Image Classification with Convolutional Neural Network (CNN)

I2 : Software Engineering

Léanna Suissa
Mathis Lair

Teacher : Y. Ait El Mahjoub

Introduction :

Throughout this course, we explored various foundational concepts in machine learning. We began by studying the Gaussian Discriminant Analysis (GDA) algorithm, advanced through perceptrons, and concluded with models such as Multi-Layer Perceptrons (MLPs) and Deep Neural Networks (DNNs). These models represent some of the most powerful techniques in machine learning today, enabling remarkable achievements in various applications.

In this report, we focus on a specific subtype of DNN known as the Convolutional Neural Network (CNN), widely used for image recognition. For our project, we applied CNNs to the Cifar 10 dataset, a collection of encrypted images designed to test model accuracy in recognizing patterns under complex transformations. CNNs excel at tasks like these and hold significant potential in fields like medicine, where they may one day play a crucial role in diagnostics and treatment planning.

prerequisites and informations :
For our project, we used several key libraries:

1. **NumPy**: Essential for handling arrays and matrix operations, allowing efficient data processing and manipulation.
2. **TensorFlow**: A machine learning framework used to build and train our Convolutional Neural Network (CNN) model.
3. **Matplotlib**: A plotting library that enabled us to visualize model performance, such as accuracy and loss, over time.
4. **Scikit-learn**: A powerful library for machine learning in Python, Scikit-learn offers tools for classification, regression, clustering, and model evaluation, making it easy to implement various machine learning algorithms.

to download them if it’s not already done please enter these command lines :

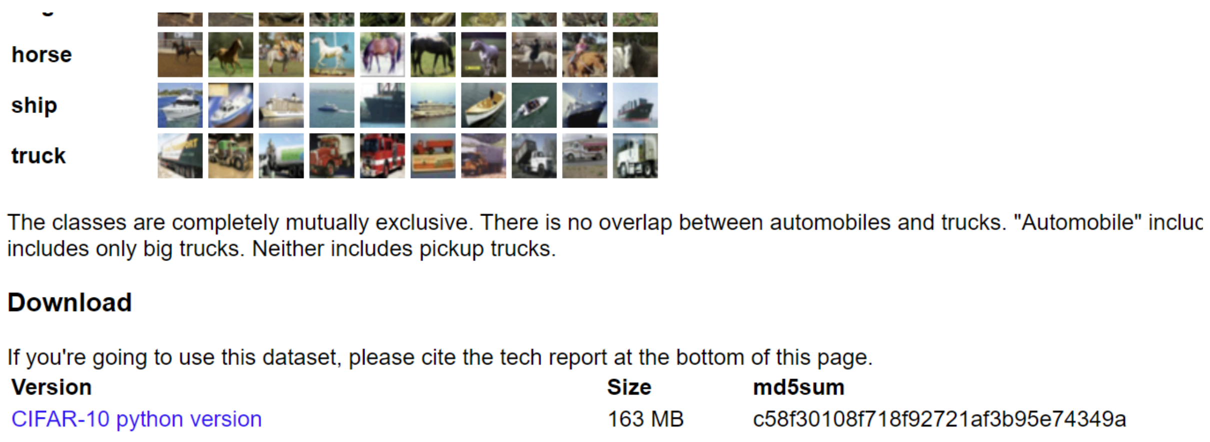
```
pip install numpy
pip install tensorflow
pip install matplotlib
```

We used GitHub as a collaborative platform to share our project code. We ensured that we could effectively track changes and access the latest updates, fostering seamless collaboration throughout the project.

Part 1 : Data Understanding and Preparation

Task 1 : Load the CIFAR-10 Dataset

1. <u>First, we go on the website given and load the python version of the CIFAR-10</u>



The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" incluc
includes only big trucks. Neither includes pickup trucks.

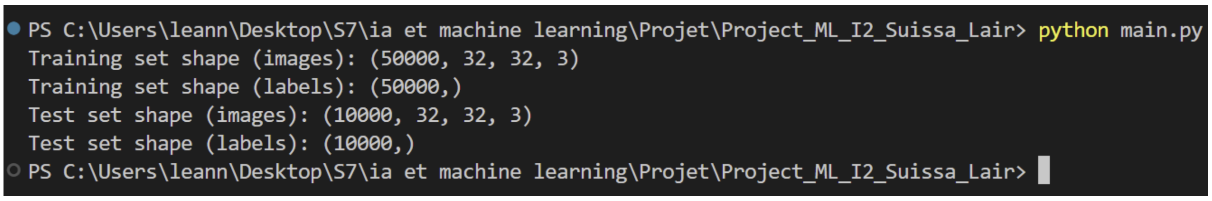
Download

If you're going to use this dataset, please cite the tech report at the bottom of this page.

Version	Size	md5sum
CIFAR-10 python version	163 MB	c58f30108f718f92721af3b95e74349a

We then download the latest version as you can see up here. We also created a file to store the project and put the dataset in it.

