
# Neural Networks

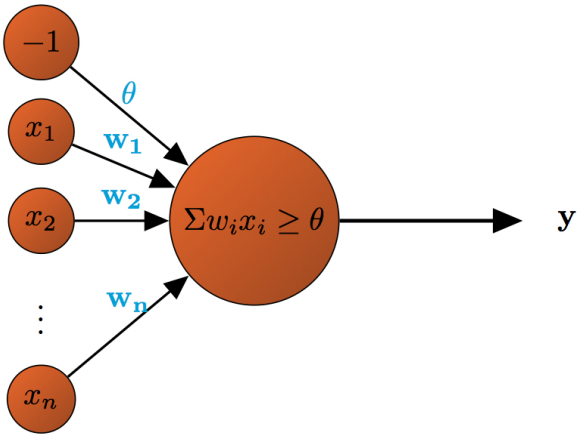
**The purpose of this document is to review neural networks, discuss training rules and provide an example illustrating backpropagation.**



In lesson three of the course, Michael covers neural networks. These notes are intended to fill in some details about the various training rules.

As stated in the lectures, a neural network is a learning structure designed to mimic the function of a web of biological neurons. The most basic (artificial) neural network is one that consists of just a single neuron, called a perceptron. We begin by recounting how a perceptron works, and discussing the most basic training rule: the perceptron rule.

***Perceptrons and the Perceptron Rule:***



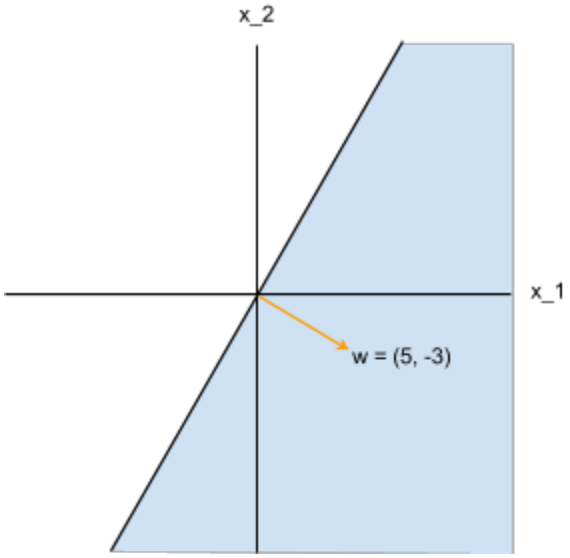

In the picture above, $\theta$ is a firing threshold, the $w_i$ are weights, $x_i$ are the inputs, and $y_i$ is the (discrete) output. In general, neural networks can have continuous output, but we will restrict to discrete output for the purposes of this document. Mathematically, the perceptron computes output according to the following rule:

$$\hat{y} = \chi\left( \sum_{i=1}^{n} w_i x_i - \theta \right) = \chi( w \cdot x - \theta ) .$$

Here, $\chi(a)$ is a characteristic function defined by

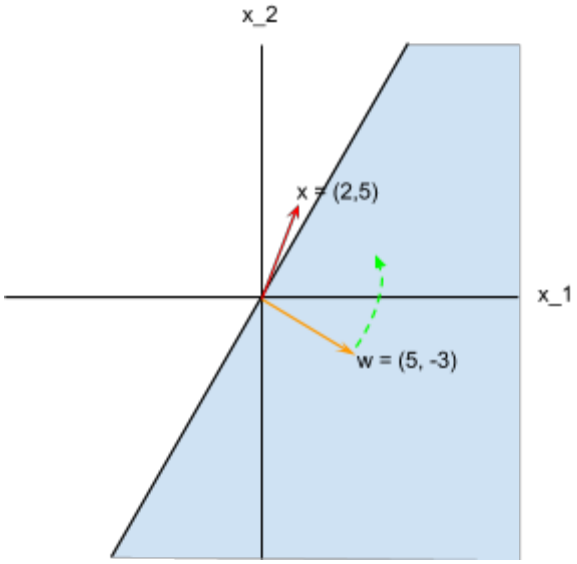
$$\chi(a) = 1 \text{ if } a \geq 0 \text{ and } \chi(a) = 0 \text{ if } a < 0 .$$

We can think of the perceptron as a hyperplane in n dimensions, perpendicular to the vector $w = (w_1, w_2, \ldots, w_n)$. The perceptron classifies things on one side of the hyperplane as positive and things on the other side as negative. Below is a picture of the hyperplane for $w = (5, -3)$ with $\theta = 0$.



