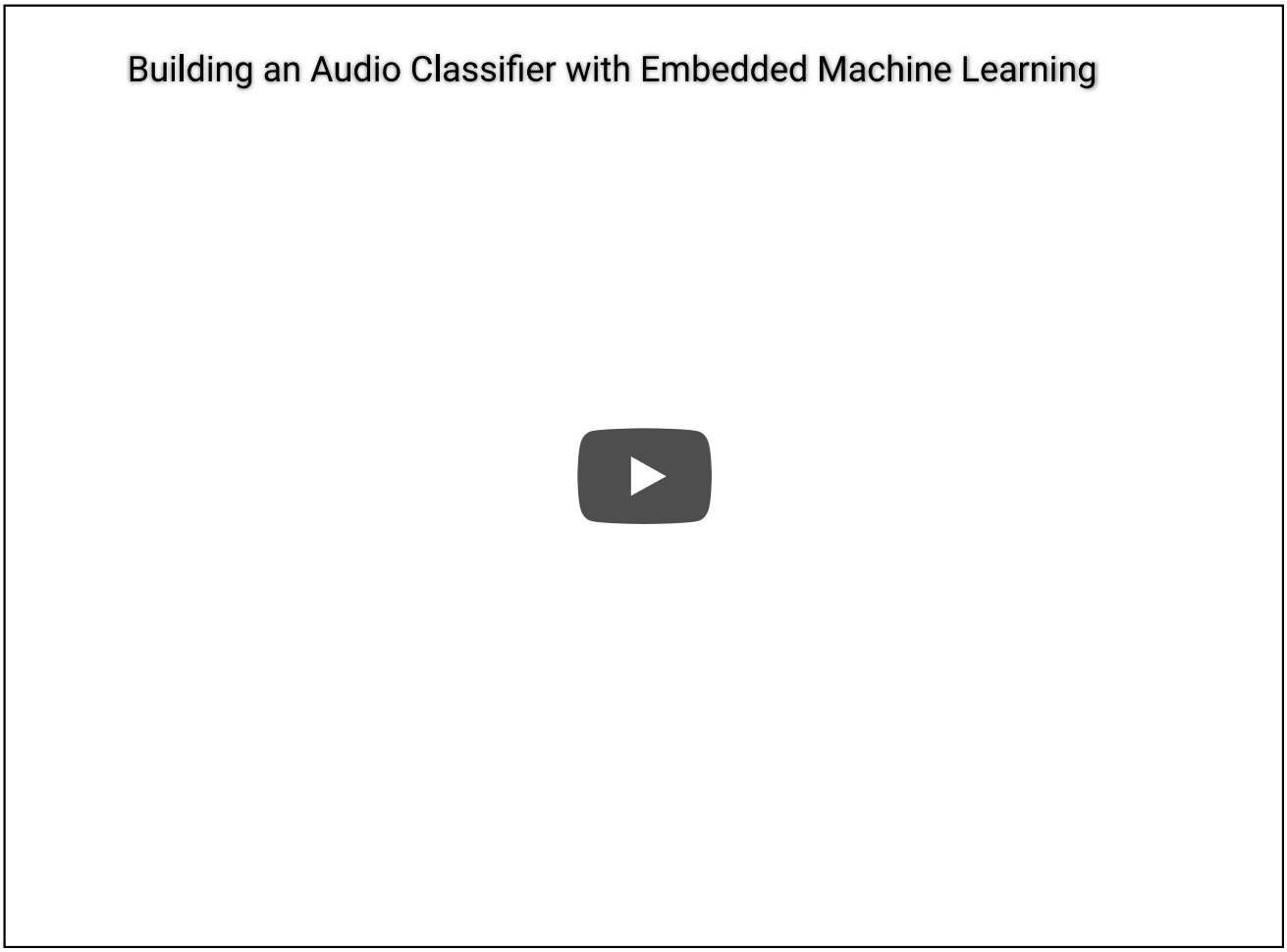
Recognize sounds from audio

In this tutorial, you'll use machine learning to build a system that can recognize when a particular sound is happening—a task known as *audio classification*. The system you create will be able to recognize the sound of water running from a faucet, even in the presence of other background noise.

You'll learn how to collect audio data from microphones, use signal processing to extract the most important information, and train a deep neural network that can tell you whether the sound of running water can be heard in a given clip of audio. Finally, you'll deploy the system to an embedded device and evaluate how well it works.

At the end of this tutorial, you'll have a firm understanding of how to classify audio using Edge Impulse.

There is also a video version of this tutorial:



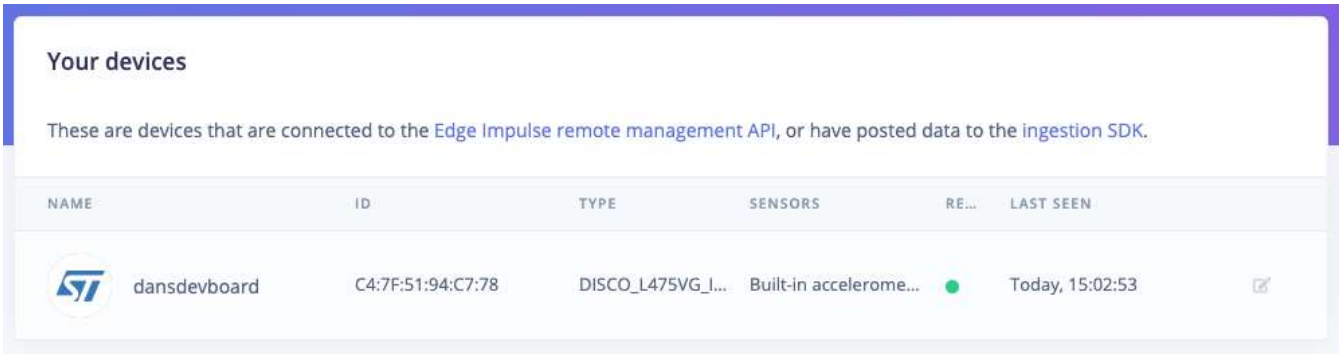
Do you want a device that listens to your voice? We have a specific tutorial for that! See Responding to your voice.

1. Prerequisites

For this tutorial you'll need a supported device. Follow the steps to connect your development board to Edge Impulse.

- ST B-L475E-IOT01A
- Arduino Nano 33 BLE Sense
- Eta Compute ECM3532 AI Sensor
- Eta Compute ECM3532 AI Vision
- Himax WE-I Plus
- Nordic Semiconductor nRF52840 DK
- Nordic Semiconductor nRF5340 DK
- Silicon Labs Thunderboard Sense 2
- Sony's Spresense
- Raspberry Pi 4
- Any mobile phone - the easiest option if you don't have one of the above.

If your device is connected under **Devices** in the studio you can proceed:



Devices tab with the device connected to the remote management interface.

