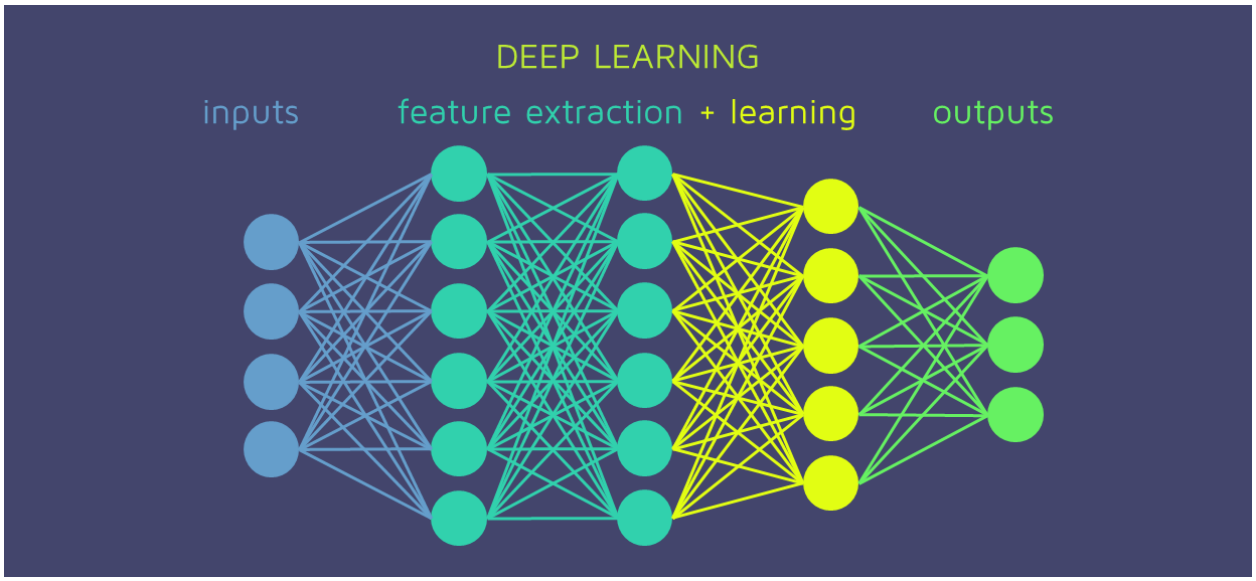
Manually Implemented Convolutional Neural Network using only Numpy

Sebastian Matiz

Rutgers University

Intro to Deep Learning Final Project



Manual Convolutional Neural Network

Abstract

For my final project in Introduction to Deep Learning I designed and tested a Manually Implemented Convolutional Neural Network using only Numpy. This CNN design was based on the Lenet-5 network.

Manually Implemented Convolutional Neural Network using only Numpy

A Convolutional Neural Network is a deep neural network that uses convolution operations for feature extraction. Following the convolutions, some sort of pooling method is used to subsample the extracted features. This process is repeated until we reach the fully connected layers, followed by a softmax layer that returns our predicted probabilities.

