NEURAL NETWORKS - DEEP LEARNING FIRST ASSIGNMENT

Dimitrios Diakoloukas 10642, Electrical and Computer Engineering, *AUTH, Student*

***Abstract*—This project implements and compares three models—k-Nearest Neighbor (k-NN), Convolutional Neural Network (CNN), and Multi-Layer Perceptron (MLP)—for multi-class classification on the CIFAR-10 dataset. Each model's performance was evaluated based on accuracy, using both training and testing data. The CNN demonstrated superior feature extraction and classification capabilities, while the MLP provided insight into performance with fully connected layers. The k-NN model served as a baseline comparison. Results highlight each model's strengths and limitations, offering a detailed analysis of their effectiveness on CIFAR-10.**

I. Introduction

IMAGE classification is an essential task in computer vision, used in areas like autonomous driving and facial recognition. This project tests variations of three models—k-Nearest Neighbor (k-NN), Convolutional Neural Network (CNN), and Multi-Layer Perceptron (MLP)—on the CIFAR-10 dataset, which has 60,000 images in 10 classes.

The k-NN models serve as a simple, distance-based baseline. The CNNs use convolutional layers to capture patterns in images, making them especially effective for this task. The MLPs, a fully connected network, show how traditional neural networks handle image classification.

Each model's accuracy is evaluated on CIFAR-10 to compare their strengths and weaknesses, highlighting the advantages of more advanced models like CNNs over simpler ones like k-NN.

II. Data Loading and Preprocessing

This section explains how Iloaded the CIFAR-10 dataset and prepared it so it could be used with the k-Nearest Neighbor (k-NN) algorithm.

A. *Loading CIFAR-10 Data*

The CIFAR-10 dataset contains 60,000 small color images (32x32 pixels) across 10 different classes. There are 50,000 images for training and 10,000 images for testing. Each image is stored as a set of pixel values, and the data is divided into multiple files.

I used the following steps to load the data:

- **Unpacking Data Files:** The training data is stored in five files, each containing 10,000 images. I created a helper function called unpickle to open each file and load the data inside.
- **Combining Training Files:** I combined the data from all five training files into one large set. This gave us a total of 50,000 images for training. I did the same for the labels, so each image has a label showing which class it belongs to.
- **Loading Test Data:** The test data is stored in a separate file. I loaded this file in the same way to get 10,000 test images and their labels.

B. *Data Preprocessing*

After loading the data, I prepared it for use with machine learning by following these steps:

- **Reshaping Images:** Each image was originally stored as a long row of numbers (a flat array). I reshaped these numbers back into the original 32x32 size with 3 color channels (Red, Green, and Blue).
- **Normalizing Pixel Values:** The pixel values in each image were originally between 0 and 255. To make the data easier to work with, I scaled these values to be between 0 and 1 by dividing each value by 255. This helps the model perform better and faster.

The final output of our data loading and preprocessing is:

- `x_train`: A set of 50,000 training images, each with shape (32, 32, 3) and pixel values between 0 and 1.
- `y_train`: A set of 50,000 labels for the training images.
- `x_test`: A set of 10,000 test images, also with shape (32, 32, 3) and pixel values between 0 and 1.
- `y_test`: A set of 10,000 labels for the test images.

This data is now ready to be used in our CNN and MLP algorithms as well as the k-NN algorithm, making it easy to classify each image based on its pixel values.

III. The k-Nearest Neighbor (k-NN) Algorithm

The k-Nearest Neighbor (k-NN) algorithm is a simple, method used for classification tasks. In classification, the algorithm works by identifying the ”k” closest data points (or neighbors) to a given input and predicting the class based on these neighbors.

The steps of the k-NN algorithm are as follows:

1) **Choosing the Number of Neighbors (k):** The user selects the number of nearest neighbors (k) that will be used to make the prediction. A small k value can make the model more sensitive to noise and a larger k value may make it too generalized.
2) **Calculating Distance:** For a new data point, the algorithm calculates the distance between this point and all

points in the training data. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance, with Euclidean distance being the most widely used in k-NN classification and the one I used in our implementations later on.

3) **Finding the k Nearest Neighbors:** After calculating the distances, the algorithm identifies the "k" closest data points in the training set.
4) **Assigning the Class Label:** The algorithm assigns the new data point to the class that appears most frequently among the k neighbors. For example, if $k = 5$ and three neighbors are in class A while two are in class B, the new point is classified as class A.

