BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Voice and image recognition
Project SUR

Bc. Jan Holáň, xholan11@stud.fit.vutbr.cz
Bc. Matej Horník, xhorni20@stud.fit.vutbr.cz

April 22, 2024

Contents

1 Used libraries and technology

The code was written in **Python 3.11.7**.

Library **ikrlib.py** has been used while working on some models taking part in voice recognition *(available from https://www.fit.vutbr.cz/study/courses/SUR/public/projekt_2023-2024/)*.

Neural networks for both tasks were trained with use of library **pytorch** *(pip install torch)*.

Other libraries used in project are **numpy** *(pip install numpy)*, **matplotlib** *(pip install matplotlib)*, **scipy** *(pip install scipy)*.

2 Voice recognition

2.1 Intro

The PCA, LDA and GMM models were created according to the above (and recommended) example implemented in the demo_genderID.py file. The implementation of these parts of the code in the example was modified to match the requirements of python 3.11.7, and at the same time changes were made to some parameters to determine the best model settings.

Next, convolution neural network was created and trained.

For training of each model were used data target_train and non_target_train. For testing then data target_dev and non_target_dev.

Since the previously mentioned parts of LDA, PCA and GMM have been implemented, we focused on the our own neural network, which is also described more precisely in this document.

2.2 Data preprocessing

Each recording was truncated by 1.5 seconds because (we were told) it contained heavy noise and a moment of silence at the very beginning of the recording. This was done both in code in voiceRecognition.ipynb, where it was used while loading data for training of neural network, and also in the library ikrlib.py in function wav16khz2mfcc, which has been used while working on PCA, LDA and GMM.

2.3 PCA

This method shows, that data are almost inseparable, no matter how many eigenvectors are used. As the picture 1 proofs, no direction is able to be used in this space to separate the target (blue) and non-target (red) data. This situation repeats in some form as we tried to use any possible different number of eigenvectors. As no wise threshold can be found, no test has been done there.

2.4 LDA

This method was more accurate and usable than PCA. In Figure 2, we can see that the training data is more reasonably distributed in a two-dimensional space. This model also provides better results - the threshold was set to 0 and the target data was detected with 100% accuracy and the non-target data with 85% accuracy. Remarkably, the model (also from the demo_genderID.py example) trained only as if the data came from two Gaussian distributions provided even better results on the non-target data with 88.3% accuracy.

2.5 GMM

A GMM model using three Gaussian components for target data and four for non-target data was found to be the best. The more components were set, the more the model was over-trained. 100 training runs were set - the log-likelihood of the target data stopped increasing after about 40 iterations, but the likelihood of the non-target data reached this point after about 100 iterations. Although the graph in figure 3 shows a significant learning rate only in the first 15 iterations, the accuracy of the model improved by approximately 3% at 100 training runs, so we decided to take the risk of potential over fitting and let the GMM train on 100 cycles. The model provides good results at this setting - 100% accuracy in classifying target data and 93.6% accuracy in classifying non-target data.

