AlexNet

Project Purpose

The purpose of this project is to implement the **AlexNet Convolutional Neural Network (CNN)** architecture from scratch using the **CIFAR-10 dataset** to perform image classification. Through this implementation, the project aims to demonstrate core deep learning concepts such as:

- Building and training CNN models using **TensorFlow and Keras**
- Applying **data preprocessing and augmentation**
- Visualizing training metrics like **accuracy and loss**
- Understanding architectural components of classic CNNs like AlexNet
- Evaluating model performance using real-world image data

This project provides a foundational understanding of how deep learning models operate, while also showcasing practical techniques used in building image classification pipelines.

Implementation Details

- Framework Used: The entire project is implemented using TensorFlow and Keras, two of the most widely-used libraries for building deep learning models.
- Dataset: CIFAR-10, a labeled dataset of 60,000 32x32 color images across 10 classes.
- Preprocessing:
    - Images are standardized and resized to 64x64 pixels.
    - tf.data.Dataset API is used for efficient data loading and pipeline creation.
- Model Architecture:
    - Based on AlexNet, composed of 5 convolutional layers and 3 fully connected layers.

    - Includes Batch Normalization, MaxPooling, Dropout, and ReLU activation.

- Training:
    - Optimizer: Stochastic Gradient Descent (SGD) with a learning rate of 0.001.
    - Loss Function: Sparse Categorical Cross entropy
    - Model trained over 20 epochs, with training and validation metrics visualized.
- Evaluation:
    - Performance evaluated using accuracy.
    - Visualization of model predictions on sample test images.

CIFAR-10 Dataset

- **Description**: 60,000 32x32 color images in 10 classes (6,000 images per class)
- **Classes**: Aircraft, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks
- **Train/Test Split**: 50,000 training images and 10,000 test images
- **Challenges**: Small image size, diverse object appearances, complex backgrounds
- **Relevance**: A standard benchmark in computer vision that's complex enough to test CNN capabilities but small enough for educational purposes

---

Introduction

The main content of this report will present how the AlexNet Convolutional Neural Network(CNN) architecture is implemented using TensorFlow and Keras.

Here are some of the key learning objectives from this paper:

1. Introduction to neural network implementation with Keras and TensorFlow
1. Data preprocessing with TensorFlow
2. Training visualization with TensorBoard
3. Description of standard machine learning terms and terminologies

4. AlexNet Implementation

AlexNet AlexNet is a deep convolutional neural network (CNN) developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton.

Key Features: Architecture: 8 layers (5 convolutional + 3 fully connected)

Activation Function: Uses ReLU for faster training

Dropout: Prevents overfitting in fully connected layers

GPU Acceleration: Trained on two NVIDIA GTX 580 GPUs for efficiency

Data Augmentation: Used image translations and reflections to improve generalization AlexNet CNN is probably one of the simplest methods to approach understanding deep learning concepts and techniques.

AlexNet is not a complicated architecture when it is compared with some state of the art CNN architectures that have emerged in the more recent years.

AlexNet is simple enough for beginners and intermediate deep learning practitioners to pick up some good practices on model implementation techniques.

Libraries

```
import numpy as np  import
pandas as pd  import
tensorflow as tf from
tensorflow import keras
import matplotlib.pyplot as
plt import os import time

import os for dirname, _, filenames in
os.walk('/kaggle/input'):
    for filename in filenames:
print(os.path.join(dirname, filename))
```

```
/kaggle/input/cifar10-python/cifar-10-python.tar.gz
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_1
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_2
/kaggle/input/cifar10-python/cifar-10-batches-py/batches.meta
/kaggle/input/cifar10-python/cifar-10-batches-py/test_batch
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_3
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_5
/kaggle/input/cifar10-python/cifar-10-batches-py/data_batch_4
/kaggle/input/cifar10-python/cifar-10-batches-py/readme.html
```

Dataset

```
(train_images, train_labels), (test_images, test_labels) =
keras.datasets.cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 3s 0us/step
```

In order to reference the class names of the images during the visualization stage, a python list containing the classes is initialized with the variable name CLASS_NAMES.

```
CLASS_NAMES= ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
```

TensorFlow provides a suite of functions and operations that enables easy data manipulation and modification through a defined input pipeline.