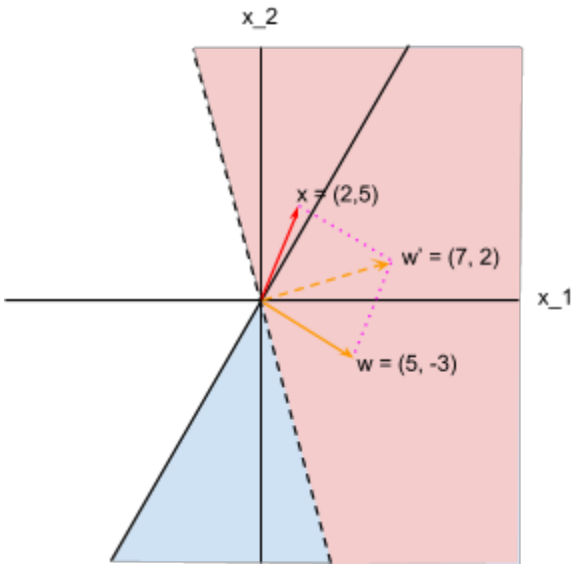

In order for the perceptron to be useful as a structure for learning, we need to be able to train it with new data. To see how we might do this, consider the perceptron with two inputs $x_1$ and $x_2$. Let's assume that the weights and firing threshold are as above. Now suppose we are given the training point $(2,\ 5,\ 1)$ as a $(x_1,\ x_2,\ y)$- tuple.

We can see that as of now, our perceptron doesn't do a very good job with this training point. The perceptron outputs $\hat{y} = 0$, when the training data has a target $y = 1$. How could we change the perceptron to predict better? The answer is that our best bet is to try to change the weights (since we can't really change the data...). Geometrically, we might think of this as rotating the hyperplane to put the training data on the correct side of the boundary.




How should we perform this rotation? By this, we are asking for a method of rotating the plane that we can consistently repeat in similar situations.

One possible answer is that we could create a new weight vector $w'$ by adding $x$ and $w$ together. Geometrically, this will give us the new weight vector shown below together with the new hyperplane and classification region:




Perfect! Our new weight vector allows the perceptron classify the data $x = (2, 5)$ correctly. So, the big question is: can we derive a

general rule to follow that captures the procedure above? One possibility is the rule:

$$w' = w + x.$$

While this is a good guess (this rule does capture the situation above), it fails in the case that our perceptron incorrectly classifies an input as positive when the target is negative. Why? Well, in this second kind of misclassification, we actually want to rotate *away* from the data point in order to put the point on the negative side of the boundary (it is a *very* worthwhile exercise to draw some examples to convince yourself of this!). The rule also doesn't capture the situation where we classify the data correctly, in which case we shouldn't need to update the weights at all. In order to account for both types of misclassification and the possibility of correct classification we need to make a couple of observations. If $y$ denotes the target output and $\hat{y}$ denotes the output of the perceptron, then:

- $y - \hat{y} = 0$ when we classify a data point correctly.
- $y - \hat{y} = 1$ when we incorrectly classify a positive data point (+1) as negative (-1).
- $y - \hat{y} = -1$ when we incorrectly classify a negative data point as positive.

