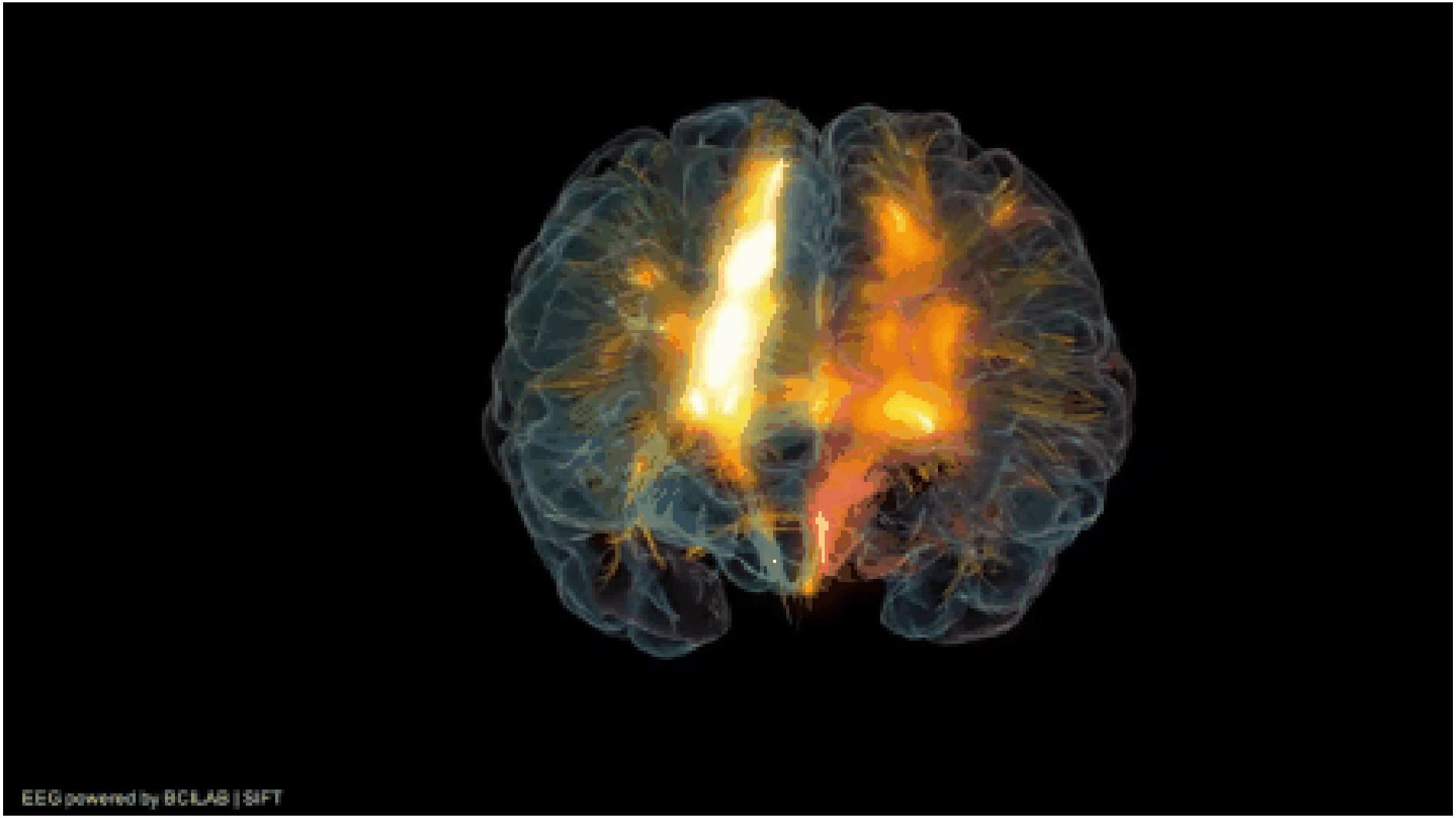
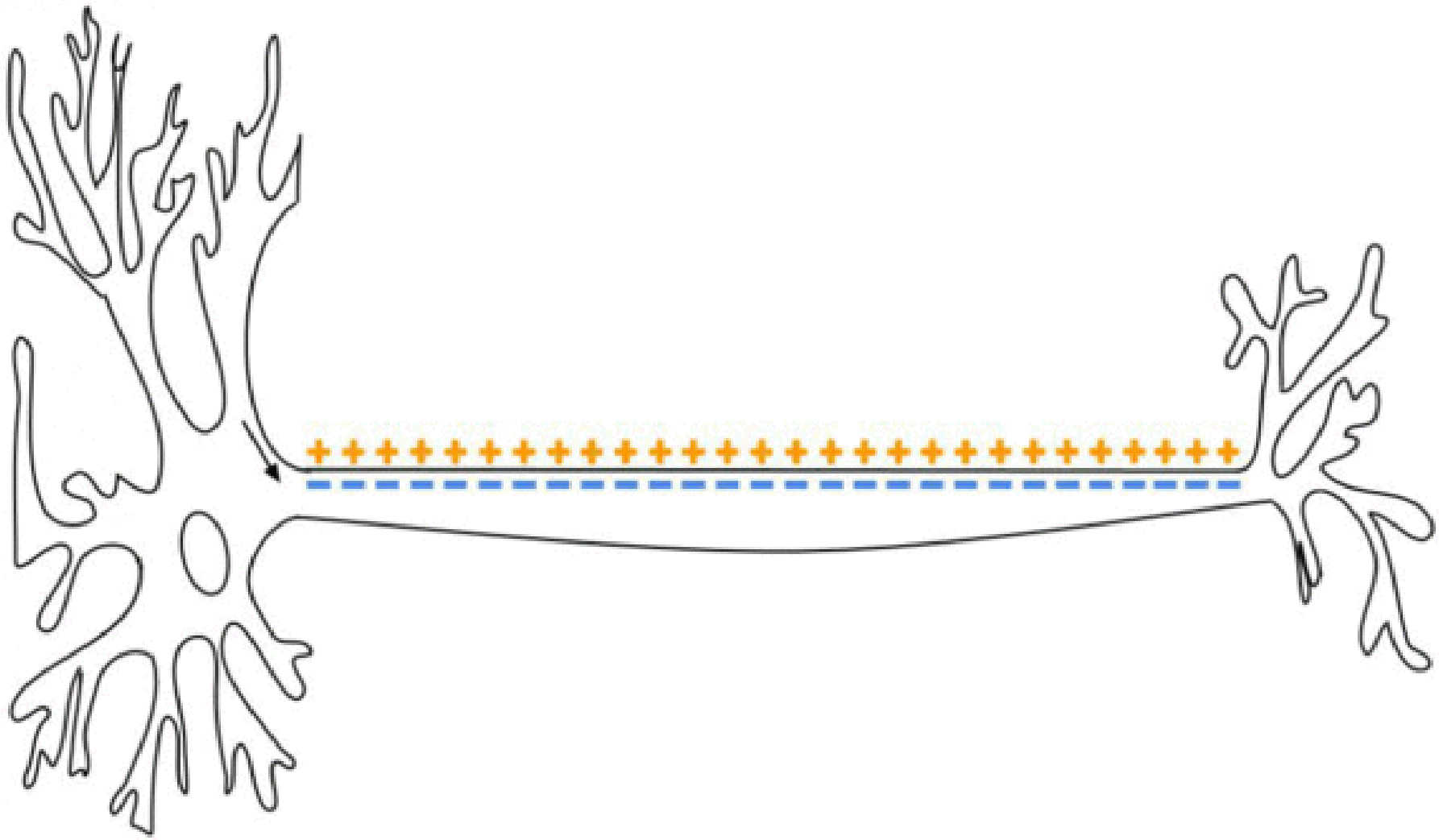
Neural Networks