A. *Advantages and Disadvantages of k-NN*

Advantages:

- Simplicity: k-NN is easy to implement and understand.
- **No Training Phase:** Not complicated because the computation happens during prediction.
- **Flexibility:** Works well with various distance metrics (like Euclidean, Manhattan, etc) and can be adjusted for specific tasks by changing the value of k.

Disadvantages:

- **Computational Cost:** Since k-NN requires calculating distances for all points in the training set, it can be slow and computationally expensive with large datasets. Since this is a major disadvantage I used only 2000 training examples in the implementation without PCA (Principal Component Analysis). In our implementation with PCA I used the entire CIFAR-10 dataset (60000 32x32 colour images in 10 classes) since PCA can reduce its dimensionality while keeping as much variation as possible making the k-NN algorithm operate quicker.
- **Sensitive to Scale:** k-NN needs preprocessing (like normalization) for better performance.
- **Sensitive to Imbalanced Data:** If one class has many more examples than others, k-NN might favor that class in its predictions.

That is why k-NN gave us worse accuracy results overall than the Convolutional Neural Network (CNN), and Multi-Layer Perceptron (MLP). For this project though it provides a simple comparison to evaluate the effectiveness of more complex neural network models like CNN and MLP.

B. *Approach of the Task*

1. Custom Implementation of k-NN Without PCA

In this approach, I implemented k-NN from scratch, using Euclidean distance to measure similarity between points. Each image was flattened into a 1D vector, and I used a subset of 2000 samples for both training and testing to keep it manageable, as the algorithm was quite slow.

- **Distance Calculation**: The distance between each test image and all training images is calculated.
- **Classification**: Each test image is labeled based on the most common label among its k nearest neighbors.
- **Nearest Centroid**: Ialso implemented a centroid-based classifier, which assigns each test image to the nearest class centroid.
- **Results**:
  1) **k-NN with $k = 1$ Accuracy:** 0.244
  2) **k-NN with $k = 3$ Accuracy:** 0.255
  3) **Nearest Centroid Accuracy:** 0.271

2. Custom Implementation of k-NN with PCA

In this approach, Iused Principal Component Analysis (PCA) to reduce the data to 100 features, making the algorithm faster. Both k-NN and centroid classification were then applied to this simplified dataset.

- **Dimensionality Reduction**: PCA was used to reduce each image to 100 features.
- **k-NN and Centroid Classification**: Both k-NN and nearest centroid classifiers were used on the reduced data.
- **Results**:
  1) **k-NN with $k = 1$ Accuracy:** 0.3863
  2) **k-NN with $k = 3$ Accuracy:** 0.3893
  3) **Nearest Centroid Accuracy:** 0.2767

3. Using sklearn Library for k-NN and Centroid Classifier with PCA

In this approach, Iused the KNeighborsClassifier and NearestCentroid from sklearn. PCA was applied to reduce each image to 100 features, making it faster to process. The optimized classifiers from sklearn gave us an efficient and scalable implementation.

- **Optimized Implementation**: KNeighborsClassifier and NearestCentroid offer fast, built-in methods for k-NN and centroid classification.
- **Parameter Tuning**: sklearn makes it easy to adjust settings like the number of neighbors k.
- **Efficiency**: Using sklearn improved performance and sped up the calculations on large datasets.
- **Results**:
  1) **k-NN with $k = 1$ Accuracy:** 0.3856
  2) **k-NN with $k = 3$ Accuracy:** 0.3691
  3) **Nearest Centroid Accuracy:** 0.2767

Each approach offers a unique perspective, from manual implementation to more efficient, library-based methods suitable for practical use and are widely known.

IV. MLP Models Approaches

A Multilayer Perceptron (MLP) is a basic type of neural network used in deep learning. It is called a feedforward network because data moves in one direction, starting from the input layer, passing through hidden layers, and ending at the output layer. MLPs can learn to model complex patterns, making them useful for tasks like classifying data or predicting outcomes. In our case I have build 4 MLP models to make predictions in the CIFAR-10 dataset.

