Laboratories on Audio Deep Learning

with Nvidia Jetson Nano

SmartComputerLab

Table of Contents

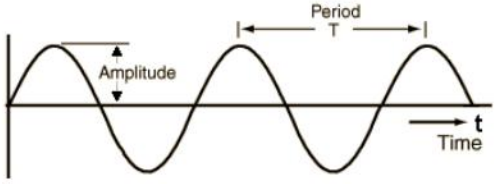
0. Introduction : State-of-the-Art Techniques

Before starting the practical exercices on Nvidia Jetson we need an introduction to the world of disruptive deep learning audio applications and architectures. We all need to know about Spectrograms, in Plain English.
About the Computer Audio and NLP applications and many groundbreaking use cases for deep learning with audio data that are transforming our daily lives.
In this part we are going to explore the fascinating world of audio deep learning. Our goal is to understand not just how something works but why it works that way.

In this first section, since this area may not be as familiar to people, we will introduce the topic and provide an overview of the deep learning landscape for audio applications. We will understand what audio is and how it is represented digitally.

0.1 What is sound?

We all remember from school that a sound signal is produced by variations in air pressure. We can measure the intensity of the pressure variations and plot those measurements over time.
Sound signals often repeat at regular intervals so that each wave has the same shape. The height shows the intensity of the sound and is known as the amplitude.

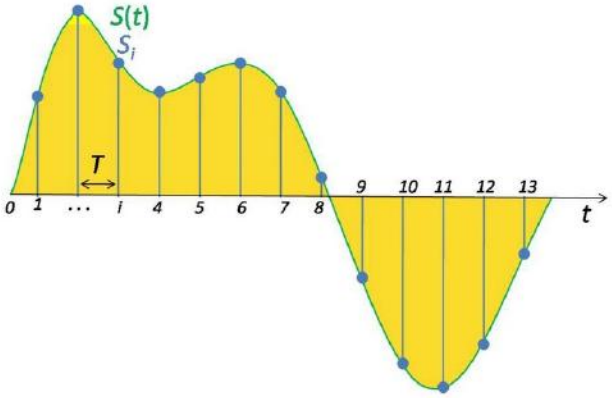


The time taken for the **signal** to complete **one full wave** is the **period**. The **number of waves** made by the signal **in one second** is called the **frequency**. The **frequency is the reciprocal of the period**. The **unit of frequency** is **Hertz**.
The majority of sounds we encounter may not follow such simple and regular periodic patterns. But signals of different frequencies can be added together to create **composite signals** with more complex repeating patterns. All sounds that we hear, including our own human voice, consist of **waveforms** like these.
For instance, this could be the sound of a musical instrument.

The human ear is able to differentiate between different sounds based on the '**quality**' of the sound which is also known as **timbre**.

0.2 How do we represent sound digitally?

To digitize a sound wave we must turn the signal into a series of numbers so that we can input it into our models. This is done by measuring the amplitude of the sound at fixed intervals of time.



Each such measurement is called a **sample**, and the sample rate is the number of samples per second. For instance, a common sampling rate is about 44,100 samples per second. That means that a 10-second music clip would have 441,000 samples!

0.3 Preparing audio data for a deep learning model

Till a few years ago, in the days before Deep Learning, machine learning applications of Computer Vision used to rely on traditional image processing techniques to do feature engineering. For instance, we would generate hand-crafted features using algorithms to detect corners, edges, and faces. With NLP applications as well, we would rely on techniques such as extracting N-grams and computing Term Frequency.

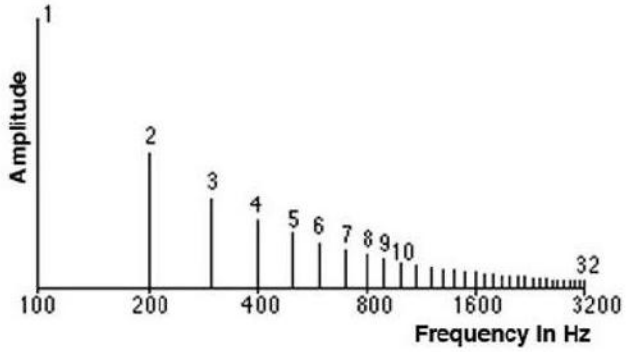
Similarly, audio machine learning applications used to depend on **traditional digital signal processing techniques** to extract features. For instance, to understand human speech, audio signals could be analyzed using **phonetics concepts** to extract elements like **phonemes**. All of this required a lot of **domain-specific expertise** to solve these problems and tune the system for better performance.

However, in recent years, as Deep Learning becomes more and more ubiquitous, it has seen tremendous success in handling audio as well. With deep learning, the **traditional audio processing techniques are no longer needed**, and we can rely on standard data preparation without requiring a lot of manual and custom generation of features.

What is more interesting is that, with deep learning, **we don't actually deal with audio data in its raw form**. Instead, the common approach used is to **convert the audio data into images** and then use a standard CNN architecture to process those images! Really?
Convert sound into pictures? That sounds like science fiction. The answer, of course, is fairly commonplace and mundane. This is done by generating **Spectrograms from the audio**. So first let's learn what a **Spectrum** is, and use that to understand **Spectrograms**.

0.3.1 Spectrum

As we discussed earlier, signals of different frequencies can be added together to create **composite signals**, representing any sound that occurs in the real-world. This means that **any signal consists of many distinct frequencies** and can be expressed as the sum of those frequencies.
The **Spectrum is the set of frequencies** that are combined together to produce a signal. eg. the picture shows the spectrum of a piece of music.



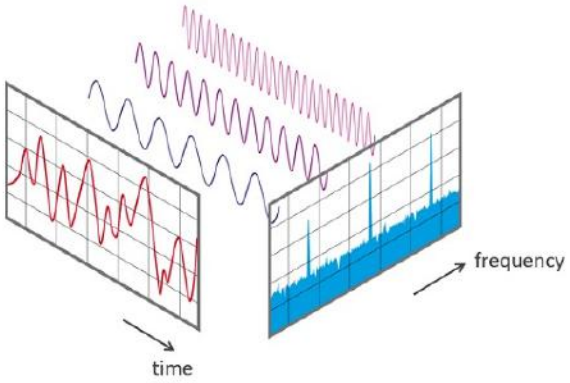
The **Spectrum plots all of the frequencies** that are present in the signal along with the strength or amplitude of each frequency. The **lowest frequency** in a signal called the **fundamental frequency**. Frequencies that are whole number **multiples of the fundamental frequency** are known as **harmonics**.

For instance, if the fundamental frequency is 200 Hz, then its harmonic frequencies are 400 Hz, 600 Hz, and so on.

0.3.2 Time Domain vs Frequency Domain

The **waveforms** that we saw earlier showing **Amplitude** against **Time** are one way to represent a sound signal. Since the **x-axis** shows the range of **time values** of the signal, we are viewing the signal in the **Time Domain**.

The Spectrum is an alternate way to represent the same signal. It shows **Amplitude** against **Frequency**, and since the **x-axis** shows the **range of frequency values** of the signal, ***at a moment in time***, we are viewing the signal in the **Frequency Domain**.

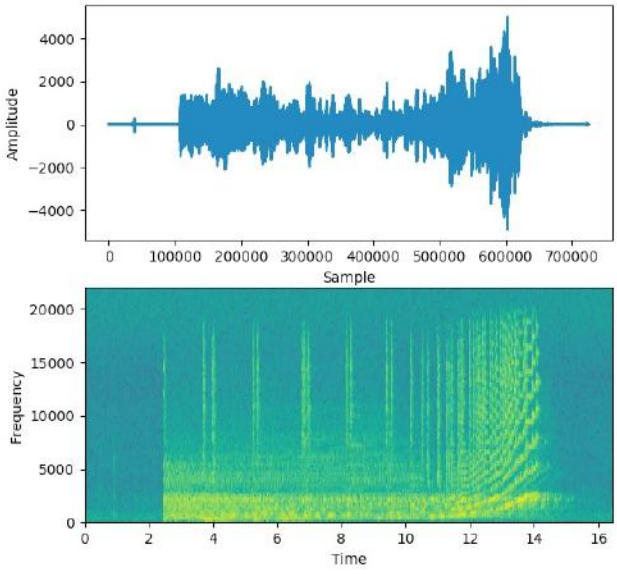


0.3.3 Spectrograms

Since a signal produces different sounds as it varies over time, its constituent **frequencies also vary with time**. In other words, its **Spectrum varies with time**.

A **Spectrogram** of a signal plots its Spectrum over time and is like a '**photograph**' of the signal. It plots **Time** on the **x-axis** and Frequency on the y-axis. It is as though we took the Spectrum again and again at different instances in time, and then joined them all together into a single plot.

It uses **different colors** to indicate the **Amplitude** or strength **of each frequency**. The brighter the color the higher the energy of the signal. Each vertical '**slice**' of the Spectrogram is essentially the **Spectrum** of the signal **at that instant in time** and shows how the signal strength is distributed in every frequency found in the signal at that instant.
In the example below, the **first picture displays the signal in the Time domain** ie. **Amplitude vs Time**. It gives us a sense of **how loud** or quiet a clip is at any point in time, but it gives us **very little information** about which **frequencies** are present.



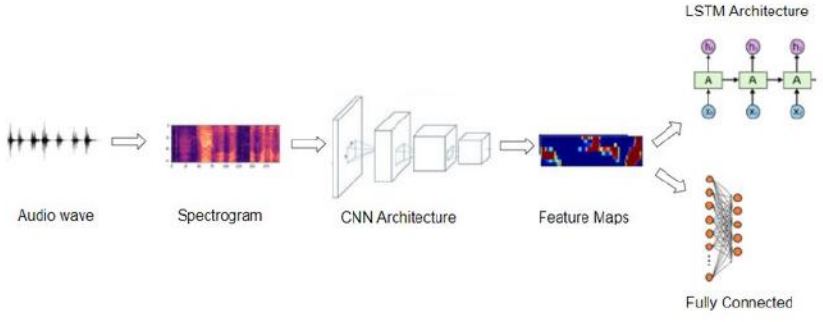
The second picture is the **Spectrogram** and displays the signal 'samples) in the **Frequency domain**.

0.3.4 Generating Spectrograms

To generate spectrograms we need to use **Fourier Transforms**! Fortunately, we don't need to recall all the mathematics, there are very convenient Python library functions that can generate spectrograms for us in a single step. We'll see those in the next section.

0.4 Audio Deep Learning Models

Now that we understand what a Spectrogram is, we realize that it is an equivalent compact representation of an audio signal, somewhat like a 'fingerprint' of the signal. It is an elegant way to capture the essential features of audio data as an image.



So most deep learning audio applications use Spectrograms to represent audio. They usually follow a procedure like this:

- Start with raw audio data in the form of a wave file.
- Convert the audio data into its corresponding spectrogram.
- Optionally, use simple audio processing techniques to augment the spectrogram data. (Some augmentation or cleaning can also be done on the raw audio data before the spectrogram conversion)
- Now that we have image data, we can use **standard CNN architectures** to process them and extract feature maps that are an encoded representation of the spectrogram image.

The next step is to **generate output predictions** from this encoded representation, depending on the problem that you are trying to solve.

