Week 5: Backpropagation

Yunwen Lei

School of Computer Science, University of Birmingham

1 Computation Graph

1.1 Chain Rule on An Example

Let us recall chain rule to compute the derivative of composite functions. It decomposes the derivative of a composite function to derivatives of simple functions.

Theorem 1 (Chain Rule: One Dimensional Case)**.** *For one dimensional differentiable functions f and g, we have*

$$\frac{d}{dx}f(g(x)) = f'(g(x)) \cdot g'(x)$$

Theorem 2 (Chain Rule: Multivariate Case)**.** *For $f(u_1, \ldots, u_m)$ with $u_i = g_i(x_1, \ldots, x_n), i = 1, \ldots, m$, then*

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}.$$

Example 1 (Chain rule on a practical example)**.** Suppose we want to compute the derivative of

$$C = \frac{1}{2}\big(\sigma(wx+b) - y\big)^2, \quad w, b \in \mathbb{R}.$$

According to chain rule, we can directly compute the derivative as follows

$$\begin{aligned}
\frac{\partial C}{\partial w} &= \frac{\partial}{\partial w}\Big[\frac{1}{2}\big(\sigma(wx+b) - y\big)^2\Big] \\
&= \frac{1}{2}\frac{\partial}{\partial w}\big(\sigma(wx+b) - y\big)^2 \\
&= \big(\sigma(wx+b) - y\big)\frac{\partial}{\partial w}\big(\sigma(wx+b) - y\big) \\
&= \big(\sigma(wx+b) - y\big)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) \\
&= \big(\sigma(wx+b) - y\big)\sigma'(wx+b)x
\end{aligned}
\qquad
\begin{aligned}
\frac{\partial C}{\partial b} &= \frac{\partial}{\partial b}\Big[\frac{1}{2}\big(\sigma(wx+b) - y\big)^2\Big] \\
&= \frac{1}{2}\frac{\partial}{\partial b}\big(\sigma(wx+b) - y\big)^2 \\
&= \big(\sigma(wx+b) - y\big)\frac{\partial}{\partial b}\big(\sigma(wx+b) - y\big) \\
&= \big(\sigma(wx+b) - y\big)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= \big(\sigma(wx+b) - y\big)\sigma'(wx+b)
\end{aligned}$$

While this approach is correct, it is not efficient to implement in computers. The underlying reason is that there are some repeated computations

- $\big(\sigma(wx+b) - y\big)$ appear in both $\frac{\partial C}{\partial \mathbf{w}}$ and $\frac{\partial C}{\partial b}$
- $\sigma'(wx+b)$ appear in both $\frac{\partial C}{\partial \mathbf{w}}$ and $\frac{\partial C}{\partial b}$

This means that we have computed $\big(\sigma(wx+b) - y\big)$ and $\sigma'(wx+b)$ twice. The underlying reason is that we compute the partial derivative for both $\mathbf{w}$ and b from scratch. In the computation w.r.t. $\mathbf{w}$ we have already obtained some useful results. These results can be used in the computation w.r.t. b. Furthermore, there is no structure in this computation. This makes the programming difficult.

Example 2 (A more structured way)**.** We can do the computation in a more structured way. The basic idea is to introduce some intermediate variables. In this way, we can store some intermediate results to avoid repeated computation.

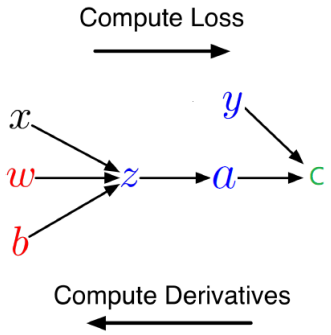
$$C = \frac{1}{2}\big(\sigma(wx+b) - y\big)^2, \quad w, b \in \mathbb{R}.$$

Computing the loss:

$$\begin{aligned} z &= wx + b \\ a &= \sigma(z) \\ C &= \frac{1}{2}(a-y)^2 \end{aligned}$$

Computing the derivatives:

$$\begin{aligned} \frac{\partial C}{\partial a} &= a - y \\ \frac{\partial C}{\partial z} &= \frac{\partial C}{\partial a} \cdot \sigma'(z) \\ \frac{\partial C}{\partial w} &= \frac{\partial C}{\partial z} \cdot x \\ \frac{\partial C}{\partial b} &= \frac{\partial C}{\partial z} \end{aligned}$$



- Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives!
- Note $\frac{\partial C}{\partial z}$ is used twice. We can store it once it is computed, which avoids a repeated computation.
- This structure makes it easy to implement the gradient computation by python. For example, we can introduce a variable for $\frac{\partial C}{\partial z}$.

We represent the structure in the above figure. From this figure, the computation of the loss and the computation of the derivative use different directions. For the loss computation, we first compute z, and then a and then C (from bottom to top). For the gradient computation, we first compute the derivative w.r.t. a, and then z, and finally w and b (from top to bottom). This is due to the chain rule.

We save computation by introducing intermediate variables!

1.2 Computation Graph

In the above example, we represent the computation via a graph. This is an example of computation graph. Computation graph is extremely helpful for us to understand the computation, and to implement them in programming language. Here is the motivation

- For MLPs, it is impossible to derive by-hand the gradients since there are a huge number of parameters.
- We need to decompose complex computations into several sequences of much simpler calculations. This decomposition requires a good structure in the computation. This is the case for MLPs since MLPs consist of neurons structured into different layers. We can use this structure to develop powerful algorithms for gradient computation.

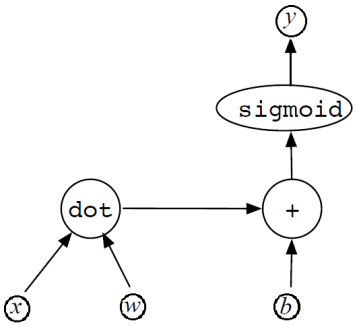
Definition 1 (Computation Graph)**.** A computation graph is a directed acyclic graph. We have a set of nodes and between two nodes there may be a directed edge.