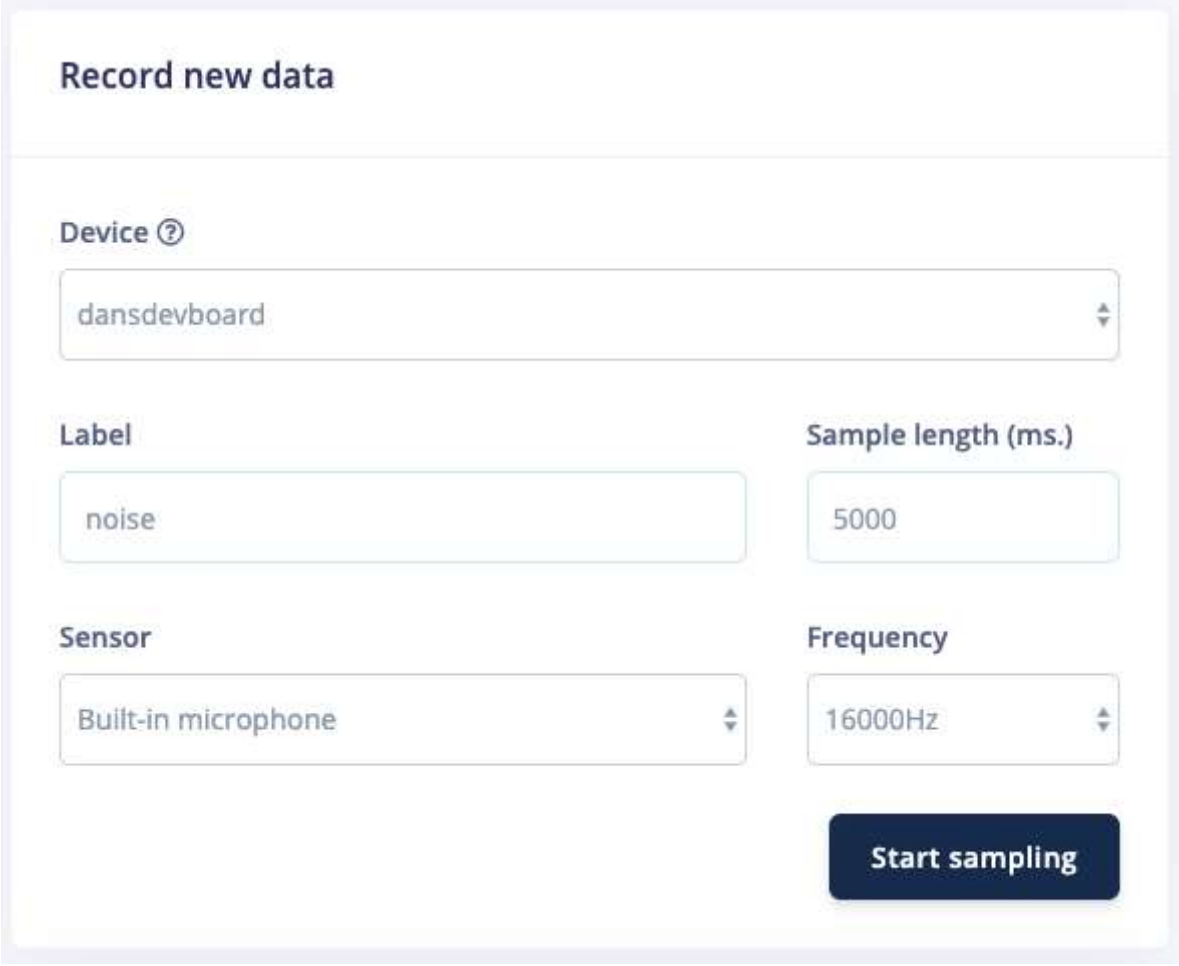
Device compatibility

Edge Impulse can ingest data from any device - including embedded devices that you already have in production. See the documentation for the Ingestion service for more information.

that. You'll also need some examples of typical background noise that *doesn't* contain the sound of a faucet, so the model can learn to discriminate between the two. These two types of examples represent the two *classes* we'll be training our model to detect: background noise, or running faucet.

You can use your device to collect some data. In the studio, go to the **Data acquisition** tab. This is the place where all your raw data is stored, and - if your device is connected to the remote management API - where you can start sampling new data.

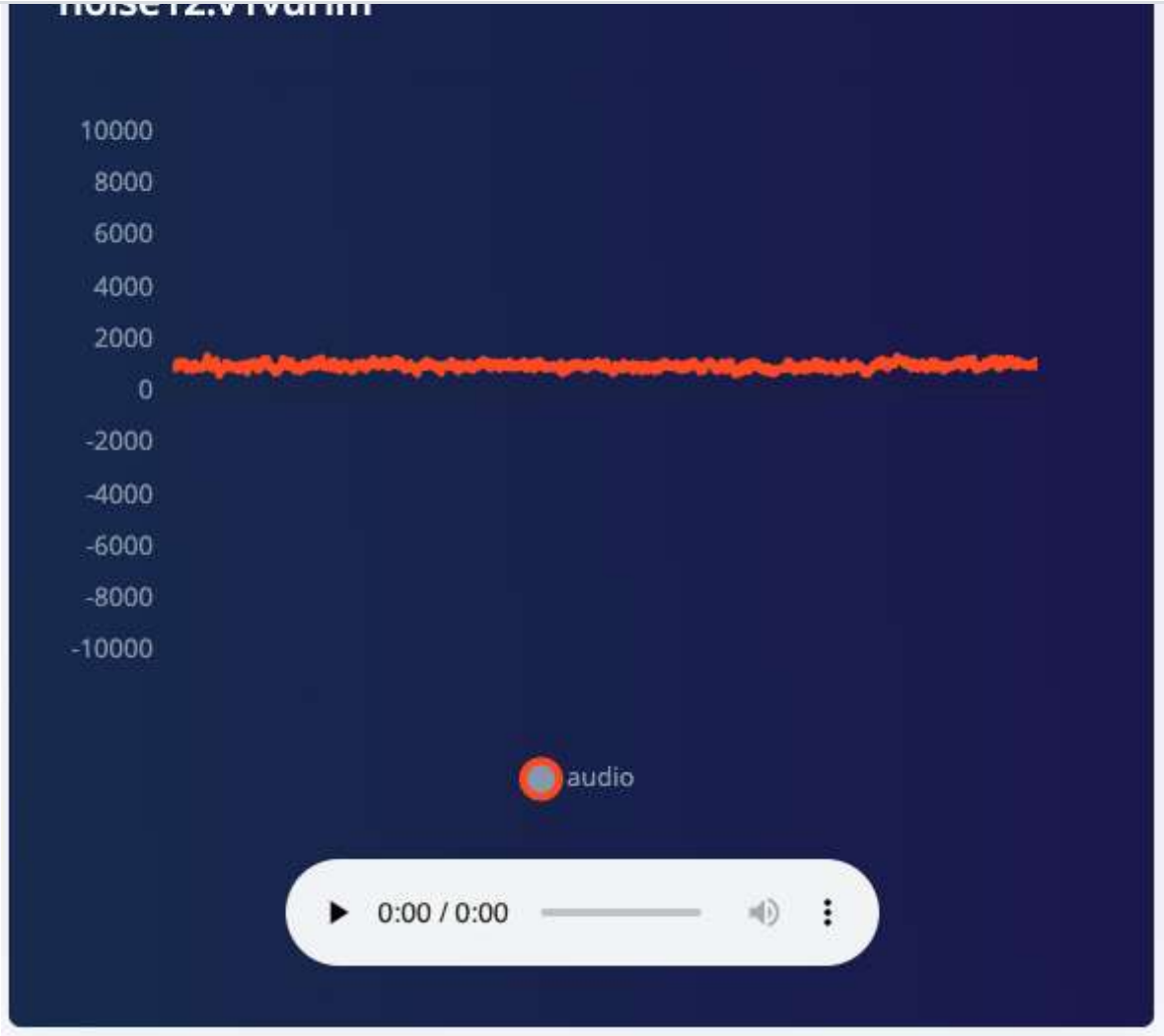
Let's start by recording an example of background noise that doesn't contain the sound of a running faucet. Under **Record new data**, select your device, set the label to `noise`, the sample length to `1000`, and the sensor to `Built-in microphone`. This indicates that you want to record 1 second of audio, and label the recorded data as `noise`. You can later edit these labels if needed.



Record new data screen.

After you click **Start sampling**, the device will capture a second of audio and transmit it to Edge Impulse. The LED will light while recording is in progress, then light again during transmission.

When the data has been uploaded, you will see a new line appear under 'Collected data'. You will also see the waveform of the audio in the 'RAW DATA' box. You can use the controls underneath to listen to the



Audio waveform

3. Build a dataset

Since you now know how to capture audio with Edge Impulse, it's time to start building a dataset. For a simple audio classification model like this one, we should aim to capture around 10 minutes of data. We have two classes, and it's ideal if our data is balanced equally between each of them. This means we should aim to capture the following data:

- 5 minutes of background noise, with the label "noise"
- 5 minutes of running faucet noise, with the label "faucet"

Real world data

In the real world, there are usually additional sounds present alongside the sounds we care about. For example, a running faucet is often accompanied by the sound of dishes being washed, teeth being brushed,

during real-world usage.

