WHITE PAPER

Deep Learning for Signal Processing with MATLAB

MathWorks®

Introduction

Deep learning networks have been used for image classification for many years, but they are powerful tools for signal data as well. A deep learning network can do everything a mathematical model can do without requiring you to know which signal features to bring into the model. The network itself decides these features during the training process. Whether the input data is a 1D signal, time-series data, or even text, deep learning networks such as convolutional neural networks (CNNs) make it possible to process data in new ways.

This white paper reviews some deep learning basics and then describes three signal processing examples:

- Speech command recognition
- Remaining useful life (RUL) estimation
- Signal denoising

These examples will show you how deep learning with MATLAB® can make signal processing tasks quicker and with more accurate results.

Deep Learning Basics

Deep learning is a subtype of machine learning in which a model learns to perform classification and regression tasks directly from images, text, or sound. With machine learning or traditional signal processing, you would manually select the relevant features in the data. With deep learning, the model learns and abstracts the relevant information automatically as the data passes through the network.

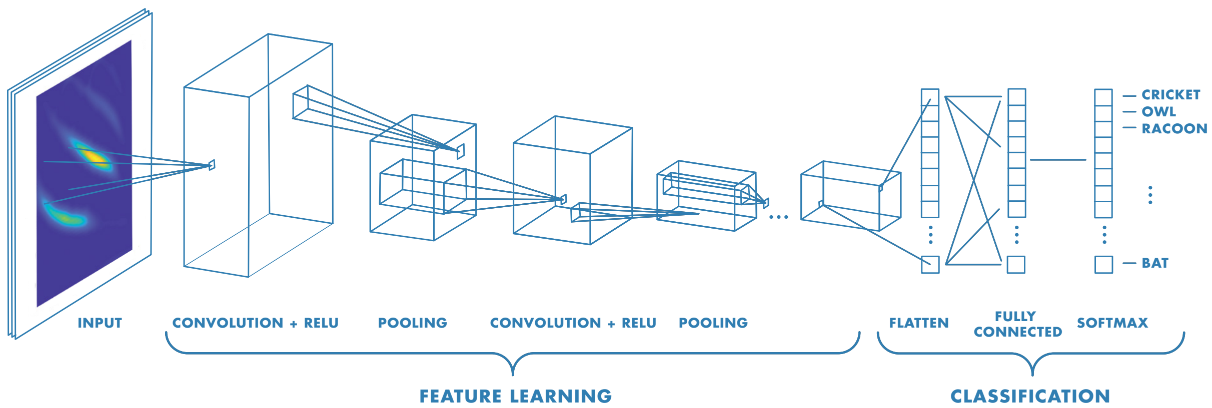
Deep learning is usually implemented using a neural network architecture. The term “deep” refers to the number of layers in the network—the more layers, the deeper the network. The layers are interconnected via nodes, or neurons, with each hidden layer using the output of the previous layer as its input.

Deep Learning Networks

Two of the most popular deep learning networks are:

- Convolutional neural network (CNN, or ConvNet)
- Long short-term memory (LSTM)

A CNN is composed of an input layer, an output layer, and several hidden layers in between. Three of the most common layers are convolution, activation or ReLU, and pooling. These operations are repeated over tens or hundreds of layers, with each layer learning to detect different features in the input data.



Architecture of a CNN comprising commonly used layers such as convolution, ReLU, and pooling.

An LSTM is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data. Unlike a CNN, an LSTM can remember the state of the network between predictions.

The core components of an LSTM network are a sequence input layer and an LSTM layer. A sequence input layer incorporates time-series data into the network. An LSTM layer learns long-term dependencies between time steps of sequence data over time.

Selecting a Network

A CNN is typically used in signal and time series prediction when the input signal has been converted into an "image" by transforming it from 1D into a 2D representation.

An LSTM is good for classifying sequence and time-series data, when the prediction or output of the network must be based on a remembered sequence of data points.

Considerations for Signal Data

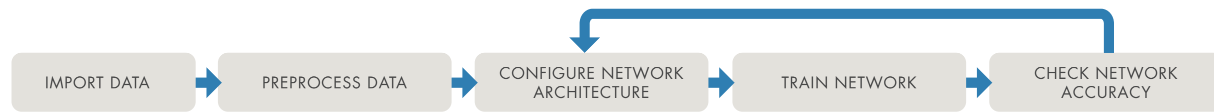
Deep learning with signal data typically requires more preprocessing and feature extraction than deep learning with image data. Often, you can pass raw images into a network and get results, but the same workflow is highly unlikely to work with signals. Most raw signal data is noisy, variable, and smaller than image data. Careful preprocessing of the data available ensures the best possible model.

Should you use machine learning instead?

If you understand your data but have a limited amount, you may get better results if you use machine learning and manually extract the most relevant features. While the results can be impressive, this approach requires more signal processing knowledge than deep learning.

The Workflow

The main steps in the deep learning workflow are as follows:



No matter which network configuration you use, or which problem you are using deep learning to solve, the process is always iterative: If the network does not give you the accuracy you're looking for, you go back and change the setup to see if there are ways to improve it.

Each of the following examples follows this basic workflow.

