A Brief Introduction to Neural Networks (I)

Jui-Chung (Ray) Yang

National Tsing Hua University

May 2020

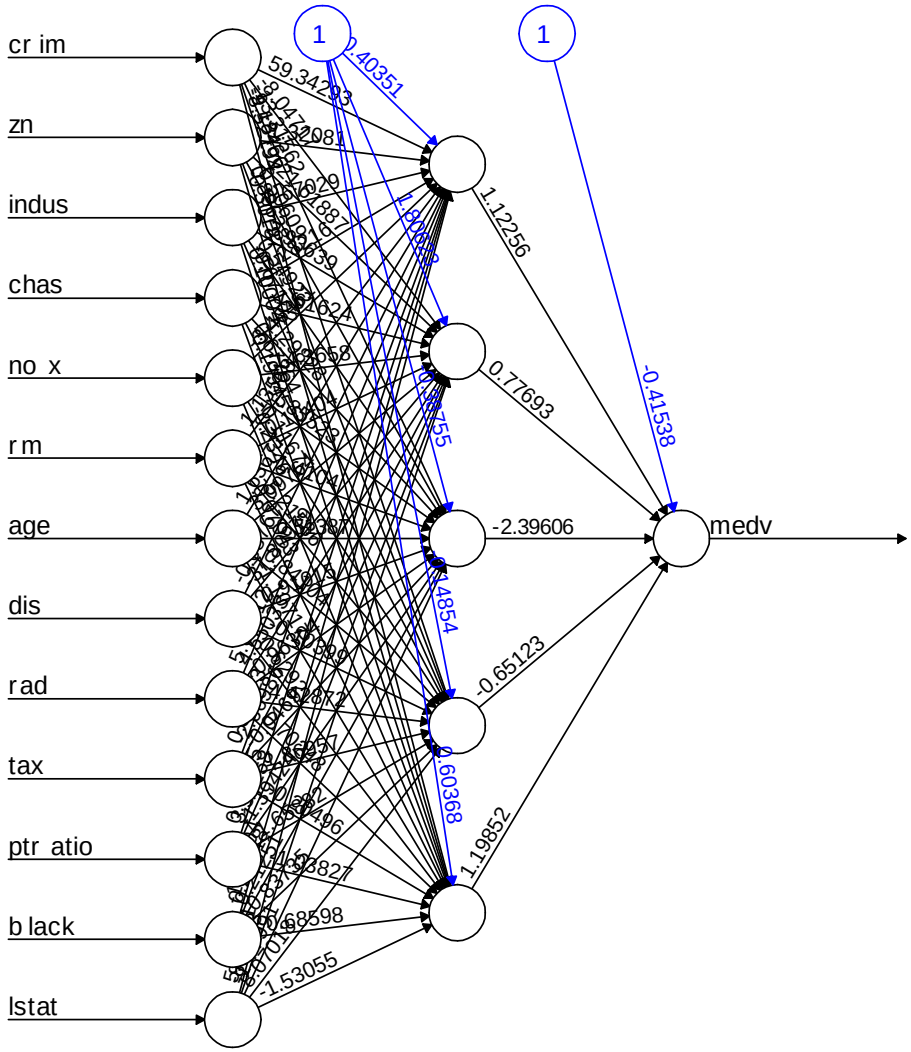
Reference

- Efron, B. and T. Hastie (2016), *Computer Age Statistical Inference: Algorithms, Evidence and Data Science*, Cambridge. `https://web.stanford.edu/~hastie/CASI/index.html`
- Kuan, C.-M. (2008) "Artificial neural networks," in New Palgrave Dictionary of Economics, S. N. Durlauf and L. E. Blume (eds.), Palgrave Macmillan. `https://homepage.ntu.edu.tw/~ckuan/pdf/Neural-Rev.pdf`
- Hastie, T., R. Tibshirani, and J. Friedman (2009), *The Elements of Statistical Learning*, 2nd edition, Springer-Verlag. `https://web.stanford.edu/~hastie/ElemStatLearn/`

Neural Networks / Deep Learning

- Introduced in the mid 1980s, *neural networks* marked a shift of predictive modeling towards computer science and machine learning.
  - Inspired by the architecture of the human brain, an NN is regarded as a *universal approximator* — a machine that *with enough data* could learn any smooth predictive relationship.
  - Somewhat sidelined in the mid 1990s due to the lack of computing power and ideal learning tasks...
- In the recent decade we've witnessed the re-emergence of NNs.
  - The reincarnation now being called *deep learning*.
  - Thanks to massive improvements in computer resources, some innovations, and niches such as image and video classification, speech and text processing, and *AlphaGo*.

