Building a Convolution Neural Network from Scratch: Aditya Kulkarni

Introduction

This project aims to build a convolutional neural network without the use of any deep learning libraries. Effectively, only Python and NumPy will be used for the development of this project. However, TensorFlow Keras is used in one case, which is to import the MNIST handwriting dataset.

In my implementation, I follow the general pipeline of a convolution neural network, with Convolution → ReLU → Pooling → Fully Connected → Softmax → Loss. In this project, we use cross-entropy loss, $-\sum_{i=0}^{N} p_i \cdot \ln(y_i)$, to evaluate the Softmax output, which consists of 10 logits corresponding to the probabilities of classes 0 to 9.

Overall, the goal of this project is to ideally achieve above an 85% test accuracy on the MNIST handwriting dataset with at least one CNN architecture.

The Convolution Layer

The Convolution layer, and its initialization, forward, and backward pass algorithms, sit in **Convolution.py**.

To begin, we provide to the layer the parameters of **filter_size**, **num_filters**, **depth**, and **reg**. From here, in the initialization, we create **filters**, which represents the filters of the layer in the form: N * H * W * C, where N is the number of filters, H = W is the size of the filter, and C is the depth of the filter.

The forward method begins by saving the input into **last_input** for backpropagation, then initializing variables to help with the convolution operation. The output is stored respectively into the variable **output**, with the shape of H * W * C, where H = W is the new dimension using the formula $Dimension_{output} = H_{input} - N + 1$, and C is **num_filters**. We then use the convolution equation $O(i, j, k) = \sum_{m=0}^{H_{in}-1} \sum_{n=0}^{W_{in}-1} \sum_{o=0}^{C_{in}-1} w(m, n, o) \cdot x(i + m, j + n, k) + b$ to get a region for all channels and convolve it with each filter and bias, then return this output.

The backwards method begins by initializing the gradients necessary for the convolution operation. Let us first discuss the gradient with respect to the weights of the filters ∇w , whose partial derivative is $\frac{\partial L}{\partial w}$. This partial derivative is the input multiplied with the incoming gradient. However, in the case of a convolution, this will be applied over every region on the input image the size of the filter, and subsequently summed up. Thus, it can be defined as $\frac{\partial L}{\partial w_k} = \sum_{m=0}^{H_{out}-1} \sum_{n=0}^{W_{out}-1} \frac{\partial L}{\partial Y_{i,\,j,\,k}} \cdot x(i + m,\, j + n,\, k)$, for all filters. The bias has a partial derivative of 1. This means that the bias gradient ∇b will be an accumulation of the incoming gradient multiplied by 1, $\frac{\partial L}{\partial b_k} = \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \frac{\partial L}{\partial Y_{i,\,j,\,k}} \cdot 1$, for all filters. Finally, the gradient with respect to the input ∇x is calculated by summing the filter multiplied by the incoming gradient for all regions of $\frac{\partial L}{\partial x}$, as follows: $\frac{\partial L}{\partial x_{m,\,n,\,o}} = \sum_{k=0}^{N-1} \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \frac{\partial L}{\partial Y_{i,\,j,\,k}} \cdot w_k(m,\, n,\, o)$. Finally, we update the **filters** (with regularization) and **biases**, then return ∇x .

The Pooling Layer

The Pooling layer, and its initialization, forward, and backward pass algorithms, sit in **Pooling.py**. We first provide to the layer the singular parameter of **pool_size**. This is the only input necessary for initialization.

From here, we define the forward method, which, similar to the convolution layer, saves **last_input** for backpropagation. It stores the output into the variable **output**, with the shape of H * W * C. H = W is the new dimension using the formula $Dimension_{output} = H_{input} \div 2$. We then use the pooling equation $O(i,\, j,\, k) = \max_{a,\, b,\, c \,\epsilon\, [0,\, S]} x(i + a,\, j + b,\, k + c)$, where S is the pool size, and return the output.

