Table of Contents

Introduction

During Winter Quarter 2021 at California Polytechnic State University at San Luis Obispo, I did an independent study under Professor Lubomir Stanchev. The target of the independent study was to implement a convolutional neural network in java comparable to the neural network

functionality supported by the keras package in Python. This report outlines how to the network, and the details of my implementation. All items referred to in this report can be found in this github repository.

External Documentation

Using the library

To run a java program using the network, these libraries must all be included in the classpath. The library "cnn.jar" contains all the functionality needed by the NeuralNetwork class. The folder of libraries called "matMulLibs" contains matrix multiplication libraries from EJML. In the linked GitHub repository, there is an example of an application of the CNN to the MNIST handwritten digits dataset. I will cover how to run the software with and without the InteliJ IDE.

With IntelliJ

To use this software in IntelliJ, you will first need to have the necessary libraries installed somewhere on your machine. In IntelliJ, navigate to File -> Project Structure -> Libraries. In the pane to the right of the Project Settings pane, click the plus icon at the top. Then, navigate to and choose the cnn.jar file. After adding the cnn library, click the plus icon and chose the "matMultLibs" folder. After completing these two steps the NeuralNetwork class should work as intended.

Without Intellij

To use this software without IntelliJ, the necessary libraries need to be included in the classpath when compiling and running the software. In the digit recognition example, the libraries are stored in a folder in the same directory as the DigitRecognition source file. Also, note that the data used to train the network is compressed upon downloading. For the program to run as intended, the data.zip file must be uncompressed. Then, in the root folder of the example directory, run the following two commands to compile and run the program.

```
javac -cp .:./libs/cnn.jar:./libs/matMulLibs/* *.java
java -cp .:./libs/cnn.jar:./libs/matMulLibs/* DigitRecognition
```

Both of these commands include the necessary libraries in the class path.

Using the NeuralNetwork Object

Steps:

1. Build the network by adding layers.
2. Compile the network.
3. Train the network.

Note: You must import:

1. UI.NeuralNetwork

It may be necessary to import:

2. Sequential.LossFunctions.*
3. Sequential.Optimizers.*

4. Sequential.Exceptions.*

Adding Layers

These are the functions to use when adding layers to the network. Note, the underlined parameters are required, and the rest are generally optional. See the javadoc for complete information. The first layer added to the network must always be given an inputSize. If no activation function is provided while adding a convolutional or dense layer, and no activation layer is added after one of these layers, then one will be added automatically (relu if not last layer, softmax if last layer).

addConv (numFilters, filterDims, inputSize, strideLength, actFunc)

- **numFilters**: integer representing the number of filters the layer should use.
- **filterDims**: two element integer array indicating the height of the filter with the first element and the width of the filter with the second element.
- **inputSize**: three element integer array indicating the depth, height, and width of the input.
- **strideLength**: two element integer array indicating the horizontal and vertical stride for moving the filters. Default is [1, 1]
- **actFunc**: String indicating which layer should follow the convolutional layer. If it is not supplied, and an activation layer is not explicitly added after this layer, a Relu layer will be added to the network.

addMaxPool (poolDims, inputSize, strideLength)

- **poolDims**: two element integer array indicating the height and width of the pool.
- **inputSize**: three element integer array indicating the depth, height and width of the input.
- **strideLength**: two element integer array indicating the horizontal and vertical stride for moving the filters. Default is to use the poolDims for the strideLengths.

addDense (numNodes, inputSize, actFunc)

- **numNodes**: integer representing the number of nodes that should be in the layer.
- **inputSize:** three element integer array indicating the depth, height, and width of the input.
- **actFunc**: String indicating which layer should follow the convolutional layer. If it is not supplied, and an activation layer is not explicitly added after this layer, a Relu layer will be added to the network.

The functions for adding activation layers to the network are:
addRelu()
addSigmoid()
addSoftmax()

Compiling

The function for compiling the network is below. None of these parameters are required. See the javadoc for complete information.

