Zalando Clothing Classificaton

January 1, 2023

Authors:

Kristian Graven Hansen (krgh@itu.dk)

Lukas Sarka (lsar@itu.dk)

Mikkel Geisler (mgei@itu.dk)

1 Introduction

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets

2 Data and Preprocessing

2.1 Dataset

The Fashion-MNIST is a dataset of Zalando article images, consisting of a training set of 60,000 samples and 10,000 test samples. The dataset used here is a small portion of the original dataset, 10,000 training and 5,000 test samples. Each sample is a 28x28 pixels gray-scale image with a label indicating the type of clothing item associated with each.

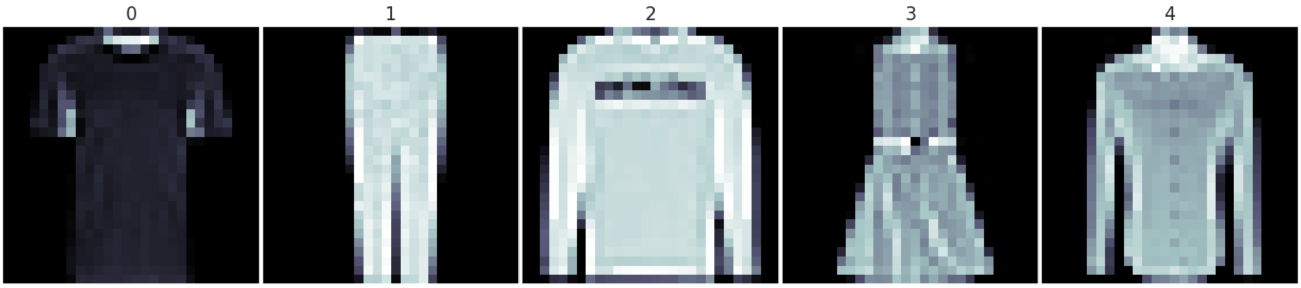

Figure 1: One sample from each class

2.2 Naming Conventions

The exploration of samples within each class gave rise to more appropriate names. Class 0 became T-shirt etc. Below is the naming conventions, which will be used throughout the report for better readability.

Label	**0**	**1**	**2**	**3**	**4**
Type of clothing	T-shirt/Top	Trousers	Pullover	Dress	Shirt

Table 1: Mapping from class-labels to clothing type

2.3 Data Cleaning

The dataset provided was already in a cleaned state, which was verified by checking for missing values, and checking that the pixel values were no greater than 255 and no smaller than 0.

2.4 Preprocessing

The pixels values were in the range $[0, 255]$. This range was normalized to $[0, 1]$ by dividing each pixel by 255. This was done to improve training time as working with small number is slightly faster computationally, and because neural networks prefers to work with small numbers.

2.5 Class Distribution

Whether or not a machine learning model can learn to predict classes well depends to a high degree on how those classes are distributed within our training and training dataset. Both the datasets are extremely balanced, with the test being fully balanced (1000 of each class). This is illustrated on the plots below

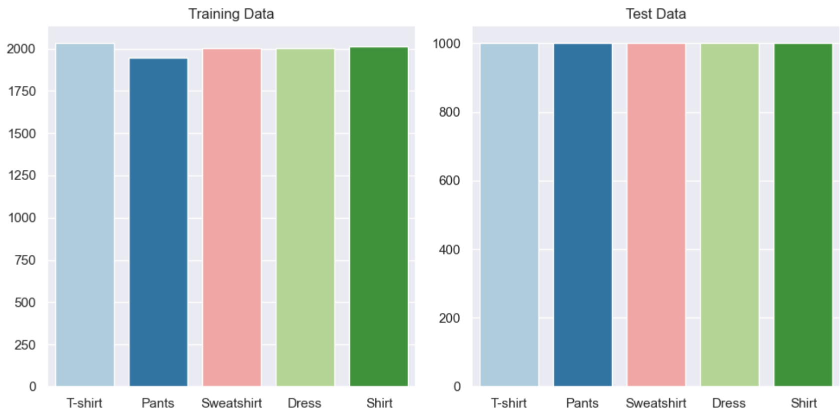

Figure 2: Distribution of clothing items in our training and test dataset

3 Exploratory Data Analysis

One simple yet useful way to explore a dataset is to visualize the feature distribution for each class. In our situation this is simply not possible and not very insightful as we have too many features. To be precise $728 = 28 \times 28$ features/pixels. This doesn't mean the dataset can't be visualized, which we will discuss in the next section.

