# Group 4

## Neural Network Image Classification

G4 Iswarya, Roland, Ralitza ,Gulmehak

# Solution Overview

- **Objective:**
  Develop and deploy a deep learning image classification system using convolutional neural networks (CNNs) on the CIFAR-10 dataset to identify images across 10 categories such as airplanes, cars, and animals.

- **Models Created:**
  - A couple of custom baseline CNNs built from scratch -
  - A transfer learning model using VGG16 pretrained on ImageNet for improved accuracy.
  - Additional experimental models for comparison.

- **Goals:** Train multiple models; evaluate performance metrics (accuracy, precision, recall, F1 score), and integrate the best-performing ones into an interactive platform.

- **Output:** A locally running Streamlit app, a Gradio App, and a publically available Hugging Face app that allows users to select a pre-trained model, upload an image, and receive instant classification prediction

# Dataset: Cifar10

- **Manageable size:** It contains 60,000 color images (32×32 pixels), which is small enough to train models on a standard laptop without needing high-end hardware.
- **Balanced and diverse:** It includes 10 evenly distributed classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck), testing your model's ability to distinguish between different object types.
- **Readily available:** CIFAR-10 is built into TensorFlow and Keras, so it can be loaded easily with one command, reducing setup friction.

# Data PreProcessing - why?

**Standardizing the image data — resizing, normalizing, and encoding — ensures models can learn patterns effectively and make accurate predictions on new inputs.**

- **Normalized pixel values** by dividing all RGB pixel intensities by 255.0, converting them from the original 0–255 range to 0–1. This helps the neural network learn more efficiently and improves gradient stability during training.
- **Converted images to numerical arrays** so they could be processed by Keras.
- **One-hot encoded the class labels** (during training) so the model could output probabilities across the 10 CIFAR-10 categories.
- In the Demo apps, the **same preprocessing pipeline** was used (`image.load_img`, `image.img_to_array`, normalization, and reshaping) to ensure uploaded images matched the format of the training data.

# Optimization Techniques

- **We experimented with different optimizers,** including Adam (adaptive learning rate optimization) and Gradient Descent , to identify which gave the best accuracy and convergence speed.
- **Learning rate tuning:** By testing learning rate values we controlled how quickly your model adjusted its weights.
- **Dropout layers:** We added Dropout to prevent overfitting by randomly disabling neurons during training, improving model generalization.
- **Early stopping:** We implemented EarlyStopping to halt training automatically when validation loss stopped improving, avoiding wasted epochs and overfitting.
- **Transfer learning:** The VGG16-based model reused pre-trained ImageNet weights, which accelerating training and Should have improved the accuracy.

