**Practical No.1**

**Title: Study of Deep learning Packages: Tensorflow, Keras, Theano and PyTorch.**

**Document the distinct features and functionality of the packages.**

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')


# Import necessary libraries
from keras.datasets import mnist
from matplotlib import pyplot as plt


# Load the dataset
(train_X, train_y), (test_X, test_y) = mnist.load_data()


# Display the shape of the dataset
print('X_train:', train_X.shape)
print('Y_train:', train_y.shape)
print('X_test:', test_X.shape)
print('Y_test:', test_y.shape)


# Plot the first 9 images from the training dataset
for i in range(9):
   plt.subplot(330 + 1 + i)
   plt.imshow(train_X[i], cmap=plt.get_cmap('gray'))
plt.show()
```

**Code Explaination :**

```
from google.colab import drive
```

- Imports the drive module from Colab's helper package. This gives access to drive.mount() so you can attach your Google Drive.

```
drive.mount('/content/drive')
```

- Mounts (connects) your Google Drive filesystem to the Colab virtual machine at the path /content/drive.
- When run in Colab you'll be prompted to authenticate (choose account and paste an auth token) unless Drive was already mounted earlier. After mounting, your Drive files are accessible under /content/drive/MyDrive/.

from keras.datasets import mnist

- Imports the mnist dataset helper from Keras. mnist provides a convenience function load_data() that downloads (the first time) and returns the MNIST dataset arrays.

from matplotlib import pyplot as plt

- Imports Matplotlib's pyplot module and aliases it as plt. pyplot provides plotting functions like imshow, subplot, and show.

(train_X, train_y), (test_X, test_y) = mnist.load_data()

- Calls mnist.load_data() which returns two tuples: (x_train, y_train) and (x_test, y_test).
- train_X is a NumPy array of training images (shape: (60000, 28, 28)), train_y is the corresponding labels (shape: (60000,)).
- test_X and test_y are the test images and labels (usually (10000, 28, 28) and (10000,)).
- The dataset is grayscale 28×28 pixel images of handwritten digits (0–9). On first run this function downloads mnist.npz from TensorFlow's storage.

 print('X_train:', train_X.shape)

- Prints the shape (dimensions) of the train_X array. Example output: X_train: (60000, 28, 28).

 print('Y_train:', train_y.shape)

- Prints the shape of the training labels array. Example: Y_train: (60000,).

 print('X_test:', test_X.shape)

- Prints the shape of the test images array. Example: X_test: (10000, 28, 28).

 print('Y_test:', test_y.shape)

- Prints the shape of the test labels array. Example: Y_test: (10000,).

for i in range(9):

- Starts a loop with i from 0 to 8 (9 iterations). Each iteration will prepare one subplot and show one image.

plt.subplot(330 + 1 + i)

- plt.subplot() creates/selects a subplot in a grid. 330 + 1 + i is a compact way to express a 3×3 grid (3 rows × 3 columns) and the (i+1)th position.
- Explanation of 330 + 1 + i: 330 means 3 rows, 3 columns, and 0 (placeholder) — adding 1 + i gives the 1-based subplot index (1..9). Equivalent and clearer alternatives: plt.subplot(3, 3, i+1).

plt.imshow(train_X[i], cmap=plt.get_cmap('gray'))

- Displays the ith training image in the current subplot.
- train_X[i] is a 2D array (28×28). cmap=plt.get_cmap('gray') renders it in grayscale. Without cmap, Matplotlib may use a default colormap which can color the image.

plt.show()

- After the loop finishes, plt.show() renders the figure (the 3×3 grid) to the output cell. In many environments each imshow would display immediately; grouping them and calling plt.show() ensures they appear together as one figure.

Oral Questions and Answers

1. What is the aim of this experiment?

Answer:
To study and install deep learning packages — TensorFlow, Keras, Theano, and PyTorch — and understand their distinct features and functionality.

---

2. What is TensorFlow?

Answer:
TensorFlow is an open-source deep learning library developed by Google for numerical computation and building machine learning models using data flow graphs.

---

3. What is Keras?

Answer:
Keras is a high-level neural network API written in Python that runs on top of TensorFlow. It simplifies model creation, training, and testing.

---

4. What is Theano?

Answer:
Theano is a numerical computation library in Python that allows defining, optimizing, and evaluating mathematical expressions involving multi-dimensional arrays. It was one of the earliest deep learning frameworks.

---

5. What is PyTorch?

Answer:
PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It provides dynamic computation graphs and is widely used for research and production.

---

6. What are the differences between TensorFlow and PyTorch?

Answer:

- TensorFlow uses static computation graphs (though later versions added dynamic graphs).
- PyTorch uses dynamic computation graphs (eager execution).
- PyTorch is easier for debugging; TensorFlow is better for large-scale deployment.

---

7. Why do we create a virtual environment before installing TensorFlow?

Answer:
To isolate dependencies and prevent conflicts between different Python projects or versions.

---

8. What command is used to create a virtual environment in Ubuntu?

Answer:

python3 -m venv virtualenv

---

9. What command is used to activate a virtual environment?

Answer:

source virtualenv/bin/activate

---

10. What command installs TensorFlow inside the virtual environment?

Answer:

pip install --upgrade tensorflow

---

11. What is the prerequisite for installing Keras?

Answer:
Python version 3.5 or above, and TensorFlow must be installed since Keras runs on top of TensorFlow.

---

12. How can you verify that Keras is successfully installed?

Answer:
By running the command:

pip3 show keras

---

13. Which libraries are commonly used with TensorFlow and Keras for data handling?

Answer:
NumPy, Pandas, and Scikit-learn (sklearn).

---

14. What is the function of NumPy in deep learning?

Answer:
NumPy handles numerical computations and provides support for multi-dimensional arrays and matrices.

---

15. What is the function of Pandas in machine learning?

Answer:
Pandas is used for data analysis and manipulation, offering powerful data structures like DataFrames.

---

16. What is Scikit-learn used for?

Answer:
It provides tools for machine learning such as classification, regression, clustering, and data preprocessing.

---

17. What are the main requirements for installing Theano?

Answer:
Python3, Python3-pip, NumPy, SciPy, and BLAS libraries.

---

18. Write a simple Theano example and explain it.

Answer:

import theano.tensor as T

from theano import function

x = T.dscalar('x')

y = T.dscalar('y')

z = x + y

f = function([x, y], z)

f(5, 7)

This code defines two symbolic variables x and y, adds them, compiles into a function f, and then computes f(5,7).

---

19. How do you install PyTorch on Ubuntu using pip?

Answer:

pip3 install torch==1.8.1+cpu torchvision==0.9.1+cpu torchaudio==0.8.1 -f https://download.pytorch.org/whl/torch_stable.html

---

20. What is the conclusion of this experiment?

Answer:
TensorFlow, Keras, Theano, and PyTorch were successfully installed. Each has unique features suitable for different deep learning applications and environments.

## Practical No- 2

## Title : Implementing Feedforward neural networks

