Simple Neural Network Classifiers

Shane T. Mueller shanem@mtu.edu

2021-03-03

Artificial Neural Networks have been used by researchers since at least the 1950s, and rose in prominence when computing resources became widely available. There have been at least 3-4 eras of ANN, with each success and exuberance followed by a quiescence and failure or replacement by other tools. There has always been an intetrplay in this area between computational neuroscientists, computational vision, machine learning, psychology, computer science, and statisticians, with many advances coming by modeling human processes and then applying these algorithms to real-life problems.

- In the 1950s, notions of the ANN were introduced (Perceptron, McCollough & Pitts)
- In the 1960s, inadequacy of these methods were found (XOR problems), and
- In the 1970s and 1980s, back-propagation provided a reasonable learning approach for multi-layer neural networks for supervised classification problems (Rumelhart & McClelland).
- In the 1970s-1990s, other advances in self-organizing maps, un-supervised learning and hebbian networks provided alternate means for representing knowledge.
- In the 1990s-2000s, other machine learning approaches appeared to take precedence, with lines blurring between ANNs, machine classification, reinforcement learning, and several approaches that linked supervised and unsupervised models (O'Reilly, HTMs, Grossberg).
- In the 2000s, Bayesian approaches were foremost
- In the 2010s, we have seen a resurgence of deep-learning methods. The advances here have been driven by (1) advances in software that allow us to use GPU (graphics cards) to efficiently train and use networks (2) large data labeled data sets, often generated through amazon mechanical turk or as a byproduct of CAPTCHA systems; (3) using 15+ hidden layers; (4) effective use of specific types of layer architectures, including convolutional networks that de-localize patterns from their position in an image.

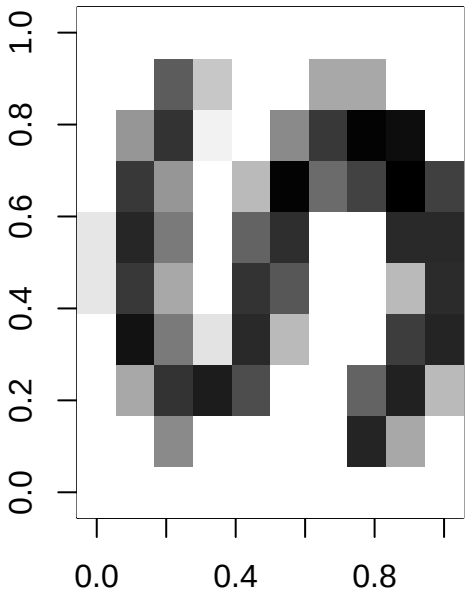
A simple two-layer neural network can be considered that is essentially what our multinomial regression or logistic regression model is doing. Inputs include a set of features, and output nodes (possibly one per class) are classifications that are learned. Alterately, a pattern can be learned by the output nodes. The main issue here is estimating weights, which are done with heuristic error-propagation approaches rather than MLE or least squares. This is inefficient for two-layer problems, but the heuristic approach will pay off for more complex problems.

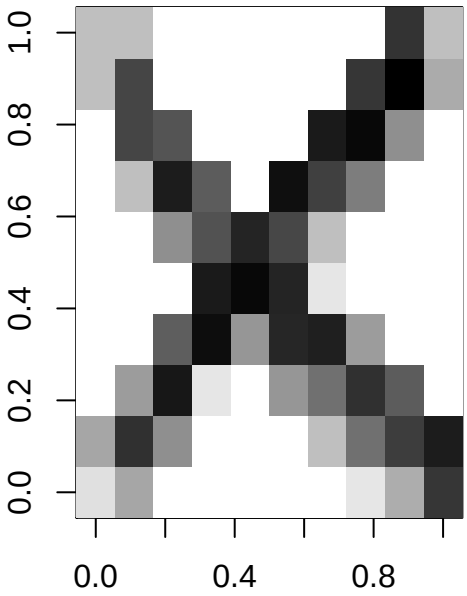
The main advance of the late 1970s and early 1980s was adoption of 'hidden layer' ANNs. These hidden layers permit solving the XOR problem: when a class is associated with an exclusive or logic. The category structure is stored in a distributed fashion across all nodes, but there is a sense in which the number of hidden nodes controls how complex the classification structure that is possible. With a hidden layer, optimization is difficult with traditional means, but the heuristic approaches generally work reasonably well.

We will start with an image processing example. We will take two images (S and X) and sample features from them to make exemplars:

```
# library(imager) s <-load.image('s.data.bmp') x <- load.image('x.bmp') svec <-
# as.vector(s) xvec <- as.vector(x) write.csv(svec,'s.csv')
# write.csv(xvec,'x.csv')

## here, the lower the value, the darker the image.
```

```
svec <- read.csv("s.csv")
xvec <- read.csv("x.csv")

## reverse the numbers
svec$x <- 255 - svec$x
xvec$x <- 255 - xvec$x

par(mfrow = c(1, 2))

image(matrix(svec$x, 10, 10, byrow = T), col = grey(100:0/100))
image(matrix(xvec$x, 10, 10, byrow = T), col = grey(100:0/100))
```

