Stanisław Bzdęga, 236448

Voice MNIST model – Machine Learning project

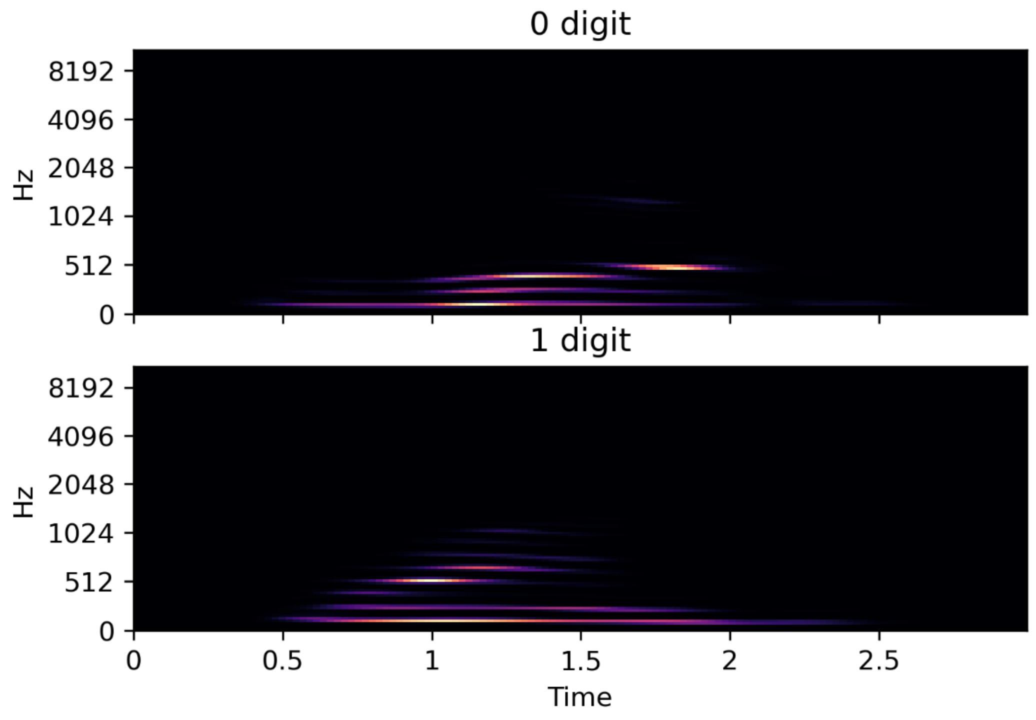
The main goal of my project was to build and analyse a deep learning model that, given a short .wav sound file, determines spoken digits. I have found a relevant dataset on Kaggle, where 60 different speakers were reading all digits 50 times:

https://www.kaggle.com/sripaadsrinivasan/audio-mnist

In total there were 30 000 of voice samples, what I found as a sufficient number to train my model.

There are several approaches to this problem. I made a use of Fast Fourier Transform (FFT) in order to generate Mel Spectrograms for each sample. It is further taken as the input of the neural network, thus the problem is reduced to the 2D picture classification.

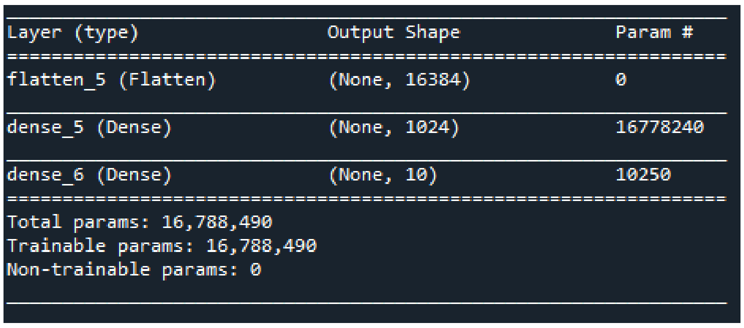
For the data pre-processing I used a Python library **LIBROSA.** Here I would like to present some Mel spectrograms generated from voice samples from the considered dataset, in order to illustrate the first stage of my model:



As the default size of a Mel spectrogram is (128, 128) I have chosen to set those numbers as my input layer.

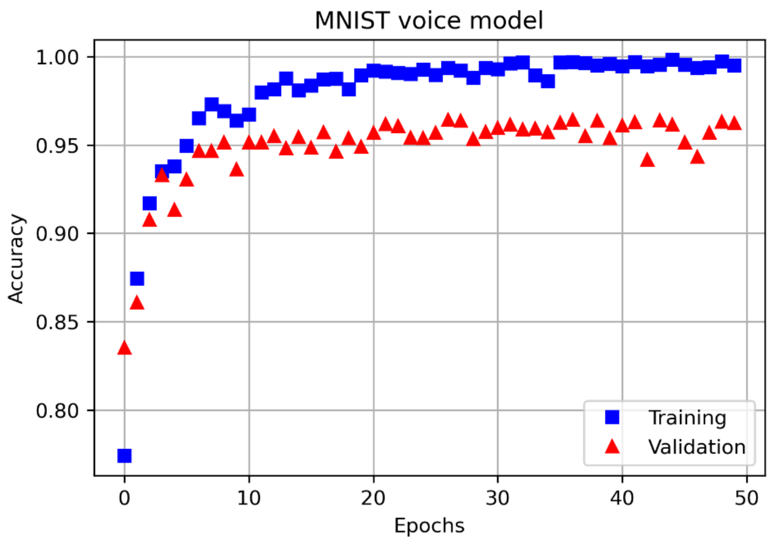
NETWORK STRUCTURE

The simplest possible network to start with, consists only of one hidden dense layer and the output layer. Number of neurons in the hidden layer was set to 1024, since the input layer is very big (for 128x128). So the initial model is as follows:



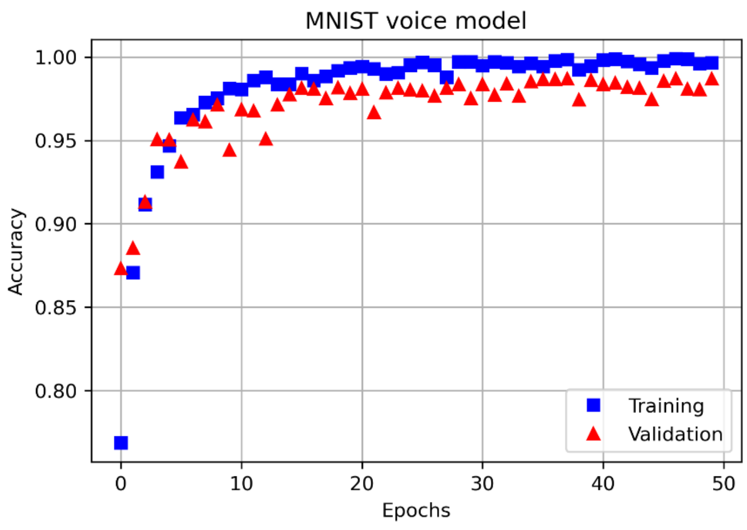
```
Layer (type)                 Output Shape              Param #
=================================================================
flatten_5 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_5 (Dense)              (None, 1024)              16778240
_________________________________________________________________
dense_6 (Dense)              (None, 10)                10250
=================================================================
Total params: 16,788,490
Trainable params: 16,788,490
Non-trainable params: 0
_________________________________________________________________
```

All activation functions for non-output layers were chosen as "**relu**", "**softmax**" is used for an output. The network was trained with the **batch size = 32** on Nvidia GPU (GTX 1060). Results are shown below:



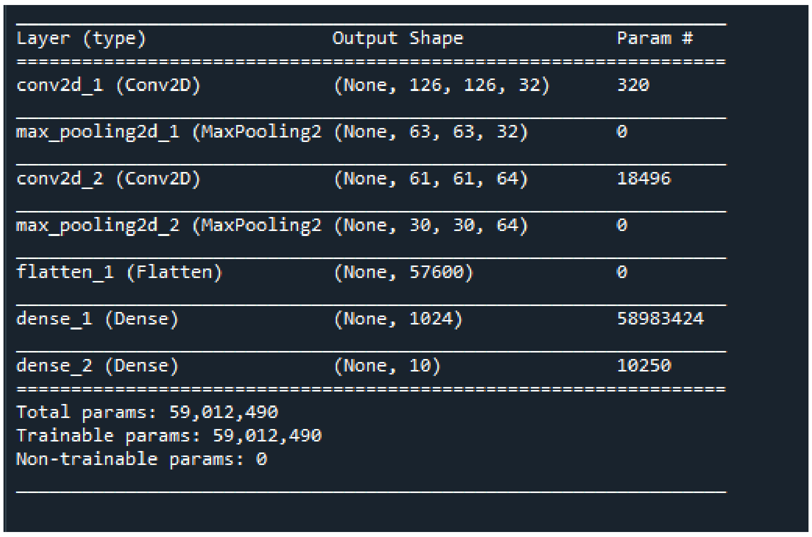
Results are surprisingly good, as for that simple structure, yet the validation result is not stable for final epochs. Probably having more distorted samples would heavily impact the model accuracy. In the case of image analysis we should introduce a Convolutional 2D layers.

So the next iteration of my model has one Conv2D layer (32 filters, 3x3 kernel) just next to the input layer. One can observe significant accuracy boost:



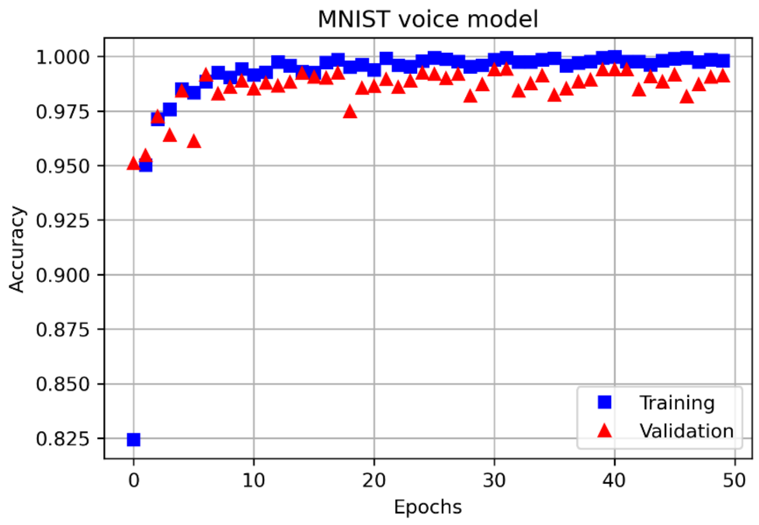
However after that improvement we have 130 mlns. of trainable parameters in our model. That is pretty much! The network has a very huge size and training it is time consuming.

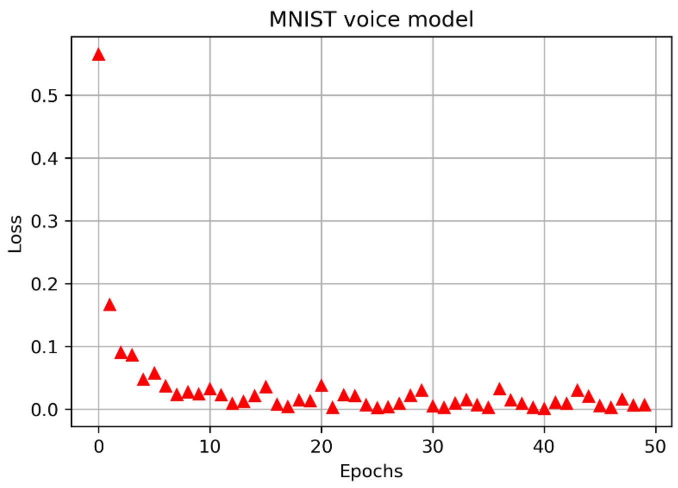
So the reasonable next step is to setup an additional Conv2D layer (64 filters, 3x3 kernel), together with MaxPooling2D (2,2) which reduces the size of an output twice (each time used):



