Published in Towards Data Science



Prasanna Sethuraman Follow

Aug 9, 2020 · 10 min read ★ · Listen



A FUN COMPARISON OF MACHINE LEARNING PERFORMANCE WITH TWO KEY SIGNAL PROCESSING ALGORITHMS — THE FAST FOURIER TRANSFORM AND THE LEAST MEAN SQUARES PREDICTION.

# Machine Learning and Signal Processing

A look at machine learning and neural networks from a Signal Processing perspective.

Signal processing has given us a bag of tools that have been refined and put to very good use in the last fifty years. There is autocorrelation, convolution, Fourier and wavelet transforms, adaptive filtering via Least Mean Squares (LMS) or Recursive Least Squares (RLS), linear estimators, compressed sensing and gradient descent, to mention a few. Different tools are used to solve different problems, and sometimes, we use a combination of these tools to build a system to process signals.

Machine Learning, or the deep neural networks, is much simpler to get used to because the underlying mathematics is fairly straightforward regardless of what network architecture we use. The complexity and the mystery of neural networks lie in the amount of data they process to get the fascinating results we currently have.

### Time Series Prediction

This article is an effort to compare the performance of a neural network for a few key signal processing algorithms. Let us look at time series prediction as the first example. We will implement a three layer sequential deep neural network to predict the next sample of a signal. We will also do it the traditional way by using a tap delay filter and adapting the weights based on the mean square error — this is the LMS filtering, an iterative approach to the optimal Weiner filter for estimating signal from noisy measurement. We will then compare the prediction error between the two methods. So, let us get started with writing the code!

Let us first import all the usual python libraries we need. Since we are going to be using the TensorFlow and Keras framework, we will import them too.

```
#The usual collection of indispensables
import numpy as np
import matplotlib.pyplot as plt
import datetime
import scipy.fftpack

# And the tf and keras framework, thanks to Google
import tensorflow as tf
from tensorflow import keras
```

python_imports.py hosted with ♥ by GitHub view raw

### Prediction with Neural Networks

Let us start building our 3 layer Neural network now. The input layer takes 64 samples and produces 32 samples. The hidden layer

We will use the Adam optimizer without bothering about what it is. That is the benefit of TensorFlow, we don't need to know every detail about all the processing required for neural network to build one using this amazing framework. If we find out that the Adam optimizer doesn't work as well, we will simply try another optimizer — RMSprop for example.

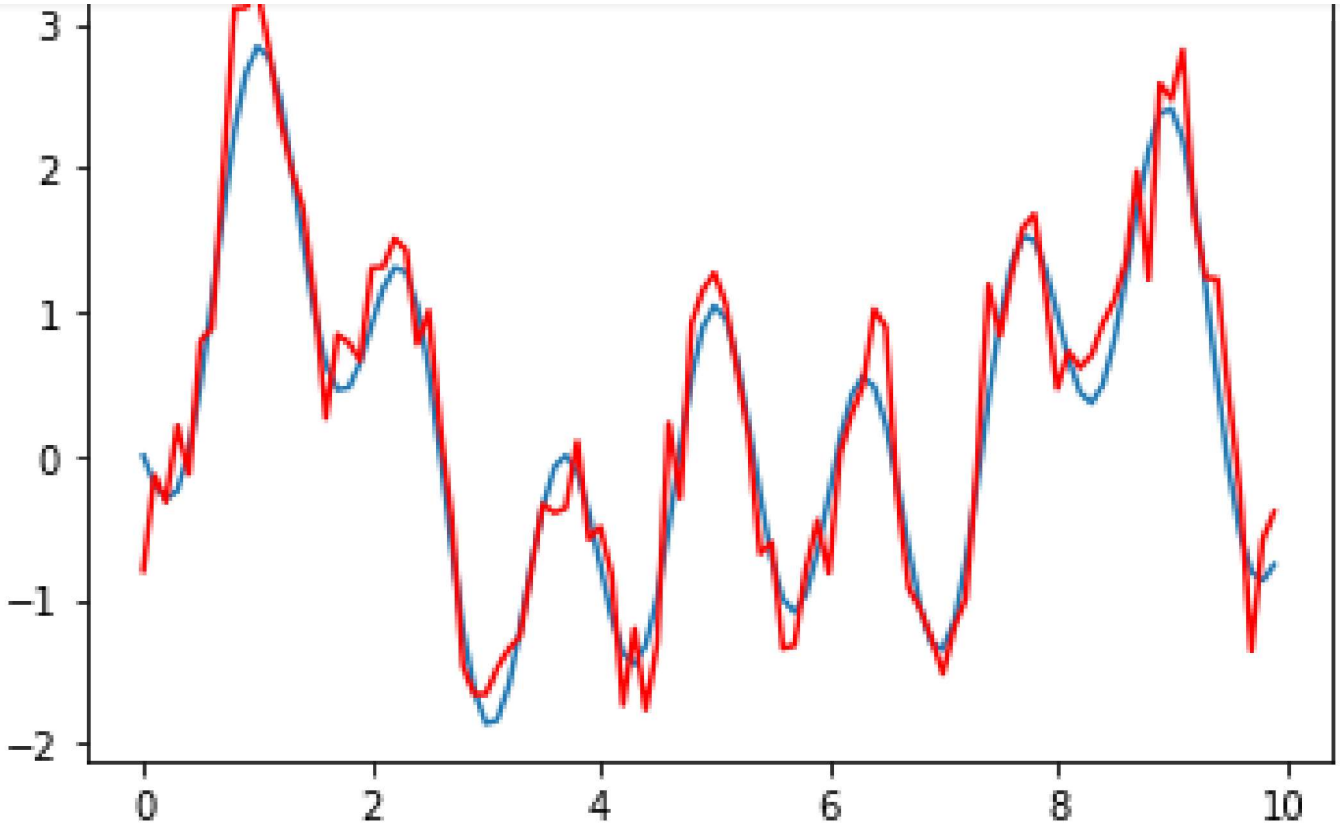
```
# Time series prediction model
def dnn_keras_tspred_model():
  model = keras.Sequential([
    keras.layers.Dense(32, activation=tf.nn.relu,
                       input_shape=(train_data.shape[1],)),
    keras.layers.Dense(8, activation=tf.nn.relu),
    keras.layers.Dense(1)
  ])
  optimizer = tf.keras.optimizers.Adam()
  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae'])
  model.summary()
  return model
```

dnn_tspred.py hosted with ❤ by GitHub view raw

Let us now create a time series, a simple superposition of sine waves. We will then add noise to it to mimic a real world signal.

```
num_train_data = 4000
num_test_data = 1000
timestep = 0.1
tm =  np.arange(0, (num_train_data+num_test_data)*timestep, timestep);
y = np.sin(tm) + np.sin(tm*np.pi/2) + np.sin(tm*(-3*np.pi/2))
SNR = 10
ypn = y + np.random.normal(0,10**(-SNR/20),len(y))

plt.plot(tm[0:100],y[0:100])
plt.plot(tm[0:100],ypn[0:100],'r') # red one is the noisy signal
plt.show()
```

sine_superposition.py hosted with ❤ by GitHub view raw




Now that we have the data, let us think about how to feed this data to the neural network for training. We know the network takes 64 samples at the input and produces one output sample. Since we want to train the network to predict the next sample, we want to use the 65th sample as the output label.

