Machine Learning Project

Antonia-Bianca Zserai, Daria Damian, Karlo Krivaja

January 6, 2023

Abstract

Image classification is a complex process that can be impacted by many factors, making it essential to understand current practices and problems associated with the process. This paper implements and examines three different non-parametric classifiers to cross compare their accuracy and viability for use in the specific case of simple image classification.A deep learning technique was build from scratch by implementing a simple Feed-forward neural network, in conjunction with a reference implementation from the Tensorflow module on the same dataset. A Decision tree model was also built from scratch along with a reference sklearn python library implementation. The third classifier was chosen as a standard implementation of a Support Vector machine from the sklearn library. We discuss in more detail each of the implementations, specific hyper-parameter choices, and improvements that can be done to achieve better results. The results show, that on our specific image data set consisting of different clothing articles, the neural network and support vector machine models perform in a similar manner with regards to accuracy, and both outperform the simple Decision tree implementation in accuracy.

1 Introduction

With development of new technologies, the human race has tried to improve quality of life by automatizing everyday processes with the goal of making human participation in such processes obsolete. We observe such improvements every day in the form of virtual assistants (Siri,Alexa),online customer support, video surveillance, self driving cars and facial recognition just to name a few. Last one is out of particular interest to us, facial recognition, or more generally image recognition and classification. Researchers have tried for years to apply different computational techniques to find optimal ways to classify image data. These techniques might be different but they all have a common goal of classification of detected objects and patterns in the array representation of images into various categories.

Our paper will focus on three of the most commonly used techniques. Neural networks are models which are made to mimic the neural signaling inside the human brain and to find complex relations in data without any pre assumptions by applying activation functions to the data. Decision tree classifiers are very commonly used method for image classification due to its easy representation in form of root-branch-leaf analogy, readability and decent computational results without much complexity. The last used method of Support vector machines works great for non-linear data such as images and reacquires smaller amounts of training data.

In our paper we will attempt to apply these tree methods to our dataset of clothing images to see each individual performance, find similarities and differences, and perhaps be able to explain the same with the method natures and specific implementation setups.

2 Data and Preprocessing

Training images are an essential part of any image recognition model, as nn-parametric classification heavily rely on them to create a perception of what certain classes should look like, what their characteristics should be or where they should lay in a specific n-dimensional space. Without these training images, the accuracy of the model decreases significantly. To ensure the accuracy of the model, it is important that data sets used for training contain a variety of images which helps to understand different aspects such as color, shape, size etc. These features are self contained within the data in one way or another and create a basis for predictive models.

2.1 Datasets

Our analysis was conducted through a dataset containing 15,000 labelled clothing images taken from the Zalando website (Xiao et al., 2017). Each image was a 28x28 grayscale representation of a t-shirt/top, trousers, a pullover, a dress or a shirt. The dataset was split into two groups of 10,000 images for the training set and 5,000 images for the test set. Every line of the dataset contained information such as pixel values (ranging from 0 to 255) and clothing category (ranging from

0 to 4)(Figure 1.).The images are represented by an array consisting of 784 image pixels and 1 class representation scalar. Our project focuses on providing an accurate analysis of the image and clothing category by using the training set to create an accurate prediction model for the test set. This is done by using various algorithms to predict the clothing category accurately depending on the pixel values of the image.

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

Figure 1: Categories of clothes

2.2 Preprocessing

The preprocessing of images is a crucial step in the feature extraction process, which plays a key role in content-based image classification. This preprocessing may include detection and restoration of bad lines, geometric rectification and image registration. All these processes make the extracted features more suitable for the clothing images' classification accuracy. In the method comprising of machine learning, at first classical data was split into training and and test data along with labels. This was followed by preprocessing classical dataset with feature dimension 5 using standardization, principal component analysis and normalization. This further helped to improve the accuracy of the clothing images' classification

After loading the dataset into panda's dataframes, we were able to determine the shape and size of the data, as well as what kind of data was available (10000, 785). To better understand the data, we plotted the number of images in each label group. This plot showed that any label group had a maximum number close to 2033, which was why we decided to plot it in a log scale (Figure 2.). This allowed us to gain a better understanding of the data and its distribution. It is important to note that the split is almost even so the log transformation is just a means of visualization and in no way suggests a bad category split in the data itself.

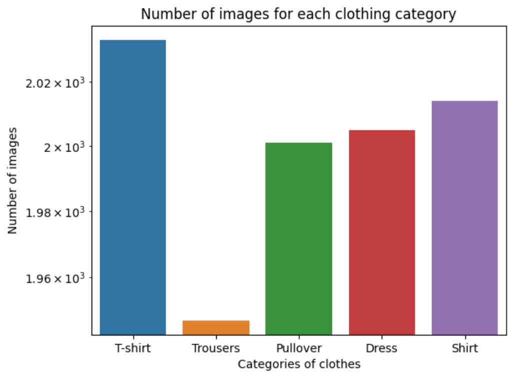


Figure 2: Log Plot for the number of images in each clothing category

2.3 Visualization of data

The average picture is calculated by taking the average value of each pixel across all observations. We utilized **bicubic interpolation** to display it, a 2D system that combines polynomial algorithms to sharpen, expand, and upgrade digital pictures from a one-pixel grid to another. This enables smoother re-sampling with fewer errors. Because the colors represent function values, we can see that they are not radially symmetric. This generated a comprehensive representation of the data set, which was then utilized to construct general images of each group (Figure 3.).

