Moving beyond simple backpropagation

The backpropagation era eventually ended when researchers realized that the backpropagation algorithm by itself was not powerful enough to solve more difficult problems. Researchers realized that one requirement for solving more difficult problems was to use networks with more hidden layers. Unfortunately, simple backpropagation alone does not work well on such deep networks.

A series of small improvements

Over time, various researchers came up with a lengthy list of small improvements that made it easier for backpropagation to work well with deep networks. Some of these improvements included

- Normalization
- Better random weight initialization
- Better activation functions
- Better loss functions
- Improvements to gradient descent
- Specialized network architectures
- Transfer learning

Collectively, these small improvements helped usher in the current *deep learning* era.

These notes will introduce you to some of these improvements.

Normalization

All problems involve having a network take as its input a list of features. In some cases, the distinct features in the input vector will vary quite a bit in both their magnitude and their variability. This in turn causes problems for gradient descent, because it may force the network to assign larger weights to connections coming from inputs that have smaller values and smaller weights to connections coming from inputs that typical have larger values. This in turn has the effect of making the loss surface more complex, which in turn will make it harder for gradient descent to find a minimum.

The solution to these problems is to use a *normalization* process on the input data. The normalization process consists of two steps:

1. For each input feature compute the mean value μ and the standard deviation σ for that feature across each input in the training set.

2. Replace each feature x with $(x - \mu)/\sigma$

After normalization each feature will have an average value of 0. This in turn makes it easier to build a network that has no bias on any of its hidden units.

Beyond applying normalization to the input layer researchers eventually also realized that it can be useful to apply normalization deeper in the network as well. We will eventually see how to use a process called *batch normalization* to normalize the outputs of a hidden layer.

Better weight initialization

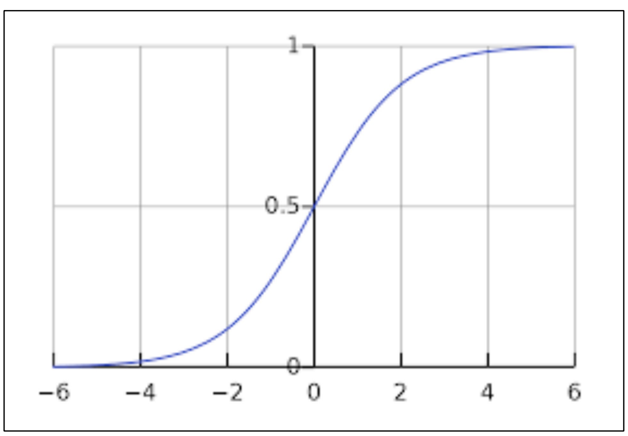
When we set up a neural network one of the things we have to do is to assign an initial set of weights to all of the connections. In the backpropagation era this was typically done by assigning weights randomly to all of the connections. Eventually researchers learned that this simple algorithm for weight initialization could cause problems for some network architectures. One problem that arose was that units in different hidden layers and sometimes even units within the same hidden layer would differ in the number of input connections they had. If one unit has more incoming connections than another and the weights to all of these connections are assigned randomly, the unit with more incoming connections would see summed inputs that had more extreme statistical properties.

Eventually researchers realized that they needed to scale the initial random weights by dividing the initial random weights by square root of the *fan in factor* for the unit the connection runs to. The fan in factor for a hidden unit is simply the count of how many connections come in to that unit.

Better activation functions

The backpropagation era introduced the widespread use of the sigmoid activation function.

$$a(x) = \frac{1}{1 + e^{-x}}$$



This activation function replaced the simple threshold activation function of the Perceptron era, since the latter is not differentiable.

Over time, researchers realized that the sigmoid activation function had a number of shortcomings:

- It is harder to compute, and has a complicated looking derivative. These problems made both forward propagation and backpropagation more expensive.
- It has a derivative that becomes quite small as you move away from 0. This led to what was known as the *vanishing gradients problem*, which caused backpropagation to stall in some scenarios.

To work around these problems researchers started to experiment with alternative activation functions. Over time, an alternative that emerged as particularly successful in practice was the *rectified linear activation function*, or ReLU for short:

ReLU has a number of advantages. It is non-linear, like sigmoid, but is significantly easier to compute with. Further, ReLU does not suffer from the vanishing gradients problem, as long as the unit's inputs pass the 0 threshold at which ReLU turns on.

Better loss functions

In the backpropagation era the most commonly used loss function was the mean squared error. To compute the loss we would feed a set of inputs to a network and generate a list of output vectors $\vec{o}$. The loss function would compare the components of the output vectors to a corresponding set of target vectors $\vec{t}$ that we wanted the network to produce:

$$L(\vec{w}) = \sum_{i=1}^{n} \frac{1}{k} \sum_{j=1}^{k} (o_{i,j} - t_{i,j})^2$$

Here $o_{i,j}$ is the jth component of the ith output vector.

