HUDM 6026: Computational Statistics

Artificial Neural Networks

References

- *The Elements of Statistical Learning* (2009), by T. Hastie, R. Tibshirani, and J. Friedman.

- *Machine Learning: A Probabilistic Perspective* (2012), by K. Murphy

Lesson Goals:

- Understand artificial neural networks and the back-propagation algorithm for both regression and classification problems.

Artificial Neural Networks

- The central idea of *artificial neural networks* (ANN) is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features.

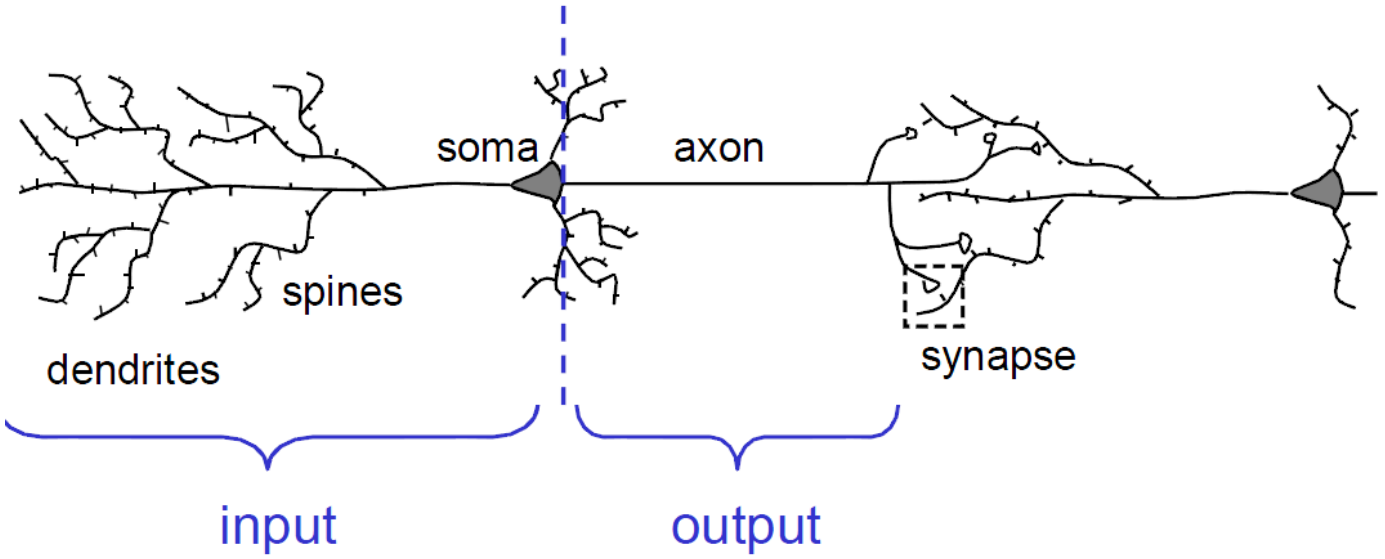
- This machine learning method originated as an algorithm trying to mimic neurons in the brain.

- The brain has extraordinary computational power to represent and interpret complex environments.

- ANN attempts to capture this mode of computation.

Artificial Neural Networks (cont.)

- Specifically, ANN is a formalism for representing functions, inspired from biological systems and composed of parallel computing units, each computing a simple function.

