JavaFX Convolutional Network

by

Ronald Cook

www.linkedin.com/in/ronald-cook-programmer

September 2022

JavaFX Convolutional Network

Introduction

The purpose of this project is to learn the implementation details of a convolutional network application programmed in Java, and with a user interface coded in JavaFx. 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 network. If no such library was available, how would you implement a convolutional network from scratch in Java?

This document acts a user guide, and includes a sketchy overview of the network math operations. To actually learn the details, you must walk through some code line by line.

Prerequisites

Math

It is assumed you have already read some tutorials on convolutional networks. Understanding these 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.

References:

Matrix algebra:

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/

Matrix calculus:

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

Multiple Linear Regression:

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

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

Linear Regression – Gradient Descent:

https://towardsdatascience.com/linear-regression-simplified-ordinary-least-square-vs-gradient-descent-48145de2cf76

https://www.analyticsvidhya.com/blog/2021/04/gradient-descent-in-linear-regression/

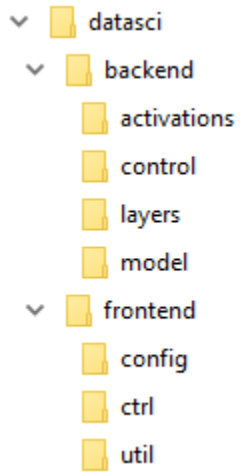
Convolution Network – Gradient Descent:

https://builtin.com/data-science/gradient-descent

How is multiple linear regression (MLR) related to convolutional networks? Review the references above, and note the matrix solution to find the MLR model parameters. The network model is also based on a matrix formulation. The objective is the same - to find the set of parameters that produces the best output prediction. Unfortunately, in the convolutional network, there may be hundreds of thousands of parameters, which usually makes the MLR model computationally intractable. Instead, the gradient descent approach is used in the convolutional network to gradually update the parameters until a solution is found.

Application Code

The application code was divided into front end and back end sections, as shown in the file directory structure.



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 19 JDK may be downloaded from:

https://jdk.java.net/19/ -> JDK 19 download

The JavaFx 19 SDK may be downloaded from:

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

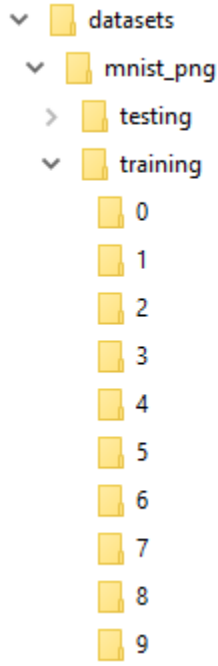
19 SDK download

The IntelliJ IDE was updated with Java JDK 19 and library JavaFx SDK 19 to build the application. The windows batch file, FxConvoNet.bat, may be used to run the application.

Dataset

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.



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 training 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 randomly shuffled before the network training or testing task begins.

Note: even though this application was based on processing handwritten digits from the MNIST dataset, other image files could be loaded into the same folder structure. The folder names are merely labels for the images, but the images do not need to be digits.

Project Overview

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.

References:

Hyperparameters:

https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a

Front end

The front end consists of several JavaFx screens. Why do we need a front end user interface? We don't. Most tutorials show hard-coded networks. With that approach, to see what happens if you change a hyperparameter, the code itself must be updated. The advantage of a JavaFx user interface is allowing the user to change hyperparameters in a form, without changing the code. Then the form data can be passed directly to the back end, or can be saved to a file for future runs. Another advantage of JavaFx is being able to create charts of network performance.

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

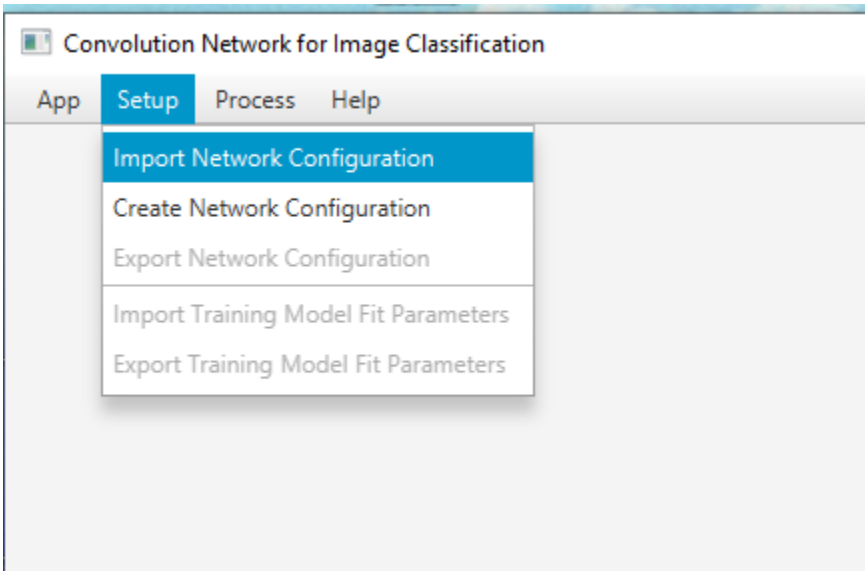
initHelpMenu

stage.setScene

stage.show

Setup menu

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



The Setup menu -> Import Network Configuration opens the file dialog to read in the network configuration file, which is in json format. Two sample files are included with this application in the config subdirectory. 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

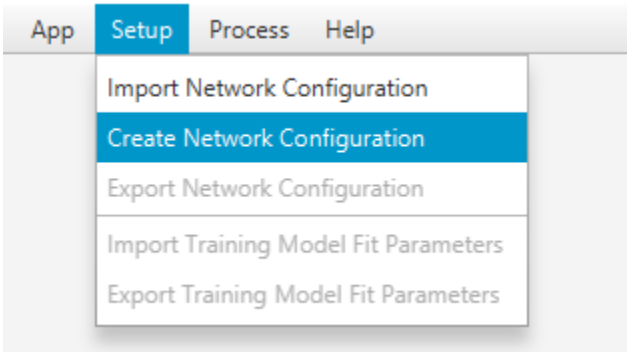
createInputConfig

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.



