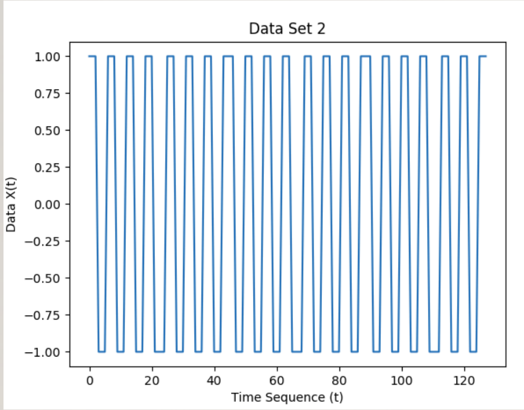
APPLICATION OF FILTERING AND DROPOUT IN NN USING KERAS

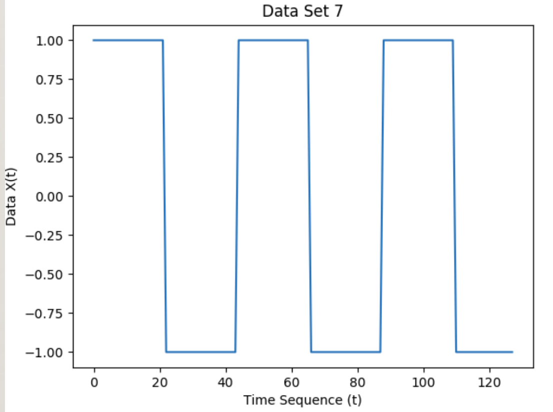
DR. MATTHEW SMITH

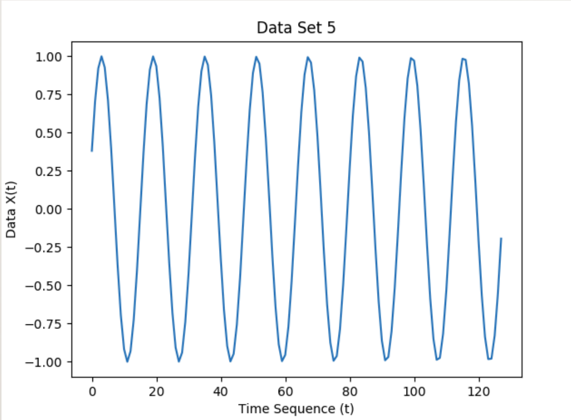
ADACS, SWINBURNE UNIVERSITY OF TECHNOLOGY

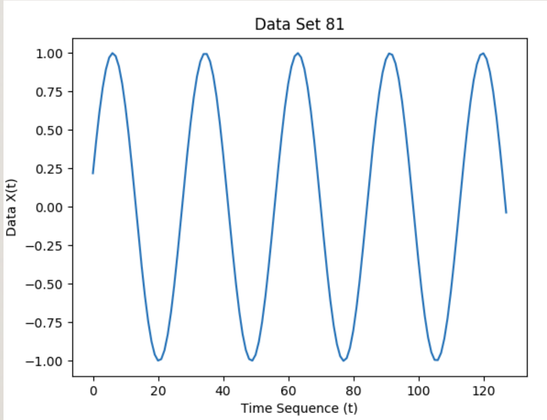
PROBLEM DEFINITION

- In the previous section, the classification problem we encountered was quite simple – two distinct different types of signals, resulting in a relatively simple classification task:









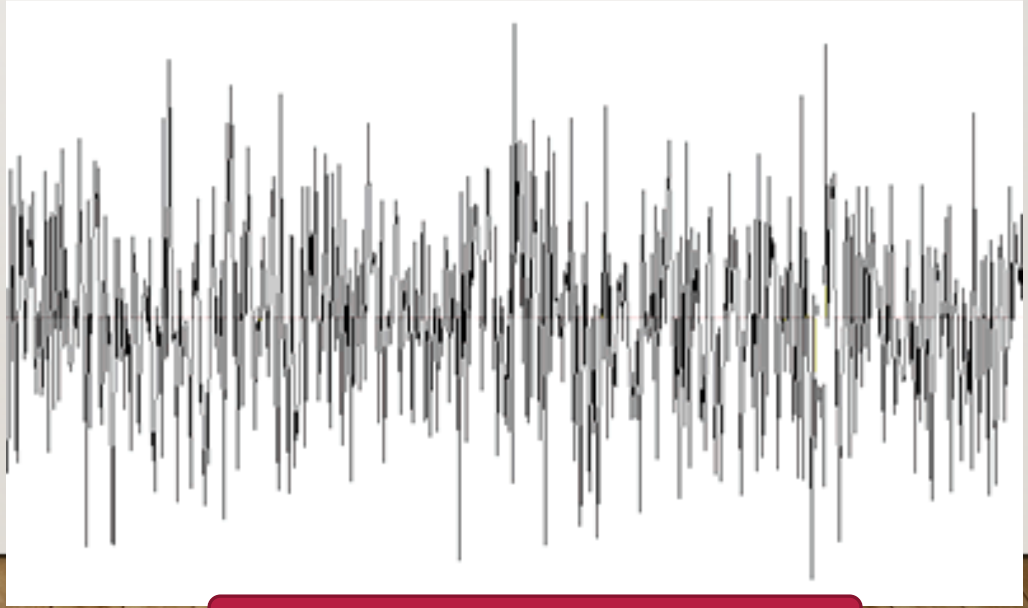
$$x(t) = square(0.1 + 0.3R_F)$$

$$x(t) = \sin(0.1 + 0.3R_F)$$

PROBLEM DEFINITION

- In reality, however, many signals which require classification are not so clean, and include **noise**.
- This noise can be in the form of grain (in an image), or one-dimensional white noise in an array of data (like that encountered in audio analysis).

Image Noise (courtesy: wiki)



Audio Noise

PROBLEM DEFINITION

- Humans, being experts in pattern recognition, are often able to understand the nature of the noise, and then see past it when performing classification tasks.
- The problem is that the Neural Network – which will search for features in the data – will not be able to understand the nature of random noise (natively).
- The purpose of this session is to employ two strategies to aid us in overcoming the problem:
  - Use filtering and smoothing to help us remove noise from the signal using scipy filters, and
  - Use dropout to prevent the NN overfitting to noise.

FILTERING IN PYTHON USING SCIPY

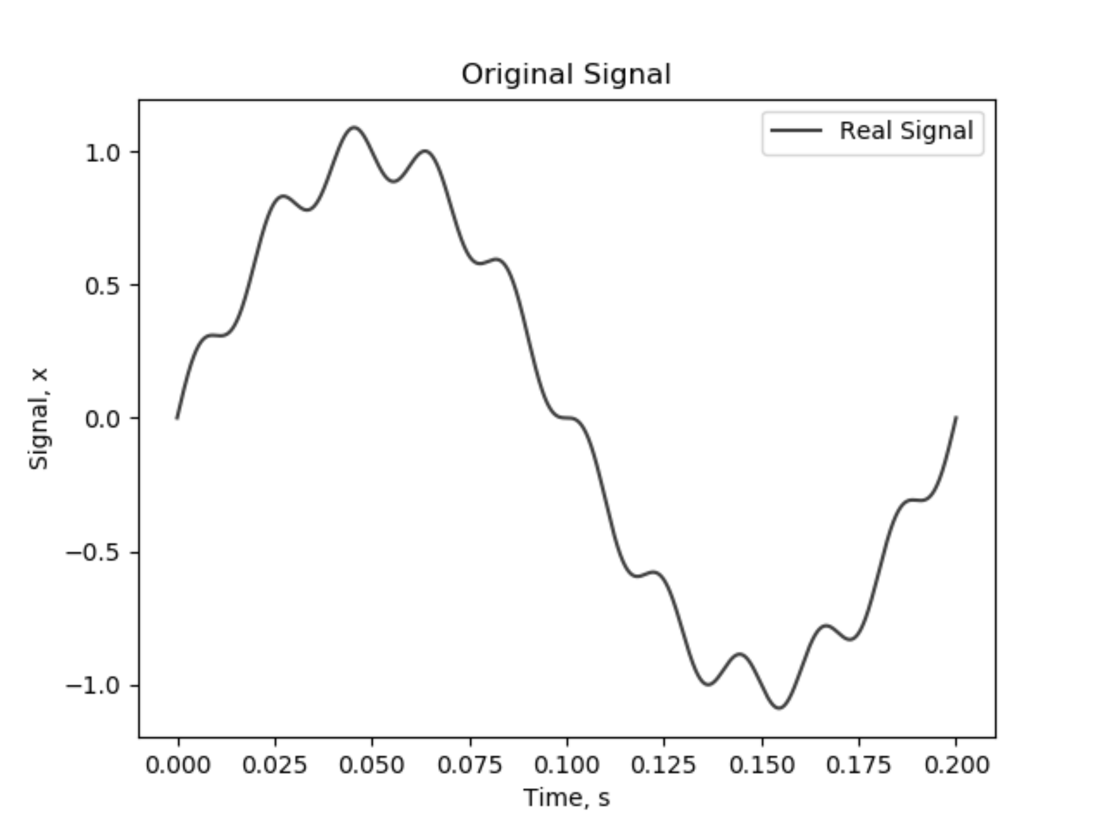
- Here is the python script we shall use to test out our low pass filter of choice – the Butterworth filter.

