Image Classification for Fashion-MNIST

Aim

To investigate the performance of different models (with a focus on neural networks) to classify the images in the Fashion-MNIST dataset. Please see `requirements.txt` to run the code.

Visualising the Data

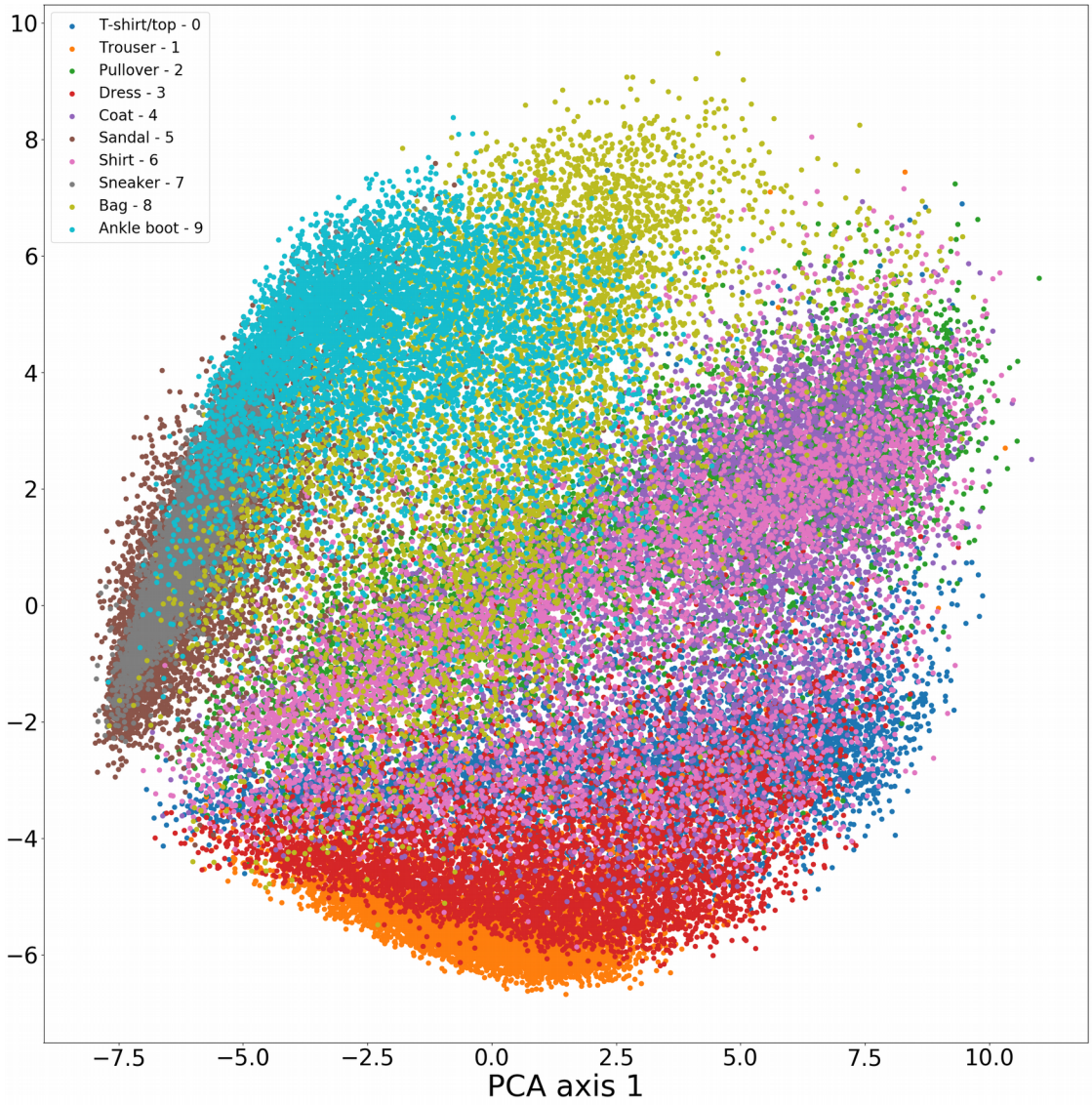

Figure 1: PCA of training data after preprocessing

In order to visualise the data, we can use the dimensionality reduction technique, principal component analysis, to present the data on a scatter diagram. This is shown in Figure 1 which was generated using `dataVisualisation.py`.