The first input set is therefore from sample 0 to sample 63 (the first 64 samples) and the first label is sample 64 (the 65th sample). The second input set can either be a separate set of 64 samples (non-overlapping window), or we can choose to have a sliding window and take 64 samples from sample 1 to sample 64. Let us follow the sliding window approach, just to generate a lot of training data from the time series we have.

Also note that we are using the noisy samples as the input while using the noiseless data as the label. We want the neural network to predict the actual signal even in presence of noise.

```
# prepare the train_data and train_labels
dnn_numinputs = 64
num_train_batch = 0
train_data = []
for k in range(num_train_data-dnn_numinputs-1):
  train_data = np.concatenate((train_data,ypn[k:k+dnn_numinputs]));
  num_train_batch = num_train_batch + 1
train_data = np.reshape(train_data, (num_train_batch,dnn_numinputs))
train_labels = y[dnn_numinputs:num_train_batch+dnn_numinputs]
```

dnn_tspred_datasets.py hosted with ❤ by GitHub view raw

Let us look at the sizes of the time series data and the training data. See that we generated 5000 samples for our time series data, but we created 3935 x 64 = 251840 samples of input data to our neural network.

The shape of train_data is the number of input sets x input length. Here, we have 3935 batches of input, each input being 64 samples

```
(5000,) (3935, 64) (3935,)
```

We are now ready to train the neural network. Let us instantiate the model first. The model summary provides information on how many layers, what is the output shape, and the number of parameters we need to train for this neural network.

For the first layer, we have 64 inputs and 32 outputs. A dense layer implemets the equation $y = f(Wx + b)$, where $f$ is the activation function, $W$ is the weight matrix and $b$ is the bias. We can immediately see that $W$ is a 64 x 32 matrix, and $b$ is a 32 x 1 vector. This gives us 32 x 64 + 32 = 2080 parameters to train for the first layer. The reader can do similar computations to verify the parameters for second and third layer, as an exercise in understanding. After all, you wouldn't be reading this article unless you are a beginner to Machine Learning and eager to get started :)

```
model = dnn_keras_tspred_model()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                2080
_________________________________________________________________
dense_1 (Dense)              (None, 8)                 264
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 9
=================================================================
Total params: 2,353
Trainable params: 2,353
Non-trainable params: 0
_________________________________________________________________
```

Alright, onward to training then. Since we are using the Keras framework, training is as simple as calling the fit() method. With TensorFlow, we need to do a little more work, but that is for another article.

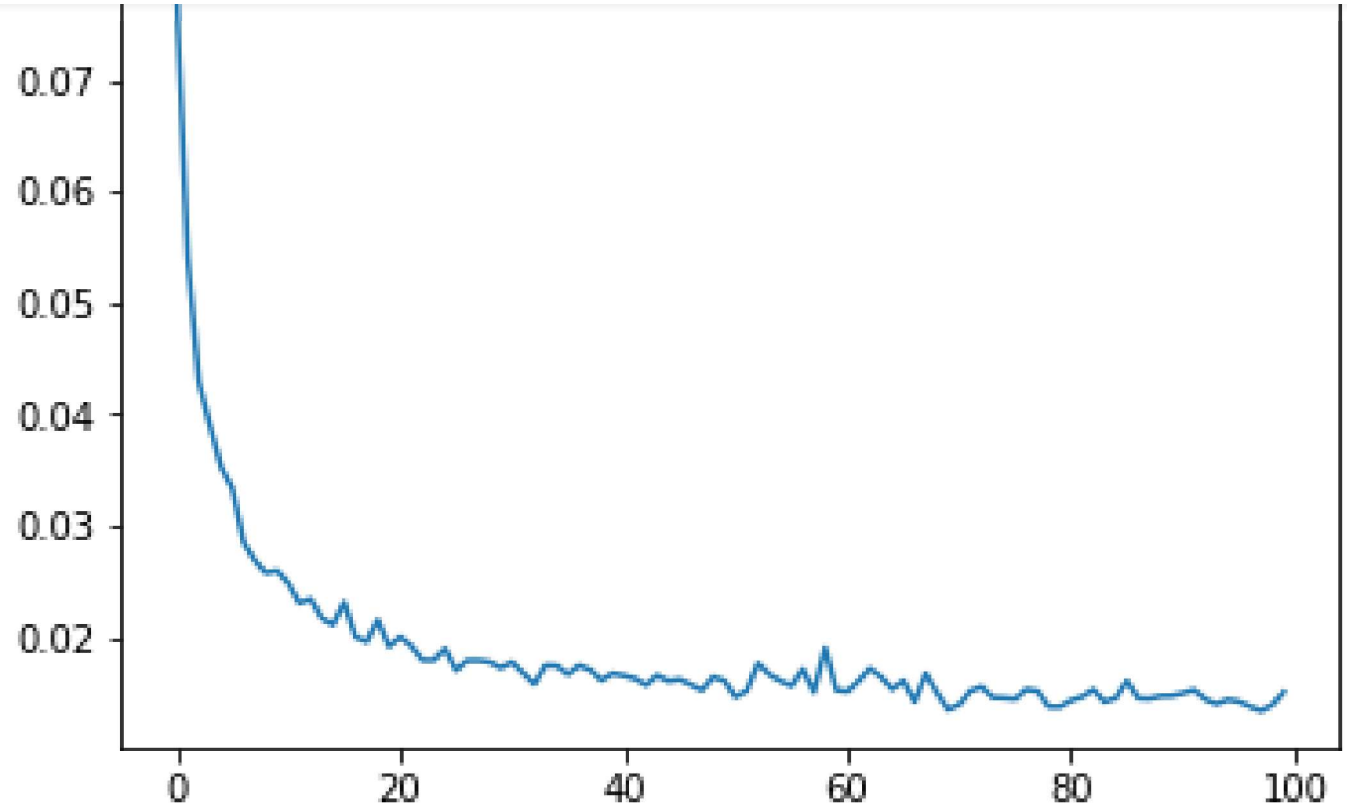
Let us use 100 epochs, which just means that we will use the same training data again and again to train the neural network and will do so 100 times. In each epoch, the network uses the number of batches of input and label sets to train the parameters.

