Part 1: Theoretical Understanding

1. Short Answer Questions

Q1: Explain the primary differences between TensorFlow and PyTorch. When would you choose one over the other?

TensorFlow and PyTorch are two of the most popular deep learning frameworks. They serve similar purposes but have key philosophical and practical differences that can influence your choice.

Feature	TensorFlow	PyTorch
Computation Style	Static computation graphs (TensorFlow 1.x), optional eager execution in TF 2.x	Dynamic computation graph (define-by-run)
Ease of Use	More complex historically (improved in TF 2.x + Keras)	Very intuitive and Pythonic — feels like NumPy
Deployment	Strong industry deployment tools (TF Serving, TF Lite, TF.js)	Improving deployment (TorchScript, TorchServe), but traditionally less mature
Performance	Highly optimized for production and distributed training	Great performance, strong Python ecosystem, excellent GPU utilization
Ecosystem	Large ecosystem for production: Keras, TF-Lite, TF-Serving, TF-JS	Very popular in research; supported by fastai, Hugging Face, etc.
Community Strength	Wider in industry + mobile/embedded apps	Stronger in research & academia
Debugging	Harder with static graph (much easier with TF2 eager mode)	Easy debugging thanks to dynamic graphs

Q2: Describe two use cases for Jupyter Notebooks in AI development.

Here are **two common use cases** for Jupyter Notebooks in AI development:

1. Experimentation and Prototyping Machine Learning Models

Jupyter Notebooks are widely used by data scientists and ML engineers to:

- Explore datasets interactively
- Build and test ML models step-by-step
- Visualize data and model outputs in real time
- Adjust hyperparameters and compare model performance

Example:
Developing and testing different neural network architectures on a dataset (e.g., training CNN models for image classification).

Data Analysis & Visualization

Before training models, Jupyter Notebooks are ideal for:

- Cleaning and preprocessing data
- Performing exploratory data analysis (EDA)
- Creating visualizations (matplotlib, seaborn, plotly)
- Identifying patterns or anomalies in datasets

Example:
Analyzing a customer dataset to find correlations, outliers, and distributions before building an AI-powered recommendation system.

Q3: How does spaCy enhance NLP tasks compared to basic Python string operations?

Capability	Basic Python Strings	spaCy
Text Understanding	Only handles raw characters & substrings	Performs tokenization, POS tagging, parsing, NER
Linguistic Features	None (no grammar or structure)	Understands syntax, grammar, sentence boundaries
Named Entity Recognition (NER)	Not possible	Recognizes names, dates, locations, orgs, etc.
Dependency Parsing	Not possible	Understands sentence structure & relationships
Sentence Segmentation	Manual, error-prone	Automatic & language-aware
Speed & Efficiency	Slow for large text	Optimized for large-scale NLP (Cython backend)
Machine Learning Integration	No	Has pretrained models & training pipelines

In Simple Terms

Basic Python string functions only help with **formatting and finding text patterns**, like:

text.lower(), split(), replace(), find()

Whereas **spaCy actually understands language**, allowing tasks like:

- Extracting people, places, dates
- Understanding grammatical roles (subject, verb, object)
- Lemmatization (turning "running" → "run")
- Recognizing sentence boundaries and context

Example

Basic Python can do:

"John lives in London".split()

Output:
['John', 'lives', 'in', 'London']

spaCy can do:

- “John” → Person
- “London” → Location
- Verb: “lives”
- Sentence subject: John

Why spaCy Matters

spaCy is designed for **real-world NLP applications**, such as:

- Chatbots & virtual assistants
- Information extraction
- Resume parsers & document analyzers
- Named entity extraction from business/legal text

2. Comparative Analysis

Compare Scikit-learn and TensorFlow in terms of:

- o Target applications (e.g., classical ML vs. deep learning).
- o Ease of use for beginners.
- o Community support.

