Neural Network on Poker Hands

Petru Pascu

Introduction

The aim of this project is to provide some insight on the statistical instruments used in a neural network, to build one from the ground up and to apply it on a poker hands data set. The task of the algorithm will be to predict the poker hand given five cards. Note that this can be easily done in a deterministic way, but we are interested in the statistical approach. We don't intend to obtain good predictions, but rather to explore the mechanism of a neural network. For the mathematical part, we mainly studied [1] and [2] and for implementation we followed [3] and [4]. The data can be found at [5].

The Perceptron Algorithm

We find it useful to start by looking at the perceptron algorithm. It corresponds to a two-class model with an input vector $\mathbf{x}$, used to construct a linear model of the form

$$y(\mathbf{x}) = f(\mathbf{w}^T\mathbf{x})$$

where the nonlinear *activation function* $f(\cdot)$ is given by a step function of the form

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

The vector $\mathbf{x}$ typically includes a bias component $x_0 = 1$. Together with the vector $\mathbf{x}$, we also have the labels or target values in our data set: $t = +1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$, which match the choice of the activation function.

In order to determine the parameters $\mathbf{w}$, the idea of minimizing an error function that we know from linear regression, applies here as well. The question is how do we choose this function? We note that we are seeking a weight vector $\mathbf{w}$ such that patterns $\mathbf{x}_n$ in class $\mathcal{C}_1$ will have $\mathbf{w}^T\mathbf{x}_n > 0$, whereas patterns in class $\mathcal{C}_2$ have $\mathbf{w}^T\mathbf{x}_n < 0$. Using the $t \in \{-1, +1\}$ target coding scheme it follows that we would like all patterns to satisfy $\mathbf{w}^T\mathbf{x}_n t_n > 0$. Our error function, known as the *perceptron criterion*, will associate zero error with any pattern that is correctly classified, whereas for a misclassified pattern $\mathbf{x}_n$ it tries to minimize the quantity $-\mathbf{w}^T\mathbf{x}_n t_n$. The perceptron criterion is therefore given by

$$E_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^T x_n t_n$$

where $\mathcal{M}$ denotes the set of all misclassified patterns.

To minimize this function, we apply the gradient descent algorithm, which we will also use in the training of our neural network. The change in the weight vector $\mathbf{w}$ is given by

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E_P(\mathbf{w}) = \mathbf{w}^t + \eta \mathbf{x}_n t_n$$

where η is the learning rate parameter and t is an integer that indexes the steps of the algorithm. Because the perceptron function $y(\mathbf{x}, \mathbf{w})$ is unchanged if we multiply $\mathbf{w}$ by a constant, we can set the learning rate parameter η equal to 1 without loss of generality.

The perceptron algorithm has a simple interpretation, as follows. We cycle through the training patterns in turn, and for each pattern $\mathbf{x}_n$ we evaluate the perceptron function $y(\mathbf{x}_n)$ given above. If the pattern is correctly classified, then the weight vector remains unchanged, whereas if it is incorrectly classified, then for class $\mathcal{C}_1$ we add the vector $\mathbf{x}_n$ onto the current estimate of weight vector $\mathbf{w}$ while for class $\mathcal{C}_2$ we subtract the vector $\mathbf{x}_n$ from $\mathbf{w}$.

Interpretation and Generalization

Reinterpreting some of the concepts in the perceptron method, we are able to generalize to multiclass models.

We want to change the activation function we saw above such that it gives us a probabilistic interpretation, as follows: f is a function with values in $(0, 1)$ so that the "more negative" $\mathbf{w}^T\mathbf{x}$ is, the less likely is that the $x's$ belong to class $\mathcal{C}_1$ and the more likely is that they belong to class $\mathcal{C}_2$ and vice versa if $\mathbf{w}^T\mathbf{x}$ is positive. So we want a function that maps the real axis smoothly (if possible) to the interval $(0, 1)$. One choice is the *sigmoid function*

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

According to our interpretation,

$$\sigma(a) = p(\mathcal{C}_1|\mathbf{x}) = 1 - p(\mathcal{C}_2|\mathbf{x})$$

This model would work well for two-class classification. What if we have to classify objects from $K > 2$ classes?

