Collecting training data to train an LSTM to classify a finite number of human intents from voice with slight negative latency, and solving the use-mention distinction

1st Isaac Leonard  
Applied Sciences  
IBM Watson and Cloud Platform  
ifleonar@us.ibm.com

2nd Chris Dye  
Applied Sciences  
IBM Watson and Cloud Platform  
dyec@us.ibm.com

Abstract—It is desirable to have a model which takes as input a stream of audio and returns the action which the user is currently telling the machine to perform. Given a good set of labeled audio, it is relatively easy to train a Long Short-Term Memory (LSTM) model to do this. However obtaining such a set of labeled audio is difficult. We present Aural2, a data collection and labeling infrastructure which helps users to quickly collect high value training data with which it trains an LSTM model to accurately transform a stream of audio into the probability, for any given action, that the user is currently telling Aural2 to perform it. The models trained by Aural2 are usually capable of correctly classifying the user’s intent before the user has finished speaking; assuming latency to be measured from end of utterance, they have slight negative latency. Furthermore, the model will learn to integrate past context into its classifications, allowing it to learn to accurately solve the use-mention distinction, and ignore commands directed at other voice assistants, all without itself requiring the use of wake-words.

We describe the architecture of Aural2, its advantages over existing systems, its failings, and future directions for development.

Index Terms—machine learning, artificial neural networks, LSTM, speech recognition, training data collection

I. INTRODUCTION

It is often useful for users to be able to control machines via voice. To do this, we need a model that takes a real-time stream of audio and returns the action which the user wishes the machine to perform. There exist many systems which perform this task [1] [2] [3]. Most of these systems first transcribe the audio into text using full vocabulary speech to text (STT), and then use word level natural language parsing (NLP) systems on the resulting text [4] [5]. NLP on top of STT has numerous advantages. Behaviors may be written purely in text, making development of new skills easy. Assuming sufficiently advanced NLP, arbitrarily complex instructions may be transmitted from user to machine. Even if the NLP used is not sufficiently powerful to understand the full complexity of human language, it can still understand multidimensional commands [4].

However, such systems fall short in certain respects. Text contains less information than the audio from which it was transcribed. STT loses important information regarding inflection and tone. Purely from an information-theoretic perspective, we should expect a system based on NLP on top of STT to be less accurate than a system which processes sound into actions directly.

Even the most advanced NLP systems are unable to reliably differentiate between the mention of a word and its use, especially in context-poor situations. To prevent spurious actions, most voice control systems require that the user speak a predefined wake word not found in normal language before each command.

Both STT and NLP require significant data to train and hardware to run. This makes them difficult to run on resource-constrained devices, requiring that audio be sent to more powerful hardware in the cloud for processing. Constant streaming of audio is costly in bandwidth and is a potential privacy risk, not to mention that it incurs at least one round trip of latency to the nearest data center. To minimize the audio sent to the cloud, most voice control systems require that the user speak a locally detectable wake word to trigger audio transmission only when needed.

Both because current NLP is unable to solve the use-mention distinction with sufficient accuracy, and because STT and NLP are too expensive in RAM and CPU to run on inexpensive hardware, most current systems must require users to speak a wake word which is both uncommon in normal speech and can be detected by the small amount of local audio analytics before speaking any command. Coupled with the latency incurred by data transfer to and from the cloud, traditional voice interfaces are painfully slow in comparison to in-person interaction with a human.

A system built around Aural2 can run on relatively inexpensive hardware and can solve the use-mention distinction with far better accuracy than most voice assistants, thereby allowing it to be used with no wake word. It not only avoids the latency of network communication, but also correctly classifies the
user’s intent somewhat before the user has finished speaking, compensating for whatever other latency may be unavoidable in the system.

However, to achieve these advantages, Aural2 must make significant sacrifices in other areas such as number and complexity of actions, making it unsuitable for many applications.

