Neural Networks - Deep Learning First Assignment

Dimitrios Karatis 10775, *Electrical and Computer Engineering, AUTH*

***Abstract*—This document provides an overview of the neural network model training and evaluation process using the CIFAR-10 dataset. The focus is on leveraging convolutional neural networks (CNNs) to classify images into ten categories, such as airplanes, cars, and animals, making this a challenging yet foundational problem in computer vision.**

***Index Terms*—Neural networks, deep learning, CIFAR-10, image classification, computer vision.**

I. Introduction

TO effectively train and evaluate the neural network models presented in this work, the **CIFAR-10 Dataset** was selected as the primary data source (available at: CIFAR-10 Dataset). CIFAR-10 is a popular image classification dataset containing 60,000 32x32 color images divided into ten classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. This dataset serves as a benchmark in computer vision and deep learning for evaluating classification model performance.

II. Dataset Description

The CIFAR-10 dataset consists of 50,000 training images and 10,000 test images, each labeled with one of the ten classes. Due to its diversity, it presents a complex classification problem and provides a comprehensive platform for testing and training CNN architectures in the context of image recognition.

III. Interim Assignment

A. Loading and Preparing the Dataset

The archive of this dataset contains the files data_batch_1, data_batch_2, ..., data_batch_5, as well as a test_batch and a batches.meta file, which contains the label names in terms of string charachters, and which I didn't use. Each of these files is a Python "pickled" object produced with cPickle. To load the image data and labels found in the CIFAR's archive I used the ***unpickle*** function found in the official CIFAR's website. This function deserializes a binary file containing a pickled object and returns a dictionary with the corresponding image data and labels. This way, each of the batch files contains a dictionary with the following elements:

- **data** – a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.
- **labels** – a list of 10000 numbers in the range 0-9. The number at index i indicates the label of the ith image in the array data.

Following that, the ***load_cifar10_data*** function reads and combines the five training batches from the specified directory, each containing images and corresponding labels. The images (x_train) and labels (y_train) from each batch are appended and then converted into arrays. It then loads a separate test batch for the test images (x_test) and labels (y_test). The ***preprocess*** matrix indicates whether or not normalization, augment by rotation, validation set (for Early Stopping) and PCA (Principal Components Analysis) will be used or not. If the first element of the matrix is set to 1, normalization is active etc. Finally, the function returns the preprocessed training, test, and validation datasets, along with the PCA-related parameters such as the number of components, explained variance ratio, and the PCA and scaler objects, if needed. The last ones are used in order to find the optimal number of PCA components needed to explain a percentage of the variance in the data.

The ***custom_rotation*** function augments an image dataset by adding randomly rotated versions of the images, enhancing variability to improve model generalization. Each flattened image (shape (3072,)) is reshaped into its original 3D form (32, 32, 3), rotated by a random angle between -20 and 20 degrees (excluding 0), and then flattened back. The function collects these rotated images and their corresponding labels, concatenates them with the original dataset, and returns the augmented dataset. This process increases the dataset size and helps the model handle rotational variations in real-world scenarios, reducing overfitting and improving robustness.

The ***apply_pca*** function reduces the dimensionality of image data while retaining a specified amount of variance, improving computational efficiency and potentially enhancing model performance. It first flattens the 3D image data into 1D arrays and standardizes them to have a mean of 0 and a standard deviation of 1. Then, it computes the cumulative explained variance using PCA on the training data and determines the minimum number of components needed to retain a specified variance threshold (default 95%). PCA is reapplied with this optimal number of components, transforming the training, testing, and optionally validation data into lower-dimensional

This document was prepared by Dimitrios Karatis as part of the first assignment for the Neural Networks course at Aristotle University of Thessaloniki.

representations. The function saves and reloads the PCA model for future use and returns the transformed datasets, the number of components, the explained variance ratios, the PCA and scaler objects. The code for all the functions used to load and preprocess the dataset can be found at the ***dataset_functions.py***. Also, in the ***paths_config.py*** file, the user can change the necessary paths.

```
# Function to load cifar-10 data
# preprocess is matrxix indicating:
# [using normalize, using roatations, using
    validation set for EarlyStopping, using PCA]
def load_cifar10_data(self, data_dir, preprocess
    ):
    x_train, y_train, x_test, y_test, x_val,
        y_val = [], [], [], [], [], []
    explained_variance_ratio = None
    pca_object = None
    scaler_object = None
    n_components = 0

    # Load training folder
    for i in range(1, 6):  # Load the 5 training
         batches
        batch = self.unpickle(f"{data_dir}/
            data_batch_{i}")
        x_train.append(batch[b'data'])
        y_train.extend(batch[b'labels'])
    x_train = np.concatenate(x_train)
    y_train = np.array(y_train)

    # Load test folder
    test_batch = self.unpickle(f"{data_dir}/
        test_batch")
    x_test = test_batch[b'data']
    y_test = np.array(test_batch[b'labels'])

    # Check if validation split is enabled
    if preprocess[2] == 1:
        # Split the training data into training
            and validation sets
        x_train, x_val, y_train, y_val =
            train_test_split(x_train, y_train,
            test_size=0.2, random_state=42)

    # Check if rotation augmentation is enabled
    if preprocess[1] == 1:
        x_train, y_train = self.custom_rotation(
            x_train, y_train)

    # Check if normalization is enabled
    if preprocess[0] == 1:
        x_train = self.normalize_pixels(x_train)
        x_test = self.normalize_pixels(x_test)
        # Normalize x_val if it exists
        if len(x_val) > 0:
            x_val = self.normalize_pixels(x_val)

    # Check if PCA is enabled
    if preprocess[3] != 0.0:
        x_train, x_test, x_val, n_components,
            explained_variance_ratio, pca_object
            , scaler_object = self.apply_pca(
            x_train, x_test, x_val, preprocess
            [3])
        self.plot_cumulative_variance(
            explained_variance_ratio)

    return x_train, y_train, x_test, y_test,
        x_val, y_val, n_components,
        explained_variance_ratio, pca_object,
        scaler_object
```

B. Training and Evaluating Classifiers

After loading the dataset we then need to initialize the classifiers, and train every single one of them to a set of training images (img_train) with corresponding labels (label_train). After training, the classifiers are evaluated on a test set (img_test, label_test), where they predict labels for the test images and compute accuracy based on these predictions. The initialization, training and evaluation has been done by using the KNeighborsClassifier and NearestCentroid packages from sklearn.

