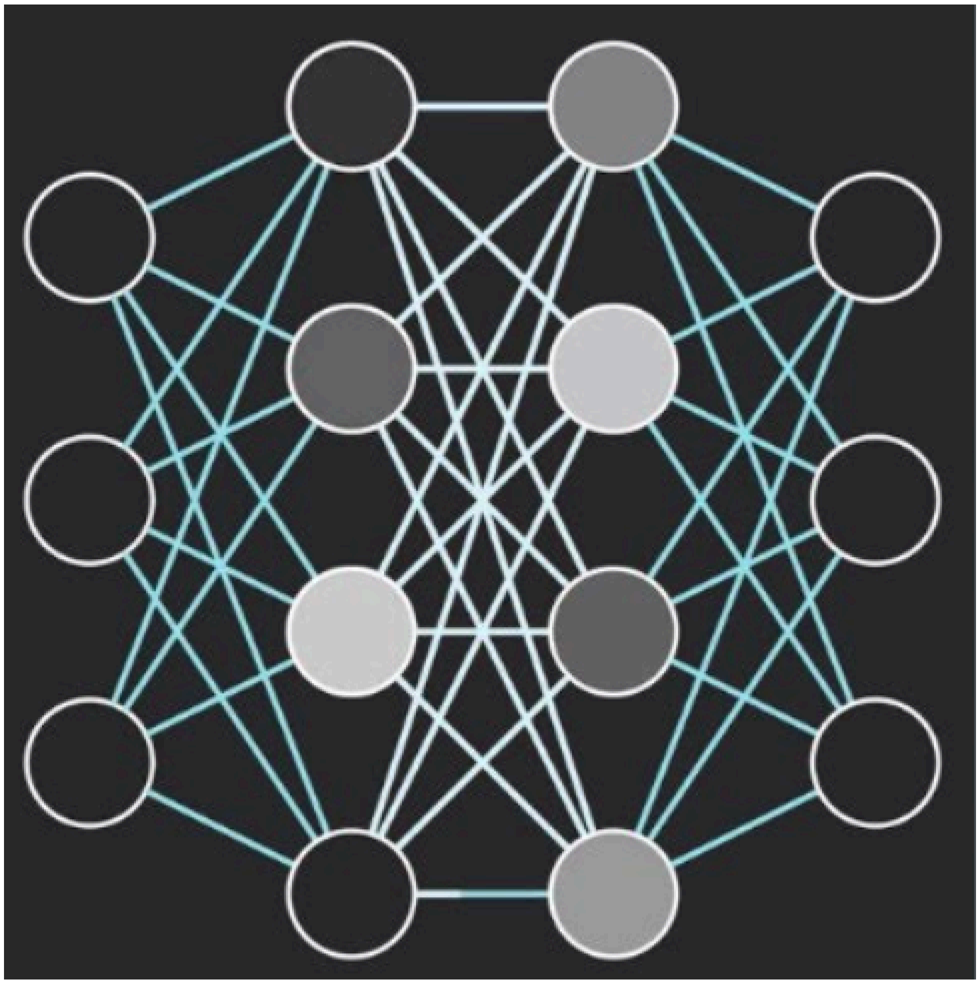
MNIST HAND-WRITTEN DIGITS CNN FROM SCRATCH

LeNet ARCHITECTURE



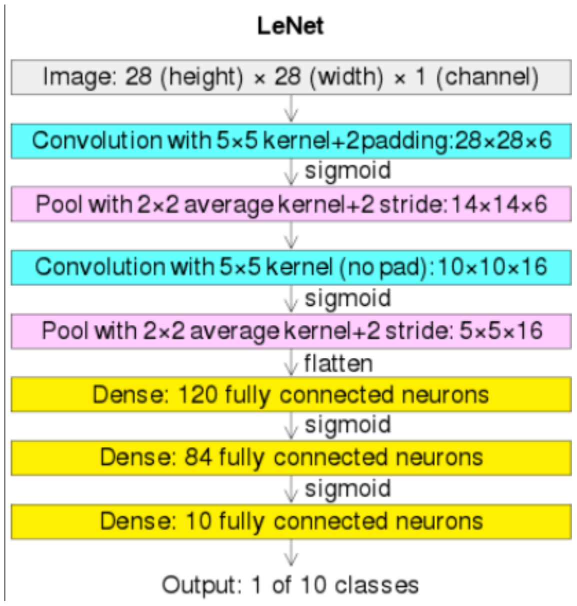
Abe Gizaw

Introduction

This paper documents my implementation of a LeNet Convolutional Neural Network (CNN) model built from scratch in Python. It recreates the architecture of LeNet, specifically designed for the classification of handwritten digits. The focus lies on replicating the original structure and parameters of LeNet, offering a detailed walkthrough of the results of different architectures.

Architecture

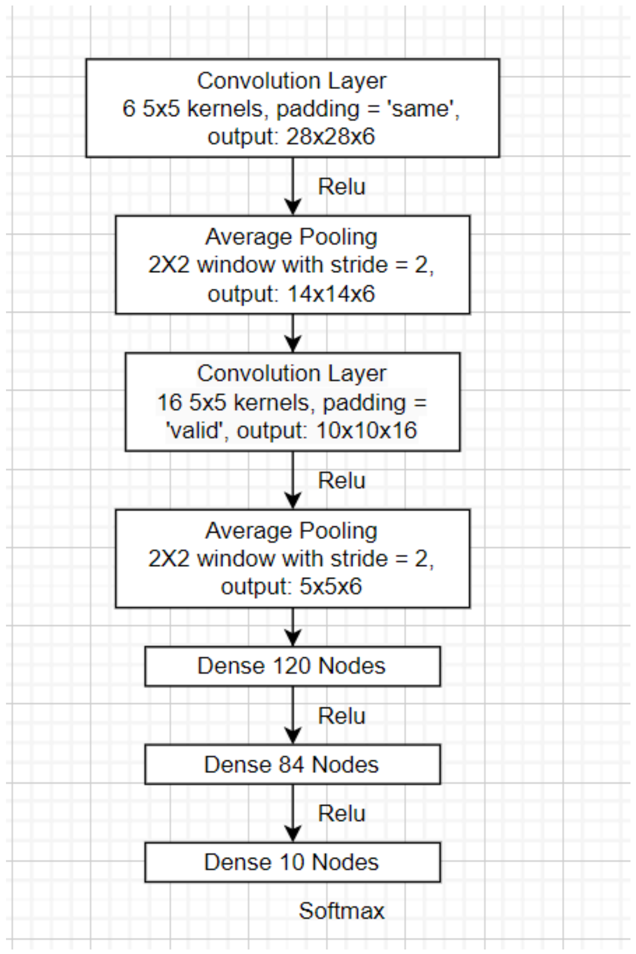
Goal:


LeNet
Image: 28 (height) × 28 (width) × 1 (channel)
Convolution with 5×5 kernel+2padding:28×28×6
sigmoid
Pool with 2×2 average kernel+2 stride:14×14×6
Convolution with 5×5 kernel (no pad):10×10×16
sigmoid
Pool with 2×2 average kernel+2 stride: 5×5×16
flatten
Dense: 120 fully connected neurons
sigmoid
Dense: 84 fully connected neurons
sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes

LeNet uses 2 iterations of convolution, with the sigmoid activation function between each layer, and average pooling.

The Feedforward consists of 3 dense layers also using the sigmoid activation function

Result:


Convolution Layer
6 5x5 kernels, padding = 'same', output: 28x28x6
Relu
Average Pooling
2X2 window with stride = 2, output: 14x14x6
Convolution Layer
16 5x5 kernels, padding = 'valid', output: 10x10x16
Relu
Average Pooling
2X2 window with stride = 2, output: 5x5x6
Dense 120 Nodes
Relu
Dense 84 Nodes
Relu
Dense 10 Nodes
Softmax

The only major difference is the use of the Relu activation function instead of sigmoid.

Relu is generally the best activation function to use, and I was also running into problems getting Sigmoid to work lol.

