Session 13 - AI for Sound

Submit Assignment

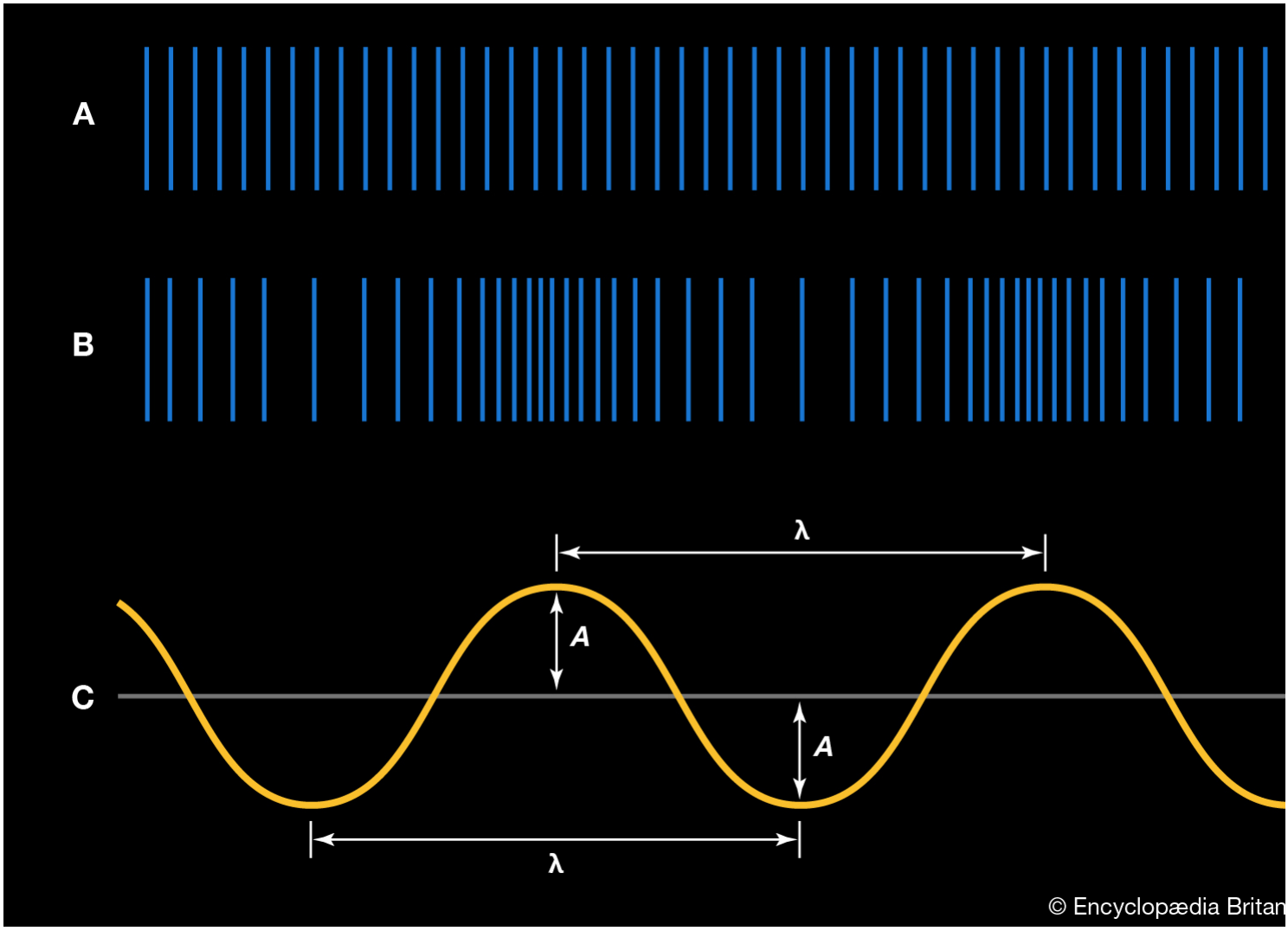
Due Saturday by 11:59pm **Points** 1,000 **Submitting** a text entry box or a website url
Available Nov 7 at 9am - Nov 14 at 11:59pm 8 days

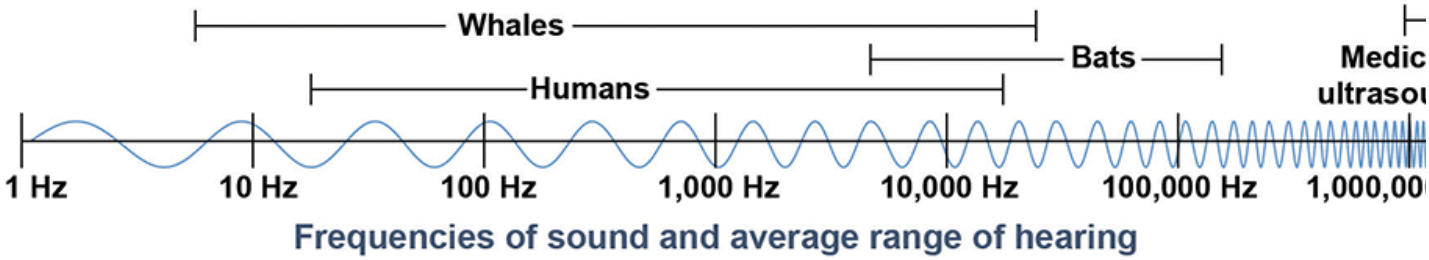
Session 13 - AI for Audio

Sound

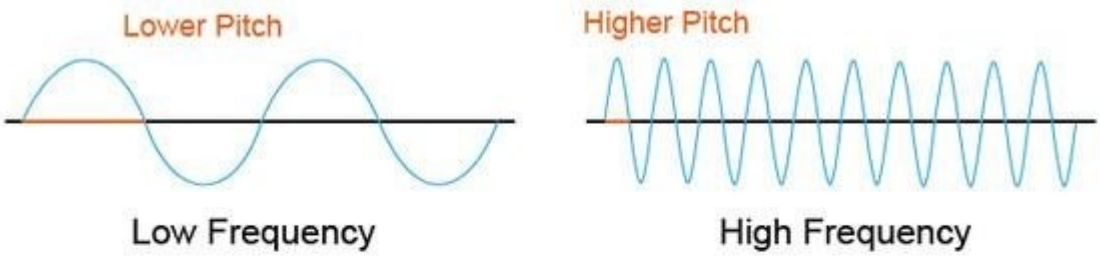
Sound:

