Machine Learning Project
BSMALEA1KU

Hubert Wójcik (huwo@itu.dk) and Paula Menshikoff (pmen@itu.dk)

December 2022

1 Introduction

Fast shopping is a term used to define the design, marketing, and sales approach to make clothing more accessible and cheaper. Online shopping is an easy and comfortable way to buy clothing online through websites like Zalando. Companies' incomes increase as they facilitate their customers' shopping experience.

On this basis, we have been given the task to build a machine learning model that was able to classify different types of clothing based on an image of the item to make it easier for customers to filter their search by clothing item.

2 Data Description

The data used for this project was obtained from a study carried out in 2017 (Xiao, Rasul, and Vollgraf 2017). The data set is composed of 15,000 labeled images of clothing items from 5 categories (t-shit/top, trousers, a pullover, a dress, or a shirt). The training set is composed of 10,000 images, whereas the test set has 5,000 images. Each image is a grayscale 28x28 picture in NPY format

2.1 Data Cleaning

Before analysing the pictures, data integrity checks were carried out. We looked for missing values, missing labels, and wrong classification of the pictures. However, no inconsistencies were found. Thus, no further data cleaning was necessary.

3 Exploratory Data Analysis

3.1 Class Distribution

Figures 1 and 2 display the class distribution in the training and test split. The distribution of classes was even amongst both splits. This is worth mentioning because an uneven distribution of the classes could result in major challenges for any machine learning model.

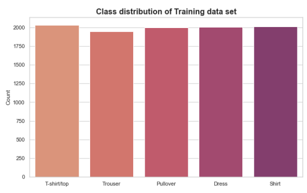


Figure 1: Class distribution of training data set

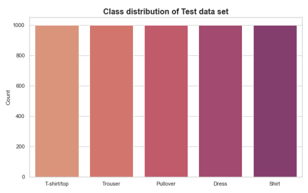


Figure 2: Class distribution of test data set

3.2 Distribution of Features for Each Class

Pictures 3 and 4 display the mean and median images respectively. The mean and median are calculated from the individual pixel values within each class, which gives an intuition of how similar the pictures are to each other. With the figures, we can see that the whiter the pixel the higher the mean, meaning that that pixel is likely to appear among all images of the same class. This also means that all items in the data set appear to be centered, which could benefit the functioning of the machine learning models.

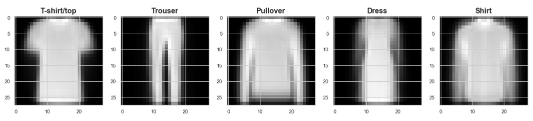


Figure 3: Average image based on mean

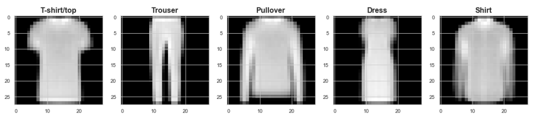


Figure 4: Average image based on median

Figure 5, on the other side, displays the standard deviation of the pixel values within each class. The aforementioned figure shows the deviation of each pixel value from the mean for each pixel. The darker areas indicate lower variation among the pixel values.

3.3 Principal Component Analysis

Figure 6 shows the first three principal components of the data set. The visualization provides an understanding of the difficulty of the classification problem. The principal component analysis is a variance-focused approach. PCA is a feature-based classification technique that is characteristically used for image recognition (Bajwa et al. 2008). PCA is based on the principal features of an image and these features discreetly represent an image. PCA is used in this project to facilitate the construction

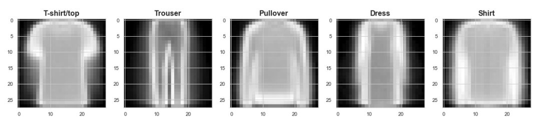


Figure 5: Average image based on standard deviation

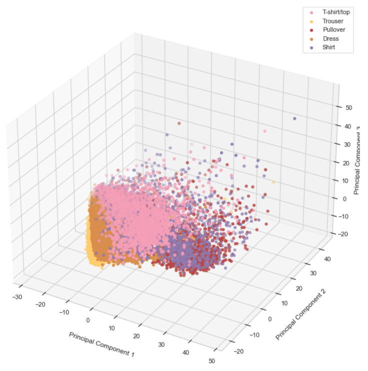


Figure 6: Principal Component Analysis

of the decision tree algorithm. The designed system reads features of gray-level images to create an image space. This image space is used for the classification of images using our custom decision tree. In the testing phase, a new clothing image is classified by comparing it with the specified image space using the PCA algorithm.