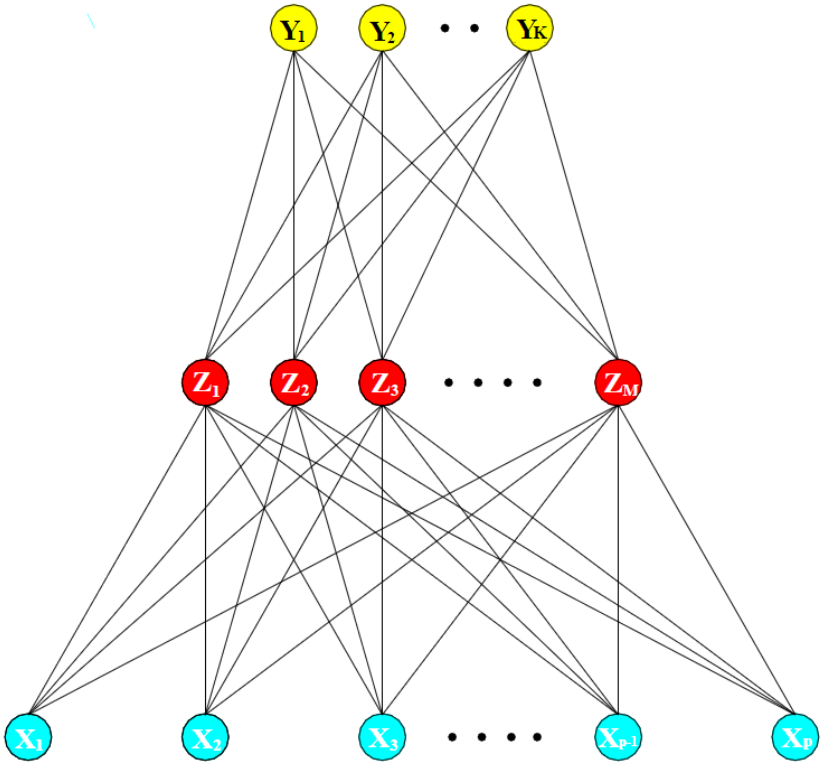


- Parts of the neuron: dendrites, soma (body), axon, synapses

Artificial Neural Networks (cont.)

- We will describe the most widely used ANN, the single layer perceptron (single hidden layer back-propagation network).

- A neural network is a two-stage regression or classification model, typically represented by a network diagram.



Artificial Neural Networks (cont.)

- In regression, typically $K = 1$ and there is only one output unit Y_1 at the top of the network diagram.
- These networks, however, can handle multiple quantitative responses in a seamless fashion (multivariate regression).
- For K-class classification, there are K units at the top of the network diagram, with the k^{th} unit modeling the probability of class k.
- There are K target measurements Y_k, $k = 1,\ldots, K$, each being coded as a 0 – 1 variable for the k^{th} class.

Artificial Neural Networks (cont.)

- Derived features Z_m are created from linear combinations of the inputs X, and then the target Y_k is modeled as a function of the linear combinations of the Z_m:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X),\ m = 1, \ldots, M,$$
$$T_k = \beta_{0k} + \beta_k^T Z,\ k = 1, \ldots, K,$$
$$f_k(X) = g_k(T),\ k = 1, \ldots, K,$$

where $Z = (Z_1, Z_2, \ldots, Z_M)$ and $T = (T_1, T_2, \ldots, T_K)$

- The *activation function* $\sigma(v)$ is typically the sigmoid (logistic) function: $\frac{1}{1+e^{-v}}$

Artificial Neural Networks (cont.)

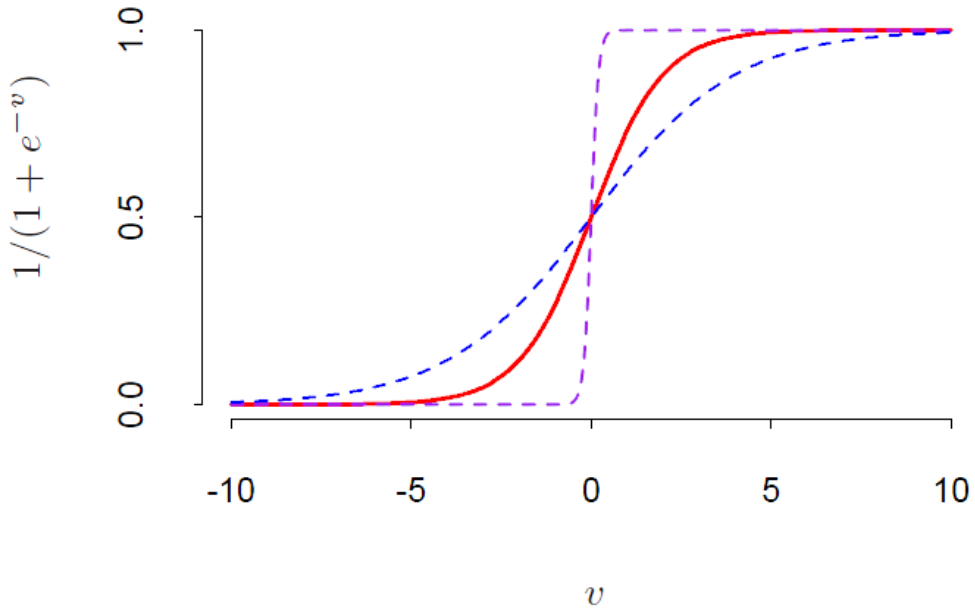


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1+\exp(-v))$ *(red curve), commonly used in the hidden layer of a neural network. Included are* $\sigma(sv)$ *for* $s = \frac{1}{2}$ *(blue curve) and* $s = 10$ *(purple curve). The scale parameter* s *controls the activation rate, and we can see that large* s *amounts to a hard activation at* $v = 0$*. Note that* $\sigma(s(v - v_0))$ *shifts the activation threshold from* 0 *to* v_0*.*

Artificial Neural Networks (cont.)