```
# Implementing feedforward neural networks with Keras and TensorFlow

# import the necessary packages
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt

# grab the MNIST dataset
print("[INFO] accessing MNIST...")
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# flatten and normalize the data
x_train = x_train.reshape((x_train.shape[0], -1)).astype('float32') / 255
x_test  = x_test.reshape((x_test.shape[0], -1)).astype('float32') / 255

# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test  = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define the network architecture
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
```

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

# Train the model
H = model.fit(x_train, y_train, epochs=15, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the network
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy*100:.2f}%')

# Plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(H.history['loss'], label="train_loss")
plt.plot(H.history['accuracy'], label="Accuracy")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```

**Code Explanation:**

- numpy (np): numerical arrays and operations (often used for data manipulation).
- tensorflow (tf): main DL framework used here; Keras is bundled inside tf.keras.
- mnist loader: convenience function to load the MNIST dataset (built into Keras).
- Sequential: simple model container that stacks layers linearly.
- Dense: fully connected (aka dense) layer type.
- matplotlib.pyplot (plt): plotting library for charts/figures.
- print(...) just logs that we're about to load the dataset.
- mnist.load_data() downloads (first time) and returns two tuples:
  - x_train: shape (60000, 28, 28) — 60k training images.
  - y_train: shape (60000,) — labels 0–9 for training.
  - x_test: shape (10000, 28, 28) — 10k test images.
  - y_test: shape (10000,) — test labels.
  - **Flatten / reshape**

    - `reshape((x_train.shape[0], -1))` turns each 28×28 image into a vector of length `784`.
    - After this each `x_train` row is one image: shape becomes `(60000, 784)`.
- **Type conversion and normalization**
    - `.astype('float32')` converts integer pixel values (0–255) to 32-bit floats which are required by models.
    - Dividing by `255` scales pixel values to range `[0.0, 1.0]`. This helps training stability and convergence.

**Why flatten?**

- Because this is a simple feedforward (fully connected) network. Convolutional networks (CNNs) usually keep the height/width channels.

y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)

y_test  = tf.keras.utils.to_categorical(y_test, num_classes=10)

Converts integer labels (e.g., 7) into vector form (one-hot), e.g. 7 → [0,0,0,0,0,0,0,1,0,0].

Required when using categorical_crossentropy loss and the network outputs a probability distribution over 10 classes (softmax).

 equential([...]) stacks layers in order.

 Dense(64, activation='relu', input_shape=(784,))

- First hidden layer with 64 neurons and ReLU activation.
- input_shape=(784,) tells Keras the shape of each input vector.

 Two more hidden Dense(64, activation='relu') layers — i.e., 3 hidden layers in total.

 Final Dense(10, activation='softmax')

- Output layer with 10 neurons (one per digit class).
- softmax converts raw outputs into a probability distribution that sums to 1.
- • `optimizer='sgd'`: stochastic gradient descent optimizer (vanilla). Works but can be slower than Adam; good for learning basics.
- • `loss='categorical_crossentropy'`: appropriate for multi-class one-hot labels.
- • `metrics=['accuracy']`: during training Keras will report classification accuracy.
- • `fit(...)` trains the model on the training data.
- • `epochs=15`: whole dataset is passed through the network 15 times.
- • `batch_size=32`: the model updates weights after every 32 examples.
- • `validation_data=(x_test, y_test)`: after each epoch Keras evaluates the model on the test set and reports `val_loss` and `val_accuracy`.
- • The return `H` (History object) contains training logs: `H.history['loss']`, `H.history['accuracy']`, `H.history['val_loss']`, `H.history['val_accuracy']`.

**Interpreting the printed logs (example lines you pasted):**

- During each epoch Keras prints training and validation loss/accuracy. Over epochs you expect training loss decrease and accuracy increase; validation metrics show generalization.
- `evaluate(...)` computes loss and metrics on `x_test` and `y_test` (returns values in the order you requested: loss, then accuracy).

- `print(...)` formats accuracy as a percentage with two decimals.

**Note:** `model.evaluate` prints a progress line (e.g., `313/313 … - accuracy: 0.9608 - loss: 0.1332`) and returns numeric values you store in variables.

- plt.style.use("ggplot"): sets a plotting style (visual preset).
- plt.figure(figsize=(10, 4)): creates a figure 10×4 inches.
- plt.subplot(1, 2, 1): prepares the left subplot in a 1×2 grid (this code seems to only create the left plot — you could add a second subplot for validation curves).
- plt.plot(H.history['loss'], label="train_loss"): plots training loss vs epoch.
- plt.plot(H.history['accuracy'], label="Accuracy"): plots training accuracy on same axes.
- plt.title/ xlabel/ ylabel/ legend: labeling and legend.

**Caveat:** plotting loss (usually ranges like 0–2) and accuracy (0–1) on the same y-axis mixes metrics with different scales — it's readable for rough trends but you may prefer:

- Plot loss and validation loss together.
- Plot accuracy and validation accuracy together.
- Or use two y-axes if you must show both on one plot.

**1. What is the aim of this experiment?**

**Answer:**
The aim of this experiment is to implement a Feedforward Neural Network (FNN) using **Keras and TensorFlow**, train it on a dataset (such as MNIST or CIFAR-10), evaluate its performance, and visualize the training loss and accuracy.

---

**2. What is a Feedforward Neural Network (FNN)?**

**Answer:**
A Feedforward Neural Network is the simplest type of artificial neural network where the data moves only in one direction—from input to hidden layers to output—without looping back. It is mainly used for **classification and regression** problems.

---

**3. What is the MNIST dataset?**

**Answer:**
MNIST (Modified National Institute of Standards and Technology) is a dataset of **handwritten digits (0–9)**.

- It contains **60,000 training** and **10,000 testing** images.
- Each image is **28×28 pixels** in grayscale.

---

**4. What are the libraries required for this practical?**

**Answer:**

- **TensorFlow / Keras** – For building and training neural networks.

- **NumPy** – For numerical computations and array handling.
- **Matplotlib** – For plotting graphs of accuracy and loss.
- **Scikit-learn** – For evaluation metrics (like precision, recall, F1-score).

---

**5. What is the purpose of one-hot encoding?**

**Answer:**
One-hot encoding converts **categorical integer labels** (like 0–9) into **binary vectors**.
Example:
3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
It allows the model to output probabilities for each class during classification.

---

**6. What is the role of the Sequential model in Keras?**

**Answer:**
The Sequential model allows you to build a neural network layer-by-layer in a **linear stack**, which is suitable for most feedforward architectures.

---

**7. What is the purpose of the Dense layer?**

**Answer:**
The Dense layer is a **fully connected layer**, where each neuron is connected to every neuron in the previous layer. It performs the core computation in a neural network.

---

**8. Why do we normalize the input data?**

**Answer:**
Normalization scales the input pixel values (0–255) to a smaller range (0–1).
This helps the neural network train faster and achieve better accuracy by preventing large input values from dominating the learning process.

---

**9. Which activation functions are commonly used?**

**Answer:**

- **ReLU (Rectified Linear Unit):** Used in hidden layers for faster training and avoiding vanishing gradients.
- **Softmax:** Used in the output layer for multi-class classification problems.

---

**10. What optimizer is used in this practical?**

**Answer:**
The **SGD (Stochastic Gradient Descent)** optimizer is used.
It updates the model's weights based on each batch of data to minimize the loss function.

---

**11. What is the loss function used here and why?**

**Answer:**
The loss function used is **Categorical Cross-Entropy**, suitable for **multi-class classification** tasks such as MNIST.

---

**12. What does the model's fit() function do?**

**Answer:**
The fit() function trains the model on training data.
It requires input features, labels, number of epochs, and batch size as parameters.

---

**13. What are epochs and batch size?**

**Answer:**

- **Epoch:** One complete pass of the training dataset through the model.
- **Batch size:** The number of samples processed before updating the model weights.

---

**14. How is model performance evaluated?**

**Answer:**
The model is evaluated using the evaluate() function which returns **loss** and **accuracy** on the test dataset.
Additional metrics like **Precision**, **Recall**, and **F1-score** can be calculated using classification_report().

---

**15. What accuracy did your model achieve?**

**Answer:**
The model achieved approximately **96% accuracy** on the MNIST test dataset after 15 epochs.

---

**16. What is the purpose of plotting accuracy and loss curves?**

**Answer:**
To visually analyze the model's performance:

- **Loss curve:** Shows how error reduces over epochs.
- **Accuracy curve:** Shows improvement in prediction accuracy over epochs.
  This helps to detect **overfitting** or **underfitting**.

---

**17. What is the difference between TensorFlow and Keras?**

**Answer:**

- **TensorFlow** is a deep learning framework developed by Google.
- **Keras** is a **high-level API** that runs on top of TensorFlow, simplifying the creation and training of models.

---

**18. What is the difference between MNIST and CIFAR-10 datasets?**

**Answer:**

| Feature | MNIST | CIFAR-10 |
|---|---|---|
| Type | Handwritten digits | Real-world color images |
| Image Size | 28x28 (Grayscale) | 32x32 (RGB) |
| Classes | 10 (digits 0–9) | 10 (objects like car, dog, ship, etc.) |

---

**19. What does model.predict() do?**

**Answer:**
It takes input data and returns the **predicted class probabilities** for each sample.
The class with the **highest probability** is considered the model's prediction.

---

**20. What are the key performance metrics in classification tasks?**

**Answer:**

- **Accuracy** = (Correct predictions / Total predictions)
- **Precision** = TP / (TP + FP)
- **Recall** = TP / (TP + FN)
- **F1-score** = 2 × (Precision × Recall) / (Precision + Recall)

---

**21. What is the role of LabelBinarizer in this experiment?**

**Answer:**
LabelBinarizer converts the class labels into **binary one-hot encoded vectors**, making them suitable for neural network output layers.

---

**22. What are some limitations of Feedforward Neural Networks?**

**Answer:**

- They can't handle sequential or time-based data well.
- Require large datasets and computational resources.
- Susceptible to **overfitting** if not properly regularized.

---

**23. What improvements can be made to increase model accuracy?**

**Answer:**

- Use more hidden layers or neurons.
- Increase epochs.

- Use **Adam** optimizer instead of SGD.
- Apply **Dropout** for regularization.
- Normalize data properly.

---

**24. What is the output layer size in this experiment?**

**Answer:**
The output layer contains **10 neurons**, corresponding to the **10 digit classes (0–9)** in the MNIST dataset.

---

**25. What conclusion did you draw from this experiment?**

**Answer:**
The Feedforward Neural Network implemented using **Keras and TensorFlow** successfully learned to classify handwritten digits from the MNIST dataset with high accuracy (~96%). It demonstrates the effectiveness of neural networks for image classification.

# Practical No 3

