AI & Machine Learning Skills Assessment.
Part 1: Short Answer Questions.

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

Answer:

TensorFlow and PyTorch are the two most dominant open-source deep learning frameworks. While they both serve the same purpose – building and training neural networks – they have distinct philosophies and strengths.

Here's a breakdown of their primary differences:

1. Computation Graph:
   - Dynamic vs. Static (Historical Difference) PyTorch (Dynamic Graph - "Define-by-Run"): PyTorch's computational graph is built dynamically as operations are executed. This means you can define, change, and debug your network on the fly, much like regular Python code. Pros: Easier debugging, more flexible for research and rapid prototyping, better for models with dynamic architectures (e.g., RNNs).

   - TensorFlow (Static Graph - "Define-and-Run" in TF 1.x, now mostly Dynamic in TF 2.x): In TensorFlow 1.x, you first defined the entire computational graph and then executed it. TensorFlow 2.0 introduced "Eager Execution" by default, which functions very similarly to PyTorch's dynamic graph. However, it still retains the ability to convert models into static graphs for optimized deployment. Pros (of static graphs): More opportunities for graph-level optimizations, easier deployment to various platforms (mobile, embedded devices).

2. Ease of Use and "Pythonic" Feel :
   - PyTorch: Often lauded for its more "Pythonic" and intuitive API. It integrates seamlessly with Python's ecosystem and feels familiar to NumPy users. TensorFlow: While TensorFlow 2.0 has made significant strides in improving its API (largely through Keras integration), it historically had a steeper learning curve.

3. Deployment and Production Readiness :
   - TensorFlow: Historically has a strong advantage in production. It offers a comprehensive ecosystem including TensorFlow Serving, TensorFlow Lite (for

mobile/embedded), TensorFlow.js (for web), and TensorFlow Extended (TFX) for end to-end ML pipelines.

- PyTorch: Has significantly improved its deployment story with tools like TorchServe and support for the ONNX format. It is catching up but is sometimes considered less mature for large-scale production than TensorFlow's ecosystem.

When to Choose One Over the Other:

Choose PyTorch if:

- You are in research or academia, prioritizing rapid prototyping and experimentation.
- You prioritize flexibility, easy debugging, and a more "Pythonic" feel. You are working with dynamic neural network architectures (e.g., RNNs with

variable sequence lengths).

Choose TensorFlow if:

- You are deploying models to production at scale, especially across multiple platforms (server, mobile, web).
- You need to leverage Google's TPUs for large-scale training. You are working in an enterprise environment where its mature and extensive deployment ecosystem is a key advantage.

.

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

1. Exploratory Data Analysis (EDA) and Preprocessing:
    - Jupyter Notebooks are ideal for the initial phase of any AI project. Their interactive, cell-based structure allows data scientists to load data (e.g., using Pandas), inspect it, and visualize it (using Matplotlib or Seaborn) in an iterative fashion. Each step—like loading data, checking for missing values, generating summary statistics, and plotting distributions—can be executed in a separate cell. The output is displayed inline, providing immediate feedback. This makes the process of understanding data, cleaning it, and engineering new features transparent, reproducible, and easy to share.
2. Model Prototyping, Experimentation, and Evaluation:
    - Notebooks provide an excellent environment for rapidly building and testing different models. A developer can define a model architecture (using Scikit-learn, TensorFlow, or PyTorch), train it on the data, and immediately evaluate its performance using metrics and visualizations, all within the same document. It's highly efficient for

experimenting with different hyperparameters (like learning rates or model complexity) because one can simply change a value in a cell and re run the subsequent training and evaluation cells. This iterative loop of "tweak, train, evaluate" is central to model development and is greatly streamlined by the notebook interface.

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

- spaCy enhances NLP tasks by providing linguistic understanding and a pipeline of pre-trained models, moving far beyond the simple character manipulation of basic Python strings.

1. Linguistic Understanding vs. Character Manipulation:
    - Basic Strings: Operations like `.split()`, `.lower()`, and `replace()` treat text as a sequence of characters. They have no knowledge of grammar, word meanings, or sentence structure.
    - spaCy: When spaCy processes text, it creates a `Doc` object where each token is enriched with linguistic annotations.

