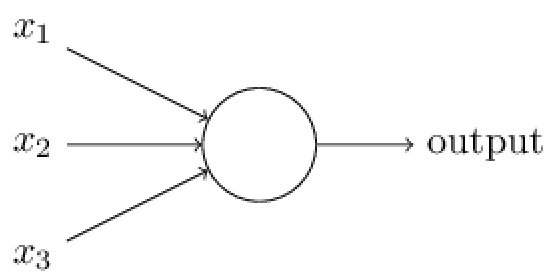
Tim Sternberg
CS462 Research Project
5/10/2018

Neural Networks

Architecture

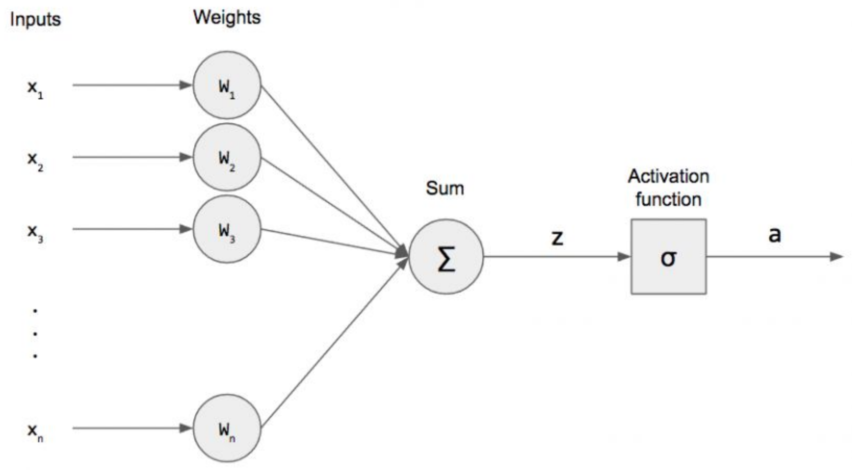
What is a neural network, and how does it work? In order to build on this question, first it's worth mentioning Perceptrons, which was a type of artificial neuron that came about in the 1950's by Frank Rosenblatt. Perceptrons take in multiple binary inputs to produce a single binary output.

from *Neural Networks and Deep Learning Ch.1* by *Michael Nielsen*



In this simple example, there's three simple x values, 1 through 3. They act as the inputs, some computational function occurs, and there's a resulting output. What Rosenblatt proposed was a simple idea of adding weights to each of the inputs, which acted as statements to the output about "how much do we care about each input".

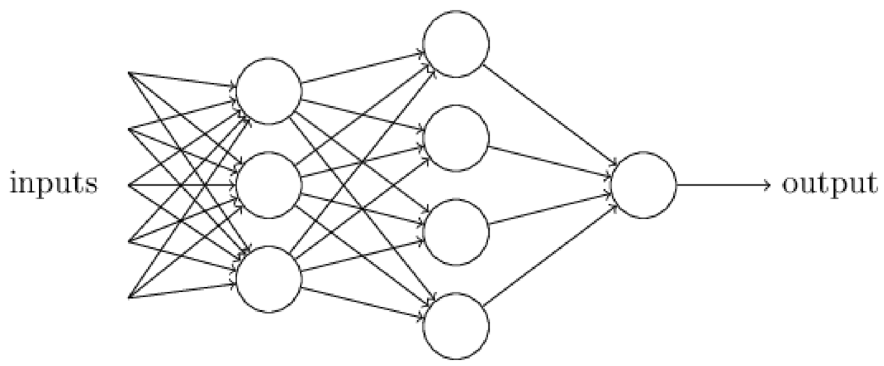
from *Perceptrons: The First Neural Networks* by *Mohit Deshpande*



This diagram shows precisely what would happen with each input, weight, and what effect they have. Each input corresponds to a weight, which then adds up to a ∑ sum, which is compared to an activation value. If the sum is greater than the activation value, then the function is fired.

For example, let's say we have an an agent whose function is to decide whether to bring a raincoat outside. x_1 is the fact that it's Tuesday, x_2 is the fact that it is raining currently, x_3 is that it will be raining later too, and the activation function value is 1. In terms of weights, our agent doesn't care about the future too much, and the fact that it's Tuesday versus any other day has no special impact on whether it needs a raincoat. But if it's currently raining then it definitely is going to want one. So we can represent this as "$x_1 * weight = 0 * 0 = 0$", "$x_2 * weight = 1 * 0 = 0$", and "$x_3 * weight = 1 * 2 = 2$". Our activation function value is 1, so we add up 0+0+2 and get 2. This value is greater (or equal to) than the activation function value so the agent would decide to bring a raincoat. Note however, these values are arbitrary and there can be an infinite amount of inputs and any weight value (as an integer, since we have binary values) associated with them. Clearly this isn't what is exactly going on in our brain, but it's certainly a step towards it. It's also fair to say that some complex system of Perceptrons can make what appear as very subtle decisions.