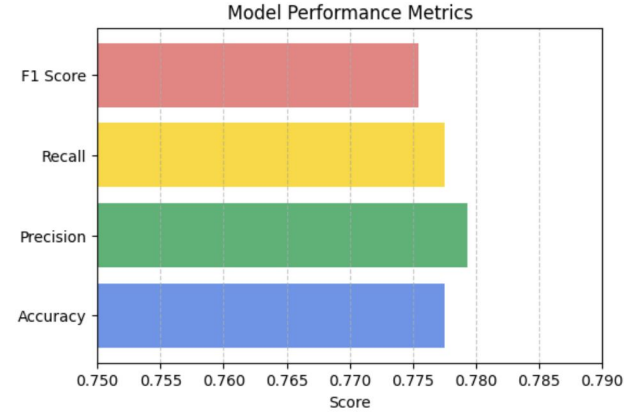
# Models - Excel table

| | Roland | Roland | Ishu | Ishu | Ishu | Ishu | Ralitza | Ralitza | Ralitza | Gulmehak |
|---|---|---|---|---|---|---|---|---|---|---|
| | **M1 Baseline Roland** | **M2 VGG16 Transfer Le** | M1 Ishu | M2 Ishu | M3 Ishu | baseline CNN | **baseline_CNN_adam_m1** | **sgd_m1** | **adam_m2** | |
| **Dataset** | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | Animals-10 |
| **Data Processing** | | | | | | | | | | |
| | | | input_shape = (32, 32, 3) | | | input_shape = (32, 32, 3) | Input(shape=(32, 32, 3)) | Input(shape=(32, 32, 3)) | Input(shape=(32, 32, 3)) | input_shape=(128, 128, 3) |
| **Architechture** | | | data_augmentation(inputs) | | | data_augmentation(inputs) | data_augmentation(inputs) | data_augmentation(inputs) | data_augmentation(inputs) | data_augmentation(inputs) |
| | Conv2D(32, 3x3) | | Conv2D(32, (3,3) | Conv2D(32, (3,3) | Conv2D(32, (3,3) | Conv2D(32, (3,3) | conv_block (32) | conv_block (32) | conv_block (32) | conv2d_24 (Conv2D) |
| | MaxPooling2D(2x2) | | MaxPooling2D((2,2) | MaxPooling2D((2,2) | MaxPooling2D((2,2) | MaxPooling2D((2,2) | layers.MaxPooling2D() | layers.MaxPooling2D() | layers.MaxPooling2D() | max_pooling2d_12 (MaxPooling2D) |
| | Conv2D(32, 3x3) | | Conv2D(64, (3,3) | Conv2D(32, (3,3) | Conv2D(32, (3,3) | Conv2D(32, (3,3) | layers.Dropout(0.2) | layers.Dropout(0.2) | layers.Dropout(0.2) | dropout_16 (Dropout) |
| | MaxPooling2D(2x2) | | MaxPooling2D((2,2) | MaxPooling2D((2,2) | MaxPooling2D((2,2) | MaxPooling2D((2,2) | conv_block(64) | conv_block(64) | conv_block(64) | conv2d_26 (Conv2D) |
| | Conv2D(64, 3x3) | | | Conv2D(64, (3,3) | Conv2D(64, (3,3) | Conv2D(64, (3,3) | layers.MaxPooling2D() | layers.MaxPooling2D() | layers.MaxPooling2D() | max_pooling2d_13 (MaxPooling2D) |
| | Conv2D(64, 3x3) | | | MaxPooling2D((2,2) | MaxPooling2D((2,2) | MaxPooling2D((2,2) | layers.Dropout(0.3) | layers.Dropout(0.3) | layers.Dropout(0.3) | dropout_17 (Dropout) |
| | MaxPooling2D(2x2) | | | | | | conv_block(128) | conv_block(128) | conv_block(128) | conv2d_28 (Conv2D) |
| | Dropout(0.25/0.5) | | layers.Dropout(0.25) | Dropout(0.5), | Dropout(0.5), | Dropout(0.25) | layers.MaxPooling2D() | layers.MaxPooling2D() | layers.MaxPooling2D() | (MaxPooling2D) |
| | Flatten | | layers.Flatten() | flatten | flatten | Flatten() | layers.Dropout(0.4) | layers.Dropout(0.4) | layers.Dropout(0.4) | dropout_18 (Dropout) |
| | Dense(512, relu) | | layers.Dense(512, activation= | Dense(512, activation='relu') | Dense(512, activation='relu') | Dense(512, activation='relu') | GlobalAveragePooling2D() | GlobalAveragePooling2D() | GlobalAveragePooling2D() | flatten (Flatten) |
| | Dropout(0.5) | 0.5 | layers.Dropout(0.5) | Dropout(0.5), | Dropout(0.5), | Dropout(0.5) | Dense(256, ) | Dense(256, ) | Dense(256, ) | dense_8 (Dense) |
| | Dense(10, softmax) | softmax | layers.Dense(10, activation='s | Dense(10, softmax) | Dense(10, softmax) | Dense(10, softmax) | Dropout(0.5) | Dropout(0.5) | Dropout(0.5) | dropout_19 (Dropout) |
| | | | | | | | Dense(10,softmax ) | | | |
| **Hyperparameters** | | | | | | | | | | |
| **LR** | 0.001 | | 0.001 | 0.001 | 0.002 | 0.001 | 1e-3 (0.001) | 0.01 | | 1.00E-03 |
| **Optimizer** | Adam | | Adam | Adam | Adam | Adam | Adam | SGD | AdamW | Adam |
| **Epochs** | 10 | 2 | 10 | 10 | 20 | 50 | 50 | 50 | 50 | 60 |
| **Batch** | 64 | 128 | 64 | 64 | 64 | 64 | 32 | 32 | 32 | 32 |
| **Training time** | 30mins | 15 years | | 0.77 minutes | 1.11 minutes | 20.42 minutes | 21 | 19 | | 40 with early stopping |
| **Final Loss** | 0.6133 | 1.22 | 0.6022257805 | 0.7302 | 0.6887 | 0.4978279769 | 0.4078 | 0.4422 | 1.0958 | 0.064 |
| **Final Accuracy** | 0.785 | 0.529 | 0.798600018 | 0.7808 | 0.9079 | 0.8307999969 | 0.8958 | 0.8834 | 0.7878 | 0.9803 |
| **Observations** | Could probably increse the | Need more epochs but cannot access collab | | early spotting in cnn | | The use of data augmentation, batch normalization, and learning rate control contributed to improved generalization and stable training performance." | | | | |

