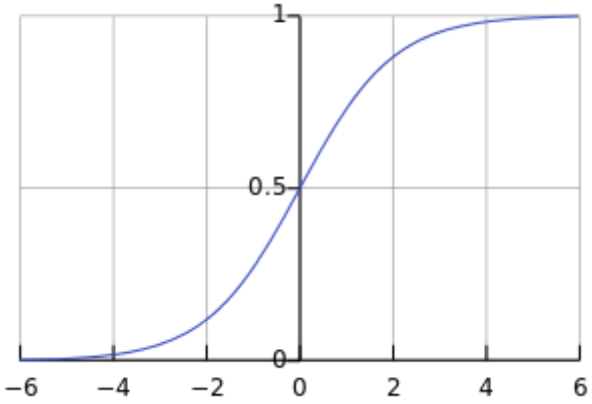
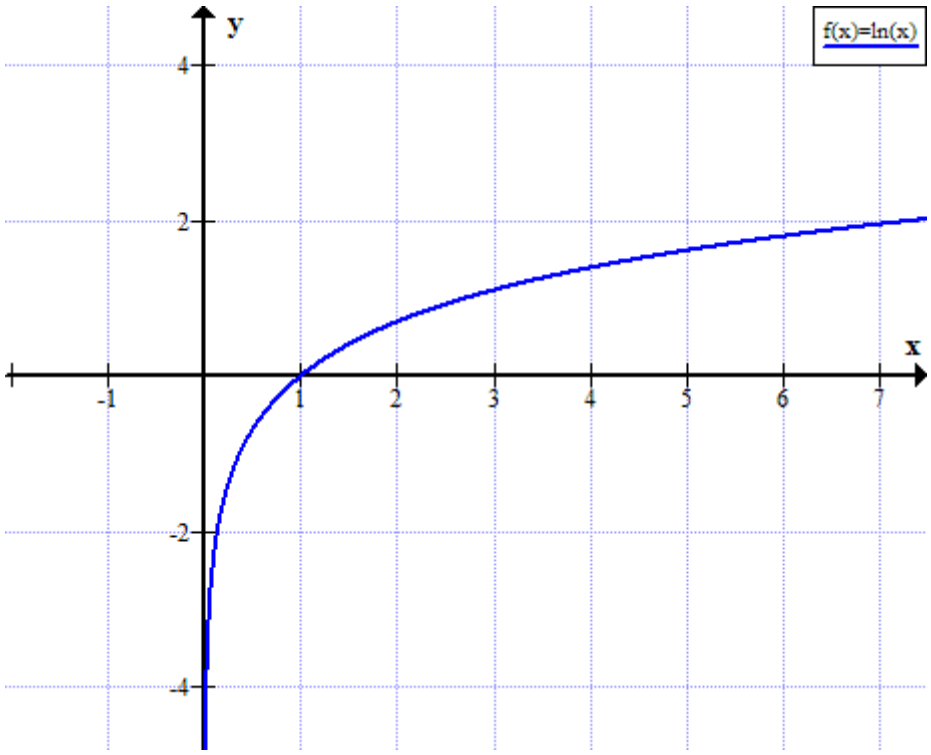
chapter 3: improving the way neural nets learn

- in this chapter, we'll look at how to learn better. we'll talk about a better cost function, how neural nets can generalize better beyond the training data, a better way to initialize weights and some heuristics to choose hyper parameters for the network.
- first, convince yourself that we haven't given any reason for stochastic gradient descent to come to the same set of weights after a fixed number of epochs of learning. this does depend on the initialization of weights. the other source of randomness is in how we pick the mini batches.
- specifically, an unlucky initialization of weights could lead to the cost being not-so-optimized after a fixed number of epochs, as compared to a more fortunate set of initial weights. this all really just depends on the shape of the cost curve (specifically, the values of $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ at different points on the cost curve).
- Saying "learning is slow" is really the same as saying that those partial derivatives are small.
- From chapter 2, note the cost equations of backpropagation:
  - $\delta^L = (a^L - y) \odot \sigma'(z^L)$ (for the final layer)
  - $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$ (the equation that "propagates back the error")

  Notice that in the above equations, the $\sigma'(z)$ term is really low when z is either high or z is low (that means one or more of the weights are too high or too low). When this is true, the rate of learning is low because $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ are low.