Method

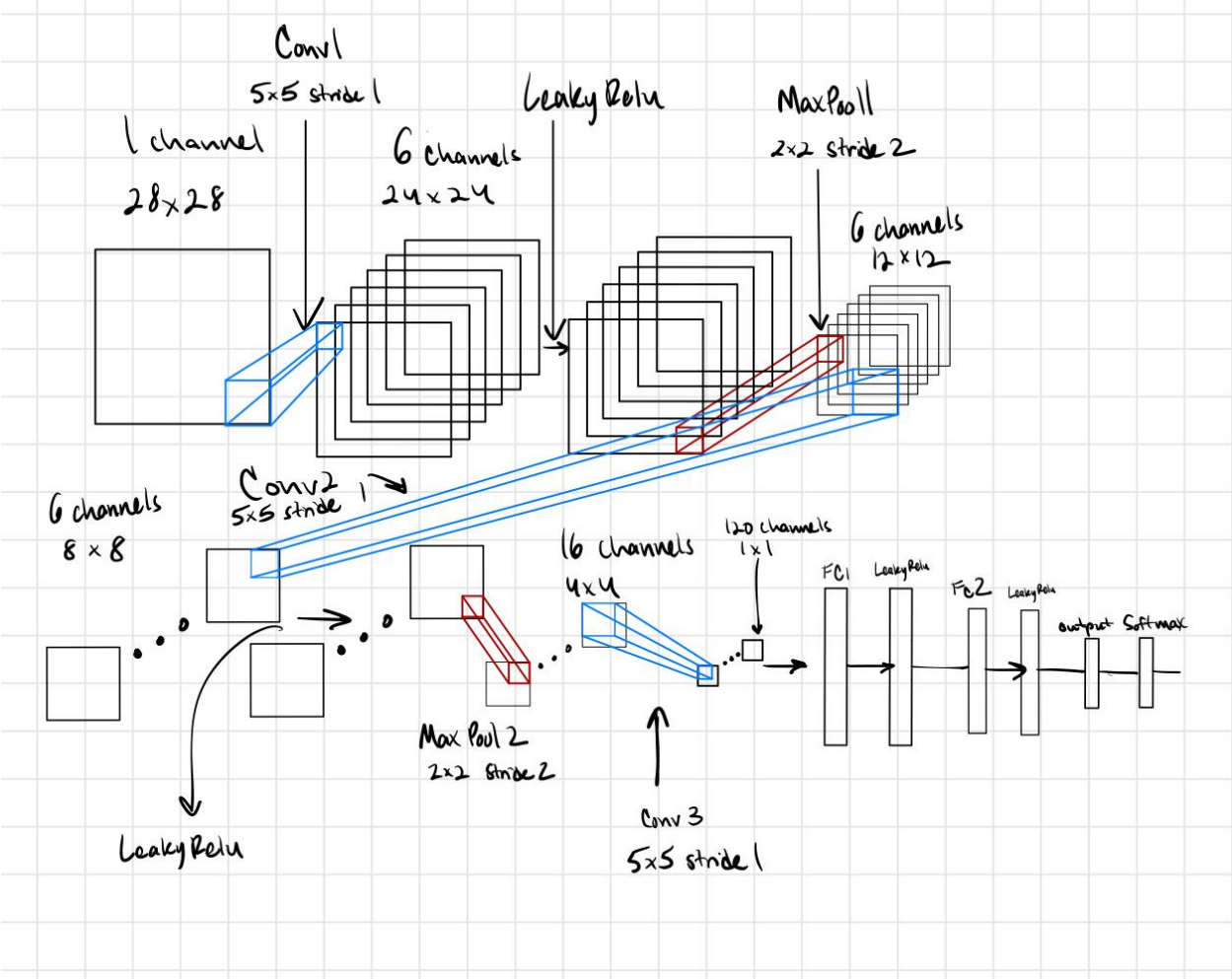
Building of the Network

For the network architecture, I decided to base my CNN on the Lenet-5 network. I decided to base it on the Lenet-5 network because the Lenet-5 has been shown to work very well with classification datasets similar to MNIST. The structure of my network is given below:

- Convolution Layer 1
- Leaky ReLu
- Maxpool
- Convoluion Layer 2
- Leaky ReLu
- Maxpool
- Convolution Layer 3
- Fully Connected Layer 1
- Leaky Relu
- Fully Connected Layer 2

Manual Convolutional Neural Network

- Leaky Relu
- Output Layer
- Softmax



Architecture Visualized.

Forward Propagation. Since I developed this CNN using just numpy, I needed to create functions for the operations used in a CNN such as 2D Convolutions, Maxpooling, Softmax, and

Manual Convolutional Neural Network

Leaky Relu. I also created classes for Convolution Layers, Kernels/Filters, Maxpooling, and Fully Connected Layers.

