The Clever Machine

Topics in Computational Neuroscience & Machine Learning

A Gentle Introduction to Artificial Neural Networks

SEP 11
Posted by **dustinstansbury**

Introduction

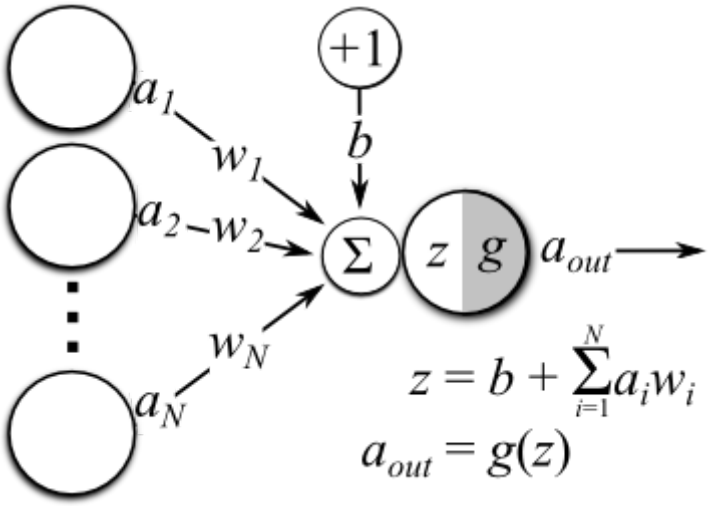
Though many phenomena in the world can be adequately modeled using linear regression or classification, most interesting phenomena are generally nonlinear in nature. In order to deal with nonlinear phenomena, there have been a diversity of nonlinear models developed. For example parametric models assume that data follow some parameteric class of nonlinear function (e.g. polynomial, power, or exponential), then fine-tune the shape of the parametric function to fit observed data. However this approach is only helpful if data are fit nicely by the available catalog of parametric functions. Another approach, kernel-based methods, transforms data non-linearly into an abstract space that measures distances between observations, then predicts new values or classes based on these distances. However, kernel methods generally involve constructing a kernel matrix that depends on the number of training observations and can thus be prohibitive for large data sets. Another class of models, the ones that are the focus of this post, are artificial neural networks (ANNs). ANNs are nonlinear models motivated by the physiological architecture of the nervous system. They involve a cascade of simple nonlinear computations that when aggregated can implement robust and complex nonlinear functions. In fact, depending on how they are constructed, ANNs can approximate any nonlinear function, making them a quite powerful class of models (note that this property is not reserved for ANNs; kernel methods are also considered "universal approximators"; however, it turns out that neural networks with multiple layers are more efficient at approximating arbitrary functions than other methods. I refer the interested reader to more in-depth discussion (http://yann.lecun.com/exdb/publis/pdf/bengio-lecun-07.pdf) on the topic.).

In recent years ANNs that use multiple stages of nonlinear computation (aka "deep learning") have been able obtain outstanding performance on an array of complex tasks ranging from visual object recognition to natural language processing. I find ANNs super interesting due to their computational power and their intersection with computational neuroscience. However, I've found that most of the available tutorials on ANNs are either dense with formal details and contain little information about implementation or any examples, while others skip a lot of the mathematical detail

and provide implementations that seem to come from thin air. This post aims at giving a more complete overview of ANNs, including (varying degrees of) the math behind ANNs, how ANNs are implemented in code, and finally some toy examples that point out the strengths and weaknesses of ANNs.

Single-layer Neural Networks

The simplest ANN (Figure 1) takes a set of observed inputs $\mathbf{a} = (a_1, a_2, ..., a_N)$, multiplies each of them by their own associated weight $\mathbf{w} = (w_1, w_2, ...w_N)$, and sums the weighted values to form a pre-activation z.Oftentimes there is also a bias b that is tied to an input that is always +1 included in the preactivation calculation. The network then transforms the pre-activation using a nonlinear activation function $g(z)$ to output a final activation a_{out}.



(https://theclevermachine.files.wordpress.com/2014/09/perceptron2.png)
Figure 1: Diagram of a single-layered artificial neural network.

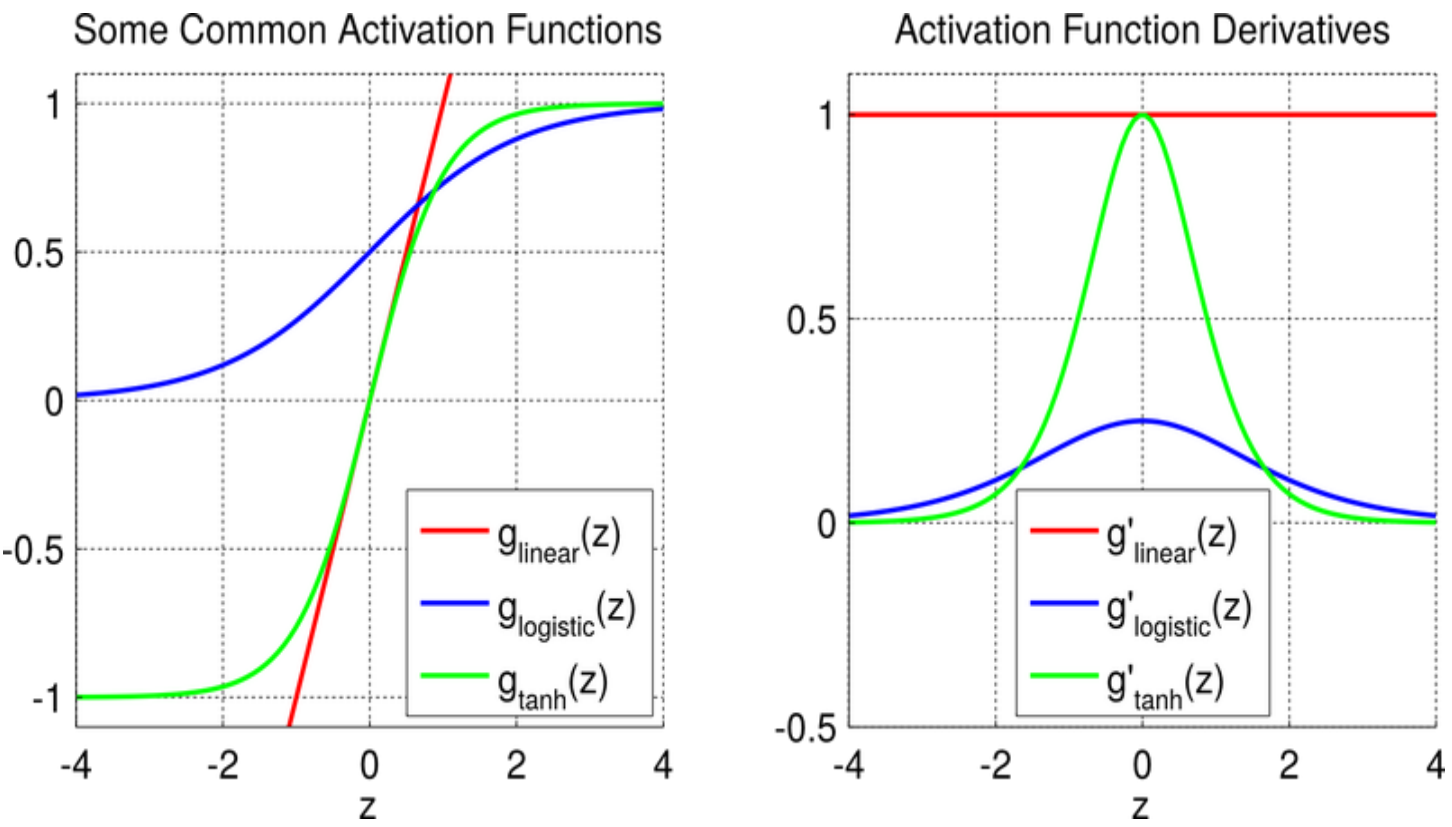
There are many options available for the form of the activation function $g(z)$, and the choice generally depends on the task we would like the network to perform. For instance, if the activation function is the identity function:

$$g_{\text{linear}}(z) = z,$$

which outputs continuous values $a_{linear} \in (-\infty, \infty)$, then the network implements a linear model akin to used in standard linear regression. Another choice for the activation function is the logistic sigmoid:

$$g_{\text{logistic}}(z) = \frac{1}{1+e^{-z}},$$

which outputs values $a_{logistic} \in (0, 1)$. When the network outputs use the logistic sigmoid activation function, the network implements linear binary classification. Binary classification can also be implemented using the hyperbolic tangent function, $\tanh(z)$, which outputs values $a_{\tanh} \in (-1, 1)$ (note that the classes must also be coded as either -1 or 1 when using $\tanh$. Single-layered neural networks used for classification are often referred to as "perceptrons," a name given to them when they were first developed in the late 1950s.