October 14, 2019

An example of neural network

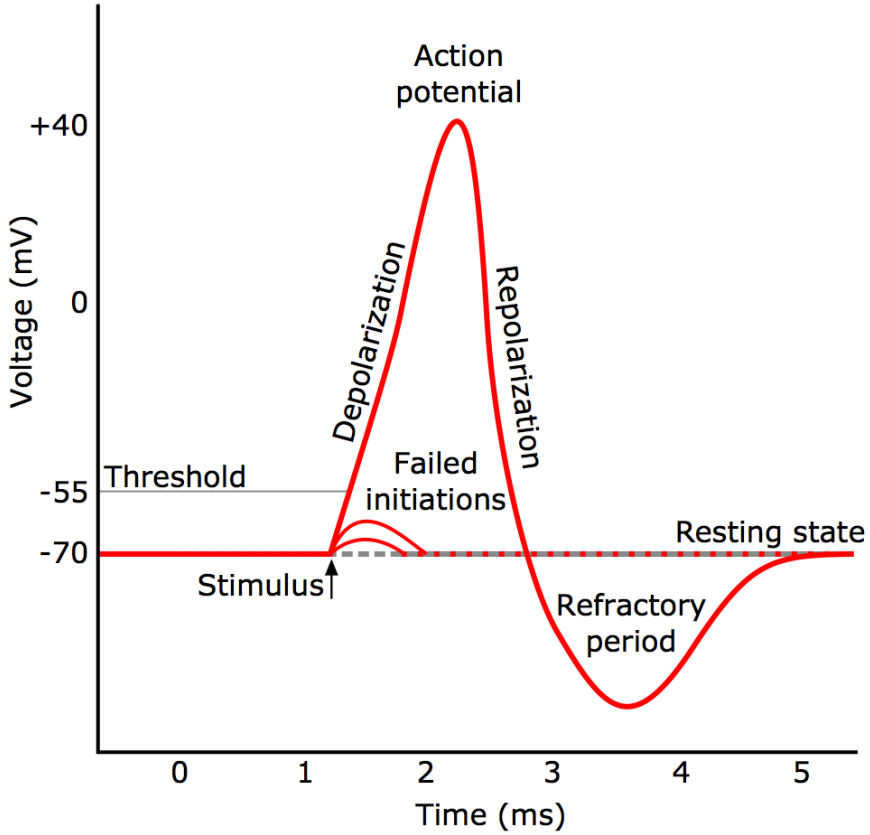


Neuron firing

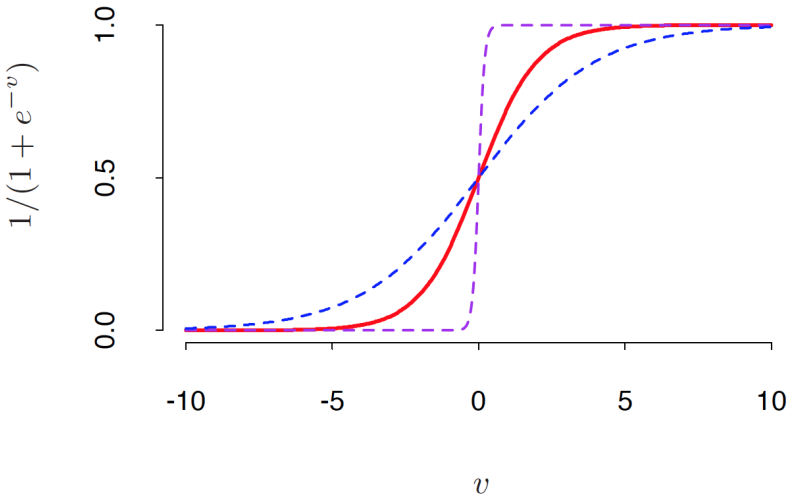


Stimulating a neuron

Change of voltage during firing



Probability of firing by a neuron



Sigmoid function

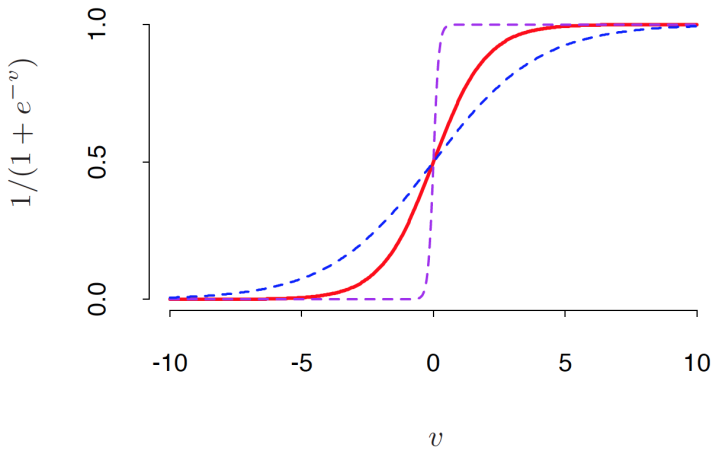
- The following function may represent the probability of firing of a neuron given a stimuli v

$$\sigma(v) = \frac{1}{1+e^{-v}} = \frac{e^{v}}{1+e^{v}}.$$

  Does this function resembles something we have seen before? Think **logistic regression**! More generally