- Neural network diagrams are sometimes drawn with an additional *bias* unit feeding into every unit in the hidden and output layers.

- Thinking of the constant "1" as an additional input feature, this bias unit captures the intercepts α_{0m} and β_{0k} in the model.

- The output function $g_k(T)$ allows a final transformation of the vector of outputs T.

- For regression, we typically choose the identity function $g_k(T) = T_K$

Artificial Neural Networks (cont.)

- Early work in K-class classification also used the identity function, but this was later abandoned in favor of the *softmax* function:

$$g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$$

- Note that this is exactly the transformation used in the multilogit model, which produces positive estimates that sum to one.

- The units in the middle of the network, computing the derived features Z_m, are called *hidden units* because these values are not directly observed.

Artificial Neural Networks (cont.)

- We can think of the Z_m as a basis expansion of the original inputs X; the neural network is then a standard linear model, or linear multilogit model, using these transformations as inputs.

- However, the parameters of the basis functions are learned from the data, which is an important enhancement over the basis expansion techniques.

- Notice that if σ is the identity function, then the entire model collapses to a linear model in the inputs.

Artificial Neural Networks (cont.)

- Thus, a neural network can be thought of as a nonlinear generalization of the linear model, both for regression and classification.

- By introducing the nonlinear transformation σ, it greatly enlarges the class of linear models.

- In general, there can be more than one hidden layers.

- The neural network model has unknown parameters, often called *weights*.

Artificial Neural Networks (cont.)

- We seek values for the weights that make the model fit the training data well.

- We denote the complete set of weights by θ, which consists of:

$$\{\alpha_{0m}, \alpha_m;\ m = 1, 2, \ldots, M\}\ \ M(p+1) \text{ weights},$$
$$\{\beta_{0k}, \beta_k;\ k = 1, 2, \ldots, K\}\ \ K(M+1) \text{ weights}.$$

- For regression, we use RSS as our measure of fit (error function):

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2.$$

Artificial Neural Networks (cont.)

- For classification, we use squared error or cross-entropy (deviance):

$$R(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log f_k(x_i),$$

and the corresponding classifier $G(x) = \text{argmax}_k f_k(x)$.

- With the softmax activation function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood.

Artificial Neural Networks (cont.)

- Typically, we do not want the global minimizer $R(\theta)$, as this is likely to be an overfit solution.
- Instead, some regularization is need; this is achieved directly through a penalty term, or indirectly by early stopping.
- The generic approach to minimizing $R(\theta)$ is by gradient descent, called *back-propagation* in this setting.
- The gradient can be easily derived using the chain rule for differentiation.

Artificial Neural Networks (cont.)

- This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

- Here is back-propagation in detail for squared error loss.

- Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then:

$$
\begin{aligned}
R(\theta) &\equiv \sum_{i=1}^{N} R_i \\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2,
\end{aligned}
$$

Artificial Neural Networks (cont.)

- With the following derivatives:

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi},$$

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{i\ell}.$$

- Given these derivatives, a gradient descent update at the $(r+1)$st iteration has the form (where γ_r is the *learning rate*):

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}},$$

$$\alpha_{m\ell}^{(r+1)} = \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(r)}},$$

Artificial Neural Networks (cont.)

- We can simplify the derivatives to:

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi},$$
$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = s_{mi} x_{i\ell}.$$

- The quantities δ_{ki} and s_{mi} are "errors" from the current model at the output and hidden layer units, respectively. Note that the output layer errors $\delta_{ki} = (\hat{f}_k(x_i) - f_k(x_i))$

- From their definitions, these errors satisfy the *back-propagation equations*:

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki},$$

Artificial Neural Networks (cont.)