from *Neural Networks and Deep Learning Ch.1* by *Michael Nielsen*

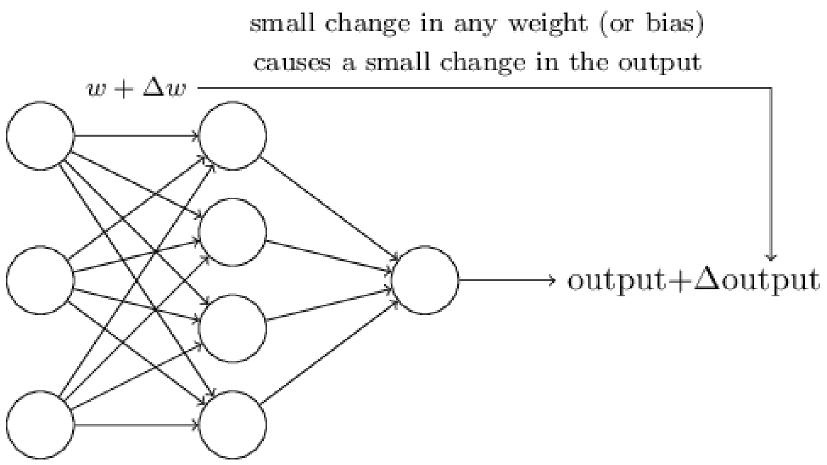


Each input neuron doesn't necessarily have to directly connect to the output neuron, or what makes the actual decision. There can be multiple layers of neurons, subsequently making their own decisions (or computations) to then reach the output layer, where the decision we care about is actually calculated. Above is an example of a network which uses multiple layers to finally come to a decision. The initial column, or “layer” of Perceptrons simply adds weights to the input. The second layer is making decisions based off of the initial decisions at a potentially more complex level. The final decision comes from the third layer, which happens to be just one perceptron, and is the final decision. Using this method of multiple layers, a much more intricate decision making process can occur, allowing for a fine-tuned network which may have not been possible with a network with just an input and output layer.

Perceptrons can achieve certain tasks at a shallow level, but given a need for more complex decision making and autonomous learning, our system of Perceptrons which act almost as logic gates may not suit our needs. But what if we could implement a type of neural network which could do all the fine tuning for us? As it made an error, it adjusts it according to how badly it failed, or how well it succeeded.

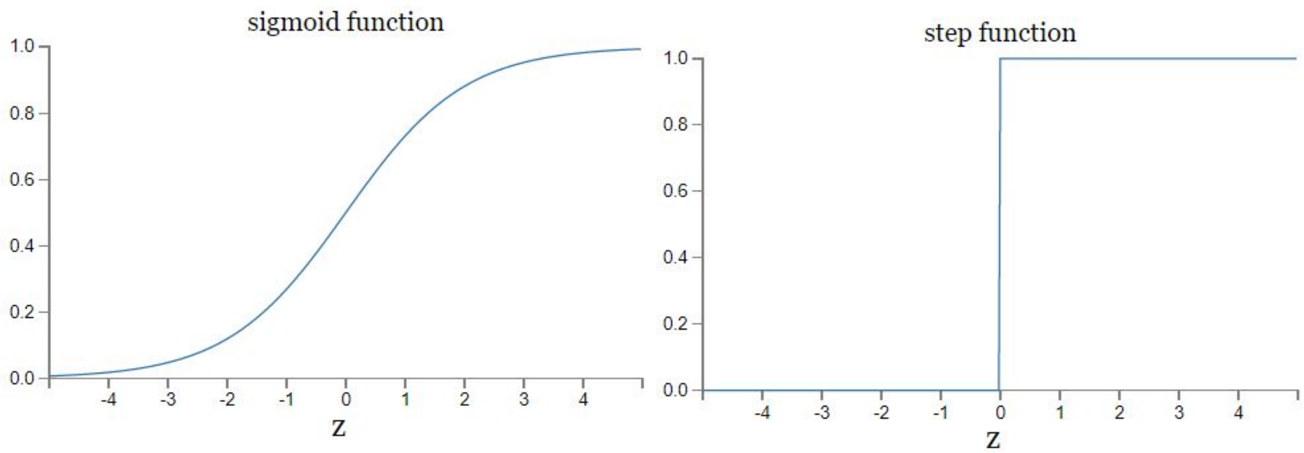
Sigmoid Neurons

from *Neural Networks and Deep Learning Ch.1* by *Michael Nielsen*


small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

Ideally when our neural network is learning, its reactions to errors should be somewhat similar in magnitude to the error itself. If the error is small, then its change to its knowledge should be respectively small. If the error is significant, its change should be too. For example, if our neural network recognizes a handwritten digit “1” as “8”, that’s a pretty serious error and there should be some hefty recalculations as a result. However if it misidentified “9” as an “8”, that’s almost understandable and the shifting of our neural network shouldn’t be as big of a change. But is this what happens with Perceptrons? Because Perceptrons are binary, meaning only 1 and 0, it's easy for our neural network to have the behavior of literally flipping like 0 and 1, with little room for slight adjustment. So although for one particular case it might classify it correctly, the entire network would adjust in a way that isn’t correct for anything else. It would be hard to monitor the network and gradually allow it learn. So why talk about Perceptrons in the first place if they don’t serve their intended purpose? The concept is beneficial because it’s a good introduction to the idea of how artificial neural networks, and why we need something that allows gradual learning.

