Neural Networks

Author: Ryan Kortvelesy
November 28, 2016

Neural networks are a branch of machine learning that draws inspiration from the brain. Human brains have 100 billion neurons, each of which acts as a trigger for small electrical impulses. Neurons, like computers, are binary: they either fire, or they don't. They decide whether or not to fire based on the input electrical signal from their connections, called "synapses". A synapse is strengthened if it is used often, and weakened if it is neglected. As the synapses change, the brain learns. The strength of all of the synapses in the brain dictates an individual's personality, as well as how they think and how they react to stimuli. For example, when we look at the color blue, our optic nerves send a certain electrical signal into our brains. As this signal reaches each layer of neurons, some synapses will cause their neurons to fire, and the electrical signal from those neurons will be passed into the next layer. Our synapses have been trained such that a certain set of neurons will always fire when passed this signal from our eyes, which is equivalent to thinking of the color blue. As different sets of firing neurons are combined, the brain can generate more complex thoughts, such as the unique combination of panic and despair that arises when one procrastinates on a research paper.

Neural networks function similarly. However, they use arrays of weights instead of synapses (larger weights represent stronger synapses), and arrays of variables instead of neurons. The data in each input neuron is multiplied by a weight and added together, then put through an activation function to determine the value of the output neuron.

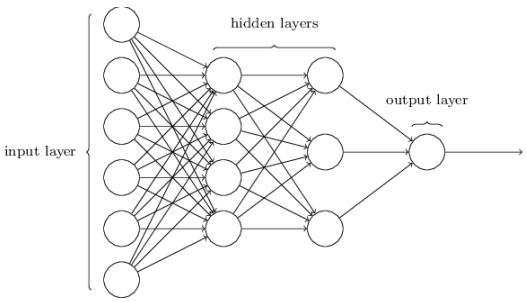
There are, however, a couple differences between neural networks a brains. First, brains use analog signals in synapses and digital signals in neurons. Neural networks, on the other hand, can use analog information in the neurons as well as the weights. Second, the connectivity is different. While neural networks are divided into layers, with each neuron in one layer connected to each one in the next layer, the connections in the brain are a lot messier. Every neuron has about 10,000 connections with its neighbors. This means that neurons aren't necessarily connected with all neurons in the next layer, and there is the possibility for closed loops.

There are two main parts to building a neural network: forward-propagation and back-propagation. Forward-propagation is the act of using a neural network with the weights in their current state to compute an answer. Back-propagation is the act of training a neural network with training data. In this paper, we will explore the effect of layer size, number of layers, activation function, and unit bias on forward-propagation and how to use numerical gradient descent or back-propagation to update the weights in a neural network.

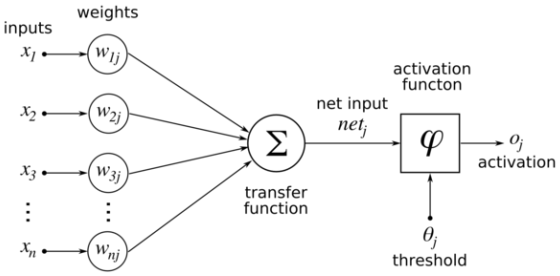
***

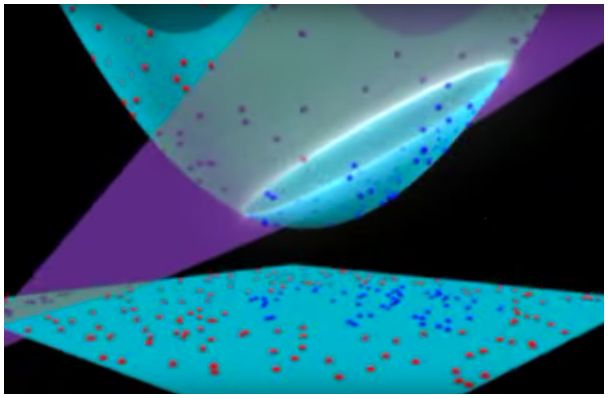
In a neural network, the first layer is the same size as the number of inputs, the hidden layers (middle layers) can be arbitrarily large, and the last layer is the same size as the desired number of outputs. When forward-propagation is performed, each neuron in a given layer is set to the quantity of the sum of an array of weights times all of the values in the previous layer put through an activation function. Sometimes, a step function is used so the output is binary. Other times, functions like sigmoid (goes between 0 and 1) or tanh (goes between -1 and 1) are used because they can spread apart data that is clustered together. The activation function is very important because it applies a nonlinearity to the answer. Without it, the output would simply be a linear combination of the