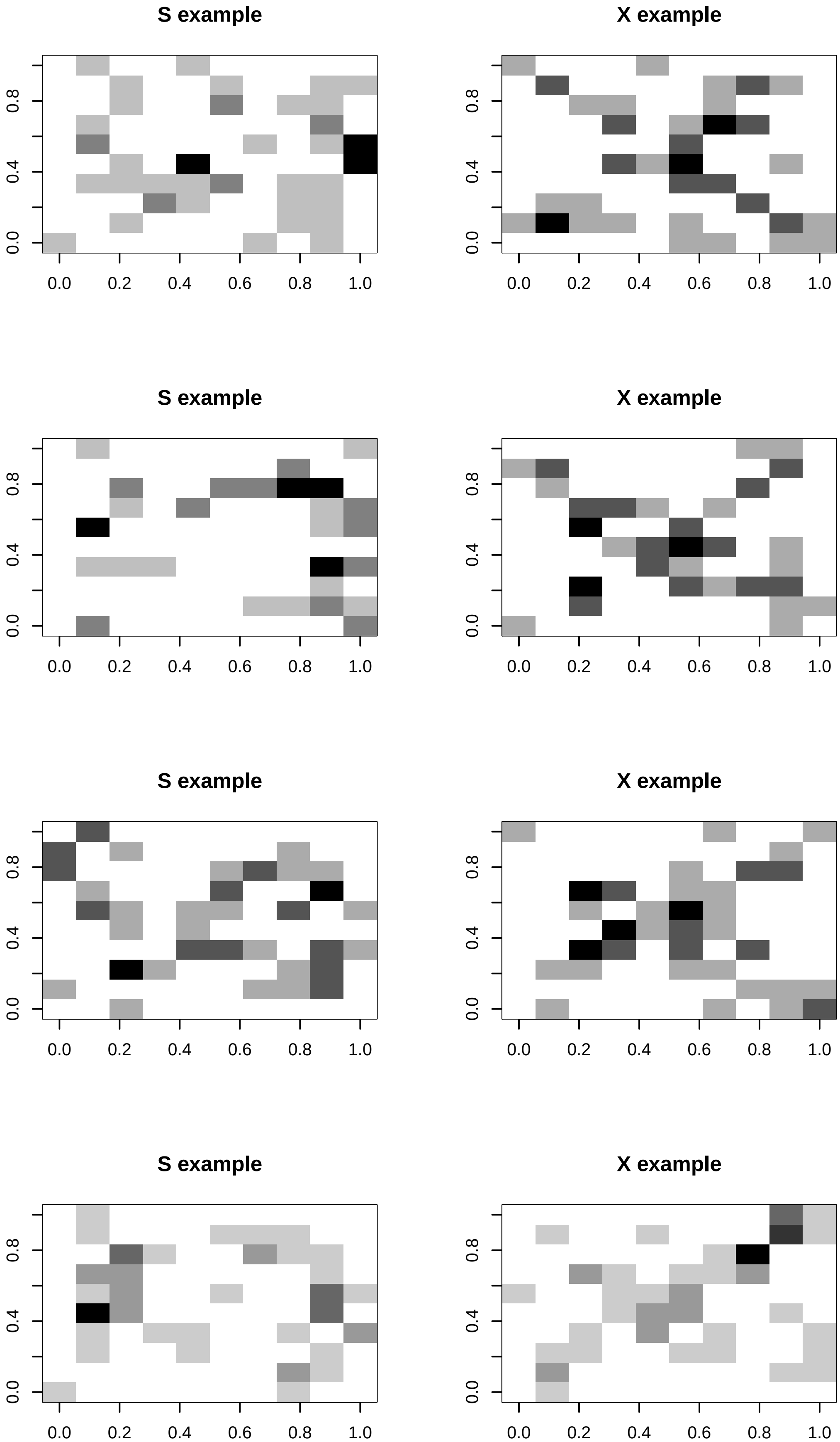




To train the model, we will sample 500 examples of each template:

```
dataX <- matrix(0, ncol = 100, nrow = 250)
dataS <- matrix(0, ncol = 100, nrow = 250)
letter <- rep(c("x", "s"), each = 250)

par(mfrow = c(4, 2))

for (i in 1:250) {
    x <- rep(0, 100)
    xtmp <- table(sample(1:100, size = 50, prob = as.matrix(xvec$x), replace = T))
    x[as.numeric(names(xtmp))] <- xtmp/max(xtmp)

    s <- rep(0, 100)
    stmp <- table(sample(1:100, size = 50, prob = as.matrix(svec$x), replace = T))
    s[as.numeric(names(stmp))] <- stmp/max(stmp)

    if (i <= 4) {
        image(matrix(s, 10, 10, byrow = T), main = "S example", col = grey(100:0/100))
        image(matrix(x, 10, 10, byrow = T), main = "X example", col = grey(100:0/100))
    }

    dataX[i, ] <- x
    dataS[i, ] <- s
```

```
}
```


S example
X example
S example
X example
S example
X example
S example
X example
0.0
0.4
0.8
0.0 0.2 0.4 0.6 0.8 1.0

```
data <- rbind(dataX, dataS)
```

Let's train a neural network on these. Note that we could probably have trained it on the prototypes–maybe from the mnist library examined in earlier chapters.

```
library(nnet)
options(warn = 2)
# model <- nnet(letter~data,size=2) #This doesn't work.

# Let's transform the letter to a numeric value
model <- nnet(y = as.numeric(letter == "s"), x = data, size = 2)
```

```
# weights:  205
initial  value 132.515864
final  value 0.000000
converged
```

```
model
```

```
a 100-2-1 network with 205 weights
options were -
```

```
# alternately, try using letter as a factor, and use a formula. This also plays
# with some of the learning parameters
merged <- data.frame(letter = as.factor(letter), data)
model2 <- nnet(letter ~ ., data = merged, size = 2, rang = 0.1, decay = 1e-04, maxit = 500,
    trace = T)
```

```
# weights:  205
initial  value 347.762887
iter  10 value 0.779448
iter  20 value 0.105397
iter  30 value 0.081781
iter  40 value 0.079592
iter  50 value 0.076442
iter  60 value 0.072840
iter  70 value 0.070106
iter  80 value 0.067371
iter  90 value 0.063322
iter 100 value 0.059827
iter 110 value 0.057483
iter 120 value 0.055696
iter 130 value 0.055383
iter 140 value 0.055340
iter 150 value 0.055332
iter 160 value 0.055330
iter 170 value 0.055329
iter 180 value 0.055328
iter 190 value 0.055325
iter 200 value 0.055323
final  value 0.055323
converged
```

Examining the model

The model contains a lot of information, including some parameters it is constructed under, and all of the fitted parameters.

