“Neural Networks for Fashion Item Recognition”

Authors: **Andreadis Georgios**, f3352101 | **Moscholios Filippos - Michael**, f3352116

Data and Objective of the task

The purpose of this project is the recognition of a fashion item (shirt, coat, bag, sandal etc.). Given a set of images of these kinds of items and their labels (their kind) we will train two neural network models with different approaches, in order to learn to classify as correctly as possible a new unknown fashion item. The data were downloaded directly from Keras. We had at our disposal 6000 images for the training of our algorithms and 1000 for the final evaluation. Our code for the above problem is given here.

Data Exploratory

To begin with, we start by standardizing each pixel for all of our images, dividing them by 255 in order to bring our dataset values into [0,1]. This is a standard technique when it comes to images, in order to standardize them for stability reasons. To verify that the above transformation did not change the images and the task we display the **Figure 1**, containing the images with their corresponding labels that we must predict later.

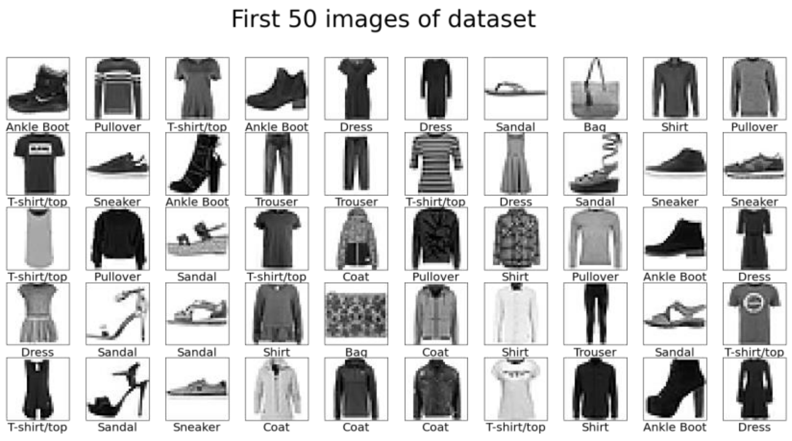


Figure 1. First 50 images of dataset and their class label

Afterwards, we would like to check the frequency of each class across the training and testing subsets. **Figure 2** showcases that the frequency of our classes are equally distributed inside each of the two subsets. Specifically, the training set consists of 6000 images from each item and the test set consists of 1000 images from each item.

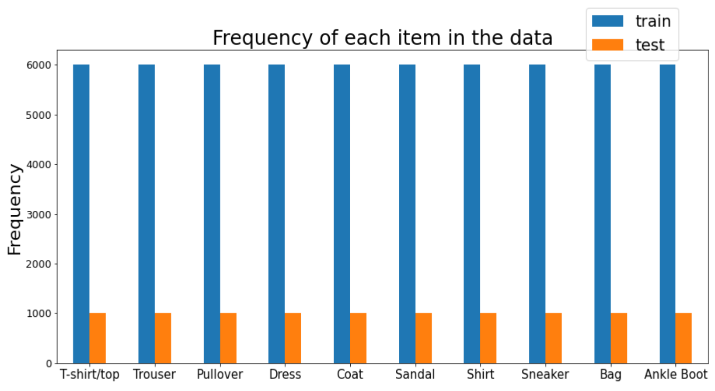


Figure 2: Frequencies of classes across Train and Test sets

For the tuning of the two models that we tried, we needed a validation set, in order to evaluate each time the model. So, we kept a 10% of the train set, as the validation set, and we trained the data only on the remaining 90%, using the rest as an evaluation set, in order to monitor the performance.