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 126, 126, 32)      320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 63, 63, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 61, 61, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 30, 30, 64)        0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 57600)             0         
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              58983424  
_________________________________________________________________
dense_2 (Dense)              (None, 10)                10250     
=================================================================
Total params: 59,012,490
Trainable params: 59,012,490
Non-trainable params: 0
_________________________________________________________________
```

And the resulting output is as follows:

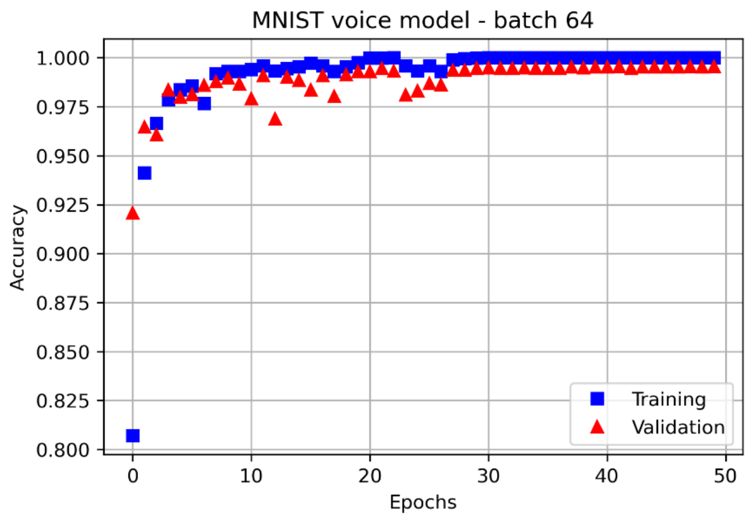


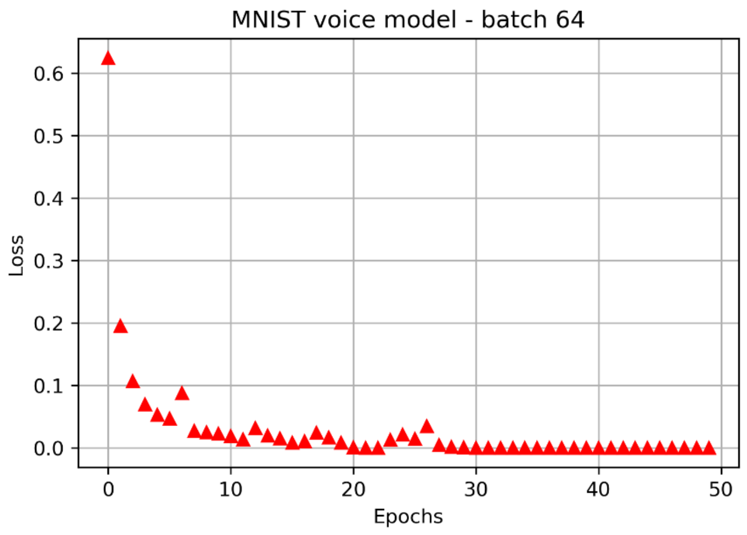


So we have reduced the number of parameters significantly, while still improving our validation accuracy.

BATCH SIZE

Let's check how the **batch size** influences our model's accuracy. In all previous network it was set to 32, the results below were generated for **batch size = 64**:





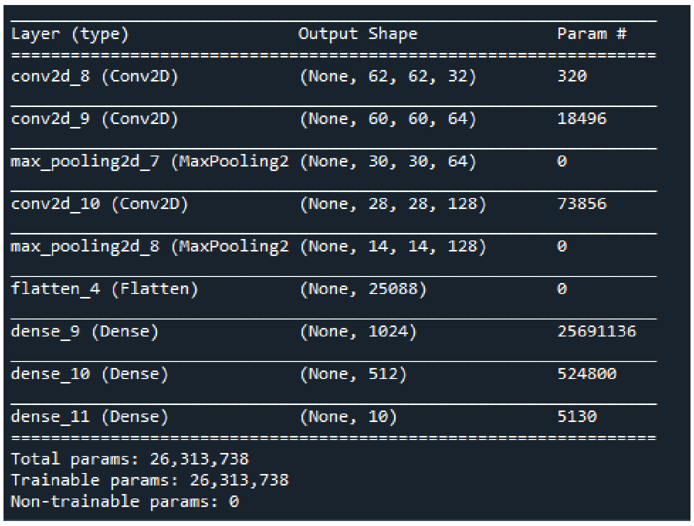
One can see, that after approximately 25 EPOCHS training accuracy stabilizes on 100%, while having insane **99,7 %** accuracy on validation. So changing batch size from 32 to 64 helped a lot.

When a batch size was increased to 150 the accuracy converged a little slower and the result was ~0.3 pp worse, so an optimal value of batch size was taken as **64.**

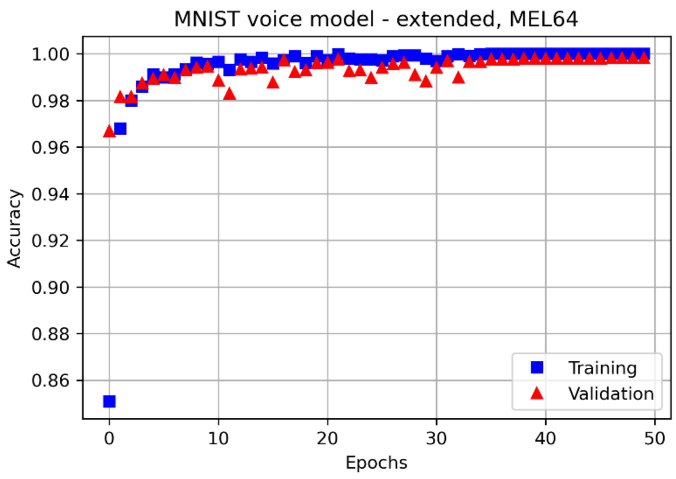
OVERFITTING – it does not happen

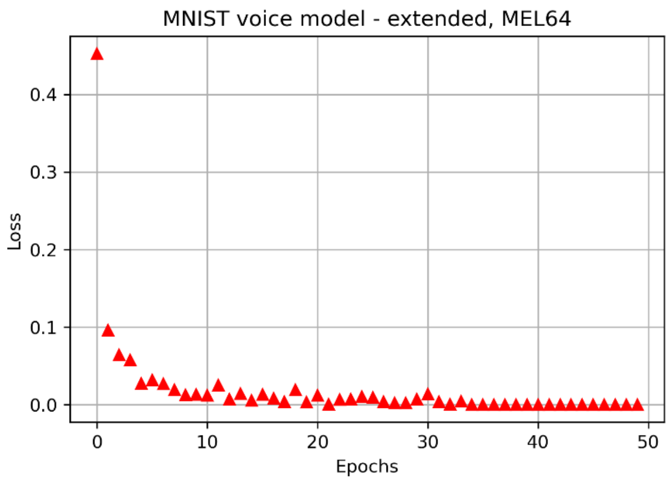
I tried to complicate the network a little bit by adding one more Conv2D layer, increasing number of filters of existing Conv2D's, as well as adding one intermediate Dense layer consisting of 1024 neurons. I found it interesting that despite my attempts to overtrain the model it does not happen.

Probably the main reason is that the data are very well prepared by FFT algorithm. Additionally this task seems to be much easier for the NN that I expected. Since that kind of complicated network does not perform better than the previous one, it in unnecessary to keep all those wages. So I will not further modify the network structure, just other parameters. Below I present, more complicated network, that I finally rejected together with its results:



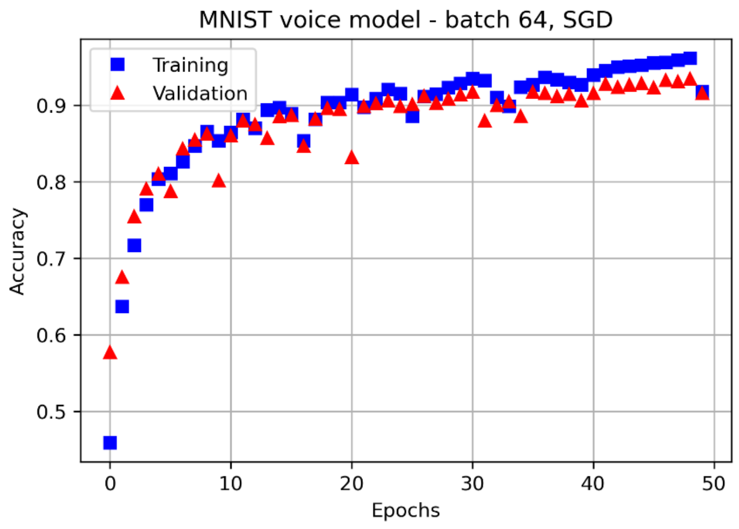
```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 62, 62, 32)        320

