Neural Networks:

Cast Your Net Wide

Tim Ferrin

March 13, 2020

Abstract

Image/pattern recognition has become very important in recent times for its ability to automate a process that historically only humans have been able to do. In this report, neural networks are used to analyze the Fashion-MNIST data set in order to classify various pictures of articles of clothing. Different parameters and their affects on the success of neural networks are discussed.

I Introduction and Overview

Humans are required to recognize patterns with almost everything they do in life. From reading words on a page to looking a picture of their friends to listening to their favorite songs. The last topic was the subject of the previous report "Music Classification: I Walk the Line(ar Discriminant Analysis)", the second topic is discussed in this report. Humans also have an astounding ability to learn new information and skills, and retain that information for long periods of time. We accomplish these feats with the vast array of neurons in our brains and the connections that are made between them. Astoundingly, there are on average 100 billion such neurons in every human brain, with each neuron being connected to about 10,000 other neurons! These processes and systems are far too complicated to model in their entirety, but we can attempt to simplify some things for a good approximation.

In order to mimic the incredible pattern recognition and learning skills that we possess, mathematicians and scientists over the last 150 years have developed simplified techniques to model the "neural networks" that we use for learning. In contrast to humans, computers aren't very good at recognizing patterns, but they are good at simple arithmetic. Hence a mathematical framework using some linear algebra and calculus is used to simulate these processes. Ultimately, such an interconnected neural network provides an interesting model of complicated human and animal physiology that can help provide insight into how our own networks work, while providing machine-learning capabilities useful to tasks that we would eventually like computers to take over entirely.

Figure 1: The first 9 images from the Fashion-MNIST data set. From left to right, top to bottom they are: Ankle boot, T-shirt/top, T-shirt/top, Dress, T-shirt/top, Pullover, Sneaker, Pullover, Sandal

Originally, the MNIST data set existed as a benchmark for testing out neural networks' effectiveness. This data consisted of 28x28 pixel black and white images of handwritten digits, with 60,000 images for training and 10,000 images for testing. Eventually neural networks became so good at classifying these images, the Fashion-MNIST data set started being used, as its patterns are slightly more complicated than handwritten digits. These pictures consist of 10 different articles of clothing, as shown in Figure 1 and ordered from 0 to 9 as T-shirt/top, Trousers, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle Boot. The algorithms in this report utilize two different applications of neural networks in order to attempt to classify this individual pictures into their respective categories.

II Theoretical Background

Before we begin the explanation of neural networks, first we must discuss logistic regression, which will be used for classification. Suppose you have a data set of (x_1, y_1), (x_2, y_2), ... , (x_N, y_N) where y_j is either 0 or 1. Thus a logistic/sigmoid function of the form

$$p = \frac{1}{1 + e^{-(mx+b)}} \tag{1}$$

is used to fit the data, where we want to find the best fit for parameters m and b. Many definitions of "best" exist, but in this case we would like to maximize the following summation known as the log likelihood

$$\frac{1}{N} \sum_{j=1}^{N} y_j \ln p_j + (1 - y_j) \ln 1 - p_j \tag{2}$$

or minimize

$$-\frac{1}{N} \sum_{j=1}^{N} y_j \ln (p_j) + (1 - y_j) \ln (1 - p_j) \tag{3}$$

which is known as the log loss or the cross entropy loss. The use of the natural log is to make the function convex, so that there is only one minimum. In implementation, gradient descent is used to find this minimum, but prior knowledge of such a technique is assumed.

When classifying data into more than 2 groups, the problem shifts into a format that requires linear algebra. If a problem required sorting n data points of x_1, x_2, ... , x_n into m groups, where each individual $y = a_1x_1 + a_2x_2 + ... + a_nx_n + b$, then p becomes a probability vector that is calculated via the following formula,

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \frac{1}{\sum_{j=1}^{m} e^{a_{j1}x_1+a_{j2}x_2+...+a_{jn}x_n+b}} \begin{bmatrix} e^{a_{11}x_1+a_{12}x_2+...+a_{1n}x_n+b_1} \\ e^{a_{21}x_1+a_{22}x_2+...+a_{2n}x_n+b_2} \\ \vdots \\ e^{a_{m1}x_1+a_{m2}x_2+...+a_{mn}x_n+b_m} \end{bmatrix} \tag{4}$$

or

$$\vec{y} = A\vec{x} + \vec{b} \qquad \vec{p} = \sigma(\vec{y}) = \frac{1}{\sum_{j=1}^{m} e^{y_j}} \begin{bmatrix} e^{y_1} \\ e^{y_2} \\ \vdots \\ e^{y_m} \end{bmatrix} \tag{5}$$

This is known as the soft max function and now we can begin our discussion of neural networks.