# Evaluation of our tested Models - Diagrams and accuracy

## Baseline CNN Model 1

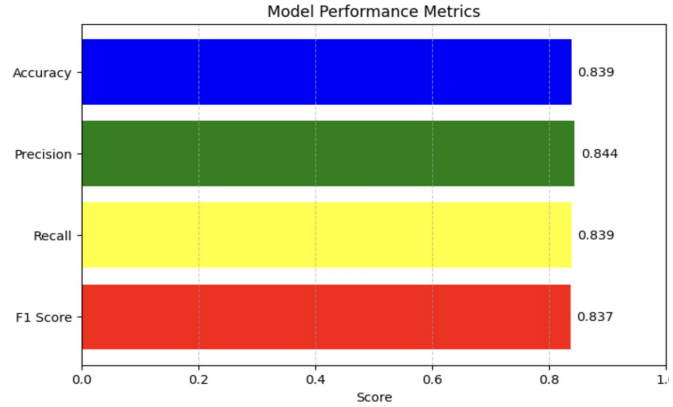
```
Final Training Accuracy: 0.7928
Test Accuracy: 0.7775
Test Precision: 0.7793
Test Recall: 0.7775
Test F1 Score: 0.7755
```




```
Precision: 0.7793047578064232
Recall: 0.7775
F1: 0.7754710386198069
Accuracy: 0.7775
```

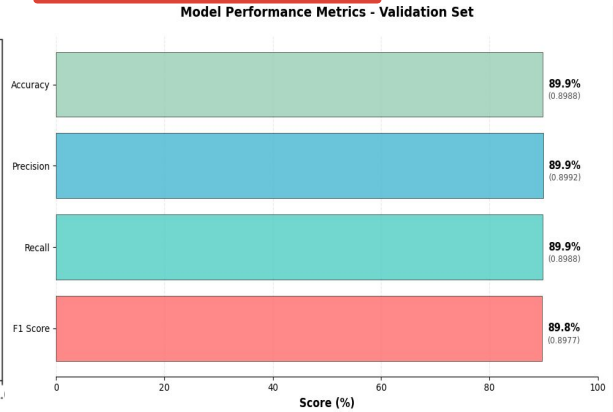
## Baseline CNN Model 2

```
Test accuracy: 0.838699996471405
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
🔍 Model Performance Metrics:
F1 Score  : 0.8365
Recall    : 0.8387
Precision : 0.8439
Accuracy  : 0.8387
```



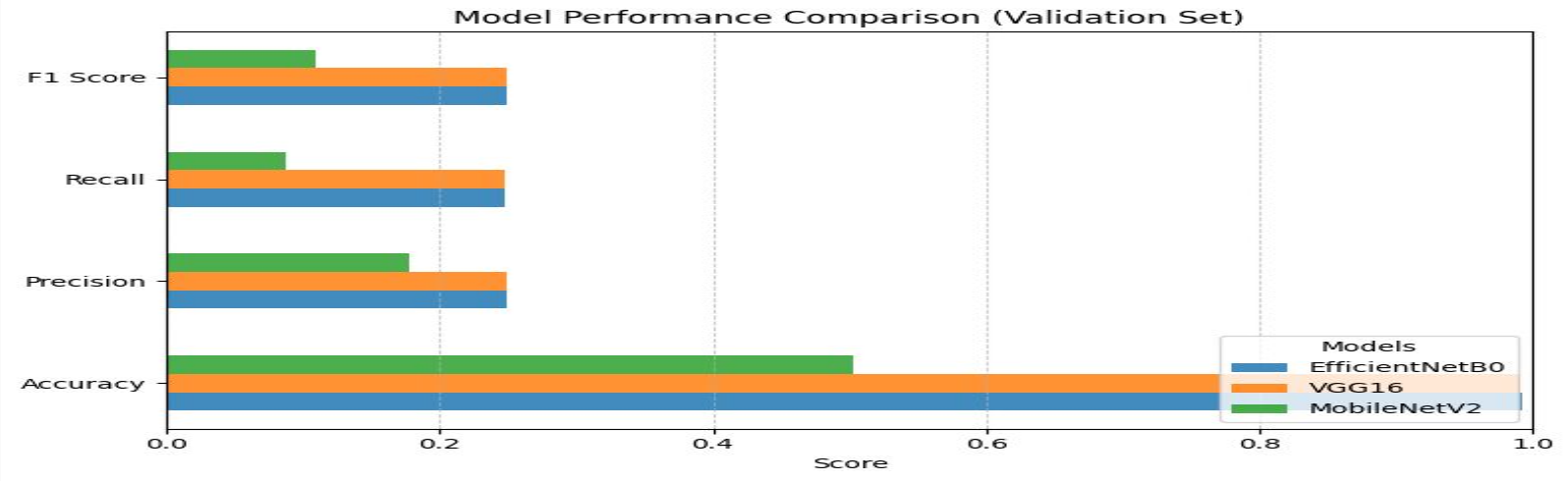

## Baseline CNN Model 3

```
:mag: Model Performance Metrics (Validation Set):
F1 Score  : 0.8977 (89.77%)
Recall    : 0.8988 (89.88%)
Accuracy  : 0.8988 (89.88%)
```



