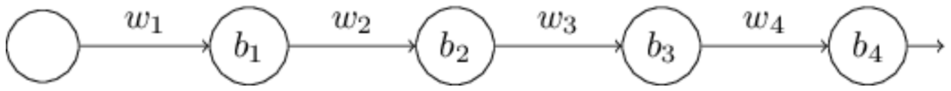
Tuning Neural Networks with Normalization

Understanding the Problem of Unstable Gradients

Vanishing Gradient Problem

The **vanishing gradient problem** describes a situation, encountered while training neural networks, in which the gradients used to update the weights shrink exponentially. As a consequence, the weights essentially stop updating, and learning stalls.

To get insight into why the vanishing gradient problem occurs, consider a 4–layer neural network with just a single neuron in each layer, as shown below:



Imagine we make a small change Δw_1 in the weight parameter w_1. That will set off a cascading series of changes in the subsequent layers of the network. This cascade propogates all the way through to a change ΔJ in the cost at the output of the final neuron. We can approximate the gradient of the cost function with respect to w_1 as:

$$\frac{\partial J}{\partial w_1} \approx \frac{\Delta J}{\Delta w_1}$$

The approximation above suggests that an expression for the gradient $\frac{\partial J}{\partial w_1}$ can be deduced by carefully tracking the effect of each step in this cascade.

a) The change Δw_1 causes a change Δa_1 in the output a_1 **from the first hidden neuron**:

$$a_1 = g_1(z_1) = g_1(w_1 a_0 + b_1)$$

$$\implies \Delta a_1 \approx \frac{\partial a_1}{\partial w_1}\Delta w_1 = \frac{\partial g_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}\Delta w_1 = g_1'(z_1)a_0\Delta w_1$$

b) The change Δa_1 causes a change Δz_2 in the weighted input z_2 **to the second hidden neuron**:

$$z_2 = w_2 a_1 + b_2$$

$$\implies \Delta z_2 \approx \frac{\partial z_2}{\partial a_1}\Delta a_1 = w_2 \Delta a_1$$

c) Combining the expressions for Δz_2 and Δa_1, we obtain a new expression describing how w_1 **propagates along the network to affect** z_2:

> $$\Delta z_2 \approx w_2 \Delta a_1 = g_1'(z_1) w_2 a_0 \Delta w_1$$

d) The change Δz_2 causes a change Δa_2 in the output a_2 **from the second hidden neuron**:

$$a_2 = g_2(z_2) = g_2(w_2 a_1 + b_2) = g_2\big(w_2 * g_1(w_1 a_0 + b_1) + b_2\big)$$

$$\implies \Delta a_2 \approx \frac{\partial a_2}{\partial w_1} \Delta w_1 = \frac{\partial g_2}{\partial z_2} \frac{\Delta z_2}{\Delta w_1} \Delta w_1 = \frac{\partial g_2}{\partial z_2} \Delta z_2 = g_2'(z_2) g_1'(z_1) w_2 a_0 \Delta w_1$$

e) The change Δa_2 causes a change Δz_3 in the weighted input z_3 **to the third hidden neuron**:

$$z_3 = w_3 a_2 + b_3$$

$$\implies \Delta z_3 \approx \frac{\partial z_3}{\partial a_2} \Delta a_2 = w_3 \Delta a_2$$

f) Combining the expressions for Δz_3 and Δa_2, we obtain a new expression describing how w_1 **propagates along the network to affect** z_3:

> $$\Delta z_3 \approx w_3 \Delta a_2 = g_2'(z_2) g_1'(z_1) w_3 w_2 a_0 \Delta w_1$$

g) The change Δz_3 causes a change Δa_3 in the output a_3 **from the third hidden neuron**:

$$a_3 = g_3(z_3) = g_3(w_3 a_2 + b_3) = g_3\Big(w_3 * g_2\big(w_2 * g_1(w_1 a_0 + b_1) a_1 + b_2\big) a_2 + b_3\Big)$$

$$\implies \Delta a_3 \approx \frac{\partial a_3}{\partial w_1} \Delta w_1 = \frac{\partial g_3}{\partial z_3} \frac{\Delta z_3}{\Delta w_1} \Delta w_1 = \frac{\partial g_3}{\partial z_3} \Delta z_3 = g_3'(z_3) g_2'(z_2) g_1'(z_1) w_3 w_2 a_0 \Delta w_1$$

h) The change Δa_3 causes a change Δz_4 in the weighted input z_4 **to the final hidden neuron**:

$$z_4 = w_4 a_3 + b_4$$

$$\implies \Delta z_4 \approx \frac{\partial z_4}{\partial a_3} \Delta a_3 = w_4 \Delta a_3$$

i) Combining the expressions for Δz_4 and Δa_3, we obtain a new expression describing how w_2 **propagates along the network to affect** z_3:

> $$\Delta z_4 \approx w_4 \Delta a_3 = g_3'(z_3) g_2'(z_2) g_1'(z_1) w_4 w_3 w_2 a_0 \Delta w_1$$

j) The change Δz_4 causes a change Δa_4 in the output a_4 **from the final hidden neuron**:

$$a_4 = g_4(z_4) = g_4(w_4 a_3 + b_4) = g_4\Bigg(w_4 * g_3\Big(w_3 * g_2\big(w_2 * g_1(w_1 a_0 + b_1) a_1 + b_2\big) a_2 + b_3\Big) a_3 + b_4\Bigg)$$

$$\implies \Delta a_4 \approx \frac{\partial a_4}{\partial w_1} \Delta w_1 = \frac{\partial g_4}{\partial z_4} \frac{\Delta z_4}{\Delta w_1} \Delta w_1 = \frac{\partial g_4}{\partial z_4} \Delta z_4 = g_4'(z_4) g_3'(z_3) g_2'(z_2) g_1'(z_1) w_4 w_3 w_2 a_0 \Delta w_1$$

k) The change Δa_4 causes a change ΔJ in the cost function (J):

$$\Delta J \approx \frac{\partial J}{\partial a_4}\Delta a_4 = \frac{\partial J}{\partial a_4} g_4'(z_4) g_3'(z_3) g_2'(z_2) g_1'(z_1) w_4 w_3 w_2 a_0 \Delta w_1$$

> $$\implies \frac{\partial J}{\partial w_1} \approx \frac{\Delta J}{\Delta w_1} = \frac{\partial J}{\partial a_4} g_4'(z_4) g_3'(z_3) g_2'(z_2) g_1'(z_1) w_4 w_3 w_2 a_0$$

To understand why the vanishing gradient problem occurs, let us write the above result in a more general form::

> $$\frac{\partial J}{\partial w_l} = a_{l-1} * w_l g_l'(z_l) * w_{l+1} g_{l+1}'(z_{l+1}) * \cdots * w_{L-1} g_{L-1}'(z_{L-1}) * w_L g_L'(z_L) * \frac{\partial J}{\partial \hat{y}}$$

All derivatives of the activation functions used in neural networks reach a maximum at $g'(0) = 1$ [except the sigmoid function, the derivative of which reaches a maximum at $\sigma'(0) = \frac{1}{4}$]. Therefore, it is usually the case that $|g_l'(z_l)| < 1$ is satisfied. If the standard approach is taken to initialize the weights in the network, then we will choose weight vaues from a Gaussian distribution with ($\mu = 0$, $\sigma = 1$). Therefore, it is often the case that $|w_l| < 1$ is satisfied as well. It follows that:

$$|w_j g_j'(z_j)| < 1, \quad 1 < j < L$$

is a very common occurance. Based on the generalized result above, $\frac{\partial J}{\partial w_j}$ vanishes exponentially as the gradient calculation is propagated from $j = L$ to $j = 1$.

Exploding Gradient Problem

The **exploding gradient problem** describes a situation, encountered while training neural networks, in which the gradients used to update the weights grow exponentially. This prevents the backpropagation algorithm from making reasonable updates to the weights, and learning becomes unstable.

