Amazon Alexa

Q: How will users interact with Alexa?

A: Users will interact with Alexa through an Amazon Echo Dot, which is a small device that can be placed on any surface, like a table or a countertop.

Q: Will users initilize an interaction with a wake word? If so, do I need to implement that?

A: Yes, you'll need to implement wake-word detection. The wake word will be "Alexa", though the exact word shouldn't matter much for this exercise.

Q: Can I assume that only a single user will be speaking at once in a relatively quiet room?

A: Yes.

Q: Should Alexa be able to handle multiple languages?

A: No. A single language is okay.

Q: Can I assume that the user is a native english speaker?

A: Yes, that's fine for this interview.

Q: What capabilities do we need to support as a voice assistant?

A: For now, let's assume that we have about a dozen "skills" that can be called on (e.g., asking for the weather forecast,ordering food, fetching a ride, playing a song, etc.).

Q: Do any of the skills require speaker recognition? For example, should Alexa behave differently depending on who's asking a question?

A: For now, let's assume that the skills don't require speaker recognition.

1. Gathering System Requirements

As with any ML design interview question, the first thing that we want to do is to gather requirements; we need to figure out what we need to build and what we don't need to build.

We're designing Amazon Alexa, a voice assistant. The voice assistant will delegate tasks to pre-created "skills" (weather, Uber, Spotify, etc).

To accomplish this we'll need to design 5 components:

- Keyword Spotter
- Automatic Speech Recognizer
- Natural Language Understanding Engine
- Dialog Manager
- Text-to-Speech Engine

We'll need to rely on systems which we won't touch on or design:

- Amazon's Echo Dot, a device used to interface with customers which includes a speaker and an array of microphones.
- AWS which is a cloud infrastructure to host our microservies.

2. Keyword Spotter (KWS)

The KWS will be hosted on the Echo Dot which will listen for the wake word. The wake word will signal the device to begin recording the customer's request. Once the customer stops speaking, the MP3 audio file will be streamed to the cloud. In our case, this will be 48 kHz sample rate at a 16-bit depth.

Since we're hosting the KWS on-device, we'll have a couple of constraints:

- The model needs to be small enough to fit in the limited memory.
- The model also has to have a limited number of parameters such that the processor can run predictions with minimal latency

Keeping these constraints in mind, we'll also need to ensure the model will provide a low enough False Rejection Rate (false negative in detecting wake word) and False Acceptance Rate (false positive in detecting wake word) so that we don't frustrate users or compromise user privacy.

Architecturally, the KWS will operate on a 1 second sliding window of audio. This sliding window comes from the fact that the 90th percentile of users require 900ms to say the wake word (WW). To get features from the sliding window audio signal, we'll be creating a spectrogram. From the 1 second of audio, we'll extract 25 millisecond (ms) subsections with an overlap of 10 ms. The short-time discrete Fourier transform will be used on each subsection to create a periodogram. A series of overlapping Mel-spaced triangle filters will be used to extract numerical features. In total, 26 features will represent a single 25 ms subsection. Finally, we take the log of all the features across all of the subsections to create a log Mel filter bank energy spectrogram. This spectrogram will be used as the features for our model. We'll use a

convolutional neural network with to predict whether or not the features contain the wake word. However, since not all wake words will align with the 1 second window, we'll need to have three possibilities output from the convolutional neural network. One, the wake word ending center of the window. Two, the wake word ending after the center of the window. And three, the wake word starting at the end of the window. Finally, to account for noise, we'll need to perform mean subtraction on the spectrogram.

3. Automatic Speech Recognizer (ASR)

The ASR will be hosted in the cloud. It will take in raw audio, featurize it with a spectrogram, and output text representing the spoken language in the raw audio.

Architecturally, the ASR will start with a 1-dimensional convolutional neural network. This will assist in representing n-grams from the phonemes. That result will feed into a bidirectional recurrent neural network which will encode a sequence provided by the convolutional neural network. Finally, the model will have a fully connected layer with a softmax outputting characters and punctuation. The connectionist temporal classification (CTC) loss function will be used to train the network. The CTC loss function assists in aligning the per character prediction corresponding to the provided spectrogram. Without it, the labelling of the dataset per character with each timestep in the spectrogram would be expensive and would also likely not generalize to different speaker paces. CTC uses beam search to efficiently evaluate every possible valid alignment of characters associated with a spectrogram.

