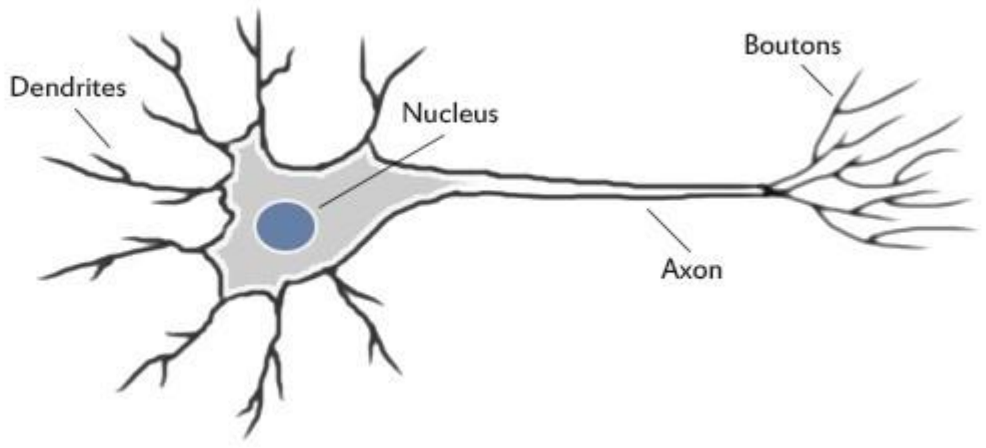
The Introduction to Neural Networks

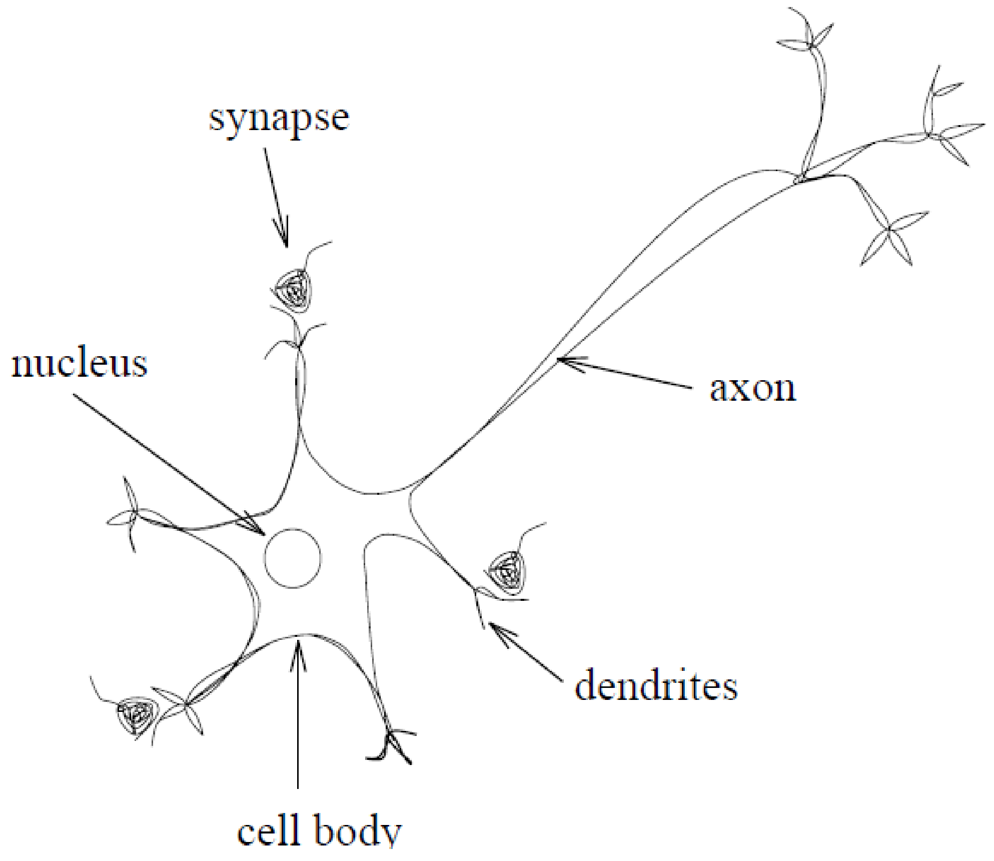
MOTIVATION

- Our brain uses the extremely large interconnected network of neurons for information processing and to model the world around us. Simply put, a neuron collects inputs from other neurons using *dendrites.* The neuron sums all the inputs and if the resulting value is greater than a threshold, it fires. The fired signal is then sent to other connected neurons through the axon.



Source: https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb

Biological Networks



1. The majority of ***neurons*** encode their outputs or activations as a series of brief electrical pulses (i.e. spikes or action potentials).

2. ***Dendrites*** are the receptive zones that receive activation from other neurons.

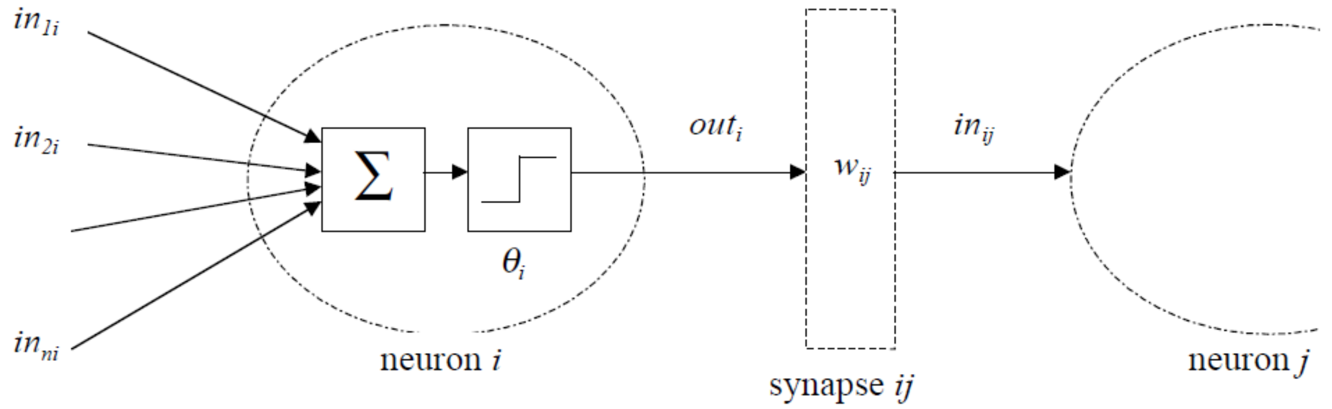
3. The ***cell body (soma)*** of the neuron's processes the incoming activations and converts them into output activations.

4. ***Axons*** are transmission lines that send activation to other neurons.

5. ***Synapses*** allow weighted transmission of signals (using ***neurotransmitters***) between axons and dendrites to build up large neural networks.

Networks of McCulloch-Pitts Neurons

- Artificial neurons have the same basic components as biological neurons. The simplest ANNs consist of a set of ***McCulloch-Pitts neurons*** labelled by indices *k*, *i*, *j* and activation flows between them via synapses with strengths w_{ki}, w_{ij}:



$$in_{ki} = out_k w_{ki} \qquad out_i = \operatorname{sgn}\left(\sum_{k=1}^{n} in_{ki} - \theta_i\right) \qquad in_{ij} = out_i w_{ij}$$

MOTIVATION

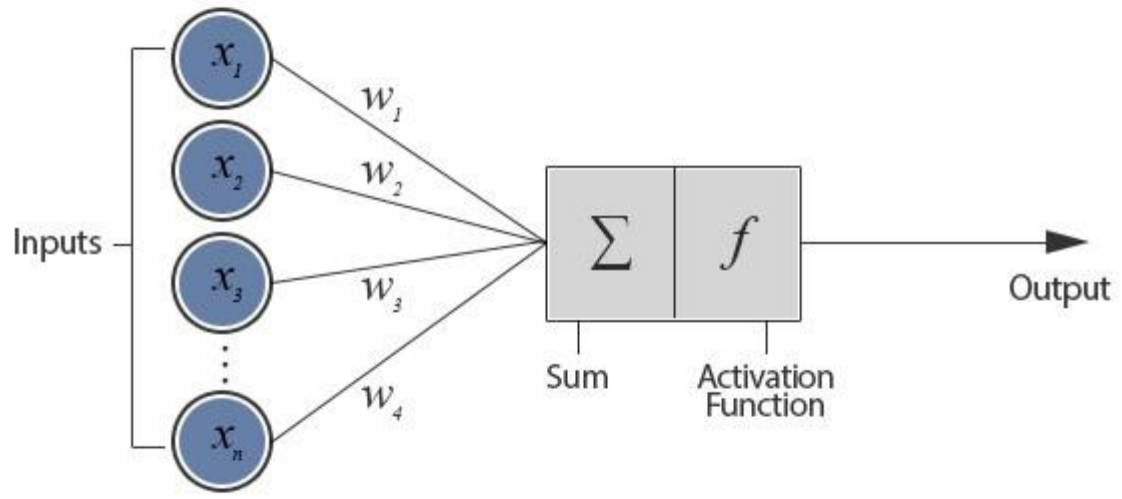
- Humans are incredible pattern-recognition machines. Our brains process ‘inputs’ from the world, categorize them (that’s a spider; that’s ice-cream), and then generate an ‘output’ (run away from the spider; taste the ice-cream). And we do this automatically and quickly, with little or no effort.
- It’s the very same system that *senses* that someone is angry at us, or involuntarily reads the stop sign as we speed past it. Psychologists call this mode of thinking ‘System 1’ (coined by Keith Stanovich and Richard West), and it includes the innate skills — like perception and fear — that we share with other animals.

MOTIVATION

- Neural networks loosely mimic the way our brains solve the problem: by taking in inputs, processing them and generating an output. Like us, they *learn* to recognize patterns, but they do this by *training* on labelled datasets. Before we get to the learning part, let's take a look at the most basic of artificial neurons: the **perceptron**, and how it processes inputs and produces an output.

