sonosco

Release 1.0

Sep 30, 2019

Installation and Getting Started

Sonosco (from Lat. sonus - sound and nōscō - I know, recognize) is a library for training and deploying deep speech recognition models.

The goal of this project is to enable fast, repeatable and structured training of deep automatic speech recognition (ASR) models as well as providing a transcription server (REST API & frontend) to try out the trained models for transcription. Additionally, we provide interfaces to ROS in order to use it with the anthropomimetic robot Roboy.

The following diagram shows a brief overview of the functionalities of sonosco:

The project is split into 4 parts that correlate with each other:

For data(-processing) scripts are provided to download and preprocess some publicly available datasets for speech recognition. Additionally, we provide scripts and functions to create manifest files (i.e. catalog files) for your own data and merge existing manifest files into one.

This data or rather the manifest files can then be used to easily train and evaluate an ASR model. We provide some ASR model architectures, such as LAS, TDS and DeepSpeech2 but also individual pytorch models can be designed to be trained.

The trained model can then be used in a transcription server, that consists of a REST API as well as a simple Vue.js frontend to transcribe voice recorded by a microphone and compare the transcription results to other models (that can be downloaded in our Github repository).

Further we provide example code, how to use different ASR models with ROS and especially the Roboy ROS interfaces (i.e. topics & messages).

CHAPTER 1

Installation

1.1 Via pip

The easiest way to use Sonosco's functionality is via pip:

```
pip install sonosco
```

Note: Sonosco requires Python 3.6 or higher.

For reliability, we recommend using an environment virtualization tool, like virtualenv or conda.

1.2 For developers

Clone the repository and install dependencies:

```
# Clone the repo and cd inside it
git clone https://github.com/Roboy/sonosco.git && cd sonosco

# Create a virtual python environment to not pollute the global setup
python -m venv venv

# Activate the virtual environment
source venv/bin/activate

# Install normal requirements
pip install -r requirements.txt

# Link your local sonosco clone into your virtual environment
pip install -e .
```

Now you can check out some of the *Getting Started* tutorials, to train a model or use the transcription server.

CHAPTER 2

Use the Transcription Server

2.1 Dockerized inference server

Get the hold of our new fully trained models from the latest release! Try out the LAS model for the best performance. Then specify the folder with the model to the runner script as shown underneath.

You can get the docker image from dockerhub under `yuriyarabskyy/sonosco-inference:1.0`. Just run `cd server && ./run.sh yuriyarabskyy/sonosco-inference:1.0` to pull and start the server or optionally build your own image by executing the following commands.

```
cd server

# Build the docker image
./build.sh

# Run the built image
./run.sh sonosco_server
```

You can also specify the path to your own models by writing `./run.sh <image_name> <path/to/models>`.

Open `http://localhost:5000` in Chrome. You should be able to add models for performing transcription by clicking on the plus button. Once the models are added, record your own voice by clicking on the record button. You can replay and transcribe with the corresponding buttons.

You can get pretrained models from the release tab in this repository.

To learn more see *Transcription Server*

CHAPTER 3

Train your first model

In the following we will give you a short tutorial how to train the Listen Attend Spell model.

First of all, it is required to have some data. For that simply download one of the publicly available datasets using one of our scripts, see the Data Section *download* .

In the next step, create a `config.yaml` file, that contains the following:

```
experiment:
  experiment_path: "path/to_directory/where/experiment/is_stored"
  experiment_name: 'getting_started_w_las'
  gloabl_seed: 1234

data:
  train_manifest: "path/to/train_manifest.csv"
  val_manifest: "path/to/val_manifest.csv"
  test_manifest: "path/to/test_manifest.csv"
  batch_size: 4
  num_data_workers: 8

training:
  max_epochs: 10
  learning_rate: 1.0e-3
  labels: "ABCDEFGHIJKLMNOPQRSTUVWXYZ' "
  test_step: 1
  checkpoint_path: 'path/to/checkpoint_dir_in_experiment'
  window_size: 0.02
  window_stride: 0.01
  window: 'hamming'
  sample_rate: 16000

model:
  encoder:
    input_size: 161
    hidden_size: 256
    num_layers: 1
```

