Simple Neural Network

What is a neural network?

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, artificial neural networks attempt to solve complicated problems, like summarizing documents or recognizing faces, with greater accuracy.

How do neural networks work?

The human brain is the inspiration behind neural network architecture. Human brain cells, called neurons, form a complex, highly interconnected network and send electrical signals to each other to help humans process information. Similarly, an artificial neural network is made of artificial neurons that work together to solve a problem. Artificial neurons are software modules, called nodes, and artificial neural networks are software programs or algorithms that, at their core, use computing systems to solve mathematical calculations.

Simple neural network architecture

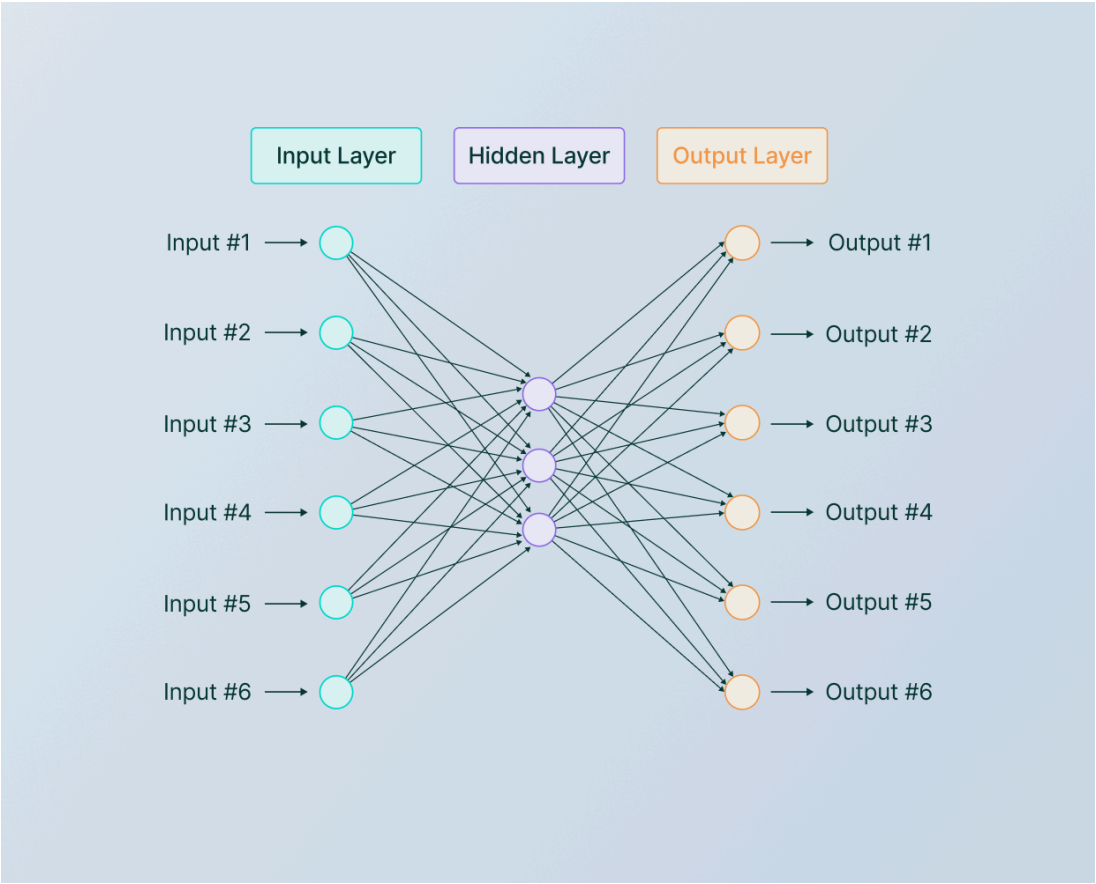
A basic neural network has interconnected artificial neurons in three layers:

Input Layer: Information from the outside world enters the artificial neural network from the input layer. Input nodes process the data, analyze or categorize it, and pass it on to the next layer.

Hidden Layer : Hidden layers take their input from the input layer or other hidden layers. Artificial neural networks can have a large number of hidden layers. Each hidden layer

analyzes the output from the previous layer, processes it further, and passes it on to the next layer.

