Deep Learning Theory and Applications

# Backpropagation

Yale

CPSC/AMTH/CBB 663

CPSC 452/552

# 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**!

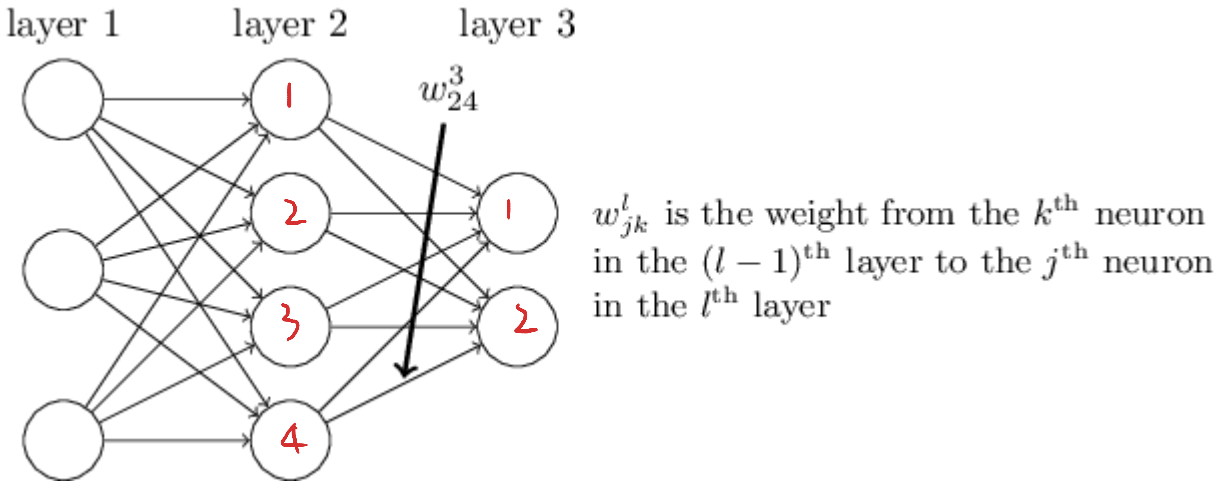
# Backpropagation history

- 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

# Preliminaries

# Matrix multiplication for computing activations

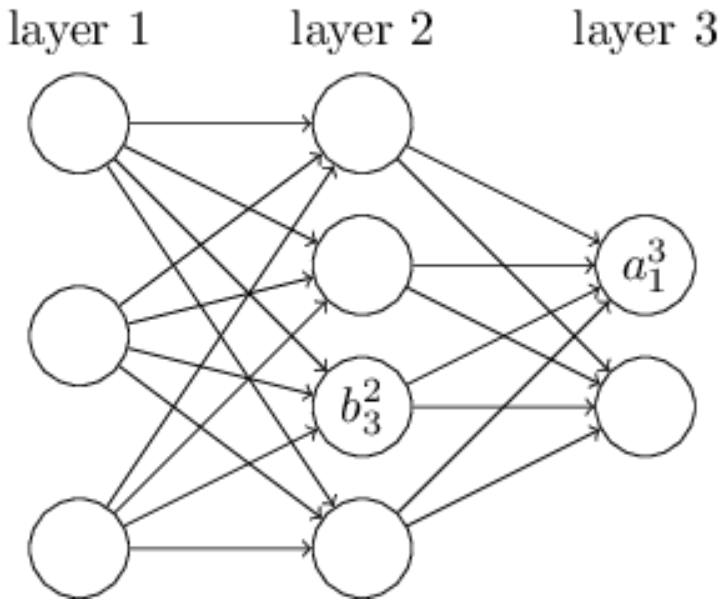
- $w_{jk}^{l}$ = the weight for the connection from the $k$th neuron in the $(l-1)$th layer to the $j$th neuron in the $l$th layer




- Question: why is $j$ the output neuron and $k$ the input neuron?
- Answer: it simplifies the matrix notation for multiplication to he right Wx+b