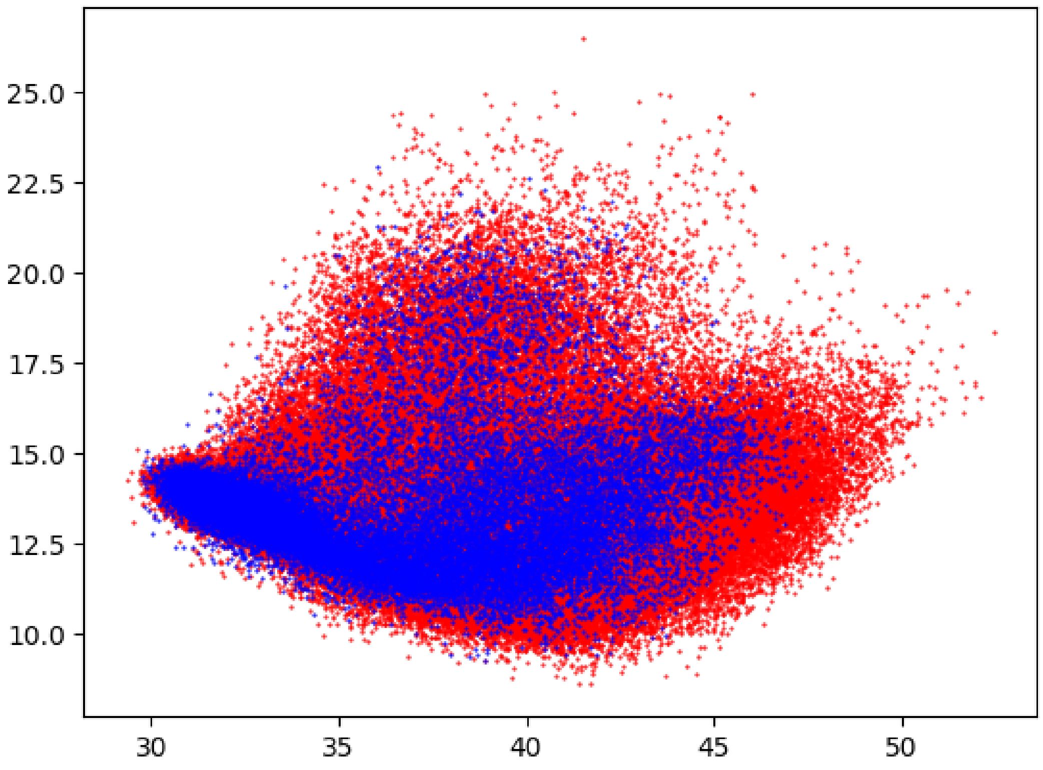

Figure 1: PCA voice recognition data segmentation

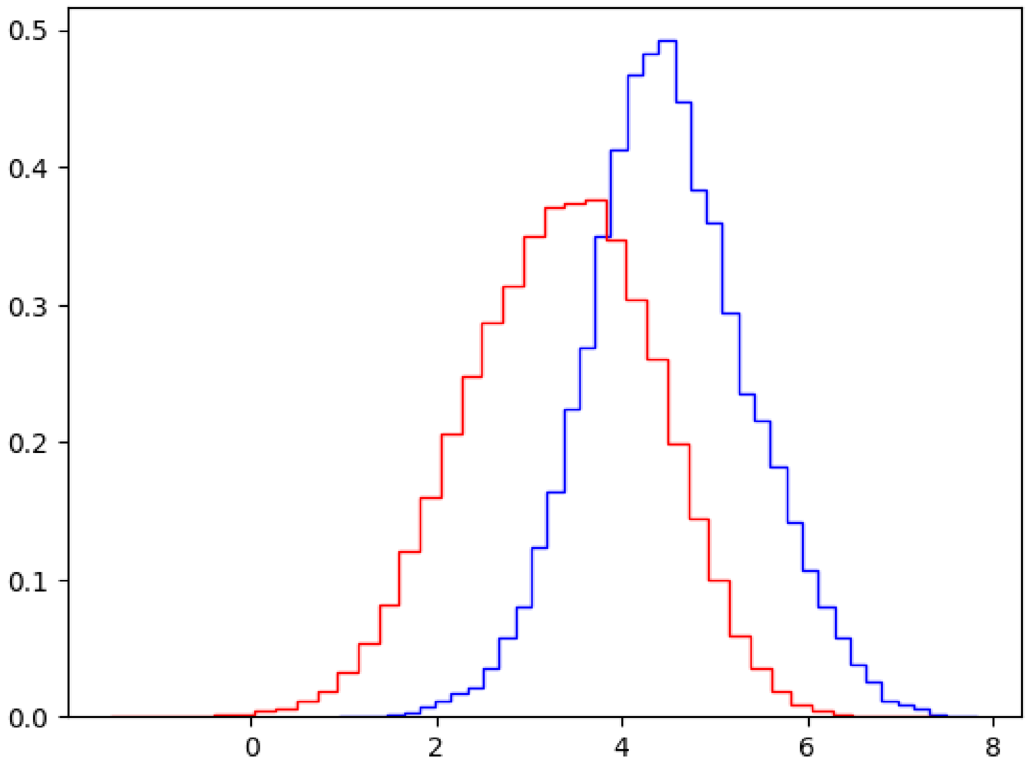

Figure 2: LDA voice recognition data segmentation

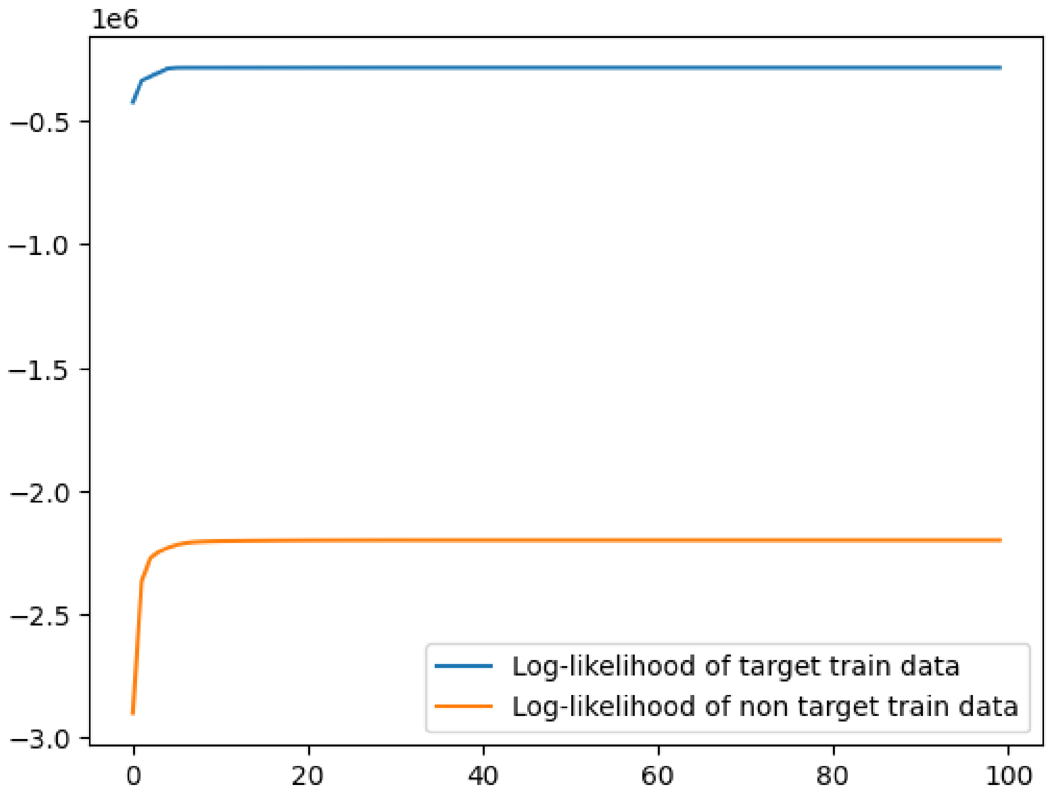

Figure 3: GMM log likelihood

2.6 Neural network

We build neural network as convolution NN. The network has five layers - three convolutional (it shows up, that the second one, which was added later, has significant impact on the whole network in this case) and two linear. The output is array of two numbers that describe the assurance of which class does the voice belong to.

We use spectrograms of the input data as input to the network. This approach led to the need to cut off all data that was longer than 3517 time units - obtained from the *scipy.io.spectrogram* function (this number comes from the longest record in the training data set). We also tried both normalized and raw data, and the normalized provided better results.

We chose SGD (Stochastic gradient descent) as the optimizer - it is associated with parameters such as the learning rate and the weight decay mentioned below.

The loss function of our model is Cross Entropy. We chose it because we are dealing with a classification problem. An important point of our solution was also to resolve the imbalance between the number of target (20) and non-target training records (132) by way of setting weights for both these parts of the dataset. Therefore, we set the weights for the target training set as a parameter of the *CrossEntropyLoss* function to 132/152 for the target data and 20/152 for the non-target data, thus balancing it. This had a large impact on the network result (without this, the network failed to recognize the target records after a few epochs).

We made some experiments, which distinct in set of parameters like weight-decay, learning rate and gradient clipping. The final version of parameters with relatively good result (of 80.0% accuracy on non-target-dev set and 100% accuracy on target-dev) set was:

- Learning rate: 0.0001
- Weight-decay: 0.005
- Gradient clip: 0.0005

The graph 4 below shows how the loss function went while training the NN.