Let us use the *datetime* to profile how long this training takes, and the history value that is returned as a Python Dictionary to get the validation loss after each epoch.

```
EPOCHS = 100
strt_time = datetime.datetime.now()
history = model.fit(train_data, train_labels, epochs=EPOCHS,
                    validation_split=0.2, verbose=0,
                    callbacks=[])
curr_time = datetime.datetime.now()
timedelta = curr_time - strt_time
dnn_train_time = timedelta.total_seconds()
print("DNN training done. Time elapsed: ", timedelta.total_seconds(), "s")
plt.plot(history.epoch, np.array(history.history['val_loss']),
           label = 'Val loss')
plt.show()
```

train_dnn_dspred_model.py hosted with ❤ by GitHub view raw

```
DNN training done. Time elapsed:  10.177171 s
```




Now that the network is trained, and we see that the validation loss has decreased over epochs to the point that it has flattened out (indicating further training doesn't yield any significant improvements), let us use this network to see how well it performs against test data.

Let us create the test data set exactly the same way as we created the training data sets, but use only that part of the time series that we have not used for training before. We want to surprise the neural network with data it has not seen before to know how well it can perform.

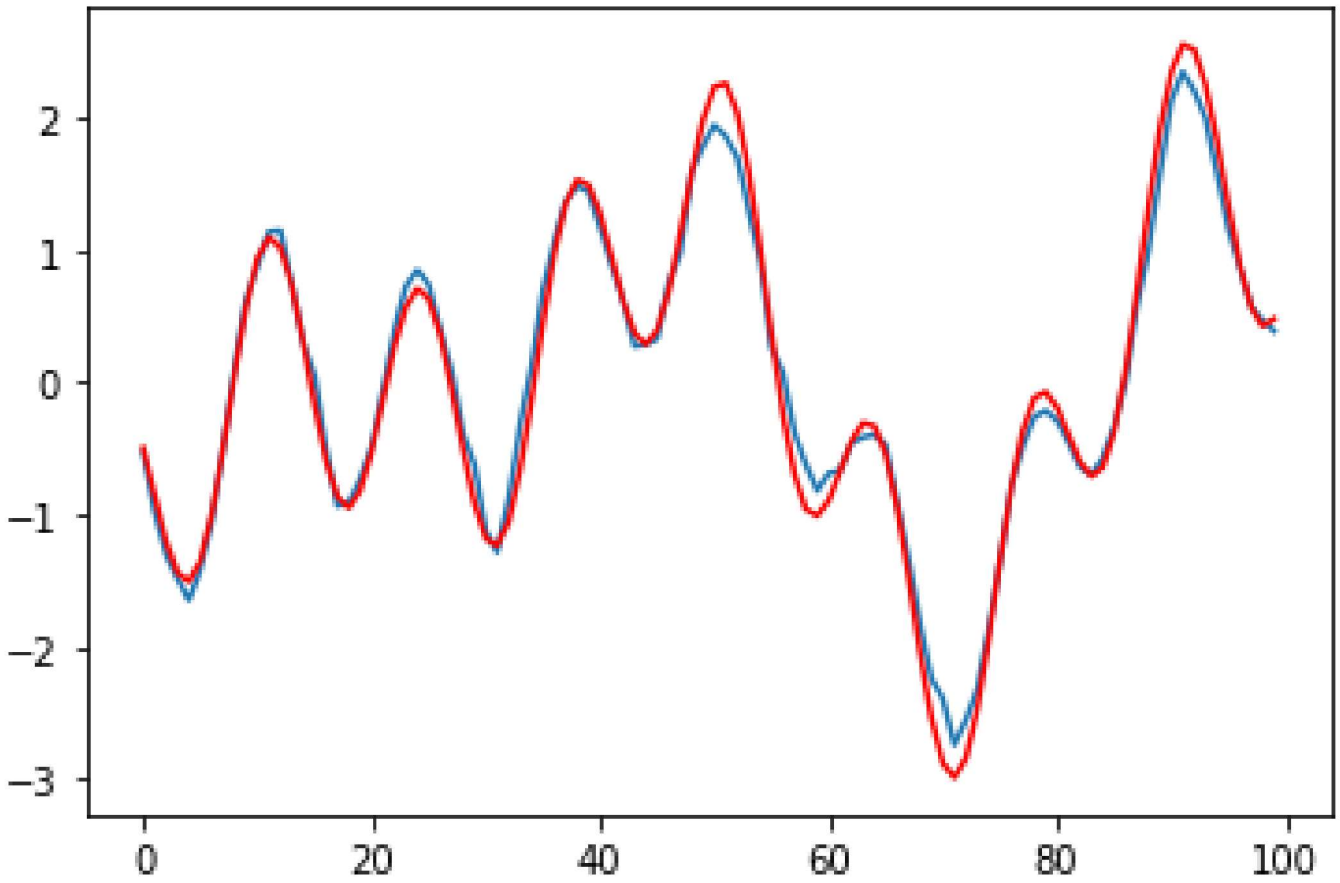
```
# test how well DNN predicts now
num_test_batch = 0
strt_idx = num_train_batch
test_data=[]
for k in range(strt_idx, strt_idx+num_test_data-dnn_numinputs-1):
  test_data = np.concatenate((test_data,ypn[k:k+dnn_numinputs]));
  num_test_batch = num_test_batch + 1
test_data = np.reshape(test_data, (num_test_batch, dnn_numinputs))
test_labels = y[strt_idx+dnn_numinputs:strt_idx+num_test_batch+dnn_numinputs]
```

test_dnn_tspred.py hosted with ❤ by GitHub view raw

We will now call the *predict()* method in Keras framework to get the outputs of the neural network for the test data set. This step is different if we use the TensorFlow framework, but we will cover that in another article.

