Generative AI, Assignment # 1

Department of Computer Science
National University of Computer and Emerging Sciences,
Islamabad, Pakistan

Due Date: March 04, 2025

Instructions

- Each student must submit the following three files packaged into a single ZIP file and named as **ROLLNO_NAME.ZIP**:
  - A Jupyter Notebook (`.ipynb`) or Python script (`.py`) with the complete implementation.
  - A PDF report written in LaTeX using Overleaf, following the Springer's LNCS paper format:
    Springer LNCS Template on Overleaf.
  - A plain text file (`.txt`) containing all the GPT prompts used for each question.
- Ensure that the code is well-structured with proper comments for each function. Include all necessary dependencies to ensure the code runs without errors.

1 Question 1: Implementing Rosenblatt's Perceptron from Scratch

1.1 Objective

This task aims to implement a single-layer perceptron model from scratch, ensuring a complete understanding of forward propagation, backward propagation, and weight updates. You will also generate a dataset, visualize it, and evaluate the model.

1.2 Instructions

You must manually implement the core perceptron functions as discussed in the class without using deep learning frameworks. However, you may use libraries such as NumPy and Matplotlib for other necessary operations like data handling, visualization, and matrix operations.

1. **Data Generation and Visualization**
   - Generate a synthetic dataset containing 500 samples with **two features** and a **binary label**.
   - Visualize the dataset by plotting data points in a two-dimensional space.
   - Split the dataset into training and testing subsets (e.g., 80% training, 20% testing).
2. **Perceptron Implementation**
   - Implement the following core functions manually (your own implementation without using any library, you must store weights and biases in matrices):
     - *Forward Pass*: Compute the weighted sum of inputs and apply an activation function (step function).
     - *Backward Pass*: Calculate the error and then apply the backpropagation to update weights using the perceptron learning rule.
   - Repeat the experiment and display the error for each iteration.
3. **Visualization and Evaluation**
   - Plot the decision boundary after training is completed.
   - Visualize the test dataset by plotting all test data points on the decision boundary.
   - Discuss how well the perceptron classifies the test data.

2 Question 2: Implementing Convolution from Scratch

2.1 Objective

The objective of this task is to develop a deeper understanding of convolution operations by implementing them manually without relying on built-in deep learning functions. This assignment will explore how convolution operations affect images and how different kernels influence output.

2.2 Instructions

1. **Implement a Generalized Convolution Function**
   - Write a function that performs **manual 2D convolution** on a grayscale image. You can manually design all loops needed without any use of libraries for the core convolution operation.
   - The function should accept the following parameters:
     - **Input image**: The grayscale image to be processed.
     - **Kernel**: A user-defined kernel (default to a random kernel if none is provided).
     - **Kernel size**: The size of the kernel matrix.
     - **Stride**: The step size for sliding the kernel.
     - **Padding**: Option to use "valid" (no padding) or "same" (zero-padding to maintain size).
     - **Mode**: Option to perform either **convolution** or **correlation**.
   - All are optional parameters except Input Image; if no other parameter is provided, it should set all of them by default.
2. **Perform Convolution with Specific Kernels**
   - Call your previously defined convolution function to a grayscale image using different kernels for various purposes such as edge detection, blur images, and sharpening images. Compare the outputs of these different kernels and analyze how they affect the image.
3. **Compare Convolution vs Correlation**
   - Test two different kernels:
     - A **symmetric kernel**, such as:

$$\begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$

     - A **non-symmetric kernel**, such as:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

   - Compare the output results of convolution and correlation operations on the same image and analyze the differences.
4. **Visualization and Analysis**
   - Display the original image.
   - Show the output images for each of the applied kernels.