<u>2. Display the set and the training size :</u>



```
PS C:\Users\leann\Desktop\S7\ia et machine learning\Projet\Project_ML_I2_Suissa_Lair> python main.py
Training set shape (images): (50000, 32, 32, 3)
Training set shape (labels): (50000,)
Test set shape (images): (10000, 32, 32, 3)
Test set shape (labels): (10000,)
PS C:\Users\leann\Desktop\S7\ia et machine learning\Projet\Project_ML_I2_Suissa_Lair>
```

We wrote a small function that you can find in our project to display the shape and size of the labels and images. The first parameter specifies the quantity, which in this case is 50,000 images, each of size 32x32 pixels with 3 RGB channels for the training set. Each image has one corresponding label. For the test set, there are 10,000 images, also sized at 32x32 pixels with 3 RGB channels, along with one label for each image.

<u>3. How many training and test samples are there in the CIFAR-10 dataset?</u>

The CIFAR-10 dataset consists of 60,000 32x32 color images divided into 10 classes, with 6,000 images per class.
It includes 50,000 training images and 10,000 test images.

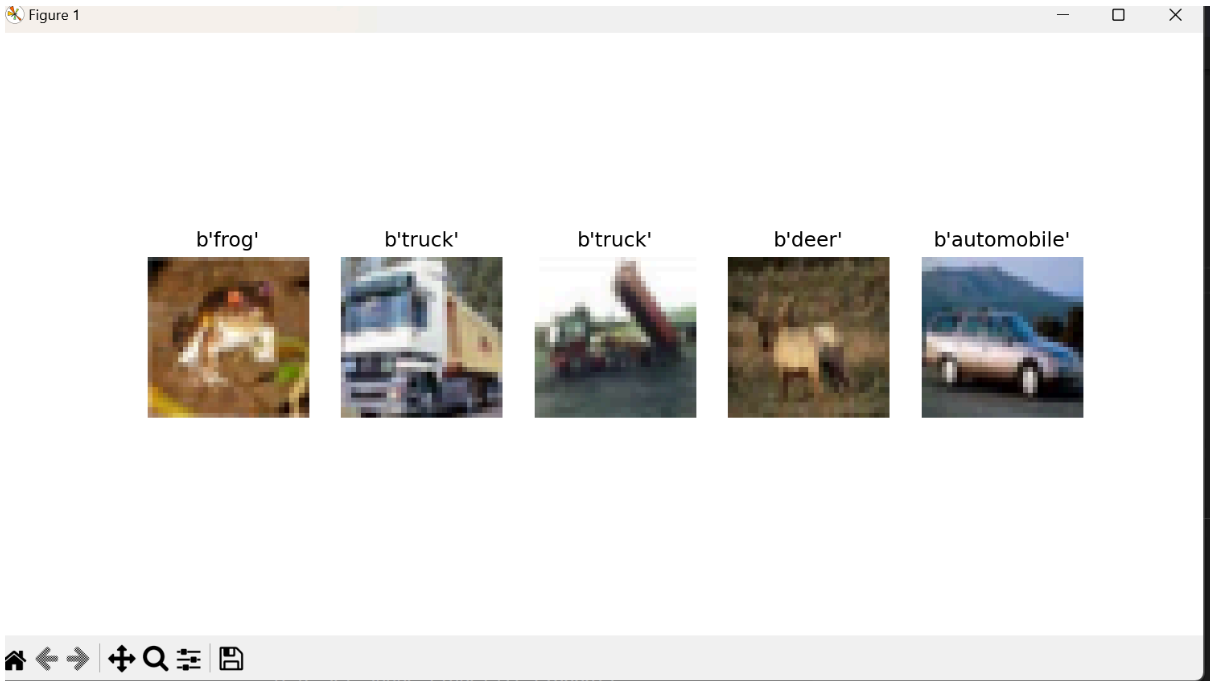
4. What is the shape of each image? how many color channels are there?

As shown in our results, the images are 32x32 pixels and have 3 color channels: red, green, and blue (RGB).

Task 2: Visualize the Dataset

1. Plot a few random images from the dataset to understand the types of images it contains.

   To plot a few random images, we decided to display the first 5 images :



However, we designed the function to allow the display of as many images as we want. You can test it in our code. The images are photographs that could have been taken by anyone, clearly featuring a single element in an environment.

2. Display the corresponding labels for the images. What type of data do the labels represent? Are they categorical or numerical ?

The labels correspond to images representing one of the following categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each label indicates the specific category of the object depicted in the image.

It represents categorical data because we established clear categories, such as automobile, truck, and frog, instead of seeking a continuous value. Categorical data refers to variables that can take on one of a limited, fixed number of possible values, often representing distinct groups or classifications. In this context, each image belongs to a specific category, allowing us to train our model to recognize and differentiate between these classes effectively.

3. What do the images look like ? Are they easy or difficult to classify based on human vision ?

The images are blurry but still recognizable to the human eye. When functioning well, the main elements remain identifiable, although the blurriness can make it somewhat challenging at times. Despite this, the objects in the images are generally easy to recognize.

Task 3 : Normalize the Data

1. Normalize the pixel values of the images from the range [0, 255] to [0, 1].

As you'll see in my project, I've included a commented section where I normalized the pixel values from the range [0, 255] to [0, 1]. Feel free to check it out for reference.

2. Explain why normalization is important in Neural Networks models.

Normalization is essential in neural networks because it ensures that input values are on a similar scale, which helps the model learn more efficiently and reduces the likelihood of gradient-related issues. By transforming data to a range like [0, 1], we prevent large values from dominating smaller ones, improve convergence speed, and make the training process more stable. Additionally, normalization often leads to better model performance and generalization by allowing neural network weights to update more consistently across layers.

