5.1 Neural Networks

Jonathan De Los Santos

2/26/2021

Contents

1 Artificial Neural Networks

1.1 Neural Network Overview

- Modeled on how our brains respond to sensory input
- Neural networks use a network of artificial "neurons" or **nodes** to solve a learning problem
- Example applications

- Speech and handwriting recognition
- Models of weather and climate patterns
- Smart device automation

1.2 Feedforward Neural Network

AKA Multilayer Perception - Connected layers where inputs influence each layers which then influence final output layer - Needs four components - Input data - Defined network architecture - Feedback mechanism - Model training approach - Consists of input, hidden, and output layers - Input layer: original features - Output: the final function

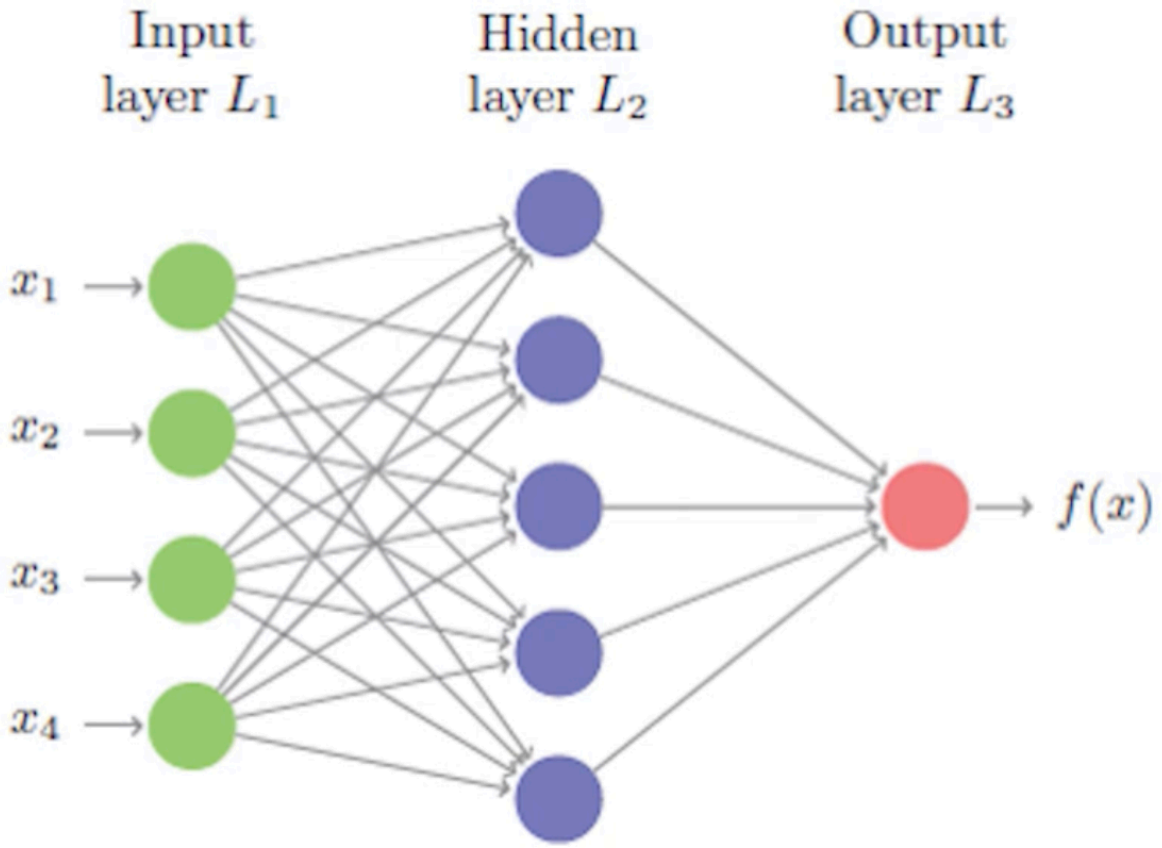


Figure 1: Feedforward Layers

1.3 Network Architecture

1.3.1 Layers and Nodes

- The building blocks of neural networks
  - Decides how complex the network will be

