AI IN SOFTWARE ENGINEERING - WEEK 3 ASSIGNMENT

BY

FALAYE IFEOLUWA DAVID

PART 1: THEORECTICAL QUESTIONS

PRIMARY DIFFERENCES BETWEEN TENSORFLOW AND PYTORCH

TensorFlow uses static computational graphs (though TensorFlow 2.x has eager execution by default), which means the graph structure is defined first and then executed. It offers excellent production deployment tools through TensorFlow Serving, TensorFlow Lite for mobile, and TensorFlow.js for web applications. The framework provides a more comprehensive ecosystem for end-to-end ML pipelines.

PyTorch uses dynamic computational graphs (define-by-run), making it more intuitive and Pythonic. The graph is built on-the-fly during execution, which makes debugging significantly easier. It's become the preferred choice in academic research due to its flexibility and ease of experimentation.

When to choose TensorFlow:

- Production deployment at scale is a priority
- You need mobile or web deployment
- Working with Google Cloud infrastructure
- Require robust MLOps tools and serving infrastructure
- Building production pipelines with TensorFlow Extended (TFX)

When to choose PyTorch:

- Research and prototyping projects
- Need intuitive debugging capabilities
- Prefer a more Pythonic, dynamic approach
- Working on cutting-edge research (most recent papers use PyTorch)
- Building custom architectures that require flexibility

TWO USE CASES FOR JUPYTER NOTEBOOKS IN AI DEVELOPMENT

Use Case 1: Exploratory Data Analysis (EDA) and Data Visualization

Jupyter Notebooks excel at interactive data exploration. You can load datasets, visualize distributions, identify patterns, and detect anomalies all in one place. The inline visualization capabilities allow data scientists to immediately see the results of their analysis alongside the code. For example, when preparing data for a machine learning model, you can quickly iterate through different preprocessing techniques, visualize feature correlations using heatmaps, and document insights directly in markdown cells.

Use Case 2: Model Experimentation and Prototyping

Notebooks provide an ideal environment for training and comparing different models iteratively. You can train a baseline model, visualize training curves, adjust hyperparameters, and immediately see the results without re-running entire scripts. This is particularly valuable when experimenting with neural network architectures or comparing multiple algorithms. The ability to save intermediate results and selectively re-run cells makes the experimentation process efficient and well-documented.

HOW SPACY'S ENHANCES NLP TASKS

spaCy is an industrial-strength NLP library that provides significant advantages over basic Python string operations:

Pre-trained Models: spaCy comes with pre-trained statistical models for tokenization, part-of-speech tagging, named entity recognition, and dependency parsing. Basic string operations would require you to manually implement complex linguistic rules.

Linguistic Annotations: It automatically provides rich linguistic annotations including lemmatization, morphological analysis, and syntactic dependencies. For example, doc.ents immediately gives you named entities, while string operations would require complex regex patterns and manual entity recognition.

Efficient Processing: spaCy is optimized for production use with Cython, making it significantly faster than pure Python implementations. It processes text in a pipeline architecture that's both efficient and extensible.

Semantic Understanding: spaCy includes word vectors and similarity calculations, enabling semantic operations beyond surface-level string matching. You can compute document similarity or find semantically related terms, which is impossible with basic string operations.

COMPARATIVE ANALYSIS: SCIKIT-LEARN VS TENSORFLOW

Target Applications

Scikit-learn:

- **Focus:** Classical machine learning algorithms
- **Best for:** Tabular data, structured datasets, traditional ML tasks
- **Algorithms:** Linear/logistic regression, decision trees, random forests, SVMs, k-means clustering, dimensionality reduction (PCA, t-SNE)
- **Use cases:** Classification on structured data, regression problems, clustering customer segments, feature engineering pipelines
- **Not suitable for:** Deep neural networks, computer vision at scale, large-scale sequence modeling

TensorFlow:

- **Focus:** Deep learning and neural networks
- **Best for:** Unstructured data (images, text, audio), complex pattern recognition
- **Algorithms:** Convolutional neural networks (CNNs), recurrent neural networks (RNNs), transformers, reinforcement learning
- **Use cases:** Image classification, object detection, natural language processing, speech recognition, generative models

- **Scalability:** Designed for distributed training across multiple GPUs/TPUs

Ease of Use for Beginners

Scikit-learn (More Beginner-Friendly):

Scikit-learn follows a consistent, simple API design. Most models use the same .fit(), .predict(), and .score() methods, making it extremely intuitive. A beginner can train a model in just 3-4 lines of code without understanding complex mathematical concepts. The documentation is excellent with clear examples for every algorithm.

Example simplicity:

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

The abstractions are high-level, hiding complexity while still being powerful. Error messages are generally clear and helpful.

