IT UNIVERSITY OF COPENHAGEN

MACHINE LEARNING

BSMALEA1KU

Fashion MNIST Classification

Authors

Viggo Y. UNMACK GASCOU	VIGA@ITU.DK
Frida NØHR LAUSTSEN	FRIL@ITU.DK
Marie HAAHR PETERSEN	MHPE@ITU.DK

Supervisors
Therese GRAVERSEN
Payam ZAHADAT

January 6, 2023

1 Introduction

Image processing is a broad field in the world of deep learning. It is applied in various ways, in everything from self-driving cars to filters on your phone. An image is generally represented as a matrix, where each entry is a pixel-value indicating the intensity of colour in that spot. There are numerous ways to classify an image, and amongst those, the most popular ones are convolutional neural networks.

The goal of this project is to implement different machine learning models that are able to classify grey-scale images from the popular Fashion MNIST data set from Zalando (Xiao et al., 2017). Through this report we will present the implementations and evaluate our models and discuss the sources of error that occured.

2 Data and Preprocessing

2.1 Data Description

The data set we are using for this project is a subset of the fashion MNIST data set. It contains $15{,}000$ 28×28 grey-scale images, that are represented as flattened arrays and is pre-split into two data sets – a training data set with $10{,}000$ images and a test data set with $5{,}000$ images and their associated labels.

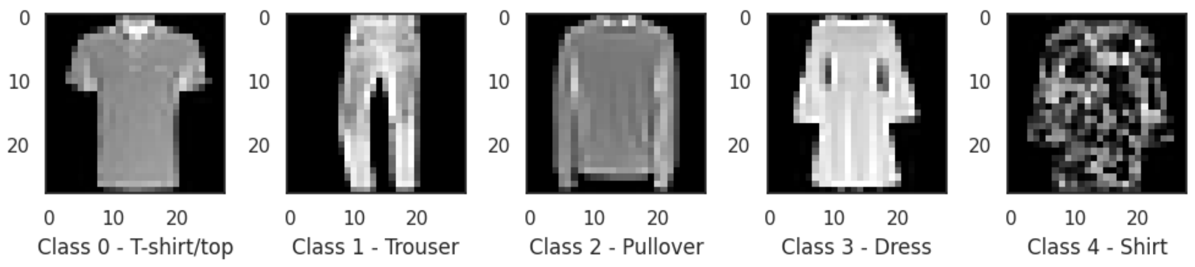

FIGURE 2.1: *A random sample from the training set of 5 images one from each class*

Each image has a total of 784 pixels and each pixel has an associated pixel-value that ranges from 0 to 255, which represents the brightness of the pixel. Each image has been assigned one of 5 labels, to encode which article of clothing the image is of (Table 2.1).

Type of clothing	T-shirt/top	Trouser	Pullover	Dress	Shirt
Label	0	1	2	3	4

TABLE 2.1: *Label encoding*

2.2 Data Cleaning

We carried out some different integrity checks of the data. Such as checking for missing values, ensuring values are integers only and in the correct range i.e. pixel-values between 0 and 255 and labels between 0 and 4. We didn't find any inconsistencies, and therefore no data cleaning was needed.

3 Exploratory Data Analysis

3.1 Class Distribution

Both the train and test split have roughly even class distributions. Each class making up approximately 20% of the data set. The training split is slightly less even than the test split, with Trouser making up marginally less and T-shirt/top making up marginally more, compared to the other classes. Having all classes represented somewhat equally benefits us when it comes to building our classifiers. We are confident that we have enough samples of each class for the model to learn from (Figure 3.1).

