Simple Audio Recognition

This tutorial will show you how to build a basic speech recognition network that recognizes ten different words. It's important to know that real speech and audio recognition systems are much more complex, but like MNIST for images, it should give you a basic understanding of the techniques involved. Once you've completed this tutorial, you'll have a model that tries to classify a one second audio clip as either silence, an unknown word, "yes", "no", "up", "down", "left", "right", "on", "off", "stop", or "go". You'll also be able to take this model and run it in an Android application.

Preparation

You should make sure you have TensorFlow installed, and since the script downloads over 1GB of training data, you'll need a good internet connection and enough free space on your machine. The training process itself can take several hours, so make sure you have a machine available for that long.

Training

To begin the training process, go to the TensorFlow source tree and run:

```
python tensorflow/examples/speech_commands/train.py
```

The script will start off by downloading the Speech Commands dataset (https://storage.cloud.google.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz), which consists of 65,000 WAVE audio files of people saying thirty different words. This data was collected by Google and released under a CC BY license, and you can help improve it by contributing five minutes of your own voice (https://aiyprojects.withgoogle.com/open_speech_recording). The archive is over 1GB, so this part may take a while, but you should see progress logs, and once it's been downloaded once you won't need to do this step again.

Once the downloading has completed, you'll see logging information that looks like this:

```
I0730 16:53:44.766740   55030 train.py:176] Training from step: 1
I0730 16:53:47.289078   55030 train.py:217] Step #1: rate 0.001000, accuracy 7.0%
```

This shows that the initialization process is done and the training loop has begun. You'll see that it outputs information for every training step. Here's a break down of what it means:

`Step #1` shows that we're on the first step of the training loop. In this case there are going to be 18,000 steps in total, so you can look at the step number to get an idea of how close it is to finishing.

`rate 0.001000` is the learning rate that's controlling the speed of the network's weight updates. Early on this is a comparatively high number (0.001), but for later training cycles it will be reduced 10x, to 0.0001.

`accuracy 7.0%` is the how many classes were correctly predicted on this training step. This value will often fluctuate a lot, but should increase on average as training progresses. The model outputs an array of numbers, one for each label, and each number is the predicted likelihood of the input being that class. The predicted label is picked by choosing the entry with the highest score. The scores are always between zero and one, with higher values representing more confidence in the result.

`cross entropy 2.611571` is the result of the loss function that we're using to guide the training process. This is a score that's obtained by comparing the vector of scores from the current training run to the correct labels, and this should trend downwards during training.

After a hundred steps, you should see a line like this:

`I0730 16:54:41.813438 55030 train.py:252] Saving to "/tmp/speech_commands_train/conv.ckpt-100"`

This is saving out the current trained weights to a checkpoint file. If your training script gets interrupted, you can look for the last saved checkpoint and then restart the script with `--start_checkpoint=/tmp/speech_commands_train/conv.ckpt-100` as a command line argument to start from that point.

Confusion Matrix

After four hundred steps, this information will be logged:

```
I0730 16:57:38.073667   55030 train.py:243] Confusion Matrix:
 [[258   0   0   0   0   0   0   0   0   0   0   0]
 [  7   6  26  94   7  49   1  15  40   2   0  11]
 [ 10   1 107  80  13  22   0  13  10   1   0   4]
 [  1   3  16 163   6  48   0   5  10   1   0  17]
 [ 15   1  17 114  55  13   0   9  22   5   0   9]
 [  1   1   6  97   3  87   1  12  46   0   0  10]
 [  8   6  86  84  13  24   1   9   9   1   0   6]
 [  9   3  32 112   9  26   1  36  19   0   0   9]
 [  8   2  12  94   9  52   0   6  72   0   0   2]
 [ 16   1  39  74  29  42   0   6  37   9   0   3]
 [ 15   6  17  71  50  37   0   6  32   2   1   9]
 [ 11   1   6 151   5  42   0   8  16   0   0  20]]
```

The first section is a confusion matrix (https://www.tensorflow.org/api_docs/python/tf/confusion_matrix). To understand what it means, you first need to know the labels being used, which in this case are "*silence*", "*unknown*", "yes", "no", "up", "down", "left", "right", "on", "off", "stop", and "go". Each column represents a set of samples that were predicted to be each label, so the first column represents all the clips that were predicted to be silence, the second all those that were predicted to be unknown words, the third "yes", and so on.