Sigmoid Neurons are similar to Perceptrons, with the difference being in what magnitude is allowed. Similar to Perceptrons, Sigmoid Neurons have inputs x_1, x_2 … x_n and weights which correspond to each value. However the inputs x and weights w can be any decimal value between 0 and 1. “0.33944…” is possible, so is “0.9893342…”, and “0.0000100…”. Our function curve changes from that of a Perceptron to that of a Sigmoid Neuron. This is extremely beneficial, because as mentioned above, we’re able to introduce gradual learning, and our network will be able to adjust itself according to the amount of error.

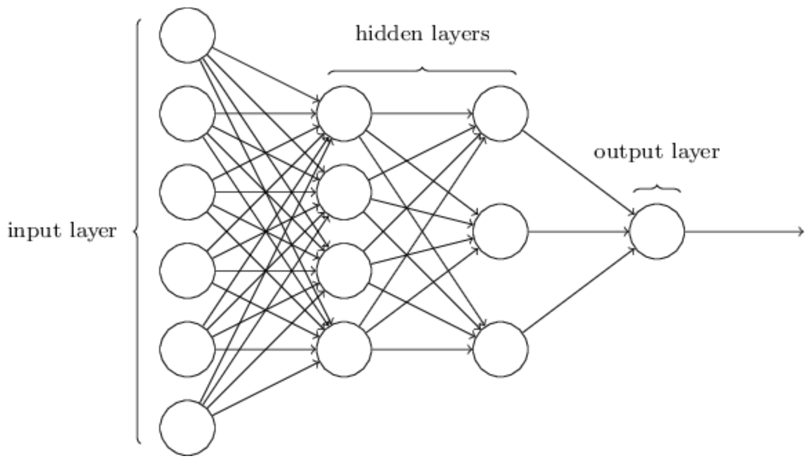


from *Neural Networks and Deep Learning Ch.1* by *Michael Nielsen*

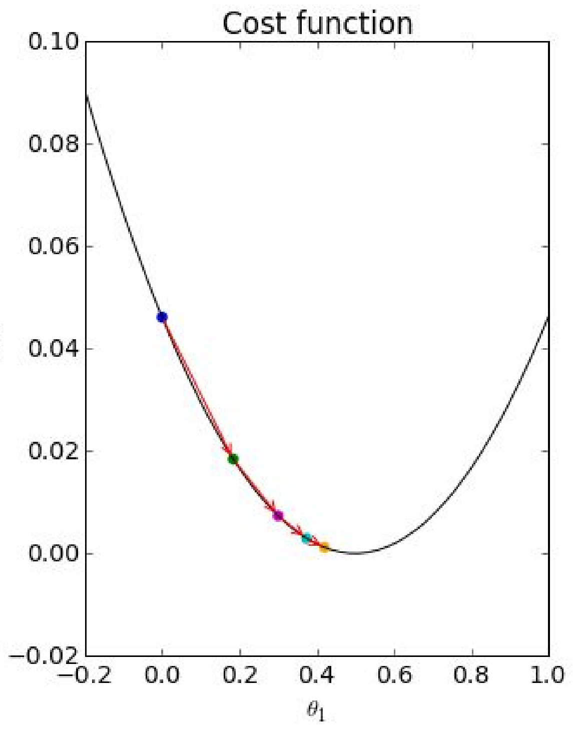
Structure Implementations

Recalling what each "layer" did in a Neural Network, the leftmost was the input layer, the rightmost was the output layer, and anything in between that was referred to as a hidden layer. There's no limit to how many hidden layers there can be. The design of a network's input and output layer is typically the same, and also straightforward. Where networks will usually begin to contrast is how the hidden layer is implemented.

