Machine Learning Report

2025-01-02

Earth Vangwithayakul (eava@itu.dk), Mathias Mortensen(mathm@itu.dk) & Zen Rehda(zreh@itu.dk)

Course: Machine Learning

Course code: BSMALEA1KU

Introduction

The Fashion-MNIST dataset, a dataset consisting of 28x28 grayscale images of various articles of clothing, is a classic database for use in machine learning. From this dataset, a question can be asked: Can you, using machine learning algorithms, adequately predict which type of clothing a previously unseen image of clothing is, by training a machine learning model on other images from the same set? In our case, we are working with a subset of this database; we have 15000 images (10000 for training, 5000 for testing and validation), with the clothes split into 5 distinct categories: t-shirts/tops, trousers, pullovers, dresses and shirts.

Exploring the dataset

Looking at our training data, each picture is defined in an array as 785 values. 784 of these values each correspond to a single grayscale value between 0 and 255, whereas the last value identifies the classification of this image.

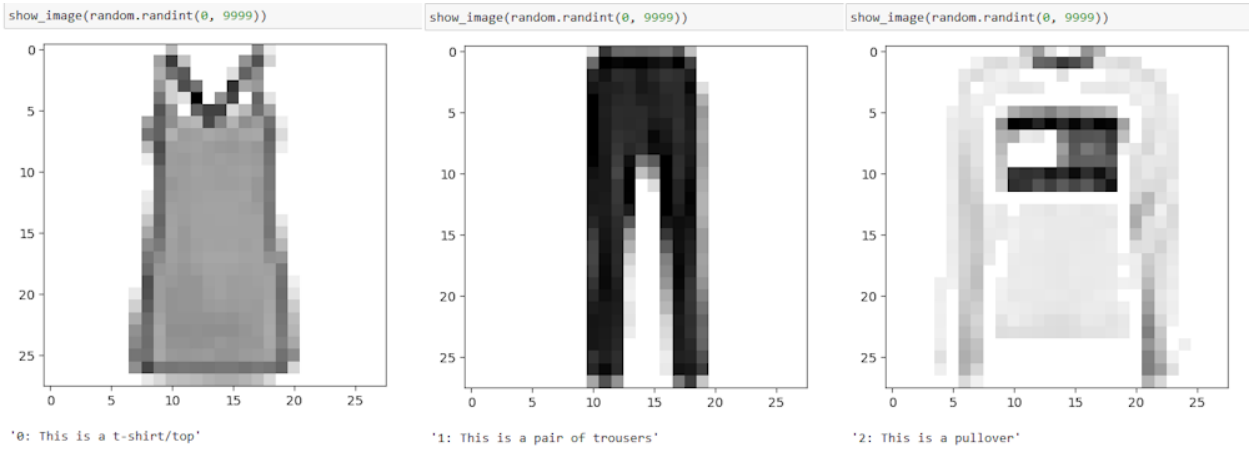


Figure 1: Each image has a corresponding categorisation identified by an integer in the range 0-4

In our training data, we have a slight but statistically insignificant imbalance between the classes.

Categories	Quantity
T-shirts/tops	2033
Trousers	1947
Pullovers	2001
Dresses	2005
Shirts	2014

To effectively work with our arrays, we have to separate the classes from the image arrays. The last values of all arrays are stored separately as our y-values, or true classes, while the other 784 values are converted into a 28x28 two dimensional numpy array, that can be both visually represented and serve as our x-values.

Data Visualisation

Our first approach to handle the data is to simplify the number of features we want to look at and train our model on. In essence, each pixel is a feature and a 784 feature model is both cumbersome and inefficient. In addition, we know from just observations that certain pixel values are clearly more favored than others,

as the images are centred. Therefore, pixels along the outer edges are less important for a potential model, and pixels elsewhere might have similar lack of importance.

To get around this issue, we first run our values through Principal Component Analysis. We transform our values into a new coordinate system and find the directions that capture the most variation and is thus most descriptive of our data. (This would be PCA feature 1 & 2). This will greatly reduce the number of features needed for our model while accurately describing a large amount of the variation in the data.

However, after converting our values into a PCA, a new issue pops up.