The Convolutional Layer class has the following attributes.

```
            self.kernelList = [] // a list of all the filters used in a convolution
layer
            self.inC = [] // a list of all the input channels
            self.outC = [] // a list of all the output channels
            self.bias = [] // a list of all the bias's that will be added to the
output channels
            self.inCDim = inCDim // number of input channels
            self.outCDim = outCDim // number of output channels
```

The Kernel/Filters class has the following attributes.

```
    # Kernel/Filter class
    class Kernel:
        def __init__(self, dim, inDim):
            self.kernel = np.random.normal(loc=0.0, scale=(.005), size=(dim, dim))
            // The values of the filters themselve are distributed as Gaussian
Distribution with mean 0 and standard deviation 0.005
```

The Maxpooling class has the following attributes.

```
    class MaxPoolingLayer:
        def __init__(self, kernelDim, stride):
            self.kernelDim = kernelDim // size of the kernel
            self.inC = [] // list of input channels before maxpooling
            self.outC = [] // list of the output channels after maxpooling
            self.stride = stride // stride for maxpooling kernel
```

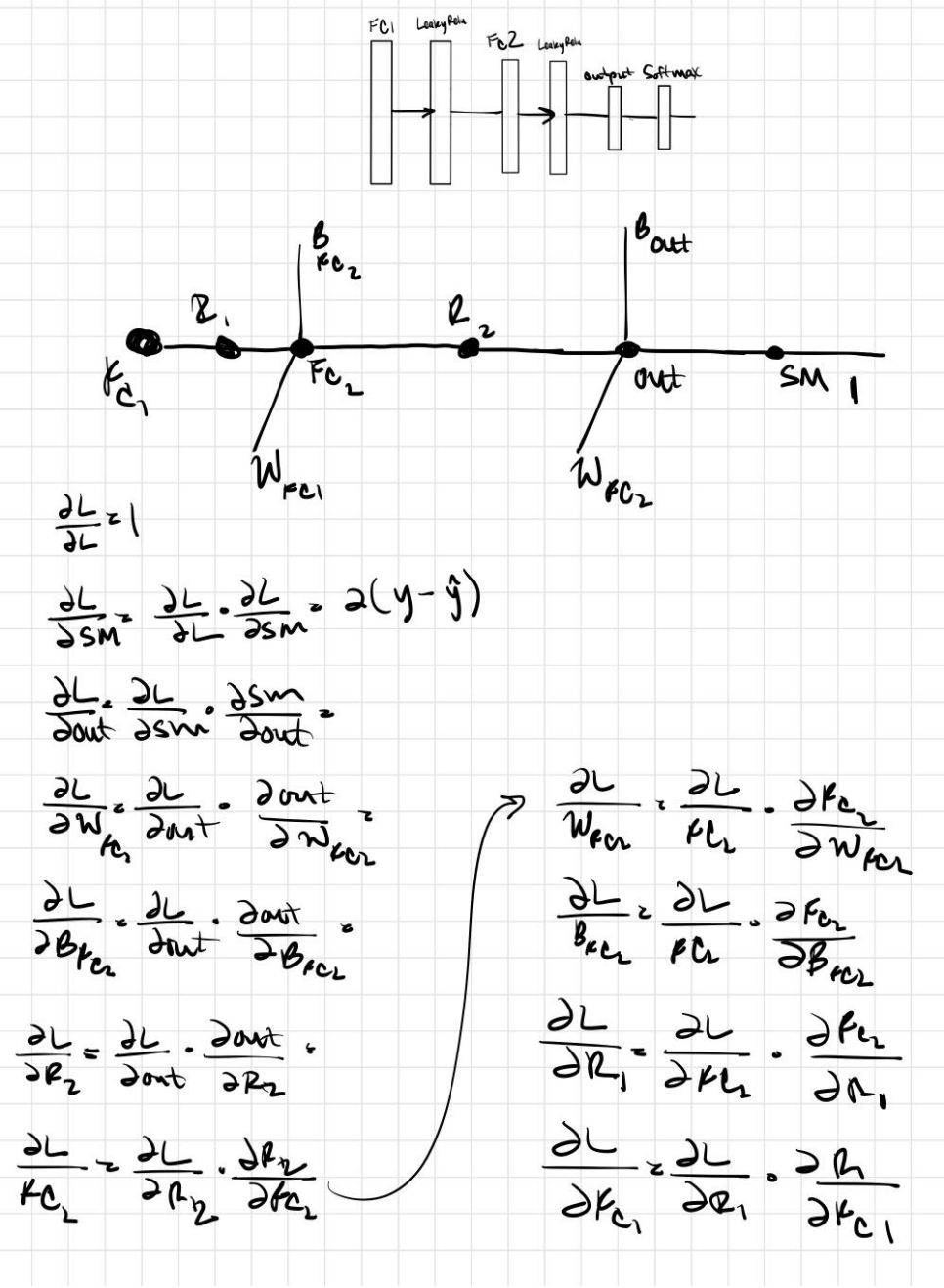
The Fully Connected Layer class has the following attributes.

```
    class FCLayer:
        def __init__(self, numNodes, numWeightsPerNode):
```

```
            self.nodeValues = np.full((numNodes, 1),0, dtype=float)
            # intialize the weights and bias as a Guassian distribution with mean 0
and standard deviation sqrt(2/(numInputNodes)
            self.bias = np.random.normal(loc=0.0, scale=(math.sqrt(2/(numNodes))),
size=(numNodes, 1))
            self.weights = np.random.normal(loc=0.0,
scale=(math.sqrt(2/(numNodes))), size=(numNodes, numWeightsPerNode))
```

