Machine Learning (Autumn 2024)

BSMALEA1KU

Elias Fischer Hegelund
elhe@itu.dk

Kristoffer Schmidt
krisc@itu.dk

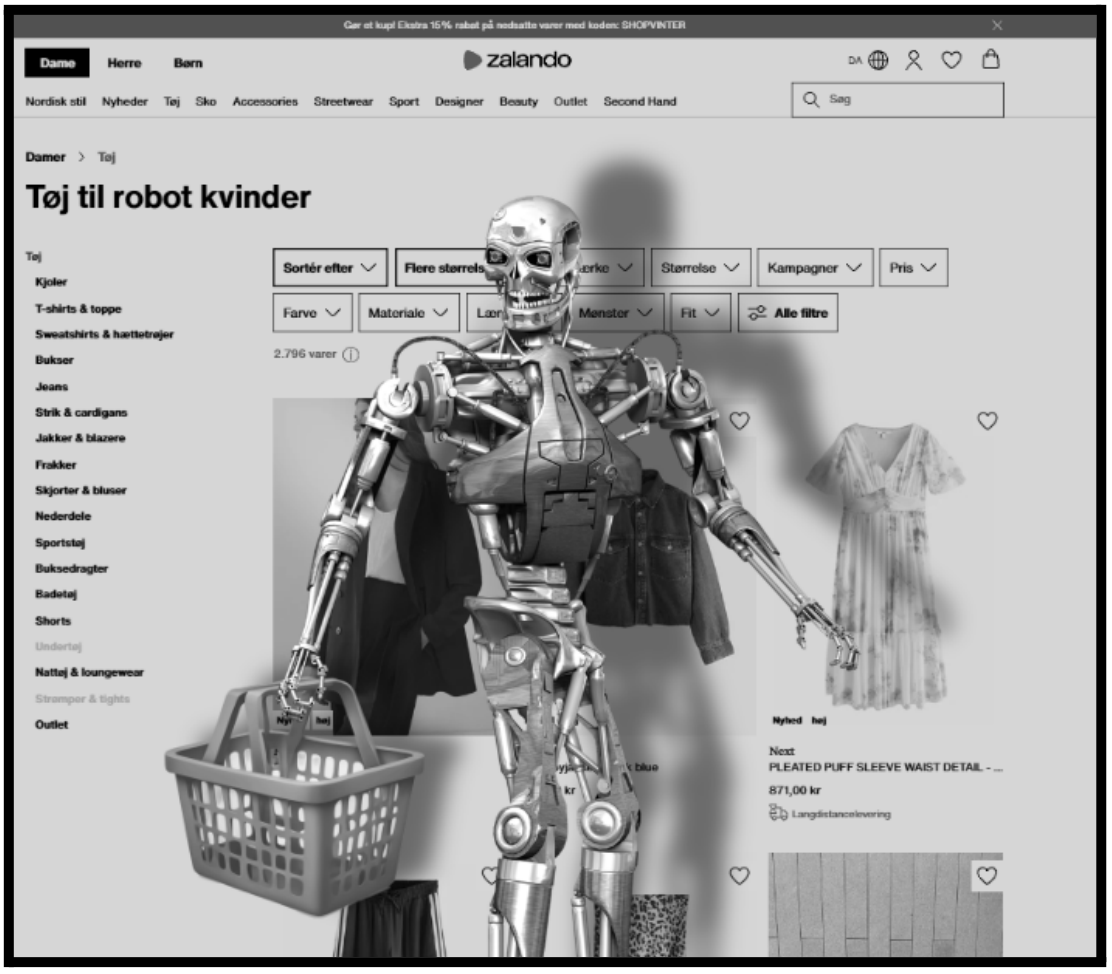

Illustration representing the use of AI in classifying Zalando clothing.[5]

Abstract:

This report investigates the classification of 28×28 grayscale images of clothing items from the Fashion-MNIST dataset into five categories: t-shirts/tops, trousers, pullovers, dresses, and shirts. Using Decision Trees, Feedforward Neural Networks, and Convolutional Neural Networks (CNNs), we compare the performance of these models in handling challenges such as distinguishing visually similar categories like shirts and pullovers. The CNN achieves the highest accuracy of 90.64%, benefiting from its ability to capture spatial patterns. Heatmaps and PCA visualizations further illuminate the dataset's structure, providing insights into model performance and category overlap.

