Part 1: Theoretical Understanding (40%)

1. Short Answer Questions

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

Differences:

- Programming Paradigm: TensorFlow (up to version 1.x) primarily used a static computational graph, requiring graph definition before execution, while PyTorch uses a dynamic computational graph (eager execution), allowing for more intuitive, imperative-style coding. TensorFlow 2.x introduced eager execution, narrowing this gap.

- Ease of Debugging: PyTorch's dynamic nature makes debugging easier as code runs line-by-line, whereas TensorFlow's static graphs (in older versions) could complicate debugging.

- Deployment: TensorFlow has stronger support for production deployment, with tools like TensorFlow Serving, TensorFlow Lite for mobile/edge devices, and TensorFlow.js for web. PyTorch has improved with TorchServe and ONNX but lags slightly in deployment ecosystems.

- API Complexity: PyTorch offers a simpler, Pythonic API, preferred for research. TensorFlow's API is more comprehensive but can feel complex for beginners.

- Community and Ecosystem: TensorFlow has a broader enterprise adoption and ecosystem, while PyTorch dominates in academic research due to its flexibility.

When to Choose:

- TensorFlow: Choose for production-grade deployment, cross-platform applications (mobile, web, edge), or when working on large-scale distributed systems. Ideal for projects requiring robust tools like TensorFlow Extended (TFX) for end-to-end ML pipelines.

- PyTorch: Choose for research, prototyping, or when rapid experimentation and debugging are priorities. Preferred for dynamic models like those in NLP or when leveraging PyTorch's strong integration with libraries like Hugging Face.

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

1. Exploratory Data Analysis (EDA): Jupyter Notebooks allow data scientists to interactively load, visualize, and analyze datasets using libraries like Pandas, Matplotlib, and Seaborn. Inline visualizations and markdown cells enable iterative exploration and documentation, ideal for understanding data distributions or identifying outliers before model building.

2. Model Prototyping and Teaching: Notebooks are widely used to prototype ML models, test algorithms, and iterate on hyperparameters with libraries like Scikit-learn or PyTorch. Their cell-based structure supports step-by-step execution, making them perfect for tutorials, workshops, or sharing reproducible experiments with colleagues or students.

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

- Pre-trained Models: spaCy provides pre-trained, high-accuracy models for tasks like tokenization, part-of-speech (POS) tagging, named entity recognition (NER), and dependency parsing, which are far more sophisticated than manual string splitting or regex-based operations in Python.

- Efficiency: spaCy is optimized for performance, processing large texts quickly with a Cython-based implementation, unlike basic Python string operations, which can be slow and error-prone for complex NLP tasks.

- Linguistic Features: spaCy offers advanced linguistic annotations (e.g., lemmatization, sentence boundaries, word vectors) that go beyond simple string manipulation, enabling deeper text understanding without custom coding.

- Pipeline Customization: spaCy allows users to create custom NLP pipelines, integrating rule-based or ML components, whereas string operations require manual implementation of each feature, increasing development time and complexity.

2. **Comparative Analysis: Scikit-learn vs. TensorFlow**

- Scikit-learn: Primarily designed for classical machine learning tasks, such as regression, classification, clustering, and dimensionality reduction. It excels in applications like predictive modeling (e.g., customer churn prediction), feature engineering, and small-to-medium-scale datasets. It is not suited for deep learning or large-scale neural networks.

- TensorFlow: Focused on deep learning, supporting complex neural network architectures for tasks like image classification, NLP, and reinforcement learning. It also handles classical ML but is overkill for simple tasks. TensorFlow is ideal for large-scale, distributed training and production deployment across platforms (e.g., mobile, web).

Ease of Use for Beginners:

- Scikit-learn: Highly beginner-friendly with a consistent, high-level API that abstracts complex mathematics. Simple functions like `fit()` and `predict()` make it easy to implement models with minimal code. Extensive documentation and tutorials further lower the learning curve.

