IT UNIVERSITY OF COPENHAGEN

Machine Learning

Project Report

Sneha Shrestha, Katarina Kraljevic, Mathis Valentin Gravil

Group: Sneha, Katarina, Mathis

September 26, 2023

Contents

1 Introduction

In the times of quarantine, online shopping and online e-commerce have gained a massive increase in usage as people were unable to visit the physical stores. The textile industry did not escape that reality. Now that a lot of clothing items can be bought online, we are starting to have massive data sets of clothing available. What if we used that image data to train an algorithm to predict the category of a given item?
In this project, we are trying to achieve exactly that using data provided by the Fashion-MNIST dataset [1]. The code was written in Python and contains five instances of diverse classification algorithms: Two implementations of a Decision Tree, one built from scratch and the other using a pre-built method. Two other replicate Feed-forward Neural Networks (again one built from scratch and the other using a pre-built method) and the last one uses the XGBoostClassifier from the xgboost package.

2 Exploratory Data Analysis

2.1 Introduction to data set

For this project we got access to an image data set containing 15,000 samples. Those images represent clothing items found on the Zalando website. It has been published by Xiao, Rasul and Vollgraf [1]. Each image has been preprocessed beforehand to become a 28x28 grayscale grid. In reality that data was given to us in the .npy format. That format is used to reconstruct NumPy arrays in a binary file format. Once reconstructed we end up with an array of 15,000 rows and 785 columns (the 784 pixel values and one additional for the label of the image) containing integer values.
We have five different categories being t-shirt/top, trouser, pullover, dress, and shirt labeled as the numerical values 0, 1, 2, 3, and 4 respectively. Furthermore, the data set is separated into training and testing data with a splitting ratio of 2/1, in the end we have 10,000 training and 5,000 testing samples. That makes for a reasonable data set size for us to get accurate models trained.

(a) T-shirt/top (b) Trouser (c) Pullover (d) Dress (e) Shirt

Figure 1: Categories of clothes.

2.2 Verification of data integrity

Before using the data to train our models, we want to ensure that the data is "clean", meaning that the classes are evenly distributed, there are no empty data points, and all 15,000 images are labeled. As shown in Figure 2, the distribution of all five classes is even. The missing values where also checked and none were found. Based on this analysis, we can conclude that our data is clean and we can proceed to analyze the relationships between classes using the principal component analysis.

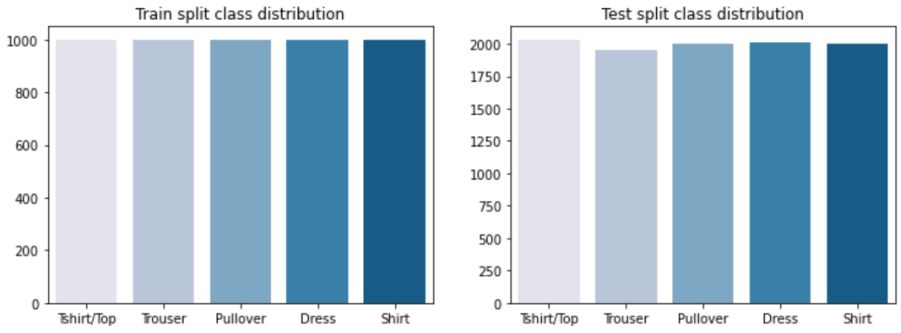

Figure 2: Class distribution for training and test datasets

2.3 Principal Component Analysis

Principal component analysis (PCA) is a technique used to reduce the dimensionality of a data set with many correlated features while retaining the variation present in the data. In our case, we applied PCA to our dataset and plotted the first two components as shown in Figure 3. This allows to visualize the relationships between the features and identify patterns in the data.