A *Feed-Forward* NN Diagram



- 13 predictors or inputs: x_j.
- 1 hidden layer with 5 hidden units, or *neurons*.
- 1 single output unit o.

Feed-Forward NN

- In the previous slide is a 3-layer *feed-forward* neural network with $p = 13$ and $p_2 = 5$:

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j, \quad a_\ell^{(2)} = g^{(2)}\left(z_\ell^{(2)}\right), \quad \ell = 1, 2, \ldots, p_2,$$

$$z^{(3)} = w_0^{(2)} + \sum_{\ell=1}^{p_2} w_\ell^{(2)} a_\ell^{(2)}, \quad o = z^{(3)}.$$

  - $\left\{w_{\ell j}^{(1)}\right\}$ and $\left\{w_\ell^{(2)}\right\}$ are the *weights*, $\left\{w_{\ell 0}^{(1)}\right\}$ and $w_0^{(2)}$ are the *bias parameters*, and $g^{(2)}(\cdot)$ is the *activation function*.

In layer L_2 13 weights and 1 bias for each of the 5 neurons $\Rightarrow (13+1) \times 5 = 70$ parameters.

In layer L_3 5 weights and 1 bias for the output $\Rightarrow 5 + 1 = 6$ parameters.

- As seen before, an NN, even with only 3 layers, can be highly parametrized.
- More generally, we can consider a K-layer NN:

$$z_\ell^{(k)} = w_{\ell 0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{\ell j}^{(k-1)} a_j^{(k-1)}, \quad a_\ell^{(k)} = g^{(k)}\left(z_\ell^{(k)}\right),$$

for $\ell = 1, 2, \ldots, p_k$, where $a_j^{(1)} = x_j$ and $p_1 = p$.

- Or, in the matrix-vector notation,

$$\mathbf{z}^{(k)} = \mathbf{W}^{(k-1)} \mathbf{a}^{(k-1)}, \quad \mathbf{a}^{(k)} = g^{(k)}\left(\mathbf{z}^{(k)}\right).$$

  - $\mathbf{W}^{(k-1)}$ is the $p_k \times (p_{k-1} + 1)$ matrix of weights that go from layer L_{k-1} to L_k.
    - Note that the bias parameters $\left\{w_{\ell 0}^{(k-1)}\right\}$ are absorbed into $\mathbf{W}^{(k-1)}$.
  - $\mathbf{z}^{(k)}$ is the $p_k \times 1$ vector of linear transformation of $\mathbf{a}^{(k-1)}$.
  - $\mathbf{a}^{(k)}$ is the $(p_k + 1) \times 1$ vector of activations at layer L_k.

Activation Function

- $g^{(k)}(\cdot)$ is known as the *activation function*.
  - As the neurons in the human brain, the idea was that each neuron in the network would be a simple binary on/off.
  - In practice, people usually consider smooth and differentiable compromises:

Sigmoid or the *logistic* function delivers values in $(0,1)$.

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

Tangent hyperbolicus or the *hyperbolic tangent* function delivers values in $(-1,1)$.

$$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}},$$

- Nowadays, some other functions with cheaper gradient computations are often used:

  ReLU the *rectified linear unit*, the *rectifier*, or the *positive-part* function, delivers values in $[0, +\infty)$.

  $$g(z) = z_+ = \max(0, z).$$

  Leaky ReLU avoids flat spots and accompanying zero gradients. The nonnegative α is usually close to zero, *e.g.*, $\alpha = 0.01$.

  $$g_\alpha(z) = z_+ - \alpha z_- = \max(0, z) - \alpha \max(0, -z).$$

- $\left\{g^{(k)}\right\}$ at the inner layers can be the same or different, depending on how complicated the problem is.
- The final transformation is usually simply the identity function:

$$g^{(K)}(z) = z.$$

Classification

- For M-class classification, the number of output units is usually M, and the final activation function is usually the softmax function:

$$g_m^{(K)}(z_m, \mathbf{z}) = \frac{e^{z_m}}{\sum_{\ell=1}^{M} e^{z_\ell}},$$

  which computes a number (probability) between zero and one, and all M of them sum to one.