# 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



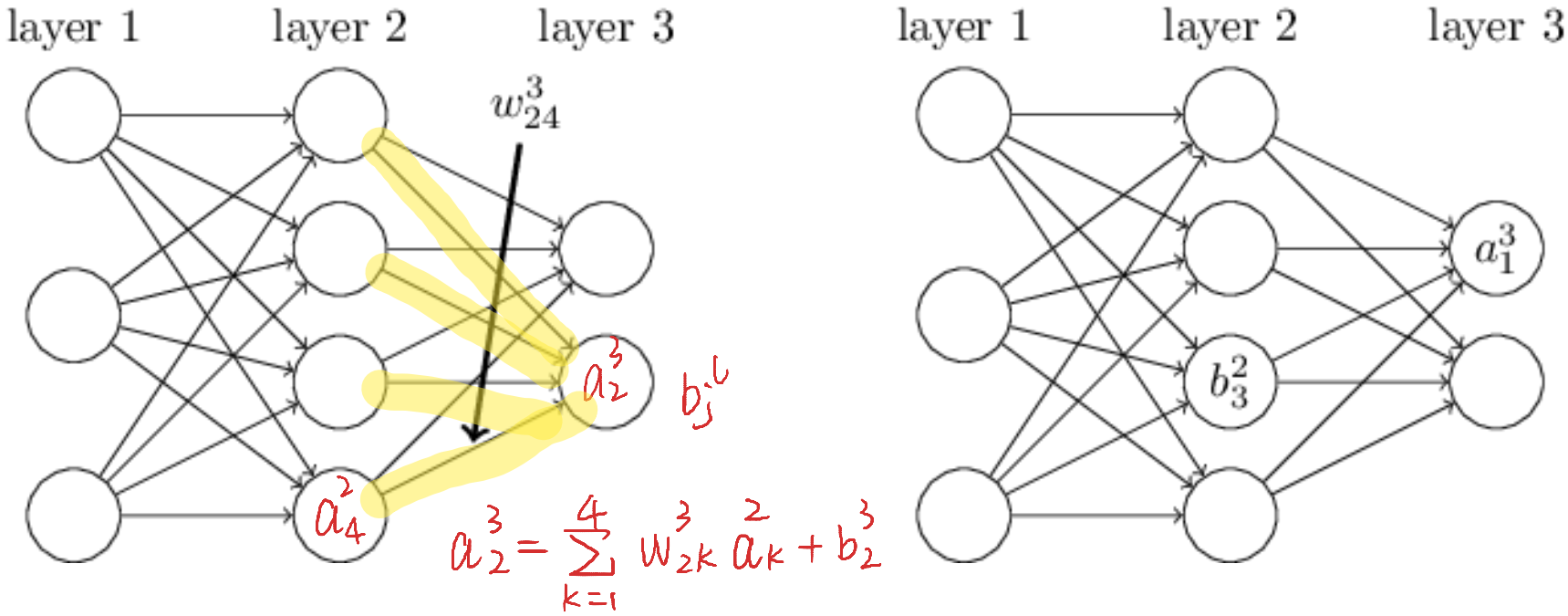
# Matrix multiplication for computing activations

- 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:

activation function

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

for linear function $\sigma = 1$



# 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

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

weight matrix

activation vector

bias vector

# Matrix multiplication for computing activations

Activation computation

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

$$= \sigma(z^l)$$

- 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

  use GPU to do compute
- 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

- 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 $x$ or batch
  - $y(x)$ 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) = output of final layer

