Introduction into Deep Learning

Neural Networks and Multi-Layered Perceptron. Backpropagation. Tips and tricks for training NNs.

Fourth Machine Learning in High Energy Physics Summer School, MLHEP 2018, August 6--12

Alexey Artemov[1,2]

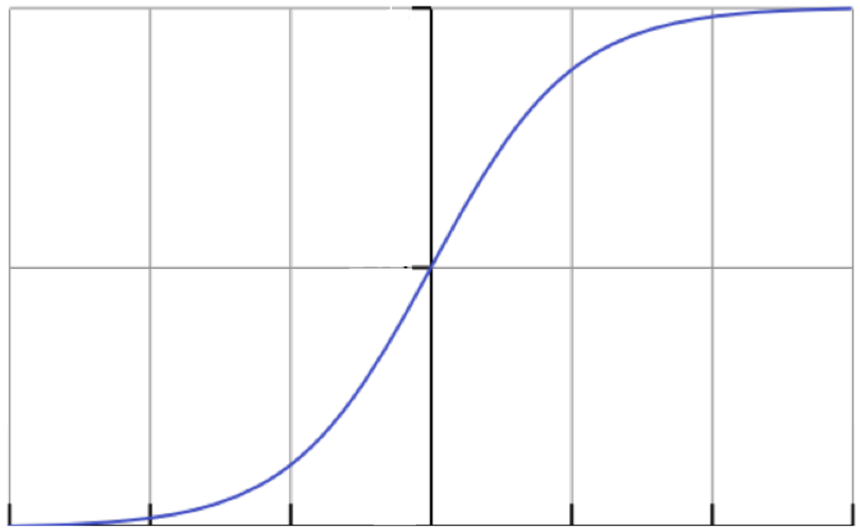
[1]Skoltech [2]National Research University Higher School of Economics

Lecture overview

- The basic principle of *deep learning*
- The one you will be applying to all problems hereinafter
- Absolutely essential for all future material
- Step-by-step example of training a neural network via backpropagation
  - You'll need the knowledge when using the advanced architectures

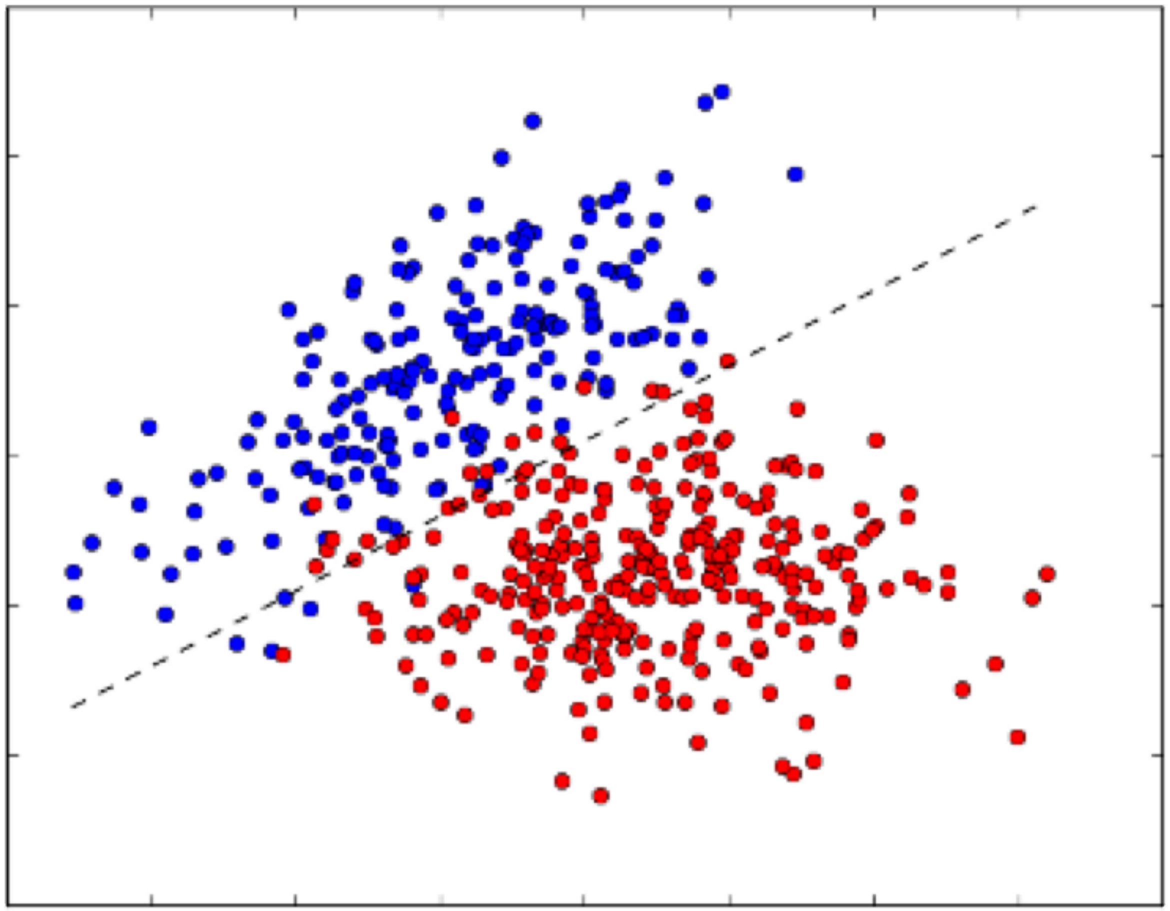
Principles of linear vs. nonlinear models

Recap: linear models

$W \cdot x + b$ →  → $P(y = +1|x)$

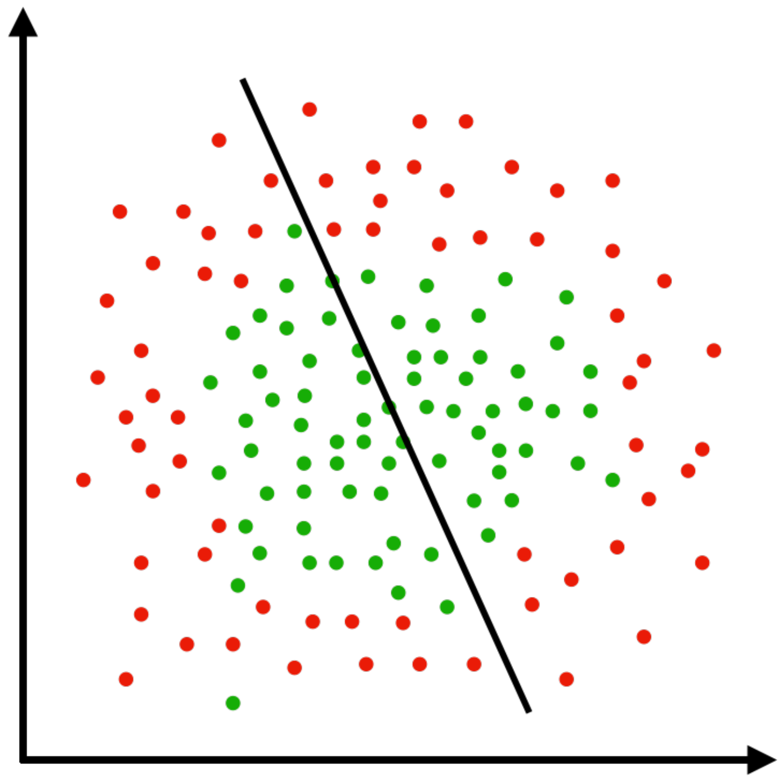
- x: features vector
- W, b: model slope and intercept

