Image Classification of the Fashion MNIST Dataset

A machine learning project report

Course code: BSMALEA1KU

Gabriel Cristian Circiu `gaci@itu.dk`
Maciej Pawel Jalocha `macja@itu.dk`
Wenzel Keil `weke@itu.dk`

January 6, 2025

Contents

Introduction

We are tasked with investigating the automatic classification of clothing product images to predict their respective type. The goal is to determine which type of clothing product any given image is, based on several models trained on already existing images of different kinds of clothes. While we can observe, with our human eyes, differences in the shapes and sizes of these images, for machines it is not as natural, and so it is imperative that we approach this task with not only a human perspective but that of a statistical one as well, that will aid us in programmatically evaluate both high- and low-level features. There are, however, obstacles. While we are given thousands of images to train the models on, this is not enough to achieve results that can always classify correctly, and with the rise of high-quality images, storing more than a few thousand images will hold challenges of its own.

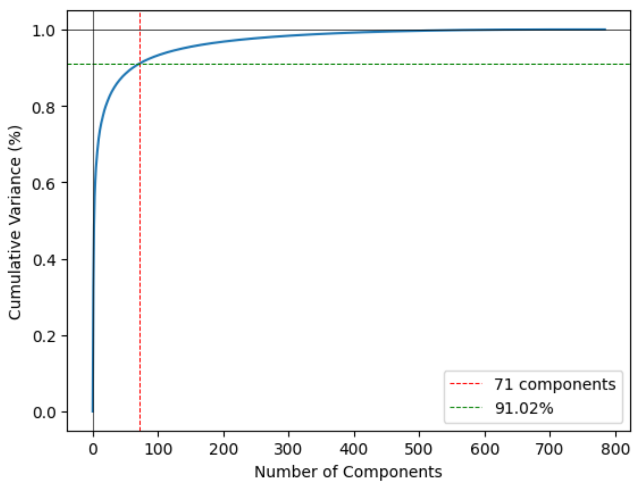
The Fashion MNIST dataset

The dataset we base our work on, is a subset of the of the 2017 project from Zalando Research, consisting of 70,000 images of different clothing items. Each image is a 28x28 grayscale image, associated with a label from 10 classes. In our project, we are using a subset of only 15,000 of the images, associated with a label from 5 classes. The dataset is then split up into 10,000 images for training and 5,000 for testing the correctness of our models.

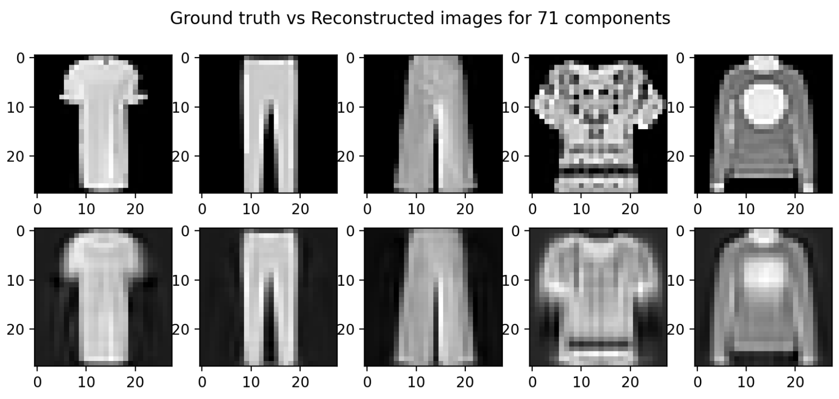
Our initial approach to the task is to perform a thorough exploratory data analysis (EDA) to better understand the data that we are working with. We have observed that there is an even distribution of classes, the images are taken from the same angle, and they are all laid out the same way for their respective class. Using this information, we could already build a template image, gathered from the averages of all the images within each class, to see if there are any outliers, as further shown in the Template Matching section. We have also observed some discrepancy with the contrast and brightness of images. These were corrected for, by naively standardizing the image luminosities by subtracting the mean and dividing by the standard deviation of the image.

