Speech-to-Text Summary

April 17, 2025

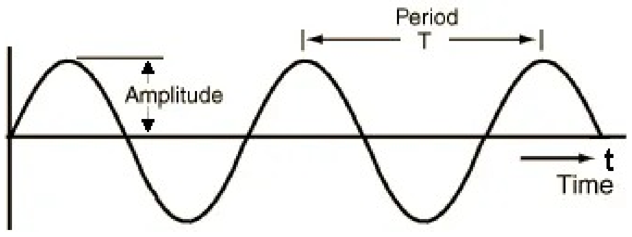
Contents

This is a summary of all the bibliography I have read for this project. I've copied and pasted the original texts and sometimes small changes were made.

1 What is sound?

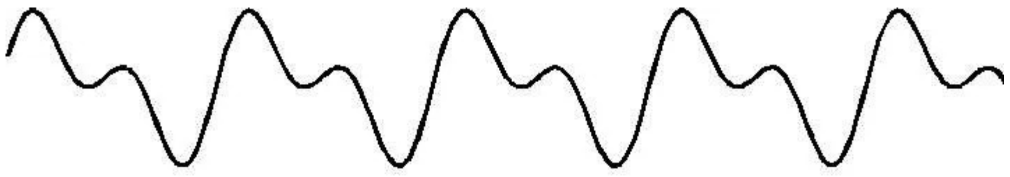
We all remember from school that a sound signal is produced by variations in air pressure. We can measure the intensity of the pressure variations and plot those measurements over time.

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.



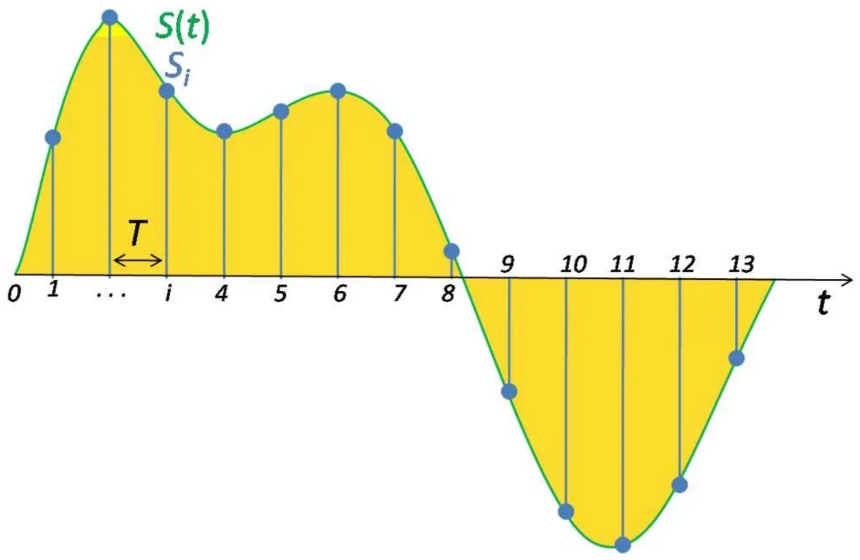
The time taken for the signal to complete one full wave is the period. The number of waves made by the signal in one second is called the frequency. The frequency is the reciprocal of the period. The unit of frequency is Hertz.

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.



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.



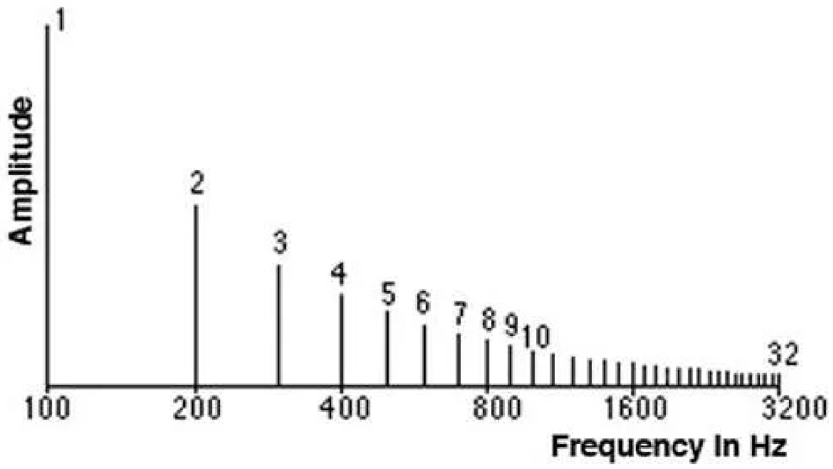
Each such measurement is called a sample, and the sample rate is the number of samples per second. For instance, a common sampling rate is about 44,100 samples per second. That means that a 10-second music clip would have 441,000 samples!

3 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.



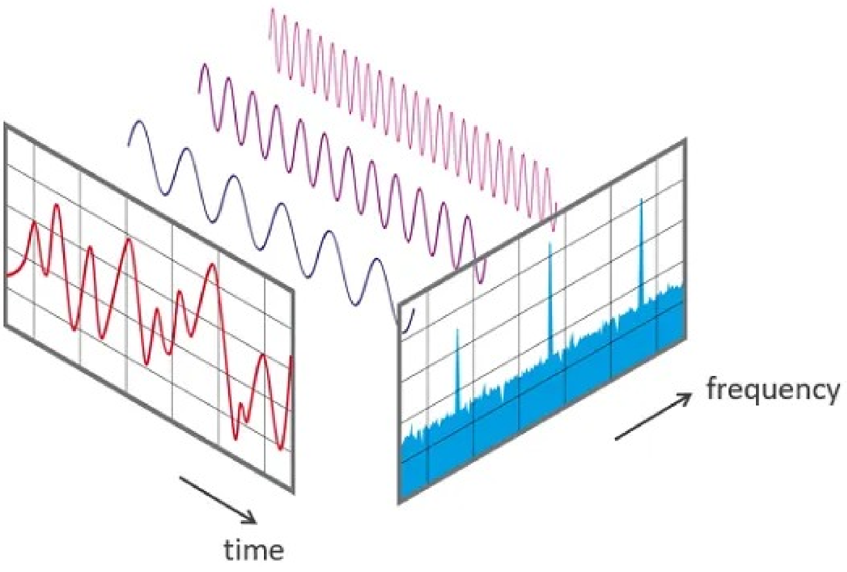
The lowest frequency in a signal called the fundamental frequency. Frequencies that are whole number multiples of the fundamental frequency are known as harmonics.

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

4 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.



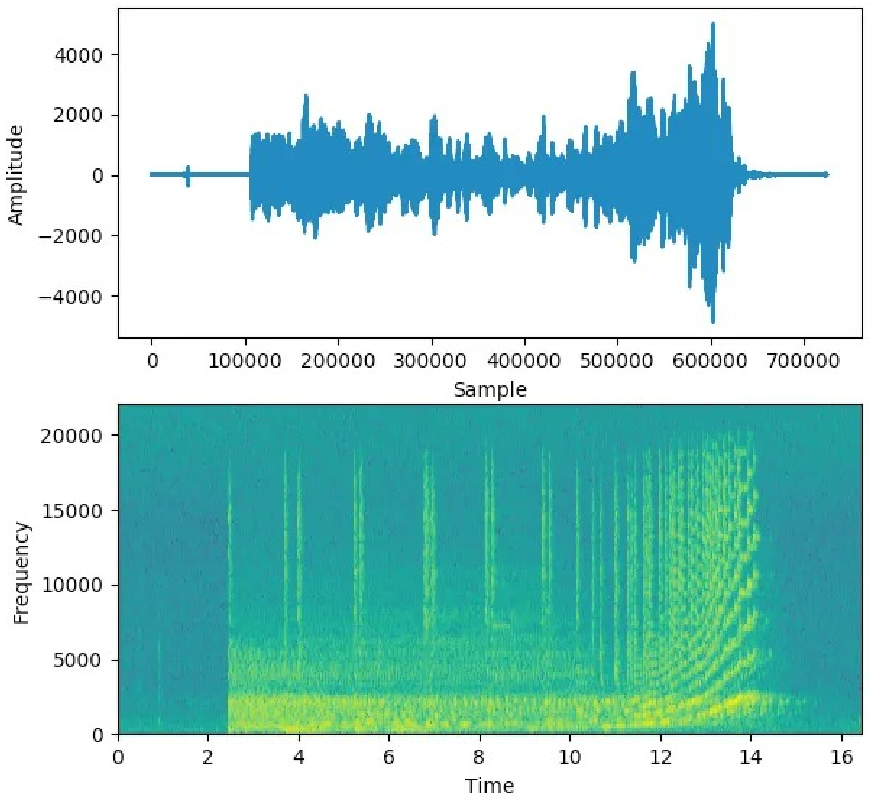
5 Spectrograms

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.



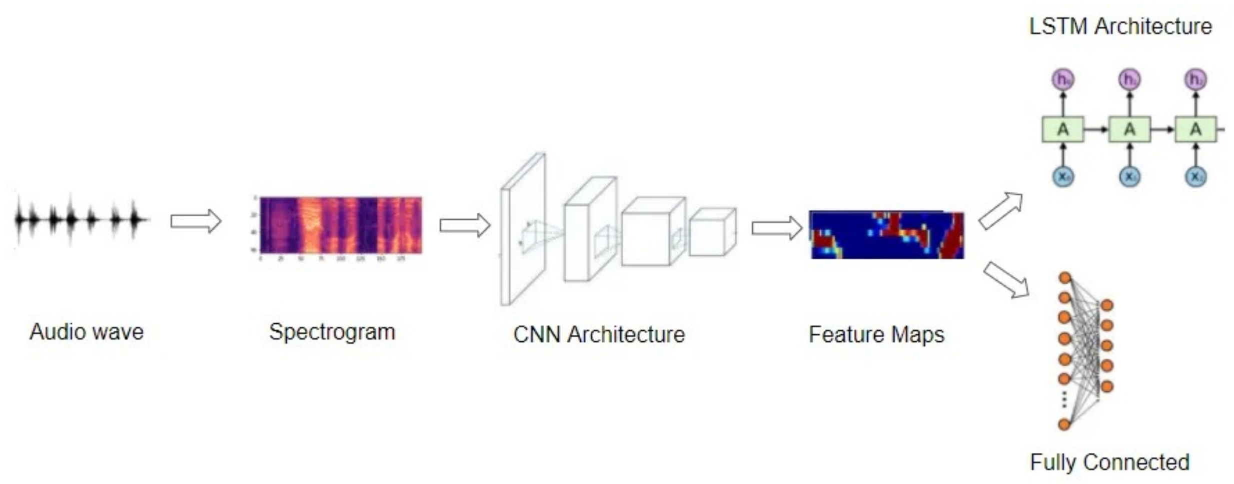
The second picture is the Spectrogram and displays the signal in the Frequency domain.