(continues on next page)

(continued from previous page)

```
    dropout: 0.2
    bidirectional: True
    rnn_type: "lstm"

  decoder:
    embedding_dim: 128
    hidden_size: 512
    num_layers: 1
    bidirectional_encoder: True
    vocab_size: 28
    sos_id: 27
    eos_id: 26
```

Under 'experiment_path' all your code, logs and checkpoints will be saved during training. Further a global seed per experiment can be set. Under 'data' the manifest files for the data need to be specified, together with batch size and the amount of workers for the dataloader. 'training' contains the maximum epochs to be trained, the initial learning rate, the labels, i.e. characters the model is trained on and test_step is the step in an epoch, where validation is performed.

'model' contains parameters for the model to be trained, in this case, the LAS model requires parameters for its encoder and decoder.

Let's setup the logger:

```
import logging
reload(logging)
logging.basicConfig(level=logging.INFO)
```

In order to start training, we first need to import all required functions:

```
import torch
from sonosco.common.path_utils import parse_yaml
from sonosco.datasets import create_data_loaders

from sonosco.training import Experiment, ModelTrainer
from sonosco.serialization import Deserializer
from sonosco.decoders import GreedyDecoder
from sonosco.models import Seq2Seq

from sonosco.training.metrics import word_error_rate
from sonosco.training.metrics import character_error_rate
from sonosco.training.losses import cross_entropy_loss
```

Let's load the created .yaml file and create the dataloaders:

```
config = parse_yaml('path/to/config.yaml')
device = torch.device("cpu")
train_loader, val_loader, test_loader = create_data_loaders(**config['data'])
```

For the model trainer, we can create a dict, that is then just passed for initialization:

```
training_args = {
    'loss': cross_entropy_loss,
    'epochs': config['training']["max_epochs"],
    'train_data_loader': train_loader,
    'val_data_loader': val_loader,
    'test_data_loader': test_loader,
    'lr': config['training']["learning_rate"],
```

(continues on next page)

(continued from previous page)

```
    'custom_model_eval': True,
    'metrics': [word_error_rate, character_error_rate],
    'decoder': GreedyDecoder(config['training']['labels']),
    'device': device,
    'test_step': config['training']["test_step"]}
```

With the following code, you can now easily start training and continue it:

```
experiment = Experiment.create(config, logging.getLogger())

CONTINUE = False

if not CONTINUE:
    model = Seq2Seq(config['model']["encoder"], config['model']["decoder"])
    trainer = ModelTrainer(model, **training_args)
else:
    loader = Deserializer()
    trainer, config = loader.deserialize(ModelTrainer, config['training']["checkpoint_
↪path"], {
            'train_data_loader': train_loader,'val_data_loader': val_loader, 'test_
↪data_loader': test_loader,
    }, with_config=True)

experiment.setup_model_trainer(trainer, checkpoints=True, tensorboard=True)

try:
    if not CONTINUE:
        experiment.start()
    else:
        experiment.__trainer.continue_training()
except KeyboardInterrupt:
    experiment.stop()
```

We now go through this snippet in detail: First, we set up the experiment and set the bool `CONTINUTE=FALSE` so that we start the training. We setup the modeltrainer with the las model and all the parameters we specified in the `training_args` dictionary.

Now we register the modeltrainer to the experiment and start it.

The try-except block catches keyboard interuptions, where the experiment will then save the model checkpoint aswell as the model trainer. This serialized model trainer can then be used to continue training, just by setting the `CONTINUE=True` and rerunning the script. What happens now is, that the modeltrainer, that is saved at the path, specified in the config file, is deserialized and continues training.

That's it ! You successfully train an LAS model.

For a more detailed description of each component, have a look at general description *training* of the model training process and its components.

