What is a Tensor?

A **tensor** is a **mathematical object** that stores data in a **multi-dimensional array** form. Think of it as an extension of:

- **Scalar (0D)** → Single number
- **Vector (1D)** → Array of numbers
- **Matrix (2D)** → Table of numbers
- **Tensor (nD)** → Generalized form for n dimensions

In deep learning (PyTorch, TensorFlow), **tensors are the core data structure** for storing:

- Input data (images, text, audio)
- Model parameters (weights, biases)
- Intermediate computations

Tensor Dimensions

Dimension	Name	Example
0D	Scalar	`42`
1D	Vector	`[1, 2, 3]`
2D	Matrix	`[[1, 2], [3, 4]]`
3D	3D Tensor	Stack of matrices (e.g., RGB image)
nD	n-Dimensional Tensor	e.g., batch of images, videos

Tensor Properties

- **Shape** → The number of elements along each dimension (e.g., `(3,4)` for a 3×4 matrix).
- **Rank** → Number of dimensions (e.g., scalar rank=0, vector rank=1).
- **Data type (dtype)** → `float32`, `int64`, etc.
- **Device** → Where it's stored (CPU or GPU).

Tensors vs NumPy vs Pandas

Feature	Tensors (TF/PyTorch)	NumPy (ndarray)	Pandas (Series/DataFrame)
Main Use	Deep learning, GPU ops	Numerical computing	Data analysis, tabular data
Dimensionality	nD (0D–nD)	nD (0D–nD)	1D & 2D
Data Type	Homogeneous	Homogeneous	Heterogeneous
GPU Support	Yes	No	No
Ecosystem	TF/PyTorch	SciPy, scikit-learn	Data science stack
File Handling	Not direct	Limited	Excellent (CSV, Excel, SQL)
Indexing	Integer-based	Integer-based	Label + integer-based

A **GPU** (Graphics Processing Unit) and a **TPU** (Tensor Processing Unit) are both hardware accelerators, but they're optimized for different types of workloads—especially in AI and deep learning.

1. GPU (Graphics Processing Unit)

- **Origin**: Originally designed for rendering graphics in games and visual applications.
- **Architecture**: Many parallel cores optimized for matrix and vector operations.
- **Use in AI**:
  - Excellent for training and inference of deep learning models.
  - Flexible — works with many frameworks (TensorFlow, PyTorch, MXNet, etc.).
- **Strengths**:
  - Versatile (can handle AI, scientific computing, video processing).
  - Wide hardware availability (NVIDIA, AMD).
- **Example Models**: NVIDIA RTX 4090, A100, AMD MI200 series.

2. TPU (Tensor Processing Unit)

- **Origin**: Custom-built by Google (2016) for accelerating tensor operations in deep learning.
- **Architecture**: ASIC (Application-Specific Integrated Circuit) optimized for **matrix multiplication** and tensor computations in TensorFlow.
- **Use in AI**:
  - Primarily used for Google's deep learning tasks, especially in TensorFlow & JAX.
  - Optimized for **training large models** and high-throughput inference.
- **Strengths**:
  - Extremely efficient for tensor-heavy workloads.
  - Lower latency and power consumption for AI tasks.
- **Availability**: Mostly on Google Cloud (not widely available for personal use).

PyTorch vs TensorFlow

Feature	PyTorch	TensorFlow
Developer	Facebook's AI Research (FAIR) Lab	Google Brain
First Release	2016	2015
Programming Style	**Dynamic computation graph** (define-by-run) – graph is built at runtime	**Static computation graph** (define-and-run, in TF 1.x) – but TF 2.x supports eager execution
Ease of Use	Pythonic, intuitive, great for research & experimentation	More verbose (TF 1.x), TF 2.x is more beginner-friendly
Debugging	Easy – uses standard Python debugging tools	More complex in TF 1.x, easier in TF 2.x

Performance	Excellent GPU acceleration via CUDA	Excellent GPU & TPU acceleration
Model Deployment	TorchScript, ONNX export, PyTorch Serve	TensorFlow Serving, TensorFlow Lite, TensorFlow.js
Community & Ecosystem	Huge research community, widely adopted in academia	Strong industry adoption, Google's ecosystem integration
Pre-trained Models	TorchHub, Hugging Face integration	TensorFlow Hub, TF Model Garden
Mobile & Embedded Support	PyTorch Mobile	TensorFlow Lite
Visualization	Uses TensorBoard (via add-on)	Built-in TensorBoard support
Production Ready	Gaining momentum in production (Meta, Tesla, OpenAI use it)	Very mature for production, used in Google services
Best Suited For	Research, fast prototyping, NLP (Transformers)	Production pipelines, cross-platform deployment

1.Create Tensors and perform basic operations with tensors

```
import numpy as np

# Create tensors (2D arrays)
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("Tensor A:\n", A)
print("Tensor B:\n", B)

# Basic operations
add_result = A + B
sub_result = A - B
mul_result = A * B         # Element-wise multiplication
div_result = A / B         # Element-wise division
matmul_result = A @ B      # Matrix multiplication
print("\nAddition:\n", add_result)
print("\nSubtraction:\n", sub_result)
print("\nElement-wise Multiplication:\n", mul_result)
print("\nElement-wise Division:\n", div_result)
print("\nMatrix Multiplication:\n", matmul_result)
```

using tensor flow

```
import tensorflow as tf

# Create constant tensors
A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
B = tf.constant([[5, 6], [7, 8]], dtype=tf.float32)

# Basic operations
```

```
add_result = tf.add(A, B)          # Addition
sub_result = tf.subtract(A, B)     # Subtraction
mul_result = tf.multiply(A, B)     # Element-wise multiplication
div_result = tf.divide(A, B)       # Element-wise division
matmul_result = tf.matmul(A, B)    # Matrix multiplication

print("Addition:\n", add_result.numpy())
print("Subtraction:\n", sub_result.numpy())
print("Multiplication:\n", mul_result.numpy())
print("Division:\n", div_result.numpy())
print("Matrix Multiplication:\n", matmul_result.numpy())
```