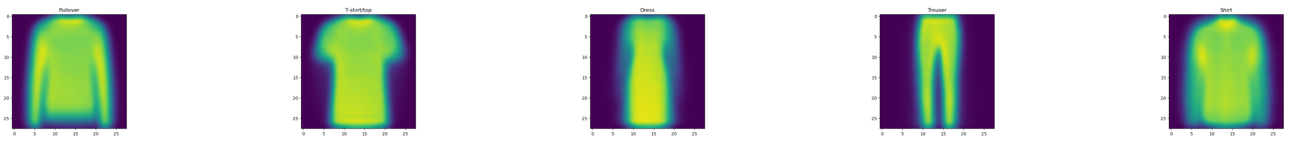


Figure 3: Average Images of each category

By using the average images (Figure 4), we were able to compute the difference between two different classes, providing us with an even more detailed understanding of our data. With this information, we can effectively compare and measure how much variation there is between individual images(Figure 5).

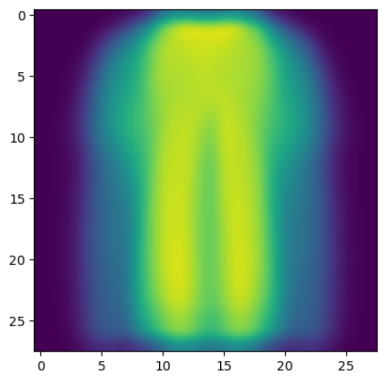


Figure 4: Average Image

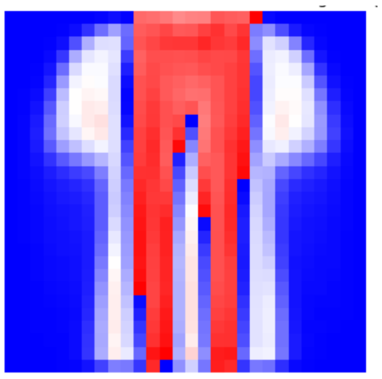


Figure 5: Difference between average images of t-shirt and trousers

Lastly, we used a dimension reduction technique such as the principal component analysis (PCA) to visualize the components that describe each class the best. The eigenimages(Figure 6), which is essentially the eigenvectors (components) of PCA of our image matrix, can be reshaped into a matrix and be plotted. It's also called eigenfaces as this approach was first used for facial recognition research. Here are visualized the principal components that describe 70% of variability for each class. Figure 7 shows a PCA graphic with two dimensions that we do not utilize in future statistical computations because of its low variance(Figure 8.).

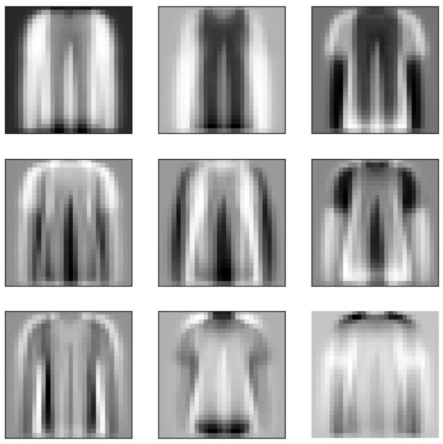


Figure 6: Plot of eigenimages in a grid

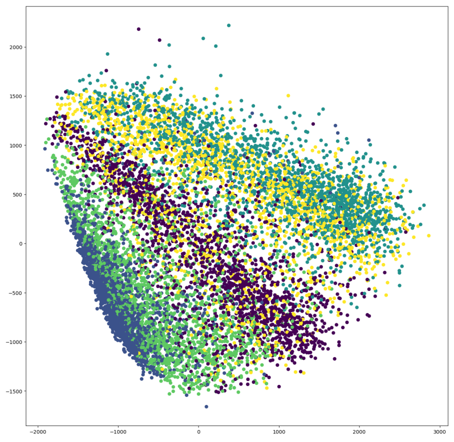


Figure 7: PCA with 2 dimensions

PCA's main goal is to decrease the number of variables in a data collection while retaining as much information as possible. So, after analyzing it, we wanted to evaluate how much information the top 30 primary components preserved.

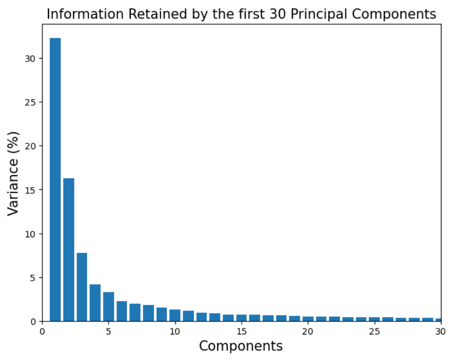


Figure 8: Information retention

We utilized the Tensorflow Keras utils library to normalize the data. For image classification, TensorFlow is used in connection with Python as the programming language. It uses dataflow and differentiable programming to carry out a variety of tasks centered on deep neural network training and inference. It allowed us to generate a list of tuples of the shape (x,y), where x is a (784,1) shaped NumPy array and y is the resultant classification component for each of these inputs. The result is a five-dimensional vector which will be used in the neural network training with 1 at the v position and 0 elsewhere to indicate the classes (0,1,2,3,4).