CHAPTER 4

Downloading publicly available datasets

We provide scripts to download and process the following publicly available datasets:

- An4 - Alphanumeric database
- Librispeech - reading english books
- TED-LIUM 3 (ted3) - TED talks
- Voxforge
- common voice (old version)

Simply run the respective scripts in `sonosco > datasets > download_datasets` with the output_path flag and it will download and process the dataset. Further, it will create a manifest file for the dataset.

For example

```
python an4.py --target-dir temp/data/an4
```

CHAPTER 5

Creating a manifest from your own data

If you want to create a manifest from your own data, order your files as follows:

```
data_directory
└──txt
|   |   transcription01.txt
|   |   transcription02.txt
|
└──wav
    |   audio01.wav
    |   audio02.wav
```

To create a manifest, run the `create_manifest.py` script with the data directory and an outputfile to automatically create a manifest file for your data.

For example:

```
python create_manifest.py --data_path path/to/data_directory --output-file temp/data/
↪manifest.csv
```

CHAPTER 6

Merging manifest files

In order to merge multiple manifests into one, just specify a folder that contains all manifest files to be merged and run the `merge_manifest.py` . This will look for all .csv files and merge the content together in the specified output-file.

For example:

```
python merge_manifest.py --merge-dir path/to/manifests_dir --output-path temp/
↪manifests/merged_manifest.csv
```

CHAPTER 7

Model Training

One goal of this framework is to keep training as easy as possible and enable keeping track of already conducted experiments.

7.1 Analysis Object Model

For model training, there are multiple objects that interact with each other.

For Model training, one can define different metrics, that get evaluated during the training process. These metrics get evaluated at specified steps during an epoch and during validation. Sonosco provides different metrics already, such as Word Error Rate or Character Error Rate. But additional metrics can be created in a similar scheme. See *Metrics* .

Additionally, callbacks can be defined. A Callback is an arbitrary code that can be executed during training. Sonosco provides for example different Callbacks, such as Learning Rate Reduction, ModelSerialisationCallback, TensorboardCallback, . . . Custom Callbacks can be defined following the examples. See *Callbacks* .

Most importantly, a model needs to be defined. The model is basically any torch module. For (de-) serialisation, this model needs to conform to the *Serialisation Guide* . Sonosco provides already existing model architectures that can be simply imported, such as Listen Attend Spell, Sequence to Sequence with Time-depth Separable Convolutions and DeepSpeech2. See *Models* .

We created a specific AudioDataset Class that is based on the pytorch Dataset class. This AudioDataset requires an AudioDataProcessor in order to process the specified manifest file. Further we created a special AudioDataLoader based on pytorch's Dataloader class, that takes the AudioDataset and provides the data in batches to the model training.

Metrics, Callbacks, the Model and the AudioDataLoader are then provided to the ModelTrainer. This ModelTrainer takes care of the training process. See *Train your first model* .

The ModelTrainer can then be registered to the Experiment, that takes care of provenance. I.e. when starting the training, all your code is time_stamped and saved in a separate directory, so you can always repeat the same experiment. Additionally, the serialized model and modeltrainer, logs and tensorboard logs are saved in this folder.

Further, a Serializer needs to be provided to the Experiment. This object can serialize any arbitrary class with its parameters, that can then be deserialized using the Deserializer. When the `Experiment.stop()` method is called,

the model and the ModelTrainer get serialized, so that you can simply continue the training, with all current parameters (such as epoch steps,. . .) when deserializing the ModelTrainer and continuing training.