Here’s a clear comparison of **Scikit-learn vs TensorFlow** across the requested points:

1. Target Applications

Category	Scikit-learn	TensorFlow
Primary Focus	**Classical Machine Learning**	**Deep Learning / Neural Networks**
Typical Algorithms	Decision trees, SVM, Random Forest, Logistic Regression, K-Means, etc.	Neural networks, CNNs, RNNs, Transformers, large-scale DL architectures
Best Use Cases	Tabular data, traditional statistical modeling, quick prototyping	Complex tasks like image recognition, NLP, speech processing, large-scale AI systems

---

2. Ease of Use for Beginners

Aspect	Scikit-learn	TensorFlow
Learning Curve	**Very beginner-friendly**	Steeper learning curve (improved with Keras)
Code Complexity	Simple APIs: .fit(), .predict()	More complex model building and training

Aspect	Scikit-learn	TensorFlow
Ideal For	New learners exploring ML fundamentals	Users stepping into deep learning and neural networks

3. Community Support

Category	Scikit-learn	TensorFlow
Community Size	Large academic & data science community	Very large global AI community (industry + research)
Backing	Open-source community	Backed by **Google**, widely used in industry
Resources	Many tutorials & docs	Massive number of resources, official courses, frameworks (TF Lite, TF Serving)

Part 2: Practical Implementation

Task 1: Classical ML with Scikit-learn

- **Dataset**: Iris Species Dataset
- **Goal**:
  1. Preprocess the data (handle missing values, encode labels).
  2. Train a **decision tree classifier** to predict iris species.
  3. Evaluate using accuracy, precision, and recall.
- **Deliverable**: Python script/Jupyter notebook with comments explaining each step.

Notes and quick summary:

- The notebook shows two options to load the dataset: a Kaggle-API download (commented) and a fallback using `sklearn.datasets` (which is identical to the Kaggle/UCI Iris data).
- It includes preprocessing (missing-value imputation example), label encoding, training a `DecisionTreeClassifier`, and evaluation with **accuracy**, **precision**, and **recall** (macro + per-class via `classification_report`).
- The Kaggle dataset page referenced: https://www.kaggle.com/uciml/iris

# Iris Species — Decision Tree Classifier

This Jupyter notebook shows how to:

1. Load the Iris dataset (option to download from Kaggle or use `sklearn` fallback),

2. Preprocess the data (handle missing values, encode labels),

3. Train a Decision Tree classifier, and

4. Evaluate using accuracy, precision, and recall.

The Kaggle dataset referenced: `https://www.kaggle.com/uciml/iris`.

```
# Imports
import os
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score, precision_score, recall_score, classification_report
from sklearn import datasets


# Show versions for reproducibility
import sklearn, sys
print("pandas:", pd.__version__)
print("numpy:", np.__version__)
print("sklearn:", sklearn.__version__)
print("python:", sys.version.splitlines()[0])
```

## 1) Loading the dataset

Two ways to load the data:

- **Option A (recommended if you have Kaggle CLI configured)**: download `iris.csv` from the Kaggle dataset page `uciml/iris` using the Kaggle API.

- **Option B (fallback)**: load the identical dataset directly from `sklearn.datasets` (no internet or Kaggle credentials needed).

If you want Option A, make sure you have `kaggle` installed and API credentials configured: https://www.kaggle.com/docs/api

```
# Option A: Load from Kaggle (commented out).
# To use this, uncomment and ensure kaggle CLI is installed and configured.
# !pip install kaggle
# from kaggle.api.kaggle_api_extended import KaggleApi
# api = KaggleApi()
# api.authenticate()
# # Download dataset to /tmp or working dir
# api.dataset_download_files('uciml/iris', path='.', unzip=True)
# df = pd.read_csv('Iris.csv')  # adjust filename if needed
# Option B: Load from sklearn (fallback, works without Kaggle)
iris = datasets.load_iris(as_frame=True)
df = iris.frame.copy()
# sklearn's frame includes target as numeric and target_names
df.head()
# Quick inspection
print('Shape:', df.shape)
print('\nColumns:', df.columns.tolist())
display(df.head())
display(df.tail())
display(df.describe())
## 2) Preprocessing
```

Steps:

1. Handle missing values (example uses `SimpleImputer` — Iris has no missing values but the code generalizes).

2. Encode labels (if labels are strings).

3. Split into train/test.

```
# Make a copy for preprocessing
data = df.copy()
# If dataset contains an 'Id' or similar column, drop it
for col in ['Id', 'id', 'ID']:
 if col in data.columns:
 data = data.drop(columns=[col])


# If target is named 'species' or 'target', handle accordingly.
# The sklearn frame has 'target' numeric and 'target_names' in iris, so we create a 'species'
column for demonstration.
if 'species' not in data.columns:
    # create species (string labels) for the demo
    data['species'] = data['target'].apply(lambda i: iris.target_names[i])


# Features and target
X = data.drop(columns=['target', 'species'], errors='ignore')
y = data['species']

print('Feature columns:', X.columns.tolist())
print('Target example values:', y.unique())

# 1) Handle missing values: numeric imputation

numeric_cols = X.select_dtypes(include=[np.number]).columns.tolist()

imputer = SimpleImputer(strategy='mean')
```

```
X[numeric_cols] = imputer.fit_transform(X[numeric_cols])


# If there were categorical features, you'd impute/fill them like:

# cat_cols = X.select_dtypes(exclude=[np.number]).columns.tolist()

# X[cat_cols] = X[cat_cols].fillna('missing')


print('Any missing values left in X?', X.isnull().any().any())

# 2) Encode the labels (target)

le = LabelEncoder()

y_encoded = le.fit_transform(y)

print('Classes:', le.classes_)

print('Encoded labels sample:', y_encoded[:10])

# Train-test split

X_train, X_test, y_train, y_test = train_test_split(

X, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded)


print('Train shape:', X_train.shape, 'Test shape:', X_test.shape)
```

3) Train a Decision Tree classifier

We use default hyperparameters for demonstration. In practice you'd tune with GridSearchCV or similar.

```
# Train

clf = DecisionTreeClassifier(random_state=42)

clf.fit(X_train, y_train)


# Feature importances and a quick sanity check

importances = clf.feature_importances_
```

```
for feat, imp in zip(X.columns, importances):

    print(f'{feat}: {imp:.4f}')
```

4) Evaluation

We compute accuracy, precision, recall. For multiclass precision/recall, we show macro and per-class values.

```
# Predictions

y_pred = clf.predict(X_test)



# Metrics

acc = accuracy_score(y_test, y_pred)

prec_macro = precision_score(y_test, y_pred, average='macro', zero_division=0)

rec_macro = recall_score(y_test, y_pred, average='macro', zero_division=0)


print(f'Accuracy: {acc:.4f}')

print(f'Precision (macro): {prec_macro:.4f}')

print(f'Recall (macro): {rec_macro:.4f}')

print('\nClassification report:')

print(classification_report(y_test, y_pred, target_names=le.classes_))

# Optional: save the trained model to disk

import joblib

os.makedirs('models', exist_ok=True)

joblib.dump(clf, 'models/decision_tree_iris.pkl')

print('Saved model to models/decision_tree_iris.pkl')
```

Conclusion

This notebook demonstrated loading the Iris dataset, simple preprocessing (including missing-value imputation and label encoding), training a Decision Tree classifier, and evaluating it using accuracy, precision, and recall. For production or better performance:

- Tune hyperparameters (GridSearchCV / RandomizedSearchCV).

- Use cross-validation for robust estimates.

- Consider simpler models or ensembling if needed.

Task 2: Deep Learning with TensorFlow/PyTorch

- **Dataset**: MNIST Handwritten Digits
- **Goal**:
    1. Build a **CNN model** to classify handwritten digits.
    2. Achieve >95% test accuracy.
    3. Visualize the model's predictions on 5 sample images.
- **Deliverable**: Code with model architecture, training loop, and evaluation.