Output Layer: The output layer gives the final result of all the data processing by the artificial neural network. It can have single or multiple nodes. For instance, if we have a binary (yes/no) classification problem, the output layer will have one output node, which will give the result as 1 or 0. However, if we have a multi-class classification problem, the output layer might consist of more than one output node.



Program for Simple Neural Network:

Step 1: Import Libraries

```
import numpy as np

import tensorflow as tf

from sklearn.datasets import make_classification

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

import matplotlib.pyplot as plt
```

- numpy: Library for numerical operations.
- tensorflow: Deep learning framework for building and training neural networks.
- make_classification: Function to generate a synthetic classification dataset.
- train_test_split: Utility for splitting the dataset into training and testing sets.
- StandardScaler: Used to standardize (normalize) the dataset.
- accuracy_score, confusion_matrix, classification_report: Metrics for evaluating the model.
- matplotlib.pyplot: Library for creating visualizations.

Step 2: Generate Synthetic Dataset

```
X, y = make_classification(n_samples=1000, n_features=10, random_state=38)
```

- make_classification: Generates a synthetic dataset with 1000 samples, 10 features, and a random seed for reproducibility.

Step 3: Split Dataset

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- train_test_split: Splits the dataset into training (80%) and testing (20%) sets.

Step 4: Standardize Features

```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

- StandardScaler: Standardizes features by removing the mean and scaling to unit variance.

Step 5: Define and Train the Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_split=0.1)
```

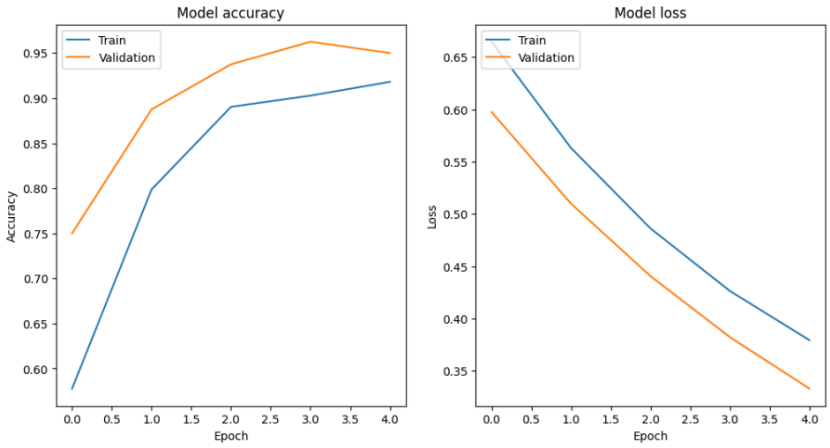
- Defines a simple neural network with one hidden layer (64 neurons, ReLU activation) and an output layer (1 neuron, sigmoid activation).
- Compiles the model with the Adam optimizer, binary cross-entropy loss, and accuracy metric.
- Trains the model on the training data for 5 epochs with a batch size of 32.

```
Epoch 1/5
23/23 [==============================] - 1s 12ms/step - loss: 0.6646 - accuracy: 0.5778 - val_loss: 0.5974 - val_accuracy: 0.7500
Epoch 2/5
23/23 [==============================] - 0s 3ms/step - loss: 0.5630 - accuracy: 0.7986 - val_loss: 0.5098 - val_accuracy: 0.8875
Epoch 3/5
23/23 [==============================] - 0s 4ms/step - loss: 0.4861 - accuracy: 0.8903 - val_loss: 0.4405 - val_accuracy: 0.9375
Epoch 4/5
23/23 [==============================] - 0s 4ms/step - loss: 0.4263 - accuracy: 0.9028 - val_loss: 0.3822 - val_accuracy: 0.9625
Epoch 5/5
23/23 [==============================] - 0s 4ms/step - loss: 0.3793 - accuracy: 0.9181 - val_loss: 0.3330 - val_accuracy: 0.9500
```

Step 6: Plot Training History

```
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val_accuracy'])

plt.title('Model accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.show()
```

- Plots the training and validation accuracy in one subplot and the training and validation loss in another subplot.



Step 7: Print Model Summary

```
model.summary()
```

- Prints a summary of the neural network architecture and the number of parameters.