II. TECHNOLOGIES USED

A. Tensorflow Compute Graph (TF GraphDef)

Tensorflow (TF) Compute Graph is a purely functional language for defining graphs of transformations on tensors, which the Tensorflow runtime lazily evaluates using the best hardware available at runtime. Each node in the graph takes zero or more tensors as input, and returns one or more tensors as output. As recursion and loops are forbidden, TF compute graphs are provably halting, execute in approximately fixed time, and are not Turing complete. A TF graph can be stored as a GraphDef protobuf file. This GraphDef is cross-platform, able to be evaluated by the Tensorflow runtime on any supported hardware, whether that be an x86 or ARM CPU, or NVIDIA GPU.

A tensor in this context is an n-dimensional array of numbers. The shape of a tensor is denoted with a list of its dimensions. For example, [1, 2, 3] denotes a three-dimensional tensor: a list containing one list of two lists of three numbers. All numbers are float32 unless otherwise noted.

Although it is possible to write a text-encoded GraphDef by hand, it is far more common to construct the graph using some more general purpose language such as Python or Golang, either for export as a GraphDef for later use, or for immediate evaluation.

Aural2 does both, constructing the main training graph in Python at build time, and the numerous other supporting graphs in Golang at initialization time.

Aural2’s extensive use of TF compute graphs allow it to take advantage of dedicated hardware accelerators such as NVIDIA GPUs, while still running with full capabilities, albeit somewhat slower, on a generic CPU.

B. Mel-frequency cepstral coefficient (MFCC)

It is computationally expensive to train a neural net (NN) directly on the waveform of audio [6]. Therefore, it is common practice to train the NN on fingerprints of windows of the waveform [7]. A Fourier transform of a window of audio reduces it to a list of amplitudes in the frequency domain. Mel-frequency cepstral coefficient (MFCC) remaps the frequency domain information produced by a Fourier transform to a scale optimized for human speech. In the configuration used in Aural2, MFCC uses 13 frequency bins each represented as a float32.

C. Long short-term memory neural nets (LSTM)

An LSTM can be thought of as a Recurrent Neural Net (RNN) augmented with persistent memory [8].

Each cell of an RNN takes as input the output of the previous cell concatenated with the input information and returns an output which is sent to the next cell. This allows the RNN to recognise patterns in time-series data of arbitrary length. However, for various reasons, RNNs have difficulty remembering long-term state. LSTMs solve this problem by augmenting an RNN with persistent memory which it can read from and write to, thereby allowing it to persist information for arbitrarily long periods.

Aural2 currently uses a stack of two LSTMs, the first taking as input a series of MFCCs of audio, and the second taking the output of the first and producing an embedding of the user’s intent. Both LSTMs have a state of size 64. As each LSTM is passing forward both the RNN’s information and the state of its memory, a total of 256 float32s are being passed forward in each iteration.

III. ARCHITECTURE

A. Elements

- **Microphone**: Measures changes in air pressure. Returns a stream of 16,000 Hz int16 audio.
- **Step MFCC**: A TF graph which takes 1024 bytes (512 samples, or ~32ms) of audio, and returns the [13] floats of an MFCC.
- **Step Inference LSTM**: Every ~32ms, takes an MFCC from Step MFCC, updates the [256] memory of the LSTM accordingly, and returns [50] softmaxed probabilities for each intent.
- **vsh**: Takes the list of 50 intent probabilities from the Step inference graph, and, if the threshold is reached, triggers an action.
- **10-second ring buff**: Takes audio from the microphone, and stores it for 10 seconds. When triggered by vsh, writes the past 10 seconds of audio to the file system.
- **Raw audio on FS**: Stores 10-second clips of raw audio given to it by the ring buffer.
- **Labeling UI**: Displays each clip to the user, who labels durations within which a non-nil intent was expressed.
- **MFCC Graph**: Takes a whole 10-second clip of audio and returns the [312, 13] MFCCs for the whole clip.
- **Map of MFCCs**: Stores slices of 312 MFCCs.
B. Hyperparameters

- **Sequence length**: The number of LSTM cells to unroll when training. Currently 100, which at current window size is ~3 seconds. Higher values mean slower build and training. Shorter values cost ability to learn long sequences, and consequently slower gradient descent along with less training overhead, but with greater compute cost per mini-batch.

