**JavaFX Convolution Network**

**Introduction**

The purpose of this project is to learn the implementation details of a convolution network application programmed with Java and JavaFx. It is assumed you already know how to install the Java software. Even though this sample implements a network, it is not intended to be reusable. Normally, you would use a library such as Deeplearning4j for building a production neural network. If no such library was available, how would you implement a neural network from scratch?

This document acts a user guide, and includes a simplistic overview of neural network theory. To actually learn the detials, you must walk through the code line by line.

**Prerequisites**

Understanding neural networks requires knowledge of matrix algebra, partial derivatives, and statistics. Previous experience with linear regression or multilinear regression is also helpful. The references below are for refreshing your memory. To really learn the math, you should have completed classes in each of the topics mentioned above.

Some references:

<https://www.quantstart.com/articles/scalars-vectors-matrices-and-tensors-linear-algebra-for-deep-learning-part-1/>

<https://www.quantstart.com/articles/matrix-algebra-linear-algebra-for-deep-learning-part-2/>

<https://www.quantstart.com/articles/matrix-inversion-linear-algebra-for-deep-learning-part-3/>

<https://towardsdatascience.com/matrix-calculus-for-data-scientists-6f0990b9c222>

<https://online.stat.psu.edu/stat462/node/132/>

<https://reliawiki.org/index.php/Multiple_Linear_Regression_Analysis>

The application code wasdivided into front end and back end sections, as shown in the diagram.

Chart

Description automatically generated with medium confidence

See the pom.xml and package.json files for the versions of each component in the application. These include three jackson jar files for json processing, which should be copied to the lib folder to allow local access in the batch file.

The Java 18 JDK may be downloaded from:

https://jdk.java.net/18/ ->

JDK 18.0.2.1 download

The JavaFx 18 SDK may be downloaded from:

https://gluonhq.com/products/javafx/ ->

18.0.2 SDK download

The IntelliJ IDE was updated with Java JDK 18.0.2.1 and library JavaFx SDK 18 to build the jar file. The windows batch file, FxConvoNet.bat, may be used to run the application from the jar file.

You must download the MNIST dataset, which contains thousands of images of handwritten digits 0 to 9. The png image format used by this application was downloaded from: https://github.com/myleott/mnist\_png.

The file structure will look like the following, where the digit images are separated into individual folders 0, 1, ... 9.

A picture containing chart

Description automatically generated

The advantage of having separate folders is to allow the user to examine specific digits. Another advantage of having separate folders is that we can quickly observe how many images there are for each digit.

Note that even though the total number of training images is over 40,000, the individual digit folders do not contain the same number of files. The folders for digits 4 and 5 contain only about 2700 images, while the digit 7 folder contains about 6200 images. Therefore, to avoid biased training results, the total number of samples to load should not exceed 27000, 2700 from each folder. This will ensure reading an equal number of images from each traning digit folder.

Similarly for the testing dataset, the digit folders do not contain equal numbers of images. The digit 5 folder contains 892 images, while digit 1 folder contains 1135 images. Therefore, to avoid biased testing results, the total number of samples to load should not exceed 8900, 890 from each folder. This will ensure reading an equal number of images from each testing digit folder.

After reading the separate image files, they are combined and randonly shuffled before the network training or testing task begins.

**Code Structure**

The project is a desktop application which assumes a wide screen size. The front end provides a user interface written in JavaFx, while the back end is launched via a JavaFx concurrent task. There is no server involved. For Java programmers who have not used JavaFx, the sample code provides examples of a menu bar, tab panel, data entry form, concurrent task, and output charts. The back end code includes network layers, activation functions, a matrix library, and json utilities.

**Frontend**

The front end consists of several JavaFx screens. The main program is FxConvoMain, which may be invoked with the windows batch file: FxConvoNet.bat. The startup code sequence outline is:

FxConvoMain

main

Application.launch

start

initLogging

initComponents

tabs.initPanel

fxMenu.initViewer

fxMenu.initMenuBar

initMenuApp

initSettingsMenu

initOptionMenu

initHelpMenu

stage.setScene

stage.show

The Setup menu has options for creating, exporting, and importing the network configuration.