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 64)                704

 dense_7 (Dense)             (None, 1)                 65

=================================================================
Total params: 769 (3.00 KB)
Trainable params: 769 (3.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Step 8: Evaluate the Model on the Test Set

```
y_pred_proba = model.predict(X_test)

y_pred = (y_pred_proba > 0.5).astype(int)

test_accuracy = accuracy_score(y_test, y_pred)

conf_matrix = confusion_matrix(y_test, y_pred)
```

```
classification_rep = classification_report(y_test, y_pred)

print(f"Test accuracy: {test_accuracy * 100:.2f}%")

print("Confusion Matrix:")

print(conf_matrix)

print("Classification Report:")

print(classification_rep)
```

```
7/7 [==============================] - 0s 2ms/step
Test accuracy: 91.00%
```

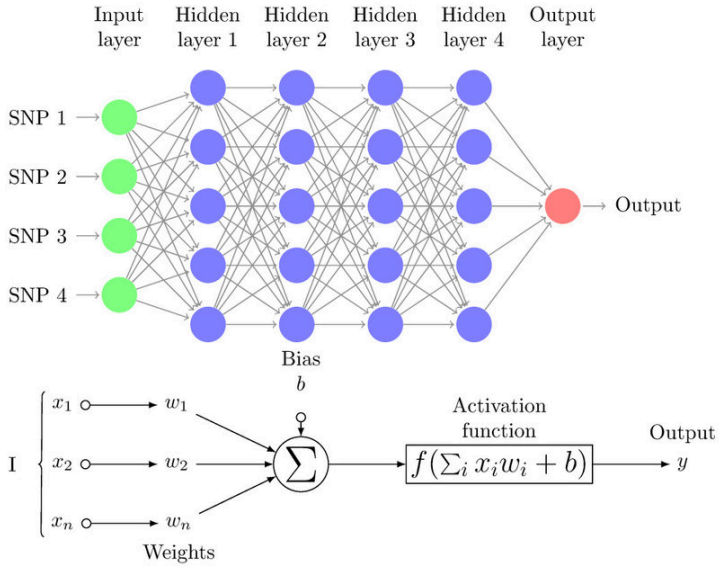
```
[ ] print("Confusion Matrix:")
    print(conf_matrix)

Confusion Matrix:
[[93 13]
 [ 5 89]]
```

```
[ ] print("Classification Report:")
    print(classification_rep)

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.88      0.91       106
           1       0.87      0.95      0.91        94

    accuracy                           0.91       200
   macro avg       0.91      0.91      0.91       200
weighted avg       0.91      0.91      0.91       200
```

- Evaluates the trained model on the test set, calculates predictions, and computes accuracy, confusion matrix, and classification report.

This code provides a comprehensive example of building, training, and evaluating a simple neural network for binary classification using TensorFlow and Keras. It's a good starting point for beginners to understand the workflow of creating and assessing a basic neural network.

Multilayer Perceptron



A multi-layered perceptron (MLP) is one of the most common neural network models used in the field of deep learning. Often referred to as a “vanilla” neural network, an MLP is simpler than the complex models of today’s era. However, the techniques it introduced have paved the way for further advanced neural networks. The multilayer perceptron (MLP) is used for a variety of tasks, such as stock analysis, image identification, spam detection, and election voting predictions.

The Basic Structure

A multi-layered perceptron consists of interconnected neurons transferring information to each other, much like the human brain. Each neuron is assigned a value. The network can be divided into three main layers.

Input Layer: This is the initial layer of the network which takes in an input which will be used to produce an output.

Hidden Layer(s) : The network needs to have at least one hidden layer. The hidden layer(s) perform computations and operations on the input data to produce something meaningful.

Output Layer: The neurons in this layer display a meaningful output.

Connections : The MLP is a feedforward neural network, which means that the data is transmitted from the input layer to the output layer in the forward direction. The connections between the layers are assigned weights. The weight of a connection specifies its importance. This concept is the backbone of an MLP's learning process. While the inputs take their values from the surroundings, the values of all the other neurons are calculated through a mathematical function involving the weights and values of the layer before it.

