KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)

Deemed to be University U/S 3 of UGC Act, 1956

# SUMMER TRAINING PROJECT ON MLOP

**TOPIC**: OBJECT DETECTION USING CIFAR DATASET

Under The Guidance
Of
Suprava Patnaik

By:
ABIR SARKAR
ROLL NO.: 2230057
ECsc

## CIFAR Dataset:

The **CIFAR** dataset is a well-established benchmark in the field of machine learning, specifically designed for **image classification**. Comprising **60,000 color images**, each of **size 32x32 pixels**, the dataset is segmented into **10 distinct classes**, each representing a different object or creature. Each class contains an equal distribution, boasting 6,000 images. From the total image count, 50,000 are designated for training while the remaining 10,000 are set aside for testing.

## Objectives:

### Data Insights and Exploration

- Familiarize with the CIFAR dataset.
- Visually inspect sample images from various classes to understand data distribution.

### Comprehensive Data Preprocessing

- Normalize pixel values of the images to enhance model training efficiency.

- Convert image labels into a one-hot encoded format suitable for classification tasks.
- Implement data augmentation techniques to increase the dataset's variability and improve model generalization.

**Architectural Design using Keras**

- Design a Convolutional Neural Network (CNN) tailored for the  CIFAR dataset using the **Keras** framework.
- Incorporate mechanisms such as dropouts and regularizations to counteract overfitting.

**Model Training Process**

- Train the CNN using the prepared dataset.
- Utilize callbacks to adjust the learning rate dynamically and halt the training early if no improvements are detected, restoring the best model weights from the training.

**Learning Analysis**

- Visualize the model's learning curves, observing both training and validation performance metrics over epochs.

**Model Evaluation**

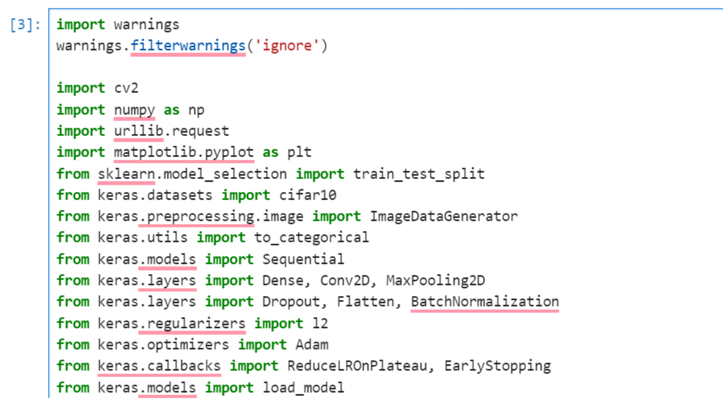
- Assess the trained model's accuracy and loss on the unseen test data to determine its robustness.

**Real-world Generalization Check**

- Evaluate the model's predictive capability using an image not part of the CIFAR-10 dataset to gauge its real-world applicability.

1. Import Necessary Libraries **-**

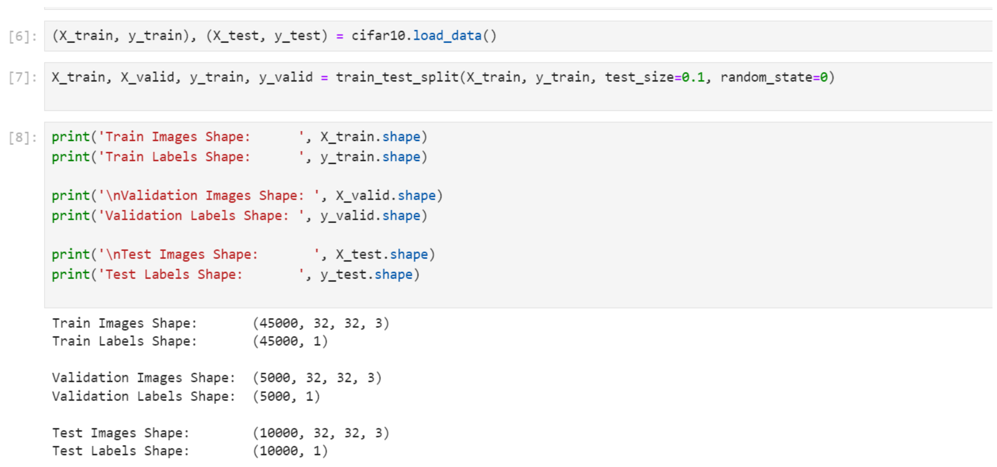
We're importing all the necessary libraries to kick off our project. We'll be relying on TensorFlow and Keras to handle the image data, craft our model, and optimize it for best performance:



```
[3]: import warnings
     warnings.filterwarnings('ignore')

     import cv2
     import numpy as np
     import urllib.request
     import matplotlib.pyplot as plt
     from sklearn.model_selection import train_test_split
     from keras.datasets import cifar10
     from keras.preprocessing.image import ImageDataGenerator
     from keras.utils import to_categorical
     from keras.models import Sequential
     from keras.layers import Dense, Conv2D, MaxPooling2D
     from keras.layers import Dropout, Flatten, BatchNormalization
     from keras.regularizers import l2
     from keras.optimizers import Adam
     from keras.callbacks import ReduceLROnPlateau, EarlyStopping
     from keras.models import load_model
```