- Compare side-by-side:
  - Manually implemented convolution vs. NumPy-based convolution.
  - Convolution vs. Correlation for the chosen kernels.
- Write a detailed description of your experiments in your report, specifically analyzing the following:
  - The effect of different kernels on the image.
  - The impact of kernel size, stride, and padding on the output.
  - Observations from convolution vs. correlation results.
  - Perform some analysis to identify some kernels that are useful for edge detection.
  - Perform some analysis to identify kernels that can blur images.
  - Perform some analysis to identify kernels that can sharpen images.
  - Explain if you apply multiple kernels on the same image what will be the advantage, describe it based on the observations you made from experiments.

3 Question 3: Implementing a CNN for CIFAR-10

Note: For this question, you can use any libraries you wish.

3.1 Objective

The objective of this task is to build a Convolutional Neural Network (CNN) for image classification using the CIFAR-10 dataset. Students will explore CNN architecture, feature extraction, and model evaluation by implementing and training a CNN model using deep learning frameworks.

3.2 Dataset

For this task, use the CIFAR-10 dataset available at:
Hugging Face - CIFAR-10 Dataset

CIFAR-10 consists of 60,000 images (32x32 pixels, RGB, 10 classes), making it a suitable dataset for CNN training.

3.3 Instructions

1. **Dataset Preparation**
   - Load the CIFAR-10 dataset from Hugging Face.
   - Preprocess the images:

    - Normalize pixel values (scale between 0 and 1).
    - Convert labels into one-hot encoded format.
    - Split the dataset into training (80%) and testing (20%) subsets.

2. **Implement a CNN Classifier**
    - Build a Convolutional Neural Network using a deep learning framework (TensorFlow/Keras or PyTorch).
    - The CNN architecture should include:
        - **Convolutional Layers**: Extract features from images.
        - **ReLU Activation Function**: Introduce non-linearity.
        - **Pooling Layers (Max/Average)**: Reduce spatial dimensions.
        - **Fully Connected Layers**: Learn complex patterns for classification.
        - **Softmax Output Layer**: Generate class probabilities.
    - Train the model using an appropriate optimizer (e.g., Adam, SGD).

3. **Evaluate and Compare Model Performance**
    - Evaluate model accuracy on test data.
    - Perform data augmentation (e.g., flipping, rotation) and analyze its impact.
    - Compare:
        - Model trained **without augmentation**.
        - Model trained **with augmentation**.
    - Visualize the loss and accuracy curves.

4. **Feature Map Visualization**
    - Extract and visualize feature maps from different layers.
    - Show how early layers capture edges, while deeper layers capture high-level features.

5. **Ablation Study: Impact of Hyperparameters on Accuracy**
   To better understand the effect of different hyperparameters on CNN performance, conduct an ablation study by modifying the following four hyperparameters and observing their impact on accuracy:
    - **Learning Rate**: Experiment with at least three different learning rates (e.g., 0.001, 0.01, 0.1) and analyze how they affect model convergence and accuracy.
    - **Batch Size**: Train the model with different batch sizes (e.g., 16, 32, 64) and compare how it influences training time and performance.

- **Number of Convolutional Filters**: Vary the number of filters in the convolutional layers (e.g., 16, 32, 64) and observe the effect on feature extraction and accuracy.
- **Number of Layers**: Modify the number of convolutional layers in the model (e.g., 3, 5, 7 layers) and compare how deeper models perform compared to shallower ones.

3.4 Evaluation and Comparison of Model Performance

3.4.1 Performance Metrics

To assess and compare the CNN models, evaluate the following metrics: Accuracy,Precision , Recall, F1-Score, Confusion Matrix

3.4.2 Comparison of Models

Train and evaluate two models:

1. **Model without Data Augmentation**
2. **Model with Data Augmentation** (e.g., flipping, rotation, shifting)

For both models, compute the above metrics and present the results in a tabular format:

Table 1: Performance Metrics Comparison of CNN Models

Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Without Augmentation	[Value]	[Value]	[Value]	[Value]
With Augmentation	[Value]	[Value]	[Value]	[Value]

3.4.3 Confusion Matrix Visualization

For both models, generate and visualize the confusion matrix to analyze the classification performance:

- Plot the confusion matrix as a heatmap.
- Analyze misclassified categories.

