ENGG 406 Coursework

Artificial Neural Networks as a Surrogate Model

William Kane

201068461

Introduction

The objective of this report is to assess the probability of failure of high-fidelity model using an artificial neural network as a surrogate model.

Highly reliable systems are designed to have a low probability of failure, e.g. P(fail) = 10^{-4}. Models of complex systems are often infeasible to analyse analytically so require numerical techniques such as a Monte Carlo simulation to estimate P(fail) but for some high-fidelity models even this comes with a huge computational cost. Artificial neural networks (ANN) can be used as a substitute as they can be trained to mimic an equation.

In this report an ANN will be trained to mimic equation 1 using the python package TensorFlow. The accuracy will be analysed, P(fail) will be estimated, and it will be compared with a crude Monte Carlo method.

The model to be mimicked is

$$f(X) = 15.59 \times 10^4 - \frac{X_1\, X_2^2}{2\, X_2^3} \times \frac{X_4^2 - 4\, X_5\, X_6\, X_7^2 + X_4(X_6 + 4\, X_5 + 2\, X_6\, X_7)}{X_4\, X_5\, (X_4 + X_6 + 2\, X_6\, X_7)} \qquad (1)$$

Where $X_1, \dots, X_7$ are normal random variables with means and standard deviations given by

Variable	Mean	Std
X_1	350	35
X_2	50.8	5.08
X_3	3.81	0.381
X_4	173	17.3
X_5	9.38	0.938
X_6	33.1	3.31
X_7	0.036	0.0036

Failure is defined as $f(X) \leq 0$

Training a Neural Network

Preparation

To allow for easy modelling let $f(x) = 155900 - g(x)$. This means the event $f(X) \leq 0$ is mathematically equivalent to $g(X) \geq 155900$. It was found during training that this method is results in a significantly more accurate neural network.

With the problem defined thus, the equation can be set up in python, where $g(X)$ is a function with 7 arguments called *eqn*.

[1]:
```
# Define g(x)
def eqn(a,b,c,d,e,f,g):
    f = ( ( a*(b**2) )/( 2*(b**3) ) )*( ( d**2 - 4*e*f*(g**2) + d*(f + 4*e + 2*f*g) )/( d*e*(d+f+2*f*g) ))
    return f
```

[2]:
```
# Define means and std of all 7 Normal RVs
m1, s1 = 350, 35
m2, s2 = 50.8, 5.08
m3, s3 = 3.81, 0.381
m4, s4 = 173, 17.3
m5, s5 = 9.38, 0.938
m6, s6 = 33.1, 3.31
m7, s7 = 0.036, 0.0036
```

Generating Data

For this section the following imports are made.

[3]:
```
# Imports
import numpy as np
import pandas as pd       # For the dataframe class
import seaborn as sns     # For plotting
sns.set(color_codes=True) # Config sns
from pyDOE import lhs     # For Latin hypercube
import scipy.stats as st  # For stats
```

Next training and test data need to be generated. A sample of 10,000 is taken from a 7 factor Latin-hypercube and is fed into the inverse-cdf of each normal distribution for the random variables. Latin hypercube sampling like this is used to increase the accuracy of the neural network. This results in a 7x10,000 matrix where each row corresponds to a sample from each variable. This matrix is turned into a Pandas DataFrame object for easier manipulation, called *ds* for dataset.

[4]:
```
# Generate sample
ss = 10000     # Sample size
means = [m1,m2,m3,m4,m5,m6,m7]
cov = np.diag([s1,s2,s3,s4,s5,s6,s7])

# Latin hypercube sampling
latin = lhs(7, samples=ss) # 7D lhc between 0 and 1
for i in range(7):
    latin[:,i] = st.norm(loc=means[i], scale=cov[i,i]).ppf(latin[:,i]) # reverse cdf: [0,1] -> Normals
X = latin

# Convert to pandas datafame
ds = pd.DataFrame(X)
```

Using *eqn* and *ds*, the outputs for all 10,000 samples are calculated and added to the DataFrame in a new column. The first 5 rows of the DataFrame are displayed and all the entries appear appropriate and as expected. Note the variables are indexed 0 to 6 for $X_1, \ldots, X_7$.