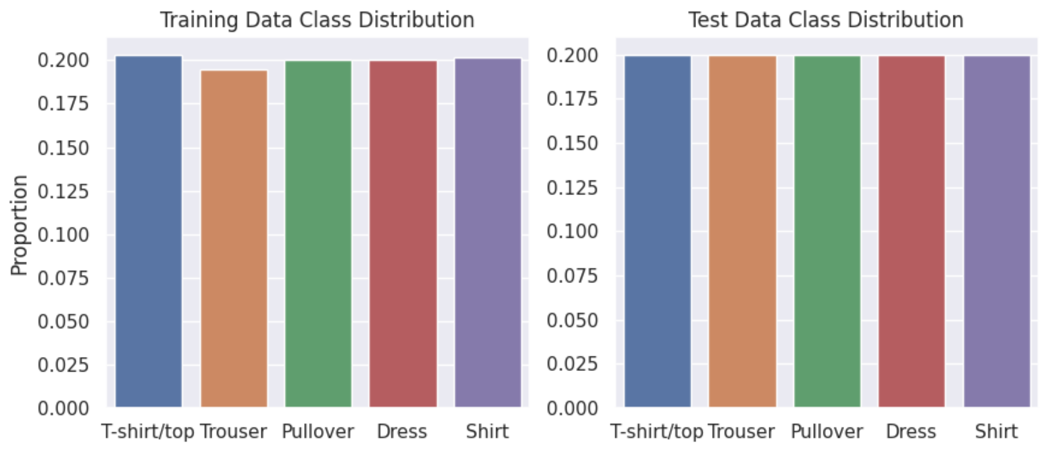

FIGURE 3.1: *Class distribution of train and test split*

3.2 Principle Component Analysis

Principle Component Analysis (PCA) is an unsupervised machine learning method, and is a form of feature extraction which compresses the data while maintaining as much of the information as possible. We performed PCA and transformed the data further by scaling the transformed variables with the square root of the eigenvalues. This is a preprocessing step called sphering and it transforms the data into a set of new uncorrelated variables with variance 1. With a data set like Fashion MNIST with 784 features, dimensionality reduction can be very useful for transforming the data from a high-dimensional space to a low- dimensional space. Dimensionality reduction can improve computational performance, improve model performance by avoiding overfitting, and lastly is great for visualisation.

We performed sphering on the unscaled version of the training set. We were able to use the unscaled version of the data set, since all the features are measured in the same unit. From the kernel density function in the pair plot of the first five principal components in the train split, it is clear to see that there is a lot of class overlap present in the PCA data set (Appendix A). We would expect that the other principal components will portray the same tendency of class overlap (James et al., 2021). This poses a challenge to the classifier, since it becomes difficult to distinguish the clothing items from each other. The first principle component together with the second principle component present, what looks like, the clearest class separability.

Looking at the plot of the cumulative explained variance ratio (CVR) we see that using only a fraction of the extracted features we are still able to preserve a significant amount of the information in the original data set, in terms of explained variance (Figure 3.2). In fact, we are able to preserve 90% of the explained variance with the first 62 principal components. This is a reduction of the feature space by 92%.

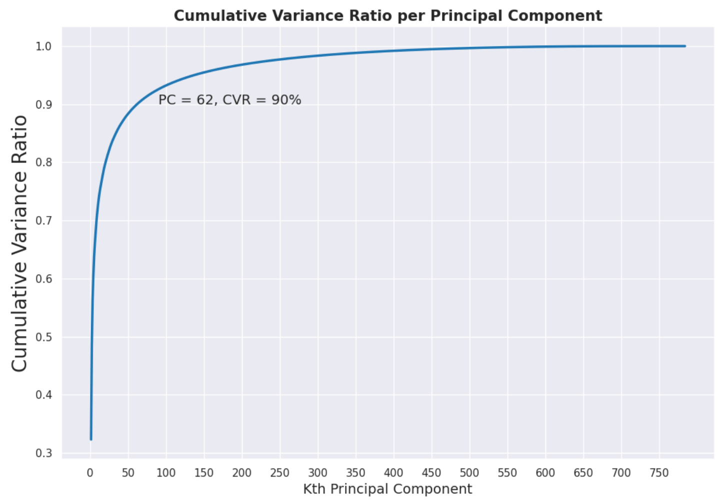

FIGURE 3.2: *Plot of the cumulative variance ratio for the principal components*

4 Implementations

4.1 Decision Tree Classifier

4.1.1 Implementation

A Decision Tree Classifier is an easy interpretable, discriminative, white box model that makes very few assumptions about the training data (Géron, 2019). A Decision Tree has a tree like structure, which makes it convenient to map out the decision making, and the possible outcomes. You start at the root of the tree, moving along the branches of the tree based on the value of the data point and end up in a leaf, where a prediction is made.

We built the decision tree using a `Node()` class data structure. A node can be either a root, internal or leaf node. The root node and internal nodes hold information about the feature (`self.feature`) and value (`self.threshold`) for the best split for the given subset in that node. Furthermore it contains information about the left and right child. For the leaf nodes a class is assigned to the `self.majority_class` variable, which contains the class that the specific leaf node predicts to. Information about class probabilities is saved in the variable `self.class_probs`.

The actual fitting happens when we build the tree. We start at the root node and compute the information gain (Equation 4.1), using an impurity function for each possible split of each feature. The choice of impurity criterion is a hyper parameter that is chosen before building the structure. This can be either Gini Impurity or Entropy.

$$\Phi(t) - (p_R\Phi(t_R) + p_L\Phi(t_L)) \tag{4.1}$$

We save the indices for the split with the highest information gain and split the data into two subsets using those. This happens in the function `_grow()`, where it is recursively called twice for the left and right child nodes, passing the new subsets of data along with the current depth. This happens repeatedly until one of three criteria are met. Either when there are fewer samples in the leaf node than specified in `self.min_samples_in_leaf`, when it hits the specified maximum depth in `self.max_depth`, or if the leaf node contains only one class.

In the function `_traverse()` we can predict either the class or the probability, when the tree has been built. A data sample is classified by starting at the root node. We move down the tree either left or right according to the threshold of the current node, until we have reached a leaf node. The sample is classified to the given leaf node's `self.majority_class`.

4.1.2 Evaluating correctness

In order to evaluate the correctness of the our `DecisionTreeClassifier` we compared it to the scikit- learn decision tree classifier from `sklearn.tree`. Both classifiers were initialised with the same hyper parameters, and both fitted on the same training and validation split of the Fashion MNIST training data. The resulting accuracies were the same. Furthermore we compared the predictions, and they turned out completely identical. Overall, our implementation performs as well as the scikit-learn classifier.

