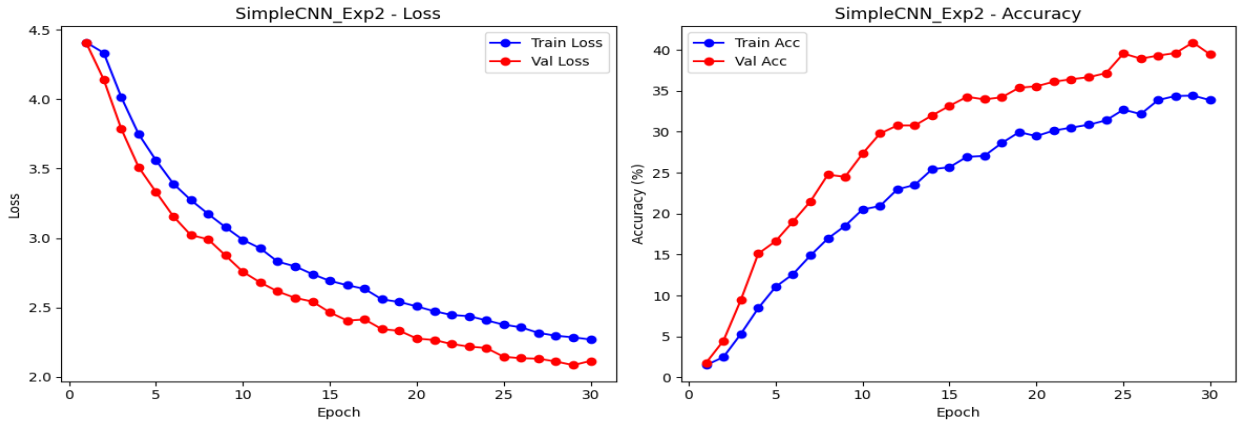

Figure 2: Training and Validation Curves for SimpleCNN Experiment2

The training and validation results for `SimpleCNN` Experiment 3, with hyperparameters `{'lr': 0.01, 'batch_size': 32, 'num_epochs': 30, 'optimizer_name': 'SGD', 'weight_decay': 0.0}`, showed big improvement over 30 epochs. Training loss was reduced to 1.8461, and accuracy reached 45.24%. Validation loss dropped to 1.6425, and validation accuracy improved to 53.13%. The high learning rate of 0.01 made faster convergence, seen in the steep drop of loss curves in early epochs. The model generalized well to validation data, shown by close match between training and validation curves. This setup had the best validation performance for the `SimpleCNN`, showing the effectiveness of SGD optimizer and chosen learning rate for this dataset and architecture.

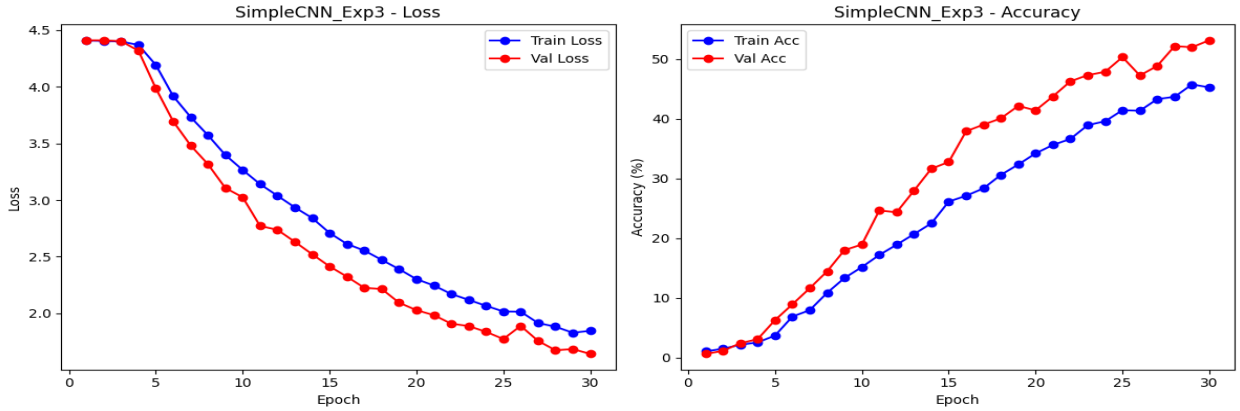

Figure 3: Training and Validation Curves for SimpleCNN Experiment3

The training and validation results for `DeeperCNN` Experiment 1, with hyperparameters `{'lr': 0.001, 'batch_size': 32, 'num_epochs': 30, 'optimizer_name': 'Adam', 'weight_decay': 0.0}`, showed big improvement after 30 epochs. Training loss was reduced to 1.0563, and accuracy reached 66.41%. Validation loss dropped to 1.4310, and accuracy improved to 60.32%. The learning rate 0.001 helped stable training, seen by the loss curves going down every epoch. The model worked well on validation data because training and validation metrics stayed close. This setup gave the best results in all previous experiments, showing Adam optimizer and these hyperparameters worked best for `DeeperCNN`.