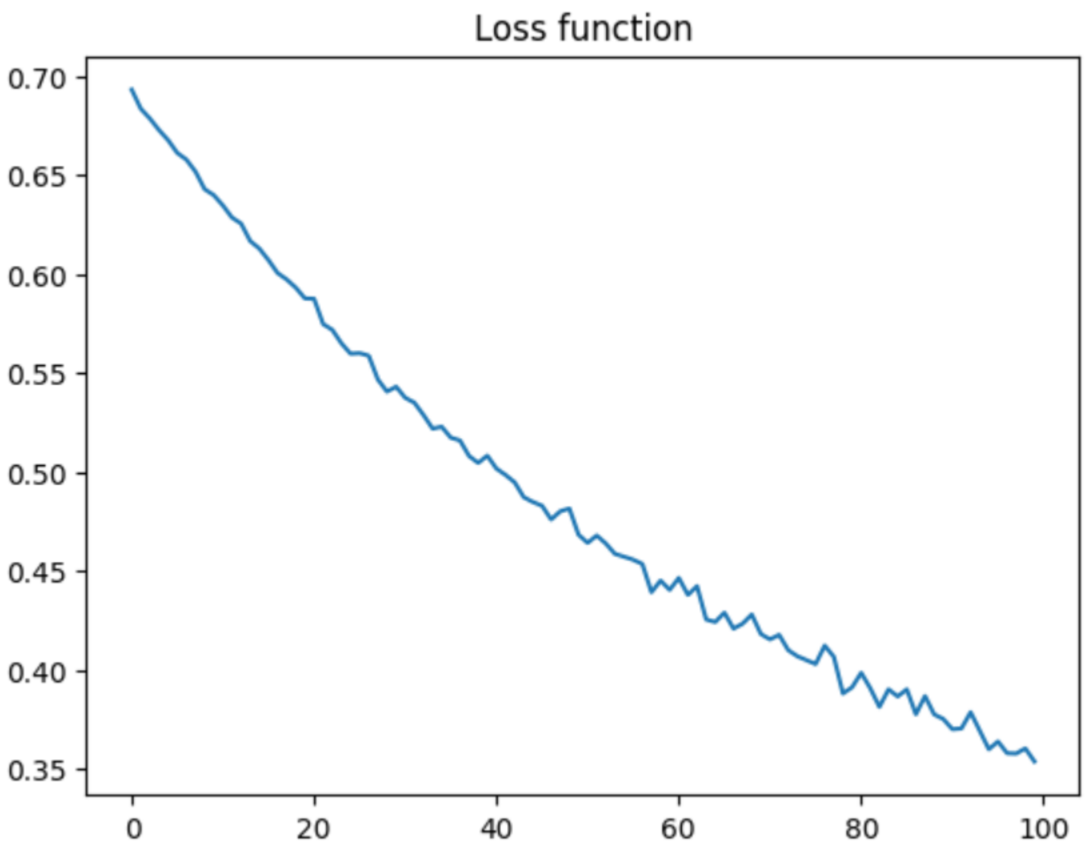

Figure 4: Neural network - loss function

It must be said, that the network was trained on only 100 epochs. Any higher number of epochs made the model seem overtrained (we tried to create augmented data, but abandoned this idea because the network starts to deteriorate when training on them - so this task could be definitely improved).

3 Face Recognition

3.1 Dataset

The dataset contains two classes: `target` and `non_target`. We aimed to create a classifier capable of accurately classifying the data into these two classes. The image data consists of three channels in RGB format, with each image's size being 80x80 pixels. All images are close-ups of faces.

Dataset Details:

- **Training Set**: 151 examples
  - **Target Class**: 20 examples
  - **Non-Target Class**: 131 examples
- **Test Set**: 70 examples
  - **Target Class**: 10 examples
  - **Non-Target Class**: 60 examples

3.2 Introduction

In using images to classify the target, we decided to only use neural networks, as they are suitable for images containing spatial information. The main problem we anticipated was the low amount of data, which could easily lead to overfitting and difficulties in generalizing to test data. We addressed this issue by employing data augmentation.

3.3 Models

When deciding which models to use for these images, we opted for slightly more advanced architectures than simple convolutional layers. The first model we tried was *ResNet9*. With just a few basic data augmentation transformations, the model performed well. We also experimented with other similar architectures to see if we could further improve performance.

Model Architectures:

- **ResNet9**: 6,571,779 parameters
- **PreActResNet18**: 11,167,042 parameters
- **ConvMixer768_20**: 13,249,538 parameters
- **ConvMixer768_32**: 21,129,218 parameters

When testing the *ConvMixer* architectures, we noticed they weren't performing well with few epochs compared to the *ResNet* architectures, so we discontinued their use.

3.4 Training and Hyperparameters

The code for training the models is located in *src/trainmodel.py*. We used a simple optimizer and did not experiment with other optimizers, as we found Stochastic Gradient Descent (SGD) provided satisfactory results. We utilized SGD with momentum. Instead of a learning rate scheduler, we manually lowered the learning rate for further training and improvements. The value to which we lowered it was determined through testing and experimentation to avoid large spikes in train or test loss.