$$\sum_{j=1}^{N} tanh(\alpha_j x_j + \beta)$$



inputs, regardless of the number or size of the hidden layers. Consequently, applying a linear activation function is the same as having no activation function at all. In that case, the neural network would simply act as regression. If you picked a list of points (x,y) and trained it in a neural network with no hidden layers with an input of [1 x x^2] and an output of [y], then the weights would be the same as the coefficients in the quadratic regression function. However, with a nonlinear activation function, it is possible to map layers to a higher dimensional space so that they are separable by a hyperplane. If there are n neurons in layer A and m neurons in layer b, then the function f that maps A to B can be written as f: A → B, where A ∈ $\mathbb{R}^n$ and B ∈ $\mathbb{R}^m$. So, even if the data isn't separable in $\mathbb{R}^n$, it may still be separable by a hyperplane of degree m-1 after the data is mapped to dimension $\mathbb{R}^m$. If there are less than or equal to m-1 data points, then the data is guaranteed to be linearly separable. The universal approximation theorem states that "a feed-forward network containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$" [1] (see appendix for proof). In other words, we can use a neural network to approximate bounding regions in high-dimensional space. Because the number of neurons in a given layer corresponds with the dimension of this bounding manifold, it is easier to accurately separate data if there are more layers.

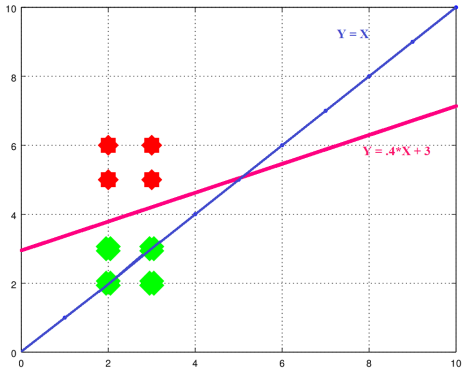




***

Another way to improve the separability of data in neural networks is to add more layers. As the data propagates forward through the layers, it progressively becomes more separated, until ideally the data is clustered in tight, distant groups that make it easy for the last activation function to distinguish.

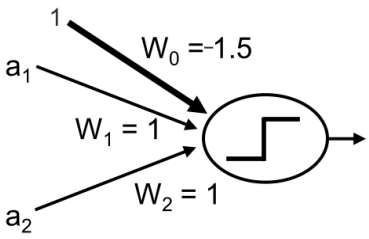
In each layer, we add an extra term called "unit bias", which is a constant offset. To understand this, it is easiest to consider a network that has only one input. If we don't have any unit bias, then we can only separate the data over the line y = mx, where m is a weight that we solve for. That is not very helpful for data that is only separable with a line that does not go through the origin. If we add a 1 to the list of inputs (and to each hidden layer), then we can separate the data over the line y = mx+b, where b is the weight applied to the unit bias.



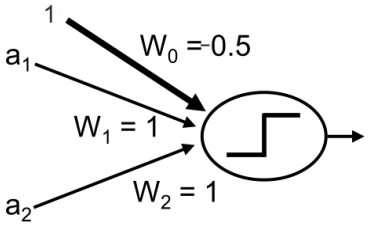
But what about data that isn't linearly separable? Is it possible to train a classifier for that without mapping to higher dimensions? Sure it is. Let us consider a neural network of perceptrons (binary

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

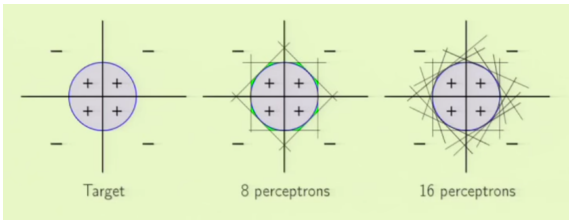
classifiers, using a step function) with constant layer size, so they don't map to a higher dimension. For simplicity, let us say the layer size is 3 (two inputs, one unit bias). The two inputs to each layer are binary linear separators–lines that assign a 1 to any point on one side, and 0 to any point on the other side. The perceptron can perform AND or OR operations to these binary inputs. If the weight on the unit bias is -1.5, it functions as an AND, because both inputs have to be 1 for the output to be greater than zero (which becomes 1 after going through the step function). If the weight is -0.5, the perceptron functions as OR, because if either input is 1 the output is greater than zero. If all of the weights are negated, then the output is negated. You can't perform an XOR with one perceptron, but you can create it with an OR followed by a NAND. In fact, as more and more perceptrons are added, it becomes possible to divide more and more complex regions[2]. Combining n layers of perceptrons can generate an n-sided polygon or a linear spline with n-1 points.