# Evaluation of our tested Models - Diagrams and accuracy

```
Results:
EfficientNetB0: acc=0.9931  P=0.2497  R=0.2480  F1=0.2489
VGG16: acc=0.9910  P=0.2497  R=0.2474  F1=0.2485
MobileNetV2: acc=0.5026  P=0.1780  R=0.0866  F1=0.1088
```



# Model CNN_adam_m1

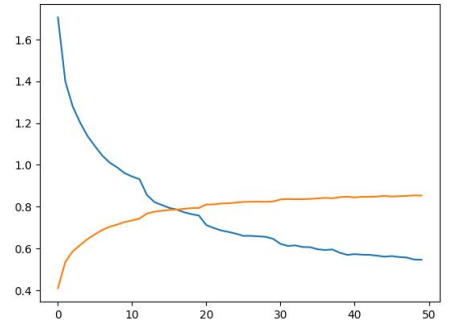
## Selected model (Intro)

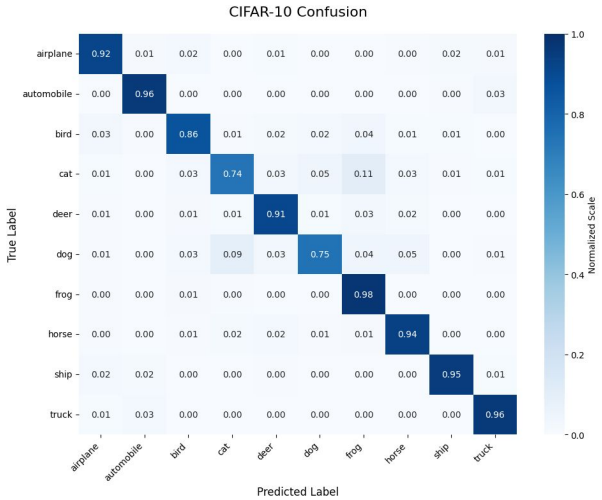
- **CIFAR-10 Dataset**
- **Model CNN_adam_m1:**
  - Accuracy (0.8958)
  - Loss (0.407)
- **Data Preprocessing:**
  - Numbered labels
  - Flattened (squeeze)
  - Normalized Dataset

Model: "cifar10_cnn"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer (InputLayer) | (None, 32, 32, 3) | 0 |
| data_augmentation (Sequential) | (None, 32, 32, 3) | 0 |
| sequential (Sequential) | (None, 32, 32, 32) | 10,336 |
| max_pooling2d (MaxPooling2D) | (None, 16, 16, 32) | 0 |
| dropout (Dropout) | (None, 16, 16, 32) | 0 |
| sequential_1 (Sequential) | (None, 16, 16, 64) | 55,808 |
| max_pooling2d_1 (MaxPooling2D) | (None, 8, 8, 64) | 0 |
| dropout_1 (Dropout) | (None, 8, 8, 64) | 0 |
| sequential_2 (Sequential) | (None, 8, 8, 128) | 222,208 |
| max_pooling2d_2 (MaxPooling2D) | (None, 4, 4, 128) | 0 |
| dropout_2 (Dropout) | (None, 4, 4, 128) | 0 |
| global_average_pooling2d (GlobalAveragePooling2D) | (None, 128) | 0 |
| dense (Dense) | (None, 256) | 33,024 |
| dropout_3 (Dropout) | (None, 256) | 0 |
| dense_1 (Dense) | (None, 10) | 2,570 |

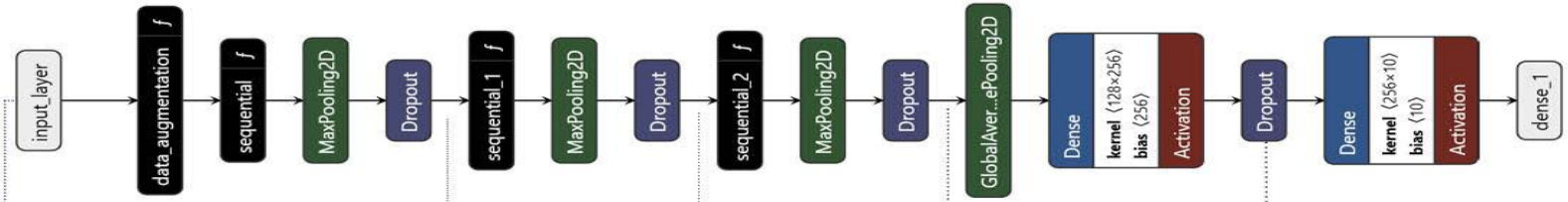
Total params: 323,946 (1.24 MB)
Trainable params: 323,050 (1.23 MB)
Non-trainable params: 896 (3.50 KB)