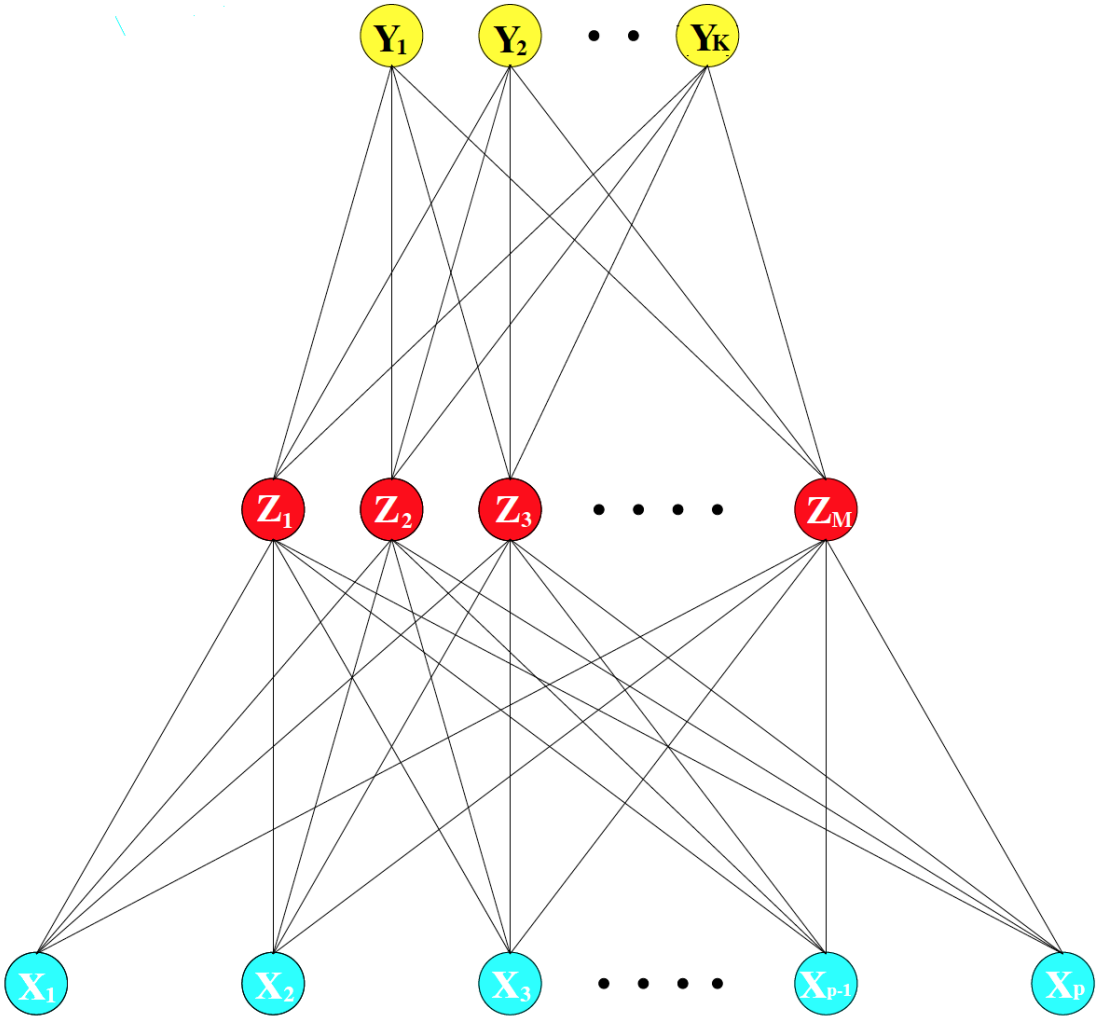
$$\sigma_s(v) = \sigma(sv) = \frac{1}{1+e^{-sv}}.$$

- The function for $s = 1/2$ (blue curve) and $s = 10$ (red curve)



- Very large s yields so called hard firing (jump function).

Neural network



One layer neural network

A layer is the middle unobserved part of the network.

- X_j's - stimuli $j = 1, \ldots, p$
- Z_m's - neurons $m = 1, \ldots, M$
- Y_k's - responses $k = 1, \ldots, K$

Neural network for regression and classification

- Function $Z = (Z_1, \ldots, Z_M)$ of the p-dimensional inputs X:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \ m = 1, \ldots, M$$

- The output

$$Y_k = g_k(T), \ k = 1, \ldots, K$$

where $T = (T_1, \ldots, T_K)$:

$$T_k = \beta_{0k} + \beta_k^T Z, \ k = 1, \ldots, K,$$

- Two standard cases of g_k:
  - the linear case (for **regression** model)

$$g_k(T) = T_k,$$

  - the multilogit linear model (for **classification**)

$$g_k(T) = \frac{e^{T_k}}{e^{T_1} + \cdots + e^{T_K}},$$

Parameters of neural network

- *Hidden units* $Z_1, \ldots, Z_M$ are not directly observed and their number has to determined (usually large number is used $M = 20, \ldots, 100$)
- The numerical parameters **weights** have to be learned from the data

$$\{\alpha_{0m}, \alpha_m; m = 1, 2, \ldots, M\}\text{–}M(p+1) \text{ weights}$$

$$\{\beta_{0k}, \beta_k; k = 1, 2, \ldots, K\}\text{–}K(M+1) \text{ weights}$$

- The parameters are determined in a recursive way by minimizing an appropriate function
- All the parameters put together are denoted by θ

Highlights

- *Hidden units* $Z_1, \ldots, Z_M$ are non-linear functions of linear combinations of the inputs.
- The non-linearity is through σ.
- The outputs T_k are again linearly transformed
- The parameters of the model are learned from the data
- Another simple way accounting on non-linearity
- The sigmoid function is approximately linear near zero, the model is equivalent to linear model
- With increase of the arguments in σ the model becomes non-linear
- By taking linear combinations of the inputs we go beyond simple additive models

Criteria of optimality

- We use standard citeria of optimality
  - The Sum of Squares, for **regression**

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

  - The prediction of $(y_1, \ldots, y_K)$ is

$$(g_1(T(\sigma(x))), \ldots, g_K(T(\sigma(x))).$$

  - Cross-entropy (deviance) for **classification**

$$R(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} log\left(g_k(T(\sigma(x_i)))\right), x_i \in \mathbb{R}^p.$$

  - The classiffier is 'winner-takes-all'

$$G(x) = \text{argmax}_k g_k(T(\sigma(x)))$$

Back-propagation

- The problem with finding the minimum is that the function is not convex (concave).
- The back-propagation method of fitting the model is based on the gradient descent method.
- It updates parameters in the direction of the gradient by scaling it by a learning rate.
- The main value of the neural networks is that the steps in the algorithm can be fairly easily obtained due to a simple form of the derivatives.
- Recall the computed derivatives in the logistic regression model.
- Initial values of the parameters (weights) are typically chosen uniform over the range $[-0.7, 0.7]$. This applies if that input variables has been standardized with respect to their mean and standard deviation.
- We do not discuss the details of the algorithms.