Each row represents clips by their correct, ground truth labels. The first row is all the clips that were silence, the second clips that were unknown words, the third "yes", etc.

This matrix can be more useful than just a single accuracy score because it gives a good summary of what mistakes the network is making. In this example you can see that all of the entries in the first row are zero, apart from the initial one. Because the first row is all the clips that are actually silence, this means that none of them were mistakenly labeled as words, so we have no false negatives for silence. This shows the network is already getting pretty good at distinguishing silence from words.

If we look down the first column though, we see a lot of non-zero values. The column represents all the clips that were predicted to be silence, so positive numbers outside of the first cell are errors. This means that some clips of real spoken words are actually being predicted to be silence, so we do have quite a few false positives.

A perfect model would produce a confusion matrix where all of the entries were zero apart from a diagonal line through the center. Spotting deviations from that pattern can help you figure out how the model is most easily confused, and once you've identified the problems you can address them by adding more data or cleaning up categories.

Validation

After the confusion matrix, you should see a line like this:

```
I0730 16:57:38.073777 55030 train.py:245] Step 400: Validation accuracy = 26.3% (N=3093)
```

It's good practice to separate your data set into three categories. The largest (in this case roughly 80% of the data) is used for training the network, a smaller set (10% here, known as "validation") is reserved for evaluation of the accuracy during training, and another set (the last 10%, "testing") is used to evaluate the accuracy once after the training is complete.

The reason for this split is that there's always a danger that networks will start memorizing their inputs during training. By keeping the validation set separate, you can ensure that the model works with data it's never seen before. The testing set is an additional safeguard to make sure that you haven't just been tweaking your model in a way that happens to work for both the training and validation sets, but not a broader range of inputs.

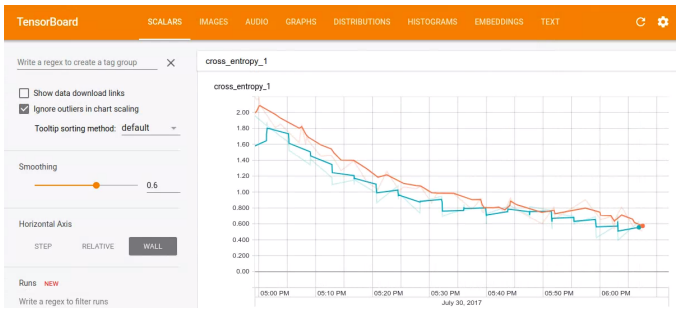
The training script automatically separates the data set into these three categories, and the logging line above shows the accuracy of model when run on the validation set. Ideally, this should stick fairly close to the training accuracy. If the training accuracy increases but the validation doesn't, that's a sign that overfitting is occurring, and your model is only learning things about the training clips, not broader patterns that generalize.

Tensorboard

A good way to visualize how the training is progressing is using Tensorboard. By default, the script saves out events to /tmp/retrain_logs, and you can load these by running:

```
tensorboard --logdir /tmp/retrain_logs
```

Then navigate to http://localhost:6006 (http://localhost:6006) in your browser, and you'll see charts and graphs showing your models progress.



Training Finished

After a few hours of training (depending on your machine's speed), the script should have completed all 18,000 steps. It will print out a final confusion matrix, along with an accuracy score, all run on the testing set. With the default settings, you should see an accuracy of between 85% and 90%.