Notice that, at the same time, according to Bayes' theorem,

$$p(\mathcal{C}_1|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} = \frac{1}{1 + e^{-a}} = \sigma(a)$$

where

$$a = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} = \ln \frac{p(\mathcal{C}_1|\mathbf{x})}{p(\mathcal{C}_2|\mathbf{x})}$$

Recall that, in our model, a is a linear function of $\mathbf{x}$ with weights $\mathbf{w}^T$. In the $K > 2$ case,

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_{j=1}^{K-1} p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} = \frac{e^{a_k}}{1 + \sum_{j=1}^{K-1} e^{a_j}}$$

for each $k \in \{1, ..., K-1\}$, where

$$a_k = \ln \frac{p(\mathcal{C}_k|\mathbf{x})}{p(\mathcal{C}_K|\mathbf{x})}$$

and

$$p(\mathcal{C}_K|\mathbf{x}) = \frac{1}{1 + \sum_{j=1}^{K-1} e^{a_j}}$$

or

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{e^{a_k}}{\sum_{j=1}^{K} e^{a_j}}$$

with

$$a_k = \ln p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)$$

for all $k \in \{1, ..., K\}$. The a_k's will be linear functions of the input vector $\mathbf{x}$. This function is known as the *normalized exponential* or the *softmax function.*

Neural Networks

The idea of neural networks is to apply weighted sums and activation functions several times. First, we construct M linear combinations of the input variables $x_1, ..., x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

where $j = 1, ..., M$ and the superscript (1) indicates that the corresponding parameters are in the first "layer" of the network. We shall refer to the parameters $w_{ji}^{(1)}$ as *weights* and the parameters $w_{j0}^{(1)}$ as *biases*. The quantities a_j are known as *activations*. Each of them is then transformed using a differentiable, nonlinear *activation function* $h(\cdot)$ to give

$$z_j = h(a_j).$$

These quantities are called *hidden units*. The nonlinear functions $h(\cdot)$ are generally chosen to be the sigmoid, the hyperbolic tangent *tanh* or the *rectified linear unit*, ReLU for short, given by $\max(0, x)$. The hidden units are again combined to give *output unit activations*

$$a_k = \sum_{i=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

where $k = 1, ..., K$ and K is the total number of outputs. This transformation corresponds to the second layer of the network, and again $w_{k0}^{(2)}$ are bias parameters. Finally, the output unit activations are transformed using an appropriate activation function (which, in the $K > 2$ classes case, is the softmax discussed above) to give a set of network outputs y_k. The overall network function will take the form

$$y_k(\mathbf{x}, \mathbf{w}) = softmax(\sum_{i=1}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}) + w_{k0}^{(2)})$$

where the set of all weight and bias have been grouped together into a vector $\mathbf{w}$. Of course, one can choose more than two layers to form the neural network.

Network Training

We refer back to the idea of determining the weights by minimizing an appropriate error function (also known as loss function). The process of finding this minimum (actually the argmin) is called network training. We are given a training set comprising a set of input vectors $\{\mathrm{x}_n\}$, where $n = 1, ..., N$, together with a corresponding set of target vectors $\{\mathbf{t}_n\}$, where $\mathbf{t}_n$ for x_n belonging to class $\mathcal{C}_k$ is a binary vector with all elements zero except for element k, which equals one. If the entries of the vector $\{\mathbf{t}_n\}$ were independent, that is, if every entry was either 1 or 0 independent of the others, then, since t_{nk} is Bernoulli distributed, the conditional distribution would have been

$$p(\mathbf{t}_n|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k} [1 - y_k(\mathbf{x}, \mathbf{w})]^{1-t_k}$$

But as we said, the entries of a $\mathbf{t}_n$ are dependent on each other (if one of them is 1, the other have to be 0) and the conditional distribution looks like this

$$p(\mathbf{t}_n|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k}$$

How do we figure out this formula? In this case, we have to think about $\mathbf{t}$ as a variable in itself and ask what is the probability that $\mathbf{t}$ is a vector of all zeros, except the k^{th} entry, which is 1? This is nothing else but

the probability that the k^{th} entry is 1 (represented by $y_k(\mathbf{x}, \mathbf{w})$), as the other are forced to be 0, and this is exactly what the above formula expresses. The corresponding likelihood function, which is to be maximized, is then

$$p(\mathbf{T}|\mathbf{w}) = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k}$$

where $\mathbf{T}$ denotes the matrix of all vectors $\mathbf{t}_n$. But we are looking for an error function, that we would like to minimize. The standard trick here is to take the negative logarithm of the likelihood function, so that maximizing the latter is equivalent to minimizing the former. We obtain the so-called *cross-entropy error function*