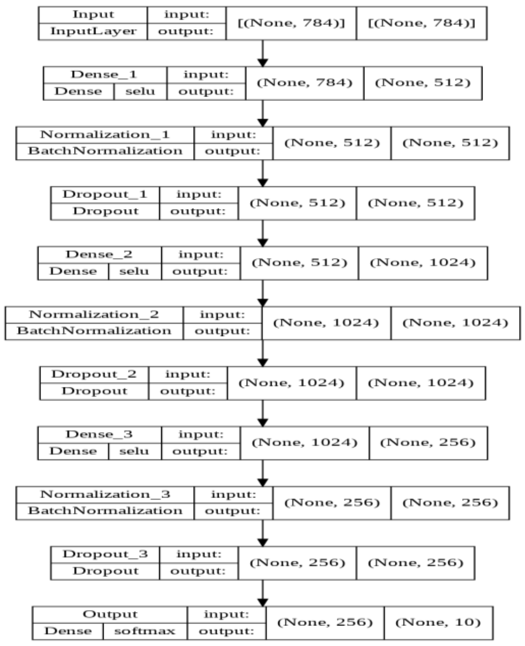
MLP approach

First, we trained a multilayered perceptron (MLP). We feed the model with the images as vectors, instead of matrices, so we need to flatten our images. This Neural Network contains some hidden layers, each of which contains some dense units. After each layer we add batch normalization, which normalizes the data of each batch. This method speeds up the training process, allowing us to pass higher learning rate values. Moreover, batch normalization works as a regularization technique. For regularization, we also use Dropout after each dense layer and the classic Ridge regularization in each dense unit. With these techniques we aim to make our model more robust to the overfitting problem.
In order to find the best model we perform hyperparameter tuning using Bayesian Optimization. We tried to tune only a few hyperparameters, the most significant or the ones that have a bigger impact on the accuracy of the model.
Thus, the search space regarding the hidden layers is the following:

- **Number of hidden layers: from 1 to 3**
  The number of layers of the deep neural network. After some trials we observed that the logistic regression (0 hidden layers) is very simple for our complicated problem. Thus, we decided to start the searching from 1 layer. We also decided to search only until 3 layers of units, because for more than three the algorithm becomes much more slow, without a significant impact to the overall accuracy.
- **Number of dense units of the layers [1024, 512, 256]**
  How many units are needed for achieving high accuracy. At each layer the tuner search from the beginning the number of units, so it is not necessary that two layers of the same model would have the same number of hidden units. We observed that the more the units the better the algorithm in terms of accuracy, but the differencies are not so important to increase a lot this number. So we stopped at 1024 units, because for more than this number the algorithm will become very slow.
- **λ_2 regularization for each dense layer [0.01, 0.001]**
  Implement kernel regularizer with λ_2 regularization, in order to prevent overfitting, but not undermine the accuracy of the validation set. With this approach we set a penalty for big weights of the layers that can force the network to learn extremely well the train data, without the capability to generalize well.
- **Dropout rate for each dense layer [0.1, 0.3]**
  Implement dropout in order to prevent overfitting but tune the rate of dropout in order not to undermine the validation accuracy. Also, a good practice is the rate of dropout in the hidden layers to be less than the rate before the final output layer. This is because; in the beginning we want the network to see almost all data each time, in order to choose a more representative direction towards the minimum. However, in our case, because our task is not extremely complicated, we do not risk much to overfit. Hence, there is no point for doing that in our data.

A major factor affecting the performance of our model is the learning rate of the back propagation of the algorithm. Since we tune in few epochs, the higher learning rate tends to perform better, because the lower one makes smaller steps towards the minimum and thus it takes longer regarding the convergence. If the opposite were true, it would mean that the maximum jumps further ahead than the local minimum. Due to the fact that the space is extremely non convex, this case happens often. For this reason, we use a scheduler, which after a specified epoch we start to slowly reduce the learning rate, so that it makes slower steps to avoid jumping over the minimum and have better convergence. Hence, after the dense layers, we tuned the learning rate of the SGD algorithm. We also tried to use and tune the momentum, which speeds up the learning process to the relevant direction, in order to faster the convergence.

- **learning rate of the algorithm [0.01, 0.001]**
  The learning rate of the batch gradient descent. Afterwards we use a scheduler, with which we adapt customly exponentially the learning rate after some epochs. We observed that the models with learning rate 0.01 perform better than the other ones.
- **momentum [0.5, 0.9]**
  The momentum in batch gradient descent accelerates the gradient descent in the relevant direction, specified by the gradient of the loss, and dampens oscillations. So
  momentum will make the convergence faster, with fewer oscillations around the local minimum.