(https://theclevermachine.files.wordpress.com/2014/09/act-funs.png)

Figure 2: Common activation functions functions used in artificial neural, along with their derivatives

To get a better idea of what these activation function do, their outputs for a given range of input values are plotted in the left of Figure 2. We see that the logistic and tanh activation functions (blue and green) have the quintessential sigmoidal "s" shape that saturates for inputs of large magnitude. This behavior makes them useful for categorization. The identity / linear activation (red), however forms a linear mapping between the input to the activation function, which makes it useful for predicting continuous values.

A key property of these activation functions is that they are all smooth and differentiable. We'll see later in this post why differentiability is important for training neural networks. The derivatives for each of these common activation functions are given by (for mathematical details on calculating these derivatives, see this post (https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/)):

$$\begin{aligned} g'_{\text{linear}}(z) &= 1 \\ g'_{\text{logistic}}(z) &= g_{\text{logistic}}(z)(1 - g_{\text{logistic}}(z)) \\ g'_{\tanh}(z) &= 1 - g^2_{\tanh}(z) \end{aligned}$$

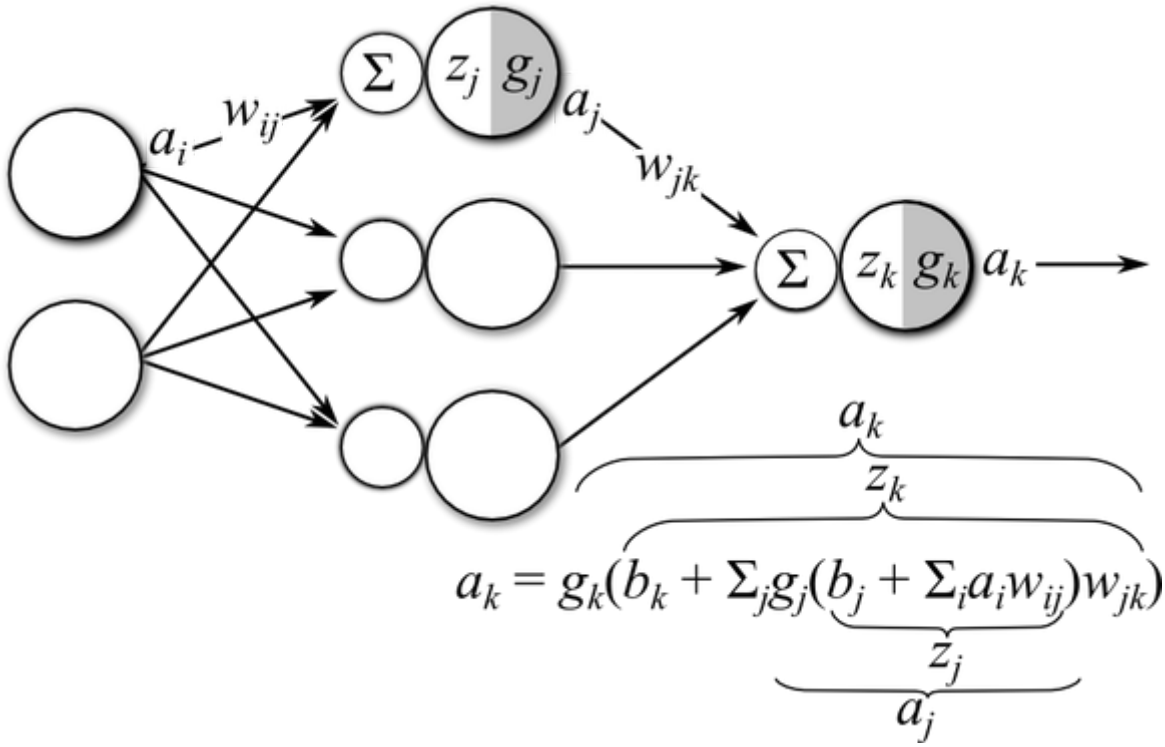
Each of the derivatives are plotted in the right of Figure 2. What is interesting about these derivatives is that they are either a constant (i.e. 1), or are can be defined in terms of the original function. This makes them extremely convenient for efficiently training neural networks, as we can implement the gradient using simple manipulations of the feed-forward states of the network.

Code Block 1: Defines standard activation functions and generates Figure 2:

```
+ expand source
```

Multi-layer Neural Networks

As was mentioned above, single-layered networks implement linear models, which doesn't really help us if we want to model nonlinear phenomena. However, by considering the single layer network diagrammed in Figure 1 to be a basic building block, we can construct more complicated networks, ones that perform powerful, nonlinear computations. Figure 3 demonstrates this concept. Instead of a single layer of weights between inputs and output, we introduce a set of single-layer networks between the two. This set of intermediate networks is often referred to as a "hidden" layer, as it doesn't directly observe input or directly compute the output. By using a hidden layer, we form a mult-layered ANN. Though there are many different conventions for declaring the actual number of layers in a multi-layer network, for this discussion we will use the convention of the number of *distinct sets of trainable weights* as the number of layers. For example, the network in Figure 3 would be considered a 2-layer ANN because it has two layers of weights: those connecting the inputs to the hidden layer (w_{ij}), and those connecting the output of the hidden layer to the output layer(w_{jk}).



(https://theclevermachine.files.wordpress.com/2014/09/neural-net1.png)

Figure 3: Diagram of a multi-layer ANN. Each node in the network can be considered a single-layered ANN (for simplicity, biases are not visualized in graphical model)

Multi-layer neural networks form compositional functions that map the inputs nonlinearly to outputs. If we associate index i with the input layer, index j with the hidden layer, and index k with the output layer, then an output unit in the network diagrammed in Figure 3 computes an output value a_k given and input a_i via the following compositional function:

$$a_{\text{out}} = a_k = g_k(b_k + \sum_j g_j(b_j + \sum_i a_i w_{ij})w_{jk}.$$

Here z_l is the is the pre-activation values for units for layer l, $g_l()$ is the activation function for units in that layer (assuming they are the same), and $a_l = g_l(z_l)$ is the output activation for units in that layer. The weight $w_{l-1,l}$ links the outputs of units feeding into layer l to the activation function of

units for that layer. The term b_l is the bias for units in layer l.

As with the single-layered ANN, the choice of activation function for the output layer will depend on the task that we would like the network to perform (i.e. categorization or regression), and follows similar rules outlined above. However, it is generally desirable for the hidden units to have *nonlinear* activation functions (e.g. logistic sigmoid or tanh). This is because multiple layers of linear computations can be equally formulated as a single layer of linear computations. Thus using linear activations for the hidden layers doesn't buy us much. However, as we'll see shortly, using linear activations for the output unit activation function (in conjunction with nonlinear activations for the hidden units) allows the network to perform nonlinear regression.