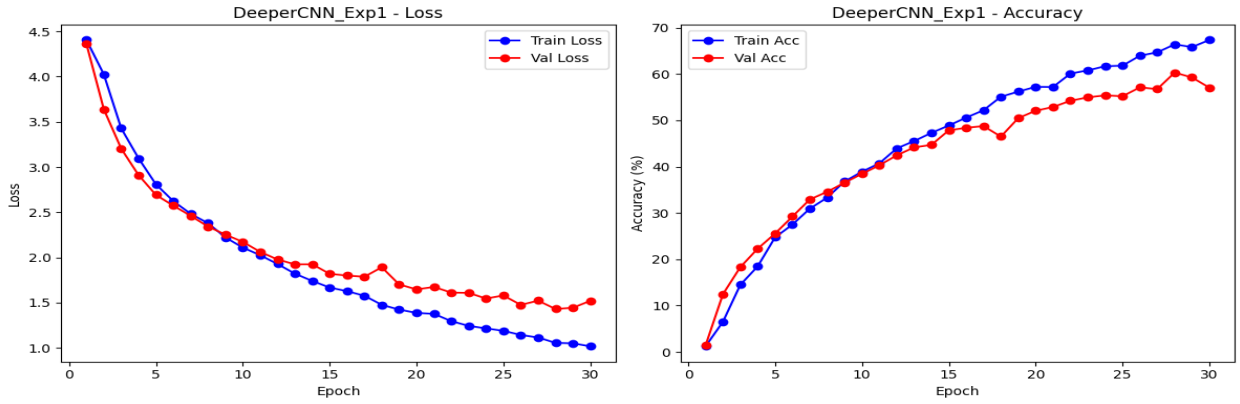

Figure 4: Training and Validation Curves for DeeperCNN Experiment1

The training and validation results for `DeeperCNN` Experiment 2, with hyperparameters `{'lr': 0.0005, 'batch_size': 32, 'num_epochs': 30, 'optimizer_name': 'Adam', 'weight_decay': 0.0001}`, showed steady improvements over 30 epochs. Training loss was reduced to 1.7514, with an accuracy of 47.47%. Validation loss dropped to 1.9931, and validation accuracy reached 44.72%. The lower learning rate of 0.0005 and small weight decay of 0.0001 helped make the training more stable, seen in the smooth loss curves and slow accuracy increases. The results were good but showed slightly lower performance than the best experiment, showing how hyperparameters affected the model's learning and generalizing.

