Option #1: Hand-Made Shallow ANN in Python

Scott Miner

Colorado State University – Global Campus

Abstract

This paper presents the development and implementation of a basic 2-layer Artificial Neural Network (ANN) that employs static backpropagation for predicting the next number in a given sequence. The ANN is constructed using the Python NumPy library and is adapted from existing code. The network's efficacy is evaluated using mean squared error (MSE) over 10,000 training epochs, demonstrating its ability to learn patterns in various sequences. Future enhancements, including the incorporation of bias and alternative activation functions, are proposed to improve the model's performance and generalizability.

```python
import math
import numpy as np
from sklearn.preprocessing import MinMaxScaler

PCT_TRAINING = 80
EPOCHS = 1000

class neural_network(object):
  def __init__(self):
    #parameters
    self.inputLayerSize = 2
    self.outputLayerSize = 1
    self.hiddenLayerSize = 3

    #weights
    # weight matrix of dimension (size of layer l, size of layer l-1)

    # weight matrix from input to hidden layer
    self.W1 = np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
    # weight matrix from hidden to output layer
    self.W2 = np.random.randn(self.hiddenLayerSize, self.outputLayerSize)
```

Figure 1. Constructor for the neural_network class

```python
# sequence is the user input
seq = [int(x) for x in input('Input a series of numbers separated by spaces (Press enter when done): ').split()]

# create record id
int_id = list(range(len(seq)))

# create matrix
sequence_of_integers = np.column_stack((int_id, seq))
# slice matrix on second value
follow_up_sequence = sequence_of_integers[1:,1]
follow_up_sequence = np.array(follow_up_sequence)
follow_up_sequence = follow_up_sequence.reshape(follow_up_sequence.shape[0],-1)

# all x and y
x_all_orig = np.array((sequence_of_integers), dtype=float)
y_orig = np.array((follow_up_sequence), dtype=float) # output

# scale all x and y
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
x_all_trans = scaler_x.fit_transform(x_all_orig)
y_trans = scaler_y.fit_transform(y_orig)

# split data
num_rows = np.shape(x_all_trans)[0]
splitPoint = math.trunc(num_rows * (PCT_TRAINING / 100))

# create training and validation data sets using split point
X_train = np.split(x_all_trans, [splitPoint])[0]
x_validation = np.split(x_all_trans, [splitPoint])[1]
y_to_pass_to_train_function = y_trans[:splitPoint,:]
```

Figure 2. Python code to create training and validation datasets from user input

```python
for i in range(EPOCHS): # trains the nn
  print("# " + str(i) + "\n")
  print("Training Data Input: \n" + str(scaler_x.inverse_transform(X_train)))
  print("Training Data Output: \n" + str(scaler_y.inverse_transform(y_to_pass_to_train_function)))
  print("Training Data Predicted Output: \n" + str(scaler_y.inverse_transform(nn.forward(X_train))))

  # mean squared error
  print("Loss: \n" + str(np.mean(np.square(y_to_pass_to_train_function - nn.forward(X_train)))))
  print("\n")
  nn.train(X_train, y_to_pass_to_train_function)

nn.saveWeights()
nn.predict()
```

Figure 3. The *for* loop that trains the ANN

```python
def forward(self, X):
  # forward propagation through our network
  # dot product of X (input) and first set of weights and bias
  self.z = np.dot(X, self.W1)
  # activation function
  self.z2 = self.sigmoid(self.z)
  # dot product of hidden layer (z2) and second set of weights and bias
  self.z3 = np.dot(self.z2, self.W2)
  # final activation function
  o = self.sigmoid(self.z3)
  return o

def backward(self, X, y, o):
  # backward propagate through the network
  # error in output
  self.o_error = y - o
  # applying derivative of sigmoid to error
  self.o_delta = self.o_error*self.sigmoidPrime(o)

  self.z2_error = self.o_delta.dot(self.W2.T) # z2 error: how much our hidden layer weights contributed to output error
  self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2 error

  self.W1 += X.T.dot(self.z2_delta) # adjusting first set (input --> hidden) weights
  self.W2 += self.z2.T.dot(self.o_delta) # adjusting second set (hidden --> output) weights

def sigmoid(self, s):
  # activation function
  return 1/(1+np.exp(-s))

def sigmoidPrime(self, s):
  #derivative of sigmoid
  return s * (1 - s)

def train(self, X, y):
  o = self.forward(X)
  self.backward(X, y, o)

def predict(self):
  print("Predicted data based on trained weights: ")
  print('Validation Data Input: \n' + str(scaler_x.inverse_transform(x_validation)))
  #print("Input (scaled): \n" + str(x_validation))
  print("Validation Data Output: \n" + str(scaler_y.inverse_transform(self.forward(x_validation))))
```