Universal Approximator

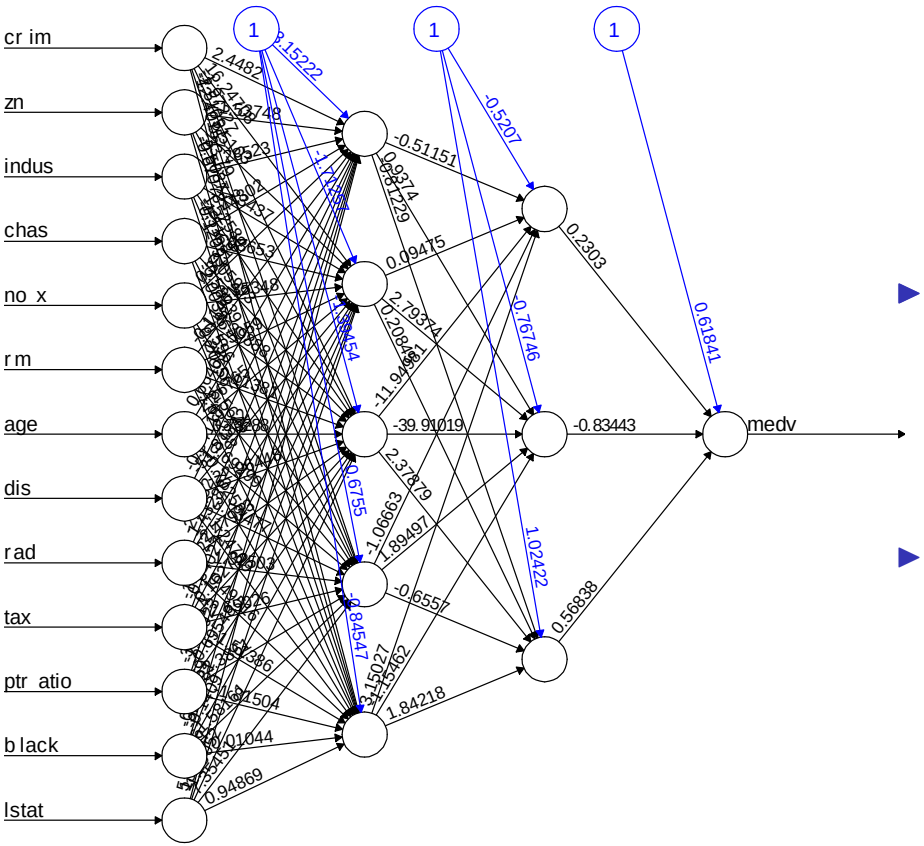
- What makes the NN a useful econometric tool is its *universal approximation* property.
  - A multi-layered NN with a large number of hidden units can well approximate a large class of functions.
- *E.g.*, an NN with one hidden layer and one single output unit (using the identity function) can be written as:

$$o = w_0^{(2)} + \sum_{\ell=1}^{p_2} w_\ell^{(2)} g^{(2)} \left(w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j \right).$$

  - When $\left\{ w_{\ell j}^{(1)} \right\}$ are fixed, the network reduces to a basis expansion.
  - An important enhancement is that ($\left\{ w_{\ell j}^{(1)} \right\}$), the parameters of basis functions, are learned from the data.
  - For a formal treatment, see Hornik (1991), Barron (1993), or Kuan (2008).

Fitting a Neural Network

- As we have seen, a neural network is a complex, hierarchical function $f(\mathbf{x}; \mathcal{W})$ of the the feature vector $\mathbf{x}$, and the collection of weights $\mathcal{W}$.



- For the feed-forward NN of K layers with p_1, p_2, ..., p_K neurons, the number of parameters is $\sum_{k=1}^{K-1} p_{k+1}(p_k + 1)$.
- In this example, the number of parameters is $5 \times 14 + 3 \times 6 + 1 \times 4 = 92$.

