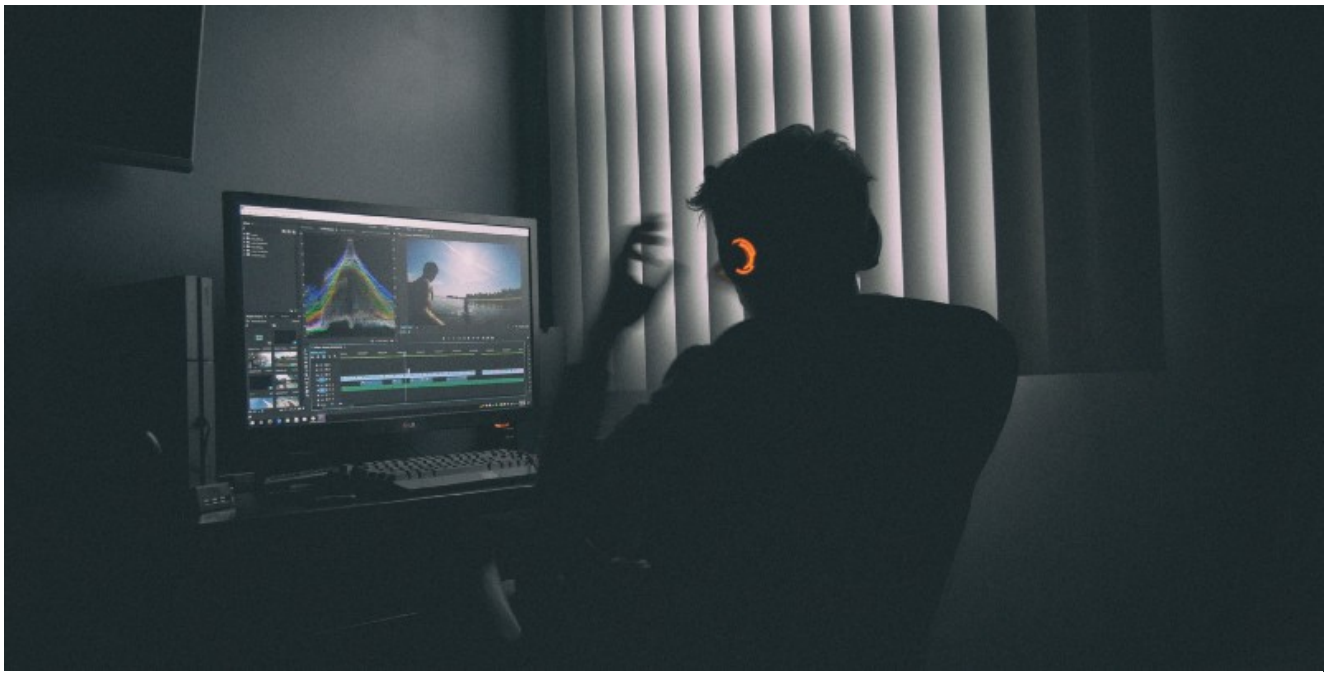
7 MARCH 2018 / #MACHINE LEARNING

How to use sound classification with TensorFlow on an IoT platform



by Nikolay Khabarov

Introduction

There are many different projects and services for human speech recognition, such as Pocketsphinx, Google's Speech API, and many

transform it to text with pretty good accuracy. But none of them can determine different sounds captured by the microphone. What was on record: human speech, animal sounds, or music playing?

We were faced with this task, and decided to investigate and build a few sample projects which would be able to classify different sounds using machine learning algorithms.

This article describes which tools we chose, what challenges we faced, how we trained our model for TensorFlow, and how to run our open source project.

We can also supply the recognition results to DeviceHive (the IoT platform) to use them in cloud services for 3rd party application.

Choosing Tools and a Classification Model

First, we needed to choose which software would work best with neural networks. The first suitable solution that we found was Python Audio Analysis.

The main problem in machine learning is finding a good training dataset. There are many datasets for speech recognition and music classification, but not a lot for random sound classification. After some research, we found the urban sound dataset.

After some testing, we were faced with the following problems:

- pyAudioAnalysis isn't flexible enough. It doesn't take a wide variety of parameters, and some of them calculate on the fly (for example, the number of training experiments based on number of samples — and you can't alter this).

- The dataset only has 10 classes, and all of them are "urban."