- We can see from this graph that when the neuron's output is close to 1, the curve gets very flat, and so $\sigma'(z)$ gets very small. This is the origin of the learning slowdown.
- Cross entropy cost: (we'll define it now and see why defining it this way might be advantageous in a bit):
  - $C = \frac{-1}{n} \sum_x [y ln(a) + (1-y) ln(1-a)]$
- Ok, time to break this crazy looking thing down. First note the shape of ln (x) below:



- Note that the domain of $\sigma(z)$ is (0, 1). It doesn't take on 0 or 1 itself.
- Convince yourself why this cost function is good. If a predicts y the cost is low, if not the cost is high. Its easy to convince yourself why this is true if y can only take on values 0 and 1. Also, note that the cost is always positive given the (0, 1) domain. These are all properties the cross entropy cost function shares with the quadratic cost function.
- to see why this is a valid cost function even if the only possible values for y are not 0 and 1, graph the cross entropy function in a 3d graphing calculator. you can see how cost is least (not 0 though) when y = a and a lot more otherwise.
- why might this solve the slow learning problem? consider the derivative of this cross entropy cost function with respect to weights:
  - $\frac{\partial C}{\partial w} = \frac{1}{n}\sum_x (\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}) * \frac{\partial \sigma(z)}{\partial w_j}$

  Very conveniently, the above reduces to:

  $\frac{\partial C}{\partial w} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$

  Similarly,

  $\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z) - y)$

  This is great because the greater the error, the faster the learning. We'll see why we get lucky and the intuition behind this in a bit. Note that this is only true because we are using the sigmoid function as the activation function. This simplification of the expression and convenience is only because the activation function is the sigmoid function.

  caveat: this problem of slowed down learning also only shows up because we use the sigmoid function as the activation function. i.e, the origin of the problem is the derivative of the sigmoid function being low.
- Although we've stated (and is easily provable) why we achieved this super nice "more the error, faster the learning" property, it is not intuitively clear what about the cross entropy function gives us this benefit. We will come back to this question later.
- We sum up the cross entropy function across all neurons of the output layer:

$C = \frac{-1}{n} \sum_x \sum_j [y_j ln(a_j^L) + (1 - y_j) ln(1 - a_j^L)]$

- the cross-entropy is nearly always the better choice, provided the output neurons are sigmoid neurons. this is not true if the neurons are not sigmoidal.
- in machine learning, after we've crossed the 90% accuracy rate, even small improvements are very good. lets say error we go from 96% to 96.5%. Look at the reduction in error rate: 1 in 8 inputs that were being misclassified before are now classified correctly.
- also, note that in machine learning, when we get an improvement, we should be skeptical of considering them "proofs" that one model is better than the other. cross entropy could do better than quadratic cost but how do we know we choose the perfect set of hyper-parameters when evaluating each. often we don't know that. yet, such improvements, if evaluated reasonably, provide evidence of improvement, even though not proofs.
- neuron saturation is an important problem in neural nets.
- cross entropy cost is not a magical thing we came up with that can be proved to work. if we want a $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ that does not have a $\sigma'(z)$ term, we can in fact get that by starting at:
  $\frac{\partial C}{\partial b} = (\sigma(z) - y)$. on integrating, we can arrive at the cross entropy cost function.
- note that there is still the x_j term in the derivate of the cost function. this slows down learning when it has a low value, but this cannot really be eliminated because this comes from $\frac{\partial z}{\partial w} = x_j$ and is not coming from the cost function.
- The idea of softmax is to define a new type of output layer for our neural networks.
- softmax is a term for the usage of the following *softmax* function for the output layer:
  $a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$
- Note that this has the property that $\sum_j a_j^L = 1$. Also, the output activations are all positive, since the exponential function is positive. This enables us to interpret the final layer output *kind of* like it is a set of probabilities (since they add up to 1 and are all positive).
- In many problems it's convenient to be able to interpret the output activation a_j^L as the network's estimate of the probability that the correct output is j.
  - question: why use e^x to generate probabilities? why not just use some other function?
- Note that for the softmax layer, the output is not just a function of the corresponding weighted input.
- With softmax, our cost function can just be $C = -ln(a_y^L)$ (where $a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$) → this is log likelihood cost, where y is the correct class. This is because the output a_j^L has already taken care of its relation to the outputs of the other output layer neurons. So the log-likelihood cost behaves as we'd expect a cost function to behave.
- Turns out if we use softmax + log likelihood cost, the expressions for $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ for the outer layers are the same as that if we had used sigmoid output layers and cross entropy cost. (pretty easy to derive this). In fact, it's useful to think of a softmax output layer with log-likelihood cost as being quite similar to a sigmoid output layer with cross-entropy cost.
- which one should we use? for many problems, both approaches work quite well. influential academic papers often use softmax + log likelihood cost.
  - why?
- softmax plus log-likelihood is worth using whenever you want to interpret the output activations as probabilities. can be useful with classification problems involving disjoint classes.
- softmax is so called because the probability distribution it produces is smooth ("softened") as compared to something like
  $a_j^L = \frac{e^{cz_j^L}}{\sum_k e^{cz_k^L}}$ with $c > 1$. (when $c = 1$, we get softmax).