3.1 Principal Component Analysis

As mentioned before due to our large dataset it is hard to visualize feature distributions, this is where principal component analysis or PCA can be used instead. Principal Component Analysis (PCA) is a dimensionality reduction technique used to reduce the number of features in a dataset, while still preserving the most important information. It does this by transforming variables into a new set of variables, called principal components, which are uncorrelated from each other and explain the maximum amount of variance in the data. Below we explore the relationships between the first 3 principal components. These 3 principal components represent the direction of most variance in the original data

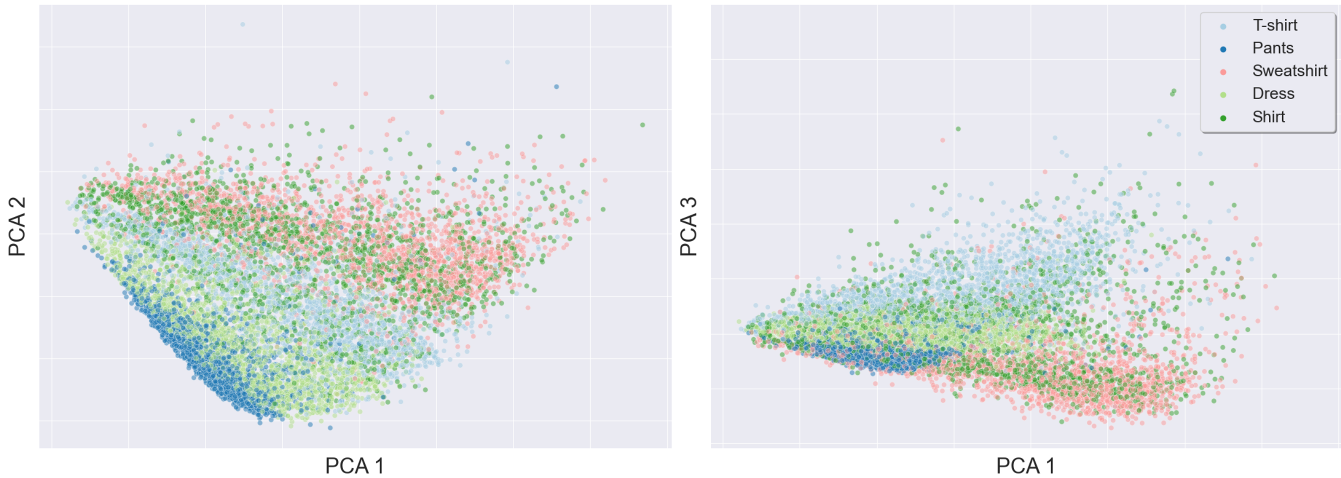

Figure 3: Visualizing some relationships between first 3 PCA's

4 Decision Tree

In this section, we will discuss how the `DecisionTreeClassifier()` model is implemented and how the hyperparameters are chosen, as well as the results of using this model.

A decision tree is a supervised machine learning model that can be used for classification or regression tasks. It works by building a tree-like structure, where at each internal node, the algorithm evaluates all the available features and selects the split that maximizes the purity of the resulting splits. This process is repeated until a stopping condition is reached, such as a maximum depth or no further improvement in purity. The resulting leaf nodes are also known as decision regions, and the prediction for a sample is made by traversing the tree to find the correct decision region and then predicting the most common class in that region. Decision trees are popular because they are easy to understand and interpret due to their tree-like structure, and they can handle different types of data and decision boundaries. However, during training, the algorithm only considers the best split for each internal node without considering the potential negative effects on future splits, which is known as a greedy approach.

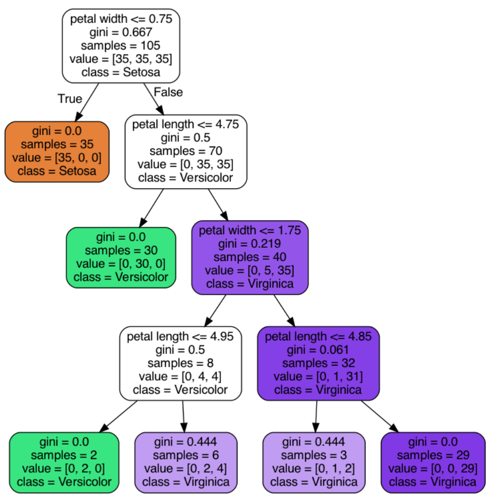

Figure 4: Simple Decision Tree

4.1 Implementation

As this project is focused on classification rather than regression a classification tree was implemented. It was done in Python using two classes:

1. `Node()`
2. `DecisionTreeClassifier()`

4.1.1 Node()

The `Node()` class has the most responsibility of the two. It holds the data and passes it down the tree constantly splitting itself into more nodes with the `_split()` method. This is done by finding the best split with the `_get_best_split()` method based on the impurity measure specified. The best split refers to the feature and cutoff value that leads to the highest gain in purity in the two resulting nodes. After finding the best split, it can then create two new nodes, `self.left` and `self.right` if all split criteria are fulfilled. The two criteria implemented are `max_depth` and `min_samples_split` and are discussed further in the `DecisionTreeClassifier()` implementation.