For this tutorial, you should try to capture the following:

- Background noise
  - 2 minutes of background noise without much additional activity
  - 1 minute of background noise with a TV or music playing
  - 1 minute of background noise featuring occasional talking or conversation
  - 1 minutes of background noise with the sounds of housework
- Running faucet noise
  - 1 minute of a faucet running
  - 1 minute of a different faucet running
  - 1 minute of a faucet running with a TV or music playing
  - 1 minute of a faucet running with occasional talking or conversation
  - 1 minute of a faucet running with the sounds of housework

It's okay if you can't get all of these, as long as you still obtain 5 minutes of data for each class. However, your model will perform better in the real world if it was trained on a representative dataset.

> **Dataset diversity**
>
> There's no guarantee your model will perform well in the presence of sounds that were not included in its training set, so it's important to make your dataset as diverse and representative of real-world conditions as possible.

Data capture and transmission

The amount of audio that can be captured in one go varies depending on a device's memory. The ST B-L475E-IOT01A developer board has enough memory to capture 60 seconds of audio at a time, and the Arduino Nano 33 BLE Sense has enough memory for 16 seconds. To capture 60 seconds of audio, set the sample length to `60000`. Because the board transmits data quite slowly, it will take around 7 minutes before a 60 second sample appears in Edge Impulse.

Once you've captured around 10 minutes of data, it's time to start designing an Impulse.

Prebuilt dataset

Alternatively, you can load an example test set that has about ten minutes of data in these classes (but how much fun is that?). See the Running faucet dataset for more information.

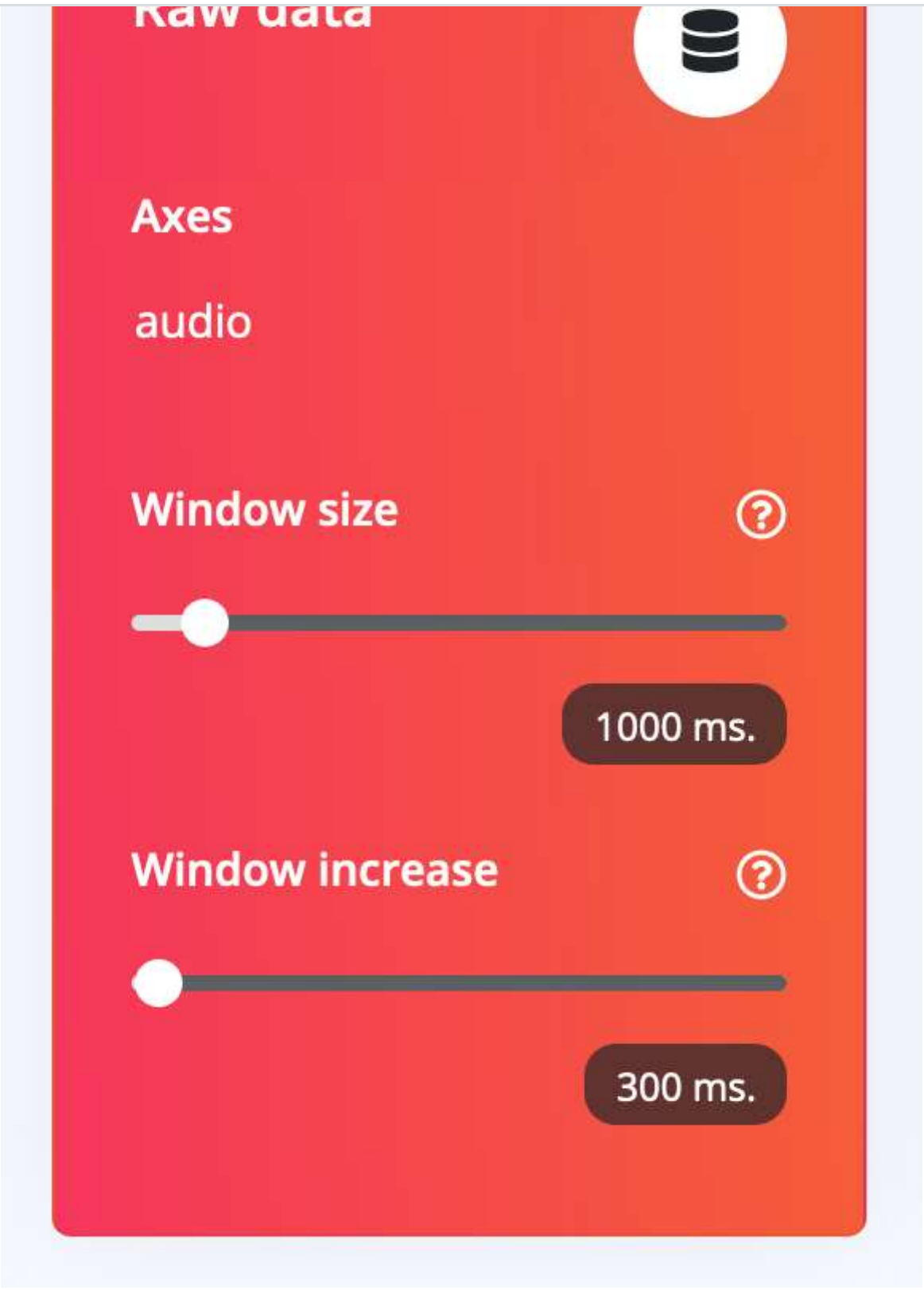
raw data easier to process, while learning blocks learn from past experiences.

For this tutorial we'll use the "MFE" signal processing block. MFE stands for Mel Frequency Energy. This sounds scary, but it's basically just a way of turning raw audio—which contains a large amount of redundant information—into simplified form.

> ### Spectrogram block
>
> Edge Impulse supports three different blocks for audio classification: MFCC, MFE and spectrogram blocks. If your accuracy is not great using the MFE block you can switch to the spectrogram block, which is not tuned to frequencies for the human ear.

We'll then pass this simplified audio data into a Neural Network block, which will learn to distinguish between the two classes of audio (faucet and noise).

In the studio, go to the **Create impulse** tab. You'll see a *Raw data* block, like this one.



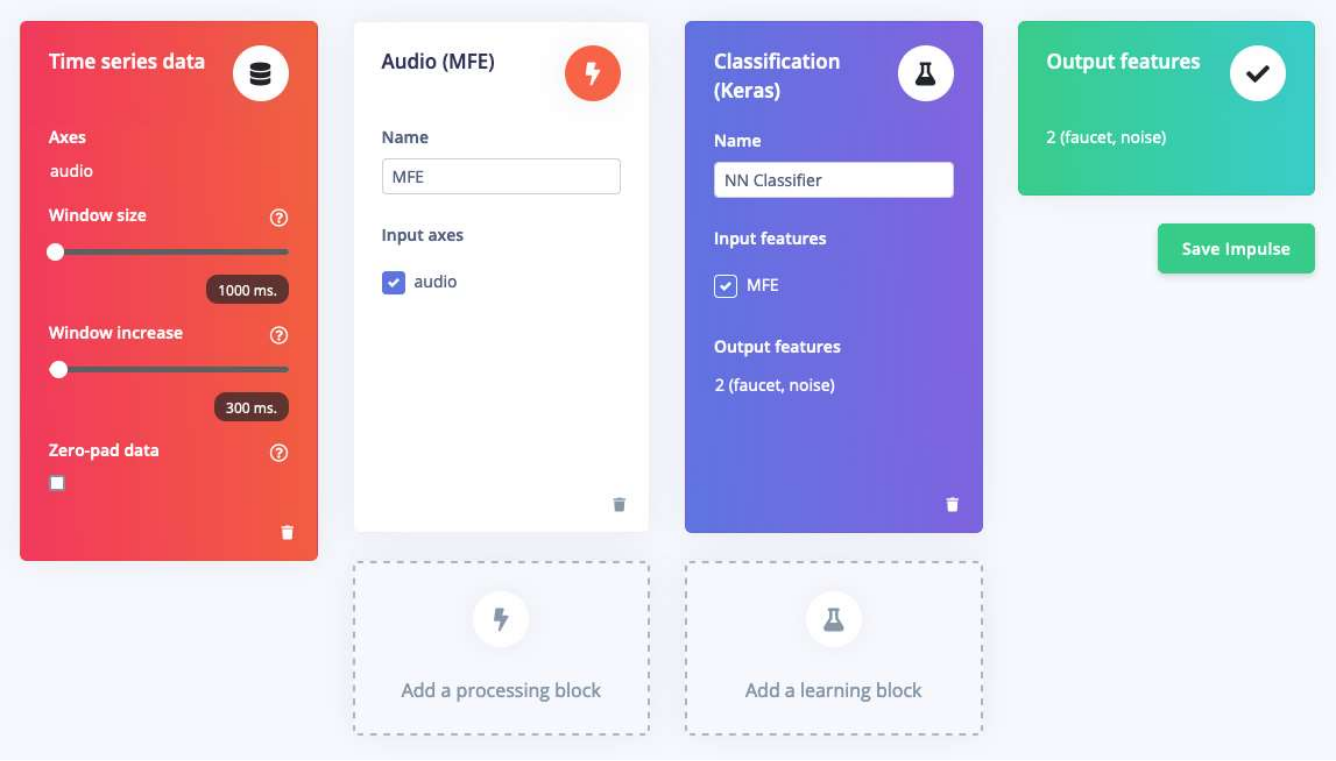
The Raw data block with updated parameters.