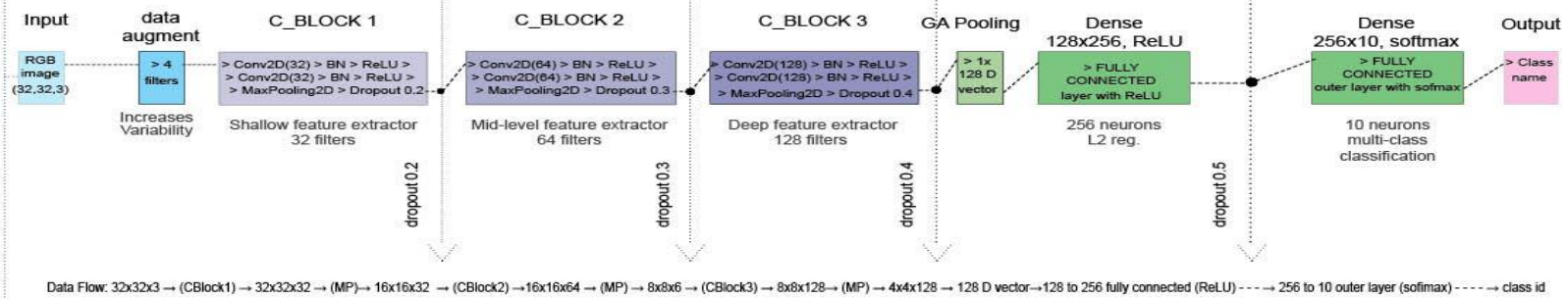



# Model CNN_adam_m1

## model architecture_technical




CNN model architecture
Diagram generated via netron.app




**Data Preprocessing**

- num. labels; flattened (squeeze); one-hot encoded
- normalize dataset

CNN model architecture_technical diagram

# Model CNN_adam_m1

## model architecture_spatial

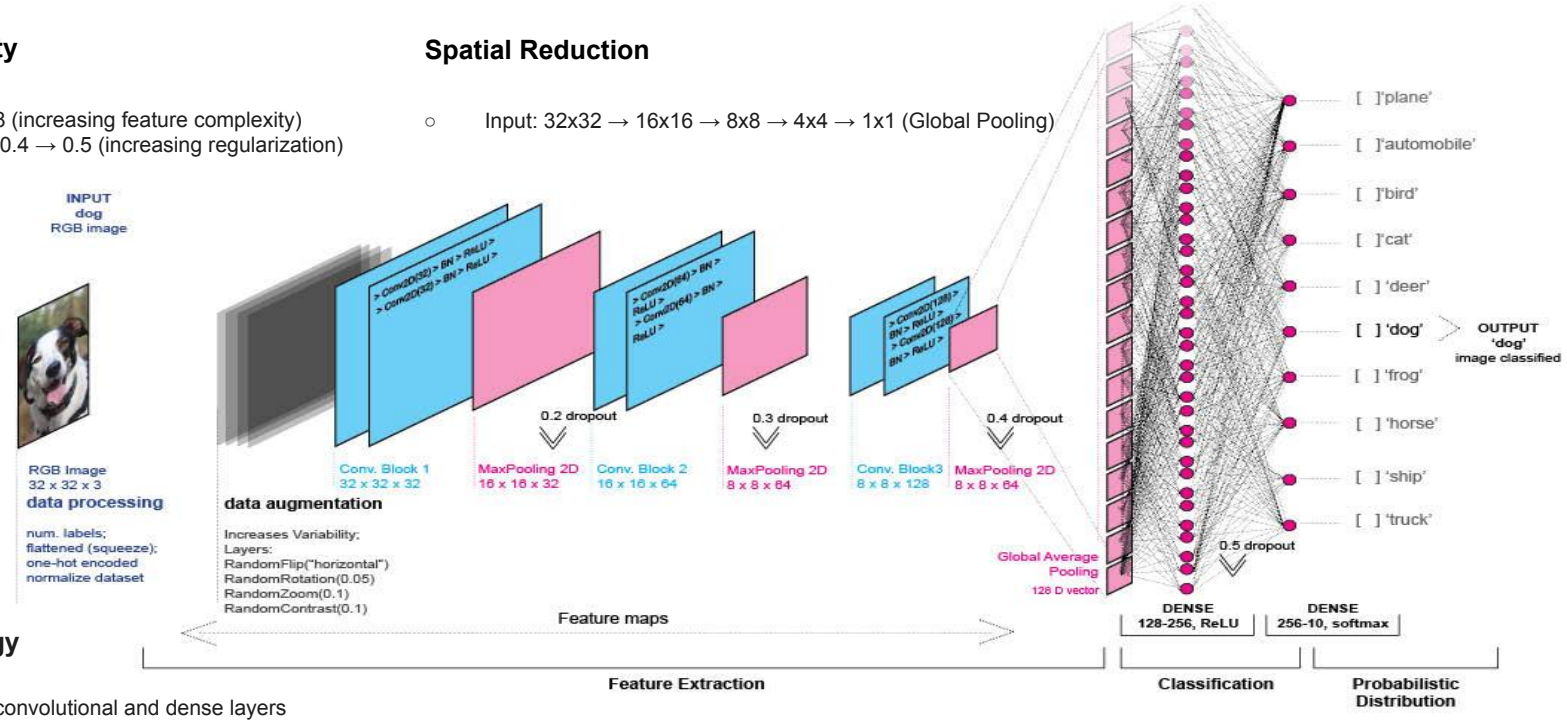
### Design Philosophy

#### Progressive Complexity

- Filters: 32 → 64 → 128 (increasing feature complexity)