A. *All approaches and implementations*

*1) **Initial Approach: Single-Layer MLP for CIFAR-10**:*

*a) **Explanation of initial approach**:* To begin I implemented a simple MLP with a single hidden layer to classify images from the CIFAR-10 dataset. My aim was to evaluate the basic performance of an MLP architecture and identify its strengths and limitations before attempting some more complex models.

*b) **Model Architecture**:* The implemented MLP model had the following structure:

- **Input Layer:** The input layer flattened each 32x32x3 image into a vector of size 3072, making it suitable for processing by a fully connected layer.
- **Hidden Layer:** A single hidden layer with 128 neurons was used, applying the ReLU (Rectified Linear Unit) activation function to introduce non-linearity.
- **Dropout:** A dropout layer with a rate of 0.5 was added after the hidden layer to reduce overfitting by randomly deactivating 50% of the neurons during training.
- **Output Layer:** The output layer consisted of 10 neurons (one for each class), using the softmax activation function to compute probabilities for each class.

The model was compiled with the Adam optimizer and the categorical crossentropy loss function, which are commonly used for classification tasks. The training process involved 20 epochs with a batch size of 128.

*c) **Results**:* During training, I tracked the accuracy and loss for both the training and validation datasets. Figure 1 shows the model's accuracy and loss over the 20 epochs.

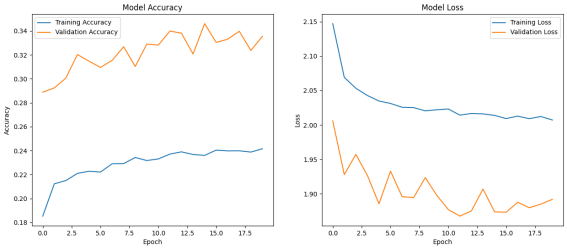

Fig. 1. Training and Validation Accuracy and Loss for Single-Layer MLP.

After training, the model achieved a test accuracy of around ***0.3355***

*d) **Sample Predictions and Observations**:* To further evaluate the model, I visualized a selection of correctly and incorrectly classified images from the test set. Figure 2 displays 7 correct and 7 incorrect predictions.

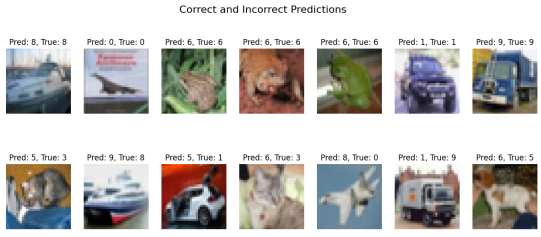

Fig. 2. Examples of Correct and Incorrect Predictions for Single-Layer MLP.

Some of my key observations include:

- The model performed well on simpler images with clear features but I think it might have struggled with some more complex images.
- The lack of multiple hidden layers reduces the ability to capture complex patterns.
- Overfitting might have happened to some extent by the dropout layer, but the model still seems to generalize well.

*2) **MLP with more layers for CIFAR-10**:*

*a) **Model Explanation**:* Thw

After trying a single-layer MLP, I built a more advanced model with two hidden layers to improve the performance. The goal was to help the network learn more complex patterns in the CIFAR-10 dataset and achieve better accuracy.

*b) **Model Architecture**:* The improved MLP model was designed with the following layers:

- **Input Layer:** The input layer flattened each 32x32x3 image into a vector of 3072 values.
- **First Hidden Layer:** This layer had 256 neurons and used the ReLU activation function to introduce non-linearity.
- **Dropout Layer:** A dropout rate of 0.3 was applied after the first hidden layer to reduce overfitting.
- **Second Hidden Layer:** This layer had 128 neurons and also used the ReLU activation function.
- **Dropout Layer:** Another dropout rate of 0.3 was applied after the second hidden layer for regularization.
- **Output Layer:** The output layer had 10 neurons, one for each class in CIFAR-10, and used the softmax activation function to calculate class probabilities.

The model was trained using the Adam optimizer and the categorical crossentropy loss function for 15 epochs with a batch size of 60.

*c) **Results**:* During training, I tracked the accuracy and loss for both the training and validation datasets. Figure 3 shows the accuracy and loss for the model over the 15 epochs.