[5]:
```
# Generate outputs, g(each sampling)
out = []
for i in range(0, ss):
    y = eqn(*X[i])
    out.append(y)
#out
```

[6]:
```
# Add oututs to dataframe
ds['output'] = out
ds.head()
```

[6]:

	0	1	2	3	4	5	6	output
0	283.464377	42.622090	3.776587	194.985163	9.659426	28.942795	0.038946	0.403051
1	347.060992	49.418671	3.405324	164.981166	9.338923	39.516518	0.034388	0.443764
2	346.596246	55.819743	4.009463	174.059800	8.668830	29.279785	0.030322	0.418668
3	335.839553	53.363500	4.235054	200.221658	9.264288	36.984463	0.040844	0.392040
4	331.181948	43.696951	3.779803	155.083349	11.036799	35.497052	0.035957	0.421816

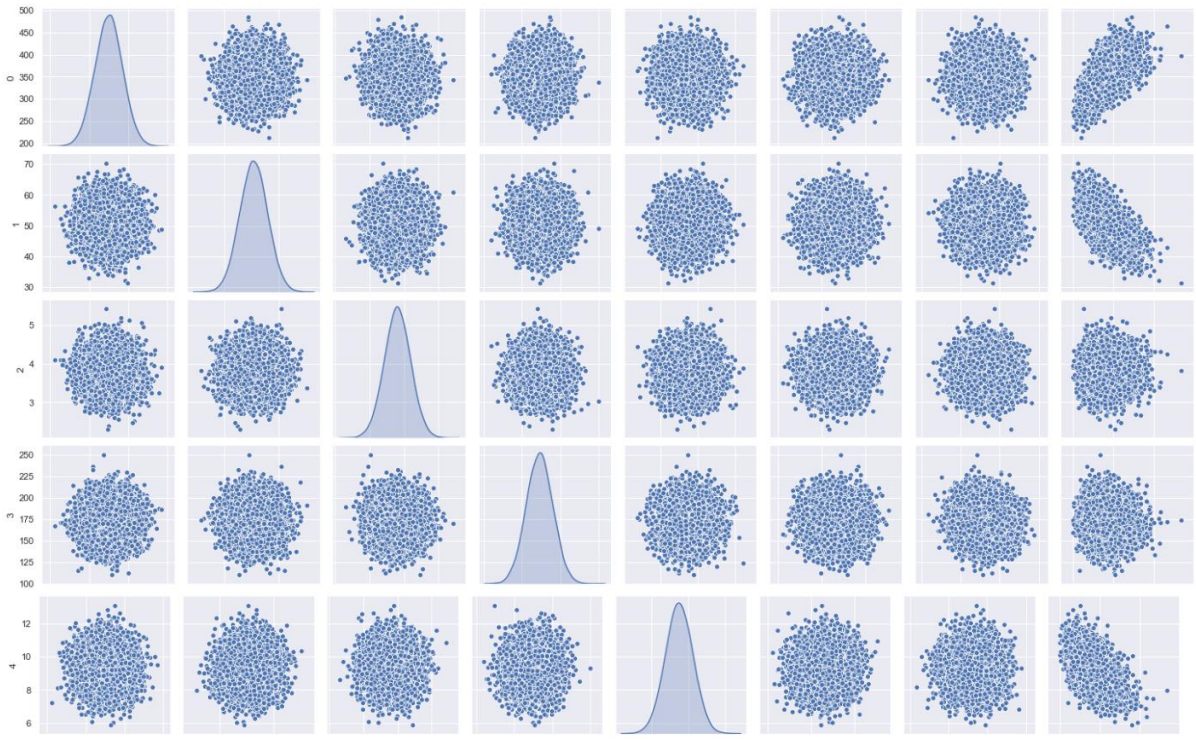
To train and test a neural network the dataset is split 80:20 into training data and test data sets.