Figure 4. Additional functions of the neural_network class

```
DEBUG CONSOLE    AZURE    TERMINAL    PROBLEMS    GITLENS    SQL SERVER    SQL CONSOLE    OUTPUT

*************************************************************************************
*                                                                                   *
*   ARTIFICIAL NEURAL NETWORK (ANN) FOR PREDICTING THE NEXT NUMBER IN A SEQUENCE    *
*                                                                                   *
*************************************************************************************
*                                                                                   *
* Source:                                                                           *
*    The program is inspired by and modifies the source code available at:          *
*                https://enlight.nyc/projects/neural-network                        *
*                                                                                   *
*************************************************************************************
* User Input:                                                                       *
*    Enter a sequence of numbers separated by spaces (e.g., 6 8 10 12 14 16 18)     *
*    and press 'Enter'. The ANN will attempt to predict the next number in the      *
*    sequence based on the input.                                                   *
*                                                                                   *
* Default Parameters:                                                               *
*    Training Epochs     = 10,000                                                   *
*    Training Data       = 80%                                                      *
*    Validation Data     = 20%                                                      *
*                                                                                   *
* ANN Structure:                                                                    *
*    1. Input Layer:   2 Neurons (X, N)                                             *
*    2. Hidden Layer:  3 Neurons (Configurable)                                     *
*    3. Output Layer:  1 Neuron  (Prediction)                                       *
*                                                                                   *
* Activation Function: Sigmoid                                                      *
* Loss Function:       Mean Square Error (MSE)                                      *
*                                                                                   *
*************************************************************************************
* Overview:                                                                         *
*    This program uses an Artificial Neural Network (ANN) to predict the next       *
*    number in a sequence based on the user's input. The ANN consists of three      *
*    layers: an input layer, a hidden layer, and an output layer. The input layer   *
*    has 2 neurons, the hidden layer has a default of 3 neurons (configurable), and *
*    the output layer has 1 neuron that represents the prediction.                  *
*                                                                                   *
*    During the training process, the ANN performs forward and backward             *
*    propagation to adjust its weights based on the input data. It uses the         *
*    Sigmoid activation function for hidden layers during forward-propagation and   *
*    its derivative during backward-propagation.                                    *
*                                                                                   *
*    The Mean Square Error (MSE) loss function is used to evaluate the performance  *
*    of the ANN during training. After a specified number of training epochs        *
*    (iterations), the ANN uses the trained weights to make predictions based on    *
*    validation data.                                                               *
*                                                                                   *
*************************************************************************************

Input a series of numbers separated by spaces (Press enter when done):
```

