Homework 5: Neural Networks and Classifying Fashion MNIST

Joshua Yap
AMATH 482
Winter 2020

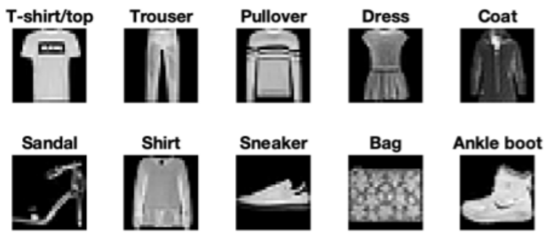
March 14, 2020

Abstract

This homework walks through optimizing a neural network to classify fashion_mnist data using two different types of networks. The emphasis is on fine-tuning model hyperparameters to get optimal results while avoiding overfitting.

1 Introduction

In class we looked at how neural networks can be used to classify an MNIST dataset of handwritten digits. This homework extends the application of neural networks to also classify a fashion MNIST dataset consisting of images of clothing items. One image of each class is shown in figure 1(a). The dataset has 60,000 training images and 10,000 test images, each of which is 28×28 in size. The labels for the data are integers from 0 to 9 that map to the classes as in table 1(b). The task is split into 2 parts; Part I requires us to classify the images using a fully-connected neural network; Part II requires us to classify them using a convolutional neural network. The aim is to tune the hyperparameters in order to optimize the network and obtain the best classification accuracy on the test data.



(a) Example images of each class in fashion_mnist.

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

(b) Labels corresponding to classes.

Figure 1: Fashion_MNIST dataset information.

2 Theoretical Background

A neural network is a machine learning algorithm based on biological neurons that allows a computer to take in information it has not processed before and output a prediction (usually a classification or regression) about that input. In order to do this, the network must be trained on a set of training data so that it can recognize patterns.

2.1 Fully-connected Neural Networks

Every neuron in one layer is connected to every neuron in the adjacent layers. The two most basic layers are the input and output layers. The width (number of neurons) of these layers are determined by the size of the input data and the number of classes respectively. In our problem, the input layer has 784 neurons and the output layer has 10 neurons. We have the option to add hidden layers between the input and output layers to capture more complexity in the data.

The connections between neurons are weighted such that certain patterns of activated neurons in one layer activates a neuron in the next layer. These weights are determined during training to best predict the output. An activation function takes the weighted inputs from all neurons in the preceding layer and determines what value to output. Common activation functions include sigmoids (s-shaped functions) like hyperbolic tangent or the logistic function as well as the Recitfied Linear Unit (ReLU). The output can be expressed as

$$\vec{y} = \sigma(\mathbf{W}\vec{x} + \vec{b}), \tag{1}$$

where

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}, \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}, \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}. \tag{2}$$

$\mathbf{W}$ is the weight matrix and $\vec{b}$ are the bias terms in the network which normalize the threshold activations to zero.

2.2 Convolutional Neural Networks (CNNs)

CNNs move a kernel over an image to produce a feature map. The kernel in this case is a 5×5 matrix of numbers that are multiplied element-wise with the part of the image it overlaps. We use multiple kernels to look for different features and combine the features to find patterns. Each convolution of a kernel with an image is called a feature map, and multiple feature maps are combined to produce a convolutional layer. We combine convolutional layers with fully-connected layers to build the network.

2.3 Overfitting

A concern with neural networks is that the network may simply memorize patterns in the training data and therefore perform well in training but may be unable to accurately predict output for new data. Neural networks therefore need a lot of data to train on and they need to be tested on new data when tuning hyperparameters to ensure that they are not being overfit.

3 Algorithm Implementation and Development

Pre-process data: The training data is split into 55,000 training inputs and outputs, and 5,000 validation inputs and outputs. This allows us to ensure the model is not being overfit.

Train fully-connected network: Start with a baseline network shown in class and adjust hyperparameters to see if we can increase accuracy of the validation data. These hyperparameters include network depth, layer width, learning rate, regularization parameters, activation functions and the optimizer.