Training neural networks & gradient descent

Training neural networks involves determining the network parameters that minimize the errors that the network makes. This first requires that we have a way of quantifying error. A standard way of quantifying error is to take the squared difference between the network output and the target value:

$$E = \frac{1}{2}(\text{output} - \text{target})^2$$

(Note that the squared error is not chosen arbitrarily, but has a number of theoretical benefits and considerations. For more detail, see the following post (https://theclevermachine.wordpress.com/2012/02/13/cutting-your-losses-loss-functions-predominance-of-sum-of-squares/)) With an error function in hand, we then aim to find the setting of parameters that minimizes this error function. This concept can be interpreted spatially by imagining a "parameter space" whose dimensions are the values of each of the model parameters, and for which the error function will form a surface of varying height depending on its value for each parameter. Model training is thus equivalent to finding point in parameter space that makes the height of the error surface small.

To get a better intuition behind this concept, let's define a super simple neural network, one that has a single input and a single output (Figure 4, bottom left). For further simplicity, we'll assume the network has no bias term and thus has a single parameter, w_1. We will also assume that the output layer uses the logistic sigmoid activation function. Accordingly, the network will map some input value a_0 onto a predicted output a_{out} via the following function.

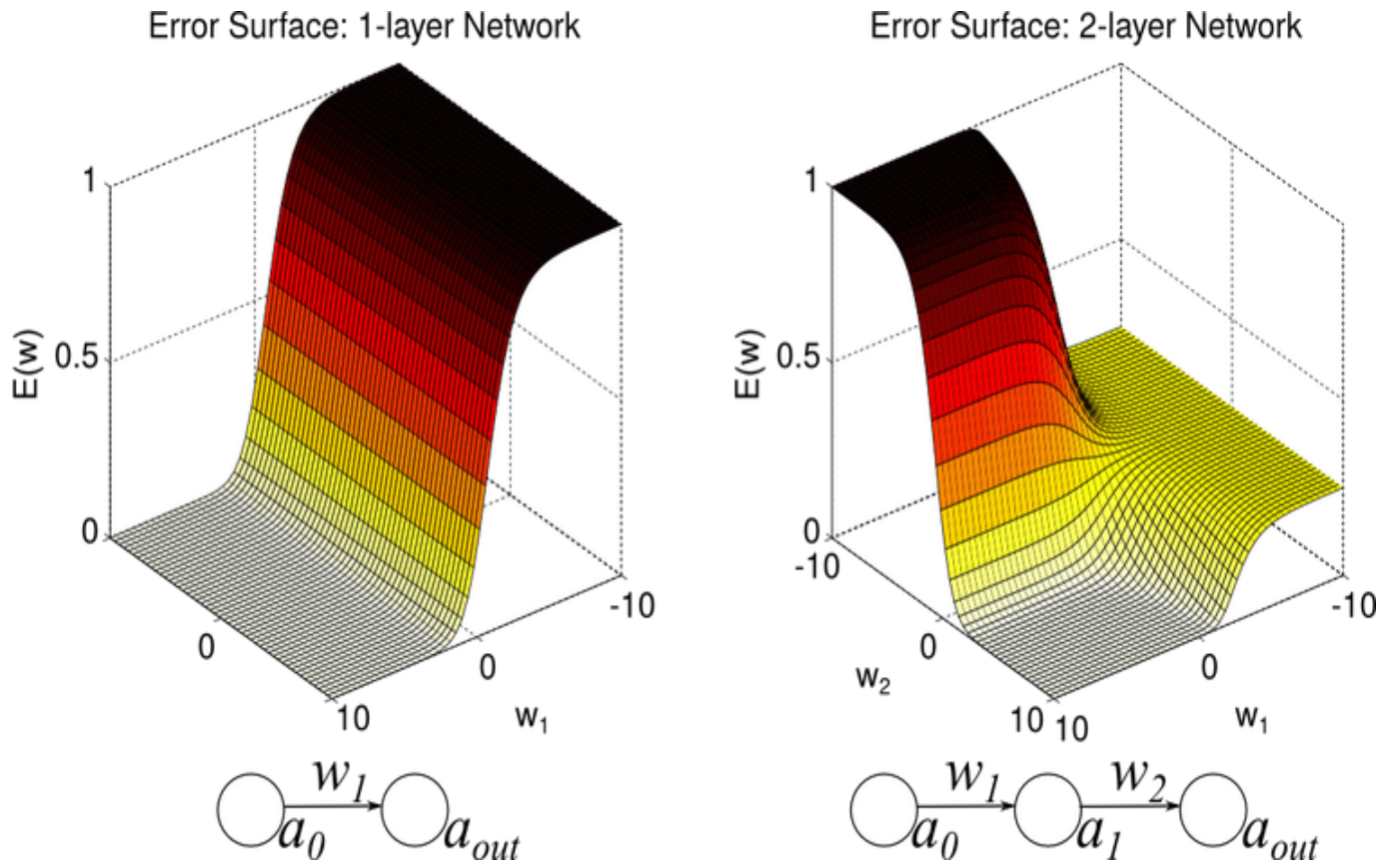
$$a_{\text{out}} = g_{\text{logistic}}(a_0 w_1)$$

Now let's say we want this simple network to learn the identity function: given an input of 1 it should return a target value of 1. Given this target value we can now calculate the value of the error function for each setting of w_1. Varying the value of w_1 from -10 to 10 results in the error surface displayed in the left of Figure 4. We see that the error is small for large positive values of w_1, while the error is large for strongly negative values of w_1. This not surprising, given that the output activation function is the logistic sigmoid, which will map large values onto an output of 1.

Things become more interesting when we move from a single-layered network to a multi-layered network. Let's repeat the above exercise, but include a single hidden node between the input and the output (Figure 4, bottom right). Again, we will assume no biases, and logistic sigmoid activations for both the hidden and output nodes. Thus the network will have two parameters: (w_1, w_2). Accordingly the 2-layered network will predict an output with the following function:

$$a_{\text{out}} = g_{\text{logistic}}(g_{\text{logistic}}(a_0 w_1) w_2)$$

Now, if we vary both w_1 and w_2, we obtain the error surface in the right of Figure 4.



(https://theclevermachine.files.wordpress.com/2014/09/nnet-error-surface8.png)

Figure 4: Error surface for a simple, single-layer neural network (left) and a 2-layer network (right). The goal is to map the input value 1 to the output value 1.

We see that the error function is minimized when both w_1 and w_2 are large and positive. We also see that the error surface is more complex than for the single-layered model, exhibiting a number of wide plateau regions. It turns out that the error surface gets more and more complicated as you increase the number of layers in the network and the number of units in each hidden layer. Thus, it is important to consider these phenomena when constructing neural network models.

Code Block 2: generates Figure 4 (assumes you have run Code Block 1):

```
+ expand source
```