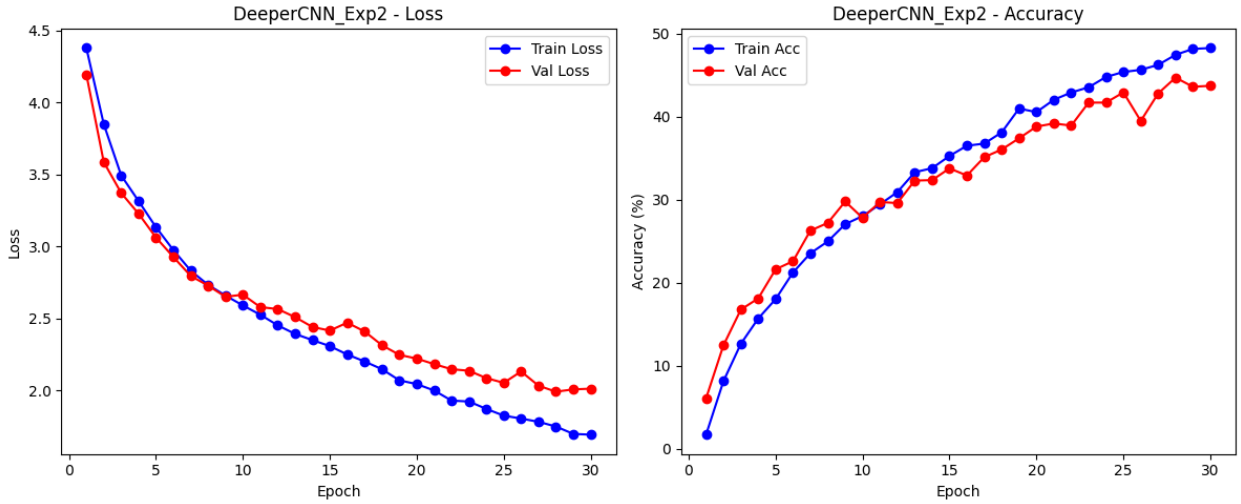

Figure 5: Training and Validation Curves for DeeperCNN Experiment2

The training and validation results for `DeeperCNN` Experiment 3, with hyperparameters `{'lr': 0.01, 'batch_size': 32, 'num_epochs': 30, 'optimizer_name': 'SGD', 'weight_decay': 0.0}`, showed big improvements in both loss and accuracy over 30 epochs. The training loss was reduced to 1.8359, with an accuracy of 46.78%. The validation loss reached 2.0115, and the validation accuracy was 45.64%. The high learning rate of 0.01 helped the model learn faster, seen in the sharp drop in loss during the first epochs. Even with the quick progress, the validation metrics showed a small gap compared to training metrics, suggesting that the learning rate or regularization could be improved for better generalization. This experiment showed that SGD with a higher learning rate helped speed up the learning process for this model.

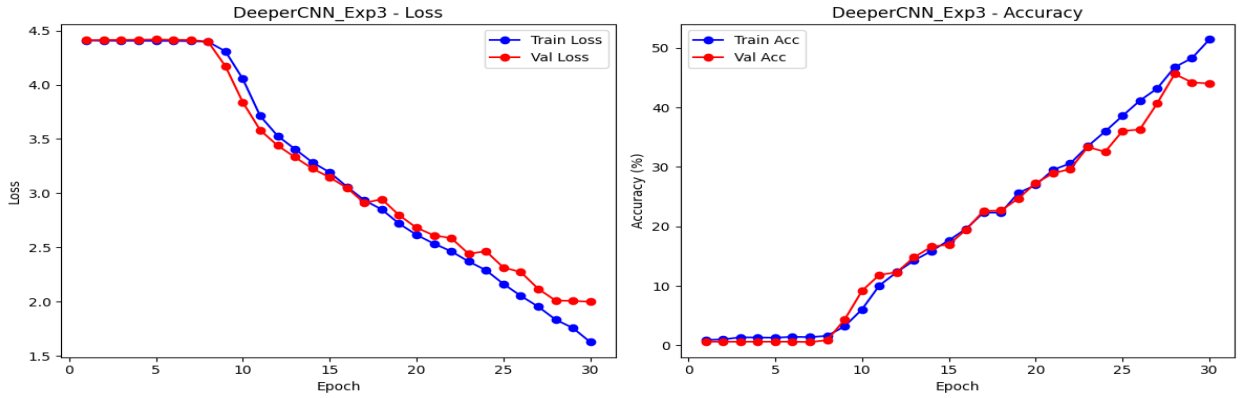

Figure 6: Training and Validation Curves for DeeperCNN Experiment3

Table 1 below showed a full comparison of training and validation metrics for all experiments with `SimpleCNN` and `DeeperCNN` models. DeeperCNN_Exp1 was the best model for all simple and deep models, with the lowest validation loss (1.4310) and the highest validation accuracy (60.32%). This success was because of the combination of the Adam optimizer, a moderate learning rate of 0.001, and no weight decay, which helped the model learn well and work efficiently. The best performance of DeeperCNN_Exp1 showed that a deeper architecture with good hyperparameters worked well for this dataset.

For `SimpleCNN` experiments, SimpleCNN_Exp3 did the best, with a validation loss of 1.6425 and a validation accuracy of 53.13%. This showed that the SGD optimizer with a high learning rate of 0.01 helped the model learn faster and generalize well. Even though `DeeperCNN` did better than `SimpleCNN` in all tests, the results showed how important it is to choose the right hyperparameters for each model. Overall, the experiments showed that deeper architectures like `DeeperCNN` worked better and that careful hyperparameter choices are needed for the best results.

Table 1: Comparison of Training and Validation Metrics Across Experiments

	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
SimpleCNN_Exp1	2.2197	35.21%	2.0664	43.06%
SimpleCNN_Exp2	2.2849	34.42%	2.0848	40.91%
SimpleCNN_Exp3	1.8461	45.24%	1.6425	53.13%
DeeperCNN_Exp1	1.0563	66.41%	1.4310	60.32%
DeeperCNN_Exp2	1.7514	47.47%	1.9931	44.72%
DeeperCNN_Exp3	1.8359	46.78%	2.0115	45.64%

2.2 Task2: Retrain the Network Selected from Task 1 after Doing Data Augmentation

2.2.1 Data Augmentation Functions

The data augmentation functions were used to make the dataset bigger by adding random changes to the images. This helped make the dataset more diverse and improved how well the model learned during training. The function `apply_random_augmentations` added two types of changes to grayscale images: a random flip of the image from left to right (50% chance) and a random rotation between -15 and +15 degrees. These changes were made using the Python Imaging Library (`PIL`). The images stayed in grayscale after the changes. By adding these random changes, new versions of the same image were created, making the model better at handling different distortions in real-world data.