Figure 5. Program instructions and user interface

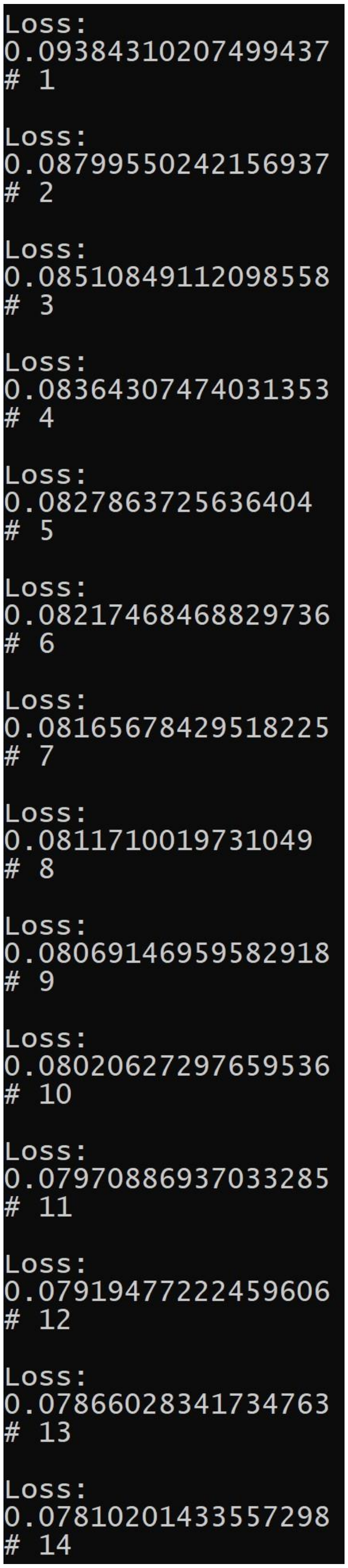

Figure 6. Loss function decreasing over initial training epochs

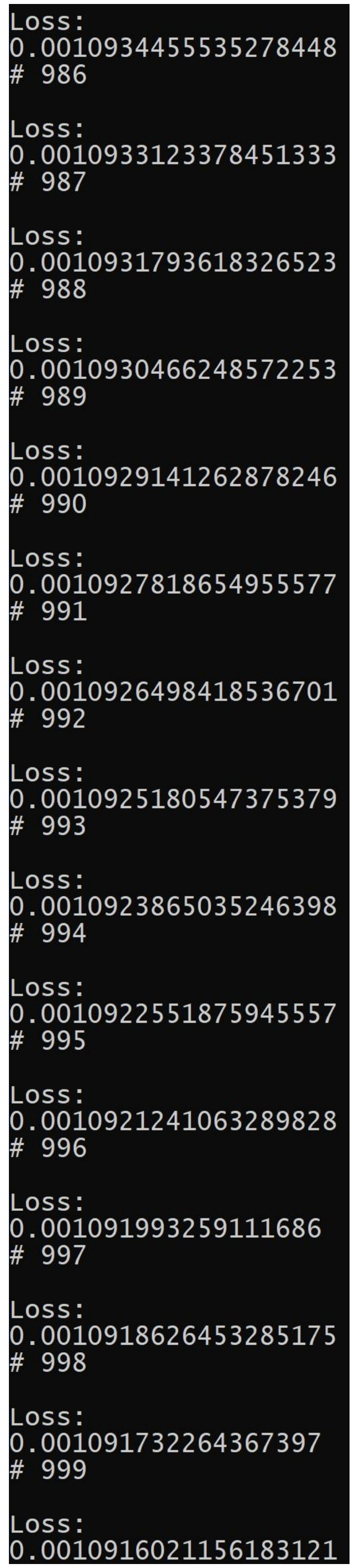

Figure 7. Loss function decreasing over final training epochs

```
# 9999

Training Data Input:
[[0. 1.]
 [1. 2.]
 [2. 3.]
 [3. 4.]
 [4. 5.]
 [5. 6.]
 [6. 7.]
 [7. 8.]]
Training Data Output:
[[2.]
 [3.]
 [4.]
 [5.]
 [6.]
 [7.]
 [8.]
 [9.]]
Training Data Predicted Output:
[[2.49768912]
 [2.96462118]
 [3.74540422]
 [4.84363453]
 [6.08969793]
 [7.22899977]
 [8.10503762]
 [8.70817601]]
Loss:
0.0009665938417343564
Predicted data based on trained weights:
Validation Data Input:
[[ 8.  9.]
 [ 9. 10.]]
Validation Data Output:
[[9.10158918]
 [9.35438224]]
```

