Amath482 – Winter 2020

Neural Networks for Classifying Fashion MNIST

Chenxi Di (1663044)

March 13, 2020

Abstract

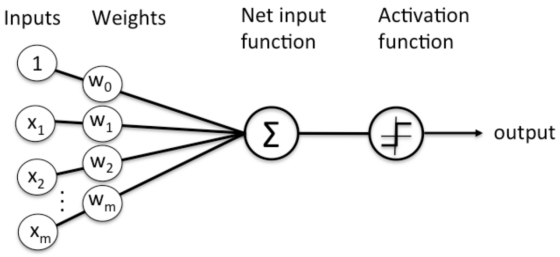
This project aims to train a neural network model to build a classifier for the popular Fashion-MNIST dataset of 10 different classes of fashion items. This project will work through a series of simple neural network architectures and compare their performance on the Fashion-MNIST data.

I. Introduction and Overview

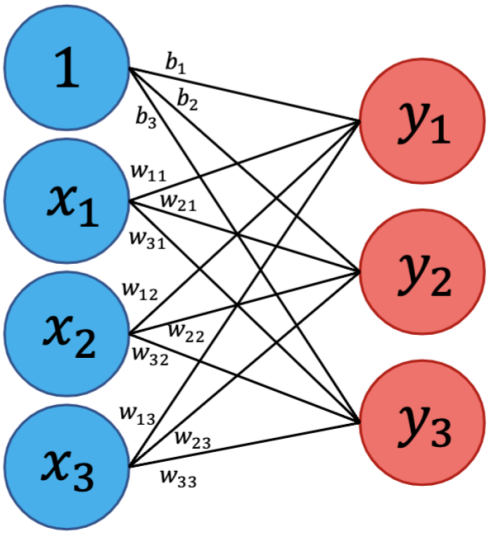
Motivated by the biological neuron, with powerful digital technology and abundant data, neural networks have become a large part of image analysis. The goal of this project is to train a fully-connected neural network model (part 1) and a convolutional neural network model (part2) on a dataset of 10 different classes of fashion items. Through the process, we will try compare different neural network architectures with different parameters to achieve a better performance on the test set.

II. Theoretical Background

Neural networks are a set of algorithms, modelled loosely after the human brain. They are designed to recognize patterns and do classification. They are composed of several layers which are made up of neurons. The layers on the left are called the input layer, and they output same number as they input.



(a) Diagram of Neural Network



(b) Perceptron

Figure 1: Elements of Neural Network

Let the first neuron on the input layer be the bias neuron which always outputs a 1. Let x1,x2, ... x_m be the other neurons/nodes on the input layer. And each connection has a weight (w1, w2, ... w_m) associated with it. Then the input-weight products are summed and the sum is passed through a so-called activation function, to determine whether and to what extent that signal should progress further through the network to affect the ultimate outcome, for example classification.

A perceptron is composed of a layer of linear threshold units. Each one is connected to the input layer and has a different set of weights. Let

$$x = \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}, A = \begin{pmatrix} w11 & w12 & w13 \\ w21 & w22 & w23 \\ w31 & w32 & w33 \end{pmatrix}, b = \begin{pmatrix} b1 \\ b2 \\ b3 \end{pmatrix}, y = \begin{pmatrix} y1 \\ y2 \\ y3 \end{pmatrix}$$

Then the calculation of y can be written as

$$y = \sigma(Ax + b) \tag{1}$$

This σ is an activation function. By combining multiple perceptrons, we could solve more complicated problems. This generats a multilayer perceptron(MLP). Written as an equation, this neural network is

$$y = \sigma(A_2\sigma(A_1x + b_1) + b_2) \tag{2}$$

Some popular activation function includes:

1. the hyperbolic tangent function which has the same shape as logistic function but from -1 to 1:

$$\sigma(x) = tanh(x) \tag{3}$$

2. the rectified linear unit(ReLU) which zero out all the entries that are negative and only keep the positive entries:

$$\sigma(x) = max(0, x) \tag{4}$$

3. the Softmax to map the non-normalized output of a network to a probability distribution over predicted output classes:

$$\sigma(x) = \frac{e^{x_i}}{\sum_{i=1}^{K} e^{x_i}}, \text{i} = 1 \ ... \ \text{K} \tag{5}$$