# Cost function assumptions

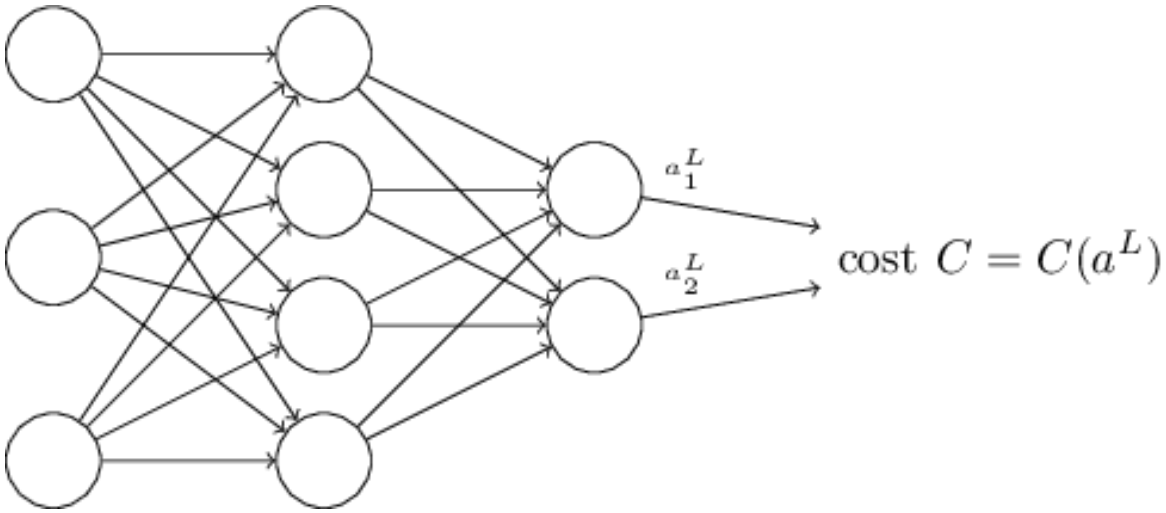
- Example cost function: quadratic cost

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

- **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

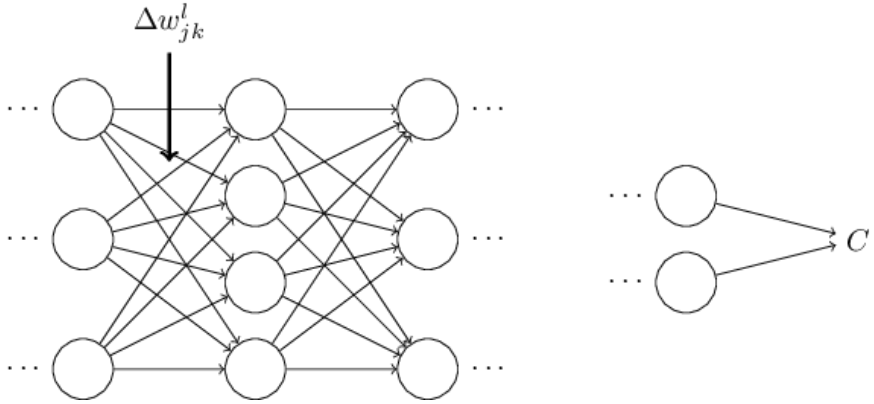



- For quadratic cost:

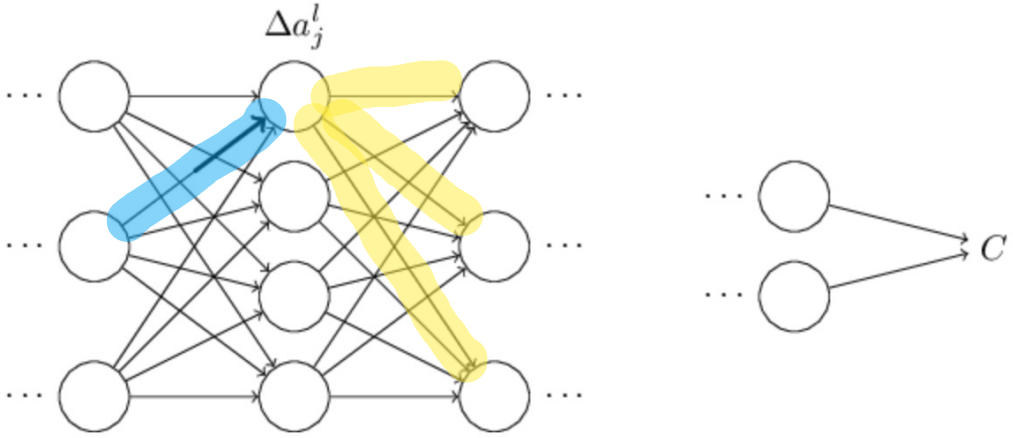
$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j \left(y_i - a_j\right)^2$$