Figure 8. Results after training the ANN over 10,000 epochs on the sequence from 1 to 10 by 1

```
# 9999

Training Data Input:
[[  0. 115.]
 [  1. 110.]
 [  2. 105.]
 [  3. 100.]
 [  4.  95.]
 [  5.  90.]
 [  6.  85.]
 [  7.  80.]]
Training Data Output:
[[110.]
 [105.]
 [100.]
 [ 95.]
 [ 90.]
 [ 85.]
 [ 80.]
 [ 75.]]
Training Data Predicted Output:
[[107.89662545]
 [105.27941979]
 [100.97666987]
 [ 95.45941631]
 [ 89.63240384]
 [ 84.22552793]
 [ 79.62682534]
 [ 75.9661034 ]]
Loss:
0.00046140471201572026
Predicted data based on trained weights:
Validation Data Input:
[[ 8. 75.]
 [ 9. 70.]
 [10. 65.]]
Validation Data Output:
[[73.20291501]
 [71.19870792]
 [69.78128874]]
```

Figure 9. Results after training the ANN over 10,000 epochs on the sequence from 115 to 65 by 5

```
# 9999

Training Data Input:
[[  0.   0.]
 [  1.   1.]
 [  2.   1.]
 [  3.   2.]
 [  4.   3.]
 [  5.   5.]
 [  6.   8.]
 [  7.  13.]
 [  8.  21.]
 [  9.  34.]
 [ 10.  55.]
 [ 11.  89.]
 [ 12. 144.]
 [ 13. 233.]
 [ 14. 377.]
 [ 15. 610.]]
Training Data Output:
[[  1.]
 [  1.]
 [  2.]
 [  3.]
 [  5.]
 [  8.]
 [ 13.]
 [ 21.]
 [ 34.]
 [ 55.]
 [ 89.]
 [144.]
 [233.]
 [377.]
 [610.]
 [987.]]
Training Data Predicted Output:
[[  1.33851036]
 [  1.66400069]
 [  2.28891918]
 [  3.47680738]
 [  5.6660345 ]
 [  9.59723615]
 [ 16.4124536 ]
 [ 27.83931072]
 [ 46.32630551]
 [ 75.28246154]
 [119.33874514]
 [184.89109554]
 [280.92391891]
 [420.0960474 ]
 [618.66119726]
 [890.9542758 ]]
Loss:
5.963651268445116e-05
Predicted data based on trained weights:
Validation Data Input:
[[  16.  987.]
 [  17. 1597.]
 [  18. 2584.]
 [  19. 4181.]]
Validation Data Predicted Output:
[[1231.30462006]
 [1588.52499736]
 [1874.29145802]
 [2031.07140088]]
Press any key to continue . . .
```

Figure 10. Results after training the ANN over 10,000 epochs on the Fibonacci sequence

Table of Contents

Option #1: Hand-Made Shallow ANN in Python

Artificial Neural Networks (ANNs) have been widely used in various domains, including finance, marketing, information systems, manufacturing, operations, and medical data classification tasks (Dreiseitl & Ohno-Machado, 2002; Sharda *et al.*, 2020). Gupta (2013) describes ANNs as "massively parallel computing systems consisting of an extremely large number of simple processors with many interconnections" (p. 24). This paper discusses the development and implementation of a basic 2-layer ANN that uses static backpropagation to predict the next number in a given sequence. The network is built using the Python NumPy library and is based on a modification of existing code provided by Shamdasani (2020).

ANN Structure and Components

The implemented ANN consists of three primary components: an input layer with two neurons, a hidden layer with three neurons, and an output layer with one neuron. Additionally, weights between the layers are included to facilitate forward propagation.

The input layer receives data in the form of a matrix and converts each number in the user-input sequence into a pair of *x* and *y* coordinates. The network uses forward-propagation to process the input data by multiplying the input layer by a series of randomly generated weights and applying the *sigmoid activation function* to each hidden layer. The output layer returns a prediction, which is then compared to the actual value. The error between the predicted and actual values is used to populate the loss function, which in turn is utilized to adjust the weights using *backpropagation*.

