NLP Assignment -1

Max Marks: 10
Deadline: 25/02/2025

Assignment: Feedforward Neural Network (FFNN) for MNIST Classification

Note: blue color font text is to be answered by students in the report.

Part 1: Implementing the Basic Structure of FFNN

Objective

In this assignment, you will build a **Feedforward Neural Network (FFNN)** from scratch, allowing for multiple layers. You will initialize network parameters dynamically, implement forward propagation, and set up gradient descent for learning.

This assignment will help you develop a deeper understanding of **neural networks** by coding each step yourself rather than relying on pre-built libraries like TensorFlow or PyTorch.

1. **Initialize the Network:**

Write a Python function **initialize_parameters(layer_sizes)** that initializes the weights and biases for each layer of the FFNN.

Input: layer_sizes: A list of integers representing the number of neurons in each layer (e.g., [784, 128, 10] for input, hidden, and output layers).

Output: parameters: A dictionary containing initialized weights and biases for each layer. Weights should be initialized with small random values (e.g., from a normal distribution with mean 0 and standard deviation 0.01). Biases should be initialized to zero.

Example: parameters = initialize_parameters([784, 128, 10])

2. Implement Forward Propagation:

Write a function forward_propagation(X, parameters, activations) that computes the output of the network for a given input X.

Input:

- X: Input data (numpy array of shape (784, m), where m is the number of examples).
- parameters: Dictionary containing weights and biases.

- activations: List of activation functions for each layer (e.g., ["sigmoid", "softmax"]).

Output:

- A: Network output (numpy array of shape (10, m)).
- caches: List of dictionaries containing intermediate values (e.g., activations) needed for backpropagation.

caches: A list of dictionaries containing intermediate values (e.g., activations) needed for backpropagation.

Details:

- Iterate through each layer and compute its output using the appropriate activation function.
- Use the sigmoid activation function for hidden layers and the softmax function for the output layer.

3. **Compute the Loss:**

- Implement the cross-entropy loss function:
- Write a function compute_loss(y, A) that computes the loss given the true labels `y` and the predicted probabilities A.

Input:

- y: True labels (one-hot encoded).
- A: Predicted probabilities.

Output: loss: Cross-entropy loss.

4. Implement Backpropagation:

Write a function backward_propagation(X, y, parameters, caches, activations) that computes the gradients of the loss with respect to the weights and biases for each layer.

Input:

- X: Input data.
- y: True labels (one-hot encoded).
- parameters: Dictionary containing weights and biases.
- caches: List of dictionaries containing intermediate values from forward propagation.
- activations: List of activation functions.

Output: grads: Dictionary containing gradients of the loss with respect to weights and biases for each layer.

Details:

Iterate through each layer in reverse order and compute its gradients using the appropriate activation function derivatives.

5. Update Parameters:

Implement gradient descent to update the weights and biases:

$$W^{[l]} = W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial L}{\partial b^{[l]}}$$

Where alpha is the learning rate (e.g., 0.01). And *l* is the *l*th layer

Write a function update_parameters(parameters, grads, learning_rate) that updates the weights and biases for each layer using the computed gradients.

Input:

- parameters: Dictionary containing weights and biases.
- grads: Dictionary containing gradients.
- learning_rate: Learning rate (e.g., 0.01).

Output: parameters: Updated dictionary of weights and biases.

6. Training Loop:

- Create a training loop that iterates through the dataset for a specified number of epochs.
- Perform forward propagation, compute the loss, perform backpropagation, and update the parameters.
- Track the loss and accuracy during training to monitor progress. Also plot the loss and accuracy curve for every epoch.

Example:

```
for epoch in range(num_epochs):
```

```
# Forward propagation
A, caches = forward_propagation(X_train, parameters, activations)

# Compute loss
loss = compute_loss(y_train, A)

# Backward propagation
grads = backward_propagation(X_train, y_train, parameters, caches, activations)

# Update parameters
parameters = update_parameters(parameters, grads, learning_rate)

# Print loss and accuracy (optional)
if epoch % 10 == 0:
    accuracy = compute_accuracy(y_train, A)
    print(f"Epoch {epoch}: Loss = {loss}, Accuracy = {accuracy}")
```

7. Evaluate the Model:

Write a function evaluate_model(X_test, y_test, parameters, activations) that computes the accuracy of the model on a test dataset.

Function: evaluate_model(X_test, y_test, parameters, activations)

Input:

- X_test: Test data.
- y_test: True labels for test data (one-hot encoded).
- parameters: Dictionary containing trained weights and biases.
- activations: List of activation functions.