For each classifier, the accuracy is printed as a percentage, allowing a comparison of performance. Additionally, the function ***generate_conf_matrix*** generates a confusion matrix for each classifier, helping to visually assess where the models correctly or incorrectly classified the test images, and making it easier to analyze the strengths and weaknesses of each approach.

Results:

```
k-NN (k=1) Accuracy: 35.39%
k-NN (k=3) Accuracy: 33.03%
Nearest Centroid Accuracy: 27.74%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 1. Results for different classifiers (original dataset)

```
k-NN (k=1) Accuracy Normalized: 35.39%
k-NN (k=3) Accuracy Normalized: 33.03%
Nearest Centroid Accuracy Normalized: 27.74%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 2. Results for different classifiers (normalized dataset)

```
k-NN (k=1) Accuracy (Augmented): 35.32%
k-NN (k=3) Accuracy (Augmented): 33.45%
Nearest Centroid Accuracy (Augmented): 27.30%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 3. Results for different classifiers (augmented dataset)

```
k-NN (k=1) Accuracy Normalized (Augmented): 35.49%
k-NN (k=3) Accuracy Normalized (Augmented): 33.93%
Nearest Centroid Accuracy Normalized (Augmented): 27.28%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 4. Results for different classifiers (normalized and augmented dataset)

As we can see, all of the classifiers yield roughly the same results, even if we perform normalization, rotation, augmentation or both on the dataset beforehand. By normalization, I mean dividing the pixel values of the data images by 255, so that their respective range now becomes from zero to one. Also, by analyzing the results it is evident that there are limitations when it comes to using simple methods like k-Nearest

Neighbors (k-NN) and Nearest Centroid for complex image data, since we can see that the accuracy on those methods is extremely low. More specifically, the k-NN classifier with k=1 performed best, with an accuracy of 35.49%, while the one with k=3 scored slightly lower at 33.93%. This means that looking at just the closest image gives slightly better results than averaging across more neighbors. The nearest centroid method did the worst, with an accuracy of only 27.28%, likely because averaging all images in a class loses important details. Normalizing the data didn't change the results, which suggests that pixel scaling alone isn't enough to improve these models, although there were some minuscule changes when applying the augmentation. Overall, these results suggest that CIFAR-10 is too complex for these simple classifiers, and more advanced methods like deep learning would likely perform better. Below we can see the Confusion Matrices in the case that we use normalization and augmentation for each one of the classifiers. The implementations of the classifiers can be found at the ***classifier.py*** file and the corresponding main function at the ***main_classifiers.py***.

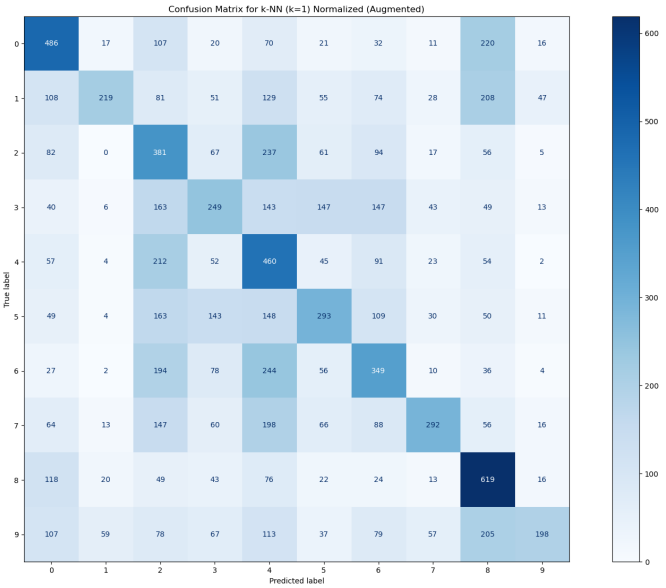

Fig. 5. k-NN (k=1) confusion matrix (normalized and augmented dataset)

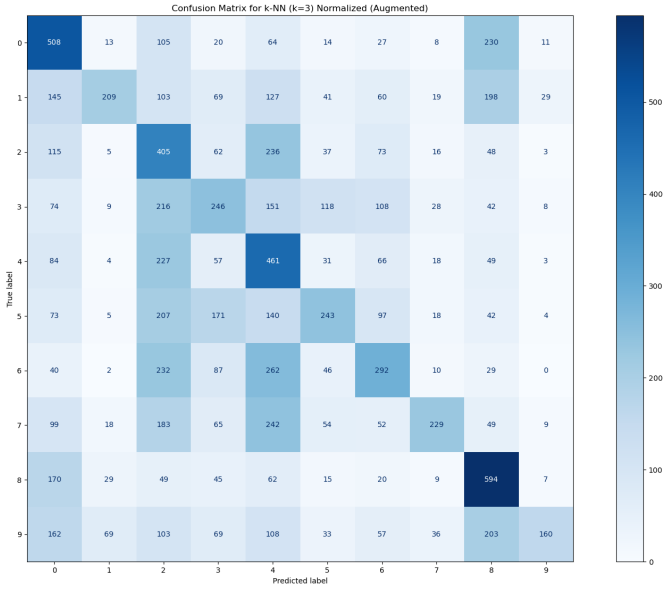

Fig. 6. k-NN (k=3) confusion matrix (normalized and augmented dataset)