3 Methodology

The machine learning approach enhances picture categorization performance. This section will go through three machine learning models, the way we implemented them and the results obtained given the datasets.

3.1 Neural Network

3.1.1 Network basics

Neural networks represent the cornerstone of so called Deep learning. They are a very specified set of algorithms which mimic the way our brains work, and the way our neurons transmit signals to one another. A neural network consists of multiple layers, more precisely the input layer, hidden layers and the output layer. The layers are comprised of neurons that connect to neurons of another layer. The input layer receives an input vector p containing network features to be analyzed. Each input is then linked to one of the neurons in the hidden layers, with weights assigned to each connection. Each model chooses a specific activation function to convert the input variables and obtain the output variables. After crossing through the activation function, the variables are passed to the next layer of the network. This is why these networks are referred to as **Feed-forward networks**.

When discussing the use of neural networks for classification, we may want to imagine the output layer representing each of the distinct classes. In an ideal situation, the greatest output value will undoubtedly reflect the desired class. Since this will not be the case from the very beginning, we had to recompute the weights and activation's of each layer in order to get good classification results.

To evaluate this accuracy we used the cost function. **The cost function** represents the difference between the acquired value and the actual value, and tells us for each of the neurons if their values need to reduce or increase. The goal is to gradually converge to the minimum of the cost function, by adjusting weights and biases through gradient descent which allows adjustment in the right direction.

3.1.2 Our case

We applied the same logic presented onto our data to create a neural model from scratch. The input layer of our network consists of 784 neurons directly uploaded from our data. Each of the neurons represents a feature of a single photo that is being analyzed, that is each individual pixel in the photos 28x28 grid. The last point in each row of the data, the 785th point, was excluded from the input data because it numerically represents the class of each photo. In our model the output layer consists of 5 neurons, each representing one of the classes. The actual training process begins by initializing the initial weights and biases with a normal (0,1) distribution. Prior the training, the input data was normalised, so the choice of a Gaussian distribution for the weights an biases seemed like a good starting point which would be easily used for computations in the rest of the training. Due to the limitations of our computers, we made the decision to use a mini batch approach to training, to increase speed and find errors more efficiently. The choice of hidden layer number and sizes, as well as the batch size was done by experimenting and running a few different setups to see what gain or loss we have in accuracy vs speed.

Back-propagation was run in order to acquire the Cost function to help regulate the weights and biases. The activation in propagation of the neurons was achieved by using the **Sigmoid activation function**. This is a frequent activation function choice, although it may not be the best. We opted to utilize it for simplicity and to test whether we could achieve good performance in this particular scenario. We then used the derivatives and activation function to backtrack to the initial weights and biases and acquire the desired changes to be applied and get the new updated values.

The gradient computed calculates the difference between the true value and the expected value, thus giving us an insight in which way do we need to remedy the weights to approach the desired minimum of the cost function. The amount of the movement is determined by the learning rate parameter that we select. This was also optimized through testing by altering a few different values and observing the impact on prediction.

We calculated the accuracy by feeding in the test data to obtain the real classification, then summing all of the classifications in the test data that were equivalent to the predicted classification and dividing it by the total number of inputs from the test data.

3.1.3 Our implementation

We implemented the neural network in an object oriented matter. This choice was made due to the sole nature of the networks, and the way they are made to mirror natural neurons inside the human body. As such, each network represents a different body, and it is saved as such, different individual objects. This makes it very easy to tamper with the different input parameters and easily compare objects after done so to find the optimal setup. The network object class created consists of 7 methods within its body.

The **initialization method** represents the basis of the network, and it initializes the layer number and sizes, as well as the weights and biases, using the random normal distribution. Through this method, t he network object was created.

The second calling method is the **GD** (Gradient descent) method. In this method, we pass the training and test data, as well as set the batch size, number of runs (epochs), and learning rate. For convenience, we also generate a result list that retains the accurate prediction values of each run for easier visualization following analysis. In this technique, batches are generated at random based on batch size. The update method is then applied to each batch. This is done on the training data. Once the model has been trained, the test data is passed to the evaluate method and the resulting scores are printed and saved into a list.

The update method has 2 input variables passed, the batch and the learning rate. The batch is a small subset of the data passed by the GD method, and the learning rate is the predetermined hyper-parameter we already defined. By passing the batches we ensure that each individual part is executed in a speedy manner and also we can perceive errors a lot faster. Our training data is after all of shape (10000x784), which would be quite taxing. In order to do so, we first create a multidimensional array of 0s in the shape of the weights and the biases. These arrays will be used to store the change in values that we will get from running back-propagation method on each sample in the batch. The weights are then updated by using the initial weights, learning rate and the weight change obtained by back-propagation.

