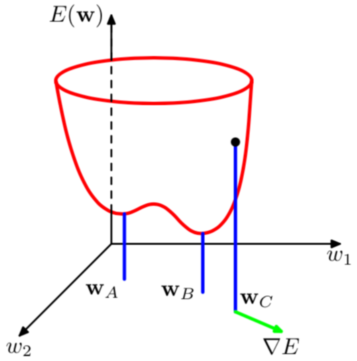
Neural Networks: Backpropagation[1]

For a single perceptron, computing the partial derivatives was trivial. **For a multilevel network with multiple neurons per layer, it can be hairy. This is where backprop comes to the rescue. It is a simple and efficient way to compute partial derivatives with respect to weights in a neural network.**

Similar to perceptron, the **gradient descent method** is used for getting to a local minimum of a smooth training error surface.

$\underline{w} = \{ \underline{W}^{(1)}, \underline{w}^{(2)}, \ldots, W^{(L)} \}$



The gradient descent method is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, **one takes steps proportional to the negative of the gradient**. We initialize the weights to $\mathbf{w}(0)$ and for $t = 1, 2, \ldots$ update the weights by taking a step in the negative gradient direction,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla E\big(\mathbf{w}(t)\big)$$
$$= \mathbf{w}(t) - \eta \cdot \frac{\partial E}{\partial \mathbf{w}}$$

where $0 < \eta < 1$ is a constant that defines the learning rate. We call this (batch) gradient decent[2]. To implement gradient descent, we need the gradient.

Let us consider the sigmoidal multi-layer neural network with $\theta(x) = \tanh(x)$. Since $h(\mathbf{x})$ is smooth, we can apply gradient descent to the resulting error function. To do so, we need the gradient $\nabla E(\mathbf{w})$. Recall that the weight vector $\mathbf{w}$ contains all the weight matrices $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L)}$, and we need the derivatives with respect to all these weights. For the weights $\mathbf{W}^{(l)}$ for $l = 1, 2, \ldots, L$:

[1] Major contents are adapted from dynamic e-chapters of Abu-Mostafa, Y S, Magdon-Ismail, M., Lin, H-T (2012) *Learning from Data*, AMLbook.com.

[2] In batch gradient decent, the gradient is computed for the error on **the whole data set** before a weight update is done.

$$\begin{aligned}\mathbf{W}^{(l)}(t+1) &= \mathbf{W}^{(l)}(t) - \eta \cdot \nabla E\big(\mathbf{W}^{(l)}\big) \\ &= \mathbf{W}^{(l)}(t) - \eta \cdot \frac{\partial E}{\partial \mathbf{W}^{(l)}}\end{aligned}$$

The normalized total error is the sum of the point-wise errors over the N data points $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_N, y_N)$ normalized by N:

$$E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathrm{e}_n$$

where a pointwise error $\mathrm{e}_n = \mathrm{e}(h(\mathbf{x}_n), y_n)$. For the squared error $\mathrm{e} = (h(\mathbf{x}) - y)^2$. To compute the gradient of E, we need its derivative with respective to each weight matrix:

$$\frac{\partial E}{\partial \mathbf{W}^{(l)}} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial \mathrm{e}_n}{\partial \mathbf{W}^{(l)}}$$

Remarks:

1. The basic building block in the above equation is the partial derivative of error on **a data point** e with respect to the $\mathbf{W}^{(l)}$. We now derive an elegant algorithm know as *backpropagation* to compute these quantities efficiently.
2. We describe backpropagation to get the partial derivative of the squared error e, but the algorithm is general enough to get the partial derivatives of any error function e_n with respect to the weights.

