DETERMINANT APPROXIMATION WITH MULTI LAYER PERCEPTRON

JESUA EPEQUIN

1. Introduction

In [1], Hornik, Stinchcombe and White proved that multilayer feed-forward networks can compute any function at all. This *universality* theorem is even more striking considering that it also asserts that the approximation can accomplished with single layer intermediate between the input and the output neurons. In this short report we find an instance of this result: we approximate the determinant function.

2. The algorithm

First we are importing the Python packages needed. We use numpy for numerical processing and keras for setting the neural network. From keras, we are importing the Sequential model type and the Dense layer type.

```
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

Second, we initialize number of samples, size of the matrices and the range of their coefficients

```
samples = 100
matrixSize = 2
coeff = [0, 100]
```

The following step is to generate random matrices. According to the values above we need 100 matrices of size 2x2 with coefficients randomly chosen in the interval [0,100]. We use numpy at this step. Note that we

```
matrices = []
determinants = []
for i in range(samples):
    matrix = np.random.randint(coeff[0], high = coeff[1], size =
(matrixSize,matrixSize))
    matrices.append(matrix.reshape(matrixSize**2,))
```

```
    determinants.append(np.array(np.linalg.det(matrix)).reshape(1,))
matrices = np.array(matrices)
determinants = np.array(determinants)
```

We then select the number of hidden layers for the model and the number of neurons in each. In this case we will build a model with 1 hidden layer and 16 neurons

```
layers = 1
neurons = 16
```

It is now time to define the MLP: a sequential dense model with 1 hidden layer and 16 neurons. The input layer has 4 neurons (the elements of the matrix) and the output laye only 1 (the determinant of the matrix). Note that we are using the activation function $x \mapsto x^2$ for the hidden layer, and a linear activation function for the output layer. We will explain the reason in the following sections.

```
model = Sequential()
model.add(Dense(neurons, input_dim = matrixSize**2, activa-
tion=lambda x:x*x))
for i in range(layers-1):
    model.add(Dense(neurons, activation=lambda x:x*x))
model.add(Dense(1))
```

Next, we compile the model. We use mean squared loss and the efficient adam version of stochastic gradient descent to optimize the model.

```
model.compile(loss='mse', optimizer='adam')
```

We can now train our model on the data created above. We save the result in the *history* object, it stores the training metrics (loss) that we need later. Note that the split between the train and test sets (67% and 33% in this case) is done in this step. Also, we go over the training set 1000 times (epochs = 1000).

```
history = model.fit(matrices, determinants, epochs = 1000, batch_size
= 100, verbose = 0, validation_split = 0.33)
```

Finally, we print the square root of mean square error (RMSE) on the test set, and some of the parameters used in our model.