- TensorFlow: Less beginner-friendly due to its steeper learning curve, especially for deep learning concepts like defining layers, optimizers, and loss functions. TensorFlow 2.x's Keras API simplifies usage, but it still requires understanding of neural network architecture and debugging computational graphs, which can overwhelm novices.

Community Support:

- Scikit-learn: Has a strong, mature community with extensive documentation, tutorials, and Stack Overflow answers. Its focus on classical ML ensures stable, well-tested features, but the community is smaller than TensorFlow's due to its narrower scope. Contributions are active, with regular updates for new algorithms and improvements.

- TensorFlow: Boasts a massive, global community driven by Google's backing, with widespread adoption in industry and academia. It has abundant resources, including official guides, forums, GitHub issues, and third-party tutorials. The ecosystem is dynamic, with frequent updates, but its complexity can lead to fragmented support for niche issues. PyTorch's growing popularity has slightly reduced TensorFlow's research dominance.

Summary: Scikit-learn is ideal for beginners and classical ML tasks, offering simplicity and a strong community for traditional applications. TensorFlow is better for deep learning, large-scale systems, and production deployment, with a broader but more complex ecosystem.

1. Ethical Considerations

Potential Biases in MNIST and Amazon Reviews Models

MNIST Model:

- Bias Source: The MNIST dataset is well-balanced for digits 0–9, but if you train on a subset or a different handwriting dataset, you might have more samples of certain digits, or digits written in a style from a particular demographic (e.g., children vs. adults, different countries).

- Impact: The model may perform better on some digits or handwriting styles, and worse on others, leading to unfair predictions for underrepresented groups.

Amazon Reviews Model:

- Bias Source: Reviews may be skewed toward certain products, brands, or user demographics. Sentiment analysis may misclassify reviews with slang, sarcasm, or from non-native speakers.

- Impact: The model may unfairly rate products/brands or misinterpret sentiment for certain groups.

Mitigating Biases with Tools

TensorFlow Fairness Indicators:

- What it does: Provides metrics and visualizations to evaluate model performance across different subgroups (e.g., gender, age, region).

- How to use: After training your model, use Fairness Indicators to check if accuracy, precision, or recall is significantly lower for any subgroup. If so, you can:

  - Collect more data for underrepresented groups.

  - Rebalance your training data.

  - Adjust your model or loss function to penalize unfairness.

spaCy's Rule-Based Systems:

- What it does: Lets you define custom rules to supplement or override statistical models.

- How to use: For NER or sentiment, you can add rules to:

  - Recognize product/brand names that the model misses (e.g., new brands, slang).

  - Correct for known misclassifications (e.g., sarcasm, negations).

  - Ensure fairer treatment of language variations.

2. **Troubleshooting Challenge (Explanation Only)**

When working with TensorFlow (or any deep learning framework), common issues in classification scripts include:

1. Dimension Mismatches:

   - What happens: The model expects input data in a certain shape (e.g., images as `(batch, height, width, channels)`), but the data is provided in a different shape (e.g., missing the channel dimension).

   - How to fix: Always check the expected input shape for each layer. For grayscale images, add a channel dimension (e.g., reshape from `(28, 28)` to `(28, 28, 1)`).

2. Incorrect Loss Functions:

   - What happens: Using a regression loss (like Mean Squared Error) for a classification problem, or vice versa, leads to poor training or errors.

   - How to fix: For multi-class classification, use `categorical_crossentropy` (with one-hot labels) or `sparse_categorical_crossentropy` (with integer labels). For regression, use `mse` or `mae`.

3. Label Format Issues:

- What happens: The model expects labels in one format (e.g., one-hot encoded), but receives them in another (e.g., integers).

- How to fix: Ensure your labels match the loss function. Use one-hot encoding for `categorical_crossentropy`, or integer labels for `sparse_categorical_crossentropy`.

4. Output Layer Activation:

- What happens: The output layer lacks the correct activation function (e.g., missing `softmax` for classification), so outputs are not valid probabilities.

- How to fix: For multi-class classification, use `softmax` activation in the output layer.