We can see when using summary() that we really have two really large regressions from the 100 input features to 2 hidden nodes, and a single model with an intercept (b=bias) and two parameters from each hidden node to the output node. The default is 'logistic output units', which means this last model is essentially a logistic regression.

```
str(model)
```

```
List of 15
 $ n             : num [1:3] 100 2 1
 $ nunits        : int 104
 $ nconn         : num [1:105] 0 0 0 0 0 0 0 0 0 0 ...
 $ conn          : num [1:205] 0 1 2 3 4 5 6 7 8 9 ...
 $ nsunits       : int 104
 $ decay         : num 0
 $ entropy       : logi FALSE
 $ softmax       : logi FALSE
 $ censored      : logi FALSE
 $ value         : num 0
 $ wts           : num [1:205] -11.752 -0.185 -0.865 -0.276 -0.109 ...
 $ convergence   : int 0
 $ fitted.values: num [1:500, 1] 0 0 0 0 0 0 0 0 0 0 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : NULL
 $ residuals     : num [1:500, 1] 0 0 0 0 0 0 0 0 0 0 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : NULL
 $ call          : language nnet.default(x = data, y = as.numeric(letter == "s"), size = 2)
 - attr(*, "class")= chr "nnet"
```

```
summary(model)
```

```
a 100-2-1 network with 205 weights
options were -
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1  i7->h1  i8->h1  i9->h1
 -11.75   -0.18   -0.87   -0.28   -0.11   -1.35   -0.63   -0.82   -0.54   -0.99
i10->h1 i11->h1 i12->h1 i13->h1 i14->h1 i15->h1 i16->h1 i17->h1 i18->h1 i19->h1
  -0.42   -1.00   -1.28   -2.24   -2.79   -2.44   -2.51   -2.56   -1.63   -0.72
i20->h1 i21->h1 i22->h1 i23->h1 i24->h1 i25->h1 i26->h1 i27->h1 i28->h1 i29->h1
  -1.03   -0.57   -1.77   -3.89   -2.49   -1.50   -2.40   -2.86   -3.41   -1.83
i30->h1 i31->h1 i32->h1 i33->h1 i34->h1 i35->h1 i36->h1 i37->h1 i38->h1 i39->h1
  -0.23    0.19   -0.73   -2.73   -1.57   -1.81   -1.42   -0.30   -1.24   -1.46
i40->h1 i41->h1 i42->h1 i43->h1 i44->h1 i45->h1 i46->h1 i47->h1 i48->h1 i49->h1
   0.30   -0.05    0.29   -2.25   -3.85   -3.57   -3.31   -1.89   -0.90   -0.73
i50->h1 i51->h1 i52->h1 i53->h1 i54->h1 i55->h1 i56->h1 i57->h1 i58->h1 i59->h1
  -0.39    0.05   -0.54   -0.69   -2.15   -3.04   -3.15   -4.86   -2.09   -0.84
i60->h1 i61->h1 i62->h1 i63->h1 i64->h1 i65->h1 i66->h1 i67->h1 i68->h1 i69->h1
  -0.57   -0.55   -0.92   -1.38   -1.16   -0.98   -0.61   -2.64   -4.24   -1.59
i70->h1 i71->h1 i72->h1 i73->h1 i74->h1
  -0.53   -0.50   -3.56   -3.26   -1.25
 [ reached getOption("max.print") -- omitted 26 entries ]
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2  i7->h2  i8->h2  i9->h2
  -1.88   -0.82   -4.84    0.13    0.33    0.80    1.52    0.09    0.06   -4.03
i10->h2 i11->h2 i12->h2 i13->h2 i14->h2 i15->h2 i16->h2 i17->h2 i18->h2 i19->h2
```

```
  -3.49   -4.60   -9.69   -0.40    9.37    7.70    8.75    4.37   -4.35   -7.77
i20->h2 i21->h2 i22->h2 i23->h2 i24->h2 i25->h2 i26->h2 i27->h2 i28->h2 i29->h2
  -2.53    0.55    0.38   -1.43   -1.64    4.77    0.17   -6.47    0.79    7.67
i30->h2 i31->h2 i32->h2 i33->h2 i34->h2 i35->h2 i36->h2 i37->h2 i38->h2 i39->h2
  -0.64    0.14   -0.77    7.37   -9.23  -11.98   -8.43   -8.76    1.40    2.84
i40->h2 i41->h2 i42->h2 i43->h2 i44->h2 i45->h2 i46->h2 i47->h2 i48->h2 i49->h2
   0.44   -0.88    0.15    7.31    4.67   -3.46   -2.88    3.04   -0.27   -0.07
i50->h2 i51->h2 i52->h2 i53->h2 i54->h2 i55->h2 i56->h2 i57->h2 i58->h2 i59->h2
  -0.39   -0.56   -0.27   -4.72   -7.54   -3.68   -0.10    0.10    6.09   -0.04
i60->h2 i61->h2 i62->h2 i63->h2 i64->h2 i65->h2 i66->h2 i67->h2 i68->h2 i69->h2
  -0.26    0.53   -2.63   -6.46  -10.03   -1.41   -2.76   -3.93   -1.73    2.71
i70->h2 i71->h2 i72->h2 i73->h2 i74->h2
  -0.42   -1.67    3.51   -2.17   -3.97
 [ reached getOption("max.print") -- omitted 26 entries ]
  b->o  h1->o  h2->o
-15.20 -17.31  36.60
```

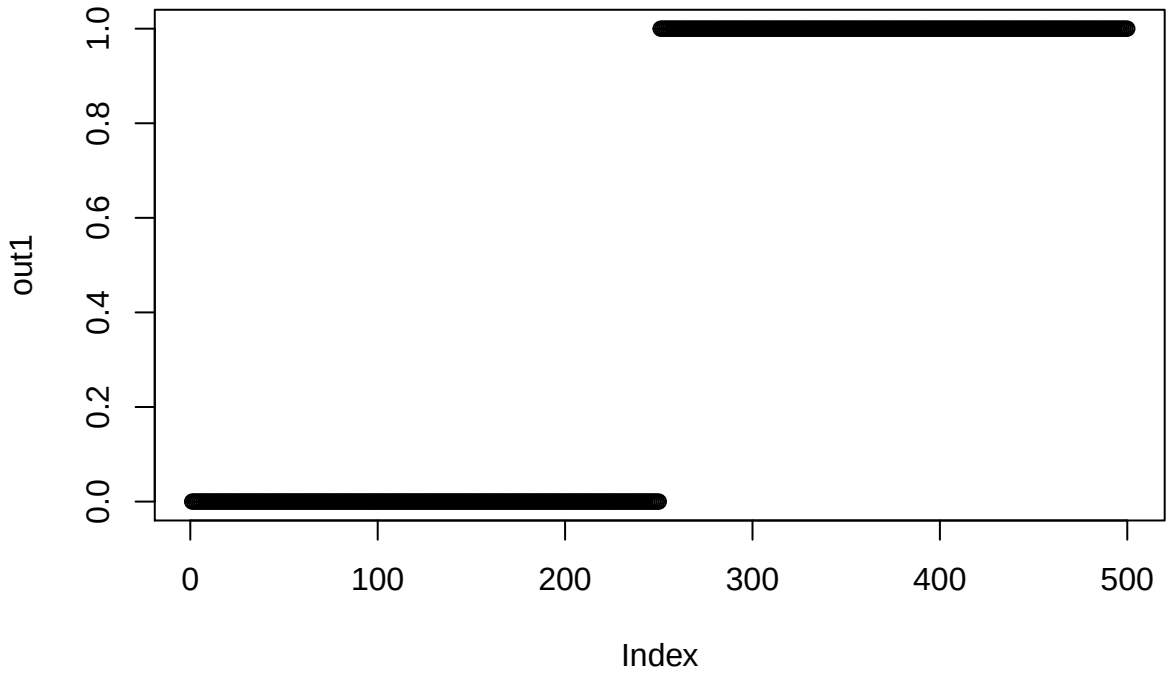
Some of the arguments we can control are:

- How the final output classifier works (options involve linout, entropy, softmax, censored). These control fitting algorithms and approaches, and may impact speed of convergence. The default settings look like they are essentially using least-squared to fit individual nodes.
- subset: allowing you to train on a subset for cross-validation
- mask: allowing only some of the input features to be trained.
- Wts: initial parameter settings. You could train a model on some data, and use those weights to then re-train on new data, for example.
- decay: this probably controls how far back data in a series are examined. It may allow for a model to adapt to a changing environment better.
- maxit and trace: fitting arguments.
- weights: strength of each case. You may have some cases you want to train on more strongly/often.

