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?

Feature	**TensorFlow (TF)**	**PyTorch (PT)**
Computational Graph	**Static** (defined before execution)	**Dynamic** (defined during execution)
Debugging	More difficult due to static graph (though this has changed with Eager Execution)	Easier, as it integrates well with standard Python debugging tools
Deployment	Excellent support for production deployment (TF Serving, TF Lite)	Improved, but traditionally less robust than TF (e.g., TorchServe)
Adoption	Industry (Google, large enterprises) and Production	Research and Rapid Prototyping
API	High-level (Keras) and Low-level (Eager Execution)	More Pythonic and object-oriented (similar to NumPy)

When to Choose Which:

- **Choose TensorFlow when:**
  - The primary goal is **production deployment** at scale (web, mobile, embedded devices).
  - You need **mature visualization tools** like TensorBoard for monitoring.
  - You prefer a **ready-made high-level API** (Keras) for fast iteration on standard models.
- **Choose PyTorch when:**
  - The goal is **cutting-edge research** or rapid prototyping of complex, non-standard models.
  - You need to use **dynamic graph structures** (like in certain RNNs, sequence models, or tree-based models).
  - You value a more **Pythonic programming experience** and easier debugging.

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

1. **Exploratory Data Analysis (EDA) and Visualization:** Jupyter Notebooks allow AI engineers to load a dataset, perform statistical analysis, and visualize data

distributions **interactively** (using libraries like Pandas, Matplotlib, and Seaborn). This is crucial for understanding data quality, identifying outliers, and choosing the right features and models before starting deep learning.

2. **Rapid Prototyping and Model Experimentation:** They enable engineers to define a model, train it on a small subset of data, and **immediately see the results** (metrics, predictions, and even visualizations of the model architecture) in the same document. This cell-by-cell execution greatly speeds up the iterative process of testing different hyperparameter configurations and model architectures.

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

spaCy provides a robust, pre-trained, and highly optimized framework that performs **linguistic-aware processing**, far surpassing simple string operations (like `str.split()`, `str.replace()`, or regular expressions).

- **Linguistic Context:** Basic string operations treat text as just characters. spaCy understands text in a **linguistic context**, providing functionalities like:
  - **Tokenization:** Intelligent splitting into words, punctuation, and even contractions (e.g., "don't" → "do" and "n't").
  - **Part-of-Speech (POS) Tagging:** Assigning grammatical labels (e.g., 'NOUN', 'VERB') to each token.
  - **Named Entity Recognition (NER):** Identifying real-world objects like people, organizations, and dates.
- **Efficiency:** spaCy is written in Cython, making its pipeline operations significantly **faster and more memory-efficient** for processing large volumes of text data compared to writing equivalent logic using pure Python and regex.

2. Comparative Analysis: Scikit-learn vs. TensorFlow

Feature	**Scikit-learn (sklearn)**	**TensorFlow (TF)**
Target Applications	**Classical Machine Learning (ML):** Classification (e.g., SVM, Naive Bayes), Regression, Clustering, Dimensionality Reduction. **Focus on tabular data.**	**Deep Learning (DL):** Complex tasks like Image/Video Processing (CNNs), Natural Language Processing (RNNs/Transformers), and Generative Modeling. **Focus on unstructured data.**
Ease of Use for Beginners	**Extremely Easy:** Simple, consistent API for all models (`.fit()`, `.predict()`). Minimal boilerplate code. Excellent for fast onboarding.	**Moderate to Harder:** Requires understanding of tensors, graph definition (even with Keras), and low-level DL concepts (layers, activation functions, optimizers).

Community Support	**Mature and Stable:** Very active, high-quality documentation, considered the standard for classical ML in Python.	**Massive and Rapidly Evolving:** Backed by Google, immense resources, frequent updates, and a vast community for troubleshooting complex DL problems.