Table of contents:

1. Introduction

Classifying clothing images is a practical problem, especially for applications like online shopping. We're working with the Fashion-MNIST dataset, which includes grayscale images of five clothing types: t-shirts/tops, trousers, pullovers, dresses, and shirts.

This report explores three machine learning models—Decision Trees, Neural Networks, and CNNs—to compare their performance. Some categories, like shirts and pullovers, are particularly challenging because they look quite similar, making the classification task more interesting.

2. Exploratory data analysis

The follow chapter contains information about and visualisations of the images in the dataset

2.1 The dataset

In this dataset we have 28x28 grayscale images of different categories of clothes, taken from the Zalando retail website, provided by Xiao, H., et. al., 2017
The images are categorized into five different types of clothing, t-shirt/top, trousers, pullovers, dresses and shirts.
In our dataset we have 10k images we use to train our models with, which are stored as npy, and we have 5k images used for testing the models. We have exactly 1k of each category, so the classes are uniformly distributed.

2.2 Heat map of pixel location in each category

To get a better idea of what each category represents, as well as the general layout of the pixels, we made heatmaps of the clothes in each category. The heatmap works by counting the amount of images that have a non 0 pixel in each location. The heatmap is then generated with the value 255 (white) for a pixel if all the images in the category have a pixel and 0 (black) if pixels are never present at that location. If pixels are only sometimes present a gray pixel is then made as the fraction that is present (0-1) times 255.

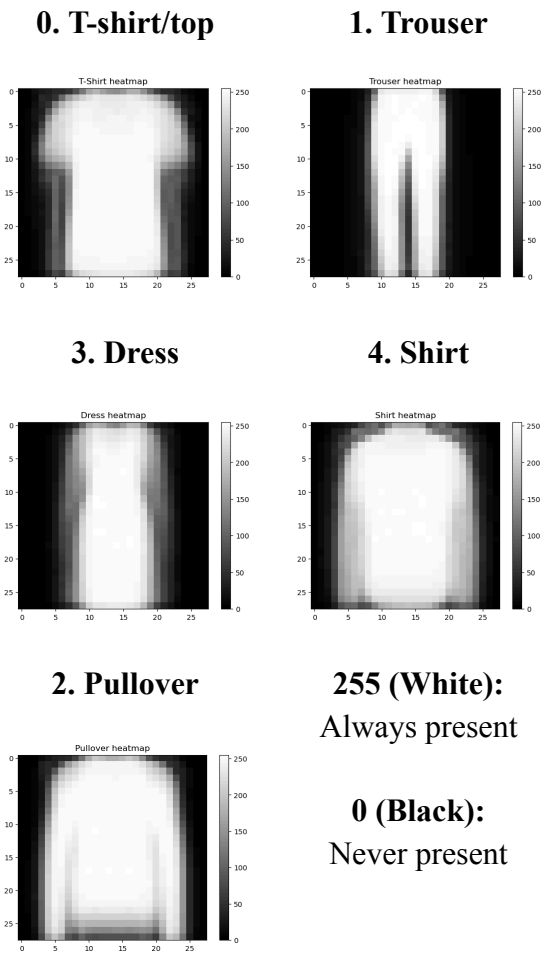

Figure 1: Clothing category heatmaps

We can see that some of the images from the heatmap look similar, especially the pullover and the shirt. There is a difference in the color of the arms, the arms in the shirt are more gray, while in the pullover they are perfectly white, which points us to the information that pullovers always have sleeves while shirts sometimes do not, but then they might get confused with the t-shirt. From this we might hypothesize that our model will have a harder time differentiating between Pullovers and Shirts than other categories, it will be interesting to discover later if that is true.

3. Visualization of data.

3.1 Area/circumference visualization

To get a general idea of how easy it would be to distinguish the different clothes types from each other, we decided to plot all the images/their class in a 2 dimensional plot. What we want the axis to represent is of course highly arbitrary, but since we wanted two axis that could express more than just pixel values at some specific location (which is the base features) we decided to calculate the "area" as the sum of all the values in the picture, and use convolution to find the edges, and thereby the circumference.

