CMPE 49T
Homework 2

November 15, 2024

1 Introduction

Pneumonia is a significant respiratory disease, often detectable through chest X-ray imaging. In this project, we aim to develop a neural network-based classification model for the detection of pneumonia in X-ray images. Utilizing the PneumoniaMNIST dataset, subset of the MedMNIST dataset.

Dataset source: https://github.com/MedMNIST/MedMNIST. The PneumoniaMNIST dataset, part of the MedMNIST collection, is specifically designed for lightweight experimentation with medical imaging. This dataset consists of grayscale images (28x28 pixels) categorized as either normal or pneumonia.

- **Dataset Source:** *MedMNIST PneumoniaMNIST* [1]
- **Image Size:** 28x28 pixels (grayscale)
- **Classes:** Binary labels – 0 (Normal) and 1 (Pneumonia)
- **Pre-split Subsets:** Training, Validation, and Test sets

The dataset's compact size and manageable image resolution make it ideal for rapid experimentation with neural network models. The dataset is already split into train/validation/test structure with labels included. Read through the github page for further instructions to how to get these informations.

2 Exploratory Data Analysis (EDA)

1. **Class Distribution**: Calculate and display the class distribution in your training, validation, and test sets. Note any imbalance, and be prepared to handle this during training using class weights.

2. **Sample Visualization**: Visualize several examples from **both** classes to understand the differences in appearance. Discuss on what you see.

[1] `https://zenodo.org/record/5208230`

3. **Pixel Intensity Distribution**: Plot the pixel intensity distribution for each class (Normal and Pneumonia) in each dataset to comment any differences. A sample histogram is shown below:

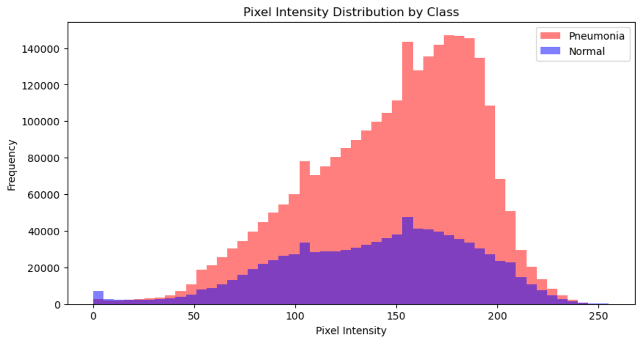

Figure 1: Pixel Intensity Distribution by Class

4. **Calculate class weights**: for training data, you can use this information later on while training the NN for adjusting the weights if you want.

5. **Data Normalization** : Normalize the pixel values to a range of [0, 1] by dividing each pixel by 255.0. This ensures consistent input scaling.

Function to Implement:

```
def load_and_preprocess_data(data_dir):
    # Implement data loading, normalization, and optional augmentation
```

3 Network Structure

The neural network architecture for this assignment is structured as follows:

- **Input Layer**: Flatten the 28x28x1 grayscale image (with Normalized pixel values)
- **First Hidden Layer**:
  - **Convolution Layer**: Applies a 3x3 kernel to extract feature representations from the input.
  - **Activation**: Pick one: ReLU or Sigmoid activation, specified as a parameter.
  - **Pooling**: Apply max pooling with a 2x2 pool size to reduce dimensionality.

- **Second Hidden Layer**:
  - **Convolution Layer**: Use the other 3x3 kernel to further extract features.
  - **Activation**: Choose a different activation function, specified as a parameter.
  - **Pooling**: Max pooling with a 2x2 pool size.
- **Flatten Layer**: Flatten the output from the second hidden layer into a 1D vector to prepare it for the fully connected layer.
- **Fully Connected Layer (Dense Layer)**: Connect the flattened output to a dense layer with a small number of neurons (e.g., 16 or 32). Follow this layer with a dropout layer for regularization to reduce overfitting.
- **Output Layer**: A single neuron with sigmoid activation to perform binary classification, outputting a probability indicating the likelihood of the "Pneumonia" class.

***** It is important to use a different activation function of your choice and a different kernel (filter)for each layer mentioned above.**

3.1 Some Useful Tips

- **reset parameters**: You might want to implement a reset parameters method to make sure that each training session starts from scratch. Also you might want to initialize parameters to small non-zero values.
- **loss function**: You can implement and experiment with different loss functions. One of them has to be BCE since it suits best for this type of problem. Make sure that your code doesn't run into Nan and inf values while taking logarithms!
- **try small and expand**: Make sure that your architecture can take 1 image data predict its class first. Later Expand the code to go through all train data set. This will help with dimension matching between fucntions and layers.
- **Chatgpt etc**: Feel free to use chatgpt for debugging your code or get ideas if you are stuck. However Do Not Use it to copy paste the code since we will use online tools to check for AI generated code! (Ohh , the irony!)