Backpropagation : Backpropagation is a technique used to optimize the weights of an MLP using the outputs as inputs. In a conventional MLP, random weights are assigned to all the connections. These random weights propagate values through the network to produce the actual output. Naturally, this output would differ from the expected output. The difference between the two values is called the error. Backpropagation refers to the process of sending this error back through the network, readjusting the weights automatically so that eventually, the error between the actual and expected output is minimized. In this way, the output of the current iteration becomes the input and affects the next output. This is repeated until the correct output is produced. The weights at the end of the process would be the ones on which the neural network works correctly.

Program For Multi Layer Perceptron

Import Libraries:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models, datasets

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

import matplotlib.pyplot as plt
```

- Import necessary libraries, including TensorFlow, scikit-learn metrics, and Matplotlib for visualization.

Load and Preprocess the MNIST Dataset:

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

train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

train_labels = tf.keras.utils.to_categorical(train_labels)

test_labels = tf.keras.utils.to_categorical(test_labels)
```

- Load the MNIST dataset and preprocess it by reshaping images and normalizing pixel values.

Build the Neural Network Model:

```
model = models.Sequential()

model.add(layers.Flatten(input_shape=(28, 28, 1)))

model.add(layers.Dense(128, activation='relu'))

model.add(layers.Dense(10, activation='softmax'))
```

- Create a simple neural network model with a Flatten layer, a dense hidden layer with ReLU activation, and an output layer with softmax activation.

Compile the Model:

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

- Compile the model using the Adam optimizer, categorical cross-entropy loss, and accuracy as the metric.

Train the Model:

```
history = model.fit(train_images, train_labels, epochs=5, batch_size=64,
validation_data=(test_images, test_labels))
```

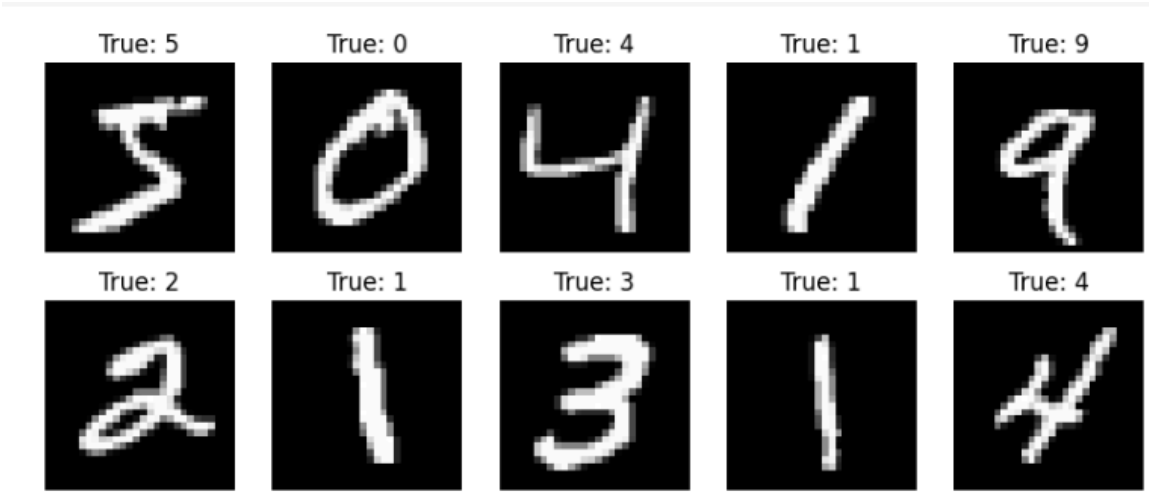
- Train the model for 5 epochs with a batch size of 64, using the training and validation data.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/5
938/938 [==============================] - 4s 4ms/step - loss: 0.3055 - accuracy: 0.9153 - val_loss: 0.1688 - val_accuracy: 0.9523
Epoch 2/5
938/938 [==============================] - 5s 5ms/step - loss: 0.1365 - accuracy: 0.9603 - val_loss: 0.1194 - val_accuracy: 0.9640
Epoch 3/5
938/938 [==============================] - 4s 4ms/step - loss: 0.0944 - accuracy: 0.9729 - val_loss: 0.1026 - val_accuracy: 0.9695
Epoch 4/5
938/938 [==============================] - 2s 2ms/step - loss: 0.0728 - accuracy: 0.9789 - val_loss: 0.0975 - val_accuracy: 0.9704
Epoch 5/5
938/938 [==============================] - 2s 3ms/step - loss: 0.0575 - accuracy: 0.9830 - val_loss: 0.0893 - val_accuracy: 0.9723
```

Visualize Input Data:

```
plt.figure(figsize=(10, 4))

for i in range(10):

    plt.subplot(2, 5, i + 1)

    plt.imshow(train_images[i].reshape(28, 28), cmap='gray')

    plt.title(f'True: {np.argmax(train_labels[i])}')

    plt.axis('off')

plt.show()
```

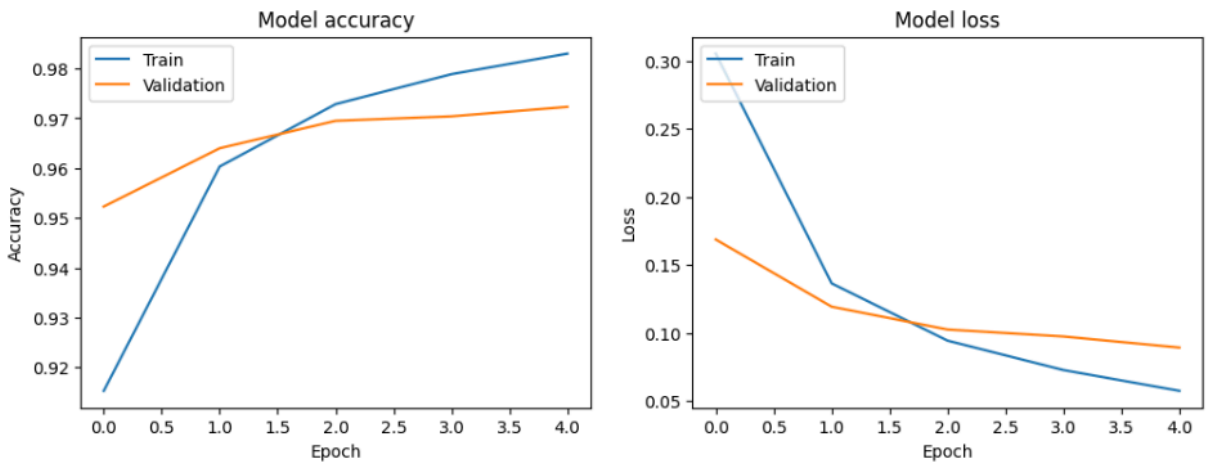
- Visualize the first 10 training images and their true labels.



Plot Training History:

```
plt.figure(figsize=(12, 4))

# Plot training & validation accuracy values

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train')

plt.plot(history.history['val_accuracy'], label='Validation')

plt.title('Model accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(loc='upper left')


# Plot training & validation loss values

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train')

plt.plot(history.history['val_loss'], label='Validation')

plt.title('Model loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(loc='upper left')


plt.show()
```

- Plot the training and validation accuracy, as well as the training and validation loss over epochs.



Evaluate the Model:

```
test_loss, test_accuracy = model.evaluate(test_images, test_labels)

print(f'Test accuracy: {test_accuracy * 100:.2f}%')
```

- Evaluate the model on the test set and print the test accuracy.

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0893 - accuracy: 0.9723
Test accuracy: 97.23%
```

Generate Predictions and Analyze Results:

```
y_pred_proba = model.predict(test_images)

y_pred = np.argmax(y_pred_proba, axis=1)



true_labels = np.argmax(test_labels, axis=1)



conf_matrix = confusion_matrix(true_labels, y_pred)

print("Confusion Matrix:")

print(conf_matrix)
```

```
class_report = classification_report(true_labels, y_pred)

print("Classification Report:")

print(class_report)
```

```
[ ]
# Confusion Matrix
conf_matrix = confusion_matrix(true_labels, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

```
313/313 [==============================] - 0s 1ms/step
Confusion Matrix:
[[ 973    0    1    2    0    0    0    3    1    0]
 [   1 1119    3    2    0    1    3    2    4    0]
 [   4    2  998    7    2    0    3   11    4    1]
 [   0    1    0  991    0    0    0    5    4    9]
 [   5    0    3    1  953    0    2    3    1   14]
 [   4    0    0   16    2  855    3    1    3    8]
 [   7    2    1    2    1    5  938    0    2    0]
 [   1    3    8    2    0    0    0 1008    0    6]
 [   8    0    7   11    8    7    5    8  908   12]
 [   4    3    0    5   10    0    1    6    0  980]]
```