Spectrograms are produced using Fourier Transforms to decompose any signal into its constituent frequencies.

A Spectrogram chops up the duration of the sound signal into smaller time segments and then applies the Fourier Transform to each segment, to determine the frequencies contained in that segment. It then combines the Fourier Transforms for all those segments into a single plot.

6 Audio Deep Learning Models

A typical pipeline used by audio deep learning models



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

For a Speech-to-Text problem, you could pass it through some RNN layers to extract text sentences from this encoded representation.

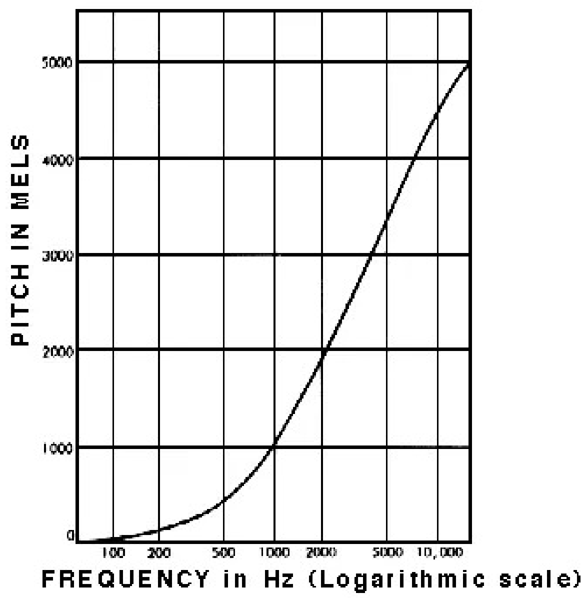
7 How do humans hear frequencies?

The way we hear frequencies in sound is known as pitch. It is a subjective impression of the frequency. So a high-pitched sound has a higher frequency than a low-pitched sound. Humans do not perceive frequencies linearly. We are more sensitive to differences between lower frequencies than higher frequencies.

We hear them on a logarithmic scale rather than a linear scale.

7.0.1 Mel Scale

The Mel Scale was developed to take this into account by conducting experiments with a large number of listeners. It is a scale of pitches, such that each unit is judged by listeners to be equal in pitch distance from the next.

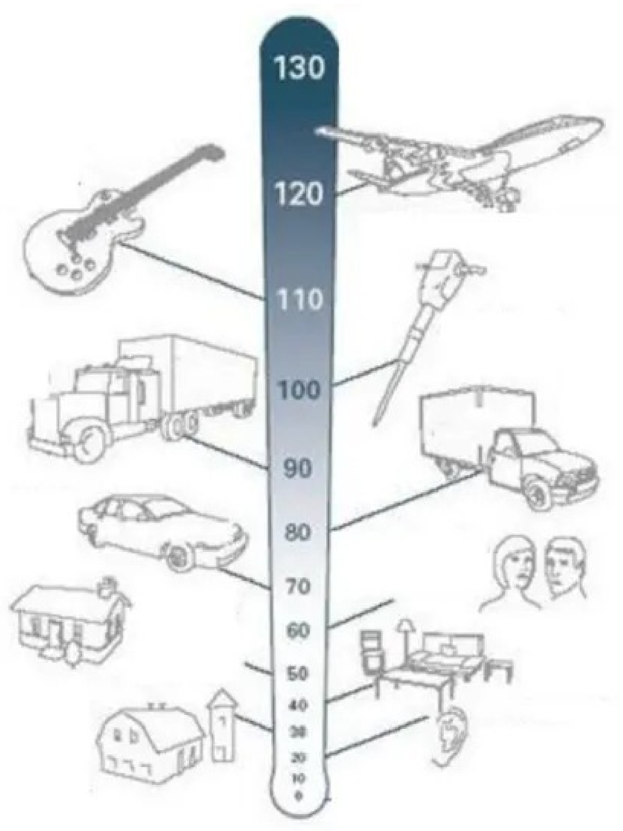


8 How do humans hear amplitudes?

The human perception of the amplitude of a sound is its loudness. And similar to frequency, we hear loudness logarithmically rather than linearly. We account for this using the Decibel scale.

8.1 Decibel Scale

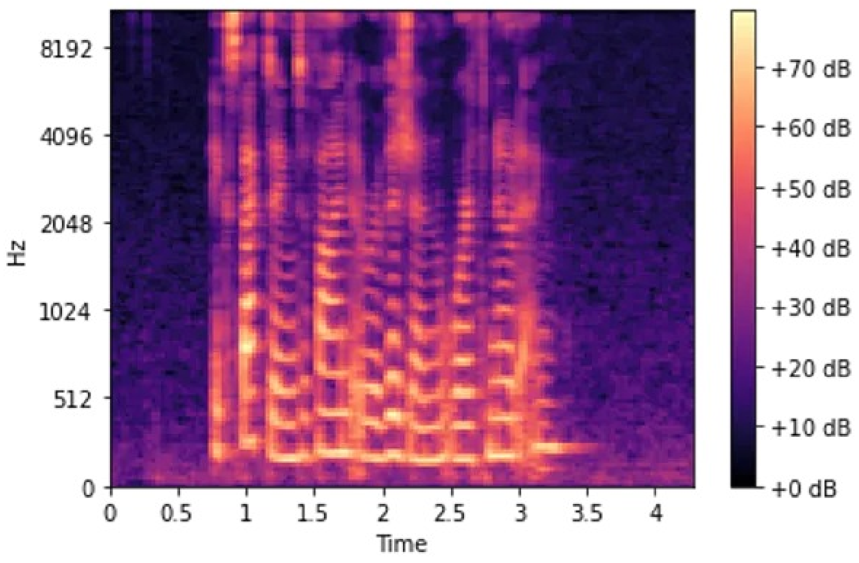
On this scale, 0 dB is total silence. From there, measurement units increase exponentially. 10 dB is 10 times louder than 0 dB, 20 dB is 100 times louder and 30 dB is 1000 times louder. On this scale, a sound above 100 dB starts to become unbearably loud.



9 Mel Spectrograms

A Mel Spectrogram makes two important changes relative to a regular Spectrogram that plots Frequency vs Time.

- It uses the Mel Scale instead of Frequency on the y-axis.
- It uses the Decibel Scale instead of Amplitude to indicate colors.



10 Spectrograms Optimization with Hyper-parameter tuning

To really get the best performance for our deep learning models, we should optimize the Mel Spectrograms for the problem that we're trying to solve.

There are a number of hyper-parameters that we can use to tune how the Spectrogram is generated. For that, we need to understand a few concepts about how Spectrograms are constructed.

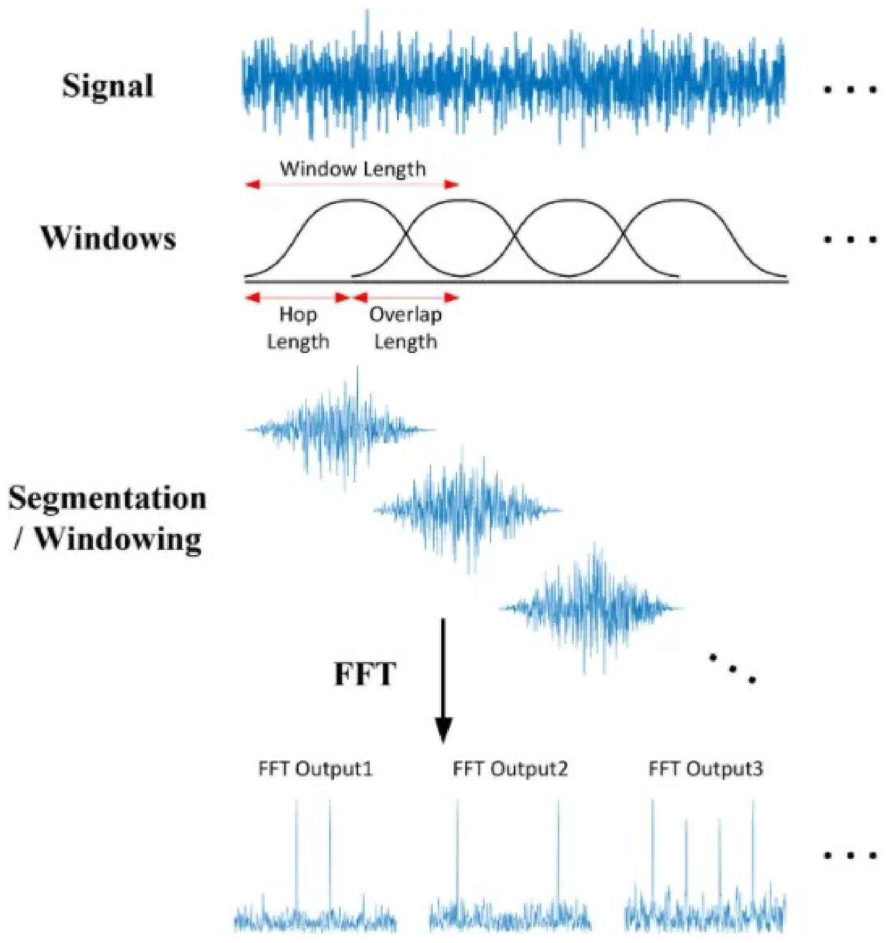
10.1 Fast Fourier Transform (FFT)

One way to compute Fourier Transforms is by using a technique called DFT (Discrete Fourier Transform). The DFT is very expensive to compute, so in practice, the FFT (Fast Fourier Transform) algorithm is used, which is an efficient way to implement the DFT.

However, the FFT will give you the overall frequency components for the entire time series of the audio signal as a whole. It won't tell you how those frequency components change over time within the audio signal. You will not be able to see, for example, that the first part of the audio had high frequencies while the second part had low frequencies, and so on.

10.2 Short-time Fourier Transform (STFT)

To get that more granular view and see the frequency variations over time, we use the STFT algorithm (Short-Time Fourier Transform). The STFT is another variant of the Fourier Transform that breaks up the audio signal into smaller sections by using a sliding time window. It takes the FFT on each section and then combines them. It is thus able to capture the variations of the frequency with time.



This splits the signal into sections along the Time axis. Secondly, it also splits the signal into sections along the Frequency axis. It takes the full range of frequencies and divides it up into equally spaced bands (in the Mel scale). Then, for each section of time, it calculates the Amplitude or energy for each frequency band.

Let's make this clear with an example. We have a 1-minute audio clip that contains frequencies between 0Hz and 10000 Hz (in the Mel scale). Let's say that the Mel Spectrogram algorithm:

- Chooses windows such that it splits our audio signal into 20 time-sections.
- Decides to split our frequency range into 10 bands (ie. 0-1000Hz, 1000-2000Hz, ..., 9000-10000Hz).