Output: accuracy: Accuracy of the model on the test data.

Details:

Use the trained model to make predictions on the test data. Compare the predictions with the true labels to compute accuracy.

PART 2: Answer the following questions

Tasks 1: Activation Function

1. **Construct a 5-Layer FFNN:**
    - Modify your existing FFNN to have 5 layers, with the following structure:
        - Input Layer: 784 neurons
        - Hidden Layer 1: 256 neurons
        - Hidden Layer 2: 128 neurons
        - Hidden Layer 3: 64 neurons
        - Hidden Layer 4: 32 neurons
        - Output Layer: 10 neurons (softmax activation for classification)
    - The activation function for hidden layers should be adjustable (Sigmoid, ReLU, or Tanh).
    - Ensure your implementation supports easy modification of activation functions per layer.
2. **Train the 5-Layer FFNN:**
    - Use the same training setup as in Part 1 (cross-entropy loss, gradient descent, multiple epochs).
    - Train the model with different activation functions (Sigmoid, ReLU, Tanh) and store the activations at each layer.
3. **Analyze Activation Distributions:**
    - After training, plot histograms of the activation values for each hidden layer.
    - Compare how the distribution of activations changes with different activation functions.
    - Suggested steps:
        - Collect activations from forward propagation.
        - Plot histograms for each hidden layer.
        - Compare histograms across different activation functions.
    - Identify which activation function results in a good spread of activation values (avoiding saturation or dead neurons).
    - Discuss:
        - Which activation function retains good variation across layers?
        - Which function causes vanishing or exploding gradients?
        - Which activation function helps deeper networks train more effectively?
4. **Compare Training Performance:**
    - Plot the loss curve and test accuracy graph for different activation functions.

Visualization:

Loss curve

X-axis: Epoch number.
Y-axis: Training loss.

Accuracy Graph:

X-axis: activation function

Y-axis: accuracy value

- Analyze which activation function helps the model converge faster and generalize better.
- Explain why certain activations work better for deeper networks.

Task 2: Implement Batch Normalization and Analyze Its Effects

Batch normalization stabilizes learning by normalizing activations, leading to faster convergence and better generalization. In this task, you will implement batch normalization and analyze how it impacts gradient flow, activation distributions, and model accuracy.

Steps to Complete the Task:

1. Implement Batch Normalization

- Modify the **forward propagation** function to include batch normalization after each layer (except the output layer).
- Normalize the activations for each layer using the batch mean and variance:

$$\hat{z}^{(l)} = \frac{z^{(l)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where:

- μ is the mini-batch mean.
- σ^2 is the mini-batch variance.
- ϵ is a small constant for numerical stability.

Scale and shift the normalized activations with trainable parameters γ and β:

$$a^{(l)} = \gamma \hat{z}^{(l)} + \beta$$

- Modify the **backpropagation** function to compute gradients for γ and β.

2. Analyze the Impact on Activation Distributions

- Train the network **with and without batch normalization**.
- **Plot histograms of activation values** for different layers before and after applying batch normalization.

Visualization:

- **X-axis:** Activation values.
- **Y-axis:** Frequency count.
- **Observation:** A well-balanced activation distribution should be centered around zero, avoiding saturation at extreme values (0 or 1 for sigmoid, negative values for ReLU).

3. Study Its Effect on Gradient Flow

- Compute and **plot the gradient magnitudes** of different layers for models trained with and without batch normalization. Write your observations.

Visualization:

- **X-axis:** Layer index (1st hidden layer → Output layer).
- **Y-axis:** Mean gradient magnitude.
- Compare the gradients at different depths:
    - If gradients shrink exponentially as we go deeper, it indicates a **vanishing gradient problem**.
    - If gradients explode for deeper layers, it indicates an **unstable training process**.
    - Batch normalization should help maintain gradient magnitudes across layers.

4. Evaluate the Impact on Convergence Speed and Model Accuracy

- Train both models (with and without **batch** normalization) for **a fixed number of epochs**.

Plot the training loss over epochs to compare convergence speed.

Visualization:

- **X-axis:** Epoch number.
- **Y-axis:** Training loss.
- Compare how many epochs each model takes to reach 95% accuracy.

Plot the final test accuracy for both models.
Visualization:

- **X-axis:** Model type (With vs. Without Batch Norm).
- **Y-axis:** Test accuracy.

Questions (based on the experiments conducted by you)

- How does batch normalization affect activation saturation?

- Does batch normalization improve gradient flow? Why?
- How much faster does training converge with batch normalization?
- Are there cases where batch normalization might **not** be beneficial?