```
# Implementing feedforward neural networks with Keras and TensorFlow
# import the necessary packages
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Dropout
import matplotlib.pyplot as plt

# grab the MNIST dataset
print("[INFO] accessing MNIST...")
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# reshape and normalize the data
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1)).astype('float32') / 255
x_test  = x_test.reshape((x_test.shape[0], 28, 28, 1)).astype('float32') / 255

# build the model
model = Sequential()
# Convolutional layers
model.add(Conv2D(28, kernel_size=(3, 3), input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
# Fully connected layers
model.add(Dense(200, activation="relu"))
model.add(Dropout(0.3))
model.add(Dense(10, activation="softmax"))

# model summary
model.summary()
```

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

# train the model
model.fit(x_train, y_train, epochs=2)

# evaluate the network
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy*100:.2f}%')

# show one test image and predict its class
image = x_test[9]
plt.imshow(image.squeeze(), cmap='Greys')
plt.show()
image = image.reshape(1, 28, 28, 1)
prediction = model.predict(image)
print(np.argmax(prediction))
```

**Code Explanation :**

- numpy as np: general numerical operations and array utilities.
- tensorflow as tf: main deep learning framework; tf.keras is Keras bundled with TensorFlow.
- mnist loader: convenient function to load the MNIST dataset (images + labels).
- Sequential: Keras model container that stacks layers linearly.
- Dense, Conv2D, MaxPooling2D, Flatten, Dropout: layer types used in the network:
  - Conv2D: 2D convolution layer for images.
  - MaxPooling2D: spatial downsampling.
  - Flatten: convert multi-dim tensor to 1D vector.
  - Dense: fully connected layer.
  - Dropout: regularization that randomly disables neurons during training.
- matplotlib.pyplot as plt: plotting functions.
- Prints a short info message.
- mnist.load_data() returns:
  - x_train: shape (60000, 28, 28) — 60k grayscale training images.
  - y_train: shape (60000,) — training labels (integers 0–9).

- x_test: shape (10000, 28, 28) and y_test: (10000,) for testing.

 **Reshape:** adds a channel dimension (28,28,1) because Conv2D expects input (height, width, channels). After this:

- x_train shape becomes (60000, 28, 28, 1).

 **.astype('float32'):** convert integer pixels (0–255) to float32 for numerical stability.

 **/ 255:** normalize pixel values into [0.0, 1.0]. Normalization helps training converge faster and avoids large gradient steps.