- Dropout: 0.2 → 0.3 → 0.4 → 0.5 (increasing regularization)

#### Spatial Reduction

- Input: 32x32 → 16x16 → 8x8 → 4x4 → 1x1 (Global Pooling)




CNN model architecture_spatial diagram

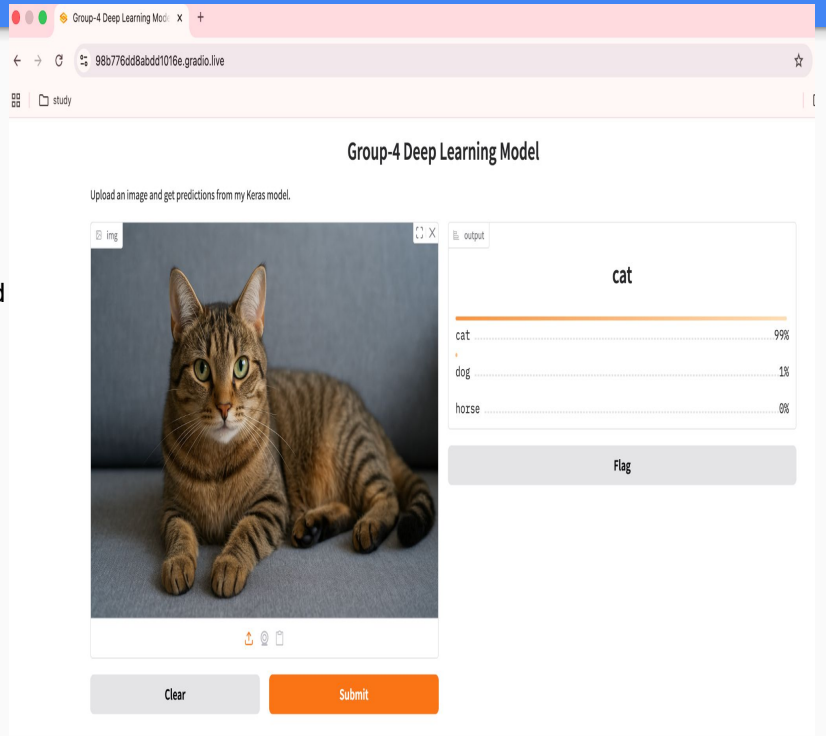
#### Regularization Strategy

- **L2 Weight Decay:** All convolutional and dense layers
- **BatchNorm:** After every convolution)
- **Dropout:** Increasing rates through network
- **Data Augmentation:** Input-level regularization