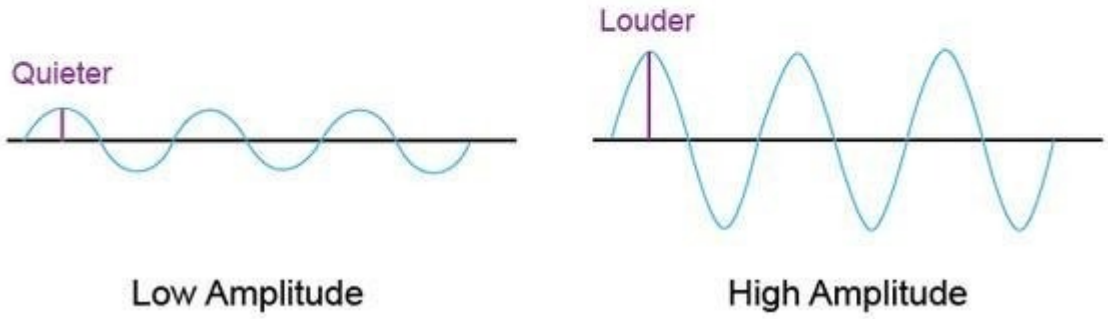
- Produced by the vibration of an object
- Vibrations determine the oscillation of air molecules
- Alternations of air pressure cause a wave





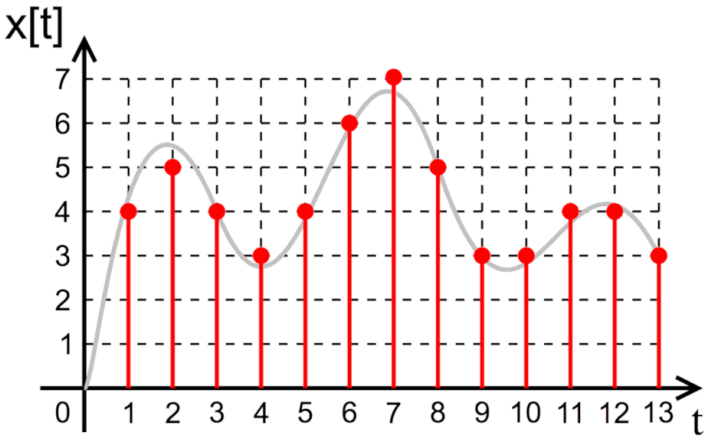
Frequencies of sound and average range of hearing



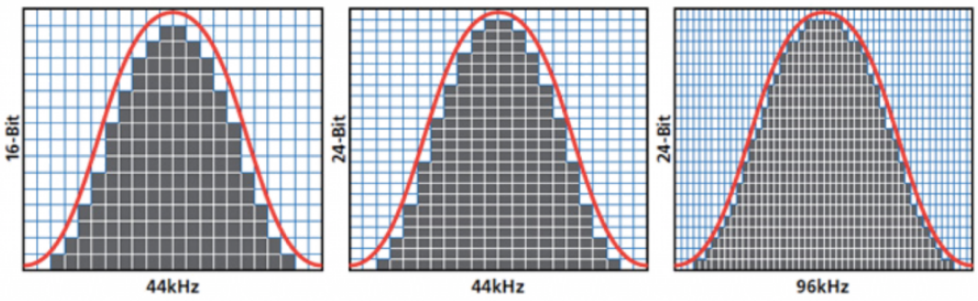


Analog-Digital Conversion (ADC):

- Signal sampled at uniform time intervals
- Amplitude quantized with a limited number of bits



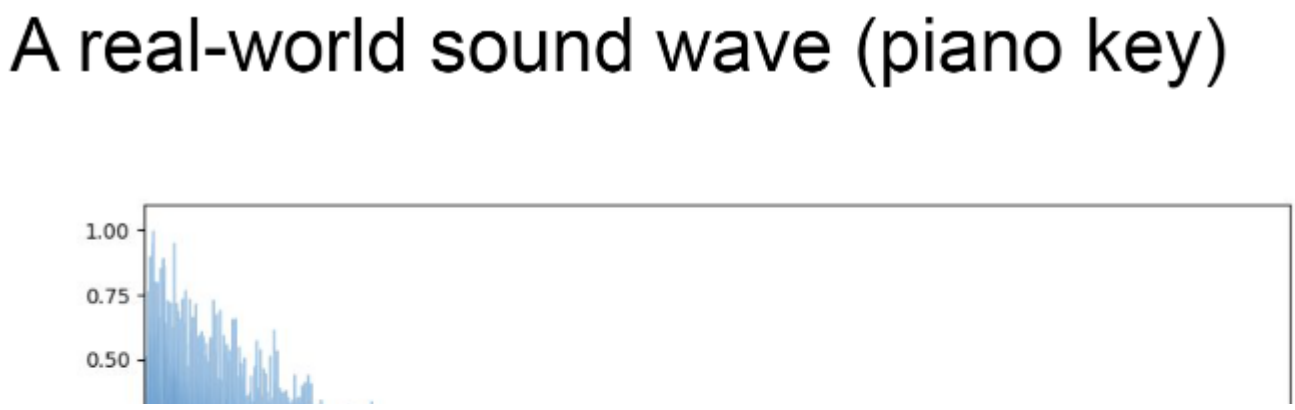
- Sample Rate = 44,100 Hz
- Bit depth = 16-24 bits/channel

