Faiza Abdullah

Lab 03

ITAI 2376 Deep Learning in Artificial Intelligence

Professor: [Patricia Mcmanus](https://eagleonline.hccs.edu/courses/282423/users/264039)

**Application of Deep Learning to Text and Image Data: Module 1, Lab 1**

**Getting Started with PyTorch**

**INTRODUCTION**

This reflection covers my learning experience with Lab 1, which consisted of two main notebooks focusing on PyTorch fundamentals and basic neural network implementation. The lab aimed to build foundational understanding of tensor operations and neural network architecture using PyTorch.

**DESCRIPTION OF EXPERIENCE**

The exercise began by examining the model's architecture through its summary, which revealed its complex structure of multiple layers. The lab introduced PyTorch, a deep learning framework, and covered essential operations such as tensor manipulation, indexing, slicing, and performing mathematical operations. Additionally, the lab emphasized the importance of automatic differentiation and GPU utilization for efficient deep learning computations. The lab environment provided pre-written code cells that handled the technical aspects, allowing us to focus on understanding the conceptual framework and making/interpreting predictions.

**PERSONAL REFLECTION**

**Thoughts and Feelings:** Initially, I was excited to explore PyTorch, given its importance in deep learning. However, I also anticipated challenges in understanding the syntax and functionality of PyTorch, especially regarding tensors and automatic differentiation.

**Analysis and Interpretation:** The lab provided a hands-on experience with PyTorch tensors, which are fundamental to neural network computations. One key takeaway was understanding how PyTorch enables seamless GPU acceleration, making computations faster. Another crucial aspect was how PyTorch automatically computes gradients using the autograd package, simplifying the optimization process. For enlightenment, I tried interpreting outputs step by step.

**Lab 1-1: Getting Started with PyTorch**

1. **Tensor Creation (Creating Numbers)**

x = torch.arange(12)

*# Output: tensor([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])*

*# Imagine you're counting from 0 to 11*

x.shape

*# Output: torch.Size([12])*

**Understanding**:

* Demonstrates how to create a basic tensor using torch.arange()
* Shows how to check the shape of a tensor

Think of this like creating a row of numbered boxes where instead of writing numbers manually, you are telling the computer: "Give me numbers from 0 to 11".

1. **Tensor Reshaping (Reshaping Numbers)**

x\_reshaped = x.reshape(3, 4)

*# Output:*

*# tensor([[ 0, 1, 2, 3],*

*# [ 4, 5, 6, 7],*

*# [ 8, 9, 10, 11]])*

*# Rearranging the boxes into 3 rows, 4 columns*

**Understanding**:

* Illustrates how to reshape tensors
* Shows flexibility in tensor manipulation
* Demonstrates implicit reshaping using -1

**Simple Explanation**:

* Imagine you have a long line of 12 toys
* Now you want to arrange them in 3 rows of 4 toys each
* PyTorch helps you rearrange without losing any toys
* It's like reorganizing your bookshelf without adding or removing books

1. **3. Creating Special Number Grids (Tensor Initialization)**

zeros\_tensor = torch.zeros((2, 3, 4))

ones\_tensor = torch.ones((2, 3, 4))

random\_tensor = torch.randn(3, 4)

**Understanding**:

* Multiple ways to initialize tensors
* Different initialization methods for various neural network needs
* Random initialization for weights
* Zero/One initialization for specific purposes / bias initialization

**Simple Explanation**:

* **Zeros**: Like creating a grid of blank white papers
* **Ones**: Like creating a grid where every paper is filled with the number 1
* **Random**: Like rolling dice to fill each square with a different number
* Useful for different tasks, just like how you might need blank, pre-marked, or randomly filled papers for different projects

1. **Checking Number Properties (Tensor Attributes)**

x.shape, x.numel(), x.dtype

*# Output:*

*# (torch.Size([3, 4]), 12, torch.float32)*

*# Like checking the details of your toy collection*

**Understanding**:

* How to inspect tensor properties
* Understanding tensor dimensions
* Checking data type and number of elements