4.2 Feed-Forward Neural Network

4.2.1 Implementation

Artificial neural networks are versatile, powerful and highly complex models that can tackle almost any machine learning problem. They are inspired by the giant network of biological neurons found in the brain. The artificial neural network contains neurons which activate the neurons in the next layer in the same fashion as biological neurons are connected to other neurons via synaptic terminals (Géron, 2019). There are many design choices you can make when constructing a feed-forward neural network, such as choosing which activation function and weight initialisation to use, and deciding whether or not to use drop out regularisation along with many other components that could improve the model's performance. Our implementation is limited to only be able to handle multi- class classification with softmax as the output activation function and Leaky ReLU activation function for hidden layers. Furthermore the network is fully connected, the loss function is categorical cross entropy, however the number of neurons and number of layers are fully adjustable.

The fully connected network `NeuralNetworkClassifier()` is constructed using the `Dense Layer()` class object. Each `DenseLayer()` is initialised in the desired input and output dimension. Each layer has the class attributes `self.weights` which is initialised using He-normal for our hidden layers and `self.biases` is initialised as a vector of zeros. For the output layer, GlorotNormal is used as initialisation. The main purpose of the `DenseLayer()` class is to perform the forward pass using the `forward()` function. The function takes a matrix $\mathbf{X}$ as input. For the first hidden layer $\mathbf{X}$ is the actual data set and for later layers it is a matrix of activations from the previous layer(s). It performs the forward pass, i.e. it computes the linear combination of `self.weights`, $\mathbf{X}$ and `self.biases` using the given activation function `self.activation`. This is shown in the code snippet down below, from the `DenseLayer()` class instance, line 64-65.

```
self.z = X.dot(self.weights) + self.biases
return self.activation(self.z)
```

The `NeuralNetworkClassifier()` class takes as input a list of initialised `DenseLayer()` objects. The actual fitting happens when the function `fit()` is called with `X` and `y` as input. Given that the weights and biases are randomly initialised, we need to optimise them in order to reach the minimum of the loss function, yielding in optimal predictions. When training a Neural Network you make a forward pass with the random parameters, calculate the loss, and optimise the network with the help of back propagation. Back propagation gradually updates the weights and biases by using gradient descent. We chose to implement it in a vectorised fashion. This can be achieved in a few lines of code (Appendix B) because the same chain-rule pattern is used to update the parameters in each layer (Figure 4.1). Using this fact we can split the back propagation into two steps. Following the chain rule we simply use the partial derivatives from previous steps in order to obtain the partial derivative for the current step. We have $0, 1, ..., \ell$ layers, with the ℓth layer being the last (output) layer in the neural network. And we have $0, 1, .., k$ weight and bias matrices, with the kth matrices being the last set of parameters. In Algorithm 4.1 the back propagation steps from Figure 4.1 are explained in detail.

Algorithm 4.1 Back propagation

Let $\boldsymbol{\delta}$ denote the partial derivatives of the previous steps (starting from output layer) right before the weights and biases. E.g. $\boldsymbol{\delta}$ is the chain of previous derivatives up until a set of parameters $\boldsymbol{w}^{(i)}$ and $\boldsymbol{b}^{(i)}$, and would then be $\frac{\partial L}{\partial \boldsymbol{Z}^{(i+1)}}$.

Step 1: Compute the partial derivative of the linear combinations w.r.t the loss function in order to compute the gradients of the weights and biases in the last layer

$$\boldsymbol{\delta} = \frac{\partial L}{\partial \boldsymbol{Z}^{(\ell)}} = \boldsymbol{a}^{(\ell)} - \boldsymbol{y}$$

Gradients for the last set of weights and biases is then

$$\frac{\partial L}{\partial \boldsymbol{w}^{(k)}} = \left(\boldsymbol{a}^{(l-1)}\right)^T \cdot \boldsymbol{\delta} \qquad \text{and} \qquad \frac{\partial L}{\partial \boldsymbol{b}^k} = \boldsymbol{\delta}$$

Step 2: The second step is repeated $k - 1$ times such that the gradients for $\boldsymbol{w}^{(k-1)}, ..., \boldsymbol{w}^{(1)}$ and $\boldsymbol{b}^{(k-1)}, ..., \boldsymbol{b}^{(1)}$ is computed. For each step $i \in \{1, 2, .., k-1\}$

$$\frac{\partial L}{\partial \boldsymbol{Z}^{(\ell-i)}} = \boldsymbol{\delta} \cdot \left(\boldsymbol{w}^{(k-i+1)}\right)^T \odot f'(\boldsymbol{Z}^{(l-i)})$$

Now we update $\boldsymbol{\delta}$ for the next set of parameters.

$$\boldsymbol{\delta} := \frac{\partial L}{\partial \boldsymbol{Z}^{(\ell-i)}}$$

Gradients for the weights and biases of layer $k - i$ is then

$$\frac{\partial L}{\partial \boldsymbol{w}^{(k-i)}} = \left(\boldsymbol{a}^{(l-i-1)}\right)^T \cdot \boldsymbol{\delta} \qquad \text{and} \qquad \frac{\partial L}{\partial \boldsymbol{b}^{(k-i)}} = \boldsymbol{\delta}$$

Step 3: When all the gradients have been computed all the weights and biases for all layers are then updated using the following general formula:

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{w}}$$
$$\boldsymbol{b} := \boldsymbol{b} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{b}}$$

The back propagation in Figure 4.1 is repeated for every batch. When all the batches of the training data have been back propagated and all parameters have been updated, one epoch is completed – both the number of epochs and batches are specified before training.

