Deep Learning Theory and Applications

# Backpropagation

Yale

# Calculating the gradients

- We showed how neural networks can learn weights and biases
  - Gradient descent/stochastic gradient descent
- How do we calculate the gradients at each node in each layer?

- Answer: **Backpropagation**!

# Backpropagation history

- Introduced in 1970’s
- Unappreciated until 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams
  - Described several neural networks where backpropagation works far faster than earlier learning approaches
- Today, backpropagation is the “workhorse” of learning neural networks

# Why study backpropagation?

- Why not treat backpropagation like a black box?
- Reason: to obtain understanding
  - Backpropagation provides an expression for $\frac{\partial C}{\partial w}$, the partial derivative of the cost function $C$ wrt any weight $w$ (or bias $b$)
  - I.e., backpropagation tells us how quickly the cost changes when changing weights and biases
  - $\Rightarrow$ backpropagation gives us detailed insights into how changing the weights and biases changes the overall network

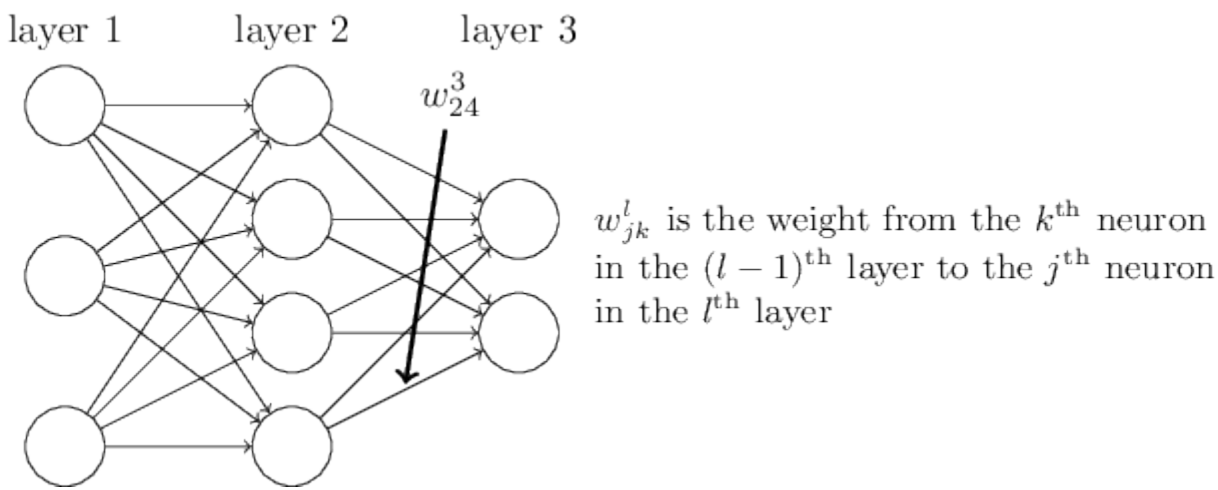
# Outline

1. Preliminaries
   - Matrix multiplication for computing applications
   - Cost function assumptions
   - Hadamard product
   - Error at the node
2. Four fundamental equations of backpropagation
   - The equations
   - Proofs
3. The backpropagation algorithm
4. Is backpropagation fast?
5. Backpropagation: the big picture

# Preliminaries

# Matrix multiplication for computing activations

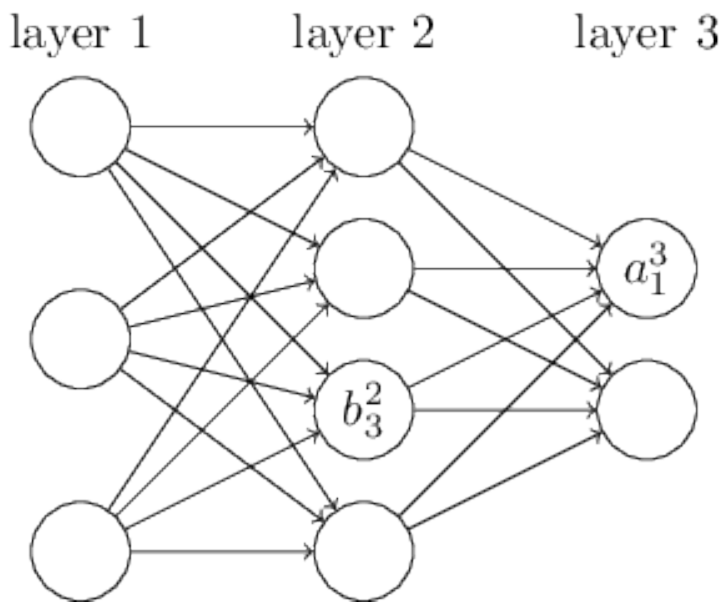
- $w^l_{jk}$ = the weight for the connection from the $k$th neuron in the $(l-1)$th layer to the $j$th neuron in the $l$th layer




- Question: why is $j$ the output neuron and $k$ the input neuron?
- Answer: it simplifies the matrix notation

