PLP ACADEMY

SOFTWARE DEVELOPMENT

FEBRUARY 2025 COHORT VII

Specialization: Artificial Intelligence in Software Engineering

GROUP 100

TEAM MEMBERS:

1. STEVE ASUMBA – steveasumba@gmail.com

2. COMFORT NDUNGU – comfort.ndungu.w@gmail.com

3. JORAM MWANYIKA – jorammwanyika@gmail.com

4. ADEOSUN JUWON – juwona71@gmail.com

5. EMMANUELLA AIMALOHI ILEOGBEN - emmanuellaileogben@gmail.com

WEEK 3 ASSIGNMENT: POPULAR AI TOOLS AND FRAMEWORKS

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?

TensorFlow and PyTorch are two most popular and widely used open-sauce deep learning frameworks. While both are powerful and capable of building complex neural networks, they have fundamental differences in their design philosophies, which lead to distinct advantages and disadvantages depending on the use case.

The primary differences are as follows:

a. Debugging –

TensorFlow allows debugging static graphs in older TensorFlow versions is more complex, often requiring the use of TensorBoard or other specific TensorFlow debugging tools.

PyTorch is due to its dynamic graph, debugging is generally easier. You can use standard Python debuggers (like pdb) to step through the code and inspect tensors and gradients at any point.

b. Computational Graphs –

TensorFlow is challenging to debug static graphs (pre-TF 2x). Eager Execution became the default, bringing dynamic graph capabilities similar to PyTorch. This significantly bridged the gap in ease of use and debugging. However, TensorFlow still retains the option for static graphs (tf.function) for performance optimization and deployment.

PyTorch uses dynamic computation graphs, also known as "define-by-run." This means the computational graph is built and modified on the fly as operations are executed. It's ideal for research and rapid prototyping, where you might frequently modify network architectures or handle variable-length inputs.

c. Production Deployment –

TensorFlow has historically had an advantage in production deployment due to its robust ecosystem of tools designed for serving and deployment, such as TensorFlow Serving, TensorFlow Lite (for mobile and IoT devices), and TensorFlow.js (for web browsers).

PyTorch has made significant progress in production deployment with tools like TorchScript (for serialization and deployment in C++ environments) and ONNX (Open Neural Network Exchange) support, which allows models to be exported to other frameworks for inference.

d. Community and Adoption –

TensorFlow is Backed by Google, TensorFlow has a massive community, extensive documentation, and a long history of industrial adoption. It's often favored by larger enterprises for its comprehensive end-to-end solutions.

PyTorch has gained immense popularity in the research community and academia due to its flexibility and ease of experimentation. Its community has grown rapidly and is highly active which was Developed by Meta (Facebook).

e. Ease of Use and Pythonic Nature –

PyTorch is often praised for its more "Pythonic" feel. Its API is more integrated with standard Python programming practices, making it feel more like an extension of Python. Many users find PyTorch's syntax more intuitive and easier to learn, especially for those already proficient in Python.

TensorFlow's original design, with its sessions and placeholders, felt less like typical Python programming. While TensorFlow has made strides in ease of use with TensorFlow 2.0 and Keras (which is now integrated into TensorFlow), some users still find it to have a steeper learning curve compared to PyTorch, especially when dealing with its lower-level APIs.

f. Visualization –

TensorFlow boasts a powerful visualization tool called **TensorBoard**, which provides comprehensive capabilities for visualizing training metrics, model graphs, histograms of weights, and more.

While PyTorch users can also use TensorBoard (with a bit of integration), its native visualization tool, Visdom, is generally considered less feature-rich than TensorBoard. However, there are other community-driven tools and integrations available.

Q1.5 When would you choose one over the other?

1. You choose TensorFlow when:

a. You are deploying models to production at scale in robust deployment system, multi-platform support and scale graph advantages when explicitly using with tf.fuction.

b. You require strong enterprise support and a well-established industrial presence.

c. You need advanced visualization and debugging tools (especially TensorBoard).

d. You are working with Google Cloud TPUs (Tensor Processing Units).

e. You are working on a project that benefits from Keras as a high-level API.