LAB PROGRAM 2: Create Tensors and Apply Split & Merge Operations + Statistics Operations

```
import numpy as np
# Create a 3x4 tensor
data = np.arange(1, 13).reshape(3, 4)
print("Original Tensor:\n", data)

# Split into 2 parts along columns (axis=1)
splits = np.split(data, 2, axis=1)
print("\nSplit Parts:")
for part in splits:
    print(part)

# Merge back into original tensor
merged = np.concatenate(splits, axis=1)
print("\nMerged Tensor:\n", merged)

# Statistics
mean_val = np.mean(data)
max_val = np.max(data)
min_val = np.min(data)
sum_val = np.sum(data)
std_val = np.std(data)

print("\nStatistics:")
print(f"Mean: {mean_val}")
print(f"Max: {max_val}")
print(f"Min: {min_val}")
print(f"Sum: {sum_val}")
print(f"Standard Deviation: {std_val}")
```

using tensor flow

```
import tensorflow as tf

# Create a tensor
data = tf.constant([[1, 2, 3, 4],
                    [5, 6, 7, 8],
```

```
                [9, 10, 11, 12]], dtype=tf.float32)

print("Original Tensor:\n", data.numpy())

# Split into 2 parts along columns (axis=1)
splits = tf.split(data, num_or_size_splits=2, axis=1)

print("\nSplit Parts:")
for part in splits:
    print(part.numpy())

# Merge back (concatenate along columns)
merged = tf.concat(splits, axis=1)
print("\nMerged Tensor:\n", merged.numpy())

# Statistics
mean_val = tf.reduce_mean(data)
max_val = tf.reduce_max(data)
min_val = tf.reduce_min(data)
sum_val = tf.reduce_sum(data)
std_val = tf.math.reduce_std(data)

print("\nStatistics:")
print(f"Mean: {mean_val.numpy()}")
print(f"Max: {max_val.numpy()}")
print(f"Min: {min_val.numpy()}")
print(f"Sum: {sum_val.numpy()}")
print(f"Standard Deviation: {std_val.numpy()}")
```

Design single unit perception for classification of iris dataset without using predefined models

Implementation of AND Gate

```
import numpy as np
import tensorflow as tf

# Example input data (4 samples, 2 features each)
X = tf.constant([[0.0, 0.0],
                 [0.0, 1.0],
                 [1.0, 0.0],
                 [1.0, 1.0]], dtype=tf.float32)

# Manually define weights and bias
W = tf.constant([[2.0],   # weight for feature 1
                 [2.0]],  # weight for feature 2
                dtype=tf.float32)
b = tf.constant([-3.0], dtype=tf.float32)  # bias
```

```
# Forward pass: z = XW + b
z = tf.matmul(X, W) + b

# Activation (step function or sigmoid)
y_step = tf.cast(z >= 0, tf.int32)# hard threshold (binary
output)
y_sigmoid = tf.math.sigmoid(z) # soft probability

print("Inputs:\n", X.numpy())
print("Linear combination (z):\n", z.numpy())
print("Step output (Perceptron):\n", y_step.numpy())
print("Sigmoid output (probability):\n", y_sigmoid.numpy())
```

IRIS DataSet

1. Import Libraries

```
import tensorflow as tf
import pandas as pd
import numpy as np
```

- **tensorflow** → to handle tensors, matrix multiplication, and activation functions.
- **pandas** → to load the CSV dataset.
- **numpy** → for numerical operations (like converting labels).

2. Load Dataset

```
df = pd.read_csv("iris.csv")
```

- Reads the **iris.csv** file into a dataframe.
- Standard Iris dataset has **4 features** (sepal length, sepal width, petal length, petal width) and **1 label** (species).

3. Select Features

```
X = df.iloc[:, 0:2].values   # First 2 features (sepal length, sepal width)
y = df.iloc[:, -1].values    # Last column → species
```

- Only uses **2 features** for simplicity (so we can visualize easily).
- `y` contains class labels: `"Iris-setosa"`, `"Iris-versicolor"`, `"Iris-virginica"`.

4. Convert to Binary Classification

```
y_binary = np.where(y == "Iris-setosa", 1.0, 0.0).astype(np.float32)
```

- We simplify the problem to **"Setosa vs Not-Setosa"**.
- If label = `"Iris-setosa"` → `1.0`

- Otherwise → `0.0`

So we now have a **binary classification problem**.

5. Normalize Features

```
X = (X - X.mean(axis=0)) / X.std(axis=0)
```

- Perceptrons work better if input values are on the same scale.
- This standardization ensures each feature has:
  - **Mean = 0**
  - **Standard deviation = 1**

6. Convert Data to TensorFlow Tensor

```
X_tf = tf.constant(X, dtype=tf.float32)
```

- Converts `numpy` array into a **TensorFlow tensor**, so we can apply matrix operations.

7. Define Weights and Bias

```
W = tf.constant([[1.0],
                 [-1.0]], dtype=tf.float32)
b = tf.constant([0.2], dtype=tf.float32)
```

- Since we have **2 input features**, the weight matrix is **2×1**.
- Bias is just a single number.
- These are manually chosen (not trained).

Mathematically:

$$z = XW + b$$

8. Forward Pass (Linear Combination)

```
z = tf.matmul(X_tf, W) + b
```

- `X_tf` is shape `(150, 2)`
- `W` is shape `(2, 1)`
- Result `z` is `(150, 1)` → each row is the weighted sum for one flower.

9. Step Activation (Hard Perceptron)

```
y_pred_step = tf.cast(z >= 0, tf.int32)
```

- If `z >= 0` → output = `1`
- Else → output = `0`
- This is the **classic perceptron decision rule**.

10. Sigmoid Activation (Soft Output)

```
y_pred_sigmoid = tf.math.sigmoid(z)
```

- Sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Outputs probability between **0 and 1**.
- Useful if you want **confidence** instead of a hard decision.

11. Print Results

```
print("First 10 True labels (Setosa=1, Others=0):", y_binary[:10])
print("First 10 Step outputs:", y_pred_step.numpy().flatten()[:10])
print("First 10 Sigmoid probabilities:",
y_pred_sigmoid.numpy().flatten()[:10])
```

- Shows comparison between:
  - True labels (`y_binary`)
  - Perceptron hard outputs (`y_pred_step`)
  - Probabilities (`y_pred_sigmoid`)

Summary

- This program **does not train weights** — it just demonstrates how a **single perceptron computes outputs** with chosen weights & bias.
- **Step function** → classification (hard decision).
- **Sigmoid function** → probability interpretation.