3. What are the benefits of normalizing the pixel values in an image dataset ?

Normalizing pixel values in an image dataset has several benefits: it helps the model learn better by putting all values on a similar scale, speeds up convergence by stabilizing gradients, reduces bias by making images more consistent, and simplifies preprocessing since padding and other adjustments are minimized.

4. How does normalization impact the performance of neural networks ?

Normalization has a positive impact on neural network performances by improving training stability, speeding up convergence, and reducing the risk of overfitting. With normalized inputs, neural networks can learn patterns more effectively, leading to higher accuracy and better generalization on new data.

Task 4 : Apply One-Hot Encoding to the Labels

1. Convert the labels into one-hot encoded vectors.

To see how I converted the labels into one-hot encoded vectors, check the code where I've included comments.

2. Why is it necessary to apply one-hot encoding to the class labels in a classification problem ?

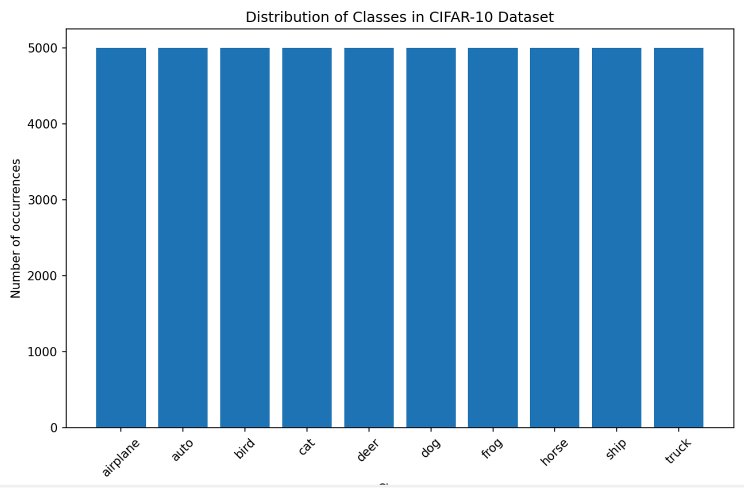
Applying one-hot encoding to class labels is necessary in classification because it lets the model understand each class as a unique category instead of a ranked value. Without it, the model might interpret classes as having an order, which could mess up training. One-hot encoding makes sure each class is treated equally, so the model learns to distinguish between them properly.

3. What does one-hot encoding represent, and how does it help in training a neural network ?

One-hot encoding represents each class label as a binary vector, where only one element is "1" (indicating the class) and the rest are "0". This helps in training a neural network by clearly separating classes so the model doesn't confuse them as being related. It forces the network to focus on the specific features of each class, which makes learning more accurate and effective.For example, if the class "dog" is the first class, the vector will look like (1,0,0,0,0,0,0,0,0,0).

Task 5 :Explore The Class Distribution

1. Plot a bar chart to show the distribution of classes.



2. Are the classes balanced in the CIFAR-10 dataset, or are some classes over/underrepresented ?

The classes in the CIFAR-10 dataset are balanced, as each class contains 5,000 images. This equal distribution helps ensure that the model has enough data to learn from each category without bias toward any particular class.

3. Why is it important to analyze the class distribution before training a model ?

Analyzing the class distribution before training a model is important because it helps identify any imbalances that could affect performance. If some classes are overrepresented or underrepresented, the model may become biased, leading to poor predictions for the less common classes. Understanding the distribution allows for appropriate adjustments, like data augmentation or resampling, to ensure the model learns effectively from all classes.

2.2 Data Modelling

1. Reading the documentation

To develop the algorithm, I read the provided documentation. This was essential for understanding the requirements and guidelines for implementing the model correctly. on the CIFAR-10 website i was able to get the function “unpickle” that will be used in our algorithm.

2. Adapting LeNet-5 for RGB Images

LeNet-5 is a convolutional neural network that works by taking in 32x32 grayscale images. It starts with convolutional layers that apply filters to extract features from the images, followed by subsampling layers that reduce the size while keeping important information. After several of these layers, the network flattens the features and passes them through fully connected layers to make predictions about the class of the input image. This structure allows LeNet-5 to efficiently learn and classify images by reducing the number of parameters and improving its ability to generalize. LeNet-5 was originally designed for grayscale images, so adjustments were necessary to enable it to recognize our RGB images effectively.

3. Implementing the VGG Architecture

As specified in the guidelines, we employed the VGG architecture. The VGG-like architecture consists of several layers designed to extract features from images. Each VGG block contains:

- Convolutional Layers: 2 layers with 32 filters and a kernel size of (3, 3).
- MaxPooling Layer: A kernel size of (2, 2).

Following our class discussions and video demonstrations, we constructed three layers as instructed:

- First Block: Learns simple patterns like edges and textures.
- Second Block: Builds on the first block's features to recognize shapes and more complex structures.
- Third Block: Captures high-level representations, enabling the model to recognize specific objects or patterns in the images.

In the main file, we call the `compile` function, which processes the VGG blocks through the flattening and dense layers. The flattening layer converts the 3D output into a 1D array. The dense layer was configured with 128 neurons in the output layer, as specified in the guidelines. We applied the softmax and ReLU activation functions.