Each raw sample is sliced into multiple windows, and the *Window increase* field controls the offset of each subsequent window from the first. For example, a *Window increase* value of 1000 ms would result in each window starting 1 second after the start of the previous one.

By setting a *Window increase* that is smaller than the *Window size*, we can create windows that overlap. This is actually a great idea. Although they may contain similar data, each overlapping window is still a unique example of audio that represents the sample's label. By using overlapping windows, we can make the most of our training data. For example, with a *Window size* of 1000 ms and a *Window increase* of 100 ms, we can extract 10 unique windows from only 2 seconds of data.

Make sure the the *Window increase* field is set to 300 ms. The *Raw data* block should match the screenshot above.

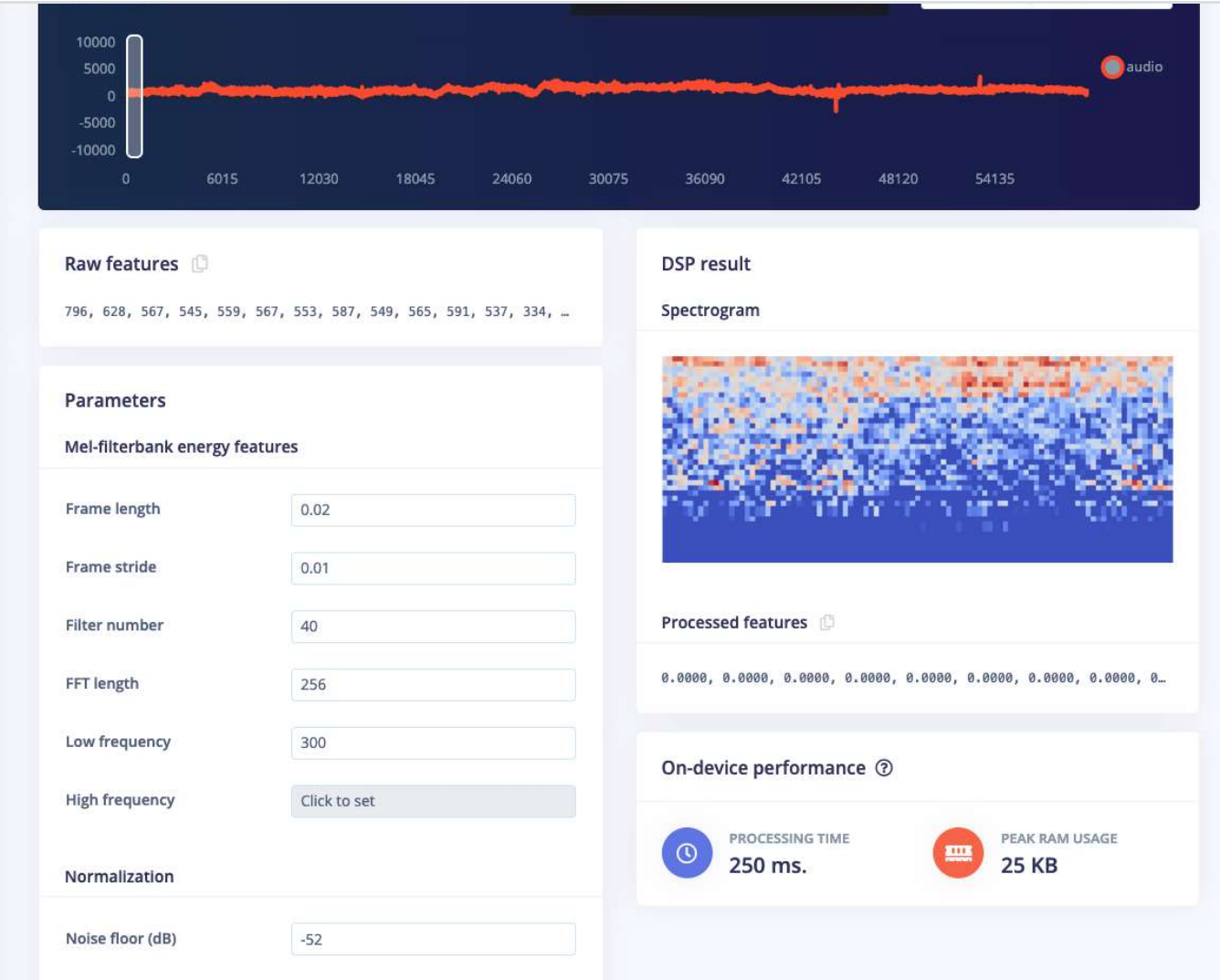
Next, click **Add a processing block** and choose the 'MFE' block. Once you're done with that, click **Add a learning block** and select 'Classification (Keras)'. Finally, click **Save impulse**. Your impulse should now look like this:



The impulse, with one processing block and one learning block.

5. Configure the MFE block

Now that we've assembled the building blocks of our Impulse, we can configure each individual part. Click on the **MFE** tab in the left hand navigation menu. You'll see a page that looks like this:

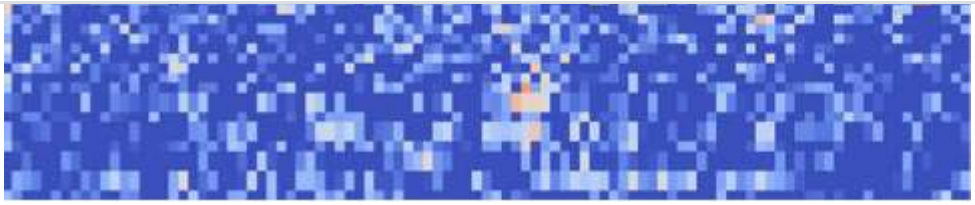


The MFE page.

This page allows you to configure the NFE block, and lets you preview how the data will be transformed. The right of the page shows a visualization of the MFE's output for a piece of audio, which is known as a spectrogram.

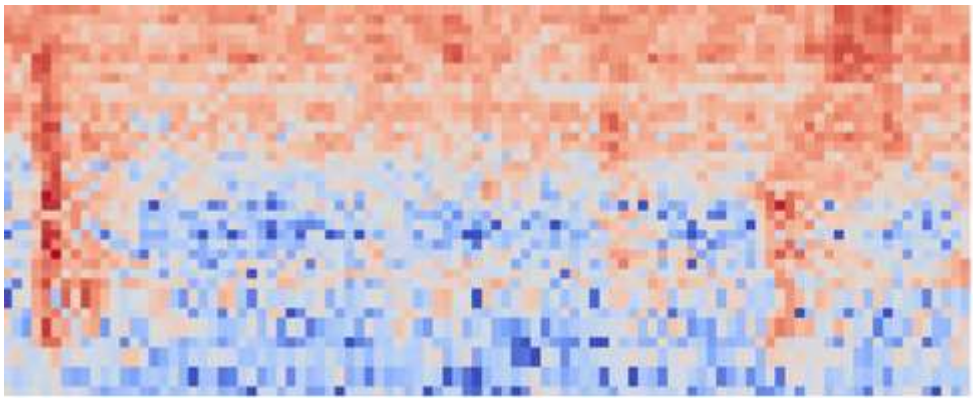
The MFE block transforms a window of audio into a table of data where each row represents a range of frequencies and each column represents a span of time. The value contained within each cell reflects the amplitude of its associated range of frequencies during that span of time. The spectrogram shows each cell as a colored block, the intensity which varies depends on the amplitude.

