Machine Learning II
Week 8 Lecture – 4th December 2019
Introduction to Artificial Neural Networks

Contents

Note the diagrams in this lecture are hand drawn on the blackboard and will not be uploaded to Moodle

Artificial Neural Networks (NNs) can be used for both regression problems and classification problems. We start with Regression because it is easier.

Data Structure and notation

The data Matrix $\boldsymbol{X}$ has n rows (the elements or observations) and p columns (variables).

We conceptualise the NN as processing each observation (row) one after the other. For an arbitrary observation we will drop the row index and represent it as a p-dimensional vector $\boldsymbol{x}$.

The predicted value for the same arbitrary observation $\boldsymbol{x}$ is $f(\boldsymbol{x})$.

The outcome variable is $\boldsymbol{y}$ a vector of n observations. The observed outcome value for $\boldsymbol{x}$ is y.

The *cost* or *loss* for this observation $\boldsymbol{x}$ is $R = (y - f(\boldsymbol{x}))^2$

When we consider all of the observations, the index $i = 1, \ldots, n$ will be used.

First Neural Network

Each NN can be visualised using a network graph. A very simple NN for introductory purposes is: Diagram 1

The NN has 3 **Layers** an *input layer* which receives the data for observation $\boldsymbol{x}$, a *hidden layer* and an output layer whose output value is the predicted value $f(\boldsymbol{x})$, which hopefully is a value close to y..

There are two nodes in the input layer ($p = 2$, i.e. two variables in the input data), just one node in the hidden layer and one node in the output layer.

- The hidden layer node receives a weighted sum of the data and a constant is added (called a bias).

$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}.$$

- This value is then transformed by a function $\sigma(\cdot)$.
- The hidden layer node outputs a value, which is called the *activation* of the node

$$a_1^{(1)} = \sigma(z_1^{(1)}) = \sigma(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)})$$

- The output layer in this example receives activation from the hidden layer multiplies it by a weight $w_{11}^{(2)}$ and adds another bias.
  The result is then transformed using another function $g_1(\cdot)$:

$$a_1^{(2)} = g_1(w_{11}^{(2)} a_1^{(1)} + b_1^{(2)}).$$

- The fitted/predicted value is the activation at the output layer.

$$f(\boldsymbol{x}) = a_1^{(2)}$$

- The superscript $^{(\cdot)}$ indicates which layer we are dealing with. The zero-th layer is the input layer. The last layer L is the output layer and all layers in between are hidden layers.

The NN can now be graphically represented with coefficients.
Diagram 2

Note the order of the $w_{st}^{(\cdot)}$ weights, which corresponds to "from node t to node s".

Each hidden and output layer node is represented as one circle, but carries out two transformations, which could be represented by two circles.
Diagram 3

The most common activation function $\sigma(\cdot)$ is the sigmoid or inverse logistic function

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

In early neural networks, the activation function was the socalled *heaviside* function

$$\sigma(v) = \begin{cases} 1 & \text{for } v > 0 \\ 0 & \text{for } v \leqslant 0. \end{cases}$$

The node is “activated” if the combination of the inputs exceed the threshold zero. This is a simple model for how neurons in the brain function, hence the name artificial neural network.

For regression the identity function is often used as the output function

$$g_1(v) = v.$$

A simple numerical example

Let $x_1 = 4, x_2 = 7$ and $y = 4.1$.

The weights for the first node are $w_{11}^{(1)} = 0.1, w_{12}^{(1)} = 0.2$ and $b_1^{(1)} = 1.2$
The weights for the second node (output layer) are $w_{11}^{(2)} = 0.5$ and $b_1^{(2)} = 3.5$

The activation function is the sigmoid function, and their output function is the identity function.

The weighted inputs are $z_1^{(1)} = 0.1 \times 4 + 0.2 \times 7 + 1.2 = 3$.
The activation of the hidden layer node is $a_1^{(1)} = \sigma(3) = \dfrac{1}{1 + e^{-3}} = 0.9526$

The output is then $a_1^{(2)} = g_1(0.5 \times 0.9526 + 3.5) = 3.976$

A simple numerical example: Diagram 4

It is important to note that all the weights and biases are the parameters of the NN model.