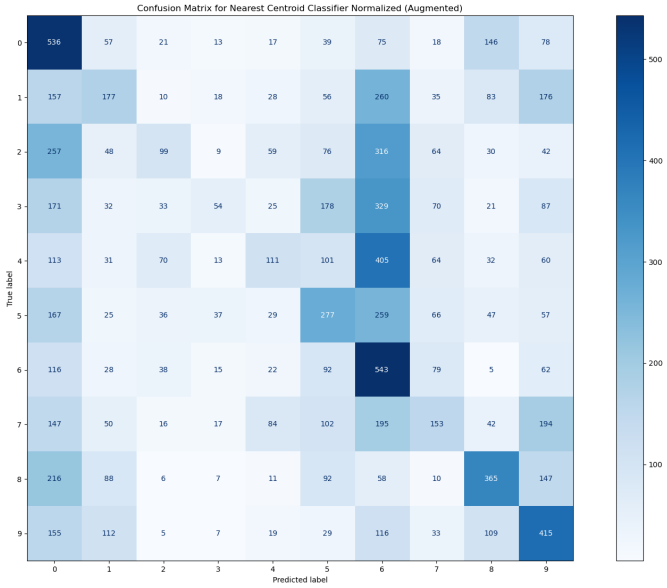

Fig. 7. Nearest Centroid confusion matrix (normalized and augmented dataset)

C. Training and Evaluating Multilayer Perceptron (MLP)

To mitigate the limitations of the k-NN and nearest centroid classifiers, which may struggle with the high-dimensional and complex nature of the CIFAR-10 dataset, I implemented my own Multi-Layer Perceptron (MLP). The MLP leverages multiple layers of neurons with nonlinear activation functions, allowing it to learn more complex patterns and relationships within the data. This approach aims to create a more robust network capable of better handling the challenges posed by the dataset and improving classification performance. The code for my own implementation of an MLP can be found in the ***myMLP.py*** file and the corresponding main function, in the ***main_myMLP.py***. At the end of the main function i am also using images from the web to test the network under real life conditions. The images used for this are located in the ***cifar10Testbench*** folder. The final implementation of my MLP network works like this:

The myMLP class creates a custom Multi-Layer Perceptron (MLP) model for image classification tasks. It is initialized with parameters such as the number of output classes, the number of neurons in each hidden layer, activation functions, input shape, dropout rate, and learning rate. The MLP is constructed by initializing weights and biases using Xavier initialization to ensure stable gradients during training.

The forward pass through the network calculates activations for each layer, applying the respective activation functions (ReLU or Tanh) and optionally applying dropout for regularization during training. The output layer uses the softmax function to convert raw scores into probabilities for classification.

During training, the model uses mini-batch gradient descent to update the parameters based on the computed gradients from the backward pass (backpropagation). The loss function used is sparse categorical cross-entropy, and early stopping is implemented to prevent overfitting by halting training when the validation loss doesn't improve for a specified number of epochs.

The class also includes methods for predicting class labels on unseen data, evaluating the model's performance, and plotting training and validation accuracy over the epochs to visualize learning progress. The train method is used to train the MLP, while the evaluate method computes the accuracy and prediction probabilities on the test set.

FIRST STEPS:

When I first created the network, I didn't use Xavier initialization for the weights and biases. During initial training with various numbers of layers and neurons, the training, validation, and test accuracy remained around 0.1, indicating that the model was essentially random guessing and failing to learn from the data. This pointed to an issue in the code, prompting me to search for a solution. I then discovered the Xavier initialization method, which I implemented, and it finally led to more logical results. Xavier initialization is used in this MLP model along with normalization to ensure that the weights are set to values that help prevent issues with vanishing or exploding gradients. This initialization method sets the weights according to a distribution with a variance that is inversely proportional to the number of input units to the layer. By doing so, it helps maintain the scale of the activations throughout the network while improving the overall stability of training. *The hierarchy of the experiments went like this:*

1. Baseline model: First i trained a simple architecture with the raw dataset (i only did normalization on it).
2. Optimizing Core Architecture: Next i tuned the number of layers and neurons to improve performance to a point.
3. Regularization: After that i added dropout, early stopping and adjusted their parameters, while also adding random shuffling to the training set.
4. Data Improvements: Then I applied augmentations (i only did rotations, i lacked the time) to enhance generalization.
5. Advanced Features: Lastly I experimented with PCA.

1. Baseline model:

Initially, I began with a single layer consisting of 64 neurons, using the 'tanh' activation function, and no dropout. The model was trained with a batch size of 32 for 100 epochs, a learning rate of 0.01, and a preprocessing matrix of [1,0,1,0], meaning use only normalization and split to validation set. I also set the early stopping patience to 1000, effectively disabling early stopping. **The results were:**

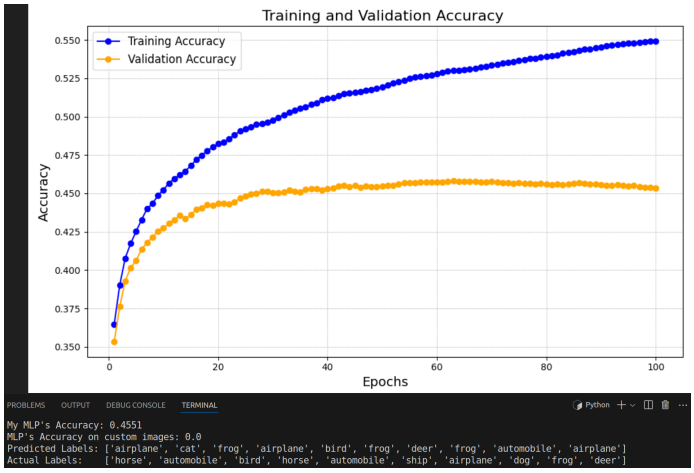

Fig. 8. Training and validation results. (Time it took: 4mins in Aristotelis Cluster)