4 Implementation

4.1 Neural Network

In order to improve the clarity and maintainability of our code, we split our code into three classes.

1. `MiniBatchGD` - This class is responsible for dividing the data into mini-batches of a specified size. It has a single method, `sample()`, which returns a random batch of data from the dataset.
2. `ActivationFunction` - This class handles activation functions, including the ability to specify the function name and learning rate as constructor parameters. It includes implementations of several common activation functions, such as sigmoid, tanh, and leaky ReLU, as well as their derivatives. This class enables us to easily change the activation function used in the network.
3. `NeuralNetwork` - This is the main class that performs the forward pass, backpropagation, and other necessary steps for creating a neural network. It also includes methods for making predictions and evaluating the model's performance.

When initializing this class, we pass in various parameters such as the input layer size, hidden layer size, number of output classes, etc. These parameters are used to generate weight and bias matrices: The weights were created using He initialization, as it is perhaps one of the most popular initialization schemes in the deep learning community, especially when the ReLU activation function is concerned

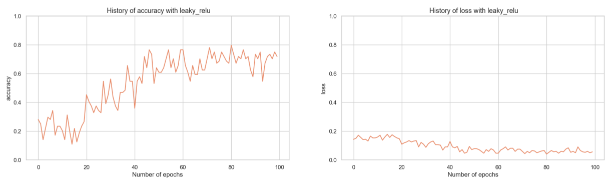


Figure 7: History of Neural Network's accuracy and loss

(Lu et al. 2020). We have experienced the dying ReLU issue, thus choosing the right initialization technique was vital.

Given the constraints on computational resources and the potential challenges in training, it was decided to use a shallow feed-forward neural network. Therefore, the model consists of one hidden layer and an output layer, requiring the creation of two weight matrices and two bias matrices.

During the forward pass, a dot product of our training data and weights for both layers was computed, and the bias was added to the result. Then, both results were put in an activation function of choice. Finally, we created an array called `self.outputs` to store the 1-hot encoded outputs of the model.

The backpropagation process involved calculating the gradient of loss functions for both layers using the chain rule. This process resulted in the creation of four matrices containing updated values: two matrices for weights and two matrices for biases. A function was then called to apply these updates to the weights and biases. Each update was also normalized. Moreover, we computed accuracy to track the algorithm's progress.

To train the network, we set up a loop to run for a specified number of epochs. Within this loop, we iterated over the data using mini-batches and appended the weights updates to an array. The final update was obtained by averaging the updated weights and biases. Then, the `update_weights()` function was called to apply these updates to the weights and biases matrices. The training function had two parameters on top of `X` and `y`: the number of `epochs` to run and the `batch_num`, which is to specify how many times we sample from the dataset per epoch.

4.1.1 Assessing Correctness

To evaluate the accuracy of our neural network, we implemented a simple neural network using Keras and found that it had an accuracy of 81% on the test data. In comparison, our model had an accuracy of 70% on the test data, indicating that there is room for improvement in our implementation. However, the difference in accuracy is not very large, which suggests that our model is still reasonable.

Unlike our implementation, the Keras model did not have any issues with misclassifying pullovers as shirts, which results in way better accuracy for pullovers. Apart from that, it had a slightly higher classification accuracy for all classes except t-shirts (almost the same), which also shows the cause of the difference in performance between the two models. The details of our neural network's results are discussed in 5.1.3.