**Simple Explanation**:

* **Shape**: How many rows and columns you have
* **Number of Elements**: Total count of items
* **Data Type**: What kind of numbers you're using (whole numbers, decimal numbers)
* Similar to checking quantity of your toys, their arrangement, and what they're made of

1. Indexing and Slicing

x[0, 0] *# Access specific element*

x[1, 2] = 9 *# Modify specific element*

x[0:2, :] = 12 *# Slice and modify*

**Understanding**:

* Precise element access in multidimensional tensors
* Modifying specific tensor elements
* Slicing techniques in PyTorch

1. **Playing with Numbers (Tensor Operations)**

x + y *# Element-wise addition*

torch.dot(x, y) *# Special math operation - Dot product*

torch.matmul(A, x) *# Matrix multiplication*

**Understanding**:

* Element-wise operations
* Linear algebra operations
* Matrix multiplication techniques

**Simple Explanation**:

* **Addition**: Like combining two sets of toys
* **Dot Product**: A special way of comparing two sets of numbers
* **Matrix Multiplication**: A complex way of transforming entire collections of numbers
* Think of it like advanced puzzle solving with numbers

1. **Learning from Mistakes (Automatic Differentiation)**

x.requires\_grad\_(True)

y = 0.6 \* torch.dot(x, x)

y.backward()

x.grad

**Understanding**:

* Setting up gradient tracking
* Automatic gradient computation
* Backpropagation mechanism

**Simple Explanation**:

* Imagine a robot that learns to throw a ball
* Each throw, it remembers how far it was from the target
* Next time, it adjusts its throw based on previous mistakes
* PyTorch does this automatically for mathematical calculations

**Lab 1-2: Neural Network Architecture - Building a Simple Decision Maker**

1. **Creating Practice Data (Dataset Creation)**

x, y = make\_circles(

n\_samples=750,

shuffle=True,

random\_state=42,

noise=0.05,

factor=0.3

)

**Understanding**:

* Creating synthetic datasets
* Control over dataset characteristics
* Preparing data for machine learning

**Simple Explanation**:

* Imagine drawing two overlapping colored circles
* Blue dots in one circle, red dots in another
* Slightly blurry edges to make it challenging
* This is how machines learn to distinguish between different groups

1. **Creating a Simple Brain (Neural Network Architecture)**

net = nn.Sequential(

nn.Linear(2, 1), *# Linear layer*

nn.Sigmoid() *# Activation function*

)

**Understanding**:

* Basic neural network structure
* Using Sequential API
* Combining linear transformation and activation

**Simple Explanation**:

* Like creating a simple light switch
* Input: Two pieces of information
* Output: Likelihood of being in one group or another
* The Sigmoid function is like a dimmer switch that gives a percentage instead of just on/off

1. **Weight Initialization**

def xavier\_init\_weights(m):

if type(m) == nn.Linear:

torch.nn.init.xavier\_uniform\_(m.weight)

torch.nn.init.zeros\_(m.bias)

net.apply(xavier\_init\_weights)

**Understanding**:

* Importance of weight initialization
* Xavier initialization technique
* Applying initialization to neural network layers

1. **Teaching the Brain (Training Process - Loss and Optimization)**

loss = nn.BCELoss(reduction="none")

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

**Understanding**:

* Choosing appropriate loss function
* Selecting optimization algorithm
* Configuring learning rate
* Training loop implementation - showing decreasing loss over epochs
* Loss tracking and visualization

**Simple Explanation**:

* **Loss Function**: Like a teacher grading answers
* **Optimizer**: Like a coach helping improve performance
* Learns by repeatedly trying, checking mistakes, and adjusting

1. Training Loop

for epoch in range(num\_epochs):

optimizer.zero\_grad()

output = net(X)

L = loss(output, y).sum()

L.backward()

optimizer.step()

**Understanding**:

* Complete training process
* Gradient computation
* Parameter updates
* Loss calculation