The code sequence outline is:

AppMenu.initSettingsMenu

configView.createAllLayers

initNetConfigPanel

createGeneralConfig

createInputConfig

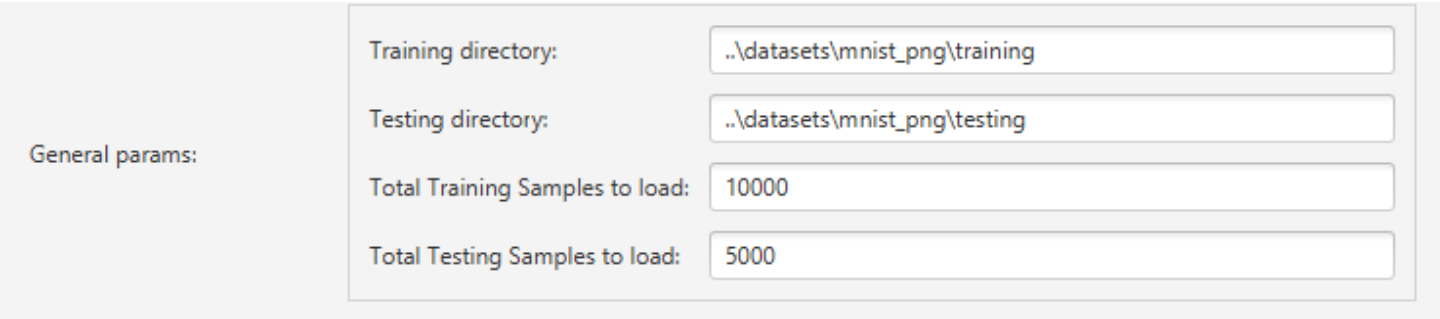
createConvoPoolConfig

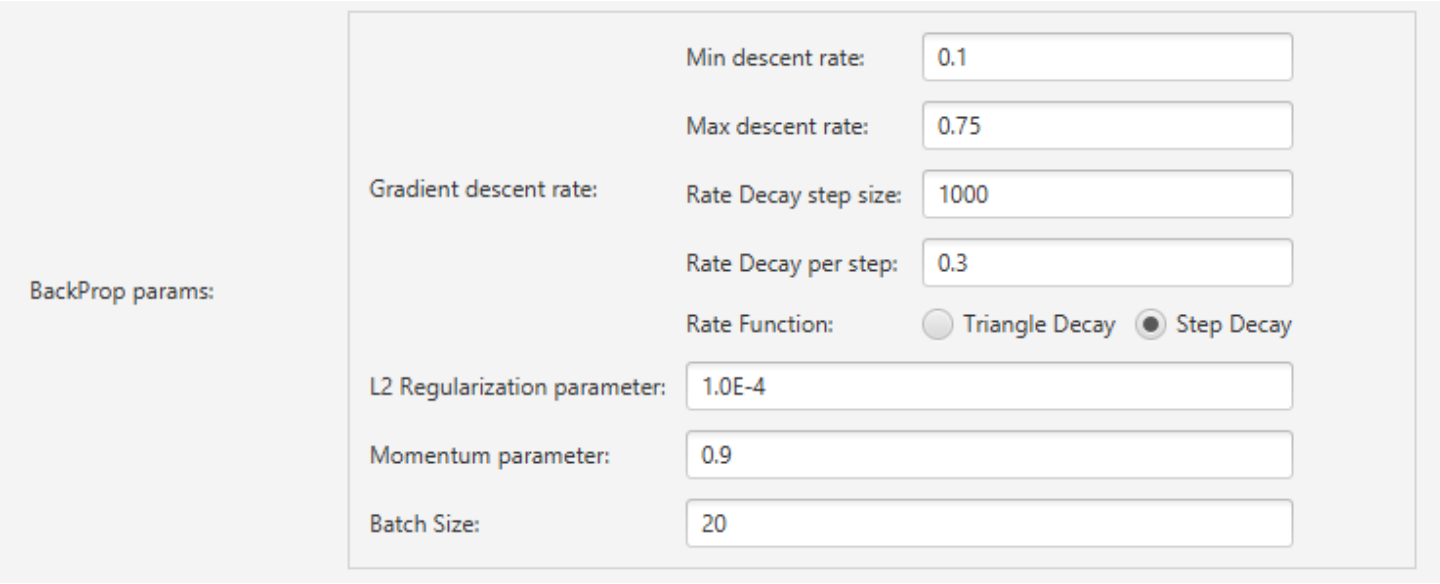
createInternalConfig

createOutputConfig

Import config panel

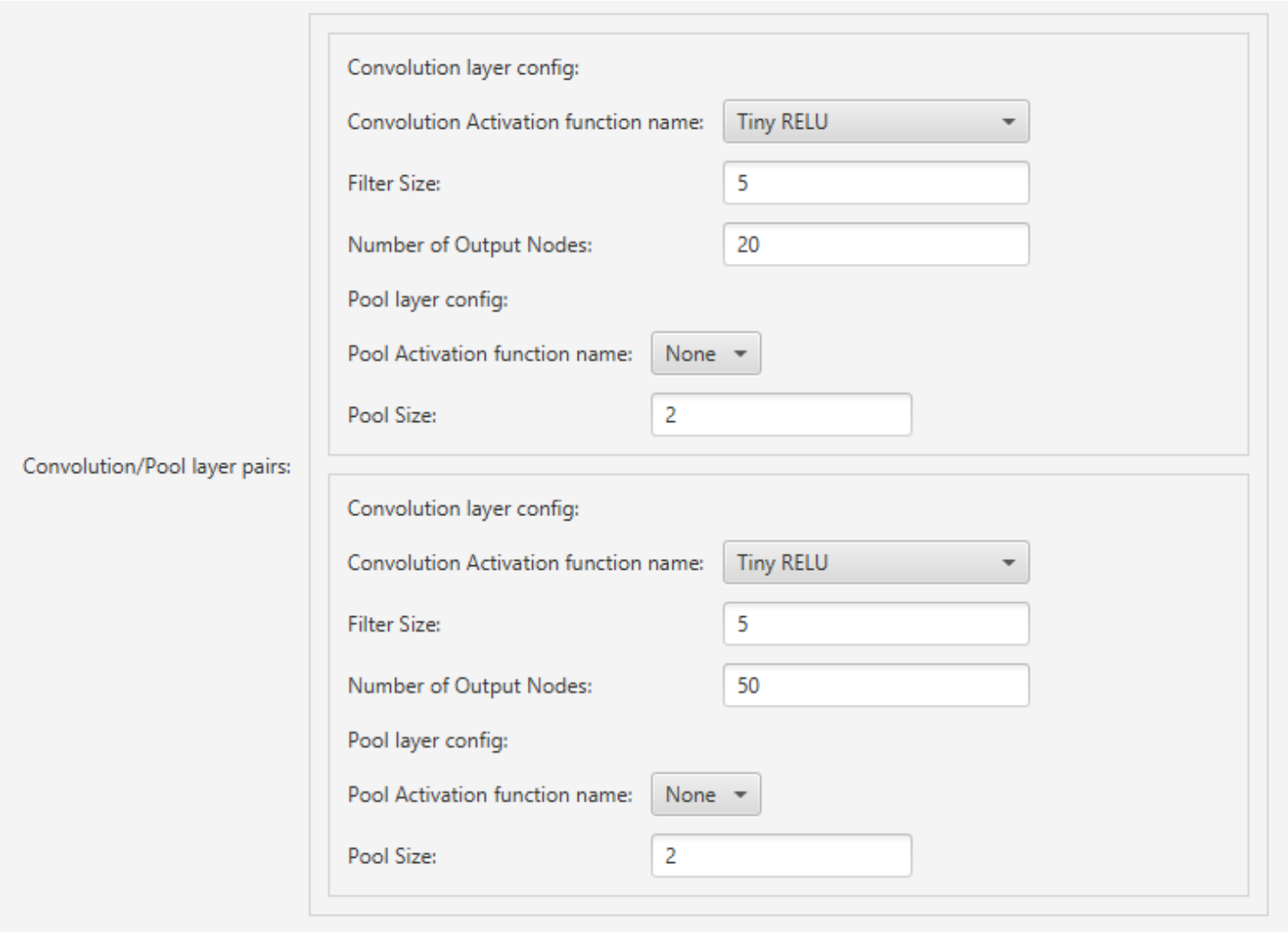
The Import config panel contains all the basic information to perform a network training run. The General parameters includes the directories where the image data is located, number of samples to load. The back propagation parameters panel contains properties used in updating the weights and biases. The gradient descent sub-panel includes minimum rate, maximum rate, rate decay parameters, and type of rate decay function: triangle or step. The L2 regularization factor is a small number which reduces the weight matrix on each update. The Momentum parameter adds a fraction of the previous gradient update to the current update. The batch size is the number of samples to process before applying the back propagation updates.



