**Lab 01: Getting Started with PyTorch**

This lab introduced the fundamental concepts of PyTorch, focusing on tensor initialization, manipulation, and key operations. I explored various methods for creating tensors, such as torch.zeros(), torch.ones(), and torch.randn(), indexing and slicing, complex operations. Also examined how PyTorch’s automatic differentiation simplifies gradient computations for neural networks. One key takeaway was recognizing tensors as the fundamental building blocks of deep learning architecture. PyTorch is a powerful framework for deep learning applications.

**Summary of Key Learnings**

This notebook introduced PyTorch as a deep learning framework, focusing on data manipulation and differentiation, both essential for training neural networks. The key concepts covered include:

* Creating tensors and converting them to/from NumPy arrays
* Indexing and slicing tensors efficiently
* Performing common tensor operations (e.g., addition, multiplication, reshaping)
* Utilizing automatic differentiation for optimization
* Understanding how PyTorch differs from NumPy, particularly with GPU acceleration

**Insights & Understanding**

* PyTorch’s tensor class is crucial for deep learning as it enables efficient computations, especially with GPU support.
* Manipulating tensors is fundamental for preprocessing data before feeding it into models.
* Differentiation is a core part of optimization algorithms like backpropagation, which is key to training deep learning models.

**Challenges Encountered**

One challenge I faced was understanding how PyTorch manages tensors across different hardware (CPU vs. GPU). This understanding is crucial because efficient tensor operations form the backbone of AI applications like image and text processing. Additionally, getting comfortable with indexing and slicing tensors, especially in higher dimensions, took some practice. Another tricky part was ensuring “requires\_grad=True”, which enables gradient tracking for optimization.

**Application & Relevance**

These concepts are foundational for building neural networks and are applicable in real-world AI tasks such as:

* Image recognition (e.g., CNNs using PyTorch for classification)
* NLP (token embeddings and sequence modeling)
* Recommendation systems (e.g., collaborative filtering with deep learning)

Mastering efficient tensor mathematical operations is essential for scalability and faster model training, making PyTorch a valuable tool for AI practitioners.

**Code and Experimentation**

* Deepening my understanding
* Conducting hands-on experiments
* Creating and manipulating tensors
* Performing tensor operations
* Using gradient tracking

**Conclusion**

This lab provided a strong foundation in tensor operations, that is essential for implementing deep learning models. By practicing indexing, slicing, broadcasting, and differentiation, I now have a better grasp of how PyTorch optimizes computations for AI applications. Further experimentation with GPU acceleration and gradient-based optimization will enhance my ability to design scalable AI models.

**Lab 02: How Neural Networks Learn**

**Summary of Key Learnings**

This lab focused on building a simple neural network using logistic regression as the base model. The primary objectives included:

* Generating a simulated dataset for binary classification
* Defining a neural network using matrix multiplications and activation functions
* Implementing forward and backward propagation to train the model
* Using loss functions and optimization algorithms to improve performance
* Training the model with multiple epochs and evaluating improvements over time

Key takeaways include the importance of gradient descent in training neural networks and how adjusting hyperparameters like the learning rate impacts convergence speed and model stability.

**Insights & Understanding**

Forward and backward propagation are fundamental to training neural networks. Forward propagation computes the predictions, while backward propagation updates the weights to minimize error. Gradient accumulation in PyTorch needs careful handling. Using optimizer.zero\_grad() is crucial to prevent incorrect weight updates from accumulating past gradients.

* A high learning rate can lead to instability (loss oscillations or divergence).
* A low learning rate results in slow convergence, making training inefficient.
* Sigmoid is useful for binary classification but can suffer from vanishing gradients.
* SGD (Stochastic Gradient Descent) helps optimize models efficiently, though it may require fine-tuning of batch sizes and momentum for better convergence.

**Challenges Encountered**

One of the key challenges encountered during the lab was setting up gradients before backpropagation using optimizer.zero\_grad() in order to avoid unexpected weight updates that can affect the training process. Additionally, finding the right balance between batch size, the number of epochs, and overall model performance proved to be complex, requiring careful tuning to achieve optimal results. Another issue observed was the shrinking of gradients in deeper models, which made training inefficient and slowed down optimization.

**Application & Relevance**

* The foundation of this lab applies to more complex architectures like CNNs, RNNs, and Transformers.
* Understanding backpropagation and optimization techniques helps in improving performance.
* Learning rate, batch size, and activation functions significantly impact training efficiency.
* Concepts from this lab are essential for text classification, image recognition, and predictive modeling.

**Code and Experimentation**

* Generating a dataset
* Defining the model using PyTorch
* Setting up loss and optimizer
* Training a model
* Making predictions

**Conclusion**

This lab provided hands-on experience with the core components of neural networks, from defining models and loss functions to training and optimizing. The lessons learned are valuable for scaling up to deep learning architectures like CNNs and Transformers. Experimenting with activation functions, optimization techniques, and learning rates improved my understanding of model training dynamics.