To be able to access these methods and procedures, it is required that we transform our dataset into an efficient data representation TensorFlow is familiar with. This is achieved using the **tf.data.Dataset API**.

More specifically, **tf.data.Dataset.from_tensor_slices** method takes the train, test, and validation dataset partitions and returns a corresponding TensorFlow Dataset representation.

```
train_ds=tf.data.Dataset.from_tensor_slices((train_images,train_labels))
test_ds=tf.data.Dataset.from_tensor_slices((test_images,test_labels))
```

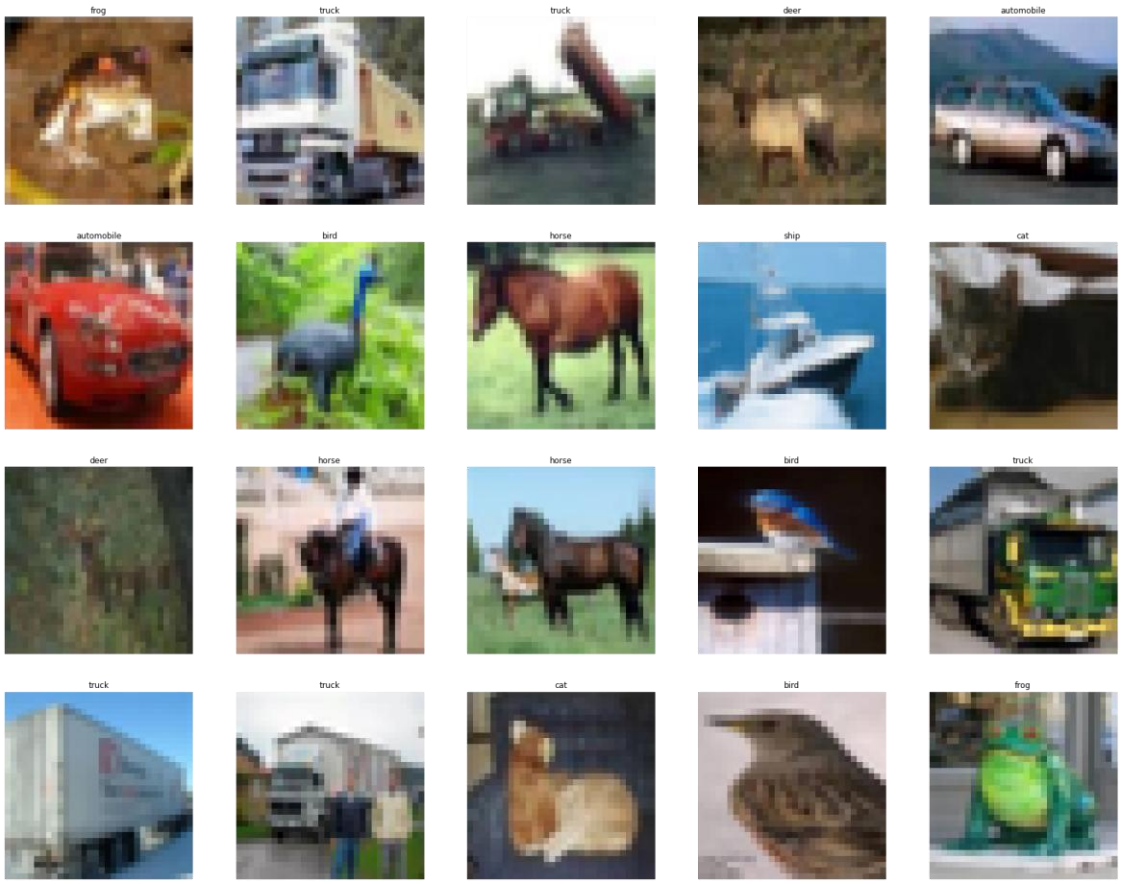
Preprocessing

Usually, preprocessing is conducted to ensure the data utilized is within an appropriate format.

First, let's visualize the images within the CIFAR-10 dataset.