Backpropagation is based on several applications of the chain rule to write partial derivatives in layer l using partial derivatives in layer $(l+1)$. To describe the algorithm, we define the **sensitivity vector** for layer l, which is the sensitivity (gradient) of the error e with respect to the input signal $\mathbf{s}^{(l)}$ that goes into layer l. We denote the sensitivity by $\boldsymbol{\delta}^{(l)}$,

$$\boxed{\boldsymbol{\delta}^{(l)}} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(l)}}$$

$$\frac{\partial e}{\partial \underline{W}^{(l)}}$$

The sensitivity quantifies how the error e changes with $\mathbf{s}^{(l)}$. Using the sensitivity, we can write the partial derivatives with respect to the weights as:

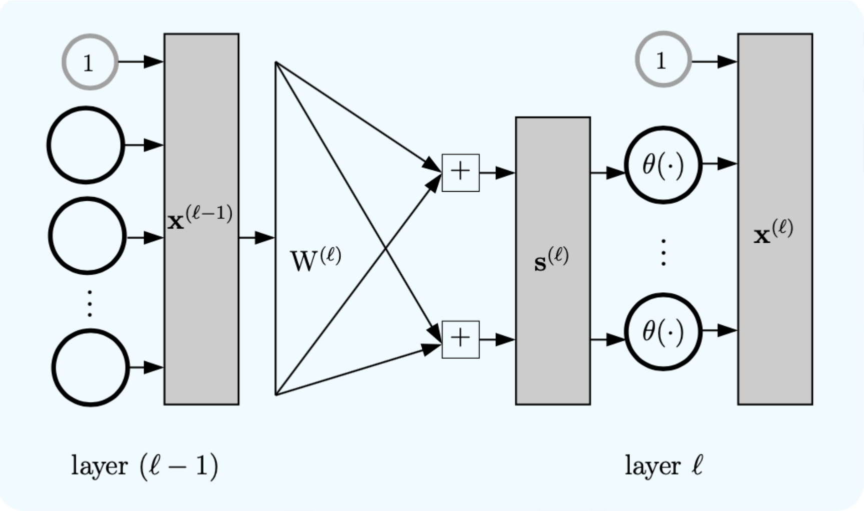
$W^{(\ell)} \quad (d^{(\ell-1)}+1) \times d^{(\ell)}$

***

(1) $$\boxed{\frac{\partial \mathrm{e}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)}\left(\boldsymbol{\delta}^{(l)}\right)^{\mathrm{T}}}$$

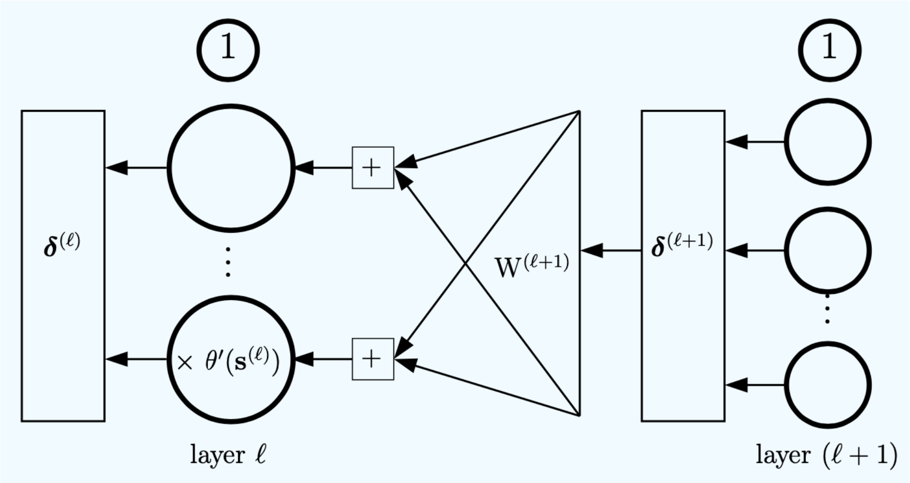
1. We will derive the equation later but for now let's examine it closely. The partial derivatives on the left form a weight matrix with dimensions $(d^{(l-1)} + 1) \times d^{(l)}$ and the 'outer product' of the two vectors on the right give exactly such a matrix.