There are two steps to conditions that need to be met in order to induce an exploding gradient. First, all the weights in the network need to be large, for example $w_1 = w_2 = w_3 = w_4 = 100$. Second, the biases are chosen such that the $g_j'(z_j)$ terms are not too small. This is actually trivial – simply choose the biases that ensure the weighted input to each neuron is zero ($z_j = 0$), since this leads to the maximum gradient of the activation function g_j. For instance, to ensure that $z_1 = w_1 a_0 + b_1 = 0$ one would set $b_1 = -100 a_0$. When these two conditions are met, we would have $|g_j'(z_j)| > 1$. Based on the generalized result above, $\frac{\partial J}{\partial w_j}$ increases exponentially as the gradient calculation is propagated from $j = L$ to $j = 1$.

Prevalence of the Vanishing Gradient Problem

In order to avoid the vanishing gradient problem, we require $|w_j g_j'(z_j)| \geq 1$. It is misleading to think this could happen easily if w is very large, because the $g_j'(z_j)$ term also depends on w, since $z_j = w_j a_{j-1} + b_j$, that is, when making w large, we need to be careful that we're not simultaneously making $g_j'(w_j a_{j-1} + b_j)$ small. These conditions are rare enough that they actually turn out to be a considerable constraint. It is far more likely that, when we make w large, we are also making $z_j = w_j a_{j-1} + b_1$ large. **Therefore, vanishing gradients are much more common than exploding gradients.**

Normalization in Deep Learning

Before training a neural network, the data should be normalized by performing some kind of feature scaling, as this can dramatically improve the training process. **Normalization**, in the context of deep learning, refers to the practice of transforming the input data such that all features are on a similar scale, usually ranging from 0 to 1.

Advantages of Normalizing the Input Data

Feature Scaling

The first reason, quite evident, is that for a dataset with multiple inputs we'll generally have different scales for each of the features. We can make the same considerations for datasets with multiple targets. This situation could give rise to greater influence in the final results for some of the inputs, with an imbalance not due to the intrinsic nature of the data but simply to their original measurement scales. Normalizing all features in the same range avoids this type of problem.

Mitigation of the Vanishing Gradient Problem

Another reason for input normalization is related to the gradient problem we mentioned in the previous section. The rescaling of the input within small ranges gives rise to even small weight values in general, and this makes the output of the units of the network near the saturation regions of the activation functions less likely. Furthermore, it allows us to set the initial range of variability of the weights in very narrow intervals, typically [-1: 1].

One way to speed up training of your neural networks is to normalize the input. In fact, even if training time were not a concern, normalization to a consistent scale (typically 0 to 1) across features should be used to ensure that the process converges to a stable solution. Similar to some of our previous work in training models, one general process for standardizing our data is subtracting the mean and dividing by the standard deviation.

Optimizers

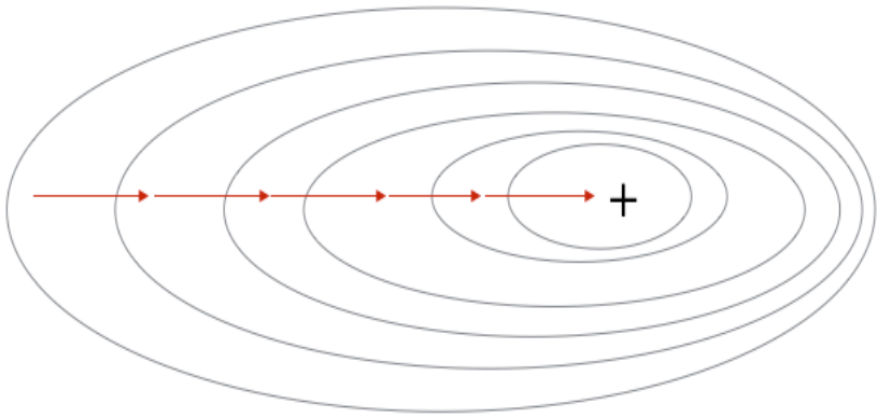
Optimizers are algorithms that adjust the parameters of a neural network in order to reduce the loss. In addition to gradient descent, alternative optimizers exist. One issue with gradient descent is that it oscillates to a fairly big extent, because the derivative is bigger in the vertical direction.