3.2 Kernel Selection and Usage

In this neural network, we will use specific convolutional kernels to enhance feature extraction in the hidden layers. Convolutional kernels, also known as filters, are matrices that, when applied to an image, highlight specific patterns like edges or textures. For this assignment, you will implement the following kernels:

- **Sobel Kernel**: A commonly used edge detection filter that highlights vertical edges in images.
- **Emboss Kernel**: A filter that emphasizes textures, making images appear embossed or raised.

***** If you want to use different filters comment on your choice of kernel and why that particular kernel would be better choice for this problem (because there are better ones)**

3.3 Additional Resources

Creating neural network functions can be challenging, especially when handling matrix operations and ensuring correct dimensions throughout the layers. We highly recommend watching Andrew Ng's lecture on this topic, which is available on YouTube. His explanations will be particularly useful for understanding how to set up each function correctly.

In particular, the section at **5:58 (Getting Your Matrix Dimensions Right)** is crucial for this assignment. This part of the video delves into matrix dimension alignment, which is a common source of errors when implementing neural networks.

You can find the video here:

- Andrew Ng: Neural Networks and Deep Learning Complete Course

Reviewing this content will greatly aid in implementing functions like `forward_propagation`, `backprop_dense_layer`, and `update_parameters`, where matrix dimensions need to be carefully managed.

4 Training, Validation, and Evaluation

After implementing the neural network functions, it is time to train your model on the training dataset, validate it on the validation dataset, and finally test its performance on the test dataset. Follow the steps below to ensure comprehensive evaluation of your model.

Training and Validation

1. **Training the Model**: Use the training data to train your model for a specified number of epochs. Track the training and validation loss at each epoch to monitor the model's performance.

2. **Plotting Training and Validation Loss**: Plot the training and validation loss over epochs. This will help you visualize the convergence of the model and check for signs of overfitting or underfitting. An example of the expected plot format is shown below:

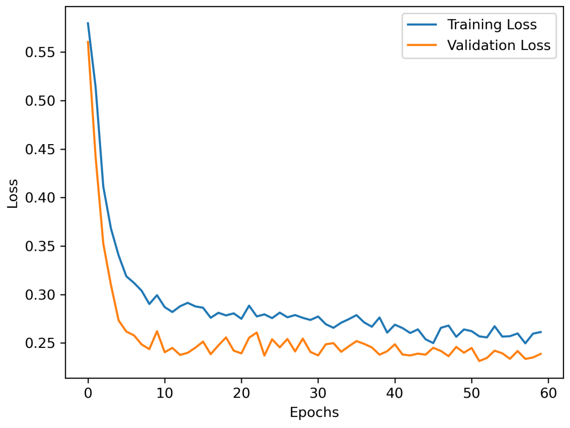

Figure 2: Training and Validation Loss

Evaluation on Validation and Test Sets

Validation Metrics: After each epoch, evaluate the model on the validation set using the following metrics:
- **Accuracy**
- **Precision**
- **Recall**
- **F1-Score**

Plot these metrics over epochs to observe how well the model generalizes. Below is an example of a metrics plot:

Testing the Model: Once training is complete, evaluate the model on the test dataset to obtain final performance metrics. Calculate and report the following:

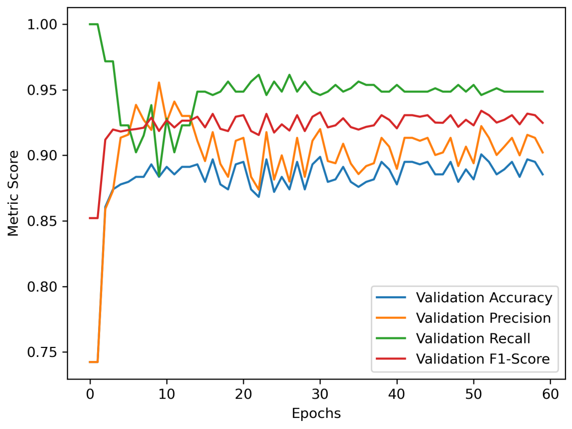

Figure 3: Validation Metrics: Accuracy, Precision, Recall, F1-Score

- **Confusion Matrix**: Display the confusion matrix to show the breakdown of true positives, false positives, true negatives, and false negatives.