- For instance, for an **audio classification problem**, you would pass this through a Classifier usually consisting of some fully connected Linear layers.
- For a **Speech-to-Text problem**, you could pass it through some **RNN** layers to extract **text sentences** from this encoded representation.

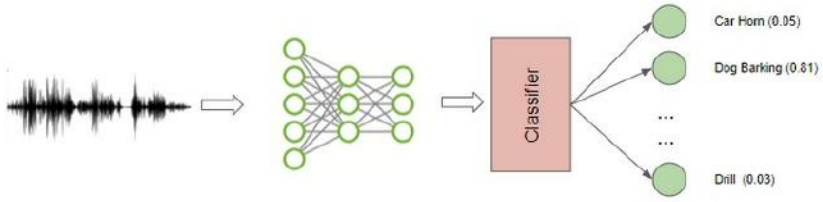
Of course, we are skipping many details, and making some broad generalizations, but in this introduction, we are staying at a fairly high-level. In the following sections, we'll go into a lot more specifics about all of these steps and the architectures that are used.

0.5 What problems does audio deep learning solve?

Audio data in day-to-day life can come in innumerable forms such as human speech, music, animal voices, and other natural sounds as well as man-made sounds from human activity such as cars and machinery.
Given the prevalence of sounds in our lives and the range of sound types, it is not surprising that there are a vast number of usage scenarios that require us to process and analyze audio. Now that deep learning has come of age, it can be applied to solve a number of use cases.

0.5.1 Audio Classification

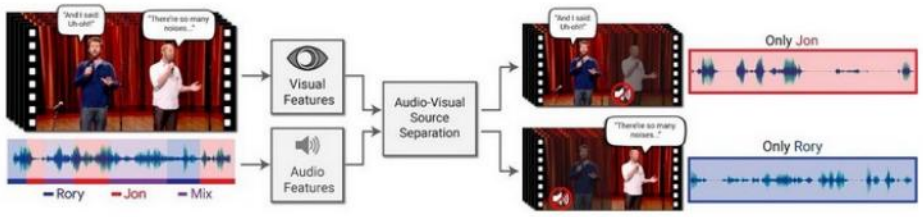
This is one of the most common use cases and involves taking a **sound** and assigning it to **one of several classes**. For instance, the task could be to identify the type or source of the sound. eg. is this a car starting, is this a hammer, a whistle, or a dog barking.



Obviously, the possible applications are vast. This could be applied to detect the failure of machinery or equipment based on the sound that it produces, or in a surveillance system, to detect security break-ins.

0.5.2 Audio Separation and Segmentation

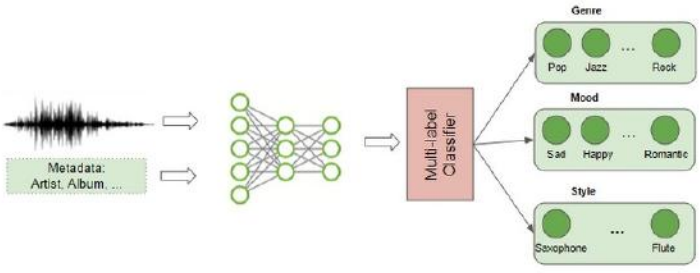
Audio Separation involves isolating a signal of interest from a mixture of signals so that it can then be used for further processing. For instance, you might want to separate out **individual people's voices** from a lot of background noise, or the sound of the violin from the rest of the musical performance.



Audio Segmentation is used to highlight relevant sections from the audio stream. For instance, it could be used for diagnostic purposes to detect the different sounds of the human heart and detect anomalies.

0.5.3 Music Genre Classification and Tagging

With the popularity of music streaming services, another common application that most of us are familiar with is to identify and **categorize music based on the audio**. The content of the music is analyzed to figure out the genre to which it belongs. This is a multi-label classification problem because a given piece of music might fall under more than one genre. eg. rock, pop, jazz, salsa, instrumental as well as other facets such as 'oldies', 'female vocalist', 'happy', 'party music' and so on.

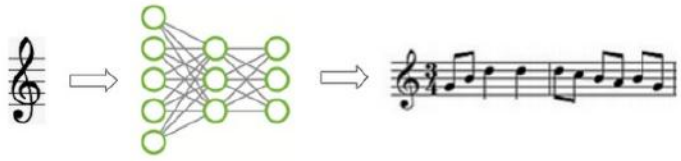


Of course, in addition to the audio itself, there is metadata about the music such as singer, release date, composer, lyrics and so on which would be used to add a rich set of tags to music.

This can be used to index music collections according to their audio features, to provide music recommendations based on a user's preferences, or for searching and retrieving a song that is similar to a song to which you are listening.

0.5.4 Music Generation and Music Transcription

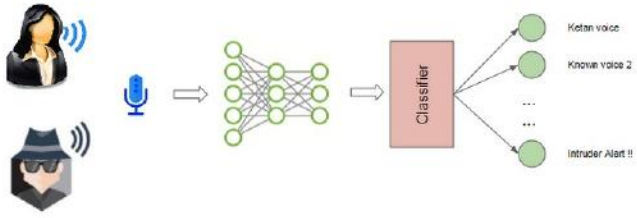
We have seen a lot of news these days about deep learning being used to programmatically generate extremely authentic-looking pictures of faces and other scenes, as well as being able to write grammatically correct and intelligent letters or news articles.



Similarly, we are now able to generate synthetic music that matches a particular genre, instrument, or even a given composer's style.
In a way, Music Transcription applies this capability in reverse. It takes some acoustics and annotates it, to create a music sheet containing the musical notes that are present in the music.

0.5.5 Voice Recognition

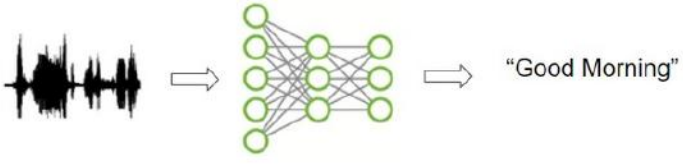
Technically this is also a classification problem but deals with recognizing spoken sounds. It could be used to **identify the gender of a speaker**, or their name (eg. is this Bill Gates or Tom Hanks, or is this Ketan's voice vs an intruder's)



We might want to detect human emotion and identify the mood of the person from the tone of their voice eg. is the person happy, sad, angry, or stressed.
We could apply this to animal voices to identify the type of animal that is producing a sound, or potentially to identify whether it is a gentle affectionate purring sound, a threatening bark, or a frightened yelp.

0.5.6 Speech to Text and Text to Speech

When dealing with human speech, we can go a step further, and not **just recognize the speaker**, but **understand** what they are saying. This involves extracting the words from the audio, in the language in which it is spoken and transcribing it into text sentences.
This is **one of the most challenging applications** because it deals not just with analyzing audio, but also with **NLP** and requires developing some basic language capability to **decipher distinct words** from the uttered sounds.



Conversely, with **Speech Synthesis**, one could go in the other direction and take written text and generate speech from it, using, for instance, an artificial voice for conversational agents.
Being able to understand human speech obviously enables a huge number of useful applications both in our business and personal lives, and we are only just beginning to scratch the surface.
The most well-known examples that have achieved widespread use are virtual assistants like Alexa, Siri, Cortana, and Google Home, which are consumer-friendly products built around this capability.

Conclusion

In this introduction, we have stayed at a fairly high-level and explored the breadth of audio applications, and covered the general techniques applied to solve those problems.

In the following laboratory, we will go into more of the technical details of pre-processing audio data and generating spectrograms. We will take a look at the hyperparameters that are used to tune performance.

That will then prepare us for delving deeper into a couple of end-to-end examples, starting with the Classification of ordinary sounds and culminating with the much more challenging Automatic Speech Recognition, where we will also cover the fascinating CTC algorithm.

Reading 1

Audio Deep Learning Made Simple: Automatic Speech Recognition (ASR), How it Works

Speech-to-Text algorithm and architecture, including Mel Spectrograms, MFCCs, CTC Loss and Decoder, in Plain English.

Over the last few years, Voice Assistants have become ubiquitous with the popularity of Google Home, Amazon Echo, Siri, Cortana, and others. These are the most well-known examples of Automatic Speech Recognition (ASR). This class of applications starts with a clip of spoken audio in some language and extracts the words that were spoken, as text.
For this reason, they are also known as **Speech-to-Text algorithms**.

Of course, applications like Siri and the others mentioned above, go further. Not only do they extract the text but they also interpret and understand the semantic meaning of what was spoken, so that they can respond with answers, or take actions based on the user's commands.

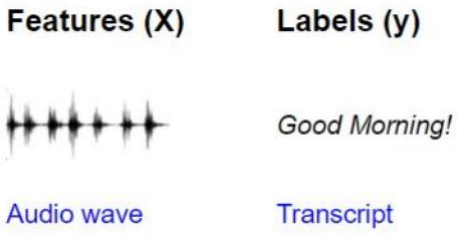
In this reading, I will focus on the core capability of Speech-to-Text using **deep learning**. Our goal throughout will be to understand not just how something works but why it works that way.

1.1 Speech-to-Text

As we can imagine, human speech is fundamental to our daily personal and business lives, and Speech-to-Text functionality has a huge number of applications. One could use it to transcribe the content of customer support or sales calls, for voice-oriented chatbots, or to note down the content of meetings and other discussions.
Basic audio data consists of sounds and noises. Human speech is a special case of that.

Problems like audio classification start with a sound clip and predict which class that sound belongs to, from a given set of classes. For Speech-to-Text problems, your training data consists of:

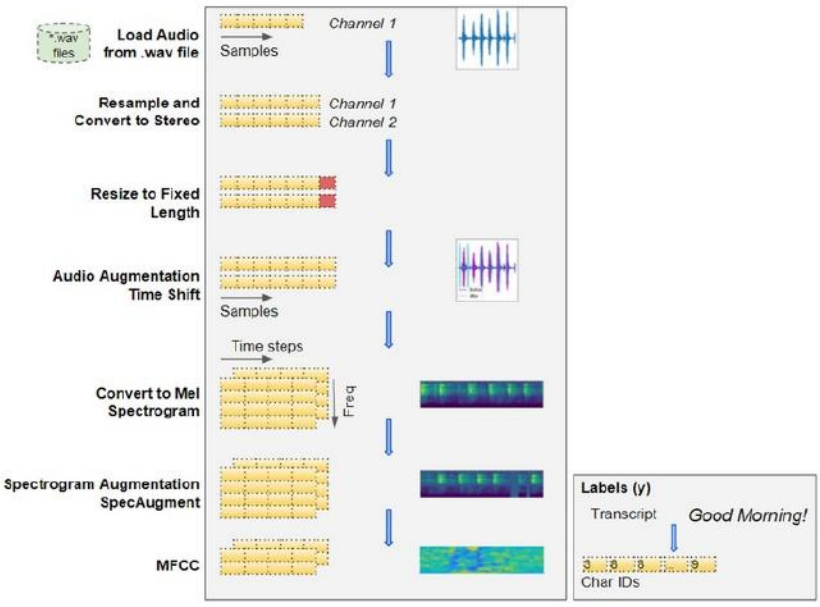
- Input **features** (**X**): audio clips of spoken words
- Target **labels** (**y**): a text transcript of what was spoken



The goal of the model is to learn how to take the input audio and predict the text content of the words and sentences that were uttered.