The final output of the algorithm is a 2D Numpy array of shape (10, 20) where:

- Each of the 20 columns represents the FFT for one time-section.
- Each of the 10 rows represents Amplitude values for a frequency band.

10.3 Mel Spectrogram Hyperparameters

This gives us the hyperparameters for tuning our Mel Spectrogram. We'll use the parameter names that Librosa uses. (Other libraries will have equivalent parameters)

Frequency Bands

- fmin - minimum frequency
- fmax - maximum frequency to display
- n_mels - number of frequency bands (ie. Mel bins). This is the height of the Spectrogram

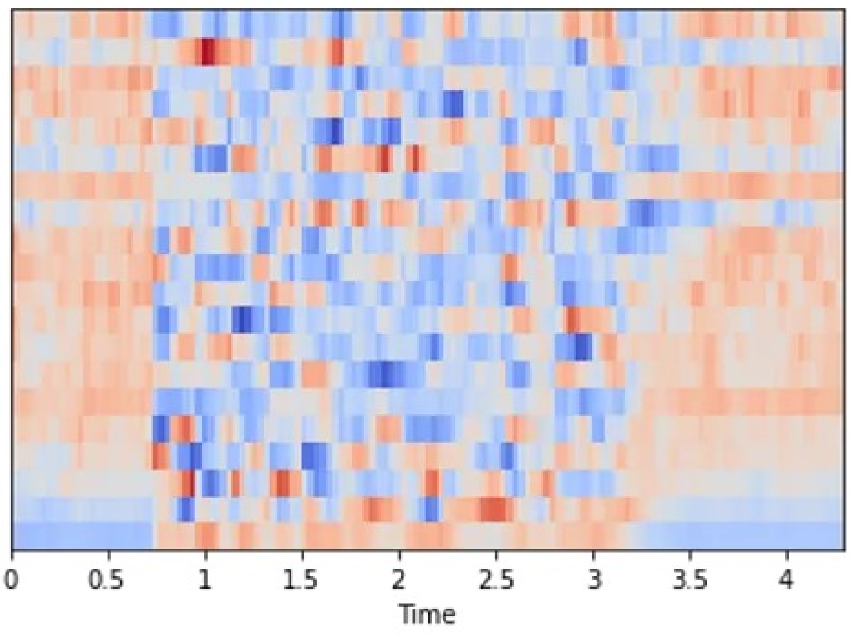
Time Sections

- n_fft - window length for each time section (the name for tf.signal.stft is fft_length)
- hop_length - number of samples by which to slide the window at each step. Hence, the width of the Spectrogram is = Total number of samples / hop_length (the name for tf.signal.stft is frame_step)

11 MFCC (for Human Speech)

Mel Spectrograms work well for most audio deep learning applications. However, for problems dealing with human speech, like Automatic Speech Recognition, you might find that MFCC (Mel Frequency Cepstral Coefficients) sometimes work better.

These essentially take Mel Spectrograms and apply a couple of further processing steps. This selects a compressed representation of the frequency bands from the Mel Spectrogram that correspond to the most common frequencies at which humans speak.



Aplying to the same audio file, the Mel Spectrogram had shape (128, 134), whereas the MFCC has shape (20, 134). The MFCC extracts a much smaller set of features from the audio that are the most relevant in capturing the essential quality of the sound.

12 Data Augmentation

A common technique to increase the diversity of your dataset, particularly when you don't have enough data, is to augment your data artificially. We do this by modifying the existing data samples in small ways.

For instance, with images, we might do things like rotate the image slightly, crop or scale it, modify colors or lighting, or add some noise to the image. Since the semantics of the image haven't changed materially, so the same target label from the original sample will still apply to the augmented sample. eg. if the image was labeled as a cat, the augmented image will also be a cat.

But, from the model's point of view, it feels like a new data sample. This helps your model generalize to a larger range of image inputs.

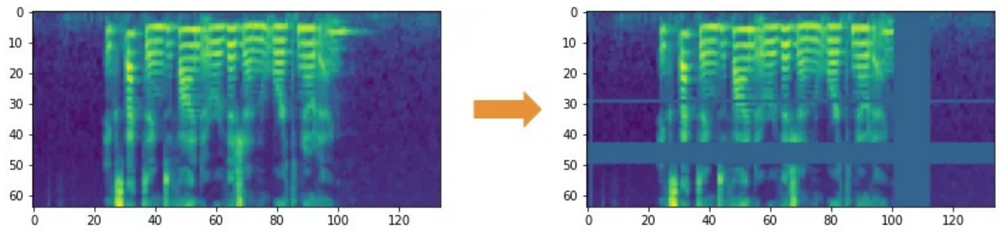
Just like with images, there are several techniques to augment audio data as well. This augmentation can be done both on the raw audio before producing the spectrogram, or on the generated spectrogram. Augmenting the spectrogram usually produces better results.

12.1 Spectrogram Augmentation

The normal transforms you would use for an image don't apply to spectrograms. For instance, a horizontal flip or a rotation would substantially alter the spectrogram and the sound that it represents.

Instead, we use a method known as SpecAugment where we block out sections of the spectrogram. There are two flavors:

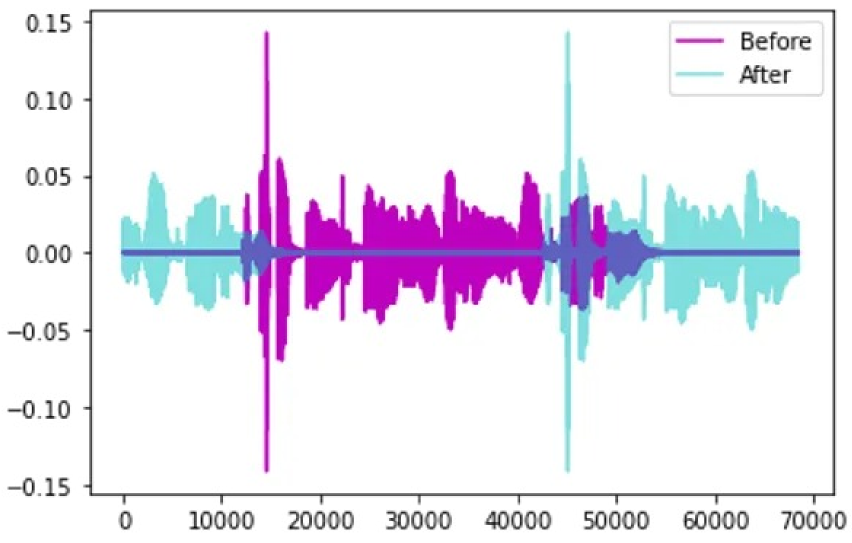
- Frequency mask - randomly mask out a range of consecutive frequencies by adding horizontal bars on the spectrogram.
- Time mask - similar to frequency masks, except that we randomly block out ranges of time from the spectrogram by using vertical bars.



12.2 Raw Audio Augmentation

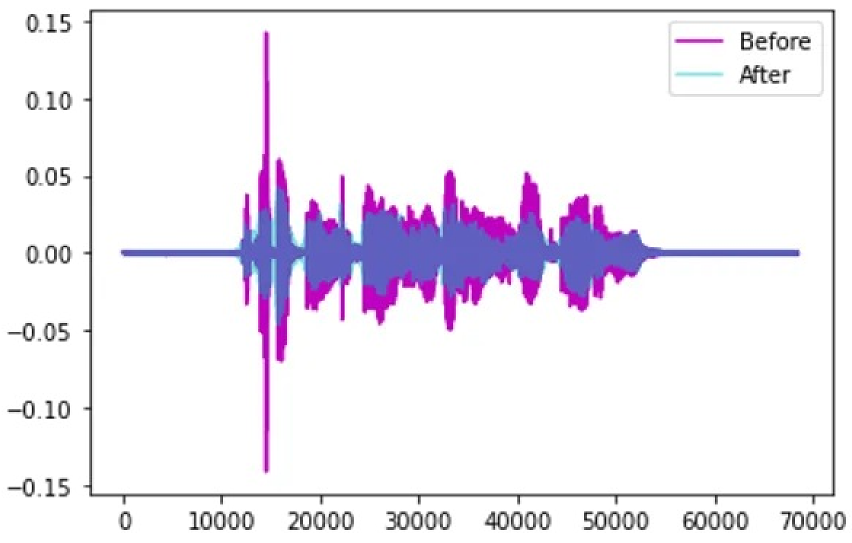
There are several options:

- Time Shift - shift audio to the left or the right by a random amount.
    - For sound such as traffic or sea waves which has no particular order, the audio could wrap around (ie. the end of the audio could be at the start).

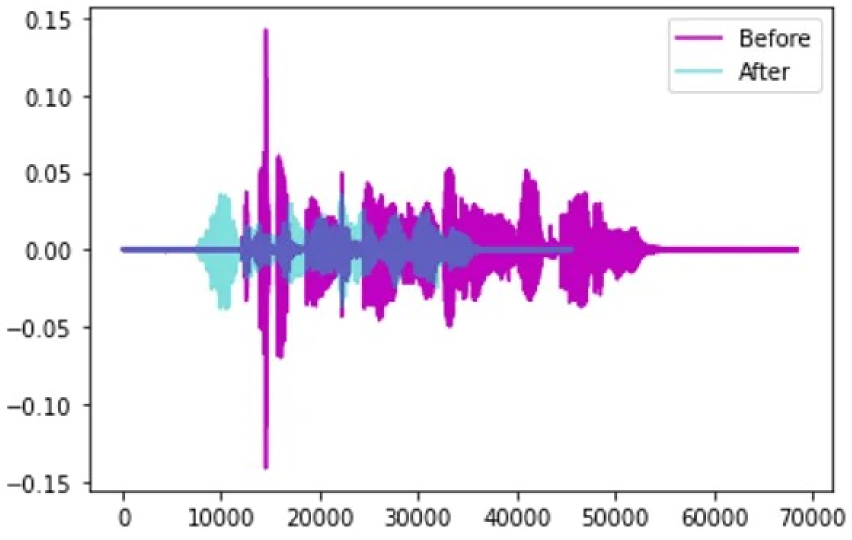


  - Alternately, for sounds such as human speech where the order does matter, the gaps can be filled with silence.