The CIFAR-10 images have small dimensions, which makes visualization of the actual pictures a bit difficult.

```
plt.figure(figsize=(30,30)) for i,(image,label) in
enumerate(train_ds.take(20)):
    #print(label)
ax=plt.subplot(5,5,i+1)
plt.imshow(image)
plt.title(CLASS_NAMES[label.numpy()[0]])
plt.axis('off')
```



We'll create a function called process_images.

This function will perform all preprocessing work that we require for the data. This function is called further down the machine learning workflow.

```
def process_image(image,label):
    image=tf.image.per_image_standardization(image)
image=tf.image.resize(image,(64,64))
        return
image,label
```

Data Pipeline

So far, we have obtained and partitioned the dataset and created a function to process the dataset. The next step is to build an input pipeline.

An input/data pipeline is described as a series of functions or methods that are called consecutively one after another.

Input pipelines are a chain of functions that either act upon the data or enforces an operation on the data flowing through the pipeline.

```
train_ds_size=tf.data.experimental.cardinality(train_ds).numpy()
test_ds_size=tf.data.experimental.cardinality(test_ds).numpy()
print('Train size:',train_ds_size) print('Test
size:',test_ds_size)
```

```
Train size: 50000 Test
size: 10000
```

For our basic input/data pipeline, we will conduct three primary operations:

1. Preprocessing the data within the dataset

2. Shuffle the dataset

3. Batch data within the dataset

```
  train_ds=(train_ds
          .map(process_image)
          .shuffle(buffer_size=train_ds_size)
          .batch(batch_size=32,drop_remainder=True)
         ) test_ds=(test_ds
          .map(process_image)
          .shuffle(buffer_size=test_ds_size)
          .batch(batch_size=32,drop_remainder=True)
)
```

Model Implementation

Within this section, we will implement the AlexNet CNN architecture from scratch. Through the utilization of Keras Sequential API, we can implement consecutive neural network layers within our models that are stacked against each other.

Here are the types of layers the AlexNet CNN architecture is composed of, along with a brief description:

Convolutional layer:

A convolution is a mathematical term that describes a dot product multiplication between two sets of elements. Within deep learning the convolution operation acts on the filters/kernels and image data array within the convolutional layer. Therefore a convolutional layer is simply a layer the houses the convolution operation that occurs between the filters and the images passed through a convolutional neural network.

Batch Normalisation layer:

Batch Normalization is a technique that mitigates the effect of unstable gradients within a neural network through the introduction of an additional layer that performs operations on

the inputs from the previous layer. The operations standardize and normalize the input values, after that the input values are transformed through scaling and shifting operations.

MaxPooling layer:

Max pooling is a variant of sub-sampling where the maximum pixel value of pixels that fall within the receptive field of a unit within a sub-sampling layer is taken as the output. The max-pooling operation below has a window of 2x2 and slides across the input data, outputting an average of the pixels within the receptive field of the kernel.

Flatten layer:

Takes an input shape and flattens the input image data into a one-dimensional array.

Dense Layer:

A dense layer has an embedded number of arbitrary units/neurons within. Each neuron is a perceptron.

The code snippet represents the Keras implementation of the AlexNet CNN architecture.

```
model=keras.models.Sequential([
    keras.layers.Conv2D(filters=128, kernel_size=(11,11), strides=(4,4),
activation='relu', input_shape=(64,64,3)),
keras.layers.BatchNormalization(),
keras.layers.MaxPool2D(pool_size=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1),
activation='relu', padding="same"),
keras.layers.BatchNormalization(),
keras.layers.MaxPool2D(pool_size=(3,3)),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"),
keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(1,1), strides=(1,1),
activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(1,1), strides=(1,1),
activation='relu', padding="same"),
keras.layers.BatchNormalization(),
keras.layers.MaxPool2D(pool_size=(2,2)),     keras.layers.Flatten(),
    keras.layers.Dense(1024,activation='relu'),
keras.layers.Dropout(0.5),
    keras.layers.Dense(1024,activation='relu'),
keras.layers.Dropout(0.5),
    keras.layers.Dense(10,activation='softmax')


])
```

Training and Results

To train the network, we have to compile it.