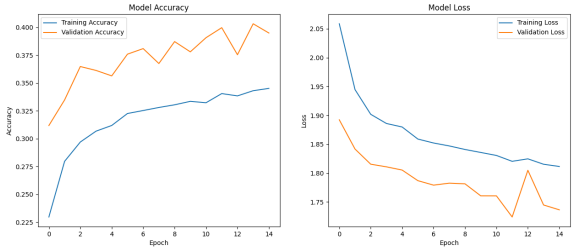

Fig. 3. Training and Validation Accuracy and Loss for Multi-Layer MLP.

The improved model performed better than the single-layer MLP. It achieved a test accuracy of around **0.4049**, which is a significant improvement.

*d) **Sample Predictions**:* To check how well the model classified the images, I visualized some examples. Figure 4 shows 7 correctly classified and 7 incorrectly classified images from the test set.

*e) **Observations**:* The additional hidden layer and dropout made the model better at understanding the CIFAR-10 images. Some key takeaways:

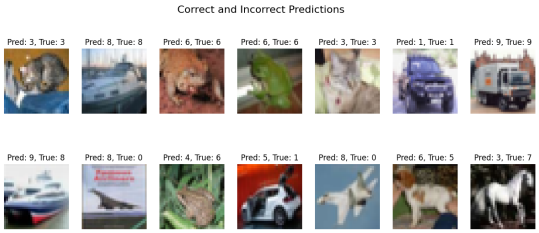

Fig. 4. Examples of Correct and Incorrect Predictions for Multi-Layer MLP.

- The model learned more complex features, which improved its ability to classify the images correctly.
- Dropout layers reduced overfitting, as the validation accuracy stayed close to the training accuracy.
- However, the model still had difficulty with challenging images, which might need more advanced methods like convolutional layers to handle.

This improved MLP showed better performance than the simpler model and provided valuable insights for further improvements.

*3) **Best Approach: Advanced MLP for CIFAR-10**:*

*a) **Explanation of Best Approach**:* Building on the improvements in the multi-layer MLP, I designed an even more advanced MLP with three hidden layers, batch normalization, and higher dropout rates. To be more specific lets explain some of the terms I used. To begin with I used batch normalization as staded above. Batch normalization is a technique used to make training faster and more stable. It normalizes the inputs of each layer by adjusting and scaling them, ensuring the data has a mean of 0 and a standard deviation of 1 during training. This helps the network converge quicker and reduces sensitivity to initialization. Dropout on the other hand is a regularization method that randomly turns off a percentage of neurons during training. This prevents the model from relying too much on specific neurons, reducing overfitting and improving generalization to new data. The goal of this approach was to maximize the model's ability to classify the CIFAR-10 images by increasing its complexity and adding better regularization techniques.

*b) **Model Architecture**:* The advanced MLP model was designed with the following layers:

- **Input Layer:** The input layer flattened each 32x32x3 image into a vector of 3072 values.
- **First Hidden Layer:** This layer had 512 neurons and used the ReLU activation function. Batch normalization was applied to stabilize learning, followed by a dropout layer with a rate of 0.4 to reduce overfitting.
- **Second Hidden Layer:** This layer had 256 neurons, again with ReLU activation, batch normalization, and a dropout rate of 0.4.
- **Third Hidden Layer:** This layer had 128 neurons, ReLU activation, batch normalization, and a smaller dropout rate of 0.3.
- **Output Layer:** The final layer had 10 neurons, one for each class in CIFAR-10, and used the softmax activation function to calculate class probabilities.

The model was trained using the Adam optimizer with a learning rate of 0.001 and the categorical crossentropy loss function for 30 epochs with a batch size of 60.

*c) **Results**:* The training and validation accuracy and loss were tracked throughout the 30 epochs. Figure 5 shows the accuracy and loss during training.

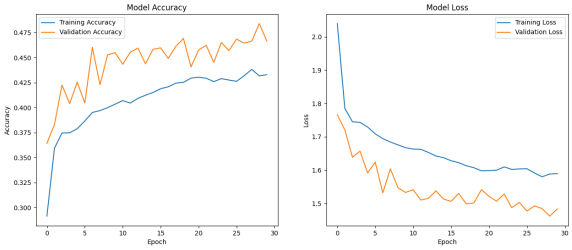

Fig. 5. Training and Validation Accuracy and Loss for Advanced Multi-Layer MLP.