- The top sections are typical imports – don't forget to use tkagg if you want to use X11 forwarding with matplotlib on ozstar (otherwise you won't see anything).

```
# Filtering Demonstration
# Dr. Matthew Smith, ADACS @ Swinburne University of Technology
# Generate a fake signal, add noise and perform filtering.
# Questions? Email: msmith@astro.swin.edu.au

import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

# Create a figure
plt.figure

# Generate time
t = np.linspace(0,0.2,2001)
# Compute genuine signal
x = np.sin(2.0*np.pi*5.0*t) + 0.1*np.sin(2.0*np.pi*50.0*t)
plt.plot(t,x,'k',alpha=0.75)
# Add some noise
x = x+ np.random.randn(len(t))*0.05

# Peform filtering using Butterworth
b,a = signal.butter(3,0.005)
zi = signal.lfilter_zi(b,a)
z, _ = signal.lfilter(b,a,x, zi=zi*x[0])
filt_x = signal.filtfilt(b,a,x)

# Plot
plt.plot(t,x,'b',alpha=0.75)
plt.plot(t,filt_x,'r')
plt.title('Comparison of filtered result and original signal')
plt.xlabel('Time, s')
plt.ylabel('Signal, x')
plt.legend(('Real Signal','Signal+Noise','Filtered Result'))
plt.show()
```

FILTERING IN PYTHON USING SCIPY

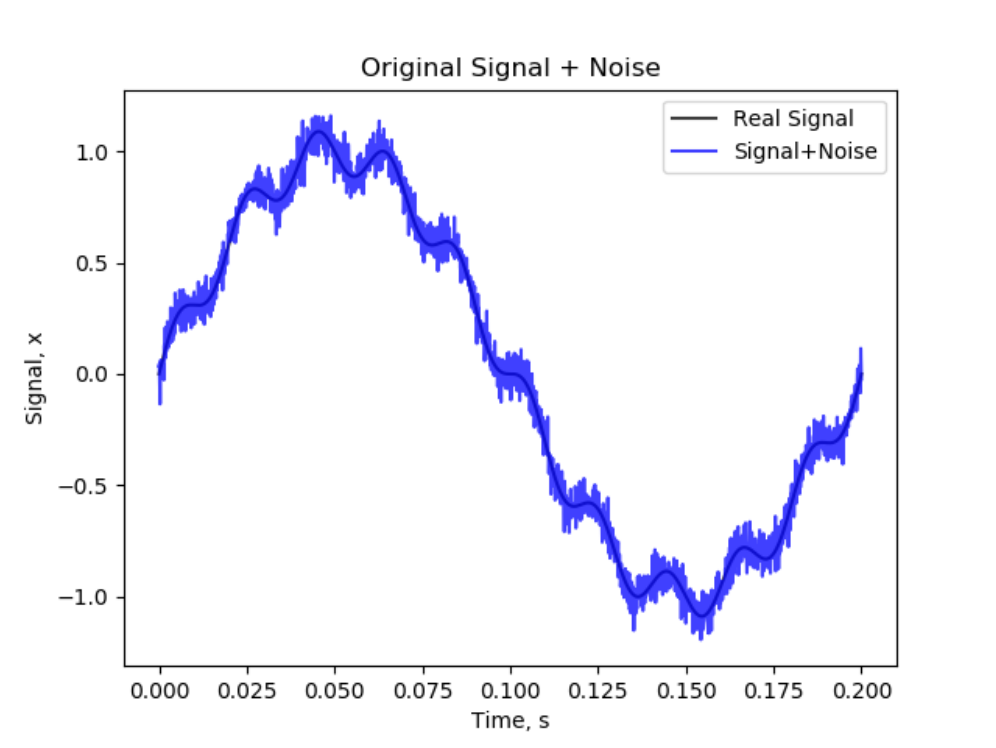
- The first real port of call is to generate the signal data – our pure signal looks like this:




```
# Filtering Demonstration
# Dr. Matthew Smith, ADACS @ Swinburne University of Technology
# Generate a fake signal, add noise and perform filtering.
# Questions? Email: msmith@astro.swin.edu.au

import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

# Create a figure
plt.figure

# Generate time
t = np.linspace(0,0.2,2001)
# Compute genuine signal
x = np.sin(2.0*np.pi*5.0*t) + 0.1*np.sin(2.0*np.pi*50.0*t)
plt.plot(t,x,'k',alpha=0.75)
# Add some noise
x = x+ np.random.randn(len(t))*0.05

# Peform filtering using Butterworth
b,a = signal.butter(3,0.005)
zi = signal.lfilter_zi(b,a)
z, _ = signal.lfilter(b,a,x, zi=zi*x[0])
filt_x = signal.filtfilt(b,a,x)

# Plot
plt.plot(t,x,'b',alpha=0.75)
plt.plot(t,filt_x,'r')
plt.title('Comparison of filtered result and original signal')
plt.xlabel('Time, s')
plt.ylabel('Signal, x')
plt.legend(('Real Signal','Signal+Noise','Filtered Result'))
plt.show()
```

FILTERING IN PYTHON USING SCIPY

- After which we add noise – this noise is normally distributed noise with a mean of 0 and variance of 0.025 (that's 0.5^2)



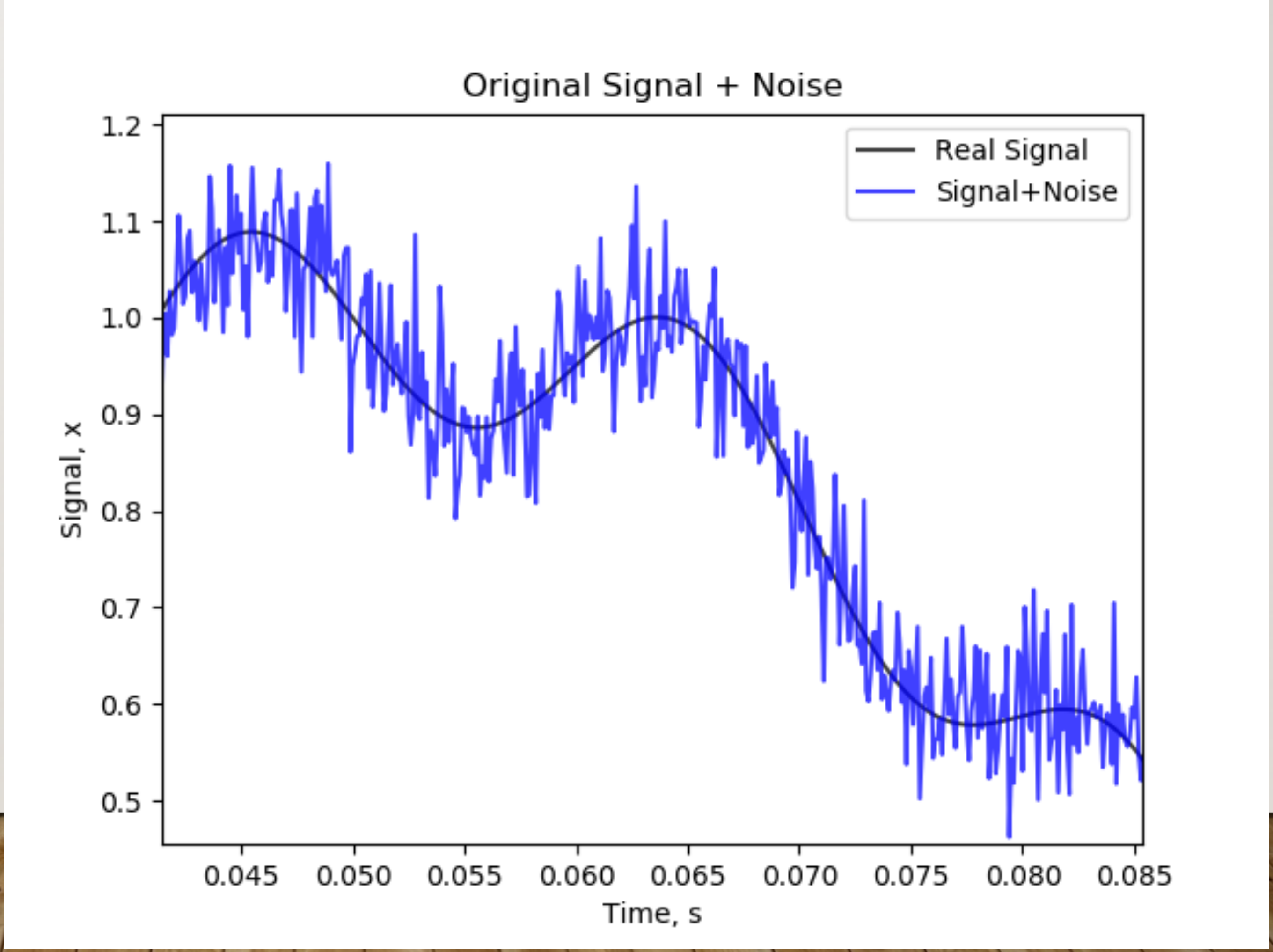

```python
# Filtering Demonstration
# Dr. Matthew Smith, ADACS @ Swinburne University of Technology
# Generate a fake signal, add noise and perform filtering.
# Questions? Email: msmith@astro.swin.edu.au

import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

# Create a figure
plt.figure

# Generate time
t = np.linspace(0,0.2,2001)
# Compute genuine signal
x = np.sin(2.0*np.pi*5.0*t) + 0.1*np.sin(2.0*np.pi*50.0*t)
plt.plot(t,x,'k',alpha=0.75)
# Add some noise
x = x+ np.random.randn(len(t))*0.05

# Peform filtering using Butterworth
b,a = signal.butter(3,0.005)
zi = signal.lfilter_zi(b,a)
z, _ = signal.lfilter(b,a,x, zi=zi*x[0])
filt_x = signal.filtfilt(b,a,x)

# Plot
plt.plot(t,x,'b',alpha=0.75)
plt.plot(t,filt_x,'r')
plt.title('Comparison of filtered result and original signal')
plt.xlabel('Time, s')
plt.ylabel('Signal, x')
plt.legend(('Real Signal','Signal+Noise','Filtered Result'))
plt.show()
```

FILTERING IN PYTHON USING SCIPY

- Let's have a closer look:

- Our mission will be to filter this result to remove the noise, which is present in a higher frequency than the two lower frequencies present in the real signal.

