1 Convolutional Neural Networks [60 marks]

In this question, we will implement a convolutional neural network from scratch for MNIST (digit classification dataset). The CNN model will involve the following layers described below.

1.1 Convolution Layer

A convolution layer is a fundamental building block in convolutional neural networks (CNNs) designed to effectively capture spatial hierarchies in data such as images. It operates by convolving input data with learnable filters, also known as kernels or weights, to produce feature maps.

Let us denote:

- X to be the input vector with dimensions $D \times D$ to the convolutional layer, typically representing an image or a feature map from the preceding layer.
- K to be the set of learnable filters/kernels with dimensions $N_f \times F \times F$, where F is the filter size, and N_f is the number of output channels.
- b to be the bias term withm dimensions N_f associated with each filter in the convolutional layer.
- Z to be the output vector of the convolutional layer, containing feature maps.

The forward pass of convolution layer can be expressed mathematically as follows:

Forward Pass

$$Z(f,i,j) = \sum_{m=0}^{F-1}\sum_{n=0}^{F-1} X(i+m, j+n) \times K(f,m,n) + b(f)$$

Simplifying the equation involves utilizing the convolution operation denoted by $\circledast$ between the matrices X and K_f.

$$Z_f = X \circledast K_f + b_f$$

For the backward pass, let us assume that we have computed the gradient of the loss function with respect to the output of the convolutional layer, denoted as $\frac{\partial \mathcal{L}}{\partial Z}$ (output error), where $\mathcal{L}$ represents the loss function.

Backward Pass:

$$\begin{aligned}
\text{Filter Error} : \frac{\partial \mathcal{L}}{\partial K_f} &= X \circledast \frac{\partial \mathcal{L}}{\partial Z_f}, \\
\text{Bias Error} : \frac{\partial \mathcal{L}}{\partial b_f} &= \sum_{\text{all elements}} \frac{\partial \mathcal{L}}{\partial Z_f}, \\
\text{Input Error} : \frac{\partial \mathcal{L}}{\partial X} &= \sum_{f=0}^{N_f-1} K_f^{*} \hat{\circledast} \frac{\partial \mathcal{L}}{\partial Z_f},
\end{aligned}$$

where K_f^* is filter matrix K_f rotated by 180° and $\hat{\circledast}$ is full convolution. Read about matrix rotation and full convolution from these slides on CNN backpropagation.

Update filter kernel K_f and bias term b_f as:

$$K_f = K_f - \eta \cdot \frac{\partial \mathcal{L}}{\partial K_f}$$

$$b_f = b_f - \eta \cdot \frac{\partial \mathcal{L}}{\partial b_f}$$

where η is the learning rate.

1.2 ReLU Layer

A Rectified Linear Unit (ReLU) layer is a type of activation function commonly used in neural networks to introduce non-linearity. It helps models learn complex patterns by introducing the ability to model nonlinear relationships between input features and output predictions.

Denote:

- X as the input vector of any dimensions to the ReLU layer.
- Y as the output vector having same dimension as input vector of the ReLU layer.

Forward Pass:

$$Y = \max(0, X)$$

In other words, for each element x in the input tensor X, the ReLU activation function computes the maximum between x and 0, producing the corresponding element in the output tensor Y.

For the backward pass, let us assume we have computed the gradient of the loss function with respect to the output of the ReLU layer, denoted as $\frac{\partial \mathcal{L}}{\partial Y}$ (output error), where $\mathcal{L}$ represents the loss function.

Backward Pass:

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y} \cdot \mathbb{1}(X > 0)$$

where $\mathbb{1}(X > 0)$ is an indicator function that returns 1 if the corresponding element in X is greater than 0, and 0 otherwise.

1.3 Max Pooling Layer

Max Pooling is a common operation in convolutional neural networks (CNNs) used for downsampling and extracting dominant features from feature maps. It helps reduce the spatial dimensions of the input, making the model more computationally efficient and less sensitive to small variations in the input data.