As we see, the prediction from neural network is very close to the actual noise free data!

```
dnn_predictions = model.predict(test_data).flatten()
keras_dnn_err = test_labels - dnn_predictions
plt.plot(dnn_predictions[0:100])
```




**Prediction with LMS algorithm**

We will use a $L=64$ tap filter to predict the next sample. We don't need that large a filter, but let us keep the number of inputs per output sample same as what we used for neural network.

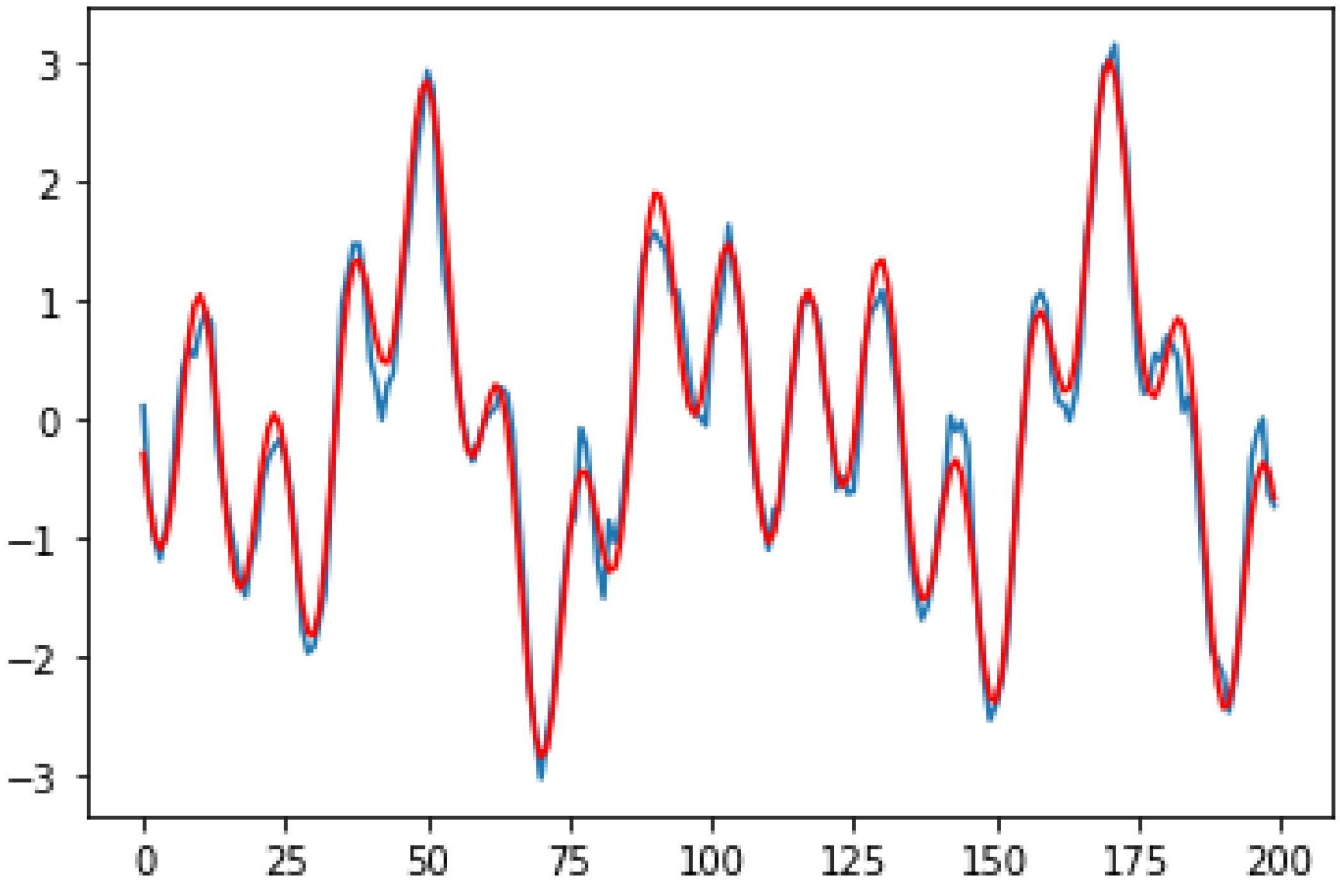
The filter coefficients (or weights) are obtained by computing the error between predicted and measured sample, and adjusting the weights based on the correlation between mean square error and the input measurements.

As you see in the code, *yrlms[k]* is the filter output when the inputs are *ypn[k-L:k]*, the error is computed as the difference between the noisy measured value *ypn[k]* and the filter output *yrlms[k]*. The correlation between measurement and error is given by the product of *ypn[k-L:k]* and *e*, and *mu* is the LMS step size (or learning rate).

As we see, the LMS prediction is equally good, despite having much lower complexity.

```
#LMS
M = 1000
L = 64
yrlms = np.zeros(M+L)
#wn = np.random.normal(0,1,L)
wn = np.zeros(L)
print(wn.shape, yrlms.shape)
mu = 0.005
for k in range(L,M+L):
  yrlms[k] = np.dot(ypn[k-L:k],wn)
  e = ypn[k]- yrlms[k]
  wn=wn+(mu*ypn[k-L:k]*e)

plt.plot(yrlms[600:800])
```

