1. Short Answer Questions

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

Differences

Execution Paradigm:

TensorFlow Emphasized static computational graphs. You define the graph first, then run data through it. This was good for deployment but could be less flexible for research.

PyTorch is Known for its dynamic computational graphs built as operations are executed, offering more flexibility, easier debugging, and more intuitive control flow. TensorFlow has since adopted "Eager Execution" to mimic this.

Debugging

Dynamic graphs in PyTorch and TensorFlow Eager are generally easier to debug because you can inspect values at each step, similar to regular Python code.

Deployment

TensorFlow historically had a more mature ecosystem for deploying models to production, mobile, and edge devices. PyTorch has caught up significantly with tools like TorchServe.

Community & Adoption

Both have massive communities. PyTorch is often favored in research due to its flexibility, while TensorFlow is widely adopted in industry for large-scale deployments.

When to Choose One Over the Other

Choose PyTorch if you're doing cutting-edge research, rapid prototyping if you prefer easier debugging.

Choose TensorFlow if you're building large-scale production systems, deploying to various platforms (mobile, web, edge), or working within a Google-centric ecosystem. Modern TensorFlow with Keras and Eager Execution blurs these lines significantly.

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

Use Case 1

Exploratory Data Analysis (EDA) and Visualization:

Jupyter Notebooks are ideal for interactively loading, cleaning, transforming, and visualizing datasets. You can run small code cells to inspect data distributions, identify outliers, check for missing values, and create plots (e.g., histograms, scatter plots) directly within the notebook. This iterative process helps understand the data before model building.

Example: Loading a CSV into a Pandas DataFrame, checking df.info(), df.describe(), generating matplotlib plots to visualize relationships, and documenting findings with Markdown text.

Use Case 2

Model Prototyping and Iterative Experimentation

They allow data scientists and engineers to build and test machine learning models incrementally. Each cell can represent a step (e.g., data loading, feature engineering, model definition, training, evaluation). You can quickly modify a parameter, re-run a few cells, and observe the immediate impact on model performance without rerunning the entire script.

Example: Defining a neural network architecture in one cell, training it in another, and then immediately plotting the training loss and accuracy in a third, iterating on hyperparameters. This is perfect when testing code incrementally.

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

Basic Python String Operations Limitations

split() for tokenization might break words inappropriately (e.g., "don't" becomes "don", "t").
lower() for normalization is simple but doesn't handle complex linguistic variations (e.g., "USA" vs. "U.S.A.").
No inherent understanding of grammar, context, or semantic relationships.
How spaCy Enhances NLP

Linguistic Annotation
spaCy processes text into a Doc object, providing access to tokens with rich linguistic features (e.g., identifying "running" as a verb and its base form "run").
Named Entity Recognition (NER)
It can automatically identify and classify "named entities" like persons, organizations, locations, dates, product names, etc., which is impossible with basic string methods.

Efficiency and Performance
spaCy is designed for production-level speed and efficiency, especially with large text datasets, unlike custom string scripts that can be slow.

Pre-trained Models
It comes with highly optimized pre-trained models for various languages, allowing you to perform complex NLP tasks out-of-the-box without training your own models from scratch.

Vectorization and Word Embeddings
spaCy provides access to word vectors (if using larger models), enabling semantic understanding (e.g., knowing "king" is similar to "queen").

2. Comparative Analysis (Scikit-learn vs. TensorFlow)

Feature	Scikit-learn	TensorFlow
Target Applications	Classical Machine Learning algorithms: Classification (SVM, Decision Trees, Logistic Regression), Regression (Linear, Ridge), Clustering (K-Means, DBSCAN), Dimensionality Reduction (PCA), etc. Best for structured, tabular data.	Deep Learning: Neural networks (CNNs, RNNs, Transformers), Generative AI. Ideal for unstructured data like images, audio, text, and large-scale, complex patterns.
Ease of Use for Beginners	Very High: Simple, consistent API (.fit(), .predict(), .transform()). Minimal setup, easy to grasp core concepts. Excellent documentation and examples.	Moderate to High (with Keras): Keras API simplifies deep learning, but understanding underlying concepts (tensors, graphs, optimizers) can be challenging initially. More verbose for custom models.
Community Support	High: Mature library with extensive documentation, tutorials, and a very active	Very High: One of the most popular ML frameworks globally. Massive community,

	community. Widely used in academia and industry for traditional ML.	extensive documentation, Google support, countless tutorials, research papers, and pre-trained models.