We opted to use weight decay in our training to regularize the loss, given the small size and lack of diversity in the dataset. Additionally, we employed gradient clipping to further regularize the training process. We experimented with different values for both parameters to determine the most effective settings. We also experimented with batch size initially and settled on 16, as smaller batch sizes proved unstable during training. Due to the limited number of target examples, we aimed to update the gradients when the model encountered both target and non-target samples.

The choice of learning rate was crucial in our training process. We started with a very small initial learning rate for both *ResNet* models, which was further reduced by a factor of 10 in later epochs for continued training. The starting learning rate for *ResNet9* was 0.000012, and for *PreActResNet18* was 0.000025. Both were reduced by a factor of 10 in the final epochs. This can be observed in the provided training graphs. *ResNet9* required slightly more epochs to achieve high accuracy. The number of epochs for which the models were trained can be seen in the graphs in Section 3.5.

Data augmentation significantly improved performance on test data in our experiments. Initially, we tested only a few techniques, such as random horizontal flips and random crops. Later, we employed various transformations available in the PyTorch library. We implemented one that randomly cuts a square from the image, which also proved to be effective. All transformations can be found in *src/config.py*. We also experimented with normalizing the data using the standard mean and deviation calculated from the training data, but this did not yield improvement. Ultimately, we normalized the data to values between 0 and 1.

3.5 Results

Training results and graphs are provided for the data. See Figure 5 for *ResNet9* and Figure 6 for *PreActResNet18*. Achieved results in numbers are shown in table 1

Model	**Test Loss**	**Test Acc**	**Train Loss**	**Train Acc**
ResNet9	0.0191	1.0000	0.0062	1.0000
PreActResNet18	0.0077	1.0000	0.0058	1.0000

