Assignment No: 01

Aim: Study of Deep learning Packages: Tensorflow, Keras, Theano and PyTorch.Document the distinct features and functionality of the packages.

Problem Statement:

Study of following Deep learning Packages:

i. Keras

ii. PyTorch

iii.TensorFlow

iv. Theano

Theory:

A) Keras is an effective high-level neural network Application Programming Interface (API) written in Python. This open-source neural network library is designed to provide fast experimentation with deep neural networks, and it can run on top of CNTK, TensorFlow, and Theano. Keras focuses on being modular, user friendly, and extensible. It doesn't handle low-level computations; instead, it hands them off to another library called the Backend. Following are the distinct features of Keras,

- Keras is an API that was made to be easy to learn. It offers consistent & simple APIs, reduces the actions required to implement common code, and explains user error clearly.
- Prototyping time in Keras is less. This means that your ideas can be implemented and deployed in a shorter time. Keras also provides a variety of deployment options depending on user needs.
- Languages with a high level of abstraction and inbuilt features are slow and building custom features can be hard. But Keras runs on top of TensorFlow and is relatively fast. Keras is also deeply integrated with TensorFlow, so you can create customized workflows with ease.
- Keras is used commercially by many companies like Netflix, Uber, Square, Yelp, etc. which have deployed products in the public domain which are built using Keras.

Apart from this, Keras has features such as:

- It runs smoothly on both CPU and GPU.
- It supports almost all neural network models.
- It is modular in nature, which makes it expressive, flexible, and apt for innovative research.

Steps to build model in Keras

1. Define a network: In this step, you define the different layers in our model and the connections between them. Keras has two main types of models: Sequential and Functional models. You choose which type of model you want and then define the dataflow between them.
2. Compile a network: To compile code means to convert it in a form suitable for the machine to understand. In Keras, the model.compile() method performs this function. To compile the model, we define the loss function which calculates the losses in our model, the optimizer which reduces the loss, and the metrics which is used to find the accuracy of our model.
3. Fit the network: Using this, we fit our model to our data after compiling. This is used to train the model on our data.
4. Evaluate the network: After fitting our model, we need to evaluate the error in our model.
5. Make Predictions: We use model.predict() to make predictions using our model on new data.

B) PyTorch is an optimized Deep Learning tensor library based on Python and Torch and is mainly used for applications using GPUs and CPUs. PyTorch is favored over other Deep Learning frameworks like TensorFlow and Keras since it uses dynamic computation graphs and is completely Pythonic. It allows scientists, developers, and neural network debuggers to run and test portions of the code in real-time. Thus, users don't have to wait for the entire code to be implemented to check if a part of the code works or not.

The two main features of PyTorch are:

- Tensor Computation (similar to NumPy) with strong GPU (Graphical Processing) support
- Automatic Differentiation for creating and training deep neural networks

In machine learning, when we represent data, we need to do that numerically. A tensor is simply a container that can hold data in multiple dimensions. In mathematical terms, however, a tensor is a fundamental unit of data that can be used as the foundation for advanced mathematical operations. It can be a number, vector, matrix, or multi-dimensional array like Numpy arrays. Tensors can also be handled by the CPU or GPU to make operations faster. There are various types of tensors like Float Tensor,

Double Tensor, Half Tensor, Int Tensor, and Long Tensor, but PyTorch uses the 32-bit Float Tensor as the default type.

- Mathematical Operations

The codes to perform mathematical operations are the same in PyTorch as in Numpy. Users need to initialize two tensors and then perform operations like addition, subtraction, multiplication, and division on them.

- Matrix Initialization and Matrix Operations

To initialize a matrix with random numbers in PyTorch, use the function randn() that gives a tensor filled with random numbers from a standard normal distribution. Setting the random seed at the beginning will generate the same numbers every time you run this code. Basic matrix operations and transpose operation in PyTorch are also similar to NumPy

Common PyTorch Modules

In PyTorch, modules are used to represent neural networks.

- **Autograd**

The autograd module is PyTorch's automatic differentiation engine that helps to compute the gradients in the forward pass in quick time. Autograd generates a directed acyclic graph where the leaves are the input tensors while the roots are the output tensors.

- **Optim**

The Optim module is a package with pre-written algorithms for optimizers that can be used to build neural networks.

- **nn**

The nn module includes various classes that help to build neural network models. All modules in PyTorch subclass the nn module.

C) TensorFlow

TensorFlow is an open-source library developed by Google primarily for deep learning applications. It also supports traditional machine learning. TensorFlow was originally developed for large numerical computations without keeping deep learning in mind. However, it proved to be very useful for deep learning development as well, and therefore Google open-sourced it.TensorFlow accepts data in the form of multi-dimensional arrays of higher dimensions called tensors. Multi-dimensional arrays are very handy in handling large amounts of data.

TensorFlow works on the basis of data flow graphs that have nodes and edges. As the execution mechanism is in the form of graphs, it is much easier to execute TensorFlow code in a distributed manner across a cluster of computers while using GPUs.

Deep learning applications are very complicated, with the training process requiring a lot of computation. It takes a long time because of the large data size, and it involves several iterative processes, mathematical calculations, matrix multiplications, and so on. If you perform these activities on a normal Central Processing Unit (CPU), typically it would take much longer. Graphical Processing Units (GPUs) are popular in the context of games, where you need the screen and image to be of high resolution. GPUs were originally designed for this purpose. However, they are being used for developing deep learning applications as well. One of the major advantages of TensorFlow is that it supports GPUs, as well as CPUs. It also has a faster compilation time than other deep learning libraries, like Keras and Torch.TensorFlow allows you to create dataflow graphs that describe how data moves through a graph. The graph consists of nodes that represent a mathematical operation. A connection or edge between nodes is a multidimensional data array. It takes inputs as a multi-dimensional array where you can construct a flowchart of operations that can be performed on these inputs.

TensorFlow Architecture

Tensorflow architecture works in three significant steps:

- Data pre-processing - structure the data and brings it under one limiting value
- Building the model - build the model for the data
- Training and estimating the model - use the data to train the model and test it with unknown data

Introduction to Components of TensorFlow

Tensor: Tensor forms the core framework of TensorFlow. All the computations in TensorFlow involve tensors. It is a matrix of n-dimensions that represents multiple types of data. A tensor can be the result of a computation or it can originate from the input data.

Graphs: Graphs describe all the operations that take place during the training. Each operation is called an op node and is connected to the other. The graph shows the op nodes and the connections between the nodes, but it does not display values.

D) Theanois a Python library for fast numerical computation that can be run on the CPU or GPU.It is a key foundational library for Deep Learning in Python that you can use directly to create Deep Learning models or wrapper libraries that greatly simplify the process.

Theano is a Python library that allows us to evaluate mathematical operations including multi- dimensional arrays so efficiently. It is mostly used in building Deep Learning Projects. It works a way faster on Graphics Processing Unit (GPU) rather than on CPU. Theano attains high speeds that give a tough competition to C implementations for problems involving large amounts of data. It can take advantage of GPUs which makes it perform better than C on a CPU by considerable orders of magnitude under some certain circumstances. It knows how to take structures and convert them into very efficient code that uses numpy and some native libraries. It is mainly designed to handle the types of computation required for large neural network algorithms used in Deep Learning. That is why, it is a very popular library in the field of Deep Learning.

How to install Theano:

pip install theano

Several of the symbols we will need to use are in the **tensor** subpackage of Theano. We often import such packages with a handy name, let's say, T.

Theano is a sort of hybrid between numpy and sympy, an attempt is made to combine the two into one powerful library. Some advantages of theano are as follows import theano.tensor as T

- Stability Optimization: Theano can find out some unstable expressions and can use more stable means to evaluate them
- Execution Speed Optimization: As mentioned earlier, theano can make use of recent GPUs and execute parts of expressions in your CPU or GPU, making it much faster than Python
- Symbolic Differentiation: Theano is smart enough to automatically create symbolic graphs for computing gradients. It supports convolutional networks and recurrent networks, as well as combinations of the two

Conclusion:

Thus, we have studied the distinct features of Keras, TensorFlow, PyTorch and Theano python libraries used for Deep Learning Implementation.