2. The partial derivatives $\frac{\partial \mathrm{e}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)}\left(\boldsymbol{\delta}^{(l)}\right)^{\mathrm{T}}$ have contributions from two components. (i) $\mathbf{x}^{(l-1)}$: the output vector of the layer from which the weights originate; the larger the output, the more sensitive e is to the weights in the layer. (ii) $\boldsymbol{\delta}^{(l)}$: the sensitivity vector of the layer into which the weights go; the larger the sensitivity vector, the more sensitive e is to the weights in that layer.



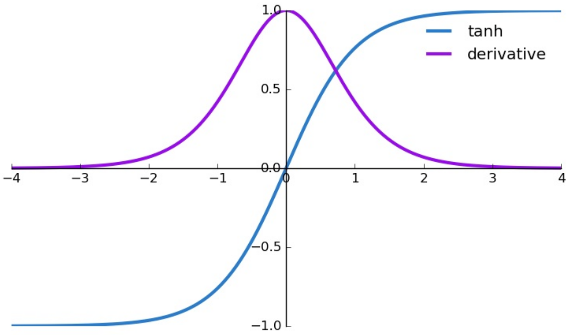
The outputs $\mathbf{x}^{(l)}$ for every layer $l \geq 0$ can be computed by a forward propagation. So to get the partial derivatives, it suffices to obtain the sensitivity vectors $\boldsymbol{\delta}^{(l)}$ for every layer $l \geq 1$ (remember that there is no input signal $\mathbf{s}^{(0)}$).

It turns out that the sensitivity vectors can be obtained by running a slightly modified version of the neural network backwards, and hence the name backpropagation. In forward propagation, each layer outputs the vector $\mathbf{x}^{(l)}$ and in backpropagation, each layer outputs (backwards) the vector $\boldsymbol{\delta}^{(l)}$. In forward propagation, we compute $\mathbf{x}^{(l)}$ from $\mathbf{x}^{(l-1)}$ and in backpropagation, we compute $\boldsymbol{\delta}^{(l)}$ from $\boldsymbol{\delta}^{(l+1)}$. The basic idea is illustrated in the following figure:



As you can see in the figure, the neural network is slightly modified only in that we have changed the activation function for the nodes. In forward propagation, the activation was sigmoid $\theta(\cdot)$. In backpropagation, the activation is multiplication by $\underline{\theta'}(\mathbf{s}^{(l)})$, where $\mathbf{s}^{(l)}$ is the input to the node. So the activation function is now different for each node, and it depends on the input to the node, which depends on $\mathbf{x}$. This input was computed already in the forward propagation.

Example: For the hyperbolic tangent $\tanh(\cdot)$ activation function, the derivative of $\tanh(x) = 1 - \tanh^2(x)$

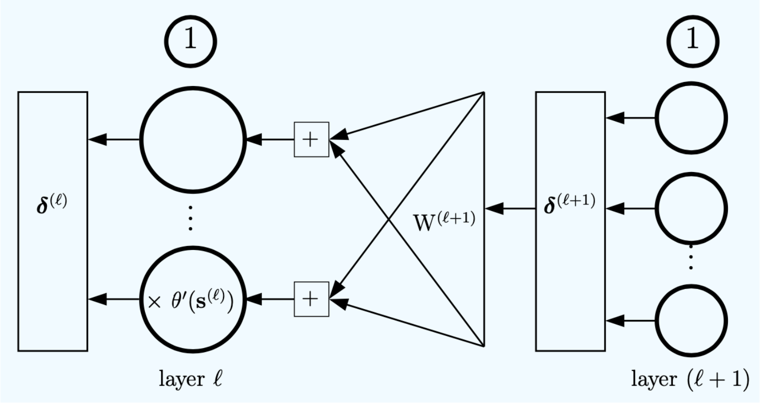


$$\xrightarrow{s^{(\ell)}} \boxed{\theta} \xrightarrow{x^{(\ell)}}$$

$$\tanh(\underline{s}^{(\ell)}) \to \underset{\sim}{x}^{(\ell)}$$

Thus, $\underline{\tanh'(\mathbf{s}^{(l)})} = 1 - \tanh^2(\mathbf{s}^{(l)}) = 1 - \underset{\sim}{\mathbf{x}}^{(l)} \otimes \underset{\sim}{\mathbf{x}}^{(l)}$, where $\otimes$ denotes component-wise multiplication.