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w})$$

Error Backpropagation

To minimize the error, we would like to make use of the gradient descent algorithm, which we have seen above. The idea of the algorithm is simple: as the gradient of the error tells us the direction in which the error grows the most (the direction of the steepest ascent), we should go in the opposite direction (that of the steepest descent) and that is why we subtract the gradient scaled by a number η called *learning rate*. The question is how do we find an efficient technique for evaluating the gradient. This can be achieved using a local message passing scheme in which information is sent alternately forwards and backwards through the network and is known as *error backpropagation*, or sometimes simply as *backprop*. We derive the backpropagation algorithm for a general network with arbitrary differentiable nonlinear activation functions and a broad class of error functions. We will apply the resulting formulae on our choice of network architecture and activation functions for our poker hands data set.

Many error functions of practical interest, like the one we derived above, comprise a sum of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

We shall consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function.

In a *feed-forward* network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i + b_j \tag{1}$$

where z_i is the activation of a unit, or input, that sends a connection to unit j, w_{ji} is the weight associated with that connection and b_j the bias. The sum in (1) is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h(a_j) \tag{2}$$

Note that one or more of the variables in the sum (1) could be an input, and similarly, the unit j in (2) could be an output.

For each pattern in the training set, we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (1). This process is often called *forward propagation* because it can be regarded as a forward flow of information through the network.

Now we consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input pattern n. However, in order to keep the notation uncluttered, we shall omit the subscript n from the network variables. First we note that E_n depends on

the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{3}$$

We now introduce a useful notation

$$\delta_j = \frac{\partial E_n}{\partial a_j} \tag{4}$$

where the $\delta' s$ are often referred to as *errors* for reasons we shall see shortly. Using (1), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{5}$$

Substituting (4) and (5) into (3), we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{6}$$

Similarly, for biases we obtain

$$\frac{\partial E_n}{\partial b_j} = \delta_j \tag{7}$$

Equation (6) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight and the gradient with respect to the biases is given by the errors. Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (6) and (7).

Let's compute the $\delta' s$. In our case, we use the softmax function as the activation function for the last layer, therefore

$$\begin{aligned} E &= -\sum_{j=1}^{K} t_j \ln y_j = -\sum_{j=1}^{K} t_j \ln z_j = \\ &= -\sum_{j=1}^{K} t_j \ln softmax(a_j) = -\sum_{j=1}^{K} t_j \ln \frac{e^{a_j}}{\sum_i e^{a_i}} = \\ &= -\sum_{j=1}^{K} t_j (a_j - \ln(\sum_i e^{a_i})) \end{aligned}$$

hence

$$\begin{aligned} \delta_k = \frac{\partial E}{\partial a_k} &= -\sum_{j=1}^{K} t_j (\chi_{j=k} - \frac{e^{a_k}}{\sum_i e^{a_i}}) = \\ &= -\sum_{j=1}^{K} t_j (\chi_{j=k} - z_k) = \\ &= z_k \sum_{j=1}^{K} t_j - \sum_{j=1} t_j \chi_{j=k} = \\ &= z_k - t_k \end{aligned}$$

These are the $\delta' s$ for the output units. To evaluate the $\delta' s$ for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{8}$$

where the sum runs over all units k to which unit j sends connections. Note that the units labelled k could include other hidden units or output units. In writing down (8), we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by (4) into (8), and make use of (1) and (2), we obtain the following the following *backpropagation* formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{9}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the $\delta' s$ backwards from units higher up in the network. Note that the summation in (9) is take over the first index on w_{kj} (corresponding to backward propagation of information through the network), whereas in the forward propagation equation (3) it is taken over the second index. Because we already know the values of the $\delta' s$ for the output units, it follows that by recursively applying (9) we can evaluate the $\delta' s$ for all of the hidden units in a feed-forward network.