# Matrix multiplication for computing activations

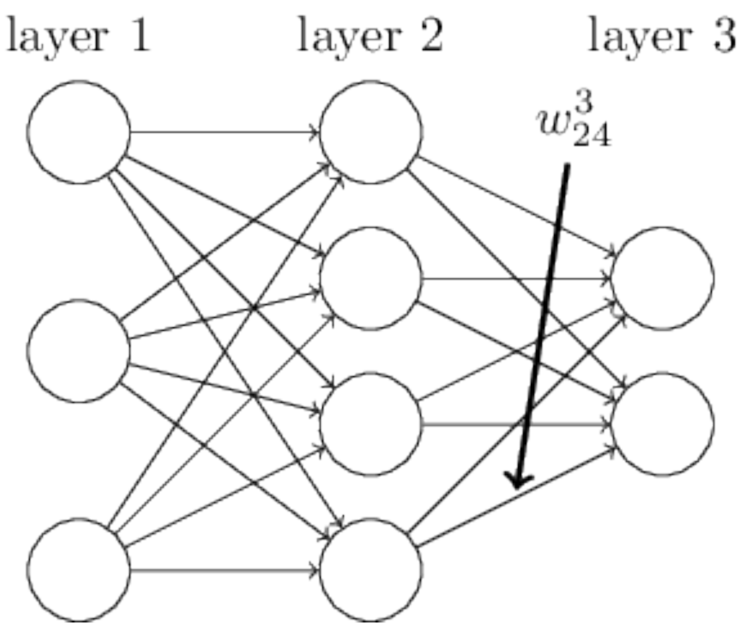
- $b_j^l$ = bias of the $j$th neuron in the $l$th layer
- $a_j^l$ = activation (output) of the $j$th neuron in the $l$th layer

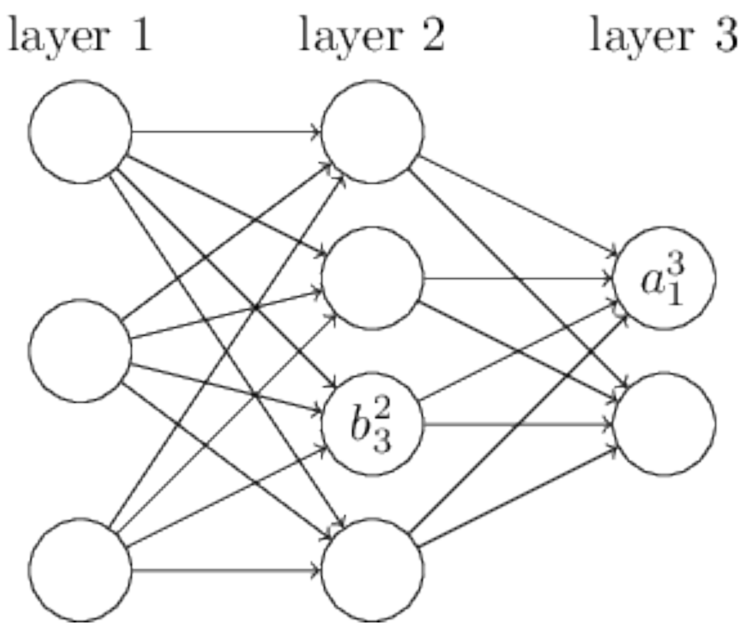


# Matrix multiplication for computing activations

- The activation $a_j^l$ of the $j$th neuron the $l$th layer is related to the activations in the $(l-1)$th layer:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$



# Matrix multiplication for computing activations

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

- Rewrite in matrix form:
- $w^l$ = a ***weight matrix*** for layer $l$
  - Entries are the weights connecting to the $l$th layer of neurons; i.e. the entry in the $j$th row and $k$th column is $w_{jk}^l$
- Bias vector $b^l$ for the $l$th layer ($b_j^l$ in the $j$th component)
- Activation vector $a^l$ for the $l$th layer ($a_j^l$ in the $j$th component)

# Matrix multiplication for computing activations

## Vectorizing a function

- Apply the function element-wise
  - I.e., components of $\sigma(v)$ are $\sigma(v)_j = \sigma(v_j)$
- Example: $f(x) = x^2$
  - Vectorized form of $f$ has the following effect:

$$f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix}$$

## Activation computation

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

# Matrix multiplication for computing activations

Activation computation

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

- Provides a global view of layer-layer relationships
  - Apply weight matrix to activations, add bias vector, then apply $\sigma$
  - Easier and more succinct than neuron-by-neuron view
- Matrix and vector computations are fast

- We will also use an intermediate quantity: $z^l = w^l a^{l-1} + b^l$
  - $z^l$ is the ***weighted input*** to the neurons in layer $l$
  - $a^l = \sigma\left(z^l\right)$
  - $z_j^l$ is the weighted input to the activation function for neuron $j$ in layer $l$

# Cost function assumptions

- Goal of backpropagation: compute the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function $C$ to any weight $w$ or bias $b$ in the network
- Example cost function: quadratic cost

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

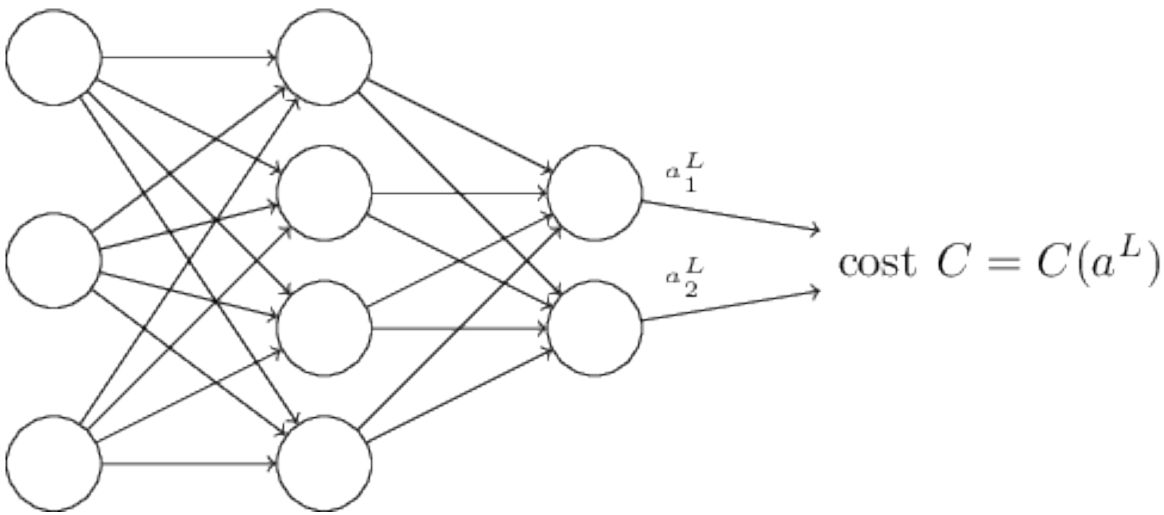
  - Sum is over training examples $x$
  - $y(x)$ is the corresponding desired output
  - $a^L(x)$ is the vector of activation output of the network when $x$ is input ($L$ denotes the number of layers in the network)