THE PERCEPTRON

- Perceptrons were developed way back in the 1950s-60s by the scientist Frank Rosenblatt, inspired by earlier work from Warren McCulloch and Walter Pitts. While today we use other models of artificial neurons, they follow the general principles set by the perceptron.



- As you can see, the network of nodes sends signals in one direction. This is called **a feed-forward network**.
- The figure depicts a neuron connected with *n* other neurons and thus receives *n* inputs (x_1, x_2, ….. x_n). This configuration is called a **Perceptron.**

THE PERCEPTRON

- Let’s understand this better with an example. Say you bike to work. You have two factors to make your decision to go to work: the weather must not be bad, and it must be a weekday. The weather’s not that big a deal, but working on weekends is a big no-no. The inputs have to be binary, so let’s propose the conditions as yes or no questions. Weather is fine? 1 for yes, 0 for no. Is it a weekday? 1 yes, 0 no.
- Remember, I cannot tell the neural network these conditions; it has to learn them for itself. How will it know which information will be most important in making its decision? It does with something called **weights**. Remember when I said that weather’s not a big deal, but the weekend is? Weights are just a numerical representation of these preferences. A higher weight means the neural network considers that input more important compared to other inputs.

Source: https://towardsdatascience.com/a-beginners-guide-to-neural-networks-b6be0d442fa4

THE PERCEPTRON

- For our example, let's purposely set suitable weights of 2 for weather and 6 for weekday. Now how do we calculate the output? We simply multiply the input with its respective weight, and sum up all the values we get for all the inputs. For example, if it's a nice, sunny (1) weekday (1), we would do the following calculation:

$$total = (nice..weather?) \times (weight_1) + (weekday?) \times (weight_2)$$

$$total = (1 \times 2) + (1 \times 6) = 8$$

- This calculation is known as a **linear combination**. Now what does an 8 mean? We first need to define the **threshold value.** The neural network's output, 0 or 1 (stay home or go to work), is determined if the value of the linear combination is greater than the threshold value. Let's say the threshold value is 5, which means that if the calculation gives you a number less than 5, you can stay at home, but if it's equal to or more than 5, then you have to go to work.

THE PERCEPTRON

- You have just seen how weights are influential in determining the output. In this example, we set the weights to particular numbers that make the example work, but in reality, we set the weights to random values, and then the network adjusts those weights based on the output errors it made using the previous weights. This is called **training** the neural network.

TRAINING IN PERCEPTRONS

- Try teaching a child to recognize a bus?
- You show her examples, telling her, “This is a bus. That is not a bus,” until the child learns the concept of what a bus is. Furthermore, if the child sees new objects that she hasn’t seen before, we could expect her to recognize correctly whether the new object is a bus or not.
- *This is exactly the idea behind the perceptron.*

TRAINING IN PERCEPTRONS

- Input vectors from a training set are presented to the perceptron one after the other and weights are modified according to the following equation,
- For all inputs i,

W(i) = W(i) + a*g'(sum of all inputs)*(T-A)*P(i),

where g' is the derivative of the activation function, and a is the learning rate

- Here, W is the weight vector. P is the input vector. T is the correct output that the perceptron should have known and A is the output given by the perceptron.

TRAINING IN PERCEPTRONS

- When an entire pass through all of the input training vectors is completed without an error, the perceptron has learnt!
- At this time, if an input vector P (already in the training set) is given to the perceptron, it will output the correct value. If P is not in the training set, the network will respond with an output similar to other training vectors close to P.

WHAT IS THE PERCEPTRON ACTUALLY DOING?

- The perceptron is adding all the inputs and separating them into 2 categories, those that cause it to fire and those that don’t. That is, it is drawing the line:

$$w_1x1 + w_2x2 = t,$$

where t is the threshold.

$$output = \begin{cases} 0 \text{ if } \sum_i w_i x_i \ < \ threshold \\ 1 \text{ if } \sum_i w_i x_i \ \geqslant \ threshold \end{cases}$$

WHAT IS THE PERCEPTRON ACTUALLY DOING?

- To make things a little simpler for *training* later, let's make a small readjustment to the above formula. Let's move the *threshold* to the other side of the inequality, and replace it with what's known as the neuron's *bias.* Now we can rewrite the equation as:

$$output = \begin{cases} 0 \text{ if } \sum_i w_i x_i \ + \ bias < 0 \\ 1 \text{ if } \sum_i w_i x_i \ + \ bias \geqslant 0 \end{cases}$$

- Effectively, *bias = — threshold.* You can think of *bias* as how easy it is to get the neuron to output a 1 — with a really big bias, it's very easy for the neuron to output a 1, but if the bias is very negative, then it's difficult.

LIMITATION OF PERCEPTRONS

- Not every set of inputs can be divided by a line like this. Those that can be are called *linearly separable*. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly.

ACTIVATION FUNCTION

- A function that transforms the values or states the conditions for the decision of the output neuron is known as an **activation function**.
- What does an artificial neuron do? Simply, it calculates a “weighted sum” of its input, adds a bias and then decides whether it should be “fired” or not.
- So consider a neuron.

$$Y = \sum (weight * input) + bias$$

ACTIVATION FUNCTION

- The value of Y can be anything ranging from -inf to +inf. The neuron really doesn’t know the bounds of the value. So how do we decide whether the neuron should fire or not (why this firing pattern? Because we learnt it from biology that’s the way brain works and brain is a working testimony of an awesome and intelligent system).
- We decided to add “activation functions” for this purpose. To check the Y value produced by a neuron and decide whether outside connections should consider this neuron as “fired” or not. Or rather let’s say — “activated” or not.

ACTIVATION FUNCTION

- If we do not apply an Activation function, then the output signal would simply be a simple ***linear function****.* A *linear function* is just a polynomial of **one degree.**
- A linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data.
- A Neural Network without Activation function would simply be a **Linear Regression Model,** which has limited power and does not performs good most of the times.
- We want our Neural Network to not just learn and compute a linear function but something more complicated than that.
- Also, without activation function our Neural network would not be able to learn and model other complicated kinds of data such as images, videos , audio , speech etc. That is why we use Artificial Neural network techniques such as **Deep learning to make sense of something complicated ,high dimensional, non-linear -big datasets, where the model has lots and lots of hidden layers in between and has a very complicated architecture which helps us to make sense and extract knowledge form such complicated big datasets.**

ACTIVATION FUNCTION

- ***Activation** functions* are really important for a Artificial Neural Network to learn and make sense of something really complicated and non-linear complex functional mappings between the inputs and response variable. They introduce non-linear properties to our network.
- **Their main purpose is to convert an input signal of a node in a A-NN to an output signal.** That output signal now is used as a input in the next layer in the stack.

WHY DO WE NEED NON-LINEARITIES?

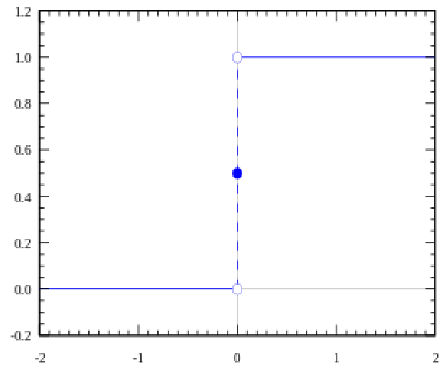
- Non-linear functions are those which have degree more than one and they have a curvature when we plot a Non-Linear function. Now we need a Neural Network Model to learn and represent almost anything and any arbitrary complex function which maps inputs to outputs. Neural-Networks are considered ***Universal Function Approximators***. It means that they can compute and learn any function at all. Almost any process we can think of can be represented as a functional computation in Neural Networks.

- Hence it all comes down to this, we need to apply an Activation function f(x) so as to make the network more powerful and add ability to it to learn something complex and complicated form data and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence using a non linear Activation, we are able to generate non-linear mappings from inputs to outputs.

TYPES OF ACTIVATION FUNCTIONS*

Step function

- Activation function A = “activated” if Y > threshold else not
- Alternatively, A = 1 if Y > threshold, 0 otherwise
- Well, what we just did is a “step function”, see the below figure.



- **DRAWBACK:** Suppose you are creating a binary classifier. Something which should say a “yes” or “no” (activate or not activate). A Step function could do that for you! That’s exactly what it does, say a 1 or 0. Now, think about the use case where you would want multiple such neurons to be connected to bring in more classes. Class1, class2, class3 etc. What will happen if more than 1 neuron is “activated”. All neurons will output a 1 (from step function). Now what would you decide? Which class is it? Hard, complicated.

* https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f

TYPES OF ACTIVATION FUNCTIONS

Linear function

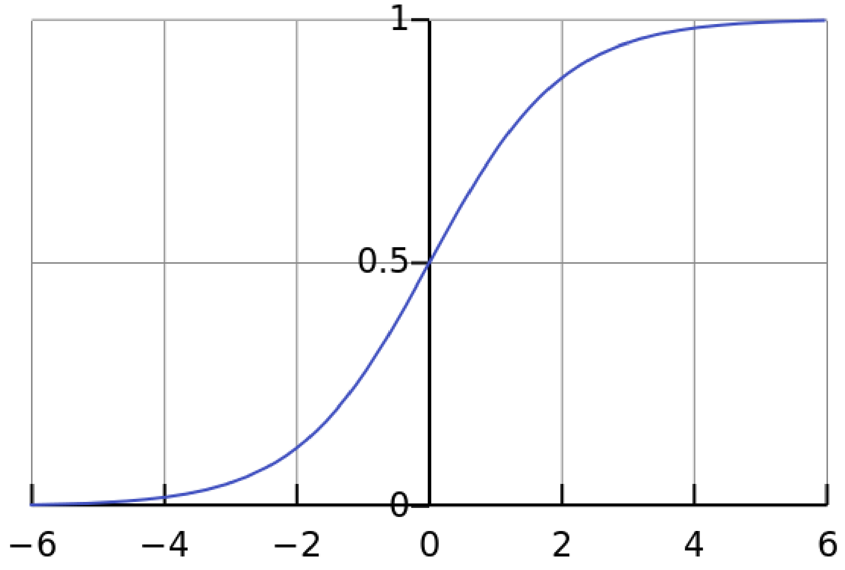
- A = cx
- A straight line function where activation is proportional to input (which is the weighted sum from neuron).
- This way, it gives a range of activations, so it is not binary activation. We can definitely connect a few neurons together and if more than 1 fires, we could take the max and decide based on that. So that is ok too. Then what is the problem with this?
- A = cx, derivative with respect to x is c. That means, the gradient has no relationship with X. It is a constant gradient and the descent is going to be on constant gradient. If there is an error in prediction, the changes made by back propagation is constant and not depending on the change in input.