The patterns visible in a spectrogram contain information about what type of sound it represents. For example, the spectrogram in this image shows a pattern typical of background noise:



Spectrogram of background noise.

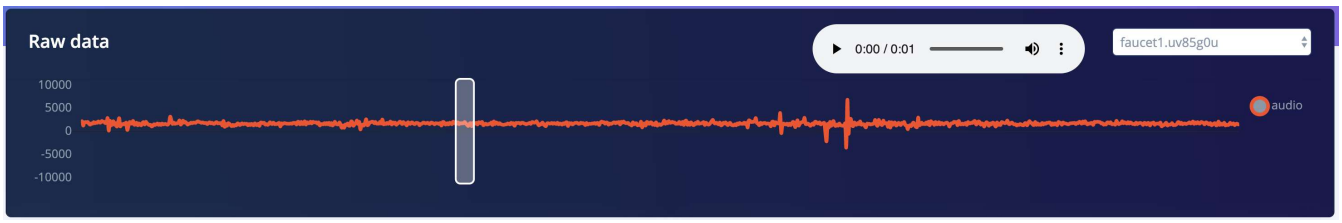
You can tell that it is slightly different from the following spectrogram, which shows a pattern typical of a running faucet:



Spectrogram of a running faucet.

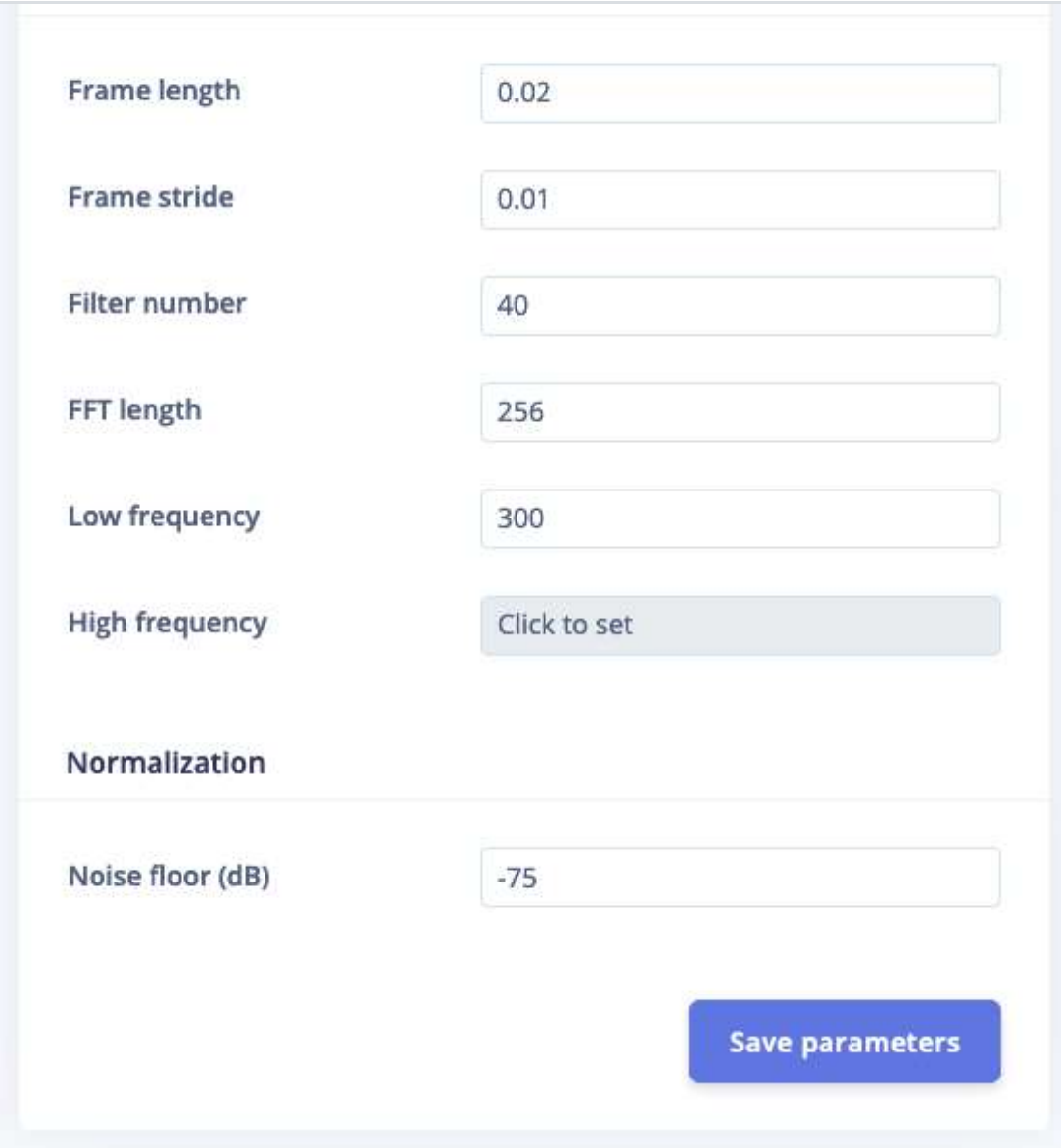
These differences are not necessarily easy for a person to describe, but fortunately they are enough for a neural network to learn to identify.

It's interesting to explore your data and look at the types of spectrograms it results in. You can use the dropdown box near the top right of the page to choose between different audio samples to visualize, and drag the white window on the audio waveform to select different windows of data:



Audio waveform and sample dropdown box.

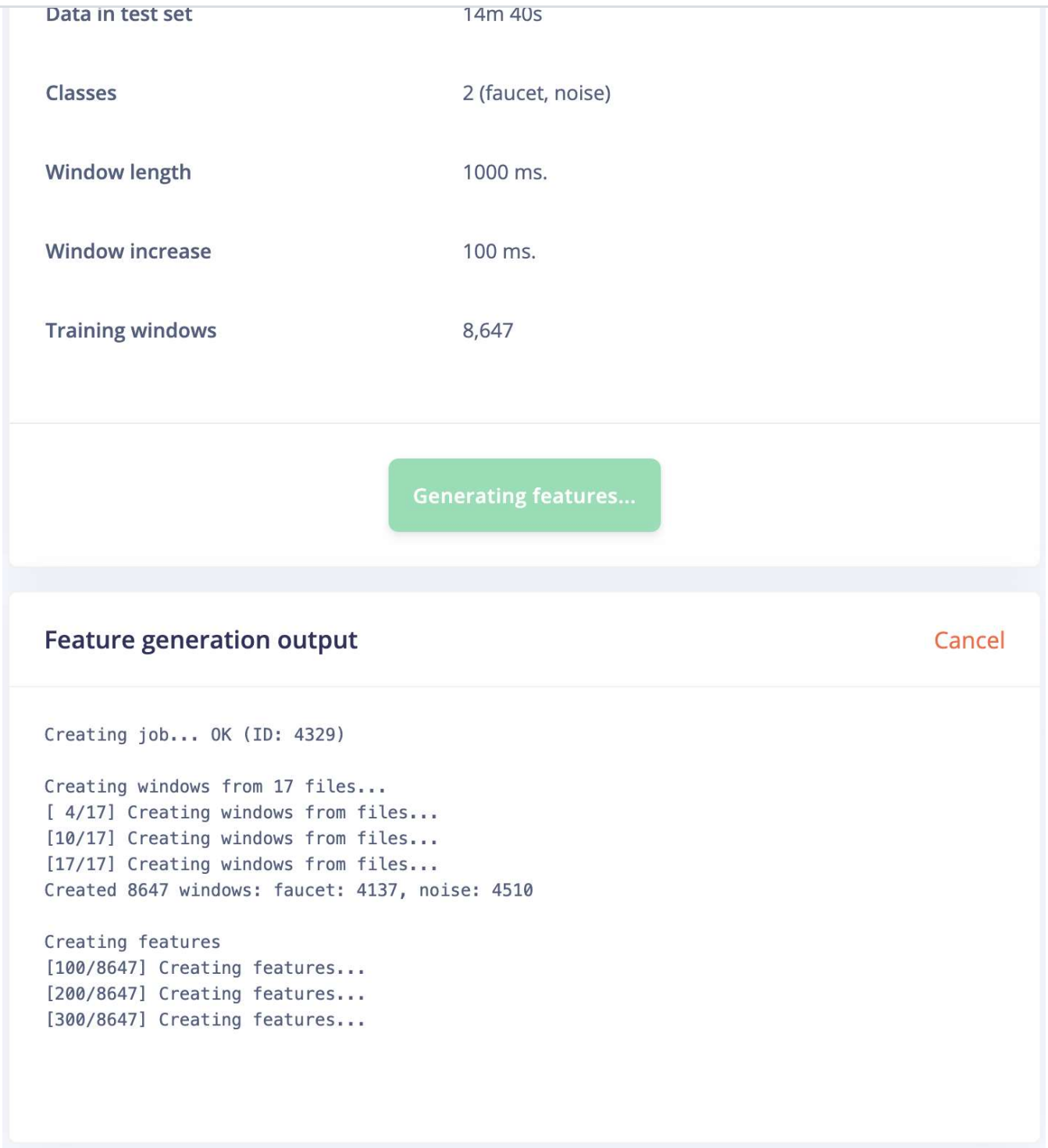
There are a lot of different ways to configure the MFCC block, as shown in the Parameters box:



The MFE parameters box.

Handily, Edge Impulse provides sensible defaults that will work well for many use cases, so we can leave these values unchanged. You can play around with the noise floor to quickly see the effect it has on the spectrogram.

The spectrograms generated by the MFE block will be passed into a neural network architecture that is particularly good at learning to recognize patterns in this type of tabular data. Before training our neural network, we'll need to generate MFE blocks for all of our windows of audio. To do this, click the *Generate features* button at the top of the page, then click the green *Generate features* button. If you have a full 10 minutes of data, the process will take a while to complete:

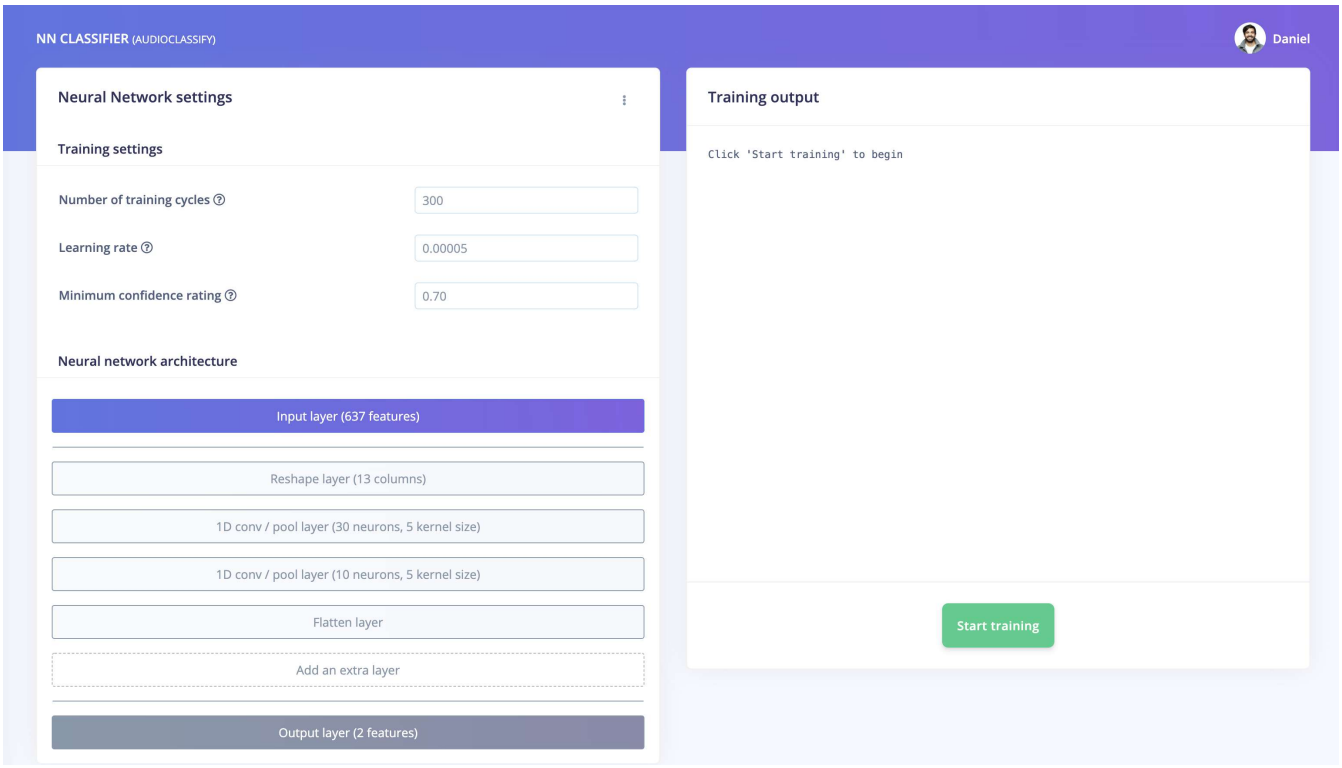


Running the feature generation process.

Once this process is complete the feature explorer shows a visualization of your dataset. Here dimensionality reduction is used to map your features onto a 3D space, and you can use the feature explorer to see if the different classes separate well, or find mislabeled data (if it shows in a different cluster). You can find more information in visualizing complex datasets.

loosely after the human brain, that can learn to recognize patterns that appear in their training data. The network that we're training here will take the MFE as an input, and try to map this to one of two classes—noise, or faucet.

Click on **NN Classifier** in the left hand menu. You'll see the following page:



The NN Classifier page.

A neural network is composed of layers of virtual "neurons", which you can see represented on the left hand side of the NN Classifier page. An input—in our case, an MFE spectrogram—is fed into the first layer of neurons, which filters and transforms it based on each neuron's unique internal state. The first layer's output is then fed into the second layer, and so on, gradually transforming the original input into something radically different. In this case, the spectrogram input is transformed over four intermediate layers into just two numbers: the probability that the input represents noise, and the probability that the input represents a running faucet.

During training, the internal state of the neurons is gradually tweaked and refined so that the network transforms its input in *just* the right ways to produce the correct output. This is done by feeding in a sample of training data, checking how far the network's output is from the correct answer, and adjusting the neurons' internal state to make it more likely that a correct answer is produced next time. When done thousands of times, this results in a trained network.

A particular arrangement of layers is referred to as an *architecture*, and different architectures are useful for different tasks. The default neural network architecture provided by Edge Impulse will work well for our

complete, you'll see the *Model* panel appear at the right side of the page:

The Model panel.

Congratulations, you've trained a neural network with Edge Impulse! But what do all these numbers mean?

On the left hand side of the panel, *Accuracy* refers to the percentage of windows of audio that were correctly classified. The higher number the better, although an accuracy approaching 100% is unlikely, and is often a sign that your model has *overfit* the training data. You will find out whether this is true in the next stage, during model testing. For many applications, an accuracy above 80% can be considered very good.