from *Neural Networks and Deep Learning Ch.1* by *Michael Nielsen*

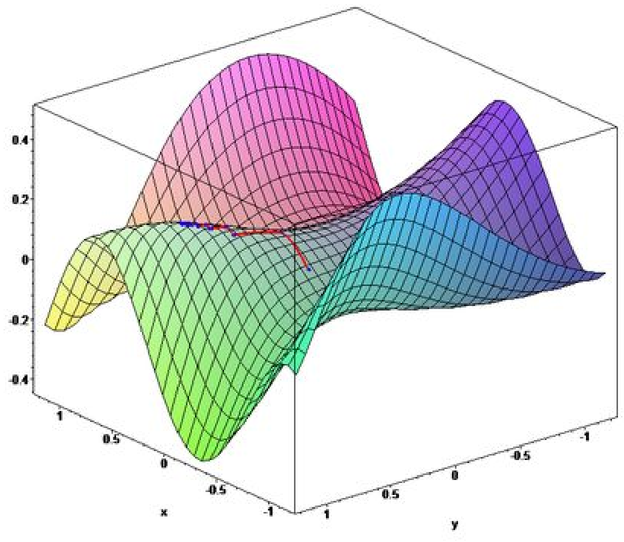


The idea that the input layer goes to the layer to the right of it, and continue on until it goes to the output layer is called a *feedforward neural networks*. The layer to the left of another is acting as its own input layer. No layer on the right is going back and inputting information back during the initial calculations. Something called back propagation does do something along these lines, but this will be talked about later. If each neuron was able to loop back to previous neurons, the network would contain have multiple loops and would end up being a serious concept to tackle. There could be thousands of connections throughout the network, each being interconnected with each other across multiple layers. Models similar to this are *recurrent neural networks*. The idea is to have each neuron fire for a limited amount of time before going inactive. This prevents infinite loops, which would be really bad and could easily crash a computer in seconds. Each neuron would stimulate some other one, and this would go on for some finite amount of time. This implementation could accomplish the same thing as a feedforward network, however right now feedforward designs have proved to be much more powerful. Despite this, it doesn't take away how interesting recurrent networks are. They act in a way which would be similar to how our actual brains are firing neurons. It's possible for recurrent neural networks to solve problems that feedforward networks are incapable of doing.



Gradient Descent

In order to get an idea of why gradient descent is a foundation for neural networks, imagine a cost function C which takes in some input and spits out some output, similar to the graph on the right. Each point in the graph is a result of the cost function.



Graph from Visualizing the gradient descent method from wikipedia.org/wiki/Gradient_descent

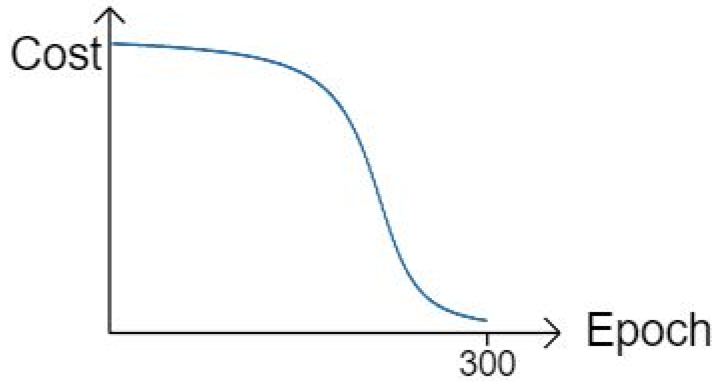
However in our case of neural networks, imagine the “cost” as the amount of error. Clearly the intention of the network will be to have as low of an error/cost as possible. In other words, it wants to be at that local minima of the cost function. It’s beneficial to imagine the result of the network adjusting itself to be looking around the graph trying to find the lowest cost. However as a function gets more complicated, which is often the case in neural networks, explicitly finding the local minimum may not necessarily be possible. Take for example the graph above which depicts a 3D plane with multiple slopes and local minima. A more feasible tactic to find a local minima would be to start at any position on the graph (meaning, randomly select a position) and from there, depending on the slope shift the position in a way that it decreases in the Y position. After doing this repeatedly, each time taking an appropriate step to get the lowest output from the cost function, eventually you should find a local minimum of the function. If the step sizes for each “nudge” is proportional to the slope, then at a large slope will result in a large step, and as you approach the minimum the step will get smaller. This behavior is similar to a ball falling down a hill, and prevents overshooting the local minimum. However, it’s also possible that each separate run of this the function may not result in the overall minimum. And depending on where the random initial position is, there may be different results each time.

The application of gradient descent is largely used in what drives a neural network to learn. When an artificial neural network receives input, the information is relayed to the following nodes. Although this is the implementation of a feedforward network, this technique specifically refers to forward propagation. Gradient descent comes into play when the network is wrong. Initially if the network starts out with randomized weights, the output is bound to be seriously incorrect. Backpropagation is a technique to go backwards in the network, updating the weights as it goes along, once the decision has been made. The total error at the final output nodes is propagated backward in the network to each node and the weights are adjusted using gradient descent. The mathematics behind back propagation is extremely heavy, and an essay of this size wouldn’t even be enough to cover it all. This is why rather than talking about the equations, the focus is on the effect of back propagation.