The next solution that we found was Google AudioSet. It is based on labeled YouTube video segments and can be downloaded in two formats:

1. CSV files describing, for each segment, the YouTube video ID, start time, end time, and one or more labels.
2. Extracted audio features that are stored as TensorFlow Record files.

These features are compatible with YouTube-8M models. This solution also uses the TensorFlow VGGish model as the feature extractor. It covered many of our requirements, and was therefore the best choice for us.

Training the Model

The next task was to figure out how the YouTube-8M interface worked. It's designed to work with videos, but fortunately can work with audio as well. This library is pretty flexible, but it has a hardcoded number of sample classes. So we modified this a little bit to pass the number of classes as a parameter.

YouTube-8M can work with data of two types: aggregated features and frame features. Google AudioSet can provide data as features as we noted before. Through a little more research, we discovered that the features are in frame format. We then needed to choose the model to be trained.

Resources, Time, and Accuracy

GPUs are a more suitable choice for machine learning than CPUs. You can find more info about this here. So we will skip this point and go directly to our setup. For our experiments, we used a PC with one NVIDIA GTX 970 4GB.

In our case, the training time didn't really matter. We should mention that one-two hours of training was enough to make an initial decision about the chosen model and its accuracy.

Of course we wanted to get the best accuracy possible. But to train a more complex model (potentially better accuracy), you would need more RAM (video RAM in case of GPU) to fit it in.

Choosing the Model

A full list of YouTube-8M models with descriptions is available here. Because our training data was in frame format, frame-level models had to be used. Google AudioSet provided us with a data set split into three parts: balanced train, unbalanced train, and evaluation. You can get more info about them here.

A modified version of YouTube-8M was used for training and evaluation. It's available here.

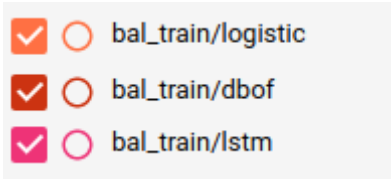
Balanced Train

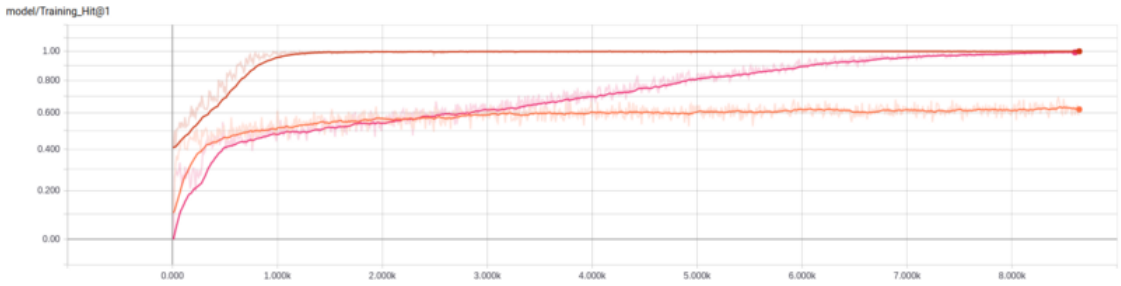
The training command looks like this:

python train.py –train_data_pattern=/path_to_data/audioset_v1_embeddings/bal_train/.tfrecord –num_epochs=100 –learning_rate_decay_examples=400000 –feature_names=audio_embedding –feature_sizes=128 –frame_features –batch_size=512 –num_classes=527 –train_dir=/path_to_logs –model=ModelName*

For LstmModel we changed the base learning rate to 0.001 as the documentation suggested. Also we changed the default value of lstm_cells to 256, because we didn't have enough RAM for more.

Let's see the training results:



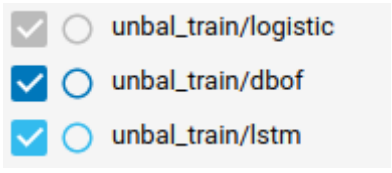


Model nameTraining timeTraining last step hitEvaluation average hitLogistic14m 3s0.58590.5560Dbof31m 46s1.0000.5220Lstm1h 45m 53s0.98830.4581

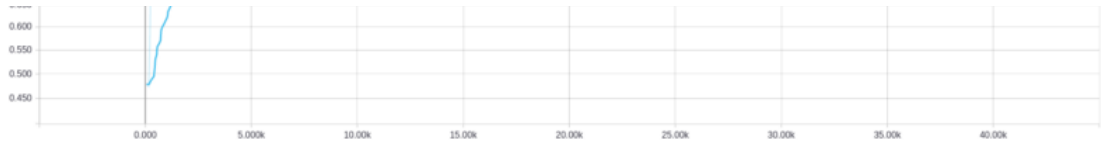
As you can see, we got good results during the training step — but this doesn't mean we would get good results on the full evaluation.