TYPES OF ACTIVATION FUNCTIONS

Sigmoid function

$$A = \frac{1}{1+e^{-x}}$$

This looks smooth and “step function like”. What are the benefits of this? It is nonlinear in nature. Combinations of this function are also nonlinear! Great. Now we can stack layers. What about non binary activations? Yes, that too! It will give an analog activation unlike step function. It has a smooth gradient too.



And if you notice, between X values -2 to 2, Y values are very steep. Which means, any small changes in the values of X in that region will cause values of Y to change significantly. That means this function has a tendency to bring the Y values to either end of the curve.

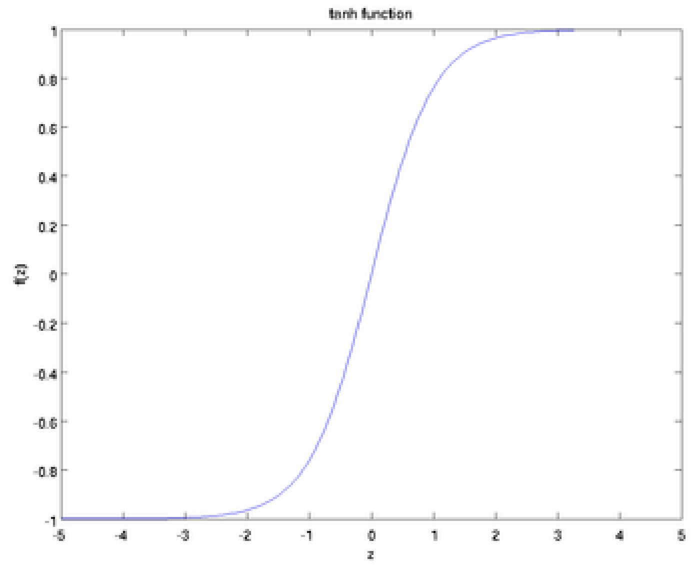
- Looks like it’s good for a classifier considering its property? Yes ! It tends to bring the activations to either side of the curve (above x = 2 and below x = -2 for example). Making clear distinctions on prediction.
- Another advantage of this activation function is, unlike linear function, the output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range. It won’t blow up the activations then. This is great.
- Sigmoid functions are one of the most widely used activation functions today. Then what are the problems with this?
- If you notice, towards either end of the sigmoid function, the Y values tend to respond very less to changes in X. What does that mean? The gradient at that region is going to be small. It gives rise to a problem of “vanishing gradients”. So what happens when the activations reach near the “near-horizontal” part of the curve on either sides?
- Gradient is small or has vanished (cannot make significant change because of the extremely small value). The network refuses to learn further or is drastically slow. There are ways to work around this problem and sigmoid is still very popular in classification problems.

TYPES OF ACTIVATION FUNCTIONS

Tanh Function

- Another activation function that is used is the tanh function.

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$



This looks very similar to sigmoid. In fact, it is a scaled sigmoid function!

$$tanh(x) = 2\, sigmoid(2x) - 1$$

- This has characteristics similar to sigmoid that we discussed above. It is nonlinear in nature, so great we can stack layers! It is bound to range (-1, 1) so no worries of activations blowing up. One point to mention is that the gradient is stronger for tanh than sigmoid (derivatives are steeper). Deciding between the sigmoid or tanh will depend on your requirement of gradient strength. Like sigmoid, tanh also has the vanishing gradient problem.
- Tanh is also a very popular and widely used activation function. Especially in time series data.

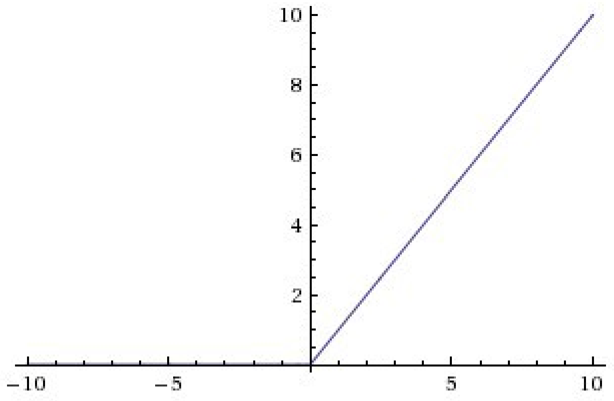
TYPES OF ACTIVATION FUNCTIONS

ReLu

- Later, comes the ReLu function,

$$A(x) = \max(0,x)$$

The ReLu function is as shown above. It gives an output x if x is positive and 0 otherwise.



- At first look, this would look like having the same problems of linear function, as it is linear in positive axis. First of all, ReLu is nonlinear in nature. And combinations of ReLu are also non linear! (in fact it is a good approximator. Any function can be approximated with combinations of ReLu). Great, so this means we can stack layers. It is not bound though. The range of ReLu is [0, inf). This means it can blow up the activation.
- Another point to discuss here is the sparsity of the activation. Imagine a big neural network with a lot of neurons. Using a sigmoid or tanh will cause almost all neurons to fire in an analog way. That means almost all activations will be processed to describe the output of a network. In other words, the activation is dense. This is costly. We would ideally want a few neurons in the network to not activate and thereby making the activations sparse and efficient.
- ReLu give us this benefit. Imagine a network with random initialized weights (or normalized) and almost 50% of the network yields 0 activation because of the characteristic of ReLu (output 0 for negative values of x). This means a fewer neurons are firing (sparse activation) and the network is lighter. ReLu seems to be awesome! Yes it is, but nothing is flawless.. Not even ReLu.

- Because of the horizontal line in ReLu (for negative X), the gradient can go towards 0. For activations in that region of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called **dying ReLu problem**. This problem can cause several neurons to just die and not respond making a substantial part of the network passive. There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component . For example, $y = 0.01x$ for $x<0$ will make it a slightly inclined line rather than horizontal line. This is leaky ReLu. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep neural nets.

NOW WHICH ONE DO WE USE?

- Does that mean we just use ReLu for everything we do? Or sigmoid or tanh? Well, yes and no.
- When you know the function you are trying to approximate has certain characteristics, you can choose an activation function which will approximate the function faster leading to faster training process. For example, a sigmoid works well for a classifier, because approximating a classifier function as combinations of sigmoid is easier than maybe ReLu, for example. Which will lead to faster training process and convergence. You can use your own custom functions too! If you don’t know the nature of the function you are trying to learn, then maybe you can start with ReLu, and then work backwards. ReLu works most of the time as a general approximator!

MULTI-LAYERED NEURAL NETWORKS

- Once a training sample is given as an input to the network, each output node of the single layered neural network (also called **Perceptron**) takes a weighted sum of all the inputs and pass them through an activation function and comes up with an output. The weights are then corrected using the following equation,

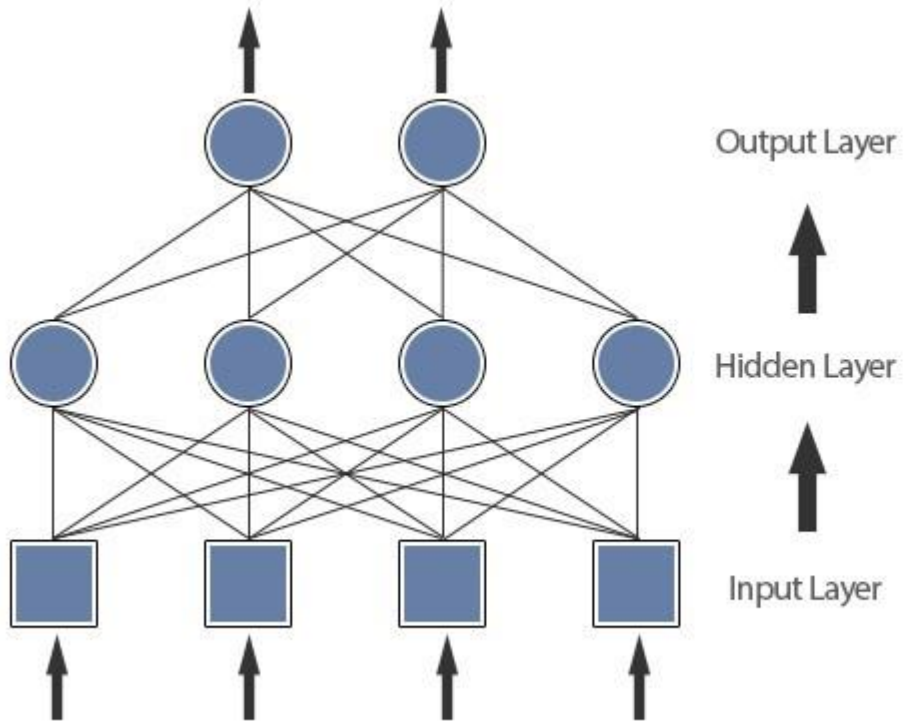
For all inputs i,

$$\mathbf{W(i) = W(i) + a*g'(\text{sum of all inputs})*(T-A)*P(i),}$$

where a is the learning rate and g' is the derivative of the activation function.

MULTI-LAYERED NEURAL NETWORKS

- This process is repeated by feeding the whole training set several times until the network responds with a correct output for all the samples. The training is possible only for inputs that are linearly separable. This is where multi-layered neural networks come into picture.



MULTI-LAYERED NEURAL NETWORKS

- Each input from the input layer is fed up to each node in the hidden layer, and from there to each node on the output layer. We should note that there can be any number of nodes per layer and there are usually multiple hidden layers to pass through before ultimately reaching the output layer.
- But to train this network we need a learning algorithm which should be able to tune **not only** the weights between the output layer and the hidden layer **but also** the weights between the hidden layer and the input layer.

