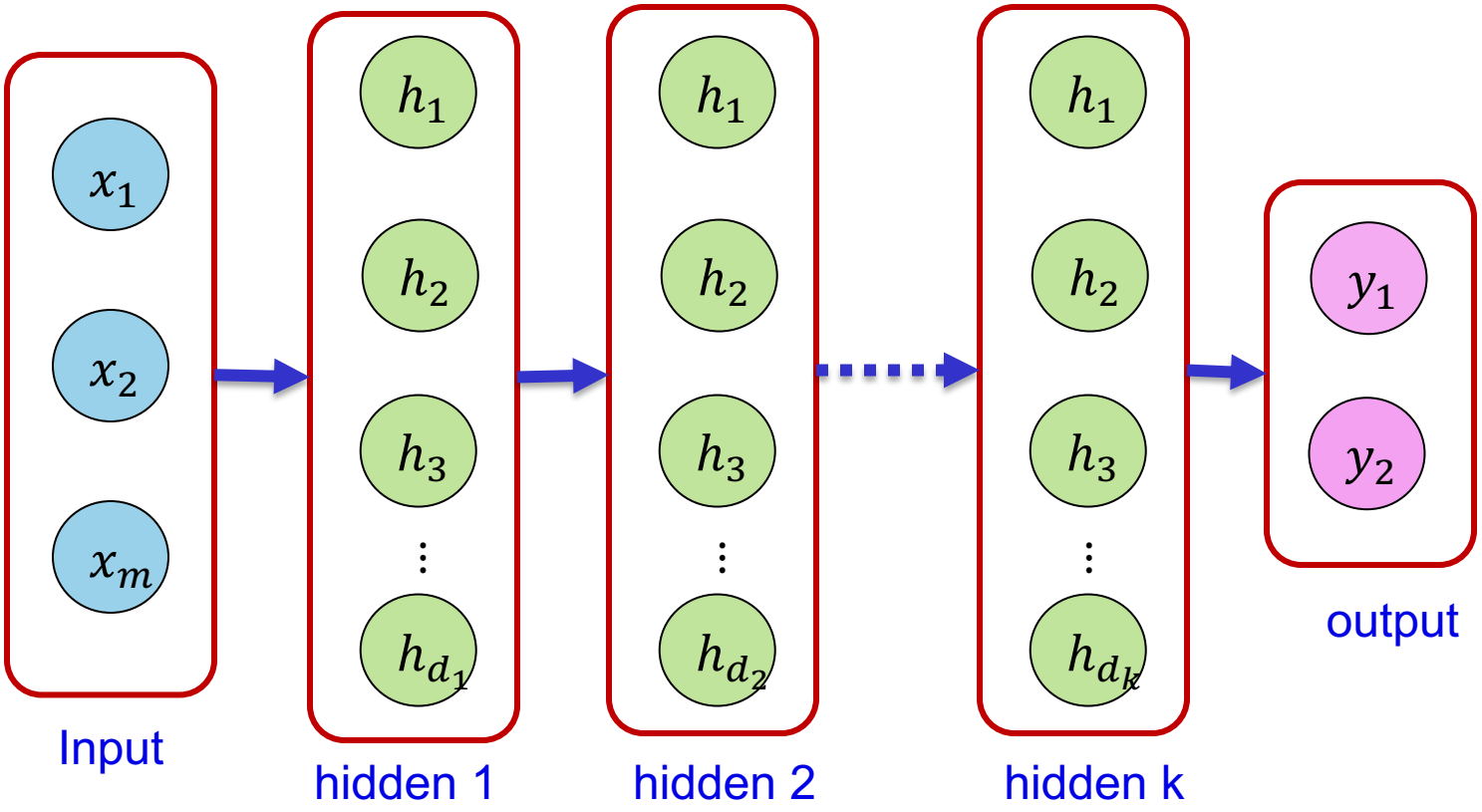
Previous Lesson: Multilayer Networks

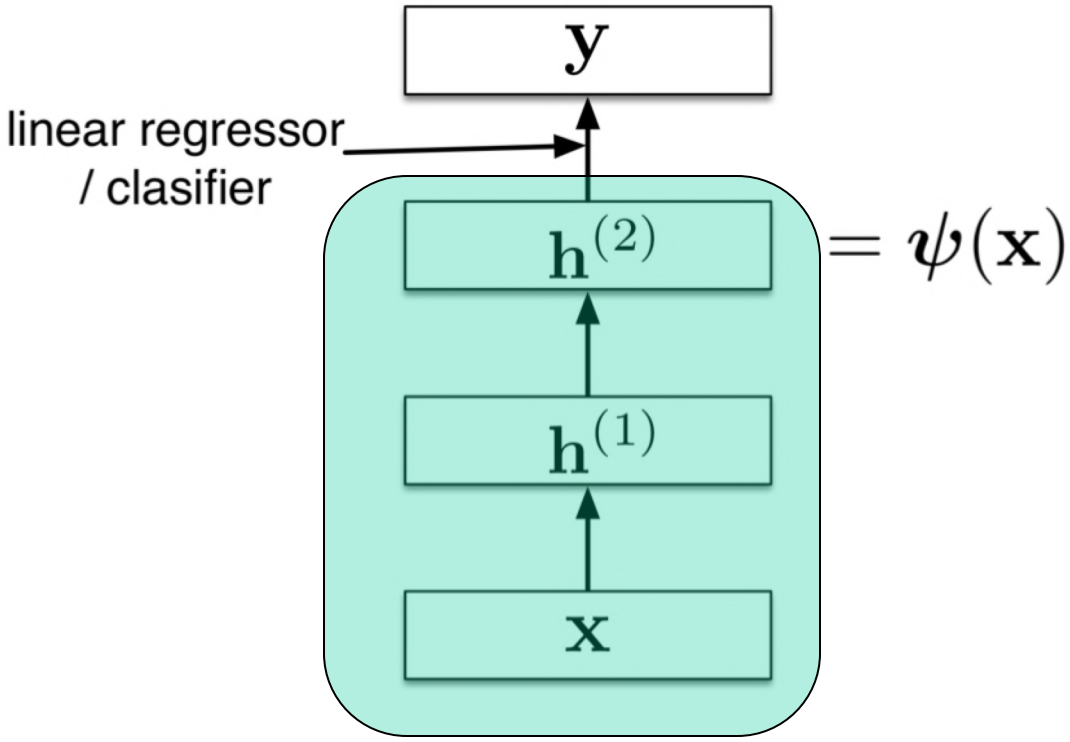


$$\boldsymbol{h}^{[1]} = f_1\left(W^{[1]}\boldsymbol{x}\right) = f_1\left(W^{[1]}\boldsymbol{h}^{[0]} + \boldsymbol{b}^{[1]}\right)$$

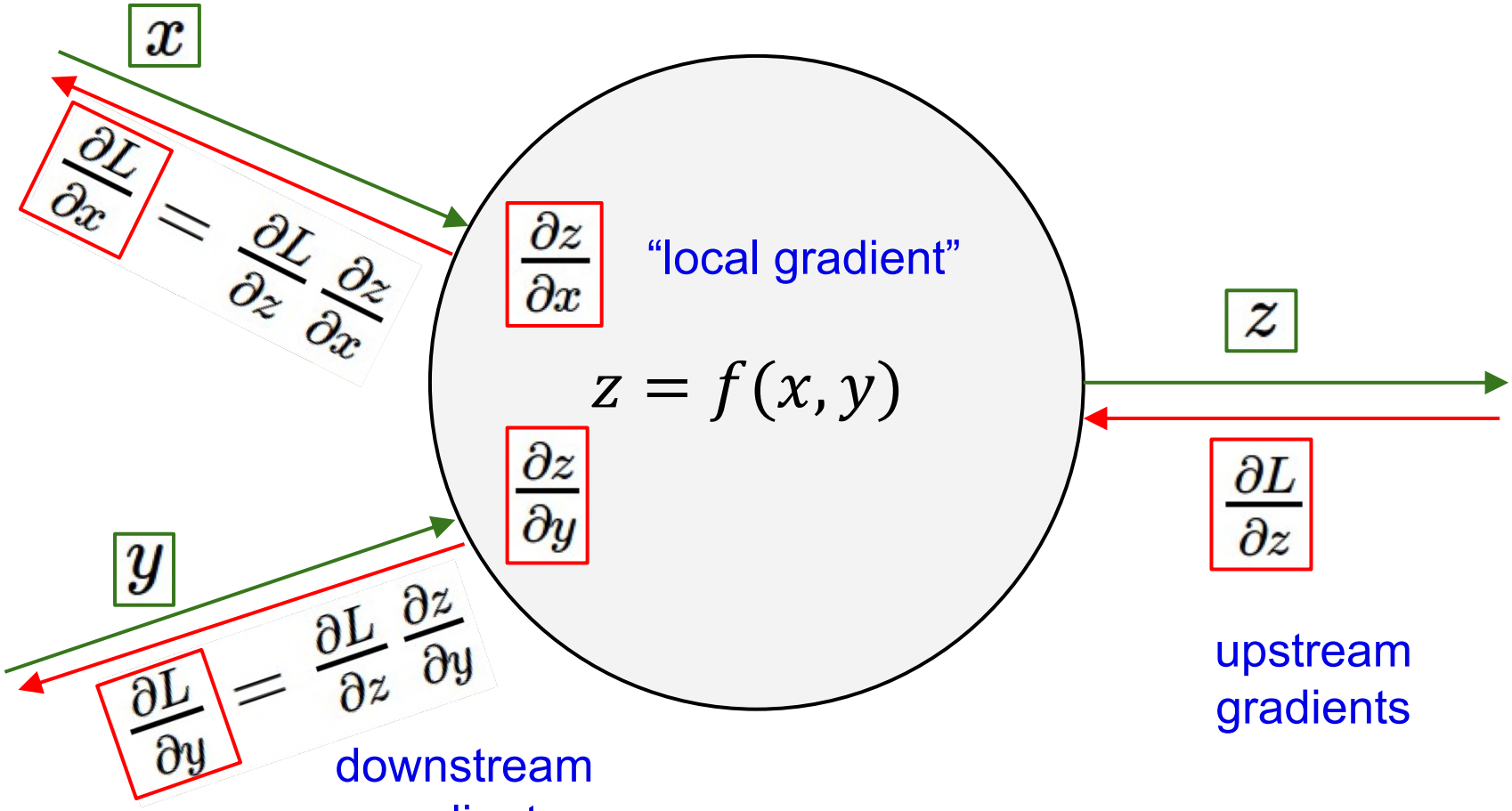
$$\boldsymbol{h}^{[i]} = f_i\left(W^{[i]}\boldsymbol{h}^{[i-1]} + \boldsymbol{b}^{[i]}\right) \qquad f_i \text{ – activation functions}$$

Previous Lesson: Feature Learning

- Multilayer networks can be viewed as an automatic way of learning features:

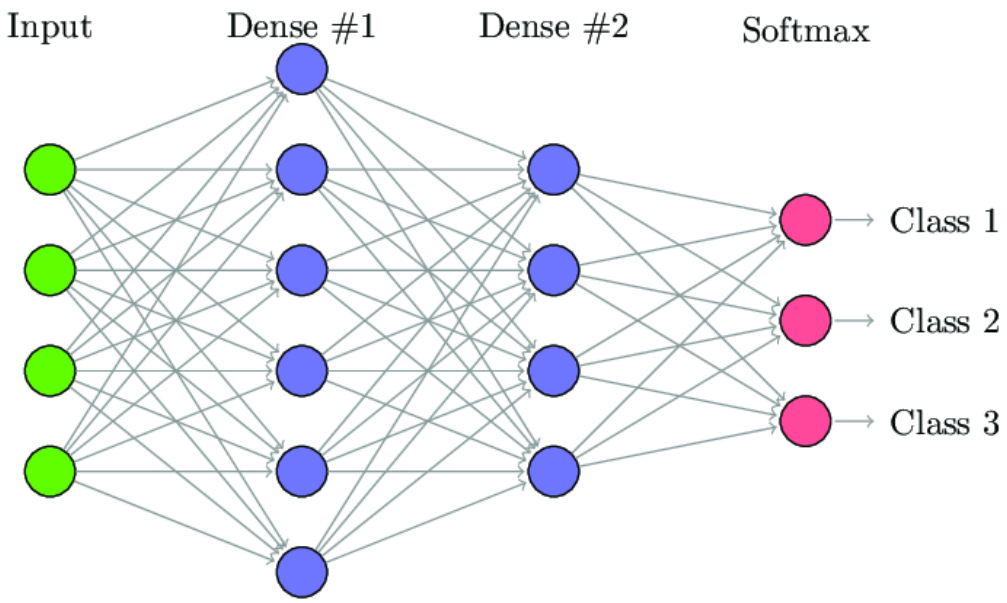


Previous Lesson: Backpropagation

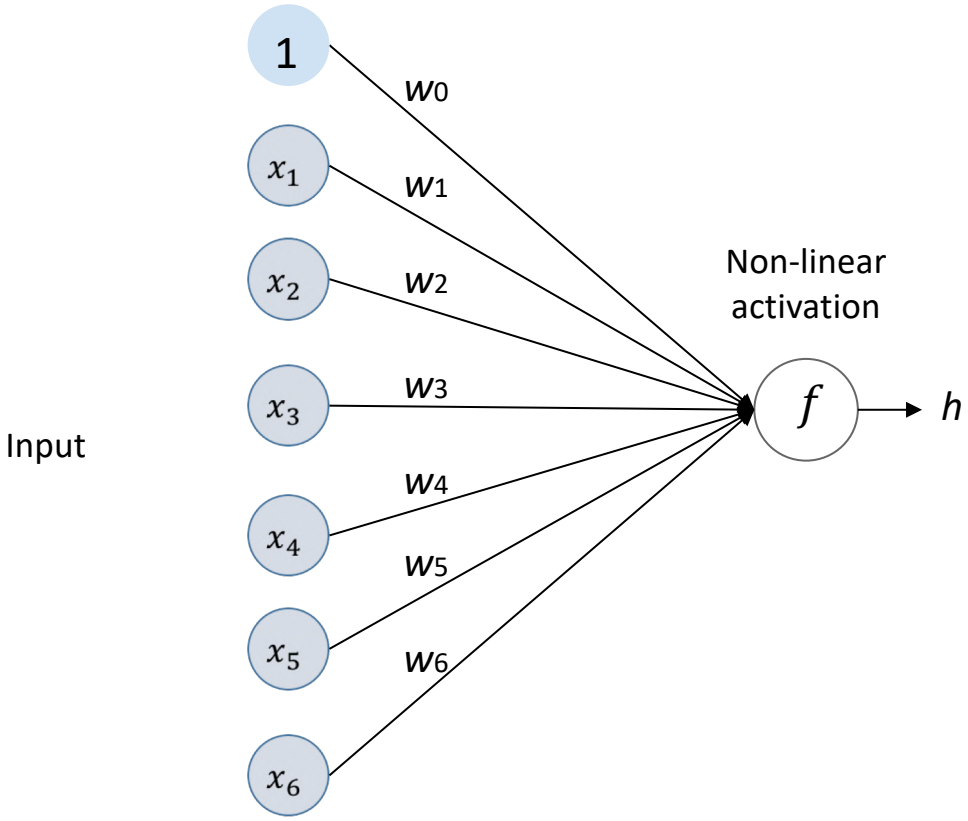


downstream gradient = upstream gradient * local gradient

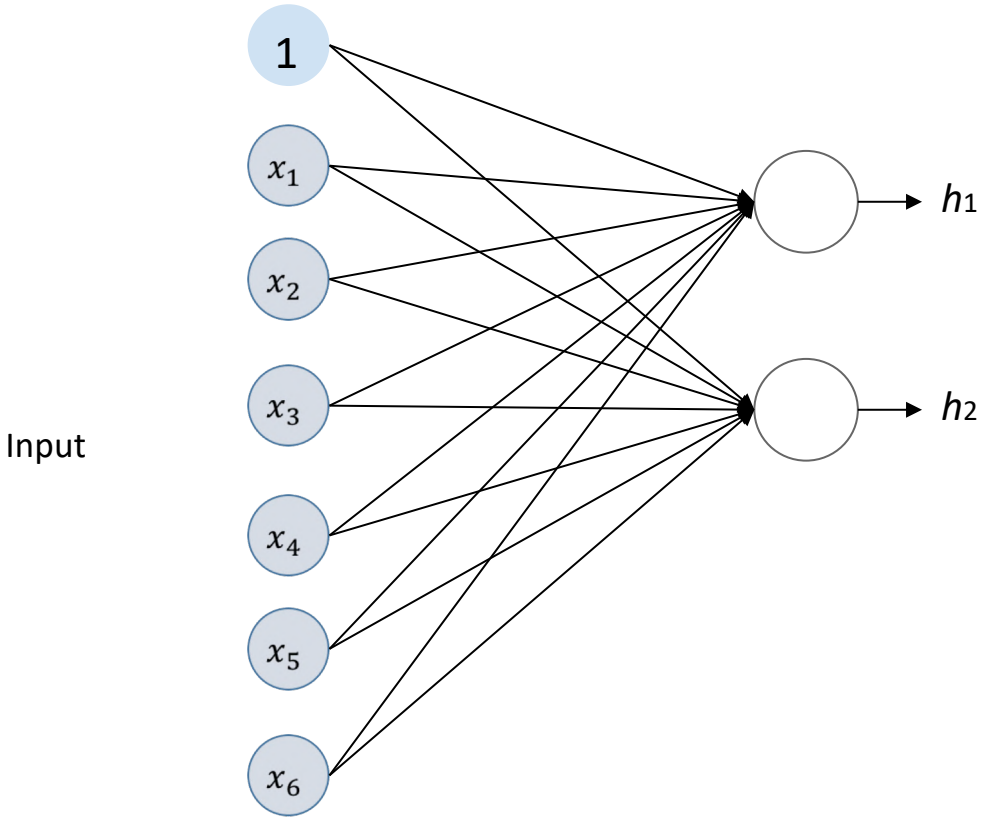
Tensor Representation



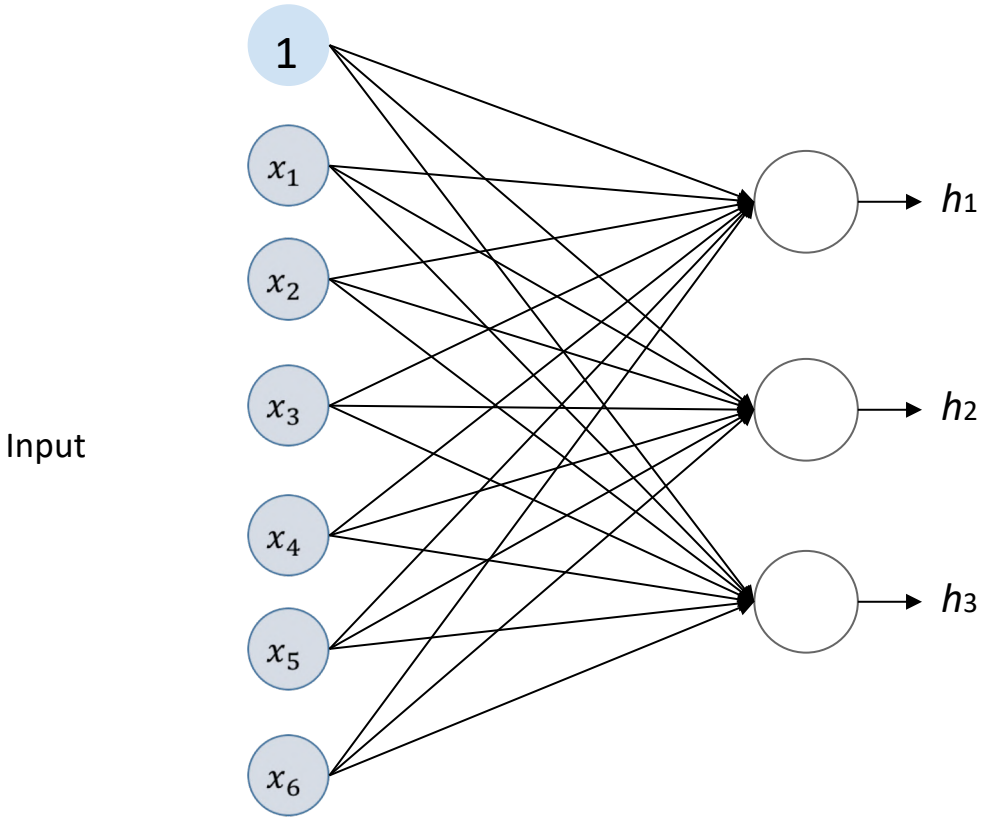
Neural Network Representation



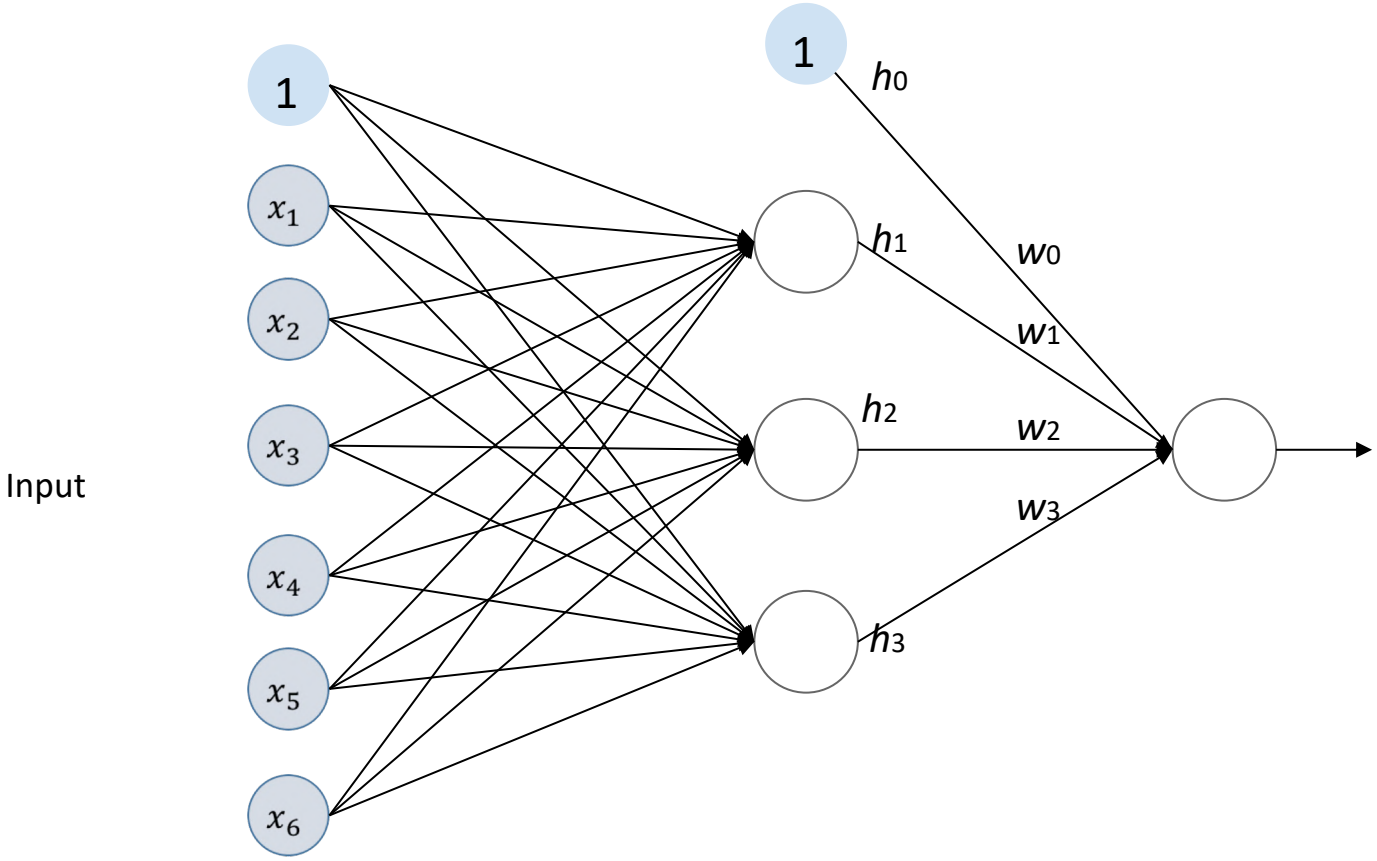
Neural Network Representation



Neural Network Representation

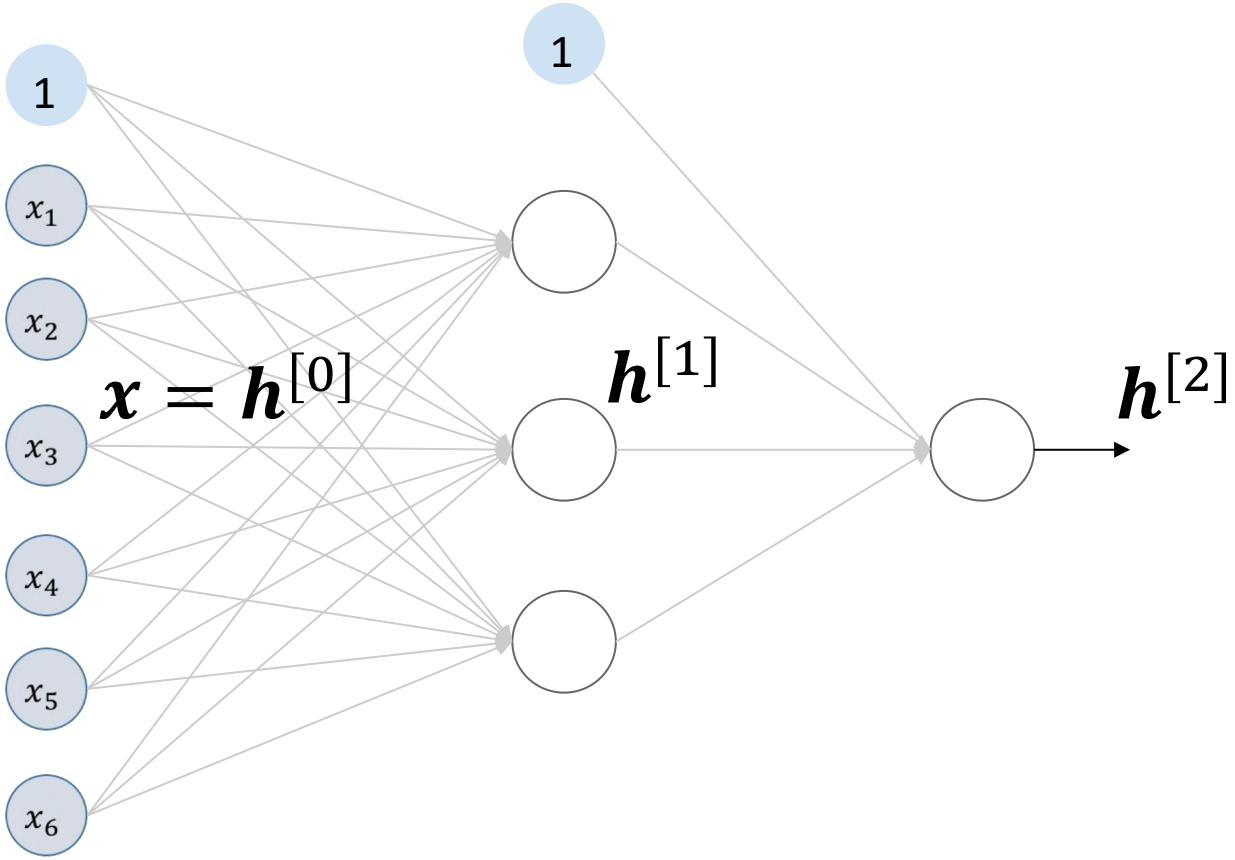


Neural Network Representation

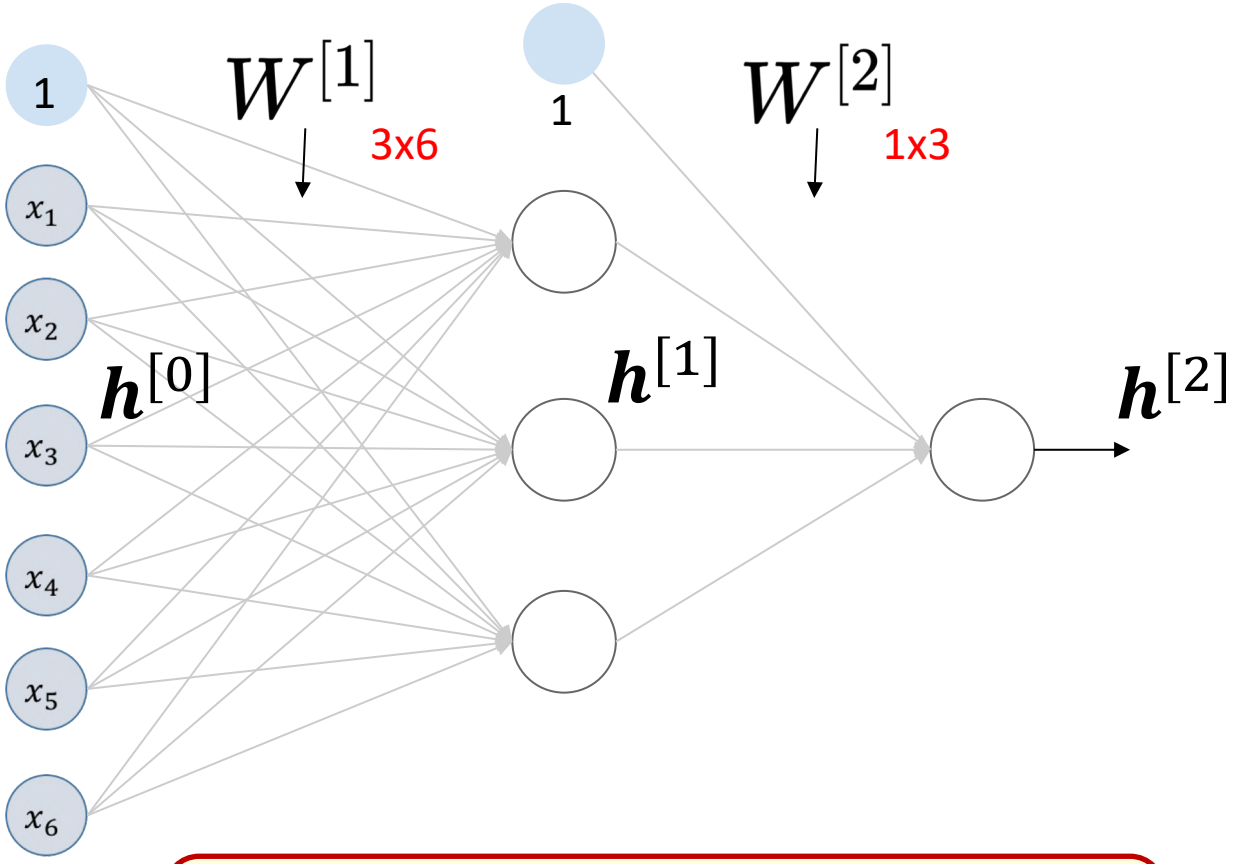


Neural Network Representation

Use superscripts [*]
to represent layers



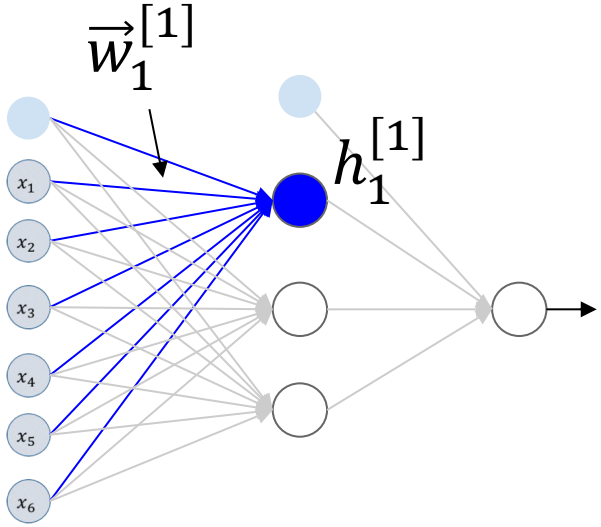
Neural Network Representation



$$\boldsymbol{h}^{[i]} = f_i\left(W^{[i]}\boldsymbol{h}^{[i-1]} + \boldsymbol{b}^{[i]}\right)$$

Forward pass for a given input

$$f\left[\begin{pmatrix} \leftarrow & w_1^{[1]} & \rightarrow \\ \leftarrow & w_2^{[1]} & \rightarrow \\ \leftarrow & w_3^{[1]} & \rightarrow \end{pmatrix}\begin{pmatrix} \uparrow \\ \boldsymbol{x} \\ \downarrow \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1^{[1]} \\ \boldsymbol{b}_2^{[1]} \\ \boldsymbol{b}_3^{[1]} \end{pmatrix}\right] = \begin{pmatrix} \boldsymbol{h}_1^{[1]} \\ \boldsymbol{h}_2^{[1]} \\ \boldsymbol{h}_3^{[1]} \end{pmatrix}$$

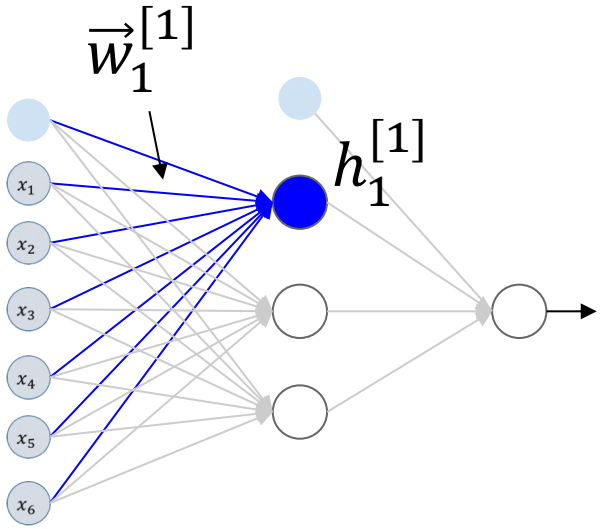


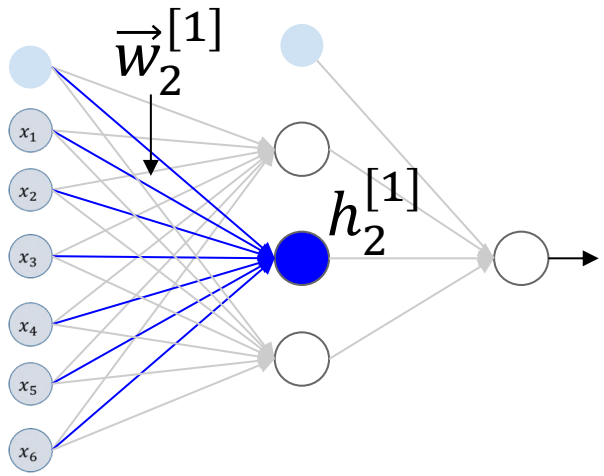
$$z_1^{[1]} = \boldsymbol{w}_1^{[1]T}\boldsymbol{x} + b_1^{[1]}$$

$$h_1^{[1]} = f\left(z_1^{[1]}\right)$$

Forward pass for a given input

$$f\left[\begin{pmatrix} \leftarrow & w_1^{[1]} & \rightarrow \\ \leftarrow & w_2^{[1]} & \rightarrow \\ \leftarrow & w_3^{[1]} & \rightarrow \end{pmatrix}\begin{pmatrix} \uparrow \\ \boldsymbol{x} \\ \downarrow \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1^{[1]} \\ \boldsymbol{b}_2^{[1]} \\ \boldsymbol{b}_3^{[1]} \end{pmatrix}\right] = \begin{pmatrix} \boldsymbol{h}_1^{[1]} \\ \boldsymbol{h}_2^{[1]} \\ \boldsymbol{h}_3^{[1]} \end{pmatrix}$$





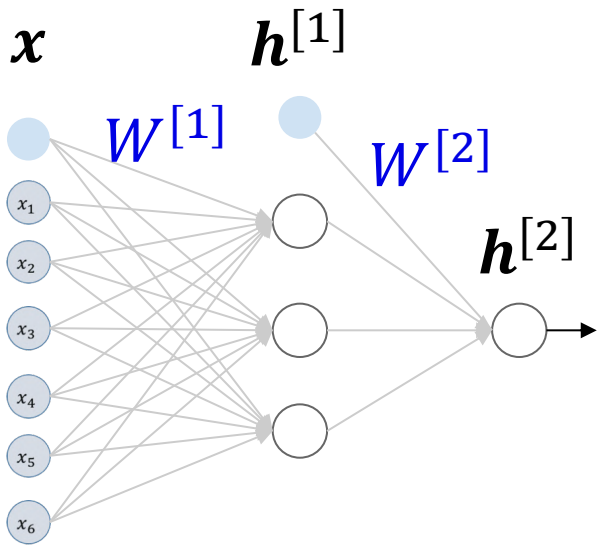
$$z_1^{[1]} = \boldsymbol{w}_1^{[1]T}\boldsymbol{x} + b_1^{[1]}$$

$$h_1^{[1]} = f\left(z_1^{[1]}\right)$$

$$z_2^{[1]} = \boldsymbol{w}_2^{[1]T}\boldsymbol{x} + b_2^{[1]}$$

$$h_2^{[1]} = f\left(z_2^{[1]}\right)$$

Forward pass for a given input



$$\boldsymbol{h}^{[1]} = f\left(W^{[1]}\,\boldsymbol{x} + \boldsymbol{b}^{[1]}\right)$$

3x6 6X1 3X1

$$\boldsymbol{h}^{[2]} = f\left(W^{[2]}\,\boldsymbol{h}^{[1]} + \boldsymbol{b}^{[2]}\right)$$

1x3 3X1 1X1

Sometimes it is written in a row form:

$$\boldsymbol{h}^{[1]T} = f\left(\boldsymbol{x}^T W^{[1]T} + \boldsymbol{b}^{[1]T}\right)$$

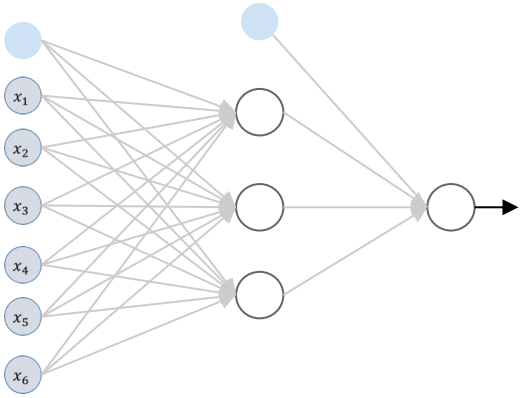
$$\boldsymbol{h}^{[2]T} = f\left(\boldsymbol{h}^{[1]T} W^{[2]T} + \boldsymbol{b}^{[2]T}\right)$$

Forward pass for multiple inputs

Use superscripts (*)
to represent samples

Input instances

$$X = \begin{pmatrix} | & | & | & | \\ x^{(0)} & x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | & | \end{pmatrix}$$



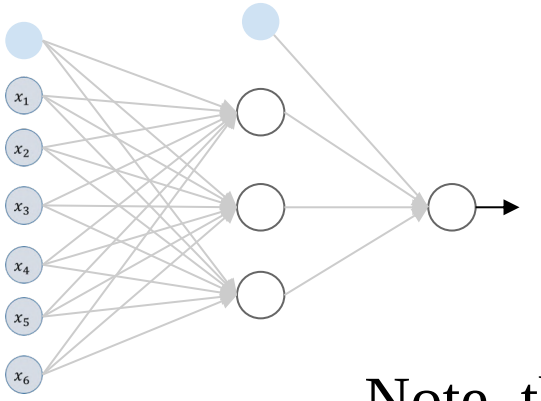
$$\boldsymbol{h}^{[1](0)} = f\left(W^{[1]}\boldsymbol{x}^{(0)} + \boldsymbol{b}^{[1]}\right)$$
$$\boldsymbol{h}^{[1](1)} = f\left(W^{[1]}\boldsymbol{x}^{(1)} + \boldsymbol{b}^{[1]}\right)$$
$$\boldsymbol{h}^{[1](2)} = f\left(W^{[1]}\boldsymbol{x}^{(2)} + \boldsymbol{b}^{[1]}\right)$$
$$\boldsymbol{h}^{[1](3)} = f\left(W^{[1]}\boldsymbol{x}^{(3)} + \boldsymbol{b}^{[1]}\right)$$

$$H^{[1]} = f\left(W^{[1]}\,X + \boldsymbol{b}^{[1]}\right)$$

Forward pass for multiple inputs

Input instances

$$X = \begin{pmatrix} | & | & | & | \\ x^{(0)} & x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | & | \end{pmatrix}$$



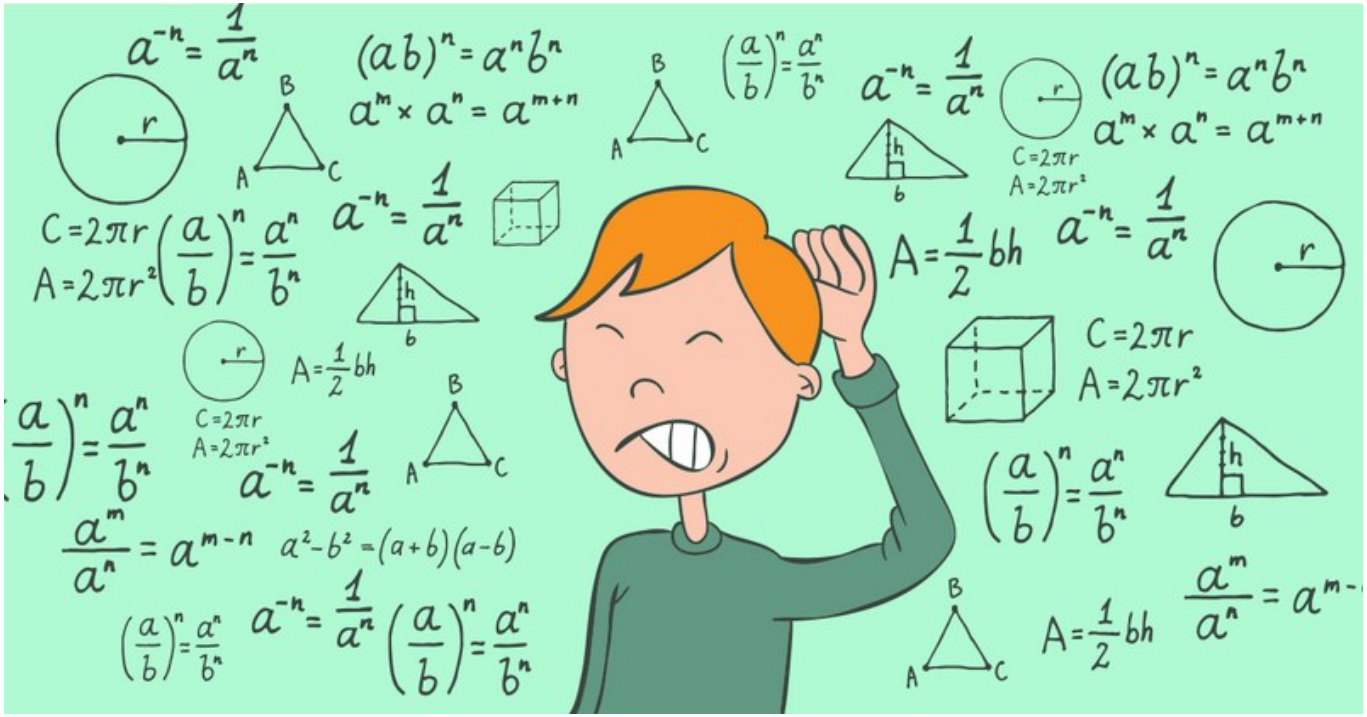
$$\overset{3\times4}{H^{[1]}} = f\left(\overset{3\times6}{W^{[1]}}\ \overset{6\times4}{X} + \overset{3\times4}{\boldsymbol{b}^{[1]}}\right)$$

$$\overset{1\times4}{H^{[2]}} = f\left(\overset{1\times3}{W^{[2]}}\ \overset{3\times4}{H^{[1]}} + \overset{1\times4}{\boldsymbol{b}^{[2]}}\right)$$

$$H^{[1]} = \begin{pmatrix} | & | & | & | \\ h^{[1](0)} & h^{[1](1)} & h^{[1](2)} & h^{[1](3)} \\ | & | & | & | \end{pmatrix}$$

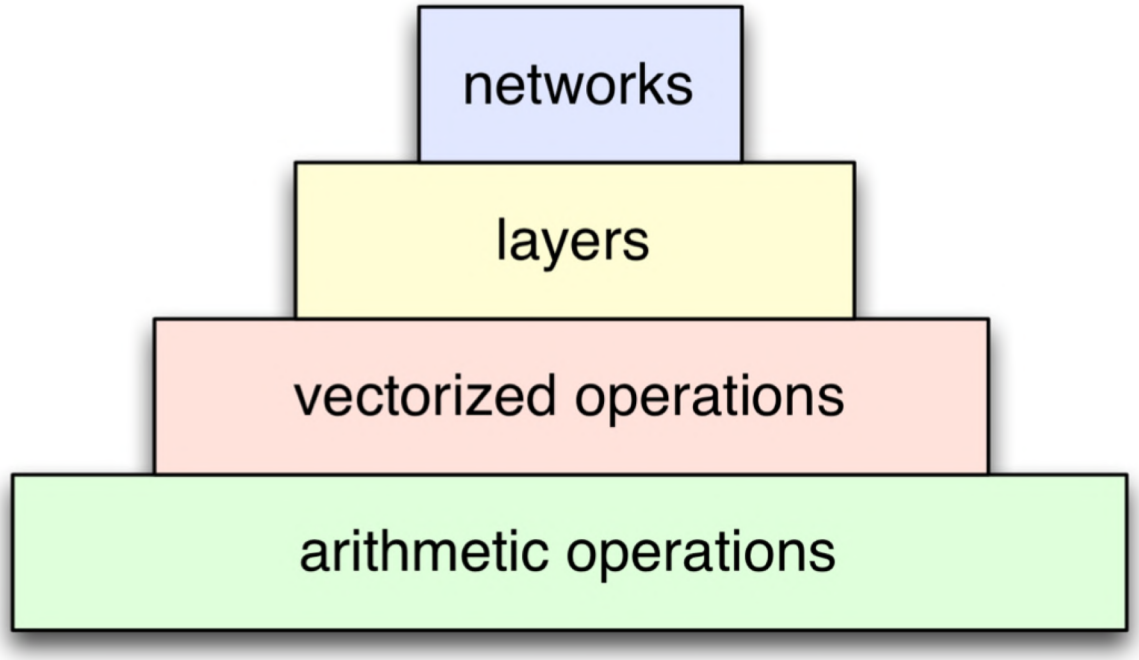
Note, the row form gives $H^{[1]T} = f\left(X^T\ W^{[1]T} + \boldsymbol{b}^{[1]T}\right)$

Backpropagation with Tensors



Levels of Abstractions

- We need to think at multiple levels of abstraction.
- So far, we've studied BP at the level of arithmetic operations.
- But we can apply BP at the level of vectorized operations.