Unbalanced Train

Then we tried the unbalanced train dataset. It has a lot more samples, so we changed the number of training epochs to 10 (should change to 5 at least, because it took significant time to train).



Donate



Model nameTraining timeTraining last step hitEvaluation average hitLogistic2h 4m 14s0.87500.5125Dbof4h 39m 29s0.88480.5605Lstm9h 42m 52s0.86910.5396

Train Logs

If you want to examine our training logs, you can download and extract train_logs.tar.gz. Then run *tensorboard –logdir /path_to_train_logs/* and go to http://127.0.0.1:6006

More About Training

YouTube-8M takes many parameters, and a lot of them affect the training process.

For example: You can tune the learning rate and number of epochs that will change the training process a lot. There are also three different functions for loss calculation and many other useful variables that you can tune and change to improve the results.

Using the Trained Model with Audio

Capture Devices

Now that we had some trained models, it was time to add some code to interact with them.

Capture Mic

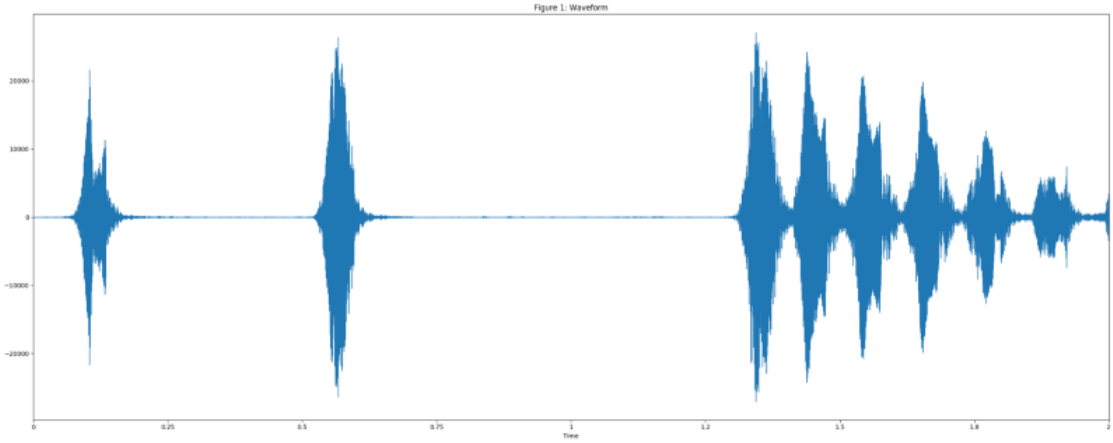
We needed to somehow capture audio data from a microphone. We used PyAudio. It provides a simple interface and can work on most

Sound Preparation

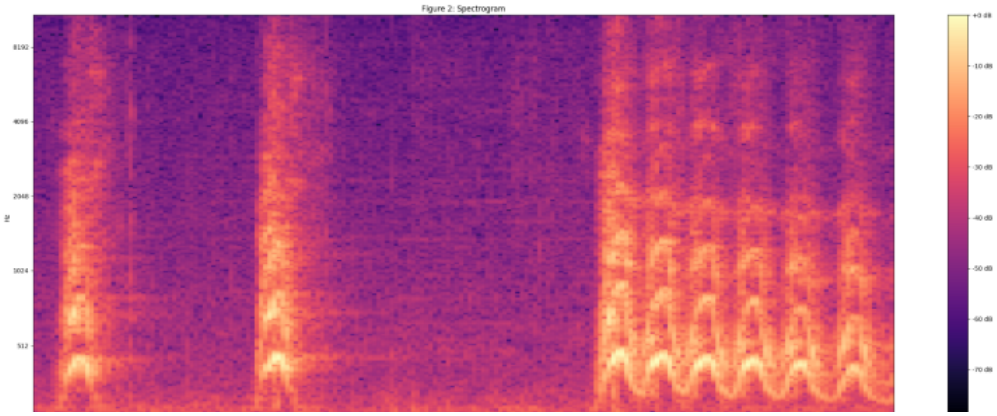
As we mentioned before, we used the TensorFlow VGGish model as the feature extractor. Here is a short explanation of the transformation process:

The “dog bark” example from the UrbanSound dataset was used for visualization.