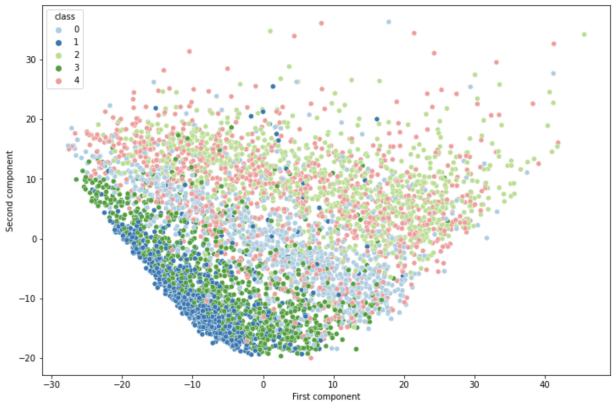

Figure 3: Scatter plot of the first two principal components in training data

The first two principal components depicts that the classes 0, 2, and 4, which are T-shirt/top, pullover, and shirt, respectively share similar features distribution and thus overlap in the first two principal components. This might bring a challenge to the classifiers when trying to precisely label them. On the other hand, class 1 (i.e trousers) is well-separated from the other class which might help the classifiers to easily classify it accurately.

3 Implementation of classifiers

3.1 Neural Network Implementation

The feed forward neural network was implemented using two classes:
1. class **Dense_layer()** - This class takes an input X and produce an output Y as a linear combination of the weights and the input passed through a non linear activation function
2. class **FFNeuralNetwork()** - This class is the training loop of the neural network where training and prediction takes place.

The **Dense_layer()** class is initialised with giving it its parameters i.e the number of inputs, the number of neurons in the two hidden layers and the number of outputs. Upon initialization, the weights matrix and a bias vector is created for each layer based on the dimensions. The weights and biases are initialized from a uniform random distribution and saved to a dictionary for easier access. To make our neural network capable of learning complex non-linear functions, the weighted sum of the input and weights has to pass through an activation function. The

self.ReLu method performs a threshold operation to each input element Z where values less than zero are set to zero when the parameter derivative is set to False (by default) and returns the derivative when the parameter derivative is set to True.
The **self.sigmoid** method is another activation function that takes in real numbers and squashes it to a real-valued output between 0 and 1 when the derivative is set to False (default case).
The **self.softmax** method is used as an activation function in the output layer of the model that predict the multinomial probability distribution individually ranging from 0 to 1 where the sum of all should be 1.
The **self.forward_pass** method computes the weighted sum between the input and the first hidden layer's weights and then add the bias. Then the result is passed through an activation function of choice (ReLu or sigmoid) and the process is repeated for the second hidden layer but by adding weights and biases to weighted sum of first hidden layer. Then the weighted sum of second layer is passed through the soft-max function that returns the predicted output.
The **self.one_hot** method does the job of one hot encoding for the true output labels by preprocessing and converting the categorical labels to be provided for back propagation to calculate the derivatives.
The **self.backward_prop** method calculates the derivatives of the activation function for every parameter with respect to the loss function and saves it to the dictionary. Once we calculate the derivatives, we update the previous weights using **self.update_params** in order to minimize the loss.

Finally, the **FFNeuralNetwork()** class serves as a high level client facing API and takes the number of epochs and learning rate as parameters. The class is initialized with giving a list of dimension of all layers which are all instances of class **Dense_layer()**.
The **self.fit** method is called to train to network which takes two parameters: X (input dataset) and y (labels). For each epoch all batches of size of training set are fed forward through the network and their respective losses are propagated back to update the weights in the network.
The **self.get_prediction** method gives corresponding predictions for the inputs fed forward into the network.

Correctness

Once the neural network was implemented from the scratch, it was trained on the training data set for 500 training epochs. Figure 4 shows the training accuracy history of our custom neural network.

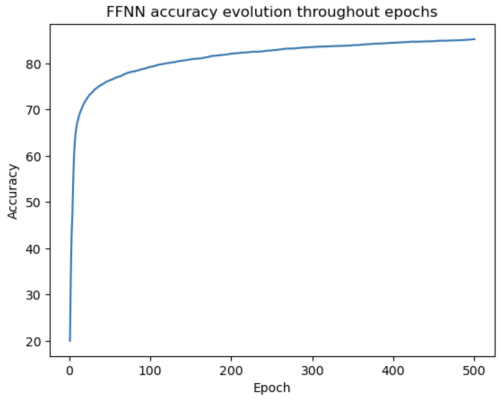

Figure 4: Accuracy evolution of custom neural network throughout epochs

The implemented neural network contains two hidden, sigmoid activated layer of 128 and 64 neurons respectively trained with a learning rate of 0.1. The loss decreases continuously and the training accuracy increases to approximately 85%. In overall the results were generated as expected and to validate the custom implementation, the results were also compared to results generated using neural network built using PyTorch which we will see in the next chapter.