Overfitting and the weight decay regularization

- Neural networks tend to overfit the data, so originally, the optimization was stopped before finding the maximum to avoid the problem.
- Another more elegant solution is adding to $R(\theta)$ a penalty term $\lambda J(\theta)$, where
$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{km}^2.$$
- $\lambda \geq 0$ is a tuning parameter called the *weight decay* and its larger values tend to shrink the parameters (weights) toward zero.
- Cross-validation is used to choose λ.

Mixture of Gaussian clusters – the benchmark model

- **BLUE class**: 10 means m_k, $k = 1, \ldots, 10$ are generated from a bivariate Gaussian distribution $N((1,0), \mathbf{I})$ and subsequently 100 observations are generated independently each time by taking at random one of m_k's and than simulating a value from $N(m_k, \mathbf{I}/5)$.
- **ORANGE class** has been created in a similar fashion except the initial means were simulated from $N((0,1), \mathbf{I})$.
- These simulated data constitute a training sample of 200 observations
- For the testing different approaches to the fit 10 000 data from the model has been simulated.
  - **Would you be able to write a program in R to simulate data from this model?**

Linear classifier

X is 200 $\times$ 2 input data matrix and the orthogonal projection $\hat{\mathbf{y}}$ of the observation 0(blue)-1(orange) vector **y** to the space of spanned by the columns of the design matrix **X**. This projection can be expressed as the matrix $\mathbf{P} = \mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'$, so that

$$\hat{\mathbf{y}} = \mathbf{P}\mathbf{y}$$
$$\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

We define define the classification rule $\mathbf{x}^T\hat{\boldsymbol{\beta}} > 0.5$.

Would you be able to write a program in R to evaluate this classifier?

FIGURE 2.1. *A classification example in two dimensions. The classes are coded as a binary variable (BLUE = 0, ORANGE = 1), and then fit by linear regression. The line is the decision boundary defined by $x^T\hat{\beta} = 0.5$. The orange shaded region denotes that part of input space classified as ORANGE, while the blue region is classified as BLUE.*

Nearest neighbors classifiers

Another simple classifier, take k closest neighbors and the dominant class among them is used as a classifier.

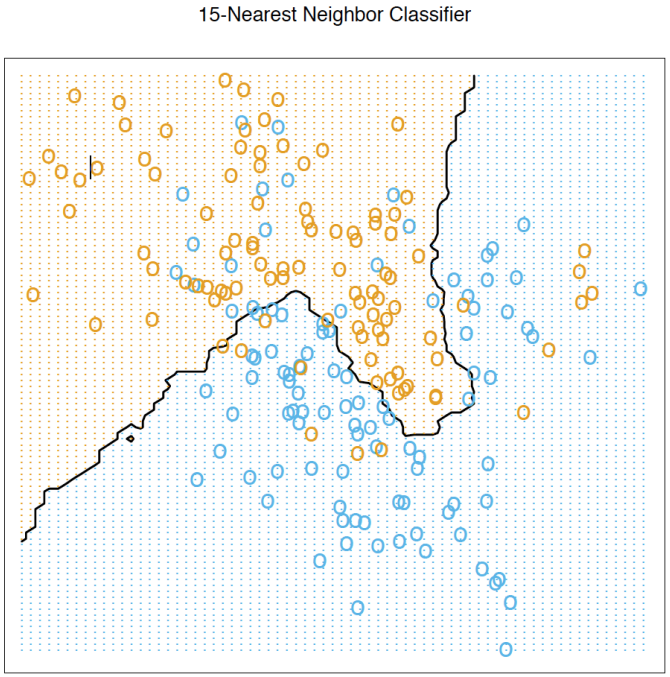


FIGURE 2.2. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$) *and then fit by* 15*-nearest-neighbor averaging as in (2.8). The predicted class is hence chosen by majority vote amongst the* 15*-nearest neighbors.*

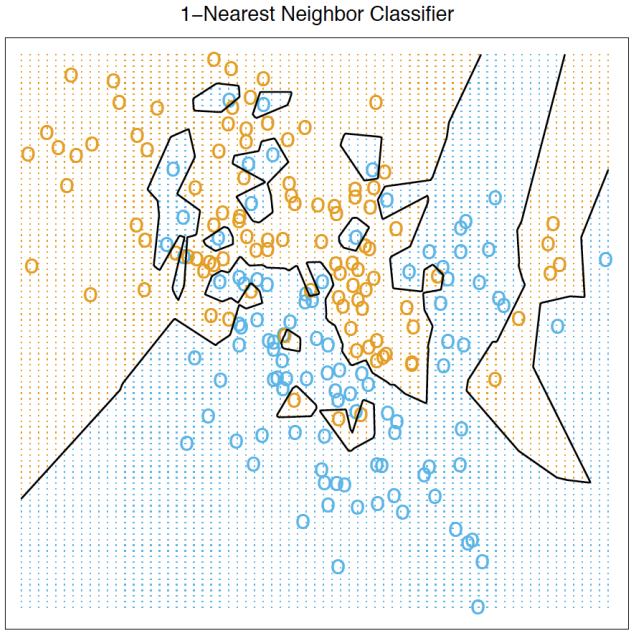


FIGURE 2.3. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$)*, and then predicted by* 1*-nearest-neighbor classification.*

Optimal classifier vs. the nearest neighbors

Since we know the model from which the data are simulated, we can derive the optimal classifier and compare the other classifiers to this one using generated 10 000 testing data.