2. You choose PyTorch when:

a. You are in a research environment or doing rapid prototyping, in dynamic computational graphs, easy debugging, and a more pythonic feel.

b. You prefer a more imperative programming style.

c. You prioritize flexibility and customizability.

d. You are working on a project with a strong academic or open-source community focus.

In Conclusion:

The landscape has evolved significantly with TensorFlow 2.0 adopting eager execution, making it more similar to PyTorch in terms of ease of use and debugging for research. However, their core strengths still diverge:

PyTorch for Research and Experimentation: Its flexibility, Pythonic nature, and dynamic graphs make it a favorite for academics, researchers, and anyone who needs to iterate quickly and experiment with novel ideas.

TensorFlow for Production and Scalability: Its robust deployment tools, strong enterprise backing, and optimization capabilities for inference make it a strong choice for bringing models to large-scale production environments.

Ultimately, both frameworks are highly capable, and expertise in either will serve you well in the deep learning field. The "best" choice often comes down to your specific project requirements, team's existing skill set, and your personal comfort level with each framework's design philosophy.

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

Jupyter Notebooks are a versatile tool in Ai development, offering an interactive environment that combines code execution, visualization, and documentation.

1. **Exploratory Data Analysis (EDA) and Model Prototyping:** AI development, particularly in machine learning and deep learning, heavily relies on understanding and preparing the data. Jupyter Notebooks provide an ideal environment for performing Exploratory Data Analysis (EDA) and subsequent data preprocessing steps. Data scientists can import datasets, visualize distributions, identify correlations, detect outliers, handle missing values, and perform feature engineering all within a single, interactive document.

How Jupyter Notebooks Facilitate This:

a. Interactive Code Execution: Data scientists can write and execute code cells incrementally, allowing them to see the immediate results of each transformation or analysis step. For example, after loading data, they can run a cell to view the head of the DataFrame, another to check data types, and subsequent cells to generate histograms or scatter plots.

b. Rich Visualizations: Jupyter Notebooks seamlessly integrate with popular data visualization libraries like Matplotlib, Seaborn, Plotly, and Bokeh. This allows developers to create static, interactive, or animated plots directly within the notebook to uncover patterns, anomalies, and relationships in the data. For instance, visualizing a feature's distribution before and after normalization can be done side-by-side.

c. Narrative and Documentation**:** Beyond just code, markdown cells allow for rich text, equations, and images. This enables data scientists to document their thought process, explain decisions made during data cleaning or feature engineering, and summarize findings from their EDA. This creates a self-contained, reproducible, and easily understandable record of the data preparation pipeline.

d. Iterative Refinement: EDA and model prototyping are rarely linear. Jupyter Notebooks support an iterative workflow where data scientists can try different approaches, observe the results, and quickly revert or modify steps as needed, without having to re-run the entire script.

Example Scenario: A machine learning engineer is tasked with building a model to predict customer churn. In a Jupyter Notebook, they would:

a. Load the customer dataset using pandas.

b. Use df.info() and df.describe() to get a summary.

c. Plot histograms of numerical features to check their distributions and identify skewness.

d. Create box plots to detect outliers in customer spending habits.

e. Use df.isnull().sum() to identify missing values and then write code in separate cells to impute them (e.g., using mean, median, or a more sophisticated method), observing the impact immediately.

f. Generate correlation matrices (e.g., using sns.heatmap) to understand relationships between features.

g. Perform one-hot encoding on categorical features and visualize the new feature distributions.

h. Add markdown cells to explain why certain preprocessing steps were chosen (e.g., "Outliers in 'age' were capped at 90 which is due to domain knowledge").

2. **Model Training, Development and Evaluation:** Jupyter Notebooks are an excellent environment for the entire lifecycle of AI model development, from defining the model architecture to training, evaluating performance, and even hyperparameter tuning. Their interactive nature allows developers to build, train, and test models incrementally, observing the effects of each change.