Input | InputLayer | input: | output: | [(None, 784)] | [(None, 784)]
Dense_1 | Dense | selu | input: | output: | (None, 784) | (None, 512)
Normalization_1 | BatchNormalization | input: | output: | (None, 512) | (None, 512)
Dropout_1 | Dropout | input: | output: | (None, 512) | (None, 512)
Dense_2 | Dense | selu | input: | output: | (None, 512) | (None, 1024)
Normalization_2 | BatchNormalization | input: | output: | (None, 1024) | (None, 1024)
Dropout_2 | Dropout | input: | output: | (None, 1024) | (None, 1024)
Dense_3 | Dense | selu | input: | output: | (None, 1024) | (None, 256)
Normalization_3 | BatchNormalization | input: | output: | (None, 256) | (None, 256)
Dropout_3 | Dropout | input: | output: | (None, 256) | (None, 256)
Output | Dense | softmax | input: | output: | (None, 256) | (None, 10)
Figure 3: Qualified MLP architecture after

We experimented also by tuning the activation gates in order to find better convergence without achieving any significant difference. So, we decided to use the SeLU activation function, which is like the well-known ReLU for x ≥ 0**,** but for x<0 has an exponential behavior, and it does not zero the function. SeLU has the advantages of ReLU and can be efficiently combined with Batch Normalization.
After the hyperparameter tuning, the best model's architecture chosen by Bayesian Optimization is provided in **Figure 3**.

Training and Validation Curves

After having found the best model, we fit it again, for more epochs, using a scheduler which reduces the learning rate after the 10th epoch. In **Figures 4 and 5** we provide the model's accuracy and loss curves, respectively, for this model. We see that in the beginning the validation accuracy is above the training accuracy, which is not logical. As the epochs pass, the training accuracy goes up and the validation accuracy ranges between 0.87 and 0.89, staying under

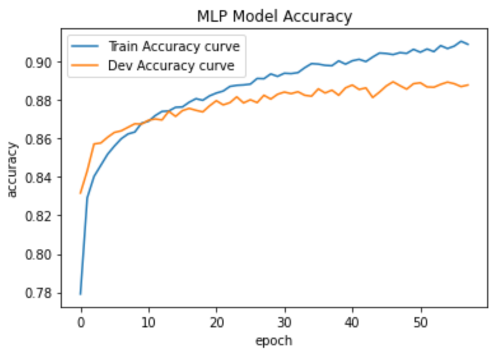


Figure 4. Model Accuracy curves.

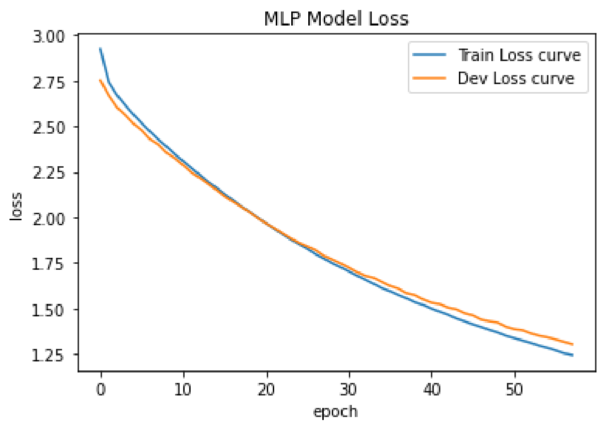
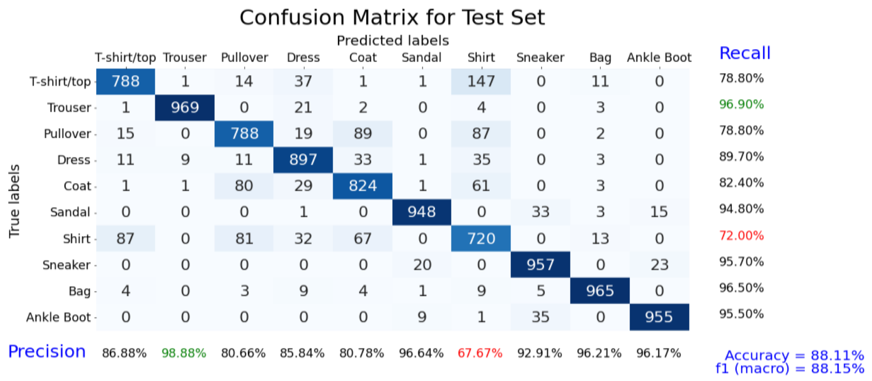


Figure 5. Model Loss curves.

the training. The trend is that the training curve goes up, while the validation one stays in the same values. For this reason we used early stopping, in order to not keep going on epochs if we do not have an improvement for 15 epochs. It seems like the model tends to overfit, but at the 55th epoch the two curves have no noticeable difference. Observing the model loss curves for training and developing data, we can verify the stability of the loss metric, as its curve is very smooth. However, our task is a classification one, so this metric is not suitable for validation purposes. Also, the classes are totally balanced as we saw previously, thus accuracy is a good metric for validation and evaluation. In cases where we had imbalanced classes, precision, recall and f1 score would be more suitable and reliable metrics.