If for a given layer, every neuron is connected to every neuron in the previous layer and all of the layers are of that type, we say that it is a fully-connected network. The number of neurons in each layer is called the width. The number of layers is the depth of the network. A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in multilayer neural network. The input is m * m * r image where m is the height and width of the image and r is the number of channels (grey image r = 1). The convolutional layer will have k filters (or kernels) of size n * n. Each map is then subsampled with mean/max pooling over p * p contiguous regions where p ranges from 2 for small images to 5 for large images. For classification, we could use softmax function for the output layer and then use the sparse categorical crossentropy as loss function. To find the weights/bias that minimizes the loss, we train and update our model. Optimization algorithm controls how the weights are adjusted. Some common optimizers are Adaptive Moment Estimation (Adam) and Stochastic Gradient Descent (SGD) which is a variant of gradient descent only computing on a random selection of data examples. Adam is a variation of mini-batch gradient descent with momentum. Overfitting happens when the model fits the training set very well but does terrible performance on prediction. We could adjust the model to avoid overfitting by comparing the result in validation set and training set and use some regularization methods such as L2 regularization where we add a component that will penalize large weights.

III. Algorithm Implementation and Development

Part I Fully Connected Network

For part 1, we will adjust the width and depth of the fully-connected network and compare the performance using different optimizers and different regularization parameters, learning rate:

Experiment 1: We will start with a simple 3-layer Neural Network using Adam optimizer(learning rate = 0.001) and L2 regularization(0.0001): in the first layer, we flatten the data. The second layer is a dense layer which uses ReLu activation function and has 516 neurons. The last layer is a dense layer with a softmax activation function and 10 neurons that classifies 10 categories. At the last/15th epoch, the validation accuracy achieves 89.44% while the training set accuracy achieves 90.2%

Experiment 2: Then, we add two more hidden layers (one with 300 neurons, another with 128 neurons) to see if more layers mean higher accuracies. At the last/15th epoch, the validation accuracy achieves 88.06% while the training set accuracy achieves 89.36%. So there is no improvement when our model is deeper.

Experiment 3: Furthermore, we will again use the simple 3-layer Neural Network, but we will use Stochastic Gradient Descent (SGD) as optimizer. At the last/15th epoch, the validation accuracy achieves 84.3% while the training set accuracy achieves 84.5%. So Stochastic Gradient Descent (SGD) worsen the model.

Experiment 4: Lastly, we will use the simple 3-layer Neural Network, but we put 128 neurons on the second dense layer to see if width affects performance. At the last/15th epoch, the validation accuracy achieves 88.48% while the training set accuracy achieves 90.22% So probably wider layer will be better.

Experiment 5: Then, I repeat the Experiment 1 but try different learning rate and regularization parameter. By changing the regularization parameter from 0.0001 to 0.001, validation set accuracy becomes more closed to the training set accuracy, i.e, it reduces the probability of overfitting. Eventually, at the 15th epoch, the validation accuracy achieves 86.86% while the training set accuracy achieves 86.98%. On the other hand, when I decrease the learning rate by ten times, the validation accuracy achieves 88.7% while the training set accuracy achieves 90.43%. Based on my previous experiments, I choose Experiment 1's model as final model for fully-connected network.

Part II Convolutional Neural Networks

For part 2, we will adjust the convolutional neural net work structure, the number of filters for convolutional layers, kernel sizes and pooling layers. The following models are all built/compiled based on Adam optimizer(learning rate = 0.001) and L2 regularization(0.0001). The last layers use softmax activation function.

Experiment 1: We then try Input -> convolutional layer(numfilter = 6, kernelsize = (5,5), zero-padding) -> averagePooling layer(poolsize = 2) -> convolutional layer(numfilter = 16, kernelsize = (5,5), no padding) -> averagePooling layer(poolsize = 2) -> convolutional layer(numfilter = 120, kernelsize = (5,5), no padding) -> flatten ->dense(84 nodes) -> out, using tanh activation. At the last epoch (5th), the validation accuracy achieves 88.47% while the training set accuracy achieves 88.58%.