**Lab 03: First Example of Neural Networks**

This lab involved building and training a simple neural network for classification. I learned how to define a model using torch nn module, set up a training loop, and evaluate model performance with accuracy metrics. A key insight was understanding how adding layers and neurons influences model complexity. More layers help the network learn essential patterns but increase the risk of overfitting. One challenge was selecting the right number of hidden units. Too few led to underfitting, while too many caused overfitting. Using validation loss as a guide helped me find the right balance. Experimenting with batch sizes revealed that smaller batches introduced more noise, which improved the model's ability to generalize, while larger batches led to smoother updates but required more memory. This lab provided hands-on experience with neural networks, which is essential for real-world AI applications such as image classification and NLP tasks. It also laid a solid foundation in PyTorch, preparing me for more advanced deep learning concepts.

**Summary of Key Learnings**

In this lab I explored the fundamentals for text data processing. The lab covered building a neural network using PyTorch’s torch.nn module, training it with a classification task, and validating its performance using accuracy metrics. I gained insights into hyperparameter tuning, dropout layers for overfitting prevention, and the impact of batch sizes on generalization. The hands-on experience reinforced how different architectural choices affect model performance and how neural networks can be applied to real-world datasets.

**Insights & Understanding**

* Neural networks with multiple layers improve pattern recognition but increase computational complexity.
* Dropout layers are effective in reducing overfitting by randomly deactivating neurons during training.
* Selecting an appropriate number of hidden units is crucial (too few leads to underfitting, too many cause overfitting).
* Smaller batch sizes introduce noise, improving generalization, while larger batches lead to smoother but more memory updates.
* Monitoring validation loss is key to determining when a model starts overfitting.

**Challenges Encountered**

* Finding the right number of layers and neurons required multiple iterations to avoid underfitting or overfitting.
* Adjusting batch size, learning rate, and dropout rate required careful observation of loss trends.
* Encoding categorical features effectively while preserving meaningful information was challenging.

**Application & Relevance**

* This lab provided essential skills for working with text-based AI applications such as sentiment analysis, chatbot development, and recommendation systems.
* Understanding neural network architecture is critical for designing models in image classification, NLP, and structured data tasks.
* Hyperparameter tuning techniques learned here will be valuable when working on more complex deep learning models, including transformers and CNNs.

**Code and Experimentation**

* Preprocessed and encoded text data from the Austin Animal Center dataset.
* Implemented model PyTorch’s torch.nn.module.
* Trained and validated the model while adjusting hyperparameters.
* Monitored performance metrics (loss and accuracy) to refine model architecture.
* Experimented with dropout layers to assess their effect on reducing overfitting.

**Conclusion**

This lab was a valuable hands-on experience in building and training multilayer perceptron for text classification. It reinforced the importance of architectural choices, data preprocessing, and hyperparameter tuning in deep learning. The role of dropout in preventing overfitting and the impact of batch sizes on model generalization are crucial for more advanced projects in AI and NLP. Refining these skills by experimenting with deeper architectures, different activation functions, and optimization strategies will be beneficial. This lab also laid a strong foundation for working with more complex neural networks, such as convolutional neural networks (CNNs) and transformers. And it is a great step toward mastering deep learning techniques for real-world applications

**Summary of AWS Labs**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Lab** | **Summary of Key Learnings** | **Insights & Understanding** | **Challenges Encountered** | **Application & Relevance** | **Code and Experimentation** |
| **Lab 01: Getting Started with PyTorch** | Covered the basics of PyTorch, including tensors and operations. Learned how to create and manipulate tensors and perform basic computations. | Gained insight into how PyTorch simplifies tensor operations compared to NumPy. | Faced initial challenges with understanding how tensors work. Resolved by reviewing examples. | Understanding tensors is crucial for AI applications like deep learning. Plan to use PyTorch in future projects involving NLP and computer vision. | Experimented with different tensor operations. Observed performance benefits in specific cases. |
| **Lab 02: How Neural Networks Learn** | Explored how neural networks adjust weights using backpropagation and gradient descent. Implemented a simple learning model. | Gained a deeper understanding of how optimization works in neural networks. Gradient descent fine-tunes parameters effectively. | Before was struggling how gradients are computed. Clarified by visualizing backpropagation step-by-step. | Essential knowledge for training AI models. Helps in building models that optimize performance effectively. | The learning rate significantly affects optimization speed. |
| **Lab 03: First Example of Neural Networks** | Built and trained a basic neural network using PyTorch. Explored different layers and activation functions. | Realized how flexible PyTorch is for building networks. Usage of defining layers. | Difficulty with choosing appropriate hyperparameters. Trial and errors. | Important for AI model development, particularly in NLP and image recognition tasks. | Optimized the hidden layers and activation functions. Found that increasing layers improved accuracy but also increased training time. |

**Resources:**

1. <https://pytorch.org/>
2. [Build the Neural Network — PyTorch Tutorials 2.6.0+cu124 documentation](https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html)
3. [Learn to Build a Neural Network From Scratch — Yes, Really. | by Aadil Mallick | Medium](https://medium.com/@waadlingaadil/learn-to-build-a-neural-network-from-scratch-yes-really-cac4ca457efc)