- Using this, the gradient descent updates can be implemented with a two-pass algorithm.
- In the *forward pass*, the current weights are fixed and the predicted values $\hat{f}_k(x_i)$ are computed.
- In the *backward pass*, the errors δ_{ki} are computed, and then back-propagated via the back-propagation equations to give the errors s_{mi}
- Both sets of errors are then used to compute the gradients for the updates.

Artificial Neural Networks (cont.)

- This two-pass procedure is what is known as back-propagation.
- It has also been called the *delta rule*.
- The computational components for cross-entropy have the same form as those for the sum of squares function.
- The advantage of back-propagation are its simple, local nature. In the back-propagation algorithm, each hidden unit passes and receives information only to and from units that share a connection.

Artificial Neural Networks (cont.)

- Usually starting values for weights are chosen to be random values near zero.
- Hence, the model starts out nearly linear and becomes nonlinear as the weights increase.
- Use of exact zero weights leads to zero derivatives and perfect symmetry, and the algorithm never moves.
- Starting instead with large weights often leads to poor solutions.

Artificial Neural Networks (cont.)

- Often neural networks have too many weights and will overfit the data at the global minimum of R.

- An explicit method for regularization is *weight decay*, which is analogous to ridge regression used for linear models.

- We add a penalty to the error function $R(\theta) + \lambda J(\theta)$, where:

$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{m\ell} \alpha_{m\ell}^2$$

and λ is a tuning parameter.

Artificial Neural Networks (cont.)

- Larger values of λ will tend to shrink the weights toward zero; typically cross-validation is used to estimate λ.

- Other forms for the penalty (such as *weight elimination*) are:

$$J(\theta) = \sum_{km} \frac{\beta_{km}^2}{1+\beta_{km}^2} + \sum_{m\ell} \frac{\alpha_{m\ell}^2}{1+\alpha_{m\ell}^2},$$

- Also, since the scaling of the inputs determines the effective scaling of the weights in the bottom layer, it can have a large effect on the quality of the final solution; thus, it is best to standardize all inputs.

Artificial Neural Networks (cont.)

- Generally, it is better to have too many hidden units than too few.
- With too few hidden units, the model might not have enough flexibility to capture the nonlinearities in the data.
- With too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization is used.
- Typically the number of hidden units is somewhere in the range of 5 to 100, with the number increasing with the number of inputs and number of training cases.

Artificial Neural Networks (cont.)

- The error function is nonconvex, possessing many local minima, so one must try a number of random starting configurations and the choose the solution giving the lowest error.

- Probably a better approach is to use the average predictions over the collection of networks as the final prediction.

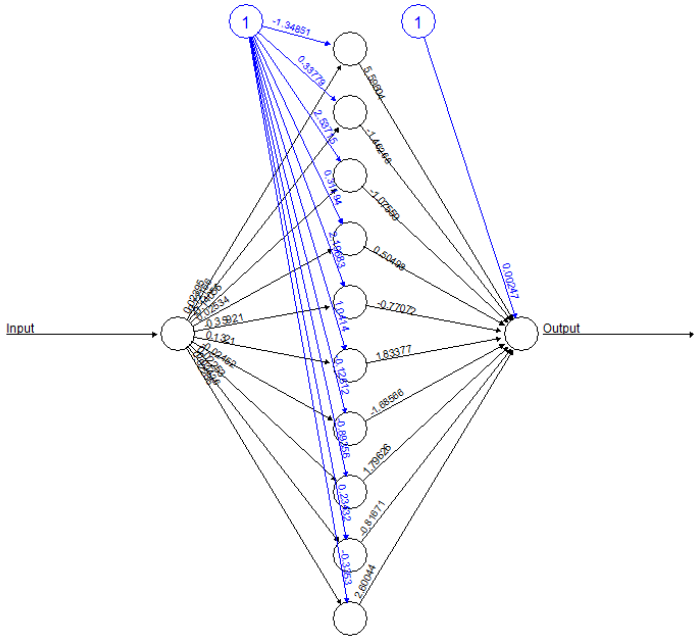
- Another approach is via *bagging*, which averages the predictions of networks training from randomly perturbed version of the training data.

ANN – Example 1 R Code

- In this first example, we use the R library “neuralnet” to train and build a neural network that is able to take a number and calculate the square root.
- The ANN will take a single input (the number you want square rooting) and produce a single output (the square root of the input).
- The ANN will contain 10 hidden neurons to be trained within each layer.
- The ANN does a reasonable job at finding the square root!