- regularization: models with a large number of free parameters can describe an amazingly wide range of phenomena. Even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of the given size, without capturing any genuine insights into the underlying phenomenon. they'll work well on data they've seen before, but the true test is how well do you do on data you haven't seen before?
- neural nets can easily run into millions, and even billions of free parameters (most of which are weights and biases). given the above statement, should we trust them?
- Overfitting is the thing that happens when the model learns peculiarities of the training data, does really well on the training data, but much worse on the test data. the model hasn't learnt general principles to help it do well like it has learnt the peculiarities in training data.
- ways to avoid overfitting:
  - *early stopping:* we compute the classification accuracy on the validation data at the end of each epoch of training on the training data.. Once the classification accuracy on the validation data has saturated, we stop training. This strategy is called *early stopping*. the idea is to use the validation data to evaluate different trial choices of hyper-parameters such as the number of epochs to train for, the learning rate, the best network architecture, and so on. We use such evaluations to find and set good values for the hyper-parameters. To put it another way, you can think of the validation data as a type of training data that helps us learn good hyper-parameters. This approach to finding good hyper-parameters is sometimes known as the *hold out* method
  - *regularization* is a set of techniques to also reduce overfitting. we'll see 4 instances of this class of techniques here, listed below. before that, let us see why large values of weights might be bad for generalization.
    the 4 regularization techniques:
    - *L2 regularization:*
      $C = C_0 + \frac{\lambda}{2n} \sum_w w^2$, where λ is the regularization parameter and n, as usual, the training set size.

      Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first part of the cost function. The relative importance of the two elements of the compromise depends on the value of λ.
      why do smaller weights mean better generalization?
      side story: occam's razor: given two models that quite accurately fit a set of given data points, it is more surprising for the more complex one to be true. now this is very rough, has no logical basis and what does "complex" even mean anyway? but its still a useful heuristic. why? because it often works well pretty often. I know. that's frustrating.
      now how does this apply to neural nets? The smallness of the weights means that the behavior of the network won't change too much if we change a few random inputs here and there. That makes it difficult for a regularized network to learn the effects of local noise in the data. Instead, a regularized network learns to respond to types of evidence which are seen often across the training set.
      - question: why might this be true?

      but remember: the true test of a model is not simplicity, but rather how well it does in predicting new phenomena, in new regimes of behavior.
    - *L1 regularization*:
      $C = C_0 + \frac{\lambda}{2n} \sum_w |w|$. From this, we can derive the weight update rule:

In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w. And so when a particular weight has a large magnitude, |w||w|, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when |w||w|is small, L1 regularization shrinks the weight much more than L2 regularization. The net result is that L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero.

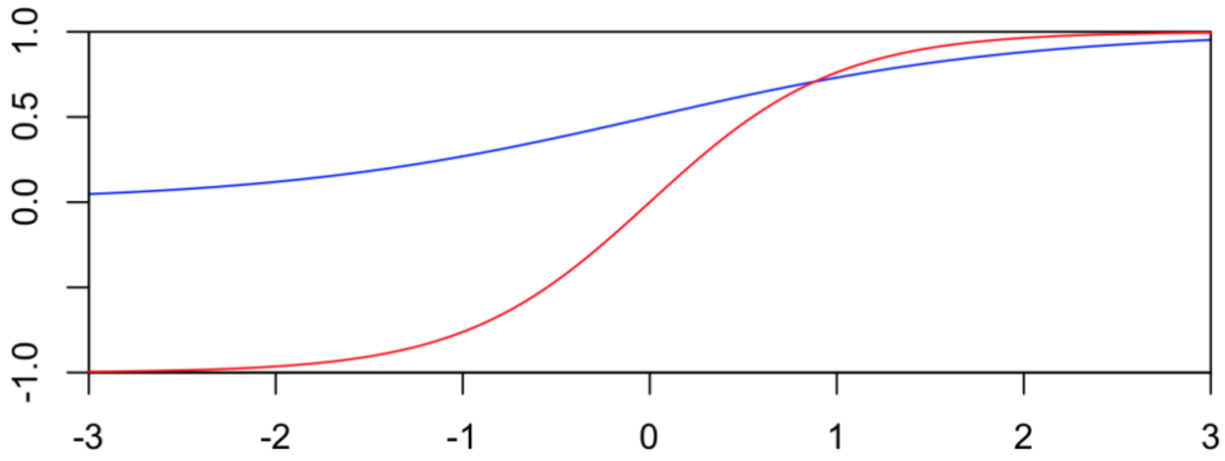
- *Dropout:* With dropout, we drop about half the neurons, forward propagate, backpropagate to update the weights and update the weights. for the next mini-batch, we restore the dropped neurons and drop another set of neurons. When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons. when we dropout different sets of neurons, it's rather like we're training different neural networks. And so the dropout procedure is like averaging the effects of a very large number of different networks. The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting. “This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.”