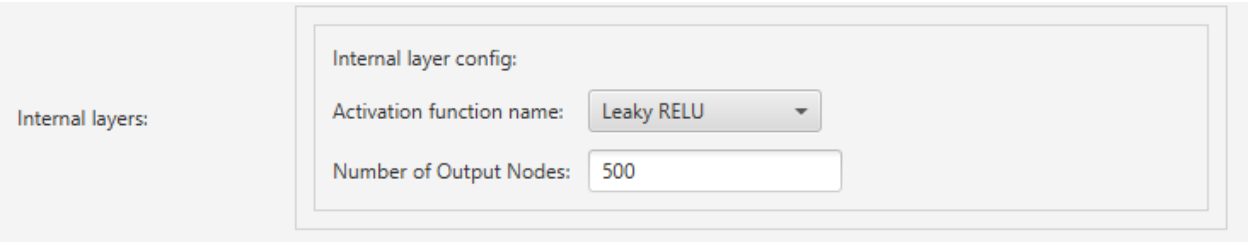


The Input layer panel has input fields for image pixel height and width which is necessary for matrix sizing, described in the back end section.

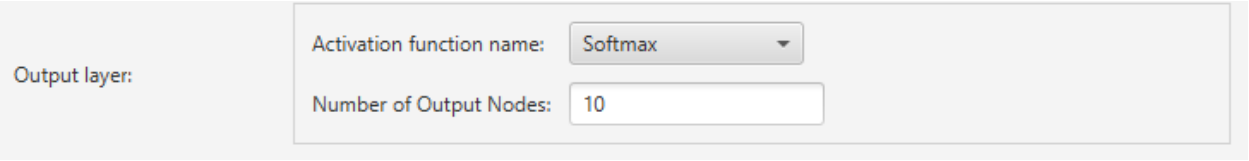
The Convolution/Pool Layer panel contains sizing information, and options to choose the action function. The Pool activation function is set to None and cannot be changed.



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.



The Output Layer panel has fields for activation function and output node size, which equals 10 for the digit images. If the size is less than 10, only those folders will be read. For example, if the size is 6, only image folders 0 through 5 will be read. This feature allows processing fewer than 10 digits, and also allows analysis of other types of image files that require less than 10 folders.



How do we determine what options to enter to produce the best results? We must resort to trial and error. We can study successful network configurations by other data scientists. The example shown here is basically the same as an example from Deeplearning4j, with two convolution/pool layers, one internal layer, and the output layer. The hyperparameters and activation functions are different. Multiple scenarios should be run to compare performance.

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

AppMenu.initSettingsMenu

configView.exportConfig

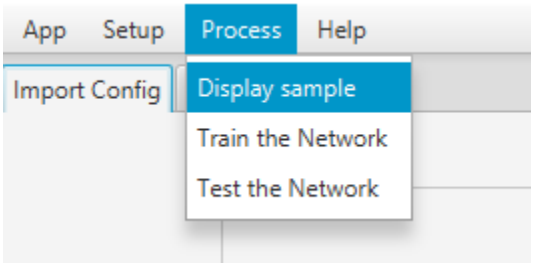
ViewUtil.openCreateFileDialog

FileUtil.getOutputStream

JsonUtil.configToJson

Display sample

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



The code sequence outline is:

AppMenu.initOptionMenu

pixelView.displayImage

imageFileDialog

initData

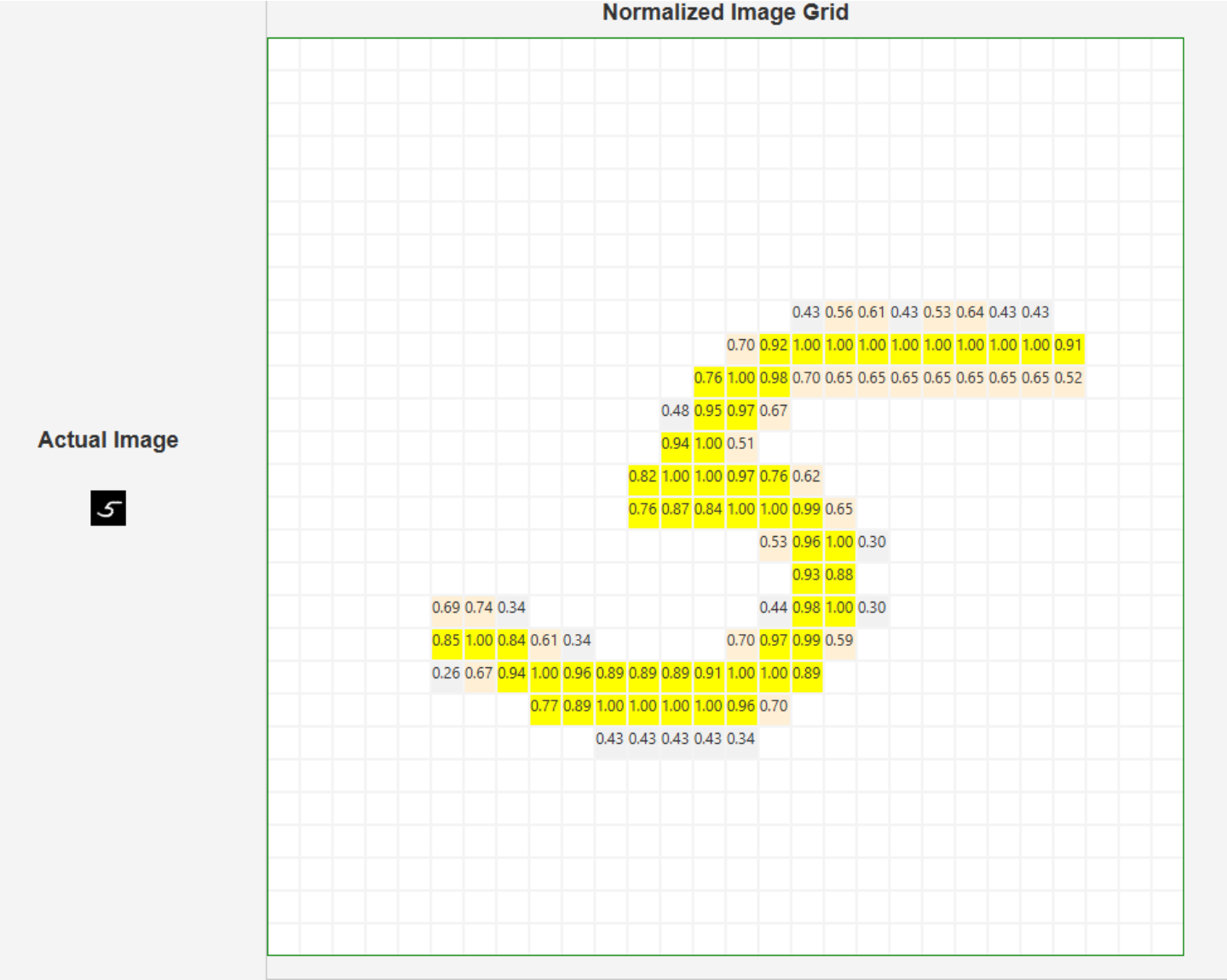
ImageDataUtil.loadImageData

viewData

imageView.setImage

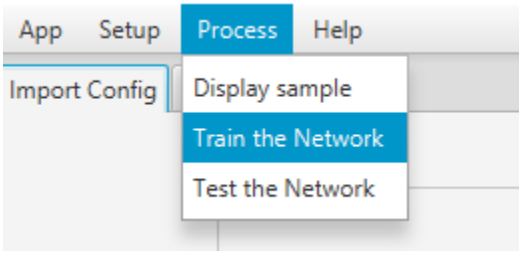
ImagePane(imageMatrix)

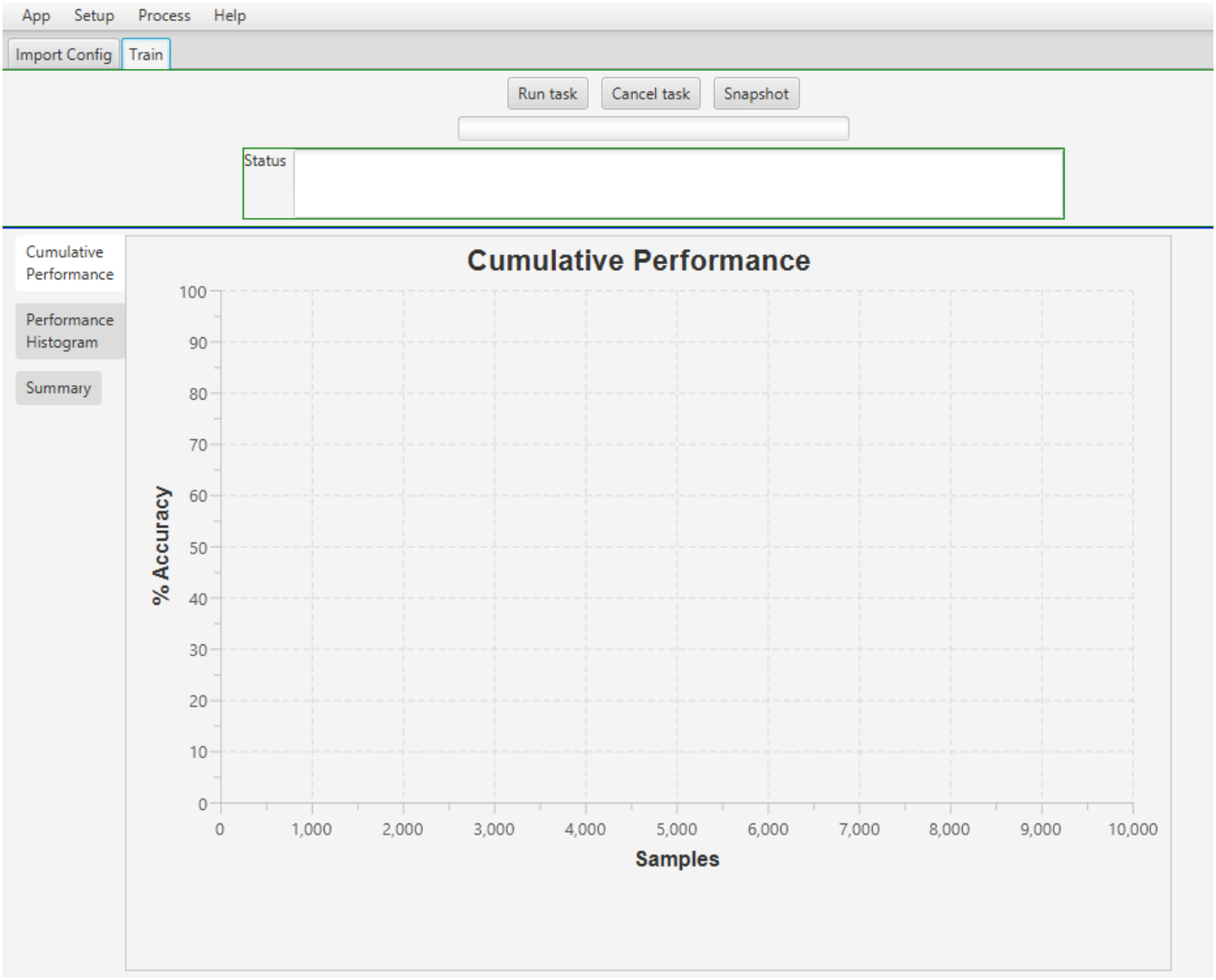
The actual image is shown on the left of the screen. On the Normalized Image Grid, the pixel values are displayed after being normalized 0.0 to 1.0, where 0 is black and 1 is white. Values less than 0.25 are not displayed.