Compile(lossFunc, opt, metrics)

- **lossFunc**: Object that implements the LossFunction interface that the network will use during training.
- **opt**: Object that implements the Optimizer interface that will be used to train the network.
- **metrics**: Array of Strings indicating which metrics the network should track during training and evaluation.

Training

The function for training the network is

Fit(x, y, batchSize, epochs)

- **x**: Four-dimensional array of doubles. Can be thought of as an array of three-dimensional inputs.
- **y:** Two-dimensional array of doubles representing an array of the desired outputs.
- **batchSize**: Number of inputs that should be passed through the network at a time during training.
- **epochs**: Number of times the training data should be used to train the network.

Evaluating

The function for evaluating the network is

Evaluate(x, y)

- **x**: Four-dimensional array of doubles. Can be thought of as an array of three-dimensional inputs.
- **y:** Two-dimensional array of doubles representing an array of the desired outputs.

Predicting

Predict(x)

- **x**: Four-dimensional array of doubles. Can be thought of as an array of three-dimensional inputs.

Loading/Storing

Function for loading:

Load(path)

- **path**: String representing the path to the location of the serialized network to be read.

Function for saving:

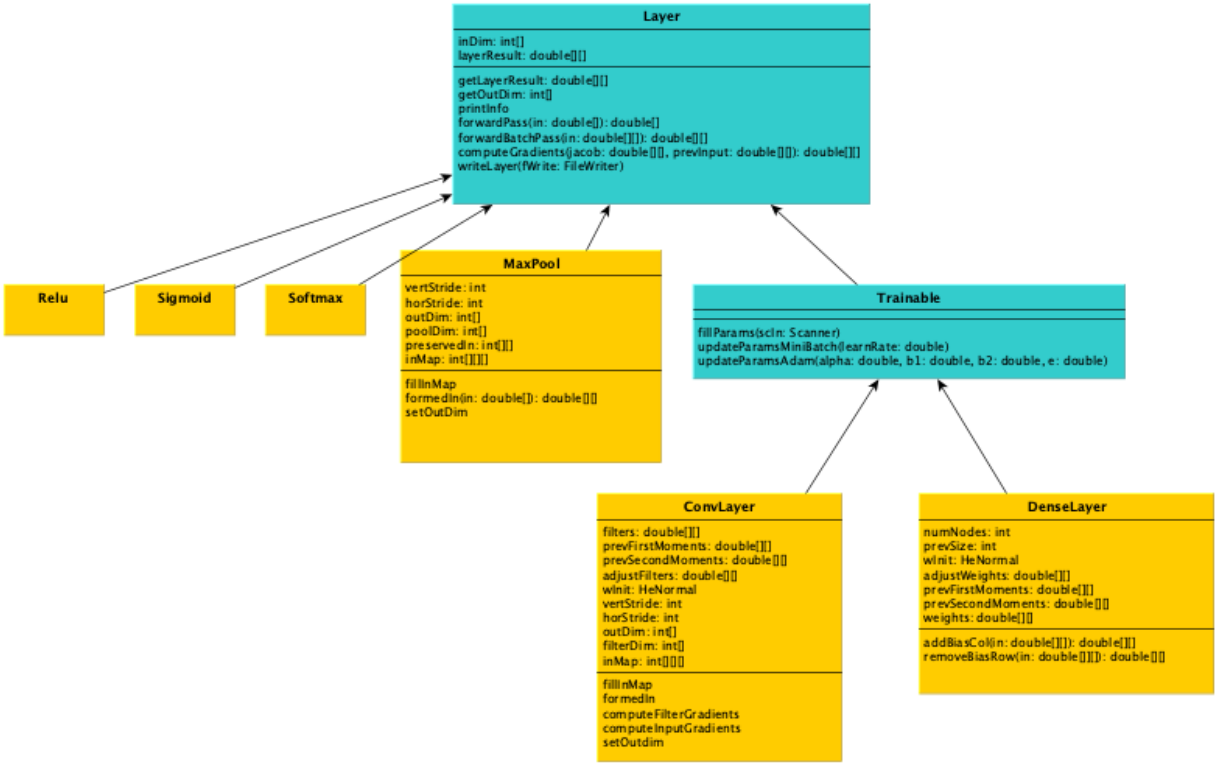
Save(path)

- **path**: String representing the location where the network should be serialized.

