Neural Networks

Joe Henthorn
University of Washington

Winter 2020

Abstract: The goal of this paper is to expond on the history and foundations of neural networks and to demonstrate how to use the Neural Network functions in MatLab to solve a classification problem.

1 Introduction

Neural networks are a subclass of machine learning that uses networks of artificial neurons to solve a system of equations such that the solution leads to an activation event or a null event. In this case, an activation event resembles a biological neuron in that one logical node can act like a switch to initiate a forward transmission and propagation of information to another downstream node. This document is a guide for understanding Neural network history, architecture, and the applications of this algorithm. The Idea of an artificial neuron comes from the work of McCullock and Pits (1943) who developed simple AND and OR computational nodes. This idea was extended in by Rosenblatt (1957) by the development of the perceptron that used weights to map an input to a linear threshold unit node where it would then make a binary classification. The stacking of these perceptrons became the first neural network (NN) architectures, and since has expanded to include many more activation functions and convolutional filter layers. Here we will discuss how to build fully connected and convolutional neural networks in MatLab.
For part one of this example, we will test a fully connected network and look at its accuracy, loss function, and the confusion matrix. Part two of this example will build and test a convolutional NN, and compare its accuracy, loss function, and the confusion matrix.

This is a terminology heavy paper and a list of terms will greatly aid the reader.

Layer: Refers to the number of layers in a network.
Width: Refers to the number of nodes per layer.
Fully connected: Feed forward layers. One layer feeds into the next.
Loss function: Refers to the objective function being minimized. Also known as the cost function. In this example we will use the cross-entropy loss function.
Over fitting: When there are two many parameters to define a solution set. In this case, over fitting occurs when the network is two large.
Confusion matrix: A matrix depicting the results of classification, where correct guesses are on the diagonal and errors are on the off diagonal.
Convolution: The convolution function is essentially a blending function that takes two separate functions or values and "blends" them together to form a single composite output.
Padding: Refers to adding zeros around a sub-set of data in a convolutional layer to make it the same size as the original input.
Stride: Refers to the step size for the field of perception in a convolutional layer.

2 Theoretical Background

The core of this method comes from the simplest model for mapping inputs to outputs. Linear regression, and the root,mean,squared error function form the foundation for the network computation.

$$\hat{y}_j = mx_j + b \tag{EQ:1}$$

$$M_{S_E} = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(y_i - \hat{y_j})^2} \tag{EQ:2}$$

For this general case we further extend our model to multiple linear regression with multiple in independent variables $(x_1, x_2, ..., x_n)$. With order pairs of $(\vec{x_1}, y_1), (\vec{x_2}, y_2), ...(\vec{x_n}, y_n)$. This step will later form the layer input our our neural networks.

$$\hat{y} = \alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_n x_n \tag{EQ:3}$$

To predict multiple outputs we can extend our model to multivariate linear regression.

$$\begin{aligned} y_1 &= \alpha_{1,1}x_1 + \alpha_{1,2}x_2 + ... + \alpha_{1,n}x_n + b_1 \\ y_2 &= \alpha_{2,1}x_1 + \alpha_{2,2}x_2 + ... + \alpha_{2,n}x_n + b_2 \\ &\vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ y_n &= \alpha_{m,1}x_1 + \alpha_{m,2}x_2 + ... + \alpha_{m,n}x_n + b_n \end{aligned} \tag{EQ:4}$$

Here we can employ linear algebra to redefine terms and simplify things.

$$\text{"}Inputs\text{"} \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad , \quad \text{"}Outputs\text{"} \quad \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{EQ:5-8}$$

$$\text{"}Weights\text{"} \quad A = \begin{pmatrix} \alpha_{1,1} & \dots & \alpha_{n,1} \\ \alpha_{2,1} & \alpha_{2,2} & \dots \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \dots & \alpha_{m,n} \end{pmatrix} \quad , \quad \text{"}Bias\text{"} \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

