LAB 5

COVID-19 DETECTION FROM COUGHS: AUDIO CLASSIFICATION

Spoken and Written Language Processing

DATA SCIENCE AND ENGINEERING
Polytechnic University of Catalonia

Alina Castell Blasco - 49188689S

Adrián Cerezuela Hernández - 48222010A

Claudia Len Manero - 53869554

Ramon Ventura Navarro - 21785256R

May-June 2024

Contents

1 Introduction 1

2 Experiments with VGG approach 1

3 Experiments with HuBERT approach 2

3.1 Hyperparameters tuning 2

3.2 Simpler final projection layers 4

3.3 Unfreeze last two layers of HuBERT model 5

3.4 Other pre-trained models 5

4 Weighted average of layer's hidden states with trained parameter 6

5 Larger batch sizes o models using gradient accumulation 7

6 Experiments with other 'Speech embeddings' 8

6.1 Wav2vec 8

6.2 PASE model 10

7 Improving cross-validation strategy 12

8 Conclusions 12

1 Introduction

In this assignment we are asked to work on an audio classification task which consists in COVID-19 diagnosis using a small dataset of breathing, cough, and voice recordings. This entails categorizing audio data into distinct predefined groups according to its content or attributes through the usage of machine learning. Firstly, our data must be explored in order to know better relations and acquire insights of it. Then we will be able to test with different models that might include some features seen in class along the course. The conclusions will be discussed in the final point of the report.

2 Experiments with VGG approach

In this section we will try to improve the VGG model given that achieves an AUC of 64.9%. For that we did some parameter optimization. We tried different window types, model sizes, pooling layer combinations, number of Fast Fourier Transform (n_fft) parameter, number of Mel Bands (nmels), the window size and window stride. Next, we analyze the results given.

The strategy followed was adding the best working features in each iteration, meaning every new improved model was then used as the base one to continue testing variations.

Performance of VGG model									
Architecture	Pooling	Window	window_size	window_stride	n_fft	nmels	Loss	AUC	Epoch
VGG13	max-mean	Hamming	0.04	0.02	2048	80	0.7669	65.2%	9
VGG13	max-mean	Hann	0.04	0.02	2048	80	0.6619	65.4%	10
VGG13	max-mean	Blackman	0.04	0.02	2048	80	0.6850	**65.9%**	10
VGG13	max-mean	Blackman	0.04	0.02	1024	80	0.6658	65.7%	10
VGG13	max-mean	Blackman	0.04	0.02	4096	80	0.6714	65.9%	8
VGG13	max-mean	Blackman	0.04	0.02	4096	40	0.6534	67.3%	10
VGG13	max-mean	Blackman	0.04	0.02	4096	60	0.7125	65.7%	12
VGG13	max-mean	Blackman	0.04	0.02	2048	40	0.6630	66.1%	11
VGG13	max-mean	Blackman	0.04	0.02	2048	60	0.7510	66.3%	9
VGG13	max-mean	Blackman	0.02	0.02	4096	40	0.6649	66.9%	11
VGG13	max-mean	Blackman	0.015	0.02	4096	40	0.6605	66.6%	9
VGG13	max-mean	Blackman	0.04	0.015	4096	40	0.6435	68.7%	11
VGG13	max-mean	Blackman	0.04	0.025	4096	40	0.7023	66.5%	11
VGG13	avg	Blackman	0.04	0.015	4096	40	0.6490	69.7%	18
VGG13	max	Blackman	0.04	0.015	4096	40	0.6540	67.5%	13
VGG11	**avg**	**Blackman**	**0.04**	**0.015**	**4096**	**40**	**0.6484**	**70.9%**	**16**
VGG16	avg	Blackman	0.04	0.015	4096	40	0.6795	70.7%	17
VGG19	avg	Blackman	0.04	0.015	4096	40	0.6839	68.5%	16

Table 1: Hyperparameter tuning for VGG model.

To start our experiments we tried different windows as it is crucial in transforming the data into the frequency domain.

The base model was using Hamming window, which is good reducing spectral leakage Hamming Reduced spectral leakage and provides good balance of resolution, the Hann window also reduces leakage and has a symmetrical shape and the Blackman window has minimal spectral leakage. We found that the best window in our case was the Blackman window, giving us a 0.5% or more improvement in AUC.

Next, we tried different n_fft values, normally higher n_fft gives better frequency resolution and richer feature representation but can cause overfitting. A lower n_fftgives better time resolution and reduced computational cost but gives lower frequency resolution, less detailed feature representation and creates potential information loss.

We tried with $n\_fft = 1024$, $n\_fft = 2048$ and $n\_fft = 4096$ and obtained the same AUC for both 2048 and 4096 tests, that is why for the next optimization be used both $n\_fft = 2048$ and $n\_fft = 4096$.

Our next modification was $nmels$ as and tried changing the default $nmels = 80$ to $nmels = 60$ and $nmels = 40$ and found that the model using $nmels = 40$ and $n\_fft = 4096$ had the best performance with $AUC = 67.3\%$. This is because higher nmels values have a potential risk for overfitting while lower nmels value reduce this risk.

To continue with our experiments we changed the window size of the model, which controls the duration of the sliding window used for computing the Mel spectrogram. We tested $window\_size = 0.15$ and $window\_size = 0.2$ as lower window sizes allow the model to better represent all changes in the audio and catching all features. In this case, neither of the tried values outperformed the previous window_size value. This may be because of the risk of aliasing associated with lower window sizes and its limited contextual information.