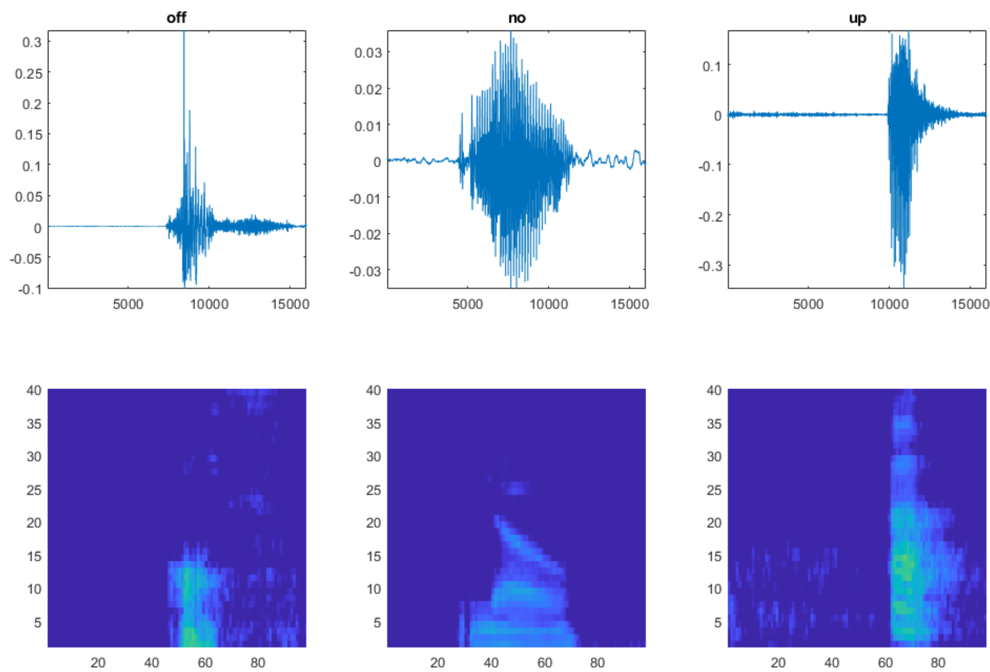
Example 1. Speech Command Recognition: Classifying Speech Audio Files

We want to train a simple deep learning model to recognize and identify individual speech commands in an audio recording.

In classic signal processing, we would need to identify and extract the relevant features (components of the signal that identify speech) from the signal and pass them to a decoder model. Carefully preprocessing signals (for example, to reduce background noise or reverberation) and selecting a specific set of features could still yield poor results unless the full system is skillfully adapted to the problem in hand.

With deep learning, while we still need to extract relevant speech features from the signal, we can then approach the problem as an image classification problem. The first step is to convert the audio files into spectrograms—and since a spectrogram is a 2D visualization of the signals in a 1D audio file, we can use it as input to a CNN, just as we would use an actual image.

The result is a robust model that handles the nuances of a signal. Without deep learning, a signal processing engineer would have to identify, uncover, and process the relevant information from the signal manually.



Top: Original audio signals. Bottom: The audio signals' corresponding spectrograms.

Importing the Data

This example uses a data set from TensorFlow™, an open-source software library for high-performance numerical computation.

Follow along in MATLAB: *Get the full example and a link to the data set*

Preparing the Data

To prepare the data for efficient training of a CNN, we convert the speech waveforms to spectrograms using the `melSpectrogram` command, a variation on the simpler spectrogram function in MATLAB. Speech processing is a specialized form of audio processing, with features (components of the signal that identify the speech) localized in specific frequencies. Because we want the CNN to focus on the frequency areas in which speech is most relevant, we use Mel spacing for frequencies, which distributes sensitivity similarly to how humans hear.

A spectrogram is just one of many time-frequency transforms to represent raw signal data as a 2D image. Other commonly used techniques include wavelet transforms.

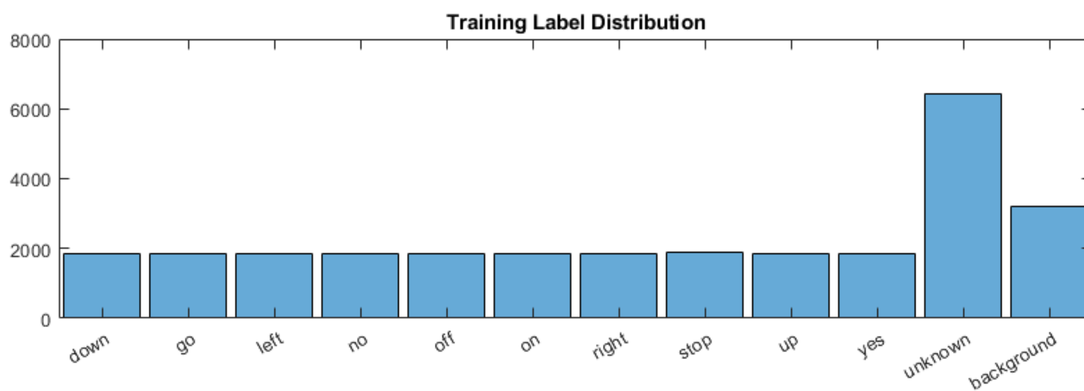
Wavelet transforms (such as scalograms and continuous wavelet transforms) can produce spectrogram-like 2D representations with increased time resolution, especially for non-periodic signal components.

Wavelets are also used in an automated feature extraction technique called *wavelet scattering* or "invariant scattering convolutional networks." This technique mimics the features learned in the first few layers of a CNN. However, they use fixed weight patterns, which don't need to be learned from data, significantly reducing requirements for network complexity and training data size.

Next, we divide the data into three sets:

- **Training data:** the original input data, which the network will use to learn and understand features
- **Validation data:** to be used while the network is training to verify the algorithm is learning features while also generalizing its understanding of the data
- **Testing data:** data the network has never seen before, which will be passed into the network after training to assess the network performance while making sure the result is unbiased

We distribute the training data evenly between the classes of words we want to classify.