4. Natural Language Understanding Engine (NLU)

The NLU engine will be hosted in the cloud. It will take in text output from the ASR and transform it into a structured format expected by the Dialog Manager. This format will require the NLU engine to extract a domain, an intent, and a number of slots. The domains consist of RideServices, Music, Groceries, etc. Intents are associated with an individual domain. For instance, the Music domain could have the intent PlaySong if a user wants to play a song on Spotify or GetArtistFromSongTitle if a user wants to know the artist of a given song. Finally, slots provided the information required with the intent. For the PlaySong intent, a slot would require the song title and perhaps an artist if it can't be implied. Each intent will be associated with a core utterance. For instance, RideService intent may use 'Call me an Uber to X.' as a core utterance. Slots will also be specified for an intent or 'skill'. An example of a slot for the RideService intent could be a destination such as work or airport. These slots can be refined after the skill is created based on data obtained after the launch.

Architecturally, the model we create will have to be a multi-tasking model. Each domain will be a task for the model to learn since each domain will have separate features and labels. The first layer of the model will be an embedding layer. This will allow the text produced by the ASR to

be represented in a euclidean space. This means words will be further or closer to one another based on their usage similarities. A pre-trained embedding layer can be used here such as fastText. The embedding layer will feed into a shared bidirectional LSTM to encode the sequence of words. Then, two more bidirectional LSTM layers will be used. One for intent classification and the other for slot tagging. These two layers will be optimized independently for each domain while the original shared bidirectional LSTM will be trained across domains. This shared layer is what creates a multi-task model. The intent classification layer will use a softmax to predict which intent the user is specifying. The slot tagging layer will label each word in the input based on the inside-outside-beginning (IOB) format. Both the intent classification layer and the slot tagging layer use a loss function which incorporates the performance of each other as well as itself. This means the objective is in the category of 'joint learning'. In the case of a new intent being added, we'll need to be able to use data from other intents because there will be limited training examples for newly created skills. We can use a technique called transfer learning. This can be done by pre-training the network with popular intents. The pre-trained network will then be equipped with untrained softmaxes and then further trained with the limited examples available for the new intent resulting in a network that leverages popular intents to support new intents.

5. Dialog Manager (DM)

The DM will control the flow of the conversation of the voice assistant in accordance with the requirements of a specific intent. As well, the DM will be responsible for delegating requests to dedicated services to fulfill users' intents. For instance, if a user says 'Call me an Uber' the DM will specify that more information is necessary and reply with 'Where would you like to go?' After gathering the required information for the RideService intent, the DM will call the micro-service dedicated to maintaining the RideService logic to complete the user's request and arrange an Uber. The DM will have access to a set of schemas which will list required and optional slots for a given intent. For example if a user specifies 'Play a song', the DM will be able to look up the PlayMusic intent schema and see that at least a genre is required. As well, the schema will specify text which will be sent to the text-to-speech engine such that the genre field can be populated e.g. 'What genre of music would you like to play?' As well, the DM will provide validation or inform users of problems reported by intent services e.g. 'Your Uber is on the way.' The DM will also be responsible for maintaining humanistic qualities. For instance, if a micro-service is taking too long to respond, the DM can fill the silence with 'Hmm, let me think about that'.

6. Text-to-speech Engine (TTS)

The TTS will be responsible for taking the output of the DM and transforming it into an audio signal to be sent to the Echo Dot and played back for the user to hear. This will allow users to get confirmations of orders, have questions answered, or to hear a weather forecast.

Architecturally, the components of TTS include:

- Spectrogram Prediction
- Neural Vocoder

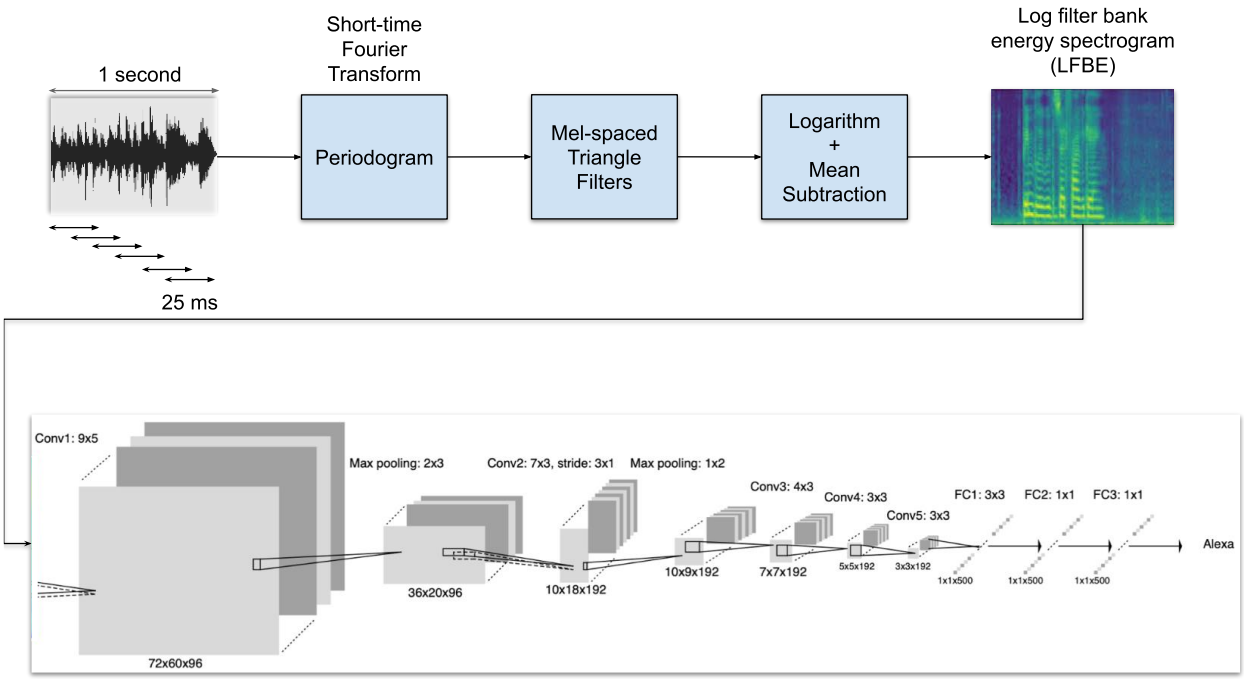
The spectrogram prediction model will be responsible for producing a spectrogram from the text generated by the DM. The neural vocoder will take the produced spectrogram and transform it into an audio signal.