ANN – Example 1 R Code (cont.)

```r
## Example 1: Create a ANN to perform square rooting

library(neuralnet)  # For Neural Network

# Generate 50 random numbers uniformly distributed between 0 and 100
# and store them as a dataframe
traininginput <-  as.data.frame(runif(50, min = 0, max = 100))
trainingoutput <- sqrt(traininginput)

# Column bind the data into one variable
trainingdata <- cbind(traininginput,trainingoutput)
colnames(trainingdata) <- c("Input","Output")

# Train the neural network
# Going to have 10 hidden layers
# Threshold is a numeric value specifying the threshold for the partial
# derivatives of the error function as stopping criteria.
net.sqrt <- neuralnet(Output ~ Input, trainingdata, hidden = 10, threshold = 0.01)
print(net.sqrt)

# Plot the neural network
plot(net.sqrt)

# Test the neural network on some training data
testdata <- as.data.frame((1:10)^2) # generate some squared numbers
net.results <- compute(net.sqrt, testdata) # run them through the neural network

# Let's see what properties net.sqrt has
ls(net.results)

# Let's see the results
print(net.results$net.result)

# Let's display a better version of the results
cleanoutput <- cbind(testdata,sqrt(testdata), as.data.frame(net.results$net.result))
colnames(cleanoutput) <- c("Input","Expected Output","Neural Net Output")
print(cleanoutput)
```



	Input	Expected Output	Neural Net Output
1	1	1	1.027993874
2	4	2	2.005192003
3	9	3	3.001567330
4	16	4	3.986023794
5	25	5	5.003264027
6	36	6	6.011652248
7	49	7	6.990255178
8	64	8	8.000067508
9	81	9	9.016304667
10	100	10	9.960483101

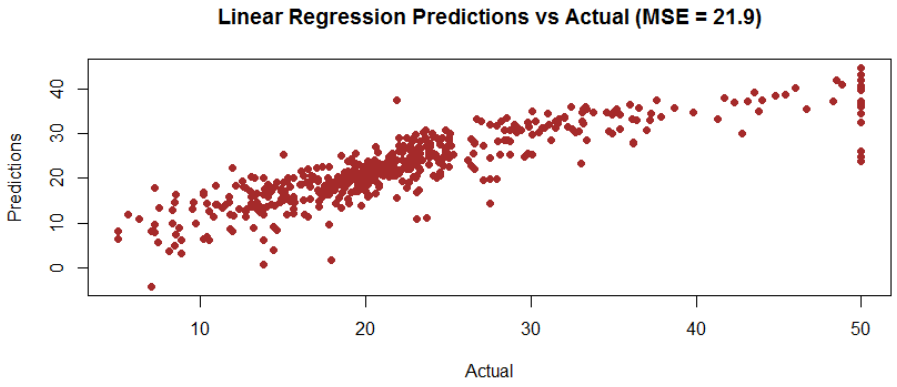
ANN – Example 2 R Code

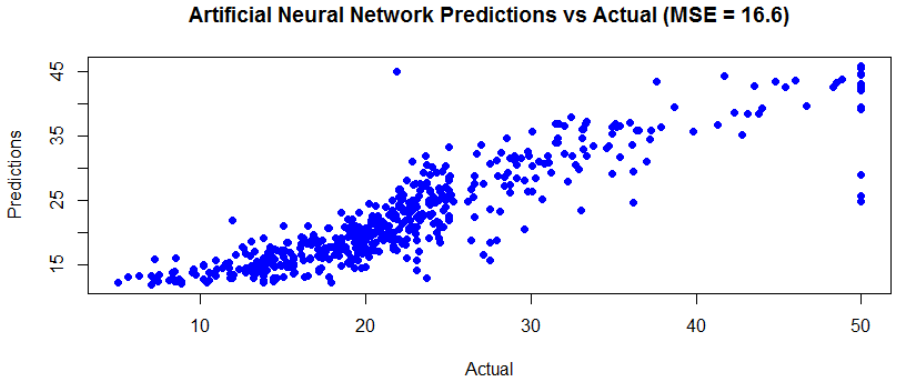
- In this second example, we use the R library “nnet” to train and build a neural network that is able to predict median value of owner-occupied homes.
- This example of regression with neural networks is compared with multiple linear regression.
- The data set is housing data for 506 census tracts of Boston from the 1970 census.
- We see that the ANN outperforms linear regression in terms of lower MSE.