- **Clip duration**: The duration of each saved clip of audio. Currently 10 seconds. Must be somewhat longer than the duration of the longest command + the time needed for the user to recognize a mistake and tell Aural2 to save audio. The user must listen to the whole clip when labeling, so excessively long clips waste the user’s time.

- **Window size**: The number of samples of audio to use in each MFCC. Currently 512 samples. Smaller windows will increase time resolution, and therefore compute cost, at some (probably small) cost to the model’s ability to learn long term patterns. Extremely small windows will cost low-frequency accuracy.

- **MFCC size**: The number of bins in the MFCC. Currently 13 because that was the default in the TensorFlow implementation. Larger values will get better frequency resolution at some cost to model size and compute cost.

- **Mini-batch size**: The number of labeled sequences on which to compute loss when training. Currently 7. Larger values will result in better accuracy in loss calculation, and consequently smoother gradient descent along with less training overhead, but with greater compute cost per mini-batch.

- **LSTM state size**: The number of float32s in which the LSTM stores its state. Currently 64. Larger values allow the model to remember more complex state while requiring more memory and compute. Extremely large values may allow overfitting.

- **Label sets in DB**: Stores sets of user-created labels for each clip.

- **Sampler**: Converts a set of periods of intent into a list of 312 integer intent IDs.

- **Map of Targets**: Stores slices of intent IDs.

- **Mini-batch generator**: Takes corresponding slices from corresponding elements from the map of the MFCCs, and the map of targets, and combines them into a ([7, 100, 13], [7, 100]) mini-batch, which is written to the buffered channel.

- **Buffered Channel**: Holds a buffer of, at most, 3 mini-batches.

- **Train LSTM**: Reads mini-batches from the buffered channel, computes loss on the current state of the variables, and backpropagates this loss over the variables.

Note that the step inference and training LSTM graphs share weight and bias variables.

Not shown or discussed are various TF graphs for generating visualizations of data.

C. LSTM Model

Aural2 uses a fairly standard stacked LSTM. Sound is recorded at a sample rate of 16,000 Hz with 16-bit depth. 512-sample windows of this audio are fed into a TF graph to compute the MFCC, producing a tensor of shape [13]. This tensor is used as the input to the first LSTM. This LSTM produces an output of the same size as its state which is used as the input for the second LSTM. The output of the second LSTM is matmul-ed, and softmax-ed into a list of 50 floats between 0 and 1. The nth element of this list is the probability that the world is in state n. As the world can be in one and only one state, the probabilities of the various states must sum to 1.

D. Loss

When training, we try to minimize the loss of the model. In this context, loss is calculated as the sum of the square of the differences between actual and target.

Recall that the actual and target are lists of one-hot embedded states. Take the example of a vocabulary of three states. Looking at only one timestep, if the target is [0.0, 1.0, 0.0], and the actual is [0.6, 0.3, 0.1], the loss is:

\[
(0.0 - 0.6)^2 + (1.0 - 0.3)^2 + (0.0 - 0.1)^2 =
-0.6^2 + 0.7^2 + 0.1^2 =
0.36 + 0.49 + 0.01 = 0.86
\]

We see that a model can minimize its loss by reducing the differences between its actual output and the target. Due to the nonlinearity of squaring, large differences in one element of the output are punished disproportionately. The model is therefore cautious in the absence of good information. If the information which the model possesses is of approximately equal likelihood to be observed in worlds which are in each of the three states, a good model will output ~0.33 for all outputs so as to minimize loss.