Evenly distributed training data.

To reduce false positives, we include a category for words likely to be confused with the intended categories. For example, if the intended word is "on," then words like "mom", "dawn", and "won" would be placed in the "unknown" category. The network does not need to know these words, just that they are NOT the words to recognize.

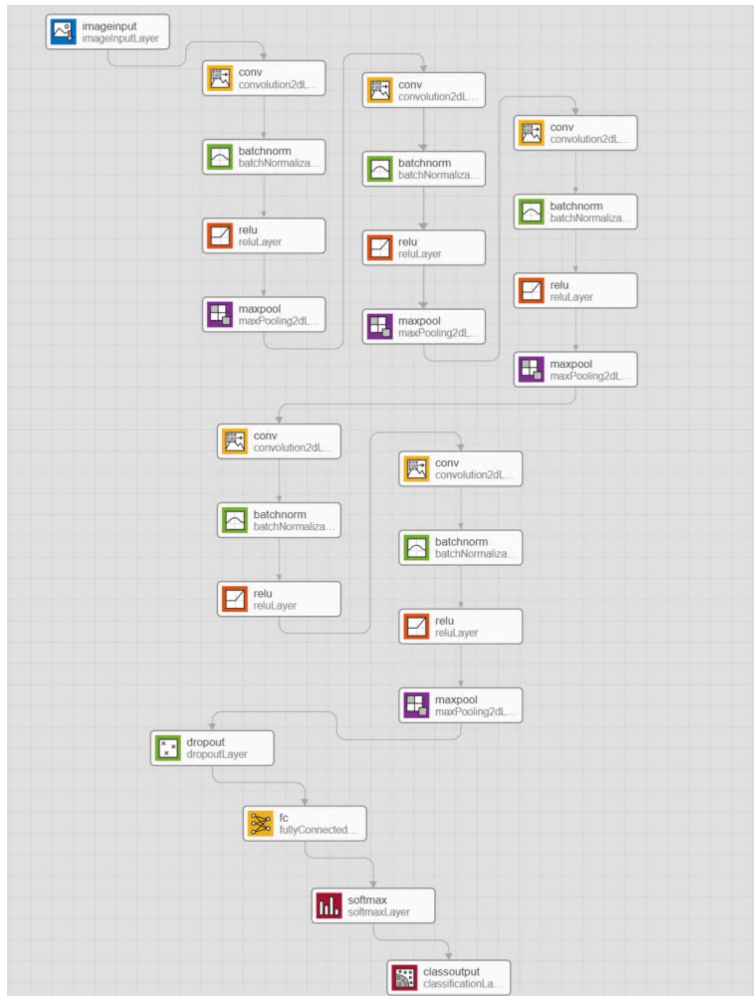
Configuring the Network Architecture

Because we are using the spectrogram as an input, the structure of our CNN can be similar to other networks used for image classification. For this application, we'll use a network consisting of 48 layers.

> **How do I define a network architecture?**
> A common approach is to start with a network in a research paper or other published source and modify it as needed.

A network consisting of 48 layers sounds complex, but the basic structure is simple: It's a recurring series of convolution, batch normalization, and ReLU layers.

The entire model can be visualized as follows:



It is up to the designer of the network to determine how many iterations of convolution, batch normalization, and ReLU to perform, but keep in mind that the more complex the network, the more data required to train and test it.

Training the Network

Before beginning the training, we specify the training options, such as:

- Number of epochs (complete passes through the training data)
- Learning rate (speed of training)
- Processor (generally, a CPU or GPU)

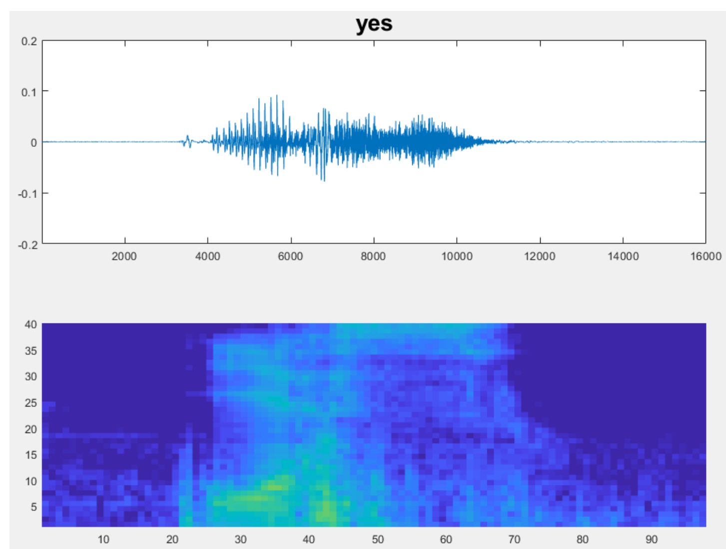
We then pass samples of the signal into the network, plotting progress as it trains.

The network will learn to identify unique features in these samples and classify them accordingly. As we're dealing with spectrogram data, the learning process is essentially image classification: The initial layers of the network learn broad features in the spectrogram, such as color, thickness, and orientation of lines and shapes. As training continues, the network identifies more fine-grained features of the individual signals.

Checking Network Accuracy

After the model has been trained, either because it's completed the number of epochs we specified or because it's reached a specified stopping point, it will classify the input image (spectrogram) into the appropriate categories.