Assignment No: 02

Aim: To Implement Feedforward neural networks with Keras and TensorFlow.

Problem Statement:

a. Import the necessary packages

b. Load the training and testing data (MNIST/CIFAR10)

c. Define the network architecture using Keras

d. Train the model using SGD

e. Evaluate the network

f. Plot the training loss and accuracy.

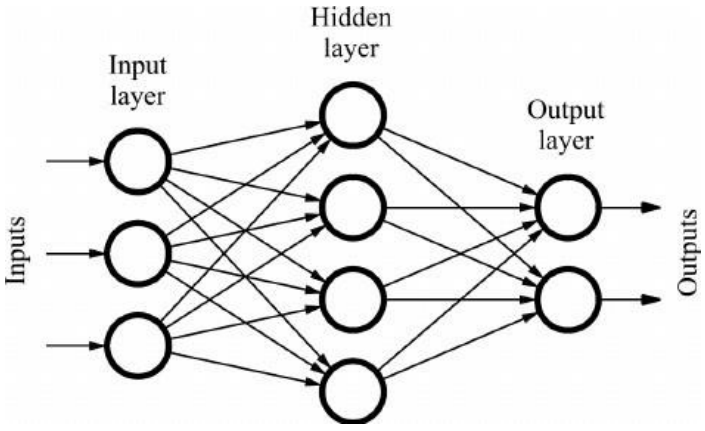
Objectives:

1) Apply feedforward neural network on MNIST/CIFAR dataset for classification
2) Evaluate the performance of models and plot the loss and accuracy graph

Theory:

A feedforward neural network is a type of artificial neural network in which nodes' connections do not form a loop. Often referred to as a multi-layered network of neurons, feedforward neural networks are so named because all information flows in a forward manner only.The data enters the input nodes, travels through the hidden layers, and eventually exits the output nodes. The network is devoid of links that would allow the information exiting the output node to be sent back into the network.

The following are the components of a feedforward neural network:



Layer of input

It contains the neurons that receive input. The data is subsequently passed on to the next tier. The input layer's total number of neurons is equal to the number of variables in the dataset.

Hidden layer

This is the intermediate layer, which is concealed between the input and output layers. This layer has a large number of neurons that perform alterations on the inputs. They then communicate with the output layer.

Output layer

It is the last layer and is depending on the model's construction. Additionally, the output layer is the expected feature, as you are aware of the desired outcome.

Neurons weights

Weights are used to describe the strength of a connection between neurons. The range of a weight's value is from 0 to 1.

Cost Function in Feedforward Neural Network

The cost function is an important factor of a feedforward neural network. Generally, minor adjustments to weights and biases have little effect on the categorized data points. Thus, to determine a method for improving performance by making minor adjustments to weights and biases using a smooth cost function.

The mean square error cost function is defined as follows:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Where,

w = weights collected in the network

b = biases

n = number of training inputs

a = output vectors

x = input

‖v‖ = usual length of vector v

Loss Function in Feedforward Neural Network

A neural network's loss function is used to identify if the learning process needs to be adjusted.

The cross-entropy loss for binary classification is as follows.

Cross Entropy Loss:

$$L(\Theta) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

The cross-entropy loss associated with multi-class categorization is as follows:

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Gradient Learning Algorithm

Gradient Descent Algorithm repeatedly calculates the next point using gradient at the current location, then scales it (by a learning rate) and subtracts achieved value from the current position (makes a step) (makes a step). It subtracts the value since we want to decrease the function (to increase it would be adding) (to maximize it would be adding). This procedure may be written as:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

There's a crucial parameter η which adjusts the gradient and hence affects the step size. In machine learning, it is termed learning rate and has a substantial effect on performance.

- The smaller the learning rate the longer GD converges or may approach maximum iteration before finding the optimal point
- If the learning rate is too great the algorithm may not converge to the ideal point (jump around) or perhaps diverge altogether.

CODE:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
import numpy as np

# Load and prepare the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
# Normalize the pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the feedforward neural network model
model = Sequential([
    # Flatten layer to transform the 28x28 image into a 1D vector of 784 pixels
    Flatten(input_shape=(28, 28)),

    # A hidden layer with 128 neurons and ReLU activation
    Dense(128, activation='relu'),

    # The output layer with 10 neurons (for digits 0-9) and Softmax activation for probabilities
    Dense(10, activation='softmax')
])

# Compile the model
# Using Adam optimizer, sparse_categorical_crossentropy loss for integer labels, and accuracy as a metric
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Print a summary of the model's architecture
model.summary()

# Train the model on the training data for 5 epochs
print("\nTraining the model...")
model.fit(x_train, y_train, epochs=5)

# Evaluate the model on the test data
print("\nEvaluating the model...")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc*100:.2f}%')