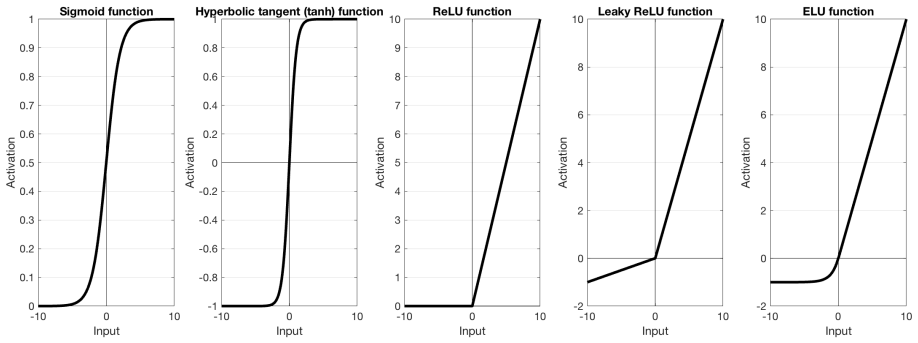
More succinctly we can write our multivariate equation for linear regression as:

$$\vec{y} = A\vec{x} + \vec{b} \tag{EQ:9}$$

To build our neural network we need to feed our output vector into a activation. Classically, the sigmoid σ(x) curve function used with logistic regression has been used as a good activation function.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{EQ:10}$$

In current practice, there are many activation functions that will be listed here but not investigated in depth. The most popular include the Rectified Linear Unit (ReLU), hyperbolic tangent (tanh(x)), Exponential Linear Unit (ELU), and the max output functions.



.

Figure 1: Plot of activation functions for sigmoid, hyperbolic tangent, rectified linear unit, and exponential linear unit.

Here we can use this logistic function to interpolate our outputs between 0 and 1 like a probability (P).

$$P_y = \frac{1}{1 + e^{-(mx+b)}} \tag{EQ:11}$$

Here we can use this logistic function to interpolate our outputs between 0 and 1 like a probability (P), where we want to maximize:

$$\begin{cases} \ln(1-P) & \text{if } y_j = 0 \\ \ln(P) & \text{if } y_j = 1 \end{cases}$$

$$P_y = \frac{1}{1 + e^{-(mx+b)}} \tag{EQ:12}$$

The error/objective function that we want to minimize is the log loss or cross entropy loss equation:

$$Cross \quad entropy \quad loss = \frac{1}{N} \sum_{j=1}^{N} \quad [y_j \ln(P_j) + (l - y_j) \ln(1 - P_j)] \tag{EQ:13}$$

For multinomial logistic regression we can restate our probability function as:

$$\vec{P} = \frac{1}{1 + e^{-\vec{y}}} \tag{EQ:14}$$

This leads to another activation function that we will use for classification. The softmax function will typically be the last activation function we use in a neural net on our output layer.

$$\vec{P} = \frac{1}{\sum_{j=1}^{M} e^{-y_j}} \tag{EQ:15}$$

Each of the previous functions represented a single layer of a neural network. We can extrapolate this for many layers in a neural network such that we compose a composite function that includes a series of inputs, weights, and activation functions.

$$\vec{Y} = \sigma(A_m \vec{x_n} \ldots \sigma(A_2 \sigma(A_1 \vec{x_1} + \vec{b_1}) + \vec{b_2}) \ldots + \vec{b_n}) \tag{EQ:16}$$

3 Methods & Algorithm Implementation

First we train a basic feed forward network. In this example we use the Mnist fashion data set that is publicly available (6000 samples). First the data needs to be converted to double precision numbers and reshaped to feed into the MatLab neural network functions. Next, we then take a small subset (5000 samples) of our training data to use as validation data. We also do the same for the training labels. Then we use the categorical command for all label data.
We are now ready to build our feedforward network. We define the number of epochs and the size of each layer. Then we define the layer structure size with **imageInputLayer()**. We use the **fullyConnected-Layer()** command to build the layer, and we use the **reluLayer** command as our activation function. The last layer of the network should always be the size of the classification system we use. In this case the last layer is size 10, and we use the **softmaxLayer** activation function for the classification activation function.