Our next variation was for the window_stride value and we tried $window\_stride = 0.15$ and $window\_stride = 0.25$. In this iteration we found that a lower value was better for our model giving us an improvement of 1.4% in AUC.

Before ending our experiments by trying different model sizes, we changed the pooling layers and tried Average Pooling and Max Pooling instead of Max-Mean. Here we found that Average Pooling was working better than the other 2 strategies in our model, giving us $AUC = 69.7\%$.

Finally, we changed the models size and tried all the other proposed variants and found the best working ones were VGG11 and VGG16, which differed in AUC by 0.2%.

To try and improve the final results we augmented the patience to 10 epochs and computed the 2 best performing models again.

Performance of VGG model with patience 10									
Architecture	Pooling	Window	window_size	window_stride	n_fft	nmels	Loss	AUC	Epoch
VGG11	**avg**	**Blackman**	**0.04**	**0.015**	**4096**	**40**	**0.6971**	**71.2%**	**33**
VGG16	avg	Blackman	0.04	0.015	4096	40	0.7487	70.5%	31

Table 2: Training of best models with $patience = 10$

In the VGG16 architecture it didn't improve results, but for our best model using the VGG11 it actually improved results by 0.3%.

To conclude this section we have found that the best VGG model for COVID-19 classification based on audios is the one trained in 33 epochs with $patience = 10$ using the VGG11 architecture, Average Pooling layer, the Blackman window, $window\_size = 0.04$, $window\_stride = 0.015$, $n\_fft = 4096$ and $nmels = 40$. It improves

3 Experiments with HuBERT approach

In this section we have experimented with the HuBERT model baseline code to try and find the best combination of modifications, that is, the one that achieves highest AUC and lowest loss. In the following sections we explain these modifications, their descriptions and results obtained.

3.1 Hyperparameters tuning

Started modifications on hyperparemeters: Batch size=(22, 18, 12), Optimizer=(ADAM, SGD), LearningRate=(0.0002, 0.002, 0.005, 0.01), Patience=(5, 10).

Preliminary results table can be seen below. Note we also specify the epoch number at which the results were found, if those numbers are lower than 5, another experiment has been tried with the same hyperparameters but increasing the patience value:

Performance of HuBERT model						
BatchSize	Optimizer	Lr	Patience	Loss	AUC	Epoch
22	ADAM	0.0002	5	Original	**70.0%**	11
22	ADAM	0.002	5	0.6364	71.1%	4
22	ADAM	0.002	10	0.6374	71.7%	4
22	ADAM	0.005	5	0.6930	50.5%	1
22	ADAM	0.005	10	0.6930	50.5%	1
22	ADAM	0.01	5	0.6932	50.0%	1
22	ADAM	0.01	10	0.6932	50.0%	1
18	ADAM	0.0002	5	0.6414	**69.7%**	9
18	ADAM	0.002	5	0.6529	67.5%	2
18	ADAM	0.002	10	0.6513	68.5%	12
18	ADAM	0.005	5	0.7203	58.6%	2
18	ADAM	0.01	5	0.8574	52.6%	2
12	ADAM	0.0002	5	0.6452	69.0%	1
12	ADAM	0.0002	10	0.6452	**69.5%**	9
12	ADAM	0.002	5	0.6550	68.6%	3
12	ADAM	0.002	10	0.6550	68.6%	3
22	SGD	0.0002	5	0.6903	66.5%	50
22	SGD	0.002	5	0.6857	**68.1%**	6
18	SGD	0.0002	5	0.6895	67.0%	50
18	SGD	0.002	5	0.6701	68.0%	8
12	SGD	0.0002	5	0.6864	67.5%	48
12	SGD	0.002	5	0.6630	67.8%	6

Table 3: Hyperparameter tuning for HuBERT model.

From the results in the table we can extract some conclusions:

- Clearly the Learning Rates of 0.01 and 0.005 are too big for the ADAM optimizer since their AUC is the worst result of all, even with 10 epochs of patience. As you can see we only tried increasing the number of patience for the first trials with these learning rate values, then we saw that there was no improvement and we did not tried with the lower batch sizes experiments.
- The trials with 10 epochs of patience were the ones that took longer, duplicating in some cases the execution time. Despite that, the increase of patience improves the performance with respect to the patience of 5 epochs, so we can safely assume that a patience of 10 is beneficial for the model.
- As for the optimizers, Adam adapts the learning rate for each parameter using estimates of first and second moments of the gradients, while SGD uses a constant learning rate. SGD may require more epochs to converge, often getting stuck in local minima, whereas Adam converges faster and more efficiently. Adam's rapid learning can sometimes lead to instability, potentially causing divergence or failing to sustain training over many epochs, while SGD's slower, steady updates allow it to train over more epochs. This explains why the SGD optimizer often reaches the limit of 50 epochs and presents a more stable evolution of the AUC results than the Adam optimizer, which obtains better results overall.
- Also, the bigger the batch size the better the AUC.
- We don't seem to outperform the original model with any of the configurations tried. Therefore we will continue the study mainly focusing on this configuration.