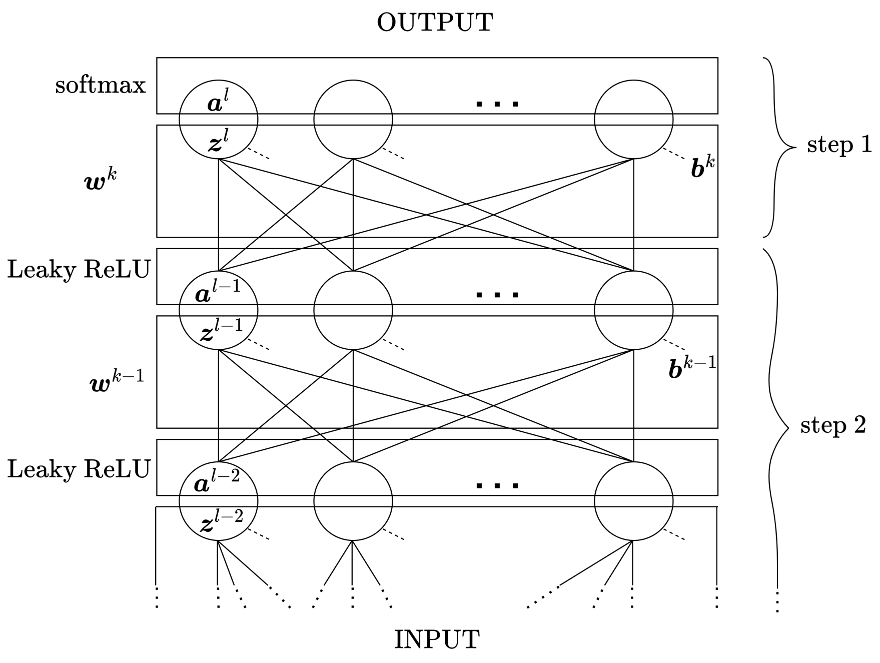

FIGURE 4.1: *Illustration of a general feed-forward neural network*

4.2.2 Predicting

Predicting in a feed-forward neural network is done by performing a forward pass. As mentioned earlier, a forward pass is, as the name suggests, the data points passing through the network, and one by one the weighted sum is computed together with the weights and biases. The weighted sum is then passed through the given activation function, and the process is then repeated until the last layer is reached. In the last layer the five outputs are passed through the softmax activation function and a prediction is made to the class with the highest probability.

4.2.3 Evaluating Correctness

A neural network aims at approximating a function that can map attributes to an output, and is rather good at it. Even simple networks can approximate any continuous function with reasonable accuracy. We constructed a reference implementation from the keras API (Chollet et al., 2015) of the TensorFlow library, yielding a accuracy of 73% with the same parameters as our own model. But using a reference implementation to compare to our own implementation is not informative enough, because it is difficult to reproduce the exact same model. Therefore we decided to look at other methods to evaluate our model.

An essential indicator that an implementation is correct, is if the model is able to learn the patterns of the training set. That is, we train the model and obtain a reasonable accuracy. Because of the flexibility of the learning method, we also expect the model to be able to completely overfit the training data. Lastly, since we train the model using gradient descent, we expect the training accuracy to increase for each epoch, and the loss to decrease. We first evaluate our implementation, on two toy data sets checking if we are able to overfit the data i.e., obtaining a training accuracy close to 100%, using a simple network. We train the two data sets with their respective model both having one hidden layer with 20 neurons. As expected, the two models were able to overfit their training data.

Then we evaluate our implementation again on a stratified subset of 500 samples of the given fashion

MNIST training set. We train a simple network with 1 hidden layer of 20 neurons with Leaky ReLU as activation function and softmax as the output activation function. We used 30 batches and 200 epochs with a learning rate of 0.0001. The loss history and training accuracy plotted against epochs is depicted in Figure 4.2.