- Let's create our filter.



FILTERING IN PYTHON USING SCIPY

- To get things going, we use the signal.butter function to create parameters a and b.
- These are the filter coefficients – basically one dimensional arrays of length (ORDER+1).
- These are created when we call the function:

b,a = signal.butter(ORDER,WN)

```python
# Filtering Demonstration
# Dr. Matthew Smith, ADACS @ Swinburne University of Technology
# Generate a fake signal, add noise and perform filtering.
# Questions? Email: msmith@astro.swin.edu.au

import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

# Create a figure
plt.figure

# Generate time
t = np.linspace(0,0.2,2001)
# Compute genuine signal
x = np.sin(2.0*np.pi*5.0*t) + 0.1*np.sin(2.0*np.pi*50.0*t)
plt.plot(t,x,'k',alpha=0.75)
# Add some noise
x = x+ np.random.randn(len(t))*0.05

# Peform filtering using Butterworth
b,a = signal.butter(3,0.005)
zi = signal.lfilter_zi(b,a)
z, _ = signal.lfilter(b,a,x, zi=zi*x[0])
filt_x = signal.filtfilt(b,a,x)

# Plot
plt.plot(t,x,'b',alpha=0.75)
plt.plot(t,filt_x,'r')
plt.title('Comparison of filtered result and original signal')
plt.xlabel('Time, s')
plt.ylabel('Signal, x')
plt.legend(('Real Signal','Signal+Noise','Filtered Result'))
plt.show()
```

FILTERING IN PYTHON USING SCIPY

- There is quite a lot of control systems theory associated with the use of low pass filters, which will not be covered here. Hence we will only briefly cover the details:
- In this code, we employ a 3rd order filter (ORDER=3). This roughly corresponds to the "range" of data examined when performing filtering on any single value in our array of data.
- The value of Wn – in this case – is a normalised cutoff frequency.

```
# Filtering Demonstration
# Dr. Matthew Smith, ADACS @ Swinburne University of Technology
# Generate a fake signal, add noise and perform filtering.
# Questions? Email: msmith@astro.swin.edu.au

import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

# Create a figure
plt.figure

# Generate time
t = np.linspace(0,0.2,2001)
# Compute genuine signal
x = np.sin(2.0*np.pi*5.0*t) + 0.1*np.sin(2.0*np.pi*50.0*t)
plt.plot(t,x,'k',alpha=0.75)
# Add some noise
x = x+ np.random.randn(len(t))*0.05

# Peform filtering using Butterworth
b,a = signal.butter(3,0.005)
zi = signal.lfilter_zi(b,a)
z, _ = signal.lfilter(b,a,x, zi=zi*x[0])
filt_x = signal.filtfilt(b,a,x)

# Plot
plt.plot(t,x,'b',alpha=0.75)
plt.plot(t,filt_x,'r')
plt.title('Comparison of filtered result and original signal')
plt.xlabel('Time, s')
plt.ylabel('Signal, x')
plt.legend(('Real Signal','Signal+Noise','Filtered Result'))
plt.show()
```

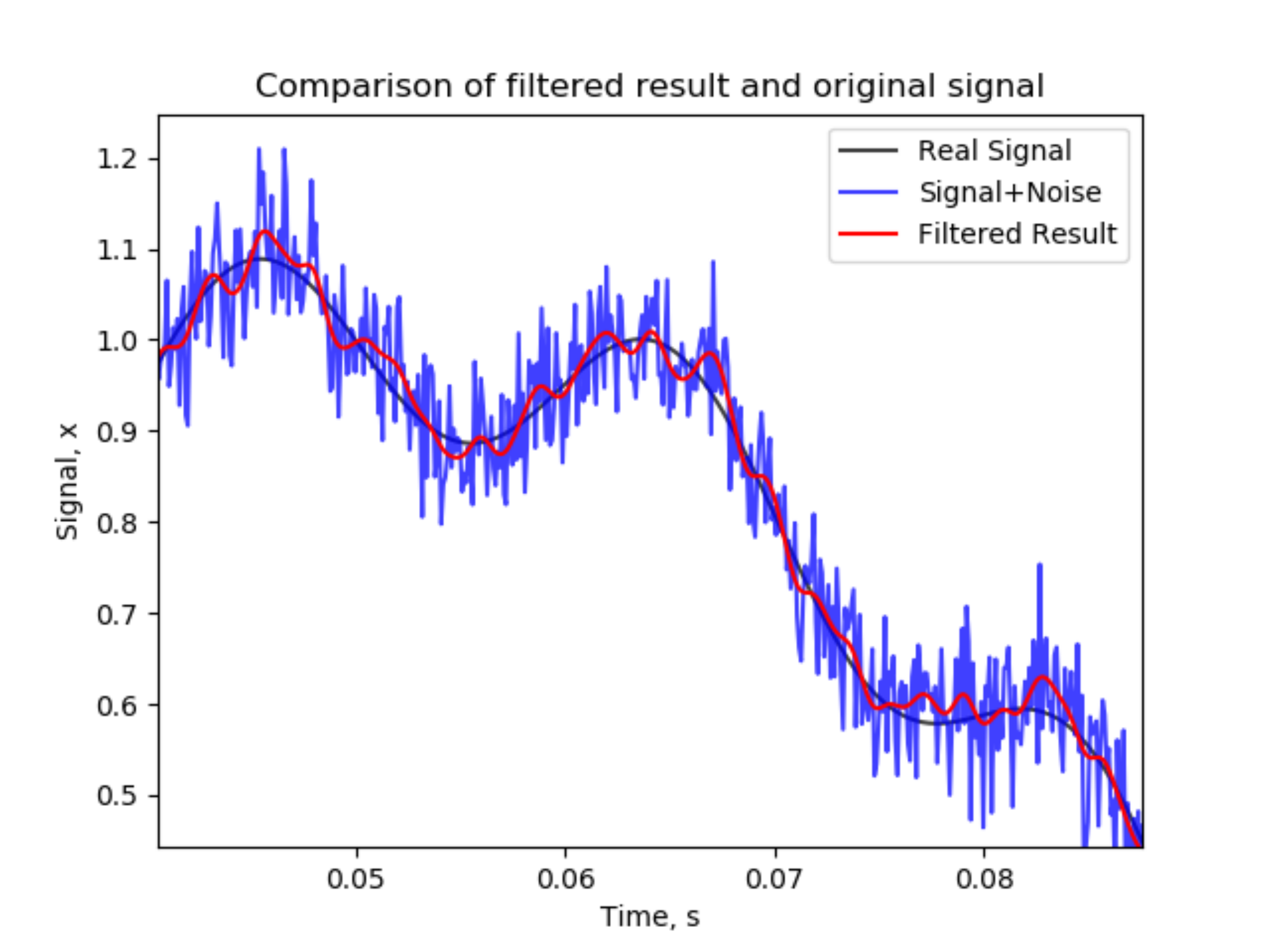
FILTERING IN PYTHON USING SCIPY

- The final part of this demonstration is simply plotting the result using matplotlib.
- Let's have a quick look at the influence of the value of WN (here = 0.005) on the final result…

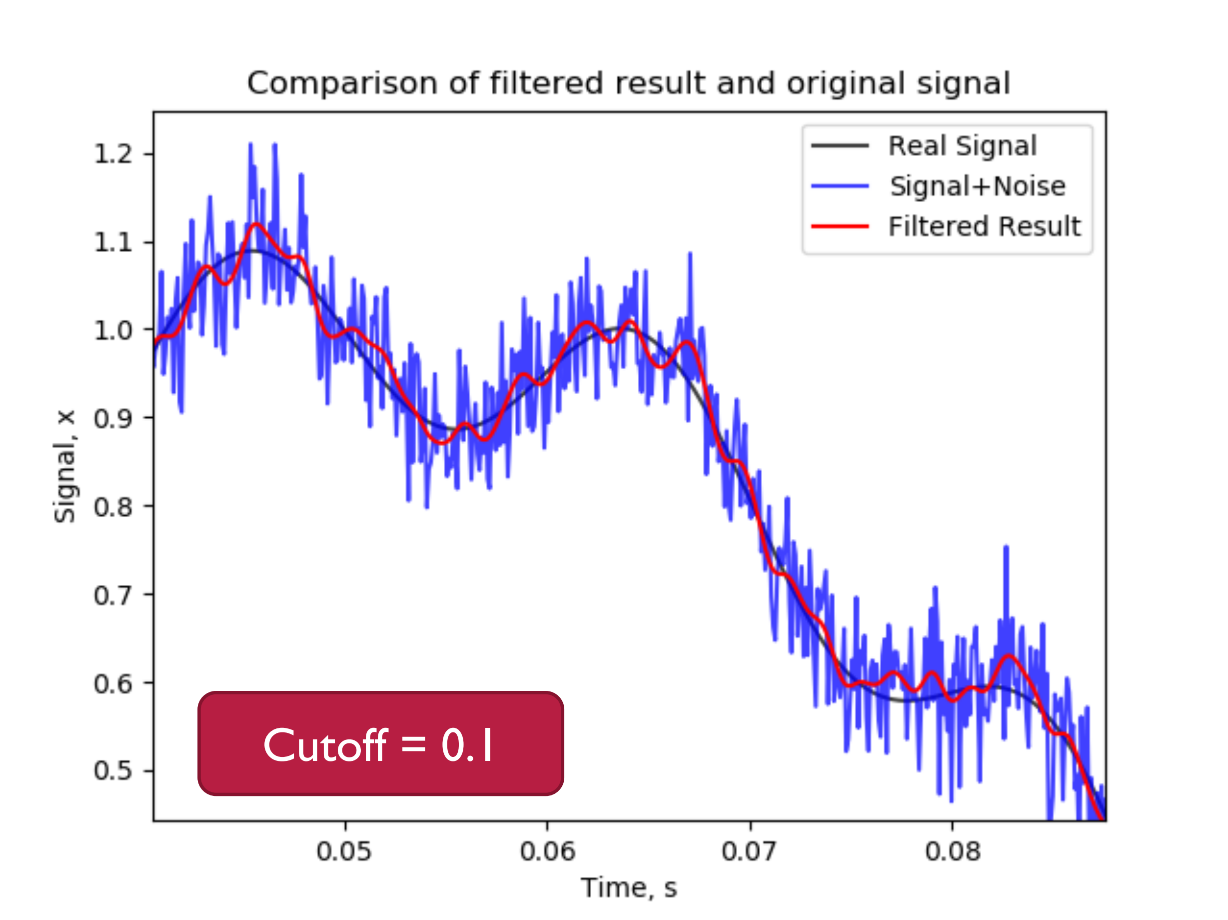
```python
# Filtering Demonstration
# Dr. Matthew Smith, ADACS @ Swinburne University of Technology
# Generate a fake signal, add noise and perform filtering.
# Questions? Email: msmith@astro.swin.edu.au

import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

# Create a figure
plt.figure

# Generate time
t = np.linspace(0,0.2,2001)
# Compute genuine signal
x = np.sin(2.0*np.pi*5.0*t) + 0.1*np.sin(2.0*np.pi*50.0*t)
plt.plot(t,x,'k',alpha=0.75)
# Add some noise
x = x+ np.random.randn(len(t))*0.05

# Peform filtering using Butterworth
b,a = signal.butter(3,0.005)
zi = signal.lfilter_zi(b,a)
z, _ = signal.lfilter(b,a,x, zi=zi*x[0])
filt_x = signal.filtfilt(b,a,x)

# Plot
plt.plot(t,x,'b',alpha=0.75)
plt.plot(t,filt_x,'r')
plt.title('Comparison of filtered result and original signal')
plt.xlabel('Time, s')
plt.ylabel('Signal, x')
plt.legend(('Real Signal','Signal+Noise','Filtered Result'))
plt.show()
```