We can see that the most distinct class is trousers. This makes sense because it is the only item of legwear so it will not have many common features with the other classes. From the top left of Figure 1, we can see a lot of overlap between ankle boots, sneakers and sandals. This is most likely

because they are all items of footwear and will have similar features as a result. The same can be said for t-shirts, shirts, and coats which are all upper-body items of clothing. These two sets of classes are therefore likely to be harder to classify.

Preprocessing the data

The gray-scale pixel values are in the range [0, 255]. This is a relatively small range so normalisation is not strictly necessary.[1] However, the pixel values will divided by 255 so that they will all be in the range [0, 1]. This will help with numerical stability when exponentiating with large numbers.

The next step will be zero-centering the data. This can be done using mean subtraction. This is done by first calculating the mean image which consists of the mean pixel value for each pixel position for every image in the training data. This mean image is saved and is subtracted from each training and test image.

Preprocessing the data makes classification less sensitive to changing the parameters of the model, as shown by Figure 2.

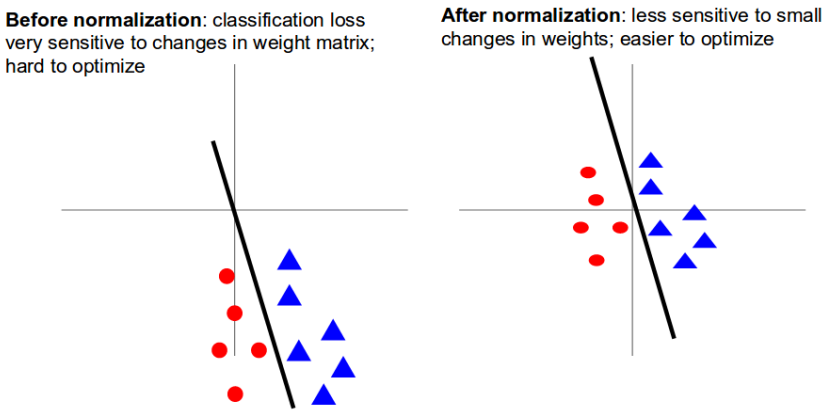

Figure 2: Classifying data before and after zero-centering[2]

The code for the data preprocessing can be found in `dataPreprocessing.py`.

Experimenting with different Models

Model 1: Linear Model

We will start by first training a linear model adapted from the Imperial College Data Science workshop on neural networks[3]. For input image **x**. The linear model **f** will attempt to classify the image using:

$$f(x) = xW + b$$

Where **W** is the matrix of the weights and **b** is the bias vector.

Initially, the weights will be initialised to small random values and biases set to zero. The model's performance will be evaluated using the cross entropy loss function and will be trained using backpropagation and batch gradient descent.

Having minimised the loss, we can now test the model on the test data.

The results of the testing are as follows:

Overall Accuracy: 83.36%

Class	Linear Model
T-shirt/top - 0	82.10%
Trouser - 1	94.80%
Pullover - 2	72.00%
Dress - 3	86.60%
Coat - 4	75.60%
Sandal - 5	89.90%
Shirt - 6	55.50%
Sneaker - 7	90.70%
Bag - 8	93.10%
Ankle boot - 9	93.30%