1. Sequential() — create an empty sequential model.
2. Conv2D(28, kernel_size=(3,3), input_shape=(28,28,1))
    - First conv layer with **28 filters** (each filter produces one output channel).
    - kernel_size=(3,3) means 3×3 receptive fields.
    - input_shape=(28,28,1) defines the input dimensions for the first layer.
    - Resulting output shape after conv: (None, 26, 26, 28) because valid padding (default) reduces spatial dims from 28→26.
    - **Note/warning:** Keras sometimes warns about passing input_shape directly to a layer; using tf.keras.Input(shape=(28,28,1)) or model = Sequential([Input(shape=(28,28,1)), ...]) is preferred style.
3. MaxPooling2D(pool_size=(2,2))
    - Downsamples spatial dimensions by a factor of 2 → after pooling: (None, 13, 13, 28).
4. Flatten()
    - Flattens (13,13,28) → 13*13*28 = 4732 features per sample.
5. Dense(200, activation="relu")
    - Fully connected layer with 200 neurons and ReLU activation.
6. Dropout(0.3)
    - During training, randomly drops 30% of the neurons in the previous layer to reduce overfitting.
7. Dense(10, activation="softmax")
    - Output layer with 10 neurons (one per digit class).
    - softmax outputs a probability distribution over the 10 classes.

 Prints a table of each layer, its output shape, and the number of parameters.

 Your summary showed:

- Conv2D output (None, 26, 26, 28) and Param # = 280.
    - Param calculation: (kernel_height * kernel_width * input_channels * filters) + bias = (3*3*1*28) + 28 = 252 + 28 = 280.
- Flatten → 4732 features.
- Dense(200) params = 4732*200 + 200 = 946,600 (weights + biases).

- Dense(10) params = 200*10 + 10 = 2010.
- Total params: 948,890. Trainable params same since no frozen layers.

 optimizer='adam': Adam optimizer (adaptive learning rates) — usually faster, robust choice.

 loss='sparse_categorical_crossentropy':

- Use this when labels are integers (0..9), **no need to one-hot encode**.
- If labels were one-hot vectors, you'd use categorical_crossentropy.

 metrics=['accuracy']: Keras will compute and report accuracy during training/evaluation.

 Trains on training data for 2 epochs (full passes over the training set).

 Defaults: batch_size=32 if not provided.

 The printed lines show training accuracy and loss per epoch:

- Epoch 1: accuracy ~0.8935, loss ~0.3502
- Epoch 2: accuracy ~0.9728, loss ~0.0882

 Rapid improvement indicates the model learns MNIST quickly (reasonable given this architecture).

 evaluate returns loss and metrics on the test dataset.

 Example result: accuracy: 0.9757 - loss: 0.0757 → printed as Test accuracy: 98.07%.

 This indicates very good generalization for MNIST with this simple CNN.

- x_test[9] is a single image with shape (28,28,1).
- .squeeze() removes the channel dimension for plotting → shape (28,28).
- cmap='Greys' displays the image as grayscale.
- • `image.reshape(1,28,28,1)`: add a batch dimension, so shape is `(1,28,28,1)` as model expects batches.
- • `model.predict(image)` returns a `(1,10)` array: predicted probabilities for each class.
- • `np.argmax(prediction)` finds the index of the maximum probability → predicted digit (0–9).
- • Example printed value `9` indicates the model predicted class `9` for that image (your output showed `9`).

### 🧠 1. What is Image Classification?

**Answer:**
Image classification is the process of categorizing and labeling groups of pixels or vectors within an image based on specific rules. In deep learning, Convolutional Neural Networks (CNNs) are used to automatically learn features from images and classify them into categories such as cats, dogs, digits, etc.

---

### 🧠 2. What are the 4 main stages in building an image classification model?

**Answer:**

1. **Loading & Preprocessing Data** – Import and resize images, assign labels, normalize pixel values.
2. **Defining Model Architecture** – Create CNN layers (Conv2D, MaxPooling, Flatten, Dense).
3. **Training the Model** – Compile and fit the model with training data.

4. **Evaluating Performance** – Test model accuracy on unseen data.

---

### 🧠 3. What is the role of preprocessing in image classification?

**Answer:**
Preprocessing converts raw images into a format suitable for the model — resizing images, scaling pixel values (e.g., dividing by 255), converting labels to categorical form, and splitting data into training/testing sets.

---

### 🧠 4. Which libraries are used and why?

**Answer:**

- **TensorFlow / Keras** – For building and training neural networks.
- **NumPy** – For numerical operations on arrays.
- **Pandas** – For data handling and manipulation.
- **Scikit-learn (sklearn)** – For splitting data and evaluating models.
- **Matplotlib** – For visualizing data and results.

---

### 🧠 5. What is a Convolutional Neural Network (CNN)?

**Answer:**
CNN is a deep learning architecture designed for image processing. It uses convolutional layers to detect spatial hierarchies and features like edges, corners, and textures.

---

### 🧠 6. Explain the key layers of CNN.

**Answer:**

1. **Conv2D (Convolutional Layer):** Extracts features from the image.
2. **MaxPooling2D (Pooling Layer):** Reduces spatial size and computation.
3. **Flatten:** Converts 2D feature maps into 1D vectors.
4. **Dense (Fully Connected Layer):** Performs classification based on extracted features.
5. **Dropout:** Reduces overfitting by randomly ignoring some neurons during training.

---

### 🧠 7. Why do we normalize images (divide by 255)?

**Answer:**
To scale pixel values (0–255) into a smaller range (0–1), which helps the model converge faster and improves accuracy.

---

### 🧠 8. What is the purpose of train_test_split() from sklearn?

**Answer:**
It splits the dataset into **training** and **testing** parts to evaluate model performance on unseen data.

---

### 🧠 9. What is one-hot encoding and why is it used?

**Answer:**
It converts class labels (like 0–9) into binary vectors. Example:
3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
This helps in multi-class classification problems.

---

### 🧠 10. What is the activation function used in CNN?

**Answer:**

- **ReLU (Rectified Linear Unit):** For hidden layers; introduces non-linearity.
- **Softmax:** For output layer; converts logits to probabilities for each class.

---

### 🧠 11. What optimizer and loss function are used?

**Answer:**

- **Optimizer:** Adam or SGD (Stochastic Gradient Descent).
- **Loss Function:** categorical_crossentropy or sparse_categorical_crossentropy.

---

### 🧠 12. What is an epoch and batch size?

**Answer:**

- **Epoch:** One full pass of training data through the model.
- **Batch size:** Number of samples processed before updating weights.

---

### 🧠 13. What is Dropout and why is it used?

**Answer:**
Dropout randomly turns off a fraction of neurons during training to prevent overfitting and improve generalization.