Data Types

3 types of mathematical data objects:

Type	Scalar	Vector	Matrix
	x, y	$\mathbf{x}, \mathbf{y}$	X, Y

Vector (rank 1 array)

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

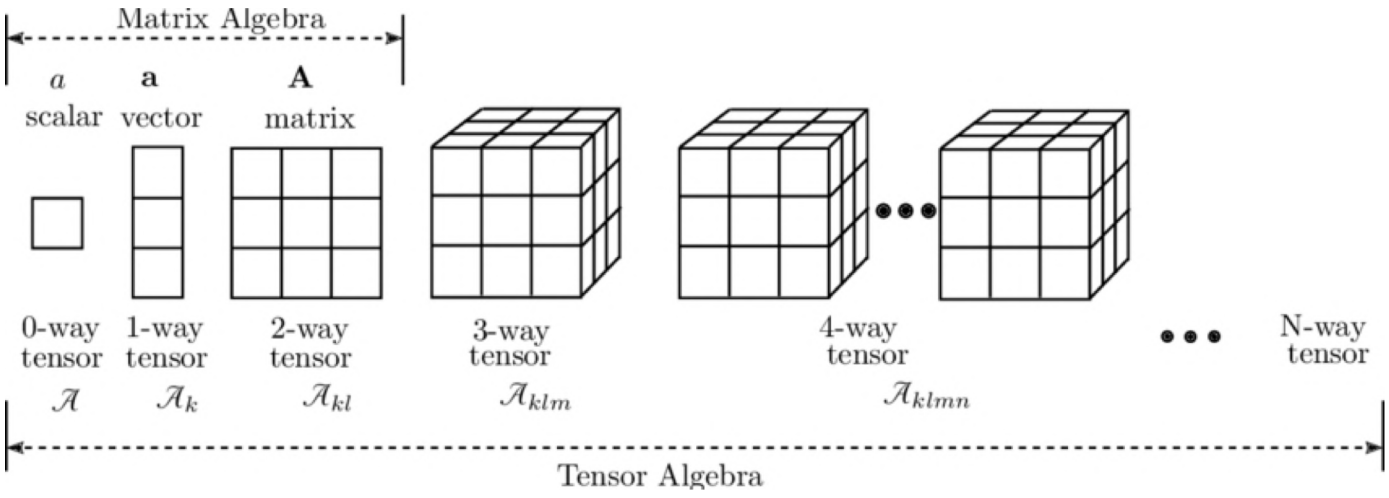
Matrix (rank 2 array)

$$Y = \begin{bmatrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \cdots & y_{2n} \\ \vdots & \vdots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix}$$

Tensor

A tensor is a generalized matrix:

- Scalar is a 0-D tensor
- Vector is a 1-D tensor
- Matrix is a 2-D tensor
- Tensor is a multi-dimensional matrix.



Tensor Derivatives

There are 6 common types of matrix derivatives:

Derivative ↓	Scalar	Vector	Matrix
Scalar	$\frac{\partial y}{\partial x}$	$\frac{\partial \mathbf{y}}{\partial x}$	$\frac{\partial \mathrm{Y}}{\partial x}$
Vector	$\frac{\partial y}{\partial \mathbf{x}}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$	
Matrix	$\frac{\partial y}{\partial \mathrm{X}}$		

Derivatives Rules

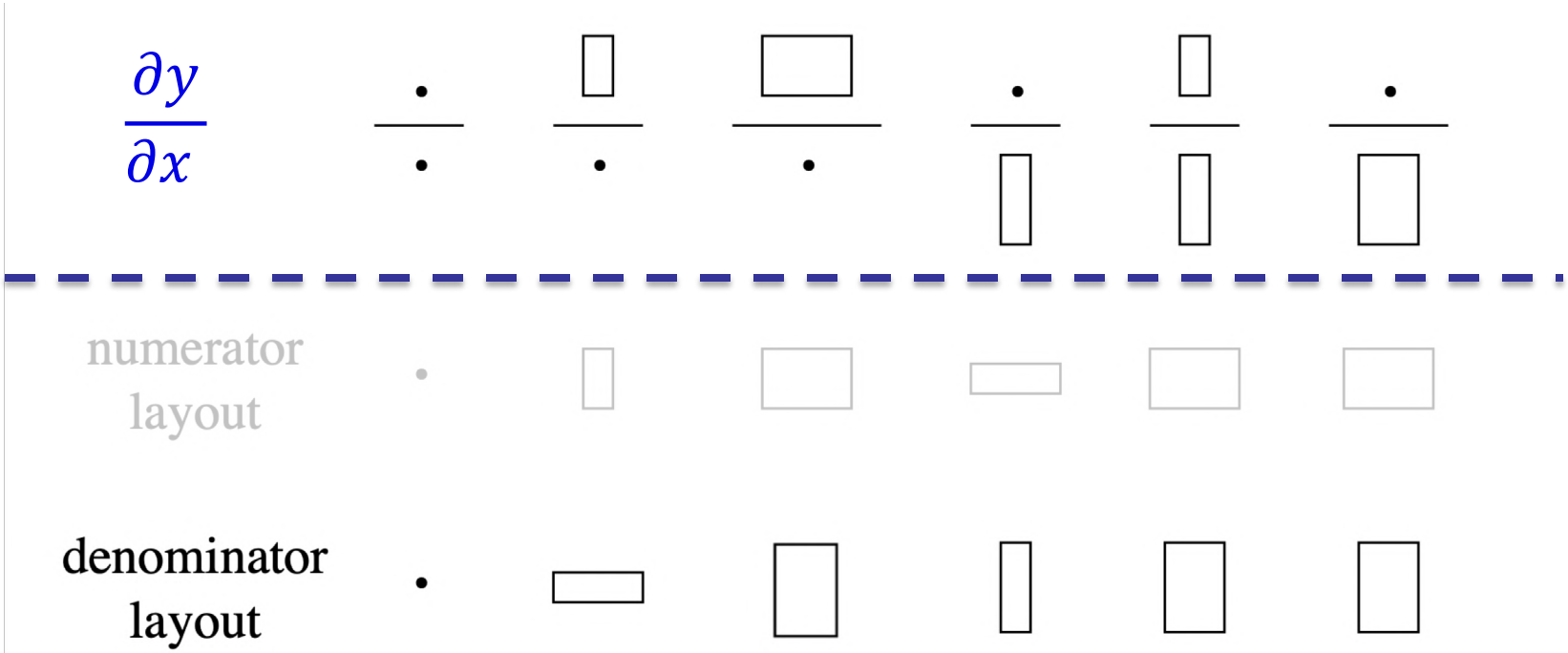
(SS1) $$\frac{\partial(u+v)}{\partial x} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial x}$$

(SS2) $$\frac{\partial uv}{\partial x} = u\frac{\partial v}{\partial x} + v\frac{\partial u}{\partial x}$$ (product rule)

(SS3) $$\frac{\partial g(u)}{\partial x} = \frac{\partial g(u)}{\partial u}\frac{\partial u}{\partial x}$$ (chain rule)

(SS4) $$\frac{\partial f(g(u))}{\partial x} = \frac{\partial f(g)}{\partial g}\frac{\partial g(u)}{\partial u}\frac{\partial u}{\partial x}$$ (chain rule)

Dimensionality of Derivatives



Denominator layout : when you take a derivative w.r.t. a **scalar**, the dimensions swap: $X \in R^{m \times n} \implies \frac{dX}{ds} \in R^{n \times m}$

- When you take a derivative of a **scalar** w.r.t. **matrix** the dimensions do not swap.

Derivatives by Scalar

(Denominator Layout Notation)

- Scalar by scalar : $\frac{\partial y}{\partial x}$ $y \in \mathbb{R}, x \in \mathbb{R}$

- Vector by scalar : $\frac{\partial \mathbf{y}}{\partial x} = \left[\frac{\partial y_1}{\partial x}, \cdots, \frac{\partial y_m}{\partial x}\right]$ $\boldsymbol{y} \in \mathbb{R}^m, x \in \mathbb{R}$

- Matrix by scalar: $\frac{\partial \mathrm{Y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \cdots & \frac{\partial y_{m1}}{\partial x} \\ \vdots & \cdots & \vdots \\ \frac{\partial y_{1n}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$ $\mathrm{Y} \in \mathbb{R}^{m \times n}, x \in \mathbb{R}$

Derivatives by Vector

- Scalar by vector: $\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$ $y \in \mathbb{R}, \ \boldsymbol{x} \in \mathbb{R}^n$

- Vector by vector:

$\boldsymbol{y} \in \mathbb{R}^m, \boldsymbol{x} \in \mathbb{R}^n$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \uparrow & & \uparrow \\ \frac{\partial y_1}{\partial \mathbf{x}} & \cdots & \frac{\partial y_m}{\partial \mathbf{x}} \\ \downarrow & & \downarrow \end{bmatrix}$$

Derivatives by Matrix

- Scalar by matrix:

$$\frac{\partial \mathrm{y}}{\partial X} = \begin{bmatrix} \frac{\partial y}{\partial \mathrm{x}_{11}} & \cdots & \frac{\partial y}{\partial \mathrm{x}_{1n}} \\ \vdots & & \vdots \\ \frac{\partial y}{\partial \mathrm{x}_{m1}} & \cdots & \frac{\partial y}{\partial \mathrm{x}_{mn}} \end{bmatrix}$$

$y \in \mathbb{R}$

$X \in \mathbb{R}^{mxn}$

- Note, that the original matrix was $m \times n$ matrix and the derivative provides an m$\times n$ matrix.

Vector Derivatives: Index Notations

$x \in \mathbb{R}, y \in \mathbb{R}$ | $\boldsymbol{x} \in \mathbb{R}^N, y \in \mathbb{R}$ | $\boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{y} \in \mathbb{R}^M$

Regular derivative: | Derivative is **Gradient**: | Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

$$\frac{\partial y}{\partial \boldsymbol{x}} \in \mathbb{R}^N$$

$$\left(\frac{\partial y}{\partial \boldsymbol{x}}\right)_i = \frac{\partial y}{\partial x_i}$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \in \mathbb{R}^{N \times M}$$

$$\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)_{i,j} = \frac{\partial y_j}{\partial x_i}$$

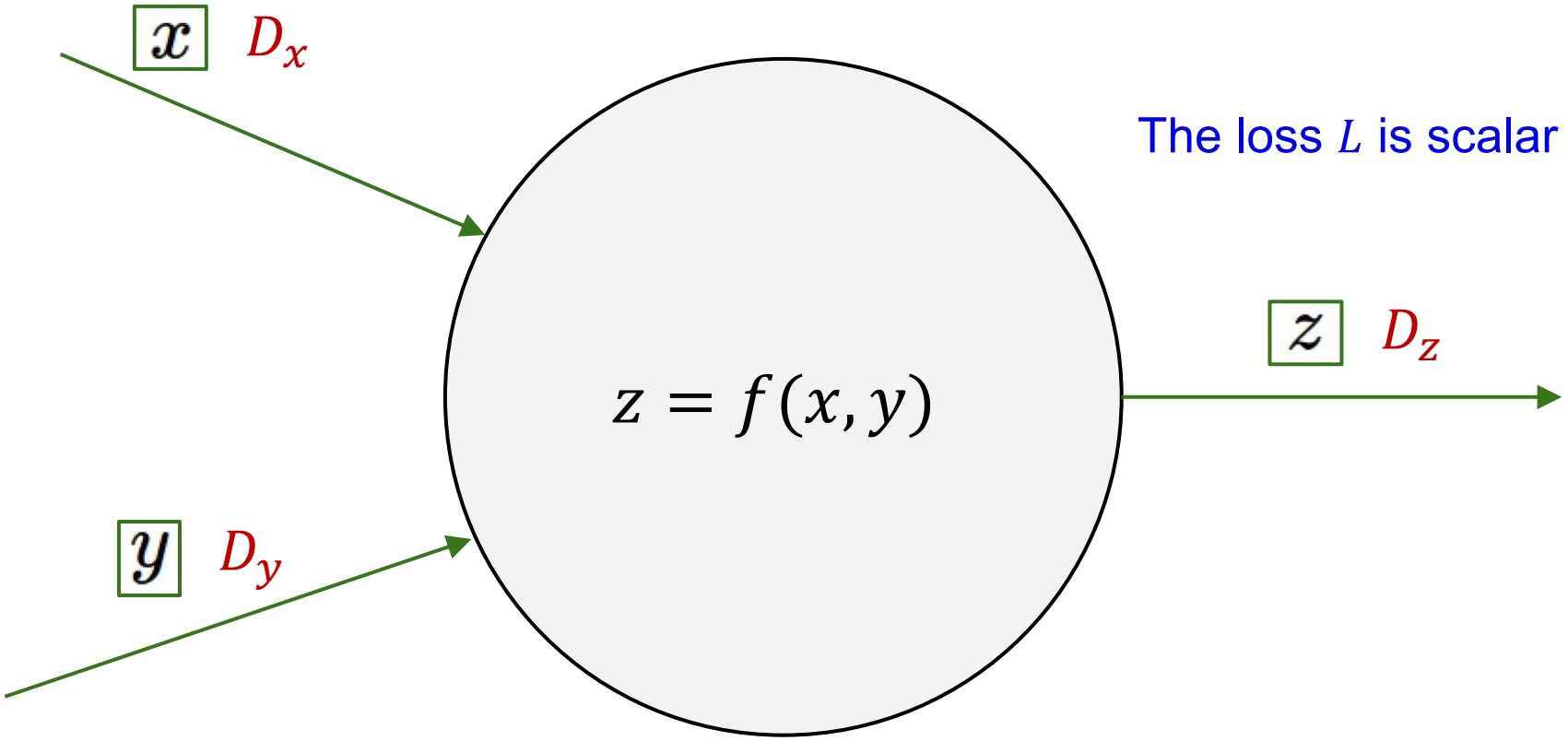
If x changes by a small amount, how much will y change?

For each x_i, if it changes by a small amount, how much will y change?

For each x_i and y_j, if x_i changes by a small amount, how much will y_j change?

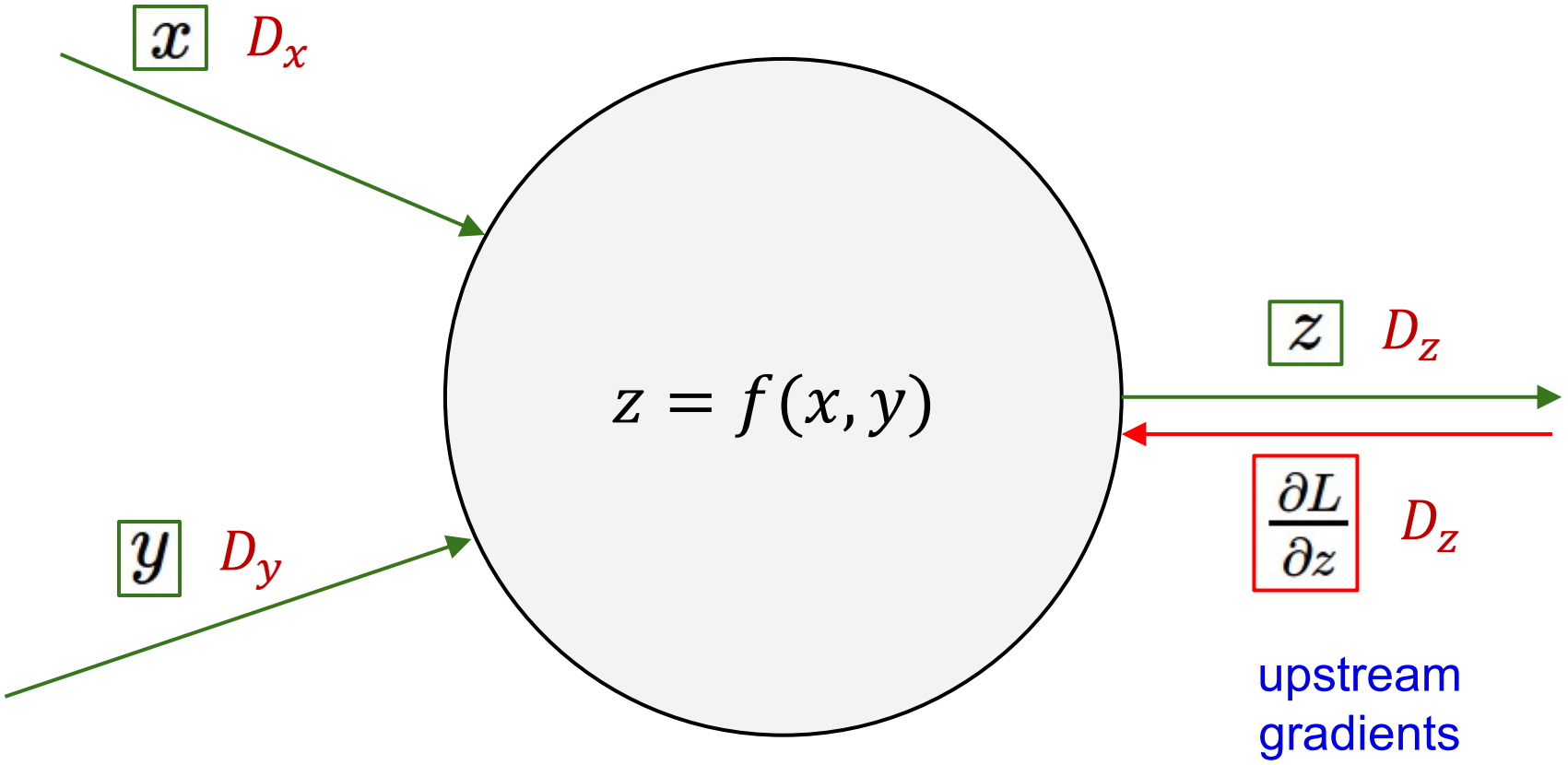
From: Justin Johnson

Backpropagation with Vectors



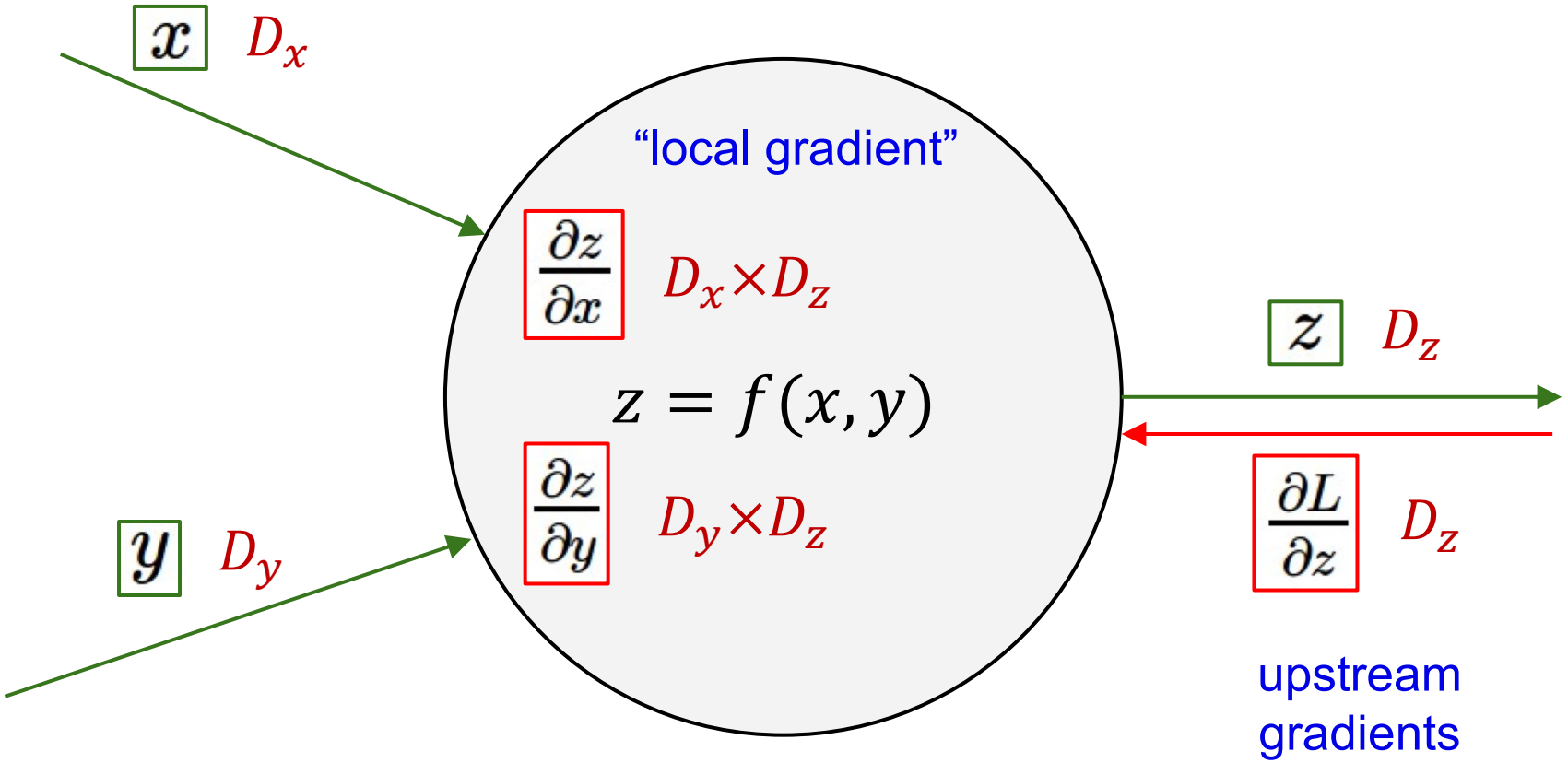
Source: Justin Johnson

Backpropagation with Vectors



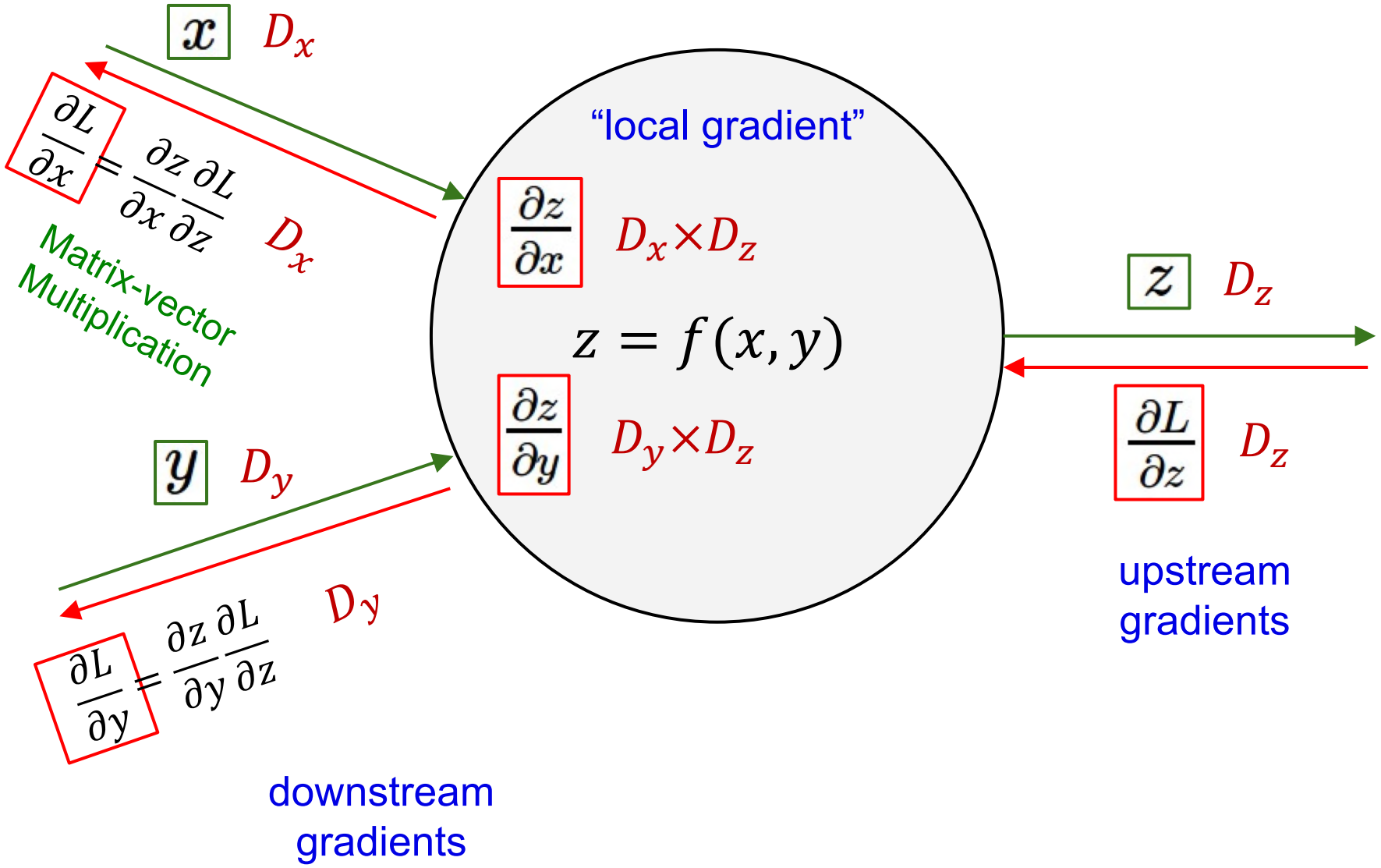
Source: Justin Johnson

Backpropagation with Vectors



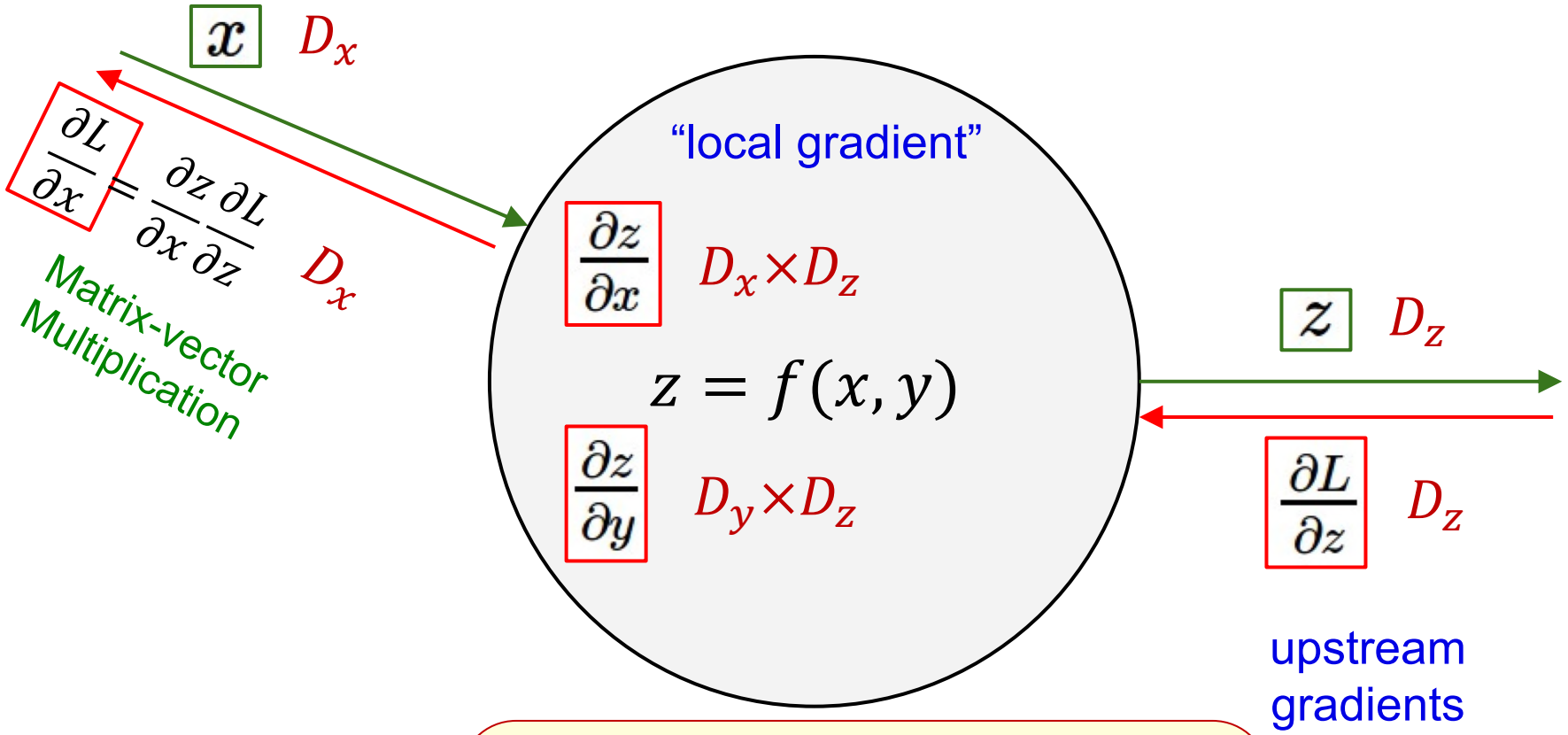
Source: Justin Johnson

Backpropagation with Vectors



Source: Justin Johnson

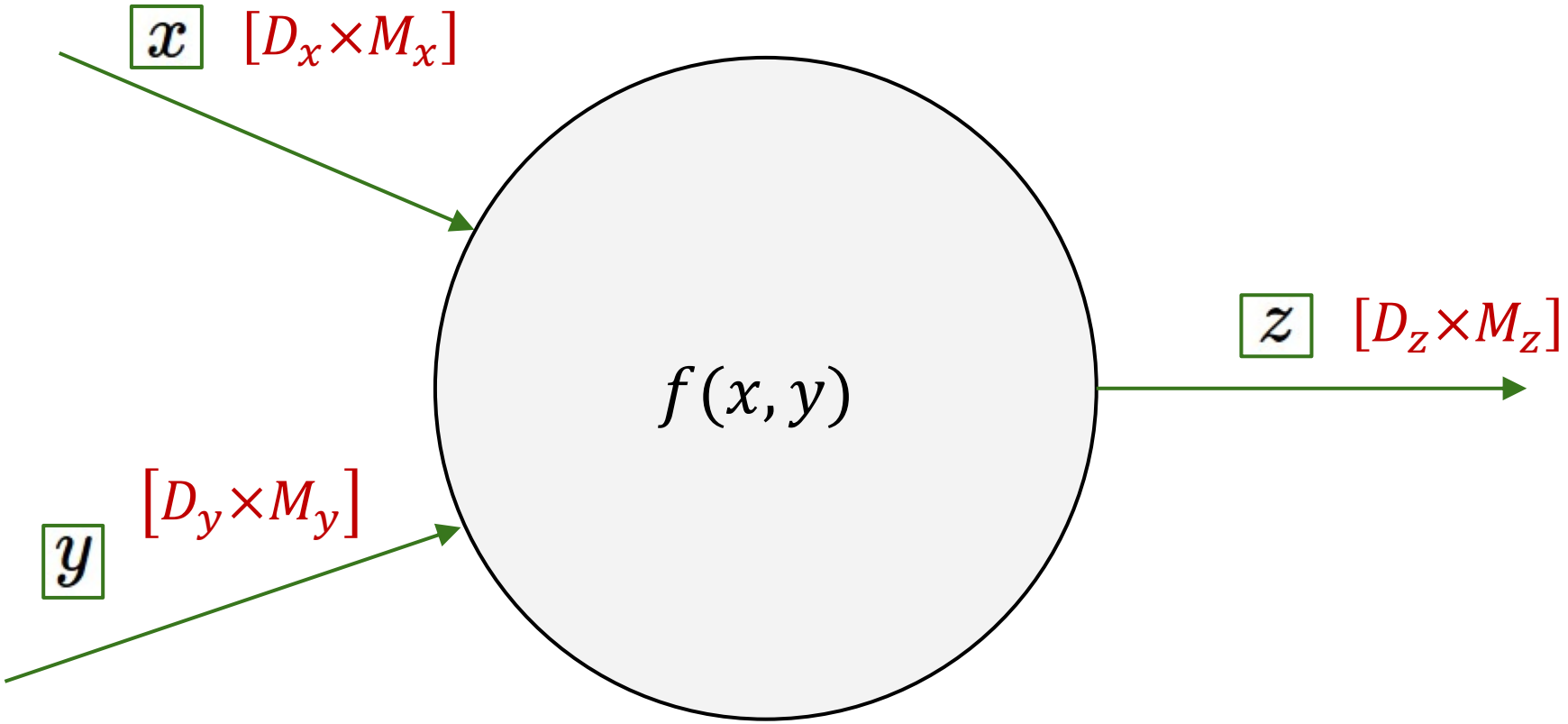
Backpropagation with Vectors



$$\left(\frac{\partial L}{\partial x}\right)_i = \sum_j \left(\frac{\partial z}{\partial x}\right)_{i,j} \left(\frac{\partial L}{\partial z}\right)_j$$

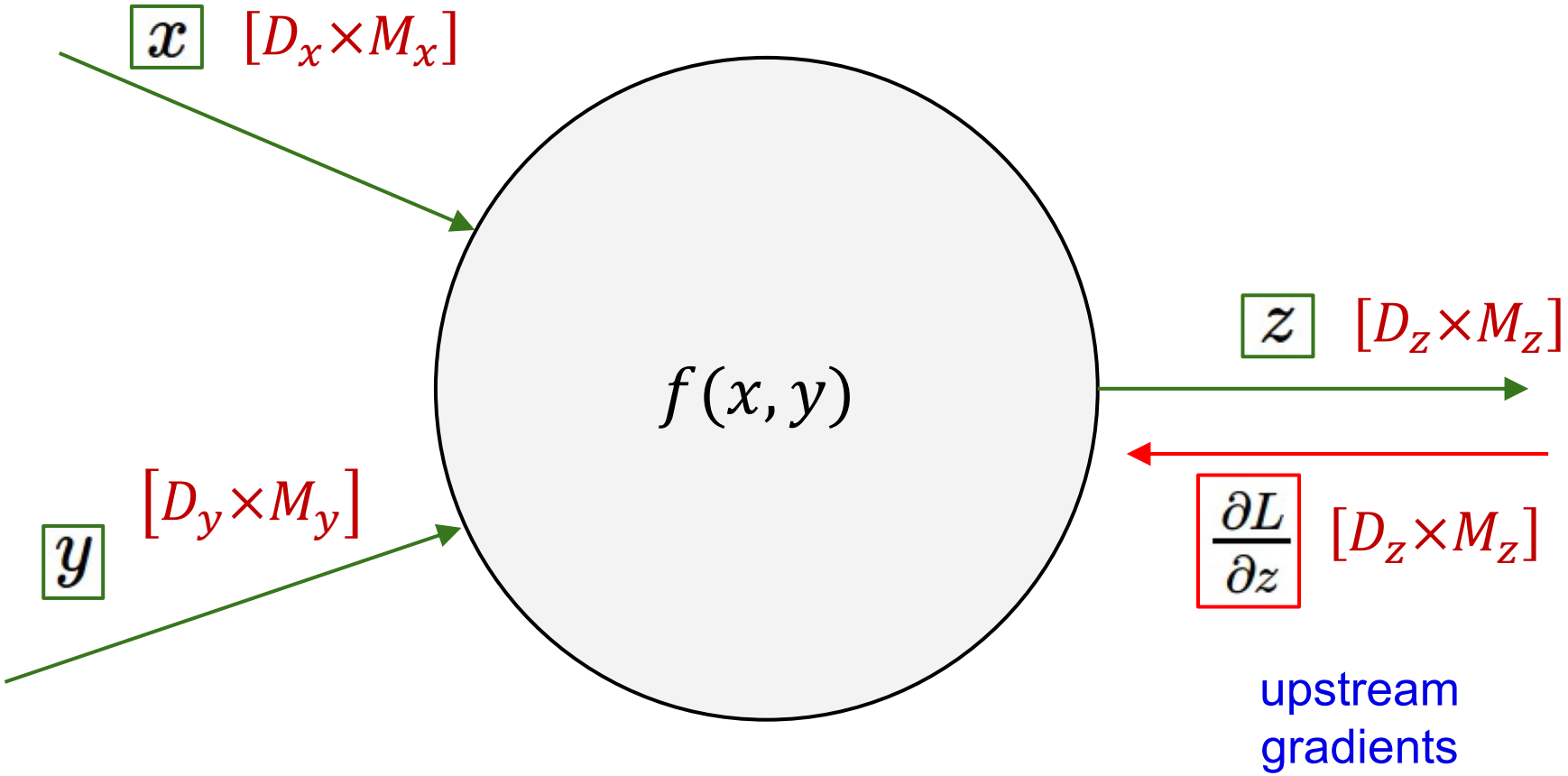
$$i \in [1..D_x], j \in [1..D_z]$$

Backpropagation with Matrices (Tensors)



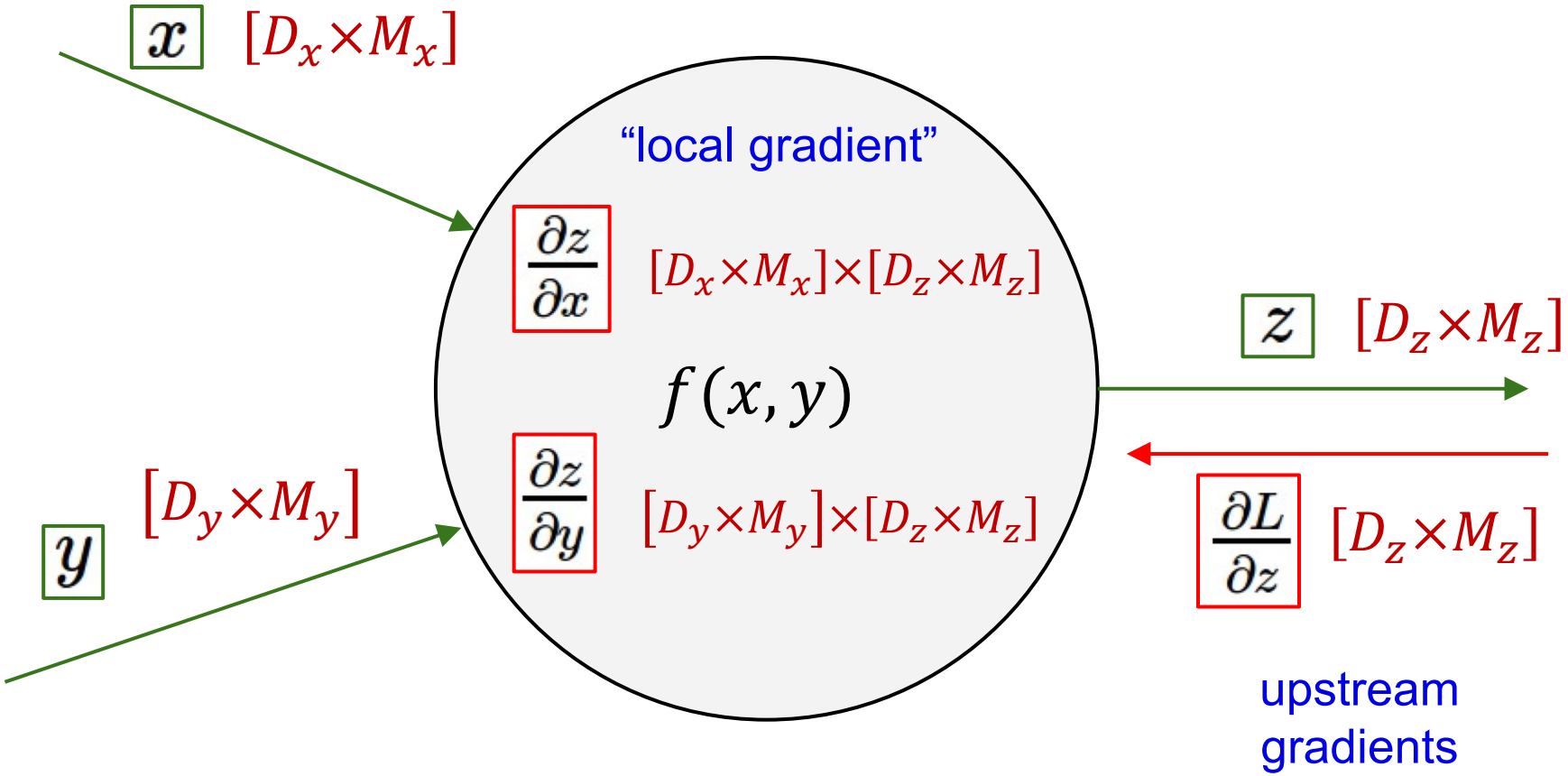
Source: Justin Johnson

Backpropagation with Matrices (Tensors)



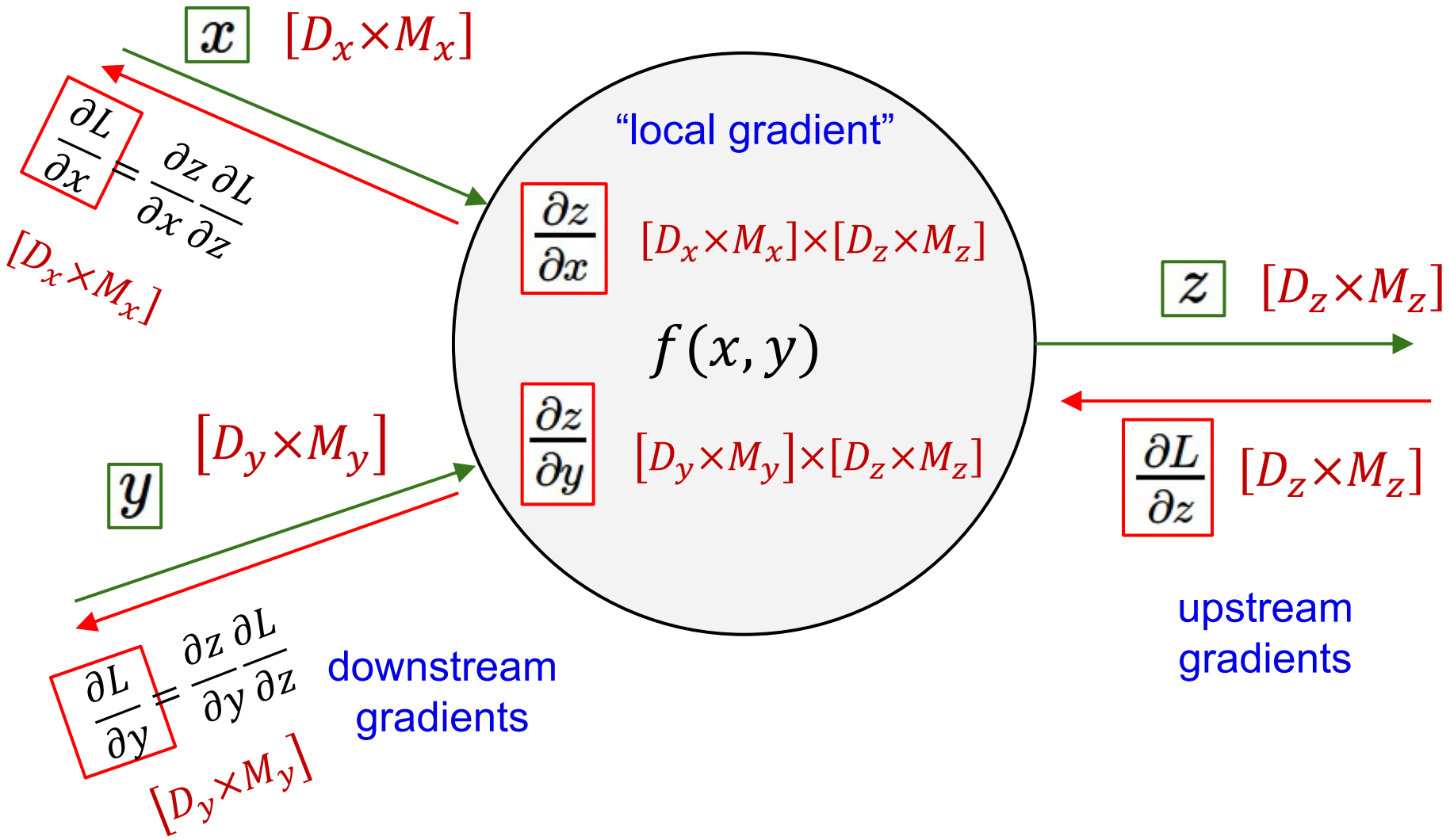
Source: Justin Johnson

Backpropagation with Matrices (Tensors)



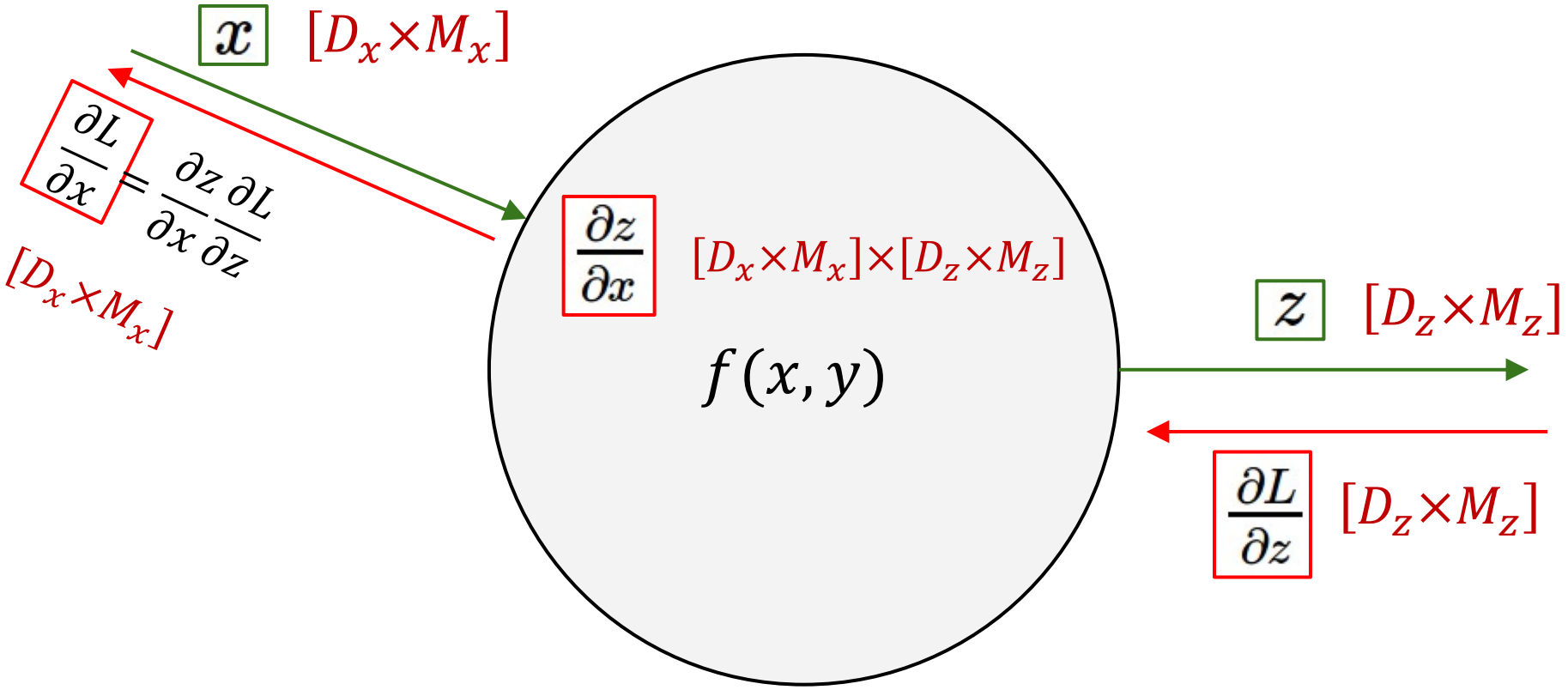
Source: Justin Johnson

Backpropagation with Matrices (Tensors)



Source: Justin Johnson

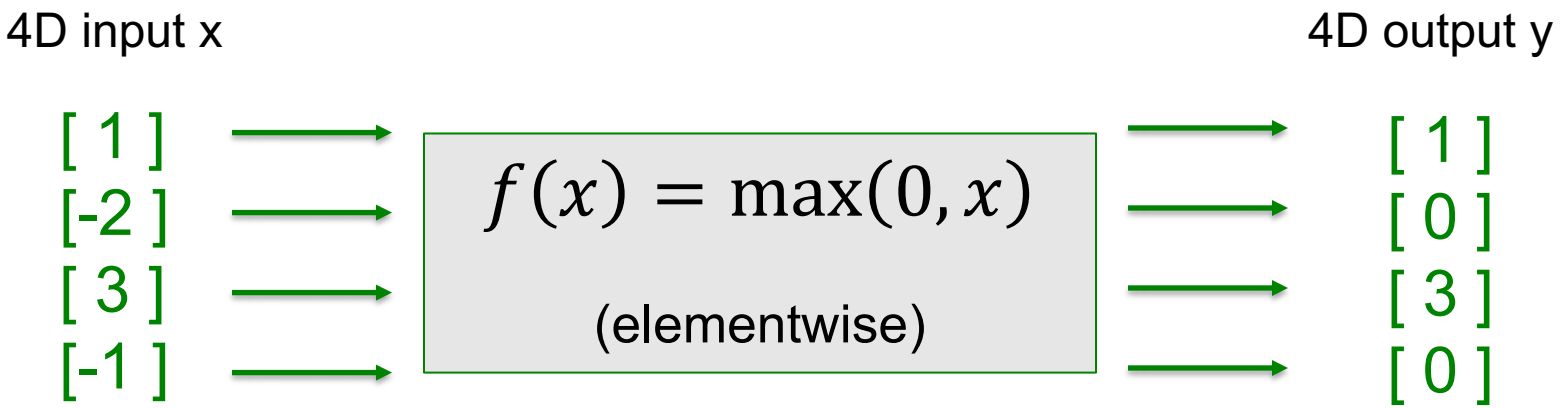
Backpropagation with Matrices (Tensors)



$$\left[\frac{\partial L}{\partial x}\right]_{i,j} = \sum_{k,l} \left[\frac{\partial z}{\partial x}\right]_{i,j,k,l} \left[\frac{\partial L}{\partial z}\right]_{k,l}$$