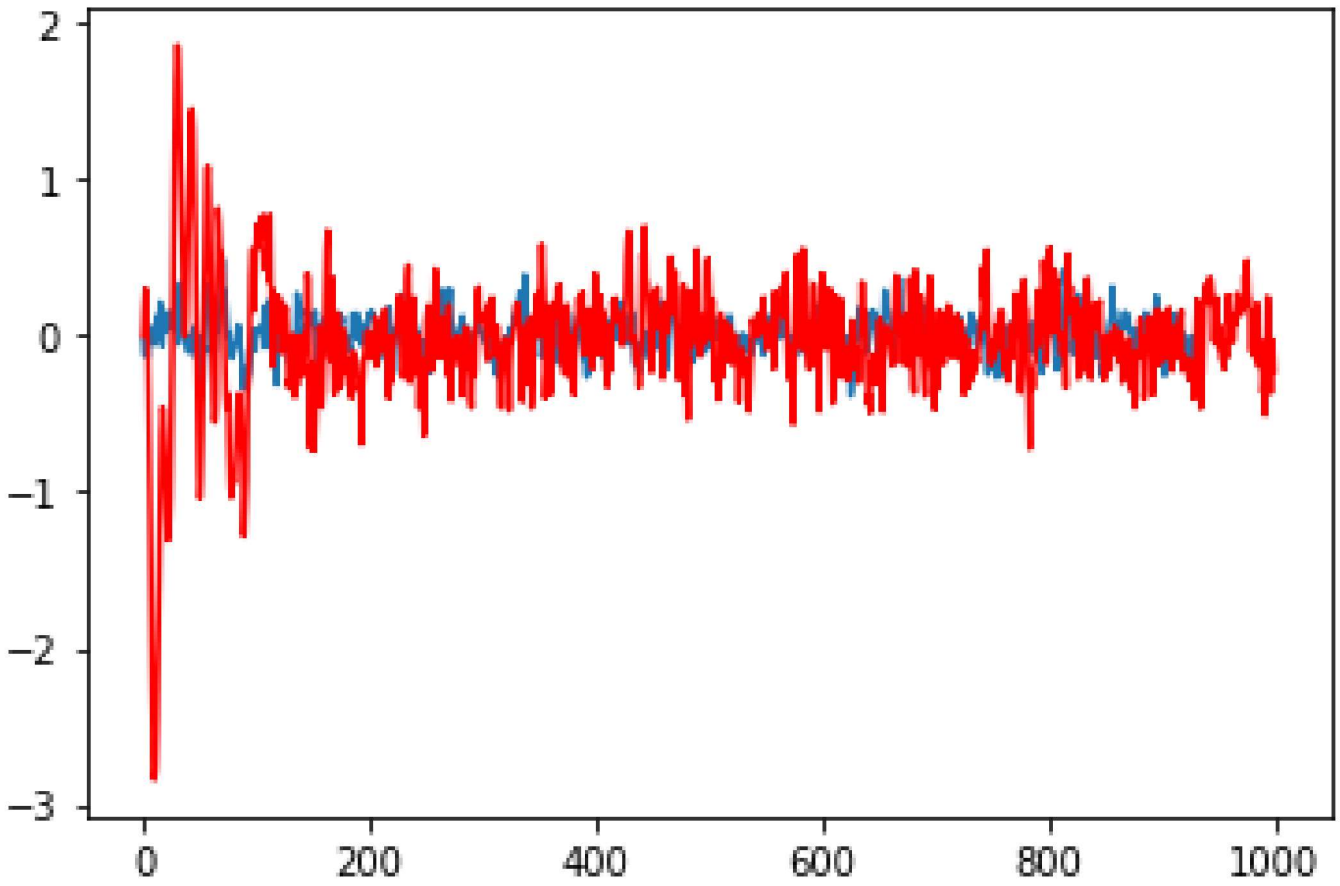
```
(64,) (1064,)
```




**Comparing Prediction Results between LMS and Neural Network**

Before we close this section, let us compare the error between LMS prediction and the neural network prediction. To be fair, I ignored the initial portion of the LMS to give it time to converge when measuring the mean square error and SNR. Despite that, we see that the neural network performance is 5 dB better than the LMS performance!

```
dnn_err = dnn_predictions - test_labels
lms_err = yrlms[0:M] - y[0:M]
plt.plot(dnn_err)
plt.plot(lms_err,'r')
plt.show()

dnn_mse = 10*np.log10(np.mean(pow(np.abs(dnn_err),2)))
lms_mse = 10*np.log10(np.mean(pow(np.abs(lms_err[200:M]),2)))
lms_sigpow = 10*np.log10(np.mean(pow(np.abs(y[0:M]),2)))
dnn_sigpow = 10*np.log10(np.mean(pow(np.abs(test_labels),2)))

#print(dnn_mse, dnn_sigpow, lms_mse, lms_sigpow)
print("Neural network SNR:", dnn_sigpow - dnn_mse)
print("LMS Prediction SNR:", lms_sigpow - lms_mse)
```




```
Neural network SNR: 19.986311477279084
LMS Prediction SNR: 14.93359076022336
```

**Fast Fourier Transform**

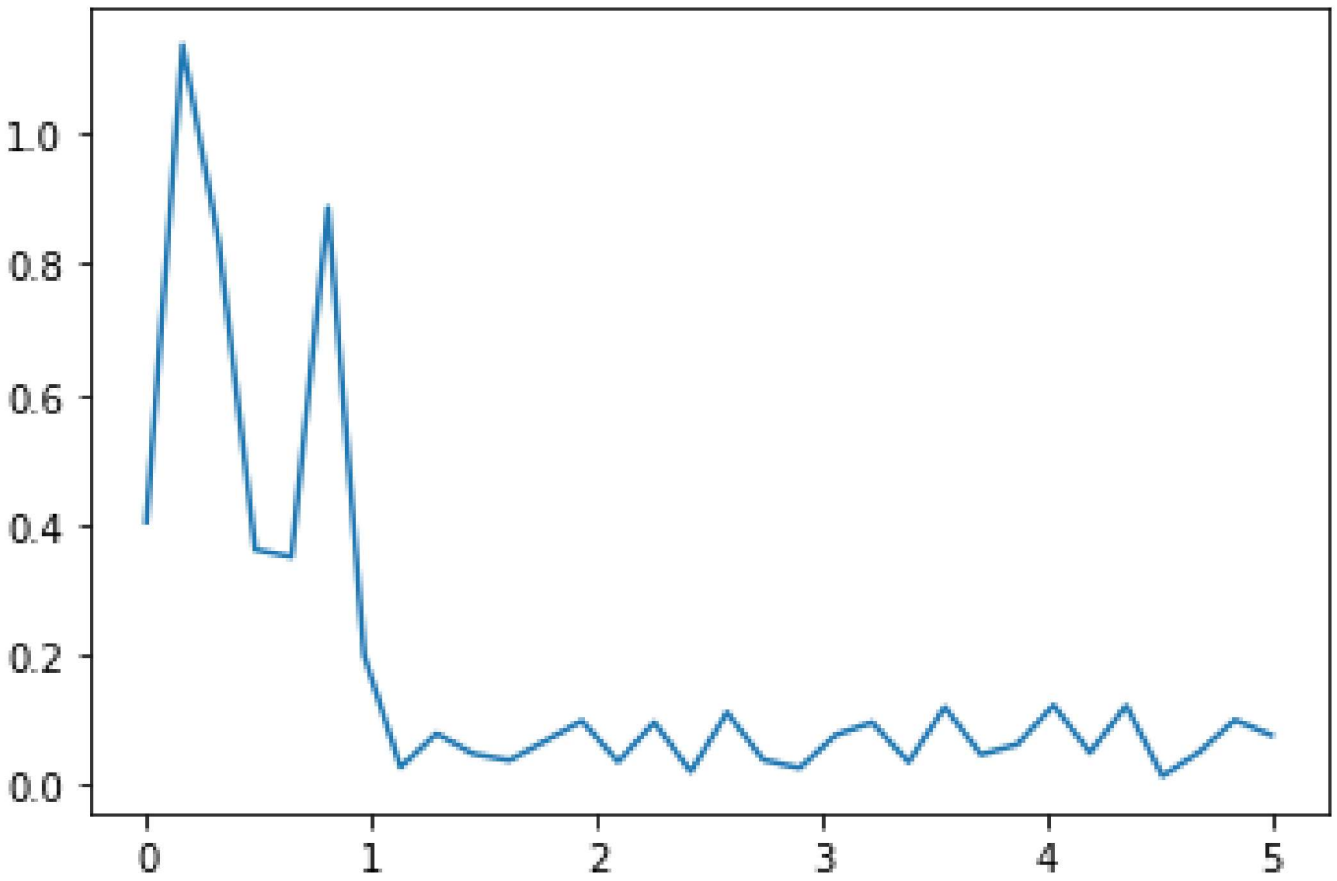
Alright, a neural network beat LMS by 5 dB in signal prediction, but let us see if a neural network can be trained to do the Fourier Transform. We will compare it to the FFT (Fast Fourier Transform) from SciPy FFTPack. The FFT algorithm is at the heart of signal processing, can the neural network be trained to mimic that too? Let us find out…