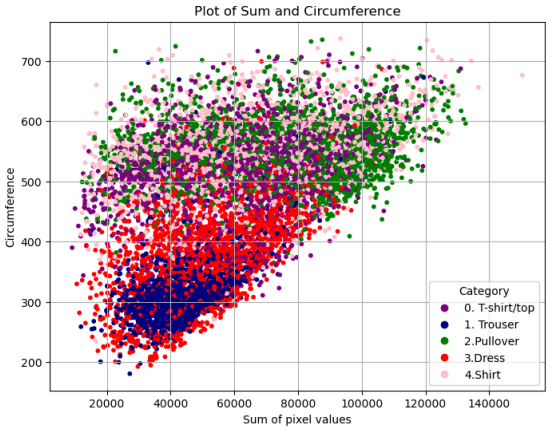

Figure 2: Sum of pixels vs circumference

In the plot there is a noticeable separation between the classes, it is minor, and there is a lot of overlap, but we can definitely conclude that there is a difference. As previously hypothesised we can see that there is a significant overlap between shirts and pullovers, and also with shirts and T-shirts/tops, which makes sense, since as we saw earlier, they are very similar products. We expect the long sleeved shirts have the overlap with pullovers, and the short sleeved shirts with the T-shirts/top category.

3.2 PCA visualization

A PCA plot displays a linear combination of features that creates the maximum variance. Below we see two axes PC1 and PC2 that try to create the most variance by a linear combination of features, while being orthogonal to each other. In this case we see that in the PCA feature space we get a lot of clear separability for most of our classes. For instance, "Trouser" and "Dress" show a slight overlap but remain relatively well separated in the feature space. In contrast, "Shirt" exhibits considerable overlap with both "T-shirt/Top" and "Pullover." This overlap is understandable: a "Shirt" with long sleeves can resemble a "Pullover," as indicated by our heatmaps, which also show that some shirts have long sleeves. Similarly, the overlap between "Shirt" and "T-shirt" can be explained by the fact that both types of clothing often share similar attributes, making this overlap self-explanatory.

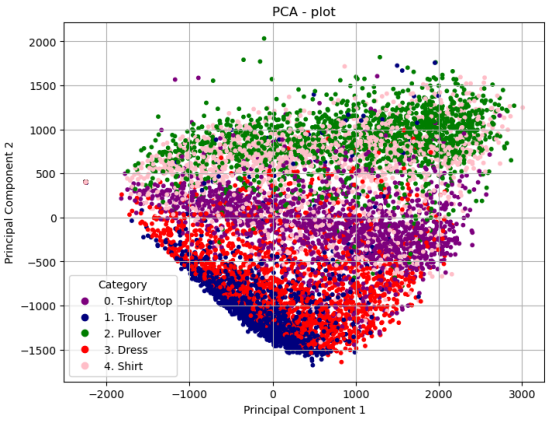

Figure 3: PCA plot with class visualisation

4. Details on implementations.

This chapter examines the implementation and evaluation of three classifiers: M1, a Decision Tree; M2, a Feedforward Neural Network; and M3, a Convolutional Neural Network (CNN). M1 and M2 were implemented from scratch, focusing on their design and training processes, while M3 utilized the TensorFlow library to streamline development with GPU support. The methods are compared based on their implementation approaches and performance outcomes.

4.1 Classifier M1. Decision tree

Our first classifier to handle the problem is a decision tree.

4.1.1 Decision tree - General information and terminology

A finished decision tree is a flowchart-like structure consisting of three main components: the root, decision nodes, and leaves. These terms are derived from tree data structures. In a decision tree, the root acts as the starting point and poses an initial "question" based on a feature. To classify an object, you begin at the root, and depending on whether or not the object says yes or no to the “question”, you then go to either the left or the right decision node. This process repeats with subsequent decision nodes, directing the object through the tree until it reaches a leaf. A leaf represents the final prediction, either as a class label (classification) or a numerical value (regression).

Unlike many other classifiers, decision trees are based on a greedy algorithm, which means that when you “train” them they do what's best at any given point, with no consideration for what might have been beneficial in the future.

4.1.2 How our decision tree is trained

Our decision tree is trained using two functions: find_best_split and split_data. The function split_data is the main function responsible for generating the structure of the tree. It takes two inputs: “data” and “level”. The “data” input is a dataframe, first time calling the function you feed it the entire training-data dataframe. The function is then responsible for checking whether the input-data is “pure” and if it is, then it makes a leaf node, if the data is not pure, it sends the data to the find_best_split function.