The advanced model performed significantly better than the previous MLPs. It achieved a test accuracy of approximately **0.47439**, showing its ability to handle better the complexities of the CIFAR-10 dataset comparing it to the previous two models.

*d) **Sample Predictions**:* To further evaluate the model, I visualized some examples of correctly and incorrectly classified images from the test set. Figure 5 displays 7 correct and 7 incorrect predictions.

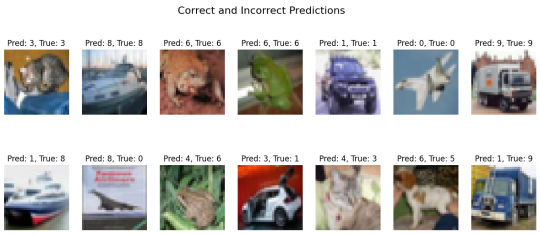

Fig. 6. Examples of Correct and Incorrect Predictions for Advanced Multi-Layer MLP.

*e) **Observations**:* The advanced MLP model achieved the best performance among all the approaches. Some of my takeaways include:

- The increased number of neurons and hidden layers allowed the model to capture more complex patterns in the data.
- Batch normalization stabilized training and improved convergence, while dropout effectively reduced overfitting.
- Despite the improvements, the model occasionally struggled with challenging images which means that it could be even better with some convolutional layers in the implementation, and that might even produce better results.

This final keras based MLP approach used fully connected architectures combined with techniques like batch normalization and dropout.

*4) **Final Approach: Custom attempt on building MLP for CIFAR-10**:*

*a) **Explanation of Custom Approach**:* In this code I manually tried to implement a Multilayer Perceptron from scratch using NumPy. Unlike the Keras implementations, this code has the following features and some disadvantages but was created for experimenting purpose even though it has advantages over custom adjustments. So some important comparisons are the following:

- **Control of the neural network implementation**:
  - The custom code gives you full control over details like how weights are initialized, the learning rate, and how dropout is applied.
  - Keras is easier to use but hides some of the details from the user.
- **Optimization**:
  - The custom code uses a simple method to update weights with a fixed learning rate.
  - Keras uses more advanced optimizers like Adam, which adjust the learning rate automatically.
- **Batch Normalization**:
  - Keras includes batch normalization to make training faster and more stable.
  - The custom implementation does not include this feature.
- **Performance**:
  - The custom code runs slower because NumPy doesn't use GPUs.
  - Keras is much faster because it can use GPUs for training.
- **Flexibility**:
  - The custom code lets you change everything, which is good for testing new ideas.
  - Keras is better for quickly building and testing models.
- **Visualization and Output**:
  - Both codes create similar output, like accuracy and loss graphs and sample predictions.
  - Keras does this automatically, but the custom code calculates it manually.

Below I will now demonstrate the code implementation:

- **Weight Initialization:** The model starts by assigning random values (called weights) to each connection in the network. These weights are initialized using a method called He initialization, which makes sure the values are neither too large nor too small. This helps the model learn faster and avoid problems like vanishing gradients (when updates become very small). The specific method used (**He Initialization**) is shown down below:

$$W \sim \mathcal{N}(0, \frac{2}{n_{\text{in}}})$$

  - W: The weight matrix being initialized.
  - $\mathcal{N}(0, \frac{2}{n_{\text{in}}})$: A normal distribution with a mean of 0 and a variance of $\frac{2}{n_{\text{in}}}$.
  - n_{in}: The number of input neurons (or the size of the previous layer).