4.1.2 DecisionTreeClassifier()

The `DecisionTreeClassifier()` is the classifier class. which has the `fit()` and `predict()` methods and some 'less' important methods `get_depth()` and `get_n_leaves()`, that can be called after the model has been fitted. The `fit(X, y)` method takes the following two parameters, a feature matrix `(X)` and class labels `(y)` This data fed into the first node in the tree (the root). The purity of the node is then found and if a split can improve the purity and all split criteria are fulfilled the node calls `_split()` on itself. This happens recursively and is how the tree-structure is created. The `predict(X)` method takes an array of samples or a single sample, and predicts the corresponding label. It works by traversing down the tree until a leaf node is reached, and then predicting the most frequent class in that node. The constructor take the following 4 parameters.

- `max_depth` - (An integer - the maximum depth to which the tree can grow)
- `min_samples_split` - (An integer - minimum number of samples in a node before for it to split)
- `criterion` - (A string - impurity measure (gini or entropy) to use when splitting)
- `random_state` - (An integer - used to set the random seed for reproducible results)

4.2 Correctness

To validate the correctness of our implementation a thorough comparison with the sklearn version was done. First a comparison of 100 sklearn models and 100 of our models accuracy score was done using the `load_digits()` dataset from `sklearn.datasets` This can be seen in Figure 5.

Each decision tree used in Figure 5 is fitted with the default parameters of both the sklearn and our model. This means that the `max_depth` is not restricted and the `min_samples_split` is set to 2.

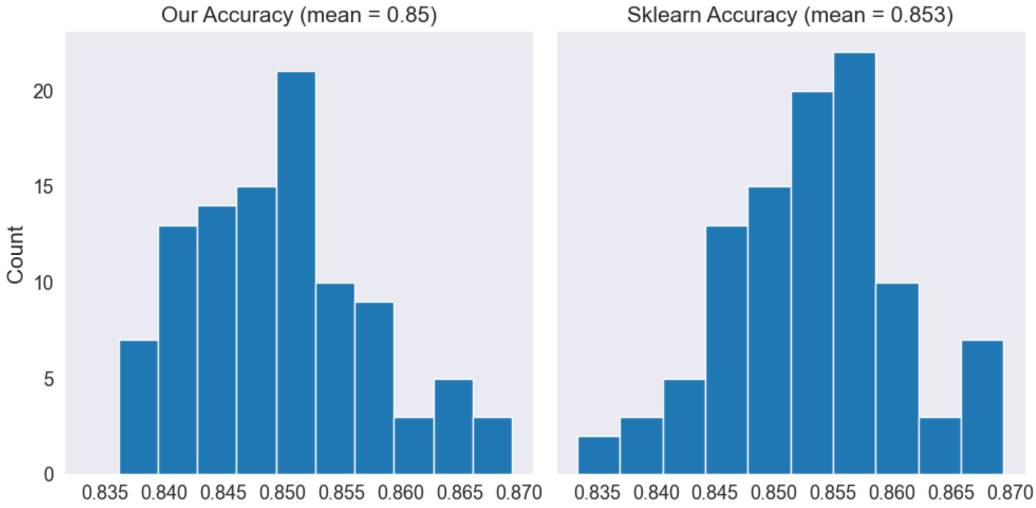

Figure 5: Accuracy distribution from 100 models

A decision tree has randomness involved in the training process. This is because if two splits yield the same gain one of them will be chosen at random. This means the models might perform slightly different in a single test but very similar on average.

The resulting tree depths and number of leaves were also compared and showed that the decision trees in both implementations grow to the same depth and with the same number of leaf nodes.

Lastly a comparison of **some** of the actual splits (best feature and cutoff) were done to assert correctness and as a sanity check, and the two implementations also agreed on this.