The **backprop method** is used to run back-propagation on given input sample and its respective expected value. It returns the gradient for the cost function Cx, in the form of a tuple for weights and biases. Similarly to the update method, we initialize arrays of 0s for the weights and biases, as well as set the activation's to the current sample, create an activation list to store all the activation's and a list to hold the z values (the representation of a network value expressed through weights and activation's per layer coupled with bias). Since we are working with vectors, the z values can be obtained by using the numpy dot product. Then the activation function which was predefined as a sigmoid function is run on the z value to obtain the activation. To achieve the needed changes in bias and weights, all we needed to do was to use the chain rule and the described method cost derivative, to backtrack and obtain the values using simple dot products. We iterated with negative indices back from the maximum layer number to the second layer.

The **cost derivative method** takes into account the output activation computed with the activations in the backprop approach, as well as the expected value. It returns a simple difference between the expected value and the output activation.

The **evaluate method** is used to evaluate the supplied test data presented. The input is the test data, and the output is the number of correct outcomes. We use the feed method to see whether the highest class corresponds to the position holding 1 in the proper class vector.

The **feed method** is essentially forward propagation with the activation function (in this instance, sigmoid) applied to the weights, biases, and activations. It returns the network output for a specific activation input.

We ran 5 alternative setups, taking just the most significant results to check whether any of the figures changed considerably. Once the optimal bat size was determined, we modified the learning rate. As can be seen in the graph, there is no major change, as predicted.

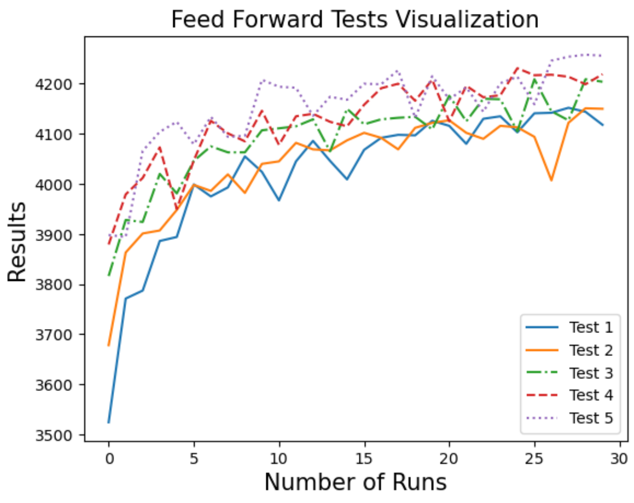


Figure 9: Feed Forward: Tests Visualization

3.1.4 Reference implementation

To compare and asses the accuracy and validity of our implementation, we ran a reference implementation using the Tensorflow and Keras modules. We used a Sequential model with batch size 128, 256 hidden units and 3 different layers with activation sigmoid functions for reproducability and comparison with our method. We fit the model with the training data, ran it on 20 epochs with accuracy as a metric. The accuracy metric obtained was 0.807.

3.2 Support vector machine

3.2.1 Basics

SVMs are a common Supervised Learning systems that use a hypothesis space of linear functions in a hyperspace and are taught with an optimization theory learning algorithm that applies a statistical learning theory learning bias.(Roweis and L. K. Saul,2000). The SVM method seeks to determine the best line or decision boundary to split n-dimensional space into classes in order to efficiently place a new data point in the correct category. For this to be done, SVM picks the extreme points/vectors that contribute to the formation of the hyperplane.

$\beta_0+\beta_1X_1+...+\beta_pX_p = 0$ defines a p-dimensional **hyperplane**, again in the sense that if a point $X = (X1, X2, ..., Xp)^T$ in p-dimensional space (i.e. a vector of length p) satisfies the equation, then X lies on the hyperplane.

The goal of Classification via SVM is to find a computationally efficient way of learning good separating hyperplanes in hyperspace, where ’good’ hyperplanes optimize the generalizing bounds and ’computationally efficient’ algorithms are those that can deal with substantial sample sizes.

SVM uses kernels to enlarge the vector solution space. A kernal is a non-linear generalization of a vector inner product given as a function of K.

For degree d polynomials, the polynomial kernel is defined as:

$$K(x_1, x_2) = (x_1^T x2 + c)^d \tag{1}$$

where c is a constant and x1 and x2 are vectors in the original space. When the support vector classifier is combined with a non-linear kernel such as, the resulting classifier is known as a **support vector machine**:

$$f(x) = \beta_0 + \sum_{i\in S} \alpha_1 K(x, x_i) \tag{2}$$

3.2.2 Our case

Because in our case a separating hyperplane exists, we can utilize it to build a very natural classifier: a test observation to which is assigned a class based on the side of the hyperplane that it is on.

There are an endless number of hyperplanes that can divide the observations without touching them. Therefore, we selected one: the maximum margin hyperplane, a natural option since it is the separating hyperplane that is farthest from the training observations.