**Real-World Comparison**

🧩 **Toy Sorting Machine**

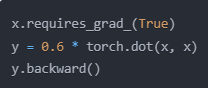
* Input: Mixed toys
* Brain: Simple decision rules
* Learning: Gets better at sorting with practice
* Example: Sorting LEGO pieces by color or size

🚗 **Self-Driving Car**

* Input: Camera and sensor data
* Brain: Complex decision network
* Learning: Improves at recognizing road signs, obstacles
* Similar process, but much more complex

**Connections to Theoretical Knowledge:** This progression from basic tensor operations to a complete neural network implementation provides a comprehensive foundation for understanding deep learning with PyTorch. The labs effectively bridged theoretical concepts with practical implementation, showing how mathematical operations translate into code and how they combine to create a learning system. The outputs and visualizations throughout the labs helped reinforce understanding by providing immediate feedback on operations and showing the learning process in action through loss curves and data visualizations. Additionally, learning about automatic differentiation deepened my understanding of backpropagation in neural networks.

**Critical Thinking:** While working through the lab, I realized how PyTorch streamlines complex computations. However, I encountered difficulties in understanding the labs at first however exploring the documentation helped clarify the flow. One of the most enlightening aspects was understanding how PyTorch handles automatic differentiation. The concept of requires\_grad=True and how it enables backpropagation was particularly interesting:



The visualization of the training process through loss plots helped solidify my understanding of how neural networks learn over time. The use of simulated data in Lab 1-2 was particularly effective as it provided a controlled environment to understand model behavior.

**Challenges Encountered:**

1. Understanding tensor dimensionality: Initially, I found it challenging to grasp how reshaping tensors worked, especially when dealing with matrix multiplication requirements.
2. GPU vs. CPU computation: The concept of device allocation (cuda vs. cpu) took some time to understand fully.

**DISCUSSION OF IMPROVEMENTS AND LEARNING**

**Personal Growth:** This journey improved my confidence in navigating deep learning tools and interpreting model predictions. It enhanced my ability to work with PyTorch and gave me learning platform to perform tensor operations and using PyTorch’s autograd for differentiation.

**Skills Developed:**

* Understanding of AWS Labs pertinent to Deep Learning
* Understanding basic PyTorch operations, including tensor creation / manipulation, indexing, and mathematical computations.
* Understanding how PyTorch’s autograd automates gradient computation / differentiation
* Basic neural network implementation
* Awareness of GPU acceleration for efficient deep learning model training and optimization concepts

**Future Application:** The knowledge gained from this lab will be invaluable in building and training deep learning models. The ability to manipulate tensors effectively and leverage GPU acceleration will be crucial in future AI projects, particularly in computer vision and natural language processing applications.

**CONCLUSION**

The lab provided a solid foundation in PyTorch fundamentals and neural network implementation. PyTorch is like a smart, adaptable brain-building toolkit:

* Creates number collections easily
* Performs complex mathematical operations
* Learns from mistakes automatically
* Helps build systems that improve with practice

The most valuable takeaway was understanding how PyTorch's automatic differentiation system works, as this forms the backbone of neural network training. It transforms complex mathematical concepts into a playful, learning-oriented process that even non-technical people can understand. This explanation bridges the technical details with accessible analogies, making the lab's complex concepts more digestible for someone without a technical background.

**Application of Deep Learning to Text and Image Data: Module 1, Lab 2**

**Multilayer Perceptron and Dropout Layers**

**INTRODUCTION**

This report documents my learning experience with Lab 2, which focused on implementing a simple neural network with multiple layers and utilizing dropout layers to prevent overfitting. The primary objective of this lab was to understand the architecture of a Multilayer Perceptron (MLP), train the neural network with multiple layers to classify fashion items from the Fashion-MNIST dataset, and observe the effects of adding dropout layers to regularize the model.