- *artificially expanding the training set:* The idea is to introduce random noise, that we know will occur in the input data. It is not always easy to introduce noise to expand the training data such that the it remains representative of real world data that the network might see. For example, on the MNIST digit dataset, we can introduce a bit of rotations to the available training data. We can expand our training data by making *many* small rotations of *all* the MNIST training images, and then using the expanded training data to improve our network's performance. we need to be sure to apply operations such that the result is still representative of real world data.

- caution: Many papers focus on finding new tricks to wring out improved performance on standard benchmark data sets. Such claims are often genuinely interesting, but they must be understood as applying only in the context of the specific training data set used (not necessarily a general technique).
- The message to take away, especially in practical applications, is that what we want is both better algorithms *and* better training data. It's fine to look for better algorithms, but make sure you're not focusing on better algorithms to the exclusion of easy wins getting more or better training data. It's entirely possible that the "improvement" due to the great new technique would disappear on a larger data set.
- weight initialization:
  - note that the strategy of using cross entropy cost or softmax (with log likelihood cost) only helped the output neuron saturation issue. it doesn’t address saturation in the hidden layer neurons.
  - Michael first initialized the hidden layer neurons by sampling from a standard normal distribution. the downside of this method is that when you compute $w^T x + b$ the sum has a good chance of being very high or very low, which leads to saturation. Instead, if we start with super small gaussian variance ($\frac{1}{\sqrt{n_{in}}}$ where n_{in} is the number of incoming edges) instead of variance 1, then $w^T x + b$ tends to be at a place where it can learn a lot faster.
  - Michael shows that with standard normal initialization vs the second method, we converge to a given accuracy in a lot fewer iterations of training. In his experiment, this new way to initialize does not change the accuracy, just gets us to a given accuracy in a lot fewer iterations.

    - L2 regularization sometimes automatically gives us something similar to the new approach to weight initialization since it emphasizes w_{ij}'s that are not too big. The initial few iterations of training are dominated by decreasing value of weight magnitudes if we start off with high weights → this tells us something interesting → whichever route we get there, not too large weights seem to be really useful.
- Side note → implementing $L2$ regularization is just a 1 line change! "*This is, by the way, common when implementing new techniques in neural networks. We've spent thousands of words discussing regularization. It's conceptually quite subtle and difficult to understand. And yet it was trivial to add to our program! It occurs surprisingly often that sophisticated techniques can be implemented with small changes to code.*" - *Michael*
- How to find good hyperparameters:
    - It's easy to feel lost in hyper-parameter space. This can be particularly frustrating if your network is very large, or uses a lot of training data, since you may train for hours or days or weeks, only to get no result.
    - During the early stages (when the network is just initialized and has learnt nothing) you should make sure you can get quick feedback from experiments. this means significantly reducing training data size, printing out output frequently to monitor progress and using a smaller network. The whole goal is to iterate thru a bunch of hyperparameters quickly to find good candidates to explore further)
    - we'll start with finding a good η (learning rate) and λ ($L2$ regularization parameter). To find a good order of magnitude for learning rate, just plot the error for a few choices and see how the error changes. some will be too small (slow decrease in cost), some will be too large (we'll overshoot the minimum error) and will see a bunch of crazy oscillations. the right value is inbetween these.
    - it's often advantageous to vary the learning rate. Early on during the learning process it's likely that the weights are badly wrong. And so it's best to use a large learning rate that causes the weights to change quickly. Later, we can reduce the learning rate as we make more fine-tuned adjustments to our weights. One idea is to hold the learning rate constant until the training cost starts to get worse at which point we reduce it by, say, a factor of 2, 4, 8, 16...maybe until 1024, reducing each time it gets worse.
    - Use early stopping to determine the number of training epochs. This makes setting the number of epochs very simple. In particular, it means that we don't need to worry about explicitly figuring out how the number of epochs depends on the other hyper-parameters. Instead, that's taken care of automatically. Furthermore, early stopping also automatically prevents us from overfitting.
    - I suggest starting initially with no regularization (λ=0), and determining a value for η, as above. Using that choice of η, we can then use the validation data to select a good value for λ. Start by trialling λ = 1.0, and then increase or decrease by factors of 10, as needed to improve performance on the validation data. Once you find a good order of magnitude, fine tune. That done, you should return and re-optimize η again.
    - Mini batch size: Remember that matrix libraries and parallel compute lets us compute weight updates for a mini batch faster than it takes to do it sequentially, although may not be much faster. Too small, and you don't get to take full advantage of the benefits of good matrix libraries optimized for fast hardware. Too large and you're simply not updating your weights often enough. What you need is to choose a compromise value which maximizes the speed of learning. We need our mini batch to be big enough to get a good sense of the general direction we must go in the hill.
    - A great deal of work has been done on automating the process of finding hyperparameters. A common technique is *grid search*, which systematically searches through a grid in hyper-parameter space.