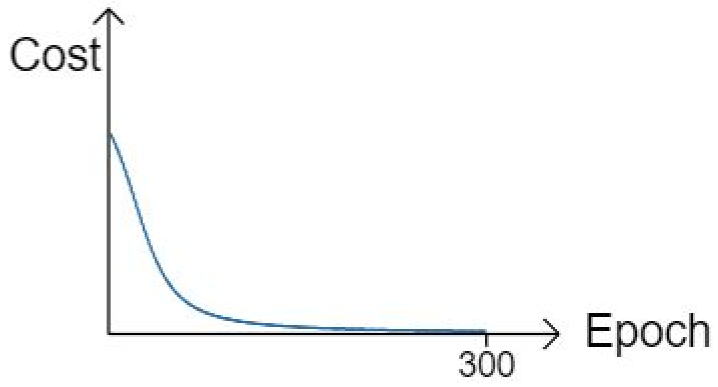
Where Neural Networks Can Be Improved

Now that the fundamentals of neural networks have been covered, the next question that should be asked is how to improve what we already have. To start, take a look at how the neural network itself is learning. More specifically, think about how it should be learning. Ideally, when we design a neural network we want it to both replicate how the biological system we’re modeling it after is operating, and also behave in a similar manner. That being said, in the realm of learning, how do we as humans learn? Someone first learning guitar is bound to make mistakes at first, similar to an artificial neural network starting out. Playing a G chord can prove to be rather difficult at first, but given enough practice it becomes one of the easiest chords to play. That practice however, comes with making mistakes. And seeing those mistakes is how we're able to identify what was wrong, and we adjust our methods accordingly. The neural network, given its current cost function, does eventually do this. However the difference lies in *when* it does this.

from *Neural Networks and Deep Learning Ch.3* by *Michael Nielsen*



The graph on the left is an example of the neural network starting out with a result that was very wrong. The desired cost is as close to 0 as possible, and it started very high. It wasn't until around epoch 150 (around halfway through) that it began to drastically adjust itself. *Note that an epoch is the number of times that specific training vector has been used in each iteration*.



Contrast that to this graph. The difference is that here the neuron here starts out at a lower cost. The conclusion to derive here is that given a higher cost/error, the network takes longer to adjust. This is a considerable difference than what we want from our neural network iff we want it to replicate human behavior. If a person learning guitar plays an D minor chord instead of a G chord, the sound is night and day. It's easy to distinguish the error, both in where the fingers on the fretboard should be, but also what sound should be produced. If we took the approach our neural network is taking, then we would have to play the D minor chord in quite a few alterations before realizing it's not what we wanted.

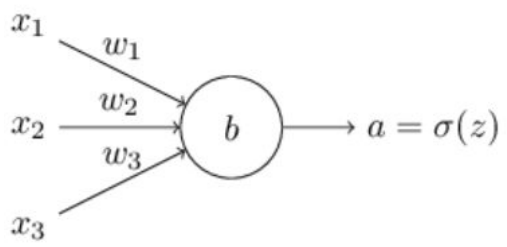
The root of the problem comes down to the cost function. Each neuron learns by adjusting the weight and bias which is ultimately dictated by the partial derivatives of the cost function: *∂C/∂w* and *∂C/∂b,* where *w* is the weight and *b* is the bias.. If the neural network is not learning fast enough, it is because these partial derivatives are not big enough. I skimmed over the math intensive material in previous sections to focus on the more general idea of neural networks, but in essence, the cost function previously stated is the quadratic cost function, given by the following formula: $C = \frac{(y-a)^2}{2},$ "where a is the neuron's output when the training input x=1 is used, and y=0 is the corresponding desired output. (Neural Networks and Deep Learning, Michael Nielsen). Given that a = *σ(z)* where *z = wx+b*, it's possible to then derive the following:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$
$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z),$$

This is significant, because given the sigmoid function curve, if you substitute x = 1 and y = 0 it's clear why these two partial derivatives get so small. This is why the cost function behaves the way it does given a large initial error. This is constant throughout multiple implementations of neural networks, despite their purposes. It’s clear that the cost function has some serious downsides, and there needs to be an alternative.

Cross-Entropy Cost Function

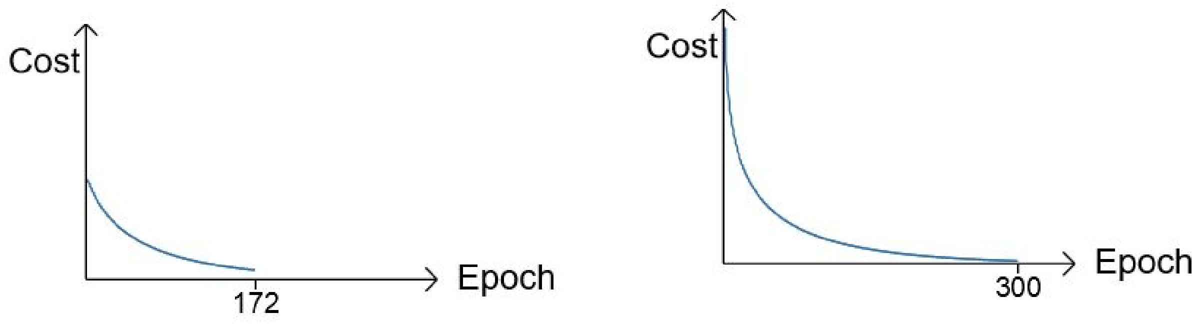
It’s clear that the basic quadratic cost function is not serving its purpose entirely as intended. The need for a more accurate and precise function becomes more clear. This can be solved with the Cross-Entropy Function, which is defined as followed.