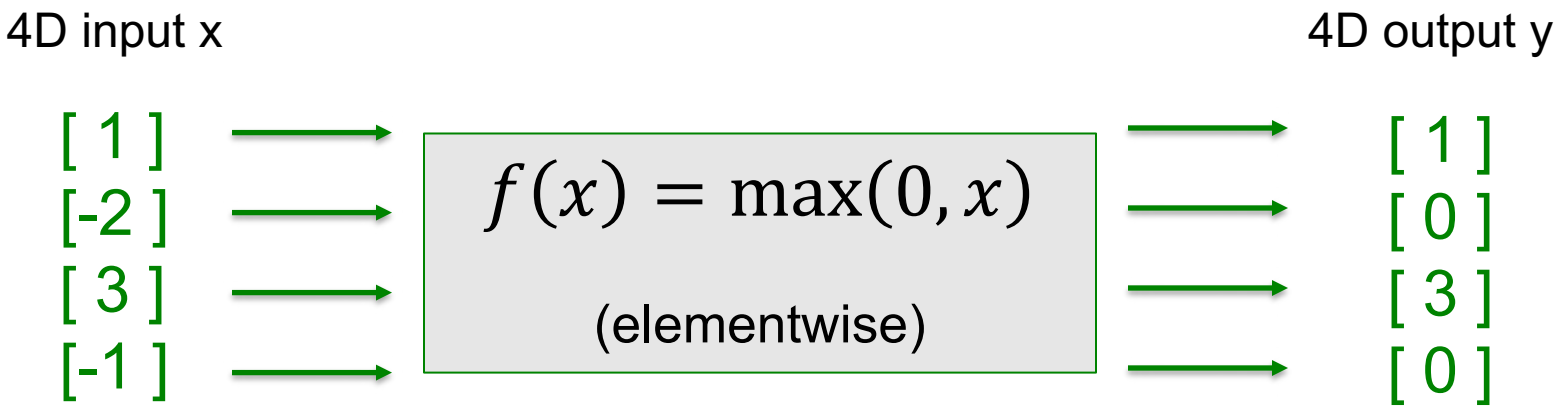
$$i, j \in [1..D_x, 1..M_x] \quad k, l \in [1..D_z, 1..M_z]$$

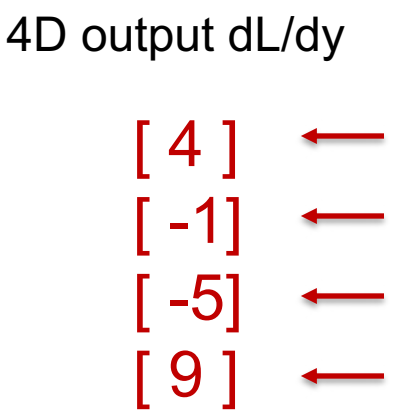
Backprop for Activation Functions



Source: Justin Johnson

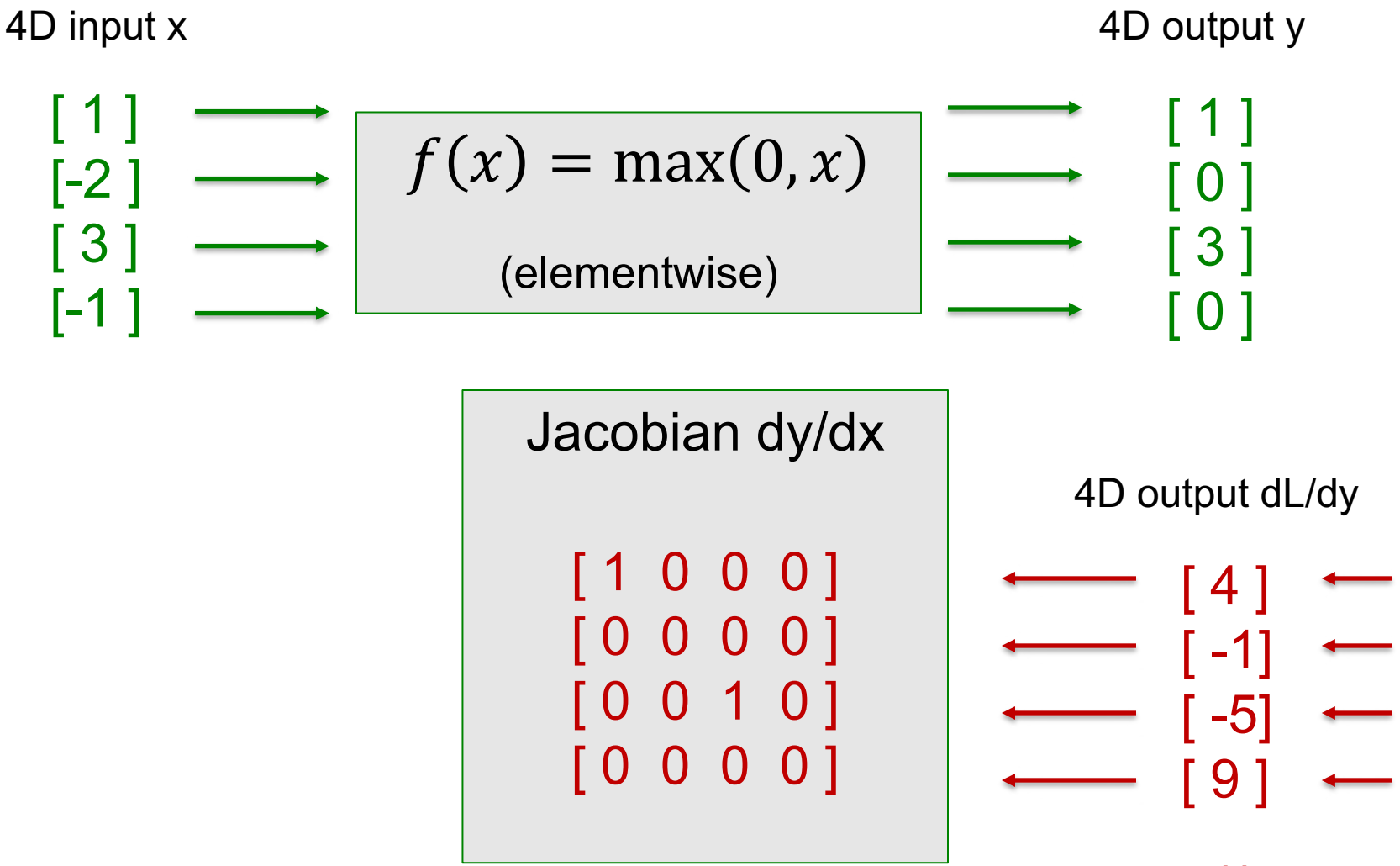
Backprop for Activation Functions





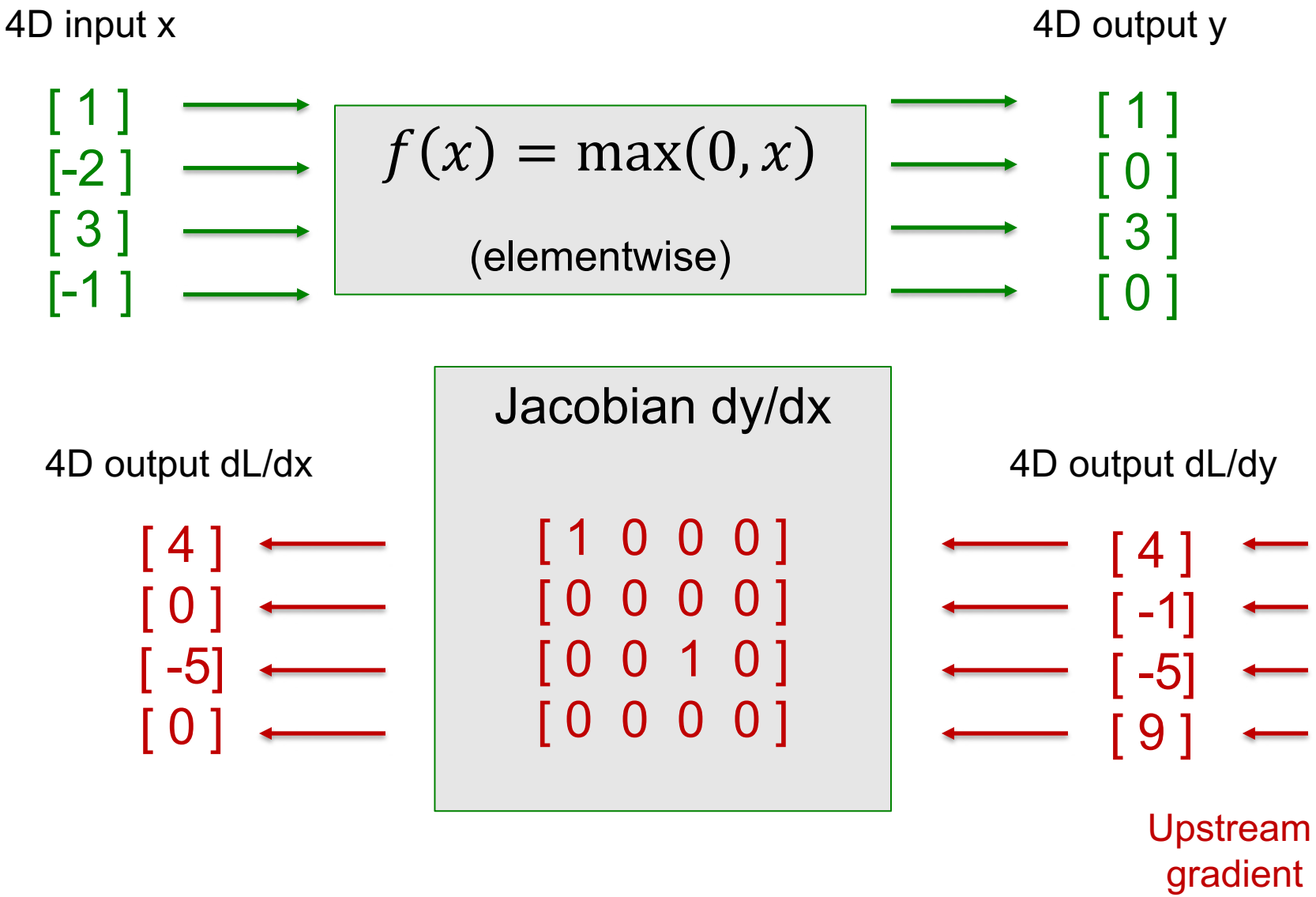
Source: Justin Johnson

Backprop for Activation Functions



Source: Justin Johnson

Backprop for Activation Functions



Source: Justin Johnson

Backprop for Activation Functions

- Jacobian is **sparse**. Off diagonal entries are all 0!
- Never **explicitly** form a Jacobian.
- Instead use **implicit** calculations.

Jacobian dy/dx

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

dy/dx dL/dy

$$\left(\frac{\partial L}{\partial x}\right)_i = \begin{cases} \left(\frac{\partial L}{\partial y}\right)_i & if\ x_i > 0 \\ 0 & otherwise \end{cases}$$

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \qquad \leftarrow \begin{bmatrix} 4 \\ -1 \\ -5 \\ 9 \end{bmatrix}$$

From: Justin Johnson

Activation Functions: The General Case

- $\boldsymbol{h} = g(\boldsymbol{z})$ is an element-wise function on each index of ***z:***

$$\boldsymbol{h} = \begin{pmatrix} g(z_1) \\ g(z_2) \\ \vdots \\ g(z_n) \end{pmatrix}$$

- Formally, $\frac{\partial h_i}{\partial z_j} = g'(z_i) \;\; if \;\; i = j$ and 0 otherwise

- Thus, $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \begin{bmatrix} g'(z_1) & \cdots & 0 \\ \vdots & & \\ 0 & \cdots & g'(z_n) \end{bmatrix}$

- Diagonal matrix represents that h_i and z_j have no dependency if $i \neq j$.

Activation Functions: The General Case

Assume, $\boldsymbol{h} = g(\boldsymbol{z})$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} = \overline{\boldsymbol{h}}$ ($\mathcal{L}$ is scalar).

$$z \longrightarrow h \longrightarrow \mathcal{L}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} = \begin{bmatrix} \frac{\partial h_1}{\partial z_1} & \cdots & 0 \\ \vdots & & \\ 0 & \cdots & \frac{\partial h_n}{\partial z_n} \end{bmatrix} \overline{\boldsymbol{h}} = g'(\boldsymbol{z}) \circ \overline{\boldsymbol{h}}$$

$$(R^{n \times n} R^{n \times 1} = R^{n \times 1})$$

- Matrix-vector multiplication is translated into a pointwise multiplication.

Commonly Used Expressions

Here, scalar a, vector $\mathbf{a}$ and matrix $\mathbf{A}$ are not functions of x and $\mathbf{x}$.

(C6) $$\frac{d\,\mathbf{a}^\top \mathbf{x}}{d\mathbf{x}} = \frac{d\,\mathbf{x}^\top \mathbf{a}}{d\mathbf{x}} = \mathbf{a}^\top$$

(C7) $$\frac{d\mathbf{x}^\top \mathbf{x}}{d\mathbf{x}} = 2\,\mathbf{x}^\top$$

(C8) $$\frac{d(\mathbf{x}^\top \mathbf{a})^2}{d\mathbf{x}} = 2\,\mathbf{x}^\top \mathbf{a}\,\mathbf{a}^\top$$

(C9) $$\frac{d\mathbf{A}\mathbf{x}}{d\mathbf{x}} = \mathbf{A}$$

(C10) $$\frac{d\,\mathbf{x}^\top \mathbf{A}}{d\mathbf{x}} = \mathbf{A}^\top$$

(C11) $$\frac{d\,\mathbf{x}^\top \mathbf{A}\mathbf{x}}{d\mathbf{x}} = \mathbf{x}^\top(\mathbf{A} + \mathbf{A}^\top)$$

Concrete Vectorial Example

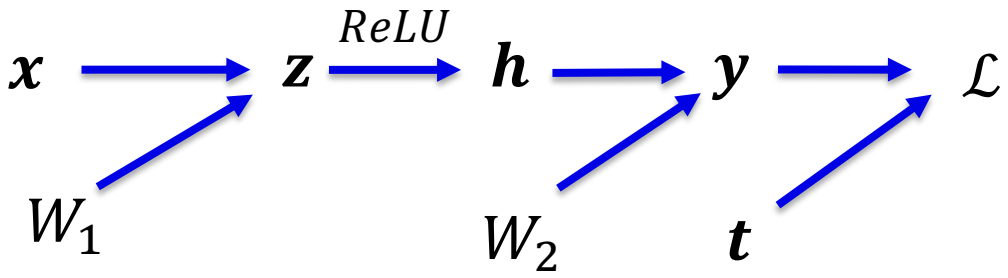
Forward Prop

$\boldsymbol{z} = W_1\boldsymbol{x}$ [nx1]=[nxm] [mx1]

$\boldsymbol{h} = ReLU(\boldsymbol{z})$ [nx1]

$\boldsymbol{y} = W_2\boldsymbol{h}$ [kx1]=[kxn] [nx1]

$\mathcal{L} = \|\boldsymbol{y} - \boldsymbol{t}\|^2$ [1x1]=[1xk] [kx1]



Concrete Vectorial Example

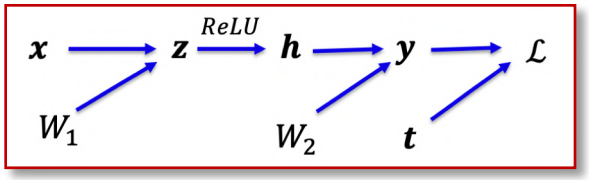
Forward Prop

$\boldsymbol{z} = W_1 \boldsymbol{x}$ nx1=(nxm) (mx1)

$\boldsymbol{h} = ReLU(\boldsymbol{z})$ nx1

$\boldsymbol{y} = W_2 \boldsymbol{h}$ kx1=(kxn) (nx1)

$\mathcal{L} = \|\boldsymbol{y} - \boldsymbol{t}\|^2$ 1x1=(1xk) (kx1)



downstream gradient = upstream gradient * local gradient

Back Propagation

$\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} = \bar{\boldsymbol{y}} = 2(\boldsymbol{y} - \boldsymbol{t})$ kx1

$\frac{\partial \mathcal{L}}{\partial W_2} = \bar{W}_2 = \bar{\boldsymbol{y}}\, \boldsymbol{h}^T$ kxn=[kx1]x[1xn]

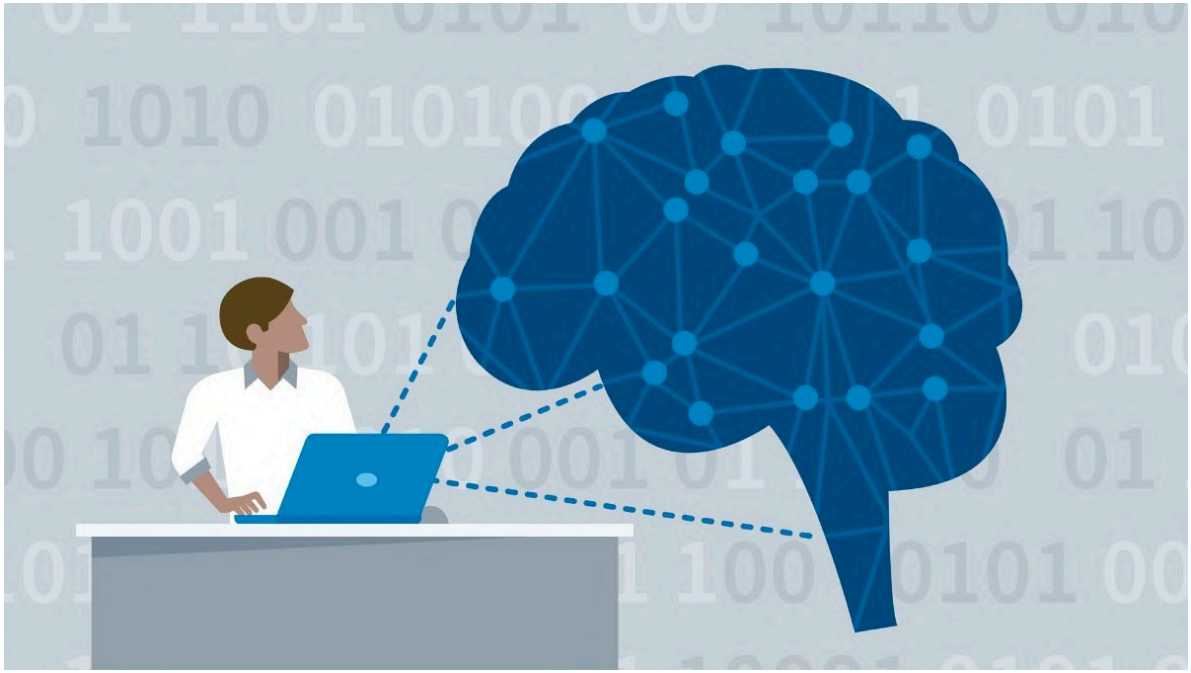
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} = \bar{\boldsymbol{h}} = W_2^T\, \bar{\boldsymbol{y}}$ nx1=[nxk][kx1]

$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \bar{\boldsymbol{z}} = \bar{\boldsymbol{h}} \circ I(\boldsymbol{z} > 0)$ nx1

$\frac{\partial \mathcal{L}}{\partial W_1} = \bar{W}_1 = \bar{\boldsymbol{z}}\, \mathbf{x}^{\mathrm{T}}$ nxm=[nx1]x[1xm]

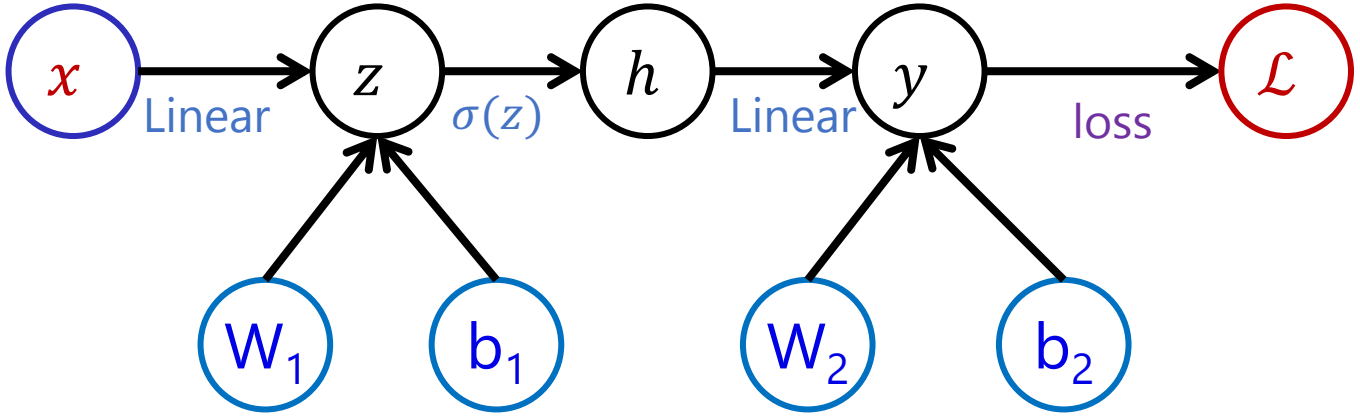
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \bar{\boldsymbol{x}} = W_1^T\, \bar{\boldsymbol{z}}$ mx1=[mxn]x[nx1]

Training and Optimization

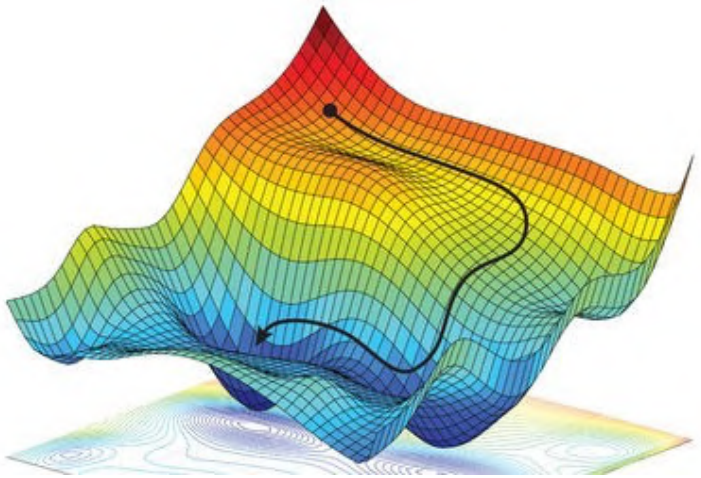


Many slides from: Stanford CS231

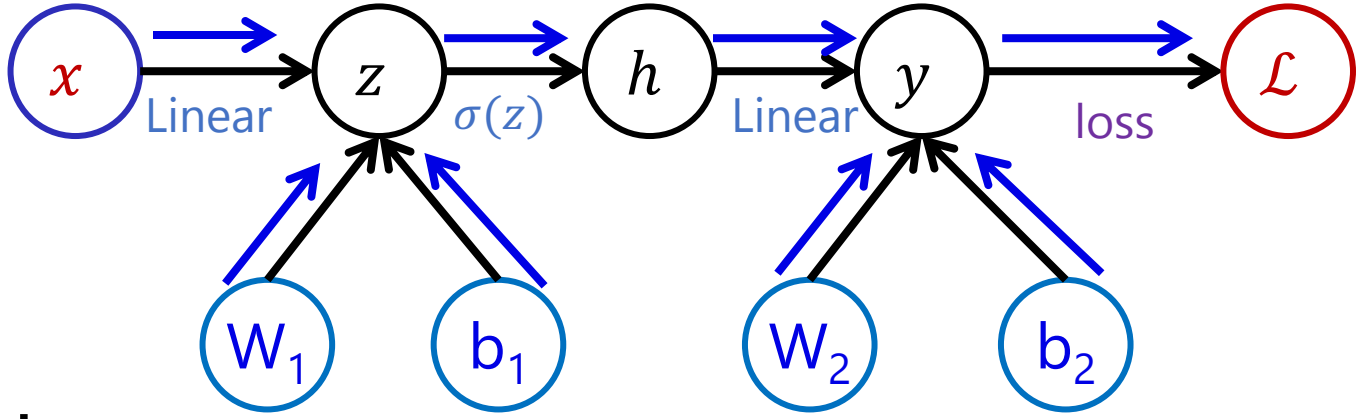
Where are we now?



- A network is represented using a computation graph.
- **Our goal**: learning the network parameters through optimization.



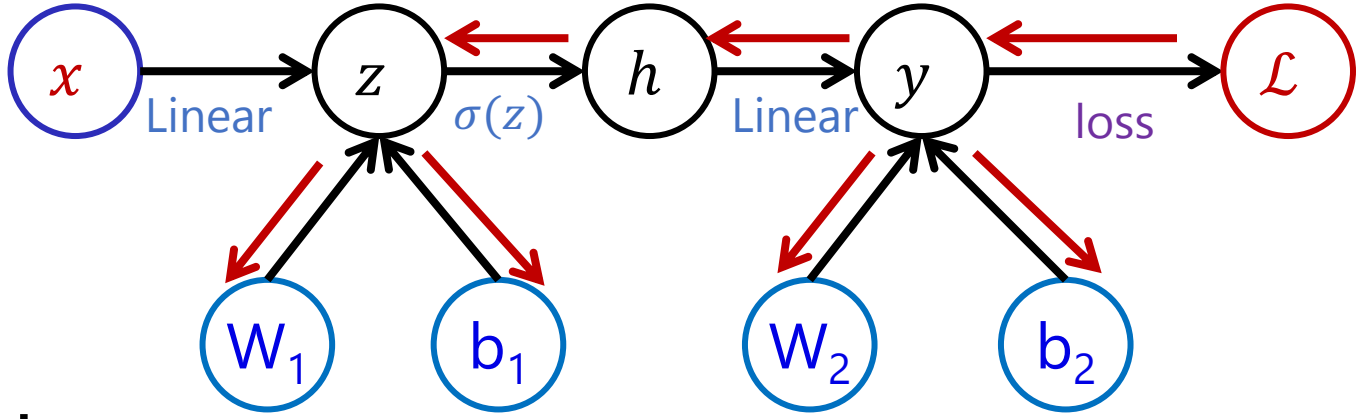
Mini-batch Stochastic GD



Loop:

1. **Sample** a batch of data
2. **Forward-prop** it through the graph (network). Get the cost.

Mini-batch Stochastic GD



Loop:

1. **Sample** a batch of data
2. **Forward-prop** it through the graph (network). Get the cost.
3. **Back-prop** to calculate the gradients
4. **Update** the parameters using the gradients.

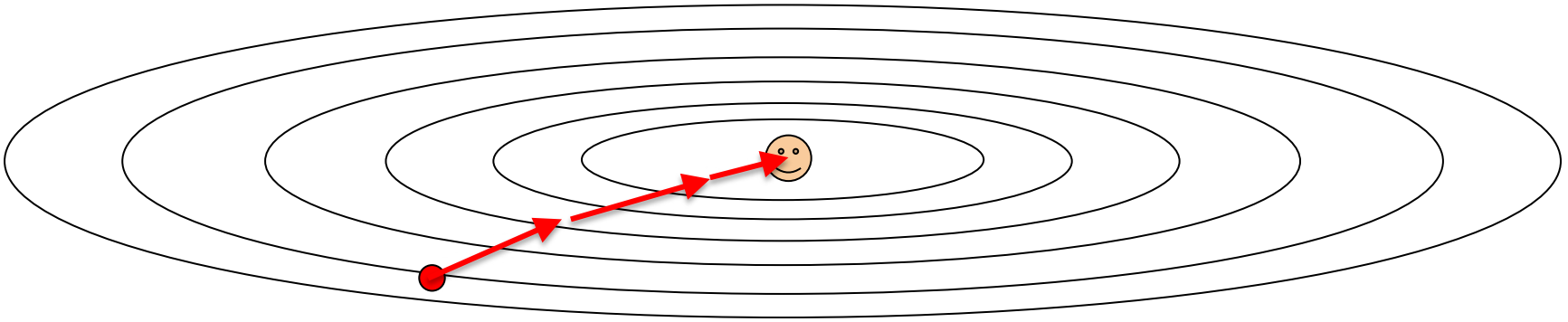
$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - \frac{\eta_t}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{L}_i(\boldsymbol{w})}{\partial \boldsymbol{w}}$$

Problems with SGD

Gradient descent suffers from several issues that hinder its convergence to the global minimum:

Narrow ravines:

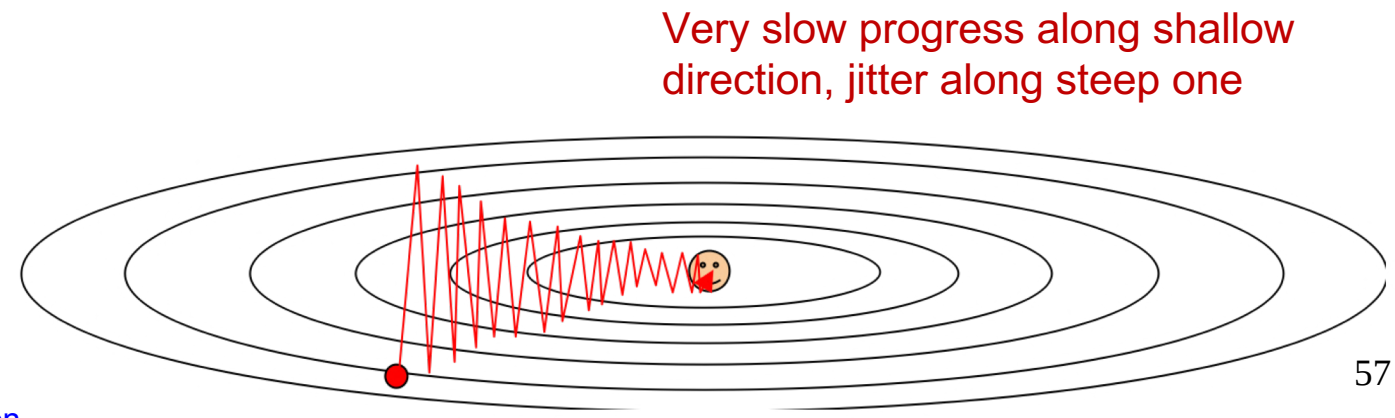
- The loss function is steep vertically but shallow horizontally.
- What is the trajectory we would like to converge towards the minimum?



Source: Stanford CS231

Problems with SGD

- The direction of steepest descent does not point at the optimal point unless the ellipse is a circle.
- The gradient is steep in the direction in which we only want to travel a short distance.
- The gradient is shallow in the direction in which we want to travel a large distance.
- This produces a zig-zag path.

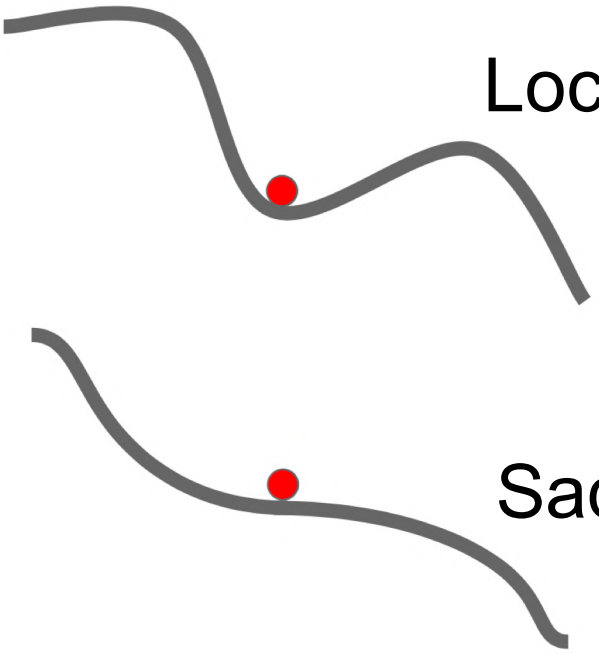


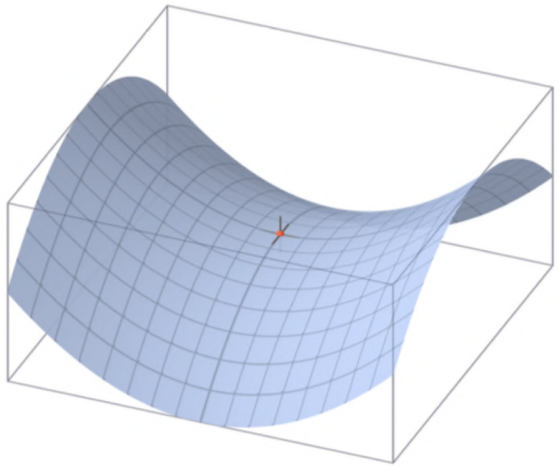
Source: Geoffrey Hinton

More Problems with SGD

Zero gradients:

- Zero gradient in local minima and saddle points.
- Saddle points are very common in high dimensions.
- Gradient descent gets stuck.





Source: Stanford CS231

Training and Optimization: Overview

- **One time setup:**

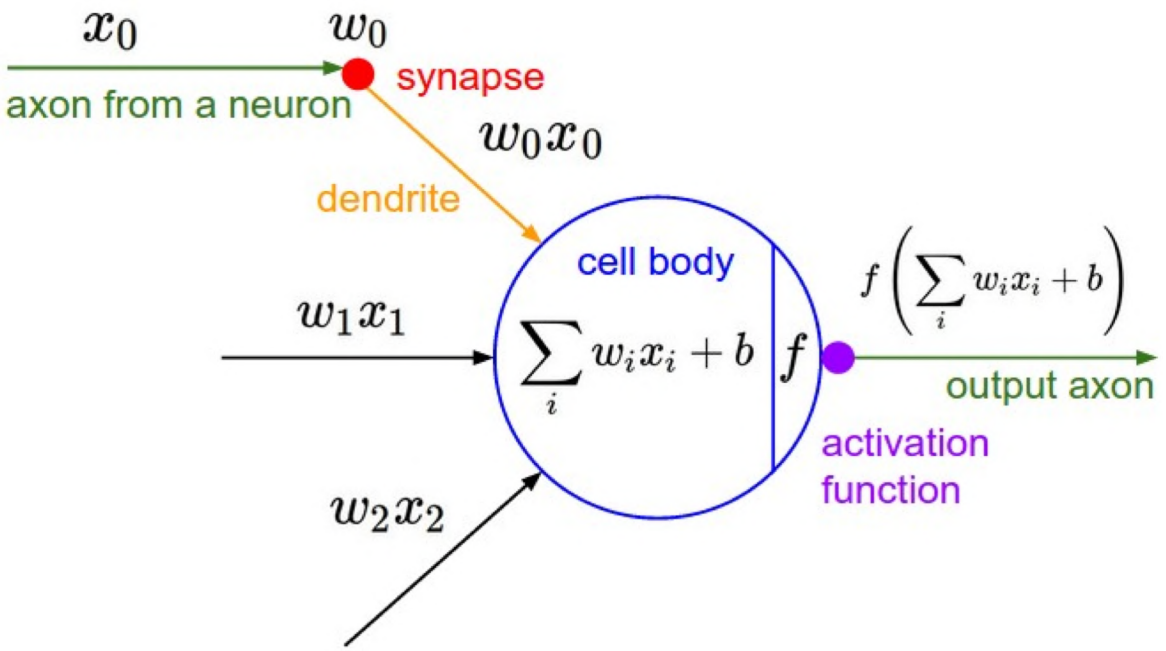
  Architecture, Activation functions, data preprocessing, weight initialization, regularization.

- **Training dynamics:**

  Learning rates schedules, large batch training, hyper-parameter optimization.

- **Evaluation**

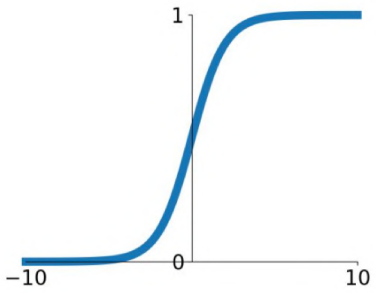
Activation Functions



Various types of activation functions

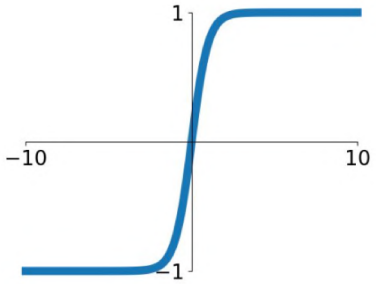
sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-z}}$$



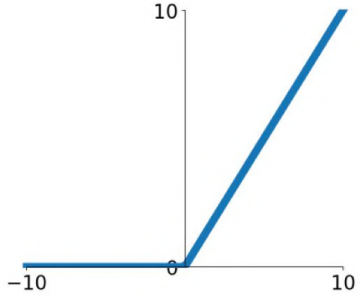
tanh:

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



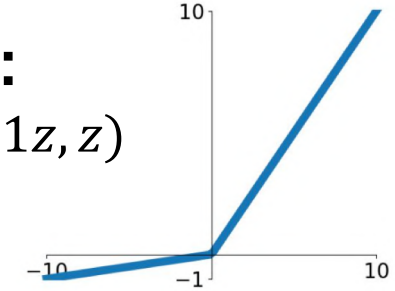
ReLU:

$$\text{ReLU}(z) = \max(0, z)$$



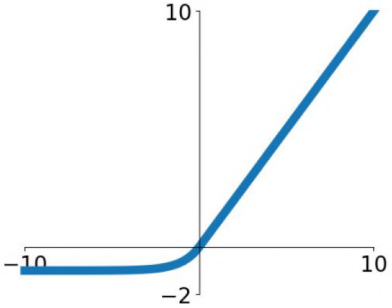
Leaky ReLu:

$$LR(z) = \max(0.1z, z)$$



ELU:

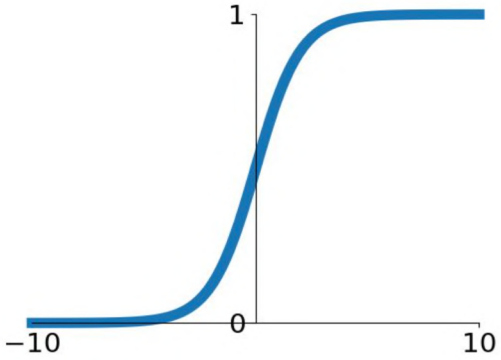
$$ELU(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^{z} - 1) & x < 0 \end{cases}$$



Source: Stanford CS231

Activations function: Sigmoid

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$



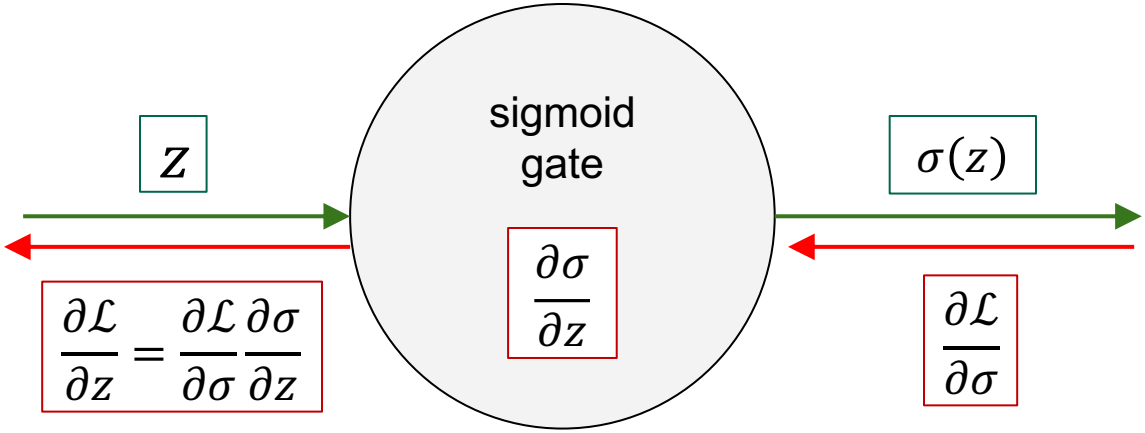
- Squashes numbers to range [0,1]
- Statistical interpretation
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

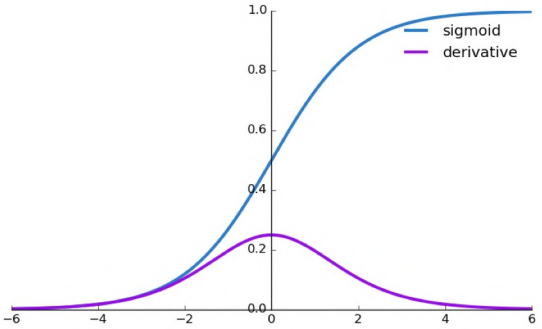
Problems:

1. Saturated neurons “kill” the gradients

Source: Stanford CS231

Activations function: Sigmoid



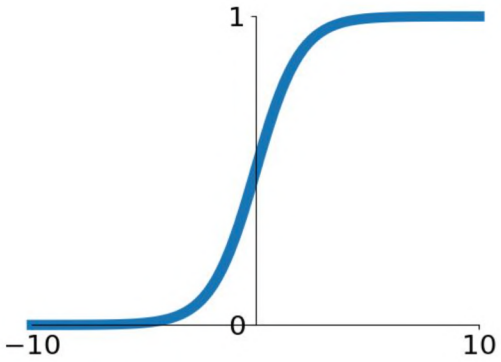


- What happens when $z = -10$?
- What happens when $z = 0$?
- What happens when $z = 10$?

Source: Stanford CS231

Activations function: Sigmoid

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$



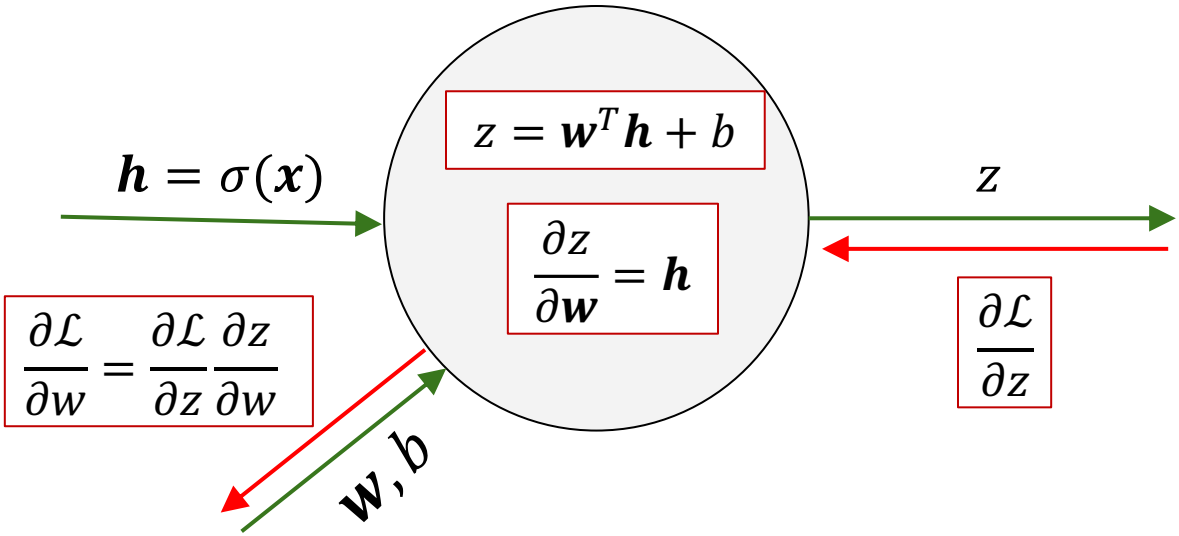
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

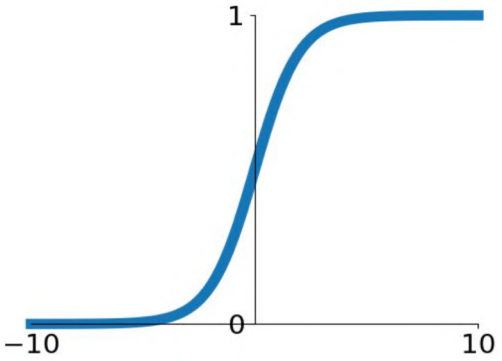
Problems:

1. Saturated neurons "kill" the gradients
2. exp(z) is compute expensive
3. Sigmoid outputs are not zero-centered

Source: Stanford CS231

Activations function: Sigmoid



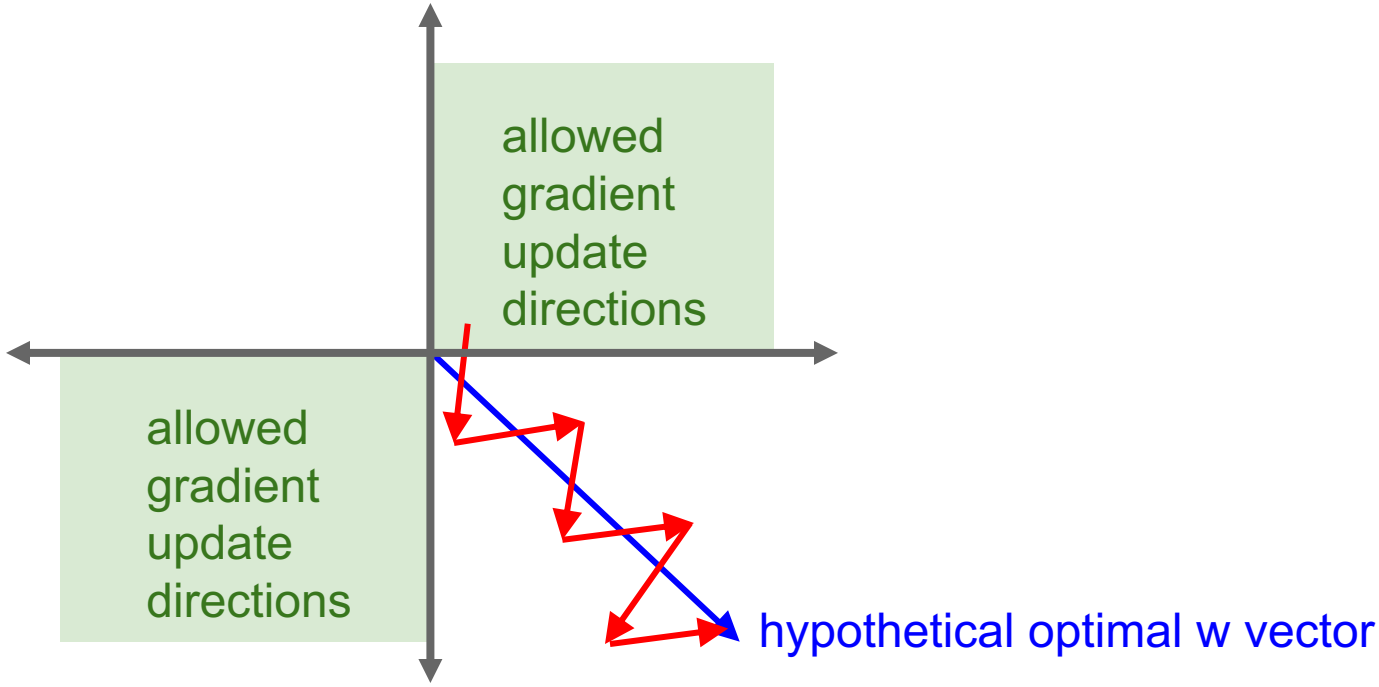


Sigmoid outputs are not zero-centered

- What happens to the gradients of w when the input to a neuron is always positive?
- Answer: $\frac{\partial \mathcal{L}}{\partial w}$ will be all positive or negative
- So, what is wrong with that?