4) Design, train and test the MLP for tabular data and verify various activation functions and optimizers tensor flow.

0) Imports

```
import tensorflow as tf
import numpy as np
import pandas as pd
```

- `pandas reads the CSV, numpy handles arrays, tensorflow.keras builds & trains the neural net.`

1) Load data & split features/labels

```
data = pd.read_csv("iris.csv")
```

```
X = data.iloc[:, :-1].values      # first 4 columns → features
y_str = data.iloc[:, -1].values   # last column → label
strings
```

- Assumes the CSV has 5 columns: 4 numeric features (sepal length/width, petal length/width) and 1 label column (species).
- Shapes (for the classic Iris dataset with 150 rows): X.shape == (150, 4), y_str.shape == (150,).

1a) Encode labels (string → int → one-hot)

```
classes = np.unique(y_str)
class_to_int = {c: i for i, c in enumerate(classes)}
y = np.array([class_to_int[label] for label in y_str])
y = tf.keras.utils.to_categorical(y, num_classes=3)
```

- np.unique finds the set of class names and (importantly) **sorts** them alphabetically.
  Example mapping might become: {'setosa':0, 'versicolor':1, 'virginica':2} (order depends on np.unique).
- After integer mapping: y.shape == (150,) with values in {0,1,2}.
- After one-hot: y.shape == (150, 3); each row like [1,0,0], [0,1,0], or [0,0,1].

Why: Neural nets trained with softmax + categorical cross-entropy expect one-hot class targets.

2) Train-test split (manual, no sklearn)

```
num_samples = X.shape[0]
indices = np.arange(num_samples)
np.random.shuffle(indices)

train_size = int(0.8 * num_samples)
train_idx, test_idx = indices[:train_size],
indices[train_size:]

X_train, X_test = X[train_idx], X[test_idx]
y_train, y_test = y[train_idx], y[test_idx]
```

- Randomly shuffles row indices, then takes ~80% for training, ~20% for testing.
  With 150 rows → train=120, test=30.
- Shapes:
  X_train (120,4), y_train (120,3), X_test (30,4), y_test (30,3) (approx).

Tip: For reproducible splits, set seeds before shuffling:

```
np.random.seed(42)
tf.random.set_seed(42)
```

2a) Feature normalization (learn on train only)

```
normalizer = tf.keras.layers.Normalization()
normalizer.adapt(X_train)
```

- Normalization is a **stateful** layer: adapt computes the training set mean/variance per feature.
- During training/inference it standardizes inputs: (x - mean) / std.
- Adapting on **train only** avoids test-set information leakage.

3) Build the MLP

```
def build_mlp(activation="relu", optimizer="adam"):
    model = tf.keras.Sequential([
        normalizer,                          # input
standardization
        tf.keras.layers.Dense(8, activation=activation),
        tf.keras.layers.Dense(3, activation="softmax")
    ])
    model.compile(optimizer=optimizer,
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model
```

- Architecture:
  Normalization → Dense(8, act) → Dense(3, softmax).
  Hidden layer has 8 neurons; output layer has 3 neurons (one per class) with softmax.
- categorical_crossentropy matches the one-hot labels.
- You'll try different activation functions (ReLU / Sigmoid / Tanh) and optimizers (Adam / SGD / RMSprop) via the loop below.

4) Train & evaluate across activations/optimizers

```
activations = ["relu", "sigmoid", "tanh"]
optimizers = ["adam", "sgd", "rmsprop"]
results = {}

for act in activations:
    for opt in optimizers:
        print(f"\nTraining with Activation={act},
Optimizer={opt}")
        model = build_mlp(activation=act, optimizer=opt)
        history = model.fit(
            X_train, y_train,
            epochs=50, batch_size=8,
            validation_data=(X_test, y_test),
            verbose=0
        )
 test_loss, test_acc = model.evaluate(X_test, y_test ,
verbose=0)
        results[(act, opt)] = test_acc
        print(f"Test Accuracy: {test_acc:.4f}")
```

This runs **9 experiments** (3 activations × 3 optimizers), each:

- Rebuilds a fresh model (so weights don't carry over).

- Trains for 50 epochs, mini-batches of 8.
- Tracks validation metrics on the test split each epoch (shown as val_* in history.history if you inspected it).
- Finally evaluates once on the test set and stores the test accuracy in results.

Notes:

- On Iris, you'll often see >90% accuracy with a simple MLP after normalization.
- verbose=0 just suppresses the training logs for cleaner output.
- You could add early stopping to avoid overfitting:
- cb = tf.keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)
- model.fit(..., callbacks=[cb])

5) Print

```
print("\nSummary of Accuracies:")
for (act, opt), acc in results.items():
    print(f"Activation={act:7} | Optimizer={opt:7} |
Accuracy={acc:.4f}")
```

- Iterates through the stored results and prints each combo's final test accuracy.
- (Optional) You can also show the best combo:
- best = max(results.items(), key=lambda kv: kv[1])

```
print(f"\nBest: Activation={best[0][0]},
Optimizer={best[0][1]}, Acc={best[1]:.4f}")
```

Activation Functions in the Program

The program tests three activation functions for the hidden layer of the MLP:

1. ReLU (Rectified Linear Unit)

- **Definition:**

$$f(x) = max(0, x)$$

- **How it works:**
  - Returns the input value if it is positive; otherwise, outputs zero.
  - Introduces non-linearity while being computationally efficient.
- **Advantages:**
  - Avoids vanishing gradient problem (common in Sigmoid/Tanh).
  - Allows faster convergence.
- **Limitations:**
  - Can suffer from "dead neurons" where weights never update if inputs become negative permanently.

2. Sigmoid

- **Definition:**

$$f(x) = \frac{1}{1+e^{-x}}$$

- **How it works:**
  - Squashes input values into a range between **0 and 1.**
- **Advantages:**
  - Good for probabilistic interpretation.
- **Limitations:**
  - Saturates for large positive/negative inputs, leading to **vanishing gradients.**
  - Slower convergence compared to ReLU.

3. Tanh (Hyperbolic Tangent)

- **Definition:**

$$f(x) = tanhx) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- **How it works:**
  - Outputs values between **-1 and 1.**
  - Centered around zero, which helps optimization compared to sigmoid.
- **Advantages:**
  - Better than Sigmoid for hidden layers due to zero-centering.
- **Limitations:**
  - Still suffers from vanishing gradient for large inputs.