Recall the basic neuron model, with multiple inputs all with corresponding weights, along with an overall bias. This value, together, is $z = \sum_j w_j x_j + b$. Therefore, it can derived a=σ(z), where σ is the Cross-Entropy Function. The function itself is defined as the image on the right, where n is the amount of training data, the sum is over all those inputs, and x and y are the desired outputs.

$$C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln(1-a) \right],$$

Although at first glance it simply looks like a more complicated function, it actually has significant improvements over the quadratic cost function. This paper won’t go into the math too heavily, but it is important to understand why this should even be defined as a cost function. The first reason being that the cost C will always be greater than 0. This is mainly due to two reasons; “all the individual terms in the sum in are negative, since both logarithms are of numbers in the range 0 to 1, and there is a minus sign out the front of the sum” (Neural Networks and Deep Learning, Michael Nielsen). The second reason this can be defined as a cost function is that if the output of the function are indeed close to what we want given the training inputs, then the function will yield an output close to zero. An example run of the new and improved cost function can be given below.



from Neural Networks and Deep Learning

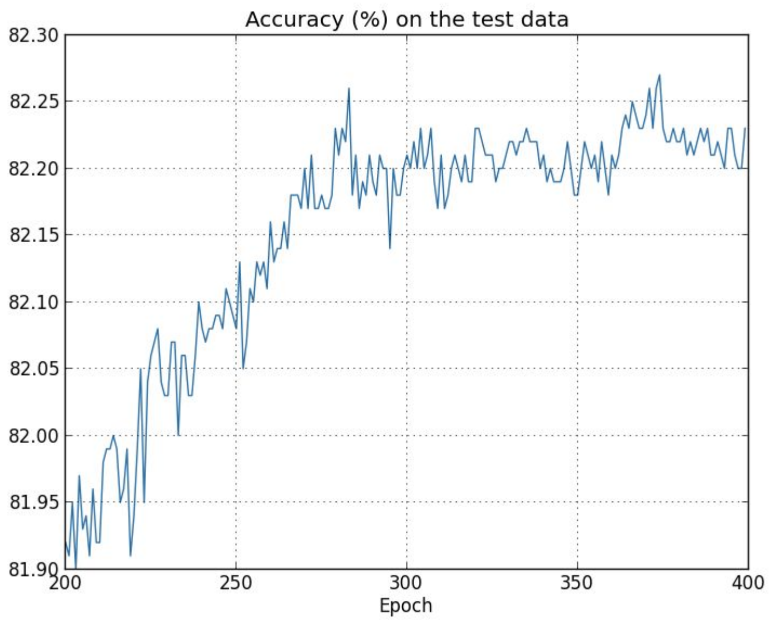
As you can see, this is a significant improvement from the quadratic cost function. Both in the case where the network starts out doing well, and when there's a significant error at first. This is what we want, because it more closely resembles actual human behavior (and is much more efficient overall).

Softmax

Although softmax layers are more typically used in deep learning, which will be talked about later, implementing this approach is still an interesting topic to talk about, and is worth noting. Essentially what a softmax does is defines a new output layer for the neural network, which works the same way as other layers but doesn't apply the sigmoid function like the other layers. Rather there is a *softmax function* applied to the inputs and their corresponding weights (and the bias of course). The math is little involved, but in essence the softmax solution addresses the slowdown problem similar to the cross-entropy cost function. However the result of applying the softmax function is that the output from the softmax layer will be a probability distribution. This is advantageous because it's easier to be able to have the output activation be the estimate of the probability that the output is what we want. If instead we were using a regular old sigmoid layer for the output layer, it's not plausible to assume the activations are a probability distribution. Therefore, there isn't such a simple derivation of what the output activation is. Overall, what the softmax approach gives us is an output activation which has more meaning attached to it, therefore making the network results easier to analyze.

Overfitting and Regularization

What is overfitting and why does it matter? Overfitting is when the neural network "overtrains", meaning it goes through the training data past a point than is what worth, given the amount of time spent spent training data. An example below represents an example of this which shows that around Epoch 280, the accuracy the network levels out. It might almost be beneficial to stop the training data around epoch 280, as the difference in accuracy between then and epoch 400 is nearly the same.