Obtaining predicted response

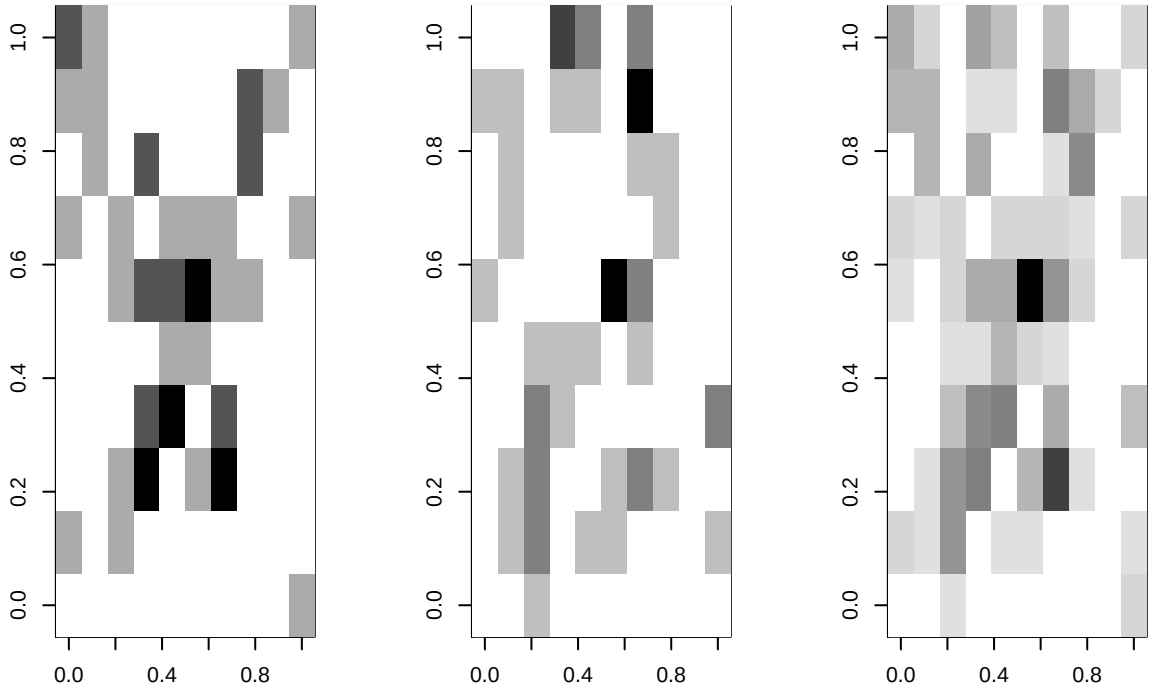
When we use predict, by default it gives us the 'raw' values–activation values returned by the trained network. Because the final layer of this network has just one node (100-2-1), it is just an activation value indicating the class (0 for X and 1 for S). The values are not exactly 0 and 1–they are floating point values very close.

```
out1 <- predict(model, newdata = data)
plot(out1)
```



If there were a combined example that was hard to distinguish we would get a different value:

```
test <- (data[3, ] + data[255, ])/2

par(mfrow = c(1, 3))
image(matrix(data[3, ], 10), col = grey(100:0/100))
image(matrix(data[255, ], 10), col = grey(100:0/100))
image(matrix(test, 10), col = grey(100:0/100))
```




```
predict(model, newdata = data[3, ] + data[4, ])
```

```
     [,1]
```

```
[1,]    0
```

```
predict(model, newdata = data[255, ])
```

```
     [,1]
[1,]    1
```

```
predict(model, newdata = test)
```

```
     [,1]
[1,]    0
```

In this case, the X shows up as a strong X, the S shows up as an S, but the combined version is a slightly weaker S.

We can get classifications using type="class":

```
predict(model2, type = "class")
```

```
 [1] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
[20] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
[39] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
[58] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
 [ reached getOption("max.print") -- omitted 425 entries ]
```

```
table(letter, predict(model2, type = "class"))
```

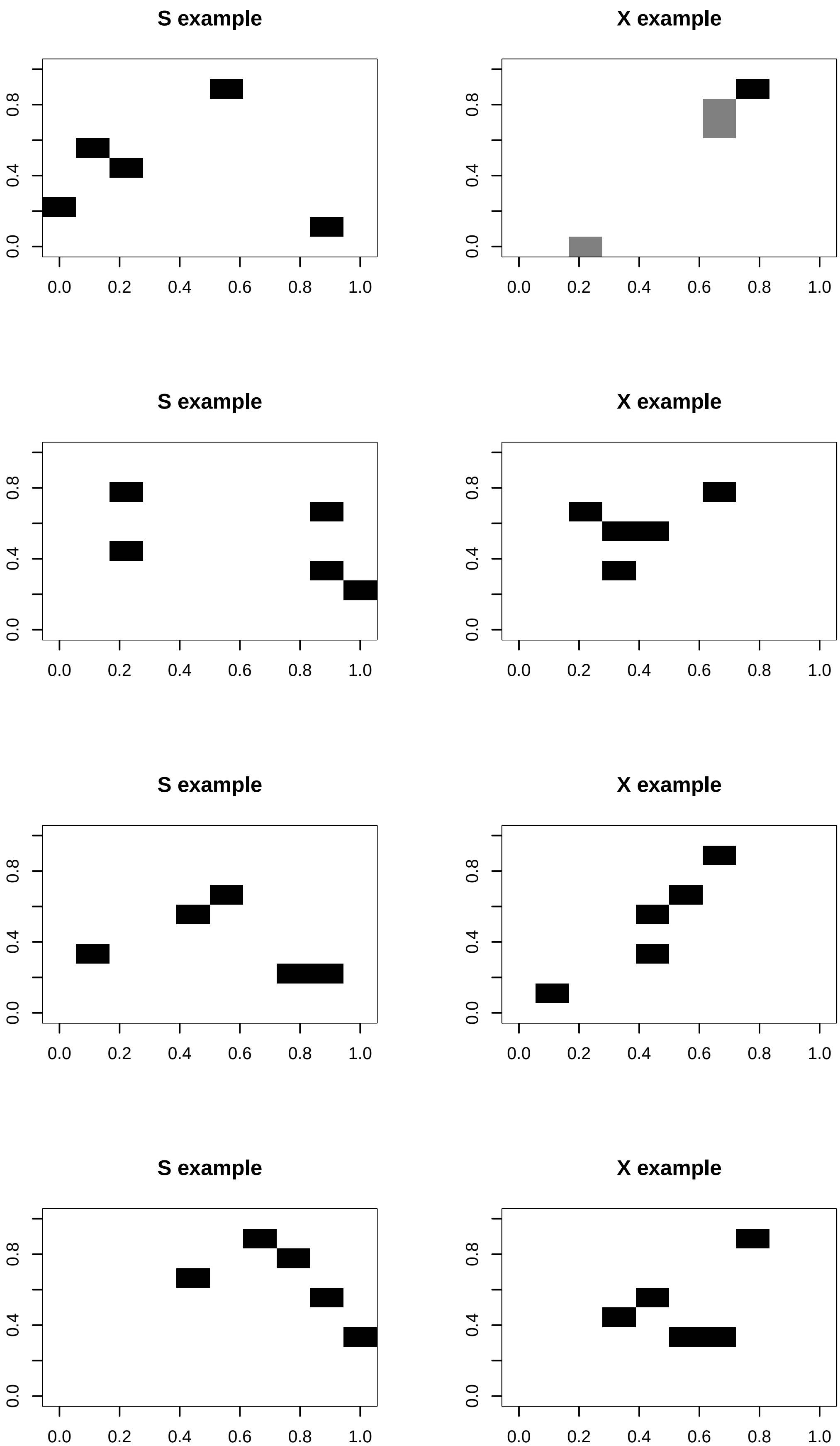
```
letter   s   x
     s 250   0
     x   0 250
```