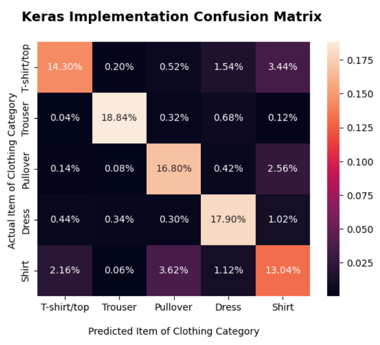


Figure 8: Neural Network Confusion Matrix

4.2 Reference Implementation - Convolutional Neural Network

We chose to use PyTorch for our "reference implementation" of a convolutional neural network because it is well-suited for image classification tasks, offers excellent performance, and the possibility to design flexible models.

Firstly, a class was implemented to process the data for use in the model. This involved reshaping the images into 28x28 matrices and selecting the target variable. Several transformations were applied to the data: conversion to PyTorch tensors, normalization of values, and grayscale conversion. The resulting dataset was wrapped in a `DataLoader` object for use in the model.

To construct the Convolutional Neural Network (CNN), we used the PyTorch documentation as a reference but modified the model to better fit our specific scenario. The CNN was implemented as a subclass of `nn.Module` called `Net`.

Our CNN had two convolutional layers (created using nn.Conv2d function), each with a kernel size of 5x5 and using a max pooling layer with a size of 2x2 and stride of 2. We chose ReLU as the activation function due to its aforementioned widespread popularity, its efficiency, and versatility. The model ended with 3 fully connected layers, with the final layer outputting 5 features.

The model was trained using PyTorch's provided functions with a cross-entropy loss function and the SGD optimizer with a momentum of 0.7. The training process involved looping through the data loader for a set number of epochs, resetting the gradients, performing forward and backward propagation, and updating the weights.

4.3 Decision Tree

The decision tree classifier was implemented using three different classes:

1. `Node` - Helper-class used to represent a node in a decision tree.
2. `InformationGain` - Helper-class that calculates the information gain of a split.
3. `DecisionTreeClassifier` - Constructor for setting up the model and determining stopping conditions

The Node() class initializes the node of a tree. It holds information about a single split at any level in a decision tree. It has the feature index, threshold, left and right nodes, and information gain.

For the leaf node, it only possesses the value or class.

The `DecisionTreeClassifier` initializes the root of the tree and determines the stopping conditions. The class builds the tree recursively with the `build_tree` function. The `build_tree()` function starts with an if statement to detect if the stopping conditions are met. If not, it looks for the best split with the `best_split()` function. The `best_split()` function loops over all the features, then we have an inner loop that loops over all the feature values present in the data, gets the current split, checks if the child is not null, computes the information gain with the class and updates the best split if necessary. It keeps building the tree recursively until it returns the decision node. When the stopping conditions are met it computes and returns the leaf node.

`InformationGain` indicates how much information a particular split gives us about the final outcome. It is initialized with the parent, right and left child, and a method. It calculates the information gain based on cross-entropy or Gini index.

The Decision Tree code has been developed based on Dutta 2023, and the link to the source code appears in the references in case it is needed.

4.3.1 Assessing correctness

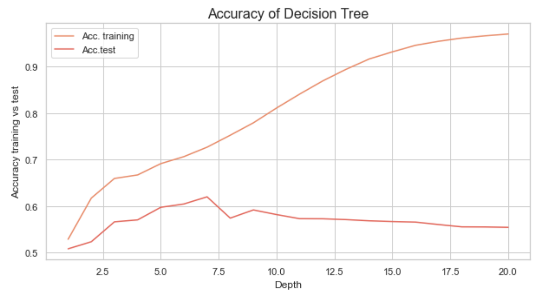


Figure 9: History of accuracy (training and test data)

As any machine learning model, a decision tree classifier is able to overfit the training dataset to 100% training accuracy. This is why the custom classifier has been tested for the same property. Figure 7 shows the training and test accuracy of the given data set. As expected, the training accuracy has a steady increase until an 100% accuracy is achieved. A decision tree with 20 as maximum depth is enough to overfit the dataset. Over-fitting occurs because the models learn more and more of the variation (and noise) as the maximum depth increases. It ends up with branches with strict rules that perfectly fit all samples in the training dataset.

Overall, the implementation behaves as expected.