Task 3: Experiment with Deeper Networks and Analyze Performance Trends

Adding more layers allows for learning complex patterns, but it introduces challenges like vanishing gradients and overfitting.

Steps to Complete the Task:

1. Train FFNNs with Different Depths

- Train networks with **3, 5, 7, and 10 layers**.
- Keep **hyperparameters constant** (learning rate, batch size, activation functions) for a fair comparison.

2. Compare Activation Distributions Across Different Depths

- **Plot activation histograms** for each layer at the start and end of training.

Visualization:

- **X-axis:** Activation values.
- **Y-axis:** Frequency.

3. Compare Training Stability and Gradient Flow

- **Plot gradient magnitudes** for different layers.

Visualization:

- **X-axis:** Layer depth.
- **Y-axis:** Gradient magnitude.

4. Evaluate Convergence Speed and Final Accuracy

Plot training loss curves for networks of different depths.

Visualization:

- **X-axis:** Epoch number.
- **Y-axis:** Training loss.

Plot test accuracy for each network depth.

Visualization:

- **X-axis:** Network depth.
- **Y-axis:** Test accuracy.

Questions (based on the experiments conducted by you)

- At what point does adding more layers **stop improving** performance (you may have to add layers more than 10)?
- How does increasing network depth affect vanishing/exploding gradients?
- How does adding more layers impact convergence speed?

Task 4: Compare Performance Using Different Weight Initialization Techniques

Weight initialization greatly impacts training stability. In this task, you will compare **Random, Xavier, and He** initialization.

Steps to Complete the Task:

1. Train Networks Using Different Initializations (write the code for following)

- **Random initialization:** Small normal distribution values.

- **Xavier initialization:**
  - Works well for sigmoid/tanh activations.
- **He initialization:**
  - Optimized for ReLU activations.

2. Compare Activation Distributions

Plot activation histograms at the start of training.

Visualization:

- **X-axis:** Activation values.
- **Y-axis:** Frequency count.
- **Observation:** Poor initialization can cause activations to saturate (e.g., all neurons output 0 for sigmoid).

3. Compare Gradient Flow

Plot gradient magnitudes across layers.

Visualization:

- **X-axis:** Layer index.
- **Y-axis:** Mean gradient magnitude.
- **Observation:** Poor initialization may cause gradients to **vanish or explode**.

4. Evaluate Convergence Speed and Final Accuracy

Plot training loss curves for each initialization method.

Visualization:

- **X-axis:** Epoch number.
- **Y-axis:** Training loss.
- **Observation:** Xavier and He initialization should **converge faster** than random initialization.

Plot test accuracy for each initialization method.

Visualization:

- **X-axis:** Initialization method.
- **Y-axis:** Test accuracy.
- **Observation:** He initialization works best for ReLU, while Xavier is better for sigmoid/tanh.

Questions (based on the experiments conducted by you)

- Which initialization method leads to **faster training**?
- How does the choice of activation function influence which initialization works best?

Final Instructions:

1. **Submission Format**:
   Ensure that your final submission contains the following:
   - Python code files for all tasks and parts of the assignment (we prefer python notebook).
   - A report (in PDF or Word format) documenting your results, and analysis for each part of the assignment.
   - Visualizations (graphs, histograms, loss curves, etc.) should be properly labeled with appropriate titles and axis labels. Make sure these are embedded in your report.
2. **Code Quality**:
   - Follow best coding practices (e.g., proper naming conventions, comments, and function documentation).
   - Keep your code clean by removing unnecessary print statements or unused code.

3. **Academic Integrity**:

    - Make sure that your solutions are your own work. Any form of plagiarism, including copying code or answers, will result in a loss of marks or potential disqualification from submitting future assignments.
    - Cite any external sources or libraries that you use beyond the basic Python libraries.
4. **Evaluation Criteria**:
   The assignment will be evaluated based on the following criteria:
    - **Correctness** of your implementation for each task.
    - **Clarity** and **depth** of your analysis in the report, including comparisons and discussions on different methods.
    - **Visualizations**: Make sure your plots are clear and effectively convey the insights you are analyzing.
    - **Code Efficiency**: The code should be efficient and easy to follow.
5. **Deadline**:
   The assignment is due by **25th February 2025**. Late submissions will incur a **penalty** unless an extension has been granted due to valid reasons. Ensure to upload your final submission on time.

6. **Help and Clarifications**:
   You may discuss the theoretical concepts with your peers, but the implementation and the analysis should be done independently.

Good luck with your assignment! Make sure to experiment, explore different architectures and methods, and gain a deeper understanding of neural networks!