  This method ensures that the variance of activations remains stable across layers, which is especially useful for networks using the **ReLU activation function**:
- **My Layers and Architecture:** The MLP has the following structure:
  - **Input Layer:** This layer takes the CIFAR-10 images, which are 32x32 pixels with 3 color channels. These images are flattened into a single row of 3072 numbers (32x32x3 = 3072 pixels).
  - **Hidden Layers:** There are two hidden layers with 256 and 128 units (neurons). These layers use an activation function called ReLU (Rectified Linear Unit), which helps the network learn complex patterns by introducing non-linearity.
  - **Output Layer:** The last layer has 10 units, one for each class in CIFAR-10. It uses the softmax activation function, which turns the output into probabilities for each class.
- **Forward Pass:** The forward pass is the process where the input data is passed through the network, layer by layer, to produce predictions. A method called `dropout` is used during training to randomly turn off some neurons, which helps prevent overfitting (where the model performs well on training data but poorly on new data).
- **Backward Pass:** After the forward pass, the model compares its predictions with the true labels to calculate the error (loss). The backward pass then adjusts the weights in the network to minimize this error. This process, called backpropagation, uses the gradient descent method to update the weights step by step.
- **Loss Function:** The model uses a loss function called categorical crossentropy. This measures how different the predicted probabilities are from the actual labels. A smaller loss means the model is making better predictions.
- **My Training Loop:** The training loop is where the model learns:
  - The data is mixed up randomly to keep it fair, and the model works on it in small parts called batches.
  - For each batch, the model makes predictions (forward pass), calculates the loss, and adjusts the weights (backward pass).
  - The training and validation performance (accuracy and loss) are recorded at every step.

*b) **Results and Visualizations**:* The results of training the custom MLP are shown in Figure 7. This figure displays how the accuracy and loss changed over 50 training epochs (iterations over the full dataset).

After training, the model achieved a test accuracy of about ***0.2543*** which means that the experimental model is still seriously struggling compared to the Keras ones as for its predictions. This is mostly due to reasons I have stated above like optimization, performance etc. To show you my model's performance, Figure 8 shows examples of images it classified correctly and incorrectly.

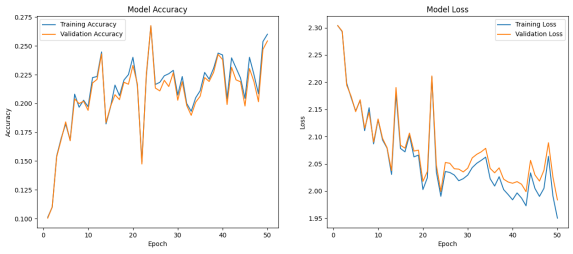

Fig. 7. Training and Validation Accuracy and Loss for Custom MLP.

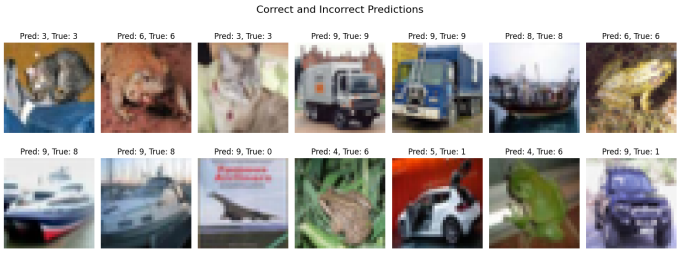

Fig. 8. Examples of Correct and Incorrect Predictions for Custom MLP.

B. *Summary about MLP implementations*

While MLP implementations of neural networks can be good and could most probably have better accuracy than the KNN algorithm there are a few other ways to achieve even better accuracy. Later in my report paper I will show you some more advanced yet more commonly used neural network architectures for predictions in image datasets.

V. CNN Model Approaches

A Convolutional Neural Network (CNN) is a type of deep learning model that is especially effective for processing image data. Unlike MLPs, CNNs use convolutional layers to automatically learn spatial patterns in images, such as edges, textures, and objects. These networks are designed to handle the hierarchical structure of images, making them more efficient and accurate for image-related tasks. For this project, I developed 2 CNN models to classify images in the CIFAR-10 dataset and improve prediction accuracy. To be more specific I implemented one of them with data augmentations and one without and I will showcase and explain the differences later on in this section.

A. *What is a CNN in a few words*

1) The input image is passed through convolutional layers, which extract features such as edges and textures.
2) The pooling layers reduce the size of the feature maps, making the model faster and more robust.
3) The extracted features are passed to fully connected layers for classification.
4) The output layer provides probabilities for each class, identifying the category of the input image.

You can see an example of a CNN architecture in Figure 9.

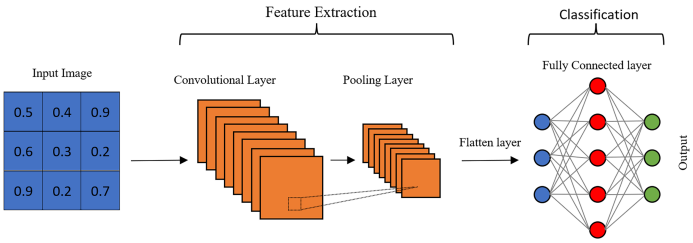

Fig. 9. Basic CNN Architecture. The input image is passed through convolutional and pooling layers for feature extraction, followed by fully connected layers for classification.