Linear dependency

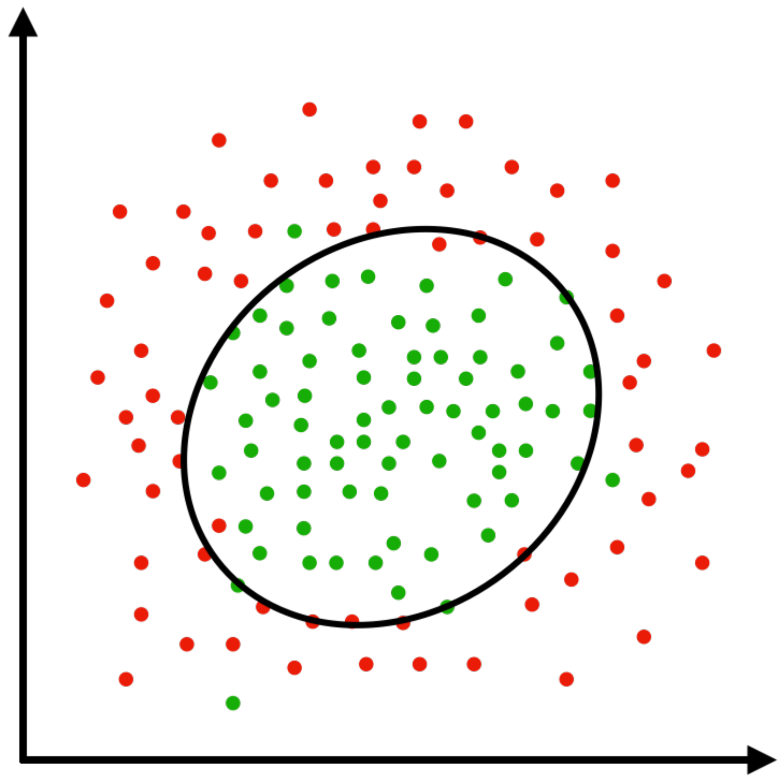


Nonlinear dependencies

What we have



What we want



Somewhat nonlinear dependencies

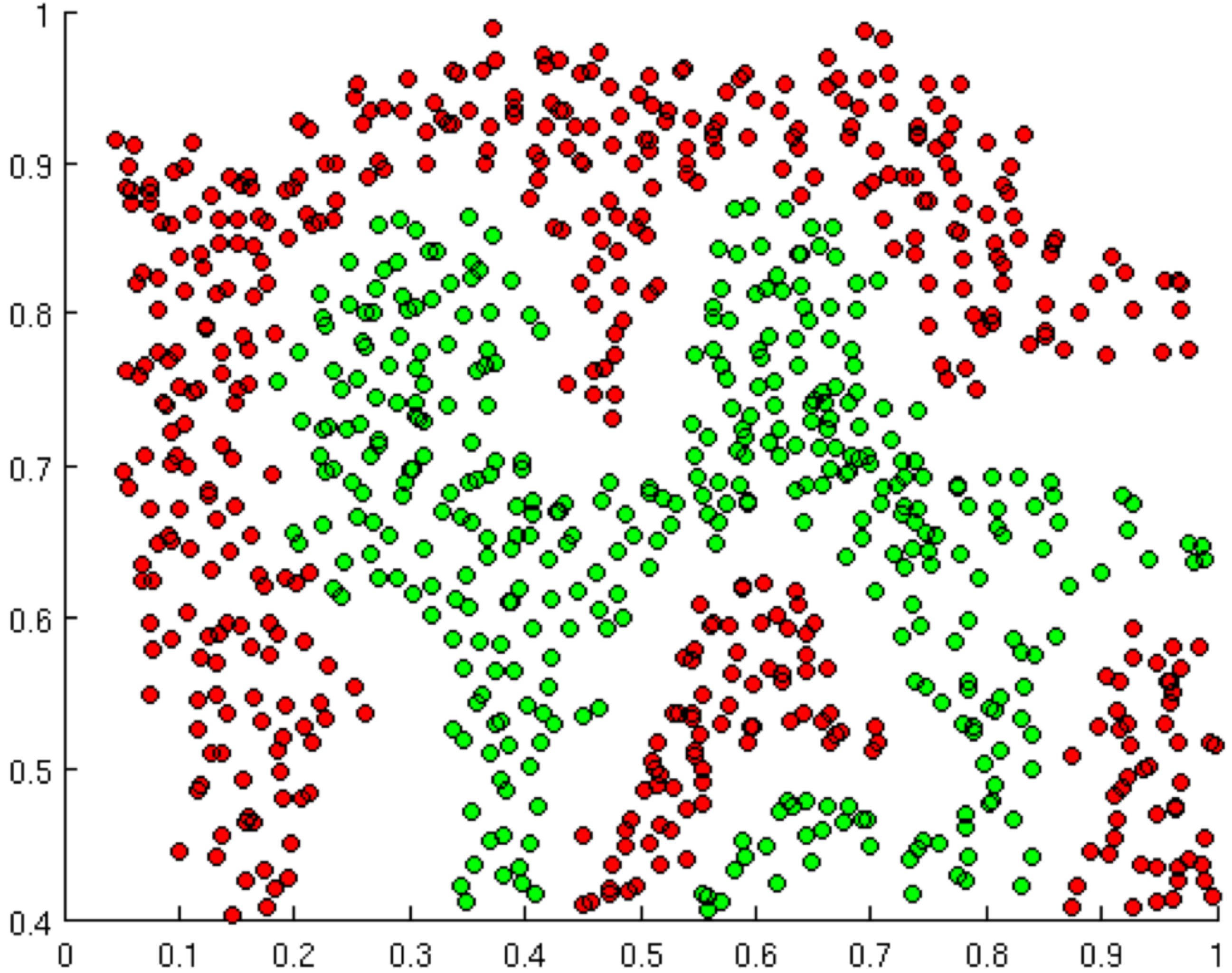
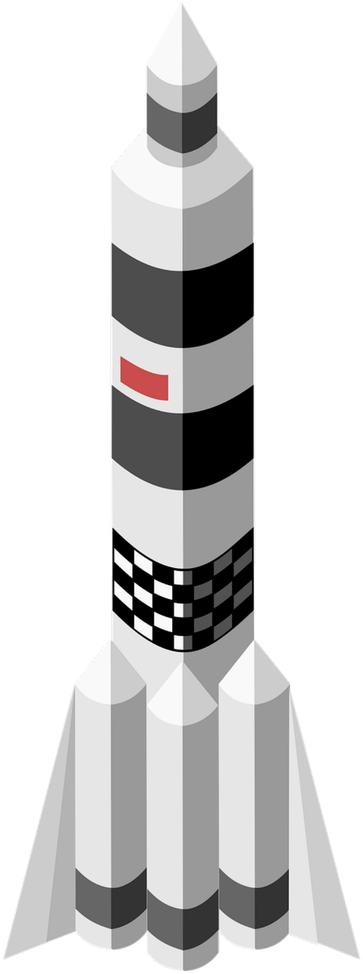


Image: Andrew Ng

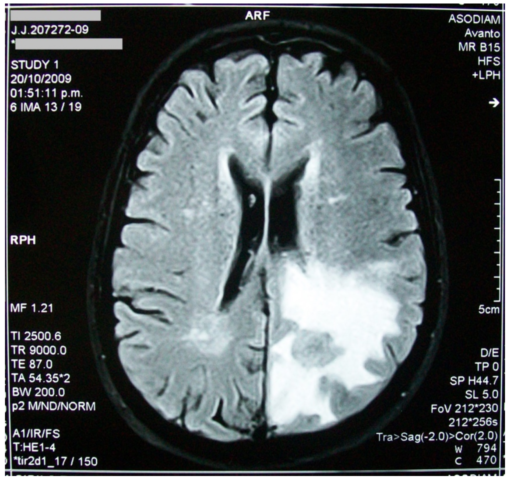
Extremely nonlinear dependencies

Most of the dependencies in this world!

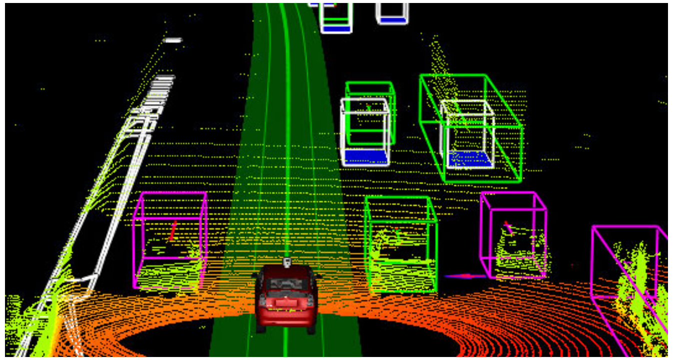


rocket

cat

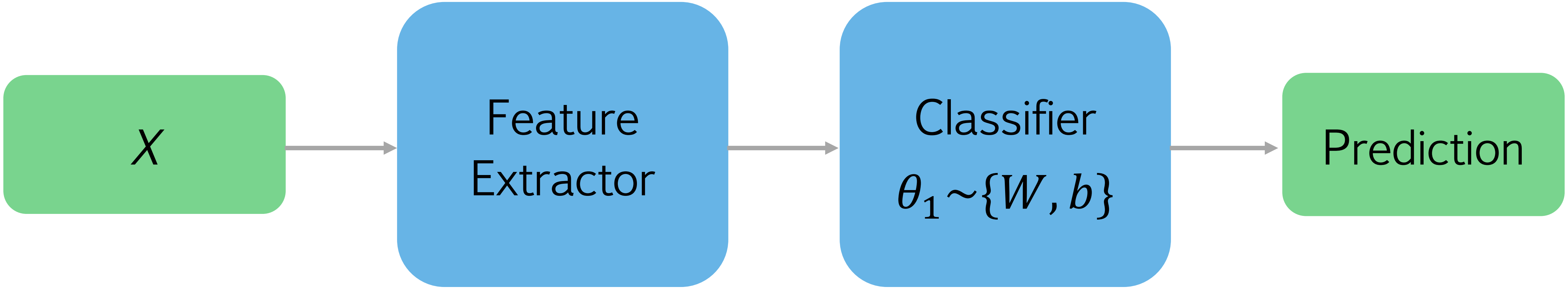


Brain tumors
as seen on MRI



Self-driving LiDAR

Extremely nonlinear dependencies



- Decouple feature extractor from the classifier
- Training and inference really can have multiple stages!

