MTH767 Final Project

Viktor Penchev

May 2024

1 Introduction

In this project, I set out to write a Convolutional Neural Network (CNN) from scratch, implementing not only the forward pass functions but also the backward passes without using any machine learning libraries. This proved to be a decent challenge and my code ended up being far from optimal. Nevertheless, I achieved reasonable results on the MNIST dataset while incorporating a good amount of functionality into the layers.

2 Task and scope

The goal of this project was to implement a CNN all by myself and then test my model on the MNIST dataset. I set a challenge for myself to not use any machine learning libraries, such as Pytorch and TensorFlow, and to write all the forward and backward pass functions mainly using NumPy.

In terms of model architecture, I implemented the following layers:

- Convolutional Layer
- Max Pooling
- Instance Norm
- Flatten (Reshape) Layer
- Dense Layer
- ReLU Activation
- Softmax Activation

I trained 4 models of different depths on the MNIST dataset, with the best validation accuracy falling somewhere around 96%. After experimenting with different CNN layouts, I also tested the networks' performance when faced with adversarial examples generated using the Fast Gradient Sign Method.

3 Dataset

The MNIST dataset is made up of 28x28 grayscale images of hand-written digits between 0 and 9 and corresponding labels. CNNs have been shown to perform great on this dataset and the small image resolution makes the task manageable in terms of compute and complexity.

4 Model Architecture

4.1 Convolutional Layer

A Convolutional Layer is a fundamental building block of Convolutional Neural Networks (CNNs). It performs a convolution operation on the input data, which helps in extracting features and learning spatial hierarchies. The layer consists of a set of learnable filters (kernels) that slide across the input, computing the dot product between the filter and the input at each position. This operation produces a feature map that captures local patterns and structures in the input. In the provided code, the `ConvolutionalLayer` class implements the functionality of a convolutional layer. The layer is initialized with the input shape, output depth, kernel size, padding, stride, and a boolean flag for including bias. The weights of the layer are randomly initialized using a normalized initialization scheme, and the biases are initialized to zero. The forward pass of the convolutional layer involves the following steps:

1. Padding the input to preserve the spatial dimensions if necessary.
2. Initializing the output tensor with the bias values.
3. Performing cross-correlation between the input and the kernels using the `signal.correlate2d` function from the `scipy` library.
4. Applying the stride to downsample the output if required.

Mathematically, the forward pass of a convolutional layer can be expressed as:

$$y_{i,j,k} = \sum_{m} \sum_{n} \sum_{c} w_{k,c,m,n} \cdot x_{i+m,j+n,c} + b_k \tag{1}$$

where $y_{i,j,k}$ is the output at position (i, j) in the k-th feature map, $w_{k,c,m,n}$ is the weight at position (m, n) in the k-th kernel for the c-th input channel, $x_{i+m,j+n,c}$ is the input value at position $(i + m, j + n)$ in the c-th input channel, and b_k is the bias for the k-th output channel. The backward pass of the convolutional layer involves computing the gradients of the weights, biases, and input with respect to the output gradient. The key steps are:

1. Creating a dilated output gradient matrix which allows us to easily calculate the gradients when there is stride.

2. Computing the gradients of the kernels using cross-correlation between the input and the dilated output gradient.

3. Computing the gradients of the input using convolution between the dilated output gradient and the transposed kernels.

4. Updating the weights and biases using the computed gradients and the learning rate.

5. Cropping the padding from the input gradients if necessary.

The gradients of the weights and input can be calculated using the following equations[2]:

$$\frac{\partial y_{i,j,k}}{\partial w_{k,c,m,n}} = \sum_i \sum_j x_{i+m,j+n,c} \tag{2}$$

$$\frac{\partial y_{i,j,k}}{\partial x_{i,j,c}} = \sum_k \sum_m \sum_n w_{k,c,m,n} \cdot \frac{\partial y_{i,j,k}}{\partial y_{i-m,j-n,k}} \tag{3}$$

where $\frac{\partial y_{i,j,k}}{\partial w_{k,c,m,n}}$ is the gradient of the output at position (i, j) in the k-th feature map with respect to the weight at position (m, n) in the c-th input channel and k-th output channel, and $\frac{\partial y_{i,j,k}}{\partial x_{i,j,c}}$ is the gradient of the output at position (i, j) in the k-th feature map with respect to the input at position (i, j) in the c-th input channel.