- Layers are considered density, meaning they are all fully connected to all other nodes
  - More layers = more opportunities to learn about new features
- Hidden layer
  - Layer between input and output layers
  - Processes the signal from input nodes prior to reaching output node
  - **Fully connected**: Every node in one layer is connected to every node in other layers, typical but not required
- Input nodes
  - Receive unprocessed signals from the input data
  - Each input node is responsible for processing a single feature
- Output nodes
  - Uses its own activation function to generate the final prediction
  - Driven by the type of modeling:
    - Regression: one node with continuous numerical output
    - Classification
      - Binary problem: one node with probability of success
      - Polynomial: number of nodes equal to number of classes predicted and the probability of each class ### Selecting Number of Layers/Nodes
  - As computational abilities increase, we can increase the number of hidden layers
    - **Deep** neural networks: multiple hidden layers
    - **Deep Learning**: training deep neural networks
  - Generally 2-5 hidden layers
    - Largely determined by number of functions in the data
    - Nodes per layer is usually $\leq$ the number of functions
  - More layers and nodes means more computation needed
  - In a feedforward network, the number of nodes in each layer (including output nodes) can be varied and multiple outcomes can be predicted
  - A greater number of neurons makes a model that closely reflects training data
    - May lead to overfitting
  - Use the fewest nodes that result in adequate performance

1.4 Recurrent Network

- Allows the signal to travel in both directions using loops
- Uses **delays** to increase the power
- Rarely used in practice

1.5 Artificial Neuron

- Defines the relationship between the input signal and output signal (y-variable)
- Each signal is weighted (w_x) according to their significance
- Signals are passed to an activation function which includes the weighted signals