With these observations in mind, we propose the rule

$$w' = w + (y - \hat{y})x$$

as a means to update the weight vector. As it turns out, this is (nearly) the perceptron training rule! In practice, we might want to control how much the hyperplane can rotate. This is done by including a multiplicative "learning rate" $\eta$ in the formula above. The final result is:

$$w' = w + \eta(y - \hat{y})x$$

When written in terms of vector components, and denoting the last term by $\Delta w$, we have the familiar form seen in the lectures:

$$\begin{aligned} w_i' &= w_i + \Delta w_i \\ &= w_i + \eta(y - \hat{y})x_i \end{aligned}$$

We note that it can be shown that the procedure given above for updating weights can be shown to converge within a finite number of applications, and it will correctly classify all training examples, provided $\eta$ is sufficiently small and the training examples are linearly separable[1].

***Gradient Descent / Delta Rule:***

The perceptron rule outlined above works fine when the data is linearly separable, but can fail to converge otherwise. For more complicated data, we need a better training rule. One idea is to create a function that measures how much error we have, and then try to adjust the weights to minimize that error. A first go at creating this error function might look like:

$$E(w) = \sum_{d \in D} |y_d - \hat{y}_d| ,$$

where $D$ is the set of all training examples. This makes some sense; we are just summing up the magnitude of the error over all training examples. However, there is a problem with this formulation. In order to move towards the minimum of $E(w)$, we will need to use derivatives, but neither the absolute value function $|\cdot|$ nor the thresholded perceptron output $\hat{y}$ are differentiable. To fix these problems people have traditionally considered the unthresholded perceptron given by

$$\hat{y} = w \cdot x ,$$

and replaced the absolute value function $|\cdot|$ with a quadratic. The resulting error formula is:

$$E(w) = \frac{1}{2} \sum_{d \in D} (y_d - w \cdot x_d)^2 .$$

The extra factor of $1/2$ is just to make the derivative expression simpler, and is not necessary. The idea here is that if we can choose weights to make the unthresholded perceptron, or *linear unit*, produce values $w \cdot x_d$ that are close to the true values $y_d$, then the thresholded

[1] Minsky, Marvin, and Papert Seymour. "Perceptrons." (1969).

perceptron will produce good values as well.

To update the weights, we use the gradient descent method described in the lectures. Also, a derivation of the gradient of the error function is also given in the lectures. The result is

$$\Delta w_i = \eta \sum_{d \in D} (y_d - w \cdot x_d) x_{id} .$$

Note that for a single data point, the training rule from gradient descent takes the form

$$\Delta w_i = \eta (y - w \cdot x) x_i ,$$

which is very similar to the perceptron training rule!

***Neural Networks:***

Great, so now that we have an advanced and flexible training rule for our linear units, we can start linking the units together to form networks. With our newly created networks we will be able to model ever-more-sophisticated functions, right?

Well, not quite. It turns out we still have a big problem.

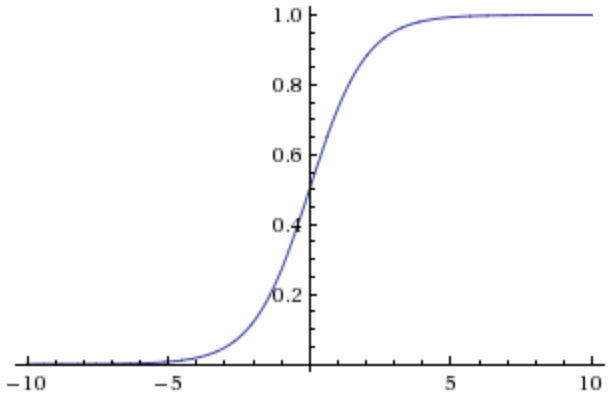
To use gradient descent above, we had to use a linear unit. This unit is just a linear function, and when you link a lot of these units together you get a linear combination of linear functions, which is …. linear. So we have a conundrum; we want the nonlinearity of the thresholded perceptron, with the robust training rule given by gradient descent.