The function `augment_and_save_dataset` was made to automate the process of adding changes to all images in a dataset. The dataset was stored in folders for different categories. For each image, the original was copied to a new folder, and a set number of new images were made using `apply_random_augmentations`. The folder structure was kept the same, and a random seed was used so the process could be repeated with the same results. This method kept the original dataset safe and made it larger by adding more images with changes. By creating realistic differences in the data, this technique helped the model perform better on new, unseen images during testing.

2.2.2 Create Augmented Dataset

The dataset was made bigger by creating two new versions of every original image from the source folder (`isolated_words_per_user`). These new images, with the original ones, were saved in a new folder (`isolated_words_per_user_aug`). This made the dataset larger and more diverse, helping the model train better.

2.2.3 Custom Dataset (Pointing to the Augmented Folder)

The `ArabicHandwritingDataset` class and `get_data_loaders` function were made to work with an Arabic handwriting dataset that was augmented. The dataset was saved in a folder, and each subfolder showed one user, with their images inside. The `get_data_loaders` function split the dataset into training and validation parts, using a validation ratio (default was 20%). A fixed random seed was used so the split stayed same every time. The function gave `DataLoader` objects to make batching easy for training and validation. It also gave the total number of user classes, helping in training deep learning models efficiently with the augmented dataset.

2.2.4 Define DeeperCNN Architecture

The `DeeperCNN` architecture was the same as task1.

2.2.5 Training, Plotting, and Saving Utilities

The training, plotting, and saving utilities were the same as in Task 1. These included:

- `train_model`: Used to train the model and check its performance over epochs.
- `plot_curves`: Created line plots to show training and validation losses and accuracies.
- `save_model_h5`: Saved the trained model's parameters in an .h5 file for later use.

2.2.6 Main Execution

The main code for Task 2 trained the `DeeperCNN` model from start on the augmented dataset using best settings from Task 1. First, a random seed was set for reproducibility, and the device (CPU or GPU) was selected for training. The augmented dataset was loaded with training-validation split, and number of classes (users) was found. The `DeeperCNN` model was created and trained for 30 epochs. The Adam optimizer was used with learning rate 0.001, batch size 32, and no weight decay. During training, losses and accuracies for training and validation were recorded. The best validation accuracy was printed, and trends of performance were shown with plotted curves. The trained model was saved as `best_model_task2_aug.h5`, ready for test or use later.

The training and validation results for the `DeeperCNN` model, trained again on the augmented dataset, showed big improvement over 30 epochs. The training loss was reduced to 0.3784, and a high training accuracy of 87.19% was reached. This means the model learned patterns in training data very well. The validation loss became stable at 1.1345, and validation accuracy improved to 70.36%. This showed good generalization on new data. The training and validation curves followed similar trends, with validation metrics close to training metrics. This means the model was well-regularized and did not overfit. This good performance showed that data augmentation and the best hyperparameters helped the model learn and generalize better.