Test optimized model on test data: Once we have found hyperparameter settings that give us good validation accuracy, we run the model on the test data to see how well it does on data it has not seen before.

Build CNN: The CNNs used in this homework follow the template of LeNet-5, a multi-layer neural network built by Yann LeCun. It alternates between convolutional layers and pooling layers (these decrease the size of layers), finishing with a fully-connected layer.

Train CNN: Instead of using a fully-connected network, we now use a CNN on the same data and adjust a different set of hyperparameters including the number of kernels, kernel sizes, strides, padding options and pool sizes.

Test CNN: Once the optimal settings have been found, run this model on the test data.

4 Computational Results

4.1 Part I

The baseline hyperparameters for the network were as shown in table 1 and the corresponding learning curves are in figure 2. These settings gave a validation accuracy of 0.8770, meaning the model predicted 87.7% of outputs correctly. I started exploring by adjusting the number of hidden layers and the width of these layers. I then increased the learning rate to 0.01, which gave poor results. I then tried L1 regularization which gave

Table 1: Baseline hyperparameters for fully-connected network

Depth	Width	Learning Rate	Regularization	Activation Function	Optimizer
4	784,300,100,10	0.001	L2(0.0001)	relu	Adam

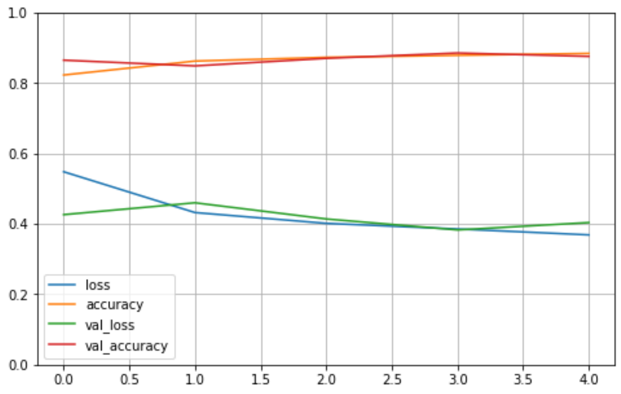


Figure 2: Learning curves for baseline hyperparameters. The focus on this was to make sure the val_loss curve stayed below the loss curve to prevent overfitting.

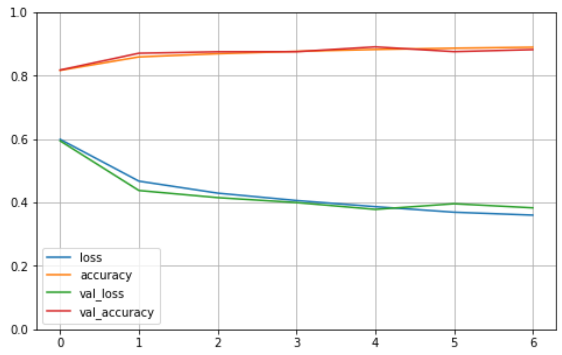
similar results. When changing activation functions to tanh and sigmoid, I found that they did not improve the model. The learning curves looked different for optimizers SGD and Adagrad. While all of these use gradient descent algorithms, it was obvious to me that I was converging to a minimum in the loss function when using SGD by the shape of the learning curves. It took more epochs to converge but I could be fairly certain that it would not overfit the data. The val_loss and loss curves for SGD looked like exponential decay with the loss curve always staying above even when run for 15 epochs. However, I could not achieve the same accuracy as with Adam even though I was more confident that I would not be overfitting the model.

The final model chosen had hyperparameters in table 2 and learning curves in figure 3(a). The validation accuracy for this model was **0.8910** and the accuracy on test data was **0.8683**.