conv2d_9 (Conv2D)            (None, 60, 60, 64)        18496

max_pooling2d_7 (MaxPooling2 (None, 30, 30, 64)        0

conv2d_10 (Conv2D)           (None, 28, 28, 128)       73856

max_pooling2d_8 (MaxPooling2 (None, 14, 14, 128)       0

flatten_4 (Flatten)          (None, 25088)             0

dense_9 (Dense)              (None, 1024)              25691136

dense_10 (Dense)             (None, 512)               524800

dense_11 (Dense)             (None, 10)                5130
=================================================================
Total params: 26,313,738
Trainable params: 26,313,738
Non-trainable params: 0
```

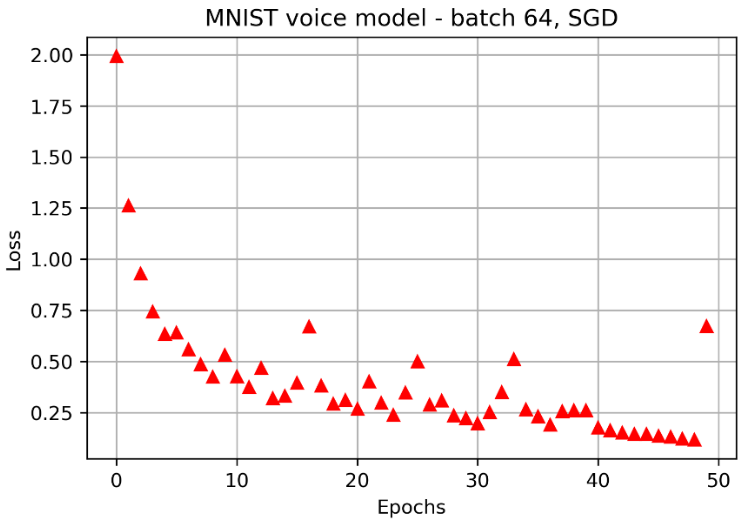




OPTIMIZERS

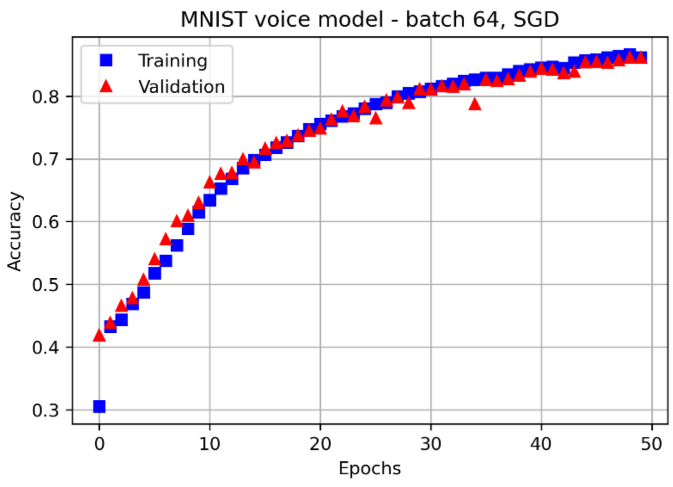
All previously generated results were using “**Adam**” (with default settings) optimizer during compilation. However I wanted to investigate what happens if I use **Stochastic Gradient Descent** optimizer. Firstly I set learning rate to **r = 0.01** :

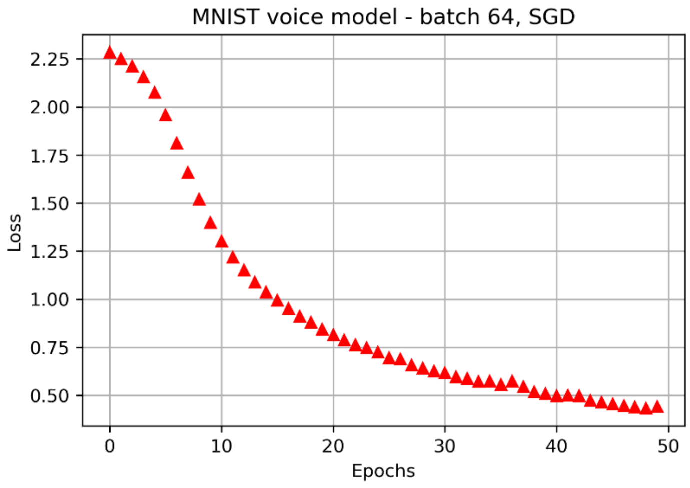




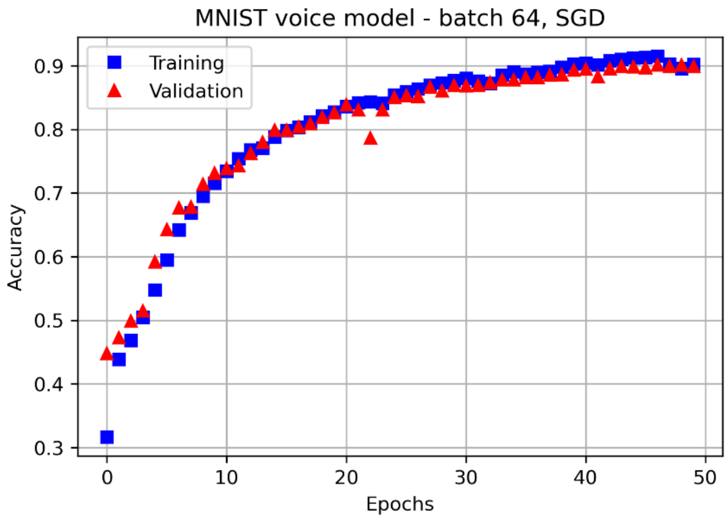
As for the same range of epochs results are worse, and what is the most important, there are noticeable jumps in the loss functions, what means that the optimizer tends to jump over the minima. Therefore it can be concluded that the learning rate for this algorithm is too big.

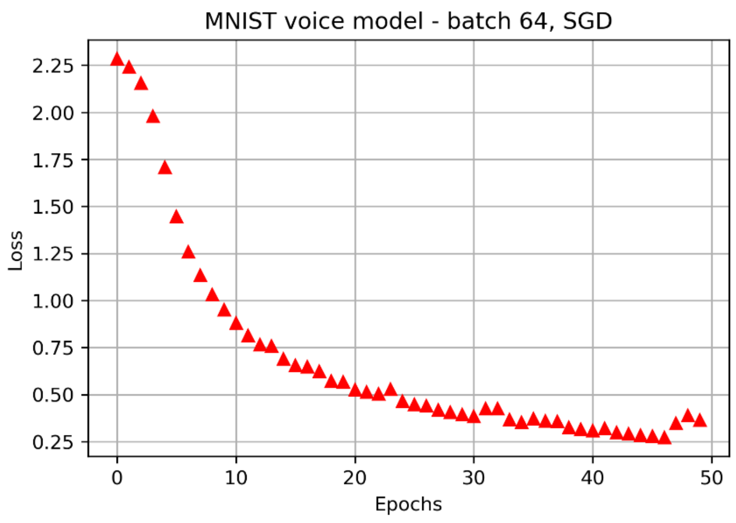
After changing it to **r = 0.001,** following results were obtained:





Here, we have extremely small variance in the loss function, but unfortunately, as expected, the system is learning extremely slow. So finally I tried **r = 0.002**:

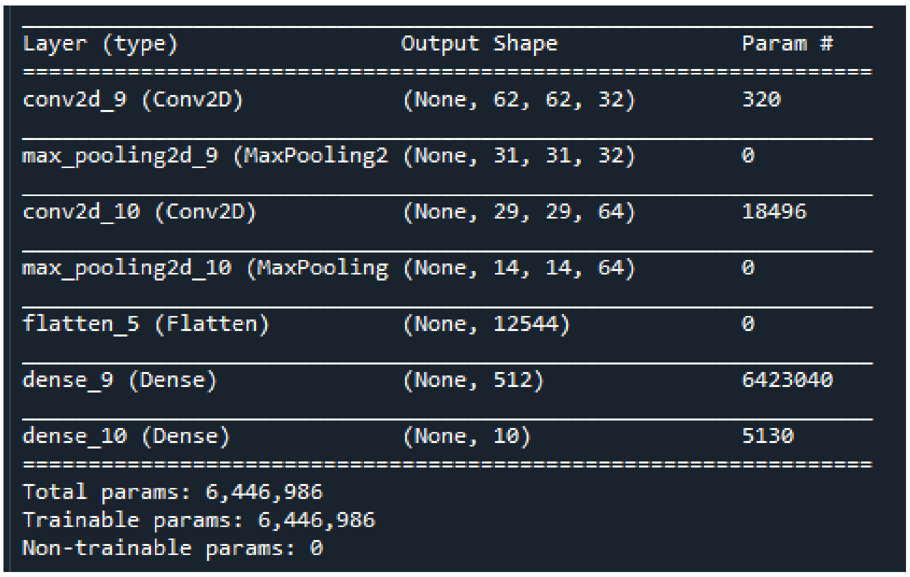




Concluding, **SGD** in the case of considered problem is less accurate then **Adam** optimizer. Additionally it takes noticeably more EPOCHS to converge, so I will not further take **SGD** into consideration for this model.

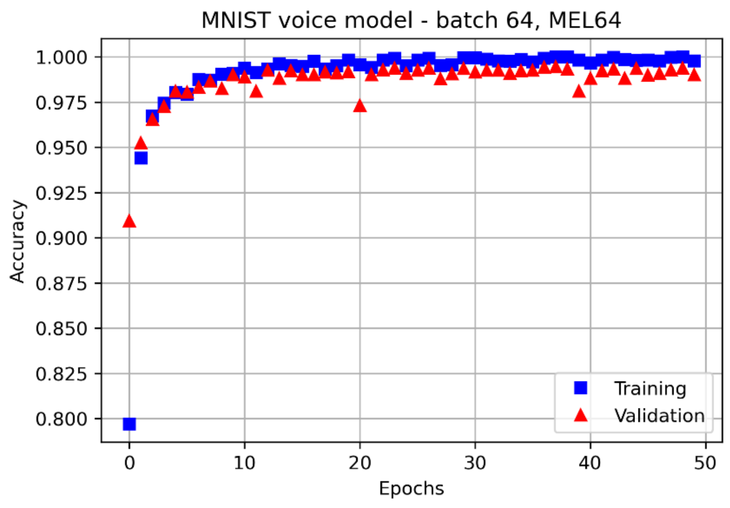
INPUT SIZE (and hidden layer)

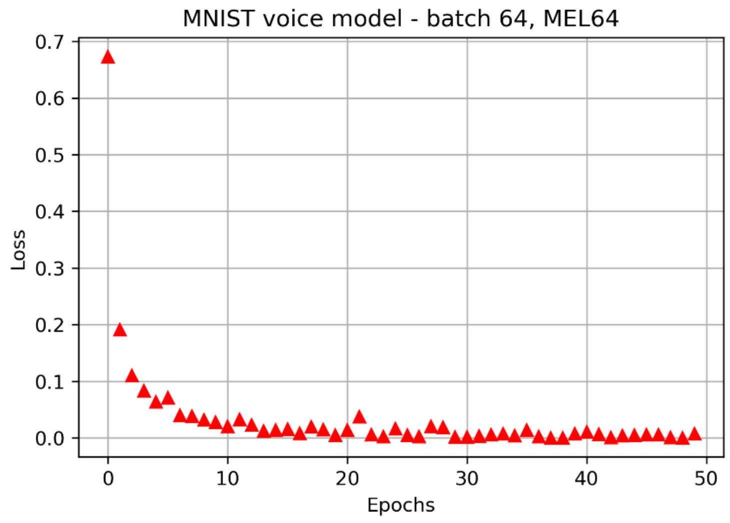
In order to further reduce the size of a network I changed the resolution of an output of FFT to 64x64. Also the size of a hidden layer was changed from 1024 to 512, since it was no necessary to keep it so big, when other layers were rescaled. The summary of the final network:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 62, 62, 32)        320
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 29, 29, 64)        18496
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 14, 14, 64)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 12544)             0
_________________________________________________________________
dense_9 (Dense)              (None, 512)               6423040
_________________________________________________________________
dense_10 (Dense)             (None, 10)                5130
=================================================================
Total params: 6,446,986
Trainable params: 6,446,986
Non-trainable params: 0
```