CHAPTER 8

Metrics

You can define metrics that get evaluated every epoch step. An example metric is the word error:

```
def word_error_rate(model_out: torch.Tensor, batch: Tuple, context=None) -> float:
    inputs, targets, input_percentages, target_sizes = batch

    # unflatten targets
    split_targets = []
    offset = 0
    for size in target_sizes:
        split_targets.append(targets[offset:offset + size])
        offset += size

    out, output_sizes = model_out

    decoded_output, _ = context.decoder.decode(out, output_sizes)
    target_strings = context.decoder.convert_to_strings(split_targets)
    wer = 0
    for x in range(len(target_strings)):
        transcript, reference = decoded_output[x][0], target_strings[x][0]
        try:
            wer += decoder.wer(transcript, reference) / float(len(reference.split()))
        except ZeroDivisionError:
            pass
    del out

    wer *= 100.0 / len(target_strings)
    return wer
```

The metric is some arbitrary function that gets the model output, the batch and the context, which is the modeltrainer, so that within the metric you can access all parameters of the modeltrainer. The metric returns then the metric to the model trainer, that prints it out every epoch step and can be used from within the [Callbacks](#callbacks).

Sonosco already provides predefined metrics, such as Character Error Rate (CER) and Word Error Rate (WER).

CHAPTER 9

Callbacks

A callback is an arbitrary code that can be executed during training. When defining a new callback all you need to do is inherit from the AbstractCallback class:

```
@serializable
class AbstractCallback(ABC):
    """
    Interface that defines how callbacks must be specified.
    """

    @abstractmethod
    def __call__(self, epoch: int, step: int, performance_measures: dict, context:␣
↪ModelTrainer) -> None:
        """
        Called after every batch by the ModelTrainer.

        Args:
            epoch (int): current epoch number
            step (int): current batch number
            performance_measures (dict): losses and metrics based on a running average
            context (ModelTrainer): reference to the calling ModelTrainer, allows to␣
↪access members

        """
        pass

    def close(self) -> None:
        """
        Handle cleanup work if necessary. Will be called at the end of the last epoch.
        """
        pass
```

Each callback is called at every epoch step and receives the current epoch, the current step, all performance measures , i.e. a list of all metrics and the context, which is the modeltrainer itself, so you can access all parameters of the modeltrainer from within the callback.

Sonosco already provides a lot of callbacks:

- **Early Stopping: Early Stopping to terminate training early if the monitored metric did not improve** over a number of epochs.
- Gradient Collector: Collects the layer-wise gradient norms for each epoch.
- History Recorder: Records all losses and metrics during training.
- Stepwise Learning Rate Reduction: Reduces the learning rate of the optimizer every N epochs.
- Scheduled Learning Rate Reduction: Reduces the learning rate of the optimizer for every scheduled epoch.
- Model Checkpoint: Saves the model and optimizer state at the point with lowest validation error throughout training.

Further, sonosco provides a couple of callbacks that store information for tensorboard: * TensorBoard Callback: Log all metrics in tensorboard. * Tb Text Comparison Callback: Perform inference on a tds model and compare the generated text with groundtruth and add it to tensorboard. * Tb Teacher Forcing Text Comparison Callback: Perform decoding using teacher forcing and compare predictions to groundtruth and visualize in tensorboard. * Las Text Comparison Callback: Perform inference on an las model and compare the generated text with groundtruth and add it to tensorboard.

CHAPTER 10

Models

Sonosco provides some predefined deep speech recognition models, that can be used for training:

10.1 Deep Speech 2

We took the pytorch implementation of the Deep Speech 2 model from Sean Naren and ported it to the sonosco serialization guidelines.

10.2 Listen Attend Spell (LAS)

We took the pytorch implementation of the Listen Attend Spell model from AzizCode92 and ported it to the sonosco serialization guidelines. This model can be imported using: `from sonosco.models import Seq2Seq`

10.3 Sequence-to-Sequence Model with Time-Depth Separable Convolutions

We implemented a sequence-to-sequence model with Time-Depth separable convolutions in pytorch, following a paper from Facebook AI .

These models can be simply imported and used for training. (See *Train your first model*)

CHAPTER 11

Serialisation Guide

11.1 Serialization

Sonosco provides it's own serialization method. It allows to save complete state of your training. Including parameters and meta-parameters.