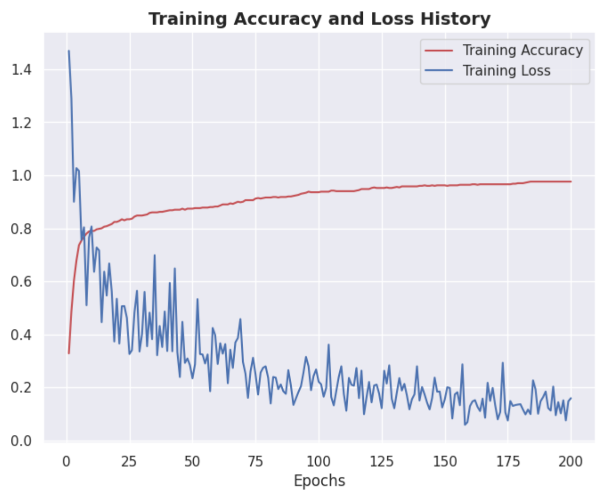

FIGURE 4.2: *Over fitting subset of training set with our own implementation of the neural network*

It is clear that the model is able to overfit the training data and that the accuracy increases while the loss decreases. Notice that the loss does not decrease for each epoch, but is rather jagged because of the small batch size. We can conclude that all the evaluations of the model indicate that the model has been implemented correctly and is in fact able to learn the patterns of the data it has been given.

5 Results

5.1 Decision Tree

5.1.1 Hyper Parameters

To reduce the computational cost of training the classifier, we used a subset of the the extracted features from the principle components analysis. We determined our features to be the first 62 principal components. We saw earlier that the first 62 principal components had a cumulative explained variance ratio of 90%. We found that a CVR of 90% was sufficient since it reduced the feature dimensionality and thereby the cost of training while also preserving most of the information from the original data set. This data set of 62 standardised principal components from the training set was then used to find the best hyper parameters for our model. The different hyper parameters our model provides limits to (splitting) `criterion`, `max_depth` and `min_samples_leaf`. Given that we only have three hyper parameters, we found it suitable to perform a grid search to find the hyper parameters that yield the best performing classifier. For the grid search, we chose `max_depth` with a range from 1 to 20, and `min_samples_leaf` with a range from 1 to 30 and the splitting criterion to be either Gini or Entropy.

The `GridSearchCV` from scikit-learn considers all possible combinations of the parameters. For each combination it fits the model and computes the mean accuracy, using k-fold cross validation with k set to 5. We are then able to call `best_params_` in order to obtain the combination of the given hyper parameters that results in the best accuracy score out of the given options. The hyper parameters that resulted in the best performance on the validation splits were a max depth of 15, minimum number of samples in a leaf node 21 and Gini Impurity as the splitting criterion.

5.1.2 Results and Performance

The fitted and trained decision tree, with the best performing hyper parameters, resulted in an accuracy score of 76% on the test split. As depicted in (Table 5.1) the classifier is insufficient at predicting class 4 (Shirt), looking at the F1-score, but adequate at predicting the other classes.

	T-shirt/top	Trouser	Pullover	Dress	Shirt	
Precision	0.73	0.96	0.75	0.80	0.57	
Recall	0.72	0.91	0.77	0.82	0.57	
F1-score	0.73	0.93	0.76	0.81	0.57	
Training Accuracy						0.90
Test Accuracy						0.76

TABLE 5.1: *Decision tree results*

A decision tree is a model with high variance. This means that a small change in the data can result in a major change in the structure of the tree and ultimately convey a different result. We make our prediction based on one decision tree only, and therefore expect the classifier to have high variance. This might explain the considerable drop in accuracy when evaluating the model on the test set. Often one would use bagging or boosting to prevent this.

Using principle component analysis for dimensionality reduction is a very useful technique, considering the fact that it can prevent overfitting, improve performance and make computations less expensive. Nevertheless, principle components are troublesome to interpret. Even though a decision tree classifier is an easyly interpretable model, it is difficult to illustrate which features are more important than others when we use principle components. Additionally, to some extent, a part of the information is lost when dimensions are reduced. Choosing the number of principal components is a trade-off between dimensionality and information loss. Choosing a small number of principal components amounts in a substantial dimensionality reduction, and thus reducing computation. However, it also increases information loss.

5.2 Neural Network

5.2.1 Hyper Parameters and Architecture

The selection of hyper parameters in our feed-forward neural network implementation was boundless, considering the fact that the structure of the model can be so vast. Therefore systematic optimisation strategies for a complex model like this, can be very exhaustive and computationally expensive. Hence the model selection in this case, was based on trial and error.

The final structure of our feed-forward neural network consisted of three layers with a single hidden layer. The first (input) layer with 32 neurons. The hidden layer consisted of 16 neurons, and the last (output) layer consisted of 5 neurons. Both the input layer and the hidden layer had Leaky ReLU activation functions, and the output layer had a softmax activation function. We standardised the input with the scikit-learn `StandardScaler` instance before feeding it to the network. We ran 75 epochs with the training data split into 100 batches and a learning rate of 0.0001.

5.2.2 Results and Performance

The fitted and trained Neural Network performed with a training accuracy of 99% and a test accuracy of 85%. Looking at Table 5.2, it is clear to see from the F1-score that the model is great at predicting class 1 (Trousers), but less precise at class 4 (Shirts).