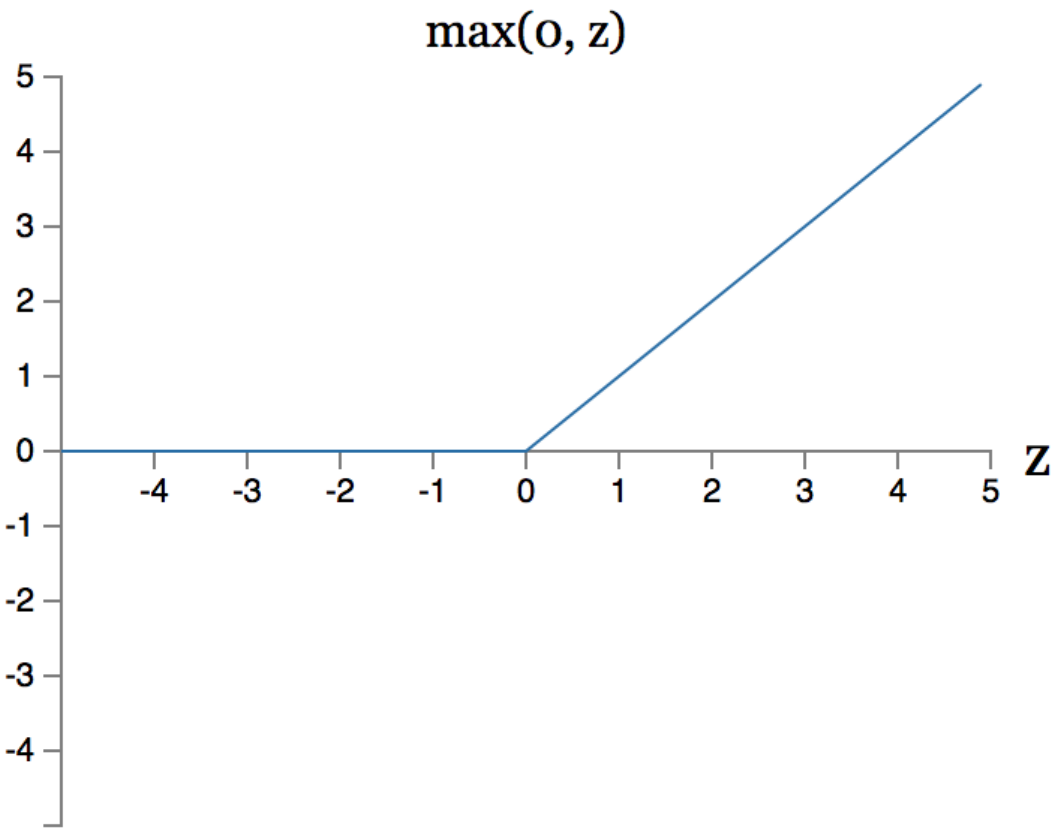
  - You should be on the lookout for signs that things aren't working, and be willing to experiment. In particular, this means carefully monitoring your network's behavior, especially the validation accuracy.
  - There is a saying common among writers that books are never finished, only abandoned. The same is also true of neural network optimization: the space of hyper-parameters is so large that one never really finishes optimizing, one only abandons the network to posterity. There's always another trick you can try to improve performance.
  - Setup your tools so as to get to a good state fast and then leave it or proceed further depending on how important that model is.
- An alternative to stochastic gradient descent is Newton's method (that we talked about in the cs 229 ML class notes). This method involves using Hessians. A big drawback of this method is that for neural nets the Hessian gets ridiculously large and then we have to compute its inverse! Not happening. But that line of thinking does inspire something called momentum based gradient descent.
- An alternative method that speeds up gradient descent is to use something called momentum based gradient descent. Here, instead of updating the weight as $w \to w' = w - \eta \nabla C$, we do:
  $v \to v' = \mu v - \eta \nabla C$
  $w \to w' = w + v'$
  which is like updating velocity instead of updating position.
- In the above, notice that if a few gradients are in almost the smallest direction, it speeds up in that direction. On the other hand if after a while, the direction changes, we don't immediately "jerkily" change direction. we go back slowly, so as to find the peak/trough and not overshoot it.
- μ should be between 0 and 1. Think about what can go wrong if its greater than 1 or less than 0. μ is called the momentum co-efficient.
- In practice, often neurons built with functions other than sigmoid work better in terms of speed of learning and generalization. one such is $tanh$ (hyperbolic tan or tanch) → if you don't remember this go back to the notes on transcendental functions from Herb Gross' calculus class notes.
- Note that tanch is just sigmoid, but scaled. $tanh(z) = 2\sigma(z) - 1$. Note that instead of $0, 1$ it takes values in $-1, 1$. According, we'll have to re-scale our outputs.
- $tanh$ vs σ:



- Why does $tanh$ perform better on some tasks? We don't know but here's a conjecture:

> "Suppose we're using sigmoid neurons, so all activations in our network are positive. Let's consider the weights w^{l+1}_{jk} input to the j^{th} neuron in the $l+1^{th}$ layer. The rules for backpropagation tell us that the associated gradient will be $a^l_k \delta^{l+1}_j$. Because the activations are positive the sign of this gradient will be the same as the sign of δ^{l+1}_j. What this means is that if δ^{l+1}_j is positive then *all* the weights w^{l+1}_{jk} will decrease during gradient descent, if its negative, then all will increase. In other words, all weights to the same neuron must either increase together or decrease together. That's a problem, since some of the weights may need to increase while others need to decrease. That can only happen if some of the input activations have different signs. That suggests replacing the sigmoid by an activation function, such as $tanh$, which allows both positive and negative activations."

- Again, notice that the above is a heuristic argument, far from a proof.
- Rectified linear units are when the neuron uses $max(0, w^T x + b)$ as each neuron.



- Some recent work on image recognition has found considerable benefit in using rectified linear units through much of the network. However, as with $tanh$ neurons, we do not yet have a really deep understanding of when, exactly, rectified linear units are preferable, nor why.
- recall that sigmoid neurons stop learning when they saturate, i.e., when their output is near either 0 or 1. $tanh$ neurons suffer from a similar problem when they saturate. By contrast, increasing the weighted input to a rectified linear unit will never cause it to saturate, and so there is no corresponding learning slowdown. On the other hand, when the weighted input to a rectified linear unit is negative, the gradient vanishes, and so the neuron stops learning entirely. So really, we just don't know ¯_(ツ)_/¯.

- "I've painted a picture of uncertainty here, stressing that we do not yet have a solid theory of how activation functions should be chosen." - Michael

> ***Question:*** *How do you approach utilizing and researching machine learning techniques that are supported almost entirely empirically, as opposed to mathematically? Also in what situations have you noticed some of these techniques fail?*
>
> **Answer:** You have to realize that our theoretical tools are very weak. Sometimes, we have good mathematical intuitions for why a particular technique should work. Sometimes our intuition ends up being wrong [...] The questions become: how well does my method work on this particular problem, and how large is the set of problems on which it works well.
>
> *- Question and answer with neural networks researcher Yann LeCun*

- Basically, there very often isn't great theoretical foundations for the techniques we study in neural networks. Very often, the "best" research papers you'll see that stories in a similar style appear in many research papers on neural nets, often with thin supporting evidence. It's a field still being explored. Should you reject these stories/arguments just because they don't have a solid theoretical grounding? No! It is what it is currently. Take it, critique it, think about it, choose some of them to do your own research to try to contribute theoretical foundations to the field. But realize that it is well acknowledged that there isn't theoretical backing, but it works and we should all investigate further and use it.