AN2DL: Second Homework Report

Les Choristes

Pietro Marco Gallo Michele Guerrini Gabriele Maucione

Introduction

The aim of the second homework of the AN2DL course is to design the architecture of a neural network and train it to perform as accurately as possible a multinomial classification task over a dataset of $(2429 \times 36 \times 6)$-sized time series, divided in 12 classes.

After incessant experimenting, our group has designed a deep hybrid neural network, made of a convolutional part and using the attention technique, which achieved a final accuracy score of 71.27% on the test set of the second phase of the challenge.

Neural Network Architecture

Our best model is characterized by a medium-high level of complexity, since it is composed of over 3.1 million parameters. More specific details about their distribution in each layer can be found in Fig. 1.

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
Input (InputLayer)	[(None, 36, 6)]	0	
normalization_4 (Normalization)	(None, 36, 6)	0	Input[0][0]
conv1d_52 (Conv1D)	(None, 36, 256)	4864	normalization_4[0][0]
conv1d_56 (Conv1D)	(None, 36, 256)	7936	normalization_4[0][0]
conv1d_60 (Conv1D)	(None, 36, 256)	6400	normalization_4[0][0]
conv1d_53 (Conv1D)	(None, 36, 256)	196864	conv1d_52[0][0]
conv1d_57 (Conv1D)	(None, 36, 256)	327936	conv1d_56[0][0]
conv1d_61 (Conv1D)	(None, 36, 256)	262400	conv1d_60[0][0]
conv1d_54 (Conv1D)	(None, 36, 256)	196864	conv1d_53[0][0]
conv1d_58 (Conv1D)	(None, 36, 256)	327936	conv1d_57[0][0]
conv1d_62 (Conv1D)	(None, 36, 256)	262400	conv1d_61[0][0]
max_pooling1d_13 (MaxPooling1D)	(None, 18, 256)	0	conv1d_54[0][0]
max_pooling1d_14 (MaxPooling1D)	(None, 18, 256)	0	conv1d_58[0][0]
max_pooling1d_15 (MaxPooling1D)	(None, 18, 256)	0	conv1d_62[0][0]
dropout_30 (Dropout)	(None, 18, 256)	0	max_pooling1d_13[0][0]
dropout_32 (Dropout)	(None, 18, 256)	0	max_pooling1d_14[0][0]
dropout_34 (Dropout)	(None, 18, 256)	0	max_pooling1d_15[0][0]
conv1d_55 (Conv1D)	(None, 18, 512)	393728	dropout_30[0][0]
conv1d_59 (Conv1D)	(None, 18, 512)	655872	dropout_32[0][0]
conv1d_63 (Conv1D)	(None, 18, 512)	524800	dropout_34[0][0]
average_pooling1d_13 (AveragePo	(None, 9, 512)	0	conv1d_55[0][0]
average_pooling1d_14 (AveragePo	(None, 9, 512)	0	conv1d_59[0][0]
average_pooling1d_15 (AveragePo	(None, 9, 512)	0	conv1d_63[0][0]
dropout_31 (Dropout)	(None, 9, 512)	0	average_pooling1d_13[0][0]
dropout_33 (Dropout)	(None, 9, 512)	0	average_pooling1d_14[0][0]
dropout_35 (Dropout)	(None, 9, 512)	0	average_pooling1d_15[0][0]
concatenate_4 (Concatenate)	(None, 27, 512)	0	dropout_31[0][0] dropout_33[0][0] dropout_35[0][0]
attention_4 (attention)	(None, 512)	539	concatenate_4[0][0]
dense_10 (Dense)	(None, 12)	6156	attention_4[0][0]

Total params: 3,174,695
Trainable params: 3,174,695
Non-trainable params: 0

Figure 1: Summary of the model architecture.

Firstly, time series are inputted as Numpy arrays with `numpy.load`, a stage wherein they are divided into two sets: training (80%) and validation (20%). Afterwards, a `Normalization` layer is applied in order for the features of the time series to follow a normal distribution $\mathcal{N}(0,\, 10)$. Then, the normalized arrays are fed to three convolutional subnets, each made of four 1D convolutional layers, two `Pooling` and two `Dropout` ones. The only difference among the convolutional subnets is the kernel size parameter, which is assigned a value of 3, 4, 5. These subnets are subsequently concatenated and the processed arrays pass through an `Attention` layer, created with a custom-made class. Finally, a fully connected part made of two `Dense` and one `Dropout` layer determine, through the `softmax` activation function of the last output `Dense` layer, the probabilities of the time series to be classified in each class.

The described model is compiled with Categorical Crossentropy as loss function, and with `Adam` optimizer, as they seemed to be the most suitable for the faced classification task.
The mini-batch size used in training is of 8 samples, as small sizes turned out to work better with `Adam`.

The only used callback function is early stopping with a very high patience of 70 epochs, in order for the neural network's loss function to have a higher chance to reach a better optimum.

The Steps Behind the Final Model

Before coming to the just presented final architecture, many different solutions were tried: not all of these were successful, but each made us understand something more and contributed as a step towards the final result.

Originally, in order to approximately evaluate different architectures in advance, we divided the dataset in three subsets: training (70%), validation (15%) and test (15%). In such a way, we kept the test set to obtain an accuracy score beforehand and a confusion matrix for each architecture we tried in the development phase. This division was made by means of the `validation_split` parameter within the `fit()` function, and of `sklearn.model_selection.train_test_split`. Afterwards, the test set was discarded in favor of a bigger training set, because the evident class imbalance always put such a significantly small number of samples of the least populated classes inside the test set, that the results in predicting their class were inaccurate.

