Tutorial for how to use our Contrastive Predictive Coding Framework for experiments on raw audio data

In this tutorial, we describe how to use our framework for custom projects.

The following assumes that you want to run the experiments locally on a GPU but it is also possible to do the same on Google Colab.

Libraries needed:

On a local machine you want to install tensorflow_io, which requires you to have tensorflow version 2.4.0. Unfortunately it does not work with 2.4.1. You also want the packages matplotlib, seaborn, numpy, scipy and sklearn.

Step-by-step to completed training with customization

The first step is to have a folder of audio files located at /.../CPC_Framework/data/custom_audio_dataset/

This folder needs to be filled with .wav files with a known sample rate and a known minimal duration. If you intend to visualize the learned embeddings, the filenames must indicate the labels (e.g. 7seven00032.wav for an audio file that contains a person saying the word seven).

The file only that you will absolutely need to modify to get a CPC model to train on your data is /.../CPC_Framework/scripts/params.py

In params.py, you will first need to set your training data path. This can be done in the following part of params.py:

```
111  # 1dconv encoder params (raw audio data)
112  if modelname == "1dconv_transformer/" or modelname == "1dconv_gru/":
113      enc_model = "1d_conv"
114      # location of raw audio to train cpc
115      path_data_cpc = os.path.join(project_path, "data/fma_small_mono_wav/")
116      # location of raw audio to generate embeddings
117      path_data_train = os.path.join(project_path, "data/GTZAN/")
118      path_data_test = os.path.join(project_path, "data/test_data/")
```

With the current settings, the data used for training CPC is located in data/fma_small_mono_wav/ in line 115. You will have to change this string to “data/custom_audio_dataset/”

Next in line 117 you will want to set the data that you will obtain representations for after training the CPC on “data/custom_audio_dataset/”. This can be set to the same data or another folder containing another subset of the audio data. Here it is important to have the same audio file type .wav with the same sampling rate.

Line 118 ought to be the folder location with a third test set to test a classification model that is trained on the representations that are learned by your CPC model.

The next important step in order to be able to train a CPC model is to change the data parameters set in the data_generator_arguments dictionary:

```
125     }
126     data_generator_arguments = {
127         "T": 27,  # num. timestep until current
128         "k": 3,  # num. timestep to predict
129         "N": 8,  # num. samples (N-1 = num. negative samples)
130         "full_duration": 4,  # sec, total length of a single sequence
131         "original_sr": 22050,  # Hz, sampling rate of original data
132         "desired_sr": 4410,  # Hz, sampling rate that is desired, used to down sample
133         "data_path": path_data_cpc,  # str, where to get raw audio data from
134     }
```

Here, you will first want to set “full_duration” to a duration lower than the minimal duration of your data (in seconds). This will be how much is sampled from your audio file to obtain a full positive sample that should be split into T+k segments. Next you will want to set “original_sr” to the sampling rate of your data and “desired_sr” to the sampling rate that you want to use for training. In setting T and k, it is necessary to first check whether full_duration * desired_sr / (T+k) evaluates to an integer. If not, you can often slightly change full_duration to achieve that.

After having set T, k, full_duration, original_sr and desired_sr, you need to set the number of samples, which will determine the number of negative samples (N-1)*k audio windows are the negative samples. N will simultaneously determine the batch size for training the model.

With data_generator_arguments set, you will want to think about the structure of your CPC model.

Let’s assume you will want to also use 1D strided convolutions for the encoder but you want to have control over the architecture. There’s two ways to influence the encoder model. The first is to look at this piece of code in params.py

```
119     encoder_args = {
120         "z_dim": z_dim,
121         "stride_sizes": [5, 4, 2, 2, 2],
122         "kernel_sizes": [10, 8, 4, 4, 4],
123         "n_filters": [512, 512, 512, 512, 512],
124         "activation": tf.nn.leaky_relu,
125     }
```

by changing the length of the stride_sizes lists, you can determine the number of convolutional layers in the encoder.

If however you want to have more direct control on the Encoder model, you should look at scripts/encoder_model.py.

Here you can set your own encoder model. However if you don't want to change more code make sure that it is still called Conv1DEncoder and takes the same arguments. If you want to have your own arguments, you will need to change the encoder_args dictionary in params.py (line 119) to have the relevant arguments. Make sure that you still set the variable z_dim in params.py to set the encoder's output vector length. If you also want to change the name of your EncoderModel layer, you will need to adjust the CPC model class in scripts/cpc_model.py, where you will need to import the newly defined layer and set it in the CPC model's init.

Given that you have taken care of the Encoder, you will next want to think about what Autoregressive Model you're intending to use.

Let's assume you want to use an LSTM instead of a GRU. To adjust that, you would go into scripts/autoregressive_model.py and change the GRU_AR tensorflow keras layer class or, similar to how you changed the Encoder model, adjust the imports in scripts/cpc_model.py

You are also able to use our pre-defined attention network autoregressive model by simply specifying the model name as "1dconv_transformer/" when running train_cpc.py .