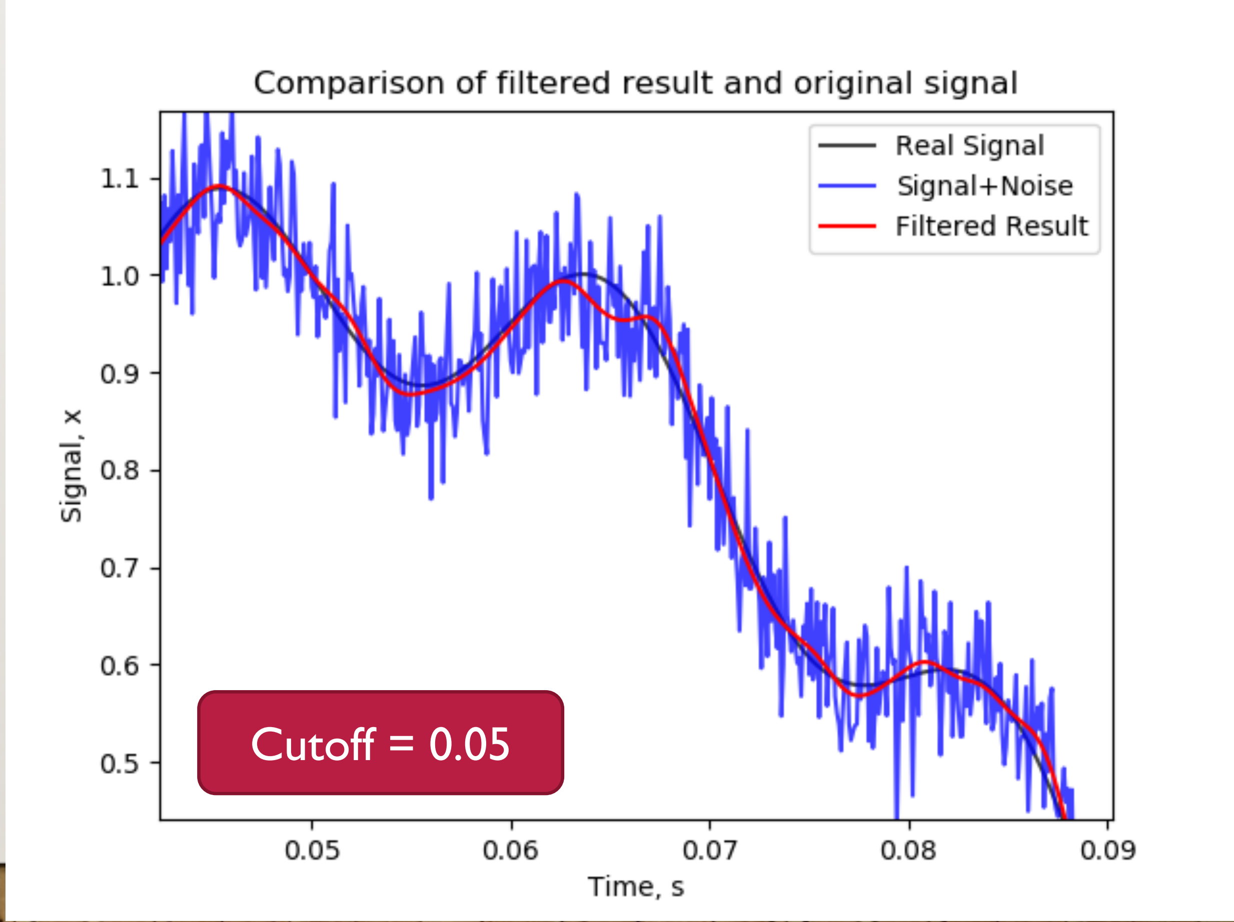
FILTERING IN PYTHON USING SCIPY

- Here, a 3rd order filter is used with a normalized cutoff value of 0.1
- We can see that the we have not successfully removed the noise – we need to lower the cutoff frequency to remove some of the higher frequency noise present.
- Let's lower the cutoff frequency and see what happens…