	T-shirt/top	Trouser	Pullover	Dress	Shirt	
Precision	0.80	0.97	0.83	0.90	0.72	
Recall	0.81	0.96	0.87	0.88	0.70	
F1-score	0.80	0.97	0.85	0.89	0.71	
Training Accuracy						0.99
Test Accuracy						0.84

TABLE 5.2: *Feed-forward neural network results*

A problem we encountered with our neural network implementation was overfitting. This is not surprising since all standard, fully connected neural network architectures are prone to overfitting (Prechelt, 2012). Our implementation does not have any methods that prevent overfitting, since we chose to keep the model simple. Normally you could prevent this by implementing early stopping. That is we break the training when a parameter update no longer improves the validation data. This is a common regularisation technique for deep learning models, to avoid poor performance. Overfitting can also be prevented by using other regularisation methods like drop-out, L1 and L2 regularisation.

5.3 Convolutional Neural Network

5.3.1 Introduction

We have chosen our third model to be a convolutional neural network. When working with normal feed-forward neural networks you lose the structure of the image, since you need to flatten the image array. This is where convolutional neural networks really shine, since the structure of the image is preserved. To some degree they mimic how we, as humans classify images which is by recognising specific patterns and features anywhere in the image that differentiate each distinct object class.

The architechture of a convolutional neural network consists of two major parts, a feature learning/extraction part and a classification part. The classification part, is constructed in the same way as a normal feed-forward neural network.

5.3.2 Convolutional and Pooling Layers

A convolutional layer is made up of a number of convolutional filters or kernels, each of these are templates that help determine whether a particular local feature is present in an image or not. These features are then combined to form higher-level features, such as parts of eyes, ears and the like. A kernel relies on a simple operation, called a *convolution*, which essentially amounts to repeatedly multiplying matrix elements and summing the results (James et al., 2021). A *pooling* layer provides a way to condense a large image into a smaller summary image. There are numerous ways to perform pooling.

Both the convolutional layers and the pooling layers help reduce the spacial size of the convolved feature. This is done to decrease the overall computational power required to process the data via dimensionality reduction. In the end the presence or absence of these high-level features help determine the probability of any given output class.

5.3.3 Hyper Parameters and Architecture

For the convolutional neural network we used the LeNet-5 architecture (Figure 5.1).

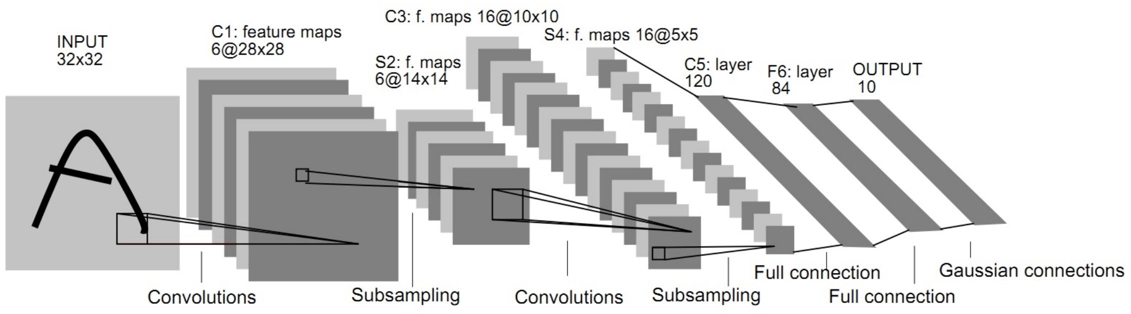

FIGURE 5.1: *LeNet-5 convolutional neural network architecture*

This architecture was proposed by Yann LeCun and others in 1998 for recognising handwritten and machine-printed characters (LeCun, Bottou, et al., 1998). Since then LeCun has been recognised as a founding father of convolutional neural networks.

Because the LeNet-5 architecture was created for recognising characters in different data sets, including the MNIST data set (LeCun, Cortes, et al., 1998), we figured that the same architecture would be well suited for the Fashion MNIST data set as well. Compared to our own feed-forward neural network we were able to implement early stopping, to prevent overfitting.

5.3.4 Results and Performance

The fitted LeNet-5 convolutional neural network performed with a training accuracy of 92% and a test accuracy of 87%. As we expected the model is better at correctly distinguishing and predicting the classes, that our own neural network struggled with (Table 5.3).

	T-shirt/top	Trouser	Pullover	Dress	Shirt	
Precision	0.80	0.98	0.88	0.95	0.75	
Recall	0.87	0.97	0.89	0.88	0.73	
F1-score	0.83	0.98	0.88	0.91	0.74	
Training Accuracy						0.92
Test Accuracy						0.87

TABLE 5.3: *LeNet-5 convolutional neural network results*

6 Discussion

The problem of assigning a class to grey-scale images of clothing came with some difficulties. Considering the different complexity of the three models that we implemented, we still see the same trends in the predictions.

6.1 Class Overlap

One of the main challenges all the models faced with the Fashion MNIST data set, was class overlap. As mentioned earlier, our feed-forward neural network wrongly classified class 4 (Shirt) as class 0 (T-shirt/top) and class 2 (Pullover). Taking a look at all the confusion matrices for the three classifiers, we see an evident tendency that class 4 (Shirt) is misclassified. This is also what we expected since the three items of clothing are visually quite similar, which explains the class overlap we discovered. When we compare that to class 1 (Trouser) which are a more unique type of clothing, it is easier for the models to distinguish, this is also depicted clearly in all of the confusion matrices.