- Node represents all the inputs and computed quantities.
- Edge represents which nodes are computed directly as a function of which other. It shows the dependency between two nodes.
  - an edge from A to B means that output of A is an input of B, that is B depends on A.

Example 3 (Sigmoid function)**.**

Here we give the computation graph for the sigmoid function.

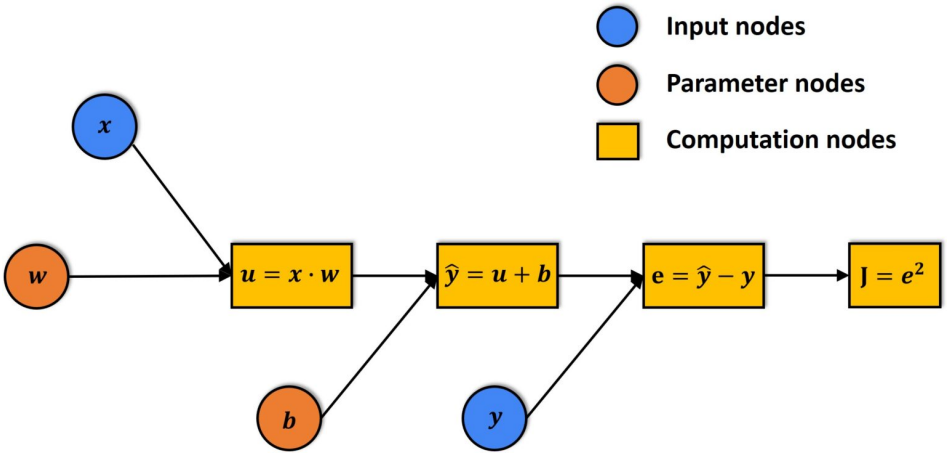
- We introduce nodes for the variable x, w, b, y and operations $\text{dot}, +, \text{sigmoid}$
- The node dot receives input from x and w, whose output is passed further to the node $+$.



Graph for 1-Layer NN

$$y = \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b)$$

Example 4 (Computation graph for linear regression)**.**

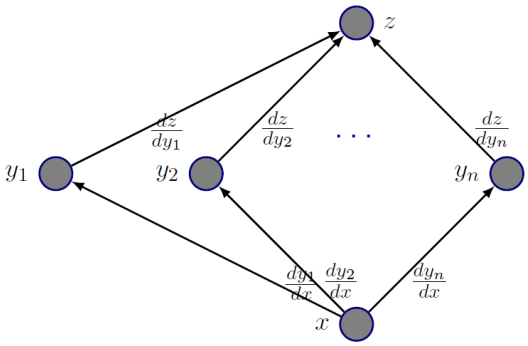


- We have two input nodes for x and y
- We have two parameter nodes for $\mathbf{w}$ and b
- We have four computation nodes
  - we create the variable u for the product of x and w
  - we create $\hat{y}$ as the prediction
  - we create e as the residual
  - we create J as the loss

1.3 Chain Rule in Computation Graph

Computation graph provides a clear explanation for chain rule. Let $\{y_1, \ldots, y_n\}$ be the successor of x. Then

$$\frac{\partial z}{\partial x} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x}$$

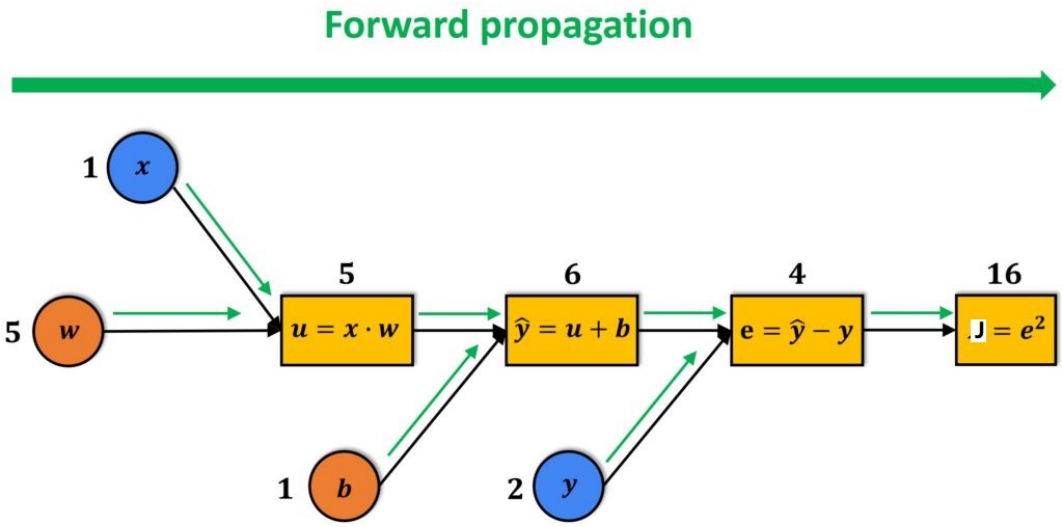


We now explain the gradient of the output z w.r.t. x. Note the gradient means how the change of the input x would change the output z. The change of x would lead to a change of y_1, which would lead to a change of z. The change of x would lead to a change of y_2, which would lead to a change of z. We introduce two types of gradients for our explanation.

- **backpropagated gradient**: the gradient of the *final* output w.r.t. successor ($\frac{\partial z}{\partial y_j}$)
- **local gradient**: the gradient of a successor w.r.t. x ($\frac{\partial y_j}{\partial x}$)

To compute the gradient of z w.r.t. x, we need to compute the gradient of z w.r.t. the successor of x (backpropagated gradient). This means that we need a backward pass in computing gradients. We first compute the backpropagated gradient, which is then multiplied by a local gradient (usually local gradient is easy to compute due to the computation graph). This finishes the computation in a route. We need to take a summation over all the routes.