Output Layer Activation – Softmax

- **Definition:**

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_j z_j}$$

- **Purpose:**
  - Converts raw outputs (logits) into probabilities across three classes (Iris species).
- **Why used:**
  - Required for **multi-class classification** with categorical_crossentropy loss.

Optimizers in the Program

The optimizer determines **how weights are updated** during training.

1. Adam (Adaptive Moment Estimation)

- **Key Idea:** Combines momentum and RMSprop.
- **Advantages:**
  - Fast convergence.
  - Adapts learning rates individually for each parameter.
  - Works well for most tasks without much tuning.

2. SGD (Stochastic Gradient Descent)

- **Key Idea:** Updates parameters in the opposite direction of the gradient with a fixed learning rate.
- **Advantages:**
  - Simple and robust.
  - Works well when learning rate is tuned properly.
- **Limitations:**
  - Convergence can be slow.
  - May get stuck in local minima without momentum.

3. RMSprop

- **Key Idea:** Maintains a moving average of squared gradients to adapt the learning rate.
- **Advantages:**
  - Works well for non-stationary objectives.
  - Handles varying gradient magnitudes better than SGD.
- **Use Cases:** Often used for RNNs and problems with noisy gradients.

Why Compare Them in the Program?

- Different combinations of **activation functions** and **optimizers** can impact:
  - Speed of convergence.
  - Ability to avoid vanishing/exploding gradients.
  - Final accuracy.
- Running multiple experiments helps identify the best combination for the Iris dataset.

Activation Functions

Activation	Range	Formula	Advantages	Limitations
ReLU	[0, ∞)	$f(x) = max(0,x)$	Fast convergence, avoids vanishing gradient in positive range	Dead neurons if weights drive input negative
Sigmoid	(0, 1)	$f(x) = \frac{1}{1+e^{-x}}$	Smooth, probabilistic interpretation	Vanishing gradient for large
Tanh	(-1, 1)	$f(x) = tanhx) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	Zero-centered output, better than Sigmoid for hidden layers	Still suffers from vanishing gradient at extremes
Softmax (Output Layer)	(0, 1), sum=1	$Softmax(z_i) = \frac{e^{z_i}}{\sum_j z_j}$	Produces class probabilities for multi-class problems	Sensitive to very large logits (requires normalization)

Optimizers

Optimizer	Update Method	Advantages	Limitations
Adam	Combines Momentum + RMSprop; adaptive learning rate for each parameter	Fast, works well without much tuning	Can converge to suboptimal minima if learning rate not adjusted
SGD	Updates parameters using gradient and fixed learning rate	Simple, works well with good learning rate tuning	Slow convergence, may get stuck in local minima
RMSprop	Uses moving average of squared gradients to scale learning rate	Handles noisy gradients, adapts learning rates effectively	May still require tuning of initial learning rate

Gradient Descent
Definition:
Gradient Descent is an optimization algorithm used to minimize a loss (or cost) function by iteratively updating model parameters (weights and biases) in the opposite direction of the gradient of the loss function.

Convergence
Definition:
Convergence refers to the point during training where the learning algorithm (e.g., gradient descent) reaches a state where further updates to the model parameters (weights and biases) result in minimal or no improvement in the loss function or accuracy.

---

5) Design and implement to classify 32x32 images using MLP using tensor flow/keras and check the accuracy.

1) Imports & environment setup

```
import os
import pickle
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
```

```
from sklearn.metrics import confusion_matrix ,
classification_report
```

- os, pickle, numpy: standard utilities for file I/O and arrays.
- tensorflow: building, training the model.
- matplotlib & seaborn: plotting (images, confusion heatmap).
- sklearn.metrics: convenience functions to compute confusion matrix and classification report (precision/recall/F1).

```
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
tf.keras.backend.clear_session()
```

- CUDA_VISIBLE_DEVICES = "-1" forces TensorFlow to run on **CPU only** (useful when GPU/CUDA drivers cause kernel crashes).
- tf.keras.backend.clear_session() clears existing TF graphs/models from memory — avoids memory buildup if you re-run cells.

2) Load CIFAR-10 batch from pickle

```
with open("data_batch_1", "rb") as f:
    batch = pickle.load(f, encoding='bytes')

X = batch[b'data']
y = np.array(batch[b'labels'])
```

- data_batch_1 is one CIFAR-10 batch (usually 10,000 samples).
- batch[b'data'] is an array shape (N, 3072) where 3072 = 32*32*3 (flattened RGB image).
- y is an integer label array (0..9) shape (N,).

3) Reshape to image tensors and normalize

```
X_images = X.reshape(-1, 3, 32, 32).transpose(0, 2, 3,
1).astype("float32") / 255.0
```

Step-by-step:

1. X.reshape(-1, 3, 32, 32) — interprets each flat row as (3,32,32) where channel-first layout is used in CIFAR files.
2. .transpose(0,2,3,1) → moves channels to last position producing (N, 32, 32, 3) which is the common TF image format.
3. .astype("float32") / 255.0 → converts pixel integers [0,255] → floats [0.0, 1.0] (standard normalization for NN inputs).

Example shapes (typical CIFAR batch):

- X originally: (10000, 3072)
- X_images: (10000, 32, 32, 3) (float32)

- y: (10000,)

4) One-hot encode labels

```
num_classes = 10
y_cat = tf.keras.utils.to_categorical(y,
num_classes=num_classes)
```

- Converts y from shape (N,) with integer labels to one-hot matrix (N, 10).
- Example: label 3 → [0,0,0,1,0,0,0,0,0,0].

5) Subset to keep memory small

```
X_train, y_train = X_images[:2000], y_cat[:2000]
X_test, y_test = X_images[2000:2500], y_cat[2000:2500]
```

- Uses **first 2000** images for training and next **500** for testing.
- This keeps RAM usage low so the notebook/kernel doesn't die.
- Shapes now: X_train (2000,32,32,3), y_train (2000,10), X_test (500,32,32,3), y_test (500,10).