3.2 Decision Tree Implementation

The decision tree was implemented using two classes:

1. class **Node()** - This class is a helper class that stores information about a node later used for the process of traversing the tree.

2. class **DecisionTree()** - This class is responsible for conducting training and prediction for the decision tree through a recursive process.

The **Node()** is initialized with several parameters, including the feature being considered, the threshold value used, and the left and right child nodes connected to it. For leaf nodes, a value parameter is used to check whether a node is a leaf node using the **self.check_leaf** method.

The **DecisionTree()** class is defined with two stopping criteria: maximum depth and the min-

imum number of samples needed for a split, as well as a root node. Once the **self.fit()** method is called, the building process begins by invoking the **self.build_tree()** method. Within this method, details about the dataset are stored and used to check whether the stopping criteria have been met using the **self.is_finished()** helper function. The **self.best_split()** method is then called, which considers all possible thresholds and feature indices to compute the information gain using the **self.information_gain()** helper function, which measures how much uncertainty can be removed with the split being considered. The **self.best_split()** method then returns the best feature and threshold values. The **self.build_tree()** method then splits the data into two subtrees using the best split, and recursively calls itself until a stopping criterion is satisfied, returning all nodes and constructing the final decision tree. The **self.predict()** method takes in a dataset and, for each sample, traverses the tree by following the appropriate branches based on the node's feature and threshold values, using the **self.traverse_tree()** helper function. When a leaf node is reached, the most common class label is computed and used as the prediction, which is stored in the leaf node.

Asserting the implementation's correctness

To determine the accuracy of the decision tree, the model was first trained using the training set. Then, the predictions were made on the test set using the **self.predict()** method. The predictions were compared to the actual labels of the test set data, and it was found that the predictions were accurate for 79% of the test set. To further verify the accuracy of the model, the results were compared with those of a well-known decision tree library, SkLearn, and, as we will see in the next chapter, the results were similar. Overall, these results indicated that the custom decision tree implementation is likely correct.

4 Details on machine learning methods

4.1 Feed Forward Neural Network

Introduction

A feed-forward neural network is a machine learning algorithm that constructs a nonlinear decision function f(X) and approximates the true relationship between some feature matrix X and a target vector y. [2] In each layer of the neural network, an activation function is

applied to ensure the non-linearity of the decision function f(X). [2] Using the technique of back-propagation a loss function is minimised while training a neural network. The gradient of the loss is computed with respect to all the weights and biases which is then used to adjust the parameters that help in reducing the error using a gradient descent step. The entire process is repeated for a desired number of times until the network gives expected results. A response vector y is produced by the output layer using a soft-max function as its activation which makes sure that each respective element ranges from 0 to 1 and sum of the elements in the vector is one. Thus the individual elements in the vector can be treated as probabilities that a data point, when feed forwarded through the neural network, belongs to different classes. The class with the largest probability is then predicted out.

Custom feed forward neural network

The final neural network classifier is a instance of **FFNeuralNetwork()** that was fitted on scaled training split using StandardScaler(). When applied the data into the custom implementation, we expected it to be time-consuming and limited in terms of complex network architectures and number of training iterations. This was proved to be true and can be explained by the recursive back propagation in the network. Therefore tuning the best hyper parameters was mostly done based on our own knowledge and experience and some trial and error. The implemented model is a double, sigmoid activated hidden layers of 128 and 64 neurons respectively, batch size of the training set (batch gradient descent) trained for 500 epochs with a learning rate of 0.1. The network computes the loss using cross-entropy. The fitted model results an accuracy score of 85% for training data and an accuracy score of 83% for test data. Figure 5 shows the classification report of the classifier model for both training and test split.