B. *My CNN implementations with explanation*

1) Initial Approach: CNN for CIFAR-10:

a) Explanation of the CNN Architecture: For my first CNN model, I again as previously with MLP created a network to classify images from the CIFAR-10 dataset. Below, I explain the structure of the CNN and how each part works:

- **Convolutional Layers:**
  - These layers use small grids of numbers called **filters** to scan the image. Filters detect patterns like edges or textures.
  - Each filter creates an output called a **feature map**, which shows where the pattern appears in the image.
  - The model starts with two convolutional layers, each using 32 filters of size 3×3. The **ReLU activation function** is applied after each convolution to keep only positive values. This helps the model learn better.
- **Batch Normalization:**
  - This step adjusts the output of the convolutional layer so that it has a mean of 0 and a standard deviation of 1.
  - This makes the training faster and more stable.
- **Pooling and Dropout:**
  - A **MaxPooling layer** reduces the size of the feature map by taking the largest value in small regions (e.g., 2×2). This keeps important information while making the data smaller and faster to process.
  - **Dropout layers** randomly turn off some neurons (connections) during training to prevent overfitting. This ensures the model doesn't rely too much on specific features and generalizes better to new data.
- **Deeper Convolutional Layers:**
  - Two more convolutional layers, each with 64 filters of size 3×3, are added to detect more detailed patterns in the image.
  - Batch normalization and MaxPooling are also applied to these layers, along with another dropout layer for regularization.
- **Fully Connected Layers:**
  - After the convolutional layers, the feature maps are **flattened** into a single long row of numbers.
  - These numbers are passed through a **dense layer** with 256 neurons. A ReLU activation function is used again to learn patterns from the features.

- Another dropout layer with a 50% rate is applied before the final layer to further reduce overfitting.
- The final layer has 10 neurons, one for each CIFAR-10 class, and uses the **softmax activation function** to produce probabilities for each class.

*b) **Training Process**:* The model was trained using the following settings:

- **Optimizer:** The **Adam optimizer** was used to adjust the weights during training. It is faster and more efficient than traditional methods like simple gradient descent.
- **Loss Function:** The **categorical crossentropy** loss function measures how different the predicted probabilities are from the true labels. The goal is to minimize this loss during training.
- **Batch Size:** The data was processed in small groups of 32 images at a time (called batches) to make training efficient.
- **Epochs:** The model was trained for 60 complete passes through the dataset (called epochs).

*c) **Results and Visualizations**:* Figure 10 shows how the model's accuracy and loss improved during training over 60 epochs. The model achieved a final test accuracy of approximately ***0.8194***. This demonstrates the model's ability to correctly classify most of the images.

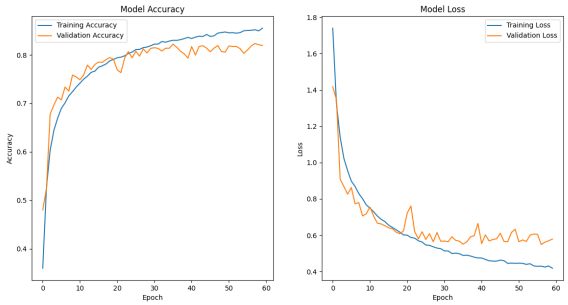

Fig. 10. Training and Validation Accuracy and Loss for the CNN Model.

To further evaluate the model, I visualized some examples of correctly and incorrectly classified images from the test set. Figure 11 shows 7 correct and 7 incorrect predictions.

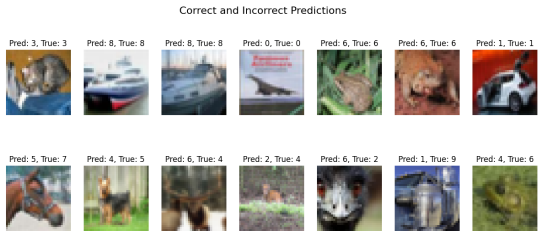

Fig. 11. Examples of Correct and Incorrect Predictions for the CNN Model.