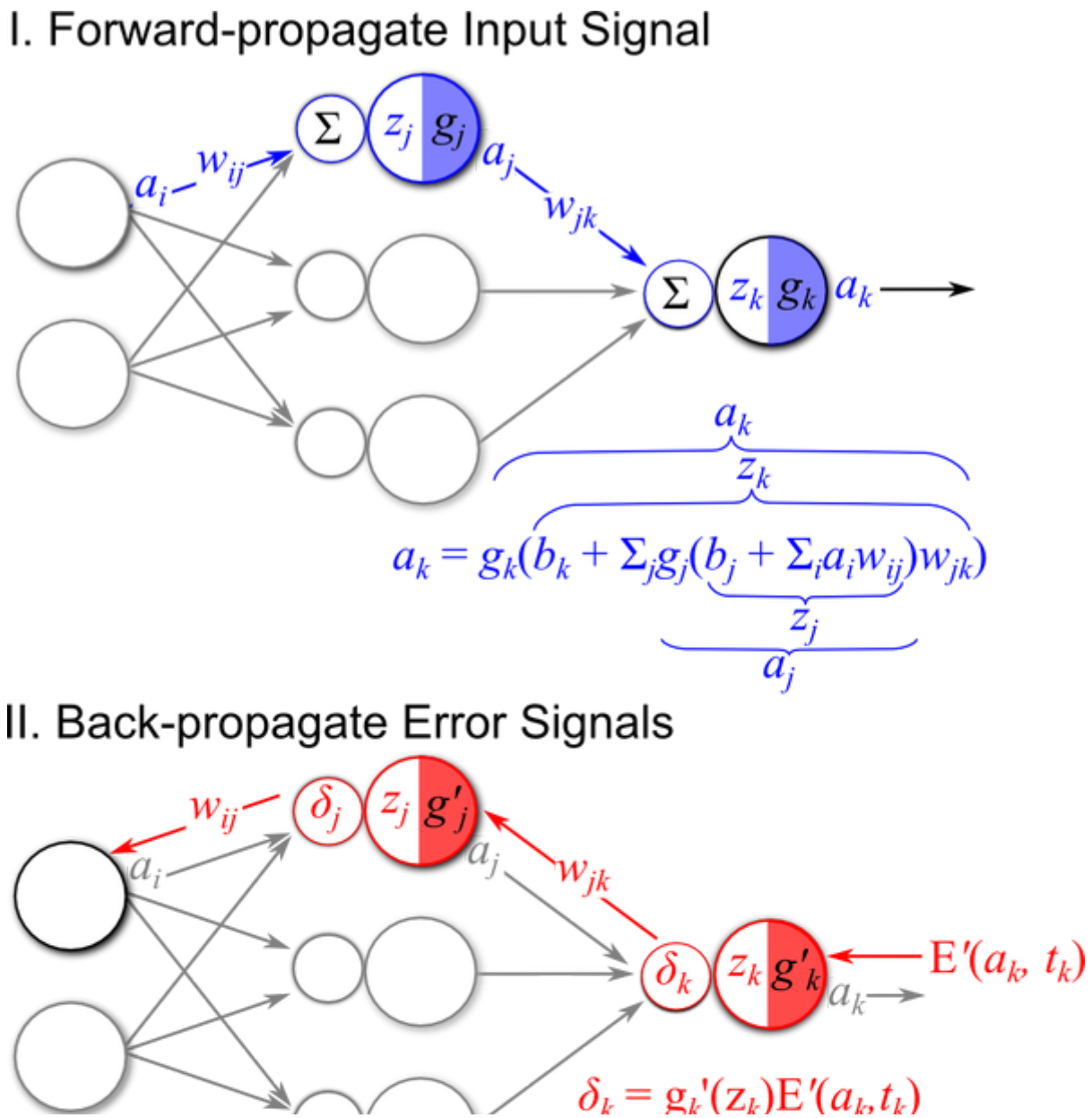
The examples in Figure 4 gives us a qualitative idea of how to train the parameters of an ANN, but we would like a more automatic way of doing so. Generally this problem is solved using *gradient descent*: The gradient descent algorithm first calculates the derivative / gradient of the error function with respect to each of the model parameters. This gradient information will give us the direction in parameter space that decreases the height of the error surface. We then take a step in that direction and repeat, iteratively calculating the gradient and taking steps in parameter space.

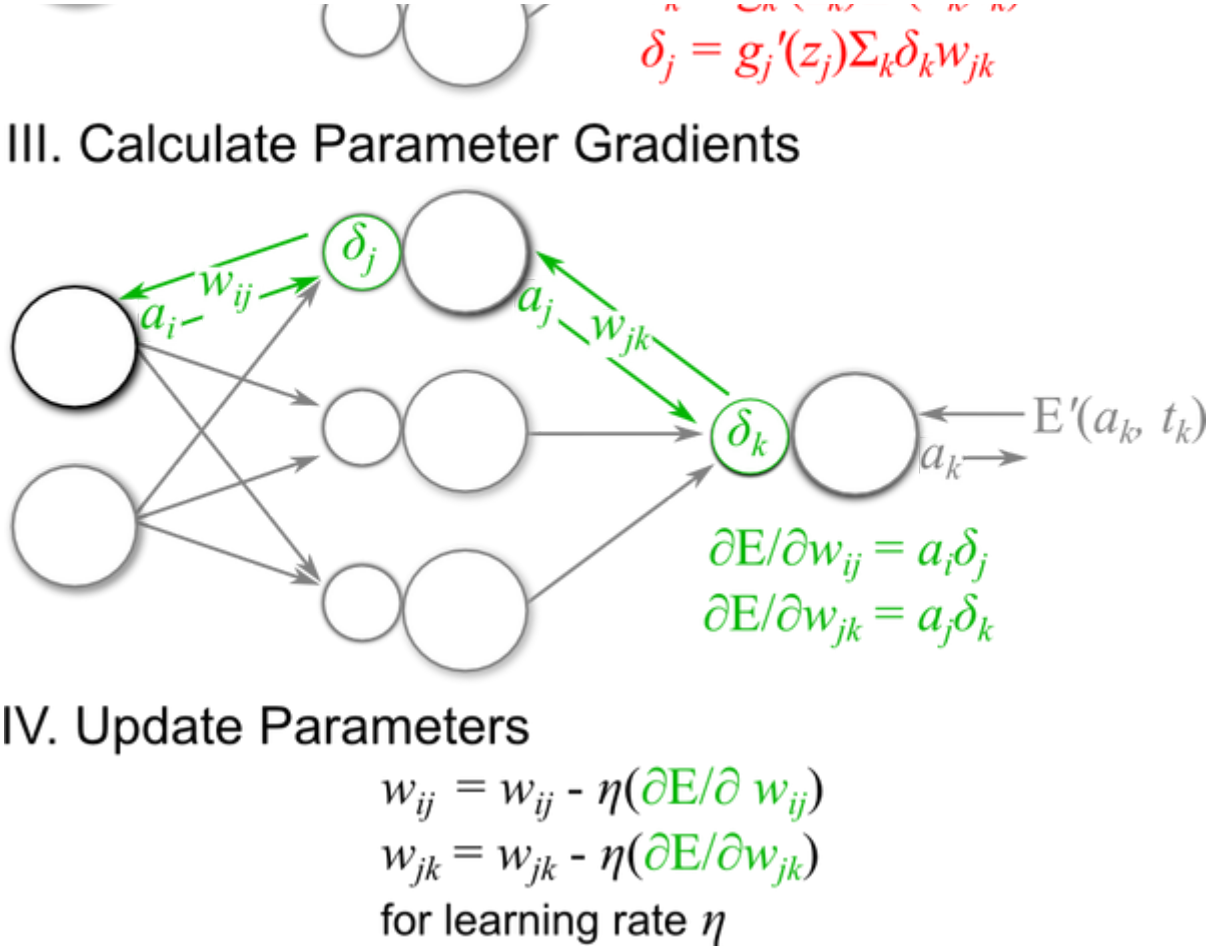
The backpropagation algorithm

It turns out that the gradient information for the ANN error surface can be calculated efficiently using a message passing algorithm known as the *backpropagation algorithm*. During backpropagation, input signals are forward-propagated through the network toward the outputs, and network errors are then calculated with respect to target variables and back-propagated backwards towards the inputs. The forward and backward signals are then used to determine the direction in the parameter space to move that lowers the network error.

The formal calculations behind the backpropagation algorithm can be somewhat mathematically involved and may detract from the general ideas behind the learning algorithm. For those readers who are interested in the math, I have provided the formal derivation of the backpropagation algorithm in the following post (https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/) (for those of you who are not interested in the math, I would also encourage you go over the derivation and try to make connections to the source code implementations provided later in the post).

Figure 5 demonstrates the key steps of the backpropagation algorithm. The main concept underlying the algorithm is that for a given observation we want to determine the degree of "responsibility" that each network parameter has for mis-predicting a target value associated with the observation. We then change that parameter according to this responsibility so that it reduces the network error.





(https://theclevermachine.files.wordpress.com/2014/09/fprop_bprop5.png)

Figure 5: The 4 main steps of the bacdkpropagation algorithm: I Forward propagate error signals to output, II Calculate output error E, and backpropagate error signal, III Use forward signal and backward signals to calculate parameter gradients, IV update network parameters.