AND



OR



We have demonstrated that neural networks can be effective with many layers or with large layer depth. However, they are most powerful when the two are combined. This is called deep learning. When designing a neural network, one should keep in mind the data being classified. Adding superfluous neurons and layers will increase a program's runtime. In addition, networks that are too deep run the risk of overfitting the data. That is, the network might fit to the outliers as well as the bulk of the data. This is analogous to choosing a degree too high when fitting a polynomial using regression.

***

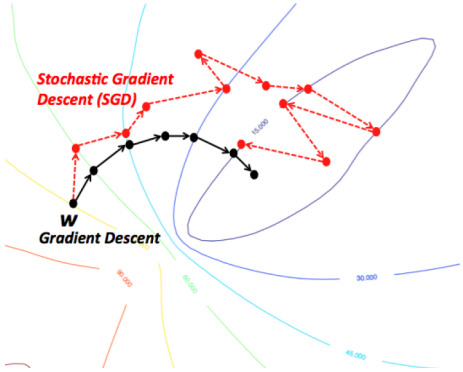
So far we have covered how neural networks classify data, but not how they learn. Neural networks are a form of supervised machine learning, meaning they learn given sample inputs and the correct outputs. The network "learns" by updating the array of weights between each layer. This can be done with back-propagation or numerical gradient descent, and can be done stochastically or in a batch.

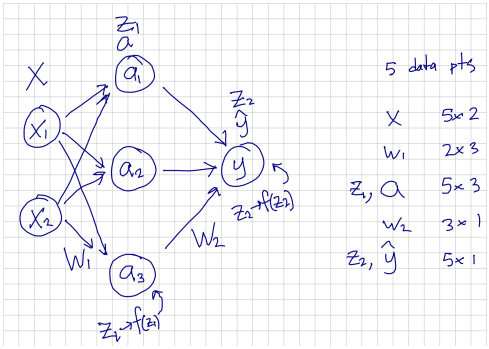
Numerical gradient descent is the most straightforward way to train a neural network, because it doesn't require solving for any derivatives analytically. Gradient descent is not specific to neural networks–it is a method that computers use to find local minima. If you can compute the gradient of a function at a given point, then you can follow that gradient in the negative direction until you reach a local minimum. In this case, the function we need to optimize is called a cost function. There are many types of cost functions, but the most common is simply the square of the euclidean norm (the sum of the squares of the residuals). In numerical gradient descent, the gradient of this function is computed numerically by finding the slope of the line between W+ε and W-ε, where W is the values of the weights (it has to be done

[2] http://work.caltech.edu/slides/slides10.pdf

for each weight separately), and ε is a sufficiently small value. This way, one can obtain a gradient for each weight, and follow the opposite direction of that gradient to an optimum.

Conversely, back-propagation requires an analytical solution. By taking the derivative of the cost function, you can solve for the gradient of the weights, and follow that to an optimum using gradient descent. In stochastic back-propagation, only one sample is considered at a time, and the program loops over the entire set of training data. Batch gradient descent, on the other hand, takes multiple samples into account at the same time. It can take the average of all of the gradients, and descend toward the optimum just once.





Now, we will derive the formula for batch gradient descent. Let:

x = array of inputs ŷ = output
W_1 = weights between input and hidden layer W_2 = weights between hidden and output layer
f = tanh(x), the activation function f' = 1-f^2
$z_1 = x*W_1$ $a = f(z_1)$
$z_2 = a*W_2$ $\hat{y} = f(z_2)$

The cost function is given by:

$$J = \frac{1}{2}(y - f(f(xW_1)W_2)^2$$

Applying the chain rule, the gradients of the weights $\partial J/\partial W_1$ and $\partial J/\partial W_2$ are:

$$\frac{\partial J}{\partial W_2} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial W_2} \qquad \frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial a}\frac{\partial a}{\partial z_1}\frac{\partial z_1}{\partial W_1}$$

Given that $\partial J/\partial y = -(y-f(f(xW_1)W_2))$, $\partial y/\partial z_2 = f'(z_2)$, and $\partial z_2/\partial W_2 = a^T$:

$$\frac{\partial J}{\partial W_2} = a^T(f(f(xW_1)W_2) - y).*f'(z_2)$$

Given that $\partial J/\partial y = -(y-f(f(xW_1)W_2))$, $\partial y/\partial z_2 = f'(z_2)$, $\partial z_2/\partial a = W_2^T$, $\partial a/\partial z_1 = f'(z_1)$, and $\partial z_1/\partial W_1 = x^T$:

$$\frac{\partial J}{\partial W_2} = x^T(((f(f(xW_1)W_2) - y).*f'(z_2))W^T.*f'(z_1))$$