E. Causality

Let us assume a world which can be in any of three states: 0, 1, or 2. At each timestep, it can change its state. However most of the time, its state does not change. We can model this world as a Markov chain where the strength of each node’s connection to itself is at least an order of magnitude greater than the strength of its connections to other nodes. We cannot directly observe the state of the world. However, there exist a few bits of information whose states at each timestep occur with different frequency and in different patterns depending on the past and present state of the world. This information we can observe.

A single timestep of information often appears in multiple states with similar frequency; it gives us only weak evidence of the state our world is currently in. From a single timestep, we cannot reduce the list of states which we could be living in to one. We must observe the information for multiple timesteps if
we are to confidently know the exact world we live in. When
the world has been in state \(a\) for many past timesteps, we have
a strong prior of the world being in state \(a\). The absence of
strong evidence of not being in state \(a\) is sufficient to give us a
reasonably strong belief that the world is in state \(a\). However,
if the state changes, that is, if we have strong evidence that
the world is no longer in state \(a\), we no longer have good
priors, and must observe many timesteps of information to
gain strong beliefs about the current state.

Imagine a simple LSTM. For the past many steps it has been
observing patterns of information which occur with far greater
frequency in worlds of state 0. Assume that the probability
of the world transitioning from state 0 to state 0 is 0.9, and
probabilities of state 0 transitioning to states 1 and 2 are
0.05. Given that the world is in state 0, there is a high prior
probability that it will be in state 0 in the next timestep.
But if the LSTM observes information which is very rarely
observed when in worlds of state 0, this is evidence sufficient
to overcome the strong prior probability and stop believing that
it is in a world of state 0. However this new information is
often observed with approximately equal frequency in worlds
of state 1 and state 2. Although the LSTM knows that it is
not in state 0, it does not know whether it is in state 1 or 2.
Recall that loss is calculated as the square of the difference
between the model’s outputs and the true state of the world.
The LSTM will therefore output probabilities of, for example,
0.06 for state 0, and probabilities of 0.47 for 1 and 2. As
it observes new information, it updates the probabilities that
it is in worlds 1 and 2. After observing many timesteps of
information more likely to be observed when in, for example,
state 2, it will once again assign a probability of, for example,
0.96 to being in state 2, and assign a high prior probability to
being in state 2 next iteration.

**F. vsh**

The LSTM model outputs a list of probabilities of intents
every 32ms. While far more useful than before, this format
of information is still far from maximally convenient for our
purposes. The simple preprocessing layer used by Aural2 is
called Voice SHEll, or vsh. It is quite primitive and may well
be replaced by some more powerful system in future. However
it has several important features necessary to make good use
of the output of the LSTM.

An LSTM trained by Aural2 tries to reduce loss. A model
which assigns high confidence to only the second part of the
utterance “Play” being the PlayMusic intent has greater
loss than a model which assigns high probability to every
part of the utterance being the PlayMusic intent. A model
which assigns a probability of 0.6 to the utterance “Play”
being the PlayMusic intent has greater loss than a model
which assigns a probability of 0.9 to the utterance being the
PlayMusic intent. The model wants to classify the whole
utterance with high probability. These same principles work
in reverse.

As a result of this, when the user first begins to utter a
command, the LSTM will output increased probabilities for
the several intents which the user could be in the process
of uttering. Because the output probabilities must sum to 1,
the several possible outputs will all individually be fairly
low; beginning to utter a command causes the maximum
probability to fall. However, as the user utters more of the
command, the LSTM will receive patterns of information
which are frequently observed in worlds in which the user is
commanding the machine to play music, and are very rarely
observed in worlds in which the user is commanding the
machine to prune the shrubbery. With time, the probability for
the PlayMusic intent will approach 1 while the probability
for the other intents will approach 0.

Once the user is finished with their command and the world
changes back to the nil state, the LSTM must again update its
beliefs, this time reverting back to outputting close to 1 for
the nil intent, and close to 0 for all others.