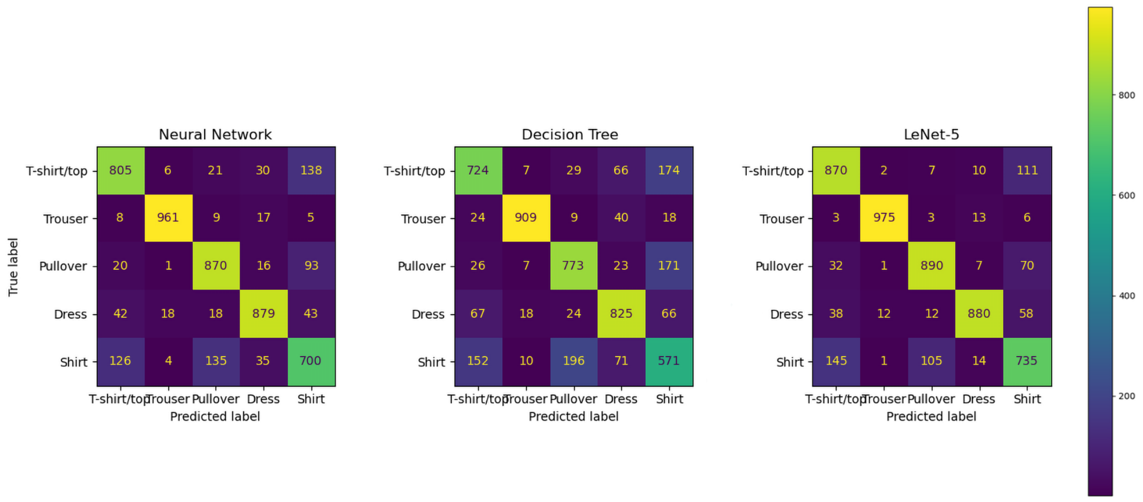

FIGURE 6.1: *Confusion matrices for all classifiers*

6.2 Model Comparison

Decision trees and tree-based approaches in general, are typically not as competitive as other machine learning approaches in terms of accuracy (James et al., 2021).

The advantage of using a decision tree classifier is that it is a highly interpretable model and it is possible to visualise each decision. Furthermore it has a shorter training time than other machine learning methods since the tree is usually built with a top-down greedy approach. However, with the given classification problem at hand the interpretability the model provides is not as useful compared to other situations. First of all, we are training the model on principal components, already at this point we are loosing a lot of interpretability. If we were to train on the original data set, we would still lack interpretability since the features are low-level features.

If interpretability is not attainable or desirable, more complex models can be used. For such highly complex data sets, like the Fashion MNIST data set, a better suited model for classification is a feed-forward neural network. Feed-forward neural networks are excellent at solving complex classification problems compared to simpler models, and therefore we get a better performance than our decision tree. Nevertheless, feed-forward neural networks are not able to capture the locality of features. Convolutional neural networks, on the other hand, are built for image analysis. They are able to process image data in two and three- dimensions with the help of convolutional and pooling layers. This is a big advantage over the other two models since both are only able to handle one-dimensional data. This is likely also a reason why our results for the convolutional neural network are better than the other two models.

7 Conclusion

Out of the three classification models we used for this task, we found that the convolutional neural network achieved the best results. The model performed with an accuracy of 87% on unseen test data, which is a reasonable result given that we encountered problems such as class overlap. The model could be improved with a more in-depth examination of the architecture, while the use of ensemble methods involving multiple models could also be explored for the purpose of improving classification performance.

References

Chollet, F., et al. (2015). Keras. https://keras.io

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Incorporated.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An introduction to statistical learning: With applications in R*. Springer Nature.

LeCun, Y., Bottou, L., Bengio, Y., & Ha, P. (1998). Gradient-based learning applied to document recognition.

LeCun, Y., Cortes, C., & Burges, C. J. C. (1998). The MNIST database of handwirtten digits. http://yann.lecun.com/exdb/mnist/

Prechelt, L. (2012). Early stopping but when? In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade: Second edition* (pp. 53–67). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_5

Xiao, H., Rasul, K., & Vollgraf, R. (2017, August 28). *Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms*.