# Cost function assumptions

- Example cost function: quadratic cost
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$
- **Assumption 1**: The cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over cost functions $C_x$ for individual training examples $x$.
  - $C_x = \frac{1}{2} \|y - a^L\|^2$ for quadratic cost
- Reason: backpropagation can calculate partial derivatives $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ for a single training example
  - $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ recovered by averaging over training examples
  - For notational simplicity, we'll assume $x$ is fixed and drop the subscript (i.e., write $C_x$ as $C$ for now)

# Cost function assumptions

- **Assumption 2**: Cost can be written as a function of the neural network outputs

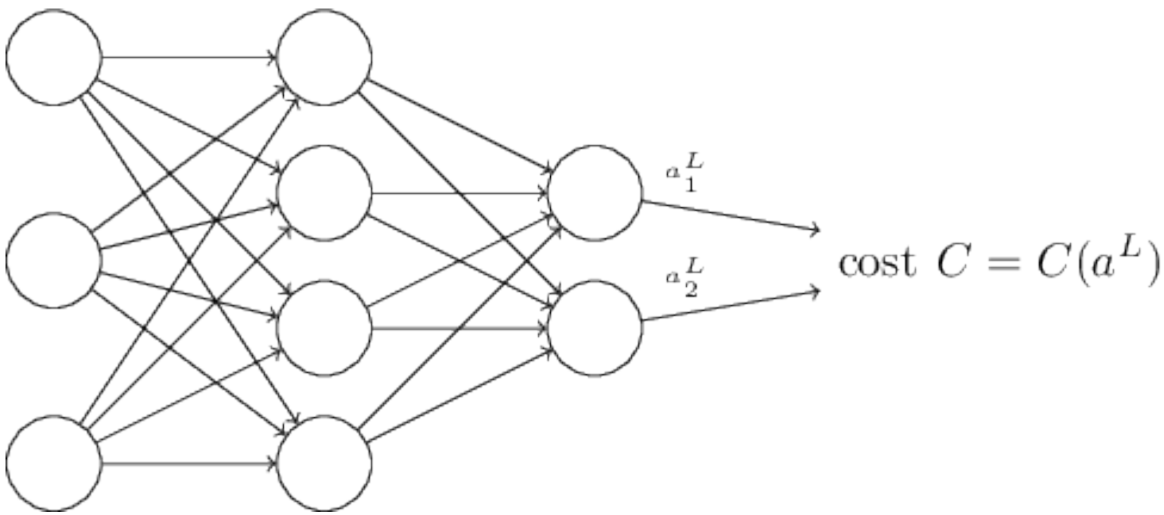



- For quadratic cost:

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j \left(y_i - a_j\right)^2$$

# Cost function assumptions

- **Assumption 1**: The cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$ over cost functions $C_x$ for individual training examples $x$.
- **Assumption 2**: Cost can be written as a function of the neural network outputs.



# Hadamard product

- Let $s$ and $t$ be vectors with same dimension
- $s \odot t$ is the ***elementwise*** product of the vectors
  - $(s \odot t)_j = s_j t_j$
- Example:

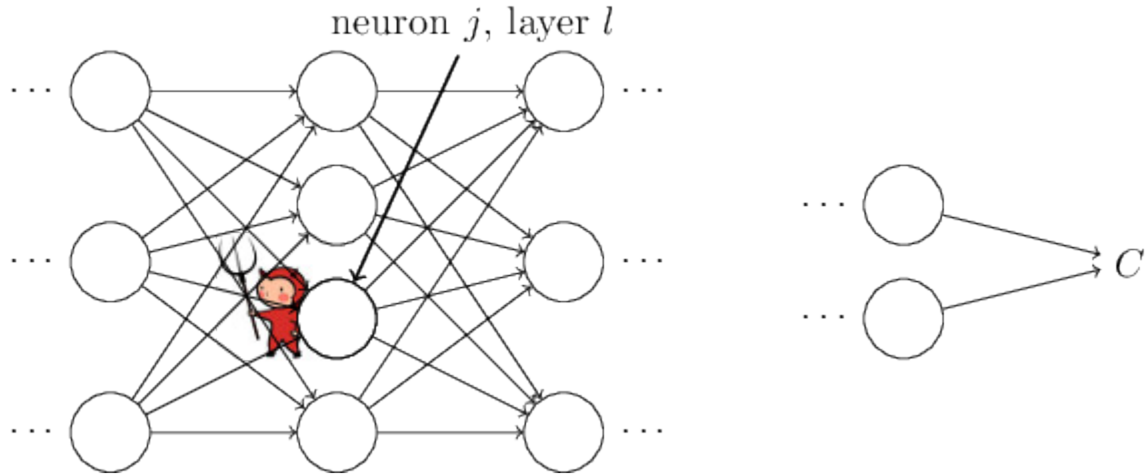
$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1*3\\2*4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix}$$