ANN – Example 2 R Code (cont.)

```
# Example 2: Create a ANN for prediction

# We give a brief example of regression with neural networks and comparison with
# multivariate linear regression. The data set is housing data for 506 census tracts of
# Boston from the 1970 census. The goal is to predict median value of owner-occupied homes.

# Load the data and insepct the range (which is 1 - 50)
library(mlbench)
data(BostonHousing)
summary(BostonHousing$medv)

# Build the multiple linear regression model
lm.fit <- lm(medv ~ ., data = BostonHousing)
lm.predict <- predict(lm.fit)

# Calculate the MSE and plot
mean((lm.predict - BostonHousing$medv)^2) # MSE = 21.89483
par(mfrow = c(2,1))
plot(BostonHousing$medv, lm.predict, main = "Linear Regression Predictions vs Actual (MSE = 21.9)",
     xlab = "Actual", ylab = "Predictions", pch = 19, col = "brown")

# Build the feed-forward ANN (w/ one hidden layer)
library(nnet)        # For Neural Network
nnet.fit <- nnet(medv/50 ~ ., data=BostonHousing, size = 2) # scale inputs: divide by 50 to get 0-1 range
nnet.predict <- predict(nnet.fit)*50 # multiply 50 to restore original scale

# Calculate the MSE and plot
mean((nnet.predict - BostonHousing$medv)^2) # MSE = 16.56974
plot(BostonHousing$medv, nnet.predict, main = "Artificial Neural Network Predictions vs Actual (MSE = 16.6)",
     xlab = "Actual", ylab = "Predictions", pch = 19, col = "blue")
```

ANN – Example 2 R Code (cont.)





- Next, we use the function train() from the R library "caret" to optimize the ANN hyperparameters decay and size.
- This library also performs resampling to give a better estimate of the error.
- Note that we scale linear regression by the same value, so the error statistics are directly comparable.

ANN – Example 2 R Code (cont.)

```
# Next, we use the function train() from the package caret to optimize the ANN
# hyperparameters decay and size, Also, caret performs resampling to give a better
# estimate of the error. We scale linear regression by the same value, so the error
# statistics are directly comparable.

library(mlbench)
data(BostonHousing)
library(caret)

# Optimize the ANN hyperpameters and print the results
mygrid <- expand.grid(.decay = c(0.5, 0.1), .size = c(4, 5, 6))
nnet.fit2 <- train(medv/50 ~ ., data = BostonHousing, method = "nnet", maxit = 1000,
                   tuneGrid = mygrid, trace = FALSE)
print(nnet.fit2)

# Scale the linear regression and print the results
lm.fit2 <- train(medv/50 ~ ., data = BostonHousing, method = "lm")
print(lm.fit2)
```

```
Neural Network

506 samples
 13 predictors

No pre-processing
Resampling: Bootstrapped (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...

Resampling results across tuning parameters:

  decay  size  RMSE    Rsquared  RMSE SD  Rsquared SD
  0.1    4     0.083   0.8       0.00856  0.0378
  0.1    5     0.0827  0.801     0.00894  0.0416
  0.1    6     0.0818  0.806     0.00866  0.0382
  0.5    4     0.0921  0.762     0.00904  0.0405
  0.5    5     0.0908  0.77      0.00837  0.0386
  0.5    6     0.0901  0.771     0.00727  0.0333

RMSE was used to select the optimal model using  the smallest value.
The final values used for the model were size = 6 and decay = 0.1.
```

- Tuned ANN RMSE = 0.0818
- Linear Regression RMSE = 0.0985

```
Linear Regression

506 samples
 13 predictors

No pre-processing
Resampling: Bootstrapped (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...

Resampling results

  RMSE    Rsquared  RMSE SD  Rsquared SD
  0.0985  0.717     0.00911  0.0473
```

ANN – Example 3 R Code

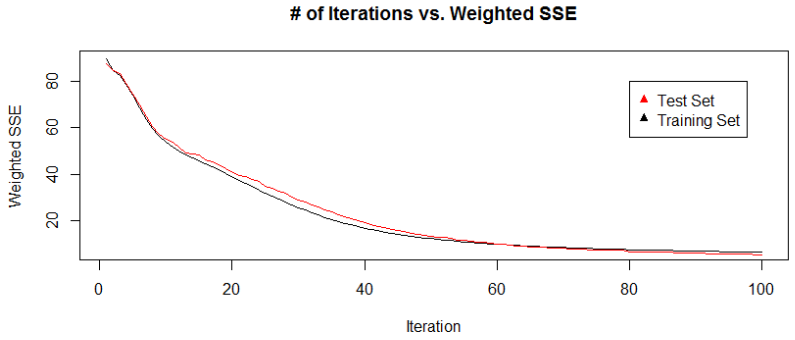
- In this third example, we use the R library "RSNNS" to train and build a neural network that is able to predict the identification of Iris plant Species.