Its performance is presented below:





One can see that although the network is relatively simple, it performs very well. It has achieved **99,2%** accuracy on the test dataset, having only 7 mln trainable parameters. It learns extremely fast, therefore, having some another set of data, for example another language, it can be easily retrained.

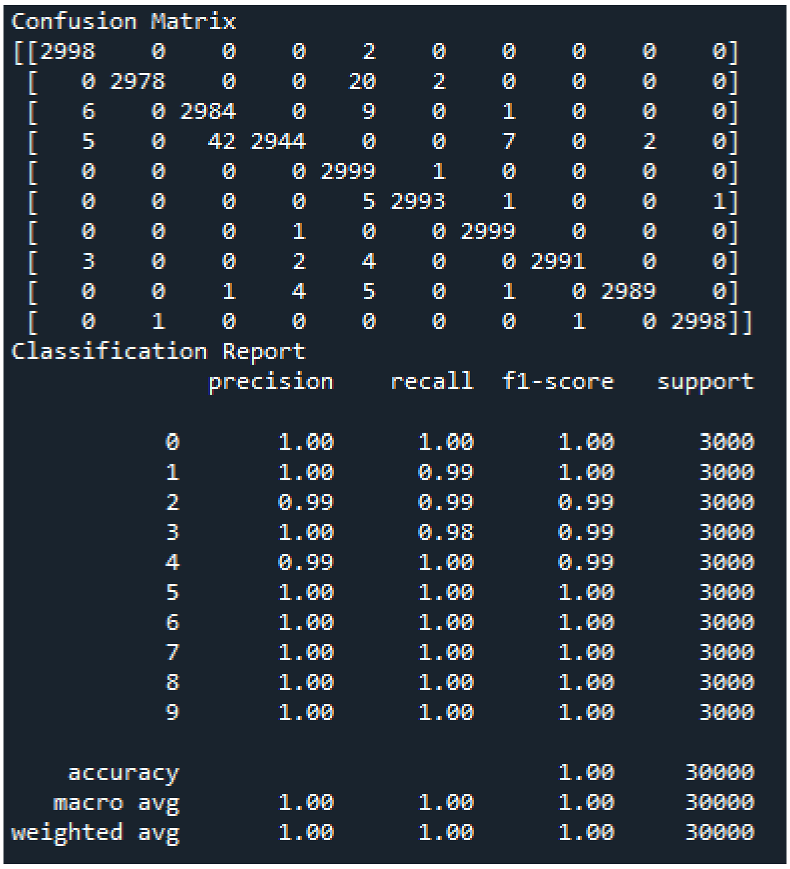
To summarize, it may be concluded that with an appropriate and careful pre-processing, considered model has achieved a **very high accuracy**. It also means that our Spectrograms based approach was indeed a good choice.