Source: Stanford CS231

Activations function: Sigmoid

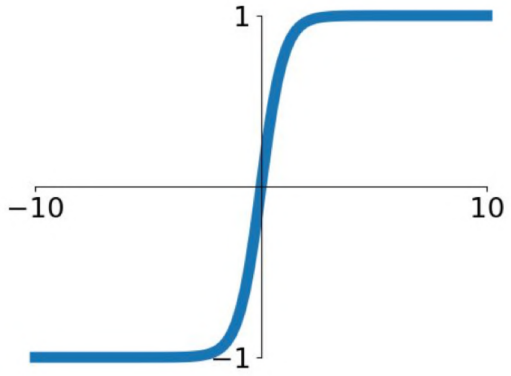


- Slow convergence to the optimal solution
- This is why we want **zero-mean data**!
- Only true for a single example. Mini-batch helps.

Source: Stanford CS231

Activations function: tanh

tanh: $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

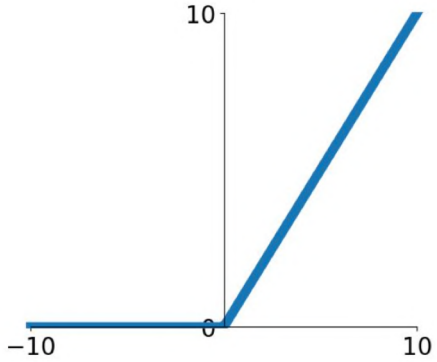


- Squashes numbers to range [-1,1]
- Zero-centered (nice)
- Still “kills” gradients when saturated (active in small range)
- Still exp(z) is compute expensive

Source: Stanford CS231

Activations function: ReLU

ReLU (Rectified Linear Unit):

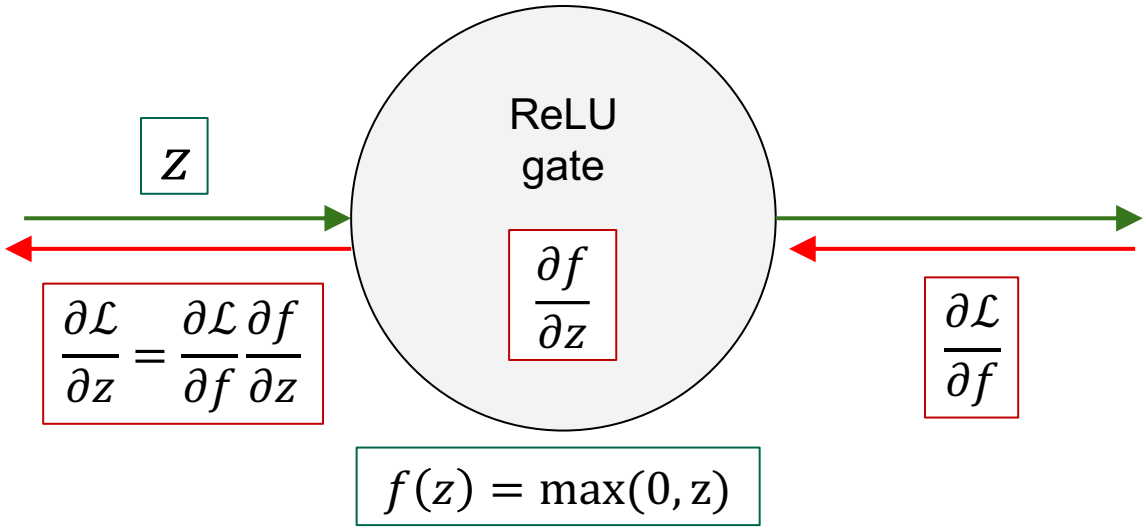


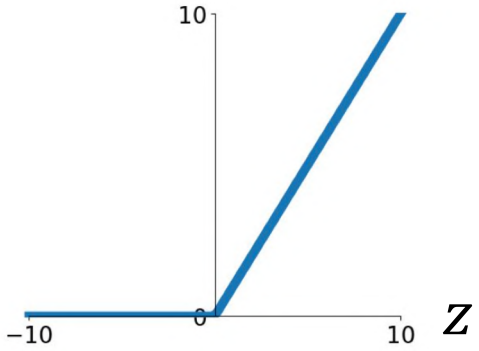
$$f(z) = \max(0, z)$$

- Does not saturate (in the $z > 0$ region)
- Computationally very efficient
- Converges much faster than sigmoid/tanh (6x)
- Not zero-centered
- “Dead” parameter regions

Source: Stanford CS231

Activations function: ReLU



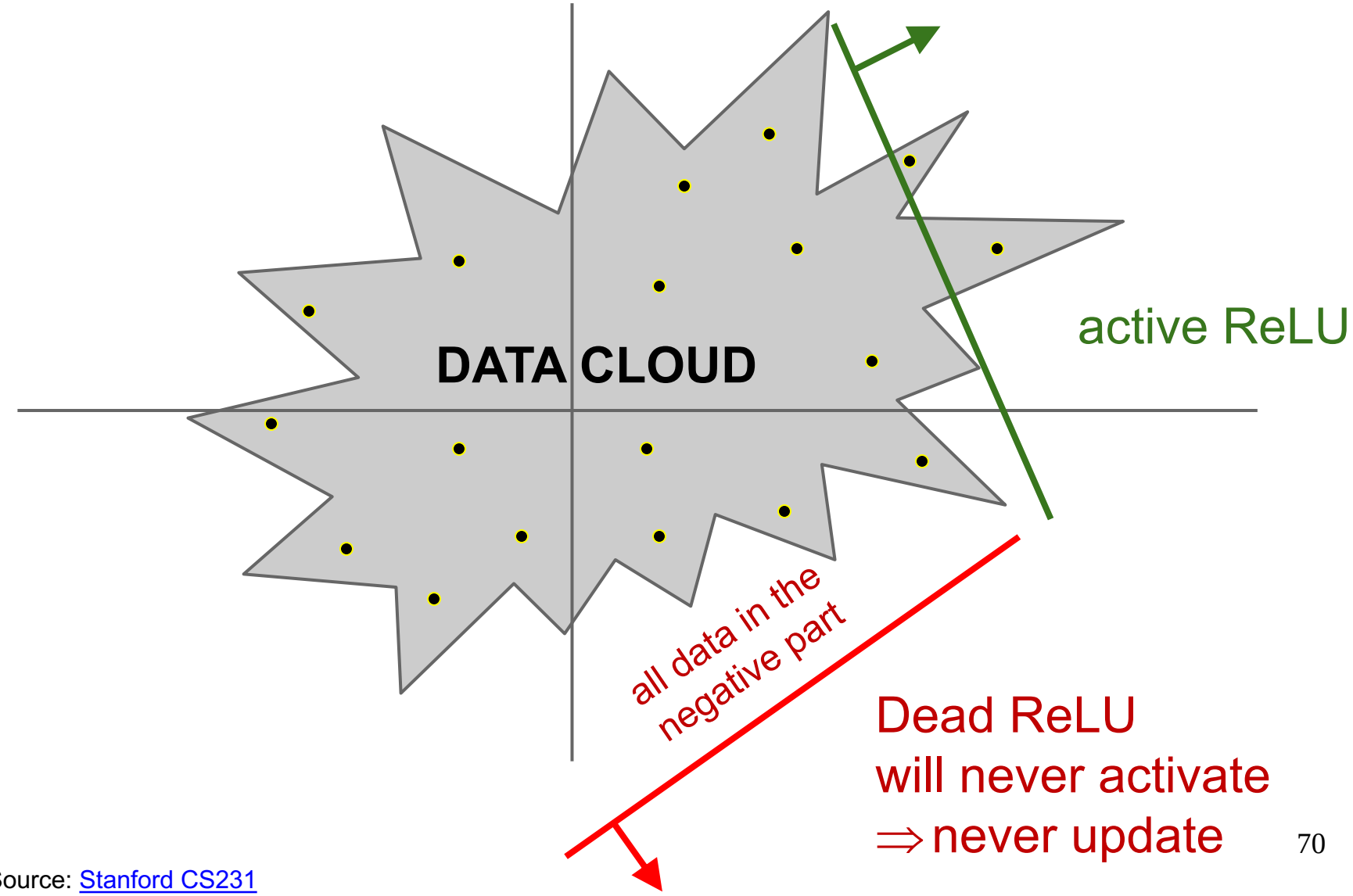


"Dead" parameter regions

- What happens when $z = 10$?
- What happens when $z = 0$?
- What happens when $z = -10$?

Source: Stanford CS231

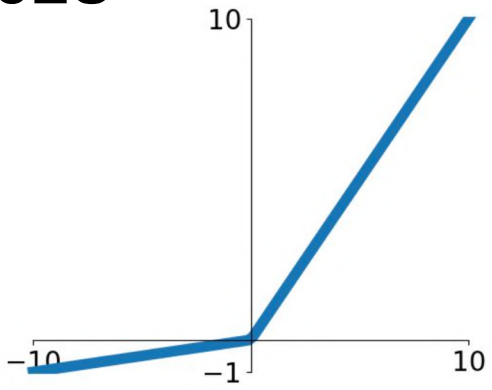
Activations function: ReLU



Source: Stanford CS231

ReLU alternatives

Leaky ReLU



$$f(z) = \max(0.01z, z)$$

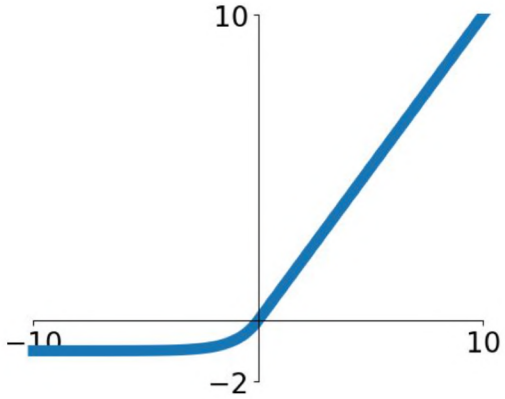
- Does not saturate
- Computationally efficient
- Converges fast
- **Will not "die"**
- Related alternative: **Parametric ReLU**

$f(z) = \max(\alpha z, z)$ and backprop into α

Source: Stanford CS231

ReLU alternatives

Exponential Linear Units (ELU)

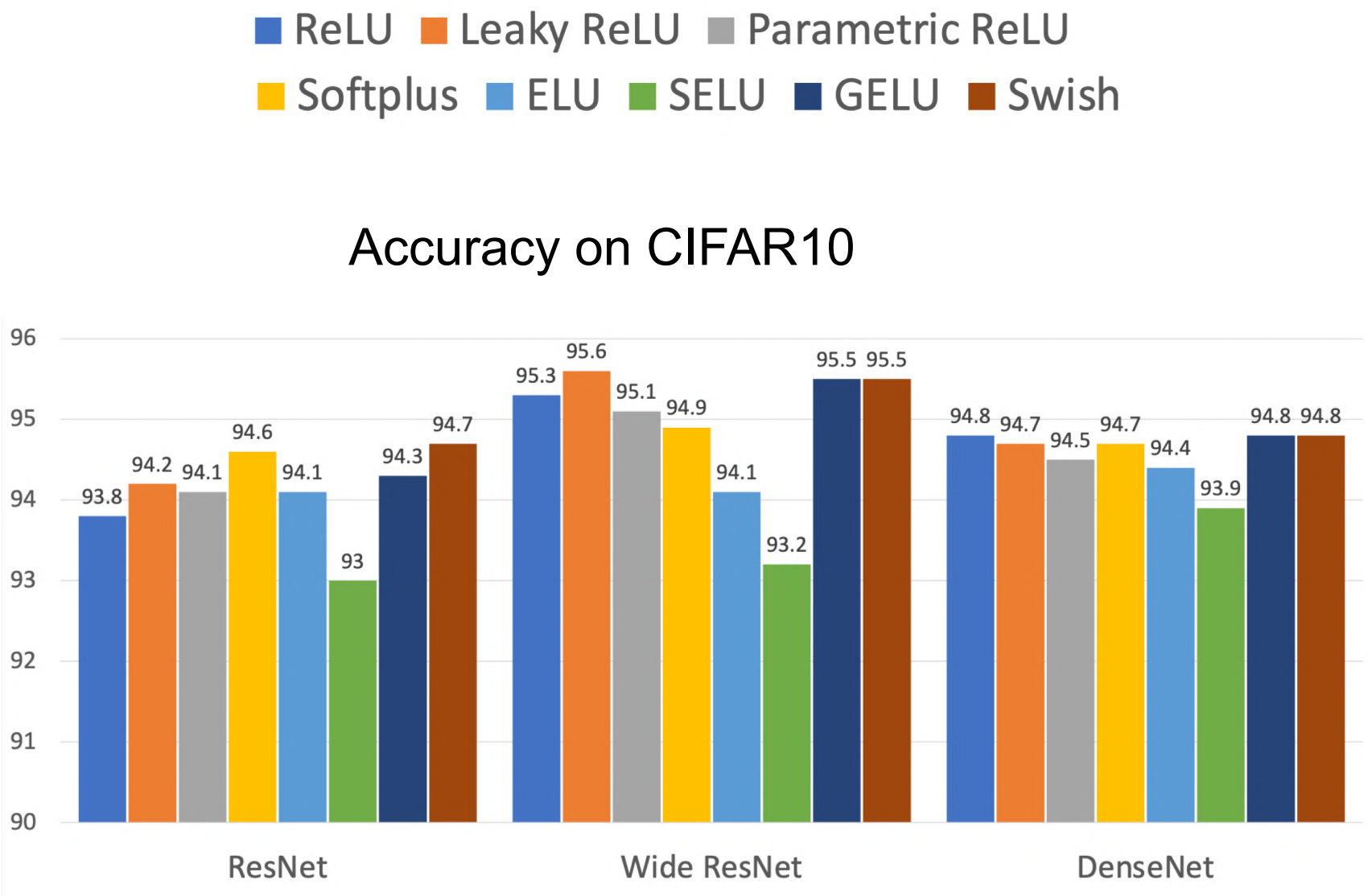


$$f(z) = \begin{cases} z & if \ z > 0 \\ \alpha(\exp(z) - 1) & z \leq 0 \end{cases}$$

- All benefits of ReLU
- Does not die
- Close to zero mean outputs
- Computation requires exp()

Source: Stanford CS231

ReLU alternatives



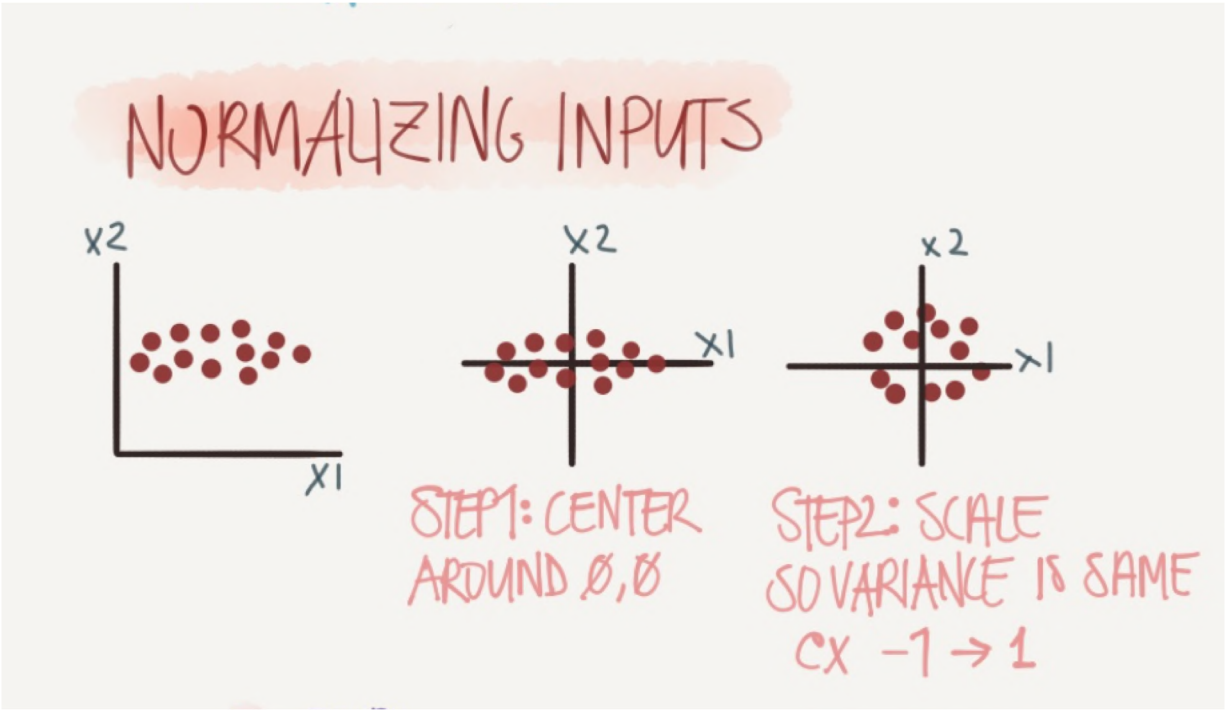
Source: Justin Johnson

TDLR: Activation functions

- Use **ReLU**. Be careful with your learning rates.
- Try out **Leaky ReLU** / **ELU.**
- Try out **tanh** but don't expect much.
- Don't use **sigmoid.**

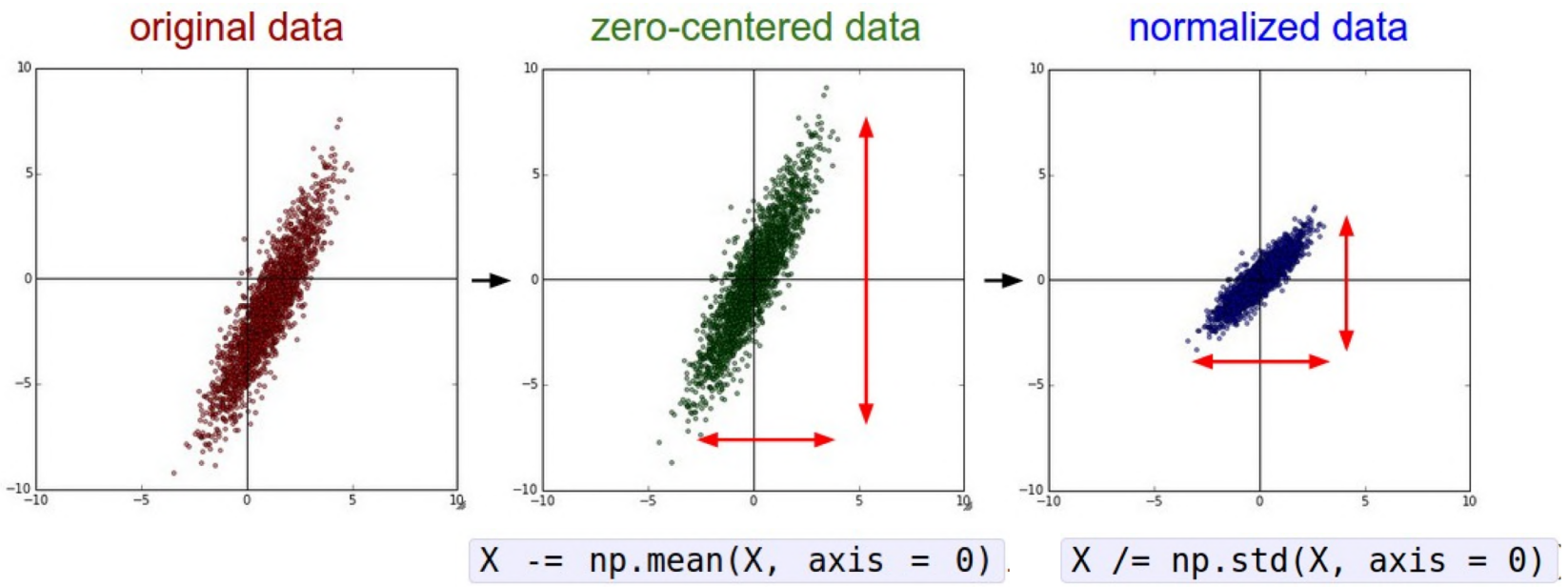
Source: Stanford CS231

Data Preprocessing



Data preprocessing

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$

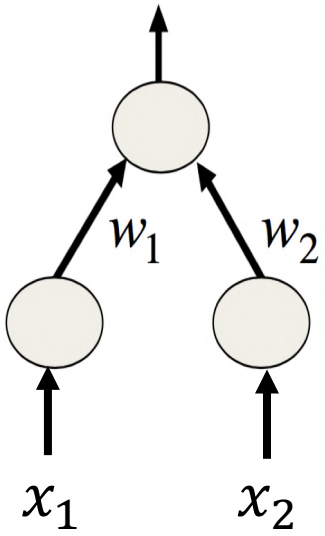


Assume $X \in [N \times D]$ is a data matrix, each example in a row.

Source: Stanford CS231

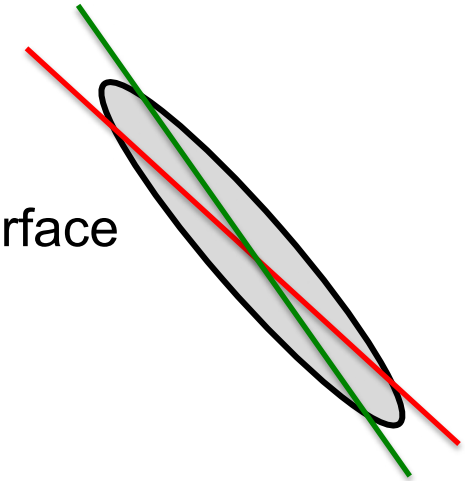
Zero centering

- When using steepest descent, shifting the input values makes a big difference.
- It usually helps to transform each component of the input vector so that it has zero mean over the whole training set.



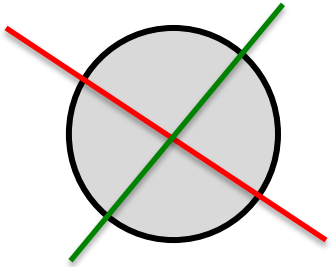
x_1	x_2	t
101	101	2
101	99	2

gives error surface



x_1	x_2	t
1	1	2
1	−1	2

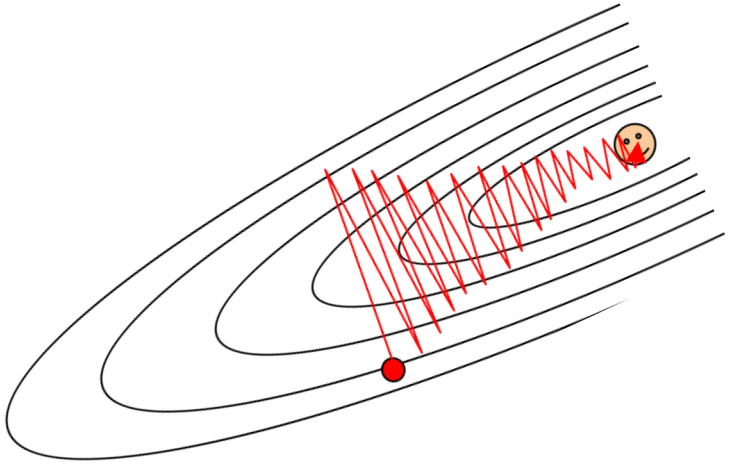
gives error surface



Source: Geoffrey Hinton

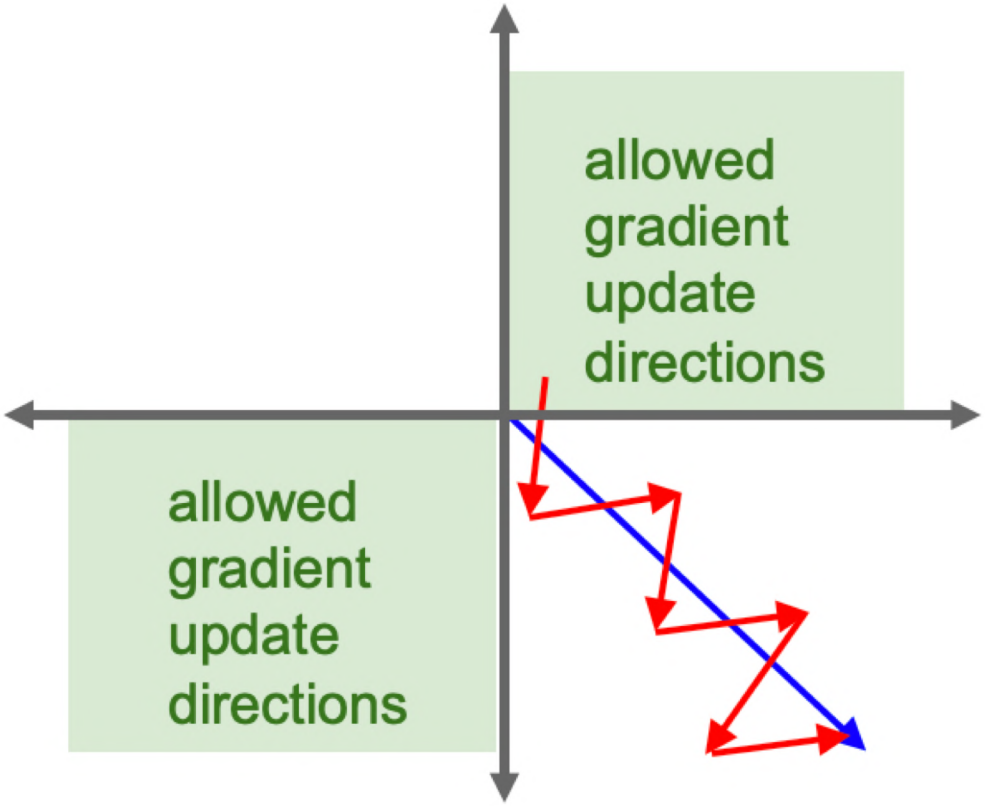
Zero centering

- Remember what happens when we have to optimize over a narrow ravine.



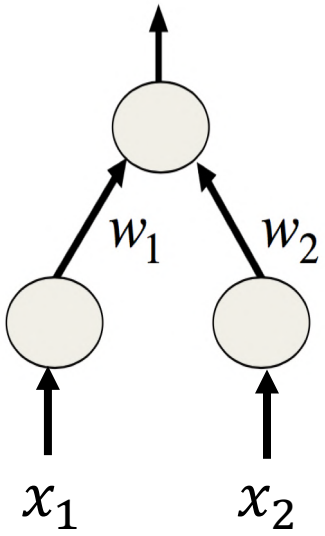
Zero centering

- Also remember what happens when the input to a neuron is always positive/negative.



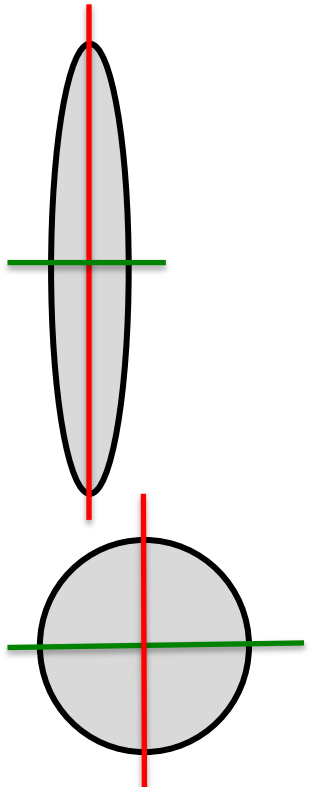
Scaling

- When using steepest descent, scaling the input values makes a big difference.
- It usually helps to transform each component of the input vector so that it has unit variance over the whole training set.



x_1	x_2	t
0.1	10	2
0.1	−10	2

gives error surface



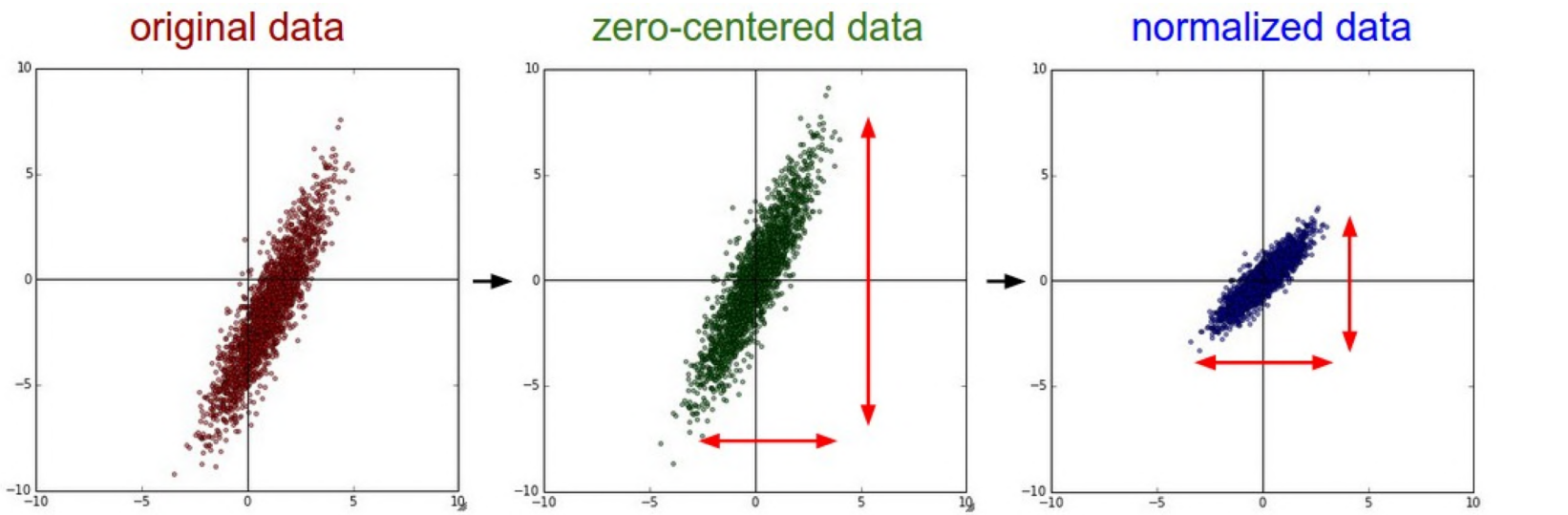
x_1	x_2	t
1	1	2
1	−1	2

gives error surface

Source: Geoffrey Hinton

Scaling

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$



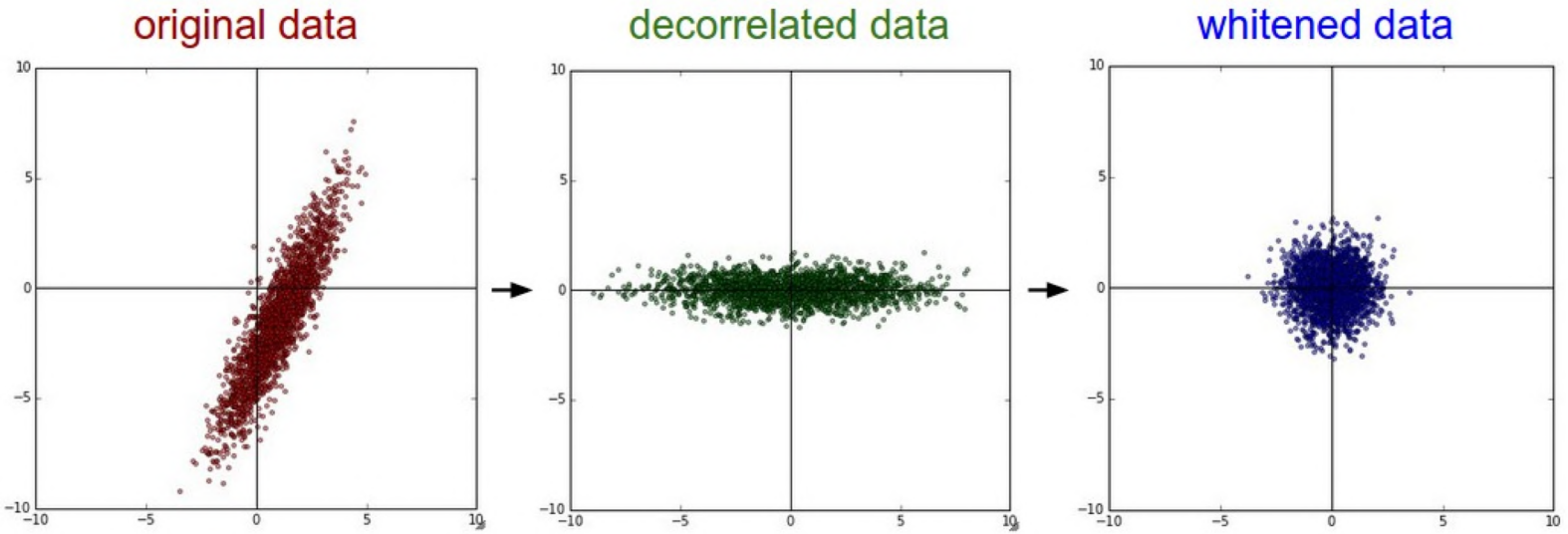
Source: Stanford CS231

Data preprocessing

- In practice, you may also see **PCA** and **Whitening** of the data.
- This can be performed using SVD:

$$\mathrm{X} \leftarrow X - mean(X, axis = 0),$$

$$[U, S, V] = svd(X^T X), \qquad X_{white} = XUS^{-\frac{1}{2}}$$



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

Source: Stanford CS231

Data preprocessing

Data preprocessing for images:

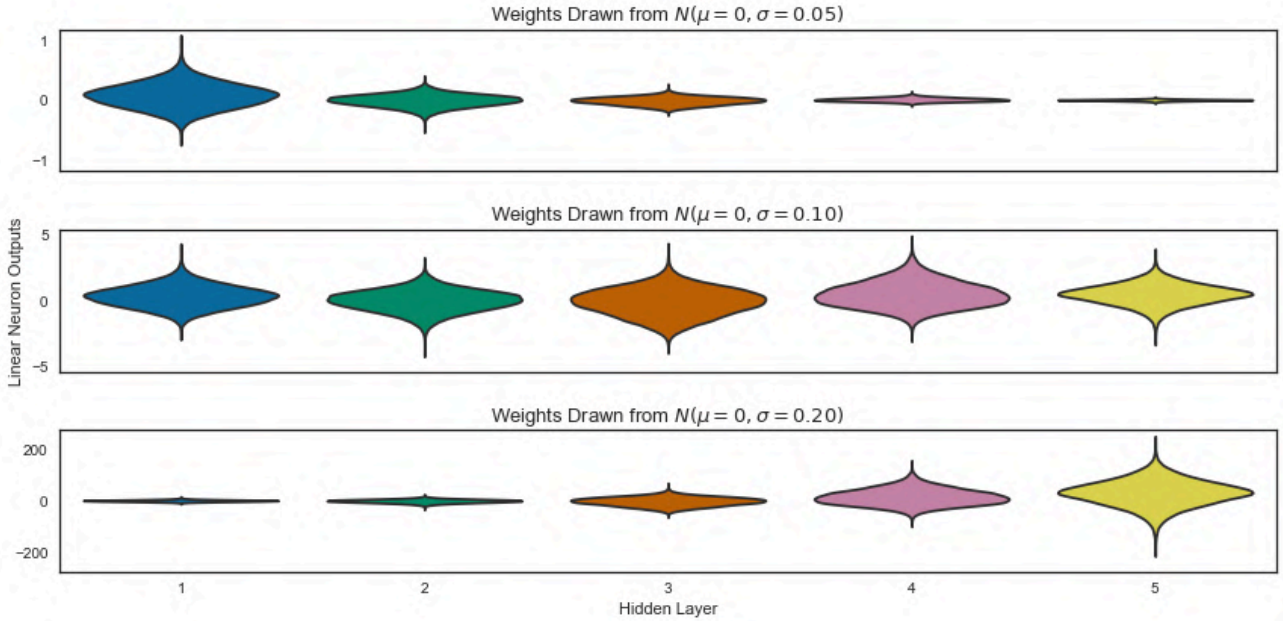
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the *mean image* (all input images need to have the same resolution). (mean image = [32,32,3])
- Subtract *per-channel* mean (mean = 3 numbers)
- Subtract *per-channel* mean and divide by *per-channel* std (std = 3 numbers)
- Be sure to apply the same transformation at training and test time! Save training set statistics and apply to test data.
- Not common to normalize variance, or to apply PCA or whitening (why?).

Image preprocessing

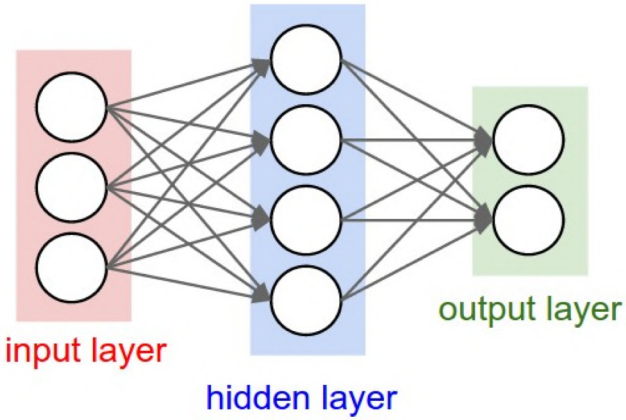
Images mean (left) and std (right) of a set of data inputs

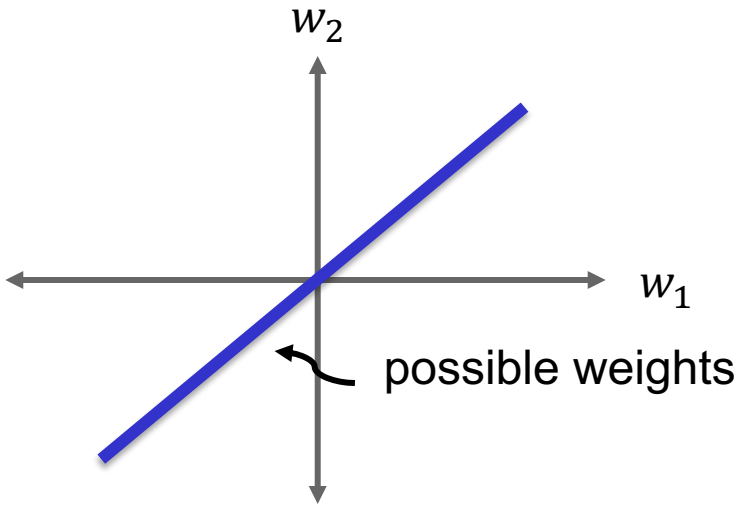
Weight Initialization



Weight initialization:

- What happens if we initializing all weights to be the same number (e.g. zero)?
- If two hidden units have exactly the same incoming and outgoing weights, they will always get exactly the same values.
- The gradients is also the same for every unit, thus, all the weights have the same values in the subsequent iterations.

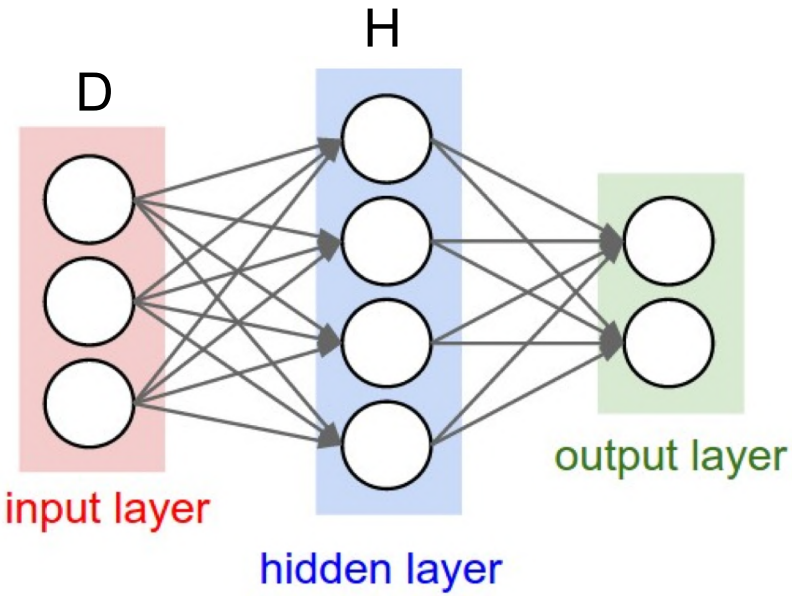




Weight initialization: Experiments

- Initiate the weights with **small random numbers** (Gaussian with zero mean and σ standard deviation): $w \sim \mathcal{N}(0, \sigma)$

$$\mathrm{W} = \sigma * \mathrm{np.random.randn(H, D)}$$

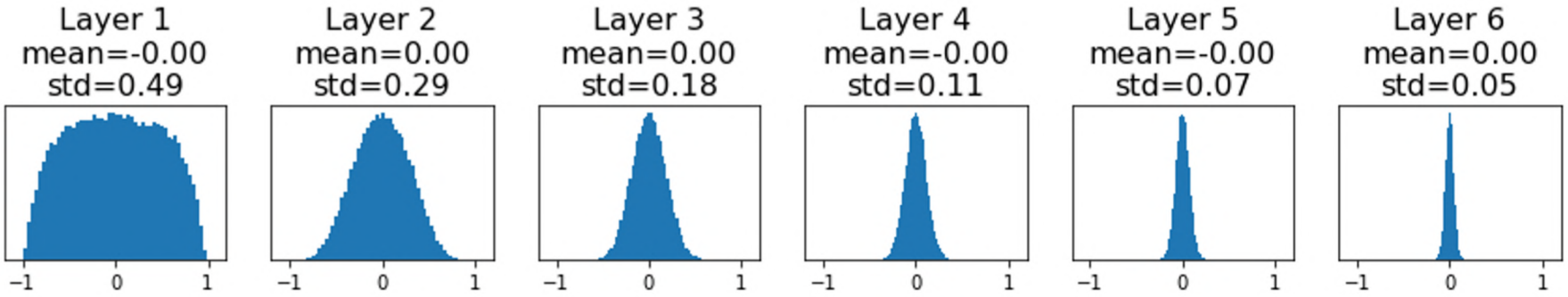


Source: Stanford CS231

Weight initialization: Experiment 1

- Forward pass for 6-layer network
- 4096 neurons on each hidden layer
- tanh non-linearities
- Initialized by randn with std=0.01

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

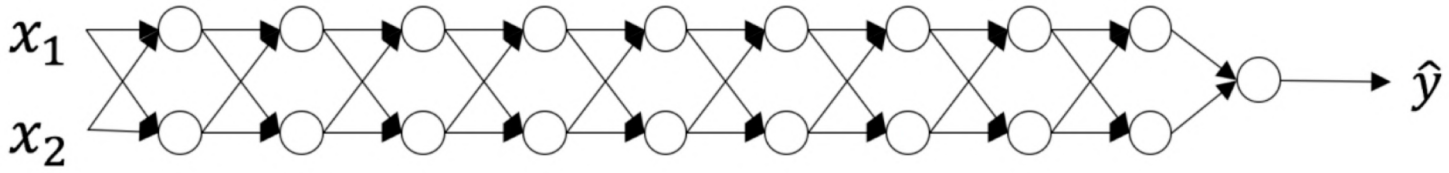


When the initialization is too small, all activations become zero!
What will be the gradients dL/dW?

Source: Justin Johnson

Weight initialization: **vanishing gradients**

- Consider a deep network with linear activation functions:



$$\hat{y} = a^{[L]} = W^{[L]}W^{[L-1]}W^{[L-2]}\dots W^{[3]}W^{[2]}W^{[1]}x$$

- Assume weights are initialized slightly smaller than the identity matrix:

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix}$$ what are the e.v.?

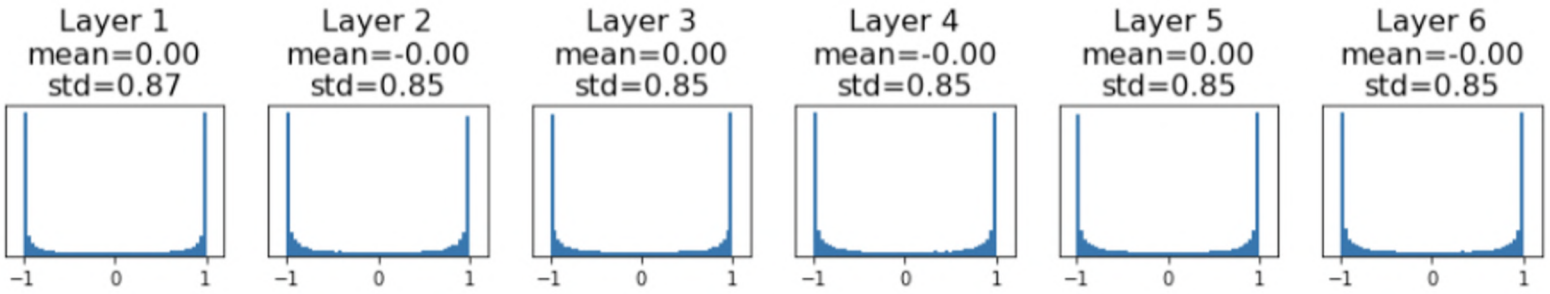
- The activation $a^{[\ell]}$ decrease exponentially with ℓ.
- When these activations are used in backward propagation, this leads to the vanishing gradient problem (Note, that if $a^{[i]} = W^{[i]}a^{[i-1]}$, then the local gradients $\frac{\partial a^{[i]}}{\partial W^{[i]}} = a^{[i-1]}$).

Source: deeplearning.ai

Weight initialization: Experiment 2

Second try: Initiate the weights with zero mean Gaussian values with 0.05 standard deviation

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```



- Almost all neurons are completely saturated, either -1 and 1.
- No learning - local gradients are all zero!

Source: Justin Johnson

Weight initialization: **exploding gradients**

- Consider the network with linear activation functions:

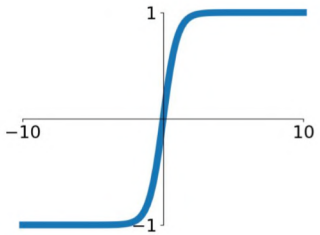
$$\hat{y} = a^{[L]} = W^{[L]} W^{[L-1]} W^{[L-2]} \dots W^{[3]} W^{[2]} W^{[1]} x$$

- Assume every weight is initialized slightly larger than the identity matrix:

$$W^{[1]} = W^{[2]} = \dots = W^{[L-1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

what are the e.v.?

- The activation $a^{[\ell]}$ increase exponentially with ℓ.
- When these activations are used in backward propagation, this leads to the exploding gradient problem.
- What will happen if ***tanh*** *is used* as activation functions?