```
[ ]
# Classification Report
class_report = classification_report(true_labels, y_pred)
print("Classification Report:")
print(class_report)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.97      0.97      1032
           3       0.95      0.98      0.97      1010
           4       0.98      0.97      0.97       982
           5       0.99      0.96      0.97       892
           6       0.98      0.98      0.98       958
           7       0.96      0.98      0.97      1028
           8       0.98      0.93      0.96       974
           9       0.95      0.97      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

- Generate predictions, convert one-hot encoded labels back to integers, and analyze the model's performance using a confusion matrix and a classification report.

Running in Google Colab:

1. Open Google Colab:
2. Go to Google Colab.
3. Create a New Notebook
4. Click on File -> New Notebook.
5. Copy and Paste
6. Copy the code above
7. Run Cells
8. Run each cell by clicking the play button or using Shift + Enter.

Running Locally:

1. Install Required Libraries:
    - Make sure you have TensorFlow, NumPy, Matplotlib, and scikit-learn installed.
    - **pip install numpy tensorflow scikit-learn matplotlib**
2. Download MNIST Dataset
    - Download the MNIST dataset from here and place it in the appropriate directory. https://www.kaggle.com/datasets/hojjatk/mnist-dataset

      `train-images-idx3-ubyte.gz`: Training set images (60,000 images)

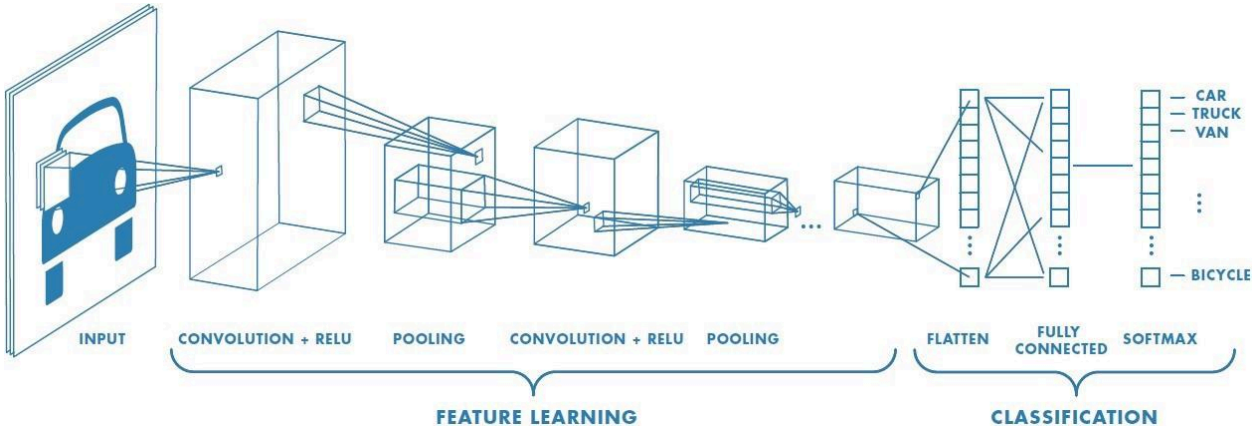
      `train-labels-idx1-ubyte.gz`: Training set labels (60,000 labels)

      `t10k-images-idx3-ubyte.gz`: Test set images (10,000 images)

      `t10k-labels-idx1-ubyte.gz`: Test set labels (10,000 labels)
3. Run Script:
    - Save the code in a Python script (e.g., mnist_classification.py) and run it using a Python interpreter.
    - **python mnist_classification.py**

Convolutional Neural Network (CNN)

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.



Now we will be using Convolutional Neural Network(CNN) to classify CIFAR images.

I will be using the Keras Sequential API, creating and training the model will take just a few lines of code.

Running Locally:

Step 1: Install Required Libraries

Make sure you have the necessary libraries installed. You can install them using the following command:

```
pip install tensorflow matplotlib
```

Step 2: Set Up Your Python Environment

Create a new Python script or Jupyter notebook in your preferred development environment.

Step 3: Copy the Code

Copy the provided code into your Python script or Jupyter notebook.