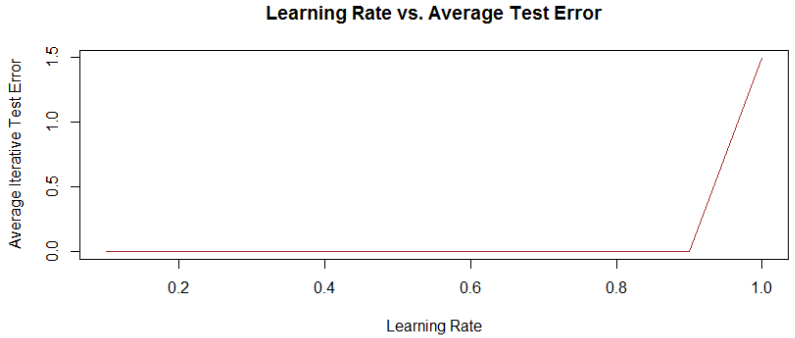
- The ANN performs the classification on the basis of plant attribute measurements: Sepal Length, Sepal Width, Petal Length, Petal Width.

- This example of classification with neural networks uses the standard back-propagation algorithm.

ANN – Example 3 R Code (cont.)

```r
# Example 3: Create a ANN for classification

# Predict the identification of Iris plant Species on the basis of plant
# attribute measurements: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width

library(RSNNS)

# Load and store the 'iris' data
data(iris)

# Generate a sample from the 'iris' data set
irisSample <- iris[sample(1:nrow(iris), length(1:nrow(iris))), 1:ncol(iris)]
irisValues <- irisSample[, 1:4]
head(irisValues)
irisTargets <- irisSample[, 5]
head(irisTargets)

# Generate a binary matrix from an integer-valued input vector representing class labels
irisDecTargets <- decodeClassLabels(irisTargets)
head(irisDecTargets)

# Split the data into the training and testing set, and then normalize
irisSample <- splitForTrainingAndTest(irisValues, irisDecTargets, ratio = 0.15)
irisSample <- normTrainingAndTestSet(irisSample)

# Train the Neural Network (Multi-Layer Perceptron)
nn3 <- mlp(irisSample$inputsTrain, irisSample$targetsTrain, size = 2, learnFuncParams = 0.1, maxit = 100,
  inputsTest = irisSample$inputsTest, targetsTest = irisSample$targetsTest)
print(nn3)

# Predict using the testing data
testPred6 <- predict(nn3, irisSample$inputsTest)

# Calculate the Confusion Matrices for the Training and Testing Sets
confusionMatrix(irisSample$targetsTrain, fitted.values(nn3))
confusionMatrix(irisSample$targetsTest, testPred6)

# Calculate the Weights of the Newly Trained Network
weightMatrix(nn3)

# Plot the Iterative Error of both training (black) and test (red) error
# This shows hows the Number of Iterations Affects the Weighted SSE
plotIterativeError(nn3, main = "# of Iterations vs. Weighted SSE")
legend(80, 80, legend = c("Test Set", "Training Set"), col = c("red", "black"), pch = 17)

# See how changing the Learning Rate Affects the Average Test Error
err <- vector(mode = "numeric", length = 10)
learnRate = seq(0.1, 1, length.out = 10)
for (i in 10){
  fit <- mlp(irisSample$inputsTrain, irisSample$targetsTrain, size = 2, learnFuncParams = learnRate[i], maxit = 50,
    inputsTest = irisSample$inputsTest, targetsTest = irisSample$targetsTest)
  err[i] <- mean(fit$IterativeTestError)
}

# Plot the Effect of Learning Rate vs. Average Iterative Test Error
plot(learnRate, err, xlab = "Learning Rate", ylab = "Average Iterative Test Error",
  main = "Learning Rate vs. Average Test Error", type = "l", col = "brown")
```





We can see the iterative error of both training and test sets, and we see how changing the learning rate affects the average test error!