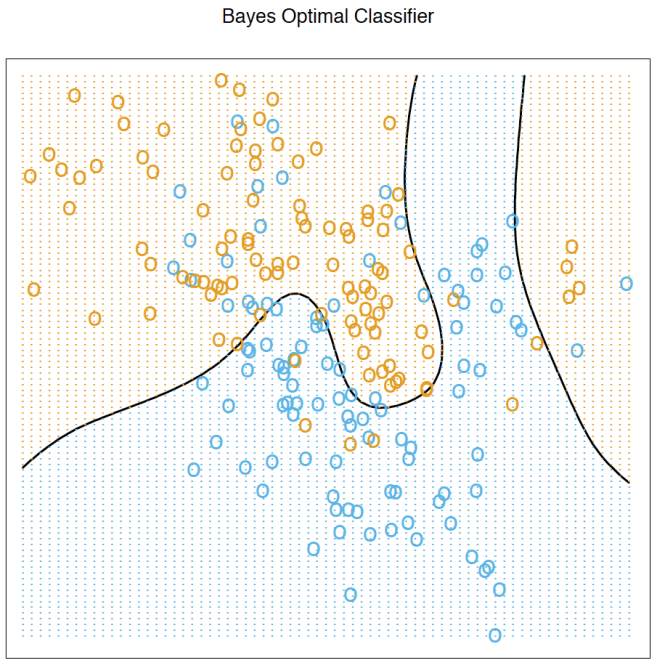


FIGURE 2.5. *The optimal Bayes decision boundary for the simulation example of Figures 2.1, 2.2 and 2.3. Since the generating density is known for each class, this boundary can be calculated exactly (Exercise 2.2).*

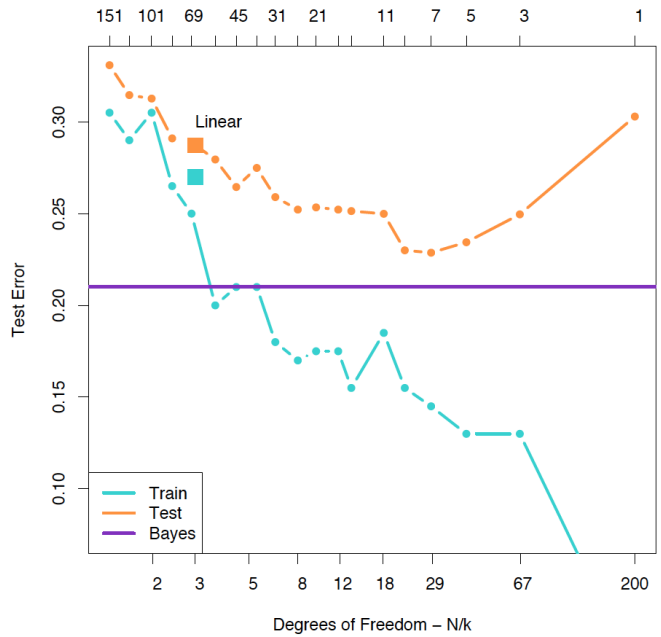
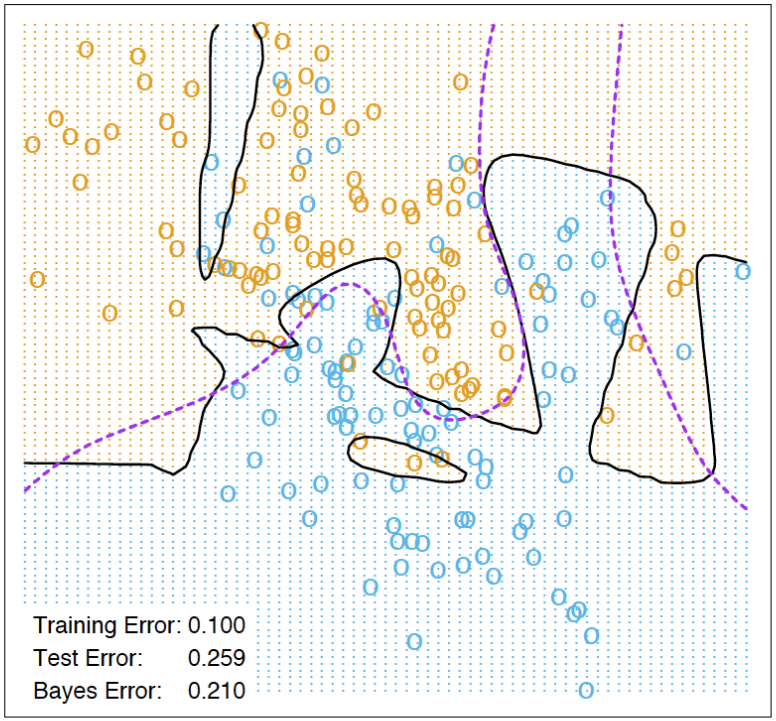


FIGURE 2.4. *Misclassification curves for the simulation example used in Figures 2.1, 2.2 and 2.3. A single training sample of size* 200 *was used, and a test sample of size* 10,000*. The orange curves are test and the blue are training error for k-nearest-neighbor classification. The results for linear regression are the bigger orange and blue squares at three degrees of freedom. The purple line is the optimal Bayes error rate.*

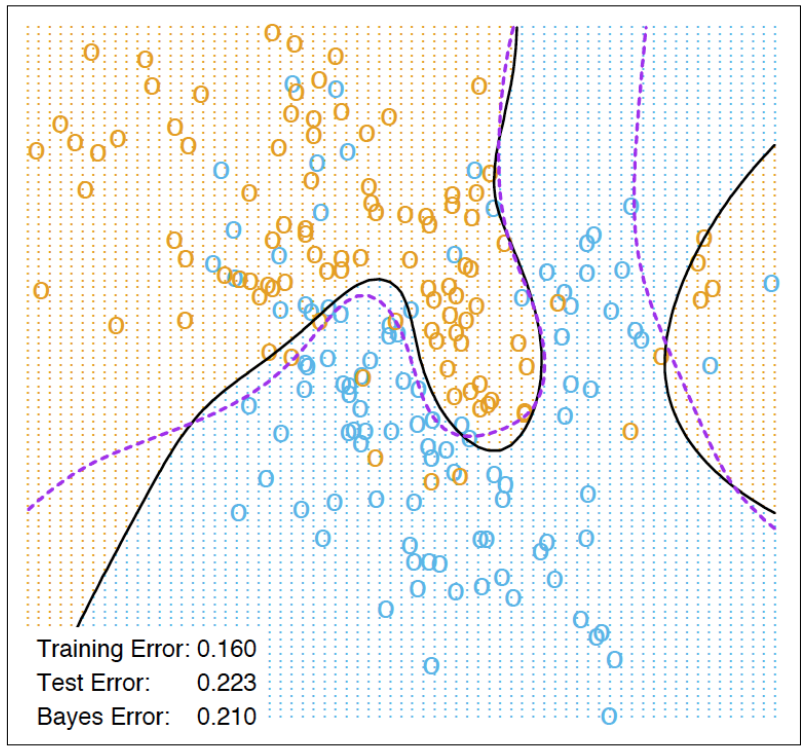
Neural networks

Dashed line represents the optimal fit while continuous lines represent the neural network fit without a weight decay on the left hand side and with a weight decay on the right one