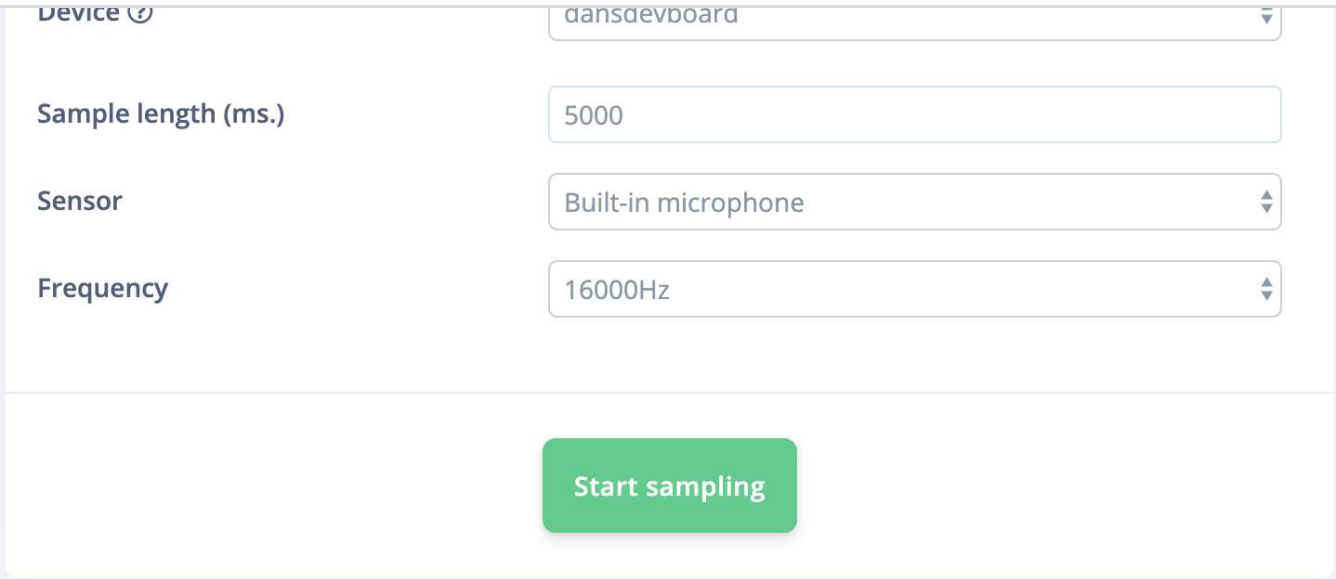
The *Confusion matrix* is a table showing the balance of correctly versus incorrectly classified windows. To understand it, compare the values in each row. For example, in the above screenshot, all of the *faucet* audio windows were classified as *faucet*, but a few *noise* windows were misclassified. This appears to be a great result though.

The *On-device performance* region shows statistics about how the model is likely to run on-device. *Inferencing time* is an estimate of how long the model will take to analyze one second of data on a typical microcontroller (here: an Arm Cortex-M4F running at 80MHz). *Peak memory usage* gives an idea of how much RAM will be required to run the model on-device.

7. Classifying new data

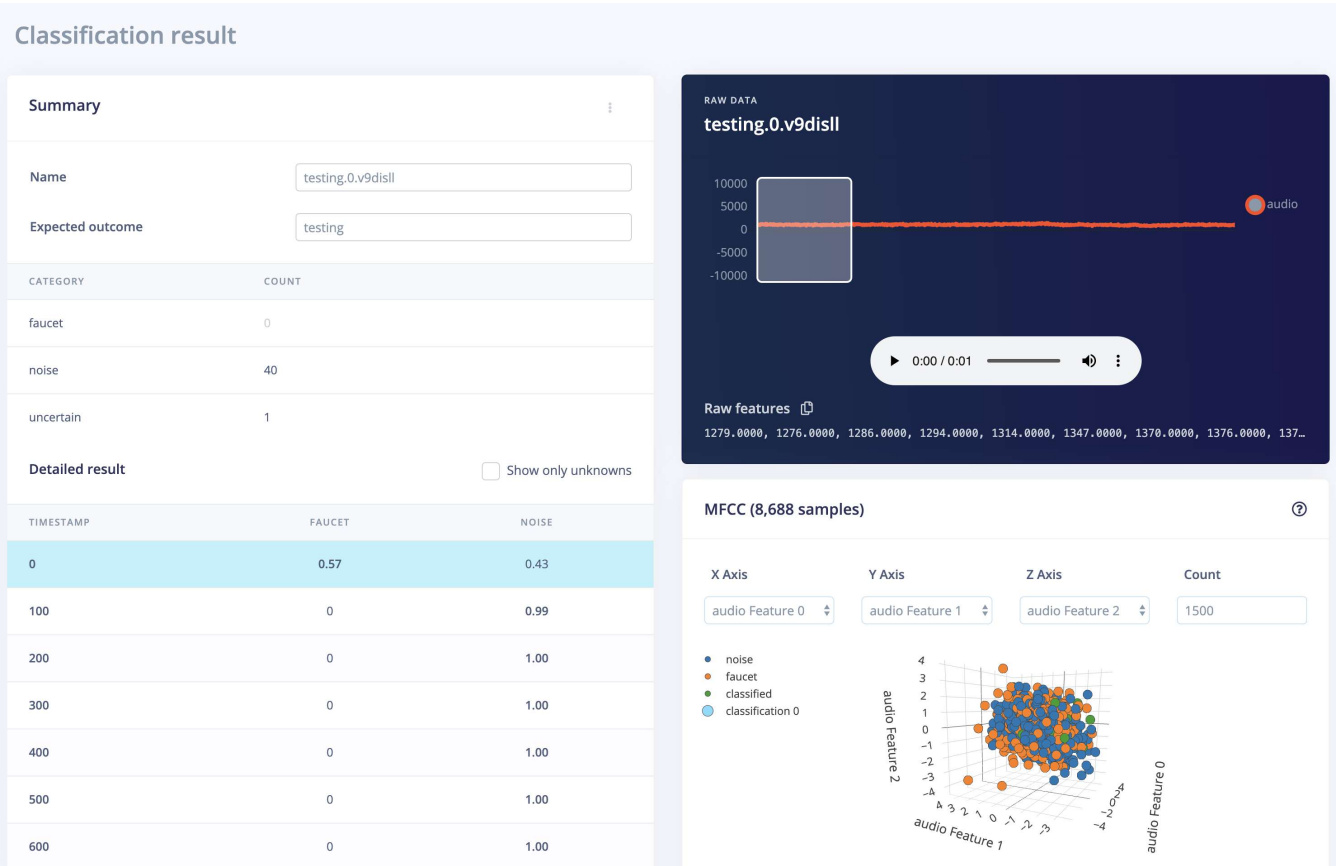
The performance numbers in the previous step show that our model is working well on its training data, but it's extremely important that we test the model on new, unseen data before deploying it in the real world. This will help us ensure the model has not learned to overfit the training data, which is a common occurrence.

Edge Impulse provides some helpful tools for testing our model, including a way to capture live data from your device and immediately attempt to classify it. To try it out, click on **Live classification** in the left hand menu. Your device should show up in the 'Classify new data' panel. Capture 5 seconds of background noise by clicking **Start sampling**:



The Classify new data panel.

The sample will be captured, uploaded, and classified. Once this has happened, you'll see a breakdown of the results:



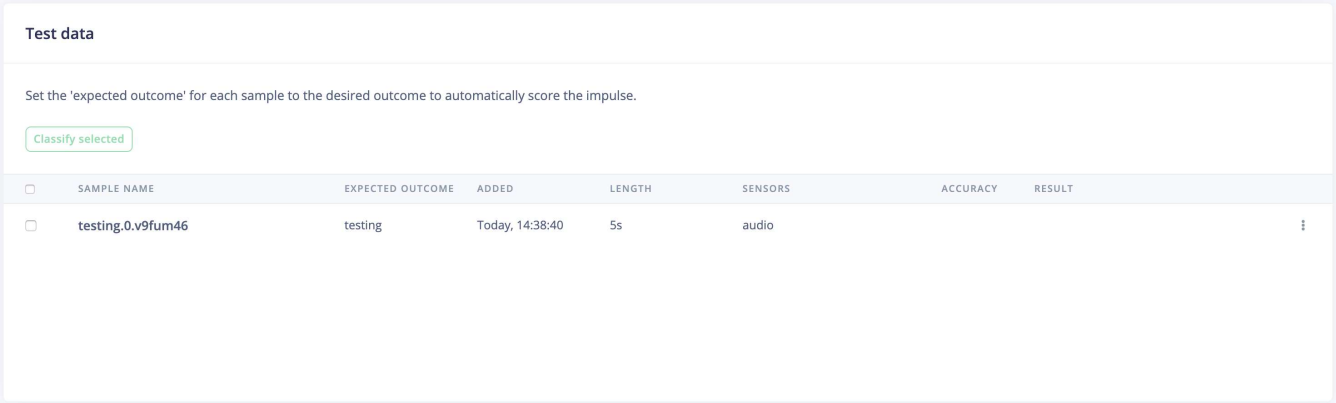
The results of classifying a new sample.