Evaluation of the MLP model

Having fitted the model, we continue with the final evaluation. The **Figure 6** is the Confusion Matrix of the model accompanied with the necessary metrics. Our model manages to correctly predict the Bags, and all the Shoes, Sandals, Ankle Boots and Sneakers. One possible explanation is that these kinds of items do not have a lot in common with the others. Also, we see that the model sometimes confuses the three different kinds of shoes. Additionally, the trousers are not similar with any other class, and thus the model predicts trouser only when it is really a trouser, but sometimes it confuses the dresses with the trousers, which is logical if the dress is long. On the other hand, it seems that when it comes to shirts, dresses, tops and pullovers, the model makes a lot of mistakes. The category with the worst rates of correctly predicted images is the one with Shirts. This happens because, if we observe the images of shirts, they include a lot of different styles of shirts that are very similar with other kinds, such as tops or dresses.



Confusion Matrix for Test Set

True labels \ Predicted labels	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle Boot	Recall
T-shirt/top	788	1	14	37	1	1	147	0	11	0	78.80%
Trouser	1	969	0	21	2	0	4	0	3	0	96.90%
Pullover	15	0	788	19	89	0	87	0	2	0	78.80%
Dress	11	9	11	897	33	1	35	0	3	0	89.70%
Coat	1	1	80	29	824	1	61	0	3	0	82.40%
Sandal	0	0	0	1	0	948	0	33	3	15	94.80%
Shirt	87	0	81	32	67	0	720	0	13	0	72.00%
Sneaker	0	0	0	0	0	20	0	957	0	23	95.70%
Bag	4	0	3	9	4	1	9	5	965	0	96.50%
Ankle Boot	0	0	0	0	0	9	1	35	0	955	95.50%
Precision	86.88%	98.88%	80.66%	85.84%	80.78%	96.64%	67.67%	92.91%	96.21%	96.17%	

Accuracy = 88.11%
f1 (macro) = 88.15%

Figure 6. MLP Confusion Matrix on the final evaluation on unseen data.