3.4.4 Loss and Accuracy Curves

- Plot training and validation loss over epochs to analyze convergence.
- Plot training and validation accuracy over epochs to evaluate overfitting or underfitting.
- Compare curves for both models and discuss key observations.

4 Question 4: Implementing a Vanilla RNN for Next-Word Prediction

Note: This is a research-oriented question, that you need to do some additional research for better understanding and implementation of some parts. You can use any libraries of your own choice.

4.1 Objective

This task aims to train a Vanilla Recurrent Neural Network (RNN) on a Shakespeare text dataset from Hugging Face for next-word prediction. Instead of using pre-trained embeddings (like Word2Vec or GloVe), students will train their own word embeddings using an Embedding Layer in TensorFlow or PyTorch.

4.2 Dataset

Use a publicly available Shakespeare text dataset from Hugging Face: Hugging Face - Shakespeare Dataset

4.3 Implementation Steps

1. **Load and Preprocess the Dataset**
    - Load the Shakespeare dataset from Hugging Face and then Tokenize words to create a vocabulary.
    - Split the dataset into training (80%) and testing (20%).
2. **Implement the Vanilla RNN Model**
    - Implement a custom RNN cell (no LSTMs or GRUs). You can use Python Classes for implementation of these layers.
    - Use a trainable Embedding Layer in TensorFlow/Keras or PyTorch to learn word representations. Set the suitable embedding size.
    - The model should process word sequences and predict the next word.
    - Use Cross-Entropy Loss and an appropriate optimizer (e.g., Adam).
3. **Train the Model and Monitor Performance**
    - Train the model using Backpropagation Through Time (BPTT).
    - Monitor training loss and validation loss across epochs.
    - Save the trained model.
4. **Generate Text Predictions**
    - Provide a seed phrase (e.g., “To be or not to”).
    - The model should generate the next word iteratively.

- Generate at least 10 words to form a complete sentence.

5. **Evaluate Model Performance**
   - Compute and report the following metrics:
     - Perplexity (Measures model uncertainty)
     - Word-level accuracy
     - Loss curve visualization
   - Compare learned embeddings with randomly initialized ones.

6. **Ablation Studies**
   - Train the RNN model using pretrained word embeddings (Word2Vec or GloVe) and compare its performance with the model trained using randomly initialized embeddings.
   - Evaluate the model using the following metrics:
     - Perplexity (Measures model uncertainty)
     - Word-level accuracy
     - Loss curve visualization
   - Plot a confusion matrix showing misclassified words.
   - Analyze and discuss the impact of using pre-trained embeddings on the model's performance.

4.4 Expected Output

- A table comparing learned vs. random embeddings:

Table 2: Comparison of Word-Level Accuracy and Perplexity for Different Embeddings

Embedding Type	**Word-Level Accuracy**	**Perplexity**
Random Embeddings	[Value]	[Value]
Learned Embeddings	[Value]	[Value]

- Generated text sequences from the model.

5 Question 5: Hyperparameter Search for CNN and RNN

- Implement hyperparameter search for both CNN and RNN models using the Random Search technique.
- Define a set of hyperparameters to search over, including:

  - Learning rate
  - Number of layers
  - Number of neurons (for RNN) or filters (for CNN)
  - Batch size
  - Optimizer (e.g., Adam, SGD, RMSprop)
  - Activation functions (e.g., ReLU, Tanh, Sigmoid)
  - Dropout rate
  - Kernel size (for CNN)
  - Stride (for CNN)
  - Weight initialization method (e.g., Xavier, He Normal)

- Use RandomizedSearchCV from Scikit-Learn or a custom random sampling approach.
- Train multiple models with different hyperparameter combinations and select the best-performing configuration based on validation accuracy.
- Test the best hyperparameter configurations for CNN and RNN models on the test dataset used in the previous question and compare their performance in terms of the evaluation metrics employed.