The find_best_split function then goes through each attribute, so the 784 pixel value columns, and for each column it then finds all the distinct values, and makes a list of the labels, sorted by the order in the attribute column. A running count of the category distributions, above or below each distinct attribute value, is then made by iterating through the list, in the order of the feature, and the weighted gini impurity is calculated for each split. After all splits have been evaluated the one with the lowest impurity gets sent back to the split_data function. The split_data function then takes its dataframe and splits it into two different ones in accordance with the criteria found, and then it recursively calls itself two times with the two new splits, this process generates a binary tree doubling its size at each level, the function then stops calling it self when the stated level or “depth” is reached. Our decision tree is stopped after 10 levels, so with hyperparameter depth=10. It only gains about 5% point accuracy increase from depth 6, and all the following depths are of negligible performance improvement.

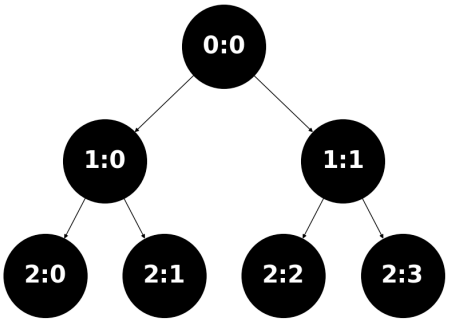

Figure 4: Illustration of perfect binary tree using our implemented node naming scheme.

Note: For any node X:Y, X is the layer the node is in. Y is the index in the layer.

$$node\ traversal\colon (X_{i+1}, Y_{i+1}) = (X_i + 1, (2 * Y_i) \vee (2 * Y_i + 1))$$

4.1.3 How our decision tree is stored and functions

The finished structure is then stored as a list of lists of lists. The first list simply holds k lists corresponding to each of the k layers of the tree. Those lists then store the 2^k nodes of that layer, so 2^0=1, 2^1=2, 2^2=4 and so forth. Visual representation can be seen in the above diagram. Each node is then stored as a list on the form:

(0.7997732, 546, 13, array([35, 1658, 31, 18, 40]), array([1998, 289, 1970, 1987, 1974]))

The first value represents the weighted gini impurity of the node, the second is the

column/feature of the “question”, in this example the column is 546, the third number is then the threshold value for whether to follow the left or the right branch to the next node/leaf. The two arrays then show the resulting class distribution of the left and right split. This is a bit redundant information and is not really necessary to store, but it makes it a lot easier to follow along manually, when error checking and troubleshooting. It could instead just have been a label for the majority class.

The leaf nodes are represented as: ('leaf', 1). The prediction function then knows that when the gini value is ‘leaf’ it should take the second value as the predicted class and then stop following the tree. Since we have chosen to store the tree as a perfect binary tree stored as lists, we still need to have values on the nodes after the leaves, for the structure to be full. Therefore we have filler nodes called “deadleaf” that are only there to fill the structure, these can never be reached by the predictor function. They could have been avoided by storing each layer as a dictionary with the key being the “location”, this would be more efficient but we did not implement that.

If the tree ends before a leaf node has been found, as in we reach a decision node on the last layer, we simply classify the object as the majority class on the left or right array, depending on the threshold. This can also be weighted if we want to prioritise one or more classes. Many other decision trees also have some criteria for earlier creation of leaves, like reaching a certain purity in the node, or having few samples, like <5, reaching the node. Since we choose to store additional information in the nodes, this can be changed in the prediction function without the need to retrain.

