Neural Computation

Week 5 - Backpropagation

Yunwen Lei

School of Computer Science, University of Birmingham

Outline

Computation Graph

Recap: Chain Rule

Chain Rule: One Dimensional Case

For one dimensional differentiable functions f and g, we have

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

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

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

video on chain rule `https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/multivariable-chain-rule/v/multivariable-chain-rule`

Chain Rule

How you would have done it in calculus class

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

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

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

What are the disadvantages of this approach?

Chain Rule: A More Structured Way

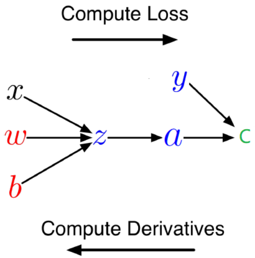
$$C = \frac{1}{2}\big(\sigma(wx + b) - y\big)^2, \quad w, b \in \mathbb{R}.$$

Computing the loss:

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

Computing the derivatives:

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



- Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives!
- Note $\boxed{\frac{\partial C}{\partial z}}$ is used twice and we can store it once it is computed.

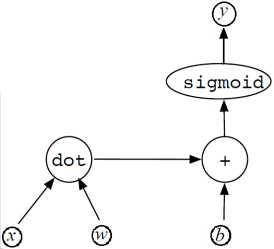
We save computation by introducing intermediate variables!

Computation Graph

- For MLPs, it is impossible to derive by-hand the gradients for a huge number of parameters
- We need to decompose complex computations into several sequences of much simpler calculations

Computation Graph: a directed acyclic graph

- Node represents all the inputs and computed quantities
- Edge represents which nodes are computed directly as a function of which other (dependency)
  - from A to B: output of A is an input of B

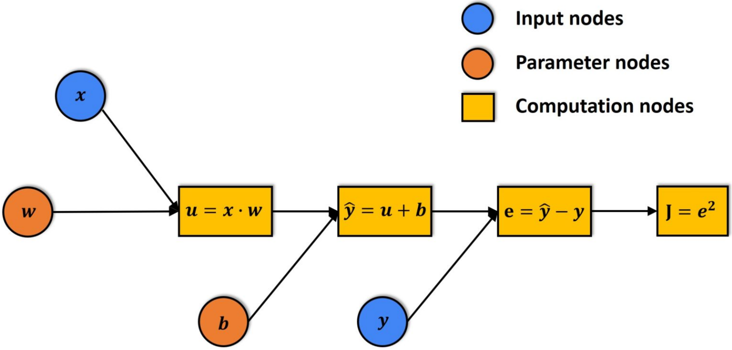


Graph for 1-Layer NN

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

Computation Graph: Example

Computation graph for linear regression

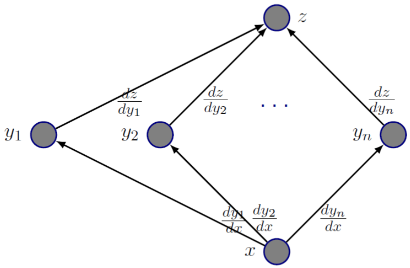


- we create the variable u for the product of x and w
- we create $\hat{y}$ as the prediction
- we create e as the residual
- we crate J as the loss

Chain Rule in Computation Graph

Let $\{y_1, \dots, y_n\}$ be the successor of x. Then

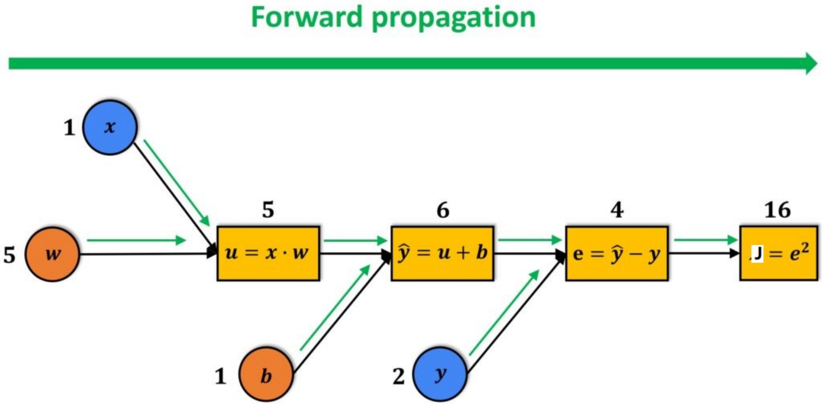
$$\frac{\partial z}{\partial x} = \sum_{j=1}^{n} \boxed{\frac{\partial z}{\partial y_j}} \cdot \boxed{\frac{\partial y_j}{\partial x}}$$