4.2 Max Pooling Layer

A Max Pooling Layer is a commonly used downsampling operation in Convolutional Neural Networks (CNNs). It reduces the spatial dimensions of the input feature maps by applying a max operation over non-overlapping subregions. The purpose of max pooling is to extract the most prominent features, reduce the computational complexity, and provide translation invariance to small shifts in the input. In the provided code, the `MaxPooling` class implements the functionality of a max pooling layer. The layer is initialized with the input shape, kernel size, padding, and stride. The output depth remains the same as the input depth, while the spatial dimensions are reduced based on the kernel size and stride. The forward pass of the max pooling layer involves the following steps:

1. Padding the input if necessary to ensure the output dimensions are consistent.

2. Initializing the output tensor.

3. Iterating over the input feature maps and applying the max operation over non-overlapping subregions defined by the kernel size and stride.

Mathematically, the forward pass of a max pooling layer can be expressed as:

$$y_{i,j,k} = \max_{m,n} x_{i \cdot s_h + m, j \cdot s_w + n, k} \tag{4}$$

where $y_{i,j,k}$ is the output at position (i, j) in the k-th feature map, $x_{i \cdot s_h + m, j \cdot s_w + n, k}$ is the input value at position $(i \cdot s_h + m, j \cdot s_w + n)$ in the k-th input feature map, s_h and s_w are the stride values in the height and width dimensions, respectively, and the max operation is performed over the subregion defined by the kernel size. The backward pass of the max pooling layer involves computing the gradients of the input with respect to the output gradient. The key steps are:

1. Initializing the input gradient tensor.
2. Iterating over the output feature maps and the corresponding subregions in the input.
3. Identifying the position of the maximum value in each subregion.
4. Propagating the output gradient to the corresponding position in the input gradient tensor.
5. Cropping the padding from the input gradients if necessary.

The gradient of the input can be calculated using the following equation[2]:

$$\frac{\partial y_{m,n,k}}{\partial x_{i,j,k}} = \mathbb{1}_{i,j,k}(m, n) \tag{5}$$

where $\frac{\partial y_{m,n,k}}{\partial x_{i,j,k}}$ is the gradient of the output at position (m, n) in the k-th output feature map with respect to the input at position (i, j) in the k-th input feature map, and $\mathbb{1}_{i,j,k}(m, n)$ is an indicator function that is equal to 1 if (i, j) is the position of the maximum value in the subregion corresponding to (m, n), and 0 otherwise.

4.3 Instance Normalisation

Instance Normalization is a normalization technique commonly used in deep learning models, particularly in style transfer and image generation tasks. It normalizes the activations of each individual instance (sample) in a batch independently, as opposed to Batch Normalization, which normalizes the activations across the entire batch. In the provided implementation, Instance Normalization is chosen instead of Batch Normalization because the network processes a single sample at a time. The `InstanceNorm` class in the code implements the functionality of an instance normalization layer. It is initialized with the number of features (channels) in the input tensor, a small constant `eps` for numerical stability, and a boolean flag `affine` indicating whether to apply learnable scale and shift parameters. The forward pass of the instance normalization layer involves the following steps:

1. Reshaping the input tensor to compute the mean and standard deviation along the feature dimension.
2. Computing the mean and standard deviation of each feature independently.
3. Normalizing the input tensor by subtracting the mean and dividing by the standard deviation.
4. Applying learnable scale and shift parameters (gamma and beta) if `affine` is True.
5. Reshaping the normalized output tensor back to the original shape.