Example 5 (Forward Propagation to Compute the Loss)**.** We now give an example to use computation graph for computing the loss function. We take a forward pass according the computation graph. In the beginning, we assign $w = 5$ and $x = 1$. The computation node then gives $u = 5$ and $\hat{y} = 6$. We continue this process until we compute the loss $J = 16$.



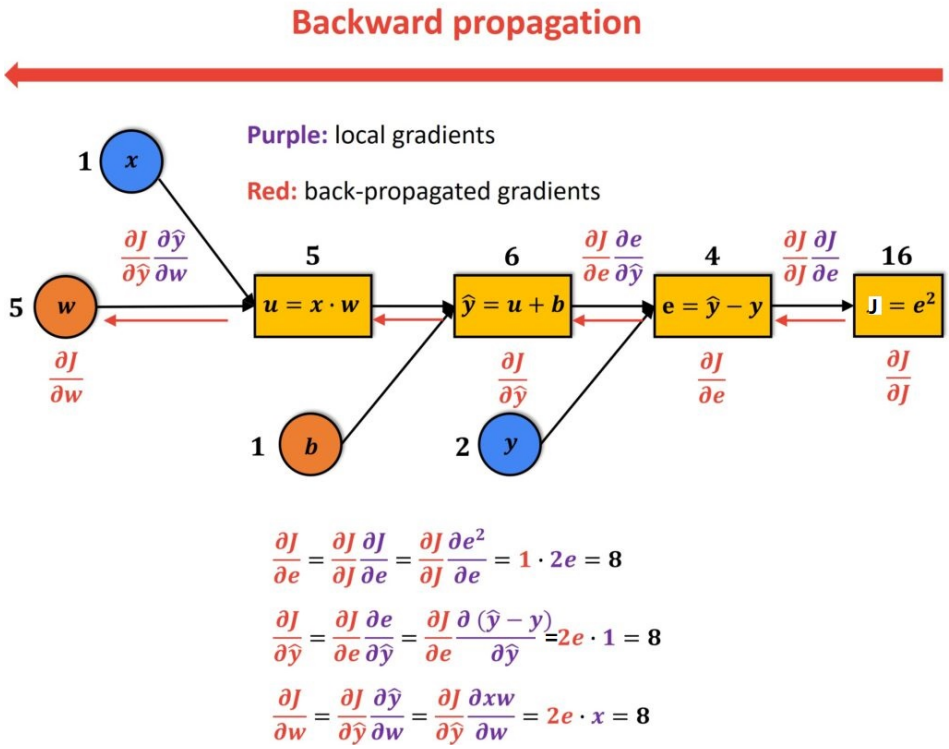
Example 6 (Backward Propagation to Compute the Gradient)**.** We now apply the backward propagation to compute the gradient for the example. We need to compute the backpropagated gradient $\frac{\partial J}{\partial J}$, $\frac{\partial J}{\partial e}$, $\frac{\partial J}{\partial \hat{y}}$ and $\frac{\partial J}{\partial u}$, sequentially. In this process, we can use the result we have already computed.

For example, to compute $\frac{\partial J}{\partial \hat{y}}$, we have

$$\frac{\partial J}{\partial \hat{y}} = \underbrace{\frac{\partial J}{\partial e}}_{A} \underbrace{\frac{\partial e}{\partial \hat{y}}}_{B}.$$

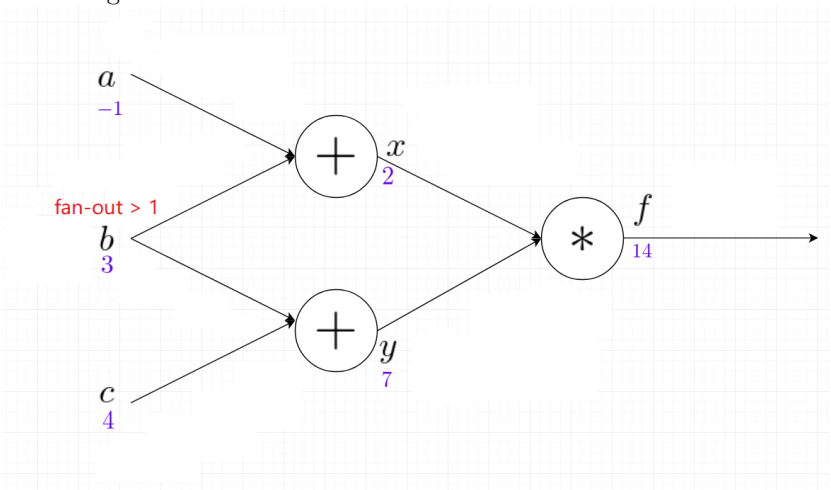
Note the term A is the backpropagated gradient for the successor of $\hat{y}$ and B is a local gradient

- We have already computed A previously. Therefore, we can use it directly.
- The local gradient B is easy to compute. In our case $B = \frac{\partial \hat{y} - y}{\partial \hat{y}} = 1$

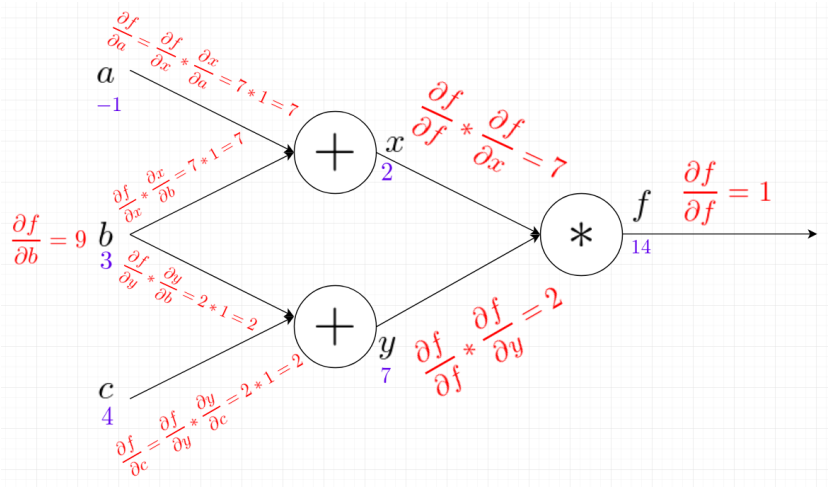


Example 7 (Another Example)**.** In our example, each node only has one child. We now consider another example where a node has fan-out greater than 1. The function is

$$f(a,b,c) = \underbrace{(a+b)}_{:=x}\underbrace{(b+c)}_{:=y}$$



Forward Pass: The above figure shows the forward pass to compute the function value.
Backward Pass: The following figure shows the backward pass to compute the gradient. We compute the backpropagated gradient $\frac{\partial f}{\partial f}$, $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, $\frac{\partial f}{\partial a}$, $\frac{\partial f}{\partial b}$, $\frac{\partial f}{\partial c}$ sequentially.