2. Optimizing Core Arichitecture:

As we can see the accuracy of the model in the test set is 0.4551 which is already higher than the accuracy of every classifier. Although, we can see that after around the 40th epoch the validation accuracy stays the same while the training accuracy rises, indicating possible overfitting. Also, the accuracy of the model using new images from the web is at 0.0, indicating that the model doesn't generalize well at all. Next, i tried to change the activation function from 'tanh' to 'relu'. **The results were:**

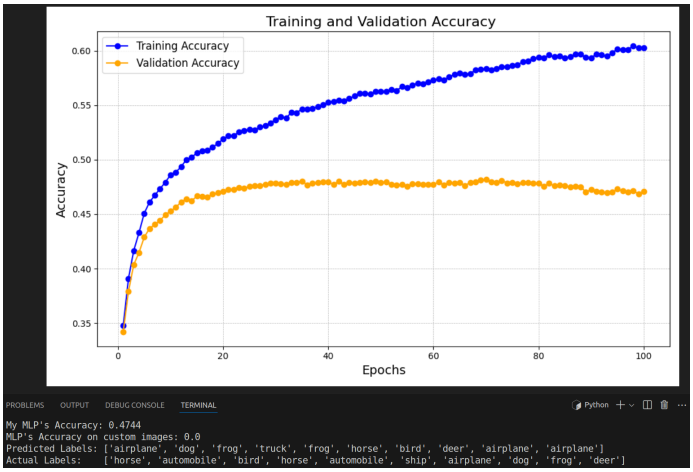

Fig. 9. Training and validation results. (Time it took: 4mins in Aristotelis Cluster)

Now the accuracy on the test set went to 0.4744 but still the accuracy of the web images is at 0.0. Now i started adding more layers. I began with adding one more layer with 128 neurons at the begging of the previous one, meaning that now we have two hidden layers with [128, 64] neurons each. **The results were:**

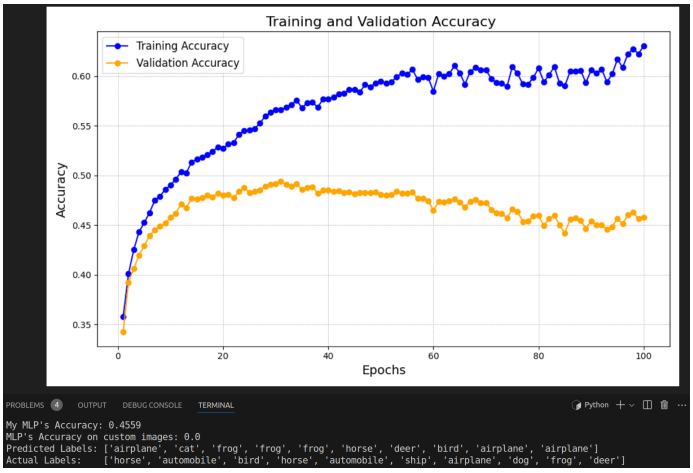

Fig. 10. Training and validation results. (Time it took: 6mins in Aristotelis Cluster)

Adding an additional layer with 128 neurons increased the model's learning capacity, but it did not lead to improved accuracy. Specifically, the test accuracy dropped to 0.4559, which is worse than before. Additionally, the training accuracy increased, while the validation accuracy began to decline noticeably, suggesting the model is overfitting with a rate worse than before. Also, we can see that around the 30th epoch the validation accuracy is at it's peak (also better than before) and then it starts declining. So by using regularization method we will have better overall results. Lastly, the model's accuracy to new images from the web is still at 0.0. Next, i tried to add one more hidden layer with 256 neurons, making them: [256 ,128, 64]. **The results were:**

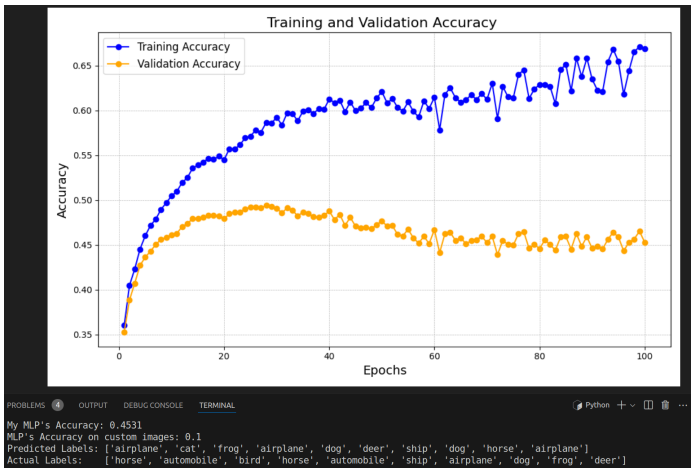

Fig. 11. Training and validation results. (Time it took: 12mins in Aristotelis Cluster)

Once again, we observe that the training accuracy improves, while the validation accuracy worsens, indicating signs of overfitting. At this stage of the experiment, I am not overly concerned about the decline in validation accuracy or the model's tendency to overfit, as this will be addressed later with regularization techniques like dropout. Additionally, the model's accuracy on web images dropped to 0.1, but this result isn't particularly significant at this point due to the evident overfitting. Following this, I adjusted the number of neurons in each layer to: [512, 256, 128]. **The results were:**

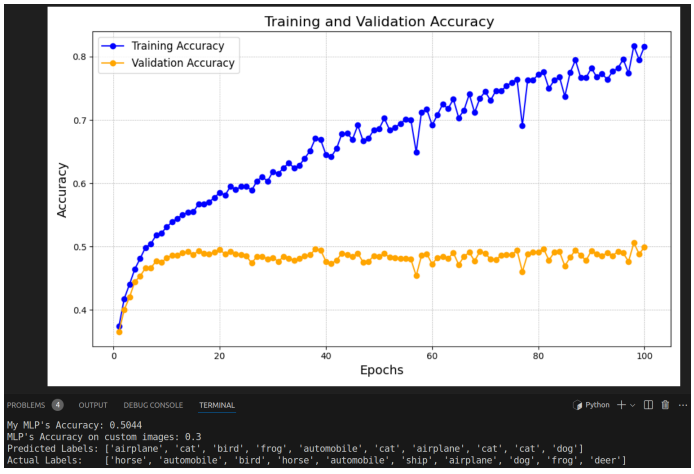

Fig. 12. Training and validation results. (Time it took: 28mins in Aristotelis Cluster)