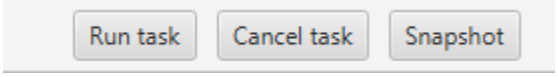
Train the Network

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

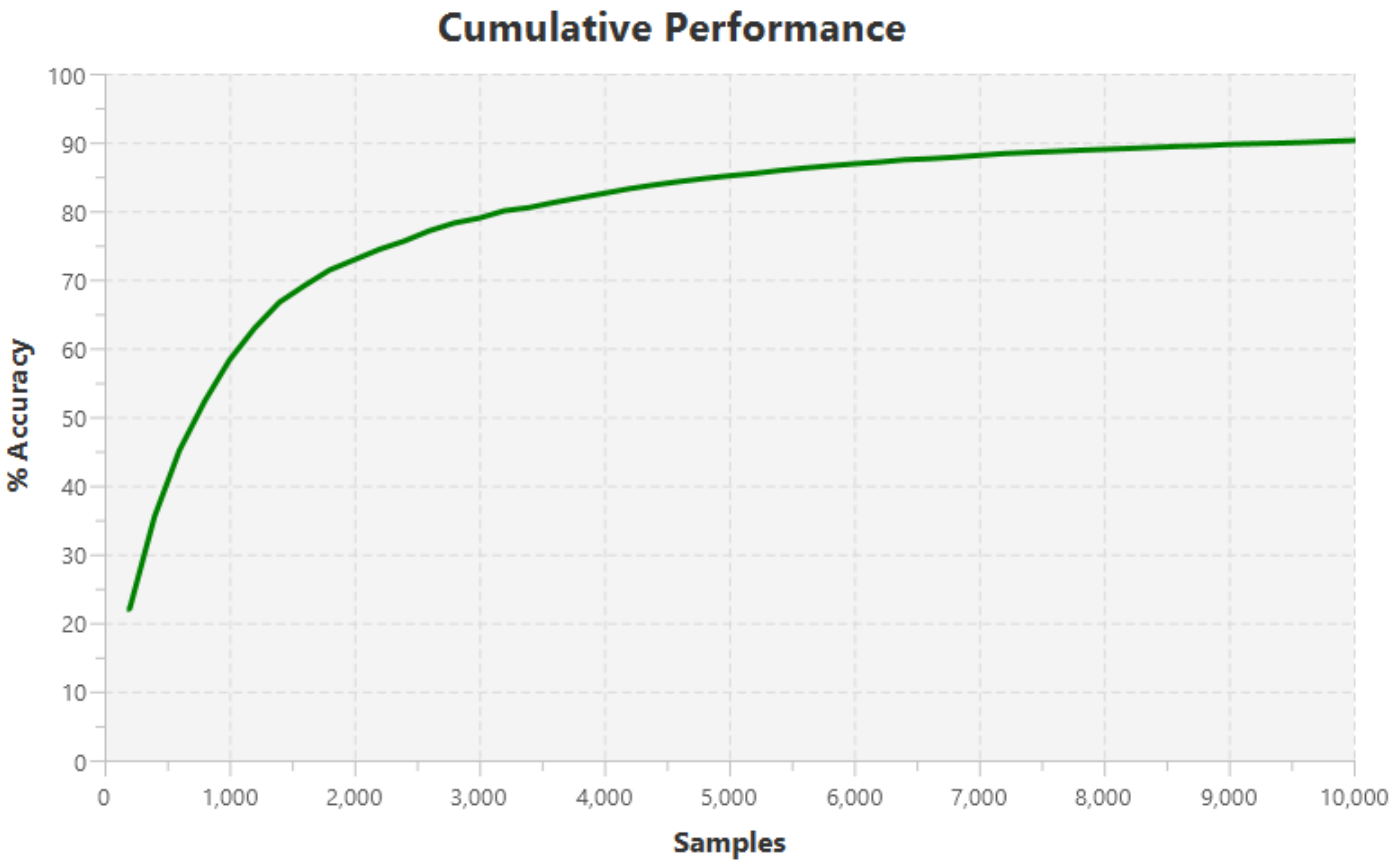


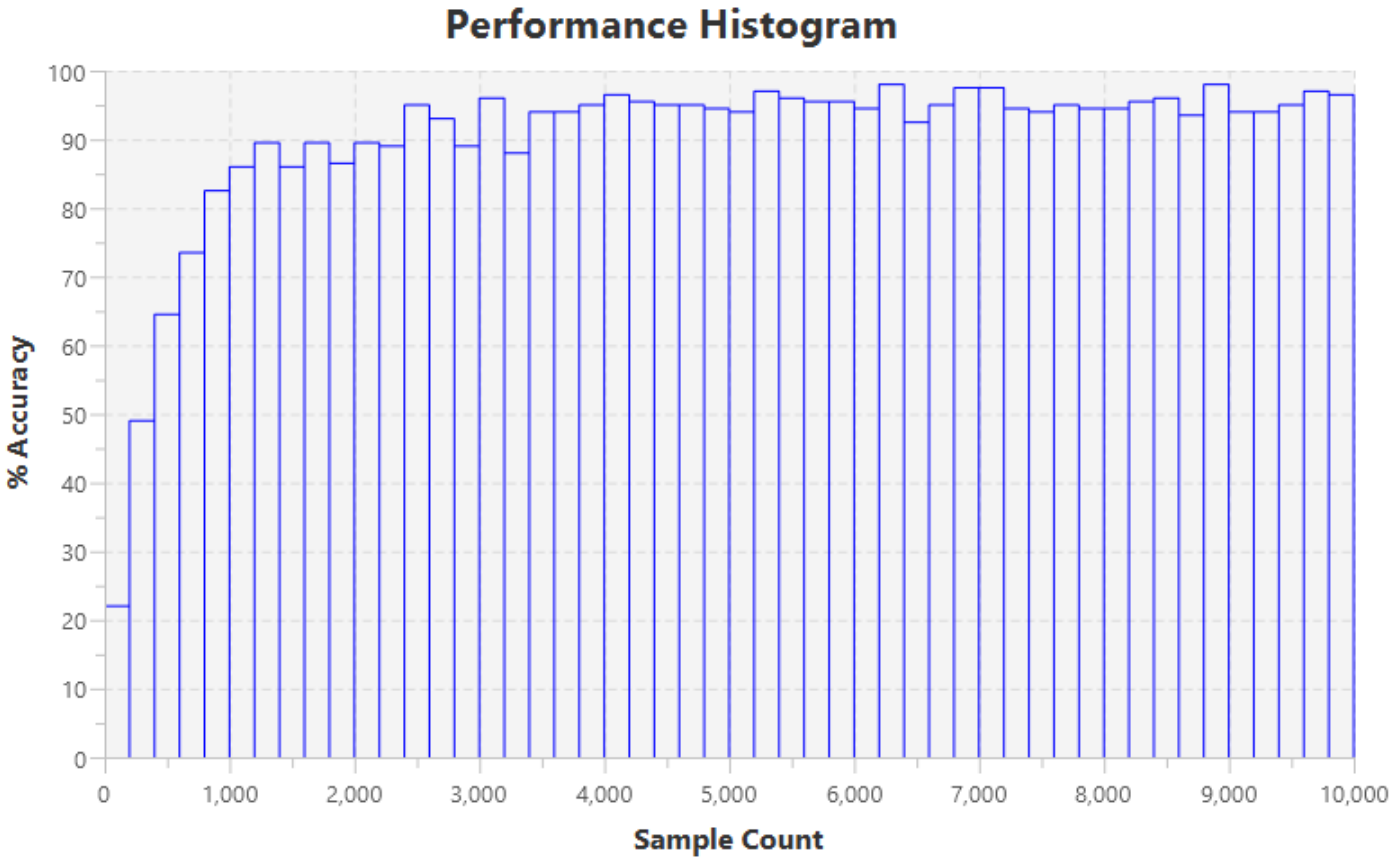


The buttons at the top of the network task screen allow the user to run the network task, cancel a running task, and to take a snapshot of the performance charts.



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

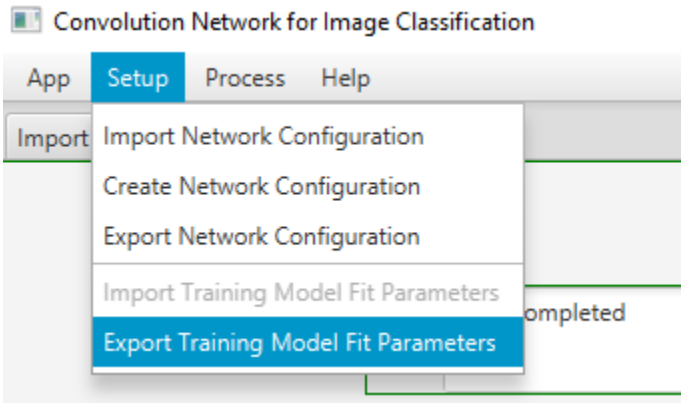



Performance Histogram
% Accuracy
100
90
80
70
60
50
40
30
20
10
0
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
9,000
10,000
Sample Count

Confusion Matrix

Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	951	0	5	2	4	7	11	2	10	8