- Referred to as the ***Hadamard product*** or ***Schur product***

# Error at the node

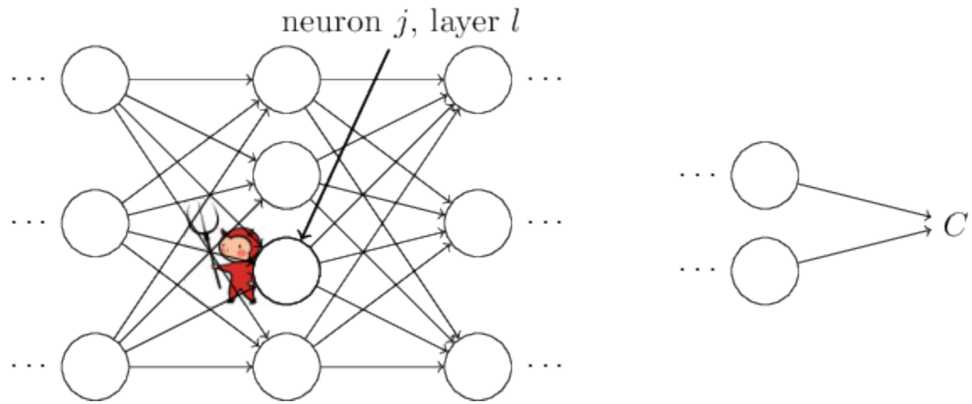
- Backpropagation helps us understand how changing weights and biases in a network changes the cost function
  - I.e., backpropagation gives us the partial derivatives $\frac{\partial C}{\partial w_{jk}^{l}}$ and $\frac{\partial C}{\partial b_{j}^{l}}$
- We first introduce an intermediate quantity $\delta_j^l$
  - $\delta_j^l$ is the ***error*** in the $j$th neuron and the $l$th layer
  - Backpropagation enables us to compute $\delta_j^l$ which we will relate to $\frac{\partial C}{\partial w_{jk}^{l}}$ and $\frac{\partial C}{\partial b_{j}^{l}}$

# Error at the node




- As input comes in, the demon adds a little change $\Delta z_j^l$ to the neuron's weighted input
  - Output becomes $\sigma\left(z_j^l + \Delta z_j^l\right)$
  - Change propagates through later layers causing $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$ change in cost

# Error at the node




- Change $\Delta z_j^l$ results in $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$ change in cost
- Case 1: $\frac{\partial C}{\partial z_j^l}$ is large (either pos. or neg.)
  - Demon can lower cost a lot by choosing $\Delta z_j^l$ with opposite sign of $\frac{\partial C}{\partial z_j^l}$
- Case 2: $\frac{\partial C}{\partial z_j^l}$ is close to zero
  - Can't decrease cost much by perturbing $z_j^l$

# Error at the node

- Case 1: $\frac{\partial C}{\partial z_j^l}$ is large (either pos. or neg.)
  - Demon can lower cost a lot by choosing $\Delta z_j^l$ with opposite sign of $\frac{\partial C}{\partial z_j^l}$
- Case 2: $\frac{\partial C}{\partial z_j^l}$ is close to zero
  - Can't decrease cost much by perturbing $z_j^l$
- $\frac{\partial C}{\partial z_j^l}$ is a measure of error in this heuristic sense
- Define $\delta_j^l = \frac{\partial C}{\partial z_j^l}$ ($\delta^l$ is the vectorized form)
- Similar results obtained by considering $\frac{\partial C}{\partial a_j^l}$

# Four fundamental equations of backpropagation

# First fundamental equation of backpropagation

- Error in the output layer:

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

- $\frac{\partial C}{\partial a_j^L}$ measures how fast cost $C$ changes as function of $j$th output
  - Example: if $C$ doesn't depend much on neuron $j$, then $\delta_j^L$ will be small
- $\sigma'(z_j^L)$ measures how fast the activation function $\sigma$ changes at $z_j^L$

# First fundamental equation of backpropagation

- Error in the output layer:

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

- All parts are easily computable
  - $z_j^L$ is computed while computing behavior of the network
    - $\sigma'\left(z_j^L\right)$ follows easily
  - $\frac{\partial C}{\partial a_j^L}$ depends on cost function
    - Quadratic cost: $C = \frac{1}{2}\sum_j\left(y_j - a_j^L\right)^2 \Rightarrow \frac{\partial C}{\partial a_j^L} = \left(a_j^L - y_j\right)$

# First fundamental equation of backpropagation

- Error in the output layer:

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

- Matrix-based form:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

  - $(\nabla_a C)_j = \frac{\partial C}{\partial a_j^L}$
  - Example: quadratic cost

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

  - Easily computed using Numpy

# Second fundamental equation of backpropagation

- Error in the $l$th layer in terms of the error in the next layer:

$$\delta^l = \left( \left( w^{l+1} \right)^T \delta^{l+1} \right) \odot \sigma'(z^l)$$

Interpretation

- Suppose $\delta^{l+1}$ is known
- Applying $\left( w^{l+1} \right)^T$ moves the error *backward* through the network
  - Gives a measure of the error at the output of the $l$th layer
- Taking the Hadamard product $\odot \sigma'(z^l)$ moves the error backward through the activation function in layer $l$
  - Gives the error $\delta^l$ in the weighted input to layer $l$

# Second fundamental equation of backpropagation

- First fundamental equation

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

- Second fundamental equation

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'(z^l)$$

- Can compute the error $\delta^l$ for any error using these equations
  1. Compute $\delta^L$
  2. Compute $\delta^{L-1}$
  3. Compute $\delta^{L-2}$
  4. And so on all the way back through the network

# Third fundamental equation of backpropagation

- The rate of change of the cost wrt any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