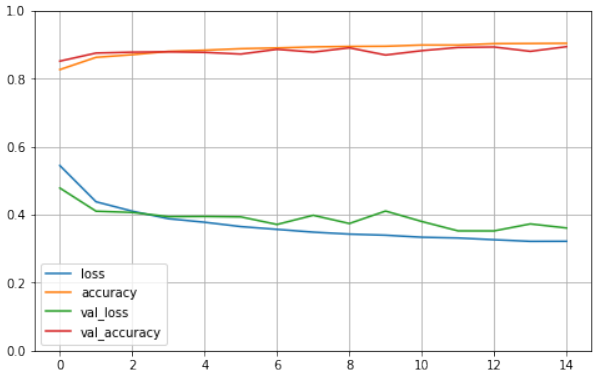
Experiment 2: We then try using Relu activation for the same structure in Experiment 1. At the last epoch (5th), the validation accuracy achieves 90.52% while the training set accuracy achieves 91.2%.

Experiment 3: From here, I change the average pooling layers to Maxpooling layer and increase the number of filters and reduce kernel size: Input -> convolutional layer(numfilter = 32, kernelsize = (3,3), zero-padding) -> MaxPooling layer(poolsize = 2) -> convolutional layer(numfilter = 64, kernelsize = (3,3), no padding) -> MaxPooling layer(poolsize = 2) -> flatten ->dense(256 nodes) -> out, using Relu activation. At the last epoch(5th), the validation accuracy achieves 92.48% while the training set accuracy achieves 93.01%. Based on previous experiment, I choose Experiment 3's model as final model for Convolutional Neural Networks.

IV. Computational Results

Fully-connected Network

The final model is listed in Experiment 1. Note that the training accuracy is very closed to the validation accuracy, so there seems no overfitting. On the test set, the prediction accuracy achieves 88.8%.

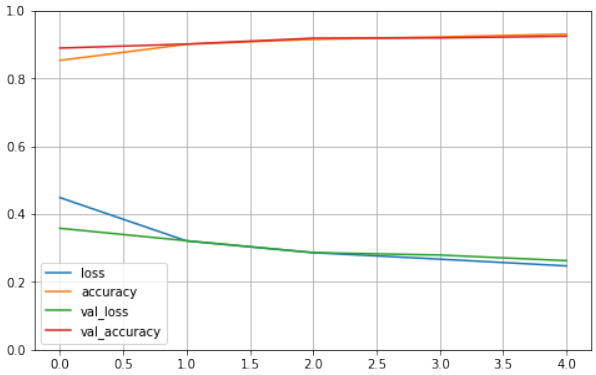


(a) Validation and Train Set Accuracy and Loss

	0	1	2	3	4	5	6	7	8	9
0	781	1	32	49	1	2	126	0	8	0
1	2	971	1	22	1	0	2	0	1	0
2	8	2	803	14	101	0	71	0	1	0
3	12	6	17	934	10	0	16	0	5	0
4	0	1	88	51	803	0	52	0	5	0
5	0	0	0	2	0	901	0	68	0	29
6	92	1	85	46	60	0	705	0	11	0
7	0	0	0	0	0	4	0	978	0	18
8	5	0	4	5	6	6	8	5	961	0
9	0	0	0	0	0	5	1	51	0	943

(b) Confusion Matrix on the Test Set

Convolutional Neural Networks



(a) Validation and Train Set Accuracy and Loss

	0	1	2	3	4	5	6	7	8	9
0	882	0	14	17	2	1	81	0	3	0
1	1	980	0	16	1	0	1	0	1	0
2	20	1	889	8	40	0	42	0	0	0
3	8	0	9	945	13	0	25	0	0	0
4	0	0	72	24	863	0	40	0	1	0
5	0	0	0	0	0	994	0	4	0	2
6	109	0	68	30	62	0	727	0	4	0
7	0	0	0	0	0	11	0	971	0	18
8	5	0	3	5	1	2	3	3	978	0
9	0	0	0	0	0	13	1	32	0	954

(b) Confusion Matrix on the Test Set

The structure of final model is listed in Experiment 3. Note that the training accuracy is very closed to the validation accuracy, so there seems no overfitting. On the test set, the prediction accuracy achieves 91.9%.