TensorFlow (Steeper Learning Curve):

TensorFlow requires understanding of neural network architectures, layers, optimization, and training loops. Even with Keras integration (which simplified things considerably), beginners need to grasp concepts like loss functions, activation functions, backpropagation, and gradient descent.

TensorFlow 2.x with Keras has improved accessibility significantly, but still requires more foundational knowledge. Debugging can be challenging when models don't converge or produce unexpected results. The framework requires more setup for data preprocessing, model architecture design, and training configuration.

Winner for beginners: Scikit-learn, by a significant margin.

Community Support

Scikit-learn:

- **Maturity:** Established since 2007, extremely stable and well-documented
- **Community:** Large user base across industry and academia
- **Resources:** Comprehensive official documentation, extensive tutorials, Stack Overflow has thousands of answered questions
- **Maintenance:** Actively maintained by core developers with consistent updates
- **Best for:** Finding solutions to classical ML problems, well-trodden paths have abundant examples

TensorFlow:

- **Backing:** Google's support provides extensive resources and continuous development
- **Scale:** One of the largest ML communities globally

- **Resources:** Official tutorials, TensorFlow Hub for pre-trained models, extensive YouTube content, Coursera specializations
- **Ecosystem:** Broader ecosystem including TensorBoard for visualization, TF Extended for production, model deployment tools
- **Research:** Strong presence in cutting-edge research and industry applications
- **Best for:** Deep learning problems, production deployment, access to state-of-the-art models

Some other Comparison are:

- **Documentation quality:** Both excellent, Scikit-learn slightly more beginner-friendly
- **Stack Overflow presence:** TensorFlow has more questions (indicating both popularity and complexity), Scikit-learn has higher answer rates
- **Updates:** TensorFlow has more frequent major updates (can be disruptive), Scikit-learn is more stable
- **Job market:** TensorFlow skills are more in-demand for deep learning roles, Scikit-learn essential for data science positions

In Summary, TensorFlow has a larger and more active community due to deep learning's prominence, but Scikit-learn's community is more accessible and beginner-friendly with better question-to-answer ratios.

PART 2: SCREENSHOTS OF MODEL OUTPUTS

Classical ML with Scikit-learn (Iris Dataset)

```
--- Classical ML with Scikit-learn (Iris) ---
Dataset loaded: 150 samples, 4 features.
No missing values found.
Train set size: 105, Test set size: 45

Training Decision Tree Classifier...
Training complete.

Model Accuracy: 0.9333

--- Classification Report ---
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      0.80      0.89        15
   virginica       0.83      1.00      0.91        15

    accuracy                           0.93        45
   macro avg       0.94      0.93      0.93        45
weighted avg       0.94      0.93      0.93        45
```

Deep Learning with TensorFlow/Keras (MNIST)

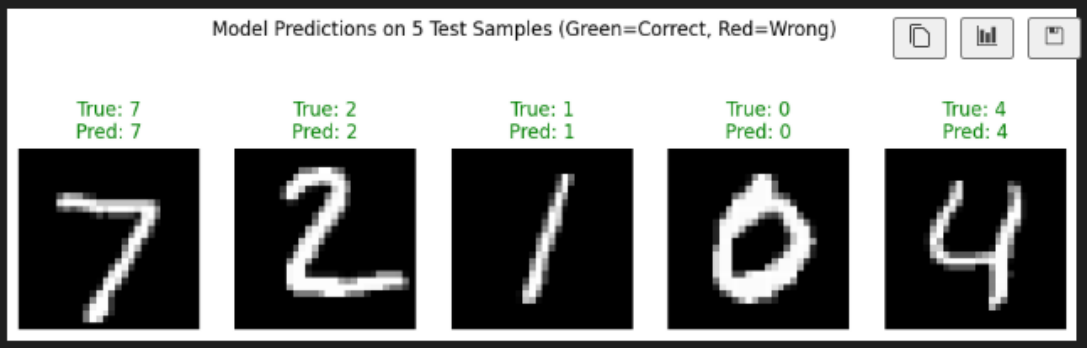
```
 Total params: 34,826 (136.04 KB)


 Trainable params: 34,826 (136.04 KB)


 Non-trainable params: 0 (0.00 B)


Training the model (this may take a few minutes)...
Epoch 1/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 50s 27ms/step - accuracy: 0.9486 - loss:
Epoch 2/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 36s 21ms/step - accuracy: 0.9824 - loss:
Epoch 3/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 37s 22ms/step - accuracy: 0.9874 - loss:
Epoch 4/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 31s 18ms/step - accuracy: 0.9898 - loss:
Epoch 5/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 28s 16ms/step - accuracy: 0.9919 - loss:
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.savi
Training complete.
Trained model saved as 'mnist_cnn_model.h5'

Test Accuracy: 98.96% (Goal > 95%)
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 198ms/step
```



