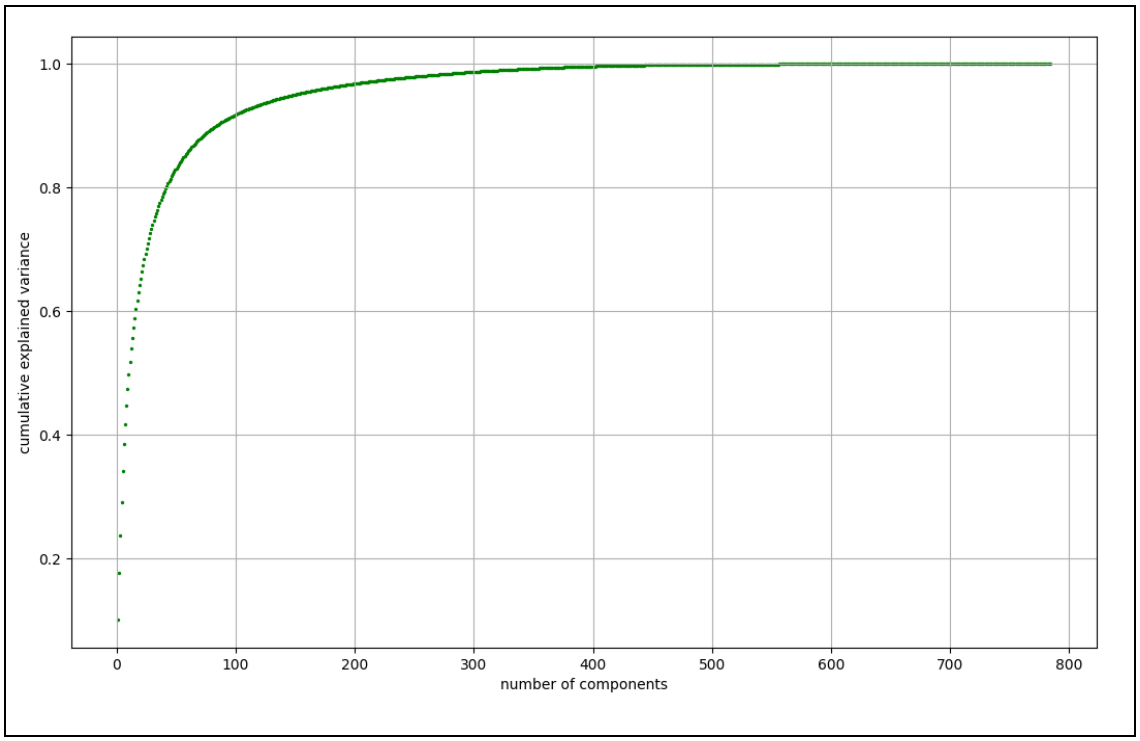
Aim

1. Choose the optimal number of principal components in the MNIST dataset. Show the graph on which the decision was made.
2. Based on the existing perceptron code, build a classifier for a reduced-dimensional dataset.
3. Train it, compare the performance indicators with the previously trained perceptron for the original dataset.
4. Draw a conclusion about the impact of dimensionality reduction using PCA on the quality of classification.

