**Computational Intelligence**

**Neural Network CA**



**March 2024**

Paul Price – B00106786

For this assignment we are tasked with creating a neural network using TensorFlow and Keras to approximate the following target function:

*f(x) = sin(x) \* sin(0.4 \* x) \* sin(2 \* x)*

The plot below shows the target function above plotted over the interval 0 – 10:

A graph of a graph

Description automatically generated

The target function takes a x value in the range 0 – 10 and generates a y value that produces the sin wave shape.

For this assignment we are provided with training and evaluation data, a train.ipynb file containing code for basic training steps and an evaluate.ipynb file that contains code to evaluate the network. The code we are given outputs the following graph:

A graph of a function

Description automatically generated

We are tasked with experimenting with the code we are given with the intention of achieving the following result:

A graph of a function

Description automatically generated

The first thing I did was ensure that the network used the Rectified Linear Unit (ReLU) activation function for any hidden layers and that the output layer node has a linear output. We are given sample code for this:  
**model.add(Dense(2, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))**

This line of code adds a fully connected layer to a neural network model with 2 neurons, ReLU activation function, and he\_uniform weight initialization, expecting one-dimensional input data. From this I determined that increasing the number in Dense(2) would increase the number of neurons in that layer. Adding more neurons could allow the neural network to learn more complex patterns in the data, as each neuron can learn to detect different features.[1] For this reason I chose to increase the number from 2 to 10 for initial testing. We are told in the brief that the network must have one input and one output so the ‘input\_dim’ number must stay the same.

Next, I looked at the loss function training algorithm. We had the choice of either SGD (Standard Gradient Descent) or ADAM(Adaptive Movement Estimation). SGD updates the model's parameters by following the gradient of the loss function whereas ADAM adjusts the step size for each parameter based on both the gradient and past gradients.[4] Rather than choosing one over the other I decided to test both and choose the one with the best results. I chose SGD for initial testing as sample code for it was already provided.

**opt = SGD(learning\_rate=0.1, momentum=0.1)**

Now that the loss function algorithm was selected I looked at the rest of the code, specifically the learning rate. The learning rate is a key hyperparameter in training neural networks. It determines the size of the steps taken during optimization.[4] A higher learning rate means larger steps, which can make the training process faster but has less accuracy. Alternatively, a lower learning rate means smaller steps, which can help in fine-tuning and accuracy but may be slower. Since our goal is to get the waves to match our primary concern is accuracy. For this reason I chose to decrease the learning rate. I also looked at the momentum. The momentum is only necessary for SGC and is used to control the algorithms direction of movement which can help smooth the line in the graph. I chose to keep it as is for now but may adjust later as needed once the graph is close to the desired result.

The last thing I looked at was the number of epochs. Epochs are the number of times the neural network is trained on the training data.[4] A higher number of epochs may take considerably longer but will yield significantly more accurate results. I chose to increase the epochs from 10 to 100 initially but may increase even further depending on results.

For my initial test run for improving the neural network the code I changed includes the following:  
**EPOCHS = 100**

**model.add(Dense(10, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))**

**opt = SGD(learning\_rate=0.05, momentum=0.1)**

This code gives the following result:  
A blue line with orange line

Description automatically generated

As we can see our code adjustments have yielded a slight improvement but is still far from what we are looking for. In an attempt to fix this I changed the loss function to ADAM and removed the momentum and it gave me the following result:

A graph with a line and orange line

Description automatically generated

As we can see this is still far from what we are looking for but is a definite improvement.

I then decide to significantly increase the number of neurons in the model from 10 to 50 which yielded the following results:

A graph with blue and orange lines

Description automatically generated

I expected this to be more accurate and came to the conclusion that the neural network should be trained on it further, for this reason I increased the number of epochs to 1000. The model did take significantly longer to load but did produce better results:  
A graph with blue and orange lines

Description automatically generated

I continued to try different parameters, most of which returned worse results. Each parameter change I tried with both SGD and ADAM. The first thing that I tried that shows a better result than the above graph was duplicating the layers. I did this by repeating the following line of code:

**model.add(Dense(50, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))**

Adding multiple identical layers can sometimes increase the model's capacity to learn complex patterns in the data.[1] Initially this did not lead to an improvement, but when I switched from ADAM to SGD the graph improved dramatically as can be seen in the graph below:

A graph with orange and blue lines

Description automatically generated

I decided to double the amount of layer again and see if that improved the graph further. This led to the following result:

A graph with orange lines

Description automatically generated

Now we can finally start to adjust the momentum to smoothen the line they’ll match better. I lowered the momentum little by little before ultimately settling on 0.025 which produced the following result:

A graph with orange lines

Description automatically generated

As we can see even though it is not perfect there is a slight improvement compared to the prior graph.