Table 1: Model performance

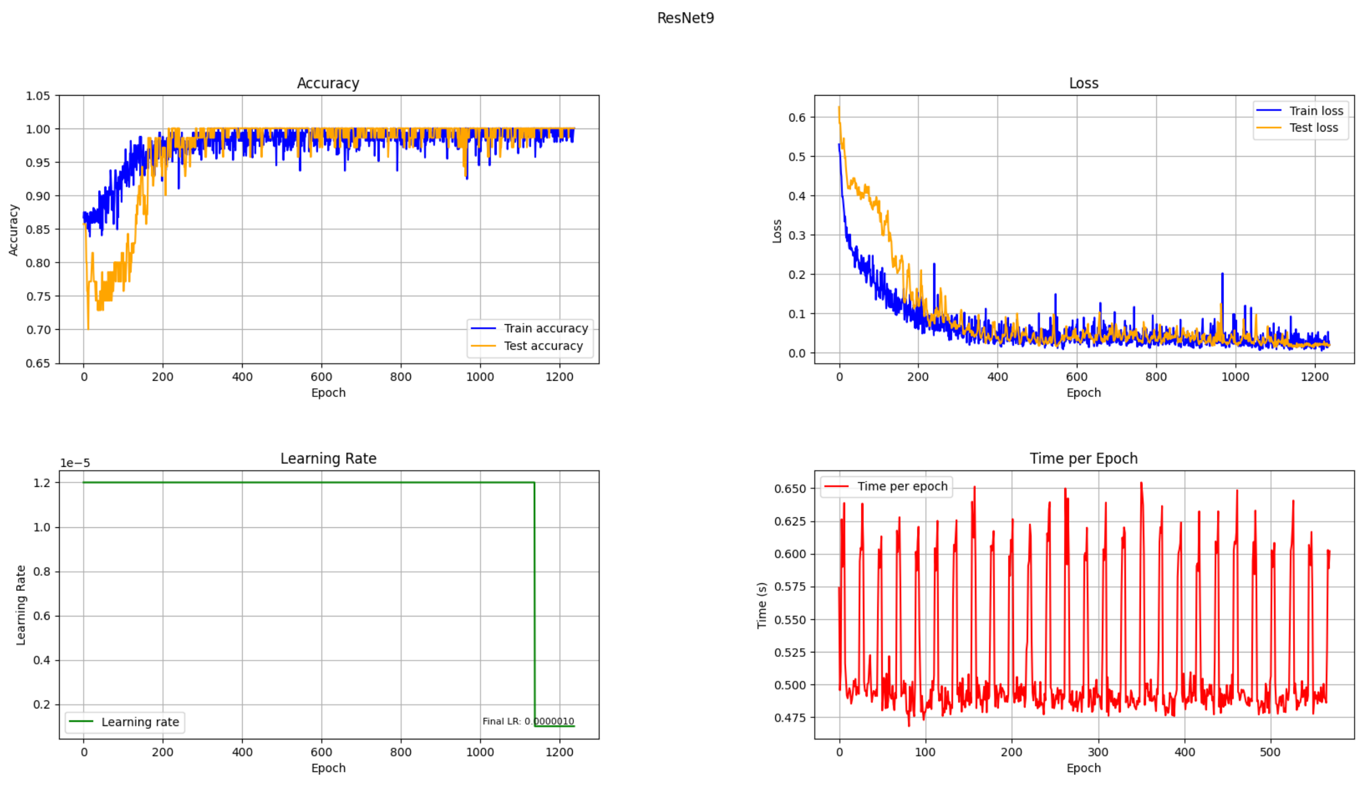

Figure 5: ResNet9

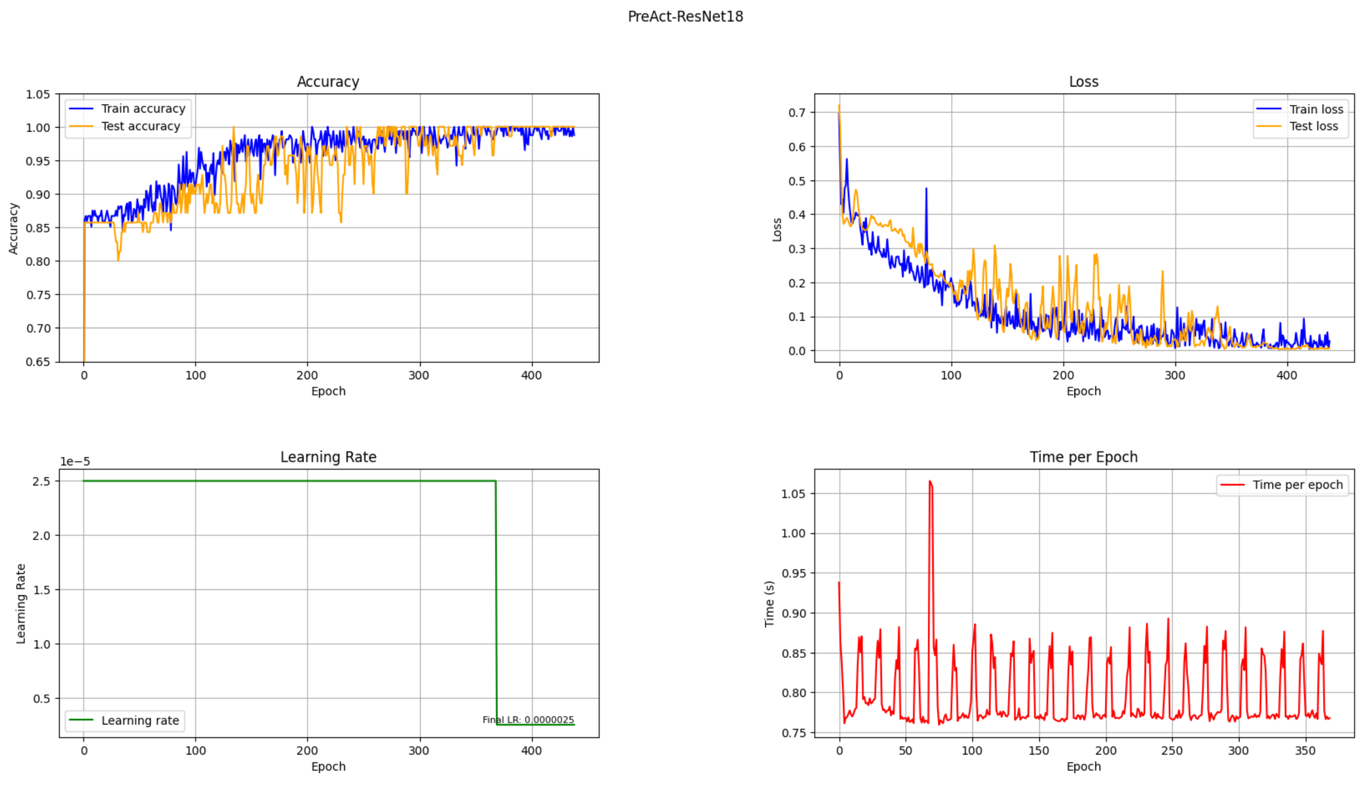

Figure 6: PreAct-ResNet18

4 Running the code

All the main code is in jupyter notebooks which are using code from folder */src/*.py*. There are implemented functions for easily working with dataset, model architectures, plotting graphs, training the models and configuration values.