Training Files

1. **TrainingFeedForward.py**
    a. Using the model without any convolution. Just using a basic FeedForward network
2. **TrainingConvolve.py**
    a. Using the model without pooling. The file has options to use 1 or 2 convolution layers
3. **Training Full.py**
    a. The full LeNet Architecture (As close as I could get😅). Includes Average Pooling and the Convolving process. The file has options to use 1 or 2 convolution layers

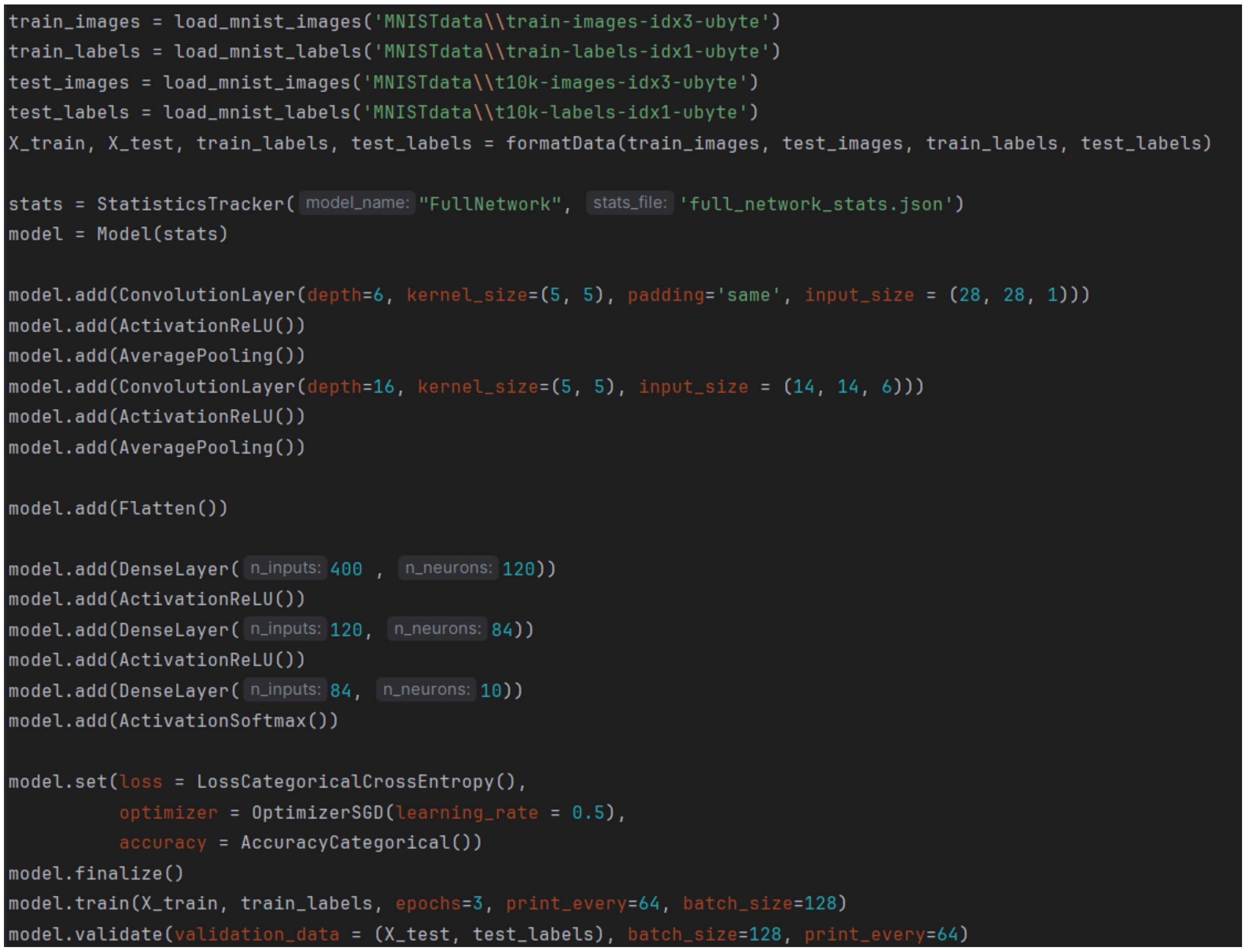
Computation Files

1. **LoadData.py**
    a. Loads the MNIST data and formats it for the model
2. **Model.py**
    a. Model Code. It allows the user to build their neural network using a very similar approach to Keras.
3. **FeedForward.py**
    a. Code for the FeedForward Network. Lots of inspiration from author/Youtuber Harrison Linsley (Sentdex) and his book Neural Networks from Scratch in Python.
4. **Convolution.py**
    a. Code for the Convolution Process. Contains the pooling and flattening layers.
5. **StatisticsTracker.py**
    a. This file keeps track of the average run time of the model's steps, epochs, training, validation, and total runtime
    b. It also keeps track of the average accuracy of each model
    c. Saves everything to a JSON file, and writes averages to a Txt file

Other Files

1. **RunNetworks.py**
    a. Runs a network x times (for training purposes)
    b. I don't recommend using this file when individually training the more complex models since we can run into defect iterations that we won't be able to back out of.
2. **TestConv.py**
    a. My brute force attempt of debugging the conv and pooling layers, and making sure they are computing on the right values

Setup Code



```
train_images = load_mnist_images('MNISTdata\\train-images-idx3-ubyte')
train_labels = load_mnist_labels('MNISTdata\\train-labels-idx1-ubyte')
test_images = load_mnist_images('MNISTdata\\t10k-images-idx3-ubyte')
test_labels = load_mnist_labels('MNISTdata\\t10k-labels-idx1-ubyte')
X_train, X_test, train_labels, test_labels = formatData(train_images, test_images, train_labels, test_labels)

stats = StatisticsTracker( model_name: "FullNetwork",  stats_file: 'full_network_stats.json')
model = Model(stats)

model.add(ConvolutionLayer(depth=6, kernel_size=(5, 5), padding='same', input_size = (28, 28, 1)))
