Report 2

Authors: Raees Kabir ✉ , ChatGPT-4o

11 November 2024

CSB195

Objective

My aim is to create a simple neural network in the R to approximate the natural logarithmic function. This task was approached using a feedforward neural network with two hidden layers and a backpropagation learning algorithm.

Model

Architecture

The neural network consists of:

- **Input Layer:** One input node representing the input value `x` (the value for which the logarithm is calculated)
- **First Hidden Layer:** 100 neurons, applying the hyperbolic tangent (`tanh`) activation function.
- **Second Hidden Layer:** 50 neurons, also using the tanh activation function.
- **Output Layer:** One neuron that outputs the predicted value of `ln(x)`.

The architecture was chosen to balance between simplicity and the ability to capture the non-linear relationship between `x` and `ln(x)`. I also tested out the ReLU and sigmoid functions but the results were not as accurate.

Training Data

Training data was generated by creating a sequence of exponentially spaced values from 0 to 5 and taking the natural logarithm of these values This range ensures the model is exposed to both smaller and larger values, providing a well-rounded dataset for learning.

Loss Function

This model uses the squared error loss function, which computes the square of the difference between the predicted logarithmic value and the actual logarithmic value for each sample.

Training Process with Mini-Batches

Inspired by a technique I learned about called "stochastic gradient descent", I trained the model using mini-batches. The training data is split into smaller batches to improve the efficiency of the learning process.

- **Mini-Batch Size:** The mini-batch size was set to 10. This means that for each update, 10 random samples are selected from the training data, and the weights are updated based on the average error across these 10 samples.
- **Epochs:** The entire training dataset is passed through the network 1000 times.
- **Shuffling:** The data is shuffled at the beginning of each epoch to prevent the model from learning patterns based solely on the order of the data.

During each epoch, the mini-batches are processed sequentially, and the weights are updated after every batch. The loss for each batch is accumulated and used to track the model's performance over time.

Backpropagation and Learning

The model uses backpropagation with mini-batch gradient descent to minimize the mean squared error between the predicted and actual logarithmic values. The gradients are computed for each layer's weights and biases using the chain rule to identify the sensitivity of each weight and bias to the error, and the weights and biases are then updated iteratively.

A learning rate is implemented with a value of 0.0001 to smoothly locate optimal values. I had originally also implemented a decay rate to gradually reduce the learning rate over time. However, I did not see a significant change.

Observations

Throughout the training process, the loss is tracked and printed every epoch to monitor the model's progress

A summary of the tracked loss is shown below with set.seed(46)

```
Epoch: 1 Loss: 9.17441
Epoch: 2 Loss: 1.14322
Epoch: 3 Loss: 0.7547635
Epoch: 4 Loss: 0.5898221
Epoch: 5 Loss: 0.4801437
Epoch: 6 Loss: 0.4034436
Epoch: 7 Loss: 0.3485684
Epoch: 8 Loss: 0.3083093
Epoch: 9 Loss: 0.2789323
Epoch: 10 Loss: 0.2564788
Epoch: 20 Loss: 0.1656534
Epoch: 50 Loss: 0.09833049
Epoch: 100 Loss: 0.07014867
Epoch: 500 Loss: 0.03896615
```

I found it surprising that the model was able to adjust for the loss so quickly at the start.

Testing

```
TestValue PredictedLog  ActualLog
1      0.100   -2.4625673 -2.3025851
2      0.325   -1.0442774 -1.1239301
3      0.550   -0.6704466 -0.5978370
4      0.775   -0.2174289 -0.2548922
5      1.000   -0.0309614  0.0000000
6     10.000    2.2927510  2.3025851
7     32.500    3.4960606  3.4812401
8     55.000    3.9935105  4.0073332
9     77.500    4.3511229  4.3502779
10   100.000    4.6191883  4.6051702
```

The mean absolute error is calculated by

$$\frac{1}{n}\sum_{i=1}^{n}|\mathrm{Predicted}_i - \mathrm{Actual}_i|$$

Where:

- n is the number of data points.
- $\mathrm{Predicted}_i$ is the predicted value for the *i*-th data point.
- Actual_i is the actual value for the *i*-th data point.