   This includes:
    - Tokenization: Intelligently splits text into words and punctuation,handling contractions like "don't" to "do", "n't".
    - Lemmatization: Finds the base form of a word (e.g., "running" -> "run").
    - Part-of-Speech (POS) Tagging: Identifies a word's grammatical role (noun, verb, adjective).
    - Dependency Parsing: Understands the grammatical relationships between words.
2. Built-in NLP Capabilities:
    - Basic Strings: To perform a task like Named Entity Recognition (NER), one would have to write complex and brittle rules using regular expressions.
    - spaCy: Comes with pre-trained statistical models that can perform complex tasks like NER out-of-the-box, accurately identifying people, organizations, and products with minimal code. It also includes tools for sentence segmentation, text classification, and more.
3. Efficiency and Scalability:
    - SpaCy is written in Cython, making it extremely fast and memory-efficient. It is designed for production use and can process large volumes of text much more quickly than custom Python loops and string operations.

Part 2: Compare Scikit-learn and TensorFlow in terms of target applications, ease of use for beginners, and community support.

Answer:

- Target Applications:
    - Scikit-learn: Focuses on classical machine learning. It is best for structured (tabular) data and tasks like classification (e.g., Logistic Regression, Random Forest), regression, and clustering (e.g., K-Means).
    - TensorFlow: Focuses on deep learning. It is primarily used for building and training complex neural networks for tasks with unstructured data like images (computer vision) and text (NLP).
- Ease of Use for Beginners:
    - Scikit-learn: Considered very high and beginner-friendly. It provides a simple, consistent API (.fit(), .predict()) that makes building models straightforward.
    - TensorFlow: Has a moderate learning curve. While its Keras API has made it much simpler, it can still be complex for non-trivial tasks that require understanding concepts like tensors and data pipelines.
- Community Support:
    - Scikit-learn: Has a strong and mature community. It is particularly praised for its excellent, clear, and comprehensive documentation.
    - TensorFlow: Has a massive and global community, backed by Google. This results in an extensive ecosystem of official tutorials, user groups, and resources driven by wide industry adoption.

Part 3: Ethics & Optimization

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?

Answer:

Potential Biases:

- MNIST Model (Dataset Bias): The model could be biased against handwriting styles that are underrepresented in the MNIST dataset. For example, it might perform worse on digits written by individuals from different cultural backgrounds (e.g., a '7' with a crossbar), people with motor disabilities, or left-handed writers, if these styles were not common in the training data.

- Amazon Reviews Model (Social & Language Bias): Sarcasm: The simple rule-based sentiment model will fail on sarcastic reviews (e.g., "The battery life is 'fantastic', it lasts a whole 2 hours."). It will incorrectly classify this as positive.

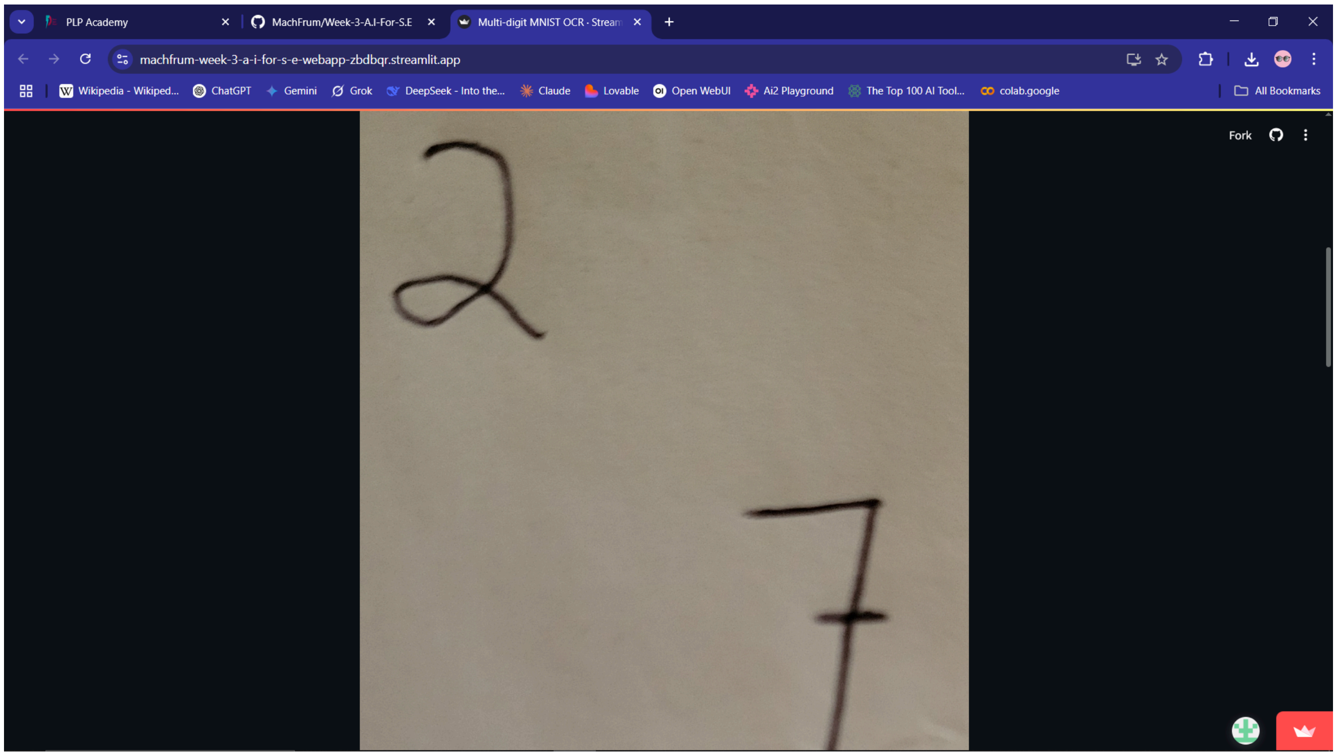
- Demographic Bias: The language used in positive/negative reviews may differ across demographic groups. If our keyword lists are based on language from one dominant group, the model will be less accurate for others. Product Bias: The NER model may be better at identifying well-known brands (like "Apple") than smaller, emerging ones, simply due to their prevalence in the training data.