To test the results, we compare the network prediction with the corresponding audio clip. The accuracy on the validation set is about 96%.



Plot and spectrogram of a word classified as "yes."

This result might be acceptable for a simple speech-recognition task, but if we wanted to implement the model in, say, a security system or a voice-activated device, we'd need higher accuracy.

This is where we really start to see the value of a deep learning approach: It gives us a variety of options for achieving greater accuracy. We can improve the result by simply adding more training samples to the network. To gain more insight into where the network is making incorrect predictions, we can visualize a confusion matrix.

Confusion Matrix for Validation Data

True Class \ Predicted Class	yes	no	up	down	left	right	on	off	stop	go	unknown	background		
yes	251	1	1		2		1			1	3	1	96.2%	3.8%
no	1	262		1						4	2		97.0%	3.0%
up			245					5		2	5	3	94.2%	5.8%
down		8		250					2	2	1	1	94.7%	5.3%
left	1	2			242						1	1	98.0%	2.0%
right		1			4	250	1						97.7%	2.3%
on			2				239	8		1	4	3	93.0%	7.0%
off			9				2	244			1		95.3%	4.7%
stop		1	3		4				236			2	95.9%	4.1%
go		4		2			1		1	245	3	4	94.2%	5.8%
unknown	1	6	4	6	10	6	9	6	7	7	835	3	92.8%	7.2%
background												400	100.0%	
	98.8%	91.9%	92.8%	96.5%	92.4%	97.7%	94.5%	92.8%	95.9%	93.5%	97.7%	95.7%		
	1.2%	8.1%	7.2%	3.5%	7.6%	2.3%	5.5%	7.2%	4.1%	6.5%	2.3%	4.3%		

The confusion matrix shows the network's predicted class (word) versus the true class. The blue shades indicate correct predictions, while beige shows incorrect predictions.

With deep learning we can keep going until we get the accuracy we want. With traditional signal processing we would need considerable expertise to understand the nuances of the signal.

Example 2. Predicting Remaining Useful Life: Sequence-to-Sequence Regression

In this example, we want to predict the amount of time an engine has left before it fails, or its *remaining useful life* (RUL). RUL is a key metric for timing engine maintenance and avoiding unplanned delays.

While many deep learning networks predict a class or category, the output we're looking for is numeric—the number of cycles until the engine will fail—making this a regression problem.

We calculate RUL using "run-to-failure" data, which shows how a sensor value changes according to the relative health of the system.

Without deep learning, the workflow for approaching this problem would be:

1. Choose which sensors are the best indicators of system health.
2. Derive a metric that uses a combination of these sensors.
3. Track data continuously over time to see if the system is in danger of failing.
4. Fit a model that returns a probability of failure or time to failure.

The first step, choosing the sensors (feature selection), is no easy task. A system often has upwards of 50 sensors, not all of which exhibit signs of stress as the system approaches failure. With deep learning, the model does the feature extraction for us. The network identifies which sensors are the best predictors of a problem.

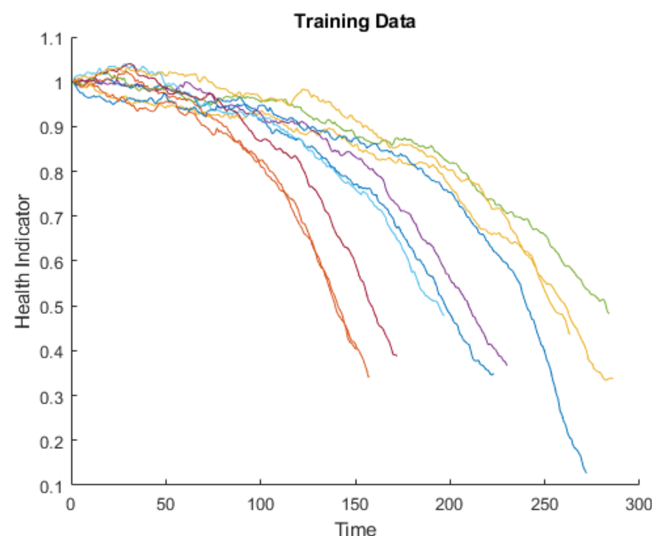
Importing the Data

We start with the Turbofan Engine Degradation Simulation data set from the NASA Prognostic Data repository.

Follow along in MATLAB: *Get the full example and a link to the data set*

The data set contains training and test observations, as well as a vector of actual RUL values to use for validation.

Each time series represents a single engine. Each engine operates normally at the start of a time series and develops a fault at some point during the series. In the training set, the fault grows to the point of system failure.



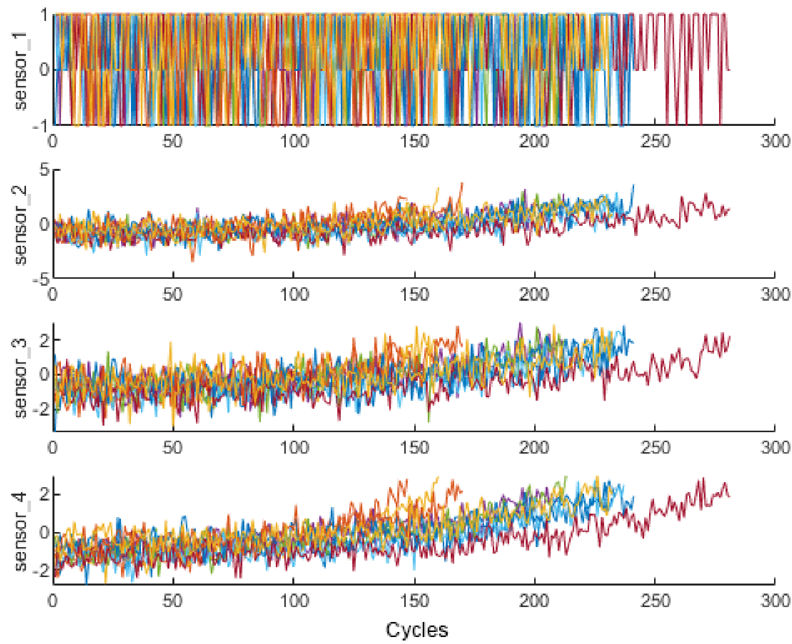
A view of training data from 0 to 300 cycles.