From the above tested data set, the mean absolute error is 0.0434, which is quite impressive for such a basic model.

The model however struggles with values which are significantly different from its trained data such as very large and small numbers.
For example,

```
TestValue PredictedLog ActualLog
1       500     5.530542  6.214608
```

Where the percent error is approximately 11.04% calculated by

$$\left|\frac{Predicted - Actual}{Actual}\right| \times 100$$

Conclusion

The neural network successfully approximates the natural logarithm function, achieving a Mean Absolute Error (MAE) of 0.0434 on test data. The model performed well for values within the training range, but struggled with extreme values, showing a percent error of 11.04% for x = 500. While the model is effective for typical inputs, expanding the training data range could improve generalization to out-of-range values.

Appendix 1: R Code

```
# tocID <- "myScripts/Raees_Kabir_Report2_v1"
# Version: 1.0
# Date:    2024-11-13
# Author:  r.kabir@mail.utoronto.ca; ChatGPT-4o
#
===========================================================================
===

# Set random seed for reproducibility
set.seed(46)

# Define the network structure
inputSize <- 1    # One input (x value)
hiddenSize1 <- 100  # First hidden layer size
hiddenSize2 <- 50   # Second hidden layer size
outputSize <- 1     # One output (log(x))

# Initialize weights randomly
weightsInputHidden1 <- matrix(runif(inputSize * hiddenSize1, -1, 1), nrow =
hiddenSize1, ncol = inputSize)
biasHidden1 <- runif(hiddenSize1, -1, 1)
weightsHidden1Hidden2 <- matrix(runif(hiddenSize1 * hiddenSize2, -1, 1),
nrow = hiddenSize2, ncol = hiddenSize1)
biasHidden2 <- runif(hiddenSize2, -1, 1)
weightsHiddenOutput <- runif(hiddenSize2, -1, 1)
```

```
biasOutput <- runif(1, -1, 1)

# Activation function - tanh
tanh <- function(x) {
  (exp(x) - exp(-x)) / (exp(x) + exp(-x))
}

# Tanh derivative for backpropagation
tanh_derivative <- function(x) {
  1 - tanh(x)^2
}

# Forward pass function with batch normalization
forward <- function(x) {
  hiddenInput1 <- weightsInputHidden1 %*% x + biasHidden1
  hiddenOutput1 <- tanh(hiddenInput1)

  hiddenInput2 <- weightsHidden1Hidden2 %*% hiddenOutput1 + biasHidden2
  hiddenOutput2 <- tanh(hiddenInput2)

  output <- sum(hiddenOutput2 * weightsHiddenOutput) + biasOutput
  return(list(output = output, hiddenOutput1 = hiddenOutput1, hiddenOutput2
= hiddenOutput2))
}

# Learning rate settings for schedule
initialLearningRate <- 0.0001

# Backpropagation with mini-batch support and learning rate schedule
backpropagation <- function(batchX, batchY, learningRate) {
  gradientWeightsHiddenOutput <- rep(0, length(weightsHiddenOutput))
  gradientBiasOutput <- 0
  gradientWeightsHidden1Hidden2 <- matrix(0, nrow = hiddenSize2, ncol =
hiddenSize1)
  gradientBiasHidden2 <- rep(0, hiddenSize2)
  gradientWeightsInputHidden1 <- matrix(0, nrow = hiddenSize1, ncol =
inputSize)
  gradientBiasHidden1 <- rep(0, hiddenSize1)

  for (i in 1:ncol(batchX)) {
    x <- batchX[, i, drop = FALSE]
    y <- batchY[i]
```

```
    forwardPass <- forward(x)
    predicted <- forwardPass$output
    hiddenOutput1 <- forwardPass$hiddenOutput1
    hiddenOutput2 <- forwardPass$hiddenOutput2

    # Calculate the error
    error <- (predicted - y)^2

    # Gradients for output layer