Neural Network - 10 Units, No Weight Decay



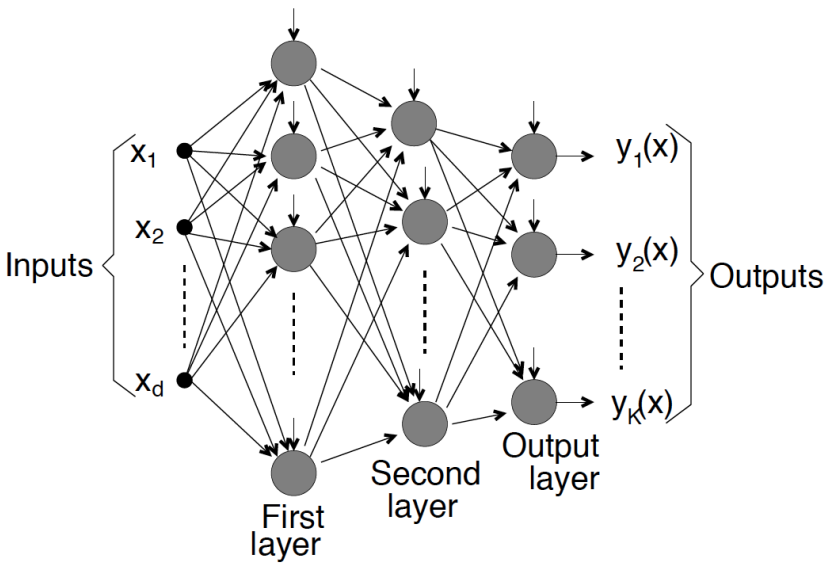
Neural Network - 10 Units, Weight Decay=0.02

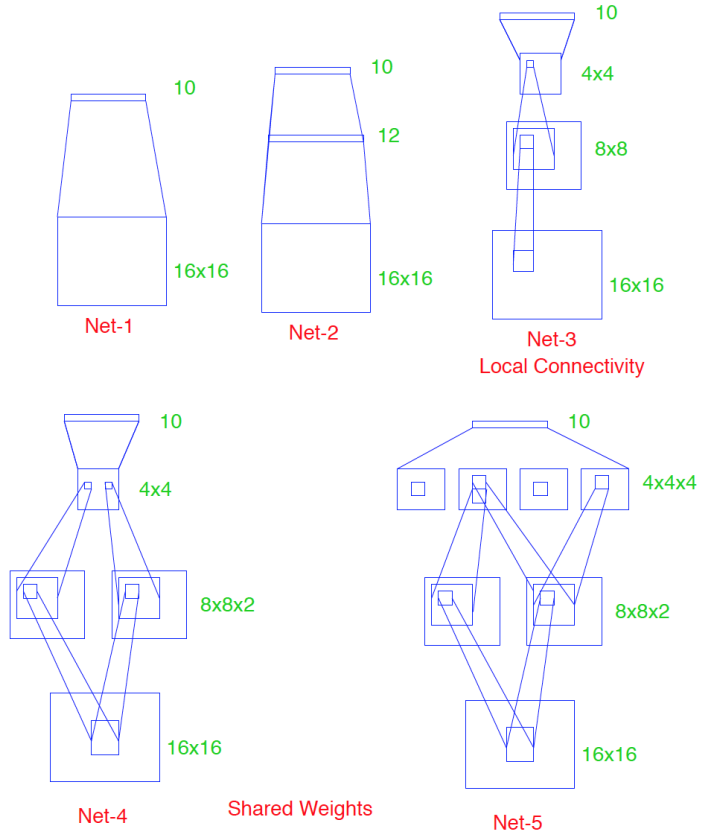


Generalizations – more layers

- 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.
- Choice of the number of hidden layers is guided by background knowledge and experimentation.
- Use of multiple hidden layers allows construction of hierarchical features at different levels of resolution.

More complex architecture





Real life example – digit recognition

- The complex neural networks were put to work to recognize digits.
- There was 320 digits in the training set and 160 in the test data.
- There are $16 \times 16 = 256$ inputs (pixels to which digits are coded).
- There 10 outputs (classification to one of the ten digits).
- Fit was made with the sum-of-squares error function.
- The predicted value $\hat{g}_k(T(\sigma(x))$ represents the estimated probability that an image has digit class k, $k = 0, 1, 2, \ldots, 9$.
- Weights close to zero mean that there is no link between a corresponding input and the hidden unit.

Performance

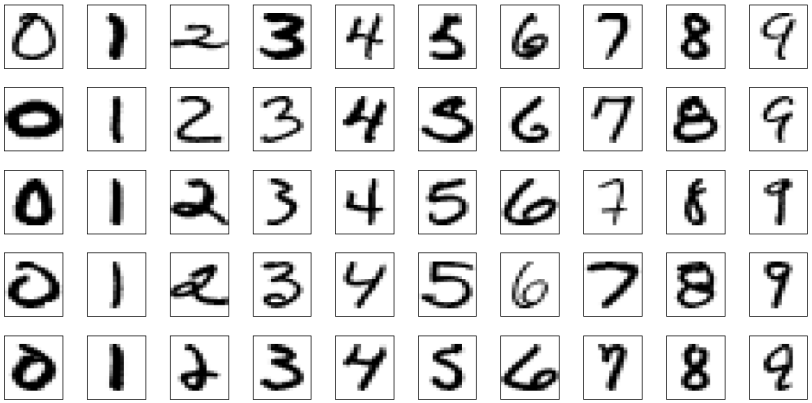


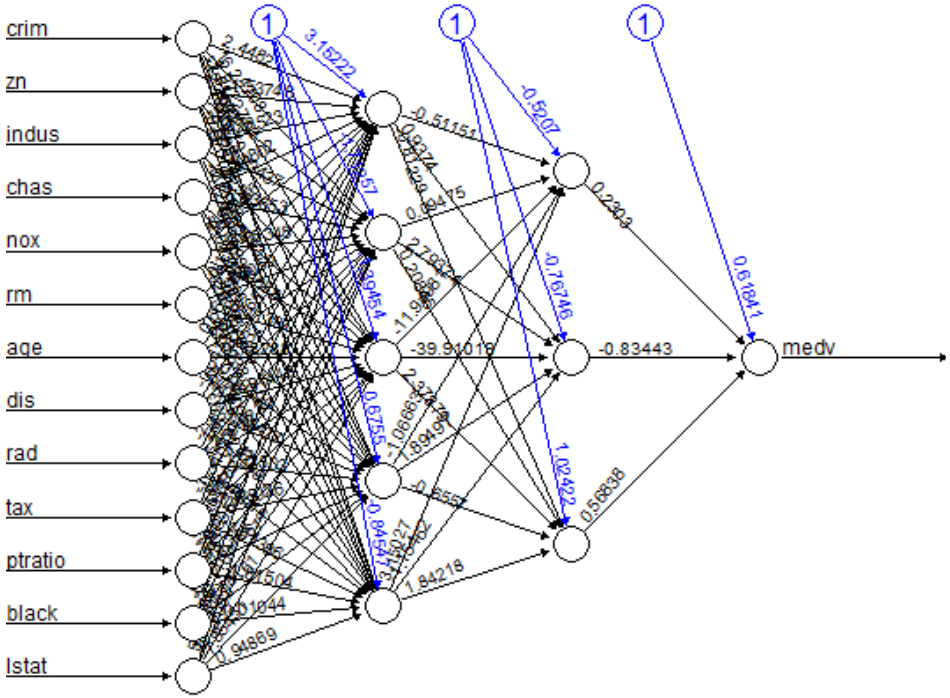
FIGURE 11.9. *Examples of training cases from ZIP code data. Each image is a* 16×16 *8-bit grayscale representation of a handwritten digit.*

