Multi-Layer Neural Networks with Sigmoid Function— Deep Learning for Rookies (2)

Nahua Kang Follow

Jun 27, 2017 · 15 min read

Chapter 1: Introducing Deep Learning and Neural Networks

Chapter 2: Multi-Layer Neural Networks with Sigmoid Function

Follow me on Twitter to learn more about life in a Deep Learning Startup.

• • •

Howdy y'all! Welcome back to my second post of the series **Deep Learning for Rookies (DLFR)**, by yours truly, a rookie ;) Feel free to refer back to my first post here or my blog if you find it hard to follow. Or highlight on this page with notes or leave a comment below! Your feedback will be highly appreciated, too.

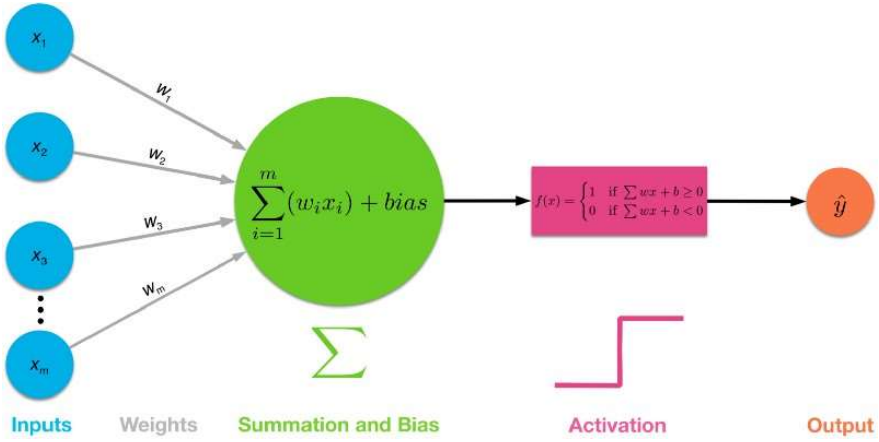
We will go deeper into neural networks this time and the post will be slightly more technical than last time. But no worries, I will make it as easy and intuitive as possible for you to learn the basics without

CS/Math background. You'll be able to brag about your understanding soon ;)

Review from DLFR 1

Last time, we introduced the field of Deep Learning and examined a simple a neural network—perceptron……or a dinosaur……ok, seriously, a single-layer perceptron. We also examined how a perceptron network process the input data we feed in and returns an output.

Key concepts: input data, weights, summation and adding bias, activation function (specifically step function), and then output. Bored yet? No worries :) I promise there will be……more jargons coming up! But you'll get used to them soon. I promise.



Graph 1: Procedures of a Single-layer Perceptron Network

Back in the 1950s and 1960s, people had no effective learning algorithm for a single-layer perceptron to learn and identify non-linear patterns (remember the XOR gate problem?). And the public lost interest in perceptron. After all, most problems in the real world are non-linear, and as individual humans, you and I are pretty darn good at the decision-making of linear or binary problems like ***should I study Deep Learning or not*** without needing a perceptron. Ok, "good" is a tricky word here as our brains are really not so rational. But I'll leave that to behavioral economists and psychologists.

Breakthrough: Multi-Layer Perceptron

Fast forward almost two decades to 1986, Geoffrey Hinton, David Rumelhart, and Ronald Williams published a paper "*Learning representations by back-propagating errors*", which introduced:

1. **Backpropagation**, a procedure to ***repeatedly adjust the weights*** so as to minimize the difference between actual output and desired output
2. **Hidden Layers**, which are ***neuron nodes stacked in between inputs and outputs***, allowing neural networks to learn more complicated features (such as XOR logic)

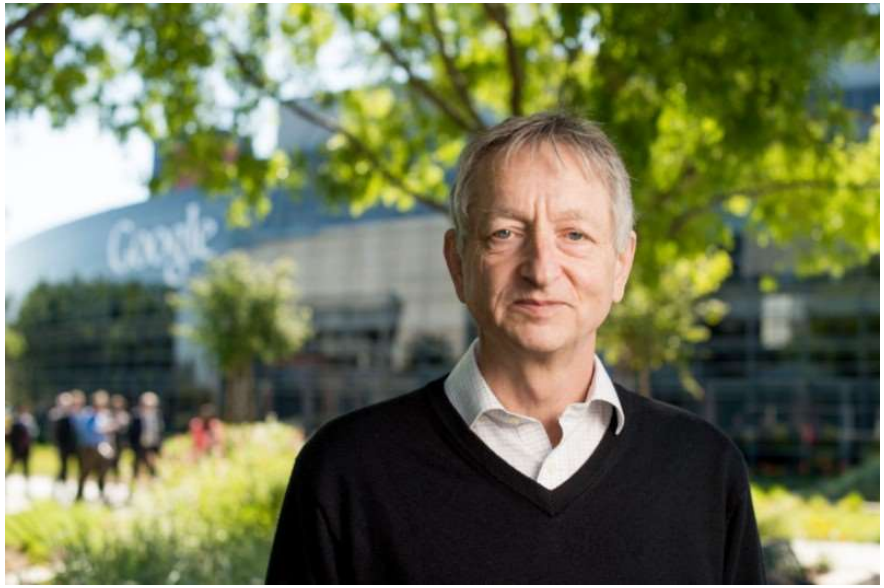
If you are completely new to DL, you should remember Geoffrey Hinton, who plays a pivotal role in the progress of DL. So that was some major news: We just thought for 20 years that there's no future

for neural networks to tackle real-world problems. Now we see the light from a beacon ashore! Let's take a look at these 2 new introductions.