After observing that batch sizes could be increased instead of decreased, we did some more experiments by changing them, the learning rate and the patience parameters, which did have a positive effect. The results are presented in the next table:

Performance of HuBERT model						
BatchSize	Optimizer	Lr	Patience	Loss	AUC	Epoch
26	ADAM	0.002	10	0.7624	67.5%	9
30	ADAM	0.002	10	0.6446	**72.3%**	6
35	ADAM	0.002	10	0.6438	69.0%	7
40	ADAM	0.002	10	0.6320	71.3%	5
60	ADAM	0.002	10	0.7403	71.1%	13

Table 4: Hyperparameter tuning for HuBERT model.

We have found a model with a **AUC** of **72.3%** by using a batch size of 30, an ADAM optimizer, learning rate of 0.002 and patience of 10. This combinations of hyperparameters presents as a potential model improvement.

Max-Mean Pooling Layers

Aside from the hyperparameter tuning, we have also tried modifying the pooling layers by introducing a *max-mean pooling* layer instead of only a *max pooling* layer. The results were surprisingly good when trying this pooling layer on the original baseline model, while when trying with this last combination of best hyperparameters the results worsened incredibly. See the results on the table below:

Performance of HuBERT model						
BatchSize	Optimizer	Lr	Patience	Loss	AUC	Epoch
22	ADAM	0.0002	5	0.6439	**72.8%**	8
30	ADAM	0.002	10	0.6866	69.4%	7

Table 5: Max-Mean Pooling layer for HuBERT model.

Given these results, we can safely state that on the first section of the HuBERT model tuning we have obtained a considerably improved model that achieves a **72.8%** of AUC.

3.2 Simpler final projection layers

The original model used a two-layer adaptor and a two-layer classifier to process the features extracted by the HuBERT model. We simplified these layers by reducing their complexity to a single linear layer each. However, to ensure sufficient non-linearity, we included a ReLU activation function in both the adaptor and the classifier layers. We execute this modified model with the baseline hyperparameters and with the fine-tuned ones.

As for the model only with simplified final projection layers, we decided to use the hyperparameters of the original baseline code and the best ones obtained in the hyperparameter tuning. We could not use the *max-mean pooling* layer since it was applied on the hidden size of the classifier layer that we have just reduced, instead we used the combination of batch size, learning rate and patience. The results are the following:

Performance of HuBERT model						
BatchSize	Optimizer	Lr	Patience	Loss	AUC	Epoch
22	ADAM	0.0002	5	0.6591	**70.9%**	7
30	ADAM	0.002	5	0.6931	50.0%	1

Table 6: Simpler final projection layers.

First, the best hyperparameter combination obtained previously has a negative effect over the reduction of the final projection layers. A solution to this issue could be to decrease the learning rate, but it does not make sense to start experimenting now since we have taken that combination as the best one. Lastly, the approach itself does have a positive effect since the AUC value is improved from the baseline code. Still, not as much as the improvement using the max-mean pooling layers.

3.3 Unfreeze last two layers of HuBERT model

In addition, we decided to unfreeze the last two layers of the HuBERT model's feature extraction. This approach, inspired by Layer-wise Network Adaptation (LNA) fine-tuning, allows the model to adapt more flexibly to the new task while retaining most of the pre-trained knowledge in the lower layers. We created a function that unfreezes the last specified layers after having freezed all the layers using the baseline code function. With this modification we try to observe if a more fine-tuned model for our specific task achieves better performance than the general HuBERT model.

The results we obtained on the first iteration provided a loss of 0.6647 on the Epoch 5 and an **AUC of 67.2%** on the Epoch 1, it stopped running on the fifth epoch. Thus, we tried increasing the patience value to 10 in order to see if further training epochs provided a higher improvement.

We see a great improvement when training the model with a patience of 10 epochs. We observe that the AUC value increases until the epoch 14 where it achieves its **best value of 73.2%**, then it decreases until it stops in the 24th epoch (which is logic given the patience value defined at 10 epochs). In particular, the behaviour of the unfreezed model is the following: starts with a validation AUC of 67.2%, then it decreases until the 8th epoch, where it reaches a 69.8%, and then stars to improve until the 14th epoch. So, we can assume that increasing the number of patience actually improves the performance of the model. Despite that, there is a clearly an effect over the execution time, when trained for 5 epochs it took roughly 11 minutes, whereas this second trial took approximately 44 minutes to complete.

In the next table we present a comparison between this unfreezed version of the original baseline code, the unfreezed version of the best hyperparameters found in the first section and the unfreezed version of the max-mean pooling version.

Performance of HuBERT model							
BatchSize	Optimizer	Lr	Patience	Pooling	Loss	AUC	Epoch
22	ADAM	0.0002	5	max	0.6169	**73.2%**	14
30	ADAM	0.002	10	max	0.6931	50.0%	1
22	ADAM	0.0002	10	max-mean	0.7155	72.6%	10

Table 7: Unfreeze of the last two layers.

Similarly to the effect seen on the section 3.2, we see that unfreezing the last two layers plus using the combination of 30 batch size, 0.002 learning rate and 10 patience worsens considerably the model's performance. We believe that reason behind these worse results is the high learning rate. It can cause instability and poor convergence, especially when fine-tuning the last layers, which can lead to less optimal adjustments in the weights. This might result in the model converging too quickly or even diverging, as observed in the single epoch training. As for the max-mean pooling it can potentially amplify noisy or conflicting gradient signals when the model's layers are unfrozen, which can lead to instability or overfitting to specific features, worsening the AUC results.

To summarize, we can safely assume that unfreezing part of the model to fine-tune the training for our specific task of COVID-19 cough detection, does make a difference in the performance. In fact, this version achieves a 73.2% better performance with respect to the original one of 70.0%. For this reason, we will be working with this new baseline code in the next sections.