4.1.4 Results of the decision tree on test data

Accuracy: 79.46%
Class accuracy: [0.748, 0.92, 0.839, 0.87, 0.598]
Confusion Matrix:

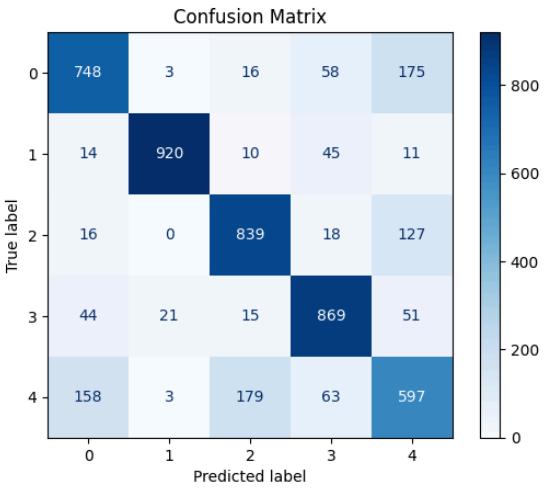

Figure 5: Confusion matrix for decision tree

We can see here that the decision tree does a pretty good job at classifying the data, with a 79% prediction accuracy. This is good, but still pretty far from perfect. As suspected earlier the class the model is worst at classifying correctly is class 4, the shirt. We can see that only about half of the shirts are classified correctly, and the two remaining quarters of the shirts are classified as t-shirts and pullovers, this is again most likely due to the sleeve problem. But maybe the neural network will find a way to deal with this? Let's find out.

4.2 Classifier M2. Feed Forward Neural Network

4.2.1 General Information and Terminology

A feedforward neural network operates by taking input values at the first layer, which is run through multiple layers of the network, in order to try and make predictions of the output values.

This happens through two key processes: the forward pass and backpropagation.

The forward pass works by taking input values from the previous layer, multiplying them by weights associated with connections to the next layer, and adding a bias term. The resulting values are then passed through an activation

function, such as sigmoid or ReLU, to introduce non-linearity. This process is repeated across all layers until the final output layer is reached. At the output layer, the results are processed through a softmax activation function to produce a probability distribution, representing the model's confidence in each class. This distribution is compared to the desired output using a loss function. In this case CrossEntropy Loss, which is well-suited for multi-class classification tasks.

Backpropagation begins after the loss is computed. Using gradient descent, the model adjusts its weights and biases by propagating the errors backward through the network. The adjustments depend on the gradients of the loss with respect to the weights and biases, scaled by a learning rate. This process can be represented mathematically as:

$$W^{(t+1)} = W^{(t)} - \alpha * \nabla L(W^{(t)})$$

where $W^{(t)}$ represents the weights at time step t, α is the learning rate, and $\nabla L(W^{(t)})$ is the gradient of the loss function with respect to the weights.

The model iterates through this process of forward and backward propagation over multiple epochs or until a predefined stopping condition, such as convergence or achieving a desired accuracy level, is met.

4.2.2 How Our Neural Network Works

Our feed forward neural network uses a function called "feedforward", which handles the training process. This function takes several arguments, including the activation function, the number of hidden layers, and the learning rate. Based on which activation function was chosen, the model automatically uses a preset combination of functions for the cost function, activation function, and weight initialization. For example we use He weight initialisation for relu type activation functions, and Glorot/Xavier for sigmoid. These initialization methods help prevent vanishing and exploding gradients. And all biases are initially set to zero.

The feedforward function performs the following steps:

1. Forward pass: Compute the pre-activation function inputs and post-activation outputs for each layer, returning these values as lists.

2. Backpropagation: For optimization of the weights and biases we iteratively perform forward passes and backpropagation to update weights and biases, minimizing the loss. The outputs, inputs, weights, and biases from the previous pass are used as inputs for the next iteration.

3. Optimization: After running a specified number of epochs, the function returns the optimized weights, biases, the number of hidden layers, and the activation function used. This information is then ready for use in predictions.

4. Prediction: The prediction function takes the weights, biases, and other outputs from the feedforward function to perform one final forward pass. The results are stored as predictions, where the maximum probability for each data point is saved as the predicted class.

5. Optimization algorithm: The neural network uses standard batch gradient descent as its optimization algorithm. The pros and cons of this is that, even though it’s slow, it steadily converges to a minimum, with few problems. A potential issue though is that this works best when we have a convex function, and for nonconvex functions it could get stuck in a local minimum.