Hmm, the first one, **backpropagation**, is mentioned in the last post. Remember that we iterated the importance of designing a neural network so that the network can learn from the difference between the ***desired output*** (what the fact is) and ***actual output*** (what the network returns) and then send a signal back to the weights and ask the weights to adjust themselves? This will make the network's output closer to the desired output next time we run it.

What about the second one, hidden layers? What is a hidden layer? A hidden layer transforms a single-layer perceptron into a multi-layer perceptron! Here's the plan, for technical reasons, ***I will focus on hidden layers in this post, and then discuss backpropagation in the next post***.

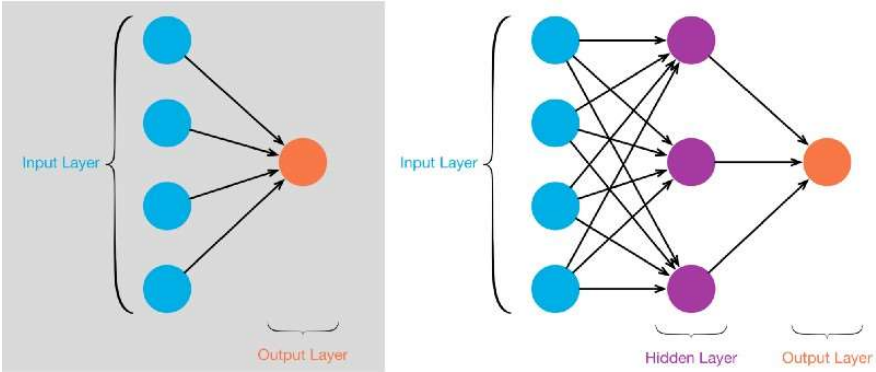
Just a random **fun fact** here: I suspect that Geoffrey Hinton holds a record for having been the oldest intern at Google :D Check out this article from New York Times and search "Hinton". Anyways, if you're already familiar with machine learning, I'm sure his course on Coursera will be a good fit.



Geoffrey Hinton, one of the most important figures in Deep Learning. Source: thestar.com

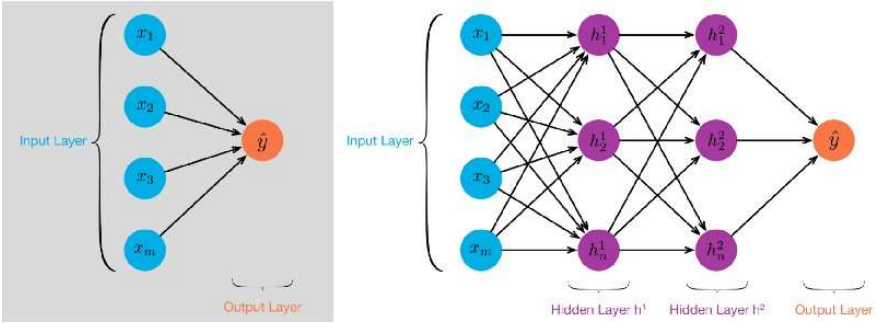
Neural Networks with Hidden Layers

Hidden layers of a neural network is literally just adding more neurons in between the input and output layers.



Graph 2: Left: Single-Layer Perceptron; Right: Perceptron with Hidden Layer

Data in the input layer is labeled as ***x*** with subscripts ***1, 2, 3, …, m***. Neurons in the hidden layer are labeled as ***h*** with subscripts ***1, 2, 3, …, n***. Note for hidden layer **it's *n* and not *m***, since the number of hidden layer neurons might differ from the number in input data. And as you see in the graph below, the hidden layer neurons are also labeled with superscript ***1***. This is so that when you have several hidden layers, you can identify which hidden layer it is: *first hidden layer has superscript 1, second hidden layer has superscript 2, and so on, like in* ***Graph 3***. Output is labeled as ***y*** with a ***hat***.



Graph 3: We label input layer as x with subscripts 1, 2, ..., m; hidden layer as h with subscripts 1, 2, ..., n; output layer with a hat

To make life easier, we will use some jargons to clear things out a bit. I know, jargons can be annoying but you will get used to them :) First, if we have ***m*** input data (***x1, x2, …, xm***), we call this ***m features***. A feature is just one variable we consider as having an influence to a specific outcome. Like our example on the outcome of ***if you decide to learn DL or not,*** we have 3 features: 1. Will you earn more money after learning DL, 2. Is the math/programming hard, 3. do you need a GPU to start with.

Secondly, when we multiply each of the m features with a weight (***w1, w2, …, wm***) and sum them all together, this is a **dot product**:

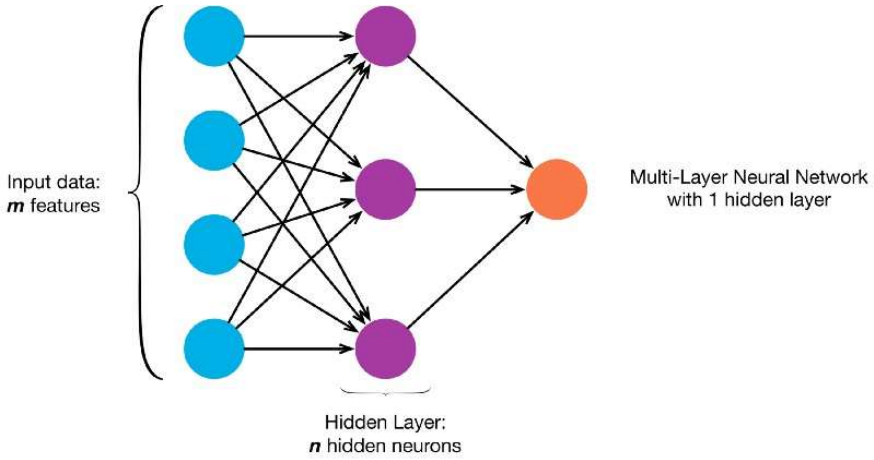
$$W \cdot X = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = \sum_{i=1}^{m} w_i x_i$$

Definition of a Dot Product

So here are the takeaways for now:

1. With ***m features*** in input ***X,*** you need ***m*** weights to perform a dot product
2. With ***n*** hidden neurons in the hidden layer, you need ***n*** sets of weights (***W1, W2, … Wn***) for performing dot products
3. With 1 hidden layer, you perform ***n*** dot products to get the hidden output ***h:*** (***h1, h2, …, hn***)
4. Then it's just like a single-layer perceptron, we use hidden output ***h:*** (***h1, h2, …, hn***) as input data that has **n features,** perform dot product with 1 set of ***n*** weights (***w1, w2, …, wn***) to get your final output **y_hat**.

The procedure of how input values are *forward propagated* into the hidden layer, and then from hidden layer to the output is the same as in **Graph 1**. Below I provide a description of how this is done, using the following neural network in **Graph 4**.



Graph 4: Example

Perform dot product for input data set $X : x_1, x_2, ..., x_m$ with weight set W^1: $w_1^1, w_2^1, ..., w_m^1$, (note the weights have superscript 1 since they are weights to hidden layer h^1's first hidden neuron h_1^1), with the dot product summation, add bias so we have a result z:

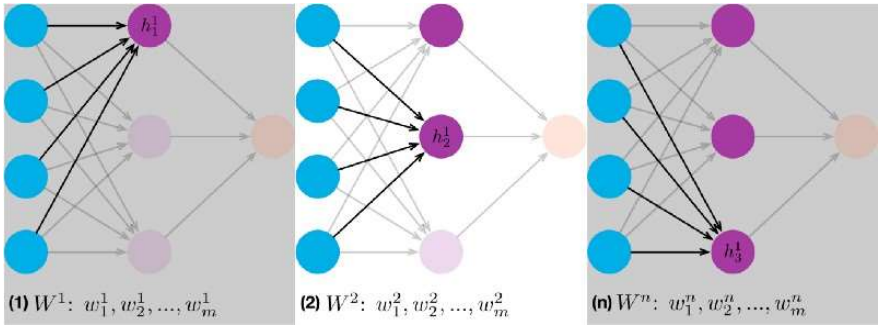
$z = \sum_{i=1}^{m} w_i x_i + bias$

Feed z into an activation function $f(z)$ so that we get an output for the hidden layer h^1's first neuron h_1^1:

$h_1^1 = f(z)$, (*note that* $f()$ *needs not be step function*)

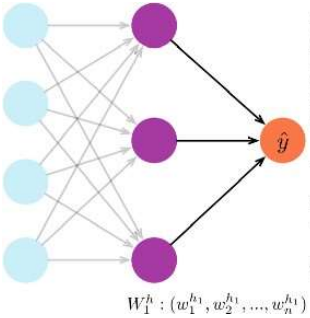
Repeat the same with input data and weights to the second hidden neuron, which consists of W^2: $w_1^2, w_2^2, ..., w_m^2$, and then we will get the output for the second hidden neuron in hidden layer h^1: h_2^1

Repeat again and again until the last weight set W^n using input data with the nth set W^n: $w_1^n, w_2^n, ..., w_m^n$ for the third hidden neuron in hidden layer h^1 to get h_n^1, so in total we have n hidden outputs: $h^1 : (h_1^1, h_2^1, ..., h_n^1)$



Graph 5: Procedure to Hidden Layer Outputs

Now the hidden layer outputs are calculated, we use them as inputs to calculate the final output.



For the final output, perform a dot product for hidden layer outputs (which we now consider as inputs) and hidden layer weights W^h. Remember that the inputs are hidden layer outputs $h^1 : (h_1^1, h_2^1, h_3^1, ..., h_n^1)$. Since we have only 1 output, our set of weights consist of a W_1^h, which is $W_1^h : (w_1^{h_1}, w_2^{h_1}, ..., w_n^{h_1})$ with n weights since we have n hidden layer inputs. Then add the sum of dot product with bias to obtain a result z:

$$z = \sum_{i=1}^{n} w_i^{h_1} h_i^1 + bias$$

Feed z into an activation function $f(z)$ so that we get an output for the output layer:

$$\hat{y} = f(z)$$

Graph 6: Procedure to Final Output

Yay! Now you understand fully how a perceptron with multiple layers work :) It is just like a single-layer perceptron, except that you have many many more weights in the process. When you are training neural networks on larger datasets with many many more features (like word2vec in Natural Language Processing), this process will eat up a

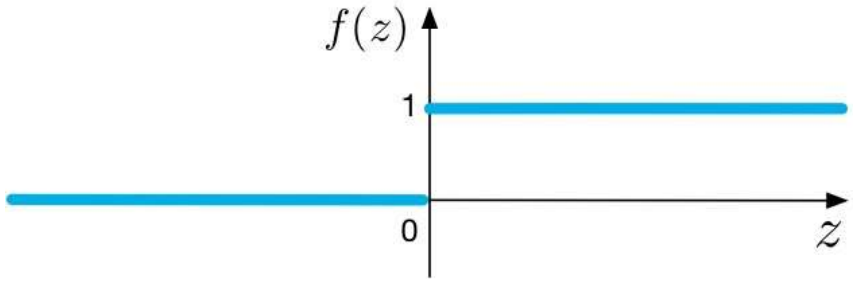
lot of memory in your computer. This was one reason why Deep Learning didn't take off until the past few years, when we began producing much better hardware that could handle the memory-consuming deep neural networks.