1.1.1 Data pre-processing

In the sound classification **Lab**, we explain, step-by-step, the transforms that are used to process audio data for deep learning models. With human speech as well we follow a similar approach. There are several Python libraries that provide the functionality to do this, with **librosa** being one of the most popular.



1.1.2 Load Audio Files

- Start with input data that consists of audio files of the spoken speech in an audio format such as **`.wav`** or **`.mp3`**.
- Read the audio data from the file and load it into a **2D Numpy** array. This array consists of a sequence of numbers, each representing a measurement of the intensity or amplitude of the sound at a particular moment in time. The number of such measurements is determined by the sampling rate. For instance, if the sampling rate was 44.1kHz, the Numpy array will have a single row of 44,100 numbers for 1 second of audio.
- Audio can have one or two channels, known as mono or stereo, in common parlance. With two-channel audio, we would have another similar sequence of amplitude numbers for the second channel. In other words, our Numpy array will be 3D, with a depth of 2.

1.1.3 Convert to uniform dimensions: sample rate, channels, and duration

- We might have a lot of variation in our audio data items. Clips might be sampled at different rates, or have a different number of channels. The clips will most likely have different durations. As explained above this means that the dimensions of each audio item will be different.
- Since our deep learning models expect all our input items to have a similar size, we now perform some data cleaning steps to standardize the dimensions of our audio data. We resample the audio so that every item has the same sampling rate. We convert all items to the same number of channels. All items also have to be converted to the same audio duration. This involves padding the shorter sequences or truncating the longer sequences.
- If the quality of the audio was poor, we might enhance it by applying a noise-removal algorithm to eliminate background noise so that we can focus on the spoken audio.

1.1.4 Data Augmentation of raw audio

- We could apply some data augmentation techniques to add more variety to our input data and help the model learn to generalize to a wider range of inputs. We could Time Shift our audio left or right randomly by a small percentage, or change the Pitch or the Speed of the audio by a small amount.

1.1.5 Mel Spectrograms

- This raw audio is now converted to Mel Spectrograms. A Spectrogram captures the nature of the audio as an image by decomposing it into the set of frequencies that are included in it.

1.1.6 MFCC

- For human speech, in particular, it sometimes helps to take one additional step and convert the Mel Spectrogram into MFCC (Mel Frequency Cepstral Coefficients). MFCCs produce a compressed representation of the Mel Spectrogram by extracting only the most essential frequency coefficients, which correspond to the frequency ranges at which humans speak.

1.1.7 Data Augmentation of Spectrograms

- We can now apply another data augmentation step on the Mel Spectrogram images, using a technique known as SpecAugment. This involves Frequency and Time Masking that randomly masks out either vertical (ie. Time Mask) or horizontal (ie. Frequency Mask) bands of information from the Spectrogram. NB: I'm not sure whether this can also be applied to MFCCs and whether that produces good results.

We have now transformed our original raw audio file into Mel Spectrogram (or MFCC) images after data cleaning and augmentation.

We also need to prepare the target labels from the transcript. This is simply regular text consisting of sentences of words, so we build a vocabulary from each character in the transcript and convert them into character IDs.
This gives us our input features and our target labels. This data is ready to be input into our deep learning model.

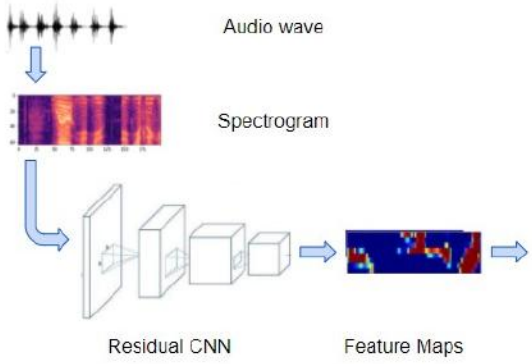
1.2 Architecture

There are many variations of deep learning architecture for ASR. Two commonly used approaches are:

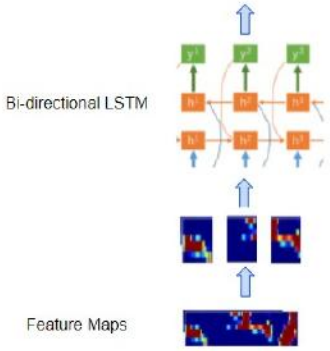
- A CNN (Convolutional Neural Network) plus RNN-based (Recurrent Neural Network) architecture that uses the CTC Loss algorithm to demarcate each character of the words in the speech. eg. Baidu's Deep Speech model.
- An RNN-based sequence-to-sequence network that treats each 'slice' of the spectrogram as one element in a sequence eg. Google's Listen Attend Spell (LAS) model.

Let's pick the first approach above and explore in more detail how that works. At a high level, the model consists of these blocks:

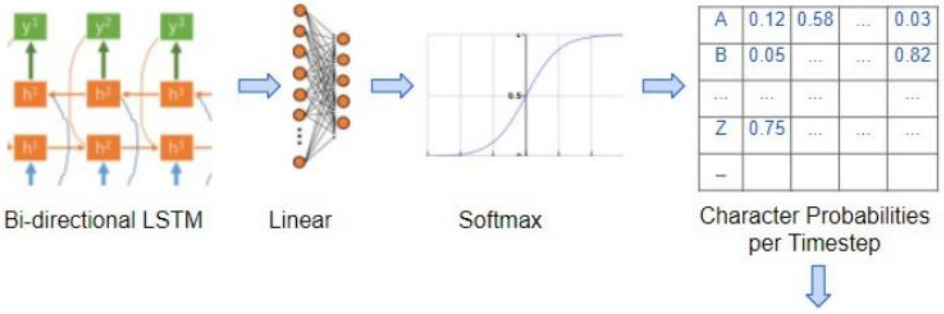
- A regular convolutional network consisting of a few Residual CNN layers that process the input spectrogram images and output feature maps of those images.



A regular recurrent network consisting of a few Bidirectional LSTM layers that process the feature maps as a series of distinct timesteps or 'frames' that correspond to our desired sequence of output characters. (An LSTM is a very commonly used type of recurrent layer, whose full form is Long Short Term Memory). In other words, it takes the feature maps which are a continuous representation of the audio, and converts them into a discrete representation.



- A linear layer with softmax that uses the LSTM outputs to produce cgaracter probabilities for each timestep of the output.

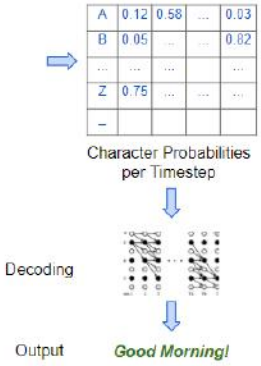


- We also have linear layers that sit between the convolution and recurrent networks and help to reshape the outputs of one network to the inputs of the other.

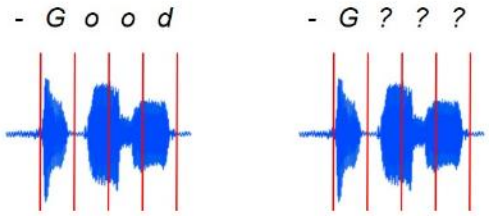
So our model takes the Spectrogram images and outputs character probabilities for each timestep or 'frame' in that Spectrogram.

1.2.1 Align the sequences

If you think about this a little bit, you'll realize that there is still a major missing piece in our puzzle. Our eventual goal is to map those timesteps or 'frames' to individual characters in our target transcript.

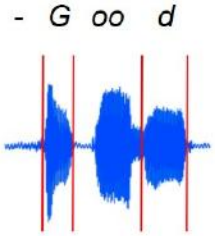


But for a particular spectrogram, how do we know how many frames there should be? How do we know exactly where the boundaries of each frame are? How do we align the audio with each character in the text transcript?



The audio and the spectrogram images are not pre-segmented to give us this information.

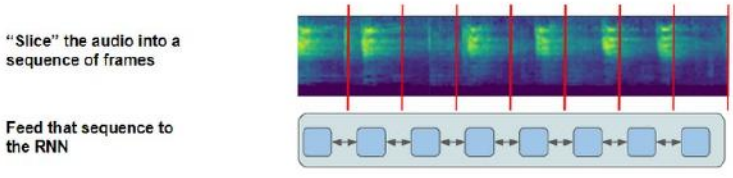
- In the spoken audio, and therefore in the spectrogram, the sound of each character could be of different durations.
- There could be gaps and pauses between these characters.
- Several characters could be merged together.
- Some characters could be repeated. eg. in the word 'apple', how do we know whether that "p" sound in the audio actually corresponds to one or two "p"s in the transcript?



This is actually a very challenging problem, and what makes ASR so tough to get right. It is the distinguishing characteristic that differentiates ASR from other audio applications like classification and so on.
The way we tackle this is by using an ingenious algorithm with a fancy-sounding name — it is called Connectionist Temporal Classification, or CTC for short. Since I am not 'fancy people' and find it difficult to remember that long name, I will just use the name CTC to refer to it

1.2.2 CTC Algorithm — Training and Inference

CTC is used to align the input and output sequences when the input is continuous and the output is discrete, and there are no clear element boundaries that can be used to map the input to the elements of the output sequence. What makes this so special is that it performs this alignment automatically, without requiring you to manually provide that alignment as part of the labeled training data. That would have made it extremely expensive to create the training datasets.
As we discussed above, the feature maps that are output by the convolutional network in our model are sliced into separate frames and input to the recurrent network. Each frame corresponds to some timestep of the original audio wave. However, the number of frames and the duration of each frame are chosen by you as hyperparameters when you design the model. For each frame, the recurrent network followed by the linear classifier then predicts probabilities for each character from the vocabulary.



The job of the CTC algorithm is to take these character probabilities and derive the correct sequence of characters. To help it handle the challenges of alignment and repeated characters that we just discussed, it introduces the concept of a 'blank' pseudo-character (denoted by "-") into the vocabulary. Therefore the character probabilities output by the network also include the probability of the blank character for each frame.
Note that a blank is not the same as a 'space'. A space is a real character while a blank means the absence of any character, somewhat like a 'null' in most programming languages. It is used only to demarcate the boundary between two characters.

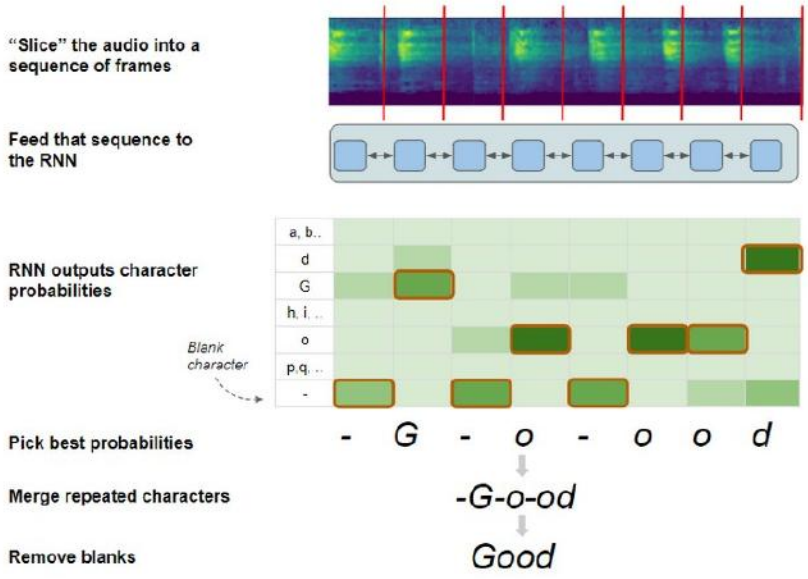
CTC works in two modes:

- **CTC Loss** (during Training): It has a ground truth target transcript and tries to train the network to maximize the probability of outputting that correct transcript.
- **CTC Decoding** (during Inference): Here we don't have a target transcript to refer to, and have to predict the most likely sequence of characters.
-

Let's explore these a little more to understand what the algorithm does. We'll start with CTC Decoding as it is a little simpler.