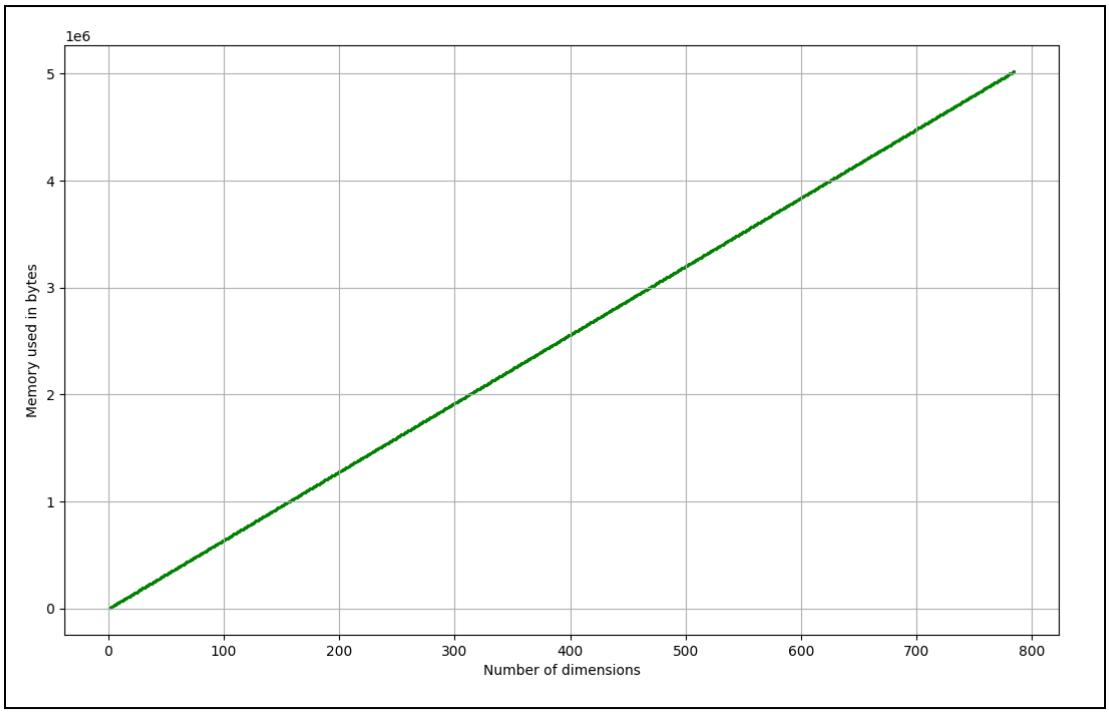
Principal components

Having a MNIST handwritten numbers dataset, we are able to see how much of data (variance) is saved in dependency of dimension reduction of our input vector (8x8). PCA - Principal Component Analysis method (Sklearn library in Python) will be used to perform this experiment.

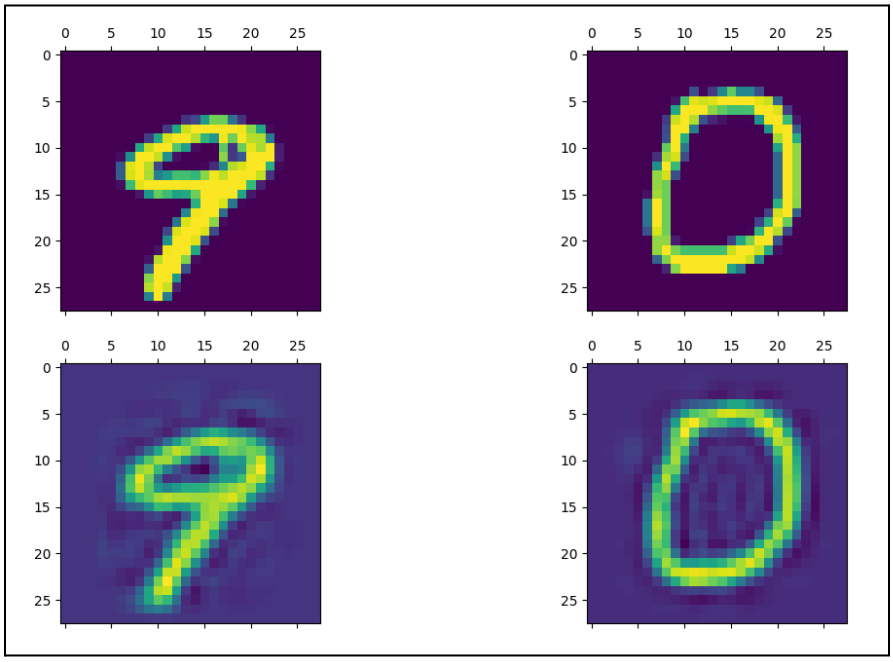
We've defined a reduced dimension dataset and computed cumulative variance for each number of dimensions. Values on the Y axis represent a percentage of data saved after dimension reduction. Although our initial vector is almost half sized, more than ~0.995% of data is saved. In the case of 70 main components, the loss will be about 10%.



On the next graph we can observe a comparison between number of dimensions and Dataframe with 800 entries (numbers) memory usage. The dependency is straightforward linear. Half of initial vector size leads to half reduction of memory use.



It is reasonable to try to reduce down to 100 inputs, the loss of data in percentage will be less than ~10 while it will be used 8 times less memory for the data.



A comparison between handwritten numbers before and after dimension reduction down to 100 elements/transforming back is visualized in the picture.