4.2.3 Results of the Neural Network on test data

Accuracy: 83.26%%
Class accuracy:
[0.869, 0.958, 0.892, 0.893, 0.551]

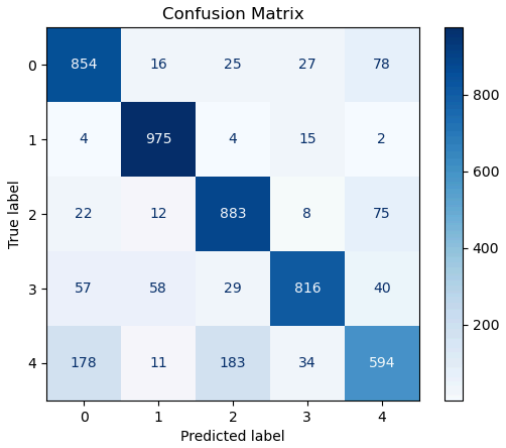

Figure 6: Confusion matrix for feed forward neural network

We see that the neural network performs slightly better than the decision tree accuracy wise, though not significantly better. For the class accuracies we see that the neural network also performs very similarly. On average for multiple runs we do observe an increase in accuracy of about 3-4%.
This indicates that the neural network still struggles with the same issues as the decision tree, like performing significantly worse on class 4 (“Shirt”) compared to other classes.

4.3 Classifier M3. Convolutional Neural Network

For the M3 classifier we wanted to test a Convolutional Neural Network, to do this we chose to implement it using the tensorflow python library. Convolutional Neural Networks (CNNs) are specifically designed for image data, making them ideal for this dataset. The core advantage of CNNs lies in their ability to process spatial information. Unlike a standard feedforward network, where all pixels are treated independently, CNNs use convolutional layers to focus on local patterns, like edges or curves, which are essential for understanding shapes in clothing images.

Another benefit of using the tensorflow library is the ability to use GPU compute with NVIDIA cuda and cudnn, this allowed us to utilize our RTX 3080 and train the model much faster than the previous models [2.2]. The benefit of this is that the time required to test each iteration of parameters was much lower, because of this we were able to test larger and more complex models for increased epochs. However this did not improve the test results, but only “overfitted” to near-perfect train accuracy, so we stuck with a more simple CNN. We chose the "Nadam" optimizer, which combines the Adam algorithm with Nesterov momentum. Adam was selected for its ease of use, as it requires minimal tuning of the learning rate, while the addition of momentum aims to accelerate convergence. (Géron, A. (2019) [4])

Another strength of CNNs is their deep architecture, which allows them to learn increasingly abstract features as we go deeper in the network. Early layers might identify basic patterns like edges, while later layers can recognize more complex structures, such as the overall shape of a dress or the texture of a pullover. This makes CNNs particularly effective for tasks where small differences, like sleeve length or neckline shape, are crucial for classification.

4.3.1 Results of the CNN on test data

Accuracy: 90.64%
Class accuracy:
[0.865, 0.99, 0.912, 0.945, 0.82]

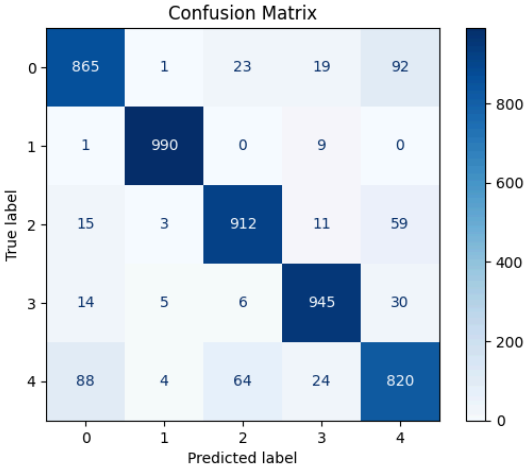

Figure 7: Confusion matrix for convolutional neural network

As we can see these numbers are noticeably better than both the results from the two previous classiers, and it does a better job at distinguishing shirts from the other categories. It is however still far from perfect, which is why we hypothesise that there might be irreducible error from the great overlap between the long or short sleeved shirts with the other classes, this is also supported by the fact that scaling up the model did not improve results. We managed to find people on the internet claiming a higher overall accuracy on the Fashion MNIST dataset, but that was with the full dataset that has both more pictures, and more classes. While intuitively more classes should reduce the accuracy, these classes are items such as shoes and bags, which seem to be very different from clothes, and therefore positively benefit the overall accuracy by being more dissimilar. We are satisfied with the results of our models

5. Interpretation and discussion of the results and models

5.1 Discussion of results

From our above models we got the following accuracies:
Decision Tree: 79.46%
Feed forward Neural Network (FNN): 83.26%
Convolutional Neural Network (CNN): 90.6%

