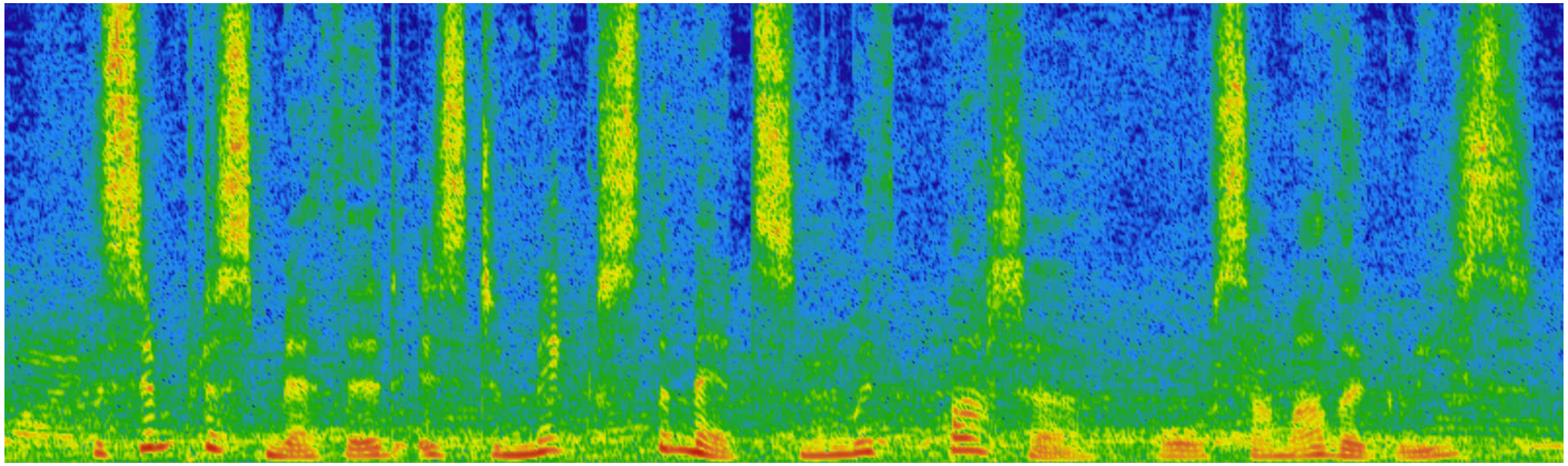
xiph.org / moz://a

RNNoise: Learning Noise Suppression

(Monty's main demo page)



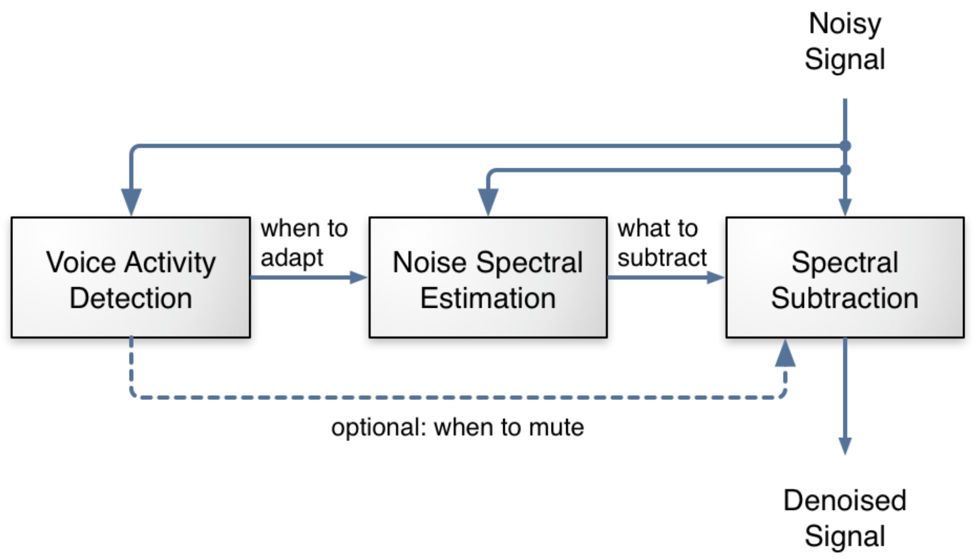
The image above shows the spectrogram of the audio before and after (when moving the mouse over) noise suppression.