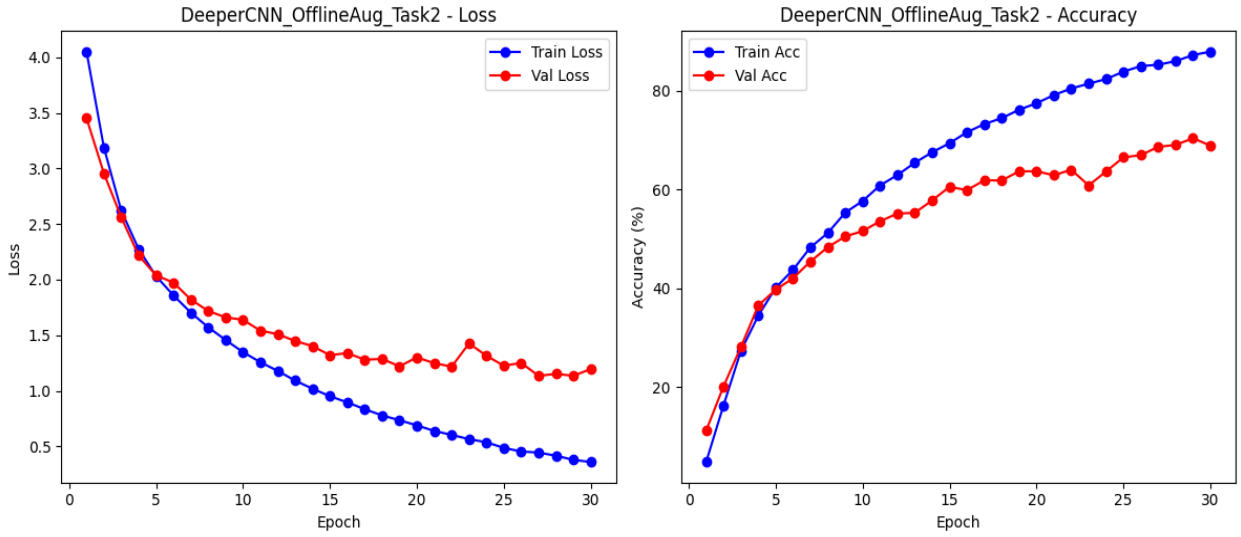

Figure 7: Training and Validation Curves for DeeperCNN with Data Augmentation

2.2.7 Comparison between Task1 and Task2

The comparison between Task 1 (`DeeperCNN` without Data Augmentation) and Task 2 (`DeeperCNN` with Data Augmentation) showed the big effect of data augmentation on model performance. In Task 1, the model had a training loss of 1.0563 and a validation loss of 1.4310. Training accuracy was 66.41%, and validation accuracy was 60.32%.

In Task 2, the results were much better. The training loss was reduced to 0.3784, and the validation loss dropped to 1.1345. Training accuracy improved to 87.19%, and validation accuracy increased to 70.36%.

The better performance in Task 2 was because data augmentation was used. Augmentation added variations to the dataset, so the model learned to generalize better. The smaller gap between training and validation accuracies in Task 2 showed the model became more robust. Also, the loss curves in Task 2 were smoother and closer, meaning overfitting was reduced.

In summary, Task 2 showed that using data augmentation helped the model learn from diverse data and improved both training and validation results.

Table 2: Comparison of Training and Validation Metrics between Task1 and Task2

	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
DeeperCNN_Task1 without Data Augmentation	1.0563	66.41%	1.4310	60.32%
DeeperCNN_Task2 with Data Augmentation	0.3784	87.19%	1.1345	70.36%

2.3 Task3: AlexNet from Scratch with Improved Setup

2.3.1 Import Libraries

For Task 3, the libraries that needed were `PyTorch` for making and training neural networks and `torchvision` for working with models, datasets, and data transformations. The `torchvision.transforms` module (imported as `T`) was used for data preprocessing and augmentation, like resizing, normalization, and image augmentations. Also, `torchvision.models` was used to build or modify models, like `AlexNet` in this task. These libraries, together with the ones already imported in Tasks 1 and 2 (like `torch`, `numpy`, `matplotlib`, and `PIL`), created a full setup for building and training `AlexNet` from scratch. They supported important tasks like data handling, visualization, and augmenting images.

2.3.2 Custom Dataset

The `ArabicHandwritingDataset` class was made special to load Arabic handwriting images for `AlexNet`. All images were resized to 224x224 pixels because this size was needed for `AlexNet`. Images were changed to RGB format using the `PIL` library, and any given transformations were applied to the images directly. If some images could not loaded, blank images were used instead, so the dataset still worked well. This version was prepared for `AlexNet` needs and stayed flexible and easy to use.

2.3.3 DataLoaders

The `get_data_loaders_updated` function was used to prepare training and validation `DataLoaders` for the augmented dataset. It included data transformations that were made special for `AlexNet`. The training transformations added random horizontal flips, rotations, and normalization. For validation, only normalization was added to keep input scaling the same. The dataset was split into training and validation parts by a given ratio, and transformations were applied to each part after splitting. `DataLoaders` were created with multi-threading (`num_workers`) and memory pinning to make data handling faster during training. The `TransformDataset` class was not needed here but could be used for adding transformations to any dataset in future. This setup gave a strong and efficient data pipeline for `AlexNet` training.

2.3.4 Build AlexNet from Scratch

The `build_alexnet` function was used to create `AlexNet` from start, without any pre-trained weights. This made it good for training on a new dataset. `AlexNet` is a famous convolutional neural network with layers for feature extraction and a fully connected classifier. In this setup, the last fully connected layer was changed to a new `nn.Linear` layer. This layer mapped to the number of classes given by the `num_classes` parameter (default was 82). This change helped `AlexNet` to work for the dataset's specific labels. The function gave an easy way to make and train `AlexNet` for custom tasks.

2.3.5 Debug Print Function

The `debug_print_val_samples` function was used to check if the validation data loader worked correctly. It got a small batch of images (default 5) from the validation dataset to check their shape and labels. The function printed the shape of the image batch (`[batch_size, channels, height, width]`) and the label tensor. This made sure that the data was loaded right and matched the needed format. It also showed the first few labels from the batch to confirm they were correct. This function was a quick tool for testing the data pipeline before training started.