---

### 🧠 14. What does model.evaluate() do?

**Answer:**
It tests the model on unseen data and returns metrics like **loss** and **accuracy**.

---

### 🧠 15. What are precision, recall, and F1-score?

**Answer:**

- **Precision:** Correctly predicted positives / Total predicted positives.
- **Recall:** Correctly predicted positives / All actual positives.
- **F1-score:** Harmonic mean of precision and recall — balances both.

---

### 🧠 16. Why do we use model.summary()?

**Answer:**
To display the architecture of the neural network, showing layers, output shapes, and total trainable parameters.

---

### 🧠 17. How can we improve model accuracy?

**Answer:**

- Use **more data** or data augmentation.
- Increase **epochs**.
- Add **more layers** or neurons.
- Use **regularization techniques** (Dropout, BatchNorm).
- Use a **learning rate scheduler**.

---

### 🧠 18. What is overfitting?

**Answer:**
Overfitting occurs when the model learns training data too well but fails to generalize to new (test) data. It can be reduced by dropout or early stopping.

---

### 🧠 19. What is the output of the CNN model in image classification?

**Answer:**
A probability distribution across all classes (e.g., for 10 categories, output like [0.1, 0.2, 0.6, ...]), and the class with the highest probability is selected as prediction.

---

### 🧠 20. What conclusion can you write for this practical?

**Answer:**
The CNN model was successfully built and trained for image classification. It efficiently learned features from the images and achieved high accuracy, demonstrating the effectiveness of deep learning for visual data.

# Practical No.4

# Title : Anomaly detection using Autoencoders

```
##
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, losses
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Model

##
(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
##
print(x_train.shape)
print(x_test.shape)
##
latent_dim = 64

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(784, activation='sigmoid'),
```

```
            layers.Reshape((28, 28))
        ])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

##
autoencoder = Autoencoder(latent_dim)
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

autoencoder.fit(
    x_train, x_train,
    epochs=10,
    shuffle=True,
    validation_data=(x_test, x_test)
)

##
encoded_imgs = autoencoder.encoder(x_test).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()

n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # original image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i])
    plt.title("Original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

```
    # reconstructed image
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i])
    plt.title("Reconstructed")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

**Code Explanation :**

✅ **Explanation:**

- matplotlib.pyplot: for plotting images.
- numpy: for array operations.
- pandas: for handling data (not directly used here).
- tensorflow: for building neural networks.
- sklearn.metrics: for evaluating model performance.
- train_test_split: used to split data (optional in this case).
- layers and losses: to define network layers and loss functions.
- Model: base class in Keras used to create custom models.
- fashion_mnist: built-in dataset of fashion items (like shoes, shirts, etc.).

---

**2. Load the Dataset**

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()
```

✅ **Explanation:**

- Loads the **Fashion MNIST** dataset.
- Each image = 28×28 grayscale pixel array.
- _ is used because we don't need labels (autoencoder is unsupervised).

---

**3. Normalize the Data**

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

✅ **Explanation:**

- Converts pixel values from 0–255 to 0–1 for faster convergence.

---

**4. Check Data Shape**

```
print(x_train.shape)
print(x_test.shape)
```

✅ **Output:**

```
(60000, 28, 28)
(10000, 28, 28)
```

This means there are **60,000 training images** and **10,000 testing images**.

---

**5. Define Latent Dimension**

```
latent_dim = 64
```

✅ **Explanation:**

- Defines the size of the compressed representation (feature vector) learned by the encoder.

---

**6. Define Autoencoder Class**

```
class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
```

✅ **Explanation:**

- Creates a custom Autoencoder model by subclassing Model.
- __init__ initializes the encoder and decoder components.

---

**7. Encoder Definition**

```
self.encoder = tf.keras.Sequential([
  layers.Flatten(),
  layers.Dense(latent_dim, activation='relu'),
])
```

✅ **Explanation:**

- **Flatten:** converts 28×28 images → 784-dimensional vector.
- **Dense:** compresses input into a 64-dimensional latent space using ReLU activation.

---

**8. Decoder Definition**

```
self.decoder = tf.keras.Sequential([
    layers.Dense(784, activation='sigmoid'),
    layers.Reshape((28, 28))
])
```

✅ **Explanation:**

- **Dense:** reconstructs image from latent vector (64 → 784 neurons).
- **Sigmoid:** ensures output pixels are between 0 and 1.
- **Reshape:** converts 784 back into a 28×28 image.

---

**9. Forward Pass**

```
def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded
```

✅ **Explanation:**

- Defines how data passes through the model (encoder → decoder → output).

---

**10. Create and Compile Model**

```
autoencoder = Autoencoder(latent_dim)
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

✅ **Explanation:**

- Creates an Autoencoder object.
- Uses **Adam optimizer** and **Mean Squared Error (MSE)** as the loss function since this is a reconstruction task.

---

**11. Train the Model**

```
autoencoder.fit(x_train, x_train, epochs=10, shuffle=True, validation_data=(x_test, x_test))
```

✅ **Explanation:**

- Both **input** and **target output** are the same images (unsupervised learning).
- Model learns to **reconstruct** input images.
- Trained for **10 epochs**, using **x_train** and validating on **x_test**.

✅ **Training Output Example:**

Epoch 1/10 - loss: 0.0395 - val_loss: 0.0134

Epoch 10/10 - loss: 0.0087 - val_loss: 0.0088

Loss decreases over epochs → model is learning successfully.

---

**12. Encode and Decode Test Images**

encoded_imgs = autoencoder.encoder(x_test).numpy()

decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()

✅ **Explanation:**

- Compresses (encoder) and reconstructs (decoder) test images.
- numpy() converts TensorFlow tensors to NumPy arrays for visualization.

---

**13. Display Original and Reconstructed Images**

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i])
    plt.title("original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Reconstructed
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i])
    plt.title("reconstructed")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

✅ **Explanation:**

- Displays 10 original images on the top row.

- Displays reconstructed (autoencoded) images on the bottom row.
- plt.gray() shows grayscale images.

**Viva / Oral Questions with Answers**

---

**1. What is an Autoencoder?**

An autoencoder is a **neural network that learns to compress data (encode)** into a smaller representation and **reconstruct (decode)** it back to the original form.

---

**2. What are the two main parts of an Autoencoder?**

- **Encoder:** Compresses the input data into a lower-dimensional latent space.
- **Decoder:** Reconstructs the original data from the compressed representation.

---

**3. Is autoencoder supervised or unsupervised learning?**

Unsupervised learning — it does not need labels; input and output are the same.

---

**4. What is the purpose of latent_dim?**

Defines the number of features in the compressed (encoded) representation — controls how much compression is done.