# Make predictions on the first few images of the test set
predictions = model.predict(x_test[:5])

# Print the predicted class and true class for the first image
predicted_class = np.argmax(predictions[0])
true_class = y_test[0]
print("\nPrediction for the first test image:")
print(f"Predicted class: {predicted_class}")
print(f"True class: {true_class}")
```

Conclusion:

Thus, we have implemented the Image classification using feed forward neural network model.During the execution of each iteration (epochs), our model'straining and validation data accuracy increases and loss decreases however significant change in the accuracy is observed after 80^{th} iteration (epoch).

Assignment No: 03

Aim: To Implement Image classification model using CNN Deep Learning Architecture.

Problem Statement:

Build the Image classification model using CNN Deep Learning Architecture by dividing the model into following 4 stages:

a. Loading and preprocessing the image data

b. Defining the model's architecture

c. Training the model

d. Estimating the model's performance

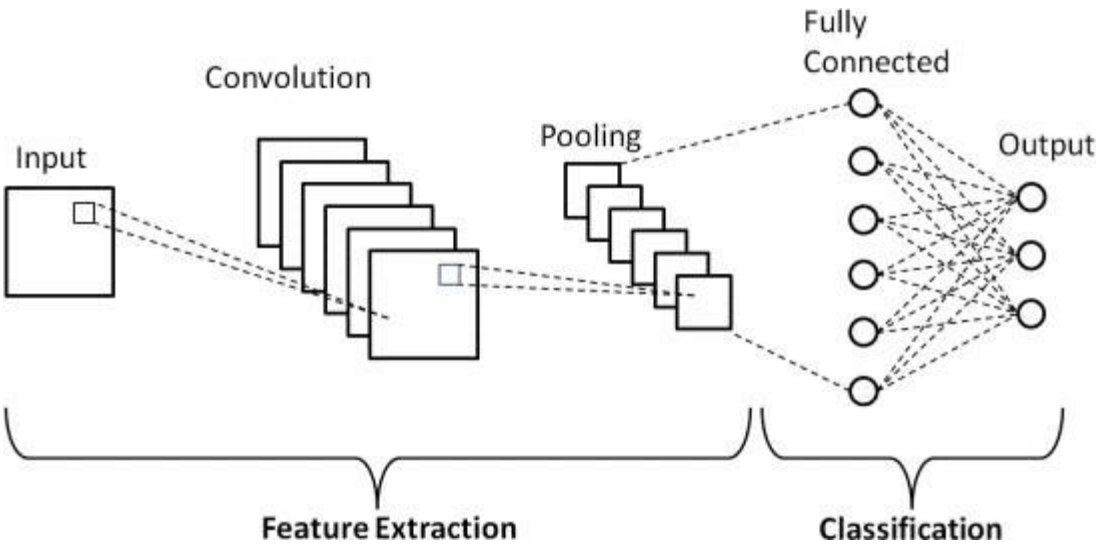
Objectives:

a) Apply CNN Deep Learning architecture on MNIST/CIFAR dataset for classification
b) Evaluate the performance of model in terms of accuracy, loss and number of epochs

Theory:

The goal of a CNN is to learn higher-order features in the data via convolutions. They are well suited to object recognition with images and consistently top image classification competitions. They can identify faces, individuals, street signs, platypuses, and many other aspects of visual data. CNNs overlap with text analysis via optical character recognition, but they are also useful when analyzing words6 as discrete textual units. They′re also good at analyzing sound. The efficacy of CNNs in image recognition is one of the main reasons why the world recognizes the power of deep learning. As Figure 4-7 illustrates, CNNs are good at building position and (somewhat) rotation invariant features from raw image data. CNNs are powering major advances in machine vision, which has obvious applications for self-driving cars, robotics, drones, and treatments for the visually impaired.

Architecture of CNN

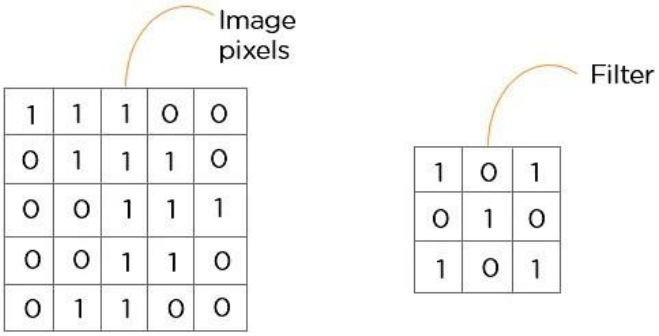


A convolution neural network has multiple hidden layers that help in extracting information from an image. The four important layers in CNN are:

1. Convolution layer
2. ReLU layer
3. Pooling layer
4. Fully connected layer

1. Convolution Layer

   This is the first step in the process of extracting valuable features from an image. A convolution layer has several filters that perform the convolution operation. Every image is considered as a matrix of pixel values. Consider the following 5x5 image whose pixel values are either 0 or 1. There's also a filter matrix with a dimension of 3x3. Slide the filter matrix over the image and compute the dot product to get the convolved feature matrix.

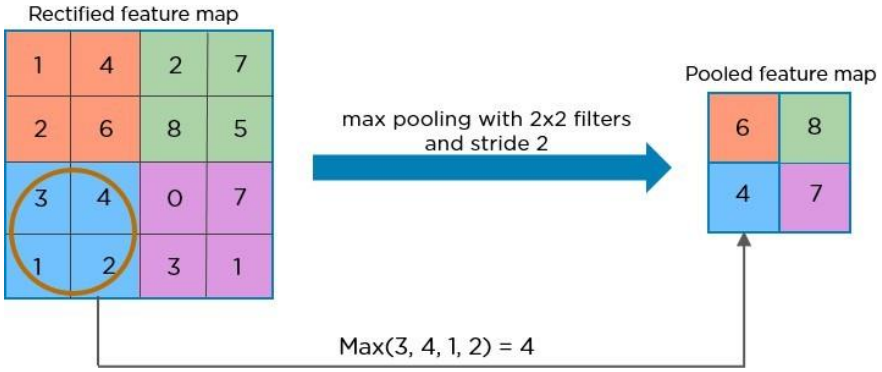


2. ReLU layer

   ReLU stands for the rectified linear unit. Once the feature maps are extracted, the next step is to move them to a ReLU layer. ReLU performs an element-wise operation and sets all the negative pixels to 0. It introduces non-linearity to the network, and the generated output is a rectified feature map. Below is the graph of a ReLU function:

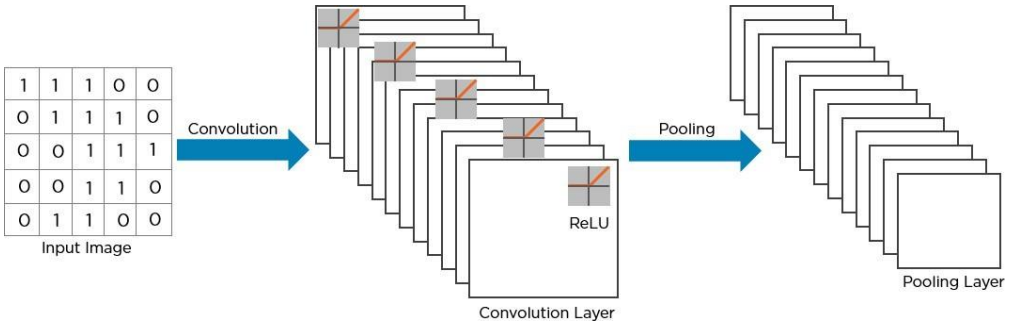
3. Pooling Layer

   Pooling is a down-sampling operation that reduces the dimensionality of the feature map. The rectified feature map now goes through a pooling layer to generate a pooled feature map.

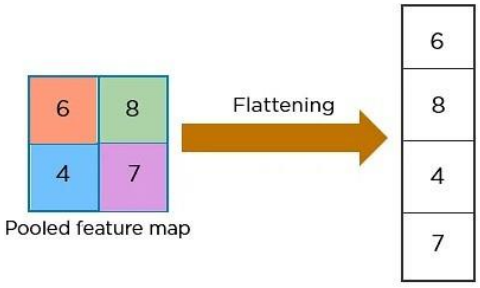


The pooling layer uses various filters to identify different parts of the image like edges, corners, body, feathers, eyes, and beak.

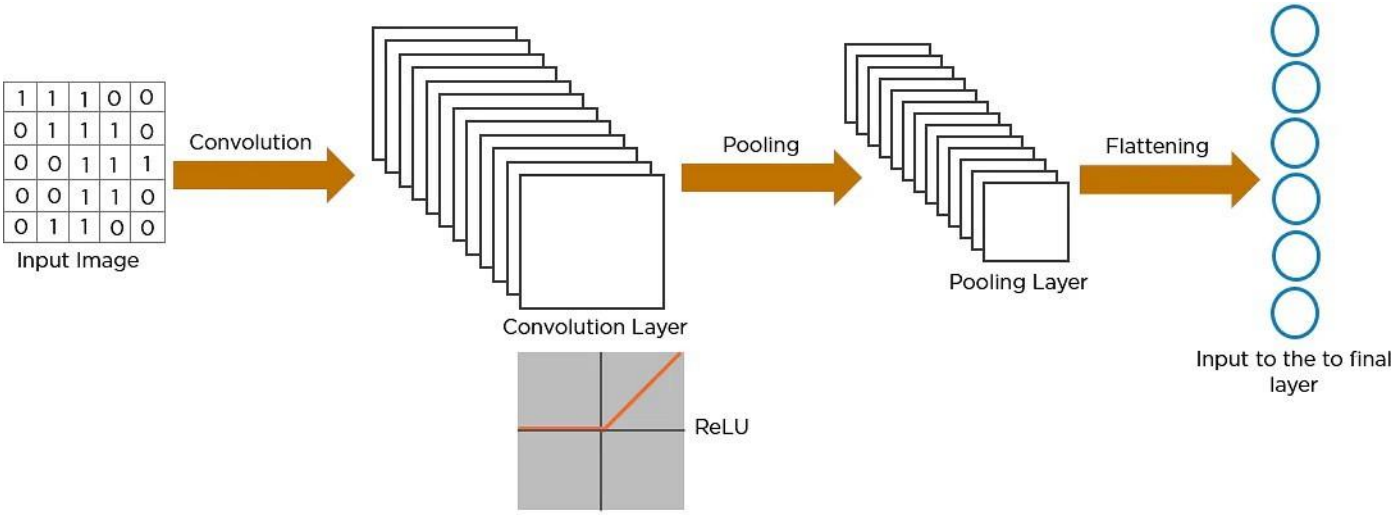
Here's how the structure of the convolution neural network looks so far:

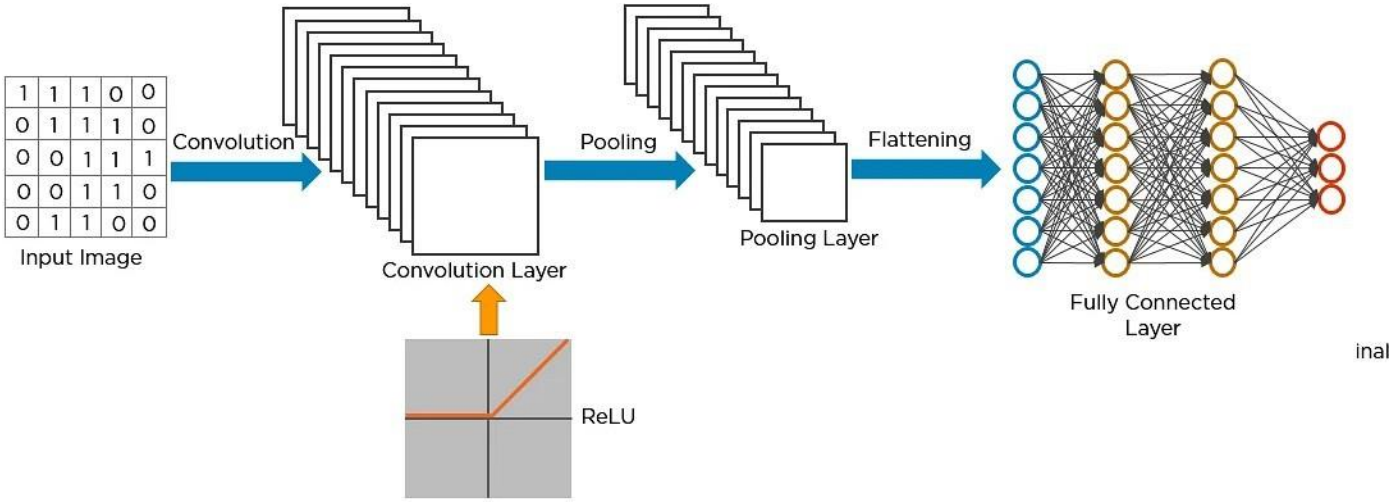


The next step in the process is called flattening. Flattening is used to convert all the resultant 2-Dimensional arrays from pooled feature maps into a single long continuous linear vector.The flattened matrix is fed as input to the fully connected layer to classify the image.



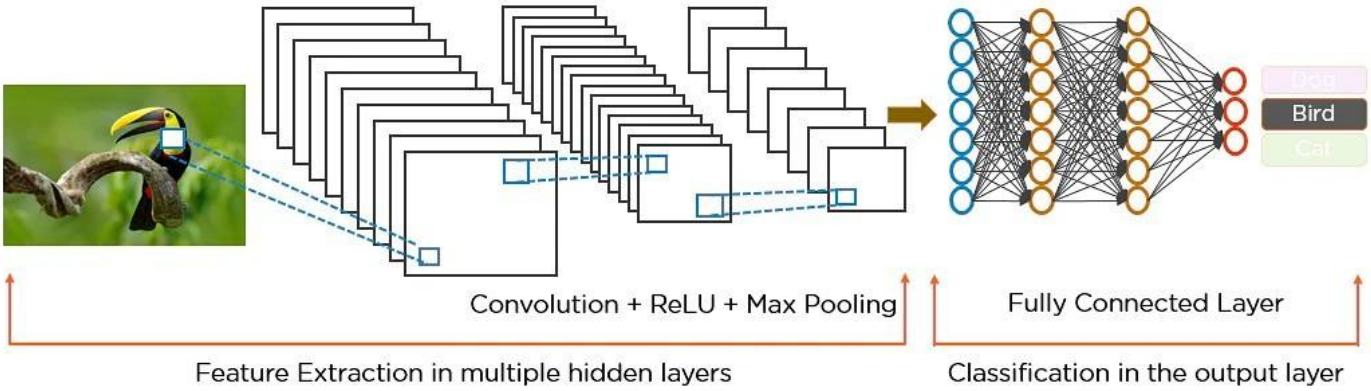
The flattened matrix is fed as input to the fully connected layer to classify the image.





- The pixels from the image are fed to the convolutional layer that performs the convolution operation
- It results in a convolved map
- The convolved map is applied to a ReLU function to generate a rectified feature map
- The image is processed with multiple convolutions and ReLU layers for locating the features
- Different pooling layers with various filters are used to identify specific parts of the image

- The pooled feature map is flattened and fed to a fully connected layer to get the final output



CODE:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import numpy as np

# 1. Load and Preprocess the Dataset
print("Loading and preparing the CIFAR-10 dataset...")
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values from [0, 255] to [0, 1]
x_train, x_test = x_train.astype('float32') / 255.0, x_test.astype('float32') / 255.0

# Convert class vectors to binary class matrices (one-hot encoding)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 2. Define the CNN Model Architecture
print("Defining the CNN model architecture...")
model = Sequential([
    # First Convolutional Block
    Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),

    # Second Convolutional Block
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),

    # Third Convolutional Block
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),

    # Flatten the 3D output to 1D before feeding it to the dense layers
```

```
Flatten(),

    # Fully Connected (Dense) Layers
    Dense(128, activation='relu'),

    # Output layer with 10 neurons (for 10 classes) and a softmax activation
    Dense(10, activation='softmax')
])

# Print a summary of the model
model.summary()

# 3. Compile the Model
print("\nCompiling the model...")
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 4. Train the Model
print("\nTraining the model...")
history = model.fit(x_train, y_train, epochs=15, batch_size=64, validation_data=(x_test, y_test))

# 5. Evaluate the Model
print("\nEvaluating the model...")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc*100:.2f}%')

# 6. Make a prediction on a single image
print("\nMaking a prediction on a single test image...")
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
# Expand the dimension of a single image to match the model's input shape
img = np.expand_dims(x_test[0], axis=0)
prediction = model.predict(img)
predicted_class = np.argmax(prediction)

print(f"The model predicts the image is a '{class_names[predicted_class]}'.")
print(f"The true class is a '{class_names[np.argmax(y_test[0])]}'.")
```

