Machine Learning Project Report
Image Classification on the Fashion-MNIST dataset

Horatiu Andrei Palaghiu

1. Introduction

The first MNIST dataset (of handwritten digits) has been used as a benchmark to learn and validate classification algorithms. Yet, the researches at Zalando suggest this dataset is too easy(with even classic ML algorithms achieving 97% accuracy easily).Thus, they created Fashion-MNIST: a dataset that better represents modern tasks, and it provides a slightly greater challenge for different classification algorithms [1].

For this task I wished to experiment with and fine-tune different classification algorithms, while comparing them to check which one provides a better solution. I looked on how other people's codes performed on the dataset online and I decided that at least 90% accuracy on the test set without too much overfitting on the train set was attainable (for newbies), so I set that as my goal.

2. The Dataset

2.1. Description

The dataset is split into a training set (of size 60.000) and test set (of size 10.000) of 28x28 grayscale images of clothing pieces. each of them is associated with a label from 10 classes: *T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag* and *Boot*.

In order to test different image classification techniques, I chose to load the built-in Keras dataset with the un-flattened images.

2.2. Preprocessing

For preprocessing, I just rescaled all the pixel values into floats on the interval [0,1] for faster computation. I also hot-encoded my labels, as this usually provides better predictions than single labels. For my K-NN and Random Forrest algorithms I chose to flatten my matrix data into vectors for further improvement on runtime.

Besides that, I found that the dataset already looks quite nice: there are no missing values and the numbers of items per label are quite balanced. Moreover, from the description of the dataset I understood that the images do not have any noise, which is cool.

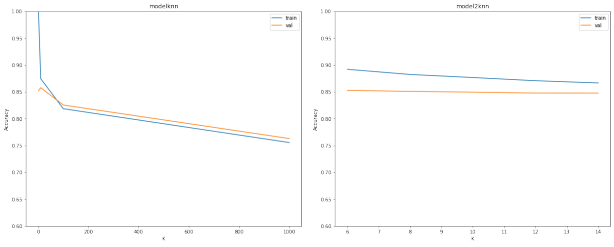
3. Methods and Experiments

Since the dataset I have is quite large and I didn't want to make assumptions about the linearity of the classes, I chose to skip using Logistic Regression. Even if the methods I chose cost more time than a simple regression, I was confident that K-NNs, CNNs, Decision Trees and Random Forrests would obtain a better accuracy.

3.1. KNearestNeighbours

This is the first approach that came to my mind, as the value of each pixel was rescaled in [0,1] and the data was large. Given the nature of my data, the Euclidian distance is the best choice for a metric. Furthermore, I chose to convert my 28x28 data into an 784 number array for smoother computation. The only hyperparameter left to tune was K in order to improve my results. I first tried to compute it for all the k values from 1 to 100($\approx \sqrt{60000}$), but it was taking eons. Then, in order to approximate k better I tried all the powers of 10 from 1 to 1000. I included 2 extreme cases in order to see how the others compare with them. Based on validation accuracy I concluded that the number of neighbours necessary for a 85% accurate calculation was small, around 10. I continued to test some small values around 10, and as expected I obtained more or less the same accuracy on the test.

I chose to stop here with my experiments as I could not make a big leap in accuracy by tuning anything more.



Accuracy plots for different values of K

3.2. Decision Trees

The decision tree is one of the most common algorithms used in machine learning, applied for both classification and regression problems, so I obviously gave it a try. The choice of the various hyperparameters was conducted using a grid searcher algorithm (i.e. GridSearchCV), providing it with different parameters to be tuned. Using the train and validation sets, the grid searcher algorithm uses a standard 5-fold cross validation. After running multiple tests, it provided us with a max_depth=10 and the criterion='entropy' as the optimal hyperparameters.

```
dt_searcher.best_estimator_
```

```
DecisionTreeClassifier(criterion='entropy', max_depth=10)
```

3.3. Random Forrests

The random forest algorithm grows a number decision trees that make their prediction, then it chooses the best classification using a major voting system that "awards" the best as the one that has the most votes. Employing the same dataset used for the decision tree, I deal with the tuning of the hyperparameters implementing the GridSearchCV algorithm. Here the best values are criterion='entropy', max_depth=20, n_estimators=50.

```
print(rf_v1.best_estimator_)
```

```
RandomForestClassifier(criterion='entropy', n_estimators=50)
```

3.4. Convolutional Neural Networks

I kept this method for last since it requires more experimentation. At first I quickly coded a 4 layer NN and I got a pretty good accuracy score of 87% on the test set. But after I tried playing with the number of layers and the other hyperparameters my results were the same, mostly even worse. I was close, yet far from my goal, so I decided I needed a game-changer. Upon extensive googling for something that works best for image classification that even I could understand, I stumbled upon convolution and pooling layers:

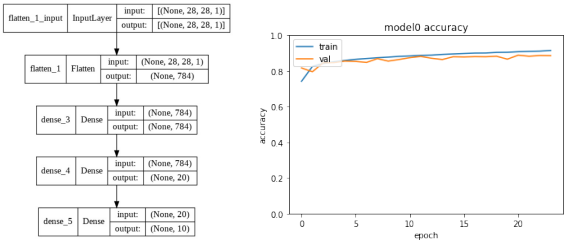
The **Conv2D** layer takes small kernels of chosen height and width and moves them around the image to 'scan' for important features, then it adds them up to obtain a final convoluted image [2].