Of course, it's possible some of the windows may be classified incorrectly. Since our model was 99% accurate based on its validation data, you can expect that at least 1% of windows will be classified wrongly—and likely much more than this, since our validation data doesn't represent every possible type of background or faucet noise. If your model didn't perform perfectly, don't worry. We'll get to troubleshooting later.

> ### Misclassifications and uncertain results
>
> It's inevitable that even a well-trained machine learning model will sometimes misclassify its inputs. When you integrate a model into your application, you should take into account that it will not always give you the correct answer.
>
> For example, if you are classifying audio, you might want to classify several windows of data and average the results. This will give you better overall accuracy than assuming that every individual result is correct.

8. Model testing

Using the *Live classification* tab, you can easily try out your model and get an idea of how it performs. But to be really sure that it is working well, we need to do some more rigorous testing. That's where the *Model testing* tab comes in. If you open it up, you'll see the sample we just captured listed in the *Test data* panel:



The Test data panel.

In addition to its training data, every Edge Impulse project also has a test dataset. Samples captured in *Live classification* are automatically saved to the test dataset, and the *Model testing* tab lists all of the test data.

To use the sample we've just captured for testing, we should correctly set its expected outcome. Click the ⋮ icon and select **Edit expected outcome**, then enter `noise`. Now, select the sample using the checkbox to the left of the table and click **Classify selected**:

☑	testing.0.v9fum46	noise	Today, 14:38:40	5s	audio	100%	41 noise

Test data classification results.

You'll see that the model's accuracy has been rated based on the test data. Right now, this doesn't give us much more information that just classifying the same sample in the *Live classification* tab. But if you build up a big, comprehensive set of test samples, you can use the *Model testing* tab to measure how your model is performing on real data.

Ideally, you'll want to collect a test set that contains a minimum of 25% the amount of data of your training set. So, if you've collected 10 minutes of training data, you should collect at least 2.5 minutes of test data. You should make sure this test data represents a wide range of possible conditions, so that it evaluates how the model performs with many different types of inputs. For example, collecting test audio for several different faucets is a good idea.

You can use the *Data acquisition* tab to manage your test data. Open the tab, and then click **Test data** at the top. Then, use the *Record new data* panel to capture a few minutes of test data, including audio for both background noise and faucet. Make sure the samples are labelled correctly. Once you're done, head back to the *Model testing* tab, select all the samples, and click **Classify selected**:

Test data

Set the 'expected outcome' for each sample to the desired outcome to automatically score the impulse.

ACCURACY 85.14%

Classify selected (21)

☑	SAMPLE NAME	EXPECTED OUTCOME	ADDED	LENGTH	SENSORS	ACCURACY	RESULT
☑	testing0.v6datj8	noise	Yesterday, 09:55:12	5s	audio	95%	39 noise, 2 uncertain
☑	classification.10.v491vsl	noise	Jan 22 2020, 14:01:57	5s	audio	70%	29 noise, 9 faucet, 3 uncertain
☑	classification.9.v48qrtp	noise	Jan 22 2020, 13:58:03	1s	audio	100%	1 noise
☑	classification.8.v48kqf5	noise	Jan 22 2020, 13:54:45	1s	audio	100%	1 noise
☑	classification11.v20t3rk	noise	Jan 21 2020, 17:01:00	10s	audio	95%	87 noise, 4 faucet
☑	classification10.v1vcpvj	noise	Jan 21 2020, 16:34:37	10s	audio	62%	57 noise, 30 faucet, 4 uncertain
☑	classification9.v1v2re1	noise	Jan 21 2020, 16:29:10	10s	audio	82%	75 noise, 10 faucet, 6 uncertain
☑	classification8.v1utjed	noise	Jan 21 2020, 16:26:18	1s	audio	100%	1 noise
☑	faucet1.v1m71av	faucet	Jan 21 2020, 13:54:11	1m 0s	audio	89%	529 faucet, 46 noise, 16 uncertain
☑	noise12.uvcljru	noise	Jan 20 2020, 16:28:54	1m 0s	audio	87%	516 noise, 45 faucet, 30 uncertain

Test results for a large number of samples.

The screenshot shows classification results from a large number of test samples (there are more on the page than would fit in the screenshot). The panel shows that our model is performing at 85% accuracy, which is 5% less than how it performed on validation data. It's normal for a model to perform less well on entirely fresh data, so this is a successful result. Our model is working well!

you do this, you should add some new test data to make up for the loss.

Testing your model helps confirm that it works in real life, and it's something you should do after every change. However, if you often make tweaks to your model to try to improve its performance on the test dataset, your model may gradually start to overfit to the test dataset, and it will lose its value as a metric. To avoid this, continually add fresh data to your test dataset.

> **Data hygiene**
>
> It's extremely important that data is never duplicated between your training and test datasets. Your model will naturally perform well on the data that it was trained on, so if there are duplicate samples then your test results will indicate better performance than your model will achieve in the real world.

9. Model troubleshooting

If the network performed great, fantastic! But what if it performed poorly? There could be a variety of reasons, but the most common ones are:

1. The data does not look like other data the network has seen before. This is common when someone uses the device in a way that you didn't add to the test set. You can add the current file to the test set by adding the correct label in the 'Expected outcome' field, clicking ⋮, then selecting **Move to training set**.
2. The model has not been trained enough. Increase number of epochs to `200` and see if performance increases (the classified file is stored, and you can load it through 'Classify existing validation sample').
3. The model is overfitting and thus performs poorly on new data. Try reducing the number of epochs, reducing the learning rate, or adding more data.
4. The neural network architecture is not a great fit for your data. Play with the number of layers and neurons and see if performance improves.

As you see, there is still a lot of trial and error when building neural networks. Edge Impulse is continually adding features that will make it easier to train an effective model.

10. Deploying to your device

With the impulse designed, trained and verified you can deploy this model back to your device. This makes the model run without an internet connection, minimizes latency, and runs with minimum power consumption. Edge Impulse can package up the complete impulse - including the MFE algorithm, neural network weights, and classification code - in a single C++ library that you can include in your embedded software.

To export your model, click on **Deployment** in the menu. Then under 'Build firmware' select your development board, and click **Build**. This will export the impulse, and build a binary that will run on your development board in a single step. After building is completed you'll get prompted to download a binary. Save this on your computer.

Flashing the device

When you click the **Build** button, you'll see a pop-up with text and video instructions on how to deploy the binary to your particular device. Follow these instructions. Once you are done, we are ready to test your impulse out.

Running the model on the device

We can connect to the board's newly flashed firmware over serial. Open a terminal and run:

```
$ edge-impulse-run-impulse
```

> **Serial daemon**
>
> If the device is not connected over WiFi, but instead connected via the Edge Impulse serial daemon, you'll need stop the daemon. Only one application can connect to the development board at a time.

This will capture audio from the microphone, run the MFE code, and then classify the spectrogram:

```
Starting inferencing in 2 seconds...
Recording
Recording OK
Predictions (DSP: 399 ms., Classification: 175 ms., Anomaly: 0 ms.):
    faucet: 0.03757
    noise: 0.96243
Starting inferencing in 2 seconds...
```

Great work! You've captured data, trained a model, and deployed it to an embedded device. It's time to celebrate—by pouring yourself a nice glass of water, and checking whether the sound is correctly classified by you model.

Machine learning is thirsty work.

11. Conclusion

Congratulations! you've used Edge Impulse to train a neural network model capable of recognizing a particular sound. There are endless applications for this type of model, from monitoring industrial machinery to recognizing voice commands. Now that you've trained your model you can integrate your impulse in the firmware of your own embedded device, see Running your impulse locally. There are examples for Mbed OS, Arduino, STM32CubeIDE, Eta Compute, and any other target that supports a C++ compiler.

Or if you're interested in more, see our tutorials on Continuous motion recognition or Adding sight to your sensors. If you have a great idea for a different project, that's fine too. Edge Impulse lets you capture data from any sensor, build custom processing blocks to extract features, and you have full flexibility in your Machine Learning pipeline with the learning blocks.

We can't wait to see what you'll build!

Updated about a month ago