Conclusion:

Thus, we have implemented the Image classification model using CNN. With above code we can see that sufficient accuracy has been met. Throughout the epochs, our model accuracy increases and loss decreases that is good since our model gains confidence with our prediction

This indicates the model is trained in a good way

1. The loss is decreasing and the accuracy is increasing with every epoch.
2. The test accuracy is the measure of how good the model is predicting so, it is observed that the model is well trained after 10 epoch

Assignment No: 04

Aim: To implement Autoencoder for anomaly detection.

Problem Statement:

Use Autoencoder to implement anomaly detection. Build the model by using:

a. Import required libraries

b. Upload / access the dataset

c. Encoder converts it into latent representation

d. Decoder networks convert it back to the original input

e. Compile the models with Optimizer, Loss, and Evaluation Metrics.

Objectives:

a) Apply Autoencoder deep learning architecture to determine anomalies in input dataset

b) Evaluate Model.

Theory:

Autoencoders are very useful in the field of unsupervised machine learning. You can use them to compress the data and reduce its dimensionality. The main difference between Autoencoders and Principle Component Analysis (PCA) is that while PCA finds the directions along which you can project the data with maximum variance, Autoencoders reconstruct our original input given just a compressed version of it.If anyone needs the original data can reconstruct it from the compressed data using an Autoencoder. Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:

- Data-specific: Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip. So we can't expect an Autoencoder trained on handwritten digits to compress landscape photos.
- Lossy: The output of the Autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression, they are not the way to go.
- Unsupervised: To train an Autoencoder we don't need to do anything fancy, just throw the raw input data at it. Autoencoders are considered an *unsupervised* learning technique since they don't need explicit labels to train on. But to be more precise they are *self-supervised* because they generate their own labels from the training data.