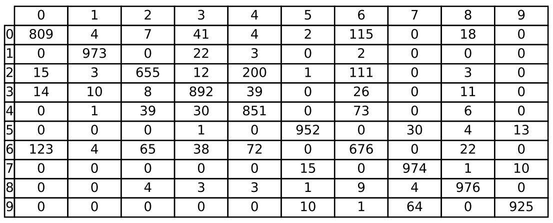
The fact that the model performed better on validation data than test data indicates that it was overfit, so I concluded that there must be other metrics to indicate that other than the learning curves, which did not indicate overfitting. Figure 3(b) shows the confusion matrix, an indication that the model failed most when the input was Coat but it predicted Pullover.

Table 2: Final hyperparameters for fully-connected network

Depth	Width	Learning Rate	Regularization	Activation Function	Optimizer
6	784,400,400,300,100,10	0.001	L2(0.0001)	relu	Adam



(a) Learning curves for final hyperparameters. The curves are shown to 7 epochs to find out where overfitting started. Test data was run to 5 epochs using this model.



	0	1	2	3	4	5	6	7	8	9
0	809	4	7	41	4	2	115	0	18	0
1	0	973	0	22	3	0	2	0	0	0
2	15	3	655	12	200	1	111	0	3	0
3	14	10	8	892	39	0	26	0	11	0
4	0	1	39	30	851	0	73	0	6	0
5	0	0	0	1	0	952	0	30	4	13
6	123	4	65	38	72	0	676	0	22	0
7	0	0	0	0	0	15	0	974	1	10
8	0	0	4	3	3	1	9	4	976	0
9	0	0	0	0	0	10	1	64	0	925

(b) Confusion matrix for test data. The largest off-diagonal element shows us what the model predicted wrongly the most.

Figure 3: Final results for Part I

4.2 Part II

The initial network architecture of the CNN has the same structure as the final one in figure 5, except it had different hyperparameters like number of kernels and pooling options. Using the Adam optimizer and tanh activation function, this yielded a validation accuracy of 0.8838. I tried other activation functions like relu and elu, as well as other optimizers. As before, I needed to step up the learning rate with SGD and Adagrad and run it to 10 epochs, but it still did not give me a higher accuracy. I found that changing average pooling to max pooling improved performance. I also found success reducing the strides (how far we move the kernel each step) to 1, but the runtime was too long. I therefore tried increasing the kernel sizes to 4 and keeping the strides at 2 so there were the same number of steps per iteration. This worked well. Reducing the dense layer to 64 neurons improved the model further. The network architecture I finally settled on is shown in figure 5. Not shown in the figure are the activation functions (elu) and optimizer (Adam). After 4 epochs, this model yielded a validation accuracy of **0.9098** and a test accuracy of **0.9033**. Not only was this higher

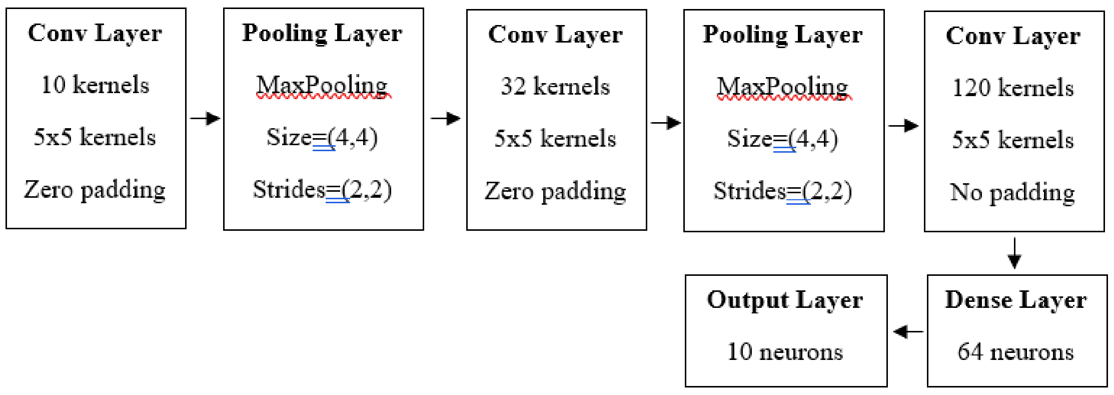
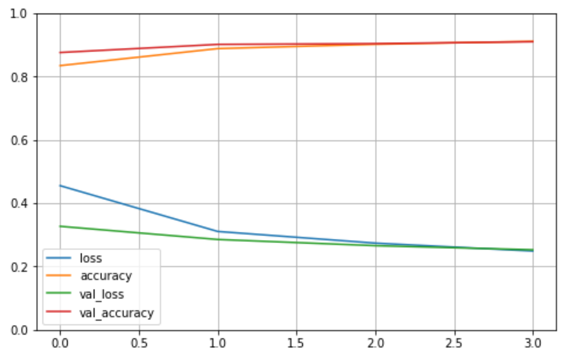
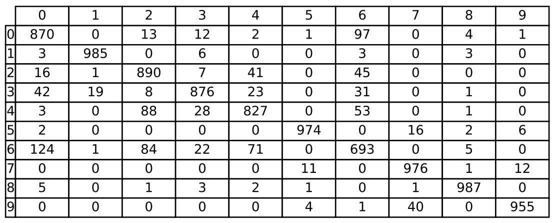