All of this takes place in the few hundred milliseconds
during which the user is saying the word “Play”.

vsh allows event handler functions to be registered for
each of the outputs. When an output is greater than the
specified threshold, it is called. A perfect LSTM will fire the
PlayMusic event every 32ms for the entire duration of the
utterance. While causing music to start playing several times
will likely not cause significant harm, other intents are not safe
to be called multiple times in quick succession. Therefore,
the primary task of vsh is to transform the LSTM’s classification
of the intent which the user is currently expressing, into single
events which fire only once for each command. This is easily
achieve by the use of a boolean variable for each output which
is set to true when the upper threshold is reached and the
event handler is called, and false when the output falls
below some lower threshold. vsh then need only check for
this variable, and call the event handler only if the variable is
false.

In this way, vsh watches the output of the LSTM every 32ms
and calls the appropriate event handler exactly once each time
the user utters a command. And as a side effect of the LSTM’s
attempt to classify the whole duration of the utterance, vsh
calls the event handler after only a small portion of the
utterance has been spoken. Thus, while information can not
travel backwards in time, still, vsh on top of an LSTM trained
on data labeled via Aural2 will take the appropriate action well
before the user has finished speaking their command; if latency
is measured from the end of the utterance, it is negative.

**IV. TRAINING**

**A. Desired behavior**

Every 32ms, the LSTM is given information about the
current frequency distribution of incoming sounds. We want it
to return the probabilities that the world is in each of a finite
set of possible states.

When we speak of world state in this context, we usually
mean states such as “The user is currently telling the machine
to play the music”, or “The user is not telling the machine
to do anything”. However Aural2 is extensible to any set of
world states about which audio frequency distribution gives information.

As an example, imagine a vocabulary consisting of the states nil, PlayMusic, SaveClip, and PruneShrubbery, with indexes 0, 1, 2, and 3 respectively. Imagine that, to begin with, the user is silent. Every 32ms, the MFCC containing information about the world is given to the LSTM which writes it to its memory. While the user is silent, we want the MFCC to output a value close to 1 for the nil state, and a value close to 0 for every other state. Now imagine that the user wants the machine to play music, and therefore starts to say the word “play”. The LSTM will receive a few MFCCs containing the white noise which begins the word “play”. This is important information about the intent which the user is currently expressing. As the past few seconds contained no utterances, the current utterance is probably using the word, not just mentioning it. However there are multiple intents which, when expressed, begin with white noise. Given the information available, the model may not know if the user is expressing the “PlayMusic” intent, or the “PruneShrubbery” intent, or if the white noise was produced by some other source. We would therefore like it to output significant, but still less than 0.5, values for states 0, 1, and 3.

Each time a 32-millisecond window passes, a new MFCC is given to the LSTM with new information about the state of the world. A new MFCC arrives, and this time, it does not contain much white noise and instead the frequency distribution looks quite like that of the “I” phoneme. Assuming that the only intent which starts with a few steps of white noise followed by the “I” phoneme is the PlayMusic intent, we would like the LSTM to start outputting a value close to 1 for state 1 of its output. At this point, vsh, having received the PlayMusic intent with greater than 0.9 confidence, will start to play the music. As the user continues speaking the word “play”, we want the LSTM to continue to output a high value for state 1. But as soon as the LSTM receives an MFCC of silence, we want it to go back to outputting a value close to 1 for state 0, and close to 0 for every other state.

This is the behavior which we would like the LSTM to exhibit. While it would perhaps be possible to code such behavior by hand, it would be tedious and likely inaccurate.

Instead, we train the LSTM on pairs of lists of input MFCCs and corresponding correct states. To do this, we must first collect audio rich in interesting states, and then annotate this audio with correct state information.

B. Data Collection

As mentioned before, Aural2 maintains a ring buffer of the past 10 seconds of audio. At any time, the past 10 seconds can be written to disk. This may be triggered by a REST API, or by the user saying “upload”, “mistake”, or otherwise expressing their intent that the audio should be saved. Clips of raw audio are stored as files in a directory on the local storage. Metadata about the raw audio clip is stored in a local boltDB or other key/value database.