5 Details on Machine Learning Models

5.1 Neural Network

5.1.1 Introduction

A neural network is a type of machine learning model that is supposed to work and generally be structured in a similar way to our brain. It stores a large number of nodes, which using the activation functions and interconnected layers process inputs and make predictions. Neural Networks are well-known techniques for classification problems. They can also be applied to regression problems.

A neural network consists of three types of layers: input, hidden, and output. The input layer receives the input data and the hidden layers process it, producing an output that is passed to the final output layer to produce the result of the network, which is a prediction.

5.1.2 Hyperparameters

During the development of our own neural network, it was important to carefully select the hyperparameters in order to achieve good performance. To facilitate this process, we utilized a grid search for a portion of the hyperparameters.

Based on the rather simple structure of our neural network, there was a possibility to have a relatively big batch size, e.g. 128 without hindering the performance. This turned out to be beneficial for the training dataset's accuracy. However, it was decided to use a smaller batch as it introduces some noise, which in theory prevents overfitting.

After a number of experiments, the final hyperparameters and parameters were chosen:

- Number of Epochs - we chose it to be 100 because each epoch does not take much time to run. Additionally, it was observed that after 100 epochs, the model's performance does not improve significantly.
- Learning rate - as we did not want to overshoot the right solution, we decided to implement a relatively small learning rate of 0.01.
- Batch size - as explained above, it was decided to choose a smaller batch size than 128. 64 was the final metric, however, there was not much difference between 64 and 32.
- Number of Batches - we have found that averaging the updated weights for 5 batches per epoch is sufficient to achieve good results.
- Activation function - two activation functions were considered for the final model: leaky ReLU and sigmoid. Ultimately, ReLU was chosen because it generally provides better results, even though it is less stable than sigmoid.
- Leaky ReLU learning rate - we initially set the learning rate for leaky ReLU to 0.001, but the model frequently produced matrices containing NaN values because the numbers were too small

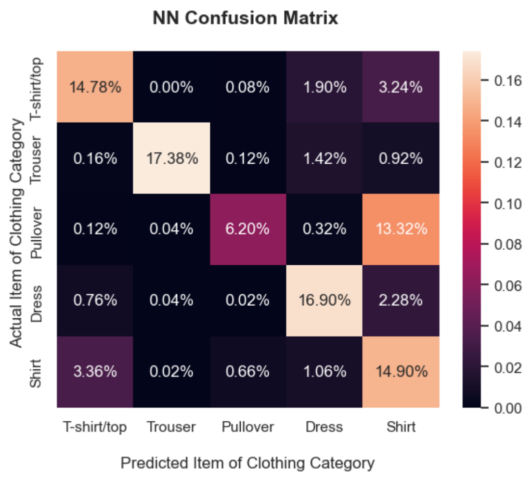


Figure 10: Neural Network Confusion Matrix

for Python to handle. We eventually found that a learning rate of 0.1 produced satisfactory results.

5.1.3 Performance Evaluation

The final model's average accuracy was 73% for the training data and 70% for the test data. From the confusion matrix, we can observe that the model had difficulties in distinguishing between a pullover and a shirt. Moreover, the activation function appears to be an important factor in the model's performance, with ReLU resulting in better overall accuracy compared to sigmoid, although the results with ReLU were not as consistent. Overall, one could expect that the model would achieve an accuracy of around 70%, given the structure of the neural network and the minimal data preprocessing that was performed. Given the limitations mentioned in 6.1, we believe that a good result has been achieved. However, one can clearly observe that there is room for improvement.

5.2 Convolutional Neural Network

5.2.1 Introduction

A convolutional neural network is a type of neural network specifically designed to process data with a grid-like topology, for instance, an image. CNNs are particularly well-suited for image classification tasks because they are able to automatically learn and extract features from the input data, rather than requiring the user to specify these features manually.

5.2.2 Hyperparameters

Most of the hyperparameters for the CNN were chosen based on the PyTorch documentation and generally accepted best practices in the field.