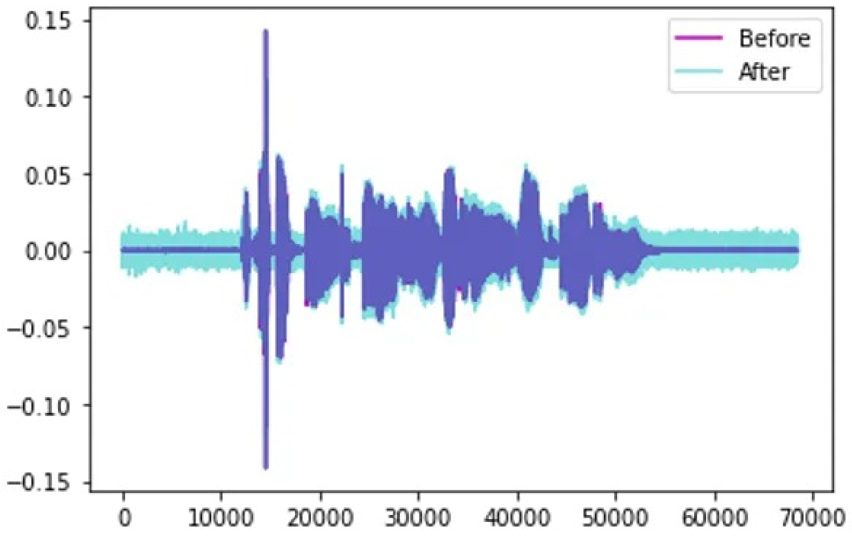
- Pitch Shift - randomly modify the frequency of parts of the sound.



- Time Stretch - randomly slow down or speed up the sound.



- Add Noise - add some random values to the sound.



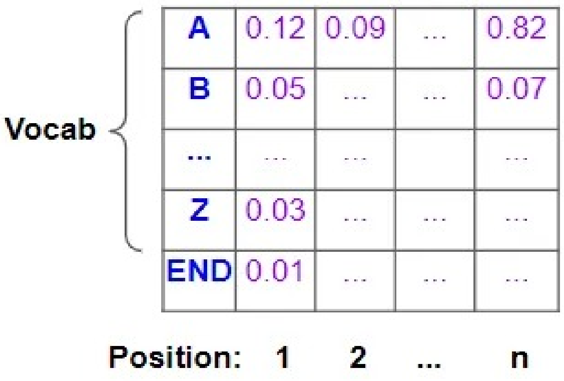
13 Gready Search and Beam Search

There are a couple of commonly used algorithms used as part of their last step to produce their final output.

- Greedy Search is one such algorithm. It is used often because it is simple and quick.
- The alternative is to use Beam Search. It is very popular because, although it requires more computation, it usually produces much better results.

Depending on the problem they're solving, NLP models can generate output as either characters or words. All of the concepts related to Beam Search apply equivalently to either, so I will use both terms interchangeably in this article.

The final output of a NLP model could be represented as:



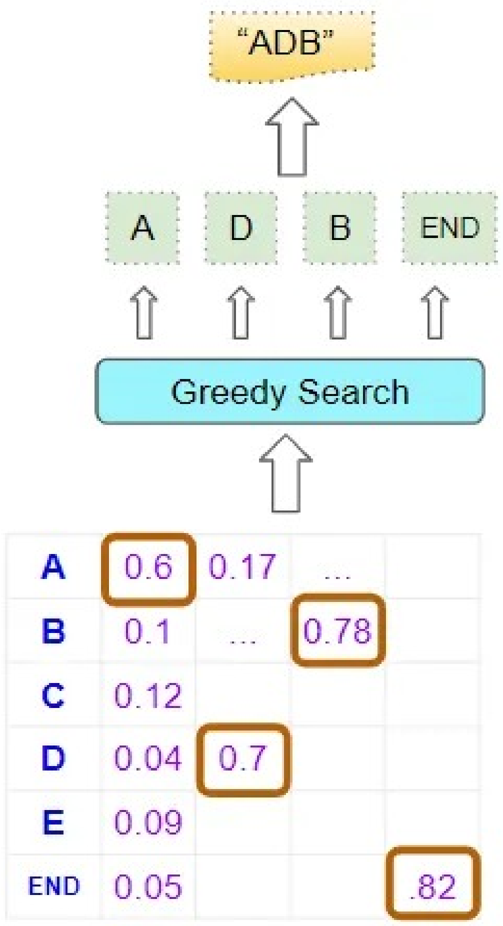
Vocab

Position:	1	2	...	n
A	0.12	0.09	...	0.82
B	0.05	...	...	0.07
...	...	...		...
Z	0.03	...	...	...
END	0.01	...	...	...

Our eventual goal, of course, is not these probabilities but a final target sentence. To get that, the model has to decide which word it should predict for each position in that target sequence.

13.1 Greedy Search

A fairly obvious way is to simply take the word that has the highest probability at each position and predict that. It is quick to compute and easy to understand, and often does produce the correct result.



Easy but it could be better.

13.2 Beam Search

Beam Search makes two improvements over Greedy Search.

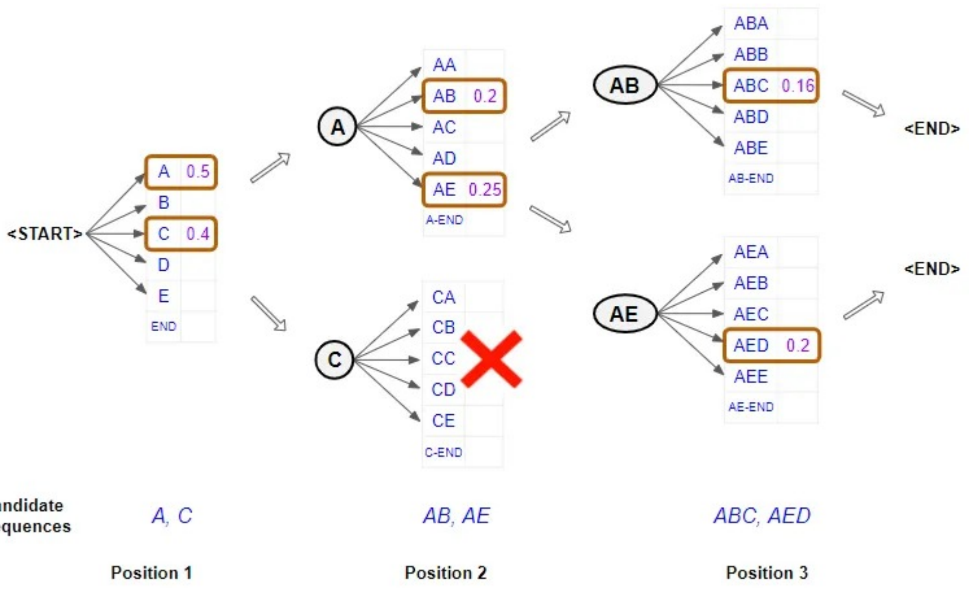
- With Greedy Search, we took just the single best word at each position. In contrast, Beam Search expands this and takes the best N words.
- With Greedy Search, we considered each position in isolation. Once we had identified the best word for that position, we did not examine what came before it (ie. in the previous position), or after it. In contrast, Beam Search picks the N best sequences so far and considers the probabilities of the combination of all of the preceding words along with the word in the current position.

The hyperparameter N is known as the Beam width.

Intuitively it makes sense that this gives us better results over Greedy Search. Because, what we are really interested in is the best complete sentence, and we might miss that if we picked only the best individual word in each position.

13.2.1 What it does

Let's take a simple example with a Beam width of 2, and using characters to keep it simple.



First Position

- Consider the output of the model at the first position. It starts with the "< START >" token and obtains probabilities for each word. It now selects the two best characters in that position, e.g., "A" and "C".

Second Position

- When it comes to the second position, it re-runs the model twice to generate probabilities by fixing the possible characters in the first position. In other words, it constrains the characters in the first position to be either an "A" or a "C" and generates two branches with two sets of probabilities. The branch with the first set of probabilities corresponds to having "A" in position 1, and the branch with the second set corresponds to having "C" in position 1.

- It now picks the **overall** two best character pairs based on the combined probability of the first two characters, from out of both sets of probabilities. So it doesn't pick just one best character pair from the first set and one best character pair from the second set. eg. "AB" and "AE"

Third Position

- When it comes to the third position, it repeats the process. It re-runs the model twice by constraining the first two positions to be either "AB" or "AE" and again generates two sets of probabilities.

- Once more, it picks the overall two best character triplets based on the combined probability of the first three characters from both sets of probabilities. Therefore we now have the two best combinations of characters for the first three positions. eg. "ABC" and "AED".

Repeat till END token

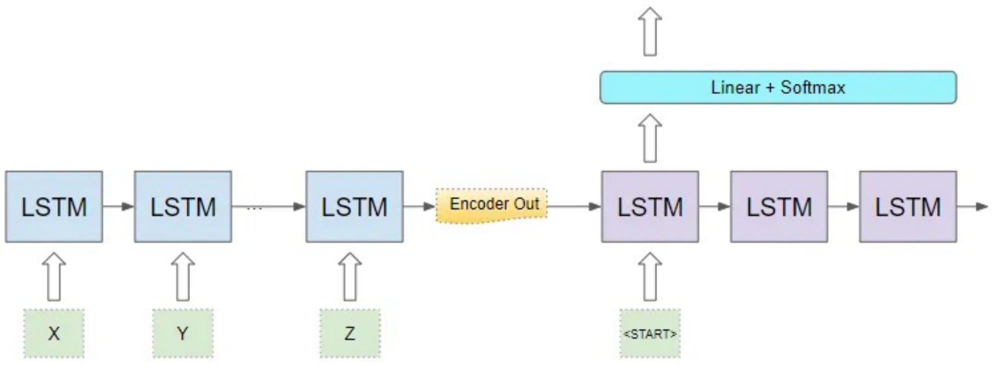
- It continues doing this till it picks an "< END >" token as the best character for some position, which then concludes that branch of the sequence.

It finally ends up with the two best sequences and predicts the one with the higher overall probability.

13.2.2 How it works

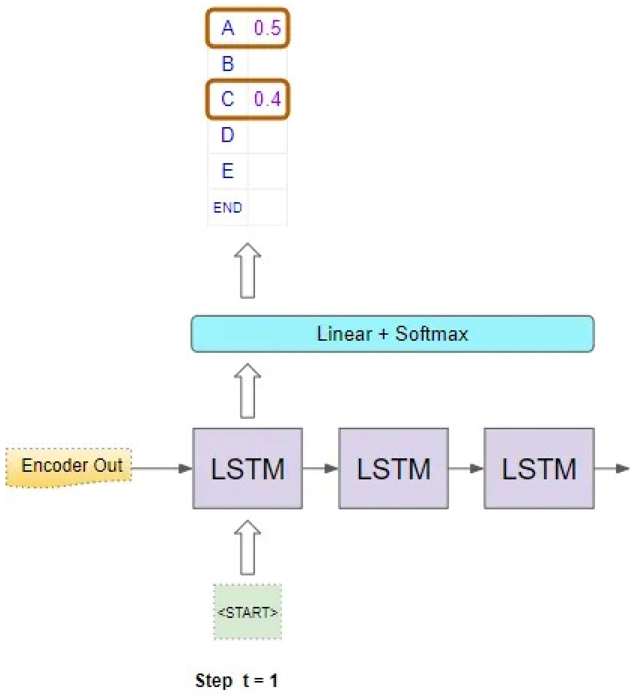
We'll continue with the same example and use a Beam width of 2.