- Easily computed from previous equations

# <u>Fourth</u> fundamental equation of backpropagation

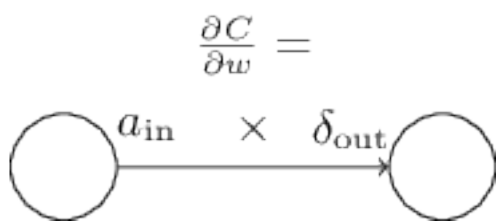
- The rate of change of the cost wrt any weight in the network

$$\frac{\partial C}{\partial w_{jk}^{l}} = a_k^{l-1}\delta_j^l$$

- Easily computed
- Simplified notation:

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}}$$

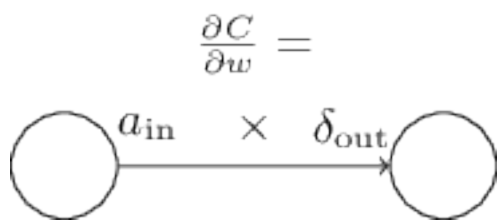
  - $a_{\text{in}}$ is the activation of the neuron input to the weight $w$
  - $\delta_{\text{out}}$ is the error of the neuron output from the weight $w$



# <u>Fourth</u> fundamental equation of backpropagation

- The rate of change of the cost wrt any weight in the network

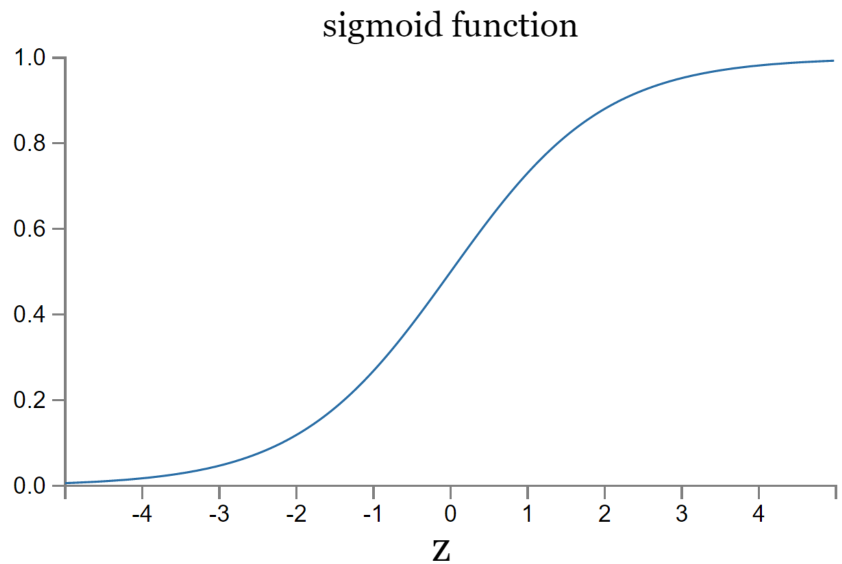
$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}$$




- If $a_{\text{in}} \approx 0$, the gradient $\frac{\partial C}{\partial w}$ will be small
  - The weight ***learns slowly*** (i.e. doesn’t change much during gradient descent)
  - I.e., weights output from low-activation neurons learn slowly

# Insights from the 4 fundamental equations

- Consider the output layer: $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$




- $\sigma$ becomes very flat when $\sigma \approx 0$ or 1
  - $\Rightarrow \sigma'(z_j^L) \approx 0$
  - $\Rightarrow$ weight in final layer will learn slowly if output neuron is ***saturated*** (i.e. $\approx 0$ or $\approx 1$)

# Insights from the 4 fundamental equations

- Similar insights for other layers:
- Weights will learn slowly if either the input neuron is low activation or the output neuron has saturated (i.e. low or high activation)
- The 4 fundamental equations hold for any activation functions
  - We can use these equations to design activation functions
  - E.g., choose $\sigma$ s.t. $\sigma'$ is always positive and never close to zero (we'll see this later in the course)
  - Understanding the 4 fundamental equations can guide us designing neural networks

# The 4 fundamental equations of backpropagation

1. $\delta^L = \nabla_a C \odot \sigma'(z^L)$
2. $\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'(z^l)$
3. $\frac{\partial C}{\partial b_j^l} = \delta_j^l$
4. $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$

# Proofs of the 4 fundamental equations of backpropagation

# Proof of the 1st equation

- By definition:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

- Use multivariate chain rule to obtain:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}$$

- Recall that $a_j^L = \sigma(z_j^L)$

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

# Proof of the 2nd equation

- Rewrite $\delta_j^l$ in terms of $\delta_k^{l+1}$ using chain rule

$$\delta_j^l = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

- Differentiating gives

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

$$\Rightarrow \delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

# Proof of the 3rd equation

- Use multivariate chain rule to obtain:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l}$$

- $\frac{\partial z_j^l}{\partial b_j^l} = 1$

$$\Rightarrow \frac{\partial C}{\partial b_j^l} = \delta_j^l$$

# Proof of the 4th equation

- Use multivariate chain rule to obtain:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l}$$

- $\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}$

$$\Rightarrow \frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

# The backpropagation algorithm

# The backpropagation algorithm

Backpropagation equations provide a way for computing the gradient of the cost function

1. **Input** $x$: Set the activation $a^1$ for the input layer
2. **Feedforward**: For each $l = 2,3, \ldots, L$, compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
3. **Output error** $\delta^L$: Compute $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. **Backpropagate the error**: For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'(z^l)$
5. **Output**: The cost function gradient is $\frac{\partial C}{\partial w_{jk}^l} = a^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

# Backpropagation with SGD

- Backpropagation computes the gradient of the cost function for a single training example $C = C_x$
- Typically, backpropagation is combined with a learning algorithm such as stochastic gradient descent