- **ROC Curve and AUC**: Plot the Receiver Operating Characteristic (ROC) curve to illustrate the model's performance at various threshold levels. Additionally, calculate the Area Under the Curve (AUC) to quantify the model's ability to discriminate between classes. An example ROC curve is shown below:

Report All Metrics: After evaluating the model, report the following metrics:
- **Accuracy**: Overall proportion of correct predictions.
- **Precision**: Proportion of true positives among predicted positives.
- **Recall (Sensitivity)**: Proportion of true positives among actual positives.
- **F1-Score**: Harmonic mean of precision and recall, balancing both metrics.
- **Specificity**: Proportion of true negatives among actual negatives.
- **AUC (Area Under Curve)**: Measures the model's ability to distinguish between classes.

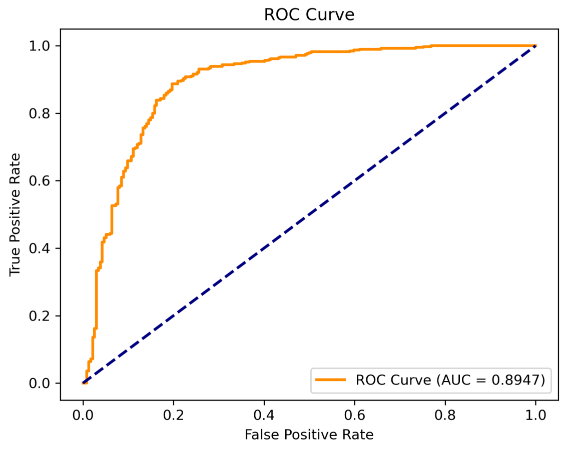

Figure 4: ROC Curve with AUC Score

Analysis and Interpretation

In addition to reporting the metrics, include a brief analysis:

- Discuss any patterns observed in the training and validation loss curves. For example, if validation loss plateaus or starts increasing, it may indicate overfitting.

- Explain the significance of each metric and what it reveals about your model's performance.

- Provide insights from the ROC curve and AUC score. A higher AUC score indicates a better-performing model in binary classification.

Submission Requirements

For the final submission, include:

- Your code and outputs.

- Plots of training/validation loss, validation metrics, confusion matrix, and ROC curve.

- A summary report of all metrics on the test set, along with your analysis and interpretation.

Following these steps will provide a thorough evaluation of your model, highlighting its strengths and areas for potential improvement.

***** In this assignment you won't be using any built in modules other than numpy until you get your results. NN architecture, all the functions related to it will be coded by you from scratch. That being sad, you can use inbuilt modules for performance measure metrics, Roc-curve etc. Also make sure to use different kernels and different activation functions for each layer of NN.**

5 Guiding Functions for Neural Network Implementation

To assist you in implementing your neural network, we have provided the names and parameters of several key functions below. These functions are designed to perform essential tasks within the network, such as activation, convolution, and pooling. Note that these are only for guidance, and you are free to implement any function in any way that achieves the same functionality.

Activation Functions

- `sigmoid(z)`: Sigmoid activation function, capped to prevent overflow.

```
def sigmoid(z):
    """Sigmoid activation function"""
    z = np.clip(z, -500, 500)  # This prevents overflow by capping values
```

- `sigmoid_derivative(z)`: Derivative of the sigmoid function, useful for backpropagation.

```
def sigmoid_derivative(z):
    """Derivative of sigmoid activation"""
```

- `relu(z)`: ReLU activation function, commonly used in deep learning.

```
def relu(z):
    """ReLU activation function"""
```

- `relu_derivative(z)`: Derivative of the ReLU function.

```
def relu_derivative(z):
    """Derivative of ReLU activation"""
```

Convolution and Pooling Functions

- `conv2d(X, kernel)`: Performs a 2D convolution on the input matrix `X` with the specified `kernel`.

```python
def conv2d(X, kernel):
    """
    Applies a 2D convolution operation.

    Parameters:
    - X: Input 2D array (e.g., image or feature map)
    - kernel: 2D array (e.g., filter or kernel)

    Returns:
    - output: 2D array after applying convolution
    """
    # Convolution operation
    # ... Fill in the rest of the function
```