Data Processing and Training

The ANN preprocesses the input data by transforming it and splitting it into training (80%) and validation (20%) datasets. During the training phase, the network uses matrix multiplication to multiply the input layer by a series of randomly generated weights, applies the *sigmoid activation function* for every hidden layer, returns an output, calculates the *error* and *gradient descent* to populate the loss function, and uses the loss function to adjust the weights. The network is trained over a minimum of 1,000 epochs.

The program uses the *mean squared error* (MSE) to calculate the loss, which is the average of the squared differences between the predicted and actual values. A perfect value is 0.0, and the result is always positive regardless of the signs of the predicted and actual values (Brownlee, 2019).

Backpropagation

The ANN utilizes *backpropagation* to train the network. The *backward()* method calculates the error between the predicted and output values and computes the derivative of the sigmoid function on the predicted result, which is then multiplied by the error to create a *delta*. The ANN uses matrix multiplication on the delta matrix and weight matrix to determine how much the weights of the hidden layer contributed to the output error, producing a second delta. The model uses these deltas to adjust the weight matrices after each epoch. The larger the delta, the more the model adjusts the weights to minimize the error (Richmond, 2017).

This iterative process allows the ANN to fine-tune the weights between the layers, ultimately leading to more accurate predictions. By repeating this process for a user-specified number of epochs, the ANN can learn the underlying patterns in the input data and make better predictions for unseen data. As the network continues to train and the errors decrease, the model converges to an optimal set of weights that minimize the overall loss, as measured by the mean squared error (MSE).

Prediction and Validation

Once the model is trained, the weights it has learned to minimize the MSE are saved to two text files: *w1.txt* and *w2.txt*. These weights are then used to create predictions on the validation data. As described, the program splits the input data into two sets: 80% training and 20% validation. Therefore, the model's prediction for the final number in the user-input sequence will not occur until the ANN completes training and the program calls the *predict()* method using the validation dataset.

Results

The ANN's performance was evaluated on three different sequences of data: (a) the numbers 1 through 10 increasing by 1, (b) the numbers 115 through 65 decreasing by 5, and (c) the first 20 digits of the Fibonacci sequence. The results show that the ANN is comparatively better at predicting incremental and decremental patterns over the Fibonacci sequence.

Conclusion and Future Work

In conclusion, this paper discussed the development and implementation of a basic 2-layer ANN that uses static backpropagation to predict the next number in a given sequence. The network is built using the Python NumPy library and is based on a modification of existing code provided by Shamdasani (2020). The ANN's performance was analyzed using mean squared error (MSE) over 10,000 training epochs.

Future improvements to the model include adding the bias and incorporating additional activation functions, such as Rectified Linear Activation (ReLU) and Hyperbolic Tangent (Tanh) (Brownlee, 2021). These enhancements can help improve the network's performance and generalizability, potentially making it suitable for a wider range of sequence prediction tasks.

References

Brownlee, J. (2019, January 27). Loss and Loss Functions for Training Deep Learning Neural Networks. *Machine Learning Mastery*. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/

Brownlee, J. (2021, January 17). How to Choose an Activation Function for Deep Learning. *Machine Learning Mastery*. https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

Dreiseitl, S., & Ohno-Machado, L. (2002). Logistic regression and artificial neural network classification models: A methodology review. *Journal of Biomedical Informatics*, *35*(5), 352–359. https://doi.org/10.1016/S1532-0464(03)00034-0

Gupta, N. (2013). Artificial neural network. *Network and Complex Systems*, *3*(1), 24–28.

Richmond, A. (2017, January 31). *A Neural Network in Python, Part 1: Sigmoid function, gradient descent & backpropagation*. Alan Richmond. https://tuxar.uk/neural-network-python-part-1-sigmoid-function-gradient-descent-backpropagation/

Shamdasani, S. (2020, May 6). *Build a Neural Network with Python*. https://enlight.nyc/projects/neural-network

Sharda, R., Delen, D., & Turban, E. (2020). *Analytics, data science, & artificial intelligence* (Eleventh edition). Pearson.