Sigmoid Neurons: An Introduction

So now we have a more sophisticatedly structured neural network with hidden layers. But we haven't solved the activation problem with the step function.

In the last post, we talked about the limitations of the linearity of step function. One thing to remember is: **If the activation function is linear, then you can stack as many hidden layers in the neural network as you wish, and the final output is still a linear combination of the original input data**. Please **make sure you read this link** for an explanation if the concept is difficult to follow. This linearity means that it cannot really grasp the complexity of non-linear problems like XOR logic or patterns separated by curves or circles.

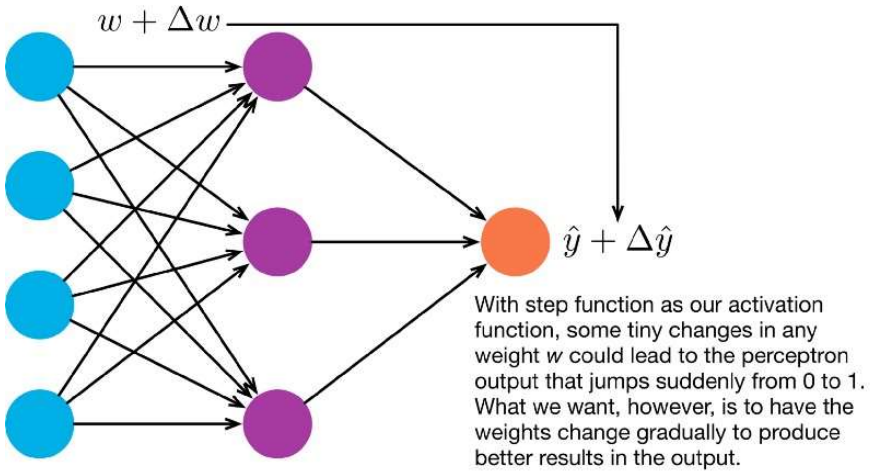
Meanwhile, step function also has no useful derivative (its derivative is 0 everywhere or undefined at the 0 point on x-axis). It doesn't work for **backpropagation**, which we will definitely talk about in the next post!



Graph 7: Step Function

Well, here's another problem: Perceptron with step function isn't very "stable" as a "relationship candidate" for neural networks. Think about it: this girl (or boy) has got some serious bipolar issues! One day (for $z < 0$), (s)he's all "quiet" and "down", giving you zero response. Then another day (for $z \geq 0$), (s)he's suddenly "talkative" and "lively", speaking to you nonstop. Heck of a drastic change! There's no transition for her/his mood, and you don't know when it's going down or up. Yeah…that's step function.

So basically, a small change in any weight in the input layer of our perceptron network could possibly lead to one neuron to suddenly flip from 0 to 1, which could again affect the hidden layer's behavior, and then affect the final outcome. Like we said already, we want a learning algorithm that could improve our neural network by gradually changing the weights, not by flat-no-response or sudden jumps. If we can't use step function to gradually change the weights, then it shouldn't be the choice.

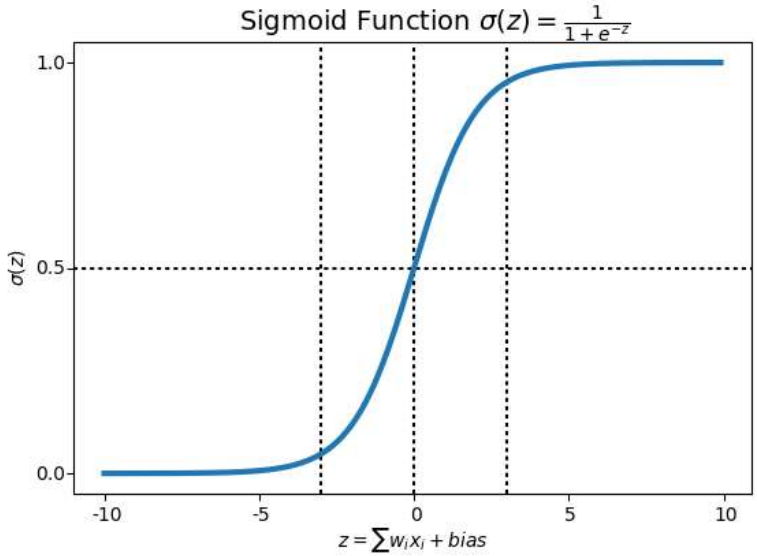


Graph 8: We Want Gradual Change in Weights to Gradually Change Outputs

Say goodbye to perceptron with step function now. We are finding a new partner for our neural network, the **sigmoid neuron**, which comes with sigmoid function (duh). But no worries: The only thing that will change is the activation function, and everything else we've learned so far about neural networks still works for this new type of neuron!