As we can see, The [512, 256, 128] model showed more significant overfitting, with the training accuracy exceeding 80% while the validation accuracy plateaued around 50%. Although the model seems to be doing much better at the traning. Also, the [512, 256, 128] model achieved an accuracy of 30% on custom images, while the [256, 128, 64] model performed worse, with only 10% accuracy. This suggests that while the smaller model has reduced overfitting, it may lack the representational capacity to generalize effectively. Lastly, i experimented with adding one more hidden layer at the end with 64 neurons, making the overall architecture: [512, 256, 128, 64]. **The results were:**

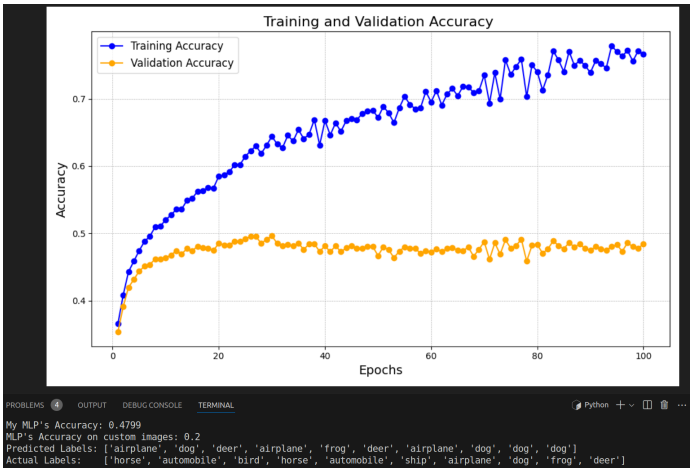

Fig. 13. Training and validation results. (Time it took: 29mins in Aristotelis Cluster)

Between the two architectures, the 3-layer model with [512, 256, 128] neurons performed better than the 4-layer model with [512, 256, 128, 64] neurons. The 3-layer model achieved a higher accuracy of 0.5044 compared to 0.4799 for the 4-layer model and demonstrated slightly better generalization with a custom image accuracy of 0.3 versus 0.2. The validation accuracy for the 3-layer model was more stable and higher than the 4-layer model, indicating less overfitting, while the 4-layer model exhibited a larger gap between training and validation accuracies, suggesting greater overfitting and instability. Therefore, the 3-layer architecture seems more suitable for further enhancements like adding dropout for improved performance.

3. Regularization:

Even though the 3-layer model looked like the better option to go with, the extra capacity of the 4-layer model might be useful if the right techniques are applied. So, I decided to try regularization and data augmentation on both models before making a final choice. To do this, I added 30% dropout, early stopping with a patience of 5, and random shuffling of the training data at the start of each epoch. **The results were:**

For the 3-layer model:

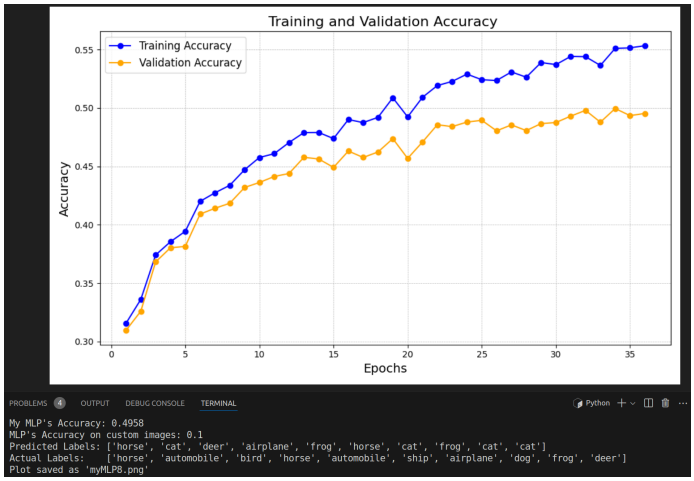

Fig. 14. Training and validation results. (Time it took: 11mins in Aristotelis Cluster)

For the 4-layer model:

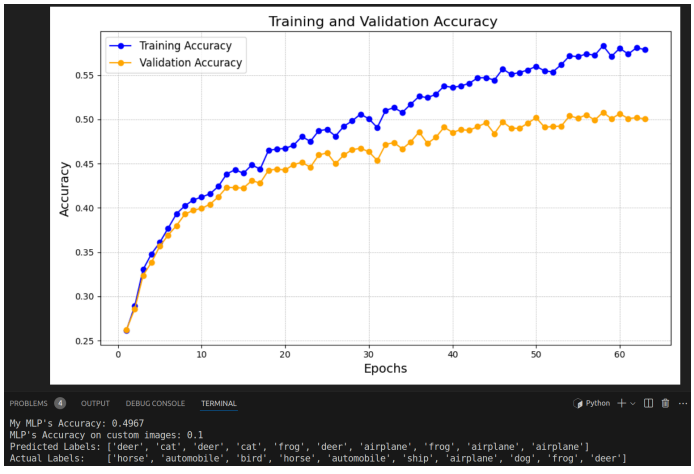

Fig. 15. Training and validation results. (Time it took: 19mins in Aristotelis Cluster)

4. Data Improvements:

After applying 30% dropout, both models showed improved generalization, with reduced training-validation gaps. The larger model ([512, 256, 128, 64]) achieved a slightly higher validation accuracy of 0.4967 compared to 0.4958 for the smaller model ([512, 256, 128]). However, the performance difference is minimal, and the smaller model converged faster with greater stability, making it more efficient. Considering its simplicity and comparable accuracy, it seems like the small model is better to proceed with. Although it is worth testing both models under the effects of training data augmentations (i only did rotations), so that's exactly what I did next. **The results were:**