$$\begin{bmatrix}1\\2\\3\end{bmatrix} \otimes \begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}1\\4\\9\end{bmatrix}$$



In the figure, layer $(l+1)$ outputs (backwards) the sensitivity vector $\boldsymbol{\delta}^{(l+1)}$, which gets multiplied by the weights $\mathbf{W}^{(l+1)}$, summed and passed into the node in layer l multiply by $\theta'(\mathbf{s}^{(l)})$ to get $\boldsymbol{\delta}^{(l)}$. Using component-wise multiplication $\otimes$, a shorthand notation for this backpropagation step is:

***

$$\boxed{\boldsymbol{\delta}^{(l)} = \theta'(\mathbf{s}^{(l)}) \otimes \left[\mathbf{W}^{(l+1)}\boldsymbol{\delta}^{(l+1)}\right]_1^{d^{(l)}}} \tag{2}$$

where the vector $\left[\mathbf{W}^{(l+1)}\boldsymbol{\delta}^{(l+1)}\right]_1^{d^{(l)}}$ contains components of $1, 2, \ldots, d^{(l)}$ of the vector $\mathbf{W}^{(l+1)}\boldsymbol{\delta}^{(l+1)}$ (excluding the bias component which has index 0).

Remark: We will derive the equation later but for now let's examine it closely. This formula $\boldsymbol{\delta}^{(l)} = \theta'(\mathbf{s}^{(l)}) \otimes \left[\mathbf{W}^{(l+1)}\boldsymbol{\delta}^{(l+1)}\right]_1^{d^{(l)}}$ is not surprising. The sensitivity $\boldsymbol{\delta}^{(l)}$ of e to inputs of layer l is proportional to:

(i) $\underline{\theta'(\mathbf{s}^{(l)})}$: the slope of the activation function in layer l (bigger slope means a small change in $\mathbf{s}^{(l)}$ will have a larger effect on $\mathbf{x}^{(l)}$),

(ii) $\underline{\mathbf{W}^{(l+1)}}$: the size of the weights going out of the layer (bigger weights mean a small change in $\mathbf{s}^{(l)}$ will have more impact on $\mathbf{s}^{(l+1)}$) and

(iii) $\underline{\boldsymbol{\delta}^{(l+1)}}$: the sensitivity in the next layer (a change in layer l affects the inputs to layer $l+1$, so if e is more sensitive to layer $l+1$, then it will also be more sensitive to layer l).

We now observe an important fact: if we know $\boldsymbol{\delta}^{(l+1)}$, then we can get $\boldsymbol{\delta}^{(l)}$. We thus use $\boldsymbol{\delta}^{(L)}$ to seed the backward process. We can get $\boldsymbol{\delta}^{(L)}$ explicitly because $\mathrm{e} = \left(\mathbf{x}^{(L)} - y\right)^2 = \left(\theta(\mathbf{s}^{(L)}) - y\right)^2$. Therefore,

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(L)}} = \frac{\partial}{\partial \mathbf{s}^{(L)}}\left(\mathbf{x}^{(L)} - y\right)^2 = 2\left(\mathbf{x}^{(L)} - y\right)\frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} = 2\left(\mathbf{x}^{(L)} - y\right)\theta'\left(\mathbf{s}^{(L)}\right)$$

With $\boldsymbol{\delta}^{(L)}$, we can compute all the sensitivities:

$$\boldsymbol{\delta}^{(1)} \longleftarrow \boldsymbol{\delta}^{(2)} \cdots \longleftarrow \underline{\boldsymbol{\delta}^{(L-1)}} \overset{(2)}{\longleftarrow} \underline{\boldsymbol{\delta}^{(L)}}.$$

Remark: since there is only one output node, $\mathbf{s}^{(L)}$ is a scalar, and so too is $\boldsymbol{\delta}^{(L)}$.