The backpropagation procedure can be therefore summarized as follows.

1. Apply an input vector $\mathbf{x}_n$ tot he network and forward propagate through the network using (1) and (2) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (4).
3. Backpropagate the $\delta' s$ using (9) to obtain the $\delta' j$ for each hidden unit in the network.
4. Use (6) and (7) to evaluate the required derivatives.

The gradient of the total error E can then be obtained by repeating the above steps for each pattern in the training set and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $h(\cdot)$. The derivation is easily generalized, however, to allow different units to have individual activation functions, simply by keeping track of which form of $h(\cdot)$ goes with which unit.

Implementation

In this section we will create a neural network in R, we will train and test it on our data set. We will also provide a scatter plot in order to visualize the results. The training set contains 25009 five-cards examples, each card being specified by its suit (this attribute can take values from 1 to 4, corresponding to hearts, spades, diamonds, clubs, respectively) and its rank (this attribute can take values from 1 to 13). Each example is labeled by a number between 0 and 9, corresponding to the poker hand that the five cards form.

```
#Read the data from the file
hands<-read.csv("poker-hand-training.txt", sep = ",")

#Shuffle the data
set.seed(234)
rows <- sample(nrow(hands))
hands <- hands[rows,]

#Rename the columns
colnames(hands) <- c("S1","C1","S2","C2","S3","C3","S4","C4","S5","C5","Hand")
```

Before we build the neural network, we have to pre-process the data. We associate with each suit a vector with 4 entries, all 0 except the one corresponding to the given suit, which will be 1. We do the same for the ranks and for the labels. We obtain $4 \cdot 5 + 13 \cdot 5 = 85$ input and 10 output units.

```r
#Create a 25009X85 data frame called X
X = data.frame(matrix(0,nrow(hands),85))

#Rename the columns
colnames(X) <- c("S1","S1","S1","S1","C1","C1","C1","C1","C1","C1","C1","C1","C1",
                 "C1","C1","C1","C1","S2","S2","S2","S2","C2","C2","C2","C2","C2",
                 "C2","C2","C2","C2","C2","C2","C2","C2","S3","S3","S3","S3","C3",
                 "C3","C3","C3","C3","C3","C3","C3","C3","C3","C3","C3","S4",
                 "S4","S4","S4","C4","C4","C4","C4","C4","C4","C4","C4","C4","C4",
                 "C4","C4","C4","S5","S5","S5","S5","C5","C5","C5","C5","C5","C5",
                 "C5","C5","C5","C5","C5","C5","C5")

#Modify X as described
j = 0
for(k in 1:nrow(X)){

  for(i in 1:ncol(hands) - 1){

    if(i == 1){
      j = 0
    }else{

      if(i %% 2 == 0){
        j = j + 4
      }else{
        j = j + 13
      }

    }

    ind = hands[k,i]
    X[k,j + ind] = 1

  }

}

#Create a 25009X10 data frame called t
t = data.frame(matrix(0,nrow(hands),10))

#Rename the columns
colnames(t) <- c("n","op","tp","tk","s","f","fh","fk","sf","rf")

#Modify t as described
for (k in 1 : nrow(hands)){
  kind = hands$Hand[k]
  t[k,kind + 1] = 1
}

#Convert t and X to matrices
X <- as.matrix(X)
t = as.matrix(t)
```

We define a numerically stable form of the softmax function, which we will use in the training process.