The backwards method for the pooling operation originally seemed complex, but I quickly realized that it is relatively simple. In this, we do not perform any complicated computation in regards to derivation. Rather, we simply identify the max value within a region from the original input, and assign $\frac{\partial L}{\partial x}$ at that location as the incoming partial

derivative at that location. The rest of the region is left as zeroes, rendering us the gradient ∇x . After we perform this for every region and channel, we return $\frac{\partial L}{\partial x}$.

The ReLU Activation Function

The ReLU Activation Function, and its forward and backward pass algorithms, sit in **ReLU.py**. No initialization takes place.

The forward method stores the **last_input**, and then simply returns the input, with all non-positive values set to zero, as follows: $f(x) = \max(0, x)$.

The backwards method of the ReLU activation function is simple and similar to the pooling backpropagation. We first compute $\frac{\partial y}{\partial x}$, which sets the value to 1 if $x > 0$, and 0 otherwise. We then chain this with the incoming gradient to only preserve gradients for values that are positive from the original input, represented as $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$. This gives us the gradient with respect to the input, ∇x , which we then pass backwards.

The Fully Connected Layer

The Fully Connected layer, and its initialization, forward, and backward pass algorithms, sit in **FullyConnected.py**. For the fully connected layer, we initialize the weight and bias parameters of the perceptron model by passing in our input and output size.

The forward method begins by saving the **last_input** for backpropagation, and then flattens the incoming input. Take a tensor of 12x12x3 - a fully connected layer is only able to interpret this by flattening it to 432 individual values. The output size is arbitrary, but the final fully connected layer must have an output size of our classes, which for MNIST [1] is 10. The output itself is a dot product between the flattened input and the weights, then added to the bias, as so: $f(x) = wx + b$, which is returned as the output.

The backwards method begins by reshaping the original flattened input into a column matrix and converting the gradient into a vector array. We then take a dot product between the column matrix and the gradient vector to obtain $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y^{(l)}}\frac{\partial y^{(l)}}{\partial w}$, where the

partial derivative $\frac{\partial y^{(l)}}{\partial w}$ with respect to the weight is I, and $\frac{\partial L}{\partial y^{(l)}}$ is the incoming gradient from our Softmax backwards pass, or even a different fully connected layer's gradient. Next, we simply set $\frac{\partial L}{\partial b}$ as the incoming gradient since $\frac{\partial y^{(l)}}{\partial b}$ is 1. Thus, $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y^{(l)}} \cdot 1$. Next, we calculate the partial derivative with respect to the input, $\frac{\partial y^{(l)}}{\partial x}$, and get W. Therefore, $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y^{(l)}} \cdot W$. These three form our gradients ∇w, ∇b, ∇x. Finally, we update our **weights** (with regularization) and **biases** matrices before returning $\frac{\partial L}{\partial x}$ reshaped as the previous input.

The Softmax Layer

The Softmax layer, and its forward and backward pass algorithms, sit in **Softmax.py**. This layer has no initialization.

The forward method stores the **last_input**, and then simply uses the Softmax equation, $y_i = \frac{e^{z_i}}{\sum_{n=0}^{N} e^{z_j}}$, over 10 outputs to compute their normalized probabilities as a 10x1 vector.

The backward pass of the Softmax layer begins by recomputing the softmax probabilities, and computing it's Jacobian matrix of partial derivatives $\frac{\partial y_i}{\partial z_j} = \{y_i \cdot (1 - y_i) \; if \; i = j\}, \{-y_i \cdot y_j \; if \; i \neq j\}$. In this, $i = j$ represents the change of a probability with respect to its own logit, whereas $i \neq j$ represents the change of a probability with respect to a different class's logit. We then take a dot product between the Jacobian and the cross-entropy gradient for all partial derivatives $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}$, and pass this back to the fully connected layer.

Building the Model

Building the model takes place in **CNN.py**. The CNN is initialized as an array of all its layers stored in **layers**, and the learning rate is also set here.

The forward method passes the input feature, a handwritten digit stored as a 28x28x1 tensor, through **layers**, adding to the regularization loss if it comes upon a layer that utilizes learning. It then generates the loss value using the cross-entropy loss formula, and adds the regularization loss. It then sets accuracy to 1 if the prediction was correct, and to 0 if not.

The backwards method simply reverses **layers**, and passes gradients back until the last layer is reached.