The compilation processes involve specifying the following items:

Loss function:

A method that quantifies 'how well' a machine learning model performs. The quantification is an output(cost) based on a set of inputs, which are referred to as parameter values. The parameter values are used to estimate a prediction, and the 'loss' is the difference between the predictions and the actual values.

Optimization Algorithm:

An optimizer within a neural network is an algorithmic implementation that facilitates the process of gradient descent within a neural network by minimizing the loss values provided via the loss function. To reduce the loss, it is paramount the values of the weights within the network are selected appropriately.

Learning Rate:

An integral component of a neural network implementation detail as it's a factor value that determines the level of updates that are made to the values of the weights of the network. Learning rate is a type of hyperparameter.

We can also provide a summary of the network to have more insight into the layer composition of the network by running the **model.summary()** function.

```
model.compile(
loss='sparse_categorical_crossentropy',
optimizer=tf.optimizers.SGD(lr=0.001),
metrics=['accuracy']
)
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
= conv2d (Conv2D)               (None, 14, 14, 128)       46592
_________________________________________________________________
_ batch_normalization (BatchNo (None, 14, 14, 128)         512
_________________________________________________________________
_ max_pooling2d (MaxPooling2D) (None, 7, 7, 128)             0
_________________________________________________________________
_ conv2d_1 (Conv2D)             (None, 7, 7, 256)         819456
_________________________________________________________________
_ batch_normalization_1 (Batch (None, 7, 7, 256)           1024
_________________________________________________________________
```

```
_ max_pooling2d_1 (MaxPooling2 (None, 2, 2, 256)              0
_________________________________________________________________
_ conv2d_2 (Conv2D)              (None, 2, 2, 256)           590080
_________________________________________________________________
_ batch_normalization_2 (Batch (None, 2, 2, 256)             1024
_________________________________________________________________
_ conv2d_3 (Conv2D)              (None, 2, 2, 256)           65792
_________________________________________________________________
_ batch_normalization_3 (Batch (None, 2, 2, 256)             1024
_________________________________________________________________
_ conv2d_4 (Conv2D)              (None, 2, 2, 256)           65792
_________________________________________________________________
_ batch_normalization_4 (Batch (None, 2, 2, 256)             1024
_________________________________________________________________
_ max_pooling2d_2 (MaxPooling2 (None, 1, 1, 256)              0
_________________________________________________________________
_ flatten (Flatten)              (None, 256)                 0
_________________________________________________________________
_ dense (Dense)                  (None, 1024)              263168
_________________________________________________________________
_ dropout (Dropout)              (None, 1024)                0
_________________________________________________________________
_ dense_1 (Dense)                (None, 1024)             1049600
_________________________________________________________________
_ dropout_1 (Dropout)            (None, 1024)                0
_________________________________________________________________
_ dense_2 (Dense)                (None, 10)                10250
=================================================================
=
Total params: 2,915,338
Trainable params: 2,913,034
Non-trainable params: 2,304
_________________________________________________________________
```

At this point, we are ready to train the network.

Training the custom AlexNet network is very simple with the Keras module enabled through TensorFlow. We simply have to call the **fit()** method and pass relevant arguments.