Feature *extraction*?

Data that is linearly non-separable in cartesian are l

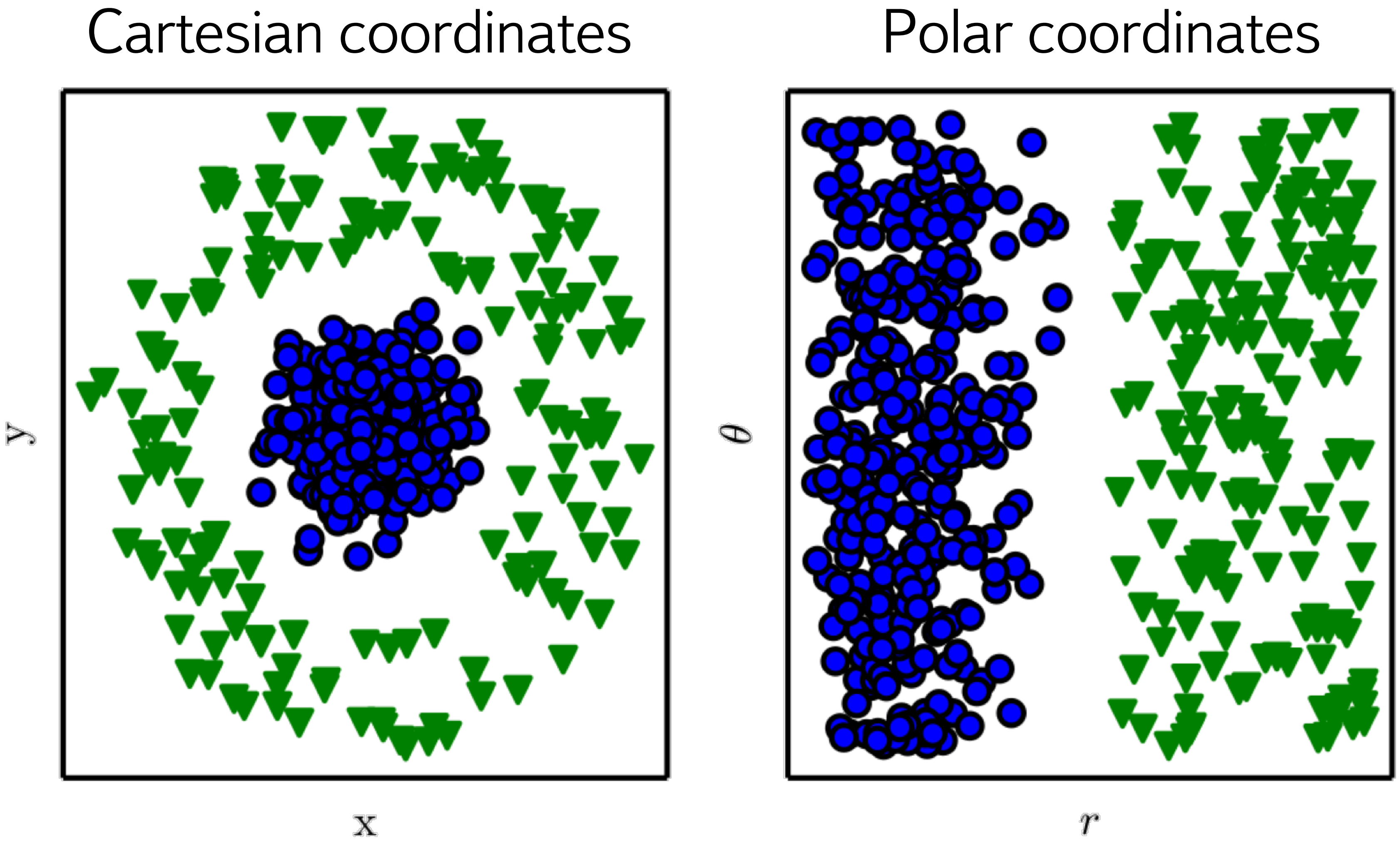


Image: Ian Goodfellow et al.

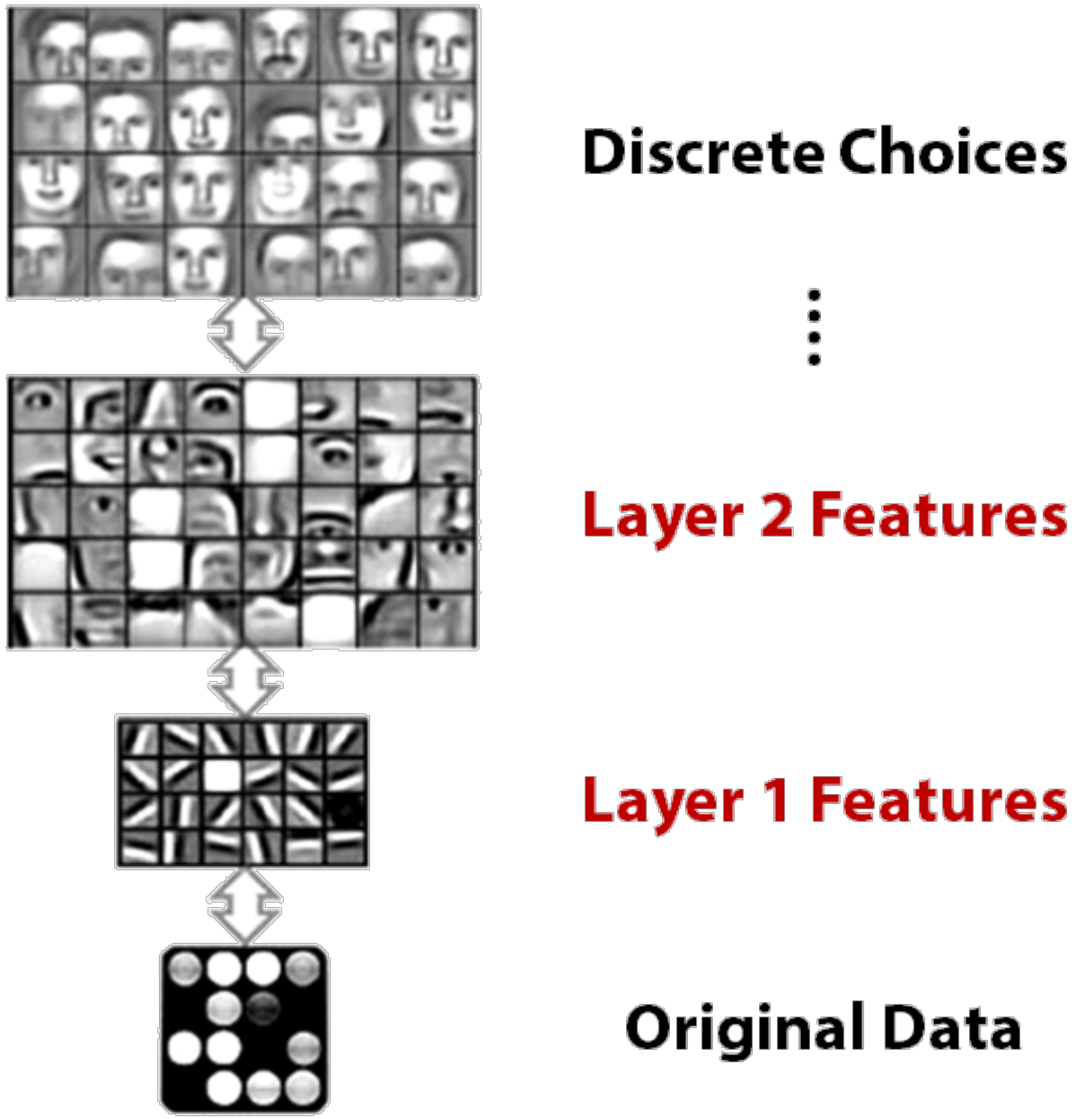
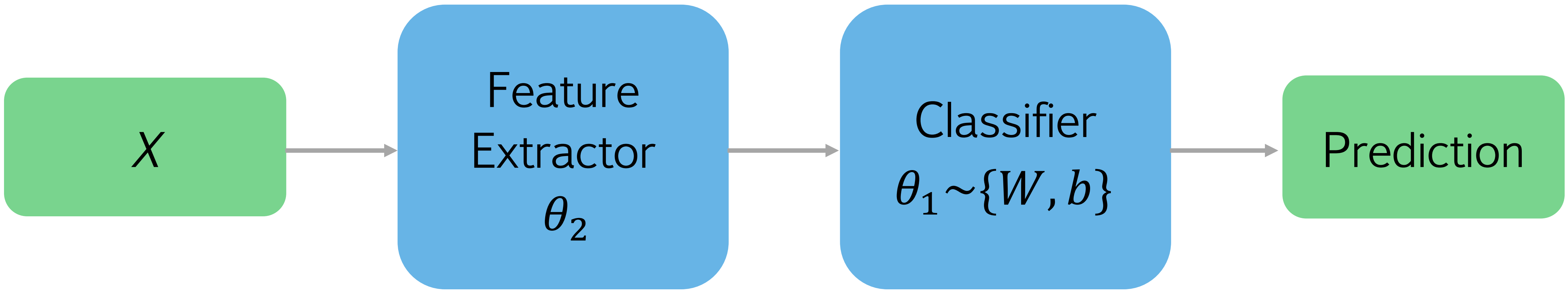