Printed lines confirm shapes:

```
print("Train set:", X_train.shape, y_train.shape)
print("Test set:", X_test.shape, y_test.shape)
```

6) Define a small MLP model

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(32, 32, 3)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])
```

- Flatten: converts (32,32,3) → vector length 3072 per sample.
- Dense(64) & Dense(32): fully connected hidden layers with ReLU nonlinearity.
- Dense(num_classes, softmax): output layer giving class probabilities across 10 classes.
- **Parameter counts** (approx):
  - First dense: 3072*64 + 64 = 196,672 params
  - Second dense: 64*32 + 32 = 2,080 params
  - Output dense: 32*10 + 10 = 330 params
  - **Total ≈ 199k parameters** (model.summary() prints exact numbers).

Important note: flattening destroys spatial structure — MLPs on raw images typically perform poorly compared with CNNs (which use conv layers to exploit locality).

7) Compile the model

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

- optimizer='adam': adaptive optimizer (common default).

- loss='categorical_crossentropy': appropriate because labels are one-hot encoded.
- metrics=['accuracy']: track accuracy during training and evaluation.

model.summary() prints layer names, output shapes, and parameter counts.

8) Train the model

```
history = model.fit(X_train, y_train,
                    epochs=5,
                    batch_size=32,
                    validation_data=(X_test, y_test),
                    verbose=1)
```

- epochs=5: number of times to iterate over training data (kept small for speed).
- batch_size=32: number of samples per gradient update.
- validation_data=(X_test, y_test): after each epoch TF evaluates loss/accuracy on test set for monitoring.
- history object contains per-epoch loss & accuracy arrays (history.history['loss'], 'val_loss', 'accuracy', etc.).

9) Evaluate on train and test sets

```
train_loss, train_acc = model.evaluate(X_train, y_train,
verbose=0)
test_loss, test_acc = model.evaluate(X_test, y_test,
verbose=0)
```

- model.evaluate returns (loss, accuracy) for the provided dataset. Gives you final metrics after training.
- Printed values:

```
print(f"Training Accuracy: {train_acc:.4f}, Training Loss:
{train_loss:.4f}")
print(f"Testing Accuracy: {test_acc:.4f}, Testing Loss:
{test_loss:.4f}")
```

10) Predictions (probabilities → class labels)

```
y_pred_probs = model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test, axis=1)
```

- model.predict returns probabilities shape (500, 10).
- np.argmax(..., axis=1) converts one-hot or probs to integer predicted labels (0..9).

11) Confusion matrix (visual)

```
cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
xticklabels=[...], yticklabels=[...])
```

- confusion_matrix(true, pred) builds a 10×10 matrix:
  - **rows** = true classes, **columns** = predicted classes.
  - Element (i,j) = number of samples of true class i predicted as class j.

- The seaborn heatmap shows misclassification patterns at a glance. Diagonal entries = correct predictions.

12) Classification report

```
print(classification_report(y_true, y_pred,
target_names=[...]))
```

- Shows **precision, recall, f1-score, support** per class:
  - **Precision** = TP / (TP + FP) — of all predicted class X, how many were correct.
  - **Recall** = TP / (TP + FN) — of all true class X, how many were found.
  - **F1-score** = harmonic mean of precision & recall.
  - **Support** = number of true samples for that class in the test set.

This report helps identify which classes the model struggles with.

****

9) Design and implement a CNN model to classify multi category JPG images with tensor flow / keras and check accuracy. Predict labels for new images.

Step-by-Step Explanation

Importing Libraries

```
import pickle
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix,
classification_report
```

- **pickle** → loads CIFAR-10 dataset stored as a binary file (.pkl or .bin).
- **numpy (np)** → handles numerical operations and reshaping arrays.
- **tensorflow (tf)** → framework for building and training the CNN.
- **matplotlib.pyplot (plt)** → plots sample images and graphs.
- **seaborn (sns)** → plots a colored confusion-matrix heatmap.
- **sklearn.metrics** → computes confusion matrix and classification report.

Optional GPU Settings

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

- Tells TensorFlow **not to use the GPU** (runs only on CPU).
  This prevents CUDA crashes or GPU memory errors if the `GPU setup is unstable.`

1 Load CIFAR-10 Data

```
with open("data_batch_1", "rb") as f:
    batch = pickle.load(f, encoding='bytes')
```

- Opens the **CIFAR-10** data file data_batch_1 in binary ('rb') mode.
- Loads the dictionary containing:
  - b'data' → flattened image pixel values.
  - b'labels' → class labels (0–9).

Extract Images and Labels

```
X = batch[b'data']
y = np.array(batch[b'labels'])
```

- X is a NumPy array of size (10000 × 3072)
  → each image is a flat vector of 3072 numbers (32×32×3).
- y is a 1-D array of integers (0–9) representing classes.

2 Reshape & Normalize

```
X_images = X.reshape(-1, 3, 32, 32).transpose(0, 2, 3,
1).astype("float32") / 255.0
```

Breakdown

1. **.reshape(-1, 3, 32, 32)**
   - Converts each flat vector into (3, 32, 32) → 3 color channels.
   - -1 means "infer the number of samples automatically".
2. **.transpose(0, 2, 3, 1)**
   - Rearranges axes to **(N, 32, 32, 3)** → TensorFlow expects this format.
3. **.astype("float32") / 255.0**
   - Converts pixel values from 0–255 → **0–1 range** for faster convergence.

Label Names

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

- Human-readable names for CIFAR-10's numeric labels 0–9.

3 One-Hot Encode Labels

```
num_classes = 10
y_cat = tf.keras.utils.to_categorical(y, num_classes =
num_classes)
```

- Converts numeric labels (y = [3, 8, 0, ...]) into one-hot encoded vectors:
  e.g., class 3 → [0,0,0,1,0,0,0,0,0,0].

- Required by categorical_crossentropy loss function.

4 Train/Test Split

```
X_train, y_train = X_images[:4000], y_cat[:4000]
X_test, y_test = X_images[4000:5000], y_cat[4000:5000]
```

- **Uses 4000 images for training and 1000 for testing (subset to reduce memory/time).**

```
print("Train set:", X_train.shape, y_train.shape)
print("Test set:", X_test.shape, y_test.shape)
```

- **Prints dataset dimensions for confirmation.**

5 Define CNN Architecture

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])
```

Explanation of Layers

Layer	Purpose
Conv2D(32, (3,3))	32 filters detect local patterns (edges, shapes).
MaxPooling2D(2,2)	Downsamples by taking the max value in 2×2 regions → reduces spatial size.
Conv2D(64)	Learns more abstract features (textures, parts).
MaxPooling2D(2,2)	Further spatial reduction.
Conv2D(128)	Detects high-level features (objects, color blobs).
Flatten()	Converts the 3D feature maps to a 1D vector.
Dense(128, relu)	Fully connected layer for learning combinations of features.
Dense(num_classes, softmax)	Output layer → gives probabilities across 10 classes.