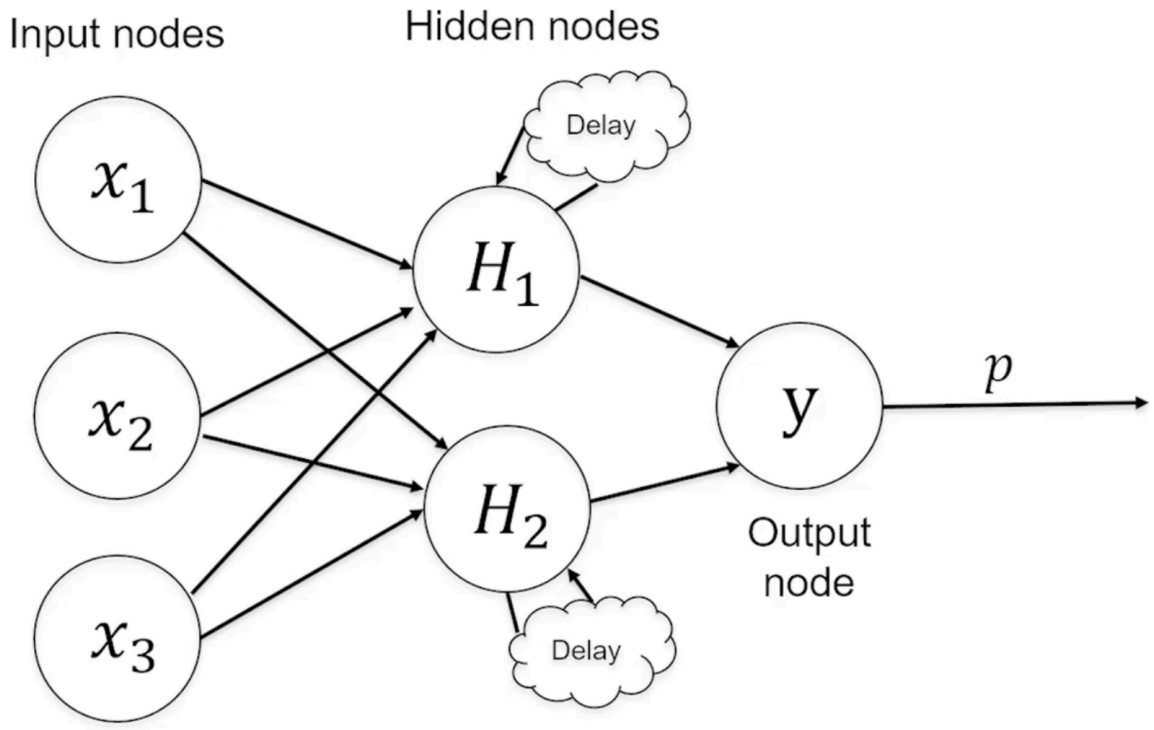


Figure 2: Recurrent Network

1.5.1 Activation Function

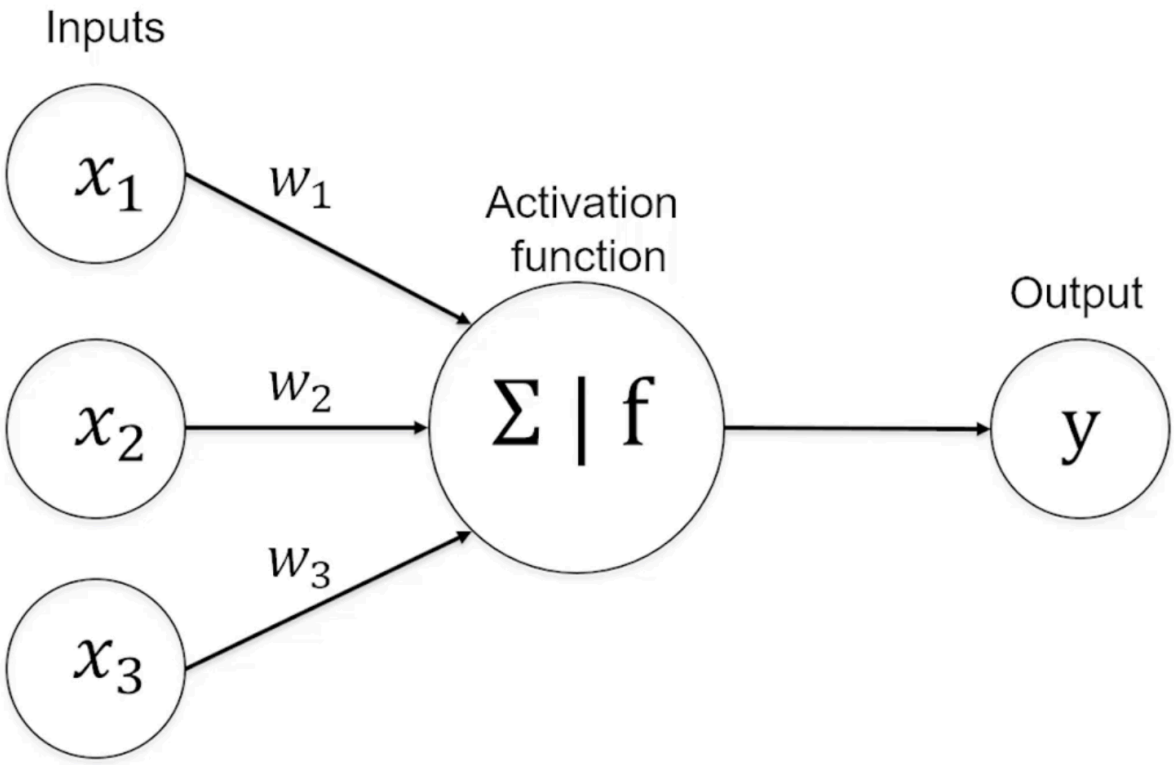
- The activation function is the mechanism by which the artificial neuron processes incoming information and passes it throughout the network
- Sums total input to determine whether it meets the threshold to fire a signal to the next layer
  - If yes: it passes on signal (activates)
  - If no: does not