Appendix

A Pairplot of the first 5 principal components

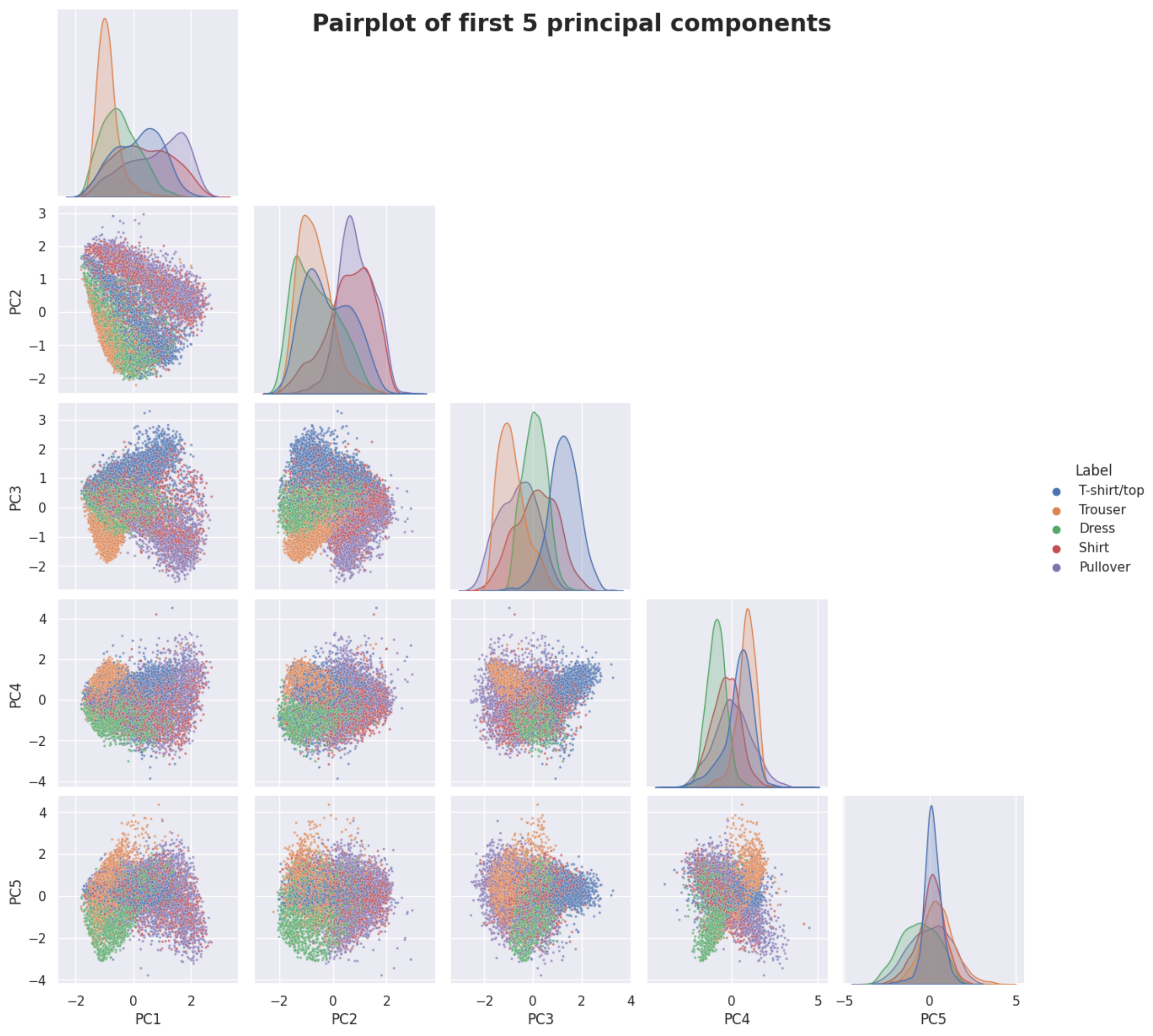

FIGURE A.1: *PCA pairplot*

B Backpropagation Python implementation

The highlighted lines are where the partial derivatives w.r.t. the different parameters are derived. Lines 11-15 are where the gradients of the loss function w.r.t. the weights and biases in the final (output) layer are calculated. Lines 21-27 are where the gradients for each of the subsequent layers (that all have Leaky ReLU activation function and therefore same derivative) are calculated.

```python
1      def _backward(self, y_batch):
2          """Computes a single backward pass all the way through the
            ↪  network.
3          Also updates the weights and biases.
4  
5          Parameters
6          ----------
7          y_batch : 2d ndarray
8              array of one-hot encoded ground_truth labels
9          """
10 
11         delta = self.activations[-1] - y_batch
12 
13         grad_bias = delta.sum(0)
14 
15         grad_weight = self.activations[-2].T @ delta
16 
17         grad_biases, grad_weights = [], []
18         grad_weights.append(grad_weight)
19         grad_biases.append(grad_bias)
20 
21         for i in range(2, len(self.layers) + 1):
22             layer = self.layers[-i + 1]
23             dzda = delta @ layer.weights.T
24             delta = dzda * leaky_relu_der(self.sums[-i])
25 
26             grad_bias = delta.sum(0)
27             grad_weight = self.activations[-i - 1].T @ delta
28             grad_weights.append(grad_weight)
29             grad_biases.append(grad_bias)
30 
31         # reverse the gradient lists so we can index them normally.
32         grad_biases_rev = list(reversed(grad_biases))
33         grad_weights_rev = list(reversed(grad_weights))
34 
35         for i in range(0, len(self.layers)):
36             self.layers[i].weights -= self.learning_rate *
                ↪  grad_weights_rev[i]
37             self.layers[i].biases -= self.learning_rate *
                ↪  grad_biases_rev[i]
```