Because audio recognition is particularly useful on mobile devices, next we'll export it to a compact format that's easy to work with on those platforms. To do that, run this command line:

```
python tensorflow/examples/speech_commands/freeze.py \
--start_checkpoint=/tmp/speech_commands_train/conv.ckpt-18000 \
--output_file=/tmp/my_frozen_graph.pb
```

Once the frozen model has been created, you can test it with the **`label_wav.py`** script, like this:

```
python tensorflow/examples/speech_commands/label_wav.py \
--graph=/tmp/my_frozen_graph.pb \
--labels=/tmp/speech_commands_train/conv_labels.txt \
--wav=/tmp/speech_dataset/left/a5d485dc_nohash_0.wav
```

This should print out three labels:

```
left (score = 0.81477)
right (score = 0.14139)
_unknown_ (score = 0.03808)
```

Hopefully "left" is the top score since that's the correct label, but since the training is random it may not for the first file you try. Experiment with some of the other .wav files in that same folder to see how well it does.

The scores are between zero and one, and higher values mean the model is more confident in its prediction.

Running the Model in an Android App

The easiest way to see how this model works in a real application is to download the prebuilt Android demo applications (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#prebuilt-components) and install them on your phone. You'll see 'TF Speech' appear in your app list, and opening it will show you the same list of action words we've just trained our model on, starting with "Yes" and "No". Once you've given the app permission to use the microphone, you should be able to try saying those words and see them highlighted in the UI when the model recognizes one of them.

You can also build this application yourself, since it's open source and available as part of the TensorFlow repository on github (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#building-in-android-studio-using-the-tensorflow-aar-from-jcenter). By default it downloads a pretrained model from tensorflow.org (http://download.tensorflow.org/models/speech_commands_v0.01.zip), but you can easily replace it with a model you've trained yourself (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#install-model-files-optional). If you do this, you'll need to make sure that the constants in the main SpeechActivity Java source file (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android/src/org/tensorflow/demo/SpeechActivity.java) like `SAMPLE_RATE` and `SAMPLE_DURATION` match any changes you've made to the defaults while training. You'll also see that there's a Java version of the RecognizeCommands module (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android/src/org/tensorflow/demo/RecognizeCommands.java) that's very similar to the C++ version in this tutorial. If you've tweaked parameters for that, you can also update them in SpeechActivity to get the same results as in your server testing.

The demo app updates its UI list of results automatically based on the labels text file you copy into assets alongside your frozen graph, which means you can easily try out different models without needing to make any code changes. You will need to updaye `LABEL_FILENAME` and `MODEL_FILENAME` to point to the files you've added if you change the paths though.

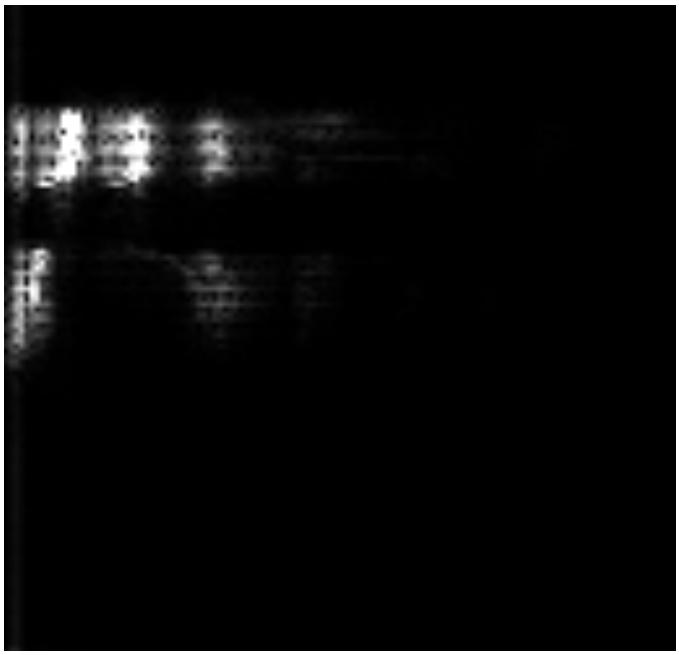
How does this Model Work?