$$z = \sum_{i=1}^{m} w_i x_i + bias$$
$$\text{Sigmoid Function is: } \sigma(z) = \frac{1}{1+e^{-z}}$$

Sigmoid Function



Graph 9: Sigmoid Function using Matplotlib

If the function looks very abstract or strange to you, don't worry too much about the details like Euler's number ***e*** or how someone came up with this crazy function in the first place. For those who aren't math-savvy, the only important thing about sigmoid function in **Graph 9** is first, its curve, and second, its derivative. Here are some more details:

1. Sigmoid function produces similar results to step function in that the output is between 0 and 1. The curve crosses 0.5 at $z=0$, which we can set up rules for the activation function, such as: If the sigmoid neuron's output is larger than or equal to 0.5, it outputs 1; if the output is smaller than 0.5, it outputs 0.

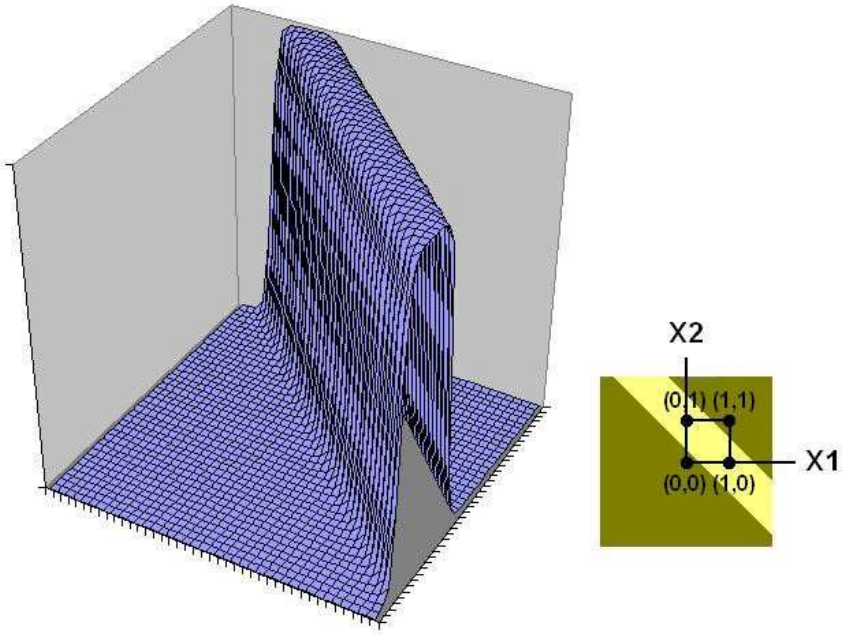
2. Sigmoid function does not have a jerk on its curve. It is smooth and it has a very nice and simple derivative of ***σ(z) * (1-σ(z)),*** which is differentiable everywhere on the curve. The calculus derivation of the derivative can be found on Stack Overflow here if you want to see it. But you don't have to know how to derive it. No stress here.

3. If ***z*** is very negative, then the output is approximately 0; if ***z*** is very positive, the output is approximately 1; but around ***z=0*** where ***z*** is neither too large or too small (in between the two outer vertical dotted grid lines in **Graph 9**), we have relatively more deviation as ***z*** changes.

Now that seems like a dating material for our neural network :) Sigmoid function, unlike step function, introduces non-linearity into our neural network model. Non-linear just means that the output we get from the neuron, which is the dot product of some inputs ***x (x1, x2, …, xm)*** and weights ***w (w1, w2, …,wm)*** plus bias and then put into a sigmoid function, cannot be represented by a linear combination of the input ***x (x1, x2, …,xm)***.

This non-linear activation function, when used by each neuron in a multi-layer neural network, produces a new "representation" of the original data, and ultimately allows for non-linear decision boundary, such as XOR. So in the case of XOR, if we add two sigmoid neurons in a hidden layer, we could, in another space, reshape the original 2D graph into something like the 3D image in the left side of **Graph 10** below. This ridge thus allows for classification of the XOR gate and it represents the light yellowish region of the 2D XOR gate in the right side of **Graph 10**. So if our output value is on the higher area of the

ridge, then it should be a true or 1 (like the weather is cold but not hot, or the weather is hot but not cold); if our output value is on the lower flat area on the two corners, then it's false or 0 since it's not right to say the weather is both hot and cold or neither hot or cold (ok, I guess the weather could be neither hot or cold…you get what I mean though… right?).



Graph 10. Representation of Neural Networks with Hidden Layers to Classify XOR Gate. Source: http://colinfahey.com/

I know these talks on non-linearity can be confusing, so please read more about linearity & non-linearity here (intuitive post with animation from a great blog by Christopher Olah), here (by Vivek

Yadav with ReLU activation function), and here (by Sebastian Raschka). Hopefully you have a sense of why non-linear activation function is important, and if not, take it easy and allow some time to digest it.

Problem solved……for now ;) We will see some different types of activation function in the near future because sigmoid function has its own issues, too! Some popular ones include ***tanh*** and ***ReLU***. That, however, is for another post.

Multi-Layer Neural Networks: An Intuitive Approach

Alright. So we've introduced hidden layers in a neural network and replaced perceptron with sigmoid neurons. We also introduced the idea that non-linear activation function allows for classifying non-linear decision boundaries or patterns in our data. You can memorize these takeaways since they're facts, but I encourage you to google a bit on the internet and see if you can understand the concept better (it is natural that we take some time to understand these concepts).