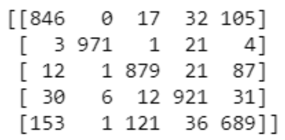
Some training observations are equidistant from the maximal margin hyperplane and lie along the dashed lines indicating the width of the margin. These three observations are known as support vectors, since they are vectors in p-dimensional space and they “support” the maximal margin hyperplane in the sense that if these points were moved slightly then the maximal margin hyperplane would move as well.

The model accuracy indicates how often is the classifier correct and our implementation result in 86%. The retrieval performance is measured by precision and recall, defined below.

$$\text{Precision} = \frac{\text{number of relevant images retrieved}}{\text{number of images retrieved}} \tag{1}$$

$$\text{Recall} = \frac{\text{number of relevant images retrieved}}{\text{number of relevant images in the database}} \tag{2}$$

Because high precision and recall rates indicate that the model is performing well, our precision and recall percentages of 86% indicate that we have a solid model. Lastly, a **confusion matrix** was created, which is a matrix used to determine the performance of the classification models for a given set of test data. For this, we must first import the sklearn library’s confusion matrix function.The function requires two parameters: the true values for test data that are known and ypred (the targeted value returned by the classifier). The confusion matrix illustrates how often label 6 (shirt) is misclassified as T-shirt, pullover and dress. If it were not for these strong similarities between the labels the overall performance would be higher.



```
[[846   0  17  32 105]
 [  3 971   1  21   4]
 [ 12   1 879  21  87]
 [ 30   6  12 921  31]
 [153   1 121  36 689]]
```

Figure 10: Confusion matrix

3.3 Decision trees

3.3.1 Basics

In machine learning, decision tree ensemble methods are particularly popular. These approaches rely on merging the predictions of numerous models to improve an existing learning algorithm. They are more successful when used with decision trees, which are frequently less accurate than other learning algorithms . They make no prior assumptions about the application problem; they have successfully applied to numerous complex problems in a variety of application fields and compare favorably with other algorithms.

The tree is structured by the root node(the top of the tree), internal nodes (or decision nodes), leafs(or termination nodes), and all the branches (edges) that connect them. For the purpose of this report, we will be focusing on **classification trees**, which are trees used to classify an input into specific sets of classes in the most optimal possible way. The algorithm works by splitting the data at particular nodes, and then calculating the cost of the split and choosing the best possible split. When talking about classification we can use the **Gini Impurity** score to see how good the split actually is.
The *Gini index*:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - (\hat{p}_{mk})) \tag{3}$$

is a measure of total variance across the K classes. A small value means a node had values of predominantly one class. A perfect class purity gives a G=0 score. To avoid complexity and over-fitting, a minimum leaf size can be set (the minimum number of items per leaf). One can also select maximum depth (the longest path from the root to a leaf) to achieve similar effect.

3.3.2 Our case

We created a basic method for implementing such trees by first creating and defining a tree with the specific hyper-parameters. We set up the maximum depth and minimum leaf sizes in order to optimize the method and avoid complexity. The data input consists of the X variable (representing the training data set of 784 pixels of each of the photos), and the Y variable (representing the corresponding expected classification value). The K classes consisting of 5 classes (numbered 0,...4) were passed into the method itself. The method also keeps track of all the classes within a node, and the most present class in a node at any time. The class ratios are used in computing the Gini score of the current node. At any moment, for each new split of the data, we check if the newly created leaf and right leaf have leaf size more or equal than the preset minimum. In the case these requirements are fulfilled we check the lowest Gini scores and take the best split at that point. If the leafs are smaller than the minimum, the split is ignored and a new one will be made. Once we find the optimal split, the left and right branches of the split are ran through the same process to find their optimal splits, all the way until we reach the end result, which ends either when the maximum depth is reached and/or all the leaves are at the minimum size. These splits will contain our individual images with their respective pixel values. The end nodes will represent individual classes on the basis of the most common class within the node. This procedure once done will represent a trained decision tree. Inputting new test data should eventually lead to classifying the individual test images by sorting them in one of the leaf nodes.

3.3.3 Our implementation

We choose to implement the Decision tree algorithm using object oriented programming. This was done in order to be able to easily run multiple trees, save them, and use it to optimize our hyper-parameters. The DT (decision tree) class that we created consists of 6 different methods.

The **initialization method** is used to establish a tree object that takes input in the form of data inputs along with their separate respective classes in a list, maximum depth, minimum leaf size and hyper-parameters as well as a list of classes.The method initializes the input samples and their classes in two distinct Numpy arrays. We also collect the sample number from the shape of the sample array. The method then calls the **set classes method** to determine the number of potential classes as well as their count in the current node.We also select the most frequent class from the node count list for the current node. The class count is then used to obtain all of the class ratios and compute the Gini index for the node. Further, the method checks if the maximum depth was reached, or if the Gini index is 0 (meaning only a single class is present) and terminates the current iteration of the method. If we did not terminate, we proceed to call the **make tree method**. The initialization method will also be called in other methods during the tree-building process, so we will discuss about computing all of these variables for a current node, whether that's the first set root node or another node along the build.

Set classes method creates a list of possible classes, as well as initializes a list of counts for each class.

