Midterm Summary

Samiksha P S
24B1258
Build Your Own GPT
Project UID 143

Introduction:

This report provides a clear overview of the concepts and practical applications in Natural Language Processing (NLP) using Python and PyTorch. The learning path covered important text cleaning techniques, traditional vectorization methods, semantic embeddings, and deep learning models. I focused on understanding both the theoretical basis and practical applications through hands-on projects. Significant milestones included creating a system for semantic similarity retrieval using Word2Vec and a sentiment analysis model using BERT embeddings with an LSTM classifier. This document summarizes the skills, tools, and models explored during my learning journey.

What I have learnt so far:

Here's a breakdown of the roadmap I followed:

1. Text Preprocessing (Cleaning the Text)

■ Tokenization

■ Stopword removal

■ Stemming (Porter, Snowball)

■ Lemmatization (WordNet lemmatizer)

■ Named Entity Recognition (NER)

■ Parts of Speech (POS) tagging

2. Converting Text into Vectors – Traditional Approaches

■ One-Hot Encoding

■ Bag of Words (BoW) — normal and binary

■ N-grams (unigrams, bigrams, trigrams)

■ TF-IDF (Term Frequency-Inverse Document Frequency)

3. Word Embeddings (Semantic Meaning)

■ Word2Vec — CBOW and Skip-gram

■ Average Word2Vec

■ Cosine similarity for semantic similarity

■ Used for implementation

4. Text Data Vocabulary Structure

■ Corpus → Documents → Sentences → Words

■ Vocabulary (unique words), Tokens, and semantic information

5. Stemming vs Lemmatization

■ The main difference: stemming cuts off suffixes, while lemmatization considers grammar and meaning.

■ Used the Snowball stemmer for better linguistic accuracy than Porter.

6. Differences Between NLTK and spaCy

■ NLTK: Research-focused, modular, great for learning

■ spaCy: Industrial-strength, fast, production-ready

7. Shortcomings of Traditional Encoding Methods

■ Sparse matrices

■ No semantic understanding

■ Out-of-vocabulary issues

■ Fixed input length challenges

8. Introduction to Neural NLP

■ RNN, LSTM, GRU for handling sequential data

■ Transformers and BERT for efficient and parallel modeling

9. Word2Vec Details

■ Implemented both CBOW and Skipgram

■ Learned how window size affects vector representation

■ Understood vector dimensions and how average Word2Vec creates fixed-length representations

10. Artificial Neural Networks

■ Understood the structure of neural networks: input layer, hidden layers, output

■ How weights, bias, and activation functions (like sigmoid) work

■ Backpropagation and the role of the loss function

■ Optimizers for updating weights and biases during training

■ Used one-hot encoding for inputs and trained small Word2Vec models with simple neural networks

11. Tensors in PyTorch (Tensors_in_PyTorch.ipynb)

■ Understood the main data structure in PyTorch — Tensor

■ Learned how to create tensors of different shapes and types

■ Practiced basic tensor operations like addition, reshaping, slicing, and matrix multiplication

■ Explored interoperability with NumPy arrays (converting between NumPy and tensors)

■ Worked with gradients using requires_grad and observed how automatic differentiation operates

12. Dataset and Dataloader (Dataset_And_Dataloader.ipynb)

■ Learned the importance of the Dataset and DataLoader classes for model training

■ Built a custom dataset class by subclassing torch.utils.data.Dataset

■ Used DataLoader to batch and shuffle data, preparing it for efficient iteration during training

■ Understood how to loop through batches using basic for-loops

■ Covered essential preprocessing for providing structured input to a model

13. PyTorch Training Pipeline (Basic) (Pytorch_Training_Pipeline.ipynb)

■ Built a basic model training loop from scratch

■ Practiced all key training steps: forward pass, loss computation, backpropagation, and optimization

■ Used a simple model to reinforce the concept of epochs, batch updates, and model convergence

■ Learned the structure of a basic PyTorch training loop without using higher abstractions