For the 3-layer model:

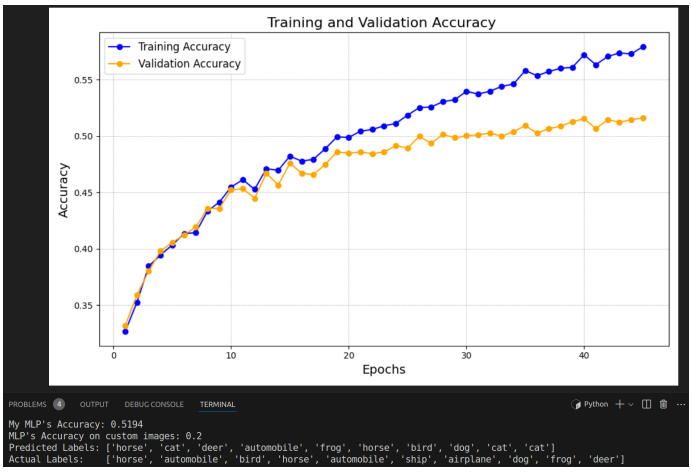

Fig. 16. Training and validation results. (Time it took: 27mins in Aristotelis Cluster)

For the 4-layer model:

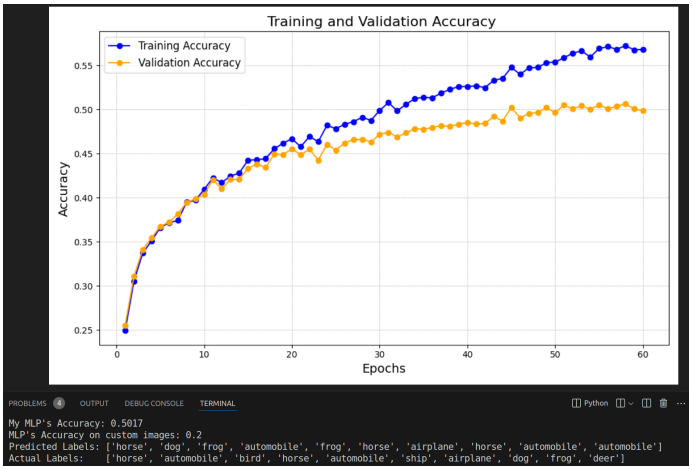

Fig. 17. Training and validation results. (Time it took: 37mins in Aristotelis Cluster)

5. Advanced Features:

Both models show steady improvement in training and validation accuracy over time, with minimal signs of overfitting. The [512, 256, 128] model does slightly better than the [512, 256, 128, 64] model, with a validation accuracy of 51.94% compared to 50.17%. It's also simpler, with fewer parameters, so the training is faster and more efficient. Since both models perform about the same on web images, the smaller one is the better pick because it's a bit more accurate and less complicated. Lastly, i experimented using PCA on the smaller model ([512, 256, 128]) with 1346 components needed for 99.0% of the variance. **The results were:**

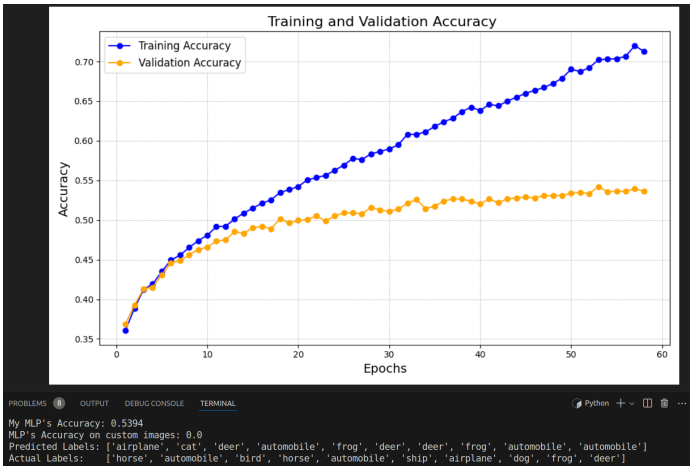

Fig. 18. Training and validation results. (Time it took: 20mins in Aristotelis Cluster)

The results show that the MLP is doing okay on the CIFAR-10 dataset, hitting about 70% training accuracy and 54% validation accuracy after 60 epochs when using PCA with 99% variance. While the training accuracy keeps improving, the validation accuracy flattens out pretty early, which suggests some overfitting. On the CIFAR-10 test set, the accuracy is 53.94%, which isn't too bad, but the performance on the custom web images is disappointing, with 0.0% accuracy. None of the predictions match the actual labels, meaning the model just isn't handling those images well. This could be because the PCA transformation, while helping on CIFAR-10, doesn't work as well for the web images since they're probably quite different from the training data in terms of features like backgrounds, lighting, or styles etc. In order to reduce overfitting i increased the dropout rate to 50%. **The results were:**

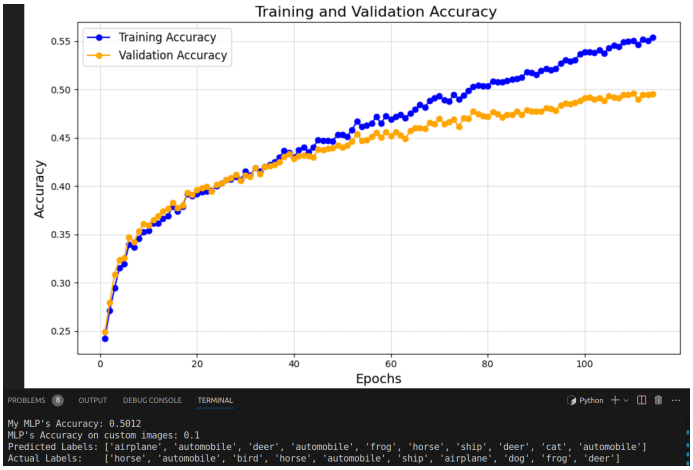

Fig. 19. Training and validation results. (Time it took: 36mins in Aristotelis Cluster)