3.4 Other pre-trained models

In this section we experimented with other HuBERT pre-trained models. First we tried the one called `MODEL = "facebook/hubert-large-ll60k"`, a model larger than the 'base' model we have been using, to see if the performance improved; which was something expected since usually performance improves with bigger models. However, we couldn't execute due to excess of memory, that is the size of the model was too big for our capacity. Thus, we decided to implement the gradient accumulation technique (explained on section 5) to see if we could solve it. We tried running the *'large'* version of the HuBERT model with a gradient accumulation of 3 but still got an error saying that the notebook was trying to use more disk space than is available. Still, the best **AUC** found before running out of disk space was of **71.1%** on the last Epoch number 17, after a constant positive evolution of the previous epochs. These results tell us that, if it weren't for the space issue, we could have obtained much better results on the following epochs

Since *'large'* version did not succeed, we tried running the code using a model we found on Hugging Face, which was the HuBERT model but a tinier version: `MODEL = "ybelkada/hubert-tiny-random"`. The results on this model were worse than expected, achieving only a **AUC: 61.4%**. We believe the reason behind this results is the size of the model used, that is why we kept on using the 'base' HuBERT model.

4 Weighted average of layer's hidden states with trained parameter

In an effort to further improve our performance with the HuBERT pre-trained model, we explored a new approach apart from the presented in the previous section: using the weighted average of layer's hidden states with trained parameters, rather than only relying on the last hidden state.

Traditionally, in recurrent neural networks (RNNs) and their variants like Long Short-Term Memory (LSTM) networks, only the last hidden state is considered for prediction. However, this approach may overlook valuable temporal information encoded in the intermediate hidden states of the network layers.

By incorporating a weighted average of these intermediate hidden states, we aim to capture a more comprehensive representation of the input audio data. This method involves assigning weights to each hidden state based on its relevance to the task at hand. These weights are learned during the training process and are utilized to compute the final representation. The utilization of weighted averages allows us to harness the collective information present across all layers of the neural network, potentially leading to a richer and more discriminative feature representation. This approach is particularly advantageous in tasks where temporal dependencies play a crucial role, such as our case of audio detection. These are the main modifications from the HuBERT class used before:

```
                                    ...
        self.hubert = HubertModel.from_pretrained(MODEL)

        num_layers = self.hubert.config.num_hidden_layers + 1  # transformer layers + inp
ut embeddings
        self.layer_weights = nn.Parameter(torch.ones(num_layers) / num_layers)

        hidden_size = self.hubert.config.hidden_size
                                    ...
        output_hidden_states = True
        # x shape: (B,E)
        outputs = self.hubert(x, output_hidden_states=output_hidden_states)

        hidden_states = outputs[1]
        hidden_states = torch.stack(hidden_states, dim=1)
        norm_weights = nn.functional.softmax(self.layer_weights, dim=-1)
        hidden_states = (hidden_states * norm_weights.view(-1, 1, 1)).sum(dim=1)

        hidden_states = self.adaptor(hidden_states)

        # pooling
        x, _ = hidden_states.max(dim=1)
                                    ...
```