In order to enable the serialization follow those steps:

1. add the `@sonosco.serialization.serializable` decorator all the classes you wish to serialize.
2. Instead of using `__init__()` method put all the fields of your object on the class level (Type annotation are mandatory!) (Currently only primitives, lists of primitives, types, callables and other serializable objects are supported). If you wish to perform some custom initialization use `__post_init__(self)` method (`__init__(self)` is auto generated!)

```
@serializable
class Example:
    arg: int = 0

    def __post_init__(self):
        pass
```

3. Sonosco will generate `__serialize__` method for you, however this should not be used directly. Instead use `sonosco.serialization.Serializer.serialize` method:

```
Serializer().serialize(Example(), "/path/to/save/object")
```

11.2 Deserialization

In order to deserialize class use: `sonosco.serialization.Deserializer.deserialize` method.

```
ex = Deserializer.deserialize(Example, "/path/to/save/object")
```

Only object serialized with sonosco `@serializable` can be deserialized!

That's it. To see a full example, check out this serialization example or check one of the models in the sonosco repository.

Check also inline docs for more details about different features of (de)serialization.

CHAPTER 12

Model Evaluation

We provide model evaluation using the bootstrapping method. So the samples that the model is evaluated on are randomly sampled from the testset with replacement.

You can pass the *Metrics* that were also used for model training and evaluate your model on that. You only need to specify:

- model - i.e. a torch nn.module
- dataloader - the test dataloader
- bootstrap size - number of samples in contained in one bootstrap
- num bootstraps - number of bootstraps to compute
- torch device
- metrics - List of metrics

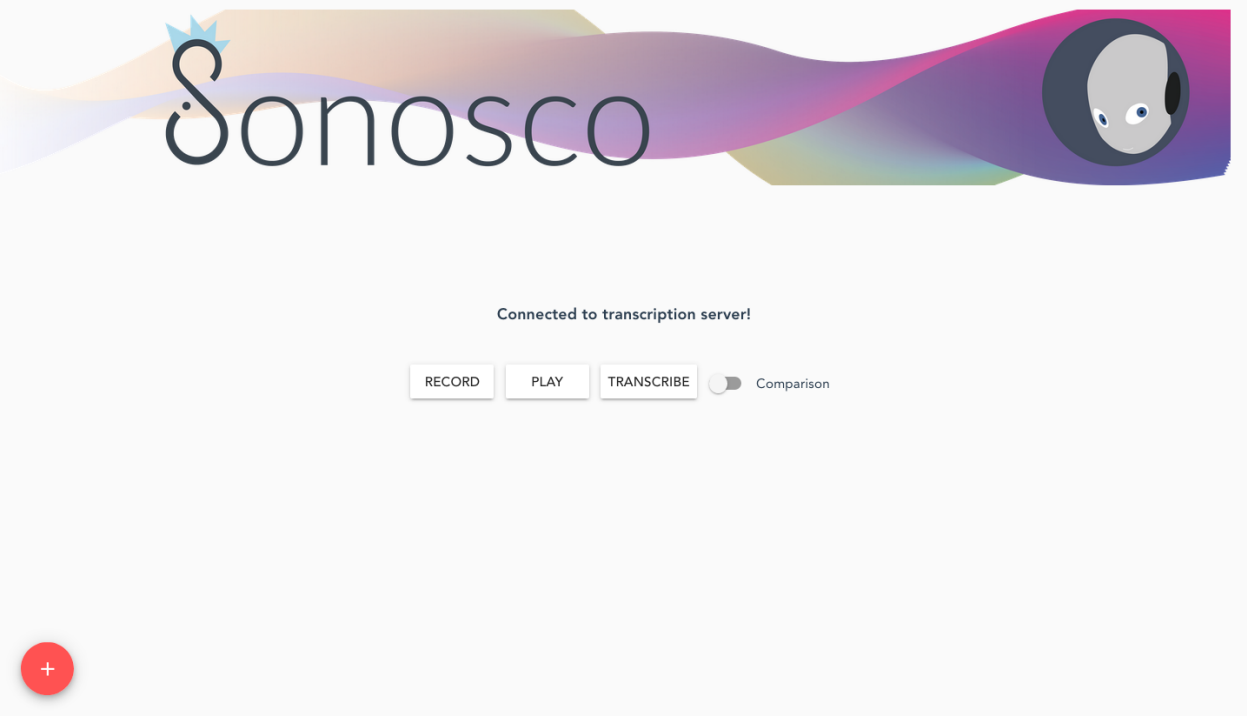
The model evaluator will log its results in tensorboard and store it as a json file in the logs folder of the experiment. With this method, one can evaluate the mean and variance per metric of the model.