Thinking of a sequence-to-sequence model (for translation for example), the Encoder and Decoder would likely be a recurrent network consisting of some LSTM layers. Alternately it could also be built using Transformers rather than a recurrent network.



Let's focus on the Decoder component and the output layers.

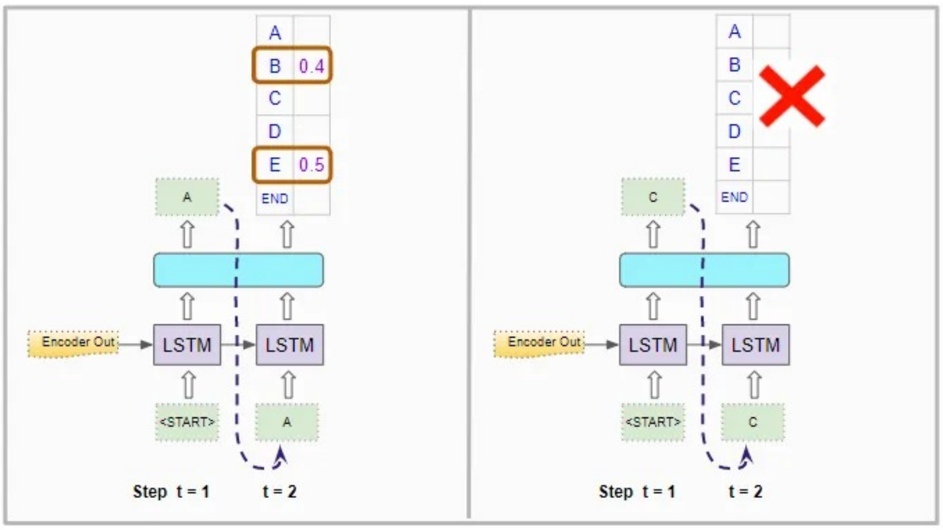
First Position
In the first timestep, it uses the Encoder's output and an input of a "< START >" token to generate the character probabilities for the first position.



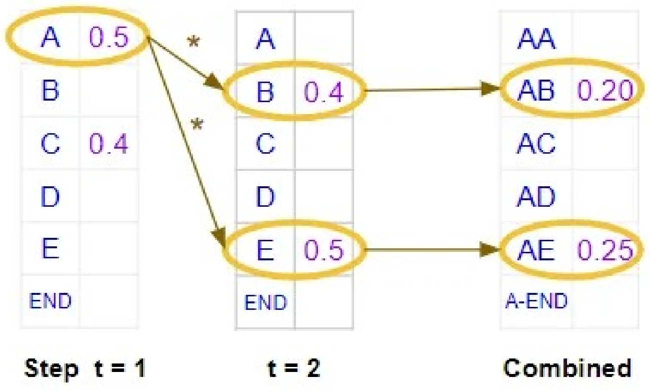
Now it picks two characters with the highest probability eg. “A” and “C”.

Second Position

For the second timestep, it then runs the Decoder twice using the Encoder’s output as before. Along with the “< START >” token in the first position, it forces the input of the second position to be “A” in the first Decoder run. In the second Decoder run, it forces the input of the second position to be “C”.



It generates character probabilities for the second position. But these are individual character probabilities. It needs to compute the combined probabilities for character pairs in the first two positions. The probability of the pair “AB” is the probability of “A” occurring in the first position multiplied by the probability of “B” occurring in the second position, given that “A” is already fixed in the first position. The example below shows the calculation.



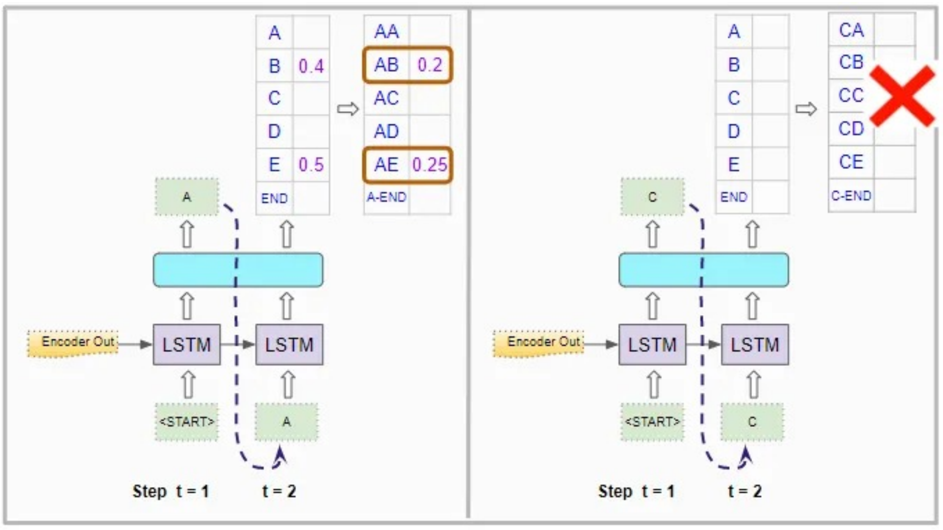
$$\text{Prob (AB | input)} = \text{Prob (A | input)} * \text{Prob (B | A, input)}$$

$$\text{Prob (AB)} = \text{Prob (A)} * \text{Prob (B | A)}$$

$$= 0.5 * 0.4$$

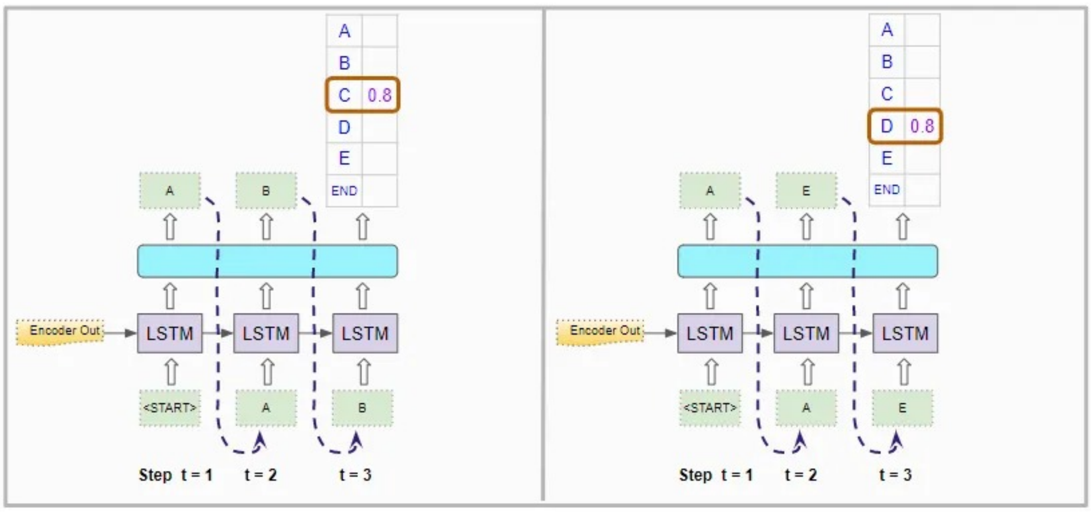
$$= 0.20$$

It does this for both Decoder runs and picks the character pairs with the highest combined probabilities across both runs. It, therefore, picks “AB” and “AE”.



Position 2 : *AB, AE*

Third Position
For the third time step, it again runs the Decoder twice as before. Along with the “< START >”token in the first position, it forces the input of the second position and third positions to be “A” and “B” respectively in the first Decoder run. In the second Decoder run, it forces the input of the second position and third positions to be “A” and “E” respectively.



It calculates the combined probability for character triples in the first three positions (similar as before).

It picks the two best ones across both runs.

Repeat till END token
It repeats this process till it generates two best sequences that end with an "< END >" token.

It then chooses the sequence that has the highest combined probability to make its final prediction.

13.2.3 Conclusion

This comes at the expense of increased computation, and longer execution times. So we should evaluate whether that tradeoff makes sense for our application's use case.

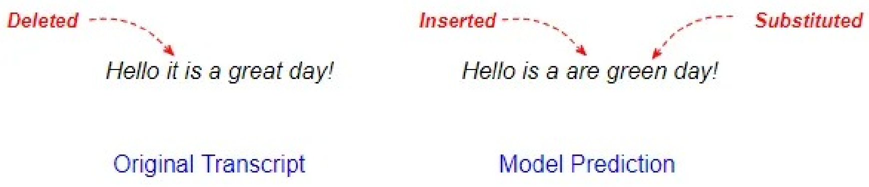
14 Metrics

14.1 Word Error Rate (WER)

Speech-to-Text applications use Word Error Rate, not Bleu Score as metric
Although Automatic Speech Recognition models also output text, the target sentence is unambiguous and usually not subject to interpretation. In this case, Bleu Score is not the ideal metric.

The metric that is typically used for these applications is Word Error Rate (WER), or its sibling, Character Error Rate (CER). 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.

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

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

$$= 0.5$$

The WER calculation is based on the Levenstein distance, which measures the differences between two words.

Although WER is the most widely used metric for Speech Recognition, it has some drawbacks:

- It does not distinguish between words that are important to the meaning of the sentence and those that are not as relevant.
- When comparing words, it does not consider whether two words are different in just a single character or are completely different.

14.2 Char Error Rate (CER)

This value indicates the percentage of characters that were incorrectly predicted. The lower the value, the better the performance of the ASR system with a CharErrorRate of 0 being a perfect score. Character error rate can then be computed as:

$$\text{CharErrorRate} = \frac{S + D + I}{N} = \frac{S + D + I}{S + D + C}$$

where

- **S** is the number of substitutions,
- **D** is the number of deletions,
- **I** is the number of insertions,
- **C** is the number of correct characters,
- **N** is the number of characters in the original transcript ($N = S + D + C$).

15 Speech-to-Text model's architecture

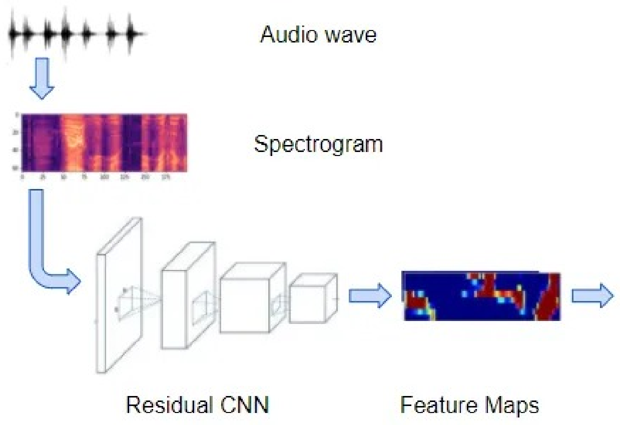
There are many variations of deep learning architecture for ASR. Two commonly used approaches are:

- 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.