*d) **Observations**:* Some key takeaways from this CNN implementation are:

- **Feature Extraction:** The convolutional layers effectively learned to detect patterns, which improved classification accuracy.
- **Regularization:** Dropout layers and batch normalization helped prevent overfitting and ensured the model worked well on new images.
- **Performance:** While the model performed well, adding more layers or data augmentation could further improve accuracy on complex images.

*2) **Enhanced Approach: CNN with Data Augmentation for CIFAR-10**:*

*a) **Explanation of the CNN Architecture**:* In this model, I extended the previous CNN architecture with a few enhancements:

- Two convolutional layers with 32 filters followed by another two layers with 64 filters were used for feature extraction, as in the earlier model.
- A larger dense layer with 512 neurons was added before the output layer for better feature learning.
- Higher dropout rates (up to 60%) were used to reduce overfitting.

*b) **Data Augmentation**:* A key difference in this model was the use of **data augmentation**. This technique creates variations of the training images to make the model more robust and improve generalization. The augmentations applied include:

- **Rotation:** Rotates the image randomly up to 10 degrees.
- **Shifting:** Shifts the image up or down, left or right by up to 10% of its dimensions.
- **Shearing:** Applies a random shearing transformation.
- **Zooming:** Zooms into the image by up to 10%.
- **Horizontal Flipping:** Flips the image horizontally.

These transformations helped the model learn to classify images under various conditions, improving its performance on unseen data.

*c) **Training Process**:* The model was trained with the following setup:

- **Data Generator:** A data generator applied augmentations during training, creating diverse inputs for the model.
- **Optimizer:** The RMSprop optimizer with a learning rate of 0.001 and a decay rate of 1×10^{-5}.
- **Batch Size and Epochs:** A batch size of 128 and 100 training epochs were used.

*d) **Results and Visualizations**:* Figure 12 shows the training and validation accuracy and loss over 100 epochs. The model achieved a test accuracy of approximately ***0.7916***. Figure 13 shows some examples of correctly and incorrectly classified images.

*e) **Observations**:* Key differences and observations in this approach:

- **Impact of Data Augmentation:** Augmentations made the model more robust to variations in input images, improving its generalization ability.
- **Lower Accuracy:** Compared to the previous CNN, this model did not perform as well as the non-augmented one.
- **Possible Risk of Overfitting:** Augmentation and higher dropout rates effectively could have portentially risen

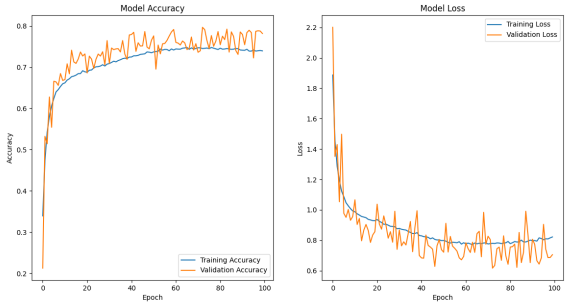

Fig. 12. Training and Validation Accuracy and Loss for the CNN with Data Augmentation.

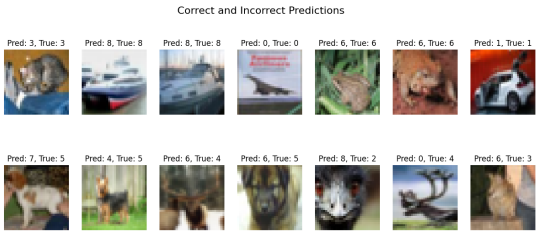

Fig. 13. Examples of Correct and Incorrect Predictions for the Augmented CNN Model.

the risk of overfitting due to the higher complexity. We can also see that there are some minor fluctuations in validation metrics which are expected due to the higher data diversity.

- **Model converged earlier:** It didn't seem to need all 100 epochs to converge.

VI. References Section

References

[1] https://www.ibm.com/topics/knn
[2] https://www.geeksforgeeks.org/introduction-convolution-neural-network
[3] https://keras.io/api/layers
[4] https://elearning.auth.gr/mod/resource/view.php?id=89261
[5] https://keras.io/api/datasets/cifar10/
[6] https://www.cs.toronto.edu/ kriz/cifar.html