# Model Deployments (1)

## Gradio Deployment

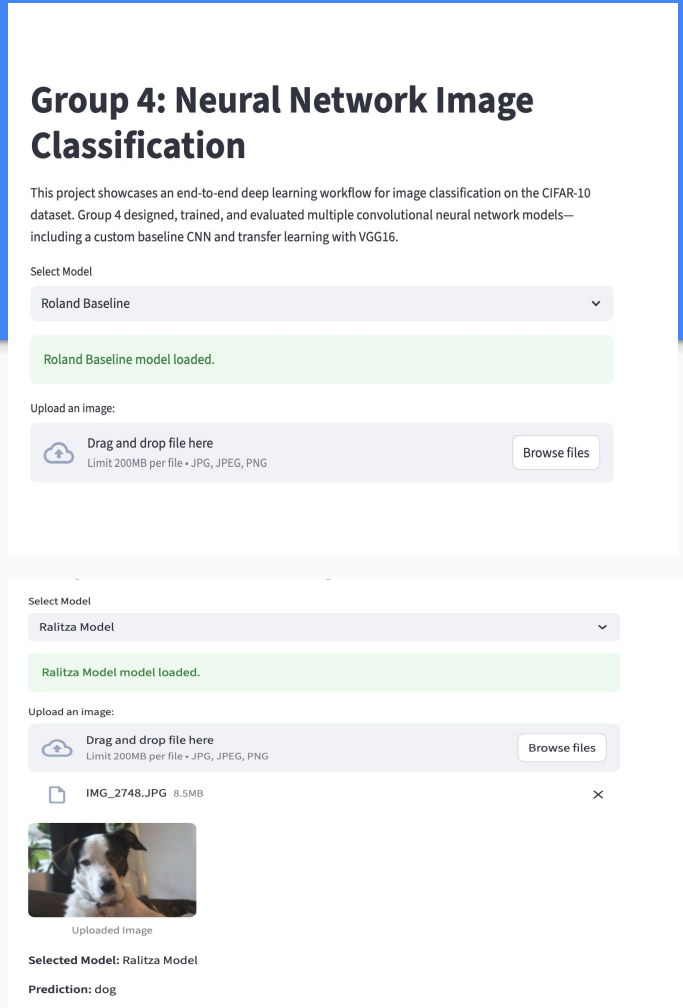
- **Objective:**
  - Create a simple, shareable web interface for testing and showcasing the model using **Gradio**.
- **Methods:**
  - Utilized **Gradio Interface** for rapid model deployment.
  - Integrated the same trained model we have designed.
  - Focused on accessibility and instant interaction through browser-based testing.
- **Steps:**
  - Defined a prediction function linked to the model.
  - Configured image input and label output components.
  - Tested locally and then deployed via Google colab.
  - Hosted the app via **Gradio link** created and tested in browser.
- **Key Points:**
  - Quickest way to demo machine learning models online.
  - No setup required for users — runs directly in the browser.
  - Provides a neat visualization of predictions and probabilities.
  - Excellent tool for collaboration, evaluation, and public sharing.



# Model Deployments (2)

## Streamlit Deployment

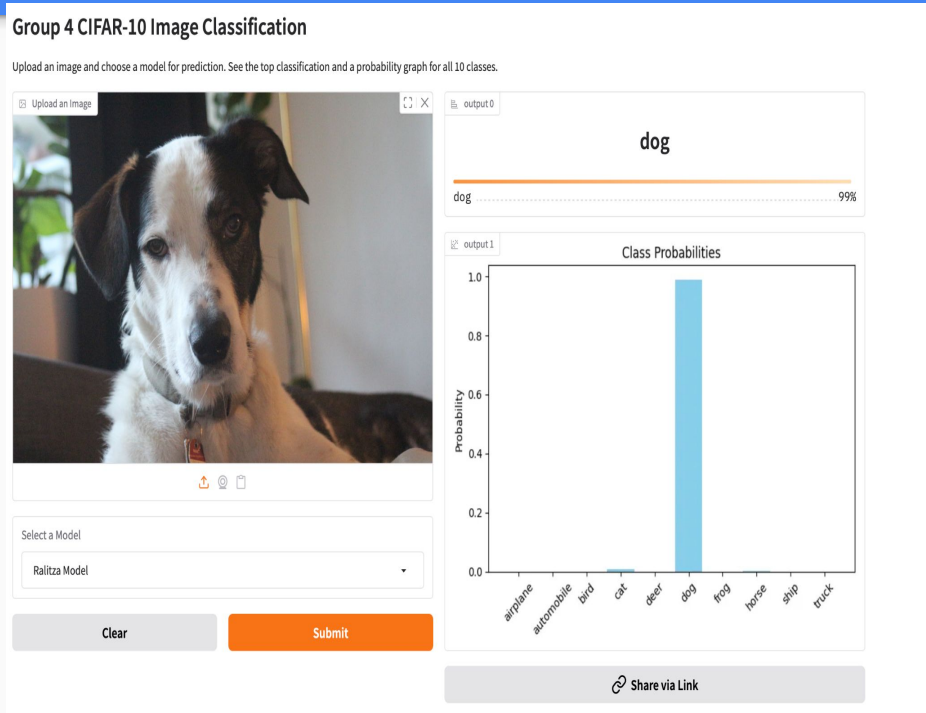
- **Objective:**
  - Deploy the trained image classification model through an interactive and user-friendly web interface using **Streamlit**.
- **Methods:**
  - **Streamlit framework** to build a lightweight, Python-based web app.
  - Integrated the trained CNN/Transfer Learning model for real-time predictions.
  - Incorporated image upload and visualization components for user interaction
- **Steps:**
  - Loaded the saved model and preprocessing pipeline.
  - Designed an upload interface for single/multiple image inputs.
  - Processed images and displayed predictions with confidence levels.
  - Hosted the app via **Streamlit Local.**
- **Key Points:**
  - No backend coding required — everything handled in Python.
  - Clean, responsive, and interactive interface.
  - Enables users to visualize both input images and model predictions instantly.
  - Ideal for demonstration and prototype-level deployment.