1.2.3 CTC Decoding

- Use the character probabilities to pick the most likely character for each frame, including blanks. eg. "*-G-o-ood*"



- Merge any characters that are repeated, and not separated by a blank. For instance, we can merge the "*oo*" into a single "*o*", but we cannot merge the "*o-oo*". This is how the CTC is able to distinguish that there are two separate "o"s and produce words spelled with repeated characters. eg. "*-G-o-od*"
- Finally, since the blanks have served their purpose, it removes all blank characters. eg. "*Good*".

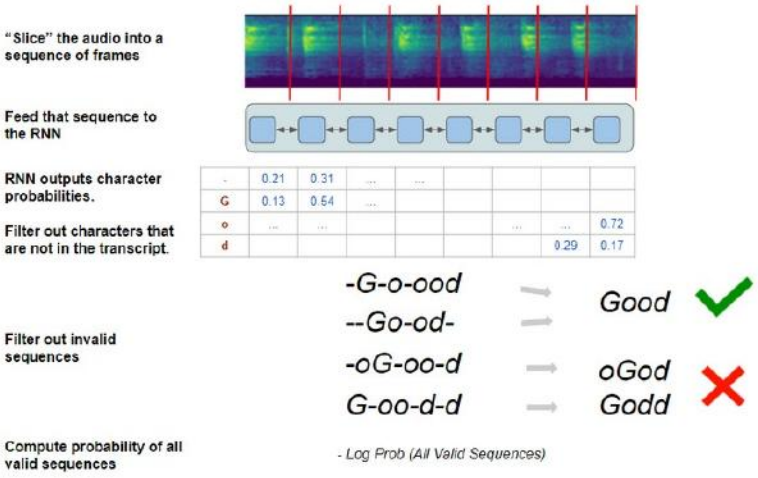
1.2.4 CTC Loss

The Loss is computed as the probability of the network predicting the correct sequence. To do this, the algorithm lists out all possible sequences the network can predict, and from that it selects the subset that match the target transcript.

To identify that subset from the full set of possible sequences, the algorithm narrows down the possibilities as follows:

- Keep only the probabilities for characters that occur in the target transcript and discard the rest. eg. It keeps probabilities only for "G", "o", "d", and "-".

- Using the filtered subset of characters, for each frame, select only those characters which occur in the same order as the target transcript. eg. Although “G” and “o” are both valid characters, an order of “Go” is a valid sequence whereas “oG” is an invalid sequence.



With these constraints in place, the algorithm now has a set of valid character sequences, all of which will produce the correct target transcript. eg. Using the same steps that were used during Inference, “*-G-o-ood”* and “ *— Go-od-*” will both result in a final output of “Good”.

It then uses the individual character probabilities for each frame, to compute the overall probability of generating all of those valid sequences. The goal of the network is to learn how to maximize that probability and therefore reduce the probability of generating any invalid sequence.

Strictly speaking, since a neural network *minimizes* loss, the CTC Loss is computed as the *negative* log probability of all valid sequences. As the network minimizes that loss via back-propagation during training, it adjusts all of its weights to produce the correct sequence.

To actually do this, however, is much more complicated than what I’ve described here. The challenge is that there is a huge number of possible combinations of characters to produce a sequence. With our simple example alone, we can have 4 characters per frame. With 8 frames that gives us 4 ** 8 combinations (= 65536). For any realistic transcript with more characters and more frames, this number increases exponentially. That makes it computationally impractical to simply exhaustively list out the valid combinations and compute their probability.

Solving this efficiently is what makes CTC so innovative. It is a fascinating algorithm and it is well worth understanding the nuances of how it achieves this. That merits a complete article by itself which I plan to write shortly. But for now, we have focused on building intuition about what CTC does, rather than going into how it works.

1.2.5 Metrics — Word Error Rate (WER)

After training our network, we must evaluate how well it performs. A commonly used metric for Speech-to-Text problems is the Word Error Rate (and Character Error Rate). It compares the predicted output and the target transcript, word by word (or character by character) to figure out the number of differences between them.

A difference could be a word that is present in the transcript but missing from the prediction (counted as a Deletion), a word that is not in the transcript but has been added into the prediction (an Insertion), or a word that is altered between the prediction and the transcript (a Substitution).

The metric formula is fairly straightforward. It is the percent of differences relative to the total number of words.

$$\text{Word Error Rate} = \frac{\text{Inserted} + \text{Deleted} + \text{Substituted}}{\text{Total words in transcript}}$$

$$= \frac{1 + 1 + 1}{6}$$

$$= 0.5$$

1.3 Language Model

So far, our algorithm has treated the spoken audio as merely corresponding to a sequence of characters from some language. But when put together into words and sentences will those characters actually make sense and have meaning?

A common application in Natural Language Processing (NLP) is to build a Language Model. It captures how words are typically used in a language to construct sentences, paragraphs, and documents. It could be a general-purpose model about a language such as English or Korean, or it could be a model that is specific to a particular domain such as medical or legal.

Once you have a Language Model, it can become the foundation for other applications. For instance, it could be used to predict the next word in a sentence, to discern the sentiment of some text (eg. is this a positive book review), to answer questions via a chatbot, and so on.

So, of course, it can also be used to optionally enhance the quality of our ASR outputs by guiding the model to generate predictions that are more likely as per the Language Model.

1.4 Beam Search

While describing the CTC Decoder during Inference, we implicitly assumed that it always picks a single character with the highest probability at each timestep. This is known as Greedy Search.
However, we know that we can get better results using an alternative method called Beam Search.

Although Beam Search is often used with NLP problems in general, it is not specific to ASR, so I'm mentioning it here just for completeness. If you'd like to know more, please take a look at my article that describes Beam Search in full detail.

1.5 Conclusion

Hopefully, this now gives you a sense of the building blocks and techniques that are used to solve ASR problems. In the older pre-deep-learning days, tackling such problems via classical approaches required an understanding of concepts like phonemes and a lot of domain-specific data preparation and algorithms.
However, as we've just seen with deep learning, we required hardly any feature engineering involving knowledge of audio and speech. And yet, it is able to produce excellent results that continue to surprise us!
And finally, if you liked this article, you might also enjoy my other series on Transformers, Geolocation Machine Learning, and Image Caption architectures.

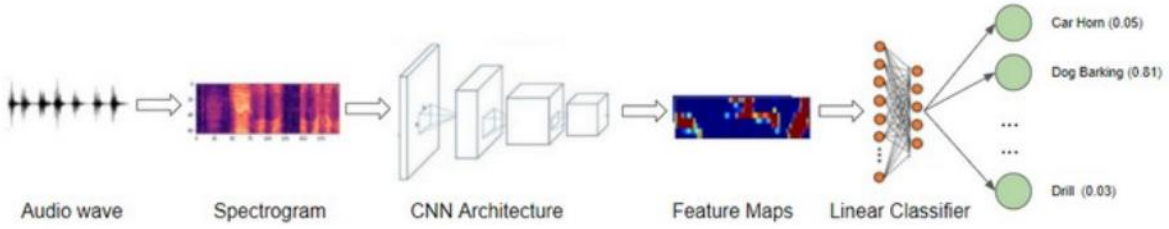
Lab 1

Sound Classification, Step-by-Step

An end-to-end example and architecture for audio deep learning's foundational application scenario, in plain English.
Sound Classification is one of the most widely used applications in Audio Deep Learning. It involves learning to classify sounds and to predict the category of that sound. This type of problem can be applied to many practical scenarios e.g. classifying music clips to identify the genre of the music, or classifying short utterances by a set of speakers to identify the speaker based on the voice.

In this lab, we will walk through **a simple demo application** so as to understand the approach used to solve such audio classification problems. The goal throughout will be to understand not just how something works but why it works that way.

Just like classifying hand-written digits using the **MNIST** dataset is considered a 'Hello World"-type problem for Computer Vision, we can think of this application as the introductory problem for audio deep learning.



There are many suitable datasets available for sounds of different types. These datasets contain a large number of audio samples, along with a class label for each sample that identifies what type of sound it is, based on the problem you are trying to address.

These class labels can often be obtained from some part of the filename of the audio sample or from the sub-folder name in which the file is located. Alternately the class labels are specified in a separate metadata file, usually in TXT, JSON, or CSV format.

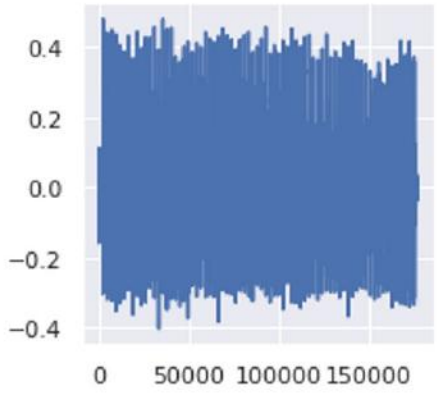
1.1 Example problem — Classifying ordinary city sounds

For our demo, we will use the **Urban Sound 8K** dataset that consists of a corpus of ordinary sounds recorded from day-to-day city life. The sounds are taken from **10 classes** such as drilling, dogs barking, and sirens. Each sound sample is labeled with the class to which it belongs.

After downloading the dataset, we see that it consists of two parts:

- **Audio files** in the `audio` folder: It has 10 sub-folders named `fold1` through `fold10`. Each sub-folder contains a number of `.wav` audio samples eg. `fold1/103074-7-1-0.wav`
- **Metadata** in the 'metadata' folder: It has a file `UrbanSound8K.csv` that contains information about each audio sample in the dataset such as its filename, its class label, the `fold` sub-folder location, and so on. The class label is a numeric **Class ID** from 0–9 for each of the 10 classes. eg. the number 0 means air conditioner, 1 is a car horn, and so on.

The samples are around 4 seconds in length. Here's what one sample looks like:

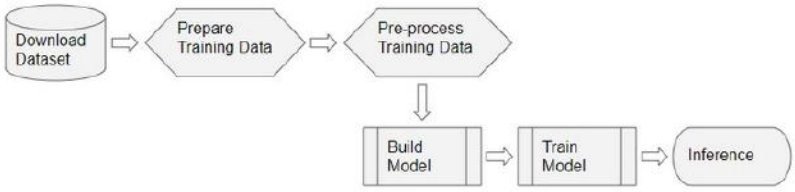


Sample_rate=44100, Channels=2, Bits=16, Encoding_type=PCM_S

The recommendation of the dataset creators is to use the folds for doing 10-fold cross-validation to report metrics and evaluate the performance of your model. However, since our goal in this article is primarily as a demo of an audio deep learning example rather than to obtain the best metrics, we will ignore the folds and treat all the samples simply as one large dataset.