How do Jupyter Notebooks Facilitate This:

a. Step-by-Step Model Construction: Developers can define different layers of a neural network or components of a machine learning pipeline in separate cells. This allows for easy modification and testing of individual parts of the model. For instance, in Keras or PyTorch, one cell might define the model architecture, another might compile it, and a third might initiate training.

b. Interactive Training Progress Monitoring: During model training, developers can display progress bars, loss curves, and accuracy metrics updated in real-time within the notebook. This immediate feedback is crucial for understanding how the model is learning and identifying issues early on. Libraries like `tqdm` or framework-specific callbacks can be used for this.

c. Direct Evaluation and Error Analysis: After training, models can be evaluated directly within the notebook. Performance metrics (accuracy, precision, recall, F1-score, RMSE, etc.) can be calculated and displayed. Confusion matrices, ROC curves, and other visual diagnostic tools can be generated to understand where the model is making errors, facilitating targeted improvements.

d. Hyperparameter Tuning and Comparison: Developers can run multiple training experiments with different hyperparameters (learning rates, batch sizes, optimizer choices, number of layers, etc.) in separate cells or iteratively, comparing the results directly in the notebook. This makes it easy to track which configurations perform best.

e. Reproducible Experiments: By saving the notebook, the entire sequence of steps, including data loading, preprocessing, model definition, training, and evaluation, is preserved. This makes it straightforward to reproduce experiments, share findings with colleagues, or revisit past work.

Example Scenario: A data scientist is building a deep learning model for image classification:

a. In one cell, they import TensorFlow/Keras or PyTorch and define the convolutional neural network (CNN) architecture.

b. In another cell, they load and preprocess the image dataset (e.g., normalizing pixel values, augmenting images).

c. A separate cell initiates the model training, with the output showing loss and accuracy for each epoch. They might include callbacks to save the best model or reduce the learning rate on plateau.

d. After training, they create a new cell to evaluate the model on the test set, displaying classification reports and confusion matrices.

e. They might then create another cell to visualize a few misclassified images, helping them understand specific failure modes of the model.

f. If the performance isn't satisfactory, they could duplicate the training cells, modify a hyperparameter (e.g., increase the number of filters or change the optimizer), and retrain the model, comparing the new results directly against the previous ones.

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

spaCy significantly enhances NLP tasks compared to basic Python string operations by providing specialized tools for linguistic accuracy, efficiency, and scalability. It is a sophisticated NLP library that provides a robust pipeline that processes raw text and enriches it with linguistic annotations, moving far beyond simple string manipulation.

How it enhances NLP

1. Intelligent Tokenization: spaCy provides highly optimized and language-specific tokenizers that understand linguistic rules. It correctly separates punctuation, handles contractions, and recognizes special cases like URLs, emails, and hashtags.

Enhancement: Instead of text.split(' ') which might give ['Hello!', 'How', 'are', 'you?'], spaCy's tokenizer would produce ['Hello', '!', 'How', 'are', 'you', '?'], providing discrete, meaningful tokens for further analysis.

2. Efficiency and Performance: spaCy is engineered for speed and efficiency, especially for large volumes of text. It's written in Cython (a superset of Python that compiles to C), making its core operations much faster than pure Python string manipulation or iterating through strings with complex regexes.

3. Pre-trained Models and Transfer Learning: spaCy comes with pre-trained statistical models for various languages. These models are trained on large corpora and are highly accurate for common NLP tasks. This means you don't have to train models from scratch for basic tasks like POS tagging or NER, saving immense time and computational resources.

Enhancement: You can load a pre-trained model (en_core_web_sm, de_core_news_md, etc.) and immediately apply its intelligence to your text, rather than trying to hand-code rules for every linguistic nuance.

4. Integration with Machine Learning Pipelines: The structured output of spaCy (Doc and Token objects with their attributes) makes it much easier to integrate into machine learning pipelines. You can easily extract features (lemmas, POS tags, named entities) to feed into text classification models, topic modeling, or other downstream NLP applications.