2. Data Preparation and Exploration-

Let's download the CIFAR-10 dataset from Keras library. Then, let's split original training data to training and validation sets.



```
[6]: (X_train, y_train), (X_test, y_test) = cifar10.load_data()

[7]: X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.1, random_state=0)

[8]: print('Train Images Shape:      ', X_train.shape)
     print('Train Labels Shape:      ', y_train.shape)

     print('\nValidation Images Shape: ', X_valid.shape)
     print('Validation Labels Shape: ', y_valid.shape)

     print('\nTest Images Shape:       ', X_test.shape)
     print('Test Labels Shape:       ', y_test.shape)

     Train Images Shape:       (45000, 32, 32, 3)
     Train Labels Shape:       (45000, 1)

     Validation Images Shape:  (5000, 32, 32, 3)
     Validation Labels Shape:  (5000, 1)

     Test Images Shape:        (10000, 32, 32, 3)
     Test Labels Shape:        (10000, 1)
```

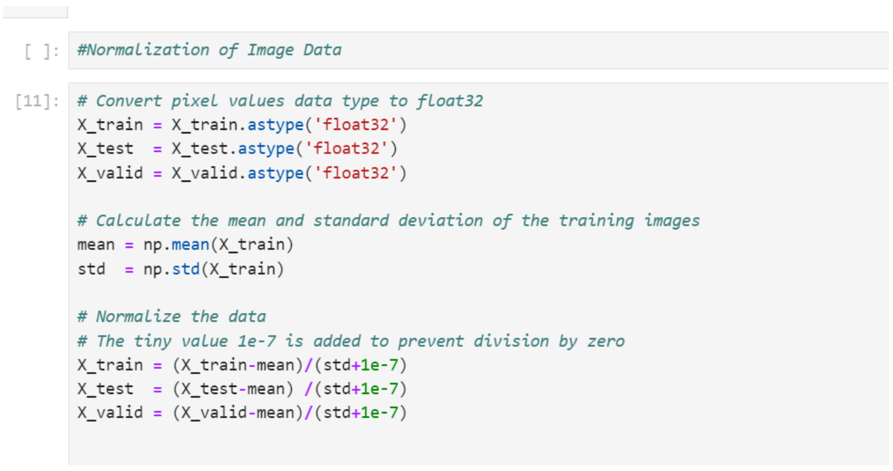
3. Data Preprocessing-

In the **Data Preprocessing** phase, we undertake essential preparatory measures to ensure our dataset is aptly primed for the modeling process:

- **Normalization of Image Data**
- **One-Hot Encoding of Labels**
- **Data Augmentation**

Normalization of Image Data:

First of all, I am going to convert the pixel values data type to float32 type, and then normalizes them by subtracting the mean and dividing by the standard deviation of the training set, enhancing the model's training efficiency and effectiveness:



```
[ ]: #Normalization of Image Data
```

```
[11]: # Convert pixel values data type to float32
X_train = X_train.astype('float32')
X_test  = X_test.astype('float32')
X_valid = X_valid.astype('float32')

# Calculate the mean and standard deviation of the training images
mean = np.mean(X_train)
std  = np.std(X_train)

# Normalize the data
# The tiny value 1e-7 is added to prevent division by zero
X_train = (X_train-mean)/(std+1e-7)
X_test  = (X_test-mean) /(std+1e-7)
X_valid = (X_valid-mean)/(std+1e-7)
```

## Data Augmentation:

Finally, I am going to implement data augmentation to artificially expand the size of the training set by creating modified versions of images in the dataset. This helps improve the model's ability to generalize, thereby reducing overfitting. Data augmentation techniques such as rotations, shifts, flips, shearing, and intensity changes introduce small variations to the existing images, creating a broader set of training samples to learn from.