Take the node b for example. This node has two children x and y. By the chain rule, we know

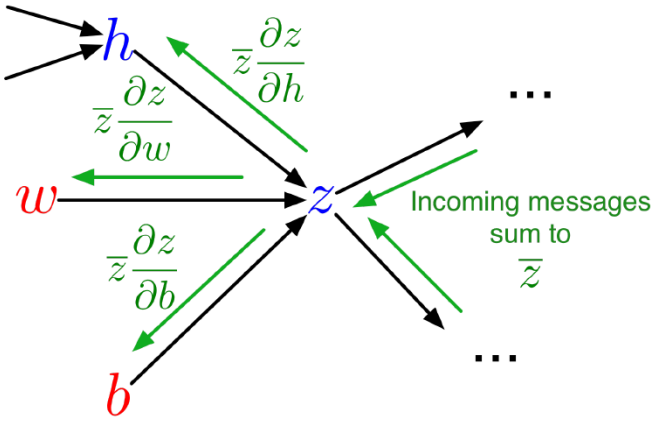
$$\frac{\partial f}{\partial b} = \overbrace{\frac{\partial f}{\partial x}}^{\text{backpropagated gradient}} \overbrace{\frac{\partial x}{\partial b}}^{\text{local gradient}} + \overbrace{\frac{\partial f}{\partial y}}^{\text{backpropagated gradient}} \overbrace{\frac{\partial y}{\partial b}}^{\text{local gradient}}.$$

In the previous computation, we have already computed the above two backpropagated gradients. Therefore, we can use these two results. The local gradients are easy to compute. This leads to

$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial b} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial b} = 7 + 2 = 9$$

Remark 1. If we want to compute the backpropagated gradient for a node, we first need to compute the backpropagated gradients for its successor. We need to take a summation over all the successors.

Remark 2 (Message Passing in Computation Graph)**.** We can understand the backward pass over computation graph as message passing.



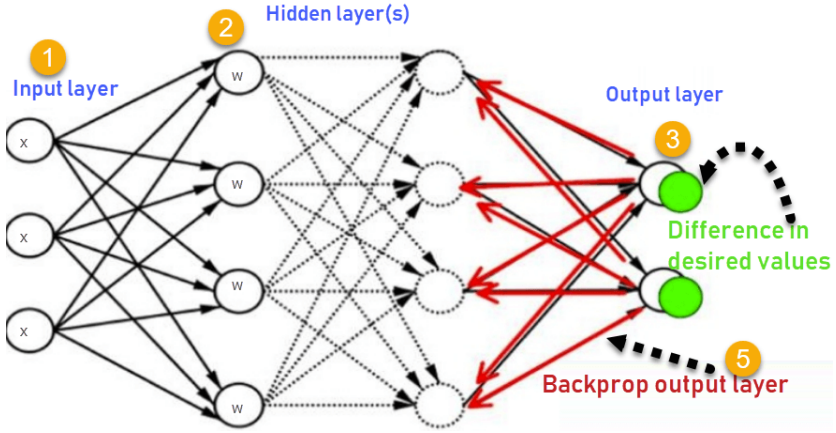
- Each node receives a bunch of messages from its children, which it aggregates to get its signal. It then passes messages to its parents. For example, the node z receives its information from all the children, and then passes it (multiplied by local gradients) to its parent nodes h, w and b.
- This provides modularity, since each node only has to know how to compute derivatives with respect to its arguments, and doesn't have to know anything about the rest of the graph.