5. Rule-based Matching: spaCy also offers powerful rule-based matching capabilities while focused on statistical models (Matcher and PhraseMatcher) that combine linguistic pattersna with token attributes. This is far more robust than raw regular expressions for specific information extraction.

2. Comparative Analysis

Compare Scikit-learn and TensorFlow in terms of:

- Target applications (e.g., classical ML vs. deep learning).
- Ease of use for beginners.
- Community support.

Scikit-learn and TensorFlow are two fundamental libraries in the Python machine learning ecosystem, but they cater to different needs and scales of problems.

Target applications

1. Sci-kit Learn:

a. Classical Machine Learning, sci-kit learn is the go-to library for traditional or classical machine learning algorithms that includes: Supervised Learning: Classification (e.g., Logistic Regression, Support Vector Machines, Decision Trees, Random Forests, Gradient Boosting) and Regression (e.g., Linear Regression, Ridge Regression, Lasso, SVR).

Unsupervised Learning: Clustering (e.g., K-Means, DBSCAN, Agglomerative Clustering) and Dimensionality Reduction (e.g., Principal Component Analysis (PCA), t-SNE, LDA).

Model Selection and Evaluation: Tools for cross-validation, hyperparameter tuning (GridSearchCV, RandomizedSearchCV), and various metrics for evaluating model performance.

Data Preprocessing: Features for scaling (MinMaxScaler, StandardScaler), encoding categorical variables (OneHotEncoder, LabelEncoder), imputation of missing values, and feature selection.

b. Structured Data excels with structured, tabular data (like CSV files or SQL database dumps) and is widely used for predictive analytics in business, finance, healthcare, and more.

c. Small to Medium-Sized Datasets - While it can handle moderately large datasets, it's primarily designed for single-machine, CPU-based computations and might become inefficient for truly "big data" problems that require distributed computing or specialized hardware.

2. TensorFlow:

It is explicitly designed for deep learning, which involves building and training complex neural networks. Its strengths lie in neural networks, (e.g. convolutional neural networks CNNs, feedforward neural networks FNNs, recurrent neural networks RNNs, and generative adversarial networks GANs) and reinforcement learning.

a. Unstructured Data: It's particularly well-suited for working with unstructured data like images, audio, video, and raw text, where deep learning models often achieve state-of-the-art results.

b. Large-Scale and Distributed Computing: TensorFlow is built for scalability. It can leverage GPUs, TPUs (Tensor Processing Units, Google's custom ASICs), and distributed computing environments to train massive models on very large datasets.

c. Production Deployment: TensorFlow provides a comprehensive ecosystem (TensorFlow Serving, TensorFlow Lite, TensorFlow.js) for deploying deep learning models across various platforms, from cloud servers to mobile devices and web browsers.

Ease of Use for Beginners

1. Sci-kit Learn:

a. Excellent for beginners: Sci-kit learn is widely considered one of the most beginner-friendly machine learning libraries.

b. It offers a highly consistent and intuitive API. Once you learn the basic workflow (.fit(), .predict(), .transform()), you can apply it to almost all algorithms.

c. It has clear documentation which is extensive, well-organized and includes numerous examples for each algorithm, making it easy to understand and implement.

d. It abstracts away much of the underlying mathematical complexity, allowing users to focus on applying algorithms rather than implementing them scratch.

e. No GPU/TPU Requirement: It runs efficiently on CPUs, for its typical use cases, meaning beginners don’t need specialized hardware to get started.