Preparing the Data

To ensure accurate predictions, we want to give the network the best possible data to work with. To simplify and clean up the data, we do the following:

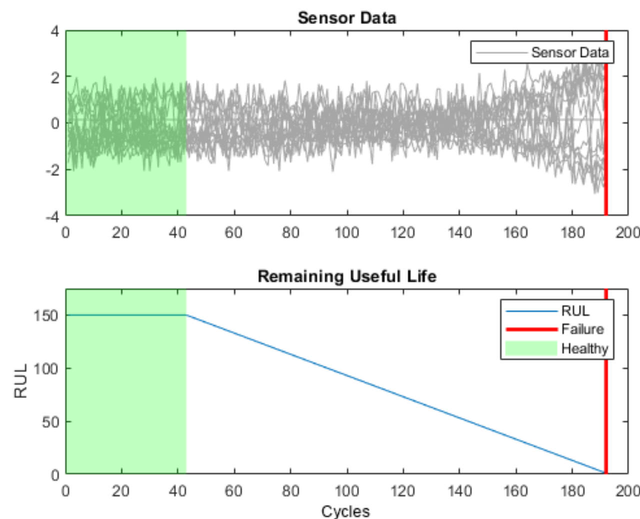
Remove signals with constant values. Signals that remain constant for all time steps may not contribute to the accuracy of the trained model, so signals with the same minimum and maximum values can be removed.

Normalize training predictors. As different signals span different ranges, we need to normalize the data. In this example, we assume a zero mean and scale between -4 and 4. To calculate the mean and standard deviation over all observations, we concatenate the sequence data horizontally.



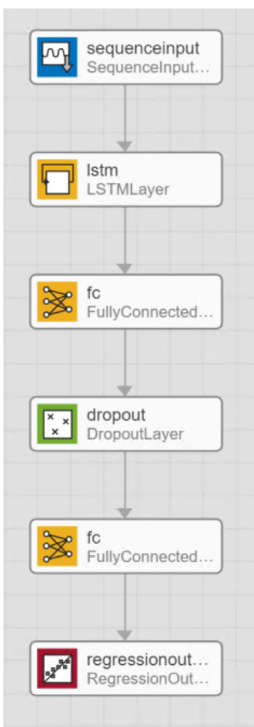
Data from sensors 1-4 after normalization.

Clip responses. Clipping the responses focuses the network on the point at which the system starts to fail. The figure below shows the sensor data for one engine (top plot) and the remaining useful life for the sensor data at each point (bottom plot). Initial values of the signal (shown in green) that are more than 150 cycles from failure have their remaining useful life value capped at 150, forcing the network to be somewhat conservative.



Top: Sensor data for one engine. Bottom: RUL for the sensor data at each point.

Configuring the Network Architecture



An LSTM is a good fit for this kind of application because it can learn dependencies between time steps of sequence data. Sensors are often interconnected, and an event that happened two cycles ago can affect the future performance of the system.

We define an LSTM network that consists of a layer with 200 hidden units and a dropout layer with dropout probability of 0.5.

Dropout probability is a tool to prevent overfitting. The network uses this value to randomly skip some data to avoid memorizing the training set. Hidden units usually number in the hundreds. Determining how many hidden units to include is a tradeoff. With too few, the model won't have enough memory to learn. With too many, the network might overfit.

Training the Network

We begin by setting two training options:

60 epochs with mini-batches of size 20 using the solver 'adam'