The solution is given to us by the sigmoid function

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

A plot of the sigmoid is given below:

Plot:




Notice that the sigmoid looks roughly like a smoothed thresholding function. Also, it turns out that the sigmoid is smooth. In addition, the sigmoid has the awesome differentiation property

$$\frac{d\sigma(x)}{dx} = \sigma(x) \cdot (1 - \sigma(x)) .$$

So we can make perceptrons (sigmoid units) that operate according to the rule

$$\hat{y} = \sigma(w \cdot x) ,$$

giving us the best of both worlds: the sigmoid units are nonlinear and we can still use gradient descent.

The training rule for a sigmoid unit, given by gradient descent is:

$$\Delta w_i = \eta \sum_{d \in D} (y_d - \sigma) \cdot \sigma \cdot (1 - \sigma) x_{id} .$$

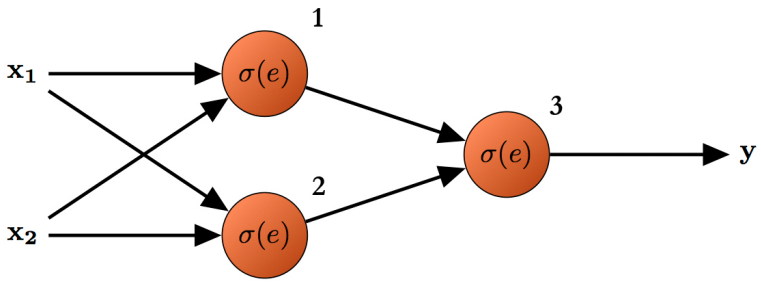
Here we are using $\sigma$ as shorthand for $\sigma(w \cdot x_d)$.

The final topic of consideration regarding neural network training rules is that of updating all of the weights in a network. Since the sigmoid units are all linked together, the inputs and outputs for a unit depend can depend on other units. The application of gradient descent to an entire network at once is called backpropagation. To see an example that demonstrates where the name comes from, continue to the next

section.

***Backpropagation:***

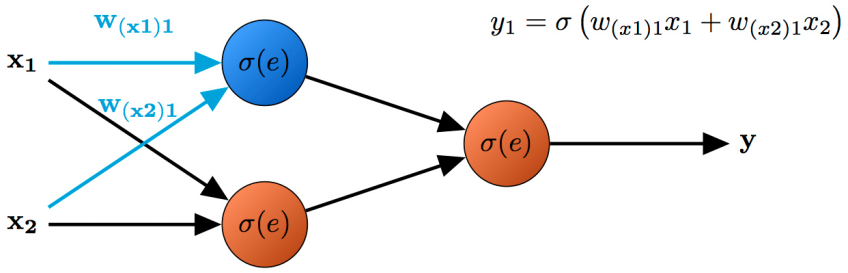
In the sketch of neural networks, Michael discusses backpropagation as a means for a neural network to learn. Here we demonstrate backpropagation for a neural network with two inputs and one output. The network is shown in the diagram below, with the nodes numbered.