V. Summary and Conclusions

Over the course of the study, we trained both of the fully connected and convolutional neural networks to classify 10 different classes of fashion item. Based on previous result, the convolutional neural network, which is more complex than fully connected neural network, performed better on classification. It is also noticeable that both the networks relatively more often misclassified between class 0 and 6, between 2 and 4 from the confusion matrix on the test set. This makes sense since 0(Top) and 6(Shirt) are similar fashion items and 2(Pullover) and 4(Coat) have similarity.

Appendix A MATLAB functions used and brief implementation explanation

`plt.figure(figsize=(n,n))` plot figures with size n * n

`partial()`: partial functions allow one to derive a function with x parameters to a function with fewer parameters and fixed values set for the more limited function.

`tf.keras.layers.Dense(units = n , activation = , kernelregularizer = )`: create a regular densely-connected NN layer with units = n, activation function and regularizer function applied to the kernel weights matrix.

`tf.keras.regularizers.l2()`: create a regularizer that applies an L2 regularization penalty.

`tf.keras.Sequential()`: create linear stack of layers.

`tf.keras.layers.Flatten()`: flattens the input but does not affect the batch size.

`model.compile(loss=, optimizer=, metrics=)`: compile defines the loss function, the optimizer and the metrics.

`Xtrain, ytrain, epochs=, validationdata=`: trains the model for a fixed number of epochs (iterations on a dataset).

`pd.DataFrame(history.history).plot(figsize=)` : plot the training history.

`model.evaluate(Xtest,ytest)`: evaluate model accuracy on test dataset.

`model.predictclasses(Xtest)`: predict the categories on the test dataset.

`confusionmatrix(ytest, ypred)`: generate confusion matrix for the test dataset.

`tf.keras.layers.Conv2D(filters, kernelsize, strides, padding)`: create 2D convolution layer (e.g. spatial convolution over images) and specify the number of filtering/kernels, the size of the filter matrix, the strides, either zero padding or no padding.

`tf.keras.layers.MaxPooling2D(poolsize=)`: Max pooling operation for spatial data and specify the pool layer size.

Appendix B MATLAB codes

```
# import packages
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
```

```
from sklearn.metrics import confusion_matrix

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

# figure view
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_train_full[i], cmap="gray")
    plt.xlabel(class_names[y_train_full[i]])
plt.show()

# construct validation set
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:]

# build model using relu
# two layor(200, 100): 85.5 / one layor (512):86
# /optimizer: sgd, 88% tf.keras.optimizers.SGD(learning_rate=0.01,
momentum=0.8, nesterov=False), default
# / learning rate adam(0.0001:88)
from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense,
```

```
activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    my_dense_layer(516),
    my_dense_layer(10, activation="softmax")
])

# compile model

model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              metrics=["accuracy"])

# accuracy
history = model.fit(X_train, y_train, epochs=15, validation_data=(X_valid,y_valid))

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()

model.evaluate(X_test,y_test)

y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)

y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots()

# hide axes
fig.patch.set_visible(False)
```

```
ax.axis('off')
ax.axis('tight')

# create table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10),
colLabels=np.arange(10), loc='center', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')

### part2 ####
# construct validation set
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]

from functools import partial

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

model = tf.keras.models.Sequential([
    my_conv_layer(32,3,input_shape=[28,28,1]),
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(64,3),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Flatten(),
```

```
    my_dense_layer(256),
    my_dense_layer(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=5, validation_data=(X_valid,y_valid))

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()

y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)

model.evaluate(X_test,y_test)

y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots()

# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')

# create table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10),
colLabels=np.arange(10), loc='center', cellLoc='center')
```

```
fig.tight_layout()
plt.savefig('conf_mat.pdf')
```

Reference

J. Nathan Kutz. (2013) Data-Driven Modeling and Scientific Computation: Methods for Complex Systems and Big Data. Oxford University Press. ISBN:978-0-19-966034-6
Irene Pylypenko, Exploring Neural Networks with fashion MNIST,
https://medium.com/@ipylypenko/exploring-neural-networks-with-fashion-mnist-b0a8214b7b7b
Enric Rovira, https://www.kaggle.com/enric1296/getting-started-fashion-mnist-0-94
Convolutional Neural Network,
http://deeplearning.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/
Tensorflow, https://www.tensorflow.org/