Training with very limited/noisy examples

This classification is actually perfect, but there was a lot of information available. Let's sample just 5 points out to create the pattern:

```
dataX <- matrix(0, ncol = 100, nrow = 250)
dataS <- matrix(0, ncol = 100, nrow = 250)
letter <- rep(c("x", "s"), each = 250)


par(mfrow = c(4, 2))

for (i in 1:250) {
    x <- rep(0, 100)
    xtmp <- table(sample(1:100, size = 5, prob = as.matrix(xvec$x), replace = T))
    x[as.numeric(names(xtmp))] <- xtmp/max(xtmp)

    s <- rep(0, 100)
    stmp <- table(sample(1:100, size = 5, prob = as.matrix(svec$x), replace = T))
    s[as.numeric(names(stmp))] <- stmp/max(stmp)

    ## plot the first few examples:
    if (i <= 4) {
        image(matrix(s, 10, 10, byrow = T), main = "S example", col = grey(100:0/100))
        image(matrix(x, 10, 10, byrow = T), main = "X example", col = grey(100:0/100))
    }
```

```
    dataX[i, ] <- x
    dataS[i, ] <- s
}
```


S example
X example
S example
X example
S example
X example
S example
X example
0.0
0.4
0.8
0.0
0.2
0.4
0.6
0.8
1.0

```r
data <- rbind(dataX, dataS)



merged <- data.frame(letter = as.factor(letter), data)
model3 <- nnet(letter ~ ., data = merged, size = 2, rarg = 0.1, decay = 1e-04, maxit = 500)
```

```
# weights:  205
initial  value 363.388947
iter  10 value 102.725078
iter  20 value 77.744217
iter  30 value 74.968193
iter  40 value 73.034036
iter  50 value 72.687594
iter  60 value 72.552117
iter  70 value 72.497809
iter  80 value 72.461359
iter  90 value 72.435675
iter 100 value 72.404905
iter 110 value 72.381593
iter 120 value 72.362333
iter 130 value 72.346915
iter 140 value 72.330550
iter 150 value 72.323357
iter 160 value 72.316607
iter 170 value 72.311707
iter 180 value 72.308358
iter 190 value 72.304723
iter 200 value 72.301975
iter 210 value 72.292961
iter 220 value 68.986191
iter 230 value 68.882630
iter 240 value 68.848261
iter 250 value 66.358980
iter 260 value 41.150098
iter 270 value 26.680561
iter 280 value 22.343506
iter 290 value 19.524840
iter 300 value 18.003927
iter 310 value 16.828821
iter 320 value 16.495417
iter 330 value 16.320449
iter 340 value 16.174379
iter 350 value 16.056865
iter 360 value 15.933173
iter 370 value 15.807782
iter 380 value 15.707452
iter 390 value 15.641587
iter 400 value 15.545632
iter 410 value 15.242668
iter 420 value 15.176163
iter 430 value 13.859431
iter 440 value 12.370288
iter 450 value 12.246867
```

```
iter 460 value 12.172793
iter 470 value 12.119114
iter 480 value 12.068077
iter 490 value 12.018581
iter 500 value 11.984600
final  value 11.984600
stopped after 500 iterations
```

```
table(letter, predict(model3, type = "class"))
```

```
letter   s   x
     s 248   2
     x   1 249
```

It still does very well, with a few errors, for very sparse data. In fact, it might be doing TOO well. That is, in a sense, it is picking up on arbitrary but highly diagnostic features. A human observer would never be confident in the outcome, because the information is too sparse, but in this limited world, a single pixel is enough to make a good guess.

Neural Networks and the XOR problem

Neural networks became popular when researchers realized that networks with a hidden layer could solve 'XOR classification problems'. Early on, researchers recognized that a simple perception (2-layer) neural network could be used for AND or OR combinations, but not XOR, as these are not linearly separable. XOR classification maps onto many real-world interaction problems. For example, two safe pharmaceuticals might be dangerous when taken together, and a simple neural network could never detect this state–if one is good, and the other is good, both must be better. An XOR problem is one in which one feature or another (but not both or neither) indicate class membership. In order to perform classification with this logic, a hidden layer is required.

Here is a class defined by an XOR structure:

```
library(MASS)
library(DAAG)
feature1 <- rnorm(200)
feature2 <- rnorm(200)
outcome <- as.factor((feature1 > 0.6 & feature2 > 0.3) | (feature1 > 0.6 & feature2 <
    0.3))
outcome <- as.factor((feature1 * (-feature2) + rnorm(200)) > 0)
```

The linear discriminant model fails to discriminate (at least without an interaction)

```
lmodel <- lda(outcome ~ feature1 + feature2)

confusion(outcome, predict(lmodel)$class)
```

```
Overall accuracy = 0.565

Confusion matrix
       Predicted (cv)
Actual  FALSE  TRUE
  FALSE 0.438 0.562
  TRUE  0.317 0.683
```

```
lmodel2 <- lda(outcome ~ feature1 * feature2)

confusion(outcome, predict(lmodel2)$class)
```

```
Overall accuracy = 0.7

Confusion matrix
       Predicted (cv)
Actual  FALSE  TRUE
  FALSE 0.635 0.365
  TRUE  0.240 0.760
```