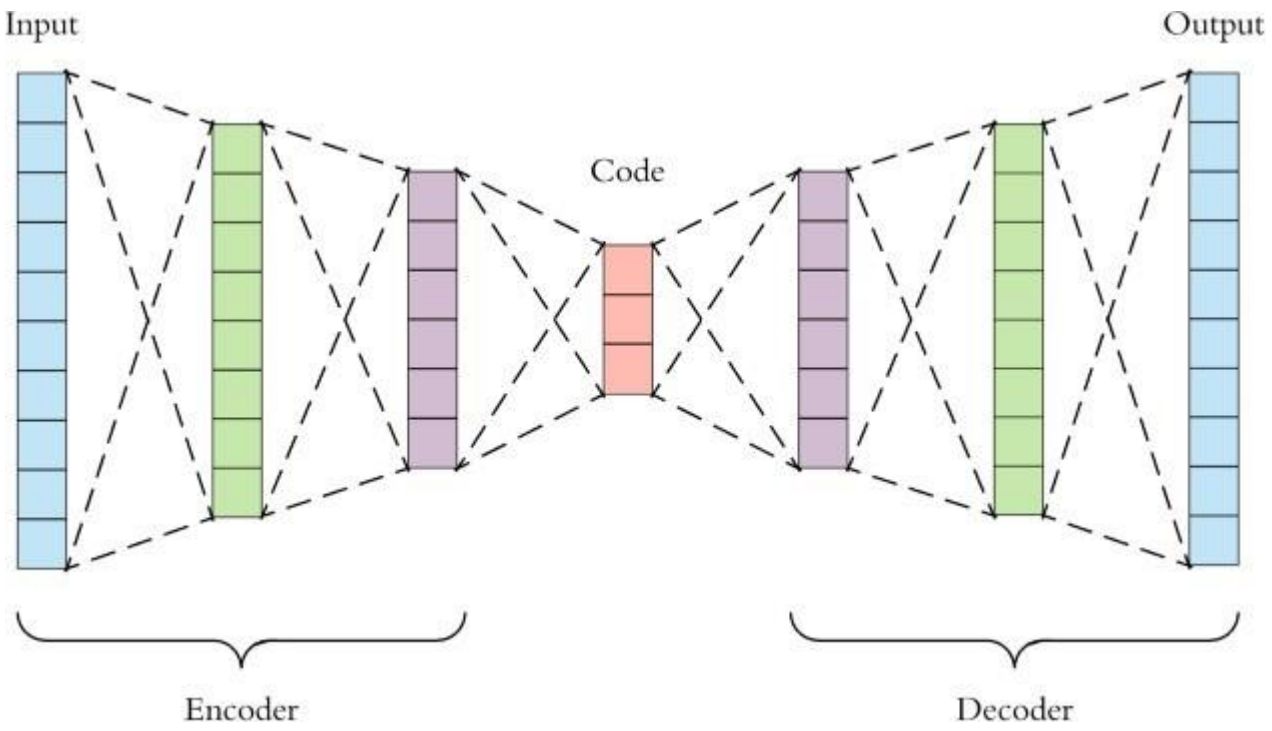
Architecture

An Autoencoder is a type of neural network that can learn to reconstruct images, text, and other data from compressed versions of themselves.

An Autoencoder consists of three layers:

1. Encoder
2. Code
3. Decoder

The Encoder layer compresses the input image into a latent space representation. It encodes the input image as a compressed representation in a reduced dimension. The compressed image is a distorted version of the original image.



The Code layer represents the compressed input fed to the decoder layer. The decoder layer decodes the encoded image back to the original dimension. The decoded image is reconstructed from latent space representation, and it is reconstructed from the latent space representation and is a lossy reconstruction of the original image.15

There are 4 hyperparameters that we need to set before training an Autoencoder:

- Code size: number of nodes in the middle layer. Smaller size results in more compression.
- Number of layers: the Autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- Number of nodes per layer: the Autoencoder architecture is called a *stacked Autoencoder* since the layers are stacked one after another. The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure. As noted above this is not necessary and we have total control over these parameters.
- Loss function: we either use *mean squared error (MSE)* or *binary cross entropy*. If the input values are in the range [0, 1] then we typically use cross entropy, otherwise we use the mean squared error.

Training of Autoencoder

Training of an Auto-encoder for data compression: For a data compression procedure, the most important aspect of the compression is the reliability of the reconstruction of the compressed data. This requirement dictates the structure of the Auto-encoder as a bottleneck.

Step 1: Encoding theinput data The Auto-encode first tries to encode the data using the initialized weights and biases.

Step 2: Decoding the input data The Auto-encoder tries to reconstruct the original input from the encoded data to test the reliability of the encoding.

Step 3: Backpropagating the error after the reconstruction, the loss function is computed to determine the reliability of the encoding. The error generated is backpropagated.

The above-described training process is reiterated several times until an acceptable level of reconstruction is reached.

After the training process, only the encoder part of the Auto-encoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are:-

- Keep small Hidden Layers: If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- Regularization: In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.
- Denoising: Another way of constraining the network is to add noise to the input and teach the network how to remove the noise from the data.

- Tuning the Activation Functions: This method involves changing the activation functions of various nodes so that a majority of the nodes are dormant thus effectively reducing the size of the hidden layers.

The different variations of Auto-encoders are:-

- Denoising Auto-encoder: This type of auto-encoder works on a partially corrupted input and trains to recover the original undistorted image. As mentioned above, this method is an effective way to constrain the network from simply copying the input.
- Sparse Auto-encoder: This type of auto-encoder typically contains more hidden units than the input but only a few are allowed to be active at once. This property is called the sparsity of the network. The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.
- Variational Auto-encoder: This type of auto-encoder makes strong assumptions about the distribution of latent variables and uses the Stochastic Gradient Variational Bayes estimator in the training process.

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt

# 1. Setup and Data Preparation
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize pixel values
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# Reshape the data to a 1D vector (784 pixels)
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

# Filter the dataset to train on only one class (e.g., digit '3')
# This is our 'normal' data
train_normal = x_train[y_train == 3]
test_normal = x_test[y_test == 3]

# 2. Build the Autoencoder Model
# Define the size of the encoded representation
encoding_dim = 32

# Input placeholder
input_img = Input(shape=(784,))

# Encoder part
```

```
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(encoding_dim, activation='relu')(encoded)

# Decoder part
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)

# The autoencoder model
autoencoder = Model(input_img, decoded)

# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='mse')

# Print the model summary
autoencoder.summary()

# 3. Train the Autoencoder
print("Training the autoencoder...")
autoencoder.fit(train_normal, train_normal,
            epochs=50,
            batch_size=256,
            shuffle=True,
            validation_data=(test_normal, test_normal))

# 4. Detect Anomalies and Visualize Results
# Create a test set with normal and anomalous data
normal_test_data = x_test[y_test == 3]
anomalous_test_data = x_test[y_test == 7]
test_data = np.concatenate([normal_test_data[:100], anomalous_test_data[:100]])
test_labels = np.array([0] * 100 + [1] * 100)  # 0 for normal, 1 for anomaly

# Get the reconstruction for the test data
reconstructions = autoencoder.predict(test_data)

# Calculate the mean squared error (reconstruction error) for each data point
mse = np.mean(np.power(test_data - reconstructions, 2), axis=1)

# Visualize the reconstruction error distribution
plt.figure(figsize=(10, 6))
plt.hist(mse[test_labels == 0], bins=50, alpha=0.7, label='Normal (Digit 3)')
plt.hist(mse[test_labels == 1], bins=50, alpha=0.7, label='Anomaly (Digit 7)')
plt.title('Reconstruction Error Distribution')
plt.xlabel('Reconstruction Error (MSE)')
plt.ylabel('Number of Samples')
plt.legend()
plt.show()