Learning rate to 0.01

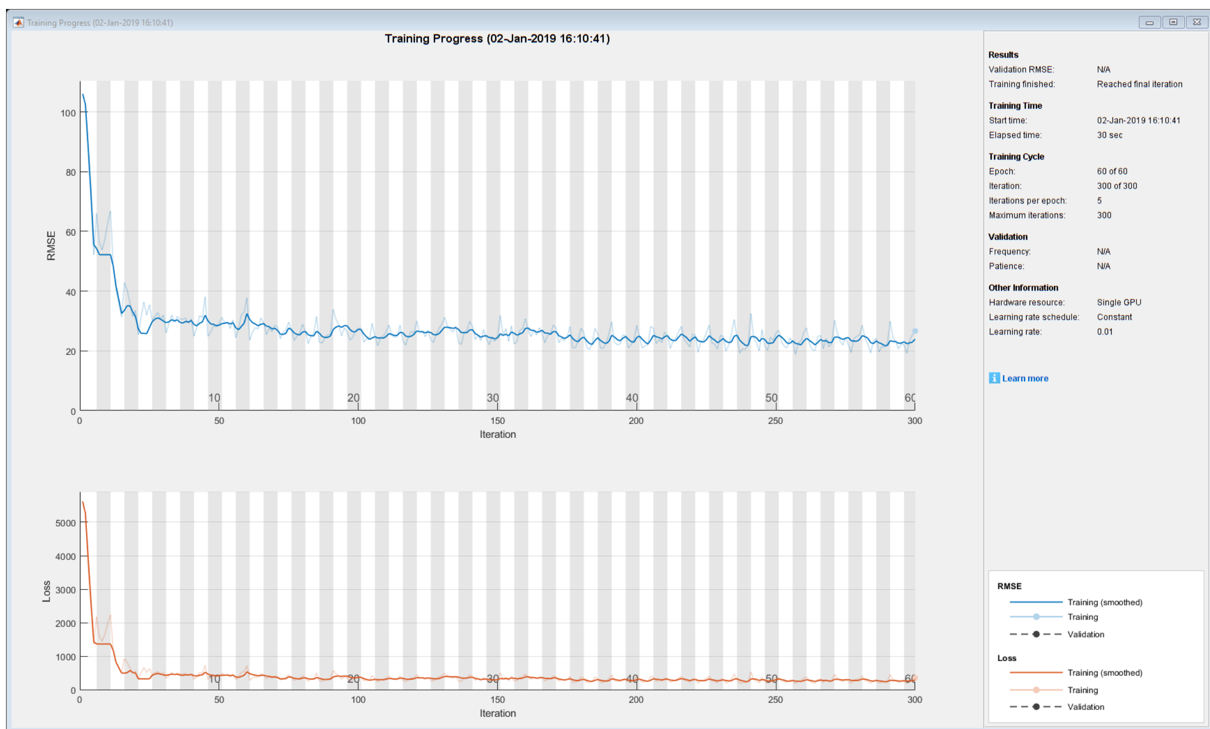
With these settings, training the network will take about two minutes. We plot the training progress while the network is running.

What Are Solvers?

Solvers are algorithms that optimize neural networks.

`'adam'` (adaptive moment estimation) is the most common solver used with LSTM networks, and it can often be used with default settings. `'adam'` keeps an element-wise moving average of both the parameter gradients and their squared values.

Get more information on `'adam'` and other solvers, including `'sgdm'` and `'rmsprop'`.



Training results for the RUL model.

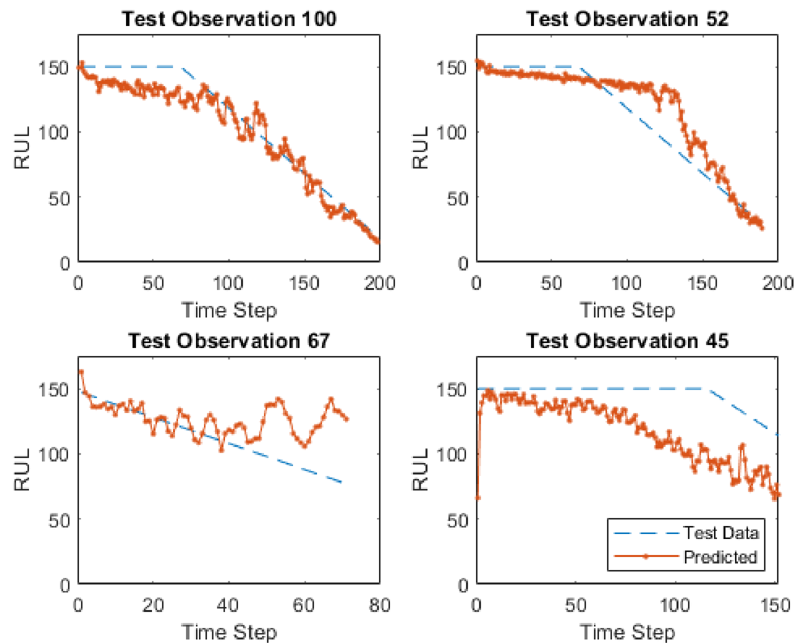
Because predicting RUL is a regression problem, the first plot shows the progression of the root mean squared error (RMSE) instead of accuracy. The lower the number, the closer the predicted time to failure to the actual values.

Checking Network Accuracy

After we train the network, we validate it with testing data.

The LSTM network makes predictions on the partial sequence one time step at a time. At each cycle, the network updates its internal state and predicts the length of time to failure—the RUL value.

To visualize the performance of the network, we select data from four test engines and plot the true remaining useful life against the prediction.



Model predictions for four engines (100, 52, 67, and 45) comparing the actual RUL (dotted blue line) and the estimated RUL (orange line).

The LSTM provides reasonably accurate time-to-failure estimates for engines 100 and 45, but less accurate estimates for engines 52 and 67. This result may indicate that we need more training data representative of the engine's performance in those cases. Alternatively, we could tweak the network training settings and see if performance improves.

In this example we have designed, trained, and tested a simple LSTM network and used that network to predict remaining useful life. By using deep learning instead of traditional signal processing, we eliminated the difficult and time-consuming task of feature extraction.

Example 3. Denoise Speech Using a Fully Connected Neural Network

We want to remove washing machine noise from speech signals while enhancing the signal's quality and intelligibility. Engineers denoise signals to improve data quality—for example, to pass a better recording to a cloud-based voice assistant engine or to increase the signal-to-noise ratio in an ECG.