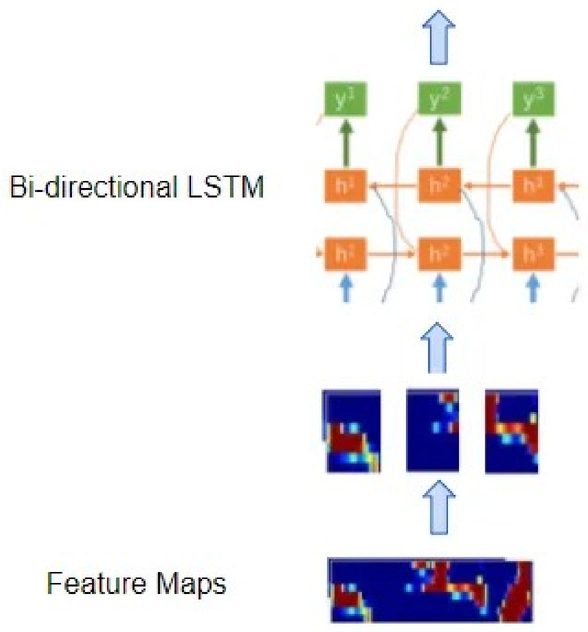
- An RNN-based sequence-to-sequence network that treats each 'slice' of the spectrogram as one element in a sequence eg. Google's Listen Attend Spell (LAS) model.

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:

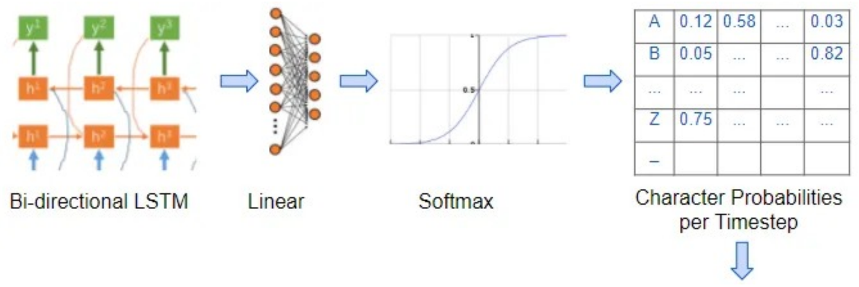
- A regular convolutional network consisting of a few Residual CNN layers that process the input spectrogram images and output feature maps of those images.



- 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 character probabilities for each timestep of the output.

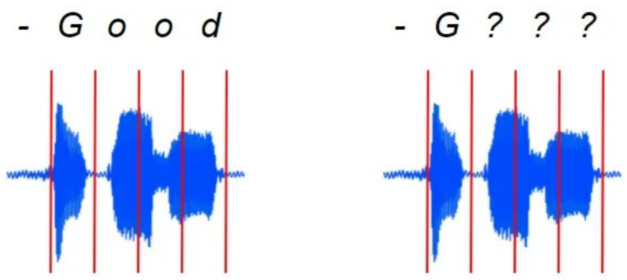


- We also have linear layers that sit between the convolution and recurrent networks and help to reshape the outputs of one network to the inputs of the other.

15.1 Align the sequences

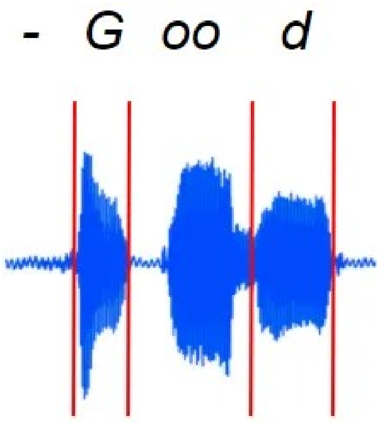
If you think about this a little bit, youâ€™ll realize that there is still a major missing piece in our puzzle. Our eventual goal is to map those timesteps or â€˜framesâ€™ to individual characters in our target transcript.

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.

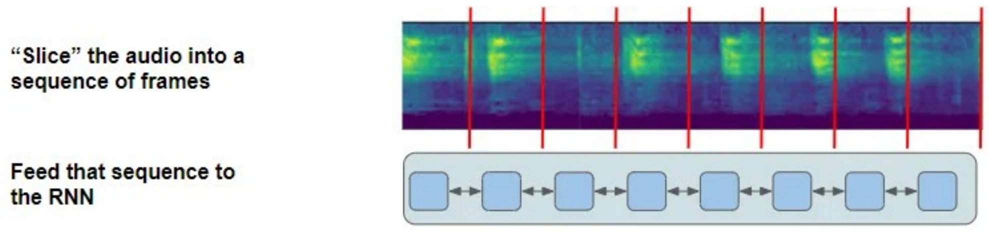
The way we tackle this is by using an ingenious algorithm called Connectionist Temporal Classification, or CTC for short.

15.2 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.


"Slice" the audio into a sequence of frames
Feed that sequence to the RNN

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.

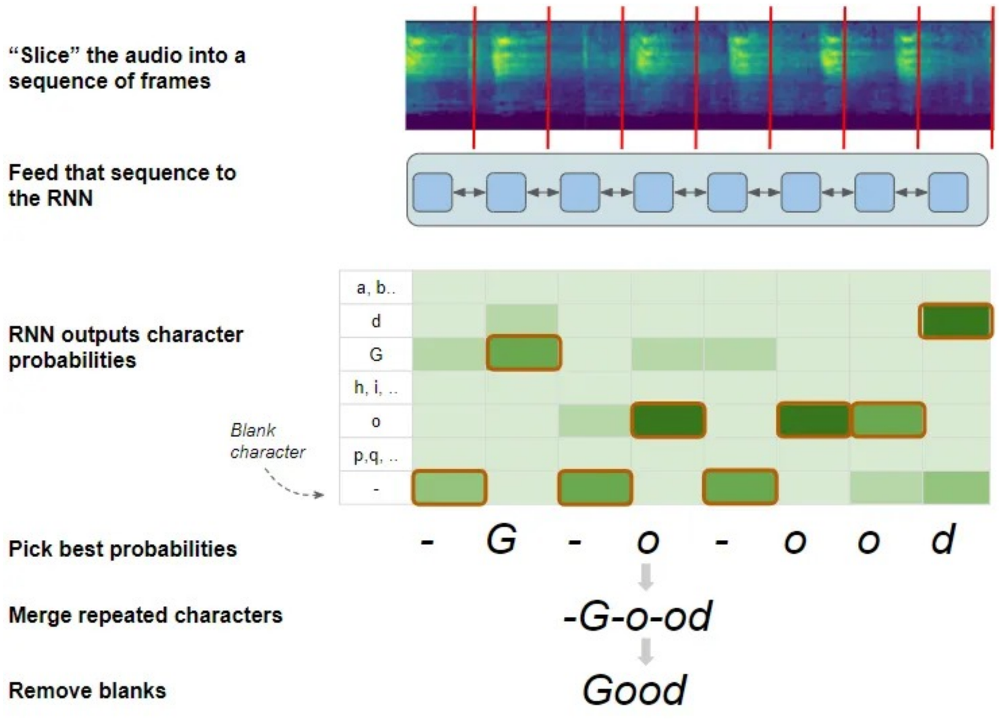
CTC works in two modes:

- 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.

15.2.1 CTC Decoding

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


"Slice" the audio into a sequence of frames
Feed that sequence to the RNN
RNN outputs character probabilities
Blank character
a, b..
d
G
h, i, ..
o
p,q, ..
-
Pick best probabilities
- G - o - o o d
Merge repeated characters
-G-o-od
Remove blanks
Good

- 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".

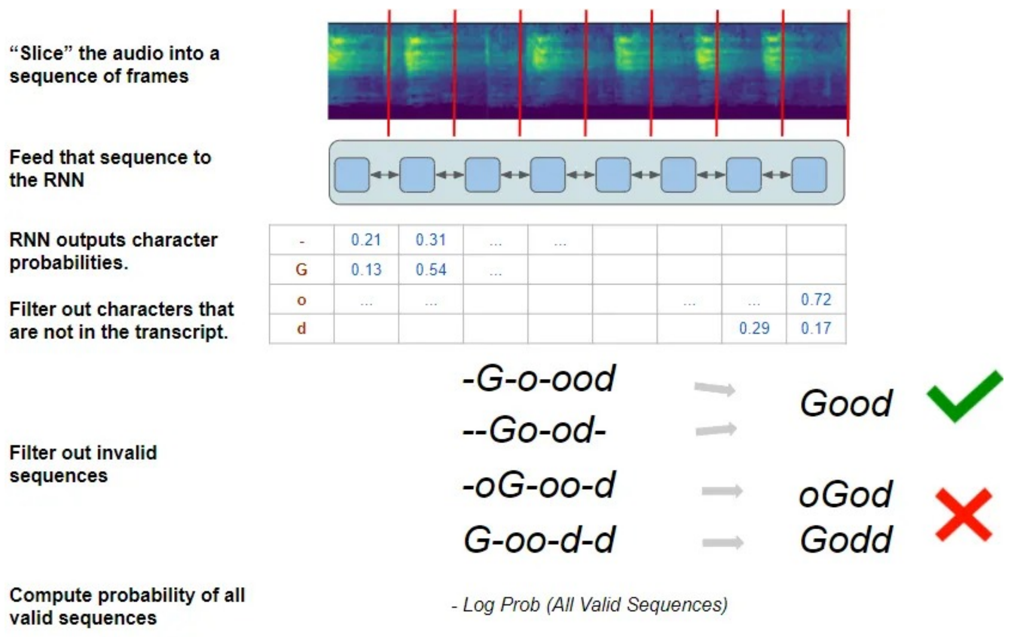
15.2.2 CTC Loss

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.


"Slice" the audio into a sequence of frames
Feed that sequence to the RNN
RNN outputs character probabilities.
Filter out characters that are not in the transcript.
Filter out invalid sequences
-G-o-ood
--Go-od-
-oG-oo-d
G-oo-d-d
Good
oGod
Godd
Compute probability of all valid sequences
- Log Prob (All Valid Sequences)

-	0.21	0.31	...	...				
G	0.13	0.54	...					
o	...	...				...	...	0.72
d							0.29	0.17

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.