Now, we've never talked about one very important point: Why on earth do we want hidden layers in neural networks in the first place? How do hidden layers magically help us to tackle complicated problems that single-layer neurons cannot do?

From the XOR example above, you've seen that adding two hidden neurons in 1 hidden layer could reshape our problem into a different

space, which magically created a way for us to classify XOR with a ridge. So hidden layers somehow twist the problem in a way that makes it easy for the neural network to classify the problem or pattern. Now we'll use a classic textbook example: Recognition of hand-written digits, to help you intuitively understand what hidden layers do.

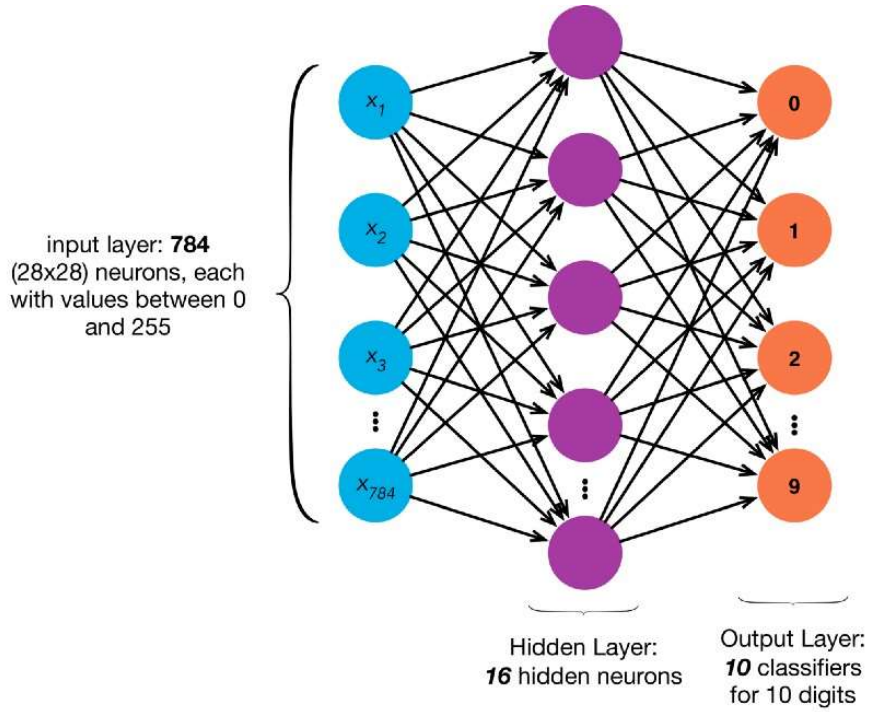
Graph 11. MNIST dataset of Hand-written Digits. Source: http://neuralnetworksanddeeplearning.com/

The digits in **Graph 11** belong to a dataset called MNIST. It contains 70,000 examples of digits written by human hands. Each of these digits is a picture of 28x28 pixels. So in total each image of a digit has 28*28=784 pixels. Each pixel takes a value beween 0 and 255 (RGB color code). 0 means the color is white and 255 means the color black.

Graph 12. MNIST digit 5, which consist of 28x28 pixel values between 0 and 255. Source: http://neuralnetworksanddeeplearning.com/

Now, the computer can't really "see" a digit like we humans do, but if we dissect the image into an array of 784 numbers like [0, 0, 180, 16, 230, …, 4, 77, 0, 0, 0], then we can feed this array into our neural

network. The computer can't understand an image by "seeing" it, but it can understand and analyze the pixel numbers that represent an image.



Graph 13: Multi-Layer Sigmoid Neural Network with 784 input neurons, 16 hidden neurons, and 10 output neurons

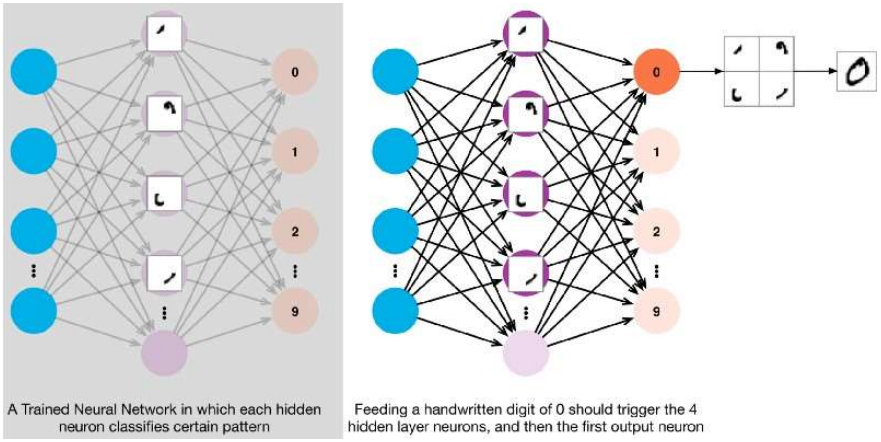
So, let's set up a neural network like above in **Graph 13**. It has 784 input neurons for 28x28 pixel values. Let's assume it has 16 hidden neurons and 10 output neurons. The 10 output neurons, returned to us in an array, will each be in charge to classify a digit from 0 to 9. So if

the neural network thinks the handwritten digit is a zero, then we should get an output array of [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], the first output in this array that senses the digit to be a zero is "fired" to be 1 by our neural network, and the rest are 0. If the neural network thinks the handwritten digit is a 5, then we should get [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]. The 6th element that is in charge to classify a five is triggered while the rest are not. So on and so forth.