Methodology

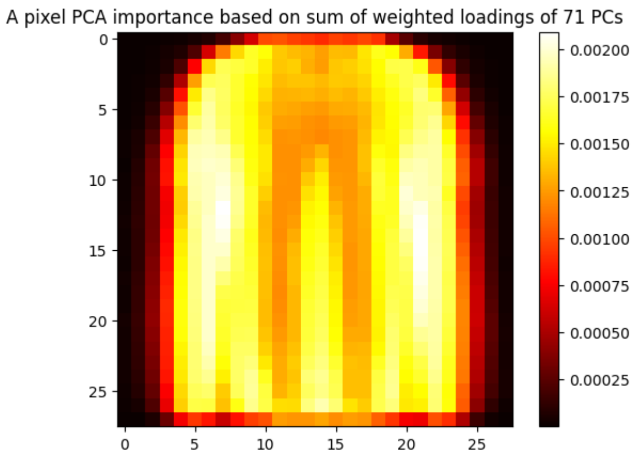
The next step was to extract features for the machine learning models. We started with Principal Component Analysis (PCA), which is a dimensionality reduction method. PCA finds the axes (principal components) that capture most patterns, so that other axes might be dropped without much information loss. We chose 71 components, which is just 10% of the original dataset, however, explains 91% of the variance in total. We picked it because the point ($\frac{71}{784}$; 0.92) was the closest to the point (0; 1), which represents the perfect state, i.e., all patterns captured by no component; the similar approach is used when picking a threshold based on ROC curve. Interestingly, this resembles the well-known Pareto Principle.



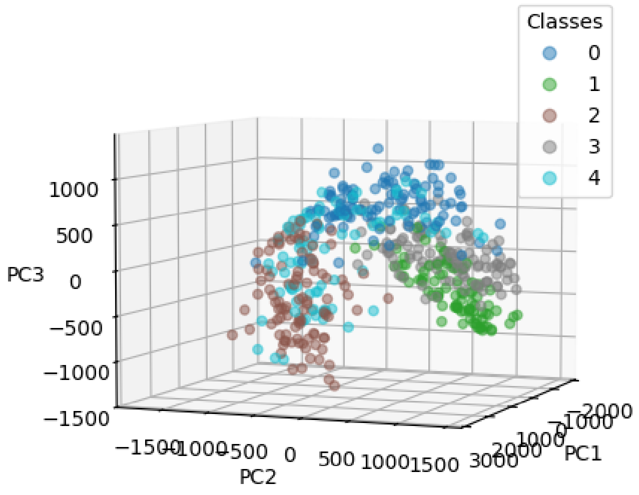
The choice of the number of components was confirmed by informal visual inspection. Indeed, reconstructions seem to retain most features allowing for successful discrimination.



The two last examples might seem disturbing as their distinct shapes are not recovered. In practice it was not a problem to worry about, because the most important pixels according to PCA lie on the left or right border of a piece of clothing. In fact, it suggests that the first 71 components are more immune to overfitting as they are not able to reconstruct irrelevant patterns.



Further visual exploration of the coordinates based on the first three components, which explain over 50% of the total variance (32%, 16%, 8%), already allows one to see some clustering.



Except for the final 'Shirt' class, the classes seem to have distinct clusters.

Alongside that, we extracted other high-level features as well, such as the circumferences, widths, and heights of the four quadrants of an image, including their variances, averages, minima and maxima. This resulted in a total of 45 more features. Finally, we have generated a template from the dataset, as mentioned before using the average pixel luminosity values of the images, and then evaluating how closely each image matches the templates, measuring the Euclidian distances of the pixels from the template created. This gave us 5 more features to work with in the end, all adding up to a total of 121 features. All features were stored in .csv files to be easily accessible for later.