**DESCRIPTION OF EXPERIENCE**

The exercise used the Fashion-MNIST dataset, which contains 28x28 pixel grayscale images of clothing items across 10 categories. We worked with PyTorch to implement a multilayer perceptron (MLP) and later enhanced it with dropout layers. The steps included:

1. **Loading the Dataset** – Using PyTorch, the Fashion-MNIST dataset was loaded and transformed into tensors for model processing.
2. **Defining the Model** – A simple neural network with a single dense layer was created to classify images based on their pixel values.
3. **Training the Neural Network** – The model was trained using stochastic gradient descent (SGD) while tracking loss and accuracy.
4. **Adding Dropout Layers** – To improve generalization and prevent overfitting, dropout layers were introduced between layers in an extended MLP model.
5. **Comparing Performance** – The effects of dropout layers on training accuracy, validation accuracy, and overall model performance were analyzed.

**PERSONAL REFLECTION**

**Thoughts and Feelings:** Initially, I found the concept of dropout layers counterintuitive - deliberately "dropping" neurons seemed like it would harm the network's performance. However, seeing the improved results helped me understand its importance in preventing overfitting.

**Analysis and Interpretation:** The lab demonstrated how overfitting occurs when a model learns training data too well but fails to generalize to new data. The dropout technique helped mitigate this issue by forcing the network to be more robust and adaptive. In layman’s terms, I learnt:

**1. Working with Fashion Images (Dataset)**

mnist\_train = torchvision.datasets.FashionMNIST(

root="data", train=True, transform=transforms.ToTensor(), download=True

)

**Simple Explanation**:

* Think of this like getting a huge catalog of clothing pictures
* Each picture is small (28x28 pixels) and in black and white
* Like having 60,000 tiny clothing photos for training
* Similar to how a retail worker learns to categorize clothes by looking at many examples

**2. Creating a Simple Brain (Basic Neural Network)**

mlp = nn.Sequential(

nn.Flatten(),

nn.Linear(in\_features=784, out\_features=10)

)

**Understanding**:

* Like creating a simple clothing sorter
* Input: A clothing image (784 pixels)
* Output: 10 possible clothing types
* Similar to training a new employee to sort clothes into 10 different departments

**3. Testing Without Training**

pred = mlp(data)

print("Predictions:", pred[:10])

print("True labels:", label[:10])

**Simple Explanation**:

* Like asking someone to sort clothes with their eyes closed
* The predictions are random guesses
* Accuracy is about 10% (pure chance)
* Similar to guessing what type of clothing an item is without any prior knowledge

**4. Training the Brain (Model Training)**

train\_losses, train\_accs, val\_accs = train\_net(

mlp, training\_loader, validation\_loader, num\_epochs=25, learning\_rate=0.03

)

**Understanding Progress**:

Epoch 1: train loss 1.035, train acc 0.694

...

Epoch 25: train loss 0.454, train acc 0.848

Think of this like:

* First day at work (Epoch 1): Gets 69% of clothes sorted correctly
* After 25 days of practice (Epoch 25): Gets 85% correct
* Similar to how a person improves with practice

**5. Making it Smarter (Adding Dropout Layers)**

mlp\_dropout = nn.Sequential(

nn.Flatten(),

nn.Linear(784, 784),

nn.ReLU(),

nn.Dropout(0.3),

nn.Linear(784, 256),

nn.ReLU(),

nn.Dropout(0.3),

nn.Linear(256, out\_classes)

)

**Simple Explanation**:

* Like training someone while randomly covering some items
* Forces them to learn multiple ways to identify clothes
* Dropout (0.3) means ignoring 30% of the information temporarily
* Similar to learning to identify clothes by their shape, texture, AND color instead of relying on just one feature

**6. Results Visualization**

plot\_losses(train\_losses, train\_accs, val\_accs)

**Understanding the Graphs**:

1. Loss Values Graph:
   * Like tracking mistakes over time
   * Starting: Many mistakes (high loss)
   * Ending: Fewer mistakes (low loss)
   * Similar to reducing errors in sorting clothes
2. Accuracy Values Graph:
   * Shows improvement in correct predictions
   * Quick improvement at first
   * Slower improvement later
   * Like learning any new skill - fast progress initially, then gradual improvement

Training without dropout is like overfilling a sponge with water—it absorbs too much and leaks. Adding dropout is like squeezing out excess water, leaving a well-balanced model that adapts better to new data.