ACTIVATION FUNCTIONS

From the very beginning all the activation functions were chosen as **"relu"**, and for the output layer **– "softmax"**. So firstly I tried to change **"softmax"** to "**sigmoid"** in the output. I was heavily surprised that the network just didn't work. After one EPOCH it stopped learning and reached accuracy around **10%**. Also when I tried to choose sigmoid as an activator for convolution layers the situation was similar.
On the other hand, for a dense layer, changing "**relu"** to "**sigmoid"** didn't influence the performance at all.
Finally I tried "**tanh"** function in the dense layer, the output was not changed at all. So the dense layer is extremely tolerant in case of activation functions.

CONFUSION MATRIX

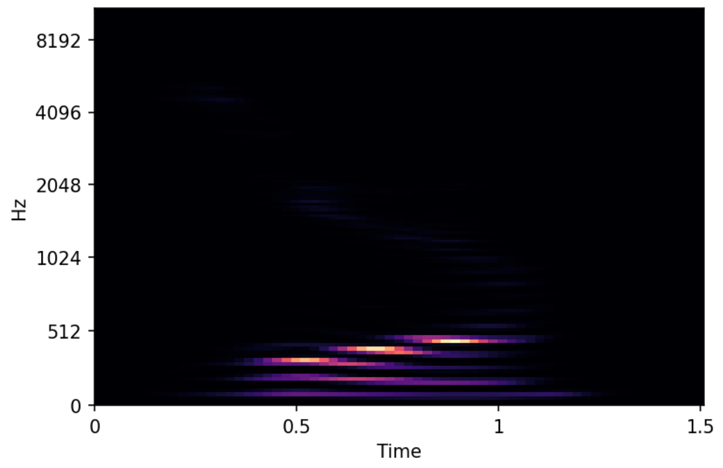
Finally I have taken all the data from the whole dataset and prepared a confusion matrix in order to see which inputs are being mistaken. One can see, that sometimes **2** is mistaken with **3** , as well as with **4**. But all at all the performance of considered model is extremely satisfying. I have also checked it with my own voice recordings, and no mistakes were made. Therefore it is not dependent on the quality or environment during recordings.



```
Confusion Matrix
[[2998    0    0    0    2    0    0    0    0    0]
 [   0 2978    0    0   20    2    0    0    0    0]
 [   6    0 2984    0    9    0    1    0    0    0]
 [   5    0   42 2944    0    0    7    0    2    0]
 [   0    0    0    0 2999    1    0    0    0    0]
 [   0    0    0    0    5 2993    1    0    0    1]
 [   0    0    0    1    0    0 2999    0    0    0]
 [   3    0    0    2    4    0    0 2991    0    0]
 [   0    0    1    4    5    0    1    0 2989    0]
 [   0    1    0    0    0    0    0    1    0 2998]]
Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3000
           1       1.00      0.99      1.00      3000
           2       0.99      0.99      0.99      3000
           3       1.00      0.98      0.99      3000
           4       0.99      1.00      0.99      3000
           5       1.00      1.00      1.00      3000
           6       1.00      1.00      1.00      3000
           7       1.00      1.00      1.00      3000
           8       1.00      1.00      1.00      3000
           9       1.00      1.00      1.00      3000

    accuracy                           1.00     30000
   macro avg       1.00      1.00      1.00     30000
weighted avg       1.00      1.00      1.00     30000
```

CNN layers visualisation

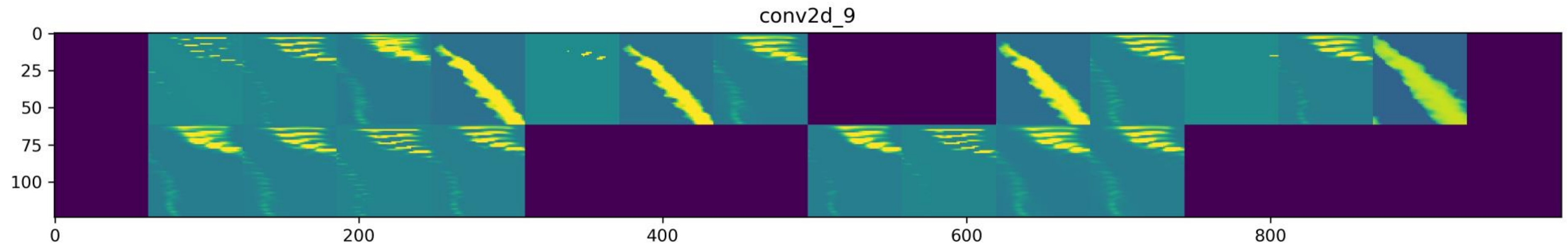
In order to understand better, what kind of shapes is being detected by convolutional layers, it is useful to visualise some intermediate activations. So for a sample input (my own recording – number 0), it was done using some built-in Keras utilities:

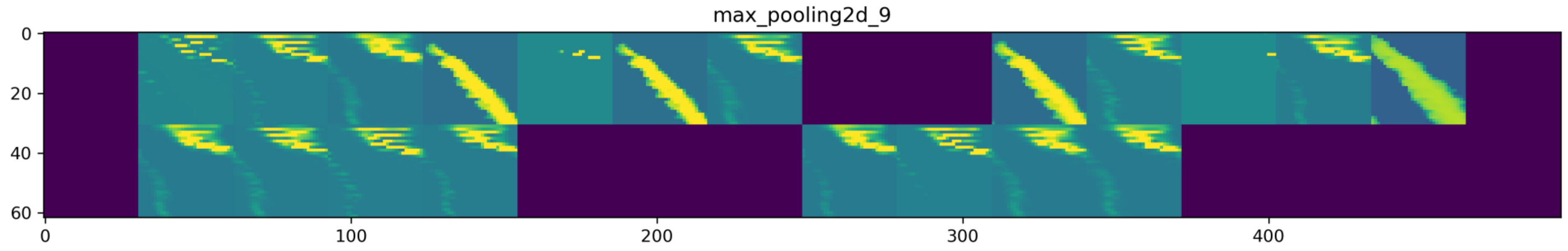


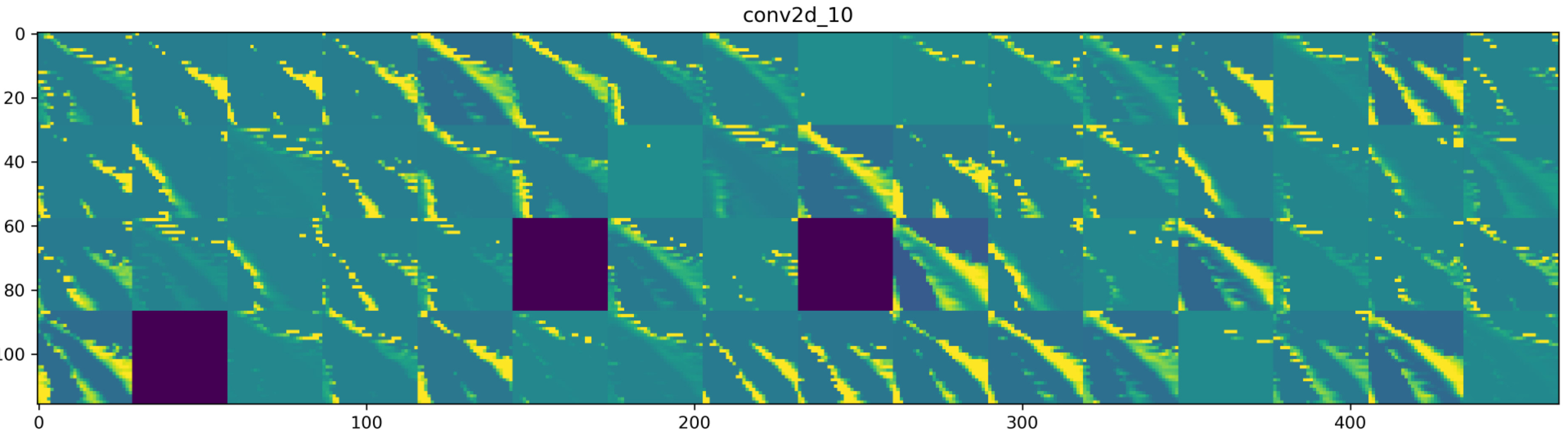
On the left, there is a Mel Spectrogram (64x64) of an input and below I presented the output of particular intermediate convolutional layers. One can see what kind of filters has been activated. Furthermore it also visible why we use Max Pooling layers – they decrease the resolution of a picture, while still presenting essential features detected by Conv2D.

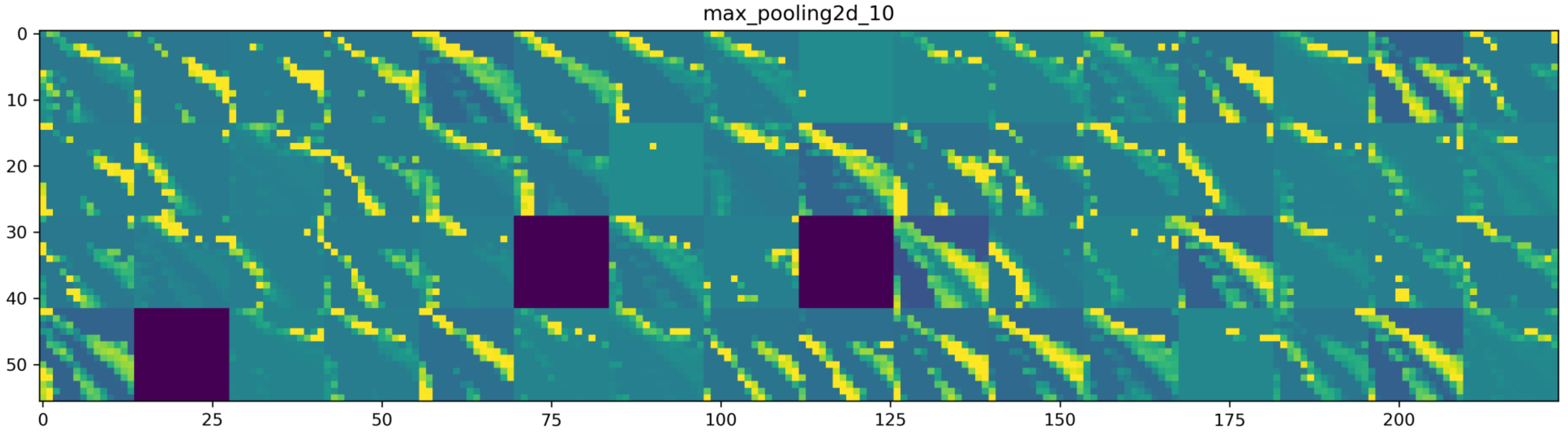
It may be concluded that for this network the shape of harmonics and overtones is essential in order to work appropriately, and those features are clearly detected by convolutional layers.

Some of filters are even extremely sensitive to very high modes, that are not visible on the presented spectrogram, but apparently they are present in the data.






conv2d_10
0
20
40
60
80
100
0
100
200
300
400


max_pooling2d_10
0
10
20
30
40
50
0
25
50
75
100
125
150
175
200