Development of machine learning models

With all the features prepared, it was time to develop all required and some optional machine learning models. We will categorize these into three groups, namely M1, M2, and M3. M1 is the implementation of two Decision Tree models, one from scratch and one using the Scikitlearn library. M2 is the implementation of two Feed Forward Neural Network models, once again one done from scratch, and one using the Tensorflow library. Finally, M3 is the implementation of a Random Forest model using the Scikitlearn library, a Convolutional Neural Network using the Tensorflow library, and a custom-designed model built from scratch, which is derived from a template matching method. For each model, we will be using the entirety of our extracted feature data excluding the five template features, as that will allow us to get the best results.

Before moving forward with developing and training our models, we split our saved feature data into both training and validation subsets, first by simply using the train_test_split method within the Scikitlearn library and then later switching it out for k-fold cross-validation using the randomized GroupKFold method from within the Scikitlearn library. Our data set was divided into 80% training and 20% validation for both cases. This will help give us a good estimate of how well our models will perform overall once we move to using the entire training dataset.

M1: Decision Trees (DT)

The Decision Tree is a supervised machine learning method for classification or regression problems, which is used for predictions based on observations. In our case, we will focus on classification. It uses if-else structures, each focusing on a single class, to define boundaries between the data points, creating distinct, non-overlapping regions within the feature space. All separation is done regarding the feature value of a data point being above or below a threshold. To further the detail of the boundaries, we run the separation recursively, akin to a nested if-else structure.

Training the Decision Tree starts by checking whether any of our stopping conditions are met. These include:

1. Reaching the maximum depth.
2. Having only one class represented in the current node.
3. Having fewer than or equal to a set number of samples remaining in the node.

If none of these conditions are met, we proceed to find the optimal split point using the **Gini Index**. This metric was chosen because it is the default impurity measure in Scikitlearn's implementation, making our model more comparable to the library version.

The Gini Index quantifies the impurity of a dataset and is defined as:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

where p_i represents the proportion of samples belonging to class i in the dataset.

The Gini Index ranges between 0 and 0.5 where a Gini Index of 0 indicates a completely pure node (all samples belong to one class) and a Gini Index of 0.5 indicates maximum impurity (an even distribution of classes).

To determine the best split, we iterate over all features and evaluate every possible threshold for each feature. This exhaustive search identifies the split point that minimizes the Gini Index, thus reducing impurity as much as possible. However, this brute-force approach can be computationally expensive, contributing to slower execution times compared to optimized library implementations. After identifying the optimal split, the process is repeated recursively on the resulting child nodes. This continues until one of the stopping conditions is satisfied, at which point the node becomes a leaf. The prediction process simply follows the constructed tree until it reaches a leaf at the end with a class annotated to it.

For the *scratch implementation* in python, we created a custom class to construct the decision tree. The tree was designed with a maximum depth of 10 and a minimum sample split of 7, as we have observed only marginal gains going beyond that at best, and at worst, we would start seeing overfitting.

For the *library implementation*, we utilized the `Scikit-learn` library, specifically the `DecisionTreeClassifier` method. Regarding hyperparameters, we set `max_depth=10`, which limits the tree to a maximum of 10 levels of depth. As before, going beyond that depth has only harmed the observed results.

M2: Feed Forward Neural networks (FFNN)

A FFNN is a parametric model which can be used for both supervised and unsupervised problems and for classification and regression. Visually, it can be represented as multiple layers (stripes) of nodes fully connected between adjacent ones, where an edge denotes that output from the node is used as input to the next. Each node denotes an operation of:

$$o_k^l = act(\sum_{i=1}^{N} o_i^{(l-1)} w_i^{(l-1)\to o_k^l} + b_k^l)$$

where each node outputs a weighted sum of inputs, i.e. outputs of all nodes in the previous layer, adds a bias, and transforms it by a non-linear function. The layers between the input and the output one are called hidden layers. Usually, the input and hidden layers have the same activation function and similar sizes, whereas the last layer has a different activation function and a few or single neuron, depending on the problem. For our setup we had:

Layer	Number of Nodes	Activation Function	Dropout Rate
Input Layer	116	ReLU	N/A
Hidden Layer 1	128	ReLU	0.5
Hidden Layer 2	128	ReLU	0.5
Hidden Layer 3	128	ReLU	0.5
Output Layer	5	Softmax	N/A

The features are scaled to a range between $[0, 1]$ to decrease bias due to their different magnitudes and provide numerical stability. Scaling the features showed significant improvements in our results.