C. Labeling

Neural nets transform one dataset into another dataset. To use supervised learning to train the neural net to turn the input data into the correct output data, we must give Aural2 many examples of the correct outputs for the various inputs. To help the user create this information, Aural2 provides a web-based labeling UI.

Aural2 serves an index page listing the audio clips which have been captured, each clip name linking to the labeling UI for that clip. The labeling UI for a given clip contains various visualizations of the clip and the labels made by the current model when processing it. The visualizations of the labels are automatically reloaded every ~1 second, allowing users to watch the output of the model change in real time. The user may listen to the audio, see visualizations of it, and create labels defining the beginning and end of each period of any given state. The labeling procedure is as follows. The audio may be played or paused via the space bar, and moved in small increments via arrow keys. As the audio plays, the cursor moves across the visualization of FFT, MFCC, LSTM state, and current classification. To create a label, the user need only hold down the key corresponding to the correct state while the audio of that state is playing. We find that, with practice, it is possible to label audio in only slightly longer than real time. Once the user has labeled all periods in the clip during which a non-nil intent was being expressed, the label set is submitted back to the server which both writes it to the local DB, and adds both the label set and the corresponding audio clip to the training data object.

![Fig. 2. Labeling UI and audio visualizations](image)

D. Training

The training data object contains two maps, one of inputs and one of targets, where each input is of shape [312, 13] and type float32 and each target is of shape [312] and type int32.

When Aural2 starts, it reads the list of label sets from the boltDB and the audio clips to which they refer, and adds them to the training data object.

When a clip and label set are added to the training data object, the clip is fed into the Clip MFCC TF graph to transform it to a tensor of shape [312, 13], which is added to the Inputs Map, and the label sets are transformed into a
list of the integer state ID at each of its 312 timesteps, which is added to the Targets Map.

In this way, a set of preprocessed inputs and targets is created from existing training data on startup, and added to when new labels are submitted.

However, these are sequences with a length of 312 steps. To train an LSTM on sequences of \( n \) timesteps, one must unroll the cells, creating a training graph containing \( n \)-copies of the LSTM cell. It is computationally expensive and unnecessary to train on 312-sample sequences. Aural2 currently trains on 100-sample sequences.

The data preparation loop is as follows:
- From the set of labeled audio clips, randomly select 7.
- For each clip:
  - At random, select two positive integers such that the second is less than 312, and exactly 100 more than the first.
  - From both the inputs and the targets for this clip, take the slice of 100 timesteps defined by the two numbers previously selected.
- We now have 7 inputs of 100 time slices, and 7 corresponding targets of 100 time slices.
- Convert these two sets to tensors: a float32 input tensor of shape \([7, 100, 13]\) and an int32 target tensor of shape \([7, 100]\).
- Write this input-target pair to the minibatch channel, blocking until there are fewer than three mini-batches in the channel.

The training loop reads a mini-batch from the mini-batch channel and evaluates the training LSTM graph on the inputs and targets, thereby updating the weights and biases.

In this way, the training loop is always supplied with a buffer of mini-batches randomly drawn from a recent state of the training data, and training data preparation is free to fully utilize CPU resources while the training loop is blocked by the GPU doing training.

It should be reiterated that the graph used for performing inference on incoming audio, the graph for performing batch inference on whole audio clips for visualizations, and the training graph, while distinct graphs, share a single set of variables. The weight and bias variables are updated by the training graph, and the accuracy with which Aural2 classifies the state of the world increases. The variables stay on the GPU, or whatever compute device TensorFlow has decided to use, and need never leave. TensorFlow transparently converts any command which they are capable of detecting locally is the wake word; for all other speech processing, they must send audio to the cloud and receive the results. This is a privacy issue as well as incurring high latency. Contrast this behavior to Aural2 which processes all audio locally, and can therefore respond in time limited only by local compute speeds. As the LSTM used by Aural2 is trained to label the whole duration of the utterance, it will usually begin outputting the intent before the utterance is finished; a whole utterance is not required to correctly classify an intent.