BACK PROPAGATION (BACKWARD PROPAGATION OF ERRORS)

- Backpropagation is a common method for training a neural network.
- The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.
- To tune the weights between the hidden layer and the input layer, we need to know the error at the hidden layer, but we know the error only at the output layer (*We know the correct output from the training sample and we also know the output predicted by the network.).*
- So, the method that was suggested was to take the errors at the output layer and proportionally propagate them backwards to the hidden layer.

BACK PROPAGATION

- For a particular neuron in output layer

for all j { **$W_{j,i}$ = $W_{j,i}$ + a*g'(sum of all inputs)*(T-A)*P(j)** *}*

- This equation tunes the weights between the output layer and the hidden layer.
- For a particular neuron *j* in hidden layer, we propagate the error backwards from the output layer, thus

Error = $W_{j,1}$ * E_1 + $W_{j,2}$ * E_2 + …..

for all the neurons in output layer.

- Thus, *for a particular neuron in hidden layer*

for all k { **$W_{k,j}$ = $W_{k,j}$ + a*g'(sum of all inputs)*(T-A)*P(k)** *}*

- This equation tunes the weights between the hidden layer and the input layer.

Learning by Gradient Descent Error Minimization

- The Perceptron learning rule is an algorithm that adjusts the network weights w_{ij} to minimize the difference between the actual outputs out_j and the target outputs $targ_j^p$. We can quantify this difference by defining the ***Sum Squared Error*** function, summed over all output units *j* and all training patterns *p*:

$$E(w_{mn}) = \frac{1}{2}\sum_{p}\sum_{j}\left(targ_j^p - out_j\left(in_i^p\right)\right)^2$$

Learning by Gradient Descent Error Minimization

- It is the general aim of network ***learning*** to minimize this error by adjusting the weights w_{mn}. Typically we make a series of small adjustments to the weights $w_{mn} \rightarrow w_{mn} + \Delta w_{mn}$ until the error $E(w_{mn})$ is 'small enough'. We can determine which direction to change the weights in by looking at the gradients (i.e. partial derivatives) of E with respect to each weight w_{mn}. Then the ***gradient descent update equation*** (with positive learning rate η) is

$$\Delta w_{kl} = -\eta \frac{\partial E(w_{mn})}{\partial w_{kl}}$$

which can be applied iteratively to minimize the error.

Practical Considerations for Gradient Descent Learning

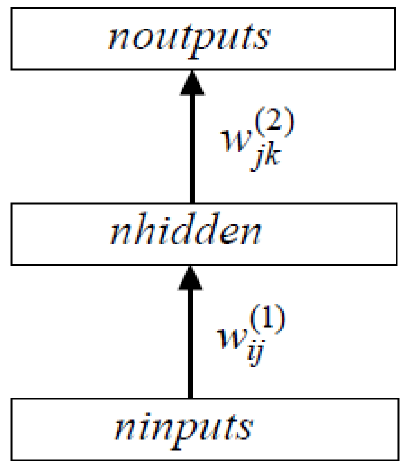
There a number of important practical/implementational considerations that must be taken into account when training neural networks:

1. Do we need to pre-process the training data? If so, how?
2. How many hidden units do we need?
3. Are some activation functions better than others?
4. How do we choose the initial weights from which we start the training?
5. Should we have different learning rates for the different layers?
6. How do we choose the learning rates?
7. Do we change the weights after each training pattern, or after the whole set?
8. How do we avoid flat spots in the error function?
9. How do we avoid local minima in the error function?
10. When do we stop training?

In general, the answers to these questions are highly problem dependent.

Multi-Layer Perceptrons (MLPs)

- To deal with non-linearly separable problems we can use non-monotonic activation functions. More conveniently, we can instead extend the simple Perceptron to a ***Multi-Layer Perceptron,*** which includes at least one hidden layer of neurons with ***non-linear*** activations functions *f(x)* (such as sigmoids):



$$out_k^{(2)} = f\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right)$$

$$out_j^{(1)} = f\left(\sum_i out_i^{(0)} w_{ij}^{(1)}\right)$$

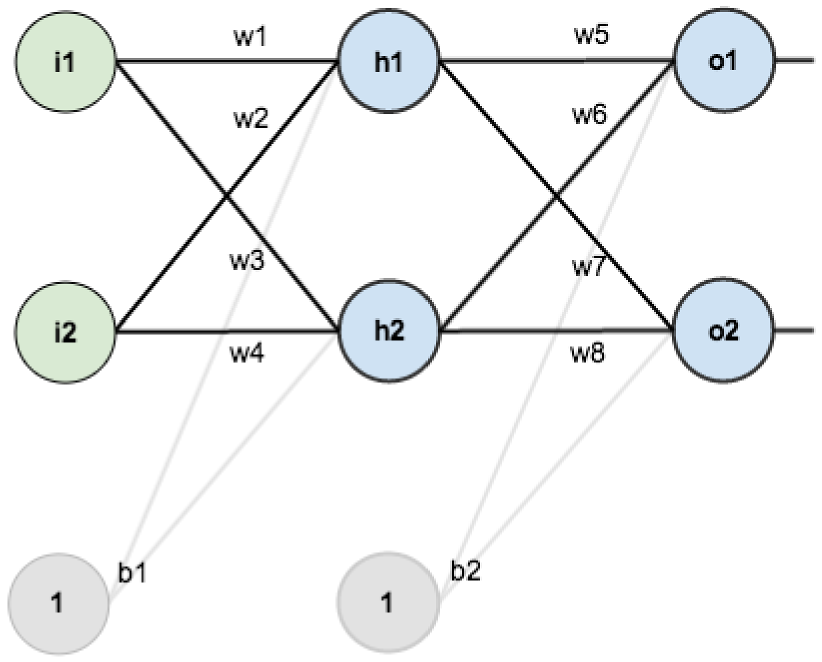
$$out_i^{(0)} = in_i$$

- Note that if the activation on the hidden layer were linear, the network would be equivalent to a single layer network, and wouldn't be able to cope with non-linearly separable problems.

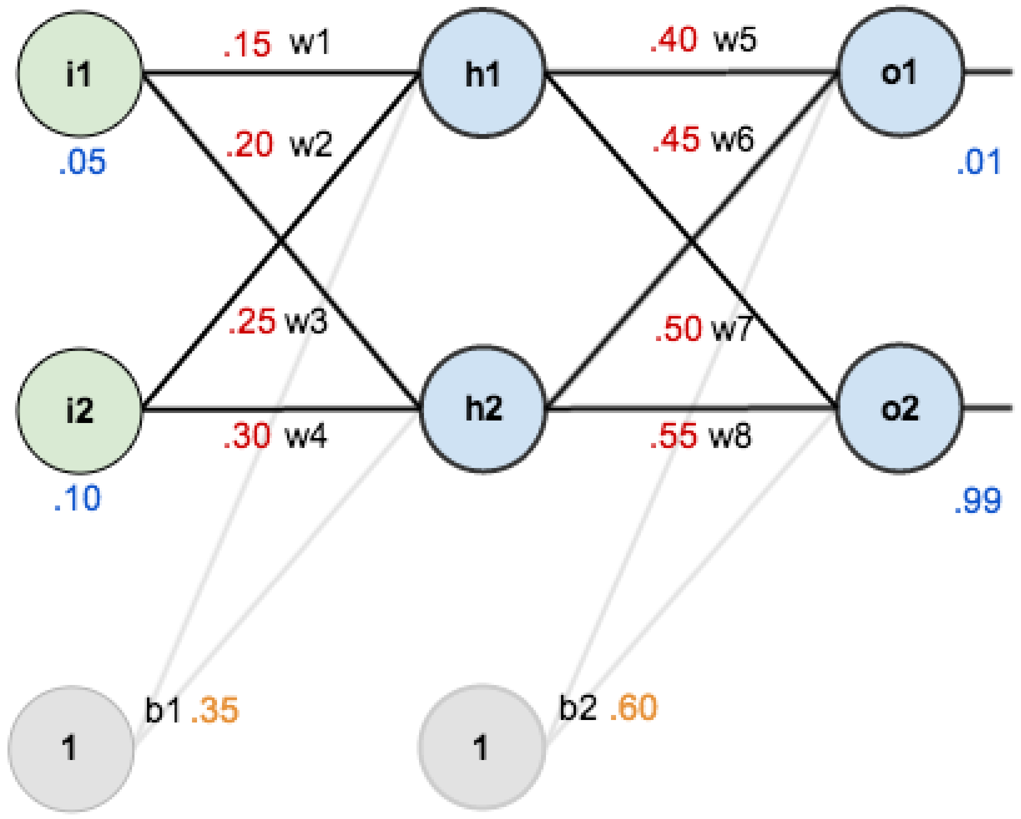
BACK PROPOGATION BY EXAMPLE

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

- Consider a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.



In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.



The Forward Pass

- To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs forward though the network.
- We figure out the *total net input* to each hidden layer neuron, *squash* the total net input using an *activation function* (here we use the *sigmoid function*), then repeat the process with the output layer neurons.
- Here's how we calculate the total net input for h_1:

$$net_{h_1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h_1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1

$$out_{h_1} = \frac{1}{1 + e^{-net_{h_1}}} = \frac{1}{1 + e^{-0.3775}} = 0.5933$$

- Carrying out the same process for h$_2$ we get $out_{h_2} = 0.5969$.
- We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.
- Here’s the output for o$_1$:

$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$
$$net_{o_1} = 0.4 * 0.5933 + 0.45 * 0.5969 + 0.6 * 1 = 1.1059$$
$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}} = \frac{1}{1 + e^{1.1059}} = 0.7514.$$

and carrying out the same process for o$_2$ we get out_{o_2} =0.7729.

Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \frac{1}{2}\sum_{i=1}^{n}(target - output)^2$$

For example, the target output for o_1 is 0.01 but the neural network output 0.7514, therefore its error is:

$$E_{total_{o_1}} = \frac{1}{2}\sum_{i=1}^{n}(\mathrm{target}_{o_1} - \mathrm{output}_{o_1})^2 = \frac{1}{2}(0.01 - 0.7514)^2 = 0.2748$$

Repeating this process for o_2 (remembering that the target is 0.99) we get: $E_{total_{o_2}}$=0.0236.

The total error for the neural network is the sum of these errors:

$$E_{total}=E_{total_{o_1}} + E_{total_{o_2}}=0.2748+0.0236=0.2984.$$

The Backwards Pass

- Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer to the target output, thereby minimizing the error for each output neuron and the network as a whole.

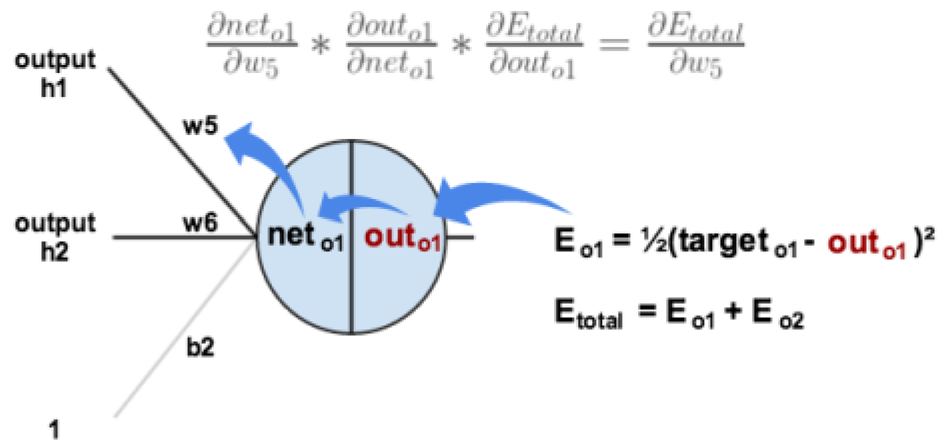
Output Layer

Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$ (the partial derivative of E_{total} with respect to w_5.

You can also say “the gradient with respect to w_5“.)

- By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$



$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$
output h1
w5
output h2
w6
b2
1
net o1
out o1
E o1 = ½(target o1 - out o1)²
E total = E o1 + E o2

- We need to figure out each piece in this equation.
- First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(\text{target}_{o_1} - \text{output}_{o_1})^2 + \frac{1}{2}(\text{target}_{o_2} - \text{output}_{o_2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o_1}} = 2 * \frac{1}{2}\left(\text{target}_{o_1} - \text{output}_{o_1}\right)^{2-1} * (-1) + 0$$

$$\frac{\partial E_{total}}{\partial out_{o_1}} = -\left(\text{target}_{o_1} - \text{output}_{o_1}\right) = -(0.01 - 0.7514) = 0.7414$$

- Next, how much does the output of o_1 change with respect to its total net input?
- The partial derivative of the logistic function is the output multiplied by 1 minus the output:

$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}}$$

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = out_{o_1}(1 - out_{o_1}) = 0.7514(1 - 0.7514) = 0.1868$$

Finally, how much does the total net input of o_1 change with respect to w_5?

$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$

$$\frac{\partial net_{o_1}}{\partial w_5} = 1 * out_{h_1} * w_5^{1-1} + 0 + 0 = out_{h_1} = 0.5933$$

- Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.7414*0.1868*0.5933=0.0822$$

You'll often see this calculation combined in the form of the delta rule:

$$\frac{\partial E_{total}}{\partial w_5} = \underbrace{-(target_{o_1} - out_{o_1}) * out_{o_1} * (1 - out_{o_1})}_{\boxed{\text{delta}}} * out_{h_1}$$

- To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.0822 = 0.3589$$

- We can repeat this process to get the new weights w_6, w_7, and w_8

$$w_6^+ = 0.408666186$$
$$w_7^+ = 0.511301270$$
$$w_8^+ = 0.561370121$$

- We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons.

Hidden Layer

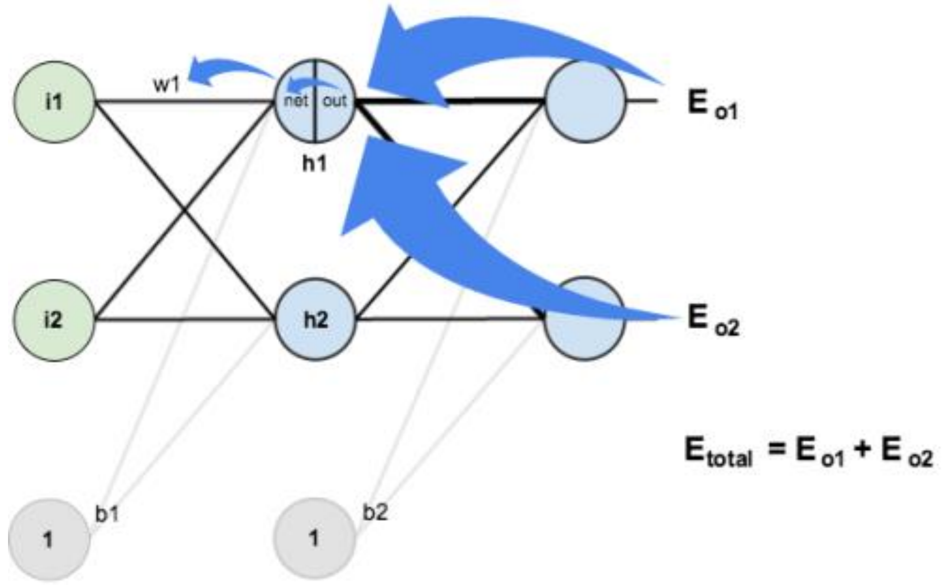
- Next, we'll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.
- Big picture, here's what we need to figure out:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



- We’re going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h_1} affects both out_{o_1} and out_{o_2} therefore the $\frac{\partial E_{total}}{\partial out_{h_1}}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial out_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial net_{o_1}} = \frac{\partial E_{o_1}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{h_1}} = 0.7414 * 0.1868 = 0.1385$$

$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$

$$\frac{\partial net_{o_1}}{\partial out_{h_1}} = w_5 = 0.40$$

Plugging them in: $\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}} = 0.1385 * 0.40 = 0.0554$

- Following the same process for $\frac{\partial E_{o_2}}{\partial out_{h_1}}$: $\frac{\partial E_{o_2}}{\partial out_{h_1}} = -0.0190$.
- Hence, $\frac{\partial E_{total}}{\partial out_{h_1}}$=$\frac{\partial E_{o_1}}{\partial out_{h_1}}$+$\frac{\partial E_{o_2}}{\partial out_{h_1}} = 0.0554 - 0.0190 = 0.0364.$
- Now that we have $\frac{\partial E_{total}}{\partial out_{h_1}}$, we need to figure out $\frac{\partial out_{h_1}}{\partial net_{h_1}}$ and then $\frac{\partial net_{h_1}}{\partial w_1}$.

$$out_{h_1} = \frac{1}{1+e^{-net_{h_1}}}$$

$$\frac{\partial out_{h_1}}{\partial net_{h_1}} = out_{h_1}\left(1 - out_{h_1}\right) = 0.5933(1 - 0.5933) = 0.2413$$

- We calculate the partial derivative of the total net input to h1 with respect to w_1 the same as we did for the output neuron:

$$net_{h_1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h_1}}{\partial w_1} = i_1$$

- Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1} = 0.0364 * 0.2413 * 0.05 = 0.00044$$

We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.00044 = 0.1498$$

$$w_2^+ = 0.1996$$
$$w_3^+ = 0.2498$$
$$w_4^+ = 0.2995$$

Finally, we’ve updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.2984. After this first round of backpropagation, the total error is now down to 0.2910. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

- For another example to understand the concept numerically please visit https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c

R EXAMPLE (https://datascienceplus.com/fitting-neural-network-in-r/)

- We are going to use the Boston dataset in the MASS package.
- The Boston dataset is a collection of data about housing values in the suburbs of Boston. Our goal is to predict the median value of owner-occupied homes (medv) using all the other continuous variables available.

```
set.seed(500)
library(MASS)
data <- Boston
head(Boston)
     crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2
6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21 28.7
```

- First we need to check that no data point is missing, otherwise we need to fix the dataset.

```
apply(data,2,function(x) sum(is.na(x)))
    crim       zn    indus     chas      nox       rm      age      dis      rad      tax  ptratio    black    lstat     medv 
       0        0        0        0        0        0        0        0        0        0        0        0        0        0 
```

- There is no missing data, good. We proceed by randomly splitting the data into a train and a test set, then we fit a linear regression model and test it on the test set.

```
index <- sample(1:nrow(data),round(0.75*nrow(data)))
train <- data[index,]
test <- data[-index,]
```

```
lm.fit <- glm(medv~., data=train)
summary(lm.fit)
Deviance Residuals:
     Min        1Q    Median        3Q       Max
-14.9143   -2.8607   -0.5244    1.5242   25.0004
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  43.469681   6.099347   7.127 5.50e-12 ***
crim         -0.105439   0.057095  -1.847 0.065596 .
zn            0.044347   0.015974   2.776 0.005782 **
indus         0.024034   0.071107   0.338 0.735556
chas          2.596028   1.089369   2.383 0.017679 *
nox         -22.336623   4.572254  -4.885 1.55e-06 ***
rm            3.538957   0.472374   7.492 5.15e-13 ***
age           0.016976   0.015088   1.125 0.261291
dis          -1.570970   0.235280  -6.677 9.07e-11 ***
rad           0.400502   0.085475   4.686 3.94e-06 ***
tax          -0.015165   0.004599  -3.297 0.001072 **
ptratio      -1.147046   0.155702  -7.367 1.17e-12 ***
black         0.010338   0.003077   3.360 0.000862 ***
lstat        -0.524957   0.056899  -9.226  < 2e-16 ***
(Dispersion parameter for gaussian family taken to be 23.26491)

    Null deviance: 33642  on 379  degrees of freedom
Residual deviance:  8515  on 366  degrees of freedom
AIC: 2290
```

```
pr.lm <- predict(lm.fit,test)
MSE.lm <- sum((pr.lm - test$medv)^2)/nrow(test)
MSE.lm
[1] 21.62976
```

Preparing to fit the neural network

- Before fitting a neural network, some preparation need to be done. Neural networks are not that easy to train and tune.
- As a ***first step***, we are going to address data preprocessing.
  It is good practice to **normalize your data** before training a neural network. Depending on your dataset, avoiding normalization may lead to useless results or to a very difficult training process (most of the times the algorithm will not converge before the number of maximum iterations allowed). You can choose different methods to scale the data (z-normalization, min-max scale, etc…). Here, we use the min-max method and scale the data in the interval [0,1]. Usually scaling in the intervals [0,1] or [-1,1] tends to give better results.