To use the model trainer, just do: `from sonosco.training import ModelEvaluator`

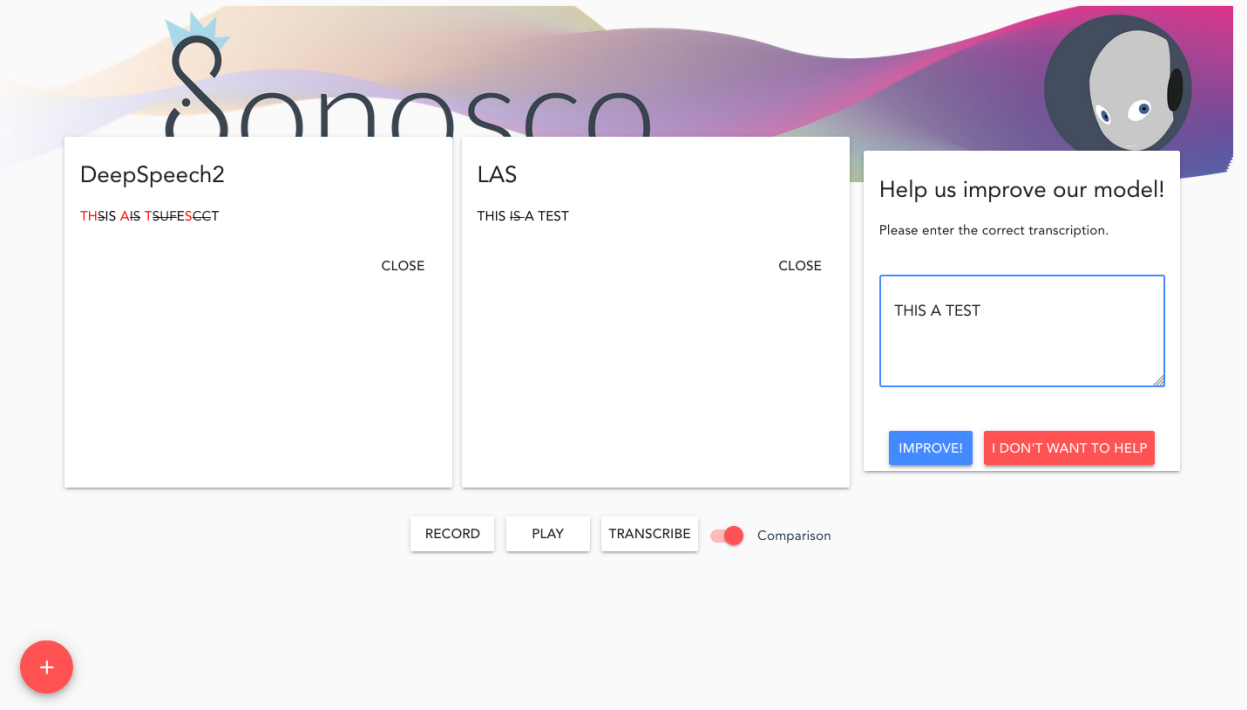
CHAPTER 13

Transcription Server

In the picture below, you see the GUI we created for our transcription server. To start this server, follow the [Quick Start](#start).



By pressing on the red plus button, one can add different models, that can be specified in a respective config file.
 Then you can record your voice, by clicking on the 'RECORD' button, listen to it by pressing the 'PLAY' button and finally transcribing it with 'TRANSCRIBE'.