As Aural2 is trained locally, its model is trained on the user’s voice recorded from actual usage. There is no need to collect a diverse training set; what would in any other model be undesirable overfitting to a single user or small group of users is Aural2’s correct and non-problematic behavior.

An advantage of sending audio to the cloud for conversion to an intent is that it allows such a system to use models running on servers in the cloud, which can be far larger than any model suitable for running on inexpensive edge devices.

Additionally, requiring a wake word has various advantages. As good as Aural2’s use-mention distinction is, it will eventually make a mistake. However, if additional safety is desired, it is simple to train Aural2 to require that some prefix, “sudo” for example, be said before particularly dangerous intents. Aural2 need only be taught that when the user says “format disk”, the user does not intend that the disk be formatted, whereas when the user says “sudo format disk”, the user does want the disk to be formatted. Aural2 will usually learn the distinction with a few examples. For less dangerous intents, no prefix need be required. In this way, the benefits of requiring a wake word may be easily gained when desired, while still preserving the ease of non-prefixed commands where occasional false positives are not so harmful.

For example, because the collection of training data is triggered by the \texttt{saveAudio} intent, all clips of audio contain an utterance which some past state of the model thought was the user telling it to save audio. If the model is well trained, it will only contain true examples, but badly trained models will collect many false positives. Because of this bias, the model quickly approaches 0 false positives for the \texttt{saveAudio} intent.

There is no standard training or test set for Aural2; each user is encouraged to generate their own training set, and to keep saving and labeling clips until Aural2 stops making too many mistakes. It is therefore difficult to provide numbers describing Aural2’s accuracy in a reproducible manner.

B. Comparison to alternatives

The closest systems to which we can compare Aural2 are Google Home and Amazon Echo. Like Aural2, Google Home and Amazon Echo respond to voice commands to perform such actions as playing or pausing music. However, unlike Aural2, they are unable to perform use-mention differentiation and therefore require the use of a wake word not common in everyday speech. Additionally, the only command which they can respond to is the wake word; for all other speech processing, they must send audio to the cloud and receive the results. This is a privacy issue as well as incurring high latency. Contrast this behavior to Aural2 which processes all audio locally, and can therefore respond in time limited only by local compute speeds. As the LSTM used by Aural2 is trained to label the whole duration of the utterance, it will usually begin outputting the intent before the utterance is finished; a whole utterance is not required to correctly classify an intent.

As Aural2 is trained locally, its model is trained on the user’s voice recorded from actual usage. There is no need to collect a diverse training set; what would in any other model be undesirable overfitting to a single user or small group of users is Aural2’s correct and non-problematic behavior.

An advantage of sending audio to the cloud for conversion to an intent is that it allows such a system to use models running on servers in the cloud, which can be far larger than any model suitable for running on inexpensive edge devices.

Additionally, requiring a wake word has various advantages. As good as Aural2’s use-mention distinction is, it will eventually make a mistake. However, if additional safety is desired, it is simple to train Aural2 to require that some prefix, “sudo” for example, be said before particularly dangerous intents. Aural2 need only be taught that when the user says “format disk”, the user does not intend that the disk be formatted, whereas when the user says “sudo format disk”, the user does want the disk to be formatted. Aural2 will usually learn the distinction with a few examples. For less dangerous intents, no prefix need be required. In this way, the benefits of requiring a wake word may be easily gained when desired, while still preserving the ease of non-prefixed commands where occasional false positives are not so harmful.

