# 1 Experiments and results

In this section we are going to present the experiments and results obtain with the developed turn-taking module both in terms of performance of the system and accuracy. In particular this section is divided into four subsection that are: "Model's inference latency" where we do an analysis of the latency introduced by the model inference; "End of turn detection latency" where we talk about how we measure the latency of the system in detecting the end of turn; "System throughput" where we talk about the throughput of the system and "System accuracy" where we make a comprehensive analysis of the system accuracy in detecting the end of turn in different scenarios, using different datasets and audio manipulation to add background noises.

The experiments are performed in local. The server runs on a MacBook Air 2018 with a CPU 8 core, Neural engine 16 core and a 8 GB unified memory UMA (Unified Memory Architecture).

## 1.1 Model's inference latency

In this section we are going to study the latency of the model's inference. This is the execution time of the processing of the audio chunk from the PyTorch model, that return the probability of the presence of voice in the audio. This latency affects the end of turn detection latency described in the next section, so it is important that it is the lowest as possible.

This latency has been computed using the *time.time()* method in the *analyze()* function of the **Analyzer** class. The computed latency has been inserted in a list and the system every 200 measurements prints on screen the updated average of the latency of the inference. The listing 1.1 shows the code used to compute the latency.

```
1 tensor = torch.from_numpy(self.int2float(frame.to_ndarray()))
2 start_inference = time.time()
```

```
new_confidence = self.vad_model(tensor, self.sampling_rate).item()
end_inference = time.time()
inference_latency = end_inference - start_inference
self.inference_latencies.append(inference_latency)

# every 200 inferences the average inference latency is updated
if(len(self.inference_latencies) % 200 == 0):
    average_inference_latency = sum(self.inference_latencies) / len(self.inference_latencies)
    print(f'Average inference latency is {average_inference_latency} computed over {len(self.inference_latencies)} measurements')
```

Listing 1: Model's inference latency

The experiments are done using different sampling rate and frames duration. The results obtained are shown in the table 1. It shows the average inference time computed over 3000 measurements.

| Duration (ms) | 8 kHz | 16kHz |
|---|---|---|
| 40 | 5.89 | 5.84 |
| 50 | 6.00 | 6.29 |
| 60 | 8.9 | 6.30 |
| 80 | 9.2 | 6.78 |
| 100 | 7.82 | 6.87 |

Table 1: Average inference time in ms

The results obtained shows an inference time of about 6 milliseconds with very small differences depending on the frame duration and sampling rate. This processing time is negligible considering the use case of our application considering the overall latency and waiting time given by the frame transmission over internet and thresholds to wait. It seems that the different inference configurations doesn't impact the inference times, this let us to choose the

best configuration when we will do the accuracy analysis without impacting the performances of the system.

## 1.2 End of turn detection latency

In this section we are going to talk about how we measured the performance in terms of the latency of the system detecting the end of turn of the user. The idea is to measure how much time passes from when the user stops talking to the actual sending of the message *Turn change confirmed*. This latency, together with the processing time to generate a response from the user sentence, constitutes the main parts responsible for the reactivity of the system. So the idea is to have the lower latency possible in order to minimize the response time of the system, to make the conversation as realistic as possible.

The best way to do it would be an automatic system that detects when you stops talking, takes the relative timestamps and when the system detects the turn change, measures the end timestamp to compute the latency. The problem is that to do it we would need a vad to measure another vad. This is not a good idea and another vad would introduce useless latency to the result.

Instead we choose to measure this latency manually. In particular the idea is that, during the experiments, the user press the space bar on the Web client when he stops talking and the system automatically computes the end of turn detection latency printing it on screen. Obviously being this a manual method it is not precise in measuring the actual latency and gives an overall idea on what the responsiveness of the system is.