Finally, we conducted tests to compare performance and effectiveness.

We trained the model on the dataset once for each run to maintain the integrity of our results. This approach was chosen to prevent any potential biases that could arise from multiple training sessions, which might skew the evaluation of the model's performance. By conducting a single training session for each evaluation, we aimed to ensure that our results accurately reflect the model's ability to generalize to the dataset. Below, you can see the results of this training process.

- LeNet-5: This model achieved a maximum accuracy of 40%, which was consistent across both training and validation sets. Although this accuracy is relatively low, the consistency suggests that the model is stable. The loss values were acceptable, indicating that there is still room for improvement through further tuning or additional training data.

- VGG 1: The model performed exceptionally well on the training set, reaching an impressive 95% accuracy. However, the maximum accuracy on the validation set was only 65%, suggesting that the model may be overfitting to the training data. This indicates that while the model has learned the training examples well, it struggles to generalize to unseen data. We might be facing overfitting here

- VGG 2: This model exhibited performance similar to VGG 1 but showed better results on the validation set, with an accuracy that is more aligned with the training results. This improvement suggests that VGG 2 has better generalization capabilities, potentially due to architectural tweaks or different hyperparameters.

- VGG 3: This model outperformed all the others, achieving up to 70% accuracy with just one training session. This significant improvement indicates that VGG 3 has a strong learning capability and can better capture the underlying patterns in the dataset, leading to enhanced performance on both the training and validation sets.

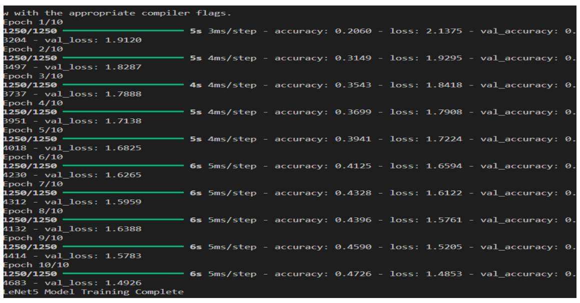
We used 3 optimizer on all algo to compare the results :

SAG, which is the most basic optimizer, where the model’s weights are updated by a fraction of the gradient of the loss function with respect to each parameter. This fraction, called the learning rate, is crucial for tuning the speed and accuracy of convergence.

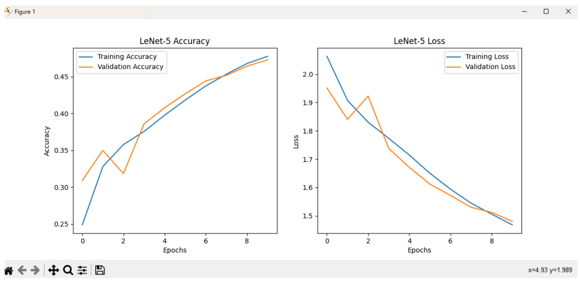
Adams, which is one of the most popular optimizers. It uses first and second moments of the gradients (the mean and uncentered variance) to adaptively adjust the learning rate for each parameter.

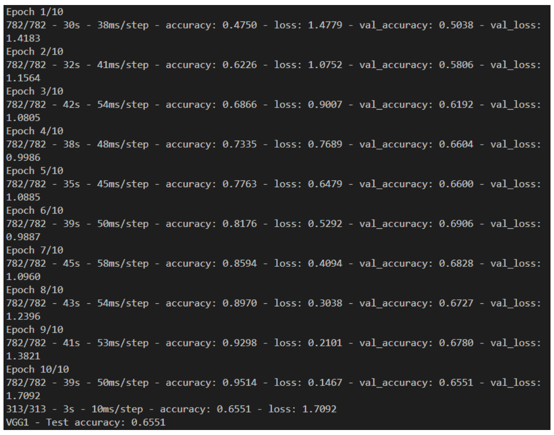
RMS is another adaptive learning rate optimizer that shares similarities with Adam.