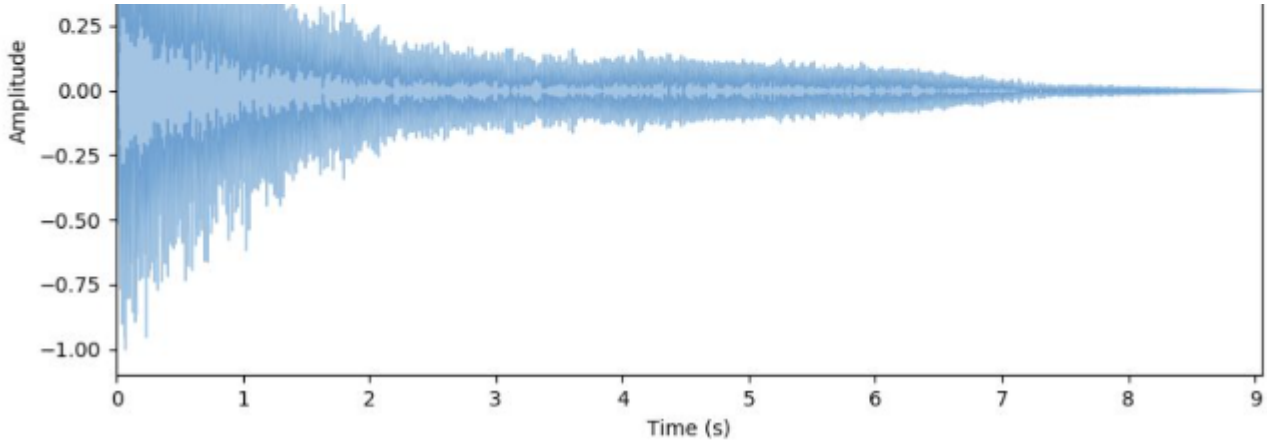


Music & Fourier Transform & Mel Spectogram

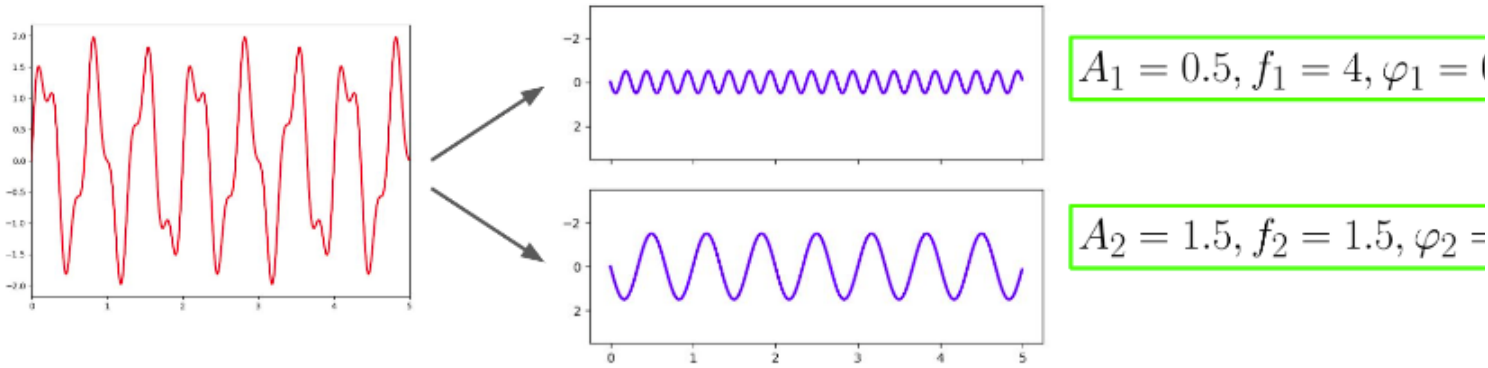
Hear Piano Note - Middle C **(https://youtu.be/FtqgqYRDTDg)**

(https://youtu.be/FtqgqYRDTDg)

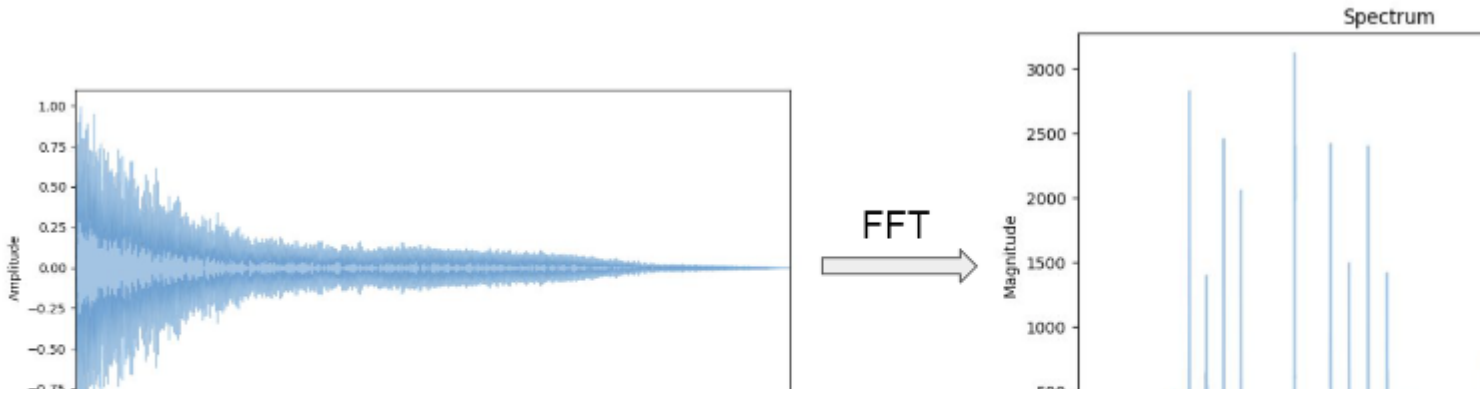


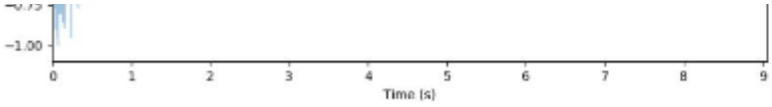


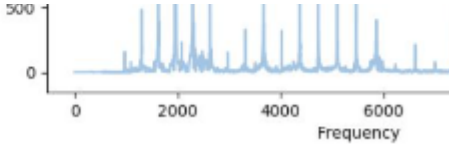
Fourier Transform - Decompose complex periodic sound into a sum of sine waves oscillating at different frequencies



$$s = A_1 \sin(2\pi f_1 t + \varphi_1) + A_2 \sin(2\pi f_2 t + \varphi_2)$$





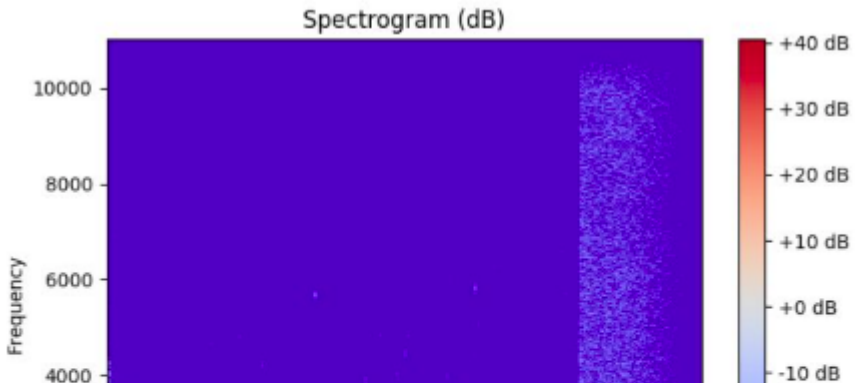


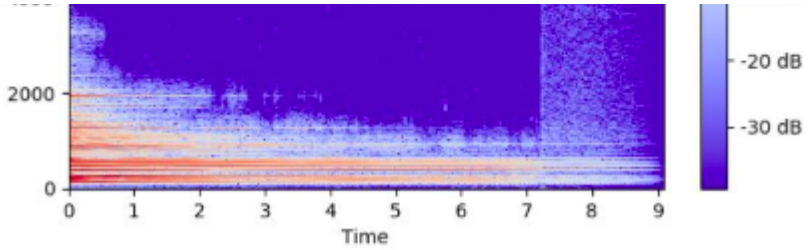
From the time domain to frequency domain

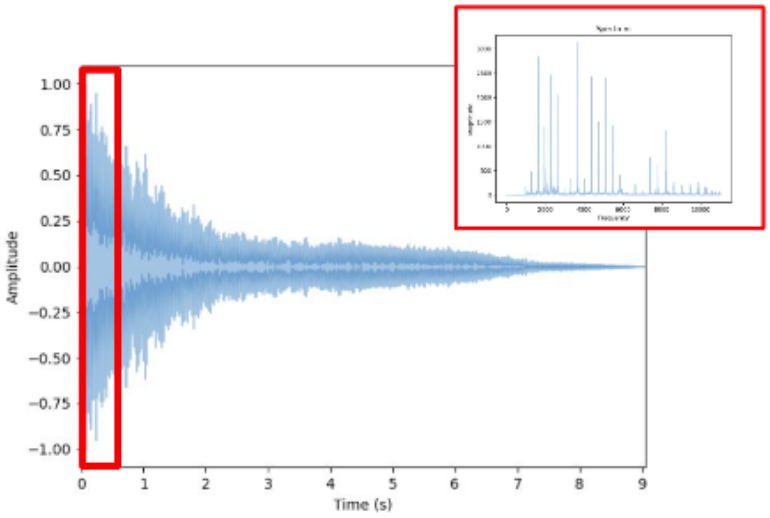
But we have lost the time information!