New classifier

Having our initial perceptron code (hand-written numbers recognition), we'll rewrite it to have fewer inputs. It should look following one:

```
# Use PCA to reduce dimension of input data
N_COMPONENTS = 100
pca = PCA(n_components=N_COMPONENTS)
pca.fit(train_images)
train_images_pca = pca.transform(train_images)
test_images_pca = pca.transform(test_images)
```

Here, the dimensions of our train and test dataset were reduced and those arrays were transformed from the shape of [numbers_total x 784], to [number_total x 100]. The attribute 'feature_vector_length' of our model instance

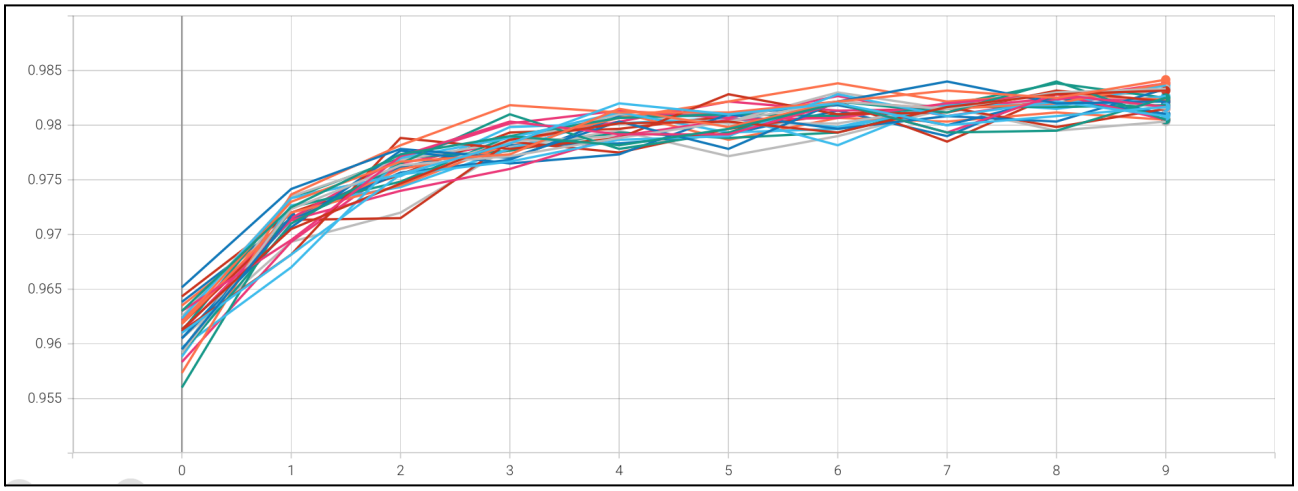
was also changed to 100. To make our model work properly, the number of nodes in our hidden layers must also be reduced to 50 and 30, from 350 and 50.

Training and testing

In some of the previous researches, we've already defined a proper combination of hyperparameters for our N.N. They are:

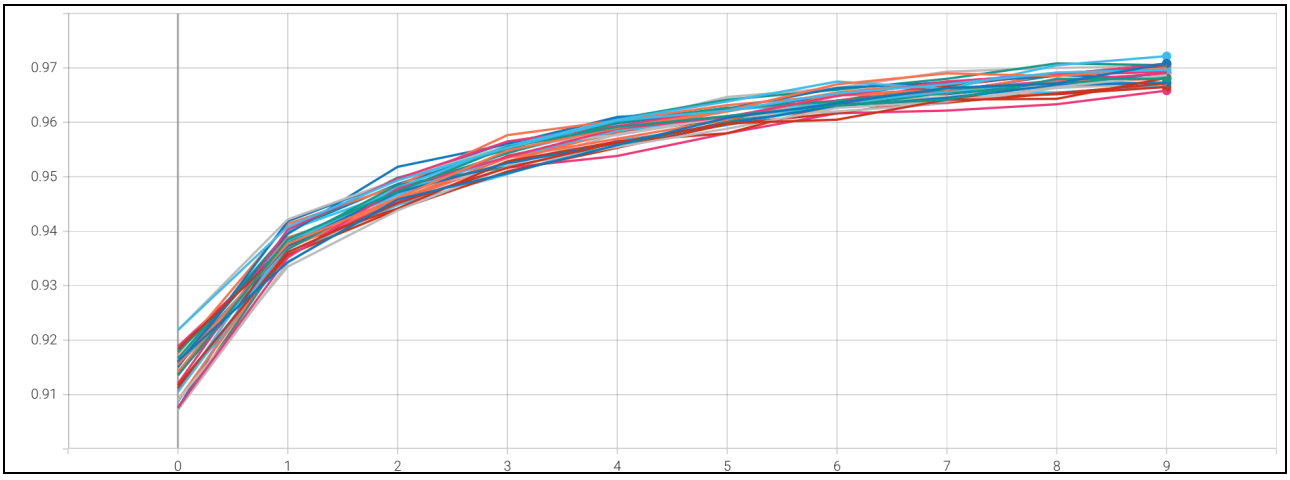
- Learning rate = 0.1
- Batch size = 250
- Dropout = 0.179
- Activation function = 'relu'
- Optimizer = 'Adam'