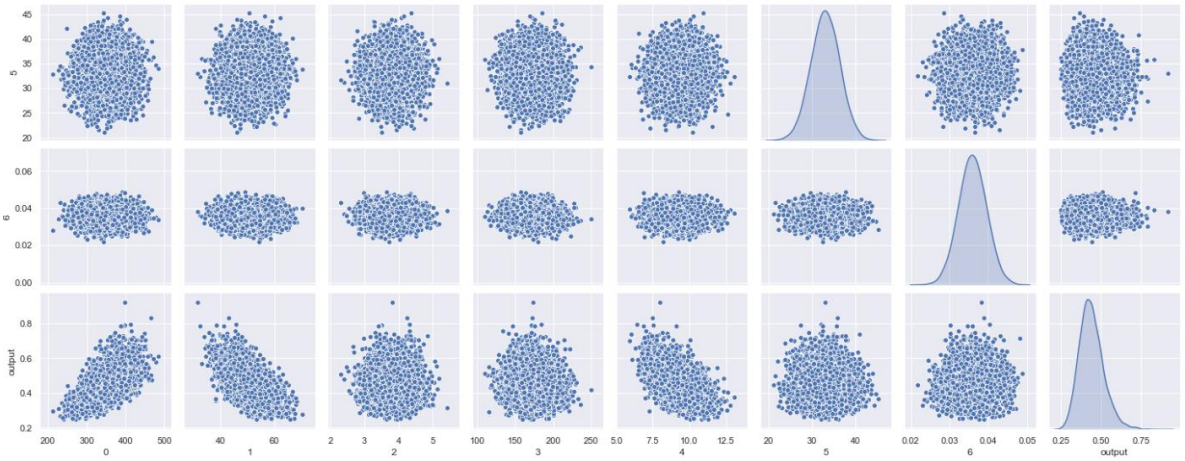
[7]:
```
# create training and testing ds, 80:20 split
d_train = ds.sample(frac=0.8,random_state=0) # 80% sample of ds
d_test = ds.drop(d_train.index)              # ds - d_train
```

Before training, all the data can be visualized. Seaborn's pair wise plot can quickly plot the kernel-density estimation (KDE) for each column and pair-wise scatter plots for all columns.

[8]:
```
%%time
# Visualise all distros
sns.pairplot(d_train, diag_kind="kde")
```

```
Wall time: 3.6 s
```





All the pair-wise plots show joint-normal properties as expected. All variable KDEs appear approximately normal. It is also noteworthy that the output KDE (bottom right) is approximately a skewed normal distribution, this is used later in analysis.

Building a Neural Network

To train a neural network in python, the TensorFlow package is used. The package was developed by Google for internal use and was published publicly in 2015, and represents one of the best packages available for machine learning in python.

TensorFlow is imported to python. As the package has support for GPU acceleration with the correct drivers, a simple command can be ran to check whether TensorFlow has recognised a usable GPU, which it did. This use of parallel processing will speed up training and other calculations made with TensorFlow.

[9]:
```
# Imports
import tensorflow as tf       # Package for machine learning
from tensorflow import keras
from tensorflow.keras import layers

# Check GPU found for GPU acceleration. Requries nvidia GPU with CUDA ANN drivers
print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

The next step is to separate the outputs from each DataFrame. These are saved as their own DataFrame.

[10]:
```
# Seperate input and output
l_train = d_train.pop('output')
l_test  = d_test.pop('output')
print(d_train.head())
print(l_train.head())
```

To speed up training the input data needs to be normalised.

[11]:
```
# Nomalise the data for faster training
train_stats = d_train.describe().transpose()
test_stats = d_test.describe().transpose()

def norm_train(x):
    return (x - train_stats['mean']) / train_stats['std']
def norm_test(x):
    return (x - test_stats['mean']) / test_stats['std']


d_train = norm_train(d_train)
d_test = norm_test(d_test)
```

Next the architecture for the ANN is set up. Initially the network will have 7 input nodes, then hidden layers with 5, 25, and 29 nodes, and a single output node. To optimise the ANN the RMSprop algorithm is used as it efficiently converges. The number of epochs is set at 1000 but an early stopping callback is used. This checks the accuracy of the ANN every 10 epochs against a validation set in the training data and stops the training when the accuracy does not improve, thus preventing overfitting. The model is complied into an object called *model*.

[12]:
```
# Set up model architecture
EPOCHS = 1000 # Early stopping will be used