Let's denote:

- X to be the input vector with dimensions $N_f \times H \times W$ to the max pooling layer, typically representing a feature map with N_f output channels from the preceding layer.
- F be the filter size for Max Pooling Layer. Assume H and W are divisible by F.
- Z as the output vector of dimensions $N_f \times H/F \times W/F$.

Forward Pass:

$$Y_{f,i,j} = \max_{(m,n)\in[F]\times[F]} (X_{f,i\times F+m,j\times F+n})$$

where: $[F]$ represents the set {0, 1, 2 ... (F-1)}.

For the backward pass, let's assume we have computed the gradient of the loss function with respect to the output of the Max Pooling layer, denoted as $\frac{\partial \mathcal{L}}{\partial Y}$ (output error), where $\mathcal{L}$ represents the loss function.

Backward Pass:

$$\frac{\partial \mathcal{L}}{\partial X_{f,i,j}} = \frac{\partial \mathcal{L}}{\partial Y_{f,\lfloor \frac{i}{F} \rfloor,\lfloor \frac{j}{F} \rfloor}} \cdot \mathbb{1}((i,j) = \operatorname*{arg\,max}_{(m,n)\in[F]\times[F]} (X_{\lfloor \frac{i}{F} \rfloor\times F+m,\lfloor \frac{j}{F} \rfloor\times F+n}))$$

where $\mathbb{1}$ is an indicator function that returns 1 for the element in the input vector that was selected as the maximum during the forward pass, and 0 otherwise.

1.4 Flatten Layer

A Flatten layer is a simple layer commonly used in neural network architectures to transform multidimensional input vectors into single dimensional vectors. It plays a crucial role in connecting the output of convolutional or pooling layers to the subsequent fully connected layers in the network.

Let's denote:

- X as the input vector of dimensions $N_f \times H \times W$ to the Flatten layer.
- Y as the output vector with dimensions $N_f \cdot H \cdot W \times 1$ of the Flatten layer.

Forward Pass:

$$Y = \text{flatten}(X)$$

For the backward pass, let's assume we have computed the gradient of the loss function with respect to the output of the Flatten layer, denoted as $\frac{\partial \mathcal{L}}{\partial Y}$ (output error), where $\mathcal{L}$ represents the loss function.

Backward Pass:

$$\frac{\partial \mathcal{L}}{\partial X} = \text{reshape}\left(\frac{\partial \mathcal{L}}{\partial Y}, \text{shape}(X)\right)$$

where reshape$(\cdot)$ is a function that reshapes the gradient tensor to match the shape of the input tensor X, and shape(X) returns the shape of X.

1.5 Fully Connected Layer (FCLayer)

A fully connected layer, also known as a dense layer, is a type of artificial neural network layer where each neuron is connected to every neuron in the preceding layer.

Let's denote:

- x as the input vector to the fully connected layer,.
- W as the weight matrix of the fully connected layer, where each row corresponds to the weights associated with one neuron, and each column corresponds to the weights connecting to one input feature.
- b as the bias vector, where each element represents the bias term associated with one neuron in the fully connected layer.
- z as the output vector of the fully connected layer.

The forward pass of a fully connected layer can be expressed mathematically as follows:

Forward Pass:

$$z = W^T x + b$$

For the backward pass, let's assume we have computed the gradient of the loss function with respect to the output of the fully connected layer, denoted as $\frac{\partial \mathcal{L}}{\partial z}$ (output error), where $\mathcal{L}$ represents the loss function.

Backward Pass:

$$\begin{aligned}\text{Weights Error}: \frac{\partial \mathcal{L}}{\partial W} &= x \cdot \frac{\partial \mathcal{L}}{\partial z}^T \\ \text{Bias Error}: \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial z} \\ \text{Input Error}: \frac{\partial \mathcal{L}}{\partial x} &= W \cdot \frac{\partial \mathcal{L}}{\partial z}\end{aligned}$$

These equations represent how the gradients flow backward through the fully connected layer during the training process via backpropagation, allowing the network to learn the optimal weights and biases to minimize the loss function.