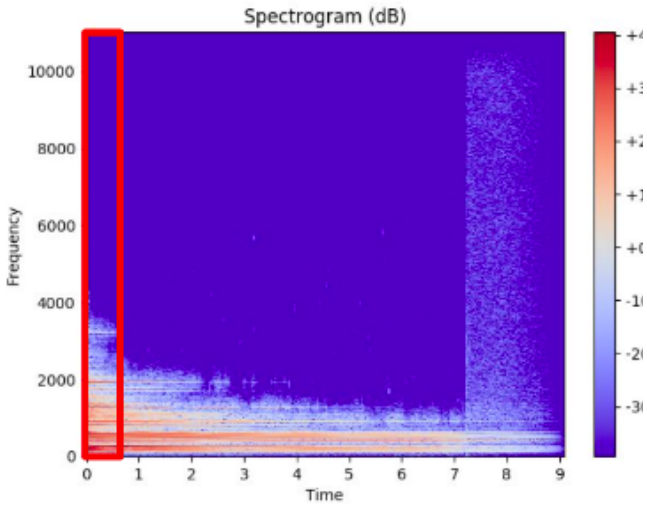
Short-Time Fourier Transform

- Computes several FFTs at different intervals
- Preserves time information
- Fixed frame size (e.g. 248 samples)
- Gives a *spectrogram* (time + frequency + magnitude)