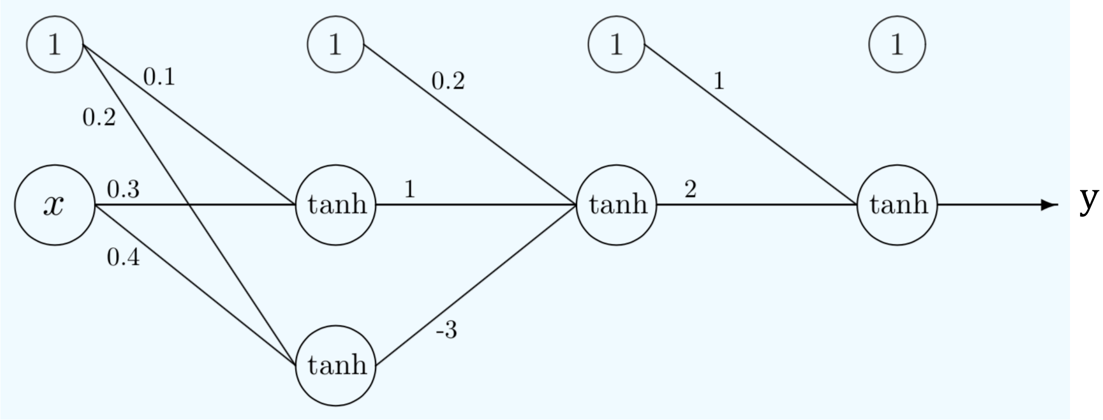
The algorithm box below summarizes backpropagation

```
Backpropagation to compute sensitivities $\delta^{(\ell)}$.
Input: a data point $(\mathbf{x}, y)$.
0: Run forward propagation on $\mathbf{x}$ to compute and save:
       $\mathbf{s}^{(\ell)}$   for $\ell = 1, \ldots, L$;
       $\mathbf{x}^{(\ell)}$   for $\ell = 0, \ldots, L$.
1: $\delta^{(L)} \leftarrow 2(x^{(L)} - y)\theta'(s^{(L)})$                [Initialization]
       $\theta'(s^{(L)}) = \begin{cases} 1 - (x^{(L)})^2 & \theta(s) = \tanh(s); \\ 1 & \theta(s) = s. \end{cases}$
2: for $\ell = L - 1$ to $1$ do                                  [Back-Propagation]
3:     Let $\theta'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$.
4:     Compute the sensitivity $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$:
           $\underline{\boldsymbol{\delta}^{(\ell)}} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$
```

(1) $\frac{\partial e}{\underline{\partial W^{(\ell)}}}$

Remark: In step 3, we assumed tanh-hidden node activation. If the hidden unit activation functions are not $\tanh(\cdot)$, then the derivative in step 3 should be updated accordingly.

Example: Let us consider a very simple network with a single data point where the input $x = 2$ and output $y = 1$.



There is a single input, and the weight matrices are:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix},\ \mathbf{W}^{(2)} = \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix},\ \mathbf{W}^{(3)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Forward propagation:

$\mathbf{x}^{(0)}$	$\mathbf{s}^{(1)}$	$\mathbf{x}^{(1)}$	$\mathbf{s}^{(2)}$	$\mathbf{x}^{(2)}$	$\mathbf{s}^{(3)}$	$\mathbf{x}^{(3)}$
$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}^T \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0.6 \\ 0.76 \end{bmatrix}$	$[-1.48]$	$\begin{bmatrix} 1 \\ -0.90 \end{bmatrix}$	$[-0.8]$	-0.66

We show above how $\mathbf{s}^{(1)} = \left(\mathbf{W}^{(1)}\right)^{\mathrm{T}} \mathbf{x}^{(0)}$ is computed.