1.5.1.1 Generalized Function

- The general function $f(x)$ used by all function types

  $y(x) = f(\sum_{i=1}^{n} w_i x_i)$

  - w: weight of signal x
  - x: the input signals
  - y: the output



Common Activation Functions

- Most common is rectified linear unit
  - Usually determining whether there is enough signal to fire (1) or not fire (0)

Linear $f(x) = x$

Rectified Linear Unit $f(x) = \begin{cases} 0, x < 0 \\ x, x > 0 \end{cases}$

Sigmoid $f(x) = \frac{1}{1+e^{-x}}$

1.5.2 Perception

In the rectified linear unit model below, y is determined by the weighted average of the input data

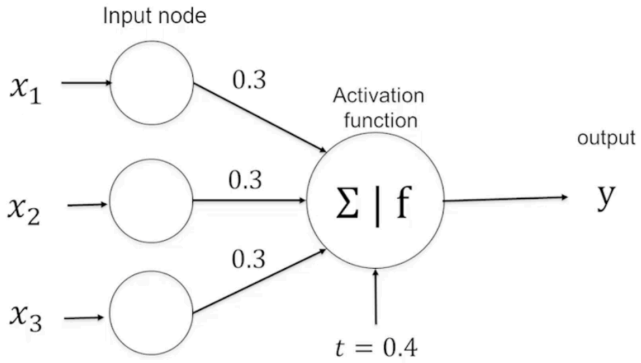
- In this case all weights are 0.3
- **Bias term**: adjusts the function output based on predetermined bias
  - Represented by t in the model below

2 Training Algorithm

3 `neuralnet` Package

These examples use a data set of various compositions of concrete. We will try to use some of these features to predict the `concrete$strength`

X_1	X_2	X_3	y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



$$\hat{y} = \begin{cases} 1, & if\ 0.3\,x_1 + 0.3\,x_2 + 0.3\,x_3 - 0.4\ > 0; \\ -1, & if\ 0.3\,x_1 + 0.3\,x_2 + 0.3\,x_3 - 0.4\ < 0 \end{cases}$$

Figure 3: Perception

3.1 Scale Data

`neuralnet` works best with normalized data - If normally distributed: standardize data - If not normally distributed: normalize data

Create a normalization function and apply it to our dataset. Running a summary on the original dataframe and the normalized version (`n.concrete`) we can see the normalization was successful.

```
concrete <- read.csv("Data Sets/5.1-Concrete.csv")

# Normalize data
normalize <- function(x){
  return ((x-min(x))/(max(x)-min(x)))
}

n.concrete <- as.data.frame(lapply(concrete, normalize))

summary(n.concrete$strength)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.2664  0.4001  0.4172  0.5457  1.0000
```

```
summary(concrete$strength)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.33   23.71   34.45   35.82   46.13   82.60
```

3.2 Partition data

75/25 training/testing split

```
# Partition
train <- n.concrete[1:773,]
test <- n.concrete[774:1030,]
```

3.3 The neuralnet Package

> Train neural networks using backpropagation, resilient backpropagation (RPROP) with (Riedmiller, 1994) or without weight backtracking

hidden argument:

```
- a vector of integers specifying the number of hidden neurons (vertices) in each layer
- The default of hidden layers is 1
```

3.4 Build the neuralnet Model

- Train the model with target variable "strength" and 8 other features
- Leave off `hidden` for now which defaults to 1

```
#install.packages("neuralnet")
library(neuralnet)
model <- neuralnet(strength~cement + slag + ash + water
                   + superplastic + coarseagg + fineagg + age,
                   data = train)
```

3.4.1 Plot neuralnet Model

The plot reveals:

- Weights of each feature
- A single hidden node
- Bias term weight (circle 1)
- SSE as "Error"
- The number of steps

```
plot(model)
```

3.5 Use compute() to Evaluate Model

Apparently a deprecated function but here we are.