# Backpropagation with SGD

1. **Input a set of training examples**
2. **For each training example** $x$: Set the input activation $a^{x,1}$ and do the following:
   - **Feedforward**: For each $l = 2,3,\ldots,L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma\left(z^{x,l}\right)$
   - **Output error** $\delta^{x,L}$: Compute $\delta^{x,L} = \nabla_a C \odot \sigma'(z^{x,L})$
   - **Backpropagate the error**: For each $l = L-1, L-2, \ldots, 2$ compute $\delta^{x,l} = \left(\left(w^{l+1}\right)^T \delta^{x,l+1}\right) \odot \sigma'(z^{x,l})$
3. **Gradient descent**: for each $l = L, L-1, \ldots, 2$ update weights $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} \left(a^{x,l-1}\right)^T$ and the biases $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

# Is backpropagation fast?

# Is backpropagation fast?

- Consider an alternative approach
- Suppose you try to not use the chain rule and regard the cost as a function of the weights directly $C = C(w)$
- Could use the approximation:

$$\frac{\partial C}{\partial w_j} \approx \frac{C\left(w + \epsilon e_j\right) - C(w)}{\epsilon}$$

  - $\epsilon > 0$ is a small positive number
  - $e_j$ = unit vector in $j$th direction
  - I.e., we're estimating the derivatives directly
  - Same idea applies to biases
- Advantage: very easy to implement
- Disadvantage: very slow

# Is backpropagation fast?

- Could use the approximation:

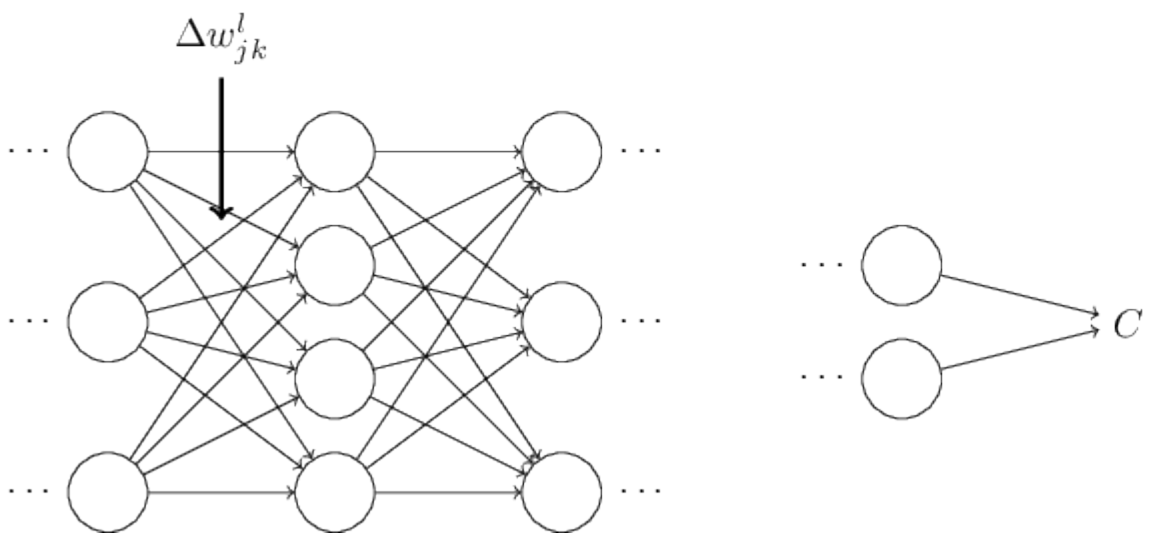
$$\frac{\partial C}{\partial w_j} \approx \frac{C(w + \epsilon e_j) - C(w)}{\epsilon}$$

- Why is it slow?
- Suppose we have a million weights in our network
  - For each weight $w_j$ we compute $C(w + \epsilon e_j)$
  - Thus we need to compute the cost function a million times, requiring a million forward passes through the network ***per training sample***
- In contrast, backpropagation computes all the partial derivatives $\frac{\partial C}{\partial w_j}$ using one forward and backward pass
  - Roughly equivalent to two forward passes $\ll$ million passes