# Set a threshold to classify anomalies
```

```
threshold = np.mean(mse[test_labels == 0]) + 2 * np.std(mse[test_labels == 0])

print(f"\nThreshold for anomaly detection: {threshold:.4f}")
print(f"Number of anomalies detected (error > threshold): {np.sum(mse > threshold)}")
print(f"Total number of test samples: {len(test_data)}")
```

Conclusion:

Autoencoders can be used as an anomaly detection algorithm when we have an unbalanced dataset where we have a lot of good examples and only a few anomalies. Autoencoders are trained to minimize reconstruction error. When we train the Autoencoders on normal data or good data, we can hypothesis that the anomalies will have higher reconstruction errors than the good or normal data.

Assignment No: 05

Aim: Implement the continuous Bag of Words (CBOW) model for text recognition in given dataset.

Problem Statement:

Implement the Continuous Bag Of Words (CBOW) Model. Task to build the model are

.

a. Data preparation

b. Generate training data

c. Train model

d. Output

Objective:

- Build CBOW model to predict the current word given context words within a specific window.
- Evaluate Model.

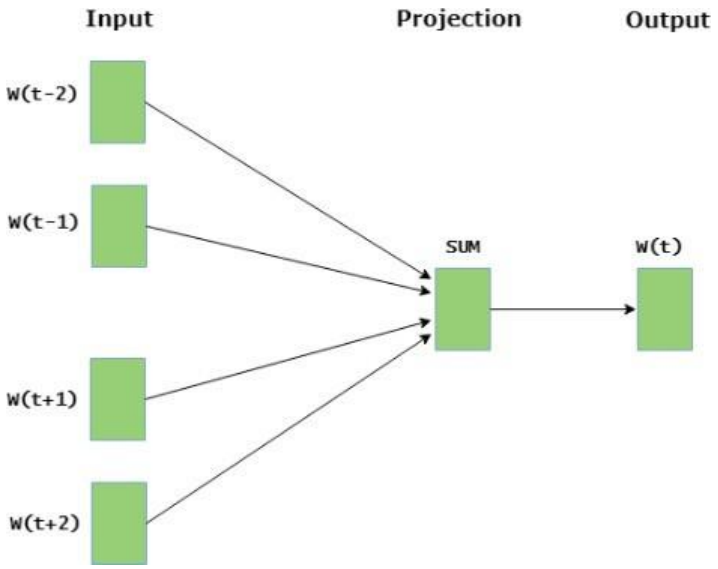
Theory:

Word Embedding is a language modeling technique used for mapping words to vectors of real numbers. Word embedding can be generated using various methods like neural networks, co-occurrence matrix, probabilistic models, etc. The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space. Word2Vec consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer. Word2Vec utilizes two architectures:

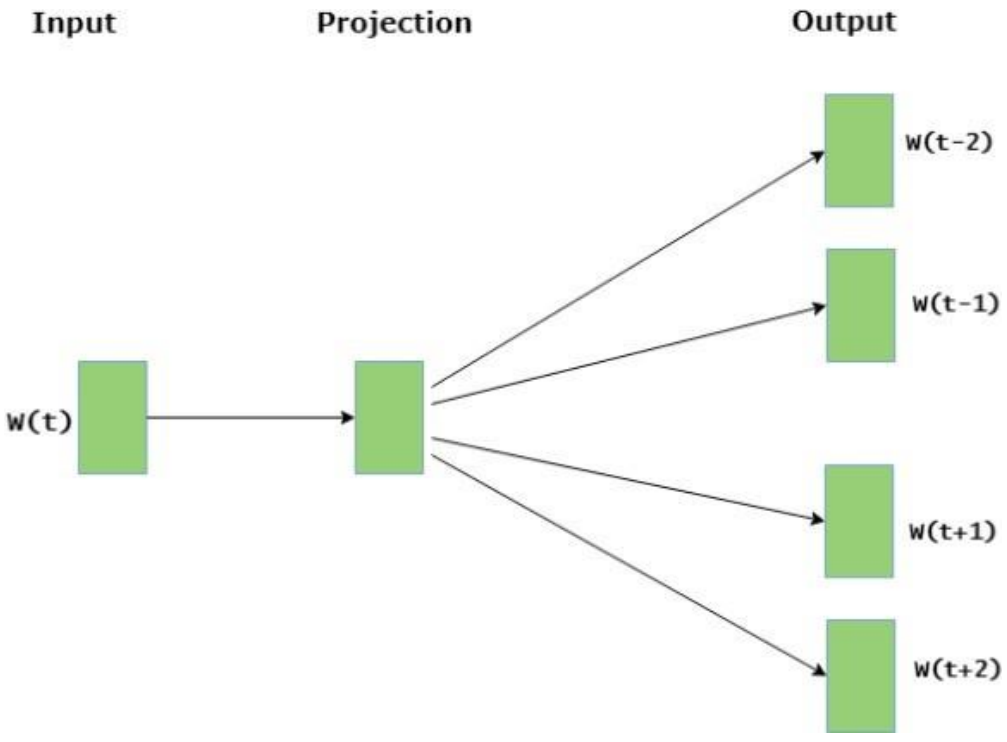
1. CBOW (Continuous Bag of Words)
2. Skip Gram

CBOW (Continuous Bag of Words): The CBOW model tries to understand the context of the words and takes this as input. It then tries to predict words that are contextually accurate.

1. CBOW model predicts the current word given context words within a specific window.
2. The input layer contains the context words and the output layer contains the current word.
3. The hidden layer contains the number of dimensions in which we want to represent the current word present at the output layer.



- **Skip Gram:** Skip gram predicts the surrounding context words within specific window given current word. The input layer contains the current word and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word present at the input layer.



The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space. For generating word vectors in Python, modules needed are nltk and gensim. Run these commands in terminal to install nltk and gensim:

- pip install nltk
- pip install gensim

Following are the steps to implement the CBOW Model,

Step 1: Download or collect the dataset from https://www.gutenberg.org/files/11/11-0.txt

Step 2: Remove all special characters and digits

Step 3: Remove all punctuation marks

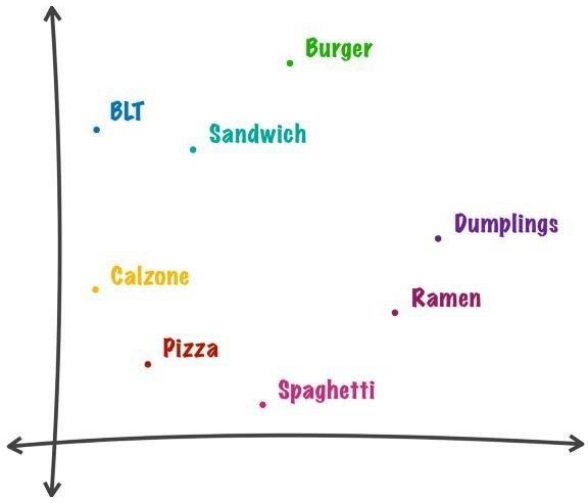
Step 4: Bring all letters to lower case

Step 5: Tokenize the document into sentences and words

Step 6: Remove all stop words.

Step 7: Train the model (Use Gensim word2vec for training the model)

To make words understood by machine learning algorithms, word embedding is used to map words into vectors of real numbers. There are various word embedding models and word2vec is one of them.In simple words, word2vec is a group of related models that are used to produce word embedding. These models are trained to construct the linguistic contexts of words. Word2vec takes a large corpus of text and produces a vector space, with each unique word in the corpus being assigned to a corresponding vector in the space.



To understand how word2vec works, let's explore some methods that we can use with word2vec such as finding similarities between words or solving analogies like this

$$f(\text{"canada"}) - f(\text{"us"}) = f(\text{"??"}) - f(\text{"hamburger"})$$

Step 8: Load the dataset and Model

Step 9: Find the most similar Words

Now we use model.most_similar() to find the top-N most similar words. Positive words contribute positively towards the similarity, negative words negatively. This method computes cosine similarity between a simple mean of the projection weight vectors of the given words and the vectors for each word in the model. The method corresponds to the word-analogy and distance scripts in the original word2vec implementation. If topn is False, most_similar returns the vector of similarity scores. restrict_vocab is an optional integer which limits the range of vectors which are searched for most-similar values. For example, restrict_vocab=10000 would only check the first 10000 word vectors in the vocabulary order. (This may be meaningful if you've sorted the vocabulary by descending frequency.)

Step 9: Predict the output word [Use predict_output_word(context_word_list,topn=10) function]

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Embedding, Lambda
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import backend as K

# --- 1. Prepare the Dataset ---
corpus = [
    "the quick brown fox jumps over the lazy dog",
    "the quick brown dog jumps over the lazy fox",
    "a man is walking on the street",
    "a dog is barking at the cat",
    "the street is busy with people"
]

# --- 2. Tokenize and Create Word-to-Index Mappings ---
tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
word_index = tokenizer.word_index
vocab_size = len(word_index) + 1
print(f"Vocabulary Size: {vocab_size}")

# --- 3. Generate CBOW Training Data ---
sequences = tokenizer.texts_to_sequences(corpus)
window_size = 2
data = []
```

```
for sequence in sequences:
    for i, word in enumerate(sequence):
        # Create a list for context words (left and right)
        context = []
        for j in range(i - window_size, i + window_size + 1):
            if j != i and 0 <= j < len(sequence):
                context.append(sequence[j])
        if context:
            data.append((context, word))

# Split into context words (X) and target word (y)
X_context = np.array([item[0] for item in data])
y_target = np.array([item[1] for item in data])

# Pad the context sequences to ensure uniform length
max_len = max(len(x) for x in X_context)
X_context_padded = pad_sequences(X_context, maxlen=max_len, padding='post')
y_target_one_hot = to_categorical(y_target, num_classes=vocab_size)

# --- 4. Build the CBOW Model ---
embedding_dim = 100

# Input for context words
input_context = Input(shape=(max_len,))

# Embedding layer to convert word indices to dense vectors
embedding_layer = Embedding(input_dim=vocab_size, output_dim=embedding_dim,
input_length=max_len)(input_context)

# Average the word vectors in the context
# The CBOW model sums or averages the context embeddings
lambda_layer = Lambda(lambda x: K.mean(x, axis=1), output_shape=(embedding_dim,))(embedding_layer)

# Output layer to predict the target word
output_layer = Dense(vocab_size, activation='softmax')(lambda_layer)

# Create the model
cbow_model = Model(inputs=input_context, outputs=output_layer)

# --- 5. Compile and Train the Model ---
cbow_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
print("\nTraining the CBOW model...")
cbow_model.fit(X_context_padded, y_target_one_hot, epochs=500, verbose=0)
print("Training complete.")

# --- 6. Visualize Word Embeddings (Optional) ---
# Get the learned word embeddings
word_embeddings = cbow_model.layers[1].get_weights()[0]

print("\nLearned Word Embeddings:")
```

```
for word, i in word_index.items():
    if i < vocab_size:
        print(f"Word: '{word}', Vector: {word_embeddings[i][:5]}...")
```

