Report Homework 2 - Augmented Tensorflop

Daniele Civati, Matteo Palazzoli, Francesco Panebianco | Prof. Matteucci

Work Process

We started out separately, working on our own models, then halfway through phase one we selected the best performing and tuned it in different ways.

Observations on Data

We started out by plotting the data, sampling a window from each class and looking at what the data might represent. We also immediately plotted the class distribution, noticing an astonishing imbalance.

Since the labels were clearly words from *Pink Floyd*'s songs, the first idea that came to mind was that it might be manipulated sound samples, even though the shape of the waveform didn't look anything like it. Using the python *wave* library[1], we converted a full signal into a *.wav*, realizing that the data probably had nothing to do with sound. Just to be sure, though, we also tried converting the data into *Mel Frequency Cepstral Coefficients*[2] and building a model on that. However, the huge failure that it produced finally put the idea of it being related to sound to rest.

The next step was plotting the correlation between channels, both the *Pearson Correlation* and *Spearman Correlation*[3]. The first one was useful to determine that channels 0, 1 and 4 were probably a good subset of channels to attempt to use in place of the whole dataset. Finally, the plotting of a correlogram[4] for each class and channel brought to the conclusion that the sequences were probably *AR* models with an order of 2 to 3, depending on the cases. This suggested the possibility of using the `statsmodels`[5] library to predict the next time steps and use them as training data as a form of data augmentation, but could not be attempted due to time constraints.

Figure1: Pearson correlation of channels

Attempts at Preprocessing

Since the amplitude of the different signals varied extensively between different classes and channels, it was likely that some form of preprocessing could have helped, even though some skepticism was raised towards the usual *mean-std* based scalers. Still, we attempted *MinMax*[6], *RobustScaling*[7] and even some custom scaling based on the logarithm of the absolute value, sometimes preserving the sign of the original sample by including a new dimension in the array for that purpose. In the end, *MinMax* and the custom log scaler performed worse than the *RobustScaler*, which however itself didn't have a significant impact on the performance.

---

[1] https://docs.python.org/3/library/wave.html
[2] https://librosa.org/doc/main/generated/librosa.feature.mfcc.html
[3] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html
[4] https://www.statsmodels.org/dev/generated/statsmodels.graphics.tsaplots.plot_acf.html
[5] https://www.statsmodels.org/dev/generated/statsmodels.tsa.ar_model.AutoReg.html
[6] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[7] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html

At some point, clipping outliers and doubling the window size seemed to have sufficient performance increase on the local performance, however such increase could not be tested on the hidden test set, due to the fact that the window size increase depended on the knowledge that a certain windowed sample belonged to a certain class (or, to be precise, that they were adjacent windows), which was of course unavailable information.

At some point even *undersampling* of the disproportionately populated *Sorrow* class was considered, but it didn't help the overall model performance as hoped. We also tried a form of oversampling, making copies of underrepresented classes, until a population of about 700 data points was reached for every class. It didn't improve the performance, therefore we removed it.

Dense and Convolutional Attempts

The very first approach, just to be sure, was a very simple fully connected neural network with 354,956 total parameters distributed in two hidden layers of 1024 and 128 neurons respectively, *swish* activation, dropout of 0.2 between every dense layer, **no** class weighting, trained with *Adam* (1e-3 lr) with a *CategoricalCrossentropy* loss, giving us a baseline accuracy of 0.3856.

The next attempt was a traditional CNN, with blocks of *Conv1D*, *MaxPooling*, an initial *BatchNormalization* and a *GlobalAveragePooling1D* before the softmax layer, yielding a performance somewhat similar to the dense approach. A few more convolutional models were trained, and will be mentioned later in the report.

LSTM and BiLSTM Attempts

After the convolutional approaches we tried some memory-based layers like LSTM. Having two LSTM layers with a total of 128 neurons each and a dropout of 0.5 before the dense layer was a frequent choice for experiments with recurrent neural networks. In fact we've noticed that the addition of other LSTM layers did not change the accuracy much. These models were trained trying different loss functions other than *Categorical Crossentropy*. *KLDivergence*[8] didn't seem to work well in this context, whereas the *SigmoidFocalCrossentropy*[9] or the base *CategoricalCrossentropy with class weighting* both gave satisfying results in different trials.
The usage of Bidirectional LSTM was prevalent in the experiments. The architecture used was similar to the LSTM one, with two layers of bidirectional LSTM with 128 neurons each. As in the previous models, we mainly used *CategoricalCrossentropy* as loss function, *Adam* as optimizer, and class weighting. In a notable attempt, that of notebook `hw-2-bilstm.ipynb`, it gave a consistent accuracy of 0.5582 and F1-score of 0.5027, handling the class imbalance better than usual.