2. TensorFlow:

a. It offers steeper learning curve which is improved TF 2.0 and Keras. This is caused by its static graph computation model which requires less intuitive for beginners with sessions and placeholders,

b. Understanding deep learning concepts even with Keras is a prerequisite (neural network architecture, backpropagation, optimizer, loss functions, tensors), which is inherently more complex than understanding classical machine learning algorithms.

c. Keras Integration (TF 2.0): With TensorFlow 2.0, the high-level Keras API became the official and default way to build models. Keras significantly simplifies building neural networks, making TensorFlow musch more accessible for beginners than before. You can define models' layer by layer with just a few lines of code.

d. The default eager execution mode in TF 2.0 makes debugging easier and code execution more intuitive, similar to regular Python.

e. For practical deep learning tasks, leveraging GPUs (and sometimes TPUs) becomes essential which adds another layer of setup and understanding.

Community Support

1. Sci-kit Learn:

a. It offers a strong and mature community, very active and supportive community. Its official documentation is a gold-standard.

b. Stack overflow: There's a massive wealth of answers and discussions on Stack Overflow with the [sci-kit learn] tag.

c. It is widely used in both academia and industry for classical machine learning tasks, leading to a broad user who contribute examples, tutorials and solutions. It has a well-defined release cycle and is known for its stability.

2. TensorFlow:

a. It offers a massive and global community which is backed by Google, it has an enormous, distributed, and highly active community.

b. Google Backing: Direct support and continuous development from Google ensure its long-term viability and cutting-edge research integration.

c. Extensive Resources: A vast array of official tutorials, guides, and documentation (including the Keras documentation, which is now part of TensorFlow's official docs).

d. The rapid evolution and frequent updates (especially between TF 1.x and 2.x) can sometimes mean that older tutorials or code snippets might not be directly compatible with the latest versions.

e. The Ecosystem of pre-trained models and reusable components contributed by Google and the community. Dedicated TensorFlow forums, mailing lists, and a huge presence on stack overflow.

Part 3: Ethics & Optimization (10%)

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?

The MNIST dataset is a widely used benchmark in machine learning, consisting of handwritten digits (0-9). While it is relatively simple and well-balanced, potential biases could still exist. Those potential biases and how tools like TensorFlow Fairness Indicators and spaCy's rule-based systems could help mitigate them.

Potential Biases in MNIST

a. Demographic Bias: The dataset consists of digits written by American Census Bureau employees and American high school students, which may not represent global handwriting styles (e.g., different cultural ways of writing digits like "1" or "7"). The impact on the model trained (MNIST) may perform poorly on digits written by non-Americans.

b. Noise and Augmentation Bias: The dataset is clean and preprocessed uniformly, which may not reflect real-world noisy or distorted digits (e.g., smudged or rotated digits).

c. Digit Imbalance: The MNIST is generally balanced (each digit has ~6K-7K samples), slight imbalances could still lead to minor biases in classification accuracy.

d. Stroke Style Bias: Some digits may be written with varying stroke thicknesses or styles (e.g,. cursive vs. print), which could favor certain writing styles over others.

Mitigating Bias with TensorFlow Fairness Indicators

TensorFlow Fairness Indicators helps evaluate model fairness across different subgroups. For MNIST, we could:

- Evaluate per-digit accuracy disparities to ensure no digit is systematically misclassified.

- Check for bias in synthetic augmentations (e.g., rotated or distorted digits) to ensure robustness.

- Compare performance on subsets (e.g., digits written by different demographic groups if metadata were available).

python

```
from tensorflow_model_analysis.addons.fairness.view import widget_view
from tensorflow_model_analysis.addons.fairness.metrics import fairness_indicators

# Evaluate model fairness across digit classes
fairness_metrics = fairness_indicators(
```

```
    thresholds= [0.5],
    labels=mnist_test_labels,
    predictions=model.predict(mnist_test_images)
)
widget_view.render_fairness_indicator(fairness_metrics)
```

Mitigating Bias with spaCy's Rule-Based Systems

While spaCy is primarily for NLP, its rule-based matching could be adapted for structured data like MNIST:

- Detect underrepresented patterns (e.g., digits with unusual strokes and flag them for additional training.

- Generate synthetic rules to augment data (e.g., perturbing digits to simulate diverse handwriting styles).

- Balance dataset subgroups by programmatically identifying and resampling rare cases.