Conclusion:

Thus, we have implemented the CBOW model on collected dataset and found that word embedding approach is very useful for text recognition. CBOW can also be apply to convert the speech to text and has many more application in natural language processing.

Assignment No: 06

Aim: Implementation of object detection using transfer learning of CNN architectures

Problem Statement:

Implementation of object detection using transfer learning of CNN architecture.

a. Load in a pre-trained CNN model trained on a large dataset
b. Freeze parameters (weights) in model's lower convolutional layers
c. Add custom classifier with several layers of trainable parameters to model
d. Train classifier layers on training data available for task
e. Fine-tune hyperparameters and unfreeze more layers as needed

Objectives:

- To detect the objects in an image by implementing transfer learning deep learning approach using existing CNN architectures.

Theory:

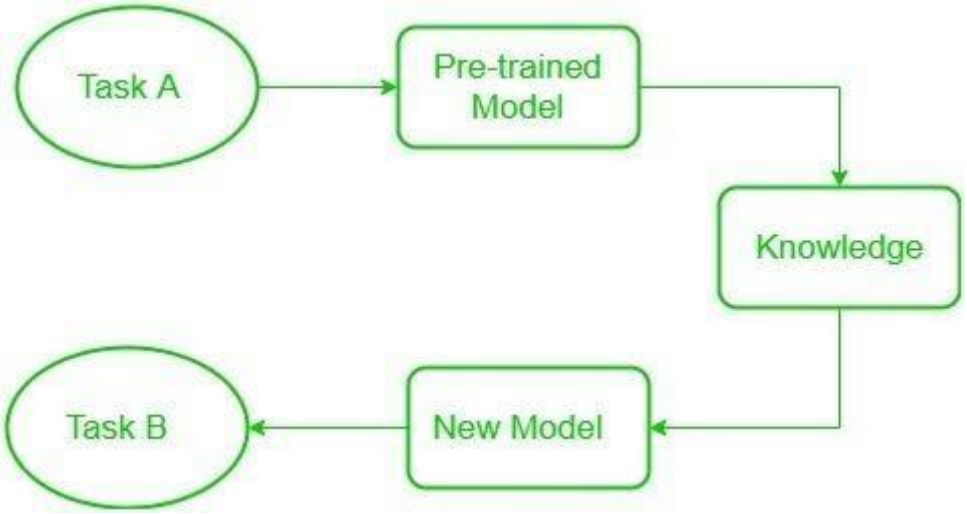
Whenever we encounter a new problem or a task, we recognize it and apply our relevant knowledge from our previous learning experiences. This makes our work easy and fast to finish. For instance, if you know how to ride a bicycle and if you are asked to ride a motorbike which you have never done before. In such a case, our experience with a bicycle will come into play and handle tasks like balancing the bike, steering, etc. This will make things easier compared to a complete beginner. Following the same approach, a term was introduced *Transfer Learning*. This approach involves the use of knowledge that was learned in some task and applying it to solve the problem in the related target task.

Need of transfer learning:

Many deep neural networks trained on images have a curious phenomenon in common: in the early layers of the network, a deep learning model tries to learn a low level of features, like detecting edges, colors, variations of intensities, etc. Such kind of features appears not to be specific to a particular dataset or a task because no matter what type of image we are processing either for detecting a lion or car. In both cases, we have to detect these low-level features. All these features occur regardless of the exact cost function or image dataset. Thus learning these features in one task of detecting lion can be used in other tasks like detecting humans.

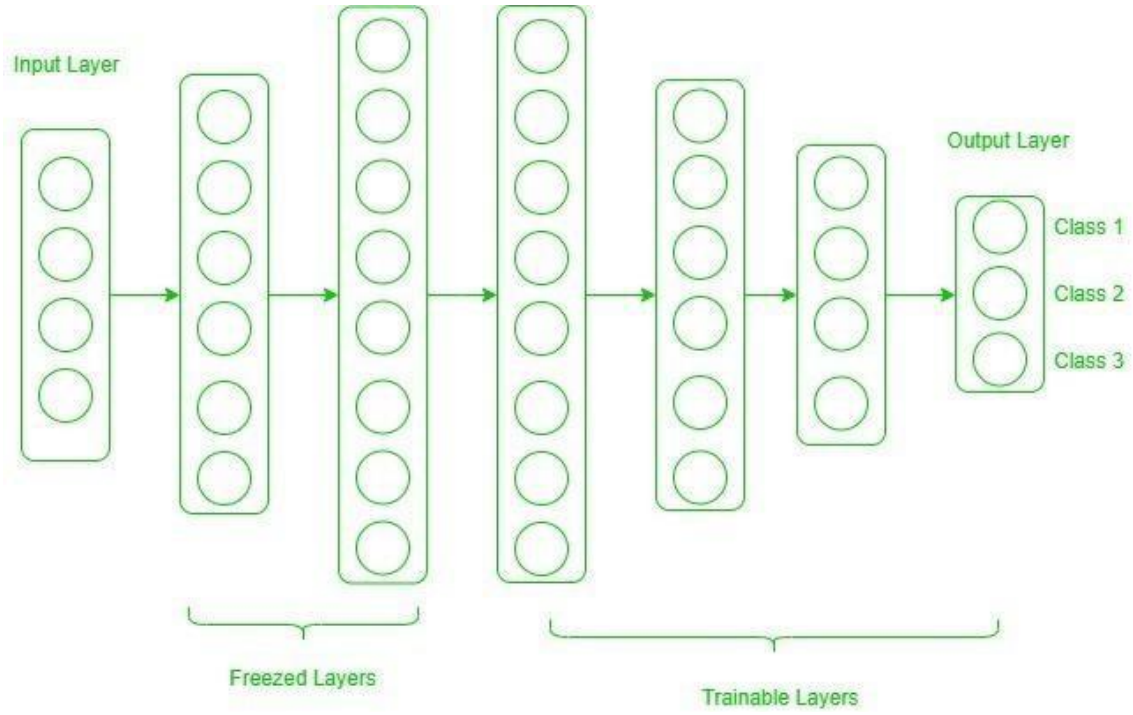
Necessity for transfer learning: Low-level features learned for task A should be beneficial for learning of model for task B. This is what transfer learning is. Nowadays, it is very hard to see people training a whole convolutional neural network from scratch, and it is common to use a pre-trained model trained on a variety of images in a similar task, e.g. models trained on ImageNet.