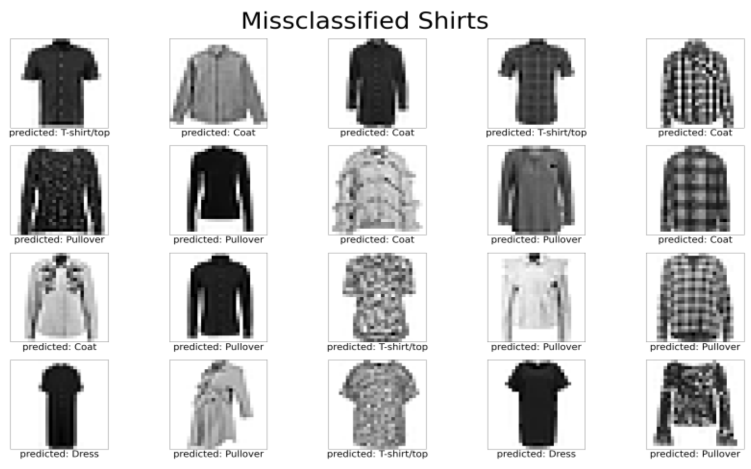


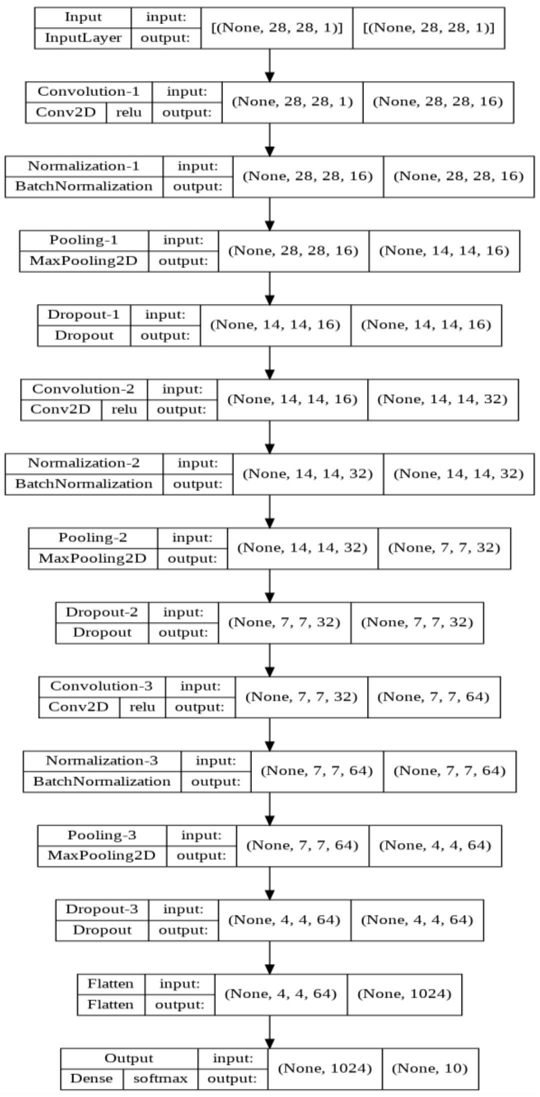
Figure 7. Shirts that MLP classified wrong and their predicted label.

For this reason, we depict some of the misclassified shirts in the next figure.
Although MLPs are very common Neural Networks, we often use a different approach when we have an image related task. This approach is the Convolutional Neural Networks, which often works better with images than the MLPs, and are more efficient with less weights to learn.

CNN approach

Convolutional Neural Networks are extremely useful for capturing patterns in images or texts. Using filters of sizes 3x3 or 5x5, we can search for specific patterns in neighborhoods of pixels in images. After a convolution layer, we use a Max Pooling (we could use Average Pooling instead), which takes the dominant pixels according to the specific kernel. With this method, after each layer the image shrinks, while the third dimension (containing the filters) expands. We use Batch Normalization, as we did in the MLP, and Dropout to prevent overfitting. Regarding the network layers, we chose to tune only the number of filters in each convolutional layer and the dropout rate. Other than that, we tune the learning rate of the Batch Gradient Descent. For the learning rate we talked about previously.
Hyperparameter tuning:

- **Number of hidden layers: from 1 to 3**
  How deep our network needs to be in order to achieve high accuracy in the validation data, meaning how many convolution layers we need. We select from 1 layer up to 3 layers. We cannot have more than 4 layers, because our images are 28x28 and with pooling with size 2x2, the image has the half the size of the previous layer each time. Also, we chose to search for less than 4 layers, due to limited time and resources
- **Number of filters of the Convolution layers [8,16]**
  How many filters are needed at each convolution layer. The trend at CNNs is that the deeper we go, the more filters are used. So gradually we shorten the width and height of the image, and we raise the third dimension (the filters). For this reason, we tuned only the filters for the first convolution layer. For the next convolution layerσ we used the formula filters×2^{i}, where i is the layer we are in (starting from i = 0), and filters the number of filters chosen by the tuner for this trial at the first hidden layer. So, if filters = 8, the first layer would have 8 filters, the second 16 and the third 32. We decided to not try for 4 filters, because



Input | InputLayer | input: | output: | [(None, 28, 28, 1)] | [(None, 28, 28, 1)]
Convolution-1 | Conv2D | relu | input: | output: | (None, 28, 28, 1) | (None, 28, 28, 16)
Normalization-1 | BatchNormalization | input: | output: | (None, 28, 28, 16) | (None, 28, 28, 16)
Pooling-1 | MaxPooling2D | input: | output: | (None, 28, 28, 16) | (None, 14, 14, 16)
Dropout-1 | Dropout | input: | output: | (None, 14, 14, 16) | (None, 14, 14, 16)
Convolution-2 | Conv2D | relu | input: | output: | (None, 14, 14, 16) | (None, 14, 14, 32)
Normalization-2 | BatchNormalization | input: | output: | (None, 14, 14, 32) | (None, 14, 14, 32)
Pooling-2 | MaxPooling2D | input: | output: | (None, 14, 14, 32) | (None, 7, 7, 32)
Dropout-2 | Dropout | input: | output: | (None, 7, 7, 32) | (None, 7, 7, 32)
Convolution-3 | Conv2D | relu | input: | output: | (None, 7, 7, 32) | (None, 7, 7, 64)
Normalization-3 | BatchNormalization | input: | output: | (None, 7, 7, 64) | (None, 7, 7, 64)
Pooling-3 | MaxPooling2D | input: | output: | (None, 7, 7, 64) | (None, 4, 4, 64)
Dropout-3 | Dropout | input: | output: | (None, 4, 4, 64) | (None, 4, 4, 64)
Flatten | Flatten | input: | output: | (None, 4, 4, 64) | (None, 1024)
Output | Dense | softmax | input: | output: | (None, 1024) | (None, 10)