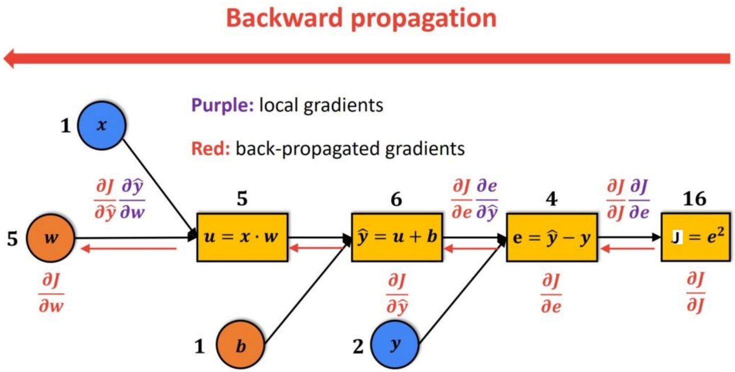
- **backpropagated gradient**: the gradient w.r.t. successor ($\frac{\partial z}{\partial y_j}$)
- **local gradient**: the gradient of a successor w.r.t. itself ($\frac{\partial y_j}{\partial x}$)
- $\frac{\partial y_j}{\partial x}$ is easy to compute due to the decomposition
- If we compute the derivative of z w.r.t. all successor of x, we can immediately get the derivative w.r.t. x.

This means we need a backward pass in computing gradients!

Forward Propagation to Compute the Loss



Backward Propagation to Compute the Gradient

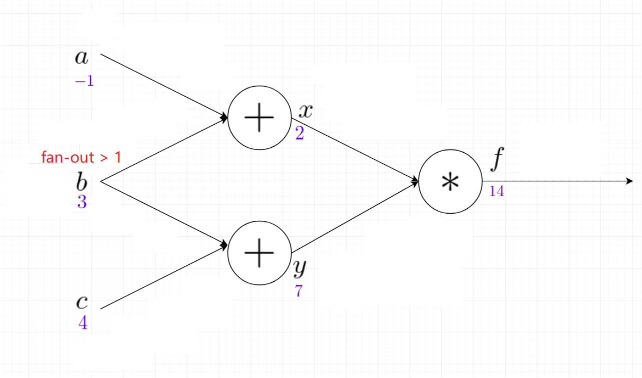


$$\frac{\partial J}{\partial e} = \frac{\partial J}{\partial J}\frac{\partial J}{\partial e} = \frac{\partial J}{\partial J}\frac{\partial e^2}{\partial e} = 1 \cdot 2e = 8$$

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial J}{\partial e}\frac{\partial e}{\partial \hat{y}} = \frac{\partial J}{\partial e}\frac{\partial\left(\hat{y} - y\right)}{\partial \hat{y}} = 2e \cdot 1 = 8$$

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w} = \frac{\partial J}{\partial \hat{y}}\frac{\partial xw}{\partial w} = 2e \cdot x = 8$$

Another Example: Forward Pass

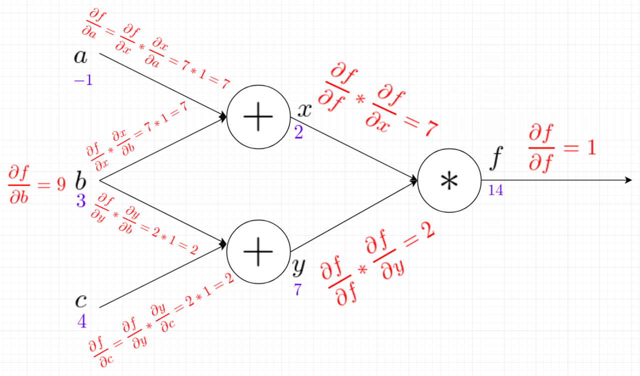


The function is

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

Node b has two children!