In order to determine the network error, we first propagate the observed input forward through the network layers. This is Step I of the backpropagation algorithm, and is demonstrated in Figure 5-I. Note that in the Figure a_k could be considered network output (for a network with one hidden layer) or the output of a hidden layer that projects the remainder of the network (in the case of a network with more than one hidden layer). For this discussion, however, we assume that the index k is associated with the output layer of the network, and thus each of the network outputs is designated by a_k. Also note that when implementing this forward-propagation step, we should keep track of the feed-forward pre-activations z_l and activations a_l for all layers l, as these will be used for calculating backpropagated errors and error function gradients.

Step II of the algorithm is to calculate the network output error and backpropagate it toward the input. Let's again that we are using the sum of squared differences error function:

$$E = \frac{1}{2}\sum_{k \in K}(a_k - t_k)^2,$$

where we sum over the values of all k output units (one in this example). We can now define an "error signal" δ_k at the output node that will be backpropagated toward the input. The error signal is calculated as follows:

$$\begin{aligned} \delta_k &= g_k'(z_k)E'(a_k, t_k) \\ &= g_k'(z_k)(a_k - t_k) \end{aligned}.$$

Thus the error signal essentially weights the gradient of the error function by the gradient of the output activation function (notice there is a z_k term is used in this calculation, which is why we keep it around during the forward-propagation step). We can continue backpropagating the error signal toward the input by passing δ_k through the output layer weights w_{jk}, summing over all output nodes, and passing the result through the gradient of the activation function at the hidden layer $g'_j(z_j)$ (Figure 5-II). Performing these operations results in the back-propagated error signal for the hidden layer, δ_j:

$$\delta_j = g'_j(z_j) \sum_k \delta_k w_{jk},$$

For networks that have more than one hidden layer, this error backpropagation procedure can continue for layers $j-1, j-2, \ldots$, etc.

Step III of the backpropagation algorithm is to calculate the gradients of the error function with respect to the model parameters at each layer l using the forward signals a_{l-1}, and the backward error signals δ_l . If one considers the model weights $w_{l-1,l}$ at a layer l as linking the forward signal a_{l-1} to the error signal δ_l (Figure 5-III), then the gradient of the error function with respect to those weights is:

$$\frac{\partial E}{\partial w_{l-1,l}} = a_{l-1}\delta_l$$

Note that this result is closely related to the concept of Hebbian learning (http://en.wikipedia.org/wiki/Hebbian_theory) in neuroscience. Thus the gradient of the error function with respect to the model weight at each layer can be efficiently calculated by simply keeping track of the forward-propagated activations feeding into that layer from below, and weighting those activations by the backward-propagated error signals feeding into that layer from above!

What about the bias parameters? It turns out that the same gradient rule used for the weight weights applies, except that "feed-forward activations" for biases are always +1 (see Figure 1). Thus the bias gradients for layer l are simply:

$$\frac{\partial E}{\partial b_l} = (1)\delta_l = \delta_l$$

The fourth and final step of the backpropagation algorithm is to update the model parameters based on the gradients calculated in Step III. Note that the gradients point in the direction in parameter space that will *increase* the value of the error function. Thus when updating the model parameters we should choose to go in the opposite direction. How far do we travel in that direction? That is generally determined by a user-defined step size (aka learning rate) parameter, η. Thu,s given the parameter gradients and the step size, the weights and biases for a given layer are updated accordingly:

$$\begin{aligned} w_{l-1,l} &\leftarrow w_{l-1,l} - \eta \frac{\partial E}{\partial w_{l-1,l}} \\ b_l &\leftarrow b_l - \eta \frac{\partial E}{\partial b_l} \end{aligned}.$$

To train an ANN, the four steps outlined above and in Figure 5 are repeated iteratively by observing many input-target pairs and updating the parameters until either the network error reaches a tolerably low value, the parameters cease to update (convergence), or a set number of parameter updates has been achieved. Some readers may find the steps of the backpropagation somewhat ad hoc. However, keep in mind that these steps are formally coupled to the calculus of the optimization

problem. Thus I again refer the curious reader to check out the derivation (https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/) in order to make connections between the algorithm and the math.

Example: learning the OR & AND logical operators using a single layer neural network

Here we go over an example of training a single-layered neural network to perform a classification problem. The network is trained to learn a set of logical operators including the AND, OR, or XOR. To train the network we first generate training data. The inputs consist of 2-dimensional coordinates that span the input values (x_1, x_2) values for a 2-bit truth table:

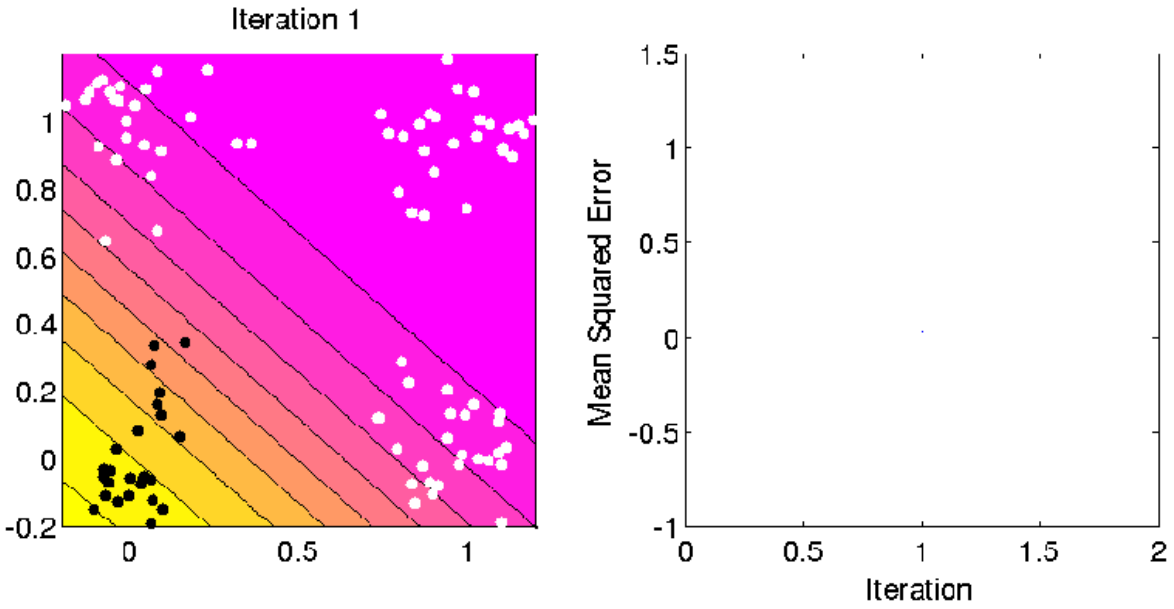
x_1	x_2	OR	AND	XOR
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

(https://theclevermachine.files.wordpress.com/2014/09/truth-table1.png)
Figure 6: Truth table values learned in classification examples

We then perturb these observations by adding Normally-distributed noise. To generate target variables, we categorize each observations by applying one of logic operators (See Figure 6) to the original (no-noisy) coordinates. We then train the network with the noisy inputs and binary categories targets using the gradient descent / backpropagation algorithm. The code implementation of the network and training procedures, as well as the resulting learning process are displayed below. (Note that in this implementation, I do not use the feed-forward activations to calculate the gradients as suggested above. This is simply to make the implementation of the learning algorithm more explicit in terms of the math. The same situation also applies to the other examples in this post).