**Real-World Comparison**

Think of this like training a new employee at a clothing store:

1. Basic Model (Without Dropout):
   * Like learning by memorizing specific examples
   * Gets good quickly but might struggle with new styles
2. Dropout Model:
   * Like learning to identify clothes under different conditions
   * Takes longer to learn but handles unusual cases better
   * More reliable in the long run

**Key Takeaways**

1. Initial Model Performance:

* The untrained model made random guesses (around 10% accuracy)
* After training, accuracy improved to about 83%
* This showed how the network learned to recognize patterns in clothing images and gets better at generalizing with dropout
* Imagine giving a child a multiple-choice quiz without any prior study; their answers would be random guesses.

1. Training Process:

* The model started with high loss (poor predictions)
* Over time, it adjusted its weights to make better predictions
* Like a student practicing, the model got better with each epoch (training round)
* Training without dropout: If you memorize a math book word-for-word, you may do well on the practice tests but struggle with slightly different problems in a real exam.
* Training with dropout: Imagine practicing different routes when learning to drive rather than just memorizing one road. This makes you a more adaptable driver in different traffic conditions.

1. Dropout Layer Impact:

* Adding dropout layers was like forcing the network to "study" with different study groups
* This prevented over-reliance on specific neurons
* The result was a more robust model that could generalize better
* Like learning to identify clothes in different lighting conditions

1. Performance:

* Basic model: Quick learner but might memorize
* Dropout model: Slower learner but more adaptable
* Both reach similar final accuracy (~82-85%)

This progression shows how we can make a computer learn to recognize clothing items, similar to how humans learn but at a much faster rate. The addition of dropout layers is like teaching someone to be more flexible and adaptable in their learning approach.

**Connections to Theoretical Knowledge:** This lab reinforced the importance of regularization techniques in deep learning. The dropout concept aligns with theoretical principles of preventing co-adaptation of neurons and ensuring better generalization of the model. The lab demonstrated several key machine learning concepts:

* Overfitting: When a model performs well on training data but poorly on new data
* Regularization: Techniques (like dropout) to prevent overfitting
* Gradient descent: The optimization process for training neural networks

**Critical Thinking:** The lab revealed some interesting insights:

* The relationship between training and validation accuracy
* The importance of choosing appropriate hyperparameters (learning rate, dropout rate)
* The trade-off between model complexity and performance

**DISCUSSION OF IMPROVEMENTS AND LEARNING**

**Personal Growth:** This lab enhanced my understanding of:

* Neural network architecture design
* The importance of preventing overfitting
* How to interpret training metrics and graphs

**Skills Developed:**

* Implementing neural networks using PyTorch
* Adding and configuring dropout layers
* Analyzing model performance through metrics and visualization

**Future Applications:** This lab will help in:

* Experimenting with different dropout rates to find an optimal balance.
* Implementing batch normalization alongside dropout for improved performance.
* Extending the model to classify more complex datasets beyond Fashion-MNIST.
* Troubleshooting model performance issues.

**CONCLUSION**

This lab provided practical experience in building and improving neural networks. The practical application of dropout layers demonstrated how neural networks can be optimized to generalize better, leading to improved performance in real-world scenarios. The most valuable insights are:

* Overfitting can be mitigated using dropout layers, making the model more resilient to variations in new data.
* Model accuracy does not solely indicate performance; validation accuracy is crucial for assessing generalization.
* Increasing epochs without dropout can lead to unnecessary memorization rather than learning useful features.