Remember we mentioned that neural networks become better by repetitively training themselves on data so that they can adjust the weights in each layer of the network to get the final results/actual output closer to the desired output? So when we actually train this neural network with all the training examples in MNIST dataset, we don't know what weights we should assign to each of the layers. So we just randomly ask the computer to assign weights in each layer. (We don't want all the weights to be 0, which I'll explain in the next post if space allows).

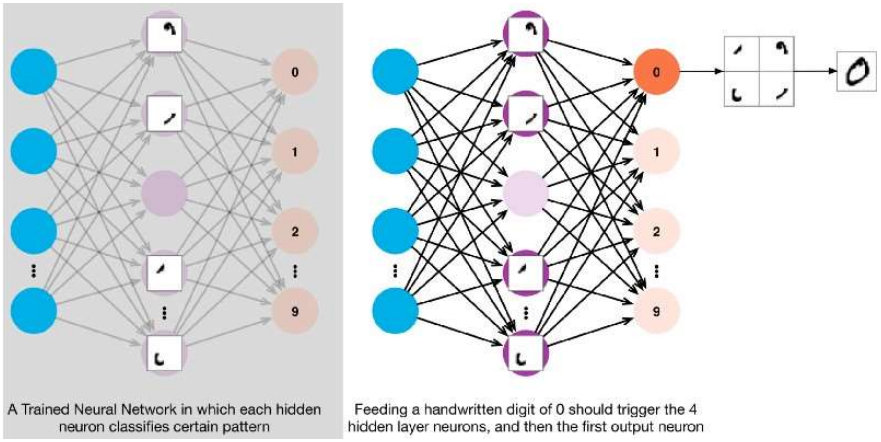
This concept of randomly initializing weights is important because each time you train a deep learning neural network, you are initializing different numbers to the weights. So essentially, you and I have no clue what's going on in the neural network until after the network is trained. A trained neural network has weights which are optimized at certain values that make the best prediction or classification on our problem. It's a black box, literally. And each time the trained network will have different sets of weights.

For the sake of argument, let's imagine the following case in **Graph 14**, which I borrow from Michael Nielsen's online book:



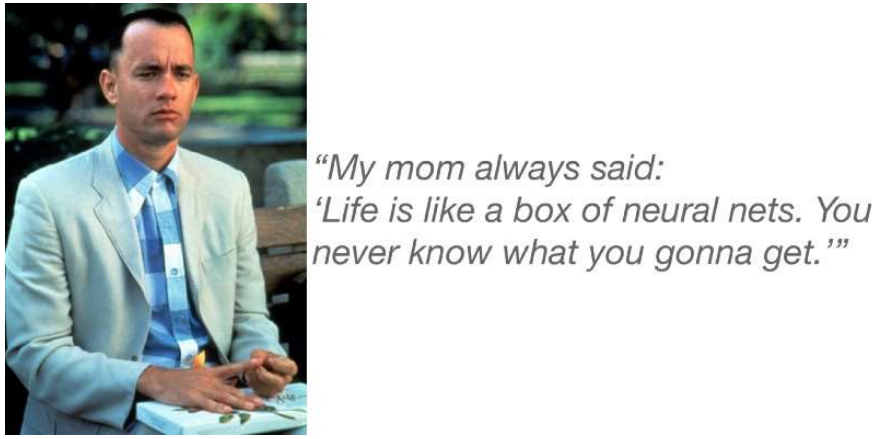
Graph 14. An Intuitive Example to Understand Hidden Layers

After training the neural network with rounds and rounds of labeled data in supervised learning, assume the first 4 hidden neurons learned to recognize the patterns above in the left side of **Graph 14**. Then, if we feed the neural network an array of a handwritten digit zero, the network should correctly trigger the top 4 hidden neurons in the hidden layer while the other hidden neurons are silent, and then again trigger the first output neuron while the rest are silent.



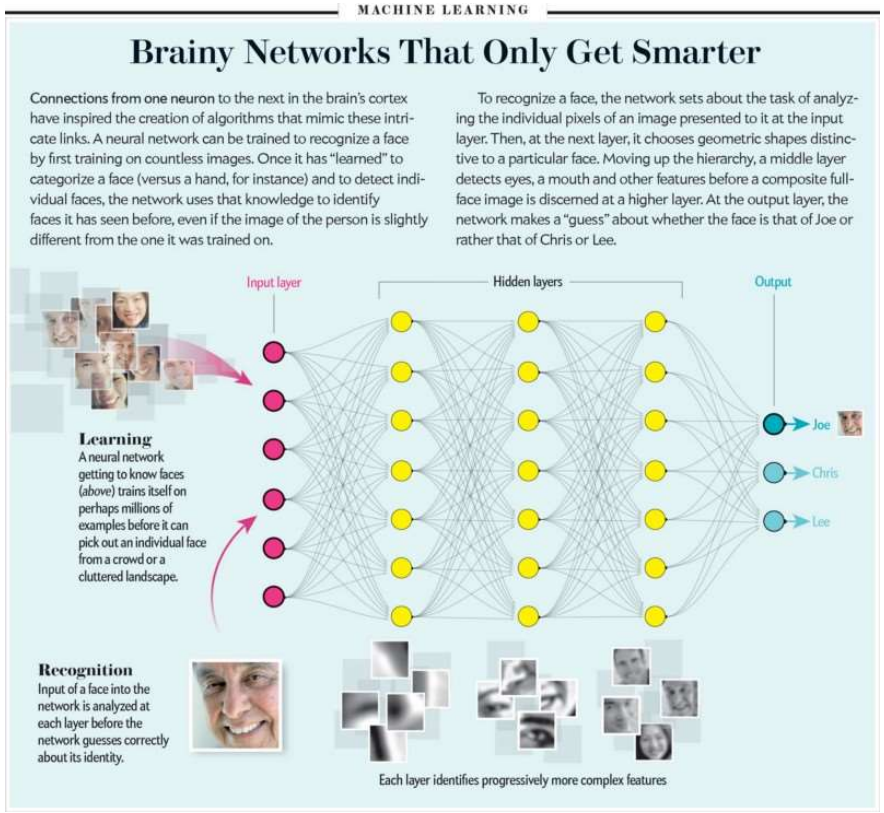
Graph 15. Neural Networks are Black Boxes. Each Time is Different.