4.3 Hyperparameters

A decision tree is very prone to overfitting to the training data. This is especially the case if the max depth or other early stopping to the growth is not controlled. This can lead to a very high training accuracy but might struggle with unseen data. Another way to control this is using post-pruning where the tree is first grown fully, and then afterwards its size is reduced. This was not implemented in our model so what was tried instead was to find the optimal `max_depth`, to reduce overfitting. Along with finding an optimal `max_depth`, a good combination of `max_depth`, `min_samples_split` and `criterion` (gini or entropy) was found. This was done `GridSearchCV()` from sklearn, where a range of values for each of the mentioned parameters was searched for. `GridSearchCV()` uses cross-validation, in this case 5-fold, on the training data, and tests all combinations specified in the following parameter grid:

- `max_depth`: [None, 5, 7, 9, 11, 12]
- `min_samples_split`: [2, 4, 8, 10]
- `criterion`: [gini, entropy]

This resulted in the following best combination of hyperparameters `max_depth=9`, `min_samples_split=4`, `criterion=gini`.

4.4 Results

Evaluation Metric	Accuracy Score
Training Accuracy	89.83%
Test Accuracy	79.32%

Class	Precision	Recall	F1-Score
T-shirt/Top	0.76	0.75	0.75
Trousers	0.97	0.92	0.95
Pullover	0.80	0.83	0.81
Dress	0.82	0.88	0.85
Shirt	0.61	0.60	0.60

Table 2: Decision Tree Performance Evaluation

The results reported in table 2 are results of our implementation of the `DecisionTreeClassifier()` with the parameters specified in section **4.3**. To reproduce these exact results it should additionally be initialized with `random_state=42`, to eliminate the randomness.

4.4.1 Discussing Results

5 Feed-Forward Neural Network

In this section, we will discuss how the NeuralNetworkClassifier() model is implemented and how the hyperparameters are chosen, as well as the results of using this model.

A feedforward neural network is a type of artificial neural network that consists of an input layer, one or more hidden layers, and an output layer. Each layer consists of a set of neurons, which are connected to the neurons in the next layer via weights. The input layer receives input data and passes it through the network to the output layer, where the output is produced.

The hidden layers process the input data and transform it into a form that can be used by the output layer to produce the desired output. The weights of the connections between neurons are adjusted during the training process to minimize the difference between the predicted output and the actual output, a process known as backpropagation.

Feedforward neural networks can be used for a wide range of tasks, including image and speech recognition, natural language processing, and predicting outcomes in complex systems. They are a popular choice for machine learning tasks due to their ability to learn and generalize from data

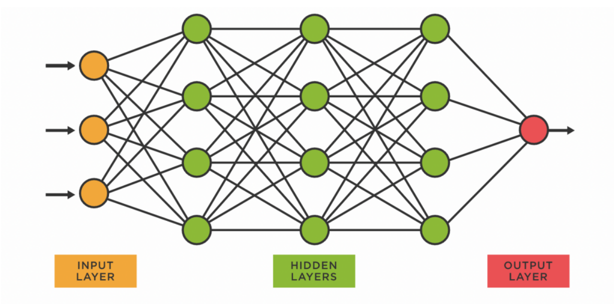

Figure 6: Simple Neural Network Architecture

5.1 Implementation

The implementation of the Feed-Forward Neural Network (FNN) was in Python using two classes.

1. `DenseLayer()`
2. `NeuralNetworkClassifier()`

5.1.1 DenseLayer()

The `DenseLayer()` represents the hidden-layers and the output layer that make up the FNN. It should be initialized with a `layer_size` and an `activation`. The `layer_size` represents the number of neurons in the layer, for example if you have a multiclass classification with 3 classes the output layer should have `layer_size=3`. The `activation` parameter is the activation function to be applied to the output of the layer. The current implemented activation-functions are Sigmoid, ReLU and Softmax, where Softmax can only be applied to the output layer.

The `DenseLayer()` has two primary methods `forward()` and `backward()`. The `forward(x, weights, bias)` method performs a forward pass through the layer, given the input data (`x`), weights (`weights`), and bias (`bias`). It calculates the dot product of the inputs and weights, adds the bias, and applies the activation function to the result to produce the output of the layer. It returns pre-activated output of the layer, and the activation of the layer. The `backward(dA_curr, W_curr, Z_curr, A_prev)` method performs the backward pass through the layer, given the gradient of the cost with respect to the current layer's activations (`dA_curr`), the current layer's weights (`W_curr`), the current layer's pre-activation values (`Z_curr`), and the previous layer's activations (`A_prev`). It calculates the gradients of the cost with respect to the weights (`dW`), bias (`db`), and previous layer's activations (`dA`) using the chain rule. The return values of both the `forward()` and the `backward()` method are needed for backpropagation which happens in the `NeuralNetworkClassifier()`.