---

**5. Why use the Mean Squared Error (MSE) loss?**

Because the goal is to reconstruct images as closely as possible to the input — MSE measures reconstruction error between original and predicted pixels.

---

**6. Why use sigmoid activation in the decoder?**

Because the image pixel values are between 0 and 1 (after normalization), sigmoid ensures the output stays in the same range.

---

**7. Why do we divide the pixel values by 255?**

To normalize pixel intensity from [0, 255] to [0, 1] — this speeds up learning and improves accuracy.

---

**8. What optimizer is used here and why?**

**Adam Optimizer** — it adapts the learning rate dynamically and converges faster than traditional SGD.

---

**9. What happens if latent dimension is too small or too large?**

- **Too small:** Model may lose important details (underfitting).
- **Too large:** Model may memorize data instead of learning features (overfitting).

---

**10. What is the role of Flatten and Reshape layers?**

- **Flatten:** Converts 2D image (28×28) → 1D vector.
- **Reshape:** Converts 1D vector back → 2D image for output visualization.

---

**11. How do you visualize reconstructed images?**

Using **matplotlib**, display original and reconstructed images side by side.

---

**12. What are real-world applications of autoencoders?**

- Image Denoising
- Dimensionality Reduction
- Anomaly Detection
- Image Compression
- Feature Extraction

---

**13. What is the difference between encoder and decoder?**

Encoder compresses the input into latent features, while decoder reconstructs data back from latent space.

---

**14. What metric tells us the model is learning?**

Decreasing **training loss** and **validation loss** over epochs indicate learning progress.

---

**15. What is the conclusion of this experiment?**

The autoencoder was successfully trained on Fashion MNIST data. It learned to compress images into a lower-dimensional latent space and reconstruct them accurately, demonstrating the concept of unsupervised feature learning.

# Practical No 5

**Title : Implement the Continuous Bag of Words (CBOW) Model.**

Code:

```
##
import re
import numpy as np
import string
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
from subprocess import check_output
from wordcloud import WordCloud, STOPWORDS
##
stopwords = set(STOPWORDS)
sentences = """We are about to study the idea of a computational process.Computational processes are abstract beings that inhabit computers.As they evolve, processes manipulate other abstract things called data.The evolution of a process is directed by a pattern of rulescalled a program. People create programs to direct processes. In effect,we conjure the spirits of the computer with our spells."""
##
wordcloud = WordCloud(
    background_color='white',
    stopwords=stopwords,
    max_words=200,
```

```
    max_font_size=40,
    random_state=42
).generate(sentences)
fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
axes.imshow(wordcloud)
axes.axis('off')
fig.tight_layout()
##
sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)
sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()
sentences = sentences.lower()
words = sentences.split()
vocab = set(words)
print(words)
print(vocab)
##
vocab_size = len(vocab)
embed_dim = 10
context_size = 2
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for i, word in enumerate(vocab)}
print(word_to_ix)
print(ix_to_word)
##
data = []
for i in range(2, len(words) - 2):
  context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]
  target = words[i]
  data.append((context, target))
  print(data[:5])
##
embeddings = np.random.random_sample((vocab_size, embed_dim))
print(embeddings)
```

**Code Explanation:**

1. **Importing Libraries**

```
import re
import numpy as np
import string
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
from subprocess import check_output
from wordcloud import WordCloud, STOPWORDS
```

- re: Regular expressions for text cleaning.
- numpy: For numerical computations and matrix operations.
- pandas: Data manipulation (not directly used here but useful for tabular data).
- matplotlib.pyplot: Plotting graphs and visualizations.
- wordcloud: For creating a word cloud of text.
- STOPWORDS: Common words to ignore in word clouds (like "the", "and").

---

2. **Setting Stopwords**

```
stopwords = set(STOPWORDS)
```

- Converts default stopwords into a Python set for easy lookup.

---

3. **Text Input**

```
sentences = """We are about to study ... with our spells."""
```

- Multiline string containing the text to analyze.

---

4. **Generate Word Cloud**

```
wordcloud = WordCloud(
    background_color='white',
    stopwords=stopwords,
    max_words=200,
    max_font_size=40,
    random_state=42
).generate(sentences)
```

- WordCloud() creates a word cloud object.

- Parameters:
    - background_color='white': white background.
    - stopwords=stopwords: ignores common words.
    - max_words=200: maximum words to display.
    - max_font_size=40: maximum font size.
    - random_state=42: ensures reproducible layout.
- .generate(sentences): creates the word cloud from the input text.

---

5. **Plot the Word Cloud**

```
fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
axes.imshow(wordcloud)
axes.axis('off')
fig.tight_layout()
```

- Creates a figure and axes for plotting.
- Displays the word cloud using imshow.
- Removes axes labels with axis('off').
- tight_layout() ensures proper spacing.

---

6. **Clean Text**

```
sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)
sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()
sentences = sentences.lower()
```

- Removes non-alphanumeric characters.
- Removes single-letter words (like "a", "I").
- Converts all text to lowercase for uniformity.

---

7. **Split Words and Create Vocabulary**

```
words = sentences.split()
vocab = set(words)
```

- words: list of words from text.
- vocab: unique words (set) in the text.

---

8. **Define Embedding Parameters**

```
vocab_size = len(vocab)
```

embed_dim = 10

context_size = 2

- vocab_size: number of unique words.
- embed_dim: dimensionality of word embeddings.
- context_size: number of words to consider before and after the target word.

---

9. **Create Word-Index Mappings**

word_to_ix = {word: i for i, word in enumerate(vocab)}

ix_to_word = {i: word for i, word in enumerate(vocab)}

- word_to_ix: maps each word to a unique integer.
- ix_to_word: maps integers back to words.

---

10. **Generate Context-Target Pairs**

```
data = []
for i in range(2, len(words) - 2):
    context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]
    target = words[i]
    data.append((context, target))
```

- Iterates over each word (skipping first and last 2 words).
- context: two words before and after the target word.
- target: current word to predict.
- data: list of (context, target) pairs for training a model.

---

11. **Initialize Random Embeddings**

embeddings = np.random.random_sample((vocab_size, embed_dim))

- Creates a random matrix of shape (vocab_size, embed_dim).
- Each row is a vector representing a word in the embedding space.
- These embeddings will later be trained using a neural network (like CBOW or Skip-gram).

# Questions and Answers – CBOW Practical

**Q1. What is the aim of the CBOW model?**
**A1.** The aim of the Continuous Bag of Words (CBOW) model is to predict a target word based on its surrounding context words in a sentence or paragraph. It is commonly used in NLP for learning word embeddings.