Pooling is then applied after each convolution layer to reduce the number of dimensions of the output (for maxpooling it takes the largest value among the pixels in the filters). This proved, as expected, to make my algorithm much faster [3].
Why not use different strides for the Conv2D layer instead of pooling? Good question. I did, thinking that even if it is way slower, it might provide better results. But for some reason, in the case of this data, I was wrong.

3.4.1 Classic NNs

First, I constructed a 4-layer NN with an early stopping algorithm that monitors the validation loss of patience 3 (which I kept for the rest of the models too, hence the large number of epochs). Initially, I loaded the data as a vector and that's how I used it for all my models for faster training time.

I saw that more layers $\neq$ better model, no matter the hyperparameters. Changes made in the batch size (from my initial 32) or the activation function for each layer (from ReLu) made the model even worse, with my record in accuracy (for the worst results) being around 79%. Very small improvements on this first model were made by experimenting with different numbers of units on the second layer, and by decreasing my validation split from 0.2 to 0.1.



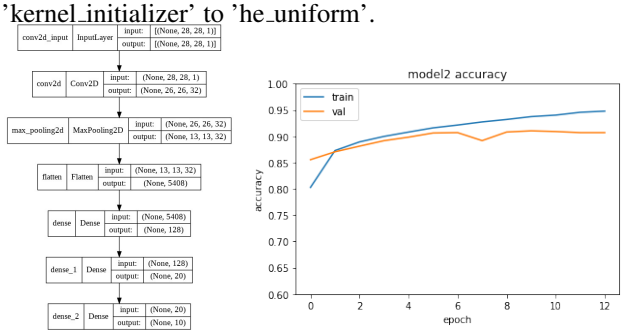
3.4.2 Implementing Conv2D and MaxPool2D

Than I began studying the low-hanging fruits of CNNs and I stumbled upon convolution layers [2]. At first I only applied one Conv2D layer with 32 filters and a filter size of 3x3, since my images were already really small. I specified my input shape as a hyperparameter just to be sure.

Then I tried both types of pooling layers (average and maximum) of size 2x2 and I decided to stick with

MaxPooling2D [3]. Before I send the convoluted to the actual NN layers, I decided to flatten it into a vector for improved runtime.

Once more, I tried understanding and playing with my new hyperparameters; the only real improvements were made when I changed the distribution for the 'kernel_initializer' to 'he_uniform'.



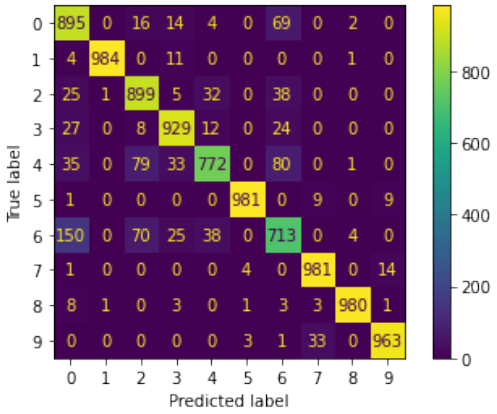
3.4.3 Getting to my final model

After studying more what I was actually doing [4] while this model was compiling, I decided that the best approach was to include multiple Conv2D+MaxPooling2D combos with increasing number of filters, so my network could identify more features in my images. Moreover, I changed the padding parameter to 'same', so that my reduction of dimensions was controlled only by the pooling alone.

After many different attempts with what I knew so far, my perfected network looks like this [see figure]. I have reached my goal, while I do realise there is still a lot of room for improvement beyond my curriculum.

4. Conclusions and further remarks

No matter the algorithm used, there are some issues that I could not get over. Any confusion matrix will show you that T-shirts get easily interpreted as shirts and coats as pullovers. This was expected, as I found out on Zalando's GitHub page[1] for the dataset that even crowd-sourced evaluation of actual humans got an accuracy of 83%. Given the low accuracy of my images, some confusion is thus expected even for the human eye.



Moreover, I suppose that further improvements could be made by ensembling techniques for CNNs, or by finding a way to generate more data from the already existing one. I thought of using PCA as well, since many of the pixels are just background, but neuronal networks already do a very well job in reducing the dimensionality of the matrices.

All in all, my final opinion is that modern CNNs surpass, at least in this case, traditional ML algorithms, and provide more insight into how to construct an optimal model for my data, while also being more fun to play with.

References

[1] The official GitHub repository for Fashion-MNIST of Zalando Research: https://github.com/zalandoresearch/fashion-mnist

[2] The official Keras documentation for Conv2D: https://keras.io/api/layers/convolution_layers/convolution2d/

[3] The official Keras documentation for MaxPooling2D: https://keras.io/api/layers/pooling_layers/max_pooling2d/

[4] *Keras Conv2D and Convolutional Layers*, by Adrian Rosebrock, 31 Dec 2018 https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/