Make tree method is used to build up a decision tree. First we initialize the maximum Gini value as something that will not be surpassed, in our case we put 2. We are looking and searching for values close to 0. Adding the computed left and right Gini values can provide a number of up to 2, which is why we choose this as the upper boundary. We next run through the samples, sort them into left and right nodes in the shape range, and calculate the Gini index at each split. After, we check to see if the index is smaller than the set index. If so we switch the Gini index for the new one and save the split information. If the resulting split has a Gini index of 2, or the leaf sizes for either of the nodes, left and right, are smaller than the minimum leaf size, we terminate and return none values for the split information. Otherwise, we proceed to call the Decision tree itself on the split information for both left and right acquired split nodes.

The **Accuracy** method is designed to output an accuracy score for a given set of data after the training of the decision three is done.The method takes the data to test it along with it's intended classification. The inputs are turned into numpy arrays, from which the number of samples is calculated. Then the **predict method** is called, and the accuracy is returned as the number of samples where the predict method gives the same classification as the expected classification.

Predict class method accepts testing data as input, then runs the Classify method to categorize each sample and returns a numpy array of predictions.

Classify method checks if self or rights splits of the node are None. If so, it returns the node class, otherwise on the same principle as build tree it goes and checks the side where the sample goes, starting from the root node, going left or right, and then running the same node just starting from the next following left and right node until reaching the None condition that was set in the beginning of the method.

3.3.4 Reference implementation

We built a Decision tree classifier with the sklearn framework to compare our results and implementation. Using the identical training and test data as the other techniques, we achieved an accuracy score of 0.7674.

4 Discussion

4.1 Neural network

Our implementation of the neural network is the most basic one using gradient descent cost regulation with back-propagation and sigmoid activation function. After running a few different set ups by changing the hyper-parameters, such as number of runs (epochs), batch size, learning rate, layer number and constitution we obtained accuracy scores between 0.81 and 0.86. There was no clear improvement in scores on different setups, since after running each a few times the scores of the same setup would also range in the aforementioned range. This lead us to believe that a simple set up with 3 layers of size (784,50,5), batch size 10 and learning rate 3, run over 30 epochs gives same results as more complex and harder to run parameters. By increasing the parameters we did not obtain better accuracy but created more complexity in the model, which is not ideal since it provides no apparent benefits. The only performace metrics used was the accuracy, that is the number of correct predictions divided by the total sample count.

To give more meaning to our result and to see how our implementation is, we ran a standard neural model from the Keras module. We set the parameters to mirror our scratch implementation parameters as closely as possible and observed the result. The accuracy score metric ranged from 0.80 to 0.82. This is within our range and as such does give expected results if the models were similar. This also confirms that our implementation is as least as good as the similar standard Keras implementation.

Now when we mention standard and basic implementations we are implying that we have not used any additional improvements to the model and performed no additional wrangling and tampering with the data besides the normalization. This also means that there are many different steps we could have taken to potentially improve the model. One improvement that could have been done is to change the activation functions. The sigmoid function is a very widely used activation function, but it encounters some problem when the gradient is below or above a certain value. This causes the gradients to saturate and give values of zero. Another issue is that it is not zero centered. To avoid this we can substitute with other functions. One common approach is to replace the last activation with the Softmax function. This function can be described as a combination of multiple sigmoid functions.Softmax calculates relative probabilities and assigns probability 1 to the highest probable and 0 to all the rest. As you might assume this can be particularly good in our multi-class classification model where we would prefer one of the output activations to be 1 and others 0. Another improvement could be to change the loss function, or perhaps change the way of initializing weights and biases. We initialized them from a Normal distribution,with mean 0 and sd 1, but we could done it using the Xavier initialization which draws from an Uniform distribution and has become a good go to for weight initialization.

Since we are analysing images, we could have taken a different approach, one in which we use convolutional networks for our analysis. Convolutional networks use kernels to extract activation features, or those features important for classification. This improves not only accuracy but also run-time. These kind of networks are commonly used for image classification since they are good at recognizing individual features that constitute a class. Since these kinds of networks were not in the scope of this paper we did not take this approach, but it is definitely an approach that could provide better results than the basic model we constructed.

4.2 Decision tree

Our decision tree is a simple tree model with predetermined maximal depth and minimum leaf size which uses the Gini index for node computations. We have tried to run the model on different sizes of training data, ranging from 1000 samples to 10000 samples, and acquired quite similar accuracy scores in the range 0.74-0.77. Changing these parameters and leaf and depth parameters did not prove to give any significant change in model prediction accuracy. The higher values were found mainly when using a larger data-set and lower when using a smaller, but as you might notice from the range the difference was not that big. To compare our scores we ran a standard Decision tree classifier from the sklearn package and got an accuracy score of around 0.76-0.77. This more or less coincides with our scores giving some proof to the validity of our general implementation.

There is a number of things we could do to improve our model. One thing to try could be to use Entropy instead of the Gini index for building trees, or even use the classification error rate, if our primary goal in this project would be accuracy. We could have also improved accuracy by implementing many other ideas into our model, such as Bagging, Random forests and Boosting to name some.