- We therefore scale and split the data before moving on:

```
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,]
```

- Note that scale returns a matrix that needs to be coerced into a data.frame.

Parameters

There is no fixed rule as to how many layers and neurons to use although there are several more or less accepted rules of thumb. Usually, if at all necessary, **one hidden layer** is enough for a vast numbers of applications.

As far as the number of neurons is concerned, it should be between the input layer size and the output layer size, usually 2/3 of the input size.

Since this is a toy example, we are going to use 2 hidden layers with this configuration: 13:5:3:1. The input layer has 13 inputs, the two hidden layers have 5 and 3 neurons and the output layer has a single output since we are doing regression.

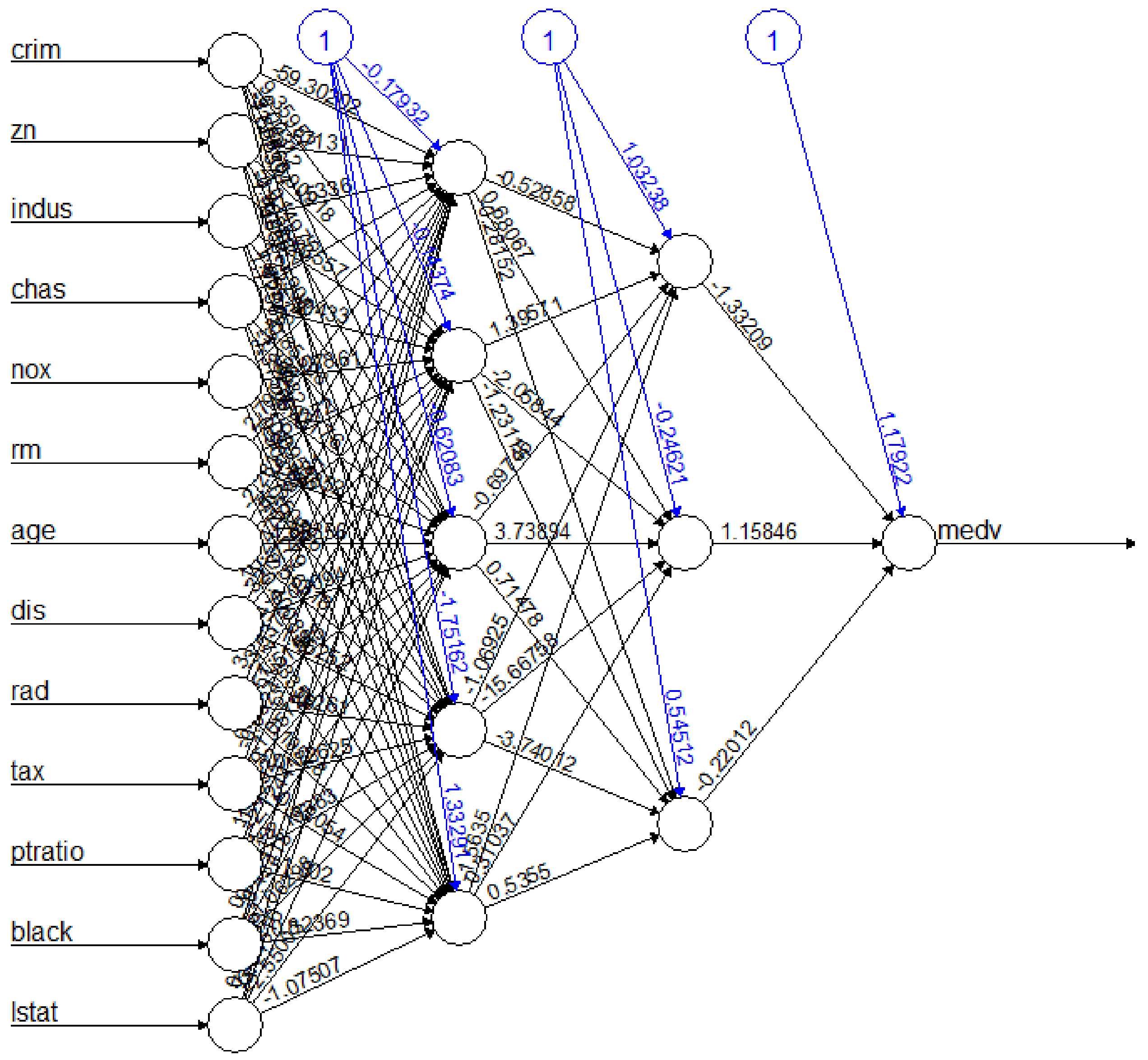
- Let’s fit the net:

```
library(neuralnet)
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)
```

- Formula y~ format does not work with neuralnet.
- Write the formula and then pass it as an argument in the fitting function.

- The hidden argument accepts a vector with the number of neurons for each hidden layer, while the argument linear.output is used to specify whether we want to do regression linear.output=TRUE or classification linear.output=FALSE

plot(nn)



The black lines show the connections between each layer and the weights on each connection while the blue lines show the bias term added in each step. The bias can be thought as the intercept of a linear model.

The net is essentially a black box so we cannot say that much about the fitting, the weights and the model. Suffice to say that the training algorithm has converged and therefore the model is ready to be used.

Predicting medv using the neural network

- Now we can try to predict the values for the test set and calculate the MSE. Remember that the net will output a normalized prediction, so we need to scale it back in order to make a meaningful comparison (or just a simple prediction).

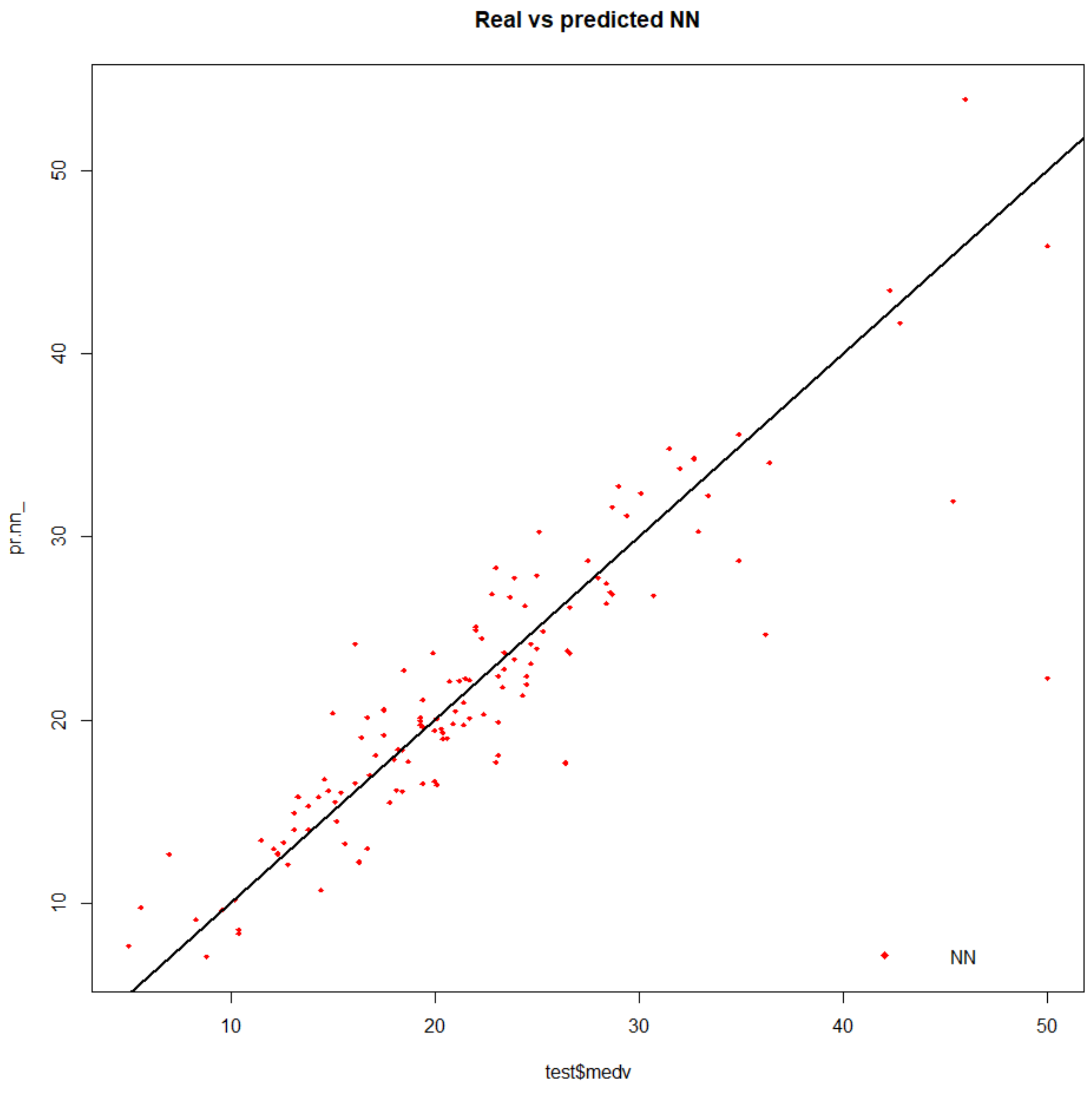
```
pr.nn <- compute(nn,test_[,1:13])

pr.nn_ <- pr.nn$net.result*(max(data$medv)-
min(data$medv))+min(data$medv)

test.r <- (test_$medv)*(max(data$medv)-min(data$medv))+min(data$medv)

MSE.nn <- sum((test.r - pr.nn_)^2)/nrow(test_)
```