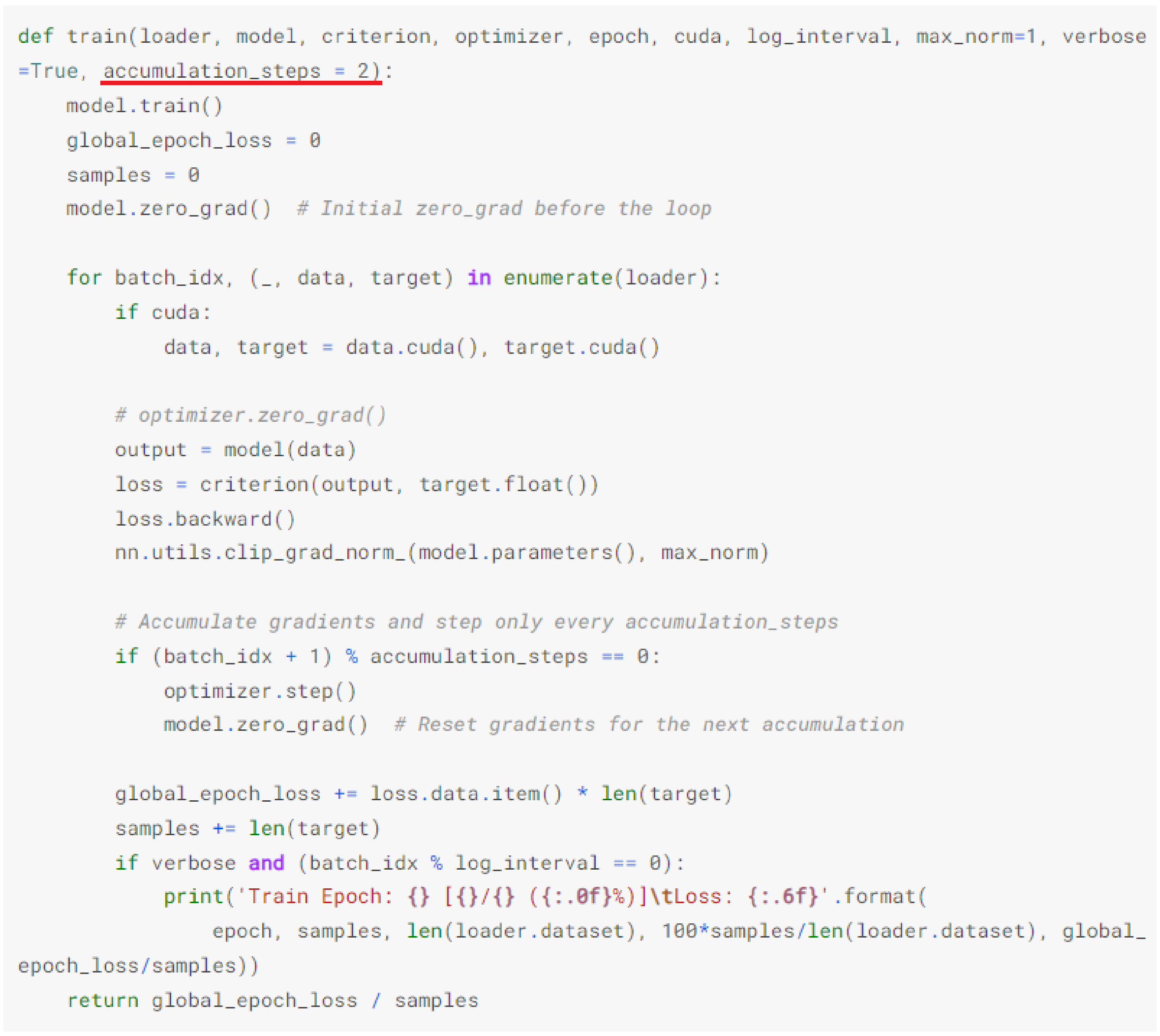
A single linear layer both for the adaptor and classifier has been used, followed by a ReLU activation function. Also the unfreezing of the last two layers has been executed. The configuration of the model with an AUC of 73.2% explained before has been used. It consists of the following parameters: *Batch size 22 — Optimizer adam — lr 0.0002 — Momentum 0.9 — Patience 10*. Running this, we find that the model's training phase lasts for 14 epochs until it stops, with the best AUC achieved at epoch 4, reaching 70.2%. This result is not bad at all, but it does not improve upon our new baseline model when implementing the weights. Therefore, the previous model will be used, utilizing only the last hidden layer as a feature for the model.

5 Larger batch sizes o models using gradient accumulation

Gradient accumulation is a powerful technique that addresses the need for larger batch sizes, which are known to facilitate better model convergence and the ability to utilize higher learning rates. This technique involves accumulating gradients over multiple mini-batches before performing a model update. By doing so, it effectively simulates a larger batch size without demanding additional memory, making it an interesting approach to overcoming batch size limitations.

This is achieved by modifying the training loop to call `optimizer.step()` after every n batches, and subsequently resetting the gradients with `optimizer.zero_grad()`. This approach not only enhances the training process but also provides a more flexible and efficient way to manage computational resources.

All of the executions in this section are based on a model with *BatchSize = 22, Optimizer = ADAM, Lr = 0.0002* and *Patience = 10* with the last two layers unfrozen, which has shown to be the best performing model overall (with **AUC** of **73.2%**). The implemented code and the results regarding this approach can be found below:



```
def train(loader, model, criterion, optimizer, epoch, cuda, log_interval, max_norm=1, verbose
=True, accumulation_steps = 2):
    model.train()
    global_epoch_loss = 0
    samples = 0
    model.zero_grad()  # Initial zero_grad before the loop

    for batch_idx, (_, data, target) in enumerate(loader):
        if cuda:
            data, target = data.cuda(), target.cuda()

        # optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target.float())
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm)

        # Accumulate gradients and step only every accumulation_steps
        if (batch_idx + 1) % accumulation_steps == 0:
            optimizer.step()
            model.zero_grad()  # Reset gradients for the next accumulation

        global_epoch_loss += loss.data.item() * len(target)
        samples += len(target)
        if verbose and (batch_idx % log_interval == 0):
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, samples, len(loader.dataset), 100*samples/len(loader.dataset), global_
epoch_loss/samples))
    return global_epoch_loss / samples
```

Performance of HuBERT model							
AccumulationSteps	BatchSize	Optimizer	Lr	Patience	Loss	AUC	Epoch
2	22	ADAM	0.0002	10	0.6426	71.1%	6
4	22	ADAM	0.0002	10	0.6205	72.1%	6
5	22	ADAM	0.0002	10	0.6575	**72.2%**	10
6	22	ADAM	0.0002	10	0.6227	71.8%	4

Table 8: Accumulation steps tuning for gradient accumulation approach.

Although this method significantly increases the effective batch size, it does not lead to an improvement in model performance. This suggests that merely increasing the batch size is insufficient. Therefore, we have also experimented with increasing the learning rate using 5 and 6 accumulation steps, as well as increasing the batch size at the same time:

Performance of HuBERT model							
AccumulationSteps	BatchSize	Optimizer	Lr	Patience	Loss	AUC	Epoch
5	22	ADAM	0.0002	10	0.6575	**72.2**%	10
5	22	ADAM	0.002	10	0.6931	55.3%	1
5	22	ADAM	0.01	10	0.7213	55.3%	2
6	50	ADAM	0.0002	10	0.6246	71.9%	8
6	50	ADAM	0.01	10	0.6989	51.9%	2

Table 9: Batch size and learning rate tuning for gradient accumulation approach.