Part 3: Ethics & Optimization (10%)

1. Ethical Considerations

A. Potential Biases in Your Models

Model	**Potential Biases**	**Explanation**
MNIST CNN (Image Classification)	**Representational Bias (Handwriting Style)**	The MNIST dataset is comprised primarily of digits written by employees of the US Census Bureau and high school students. The model may perform worse on digits written by **non-dominant demographic groups** (e.g., different nationalities, older adults, or people with certain disabilities) whose handwriting styles were underrepresented in the original training data.
Amazon Reviews (Sentiment/NER)	**Selection/Toxicity Bias (Sentiment Analysis)**	Reviews are often dominated by a small percentage of highly engaged users or "haters/fans" leading to **skewed sentiment polarity**. The rule-based model may also inadvertently assign negative sentiment to reviews that use common terms for a **minority product or brand** (if those terms were used negatively in majority reviews), or if it fails to recognize sentiment expressed in **non-standard dialects or slang**.

B. Mitigation Strategies Using AI Tools

1. Mitigation using TensorFlow Fairness Indicators (for MNIST)

Since MNIST is a simple image dataset, you would conceptually apply the mitigation to a more complex image task (e.g., face recognition). Assuming the MNIST data had **sensitive attributes** (like writer's age or gender, if available), TensorFlow Fairness Indicators (TFFI) would be used as follows:

- **Detection:** TFFI allows you to compute metrics (like **False Positive Rate** and **Accuracy**) sliced by different groups (e.g., 'young writers' vs. 'old writers').
- **Analysis:** If TFFI shows a significantly higher **False Negative Rate** (failing to correctly classify a digit) for the 'old writers' slice compared to the 'young writers' baseline, this diagnoses the bias.
- **Mitigation:** You could then use this diagnosis to apply mitigation techniques such as **resampling** (increasing the weight or number of 'old writers' samples) or employing fairness-aware loss functions to retrain the model.

2. Mitigation using spaCy's Rule-Based Systems (for Amazon Reviews)

Rule-based systems offer direct, transparent control over linguistic biases, unlike black-box deep learning models.

- **Explicit Bias Control:** If the rule-based sentiment model shows bias against a minority brand because it flagged a specific product feature term as negative, you can **directly modify the lexicon** (the list of positive/negative words).
- **Custom Tokenization/Matching:** You can use spaCy's **Matcher** or **custom components** in the pipeline to define specific, non-biased rules. For instance:
    - Create a rule to *always* recognize a specific product name as an **ORG/PRODUCT** entity regardless of its surrounding context (mitigating NER bias).
    - Implement **contextual sentiment rules** where a negative word is ignored if it's found within a specific linguistic structure (e.g., "not a bad product" is treated as neutral or positive). This adds transparency and precision that statistical models often lack.

---

2. Troubleshooting Challenge (Buggy Code)

The final deliverable requires your group to **debug and fix a provided TensorFlow script**. This tests your practical understanding of DL fundamentals.

Common TensorFlow Errors to Look For (and Fix):

Error Type	**Description & Symptom**	**Typical Fix**

Dimension Mismatches	Error on a Keras layer (e.g., Input shape (N, N, C) is incompatible with Flatten). Happens when layers expect a specific shape (e.g., 2D array) but receive another (e.g., 3D tensor).	**Explicit Reshaping:** Use tf.expand_dims() to add a channel dimension, or ensure all layers (especially Flatten after Conv2D/MaxPool2D) are compatible with the output of the preceding layer.
Incorrect Loss Function	Model compiles but training loss doesn't decrease, or output is gibberish. Occurs when loss function doesn't match the output layer/encoding.	**Matching Loss and Labels:** For multi-class classification (like MNIST): use CategoricalCrossentropy with **one-hot encoded** labels, OR use SparseCategoricalCrossentropy with **integer encoded** labels. Ensure the output layer has the correct number of units (10 for MNIST) and softmax activation.
Data Type Errors	Model crashes or throws an error about converting an array to a tensor (e.g., expected type float64, got int32).	**Type Casting:** Ensure all inputs ($\mathbf{X}$) are floating-point (.astype('float32')) and labels ($\mathbf{y}$) are integers (for sparse loss) or floats (for one-hot loss).