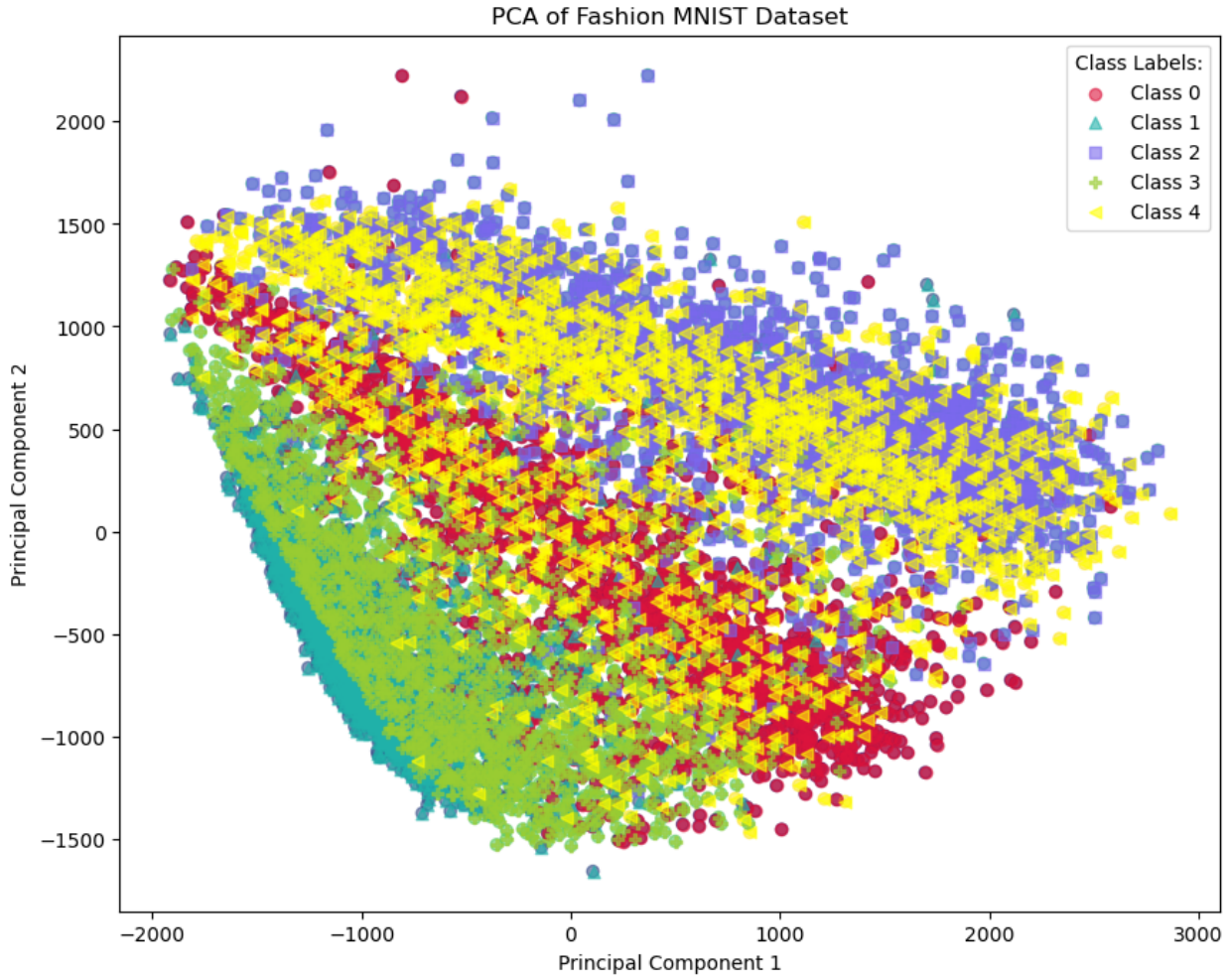
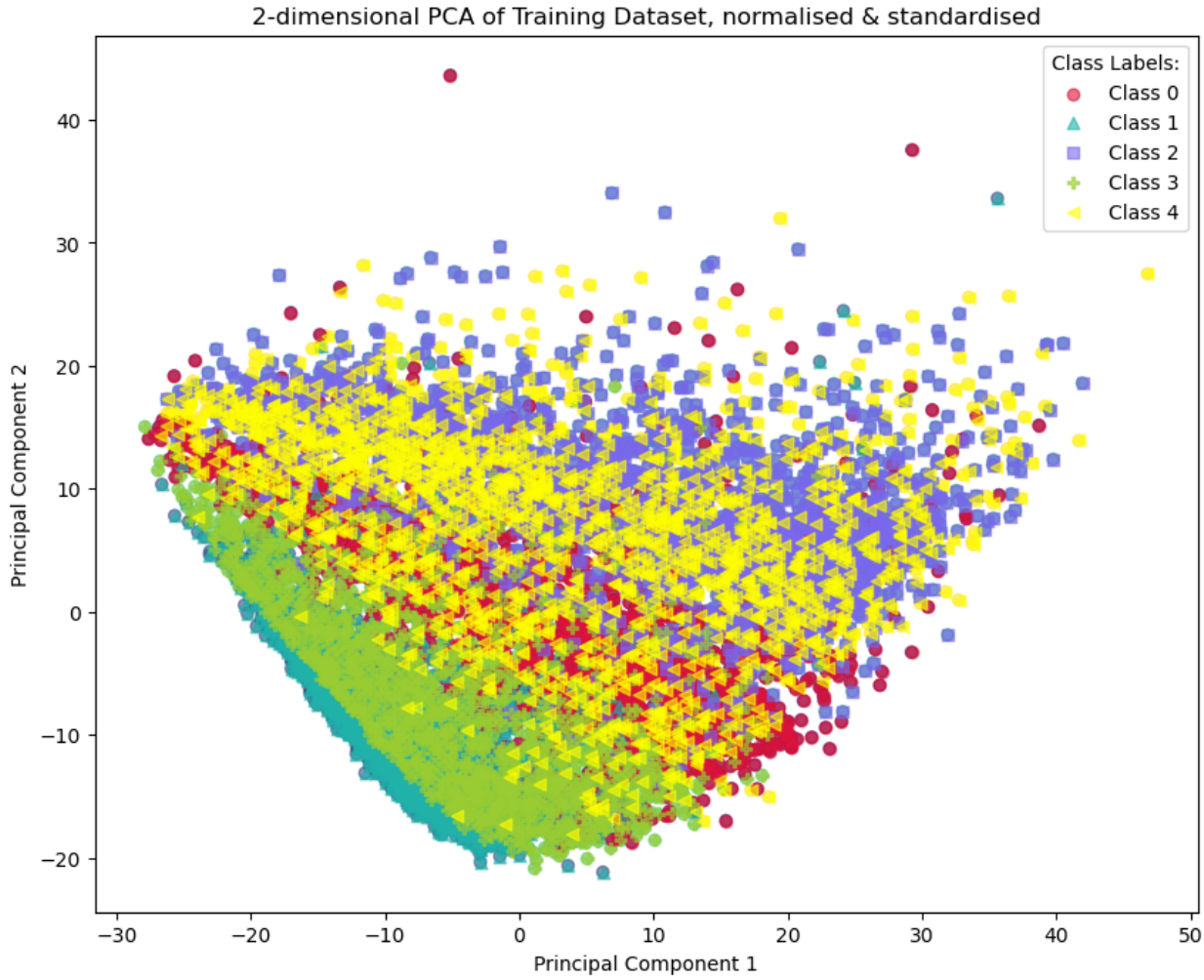


