Training Fashion MNIST using Neural Networks

Constance Wang

Abstract

In this problem, the Fashion MNIST dataset was evaluated by training neural networks. The Fashion MNIST dataset contains images of various fashion items, which can be classified into 10 distinct categories for the model to categorize. The main techniques studied were fully connected (dense) neural networks and convolutional neural networks. Convolutional neural networks generally performed better than dense neural networks but took much longer to train.

Section I – Introduction and Overview

Neural networks are a form of supervised machine learning that have become popular in recent years. This is an algorithm was inspired by biological neurons, where information is passed through different nodes to transfer information. Neural networks pass information through nodes, which it uses to predict a final target outcome. For this problem, neural networks will be used for classification (as opposed to regression). Two forms of neural networks will be studied: fully connected (dense) neural networks and convolutional neural networks.

The classification problem that is studied is to sort different fashion items. This is a commonly used classification problem to test the robustness of the neural network, and the dataset is called Fashion MNIST. The training dataset includes 60,000 different articles of clothing, shoes, and accessories that the algorithm needs to categorize. In addition, the dataset includes 10,000 images that are used to test the model's accuracy.

Section II – Theoretical Background

To understand neural networks, basic understanding of how models can be used to predict various parameters must be understood. A simplistic model that is similar to neural networks is logistic regression. In logistic regression, there is a model and a loss function that needs to be defined. The synonymous terms for neural networks is the activation function and a loss function (which is different type of function).

Logistic Regression Model: Sigmoid Logistic Function

The model that is often used for logistic regression is the sigmoid function, which takes the shape of a s-shaped graph (think population growth model). This model is particularly useful for classification because the graph that is used distinguishes between different regions (the two flat regions can represent two different distinct categories), doing so with no discontinuities in the graph. To relate input values with the desired output, a linear function of the inputs and outputs is plugged into the model. The final expression is given in Equation 1 below, which is used in the loss function:

$$y = \frac{1}{1+e^{-(mx+b)}} \qquad \text{Logistic Function with Linear Function (1)}$$

Neural Network Activation Function

The model (known as the activation function) used for neural networks is similar to the sigmoid function. The activation function is a piecewise function where the discontinuity is known as the threshold used to distinguish between different categories. This is given in Equation 2:

$$y = \begin{cases} 0 & z < threshold \\ 1 & z \geq threshold \end{cases} \qquad \text{Neural Network Activation Function (z is input function) (2)}$$

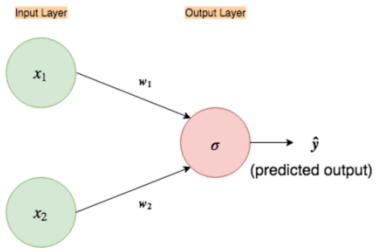


Figure 1: Neural network simple example

The function that relates the inputs and the outputs is a linear system of equations of weights and inputs. In the most simple neural network model given in Figure 1, where the inputs are given by the x values and the weights are given by the w values. The predicted output is given by the inner product between the x values and w values. Note that this is the function that relates the inputs and outputs. This function is plugged into the activation function that the loss function utilizes to perform optimizations of the weights. In addition to the weights, there is typically another input that is added to the network called a bias neuron, which is used for shifting the network.

Loss Function

The loss function utilized in neural networks derive from the loss function used in logistic regression, so both will be discussed in this section. The model used is called cross entropy loss. In logistic regression, the model with the linear function plugged in gives the probability of classifying an input value as one of the two desired classes. This means that the model can be optimized if a desired output is known. In this case, the probability of classifying the input correctly should be maximized. The function used to maximize this probability is called the loss function because losses (incorrect classifications) should be minimized. The function is given in Equation 3 below:

$$L = \frac{-1}{N}\sum_{j=1}^{N} \ln\left(p_j\right) \qquad \text{Logistic Function with Linear Function (1)}$$

Fully Connected vs. Convolutional Neural Networks: Neural Network Structure

Figure 1 is a simplistic neural network, where the input layer leads directly to the output prediction. The output prediction should contain the number of desired classes that need to be categorized, which is a fixed number. The number of inputs is how much data is used for the training. The power of neural networks, however, lies in the fact that multiple layers of nodes can be used to create a network. These extra nodes would be a part of an extra layer(s) sandwiched between the input and output layer, which are called hidden layers. In fully connected neural networks, nodes transfer information from left to right (unidirectionally). Each node from a layer to the left is connected to each node in the layer to its right, and so on and so forth.