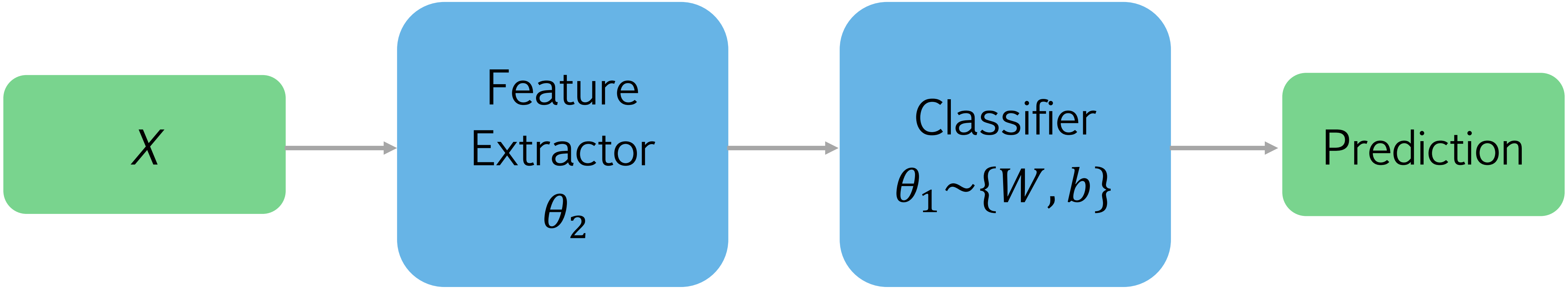
Image: Alex Burnap et al.

Feature extraction



- *Manually* extracted features
- Training is left with finding $\underset{\theta_1}{\operatorname{argmin}} L\big(y, P(y \mid x)\big)$

Can it be done automatically?

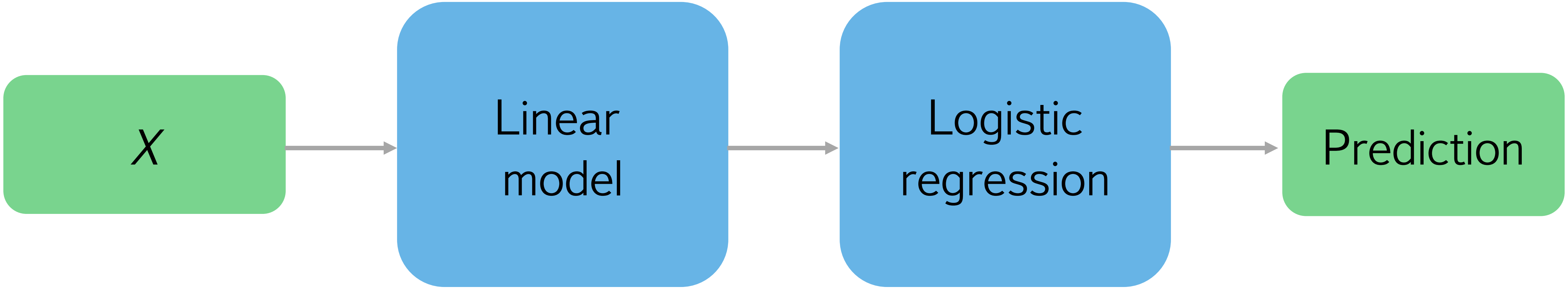


- *Automatically* extracted features
- Training still needs to find $\underset{\theta_1}{\operatorname{argmin}} L(y, P(y \mid x))$
- Yet, we face a different challenge of finding θ_2

Automatically - can feature ext

Try stacked linear models

Spoiler: Doesn't Work! Still a



- Compute features $$h_j = \sum_i w_{ij}^h x_i + b_j^h \qquad j \in \{1,2,\ldots,n\}$$
- Eventual output of the model $$y_{pred} = \sigma\left(\sum_j w_j^o h_j + b^o\right)$$
- Train by jointly minimizing loss to search for $$\underset{w^h, w^0, b^h, b^0}{\operatorname{argmin}} L(y, P(y \mid x))$$

A question

- Will stacking linear functions improve quality?

Answer: **no**

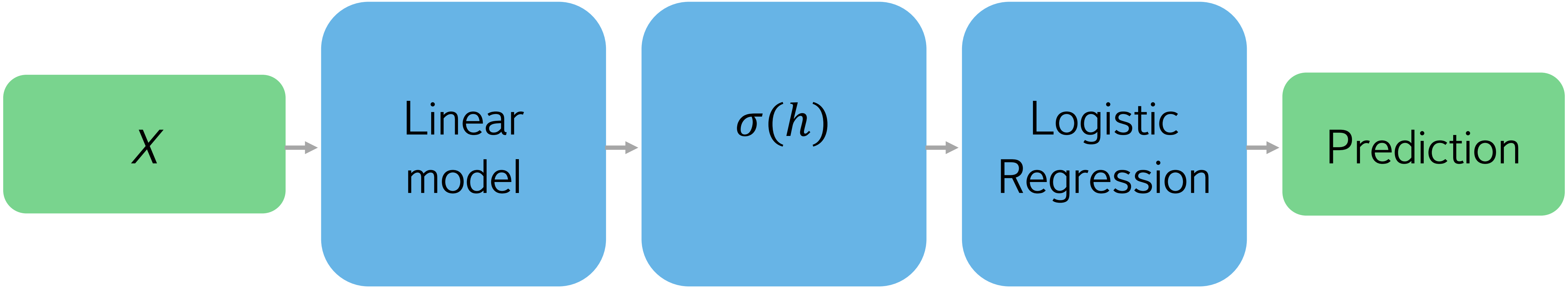
- Why?
- A combination of linear models is a linear model:

$$P(y \mid x) = \sigma\left(\sum_j w_j^o \left(\sum_i w_{ij}^h x_i + b_j^h\right) + b^o\right)$$