The choice of data augmentation techniques often depends on the specific characteristics of the dataset and the problem at hand. The **CIFAR** dataset comprises small color images of objects from 10 different classes. Given the nature of these images, some augmentation techniques are more applicable than others:

- **Rotation**: A small degree of rotation can help the model become invariant to the orientation of the object. The `rotation_range=15` means the image could be rotated randomly within -15 to 15 degrees. However, large rotations could be harmful since the CIFAR-10 images are relatively small and a big rotation might put the object outside of the image.

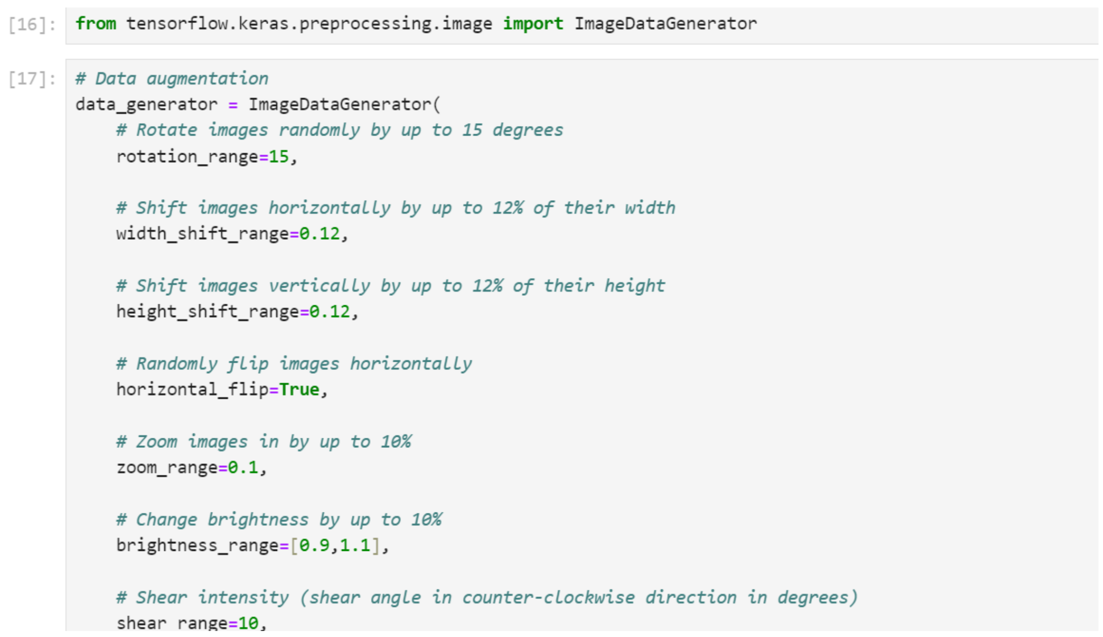
- **Width and Height shift**: Small shifts can help the model become invariant to the position of the object in the image.
  Here, `width_shift_range=0.12` and `height_shift_range=0.12`mean the image could be moved horizontally or vertically by up to 12% of its width or height respectively. Again, since the images are small, large shifts might put the object outside of the image.

- **Horizontal Flip**: A horizontal flip is a sensible choice for this dataset because for many images, the object of interest remains the same when flipped horizontally (for example, a flipped car is still a car).

- **Shear Intensity**: With `shear_range=10`, a shear intensity within the range of -10 to +10 degrees is applied. This transformation slants the shape of the image, helping the model to recognize objects in different perspectives.

- **Channel Shift Intensity**: With `channel_shift_range=0.1`, the intensities of the RGB channels are randomly shifted by up to 10% of their full scale. This can help the model handle different lighting conditions and color variations.

While some augmentation techniques like vertical flips and color jittering may not be appropriate for all classes in the CIFAR dataset, the chosen techniques are expected to help improve the robustness and generalization capability of the model.



```
[16]: from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
[17]: # Data augmentation
      data_generator = ImageDataGenerator(
          # Rotate images randomly by up to 15 degrees
          rotation_range=15,

          # Shift images horizontally by up to 12% of their width
          width_shift_range=0.12,

          # Shift images vertically by up to 12% of their height
          height_shift_range=0.12,

          # Randomly flip images horizontally
          horizontal_flip=True,

          # Zoom images in by up to 10%
          zoom_range=0.1,

          # Change brightness by up to 10%
          brightness_range=[0.9,1.1],

          # Shear intensity (shear angle in counter-clockwise direction in degrees)
          shear_range=10,
```

## 4. CNN Model Architecture-

Here is a brief explanation of the architecture:

- The network begins with **a pair of Conv2D layers**, each with **32 filters of size 3x3**. This is followed by a **Batch Normalization** layer which accelerates training and provides some level of **regularization**, helping to prevent overfitting.