Prior to training the network, we adjust the options of the network to tailor it for our needs. These settings are called hyper-parameters and they include the the optimizing algorithm, learning rate, max epochs, regularization, plotting, and defining our validation data. We use the function **trainingOptions()** to set the options and we use the **'adam'** optimizer for this example. To train our network we use the **trainNetwork()** command. Once trained we can validate the training data by using the classify(network,validation data) command. We then plot the confusion matrix with **plotconfusion(y test,y predict)** command and check the results. If the system is overfit, we need to reduce the network size, and if the network is not overfit, we can expand the network size. For testing new data, we use the same method as the validation data, and use the **classify(network,test data)** command. Well done we have done it.

Because performance on the fully connected network scheme is limited, we might try to make a convolutional neural network. For this example we recreate a famous convolutional neural network called the LeNet 5. The process is the same for the fully connected network, but there are a few more options like padding and stride. For this example we use the **'same'** option for padding, which adds zeros to a border of data such that the convolutional layer subdivided data can be made the same size as the original input data size. We can further control the stride of our convolutional function to be large or small. Small stride is analogous to finer resolution. We also add a average pooling layer for each convolutional layer. There are many more options in the neural network design application in MatLab.

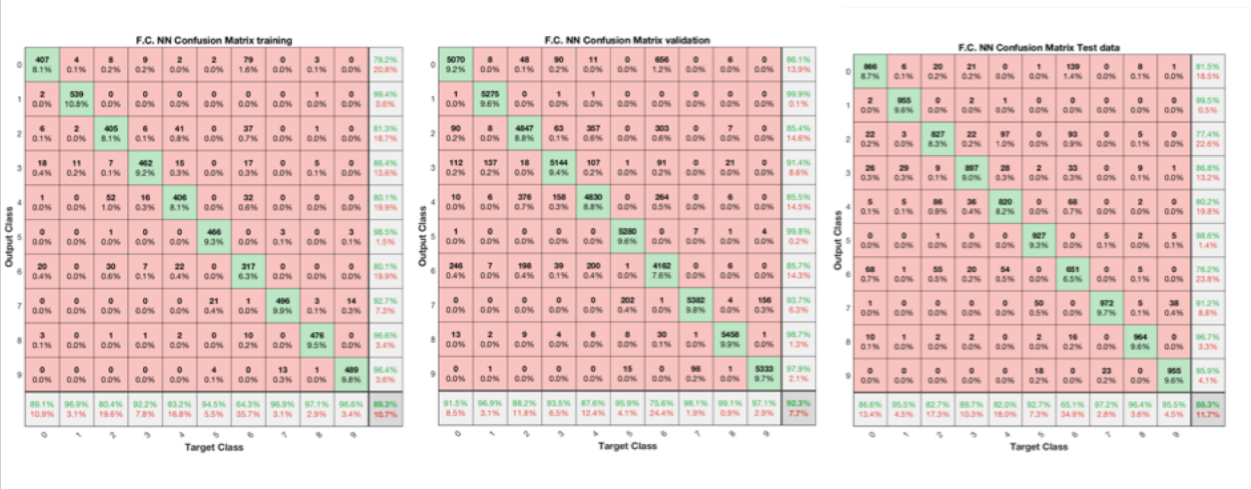
This system consists of the following order:

Input layer
- Convolutional layer 1
  - tanh activation function
    - average pooling layer 1
- Convolutional layer 2
  - tanh activation function
    - average pooling layer 2
- Convolutional layer 3
  - tanh activation function
- Fully connected layer 1
  - tanh activation function
- Fully connected layer 2
  - softmax activation function