6 Compile the Model

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

```
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- **optimizer='adam'** → adaptive gradient optimizer (efficient & widely used).
- **loss='categorical_crossentropy'** → appropriate for multi-class classification.
- **metrics=['accuracy']** → tracks accuracy during training/evaluation.

Show Model Structure

```
model.summary()
```

- Prints each layer's output shape and number of parameters.

7 Train the Model

```
history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=64,
                    validation_data=(X_test, y_test),
                    verbose=1)
```

- **.fit()** trains the CNN
  - Epochs =10 → full passes through training data.
  - batch_size=64 → number of samples per training step.
  - validation_data → evaluates on test set after each epoch.
  - verbose=1 → prints progress.

8 Evaluate Model

```
train_loss, train_acc = model.evaluate(X_train, y_train,
verbose=0)
test_loss, test_acc = model.evaluate(X_test, y_test,
verbose=0)

print(f"\n Training Accuracy: {train_acc:.4f}, Loss:
{train_loss:.4f}")
print(f" Testing Accuracy: {test_acc:.4f}, Loss:
{test_loss:.4f}")
```

- Measures performance on both train and test sets.
- **Accuracy** = correct predictions ÷ total samples.
- **Loss** = measure of prediction error (lower is better).

9 Confusion Matrix & Classification Report

```
y_pred_probs = model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test, axis=1)
```

- **y_pred_probs** → model's softmax probabilities (e.g., [0.02, 0.01, 0.9, ...]).
- **np.argmax(..., axis=1)** → converts probabilities → predicted class index.

```
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=label_names, yticklabels=label_names)
plt.title("Confusion Matrix - CIFAR-10 (data_batch_1 subset)")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

- Builds a **10×10 matrix** showing how often each class was confused with another.
- sns.heatmap() plots this matrix as a color-coded grid.

```
print(classification_report(y_true, y_pred,
target_names=label_names))
```

- Prints **precision**, **recall**, and **F1-score** per class.

Predict New Images

```
num_samples = 5
idx = np.random.randint(0, len(X_test), num_samples)
sample_images = X_test[idx]
sample_labels = y_true[idx]
sample_preds = np.argmax(model.predict(sample_images), axis=1)
```

- Randomly picks 5 test images and predicts their classes.

```
plt.figure(figsize=(10, 4))
for i in range(num_samples):
    plt.subplot(1, num_samples, i+1)
    plt.imshow(sample_images[i])
    plt.title(f"True: {label_names[sample_labels[i]]}\nPred:
{label_names[sample_preds[i]]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

- Displays those 5 images with both **true** and **predicted** labels on top.

11. Implement a CNN architecture (LeNet, Alexnet, VGG, etc) model to classify multi category Satellite images with tensor flow / keras and check the accuracy. Check whether your model is overfit / underfit / perfect fit and apply the techniques to avoid overfit and underfit.

```
import pickle
import numpy as np
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# Load data_batch_1
with open("data_batch_1", "rb") as f:
    batch = pickle.load(f, encoding='bytes')

X = batch[b'data']
y = np.array(batch[b'labels'])

# Reshape to images (N, 32, 32, 3)
X_images = X.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1).astype("float32") /
255.0

# One-hot encode labels
y_cat = to_categorical(y, num_classes=10)

# Train-Test split
X_train, X_test, y_train, y_test = train_test_split(X_images, y_cat,
test_size=0.2, random_state=42)
print("Data loaded:", X_train.shape, X_test.shape)
```

#Lenet

```
from tensorflow.keras import layers, models

model_lenet = models.Sequential([
    layers.Conv2D(6, (5,5), activation='tanh', input_shape=(32,32,3),
padding='same'),
    layers.AveragePooling2D(),
    layers.Conv2D(16, (5,5), activation='tanh'),
    layers.AveragePooling2D(),
    layers.Flatten(),
    layers.Dense(120, activation='tanh'),
    layers.Dense(84, activation='tanh'),
    layers.Dense(10, activation='softmax')
])

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

```
model_lenet.fit(X_train, y_train, epochs=5, batch_size=64,
validation_data=(X_test, y_test))
```

#AlexNet (Simplified for CIFAR-10)
#Original AlexNet used 224×224 input; we scale it down for 32×32 images.

```
model_alex = models.Sequential([
    layers.Conv2D(96, (3,3), activation='relu', input_shape=(32,32,3),
padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(384, (3,3), activation='relu', padding='same'),
    layers.Conv2D(384, (3,3), activation='relu', padding='same'),
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model_alex.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model_alex.summary()
model_alex.fit(X_train, y_train, epochs=5, batch_size=64,
validation_data=(X_test, y_test))
```

#ZF-Net (Zeiler & Fergus Network)
#A refinement of AlexNet with smaller strides and deeper visualization features.

```
model_zf = models.Sequential([
    layers.Conv2D(96, (3,3), strides=1, activation='relu',
input_shape=(32,32,3), padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(256, (3,3), strides=1, activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(384, (3,3), activation='relu', padding='same'),
    layers.Conv2D(384, (3,3), activation='relu', padding='same'),
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

```
model_zf.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model_zf.summary()
model_zf.fit(X_train, y_train, epochs=5, batch_size=64,
validation_data=(X_test, y_test))

#VGGNet (VGG-11 Simplified)
#VGG uses small 3×3 filters, deep layers, and uniform architecture.
model_vgg = models.Sequential([
    layers.Conv2D(64, (3,3), activation='relu', input_shape=(32,32,3),
padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(128, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model_vgg.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model_vgg.summary()
model_vgg.fit(X_train, y_train, epochs=5, batch_size=64,
validation_data=(X_test, y_test))