The input size corresponds to the features. The choice of 128 nodes in hidden layers was arbitrary.The dropout function was implemented in each hidden layer with $p = 0.5$ the way it was done by Hinton et al. 2012, where the dropout was applied to each layer with $p = 0.5$. *ReLU* was chosen due to its simplicity, easy implementation, and high reported effectiveness. We tested the classic *Sigmoid* activation function, which exhibited worse results. We did not test any other function. However for the output layer the *Softmax* function is used since this is the standard function for a multiclass classification problem. We did not test alternative functions or softmax modifications.

For the prediction in an FFNN, it is made by multiple transformations of the input given by the general-form matrix equation:

$$a^l = act(a^{l-1}W^{l-1\to l} + b^l)$$

Moving forward, let us break down what each part is:

a^l - a row vector containing the outputs of the current layer's neurons.

a^{l-1} - the "input" for the current layer, a row vector containing the output of the previous layer's neurones. For the first layer, the input is the actual data we "feed" our model.

$W^{(l-1)\to l}$ - a matrix of shape $(size(l-1), size(l))$ containing weights by which outputs of previous layer's nodes are scaled when passed to current layer's nodes i.e. $w_i^{(l-1)\to o_k^l}$ for all k and i, 'edges' between the two layers. Multiplication on the left side by the input vector yields a row vector with l entries. The $size(l)$, refers to the amount of nodes in the layer l.

b^l - a row vector with biases as entries, one for each node in the current layer.

$act()$ - a function performing an entry-wise non-linear transformation. Without non-linear transformations (activation functions), a model would be collapsible to a linear regression.

Training minimises the cost function by tweaking weights and biases with gradient descent. We use cross-entropy, as it is the proper function for a multiclass classification problem.

$$J(w^{0\to 1}, ..., b^1, ...) = -\frac{1}{N}\sum_{i=1}^{N} ln(\hat{y}_i) \cdot y_i$$

y - one-hot encoded ground truth row vector.

$\hat{y}$ - one-hot encoded prediction row vector. Contains the probabilities as the output of the softmax activation function:

$$\hat{y} = softmax(z^o) = \frac{e^{z^o}}{\sum_{i=1}^{5} e^{z_i^o}}$$

Both vectors have five entries corresponding to classes and each represents one datapoint. Further on, an optimiser we used, a modified version of Gradient Descent (GD), was the *Nesterov Accelerated Gradient (Descent)* (NAG) with `minibatch size=32`.

$$v_{t+1} = \beta v_t - \frac{1}{32}\alpha \sum_{i\in S} \nabla_\theta L([\theta_t + \beta v_t]; x_i; y_i)$$

$$\theta_{t+1} = \theta_t + v_t$$

$$\alpha = 0.1 \land \beta = 0.9$$

Velocities for all weights and biases are initialised to 0 in the beginning. At each step, the gradient descent velocity v is updated to be 90% of the previous velocity and 10% of the step gradient calculated *after* taking the *old* velocity step. This is different from GD with Momentum in which the gradient is computed at the current parameters, but the idea encompassed in NAG is that it is better to use gradient step to correct the velocity step after it has been taken. The velocity is a solution to faster convergence and avoiding getting stuck in a local minima. Visually, the descent would resemble a rolling ball downhill, though a bit lagging. Regarding coefficients, these β, α sum to 1 for easier interpretation.

We used mini-batch because it obtained a lower loss in comparison to full-batch in our case, likely by escaping local minima. In other words within each epoch, the training dataset is shuffled and divided into such sets of 32 points (mini-batches). During an *epoch*, gradient steps are taken on gradient computed only on corresponding 32 points. (Plots below display errors after the last batch in an epoch). For weights in hidden layers we use Glorot initialisation due to ReLU. For the last layer, He is used due to Softmax. Correct initialisation was crucial for obtaining a major improvement.