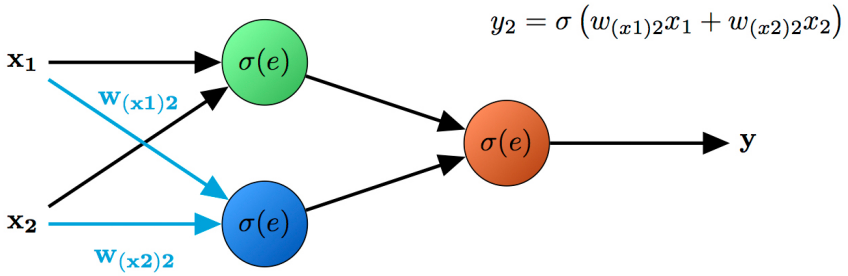

In our example, the activation function will be given by the sigmoid

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

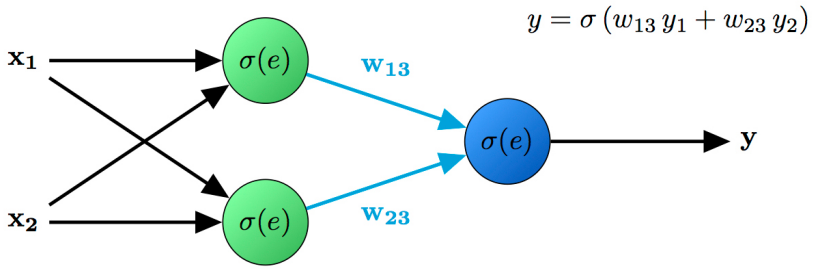
which is discussed above. To begin, the neural network is populated randomly with small weights. Here $w_{(x1)1}$ indicates the weight between the input $x_1$ and node 1. The output of node one is denoted by $y_1$, and the formula for this output is shown below.



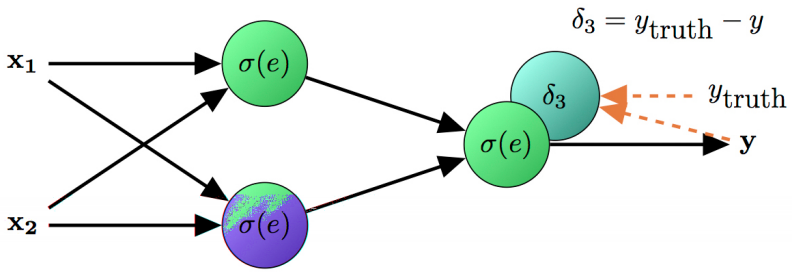

After node 1 output is computed, we can use the same method to compute $y_2$, the output at node 2. This is shown in the next diagram.




Finally, the weights $w_{13}$ and $w_{23}$ can be used along with the sigmoid function to compute the final output $y$.

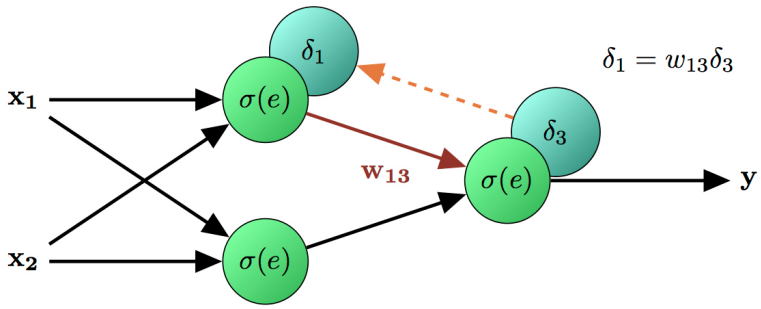



With the output $y$ in hand, we can compute the error $\delta_3$ between the true value $y_{truth}$ and $y$. This is the beginning of backpropagation of errors.