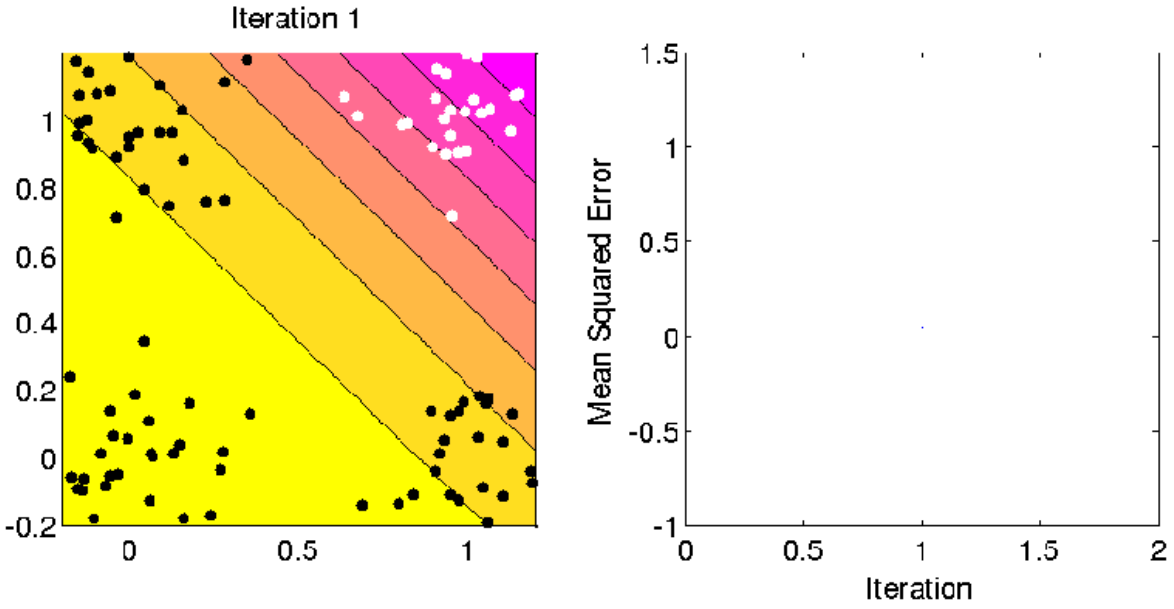
Code Block 3: Implements and trains a single-layer neural network for classification to learn logical operators (assumes you have run Code Block 1):

```
+ expand source
```



(https://theclevermachine.files.wordpress.com/2014/09/1layer-net-or.gif)
Figure 7: Single layer neural network (perceptron) learning a noisy OR mapping.

Figure 7 displays the procedure for learning the OR mapping. The left plot displays the training data and the network output at each iteration. White dots are training points categorized "1" while black dots are categorized "0". Yellow regions are where the network predicts values of "0", while magenta highlights areas where the network predicts "1". We see that the single-layer network is able to easily separate the two classes. The right plot shows how the error function decreases with each training iteration. The smooth trajectory of the error indicates that the error surface is also fairly smooth.



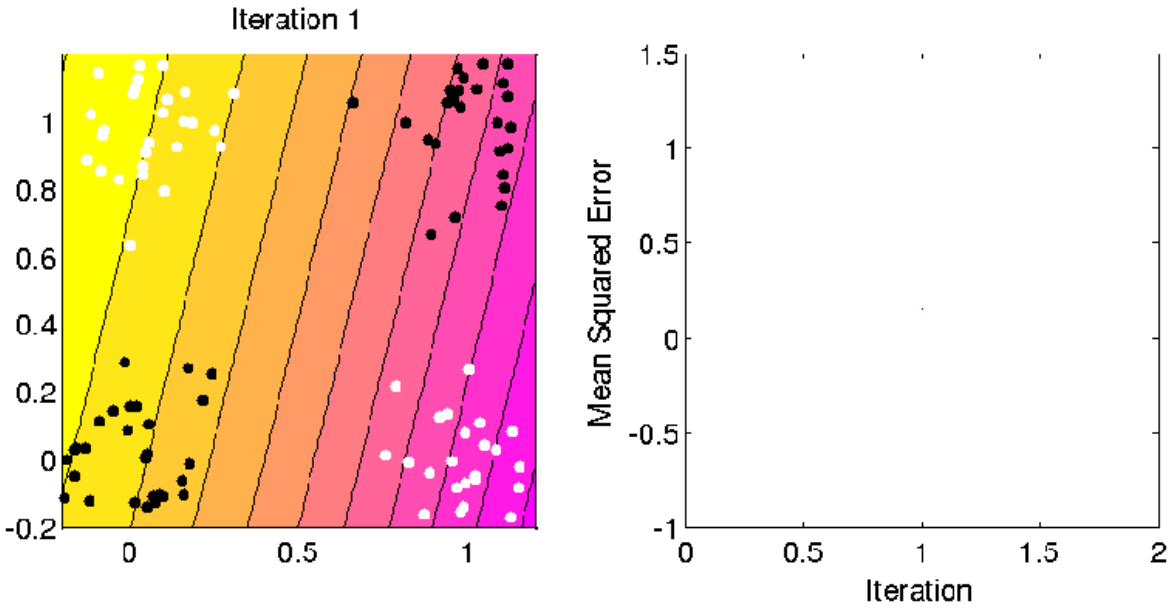
(https://theclevermachine.files.wordpress.com/2014/09/1layer-net-and1.gif)
Figure 8: Single layer neural network (perceptron) learning a noisy AND mapping.

Figure 8 demonstrates an analogous example, but instead learning the AND operator (by executing Code Block 3, after un-commenting line 11). Again, the categories can be easily separated by a plane, and thus the single-layered network easily learns an accurate predictor of the data.

Going Deeper: nonlinear classification and multi-layer neural networks

Figures 7 and 8 demonstrate how a single-layered ANN can easily learn the OR and AND operators. This is because the categorization criterion for these logical operators can be represented in the input space by a single linear function (i.e. line/plane). What about more complex categorization criterion that cannot be represented by a single plane? An example of a more complex binary classification criterion is the XOR operator (Figure 6, far right column).

Below we attempt to train the single-layer network to learn the XOR operator (by executing Code Block 3, after un-commenting line 12). The single layer network is unable to learn this nonlinear mapping between the inputs and the targets. However, it turns out we can learn the XOR operator using a multi-layered neural network.



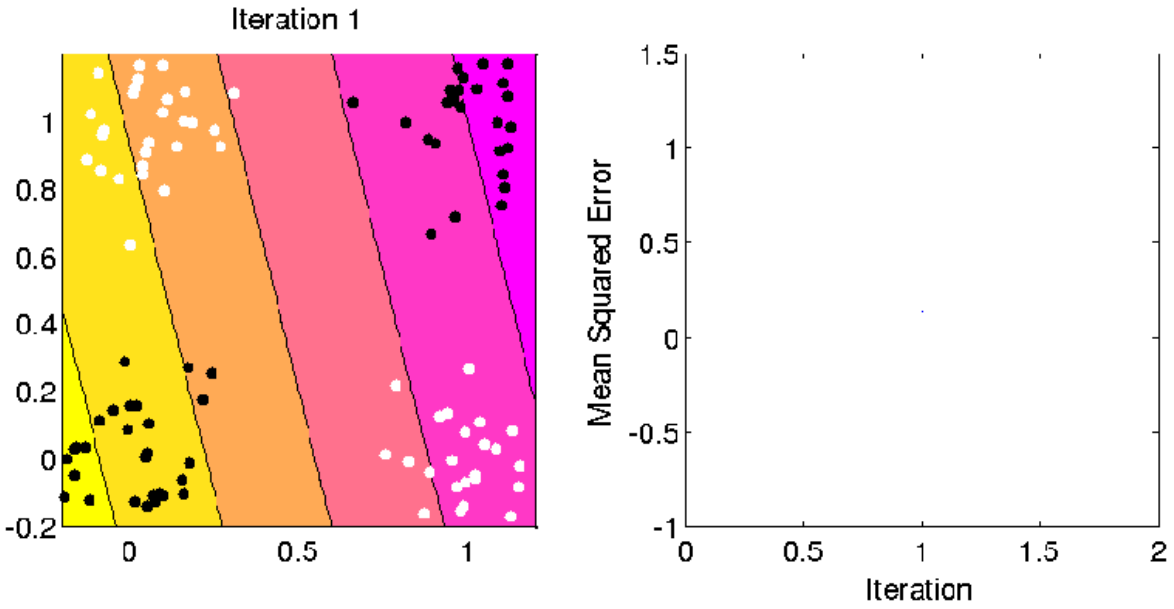
(https://theclevermachine.files.wordpress.com/2014/09/1layer-net-xor.gif)

Figure 9: Single layer neural network (perceptron) attempting to learn a noisy XOR mapping. The single layer network chokes on this nonlinear problem.