Figure 2: A 2D representation of the 2 most important PCA features.

The values in our model are quite large, ranging from -2000 to 3000. This happens as our pixel values range so widely from 0 to 255. In order to minimise distances and help our model work more efficiently, we should normalise our values to range between 0 and 1. In addition, as PCA as a model assumes a normal distribution, we should standardise the dataset. Therefore, we also apply a StandardScaler to the training data, ensuring that the mean is 0 and the standard deviation is 1.

As a result, our model values end up much more compressed and closer.



In addition, we need to figure out how many PCA features we want to include. As each PCA feature explains an amount of variation in the data, we can create a threshold where we cut off features that are insignificant. We decided to include all PCA-features up to 90% of variance, meaning we are discarding noise and less informative features. This should ensure faster running methods. This resulted in us having a PCA with 112 components.

Implementation of Machine Learning methods

As part of our project, we were to implement three methods for classification, specifically a Decision Tree, a Feed-Forward Neural Network and a third one freely chosen. The first two had to include a raw implementation without any use of python libraries beyond the standard libraries.

In our comparisons between the models, we will primarily look at the accuracy of the methods. We feel this is adequate, as we are dealing with classification issues, and the primary concern should be the methods ability to correctly classify a piece of clothing.

M1 - Decision Tree First we wanted to create a Decision Tree. This method is all about segmenting the predictor space into regions. As a method it is simple and easy to interpret, but decision trees "*are not competitive with the best supervised learning approaches*"(James et al., 2023). Therefore, we expect a middling accuracy, compared to later models.

As a reference for our raw Decision Tree implementation, we used Chris Jakuc's "Building a Decision Tree Classifier" from 2020.

For creating our decision tree, we create separate functions for fitting, finding splits, growing the tree and predicting

M2 - Feed-Forward Neural Network (FFNN) After implementing our raw neural network, we turned to a library implementation. For this, we decided on using PyTorch, a deep learning library for Python.

M3 - Convolution Neural Network (CNN) As part of our project, we decided to create a CNN using PyTorch as our implementation language. CNN is known to be a reliable image processing method, and unlike our other methods, the CNN method will by itself extract relevant features. Therefore, it would not handle the PCA data, but rather be fed the raw pixel arrays.

As a reference for our implementation of CNN, we used the official PyTorch implementation guide by Soumith Chintala. [3]

For our convolution layers, we convolve our images twice and we pool them twice. While adding more could be feasible, with a 28x28 image, after pooling twice, our images will reach a 7x7 resolution. Any further convolution would introduce some manner of data loss, as we would be left with a 3x3 image. After the convolution layer, we flatten our images into vectors, and we reduce the features until we eventually reach the 5 classes.