Table 1: Performance of the linear model by class

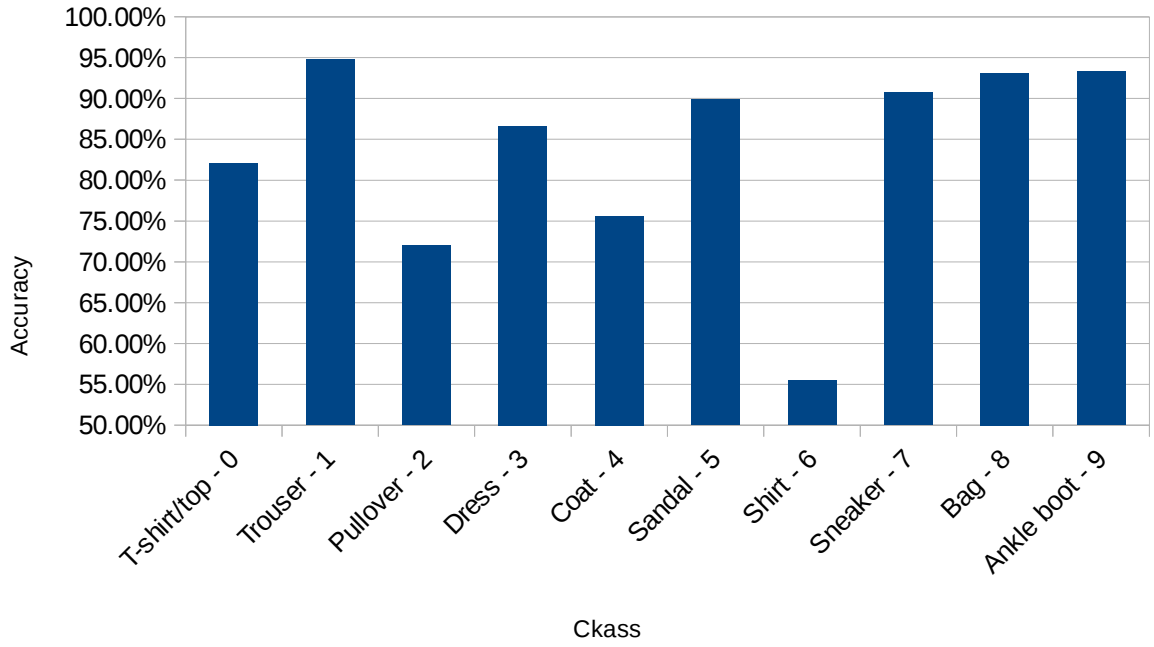

Figure 3: Performance of the linear model class by class (graph)

The linear model was most successful in classifying the trouser class. This falls in line with our previous observations when visualising the training dataset. The model is particularly struggling to classify T-shirts, pullovers, coats and shirts. This is to be expected given their overlap in Figure 1.

The training and testing of this model can be found in `linearModel.py`.

Model 2: Single Hidden Layer Feedforward Neural Network

For input image **x**. The model **f** will attempt to classify the image using:

$$f(x) = \sigma(x \times W_1 + b_1) \times W_2 + b_2$$

Where $\mathbf{W_1}$,$\mathbf{W_2}$ are the matrices of the weights and $\mathbf{b_1}$, $\mathbf{b_2}$ are the bias vectors. $\boldsymbol{\sigma}$ is the non-linear activation function ReLU. This is an artificial neural network with a single hidden layer. The idea is that the introduction of the non-linear function will assist in classifying the data.

This models performance will be evaluated using the cross entropy loss function. As this model is more computationally expensive than the linear model, we will use mini batch gradient descent for a faster convergence when minimising the loss. The weights will be updated using the Adam optimizer. The training process can be seen in the function `train_NN` in `singleHiddenLayerFeedforwardTraining.py`.

There are many hyperparameters we can choose including: initial learning_rate, hidden layer neurons, training epochs and batch size. For this demonstration we will tune the hyperparameters hidden layer neurons and the initial learning rate.

We will tune these two hyperparameters using a random search method as opposed to a grid search method as argued by James Bergstra and Yoshua Bengio.[4]