Figure 8. CNN architecture of the best model.

after some manual tuning we see that it does not give so good results, and we wanted to save time and resources.

- **Dropout rate for each convolution layer [0.1, 0.2]**
  Implement dropout in order to prevent overfitting, but tune the rate of dropout in order not to undermine the validation accuracy. So, we search between no dropout at all, and 20% dropout
- **Learning rate of Gradient Descent [0.01, 0.001]**
  The learning rate of the batch gradient descent. Learning rate of 0.01 will speed up the gradient descent, with the risk of not converging, whereas a rate of 0.001 while not jumping above the local minimum, but the method will converge more slowly. Afterwards we use a scheduler, with which we adapt customly exponentially the learning rate after some epochs. After some trials, we observed that the tuner tend to perform better for the bigger learning rate, for 30 epochs. This is because, the convergence with learning rate of 0.001 is much more slow, so we would need much more epochs, which means much more resources and time for our tuning. Also, a good possible practice may be to not use the scheduler for the small learning rate, and compare the models with a big initial learning rate which decreases after some epochs and with a small initial learning rate which stays as it is.

After the hyperparameter tuning, the best model chosen by Bayesian Optimization Tuner is **Figure 8**. We provide its architecture.

Training and Validation Curves

For this part we provide the accuracy and loss curves of the CNN model in **Figures 9 and 10**. Observing the training and developing curves for accuracy and loss, we notice that many peaks appear in the validation curve. This is due to the complexity of the model and more analytically to the dropouts and other regularization techniques we apply to prevent overfitting. The behavior of the developing curve during training suggests that, around the 40th epoch, we obtain the best results for accuracy and loss respectively. However, even though training and validation after the 40th epoch seems to move away from the training curve leading us to overfitting. We are not sure that with more resources and hence more epochs the developing curve could approach the training curve.

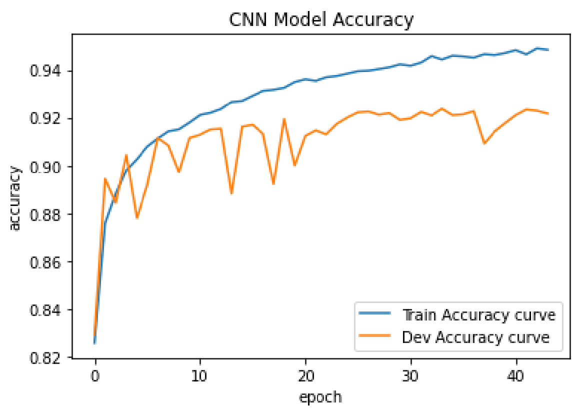


Figure 9. CNN accuracy curves.