A gradient of weights and biases, before the last layer, is computed with the chain rule i.e. backpropagation. We perform computations on matrices, with loops used only for iterating through epochs and mini-batches.

$$\frac{\partial L}{\partial W^{(l-1)\to l}} = \underbrace{\frac{\partial L}{\partial a^o}\frac{\partial a^o}{\partial z^o}}_{\text{output layer}} \underbrace{\frac{\partial z^o}{\partial a^{(o-1)}}\frac{\partial a^{(o-1)}}{\partial z^{(o-1)}} \cdots \frac{\partial a^l}{\partial z^l}}_{\text{hidden layers}} \underbrace{\frac{\partial z^l}{\partial W^{(l-1)\to l}}}_{\text{depends on } a^{(l-1)}} .$$

To avoid computing Jacobi matrix for softmax: (it is $R^m \to R^m$), we directly compute the partial derivative of the cross-entropy with respect to the z^o i.e. $\frac{\partial L}{\partial z^o} = a^o - y$. Since for other layers we use $ReLU$, its derivative can take either 0 or 1.

We are delighted to have corrected Nesterov Accelerated Gradient formula in Papers With Code during the writing of the paper. It was the first results after typing 'Nesterov Accelerated Gradient'.

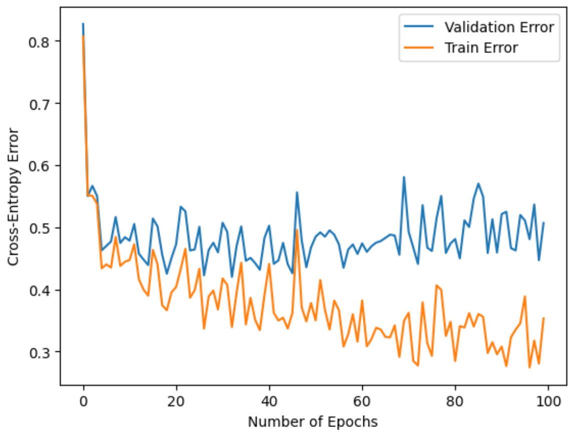


Figure 1: Momentum, no dropout

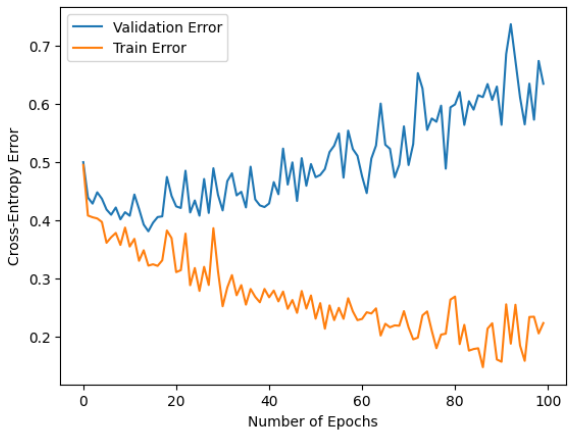


Figure 2: NAG, no dropout

In the figures above, even though one can see that NAG had a smoother error plots, converging faster and was able to get validation error below 0.4 and train error below 0.25, what Momentum was not, NAG seems to be more prone to overfitting.

As a technique to counteract over-fitting we use apply dropout on the hidden layers. During the training layers would be masked i.e. each neuron in the hidden layers would be disabled with $p = 0.5$ and multiple the outputs of the hidden layers by 2 to account for halved signal. The idea of Dropout is that it temporarily scales down the network (in different configurations), and smaller (less complex) networks tend to overfit less.