Without Optimizer

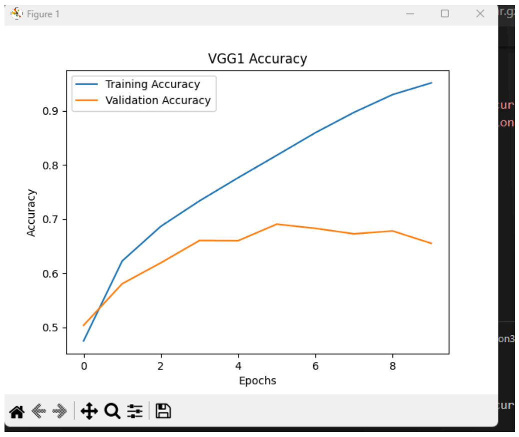


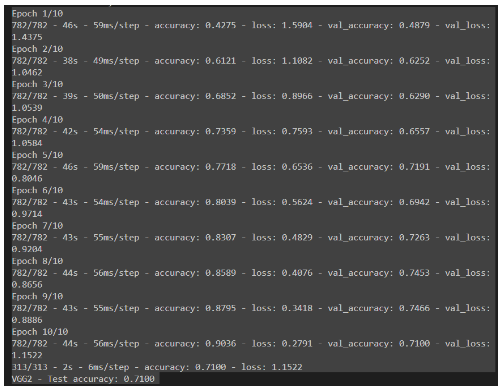
```
w with the appropriate compiler flags.
Epoch 1/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.2060 - loss: 2.1375 - val_accuracy: 0.
3204 - val_loss: 1.9120
Epoch 2/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.3149 - loss: 1.9295 - val_accuracy: 0.
3497 - val_loss: 1.8287
Epoch 3/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.3543 - loss: 1.8418 - val_accuracy: 0.
3737 - val_loss: 1.7888
Epoch 4/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.3699 - loss: 1.7908 - val_accuracy: 0.
3951 - val_loss: 1.7138
Epoch 5/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.3941 - loss: 1.7224 - val_accuracy: 0.
4018 - val_loss: 1.6825
Epoch 6/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.4125 - loss: 1.6594 - val_accuracy: 0.
4230 - val_loss: 1.6265
Epoch 7/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.4328 - loss: 1.6122 - val_accuracy: 0.
4312 - val_loss: 1.5959
Epoch 8/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.4396 - loss: 1.5761 - val_accuracy: 0.
4132 - val_loss: 1.6388
Epoch 9/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.4590 - loss: 1.5205 - val_accuracy: 0.
4414 - val_loss: 1.5783
Epoch 10/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.4726 - loss: 1.4853 - val_accuracy: 0.
4683 - val_loss: 1.4926
LeNet5 Model Training Complete
```



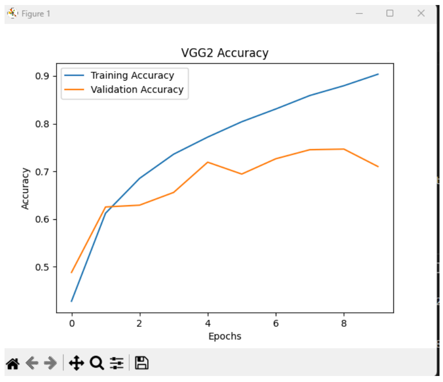


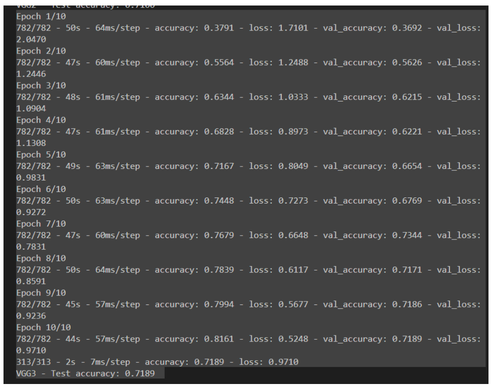
```
Epoch 1/10
782/782 - 30s - 38ms/step - accuracy: 0.4750 - loss: 1.4779 - val_accuracy: 0.5038 - val_loss:
1.4183
Epoch 2/10
782/782 - 32s - 41ms/step - accuracy: 0.6226 - loss: 1.0752 - val_accuracy: 0.5806 - val_loss:
1.1564
Epoch 3/10
782/782 - 42s - 54ms/step - accuracy: 0.6866 - loss: 0.9007 - val_accuracy: 0.6192 - val_loss:
1.0805
Epoch 4/10
782/782 - 38s - 48ms/step - accuracy: 0.7335 - loss: 0.7689 - val_accuracy: 0.6604 - val_loss:
0.9986
Epoch 5/10
782/782 - 35s - 45ms/step - accuracy: 0.7763 - loss: 0.6479 - val_accuracy: 0.6600 - val_loss:
1.0885
Epoch 6/10
782/782 - 39s - 50ms/step - accuracy: 0.8176 - loss: 0.5292 - val_accuracy: 0.6906 - val_loss:
0.9887
Epoch 7/10
782/782 - 45s - 58ms/step - accuracy: 0.8594 - loss: 0.4094 - val_accuracy: 0.6828 - val_loss:
1.0960
Epoch 8/10
782/782 - 43s - 54ms/step - accuracy: 0.8970 - loss: 0.3038 - val_accuracy: 0.6727 - val_loss:
1.2396
Epoch 9/10
782/782 - 41s - 53ms/step - accuracy: 0.9298 - loss: 0.2101 - val_accuracy: 0.6780 - val_loss:
1.3821
Epoch 10/10
782/782 - 39s - 50ms/step - accuracy: 0.9514 - loss: 0.1467 - val_accuracy: 0.6551 - val_loss:
1.7092
313/313 - 3s - 10ms/step - accuracy: 0.6551 - loss: 1.7092
VGG1 - Test accuracy: 0.6551
```