2 Backpropagation Algorithm in MLPs

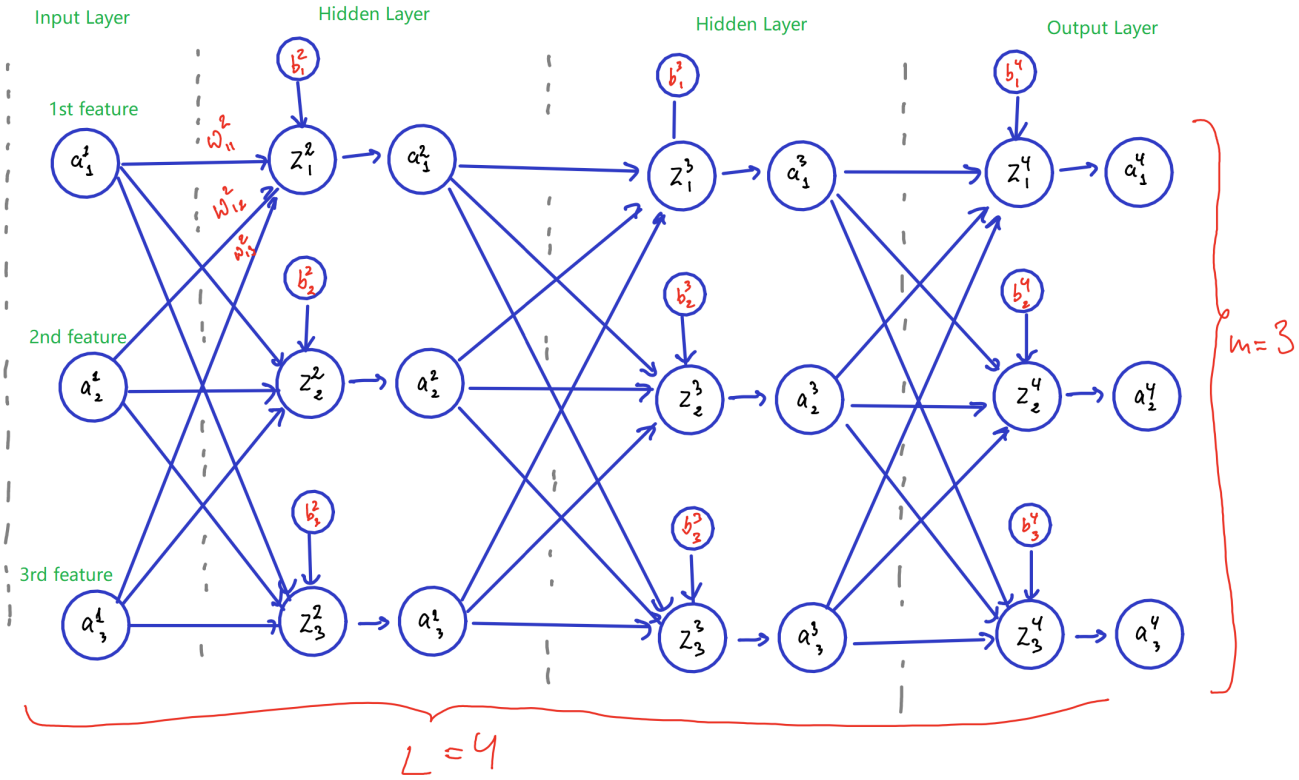
In this section, we apply the backpropagation to MLPs. Backpropagation is a ground-breakthrough algorithm to compute the gradients for neural networks. It makes the training of deep neural networks

possible. Most deep learning libraries have built-in backpropagation steps. Backpropagation appears to be found by Werbos [1974]. It was independently rediscovered around 1985 by Rumelhart, Hinton, and Williams [1986] and by Parker [1985].

The mathematical foundation of backpropagation is chain rule. Direct use of chain rule is not possible due to the large number of parameters. Neural networks are structured in several layers. Each neuron has nodes in the computation graph. We then define several backpropagated gradients as the the gradient of the *loss* w.r.t. a node in the computation graph. Due to the structure in the MLPs, we can find a recursive relationship between backpropagated gradients.



The parameters of the network are the weights ω_{jk}^{ℓ} and the biases b_j^{ℓ}. We need to compute the gradient of the loss function w.r.t. these parameters for the implementation of gradient descent. Before introducing the backpropagation algorithm in MLPs, we first recall the notations

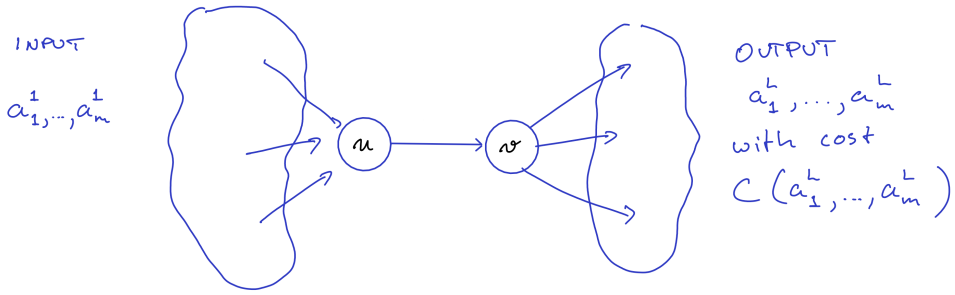


- L: number of layers (superscript 1 is "input layer", superscript L is "output layer")
- m: "width" of network (can vary between layers)

- ω_{jk}^{ℓ}: "weight" of connection between k-th unit in layer $\ell-1$, to j-th unit in layer l
- b_j^{ℓ}: "bias" of j-th unit in layer l
- $z_j^{\ell} = \sum_k \omega_{jk}^{\ell} a_k^{\ell-1} + b_j^{\ell}$: weighted input to unit j in layer ℓ
- $a_j^{\ell} = \sigma(z_j^{\ell})$: "activation" of unit j in layer ℓ, where σ is an "activation function"

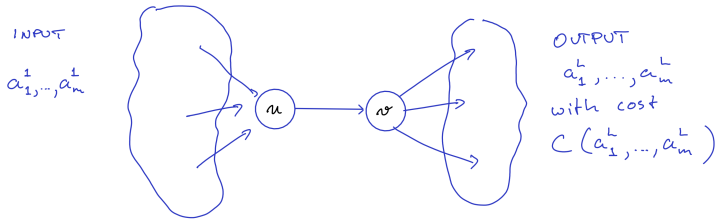
Remark 3 (General idea)**.** To apply gradient descent to optimise a weight ω (or bias b) in a network, we apply the chain rule

$$\frac{\partial C}{\partial u} = \underbrace{\frac{\partial C}{\partial v}}_{\text{back-propagated gradient}} \cdot \underbrace{\frac{\partial v}{\partial u}}_{\text{local gradient}}$$



2.1 Backpropagated Gradient

We have the following relationship between z_j^{ℓ} and ω_{jk}^{ℓ}



$$z_j^{\ell} = \sum_{k=1}^{m} \omega_{jk}^{\ell} a_k^{\ell-1} + b_j^{\ell}$$
$$a_j^{\ell} = \sigma(z_j^{\ell})$$
$$\frac{\partial z_j^{\ell}}{\partial \omega_{jk}^{\ell}} = \frac{\partial \omega_{jk}^{\ell} a_k^{\ell-1}}{\partial \omega_{jk}^{\ell}} = a_k^{\ell-1}.$$