`compute(nn object, covariate)` - `covariate` is a dataframe or matrix containing the variables (features) that had been used to train the neural network. - In this case we train 1:8 because feature 9 is our target variable - A summary of `model_results` (where we stored compute) provides - The neurons for each layer in the network - net.result which stores the predicted target value - Store the predicted value in an object (`predicted_strength`) - Looking at the net.result in predicted_strength we get a number of predicted values equal to our data set size - Because these are numeric predictions and not classifications, we can't use a confusion matrix to evaluate - Instead we run a correlation between it and the target value in the test data set - The resulting correlation is 0.71 which indicates a strong relationship

```
model_results <- compute(model, test[1:8])
summary(model_results)
```

```
##            Length Class  Mode
## neurons      2    -none- list
## net.result 257    -none- numeric
```

```
predicted_strength <- model_results$net.result
cor(predicted_strength, test$strength)
```

```
##           [,1]
## [1,] 0.7157509
```

3.6 Adding Hidden Layers to `neuralnet` Model

- Let's try adding hidden nodes to strengthen our model
- Pass in value to `hidden` argument, we'll do 5

```
model2 <- neuralnet(strength~cement + slag + ash + water
                    + superplastic + coarseagg + fineagg + age,
                    data = train, hidden = 5)
```

3.6.1 Plot Extra Layer Model

Notice the obvious increase in complexity, but also the lower error and larger number of steps

```
plot(model2)
```

3.6.2 Evaluate Extra Layer Model

- Notice increased correlation with the test data, from .71 to .75

```
model_results2 <- compute(model2, test[1:8])

predicted_strength2 <- model_results2$net.result
cor(predicted_strength2, test$strength)
```

```
##           [,1]
## [1,] 0.7593058
```

4 `nnet` Package

This example uses credit data from a German bank. Begin by importing and creating a function to transform variables to factors to use for later.

4.1 Data Prep

```
credit2 <- read.csv("Data Sets/5.0-GermanCredit.csv")
```

Notice the need to preprocess some variables to factors. To do this we'll create a function and pass in the categorical variables.

```
str(credit2)
```

```
## 'data.frame':    1000 obs. of  21 variables:
##  $ credit.rating                 : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ account.balance               : int  1 1 2 1 1 1 1 1 3 2 ...
##  $ credit.duration.months        : int  18 9 12 12 12 10 8 6 18 24 ...
##  $ previous.credit.payment.status: int  3 3 2 3 3 3 3 3 3 2 ...
##  $ credit.purpose                : int  2 4 4 4 4 4 4 4 3 3 ...
##  $ credit.amount                 : int  1049 2799 841 2122 2171 2241 3398 1361 1098 3758 ...
##  $ savings                       : int  1 1 2 1 1 1 1 1 1 3 ...
##  $ employment.duration           : int  1 2 3 2 2 1 3 1 1 1 ...
##  $ installment.rate              : int  4 2 2 3 4 1 1 2 4 1 ...
##  $ marital.status                : int  1 3 1 3 3 3 3 3 1 1 ...
##  $ guarantor                     : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ residence.duration            : int  4 2 4 2 4 3 4 4 4 4 ...
##  $ current.assets                : int  2 1 1 1 2 1 1 1 3 4 ...
##  $ age                           : int  21 36 23 39 38 48 39 40 65 23 ...
##  $ other.credits                 : int  2 2 2 2 1 2 2 2 2 2 ...
##  $ apartment.type                : int  1 1 1 1 2 1 2 2 2 1 ...
##  $ bank.credits                  : int  1 2 1 2 2 2 2 1 2 1 ...
##  $ occupation                    : int  3 3 2 2 2 2 2 2 1 1 ...
##  $ dependents                    : int  1 2 1 2 1 2 1 2 1 1 ...
##  $ telephone                     : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ foreign.worker                : int  1 1 1 2 2 2 2 2 1 1 ...
```