The architecture used in this tutorial is based on some described in the paper Convolutional Neural Networks for Small-footprint Keyword Spotting (http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf). It was chosen because it's comparatively simple, quick to train, and easy to understand, rather than being state of the art. There are lots of different approaches to building neural network models to work with audio, including recurrent networks (https://svds.com/tensorflow-rnn-tutorial/) or dilated (atrous) convolutions (https://deepmind.com/blog/wavenet-generative-model-raw-audio/). This tutorial is based on the kind of convolutional network that will feel very familiar to anyone who's worked with image recognition. That may seem surprising at first though, since audio is inherently a one-dimensional continuous signal across time, not a 2D spatial problem.

We solve that issue by defining a window of time we believe our spoken words should fit into, and converting the audio signal in that window into an image. This is done by grouping the incoming audio samples into short segments, just a few milliseconds long, and calculating the strength of the frequencies across a set of bands. Each set of frequency strengths from a segment is treated as a vector of numbers, and those vectors are arranged in time order to form a two-dimensional array. This array of values can then be treated like a single-channel image, and is known as a spectrogram (https://en.wikipedia.org/wiki/Spectrogram). If you want to view what kind of image an audio sample produces, you can run the `wav_to_spectrogram tool:

```
bazel run tensorflow/examples/wav_to_spectrogram:wav_to_spectrogram -- \
--input_wav=/tmp/speech_dataset/happy/ab00c4b2_nohash_0.wav \
--output_png=/tmp/spectrogram.png
```

If you open up **`/tmp/spectrogram.png`** you should see something like this:



Because of TensorFlow's memory order, time in this image is increasing from top to bottom, with frequencies going from left to right, unlike the usual convention for spectrograms where time is left to right. You should be able to see a couple of distinct parts, with the first syllable "Ha" distinct from "ppy".

Because the human ear is more sensitive to some frequencies than others, it's been traditional in speech recognition to do further processing to this representation to turn it into a set of Mel-Frequency Cepstral Coefficients (https://en.wikipedia.org/wiki/Mel-frequency_cepstrum), or MFCCs for short. This is also a two-dimensional, one-channel representation so it can be treated like an image too. If you're targeting general sounds rather than speech you may find you can skip this step and operate directly on the spectrograms.

The image that's produced by these processing steps is then fed into a multi-layer convolutional neural network, with a fully-connected layer followed by a softmax at the end. You can see the definition of this portion in tensorflow/examples/speech_commands/models.py (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/models.py)
.

Streaming Accuracy

Most audio recognition applications need to run on a continuous stream of audio, rather than

on individual clips. A typical way to use a model in this environment is to apply it repeatedly at different offsets in time and average the results over a short window to produce a smoothed prediction. If you think of the input as an image, it's continuously scrolling along the time axis. The words we want to recognize can start at any time, so we need to take a series of snapshots to have a chance of having an alignment that captures most of the utterance in the time window we feed into the model. If we sample at a high enough rate, then we have a good chance of capturing the word in multiple windows, so averaging the results improves the overall confidence of the prediction.

For an example of how you can use your model on streaming data, you can look at test_streaming_accuracy.cc (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/). This uses the RecognizeCommands (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/recognize_commands.h) class to run through a long-form input audio, try to spot words, and compare those predictions against a ground truth list of labels and times. This makes it a good example of applying a model to a stream of audio signals over time.

You'll need a long audio file to test it against, along with labels showing where each word was spoken. If you don't want to record one yourself, you can generate some synthetic test data using the **generate_streaming_test_wav** utility. By default this will create a ten minute .wav file with words roughly every three seconds, and a text file containing the ground truth of when each word was spoken. These words are pulled from the test portion of your current dataset, mixed in with background noise. To run it, use:

```
bazel run tensorflow/examples/speech_commands:generate_streaming_test_wav
```