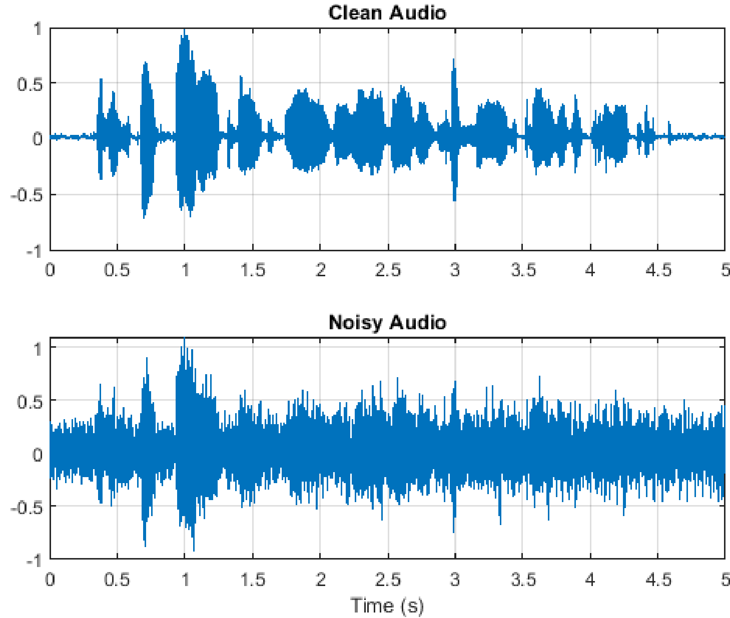
Classic signal processing techniques were (and still are) implemented to filter the noise. The challenge with these approaches (spectral subtraction, noise gating, etc.) is that the quality of the good signal is sometimes unintentionally filtered out as well, losing some of the original signal's information.

By using deep learning to denoise speech in this example, we'll be able to remove washing machine noise from a speech signal while minimizing undesired artifacts in the output speech. The output of the network can then be used to filter out noise in other similar noisy circumstances.

Importing the Data

We read in a clean speech signal and then create a noisy signal from this clean signal. We do that by adding washing machine noise from a pre-recorded wav file to the signal. This process allows us to understand how well the algorithm works by comparing a true clean signal with the final denoised signal.

Follow along in MATLAB: *Get example files and the full example*



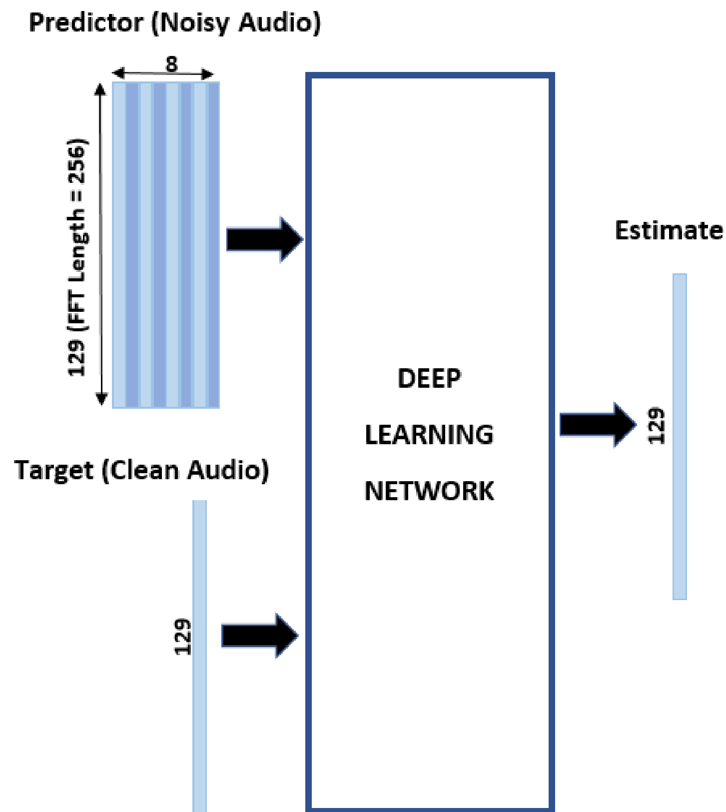
Plots of the original and noisy audio.

Preparing the Data

We set aside a collection of noisy speech examples for testing the trained algorithm.

In this example, we will transform the signal into a 2D signal using short-time Fourier transform (STFT). This technique involves breaking the signal into overlapping segments and computing the fast Fourier transform (FFT) of each segment. By taking the streaming signal in segments, we'll be able to see predictions while the signal is being acquired. At the input, we accumulate eight segments to offer the network some "memory" in time—to allow it to see not only the current signal segments but also a few past signals closely related to the present one. At the output, the network is asked to predict one segment at a time.

The process is illustrated below.



The predictor input consists of eight consecutive noisy STFT vectors, so each STFT output estimate is computed based on the current noisy STFT and the seven previous noisy STFT vectors for more context.

STFT Targets and Predictors

When transforming the audio to the frequency domain using the short-time Fourier transform (STFT), we need to apply some system parameters.

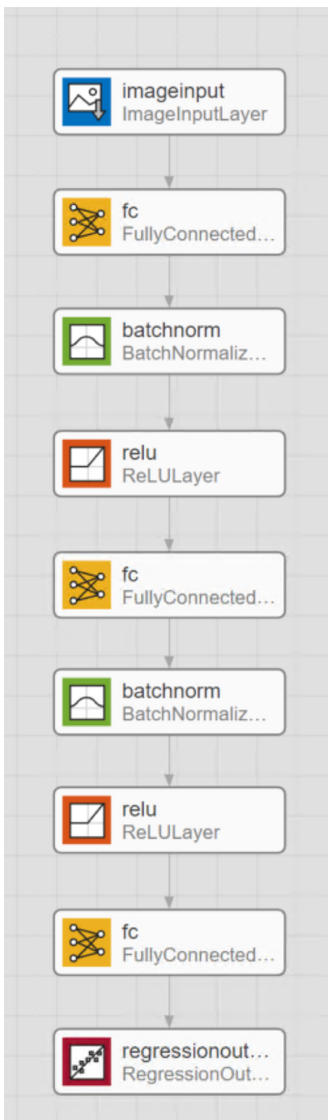
Window length. The longer the window length, the greater the resolution and complexity. A window length of 256 in this example is a reasonable compromise between high-frequency resolution and low computational complexity, given the available data.