Mathematically, the forward pass of an instance normalization layer can be expressed as:

$$y_{i,j,k} = \frac{x_{i,j,k} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} \cdot \gamma_k + \beta_k \tag{6}$$

where $y_{i,j,k}$ is the normalized output at position (i, j) in the k-th feature map, $x_{i,j,k}$ is the input value at position (i, j) in the k-th feature map, μ_k and σ_k are the mean and standard deviation of the k-th feature map, respectively, ϵ is a small constant for numerical stability, and γ_k and β_k are the learnable scale and shift parameters for the k-th feature map. The backward pass of the instance normalization layer involves computing the gradients of the input, scale, and shift parameters with respect to the output gradient. The key steps are:

1. Reshaping the output gradient tensor to match the shape of the flattened input.
2. Computing the gradients of the scale and shift parameters (if `affine` is True).
3. Computing the gradient of the input using the chain rule and the gradients of the mean, standard deviation, and scale parameters.
4. Reshaping the input gradient tensor back to the original shape.
5. Updating the scale and shift parameters using the computed gradients and the learning rate.

The gradient of the input can be calculated using the following equations[1]:

$$\frac{\partial z_k}{\partial x_i} = a\left(\frac{1}{\sigma+\epsilon}\left(\delta_{ik} - \frac{\partial \mu}{\partial x_i}\right) - \frac{x_k - \mu}{(\sigma+\epsilon)^2}\frac{\partial \sigma}{\partial x_i}\right),$$
$$\frac{\partial \mu}{\partial x_i} = \frac{1}{n}, \qquad \frac{\partial \sigma}{\partial x_i} = \frac{x_i - \mu}{n\sigma}, \qquad \frac{\partial z_k}{\partial a} = \frac{x_k - \mu}{\sigma+\epsilon}, \qquad \frac{\partial z_k}{\partial b} = 1$$

Where:

z_k is the output of the batch normalization layer at position k; x_i is the input to the batch normalization layer at position i; a and b are the learnable scale

and shift parameters μ is the mean of the batch; σ is the standard deviation of the batch n is the batch size; ϵ is a small constant added for numerical stability; δ_{ik} is the Kronecker delta, which is 1 if $i = k$ and 0 otherwise.

4.4 Flatten (Reshape) Layer

The Flatten Layer, also known as the Reshape Layer, reshapes the input tensor into a desired output shape. It is commonly used to convert multi-dimensional feature maps into a one-dimensional vector for further processing by fully connected layers. The forward pass of the flatten layer reshapes the input tensor to the specified output shape, while the backward pass reshapes the output gradient tensor back to the original input shape. The flatten layer does not involve any learnable parameters or complex mathematical operations.

4.5 Dense Layer

The Fully Connected (FC) Layer, also known as the Dense Layer, connects every neuron in the input to every neuron in the output, allowing for complex non-linear transformations of the input data. The forward pass of a fully connected layer involves computing the matrix multiplication between the input and the weights, and optionally adding a bias term. Mathematically, the forward pass can be expressed as:

$$y_j = \sum_i w_{ij} x_i + b_j \tag{7}$$

where y_j is the output of the j-th neuron, x_i is the input from the i-th neuron in the previous layer, w_{ij} is the weight connecting the i-th input neuron to the j-th output neuron, and b_j is the bias term for the j-th output neuron. The gradients of the weights and input can be calculated using the following equations:

$$\frac{\partial y_j}{\partial w_{ij}} = x_i \tag{8}$$

$$\frac{\partial y_j}{\partial x_i} = w_{ij} \tag{9}$$

where $\frac{\partial y_j}{\partial w_{ij}}$ is the gradient of the output of the j-th neuron with respect to the weight connecting the i-th input neuron to the j-th output neuron, and $\frac{\partial y_j}{\partial x_i}$ is the gradient of the output of the j-th neuron with respect to the i-th input neuron.

5 Training Strategy

In this section, I discuss the training strategy employed to train the Convolutional Neural Networks (CNNs) for image classification. I present three different CNN architectures and the training loop used to optimize their parameters.