The First Model: CNN Classifier with LSTM

Our starting point was a quite simple tailor-made convolutional neural network followed by two `LSTM` layers. The convolutional part was made up of 3 convolutional layers, two `MaxPooling` and one `Dropout` layer, in order to try to prevent overfitting.
The same loss function and optimizer of the final architecture were already used here too, as well as for most of the following configurations. This starter model reached an accuracy score of about 66% on Codalab's test set.

Triying to improve this score, the main issue was found in the presence of the `LSTM` layers, which are more suitable in case of large data streams as input. However, their removal and the consequent usage of a pure convolutional neural network proved to be less efficient than the previous combination: it scored an average accuracy of 64%, indeed.

At the same time, the variant of this model with a `Bidirectional LSTM` was tried, but it resulted in a loss of few percentage points on the accuracy of the local test set.

Different Prefab Architectures

In order to make use of other kinds of layers different from `LSTM`, more complex architectures were object of experimenting. Only the most remarkable ones will be quoted.

The first one was a configuration seen during lectures, the residual neural network. It was made of three blocks, each one with two parallel sequences of `Conv1D`, `BatchNormalization` and `Activation`[1] layers, and one sequence of just `Conv1D` and `BatchNormalization`. Then, channels were expanded only in the first two blocks with `Conv1D` and `BatchNormalization`, whilst they were not in the third one, as they only passed through a `BatchNormalization` layer. In the end, all blocks had a final `Activation`[1] layer in common and were followed by the fully connected part of the ResNet, made of a `GlobalAveragePooling` and a `Dense` layer.

The second one was an Inception classifier properly redesigned for the purpose of receiving mono-dimensional time series as input, as it was composed of `Conv1D` and 1D `Pooling` layers with respect to the 2D version used during transfer learning in the first challenge.

Regardless of the various attempts, their usage was not further prosecuted because of an achievable accuracy score which was stable within a range between 50% and 60%.

[1] In this case, the `ReLU` function.

Data Augmentation and Class Imbalance

At this point it was clear that the big class imbalance presented by the dataset was the main problem to solve and, in order to tackle it, we tried to perform Data Augmentation using the `tsaug` library. Various transformations, such as `TimeWarp`, `Quantize`, `Reverse`, `Drift` and `AddNoise`, were applied to the dataset, trying different combinations.

The majority of these transformations worsened the performance of the network by a lot, whilst the only one that seemed like it could have had a beneficial impact on the training of the model was `AddNoise`. But after trying different combinations of noise parameters, there was not a real improvement in the accuracy, so the whole idea of applying Data Augmentation was rejected.

Another idea implemented to handle the class imbalance was the computation of class weights. These have the aim to assign different weights to each class inversely with respect to their distribution, in order to compensate for their imbalance. Both a custom function and the `sklearn` method were tried, but things went almost the same way of data augmentation attempts, so in the end also this module was not implemented.

Last Steps Towards the Final Model

Given the unhappy attempts with more complex models, we came back to the original idea of mixing a CNN with a `LSTM`-made block. In this case, however, a little performance boost was granted by the introduction of regularizers (`L1L2` and `L2`, in particular) within the `Dense` layers of the fully connected part and some of the `Conv1D` ones of the convolutional part.

Additionally, up to this point feature normalization had been quite of a problem: although it would have theoretically improved the performance of the classification, in practice any attempt of normalization (e.g., StandardScaler, MinMaxScaler within [0, 1]) was giving slightly worse accuracy scores. The solution came with the introduction of the `Normalization` layer, that is also used in the final architecture. The initial attempt was to use mean 0 and variance 1, but it was empirically found that different variance values led to better results, as reported in the submitted model. Furthermore, this layer also works better in combination with the `Adam` optimizer with respect to the `BatchNormalization` one, as the latter gives better results with a big mini-batch size.

At this point, a new inspiration came from the paper "Attention is all you need" (Vaswani et al.): introducing an `Attention` layer inside the network between the convolutional part and the fully connected one. This particular new kind of layer was created with a custom-made class inheriting from `tensorflow.keras.layers.Layer`. In this custom class, initial state weights were randomly initialized following a normal distribution $\mathcal{N}(0,\ 1)$ and its biases all to zero within the `build()` function. Then, when receiving the input from the previous state in the neural network, the `call()` function was called every time and in that all computations of the new parameters occurred: computation of the alignment scores ($tanh(W\underline{x} + \underline{b})$), removal of the last dimension, computations of the weights through the `softmax` function, reshaping to tensorflow format, computation of the context vector. All specific passages can be viewed inside the Python notebook.

At first, a `SimpleRNN` layer preceeded the `Attention` one, introducing a recurrent part within the architecture. However, the presence of this layer resulted into an irrelevant change in the performances, and therefore it was removed.

After some modifications to the architecture making it deeper by adding convolutions and playing with poolings, the last implemented idea was to focus on parallelization. This was done with the aim to make the net more robust, by adding different sub-modules working at different levels. Indeed, this was realised by initially concatenating two series of convolutions, equal in structure but different for the kernel size, in order to find different information. By expanding this idea to three parallel convolutional sub-nets, we obtained the architecture submitted as final one.