Graphical user interface, text, application

Description automatically generated

The Setup menu -> Import Network Configuration opens the file dialog to read in the network configuration file, which is in json format. Then the network config screen opens in a new tab, Import Config, with all fields populated from the json file. The code sequence outline is:

AppMenu. initSettingsMenu

configView.importConfig

ViewUtil.openFileDialog

FileUtil.getInputStream

JsonUtil.jsonToConfig

configView.setAllLayers

initNetConfigPanel

createGeneralConfig

createInputConfig

createConvoPoolConfig

createInternalConfig

createOutputConfig

The user may create a new network configuration as follows. Click on Setup menu -> Create Network Configuration to open the empty network config screen in a new tab, Create Config.

Graphical user interface, text, application

Description automatically generated

The code sequence outline is:

AppMenu. initSettingsMenu

configView.createAllLayers

initNetConfigPanel

createGeneralConfig

createInputConfig

createConvoPoolConfig

createInternalConfig

createOutputConfig

The Import config panel contains all the basic information to perform a network training run. General parameters includes the directories where the image data is located, number of samples to load, and back propagation parameters. The gradient descent sub-panel includes minmum rate, maximum rate, rate decay parameters, and type of rate decay function: triangle or step.

Graphical user interface, text, application, email, website

Description automatically generated

The Input layer panel has input fields for image pixel height and width which is necessary for matrix sizing, described in the back end section.

Graphical user interface, application, Teams

Description automatically generated

The Convolution/Pool Layer panel contains sizing information, and options to choose the action function. The Pool activation function is set to None and cannot be changed.

Graphical user interface, application

Description automatically generated

The Internal Layer panel has fields for activation function and output node size. The input node size is determined by the previous layer output matrix size.

Graphical user interface, text, application, email

Description automatically generated

The Output Layer panel has fields for activation function and output node size, which equals 10 for the digit images.

Graphical user interface, application

Description automatically generated

After the entering the parameters, the user may export the network configuration. The code sequence outline is:

AppMenu. initSettingsMenu

configView.exportConfig

ViewUtil.openCreateFileDialog

FileUtil.getOutputStream

JsonUtil.configToJson

The Process menu -> Display sample item opens a file dialog to allow the user to select an image to examine.

Graphical user interface, text

Description automatically generated with medium confidence

The code sequence outline is:

AppMenu.initOptionMenu

pixelView.displayImage

imageFileDialog

initData

ImageDataUtil.loadImageData

viewData

imageView.setImage

ImagePane(imageMatrix)

The actual image is shown on the left of the screen. On the Normalized Image Grid, the pixel values are displayed after being normalized 0.0 to 1.0, where 0 is black and 1 is white. Values less than 0.25 are not displayed.

Chart

Description automatically generated

The Process menu -> Train the Network item opens the network task screen under a new tab, Train.

Graphical user interface, text

Description automatically generated

Graphical user interface, application, table

Description automatically generated

The buttons at the top of the network task screen allow the user to run the network task, cancel a running task, and to take a snapshot of the performance charts.



Currently, there are three screens that display network performance information. The Cumulative Performance Plot and Performance Hitogram are updated periodically as the task is running. The Summary Confusion Matrix is available after the run has completed.

Chart

Description automatically generated

Chart, bar chart, histogram

Description automatically generated

Chart, scatter chart

Description automatically generated

After the training run has completed, the user may export the training model parameters, which includes all weights and biases from each layer. See the Setup menu -> Export Training Model fit Parameters.

**Graphical user interface, text, application, chat or text message

Description automatically generated**

The training model parameters may be imported before running the test cases. The testing dataset should include a different set of images than the training dataset.

**Backend**

The back end implements a convolution network for image classification on the MNIST dataset, which consists of several thousand images of handwritten digits 0 to 9.

References:

ref: https://www.analyticsvidhya.com/blog/2020/02/mathematics-behind-convolutional-neural-network/

ref: <https://towardsdatascience.com/deriving-the-backpropagation-equations-from-scratch-part-2-693d4162e779>

eta triangle function:

ref: https://www.jeremyjordan.me/nn-learning-rate/

**Model Notation**

In each layer:

X = input matrix (e.g. matrix of image pixel values)

W = weight matrix

B = bias matrix

Z = W \* X + B

S = activation function (e.g. sigmoid, RELU, softmax)

Y = S(Z)

Y = output matrix

L = loss function (e.g. sum square error, log likelihood)

Note: for convolution layer, the weight matrix W is replaced with the filter matrix f, and matrix multiplication is replaced by matrix convolution.