Batch Gradient Descent

Standard gradient descent (also called **batch gradient descent**) is an optimization algorithm that iteratively updates a model's parameters in order to minimize the cost function. Batch gradient descent evaluates the cost function over the full training set, and using the gradient of this cost function to update the model parameters. Therefore, each epoch consists of 1 parameter update requiring m iterations, where m is the number of observations in the training set. The parameter update equation for batch gradient descent is:

$$\theta := \theta - \alpha \cdot \nabla_\theta J(\theta; X, y)$$

Batch gradient descent uses the entire dataset to perform just a single update. Therefore it requires a large amount of memory because the entire dataset needs to be loaded into memory for each iteration of the algorithm. Additionally, if the dataset is very large, than convergence will take a long time.

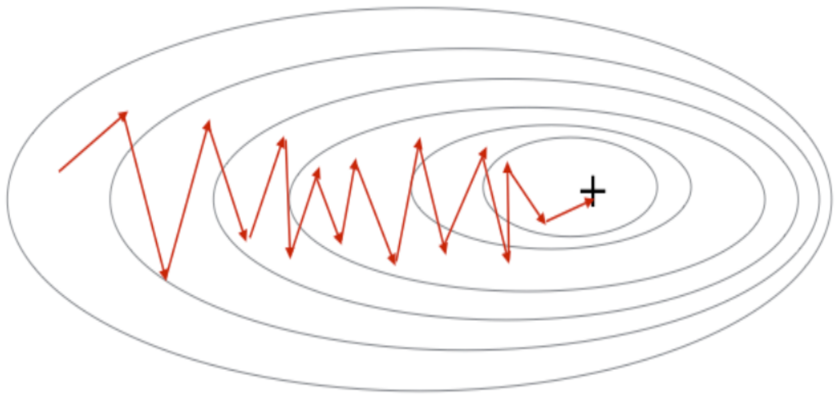


Stochastic Gradient Descent (SGD)

Stochastic gradient descent is a variant of gradient descent that chooses a random individual observation $x^{(i)}$, evaluates the loss of this observation with respect to its label $y^{(i)}$, and using the gradient of this loss to update the model parameters. Therefore, each epoch consists of m parameter updates, where m is the number of observations in the training set. The parameter update equation for stochastic gradient descent is:

$$\theta := \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i)}, y^{(i)}), \quad i = 1, \ldots, m$$

Since stochastic gradient descent updates parameters much more frequently than batch gradient descent, it takes much longer to complete a single training epoch, and thus takes longer to converge. However, it requires far less memory because only a single observation needs to be loaded into memory for each iteration of this algorithm.