Example (Hypothetical):

python

```
import spacy
from spacy.matcher import Matcher

nlp = spacy.blank("en")
matcher = Matcher(nlp.vocab)

# Rule: Detect "1" written with a base serif (common in some cultures)
pattern = [{"SHAPE": "straight"}, {"SHAPE": "horizontal"}] # Simplified example
matcher.add("SERIF_ONE", [pattern])

# Apply to images (hypothetical adaptation for digits)
# Could flag samples needing augmentation.
```

Conclusion

- TensorFlow Fairness Indicators helps quantify and mitigate biases in model predictions.

- spaCy's rule-based systems (if adapted) could help identify and augment underrepresented digit variations.

- Additional mitigation strategies:

i. Data augmentation (e.g., varying stroke styles, rotations).

ii. Diverse data collection (e.g., including international handwriting samples).

iii. Adversarial debiasing (training models to be invariant to certain biases).

2. Troubleshooting Challenge

- Buggy Code: A provided TensorFlow script has errors (e.g., dimension mismatches, incorrect loss functions). Debug and fix the code.

Below is a buggy TensorFlow script for training an MNIST classifier, along with explanations of the errors and their fixes.

Issues Identified (Buggy Code)

python

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()

# Reshape and normalize images (BUG: Missing channel dimension)
train_images = train_images.reshape((60000, 28, 28))
test_images = test_images.reshape((10000, 28, 28))

# Normalize (BUG: Incorrect scaling, should be 255.0)
train_images = train_images / 255
test_images = test_images / 255

# Define model (BUG: Missing Flatten layer, incorrect output units)
model = models.Sequential([
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile (BUG: Incorrect loss function for multi-class classification)
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Train (BUG: Missing validation split)
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Fixed Code (Debugged)

python

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()

# Reshape and normalize images (FIX: Add channel=1 for grayscale)
train_images = train_images.reshape((60000, 28, 28, 1))  # Shape: (60000, 28, 28, 1)
test_images = test_images.reshape((10000, 28, 28, 1))

# Normalize (FIX: Divide by 255.0 for [0,1] range)
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# Define model (FIX: Add Flatten, correct output units)
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28, 1)),  # Flatten 28x28 to 784 pixels
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')    # 10 classes for digits 0-9
])

# Compile (FIX: Use categorical crossentropy)
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',  # For integer labels
              metrics=['accuracy'])

# Train (FIX: Add validation data)
model.fit(train_images, train_labels,
          validation_data=(test_images, test_labels),  # Monitor test accuracy
          epochs=5,
          batch_size=64)
```

Bug Breakdown & Fixes

1. Input Shape Error

- Bug: Images were reshaped to (60000, 28, 28) but missing the channel dimension (1 for grayscale).

- Fix: Use reshape ((60000, 28, 28, 1)) for Conv2D compatibility (if used later).

2. Normalization Error

- Bug: Integer division (/ 255) truncates values to 0.

- Fix: Use floating-point division (`/ 255.0`).

3. Missing Flatten Layer

- Bug: Dense layers expect 1D input, but images are 2D (28x28).
- Fix: Add `Flatten()` before the first Dense layer.

4. Incorrect Loss Function

- Bug: `binary_crossentropy` is for binary classification.
- Fix: Use `sparse_categorical_crossentropy` (since labels are integers).

5. No Validation Data

- Bug: Training without validation makes it hard to detect overfitting.
- Fix: Add `validation_data=(test_images, test_labels)`.

Key Takeaways

- Always check input shapes (`model.summary()` helps debug).
- Use the correct loss function for the task (e.g., `sparse_categorical_crossentropy` for integer labels).
- Normalize data properly (`/ 255.0` for images).
- Validate models with a holdout set.

Bonus Task (Extra 10%)

Deploy Your Model:

Use **Streamlit** or **Flask** to create a web interface for your MNIST classifier.

Deployment Link/Demo Live Link: https://ai-tools-and-applicartions-t1wd.vercel.app/

Submit a screenshot and a live demo link.