**Matrix Sizes**

Input Layer:

(Rows, columns) of input image = (n, n)

n = image size

Convolution Layer:

filter matrix = (f,f),

f = filter size

let nf = n - f + 1

convolution output matrix = ( nf , nf )

Pool Layer:

pool matrix = (p,p)

let nfp = (n-f+1) / p = nf / p

pool output matrix = ( nfp , nfp )

number of pool output nodes:

pOut = nfp \* nfp \* numFilters

Internal Layer:

internal layer input matrix = (pOut,1)

internal layer output matrix = (iOut, 1)

Output Layer:

"output layer" input matrix = (iOut, 1)

"output layer" output matrix = (oOut, 1)

Example:

Input Layer:

input image = (28, 28), n = 28

1st Convolution Layer:

filter matrix = (5,5), f = 5

nf = 28 - 5 + 1 = 24,

convolution output matrix = (24 , 24)

numFilters = 20

1st Pool Layer:

pool matrix = (2,2), p = 2

nfp = nf/2 = 24/2 = 12,

pool output matrix = (12 , 12)

pool output nodes:

pOut = 12 \* 12 \* 20 = 2880

2nd Convolution/Pool Layer:

filter matrix = (5,5), f = 5

nf = 12 - 5 + 1 = 8,

convolution output matrix = (8 , 8)

numFilters = 50

2nd Pool Layer:

pool matrix = (2,2), p = 2

nfp = nf/2 = 8/2 = 4,

pool output matrix = (4 , 4)

pool output nodes:

pOut = 4 \* 4 \* 50 = 800

Internal Layer:

internal layer input matrix, (pOut,1) = (800, 1)

internal layer output matrix, (iOut, 1) = (500, 1)

Output Layer:

output layer input matrix, (iOut, 1) = (500, 1)

output layer output matrix, (oOut, 1) = (10, 1)

Because the neural network makes extensive use of matrix algebra, a matrix library is an essential tool for the application. Because the weight matrix, W, may contain hundreds of rows and columns, the matrix library should operate very efficiently on large matrices. The following reference compares the performance of selected matrix libraries: EJML, ND4J, Apache Commons Math, LA4J, and Colt. ND4J performed best for very large matrix multiplication. (Note: ND4J is part of the Deeplearning4j framework)

<https://www.baeldung.com/java-matrix-multiplication>

In the neural network literature, matrix flattening is often mentioned for the internal layers. This tchnique converts a two dimensional matrix into a one dimensional matrix. But then how does matrix multiplication work with flattened matrices? See the MTX java class in the backend model package. The MTX.mult method along with the Matrix class which holds the one dimensional array demonstrates matrix multiplication. The MTX library illustrates how to implement not only matrix multiplication but also all other matrix operations based on a one dimensional model. However, MTX is not intended to be reused; for production neural networks, the ND4J matrix library is recommended.

**Forward Propagation**

During forward propgation, each layer begins with input matrix X. We calculate Z = W \* X + B, and then Y = S(Z). Y becomes the matrix X for the following layer. Each layer may have different weights W, biases B, and activations S.

References:

<http://neuralnetworksanddeeplearning.com/chap1.html>

<https://www.analyticsvidhya.com/blog/2020/02/mathematics-behind-convolutional-neural-network/>

<https://victorzhou.com/blog/intro-to-cnns-part-1/>

<https://medium.com/secure-and-private-ai-math-blogging-competition/cnn-maths-behind-cnn-910eab425b5d>

<https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

<https://medium.com/machine-learning-algorithms-from-scratch/digit-recognition-from-0-9-using-deep-neural-network-from-scratch-8e6bcf1dbd3>

For the convolution layer, The weight matrix is replaced by the filter matrix, and the matrix multiplication W\*X is replaced by matrix convolution. See the reference above on how matrix convolution and max pooling work.