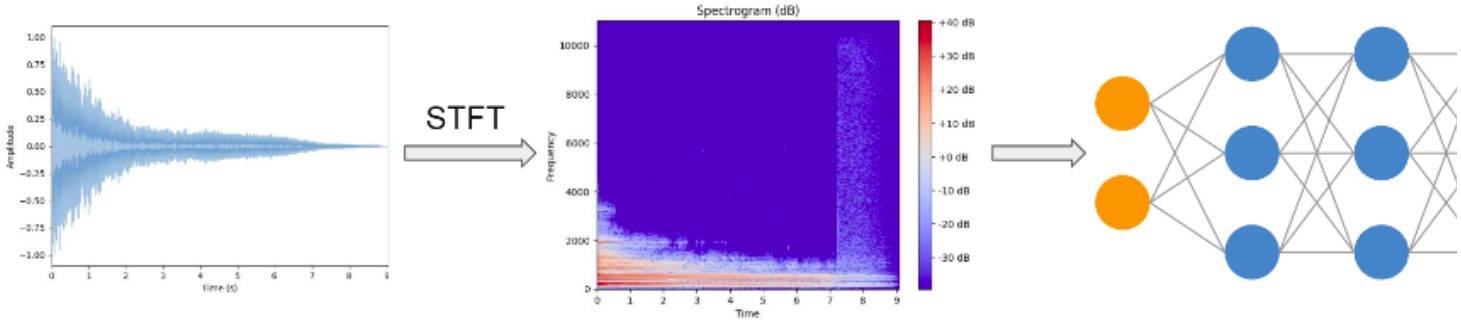






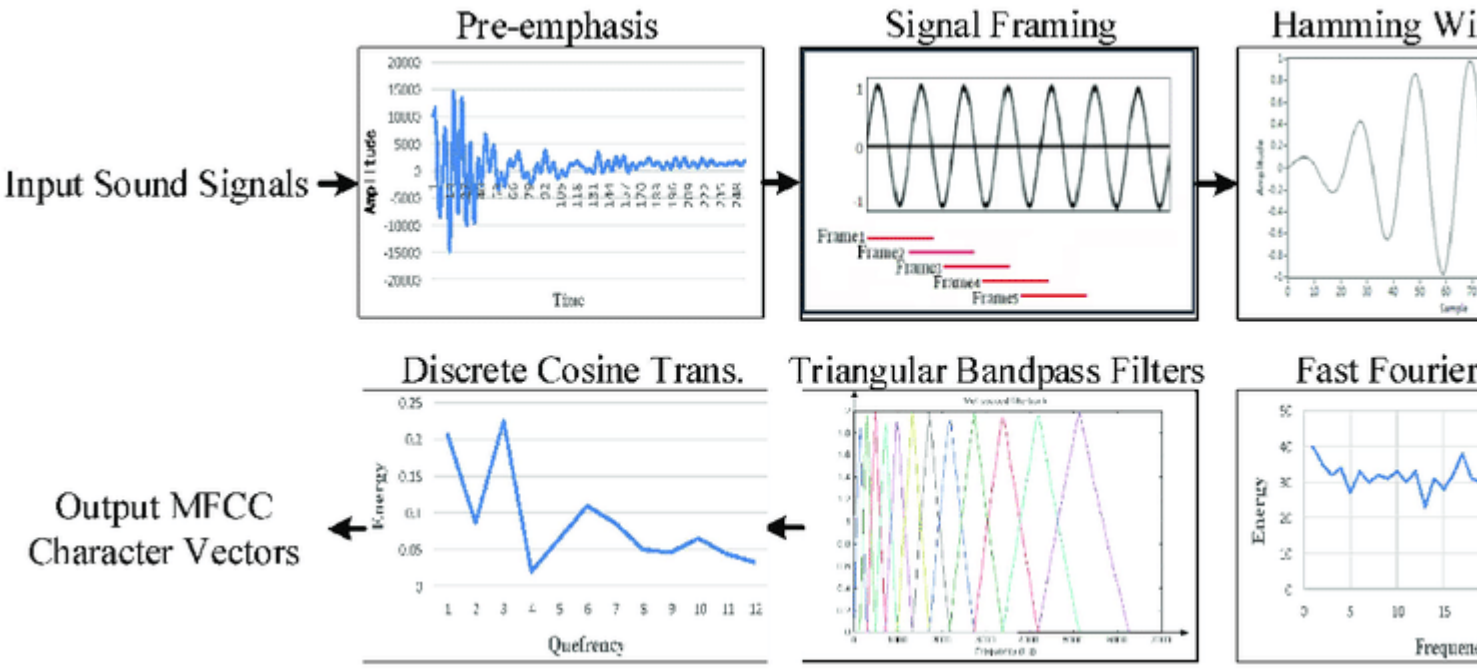


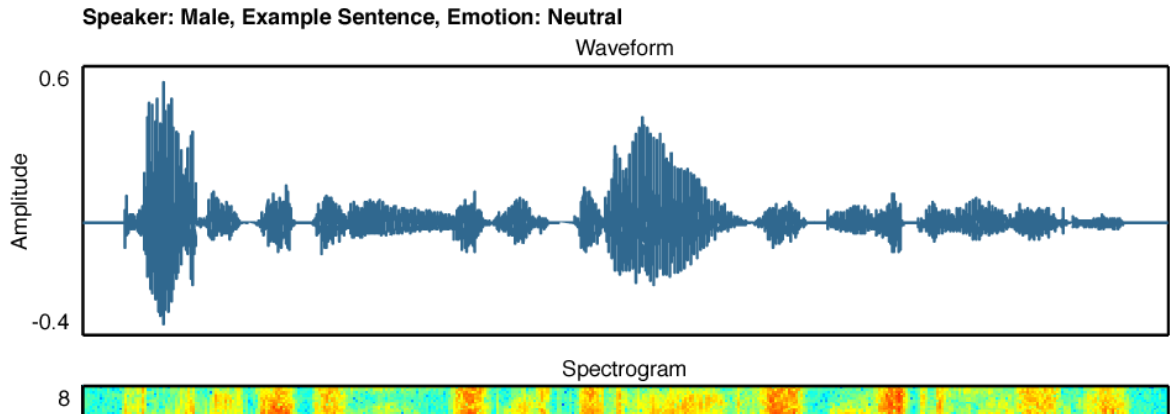
DL Preprocessing Pipeline for Audio Data



Mel Frequency Cepstral Coefficients (MFCCs)

- Capture timbral/textural aspect of sound
- Frequency domain feature
- Approximate human auditory system
- 13 to 40 coefficients
- Calculated at each frame





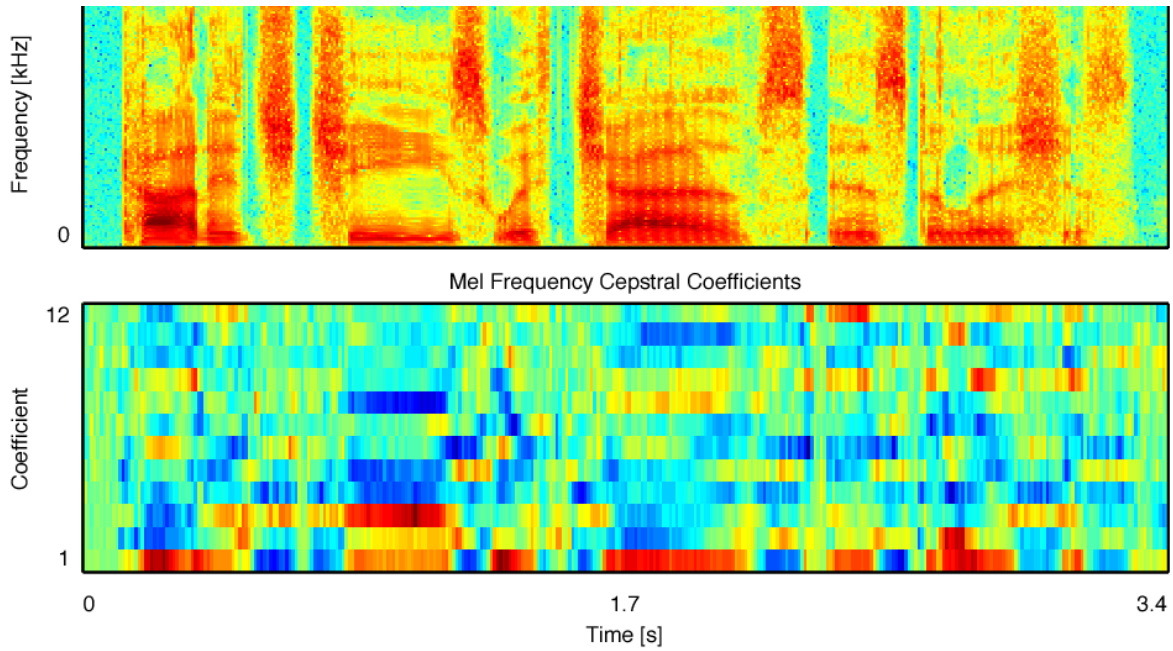
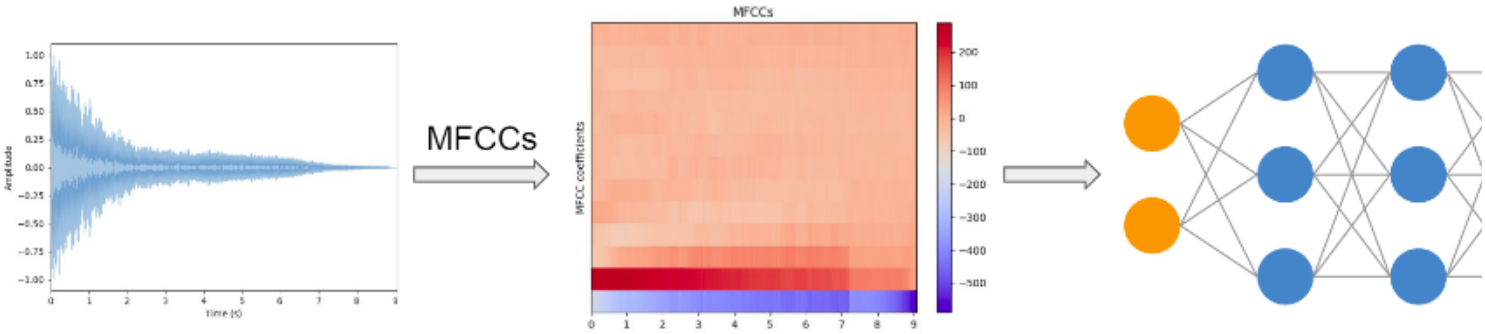


Figure 3.2: In the top the original waveform of the audio signal is repre-

- Speech Recognition
- Music Genre Classification
- Music instrument classification



Basic Audio Processing and a Simple Model

COLAB 1 (https://colab.research.google.com/drive/1z6la_zT9HbAd6zxpafDVzd1Q0klMGaA4?usp=sharing)