This will save a .wav file to **/tmp/speech_commands_train/streaming_test.wav**, and a text file listing the labels to **/tmp/speech_commands_train/streaming_test_labels.txt**. You can then run accuracy testing with:

```
bazel run tensorflow/examples/speech_commands:test_streaming_accuracy -- \
--graph=/tmp/my_frozen_graph.pb \
--labels=/tmp/speech_commands_train/conv_labels.txt \
--wav=/tmp/speech_commands_train/streaming_test.wav \
--ground_truth=/tmp/speech_commands_train/streaming_test_labels.txt \
--verbose
```

This will output information about the number of words correctly matched, how many were

given the wrong labels, and how many times the model triggered when there was no real word spoken. There are various parameters that control how the signal averaging works, including `--average_window_ms` which sets the length of time to average results over, `--clip_stride_ms` which is the time between applications of the model, `--suppression_ms` which stops subsequent word detections from triggering for a certain time after an initial one is found, and `--detection_threshold`, which controls how high the average score must be before it's considered a solid result.

You'll see that the streaming accuracy outputs three numbers, rather than just the one metric used in training. This is because different applications have varying requirements, with some being able to tolerate frequent incorrect results as long as real words are found (high recall), while others very focused on ensuring the predicted labels are highly likely to be correct even if some aren't detected (high precision). The numbers from the tool give you an idea of how your model will perform in an application, and you can try tweaking the signal averaging parameters to tune it to give the kind of performance you want. To understand what the right parameters are for your application, you can look at generating an ROC curve (https://en.wikipedia.org/wiki/Receiver_operating_characteristic) to help you understand the tradeoffs.

RecognizeCommands

The streaming accuracy tool uses a simple decoder contained in a small C++ class called RecognizeCommands (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/recognize_commands.h). This class is fed the output of running the TensorFlow model over time, it averages the signals, and returns information about a label when it has enough evidence to think that a recognized word has been found. The implementation is fairly small, just keeping track of the last few predictions and averaging them, so it's easy to port to other platforms and languages as needed. For example, it's convenient to do something similar at the Java level on Android, or Python on the Raspberry Pi. As long as these implementations share the same logic, you can tune the parameters that control the averaging using the streaming test tool, and then transfer them over to your application to get similar results.

Advanced Training

The defaults for the training script are designed to produce good end to end results in a

comparatively small file, but there are a lot of options you can change to customize the results for your own requirements.

Custom Training Data

By default the script will download the Speech Commands dataset (https://download.tensorflow.org/data/speech_commands_v0.01.tgz), but you can also supply your own training data. To train on your own data, you should make sure that you have at least several hundred recordings of each sound you would like to recognize, and arrange them into folders by class. For example, if you were trying to recognize dog barks from cat miaows, you would create a root folder called **`animal_sounds`**, and then within that two sub-folders called **`bark`** and **`miaow`**. You would then organize your audio files into the appropriate folders.

To point the script to your new audio files, you'll need to set **`--data_url=`** to disable downloading of the Speech Commands dataset, and **`--data_dir=/your/data/folder/`** to find the files you've just created.

The files themselves should be 16-bit little-endian PCM-encoded WAVE format. The sample rate defaults to 16,000, but as long as all your audio is consistently the same rate (the script doesn't support resampling) you can change this with the **`--sample_rate`** argument. The clips should also all be roughly the same duration. The default expected duration is one second, but you can set this with the **`--clip_duration_ms`** flag. If you have clips with variable amounts of silence at the start, you can look at word alignment tools to standardize them (here's a quick and dirty approach you can use too (https://petewarden.com/2017/07/17/a-quick-hack-to-align-single-word-audio-recordings/)).

One issue to watch out for is that you may have very similar repetitions of the same sounds in your dataset, and these can give misleading metrics if they're spread across your training, validation, and test sets. For example, the Speech Commands set has people repeating the same word multiple times. Each one of those repetitions is likely to be pretty close to the others, so if training was overfitting and memorizing one, it could perform unrealistically well when it saw a very similar copy in the test set. To avoid this danger, Speech Commands trys to ensure that all clips featuring the same word spoken by a single person are put into the same partition. Clips are assigned to training, test, or validation sets based on a hash of their filename, to ensure that the assignments remain steady even as new clips are added and avoid any training samples migrating into the other sets. To make sure that all a given speaker's words are in the same bucket, the hashing function (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/input_data.py)

ignores anything in a filename after '*nohash*' when calculating the assignments. This means that if you have file names like `pete_nohash_0.wav` and `pete_nohash_1.wav`, they're guaranteed to be in the same set.