If you train the neural network with a new set of randomized weights, it might produce the following network instead (**compare Graph 15 with Graph 14**), since the weights are randomized and we never know which one will learn which or what pattern. But the network, if properly trained, should still trigger the correct hidden neurons and then the correct output.



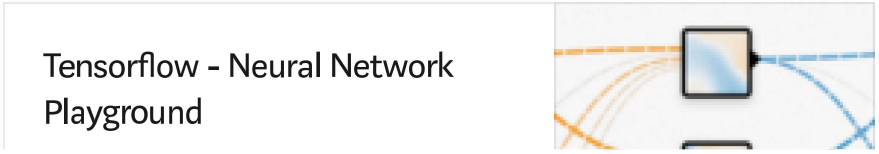
Our quote this week ;)

One last thing to mention: In a multi-layer neural network, the first hidden layer will be able to learn some very simple patterns. Each additional hidden layer will somehow be able to learn progressively more complicated patterns. Check out **Graph 16** from Scientific American with an example of face recognition :)



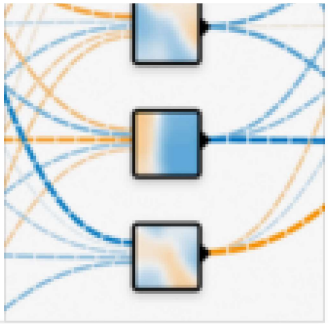
Graph 16: Each Hidden Layer Learns More Complex Features. Source: Scientific American

Some awesome people made the following website for you to play around with neural networks and see how the hidden layers work. Try it out. It's really fun!



It's a technique for building a computer program that learns from data. It is based very loosely on how we think the...

playground.tensorflow.org

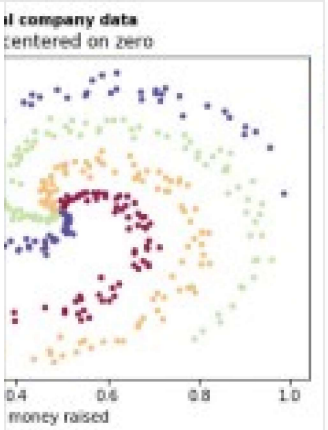


Ophir Samson wrote a nice post also explaining what a neural network is, with pretty nice visualization, and it's short and concise!

Deep learning weekly piece: what's a neural network?

For this week's piece, I'd like to focus on clarifying what a neural network is with a simple example I've put together...

medium.com



Recap

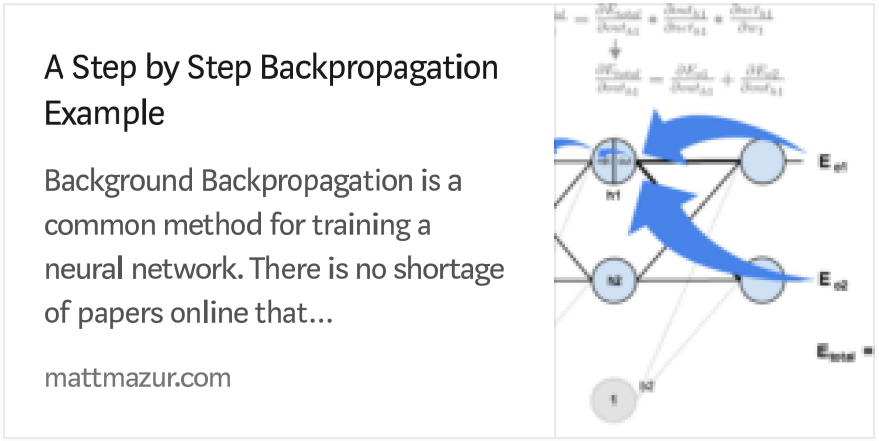
In this post, we reviewed the limitations of perceptron, introduced sigmoid neurons with a new activation function called sigmoid

function. We also talked about how multi-layer neural networks work and the intuition behind the hidden layers in a neural network.

We are almost completing a full course of understanding basic neural networks now ;) Hehe, it's not over yet! In the next post, I will talk about something called loss function and also this mysterious backpropagation that we've only mentioned but never visited! Check out the following links if you are impatient to wait:

This embedded content is from a site that does not comply with the Do Not Track (DNT) setting now enabled on your browser.

Please note, if you click through and view it anyway, you may be tracked by the website hosting the embed.



Stay tuned and, most importantly, enjoy learning :D

Did you enjoy this reading? Don't forget to follow me on Twitter!