Fitting a Neural Network

- In principle, for the training set $(\mathbf{x}_i, y_i)_{i=1}^n$ and the loss function $L(y, f(\mathbf{x}))$, one might seek to solve

$$\min_{\mathcal{W}} \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(\mathbf{x}_i; \mathcal{W})).$$

  - *E.g.*, $L(y, f(\mathbf{x})) = \frac{1}{2}(y - f(\mathbf{x}))^2$, and the model might be estimated by a nonlinear least squares (NLS).
- In practice, loss functions are usually convex only in f, but not in elements of $\mathcal{W}$.
- The minimization problem is difficult, and at best we seek good local optima.
- Nowadays, the estimation is usually via the *backpropagation*.

Computing the Gradient: Backpropagation

- The *backpropagation* computes the derivative of $L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)$ w.r.t. any of the elements of $\mathcal{W}$ for a generic pair $(\mathbf{x}, y)$, using the *chain rule for differentiation*.
  - Since the objective is an average, the overall gradient will be the average of these individual gradient elements over the training pairs $(\mathbf{x}_i, y_i)_{i=1}^n$.
- Given a training generic pair $(\mathbf{x}, y)$, we first make a forward pass through the network, which creates activations at each of the nodes $a_\ell^{(k)}$ in each of the layers.
- We would, then, like to compute an error term $\delta_\ell^{(k)}$ that measures the responsibility of each node for the error in predicting the true output y.
  - For the output activations $a^{(K)}$ these errors are easy.
  - For activations at inner layers, $\delta_\ell^{(k)}$ will be a weighted sum of the errors terms of nodes that use $a_\ell^{(k)}$ as inputs.

Algorithm (Backpropagation)

1. *Given a pair* $(\mathbf{x}, y)$*, perform a "feedforward pass," computing the activations* $a_\ell^{(k)}$ *at each of the layers* $L_2, L_3, \ldots, L_K$*; i.e. compute* $f(\mathbf{x}; \mathcal{W})$ *at* $\mathbf{x}$ *using the current* $\mathcal{W}$*, saving each of the intermediary quantities along the way.*
2. *For each output unit* ℓ *in layer* L_K*, compute*

$$\delta_\ell^{(K)} = \frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial z_\ell^{(K)}} = \frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial a_\ell^{(K)}} \dot{g}^{(K)}\left(z_\ell^{(K)}\right),$$

where $\dot{g}^{(K)}$ *denotes the derivative of* $g^{(K)}(z)$ *wrt* z*.*

   - *E.g.,* $L\left(y, f\left(\mathbf{x}\right)\right) = \frac{1}{2}\left(y - f\left(\mathbf{x}\right)\right)^2$*,* $\delta_\ell^{(K)}$ *becomes* $-\left(y - a^{(K)}\right) \cdot \dot{g}^{(K)}\left(z_\ell^{(K)}\right)$*.*

Algorithm (Backpropagation, cont'd)

3 *For layers* $k = K-1, K-2, \ldots, 2$, *and for each node* ℓ *in layer* L_k, *set*

$$\delta_\ell^{(k)} = \frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial z_\ell^{(k)}} = \left(\sum_{j=1}^{p_{k+1}} w_{j\ell}^{(k)} \delta_j^{(k+1)}\right) \dot{g}^{(k)}\left(z_\ell^{(k)}\right).$$

4. *The partial derivatives are given by*

$$\frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial w_{\ell j}^{(k)}} = \frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial z_\ell^{(k+1)}} \frac{\partial z_\ell^{(k+1)}}{\partial w_{\ell j}^{(k)}} = a_j^{(k)} \delta_\ell^{(k+1)}.$$

- Again, the matrix-vector notation simplifies these expressions. Recall that

$$\mathbf{z}^{(k)} = \mathbf{W}^{(k-1)}\mathbf{a}^{(k-1)}, \quad \mathbf{a}^{(k)} = g^{(k)}\left(\mathbf{z}^{(k)}\right).$$