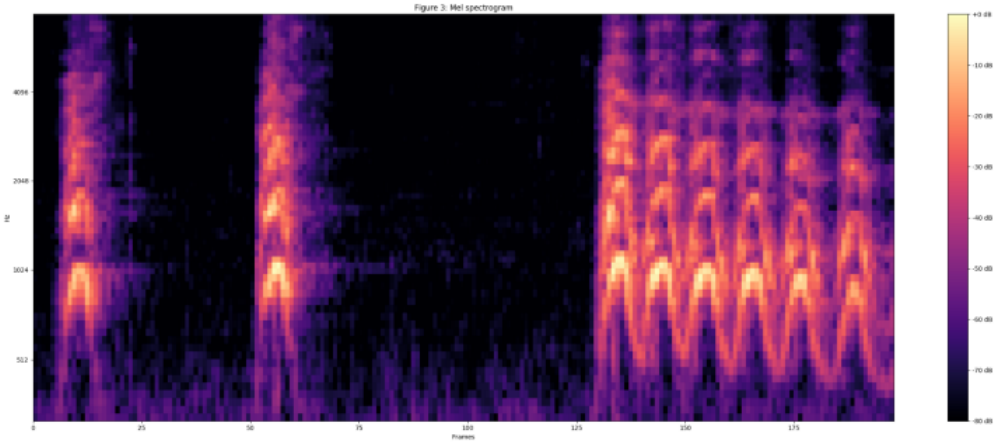
Resample audio to 16 kHz mono.



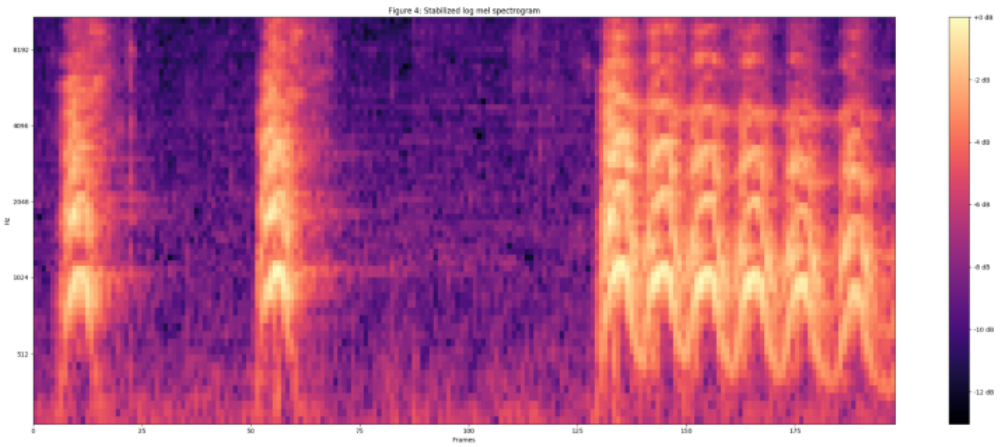
Compute spectrogram using magnitudes of the Short-Time Fourier Transform with a window size of 25 ms, a window hop of 10 ms, and a periodic Hann window.



Compute mel spectrogram by mapping the spectrogram to 64 mel bins.



Compute stabilized log mel spectrogram by applying log(mel-spectrum + 0.01) where an offset is used to avoid taking a logarithm of zero.



These features were then framed into non-overlapping examples of 0.96 seconds, where each example covers 64 mel bands and 96 frames of 10 ms each.

These examples were then fed into the VGGish model to extract embeddings.

Classifying

And finally we needed an interface to feed the data to the neural network and get the results.

We used the YouTube-8M interface as an example, but modified it to remove the serialization/deserialization step.

Here you can see the results of our work. Let's take a closer look.

Installation

PyAudio uses libportaudio2 and portaudio19-dev, so you need to install them to make it work.

Some Python libraries are required. You can install them using pip.

pip install -r requirements.txt

You also need to download and extract the archive to the project root with the saved models. You can find it here.