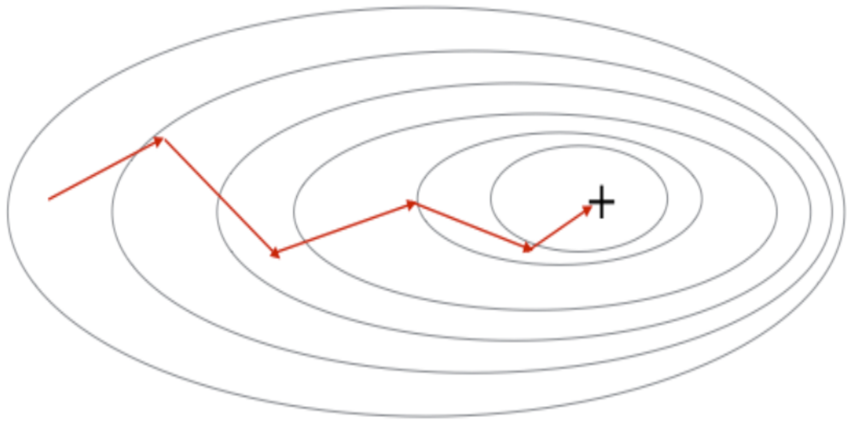


Mini-Batch Gradient Descent

Mini-batch gradient descent is a variant of gradient descent that chooses a random subset of observations $X_B^{(i)}$, evaluates the cost function of this observation with respect to their labels $y_B^{(i)}$, and using the gradient of this cost function to update the model parameters. Therefore, each epoch consists of $\frac{m}{B}$ parameter updates, each requiring B iterations (where m is the number of observations in the training set and B is the size of each mini-batch). The parameter update equation for mini-batch gradient descent is:

$$\theta := \theta - \alpha \cdot \nabla_\theta J(\theta; X_B^{(i)}, y_B^{(i)}), \quad i = 1, \ldots, \frac{m}{B}$$

Mini-batch gradient descent updates the model parameters less frequently than stochastic gradient descent, but more frequently than batch gradient descent. Its time complexity is therefore in between these two. Likewise, it requires less memory than batch gradient descent, but more memory than stochastic gradient descent. It therefore also has a memory complexity in between these two.



Gradient Descent with Momentum

Gradient descent with momentum accelerates the convergence towards the relevant direction and reduces the fluctuation in the irrelevant direction. It does this by adding a fraction (β) of the exponentially weighted average of past gradients ($V_{d\theta}$) to the gradient at the current time step ($d\theta$).

That is, the direction of the previous update is retained to a certain extent during the current update. This has the efffect of dampening oscillations, thereby improving convergence.

> $$V_{d\theta} := \beta V_{d\theta} + (1-\beta)d\theta$$
>
> $$\theta := \theta - \alpha V_{d\theta}$$

The momentum term β is an additional hyperparameter that can be tuned, however, setting a value of $\beta = 0.9$ is generally sufficient.

Root Mean Square Propagation (RMS Propagation)

RMS propagation accelerates the convergence towards the relevant direction by including a dynamic learning rate (α'). It does this by taking a fraction (β) of the exponentially weighted average of the squared past gradients ($S_{d\theta}$) and using this to calculate the dynamic learning rate from the initial learning rate (α).

> $$S_{d\theta} := \beta S_{d\theta} + (1-\beta)d\theta^2$$
>
> $$\alpha' = \frac{\alpha}{\sqrt{S_{d\theta} + \epsilon}}$$
>
> $$\theta := \theta - \alpha' d\theta$$

In the direction where we want to learn fast, the corresponding S will be small, so dividing by a small number. On the other hand, in the direction where we will want to learn slow, the corresponding S will be relatively large, and updates will be smaller.

Adaptive Moment (ADAM) Optimizer

The **ADAM optimizer** optimizer is a combination of RMS propagation and gradient descent with momentum. It updates the parameters based on the dynamic learning rate from RMS propagation (α') and dampens oscillations during convergence by incorporating the exponential moving average of past gradients ($V_{d\theta}$) and the exponential moving average of past squared gradients ($S_{d\theta}$).

> $$V_{d\theta} := \beta V_{d\theta} + (1-\beta)d\theta$$
>
> $$S_{d\theta} := \beta S_{d\theta} + (1-\beta)d\theta^2$$
>
> $$\alpha' = \frac{\alpha}{\sqrt{S_{d\theta} + \epsilon}}$$
>
> $$\theta := \theta - \alpha' V_{d\theta}$$

Generally, only α' is tuned. Typical values for the hyperparameters of the ADAM optimizer include:

- $\beta_1 = 0.9$
- $\beta_2 = 0.999$

- $\epsilon = 10^{-8}$

Hyperparameter Tuning

Below is a list of all hyperparameters that need tuning along with their level of importance:

- α [high importance]
- β (momentum) [medium importance]
- number of hidden units [medium importance]
- mini-batch size [medium importance]
- number of layers [low importance]
- learning rate decay [low importance]
- β_1, β_2, ϵ (Adam) [almost never tuned]