Here's RNNoise

This demo presents the RNNoise project, showing how deep learning can be applied to noise suppression. The main idea is to combine *classic* signal processing with deep learning to create a real-time noise suppression algorithm that's small and **fast**. No expensive GPUs required — it runs easily on a Raspberry Pi. The result is much simpler (easier to tune) and sounds better than traditional noise suppression systems (been there!).

Noise Suppression

Noise suppression is a pretty old topic in speech processing, dating back to at least the 70s. As the name implies, the idea is to take a noisy signal and remove as much noise as possible while causing minimum distortion to the speech of interest.



This is a conceptual view of a conventional noise suppression algorithm. A voice activity detection (VAD) module detects when the signal contains voice and when it's just noise. This is used by a noise spectral estimation module to figure out the spectral characteristics of the noise (how much power at each frequency). Then, knowing how the noise looks like, it can be "subtracted" (not as simple as it sounds) from the input audio.

From looking at the figure above, noise suppression looks simple enough: just three conceptually simple tasks and we're done, right? Right — and wrong! Any undergrad EE student can write a noise suppression algorithm that works... kinda... sometimes. The hard part is to make it work well, all the time, for all kinds of noise. That requires very careful tuning of every knob in the algorithm, many special cases for strange signals and lots of testing. There's always some weird signal that will cause problems and require more tuning and it's very easy to break more things than you fix. It's 50% science, 50% art. I've been there before with the noise suppressor in the speexdsp library. It kinda works, but it's not great.

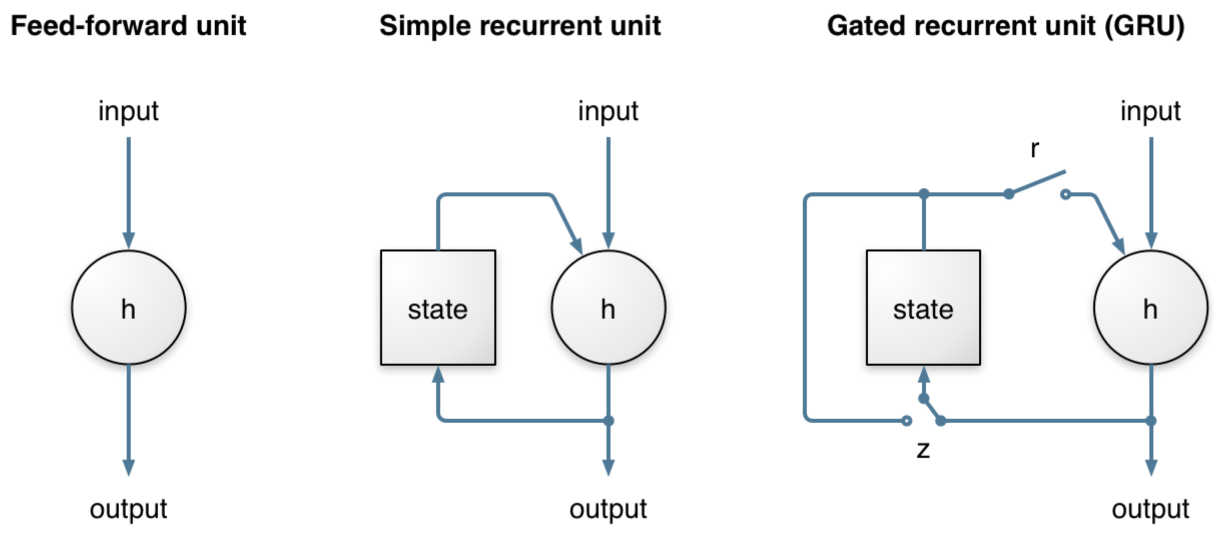
Deep Learning and Recurrent Neural Networks

Deep learning is the new version of an old idea: artificial neural networks. Although those have been around since the 60s, what's new in recent years is that:

1. We now know how to make them deeper than two hidden layers
2. We know how to make recurrent networks remember patterns long in the past
3. We have the computational resources to actually train them

Recurrent neural networks (RNN) are very important here because they make it possible to model time sequences instead of just considering input and output frames independently. This is especially important for noise suppression because we need time to get a good estimate of the noise. For a long time, RNNs were heavily limited in their ability because they could not hold information for a long period of time and because the gradient descent process involved when back-propagating through time was very inefficient (the vanishing gradient problem). Both problems were solved by the invention of *gated units*, such as the Long Short-Term Memory (LSTM), the Gated Recurrent Unit (GRU), and their many variants.