1	5	968	7	1	3	1	3	2	7	3
2	15	3	895	20	10	0	8	28	15	6
3	8	5	30	879	2	18	5	18	22	13
4	4	2	6	0	912	0	9	10	8	49
5	15	3	5	24	4	890	12	8	24	15
6	27	4	7	2	7	10	924	1	13	5
7	11	0	21	8	10	1	3	900	5	41
8	10	12	19	19	9	21	12	11	861	26
9	14	4	3	15	49	11	5	33	20	846

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.



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.

Test the Network

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

The test run rquires two data structures: Network Configuration (NetConfig) and Model Fit Parameters (FitParams).

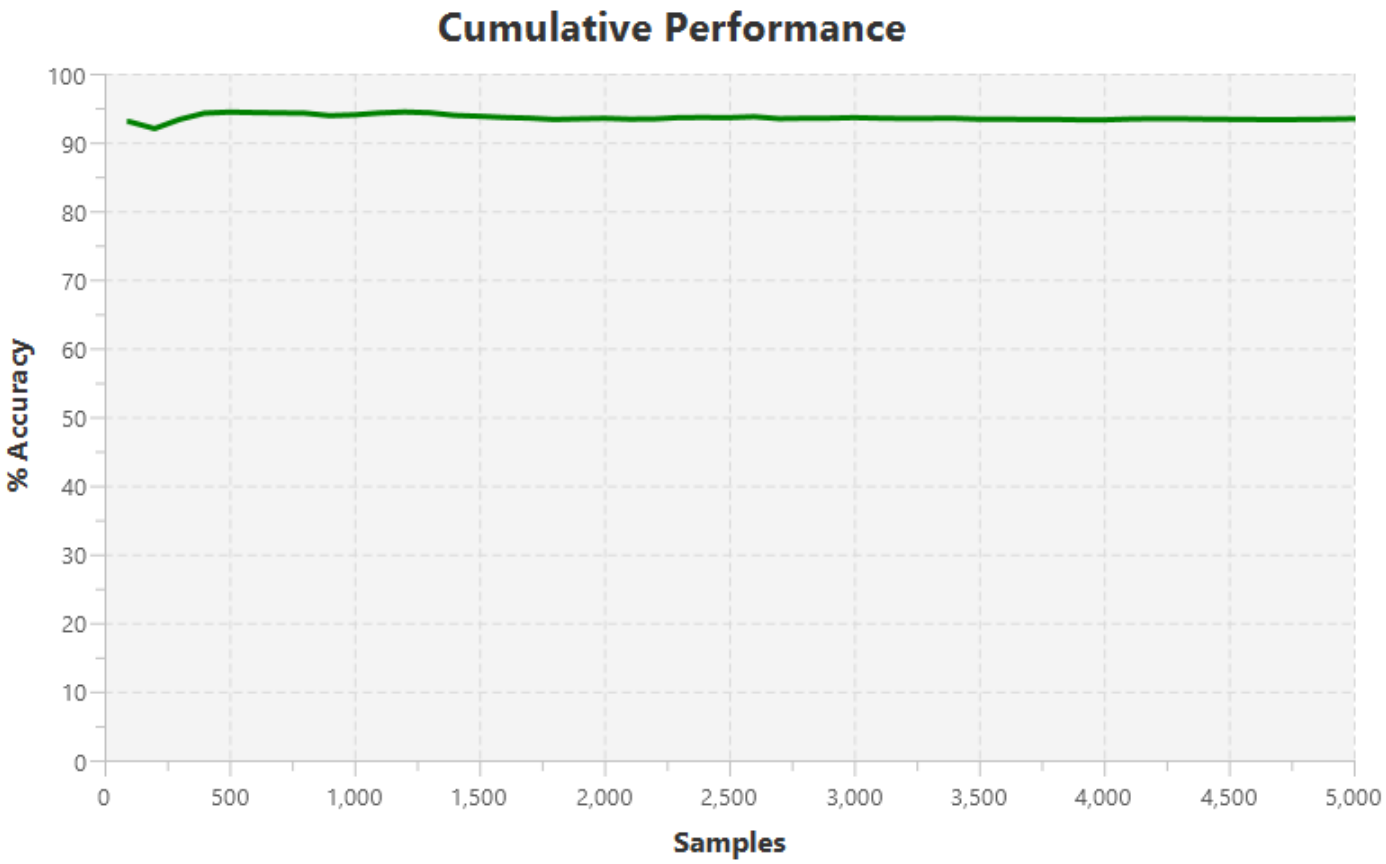
Therefore, there are two scenarios to consider:

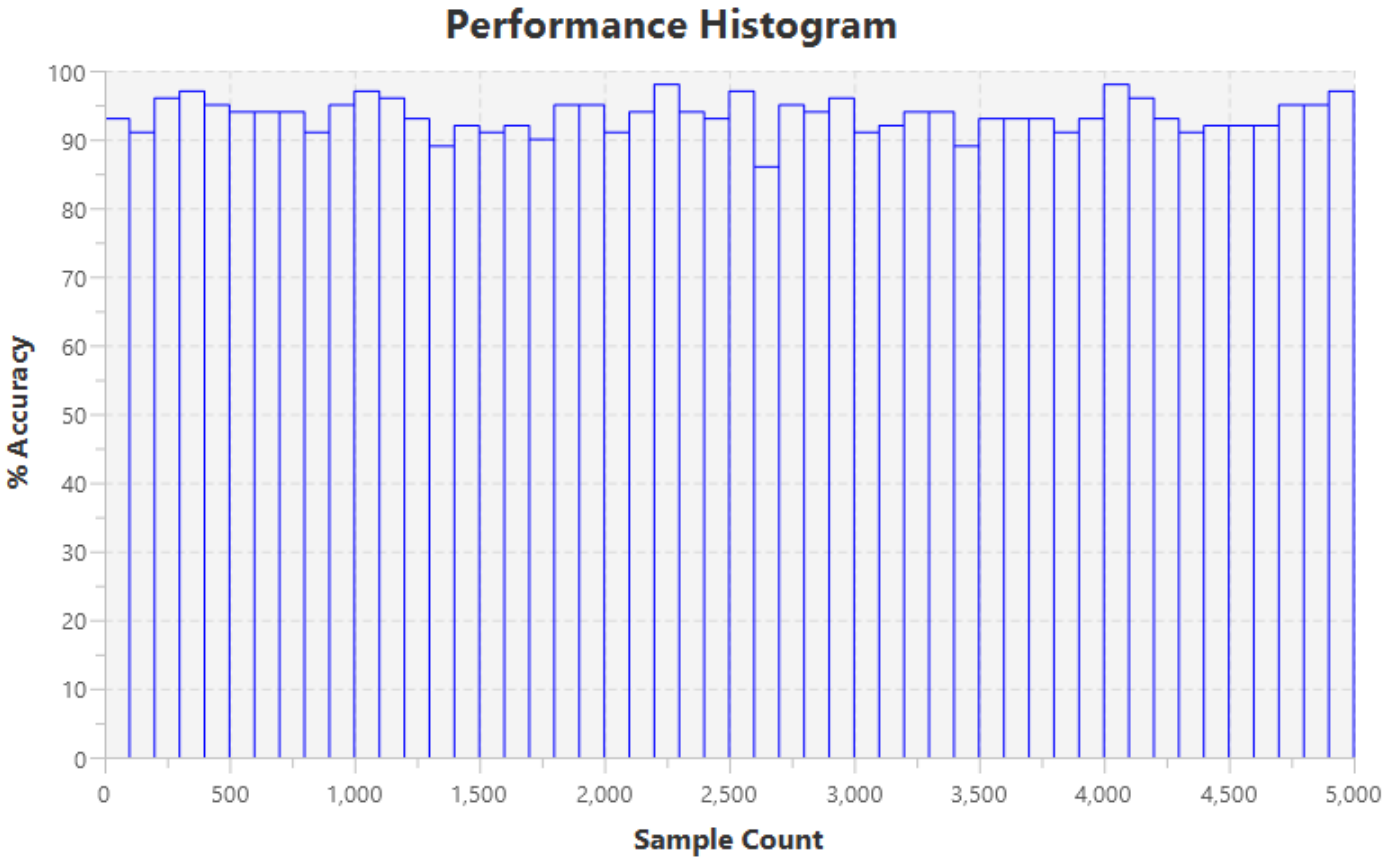
Scenario 1. After a successful training run, the two data structures are in memory. And the test may be run immediately from the Test tab, by selecting the Run task button.

Scenario 2. After a training run, export the Network Configuration and the Model Fit Parameters. Then the app may be closed for running a test later. When ready, launch the application again. Under Setup, first Import Network Configuration and then Import Training Model Fit Parameters. Under Process, select Test the Network. Then on the Test tab, select the Run task button.

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 Histogram are updated periodically as the task is running. The Summary Confusion Matrix is available after the run has completed.




Performance Histogram
% Accuracy
100
90
80
70
60
50
40
30
20
10
0
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
Sample Count

Confusion Matrix

Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	490	0	1	0	0	0	9	0	0	0
1	0	471	10	0	2	1	12	1	3	0
2	2	0	483	6	0	0	2	7	0	0
3	1	0	17	450	0	18	1	12	1	0
4	0	0	4	0	477	0	7	2	1	9
5	2	0	2	4	0	478	8	4	2	0
6	3	0	2	0	1	4	487	0	2	1
7	0	1	34	2	0	0	0	460	1	2
8	15	0	12	3	3	9	22	7	422	7
9	8	3	2	3	13	6	1	10	0	454

Back end

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

References:

Digit classification:

http://neuralnetworksanddeeplearning.com/chap1.html

Weight initialization:

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

Triangle function:

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)

nIn = number of input nodes into a layer

nOut = number of output nodes from a layer

The weights and biases must be initialized before running the network. The standard approach is to use a random gaussian distribution with mean zero and standard deviation related to the layer input size. See the

initWeight method in each layer for implementation details.

Note: for a convolutional layer, the weight matrix W is replaced with the filter matrix, 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 = size of feature map

convolution output matrix = (nf , nf)

nOut = number of output feature maps

Pool Layer:

pool matrix = (p,p)

p = pool size

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

pool output matrix = (nfp , nfp)

number of pool output nodes:

pOut = nfp * nfp * nOut

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:

input: one image matrix(28,28), nIn = 1

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

nf = 28 - 5 + 1 = 24, size of feature map

convolution output matrix = (24 , 24)

nOut = 20, number of output feature maps

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:

input: 20 pool matrices(12,12), nIn = 20

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

nf = 12 - 5 + 1 = 8, size of feature map

convolution output matrix = (8 , 8)

nOut = 50, number of output feature maps

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 convolutional network makes extensive use of matrix algebra, a matrix library is an essential tool for the application. Since 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 this application, a matrix class, MTX, has been created to perform all network matrix operations without relying on an external library. Why reinvent the wheel? There are at least three reasons: to comprehend matrix flattening, matrix convolution, and max pooling.