As we can see, the test accuracy decreased a bit but the accuracy on the web images increased from 0.0 to 0.1, indicating better generalization. The last thing I tried was using PCA with 235 components for 90.0% variance and **the results were:**

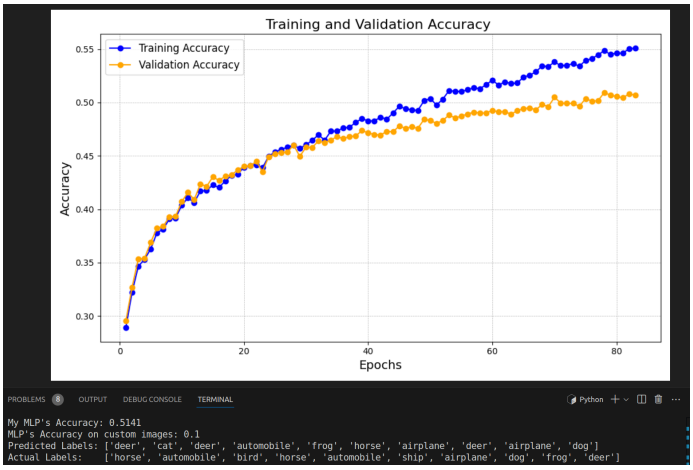

Fig. 20. Training and validation results. (Time it took: 30mins in Aristotelis Cluster)

As far as we can notice, the difference now is pretty small. Test accuracy saw a slight improvement, but the accuracy on the web images stayed the same. Overall, it seems like applying PCA hurt the performance on the web images while having little effect on the model's test accuracy. One possible explanation is that PCA might have removed information that's important for classifying the web images but not as critical for CIFAR-10. If the web images differ a lot from CIFAR-10 in terms of lighting, backgrounds, or overall style, PCA could be unintentionally cutting out features that are key for recognizing those differences. Overall, I would stick to the 3-layered model: [512, 256, 128] with a dropout of 30%, random shuffling of the training data each epoch, and a preprocess matrix of [1, 1, 1, 0], meaning use normalization, rotations, and validation set for early stopping but not PCA.

D. Training and Evaluating my custom MLP and an MLP from keras

In order to test my custom MLP's correctness and performance, I decided to implement a more optimized MLP (using tensorflow keras) with the exact same number of hidden layers and neurons as my custom MLP ([512, 256, 128]), and test both of them on the cifar10 dataset. The code for the MLP from keras can be found at the ***MLP.py*** file and the corresponding main function in the ***main_MLP.py***. That beeing said, for network architectures of [512, 256, 128], 'relu' activation functions, a dropout of 30%, early stopping patience of 5, a batch size of 32 trained for 100 epochs and a preprocess matrix of [1, 1, 1, 0], i got **these results:**

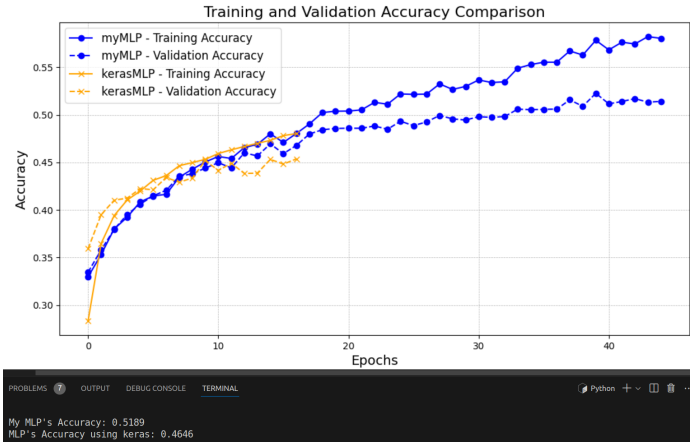

Fig. 21. Training and validation results. (Time it took: 40mins in Aristotelis Cluster)

It seems like the custom MLP ended up performing better than the Keras MLP in both training and validation accuracy, even though both models have the same architecture, hyperparameters, and dataset. This could be because of small differences in how the two were implemented, like how weights were initialized, how gradients were calculated, or how updates were applied during training. Although it is possible that my implementation has something wrong with it as well. To make sure the improvement isn't just luck or specific to this dataset, it would be better to test the custom MLP on other datasets (like I did with the MNIST below).

Lastly, because the accuracy of the model when using images from the web wasn't what I hoped it would be, I decided to test my custom MLP on a much simpler dataset, the MNIST. So first I trained and evaluated the model on the MNIST training and test sets, and then I evaluated it's performance on images of handwritten digits from the web, in order to see the output on real "world data", exactly as I did with my own custom MLP before. The corresponding main function can be found in the ***main_MNIST_MLPs.py***. That beeing said, for a network architecture of [512, 256, 128], 'relu' activation functions, a preprocess matrix of [1, 0, 1, 0] (only use normalization and validation set) a dropout of 40%, early stopping patience of 5, and a batch size of 32 trained for 100 epochs, I got **these results:**

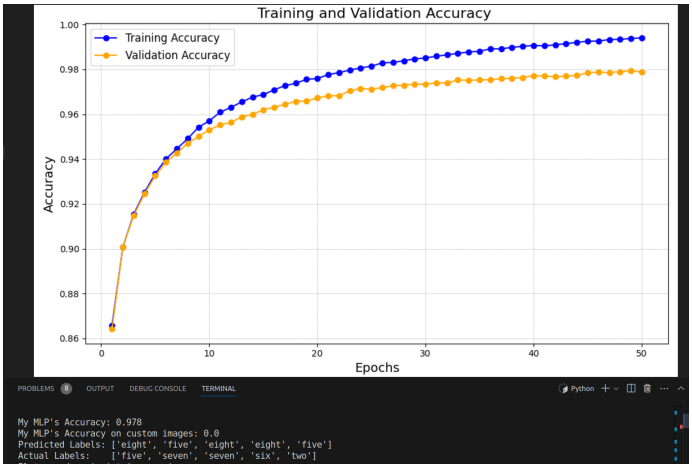

Fig. 22. Training and validation results. (Time it took: 5mins in Aristotelis Cluster)