After experimenting with various batch sizes and learning rates, both separately and together, we discovered that maintaining smaller values for both consistently produced the best outcomes. For each execution we observe that configurations that performed the best consistently favored smaller values. This highlights the effectiveness of sticking to lower batch sizes and learning rates for achieving optimal results, suggesting that alternative strategies may be more effective than the gradient accumulation approach.

6 Experiments with other 'Speech embeddings'

In this final section of experiments with *Speech Embeddings*, we are asked to try different ones. In the previous notebooks, we are given three embedding options: Mel Spectrogram + VGG, HuBERT, or DistilHuBERT. Here, we will try to use other pre-trained models from the transformers library. Specifically, we will test with *Wav2vec* and *PASE*.

6.1 Wav2vec

Wav2vec, developed by Facebook AI Research, is a state-of-the-art model for learning audio representations directly from raw waveforms, primarily used for automatic speech recognition and other audio processing tasks, including audio detection, that is the taks in which we are involved in this lab assignment.

The procedure employed is the same as for the pre-trained *HuBERT* model. The same notebook has been used, but with the downloaded model changed, as well as the class corresponding to its implementation. The model has been downloaded from "*facebook/wav2vec2-base-960h*". Specifically, the code snippets that differ from previous implementations are presented below. The *freeze_future_encoder()* method is also included, but only the main changes in the class implementation are shown.

```python
class Wav2Vec2ForAudioClassification(nn.Module):
    def __init__(self, adapter_hidden_size=64):
        super().__init__()

        self.wav2vec_model = Wav2Vec2Model.from_pretrained(MODEL)

        hidden_size = self.wav2vec_model.config.hidden_size

        self.adaptor = nn.Sequential(
            nn.Linear(hidden_size, adapter_hidden_size),
            nn.ReLU(True),
            nn.Dropout(0.1),
            nn.Linear(adapter_hidden_size, hidden_size),
        )

        self.classifier = nn.Sequential(
            nn.Linear(hidden_size, adapter_hidden_size),
            nn.ReLU(True),
            nn.Dropout(0.1),
            nn.Linear(adapter_hidden_size, 1),
        )
...
    def forward(self, x):
        # x shape: (B,E)
        x = self.wav2vec_model(x).last_hidden_state

        x = self.adaptor(x)

        # pooling
        x, _ = x.max(dim=1)

        # Multilayer perceptron
        out = self.classifier(x)
        # out shape: (B,1)

        # Remove last dimension
        return out.squeeze(-1)
        # return shape: (B)
```

As we can see, the structure remains the same, as does the procedure followed. The training of this model has been carried out both with and without the use of gradient accumulation, as explained in the section. The first thing to note is that training these models took much longer than the pre-trained *HuBERT*. Due to the slower training process, we opted to select the best configuration from the previous sections and apply *Wav2Vec* to it. In another case, as seen next with the *PASE* model, the training time allowed us to perform some tuning and find the best model within the same architecture. The configuration mentioned, as well as the results obtained, are summarized in the following table:

Performance of Wav2Vec model			
AccGradient	Loss	AUC	Epoch
accumulation_steps = 2	0.6924	63.4%	4
accumulation_steps = 4	0.6927	62.8%	6
No	0.6650	70.2%	9

Table 10: Wav2Vec architecture performance for pair of approaches

From the table, we can see that the *Wav2Vec* architecture model with the best AUC for our audio recognition task is the one that does not use the gradient accumulation technique, achieving an AUC of 70.2%. The configuration used is the one shown, which, as previously mentioned, corresponds to the best model obtained in earlier sections. No further tests were conducted with a higher number of accumulation steps because it significantly decreases the AUC percentage, much more so compared to our best *Wav2Vec* model.

At this point, we cannot claim that the *Wav2Vec* architecture is an improvement over a pre-trained *HuBERT*, as it takes much longer to train and provides slightly lower results. However, we will continue with the comparison between Speech Embeddings, aiming to find an architecture that can match or improve the results obtained in previous sections with *HuBERT*. The next alternative we will present is a *PASE* model.

6.2 PASE model

After reviewing the results with *Wav2Vec*, we will now focus on using *PASE* (Problem-Agnostic Speech Encoder) as an alternative to *HuBERT*. *PASE* is an innovative approach that combines supervised and unsupervised learning techniques to generate robust audio Problem-agnostic means that can be adapted to a wide variety of audio processing tasks without requiring significant adjustments to its basic architecture. The main goal of *PASE* is to provide an audio embedding that is rich and flexible enough to be used in various audio processing tasks, including audio event detection. Its class implementation is as follows:

```
class PASE(nn.Module):
    def __init__(self, input_size=256, hidden_size=64, num_layers=1, seqlen=800):
        super(PASE, self).__init__()

        self.pase = wf_builder('/kaggle/input/problem-agnostic-speech-encoder-pase/pase-
master/pase-master/cfg/frontend/PASE+.cfg')
        self.pase.load_pretrained('/kaggle/input/problem-agnostic-speech-encoder-pase/FE
_e199.ckpt', load_last=True, verbose=True)

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.seqlen = seqlen

        self.lstm = nn.LSTM(input_size, self.hidden_size, self.num_layers, batch_first=T
rue, dropout=0.2, bidirectional=False)

        # Adaptive Average Pooling (output has always the same seqlen size)
        self.pool = nn.AdaptiveAvgPool1d(self.seqlen)

        self.classifier = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(True),
            nn.Dropout(0.1),
            nn.Linear(hidden_size, 1),
        )

    def forward(self, x):

        # PASE needs an additional dimension
        x.unsqueeze_(1)
        # (B,1,16000 x max_seconds)

        out = self.pase(x)
        # out shape: (B,256,100 x max_seconds)

        out = self.pool(out)
        # out shape: (B,256,seqlen)

        out = out.transpose(1,2)
        # out shape: (B,seqlen,256)

        out, _ = self.lstm(out)
        # out shape: (B,seqlen,H)

        out, _ = out.max(dim=1)
        # out = out.mean(dim=1)
        # out shape: (B,H)

        out = self.classifier(out)
        # out shape: (B,1)

        return out.squeeze(-1)
```