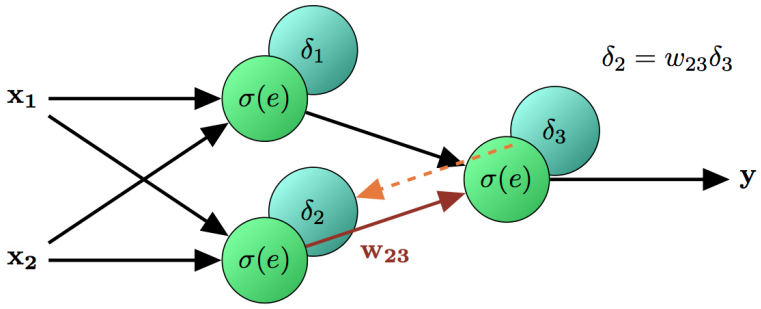



Once error $\delta_3$ is computed, we can use the weight $w_{13}$ to compute

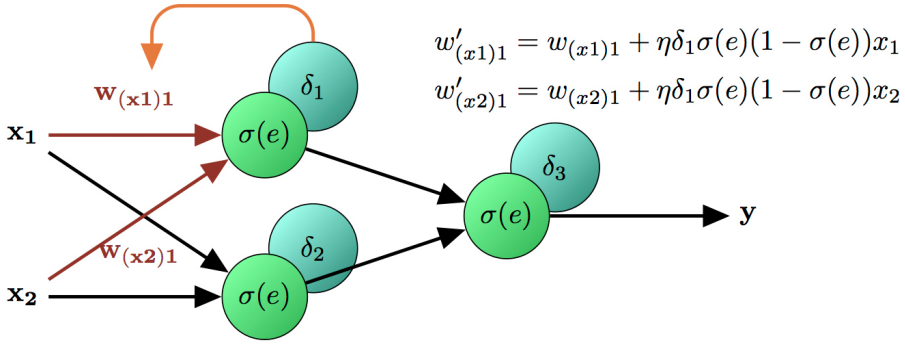
the error $\delta_1$ for node 1.



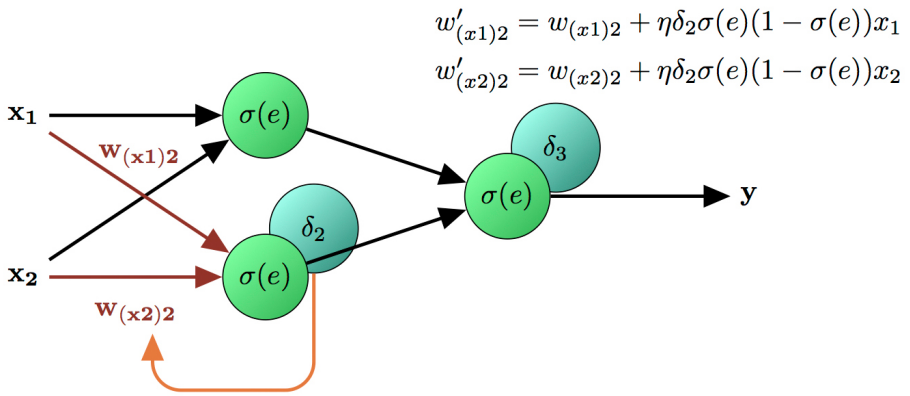

Similarly, we can can compute the error $\delta_2$ for node 2.



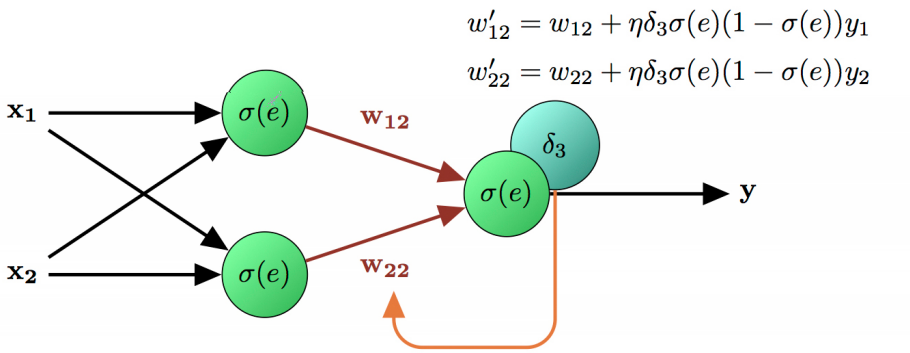

And finally, we can use the backpropagated error along with the inputs to compute updated weights $w'_{(x1)1}$ and $w'_{(x2)1}$. This takes place according to the gradient descent formula for the sigmoid function $\sigma$. In the equations below, $\eta$ is a parameter which controls the rate of learning. Also, in the formulas below, $e = w_{(x1)1} \cdot x_1 + w_{(x2)1} \cdot x_2$.



Similarly, we can compute updated weights $w'_{(x1)1}$ and $w'_{(x2)1}$. Analogous to the previous diagram, in the formulas below, $e = w_{(x1)2} \cdot x_1 + w_{(x2)2} \cdot x_2$.




Finally, we can compute updated weights $w'_{12}$ and $w'_{22}$ using the same process.




For a more in depth example with a hidden layer (and the inspiration for this example), refer to this link.