For our loss function, we decided on using cross-entropy, which is a standard for use in classification problems. In addition, in PyTorch's implementation of the CrossEntropyLoss(), softmax (giving us probabilities) is automatically applied to the logits and is therefore not added to our layers itself. (Tam, 2023)

In our tests, using either Stochastic Gradient Descent (SGD) or Adaptive Moment Estimation (Adam) as our optimizer, we saw little difference in accuracy between the two, ending with a validation and test accuracy within 1 pp of 87%. In the final implementation, we used SGD.

The learning rate set for the method ended up at 0.001. Tests showed that 0.01 left the model unable to find any local minima and get stuck at abysmal accuracies, while a rate at 0.0001 returned a similar final accuracy to 0.001, albeit requiring twice the number of epochs.

In our implementation, we included a loss threshold of 0.1 with a patience filter of 5. In other words, once the loss had dipped below 0.1 for five epochs, the training was complete.

Results & Discussion

Overall, we could see a gradual increase in accuracies as the complexity of our method increased.

One of our most successful approaches, the CNN, had almost perfect accuracy when it came to predicting pants and dresses in the dataset, but encountered issues with clothing that could appear similar, such as shirts

Bibliography

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, Jonathan Taylor. 2023. *An Introduction to Statistical Learning with Applications in Python.* Springer.

Soumith Chintala. 2023. *Deep Learning with PyTorch: A 60 Minute Blitz.* PyTorch. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html.

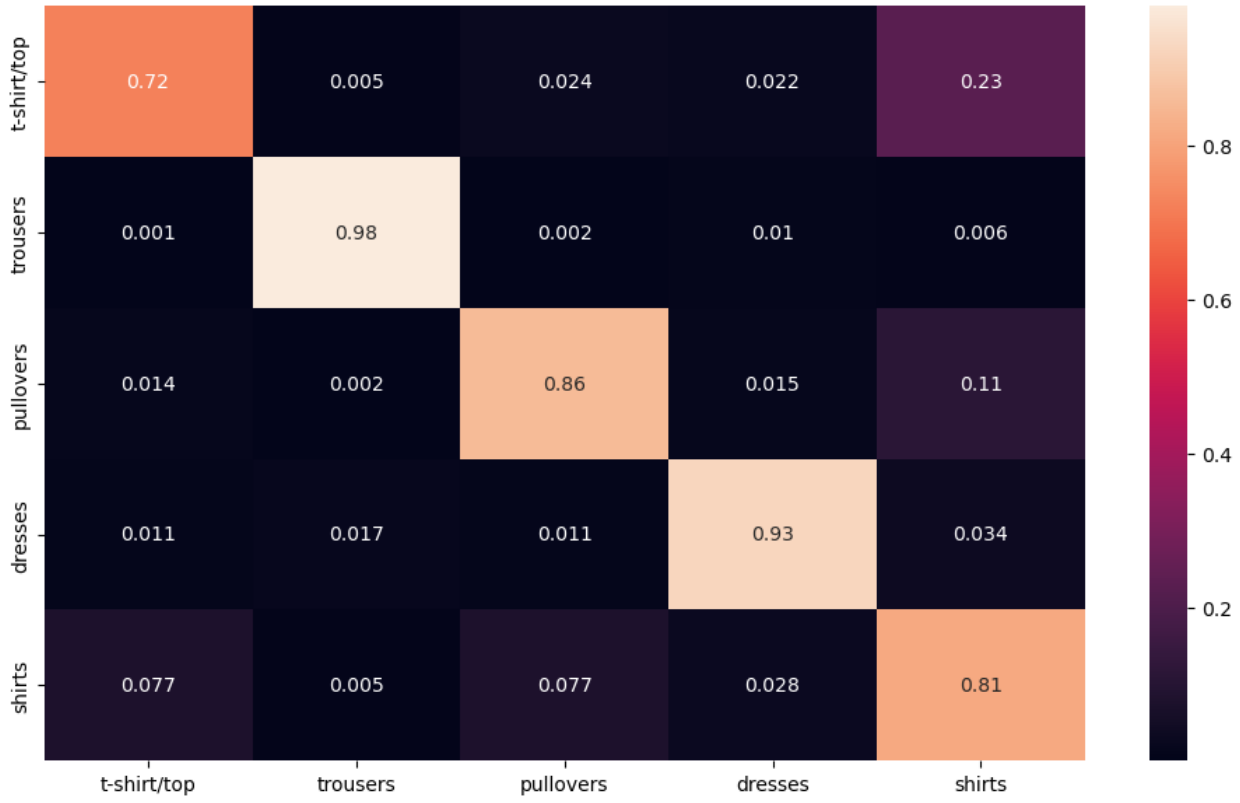


Figure 3: Final confusion matrix of CNN (Code sourced to Bernecker 2021)