Convolutional neural networks have a different structure, and are typically used in image processing. These networks employ the use of filters to sample pixel regions. The first filter layer would pick up simple details, and these details would be built upon using a different filter in the second layer. The layers build up and become more complex to determine the important features of the image.

Section III – Algorithm Implementation and Development

The neural network algorithm was implemented using Python. One feature of neural networks is that they are very easy to overfit. This is because there are many parameters that can be adjusted. Often, thousands of weights and biases can be adjusted, which is a cause for overfitting. To aid in

this, training data for neural networks are often split into training and validation data. The validation data provides in situ measurements of how well the model is training, and helps mitigate effects of overfitting. This is because overfitting will result when the model predicts the model significantly better than the validation data. For the Fashion MNIST dataset, 5000 data was used for validation, and the rest (55,000) was used for training.

To begin classification using neural networks, the Fashion MNIST dataset was loaded into Python (which is a dataset part of the Tensorflow-Keras interface). The data contains grayscale images, all of which are 28x28 in uint-8 format (color scale). These data were all divided by 255.0 to convert to double precision and normalize the data (which is often preferred for neural networks). For convolutional neural networks, an extra axis needs to be created (attributing the data as images).

To create the fully connected neural network architecture, dense layers must be created using tf.keras.layers.Dense( ), and the images must be transferred as a column of pixel values using the Flatten( ) method. The activation function used was "relu" (one of the most popular activation functions) and a regularizer is used to penalize large values of weights, which would otherwise bias the results. For the hidden layers, the "relu" activation function is used but for the output layer, the "softmax" function is used (also popular as a loss function).

To create the convolutional neural network architecture, more parameters are introduced due to the shifted filter that is used. In addition, the convolutional neural network can read the entire 28x28 neural network. Pooling is also a technique used to help reduce the number of pixels. The image can be segmented into different regions, and within these regions, either the maximum pixel or the average pixel can be taken as the representative value, reducing the size of the image. Convolutional layers were made with tf.keras.layers.Conv2D( ), and pooling could be determined using tf.keras.layers.MaxPool2D( ) for taking the maximum or tf.keras.layers.AveragePooling2D( ).

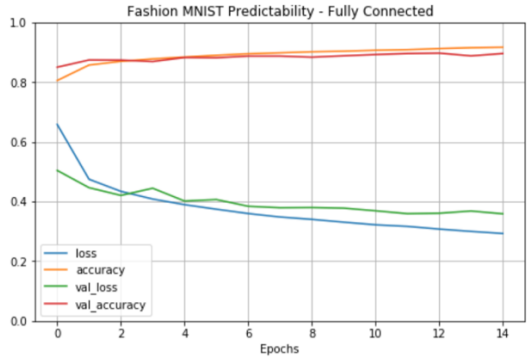
Finally, to wrap up the model, the loss function and optimizer was initialized. Then, the model could be trained using model.fit( ) and the number of training cycles could be chosen. The confusion matrix for this data can be plot to see where the model succeeds and fails to predict (the diagonals are successful predictions while the off-diagonals are incorrect predictions for each category). The model was then used to evaluate the test data, and the confusion matrix was plot once more for this data.

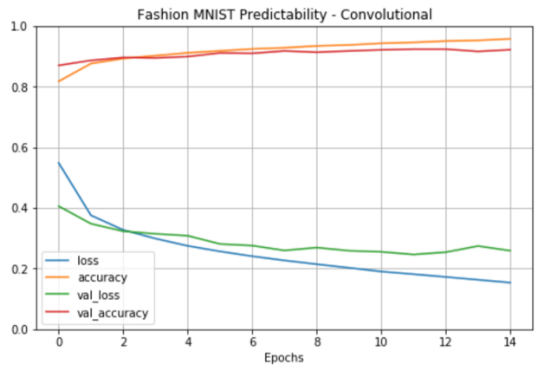
Section IV – Experimental Results

Different testing parameters can be varied to optimize both the fully connected model and the convolutional model. In both cases, it was difficult to reach even 90% accuracy with the data without more time and more complex parameters. Key literature identified the AlexNet, VGGNet, and ResNet as potential neural network architectures, but many of these algorithms could take weeks to train the data. In the final performance of both types of networks:

Neural Network	Training Data Prediction (% accuracy)	Validation Data Prediction (% accuracy)	Test Data Prediction (% accuracy)
Fully Connected	91.6	89.1	**88.3**
Convolutional	95.8	92.2	**92.0**