Below we train a two-layer neural network on the XOR dataset. The network incorporates a hidden layer with 3 hidden units and logistic sigmoid activation functions for all units in the hidden and output layers (see Code Block 4, lines 32-33).

Code Block 4: Implements and trains a two-layer neural network for classification to learn XOR operator and more difficult "ring" problem (Figures 10 & 11; assumes you have run Code Block 1):

```
+ expand source
```

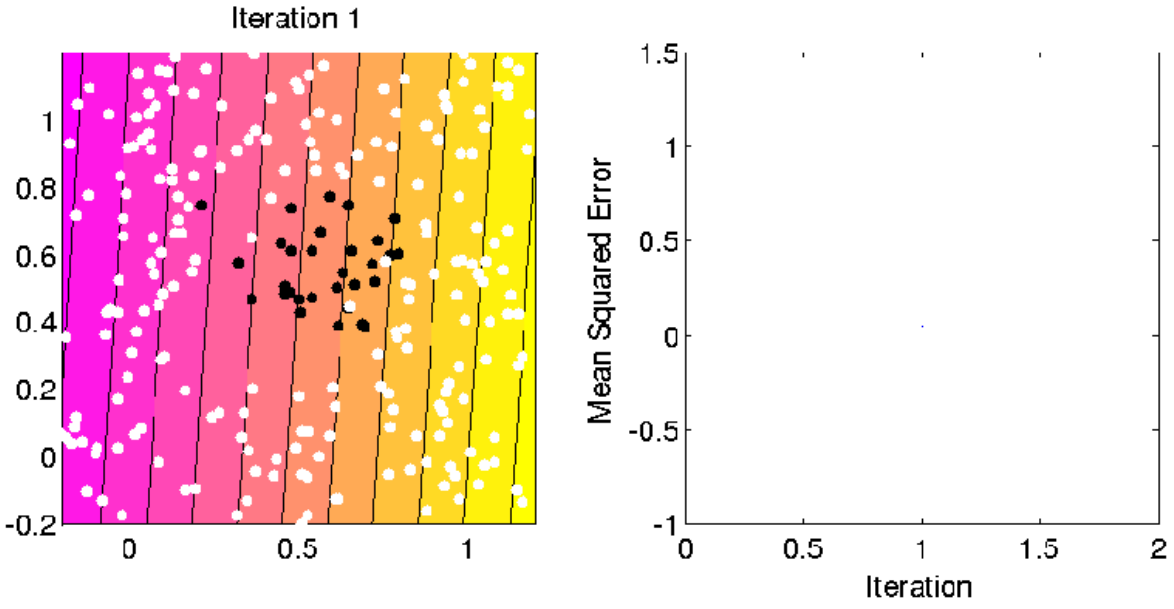


(https://theclevermachine.files.wordpress.com/2014/09/2layer-net-xor.gif)

Figure 10: A multi-layer neural network (perceptron) attempting to learn a noisy XOR mapping. The multi-layer network easily learns this nonlinear problem.

Figure 10 displays the learning process for the 2-layer network. The formatting for Figure 10 is analogous to that for Figures 7-9. The 2-layer network is easily able to learn the XOR operator. We see that by adding a hidden layer between the input and output, the ANN is able to learn the nonlinear categorization criterion!

Figure 11 shows the results for learning a even more difficult nonlinear categorization function: points in and around $(x1, x2) = (0.50.5)$ are categorized as "0", while points in a ring surrounding the "0" datapoints are categorized as a "1" (Figure 11). This example is run by executing Code Block 4 after un-commenting lines 9-11.



(https://theclevermachine.files.wordpress.com/2014/09/2layer-net-ring.gif)

Figure 11: Multilayer neural network learning a nonlinear binary classification task

Figure 11 shows the learning process. Again formatting is analogous to the formatting in Figures 8-10. The 2-layer ANN is able to learn this difficult classification criterion.

Example: Neural Networks for Regression

The previous examples demonstrated how ANNs can be used for classification by using a logistic sigmoid as the output activation function. Here we demonstrate how, by making the output activation function the linear/identity function, the same 2-layer network architecture can be used to implement nonlinear regression.

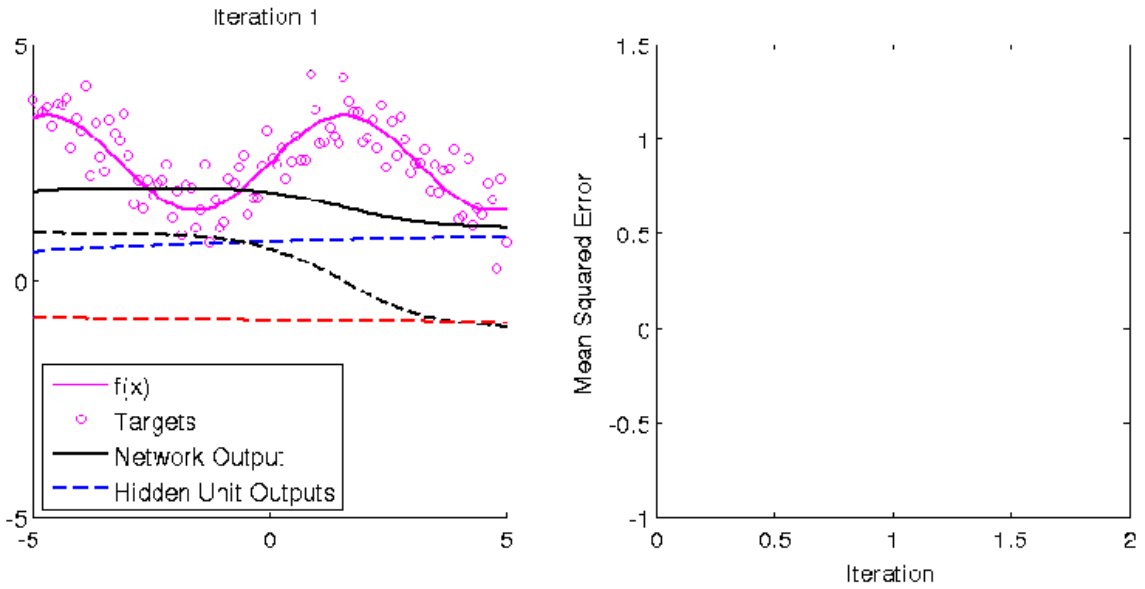
For this example we define a dataset comprised of 1D inputs, $\mathbf{x}$ that range from $(-5, 5)$. We then generate noisy targets $\mathbf{y}$ according to the function:

$$\mathbf{y} = f(\mathbf{x}) + \epsilon$$

where $f(x)$ is a nonlinear data-generating function and ϵ is Normally-distributed noise. We then construct a two-layered network with tanh activation functions used in the hidden layer and linear outputs. For this example we set the number of hidden units to 3 and train the model as we did for categorization using gradient descent / backpropagation. The results of the example are displayed below.