# Backpropagation: Error forward

- Suppose you made a small change to a weight

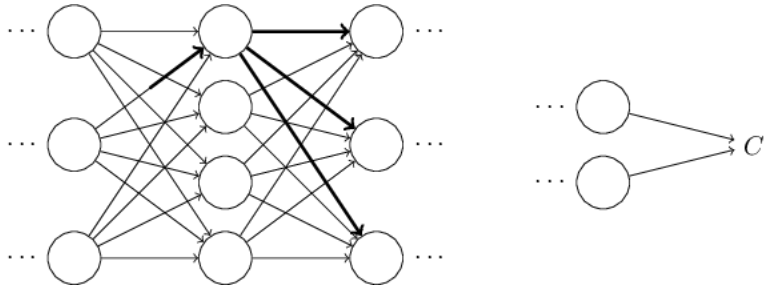



- Causes a change in the activation of the output neuron

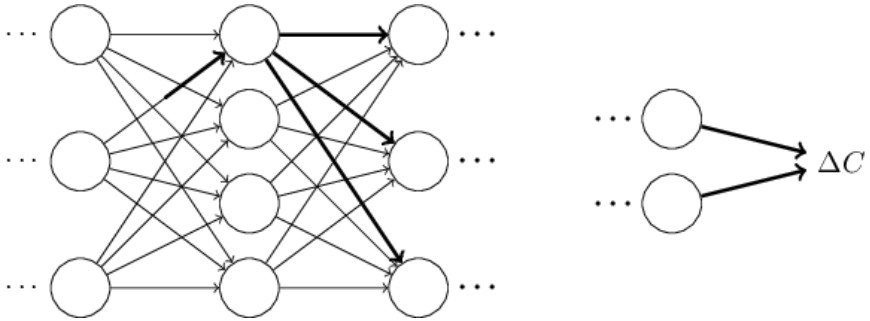


# Change propagates…

- Causes a change in all the activations in the next layer




- Then moves to the next layer, next layer …to the output and then the cost fxn




$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}.$$

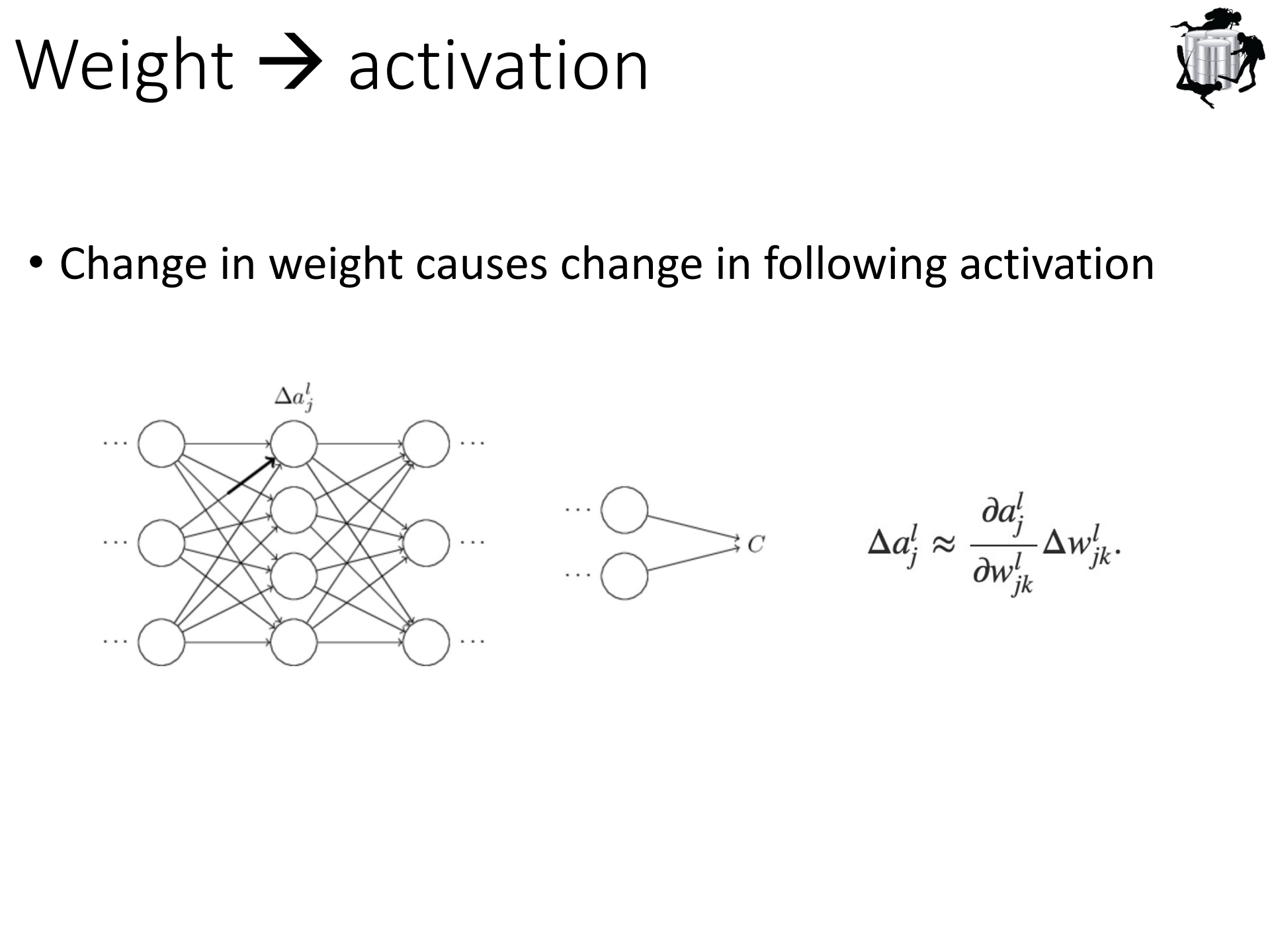
# Tracking the change

- This suggests that if we track how a weight change propagates to change the cost function then we can compute the gradient of the weight wrt cost