2.3.6 Training, Plotting, and Saving Utilities

The `train_model` function in Task 3 added a `CosineAnnealingLR` learning rate scheduler. This scheduler slowly reduced the learning rate during the epochs to make training more stable. It was controlled by the `use_scheduler` parameter. Also, the function called `debug_print_val_samples` to check the validation data loader before training began. The plotting and saving model functions were stayed same.

2.3.7 Main Execution

The main execution script started the training for `AlexNet` with a better setup. A random seed was set for reproducibility, and the device (CPU or GPU) was selected. The `get_data_loaders_updated` function was used to load the augmented dataset with right transformations for training and validation. A better `AlexNet` model was built from scratch using the `build_alexnet` function, with the number of classes taken from the dataset. Important hyperparameters, like a learning rate of 0.0005 and the Adam optimizer, were used for stable training. The `train_model` function trained the model for 30 epochs, using a learning rate scheduler (`CosineAnnealingLR`) to make convergence smoother. The script showed and printed the best validation accuracy, drew training and validation curves with `plot_curves`, and saved the trained model in an HDF5 file for later use. This process made model training efficient and easy to repeat.

The script also had a debug step to check the validation data loader. It printed a batch of validation samples to confirm the image shapes (`[32, 3, 224, 224]`) and label shapes (`[32]`), along with example labels (`[6, 16, 76, 38, 70]`). This made sure the data pipeline worked correctly before training started.

The training and validation results for the better `AlexNet` model in Task 3 showed very good performance after 30 epochs. The training loss was reduced a lot to 0.0169, and training accuracy reached 99.52%. This meant the model learned the patterns in the training data very well. The validation loss went down to 1.0663, and validation accuracy reached 80.47%, showing strong generalization on new data.

The `CosineAnnealingLR` learning rate scheduler was likely helped for smooth training and better stability. These results showed the improved setup worked well, with data augmentation,

good hyperparameters, and the `AlexNet` model design used together. It reached high validation accuracy without overfitting too much.

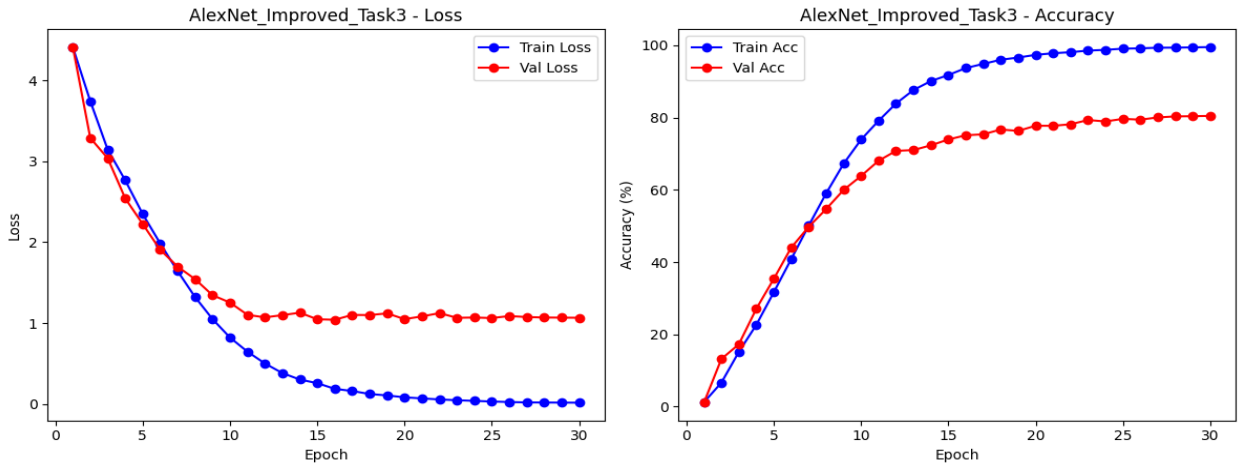

Figure 8: Training and Validation Curves for AlexNet

2.3.8 Comparison between Task2 and Task3

In Task 2, the `DeeperCNN` model with data augmentation showed strong performance. Training accuracy reached 87.19%, and validation accuracy was 70.36%. The training loss was reduced to 0.3784, and validation loss became stable at 1.1345. This showed good generalization on unseen data. The training and validation curves followed same trends, which meant the model was well-regularized with little overfitting.