Table 1: Final results using best neural network architectures tested





	0	1	2	3	4	5	6	7	8	9
0	853	1	26	19	3	2	92	0	4	0
1	2	972	2	20	2	0	2	0	0	0
2	17	0	880	12	55	0	36	0	0	0
3	18	8	18	902	27	0	23	0	4	0
4	0	1	158	31	757	0	51	0	2	0
5	0	0	0	1	0	947	0	39	1	12
6	125	1	130	28	58	0	649	0	9	0
7	0	0	0	0	0	6	0	978	0	16
8	5	0	8	7	3	1	6	5	965	0
9	0	0	0	0	0	8	1	44	0	947

	0	1	2	3	4	5	6	7	8	9
0	907	0	15	8	3	1	62	0	4	0
1	2	989	0	6	2	0	0	0	1	0
2	22	2	865	6	46	0	59	0	0	0
3	20	7	10	909	29	0	23	0	2	0
4	3	1	33	20	880	0	63	0	0	0
5	1	0	0	0	0	989	0	5	0	5
6	118	0	40	19	55	0	760	0	8	0
7	0	0	0	0	0	14	0	947	0	39
8	2	3	2	5	1	3	2	2	980	0
9	0	0	0	0	0	6	0	16	1	977

Figure 2: Predictability of fully connected (top left) network and corresponding confusion matrix (bottom left). Predictability of convolutional (top right) network and corresponding confusion matrix (bottom right).

Considering the difference between the training and validation data for the convolutional neural network, the model likely underwent slight overfitting, explaining the decrease in accuracy for validation and test data prediction. The final architectures can be seen in the code in Appendix D. In general, the "relu" activation function proves to be the best activation function across both networks, and the "softmax" function is the best for the output layers. In addition, the regularizer, loss function, and optimizer parameters were also kept the same across both networks. Most of the original architecture was kept the same for the two networks. The two dense layers from the sample code were kept for dense layers. The LeNet architecture generally was kept with convolutional layers feeding into pooling functions and then finally dense layers. Parameters were tweaked for optimization (mostly hidden layer widths). The figures above show the predictability of the Fashion MNIST and changes in loss, as well as the confusion matrix for the both networks.

Section V – Summary and Conclusions

In this study, fully connected and convolutional neural networks were studied. Convolutional neural networks generally performed better than fully connected neural networks, which make sense because convolutional neural networks are most optimal for images. This is because convolutional neural networks use filters to sample different regions of the images, which pick up the most important pieces of information about the image as a whole. With dense neural networks, each pixel value must be mapped, rather than mapping features. This means that changes in position of the particular object will cause the entire mapping to be completely different with dense layers. However, with convolutional layers, the important features of the image can be extracted so changes in position should not yield a significant difference. Unfortunately, convolutional networks take much longer to train, which can be a drawback depending on the application. In general, both

networks were able to classify the different MNIST images, and accuracy was quite consistent, ranging between 85% and 95%.

Appendix A

- tf.keras.datasets.fashion_mnist.load_data( ) was used to identify and load the Fashion MNIST dataset into Python
- np.newaxis creates a new axis (set to 1), or a new dimension for the matrix, which was used to create a new dimension for the data to be used in the convolutional network
- tf.keras.models.Sequential( ) creates a linear stack of layers (representing the model used for the entire neural network), which was used to create the neural network model for both fully connected and convolutional networks
- partial( ) allows for initialization of parameters of a particular function, which will be used as a template for convolutional and dense layers
- tf.keras.layers.Dense(nodes) was used to create dense layers, and in particular activation functions could be defined (used "relu") and regularizers could be defined (used l2 regularizer)
    - Note that nodes is the number of nodes within that layer
- tf.keras.layers.Flatten( ) creates a column vector from the matrix of pixel values used to define the image data
- tf.keras.layers.Conv2D(filter, size) was used to create convolutional layers, and activation functions could be similarly defined. Padding was not used (creates a border of zeros so the size of the output is the same size as the input). Filter defines the number of filters, and size defines the size (size squared) of the filter.
- tf.keras.layers.MaxPool2D(size) was used to reduce the number of pixels considered using pooling, where size is the size (size squared) of each region to be pooled
- .compile( ) initializes the loss and optimizer functions
- .fit( ) is used to train the data using the training and validation data
- .evaluate( ) uses the model generated on the test data
- confusion_matrix( ) was used to create the confusion matrix