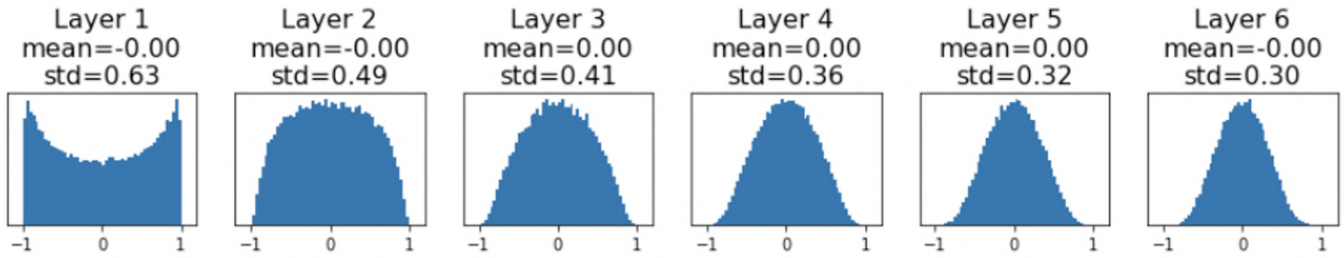


Source: deeplearning.ai

Weight initialization: Experiment 3

Third try: Initiate the weights with zero mean Gaussian values with $\frac{1}{\sqrt{D_{in}}}$ standard deviation

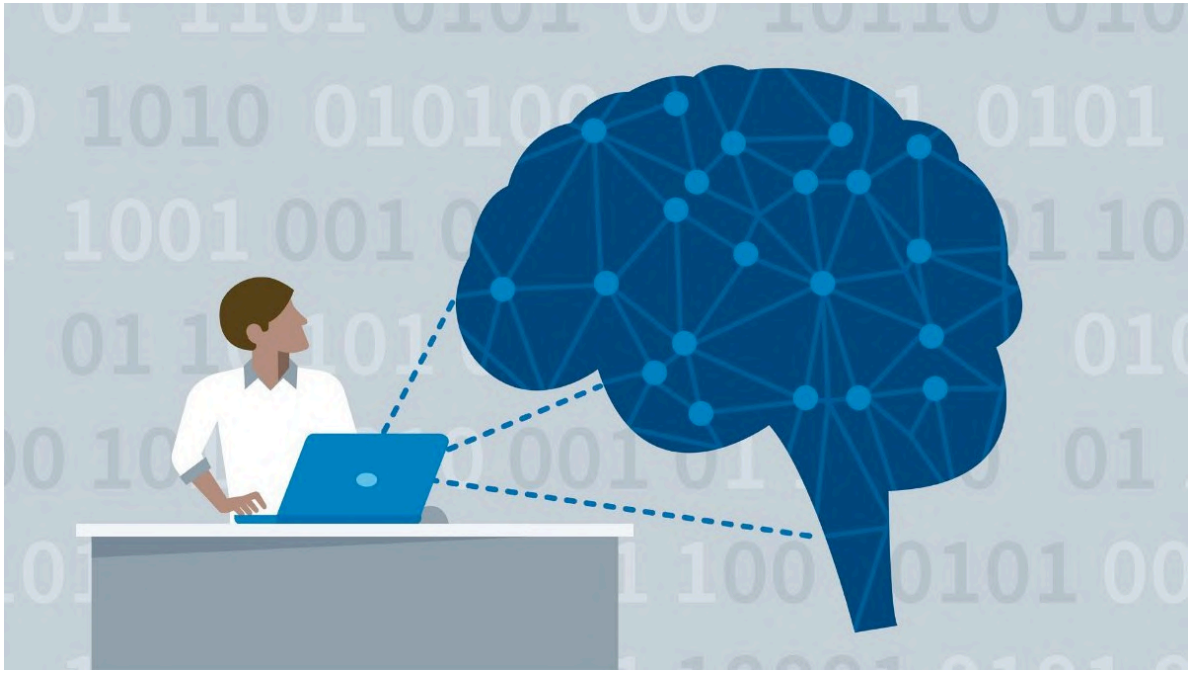
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```



- "Just right": Activations are nicely scaled for all layers!

Source: Justin Johnson

Training and Optimization
Part 2



Many slides from: Stanford CS231

Last week:

Training and Optimization:

- **One time setup:**

  activation functions

  data preprocessing

  weight initialization

  regularization.

- **Training dynamics:**

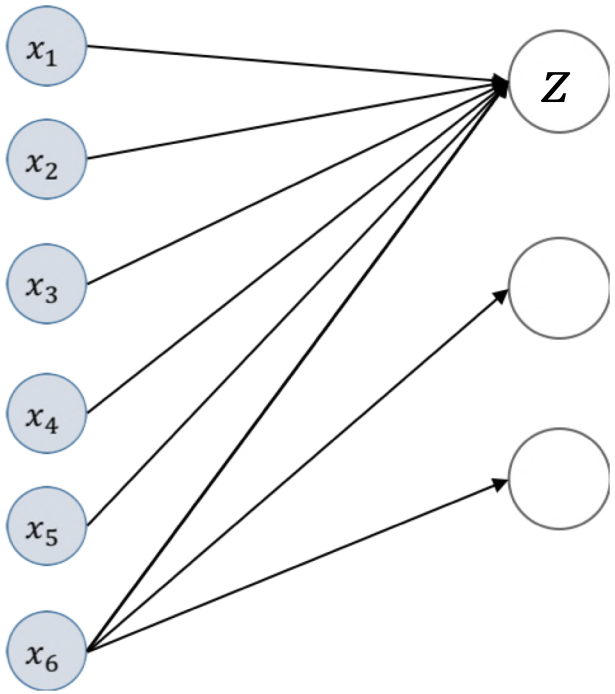
  Learning rates schedules, large batch training, hyper-parameter optimization.

- **Evaluation**

Xavier initialization

- Use randn with $\text{std} = \frac{1}{\sqrt{D_{in}}}$
- **Goal**: For each neuron, variance of input = variance of output

$$z = \boldsymbol{w}^T \boldsymbol{x} \; ; \quad z = \sum_{j=1}^{D_{in}} w_j \, x_j$$



Xavier initialization

- Use randn with $\text{std} = \frac{1}{\sqrt{D_{in}}}$
- **Goal**: For each neuron, variance of input = variance of output

$$z = \boldsymbol{w}^T \boldsymbol{x} \;; \quad z = \sum_{j=1}^{D_{in}} w_j \, x_j$$

- Assuming w_i and x_i are i.i.d.

$$Var(z) = D_{in} * Var(w_i x_i)$$

- Assuming $\boldsymbol{w}$ and $\boldsymbol{x}$ are independent and zero mean:

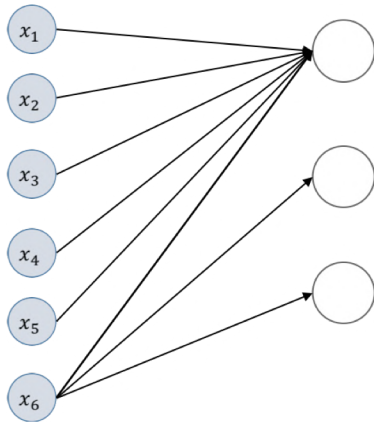
$$Var(z) = D_{in} * Var(w_i)\, Var(x_i)$$

- Setting $\text{Var}(\text{w}_\text{i}) = 1/\text{D}_\text{in}$ then

$$Var(z) = Var(x_i)$$

Weight initialization

- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
- We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to $1/\sqrt{D_{in}}$



Common heuristics:

- $\sigma = 1/\sqrt{D_{\text{in}}}$, where D_{in} is the number of inputs to a layer (Xavier initialization)
- $\sigma = 2/\sqrt{D_{\text{in}} + D_{\text{out}}}$ (Glorot and Bengio, 2010)

- Initializing biases: just set them to 0

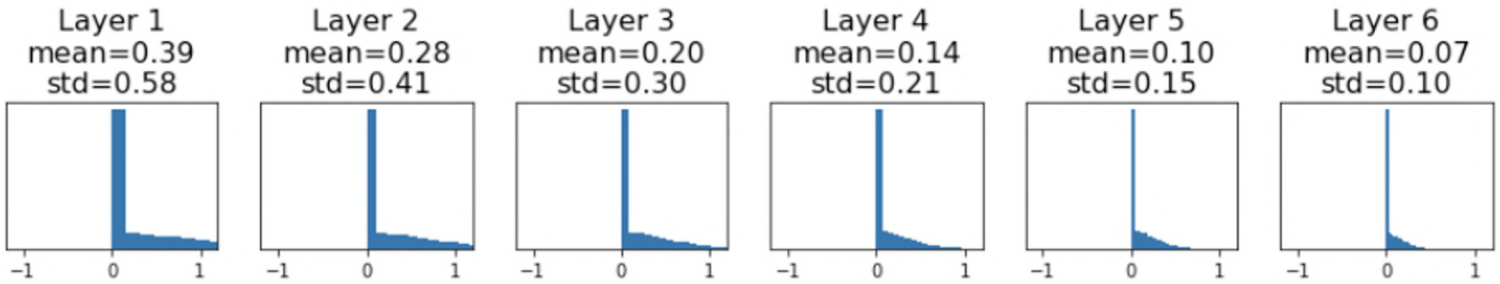
Source: Illinois CS 498

ReLU initialization

- When using the ReLU nonlinearity with $\sigma = 1/\sqrt{D_{\text{in}}}$ it breaks.

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

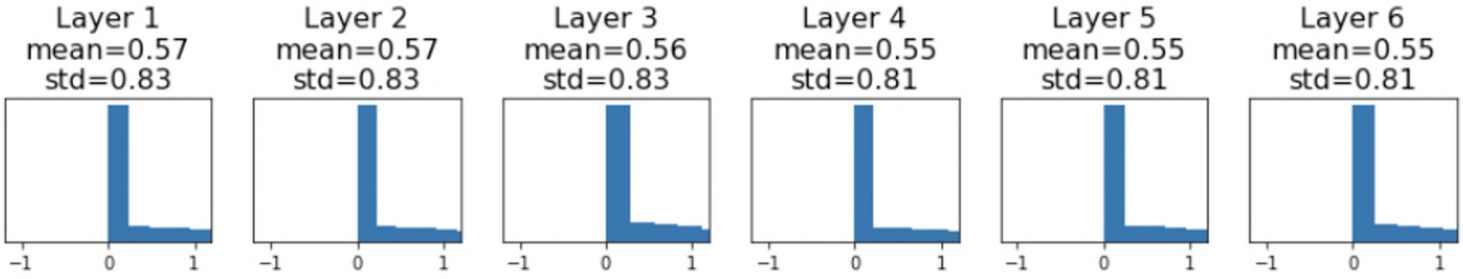


- Activations collapse to zero again!

Source: Justin Johnson

ReLU initialization

- When using ReLU $\sigma = \sqrt{2/D_{\text{in}}}$ (He et al., 2015)



- Activations nicely scaled for all layers.

Source: Justin Johnson

Batch normalization:

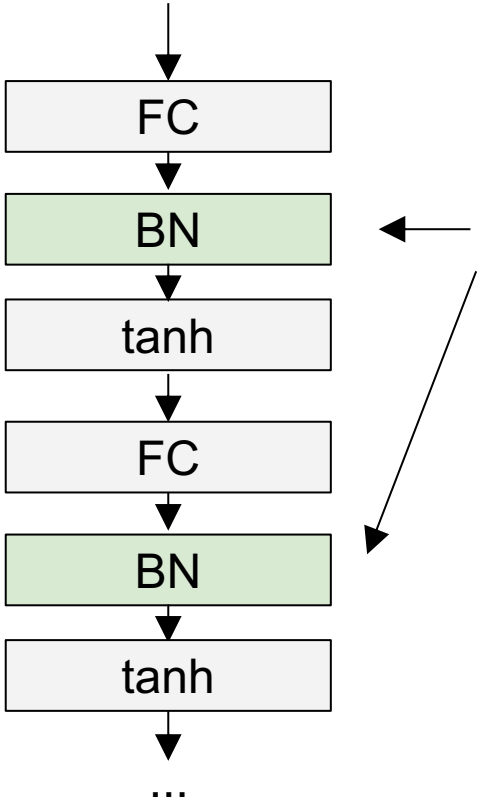
- **Key idea**: If you want a unit variance activation - just make them so.
- Statistics of activations (outputs) from a given layer across the dataset can be approximated from a mini-batch.
- For a mini-batch, compute the empirical mean and variance independently for each dimension.
- For this batch, normalize as follows (Vanilla version):

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x]}{\sqrt{Var[x] + \epsilon}}$$

$E[x]\ and\ Var[x]$ are the mean and var of the mini-batch samples

S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch normalization



Usually inserted after Fully Connected layers / (or Convolutional, as we'll see soon), and before nonlinearity.

Problem: do we really want a unit Gaussian input to a tanh layer?

Solution: Allow the network to modify the scale and bias , if needed.

Source: Stanford CS231

Batch normalization

- **Batch Normalization** – full version:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x]}{\sqrt{Var[x] + \epsilon}}$$

then allow the network to change the range if it is needed:

$$y = \gamma\, \hat{x}^{(k)} + \beta$$

where γ and β are parameters learned through the backprop.

- Note, the network can learn:

$$\gamma = \sqrt{Var[x]} \quad \text{and} \quad \beta = E[x]$$

recovering the identity mapping

Source: Stanford CS231

Batch normalization

Algorithm:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Source: Illinois CS 498

Batch normalization

Algorithm:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

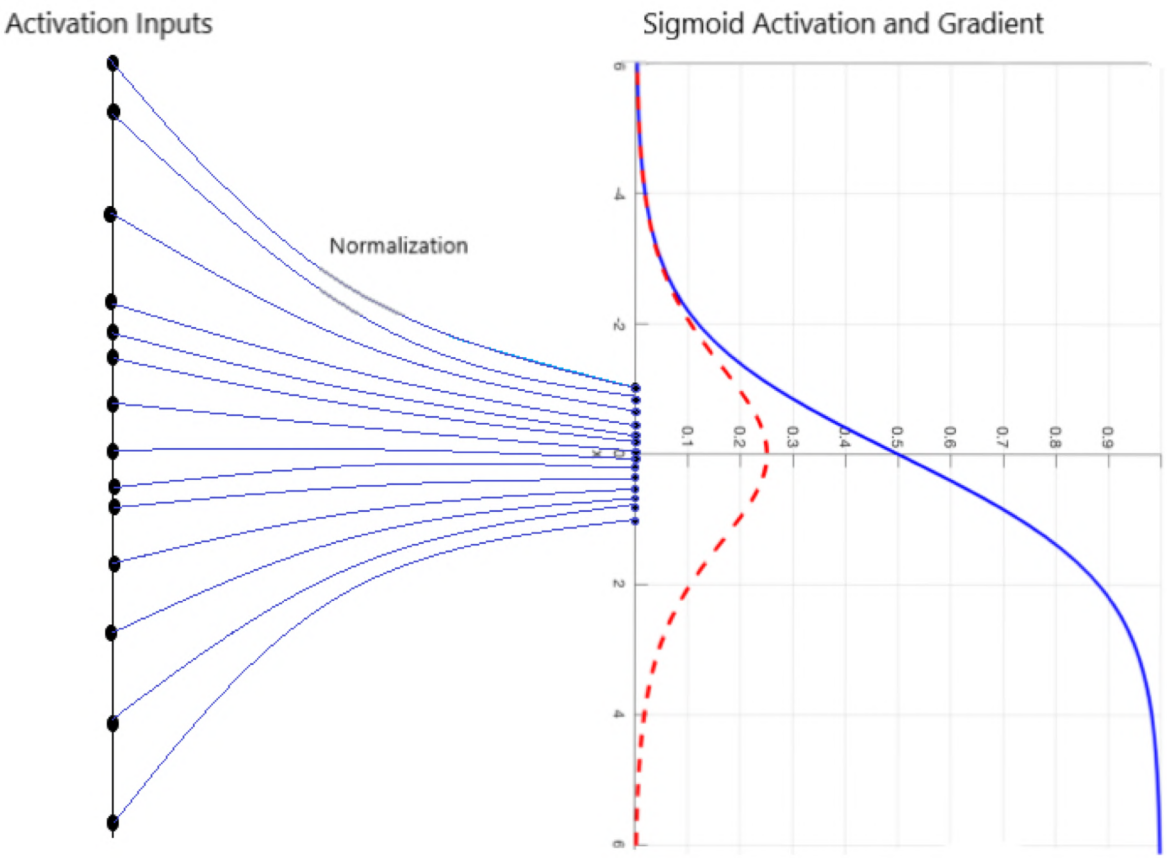
At test time Batch Norm layer functions differently:

- The mean/std are not computed based on the batch. Instead, a single fixed empirical mean/std of activations during training is used.
- Note, global mean/std can be estimated during training with running averages of x and x^2

Recall: $var(x) = E(x^2) - E^2(x)$

Source: Illinois CS 498

Batch normalization



Batch normalization

Benefits

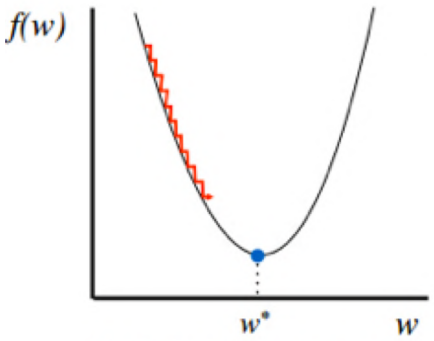
- Prevents exploding and vanishing gradients
- Keeps most activations away from saturation regions of non-linearities
- Accelerates convergence of training
- Makes training more robust w.r.t. hyperparameter choice, weight initialization

Pitfalls

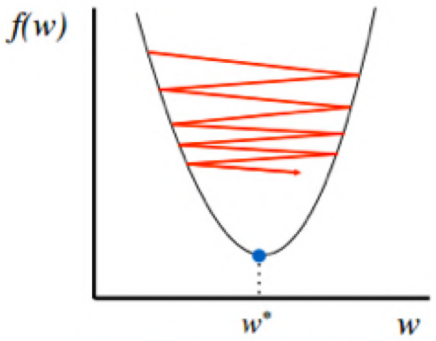
- Behavior depends on composition of mini-batches, can lead to hard-to-catch bugs if there is a mismatch between training and test regime

Source: Illinois CS 498

The Learning Rate



Too small: converge very slowly

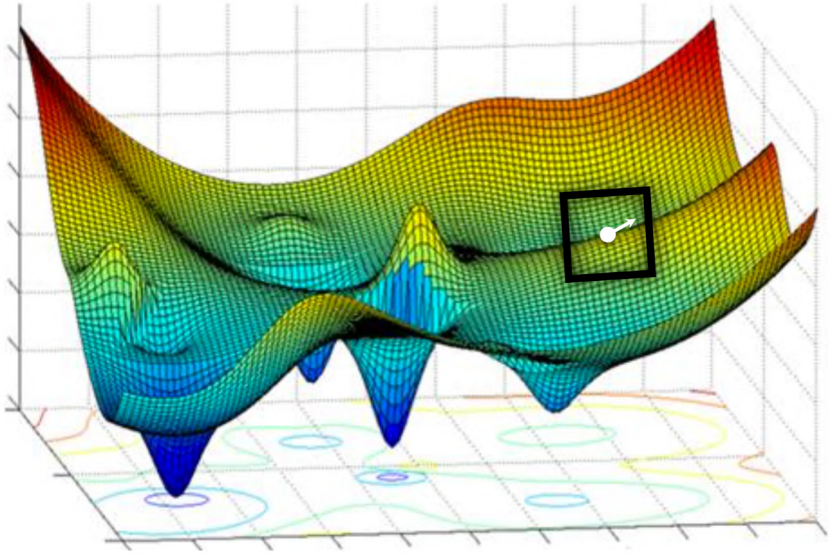


Too big: overshoot and even diverge

Gradient descent

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - \frac{\eta_t}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{L}_i(\boldsymbol{w})}{\partial \boldsymbol{w}}$$

Learning rate

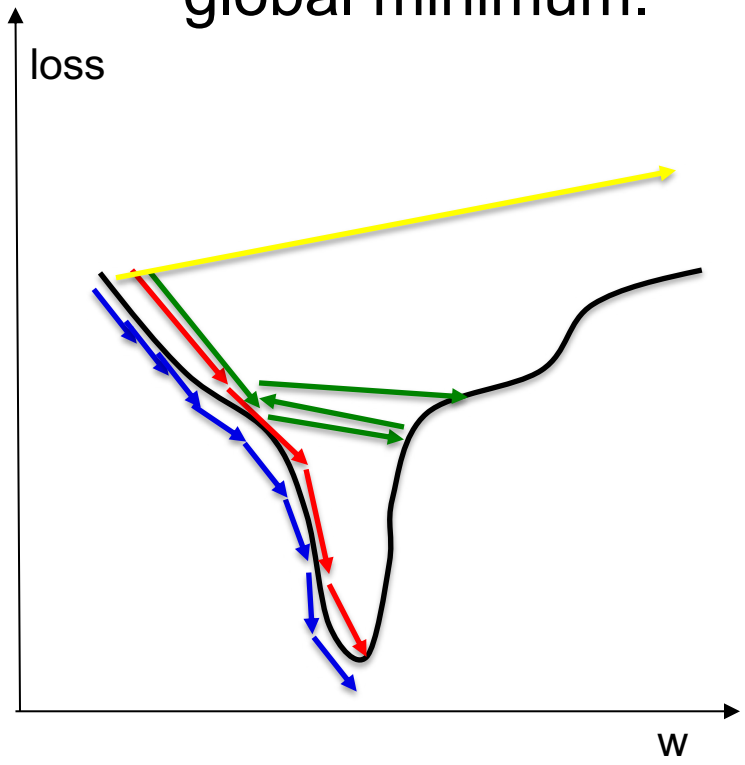


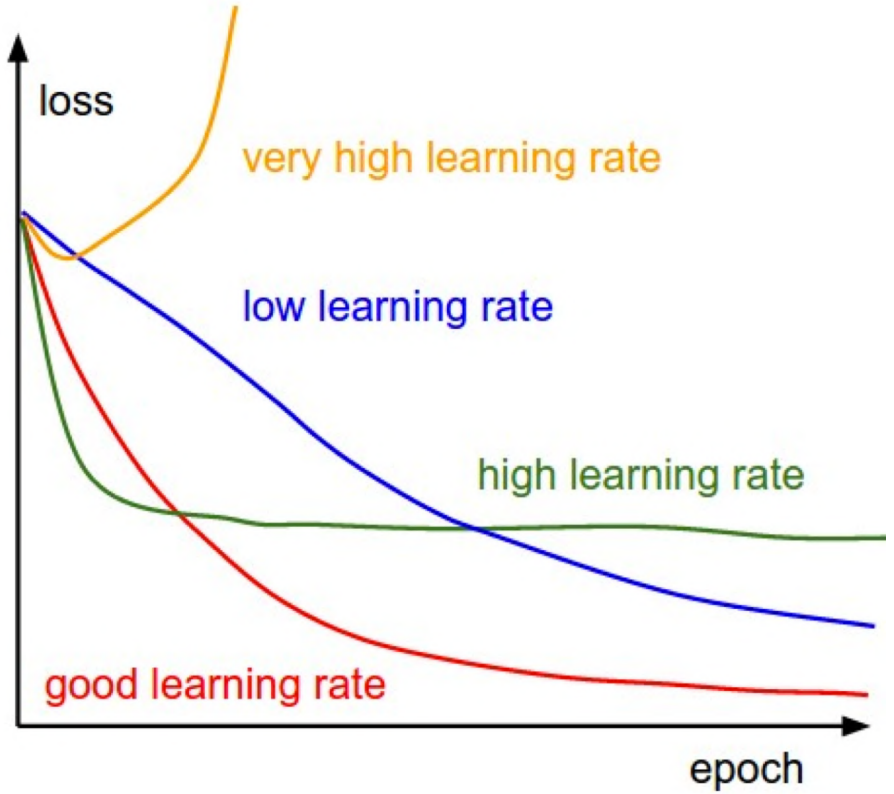
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Learning Rate

The learning rate plays a significant role in the convergence:

- If η is too small, it would take long time to converge and could stuck in a local minima or in a saddle point.
- If η is large, it may fail to converge and overshoot the global minimum.





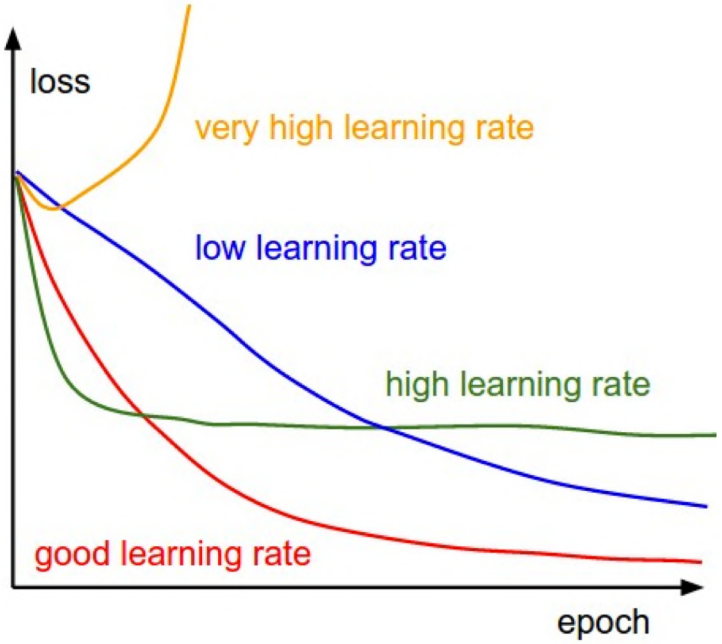
Learning Rate

How to choose the learning rate?

- Manual tuning
- Rate decay
- Momentum
- Adaptive learning rates (Adagrad, Adam)

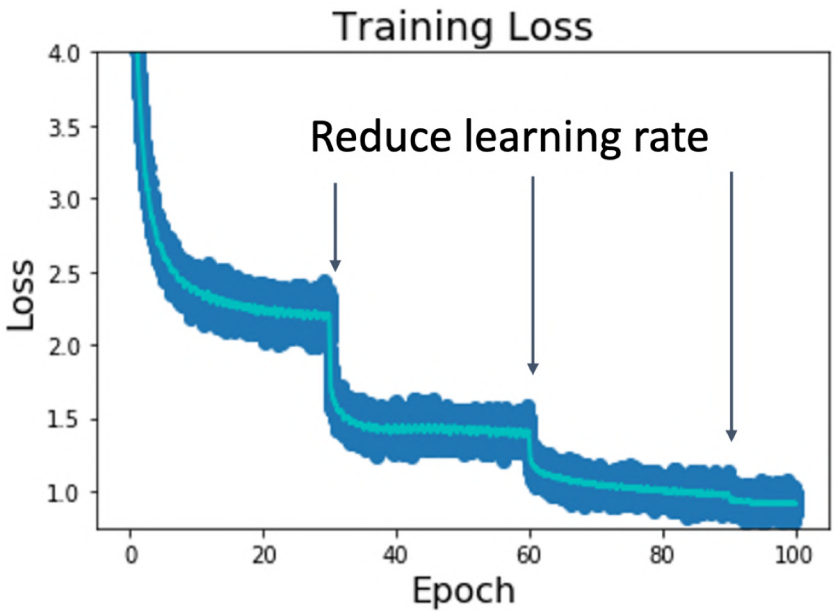
Learning Rate: One time tuning

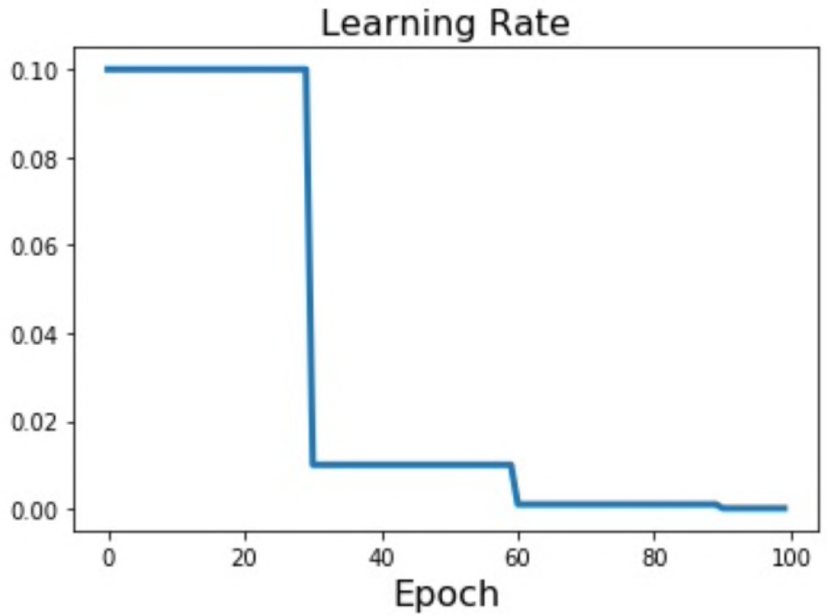
- Guess an initial learning rate.
- If the loss keeps getting worse or oscillates wildly - reduce the learning rate.
- If loss stops improving - reduce the learning rate.
- If the loss is falling fairly consistently but slowly – increase learning rate.



Learning Rate Decay: Step

Step: Reduce learning rate at a few fixed points. E.g. multiply LR by 0.1 after epochs 30, 60, and 90.

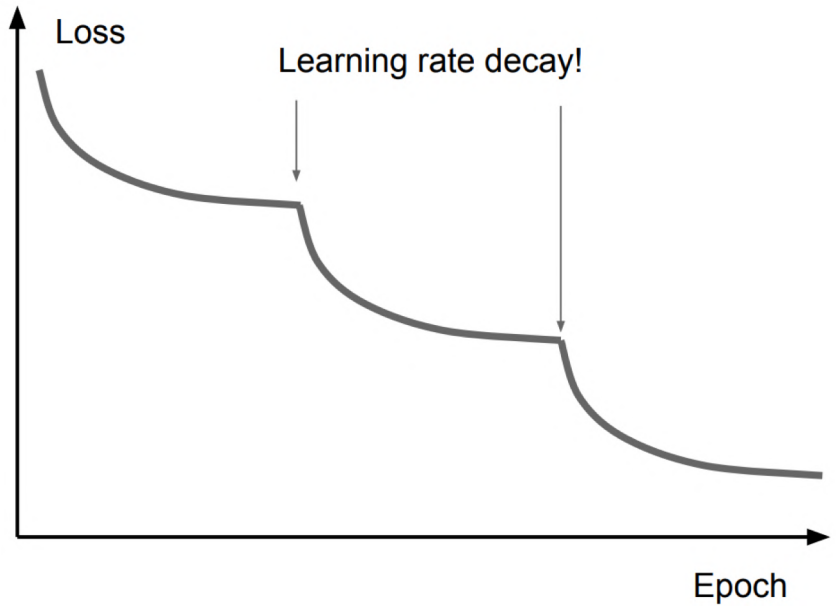




Learning Rate

Typical strategy: Use large learning rate in the early stages of the training so you can get close to the optimum. Gradually decay the learning rate to reduce the fluctuations.

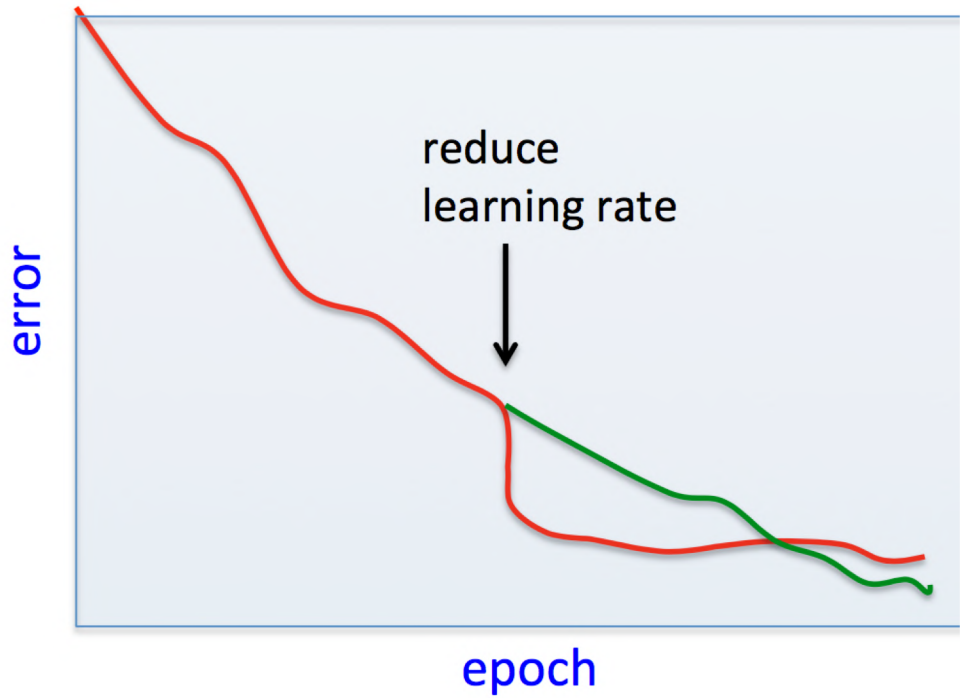
Typical loss profile:



Learning Rate

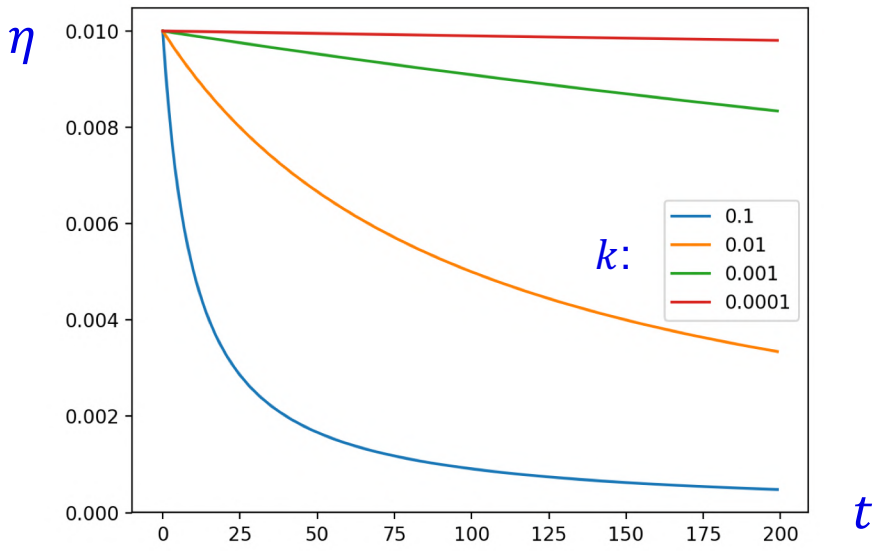
Warning: do not reduce learning-rate too soon.

You might get a quick win, but this can come at the expense of long-run performance.



Learning Rate: Online Rate Decay

- **Exponential decay**: $\eta = \eta_0 e^{-kt}$, where η_0 and k are hyperparameters, t is the iteration or epoch
- **$1/t$ decay**: $\eta = \eta_0/(1 + kt)$



1/t decay

Recall: Problems with SGD

Narrow ravines:

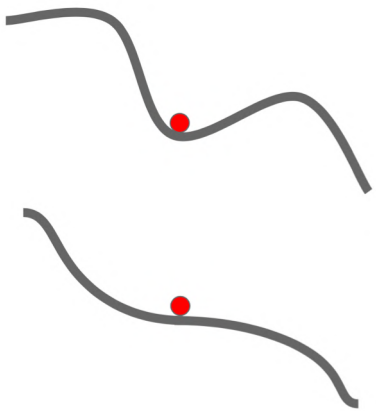
- The loss function is steep vertically but shallow horizontally.

Zero gradients:

- Zero gradient in local minima and saddle points.

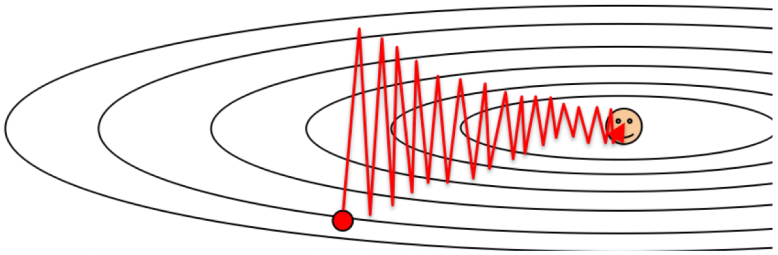
Noisy gradients:

- Especially with SGD or small mini-batch.



Local minima

Saddle point

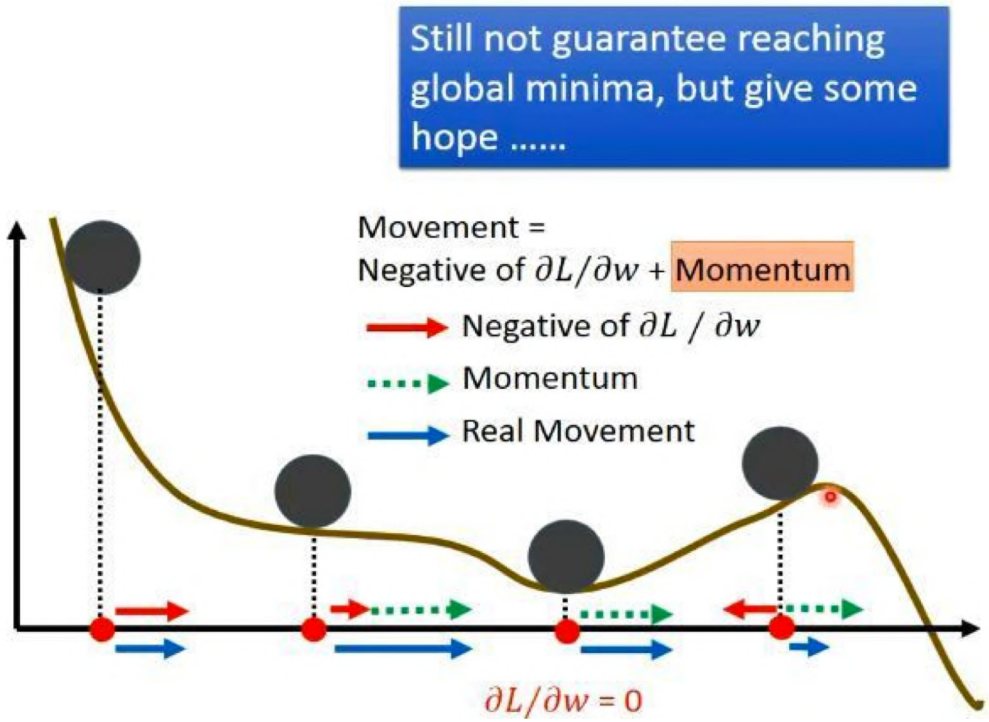


Source: Stanford CS231

Solution: Use Momentum!

Momentum:

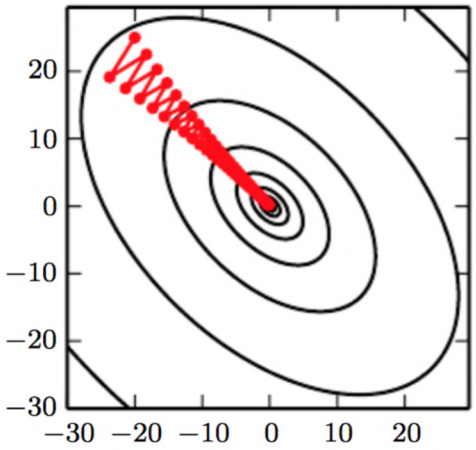
- Imagine a ball on a frictionless surface. It will accumulate speed in the downhill direction



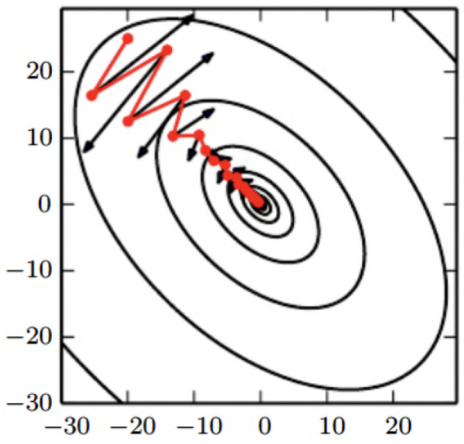
Momentum

Momentum:

- It builds up speed in directions with a gentle but consistent gradients



Without momentum



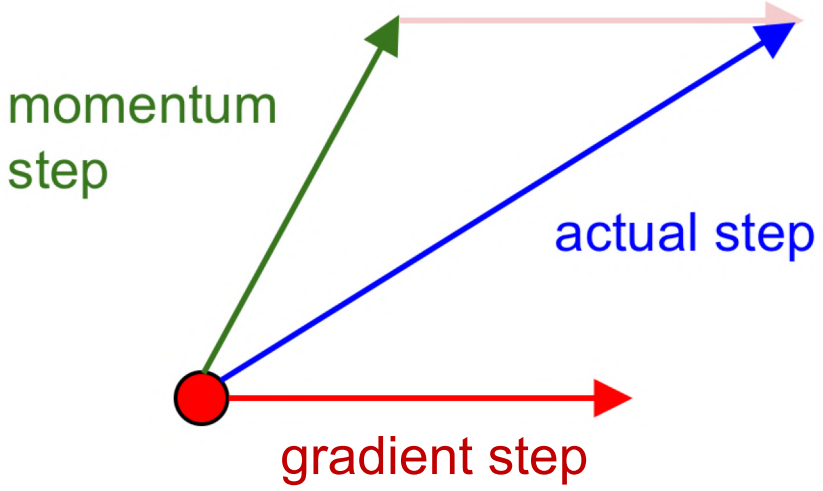
With momentum

Momentum

Momentum update:

$$\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \boldsymbol{w}}$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{v}$$

- η is the learning rate, just like in gradient descent.
- α is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99).



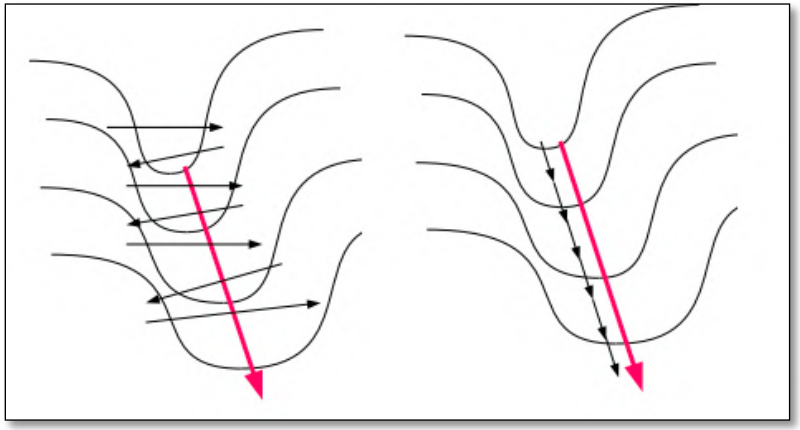
Source: Stanford CS231

Momentum

- Momentum update:

$$\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \boldsymbol{w}}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{v}$$

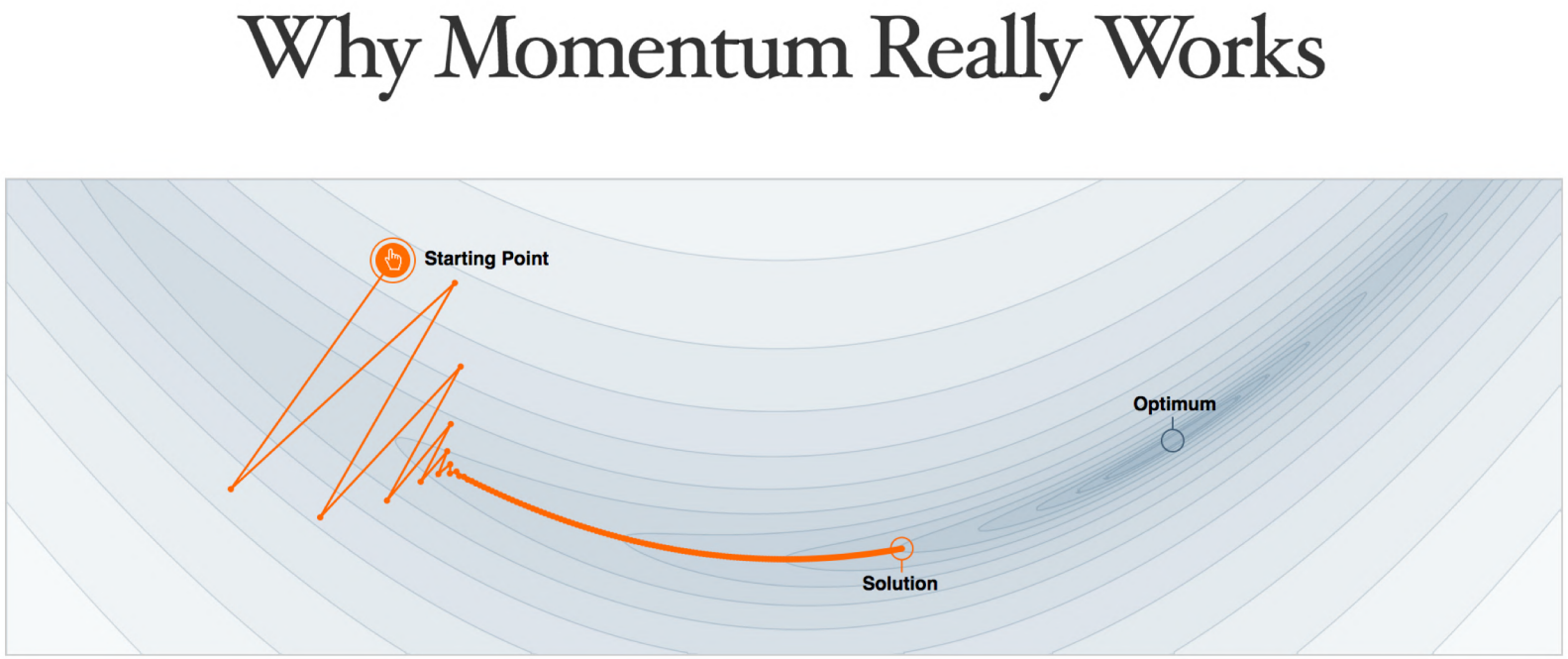


- The effect of the gradient is to increment the previous velocity. The velocity also decays by α which is slightly less than 1.
- Along the high curvature directions, the gradients cancel each other, so momentum dampens the oscillations.
- Along the low curvature directions, the gradients point in the same direction, allowing the parameters to build up speed.