But in Task 3, `AlexNet` did better. Training accuracy was 99.52%, and validation accuracy was 80.47%. The training loss was reduced a lot to 0.0169, and validation loss dropped to 1.0663. This improvement showed `AlexNet` could find more complex features because of its better architecture. The training and validation curves for `AlexNet` were smoother and closer, showing better stability and generalization than `DeeperCNN`.

In total, `AlexNet` performed much better than `DeeperCNN`, with 10.11% higher validation accuracy. It learned more efficiently and was a better choice for this dataset and task.

Table 3: Comparison of Training and Validation Metrics between Task2 and Task3

	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
DeeperCNN_Task2 with Data Augmentation	0.3784	87.19%	1.1345	70.36%
AlexNet_Task3	0.0169	99.52%	1.0663	80.47%

2.4 Task4: Pre-trained CNN Network and Transfer Learning

2.4.1 Import Libraries and Utility Functions

In Task 4, the `tqdm` library was used to show progress bars during training and evaluation. This made the process easier to see. Also, the `set_seed` function was added to set a fixed random seed (42) for `Python`, `NumPy`, `PyTorch`, and `CUDA` (if it was available). This made sure results stayed the same in all runs. It helped to check the pre-trained CNN models in a reliable way.

2.4.2 Custom Dataset and Data Augmentation

The `ArabicHandwritingDataset` class was changed to prepare handwriting images for pre-trained CNN models. Grayscale images were turned to RGB and resized to 224x224 pixels. The dataset was augmented with advanced transformations, like random horizontal flipping, rotations, color jittering, and normalization. This helped improve generalization and made the model stronger. A special transformation pipeline was used for validation data to keep it same during checking. The dataset was split into training and validation sets, and data loaders were created to handle batch processing and parallel loading. This setup made performance better for transfer learning.

2.4.3 Define Model and Training Functions

A `ResNet50` model was changed for transfer learning by replacing its last fully connected layer with a new layer for the number of classes in the dataset, so it could classify the task. Transfer learning was used with options to either fine-tune the whole model or freeze the pre-trained layers, based on the `fine_tune` parameter. The training process used a dynamic learning rate scheduler (`StepLR`) and early stopping to make training better and avoid overfitting. A flexible `train_model_transfer` function was made to handle both training and validation in each epoch, track performance, and save the best model based on validation accuracy. By adding the scheduler and early stopping, the function made sure the training process was strong and efficient, and good for transfer learning tasks.

2.4.4 Plotting and Evaluation

In Task 4, the `plot_metrics` function was used to show the training and validation loss and accuracy trends, giving a better view of how the model performed over the epochs. The `evaluate_model` function helped with a detailed evaluation by predicting labels on the validation data and collecting the predictions and true labels for checking accuracy and performance. These utilities were used to make the visualization clear and the evaluation strong for the transfer learning model.

2.4.5 Main Execution

In this section, the transfer learning process was run using a pre-trained `ResNet50` model. First, key parameters like the dataset directory, batch size, number of epochs, learning rate, and optimizer type were set. The `get_data_loaders` function loaded and prepared the dataset, splitting it into training and validation sets with the right transformations for `ResNet50`. The `build_resnet50` function changed the `ResNet50` model by replacing the fully connected layer with a new layer that could predict 82 classes. The model was moved to the right computing device (GPU or CPU), and the `train_model_transfer` function trained it using the Adam optimizer with a scheduler to adjust the learning rate. Early stopping was used to stop the training if validation accuracy did not improve, to avoid overfitting. At the end, the script saved the trained model weights as a .pth file for later use.

The structure of the `ResNet50` model displayed its deep architecture, which included many convolutional and bottleneck layers, batch normalization, and adaptive pooling. These parts made `ResNet50` very good for extracting features and classification tasks. The script showed that the `ImageNet`-pretrained weights were downloaded correctly. The model's modified fully connected layer was changed to predict 82 classes in the dataset. During training, the metrics like loss and accuracy were tracked, and the model reached its best validation accuracy, showing how effective transfer learning was with the fine-tuning of a pre-trained model. This process used `ResNet50`'s strong features for the dataset, leading to big improvements in performance.

The training and validation metrics for the `ResNet50` model over 15 epochs showed very good performance. The training loss quickly decreased, reaching almost zero by the last epoch, with training accuracy reaching 100%. The validation loss stayed low at 0.0177, and the validation accuracy reached 99.53%. The plotted curves for both loss and accuracy showed steady improvements during training, with very small difference between the training and validation metrics, which showed good generalization and no overfitting. These results showed how effective transfer learning with the `ResNet50` model and fine-tuned settings were for the dataset.