Our neural network has 85 input and 10 output variables as we said and one hidden layer with 17 hidden units. We also tried more hidden units and hidden layers, but obtained worse results. We use ReLU as activation function on the hidden layer. We set the learning rate to 1, although we had to change it to a smaller value at the end of the training process. Note that the loss here is a bit different from the one we derived above: it's scaled by $\frac{1}{n}$ (it could be regarded as the mean of the errors from each example). The scaling makes the gradient descent algorithm more stable. Without it, the gradient would blow up or vanish, in which case an adaptive learning rate could help.

```
#Numerically stable form of softmax
softmax = function(x){
  for(k in 1:nrow(x)){
    x[k,] = exp(x[k,] - max(x[k,]))/sum(exp(x[k,] - max(x[k,])))
  }
  return (x)
}

#Create the neural network. noit is the number of iterations,
#h is the number of hidden units
nnpoker <- function(X, t, learningrate = 1, h = 17, noit = 70000){

  #number of input units
  noinp = ncol(X)
  #number of output units/number of classes
  nocl = ncol(t)
  #size of data
  n = nrow(X)

  #Initialize the weight matrices randomly from a normal distribution
  #and the biases with 0
  set.seed(234)
  W1 = matrix(rnorm(noinp*h), nrow = noinp)
  B1 <- matrix(0, nrow = 1, ncol = h)
  W2 = matrix(rnorm(h*nocl), nrow = h)
  B2 <- matrix(0, nrow = 1, ncol = nocl)


  for (j in 1:noit){

    #Build the hidden layer
    #pmax(0,x) does the job of the ReLU function
    #byrow = TRUE means that the matrix is filled by rows
    hlayer <- pmax(0,X%*%W1 + matrix(rep(B1,n), nrow = n, byrow = TRUE))
    #Convert the hidden layer to matrix
    hlayer <- matrix(hlayer, nrow = n)

    #Compute the outputs
    Y <- softmax(hlayer%*%W2 + matrix(rep(B2,n), nrow = n, byrow = TRUE))

    #Compute cross-entropy error function (loss function)
    lnY <- -log(Y)
    m <- t*lnY
    m[t==0] <- 0  #Avoid infinity multiplied by 0
    loss <- sum(m)/n
```

```
    #Keep track of the error
    if (j%%100 == 0 | j == noit){
      print(paste("Iteration", j,': error', loss))
    }

    #Perform backpropagation
    delta2 <- (Y - t)/n
    GradW2 <- t(hlayer)%*%delta2  #Gradient of error wrt. the weights
    GradB2 <- colSums(delta2)     #Gradient of error wrt. the biases
    delta1 <- delta2%*%t(W2)
    delta1[hlayer <= 0] <- 0      #Here is actually  ReLU's derivative in action
    GradW1 <- t(X)%*%delta1       #Gradient of error wrt. the weights
    GradB1 <- colSums(delta1)     #Gradient of error wrt. the biases

    #Keep track of the norm of the gradient
    nm = sum((GradW1)^2) + sum((GradW2)^2) + sum((GradB1)^2) + sum((GradB2)^2)
    nm = sqrt(nm)
    if (j%%100 == 0 | j == noit){
      print(paste("Iteration", j,': norm of gradient', nm))
    }

    #Update weights and biases
    W1 <- W1-learningrate*GradW1
    B1 <- B1-learningrate*GradB1
    W2 <- W2-learningrate*GradW2
    B2 <- B2-learningrate*GradB2

    #In case the error becomes small enough
    if(loss <= 0.001){
      break
    }

  }

  return(list(W1,B1,W2,B2))

}

model <- nnpoker(X,t,learningrate = 1, h = 17, noit = 70000)
```