Mitigation Strategies:

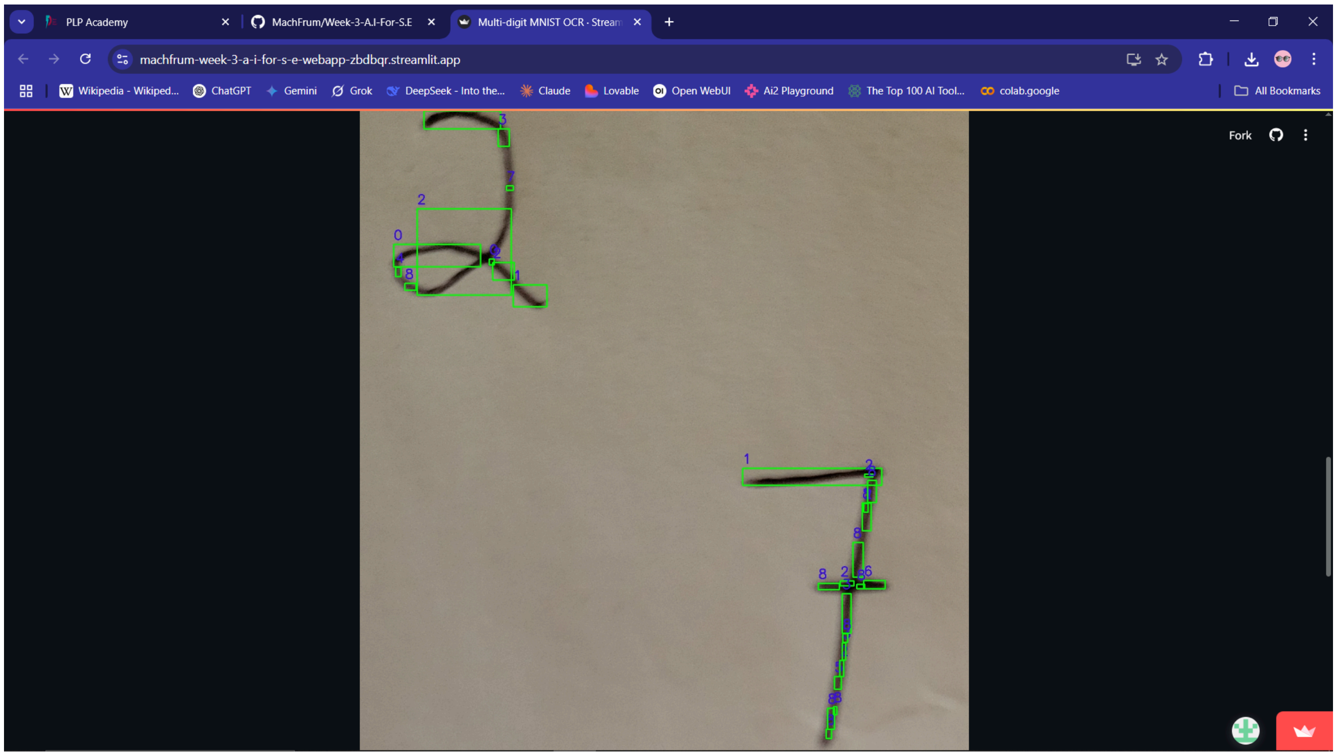
- TensorFlow Fairness Indicators (TFFI): For the MNIST model, if we had metadata for the images (e.g., `writing_style_group`), we could use TFFI to audit for bias. TFFI would compute evaluation metrics (like accuracy) for each subgroup. If it revealed that accuracy for "Style A" was 99% but only 85% for "Style B," it would prove a bias exists. This knowledge would then prompt us to mitigate it by finding more data for "Style B" (data augmentation or targeted collection). TFFI doesn't fix bias, but it is essential for finding and quantifying it.
- spaCy's Rule-Based Systems: For the Amazon Reviews model, spaCy's `Matcher` is perfect for direct mitigation. We can write specific rules to override the statistical model or our simple keyword list. To handle sarcasm, we could create a pattern that looks for a positive keyword in quotation marks or near a phrase with a negative connotation (e.g., `"fantastic" + ... + "lasts 2 hours"`). This allows us to inject human-level understanding to correct known failures and biases in the system, making the model fairer and more accurate.