```
rmse = np.sqrt(history.history['val_loss'][-1])
print("'
Validation RMSE: {}
Number of layers: {}
Number of neurons: {}
Number of samples: {}
"'.format(rmse,layers,neurons,samples))
```

Running this algorithm, we obtain a value for the RMSE on the test set together with the other parameters.

Validation RMSE: 22.009432764504275
Number of layers: 1
Number of neurons: 16
Number of samples: 100

3. Analizing the metrics

The error obtained at the end of the previous section is considerably high. There are several ways we could try improve it. We could increase the number of samples, number of layers, number of neurons in each layer, or number of epochs. We could also change the activation function for some or the standard options: sigmoid, relu, or linear. In this section we will see the effect of performing some of these changes

For the number of layers neurons and samples as in the previous section, we can plot the loss on the training and test sets (these are stored in the *history* object) over the training epochs (set to 1000). We

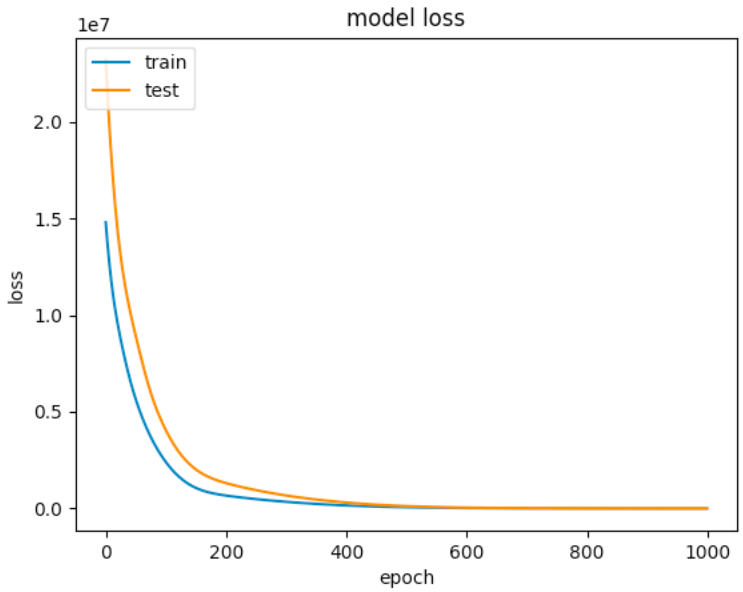

Figure 1. Error versus epochs

can see that the error curves flatten after epoch 400, the model does not learn much after this point. It is important to stress that both curves also meet at this point: the training error is smaller than the validation error for the first 400 epochs, which is reasonable.

We can now study the effect the number of samples in our dataset has on the loss functions. Having learned from Figure 1 we fix the number of epochs at 400, and leave the value for the other parameters untouched. Figure 2 shows the results. In this case we see that the

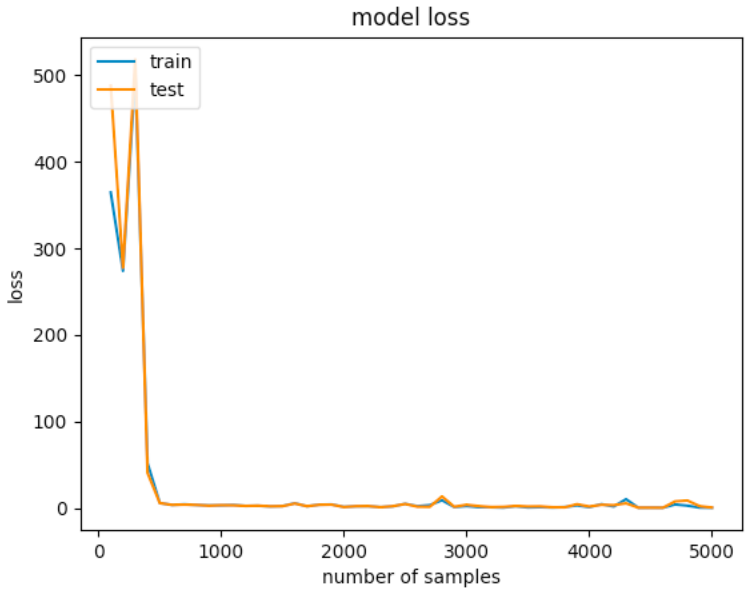

FIGURE 2. Error versus number of samples

error stabilizes after 500 samples. Beyond this number we can still get errors increasinly closer to zero (for instance, RMSE = 0.21 for 5000 samples), the convergence is however considerably slow.

The activation function also has a big impact on the loss value. In the following table we present the validation errors (RMSE) obtained for different activation functions (using 1000 samples). We realize that

Activationl	$x \mapsto x^2$	ReLU	sigmoid	tanh
RMSE	5.08	1054.70	3198.61	3095.85

our custom quadratic function vastly outperforms the other standard activations.

We can finally see how our model approximates determinants of matrices with coefficients outside the chosen interval $[0, 100]$. Results (using again 1000 samples) are show below. As we can appreciate. The error increases rapidly as we move away from the interval [0,100]. That is to be expected, since the training data was generated on this interval. It makes intuitive sense.

Interval	[0,100]	[100,200]	[200,300]	[300,400]	[400,500]
RMSE	3.24	52.20	178.92	378.35	650.70

4. Conclusion

We have verified that a neural network can approximate the determinant function with as few as 1 hidden layer. The approximation is considerably better when the number of epochs or samples is higher. This is intuitive: the bigger the dataset, the better the approximation.

Using our human intuition we chose the (customized) activation function $x \mapsto x^2$ for the hidden layer. Compared to other (standard) activation functions, the model obtained with this quadratic function does a much more accurate approximation. The reason for this lies in the known formula for the determinant. For instance, fo 2×2 matrices:

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

Allowing the activation function to be quadratic allows the model to obtain the multiplication between the elements of the matrix present in the formula above. This is not the case, for instance, if we use a activation function such as ReLU.

As a last word, even though in principle it is possible to approximate functions with only 1 hidden layer, some problems can better be solved by using multiple layers. For instance, in problems such as image recognition, using several layers allows the model to understand not just individual pixels, but also more complex concepts: edges, geometric shapes, and even multi-object scenes.

References

[1] Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural Netw. 2*, 5 (July 1989), 359366.