- Epochs number - the number of epochs was limited to 10 to keep the training process efficient, as CNNs require more computations than other types of networks. The batch size was also set

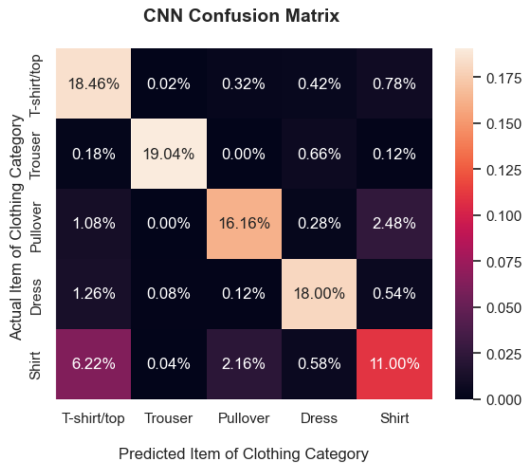


Figure 11: Convolutional Neural Network Confusion Matrix

to 8 for similar reasons.

- Learning rate - we conducted a learning rate range test to determine the best learning rate for the model. After testing various values, we found that a learning rate of 0.001 provided the best results.
- Momentum - it is recommended to use a momentum rate between 0.1 and 0.9. The momentum term in our model was set to 0.7 after experimentation showed that it yielded the best results.
- Layer Parameters - we selected a kernel size of 5x5 because it is a commonly used size, along with 3x3. Using a max pooling layer instead of average pooling yielded better results, and the stride helped to improve computational efficiency and decrease model complexity.

5.2.3 Performance Evaluation

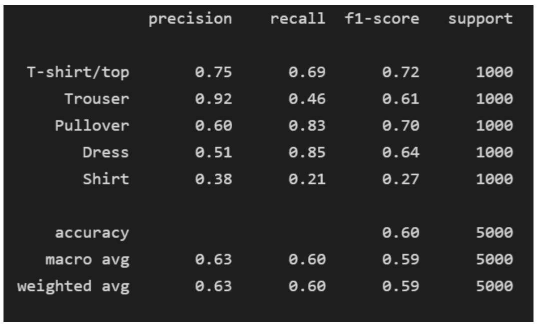
Given PyTorch's strong suit in image classification, we had high expectations even though we implemented a simple convolutional neural network. The model performed well in both feature selection and training on the provided data, resulting in an accuracy of 83.56% and F1 Score of 83.75% for the test data. The classifier did well in classifying all the classes, especially when it comes to dresses and trousers. However, it had some difficulties in distinguishing between t-shirts and shirts, as could have been expected. This issue may potentially be addressed in the future.

All in all, the reference implementation resulted in a highly effective model, and it has the potential to be further optimized in the future.

5.3 Decision Tree

5.3.1 Introduction

Decision trees are statistical methods used for classification and regression problems. These machine learning models are non-parametric models, and thus they do not approximate a function to describe



```
              precision    recall  f1-score   support

 T-shirt/top       0.75      0.69      0.72      1000
     Trouser       0.92      0.46      0.61      1000
    Pullover       0.60      0.83      0.70      1000
       Dress       0.51      0.85      0.64      1000
       Shirt       0.38      0.21      0.27      1000

    accuracy                           0.60      5000
   macro avg       0.63      0.60      0.59      5000
weighted avg       0.63      0.60      0.59      5000
```

Figure 12: Classification report from Decision Tree

the relationship between the features and target. Instead, decision trees classify data based on the information that is provided as a reference to build a hierarchical decision structure. Decision tree bases its construction on the idea of segmentation. The feature space is divided into a number of simpler regions. Thus, the biggest challenge usually is identifying the best split. In order to do so loss functions like gini-impurity, cross-entropy, or 0-1 loss, have to be minimised for each data point in the training split. However, it is computationally impossible to consider every possible split of the feature space into X regions and evaluate the loss. For this reason, a top-down, greedy algorithm is necessary to accomplish this task. It is a recursive algorithm that splits a partition of the training set in two that best divide the classes. Decision trees are built by recursively applying this algorithm until some stopping condition is achieved, or there is no mix between classes in all the regions. Finally, the prediction is based on identifying which region a new data point belongs to and then predicting the majority class in that region.