Internal Documentation

Structure

The data structure used to represent the network is an ArrayList of Layer objects. A Layer is an abstract class meant to represent an operation performed on a set of inputs to map them to a set of outputs. The hierarchy for layers is shown below.



As you can see, activation functions are treated as layers in the network. Also, layers that have trainable parameters extend the abstract class “Trainable.”

Forward Pass

A forward pass is made through the network by iteratively calling either “forwardPass,” or “forwardBatchPass” on every layer.

```
// Perform a foward pass of a batch through the network
public double[][] forwardBatchPass(double[][] input) throws InvalidDimensionException{
    for (int i = 0; i < layers.size(); i++){
        input = layers.get(i).forwardBatchPass(input);
    }
    return input;
}
```

Of course, the operations required get more complex inside each of the layers. The operations required for forward pass and forward batch pass are outlined below for each layer.

Dense Layer

Forward Pass

- W: $(m+1)$ x n = weight matrix for a dense layer with n neurons, and m inputs. The last row of the matrix contains the biases of the layer.
- X: 1 x m = Vector of inputs

$$X^* = [X, 1]$$
$$result = X^*W$$

- Resulting dimensions: 1 x n

Forward Batch Pass

- W: $(m+1)$ x n = weight matrix for a dense layer with n neurons, and m inputs. The last row of the matrix contains the biases of the layer.
- X: b x m = Matrix of inputs with batch size, b.

$$X^* = [X, 1]$$
$$result = X^*W$$

- Resulting dimensions: b x n

Convolutional Layer

InMap Operation:

Before performing the calculations involved in the convolutional layer, I reformat the input by simulating moving filter through the input. I store the locations of the input in the reformatted input in an array that I call "inMap." The reformatted input is structured as a matrix where each row represents the input values covered by a filter to make computation of the layer result easier. A visual is shown below for a 2d input for simplification. The actual implementation handles 3d input.

Input -> Formatted input (with filter size 2x2, and stride length 1x1):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{21} & a_{22} & 1 \\ a_{12} & a_{13} & a_{22} & a_{23} & 1 \\ a_{21} & a_{22} & a_{31} & a_{32} & 1 \\ a_{22} & a_{23} & a_{32} & a_{33} & 1 \end{bmatrix}$$

Input | Formatted Input

$$\begin{bmatrix} a_{11} & [0,0] & & & \\ a_{12} & [0,1] & [1,0] & & \\ a_{13} & [1,1] & & & \\ a_{21} & [0,2] & [2,0] & & \\ a_{22} & [0,3] & [1,2] & [2,1] & [3,0] \\ a_{23} & [1,3] & [3,1] & & \\ a_{31} & [2,2] & & & \\ a_{32} & [2,3] & [3,2] & & \\ a_{33} & [3,3] & & & \end{bmatrix}$$

InMap

Forward Pass:

- F: d x f = Matrix of layer parameters. d is equal to the number of parameters in a single filter which is one plus the product of the filter width, filter height, and input depth. f is equal to the number of filters in the layer.
- X: 1 x m = Vector of inputs.
    - X^*: p x d = Formatted matrix of inputs. p is equal to the number of positions the filter occupied when iterating over the layer. Each row represents the input values covered by a filter.

$$result = flatten(X^*F)$$

- Resulting dimensions: 1 x n where n is equal to p * f.