Epoch: This is a numeric value that indicates the number of time a network has been exposed to all the data points within a training dataset.

```
history=model.fit(
train_ds,     epochs=20,
validation_data=test_ds,
validation_freq=1 )
```

```
Epoch 1/20
1562/1562 [==============================] - 12s 8ms/step - loss: 2.1899 -
accuracy: 0.2356 - val_loss: 1.6931 - val_accuracy: 0.3973
Epoch 2/20
1562/1562 [==============================] - 12s 7ms/step - loss: 1.7993 -
```

```
accuracy: 0.3480 - val_loss: 1.5406 - val_accuracy: 0.4507
Epoch 3/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.6458 -
accuracy: 0.4035 - val_loss: 1.4642 - val_accuracy: 0.4761
Epoch 4/20
1562/1562 [==============================] - 12s 7ms/step - loss: 1.5332 -
accuracy: 0.4439 - val_loss: 1.3974 - val_accuracy: 0.4984
Epoch 5/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.4441 -
accuracy: 0.4799 - val_loss: 1.3233 - val_accuracy: 0.5252
Epoch 6/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.3631 -
accuracy: 0.5107 - val_loss: 1.2884 - val_accuracy: 0.5455
Epoch 7/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.2913 -
accuracy: 0.5391 - val_loss: 1.3037 - val_accuracy: 0.5391
Epoch 8/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.2293 -
accuracy: 0.5649 - val_loss: 1.2008 - val_accuracy: 0.5783
Epoch 9/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.1664 -
accuracy: 0.5842 - val_loss: 1.1956 - val_accuracy: 0.5813
Epoch 10/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.1110 -
accuracy: 0.6050 - val_loss: 1.1721 - val_accuracy: 0.5940
Epoch 11/20
1562/1562 [==============================] - 12s 8ms/step - loss: 1.0571 -
accuracy: 0.6265 - val_loss: 1.1659 - val_accuracy: 0.5910
Epoch 12/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.9977 -
accuracy: 0.6469 - val_loss: 1.1876 - val_accuracy: 0.5956
Epoch 13/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.9431 -
accuracy: 0.6690 - val_loss: 1.2044 - val_accuracy: 0.5878
Epoch 14/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.8930 -
accuracy: 0.6874 - val_loss: 1.1431 - val_accuracy: 0.6082
Epoch 15/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.8378 -
accuracy: 0.7046 - val_loss: 1.1847 - val_accuracy: 0.6060
Epoch 16/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.7839 -
accuracy: 0.7259 - val_loss: 1.1579 - val_accuracy: 0.6179
Epoch 17/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.7323 -
accuracy: 0.7432 - val_loss: 1.1991 - val_accuracy: 0.6015
Epoch 18/20
```

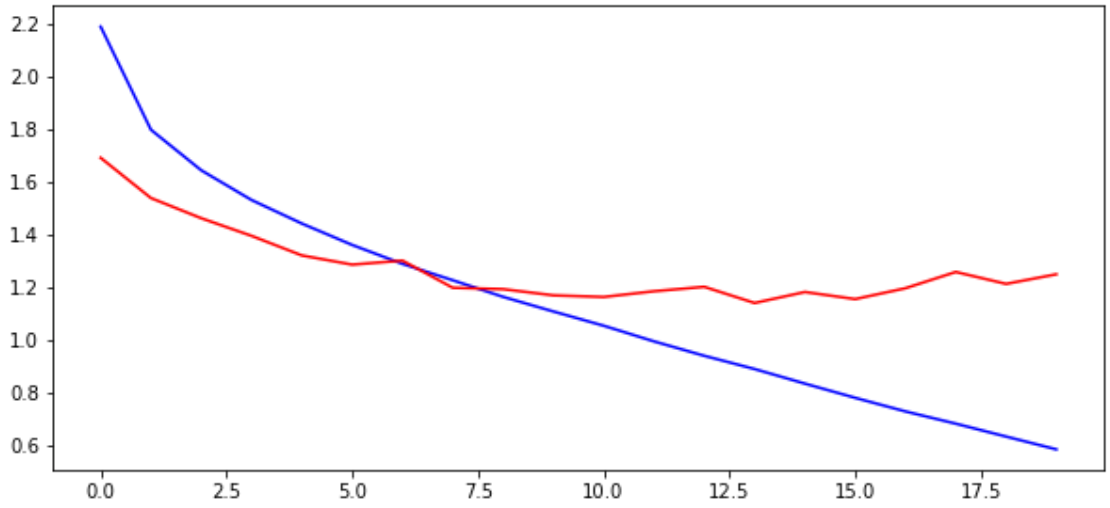
```
1562/1562 [==============================] - 12s 8ms/step - loss: 0.6860 -
accuracy: 0.7604 - val_loss: 1.2607 - val_accuracy: 0.6057
Epoch 19/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.6371 -
accuracy: 0.7738 - val_loss: 1.2155 - val_accuracy: 0.6103
Epoch 20/20
1562/1562 [==============================] - 12s 8ms/step - loss: 0.5886 -
accuracy: 0.7926 - val_loss: 1.2519 - val_accuracy: 0.6150
```