Figure 4: Final network architecture.



(a) Learning curves for final model.



	0	1	2	3	4	5	6	7	8	9
0	870	0	13	12	2	1	97	0	4	1
1	3	985	0	6	0	0	3	0	3	0
2	16	1	890	7	41	0	45	0	0	0
3	42	19	8	876	23	0	31	0	1	0
4	3	0	88	28	827	0	53	0	1	0
5	2	0	0	0	0	974	0	16	2	6
6	124	1	84	22	71	0	693	0	5	0
7	0	0	0	0	0	11	0	976	1	12
8	5	0	1	3	2	1	0	1	987	0
9	0	0	0	0	0	4	1	40	0	955

(b) Confusion matrix for CNN.

Figure 5: Network architecture and performance.

than for the fully-connected network, it also did not exhibit as much of a performance decline for validation to test. Figure 5(b) shows the confusion matrix for the CNN. In contrast to the fully-connected network, the CNN had the most trouble distinguishing a Shirt from a T-shirt/top.

5 Summary and Conclusions

The CNN performed better on this dataset, perhaps because it can identify more complex patterns than the fully-connected network. In both cases, the template code given already produced decent results, so it was a challenge to find marginal gains.

Appendices

Appendix A

Summary of Python functions used

Sequential: Creates a linear stack of layers to add to the model.

Dense Creates a dense (fully-connected) layer, allowing users to specify width, activation function, etc.

compile: Configures the model for training. Loss function, optimizer and metrics can be specified.

fit: Trains the model and identifies optimal weights and biases. Number of epochs can be specified.

Conv2D: Creates a 2D convolutional layer where a kernel is convolved with the input layer.

MaxPooling2D: Pools data to reduce spatial size, performs Max operation on each pool.

Appendix B

Python code

```
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()


# Remove 5,000 images from X_train to use as validation

# Convert ints to floats

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:]


# Train fully-connected neural network

from functools import partial


my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regularizer=tf.keras.regul


model = tf.keras.models.Sequential([

    tf.keras.layers.Flatten(input_shape=[28, 28]),

    my_dense_layer(400),

    my_dense_layer(400),

    my_dense_layer(300),

    #my_dense_layer(100),

    my_dense_layer(100),

    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))
model.evaluate(X_test,y_test)

# CNN
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]

# Train Convolutional Neural Network
from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="elu", kernel_regularizer=tf.keras.regula
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="elu", padding="valid")

model = tf.keras.models.Sequential([
    my_conv_layer(10,5,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.MaxPooling2D(4, strides=(2,2)),
    my_conv_layer(32,5,padding="same"),
    tf.keras.layers.MaxPooling2D(4, strides=(2,2)),
    my_conv_layer(120,5),
    tf.keras.layers.Flatten(),
    my_dense_layer(64),
    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=4, validation_data=(X_valid,y_valid))
model.evaluate(X_test,y_test)
```