I can now run the evaluate.ipynb file which now gives us a result similar to the result we a attempting to achieve.

A screen shot of a graph

Description automatically generated

I then had to rerun this 10 times to get the average output.

**First run:**

Mean Squared Error (MSE): 0.000028

**A graph of a function

Description automatically generated**

**Second run:**

Mean Squared Error (MSE): 0.000061

**A graph of a graph

Description automatically generated**

**Third run:**

Mean Squared Error (MSE): 0.000063

A graph of a graph

Description automatically generated

**Fourth run:**

Mean Squared Error (MSE): 0.000031

**A graph of a graph

Description automatically generated**

**Fifth run:**

Mean Squared Error (MSE): 0.000043

**A graph of a graph

Description automatically generated**

**Sixth run:**

Mean Squared Error (MSE): 0.000742

**A graph of a function

Description automatically generated**

**Seventh run:**

Mean Squared Error (MSE): 0.000032

**A graph of a graph

Description automatically generated**

**Eighth run:**

Mean Squared Error (MSE): 0.000072

**A graph of a graph

Description automatically generated**

**Ninth run:**

Mean Squared Error (MSE): 0.000037

**A graph of a function

Description automatically generated**

**Tenth run:**

Mean Squared Error (MSE): 0.000071

A graph of a function

Description automatically generated

As we can see from the above outputs our parameters work very well with each run closely matching the desired output with the exception of run number 6 which goes slightly off track. The average Mean Squared Error (MSE) for the above 10 runs is 0.000118 which we get by adding all the MSEs together and dividing by 10.

**Final changed parameters:**

*EPOCHS = 1000*

*model.add(Dense(50, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))*

*model.add(Dense(50, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))*

*model.add(Dense(50, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))*

*model.add(Dense(50, input\_dim=1, activation='relu', kernel\_initializer='he\_uniform'))*

*opt = SGD(learning\_rate=0.05, momentum=0.025)*

## Question 1

### What makes this problem different to a classification problem?

In a classification problem, the goal is to categorize data into distinct classes or groups based on input features whereas this problem involves predicting a continuous numerical value based on input features. This fact makes this problem more likely a regression problem. The output variable is continuous, meaning it can take any real value within a certain range and the model is trained using loss functions that measure the difference between the predicted and actual values, such as mean squared error (MSE). In conclusion, the key difference between classification and regression problems is the output variable, the output variable is categorical for classification and continuous for regression.[3]

## Question 2

### Why should we not use an activation function such as the sigmoid function on the hidden layers?

When activation functions such as sigmoid is used on hidden layers it can cause the gradients to get smaller as they travel through the layers. This makes it harder for the neural network to learn effectively. Functions like ReLU help prevent the gradients from getting smaller as they are passed through the layers.[2]

## Question 3

### Why do we not use an activation function on the output layer node?

We don’t use an activation function on the output layer node because we want the output of the neural network to directly represent the predicted continuous value without any transformation. In regression problems, the output can take any real value within a certain range, so applying an activation function like sigmoid, squashes the output to a specific range or converts it into probabilities and can give us incorrect prediction values. Not using an activation function allows the network to output unbounded real values directly. This ensures that the model can predict a wide range of numerical values accurately without any distortion introduced by the activation function.[2]

# References

1. Neurons and Neural Networks - [*https://www.baeldung.com/cs/neural-networks-neurons#:~:text=In%20the%20context%20of%20a,neurons%20signal%20to%20one%20another*](https://www.baeldung.com/cs/neural-networks-neurons#:~:text=In%20the%20context%20of%20a,neurons%20signal%20to%20one%20another)*.*
2. An introduction to ReLU and the Limitations of Sigmoid and Tanh Activation Functions - [*https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/*](https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/)
3. Regression vs Classification in Machine Learning - [*https://www.javatpoint.com/regression-vs-classification-in-machine-learning*](https://www.javatpoint.com/regression-vs-classification-in-machine-learning)
4. A comprehensive guide to optimizers in deep learning - [*https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/*](https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/)