Training Accuracy score – 85%
Test Accuracy score – 83%

	Precison	Recall	F1-Score
0	0.81	0.85	0.83
1	0.98	0.96	0.97
2	0.85	0.87	0.86
3	0.87	0.91	0.89
4	0.75	0.68	0.71

(a) Classification Report on Training split

	Precison	Recall	F1-Score
0	0.79	0.83	0.81
1	0.98	0.94	0.96
2	0.81	0.85	0.83
3	0.85	0.89	0.87
4	0.70	0.62	0.66

(b) Classification Report on Test split

Figure 5: Performance evaluation on custom implemented neural network

As expected from the analysis of first two principal components, class 1 was classified more accurately in comparison to other classes.

Feed forward neural network implemented using PyTorch

The neural network implemented using PyTorch uses different optimisation algorithms, a different back propagation technique, and initialise the weights differently than the custom implementation [7]. PyTorch neural network has faster running time and more complex network architecture in comparison to custom implementation. Tuning in the best hyper-parameters was done by many trials and errors hoping for the best results. The final model is a double, sigmoid activated hidden layers of 128 and 64 neurons trained for 200 epochs using batch size of 500 (stochastic gradient descent) and a learning rate of 0.01. The model computes the loss using cross-entropy. The loss can be seen continuously decreasing in Figure 6.

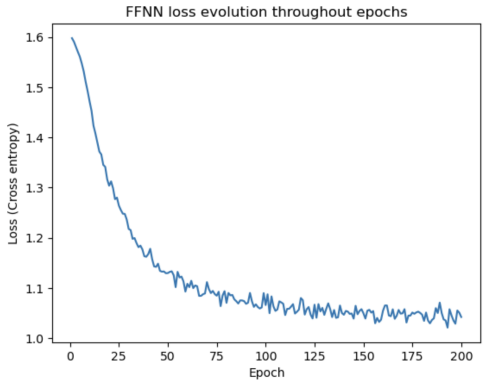

Figure 6: Loss evolution of PyTorch neural network throughout epochs

Training Accuracy score – 89%
Test Accuracy score – 84%

	Precison	Recall	F1-Score
0	0.88	0.88	0.88
1	0.99	0.95	0.97
2	0.84	0.91	0.87
3	0.94	0.95	0.94
4	0.78	0.73	0.76

(a) Classification Report on Training split

	Precison	Recall	F1-Score
0	0.69	0.87	0.77
1	1.00	0.97	0.98
2	0.79	0.90	0.84
3	0.89	0.94	0.92
4	0.83	0.56	0.67

(b) Classification Report on Test split

Figure 7: Performance evaluation on neural network implemented using PyTorch

As shown in Figure 7, the PyTorch fitted model results a training accuracy of 89% and test accuracy of 84%. The model classifies class 1 almost perfectly as its features does not overlap with most other classes.

4.2 Decision Trees

Introduction

A decision tree is a hierarchical data structure used for both classification and regression tasks. [5] It is a non-parametric model consisting of a root node, internal (decision) nodes and leaf (terminal) nodes. The algorithm starts at the root node and continues its path down through the internal nodes, making an evaluation at each step, until an output of a leaf node is reached. This evaluation is an implementation of the divide-and-conquer strategy since, at every decision node, the data is split recursively until classified under one of the class labels.

In the case of a decision tree for classification, the goodness of a split is quantified by an impurity measure. Entropy and the Gini index are both impurity measures used for determining which feature the data should be split on. The goal is to create purer sets of data. Both are maximized for completely mixed sets of data and minimized for pure sets of data.[6]

However, not every dataset can be perfectly classified. Because of its complexity, larger trees can have too little data in a given subset and the purity is more difficult to maintain. Pure leaf nodes are leaf nodes that contain data of a single class only.

Decision trees have some disadvantages as they are prone to overfitting, especially if they are not limited in their depth or properly pruned, meaning that branches splitting on features with

low importance are not removed. However, there are also several advantages including the ability to handle both continuous and categorical data, the ability to visualize the model, and the fact that their training cost grows logarithmically with the size of the training data.