1.1.1 Prepare training data

As for most deep learning problems, we will follow these steps:



The training data for this problem will be fairly simple:

- The features (X) are the audio file paths
- The target labels (y) are the class names

Since the dataset has a metadata file that contains this information already, we can use that directly. The metadata contains information about each audio file.

	slice_file_name	fsID	start	end	salience	fold	classID	class
0	100032-3-0-0.wav	100032	0.0	0.317551	1	5	3	dog_bark
1	100263-2-0-117.wav	100263	58.5	62.500000	1	5	2	children_playing
2	100263-2-0-121.wav	100263	60.5	64.500000	1	5	2	children_playing
3	100263-2-0-126.wav	100263	63.0	67.000000	1	5	2	children_playing
4	100263-2-0-137.wav	100263	68.5	72.500000	1	5	2	children_playing

Since it is a CSV file, we can use Pandas to read it. We can prepare the feature and label data from the metadata.

```
# ----------------------------
```

```
# Prepare training data from Metadata file
# ----------------------------
import pandas as pd
from pathlib import Path

download_path = Path.cwd()/'UrbanSound8K'

# Read metadata file
metadata_file = download_path/'metadata'/'UrbanSound8K.csv'
df = pd.read_csv(metadata_file)
df.head()

# Construct file path by concatenating fold and file name
df['relative_path'] = '/fold' + df['fold'].astype(str) + '/' + df['slice_file_name'].astype(str)

# Take relevant columns
df = df[['relative_path', 'classID']]
df.head()
```

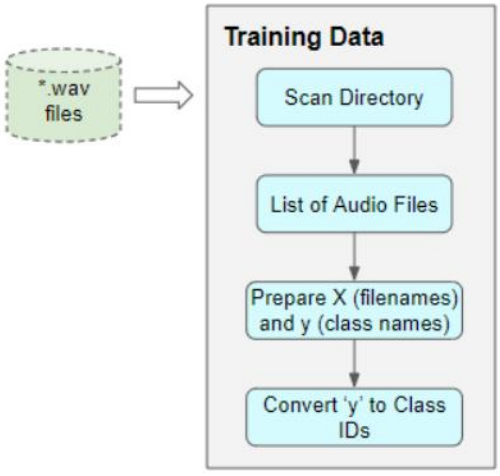
This gives us the information we need for our training data.

	relative_path	classID
0	/fold5/100032-3-0-0.wav	3
1	/fold5/100263-2-0-117.wav	2
2	/fold5/100263-2-0-121.wav	2
3	/fold5/100263-2-0-126.wav	2
4	/fold5/100263-2-0-137.wav	2

Scan the audio file directory when metadata isn't available

Having the metadata file made things easy for us. How would we prepare our data for datasets that do not contain a metadata file?

Many datasets consist of only audio files arranged in a folder structure from which class labels can be derived. To prepare our training data in this format, we would do the following:



- Scan the directory and prepare a list of all the audio file paths.
- Extract the class label from each file name, or from the name of the parent sub-folder
- Map each class name from text to a numeric class ID

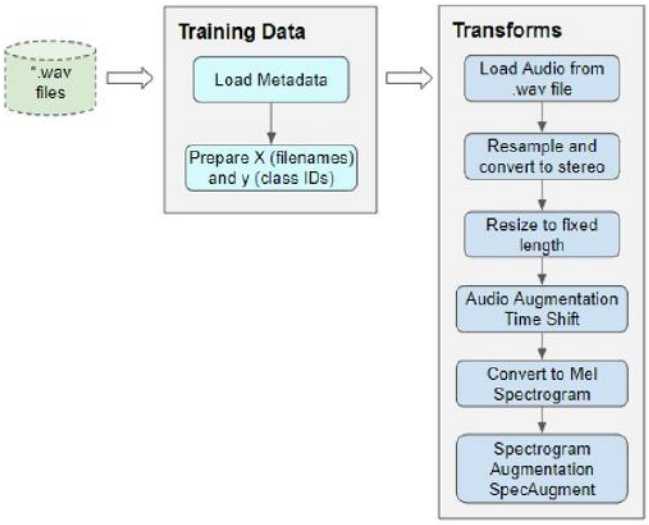
With or without metadata, the result would be the same — features consisting of a list of audio file names and target labels consisting of class IDs.

1.2 Audio Pre-processing: Define Transforms

This training data with audio file paths cannot be input directly into the model. We have to load the audio data from the file and process it so that it is in a format that the model expects.

This audio pre-processing will all be done dynamically at runtime when we will read and load the audio files. This approach is similar to what we would do with image files as well. Since audio data, like image data, can be fairly large and memory-intensive, we don't want to read the entire dataset into memory all at once, ahead of time. So we keep only the audio file names (or image file names) in our training data.
Then, at runtime, as we train the model **one batch at a time**, we will load the audio data for that batch and process it by applying a series of transforms to the audio. That way we keep audio data for only one batch in memory at a time.

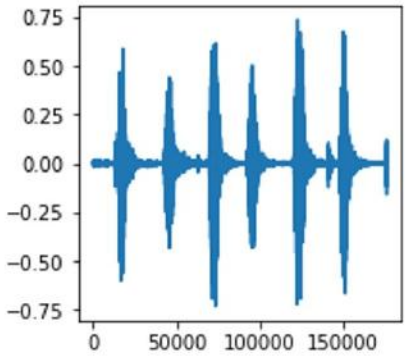
With image data, we might have a pipeline of transforms where we first read the image file as pixels and load it. Then we might apply some image processing steps to reshape and resize the data, crop them to a fixed size and convert them into grayscale from RGB. We might also apply some image augmentation steps like rotation, flips, and so on.
The processing for audio data is very similar. Right now we're only defining the functions, they will be run a little later when we feed data to the model during training.



1.2.1 Read audio from a file

The first thing we need is to read and load the audio file in `.wav` format. Since we are using **Pytorch** for this example, the implementation below uses `torchaudio` for the audio processing, but `librosa` will work just as well.

```
import math, random
import torch
import torchaudio
from torchaudio import transforms
from IPython.display import Audio

class AudioUtil():
  # ----------------------------
  # Load an audio file. Return the signal as a tensor and the sample rate
  # ----------------------------
  @staticmethod
  def open(audio_file):
    sig, sr = torchaudio.load(audio_file)
    return (sig, sr)
```



1.2.2 Convert to two channels

Some of the sound files are mono (ie. 1 audio channel) while most of them are stereo (ie. 2 audio channels). Since our model expects all items to have the same dimensions, we will convert the mono files to stereo, by duplicating the first channel to the second.

```
# ----------------------------
# Convert the given audio to the desired number of channels
# ----------------------------
@staticmethod
def rechannel(aud, new_channel):
  sig, sr = aud

  if (sig.shape[0] == new_channel):
    # Nothing to do
    return aud

  if (new_channel == 1):
    # Convert from stereo to mono by selecting only the first channel
    resig = sig[:1, :]
  else:
    # Convert from mono to stereo by duplicating the first channel
    resig = torch.cat([sig, sig])

  return ((resig, sr))
```

1.2.3 Standardize sampling rate

Some of the sound files are sampled at a sample rate of 48000Hz, while most are sampled at a rate of 44100Hz. This means that 1 second of audio will have an array size of 48000 for some sound files, while it will have a smaller array size of 44100 for the others. Once again, we must standardize and convert all audio to the same sampling rate so that all arrays have the same dimensions.

```
# ----------------------------
# Since Resample applies to a single channel, we resample one channel at a time
# ----------------------------
@staticmethod
def resample(aud, newsr):
  sig, sr = aud

  if (sr == newsr):
    # Nothing to do
    return aud

  num_channels = sig.shape[0]
  # Resample first channel
  resig = torchaudio.transforms.Resample(sr, newsr)(sig[:1,:])
  if (num_channels > 1):
```

```
    # Resample the second channel and merge both channels
    retwo = torchaudio.transforms.Resample(sr, newsr)(sig[1:,:])
    resig = torch.cat([resig, retwo])

  return ((resig, newsr))
```

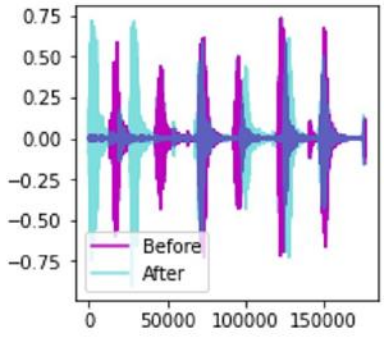
1.2.4 Resize to the same length

We then resize all the audio samples to have the same length by either extending its duration by padding it with silence, or by truncating it. We add that method to our **`AudioUtil`** class.

```
# ----------------------------
# Pad (or truncate) the signal to a fixed length 'max_ms' in milliseconds
# ----------------------------
@staticmethod
def pad_trunc(aud, max_ms):
  sig, sr = aud
  num_rows, sig_len = sig.shape
  max_len = sr//1000 * max_ms

  if (sig_len > max_len):
    # Truncate the signal to the given length
    sig = sig[:,:max_len]

  elif (sig_len < max_len):
    # Length of padding to add at the beginning and end of the signal
    pad_begin_len = random.randint(0, max_len - sig_len)
    pad_end_len = max_len - sig_len - pad_begin_len

    # Pad with 0s
    pad_begin = torch.zeros((num_rows, pad_begin_len))
    pad_end = torch.zeros((num_rows, pad_end_len))

    sig = torch.cat((pad_begin, sig, pad_end), 1)

  return (sig, sr)
```

1.2.5 Data Augmentation: Time Shift

Next, we can do data augmentation on the raw audio signal by applying a Time Shift to shift the audio to the left or the right by a random amount.