Similarly, the neural networks with an empty single layer (skip=TRUE) are not great at discriminating, but with a few hidden nodes they work well.

```
n1 <- nnet(outcome ~ feature1 + feature2, size = 0, skip = TRUE)
```

```
# weights:  3
initial  value 147.166762
final  value 136.677937
converged
```

```
confusion(outcome, factor(predict(n1, type = "class"), levels = c(TRUE, FALSE)))
```

```
Overall accuracy = 0.44

Confusion matrix
       Predicted (cv)
Actual  FALSE  TRUE
  FALSE 0.562 0.438
  TRUE  0.673 0.327
```

```
n2 <- nnet(outcome ~ feature1 + feature2, size = 3, skip = TRUE)
```

```
# weights:  15
initial  value 157.422372
iter  10 value 130.274941
iter  20 value 117.699553
iter  30 value 111.696732
iter  40 value 109.991125
iter  50 value 108.883283
iter  60 value 108.195463
iter  70 value 107.960809
iter  80 value 107.816094
iter  90 value 107.408249
iter 100 value 107.307820
final  value 107.307820
stopped after 100 iterations
```

```
confusion(outcome, factor(predict(n2, type = "class"), levels = c(FALSE, TRUE)))
```

```
Overall accuracy = 0.695

Confusion matrix
       Predicted (cv)
Actual  FALSE  TRUE
  FALSE 0.594 0.406
  TRUE  0.212 0.788
```

Because we have the XOR structure, the simple neural network without a hidden layer is essentially LDA, and in fact often gets a similar level of accuracy (53%).

IPhone Data using the NNet

To model the iphone data set, we need to decide on how large of a model we want. We need to also remember that training a model like this is not deterministic–every time we do it the situation will be a little different. Because we have just two classes, maybe a 2-layer hidden network would work.

By fitting the model several times, we can see that it performs differently every time. At times, the model does reasonably well. This does about as well as any of the best classification models. Curiously, this particular model has a bias toward Android accuracy.

```
phone <- read.csv("data_study1.csv")
phone$Smartphone <- (as.factor(phone$Smartphone))
phone$Gender <- as.numeric(as.factor(phone$Gender))
set.seed(100)
phonemodel <- nnet(Smartphone ~ ., size = 3, data = phone)
```

```
# weights:  43
initial  value 368.669183
iter  10 value 351.629336
iter  20 value 328.109320
iter  30 value 319.345472
iter  40 value 317.075499
iter  50 value 314.039955
iter  60 value 313.953479
final  value 313.953382
converged
```

```
confusion(phone$Smartphone, factor(predict(phonemodel, newdata = phone, type = "class",
    levels = c("Android", "iPhone"))))
```

```
Overall accuracy = 0.652

Confusion matrix
         Predicted (cv)
Actual    Android iPhone
  Android   0.603  0.397
  iPhone    0.313  0.687
```

Other times, it does poorly: Here, it calls everything an iPhone:

```
set.seed(101)
phonemodel2 <- nnet(Smartphone ~ ., data = phone, size = 2)
```

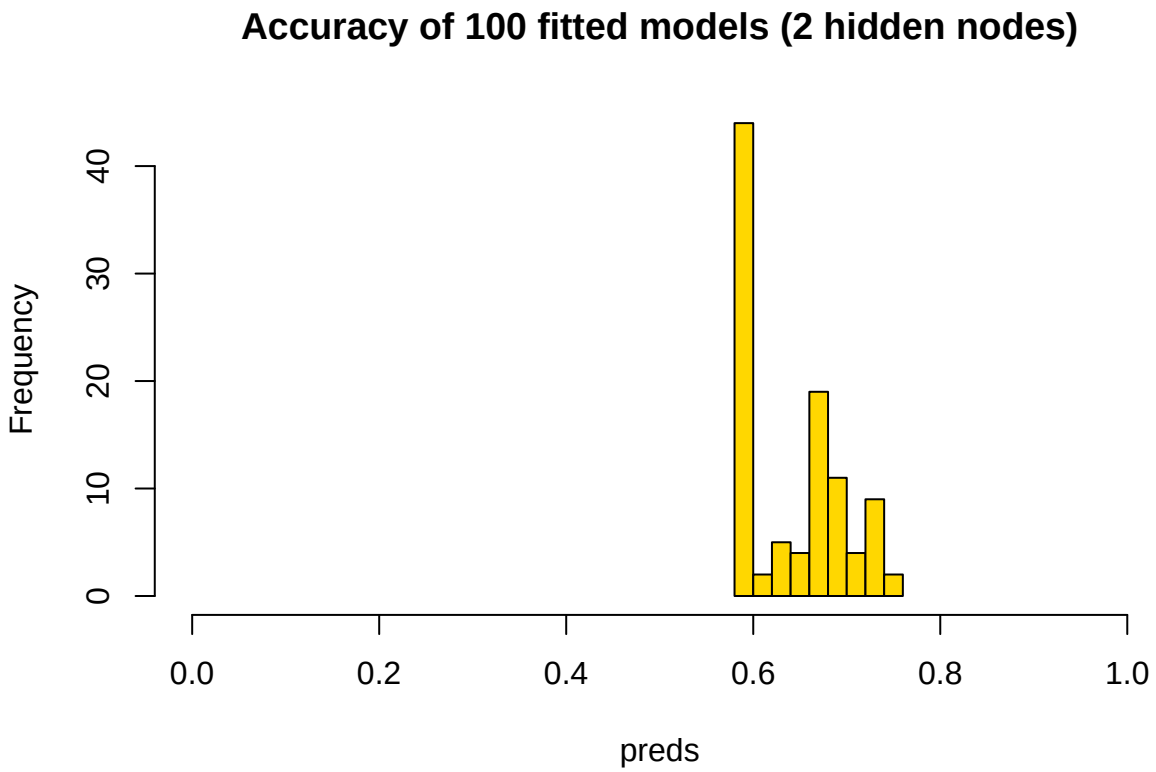
```
# weights:  29
initial  value 457.289985
final  value 358.808683
converged
```

```
confusion(phone$Smartphone, factor(predict(phonemodel2, newdata = phone, type = "class"),
    levels = c("Android", "iPhone")))
```

```
Overall accuracy = 0.586

Confusion matrix
         Predicted (cv)
Actual    Android iPhone
  Android       0      1
  iPhone        0      1
```