MNIST CNN Classification — TensorFlow Keras

Install & Import

```
!pip install tensorflow tensorflow-datasets matplotlib
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
```

Load MNIST Dataset

```
(ds_train, ds_test), ds_info = tfds.load(
    "mnist",
    split=["train", "test"],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)
```

Preprocess Dataset

```
def normalize_img(image, label):
    return tf.cast(image, tf.float32) / 255.0, label

batch_size = 64

ds_train = (
    ds_train.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
    .cache()
    .shuffle(1000)
    .batch(batch_size)
    .prefetch(tf.data.AUTOTUNE)
)

ds_test = (
    ds_test.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(batch_size)
    .cache()
    .prefetch(tf.data.AUTOTUNE)
)
```

✓ Build CNN Model

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu',
input_shape=(28,28,1)),
    tf.keras.layers.MaxPooling2D(2,2),

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

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

model.summary()
```

Train Model

```
history = model.fit(
    ds_train,
    epochs=5,
    validation_data=ds_test
)
```

Evaluate

```
model.evaluate(ds_test)
```

Predict & Visualize 5 Sample Images

```
import numpy as np

sample_data = ds_test.take(1)

for images, labels in sample_data:
    preds = model.predict(images[:5])
    pred_labels = np.argmax(preds, axis=1)

    for i in range(5):
        plt.imshow(images[i].numpy().reshape(28,28), cmap="gray")
        plt.title(f"True: {labels[i].numpy()} | Pred: {pred_labels[i]}")
        plt.axis("off")
        plt.show()
```

Task 3: NLP with spaCy

- **Text Data**: User reviews from Amazon Product Reviews.
- **Goal**:
    1. Perform **named entity recognition (NER)** to extract product names and brands.
    2. Analyze sentiment (positive/negative) using a rule-based approach.
- **Deliverable**: Code snippet and output showing extracted entities and sentiment.

Python Code — NER + Sentiment on Amazon Reviews

```
# Install spaCy English model if not installed
# !python -m spacy download en_core_web_sm

import spacy
import pandas as pd

# Load spaCy NER model
nlp = spacy.load("en_core_web_sm")

# ---- Load Amazon Reviews Dataset ----
# After downloading from Kaggle, adjust the file path below.
df = pd.read_csv("amazonreviews.csv", nrows=5)  # reading first 5 to demo

# Example if dataset structure differs:
# df = pd.read_csv("train.ft.txt", sep="\t", names=["label", "text"])

# Let's assume the column name for text is "review"
texts = df.iloc[:, -1]  # last column is review text

# ---- Rule-based Sentiment Function ----
positive_keywords = ["good", "great", "excellent", "love", "perfect",
"amazing", "best"]
negative_keywords = ["bad", "terrible", "awful", "hate", "poor", "worst",
"broken"]

def rule_based_sentiment(text):
    text_lower = text.lower()
    score = 0
    score += sum(word in text_lower for word in positive_keywords)
    score -= sum(word in text_lower for word in negative_keywords)

    if score > 0:
        return "Positive"
    elif score < 0:
        return "Negative"
    return "Neutral"

# ---- Process Reviews ----
for review in texts:
    doc = nlp(review)

    # Extract product & brand-like entities (ORG, PRODUCT)
    entities = [(ent.text, ent.label_) for ent in doc.ents if ent.label_ in
["PRODUCT", "ORG"]]

    sentiment = rule_based_sentiment(review)

