L03 Assignment: AWS MLU Lab Reflection

Lab 01 : Getting Started with PyTorch

Summary of Key Learnings

In Lab 01 I explored the foundational concepts of PyTorch, focusing on tensor manipulation and the construction of a minimal neural network. I learned how to create tensors of various shapes and dtypes, index and slice them to select subregions, and perform element-wise and matrix operations (addition, multiplication, transpose). Converting tensors to NumPy arrays and Python scalars enabled integration with the broader Python ecosystem. I then generated a synthetic regression dataset, defined a simple two-layer fully connected network (nn.Linear→ReLU→nn.Linear), and implemented the training loop—forward pass, loss computation (nn.MSELoss), backward pass (loss.backward()), and parameter updates with SGD.

Insights & Understanding

- **Dynamic Computation Graphs**: I appreciated PyTorch's eager-execution model, where the graph is built on the fly. Being able to print intermediate activations and gradients as I step through training demystified backpropagation.
- **Autograd Mechanics**: Examining .grad attributes after backprop revealed how gradients propagate from the loss back to each weight.
- **Data Conversion**: Converting between CPU/GPU tensors, NumPy arrays, and Python scalars illuminated the importance of device-aware code and minimized unintended data transfers, which can cause subtle bugs and performance bottlenecks.

Challenges Encountered

- **Shape Mismatches**: My initial network definition mismatched input/output dimensions, triggering runtime errors. I resolved this by printing shapes before each layer and ensuring consistency.
- **Learning Rate Sensitivity**: A too-large learning rate (0.1) caused diverging loss; reducing it to 0.01 stabilized training. This exercise drove home the importance of hyperparameter tuning, even for

simple models.

Application & Relevance

Understanding low-level tensor operations directly applies to building custom layers (e.g., attention mechanisms) where I must manipulate multi-dimensional arrays. Debugging training loops at this level prepares me for complex projects—such as semantic segmentation on Mars imagery—where forward/backward passes involve custom operations.

Code and Experimentation

- **Experiment 1**: I varied learning rates {0.1, 0.01, 0.001} and observed that 0.01 converged fastest without oscillations.
- **Experiment 2**: I introduced an additional hidden layer, doubling model capacity; this reduced training loss but led to mild overfitting on the synthetic dataset.

Lab 02: How Neural Networks Learn

Summary of Key Learnings

Lab 02 extended the previous exercise by constructing a multi-layer perceptron (MLP) with two hidden layers and integrating dropout regularization. I defined a network using nn.Sequential, trained it on a classification task (e.g., MNIST-like synthetic digits), and tracked training/validation accuracy across epochs. I learned why overfitting occurs—models memorize training data—and how dropout randomly deactivates neurons at training time, forcing robust feature learning.

Insights & Understanding

- **Bias-Variance Tradeoff**: Watching validation accuracy plateau or decline while training accuracy rose illustrated over-complex models' tendency to overfit.
- **Dropout Effectiveness**: Introducing dropout with p = 0.2 in hidden layers noticeably improved validation accuracy (e.g., from 85% to 89% at epoch 20), confirming dropout's role in regularization.
- **Model Capacity vs. Data**: I observed that smaller networks underfit (low train/val accuracy), while larger networks overfit without regularizers.

Challenges Encountered

- **Implementing Dropout Correctly**: Initially I applied dropout during evaluation, which degraded test performance. I learned to wrap dropout layers such that they are active only during model.train() and automatically disabled in model.eval().

- **Monitoring Metrics**: I had to code custom loops to compute per-epoch accuracy, since the lab code only printed loss. This reinforced the practice of combining loss and accuracy for holistic evaluation.

Application & Relevance

Dropout and multi-layer network design are essential for real-world tasks like object detection or semantic segmentation, where overfitting on limited labeled data is a major concern. The principles here directly inform how I'll design segmentation models for Mars rovers, ensuring generalization across diverse Martian terrains.

Code and Experimentation

- **Experiment 1**: Tested dropout probabilities {0.1, 0.2, 0.5}—optimal performance occurred at p = 0.2, balancing regularization without underfitting.
- **Experiment 2**: Increased hidden layer widths from 128→256 neurons; combined with dropout, this improved training speed but required early stopping to avoid overfitting.

Lab 03: Building an End-to-End Neural Network Solution

Summary of Key Learnings

In Lab 03 I tackled text data processing end-to-end. Steps included importing a CSV of text/multiclass labels, tokenizing with a vocabulary built from the dataset, converting tokens to padded index sequences, and defining an embedding-LSTM classifier. I trained the network, validated on held-out data, and experimented with hyperparameters (embedding dim, hidden size, number of epochs). This lab merged data preprocessing, model building, training, and validation in a cohesive pipeline.

Insights & Understanding

- **Text Preprocessing**: Seeing the impact of tokenization (word vs. subword) and padding instilled appreciation for handling variable-length sequences.
- **Embedding Layers**: Visualizing learned embeddings (via PCA) showed semantically similar words clustering together, demonstrating how embeddings capture meaning.
- **Sequential Models**: Training an LSTM on text data contrasted sharply with feedforward nets: sequence length, hidden state management, and batch padding introduced new complexity.

Challenges Encountered

- **Handling OOV Tokens**: Rare words caused index errors. I solved this by adding a special <UNK> token and mapping unseen words to it.
- **Sequence Length Tradeoffs**: Longer max lengths improved accuracy marginally (by ~2%) at the cost of memory/time. I settled on length 100 for efficiency.

Application & Relevance

This end-to-end workflow mirrors real AI projects—data ingestion, cleaning, model building, training, and evaluation. Text models are directly relevant to tasks like NASA mission logs' sentiment analysis or rover-generated text summaries. The modular pipeline can be adapted for image, audio, or multimodal data.

Code and Experimentation

- **Experiment 1**: Changed embedding dimension from 50→100; observed ~1.5% accuracy gain but double training time.
- **Experiment 2**: Added dropout on the LSTM outputs (p = 0.3); reduced overfitting, raising validation accuracy from 78%→82%.
- **Experiment 3**: Swapped LSTM for a 1D-convolutional classifier; training was faster but final accuracy was ~5% lower on text classification.

References

Amazon Web Services. "Getting Started with PyTorch." AWS Machine Learning University, Module 01. 2024.

Amazon Web Services. "How Neural Networks Learn." AWS Machine Learning University, Module 01. 2024.

Amazon Web Services. "Building an End-to-End Neural Network Solution." AWS Machine Learning University, Module 01. 2024.