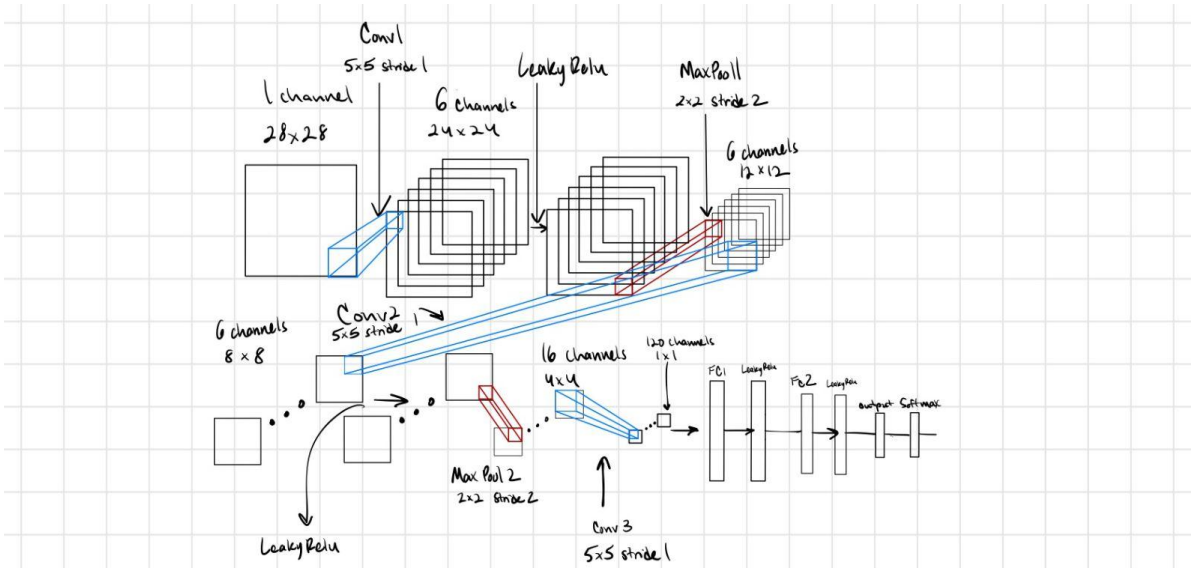
Backward Propagation. Using NumPy I created functions for finding the local gradient for 2D Convolution operations, max-pooling, Leaky Relu, and Softmax. Using the chain rule I backpropagated the loss through the fully connected layers and convolution layers. Below I have attached the calculations for finding the gradients of each layer with respect to the loss.