The Block diagram is shown below as follows:



When dealing with transfer learning, we come across a phenomenon called freezing of layers. A layer, it can be a CNN layer, hidden layer, a block of layers, or any subset of a set of all layers, is said to be fixed when it is no longer available to train. Hence, the weights of freezed layers will not be updated during training. While layers that are not freezed follows regular training procedure. When we use transfer learning in solving a problem, we select a pre-trained model as our base model. Now, there are two possible approaches to use knowledge from the pre-trained model. First way is to freeze a few layers of pre-trained model and train other layers on our new dataset for the new task. Second way is to make a new model, but also take out some features from the layers in the pre-trained model and use them in a newly created model. In both cases, we take out some of the learned features and try to train the rest of the model. This makes sure that the only feature that may be same in both of the tasks is taken out from the pre-trained model, and the rest of the model is changed to fit new dataset by training.

Freezed and Trainable Layers:



Now, one may ask how to determine which layers we need to freeze and which layers need to train. The answer is simple, the more you want to inherit features from a pre-trained model, the more you have to freeze layers. For instance, if the pre-trained model detects some flower species and we need to detect some new species. In such a case, a new dataset with new species contains a lot of features similar to the pre-trained model. Thus, we freeze less number of layers so that we can use most of its knowledge in a new model. Now, consider another case, if there is a pre-trained model which detects humans in images, and we want to use that knowledge to detect cars, in such a case where dataset is entirely different, it is not good to freeze lots of layers because freezing a large number of layers will not only give low level features but also give high-level features like nose, eyes, etc. which are useless for new dataset (car detection). Thus, we only copy low-level features from the base network and train the entire network on a new dataset. Let's consider all situations where the size and dataset of the target task vary from the base network.

Target dataset is small and similar to the base network dataset: Since the target dataset is small, that means we can fine-tune the pre-trained network with target dataset. But this may lead to a problem of overfitting. Also, there may be some changes in the number of classes in the target task. So, in such a case we remove the fully connected layers from the end, maybe one or two, and add a new fully-connected layer

satisfying the number of new classes. Now, we freeze the rest of the model and only train newly added layers.

Target dataset is large and similar to base training dataset: In such case when the dataset is large and it can hold a pre-trained model there will be no chance of overfitting. Here, also the last full-connected layer is removed, and a new fully-connected layer is added with the proper number of classes. Now, the entire model is trained on a new dataset. This makes sure to tune the model on a new large dataset keeping the model architecture the same.

Target dataset is small and different from the base network dataset: Since the target dataset is different, using high-level features of the pre-trained model will not be useful. In such a case, remove most of the layers from the end in a pre-trained model, and add new layers the satisfying number of classes in a new dataset. This way we can use low-level features from the pre-trained model and train the rest of the layers to fit a new dataset. Sometimes, it is beneficial to train the entire network after adding a new layer at the end.

Target dataset is large and different from the base network dataset: Since the target network is large and different, best way is to remove last layers from the pre-trained network and add layers with a satisfying number of classes, then train the entire network without freezing any layer.

Transfer learning is a very effective and fast way, to begin with, a problem. It gives the direction to move, and most of the time best results are also obtained by transfer learning.

CODE:

```
import os
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw, ImageFont

# --- 1. Prepare the Dataset ---
print("Downloading and preparing the dataset...")
# Use a smaller version for demonstration
dataset_name = "oxford_iiit_pet"
(ds_train, ds_test), ds_info = tfds.load(
    dataset_name, split=["train", "test"], with_info=True, data_dir='./tfds', shuffle_files=True
)

# Define label mapping and a simple preprocessing function
def preprocess(image, label):
    # Resize images to a standard size for the model
    image = tf.image.resize(image, (512, 512))
    # Normalize pixel values
    image = tf.cast(image, dtype=tf.float32) / 255.0
    return image, label
```

```
# Set up data pipelines
batch_size = 8
ds_train = ds_train.map(lambda x: (x['image'],
x['label'])).map(preprocess).batch(batch_size).prefetch(tf.data.AUTOTUNE)
ds_test = ds_test.map(lambda x: (x['image'],
x['label'])).map(preprocess).batch(batch_size).prefetch(tf.data.AUTOTUNE)

# --- 2. Load Pre-trained Model from TensorFlow Hub ---
print("Loading pre-trained EfficientDet-Lite0 model...")
# URL of the pre-trained model
model_handle = "https://tfhub.dev/tensorflow/efficientdet/lite0/detection/1"
# Load the model with its weights
detector = hub.load(model_handle)

# --- 3. Fine-tuning the Model (Dummy Example) ---
# Note: Full-scale fine-tuning requires more robust code and time.
# This part is a simplified demonstration of how the model can be used.
# In a real-world scenario, you would train a new head on top of the base model.
# The `detector` loaded above is a complete model, so we can use its `signatures` for inference.
print("Model loaded. Moving to inference as fine-tuning requires a more complex setup.")

# --- 4. Make a Prediction and Visualize Results ---
print("\nMaking a prediction on a test image...")
# Get a sample image and its ground truth labels
test_image_raw, test_label_raw = next(iter(ds_test.unbatch()))

# Convert the TensorFlow image to a NumPy array
test_image_np = test_image_raw.numpy()

# Make a single prediction
# The model expects a batch, so we add a new axis
input_tensor = tf.expand_dims(test_image_raw, 0)
detections = detector(input_tensor)

# Extract detection results
num_detections = int(detections['num_detections'][0].numpy())
detection_boxes = detections['detection_boxes'][0].numpy()[:num_detections]
detection_classes = detections['detection_classes'][0].numpy().astype(np.int64)[:num_detections]
detection_scores = detections['detection_scores'][0].numpy()[:num_detections]

# Convert the image back to a format for drawing
image_for_drawing = (test_image_np * 255).astype(np.uint8)
pil_image = Image.fromarray(image_for_drawing)
draw = ImageDraw.Draw(pil_image)

# Get the label names from the dataset info
label_names = ds_info.features['label'].names
print(f"Detected {num_detections} objects.")

# Loop through detections and draw bounding boxes
min_score_thresh = 0.5  # Adjust this threshold to filter detections
```

```
for i in range(num_detections):
    if detection_scores[i] > min_score_thresh:
        ymin, xmin, ymax, xmax = detection_boxes[i]
        class_id = detection_classes[i]
        score = detection_scores[i]

        # Convert normalized coordinates back to image coordinates
        (im_width, im_height) = pil_image.size
        (xmin, ymin, xmax, ymax) = (xmin * im_width, ymin * im_height,
                                    xmax * im_width, ymax * im_height)

        # Draw the bounding box
        draw.rectangle([(xmin, ymin), (xmax, ymax)], outline="red", width=3)

        # Get the label name and score
        label_text = f'{label_names[class_id]}: {score:.2f}'
        draw.text((xmin, ymin - 15), label_text, fill="red")

# Display the image with predictions
plt.figure(figsize=(8, 8))
plt.imshow(pil_image)
plt.title("Object Detection using Transfer Learning")
plt.axis('off')
plt.show()

print("\nCode execution complete.")
```

Conclusion:

Thus, we have studied to implement transfer learning deep learning approach whereby a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest. We also observed that transfer learning has the benefit of decreasing the training time for a neural network model and can result in lower generalization error.