The spectrogram prediction model will take the form of an encoder-decoder. The encoder will take in a series of words and encode it into a fixed representation. This representation will then be sent to the decoder to be decoded into a spectrogram. Since the encoder produces a fixed length output and the encoder input can be much longer, an attention mechanism is required to ensure that the the overall meaning doesn't get lost while condensing a long sequence to a fixed dimension. The encoder will begin with an embedding layer such that each word gets transformed into an embedding. Next, a convolutional layer is used to allow for the representation of n-grams. Lastly, the encoder is layered with a bidirectional LSTM so that the model can encode the given sequence. The attention layer is used between the encoder and decoder to assign probabilities of each input step being associated with a particular output step of the encoder. This improves the performance of the model when provided longer sequences than the fixed length representation used by the encoder. The decoder is first equipped with a unidirectional LSTM which feeds the previous timestep's output back to the attention layer. This creates a location-based attention mechanism which encourages the model to avoid common failure modes such as ignoring portions of a sequence or repeating portions of a sequence. The output of the unidirectional LSTM is fed into a single layer fully-connected neural network which will output the values of each frequency on the spectrogram at each timestep. As well, a pre-net (2-layer feed-forward neural network) is used alongside the unidirectional LSTM to provide the LSTM with the previous output of the decoder. A post-net (convolutional neural network) has shown to improve the reconstruction of the spectrogram by the decoder. Lastly, a single layer feed-forward neural network followed by a sigmoid to predict the end of the generated spectrogram.

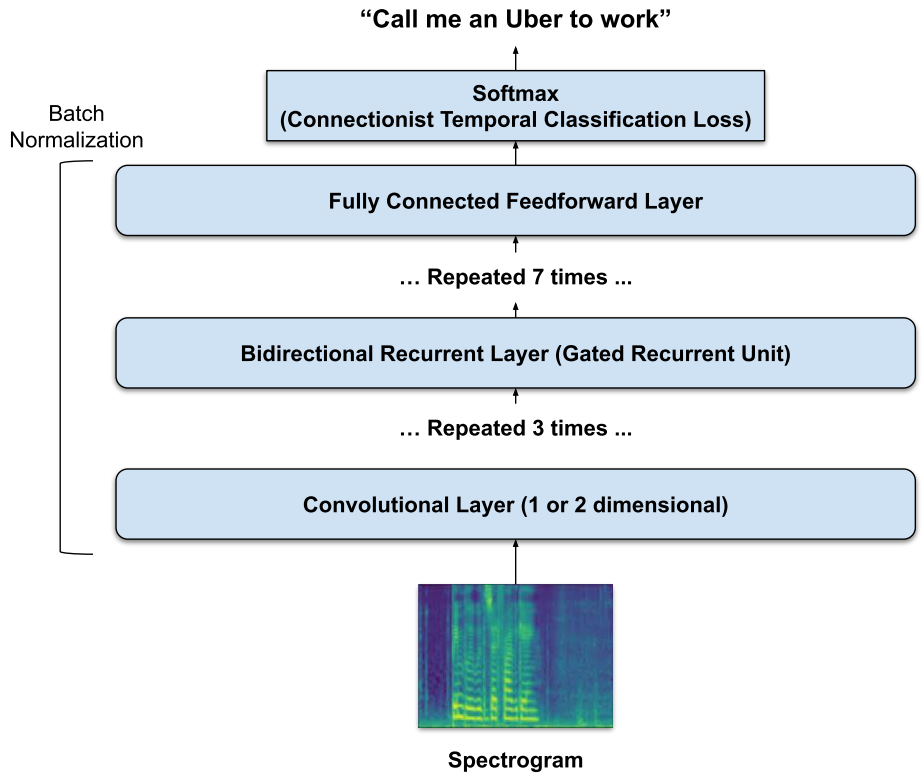
The neural vocoder is responsible for taking the generated spectrogram and creating an audio signal. A target audio signal along with the spectrogram associated with the audio are used to train the vocoder. During inference, the goal of the vocoder is the be able to synthesize an audio signal given just a spectrogram (here generated by the previous model). Architecturally, the