Source: Roger Grosse and Geoffrey Hinton

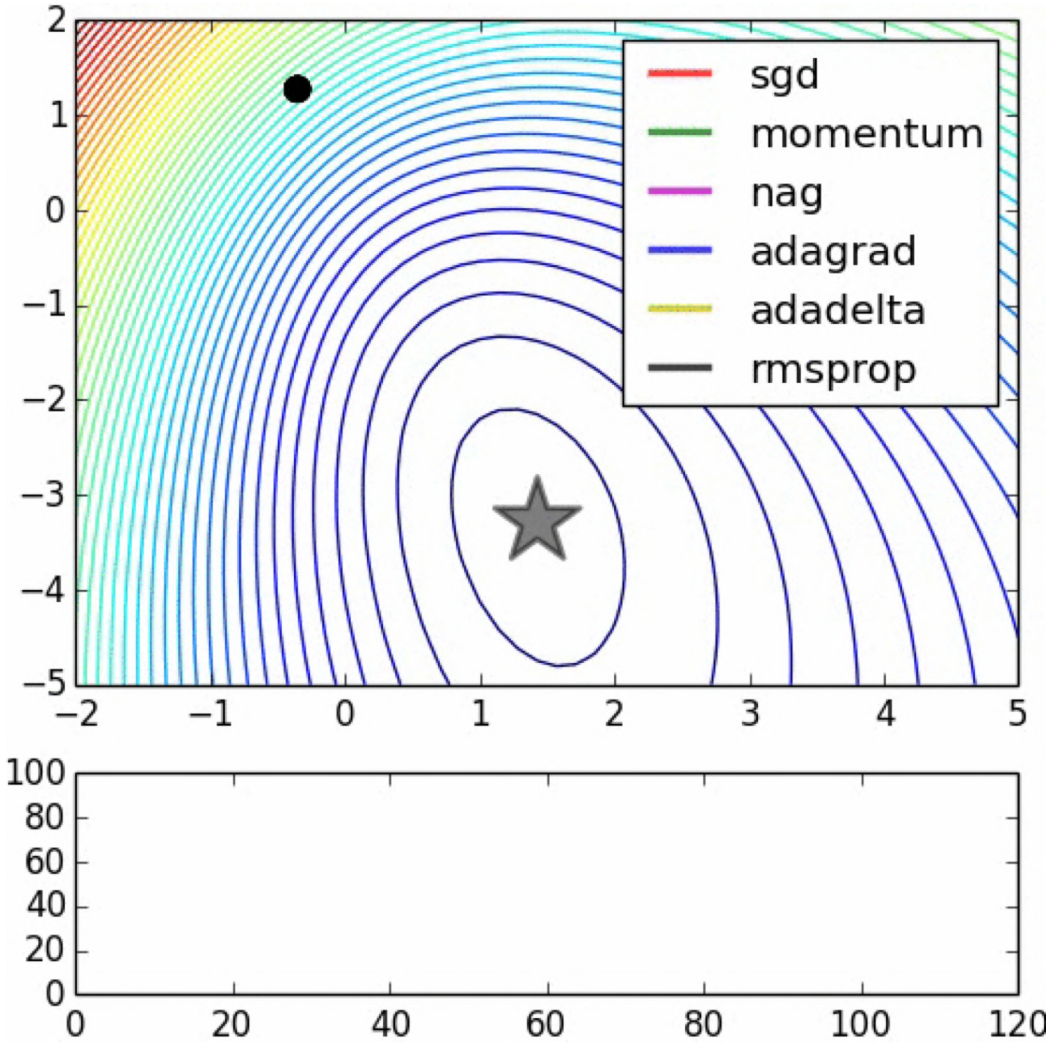
Momentum

Demo time:



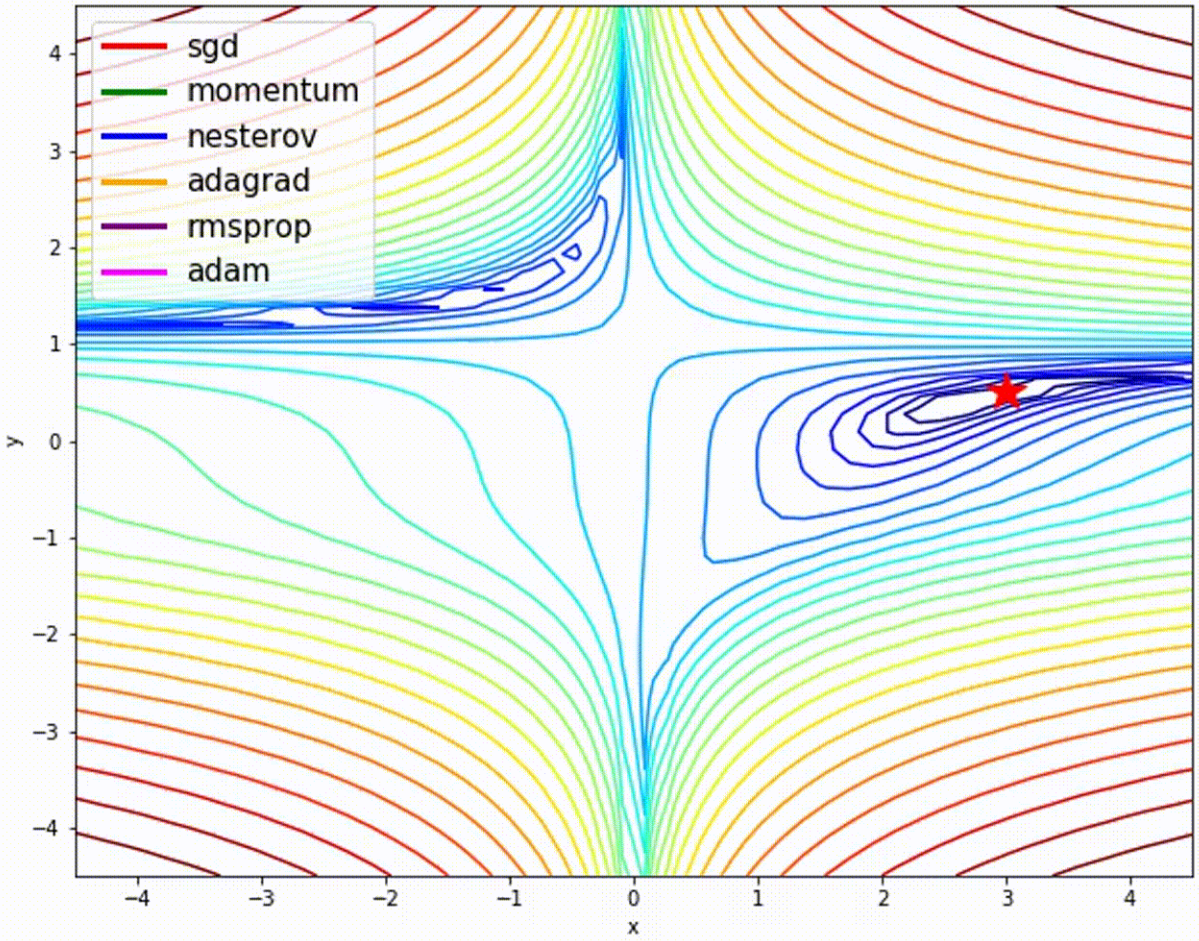
https://distill.pub/2017/momentum/

Momentum



Source: Stanford CS231

Momentum



Source: Stanford CS231

Nesterov Momentum

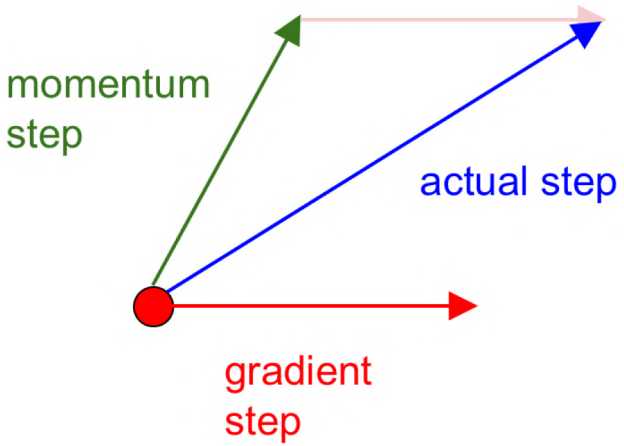
- Standard momentum:

$$\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \boldsymbol{w}} \quad ; \quad \boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{v}$$

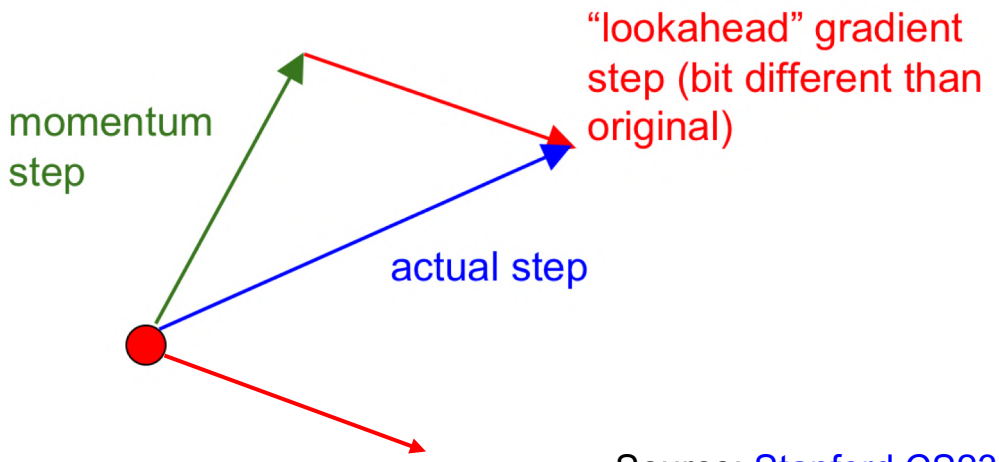
- Nesterov momentum evaluates gradient at "lookahead" position $\boldsymbol{w} + \alpha\, \boldsymbol{v}$:

$$\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta \frac{\partial \mathcal{L}(\boldsymbol{w} + \boldsymbol{v})}{\partial \boldsymbol{w}} \quad ; \quad \boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{v}$$

Momentum update

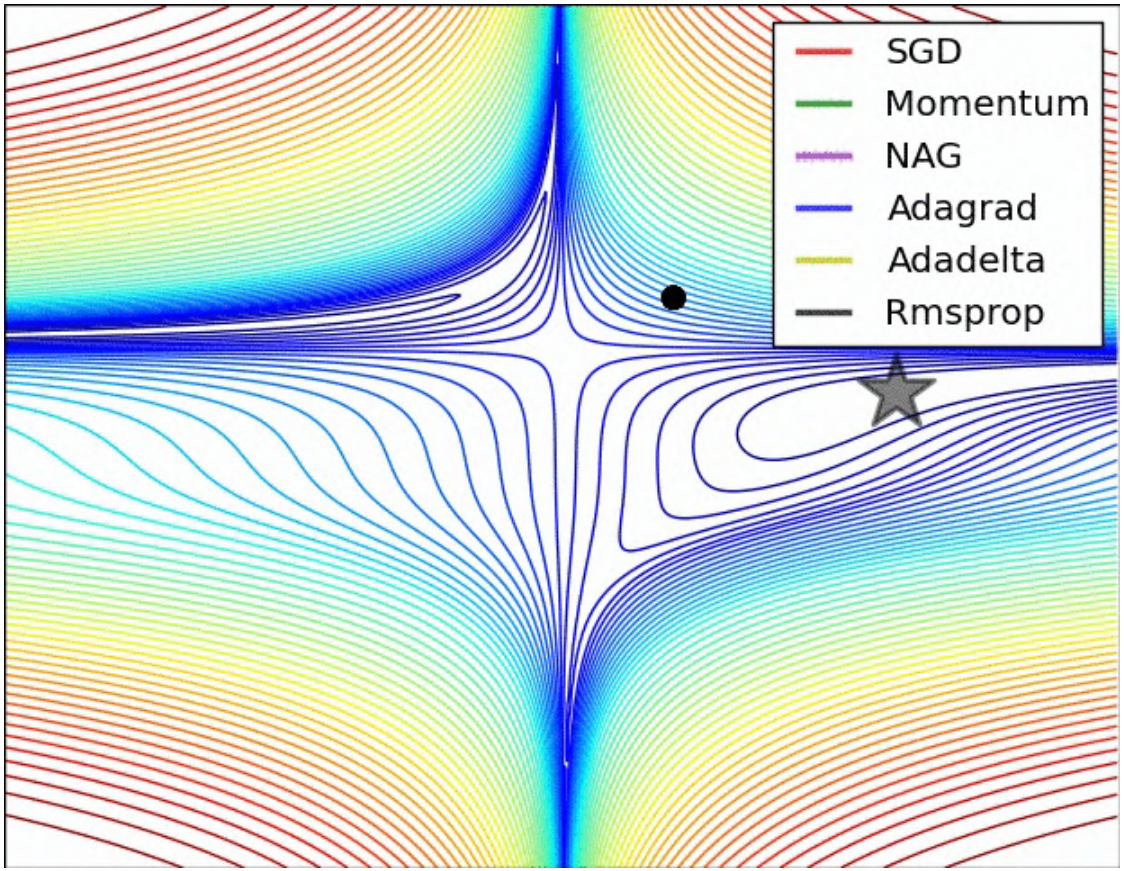


Nesterov update



Source: Stanford CS231

Nesterov Momentum



Source: Stanford CS231

Adaptive learning rates

- The magnitude of the gradient can vary widely between **different weights (**different directions**)** and can change during learning.
- This makes it hard to choose a **single** learning rate for all weights.
- **Goal**: an automatic way to tune the learning rates adaptively for **each weight.**

- **RMSProp**: Keep track the history of the gradient magnitudes, scale the learning rate for each parameter based on its own history

Source: Stanford CS231

Adaptive learning rates: RMSProp

- Scale the learning rate for each parameter based on its variance (s_k) history:

$$s_k \leftarrow \beta \cdot s_k + (1-\beta) \left\| \frac{\partial \mathcal{L}}{\partial w_k} \right\|^2$$

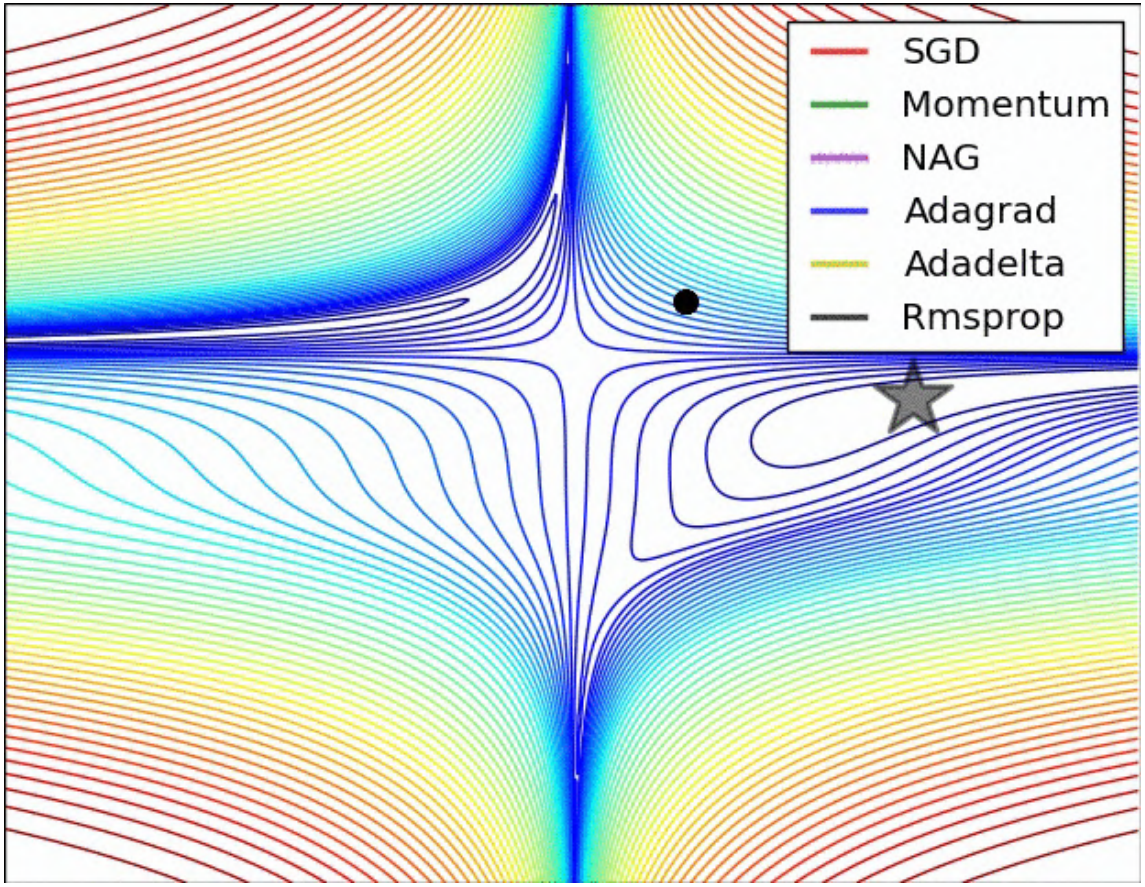
$$w_k \leftarrow w_k - \frac{\eta}{\sqrt{s_k}+\epsilon} \frac{\partial L}{\partial w_k}$$

w_k indicates a scalar (an axis in the weight space)

- Parameters with small variance get large updates and vice versa.
- Helps escapes quickly from plateaus with tiny gradients.
- Damps down oscillating path in narrow ravines.
- The decay factor β (typically ≥ 0.9) down-weight past variances exponentially.

Source: Illinois CS 498

Adaptive learning rates



Adam: adaptive learning + momentum

- Combine RMSProp with momentum:

$$v_k \leftarrow \alpha\, v_k + (1-\alpha)\frac{\partial \mathcal{L}}{\partial w_k}$$

$$s_k \leftarrow \beta\, s_k + (1-\beta)\left\|\frac{\partial L}{\partial w_k}\right\|^2$$

$$w_k \leftarrow w_k - \frac{\eta}{\sqrt{s_k}+\epsilon} v_k$$

- Default parameters from paper:

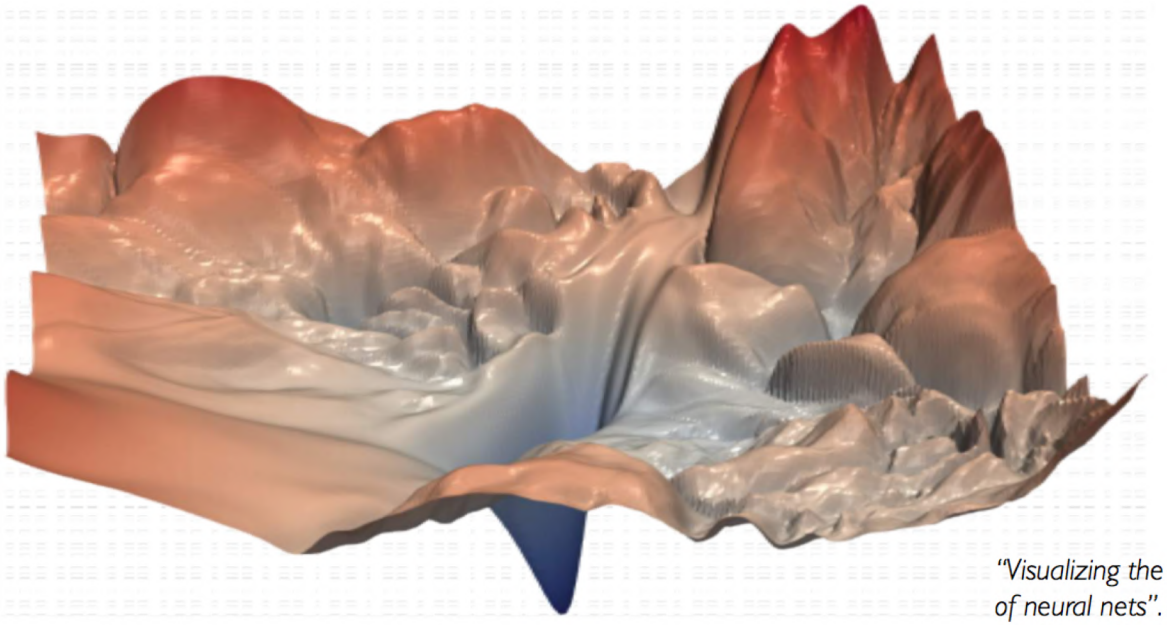
$$\alpha = 0.9,\ \beta = 0.999,\ \epsilon = 1e-8$$

D. Kingma and J. Ba, Adam: A method for stochastic optimization

Source: Illinois CS 498

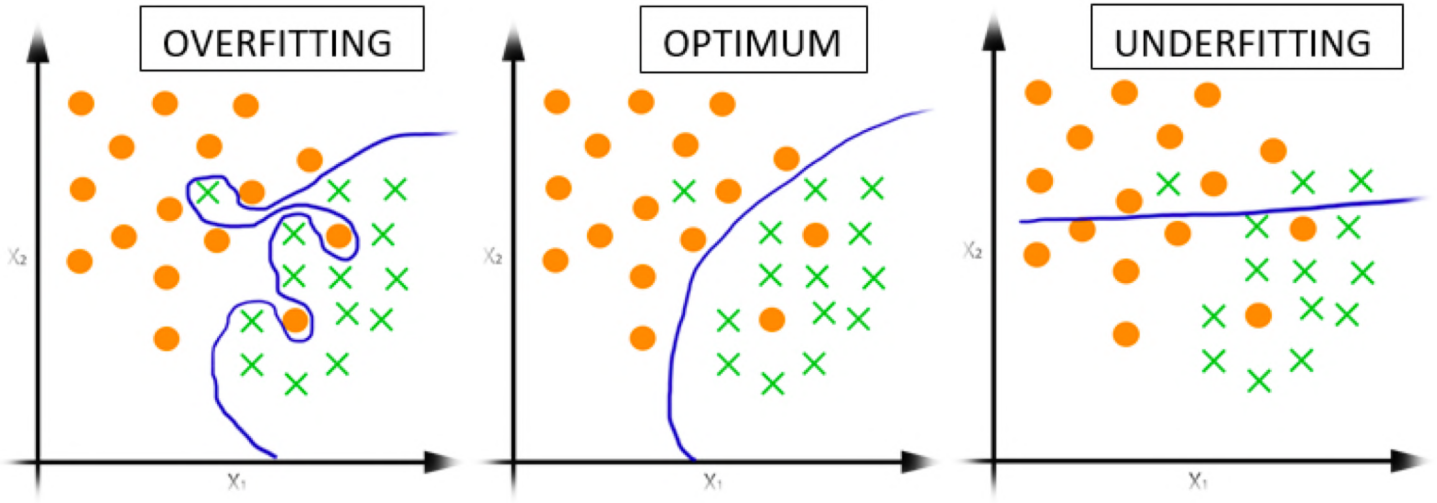
Which optimizer to use in practice?

- **Adam** is a good default choice in many cases
- **SGD+Momentum** with learning rate decay often outperforms Adam by a bit, but requires more tuning
- All methods have **learning rate** as a hyperparameter (more critical with SGD+Momentum, less critical with Adam)
- But reaching a global minimum is still a challenge:



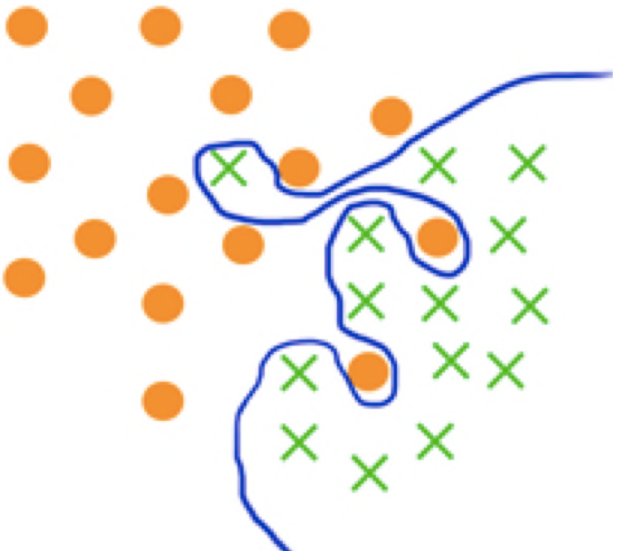
"Visualizing the loss landscape of neural nets". Dec 2017.

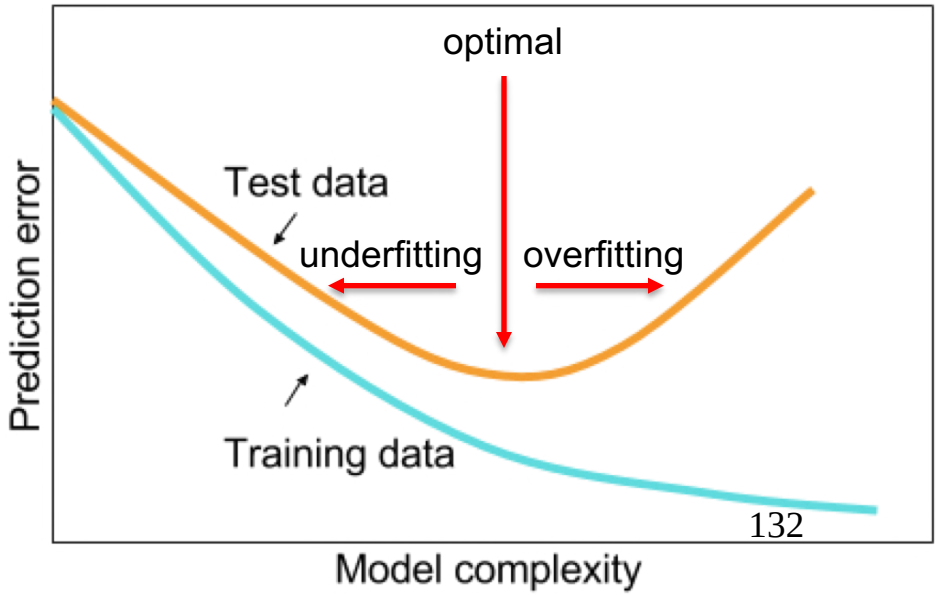
Regularization



Regularization

- Data overfitting is a common problem where we lack data and the model capacity is too high
- Regularization is ***any modification we make to the learning algorithm that is intended to reduce the generalization error, without adding more data*** .

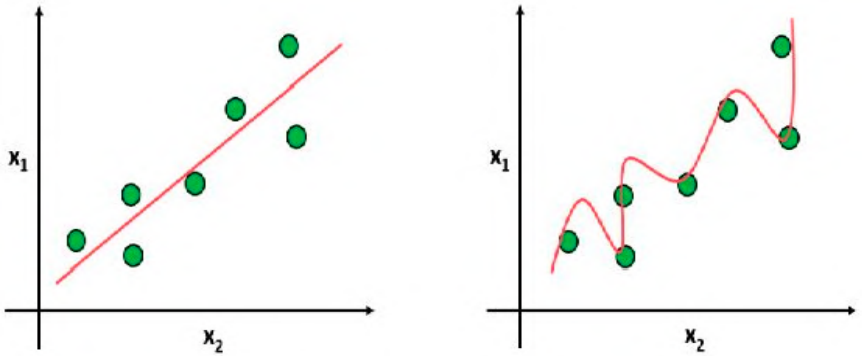




Regularization Techniques

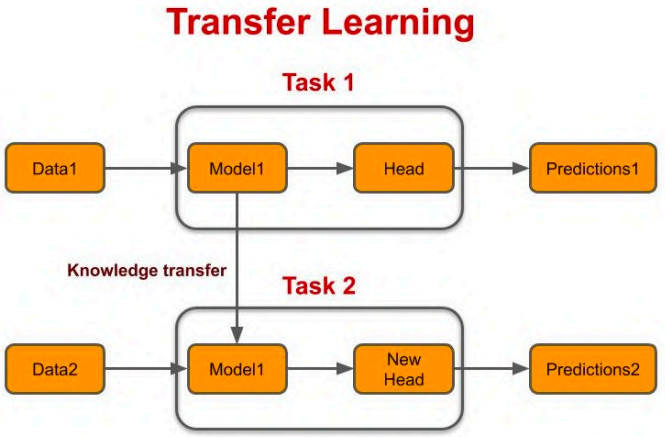
Approach 1: Constrain your model

- Early stopping
- L2/L2 weight decay
- Dropout / drop-connect



Approach 2: Get more data

- Data augmentation
- Transfer Learning
- Self-supervised learning



Regularization Techniques

- Early stopping
- L2/L2 weight decay
- Dropout / drop-connect
- Data augmentation
- Transfer Learning
- Self-supervised learning

Constrain the optimization process

Constrain the parameters

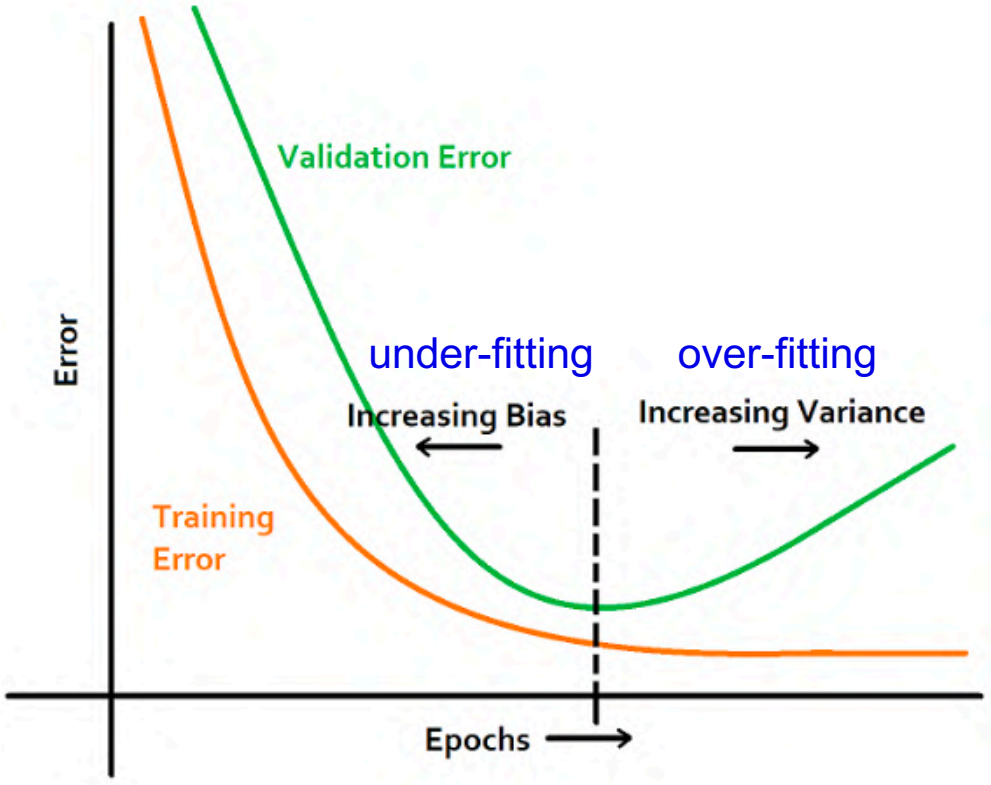
Constrain the architecture

Enrich your samples

Use prior knowledge

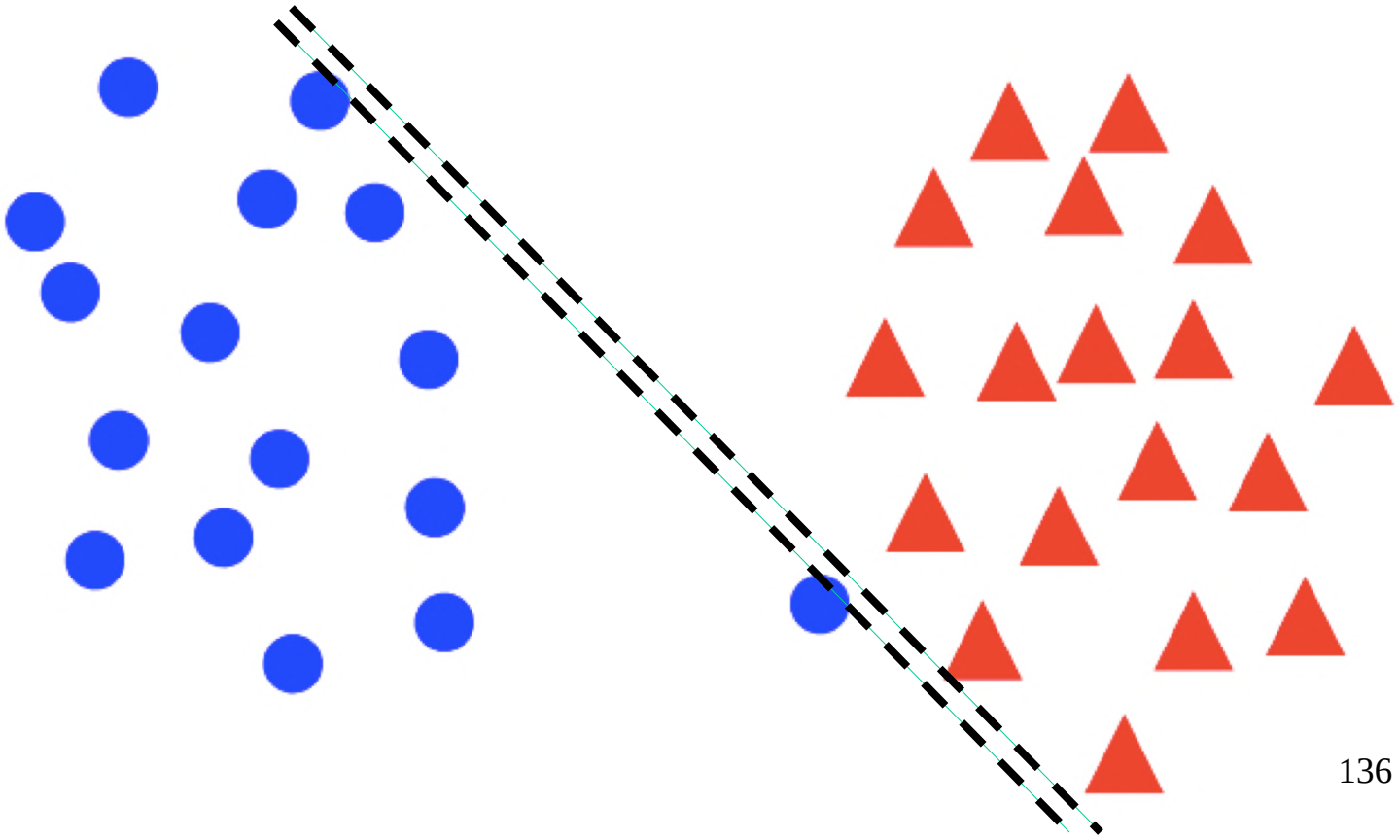
Regularization: Early stopping

- **Idea**: do not train a network to achieve too low training error
- Monitor validation error to decide when to stop



Regularization: Weight Decay

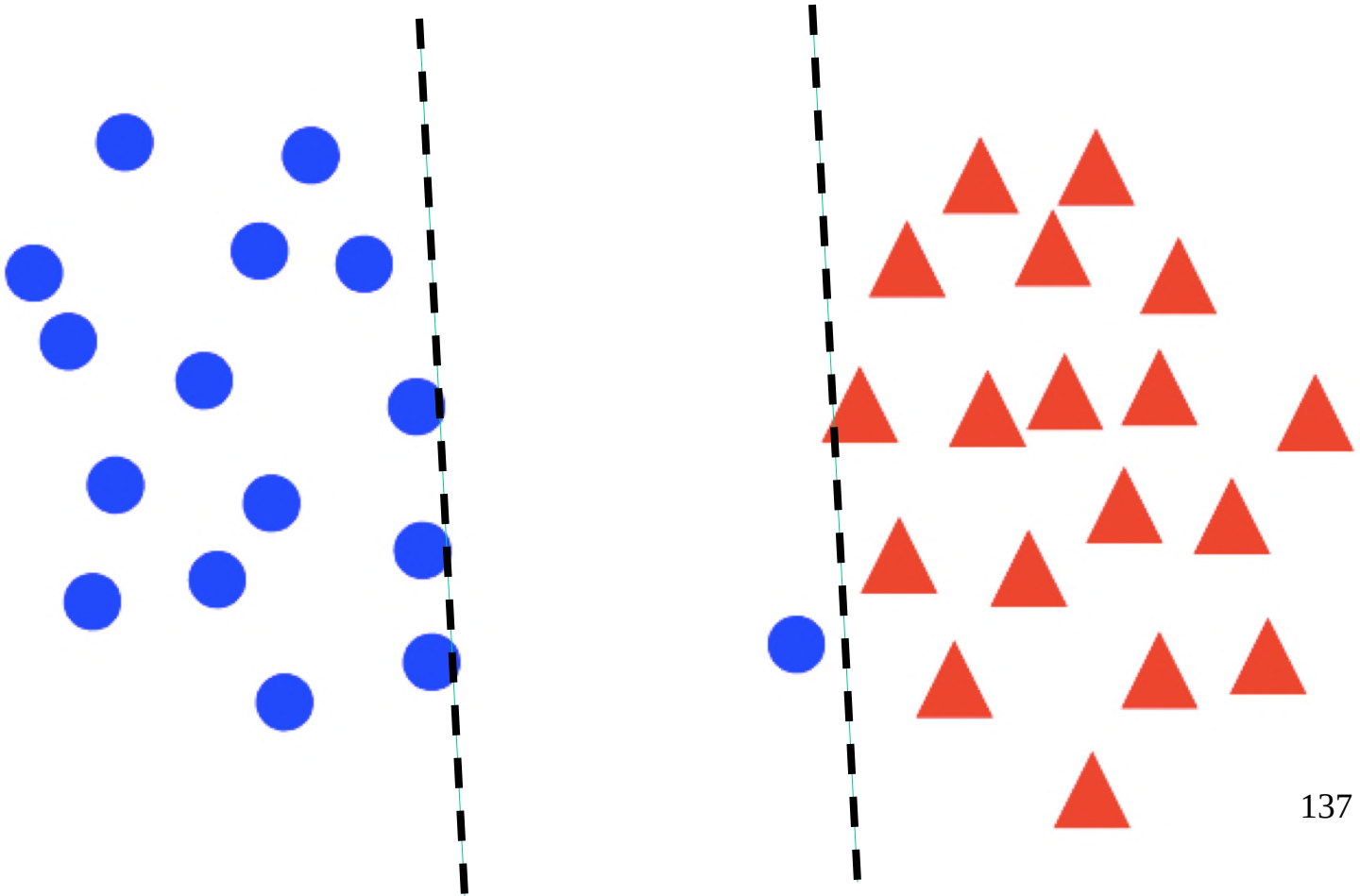
- How to separate between non-separable data, or even separable data?
- We may prefer a larger margin with a few constraints violated



Source

Regularization: Weight Decay

- How to separate between non-separable data, or even separable data?
- We may prefer a larger margin with a few constraints violated

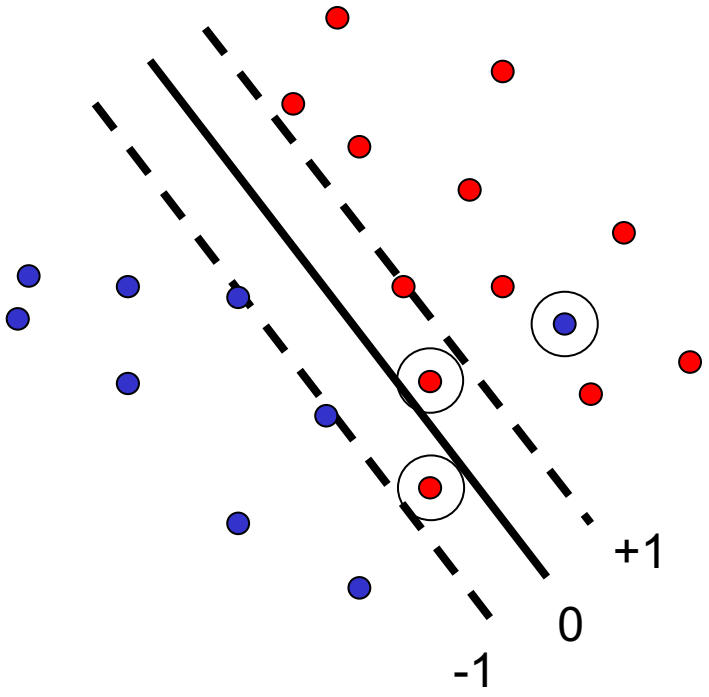


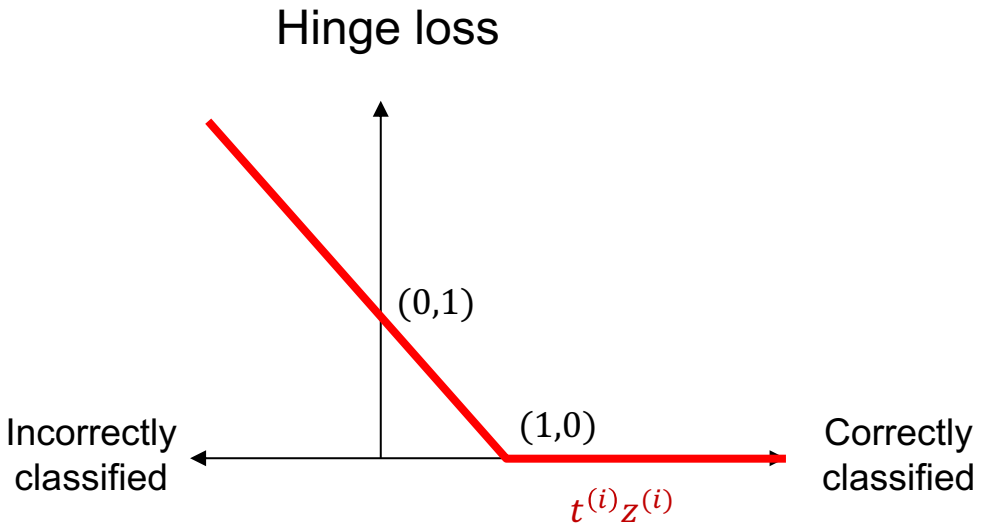
Source

“Soft margin” formulation

- Penalize margin violations using *Hinge loss*:

$$\min_w \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \max[0, 1 - t^{(i)} z^{(i)}]$$





“Soft margin” formulation

- Penalize margin violations using *hinge loss*:

$$\min_w \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \max[0, 1 - t^{(i)} z^{(i)}]$$

Maximize margin – a.k.a. ***regularization***

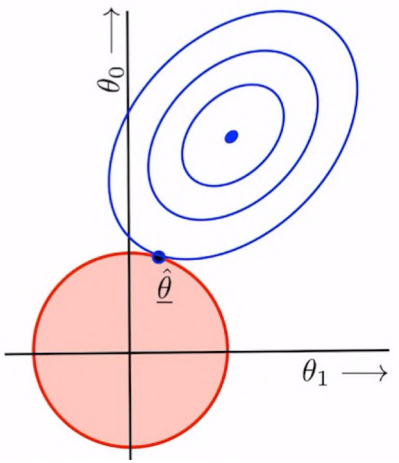
Minimize misclassification loss

- The term $\frac{\lambda}{2}\|\boldsymbol{w}\|^2$ is borrowed as a regularization term for DNNs to constrain parameters :

$$\mathcal{L}_{new} = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \mathcal{L}(w)$$

Regularization

Minimize misclassification loss



L2 regularization (weight decay)

- Regularized objective:

$$\mathcal{E}(w) = \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{n}\sum_{i=1}^{n} l(w, x^{(i)}, y^{(i)})$$

- Gradient of objective:

$$\nabla_w \mathcal{E}(w) = \lambda w + \frac{1}{n}\sum_{i=1}^{n} \nabla_w l(w, x^{(i)}, y^{(i)})$$

- SGD update:

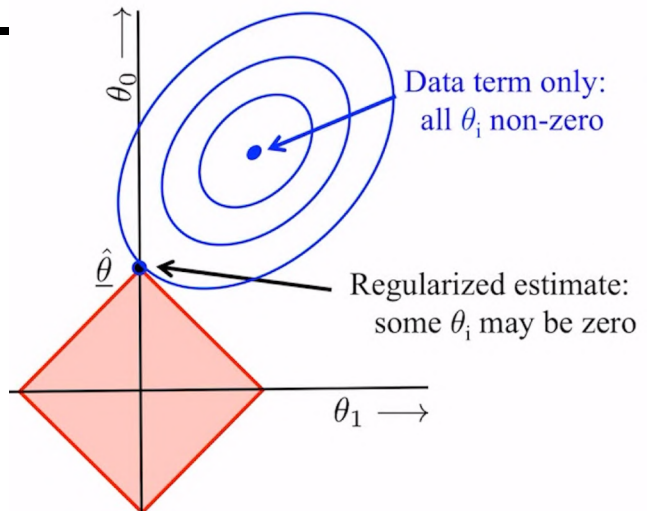
$$w \leftarrow w - \eta\lambda w - \eta\nabla_w l\left(w, x^{(i)}, y^{(i)}\right)$$

L1 regularization

- Regularized objective:

$$\mathcal{E}(w) = \lambda \|w\|_1 + \frac{1}{n}\sum_{i=1}^{n} l\left(w, x^{(i)}, y^{(i)}\right)$$

where $\|w\|_1 = \sum_j |w_j|$



- Gradient: $\nabla_w \mathcal{E}(w) = \lambda \operatorname{sgn}(w) + \frac{1}{n}\sum_{i=1}^{n} \nabla l(w, x^{(i)}, y^{(i)})$

- SGD update:

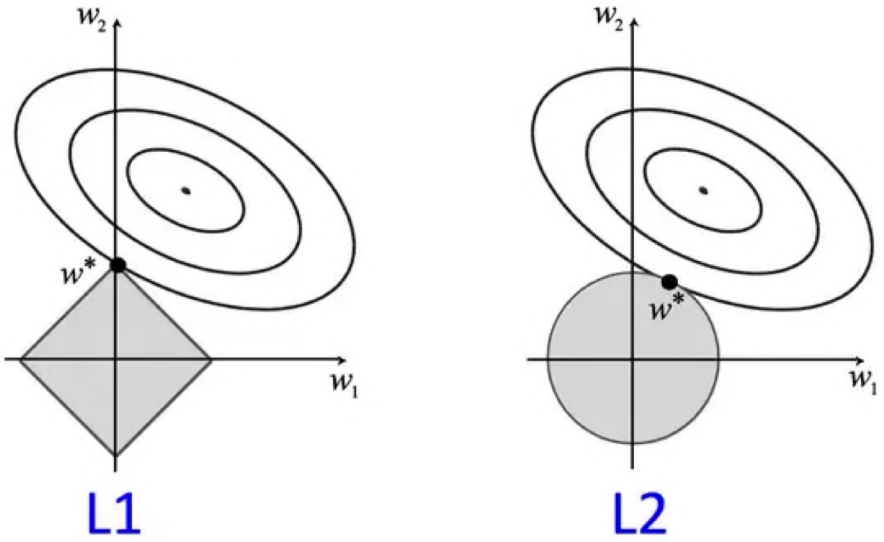
$$w \leftarrow w - \eta\lambda \operatorname{sgn}(w) - \eta \nabla l\left(w, x^{(i)}, y^{(i)}\right)$$

- Interpretation: Encouraging sparsity

L2/L1 Regularization: Motivation

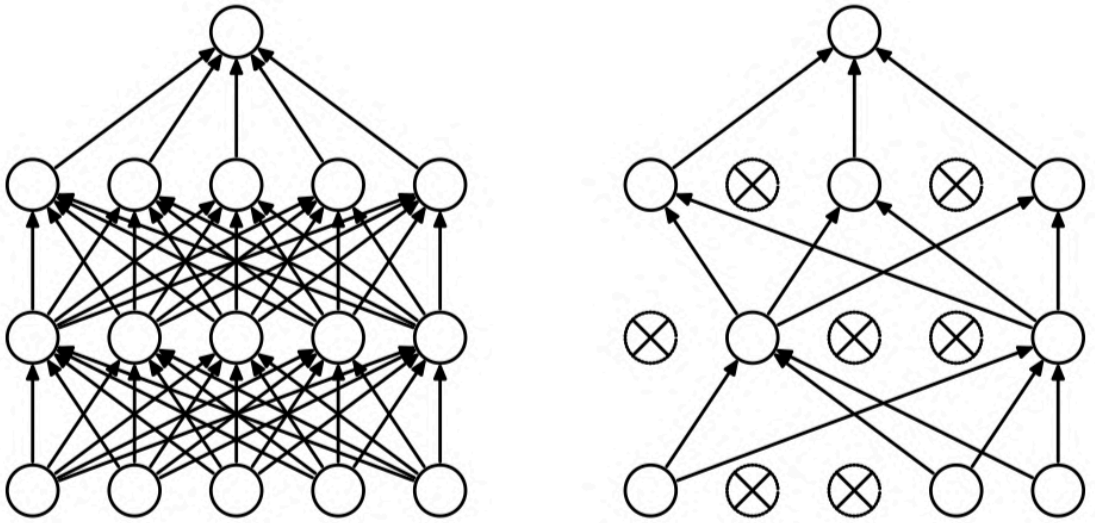
- $\boldsymbol{x} = [1 \quad 1 \quad 1 \quad 1]$
- $\boldsymbol{w}_1 = [1 \quad 0 \quad 0 \quad 0]$ ← Optimizes L_1
- $\boldsymbol{w}_2 = [0.25 \quad 0.25 \quad 0.25 \quad 0.25]$ ← Optimizes L_2

$$\boldsymbol{w}_1^T \boldsymbol{x} = \boldsymbol{w}_2^T \boldsymbol{x} = 1$$



Regularization: dropout

- At training time, in each forward pass, turn off neurons with probability *1-p*
- At test time, to have deterministic behavior, use the full network and multiply output of neuron by *p*



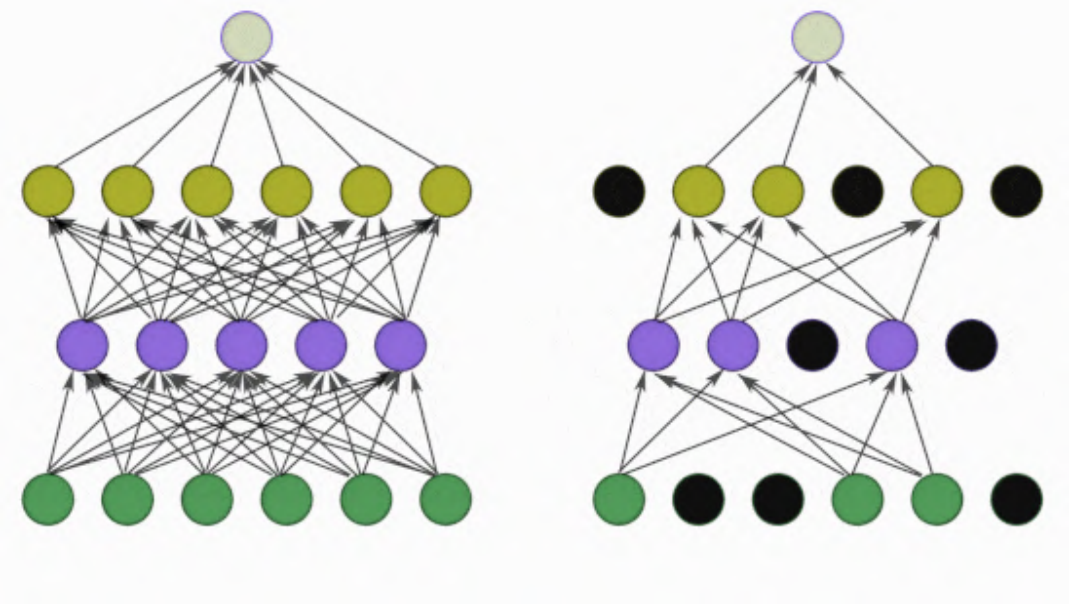
(a) Standard Neural Net

(b) After applying dropout.

Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Regularization: dropout

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p # first dropout mask
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p # second dropout mask
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

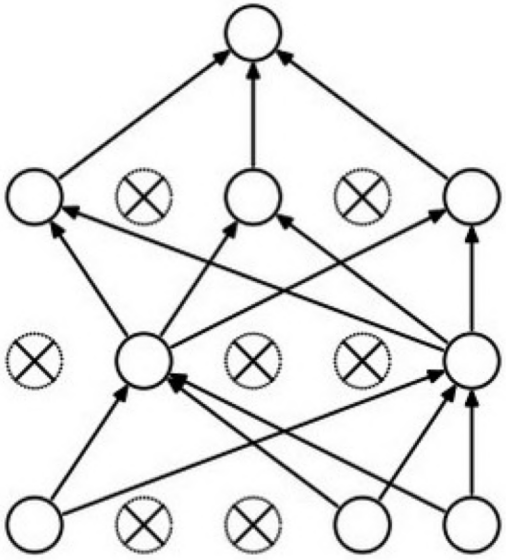


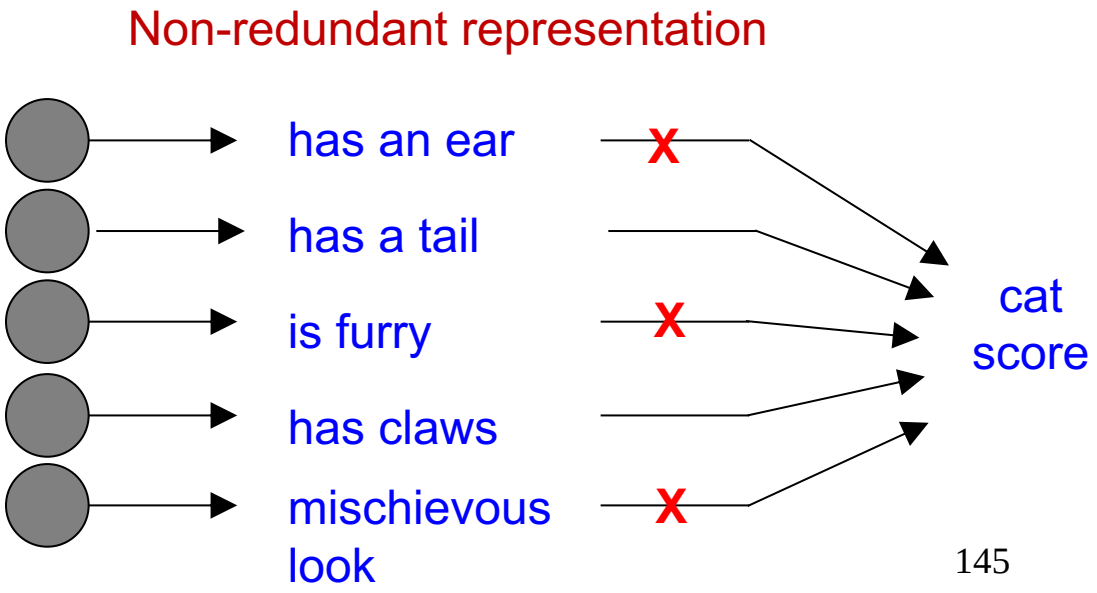
Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Regularization: dropout

Intuitions:

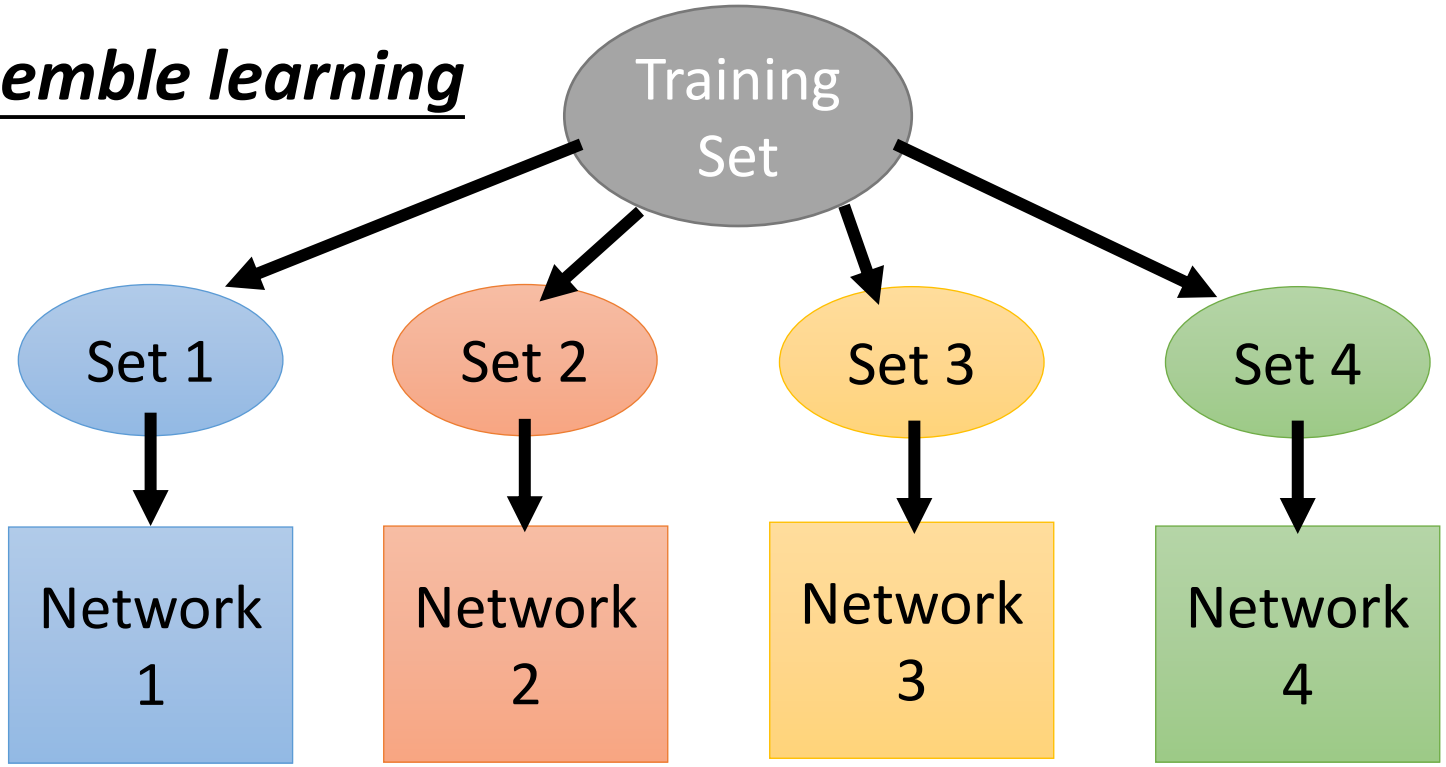
1. Forces the network to have a redundant representation
2. Decreases the complexity of the network
3. Can be seen as a large ensemble of networks that share parameters





Source: Stanford CS231

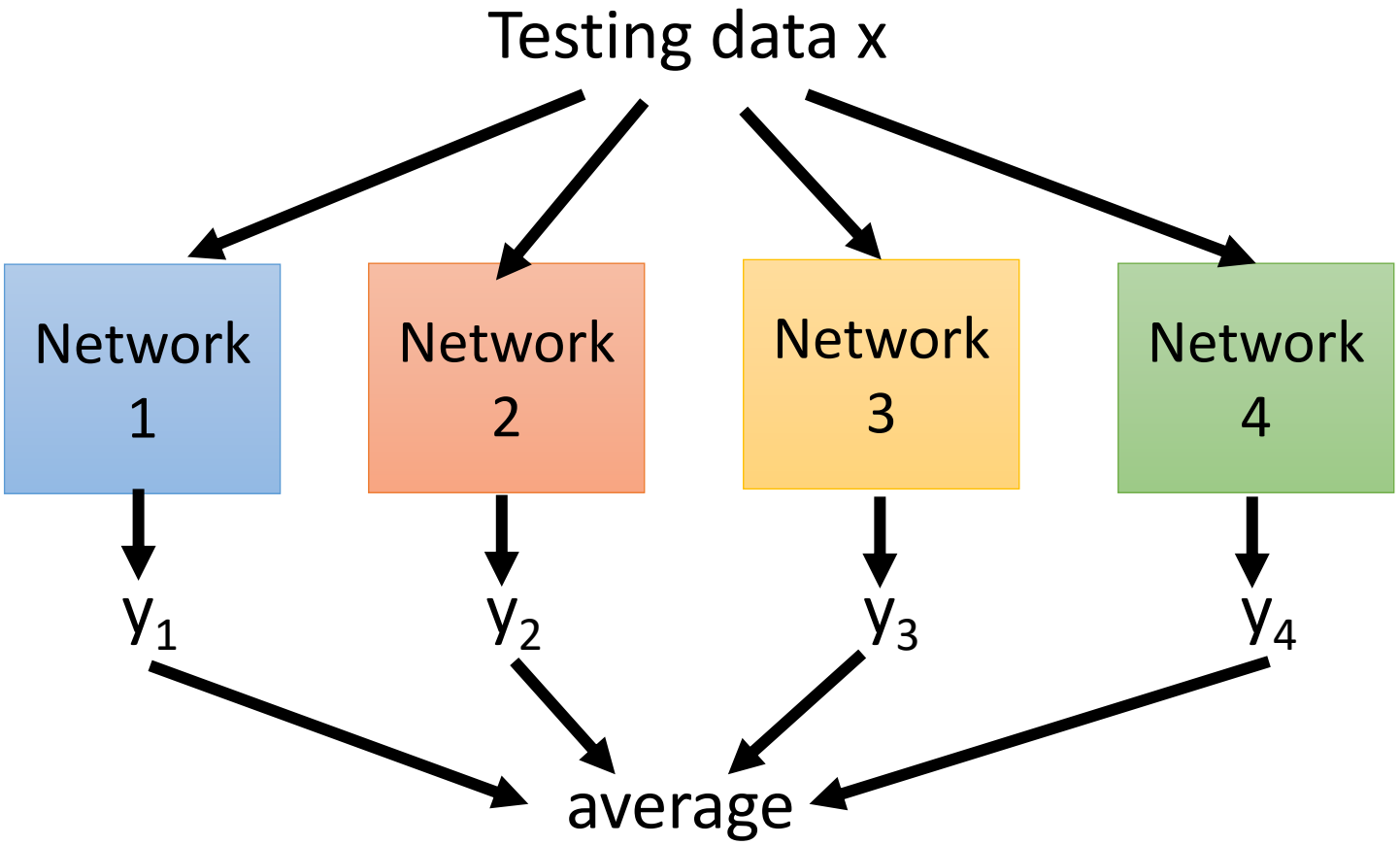
Ensemble learning



Train a bunch of networks with different structures

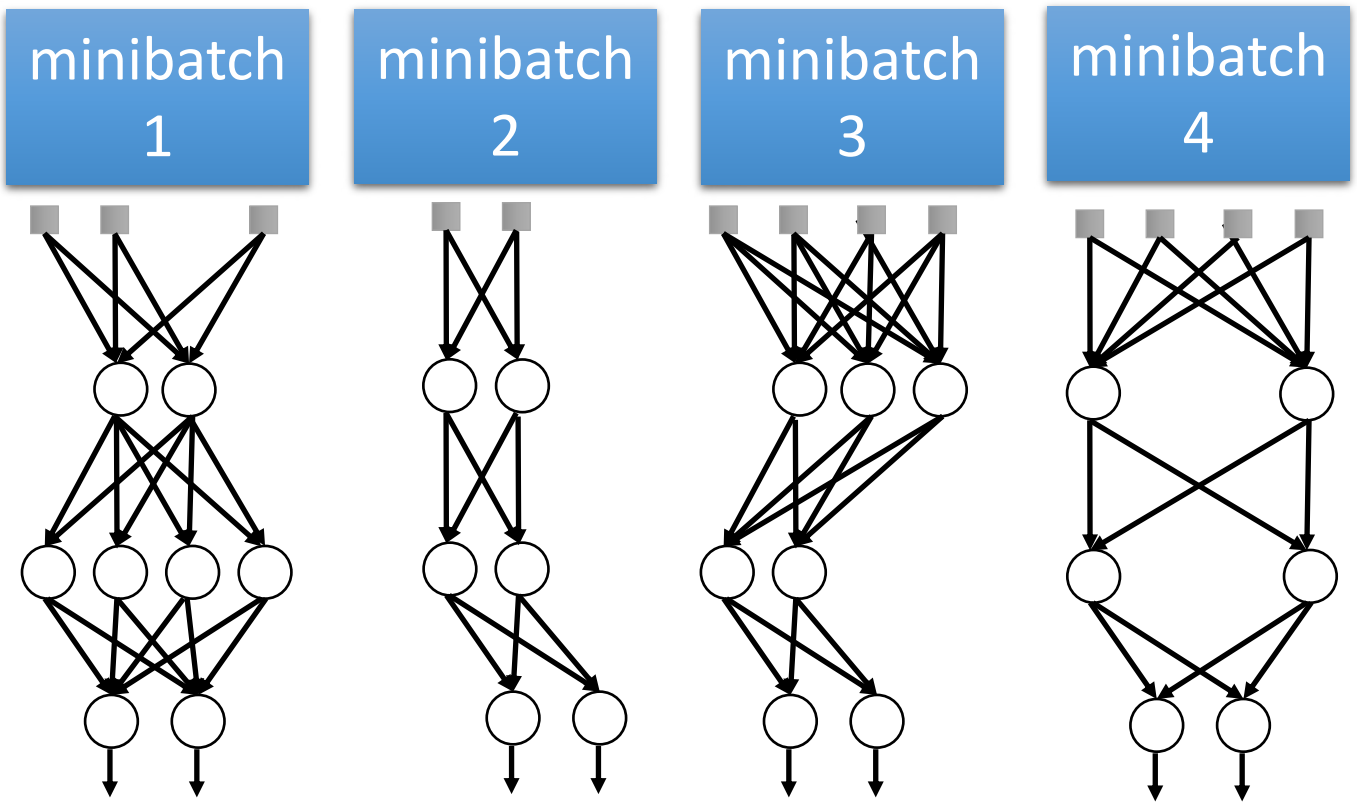
Source: Lee Hung-yi

Ensemble learning



Reduces generalization error

Source: Lee Hung-yi



Training of Dropout

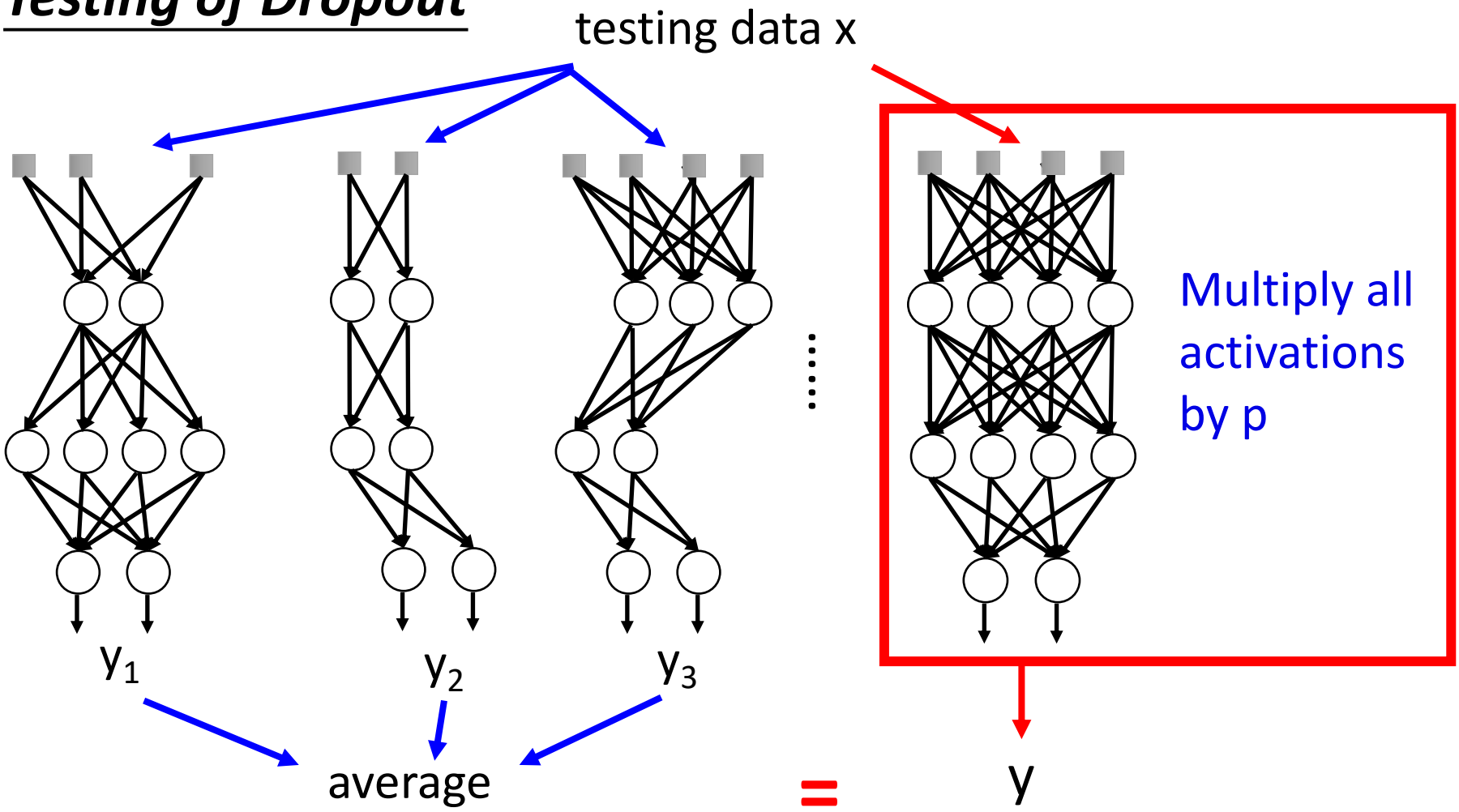
M neurons

2^M possible networks

- Using one mini-batch to train one network
- Some parameters in the network are shared

Source: Lee Hung-yi

Testing of Dropout



Source: Lee Hung-yi

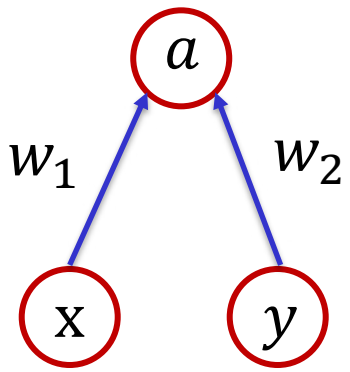
Regularization: dropout

At test time: drop nothing and multiply by the probability p.

Intuitions:

Consider a single neuron with dropout prob. at (1-p).

At training time we have:



$$E(a) = p^2 * (w_1 x + w_2 y) +$$
$$+ p(1-p) * (0x + w_2 y)$$
$$+ p(1-p) * (w_1 x + 0y)$$
$$+ (1-p)^2 * (0x + 0y) =$$
$$= p * (w_1 x + w_2 y)$$

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in forward pass

scale at test time

Source: Stanford CS231

Dropout Summary

More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

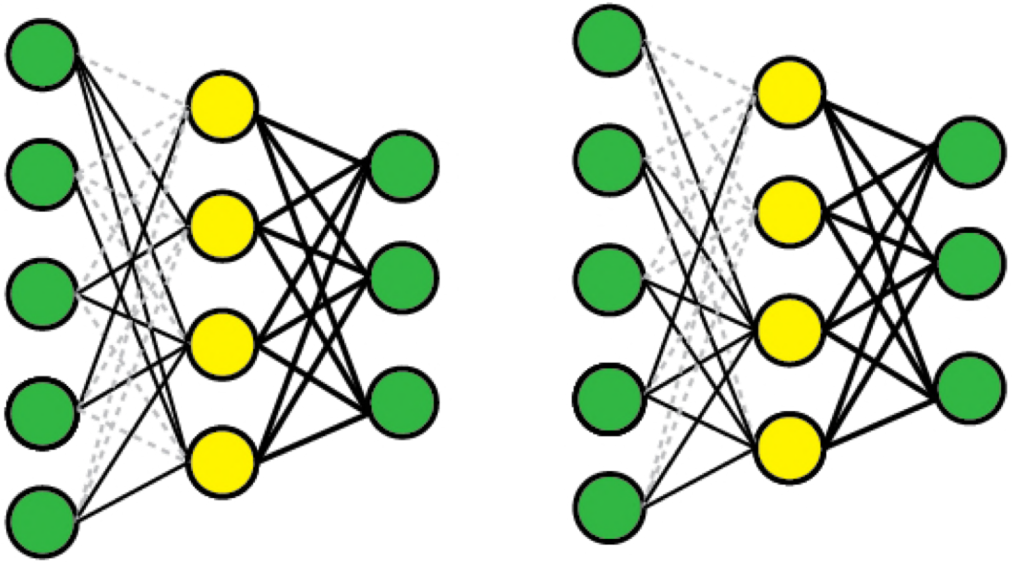
divide by p

test time is unchanged!

Source: Stanford CS231

Regularization: drop connect

- Drop-Connect works similarly to dropout, except that we disable individual weights (i.e., set them to zero), instead of nodes.
- Drop-Connect is a generalization of Drop-out because it produces more possible models, as there are almost always more connections than units.

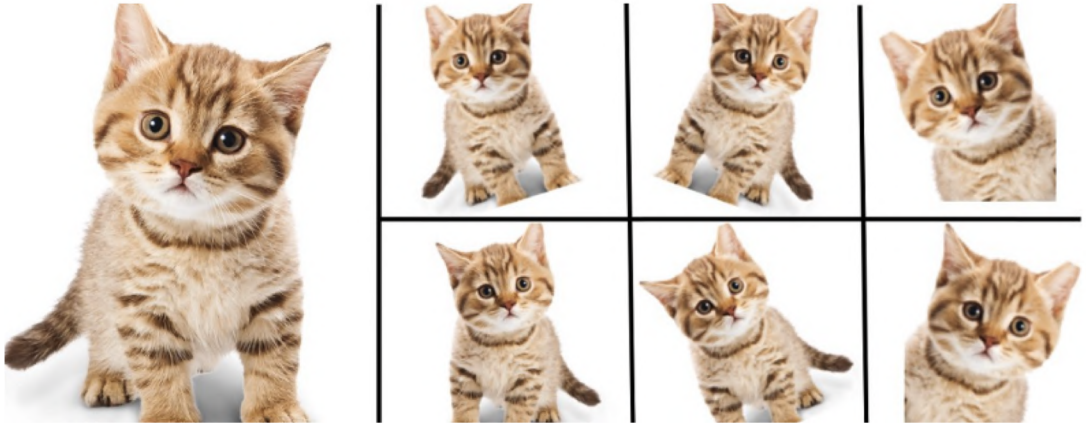


Regularization: Data augmentation

Intuition: If you have few examples – generate new examples from the existing ones.

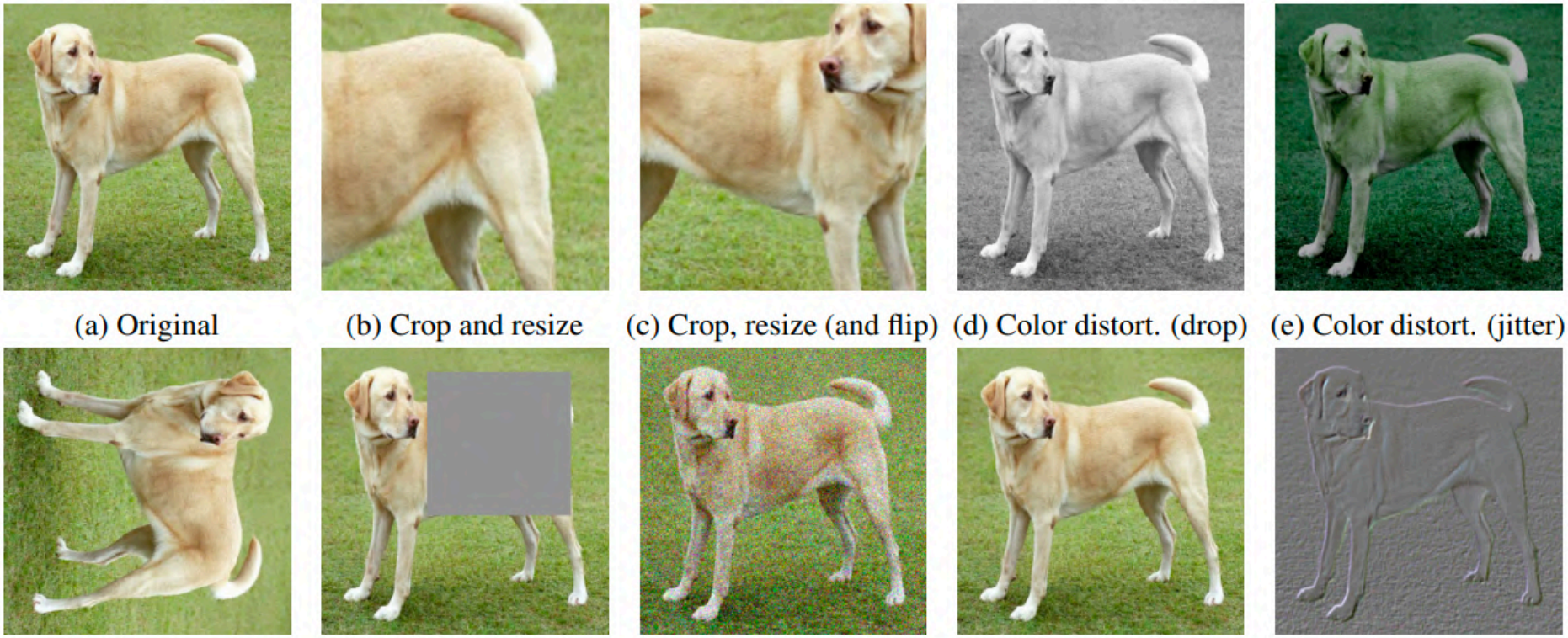
- Introduce transformations not adequately sampled in the training data
- Geometric: flipping, rotation, shearing, multiple crops
- Photometric: Color transformations, add noise, etc.

Rotations



Source: Stanford CS231

Regularization: Data augmentation

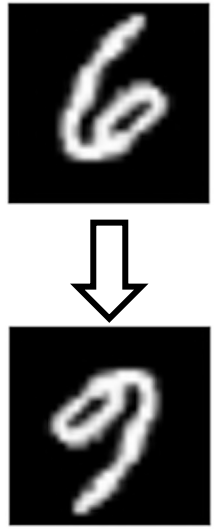


(a) Original (b) Crop and resize (c) Crop, resize (and flip) (d) Color distort. (drop) (e) Color distort. (jitter)

(f) Rotate $\{90°, 180°, 270°\}$ (g) Cutout (h) Gaussian noise (i) Gaussian blur (j) Sobel filtering

Regularization: Data augmentation

- Use many transformations. Go crazy!
- But avoid introducing obvious artifacts and bias
- Avoid changing semantics



Source: Stanford CS231

Regularization: Mixup

- **Training**: Train on random blends of images
- **Testing**: Use original images

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

CNN

Target label:
0.4 cat
0.6 dog

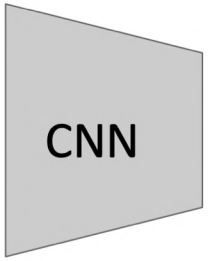
Source: Justin Johnson

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

Regularization: CutMix

- **Training**: Train on random blends of images
- **Testing**: Use original images

Replace random crops of one image with another: e.g. 60% of pixels from cat, 40% from dog



Target label:

0.6 cat

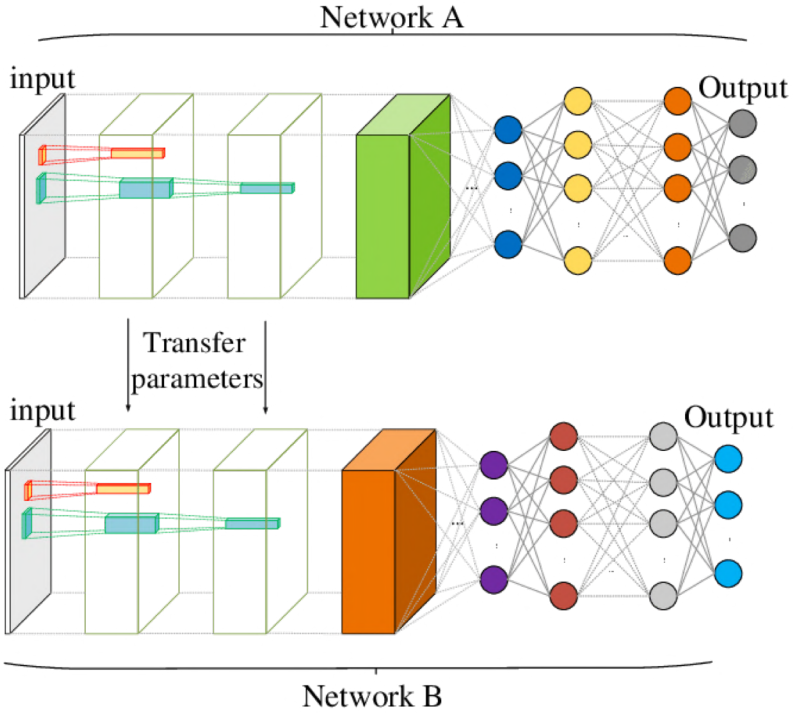
0.4 dog

Source: Justin Johnson

Yun et al, “CutMix: Regularization Strategies to Train Strong Classifiers with Localizable Features”, ICCV 2019

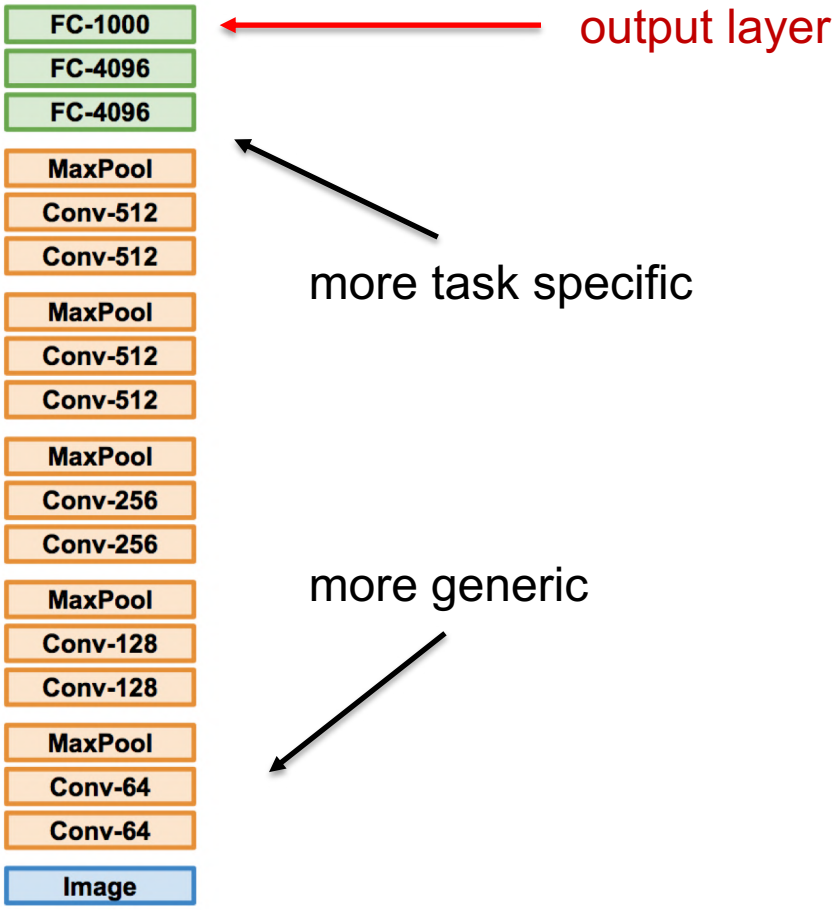
Transfer Learning

- Transfer learning is a method where a pre-trained network on one task is re-used on a second task
- Useful where we lack annotated data
- Two possibilities:
  - Use the pre-trained model as a starting point for the second-task
  - Freeze the lower part of the network and retrain the last part



Transfer Learning

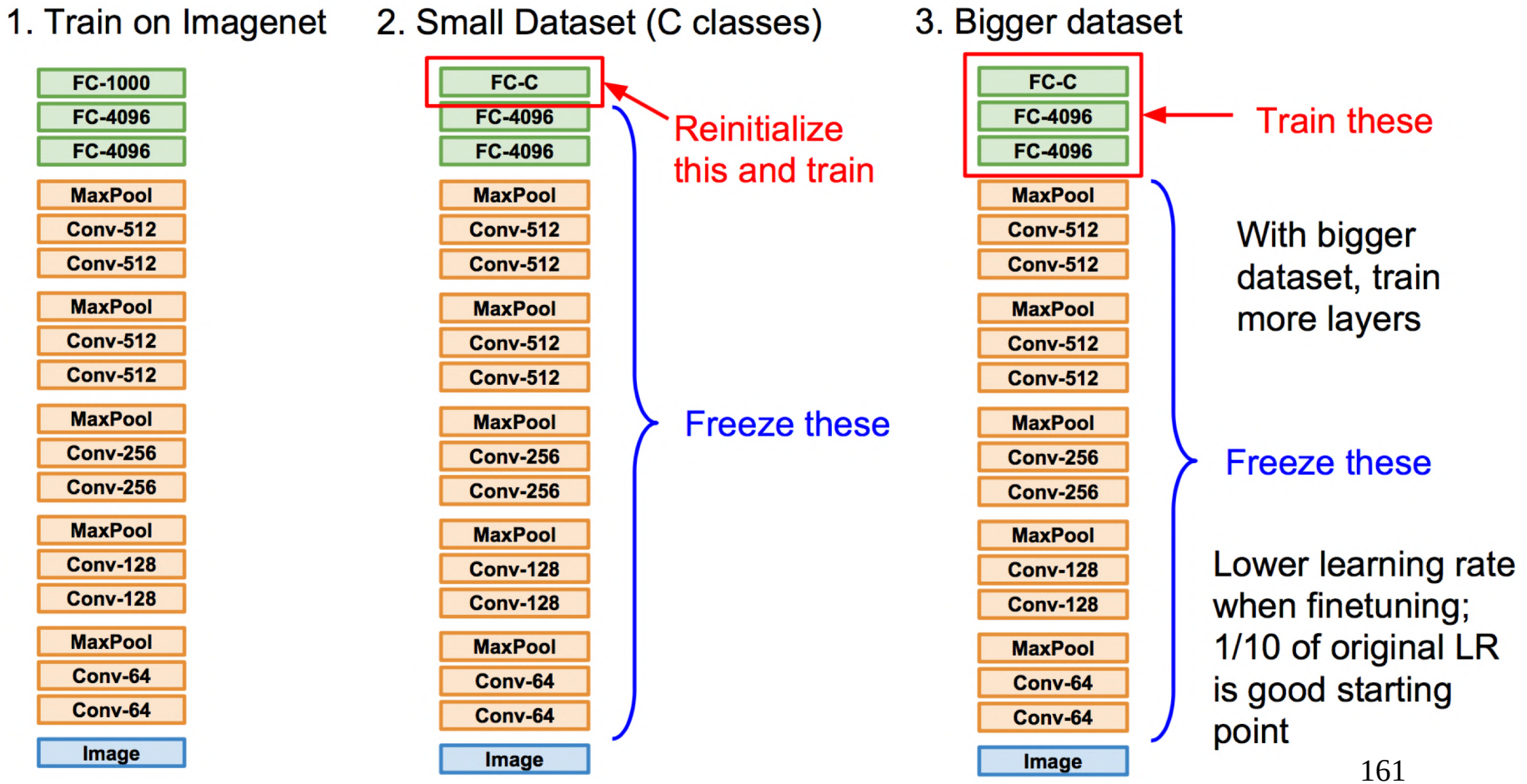
- Important observation: for similar datasets, features in lower layers are similar although the task is different.



Source: Stanford CS231

Transfer Learning

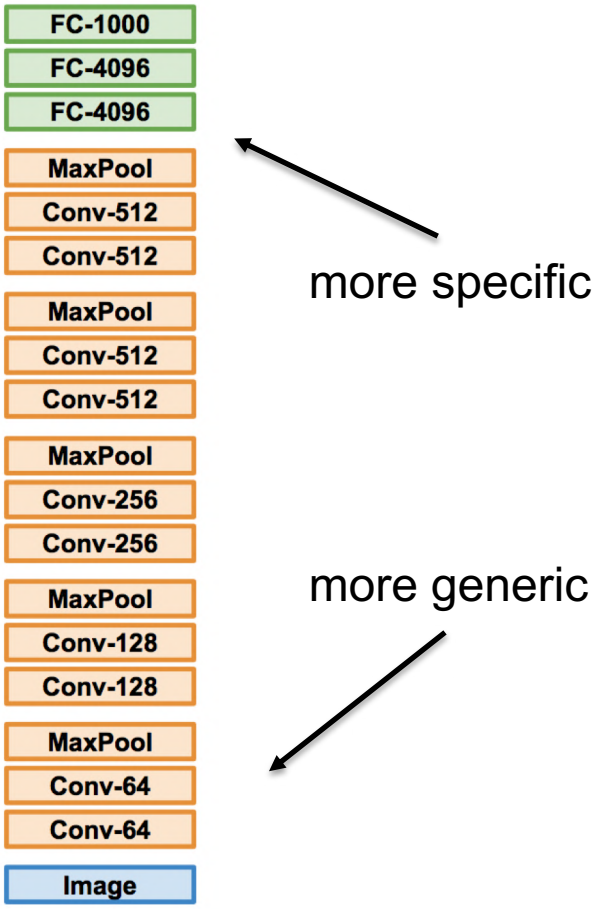
How many layers should we retrain?



Source: Stanford CS231

Transfer Learning

- How many layers should we retrain?

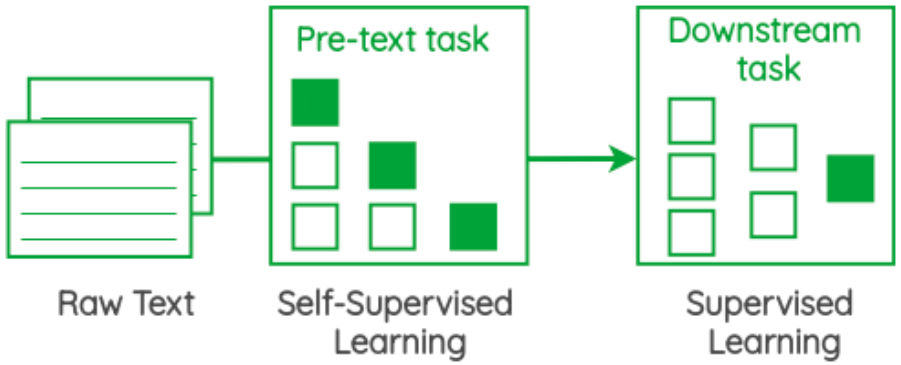


	Very similar database	**Very different database**
Very little data	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
Quite a lot of data	Fine-tune a few layers	Finetune a larger number of layers

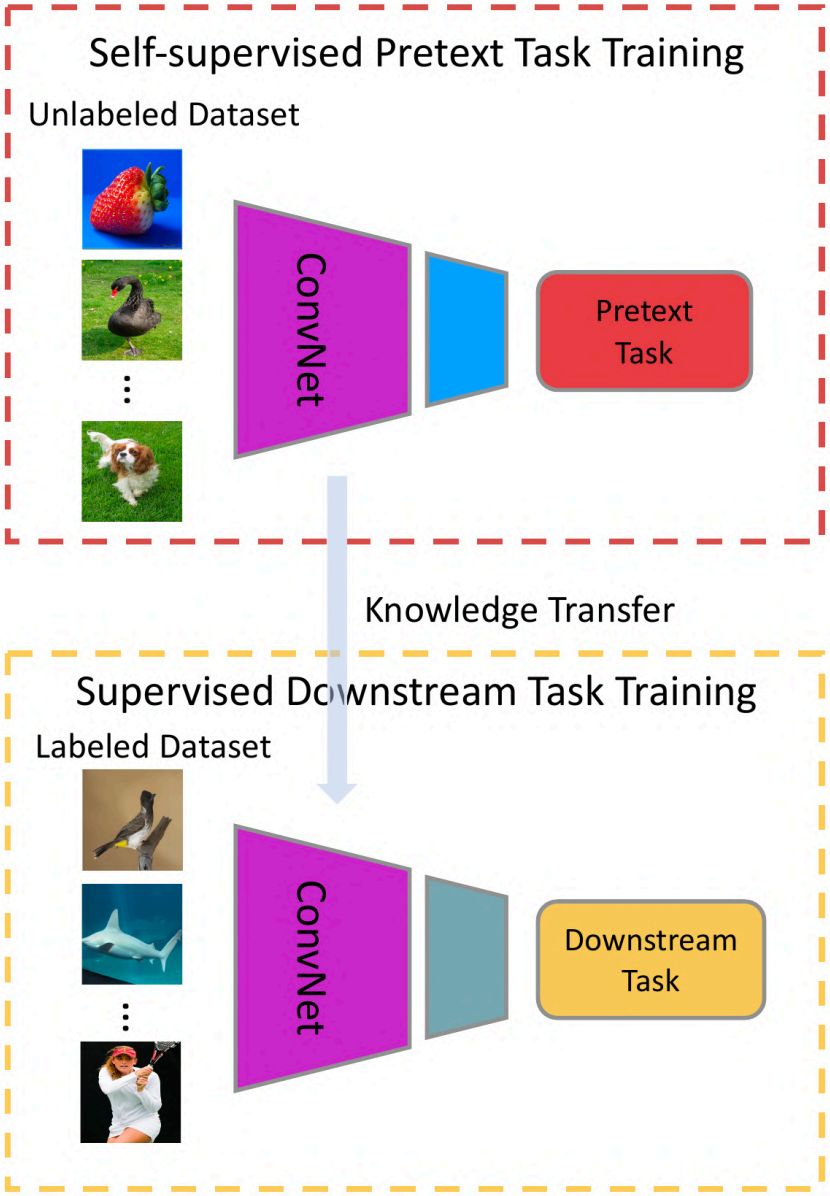
Source: Stanford CS231

Regularization: self-supervised learning

- What if we lack annotated data and we don't have a pre-trained network (for transfer learning)?
- Solution: make one!
- **Plan**: Design a task in such a way that we can generate virtually unlimited labels from our existing images and use that to learn a useful representation.
- **Self-supervised learning:** replaces the human annotation block by exploiting some property of data to set up a pseudo-supervised task called a **pretext task**.