To make a comparison, firstly we ran 30 model creations and evaluations with a dimension of 784 of our input data. The next graphs shows the results:



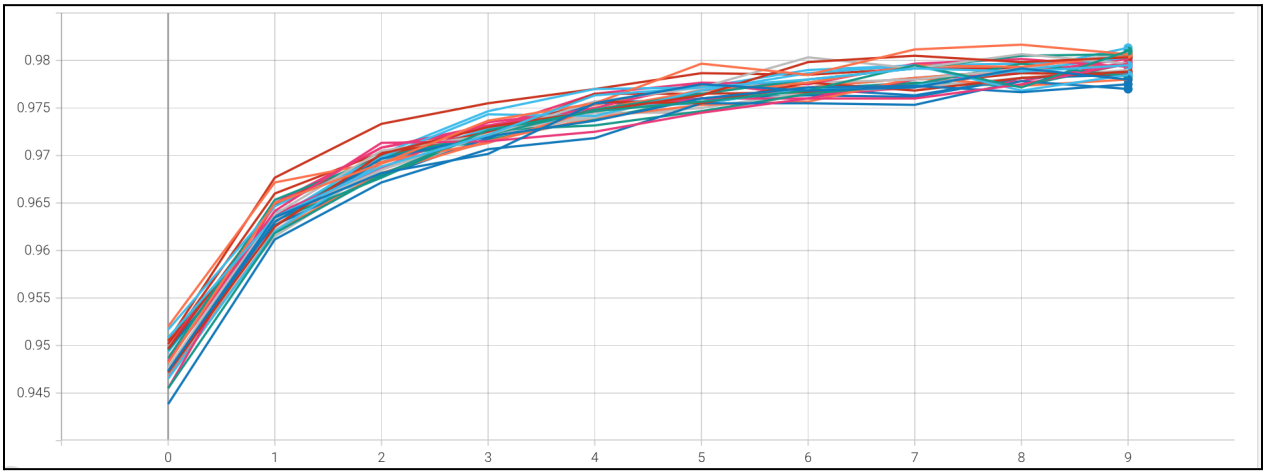
It is clearly noticeable that all evaluation values on the 10th epoch lay between 0.980 and 0.984, which is a quite good index for our model. However an average accuracy on the test data is ~0.9812, and a mean of elapsed time for the model creation and training is 19.6 seconds.

After we have reduced our dimension to 100 results were not surprising.



Although the plot lets us know that overall performance on the last epoch has dropped and now is located between ~0.965 and ~0.971 (avg. accuracy on test data is 0.960), the average time spent on model training and evaluation is 2.6 times less, with a value of ~7.3 seconds. It is not hard to see that all the trials here have a more similar pattern, while we could have seen much overfitting on the previous graph. Though it could be a result of the decrease in the number of nodes, on the other hand, our images now have less noise (which can lead to overfitting), so it's controversial.

We've decided to run a model with 400 inputs also, to find out a more accurate difference.



The minimum value of the 10th epoch on the graph is ~0.976 and maximum is ~0.981, - that is something in between our two previous tests. The average performance on the test data is ~0.9755. Like it was expected, elapsed time dropped by 1.3 times with a value of ~14.6 seconds.

Conclusion

To sum, we can say that it’s worth it to compress the input data. After a reduction of our vector by 8 times, the performance of the model dropped by 2.2% of initial one. After cutting a half, it decreased by 0.7%. The time spent on the model evaluation and training decreased linearly (7.3 for 100, 14.6 for 400). In production conditions a fast response is an important part, so it’s okay to give away 2 percent for the lighter weighted and faster MLP model.