Backpropagation:

We start from the output layer:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(L)}} = \frac{\partial}{\partial \mathbf{s}^{(L)}} \left(\mathbf{x}^{(L)} - y\right)^2 = 2\left(\mathbf{x}^{(L)} - y\right) \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} = 2\left(\mathbf{x}^{(L)} - y\right) \theta'\left(\mathbf{s}^{(L)}\right)$$

$$\boldsymbol{\delta}^{(3)} = [2 \cdot (-0.66 - 1) \cdot (1 - (-0.66)\text{^}2)] = [-1.855]$$

And recall

$$\boldsymbol{\delta}^{(l)} = \theta'\left(\mathbf{s}^{(l)}\right) \otimes \left[\mathbf{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)}\right]_1^{d^{(l)}}$$

$\boldsymbol{\delta}^{(3)}$	$\boldsymbol{\delta}^{(2)}$	$\boldsymbol{\delta}^{(1)}$
$[-1.855]$	$[(1-0.9^2)\cdot 2\cdot(-1.855)]=[-0.69]$	$\begin{bmatrix}-0.44\\ 0.88\end{bmatrix}$

↑ 請自己做

We have explicitly shown how $\boldsymbol{\delta}^{(2)}$ is obtained from $\boldsymbol{\delta}^{(3)}$.

It is now simple matter to combine the output vectors $\mathbf{x}^{(l)}$ with sensitivity vector $\boldsymbol{\delta}^{(l)}$ to obtain the partial derivatives that are needed for the gradient:

(1)

$$\frac{\partial \mathrm{e}}{\partial \mathbf{W}^{(1)}}=\mathbf{x}^{(0)}\left(\boldsymbol{\delta}^{(1)}\right)^T=\begin{bmatrix}-0.44 & 0.88\\ -0.88 & 1.75\end{bmatrix}$$

$$\frac{\partial \mathrm{e}}{\partial \mathbf{W}^{(2)}}=\mathbf{x}^{(1)}\left(\boldsymbol{\delta}^{(2)}\right)^T=\begin{bmatrix}-0.69\\ -0.42\\ -0.53\end{bmatrix}$$

$$\frac{\partial \mathrm{e}}{\partial \mathbf{W}^{(3)}}=\mathbf{x}^{(2)}\left(\boldsymbol{\delta}^{(3)}\right)^T=\begin{bmatrix}-1.85\\ 1.67\end{bmatrix}$$

We can continue the exercise and work out all the *N* data points $(\mathbf{x}_1, y_1),(\mathbf{x}_2, y_2),\cdots,(\mathbf{x}_N, y_N)$. For the batch gradient descent method, we sum up all the pointwise partial derivatives with respect to the weights:

$$\frac{\partial E}{\partial \mathbf{W}^{(l)}}=\frac{1}{N}\sum_{n=1}^{N}\frac{\partial \mathrm{e}_n}{\partial \mathbf{W}^{(l)}}$$

And update the weights for each layer:

$$\mathbf{W}^{(l)}(t+1)=\mathbf{W}^{(l)}(t)-\eta\cdot\frac{\partial E}{\partial \mathbf{W}^{(l)}}$$

Remarks: Mini-Batch Gradient Descent

1. Recall the above updating scheme is batch gradient decent (or true gradient decent, full-batch gradient decent) which means an average gradient is computed for the error on **the whole data set** before a weight update is done. The opposite is stochastic gradient descent (SGD or online gradient descent) in which we compute the gradient for a single training example and updating the weights immediately.
2. There is a clear trade-off here. Looping through the entire dataset gives us a more accurate estimate

of the gradient, but it requires many more computations before we update any weights. It turns out that a good happy medium is to use a small set of training examples known as a mini-batch. This enables more frequent weight updates (less computation per update) than true gradient descent while still getting a more accurate estimate of the gradient than when using just a single example. Further, modern hardware implementations, and in particular graphics processing units (GPUs), do a good job of computing a full mini-batch in parallel, so it does not take more time than computing just a single example.