- The pairs of Conv2D layers are followed by a **MaxPooling2D layer**, which reduces the spatial dimensions (height and width), effectively providing a form of translation invariance and reducing computational complexity. This is followed by a **Dropout layer** that randomly sets a fraction (0.2 for the first dropout layer) of the input units to 0 at each update during training, helping to prevent overfitting.

- This pattern of two Conv2D layers, followed by a Batch Normalization layer, a MaxPooling2D layer, and a Dropout layer, repeats three more times. The number of filters in the Conv2D layers doubles with each repetition, starting from 32 and going up to 64, 128, and then 256. This increasing pattern helps the network to learn more complex features at each level. The dropout rate also increases at each step, from 0.2 to 0.5.

- After the convolutional and pooling layers, a **Flatten layer** is used to convert the 2D outputs of the preceding layer into a 1D vector.

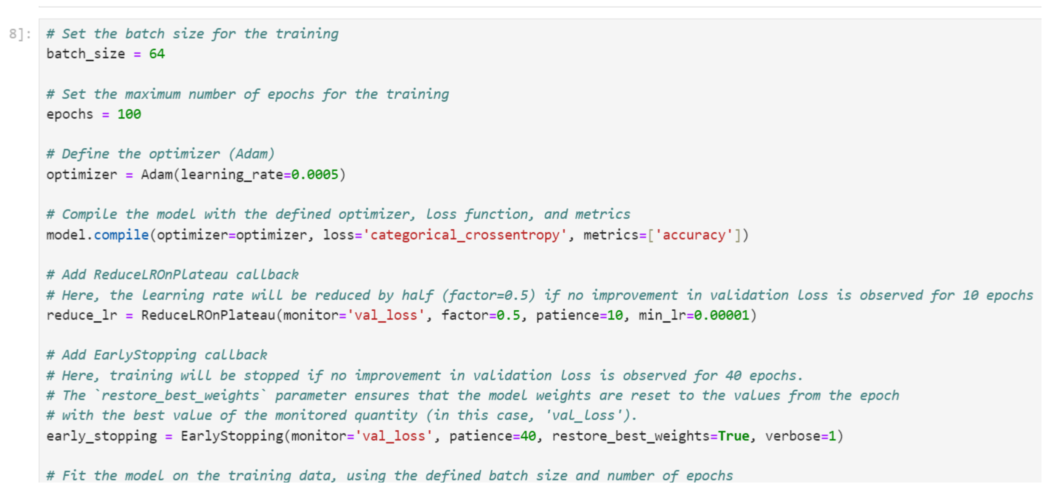
- Finally, a **Dense (or fully connected) layer** is used for classification. It has 10 units, each representing one of the 10 classes of the CIFAR dataset, and a **softmax activation function** is used to convert the outputs to probability scores for each class.

This architecture leverages the strengths of deep CNNs to learn hierarchical features from the CIFAR-10 images. Regularization techniques such as **L2 regularization**, **Dropout**, and **Batch Normalization** are also used to combat overfitting.

5. Training the CNN Model-

The training uses a batch size of 64 and will run for a maximum of 250 epochs or until the early stopping condition is met. During the training, the model's performance is evaluated on the validation data after each epoch. I've added a couple of callback functions to enhance the training process:

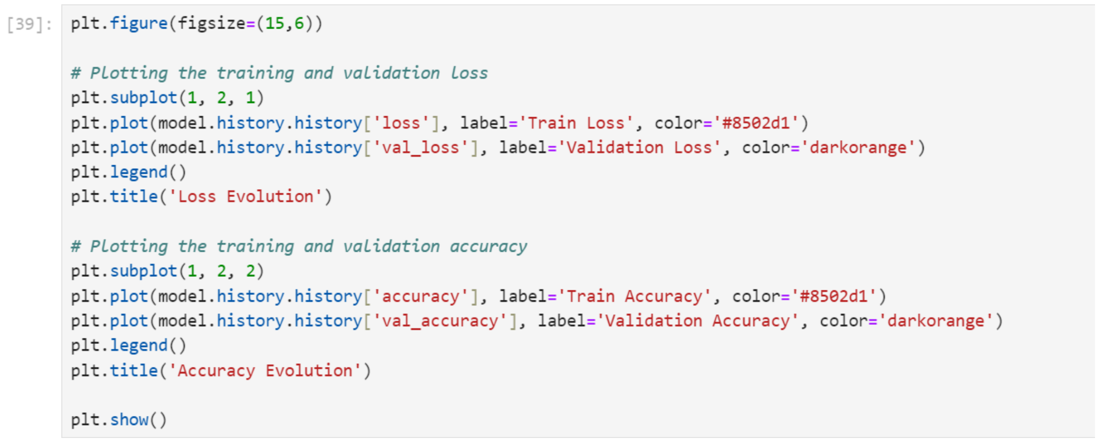
> The **ReduceLROnPlateau callback** is used to reduce the learning rate by half (factor=0.5) whenever the validation loss does not improve for 10 consecutive epochs. This helps to adjust the learning rate dynamically, allowing the model to get closer to the global minimum of the loss function when progress has plateaued. This strategy can improve the convergence of the training process.
>
> The **EarlyStopping callback** is employed to monitor the validation loss and halt the training process when there hasn't been any improvement for a certain number of epochs, ensuring that the model doesn't waste computational resources and time. Furthermore, this callback restores the best weights from the training process, ensuring we conclude with the optimal model configuration from the epochs.



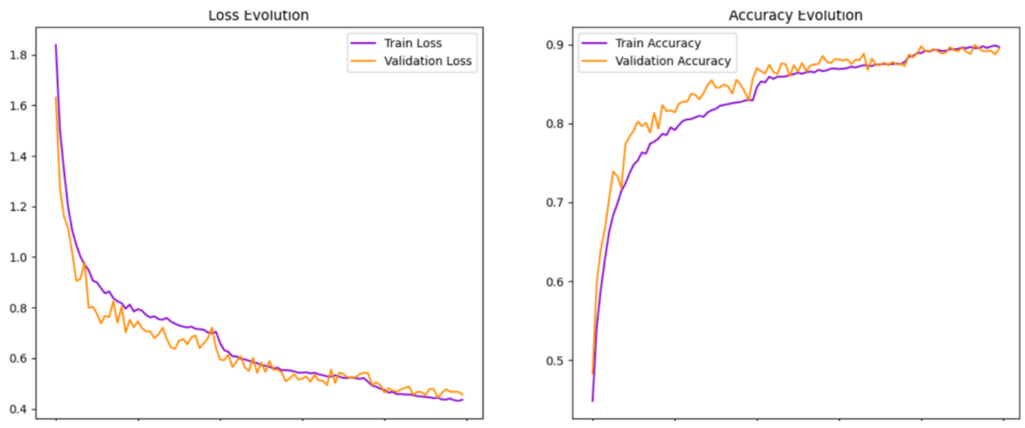
```
8]: # Set the batch size for the training
batch_size = 64

# Set the maximum number of epochs for the training
epochs = 100

# Define the optimizer (Adam)
optimizer = Adam(learning_rate=0.0005)

# Compile the model with the defined optimizer, loss function, and metrics
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# Add ReduceLROnPlateau callback
# Here, the learning rate will be reduced by half (factor=0.5) if no improvement in validation loss is observed for 10 epochs
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=10, min_lr=0.00001)

# Add EarlyStopping callback
# Here, training will be stopped if no improvement in validation loss is observed for 40 epochs.
# The `restore_best_weights` parameter ensures that the model weights are reset to the values from the epoch
# with the best value of the monitored quantity (in this case, 'val_loss').
early_stopping = EarlyStopping(monitor='val_loss', patience=40, restore_best_weights=True, verbose=1)

# Fit the model on the training data, using the defined batch size and number of epochs
```

6. Visualizing the Learning Curves-

Generate plots for visualizing the training and validation loss, and accuracy evolution over epochs using model history



```
[39]: plt.figure(figsize=(15,6))

      # Plotting the training and validation loss
      plt.subplot(1, 2, 1)
      plt.plot(model.history.history['loss'], label='Train Loss', color='#8502d1')
      plt.plot(model.history.history['val_loss'], label='Validation Loss', color='darkorange')
      plt.legend()
      plt.title('Loss Evolution')

      # Plotting the training and validation accuracy
      plt.subplot(1, 2, 2)
      plt.plot(model.history.history['accuracy'], label='Train Accuracy', color='#8502d1')
      plt.plot(model.history.history['val_accuracy'], label='Validation Accuracy', color='darkorange')
      plt.legend()
      plt.title('Accuracy Evolution')

      plt.show()
```