$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}.$$

- Lets try it!

# Weight → activation



- Change in weight causes change in following activation

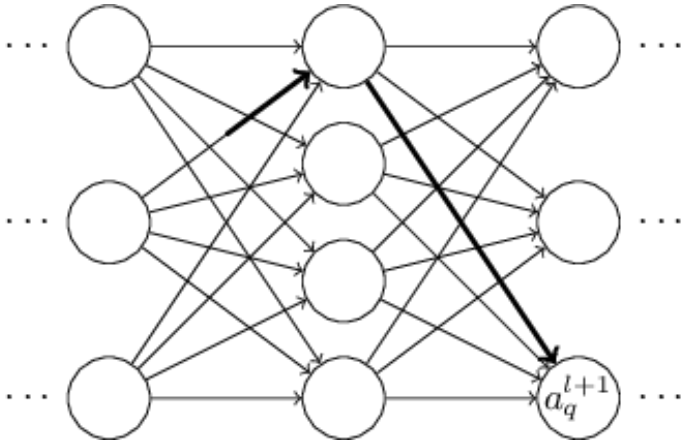


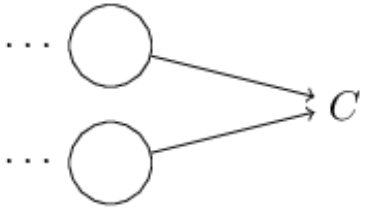




$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

# Activation→next layer Activation




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

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

# Next activation→Cost

Path through layers: $a_j^l, a_q^{l+1}, \ldots, a_n^{L-1}, a_m^L$.