Another Example: Backward Pass



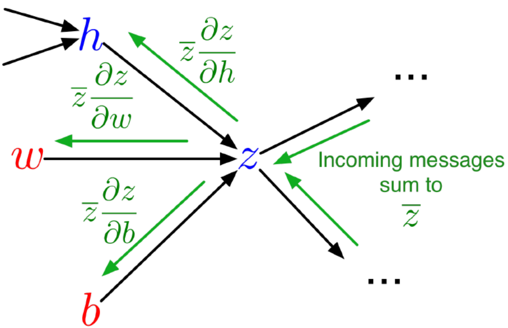
The function is

$$f(a, b, c) = (a + b)(b + c)$$

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

Gradients add at branches!

Message Passing in Computation Graph

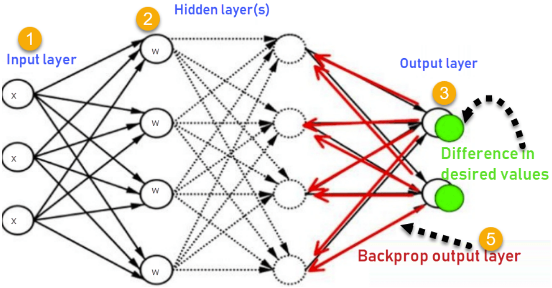


- Each node receives a bunch of messages from its children, which it aggregates to get its signal. It then passes messages to its parents.
- This provides modularity, since each node only has to know how to compute derivatives with respect to its arguments, and doesn't have to know anything about the rest of the graph.

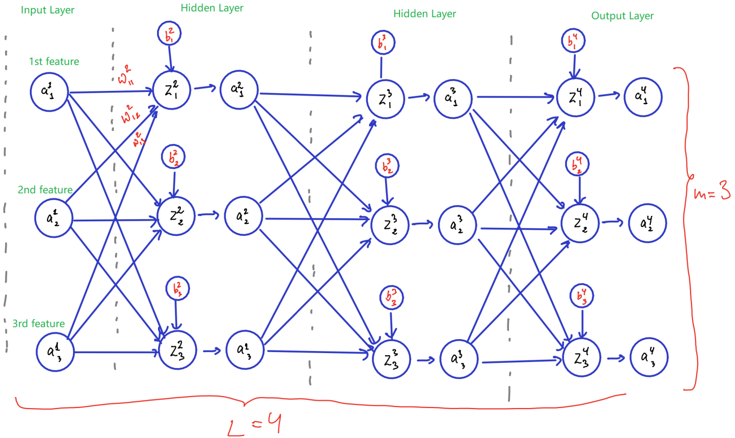
Backpropagation Algorithm in MLPs

Backpropagation Algorithm

- One of the efficient algorithms for MLPs is the Backpropagation algorithm
- Most deep learning libraries have built-in backpropagation steps
- It was re-introduced in 1986 and Neural Networks regained the popularity



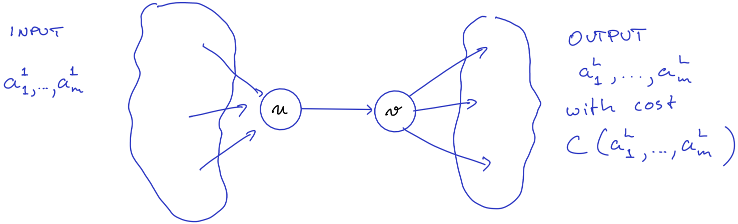
Note: backpropagation appears to be found by Werbos [1974]; and then independently rediscovered around 1985 by Rumelhart, Hinton, and Williams [1986] and by Parker [1985]



- 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"

Training of MLPs

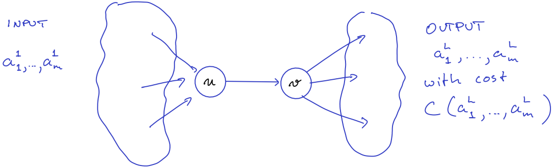
- The parameters of the network are
  - the weights ω_{jk}^{ℓ} in each layer
  - the biases b_j^{ℓ}



- General idea: To apply gradient descent to optimise a weight ω (or bias b) in a network, we apply the chain rule

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

Back-propagated Gradient



$$z_j^\ell = \sum_{k=1}^{m} \omega_{jk}^\ell a_k^{\ell-1} + b_j^\ell$$

$$a_j^\ell = \sigma(z_j^\ell)$$

$$\frac{\partial z_j^\ell}{\partial \omega_{jk}^\ell} = \frac{\partial \omega_{jk}^\ell a_k^{\ell-1}}{\partial \omega_{jk}^\ell} = a_k^{\ell-1}.$$