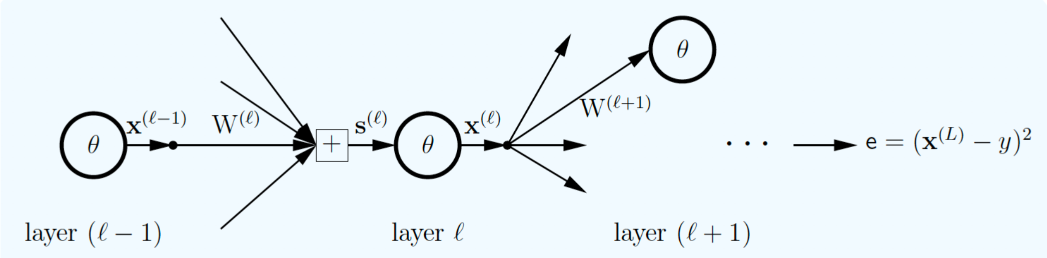
3. The terminology is confusing here. The true gradient descent method uses batches (the entire training dataset) and is also known as batch gradient descent. At the same time, there is the hybrid between batch and stochastic gradient descent that uses mini-batches, but the size of a mini-batch is often referred to as batch size. Finally, SGD technically refers only to the case where a single training example is used (mini-batch size = 1) to estimate the gradient, but the hybrid approach with mini-batches is often also referred to as SGD. thus, it is not uncommon to read statements such as "stochastic gradient descent with a mini-batch size of 64."
4. The mini-batch size is yet another parameter that can be tuned, and the current practice suggests that anything close to the range of 32 to 256 makes sense to try. Finally, SGD (mini-batch size of 1, to be clear) is sometimes referred to as online learning because it can be used in an online setting
5. Now we are also in position to introduce an important terminology. An **epoch** is one complete forward and backward pass over the whole training set. If we divide the training set of the size 10000 in 10 mini-batches, then one forward and one backward pass over a batch is called one iteration, and ten iterations (the size of the mini-batch) is one epoch.

Derivations of Equations (1) and (2)

Let us derive (1) and (2), which are core equations of backpropagation. There's nothing to it but repeated application of the chain rule. Let us start from (1):

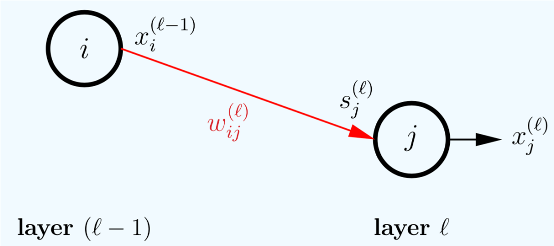
(1)
$$\frac{\partial \mathrm{e}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)}\left(\boldsymbol{\delta}^{(l)}\right)^{\mathrm{T}}$$

To begin, let's take a closer look at the partial derivative, $\frac{\partial \mathrm{e}}{\partial \mathbf{w}^{(l)}}$. The situation is illustrated in the figure below:



We can identify the following chain of dependencies by which $\mathbf{W}^{(l)}$ influences the output $\mathbf{x}^{(L)}$, and hence the error e.

$$W^{(\ell)} \longrightarrow \mathbf{s}^{(\ell)} \longrightarrow \mathbf{x}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell+1)} \cdots \longrightarrow \mathbf{x}^{(L)} = h.$$



To derive (1), we drill down to a single weight and use the chain rule. For a single weight $w_{ij}^{(l)}$, a change in $w_{ij}^{(l)}$ only affect $s_j^{(l)}$ and so by the chain rule:

$$\frac{\partial \mathrm{e}}{\partial w_{ij}^{(l)}} = \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} \cdot \frac{\partial \mathrm{e}}{\partial s_j^{(l)}} = x_i^{(l-1)} \cdot \delta_j^{(l)}$$

The component form of (1). We can derive (2) in a similar fashion and it is left as a self-exercise.