So the accuracy of the test set is not a surprise, since MNIST is a simple dataset. What is worrying, is that although we got a 0.978 on the test accuracy, the accuracy of the model on "real data" (handwritten images from the web) is at 0.0. So in order to see what's going on with that, I decided to visualize the images from the MNIST dataset. **By visualizing the first four images of the testing dataset, and the first four web images i got:**

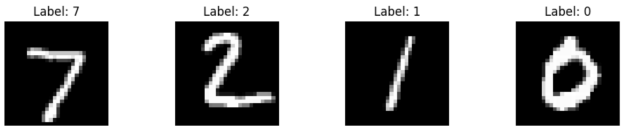

Fig. 23. The first four test images of MNIST dataset.

Fig. 24. The first four test images of my custom dataset.

What is evident is that my images have white background and black digits, while for the MNIST dataset it's the opposite. Next i tried inverting the colors of my images to see if that yields better results, and it did:

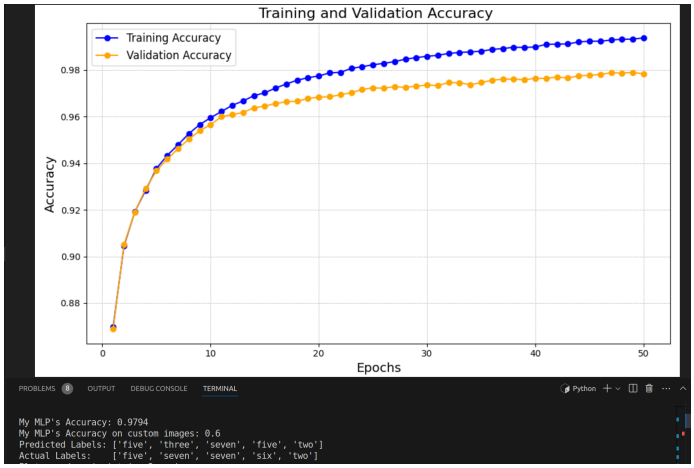

Fig. 25. Training and validation results. (Time it took: 6mins in Aristotelis Cluster)

As we can see while the test accuracy remained almost the same, the accuracy of the model on new images from the web went to 0.6. Keep in mind that for the MNIST dataset I did not use any augmentations. If I used brightness and rotation augmentations there is a high chance that the model yielded better results on the non inverted images as well.

E. Training and Evaluating Convolutional Neural Network (CNN)

Although the MLP performed better than the k-NN and nearest centroid classifiers, its results were still not fulfilling. To improve performance, I decided to try a Convolutional Neural Network (CNN) from keras, which is better suited for handling image data by extracting important features through convolutional layers. This approach aims to provide better results for image classification tasks. The code for the CNN can be found at the ***CNN.py*** file and the corresponding main function in the ***main_CNN.py***.

The CNN architecture used in the main function is composed of three convolutional blocks and a fully connected (dense) layer. Each convolutional block consists of a Conv2D layer with 32, 64, and 128 filters, respectively, using 3x3 kernels with ReLU activation, followed by batch normalization and another Conv2D layer with the same settings. Each block also includes max pooling (2x2) and dropout layers to reduce overfitting. After the convolutional layers, the output is flattened and passed through a dense layer with 256 neurons and ReLU activation. A final dense layer with 10 neurons and a softmax activation function outputs class probabilities for the CIFAR-10 dataset's 10 classes. Dropout is applied after each dense layer to further prevent overfitting. The model uses the Adam optimizer and sparse categorical cross-entropy loss for training. **The results were:**

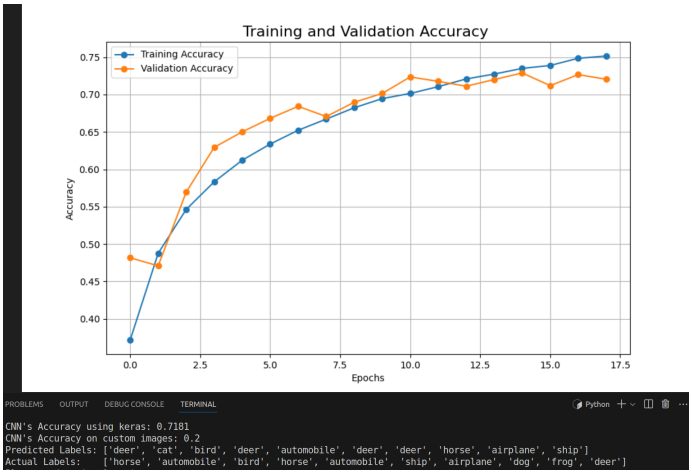

Fig. 26. Training and validation results. (Time it took: 40mins in Aristotelis Cluster)

From the graph, it is evident that the CNN trained on the CIFAR-10 dataset achieves steadily increasing accuracy for both training and validation data over 17 epochs, with minimal overfitting as the curves for training and validation accuracy remain close. The final training accuracy is approximately 0.75, while the validation accuracy is slightly lower at around 0.70, indicating the model generalizes reasonably well to unseen data within the CIFAR-10 dataset. However, the model performs poorly on custom images sourced from the web, achieving an accuracy of only 20% (0.2).

What I realized rather late was that this significant drop in the accuracy of my custom images in all of my models is likely due to the custom images having a much higher resolution and possibly different characteristics compared to the 32x32 images in CIFAR-10, leading to a domain gap that the models struggle to handle.

Aside from that, in order to get some examples of correct and incorrect classifications (Actual Labels vs Predicted Labels) needed for this project, I used my own custom images dataset, which again might not be the best idea since the model was trained in a dataset with images with much lower resolution than the one it was evaluated on.