Step 4: Run the Code

Execute the code in your Python environment. This will download the CIFAR-10 dataset, build and train the CNN model, and evaluate its performance.

Step 5: Understand the Code

Review the code to understand the key components:

- Importing Libraries: TensorFlow and other necessary libraries are imported.
- Loading Data: The CIFAR-10 dataset is loaded and normalized.
- Creating the Convolutional Base: The CNN architecture is defined using the Keras Sequential API. The convolutional base consists of Conv2D and MaxPooling2D layers.
- Adding Dense Layers: Dense layers are added on top of the convolutional base for classification.
- Compiling and Training the Model: The model is compiled using the Adam optimizer and sparse categorical crossentropy loss. It is then trained on the training data for 10 epochs.
- Evaluating the Model: The model's performance is evaluated on the test data, and accuracy is printed.

Step 6: Understand the Output

After running the code, you should see the training progress and the final test accuracy. The plotted graph will show how the accuracy changes over epochs.

Additional Notes:

- If you're using a Jupyter notebook, make sure to run each cell sequentially.
- Adjust the number of epochs or other hyperparameters based on your preferences or computational resources.
- Feel free to experiment with the model architecture or try different optimization algorithms.
- Ensure that your machine has sufficient resources (CPU/GPU) to handle the training process, especially if you decide to modify the model or increase the number of epochs.

This guide should help you get started with running the CNN for image classification using the CIFAR-10 dataset on your local system.

1.1 Importing libraries

In [1]:

```
# Import TensorFlow

import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt
```

1.2 Loading data

The **CIFAR10** dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

In [2]:

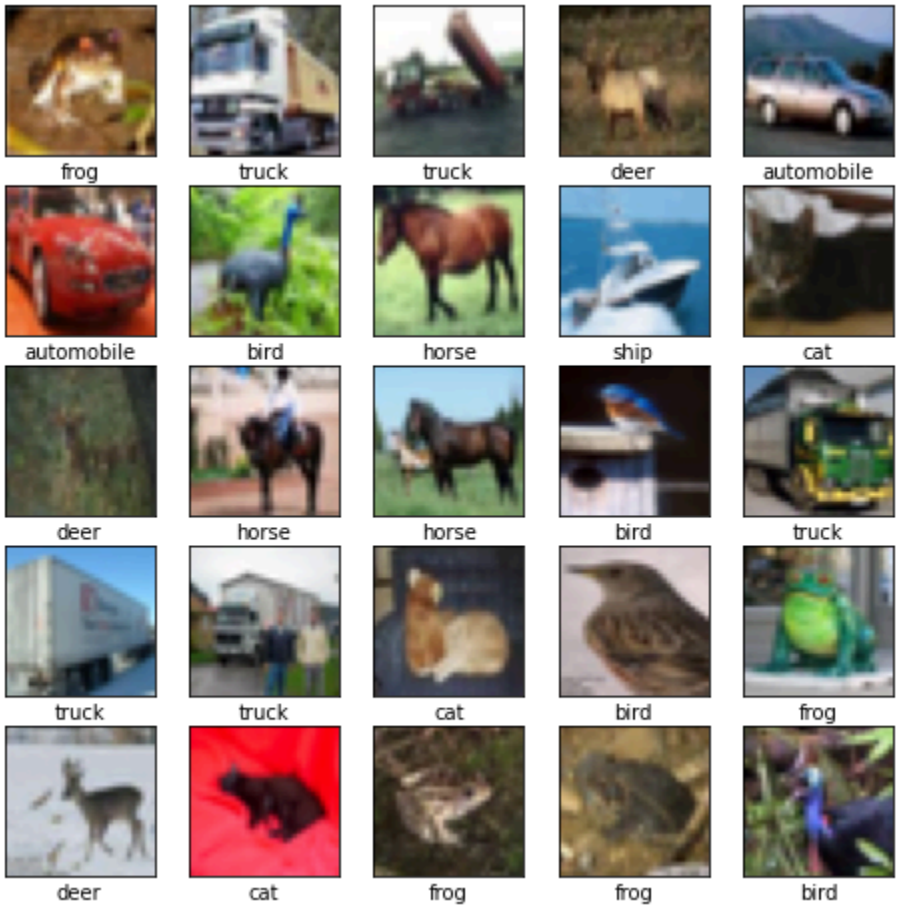
```
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()


# Normalize pixel values to be between 0 and 1

train_images, test_images = train_images / 255.0, test_images / 255.0
```

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image.