The source of the implementation is the notebook provided for the Kaggle competition. The baseline model has an AUC of 71.0%. By performing a slight hyperparameter tuning, we aim to improve this metric so that we can compare the previously obtained results with the hypothetical best possible model of *PASE*. The results obtained are shown below:

Performance of PASE model						
optimizer	lr	batch_size	momentum	Loss	AUC	Epoch
Adam	0.0001	20	0.9	0.6260	71.0%	28
Adam	0.0001	20	0.5	0.6260	71.0%	28
Adam	0.0001	15	0.9	0.6285	70.7%	28
Adam	0.0001	15	0.5	0.6285	70.7%	28
Adam	0.0005	20	0.9	0.6739	72.6%	21
Adam	**0.0005**	**20**	**0.5**	**0.6739**	**72.6%**	**21**
Adam	0.0005	15	0.9	0.6356	71.9%	14
Adam	0.0005	15	0.5	0.6356	71.9%	14
Adam	0.001	20	0.9	0.7364	72.2%	22
Adam	0.001	20	0.5	0.7364	72.2%	22
Adam	0.001	15	0.9	0.6832	71.5%	18
Adam	0.001	15	0.5	0.6832	71.5%	18
SGD	0.0001	20	0.9	0.6924	53.3%	8
SGD	0.0001	20	0.5	0.6927	52.4%	8
SGD	0.0001	15	0.9	0.6903	57.0%	41
SGD	0.0001	15	0.5	0.6925	52.8%	8
SGD	0.0005	20	0.9	0.6812	59.9%	41
SGD	0.0005	20	0.5	0.6924	53.3%	8
SGD	0.0005	15	0.9	0.6747	61.3%	46
SGD	0.0005	15	0.5	0.6903	56.9%	41

Table 11: Hyperparameter tuning for PASE model

In the table shown, we observe several points. Notably, based on previous results, patience has been set to a value of 10. Firstly, the best model gives us an AUC of 72.6%, which is not the best model obtained throughout the practice, as we have some slightly better ones in earlier sections, but it does improve upon the baseline, so we consider the result satisfactory. We can also see that in the first part of the tests, using the *Adam* optimizer, the results are not influenced by the momentum, so this hyperparameter can be disregarded in the analysis. It is also worth noting that slightly increasing the learning rate, without exceeding a certain value, slightly improves the model's performance. However, as seen previously, the *SGD* optimizer is not the best for our task, and this model demonstrates this once again.

At this point, the best result from an alternative Speech Embedding to the pre-trained *HuBERT* would be the *PASE* model we just reviewed, highlighted in bold in the table. However, it is not better than the best *HuBERT* one. Additionally, an attempt was made to compare it with a *VGGISH* model. Initially, the baseline model for this task was not provided, but an attempt was made to implement it using the version found in the personal repository of the course instructor. However, due to input incompatibility, we were unable to do so successfully.

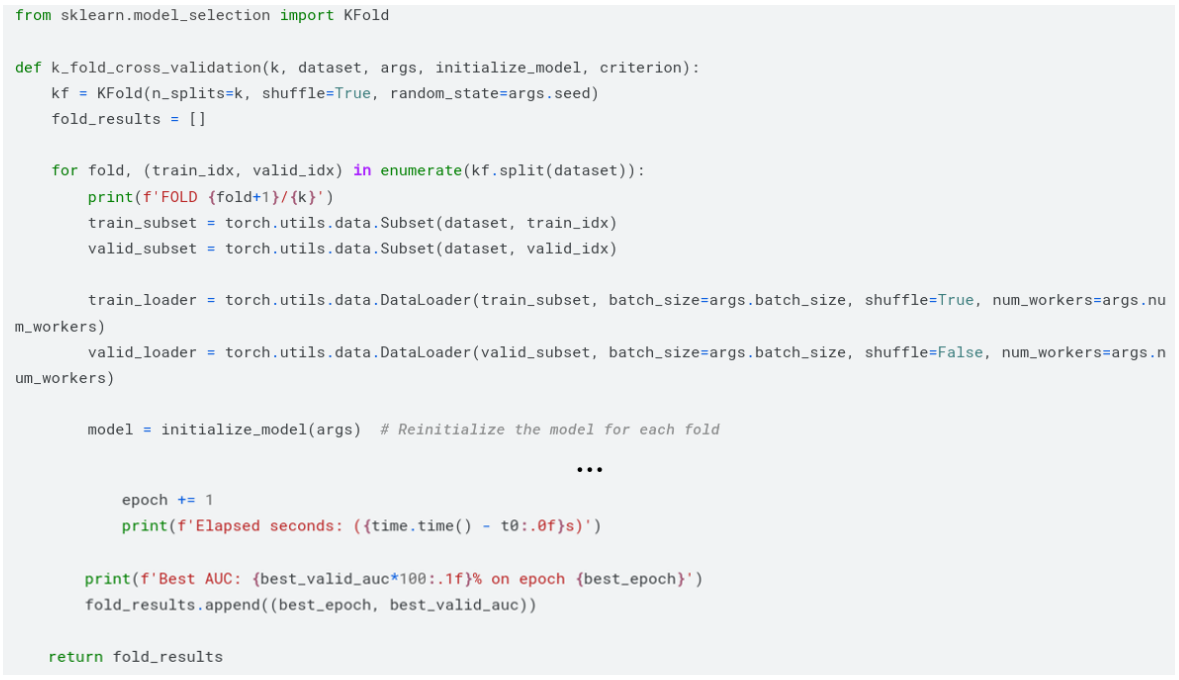
Despite this, we did not expect a better performance than either of the two models tested. This assertion is based, on the one hand, on the results obtained by the instructor in the baseline models, where the *VGGISH* is slightly inferior, and on the other hand, on the leaderboard of the competition linked in the explanatory slides of the practice from the previous year, where among all the models, the *VGGISH* had the lowest AUC.