For the implementation of the decision tree we managed to achieve roughly 79.5% accuracy, this is of course far from perfect, but still quite good, given that decision trees operate on a greedy algorithm that splits the data without regard for future splitting. Also decision trees are unable to capture more complex non-linear relationships, this is where the FNN is able to improve on the decision tree. Our FNN managed to achieve an accuracy of 83.26%, this improvement is likely due to the FNN's ability to look at more complex relationships and patterns between features and introducing non-linearity. The CNN further improved upon this with a staggering increase of almost 8 percent points, by utilising spatial information, with a final result reaching a whooping 90.6% accuracy on the unseen test data.

Ideally we of course would have hoped to achieve an even higher accuracy score of 95% or above. We tried to tune the parameters, for the model, both with which optimizer to use, the amount of layers, the size of the convolutions, and some combinations thereof, etc. We also tried utilising early stopping and data augmentation to better generalize the model and prevent overfitting, though this did not improve the results either. And even when the model was able to overfit to a degree of memorizing 100% train data accuracy, we never got above 91% in the test.

As already mentioned, the significant challenge encountered was the high similarity between shirts and pullovers, as highlighted in the heatmaps and confusion matrices. This overlap led to a more substantial misclassification rate for these categories, indicating that a portion of the misclassification might stem from irreducible error, making it difficult for any model to achieve perfect accuracy.

5.2 Comparison of M1 and M2 with reference models

M1. Decision tree: To further test the correctness of our implementation, we wanted to know if our results were consistent with what libraries like scikit-learn would get. With scikit-learn we achieved a similar accuracy score for the same hyperparameters (depth = 10, min sample split=0, min sample size=1) of 79.38%, which is pretty much identical, so we can conclude that our implementation of the decision tree works as intended.

M2. Feed-Forward Neural Network: We again compared our performance with the scikit-learn MLPclassifier, which with the same parameters yielded very similar results of 82.72% accuracy. Thus, we can conclude our implementation of the

FNN to be correct. However when training our FNN with more than 2 layers, the loss function only ever increases instead of decreasing. We are unsure on what the underlying issue resulting in this behavior is, and have not had the time to correct this mistake. This trend is not seen in the MLPclassifier, indicating that our model does have some bugs in the implementation of the math for the hidden layers, but it does not appear to be an issue when only having 2 or less hidden layers.

6. Conclusion

This project compared three models: M1 Decision Tree, M2 Feedforward Neural Network, and M3 Convolutional Neural Network for classifying clothing images from a subset of the Fashion-MNIST dataset. The Decision Tree provided a simple baseline with 79.46% accuracy, while the Feedforward Neural Network improved to 83.26%, leveraging its capacity for learning non-linear patterns. The CNN outperformed both, achieving the highest accuracy, of 90.6%, by effectively capturing spatial information.

While all models succeeded to some extent, overlapping visual features in certain categories posed some challenges. The CNN proved most suited for image data, highlighting its strength in such tasks. Future work could explore advanced techniques like ensembling to further improve performance or perhaps use some entirely different models.

7. Experimental reproducibility

7.1 Github

All the code for our project can be found at the following Github repository: Fashion_Classifier Repository

7.2 Required software and versions

For reproducibility the package versions can be seen below:

cudatoolkit = 11.2
cudnn = 8.1.0
tensorflow = 2.10
Python = 3.10
Other necessary python packages can be found on the github.

7.3 Hardware specifications

All models where run in visual studio code on a windows 10 pro desktop with the following specifications:
8-Core Intel i7-11700
10 GB Nvidia RTX 3080
32 GB 2400MHz DDR4 RAM

8. References

[1] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms.

[2] TensorFlow:

- [2.1] Guide to Keras Sequential Model. Available at: https://www.tensorflow.org/guide/keras/sequential_model
- [2.2] Install TensorFlow from Source (GPU). Available at: https://www.tensorflow.org/install/source#gpu

[3] James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). An Introduction to Statistical Learning with Python.

[4] Géron, A. (2019). Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow (2nd ed.).

[5] Front page robot image, public domain at: https://cdn12.picryl.com/photo/2016/12/31/robot-mechanical-futuristic-science-technology-f41e25-1024.png