model = keras.Sequential([
            layers.Dense(5, activation='relu', input_shape=[7], use_bias=True),
            layers.Dense(25, activation='relu',use_bias=True),
            layers.Dense(29, activation='relu',use_bias=True),
            layers.Dense(1)
            ])
optimizer = tf.keras.optimizers.RMSprop(0.001)
model.compile(loss='mse', optimizer=optimizer, metrics=['mae', 'mse'])

early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)
#This line will stop the model early if it doesnt improve much in the last 10 epochs in the validation data, prevents overfitting
```

A simple command can check the model is set up correctly.

[13]:
```
# Check Model
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 5)                 40
_________________________________________________________________
dense_1 (Dense)              (None, 25)                150
_________________________________________________________________
dense_2 (Dense)              (None, 29)                754
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 30
=================================================================
Total params: 974
Trainable params: 974
Non-trainable params: 0
_________________________________________________________________
```

As this is correct the model is fitted to the training data. The *fit* method is used and a validation split of 20% is used to create non-training validation data which is used by the early stopping callback.

[14]:
```
%%time
# Fit the model and save its history for analysis
history = model.fit(d_train, l_train, epochs=EPOCHS, validation_split = 0.2, verbose=0, callbacks=[early_stop])
```

```
Wall time: 16.7 s
```

This run took 16.7 seconds to train. Training speed will be investigated in the analysis.

Analysis

Initial run and Techniques

The Mean Absolute Error (MAE) of this neural network can be easily found from the test data.

[15]:
```
# Evaluate performance on test data
loss, mae, mse = model.evaluate(d_test, l_test, verbose=0)
print("Testing set Mean Abs Error:", mae)
```

```
Testing set Mean Abs Error: 0.008457957
```

The accuracy here is very good at MAE = 0.008. This is easily enough to test the model for the failure events. But first some analysis techniques that will be useful in exploring the performance.

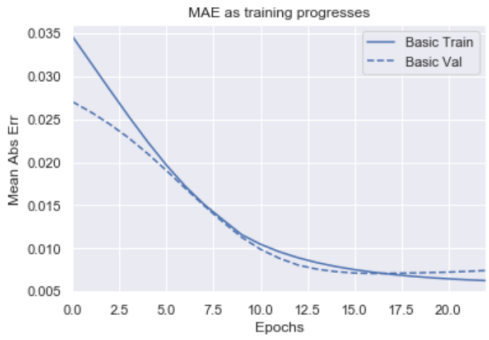
These imports are made.

[16]:
```
# Imports
#!pip install -q git+https://github.com/tensorflow/docs # run on google colab
import matplotlib.pyplot as plt
import tensorflow_docs as tfdocs   # For plting mae
import tensorflow_docs.plots
from sklearn.metrics import r2_score # For R^2
```

The TensorFlow documentation contains tools for plotting MAE as the training progresses through different epochs. This can be used to analyse the training efficiency.

[17]:
```
# Visualise accuracy of time
plotter = tfdocs.plots.HistoryPlotter(smoothing_std=2)
plotter.plot({'Basic': history}, metric = "mae")
#plt.ylim([0, .08])
plt.ylabel('Mean Abs Err')
plt.title('MAE as training progresses')
```

[17]:
```
Text(0.5, 1.0, 'MAE as training progresses')
```



It is clear from this graph how training progressed, after 15 epochs the validation set MAE began to increase, probably due to over fitting, so the training was stopped early.

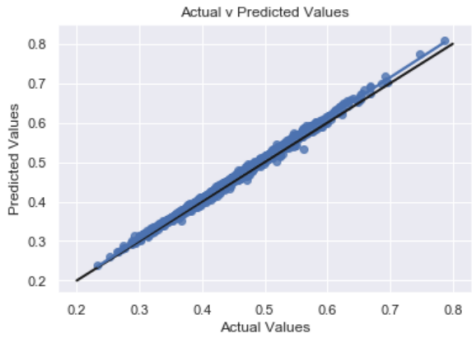
Predictions can be calculated from the test data and plotted against the actual values.

[18]:
```
# Generate predicted labels for test data
pred_test = model.predict(d_test).flatten()
print(pred_test)
```