Note z_j^{ℓ} is a function of ω_{jk}^{ℓ}. This means that z_j^{ℓ} is a successor of ω_{jk}^{ℓ}. Furthermore, ω_{jk}^{ℓ} appears only in one neuron in the layer ℓ. That is, ω_{jk}^{ℓ} has only one successor. Therefore, according to the chain rule we have

$$\frac{\partial C}{\partial \omega_{jk}^{\ell}} = \frac{\partial C}{\partial z_j^{\ell}} \cdot \frac{\partial z_j^{\ell}}{\partial \omega_{jk}^{\ell}} = \frac{\partial C}{\partial z_j^{\ell}} \cdot a_k^{\ell-1}$$
$$\frac{\partial C}{\partial b_j^{\ell}} = \frac{\partial C}{\partial z_j^{\ell}} \cdot \frac{\partial z_j^{\ell}}{\partial b_j^{\ell}} = \frac{\partial C}{\partial z_j^{\ell}}$$

"Activation" of unit k in layer $\ell - 1$

In the above computation, $\frac{\partial C}{\partial z_j^{\ell}}$ is a backpropagated gradient (the gradient of the loss w.r.t. a node), while $\frac{\partial z_j^{\ell}}{\partial \omega_{jk}^{\ell}}$ is a local gradient. Since z_j^{ℓ} is a linear function, we know that this local gradient becomes $a_k^{\ell-1}$

Property 1. To compute the derivative w.r.t. parameters, it suffices to compute the back-propagated gradient

$$\delta_j^{\ell} := \frac{\partial C}{\partial z_j^{\ell}}$$

Remark 4 (Vectorization)**.** We have derived

$$\frac{\partial C}{\partial \omega_{jk}^{\ell}} = \delta_j^{\ell} \cdot a_k^{\ell-1} \qquad \frac{\partial C}{\partial b_j^{\ell}} = \delta_j^{\ell}, \tag{2.1}$$

which can be written in terms of matrices

$$\begin{pmatrix} \frac{\partial C}{\partial \omega_{11}^\ell} & \cdots & \frac{\partial C}{\partial \omega_{1m}^\ell} \\ \vdots & \cdots & \vdots \\ \frac{\partial C}{\partial \omega_{m1}^\ell} & \cdots & \frac{\partial C}{\partial \omega_{mm}^\ell} \end{pmatrix} = \begin{pmatrix} \delta_1^\ell \cdot a_1^{\ell-1} & \cdots & \delta_1^\ell \cdot a_m^{\ell-1} \\ \vdots & \cdots & \vdots \\ \delta_m^\ell \cdot a_1^{\ell-1} & \cdots & \delta_m^\ell \cdot a_m^{\ell-1} \end{pmatrix} = \underbrace{\begin{pmatrix} \delta_1^\ell \\ \vdots \\ \delta_m^\ell \end{pmatrix}}_{:=\delta^\ell} \underbrace{\begin{pmatrix} a_1^{\ell-1}, \ldots, a_m^{\ell-1} \end{pmatrix}}_{=\left(\mathbf{a}^{\ell-1}\right)^\top}.$$

In summary, we can represent the gradient w.r.t. both the weight matrices and bias in terms of backpropagated gradients δ^ℓ

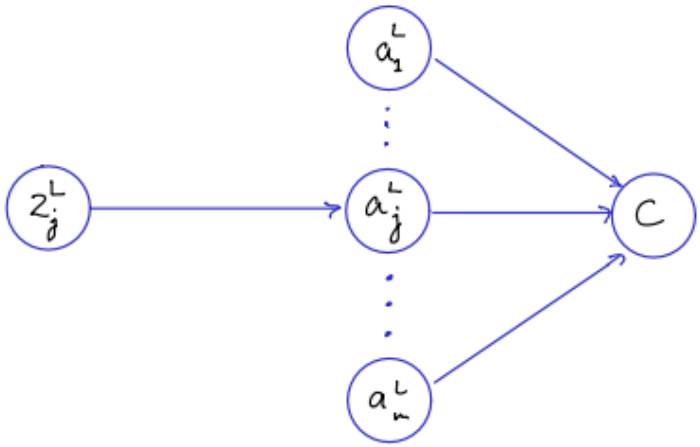
$$\frac{\partial C}{\partial \mathbf{W}^\ell} = \delta^\ell \left(\mathbf{a}^{\ell-1}\right)^\top, \qquad \frac{\partial C}{\partial \mathbf{b}^\ell} = \delta^\ell \tag{2.2}$$

2.2 Recursive Relationship on Backpropagated Gradients

It remans to compute the backpropagated gradients δ^ℓ. In this section, we provide a recursive relationship on Backpropagated Gradients. The basic idea is to first compute the backpropagated gradients δ^L for the output layer. Then we show how to compute δ^{L-1} as a function of δ^L. We repeat this process until we arrive at the input layer.

Back-propagated Gradient for Output Layer. According to the following figure, we know that z_j^L has a successor a_j^L. By the chain rule, we get the backpropagated gradient for the output layer as follows

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L).$$



$\sigma'(z_j^L)$ is easy to compute. The term $\frac{\partial C}{\partial a_j^L}$ depends on the cost function. For example, for a regression problem in m dimensions, one could define

$$C(a_1^L, \cdots, a_m^L) := \frac{1}{2} \sum_{k=1}^{m} \left(y_k - a_k^L \right)^2$$

desired output in k-th dimension

predicted output in k-th dimension

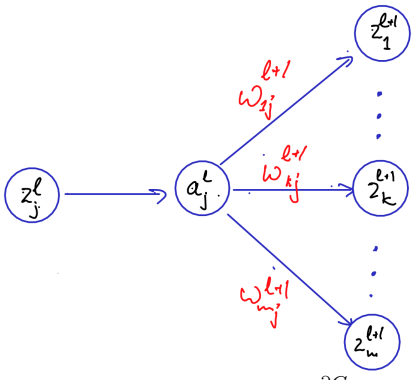
In this case, we know

$$\frac{\partial C}{\partial a_j^L} = a_j^L - y_j$$

Therefore, we get the *Back-propagated Gradient for Output Layer* as follows

$$\delta_j^L = \sigma'(z_j^L)(a_j^L - y_j). \tag{2.3}$$

Back-propagated Gradient for Output Layer. We now consider the hidden layer. According to the structure, the node z_j^ℓ has a successor a_j^ℓ, which has succors $z_i^{\ell+1}$ in the next layer ($i = 1, \ldots, m$). This is illustrated as follows



$$z_k^{\ell+1} = \sum_r \omega_{kr}^{\ell+1} a_r^\ell$$
$$a_j^\ell = \sigma(z_j^\ell)$$