The most obvious way to detect overfitting is to watch the accuracy of the network as it goes through the training data. If there's some level-out, then stop the training and move on. That being said, after how much “leveling-out” do you determine overfitting is occuring? More importantly, what if it’s temporary? Say you have 2000 epoches, and at epoch 250 a short leveling-out occurs. If the neural network were to stop training at that point, you’re avoiding a majority of the training dataset.

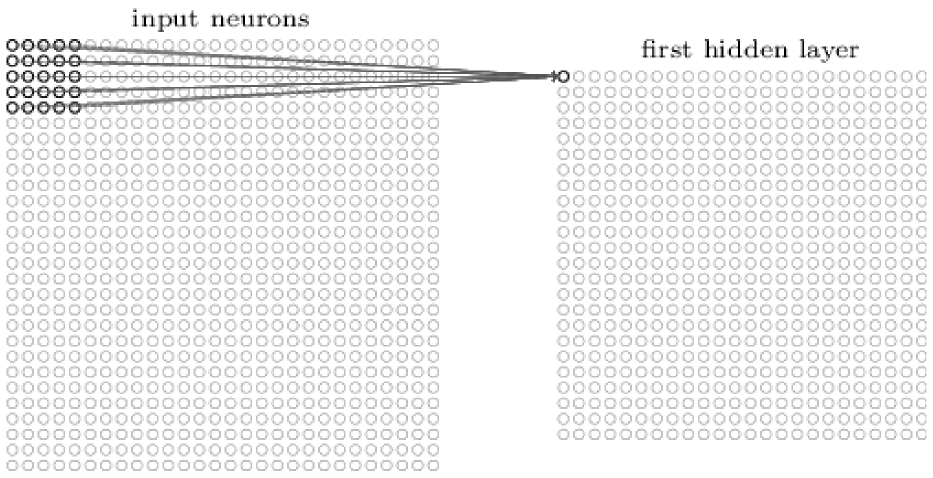
One solution which could possibly reduce overfitting is to reduce the size of the network. However this is not at all an ideal solution because larger neural networks have the potential to be considerably more powerful. Other approaches, known as regularization have proven to be both successful and cost-efficient. One of the most common approaches is called L2 regularization, which involves adding a term to the cost function. This is referred to as the regularization term. Such a term could look like $(\lambda/2n)*\sum w^2$. This is the “the sum of the squares of all the weights in the network. This is scaled by a factor $\lambda/2n$, where $\lambda>0$ is known as the regularization parameter, and n is, as usual, the size of our training set” (Neural Networks and Deep Learning, Michael Nielsen). Note that this regularization technique can also be applied to the quadratic cost function. In general, it can be written as $C = C_0$ + Regularization Term. The point of regularization is to assure that the network prefers smaller weights over larger ones, unless they improve the network a considerable amount. The reason that smaller weights are important is because given a single change in input, a neural network should not drastically change. Rather, with single small changes in input, there should be somewhat small adjustments made. In an “unregularized” network, this is not the case. Large weights can cause drastic changes, and as a result can have unpredictable changes. What’s interesting about regularization, however, is that no one really knows exactly *why* it works. There are a ton of different approaches to regularization, all which have compelling traits, and all which definitely do work.

And although gradient descent offers what in effect is self regularization, researchers don't necessarily know why. Overall, the root cause as to why regularization causes generalization within an artificial neural network is largely unknown. Note that generalization simply refers to the effect of learning. If a child sees an animal for the first time, and is shown 10 seperate pictures, they will largely be able to pick out that same animal again in the future with minimal effort and error. They've generalized what the shape of this new animal should be.

Deep Learning

As to probably be expected, the more complicated an artificial neural network gets, the harder it becomes to train. The term deep learning refers to the depth of the neural network. The networks previously introduced were comparatively shallow networks. So it's easy to think of the learning applied to a deep network as deep learning. Networks which utilize deep learning are inherently structured differently than normal feedforward networks because it's somewhat unnatural to have something such as a 2D grid be mapped to a 1D input. Convolutional Networks are an application of neural networks which use the given space to its advantage.

There's three main pieces to a convolutional network. The first being the idea of local receptive fields. Think about what a facial recognition software is doing. It's picking up on pixels in the picture and comparing it to an already existing photo.