Another approach is to prune the tree, e.g. remove the nodes which do not or extremely slightly impact the tree accuracy. This in case also reduces complexity. Feature reduction can also improve accuracy. An example of this is the PC (principal component), but in our case the most prominent component was around 30 percent, and all the rest below. This could mean a bad fit if used so we did not.

4.3 Support vector machine (SVM)

Our SVM implementation incorporates several soft margin parameters and a polynomial kernel function.

A decision boundary is used by SVM to divide data points into distinct groups. An optimization problem is attempted to be solved by a soft margin SVM while determining the decision boundary. A soft margin suggests that some data points could be incorrectly categorized. The objectives are to maximize the number of points properly classified in the training set and to increase the distance between the decision boundary and the classes (or support vectors).

Kernel functions are a powerful tool for exploring high-dimensional spaces. They allow us to perform linear discriminants on non-linear models, resulting in improved accuracy and robustness than standard linear models. In machine learning, the polynomial kernel is a function commonly used with support vector machines (SVMs) and other kernelized models to reflect the similarity of vectors (training samples) in a feature space over polynomials of the original variables. Furthermore, because they extract intrinsic qualities of data points using a kernel function, they provide more features than other techniques, such as neural networks or tree ensembles, in applications like handwritten identification, face detection, and so on. A polynomial kernel is a type of SVM kernel that maps data into a higher-dimensional space using a polynomial function. It accomplishes this by taking the dot product of the original space's data points and the polynomial function in the new space. The polynomial kernel intuitively analyses the supplied properties of input samples to discover similarity and combinations of these. Such combinations are known as interactive features in regression analysis. When the input characteristics are binary (boolean), they correspond to logical conjunctions of input features. Kernels are also useful since they can be utilized to reduce the SVM algorithm's errors. The SVM method can find a hyperplane that divides the data more accurately and with fewer errors. The degree of the polynomial and the coefficient of the polynomial are two parameters of the polynomial kernel that can be modified to increase its performance.

The C float parameter trades off correctly classifying training samples against maximizing the margin of the decision function. If the decision function is more accurate at classifying every training point, a smaller margin for greater values of C will be accepted. We defaulted our C value to 1. A lower C value value typically result in more support vectors, which could increase prediction time. A trade-off between fitting time and prediction time results from decreasing the value of C. C controls the trade-off between these two goals by adding a penalty for each misclassified data point. With a greater C value, SVM attempts to reduce the number of misclassified examples as a result of the high penalty, resulting in a decision boundary with a smaller margin. In other words, C performs in the SVM as a regularization parameter. With linear kernels, the impacts of hyperparameters are only to a limited extent. With non-linear kernels, however, the influence of hyperparameters becomes more obvious.

Furthermore, the degree integer parameter reflects the degree of the 'poly' kernel function, which is set to 3 by default and is disregarded by all other kernels.

Finally, the decision shape parameter determines whether the algorithm will produce the ovr (one-vs-rest) shape decision function as the other classifiers or the original ovo(one-vs-one) decision function. We set it to 'ovo,' which means we used the 'One-vs-One' approach for Multi Class Classification. Like one-vs-rest, one-vs-one splits a multi-class classification dataset into binary classification problems. Unlike the one-vs-rest strategy, which divides the dataset into one binary dataset for each class, the one-vs-one approach divides the dataset into one dataset for every other class. OvO predicts the model with the most predictions or votes. If the binary classification models predict a probability, then the argmax of the sum of the scores (class with the most significant sum score) is predicted as the class label. This method is recommended for support vector machines (SVM) and other kernel-based techniques, which can be implemented using the scikit-learn module. This is thought to be due to the fact that the performance of kernel methods does not scale in proportion to the size of the training dataset, and adopting subsets of the training data may mitigate this effect.

The results obtained were very similar to the ones acquired from the Feed-forward method. All of our measurements: accuracy score, precision and recall being around 86%. SVM is a great classification algorithm. It is a supervised learning technique mostly utilized for classifying data into several categories. The SVM algorithm has demonstrated good performance in classification. If the boundaries are good, it requires less data to produce satisfactory results.

4.4 Method comparison

We can clearly see that our Feed-forward methods and the SVM method provide similar results, while the Decision Tree method falls a bit behind. To see why, lets first consider the data itself. The only pre processing done on the data was normalization. This creates a normalized data set shaped 28x28 pixels, where a lot of the inputs in the array are just 0 values. The data like this is big, and sparse, and is contained within a large data set of 10 000 training samples. For a simple tree this is not ideal. It can lead to complexity and problems in running, which we have encountered, but also over-fitting. Reducing the number of features, eg our pixels would improve it a lot. Upon inspection of leaf constitution it was clear to see that most leafs were not pure, meaning they did not consist of one class. This was to be expected, but pruning and some combinatorial methods like Random Forest would fix that. In the end we believe that the simplicity of the model coupled with the size and type of the data makes it under-perform compared to other methods. It is important to mention that the mention fixes have the capability to increase the model accuracy quite a lot.