Many different implementations and modifications of the neural network have been made throughout history, although for this report we will only cover very simplified aspects of it. Essentially, collections of neurons (or nodes) represent the state of a particular data point. In a network with no "hidden layers" that initial state vector is multiplied by a weights matrix which then is fed into an activation function. The state of this vector upon its output is a collection of probabilities for which category the initial state belonged to. Initially in history, the activation function was

$$\sigma(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases} \tag{6}$$

Thus the output of the initial state vector becomes

$$\vec{y} = \sigma(A\vec{x} + \vec{b}) \tag{7}$$

When using multiple layers of this which involve matrices A_1 A_2 and bias vectors $\vec{b_1}$ and $\vec{b_2}$

$$\vec{y} = \sigma(A_2\sigma(A_1\vec{x} + \vec{b_1}) + \vec{b_2}) \tag{8}$$

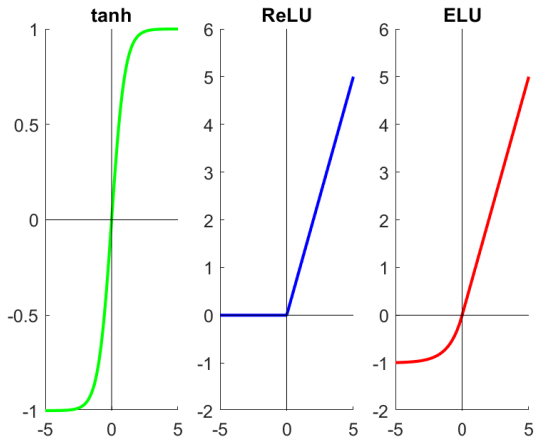

Figure 2: Other possible activation functions which consist of, from right to left, hyperbolic tangent, $\max(x, 0)$ and $\{\alpha(e^x - 1)$ for $x < 0$, x for $x \geq 0\}$.

More recently, the activation functions of hyperbolic tangent and a Rectifide Linear Unit (ReLU), and Exponential Linear Unit (ELU), among others, have been used (See Figure 2).

The first part of this report concerns fully-connected/dense layers of neurons in which all nodes in one layer are connected to all nodes in the next layer. The second part concerns Convolutional Neural Networks (CNNs) which draw on many of the same mathematical tools as dense networks. Th only difference is that periodically throught the algorithm, base level neurons apply a filter to recognize patterns in a specific, square window of pixels. This filter is then applied throughout the whole image to produce a feature map and you can have multiple feature maps (each with different features) for a single convolutional layer.

III Algorithm Implementation and Development

The algorithm for part 1 written in Python is as follows: First, the various packages required to run this code are imported. Then the data from the Fashion-MNIST data set is loaded into variables and the full training data is split into validation data and training data. The validation data is not used when building the network, it is there to verify that the model is not overfit to only the training data, analogous to a finding a 9th degree polynomial to fit 10 linearly correlated data points. The data is also converted from 8-bit integers into numbers between 0 and 1. Then the neural network model is described. This section of the code involves specifying how many layers are in the network, how many nodes are in each layer, what activation function is used, and whether or not regularization is used. Regularization with the 2 norm involves adding the term

$$\lambda \|\vec{w}_j\|_2^2 \tag{9}$$

to the loss function, where $\vec{w}$ corresponds to the weights. This helps us avoid overfitting.

After this, other hyperparamters are stated when compiling the model, including the loss

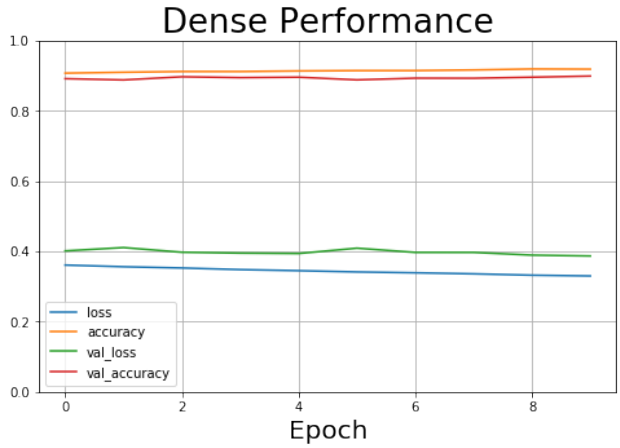

Figure 3: The performance of the Fully-connected neural network over 10 epochs.

function, the optimizer, and the metric to guage the model (in our case we use sparse categorical crossentropy, gradient descent, and accuracy). Finally, the network is trained and the accuracy of the network in classifying the training, validation, and test data is calculated.

The second part is very similar to the first, except that it uses an alternating strategy of convolutional and fully connected layers. Average pooling is used to keep the memory use of the computation down, where the average pixel values of translated windows of the image are used instead of the raw image. Alternatively, max pooling can be used where only the max pixel value is taken. Additionally, the hyperbolic tangent function is used as the activation function and zero padding is used to keep the size of the image the same when using the first convolutional layer. The algorithm shown in Appendix B is an alteration of the LeNet-5 algorithm which was shown to be very successful on the MNIST data set.