# Backpropagation: the big picture

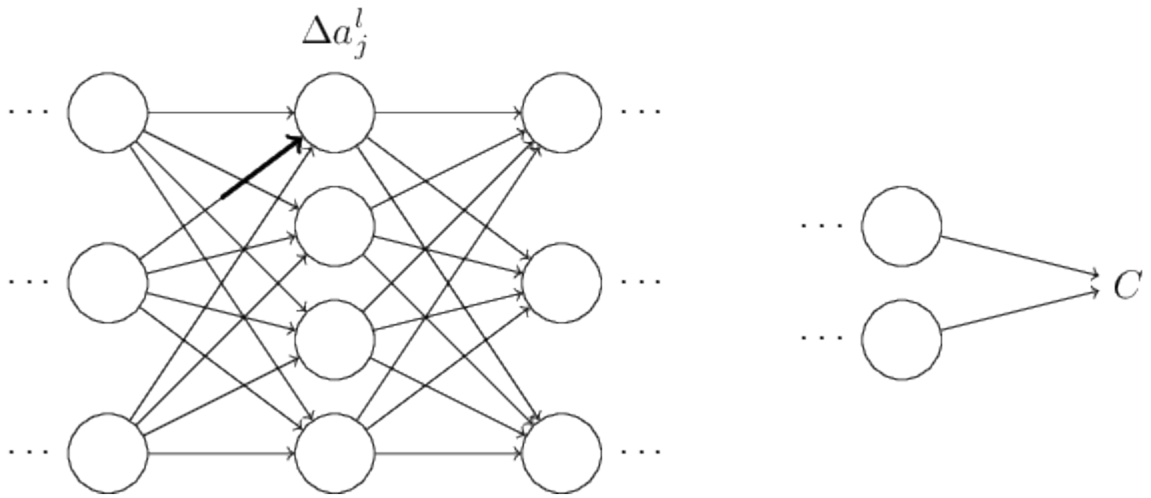
# Improved intuition

- Suppose we make a small change $\Delta w_{jk}^{l}$ to weight $w_{jk}^{l}$



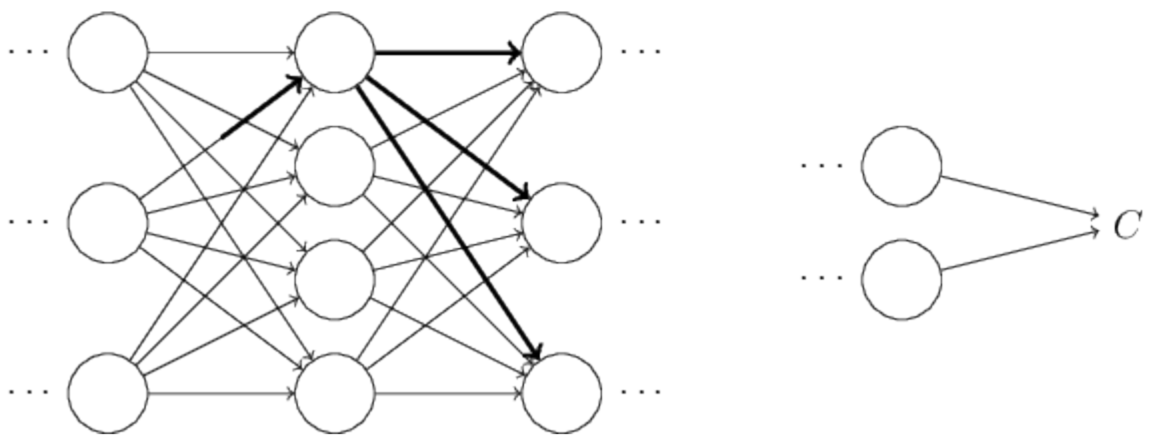
# Improved intuition

- Suppose we make a small change $\Delta w_{jk}^{l}$ to weight $w_{jk}^{l}$
- This causes a change in the output activation of the corresponding neuron



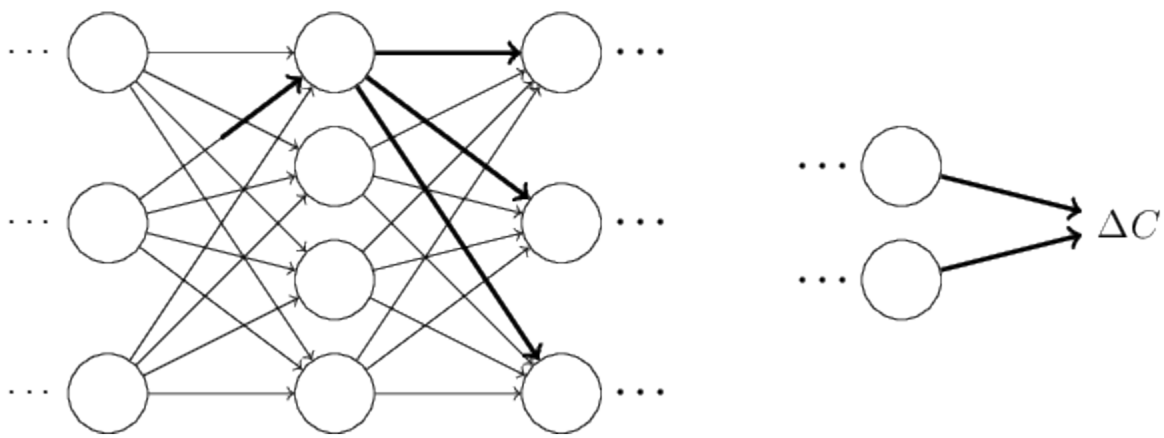
# Improved intuition

- Suppose we make a small change $\Delta w_{jk}^{l}$ to weight $w_{jk}^{l}$
- This causes a change in the output activation of the corresponding neuron
- This causes a change in <u>all</u> activations in the next layer



# Improved intuition

- Suppose we make a small change $\Delta w_{jk}^{l}$ to weight $w_{jk}^{l}$
- This causes a change in the output activation of the corresponding neuron
- This causes a change in all activations in the next layer
- And so on until the final layer



# Improved intuition

- Change in cost $\Delta C$ is related to the change $\Delta w_{jk}^{l}$:

$$\Delta C \approx \frac{\partial C}{\partial w_{jk}^{l}} \Delta w_{jk}^{l}$$

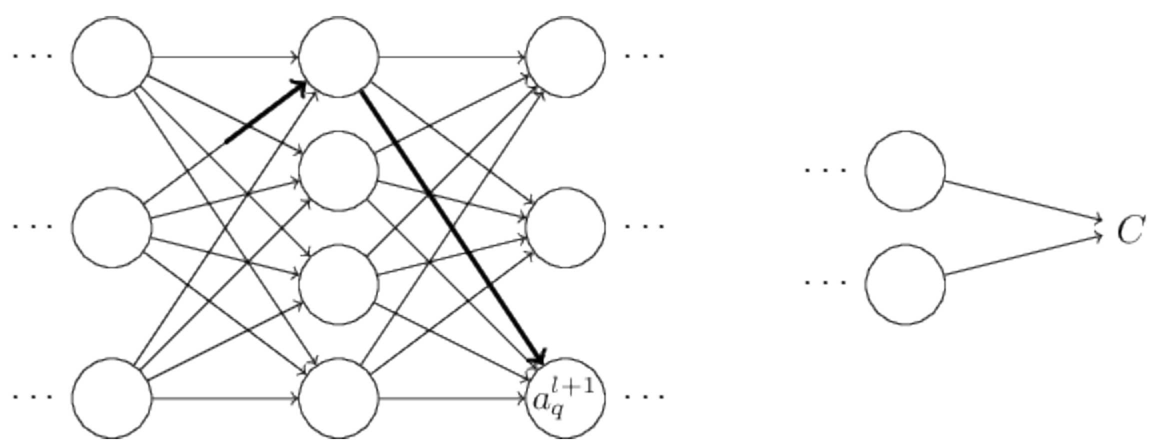
- Possible approach to calculate $\frac{\partial C}{\partial w_{jk}^{l}}$ is to track how a small change in $w_{jk}^{l}$ propagates to change $C$
  - Use easily computable quantities along the way