NLP with spaCy (NER and Sentiment Analysis)

```
--- NLP with spaCy (NER and Sentiment) ---
spaCy model 'en_core_web_sm' loaded successfully.

--- Named Entity Recognition (NER) ---

Extracted Product Names:

Extracted Brand/Company Names:
- Apple

--- Rule-Based Sentiment Analysis ---
Positive keyword count: 4
Negative keyword count: 2

Overall Review Sentiment (Rule-Based): **Positive**
```

PART 3: ETHICAL REFLECTION

ETHICAL CONSIDERATIONS

Let us examine potential biases in the two models developed and how TensorFlow Fairness Indicators or spaCy's rule-based systems can help in mitigation.

Potential Biases:

Model	Potential Bias	How it Arises
MNIST CNN (Task 2)	Style/Aesthetic Bias	The dataset predominantly consists of digits written in a specific style (often collected from US census workers). If the model is deployed in a region with very different handwriting conventions (e.g., specific European, Asian, or Arabic numerals), the performance will drop dramatically for those styles, leading to unfair results based on geographic style.
Amazon Reviews NLP (Task 3)	Demographic/Source Bias	The sentiment analysis is based on simple keywords. Reviews often use slang, regional idioms, or code-switching (mixing languages). The simple rule-based system will fail to correctly classify sentiment in reviews written by specific non-standard or regional user groups, leading to biased sentiment scores that don't reflect the true intent of those speakers.

Mitigation Strategies:

1. MNIST Mitigation with TensorFlow Fairness Indicators

TensorFlow Fairness Indicators is a suite of tools that allows for the computation of common fairness metrics (like True Positive Rate, False Negative Rate, etc.) across different demographic groups.

Strategy	Action
Data Grouping	You would first need to create a "style" or "geographic origin" feature for your MNIST dataset (if possible) or create a specialized test set.
Metric Slicing	Use the tool to compute metrics like "Accuracy on US styles" vs. "Accuracy on European styles".
Mitigation	If a disparity is found (e.g., low recall for one style), you could re-weight the minority-style data samples or apply adversarial debiasing techniques during training.

2. Amazon Reviews Mitigation with spaCy's Rule-Based Systems

While the problem addressed here was a simple rule-based sentiment system, a sophisticated spaCy pipeline can be used to improve fairness by increasing linguistic coverage.

Strategy	Action
Custom Tokenization/Lemmatization	Extend the spaCy pipeline to include custom rules for handling slang or text abbreviations (e.g., mapping "luv it" to "love it").
Named Entity (NER) Filtering	Bias often arises when a model *over-relies* on specific named entities (e.g., always associating a brand name with positive words). You could use spaCy's NER output to exclude entities from simple sentiment scoring, forcing the sentiment to be based only on descriptive terms.
Contextual Rules	Develop more sophisticated rules that analyze the dependency tree (a feature of spaCy) to determine negation (e.g., "The feature was not great"). This prevents misclassification that disproportionately affects users who use complex sentence structures.

TROUBLESHOOTING CHALLENGE (DEBUGGING TENSORFLOW)

The two most common TensorFlow/Keras errors are dimension mismatches and incorrect loss functions.

1. Debugging Dimension Mismatches (Shapes)

Dimension mismatches often occur when transitioning between different layer types (e.g., from a CNN layer to a Dense layer) or when preparing input data.

Error Type	Solution/Strategy
Input Shape Error	Check the first layer: Ensure the input_shape argument in your first layer (e.g., Conv2D or Dense) exactly matches the shape of one sample of your training data (excluding the batch dimension). *Example: For MNIST, it should be (28, 28, 1).*
Flattening Issues	Use model.summary(): After a Conv2D or Pooling layer, the output shape changes. A common mistake is to forget the Flatten() layer before the first Dense layer. model.summary() shows the exact output shape of the last MaxPooling2D layer, which helps verify the subsequent layers are correct.
Output Layer Shape	Check the final Dense layer: The last Dense layer must have units equal to the number of classes. *Example: For MNIST, units=10.*

2. Debugging Incorrect Loss Functions

Choosing the wrong loss function can lead to models that don't learn or crash during training.

Classification Type	Correct Loss Function	Incorrect/Common Mistake
Multi-Class (e.g., MNIST)	categorical_crossentropy	Using this with integer labels. Requires the labels to be One-Hot Encoded (e.g., [0,0,1,0,...]).
Multi-Class	sparse_categorical_crossentropy	Using this with one-hot encoded labels. Requires the labels to be integer-encoded (e.g., 2).
Binary (Two Classes)	binary_crossentropy	Using this with softmax activation on the output layer. Requires sigmoid activation on the output layer.