Attempts at Augmentation

In order to have more samples in the training set, we've tried to perform some time series augmentation techniques we found[10]. Firstly, we added a normal random noise ("jitter") with low variance to some samples. The result was about the same as before, slightly less. Then, we tried the method called "scaling" which random scales the time series, and "rotation", which performs

[8] https://www.tensorflow.org/api_docs/python/tf/keras/losses/KLDivergence
[9] https://www.tensorflow.org/addons/api_docs/python/tfa/losses/SigmoidFocalCrossEntropy
[10] https://github.com/uchidalab/time_series_augmentation

“for multivariate time series, flipping as well as axis shuffling”. The result was a lot worse, probably because the axis’ original position was important. We then tried some of the more sophisticated techniques that are presented in the repository, like *time warping*, that applies a random time distortion on the signal. After various tuning attempts we had a LSTM model with an augmentation pipeline consisting of jitter, scaling and time warping applied each on a copy of the training set, that performed 67% in local but worse on the hidden test set on *Codalab* (see notebook `hw-2-augmodel.ipynb`).

Attention and MultiHeadAttention Attempts

Attention is also something that was attempted from the very beginning, both as a block in a recurrent architecture and as the only type of layer in the network. The first time, as comic relief, we asked *ChatGPT*[11] to provide us with a simple implementation for self-attention. It had a 128 neuron LSTM and used the base *Attention* layer. It was of course changed and tuned to achieve better results. Layers that were tested include the basic *Attention*, *AdditiveAttention* and finally *MultiHeadAttention*, which can be seen in action in the notebook `hw-2-attention.ipynb`. This as well did not achieve greater performance than what was seen up to that point.

ResNet50V2 Attempt

An attempt that was suggested by the tutors and followed through was a *ResNet50V2_1D*[12], tested both with a doubled window (72 samples) and with a subset of channels (0,1 and 4). The model used *SigmoidFocalCrossentropy* as loss function and *Yogi*[13] (lr=4e-4) as the optimizer. *ResNet* however did not seem to outperform any of the previous models, so after numerous attempts at tuning it, it was dropped in favor of simpler models.

Final Model

After various attempts to improve the accuracy through different models and data preprocessing, we observed that the best model remains one of the first, which is the Conv1D. This model, in which we don't use data preprocessing, is composed of two Conv1D layers, with 200 filters of size 6 each and *ReLU* activation, interspersed by a *MaxPooling1D* and followed by a GlobalAveragePooling1D and the classifier, which is a two dense layer with *ReLU* and *Softmax*. The loss function is a *CategoricalCrossentropy* without class weighting while the optimizer is *Adam*. Callbacks include *EarlyStopping* and *ReduceLROnPlateau*.

```
Model: "TheMaxSideOfThePool"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input (InputLayer)           [(None, 36, 6)]           0
_________________________________________________________________
conv1d (Conv1D)              (None, 36, 200)           7400
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 18, 200)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 18, 200)           240200
_________________________________________________________________
global_average_pooling1d (Gl (None, 200)               0
_________________________________________________________________
dropout (Dropout)            (None, 200)               0
_________________________________________________________________
dense (Dense)                (None, 128)               25728
_________________________________________________________________
dense_1 (Dense)              (None, 12)                1548
=================================================================
Total params: 274,876
Trainable params: 274,876
Non-trainable params: 0
_________________________________________________________________
```

With this model we initially reached an accuracy of 40% but after tuning the batch size to 8 we reached an accuracy in the range of 64-70%. In the hidden test set on *CodaLab* the drop in accuracy was minimal, with a value of 0.681 in the first phase and 0.687 in the second one.

[11] https://chat.openai.com/chat
[12] https://github.com/hfawaz/dl-4-tsc/blob/master/classifiers/resnet.py
[13] https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/Yogi