# Improved intuition

- Relate change in weight $\Delta w_{jk}^l$ to change in activation $\Delta a_j^l$

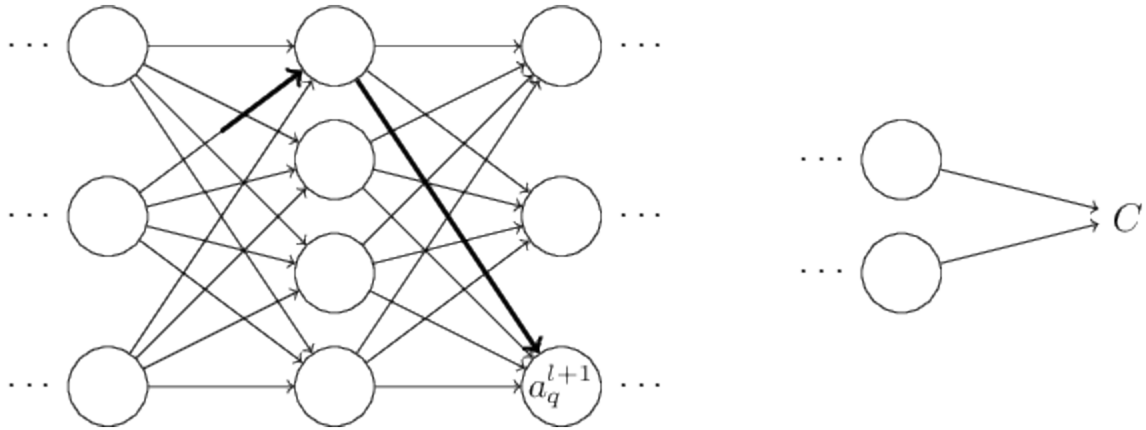
$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

- Change in activation $\Delta a_j^l$ causes changes in all activations of the $(l+1)$th layer
- Focus on $a_q^{l+1}$:



# Improved intuition

- Focus on $a_q^{l+1}$:




$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l$$

$$\Rightarrow \Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

# Improved Intuition

- Extending this to multiple layers gives:

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

- Represents the change in $C$ due to a change in $w_{jk}^l$ through one path in the network
- All paths:

$$\Delta C \approx \sum_{mn\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

# Improved Intuition

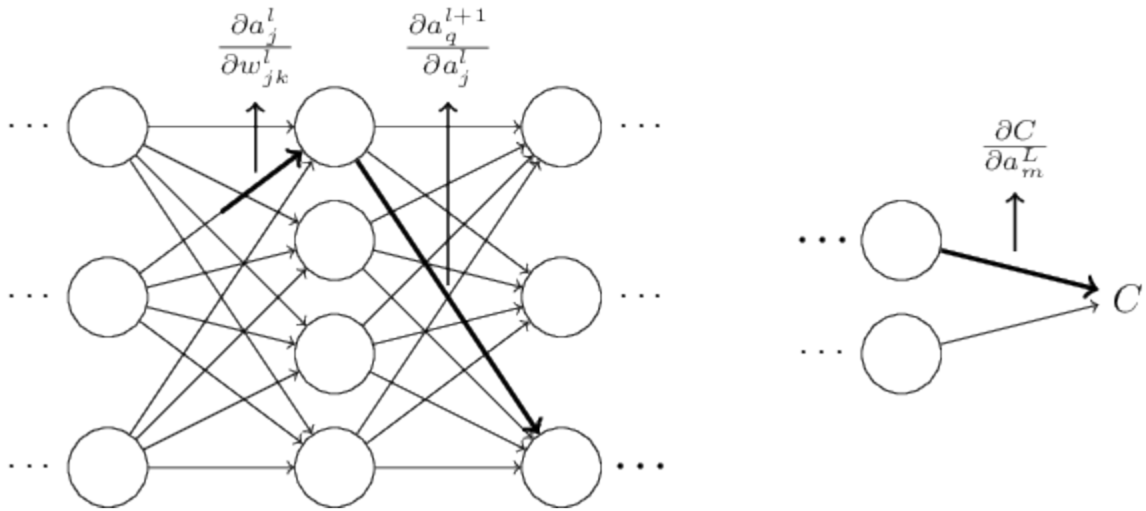
$$\frac{\partial C}{\partial w_{jk}^{l}} = \sum_{mn\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}$$

Interpretation

- Every edge between two neurons is associated with a rate factor (the partial derivative of one neuron's activation wrt the other neuron's activation)
- The rate factor for a path is the product of rate factors along the path
- Total rate of change is the sum of the rate factors of all paths from the initial weight to final cost

# Improved Intuition

- Single path




1. Could derive expressions for each partial derivative
    - Use calculus
2. Write sums as matrix multiplications
3. Simplify

Result is backpropagation

# Discovering backpropagation

1. Follow the long approach just described
2. Discover there are some obvious simplifications
3. Make those simplifications to get a shorter proof
4. Repeat until you get the nice proof we did in class

# Further reading

- Nielsen book, chapter 2
- Goodfellow et al., section 6.5