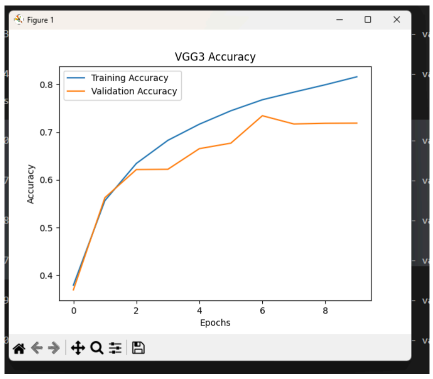


```
Epoch 1/10
782/782 - 46s - 59ms/step - accuracy: 0.4275 - loss: 1.5904 - val_accuracy: 0.4879 - val_loss:
1.4375
Epoch 2/10
782/782 - 38s - 49ms/step - accuracy: 0.6121 - loss: 1.1082 - val_accuracy: 0.6252 - val_loss:
1.0462
Epoch 3/10
782/782 - 39s - 50ms/step - accuracy: 0.6852 - loss: 0.8966 - val_accuracy: 0.6290 - val_loss:
1.0539
Epoch 4/10
782/782 - 42s - 54ms/step - accuracy: 0.7359 - loss: 0.7593 - val_accuracy: 0.6557 - val_loss:
1.0584
Epoch 5/10
782/782 - 46s - 59ms/step - accuracy: 0.7718 - loss: 0.6536 - val_accuracy: 0.7191 - val_loss:
0.8046
Epoch 6/10
782/782 - 43s - 54ms/step - accuracy: 0.8039 - loss: 0.5624 - val_accuracy: 0.6942 - val_loss:
0.9714
Epoch 7/10
782/782 - 43s - 55ms/step - accuracy: 0.8307 - loss: 0.4829 - val_accuracy: 0.7263 - val_loss:
0.9204
Epoch 8/10
782/782 - 44s - 56ms/step - accuracy: 0.8589 - loss: 0.4076 - val_accuracy: 0.7453 - val_loss:
0.8656
Epoch 9/10
782/782 - 43s - 55ms/step - accuracy: 0.8795 - loss: 0.3418 - val_accuracy: 0.7466 - val_loss:
0.8886
Epoch 10/10
782/782 - 44s - 56ms/step - accuracy: 0.9036 - loss: 0.2791 - val_accuracy: 0.7100 - val_loss:
1.1522
313/313 - 2s - 6ms/step - accuracy: 0.7100 - loss: 1.1522
VGG2 - Test accuracy: 0.7100
```