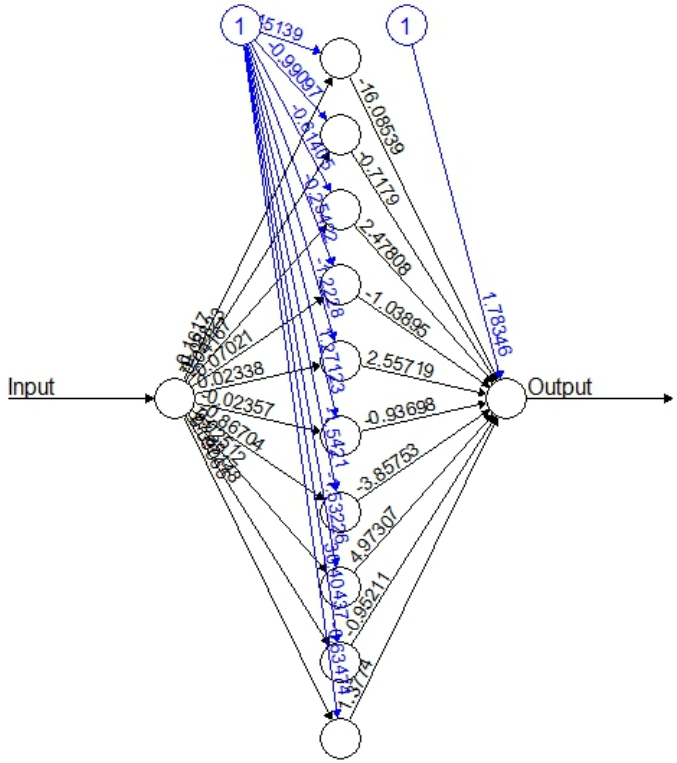
TABLE 11.1. *Test set performance of five different neural networks on a handwritten digit classification example (Le Cun, 1989).*

	Network Architecture	Links	Weights	% Correct
Net-1:	Single layer network	2570	2570	80.0%
Net-2:	Two layer network	3214	3214	87.0%
Net-3:	Locally connected	1226	1226	88.5%
Net-4:	Constrained network 1	2266	1132	94.0%
Net-5:	Constrained network 2	5194	1060	98.4%

Neuralnet R-package

Boston data and predicting the square root of a number



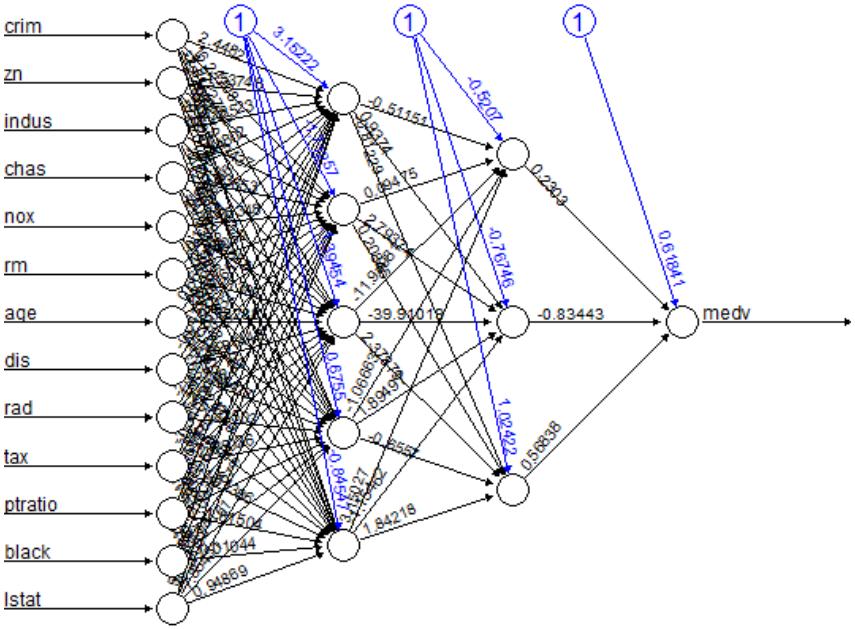


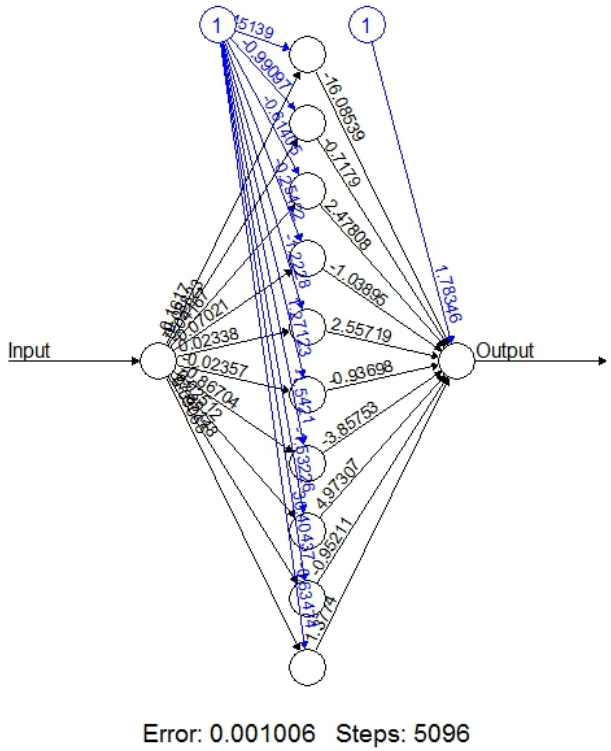
Error: 0.001006 Steps: 5096

Neuralnet – two examples

Predicting the square root of a number

Boston data





Blue lines with numbers represent additional shift (bias, intercept) in the linear components, which were denoted as α_{0m}'s and β_{0k}'s.