    dOutput <- 2*(predicted - y)

    # Gradients for hidden layer 2
    dHidden2 <- dOutput * weightsHiddenOutput * (1 - hiddenOutput2^2)

    # Gradients for hidden layer 1
    dHidden1 <- t(weightsHidden1Hidden2) %*% dHidden2 * (1 -
hiddenOutput1^2)

    # Accumulate gradients
    gradientWeightsHiddenOutput <- gradientWeightsHiddenOutput + dOutput *
hiddenOutput2
    gradientBiasOutput <- gradientBiasOutput + dOutput
    gradientWeightsHidden1Hidden2 <- gradientWeightsHidden1Hidden2 +
dHidden2 %*% t(hiddenOutput1)
    gradientBiasHidden2 <- gradientBiasHidden2 + dHidden2
    gradientWeightsInputHidden1 <- gradientWeightsInputHidden1 + dHidden1
%*% t(x)
    gradientBiasHidden1 <- gradientBiasHidden1 + dHidden1
  }

  # Update weights and biases
  batchSize <- ncol(batchX)
  weightsHiddenOutput <<- weightsHiddenOutput - learningRate *
(gradientWeightsHiddenOutput / batchSize)
  biasOutput <<- biasOutput - learningRate * (gradientBiasOutput /
batchSize)
  weightsHidden1Hidden2 <<- weightsHidden1Hidden2 - learningRate *
(gradientWeightsHidden1Hidden2 / batchSize)
  biasHidden2 <<- biasHidden2 - learningRate * (gradientBiasHidden2 /
batchSize)
  weightsInputHidden1 <<- weightsInputHidden1 - learningRate *
(gradientWeightsInputHidden1 / batchSize)
  biasHidden1 <<- biasHidden1 - learningRate * (gradientBiasHidden1 /
```

```
batchSize)
}

# Generate training data with larger range and better coverage near zero
trainData <- exp(seq(-5, 5, length.out = 1000))  # Exponentially spaced
data with larger range
trainLabels <- log(trainData)  # Labels: natural log of input data

# Define mini-batch size
batchSize <- 10

# Training loop with decreasing learning rate
epochs <- 1000
for (epoch in 1:epochs) {
  # Update learning rate based on decay
  learningRate <- initialLearningRate

  totalLoss <- 0
  indices <- sample(length(trainData))  # Shuffle data indices for each
epoch

  # Process each mini-batch
  for (batchStart in seq(1, length(trainData), by = batchSize)) {
    batchIndices <- indices[batchStart:min(batchStart + batchSize - 1,
length(trainData))]
    batchX <- matrix(trainData[batchIndices], nrow = inputSize, ncol =
length(batchIndices))
    batchY <- trainLabels[batchIndices]

    backpropagation(batchX, batchY, learningRate)  # Update weights and
biases based on mini-batch

    # Compute loss for tracking
    for (i in 1:length(batchIndices)) {
      prediction <- forward(batchX[, i, drop = FALSE])$output
      totalLoss <- totalLoss + (prediction - batchY[i])^2  # Squared error
loss
    }
  }

  # Print progress every 10 epochs
  if (epoch %% 10 == 0) {
    cat("Epoch:", epoch, "Loss:", totalLoss / length(trainData), "\n")
```

```
  }
}

# Testing the network with a diverse range of test data
testData <- c(seq(0.1, 1, length.out = 5), seq(10, 100, length.out = 5))  #
Diverse test values

testPredictions <- sapply(testData, function(x) forward(matrix(x, nrow =
inputSize))$output)
actualLog <- log(testData)  # Actual log values

# Display results
data.frame(TestValue = testData, PredictedLog = testPredictions, ActualLog
= actualLog)
```

Appendix 2: ChatGPT Conversation

https://chatgpt.com/share/67344e59-30c4-8001-b5cc-cd6a15bd05f2
https://chatgpt.com/share/67344e22-45c8-8001-9bd1-404ab9d10f69
https://chatgpt.com/share/67344d62-5804-8003-ab1e-ceab373f6cb7

[END]