MODEL PERFORMANCE:

Here's the handwritten text we uploaded.

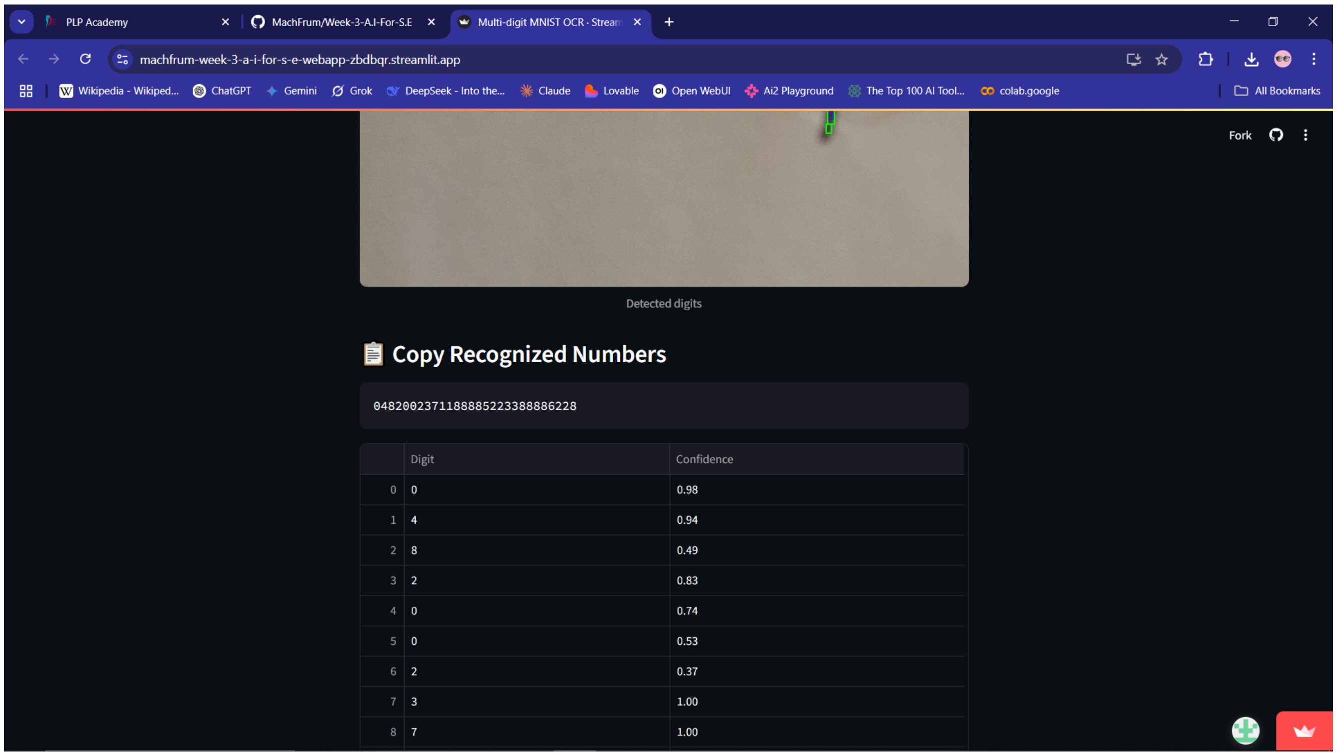


Here’s how the image got processed by the model:



Since the model is trained on the MNIST dataset, which is very focused and prone to overfitting, it primarily identifies small digits and lacks perspective. However, when examining the specific section where the model is highly focused, it correctly predicts the digit, even though in reality, it's a part of a stroke of a larger number.