Learning about square root

The neural network is going to take a single input (the number that you want square rooting) and produce a single output (the square root of the input). The middle contains 10 hidden neurons which are first trained to obtain weights.

```
Input Expected Output Neural Net Output
```

```
   Input  Expected Output  Neural Net Output
     1                 1       0.9623402772
     4                 2       2.0083461217
     9                 3       2.9958221776
    16                 4       4.0009548085
    25                 5       5.0028838579
    36                 6       5.9975810435
    49                 7      6.9968278722
    64                 8      8.0070028670
    81                 9      9.0019220736
   100                10      9.9222007864
```

The code – the training phase

```
install.packages('neuralnet')
library("neuralnet")

#Going to create a neural network to perform sqare rooting
#Type ?neuralnet for more information on the neuralnet library

#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)
```

The code for the testing phase

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

#Lets see what properties net.sqrt has
ls(net.results)

#Lets see the results
print(net.results$net.result)

#Lets 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)
```

Understanding neural network output

What is the relation between

the neural network:

- Function $Z = (Z_1, \ldots, Z_M)$ of the p-dimensional inputs X:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \ m = 1, \ldots, M$$

- The output $T = (T_1, \ldots, T_K)$:

$$T_k = \beta_{0k} + \beta_k^T Z, \ k = 1, \ldots, K,$$

the output code:

```
net.sqrt <- neuralnet(Output~Input,trainingdata, h
 plot(net.sqrt)

sqrt5=compute(net.sqrt,as.data.frame(5))
sqrt5

# $neurons
# $neurons[[1]]
# 5
# [1,] 1 5
#
# $neurons[[2]]
# [,1]      [,2]      [,3]
# [1,]    1 0.1252886 0.4861516
#
#
# $net.result
# [,1]
# [1,] 2.310066
```

and the actual mathematical relation

$$y = f(x) \quad ?$$

Specifying the network

- Let us consider only one layer with two neurons for $\sqrt{x}$ problem.
- Identify the dimensions and the form of the mathematical form of the neural network: $Z = (Z_1, \ldots, Z_M)$ of the p-dimensional inputs X:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \ m = 1, \ldots, M$$

The output $T = (T_1, \ldots, T_K)$:

$$T_k = \beta_{0k} + \beta_k^T Z, \ k = 1, \ldots, K,$$

- $M =$?, $K =$? and thus with the sigmoid activation function

$$y = \beta_{01} + \frac{\beta_1}{1 + e^{-(\alpha_{01} + \alpha_1 x)}} + \frac{\beta_2}{1 + e^{-(\alpha_{02} + \alpha_2 x)}}$$

The algorithm searches for the best β_0, β_1, α_{01}, α_{02}, α_1, α_2 for the above function to resemble $\sqrt{x}$.

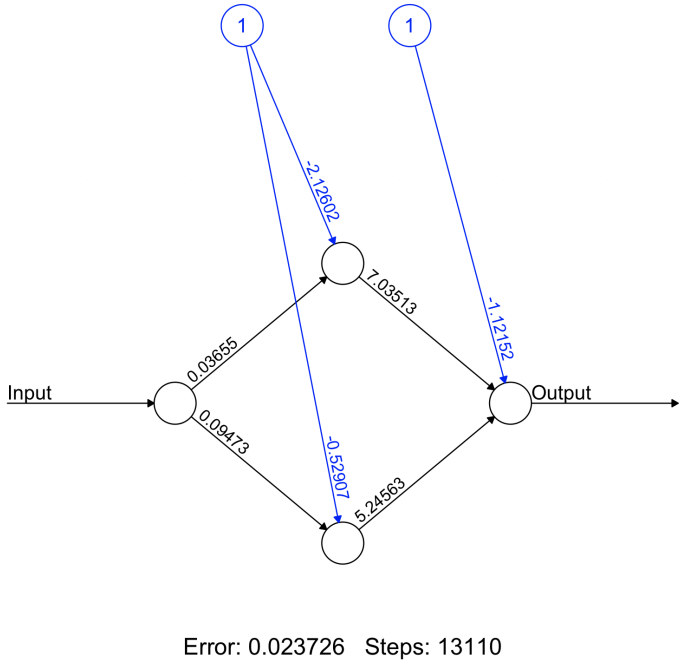
Playing with the code

```
net.sqrt <- neuralnet(Output~Input,trainingdata, hidden=2, threshold=0.01)
 plot(net.sqrt)

sqrt5=compute(net.sqrt,as.data.frame(5))
sqrt5

# $neurons
# $neurons[[1]]
# 5
# [1,] 1 5
#
# $neurons[[2]]
# [,1]      [,2]      [,3]
# [1,]    1 0.1252886 0.4861516
#
#
# $net.result
# [,1]
# [1,] 2.310066
net.sqrt$act.fct

net.sqrt$act.fct(1)
```



Connecting the dots

```
net.sqrt$weight
[[1]]
[[1]][[1]]
           [,1]        [,2]
[1,] -2.1260205 -0.52906884
[2,]  0.0365492  0.09473225

[[1]][[2]]
          [,1]
[1,] -1.121525
[2,]  7.035129
[3,]  5.245626

sqrt5$neurons[[2]]%*%net.sqrt$weights[[1]][[2]]
         [,1]
[1,] 2.310066
t(net.sqrt$weights[[1]][[1]])%*%c(1,5)
           [,1]
[1,] -1.9432745
[2,] -0.0554076

net.sqrt$act.fct(t(net.sqrt$weights[[1]][[1]])%*%
c(1,5))
          [,1]
[1,] 0.1252886
[2,] 0.4861516
```

Thus $\beta_{01} = -1.12$, $\beta_1 = 7.03$, $\beta_2 = 5.24$, $\alpha_{01} = -2.12$, $\alpha_{02} = -0.52$, $\alpha_1 = 0.036$, $\alpha_2 = 0.094$ so that

$$y = -1.12 + \frac{7.03}{1 + e^{2.12 - 0.036x}} + \frac{5.24}{1 + e^{0.52 - 0.094x}}$$

We can check this

```
-1.12+7.03/(1+exp(2.12-0.036*tests))+
  5.24/(1+exp(0.52-0.094*tests))
[1] 1.728181 1.872241 2.018863 2.167559 2.317818
2.469115 2.620920 2.772705 2.923959

nntests=compute(net.sqrt,as.data.frame(tests))
t(nntests$net.result)
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 1.715224 1.860459 2.008346 2.158389 2.310066
2.46284 2.616168 2.76951 2.92234
```