Unknown Class

It's likely that your application will hear sounds that aren't in your training set, and you'll want the model to indicate that it doesn't recognize the noise in those cases. To help the network learn what sounds to ignore, you need to provide some clips of audio that are neither of your classes. To do this, you'd create `quack`, `oink`, and `moo` subfolders and populate them with noises from other animals your users might encounter. The `--wanted_words` argument to the script defines which classes you care about, all the others mentioned in subfolder names will be used to populate an `_unknown_` class during training. The Speech Commands dataset has twenty words in its unknown classes, including the digits zero through nine and random names like "Sheila".

By default 10% of the training examples are picked from the unknown classes, but you can control this with the `--unknown_percentage` flag. Increasing this will make the model less likely to mistake unknown words for wanted ones, but making it too large can backfire as the model might decide it's safest to categorize all words as unknown!

Background Noise

Real applications have to recognize audio even when there are other irrelevant sounds happening in the environment. To build a model that's robust to this kind of interference, we need to train against recorded audio with similar properties. The files in the Speech Commands dataset were captured on a variety of devices by users in many different environments, not in a studio, so that helps add some realism to the training. To add even more, you can mix in random segments of environmental audio to the training inputs. In the Speech Commands set there's a special folder called `_background_noise_` which contains minute-long WAVE files with white noise and recordings of machinery and everyday household activity.

Small snippets of these files are chosen at random and mixed at a low volume into clips during training. The loudness is also chosen randomly, and controlled by the `--background_volume` argument as a proportion where 0 is silence, and 1 is full volume. Not all clips have background added, so the `--background_frequency` flag controls what proportion have them mixed in.

Your own application might operate in its own environment with different background noise

patterns than these defaults, so you can supply your own audio clips in the **`_background_noise_`** folder. These should be the same sample rate as your main dataset, but much longer in duration so that a good set of random segments can be selected from them.

Silence

In most cases the sounds you care about will be intermittent and so it's important to know when there's no matching audio. To support this, there's a special **`_silence_`** label that indicates when the model detects nothing interesting. Because there's never complete silence in real environments, we actually have to supply examples with quiet and irrelevant audio. For this, we reuse the **`_background_noise_`** folder that's also mixed in to real clips, pulling short sections of the audio data and feeding those in with the ground truth class of **`_silence_`**. By default 10% of the training data is supplied like this, but the **`--silence_percentage`** can be used to control the proportion. As with unknown words, setting this higher can weight the model results in favor of true positives for silence, at the expense of false negatives for words, but too large a proportion can cause it to fall into the trap of always guessing silence.

Time Shifting

Adding in background noise is one way of distorting the training data in a realistic way to effectively increase the size of the dataset, and so increase overall accuracy, and time shifting is another. This involves a random offset in time of the training sample data, so that a small part of the start or end is cut off and the opposite section is padded with zeroes. This mimics the natural variations in starting time in the training data, and is controlled with the **`--time_shift_ms`** flag, which defaults to 100ms. Increasing this value will provide more variation, but at the risk of cutting off important parts of the audio. A related way of augmenting the data with realistic distortions is by using time stretching and pitch scaling (https://en.wikipedia.org/wiki/Audio_time_stretching_and_pitch_scaling), but that's outside the scope of this tutorial.

Customizing the Model

The default model used for this script is pretty large, taking over 800 million FLOPs for each inference and using 940,000 weight parameters. This runs at usable speeds on desktop machines or modern phones, but it involves too many calculations to run at interactive speeds on devices with more limited resources. To support these use cases, there's a couple of

alternatives available:

low_latency_conv Based on the 'cnn-one-fstride4' topology described in the Convolutional Neural Networks for Small-footprint Keyword Spotting paper (http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf). The accuracy is slightly lower than 'conv' but the number of weight parameters is about the same, and it only needs 11 million FLOPs to run one prediction, making it much faster.