***

Neural networks are just a small branch of machine learning, but they have many uses. They are most often used for classification problems, but they have other applications as well. One interesting example of neural networks being used for classification is in handwriting recognition. The neural network can take in the pixels of an image of a handwritten letter as a long one-dimensional array and learn to classify which letter it is. Neural networks can also be used in robotics to tune control loops or even train robots to react to stimuli. Tuning PID control is usually a long guess-and-check process, but with a neural network it is possible to do it automatically[3]. If you give the neural network a critically damped response as training data and run several trials, the robot will begin to learn how to get from point A to point B the fastest without overshooting. The activation function even provides the added guarantee that the output will be within the bounds of the power levels (whereas with regular PID, it is possible to get a power output greater than the maximum).

I have created my own neural network in MATLAB to demonstrate the basic functionality. The functions I wrote can predict an output given inputs, provide visualization of the network training itself using batch gradient descent, automatically optimize the network, and allow a user to create points with different values on an x-y plot, then train a network to fit those values.

Appendix

optimizenn.m

```
% Automatically trains the network given a set of input data until
% it is optimized
function output = optimizenn(data,realval)
eps = 0.01;
lastcost = eps+100;
while lastcost > eps
   [output, cost] = nn(data,realval);
   if abs((cost-lastcost)/cost) < 10^-8
      clear global;
   end
   lastcost = cost;
   disp(cost);
end
```

[3] http://www.pdx.edu/sites/www.pdx.edu.sysc/files/media_assets/SySc576_FrankLewisNNsControl.pdf

nn.m

```
function [output,cost] = nn(data,realval)
% data is a vector of size m x n where m in the number or tests to run, and n
% is the number of inputs into the neural network. realval is a vector of
% the training data for the neural network. If realval is not provided, the
% program outputs the predictions given the previous training data.

% f = the activation function
% W1 = the weights to be multiplied by the data to get Z1
% W2 = the weights to be multiplied by the hidden layer to get Z2
% Z1 = the input for the activation function to yield the hidden layer vals
% Z2 = the input for the activation function to yield the answer
% data = the original input data to be processed
% a = the data in the hidden layer to be processed into a final answer
% y = the final answer

global W1;
global W2;
global Z1;
global Z2;
global a;
global f;
learningspeed = 0.1;
hiddenlayersize = 12;
f = @(z) tanh(z);
df = @(z) (1 - f(z).^2);
% Choose an activation function wisely. First, make sure that the minimum
% and maximum output value of the function are correct. Next, decide how
% you want to manipulate the output. Do you want a discrete 1 or 0 as an
% output? Then choose a step function. Do you want to be able to
% differentiate between data that is clustered in one area? Choose tanh. Do
% you just want regression without an activation function? Use linear.

% Creates weight matrices if the program has not yet been trained
% Makes sure input matrices have the correct dimensions
if isempty(W1)
   W1 = (rand(length(data(1,:))+1,hiddenlayersize)-0.5);
   W2 = (rand(hiddenlayersize+1,1)-0.5);
else
   if length(data(1,:)) ~= length(W1(:,1))-1
      error('There must be %d inputs, but the provided data matrix has %d.\n',length(W1(:,
1))-1,length(data(1,:)));
   end
end
if nargin == 2
   if length(data(:,1)) ~= length(realval)
      error('The number of data samples (%d) must equal the number of training data outputs
(%d)\n',length(data),length(realval));
```

```
    end
    [m,n] = size(realval);
    if ~(m == 1 || n == 1)
        error('The training data must be either a row or column vector. It currently has size %dx%d
\n',m,n);
    end
    if ~iscolumn(realval)
        realval = realval';
    end
end

data = [ones(size(data(:,1))) data]; % add bias unit to inputs

% If no training data is given, the program simply runs the neural network
if nargin == 1
    output = evaluate(data);
    return
end

cost = norm(evaluate(data)-realval);

% Use backpropagation to find the gradient
dJdW1 = data' * ((((evaluate(data) - realval) .* df(Z2)) * W2(2:end,1)') .* df(Z1));
dJdW2 = a' * ((evaluate(data) - realval) .* df(Z2));

% Use gradient descent to approach an optimum
W1 = W1 - learningspeed * dJdW1;
W2 = W2 - learningspeed * dJdW2;

% Run neural network with updated weights
output = evaluate(data);

return

function y = evaluate(data)
global W1;
global W2;
global Z1;
global Z2;
global a;
global f;
Z1 = data * W1;
a = ones(length(Z1(:,1)),length(Z1(1,:))+1); % add bias to hidden layer
a(:,2:end) = f(Z1); % the outputs from layer 1 don't affect bias
Z2 = a*W2;
y = f(Z2);
return
```