Next we would like to test our model. The values for the weights and biases are obtained from the above training process. However, we also provide them separately, in case someone wants to test the model without going through the training process beforehand. We reached a value of about 0.462 for the error and 0.003 for the norm of the gradient, which makes us believe that we reached a local minimum of the error function.

```
#Read the testing data set
testhands <- read.csv("poker-hand-testing.txt", sep = ",")

#Extract a sample of 5000 examples
set.seed(101)
smp <- as.matrix(testhands[sample(nrow(testhands),5000),])
colnames(smp) <- c("S1","C1","S2","C2","S3","C3","S4","C4","S5","C5","Hand")

#Pre-process the data
```

```
Xtest = matrix(0,nrow(smp),85)
j = 0
for(k in 1:nrow(smp)){

     for(i in 1:ncol(smp) - 1){

       if(i == 1){
         j = 0
       }else{

         if(i %% 2 == 0){
           j = j + 4
         }else{
           j = j + 13
         }
       }
       ind = smp[k,i]
       Xtest[k,j + ind] = 1

     }

}

#Read the values for the weights and biases if testing without training is preferred
require(tseries) #Package needed for the function 'read.matrix'
W1<-read.matrix("Weights1.txt", sep = " ", header = FALSE)
W2<-read.matrix("Weights2.txt", sep = " ", header = FALSE)
B1<-read.matrix("Biases1.txt", sep = " ", header = FALSE)
B2<-read.matrix("Biases2.txt", sep = " ", header = FALSE)

#If the training was performed, obtain the weights and biases
#W1 <- model[[1]]
#B1 <- model[[2]]
#W2 <- model[[3]]
#B2 <- model[[4]]

prediction <- function(Xtest,W1,B1,W2,B2){

  noinp = ncol(Xtest) #number of input units
  nocl = ncol(t) #number of classes
  n = nrow(Xtest) #size of data

  hlayer <- pmax(0,Xtest%*%W1 + matrix(rep(B1,n), nrow = n, byrow = TRUE))
  hlayer <- matrix(hlayer, nrow = n)
  Y <- softmax(hlayer%*%W2 + matrix(rep(B2,n), nrow = n, byrow = TRUE))

  #Interpret the values in the output units as probabilities and convert accordingly:
  #the entry with the highest value is the predicted class
  classes = matrix(0, nrow = nrow(Y), 1)
    for(k in 1:nrow(classes)){
      classes[k] = which.max(Y[k,]) - 1
    }
```

```
  return(classes)

}

#Compute the accuracy of the test
labels = as.matrix(smp[,11])
predictedclasses <- prediction(Xtest,W1,B1,W2,B2)
#predictedclasses == labels is a vector with TRUE and FALSE values. The function
# 'mean' returns the number of TRUE values divided by the total number
print(paste('training accuracy:', mean(predictedclasses == labels)))
```

We obtained a training accuracy of about 80%, which is not bad considering that we were able to train our network until the value of 0.462 of the error, which is high. However, as we will see, for our specific problem, this relatively high accuracy does not help very much. As a last part of our project, we would like to visualize our results with the help of a scatter plot. We have 11-dimensional data that we would like to represent in a 2-dimensional plane. Therefore, we use *classical multidimensional scaling* or *cmd* for short. We multiply the output variable by a convenient factor of 6 in order to isolate the classes in clusters.

```
smp[,11] = 6*smp[,11]
#Distance matrix
d <- dist(smp)
#Multidimensional scaling
scaledsmp = cmdscale(d,k = 2)
x = scaledsmp[,1]
y = scaledsmp[,2]

#Create a data frame containing the obtained x and y variables, together with the labels
data = data.frame(x,y,smp[,11])
colnames(data)  = c("x","y","Classes")

#Add a column that translates the values of the classes
fun <- function(x){if (x == 0) {"No hand"} else if (x==6){"One pair"}
else if (x == 12){"Two pairs"}else if (x == 18){"Three of a kind"}
else if (x == 24){"Straight"}else if (x == 30){"Flush"}
else if (x == 36){"Full House"} else if (x == 42){"Four of a kind"}
else if (x == 48){"Straight Flush"}else if (x == 54){"Royal Flush"}}
data$Hands <- mapply(fun, data$Classes)

#See Hands as a categorical variable. We need this conversion because we want
#a discrete color palette, not a continuous one
data$Hands <- factor(data$Hands)

#Add a column with the predicted classes
data$predictedclasses <- predictedclasses

#Translate the values of the predicted classes
fun <- function(x){ if (x == 0) {"No hand"} else if (x==1){"One pair"}
else if (x == 2){"Two pairs"}else if (x == 3){"Three of a kind"}
else if (x == 4){"Straight"}else if (x == 5){"Flush"}
else if (x == 6){"Full House"}else if (x == 7){"Four of a kind"}
else if (x == 8){"Straight Flush"}else if (x == 9){"Royal Flush"}}
data$PredictedHands <- mapply(fun, data$predictedclasses)

#Plot the data
```

```
require(ggplot2)
ggplot(data = data, aes(x = x, y = y,color = PredictedHands, shape = Hands)) +
   geom_point()+
   scale_shape_manual(values = c(1:10))
```