We will use the same signal we created before, the superposition of sine waves, to evaluate FFT as well. Let us look at the FFT ouput first.

```
# 64 point FFT
N = 64

# Using the same noisy signal used for LMS
yf = scipy.fftpack.fft(ypn[0:N])

# Let us remove noise, easy to do at the FFT output
#yc = np.zeros(N,dtype=complex)
#cidx = np.where(np.abs(yf)>(N*0.2/2))[0]
#yc[cidx]=yf[cidx]

# 0 to Fs/2, Fs = 1/Ts
```

```
plt.plot(xf, 2.0/N * np.abs(yf[:N//2]))
plt.show()
```

fft_snip.py hosted with ❤ by GitHub view raw




Let us create a neural network model to mimic the FFT now. In contrast to the model we created before where we have 64 inputs but only one output, this model needs to generate 64 outputs for every 64 sample input set.

And since FFT inputs and outputs are complex, we need twice the number of samples at the input, arranged as real followed by imaginary. Since the outputs are also complex, we again 2 x NFFT samples.

```
def dnn_keras_fft_model():
  model = keras.Sequential([
    keras.layers.Dense(NFFT*2, activation=tf.nn.relu,
                       input_shape=(train_data.shape[1],)),
    keras.layers.Dense(NFFT*2, activation=tf.nn.relu),
    keras.layers.Dense(NFFT*2)
  ])
  optimizer = tf.keras.optimizers.Adam()
  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae'])
```

To train this neural network model, let us use random data generated using *numpy.random.normal* and set the labels based on the FFT routine from the SciPy FFTPack that we are comparing with.

The rest of the code is fairly similar to the previous neural network training. Here, I am running 10,000 batches at a time, and I have an outer for loop to do multiple sets of 10,000 batches if the network needs more training. Note that this needs the model to be created outside the for loop, so that the weights are not reinitialized.

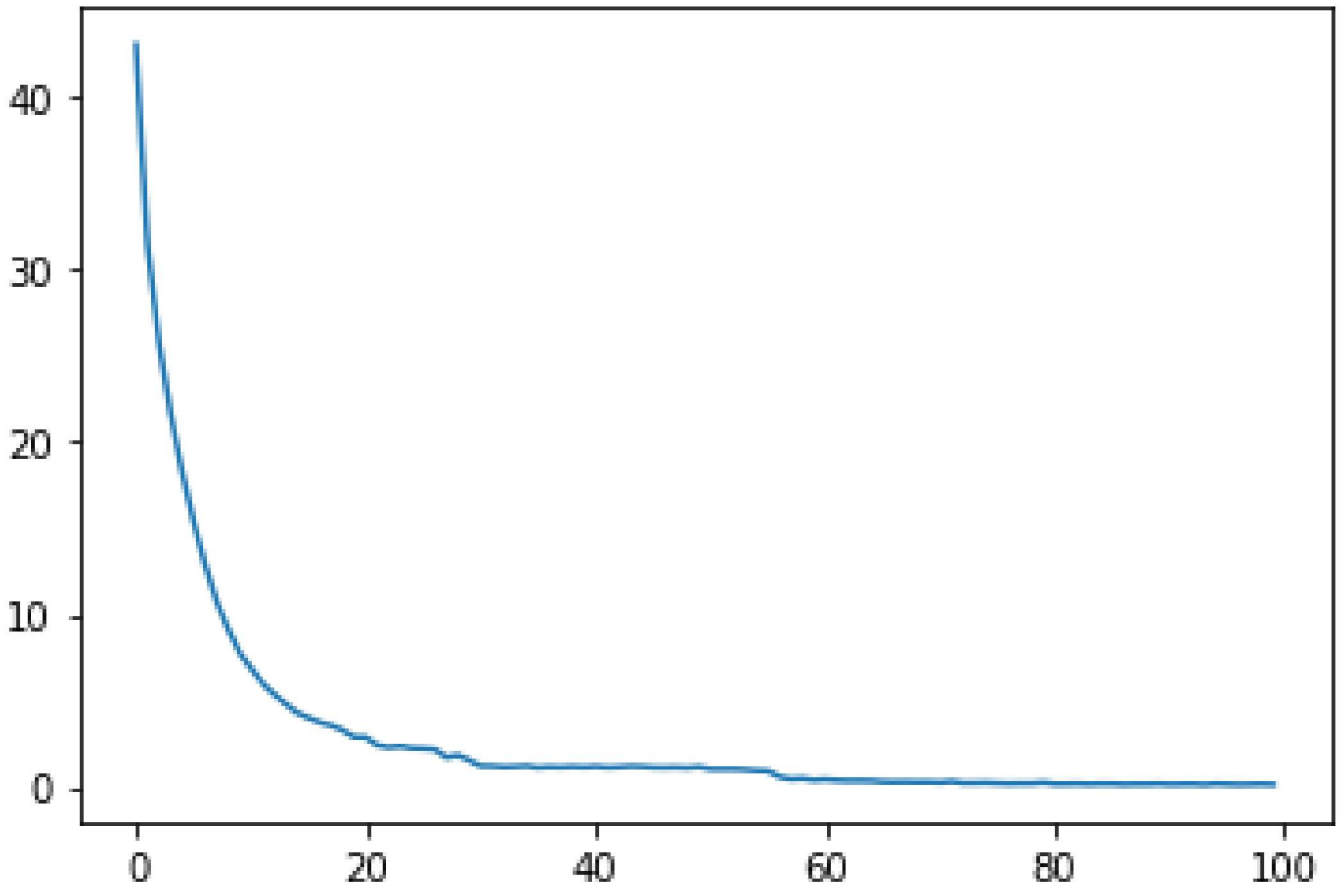
See from model summary that there are almost 50,000 parameters for just a 64 point FFT. We can reduce this a bit since we are only evaluating real inputs while keeping the imaginary parts as zero, but the goal here is to quickly compare if the neural network can be trained to do the Fourier Transform.

```
# Train the DNN for 16 point FFT
NFFT = 64
num_train_batch = 1
num_batches = 10000
train_data = np.random.normal(0,1,(num_batches, NFFT*2))
train_labels = np.random.normal(0,1,(num_batches, NFFT*2))
model = dnn_keras_fft_model()
for k in range(num_train_batch):
  for el in range(num_batches):
    fftin = train_data[el,0::2] + 1j*train_data[el,1::2]
    train_labels[el,0::2]=scipy.fftpack.fft(fftin).real
    train_labels[el,1::2]=scipy.fftpack.fft(fftin).imag
  EPOCHS = 100
  strt_time = datetime.datetime.now()
  history = model.fit(train_data, train_labels, epochs=EPOCHS,
                      validation_split=0.2, verbose=0,
                      callbacks=[])
  curr_time = datetime.datetime.now()
  timedelta = curr_time - strt_time
  dnn_train_time = timedelta.total_seconds()
  print("DNN training done. Time elapsed: ", timedelta.total_seconds(), "s")
  plt.plot(history.epoch, np.array(history.history['val_loss']),
           label = 'Val loss')
  plt.show()
  train_data = np.random.normal(0,1,(num_batches, NFFT*2))
```