According to the chain rule, to compute $\frac{\partial C}{\partial a_j^\ell}$ we need to take a summation over all of the successors

$$\frac{\partial C}{\partial a_j^\ell} = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial a_j^\ell}.$$

This leads to

$$\begin{aligned}
\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell} &= \frac{\partial C}{\partial a_j^\ell} \cdot \frac{\partial a_j^\ell}{\partial z_j^\ell} && \text{by chain rule} \\
&= \Big(\sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \cdot \frac{\partial z_k^{\ell+1}}{\partial a_j^\ell} \Big) \cdot \sigma'(z_j^\ell) && \text{chain rule wrt } \partial C / \partial a_j^\ell \\
&= \sigma'(z_j^\ell) \sum_k \delta_k^{\ell+1} \cdot \omega_{kj}^{\ell+1} && \text{by definition of back-propagated gradient } \delta_k^{\ell+1}
\end{aligned}$$

- Note $\delta_k^{\ell+1}$ has already been computed since we compute back-propagated gradients from top to bottom!
- This shows that the backpropagated gradients in the layer ℓ can be computed in terms of the backpropagated gradients in the layer $\ell + 1$.
- Therefore, we can first compute δ^L. This is used to compute δ^{L-1}, which is further used to compute δ^{L-2} and so on.

Summary. We can summarize the above discussions as follows

- Gradients w.r.t. ω_{jk}^ℓ and b_j^ℓ can be represented by back-propagated gradients

$$\frac{\partial C}{\partial \omega_{jk}^\ell} = \delta_j^\ell \cdot a_k^{\ell-1}, \qquad \frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell$$

- Back-propagated gradients can be computed in a backward manner

$$\delta_j^\ell = \begin{cases} \sigma'(z_j^L) \cdot \frac{\partial C}{\partial a_j^L}, & \text{if } \ell = L \text{ (output layer)} \\ \sigma'(z_j^\ell) \sum_k \delta_k^{\ell+1} \omega_{kj}^{\ell+1}, & \text{otherwise (hidden layer).} \end{cases} \tag{2.4}$$

Remark 5 (Vectorization)**.** We now show how to represent the computation of backpropagated gradients via matrices. We use $\odot$ to denote Hadamard product (elementwise multiplications), e.g, $(1, 2) \odot (3, 4) = (3, 8)$.

According to Eq. (2.4) with $\ell = L$, we have the following vectorization for the output layer

$$\begin{pmatrix} \delta_1^L \\ \vdots \\ \delta_m^L \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial a_1^L} \cdot \sigma'(z_1^L) \\ \vdots \\ \frac{\partial C}{\partial a_m^L} \cdot \sigma'(z_m^L) \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial a_1^L} \\ \vdots \\ \frac{\partial C}{\partial a_m^L} \end{pmatrix} \odot \begin{pmatrix} \sigma'(z_1^L) \\ \vdots \\ \sigma'(z_m^L) \end{pmatrix} \nabla_{\mathbf{a}^L} C \odot \sigma'(\mathbf{z}^L)$$

According to Eq. (2.4) with $\ell < L$, we have the following vectorization for the hidden layer

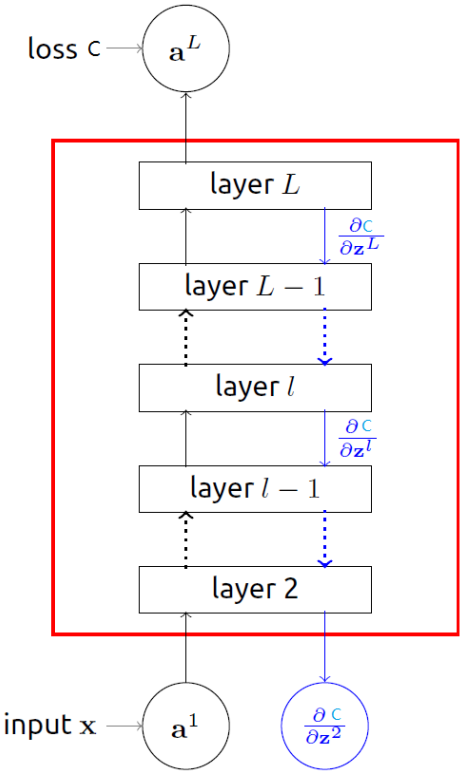
$$\begin{pmatrix} \delta_1^\ell \\ \vdots \\ \delta_m^\ell \end{pmatrix} = \begin{pmatrix} \sigma'(z_1^\ell) \cdot \sum_k \delta_k^{\ell+1} \cdot \omega_{k1}^{\ell+1} \\ \vdots \\ \sigma'(z_m^\ell) \cdot \sum_k \delta_k^{\ell+1} \cdot \omega_{km}^{\ell+1} \end{pmatrix} = \sigma'(\mathbf{z}^\ell) \odot \begin{pmatrix} \sum_k (\mathbf{W}^{\ell+1})_{1k}^\top \delta_k^{\ell+1} \\ \vdots \\ \sum_k (\mathbf{W}^{\ell+1})_{mk}^\top \delta_k^{\ell+1} \end{pmatrix}$$
$$= \sigma'(\mathbf{z}^\ell) \odot \left((\mathbf{W}^{\ell+1})^\top \delta^{\ell+1} \right)$$

In summary, we get the following vectorization of Back-propagated Gradient

$$\delta^\ell = \begin{cases} \nabla_{\mathbf{a}^L} C \odot \sigma'(\mathbf{z}^L), & \text{if } \ell = L (\text{output layer}) \\ \sigma'(\mathbf{z}^\ell) \odot \left((\mathbf{W}^{\ell+1})^\top \delta^{\ell+1} \right), & \text{otherwise (hidden layer).} \end{cases}$$

2.3 Backpropagation Algorithm

Now we are ready to explicitly state the backpropagation algorithm. We first apply forward pass to compute z^ℓ and a^ℓ for $\ell = 1, 2, \ldots, L$. After that we apply the backward pass to compute the backpropagated gradients $\delta^L, \delta^{L-1}, \ldots, \delta^2$. Note that in this process we need to use a^ℓ, which has already been computed in the forward pass.