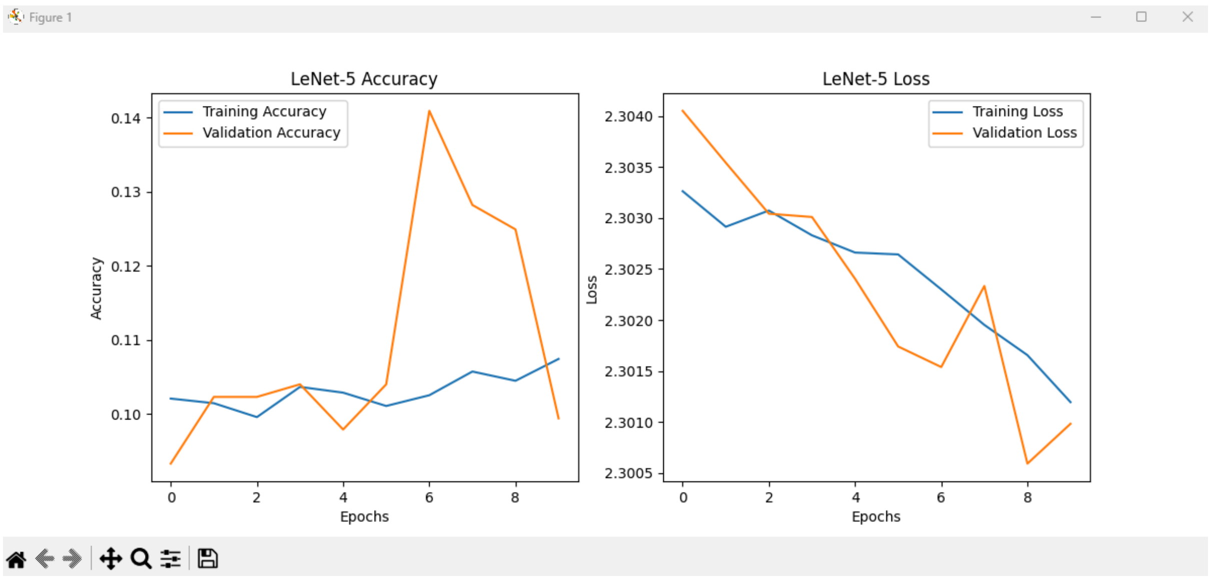


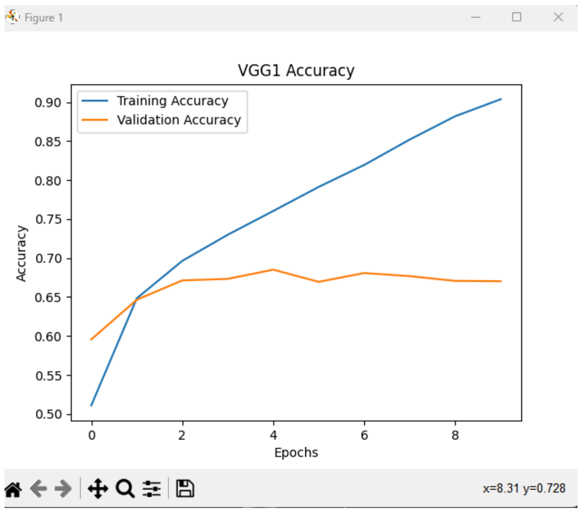


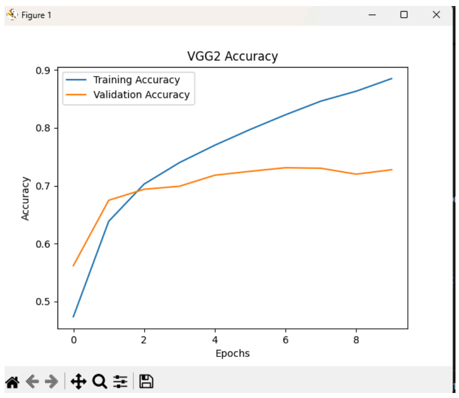
```
Epoch 1/10
782/782 - 50s - 64ms/step - accuracy: 0.3791 - loss: 1.7101 - val_accuracy: 0.3692 - val_loss:
2.0470
Epoch 2/10
782/782 - 47s - 60ms/step - accuracy: 0.5564 - loss: 1.2488 - val_accuracy: 0.5626 - val_loss:
1.2446
Epoch 3/10
782/782 - 48s - 61ms/step - accuracy: 0.6344 - loss: 1.0333 - val_accuracy: 0.6215 - val_loss:
1.0904
Epoch 4/10
782/782 - 47s - 61ms/step - accuracy: 0.6828 - loss: 0.8973 - val_accuracy: 0.6221 - val_loss:
1.1308
Epoch 5/10
782/782 - 49s - 63ms/step - accuracy: 0.7167 - loss: 0.8049 - val_accuracy: 0.6654 - val_loss:
0.9831
Epoch 6/10
782/782 - 50s - 63ms/step - accuracy: 0.7448 - loss: 0.7273 - val_accuracy: 0.6769 - val_loss:
0.9272
Epoch 7/10
782/782 - 47s - 60ms/step - accuracy: 0.7679 - loss: 0.6648 - val_accuracy: 0.7344 - val_loss:
0.7831
Epoch 8/10
782/782 - 50s - 64ms/step - accuracy: 0.7839 - loss: 0.6117 - val_accuracy: 0.7171 - val_loss:
0.8591
Epoch 9/10
782/782 - 45s - 57ms/step - accuracy: 0.7994 - loss: 0.5677 - val_accuracy: 0.7186 - val_loss:
0.9236
Epoch 10/10
782/782 - 44s - 57ms/step - accuracy: 0.8161 - loss: 0.5248 - val_accuracy: 0.7189 - val_loss:
0.9710
313/313 - 2s - 7ms/step - accuracy: 0.7189 - loss: 0.9710
VGG3 - Test accuracy: 0.7189
```