model.add(ActivationReLU())
model.add(AveragePooling())
model.add(ConvolutionLayer(depth=16, kernel_size=(5, 5), input_size = (14, 14, 6)))
model.add(ActivationReLU())
model.add(AveragePooling())

model.add(Flatten())

model.add(DenseLayer( n_inputs: 400 ,  n_neurons: 120))
model.add(ActivationReLU())
model.add(DenseLayer( n_inputs: 120,  n_neurons: 84))
model.add(ActivationReLU())
model.add(DenseLayer( n_inputs: 84,  n_neurons: 10))
model.add(ActivationSoftmax())

model.set(loss = LossCategoricalCrossEntropy(),
          optimizer = OptimizerSGD(learning_rate = 0.5),
          accuracy = AccuracyCategorical())
model.finalize()
model.train(X_train, train_labels, epochs=3, print_every=64, batch_size=128)
model.validate(validation_data = (X_test, test_labels), batch_size=128, print_every=64)
```

Params and Methods:

Model:

- statTracker: Statistics Tracker Object to keep track of runtimes and accuracies of epochs and steps

ConvolutionLayer:

- Depth: Amount of filters you want
- Kernel Size: size of filters
- Padding: padding mode (valid, same, full)
- Input_size: (height, width, depth) of image

AveragePooling:

- window_size: size of window (default 2)

- stride: stride parameter (default 2)

DenseLayer:
- n_inputs: amount of inputs
- n_neurons: the number of neurons in the layer

model.add():
- layer: Adds the layer to the model

model.set():
- loss: Define the loss function for the model
- optimizer: Define optimizer for the model
- accuracy: Define the accuracy tracker for the model

model.finalize():
- layer: Connects all the layers and tracks all trainable layers

model.train():
- X: data to train on
- y: expected values of trained data
- epochs: the number of iterations
- batch_size: size of batches (Default None)
- print_every: will print some stats every x times (Default 100)

model.validate():
- validation_data: (inputs, expected outputs)
- batch_size: size of batches. (Default None)
- print_every: will print some stats every x times (Default 100)

Data and Results

This table documents how fast the model ran and how accurate it was when finished. Note that the model had a lot of run-time variation per Step and Epoch.
Validation is much quicker than training because you only do 1 iteration with no backpropagation.
Run time will vary by computer (obv.)