To understand the similarity between the accuracy's of the remaining methods, let us first consider the similarities between the two methods. The first similarity is that they are both parametric methods, just like decision trees. The SVM uses parameters of soft-margins and kernel functions, while the neural network (NN) typically uses plenty of different parameters, such as run number, layer number and constitution, learning rate and so on. Both of them also can embody non-linearity by using non linear functions, either non linear kernel functions in case of the SVM or the non linear activation functions, such as the sigmoid we used in our NN. In the literature it is actually stated that given comparable training both methods give comparable results, though NN would be expected to perform better given more training. SVM reacquires less data to achieve good results if the boundaries are good, while NN reacquires more which creates complexity. In essence the two methods are both similar and different, they handle the same concepts in different ways, and their preference would depend on the specific case in question. Research has showed that on large data sets deep learning methods such as NN tend to outperform other methods, while on smaller data sets SVM would perform better (Wang, P., Fan, E. and Wang, P. 2021). Due to the mentioned nature of decision making this is quite logical to conclude. While our data set is not huge, it is also not small, as such, we would expect them to perform in a similar manner which was seen from our results.

5 Conclusion and future work

In conclusion, we have created two learning methods from scratch, one for Decision Trees and one for a Feed-forward neural network implementation. We also created a learning standard implementation of a Support vector machine classifier. The results show that the Decision tree classifier under-performs in comparison to the others, most likely due to the nature of the data, its size and sparseness, as well as the absence of model improvement techniques. Neural network and Support vector machine methods give similar results as we might expect due to the dimensionality of the data and the ability to handle the non-linearity efficiently. Both models could be improved by optimizing their parameters, and the neural network in particular could be massively improved by constructing convolution networks to handle data features in a manner more appropriate for image analysis.

6 Bibliography

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

James, G. et al. (2021) An introduction to statistical learning: With applications in R. Second.

S. Shubathra, P. Kalaivaani and S. Santhoshkumar, ”Clothing Image Recognition Based on Multiple Features Using Deep Neural Networks,” 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC), 2020, pp. 166-172, doi: 10.1109/ICESC48915.2020.9155959.

Byeon, E. (2020) Exploratory data analysis ideas for image classification, Medium. Towards Data Science. Available at: https://towardsdatascience.com/exploratory-data-analysis-ideas-for-image-classification-d3fc6bbfb2d2.

S. T. Roweis and L. K. Saul, —Nonlinear dimensionality reduction by locally linear embedding,‖ Science, vol. 290, no. 22, pp. 2323–2326, Dec. 2000.

Kumar, S., Khan, Z. and jain, A. (2012) “A Review of Content Based Image Classification using Machine Learning Approach,” International Journal of Advanced Computer Research, 2(5).

Wang, P., Fan, E. and Wang, P. (2021) ”Comparative analysis of image classification algorithms based on traditional machine learning and deep learning”, Pattern Recognition Letters, 141, pp. 61-67. doi: 10.1016/j.patrec.2020.07.042.

Implementation of basic ML algorithms from scratch in python... (2023). Available at: https://github.com/Suji04/ML from Scratch

Handwritten Digit Classification With Feedforward Neural Networks (2022). Available at: https://studentsxstudents.com/handwritten-digit-classification-with-feedforward-neural-networks-1aee262ff8dc

Neural Network from scratch in Python (2021). Available at: https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

Building Neural Network from Scratch in Python (2022). Available at: https://itnext.io/building-neural-network-from-scratch-in-python-71ed71d34588

EDA for Image Classification (2021). Available at: https://medium.com/geekculture/eda-for-image-classification-dcada9f2567a

A simple implementation to create and train a neural network in python. (2022). Available at: https://github.com/am1tyadav/Neural-Network-from-Scratch-Python

Canuma, P. (2022) Image Classification: Tips and Tricks From 13 Kaggle Competitions
Available at: https://neptune.ai/blog/image-classification-tips-and-tricks-from-13-kaggle-competitions

Datset, E., Thoma, M. and Cohen, B. (2018) Exploratory Data Analysis with Image Datset, Data Science Stack Exchange. Available at: https://datascience.stackexchange.com/questions/29223/exploratory-data-analysis-with-image-datset

Jauregui, A. (2021) How to code decision tree in Python from scratch Available at:
https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/

Implementation of decision tree classifier from scratch. (2021). Available at: https://github.com/alisherAbdullaev/ML-DecisionTreeClassifier

PCA (Principal Components Analysis) applied to images of faces (2018).
Available at: https://medium.com/@sebastiannorena/pca-principal-components-analysis-applied-to-images-of-faces-d2fc2c083371

GitHub - Decision Tree Algorithm written in Python with NumPy and Pandas (2022).
Available at: https://github.com/harrypnh/decision-tree-from-scratch

GitHub - Decision Tree Implementation from Scratch (2021). Available at: https://github.com/fakemonk1/decision-tree-implementation-from-scratch

Exploratory Data Analysis Ideas for Image Classification (2020). Available at: https://towardsdatascience.com/exploratory-data-analysis-ideas-for-image-classification-d3fc6bbfb2d2