FILTERING IN PYTHON USING SCIPY

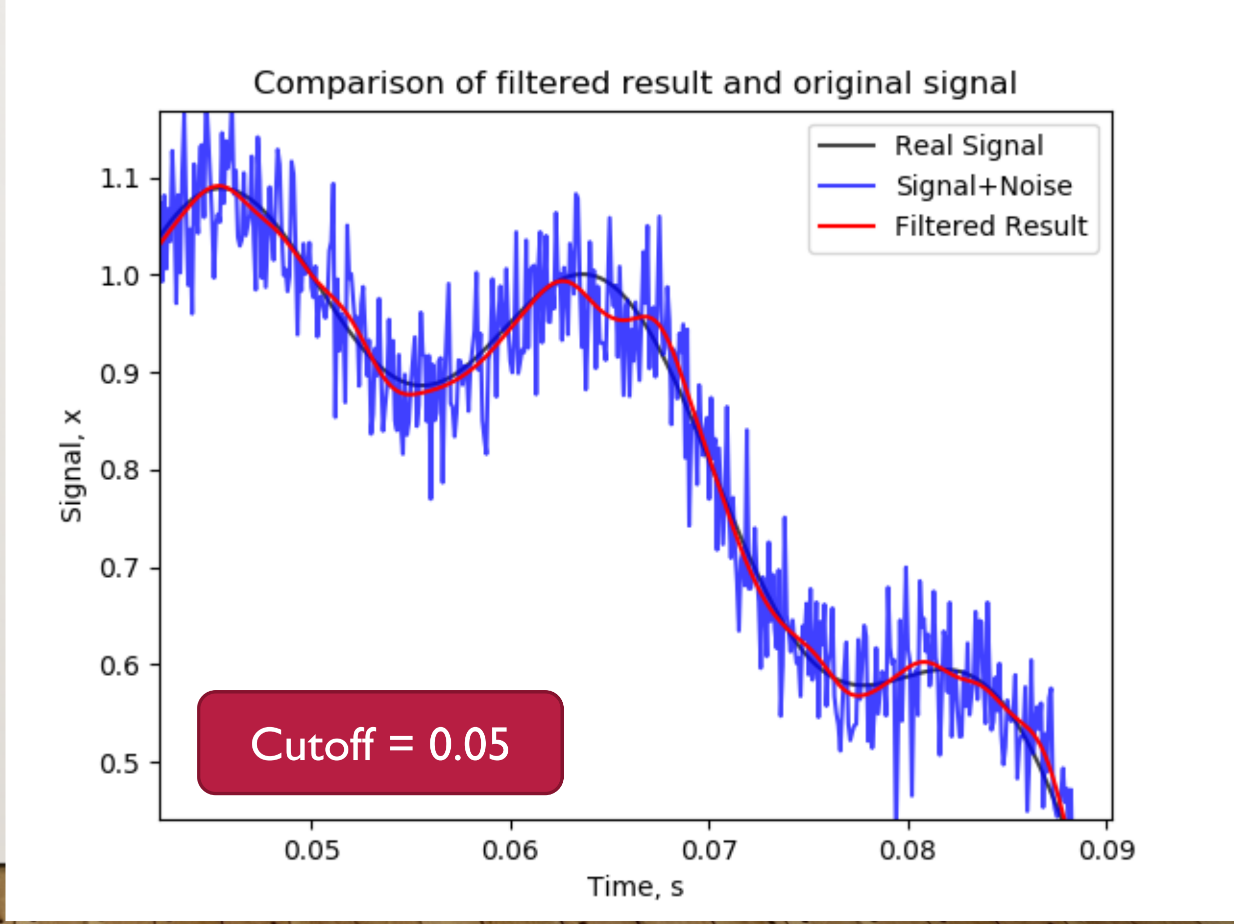




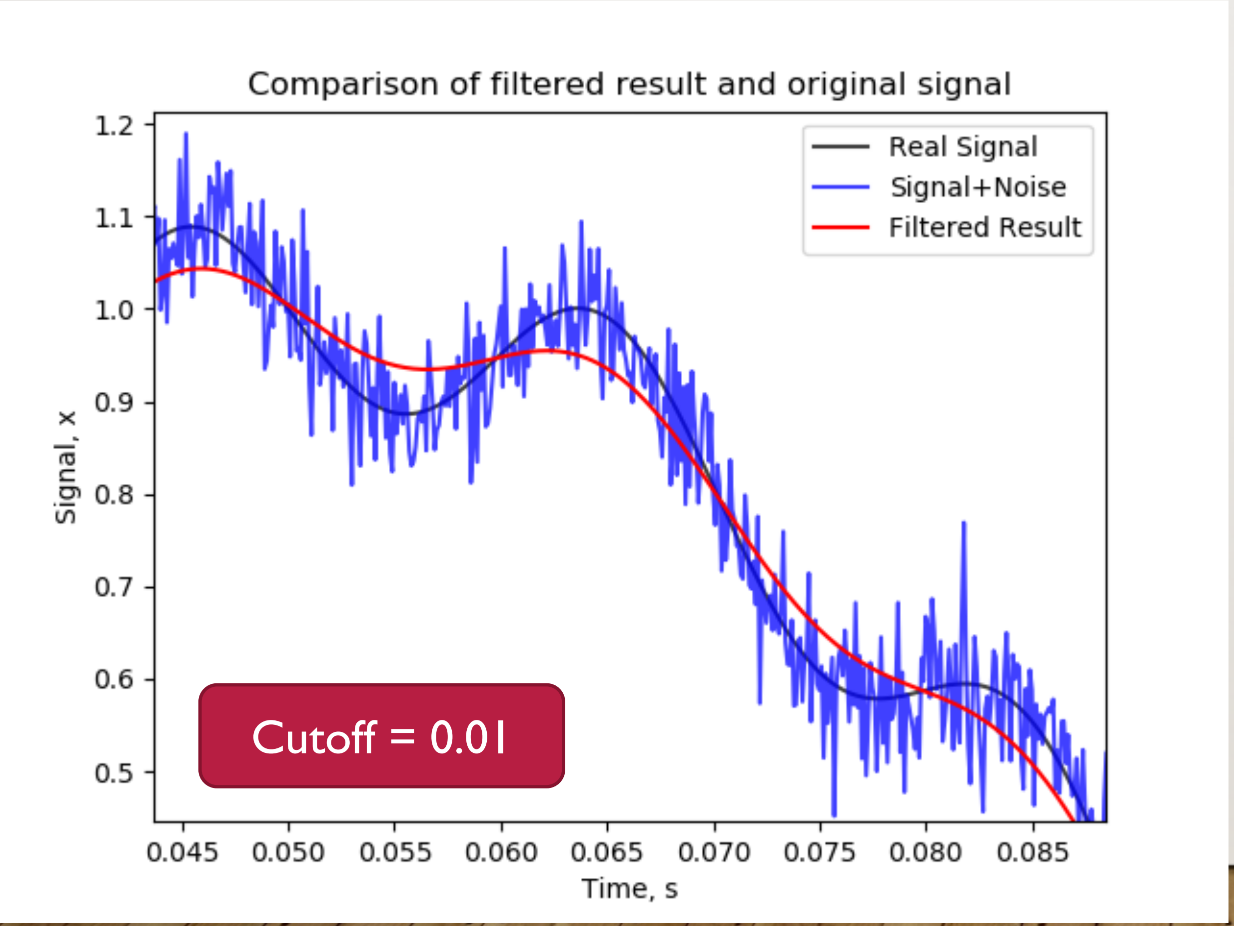
FILTERING IN PYTHON USING SCIPY

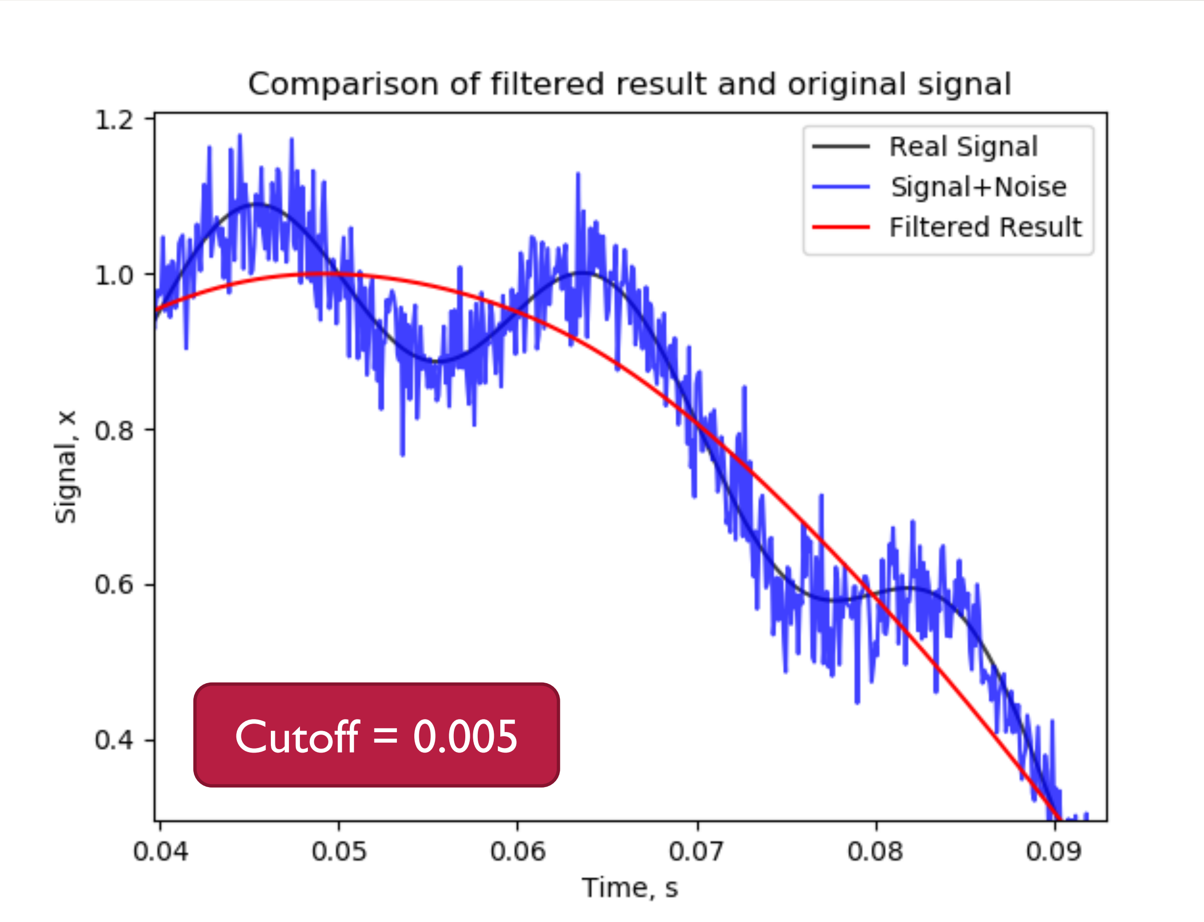
- So this result represents something closer to what we are looking to achieve - we have almost recovered the correct result.

- What happens if we continue to decrease the value of the cutoff frequency?