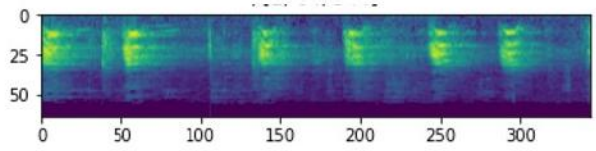



```
# ----------------------------
# Shifts the signal to the left or right by some percent. Values at the end
# are 'wrapped around' to the start of the transformed signal.
# ----------------------------
@staticmethod
def time_shift(aud, shift_limit):
  sig,sr = aud
  _, sig_len = sig.shape
  shift_amt = int(random.random() * shift_limit * sig_len)
  return (sig.roll(shift_amt), sr)
```

1.2.6 Mel Spectrogram

We then convert the augmented audio to a **Mel Spectrogram**. They capture the essential features of the audio and are often the most suitable way to input audio data into deep learning models. To get more background about this, you might want to read my articles (here and here) which explain in simple words what a **Mel Spectrogram** is, why they are crucial for audio deep learning, as well as how they are generated and how to tune them for getting the best performance from your models.

```
# ----------------------------
# Generate a Spectrogram
# ----------------------------
@staticmethod
def spectro_gram(aud, n_mels=64, n_fft=1024, hop_len=None):
  sig,sr = aud
  top_db = 80

  # spec has shape [channel, n_mels, time], where channel is mono, stereo etc
  spec = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(sig)

  # Convert to decibels
  spec = transforms.AmplitudeToDB(top_db=top_db)(spec)
  return (spec)
```

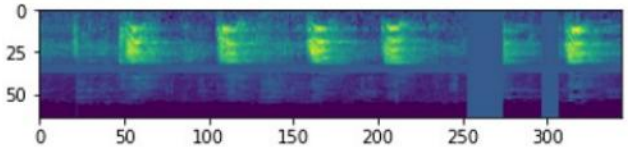


1.2.7 Data Augmentation: Time and Frequency Masking

Now we can do another round of augmentation, this time on the Mel Spectrogram rather than on the raw audio. We will use a technique called **SpecAugment** that uses these two methods:

- **Frequency mask** — randomly mask out a range of consecutive frequencies by adding horizontal bars on the spectrogram.
- **Time mask** — similar to frequency masks, except that we randomly block out ranges of time from the spectrogram by using vertical bars.

```
# ----------------------------
# Augment the Spectrogram by masking out some sections of it in both the frequency
# dimension (ie. horizontal bars) and the time dimension (vertical bars) to prevent
# overfitting and to help the model generalise better. The masked sections are
# replaced with the mean value.
# ----------------------------
@staticmethod
def spectro_augment(spec, max_mask_pct=0.1, n_freq_masks=1, n_time_masks=1):
  _, n_mels, n_steps = spec.shape
  mask_value = spec.mean()
  aug_spec = spec

  freq_mask_param = max_mask_pct * n_mels
  for _ in range(n_freq_masks):
    aug_spec = transforms.FrequencyMasking(freq_mask_param)(aug_spec, mask_value)

  time_mask_param = max_mask_pct * n_steps
  for _ in range(n_time_masks):
    aug_spec = transforms.TimeMasking(time_mask_param)(aug_spec, mask_value)

  return aug_spec
```



Mel Spectrogram after SpecAugment. Notice the horizontal and vertical mask bands

1.2.8 Define Custom Data Loader

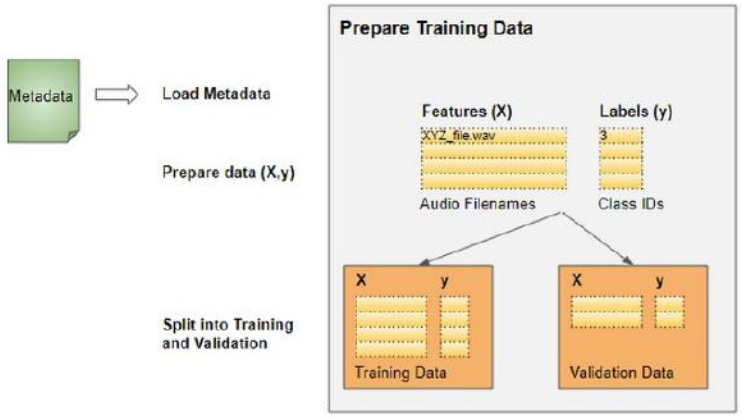
Now that we have defined all the pre-processing transform functions we will define a custom Pytorch Dataset object.
To feed your data to a model with Pytorch, we need two objects:

- A custom **Dataset** object that uses all the audio transforms to pre-process an audio file and prepares one data item at a time.
- A built-in **DataLoader** object that uses the **Dataset** object to fetch individual data items and packages them into a batch of data.

```
from torch.utils.data import DataLoader, Dataset, random_split
import torchaudio

# ----------------------------
# Sound Dataset
# ----------------------------
class SoundDS(Dataset):
  def __init__(self, df, data_path):
    self.df = df
    self.data_path = str(data_path)
    self.duration = 4000
    self.sr = 44100
    self.channel = 2
    self.shift_pct = 0.4

  # ----------------------------
  # Number of items in dataset
  # ----------------------------
  def __len__(self):
    return len(self.df)

  # ----------------------------
  # Get i'th item in dataset
  # ----------------------------
  def __getitem__(self, idx):
    # Absolute file path of the audio file - concatenate the audio directory with
    # the relative path
    audio_file = self.data_path + self.df.loc[idx, 'relative_path']
    # Get the Class ID
    class_id = self.df.loc[idx, 'classID']

    aud = AudioUtil.open(audio_file)
    # Some sounds have a higher sample rate, or fewer channels compared to the
    # majority. So make all sounds have the same number of channels and same
    # sample rate. Unless the sample rate is the same, the pad_trunc will still
    # result in arrays of different lengths, even though the sound duration is
    # the same.
    reaud = AudioUtil.resample(aud, self.sr)
    rechan = AudioUtil.rechannel(reaud, self.channel)

    dur_aud = AudioUtil.pad_trunc(rechan, self.duration)
    shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
    sgram = AudioUtil.spectro_gram(shift_aud, n_mels=64, n_fft=1024, hop_len=None)
    aug_sgram = AudioUtil.spectro_augment(sgram, max_mask_pct=0.1, n_freq_masks=2, n_time_masks=2)

    return aug_sgram, class_id
```

1.2.9 Prepare Batches of Data with the Data Loader

All of the functions we need to input our data to the model have now been defined.We use our custom **Dataset** to load the **Features** and **Labels** from our **Pandas** dataframe and split that data randomly in an **80:20 ratio** into training and validation sets. We then use them to create our training and validation Data Loaders.



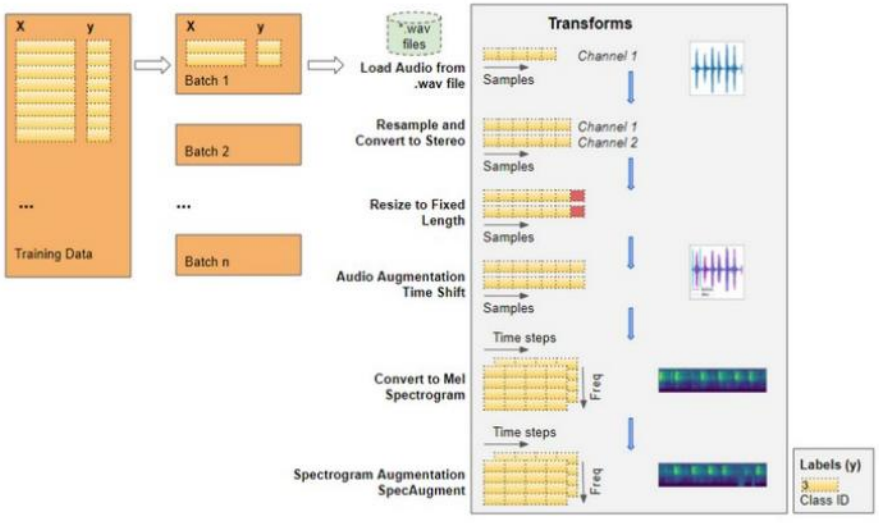

```
from torch.utils.data import
random_split

myds = SoundDS(df, data_path)

# Random split of 80:20 between training and validation
num_items = len(myds)
num_train = round(num_items * 0.8)
num_val = num_items - num_trainCreate Model
```

The data processing steps that we just did are the most unique aspects of our audio classification problem. From here on, the model and training procedure are quite similar to what is commonly used in a standard image classification problem and are not specific to audio deep learning.

Since our data now consists of Spectrogram images, we build a CNN classification architecture to process them. It has four convolutional blocks which generate the feature maps. That data is then reshaped into the format we need so it can be input into the linear classifier layer, which finally outputs the predictions for the 10 classes.

```
train_ds, val_ds = random_split(myds, [num_train, num_val])

# Create training and validation data loaders
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=16, shuffle=True)
val_dl = torch.utils.data.DataLoader(val_ds, batch_size=16, shuffle=False)
```

When we start training, the Data Loader will randomly fetch one batch of input Features containing the list of audio file names and run the pre-processing audio transforms on each audio file. It will also fetch a batch of the corresponding target Labels containing the class IDs. Thus it will output one batch of training data at a time, which can directly be fed as input to our deep learning model.



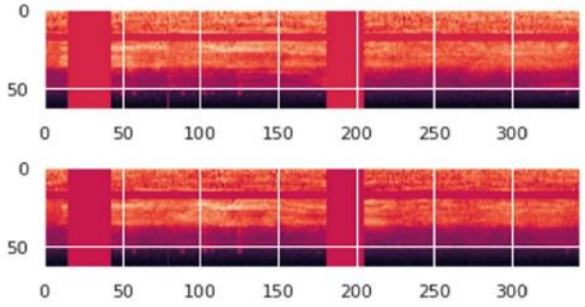
Let's walk through the steps as our data gets transformed, starting with an audio file:

- The audio from the file gets loaded into a Numpy array of shape (num_channels, num_samples). Most of the audio is sampled at 44.1kHz and is about 4 seconds in duration, resulting in 44,100 * 4 = 176,400 samples. If the audio has 1 channel, the shape of the array will be (1, 176,400). Similarly, audio of 4 seconds duration with 2 channels and sampled at 48kHz will have 192,000 samples and a shape of (2, 192,000).
- Since the channels and sampling rates of each audio are different, the next two transforms resample the audio to a standard 44.1kHz and to a standard 2 channels.
- Since some audio clips might be more or less than 4 seconds, we also standardize the audio duration to a fixed length of 4 seconds. Now arrays for all items have the same shape of (2, 176,400)
- The Time Shift data augmentation now randomly shifts each audio sample forward or backward. The shapes are unchanged.
- The augmented audio is now converted into a Mel Spectrogram, resulting in a shape of (num_channels, Mel freq_bands, time_steps) = (2, 64, 344)
- The SpecAugment data augmentation now randomly applies Time and Frequency Masks to the Mel Spectrograms. The shapes are unchanged.

Thus, each batch will have two tensors, one for the *X* feature data containing the Mel Spectrograms and the other for the *y* target labels containing numeric Class IDs. The batches are picked randomly from the training data for each training epoch.
Each batch has a shape of (**`batch_sz, num_channels, Mel freq_bands, time_steps`**)

```
(torch.Size([16,2?64,344]), torch.Size([16])
```

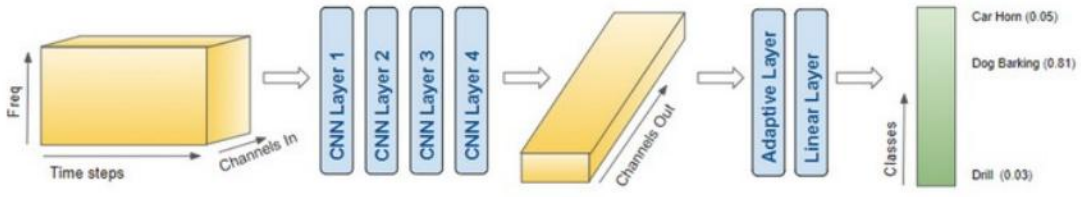
We can visualize one item from the batch. We see the Mel Spectrogram with vertical and horizontal stripes showing the Frequency and Time Masking data augmentation.



The data is now ready for input to the model.

1.2.10 Create Model

The data processing steps that we just did are the most unique aspects of our audio classification problem. From here on, the model and training procedure are quite similar to what is commonly used in a standard image classification problem and are not specific to audio deep learning.
Since our data now consists of Spectrogram images, we build a CNN classification architecture to process them. It has four **convolutional** blocks which generate the feature maps. That data is then reshaped into the format we need so it can be input into the linear classifier layer, which finally outputs the predictions for the **10 classes**.