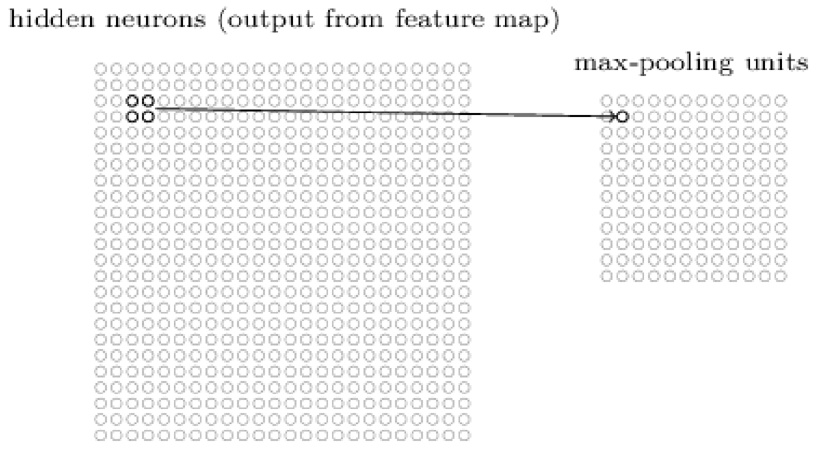
from Neural Networks and Deep Learning

In this case, the first hidden layer would consist of the initial 5x5 grid from its input layer. Each input neuron in this hidden layer has its own mapping very similar to the top left neuron, just mapped according to how it’s structured in the initial input layer. In this case, it’s a 28x28 pixel photo. However the hidden layer would be a 24x24. “This is because we can only move the local receptive field 23 neurons across (or 23 neurons down), before colliding with the right-hand side (or bottom) of the input image (Neural Networks and Deep Learning, Michael Nielsen).

The second main idea is that each input in that first hidden layer has the exact same weight. This means that each input in the hidden layer will detect the same pattern, just in a different spot, or even a different shape. This map from the input layer to the hidden layer is called a feature map. If a single hidden neuron is able to pick up on a particular shape such as a curved edge, chances are that another neuron somewhere else will have to be able to detect something similar. In addition to this, there’s less parameters needed for the network overall. Instead of 784 input neurons (for a 28x28 grid), there would only need to be 520 parameters.

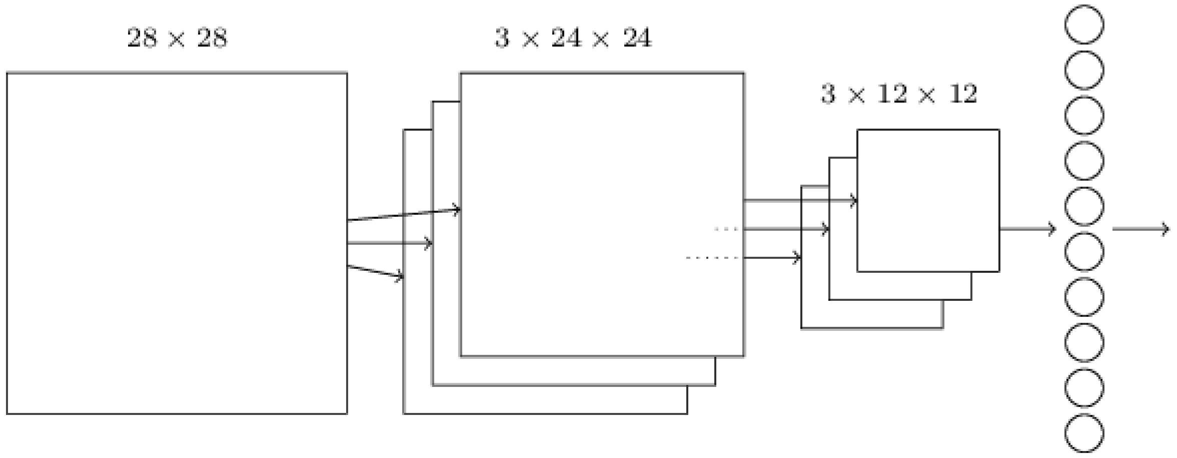
The final piece to convolutional networks is pooling layers, which are typically used right after convolutional layers in order to simplify the data being outputted. One way to do this is implement max-pooling, which outputs the maximum activation in a 2x2 space.

from Neural Networks and Deep Learning



In this case, the 24x24 grid has now been mapped to a 12x12 grid. It's like figuring out whether a specific important feature is located on the map or not. The exact location on the map isn't necessarily as important as the fact that it exists.

Putting this all together, we get the following convolutional network.



The 28x28 original input neurons encode the pixels for the original image. A 5x5 piece of the input layer is then mapped to feature maps, which is then mapped to a max-pooling layer which applies the 2x2 regions previously mentioned before. The output layer is then structured similarly to a classic feedforward network, however the path to get to it was much different.

Conclusion

Artificial neural network research is in its infancy. So many topics within neural networks are still unknown to researchers, such as the reason regularization works the way it does. But despite this, its applications are still fall reaching. Ranging from a system which recognizes hand digits and faces to something which could potentially save lives by identifying when a seizure is occuring, the impact for machine learning is nearly infinite. There’s no telling what neural networks has in store for the world.

Citations

Main Research:
http://neuralnetworksanddeeplearning.com/chap1.htm

Other literature used:
https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/1

https://www.youtube.com/watch?v=LN0PLnDpGN4

https://www.youtube.com/watch?v=IHZwWFHWa-w

https://scipython.com/blog/visualizing-the-gradient-descent-method/

https://en.wikipedia.org/wiki/Gradient_descent