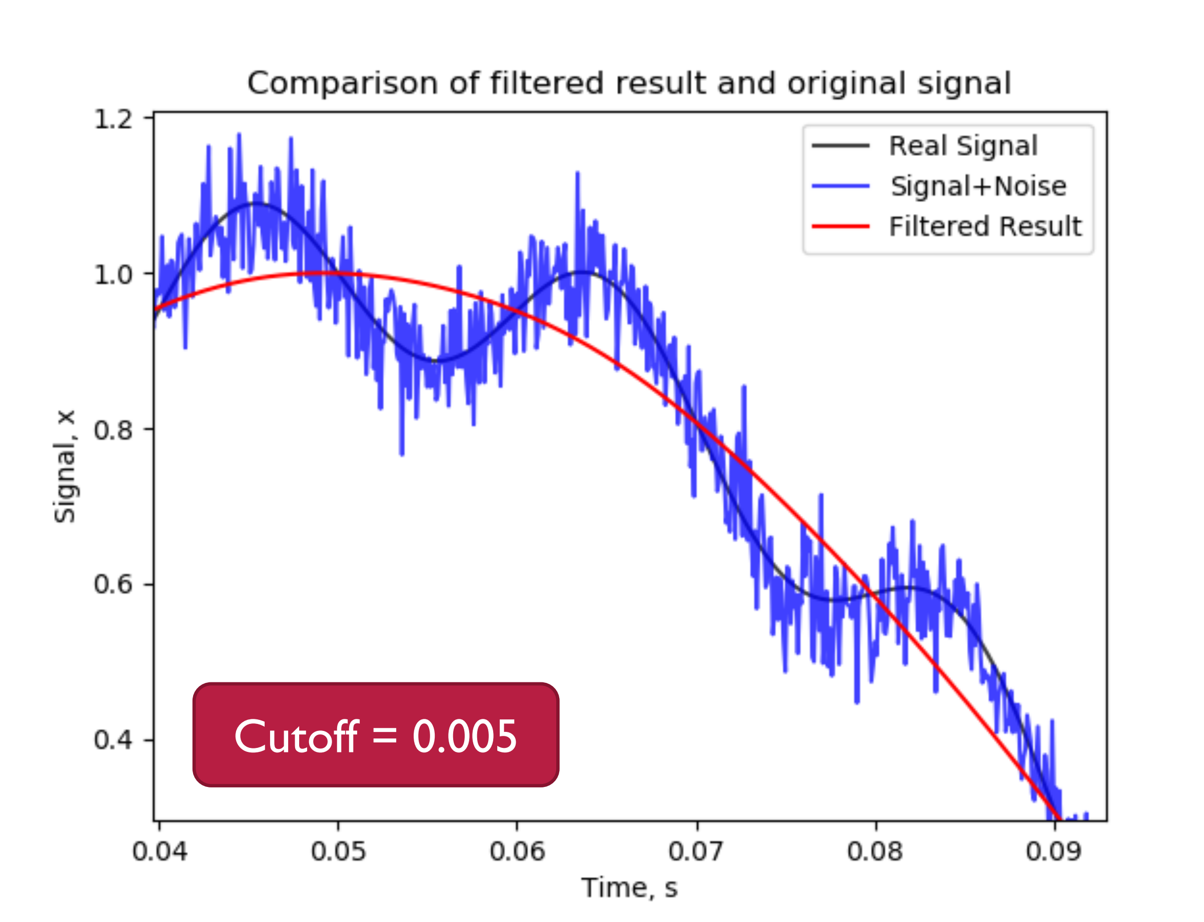
FILTERING IN PYTHON USING SCIPY





FILTERING IN PYTHON USING SCIPY

- You can see that, with a normalized cutoff frequency of 0.005, we have completely attenuated the higher frequency component of the real signal – which is bad news.
- So don't go too far!
- We can examine frequency response graphs for more insight into which frequencies are kept – google this in your own time.



FILTERING IN PYTHON USING SCIPY

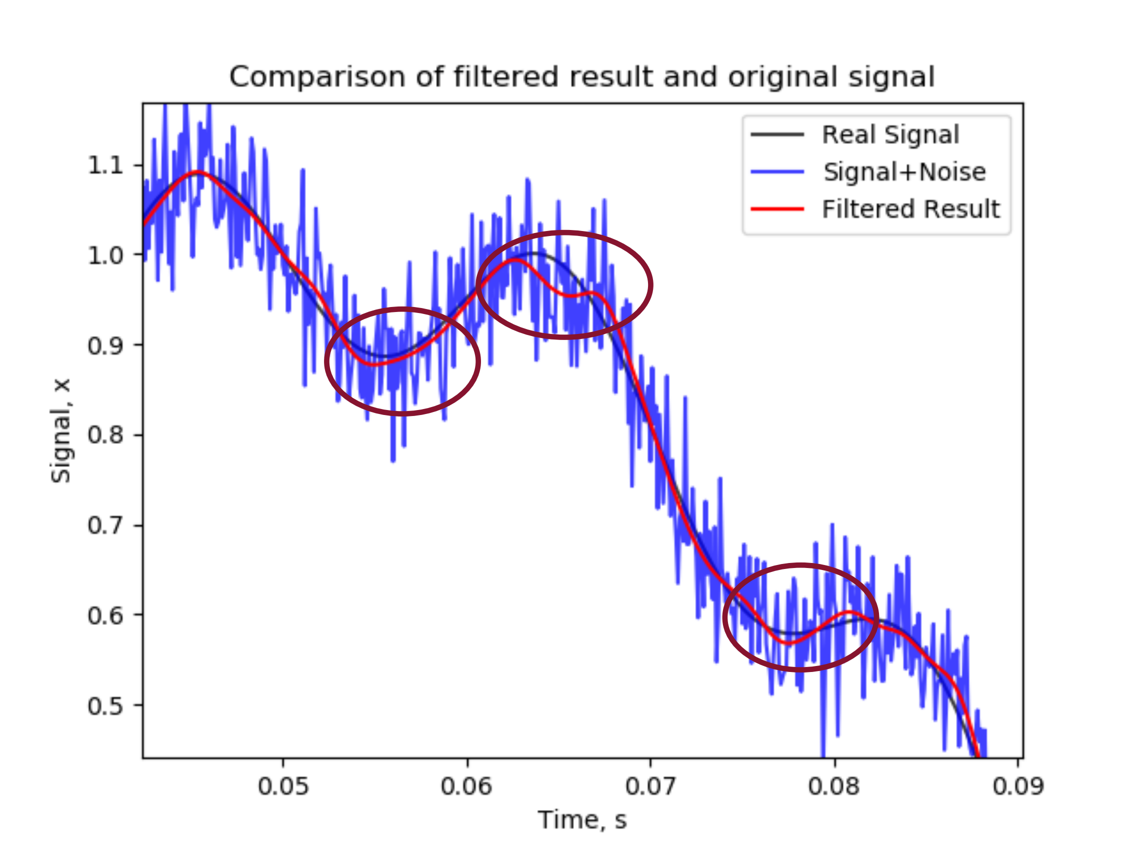
- On another note – if the contribution of the noise is larger, we will need a lower normalized cutoff frequency in order to correctly capture the signal we want.

- Using this approach, it is possible to see surprisingly small artefacts hidden under quite a lot of noise.

DROPOUT IN NEURAL NETWORKS

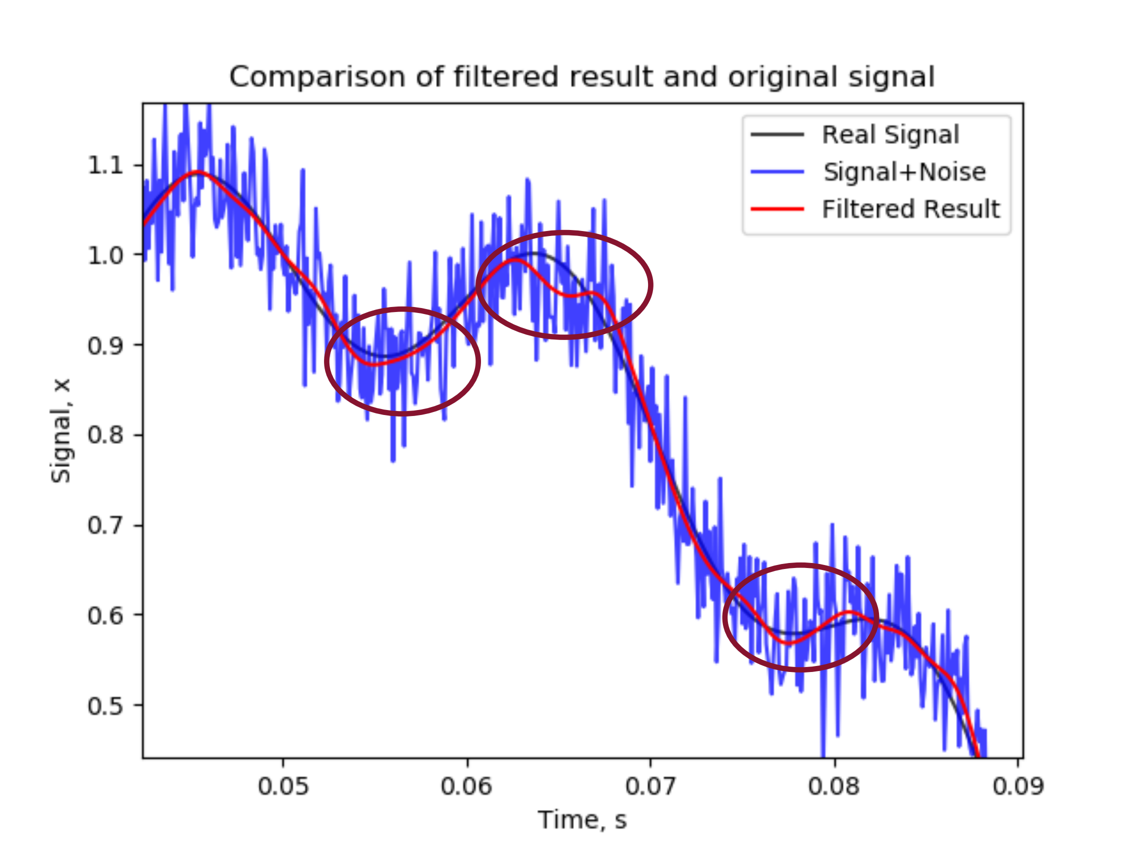
OVERFITTING

- Even after we have implemented a filter to remove noise from our sequence, we will still have something resembling noise in the result.
- Consider this signal here – we have managed to capture the basic shape of the real signal, but small perturbations remain.