Over time researchers began to discover that using the same loss measure for every network and every application was not optimal. Instead, it made more sense to match the loss function to the application.

A well-known example of this optimization occurs in networks that perform classification tasks. In such a network we seek to categorize inputs into one of k possible categories. To do this, we set up a network with k output units and interpret the output as selecting a particular category by selecting the output unit with the highest activation value and assigning the input to that category. For applications involving categorization researchers developed an alternative loss function, the *categorical cross entropy* loss function:

$$L(\vec{w}) = \sum_{i=1}^{n} \sum_{j=1}^{k} - t_{i,j} \log p_{i,j}$$

Here $p_{i,j}$ is the jth component of the ith output vector after the components of the output vector have been rescaled so that the sum of the elements of each output vector is 1. The target values $t_{i,j}$ indicate the correct category for the ith input by having a 1 in the position for the correct category and a 0 in all other positions.

Researchers learned that using a loss function that was more closely suited to the task at hand made backpropagation converge more quickly to a good set of weights.

Improvements to gradient descent

The gradient descent algorithm is the heart of network learning in backpropagation. One common problem that gradient descent suffers from is the problem of local minima in the loss surface. To keep gradient descent from descending into a shallow local minimum and getting stuck there, we can introduce the concept of *momentum* to the gradient descent process.

The original gradient descent algorithm uses an update rule that can be written as

$$\vec{V} = - \eta \frac{\partial L(\vec{W})}{\partial \vec{W}}$$

$$\vec{W} = \vec{W} + \vec{V}$$

Gradient descent with momentum updates these rules to

$$\vec{V} = \beta \vec{V} - \eta \frac{\partial L(\vec{W})}{\partial \vec{W}}$$

$$\vec{W} = \vec{W} + \vec{V}$$

The new term $\beta \vec{V}$ with $\beta < 1$ acts as a memory term to help us preserve at least some of the past history in the $\vec{V}$ term. This then encourages gradient descent to typically move past shallow local minima because the descent vector remembers some of the direction that lead us toward the mimimum, which in turn will cause gradient descent of slightly overshoot the minimum. For shallow minima this is sufficient to allow us to escape the influence of the shallow minimum.

Another problem that can occur in loss landscapes is that the landscape ends up being steeper in some directions than others. This leads to landscape features such as valleys with steep walls. As gradient descent works on descending down the valley the steep walls can cause us to bounce up and down the walls as we descend, wasting a lot of time along the way bounding up and down the walls. We can damp some of this movement out by scaling the partial derivative terms by a factor that retains a historical memory of the relative size of that particular partial derivative. The *AdaGrad algorithm* replaces the weight update rule in backpropagation with

$$A_i = A_i + \left(\frac{\partial L(\vec{W})}{\partial w_i}\right)^2$$

$$w_i = w_i - \frac{\eta}{\sqrt{A_i}} \frac{\partial L(\vec{W})}{\partial w_i}$$

A slight improvement to AdaGrad is the *RMSProp algorithm*, which uses a decay process to give the A_i term a decaying memory of past values of the partial derivative:

$$A_i = \rho A_i + (1 - \rho)\left(\frac{\partial L(\vec{W})}{\partial w_i}\right)^2$$

$$w_i = w_i - \frac{\eta}{\sqrt{A_i}} \frac{\partial L(\vec{W})}{\partial w_i}$$

Finally, the *Adam algorithm* mixes together the concepts of momemtum and the AdaGrad/RMSProp adjustments.

$$A_i = \rho A_i + (1 - \rho)\left(\frac{\partial L(\vec{W})}{\partial w_i}\right)^2$$

$$F_i = \rho_f F_i + (1 - \rho_f) \frac{\partial L(\vec{W})}{\partial w_i}$$

$$w_i = w_i - \frac{\eta}{\sqrt{A_i}} F_i$$

Specialized network architectures

As the field of neural networks expanded into different application areas researchers began to develop a host of specialized network architectures to address the needs of different problems. Over the remainder of this term we will be studying many of these specialized applications and we will meet new network architectures along the way.

The process really kicked into high gear in the deep learning era. As researchers realized the benefits of building deeper networks with more layers, they began to develop ever more elaborate specialized architectures for these deeper networks.

Transfer learning

Another widely used strategy in the deep learning era involves training a network to solve one problem and then transferring part of that network over to a new network that is meant to solve a closely related problem. The idea is that if we have already learned a set of weights for part of the network, those pretrained weights can save us from having to relearn the weights for that part of the new network.

We will first see this strategy being employed successfully later on in the section on computer vision applications.