A few more details about how the model processes a batch of data:

- A batch of images is input to the model with shape (batch_sz, num_channels, Mel freq_bands, time_steps) ie. (16, 2, 64, 344).
- Each CNN layer applies its filters to step up the image depth ie. number of channels. The image width and height are reduced as the kernels and strides are applied. Finally, after passing through the four CNN layers, we get the output feature maps ie. (16, 64, 4, 22).
- This gets pooled and flattened to a shape of (16, 64) and then input to the Linear layer.
- The Linear layer outputs one prediction score per class ie. (16, 10)

```
import torch.nn.functional as F
from torch.nn import init

# ----------------------------
# Audio Classification Model
# ----------------------------
class AudioClassifier (nn.Module):
    # ----------------------------
    # Build the model architecture
    # ----------------------------
    def __init__(self):
        super().__init__()
        conv_layers = []

        # First Convolution Block with Relu and Batch Norm. Use Kaiming Initialization
        self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(8)
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        self.conv1.bias.data.zero_()
        conv_layers += [self.conv1, self.relu1, self.bn1]

        # Second Convolution Block
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(16)
        init.kaiming_normal_(self.conv2.weight, a=0.1)
        self.conv2.bias.data.zero_()
        conv_layers += [self.conv2, self.relu2, self.bn2]

        # Second Convolution Block
        self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(32)
        init.kaiming_normal_(self.conv3.weight, a=0.1)
        self.conv3.bias.data.zero_()
        conv_layers += [self.conv3, self.relu3, self.bn3]

        # Second Convolution Block
        self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu4 = nn.ReLU()
        self.bn4 = nn.BatchNorm2d(64)
        init.kaiming_normal_(self.conv4.weight, a=0.1)
        self.conv4.bias.data.zero_()
        conv_layers += [self.conv4, self.relu4, self.bn4]

        # Linear Classifier
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=64, out_features=10)

        # Wrap the Convolutional Blocks
        self.conv = nn.Sequential(*conv_layers)

    # ----------------------------
    # Forward pass computations
    # ----------------------------
    def forward(self, x):
        # Run the convolutional blocks
        x = self.conv(x)
```

```
        # Adaptive pool and flatten for input to linear layer
        x = self.ap(x)
        x = x.view(x.shape[0], -1)

        # Linear layer
        x = self.lin(x)
        # Final output
        return x

# Create the model and put it on the GPU if available
myModel = AudioClassifier()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
myModel = myModel.to(device)
# Check that it is on Cuda
next(myModel.parameters()).device
```

1.2.11 Training

We are now ready to create the training loop to train the model.
We define the functions for the optimizer, loss, and scheduler to dynamically vary our learning rate as training progresses, which usually allows training to converge in fewer epochs.
We train the model for several epochs, processing a batch of data in each iteration. We keep track of a simple accuracy metric which measures the percentage of correct predictions.

```
# ----------------------------
# Training Loop
# ----------------------------
def training(model, train_dl, num_epochs):
  # Loss Function, Optimizer and Scheduler
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
  scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001,
                                                  steps_per_epoch=int(len(train_dl)),
                                                  epochs=num_epochs,
                                                  anneal_strategy='linear')

  # Repeat for each epoch
  for epoch in range(num_epochs):
    running_loss = 0.0
    correct_prediction = 0
    total_prediction = 0

    # Repeat for each batch in the training set
    for i, data in enumerate(train_dl):
        # Get the input features and target labels, and put them on the GPU
        inputs, labels = data[0].to(device), data[1].to(device)

        # Normalize the inputs
        inputs_m, inputs_s = inputs.mean(), inputs.std()
        inputs = (inputs - inputs_m) / inputs_s

        # Zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs), labels)
        loss.backward()
        optimizer.step()
        scheduler.step()

        # Keep stats for Loss and Accuracy
        running_loss += loss.item()

        # Get the predicted class with the highest score
        _, prediction = torch.max(outputs,1)
        # Count of predictions that matched the target label
        correct_prediction += (prediction == labels).sum().item()
```

```
        total_prediction += prediction.shape[0]

        #if i % 10 == 0:    # print every 10 mini-batches
        #    print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 10))

    # Print stats at the end of the epoch
    num_batches = len(train_dl)
    avg_loss = running_loss / num_batches
    acc = correct_prediction/total_prediction
    print(f'Epoch: {epoch}, Loss: {avg_loss:.2f}, Accuracy: {acc:.2f}')

  print('Finished Training')

num_epochs=2   # Just for demo, adjust this higher.
training(myModel, train_dl, num_epochs)
```

1.2.12 Inference

Ordinarily, as part of the training loop, we would also evaluate our metrics on the validation data. We would then do inference on unseen data, perhaps by keeping aside a test dataset from the original data. However, for the purposes of this demo, we will use the validation data for this purpose.
We run an inference loop taking care to disable the gradient updates. The forward pass is executed with the model to get predictions, but we do not need to backpropagate or run the optimizer.

```
# ----------------------------
# Inference
# ----------------------------
def inference (model, val_dl):
  correct_prediction = 0
  total_prediction = 0

  # Disable gradient updates
  with torch.no_grad():
    for data in val_dl:
      # Get the input features and target labels, and put them on the GPU
      inputs, labels = data[0].to(device), data[1].to(device)

      # Normalize the inputs
      inputs_m, inputs_s = inputs.mean(), inputs.std()
      inputs = (inputs - inputs_m) / inputs_s

      # Get predictions
      outputs = model(inputs)

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]

  acc = correct_prediction/total_prediction
  print(f'Accuracy: {acc:.2f}, Total items: {total_prediction}')

# Run inference on trained model with the validation set
inference(myModel, val_dl)
```

1.3 Conclusion

We have now seen an end-to-end example of sound classification which is one of the most foundational problems in audio deep learning. Not only is this used in a wide range of applications, but many of the concepts and techniques that we covered here will be relevant to more complicated audio problems such as automatic speech recognition where we start with human speech, understand what people are saying, and convert it to text.
And finally, if you liked this article, you might also enjoy my other series on Transformers, Geolocation Machine Learning, and Image Caption architectures.

Lab 2

Voice with jetson-voice

jetson-voice is an ASR/NLP/TTS **deep learning inference library** for Jetson Nano, TX1/TX2, Xavier NX, and AGX Xavier. It supports Python and JetPack 4.4.1 or newer.

The DNN models were trained with NeMo and deployed with TensorRT for optimized performance.
All computation is performed using the onboard GPU.

Currently the following capabilities are included:

- Automatic Speech Recognition (ASR)
  - Streaming ASR (QuartzNet)
  - Command/Keyword Recognition (MatchboxNet)
  - Voice Activity Detection (VAD Marblenet)
- Natural Language Processing (NLP)
  - Joint Intent/Slot Classification
  - Text Classification (Sentiment Analysis)
  - Token Classification (Named Entity Recognition)
  - Question/Answering (QA)
- Text-to-Speech (TTS)

The NLP models are using the DistilBERT transformer architecture for reduced memory usage and increased performance. For samples of the text-to-speech output, see the TTS Audio Samples section below.

2.1 Running the Container

jetson-voice is distributed as a **Docker** container due to the number of dependencies. There are pre-built containers images available on **DockerHub** for **JetPack 4.4.1** and newer:

```
dustynv/jetson-voice:r32.4.4    # JetPack 4.4.1 (L4T R32.4.4)
dustynv/jetson-voice:r32.5.0    # JetPack 4.5 (L4T R32.5.0) / JetPack 4.5.1 (L4T R32.5.1)
dustynv/jetson-voice:r32.6.1    # JetPack 4.6 (L4T R32.6.1)
dustynv/jetson-voice:r32.7.1    # JetPack 4.6.1 (L4T R32.7.1)
```

To download and run the container, you can simply clone this **repo** and use the **`docker/run.sh`** script:

```
$ git clone --branch dev https://github.com/dusty-nv/jetson-voice
$ cd jetson-voice
$ docker/run.sh
```

> **note**: if you want to use a USB microphone or speaker, **plug it** in ***before*** you start the container

There are some optional arguments to **`docker/run.sh`** that you can use:

- **`-r`** (**`--run`**) specifies a run command, otherwise the container will start in an interactive shell.
- **`-v`** (**`--volume`**) mount a directory from the host into the container (**`/host/path:/container/path`**)
- `--dev` starts the container in development mode, where all the source files are mounted for easy editing

The run script will automatically mount the `data/` directory into the container, which stores the models and other data files. If you save files from the container there, they will also show up under `data/` on the host.

2.2 Automatic Speech Recognition (ASR)

The **speech recognition** in **jetson-voice** is a **streaming service**, so it's intended to be used on live sources and transcribes the audio in 1-second chunks. It uses a QuartzNet-15x5 model followed by a CTC beamsearch decoder and language model, to further refine the raw output of the network. It detects breaks in the audio to determine the end of sentences.
For information about using the ASR APIs, please refer to `jetson_voice/asr.py` and see `examples/asr.py`

After you start the container, first run a test audio file (**`wav/ogg/flac`**) through `examples/asr.py` to verify that the system is functional. Run this command (and all subsequent commands) inside the container:

```
$ examples/asr.py --wav data/audio/dusty.wav

hi
hi hi this is dust
hi hi this is dusty check
hi hi this is dusty check one two
hi hi this is dusty check one two three
hi hi this is dusty check one two three.

what's the weather or
what's the weather going to be tomorrow
what's the weather going to be tomorrow in pittsburgh
what's the weather going to be tomorrow in pittsburgh.

today is
today is wednesday
today is wednesday tomorrow is thursday
today is wednesday tomorrow is thursday.

i would like
i would like to order a large
i would like to order a large pepperoni pizza
i would like to order a large pepperoni pizza.

is it going to be
is it going to be cloudy tomorrow.
```

> The first time you run each model, TensorRT will take a few minutes to optimize it.
> This optimized model is then cached to disk, so the next time you run the model it will load faster.

2.2.1 Live Microphone

To test the ASR on a mic, first list the audio devices in your system to get the audio device ID's:

```
$ scripts/list_audio_devices.sh

----------------------------------------------------
 Audio Input Devices
----------------------------------------------------
Input Device ID 1 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,0)' (inputs=16) (sample_rate=44100)
Input Device ID 2 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,1)' (inputs=16) (sample_rate=44100)
Input Device ID 3 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,2)' (inputs=16) (sample_rate=44100)
Input Device ID 4 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,3)' (inputs=16) (sample_rate=44100)
Input Device ID 5 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,4)' (inputs=16) (sample_rate=44100)
Input Device ID 6 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,5)' (inputs=16) (sample_rate=44100)
Input Device ID 7 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,6)' (inputs=16) (sample_rate=44100)
Input Device ID 8 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,7)' (inputs=16) (sample_rate=44100)
Input Device ID 9 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,8)' (inputs=16) (sample_rate=44100)
Input Device ID 10 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,9)' (inputs=16) (sample_rate=44100)
Input Device ID 11 - 'Logitech H570e Mono: USB Audio (hw:2,0)' (inputs=2) (sample_rate=44100)
Input Device ID 12 - 'Samson Meteor Mic: USB Audio (hw:3,0)' (inputs=2) (sample_rate=44100)
```

> If you don't see your audio device listed, exit and restart the container.
> USB devices should be attached *before* the container is started.