train_dnn_fft_model.py hosted with ❤ by GitHub view raw

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

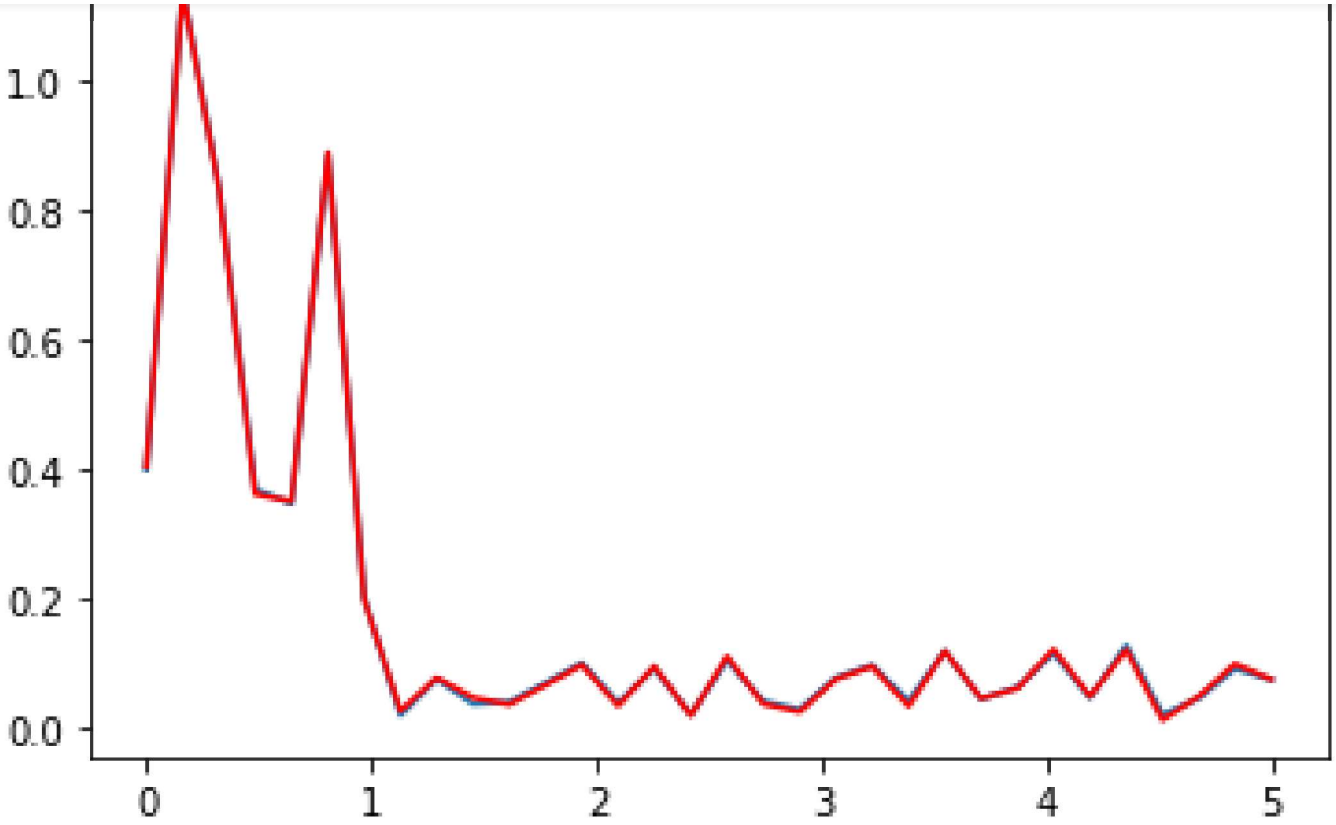
```
=================================================================
Total params: 49,536
Trainable params: 49,536
Non-trainable params: 0
_________________________________________________________________
DNN training done. Time elapsed:  30.64511 s
```