These values are applied to every observations in the data set.

If the second observation were $x_1 = 3, x_2 = -1$ $y = 4.1$.
The above model for the second observation gives:
$z_1^{(1)} = 1.3$.
The activation of the hidden layer is $a_1^{(1)} = 0.7858$ and
the activation of the output layer is $a_1^{(2)} = 3.892$, which is the fitted value.

Fitting

As with all machine learning methods we need to fit the model to the data. This is done in the usual way by minimising the loss function, which for NN-regression is the sum of squared errors SSE over the training set.

$$SSE = \sum_{i \in Tr} R_i = \sum_{i \in Tr} (y_i - f(\boldsymbol{x_i}))^2$$

Where Tr is the training data set.

The minimisation is done over all parameters $w_{11}^{(1)}$, $w_{12}^{(1)}$, $b_1^{(1)}$, $w_{11}^{(2)}$ and $b_1^{(2)}$.

Note that the number of nodes in each layer and the activation/output function are chosen by the user and are not part of the minimisation.

Training, validation and test data

Today we are only considering the most simple neural networks. In practice the the NN model can be extended in many ways e.g more nodes in the first hidden layer, more hidden layers, different choice of activation and output functions. The complexity of these models means that it is very easy to fit the dataset very well with the result that the prediction for future data is poor.

It is particularly important that the data are split into test and validation data sets. Very often we want an assessment accuracy of the model, in which case a test data set is needed as well.

The minimisation of the parameters $w_{11}^{(1)}$, $w_{12}^{(1)}$, $b_1^{(1)}$, $w_{11}^{(2)}$ and $b_1^{(2)}$ for a fixed model design is done using the training data.

If you want to make a choice about hyperparameters, such as how many nodes to fit in the first hidden layer, then this is done by fitting each model on the training data, and choosing the hyperparameters that minimise the MSE of the **validation data**.

Reporting benchmark statistics (such as the accuracy rate of a classifier) or comparing different machine learning methods must be done using “clean data” i.e. the **test data**.

Note that if the data are only split into two data sets, then data on which the model is validated are often called the test data!

Scaling

Although not totally necessary, fitting a neural network is much faster if the data are scaled (also called normalised) first.

Each variable is scaled by subtracting that variable's *training data* mean and dividing by that variable's *training data* standard deviation.

For variable x_j

$$x'_{ij} = \frac{x_{ij} - \overline{x}_j}{s_j} \qquad \text{with } \overline{x}_j = \frac{1}{|Tr|}\sum_{i \in Tr} x_{ij} \qquad \text{and } s_j = \frac{1}{|Tr| - 1}\sum_{i \in Tr}(x_{ij} - \overline{x}_j)^2$$

The output variable (for regression) also has to be scaled using the same approach.

When scaling or unscaling the validation/test data, you still use the training data mean and sd, because the coefficients were fitted using these parameters.

We will usually assume the data x_{ij} has already been scaled and omit the '.

In the previous very simple example we already have 5 parameters to vary in order to minimise the SSE. This is a five dimensional **function minimisation** problem. We can use any function minimisation algorithm to search for the best values for the 5 parameters.

In the workshop today we will use a very simple minimisation routine so that you can learn the mathematical principles of the NN Model.

This minimisation routine converges very slowly.

Next week we will take a closer look at better minimisation algorithms.
The usual algorithm for NNs is the *back propagation* algorithm.

It is rare to have just one node in the hidden layer. Even with small data sets it is common to use $M = 3$ to 20 nodes. Example and notation for one hidden layer with $M = 5$ units and $p = 6$. Diagram 5

Each edge has a weight, and there is a bias for each Node in the first and second layer. The general number of parameters (for a model with one hidden Layer) is $(p + 1)M + (M + 1) = (p + 2)M + 1$.
This NN has 41 edges. Each edge has a weight which needs to be optimised!