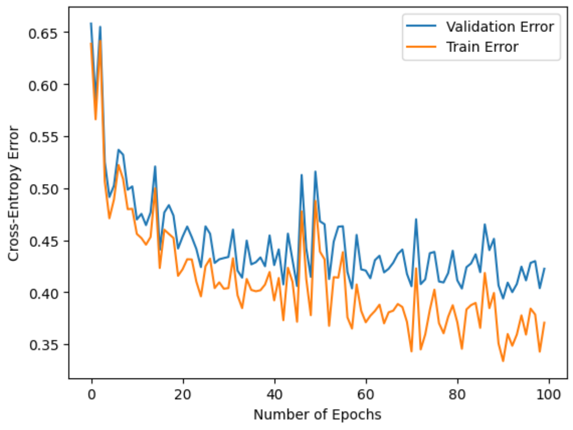


Figure 3: NAG, with dropout

Dropout indeed helped. It is seen that validation and training errors are 'following' each other. This means the dropout version is less prone to overfitting - as we wanted. OVR AUC is used as our correctness metric, as classes were balanced within the dataset.

Finally, for the FFNN implementation using libraries we reconstructed it using Tensorflow. The same architecture, NAG* with the same coefficients, the same loss, the same batch-size and the same way of initialising variables** - apart from the specific inner workings of Tensorflow the reconstruction was exact.

$*$ - Both tensorflow and Pytorch do not use the NAG as implemented in the original paper Sutskever et al., 2013, but utilise approximations. Specifically, they do not compute *shifted* gradient, but take the current gradient step first and then take the updated velocity step, as opposed to first taking velocity step and then correcting it.

$**$ - We didn't ensure that initialisation was at the exactly same points, we just ensured the initialisation was He and Glorot (Normal Distribution versions).

We use the Tensorflow implementation, directly from their documentation on NAG:

$$velocity = momentum * velocity - learning\_rate * g$$

$$w = w + momentum * velocity - learning\_rate * g$$

g is a gradient at the current place and momentum and *learning_rate* corresponds to our β and α

We assume that the above thorough explanation of the inner workings would suffice as the assertion of correctness.

M3.1: Random Forest (RF)

The Random Forest classifier model was implemented using the Scikitlearn library, and within it, the `RandomForestClassifier` method, which is an ensemble method, meaning it can utilize multiple Decision Tree models and combine them to reach a more accurate result. For the hyperparameters, we used `n_estimators=100`, which means we get an ensemble of a hundred trees. While higher number of estimators give better results, it comes with diminishing returns and the computation required steadily increases.

M3.2: Convolutional Neural Network (CNN)

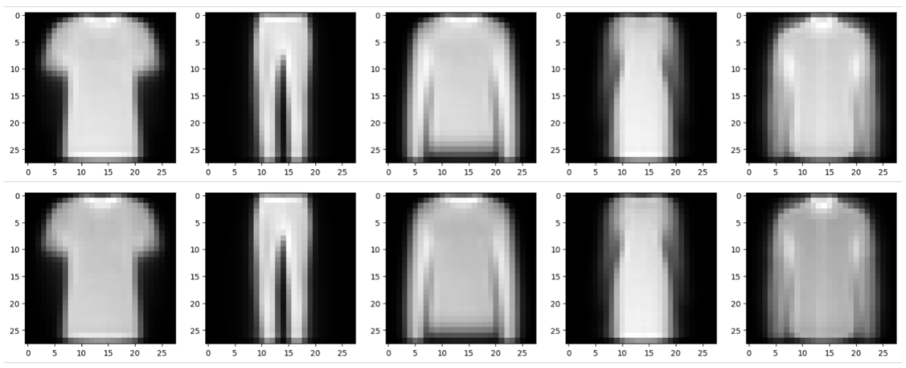
We used CNN as our reference model for comparing other models to, as these are known for superior performance with image data. CNNs apply filters called 'kernels' with learnable weights. This way CNNs can learn spatial relationships as opposed to FFNN which flattens data right in the beginning. The same optimiser, loss function and batch size was used for comparability. The setup, shown in the table below, was chosen to be simple, as time constraints did not allow for an in-depth dive into CNN.