Forward Batch Pass:

- F: d x f = Matrix of layer parameters. d is equal to the number of parameters in a single filter which is one plus the product of the filter width, filter height, and input depth. f is equal to the number of filters in the layer.
-
- X: b x m = Matrix of inputs with batch size, b.
    - X_i^*: p x d = Formatted matrix corresponding to a single input, i, in the batch. p is equal to the number of positions the filter occupied when iterating over the layer.

$$result_i = flatten(X_i^*F)$$

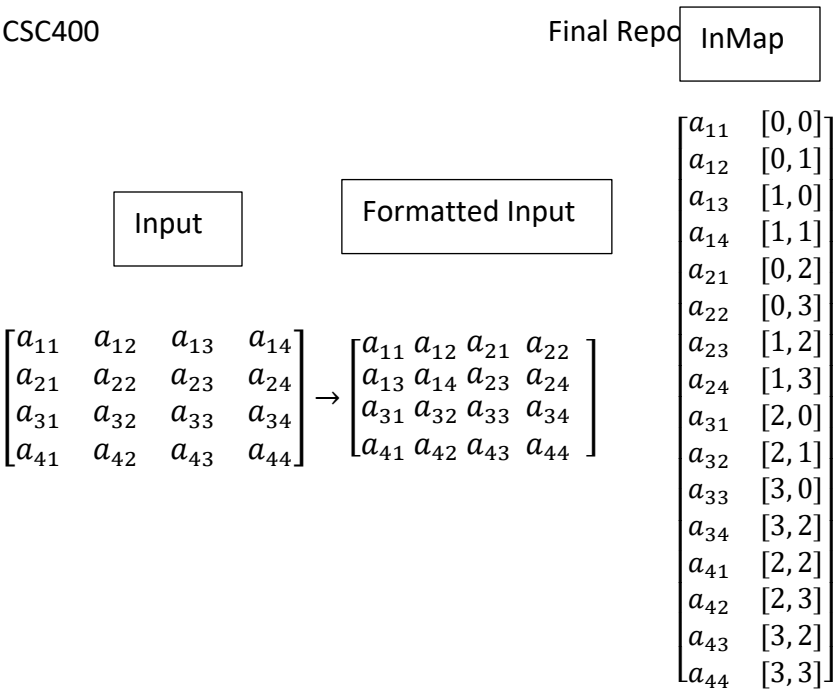
- Resulting dimensions: b x n where n is equal to p * f.

MaxPool

InMap Operation:

Before performing the calculations involved in the maxpool layer, I reformat the input by simulating moving pool through the input. I store the locations of the input in the reformatted input in an array that I call “inMap.” The reformatted input is structured as a matrix where each row represents the input values covered by a pool to make computation of the layer result easier. A visual is shown below for a 2d input for simplification. The actual implementation handles 3d input.

Input -> Formatted input (with pool size 2x2, and stride length 2x2):



Forward Pass:

- X: 1 x m = Vector of inputs.
  - X^*: p x d = Formatted matrix of inputs. p is equal to the number of positions the pool occupied when iterating over the layer. d is equal to the product of the width of the pool and the height of the pool

$$result = rowmax(X^*)^T$$

- Resulting dimensions: 1 x p.

Forward Batch Pass:

- X: b x m = Matrix of inputs with batch size, b.
  - X_i^*: p x d = Formatted matrix corresponding to a single input, i, in the batch. p is equal to the number of positions the pool occupied when iterating over the layer. d is equal to the product of the width of the pool and the height of the pool.

$$result_i = rowmax(X_i^*)^T$$

- Resulting dimensions: b x p.

Relu

Forward Pass:

- X: 1 x m = Vector of inputs

$$result_i = \max(X_i, 0)$$

- Resulting dimensions: 1 x m.

Forward Batch Pass:

- X: b x m = Matrix of inputs with batch size, b.

$$result_{i,j} = \max(X_{i,j}, 0)$$

- Resulting dimensions: b x m.

Sigmoid

Forward Pass:

- X: 1 x m = Vector of inputs.

$$sigmoid(X_i) = \frac{1}{1 + e^{-X_i}}$$

Resulting dimensions: 1 x m.

Forward Batch Pass:

- X: b x m = Matrix of inputs with batch size, b.

$$sigmoid(X_{i,j}) = \frac{1}{1 + e^{-X_{i,j}}}$$

- Resulting dimensions: b x m.

SoftMax

Forward Pass:

- X: 1 x m = Vector of inputs.

$$softmax(X)_i = \frac{e^{X_i}}{\sum_{j=1}^{m} e^{X_j}}$$

- Resulting dimensions: 1 x m.