14. Training Pipeline with Dataset and Dataloader (Pytorch_Training_Pipeline_Using_Ddataset_And_Dataloader.ipynb)

■ Combined Dataset and DataLoader into a complete training loop

■ Managed batch-wise training in an organized and scalable way

■ Made the training code cleaner and modular using PyTorch’s built-in components

■ Practiced loss tracking throughout epochs and visualizing batches

15. Training Pipeline with nn.Module (Pytorch_Training_Pipeline_Using_nn_Module.ipynb)

■ Learned how to define neural networks using nn.Module

■ Shifted from manually defined weight updates to PyTorch’s components

■ Created a custom model class with layers and a forward method

■ Used optimizers like SGD or Adam along with standard loss functions (e.g., nn.CrossEntropyLoss)

■ Understood why nn.Module makes deep learning models more flexible and clearer

16. ANN on Fashion MNIST Dataset (ANN_Fashion_MNIST_pytorch.ipynb)

■ Implemented a complete feedforward neural network for image classification

■ Worked with the Fashion MNIST dataset from torchvision.datasets

■ Normalized data and applied ToTensor transformations

■ Built a multi-layer structure, trained it, and evaluated it using accuracy metrics

■ Learned how to work with real datasets and identify performance bottlenecks

17. Model Using nn.Module (Pytorch_NN_Module.ipynb)

■ Gained insight into using nn.Sequential vs defining a custom class

■ Explored stacking layers, applying activations (e.g., ReLU), and using dropout for regularization

■ Implemented a modular training pipeline using these concepts

■ Tracked training metrics over multiple epochs

■ Reinforced the role of hyperparameters like learning rate and batch size

18. RNN-Based QA System (Pytorch_RNN_Based_QA_System.ipynb)

■ Learned how to process sequential data with Recurrent Neural Networks

■ Used RNNs for basic question-answering tasks or similar text classification

- ■ Managed variable-length sequences using padding
- ■ Gained practical experience in implementing recurrent layers like LSTM or standard RNNs
- ■ Learned how to pass hidden states through time and train the model end-to-end

Milestones Completed:

1. Milestone1: Top 5 Similar Sentences Finder

In Milestone 1, I developed a semantic similarity system using Word2Vec embeddings to find the top 5 most similar sentences to a given input query. The pipeline began by loading a dataset of 10,000 unique news headlines in JSON format. Preprocessing involved tokenization with NLTK, stopword removal, lowercasing, and lemmatization using the WordNet lemmatizer. A Word2Vec model was trained with gensim on the tokenized corpus. I created sentence representations by averaging the word vectors for each sentence (AvgWord2Vec). Then, I calculated cosine similarity between the vector of the input query and the sentence vectors in the corpus using sklearn.metrics.pairwise.cosine_similarity. I identified the top five similar sentences by sorting similarity scores in descending order. This implementation was done using Python in Google Colab and relied on libraries like NLTK, Gensim, NumPy, and scikit-learn.

2. Milestone 2+3: Sentiment Analysis Using LSTM

In Milestone 2 and 3, I built a binary sentiment classification model using a hybrid architecture that combines BERT embeddings with an LSTM-based classifier. I loaded a dataset of 10,000 movie reviews and split it into training and testing subsets in an 80:20 ratio. The bert-base-uncased tokenizer was used to tokenize text, apply padding and truncation, and insert special tokens ([CLS], [SEP]). I extracted pre-trained embeddings from the frozen bert-base-uncased model using the Hugging Face Transformers library without fine-tuning. I implemented a custom torch.utils.data.Dataset subclass (MovieReviewDataset) to manage tokenized inputs, attention masks, and labels.

The model architecture included an embedding layer (using BERT outputs), an LSTM layer, a dropout layer for regularization, and a fully connected output layer for binary classification. I trained the model with torch.nn.BCELoss (binary cross-entropy) and optimized it using the Adam optimizer. I created a training loop with forward pass, loss computation, backpropagation, and parameter updates using torch.nn modules. I tracked model performance over epochs with metrics such as accuracy and training loss. I visualized training and validation progress using matplotlib. The implementation was executed on GPU (when available) through PyTorch's device management (cuda/cpu) for efficient computation.