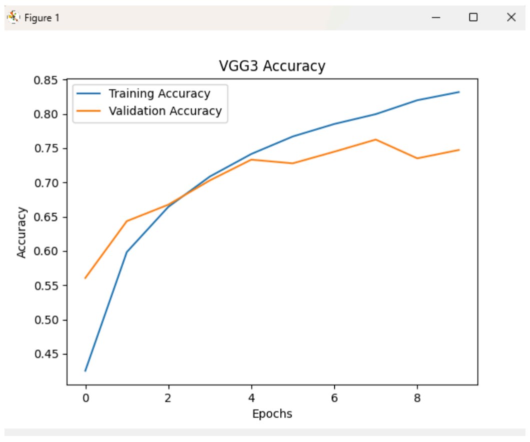


First Optimizer SAG

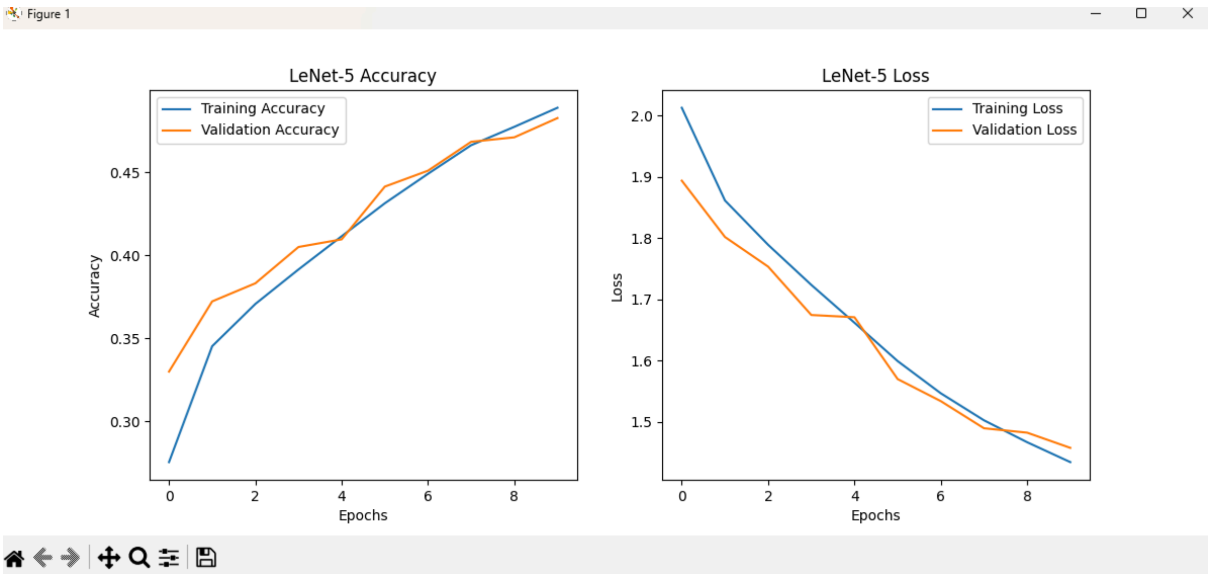




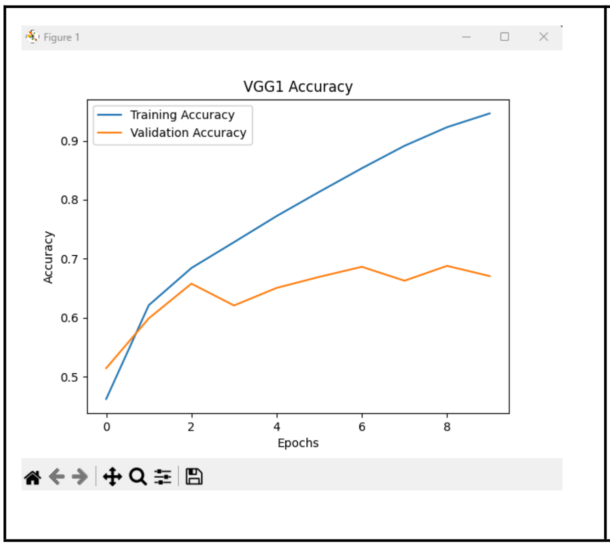


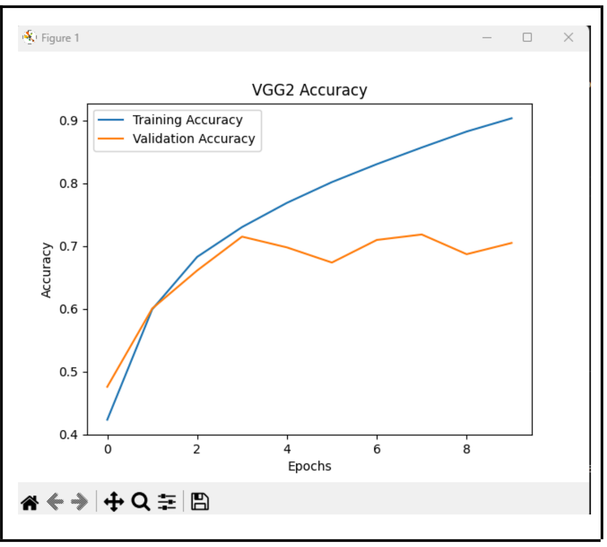


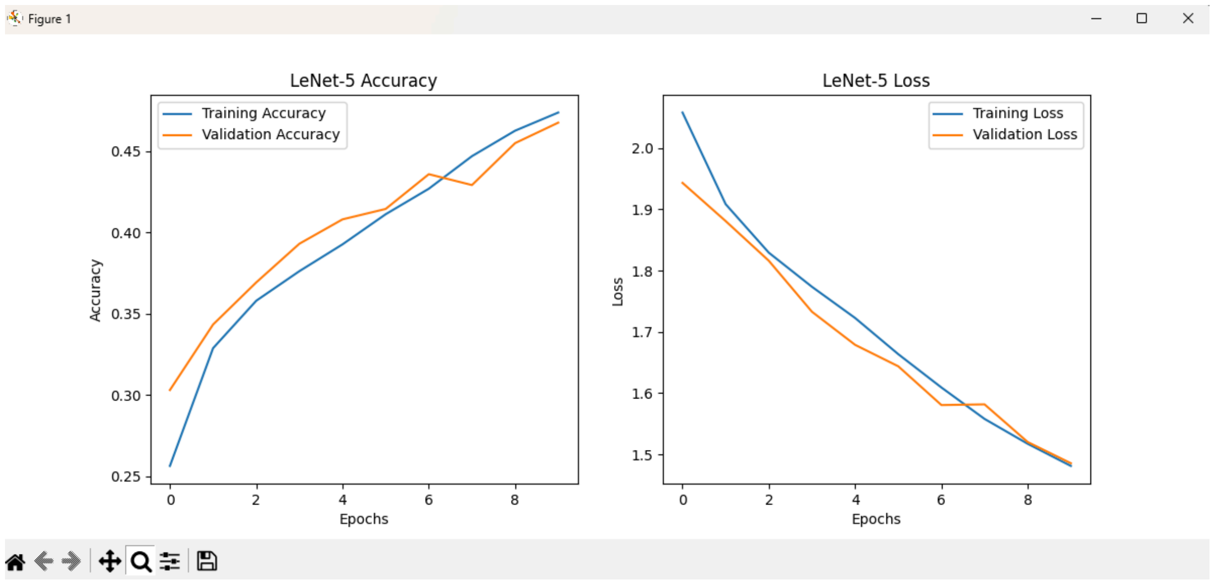
Second optimizer : adam



Third optimizer : RMS







Conclusion

Based on our observations, the VGG-3 model paired with the Adam optimizer proved to be the most effective combination, achieving a 75% accuracy on the testing set. This model structure, along with Adam's adaptive learning rate capabilities, helped it converge more quickly and effectively compared to other optimizers.