---

**Q2. Which libraries are required to implement CBOW in Python?**
**A2.** Required libraries include:

- numpy for numerical computations
- keras modules: Sequential, Dense, Embedding, Lambda, np_utils, sequence, Tokenizer
- gensim for NLP operations and loading pre-trained word vectors

---

**Q3. How is the input text prepared for CBOW?**
**A3.** Steps for data preparation:

1. Create a paragraph with 5–10 sentences.
2. Tokenize the paragraph into words.
3. Fit the tokenized data to a Keras Tokenizer.
4. Convert words into integers (word indices).

---

**Q4. How do you generate CBOW training data?**
**A4.** Training data is generated using a sliding window around each word:

- Choose a window_size (number of words on left and right of the target).
- For each word, select surrounding words as context and the word itself as the target.
- Convert context words to padded sequences.
- Convert target words to one-hot vectors.

---

**Q5. Can you show an example of generating context-target pairs?**
**A5.** Example with window size 2:

- Sentence: "The cat sat on the mat"
- Target word: "sat"
- Context words: ["The", "cat", "on", "the"]
- Convert context to sequences and target to one-hot vector.

---

**Q6. How is the CBOW neural network model defined?**
**A6.** Model definition steps:

- Use Sequential model.
- Add an Embedding layer to convert word indices to vectors.
- Use Lambda layer to average the embeddings of context words.
- Add Dense layer with softmax activation to predict target word.
- Compile with loss='categorical_crossentropy' and optimizer='adam'.

---

**Q7. How are the learned word embeddings saved?**
**A7.** Steps to save embeddings:

- Extract weights from the trained embedding layer.
- Write the word and its embedding vector to a text file in Word2Vec format.
- Example format: "word v1 v2 v3 ... vn"

---

**Q8. How are the embeddings loaded into Gensim for querying similar words?**
**A8.** Use:

```
import gensim

cbow_output = gensim.models.KeyedVectors.load_word2vec_format('/content/vectors.txt', binary=False)

similar_words = cbow_output.most_similar(positive=['word'])
```

- This allows querying for words similar to a given word based on embeddings.

---

**Q9. How do you query similar words using the CBOW model?**
**A9.** Use most_similar() method on the trained Gensim word vectors, passing the target word as positive. The model returns a list of words closest to the input word in embedding space.

---

**Q10. What is the difference between CBOW and Skip-gram models?**
**A10.**

- CBOW predicts a target word using its context words.
- Skip-gram predicts context words using a target word.
- CBOW is faster and works well with small datasets, while Skip-gram works better for rare words and large datasets.

---

**Q11. What is the role of the embedding layer in CBOW?**
**A11.** The embedding layer maps each word index to a dense vector of fixed dimensions (word embeddings), capturing semantic relationships between words.

---

**Q12. How do you evaluate the CBOW model?**
**A12.** CBOW evaluation can be done by:

1. Checking the similarity of words in embedding space.
2. Using intrinsic tests like word analogies (king - man + woman = queen).
3. Observing loss reduction during training (categorical cross-entropy loss).

---

**Q13. What are some hyperparameters in CBOW?**
**A13.** Important hyperparameters include:

- embedding_dim – size of the word vector.
- window_size – number of context words on either side.
- batch_size – number of samples per training batch.
- epochs – number of training iterations.
- optimizer – e.g., Adam.

---

**Q14. How do you pad context words for CBOW input?**
**A14.** Use keras.preprocessing.sequence.pad_sequences() to ensure all context windows have the same length, allowing batch processing.

---

**Q15. Why is one-hot encoding used for target words?**
**A15.** One-hot encoding converts target word indices into vectors of length equal to vocabulary size. This is required for categorical cross-entropy loss in the output layer.

# Practical no.6

**Title : Object detection using Transfer Learning of CNN architectures**

**Code :**

```
#import
import tensorflow_datasets as tfds
import tensorflow as tf
from tensorflow.keras.utils import to_categorical

## Loading images and labels
(train_ds, train_labels), (test_ds, test_labels) = tfds.load(
    "tf_flowers",
    split=["train[:70%]", "train[:30%]"],
    batch_size=-1,
    as_supervised=True
)

## check existing image size
train_ds[0].shape

## Resizing images
train_ds = tf.image.resize(train_ds, (150, 150))
```

```
test_ds = tf.image.resize(test_ds, (150, 150))
train_ds[0].shape

## Transforming labels to correct format
train_labels = to_categorical(train_labels, num_classes=5)
test_labels = to_categorical(test_labels, num_classes=5)

from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input

## Loading VGG16 model
base_model = VGG16(weights="imagenet", include_top=False,
                   input_shape=train_ds[0].shape)

## We will not train base model i.e. Freeze Parameters in model's lower convolutional layers
base_model.trainable = False
## Preprocessing input
train_ds = preprocess_input(train_ds)
test_ds = preprocess_input(test_ds)
## model details
base_model.summary()

#add our layers on top of this model
from tensorflow.keras import layers, models

flatten_layer = layers.Flatten()
dense_layer_1 = layers.Dense(50, activation='relu')
dense_layer_2 = layers.Dense(20, activation='relu')
prediction_layer = layers.Dense(5, activation='softmax')

model = models.Sequential([
    base_model, flatten_layer, dense_layer_1, dense_layer_2, prediction_layer
])
```

```
#Train classifier layers on training data available for task
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

history = model.fit(train_ds, train_labels, epochs=10, validation_split=0.2, batch_size=32)

#Evaluate Model
los,accurac=model.evaluate(test_ds,test_labels)
print("Loss: ",los,"Accuracy: ", accurac)
#Plotting training accuracy
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'])
plt.title('ACCURACY')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()
```

# Code Explanation:

### 1. References and Imports

```
import tensorflow_datasets as tfds
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
```

- tensorflow_datasets (tfds): For loading the TF Flowers dataset.
- tensorflow (tf): For building and training deep learning models.
- to_categorical: Converts integer labels to one-hot vectors.

---

### 2. Load the Dataset

```
(train_ds, train_labels), (test_ds, test_labels) = tfds.load(
    "tf_flowers",
```

split=["train[:70%]", "train[:30%]"],

batch_size=-1,

as_supervised=True

)

- Load TF Flowers dataset with 5 classes.
- split creates 70% train and 30% test split.
- batch_size=-1 loads the full dataset into memory.
- as_supervised=True ensures we get (image, label) pairs.

---

**3. Explore Image Shape**

train_ds[0].shape

- Check the original image dimensions. Output: (442, 1024, 3) → large and variable.

---

**4. Resize Images**

train_ds = tf.image.resize(train_ds, (150, 150))

test_ds = tf.image.resize(test_ds, (150, 150))

train_ds[0].shape

- Resizes all images to 150x150 pixels for VGG16 input.
- Output shape: (150, 150, 3).