5.1.2 NeuralNetworkClassifier()

The `NeuralNetworkClassifier()` is the classifier class. It's primary methods are `fit()` and `predict()`. Additionally, it includes a lot of helper methods that all have different responsibilities. These include initializing and updating weights and biases based on the layer sizes and input data dimensions specified. The constructor take the following 4 parameters:

- `layers` - (A list of DenseLayer objects - the layers in the network)
- `learning_rate` - (A float - how fast the weights update after each mini-batch)
- `epochs` - (An integer - the number of epochs to train the neural network for)
- `random_state` - (A integer - set the random seed for reproducible results)

The `fit(x, y, batch_size, validation_size)` method is used to train the model, and takes 4 parameters: The training data (`x`) and labels (`y`), the `batch_size` (an integer indicating the number of samples to use in each mini-batch during training). Lastly `validation_size` (a float indicating the proportion of the training data to use for validation). To keep it short the `fit()` method does the following: For each mini-batch, the model makes predictions using the `_forward()` method, computes the loss using the `delta_cross_entropy()` function, then performs backpropagation with the `_backward()` method and finally updates weights and biases using the gradients calculated with `_backward()` with the `_update_trainable_params()` method. This process is performed as many times as specified in the constructor of the class.

5.2 Correctness

For the scope of this project the correctness of our implementation was done primarily visually. The training accuracy and training loss was compared to a similar model from the Tensorflow library. The models used here has two ReLU activated hidden layers with 128 and 64 neurons respectively, a softmax activated output layer with 5 neurons (one for each clothing item), a learning rate of 0.01, and lastly for reproduction of these exact graph `random_state=42`, and `tf.random.set_seed(42)` was used.

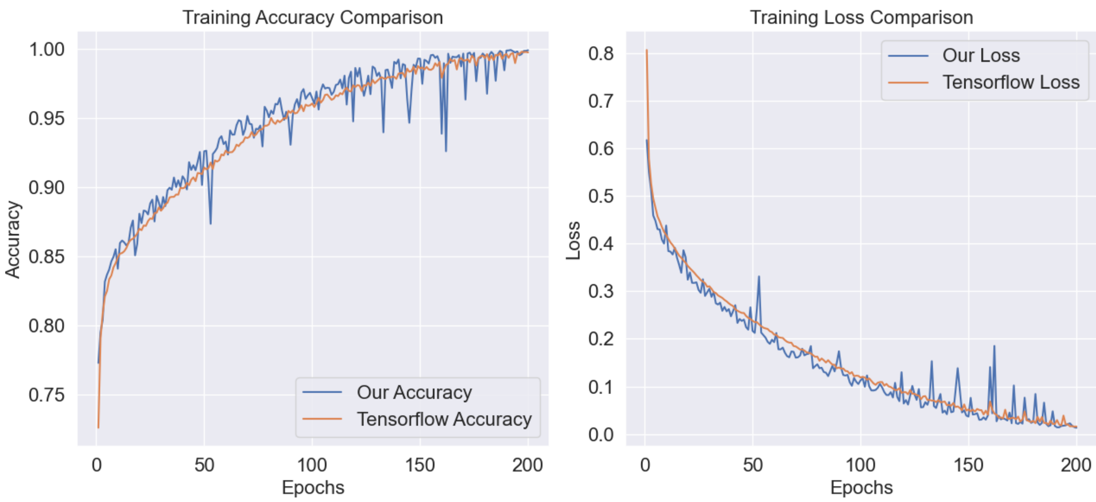

Figure 7: Comparing Accuracy and Loss with Tensorflow

As seen in Figure 7 the trend in accuracy and loss is very similar same however Tensorflow's model is obviously far better optimized so the curves are much smoother. Based on Figure 7 our model was considered correctly implemented.

5.3 Hyperparameters

Neural Networks are excellent at learning from data. This however can often lead to overfitting, when the number of epochs or learning rate is too high. We restricted ourselves to just look at the number of epochs while keeping all the other fixed, meaning, using the same hidden layers as in section **5.2** and the following parameters `learning_rate=0.01`, `batch_size=32`, `random_state=42`. This meant a number of epochs that gave a high validation accuracy while still keeping the validation loss low was optimal. Figure 8 shows the models performance during training and was used to choose 70 as the number of epochs. These parameters was then used to train an instance of the `NeuralNetworkClassifier()` and the results are reported in the following section.

Figure 8: Performance during training

5.4 Results

Evaluation Metric	Accuracy Score
Training Accuracy	94.29%
Test Accuracy	85.16%

Class	Precision	Recall	F1-Score
T-shirt/Top	0.81	0.80	0.81
Trousers	0.97	0.97	0.97
Pullover	0.86	0.87	0.86
Dress	0.90	0.89	0.90
Shirt	0.72	0.73	0.72

Table 3: FNN Performance Evaluation

5.4.1 Discussing Results