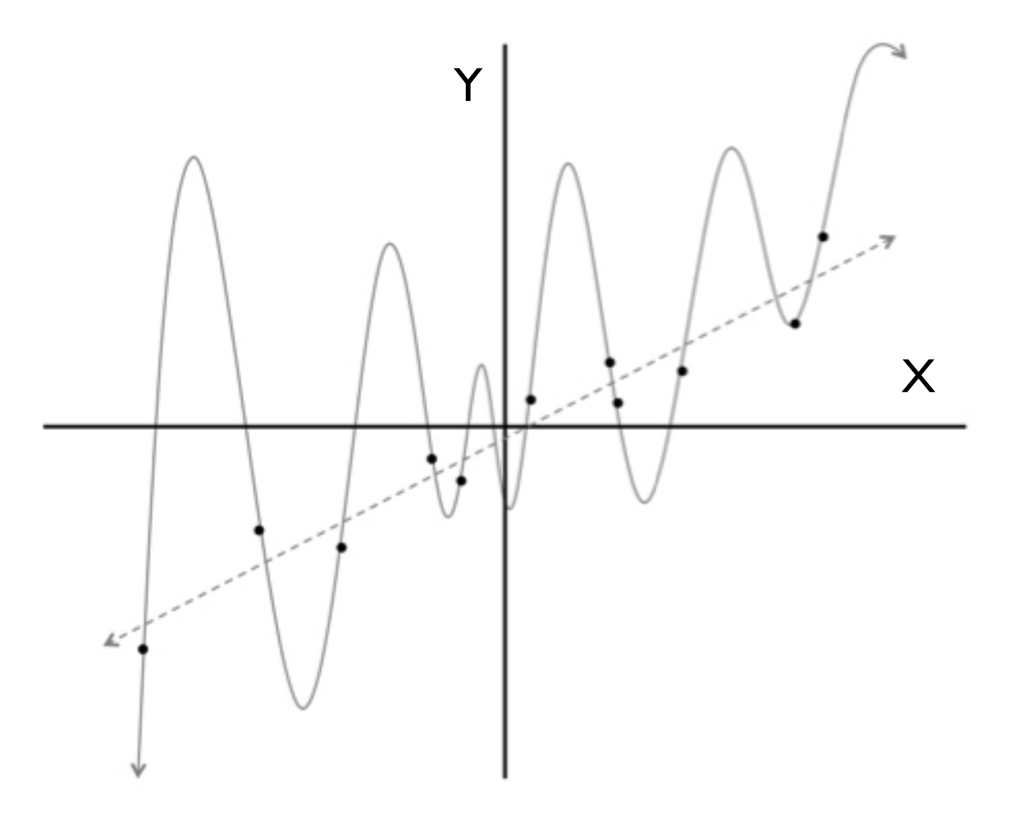
OVERFITTING

- The Neural Network which takes this signal for training has no idea that these perturbations aren't key elements of the real signal.
- Hence, it may attempt to focus on these features as part of its classification training – the process of attempting to include random elements into model construction in NN's is known as ***overfitting***.



OVERFITTING

- This is conceptually demonstrated with a regression problem (as shown on the right).
- Given the points shown, can we create a polynomial expression which demonstrates the relationship between X and Y?
- Shown is: a linear relationship ($y = mx+c$) passing in a least-squares manner through the points, and a 12^{th} order polynomial passing through all the points.



PREVENTING OVERFITTING

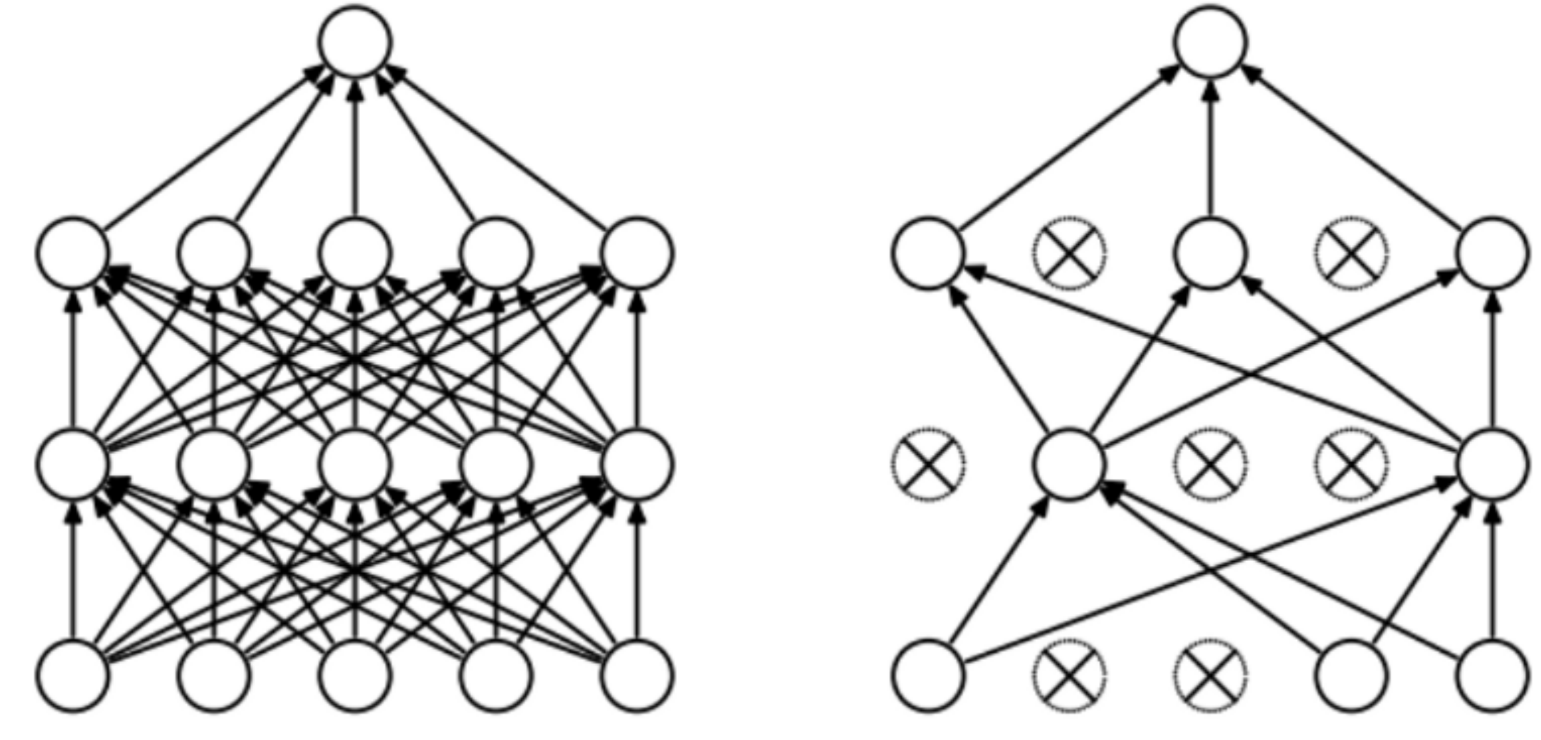
- There are two strategies we might employ to prevent overfitting:

  - Reduce the number of neural connections – use fewer hidden layers, or fewer neurons in each layer. The more connections present, the more complex a model the NN may construct – which has a tendency to overfit on features such as outlying data points or random fluctuations.

  - Use Dropout - this is the process of ignoring randomly selected neurons in your NN in the training process.

PREVENTING OVERFITTING

- By randomly removing neurons from the network, we are forcing the network to search for more persistent relationships in our network and input data.
- The consequence of this – the network takes longer to train, as it is forced to search for deeper relationships.


(a) Standard Neural Net (b) After applying dropout.

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

DROPOUT

- Wiki Definition:

> *According to [Wikipedia](#)—*
> *The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.*

- Consider our previously implemented NN for our sequence classification.

The number of neurons in the input layer is the same length as our sequence.

```
# Create our Keras model - an RNN (in Keras this is a Sequence)
model = Sequential()

# Configure our RNN by adding neural layers with activation functions
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

DROPOUT

- We can add dropout on the input by modifying our code slightly:

```
model.add(Dropout(0.2, input_shape=(N_sequence,)))
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

Simple Modification to include Dropout

```
# Configure our RNN by adding neural layers with activation functions
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

Previous Code

DROPOUT

- The first argument is the fraction of neurons to be dropped from the input layer during training. In this case, 20% of the input neurons (N_sequence of them) will be dropped randomly during training.
- Note: The input shape (or dimension) is required to be included in the first input into the Keras model – we do not need to reiterate this dimension in the 2nd argument, hence we can also write:

```
model.add(Dropout(0.2, input_shape=(N_sequence,)))
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

OK

```
model.add(Dropout(0.2, input_shape=(N_sequence,)))
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

Also OK

DROPOUT

- To add dropout in the hidden layers, we can simply place our dropout command between the layers we wish dropout to occur in.
- We cannot use dropout on the output neurons – this would be counterproductive, especially in a binary classification problem such as the ones we are looking at.

```
model.add(Dense(64, activation='relu',input_dim=N_sequence))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Again, in this case, no dimension is required as it is implied from the number of neurons specified in the previous and next layer.