---

**5. Transform Labels**

train_labels = to_categorical(train_labels, num_classes=5)

test_labels = to_categorical(test_labels, num_classes=5)

- Convert labels to **one-hot encoding** (needed for categorical_crossentropy loss).
- Example: class 2 → [0, 0, 1, 0, 0].

---

**6. Load Pretrained VGG16**

from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input

base_model = VGG16(weights="imagenet", include_top=False, input_shape=train_ds[0].shape)

- weights="imagenet": Use pre-trained weights from ImageNet.
- include_top=False: Exclude the original fully-connected classifier.
- input_shape: Input images must match resized shape (150, 150, 3).

---

**7. Freeze Base Model Layers**

base_model.trainable = False

- Prevents pre-trained convolutional layers from being updated.
- Only the custom classifier layers will train.

---

**8. Preprocess Input Images**

train_ds = preprocess_input(train_ds)

test_ds = preprocess_input(test_ds)

- Normalize images according to VGG16's preprocessing.
- Scales RGB channels to match the training distribution of ImageNet.

---

**9. Base Model Summary**

base_model.summary()

- Displays all convolutional and pooling layers of VGG16.
- Confirms that the model has ~14 million parameters, all frozen.

---

**10. Add Custom Classifier**

```
from tensorflow.keras import layers, models

flatten_layer = layers.Flatten()
dense_layer_1 = layers.Dense(50, activation='relu')
dense_layer_2 = layers.Dense(20, activation='relu')
prediction_layer = layers.Dense(5, activation='softmax')

model = models.Sequential([
    base_model, flatten_layer, dense_layer_1, dense_layer_2, prediction_layer
])
```

- Flatten: Convert 3D feature maps to 1D vector.
- Dense(50), Dense(20): Hidden layers for learning task-specific features.
- Dense(5, softmax): Output layer for 5 flower classes.

---

**11. Compile and Train**

```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
```

```
    metrics=['accuracy']
)

history = model.fit(train_ds, train_labels, epochs=10, validation_split=0.2, batch_size=32)
```

- optimizer='adam': Efficient optimizer for training.
- loss='categorical_crossentropy': Appropriate for multi-class classification.
- validation_split=0.2: 20% of training data used for validation.
- batch_size=32: Number of samples per gradient update.

---

**12. Evaluate Model**

```
loss, accuracy = model.evaluate(test_ds, test_labels)
print("Loss:", loss, "Accuracy:", accuracy)
```

- Evaluates the trained model on unseen test data.
- Outputs test loss and test accuracy.

---

**13. Plot Training Accuracy**

```
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'])
plt.title('ACCURACY')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()
```

- Plots training accuracy over epochs to visualize learning progress.

---

✅ **Summary of Workflow**

1. Load and preprocess TF Flowers dataset.
2. Resize images and convert labels to one-hot.
3. Use VGG16 as base (pre-trained, frozen).
4. Add custom classifier layers.
5. Compile and train the classifier.
6. Evaluate and visualize performance.

## Questions and Answers

**Q1. What is the aim of using transfer learning in CNNs for object detection?**
**A1.** The aim is to leverage pre-trained convolutional neural network models trained on large datasets to detect objects in a new, smaller dataset, reducing training time and improving performance. Lower layers of CNNs capture generic features like edges and textures, which can be reused for the new task.

---

**Q2. Which dataset is used for this object detection task?**
**A2.** The dataset is available at Caltech Dataset, which contains images separated into classes. The data is split into training (50%), validation (25%), and testing (25%) sets, organized into directories per class.

---

**Q3. Which libraries are required for this task?**
**A3.** Required libraries:

- PyTorch for deep learning (torch)
- torchvision for pre-trained models, datasets, and image transformations
- torch.nn for defining layers and loss functions
- torch.optim for optimizers
- torch.utils.data.DataLoader for batching the dataset

---

**Q4. How is the dataset preprocessed before feeding into the model?**
**A4.** Preprocessing involves using transforms.Compose() from PyTorch:

- **Training:** RandomResizedCrop, RandomRotation, ColorJitter, RandomHorizontalFlip, CenterCrop, ToTensor, Normalize
- **Validation:** Resize, CenterCrop, ToTensor, Normalize
- Normalization uses ImageNet mean [0.485, 0.456, 0.406] and std [0.229, 0.224, 0.225].

---

**Q5. How are the datasets and dataloaders created?**
**A5.**

```
data = {
   'train': datasets.ImageFolder(root=traindir, transform=image_transforms['train']),
   'valid': datasets.ImageFolder(root=validdir, transform=image_transforms['valid'])
}


dataloaders = {
   'train': DataLoader(data['train'], batch_size=batch_size, shuffle=True),
   'val': DataLoader(data['valid'], batch_size=batch_size, shuffle=True)
}
```

- ImageFolder reads images from directories per class.
- DataLoader batches the data for training and validation.

---

**Q6. How is the pre-trained model loaded and prepared?**
**A6.**

```
from torchvision import models
model = models.vgg16(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
```

- Loads VGG16 pre-trained on ImageNet.
- Freezes all parameters to prevent updating during training.

---

**Q7. How is the custom classifier added?**
**A7.**

```
n_inputs = model.classifier[6].in_features
n_classes = 100  # number of classes
model.classifier[6] = nn.Sequential(
    nn.Linear(n_inputs, 256),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(256, n_classes),
    nn.LogSoftmax(dim=1)
)
```

- Replaces the last classifier layer.
- Uses two fully connected layers with ReLU, Dropout, and log-softmax for output.

---

**Q8. Which loss function and optimizer are used?**
**A8.**

```
criterion = nn.NLLLoss()  # Negative Log-Likelihood Loss
optimizer = optim.Adam(model.parameters())
```

- NLLLoss is used with log-softmax output.
- Adam optimizer updates the trainable parameters.

---

**Q9. How is the model trained?**
**A9.**

```
for epoch in range(n_epochs):
    for data, targets in trainloader:
        out = model(data)
        loss = criterion(out, targets)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

- Forward pass: compute predictions
- Compute loss
- Backpropagate gradients
- Update parameters using optimizer

---

**Q10. What techniques are applied to avoid overfitting?**
**A10.**

- Dropout layers in classifier (40%)
- Early stopping based on validation performance
- Freezing pre-trained convolutional layers

---

**Q11. How is model performance evaluated?**
**A11.**

```
ps = model(data)
pred = torch.max(ps, dim=1).indices
equals = pred == targets
accuracy = torch.mean(equals.float())
```

- Compare predicted labels with true labels.
- Compute accuracy as the mean of correctly predicted samples.

---

**Q12. What is the purpose of freezing layers in transfer learning?**
**A12.** Freezing layers prevents the pre-trained weights from being updated, preserving learned features like edges, textures, and shapes. Only the custom classifier layers learn features specific to the new dataset.