    print("Review:", review)
    print("Entities:", entities if entities else "None")
    print("Sentiment:", sentiment)
    print("-" * 60)
```

Part 3: Ethics & Optimization (10%)

1. Ethical Considerations

- Identify potential biases in your MNIST or Amazon Reviews model. How could tools like **TensorFlow Fairness Indicators** or **spaCy's rule-based systems** mitigate these biases?

Bias can emerge in both your **MNIST digit classifier** and the **Amazon Reviews NER + sentiment system**, but in different ways. Here's a clear breakdown of where bias comes from and how fairness tools can help.

Potential Biases in the Models

1. MNIST CNN Model (Handwritten Digit Classification)

Source of Bias	Explanation
Biased training data	MNIST was collected mostly from US schoolchildren & Census workers → handwriting style bias.
Unequal digit representation	Some digits appear more frequently than others, affecting accuracy.
Demographic handwriting differences	Age, cultural writing habits can affect shape & stroke style.
Model overfitting to specific stroke patterns	If someone writes a "1" as a serif stick (European style), accuracy may drop.

Example Impact

The model might misclassify:

- Digits written in stylized Asian handwriting
- Non-Western numeral variations
- Very neat vs. very messy writing patterns

---

2. Amazon Reviews NER + Sentiment System

Source of Bias	Explanation
Keyword-based sentiment bias	Rule-based keywords (good, bad, etc.) may miss nuance, irony, or cultural expressions
Brand recognition bias	spaCy NER favors well-known Western brands
Noise in user reviews	Slang, dialect, emojis → harder to classify fairly
Imbalanced product categories	If most reviews are about electronics, model works worse for beauty / clothing
False positives for "ORG" entities	SpaCy may mistakenly call anything capitalized a brand

Example Impact

- Review says: *"This laptop is sick!"* → slang conflict
  Rule-based model might incorrectly flag as negative
- SpaCy may misread:
  - *"Amazon Basics"* ✅
  - *"NaijaBrandHub"* ❌ (local brand missed)

How Fairness Tools Can Help

TensorFlow Fairness Indicators (for MNIST)

Fairness Indicators helps by:

Benefit	Explanation
Performance by subgroup	Evaluate accuracy by handwriting style groups (age, region if tagged data exists)
Highlighting disparity	Detect if Arabic-style 7 or looped 9 performs worse
Calibration checks	Ensures confidence scores aren't biased across subsets
Slice evaluation	Compare accuracy per digit, or writing thickness group

How to apply

- Create sub-datasets: e.g., *slanted vs upright writing*, *thin vs thick strokes*
- Run Fairness Indicators to see error rates per group
- Retrain or augment data to fix weak areas

---

spaCy Rule-Based Controls (for Amazon Reviews)

spaCy can use:

Tool	Mitigation
Custom NER patterns	Add patterns for local brands (“Infinix”, “Tecno”, “NaijaMarket”)
Entity Linking	Reduce false positives by linking to known brand DB
Rule + ML hybrid	Combine keyword rules + sentiment ML for better fairness
Context aware rules	Use dependency parse to avoid sarcasm traps

Example Fix for Sentiment Bias

Instead of only keyword lookup:

✓ Add context:
"not good" → negative
"sick phone 😍" → positive via emoji + POS tags

✓ Add slang list by region
(e.g., Nigerian slang sentiment dictionary)

✓ Use spaCy **PhraseMatcher** to detect African / Asian / local brands

2. Troubleshooting Challenge

- **Buggy Code**: A provided TensorFlow script has errors (e.g., dimension mismatches, incorrect loss functions). Debug and fix the code.

Step-by-Step Debugging Strategy

1 Check Input Shape

Common error:

```
ValueError: Input 0 is incompatible with layer ... expected shape=(None, 28,
28, 1) but got (None, 784)
```

Fix:
Match your data shape to the model input shape.

```
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

---

2 Verify Output Layer Matches # of Classes

For 10-class classification, final layer must be:

```
Dense(10, activation='softmax')
```

If mismatch, error:

```
logits and labels must be the same shape
```

---

3 Use Correct Loss Function

Task	Correct Loss
Binary classification	`binary_crossentropy`
Multi-class (one-hot labels)	`categorical_crossentropy`
Multi-class (int labels)	`sparse_categorical_crossentropy`

Example fix:

```
loss='sparse_categorical_crossentropy'
```

---

4 Print Tensor Shapes

Add debug prints:

```
print(x_train.shape, y_train.shape)
```

---

5 Simplify Model First

Start with a minimal working model, then increase complexity.

Buggy Code Example

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1)
])

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

model.fit(x_train, y_train, epochs=5)
```

Problems

Bug	Why
Input shape wrong	MNIST is (28,28), not (28,)
Output layer wrong	Output layer = 1 but 10 classes
Loss wrong	`categorical_crossentropy` expects one-hot vectors

Fixed Code

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')  # 10 classes
])

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

model.fit(x_train, y_train, epochs=5)
```