Forward Batch Pass:

- X: b x m = Matrix of inputs with batch size, b.
  - X_k: 1 x m = The k^{th} vector of inputs in X

$$softmax(X_k)_i = \frac{e^{X_{k_i}}}{\sum_{j=1}^{m} e^{X_{k_j}}}$$

- Resulting dimensions: b x m.

Gradients

Given the gradients of the loss with respect to each of a layer's batch of outputs, a layer needs to be able to:

1. Calculate the gradients of the loss with respect to each of the layer's inputs.
2. Calculate the gradients of the loss with respect to the layer's parameters.

Loss Function

Before calculating the gradients within the layers of the network, we need to calculate the gradients of the loss function with respect to the outputs of the last layer of the network.

Mean-Squared Error

Calculation:

- X: b x m = Matrix of last layer outputs with batch size, b.
- Y: b x m = Matrix of expected outputs. Each row represents one expected output vector.

$$meanSquared(X,Y) = \frac{1}{b}\sum_{i=1}^{b}\frac{1}{m}\sum_{j=1}^{m}(Y_{ij} - X_{ij})^2$$

- Resulting dimensions: scalar.

Gradient:

$$\frac{d(meanSquared)}{dX_{ij}} = \frac{-2(Y_{ij} - X_{ij})}{bm}$$

- Resulting dimensions: b x m.

Cross Entropy

Calculation:

- X: b x m = Matrix of last layer outputs with batch size, b.
- Y: b x m = Matrix of expected outputs. Each row represents one expected output vector.

$$crossEntropy(X,Y) = \frac{1}{b}\sum_{i=1}^{b}\sum_{j=1}^{m} -Y_{ij}\log(X_{ij})$$

- Resulting dimensions: scalar.

Gradient:

$$\frac{d(crossEntropy)}{dX_{ij}} = \frac{1}{b}\sum_{i=1}^{b}\sum_{j=1}^{m}\frac{-Y_{ij}}{X_{ij}\ln(10)}$$

- Resulting dimensions: b x m.

Dense Layer

Inputs

- O: b x n = Matrix containing gradients of loss with respect to a batch of outputs with batch size, b.
- W: $(m+1)$ x n = Matrix containing the parameters for the layer. m is the number of inputs to the layer, and n is the number of outputs of the layer.

$$result = OW^T$$

- Resulting dimensions: b x m.

Params

- X: b x m = Matrix containing the batch of inputs given to the layer.

$$X^* = [X, 1]$$
$$result = X^{*T}O$$

- Resulting dimensions: $(m + 1)$ x n.

Convolutional Layer

Inputs

- F: d x f = Matrix of layer parameters. d is equal to the number of parameters in a single filter which is one plus the product of the filter width, filter height, and input depth. f is equal to the number of filters in the layer.
- O: b x n = Matrix containing gradients of loss with respect to a batch of outputs with batch size, b.
    - O_i^*: p x f = Matrix representing the gradient of the loss with respect to the i^{th} output in the batch. p is equal to the number of positions the filter occupied when iterating over the layer. Each column represents the values corresponding to a certain filter.

$$I_i^* = O_i^* F^T$$
$$I_i = reverseMapSum(I_i^*)$$

- $reverseMapSum$: Using the input map, for each input, iterate through its locations in the formatted input whose gradient is represented by I^*. Accumulate the values at said locations in I^* to achieve the gradient of the input.
- Resulting dimensions: b x m

Params

- X: b x m = Matrix containing the batch of inputs given to the layer.
    - X_i^*: p x d = Matrix representing the i^{th} input from the given batch of inputs. Each row of the matrix represents one position of the filter.

$$result_i = X_i^{*T} O_i^*$$
$$result = \sum_{i=1}^{b} result_i$$

- Resulting dimensions: d x f

Maxpool Layer

Inputs

- O: b x n = Matrix containing gradients of loss with respect to a batch of outputs with batch size, b.
    - O_i: 1 x n = Vector representing the gradient of the loss with respect to the i^{th} output in the batch. n is equal to the number of positions the pool occupied when iterating over the layer. Each column represents the values corresponding to a certain filter.

```
for input in inMap:
   sum = 0
   for formatLoc in input:
       if (formatLoc was max):
           sum += Oi[formatLoc]
```