Custom decision tree

The final decision tree classifier is an instance of **DecisionTree()** that was fitting on training split. As expected, the custom implementation of a decision tree was slower and had more limitations compared to the DecisionTree class from the SkLearn library. All the hyper-parameters were chosen manually based on our knowledge and tweaked repeatedly based on the observed results. The final decision tree had a depth of 6 and used entropy as the splitting criterion. The confusion matrix in Figure 8 shows the number of correct and incorrect predictions made by the model for each class. The fitted model has an accuracy score of 83% on the training, and 79% on the test data.

```
[[788   2  20  64 126]
 [ 10 894  11  82   3]
 [ 11   3 817  19 150]
 [ 47  14  16 873  50]
 [169   1 191  70 569]]
```

Figure 8: Confusion matrix of the custom implemented decision tree algorithm

Figure 9 shows the classification report, which supports the finding of the principal components analysis and indicates that class 1 (trousers) was classified with the highest accuracy.

Training Accuracy score – 83%
Test Accuracy score – 79%

	Precison	Recall	F1-Score
0	0.77	0.85	0.81
1	0.95	0.94	0.95
2	0.82	0.89	0.85
3	0.82	0.88	0.85
4	0.78	0.58	0.66

(a) Classification Report on Training split

	Precison	Recall	F1-Score
0	0.75	0.82	0.78
1	0.93	0.91	0.92
2	0.77	0.85	0.81
3	0.78	0.85	0.81
4	0.69	0.50	0.58

(b) Classification Report on Test split

Figure 9: Performance evaluation on decision tree implemented using SkLearn

Decision tree implementation using SkLearn

The SkLearn implementation is likely faster and more efficient due to its optimized code, use of faster programming languages, and incorporation of advanced algorithms and optimization techniques. It also has more complexity and a wider range of hyperparameters. By default, the Gini index was used as the splitting criterion, while the custom implemented used entropy. Although the resulting running time was faster, both algorithms have similar statistical results as shown in the following figures.

Training Accuracy score – 83%
Test Accuracy score – 79%

	Precison	Recall	F1-Score
0	0.79	0.82	0.81
1	1.00	0.93	0.96
2	0.83	0.86	0.84
3	0.82	0.90	0.86
4	0.70	0.62	0.66

(a) Classification Report on Training split

	Precison	Recall	F1-Score
0	0.77	0.79	0.78
1	0.97	0.89	0.93
2	0.78	0.82	0.80
3	0.79	0.88	0.83
4	0.64	0.57	0.60

(b) Classification Report on Test split

Figure 10: Performance evaluation on custom implemented decision tree

The SkLearn implementation of the decision tree achieved the same accuracy as the custom implementation, with scores of 83% on the training, and 79% on the test data. Just like in the custom implementation, class 1 (trousers) was classified most successfully by the model.

4.3 eXtreme Gradient Boosted trees

The last method that was implemented is one of the most popular machine learning model: eXtreme Gradient Boosted trees, or XGBoost. This algorithm is known for being highly efficient, flexible and portable [3]. To optimize computations, the XGBoost package includes a built-in "DMatrix" class which can be converted from a NumPy array. In addition, XGBoost has a built-in cross-validation which will be utilized in this project.

XGBoost utilizes gradient boosting, which involves fitting decision trees and calculating the residuals (the difference between the current predictions and true labels). These residuals are then used to fit a new tree. XGBoost speeds up computation through the use of parallel tree boosting, a technique that involves creating multiple decision trees in parallel.

To achieve the best results and prevent overfitting, there are a few variables in XGBoost that can be modified. For our task of multiclass classification with a single prediction per sample, the "booster=gbtree" and "objective=multi:softmax" combination is recommended.
Before tuning the hyperparameters, it is important to identify the most influential factors. In this case, the learning rate was reduced from the default value of 0.3 to 0.1 in order to reduce overfitting. A function was also used to implement cross-validation and find the optimal number of estimators for the model, as described in an online article by Aarshay Jain [4]. A basic model with default hyperparameter values was fit and used as the baseline for accuracy. The hyperparameters were then tuned to achieve the best possible results.
For the hyper-parameter tuning, GridSearchCV was used for finding the best combination of hyper-parameters. In total there are four major hyper-parameters : "max_depth", "min_child_weight", "gamma", "reg_alpha".