$$w'_i = \sum_j w_j^o w_{ij}^h \qquad b' = \sum_j w_j^o b_j^h + b^o$$

$$P(y \mid x) = \sigma\left(\sum_i w'_i x_i + b'\right)$$

The nonlinearity

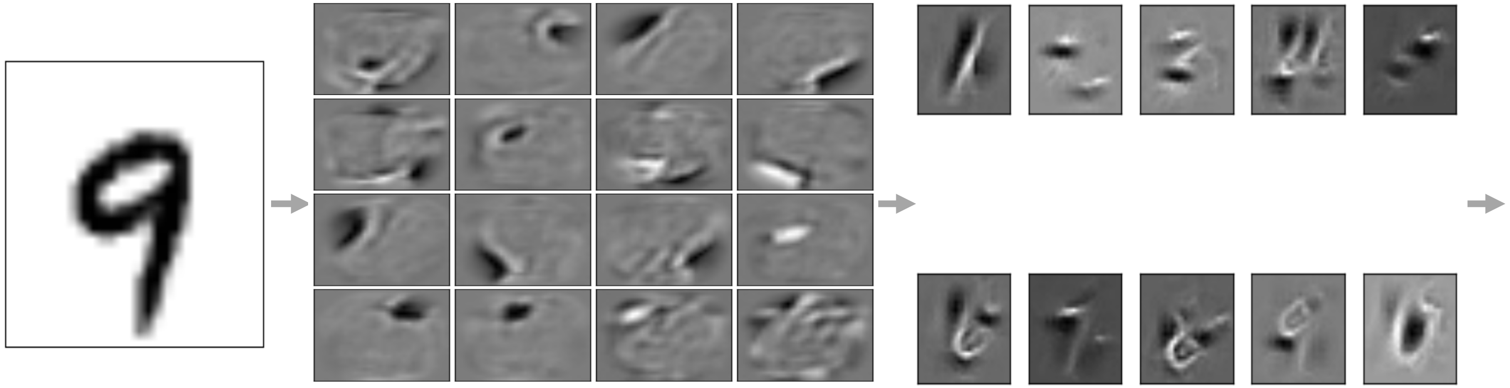


- Compute features $h_j = \sigma\left(\sum_i w_{ij}^h x_i + b_j^h\right) \quad j \in \{1,2,\ldots,n\}$
- Eventual output of the model $y_{pred} = \sigma\left(\sum_j w_j^o h_j + b^o\right)$
- Compositionality: $P(y \mid x) = \sigma\left(\sum_j w_j^o \sigma\left(\sum_i w_{ij}^h x_i + b_j^h\right) + b^o\right)$

Effect of the nonlinearity

lly we combine a bunch of non-lin

X → Linear model → $\sigma(h)$ *Nonlinearity* → Logistic Regression → Prediction



$P(\text{"1"})$
$P(\text{"2"})$
...
$P(\text{"9"})$

Balakrishnan Prabhu, https://corpocrat.com/2014/10/17/machine-learning-using-restricted-boltzmann-machines/

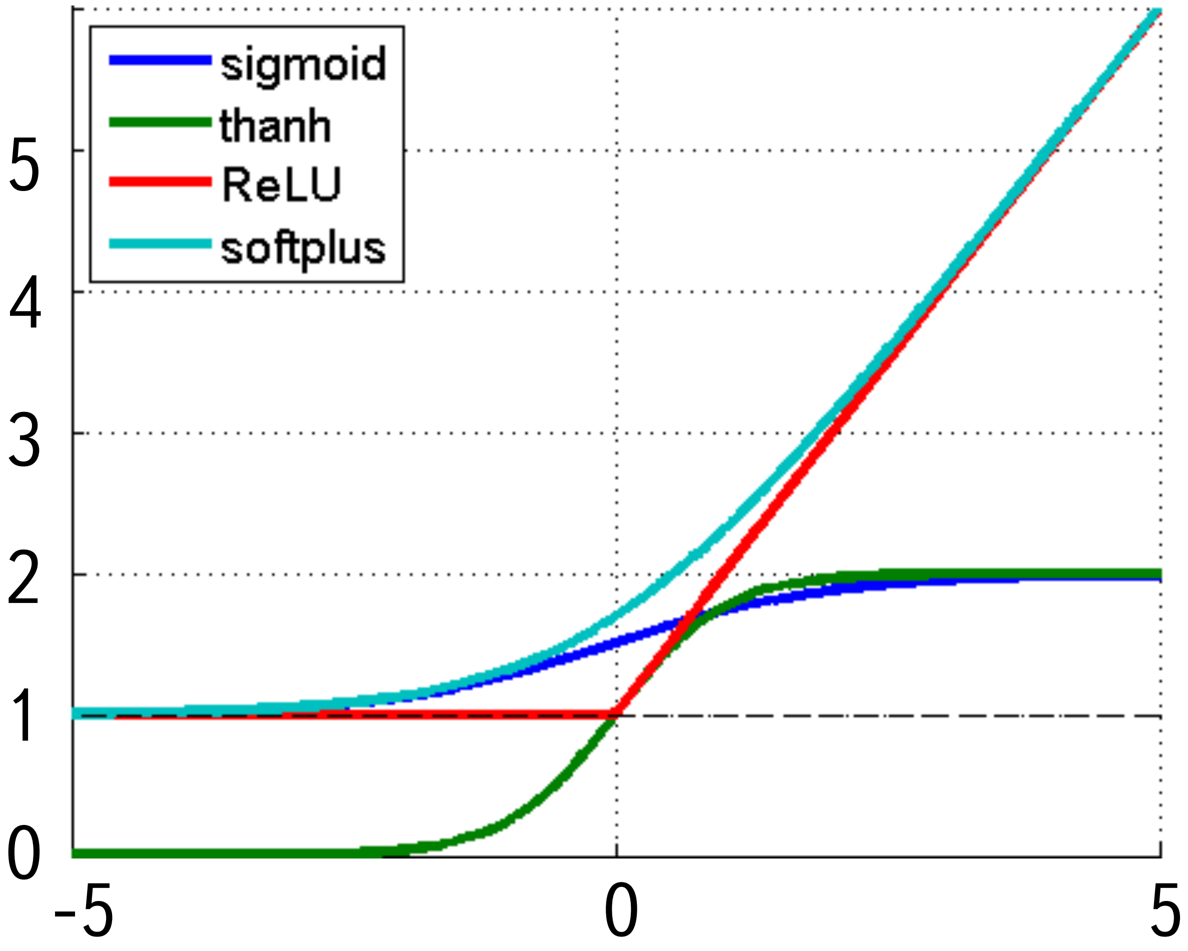
Types of nonlinearity

$$f(a) = \frac{1}{1 + e^{a}}$$

$$f(a) = \tanh(a)$$

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

$$f(a) = \log(1 + e^{a})$$



Wei Di, https://imiloainf.wordpress.com/2013/11/06/rectifier-nonlinearities/

Recap and terminology

etworks - combinations of thes

- Layer is a building block for neural network:
  – Input layer
  – Dense layer: $f(x) = Wx + b$
  – Nonlinearity layer: $f(x) = \sigma(x)$
  – A few more: we will cover later

Hidden layers

- Output layer
- Activation is layer output

ctivation is output of the output

  – i. e. some intermediate signal in the neural network

process layers in series of th

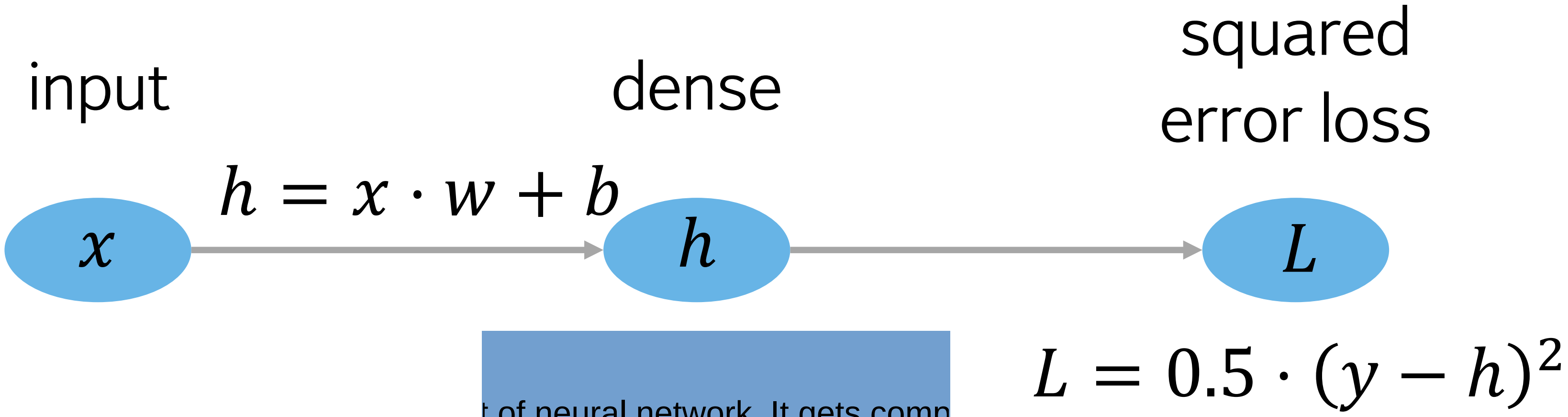
Potential caveats?