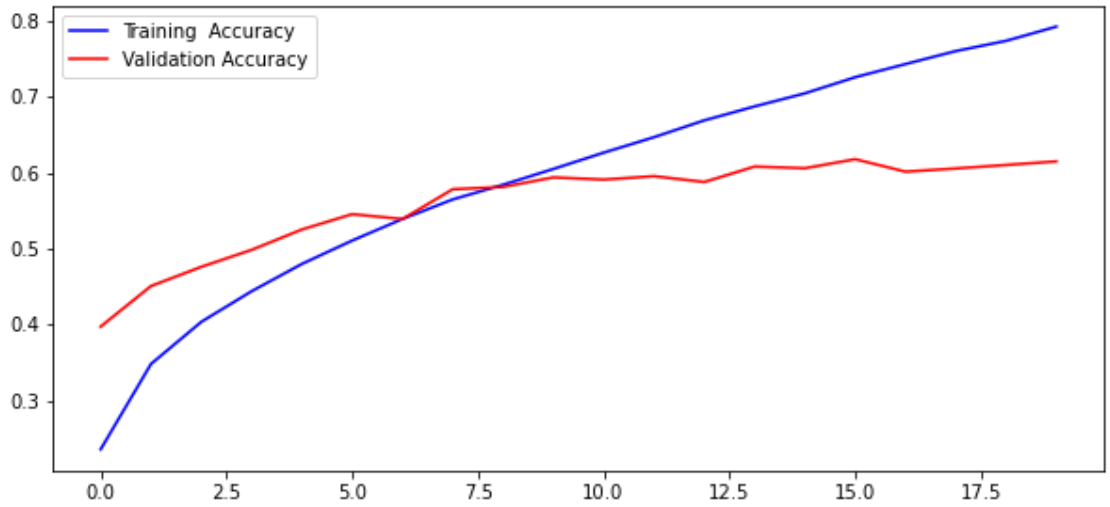
```
model.history.history.keys() dict_keys(['loss', 'accuracy', 'val_loss',
'val_accuracy'])
```

Now we will visualize the training over the different epochs .

```
f,ax=plt.subplots(2,1,figsize=(10,10))

#Assigning the first subplot to graph training loss and validation loss
ax[0].plot(model.history.history['loss'],color='b',label='Training Loss')
ax[0].plot(model.history.history['val_loss'],color='r',label='Validation
Loss')

#Plotting the training accuracy and validation accuracy
ax[1].plot(model.history.history['accuracy'],color='b',label='Training
Accuracy')
ax[1].plot(model.history.history['val_accuracy'],color='r',label='Validation
Accuracy')

plt.legend()
```

```
<matplotlib.legend.Legend at 0x7dd9300b6190>
```



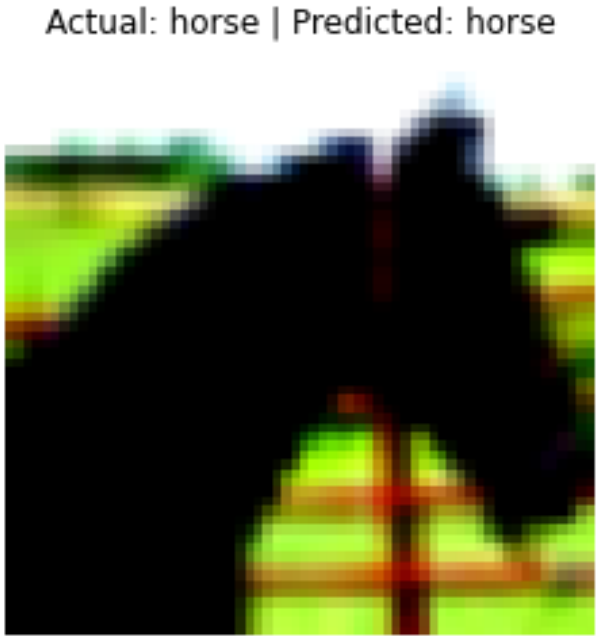



```
print('Accuracy Score = ',np.max(history.history['val_accuracy']))
```