Update weight matrix W and bias vector b as

$$W = W - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$$
$$b = b - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

where η is the learning rate.

1.6 Softmax Layer

A softmax layer is a type of layer commonly used in neural networks for multi-class classification tasks. It takes the raw output scores from the previous layer, often referred to as logits, and transforms them into a probability distribution over multiple classes. Each value in the output vector represents the probability of the corresponding class.

Let's denote:

- z as the input to the softmax layer, a vector of logits.
- s as the output of the softmax layer, a vector of probabilities.
- K as the number of classes.

The forward pass of a softmax layer can be mathematically expressed as follows:

Forward Pass:

$$s_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Here, s_i represents the i-th element of the output vector s, and z_i represents the i-th element of the input vector z. The softmax function ensures that the output vector S sums to 1 and each element lies between 0 and 1, representing probabilities.
For the backward pass, let's assume we have computed the gradient of the loss function with respect to the output of the softmax layer, denoted as $\frac{\partial \mathcal{L}}{\partial s}$ (output error), where $\mathcal{L}$ represents the loss function.

Backward Pass:

$$\text{Input Error}: \frac{\partial \mathcal{L}}{\partial z_i} = s_i \cdot \frac{\partial \mathcal{L}}{\partial s_i} - \sum_{j=1}^{K} s_i \cdot s_j \cdot \frac{\partial \mathcal{L}}{\partial s_j}$$

1.7 Tasks

You need to complete the following classes and functions inside `q1.py`:

- Implement class `ConvolutionLayer` as described above. We have already initialized the weights and bias for this layer. [25 marks]
- Implement class `ReLULayer` as described above. [10 marks]
- Implement class `MaxPoolingLayer` as described above. [15 marks]
- Implement class `FlattenLayer` as described above. [10 marks]

We have already implemented `FCLayer` and `SoftmaxLayer` in the template code. We wouldn't be checking for the training accuracy since the whole process would take significant amount of time, but it's a good sanity check to verify that the accuracy increases on your local devices.

2 CNNs using Torch [20 marks]

In this problem, we aim to classify images into different weather conditions (sunny, cloudy, rainy, snowy, foggy) using CNNs implemented with the PyTorch library.

2.1 Dataset Preparation

We are provided with a dataset containing images representing different weather conditions. Each image is of size 32 × 32 pixels and is in RGB format. Before feeding the images into the CNN, some preprocessing steps are required:

- Resize the images to a standard size of 32 × 32 pixels
- Normalize the pixel values to a range between -1 and 1
- Split the dataset into training and testing sets

The dataset is already prepared and split into the required sets for training and testing.

2.2 Model Architecture

We have given a simple CNN architecture, which consists of three layers:

- Convolutional layer with ReLU activation function
- Max-pooling layer for downsampling
- Fully connected layer for classification

2.3 Tasks

Your aim is to write the forward pass for this task and implement the training and testing loops. We have provided pointers to how the loops can be implemented in the skeleton code. However, similar to Lab 8, you are expected to go through the PyTorch documentation to understand the flow of the forward and backward passes.

You can also refer to this blog to get a better understanding of how CNNs are generally implemented with PyTorch. You have the following tasks to complete in `q2.py`:

- Complete the forward pass in `WeatherCNN` class. [5 marks]
- Write the training loop in the `train` function, which consists of performing the forward pass over the model, computing the cross-entropy loss and its gradients with respect to model parameters, and finally updating the parameters. The function returns a list comprising the loss of every 200 batches. [8 marks]
- Complete the `test` function to return the predictions of the model. [4 marks]
- Achieve an accuracy greater than 64% on the visible dataset. The final marks for this part will be given based on the accuracy achieved on the hidden dataset. [3 marks]