- "max_depth"

  This parameter decides the maximum depth of the trees that are created. A bigger value will have more accuracy because the trees are deeper and more specific. A lower value will cause the tress to be less deep and therefore less specific. For our task we found that a maximum depth of 6 was best.

- "min_child_weight"

  This parameter decides the minimum weight needed for a child to be created. Here in contrary to the first parameter, the lower the deeper the trees will end up being. For our task we found that 2 was the best value.

- "gamma"

  Also known as "min_split_loss" defines the minimum loss reduction needed to split a leaf further down the tree. In our case we found that having no gamma would improve our performance by a very small margin.

- "reg_alpha"

  This parameter defines the regularization that is done on the weights. It works by penalizing the features that increase the cost function too much. Therefore the prediction line becomes smoother and we have less over-fitting. For our task we found that a

After hyper-parameter tuning the model is defined as seen below :

```
XGBClassifier(booster='gbtree', objective='mutli:softmax', num_class=5
, learning_rate=0.1, n_estimators=389, max_depth=6, min_child_weight=4,
gamma=0, subsample=0.8, colsample_bytree=0.95, random_state=42)
```

Once the model is fitting, we get the following training and testing accuracies :

Training Accuracy score – 100%
Test Accuracy score – 87%

	Precison	Recall	F1-Score
0	1.0	1.0	1.0
1	1.0	1.0	1.0
2	1.0	1.0	1.0
3	1.0	1.0	1.0
4	1.0	1.0	1.0

(a) Classification Report on Training split

	Precison	Recall	F1-Score
0	0.82	0.87	0.84
1	1.00	0.97	0.98
2	0.86	0.89	0.88
3	0.91	0.94	0.92
4	0.78	0.70	0.74

(b) Classification Report on Test split

Figure 11: Performance evaluation on XGBoost classifier

5 Interpretation and Discussion

After implementing the five different methods we can now conclude on which one is best suited for this exercise and why, while still considering the flaws and how we could improve our results. We chose to use the accuracy as our main evaluation metric because we were looking for a prediction for each testing sample and we know that our classes are well distributed. But we also know from the Principal Component Analysis that we have links between the classes. That explains why in the classification reports from all classifiers we see that class 1 was most accurately predicted than any other classes. On the other hand, the overlapping classes 0, 2 and 4 had lower precision score.

Based on their performance measured by the expected accuracy, XGBoost was found to outperform any other models evaluated within this project. With 100% training and 87% testing accuracy it predicted 4368 photos correctly. How can we explain those results compared to the other models ?

Knowing what we know (complexity and use cases of the models), it was expected to see a similar result since XGBoost is a decision tree where we use the boosting principal so we should get better results compared to the two methods built on decision trees. FFNeuralNetwork() classifier is a complex model where the added weights and calibrated values reduces the error rate and improves the general performance of the model. This explains why both outperforms the decision trees classifier. The reason why XGBoost gives better results than neural network classifier is probably because XGBoost has more clear architecture while neural network requires a lot of linear algebraic operations with features. Neural networks also requires humongous amount of data to show their relevance.

The results that we have could be improved by using more computational power, time and tuning to squeeze every bit of accuracy possible.

Finally, images are just huge matrices with a very large number of pixels (features) and are very complex in nature. Building and training a classifier to determine the type of clothing from an image correctly is a very tedious and complex task that faces many challenges. The core challenges that we faced while doing so include large number of features with variations in scale that causes computation complexity, and class overlapping. Thus an ideal model for this task should be able to tackle the mentioned challenges.

6 References

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

[2] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. An Introduction to Statistical Leaning - with Applications in R.

[3] XGBoost documentation online source :https://xgboost.readthedocs.io/en/stable/index.html

[4] Aarshay Jain. (2016) Complete Guide to Parameter Tuning in XGBoost with codes in Python.

[5] Scikit Decision Trees online source:scikit-learn.org/stable/modules/tree.htmlclassification

[6] Ethem Alpaydin. Introduction to Machine Learning

[7] PyTorch documentation online source :https://pytorch.org/docs/stable/index.html