First, we trained the network for 10 epochs and observed validation loss to start increasing and diverging from the train loss so the final network was trained on 7 epochs.

Layer Description	**Output Dimensions**	**Number of Parameters**
Input Layer	28 x 28 x 1	0
(Note: The batch size of 32 is implicit and not included in the output dimensions)		
Reshape Layer	Converts input to 28 x 28 x 1	0
Convolutional Layer 1	28 x 28 x 16	160
(16 filters, kernel size: 3 x 3, strides: 1 x 1)		
Max Pooling Layer 1	14 x 14 x 16	0
(Pool size: 2 x 2, reduces spatial dimensions by half)		
Convolutional Layer 2	14 x 14 x 32	4,640
(32 filters, kernel size: 3 x 3, strides: 1 x 1)		
Max Pooling Layer 2	7 x 7 x 32	0
(Pool size: 2 x 2, further reduces spatial dimensions)		
Flatten Layer	Converts to a single vector of 1,568 elements	0
Dense Layer 1	128 neurons	200,832
(Fully connected layer with ReLU activation)		
Dense Layer 2	5 output classes	645
(Fully connected layer with Softmax activation for classification)		

M3.3: Template Matching

One of the methods evaluated was a template matching approach, where a representative "standard" template is derived for each category and then used for comparison against new images. Standardization of the images helped equalize the contribution of all features and reduces the risk of scale-related biases or noise. Below are the two templates before and after standardizing the data.



Next, the training set was split according to its respective categories. For each category, a template was created by averaging pixel values at each position in a 28x28 grid across all images belonging to that category. This process yielded five distinct templates—one per category—to be used during inference.

To classify a test image, we measured how closely it resembled each category's template. Specifically, we subtracted a given template from the test image and computed the Euclidean distance. The expectation was that the correct template would produce the smallest distance, effectively "nullifying" the values when compared with the corresponding category's pattern. The category associated with the template that yielded the lowest score was then assigned to the test image.

To ensure the correctness of our implementation, we performed several checks. First, we verified that each standardized image had a mean of 0 and a standard deviation of 1 by calculating and reviewing summary statistics. We also inspected a small sample of templates visually to confirm that the averaged pixel values produced a coherent representation of the corresponding category.

Before we standardized the data our testing results had an accuracy of 0.6984 with a standard deviation of 0.0076. After standardizing our data, the general scores for the validation set improved significantly, with an accuracy of 0.7721 and a standard deviation of 0.0102. This gave us better results than we have initially expected and an overall improvement over the initial approach without standardization.

Since the template matching method is based purely on averaging pixel intensities for each category and directly computing Euclidean distances to classify test images, it does not introduce any tunable hyperparameters such as learning rates or regularization parameters. The choice of template size (28x28) is derived directly from the input images and thus is fixed. Additionally, these distance scores were stored as high-level features for use in other classification methods.

Results

While some results have been mentioned before, below are brief tables of significant results such as the Accuracy and/or AUC scores of each model, which we found to be most important as the classes are evenly distributed within the dataset. Starting off with the models implemented from scratch (M1 & M2):

Model	**Training Dataset**		**Test Dataset**	
	Accuracy	**AUC**	**Accuracy**	**AUC**
Decision Tree	0.74	0.87	0.76	0.84
Feed Forward Neural Network	0.86	0.98	0.74	0.93

And finally the library implementations (M1, M2, M3):

Model	**Training Dataset**		**Test Dataset**	
	Accuracy	**AUC**	**Accuracy**	**AUC**
Decision Tree	0.78	0.89	0.76	0.88
Feed Forward Neural Network	0.69	0.92	0.62	0.90
Random Forest	0.85	0.97	0.82	0.96
Convolutional Neural Network	0.83	0.97	0.82	0.97
Template Matching	0.77	0.86	0.75	0.84

One detail to note with the above results, which is not obvious: The AUC of the Decision Tree model (scratch implementation) and Template Matching model is approximated by it's confusion matrix as the model does not provide multiple thresholds.