```
[0.40752608 0.44995964 0.45261568 ... 0.41634804 0.40148783 0.48640496]
```

[19]:
```
# Plot predicted vs actual labels
x = np.linspace(0.2,0.8,100) # Adjust for graph

# Set up fingure
fig, ax = plt.subplots()
ax.plot(x,x,'k-',lw=2)
sns.regplot(x=l_test, y=pred_test, ax=ax)
ax.set_xlabel('Actual Values')
ax.set_ylabel('Predicted Values')
ax.set_title('Actual v Predicted Values')
```




```
[20]: # R Squared score
      r2 = r2_score(l_test,pred_test)
      print("R^2 score =",r2)

      R^2 score = 0.9830907786316301
```

This run was clearly very good. The graph shows a good fit and the R^2 score is over 0.98. The black line is a 1:1 graph, and the blue line is a simple linear regression over predicted and actual values.

As this model is accurate, we can find P(fail) from the test data as follows.

```
[29]: # P fail from test data
      fail = [x for x in pred_test if x<0]   #>= 15.59*(10**4)] # Gather failures
      pFail = len(fail)/len(d_test)                    # Count and ratio

      print("From the test data, P(fail) =", pFail)

      From the test data, P(fail) = 0.0
```

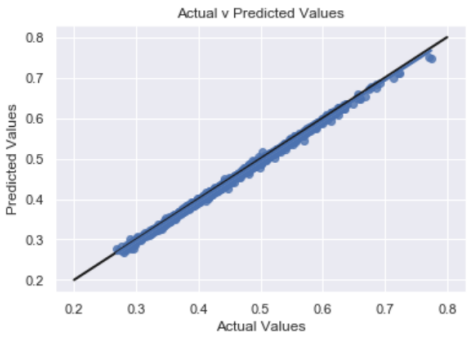
From this set of 2,000 samples we cannot find a failure.

Effects of ANN architecture

Can we find a better architecture? Consider this ANN.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 5)                 40
_________________________________________________________________
dense_5 (Dense)              (None, 25)                150
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 26
=================================================================
Total params: 216
Trainable params: 216
Non-trainable params: 0
_________________________________________________________________
```

The fitting took 33.7 second to run with a final MAE of 0.005. The fit is excellent with an R^2 of 0.993.



Making the hidden layers larger significantly increased the training times. Hidden layers of 25,25 took 43.8 seconds to run but hit an MAE of 0.003 and R^2 of 0.996.

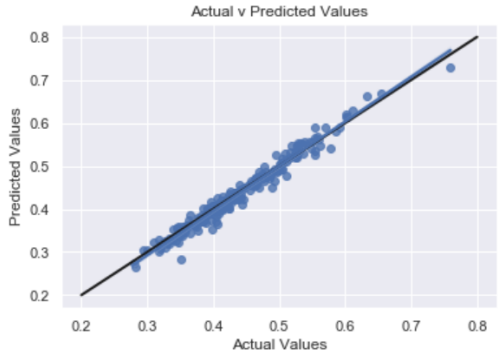
4 hidden layers of 10 nodes each took 30 seconds to run and improved the stats more with MAE = 0.002, and R2 = 0.998.

Run time is clearly proportional to number of trainable parameters while more complex models also giver better accuracy.

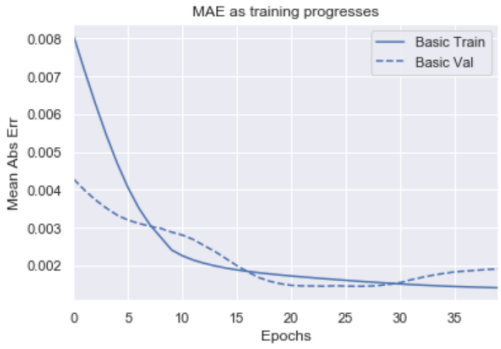
Effects of Sample Size

For this the architecture will be fixed at hidden layer of 25,25.