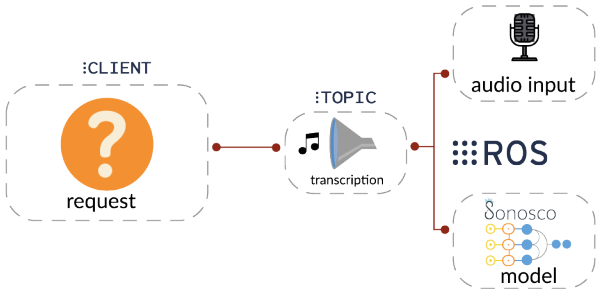
The transcription of both models is displayed in the respective block. Further, a popup shows up, where you are asked to correct the transcription. When you click on 'IMPROVE', the audio and the respective transcription are saves to `~/.sonosco/audio_data/web_collected/` . If one uses the 'Comparison' toggle, the transcriptions are additionally compared to the corrected transcription.

13.1 How to use your own model

In order to use your own model with the model trainer, additionally to the serialization guide, you need to implement an inference snippet. For this, simply follow the example of the other models. And then specify your model in the `../sonosco/server/model_loader.py` script. (Have a look at the repo)

CHAPTER 14

ROS



Sonosco provides convenient access to ROS1 server functionality. To run ROS on your machine:

1. Make sure that `roscore` is running
2. Run your server. Now all subscribers specified in config will be listening for incoming requests.

```
with SonoscoROS1(CONFIG) as server:
    server.run()
```

Example of config:

```
CONFIG = {
    'node_name': 'roboy_speech_recognition',
    'workers': 5,
    'subscribers': [
        {
            'name': 'recognition',
            'topic': '/roboy/cognition/speech/recognition',
            'service': RecognizeSpeech,
            'callback': custom_callback,

        },
        {
            'name': 'recognition_german',
```

(continues on next page)

(continued from previous page)

```
            'topic': '/roboy/cognition/speech/recognition/german',
            'service': RecognizeSpeech,
        }
    ],
    'publishers': [
        {
            'name': 'ledmode',
            'topic': '/roboy/control/matrix/leds/mode',
            'message': ControlLeds,
            'kwargs': {
                'queue_size': 3
            }

        },
        {
            'name': 'ledoff',
            'topic': '/roboy/control/matrix/leds/off',
            'message': ControlLeds,
            'kwargs': {
                'queue_size': 10
            }
        },
        {
            'name': 'ledfreez',
            'topic': '/roboy/control/matrix/leds/freeze',
            'message': ControlLeds,
            'kwargs': {
                'queue_size': 1
            }
        }
    ],



}
```

To `SonoscoROS1` you can also pass two additional parameters, namely:

```
default_asr_interface: SonoscoASR
default_audio_interface: SonoscoAudioInput
```

Those will be registered in default callback handling incoming requests:

```
audio = self.default_audio_interface.request_audio()
return self.default_asr_interface.infer(audio)
```

You can also use custom callback per subscriber, in this case you have to handle the request yourself (the default ASR and audio interfaces are ignored):

```
def custom_callback(request, publishers):
    msg = ControlLeds()
    msg.mode = 2
    msg.duration = 0
    publishers['ledmode'].publish(msg)
    with MicrophoneClient() as audio_input:
        audio = audio_input.request_audio()
```

(continues on next page)

(continued from previous page)

```
        transcription = asr.infer(audio)
    return transcription
```

List of publishers specified in the config is passed to every callback.

Example of ROS1 server usage can be found here Take a look at STT_server.py and STT_client.py

CHAPTER 15

Acknowledgements

First of all we want to thank the roboy team for providing infrastructure to train some models. We want to thank Christoph Schöller for his PyCandle library, that we took as an inspiration for our model training process. Further, we want to thank Sean Naren for his pytorch implementation of the Deep Speech 2 model. And AzizCode92 for his pytorch implementation of the LAS model.