- Hardcore overfitting
- No “golden standard” for architecture
- Computationally heavy

The backpropagation algorithm

The simplest NN ever

.com/data-science-group-iitr/loss-functions-and-optimization-algori

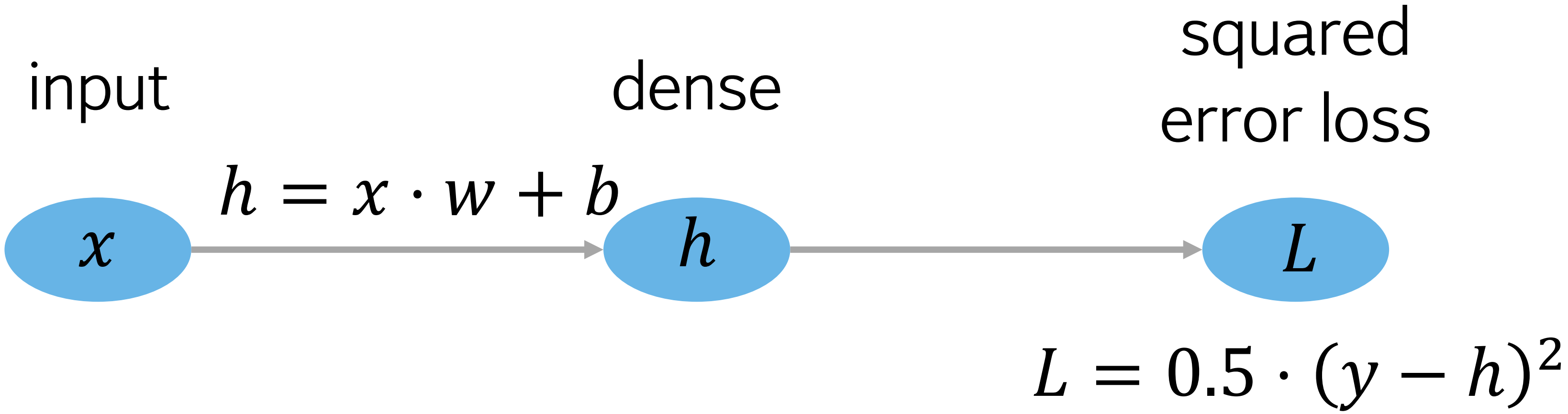


t of neural network. It gets comp

- Parameters:
  - Weight w and bias b
- Input: x
- Target: y

Also known as
least squares linear regression

The simplest NN ever

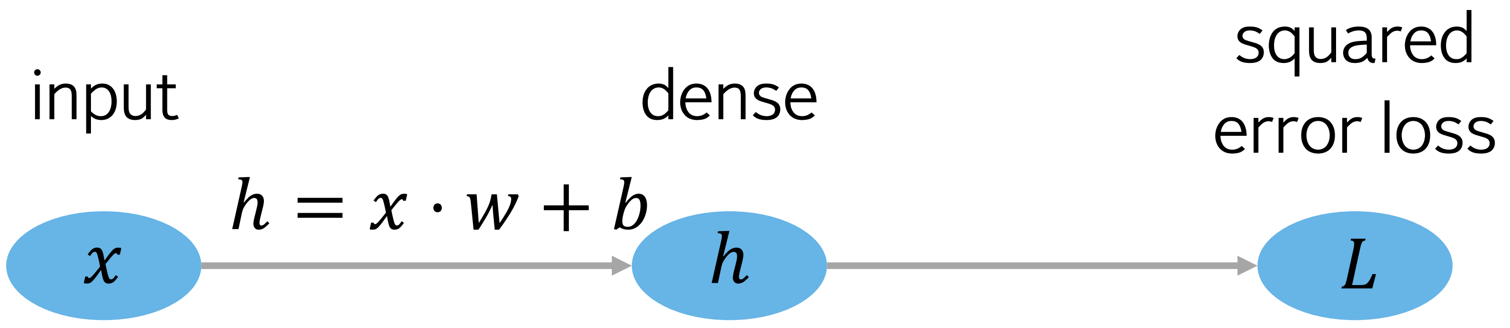


- Parameters:
  - Weight w and bias b
- Input: x
- Target: y

L is just a function of parameters, features and target:

$$L = f\big(y, g(x, w, b)\big)$$

The simplest NN ever



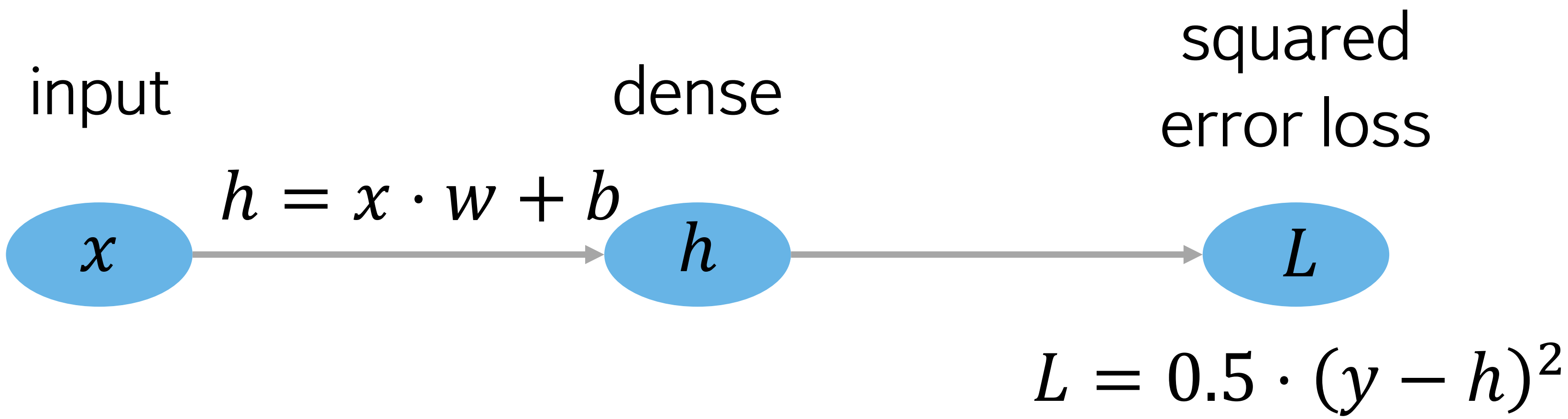
$$L = 0.5 \cdot (y - h)^2$$

- Gradient? Gradient - one way to do this

- $\dfrac{\partial L}{\partial b} = \dfrac{\partial L}{\partial h} \cdot \dfrac{\partial h}{\partial b}$