The sample size is set to 1000 and new data is generated. The ANN trained fast, in only 8.61 seconds, and the MAE = 0.011, and R^2 = 0.96. Information is sparse with this sample size as can be seen here.



With the sample size set to 100,000 training was particularly slow, taking 5 minuets 11 seconds. The MAE was 0.003 and the R^2 was 0.997. This isn't a significant improvement. The MAE graph reveals that there appears to be over fitting.



With these findings considered, a 25,25 neural network with a sample size of 10,000 will be trained for finding P(fail). A new run of this got a MAE of 0.002 and R^2 of 0.999.

Finding P(fail)

Predictions from a larger sample

To find P(fail) a crude Monte Carlo can be used on the model. A new sample can be generated after training an ANN with a sample size of 1,000,000.

[27]:
```
%%time
# Generate new larger sample
ssNew = 1000000 # Sample size

Xn = lhs(7, samples=ssNew) # 7D lhc between 0 and 1
for i in range(7):
    Xn[:,i] = st.norm(loc=means[i], scale=cov[i,i]).ppf(Xn[:,i]) # reverse cdf: [0,1] -> Normals

# Nomalise the data for model input
dsn = pd.DataFrame(Xn)
dsn_stats = dsn.describe().transpose()
def norm_new(x):
    return (x - dsn_stats['mean']) / dsn_stats['std']
dsn = norm_new(dsn)

dsn.head()
```

```
Wall time: 4.02 s
```

After using the predict method P(fail) can be found.

[28]:
```
%%time
# New sample predictions
pred_new = model.predict(dsn).flatten()
#pred_new
```

```
Wall time: 45.4 s
```

[32]:
```
# P(fail) from new predictions
failn = [x for x in pred_new if x > 155900]
pFailn = len(failn)/len(dsn)

print("Given this sample (size =",ssNew,"), P(fail) =", pFailn)
```

```
Given this sample (size = 1000000 ), P(fail) = 0.0
```

With a sample size of 1,000,000, P(fail) is still 0. It appears failure is clearly extremely unlikely.

Normal Approximation

To verify the answer, it's worth noting the KDE for the output can be approximated as a normal distribution especially further away from the mean where the skew has less effect. Using the mean and standard deviation of new predictions, P(fail) is estimated.

[33]:
```
# Checking with normal approximation

# Normal distro parameters from pred_new
sigm = np.std(pred_new)
mu = np.mean(pred_new)

# Display parameters
truemean = eqn(m1,m2,m3,m4,m5,m6,m7)
print("f(X) true mean =",truemean)
print("f(X) sample mean =", mu)
print("f(X) sample std =", sigm)

# Normalise
t = (155900-mu)/sigm

# p(z>t) from normal CDF
prob = 1 - st.norm.cdf(t)
print("Using normal approximation, P(f<0) =", prob)
```

```
f(X) true mean = 0.4333359950590159
f(X) sample mean = 0.4381743
f(X) sample std = 0.074424304
Using normal approximation, P(f<0) = 0.0
```

This method confirms P(fail) is too small to estimate.

Crude Monte Carlo

For a final check it is possible to do a crude Monte Carlo on the new data.

[36]:
```
%%time
# Direct MC on eqn outputs

# Generate Outputs
outn = []
for i in range(ssNew):
        y = eqn(*Xn[i])
outn.append(y)

# P fail
fail_outn = [x for x in outn if x > 155900]
pFail_outn = len(fail_outn)/len(outn)

print("Given this crude MC (size =",ssNew,"), P(fail) =", pFail_outn)
```

```
Given this crude MC (size = 1000000 ), P(fail) = 0.0
Wall time: 5.91 s
```

The probability is still 0.

Conclusions

The test statistic used in the normal approximation was 2094739.934458304. Trying a test statistic of 8 gives a probability of 6.661338147750939e-16 and any higher integer gives 0. As a result of this the probability is orders of magnitude away from being calculable.

As such failure should be considered essentially impossible.