IV Computational Results

Through experimentation, the author of this report has tried many different values of many different parameters of the network, including the depth of the network, the width of the layers, the learning rate, the regularization parameters, the activation functions, the optimizer, the number of filters for the convolutional layers, the kernel sizes, the strides, the padding options, and the pooling layers. For the dense network, an accuracy higher than 89.92% on the validation data was never obtained. The results for the best data he could get is shown in Figure 3 and the code is shown in Appendix B. This produced accuracies of 91.89%, and 88.31% for the training and test data. For the convolutional network the corresponding values were 89.48%, 90.65%, and 88.31% respectively. The plots for each are shown in Figures 3 and 4.

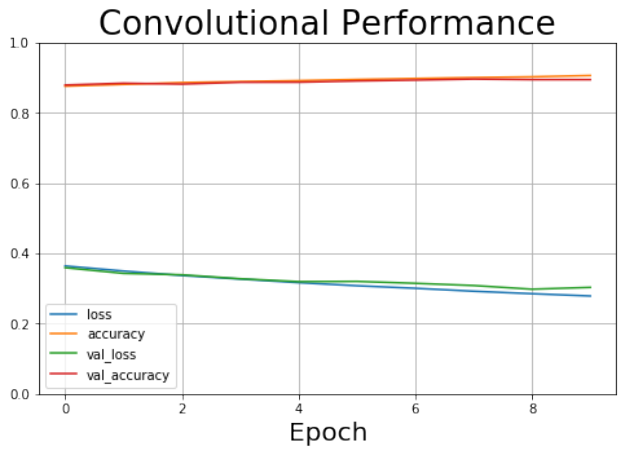

Figure 4: The performance of the Convolutional neural network over 10 epochs.

V Summary and Conclusions

As there are so many different things to tweak with both of these algorithms, it is very difficult to know exactly what is the right combination to produce better results. Many different factors each change the algorithm in subtle ways and as soon as you pin down the optimal value for one parameter, changing another completely throws the previous one out of wack. All in all this sort of analysis, like many that have been discussed in previous reports, is very frustrating and requires a lot of guess and check, but is ultimately very rewarding.

Appendix A: Python Functions/Packages Used and Brief Implementation Explanation

tensorflow – the maching learning package used for the majority of the computation

partial – allows you to repeat layers

fit – trains your network with the given parameters

confusion_matrix – computes the confusion matrix, or what how many images in each category were correctly/incorrectly identified

evaluate – computes the accuracy of the network on the testing data

Appendix B: Python Codes

Fully-connected Neural Network

```
1 #!/usr/bin/env python
2 # coding: utf-8
3
```

```
# In[2]:


import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix


# In[3]:


fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.
    load_data()


# In[18]:


plt.figure()
for k in range(9):
    plt.subplot(3,3,k+1)
    plt.imshow(X_train_full[k], cmap="gray")
    plt.axis('off')
plt.show()

print(y_train_full[:9])


# In[4]:


X_valid = X_train_full[:5000] / 255.0
X_train = X_train_full[5000:] / 255.0
X_test = X_test / 255.0

y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]


# In[56]:


```

```
from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="relu",
    kernel_regularizer=tf.keras.regularizers.l2(0.00045))

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    my_dense_layer(128),
    my_dense_layer(64),
    my_dense_layer(32),
    my_dense_layer(10, activation="softmax")
])


# In[65]:


model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate
                  =0.0001),
              metrics=["accuracy"])


# In[66]:


history = model.fit(X_train, y_train, epochs=10, validation_data=(
    X_valid, y_valid))


# In[70]:


pd.DataFrame(history.history).plot(figsize=(8,5))
plt.title('Dense Performance',fontsize=26)
plt.xlabel('Epoch',fontsize=20)
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()


# In[68]:


y_pred = model.predict_classes(X_train)
```

```
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)


# In[52]:


model.evaluate(X_test,y_test)


# In[ ]:
```

Dense Neural Network

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix


# In[2]:


fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.
    load_data()


# In[3]:


X_valid = X_train_full[:5000] / 255.0
X_train = X_train_full[5000:] / 255.0
X_test = X_test / 255.0

y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]

```

```
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]


# In[60]:


from functools import partial

my_conv_layer = partial(tf.keras.layers.Conv2D, activation="tanh",
    padding="valid")
my_dense_layer = partial(tf.keras.layers.Dense, activation="tanh",
    kernel_regularizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    my_conv_layer(6,5,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(16,5),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(120,5),
    tf.keras.layers.Flatten(),
    my_dense_layer(84),
    my_dense_layer(30),
    my_dense_layer(10, activation="softmax")
])


# In[61]:


model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate
                  =0.00025),
              metrics=["accuracy"])


# In[65]:


history = model.fit(X_train, y_train, epochs=10, validation_data=(
    X_valid, y_valid))


# In[73]:
```

```
73
74
75
76 pd.DataFrame(history.history).plot(figsize=(8,5))
77 plt.title('Convolutional Performance',fontsize=26)
78 plt.xlabel('Epoch',fontsize=20)
79 plt.grid(True)
80 plt.gca().set_ylim(0,1)
81 plt.show()
82
83
84 # In[67]:
85
86
87 model.evaluate(X_test,y_test)
88
89
90 # In[ ]:
```