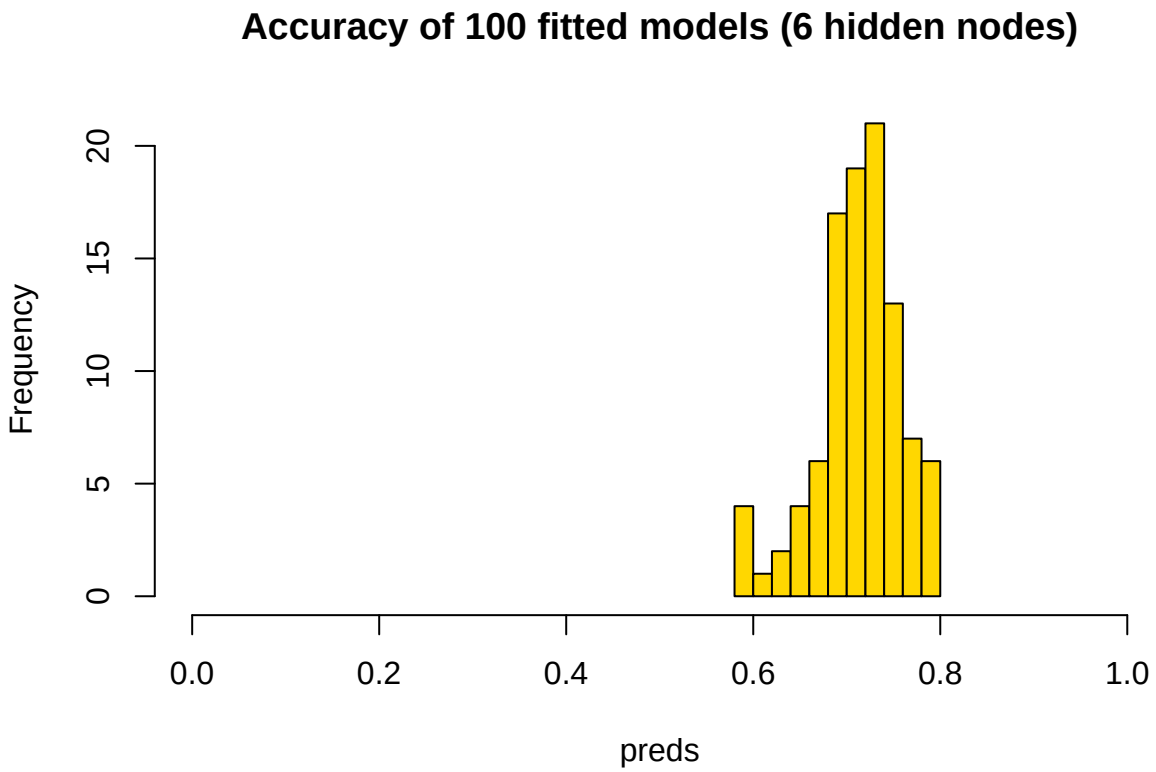
In this case, it called everything in iPhone, resulting in 58.6% accuracy. Like many of our approaches, we are doing heuristic optimization and so may end up in local optima. It is thus useful to run the model many several times and look for the best model. Running this many times, the best models seem to get around 370-390 correct, which is in the low 70% for accuracy.

```
preds <- matrix(0, 100)
for (i in 1:100) {
    phonemodel3 <- nnet(Smartphone ~ ., data = phone, size = 2)
    preds[i] <- confusion(phone$Smartphone, factor(predict(phonemodel3, newdata = phone,
        type = "class"), levels = c("Android", "iPhone")))$overall

}

hist(preds, col = "gold", main = "Accuracy of 100 fitted models (2 hidden nodes)",
    xlim = c(0, 1))
```



Do we do any better, on average, with more hidden nodes?

```
preds <- matrix(0, 100)
for (i in 1:100) {
    phonemodel3 <- nnet(Smartphone ~ ., data = phone, size = 6)
    preds[i] <- confusion(phone$Smartphone, factor(predict(phonemodel3, newdata = phone,
        type = "class"), levels = c("Android", "iPhone")))$overall

}

hist(preds, col = "gold", main = "Accuracy of 100 fitted models (6 hidden nodes)",
    xlim = c(0, 1))
```



This seems to be more consistent, and do better overall–usually above 70% accuracy. But like every model we have examined, the best models are likely to be over-fitting, and getting lucky at re-predicting their own data. It would also be important to implement a cross-validation scheme. There is no built-in cross-validation here, but you can implement one using subset functions.

```
train <- rep(FALSE, nrow(phone))
train[sample(1:nrow(phone), size = 300)] <- TRUE
test <- !train

phonemodel2 <- nnet(Smartphone ~ ., data = phone, size = 6, subset = train)
```

```
# weights:  85
initial  value 204.327276
iter  10 value 193.352724
iter  20 value 166.957659
iter  30 value 136.845896
iter  40 value 126.652225
iter  50 value 118.844038
iter  60 value 114.601579
iter  70 value 112.362718
iter  80 value 109.187074
iter  90 value 106.922061
iter 100 value 104.486444
final  value 104.486444
stopped after 100 iterations
```

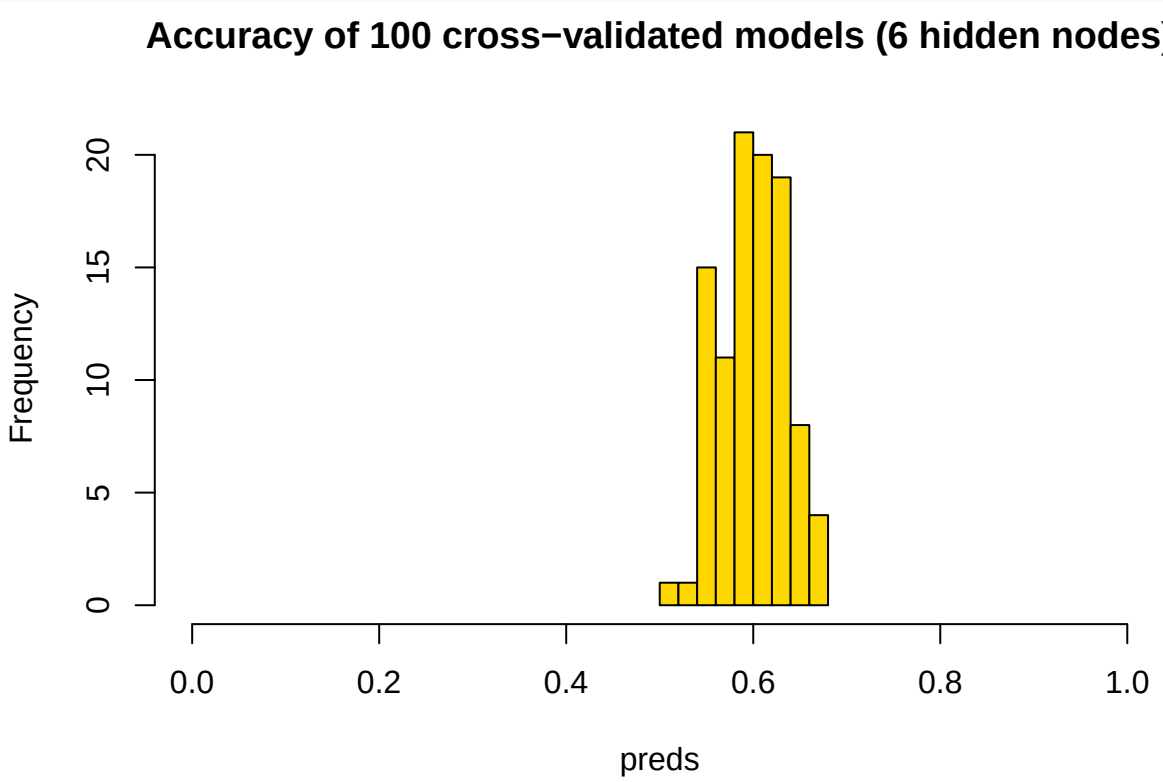
```
confusion(phone$Smartphone[test], predict(phonemodel2, newdata = phone[test, ], type = "class"))
```

```
Overall accuracy = 0.563

Confusion matrix
```

```
         Predicted (cv)
Actual    Android iPhone
  Android   0.340  0.660
  iPhone    0.273  0.727
```