Output classification layer

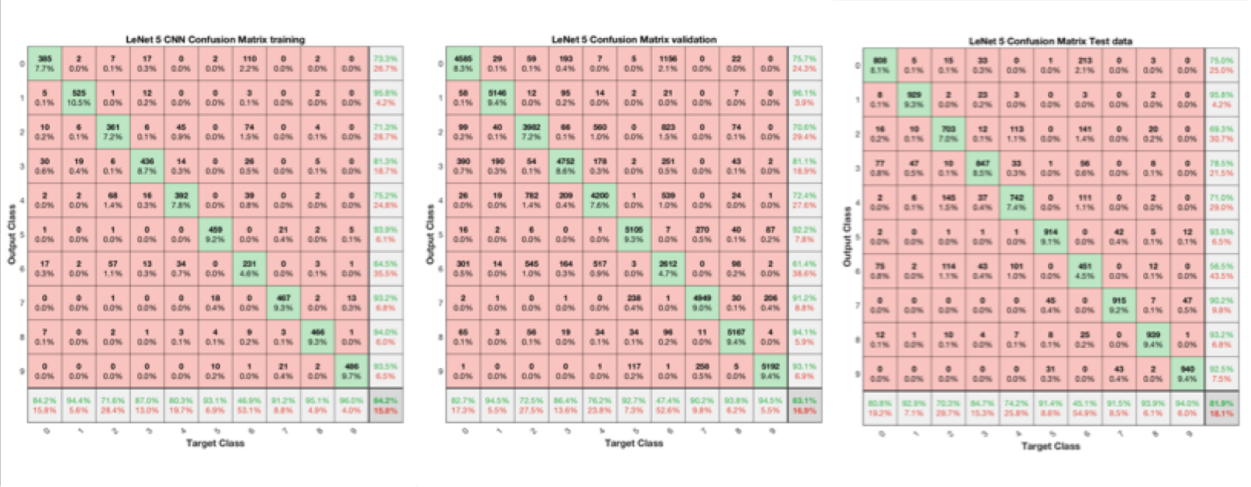
4 Computational Results

In this example we used fully connected neural networks and convolutional neural networks to classify the Mnist fashion data set. Below are the confusion matrix for each test case.



.

Figure 2: Figure shows the confusion matrices for the default fully connected network. Left matrix is training. Middle is validation. Right is training.



.

Figure 3: Figure shows the confusion matrices for the default LeNet 5 convolutional neural network. Left matrix is training. Middle is validation. Right is training.

Network	Epochs	Hidden layers	Layer nodes	Activation functions	Accuracy	Learning rate	L2Reg	Stride
Fullyconnected	10	1	555	Relu, softmax	88.3	**1e-3**	**1e-4**	n/a
LeNet 5	10	3 conv. 2 avgP 2 F.C.	Filters: 6,16,120 84, 10	tanh, softmax	88.04%	**1e-3**	**1e-4**	2,2
Custom CNN	5	3 conv. 2 avgP 3 F.C.	Filters: 16,32,120 100, 84, 10	tanh, tanh, Relu, Relu, softmax	91.4%	**1e-4**	**1e-4**	1,1

.

Figure 4: Figure shows the confusion matrices for a custom convolutional neural network (see table for details). Left matrix is training. Middle is validation. Right is training.

5 Conclusion

In this example we loaded Mnist fashion data and used Neural networks to classify the images. We used fully connected networks and convolutional networks. We used different activation functions and adjusted the hyperparameters.
Through a lot of experimentation, a good network can be produced for any problem. Consider running a parameter search with fminsearch if you have some GPU's and a lot of RAM. One of the best preforming networks for this example was a single layer fully connected network with 550 nodes. We also attempted to binerize the input data because this can sometimes improve the performance of a network training, but this resulted in worse performance for this example.
Lastly, A custom CNN was produced from trial and error experimentation, and this network demonstrated the best results. The main difference with this network was an increase in the number of filters in the first two convolutional layers and a smaller learning rate.