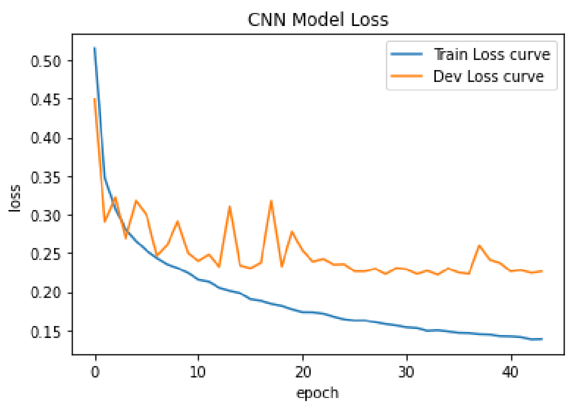
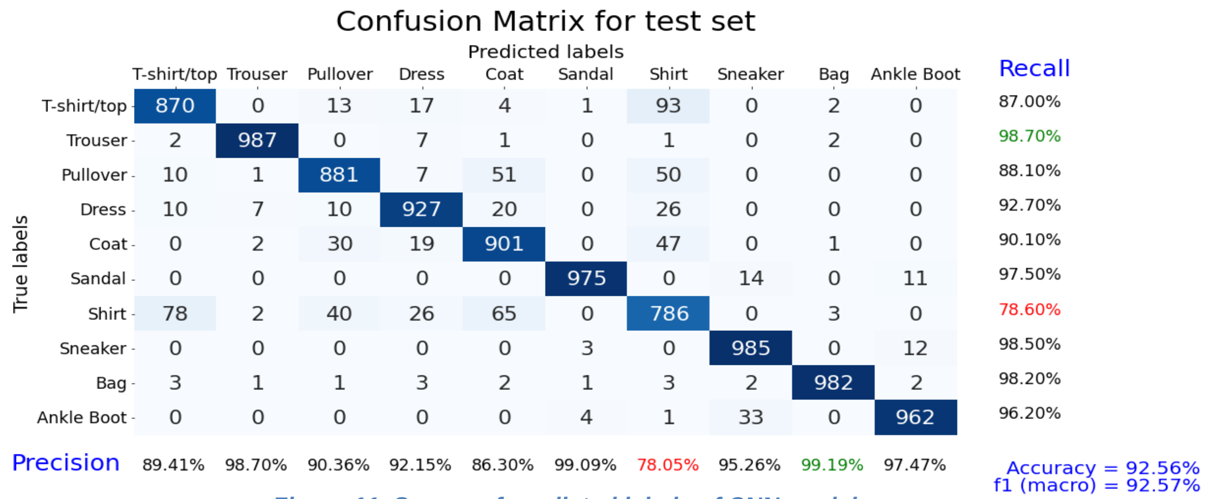


Figure 10. CNN loss curves.

Evaluation of the CNN model

Having fitted the model, it is time for the final evaluation. The following Figure is the so-called Confusion Matrix **Figure 11**, accompanied with the necessary metrics. We have high-lighted the metrics and the respective classes where the model has the best and the worst



True \ Predicted	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle Boot	Recall
T-shirt/top	870	0	13	17	4	1	93	0	2	0	87.00%
Trouser	2	987	0	7	1	0	1	0	2	0	98.70%
Pullover	10	1	881	7	51	0	50	0	0	0	88.10%
Dress	10	7	10	927	20	0	26	0	0	0	92.70%
Coat	0	2	30	19	901	0	47	0	1	0	90.10%
Sandal	0	0	0	0	0	975	0	14	0	11	97.50%
Shirt	78	2	40	26	65	0	786	0	3	0	78.60%
Sneaker	0	0	0	0	0	3	0	985	0	12	98.50%
Bag	3	1	1	3	2	1	3	2	982	2	98.20%
Ankle Boot	0	0	0	0	0	4	1	33	0	962	96.20%
Precision	89.41%	98.70%	90.36%	92.15%	86.30%	99.09%	78.05%	95.26%	99.19%	97.47%	

Figure 11. Scores of predicted labels of CNN model.

performance. Our model manages to correctly predict the Trousers, the Sandals, the Dresses, the Ankle boots, the Sneakers and the Bags. For the rest classes we observe that the model is going a little worse, excepting the Shirt class in which we notice that the model is more difficult to predict. The reasons why the model achieved higher classification performance were analyzed at the MLP confusion matrix section. The difference here is that we deal with a more complex model, which is made to generate features at the training process, which contain more details and that is the major factor why we achieve an increase even in the classes that were difficult to identify with the previous model. To verify the difficulty of our less accurate class' classification we provide a depiction of misclassified Shirts and the labeling that our model gave **Figure 12**.