# Model Deployments (3)

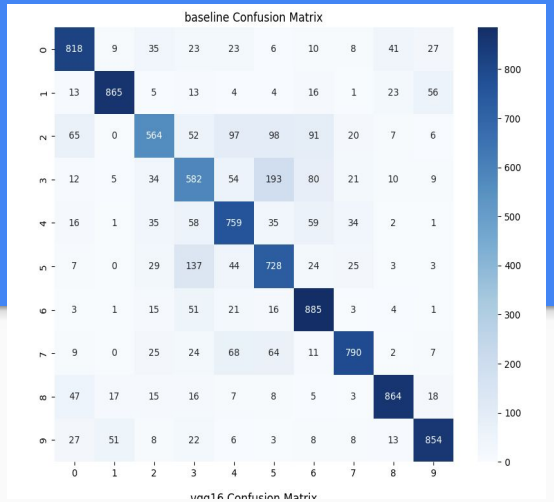
## Hugging face Deployment https://huggingface.co/spaces/Roland50/group4-cifar10-classifier

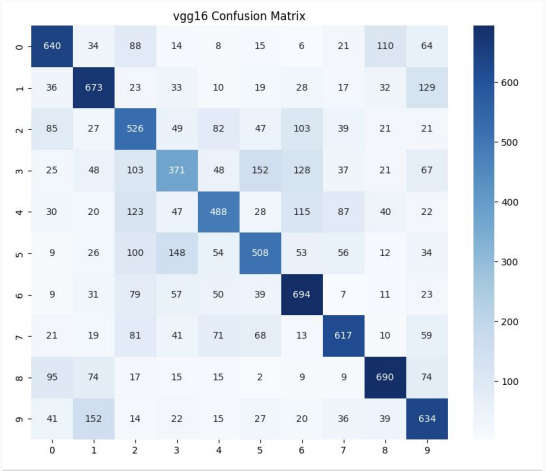
- **Objective:**
  - Host and share the trained CNN model using **Hugging Face Spaces**.
- **Methods:**
  - Used Hugging Face Spaces for deployment. Integrated the final trained model.
  - Focused on creating a public, browser-accessible demo.
- **Steps:**
  - Exported trained model and app files.
  - Created a new Space → uploaded the required files in Hugging Face.
  - Auto-built and deployed by Hugging Face.
  - Tested and shared the public app link.
- **Key Points:**
  - Cloud-hosted and instantly shareable.
  - No installation or local setup required.
  - Ideal for public demos and project showcases.



# Transfer Learning

- We tried transfer learning by adapting the pretrained VGG16 model
- We Chose VGG16 because it brings strong feature extraction from large-scale ImageNet training, enabling our models to benefit from pre-learned visual patterns and improve performance on smaller or more specific datasets like our dataset in this exercise
- This approach gives your network a “head start”—it doesn’t start learning from scratch, but uses what it already “knows” about shapes, colors, and textures to do better and faster on your small dataset. **The expected outcome** was that this adapted model would be much more accurate than one trained from nothing.. (Rolands did not as the epochs took too long to train but presumably would have given a better model)



# Recap:

- At the start of our project, we sat together and mapped out our goals. Our task was to evaluate how well a CNN can classify images. To explore this, we chose two datasets: CIFAR-10 and Animals-10 from Kaggle. We decided to work on them separately but support each other along the way.
- First, we prepared the images. We loaded them, resized them, normalized them, and even used data augmentation—like flipping and rotating—to make the model learn better.
- Then, we built our own CNN model, trained it, and tested its performance. After that, we tried Transfer Learning using a pretrained model. This is where we saw a big improvement.
- Because pretrained models already know how to detect shapes, edges, and textures from huge datasets. So they learn faster and give better accuracy, especially when our own dataset is not very large. A custom CNN has to learn everything from scratch, so it needs much more data.
- In the end, Transfer Learning clearly performed better and gave us stronger, more reliable results.

# Key Takeaways:

- **Data preparation is critical:** Proper normalization of the images, and data visualization really help
- **Model design impacts accuracy:** Layer selection and parameter tuning directly influence performance.
- **Training techniques matter:** Optimizers, batch size, and early stopping help control overfitting and improve convergence.
- **Evaluation metrics provide insight:** Accuracy, precision, recall, F1-score, and confusion matrix highlight performance.
- **Transfer learning saves time:** Pre-trained models often achieve better accuracy with less data and training effort.
- **Code clarity and documentation are essential:** Well-structured code and comments helps team collaboration.
- **Deployment completes the pipeline:** its is pretty satisfying to deploy a working app using the models you create.


# Thank you!