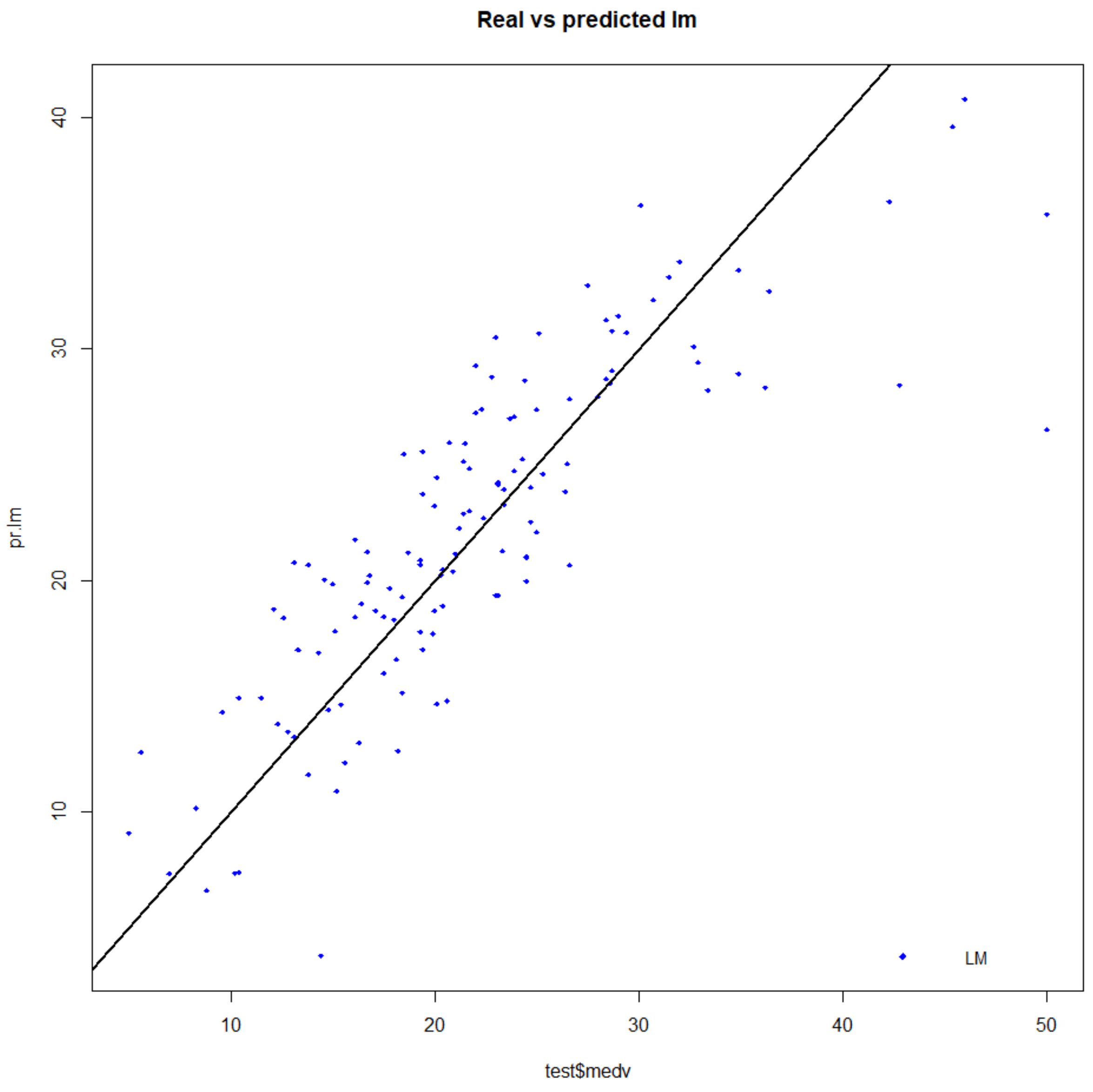
- We then compare the two MSEs

```
print(paste(MSE.lm,MSE.nn))

[1] "21.6297593507225 15.7518370200153"
```

- Apparently, the net is doing a better work than the linear model at predicting medv.
- Once again, be careful because this result depends on the train-test split performed above. Below, after the visual plot, we are going to perform a fast cross validation in order to be more confident about the results.
- A first visual approach to the performance of the network and the linear model on the test set is plotted below

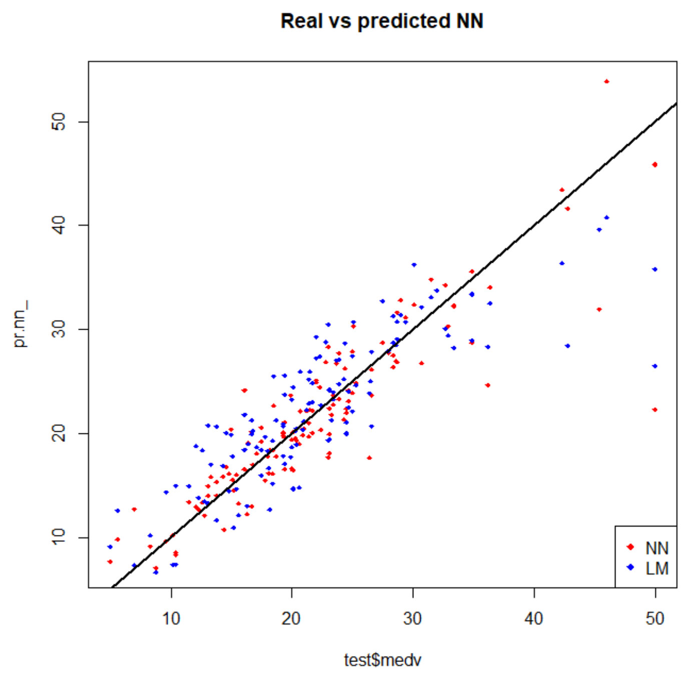
```
par(mfrow=c(1,2))
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='NN',pch=18,col='red', bty='n')
plot(test$medv,pr.lm,col='blue',main='Real vs predicted lm',pch=18, cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='LM',pch=18,col='blue', bty='n', cex=.95)
```


Real vs predicted NN
pr.nn_
test$medv
NN
10
20
30
40
50


Real vs predicted lm
pr.lm
test$medv
LM
10
20
30
40
50

- By visually inspecting the plot we can see that the predictions made by the neural network are (in general) more concentrated around the line (a perfect alignment with the line would indicate a MSE of 0 and thus an ideal perfect prediction) than those made by the linear model.

```
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
points(test$medv,pr.lm,col='blue',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend=c('NN','LM'),pch=18,col=c('red','blue'))
```



A (fast) cross validation

We are going to implement a fast cross validation using a for loop for the neural network and the cv.glm() function in the boot package for the linear model.

As far as I know, there is no built-in function in R to perform cross-validation on this kind of neural network, if you do know such a function, please let me know in the comments. Here is the 10 fold cross-validated MSE for the linear model:

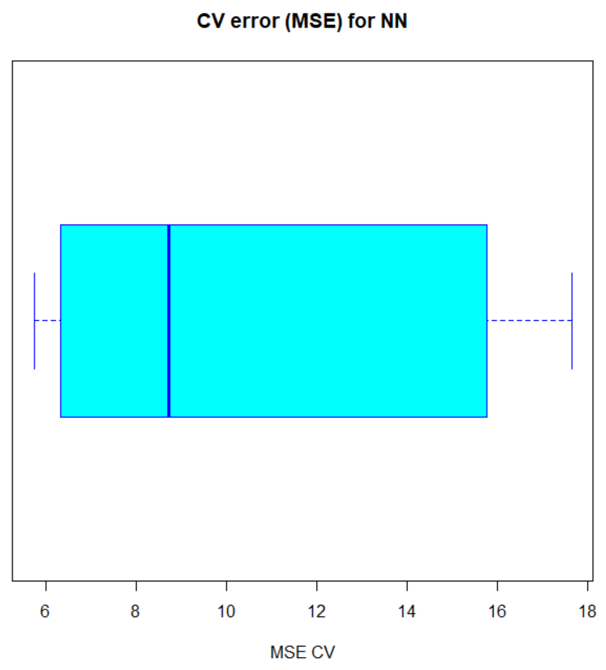
```
library(boot)

set.seed(200)

lm.fit <- glm(medv~.,data=data)

cv.glm(data,lm.fit,K=10)$delta[1]

[1] 23.83560156
```

- Now the net. We are splitting the data in this way: 90% train set and 10% test set in a random way for 10 times.
- Initialize a progress bar using the plyr library because we want to keep an eye on the status of the process since the fitting of the neural network may take a while.

```
set.seed(450)
cv.error <- NULL
k <- 10
library(plyr)
pbar <- create_progress_bar('text')
pbar$init(k)
for(i in 1:k){
    index <- sample(1:nrow(data),round(0.9*nrow(data)))
    train.cv <- scaled[index,]
    test.cv <- scaled[-index,]
    nn <- neuralnet(f,data=train.cv,hidden=c(5,2),linear.output=T)
    pr.nn <- compute(nn,test.cv[,1:13])
    pr.nn <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)
    test.cv.r <- (test.cv$medv)*(max(data$medv)-min(data$medv))+min(data$medv)
    cv.error[i] <- sum((test.cv.r - pr.nn)^2)/nrow(test.cv)
    pbar$step()
}
```

- The PLYR package is a tool for doing split-apply-combine (SAC) procedures.
- What it does do is take the process of SAC and make it cleaner, more tidy and easier.
- PLYR functions have a neat naming convention.
- `create_progress_bar` $\rightarrow$ Create progress bar object from text string.

```
mean(cv.error)
[1] 10.32697995
cv.error
17.640652805  6.310575067 15.769518577  5.730130820 10.520947119
6.121160840  6.389967211  8.004786424 17.369282494  9.412778105
boxplot(cv.error,xlab='MSE CV',col='cyan',
        border='blue',names='CV error (MSE)',
        main='CV error (MSE) for NN',horizontal=TRUE)
```



- As you can see, the average MSE for the neural network (10.33) is lower than the one of the linear model although there seems to be a certain degree of variation in the MSEs of the cross validation. This may depend on the splitting of the data or the random initialization of the weights in the net. By running the simulation different times with different seeds you can get a more precise point estimate for the average MSE.