#GoogLeNet (Inception Simplified Block)
#Inception modules concatenate multiple filter outputs.
from tensorflow.keras.layers import concatenate
def inception_module(x, filters):
    f1, f3r, f3, f5r, f5, fpp = filters
    path1 = layers.Conv2D(f1, (1,1), activation='relu', padding='same')(x)
    path2 = layers.Conv2D(f3r, (1,1), activation='relu', padding='same')(x)
    path2 = layers.Conv2D(f3, (3,3), activation='relu', padding='same')(path2)
    path3 = layers.Conv2D(f5r, (1,1), activation='relu', padding='same')(x)
    path3 = layers.Conv2D(f5, (5,5), activation='relu', padding='same')(path3)
    path4 = layers.MaxPooling2D((3,3), strides=(1,1), padding='same')(x)
    path4 = layers.Conv2D(fpp, (1,1), activation='relu', padding='same')(path4)
    return concatenate([path1, path2, path3, path4], axis=-1)

input_layer = layers.Input(shape=(32,32,3))
x = inception_module(input_layer, [32, 32, 32, 16, 16, 16])
x = layers.MaxPooling2D((2,2))(x)
x = inception_module(x, [64, 48, 64, 16, 32, 32])
x = layers.GlobalAveragePooling2D()(x)
output = layers.Dense(10, activation='softmax')(x)
```

```
model_google = models.Model(input_layer, output)
model_google.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model_google.summary()
model_google.fit(X_train, y_train, epochs=5, batch_size=64,
validation_data=(X_test, y_test))
```

#ResNet (Residual Network – Simplified)
#Skip connections prevent vanishing gradients.

```
from tensorflow.keras import Model, Input
def res_block(x, filters):
    shortcut = x
    x = layers.Conv2D(filters, (3,3), padding='same', activation='relu')(x)
    x = layers.Conv2D(filters, (3,3), padding='same')(x)
    x = layers.add([shortcut, x])
    x = layers.Activation('relu')(x)
    return x

input_layer = Input(shape=(32,32,3))
x = layers.Conv2D(32, (3,3), activation='relu', padding='same')(input_layer)
x = res_block(x, 32)
x = layers.MaxPooling2D((2,2))(x)
x = res_block(x, 64)
x = layers.GlobalAveragePooling2D()(x)
output_layer = layers.Dense(10, activation='softmax')(x)

model_resnet = Model(inputs=input_layer, outputs=output_layer)
model_resnet.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model_resnet.summary()
model_resnet.fit(X_train, y_train, epochs=5, batch_size=64,
validation_data=(X_test, y_test))
```

6. Design and implement a simple RNN model with tensor flow / keras and check accuracy.

```
# Import required libraries
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
```

```
import matplotlib.pyplot as plt
import seaborn as sns
# 1 Load Dataset
df = pd.read_csv("netflix_titles.csv.csv")
print("Dataset Loaded ")
print(df.head())

# 2 Select relevant columns
df = df[['type', 'description']].dropna()

# 3 Encode target labels (Movie=0, TV Show=1)
label_encoder = LabelEncoder()
df['type_encoded'] = label_encoder.fit_transform(df['type'])

# 4 Text preprocessing
texts = df['description'].values
labels = df['type_encoded'].values

# 5 Tokenize text data
vocab_size = 5000
tokenizer = Tokenizer(num_words=vocab_size, oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
padded = pad_sequences(sequences, maxlen=100, padding='post',
truncating='post')

# 6 Split data
X_train, X_test, y_train, y_test = train_test_split(padded, labels,
test_size=0.2, random_state=42)

# 7 Build RNN Model
model = Sequential([
    Embedding(vocab_size, 64, input_length=100),
    SimpleRNN(64, activation='tanh'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

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

# 9 Train Model
history = model.fit(X_train, y_train, epochs=5, batch_size=32,
validation_split=0.2, verbose=1)

#10 Evaluate Model
y_pred = (model.predict(X_test) > 0.5).astype("int32")
acc = accuracy_score(y_test, y_pred)
```

```
print(f"\n Test Accuracy: {acc:.4f}")

# 11 Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Movie',
'TV Show'], yticklabels=['Movie', 'TV Show'])
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix - RNN Netflix Type Classification")
plt.show()

# 12 Classification Report
print("\n Classification Report:\n")
print(classification_report(y_test, y_pred, target_names=['Movie', 'TV Show']))
```

7 Design and implement LSTM model with tensor flow / keras and check accuracy.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# 1) Load dataset
df = pd.read_csv("netflix_titles.csv.csv")   # your dataset
df = df[['type', 'description']].dropna()

# Convert labels (Movie / TV Show) → numbers
le = LabelEncoder()
df['label'] = le.fit_transform(df['type'])

texts = df['description'].values
labels = df['label'].values

# 2) Tokenization + Padding
max_words = 5000        # only keep top 5000 words
max_len = 150           # cut or pad all descriptions to length 150

tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
tokenizer.fit_on_texts(texts)

sequences = tokenizer.texts_to_sequences(texts)
padded_sequences = pad_sequences(sequences, maxlen=max_len,
padding='post')
```

```
# 3) Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    padded_sequences,
    labels,
    test_size=0.2,
    random_state=42
)

# 4) Build LSTM Model

model = models.Sequential([
    layers.Embedding(input_dim=max_words, output_dim=64,
input_length=max_len),
    layers.LSTM(64, return_sequences=False),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

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

# Print model summary
model.summary()

# 5) Train the model

history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=5,
    batch_size=32,
    verbose=1
)

# 6) Evaluate Accuracy

loss, acc = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {acc:.4f}")
print(f"Test Loss: {loss:.4f}")

# 7) Plot Loss & Accuracy Graphs
import matplotlib.pyplot as plt

# Accuracy graph
plt.plot(history.history['accuracy'], label='Train Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("LSTM Accuracy")
plt.legend()
plt.show()

# Loss graph
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("LSTM Loss")
plt.legend()
plt.show()


# 8) Predict new text

sample = ["Two teenage friends discover a dark secret in their
neighborhood."]
seq = tokenizer.texts_to_sequences(sample)
pad = pad_sequences(seq, maxlen=max_len, padding='post')

pred = model.predict(pad)[0][0]

print("\nPrediction:")
print("TV Show" if pred > 0.5 else "Movie")
```