Here are the outcome:



Detected digits

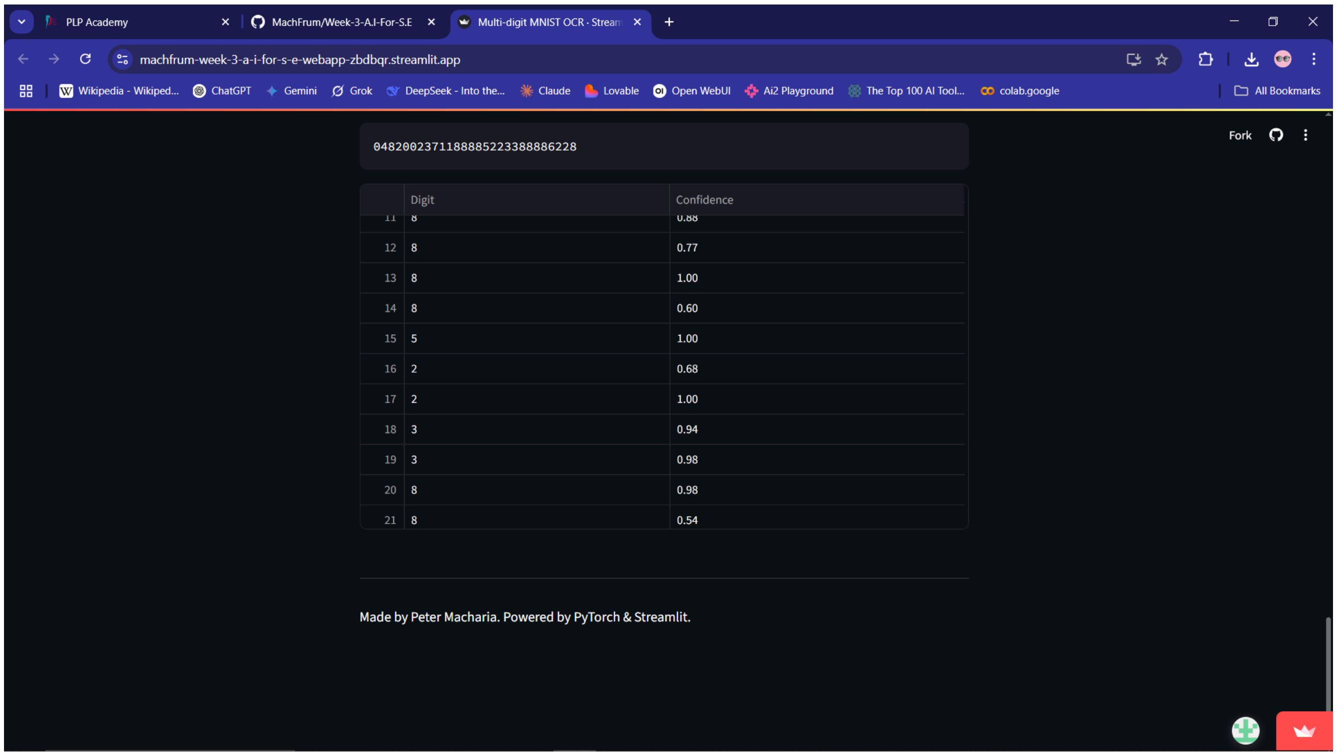
📋 Copy Recognized Numbers

```
04820023711888852233888886228
```

	Digit	Confidence
0	0	0.98
1	4	0.94
2	8	0.49
3	2	0.83
4	0	0.74
5	0	0.53
6	2	0.37
7	3	1.00
8	7	1.00

The recognized number section displays all the digits the model identified, which isn't entirely accurate. However, as mentioned, if you examine the specific subsection the model focused on, the predicted digit is often correct—or, if the digit is incorrect, it's usually because the model assigned it a low confidence score because it had a hard time.

Here's another one:



```
04820023711888852233888886228
```

	Digit	Confidence
11	8	0.88
12	8	0.77
13	8	1.00
14	8	0.60
15	5	1.00
16	2	0.68
17	2	1.00
18	3	0.94
19	3	0.98
20	8	0.98
21	8	0.54

Made by Peter Macharia. Powered by PyTorch & Streamlit.

CONCLUSION:

The dataset was the main limitation. A dataset containing real-world handwritten material would be ideal. We've identified the Street View House Numbers (SVHN) dataset, along with a few others, as suitable alternatives. This marks the next stage in the project's evolution. Our goal is to extract digits and letters from user-uploaded images and convert them into copyable text.