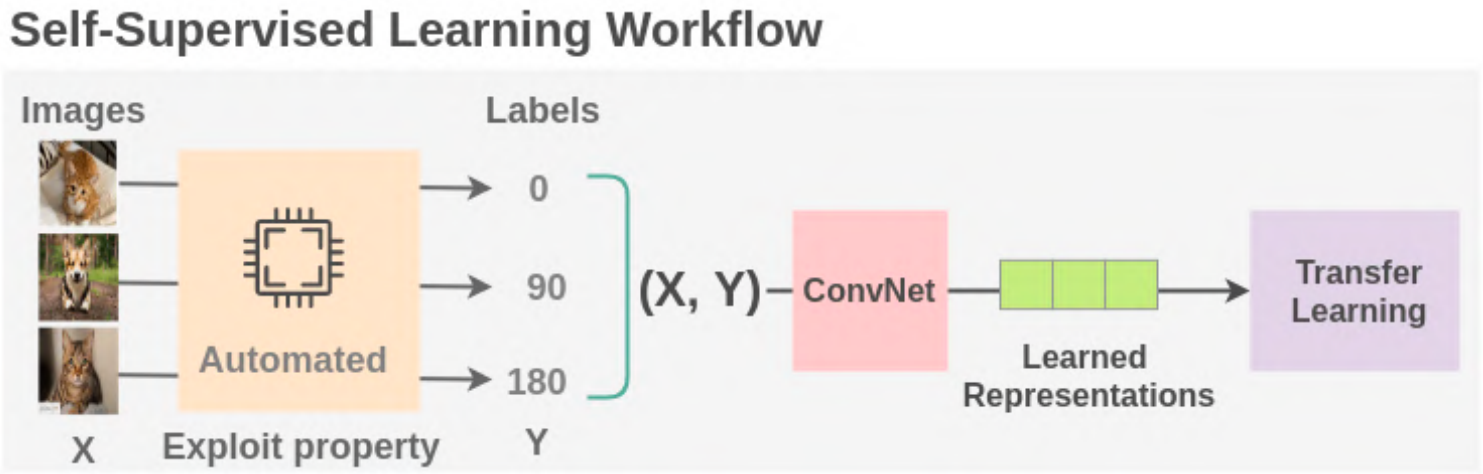


Regularization: self-supervised learning



Regularization: self-supervised learning

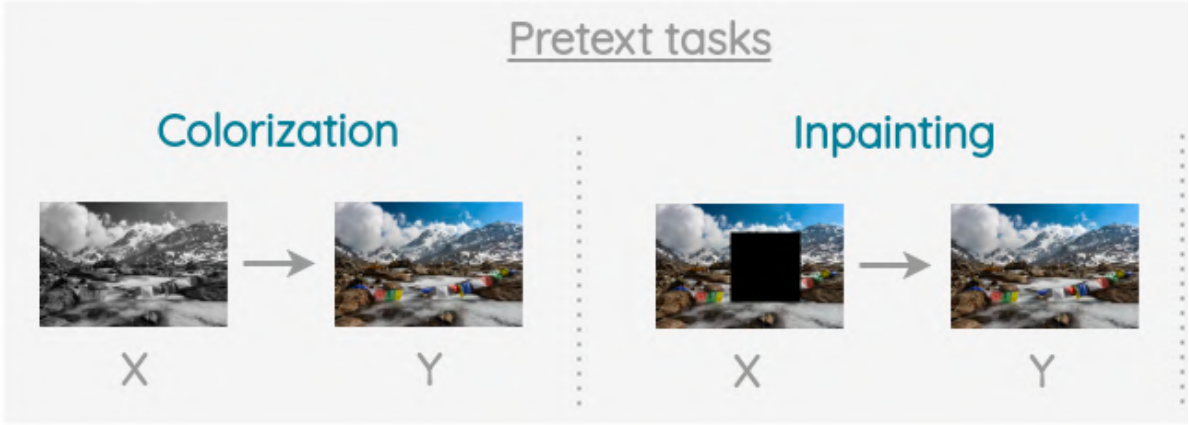
- **Pretext task example**: Instead of labeling images as cat/dog, we can rotate the images by 0/90/180/270 degrees and train a model to predict rotation.
- We can generate virtually unlimited training data from millions of images.
- Once we learn representations from these images, we can use transfer learning and train a supervised model like cats/dogs classification with very few examples

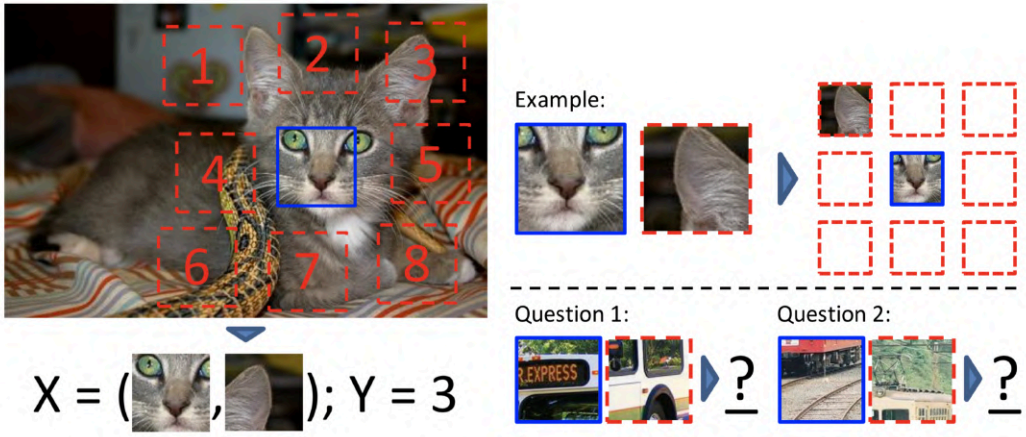


Regularization: self-supervised learning

Examples of pretext tasks:

- Predict the color of grayscale images
- Image inpainting
- Relative position
- Rotations
- Etc…

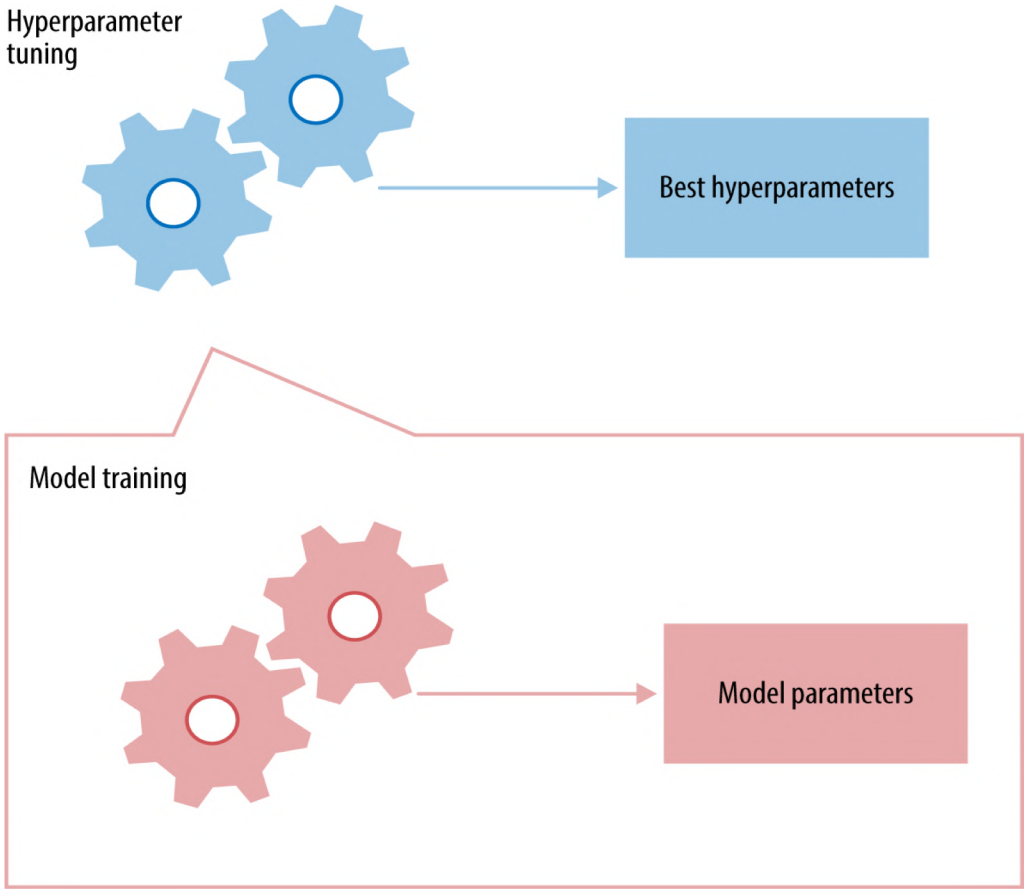




Regularization: Summary

- **General Idea**: do not train a network to achieve too low training error and high generalization error
- **Methods**: Disrupt your network and constrain the flexibility of the model
- Approaches:
  - Early stopping
  - Weight decay (L_2,L_1)
  - Dropout / drop-connect
  - Data augmentation
- **Advanced methods**:
  - Transfer Learning
  - Self-supervised learning

Hyper-parameter Tuning



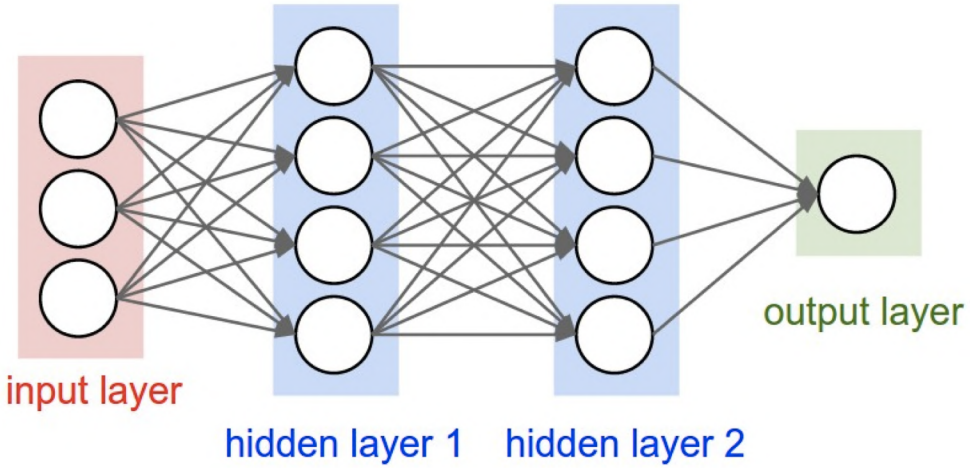
Supervised learning pipeline

1. **Collect data and labels**
2. **Specify model:** select model type, loss function, *hyperparameters*
3. **Train model:** find the *parameters* of the model that minimize the empirical loss on the training data

- What is the right methodology for step 2?
- What are the hyperparameters

Hyperparameters in multi-layer networks

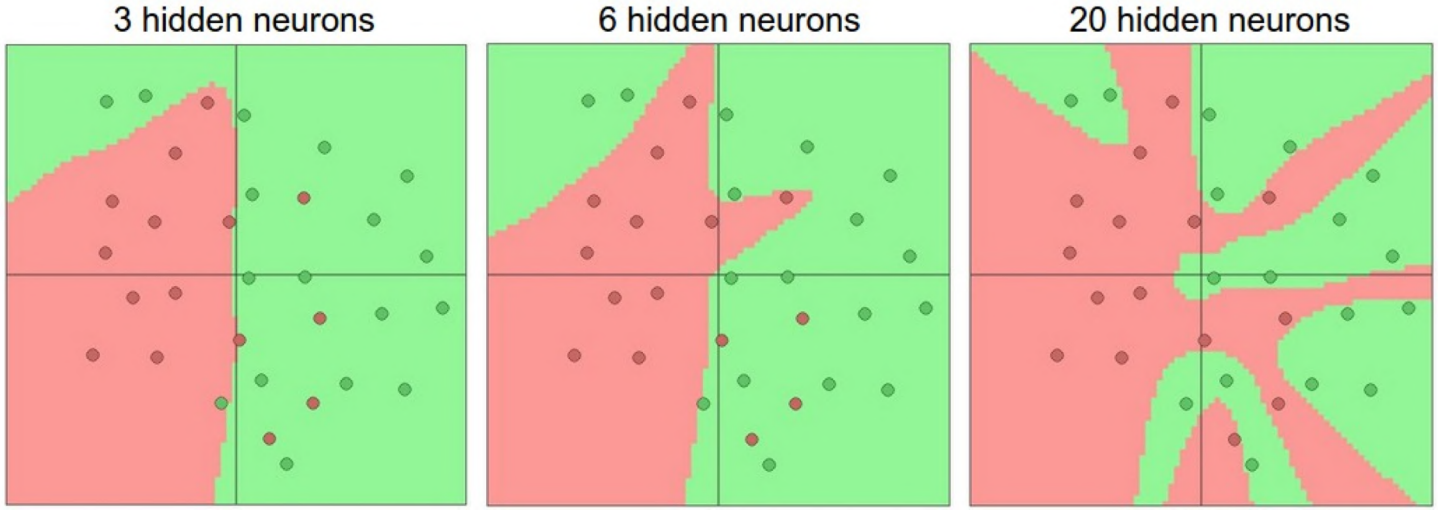
- Number of layers
- Number of units per layer



Source: Stanford 231n

Hyperparameters in multi-layer networks

- Number of layers
- Number of units per layer

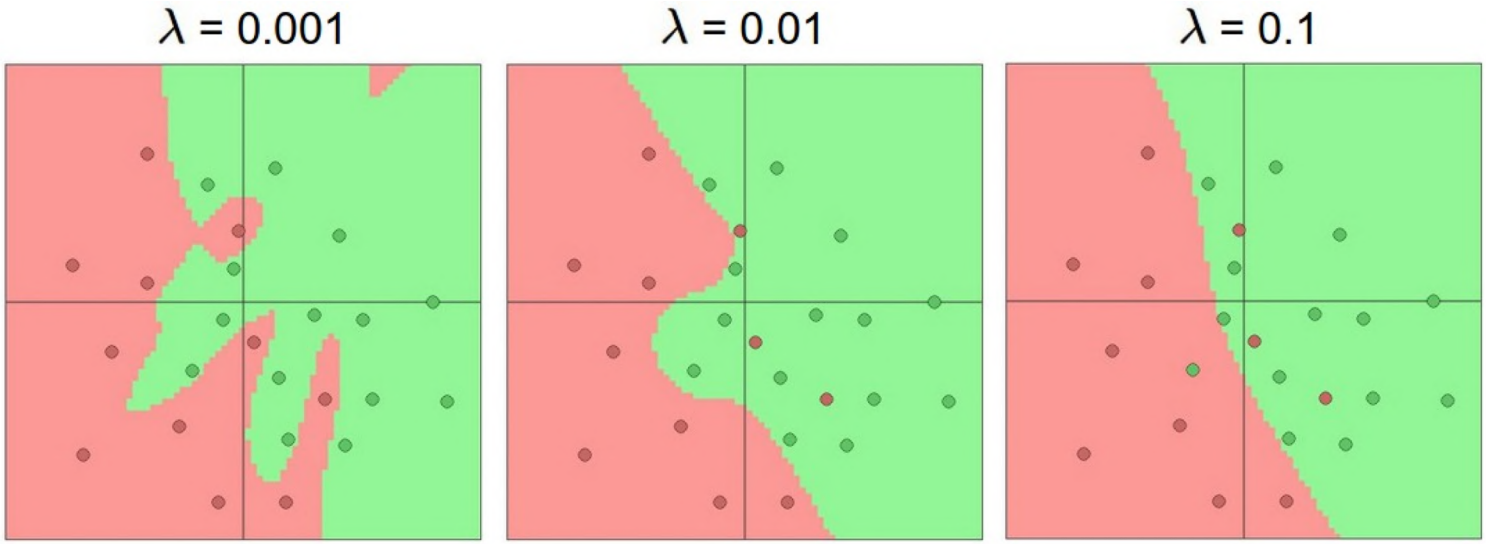


Number of hidden units in a two-layer network

Source: Stanford 231n

Hyperparameters in multi-layer networks

- Number of layers
- Number of units per layer
- Regularization L_1, L_2, λ



Source: Stanford 231n

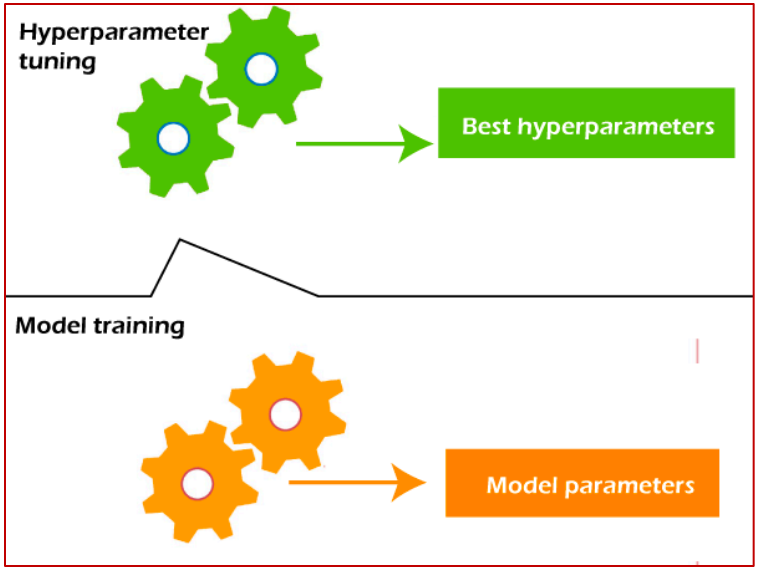
Hyperparameters in multi-layer networks

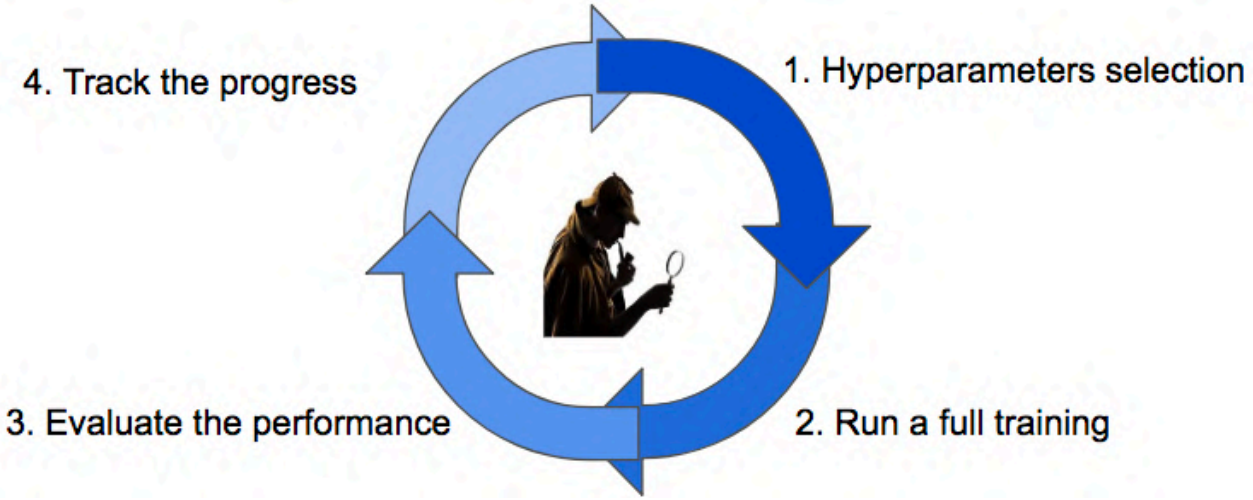
- Number of layers
- Number of units per layer
- Regularization L_1, L_2, λ
- SGD: learning rate schedule, number of epochs, minibatch size, dropout *p*, momentum strength, etc.

- We can think of our hyperparameter choices as defining the "complexity" of the model and controlling its generalization ability

Source: Stanford 231n

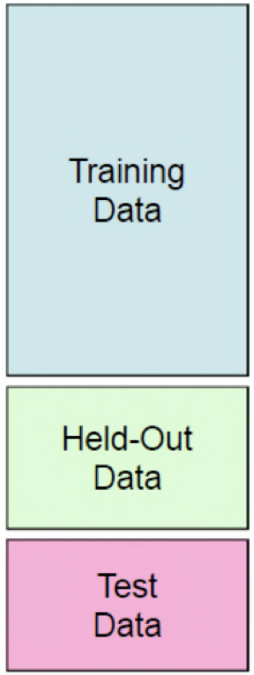
Hyperparameter tuning

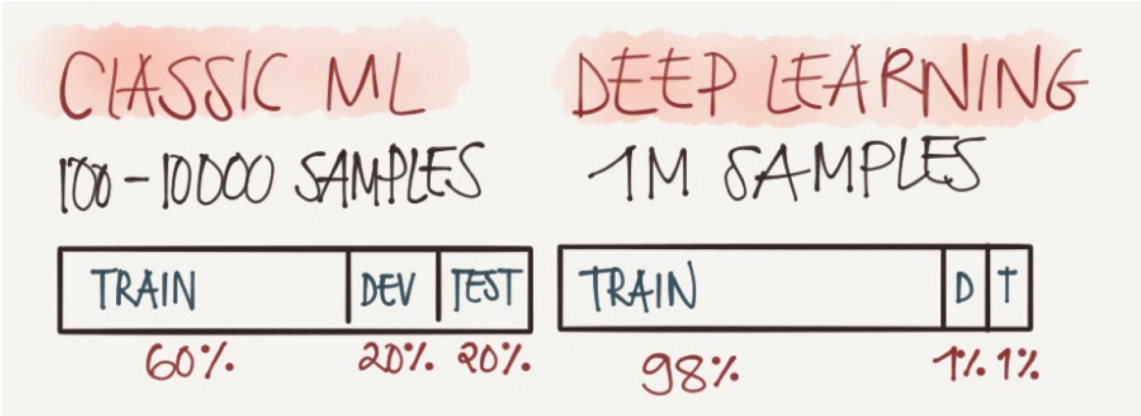




Hyperparameter Tuning

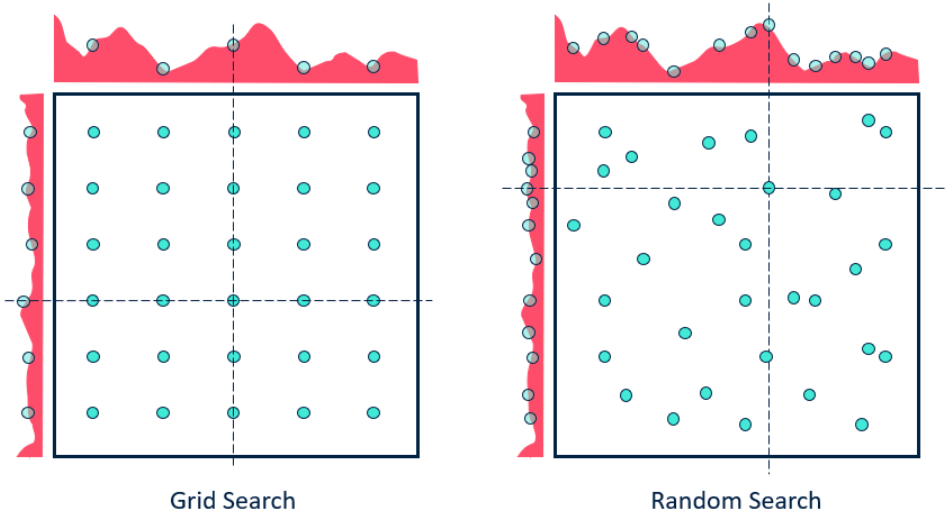
- **Training set** - used to learn the network parameters (weights, biases)
- **Validation set (Dev)** – used to perform model selection and hyperparameters tuning
- **Test set** – used to assess the fully trained model
- For deep networks, Dev and Test sets should be a small fraction of the entire set, as most of the data is needed for training





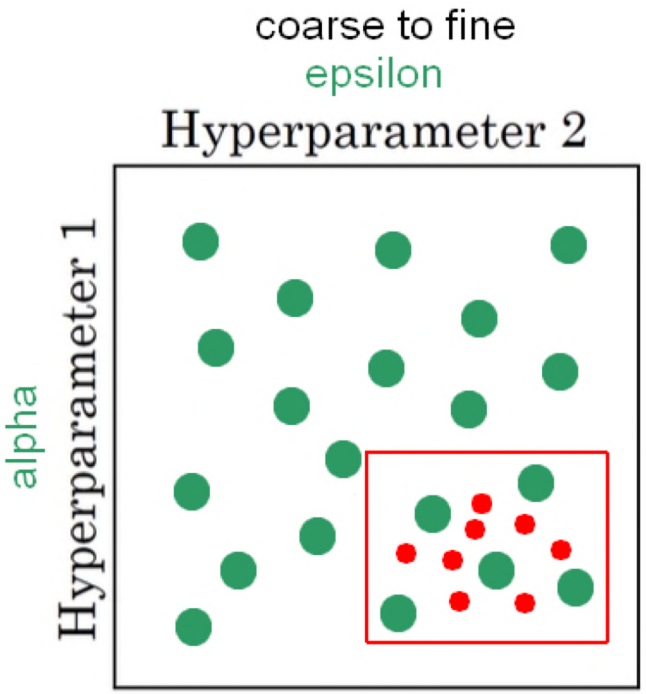
Hyperparameter search

- **Grid search** is a traditional technique for implementing hyperparameters. It brute forces all combinations.
- Grid search suffers from the curse of dimensionality if data is of high dimensional space.
- **Random search** randomly samples the search space. It explores more samples at each axis. Especially useful if one axis is more important than the others.



Coarse to fine search

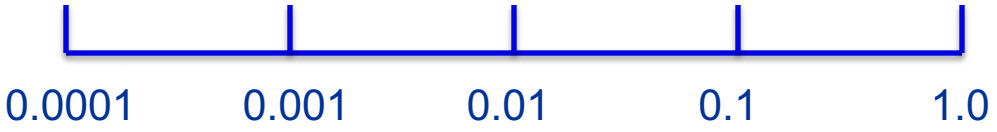
- After trying out random values, we will determine where is a region the performance is good.
- Then we refine our hyperparameter search in that region. This process is called **coarse to fine**.



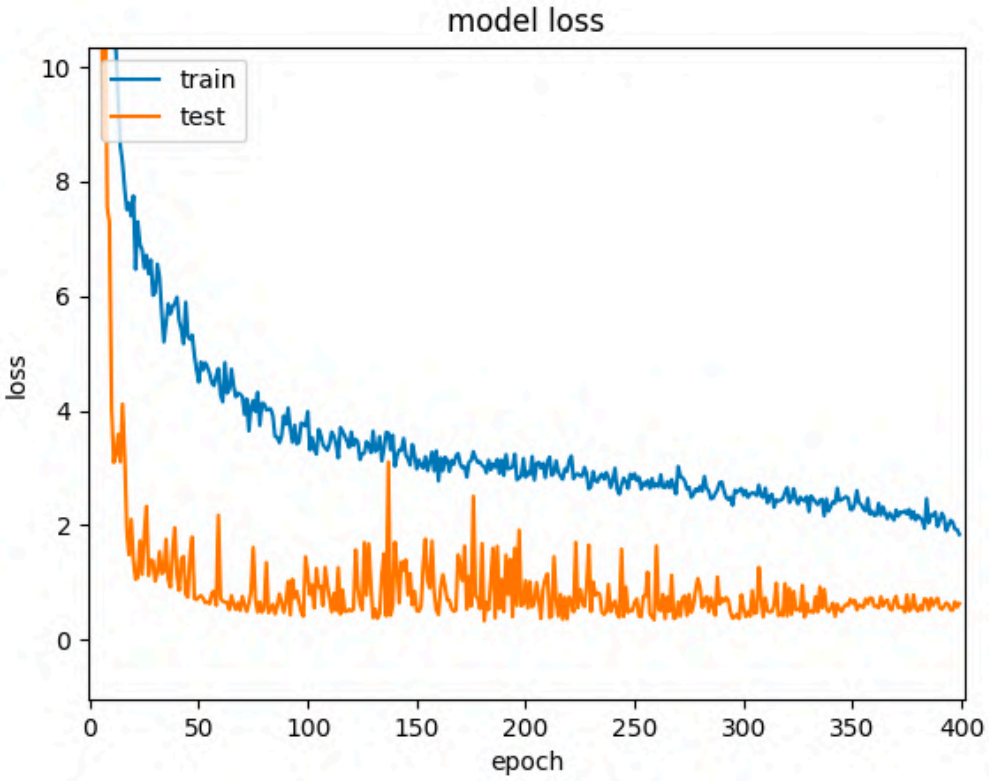
Log Scale Search

- Some hyper-parameters, such as, *learning rate* and *regularization strength*, have multiplicative effects on the training dynamics.
- Adding 0.01 to the learning rate has huge effects on the dynamics if the learning rate is 0.001, but nearly no effect if the learning rate is 10.
- For such hyperparameters search on log scale:

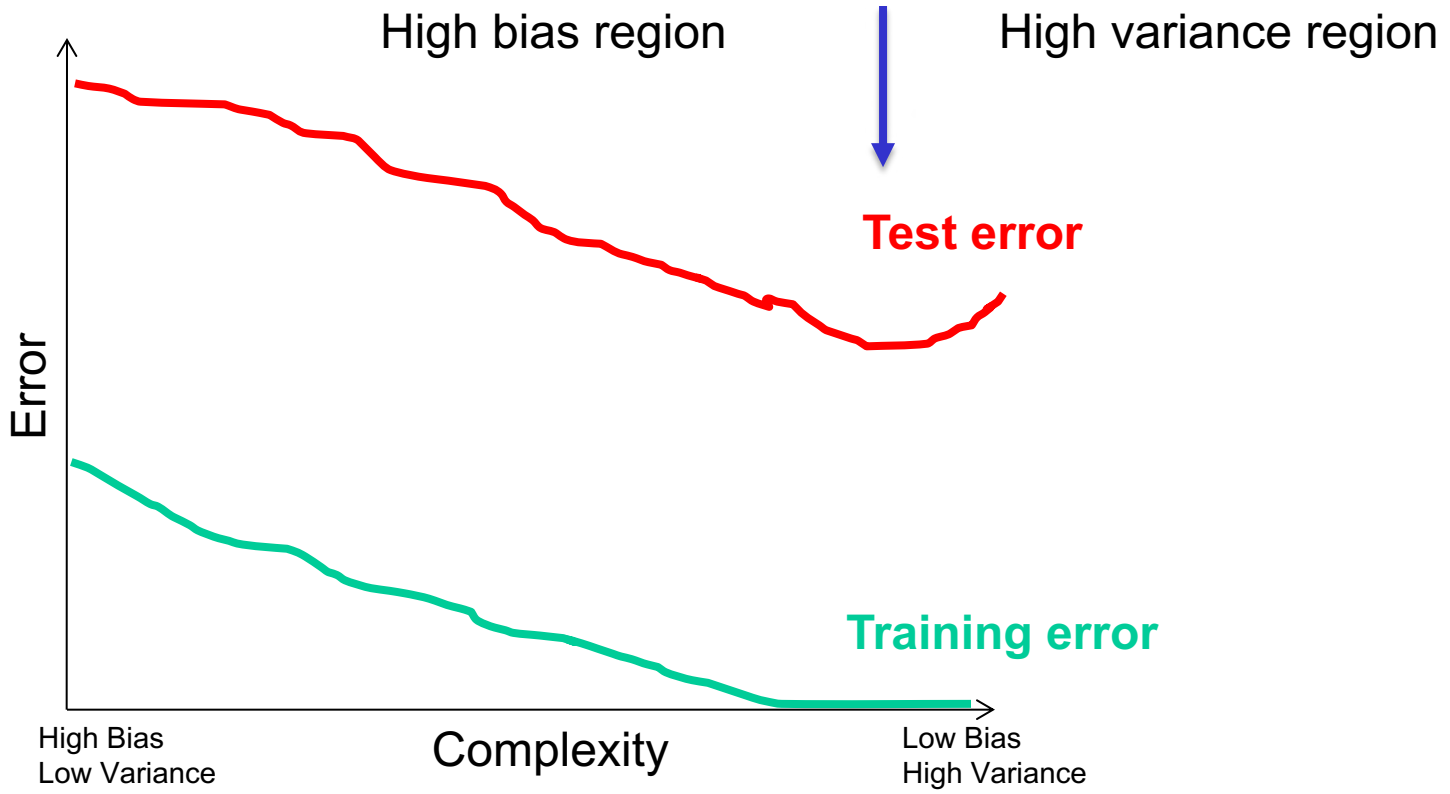
$$lealningRate = 10^{\text{uniform}(-4,0)}$$



Convergence Evaluation:

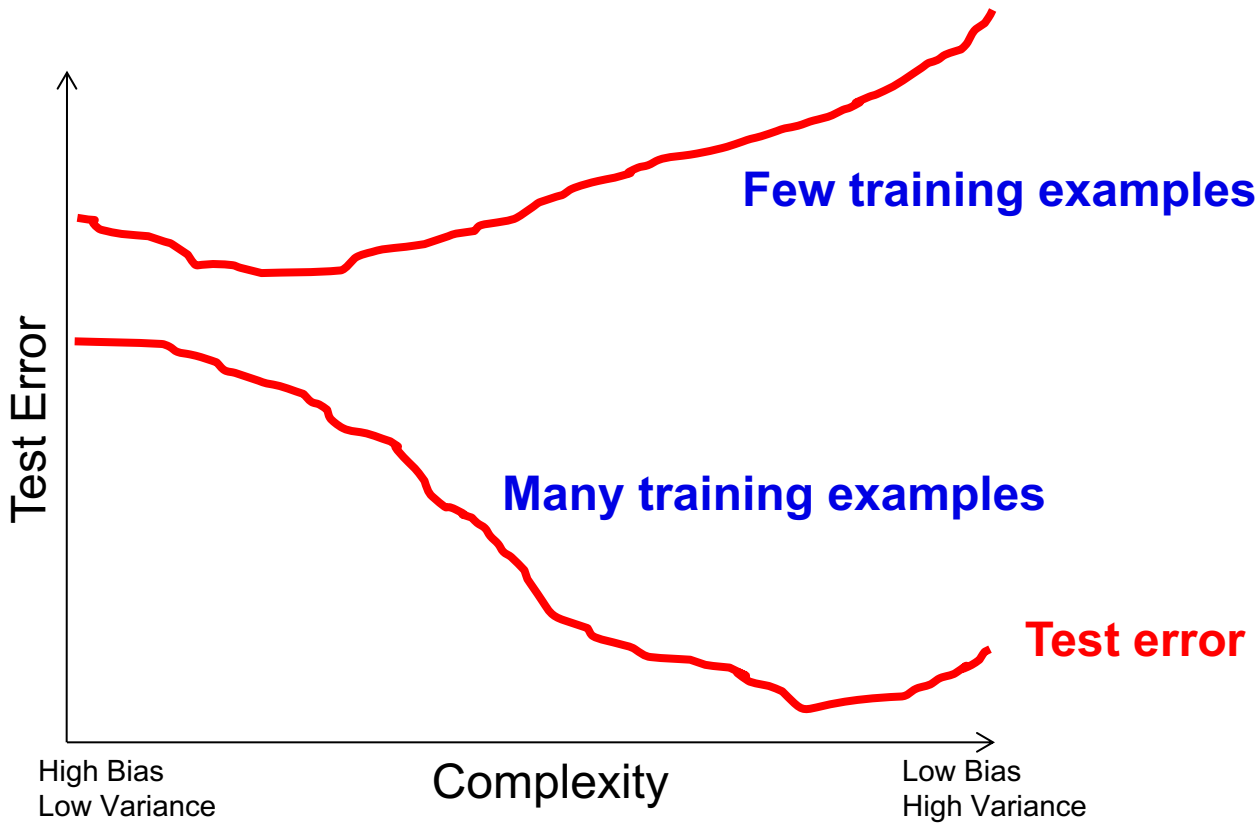


Training vs test errors



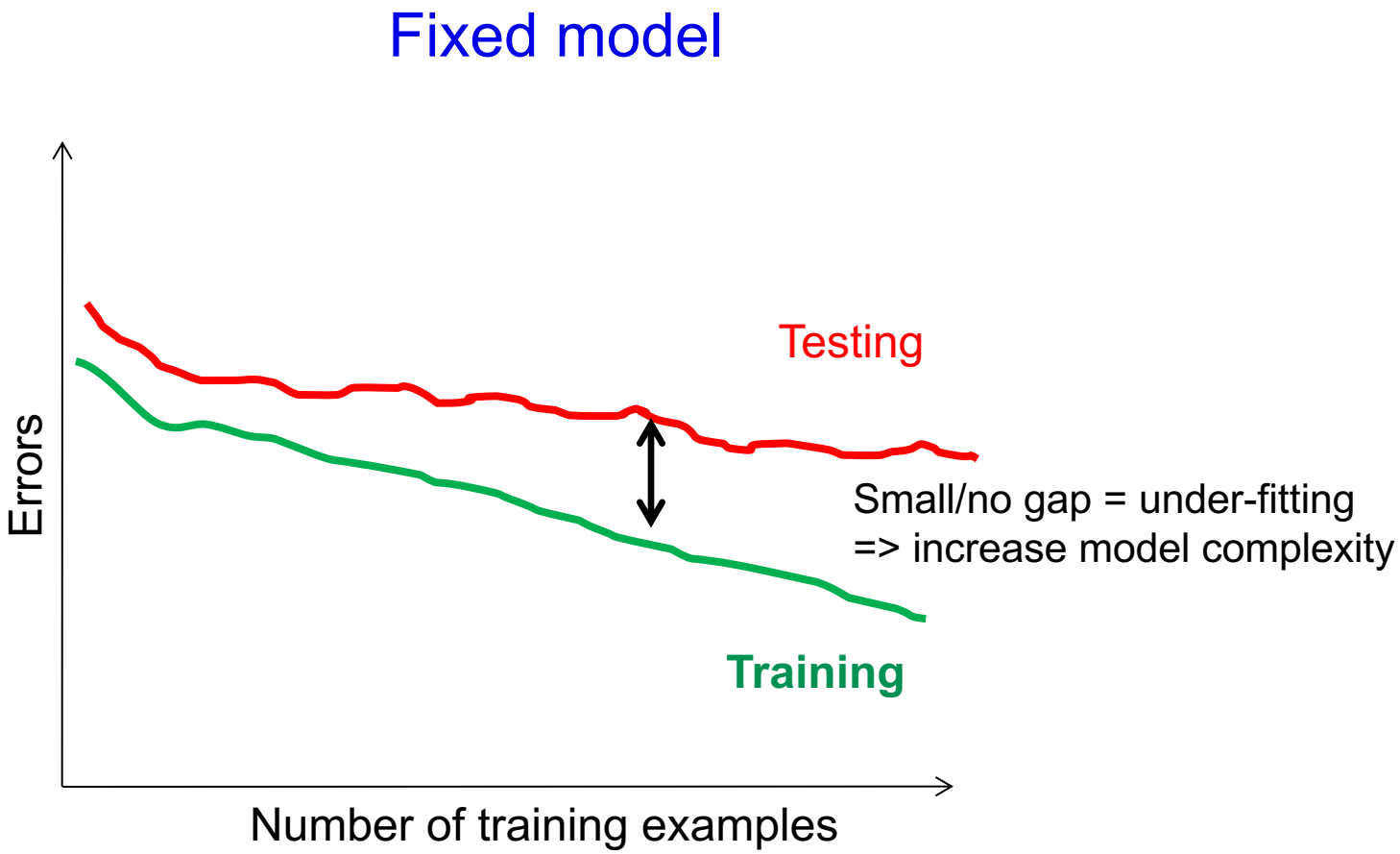
Source: D. Hoiem

Effect of training set size



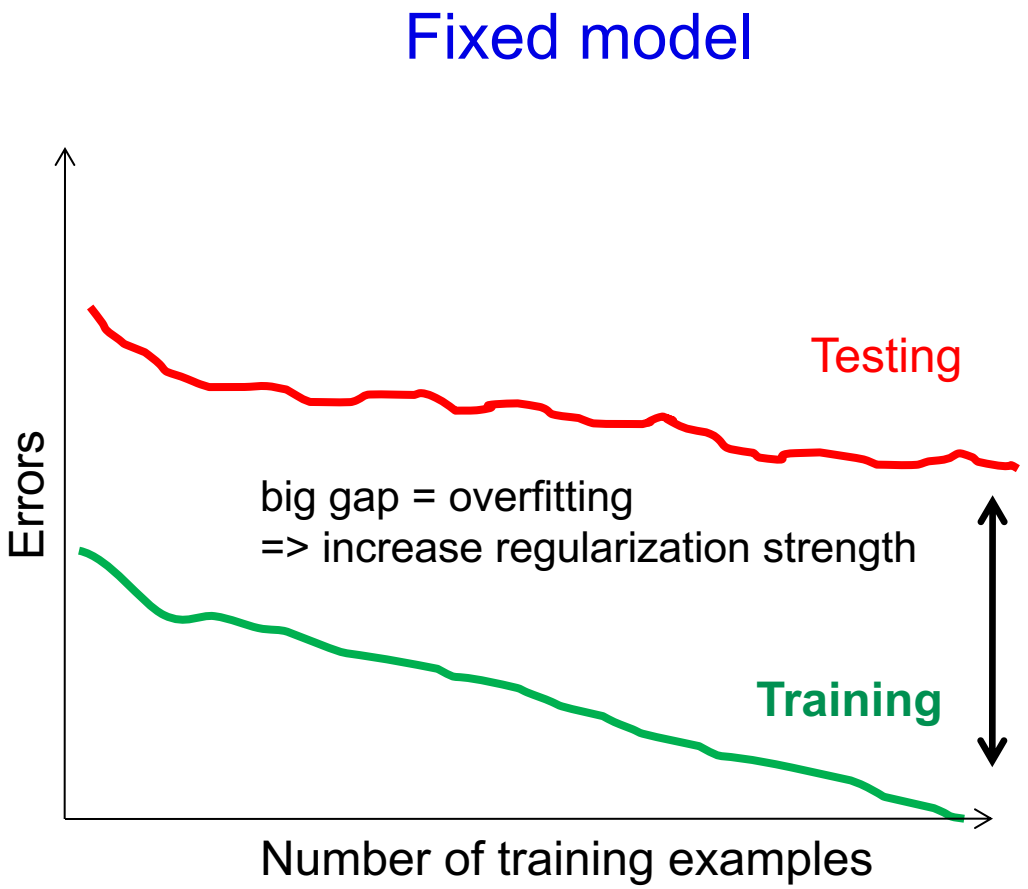
Source: D. Hoiem

Effect of training set size



Source: D. Hoiem

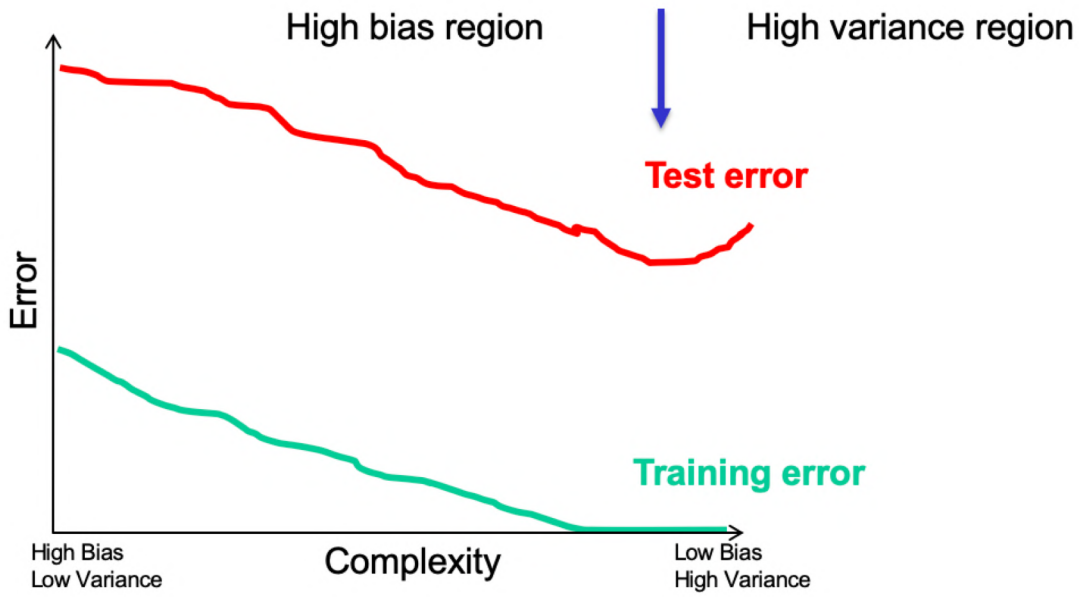
Effect of training set size



Source: D. Hoiem

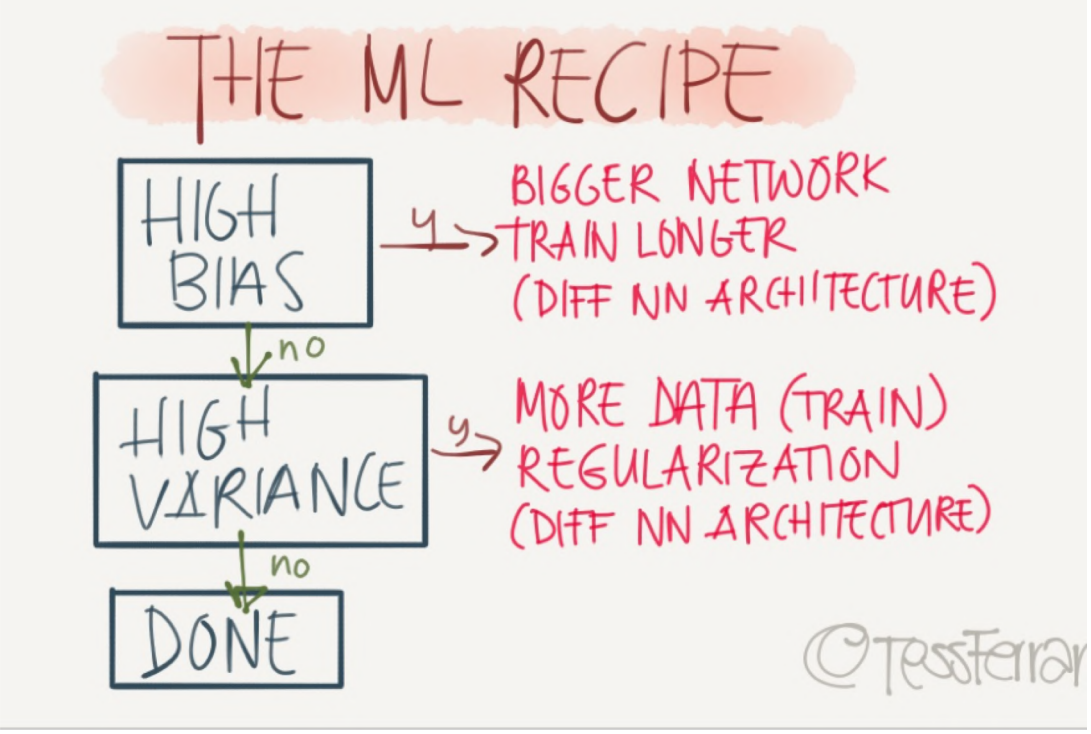
The ML Recipe

	Error			
Train	1%	15%	15%	0.5%
Test	11%	10%	30%	1%

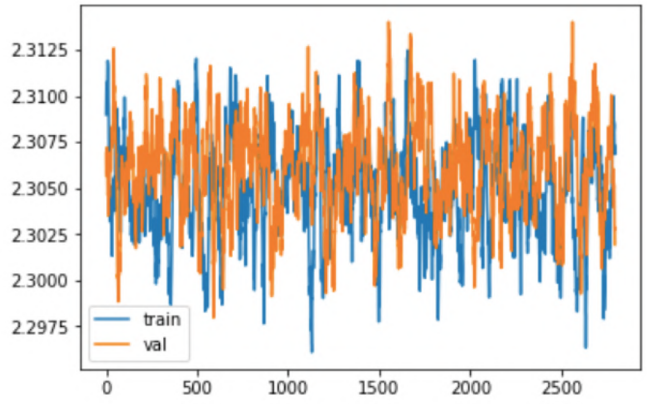


The ML Recipe

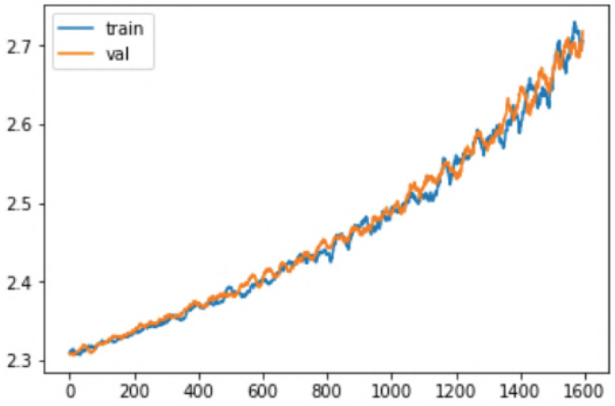
	Error			
Train	1%	15%	15%	0.5%
Test	11%	10%	30%	1%
	High Variance	High Bias	High B&V	Low B&V



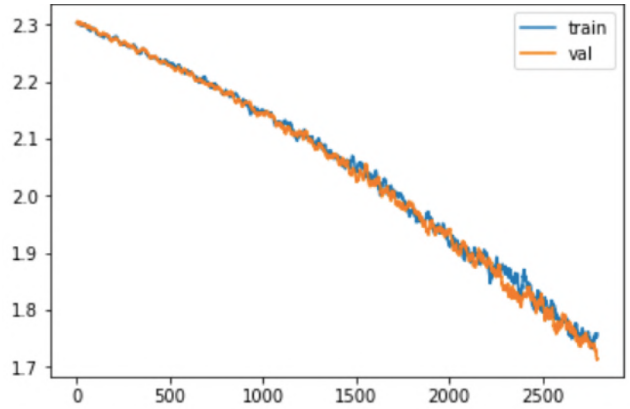
Debugging learning curve



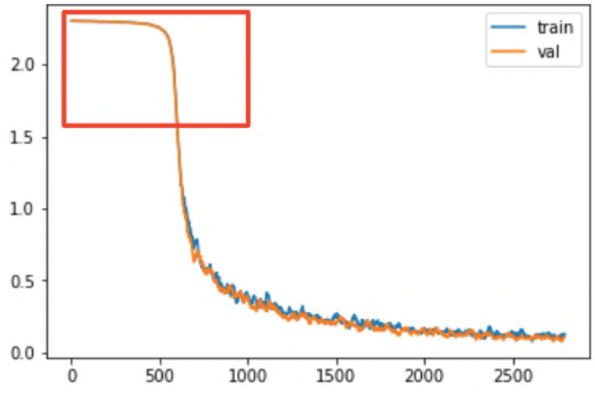
Not training
Bug in update calculation?



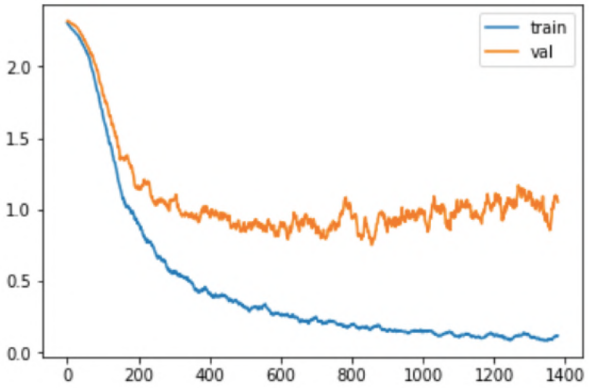
Error increasing
Bug in update calculation?



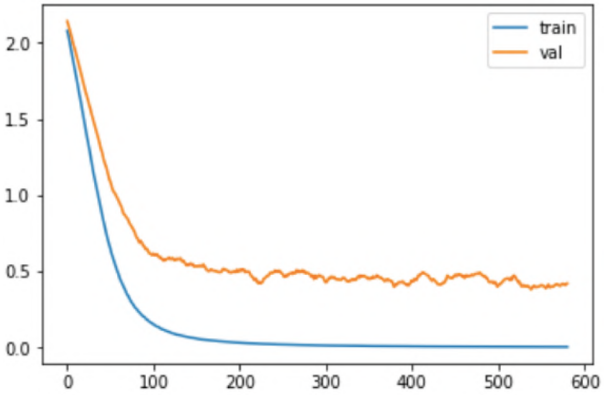
Error decreasing
Not converged yet



Slow start
Suboptimal initialization?



Overfitting



Looks good

Source: Illinois CS 498

Training Summary

- Training neural networks is still a black magic
- Process requires close “babysitting”
- For many techniques, the reasons why, when, and whether they work are in active dispute
- Read everything but don’t trust anything
- It all comes down to (principled) trial and error

Source: Illinois CS 498

THE END