s a function that maps an event or values of one or m

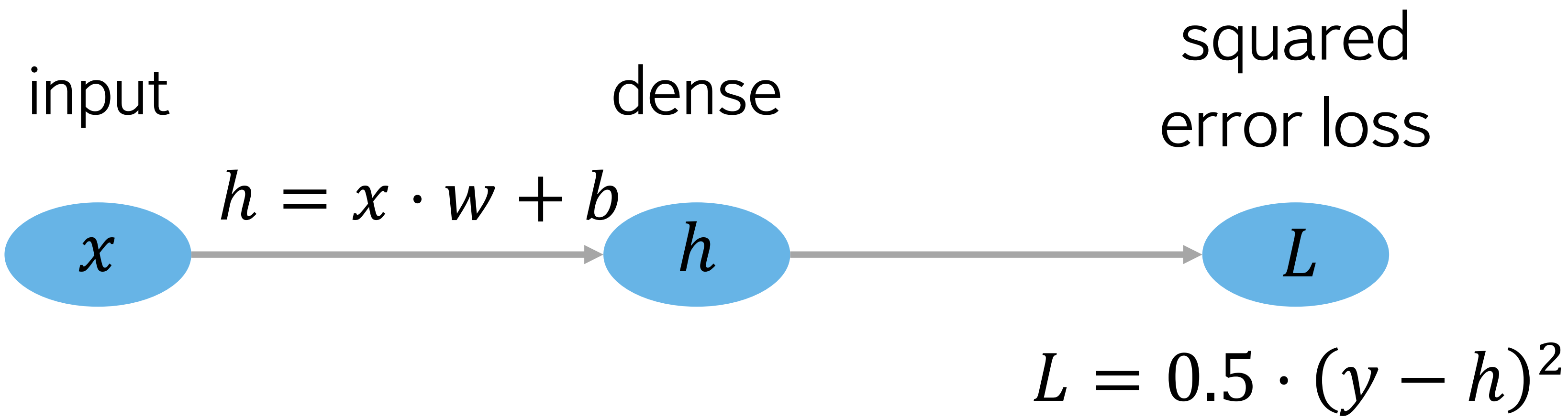
The simplest NN ever



- Let’s fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b

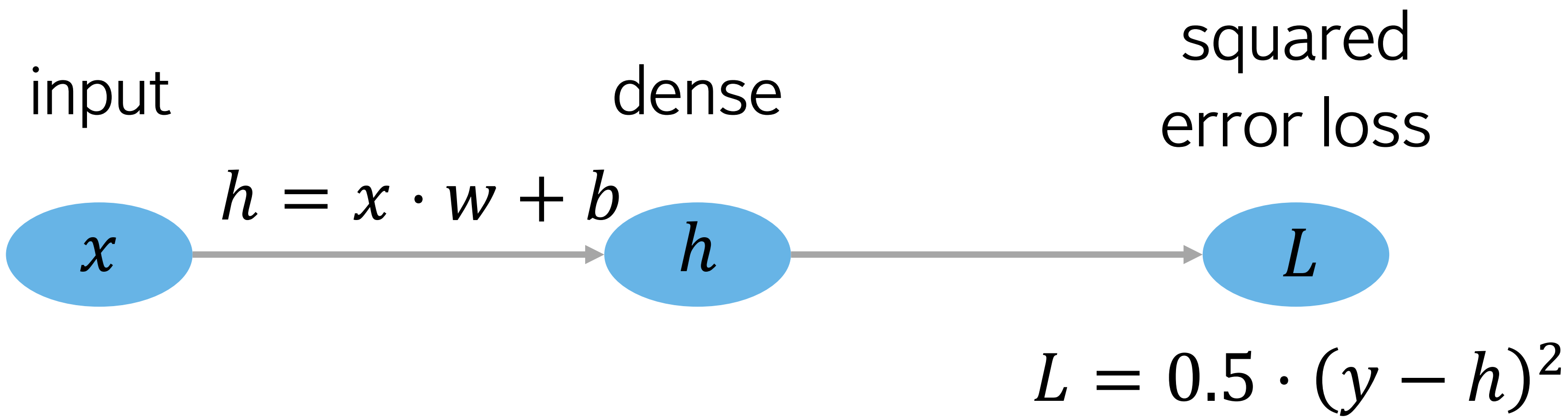
Forward pass



- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80					

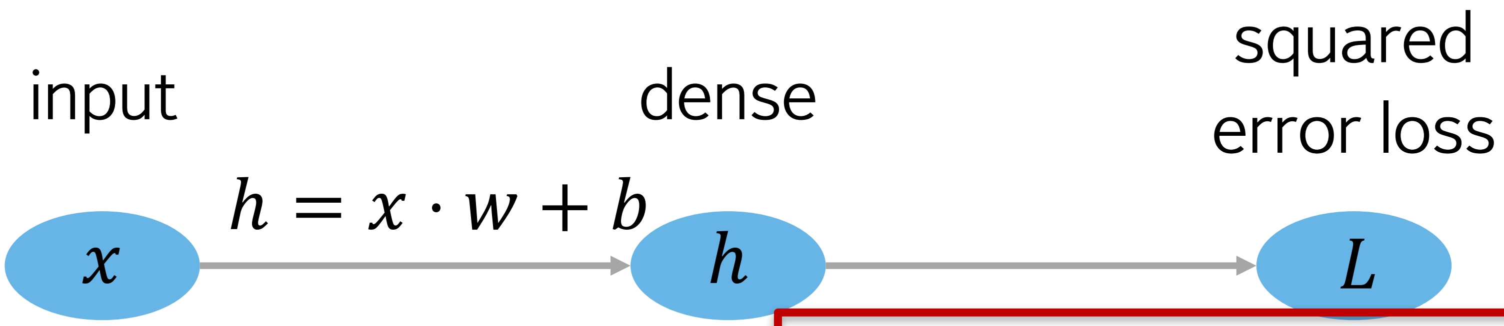
Backward pass



- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	$\boxed{-1.9}$				

Backward pass

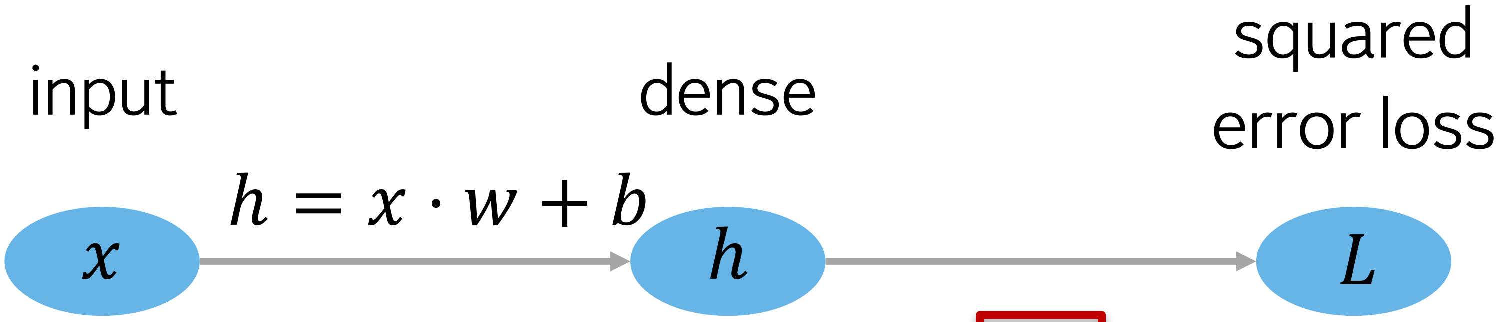


$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Backward pass

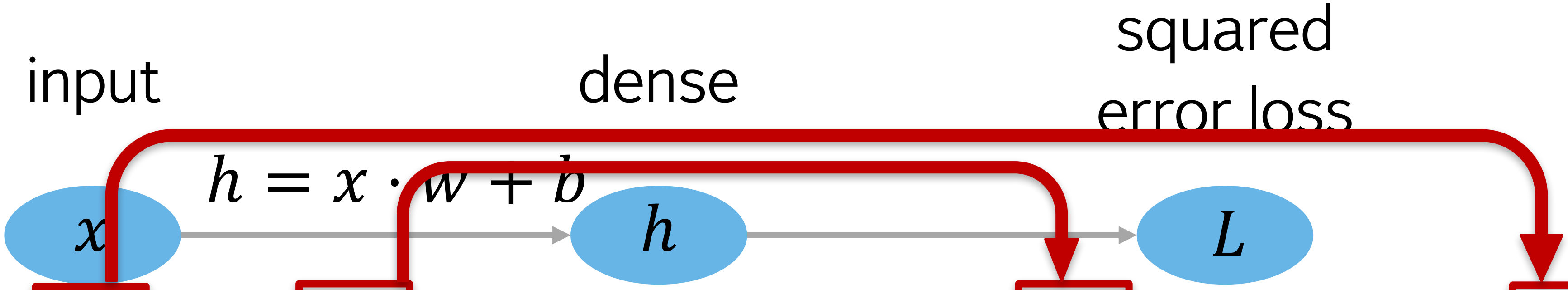


$$\frac{\partial L}{\partial w} = \boxed{\frac{\partial L}{\partial h}} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \boxed{\frac{\partial L}{\partial h}} \cdot \frac{\partial h}{\partial b}$$

- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\boxed{\frac{\partial L}{\partial h}}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Backward pass