- This is some psudocode for the maxpool backpass operation. When we perform the forward pass we store a 2D array of size b x n. This matrix stores the index in the row that was the maximum from the rowmax operation. For each inputs location in the formatted (mapped input), if that mapped input was the maximum, then add the value from O_i to the input's partial derivative.
- Resulting dimensions: b x m, where m is the size of an input example to the layer.

Relu Layer

- O: b x n = Matrix containing gradients of loss with respect to a batch of outputs with batch size, b.
- X: b x n = Matrix containing the batch of inputs given to the layer.

$$result = \begin{cases} O_{i,j} & if\ X_{i,j} > 0 \\ 0 & if\ X_{i,j} \leq 0 \end{cases}$$

- Resulting dimensions: b x n.

Sigmoid Layer

- O: b x n = Matrix containing gradients of loss with respect to a batch of outputs with batch size, b.
- X: b x n = Matrix containing the batch of inputs given to the layer.

$$result_{i,j} = O_{i,j} * \frac{e^{-X_{i,j}}}{(1 + e^{-X_{i,j}})^2}$$

- Resulting dimensions: b x n.

Softmax Layer

- O: b x n = Matrix containing gradients of loss with respect to a batch of outputs with batch size, b.
  - O_k: 1 x n = = Vector representing the gradient of the loss with respect to the k^{th} output in the batch.
- L: bxn = Layer result. b is batch size, n is output size.
- S_k: nxn = Internal partial derivative matrix. It is the partial derivative of the ith softmax result with respect to the jth input in the kth set of the batch.

$$S_{k_{i,j}} = \begin{cases} -L_{k_i} * L_{k_j} & if\ i\ != j \\ L_{k_i} * (1 - L_{k_j}) & if\ i = j \end{cases}$$

$$result_k = O_k S_k$$

- Resulting dimensions: b x n.

Optimizers

The main difference for the following two optimizers, adam and mini-batch, is the method in which layer parameters are updated. Besides that, they perform very similar operations. It looks something like this:

```
for (epoch in epochs):
   for (batch in batches):
      network.forwardBatchPass(batch)
      jacobian = getLossJacobian()
      for (layer in reverse(layers)):
         jacobian = layer.getJacobian(jacobian, previousLayerResult)
         layer.updateParameters()
```

Mini-Batch

Parameters are updated according to the following formula.

$$weight = weight - learnRate * \frac{dL}{dweight}$$

Therefore, the only parameter a minibatch optimizer takes is the $learnRate$.

Adam

An Adam optimizer needs more parameters than mini batch. Specifically, it needs:
$alpha$: The initial learn rate
$beta1$: First moment momentum
$beta2$: Second moment momentum
$epsilon$: Small value to prohibit divide by zero division

The idea of an Adam optimizer is to adjust the weights according to the first and second moments of the gradient of the weight.

$$g = \frac{dL}{dWeight}$$

$$b_1^* = beta1^{current\ epoch}$$

$$b_2^* = beta2^{current\ epoch}$$

$$a^* = alpha * \frac{\sqrt{1 - b_2^*}}{1 - b_1^*}$$

$$m_1 = b_1^* * m_{1_{prev}} + (1 - b_1^*) * g$$

$$m_2 = b_2^* * m_{2_{prev}} + (1 - b_2^*) * g^2$$
$$weight = weight - a^* * \frac{m_1}{\sqrt{m_2 + epsilon}}$$

Note: $m_{1_{prev}}$ and $m_{2_{prev}}$ can be initialized to 0.

Conclusion

I am very glad I did this project this quarter. This was the most challenging work I've done so far at Cal Poly, and I learned so much. I want to thank Professor Stanchev for advising me throughout the process and giving me the resources I needed to succeed. I hope to continue to learn about machine learning methods, and to continue challenging myself as I did this quarter.