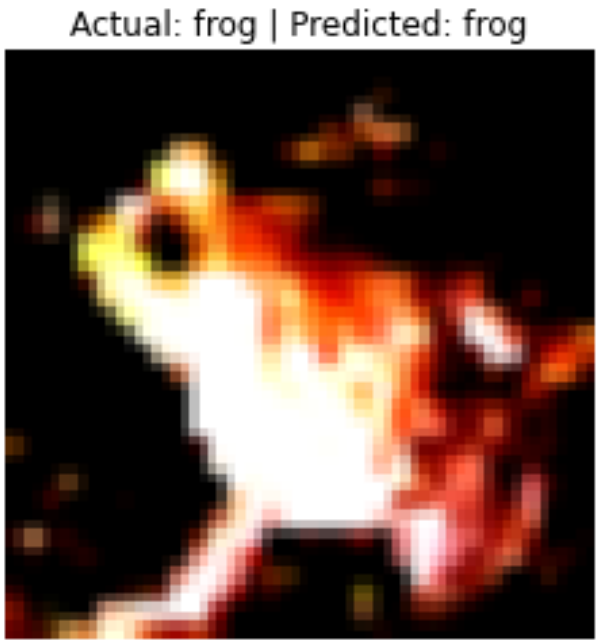
Accuracy Score = 0.6178886294364929

```
# Taking one sample from test dataset
for image, label in test_ds.take(1):
sample_image = image[0]
sample_label = label[0]    break

# Prepare image
sample_image_batch = tf.expand_dims(sample_image, axis=0)

# Predict predictions =
model.predict(sample_image_batch) predicted_class =
tf.argmax(predictions[0]).numpy() # Show image and
prediction
plt.imshow(sample_image.numpy().astype("uint8"))
```

```
plt.title(f"Actual:
{CLASS_NAMES[sample_label.numpy()[0]]} | Predicted:
{CLASS_NAMES[predicted_class]}")
plt.axis('off') plt.show()
```




```
# Taking one sample from test dataset for
image, label in test_ds.take(1):
    sample_image = image[0]
sample_label = label[0]     break

# Prepare image
sample_image_batch = tf.expand_dims(sample_image, axis=0)

# Predict predictions =
model.predict(sample_image_batch) predicted_class =
tf.argmax(predictions[0]).numpy()

# Show image and prediction
plt.imshow(sample_image.numpy().astype("uint8")) plt.title(f"Actual:
{CLASS_NAMES[sample_label.numpy()[0]]} | Predicted:
{CLASS_NAMES[predicted_class]}")
plt.axis('off') plt.show()
```



Conclusion

In this project, we successfully implemented the AlexNet Convolutional Neural Network architecture from scratch using TensorFlow and Keras, and applied it to the CIFAR-10 image classification dataset. Throughout the process, we covered essential deep learning concepts, including data preprocessing, efficient input pipelines, model construction with convolutional layers, batch normalization, max pooling, and fully connected layers.

Key takeaways from this work include:

- Understanding AlexNet's architecture and how each layer contributes to feature extraction and classification.
- Efficient data handling using the tf.data.Dataset API and preprocessing techniques such as standardization and resizing.
- Model training and evaluation, where we tracked performance across epochs and observed the network’s ability to learn and generalize.
- The visualization of predictions helped in understanding the model's real-world applicability and limitations.

While AlexNet may no longer be state-of-the-art, its relatively simple structure makes it an excellent choice for learning and understanding foundational deep learning principles. Future

improvements could include experimenting with more modern architectures, data augmentation strategies, or optimizer variations to enhance model accuracy and robustness.

Unsupervised pre-training

In the context provided, the authors anticipated significant benefits from unsupervised pre-training, particularly as computational power increases without a corresponding rise in the availability of labeled data. Although they chose not to use unsupervised pre-training in their current experiments for simplicity, they acknowledged its potential to enhance model performance in the future. Their results demonstrated that a large, deep convolutional neural network could achieve record-breaking results on challenging datasets using purely supervised learning. However, they noted that the performance of their network degrades when even a single convolutional layer is removed, emphasizing the importance of depth in achieving high accuracy. The authors believe that as they scale up their networks and potentially apply them to video data where temporal structure provides valuable information unsupervised pre-training could become a crucial component in further improving performance.