The transformer's parameters can be set in params.py here:

```
# attentionAR params
elif modelname == "1dconv_transformer/" or modelname == "2dconv_transformer/":
    ar_model = "transformer"
    ar_args = {
        "num_enc_layers": 2,  # num. transformer encoder blocks
        "num_heads": 2,  # num. multiheads for attention
        "z_dim": z_dim,
        "dff": 100,  # num. units for 1st ffn within encoder block
        "dense_units": [100, c_dim],  # num. units for additional ffn
        "activation": tf.nn.leaky_relu,  # activation for additional ffn
        "maximum_position_encoding": data_generator_arguments["T"]
        + data_generator_arguments["k"],  # maximum length of a single sequence in steps
        "rate": 0.1,  # dropout rate
    }
```

You need to define the dimensionality of the context vector that is outputted by the autoregressive model of your choice by setting the variable c_dim in params.py.

That is all. You can now already train the CPC model on your data and obtain embeddings on your specified additional paths for the train and test set. You however will want to determine how many embeddings to create in the following part of params.py:

```
# -------------- generate embeddings params -------------------------

# How often to randomly sample from a single data to get different audio segments of length
num_em_samples_per_train_data = 90
num_em_samples_per_test_data = 5

```

Setting num_em_samples_per_train_data to a value of 90, given your “full_duration” that you previously set in the data_generator_arguments is lower than the minimal length of your audio files, means that each audio file is sampled from 90 times to create an embedding. With 1000 audio files, that makes 90.000 embeddings. Similarly for the test data.

So how do you train the model and obtain the embeddings?

We provide a bash script that you can run in your terminal by typing ./run_experiment.sh

```
1  #!/usr/bin/env bash
2
3  declare -a models=("1dconv_gru/")
4  #, "1dconv_transformer/" "2dconv_gru/")
5
6  # run entire script for each model in models
7  for model in ${models[@]}
8  do
9          # train cpc_model
10         python scripts/train_cpc.py -m $model
11         # get embeddings
12         python scripts/generate_embeddings.py -m $model
13         # train classifier and plot results
14         python scripts/train_classifiers.py -m $model
15 done
```

Here you want to comment out line 14 and set the model string in line 3 to "1dconv_transformer" if you want to use the transformer.

Running this script will let your model train. But before you start your training, you should set for how many epochs it should train and how high the learning rate should be, or what optimizer to use. You do that, again, in params.py:

```
72  # ------------- CPC training params -------------------------------
73
74  epochs_cpc = 700
75  steps_per_epoch_cpc = 100  # how many batch are fed in a single epoch
76  optimizer_cpc = tf.keras.optimizers.Adam(1e-5)
77
78
```

If you use a Nvidia GPU of compute capability of 7.0, you can choose to run in mixed precision mode by setting mixed_precision = True in params.py. This will make your model run in half-precision (16 bit float) and allow you to run bigger models (since parameters take up less memory). For some architectures this is also significantly faster.

What if the training failed

The training might fail due to a multitude of reasons. Either your data arguments (T, k, sampling rate, full_duration etc.) are incompatible with each other or with the audio files that you have (audio files should ideally all have the same length to prevent errors). It might also be that the custom encoder and autoregressive model do not work with the data. In this case, use the pre-defined models and have a look at the input and output shapes.

It might also be that your model does not fit into memory. In this case, downsampling, reducing T+k, reducing N or reducing full_duration might help. Otherwise you will have to use a smaller model.

After training
After training you will have a trained model weights saved in /results/cpc/(modelname) along with a saved numpy array of the training losses.

You will also have the embeddings once the code finishes (this takes quite a while) and you could go on without the framework to analyze them yourself. This would require to define the CPC model with all the same parameters used for your raw audio data, building the model by calling it on some input and then using the load_weights(weight_path) method. Or you could just use our script in scripts/generate_tSNE.py to do the visualizations.

If you decide to use the script however, you will need to change the specifics inside the plotting functions, including names of data sets, color palette to use (a list of colors to use for the classes) and the strings for the individual categories called “classes”, that are given in the audio filenames. You would have to change those things inside scripts/generate_tSNE.py before running it:

python scripts/generate_tSNE.py -m “1dconv_transformer/”

This will take the embeddings that you created and run a t-SNE dimension reduction on it down to 2 dimensions that can be plotted. You will get the results stored in some folder in results/classifier/(modelname)/… that will be created automatically.

Changing loss function, prediction transformations and similarity measure

It might be a good idea to adjust the loss function or change how the CPC model makes the k predictions based on the context embedding c_T. By default this is just a linear projection but it can be changed to be any function that outputs k vectors of length z_dim (which is the output dimension of the encoder and is specified in params.py).

To change this, you will need to change what is done inside the Predict_z layer class in /scripts/cpc_model.py . Look at what the output dimensions are supposed to be.
Another thing that can be changed is the similarity measure calculation. This is done in the class Predict_z in the same python script. Instead of taking the exponential of the dot product of the prediction and the positive and negative encoded future time-steps, you can use any other similarity score. We have tried using a related similarity measure, which is cosine similarity which did not give any good results.

To change the loss (e.g. to change how the average over the k prediction losses is weighted - right now it’s the mean), look at the InfoNCE class, also in cpc_model.py