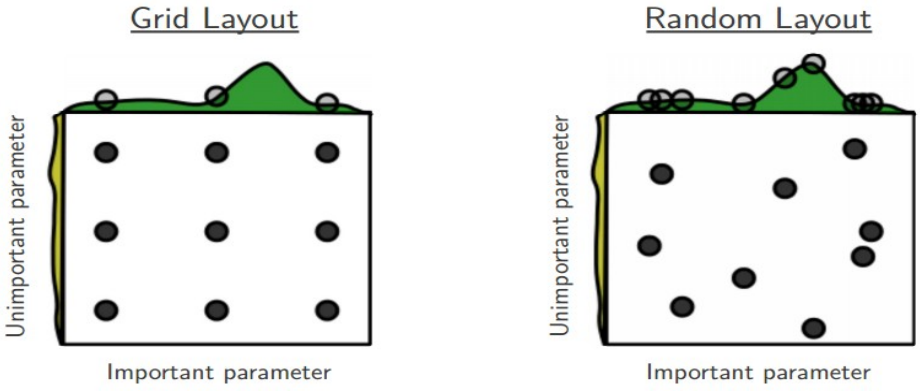

Figure 4: Grid search compared to random search for hyperparameters[5]

```
Hidden layer units: 258  Initial learning rate: 0.0036692852925073686 Loss: 1.3284739255905151
Hidden layer units: 759  Initial learning rate: 0.021790722200904347 Loss: 4.568992614746094
Hidden layer units: 351  Initial learning rate: 0.3098958845237359 Loss: 50.5041389465332
Hidden layer units: 699  Initial learning rate: 0.0015510776346285456 Loss: 0.9341930747032166
Hidden layer units: 485  Initial learning rate: 0.002251422317770052 Loss: 1.2014330625534058
Hidden layer units: 339  Initial learning rate: 0.5922142338379853 Loss: 108.67781829833984
Hidden layer units: 172  Initial learning rate: 0.060372524135748507 Loss: 2.1592905521392822
Hidden layer units: 148  Initial learning rate: 0.04964143033010138 Loss: 2.042207717895508
Hidden layer units: 668  Initial learning rate: 0.005265206145498711 Loss: 1.6690821647644043
Hidden layer units: 581  Initial learning rate: 0.007461375963931765 Loss: 1.9090712070465088
Hidden layer units: 655  Initial learning rate: 0.11222572107558558 Loss: 33.59316635131836
```

Figure 5: Sample of entries in the hyperparameter search log

The test data should not be used to test the hyperparameter configurations. This could lead to overfitting. Hence, we will use 1000 elements from the training data to test the hyperparameters. This is called the validation set.[6] Over time we can narrow the search on the range of hyperparameters for a more fine tuning. The implementation of this hyperparameter search can be found in the function `hyperparameter_search` in `singleHiddenLayerFeedforwardTraining.py`.

The results of the testing are as follows:

Overall Accuracy: 89.28%

Class	Accuracy
T-shirt/top - 0	83.00%
Trouser - 1	98.20%
Pullover - 2	83.20%
Dress - 3	89.90%
Coat - 4	83.50%
Sandal - 5	96.90%
Shirt - 6	71.70%
Sneaker - 7	91.10%
Bag - 8	97.10%
Ankle boot - 9	98.20%

Table 2: Accuracy of single hidden layer neural network by class

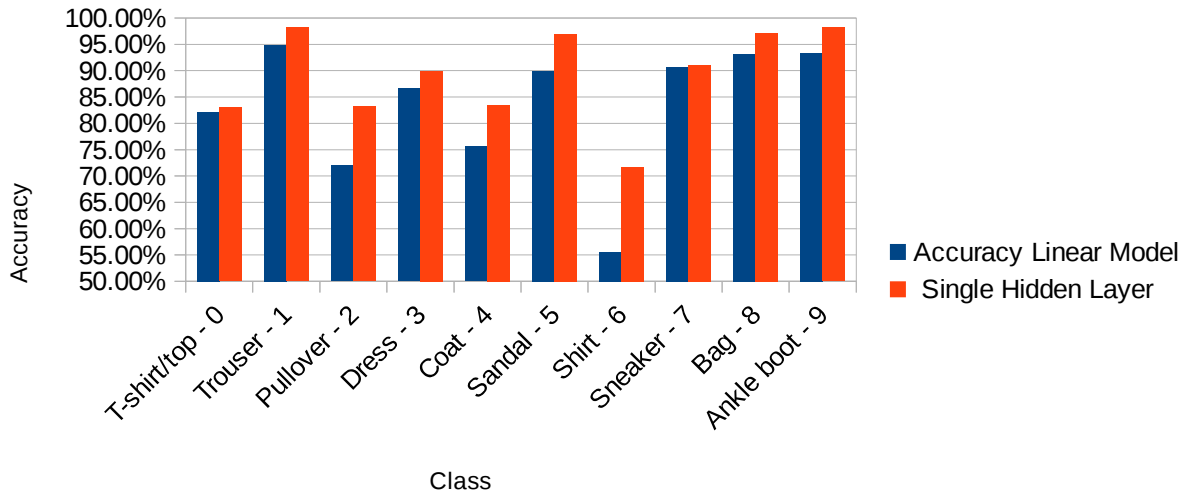

Figure 6: Comparison of accuracy per class between two models by class

The code used to test this model can be found in `singleHiddenLayerFeedforwardTesting.py`.

Model 3: Convolutional Neural Network

The next model we will try is a convolutional neural network. Although single hidden layer neural network can be used for image classification, CNNs are typically more common because they scale better for a greater input size.[7]

```
ConvNet(
  (conv1): Conv2d (1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d (8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fcl1): Linear(in_features=784, out_features=500)
  (fcl2): Linear(in_features=500, out_features=10)
)
```

Figure 7: PyTorch ConvNet class in `CNNClasses.py`

After tuning the hyperparameters, we end up with a CNN with the structure shown in Figure 7. It has two convolutional layers (both followed by max pooling layers) and two fully connected layers. It should be noted that a CNN has many more hyperparameters- such as stride, zero-padding and kernel size)- and can have many architecture configurations. This made it hard to perform the

hyperparameter search. Additionally, larger (possibly better) architectures were too computationally expensive to train.