RNNoise uses the Gated Recurrent Unit (GRU) because it performs slightly better than LSTM on this task and requires fewer resources (both CPU and memory for weights). Compared to *simple* recurrent units, GRUs have two extra *gates*. The *reset* gate controls whether the state (memory) is used in computing the new state, whereas the *update* gate controls how much the state will change based on the new input. This update gate (when off) makes it possible (and easy) for the GRU to remember information for a long period of time and is the reason GRUs (and LSTMs) perform much better than simple recurrent units.



Comparing a simple recurrent unit with a GRU. The difference lies in the GRU's r and z gates, which make it possible to learn longer-term patterns. Both are soft switches (value between 0 and 1) computed based on the previous state of the whole layer and the inputs, with a sigmoid activation function. When *the update gate z is on the left, then the state can remain constant over a long period of time — until a condition causes z to switch to the right.*

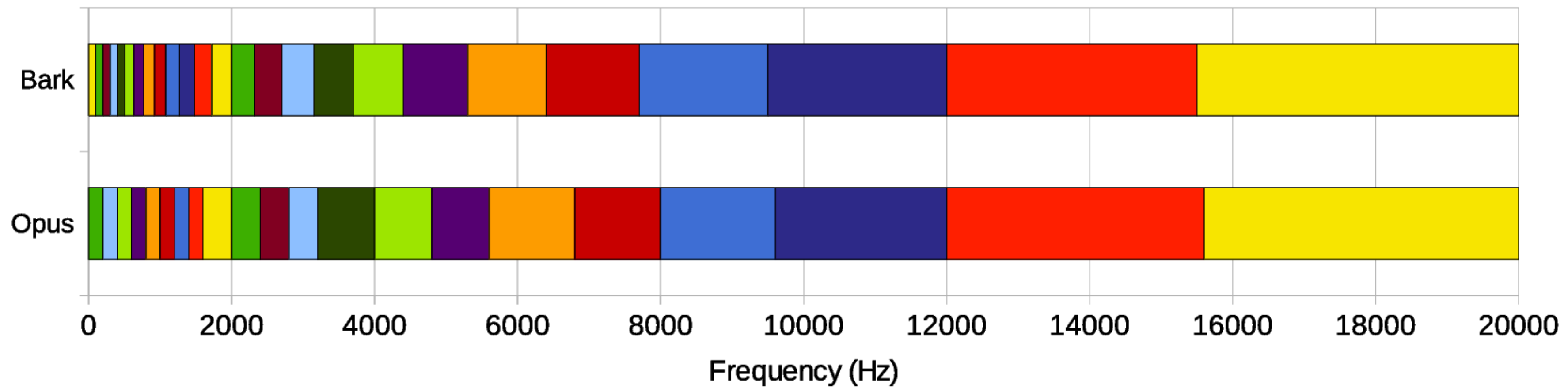
A Hybrid Approach

Thanks to the successes of deep learning, it is now popular to throw deep neural networks at an entire problem. These approaches are called *end-to-end* — it's neurons all the way down. End-to-end approaches have been applied to speech recognition and to speech synthesis On the one hand, these end-to-end systems have proven just how powerful deep neural networks can be. On the other hand, these systems can sometimes be both suboptimal, and wasteful in terms of resources. For example, some approaches to noise suppression use layers with thousands of neurons — and tens of millions of weights — to perform noise suppression. The drawback is not only the computational cost of running the network, but also the *size* of the model itself because your library is now a thousand lines of code along with tens of megabytes (if not more) worth of neuron weights.

That's why we went with a different approach here: keep all the basic signal processing that's needed anyway (not have a neural network attempt to emulate it), but let the neural network learn all the tricky parts that require endless tweaking next to the signal processing. Another thing that's different from *some* existing work on noise suppression with deep learning is that we're targeting real-time communication rather than speech recognition, so we can't afford to *look ahead* more than a few milliseconds (in this case 10 ms).