# Appendix

## train.ipynb

# Import the necessary libraries

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

from tensorflow import keras

from keras.models import Sequential

from keras.layers import Dense

from keras.optimizers import SGD

from keras.callbacks import CSVLogger

from keras import backend

from numpy import asarray

from matplotlib import pyplot

import math

import pylab

import random

import numpy as np

import pandas as pd

import time

import csv

# Globals and Funcs

# Path to training file

TRAIN\_FILE = "data/train.csv"

# Path to evaluation file

EVAL\_FILE = "data/eval.csv"

# Path to model file - weights will be saved here

MODEL\_FILE = "data/sin-model"

# Model log file

LOG\_FILE = 'data/training-log.csv'

# Setup the figure size for plots

pylab.rcParams['figure.figsize'] = (13.0, 5.0)

# Define the lists that will hold the training data

train\_x = [] # x-axis values

train\_y = [] # y-axis values

# Define the lists that will hold the evaluation data

eval\_x = [] # x-axis values

eval\_y = [] # y-axis values

# Load the training data for comparison

with open(TRAIN\_FILE, 'r') as csvfile:

    input\_data = csv.reader(csvfile, delimiter=',')

    for row in input\_data:

        train\_x.append(float(row[0]))

        train\_y.append(float(row[1]))

# Load the evaluation data

with open(EVAL\_FILE, 'r') as csvfile:

    input\_data = csv.reader(csvfile, delimiter=',')

    for row in input\_data:

        eval\_x.append(float(row[0]))

        eval\_y.append(float(row[1]))

# Prepare the training data for presentation to the neural network

# Convert the lists to numpy arrays, helps with manipulation

eval\_x = asarray(eval\_x)

eval\_y = asarray(eval\_y)

# Print the min and max just for info

print("Min Max before scaling...")

print(eval\_x.min(), eval\_x.max(), eval\_y.min(), eval\_y.max())

# Reshape arrays into into rows and cols

# we need to do this to present info to the network

eval\_x = eval\_x.reshape((len(eval\_x), 1))

eval\_y = eval\_y.reshape((len(eval\_y), 1))

# Scale the training data to values between 0 and 1

# this helps the network to train and stops any one

# value from dominating the input

scale\_x = MinMaxScaler()

scale\_y = MinMaxScaler()

eval\_x = scale\_x.fit\_transform(eval\_x)

eval\_y = scale\_y.fit\_transform(eval\_y)

# Print the min and max just for info

print("Min Max after scaling...")

print(eval\_x.min(), eval\_x.max(), eval\_y.min(), eval\_y.max())

# Load the previously trained model from disk

model = keras.models.load\_model(MODEL\_FILE)

# Evaluate the model with the random evaluation data generated above

# pred\_y is the predicated y values for our randomly generated x values (eval\_x) in the range 0 - 10

pred\_y = model.predict(eval\_x)

# report model error

print('Mean Squared Error (MSE): %.6f' % mean\_squared\_error(pred\_y, eval\_y))

# y positions for the predicated values

pred\_y\_final = scale\_y.inverse\_transform(pred\_y)

# x positions for the evaluation x values

eval\_x\_final = scale\_x.inverse\_transform(eval\_x)

# y positions for the evaluation y values

eval\_y\_final = scale\_y.inverse\_transform(eval\_y)

# Plot the predications so we can see how the network performs on the random evaluation data

# plot original training x and y data

pyplot.scatter(train\_x, train\_y, label='Training data')

# plot the correct target values

pyplot.scatter(eval\_x\_final, eval\_y\_final, label='Evaluation data')

# plot the predicated data x and y

pyplot.scatter(eval\_x\_final, pred\_y\_final, label='Predicted')

pyplot.title('math.sin(x) \* math.sin(0.4 \* x) \* math.sin(2 \* x)')

pyplot.xlabel('Input Variable (x)')

pyplot.ylabel('Output Variable (y)')

pyplot.legend()

pyplot.savefig("figs/sinwave\_result\_eval.jpg")

pyplot.show()