$$\Delta C \approx \frac{\partial C}{\partial a_m^L}\frac{\partial a_m^L}{\partial a_n^{L-1}}\frac{\partial a_n^{L-1}}{\partial a_p^{L-2}}\cdots\frac{\partial a_q^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial w_{jk}^l}\Delta w_{jk}^l,$$

Summing over all paths:

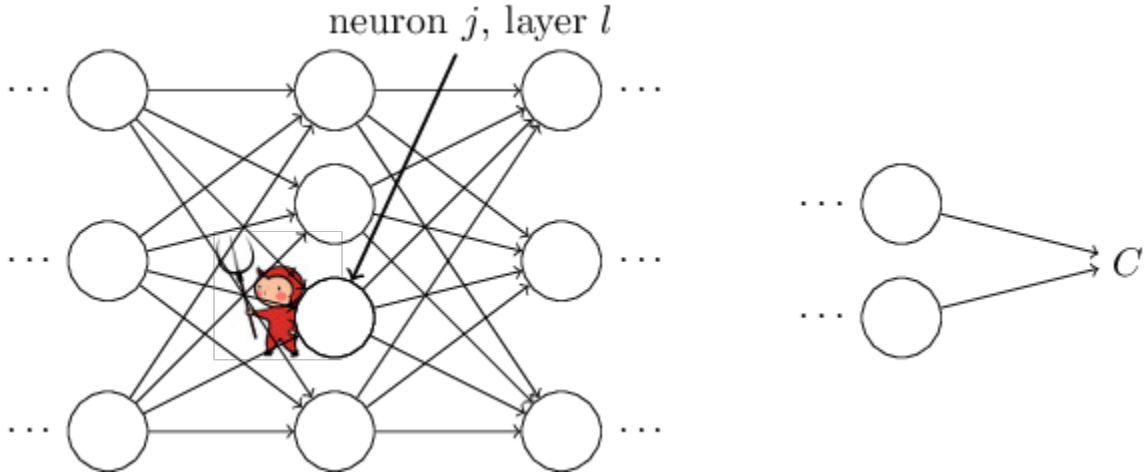
$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\ldots q}\frac{\partial C}{\partial a_m^L}\frac{\partial a_m^L}{\partial a_n^{L-1}}\frac{\partial a_n^{L-1}}{\partial a_p^{L-2}}\cdots\frac{\partial a_q^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial w_{jk}^l}$$

Already computed proceeding backwards

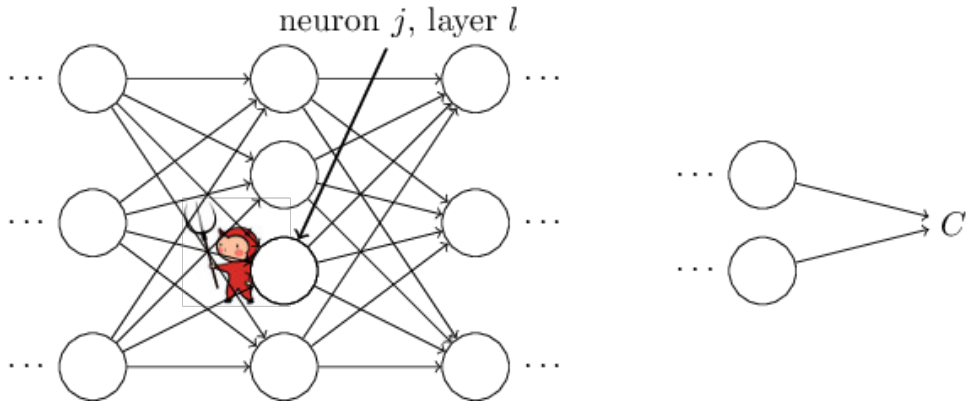
# Preliminary: Error at the node

- 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 $\frac{\partial C}{\partial b_j^l}$

express $\delta_j^l$ as a function of $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$