The example in this application follows the propagation sequence:

Input layer -> 1st Convolution layer -> 1st Pool layer -> 2nd Convolution layer -> 2nd Pool layer -> Internal layer -> Output layer

The forward propgation code sequence is:

ConvoNetTrain.fit

fitBatch

trainAllLayers

convoLayer.trainForward

poolLayer.trainForward

MTX.listToSingleCol

internalLayer.trainForward

outputLayer.trainForward

updateEval

outputLayer.setActualY

outputLayer.updateBatchLoss

**Back Propagation**

Training Objective: iterate over multiple inputs, update weight and bias to minimize the loss. We update the weights and biases going in the reverse direction, from output layer backward to input layer. Back propagation requires calculating partial derivatives using the chain rule.

References:

<https://www.mldawn.com/back-propagation-with-cross-entropy-and-softmax/>

<https://www.analyticsvidhya.com/blog/2021/06/how-does-backward-propagation-work-in-neural-networks/>

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

<https://bishwarup307.github.io/deep%20learning/convbackprop/>

Input layer <- 1st Convolution layer <- 1st Pool layer <- 2nd Convolution layer <- 2nd Pool layer <- Internal layer <- Output layer

The back propgation code sequence is:

ConvoNetTrain.backProp

outputLayer.batchLossFn

outputLayer.backProp

internalLayer.backProp

poolLayer.backProp

convoLayer.backProp

Back propagation involves relating the change in weights W and biases B to the change in output value Y. At each iteration either the predicted digit is correct, or it is the wrong digit. The difference between predicted Y and actual Y turns out to be related to the change in the loss function L with respect to the change in Z. Recall that Z = W \* X + B. Therefore, we can relate derivatives of L to derivatives of W and B.

Back propagation begins at the Output layer. In the following description, partial derivatives are represented with "d" in order to allow copying this documentation to/from the java software.

We begin by calculating the matrix partial derivative of loss as follows:

predicted y = softmax(z)

parital loss with respect to z:

Normally, calculating dL/dZ is a two step process:

dL/dZ = (dL/dY)\*(dY/dZ)

However, for the sofmax activation function, the result is simply the difference between predicted and actual y values.

dL/dZ[k] = (predicted y[k]) - (actual y[k])

where k is the class index (0 to 9 for digit classification)

Next find dL/dX for back propagation:

dL/dX = (dL/dZ)\*(dZ/dX)

Note that dL/dX for a layer is set equal to dL/dY on the previous layer. This allows back propagation as follows:

Output layer and Internal layers

dL/dW = change in loss due to change in weight

dL/dB = change in loss due to change in bias

eta = gradient descent rate

Weight correction at each iteration from i to i+1:

W(i + 1) = W(i) - eta\*(dL/dW)

Bias correction at each iteration from i to i+1:

B(i + 1) = B(i) - eta\*(dL/dB)

How to calculate dL/dW?

dL/dW = (dL/dY)\*(dY/dZ)\*(dZ/dW)

recall Z = W\*X + B

recall Y = S(Z) where S(Z) is the activation function

dY/dZ depends on which activation function is applied

dZ/dW = X

How to calculate dL/dB?

dZ/dB = I

dL/dB = (dL/dY)\*(dY/dZ)\*(dZ/dB)

Find dL/dX for back propagation:

dL/dX = (dL/dY)\*(dY/dZ)\*(dZ/dX)

How to back propagate to previous layer?

dL/dY previous layer = dL/dX current layer,

Then apply dL/dW calculation as above to update weight and bias in previous layer.

Then calculate dL/dX in previous layer for back prop input (dL/dY) to next previous layer.

Pool layer

There are no weights in the pool, and no matrix multiplication in back propagation. However, the pool value must be related back to the filter cell where the max value ocurred during forward propagation. See PoolLayer.trainForward, MTX.maxPool, MTX.poolIndex, and PoolLayer.backProp.

Convolution layer

Reference:

https://bishwarup307.github.io/deep%20learning/convbackprop/

The weight is the filter, f, and there is no bias.

Z = X # f, where # represents matrix convolution.

dZ/df = X

dL/df = (dL/dY)\*(dY/dZ) # (dZ/df)

f(i + 1) = f(i) - eta\*(dL/df)

Thus, there are matrix convolutions in place of matrix multiplications. See MTX. convolve for implementation details.

**Conclusion**

Constructing a neural network from scratch is for learning only. This document has described such an application. For Java programmers who have not used JavaFx, the sample code provides examples of a menu bar, tab panel, data entry form, concurrent task, and output charts. The back end code includes network layers, activation functions, a matrix library, and json utilities.