To use this model, you specify **`--model_architecture=low_latency_conv`** on the command line. You'll also need to update the training rates and the number of steps, so the full command will look like:

```
python tensorflow/examples/speech_commands/train \
--model_architecture=low_latency_conv \
--how_many_training_steps=20000,6000 \
--learning_rate=0.01,0.001
```

This asks the script to train with a learning rate of 0.01 for 20,000 steps, and then do a fine-tuning pass of 6,000 steps with a 10x smaller rate.

low_latency_svdf Based on the topology presented in the Compressing Deep Neural Networks using a Rank-Constrained Topology paper (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43813.pdf). The accuracy is also lower than 'conv' but it only uses about 750 thousand parameters, and most significantly, it allows for an optimized execution at test time (i.e. when you will actually use it in your application), resulting in 750 thousand FLOPs.

To use this model, you specify **`--model_architecture=low_latency_svdf`** on the command line, and update the training rates and the number of steps, so the full command will look like:

```
python tensorflow/examples/speech_commands/train \
--model_architecture=low_latency_svdf \
--how_many_training_steps=100000,35000 \
--learning_rate=0.01,0.005
```

Note that despite requiring a larger number of steps than the previous two topologies, the reduced number of computations means that training should take about the same time, and at the end reach an accuracy of around 85%. You can also further tune the topology fairly easily for computation and accuracy by changing these parameters in the SVDF layer:

- rank - The rank of the approximation (higher typically better, but results in more

computation).

- num_units - Similar to other layer types, specifies the number of nodes in the layer (more nodes better quality, and more computation).

Regarding runtime, since the layer allows optimizations by caching some of the internal neural network activations, you need to make sure to use a consistent stride (e.g. 'clip_stride_ms' flag) both when you freeze the graph, and when executing the model in streaming mode (e.g. test_streaming_accuracy.cc).

Other parameters to customize If you want to experiment with customizing models, a good place to start is by tweaking the spectrogram creation parameters. This has the effect of altering the size of the input image to the model, and the creation code in models.py (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/models.py)
will adjust the number of computations and weights automatically to fit with different dimensions. If you make the input smaller, the model will need fewer computations to process it, so it can be a great way to trade off some accuracy for improved latency. The `--window_stride_ms` controls how far apart each frequency analysis sample is from the previous. If you increase this value, then fewer samples will be taken for a given duration, and the time axis of the input will shrink. The `--dct_coefficient_count` flag controls how many buckets are used for the frequency counting, so reducing this will shrink the input in the other dimension. The `--window_size_ms` argument doesn't affect the size, but does control how wide the area used to calculate the frequencies is for each sample. Reducing the duration of the training samples, controlled by `--clip_duration_ms`, can also help if the sounds you're looking for are short, since that also reduces the time dimension of the input. You'll need to make sure that all your training data contains the right audio in the initial portion of the clip though.

If you have an entirely different model in mind for your problem, you may find that you can plug it into models.py (https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/models.py)
and have the rest of the script handle all of the preprocessing and training mechanics. You would add a new clause to `create_model`, looking for the name of your architecture and then calling a model creation function. This function is given the size of the spectrogram input, along with other model information, and is expected to create TensorFlow ops to read that in and produce an output prediction vector, and a placeholder to control the dropout rate. The rest of the script will handle integrating this model into a larger graph doing the input calculations and applying softmax and a loss function to train it.

One common problem when you're adjusting models and training hyper-parameters is that not-a-number values can creep in, thanks to numerical precision issues. In general you can solve these by reducing the magnitude of things like learning rates and weight initialization functions, but if they're persistent you can enable the **--check_nans** flag to track down the source of the errors. This will insert check ops between most regular operations in TensorFlow, and abort the training process with a useful error message when they're encountered.

上次更新日期：十月 27, 2017