# Error at the node




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

# 1st Equation Error at the Output

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

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

$\sigma'$ is derivative of $\sigma$ function

# 2nd eq error at $\delta_j^l$ (jth node, lth layer)

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

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

kth node, next layer

- Differentiating gives

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

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

# 3rd equation error of a bias

- Use multivariate chain rule to obtain:

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

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

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

# $4^{th}$ equation error of weight

- 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 4 fundamental equations of backpropagation

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

$s \odot t$ is the ***elementwise*** product of the vectors

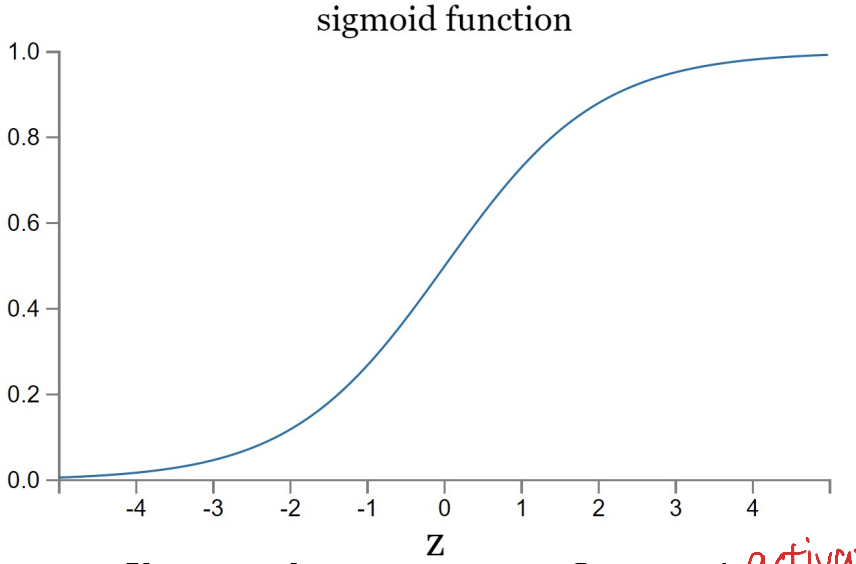
$$(s \odot t)_j = s_j t_j$$

Example:

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

# 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
  - $\Rightarrow \sigma'(z_j^L) \approx 0$ ⇒ gradient ≈0 reason: in backpropagation
  - ⇒ weight in final layer will learn slowly if output neuron is ***saturated*** (i.e. $\approx 0$ or $\approx 1$)

will always be multiplied activations in later layers

saturated gradient problem

if weights are close to input, the effect < weights close to output

# 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 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 $l = 2,3, \dots, L$, 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 $l = L-1, L-2, \dots, 2$ 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 $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}\delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

# 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

{ SGD with momentum
  2nd order hashing GD

# 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 $l = 2,3, \dots, L$ 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 $l = L-1, L-2, \dots, 2$ 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 $l = L, L-1, \dots, 2$ update weights $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} \left(a^{x,l-1}\right)^T$ and the biases $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

# 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:

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

- Why is it slow?
- Suppose we have a million weights in our network
  - For each weight $w_j$ we compute $C\left(w + \epsilon e_j\right)$
  - 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

a sweep through network

# Further reading

- Nielsen book, chapter 1, 2
- Goodfellow et al., section 6.5

③ energy based NN :
use an energy function
try to minimize it

④ neural ODE solver, adjoint method

How could you train a network without backpropagation?

① genetic algo { take 2 solutions and combine
train 2 NN
use weights from another NN,
perterbate to involve one set of
solutions and merge them

② neural tangent kernel : weights don't go far from minimum,
suitable for a infinitly wide NN