- By the chain rule,

$$\begin{aligned}
\delta^{(K)} &= \nabla_{\mathbf{z}^{(K)}} L\left(y, f\left(\mathbf{x}\right)\right) = \nabla_{a^{(K)}} L\left(y, f\left(\mathbf{x}\right)\right) \cdot \dot{g}^{(K)}\left(\mathbf{z}^{(K)}\right), \\
\delta^{(k)} &= \mathbf{W}^{(k)\prime}\delta^{(k+1)} \circ \dot{g}^{(k)}\left(\mathbf{z}^{(k)}\right), \\
\frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial \mathbf{W}^{(k)}} &= \delta^{(k+1)}\mathbf{a}^{(k)\prime},
\end{aligned}$$

where $\circ$ denotes the Hadamard (elementwise) product.

Gradient Descent

- The backpropagation algorithm computes the gradient of the loss function at a single generic pair $(\mathbf{x}, y)$.
- With n training pairs, the gradient w.r.t. $w_{\ell j}^{(k)}$ is given by

$$\Delta \mathbf{W}^{(k)} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L\left(y, f\left(\mathbf{x}; \mathcal{W}\right)\right)}{\partial \mathbf{W}^{(k)}},$$

- Given a set of starting values for all the weights, a *gradient-descent* update is

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \alpha \Delta \mathbf{W}^{(k)}, \quad k = 1, 2, \ldots, K-1,$$

  where $\alpha > 0$ is the *learning rate*.
  - Intuitively, $\Delta w_{\ell j}^{(k)} > 0$ suggests that the starting value of $w_{\ell j}^{(k)}$ is too big, and vice versa.

Stochastic Gradient Descent

- Rather than process all observations before making a gradient step, it can be more efficient to process smaller *batches* at a time.
  - Even batches of size one!
  - These batches can be sampled at random, or systematically processed.
  - For large data sets distributed on multiple computer cores, this can be essential for reasons of efficiency.
- An *epoch* of training means that all n training samples have been used in gradient steps, irrespective of how they have been grouped.

Example: Boston Housing

- `Boston` data frame in `MASS` library in `R` has housing values of 506 nbhds around Boston (Harrison and Rubinfeld, 1978).

`crim`	per capita crime rate by town.
`zn`	prop. of residential land zoned for lots over 25,000 sq.ft.
`indus`	prop. of non-retail business acres per town.
`chas`	Charles River dummy variable ($= 1$ if tract bounds river).
`nox`	nitrogen oxides concentration (parts per 10 million).
`rm`	average number of rooms per dwelling.
`age`	proportion of owner-occupied units built prior to 1940.
`dis`	wt. mean of distances to five Boston employment centres.
`rad`	index of accessibility to radial highways.
`tax`	full-value property-tax rate per \$10,000.
`ptratio`	pupil-teacher ratio by town.
`black`	$1000\,(Bk - 0.63)^2$, Bk: the prop. of blacks by town.
`lstat`	lower status of the population (percent).
`medv`	median value of owner-occupied homes in \$1000s.

neuralnet

- We use the `neuralnet` package in `R`.
  - Other popular choices includes `Python` packages such as `keras` or `PyTorch`.

```
library(neuralnet)
library(MASS)
data <- Boston
set.seed(500)
index <- sample(1:nrow(data),
round(0.75*nrow(data)))
train <- data[index,]
test <- data[-index,]
```