Training is done. Let us now test the network using the same input samples we created for LMS. We compare the neural network output to the FFT ouput and they are identical! How amazing is that!

```
fftin = np.zeros((1,2*NFFT))
fftin[:,0::2]=ypn[0:NFFT]
fftout = model.predict(fftin).flatten()
fftout = fftout[0::2] + 1j*fftout[1::2]
plt.plot(xf, 2.0/NFFT * np.abs(fftout[0:NFFT//2]))
plt.plot(xf, 2.0/N * np.abs(yf[:N//2]),'r')
plt.show()
```

plot_fft_vs_dnn_fft.py hosted with ❤ by GitHub view raw

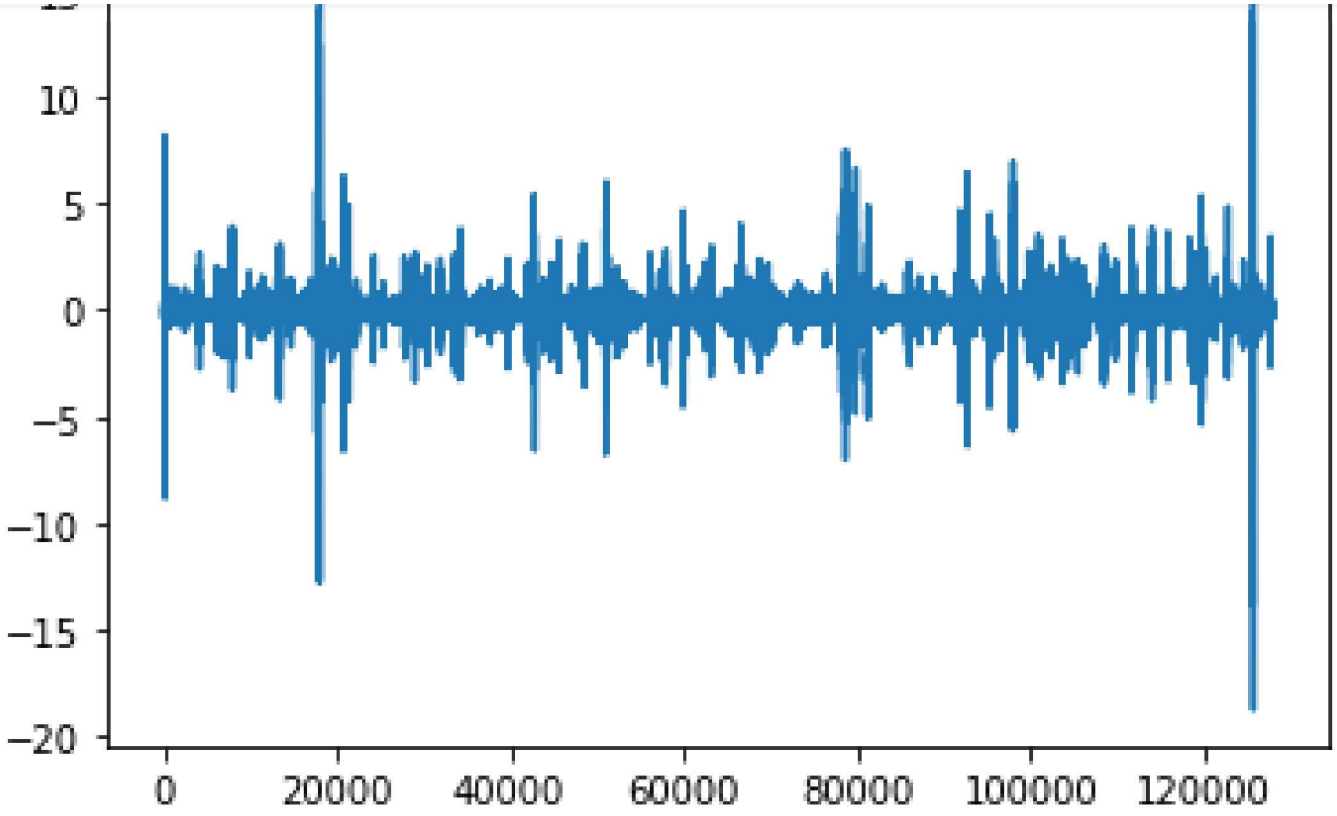



Let us do one last evaluation before we conclude this article. We will compare the neural network output with the FFT output for some random input data, and see how the mean square error and SNR looks like.

Running the code below, we get a decent 23.64 dB SNR. While we do see some samples every now and then where the error is high, for most part, the error is very small. Given that we trained the neural network for only 10,000 batches, this is a pretty good result!

```
test_data = np.random.normal(0,1,(1000, NFFT*2))
test_labels = np.random.normal(0,1,(1000, NFFT*2))
for el in range(1000):
  fftin = test_data[el,0::2] + 1j*test_data[el,1::2]
  test_labels[el,0::2]=scipy.fftpack.fft(fftin).real
  test_labels[el,1::2]=scipy.fftpack.fft(fftin).imag

dnn_out = model.predict(test_data).flatten()
keras_dnn_err = test_labels.flatten() - dnn_out
plt.plot(keras_dnn_err)
plt.show()

dnn_fft_mse = 10*np.log10(np.mean(pow(np.abs(keras_dnn_err),2)))
labels_sigpow = 10*np.log10(np.mean(pow(np.abs(test_labels.flatten()),2)))
print("Neural Network SNR compare to SciPy FFT: ", labels_sigpow - dnn_fft_mse)
```

plot_err_dnn_fft_vs_fft.py hosted with ❤ by GitHub view raw




```
Neural Network SNR compared to SciPy FFT:  23.64254974707859
```

**Summary**

Being stuck inside during Covid-19, it was a fun weekend project to compare machine learning performance to some key signal processing algorithms. We see that machine learning can do what signal processing can, but has inherently higher complexity, with the benefit of being generalizable to different problems. The signal processing algorithms are optimal for the job in terms of complexity, but are specific to the particular problems they solve. We can't use FFT in place of LMS or vice versa, while we can use the same neural network processor, and just load a different set of weights to solve a different problem. That is the versatility of neural networks.

And with that note, I'll conclude this article. I hope you had as much fun reading this as I had putting this together. Please leave your feedback too if you found it helpful and learnt a thing or two!

https://www.linkedin.com/in/prasannasethuraman/