The results of testing this model are as follows:

Overall Accuracy 89.34%

Class	Accuracy
T-shirt/top - 0	81.90%
Trouser - 1	96.90%
Pullover - 2	83.20%
Dress - 3	89.90%
Coat - 4	84.00%
Sandal - 5	97.20%
Shirt - 6	69.80%
Sneaker - 7	96.70%
Bag - 8	97.30%
Ankle boot - 9	96.50%

Table 3: Accuracy of CNN by class

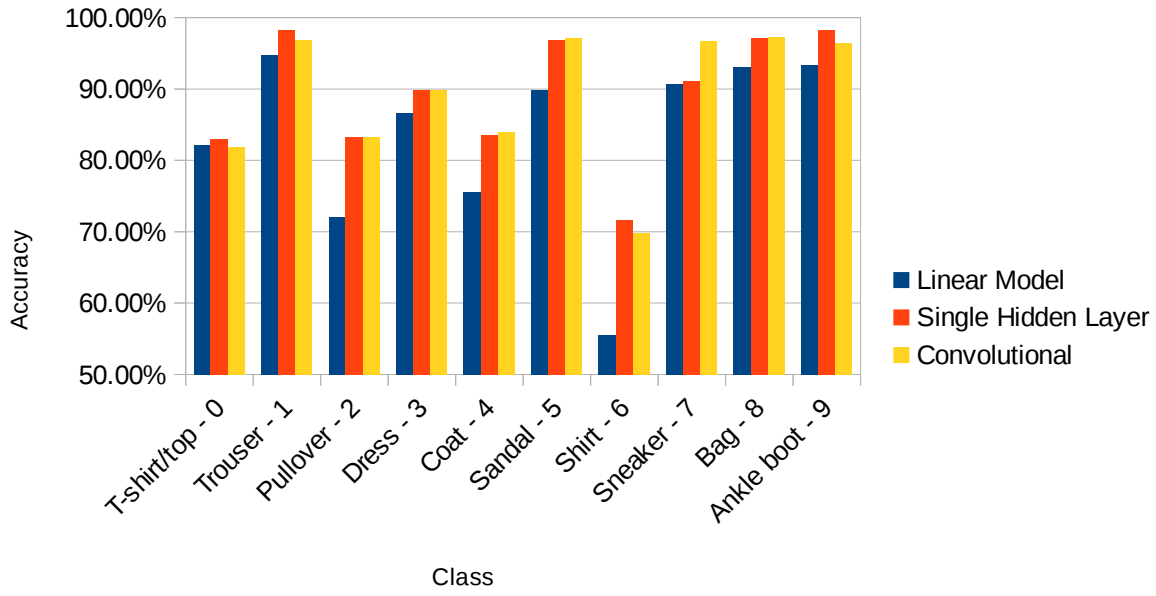

Figure 8: Comparison of the three models

The training can be found in `convolutionalNeuralNetworkTraining.py` and the testing can be found in `convolutionalNeuralNetworkTesting.py`.

The overall accuracy of the convolutional neural network is similar to the single hidden layer neural network. The performance across each class is also similar as seen in Figure 8, with a noticeable increase in performance when classifying sneakers.

Reflections/Model improvements

Given more time/resources, here is how we could potentially improve the performance of the convolutional neural network:

- Use more filters in the convolutional layers to recognise more features
- Use Xavier initialisation for the weights in the network

- Add batch normalization layers[8]
- Using an existing convolutional neural network that classifies similar data for transfer learning[9]
- Use a regularisation method such as dropout to prevent overfitting the training data[10]
- Perform a more in depth random search with other hyperparameters

References

[1], [2], [5], [6], [8], [9], [10]. Andrej Kaparthy. *CS231n: Convolutional Neural Networks for Visual Recognition*. Available from: cs231n.stanford.edu. [Accessed 28th December 2017]
[3]. Imperial College Data Science Society. *Neural Networks.* Available from: https://github.com/Imperial-College-Data-Science-Society/Neural-Networks. [Accessed 25th December 2017]
[4]. Bergstra J, Bengio Y. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research 13.* 2012. p302.
[7]. Wikipedia contributors. *Convolutional neural network.* Available from: https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=817260146 . [Accessed 28th December 2017]