Defining the problem

To avoid having a very large number of outputs — and thus a large number of neurons — we decided against working directly with samples or with a spectrum. Instead, we consider frequency *bands* that follow the Bark scale, a frequency scale that matches how we perceive sounds. We use a total of 22 bands, instead of the 480 (complex) spectral values we would otherwise have to consider.



Layout of the Opus bands vs the actual Bark scale. For RNNoise, we use the same base layout as Opus. Since we overlap the bands, the boundaries between the Opus bands become the center of the overlapped RNNoise bands. The bands are wider at higher frequency because the ear has poorer frequency resolution there. At low frequencies, the bands are narrower, but not as narrow as the Bark scale would give because then we would not have enough data to make good estimates.

Of course, we cannot reconstruct audio from just the energy in 22 bands. What we can do though, is compute a gain to apply to the signal for each of these bands. You can think about it as using a 22-band equalizer and rapidly changing the level of each band so as to attenuate the noise, but let the signal through.

There are several advantages to operating with per-band gains. First, it makes for a much simpler model since there are fewer bands to compute. Second, it makes it impossible to create so-called *musical noise* artifacts, where only a single tone gets though while its neighbours are attenuated. These artifacts are common in noise suppression and quite annoying. With bands that are wide enough, we either let a whole band through, or we cut it all. The third advantage comes from how we optimize the model. Since the gains are always bounded between 0 and 1, simply using a sigmoid activation function (whose output is also between 0 and 1) to compute them ensures that we can never do something **really** stupid, like adding noise that wasn't there in the first place.

▸ Show nerdy details

The main drawback of the lower resolution we get from using bands is that we do not have a fine enough resolution to suppress the noise between pitch harmonics. Fortunately, it's not so important and there is even an easy trick to do it (see the pitch filtering part below).

Since the output we're computing is based on 22 bands, it makes little sense to have more frequency resolution on the input, so we use the same 22 bands to feed spectral information to the neural network. Because audio has a **huge** dynamic range, it's much better to compute the log of the energy rather than to feed the energy directly. And while we're at it, it never hurts to decorrelate the features using a DCT. The resulting data is a *cepstrum* based on the Bark scale, which is closely related to the Mel-Frequency Cepstral Coefficients (MFCC) that are very commonly used in speech recognition.

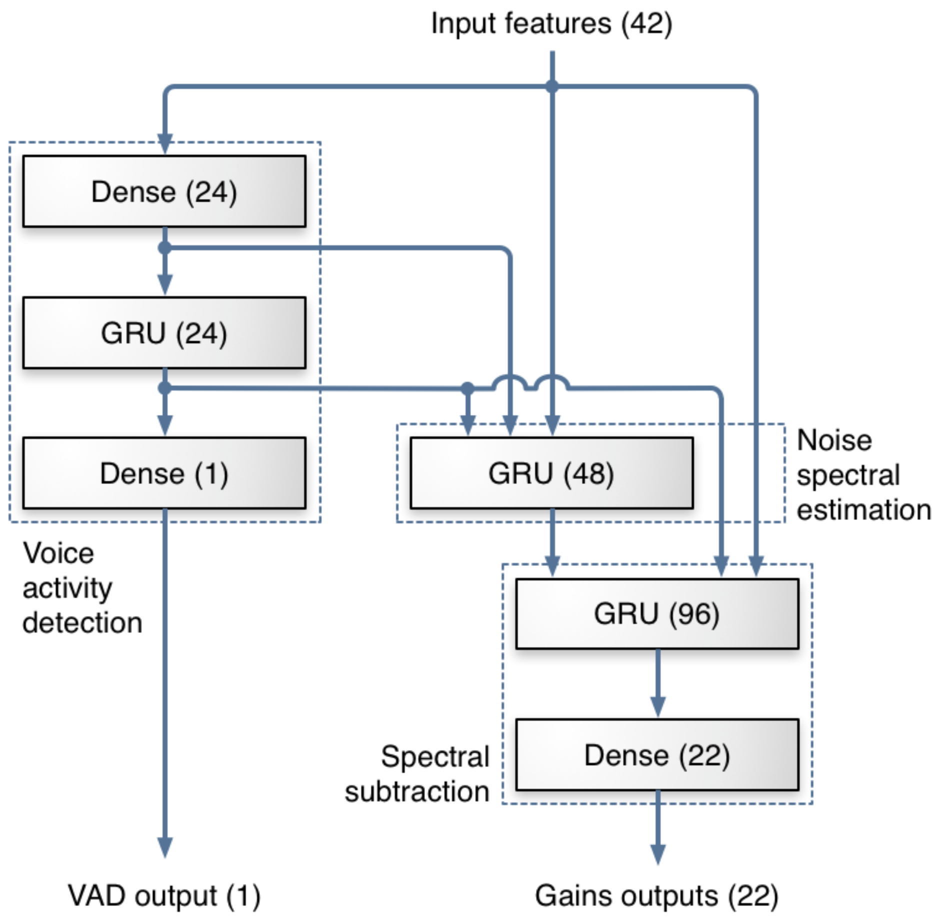
In addition to our cepstral coefficients, we also include:

- The first and second derivatives of the first 6 coefficients across frames
- The pitch period (1/frequency of the fundamental)
- The pitch gain (voicing strength) in 6 bands
- A special non-stationarity value that's useful for detecting speech (but beyond the scope of this demo)

That makes a total of 42 input features to the neural network.

Deep architecture

The deep architecture we use is inspired from the traditional approach to noise suppression. Most of the work is done by 3 GRU layers. The figure below shows the layers we use to compute the band gains and how the architecture maps to the traditional steps in noise suppression. Of course, as is often the case with neural networks we have no actual proof that the network is using its layers as we intend, but the fact that the topology works better than others we tried makes it reasonable to think it is behaving as we designed it.



Topology of the neural network used in this project. Each box represents a layer of neurons, with the number of units indicated in parentheses. Dense layers are fully-connected, non-recurrent layers. One of the outputs of the network is a set of gains to apply at different frequencies. The other output is a voice activity probability, which is not used for noise suppression but is a useful by-product of the network.

It's all about the data

Even deep neural networks can be pretty dumb sometimes. They're very good at what they know about, but they can make pretty spectacular mistakes on inputs that are too far from what they know about. Even worse, they're really lazy students. If they can use any sort of loophole in the training process to avoid learning something hard, then they will. That is why the quality of the training data is critical.

▸ Show nerdy details

In the case of noise suppression, we can't just collect input/output data that can be used for supervised learning since we can rarely get both the clean speech **and** the noisy speech at the same time. Instead, we have to artificially create that data from separate recordings of clean speech and noise. The tricky part is getting a wide variety of noise data to add to the speech. We also have to make sure to cover all kinds of recording conditions. For example, an early version trained only on full-band audio (0-20 kHz) would fail when the audio was low-pass filtered at 8 kHz.

▸ Show nerdy details

Pitch filtering

Since the frequency resolution of our bands is too coarse to filter noise between pitch harmonics, we do it using basic signal processing. This is another part of the hybrid approach. When one has multiple measurements of the same variable, the easiest way to improve the accuracy (reduce noise) is simply to compute the average. Obviously, just computing the average of adjacent audio samples isn't what we want since it would result in low-pass filtering. However, when the signal is periodic (such as voiced speech), then we can compute the average of samples offset by the pitch period. This results in a *comb filter* that lets pitch harmonics through, while attenuating the frequencies between them — where the noise lies. To avoid distorting the signal, the comb filter is applied independently for each band and its filter strength depends both on the pitch correlation and on the band gain computed by the neural network.

▸ Show nerdy details

From Python to C

All the design and training of the neural network is done in Python using the awesome Keras deep learning library. Since Python is usually not the language of choice for real-time systems, we have to implement the run-time code in C. Fortunately, running a neural network is by far easier than training one, so all we had to do was implement feed-forward and GRU layers. To make it easier to fit the weights in a reasonable footprint, we constrain the magnitude of the weights to +/- 0.5 during training, which makes it easy to store them using 8-bit values. The resulting model fits in just 85 kB (instead of the 340 kB required to store the weights as 32-bit floats).

The C code is available under a BSD license. Although as of writing this demo, the code is not yet optimized, it already runs about 60x faster than real-time on an x86 CPU. It even runs about 7x faster than real-time on a Raspberry Pi 3. With good vectorization (SSE/AVX), it should be possible to make it about 4x faster than it currently is.