Then run the ASR example with the `--mic <DEVICE>` option, and specify either the device ID or name:

```
$ examples/asr.py --mic 11

hey
hey how are you guys
hey how are you guys.

# (Press Ctrl+C to exit)
```

2.3 ASR Classification

There are other ASR models included for command/keyword recognition (MatchboxNet) and voice activity detection (VAD MarbleNet). These models are smaller and faster, and classify chunks of audio as opposed to transcribing text.

2.3.1 Command/Keyword Recognition

The MatchboxNet model was trained on 12 keywords from the Google Speech Commands dataset:

```
# MatchboxNet classes
"yes",
"no",
"up",
"down",
"left",
"right",
"on",
"off",
"stop",
"go",
"unknown",
"silence"
```

You can run it through the same ASR example as above by specifying the `--model` **`matchboxnet`** argument:

```
$ examples/asr.py --model matchboxnet --wav data/audio/commands.wav

class 'unknown' (0.384)
class 'yes' (1.000)
class 'no' (1.000)
class 'up' (1.000)
class 'down' (1.000)
class 'left' (1.000)
class 'left' (1.000)
class 'right' (1.000)
class 'on' (1.000)
class 'off' (1.000)
class 'stop' (1.000)
class 'go' (1.000)
class 'go' (1.000)
class 'silence' (0.639)
class 'silence' (0.576)
```

The numbers printed on the right are the classification probabilities between 0 and 1.

2.4 Voice Activity Detection (VAD)

The voice activity model (VAD MarbleNet) is a binary model that outputs `background` or `speech`:

```
$ examples/asr.py --model vad_marblenet --wav data/audio/commands.wav

class 'background' (0.969)
class 'background' (0.984)
class 'background' (0.987)
class 'speech' (0.997)
```

```
class 'speech' (1.000)
class 'speech' (1.000)
class 'speech' (0.998)
class 'background' (0.987)
class 'speech' (1.000)
class 'speech' (1.000)
class 'speech' (1.000)
class 'background' (0.988)
class 'background' (0.784)
```

2.5 Natural Language Processing (NLP)

There are two samples included for NLP:

- **`examples/nlp.py`** (intent/slot, text classification, token classification)
- **`examples/nlp_qa.py`** (question/answering)
-

These each use a DistilBERT model which has been fined-tuned for it's particular task. For information about using the NLP APIs, please refer to **`jetson_voice/nlp.py`** and see the samples above.

2.5.1 Joint Intent/Slot Classification

Joint Intent and Slot classification is a task of classifying an Intent and detecting all relevant Slots (Entities) for this Intent in a query. For example, in the query: `What is the weather in Santa Clara tomorrow morning?`, we would like to classify the query as a `weather` Intent, and detect `Santa Clara` as a location slot and `tomorrow morning` as a date_time slot.
Intents and Slots names are usually task specific and defined as labels in the training data. The included intent/slot model was trained on the NLU-Evaluation-Data dataset - you can find the various intent and slot classes that it supports here. They are common things that you might ask a virtual assistant:

```
$ examples/nlp.py --model distilbert_intent

Enter intent_slot query, or Q to quit:

> What is the weather in Santa Clara tomorrow morning?

{'intent': 'weather_query',
 'score': 0.7165476,
 'slots': [{'score': 0.6280392, 'slot': 'place_name', 'text': 'Santa'},
           {'score': 0.61760694, 'slot': 'place_name', 'text': 'Clara'},
           {'score': 0.5439486, 'slot': 'date', 'text': 'tomorrow'},
           {'score': 0.4520608, 'slot': 'date', 'text': 'morning'}]}

> Set an alarm for 730am

{'intent': 'alarm_set',
 'score': 0.5713072,
 'slots': [{'score': 0.40017933, 'slot': 'time', 'text': '730am'}]}

> Turn up the volume

{'intent': 'audio_volume_up', 'score': 0.33523008, 'slots': []}

> What is my schedule for tomorrow?

{'intent': 'calendar_query',
 'score': 0.37434494,
 'slots': [{'score': 0.5732627, 'slot': 'date', 'text': 'tomorrow'}]}

> Order a pepperoni pizza from domino's

{'intent': 'takeaway_order',
 'score': 0.50629586,
 'slots': [{'score': 0.27558547, 'slot': 'food_type', 'text': 'pepperoni'},
           {'score': 0.2778827, 'slot': 'food_type', 'text': 'pizza'},
           {'score': 0.21785143, 'slot': 'business_name', 'text': 'dominos'}]}
```

```
> Where's the closest Starbucks?

{'intent': 'recommendation_locations',
 'score': 0.5438984,
 'slots': [{'score': 0.1604197, 'slot': 'place_name', 'text': 'Starbucks'}]}
```

2.6 Text Classification

In this text classification example, we'll use the included sentiment analysis model that was trained on the Standford Sentiment Treebank (SST-2) dataset. It will label queries as either positive or negative, along with their probability:

```
$ examples/nlp.py --model distilbert_sentiment

Enter text_classification query, or Q to quit:

> today was warm, sunny and beautiful out

{'class': 1, 'label': '1', 'score': 0.9985898}

> today was cold and rainy and not very nice

{'class': 0, 'label': '0', 'score': 0.99136007}
(class 0 is negative sentiment and class 1 is positive sentiment)
```

2.7 Token Classification

Whereas text classification classifies entire queries, token classification classifies individual tokens (or words). In this example, we'll be performing Named Entity Recognition (NER), which is the task of detecting and classifying key information (entities) in text. For example, in a sentence: `Mary lives in Santa Clara and works at NVIDIA`, we should detect that `Mary` is a person, `Santa Clara` is a location and `NVIDIA` is a company.
The included token classification model for NER was trained on the Groningen Meaning Bank (GMB) and supports the following annotations in IOB format (short for inside, outside, beginning)

- LOC = Geographical Entity
- ORG = Organization
- PER = Person
- GPE = Geopolitical Entity
- TIME = Time indicator
- MISC = Artifact, Event, or Natural Phenomenon
-

```
$ examples/nlp.py --model distilbert_ner

Enter token_classification query, or Q to quit:
> Mary lives in Santa Clara and works at NVIDIA

Mary[B-PER 0.989] lives in Santa[B-LOC 0.998] Clara[I-LOC 0.996] and works at NVIDIA[B-ORG 0.967]

> Lisa's favorite place to climb in the summer is El Capitan in Yosemite National Park in California,
U.S.

Lisa's[B-PER 0.995] favorite place to climb in the summer[B-TIME 0.996] is El[B-PER 0.577] Capitan[I-
PER 0.483]
in Yosemite[B-LOC 0.987] National[I-LOC 0.988] Park[I-LOC 0.98] in California[B-LOC 0.998], U.S[B-LOC
0.997].
```

2.8 Question/Answering

Question/Answering (QA) works by supplying a context paragraph which the model then queries the best answer from. The `nlp_qa.py` example allows you to select from several built-in context paragraphs (or supply your own) and to ask questions about these topics.

The QA model is flexible and doesn't need re-trained on different topics, as it was trained on the SQuAD question/answering dataset which allows it to extract answers from a variety of contexts. It essentially learns to identify the information most relevant to your query from the context passage, as opposed to learning the content itself.

```
$ examples/nlp_qa.py

Context:
The Amazon rainforest is a moist broadleaf forest that covers most of the Amazon basin of South
America.
This basin encompasses 7,000,000 square kilometres (2,700,000 sq mi), of which 5,500,000 square
kilometres
(2,100,000 sq mi) are covered by the rainforest. The majority of the forest is contained within Brazil,
with 60% of the rainforest, followed by Peru with 13%, and Colombia with 10%.

Enter a question, C to change context, P to print context, or Q to quit:

> How big is the Amazon?

Answer: 7,000,000 square kilometres
Score:  0.24993503093719482

> which country has the most?

Answer: Brazil
Score:  0.5964332222938538
To change the topic or create one of your own, enter C:
Enter a question, C to change context, P to print context, or Q to quit:
> C

Select from one of the following topics, or enter your own context paragraph:
   1. Amazon
   2. Geology
   3. Moon Landing
   4. Pi
   5. Super Bowl 55
> 3

Context:
The first manned Moon landing was Apollo 11 on July, 20 1969. The first human to step on the Moon was
astronaut Neil Armstrong followed second by Buzz Aldrin. They landed in the Sea of Tranquility with
their
lunar module the Eagle. They were on the lunar surface for 2.25 hours and collected 50 pounds of moon
rocks.

Enter a question, C to change context, P to print context, or Q to quit:

> Who was the first man on the moon?

Answer: Neil Armstrong
Score:  0.39105066657066345
```

2.9 Text-to-Speech (TTS)

The text-to-speech service uses an ensemble of two models: FastPitch to generate MEL spectrograms from text, and HiFiGAN as the vocoder (female English voice). For information about using the TTS APIs, please refer to **jetson_voice/tts.py** and see **examples/tts.py**
The **examples/tts.py** app can output the audio to a speaker, wav file, or sequence of wav files.

Run it with **--list-devices** to get a list of your audio devices.

```
$ examples/tts.py --output-device 11 --output-wav data/audio/tts_test

> The weather tomorrow is forecast to be warm and sunny with a high of 83 degrees.

Run 0 -- Time to first audio: 1.820s. Generated 5.36s of audio. RTFx=2.95.
Run 1 -- Time to first audio: 0.232s. Generated 5.36s of audio. RTFx=23.15.
Run 2 -- Time to first audio: 0.230s. Generated 5.36s of audio. RTFx=23.31.
Run 3 -- Time to first audio: 0.231s. Generated 5.36s of audio. RTFx=23.25.
Run 4 -- Time to first audio: 0.230s. Generated 5.36s of audio. RTFx=23.36.
Run 5 -- Time to first audio: 0.230s. Generated 5.36s of audio. RTFx=23.35.

Wrote audio to data/audio/tts_test/0.wav

Enter text, or Q to quit:
> Sally sells seashells by the seashore.

Run 0 -- Time to first audio: 0.316s. Generated 2.73s of audio. RTFx=8.63.
Run 1 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.61.
Run 2 -- Time to first audio: 0.127s. Generated 2.73s of audio. RTFx=21.51.
Run 3 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.68.
Run 4 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.68.
Run 5 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.61.

Wrote audio to data/audio/tts_test/1.wav
```

2.9.1 TTS Audio Samples

- **Weather forecast (wav)**
- **Sally sells seashells (wav)**
-

2.10 Tests

There is an automated test suite included that will verify all of the models are working properly. You can run it with the **tests/run_tests.py** script:

```
$ tests/run_tests.py

-------------------------------------------------
 TEST SUMMARY
-------------------------------------------------
test_asr.py (quartznet)                  PASSED
test_asr.py (quartznet_greedy)           PASSED
test_asr.py (matchboxnet)                PASSED
test_asr.py (vad_marblenet)              PASSED
test_nlp.py (distilbert_qa_128)          PASSED
test_nlp.py (distilbert_qa_384)          PASSED
test_nlp.py (distilbert_intent)          PASSED
test_nlp.py (distilbert_sentiment)       PASSED
test_nlp.py (distilbert_ner)             PASSED
test_tts.py (fastpitch_hifigan)          PASSED

passed 10 of 10 tests
saved logs to data/tests/logs/20210610_1512
```

The logs of the individual tests are printed to the screen and saved to a timestamped directory.

Table of Contents