Until now i was using a pretrained resnet18 because the pretraining seemed to perform a lot better than training from scratch. I used all the weights from pretraining and only changed the output layer to size 1 instead of 1000 (for 1000 classification outputs for the images it was trained to classify). This pretraining used a 3-channel input (RGB colors). The logmel/MFCC spectrogram only has a single channel as input. In order to not change the pretrained weights, until now I copied the spectrogram to 3 channels.

This increases the amount of data in memory during training and inference. It also increases the processing time. In the following experiment, it was attempted to exchange the input dimensions and layer from 3 x height x width to 1 x height x width. The kernels change from 3x7x7 to 1x7x7. The weights for the kernels were calculated as:

***the mean over all 3 channels.***

The questions were:

1. How much does the memory usage change?

Memory usage seems to be almost identical on the PC. But the memory that the graphics card (according to CUDA functions) has still available was a lot larger for the 1 channel inputs. With 3 channels the graphics card had 2.2/11GB still available while for the 1 channel input, there still were 8/11GB available. Did it actually save 6 GB? This would be crazy!!! CHECK

1. How much does the processing time change?

Processing time decreases from 41-42 to 39-40 seconds for 1 epoch for smaller batch sizes (16, 64) but is again almost identical when the batch size is large (512). This was measured with an input of 112 logmel bins and 200 time steps. This must be verified by a longer run that counts more than a few epochs. 🡪 it was tested over multiple hyperparameter settings: for the most part the duration was a little shorter (1 run took about 10min and the 3 channel runs took on average 20-30 seconds longer but at least 1 run was actually shorter). In the end the gain may be between 3-5% in processing time but maybe these gains were for other reasons. Also it took longer when I used the augmented datasets as well which I don’t understand why….

1. How does the performance change (negatively or positively or at all)?

The performance seems to have gotten worse in terms of AUC and loss but slighty better with accuracy and tpr. Here only 1 hyperparam setting was tested. More in depth tests following.

***The weights of channel 0 (red channel I guess):***

***Adding differential learning rates to the ResNet architecture:***

Up until now, the entire model used one learning rate for all layers/weights. The learning rate decay changed this global learning rate, but it did so for all layers/weights at the same time. For a pretrained network, it makes sense to keep more information from the earlier layers (close to the input) because these early layers have learned abstract representations (simple shapes). To keep the knowledge from pretraining, we set the learning rate to be small for those layers. The layers closer to the output are not so abstract anymore and need to learn new representations of what we are trying to classify. In this case the last layer was trained to classify 1000 classes of images of ImageNet which is of course very different to a logmel spectrogram for example. This is why the learning rate at the output needs to be larger.

Chart, diagram

Description automatically generated

The ResNet model consists of 4 main layers which all include several instances of convolutional layers, batch norms and pooling. After each of those major layers, the image size is decreased while the number of kernels/filters is increased.

Since I do not want to give a conv layer a different learning rate than the following batch norm layer, I specified one learning rate for each of those major layers plus the input and output layers. In the created method, the learning rates for the input and the output layer can be specified. The layers in between get a learning rate linearly interpolated between those two values. These parameters were implemented in the hyperparameter search.

As for the learning rate decay, all of those layer wise rates get decayed by the same factors at the same time.

***Dropout2D***

After a while, the model tends to overfit and get an almost perfect score for AUC (99%) even despite the efforts to augment the data and use mix-up. To be able to keep on training without overfitting, dropout has been a popular tool. Regular dropout has been implemented in the last layer (dense layer with 512x1 weights) as well as a spatial dropout or dropout2D after each major layer of the ResNet. This results in 4 Dropout2D layers and one regular dropout layer.

Unlike the regular dropout layer, the 2D variant does not omit/set to zero single values/pixels but it does so for an entire feature map at a time. So, if you have a 28x28 picture which is fed into a conv layer with 64 kernels (and no down sampling), the output is 64x28x28 images. For each of those 64 images/feature maps, there is a chance p that all 28x28 pixels will be set to 0.

***Dropout Rates:***

The dropout rate (argument p for probability) determines how likely a single feature map will be dropped. The general idea is to set the dropout rate lower for layers closer to the input and higher towards the output. But this may be incorrect. It could be implemented as hyperparameters to search.