- `get_kernel(kernel_name)`: Returns a predefined kernel based on the name provided. Supported kernels are "sobel" and "emboss".

```python
def get_kernel(kernel_name):
    """
    Returns the specified kernel based on the given name.

    Parameters:
    - kernel_name (str): Name of the kernel ('sobel' or 'emboss').

    Returns:
    - np.array: The selected kernel matrix.
    """
    # Return the requested kernel
    # ... Complete the function
```

- `max_pooling(X, pool_size=(2, 2))`: Applies max pooling to downsample the input matrix.

```python
def max_pooling(X, pool_size=(2, 2)):
    """
    Applies max pooling to the input matrix.

    Parameters:
    - X: Input 2D array (e.g., convolved image or feature map)
    - pool_size: Tuple representing the pooling window size (default is 2x2)

    Returns:
    - output: 2D array after max pooling
    """
    # Perform max pooling
    # ... Complete the function
```

- `flatten(X)`: Flattens a 2D array into a 1D vector, necessary for connecting convolutional layers to dense layers.

```
def flatten(X):
    """
    Flattens a 2D array into a 1D vector.

    Parameters:
    - X: Input 2D array (e.g., pooled feature map)

    Returns:
    - flattened_output: 1D array (flattened version of input)
    """
    # Flatten the input
    # ... Complete the function
```

Additional Guiding Functions for Dense and Output Layers

To further assist you with building the dense and output layers of the neural network, here are several key functions. As with the previous functions, these are provided as optional templates; you are free to adapt or modify them as long as they meet the desired functionality.

- `initialize_dense_layer(input_dim, output_dim)`: Initializes the weights and biases for a dense layer.

```
def initialize_dense_layer(input_dim, output_dim):
    """
    Initializes weights and biases for a dense layer.

    Parameters:
    - input_dim: Number of input units (e.g., flattened input size)
    - output_dim: Number of output units (e.g., number of neurons in this layer)

    Returns:
    - weights: Initialized weights matrix of shape (output_dim, input_dim)
    - biases: Initialized bias vector of shape (output_dim, 1)
    """
    # Implement the weight and bias initialization
    # ...
```

- `dense_layer(X, weights, biases)`: Applies a dense layer transformation, $Z = WX + b$.

```python
def dense_layer(X, weights, biases):
    """
    Applies a dense layer transformation: Z = WX + b

    Parameters:
    - X: Input 1D array (flattened vector from previous layer)
    - weights: Weight matrix for this layer
    - biases: Bias vector for this layer

    Returns:
    - output: 1D array (result after applying dense layer)
    """
    # Apply the dense layer transformation
    # ...
```

- `apply_activation(X, activation='relu')`: Applies the specified activation function to the input, with a choice of 'relu' or 'sigmoid'.

```python
def apply_activation(X, activation='relu'):
    """
    Applies the specified activation function to the input.

    Parameters:
    - X: Input array (result from the dense layer)
    - activation: Activation function to apply ('relu' or 'sigmoid')

    Returns:
    - Activated output
    """
    # Apply the specified activation function
    # ...
```

- `apply_dropout(X, dropout_rate=0.5)`: Applies dropout to the input by randomly setting a fraction of activations to zero.

```python
def apply_dropout(X, dropout_rate=0.5):
    """
    Applies dropout by randomly setting a fraction of activations to zero.

    Parameters:
    - X: Input array (output from activation function)
    - dropout_rate: Fraction of neurons to drop (default is 0.5)

    Returns:
    - Output with dropout applied
    """
    # Implement dropout by applying a random mask
    # ...
```

- `initialize_output_layer(input_dim)`: Initializes weights and biases for the output layer.

```
def initialize_output_layer(input_dim):
    """
    Initializes weights and biases for the output layer.

    Parameters:
    - input_dim: Number of input units (should match the last dense layer output)

    Returns:
    - weights: Initialized weights matrix for the output layer
    - biases: Initialized bias for the output layer
    """
    # Initialize weights and biases for the output layer
    # ...
```

- `output_layer(X, weights, biases)`: Applies the output layer transformation with sigmoid activation, providing a probability for binary classification.

```
def output_layer(X, weights, biases):
    """
    Applies the output layer transformation with sigmoid activation.

    Parameters:
    - X: Input array (output from previous dense layer, after activation and dropout)
    - weights: Weights matrix for output layer
    - biases: Bias for output layer

    Returns:
    - Output probability (single value between 0 and 1)
    """
    # Apply the output layer transformation and sigmoid activation
    # ...
```

Guiding Functions for Forward Propagation, Backpropagation, and Training

Below are additional functions that guide the essential operations for forward propagation, loss computation, backpropagation, and parameter updates. These functions are provided with names and parameters only; you will need to complete the code implementation.