V. RESULTS

A. Quantitative performance

The neural net used by Aural2 is continuously trained in real time from a dynamically collected training set. The distribution of data added to the training set changes depending on the environment and user, which depend on the state of the Aural2 model.
C. Coexistence with other voice assistants

We find that, from a few minutes of training data, Aural2 is well able to learn to ignore commands directed at other voice assistants. Google Home and Amazon Echo can therefore be used in parallel with Aural2. Should the user wish, they may use a Google Home device, an Amazon Echo device and an instance of Aural2 to control distinct streams of music simultaneously.

We invite readers to try Aural2 so as to evaluate for themselves.

D. Shortcomings

Although perhaps superior to existing technology in latency, simplicity, and speed of training, Aural2 has several severe shortcomings. It currently uses one-hot embedding which scales linearly with number of outputs. While the currently used embedding size of 50 intents is ample for many tasks such as controlling music, interacting with simple toys, or as a safety stop for industrial equipment, it is likely impractical for many thousands of outputs, and as such, would be unsuitable for use in full-vocabulary natural language parsing.

Additionally, Aural2’s training data labeling leaves much to be desired. While it saves audio on command, this only helps collect unlabeled audio rich in states which the user thought a past state of the model had misclassified; it does nothing to label the audio with the true state.

Aural2 must run on a single machine. If that machine fails, training data and model parameters are lost. While it is possible to back up training data and models, this is inconvenient.

Likewise, Aural2 must both train and infer on the same machine. It is not possible to train the model on a powerful desktop while simultaneously running inference on a cheap ARM single-board computer (SBC). The only possibility is to copy all training data from the SBC to the desktop, train, and then copy both model and training data back to the SBC. This is inconvenient and error-prone.

VI. FUTURE DIRECTIONS

While useful as a proof of concept, Aural2 underwent various major design changes during its development and is therefore both poorly designed, badly implemented, and not fit for any serious use. Early development is now underway on Aural3 (name subject to change) which is intended to address many of Aural2’s failings and introduce various additional features.

While it is difficult to predict the exact features of Aural3, they may include:

- Heterogeneous clustering: Often we have multiple cheap SBCs with microphones as well as far more powerful machines such as desktop computers or even GPU-equipped POWER servers. If all of these heterogeneous devices are to usefully contribute to the function of a single instance of Aural3, all nodes in the cluster, from the smallest SBC to the greatest POWER9 server, must maintain a single eventually consistent state of the model parameters, the SBCs to perform inference and data selection, and the more powerful GPU machines to perform training to update the model parameters. This would allow inference and training nodes to leave the cluster without warning yet not impact intent detection. This would facilitate zero-downtime upgrades.
- Teacher network: A bidirectional RNN can use information from both the past and future to label state. It can therefore generally achieve significantly greater accuracy then a unidirectional RNN can. It has the disadvantage however of requiring information from the future and therefore being unable to run in real time. It is however suitable for use as a teacher model [9] to automatically label large amounts of training data with which to train the unidirectional student model. This would allow Aural3 to improve the accuracy of its single-layer model even when no new human-labeled data was created. Additionally, as users generally behave differently in worlds in which their intents were correctly fulfilled and worlds in which they were not, provided that the teacher model can observe the action which was actually taken, it should be able to learn to use user response to better label the user’s words in hindsight. This could potentially grant many of the advantages of reinforcement learning while avoiding many of its difficulties.
- Multidimensional intents: Often the user wishes some action to be performed on some entity. Sometimes actions take several independent parameters. It would therefore be good for the output of Aural3 to be, not unidimensional as it is in Aural2, but n-dimensional, or at the least two-dimensional, one output for intent and one for entity. As an additional refinement, it may be useful for Aural3 to elicit additional information from the user if they fail to sufficiently disambiguate all dimensions. In this sense, it can be thought of as not merely passively observing the user but also actively running experiments on the user to maximize expected information gain.
- Multimodal input: The exclusive use of audio to gain information about the state of the world is not fundamental to the architecture of Aural2 or Aural3. It should be relatively simple to add additional channels of information such as time of day or a camera.

REFERENCES

[1] https://store.google.com/product/google_home