**Application of Deep Learning to Text and Image Data: Module 1, Lab 3**

**Building an End-to-End Neural Network Solution**

## **INTRODUCTION**

This lab focused on building a complete neural network solution for processing text data, specifically working with the Austin Animal Center Dataset to predict pet adoption outcomes. The lab demonstrated how to:

* Import and preprocess text data.
* Create a multi-layer neural network.
* Train and validate the model.
* Tune hyperparameters to improve performance.

**DESCRIPTION OF EXPERIENCE**

The lab utilized the **Austin Animal Center Shelter Intakes and Outcomes** dataset, where the goal was to predict whether an animal was adopted (1) or not (0). The process included:

1. **Data Import and Exploration** – Loaded the dataset into a Pandas DataFrame and examined missing values, categorical variables, and numerical features.
2. **Data Preprocessing** –
   * Categorical data was one-hot encoded.
   * Numerical data was normalized.
   * Text features were tokenized and vectorized using CountVectorizer.
   * The dataset was split into training, validation, and test sets.
3. **Neural Network Creation** –
   * A multi-layer perceptron (MLP) was implemented using PyTorch.
   * Two hidden layers were added with **ReLU activation**.
   * Dropout layers were introduced to prevent overfitting.
4. **Training and Evaluation** –
   * The model was trained using the **cross-entropy loss function** and **stochastic gradient descent (SGD)**.
   * Performance was monitored through loss curves and accuracy metrics.

**PERSONAL REFLECTION**

**Thoughts and Feelings:** Initially, I was excited to work with text data in a neural network. However, handling categorical and textual data in machine learning required careful preprocessing, which was challenging but informative.

**Analysis and Interpretation:** The impact of dropout layers was evident, as they prevented overfitting and improved validation accuracy. Additionally, tuning hyperparameters such as learning rate and batch size significantly influenced model performance. Main learning points as per outputs:

1. **Data Processing**

* The lab used the Austin Animal Center Dataset containing pet adoption records
* Features included both numerical data (age), categorical data (pet type, sex), and text data (location, breed)

print("Dataset shape:", df.shape)

df.head()

**Understanding:** The dataset contained **95,485 rows and 13 columns**, including features like breed, age, and intake condition. Some columns had missing values.

**Simple Explanation:** Imagine sorting lost pets by characteristics like age and breed to determine their adoption likelihood.

1. **Text Data Cleaning** The lab demonstrated text processing techniques:

def cleanSentence(text, stop\_words, stemmer):

*# Lowercase and strip whitespace*

text = text.lower().strip()

*# Remove HTML tags*

text = re.compile("<.\*?>").sub("", text)

*# Remove punctuation*

text = re.compile("[%s]" % re.escape(string.punctuation)).sub(" ", text)

*# Remove extra white space*

text = re.sub("\s+", " ", text)

return text

This code standardizes text by converting to lowercase, removing HTML tags, punctuation, and extra spaces.

1. **Neural Network Architecture** The lab implemented a multilayer perceptron using PyTorch:

net = nn.Sequential(

nn.Linear(in\_features=171, out\_features=64),

nn.ReLU(),

nn.Dropout(p=0.3),

nn.Linear(in\_features=64, out\_features=64),

nn.ReLU(),

nn.Dropout(p=0.3),

nn.Linear(in\_features=64, out\_features=10)

)

This created a network with:

* + Two hidden layers with 64 neurons each
  + ReLU activation functions
  + Dropout layers to prevent overfitting

1. **Training Progress: The training output showed:**

Epoch 0. Train\_loss 0.934593 Validation\_loss 0.605567 Seconds 14.224415

Epoch 14. Train\_loss 0.438620 Validation\_loss 0.422772 Seconds 4.528893

**Understanding**:

* The model started with high error (0.93) and improved to lower error (0.44)
* Training took about 4-14 seconds per epoch
* The validation loss also decreased, showing real learning rather than memorization

**Simple Explanation:** It’s like practicing for a quiz—early attempts have more mistakes (higher loss), but over time, you improve and make fewer errors.