4.1.1 Factor Transformation Function

```
library(caret)

# Factor transformation function
to.Factor <- function (df, variables){
  for (i in variables){
    df[[i]] <- as.factor(df[[i]])
  }
  return(df)
}
```

Store the variables that need conversion in a variable.

```
var.factors <- c("credit.rating",
                 "account.balance",
                 "previous.credit.payment.status",
                 "credit.purpose",
                 "savings",
                 "employment.duration",
```

```
                "installment.rate",
                "marital.status",
                "guarantor",
                "residence.duration",
                "current.assets",
                "other.credits",
                "apartment.type",
                "bank.credits",
                "occupation",
                "dependents",
                "telephone",
                "foreign.worker"
                )
```

4.1.2 Data Partition

This time we create separate objects for test feature and class variables. For test features we leave out item 1, the target variable. For the class we include only the that variable

```
part <- createDataPartition(y = credit2$credit.rating, p = 0.6, list = FALSE)
train2 <- credit2[part,]
test2 <- credit2[-part,]

test.features <- -test2[,-1]
test.class <- -test2[,1]

#head(test.class)
```

4.1.3 Data Transformation

```
# Factor transformations
t.train <- to.Factor(df=train2, variables = var.factors)
t.test <- to.Factor(df=test2, variables = var.factors)

# Feature/class transformations
t.feature.test <- t.test[,-1]
t.class.test <- t.test[,1]
#head(t.feature.test)
```

4.2 Build `nnet` Model

4.2.1 `train()` Function

Part of `caret` library?

> Fit Predictive Models Over Different Tuning Parameters This function sets up a grid of tuning parameters for a number of classification and regression routines, fits each model and calculates a resampling based performance measure.

- `method` argument allows you to pass in custom model functions, in this case we will use `nnet` which needs to be installed and loaded from `nnet` package
- Attempts several iterations to convert the model
- Printing the model will show the various models attempted, the decay, accuracy, kappa, and a printout of the best model selected

```
print(nn.model)
```

```
## Neural Network
##
## 600 samples
##  20 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 600, 600, 600, 600, 600, 600, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##   1     0e+00  0.7046854  0.000000000
##   1     1e-04  0.7046854  0.000000000
##   1     1e-01  0.7016132  0.078741025
##   3     0e+00  0.7054193  0.003922770
##   3     1e-04  0.7046854  0.000000000
##   3     1e-01  0.6959896  0.213229182
##   5     0e+00  0.7046854  0.002308725
##   5     1e-04  0.7037292  0.024640143
##   5     1e-01  0.6990240  0.226302926
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 3 and decay = 0.
```

4.3 Evaluate `nnet` Model

Like before, we feed in our model and the test features. Because this is a classification (credit rating of 1 or 0) we can use a confusion matrix.

Our model performs with 73.25% accuracy

```
nn.prediction <- predict(nn.model, t.feature.test, type = "raw")

summary(nn.prediction)
```

```
##   0   1
##   0 400
```

```
confusionMatrix(nn.prediction, t.class.test)
```

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction   0   1
##          0   0   0
##          1 123 277
##
##                Accuracy : 0.6925
##                  95% CI : (0.6447, 0.7374)
##     No Information Rate : 0.6925
##     P-Value [Acc > NIR] : 0.5244
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.0000
##             Specificity : 1.0000
##          Pos Pred Value :    NaN
##          Neg Pred Value : 0.6925
##              Prevalence : 0.3075
##          Detection Rate : 0.0000
##    Detection Prevalence : 0.0000
##       Balanced Accuracy : 0.5000
##
##        'Positive' Class : 0
##
```