While we could also include other metrics such as Precision, Recall, F1, we have decided that it is acceptable to omit them as those would provide less relevant information compared to Accuracy or AUC.

During the training of all methods, we have noticed that the fastest for training was the Template Matching method we designed, as it involved mostly simple matrix calculations. Next was the Decision tree using the library implementation, which we suspect is due to the built in optimizations to the method, and some approximations it uses that help reduce the required data needed. As for the slowest method, it was the Decision Tree, but this time the scratch implementation. We have not implemented any optimizations outside of running it using the `joblib` library for parallel processing, and even so, the entire dataset had to be processed recursively, which is a massive undertaking for personal computers. In the middle ground sat the other methods. The Random Forest used the optimized decision trees, so even 100 of them was exceedingly fast, meanwhile the neural networks saw similar performances in both the library and scratch implementations, with the library implementations coming only slightly ahead.

As for comparing our results, we have noticed that the results do not change much between the training dataset and test dataset, except for the library implementation of the Feed Forward Neural Network, which we will get to shortly. The Convolutional Neural Network was best performing alongside the Random Forest and the Feed Forward Neural Network implemented from scratch. We have expected the neural networks to perform well as they are capable of gathering meaningful information from data that may be abstract to us, and the Random Forest was expected to perform well too, albeit above our expectations, due to it being an ensemble method and can utilize the knowledge of many individual models to gain the best results from those.

Unfortunately, on the other end of the spectrum, we have observed subpar results with the Decision Tree and the library implementation of the Feed Forward Neural Network. While the Decision Tree was expected to be not the best performer as it is a simple method that can give varying results, the Feed Forward Neural Network has left us shocked. We noticed that after 100 epochs the library implemented model achieves inferior performance of 0.62 Accuracy but with a 0.90 AUC-OVR on the test dataset. We expected even worse results in our scratch implementation, but to our surprise, it performed better. We spent considerable time investigating it, and came to the conclusion that that Tensorflow might utilise additional improvements which unfortunately lead to overfitting despite dropout layers. On the other hand it could be that approximation of NAG was not sufficient. As for the discrepancy between the Accuracy and AUC, due to time constraints, we were not able to analyse what might be the root cause, as the classes are balanced, however, this will be the focus of our near-future investigation.

Finally the Template Matching method has performed surprisingly well given that it was our first time trying to build a completely new custom model based on our current knowledge within Machine Learning, however, given that it is similar to a K-Nearest Neighbors classifier, the results may be more expected than our initial impressions.

Discussions

A few questions and topics arose from our observations and the results we got. First, we have tried to train our models on just a subset of the features, to evaluate whether the high-level features or the PCA were responsible for the most contribution. In our findings, both had similar weights in the results, however, the combined feature dataset outperformed the individual part. We recognize that there are specific cases where one would be preferred over the other, such as images where high level features can not be extracted, PCA could provide an immense help in reducing computation requirements.

Next, we must interpret what the results actually represent beyond just a score from 0 to 100 of how well our models performed. Since the classes within the dataset are equally distributed and there is no differentiation of importance between them, we opted to rely on the Accuracy and AUC scores, as those would be the most meaningful. Both were very promising, with scores of 0.93 for AUC for the FFNN and 0.97 for the CNN showed phenomenal results, leaving little room for improvement for the scale of this project.

As for real-world application, we concluded that the results are not enough if it were crucial to be correct, as any product that does not fit into these classes would give inconsistent results at best.

Conclusion and future work

While all classification results seem to be promising, we have observed that Neural Networks provide the best results, and the Random Forest classifier also does exceptionally well, considering its simplicity of expanding upon the Decision Tree classifier.

Moving forward, we would like to explore the entire Fashion MNIST dataset, and not only refine our existing models for even better classification, but also implement new ones that could tackle much larger problems in the future, or even develop generative models to acquire new insights in the world of machine learning.

For any external analysis or validation of our work, all notebooks, scripts, .CSVs, and additional information can be found on GitHub at: *https://github.com/PLtier/classification-Fashion-MNIST*