1. **Final Model Performance**

precision recall f1-score support

0 0.86 0.71 0.78 6267

1 0.80 0.91 0.85 8053

**Understanding**:

* The model was 86% accurate in identifying pets that wouldn't be adopted (0)
* It was 80% accurate in identifying pets that would be adopted (1)
* Overall accuracy was 83%, which is quite good for this type of prediction

**Simple Explanation:** Imagine predicting if a shelter pet will be adopted based on past records—sometimes you're right, but not always.

**Connections to Theoretical Knowledge:** The lab reinforced concepts such as **vectorization**, **one-hot encoding**, and **dropout regularization**, demonstrating their role in optimizing deep learning models. The goal was to build an end-to-end neural network solution, incorporating all previously learned data processing techniques.

## **Practical Applications and Insights**

1. **Real-world Applications**
   * This type of model could help animal shelters predict which pets might need extra attention or marketing to increase adoption chances
   * The techniques could be applied to other text-based prediction tasks, like customer service or product recommendations
2. **Technical Insights**
   * Text data requires significant preprocessing before it can be used in a neural network
   * Dropout layers helped prevent the model from overfitting
   * The model achieved good balance between precision and recall

## **Areas for Improvement:** The lab suggested several ways to improve the model:

* Adjusting the network architecture (layers, neurons)
* Fine-tuning hyperparameters (learning rate, batch size)
* Experimenting with different activation functions
* Testing various optimization algorithms

**DISCUSSION OF IMPROVEMENTS AND LEARNING**

**Key Takeaways:**

* Cleaning and encoding categorical and text data properly is essential for improving neural network performance.
* Dropout layers help prevent overfitting, making the model more robust.
* Monitoring **both training and validation loss** is crucial to ensuring the model generalizes well.

**Future Applications:**

* Experimenting with **different neural network architectures** (e.g., more layers, batch normalization).
* Testing different **learning rates** and **optimizers** for better performance.
* Extending the model to handle more complex text-processing tasks like **sentiment analysis.**

**CONCLUSION**

This lab provided hands-on experience in handling text data in deep learning. The structured approach—from pre-processing to model tuning—enhanced my understanding of real-world neural network applications. The final model's 83% accuracy shows the effectiveness of the approach while leaving room for potential improvements.

**REFERENCES**

<https://awsacademy.instructure.com/courses/107351>

<https://awsacademy.instructure.com/courses/107351/modules/items/10040563>

<https://awsacademy.instructure.com/courses/107351/modules/items/10040565>

<https://awsacademy.instructure.com/courses/107351/modules/items/10040566>

<https://awsacademy.instructure.com/courses/107351/modules/items/10040569>

<https://awsacademy.instructure.com/courses/107351/modules/items/10040570>

<https://awsacademy.instructure.com/courses/107351/modules/items/10040572>

<https://docs.aws.amazon.com/>

[https://pytorch.org/docs/stable/index.html#](https://pytorch.org/docs/stable/index.html)

<https://awsacademy.labs.awsevents.com/sa/lab/arn%3Aaws%3Alearningcontent%3Aus-east-1%3A470679935125%3Ablueprintversion%2FCUR-TF-200-MLUDTI-1%2F01-M1-Lab1_PyTorch%3A1.0.3-347dd832/en-US>

<https://awsacademy.labs.awsevents.com/sa/lab/arn%3Aaws%3Alearningcontent%3Aus-east-1%3A470679935125%3Ablueprintversion%2FCUR-TF-200-MLUDTI-1%2F02-M1-Lab2_Perceptron-Dropout%3A1.0.3-f4486be9/en-US>

<https://awsacademy.labs.awsevents.com/sa/lab/arn%3Aaws%3Alearningcontent%3Aus-east-1%3A470679935125%3Ablueprintversion%2FCUR-TF-200-MLUDTI-1%2F03-M1-Lab3_NN%3A1.0.3-8030ee2f/en-US>

<https://eagleonline.hccs.edu/courses/282423/files/71221184?module_item_id=19170181>