Forward Equations

1. $\mathbf{a}^1 = \mathbf{x}$
2. $\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell$
3. $\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell)$
4. $C(\mathbf{a}^{(L)}, y)$

Backward Equations

1. $\delta^L = \nabla_{a^L} C \odot \sigma'(\mathbf{z}^L)$
2. $\delta^\ell = \left((\mathbf{W}^{\ell+1})^\top \delta^{\ell+1} \right) \odot \sigma'(\mathbf{z}^\ell)$
3. $\frac{\partial C}{\partial \mathbf{W}^\ell} = \delta^\ell (\mathbf{a}^{\ell-1})^\top$
4. $\frac{\partial C}{\partial \mathbf{b}^\ell} = \delta^\ell$

Note δ^ℓ is a column vector

Algorithm 1 (Backpropagation Algorithm)**.** We now state the backpropagation algorithm.
Input: instance $(\mathbf{x}, y)$, and parameters $\mathbf{W}^\ell, \mathbf{b}^\ell, \ell = 2, \ldots, L$
Output: gradients

1: Set $\mathbf{a}^1 = \mathbf{x}$
2: **for** $\ell = 2, \ldots, L$ **do** ▷ Forward Propagation
3: Compute the activations in ℓ-th layer via

$$\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell, \quad \mathbf{a}^\ell = \sigma(\mathbf{z}^\ell)$$

4: Compute back-propagated gradient for output layer

$$\delta^L = \nabla_{a^L} C \odot \sigma'(z^L), \frac{\partial C}{\partial \mathbf{W}^L} = \delta^L (\mathbf{a}^{L-1})^\top, \frac{\partial C}{\partial \mathbf{b}^L} = \delta^L$$

5: **for** $\ell = L-1, \ldots, 2$ **do** ▷ Backward Propagation

6: Compute back-propagated gradient

$$\delta^\ell = \left((\mathbf{W}^{\ell+1})^\top \delta^{\ell+1}\right) \odot \sigma'(\mathbf{z}^\ell)$$

7: Compute gradients w.r.t. parameters

$$\frac{\partial C}{\partial \mathbf{W}^\ell} = \delta^\ell (\mathbf{a}^{\ell-1})^\top \qquad \frac{\partial C}{\partial \mathbf{b}^\ell} = \delta^\ell.$$

2.4 Gradient Descent for Feedfoward Networks

Backpropagation algorithm provides an efficient way to compute gradients. We can put these gradients into the framework of gradient descent. Note Algorithm 1 computes the gradient of the loss on a single training example $(\mathbf{x}, y)$. The dataset S has n examples.

- Assume n training examples $(\mathbf{x}^{(1)}, y^{(1)}), \ldots (\mathbf{x}^{(n)}, y^{(n)})$ and a cost function

$$C = \frac{1}{n}\sum_{i=1}^{n} C_i,$$

  where C_i is the cost on the i-th example. E.g., we can define $C_i = \frac{1}{2}(y^{(i)} - a^L)^2$ where a^L is the output of the network when $a^1 = \mathbf{x}^{(i)}$.

- Backpropagation gives us the gradient of the overall cost function as follows

$$\frac{\partial C}{\partial \mathbf{W}^\ell} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial C_i}{\partial \mathbf{W}^\ell}$$
$$\frac{\partial C}{\partial \mathbf{b}^\ell} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial C_i}{\partial \mathbf{b}^\ell}$$

  "Averaging" gradient per training example

- We can use gradient descent (or any gradient-based method) to optimize the weights $\mathbf{W}$ and biases $\mathbf{b}$

Remark 6 (Mini-Batch Gradient Descent)**.** Computing the gradient is expensive when the number of training examples n is large.

- We can randomly select a "mini-batch" $I \subset \{1, \ldots, n\}$ of size s per iteration

$$1 < s < n \Longrightarrow \text{Mini-batch gradient descent}$$
$$s = 1 \Longrightarrow \text{Stochastic gradient descent.}$$

  A common choice is $s \in [20, 100]$.

- Build stochastic gradients

$$\frac{\partial C}{\partial \mathbf{W}^\ell} \approx \frac{1}{s}\sum_{i\in I} \frac{\partial C_i}{\partial \mathbf{W}^\ell}$$
$$\frac{\partial C}{\partial \mathbf{b}^\ell} \approx \frac{1}{s}\sum_{i\in I} \frac{\partial C_i}{\partial \mathbf{b}^\ell}$$

3 Summary

Neural nets will be very large. It is impractical to write down gradient formula by hand for all parameters. We need to use the structure of neural networks.

Computation Graph: decompose complex computations into several sequences of much simpler calculations (simplify programming)

- forward pass to compute the cost (compute result of an operation and save any intermediates needed for gradient computation in memory)
- backward pass to compute the gradient based on chain rule. If we want to compute the gradient for a node, we need to compute the gradients for all the successors.

Backpropagation algorithm: recursive application of the chain rule along a computation graph to compute the gradients of all.

- It suffices to compute **back-propagated** gradients (the gradients on the weights and bias can be computed once we know the backpropagated gradients).
- **back-propagated** gradients can be computed recursively (from top to bottom).