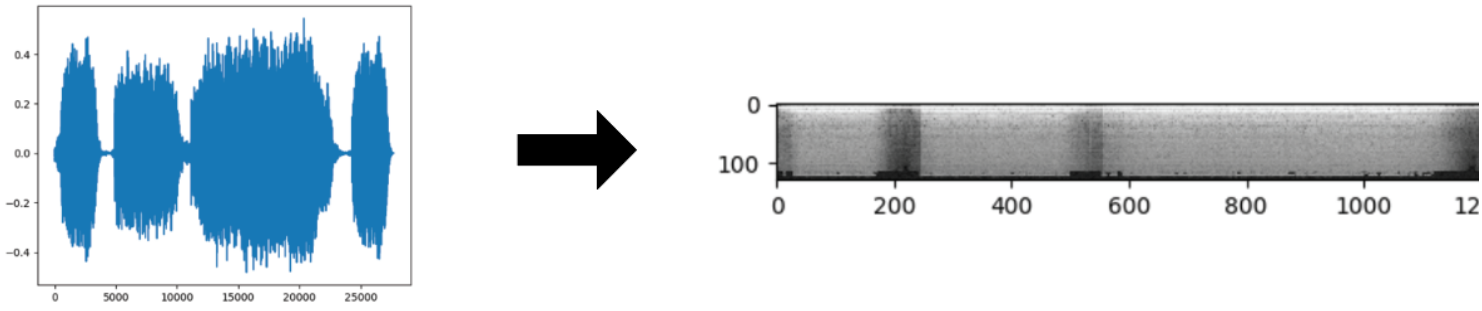
Building an end-to-end SPeech Recognition model in PyTorch

Ref (https://www.assemblyai.com/blog/end-to-end-speech-recognition-pytorch)

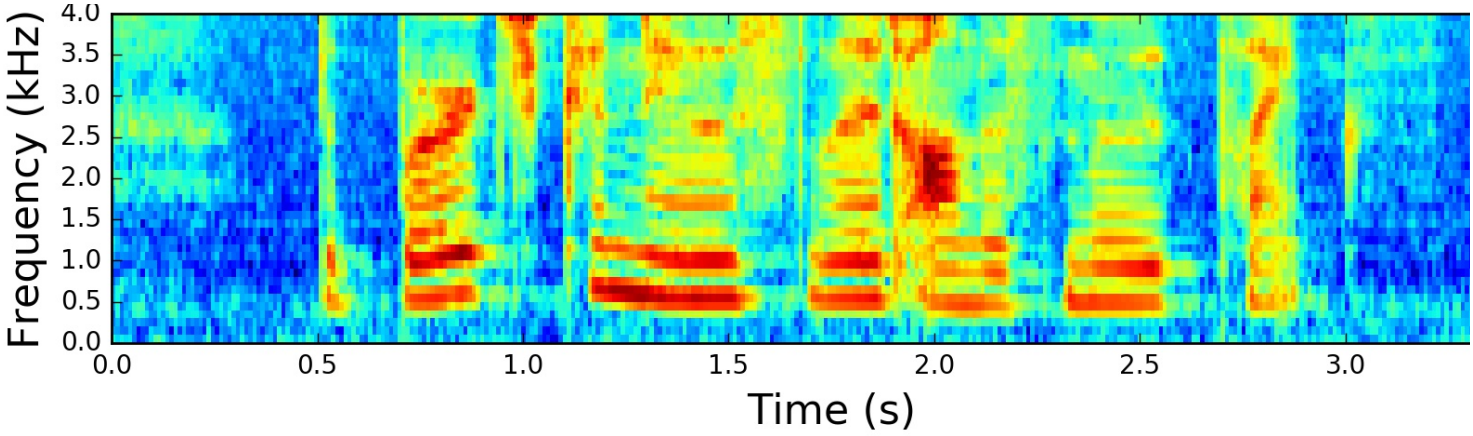
Let's walk through how one would build their own end-to-end speech recognition model in PyTorch. The model we'll build is inspired by Deep Speech 2 (Baidu's second revision of their now-famous model) with some personal improvements to the architecture. The output of the model will be a probability matrix of characters, and we'll use that probability matrix to decode the most likely characters spoken from the audio

Preparing the data pipeline

Data is one of the most important aspects of speech recognition. We'll take raw audio waves and transform them into Mel Spectrograms.



You can read more on the details about how that transformation looks from this excellent post **here** (https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html) . For this post, you can just think of a Mel Spectrogram as essentially a picture of sound.



For handling the audio data, we are going to use an extremely useful utility called `torchaudio` which is a library built by the PyTorch team specifically for audio data. We'll be training on a subset of **LibriSpeech (http://www.openslr.org/12/)** , which is a corpus of read English speech data derived from audiobooks, comprising 100 hours of transcribed audio data. You can easily download this dataset using `torchaudio`:

```
import torchaudio

train_dataset = torchaudio.datasets.LIBRISPEECH("./", url="train-clean-100", download=True)
test_dataset = torchaudio.datasets.LIBRISPEECH("./", url="test-clean", download=True)
```

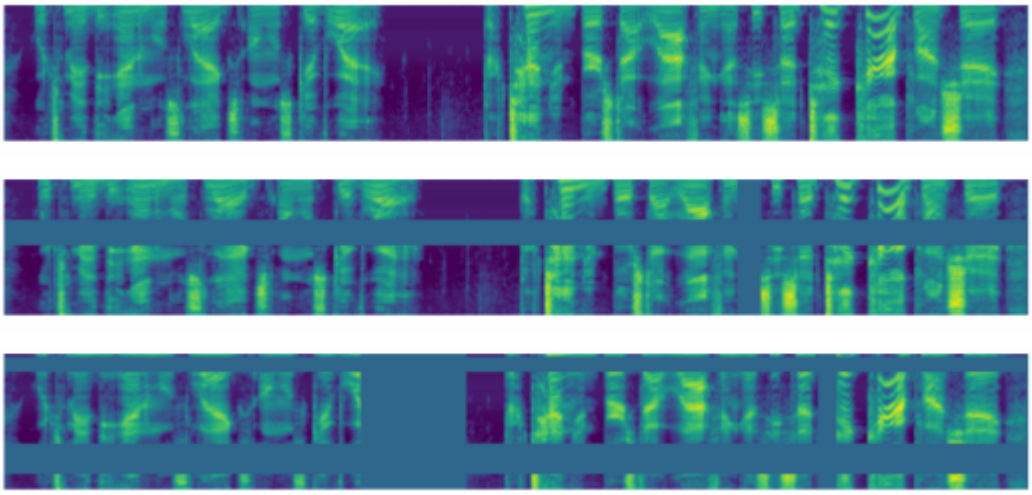
Each sample of the dataset contains the waveform, sample rate of audio, the utterance/label, and more metadata on the sample. You can view what each sample looks like from the source code **here (https://github.com/pytorch/audio/blob/master/torchaudio/datasets/librispeech.py#L40)** .

Data Augmentation - SpecAugment

Data augmentation is a technique used to artificially increase the diversity of your dataset in order to increase your dataset size. This strategy is especially helpful when data is scarce or if your model is overfitting. For speech recognition, you can do the standard augmentation techniques, like changing the pitch, speed, injecting noise, and adding reverb to your audio data.

We found Spectrogram Augmentation (SpecAugment), to be a much simpler and more effective

We found Spectrogram Augmentation (SpecAugment), to be a much simpler and more effective approach. SpecAugment, was first introduced in the paper **SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition (https://arxiv.org/abs/1904.08779)**, in which the authors found that simply cutting out random blocks of consecutive time and frequency dimensions improved the models generalization abilities significantly!