5.1 Training Loop

The training loop is implemented to train each CNN architecture on the training data and evaluate its performance on the test data. The key steps in the training loop are as follows:

1. Iterate over each CNN architecture.
2. Train the network using the `train` function with the following parameters:
    - Network architecture
    - Categorical cross-entropy loss function and its derivative
    - Training images and labels
    - Test images and labels
    - Number of epochs (50, 100, 150 respectively)
    - Learning rate (0.01)
    - Learning rate decay (0.985, 0.992, 0.995 respectively)
    - Batch size (128)
    - Test evaluation frequency (every 5 epochs)
3. Record the training errors and test scores for each network.
4. Evaluate the trained network on the test data using the `predict` function.
5. Calculate and print the test accuracy for each network.

The training loop utilizes the categorical cross-entropy loss function to measure the discrepancy between the predicted and true class probabilities. The networks are trained using gradient descent optimization with a learning rate of 0.01 and a learning rate decay proportional to the number of epochs. The batch size is set to 128, and the networks are trained for a number of epochs proportional to their complexity (50, 100, and 150 epochs for Networks 1, 2, and 3, respectively).

During training, the test accuracy is evaluated every 5 epochs to monitor the network's performance on unseen data. After training, the final test accuracy is calculated and reported for each network.

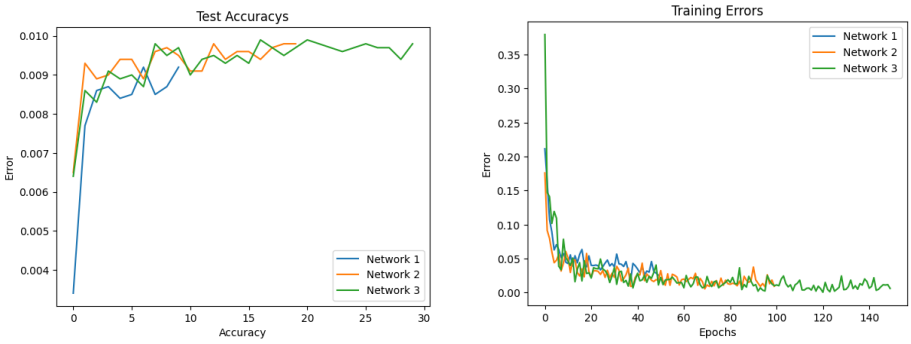
The training strategy aims to optimize the parameters of the CNN architectures to achieve high classification accuracy on the given dataset. The increasing complexity of the networks allows for capturing more intricate patterns and features in the data, potentially leading to improved performance.

6 Results

The achieved are results can be seen here:

Network	Accuracy
Network 1	0.9123
Network 2	0.9625
Network 3	0.9768

Table 1: Test accuracies for the three CNN architectures.



(a) Test accuracies during training.

(b) Training errors during training.

Figure 1: Test accuracies and training errors for the CNN architectures.

7 Conclusion

In this project, I successfully implemented a Convolutional Neural Network (CNN) from scratch using Python and NumPy, without relying on any machine learning libraries. The implemented layers included Convolutional, Max Pooling, Instance Normalization, Flatten, and Dense layers, along with ReLU and Softmax activations. By training the CNN on the MNIST dataset, I achieved a best validation accuracy surpassing 97%, demonstrating the effectiveness of the implemented architecture and training strategy. This project provided valuable insights into the inner workings of CNNs and the challenges involved in implementing them from scratch. The hands-on experience gained through this project has deepened my understanding of deep learning and has prepared me for further exploration and application of CNNs in various domains.

8 References

References

[1] L. Sinai, "Backpropagation through a layer norm," Lior Sinai, May 18, 2022. [Online]. Available: `https://liorsinai.github.io/mathematics/2022/05/18/layernorm.html`. [Accessed May 15, 2024].

[2] The Independent Code (2021). Convolutional Neural Network from Scratch — Mathematics & Python Code. [online] YouTube. Available at: https://www.youtube.com/watch?v=Lakz2MoHy6o&t=1015s [Accessed 10 May 2024].