Based on the visualizations above, it's evident that the model is performing well without signs of overfitting. This conclusion is supported by the close alignment of training and validation accuracy and loss values throughout the training process. The gap between training and validation accuracy remains minimal, indicating that the model generalizes well to unseen data. Similarly, the model's loss on validation data closely follows the training loss, reinforcing the assertion of good generalization. Therefore, the model appears to be well regularized and not overfitting to the training data.

The test accuracy of  more than 89%, our model demonstrates exceptional performance on unseen data

## 7. Performance on an Out-of-Dataset Image-

To further explore the generalization capability of our trained CIFAR-10 classification model, I'll assess its performance using an external truck image. This image, which isn't part of the CIFAR dataset, has been sourced from my GitHub repository. It provides an opportunity to see how our model behaves with real-world, out-of-dataset samples. But before taking the raw image from my Git Hub firstly I have to clone the git in the jupyter notebook so that it becomes easy to load the image which is in png format into the model to test its accuracy.

```
Error: Image not loaded correctly.
```

```
[58]: !git clone:https://github.com/ABIR200406/FINAL-CNN-PROJECT.git

git: 'clone:https://github.com/ABIR200406/FINAL-CNN-PROJECT.git' is not a git command. See 'git --help'.
```

```
[59]: !git clone https://github.com/ABIR200406/FINAL-CNN-PROJECT.git

Cloning into 'FINAL-CNN-PROJECT'...
```

```
[61]: # Fetch the raw image from GitHub
      url = "https://github.com/ABIR200406/FINAL-CNN-PROJECT/blob/main/truck_sample.png?raw=true"
      resp = urllib.request.urlopen(url)
      image = np.asarray(bytearray(resp.read()), dtype="uint8")
      image = cv2.imdecode(image, cv2.IMREAD_UNCHANGED)

      # Convert the image from BGR to RGB
      image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

After the real-time image is loaded in the model , we need to preprocess it in the same way as we did with the training and test data .

```
[62]: # Display the image
      plt.imshow(image)
      plt.xticks([])
      plt.yticks([])
      plt.grid(False)
      plt.show()
```

Then after preprocess the image in the model the model correctly predicted this single desired image.

```
[63]:  #Resize it to 32x32 pixels
      image = cv2.resize(image, (32,32))

      # Normalize the image
      image = (image-mean)/(std+1e-7)

      # Add an extra dimension because the model expects a batch of images
      image = image.reshape((1, 32, 32, 3))
```

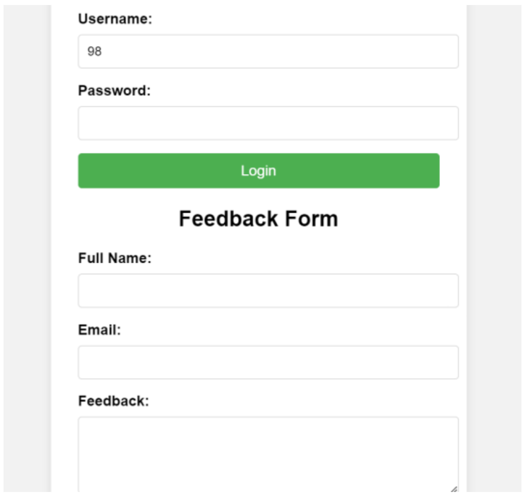
```
[64]: prediction = model.predict(image)

      1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 356ms/step
```

```
[68]: predicted_class = prediction.argmax()

      print('Predicted class: ', class_names[predicted_class])

      Predicted class:  truck
```

Deployment:

After the model is trained and save with the help of pickle .

We have create a form that can collect user  details and feedback.
The deployment of a web API using the Flask framework to create a feedback form. Flask is a micro web framework in Python that is easy to set up and well-suited for small to medium-sized applications. For this project, a virtual environment was employed to manage dependencies and ensure a clean environment for the API. The API is designed to collect user feedback via an HTML form, process the input, and store the feedback for later analysis.
Screenshot of the Login Page.


Username:

98

Password:

Feedback Form

Full Name:

Email:

Feedback:

Web link:

http://127.0.0.1:7010/templates/login.html