In PyTorch, you can use the `torchaudio` function `FrequencyMasking` to mask out the frequency dimension, and `TimeMasking` for the time dimension.

```
torchaudio.transforms.FrequencyMasking()
torchaudio.transforms.TimeMasking()
```

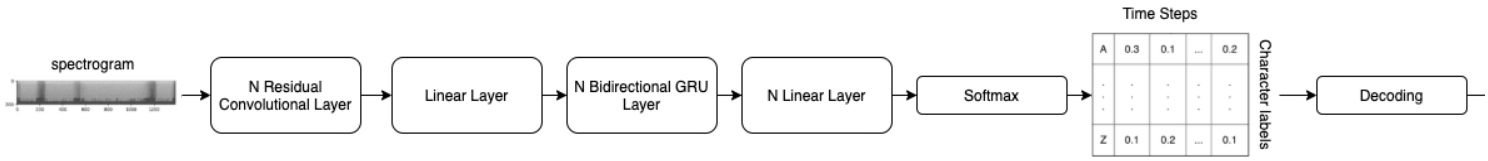
Now that we have the data, we'll need to transform the audio into Mel Spectrograms, and map the character labels for each audio sample into integer labels:

CODE (https://colab.research.google.com/drive/1Z-4MiFimY9JPWk93V0MwblXu2iS8Lzp0?usp=sharing)

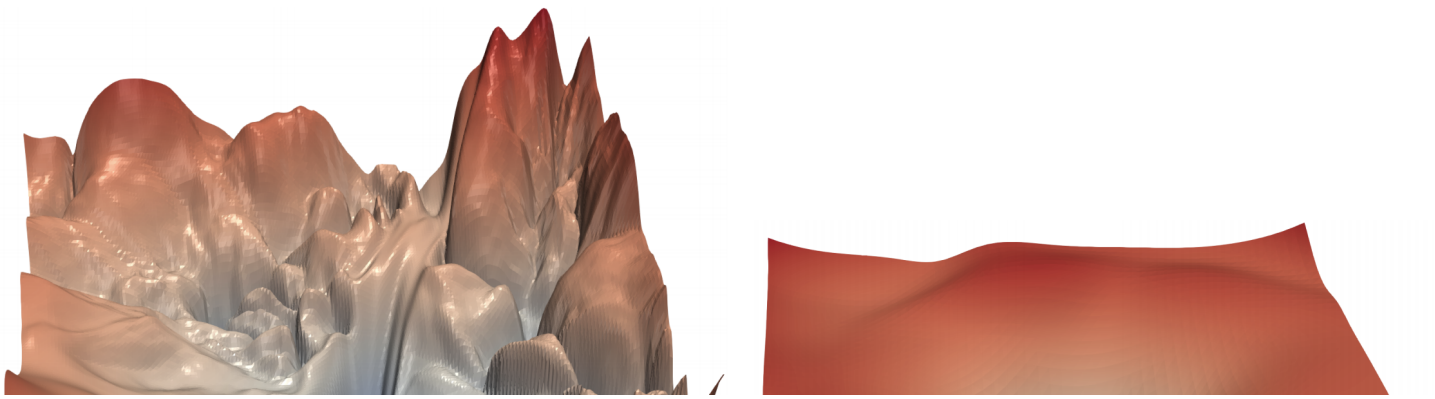
Define the Model - Deep Speech 2 (but better)

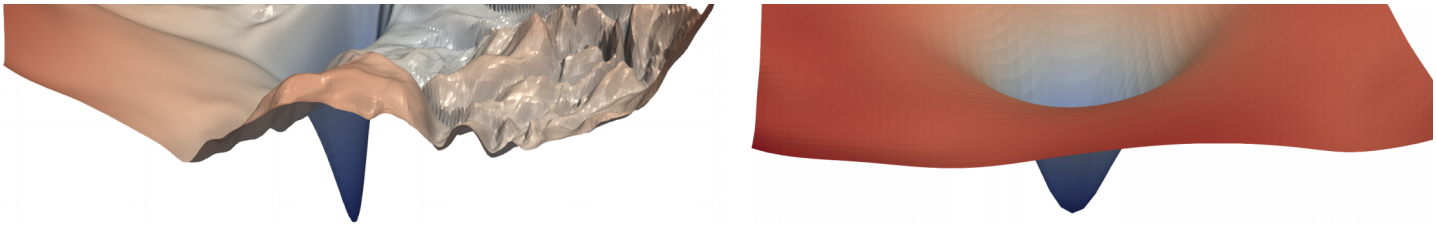
Our model will be similar to the Deep Speech 2 architecture. The model will have two main neural network modules - N layers of Residual Convolutional Neural Networks (ResCNN) to learn the relevant audio features, and a set of Bidirectional Recurrent Neural Networks (BiRNN) to leverage the learned

ResCNN audio features. The model is topped off with a fully connected layer used to classify characters per time step.



Convolutional Neural Networks (CNN) are great at extracting abstract features, and we'll apply the same feature extraction power to audio spectrograms. Instead of just vanilla CNN layers, we choose to use Residual CNN layers. Residual connections (AKA skip connections) were first introduced in the paper **Deep Residual Learning for Image Recognition (https://arxiv.org/abs/1512.03385)**, where the author found that you can build really deep networks with good accuracy gains if you add these connections to your CNN's. Adding these Residual connections also helps the model learn faster and generalize better. The paper **Visualizing the Loss Landscape of Neural Nets (https://arxiv.org/abs/1712.09913)** shows that networks with residual connections have a “flatter” loss surface, making it easier for models to navigate the loss landscape and find a lower and more generalizable minima.





(a) without skip connections (b) with skip connections

Recurrent Neural Networks (RNN) are naturally great at sequence modeling problems. RNN's processes the audio features step by step, making a prediction for each frame while using context from previous frames. We use BiRNN's because we want the context of not only the frame before each step, but the frames after it as well. This can help the model make better predictions, as each frame in the audio will have more information before making a prediction. We use Gated Recurrent Unit (GRU's) variant of RNN's as it needs less computational resources than LSTM's, and works just as well in some cases.

The model outputs a probability matrix for characters which we'll use to feed into our decoder to extract what the model believes are the highest probability characters that were spoken.