Running

Our project provides three interfaces to use.

1. Process Prerecorded Audio File

Simply run *python parse_file.py path_to_your_file.wav* and you will see in the terminal something like *Speech: 0.75*, *Music: 0.12*, *Inside, large room or hall: 0.03*

The result depends on the input file. These values are the predictions that the neural network has made. A higher value means a higher chance of the input file belonging to that class.

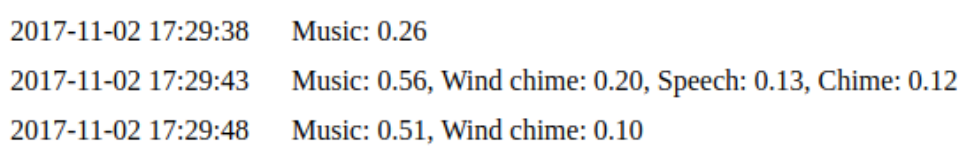
2. Capture and Process Data from Mic

python capture.py starts the process that will capture data from your mic infinitely. It will feed data to the classification interface every 5–7 seconds (by default). You can see the results in the previous example.

You can run it with *–save_path=/path_to_samples_dir/* and in this case all captured data will be stored in the provided directory in *wav* files. This function is useful in case you want to try different models with the same example(s). Use the *–help* parameter to get more info.

3. Web Interface

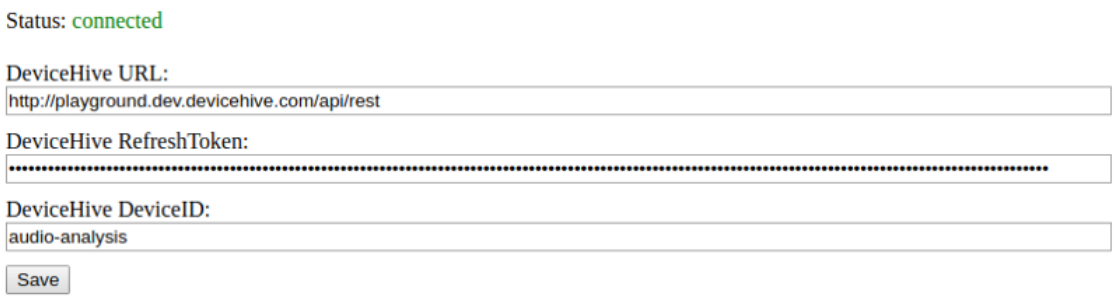
python daemon.py implements a simple web interface that is available on http://127.0.0.1:8000 by default. We used the same code as for the previous example. You can see the last ten predictions on the events (http://127.0.0.1:8000/events) page.



2017-11-02 17:29:58	Music: 0.56, Speech: 0.13
2017-11-02 17:30:03	Music: 0.56, Speech: 0.15, Snort: 0.13, Crunch: 0.10
2017-11-02 17:30:08	Speech: 0.28, Music: 0.23, Fly, housefly: 0.23, Bee, wasp, etc.: 0.14
2017-11-02 17:30:13	Speech: 0.33, Music: 0.24, Fly, housefly: 0.18, Bee, wasp, etc.: 0.11
2017-11-02 17:30:18	Fly, housefly: 0.28, Speech: 0.26, Music: 0.19, Bee, wasp, etc.: 0.17
2017-11-02 17:30:23	Speech: 0.21, Music: 0.15

IoT Service Integration

Last but not least is integration with the IoT infrastructure. If you run the web interface that we mentioned in the previous section, then you can find the DeviceHive client status and configuration on the index page. As long as the client is connected, predictions will be sent to the specified device as notifications.



Conclusion

TensorFlow is a very flexible tool, as you can see, and can be helpful in many machine learning applications like image and sound recognition. Having such a solution together with an IoT platform allows you to build a smart solution over a very wide area.

Smart cities could use this for security purposes, continuously listening for broken glass, gunfire, and other sounds related to crimes. Even in rainforests, such a solution could be used to track wild animals or birds by analyzing their voices.

The IoT platform can deliver all such notifications. This solution can be installed on local devices (though it still can be deployed somewhere as a cloud service) to minimize traffic and cloud expenses. It can also be customized to deliver only notifications instead of including the raw audio. Do not forget that this is an open source project, so please feel free to use it.

Written by Nikolay Khabarov, co-founder of DeviceHive.