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

$$\frac{\partial C}{\partial b_j^\ell} = \frac{\partial C}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial b_j^\ell} = \frac{\partial C}{\partial z_j^\ell}$$

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

Back-propagated Gradient

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

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

Vectorization

We have derived

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

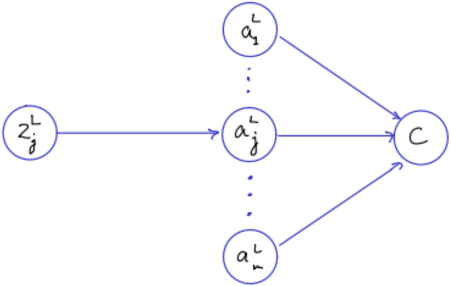
which can be written in terms of matrices

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

Vectorization

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

Back-propagated Gradient for Output Layer



- The back-propagated gradient for the output layer is

$$
\begin{aligned}
\delta_j^L = \frac{\partial C}{\partial z_j^L} &= \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} && \text{(by the chain rule)} \\
&= \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L) && \text{(because } a_j^L = \sigma(z_j^L)\text{)}
\end{aligned}
$$

Back-propagated Gradient for Output Layer

- The partial derivative $\frac{\partial C}{\partial a_j^L}$ depends on the cost function. For example, for a regression problem in m dimensions, one could define

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

desired output in k-th dimension

predicted output in k-th dimension

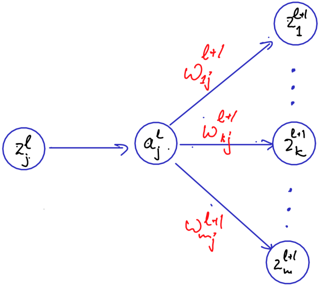
in which case

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

Back-propagated Gradient for Output Layer

$$\delta_j^L = \sigma'(z_j^L)(a_j^L - y_j)$$

Back-propagated Gradient for Hidden Layer



$$z_k^{\ell+1} = \sum_r \omega_{kr}^{\ell+1} a_r^\ell$$

$$a_j^\ell = \sigma(z_j^\ell)$$

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell} = \frac{\partial C}{\partial a_j^\ell} \cdot \frac{\partial a_j^\ell}{\partial z_j^\ell} \qquad \text{by chain rule}$$

$$= \left(\sum_k \boxed{\frac{\partial C}{\partial z_k^{\ell+1}}} \cdot \boxed{\frac{\partial z_k^{\ell+1}}{\partial a_j^\ell}} \right) \cdot \sigma'(z_j^\ell) \qquad \text{chain rule wrt } \partial C / \partial a_j^\ell$$

$$= \sigma'(z_j^\ell) \sum_k \boxed{\delta_k^{\ell+1}} \cdot \boxed{\omega_{kj}^{\ell+1}} \qquad \text{by definition of back-propagated gradient } \delta_k^{\ell+1}$$

Note $\delta_k^{\ell+1}$ has already been computed since we compute back-propagated gradients from top to bottom!

Back-propagated Gradient Summary

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

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

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

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

Back-propagated Gradient: Vectorization

$\odot$ means Hadamard product, e.g, $(1,2) \odot (3,4) = (3,8)$

Output Layer

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

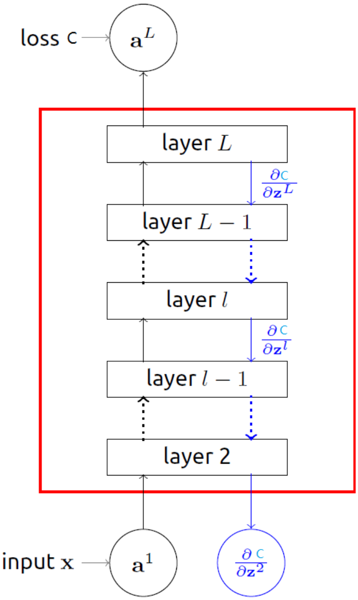
Hidden Layer

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

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

Vectorization of Back-propagated Gradient

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



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

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

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

Gradient Descent for Feedfoward Networks

- 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 **W** and biases **b**

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

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

- Build stochastic gradients

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

Common to use mini-batch size $s \in [20, 100]$

Summary

Summary

Neural nets will be very large: impractical to write down gradient formula by hand for all parameters

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

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

- It suffices to compute **back-propagated** gradients
- **back-propagated** gradients can be computed recursively (from top to bottom)

Next Lecture

Improvements of gradient descent