Manual Convolutional Neural Network



Gradients for the FC layers

Manual Convolutional Neural Network



$$\frac{\partial L}{\partial inConv3} = Full\ Conv\left(\hat{Filter}, \frac{\partial L}{\partial F_{c_1}}\right)$$

$$\frac{\partial L}{\partial K_3} = Conv\left(inConv3, \frac{\partial L}{\partial F_{c_1}}\right)$$

$$\frac{\partial L}{\partial B_3} = \frac{\partial L}{\partial F_{c_1}}$$

$$\frac{\partial L}{\partial mp_2} = \partial MaxPool$$

$$\frac{\partial L}{\partial LRelu} = \frac{\partial L}{\partial LRelu} \cdot \frac{\partial L}{\partial mp2}$$

$$\frac{\partial L}{\partial inConv2} = Full\ Conv\left(\hat{Filter}, \frac{\partial L}{\partial LRelu}\right)$$

$$\frac{\partial L}{\partial K_2} = Conv\left(inConv2, \frac{\partial L}{\partial LRelu}\right)$$

$$\frac{\partial L}{\partial B_2} = \frac{\partial L}{\partial LRelu}$$

$$\frac{\partial L}{\partial mp_1} = \partial MaxPool$$

$$\frac{\partial L}{\partial LRelu} = \frac{\partial L}{\partial LRelu} \cdot \frac{\partial L}{\partial mp1}$$

$$\frac{\partial L}{\partial K_1} = Conv\left(inConv1, \frac{\partial L}{\partial F_{c_1}}\right)$$

$$\frac{\partial L}{\partial B_1} = \frac{\partial L}{\partial LRelu}$$

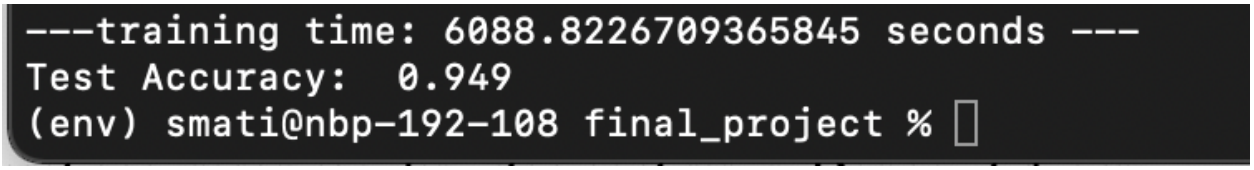
Gradients for the Convolutional layers

Manual Convolutional Neural Network

Trian. The Kernels/Filters and Bias for the Convolutional Layers were initialized as a Gaussian Distribution with mean=0 and standard deviation=0.005. The Weights and the Bias's for the Fully Connected Layers were initialized as a Gaussian Distribution with mean=0.0 and standard deviation=$\sqrt{2/(number\ of\ input\ nodes)}$. I trained the model using Stochastic Gradient Descent using a batch size of 128 that was generated randomly. I trained until a batch produced a train accuracy of above 97%. The learning rate of the model was 0.0002.

Test. The model was tested on the test partition of the MNIST dataset and achieved 94.9% accuracy after 160 epochs of training.

Results



Test Accuracy and Training time.

Resulting Loss over each Epoch.

Discussion

I decided to use Leaky Relu as opposed to Relu because at the beginning of my experimentation I saw that the Relu operation was killing all the resulting values hence Leaky Relu solved this problem.

Secondly, for the instantiation of the filters, weights, and biases I could have used cross-validation to find the optimal values, but given a time contraint I settled for trial and error.

Lastly, this implementation of a CNN is serial hence the run time is exceptionally slower as compared to the CNN models that can be made in PyTorch. In the future/given more time I would have made the functions of the CNN parallelized for optimal speedup.

References

https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

https://towardsdatascience.com/forward-and-backward-propagation-of-pooling-layers-in-convolutional-neural-networks-11e36d169bec

https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/

Intro to Deep Learning Lecture notes.