Matrix flattening converts a two dimensional matrix into a one dimensional matrix. But then how does matrix multiplication work with flattened matrices? You may have seen pictures of matrix convolution, but how is it implemented? How is max pooling accomplished? Here, you must decide whether or not to take a detour. To focus just on the network implementation, you could skip over the matrix MTX class.

But if you really want to learn about matrix flattening, matrix convolution, and max pooling, 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 class illustrates how to implement not only matrix multiplication but also all other matrix operations based on a one dimensional model. For concrete examples, see the JUnit tests for MTX in MatrixTests.java. However, keep in mind, MTX is for learning only, and is not intended to be reused. For production networks, a matrix library such as ND4J is recommended.

Forward Propagation

During forward propagation, 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. In general, matrix Z is a one column matrix with the number of rows equal to the number of output nodes, nOut. Matrix X has one column and number of rows equal to the number of input nodes, nIn. Therefore, the matrix W contains nOut rows and nIn columns. Matrix B and Y have one column and the same number of rows as Z.

References:

Convolution Network:

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 convolutional layer, the weight matrix is replaced by the filter matrix, and the matrix multiplication W*X is replaced by matrix convolution.

See the references above on how matrix convolution and max pooling work.

The general outline above shows the calculation steps for a single weight matrix. However, the convolutional layer usually contains many filter matrices. How does the calculation change for more than one filter matrix?

Let's examine the Z function closely.

Z = W * X + B

The input matrix is the output Y from the previous layer. Consider the second convolutional layer which has for example 20 input feature maps coming from the first layer. In this case X means a column 'list' of 20 matrices. What is W? Matrix W contains nOut rows, and each row has nIn columns, but each cell is a filter matrix!

For example, if the second convolutional layer has 50 output nodes, W has 50 rows, and 20 columns. Note well: each cell of W is a filter matrix (e.g. 5x5), while each cell of X is a feature map matrix (e.g. 12x12). In this example, there would be 20*50 = 1000 matrix convolutions to produce Z. Therefore, Z is one column of 50 output feature map matrices (e.g. 8x8).

As an example, how is the 3rd row of Z computed? It is the product of the 3rd row of W with the the column X, plus a bias term. Each row in W has 20 columns and each cell (filter matrix) is convolved with each cell (feature map matrix) in X, and summed over the 20 products to produce the 3rd row of Z as one output feature map matrix.

As you can see in convoLayer.trainForward, the argument is List<Matrix>. Compare with

internalLayer.trainForward, which takes a single Matrix argument for X. Also, W, a matrix of matrices, is implemented as a list of list of filter matrices in filterList in ConvoLayer.

The example in this application implements the forward propagation sequence:

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

The forward propagation 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:

Back propagation calculus:

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

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

Back propagation – loss function:

https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

Back propagation – softmax:

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

Momentum:

https://d2l.ai/chapter_optimization/momentum.html

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

https://dominikschmidt.xyz/nesterov-momentum/

L2 Regularization:

https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

The example in this application implements the following back propagation sequence:

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

The back propagation 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. Recall that Z = W * X + B, and Y = S(Z). At each sample either the predicted image is correct, or it is wrong. 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. Therefore, we can relate derivatives of L to derivatives of W and B. The math presented here is a little sketchy, and the references

provide better derivations. However, there doesn't seem to be a common notation, which makes learning more difficult.

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 the java software.

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

    predicted y = softmax(z)

    partial loss with respect to z: dL/dZ

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

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

However, for the softmax activation function, the result is simply the difference between predicted and actual y values. See reference above labeled "Back propagation – softmax".

    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 described below.

Output layer and Internal layers

Batch: a small number of samples processed through the forward propagation and back propagation before performing updates on the weights and biases. The purpose of the batch is to smooth out the updates. If the batch size is 20 samples, and if there are 20,000 samples, the number of batches would be 1000. A batch size of 1 means updates are performed on every sample.

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

batch ave[dL/dW] = average of dL/dW over batch size

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

batch ave[dL/dB] = average of dL/dB over batch size

Eta: gradient descent rate, is defined in a function to vary by sample count, and is a multiplier factor for dL/dW and dL/dB.

Momentum v: exponential average of the gradient steps

mu: momentum factor, close to 1, usually about 0.9

Momentum update after each batch from i to i+1:

v(i + 1) = mu * v(i) - eta * batch ave[dL/dW]

L2 regularization factor, lambda: small number which reduces the weight matrix on each update.

Weight correction after each batch from i to i+1:

W(i + 1) = (1 - lambda)*W(i) + v(i + 1)

See MathUtil.updateWeightMatrix for implementation details.

Bias correction after each batch from i to i+1:

B(i + 1) = B(i) - eta * batch ave[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

dL/dY is known from previous layer

dY/dZ depends on which activation function is applied

dZ/dW = X

How to calculate dL/dB?

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

dL/dY is known from previous layer

dY/dZ depends on which activation function is applied

dZ/dB = I

Find dL/dX for back propagation:

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

dL/dY is known from previous layer

dY/dZ depends on which activation function is applied

dZ/dX = W

How to back propagate to previous layer?

dL/dY previous layer = dL/dX current layer

Apply dL/dW and dL/dB calculations as above to update weight and bias in previous layer.

Proceed to 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 occurred during forward propagation. See PoolLayer.trainForward, MTX.maxPool, MTX.poolIndex, PoolLayer.backProp, and corresponding Junit tests in MatrixTests.

Convolutional layer

References:

Back propagation – convolution:

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

Back prop – convolution padding

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

Back prop - convolution multi-node input:

https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509?gi=9e0f2eebeef5

In the convolutional layer, the weight W is the filter, and there is a bias B.

Z = W * X + B, where W * X is matrix convolution

We can use 'unfolded' matrix preparation to turn convolution into matrix multiplication. See MTX.

Convolve, MTX.unfold, and MatrixTests.unfold for implementation details.

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

dL/dY : back prop input from previous layer

dY/dZ = derivative of activation function

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

dZ/dX = W

dL/dX = unfolded{(dL/dZ)} * rotated(W)

dZ/dW = X

dL/dW = unfolded{(dL/dZ)} * (dZ/dW)

v(i + 1) = mu * v(i) - eta * batch ave[dL/dW]

W(i + 1) = (1 - lambda)*W(i) + v(i + 1)

For bias:

dZ/dB = I

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

B(i + 1) = B(i) + batch ave[dL/dB]

Conclusion

Constructing a convolution 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 front end interface allows the user to create various network scenarios without modifying the code. The back end code includes network layers, activation functions, a matrix library, and json

utilities. The back propagation math has been outlined without derivations. If you examine some of the code, as well as the JUnit tests, it should help you to understand the implementation of convolutional networks.