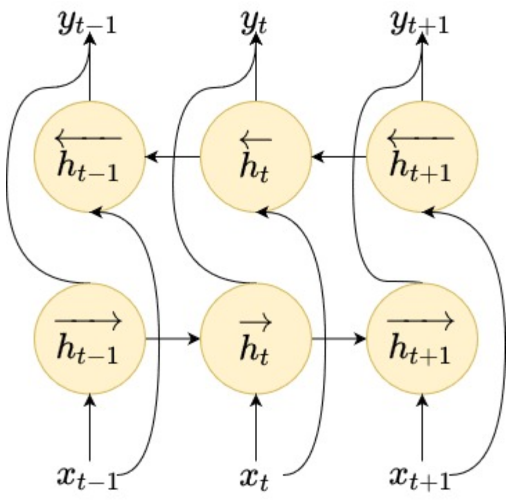
To actually do this, however, is much more complicated than what I've described here. The challenge is that there is a huge number of possible combinations of characters to produce a sequence. With our simple example alone, we can have 4 characters per frame. With 8 frames that gives us 4^8 combinations (= 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.

15.3 Recurrent Neural Networks (RNN)

RNNs perform computations on the time sequence since their current hidden state is dependent on all the previous hidden states. More specifically, they are designed to model time-series signals as well as capture long-term and short-term dependencies between different time-steps of the input.

Concerning speech recognition applications, the input signal $x = (x_1, x_2, ..., x_T)$ is passed through the RNN to compute the hidden sequences $h = (h_1, h_2, ..., h_N)$ and the output sequences $y = (y_1, y_2, ..., y_N)$, respectively. One major drawback of the simple form of RNNs is that it generates the next output based only on the previous context.



y_{t-1}
y_t
y_{t+1}
$\overleftarrow{h_{t-1}}$
$\overleftarrow{h_t}$
$\overleftarrow{h_{t+1}}$
$\overrightarrow{h_{t-1}}$
$\overrightarrow{h_t}$
$\overrightarrow{h_{t+1}}$
x_{t-1}
x_t
x_{t+1}

Figure 1: Bidirectional RNN

RNNs compute the sequence of hidden vectors **h** as:

$$h_t = H(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

where **W** are the weights, **b** are the bias vectors and **H** is the nonlinear function.

15.3.1 RNNs limitations and solutions

However, in speech recognition, usually the information of the future context is equally significant as the past context (Graves et al.). That's why instead of using a unidirectional RNN, **bidirectional RNNs (BiRNNs)** are commonly selected in order to address this shortcoming. BiRNNs process the input vectors in both directions i.e., forward and backward, and keep the hidden state vectors for each direction as shown in the above figure.

This problem can be addressed using:

- Hidden Markov Models (HMMs) to get the alignment between the input audio and its transcribed output.
- Connectionist Temporal Classification (CTC) loss, which is the most common technique.

15.3.2 What is the vanishing gradient problem?

The gradients of the loss function in neural networks approach zero when more layers with certain activation functions are added, making the network difficult to train.

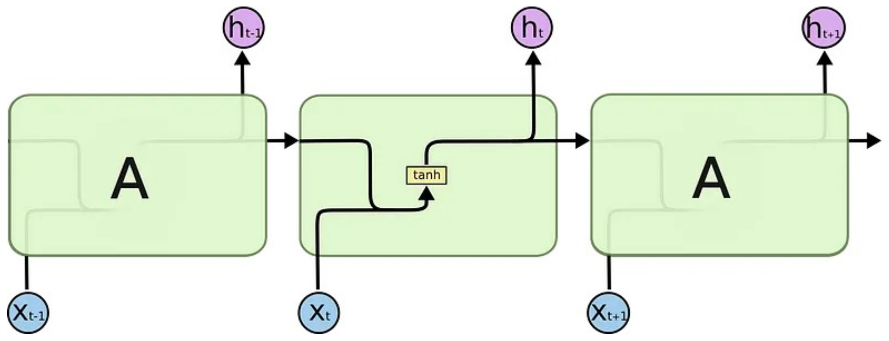
15.3.3 Long Short-Term Memory (LSTM)

LSTMs come to the rescue to solve the vanishing gradient problem. It does so by ignoring (forgetting) useless data/information in the network. The LSTM will forget the data if there is no useful information from other inputs (prior sentence words). When new information comes, the network determines which information to be overlooked and which to be remembered.

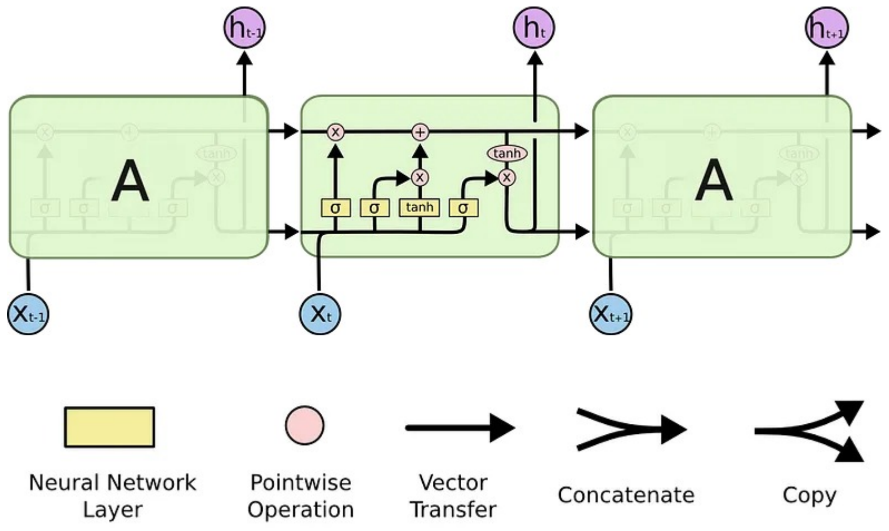
15.3.4 LSTM Architecture

Let's look into the difference between RNNs and LSTMs.

In RNNs, we have a very simple structure with a single activation function (tanh).

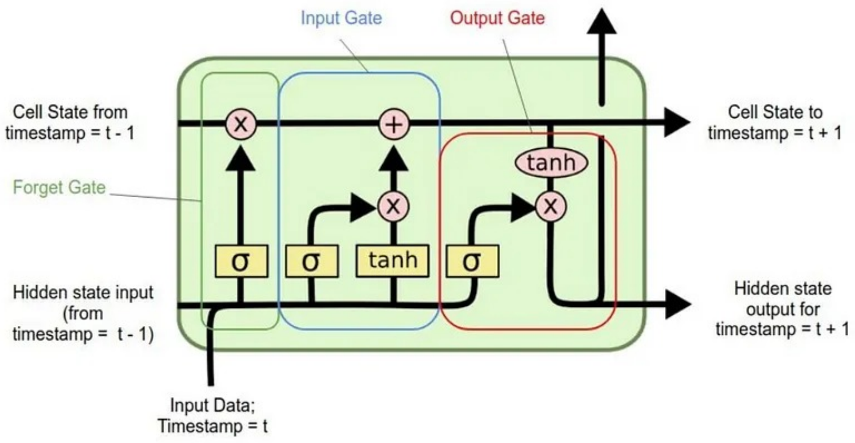


In LSTMs, instead of just a simple network with a single activation function, we have multiple components, giving power to the network to forget and remember information.



LSTMs have 4 different components, namely

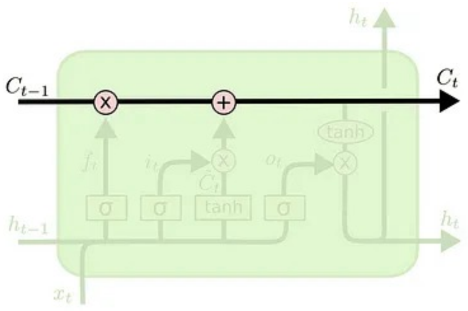
- Cell state (Memory cell)
- Forget gate
- Input gate
- Output gate


Input Gate
Output Gate
Cell State from timestamp = t - 1
Cell State to timestamp = t + 1
Forget Gate
tanh
σ
σ
tanh
σ
Hidden state input (from timestamp = t - 1)
Hidden state output for timestamp = t + 1
Input Data; Timestamp = t

Let's understand these components, one by one.

Cell State (Memory cell)

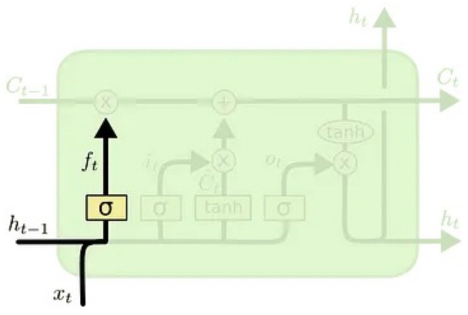
It is the first component of LSTM which runs through the entire LSTM unit. It kind of can be thought of as a conveyer belt.



This cell state is responsible for remembering and forgetting. This is based on the context of the input. This means that some of the previous information should be remembered while some of them should be forgotten and some of the new information should be added to the memory. The first operation (X) is the pointwise operation which is nothing but multiplying the cell state by the output of the forget gate (which is a number between 0 and 1). The information multiplied by 0 will be forgotten by the LSTM. Another operation is (+) which is responsible to add some new information to the state.

Forget Gate

The forget LSTM gate, as the name suggests, decides what information should be forgotten. A sigmoid layer is used to make this decision. This sigmoid layer is called the "forget gate layer".

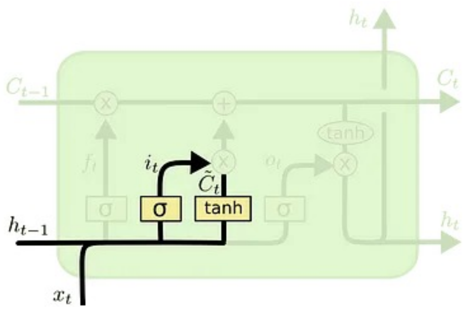


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

It does a dot product of h(t-1) and x(t) and with the help of the sigmoid layer, outputs a number between 0 and 1 for each number in the cell state C(t-1). If the output is a '1', it means we will keep it. A '0' means to forget it completely.

Input gate

The input gate gives new information to the LSTM and decides if that new information is going to be stored in the cell state.

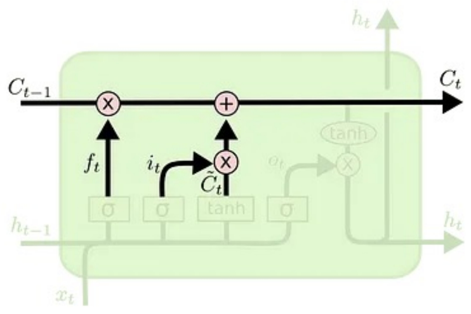


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

This has 3 parts:

- A sigmoid layer decides the values to be updated. This layer is called the "input gate layer"
- A tanh activation function layer creates a vector of new candidate values, C̃(t), that could be added to the state.
- Then we combine these 2 outputs, i(t) * C̃(t), and update the cell state.