In [3]:

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',

               'dog', 'frog', 'horse', 'ship', 'truck']


plt.figure(figsize=(8,8))

for i in range(25):

    plt.subplot(5,5,i+1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    plt.imshow(train_images[i])
```

```
    # The CIFAR labels happen to be arrays,
    #which is why we need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



1.3 Create the convolutional base

The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2D layers.

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. If you are new to these dimensions, color_channels refers to (R,G,B). In this example, you will configure our CNN to process inputs of shape (32, 32, 3), which is the format of CIFAR images. You can do this by passing the argument **input_shape** to our first layer.

In [4]:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Now, let's display the architecture of our model.

In [5]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
```

```
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

Above, we can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64).

1.4 Adding Dense Layers on the top

To complete our model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 64)) into one or more **Dense layers** to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs.

In [6]:

```
model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

In [7]:

```
model.summary()
```

```
Model: "sequential"

_________________________________________________________________

Layer (type)                 Output Shape              Param #

=================================================================

conv2d (Conv2D)              (None, 30, 30, 32)        896

_________________________________________________________________

max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0

_________________________________________________________________

conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496

_________________________________________________________________

max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0

_________________________________________________________________

conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928

_________________________________________________________________

flatten (Flatten)            (None, 1024)              0
```

```
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

So our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

1.5 Compile and train the model

In [8]:

```
# Adam is the best among the adaptive optimizers in most of the cases
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# An epoch means training the neural network with all the
```

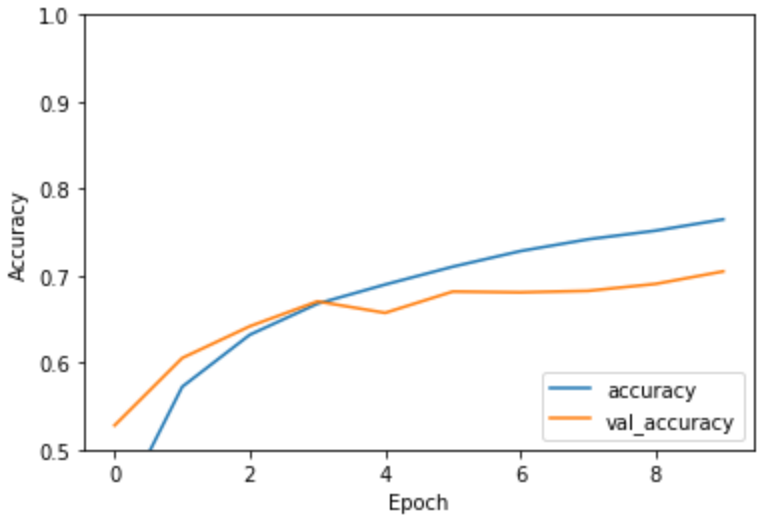
```
# training data for one cycle. Here I use 10 epochs

history = model.fit(train_images, train_labels, epochs=10,

                    validation_data=(test_images, test_labels))
```

1.6 Evaluate the model

In [9]:

```
plt.plot(history.history['accuracy'],label='accuracy')

plt.plot(history.history['val_accuracy'],label = 'val_accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.ylim([0.5, 1])

plt.legend(loc='lower right')


test_loss, test_acc = model.evaluate(test_images,

                                     test_labels,

                                     verbose=2)
```

```
313/313 - 2s - loss: 0.8843 - accuracy: 0.7049
```



In [10]:

```
print('Test Accuracy is',test_acc)
```

```
Test Accuracy is 0.7049000263214111
```

This simple CNN has achieved a test accuracy of over 70%, which is not bad

Simple NN:

https://colab.research.google.com/drive/1FNfZU4O3xfcdtlYs-8Lb84ajCRudcWE0?usp=sharing

Multilayer Perceptron:

https://colab.research.google.com/drive/1ilNgWOnRoch2nVzF2CG-ASZ8JsDn1p1q?usp=sharing

CNN

https://colab.research.google.com/drive/19wjvoqSb8p3J7052Dxkc0HvfXEkWKjhb?usp=sharing