A final note on model interpretability

Neural networks resemble black boxes a lot: explaining their outcome is much more difficult than explaining the outcome of simpler model such as a linear model. Therefore, depending on the kind of application you need, you might want to take into account this factor too. Furthermore, as you have seen above, extra care is needed to fit a neural network and small changes can lead to different results.

Solving classification problems with neuralnet

(https://datascienceplus.com/neuralnet-train-and-test-neural-networks-using-r/)

- Our goal is to develop a neural network to determine if a stock pays a dividend or not.
- In our dataset, we assign a value of **1** to a stock that pays a dividend. We assign a value of **0** to a stock that does not pay a dividend.
- Our independent variables are as follows:
  - **fcfps:** Free cash flow per share (in $)
  - **earnings_growth:** Earnings growth in the past year (in %)
  - **de:** Debt to Equity ratio
  - **mcap:** Market Capitalization of the stock
  - **current_ratio:** Current Ratio (or Current Assets/Current Liabilities)

```
setwd("your directory")
mydata <- read.csv("dividendinfo.csv")
attach(mydata)
```

First, scale your data. Failure to scale the data will typically result in the prediction value remaining the same across all observations, regardless of the input values.

```
scaleddata<-scale(mydata)
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
```

Then, we use lapply to run the function across our existing data

```
maxmindf <- as.data.frame(lapply(mydata, normalize))
head(maxmindf)
> head(maxmindf)
  dividend      fcfps earnings_growth         de      mcap current_ratio
1        0 0.54361055       0.0000000 0.26717557 0.6350575     0.3177037
2        1 0.99188641       0.3383319 0.26208651 0.7571839     0.5052078
3        1 0.54969574       0.3427127 0.03307888 0.6594828     0.6798973
4        0 0.07302231       0.5428812 0.41730280 0.4094828     0.6682316
5        1 0.58215010       0.3654591 0.45038168 0.8347701     0.8560637
6        1 0.77687627       0.2183656 0.10178117 0.7442529     0.6135678
```

We base our training data (trainset) on 80% of the observations. The test data (testset) is based on the remaining 20% of observations.

```
# Training and Test Data

trainset <- maxmindf[1:160, ]

testset <- maxmindf[161:200, ]
```

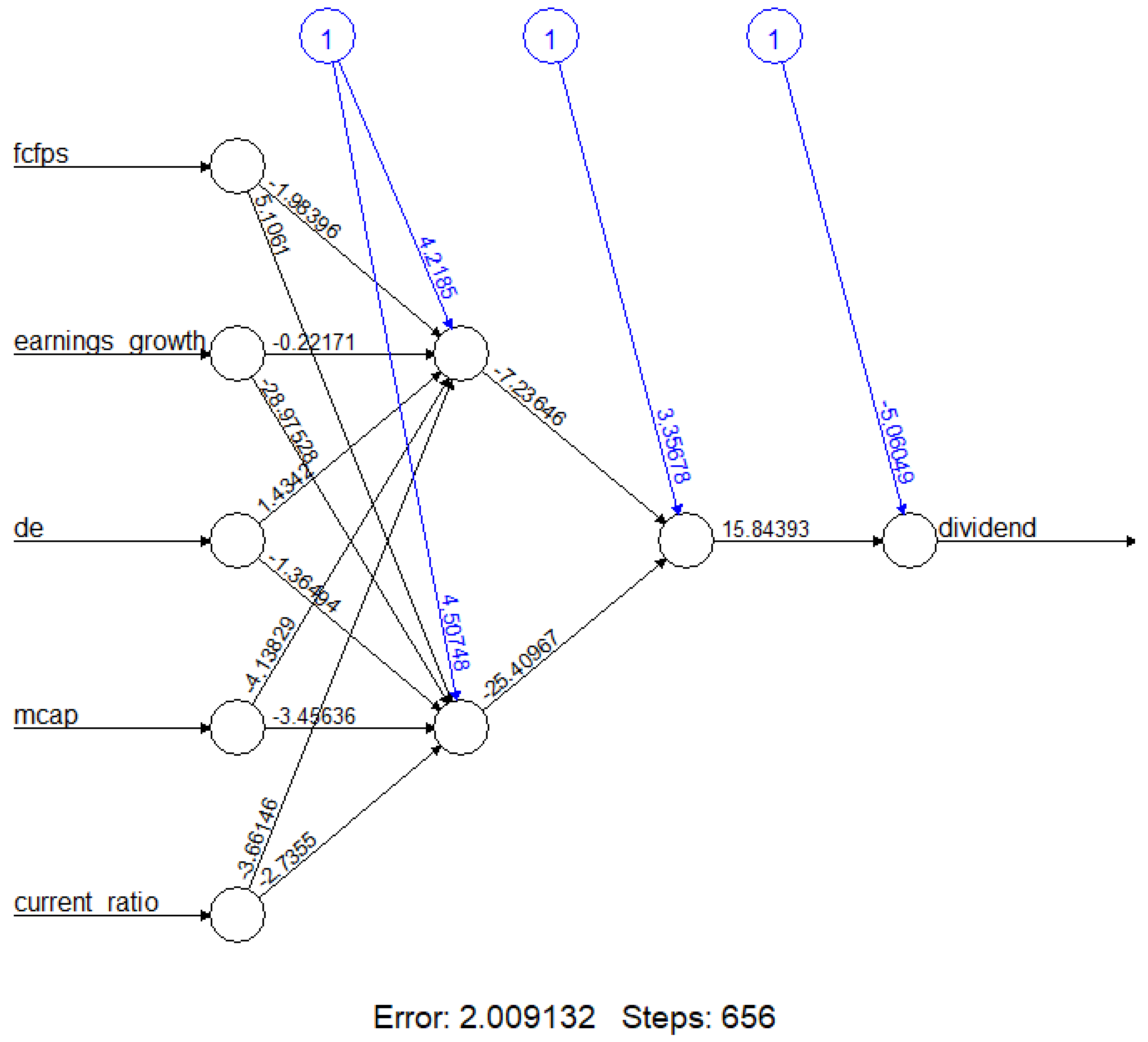
Observe that we are:

- Using neuralnet to “regress” the dependent “dividend” variable against the other independent variables
- Setting the number of hidden layers to (2,1) based on the hidden=(2,1) formula
- The linear.output variable is set to FALSE, given the impact of the independent variables on the dependent variable (dividend) is assumed to be non-linear
- The threshold is set to 0.01, meaning that if the change in error during an iteration is less than 1%, then no further optimization will be carried out by the model

```
#Neural Network
library(neuralnet)
nn <- neuralnet(dividend ~ fcfps + earnings_growth + de + mcap + current_ratio,
data=trainset, hidden=c(2,1), linear.output=FALSE, threshold=0.01)
nn$result.matrix
 1
error                               2.009132325086
reached.threshold                   0.009397262732
steps                             656.000000000000
Intercept.to.1layhid1               4.218503631309
fcfps.to.1layhid1                  -1.983962028656
earnings_growth.to.1layhid1        -0.221714930628
de.to.1layhid1                      1.434199120421
mcap.to.1layhid1                   -4.138291152052
current_ratio.to.1layhid1          -3.661464598242
Intercept.to.1layhid2               4.507481654661
fcfps.to.1layhid2                   5.106102332194
earnings_growth.to.1layhid2       -28.975275309124
de.to.1layhid2                     -1.364942921824
mcap.to.1layhid2                   -3.456363761740
current_ratio.to.1layhid2          -2.735500873964
Intercept.to.2layhid1               3.356782626838
1layhid.1.to.2layhid1              -7.236462167603
1layhid.2.to.2layhid1             -25.409665545003
Intercept.to.dividend              -5.060491300491
2layhid.1.to.dividend              15.843933229393
```

We now generate the error of the neural network model, along with the weights between the inputs, hidden layers, and outputs.

```
plot(nn)
```


1
1
1
fcfps
earnings_growth
de
mcap
current_ratio
dividend
-1.98396
5.1061
4.2185
-0.22171
-28.97528
1.4347
-1.36494
-4.13829
-3.45636
-3.66146
-2.7355
4.50748
-7.23646
-25.40967
3.35678
15.84393
-5.06049
Error: 2.009132 Steps: 656

Testing The Accuracy of The Model

- Our neural network has been created using the training data. We then compare this to the test data to gauge the accuracy of the neural network forecast.

```
#Test the resulting output
temp_test <- subset(testset, select = c("fcfps","earnings_growth", "de", "mcap",
"current_ratio"))
head(temp_test)
nn.results <- compute(nn, temp_test)
results <- data.frame(actual = testset$dividend, prediction = nn.results$net.result) >
results
    actual     prediction
161      0 0.006302516378
162      1 0.999908849573
163      0 0.013433101190
164      0 0.010097946967
165      0 0.008977085552
```

- The “subset” function is used to eliminate the dependent variable from the test data
- The “compute” function then creates the prediction variable
- A “results” variable then compares the predicted data with the actual data
- A confusion matrix is then created with the table function to compare the number of true/false positives and negatives

Confusion Matrix

- We round up our results using **sapply** and create a confusion matrix to compare the number of true/false positives and negatives.

```
roundedresults<-sapply(results,round,digits=0)
roundedresultsdf=data.frame(roundedresults)
attach(roundedresultsdf)
table(actual,prediction)

       prediction
actual  0  1
     0 17  0
     1  3 20
```

A confusion matrix is used to determine the number of true and false positives generated by our predictions. The model generates 17 true negatives (0's), 20 true positives (1's), while there are 3 false negatives.

Ultimately, we yield an 92.5% (37/40) accuracy rate in determining whether a stock pays a dividend or not.