nntesting.m

```
% Allows the user to input training data, and shows a visualization
% as the network trains itself
close all
clear
clc

iterations = 1000;
viewsize = 15;
data = 3*[1 2; 0.2 0.2; 2 1; -1 -2; -0.2 -0.2; -2 -1; -1 2; -0.2 0.2; -2 1; 1 -2; 0.2 -0.2; 2 -1];
realval = [1;1;1;1;1;1;-1;-1;-1;-1;-1;-1];

figure
hold on
plot(data(realval==1,1),data(realval==1,2),'*g');
plot(data(realval==-1,1),data(realval==-1,2),'*k');
legend('1','-1');
title('Training Data');
axis([-floor((viewsize-1)/2),ceil((viewsize-1)/2),-floor((viewsize-1)/2),ceil((viewsize-1)/2)]);
hold off
fig = figure;
[~,lastcost] = nn(data,realval);
for n = 1:iterations
    [output, cost] = nn(data,realval);
    if cost > 0.1 && abs((cost-lastcost)/cost) < 1*10^-6
        clear global;
    end
    lastcost = cost;

    classification = zeros(viewsize,viewsize);
    for x1 = -floor((viewsize-1)/2):ceil((viewsize-1)/2)
        for x2 = -floor((viewsize-1)/2):ceil((viewsize-1)/2)
            classification(x2+(1+floor((viewsize-1)/2)),x1+(1+floor((viewsize-1)/2))) = nn([x1,x2]);
        end
    end
    classification = imresize(classification,viewsize*4);
    set(groot,'CurrentFigure',fig);
    imshow(flipud(classification));
    colormap default;
    text(viewsize*1.6,viewsize*2,sprintf('Cost: %0.4f',cost),'Color','red','FontSize',14)
    pause(0.1);
end
```

Universal Approximation Theorem Proof [4]

Cybenko defines $\Pi_{y,\theta} = \{x \mid y^\intercal x + \theta = 0\}$ and $H_{y,\theta} = \{x \mid y^\intercal x + \theta > 0\}$

Let:

$$F(h) = \int_{I_n} h(y^\intercal x) \mathrm{d}\mu(x)$$

Let h the indicator function of $[\theta, \infty)$

So we have

$$F(h) = \int_{I_n} h(y^\intercal x) \mathrm{d}\mu(x) = \mu(\Pi_{y,-\theta}) + \mu(H_{y,-\theta})$$

$$\begin{aligned}
0 &= \lim_{\lambda \to +\infty} \int_{I_M} \sigma_\lambda(x) \mathrm{d}\mu(x) \qquad \text{because } \sigma \text{ discriminatory} \\
&= \int_{I_M} \lim_{\lambda \to +\infty} \sigma_\lambda(x) \mathrm{d}\mu(x) \\
&= \int_{I_M} \left(1_{y^\intercal x + \theta > 0}(x) + \sigma(\varphi) \delta_{y^\intercal x + \theta = 0}(x) \right) \mathrm{d}\mu(x) \\
&= \sigma(\varphi)\mu(\Pi_{y,\theta}) + \mu(H_{y,\theta}), \forall \varphi, \theta, y
\end{aligned}$$

This is true for all φ and σ is a sigmoid so we can take $\varphi \to +\infty$ i.e. $\sigma(\varphi) \to 1$. Which is true for all y and θ and therefore in particular for $y, -\theta$.

Picture Sources

1. https://duo.com/blog/applications-of-deep-learning-the-good-the-bad-and-the-opinion

2. http://www.kdnuggets.com/2016/10/artificial-intelligence-deep-learning-neural-networks-explained.html

3. https://www.youtube.com/watch?v=3liCbRZPrZA

4. http://stats.stackexchange.com/questions/155132/the-bias-neuron-in-a-neural-network-is-my-understanding-right

5. http://slidewiki.org/deck/1303_perceptron#tree-1303-slide-18928-7-view

6. http://work.caltech.edu/slides/slides10.pdf

7. http://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php

---

[4] http://math.stackexchange.com/questions/1453353/proof-of-cybenkos-lemma-1-in-his-approximation-by-superpositions-of-a-sigmoida