Code Block 5: Trains two-layer network for regression problems (Figures 11 & 12; assumes you have run Code Block 1):

```
+ expand source
```

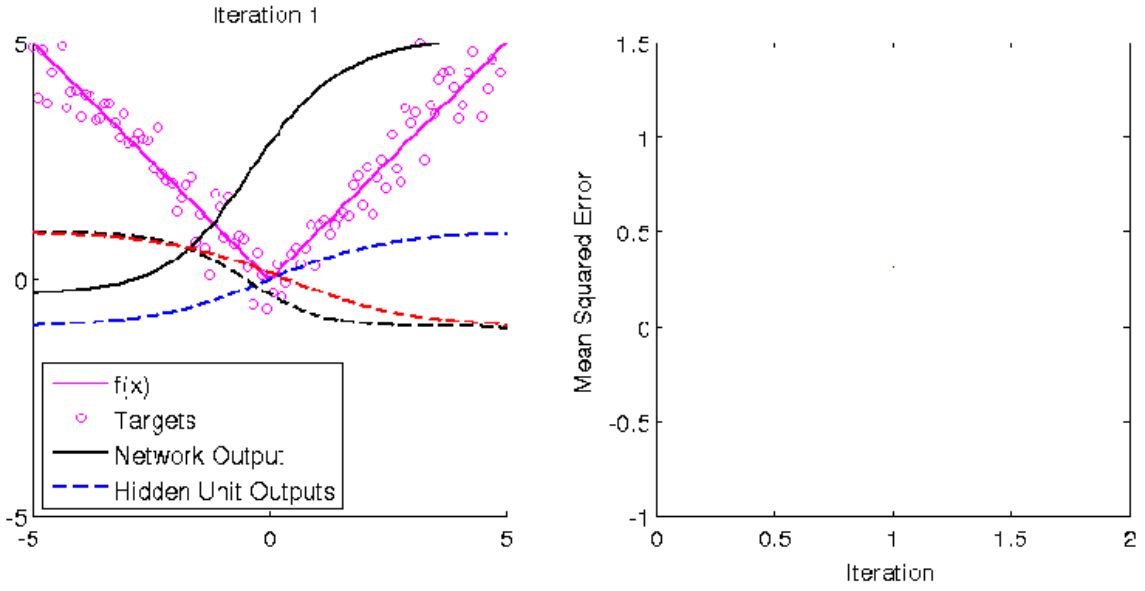


(https://theclevermachine.files.wordpress.com/2014/09/2layer-net-regression-sine.gif)
Figure 12: A two-layered ANN used for regression. The network approximates the function f(x) = sin(x) + 2.5

The training procedure for $f(x) : \sin(x) + 2.5$ is visualized in the left plot of Figure 12. The data-generating function $f(x)$ is plotted as the solid magenta line, and the noisy target values used to train the network are plotted as magenta circles. The output of the network at each training iteration is plotted in solid black while the output of each of the tanh hidden units is plotted in dashed lines. This visualization demonstrates how multiple nonlinear functions can be combined to form the complex

output target function. The mean squared error at each iteration is plotted in the right plot of Figure 12. We see that the error does not follow a simple trajectory during learning, but rather undulates, demonstrating the non-convexity of the error surface.

Figure 13 visualizes the training procedure for trying to learn a different nonlinear function, namely $f(x) : \text{abs}(x)$ (by running Code Block 5, after un-commenting out line 7). Again, we see how the outputs of the hidden units are combined to fit the desired data-generating function. The mean squared error again follows an erratic path during learning.



(https://theclevermachine.files.wordpress.com/2014/09/2layer-net-regression-abs.gif)
Figure 13: A two-layered ANN used for regression. The network approximates the function f(x) = abs(x)

Notice for this example that I added an extra implementation detail known as simulated annealing (line 118) that was absent in the classification examples. This technique decreases the learning rate after every iteration thus making the algorithm take smaller and smaller steps in parameter space. This technique can be useful when the gradient updates begin oscillating between two or more locations in the parameter space. It is also helpful for influencing the algorithm to settle down into a steady state.

Wrapping up

In this post we covered the main ideas behind artificial neural networks including: single- and multi-layer ANNs, activation functions and their derivatives, a high-level description of the backpropagation algorithm, and a number of classification and regression examples. ANNs, particularly mult-layer ANNs, are a robust and powerful class of models that can be used to learn complex, nonlinear functions. However, there are a number of considerations when using neural networks including:

- How many hidden layers should one use?
- How many hidden units in each layer?
- How do these relate to overfitting and generalization?
- Are there better error functions than the squared difference?
- What should the learning rate be?

- What can we do about the complexity of error surface with deep networks?
- Should we use simulated annealing?
- What about other activation functions?

It turns out that there are no easy or definite answers to any of these questions, and there is active research focusing on each topic. This is why using ANNs is often considered as much as a "black art" as it is a quantitative technique.

One primary limitation of ANNs is that they are supervised algorithms, requiring a target value for each input observation in order to train the network. This can be prohibitive for training large networks that may require lots of training data to adequately adjust the parameters. However, there are a set of unsupervised variants of ANNs that can be used to learn an initial condition for the ANN (rather than from randomly-generated initial weights) without the need of target values. This technique of "unsupervised pretraining" has been an important component of many "deep learning" models used in AI and machine learning. In future posts, I look forward to covering two of these unsupervised neural networks: autoencoders and restricted Boltzmann machines.

About dustinstansbury

I recently received my PhD from UC Berkeley where I studied computational neuroscience and machine learning.

View all posts by dustinstansbury »

Posted on September 11, 2014, in Classification, Gradient Descent, Machine Learning, Neural Networks, Neuroscience, Regression and tagged Backpropagation, Classification, Deep Learning, Gradient Descent, Neural Networks, Regression. Bookmark the permalink. 12 Comments.

- **Leave a comment**
- **Trackbacks 2**
- **Comments 10**

bendichter | September 11, 2014 at 1:36 pm

Dustin, this intro to ANNs is outstanding! I've seen recent interest in rectified linear activation functions. Do you have any thoughts on these?

> ***dustinstansbury*** | September 11, 2014 at 8:44 pm
> Thanks Ben, hopefully you find the post helpful.
>
> Yeah, ReLUs are pretty sweet! They exhibit some nice invariance properties that are useful for pattern recognition (for details, see work from Bengio's group on rectifier nets). I often use the softrect/softplus function, an analytic approximation to the ReLU (in MATLAB-ish syntax):
>
> g(z) = 1/k.*log(1 + exp(k*z))
> g'(z) = 1./(1 + exp(-k*z)),
>
> where k is a hyperparameter that controls the smoothness of the rectification around zero.

fifinonz | February 3, 2015 at 7:23 am
Indepth, informative yet simple! Very helpful, Thank you

dhliuvip | March 20, 2016 at 9:26 am
Reblogged this on Robotic Run and commented:
Fantastic Introduction to ANN

Sam | May 1, 2016 at 10:23 pm
This is excellent; good work sir.

Harish Narayanan | June 16, 2016 at 1:48 pm
Thank you for this excellently written post. In literally one sentence: "using linear activations for the output unit activation function (in conjunction with nonlinear activations for the hidden units) allows the network to perform nonlinear regression", you've clarified an idea I've been grasping at but couldn't get to earlier.

George Gillams | December 31, 2016 at 5:56 am
This has greatly simplified this topic for me. Thank you so much

Riva | February 18, 2017 at 7:28 am
Hi Dustin, can you please give me permission to use figure 1 (Figure 1: Diagram of a single-layered artificial neural network.) please for my research paper?

Igor | April 5, 2017 at 5:27 am
Short and simple. Love it!

Umut Can Çakmak | July 25, 2017 at 10:46 pm
Hi, is the "weight update" in the backpropagation algorithm figure "gradient descent"? Like, the first three steps are of backpropagation algorithm but then the weight update is performed with a gradient descent, right. So, if I am using Adam optimizer or something similar, then, the fourth step will not be the one in the figure but Adam's own parameter update formulations?

Thank you for this great introduction.

1. Pingback: **Distilled News | Data Analytics & R**
2. Pingback: **A Gentle Introduction to Artificial Neural Networks – Robotic Run**