## evaluate.ipynb

# Import the necessary libraries

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

from tensorflow import keras

from keras.models import Sequential

from keras.layers import Dense

from keras.optimizers import SGD

from keras.callbacks import CSVLogger

from keras import backend

from numpy import asarray

from matplotlib import pyplot

import math

import pylab

import random

import numpy as np

import pandas as pd

import time

import csv

# Globals and Funcs

# Path to training file

TRAIN\_FILE = "data/train.csv"

# Path to evaluation file

EVAL\_FILE = "data/eval.csv"

# Path to model file - weights will be saved here

MODEL\_FILE = "data/sin-model"

# Model log file

LOG\_FILE = 'data/training-log.csv'

# Setup the figure size for plots

pylab.rcParams['figure.figsize'] = (13.0, 5.0)

# Define the lists that will hold the training data

train\_x = [] # x-axis values

train\_y = [] # y-axis values

# Define the lists that will hold the evaluation data

eval\_x = [] # x-axis values

eval\_y = [] # y-axis values

# Load the training data for comparison

with open(TRAIN\_FILE, 'r') as csvfile:

    input\_data = csv.reader(csvfile, delimiter=',')

    for row in input\_data:

        train\_x.append(float(row[0]))

        train\_y.append(float(row[1]))

# Load the evaluation data

with open(EVAL\_FILE, 'r') as csvfile:

    input\_data = csv.reader(csvfile, delimiter=',')

    for row in input\_data:

        eval\_x.append(float(row[0]))

        eval\_y.append(float(row[1]))

# Prepare the training data for presentation to the neural network

# Convert the lists to numpy arrays, helps with manipulation

eval\_x = asarray(eval\_x)

eval\_y = asarray(eval\_y)

# Print the min and max just for info

print("Min Max before scaling...")

print(eval\_x.min(), eval\_x.max(), eval\_y.min(), eval\_y.max())

# Reshape arrays into into rows and cols

# we need to do this to present info to the network

eval\_x = eval\_x.reshape((len(eval\_x), 1))

eval\_y = eval\_y.reshape((len(eval\_y), 1))

# Scale the training data to values between 0 and 1

# this helps the network to train and stops any one

# value from dominating the input

scale\_x = MinMaxScaler()

scale\_y = MinMaxScaler()

eval\_x = scale\_x.fit\_transform(eval\_x)

eval\_y = scale\_y.fit\_transform(eval\_y)

# Print the min and max just for info

print("Min Max after scaling...")

print(eval\_x.min(), eval\_x.max(), eval\_y.min(), eval\_y.max())

# Load the previously trained model from disk

model = keras.models.load\_model(MODEL\_FILE)

# Evaluate the model with the random evaluation data generated above

# pred\_y is the predicated y values for our randomly generated x values (eval\_x) in the range 0 - 10

pred\_y = model.predict(eval\_x)

# report model error

print('Mean Squared Error (MSE): %.6f' % mean\_squared\_error(pred\_y, eval\_y))

# y positions for the predicated values

pred\_y\_final = scale\_y.inverse\_transform(pred\_y)

# x positions for the evaluation x values

eval\_x\_final = scale\_x.inverse\_transform(eval\_x)

# y positions for the evaluation y values

eval\_y\_final = scale\_y.inverse\_transform(eval\_y)

# Plot the predications so we can see how the network performs on the random evaluation data

# plot original training x and y data

pyplot.scatter(train\_x, train\_y, label='Training data')

# plot the correct target values

pyplot.scatter(eval\_x\_final, eval\_y\_final, label='Evaluation data')

# plot the predicated data x and y

pyplot.scatter(eval\_x\_final, pred\_y\_final, label='Predicted')

pyplot.title('math.sin(x) \* math.sin(0.4 \* x) \* math.sin(2 \* x)')

pyplot.xlabel('Input Variable (x)')

pyplot.ylabel('Output Variable (y)')

pyplot.legend()

pyplot.savefig("figs/sinwave\_result\_eval.jpg")

pyplot.show()