We can try this 100 times and see how well it does on the cross-validation set:

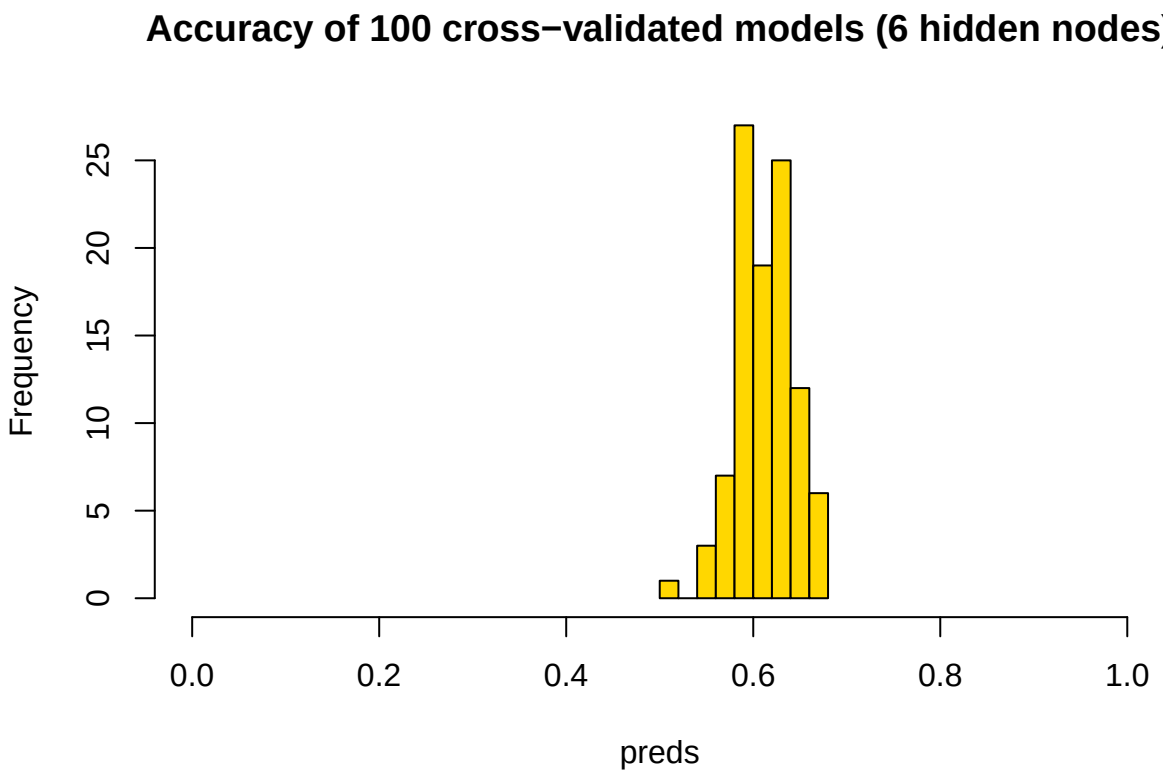
```
preds <- rep(0, 100)

for (i in 1:100) {
    train <- rep(FALSE, nrow(phone))
    train[sample(1:nrow(phone), size = 300)] <- TRUE
    test <- !train

    phonemodel2 <- nnet(Smartphone ~ ., data = phone, size = 6, subset = train)
    preds[i] <- confusion(phone$Smartphone[test], factor(predict(phonemodel2, newdata = phone[test,
        ], type = "class"), levels = c("Android", "iPhone")))$overall

}

hist(preds, col = "gold", main = "Accuracy of 100 cross-validated models (6 hidden nodes)",
    xlim = c(0, 1))
```



The cross-validation scores are typically a bit lower, with models getting around 60% on average, and up to 70% for the best. Now that we have built this, we could use average cross-validation accuracy to help select variables for exclusion. Here, let's just test the predictors we have found previously to be fairly good:

```
preds <- rep(0, 100)

for (i in 1:100) {
    train <- rep(FALSE, nrow(phone))
```

```
    train[sample(1:nrow(phone), size = 300)] <- TRUE
    test <- !train

    phonemodel2 <- nnet(Smartphone ~ Gender + Avoidance.Similarity + Phone.as.status.object +
        Age, data = phone, size = 6, subset = train)
    preds[i] <- confusion(phone$Smartphone[test], factor(predict(phonemodel2, newdata = phone[test,
        ], type = "class"), levels = c("Android", "iPhone")))$overall

}

hist(preds, col = "gold", main = "Accuracy of 100 cross-validated models (6 hidden nodes)",
    xlim = c(0, 1))
```



Most of these are better than chance, and it seems to do about as well as the full set of predictors as well.

The `neuralnet` library

The neuralnet library is perhaps a more flexible implementation, with multiple hidden layers. Here we have 2 hidden layers with two nodes each. When fitting it, it seemed to get stuck frequently and not converge or have some other error, but it does give a nice graphical visualization of the network. It seems to used more advanced backprop algorithms and activation functions. Here we have two hidden layers with two nodes each. The network does not make a lot of sense, but the model fails to converge under many larger network conditions.

```
library(neuralnet)
set.seed(10313)
train <- rep(FALSE, nrow(phone))
train[sample(1:nrow(phone), size = 300)] <- TRUE
test <- !train
```

```
phonemodel3 <- neuralnet(Smartphone ~ ., hidden = c(2, 2), data = phone[train, ])

pred <- apply(predict(phonemodel3, newdata = phone[test, ]), 1, which.max)
ptest <- phone[test, ]
acc <- (sum(ptest[pred == 1, ]$Smartphone == "Android") + sum(ptest[pred == 2, ]$Smartphone ==
    "iPhone"))/sum(test)
print(acc)
```

```
[1] 0.5764192
```

```
plot(phonemodel3, rep = "best")
```