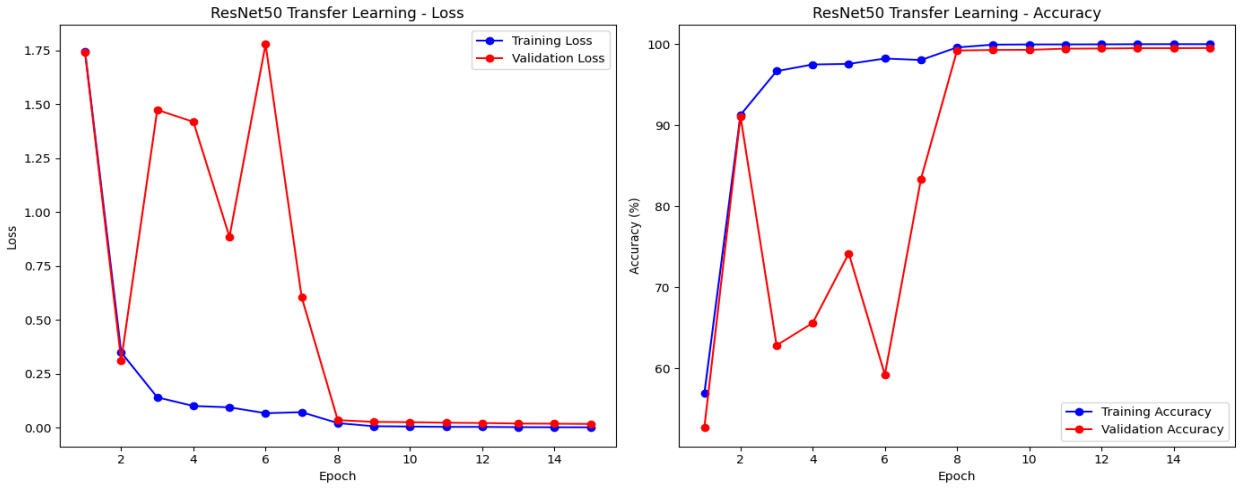

Figure 9: Training and Validation Curves for ResNet50 Transfer Learning

2.4.6 Comparison between Task3 and Task4

The comparison between Task 3 (`AlexNet`) and Task 4 (`ResNet50` with Transfer Learning) showed big differences in training speed, accuracy, and performance on validation data. In Task 3, `AlexNet` was trained from scratch with better settings. The training loss was 0.0169 and the validation loss was 1.0663. The training accuracy reached 99.52%, but the validation accuracy was 80.47%. This showed that the model worked well, but there was still a gap between training and validation. The learning curves showed a steady decrease in training and validation loss over 30 epochs, showing good progress and optimization. However, the higher validation loss suggested that the model could be better at handling new data.

In Task 4, `ResNet50` with transfer learning was used, and it performed better. The training loss was 0.0018 and the validation loss was 0.0177. The training accuracy reached 100%, and the validation accuracy improved to 99.53%. The pre-trained weights and fine-tuning allowed faster training and better generalization, seen in how quickly the training and validation metrics stabilized in just 15 epochs. The loss and accuracy curves showed steady performance, with very little difference between training and validation, showing that `ResNet50` was better at capturing complex patterns in the data. Overall, Task 4 did better than Task 3 in both accuracy and speed, showing that transfer learning works better than training from scratch, especially for complex models and big datasets.

Table 4: Comparison of Training and Validation Metrics between Task3 and Task4

	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
AlexNet_Task3	0.0169	99.52%	1.0663	80.47%
ResNet50 Transfer Learning_Task4	0.0018	100%	0.0177	99.53%

3. Conclusion

The project solved the problem of Arabic handwriting classification by testing different deep learning methods. It started with a simple CNN in Task1, and in Task 2, the performance was improved by using data augmentation. This helped the model to generalize better. In Task 3, `AlexNet`, a more advanced architecture, was trained and showed higher validation accuracy than the custom CNN. Task 4 used transfer learning with a pre-trained `ResNet50` model, which reached almost perfect accuracy and very good generalization. The experiments showed how important the choice of model, data augmentation, and pre-trained networks was for getting high performance. The goal of improving classification accuracy was reached step by step, and `ResNet50` gave the best results, making it a strong choice for real-world use.

Future work can focus on making the model stronger and work better for more tasks. One idea is to use advanced data augmentation, like `GANs`, to create fake data and solve the problem of having a small dataset. Another idea is to try lightweight models like `MobileNet` or `EfficientNet`. These models can give good accuracy but need less power, so they can work on small devices. Also, the dataset can be made bigger by adding more classes and different handwriting styles. This will help the model to work well in real life. Finally, the model can be added to an easy-to-use web or mobile app. This app can help with real-time handwriting recognition for things like learning tools or saving old documents digitally.