7 Improving cross-validation strategy

Cross-validation serves as a crucial tool in assessing the performance of models. It involves partitioning the dataset into multiple subsets, training the model on one subset, and validating it on the remaining data. This iterative process provides insights into how well the model generalizes to unseen data.

In situations where data is limited, as is the case in our study, K-fold cross-validation becomes an effective approach. With scarce data, K-fold cross-validation mitigates the risk of overfitting by ensuring that each data point is used for both training and validation multiple times by repeatedly shuffling and splitting the data into different folds.

Taking this into account, we have implemented the following function, which appropriately performs K-fold cross-validation and consists of a loop with the k iterations mentioned:



```
from sklearn.model_selection import KFold

def k_fold_cross_validation(k, dataset, args, initialize_model, criterion):
    kf = KFold(n_splits=k, shuffle=True, random_state=args.seed)
    fold_results = []

    for fold, (train_idx, valid_idx) in enumerate(kf.split(dataset)):
        print(f'FOLD {fold+1}/{k}')
        train_subset = torch.utils.data.Subset(dataset, train_idx)
        valid_subset = torch.utils.data.Subset(dataset, valid_idx)

        train_loader = torch.utils.data.DataLoader(train_subset, batch_size=args.batch_size, shuffle=True, num_workers=args.nu
m_workers)
        valid_loader = torch.utils.data.DataLoader(valid_subset, batch_size=args.batch_size, shuffle=False, num_workers=args.n
um_workers)

        model = initialize_model(args)  # Reinitialize the model for each fold
                                        ...
            epoch += 1
            print(f'Elapsed seconds: ({time.time() - t0:.0f}s)')

        print(f'Best AUC: {best_valid_auc*100:.1f}% on epoch {best_epoch}')
        fold_results.append((best_epoch, best_valid_auc))

    return fold_results
```

However, due to memory limitations, an Out Of Memory error was encountered when trying to implement more than one fold. Knowing this issue and having learned about gradient accumulation and its benefits in terms of memory efficiency, we tried to implement it to see if it could serve as a solution to our problem. Regardless of the accumulation steps we set or the number of folds in we split the data, we still experienced OOM errors after several iterations during the first fold. That being said, although this approach seemed very interesting for our particular study, we have not been able to successfully implement it, and for that reason, we cannot draw any meaningful conclusions.

8 Conclusions

Regarding the experiments with the **VGG model** approach, we succeeded in improving the given VGG model of 64.9% for COVID-19 audio classification by achieving an **AUC of 71.2%**. This best model uses VGG11 architecture with specified optimized parameters [pooling, window, window size, window stride, num. fft, num. mels] = [avg, Blackman, 0.04, 0.015 4096, 40].

With respect to the improvement of the baseline **HuBERT model** we tried experimenting on hyperparameter tuning, modifying the pooling layer, simplifying the final project layers, and training with other pre-trained models. Nevertheless, we found out that the best approach is unfreezing the last two layers of the 'base' HuBERT model using the baseline hyperparameters. Our goal was to improve the given baseline model which achieved an AUC of 70.0%, we can safely state that we succeeded in doing so by attaining an **AUC of 73.2%**. We also tested using a weighted average of all hidden states for HuBERT but found it underperformed, achieving a best AUC of 70.2% at epoch 4. In addition, we tried gradient accumulation to enhance training efficiency with limited memory. Despite implementing this technique with the best hyperparameter combination, our experiments revealed no performance improvement.

For further analysis, we also tried experimenting on the Speech Embeddings, Wav2Vec and PASE models were tested. Wav2Vec, while comparable, had longer training times and achieved a lower AUC of 70.2%. PASE, with slight hyperparameter tuning, reached an AUC of 72.6%, slightly improving over its baseline but not outperforming HuBERT. Despite an attempt to implement VGGISH, input incompatibility prevented successful testing. Ultimately, PASE emerged as the best alternative with an AUC of 72.6%, but the previous HuBERT remained the top performer in these experiments.

In our study, K-fold cross-validation aimed to mitigate overfitting with limited data by using each data point for training and validation across multiple folds. However, memory limitations caused Out Of Memory errors, even with gradient accumulation, restricting us to only one fold and preventing conclusive results from this approach.