Show Me the Samples!

OK, that's nice and all, but how does it actually **sound**? Here's some examples of RNNoise in action, removing three different types of noise. Neither the noise nor the clean speech were used during the training.

0:00 / 0:37

Suppression algorithm **No suppression** **RNNoise** **Speexdsp**

Noise level (SNR) **0 dB** **5 dB** **10 dB** **15 dB** **20 dB** **Clean**

Noise type **Babble noise** **Car noise** **Street noise**

Select where to start playing when selecting a new sample

Keep playing | Set current position as restart point | Player will **continue** when changing sample.

Evaluating the effect of RNNoise compared to no suppression and to the Speexdsp noise suppressor. Although the SNRs provided go as low as 0 dB, most applications we are targeting (e.g. WebRTC calls) tend to have SNRs closer to 20 dB than to 0 dB.

So what should you listen for anyway? As strange as it may sound, you should **not** be expecting an increase in intelligibility. Humans are so good at understanding speech in noise that an enhancement algorithm — especially one that isn't allowed to look ahead of the speech it's denoising — can only destroy information. So why are we doing this in the first place? For quality. The enhanced speech is much less annoying to listen to and likely causes less listener fatigue.

Actually, there are still a few cases where it can actually help intelligibility. The first is videoconferencing, when multiple speakers are being mixed together. For that application, noise suppression prevents the noise from all the inactive speakers from being mixed in with the active speaker, improving both quality and intelligibility. A second case is when the speech goes through a low bitrate codec. Those tend to degrade noisy speech more than clean speech, so removing the noise allows the codec to do a better job.

Try it on your voice!

Not happy with the samples above? You can actually record from your microphone and have your audio denoised in (near) real-time. If you click on the button below, RNNoise will perform noise suppression in Javascript from your browser. The algorithm runs in real-time but we've purposely **delayed it by a few seconds** to make it easier to hear the denoised output. **Make sure to wear headphones otherwise you'll**

hear a feedback loop. To start the demo, select either "No suppression" or "RNNoise". You can toggle between the two to see the effect of the suppression. If your input doesn't have enough noise, you can artificially add some by clicking the "white noise" button.

Suppression algorithm Off | No suppression | RNNoise

Noise type None | White noise

Donate Your Noise to Science

If you think this work is useful, there's an easy way to help make it even better! All it takes is a minute of your time. Click on the link below, to let us record one minute of noise from where you are. This noise can be used to improve the training of the neural network. As a side benefit, it means that the network will know what kind of noise you have and might do a better job when you get to use it for videoconferencing (e.g. in WebRTC). We're interested in noise from any environment where you might communicate using voice. That can be your office, your car, on the street, or anywhere you might use your phone or computer.

Donate a minute of noise!

Where from here?

If you'd like to know more about the technical details of RNNoise, see this paper (not yet submitted). The code is still under active development (with no frozen API), but is already usable in applications. It is currently targeted at VoIP/videoconferencing applications, but with a few tweaks, it can probably be applied to many other tasks. An obvious target is automatic speech recognition (ASR), and while we can just denoise the noisy speech and send the output to the ASR, this is sub-optimal because it discards useful information about the inherent uncertainty of the process. It's a lot more useful when the ASR knows not only the most likely clean speech, but also how much it can rely on that estimate. Another possible "retargeting" for RNNoise is making a much smarter *noise gate* for electric musical instruments. All it should take is good training data and a few changes to the code to turn a Raspberry Pi into a really good guitar noise gate. Any takers? There are probably many other potential applications we haven't considered yet.

If you would like to comment on this demo, you can do so on here.

—Jean-Marc Valin (jmvalin@jmvalin.ca) September 27, 2017

Additional Resources

1. The code: RNNoise Git repository (Github mirror)
2. J.-M. Valin, A Hybrid DSP/Deep Learning Approach to Real-Time Full-Band Speech Enhancement, arXiv:1709.08243 [cs.SD], 2017.
3. Jean-Marc's blog post for comments

Acknowledgements

Special thanks to Michael Bebenita, Thomas Daede and Yury Delendik for their help putting together this demo. Thanks to Reuben Morais for his help with Keras.