Overlap. We set the overlap at 75% to give the network a more precise time resolution and allow more information to pass through the network in the same amount of time.

Input sample rate. This input data was recorded at a rate of 48 KHz.

Frequency sample rate. Down-sampling the input data to 8KHz will work well enough for human speech and will give us a more compact network.

Number of segments. In this example we chose eight segments, including the current noisy STFT and the seven previous segments that the network will use as context from which to learn. Because each segment overlaps by 75%, this is approximately three full segments of time.



Configure Network Architecture

We are going to follow the same basic workflow that was used in the first example. While the Speech Command Recognition example used a series of convolution layers, in this example we are using fully connected layers.

A fully connected layer is connected to all activations from the previous layer. The fully connected layer "flattens" 2D spatial features into a 1D vector.

The batch normalization layers normalize the means and standard deviations of the outputs. As parameters or previous layers change during the training stage, the current layers must readjust to new distributions, which takes time. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, we use batch normalization layers and nonlinearities such as ReLU layers between convolution layers.

The ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.

We finish with a regression layer, which will give us the half-mean-squared-error loss. The regression layer outputs predictions on continuous data, such as angles, distance, or, in our case, a denoised signal.

Training the Network

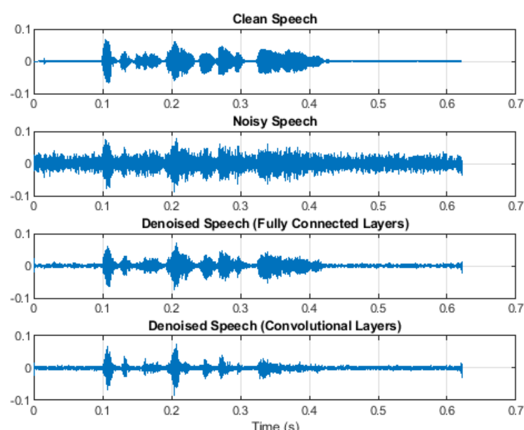
Next, we set our training options, as follows:

- Maximum epochs to 3 (the network will make three passes through the training data)
- Mini batch size to 128 (the network will look at 128 training signals at a time)
- `'Plots'` to `'training-progress'` (this makes the training plot visible)
- `'Shuffle'` to `'every-epoch'`
- `'LearnRateSchedule'` to `'piecewise'` (this will decrease the learning rate by a specified factor (0.9) every time an epoch has passed)

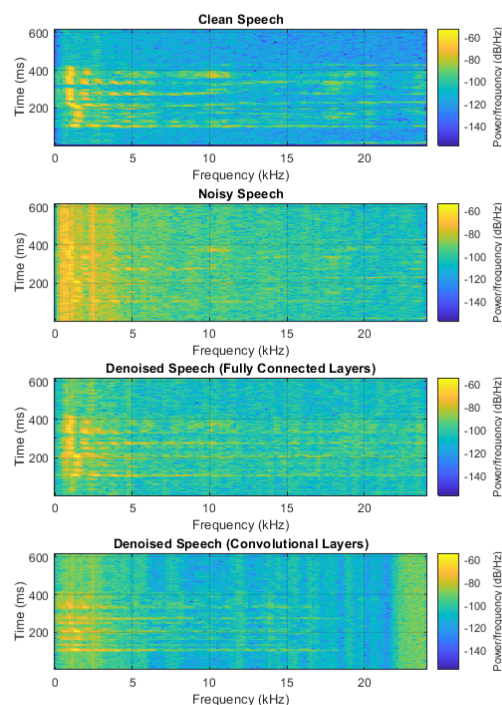
A training set of this size can take several minutes to run.

Checking Network Accuracy

After the network is trained, we can pass a noisy signal that was set aside for testing into the network and visualize the results. We can visualize the signals in a plot or as spectrograms.



The time-domain plots indicate that the background noise level has been considerably reduced.



The spectrograms tell us more about how the model is performing at different frequencies. For example, we can see it was particularly effective at removing noise at lower frequencies, where most of the speech spectrum is concentrated.

As this is audio data, you can listen to the speech to compare the quality subjectively to understand whether results are satisfactory.

The trained network is now ready for use in real time and we finish with an accurate, reusable model to denoise speech.

Run **`speechDenoisingRealtimeApp`** in MATLAB to simulate a streaming real-time version of the denoising network.

Summary

Deep learning for signal processing helps engineers create accurate, reusable models and reduces the expertise needed to extract and select features in signals and make changes to improve a model's performance.

In the speech recognition and signal denoising examples, we showed how to prepare and transform the audio data to use in a CNN and fully connected network, respectively. We visualized the performance of the models and looked at how to make changes without considerable signal processing expertise.

In the RUL example we used an LSTM and preprocessed the sensor data to give the network the best data possible and created a working model without manual feature extraction and selection.

Additional Resources

MATLAB for Deep Learning

Deep Learning for Signals and Sound

© 2021 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks.
Other product or brand names may be trademarks or registered trademarks of their respective holders.