The train method is used to to train the model itself. It takes in the inputs of the features, their corresponding labels, and an epoch number to denote how many times to forward and backward pass. For each epoch, the training data will be shuffled to prevent the model from generalizing learning. Then, the model will loop through the entire dataset, generating the output, loss, and accuracy from the final Softmax prediction. To begin backpropagation, we take the gradient of cross-entropy loss, which will result in a 10x1 vector of the partial derivatives, $\frac{\partial L}{\partial y_i} = -\frac{p_i}{y_i}$, where y_i is the probability and p_i is the true label. We feed this initial gradient into the backwards method.

The test method simply enumerates through each feature/label pair, records the accuracy, and returns the total accuracy in percentage format.

Running the Model and Results

The actual use of the model takes place in **Main.py**. Here, we are able to construct models with various types of architectures. In the Jupiter Notebook Demo, I have opted to test three architectures – a single layer CNN, a no-activation function CNN, and a deeper CNN. In this section, we will briefly discuss the dataset's importation and then the results of the different models.

The training and testing data from the MNIST dataset is imported using TensorFlow Keras, and then expanded to accommodate another dimension. The reason for this lies in how the convolution operation works regarding filters, and by converting the 28x28 data

into the form of 28x28x1, we can keep our code in **Convolution.py** cleaner and more readable. We then take a subsection of the dataset, in our case, 1000 images.

We create three different CNN's, all trained on 50 epochs with different hyperparameters:

The barebone single layer CNN is defined as Convolution 5x5x3 → → ReLU → Pooling 2 → Fully Connected → Softmax.

- 84.40% training accuracy, 80.80% test accuracy.

The no activation function CNN is defined as Convolution 5x5x3 → Pooling 2 → Fully Connected → Softmax.

- 85.40% training accuracy, 78.00% test accuracy.

The deeper CNN is defined as Convolution 7x7x10 → ReLU → Pooling 2 → Convolution 5x5x10 → ReLU → Convolution 3x3x15 → ReLU → Fully Connected → Softmax.

- 95.20% training accuracy, 86.90% test accuracy.

As expected, the deeper CNN performs the best, likely due to additional hierarchical feature extraction compared to the barebone and no activation function CNN's. It's interesting to note that the no activation function CNN overfits more than the barebone CNN, which is likely due to the additional depth and lack of non-linearity that the ReLU brings. Fortunately, through all my work, I was able to conquer my goal of an 85%+ test accuracy.

It is also important to note the model's subpar performance. State of the art techniques can achieve 95% to 99% accuracy on the MNIST dataset – so why doesn't mine?

First, the biggest issue is training. My computer does not have the capability to train on 60K images, as this would take too long, and I cannot keep my computer running a Python program for hours straight. I also would not have the resource credits to do so on

Colab or another platform. Second, my model has a high tendency to overfit, likely due to poor regularization technique. Without implementing techniques such as batch normalization, neuron dropout, etc., the model tends to overfit to the training data. Thirdly, I did not use techniques such as batching, Adam Optimizer, SGD, etc. Lastly, most of my effort went into formulating the ideas and converting equations into real code. Computing derivatives, implementing the convolution/pooling/etc. operations, ensuring that shapes match for backpropagation, and continually addressing implementation errors left me with little time to tune hyperparameters. I am confident that if I had more time to test various hyperparameters, the model's performance could see improvement.

The Takeaway

This project has been a tremendous learning experience for me. I've now learnt the complete ins and outs of a convolutional neural network, and how each layer manipulates data to produce output. Even further, I would say I have a full understanding of how a multilayer perceptron works as well, since that would just be multiple **FullyConnected.py** layers in sequence, and then a Softmax layer as the final layer. More importantly, though, is the understanding of backpropagation I now have. I completely understand the calculus and linear algebra behind how it works, and that we compute derivatives at each step, then pass it to the next step, effectively leveraging the chain rule as a way to save computational overhead. Furthermore, I believe that coding concepts are just as important as conceptual understanding. The ability I've gained from coding up this project ground up has strengthened my skills in Python and NumPy, and will make me a better overall programmer.