All this data is pulled from the StatisticsTracker class in each file.
This class puts the results in a data folder. Both on a txt and Json

Networks	Params	Run time	Accuracy (→ means epoch to epoch)

FeedForward	Lr = 0.5
	Epochs = 3
	Batch size = 128
		Step: 0.00 s
		Epoch: 0.75 s
		Training: 2.25 s
		Validation: 0.03 s
		Total: 2.28 s
			Times Ran: 48
			Epoch: 88% → 95% → 97%
			Total: 95%

Convolution (Single Conv)	Lr = 0.5
	Epochs = 3
	Batch size = 128
		Step: 0.15 s
		Epoch: 1.16 min
		Training: 3.5 min
		Validation: 4.98 s
		Total: 3.55 min
			Times Ran: 22
			Epoch: 86% → 95% → 97%
			Total: 96%

Convolution (Double Conv)	Lr = 0.5
	Epochs = 3
	Batch size = 128
		Step: 1.2 - 2 s
		Epoch: 9-17 min
		Training: 35 min
		Validation: 28 s
		Total: 35.4 min
			Times Ran: 5
			Epoch: 70% → 85% → 94%
			Total: 91%

Convolution + Avg. Pooling (Single)	Lr = 0.5
	Epochs = 3
	Batch size = 128
		Step: 1 - 3.5 s
		Epoch: 8 - 15 min
		Training: 30 min
		Validation: 42.23 s
		Total: 31 min
			Times Ran: 3
			Epoch: 85% → 94% → 96%
			Total: 92%

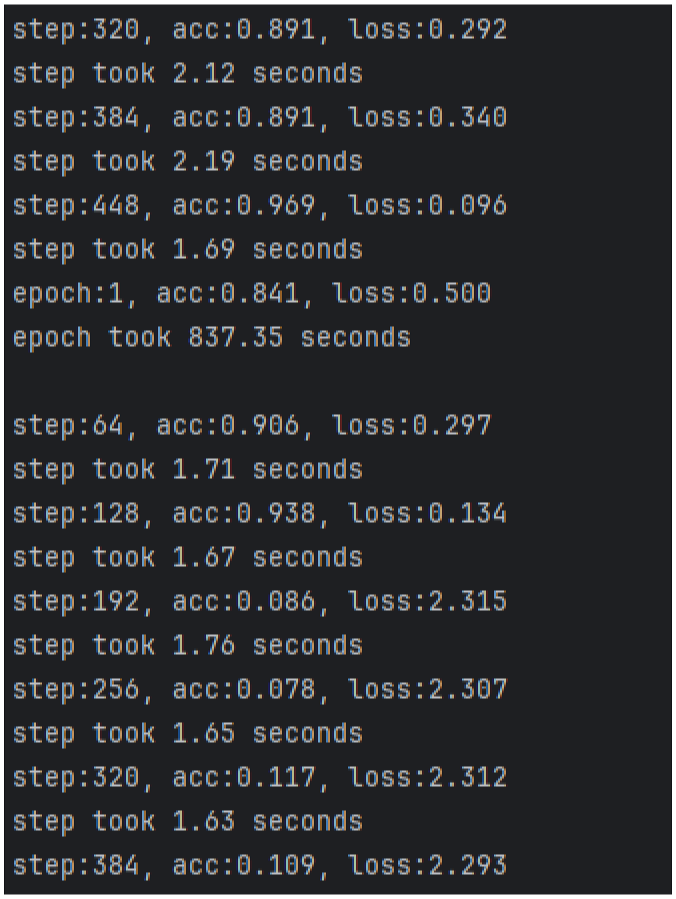
Convolution + Avg. Pooling (Double)	Lr = 0.5
	Epochs = 3
	Batch size = 128
		Step: 2.39 s
		Epoch: 21.86 min
		Training: 1.09 h
		Validation: 2.4 min
		Total: 1.1 h
			Times Ran: 9
			Epoch: 49% → 72% → 84%
			Total: 81%

Note that if running one of the more complex models, you notice that it's not learning within the first epoch (using print_every), then restart the model! It will most likely not fix itself within the 3 iterations. Some reasons for it breaking are below

Why do more Convolution layers lead to slower and lower accuracy in this model?