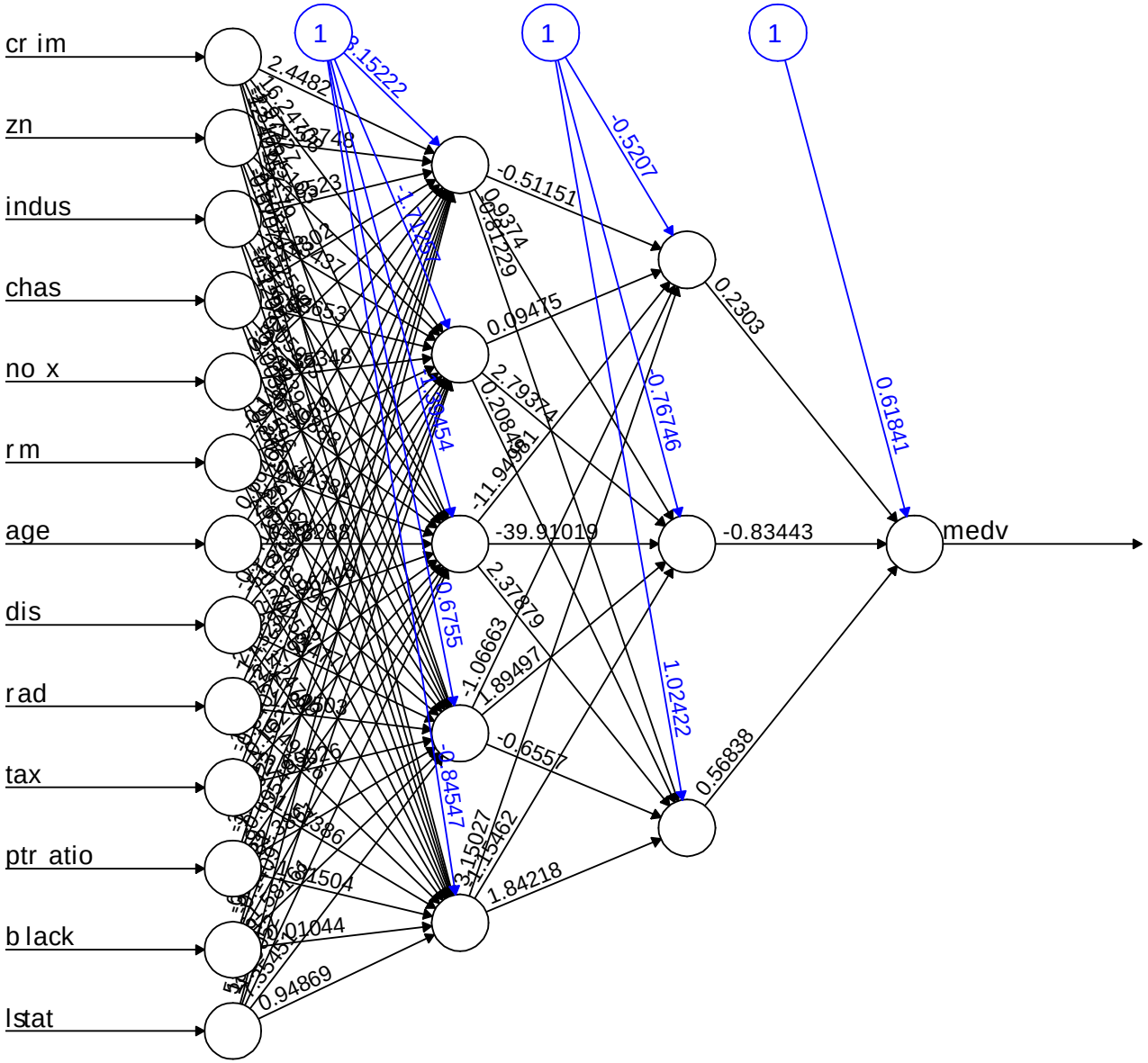
Scaling

- The scaling of the inputs can have a large effect on the quality of the final solution.
- Usually all inputs are standardized to have mean zero and standard deviation one.
  - This also allows one to choose a meaningful range for the random starting weights.
  - Usually $\mathcal{U}[-0.7, 0.7]$.

```
maxs <- apply(data, 2, max)
mins <- apply(data, 2, min)
scaled <- as.data.frame(scale(data, center = mins,
scale = maxs - mins))
train_ <- scaled[index,]
test_ <- scaled[-index,]
```

- `neuralnet` trains neural networks.
  - `hidden` is a vector of integers specifying the number of hidden neurons in each layer.

```
set.seed(500)
n <- names(train_)
f <- as.formula(paste("medv ~", paste(n[!n %in%
"medv"], collapse = " + ")))
nn <- neuralnet(f, data = train_, hidden = c(5,3),
linear.output = T)
windows()
plot(nn)
```


cr im
zn
indus
chas
no x
r m
age
dis
r ad
tax
ptr atio
b lack
lstat
1
1
1
-0.51151
0.09475
-39.91019
2.37879
-1.06663
1.89497
-0.6557
1.84218
0.2303
-0.76746
-0.83443
1.02422
0.56838
0.61841
medv