DROPOUT

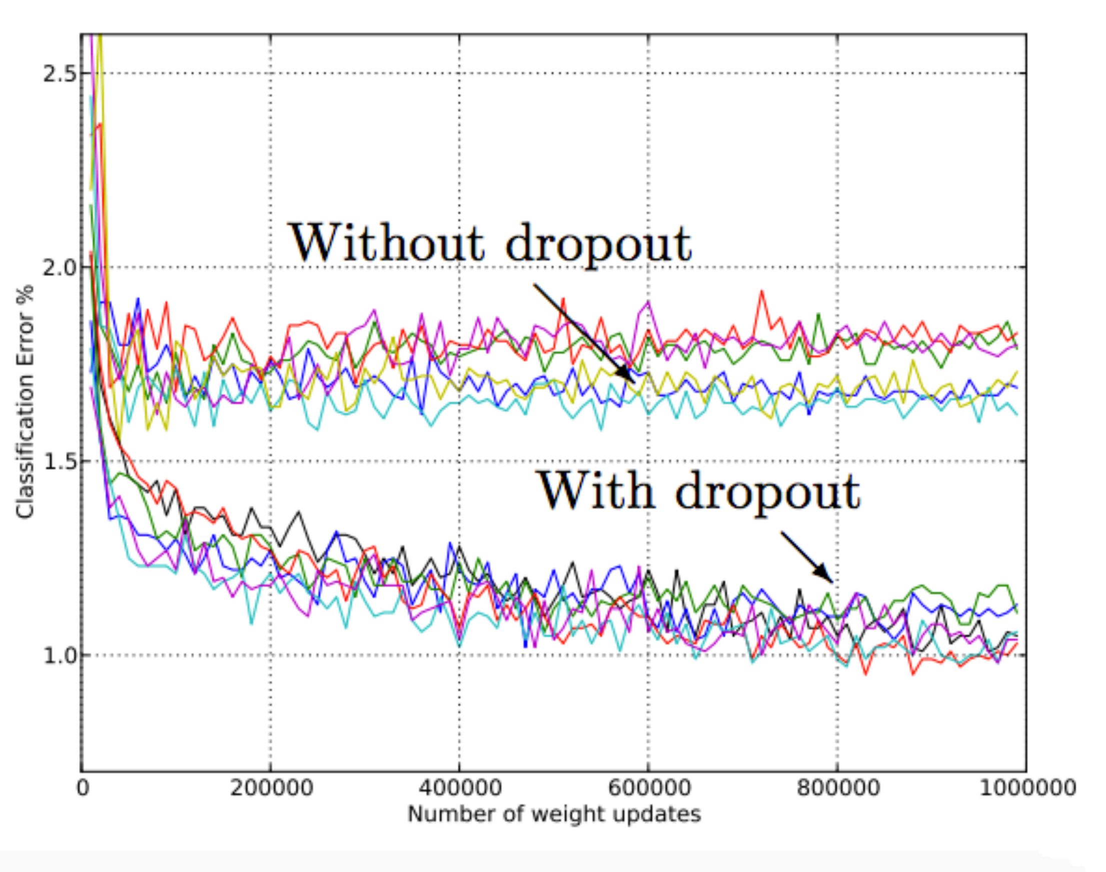
- To add dropout in the hidden layers, we can simply place our dropout command between the layers we wish dropout to occur in.
- We cannot use dropout on the output neurons – this would be counterproductive, especially in a binary classification problem such as the ones we are looking at.

```
model.add(Dense(64, activation='relu',input_dim=N_sequence))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Again, in this case, no dimension is required as it is implied from the number of neurons specified in the previous and next layer.

DROPOUT EFFECTIVENESS

- Dropout has been demonstrated to improve classification error, especially when said errors are due to overfitting.
- This is not free:
  - We often need more neurons in the hidden layers of the network;
  - The speed of training is reduced – more epochs are required before convergence occurs.
- It is also very difficult to predict what fraction of dropout to use, and where.



Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

DROPOUT EFFECTIVENESS

- Some of the published improvements due to the use of dropout seem pretty minor…

Method	Phone Error Rate%
NN (6 layers) (Mohamed et al., 2010)	23.4
Dropout NN (6 layers)	21.8
DBN-pretrained NN (4 layers)	22.7
DBN-pretrained NN (6 layers) (Mohamed et al., 2010)	22.4
DBN-pretrained NN (8 layers) (Mohamed et al., 2010)	20.7
mcRBM-DBN-pretrained NN (5 layers) (Dahl et al., 2010)	20.5
DBN-pretrained NN (4 layers) + dropout	**19.7**
DBN-pretrained NN (8 layers) + dropout	**19.7**

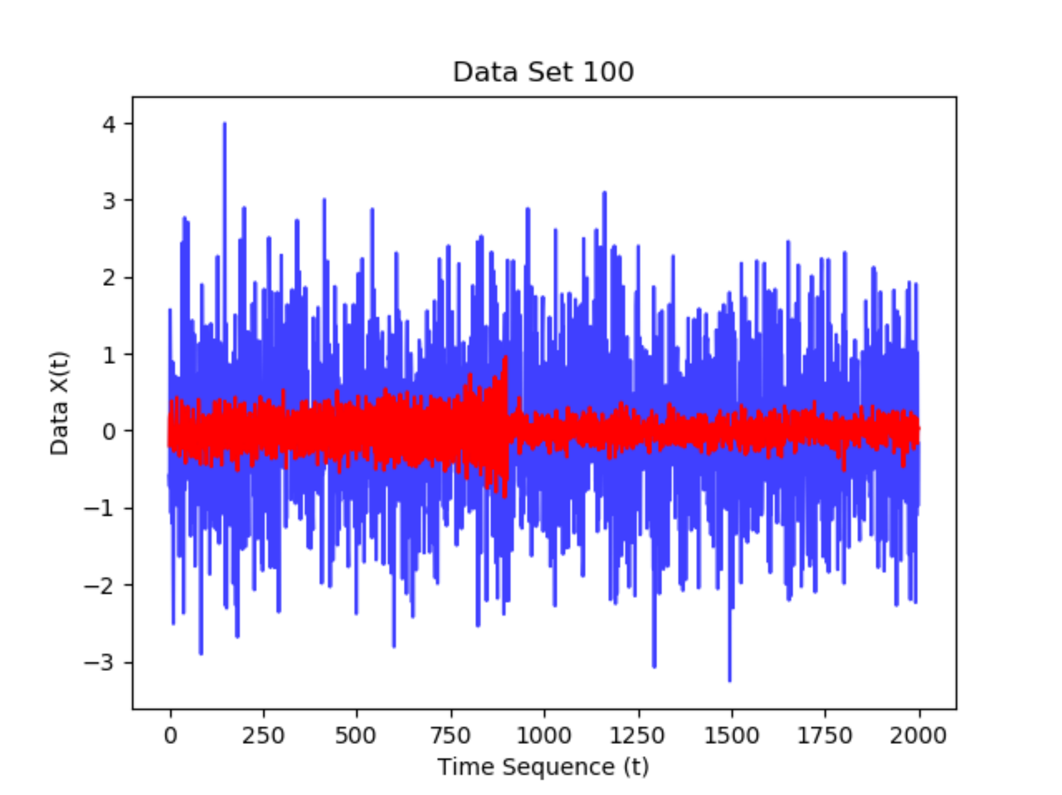
That being said, it is possible to see large improvements.

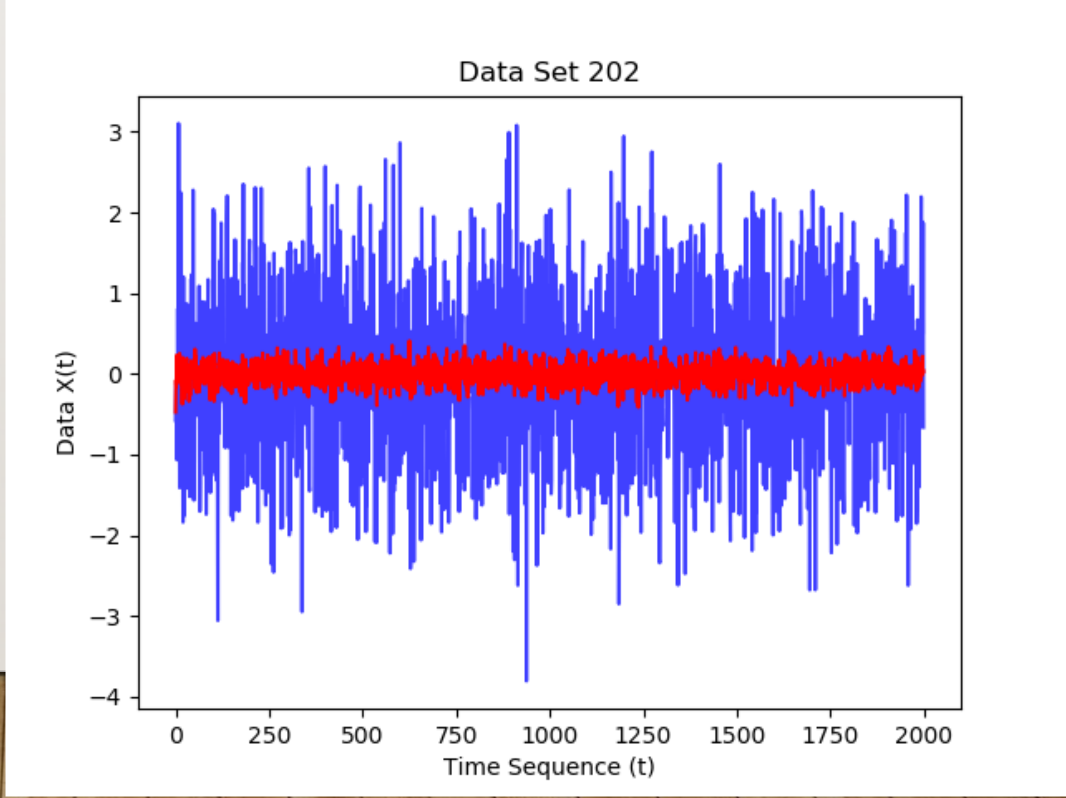
Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

APPLICATION – LIGO SIGNALS

APPLICATION – LIGO SIGNAL CLASSIFICATION

- Consider the binary classification of LIGO signals, where a gravity wave is either (i) present, or (ii) not present, and the signal is masked with Gaussian noise:





APPLICATION – LIGO SIGNAL CLASSIFICATION

- The EXACT details will not be shown here, as it is an activity for you to complete in the next phase of the workshop.
- However, the use of dropout on the input layer of neurons is useful here since (i) our data set contains large random fluctuations, and (ii) the data set is quite large.
- In this case, dropout has a significant effect on classification accuracy:

Run	1	2	3	4	5	6	7	8	9	10
Dropout	99%	94%	96%	97%	97%	98%	98%	95%	97%	97%
No Dropout	65%	69%	69%	67%	68%	67%	68%	69%	73%	65%

DRAWBACKS

- It's not all sunshine and lollypops – you will have to experiment with the many factors involved before you see an improvement with the use of dropout.
- It is likely (almost expected) that your first attempt to use dropout will produce worse performance.
- It's important to keep an eye on the loss during training to ensure convergence has been reached – if the loss is still high (> 0.05) then I recommend increasing the number of epochs. This is no guarantee of success, however.
- In the end, all ML tools require tuning of some sort – this is no exception.

ACTIVITY

- Rewrite your previous ML codes – the square and sine wave classifiers – to include Gaussian noise.
- Create a function filter_data(X) which returns the filtered data. Try using different filters, such as the Butterworth filter and the Chebyshev 1D filter. You can find help by googling "scipy filter".
- Implement dropout on the input layer, and experiment with the fraction of dropout and observe its effect on learning speed and accuracy. You may need to experiment with the number of neurons in the network, but keep it fixed initially.