When building CNNs from scratch, especially with just NumPy, you will likely encounter performance issues as the network's complexity increases. This is expected, as libraries like TensorFlow or PyTorch use highly optimized backend computations.

There are also cases where we reach defect iterations. These are iterations that don’t learn anything, or after progressive learning, end up unlearning everything, and never fix themselves. Here is an example



```
step:320, acc:0.891, loss:0.292
step took 2.12 seconds
step:384, acc:0.891, loss:0.340
step took 2.19 seconds
step:448, acc:0.969, loss:0.096
step took 1.69 seconds
epoch:1, acc:0.841, loss:0.500
epoch took 837.35 seconds

step:64, acc:0.906, loss:0.297
step took 1.71 seconds
step:128, acc:0.938, loss:0.134
step took 1.67 seconds
step:192, acc:0.086, loss:2.315
step took 1.76 seconds
step:256, acc:0.078, loss:2.307
step took 1.65 seconds
step:320, acc:0.117, loss:2.312
step took 1.63 seconds
step:384, acc:0.109, loss:2.293
```

Notice how Epoch 1 is pretty accurate, and the beginning of Epoch 2 carries this on, but after a while, we suddenly drop in accuracy by a lot. The model never rebounds from this. I assume it is either because of overfitting, or some edge case computation that sets our weights/biases/dinputs to unexpected values.

This is something I hope to fix in the future.

Here are a few reasons why adding convolutional layers or pooling may result in slower learning and less accurate results:

Increased Complexity: Each additional convolutional layer adds a significant number of operations, especially when using nested loops for convolution.

Vanishing Gradients: Deeper networks can suffer from vanishing gradients, where gradients become increasingly smaller as they propagate back through layers, making learning SLOW.

Parameter Tuning: More layers mean more parameters to tune, including the number of filters, kernel size, and learning rate. It can be harder to find the right combination that leads to efficient learning.

Optimization Difficulty: The architecture behind LeNet is pretty outdated. Without advanced optimization techniques (like momentum, Root Mean Square Propagation, or Adam), SGD can be very slow to converge, especially in deep or complex architectures.

Data Representation: Deeper layers capture more abstract representations. If the first layer hasn't learned good features, the second layer has to work with poor inputs, which can degrade performance. This is why 2 convolutions perform worse than 1 convolution

If we were to do more than 3 iterations for the more complex networks, they would become just as accurate as the feedforward network or the one convolution network. But I wanted to show that the increase of parameters leads to slower learning without more up-to-date optimization techniques (also because I am impatient)

Further Work

I plan on doing more research on why the model ends up breaking. This will just be an in-depth look at what’s happening with the weights and biases when my accuracies aren’t increasing, or suddenly decrease a lot. With this information, I will probably add some extra features such as reducing the learning rate as the training progresses, regularization, etc.

I also want my stats tracker to flag these defect iterations, and not include them (at user’s discretion) in the model’s statistics. I could use this to track the weights and biases and track them separately from the good runs.

References

- https://www.rose-hulman.edu/class/cs/csse413/assignments/cnn/Lecun98.pdf (LeCun's article on using CNNs for MNIST recognition)
- https://nnfs.io/ (Textbook on Neural Networks from Scratch in Python)
- https://www.youtube.com/watch?v=Wo5dMEP_BbI (Some YouTube videos on the above book)
- https://en.wikipedia.org/wiki/File:Comparison_image_neural_networks.svg (LeNet’s architecture)
- https://www.youtube.com/watch?v=xvFZjo5PgG0 (Visual on how CNNs learn)
- https://www.youtube.com/watch?v=z9hJzduHToc (Math behind CNN backpropagation)
- https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/ (Article about pooling)
- https://lanstonchu.wordpress.com/2018/09/01/convolutional-neural-network-cnn-backward-propagation-of-the-pooling-layers/ (Article about calculating gradients for pooling)
- https://www.youtube.com/watch?v=Lakz2MoHy6o (Another video about convolution)
- https://social.mtdv.me/watch?v=-0X4TxqEYh (The most helpful link. Very in-depth exclamation about CNNs and how to use numpy to make everything efficient and easy 🕺)