CODE **(https://colab.research.google.com/drive/1Z-4MiFimY9JPWk93V0MwbIXu2iS8Lzp0?usp=sharing)**

Picking the Right Optimizer and Scheduler - AdamW with Super Convergence

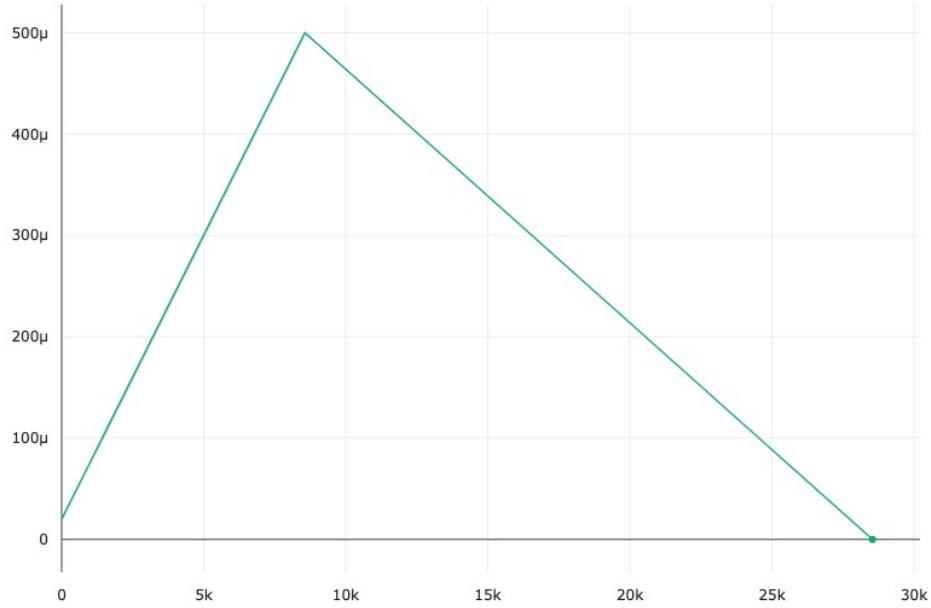
The optimizer and learning rate schedule plays a very important role in getting our model to converge to the best point. Picking the right optimizer and scheduler can also save you compute time, and help your model generalize better to real-world use cases. For our model, we'll be using **AdamW** with the **One**

Cycle Learning Rate Scheduler. **Adam** is a widely used optimizer that helps your model converge more quickly, therefore, saving compute time, but has been notorious for not generalizing as well as **Stochastic Gradient Descent** AKA **SGD**.

AdamW was first introduced in **Decoupled Weight Decay Regularization (https://arxiv.org/abs/1711.05101)**, and is considered a “fix” to **Adam**. The paper pointed out that the original **Adam** algorithm has a wrong implementation of weight decay, which **AdamW** attempts to fix. This fix helps with **Adam**'s generalization problem.

This fix helps with **Adam**'s generalization problem.

The **One Cycle Learning Rate Scheduler** was first introduced in the paper **Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates (https://arxiv.org/abs/1708.07120)**. This paper shows that you can train neural networks an order of magnitude faster, while keeping their generalizable abilities, using a simple trick. You start with a low learning rate, which warms up to a large maximum learning rate, then decays linearly to the same point of where you originally started.

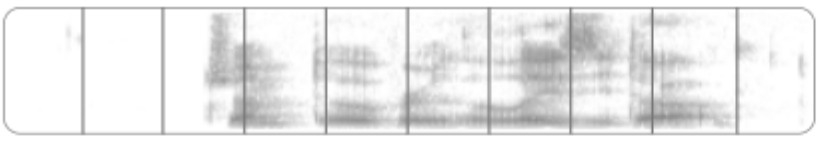


Because the maximum learning rate is magnitudes higher than the lowest, you also gain some regularization benefits which helps your model generalize better if you have a smaller set of data.

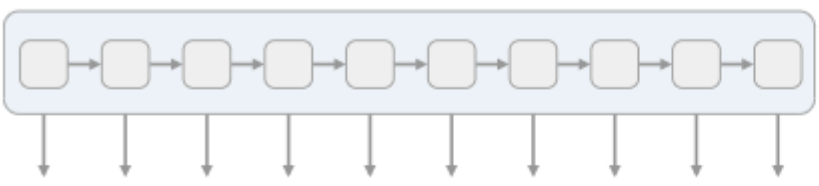
With PyTorch, these two methods are already part of the package. Optimizer **CODE (https://colab.research.google.com/drive/1Z-4MiFimY9JPWk93V0MwblXu2iS8Lzp0?usp=sharing)**

The CTC Loss Function - Aligning Audio to Transcript

Our model will be trained to predict the probability distribution of all characters in the alphabet for each frame (ie, timestep) in the spectrogram we feed into the model.



We start with an input sequenc like a spectrogram of audio.



The input is fed into an RNN, for example.

Traditional speech recognition models would require you to align the transcript text to the audio before training, and the model would be trained to predict specific labels at specific frames.

The innovation of the CTC loss function is that it allows us to skip this step. Our model will learn to align the transcript itself during training. The key to this is the “blank” label introduced by CTC, which gives the model the ability to say that a certain audio frame did not produce a character. You can see a more detailed explanation of CTC and how it works from **this excellent post (https://distill.pub/2017/ctc/)** .

The CTC loss function is also built into PyTorch.

criterion = nn.CTCLoss(blank=28).to(device)

Evaluating Your Speech Model

When Evaluating your speech recognition model, the industry standard is using the Word Error Rate (WER) as the metric. The Word Error Rate does exactly what it says - it takes the transcription your model outputs, and the true transcription, and measures the error between them. You can see how that's implemented **here (https://colab.research.google.com/drive/1IPpwx4rX32rqHKpLz7dc8sOKspUa-YKO)** . Another useful metric is called the Character Error Rate (CER). The CER measures the error of the characters between the model's output and the true labels. These metrics are helpful to measure how well your model performs.

For this tutorial, we'll use a "greedy" decoding method to process our model's output into characters that can be combined to create the transcript. A "greedy" decoder takes in the model output, which is a softmax probability matrix of characters, and for each time step (spectrogram frame), it chooses the label with the highest probability. If the label is a blank label, we remove it from the final transcript.

How to Improve Accuracy

Speech Recognition Requires a ton of data and a ton of compute resources. The example laid out is trained on a subset of LibriSpeech (100 hours of audio) and a single GPU. To get state of the art results you'll need to do distributed training on thousands of hours of data, on tens of GPU's spread out across many machines.

Another way to get a big accuracy improvement is to decode the CTC probability matrix using a Language Model and the CTC beam search algorithm. CTC type models are very dependent on this

decoding process to get good results. Luckily there is a handy **open source library (https://github.com/parlance/ctcdecode)** that allows you to do that.

This tutorial was made to be more accessible so it's a relatively small model (23 million Parameters) compared to something like BERT (340 million Parameters). It seems to be the larger you can get your network, the better it performs, although there are diminishing returns. A larger model equating to better performance is not always the case though, as proven by OpenAI's research **Deep Double Descent (https://openai.com/blog/deep-double-descent/)**

(https://openai.com/assets/deep-double-descent/).

This model has 3 residual CNN layers and 5 Bidirectional GRU layers which should allow you to train a reasonable batch size on a single GPU with at least 11GB of memory. You can tweak some of the hyperparameters in the main function to reduce or increase the model size for your use case and compute availability.

Assignment

- Train both the models. Try to move the one you like to Lambda, and as usual, your submit your Lambda link.