spectrogram is first fed into a transpose convolution, sometimes called a deconvolution because it does the same thing as a convolution but in reverse. This effectively upsamples the spectrogram allowing us to match the dimensions of the target audio signal input. The target audio signal is sent through a 1 by 1 convolutional layer to add channels to the signal. It doesn't change the signal dimensions apart from the depth. These added channels are sometimes called residual channels. The convolved audio signal is then sent through a dilated convolutional layer which does the same thing as a normal convolution except that there are gaps in the values considered for convolution. Dilated convolutions allow for a greater receptive field while maintaining the same number of parameters. The output of the dilated convolution is added to the output of the transpose convolution. This sum is then fed separately into a tanh and a sigmoid. This creates a 'gating' mechanism similar to what is seen in an LSTM. The output of the gate is used twice. Once for being fed into a subsequent layer, and the other for being added to the original target audio signal to then be used as the second input to a subsequent layer. Here the subsequent layer is a residual layer which consists of the the same components discussed in this paragraph. A series of 30 residual layer outputs are summed up, sent through a series of ReLUs and 1 by 1 convolutions to be finally sent through a mixture of logistic distributions (MoL). The MoL allows us to predict the mean and standard deviation of a logistic distribution from which we select the most probable value distribution. A mixture of logistic distributions is used (here 10) for the cases in which the true distribution does not exactly follow a logistic. The mixture can roughly approximate many more distributions. The most probable value from the mixture of these distributions is selected as the value for the synthesized audio at that timestep. Since each step of the audio is represented with 16 bits, we approximate the value produced by the MoL to the nearest 16-bit number. Since our audio sample rate is 48 kHz, we'll use this TTS engine to synthesize 48000 values per second of audio.

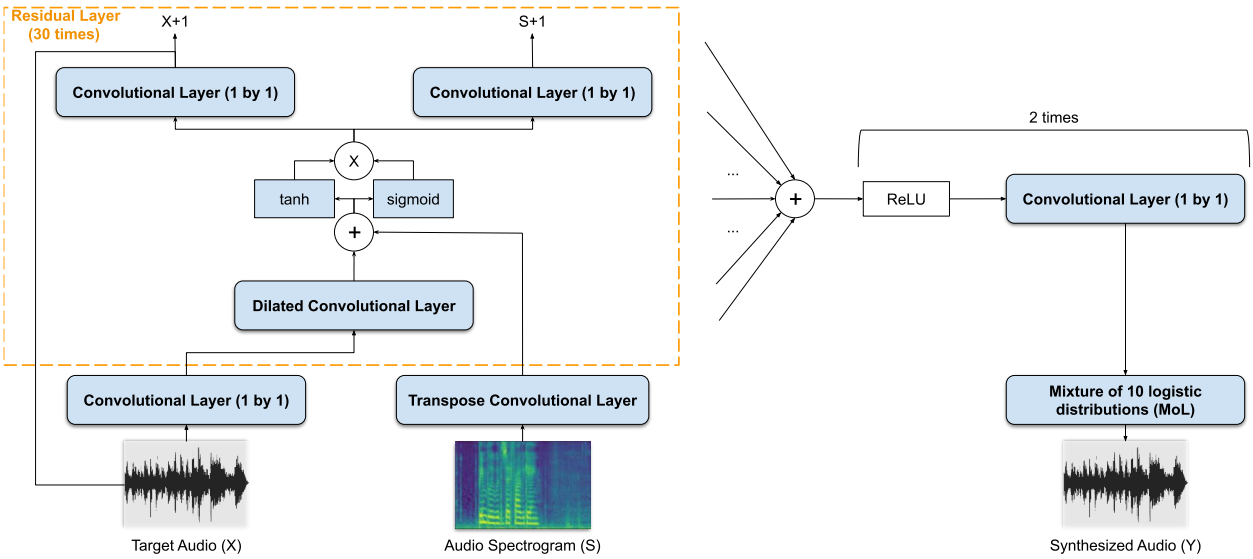
7. KWS Diagram



8. ASR Diagram



9. TTS Neural Vocoder Diagram



10. TTS Spectrogram Prediction Diagram