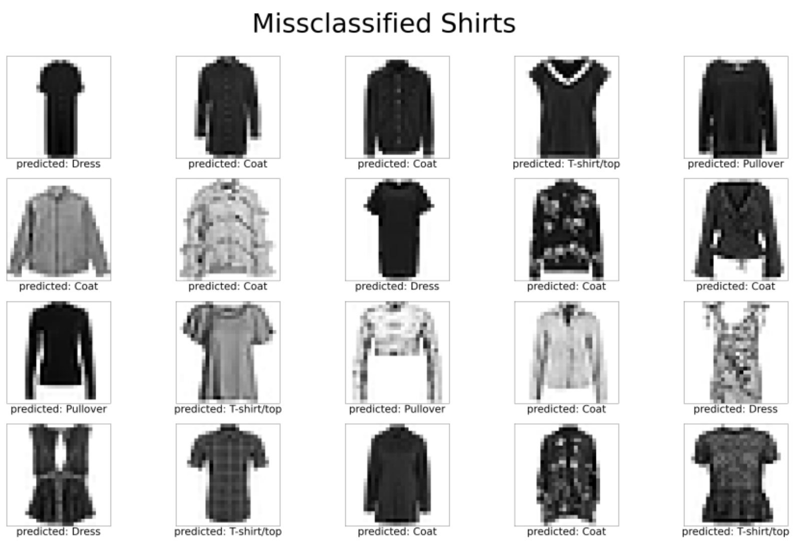


Figure 12. Shirts that CNN labeled wrong.

Comparison of results

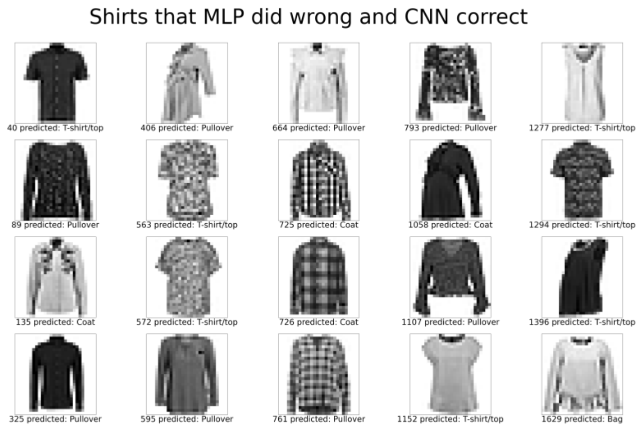


Figure 13. Shirts that CNN corrected the MLP predictions.

Taking everything into consideration, the best MLP achieved a training accuracy of 88.1% with the less accurate class of shirts spotting it with approximately 70%. CNN achieved a 92.6% accuracy with the least accurate class of shirts at 76%. We understand from this that indeed this problem needs a more complex architecture to solve it more effectively. Thus, we illustrate In **Figure 13** some images of shirts, where MLP confused, but CNN manages to label correctly. Moreover, we should mention that for the MLP the best detected class was the Trouser with approximately 98% accuracy and for the CNN the best detected class was the Bag with 99% accuracy. Finally, we observe that for the rest classes CNN achieving higher score than the MLP, for instance the TOP/T-SHIRT class MLP achieving a score of 78% while CNN achieving 87% respectively.

Conclusion, further work and proposals

Concluding, because our two models performed well enough in general, we thought that it could be a good idea to combine them inspired by the logic of ensembling. First in order to examine whether it is worth combining them, we measure the Kendall correlation among the predictions of our two models. Given a correlation of 89% we expected the combination to result in a non-statistical difference. Hence, there is no point for an ensemble model. We evaluate this observation, by running a voting model, and seeing that the overall accuracy is somewhere between the two models. However, if we were interested in one specific class (for example the Shirts, for which both our models make a lot of mistakes), it would be a good idea to test if a combination of the two models fix a lot of misclassified shirts. Furthermore, whether to use or not a voting schema it depends also on the nature of the task and data. If the classes in our data are imbalanced, then It would be beneficial for the rare class the usage of an ensemble model. For instance, for a problem which the subject is to predict cancer, a small little improvement on the class of positive in cancer is more crucial than a small drop on the predictions of the negative class.

For future work something we strongly believe that would lead us in higher performance is transfer learning. With this method, we load a pre-trained model like Resnet 50 which is already trained in numerous, and on top of this, we add a task-specific Neural Network, to use this model for our data. In the beginning we freeze the weights of the pre-trained model, and we adjust only the weights of our task-specific neural network. As we proceed, we defrost the weights, and we train the model as a whole. In many cases, this method is extremely beneficial. Although, in our task is not so obligatory, as it is not so complicated.