In particular this is obtained getting the timestamp on the JavaScript client of when the space bar is pressed. Then this timestamp is passed to the Server thought a WebRTC Data channel. This received timestamp is then saved in the Analyzer object and will be compared with the timestamp obtained in

the moment of the actual sending of the *Turn changed confirmed* message, that is when the actual turn change happens in the *set_state()* method. The listing 1.2 shows the simple JavaScript code on the client while the listing 1.2 shown how the message on the data channel in handled on the server.

```
// Add an event listener to the document for the keydown
    event
document.addEventListener('keydown', function(event) {
    // Check if the pressed key is the space bar (key " ")
    if (event.key === " ") {
        end_of_turn = Date.now();
        //send the time over the data streaming
        dc.send(end_of_turn);
    }
});
```

Listing 2: Client implementation of the method to take the timestamp

```
@pc.on("datachannel")
    def on_datachannel(channel):
        @channel.on("message")
        def on_message(message):
            analyzer.end_of_turn_timestamp = int(message)
```

Listing 3: Server handling of the client timestamp

Even if we know that this method is not very precise, in order to give an indicative overview of the latency introduced by the system in this work we tried to make an experiment to in some way provide an indicative value for the latency. The experiment consisted in taking 30 measurement and considering the average value as the latency of the system. In particular the obtained value was of 796 ms with a *confirmed_silence_threshold* of 700 ms. This means that the system would have an ideal latency of 700 ms because it has to wait at least the *confirmed_silence_threshold* to confirm the end of turn detection, while our system introduces 96 ms of difference from this value. In the end considering the order of magnitude of the ideal latency the

system provide a good result in terms of the additional latency introduced since it adds a latency that is the 14 % of the ideal value. It is important to consider that the experiment is performed locally and brings and additional latency that is coherent with the type of operations done by the system, as for example the buffering created by WebRTC.

## 1.3 System throughput

In this section we are going to talk about the throughput of the system, that is how many audio chunks the system is able to process in a given period of time. The obvious idea is that the system should process the frames at an higher rate with respect to the rate at which they arrive in order to avoid to create a buffer of incoming frames to process.

In particular we measure the time that passes from one audio frame processing to the other, that is the rate at which the system updates the state of the communication. This is done simply taking the timestamp at which the last state is updated and comparing it with the last timestamp taken. The average of this value is then updated every 100 processing. The implementation is found in the listing 1.3

```
new_end_analyze = time.time()
gap = new_end_analyze - self.old_end_analyze
if self.old_end_analyze != 0:
    self.analyze_gaps.append(gap)
    print(f'Analyze gap: {gap}')
self.old_end_analyze = new_end_analyze
if(len(self.analyze_gaps) > 0 and len(self.analyze_gaps) %
   100 == 0):
    average_analyze_gap = sum(self.analyze_gaps) / len(self.
   analyze_gaps)
    print(f'Average analyze gap: {average_analyze_gap}')
```

Listing 4: System throughput analysis implementation

The results obtained about the single "analyze gaps", that is the time gap between two processing, depends on the value of the frame duration specified in the configuration file while the average analyze gap is very near the ideal value of the frame duration parameter. In particular if for example the frame duration parameter is 40 ms the single analyze gaps and the average analyze gap is near that value while if the frame duration parameter is 50 ms the single analyze gaps are between 40 ms and 60 ms with the average value that is very near the 50ms ideal value. This happen because the audio frames coming from WebRTC has a standard duration of 20 ms and in order to process them we collect them in a FIFO buffer until we have all the samples for the desired frame duration. This means that in order to process the first audio frame we have to wait 3 packets of 20 ms each waiting in total 60 ms. Once we process the 50 ms audio frame we have to wait "only" other 2 packets this time for a total of 40 ms because we have already stored 10 ms of audio samples in the buffer. The next time we have to wait again 60 ms and so on. This happen only when the frame duration in not a multiple of 20 ms. When that is the case we have only to wait the correct number of packets because we are going to use them all. For example with a frame duration of 40 ms every time we need to wait 2 packets of 20 ms each that arrive from WebRTC and we can continue with the processing immediately.

Regarding the average analyze gap instead the value is really near the ideal value of the frame duration parameter. For example in an experiment with the frame duration at 50 ms the average analyze gap value computed over 300 measurements is of 49.78 ms. In a second experiment with the frame duration value of 40 ms the average analyze gap computed over 300 measurements was of 39.73 ms.