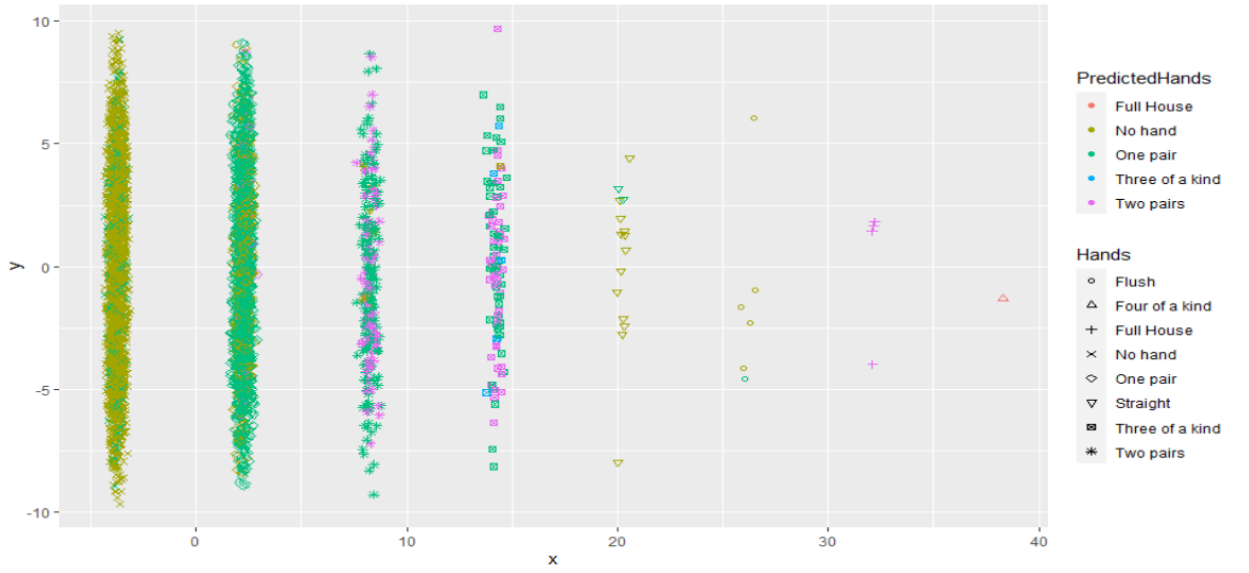

Figure 1: True and predicted poker hands

In this plot, the shape tells us the true class and the color tells us the predicted class. Looking at the legend, we can figure out the fails of our model. We see that for the classes "No hand", "One pair" or "Two pairs", the number of fails is still acceptable, while the majority of hands corresponding to the rest of the classes are mispredicted. This happens because the classes are far from being uniformly distributed over our data set, hence for the classes that occur rarely there are few observations to train the network. For example, in the training set there are 12493 instances of "No hand", 10599 of "One pair" and 1206 of "Two pairs", compared to 6, 5 and 5 of "Four of a kind", "Straight flush" and "Royal flush", respectively. The distribution is similar in the testing set and this is why not all the classes are represented in the plot.

Conclusion

We expected that the neural network would work better with two or more hidden layers, which was not the case. We also tried to increase or decrease the number of hidden units, but the results have worsened. The accuracy we obtained is good relative to the simple instruments we used for training and it could have led to better predictions on a data set with uniformly distributed classes. The experience of creating a neural network from the ground up is helpful for a good understanding of the machinery behind it. It brings satisfaction, but it is not recommended from a practical point of view, since there are a lot of built-in packages that perform much better than our model.

References

[1] Christopher M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[2] Trevor Hastie, Robert Tibshirani, Jerome Friedman, The Elements of Statistical Learning. Data Mining, Inference, and Prediction, Springer, 2016.

[3] (n.d.). https://www.kaggle.com/russwill/build-your-own-neural-network-in-r.

[4] (n.d.). https://medium.com/@akashg/predicting-poker-hand-using-neural-networks-83ed7d0bfc6a.

[5] (n.d.). https://archive.ics.uci.edu/ml/datasets/Poker+Hand.

[6] (n.d.). https://doug919.github.io/notes-on-backpropagation-with-cross-entropy/.

[7] (n.d.). https://www.deeplearning.ai/ai-notes/initialization/.