- `forward_propagation(X, parameters, kernels, activation1='relu', activation2='sigmoid', dropout_rate=0.5)`: Performs forward propagation through the entire network.

```
def forward_propagation(X, parameters, kernels, activation1='relu', activation2='sigmoi
    """
    Perform forward propagation through the entire network.

    Parameters:
    - X: Input image (28x28)
    - parameters: Dictionary containing weights and biases for dense and output layers
    - kernels: List of convolution kernels, e.g., [sobel, emboss]
    - activation1: Activation function for first conv layer ('relu' or 'sigmoid')
    - activation2: Activation function for second conv layer ('relu' or 'sigmoid')
    - dropout_rate: Dropout rate for regularization

    Returns:
    - final_output: Output probability from the network
    - cache: Dictionary of intermediate values for backpropagation
    """
    # Implement the forward propagation logic here
    # ...
```

- `compute_loss(Y, Y_hat, loss_type='binary_cross_entropy')`: Computes the loss between true labels and predictions.

```
def compute_loss(Y, Y_hat, loss_type='binary_cross_entropy'):
    """
    Computes the loss between true labels and predictions.

    Parameters:
    - Y: True labels (1 or 0)
    - Y_hat: Predicted probabilities
    - loss_type: Type of loss function to use ('binary_cross_entropy' or 'mean_squared_

    Returns:
    - Loss value
    """
    # Implement the loss computation
    # ...
```

- `backprop_output_layer(Y, Y_hat, cache)`: Calculates gradients for the output layer.

```
def backprop_output_layer(Y, Y_hat, cache):
    """
    Calculates gradients for the output layer.

    Parameters:
    - Y: True labels (1 or 0)
    - Y_hat: Predicted probabilities
    - cache: Dictionary containing intermediate values from forward propagation

    Returns:
    - dW_out: Gradient of weights for output layer
    - db_out: Gradient of biases for output layer
    - dA_prev: Gradient with respect to the previous layer's activations
    """
    # Implement backpropagation for the output layer
    # ...
```

- `backprop_dense_layer(dA, cache, activation='relu')`: Calculates gradients for the dense layer.

```
def backprop_dense_layer(dA, cache, activation='relu'):
    """
    Calculates gradients for the dense layer.

    Parameters:
    - dA: Gradient of activations from the next layer
    - cache: Dictionary of intermediate values
    - activation: Activation function used in this layer ('relu' or 'sigmoid')

    Returns:
    - dW_dense: Gradient of weights for dense layer
    - db_dense: Gradient of biases for dense layer
    - dA_prev: Gradient with respect to previous layer's activations
    """
    # Implement backpropagation for the dense layer
    # ...
```

- `update_parameters(parameters, grads, learning_rate=0.01)`: Updates weights and biases using gradient descent.

```
def update_parameters(parameters, grads, learning_rate=0.01):
    """
    Updates weights and biases using gradient descent.

    Parameters:
    - parameters: Dictionary containing current weights and biases
    - grads: Dictionary containing gradients for each layer
    - learning_rate: Learning rate for gradient descent

    Returns:
    - parameters: Updated parameters
    """
    # Implement parameter update logic here
    # ...
```

- `train_model(X, Y, parameters, kernels, epochs=10, learning_rate=0.01, loss_type='binary_cross_entropy')`: Trains the model over multiple epochs, updating parameters based on computed gradients.

```
def train_model(X, Y, parameters, kernels, epochs=10, learning_rate=0.01, loss_type='bi
    """
    Trains the model over multiple epochs.

    Parameters:
    - X: Training data
    - Y: True labels
    - parameters: Dictionary of initial weights and biases
    - kernels: List of kernels for convolutional layers
    - epochs: Number of training epochs
    - learning_rate: Learning rate for gradient descent
    - loss_type: Loss function type ('binary_cross_entropy' or 'mean_squared_error')

    Returns:
    - parameters: Updated parameters after training
    - losses: List of loss values for each epoch
    """
    # Implement the training loop with forward propagation,
    # loss calculation, backpropagation, and parameter update
    # ...
```

These functions will help you structure your NN. Remember, these are only guiding templates, and you may define these functions differently as long as they achieve the intended purpose within the network. Each of these functions plays a key role in the network. While we've provided sample parameters and descriptions, you are encouraged to adapt or modify these functions to fit your implementation style. The final goal is to achieve the specified functionality, regardless of the exact structure of the code.