Multiple hidden layers

Instead of adding more nodes to one hidden layer, we can add more hidden layers. Example: 3 hidden layers $L = 4$, $M_1 = 3$ $M_2 = 2$,$M_3 = 2$ and $p = 4$. Diagram 6

The output layer

So far we have used one output node with:

$$a_1^{(L)} = g_1\left(\sum_{j=1}^{M_{L-1}} w_{1j}^{(L)} a_j^{(L-1)} + b_1^{(L)}\right).$$

The outcome variable can be multidimensional in which case there are as many output nodes as there are dimensions in y.

Example: if the outcome variable consists of the two standard blood pressure measurements, systolic and diastolic, then a NN predicting these outcomes requires two output nodes

$$a_1^{(L)} = g_1\left(\sum_{j=1}^{M_{L-1}} w_{1j}^{(L)} a_j^{(L-1)} + b_1^{(L)}\right) \text{ and } a_2^{(L)} = g_2\left(\sum_{j=1}^{M_{L-1}} w_{2j}^{(L)} a_j^{(L-1)} + b_2^{(L)}\right).$$

The predictor function is an $\mathbb{R}^p \to \mathbb{R}^2$ function $f(\boldsymbol{x}) = (a_1^{(L)}, a_2^{(L)})$.

Neural network classifiers

Classification NNs have the same configuration for the input and hidden layers, but the output layertakes account of the fact that the output variable takes one of K values.

Examples:

- Binary classifier ($K=2$), $y \in \{\text{Yes}, \text{No}\}$ usually internally coded as $\{0, 1\}$
- $K=3$ classifier: Iris dataset.
- $K=10$ classifier: handwritten digits 0 to 9.
- $K=62$ classifier: handwritten upper case letters, lower case letters and digits 0 to 9.

In K-class classification there are K output nodes one for each class.

In NN-regression we used the identity function at the output layer.
For NN-classifiers the usual output function is the sigmoid function, so

$$g_k(v) = \sigma(v)$$

The activation of each output node is

$$a_k^{(L)} = \sigma\left(\sum_{j=1}^{M_{L-1}} w_{kj}^{(L)} a_j^{(L-1)} + b_1^{(L)}\right)$$

The activation is a number between 0 and 1.

The K activations are numbers between 0 and 1 but how do we obtain our prediction?

The answer depends on what we actually want.

The *predicted value* will be 1 for the output node with the largest activation.

$$f(\boldsymbol{x}) = \arg\max_k \{a_k^{(L)}\}$$

If instead we want to predict the probabilities for each class $\pi_k(\boldsymbol{x})$, we need to standardise the activations, because they do not add up to one.

$$\pi_k(\boldsymbol{x}) = \frac{a_k^{(L)}}{\sum_{k=1}^{K} a_k^{(L)}}$$

Loss function for classification

To fit the NN classifier we minimise a loss function, comparing the K output values $a_k^{(L)}$ with the one observed value y.

To do this we redefine y as a K-dimensional binary vector.

Eg.
In the Iris dataset the 60th observation has species *versicolor* from the classes *setosa*, *versicolor* and *virginica*.

For a NN we redefine y_{60} as the 3-dim. binary vector $y_{60} = (0, 1, 0)$.

As with NN-regression, we can use the squared error loss for the loss in the i-th observation: $R_i = \sum_{k=1}^{K} (\pi_k(\boldsymbol{x}_i) - y_{ik})^2$

For classification the so called cross-entropy or deviance is used as the loss function more often than the squared error loss:

$$R_i = -\sum_{k=1}^{K} y_{ik} \log(\pi_k(\boldsymbol{x_i})).$$

By using the continuous values of $\pi_k(\cdot)$ rather than the binary predicted values, the loss function is continuous, making it much easier to minimise using numerical methods.

Workshop

We will use R-Code to create and optimise neural networks, both a regression NN and a classifier with $K = 3$. The aim is to understand the mathematical aspects of the NN and the concept of minimising the loss function

A source file and simulated data sets are provided. The neural network is fitted using a user defined R-function called `NN`.

You will need to fill in some of the gaps in the code.

The minimising routine is explained in the worksheet. It is easy to code but the convergence is very slow. Nevertheless, the fit can be surprisingly good even when using a small number of nodes.