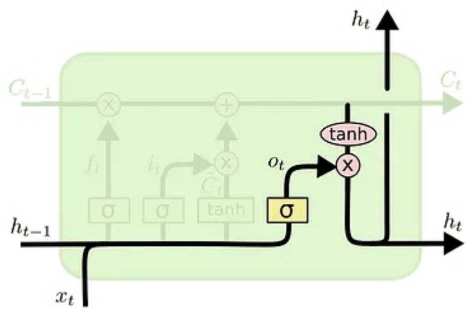
The new cell state C(t) is obtained by adding the output from forget and input gates.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output gate

The output of the LSTM unit depends on the new cell state.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

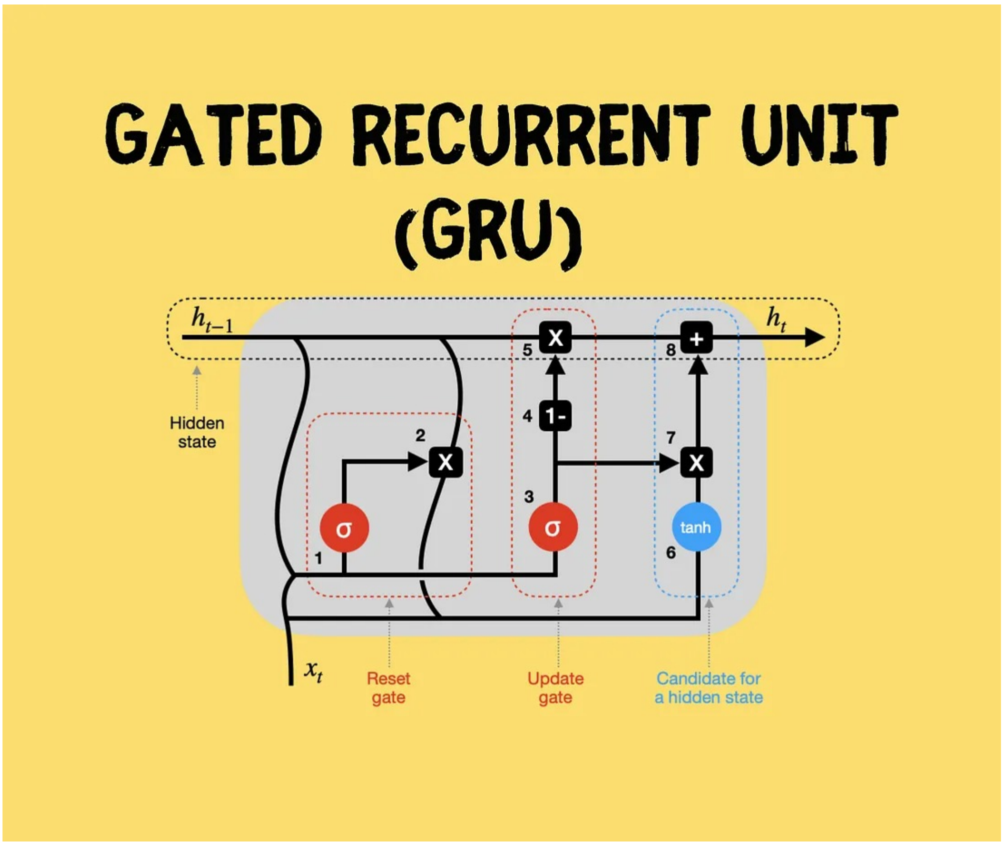
First, a sigmoid layer decides what parts of the cell state we're going to output. Then, a tanh layer is used on the cell state to squash the values between -1 and 1, which is finally multiplied by the sigmoid gate output.

15.3.5 Gated Recurrent Unit (GRU)

GRU stands for Gated Recurrent Unit, which is a type of recurrent neural network (RNN) architecture that is similar to LSTM (Long Short-Term Memory).

Like LSTM, GRU is designed to model sequential data by allowing information to be selectively remembered or forgotten over time. However, GRU has a simpler architecture than LSTM, with fewer parameters, which can make it easier to train and more computationally efficient.

The main difference between GRU and LSTM is the way they handle the memory cell state. In LSTM, the memory cell state is maintained separately from the hidden state and is updated using three gates: the input gate, output gate, and forget gate. In GRU, the memory cell state is replaced with a "candidate activation vector" which is updated using two gates: the reset gate and update gate.



The reset gate determines how much of the previous hidden state to forget, while the update gate determines how much of the candidate activation vector to incorporate into the new hidden state.

Overall, GRU is a popular alternative to LSTM for modeling sequential data, especially in cases where computational resources are limited or where a simpler architecture is desired.

15.3.6 GRU Architecture

The GRU architecture consists of the following components:

- **Input layer:** The input layer takes in sequential data, such as a sequence of words or a time series of values, and feeds it into the GRU.
- **Hidden layer:** The hidden layer is where the recurrent computation occurs. At each time step, the hidden state is updated based on the current input and the previous hidden state. The hidden state is a vector of numbers that represents the network's "memory" of the previous inputs.
- **Reset gate:** The reset gate determines how much of the previous hidden state to forget. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the previous hidden state is "reset" at the current time step.

- **Update gate:** The update gate determines how much of the candidate activation vector to incorporate into the new hidden state. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the candidate activation vector is incorporated into the new hidden state.
- **Candidate activation vector:** The candidate activation vector is a modified version of the previous hidden state that is "reset" by the reset gate and combined with the current input. It is computed using a tanh activation function that squashes its output between -1 and 1.
- **Output layer:** The output layer takes the final hidden state as input and produces the network's output. This could be a single number, a sequence of numbers, or a probability distribution over classes, depending on the task at hand.

15.3.7 Pros and Cons of GRU

Here are some pros and cons of using GRU:

Pros:

- GRU networks are similar to Long Short-Term Memory (LSTM) networks, but with fewer parameters, making them computationally less expensive and faster to train.
- GRU networks can handle long-term dependencies in sequential data by selectively remembering and forgetting previous inputs.
- GRU networks have been shown to perform well on a variety of tasks, including natural language processing, speech recognition, and music generation.
- GRU networks can be used for both sequence-to-sequence and sequence classification tasks.

Cons:

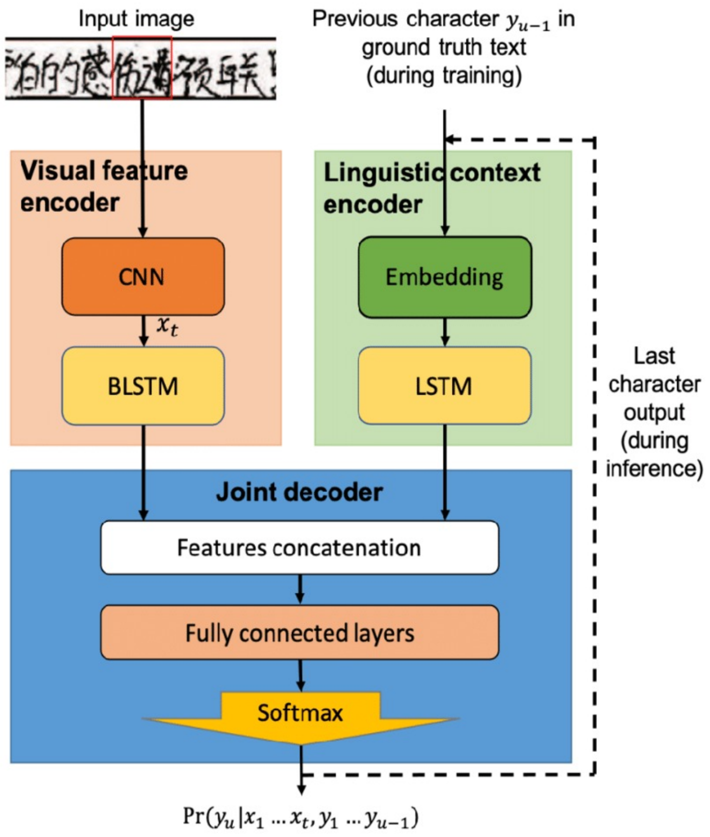
- GRU networks may not perform as well as LSTMs on tasks that require modeling very long-term dependencies or complex sequential patterns.
- GRU networks may be more prone to overfitting than LSTMs, especially on smaller datasets.
- GRU networks require careful tuning of hyperparameters, such as the number of hidden units and learning rate, to achieve good performance.
- GRU networks may not be as interpretable as other machine learning models, since the gating mechanism can make it difficult to understand how the network is making predictions.

15.4 RNN-Transducer

In other works, an architecture commonly known as RNN-Transducer, has also been employed for ASR. This method combines an RNN with CTC and a separate RNN that predicts the next output given the previous one. It determines a separate probability distribution $P(y_k|t, u)$ for every timestep t of the input and time-step u of the output for thek-th element of the output y.

An encoder network converts the acoustic feature x_t at time-step t to a representation and a prediction network takes the previous label y_{u-1} and generates a new representation. The joint network is a fully-connected layer that combines the two representations and generates the posterior probability.

In this way the RNN-Transducer can generate the next symbols or words by using information both from the encoder and the prediction network based on if the predicted label is a blank or a non-blank label. The inference procedure stops when a blank label is emitted at the last time-step.



Input image
Previous character y_{u-1} in ground truth text (during training)
Visual feature encoder
CNN
x_t
BLSTM
Linguistic context encoder
Embedding
LSTM
Last character output (during inference)
Joint decoder
Features concatenation
Fully connected layers
Softmax
$\Pr(y_u|x_1 \ldots x_t, y_1 \ldots y_{u-1})$

Graves et al. tested regular RNNs with CTC and RNN-Transducers in TIMIT database using different numbers of layers and hidden states.

In the table below, it is shown that RNN-T with 3 layers of 250 hidden states each has the best performance of 17.7% phoneme error rate (PER), while simple RNN-CTC models perform worse with PER>18.4%.

Network	Weights	Epochs	PER
CTC-3L-500H-TANH	3.7M	107	37.6%
CTC-1L-250H	0.8M	82	23.9%
CTC-1L-622H	3.8M	87	23.0%
CTC-2L-250H	2.3M	55	21.0%
CTC-3L-421H-UNI	3.8M	115	19.6%
CTC-3L-250H	3.8M	124	18.6%
CTC-5L-250H	6.8M	150	18.4%
TRANS-3L-250H	4.3M	112	18.3%
PRETRANS-3L-250H	**4.3M**	**144**	**17.7%**

References

[1] *Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques*

[2] *Audio Deep Learning Made Simple (Part 2): Why Mel Spectrograms perform better*

[3] *Audio Deep Learning Made Simple (Part 3): Data Preparation and Augmentation*

[4] *Foundations of NLP Explained Visually: Beam Search, How It Works*

[5] *Foundations of NLP Explained - Bleu Score and WER Metrics*

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

[7] *Speech Recognition: a review of the different deep learning approaches*

[8] *Introduction to Long Short-Term Memory (LSTM)*

[9] *Understanding Gated Recurrent Unit (GRU) in Deep Learning*

[10] *Char Error Rate*