5.3.2 Hyperparameters

As with the Neural Network, we focus on the hyperparameters in order to achieve a good performance. After experiments were run, the recommended parameters for our custom model are:

- max_depth of 6 since the decision tree adjusts itself to the data and an increase in the depth would end up in an overfitted model.
- Mode which can be gini or cross-entropy, but cross entropy was chosen for this custom model.
- min_samples_split a fairly low value of 3.

5.3.3 Decision Tree Performance Evaluation

The best generalizing model achieved an average accuracy score in the test accuracy of 60% for a maximum depth of 6. However, this is not a surprise given that decision trees are not the most suitable tool for image classification. From the classification report, we can see that the decision tree struggles with identifying shirts, but has better F1 scores when it comes to T-shirts, dresses and pullovers, and trousers.

6 Interpretation and Discussion of the Results

6.1 Limitations and General Challenges

The provided dataset being large and well-balanced made the project easier. However, we faced some challenges and limitations, including a lack of sufficient computational power to create and train a very deep neural network on the entire dataset, and limited time to explore various methods of parameter initialization, training, and model analysis. While these limitations did not significantly impact the project, which was for academic purposes, we are positive that there are various opportunities to further improve the model.

6.2 Comparison of Model Performances

The results of the three models implemented in the project varied for the provided clothing dataset. The neural network implementation produced a satisfactory model that could be improved in the future. On the other hand, the convolutional neural network using PyTorch achieved better results. This is likely due to the fact that CNN is a particularly effective method for image classification, and PyTorch is a powerful and very helpful tool for building neural networks. The decision tree model, which is not commonly used for image classification, had the lowest accuracy.

6.2.1 Results per class

The convolutional network module performs well for all classifications except for distinguishing between t-shirts and shirts. The decision tree module has difficulty differentiating between dresses and trousers, and pullovers and shirts. Additionally, the decision tree module has lower overall accuracy compared to the other two modules. Our own implementation tends to classify shirts as pullovers, which significantly affects its overall accuracy.

6.2.2 Other aspects

Our neural network had the fastest performance per epoch but required a large number of epochs to converge. The PyTorch convolutional neural network required more matrix resources for feature extraction but only needed 10 epochs to achieve good results. The decision tree module was prone to overfitting and took a long time to build for the given dataset.

6.3 Interpretation of Model Performances

The results, particularly the confusion matrices, clearly show that the convolutional neural network is the most reliable and versatile model. Despite its fair accuracy, our implementation of a neural network may potentially lack a more complex structure and finer details that could help improve its performance in certain areas. The decision tree is not suitable for image classification, as it is prone to overfitting.

Overall, in the future, it could be beneficial to try to enhance the CNN by modifying its convolutional structure. It may also be useful to create a deeper neural network than our implementation and optimize the hyperparameters further. In our opinion, the decision tree is not suitable for this dataset and task.

6.4 Conclusion

The analysis clearly demonstrates that the reference Pytorch convolutional neural network implementation was the best classifier for the given data, but our own neural network implementation also performed well. The decision tree algorithm is generally not often applied in image classification tasks, and this fact also applies to our dataset. In the future, it would be worth exploring ways to improve or extend the neural network, or to optimize the PyTorch implementation, as it allows for a lot of flexibility in network structure.

References

Bajwa, Imran et al. (Jan. 2008). "Feature Based Image Classification by using Principal Component Analysis". In: *Journal of Graphics, Vision and Image Processing (GVIP)* 09, pp. 11–17.

Xiao, Han, Kashif Rasul, and Roland Vollgraf (Sept. 2017). *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv:1708.07747 [cs, stat]. URL: `http://arxiv.org/abs/1708.07747` (visited on 12/31/2022).

Lu, Lu et al. (Nov. 2020). "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28, pp. 1671–1706. DOI: `10.4208/cicp.OA-2020-0165`.

Dutta, Sujan (Jan. 2023). *ML_from_Scratch*. original-date: 2019-03-11T07:16:40Z. URL: `https://github.com/Suji04/ML_from_Scratch/blob/e3dda5c9dadc02f1746d6d7837d6de7ad45c51b9/decision%20tree%20classification.ipynb` (visited on 01/06/2023).