8 Design and implement GRU model with tensor flow / keras and check accuracy.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, GRU, Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# 1. LOAD NETFLIX DATASET

df = pd.read_csv("netflix_titles.csv.csv")   # your dataset


# Keep only necessary columns
df = df[['type', 'description']].dropna()
```

```
# Encode labels: Movie=0, TV Show=1
label_encoder = LabelEncoder()
df['label'] = label_encoder.fit_transform(df['type'])

texts = df['description'].astype(str).tolist()
labels = df['label'].values

# 2. TEXT TOKENIZATION

vocab_size = 5000
max_len = 100

tokenizer = Tokenizer(num_words=vocab_size, oov_token="<OOV>")
tokenizer.fit_on_texts(texts)

sequences = tokenizer.texts_to_sequences(texts)
padded = pad_sequences(sequences, maxlen=max_len, padding='post')

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    padded, labels, test_size=0.2, random_state=42
)

# 3. BUILD GRU MODEL

model = Sequential([
    Embedding(vocab_size, 64, input_length=max_len),
    GRU(64, return_sequences=False),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

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

# Show summary
model.summary()

# 4. TRAIN MODEL

history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=5,
    batch_size=64
)
```

```
# 5. ACCURACY & LOSS

loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\n Test Accuracy: {acc:.4f}")
print(f" Test Loss: {loss:.4f}")
```

12 Implement an Auto encoder to de-noise image.

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
UpSampling2D

# 1. Load CIFAR-10 data_batch_1

with open("data_batch_1", "rb") as f:
    batch = pickle.load(f, encoding='bytes')

X = batch[b'data']
X = X.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
X = X.astype("float32") / 255.0    # normalize

print("Dataset loaded:", X.shape)

# 2. Add Gaussian noise

noise_factor = 0.2
X_noisy = X + noise_factor * np.random.randn(*X.shape)
X_noisy = np.clip(X_noisy, 0., 1.)

# Split into training/testing
X_train, X_test = X_noisy[:4000], X_noisy[4000:5000]
Y_train, Y_test = X[:4000], X[4000:5000]   # original clean images

# 3. Build Convolutional Autoencoder

input_img = Input(shape=(32, 32, 3))

# --- Encoder ---
x = Conv2D(32, (3,3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2,2), padding='same')(x)

x = Conv2D(16, (3,3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2,2), padding='same')(x)
```

```
# --- Decoder ---
x = Conv2D(16, (3,3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2,2))(x)

x = Conv2D(32, (3,3), activation='relu', padding='same')(x)
x = UpSampling2D((2,2))(x)

decoded = Conv2D(3, (3,3), activation='sigmoid', padding='same')(x)

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

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

autoencoder.summary()

# 4. Train Autoencoder

history = autoencoder.fit(
    X_train, Y_train,
    epochs=10,
    batch_size=64,
    validation_data=(X_test, Y_test)
)


# 5. Evaluate model

loss = autoencoder.evaluate(X_test, Y_test, verbose=0)
print(f"\n Test Reconstruction Loss (MSE): {loss:.4f}")


# 6. Denoise images and visualize


decoded_imgs = autoencoder.predict(X_test[:10])

plt.figure(figsize=(10, 4))
for i in range(10):
    # Noisy
    ax = plt.subplot(2, 10, i+1)
    plt.imshow(X_test[i])
    plt.title("Noisy")
    plt.axis('off')

    # Denoised
    ax = plt.subplot(2, 10, i+11)
    plt.imshow(decoded_imgs[i])
    plt.title("Clean")
```

```
    plt.axis('off')

plt.tight_layout()
plt.show()
```

13. Implement a GAN application to convert images

```
import os
import pickle
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# 1. LOAD CIFAR-10 BATCH

with open("data_batch_1", "rb") as f:
    batch = pickle.load(f, encoding="bytes")

X = batch[b"data"]
X = X.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
X = (X.astype("float32") - 127.5) / 127.5    # normalize to [-1, 1]

BUFFER_SIZE = 5000
BATCH_SIZE = 64

dataset =
tf.data.Dataset.from_tensor_slices(X).shuffle(BUFFER_SIZE).batch(BATCH_S
IZE)


# 2. BUILD GENERATOR

def build_generator():
    model = tf.keras.Sequential([
        layers.Dense(8*8*256, use_bias=False, input_shape=(100,)),
        layers.BatchNormalization(),
        layers.LeakyReLU(),

        layers.Reshape((8, 8, 256)),

        layers.Conv2DTranspose(128, (5,5), strides=2, padding='same',
use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),

        layers.Conv2DTranspose(64, (5,5), strides=2, padding='same',
use_bias=False),
        layers.BatchNormalization(),
```

```
        layers.LeakyReLU(),

        layers.Conv2DTranspose(3, (5,5), strides=1, padding='same',
use_bias=False, activation='tanh')
    ])
    return model

generator = build_generator()


# 3. BUILD DISCRIMINATOR

def build_discriminator():
    model = tf.keras.Sequential([
        layers.Conv2D(64, (5,5), strides=2, padding='same', input_shape=[32,
32, 3]),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Conv2D(128, (5,5), strides=2, padding='same'),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Flatten(),
        layers.Dense(1)
    ])
    return model

discriminator = build_discriminator()


# 4. LOSS FUNCTIONS + OPTIMIZERS

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# 5. TRAINING LOOP

EPOCHS = 2000
```

```
noise_dim = 100
seed = tf.random.normal([16, noise_dim])

@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    # Train Generator
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_gen = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_disc = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_gen,
generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_disc,
discriminator.trainable_variables))

    return gen_loss, disc_loss

def train(dataset, epochs):
    for epoch in range(epochs):
        for image_batch in dataset:
            g, d = train_step(image_batch)

        if epoch % 200 == 0:
            print(f'Epoch {epoch}  Generator Loss: {g:.4f}  Disc Loss: {d:.4f}')
            generate_and_save(epoch)

# 6. IMAGE GENERATION

def generate_and_save(epoch):
    predictions = generator(seed, training=False)
    predictions = (predictions + 1) / 2.0

    plt.figure(figsize=(4,4))
    for i in range(16):
        plt.subplot(4,4,i+1)
        plt.imshow(predictions[i])
        plt.axis("off")
    plt.suptitle(f'Epoch {epoch}')
```

```
    plt.show()

# 7. TRAIN GAN

train(dataset, EPOCHS)
```