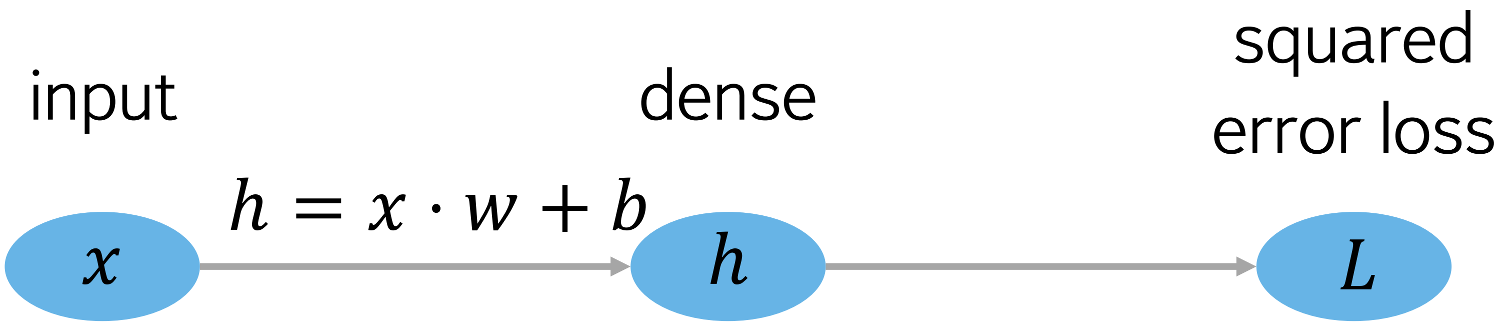


$$\boxed{\frac{\partial h}{\partial b}} = 1 \qquad \boxed{\frac{\partial h}{\partial w}} = x \qquad \frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \boxed{\frac{\partial h}{\partial w}} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \boxed{\frac{\partial h}{\partial b}}$$

- Let's fit
  - $y = 3, x = 1$ single poi
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Backward pass



$$\frac{\partial h}{\partial b} = 1 \qquad \frac{\partial h}{\partial w} = x \qquad \frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	$\boxed{-1.9}$	$\boxed{-1.9}$		

Backward pass

input $\qquad$ dense $\qquad$ squared error loss

$$x \xrightarrow{h = x \cdot w + b} h \longrightarrow L$$

$$\boxed{\frac{\partial L}{\partial w}} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \boxed{\frac{\partial L}{\partial b}} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

rs are updated as it progresses through *backpro

- Update parameters:

$$w \mathrel{-}= \eta \boxed{\frac{\partial L}{\partial w}}$$

$$b \mathrel{-}= \eta \boxed{\frac{\partial L}{\partial b}}$$

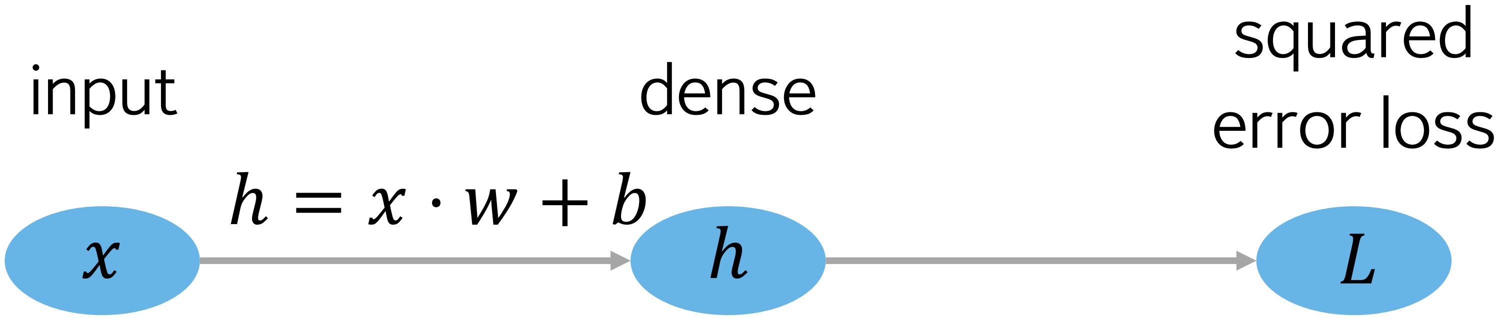
$$\eta = 0.2$$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9		

We choose eta ourselves - design feature.

riable to change. How many percents of your weight do y

Backward pass



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Update parameters:

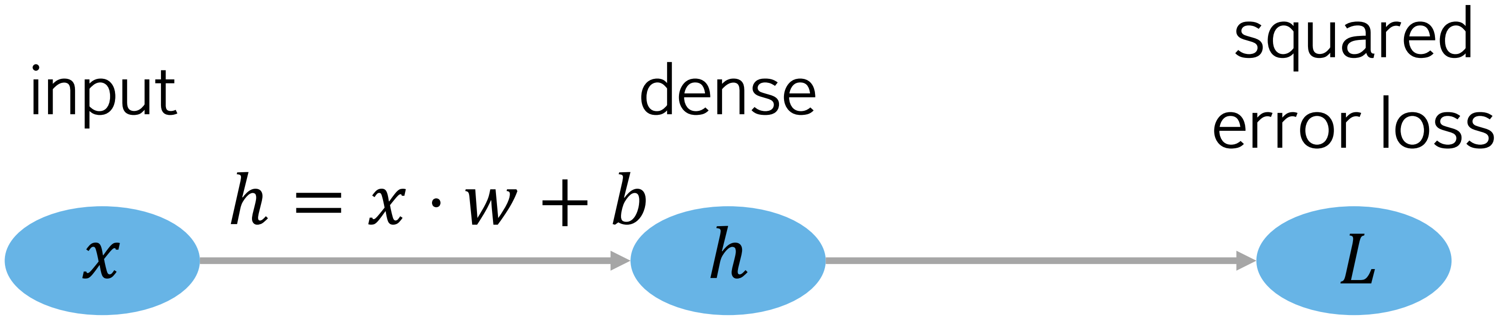
$$w \mathrel{-}= \eta \frac{\partial L}{\partial w}$$

$$b \mathrel{-}= \eta \frac{\partial L}{\partial b}$$

$$\eta = 0.2$$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9	$\boxed{0.48}$	$\boxed{1.38}$

After a few more updates...



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Update parameters:

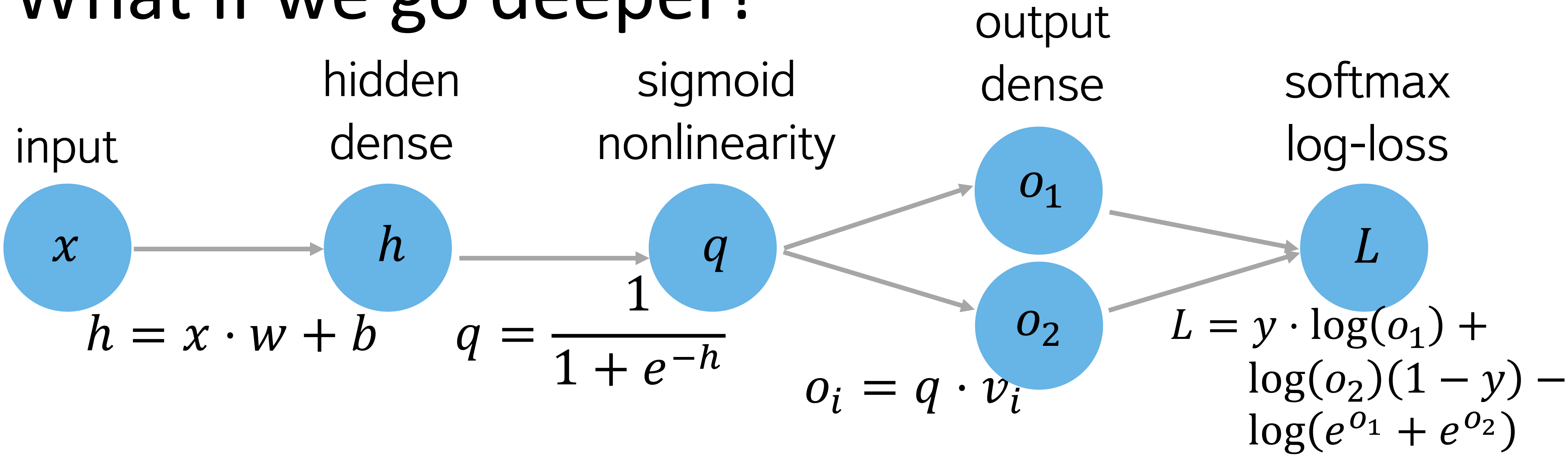
$$w \mathrel{-}= \eta \frac{\partial L}{\partial w}$$

$$b \mathrel{-}= \eta \frac{\partial L}{\partial b}$$

$$\eta = 0.2$$

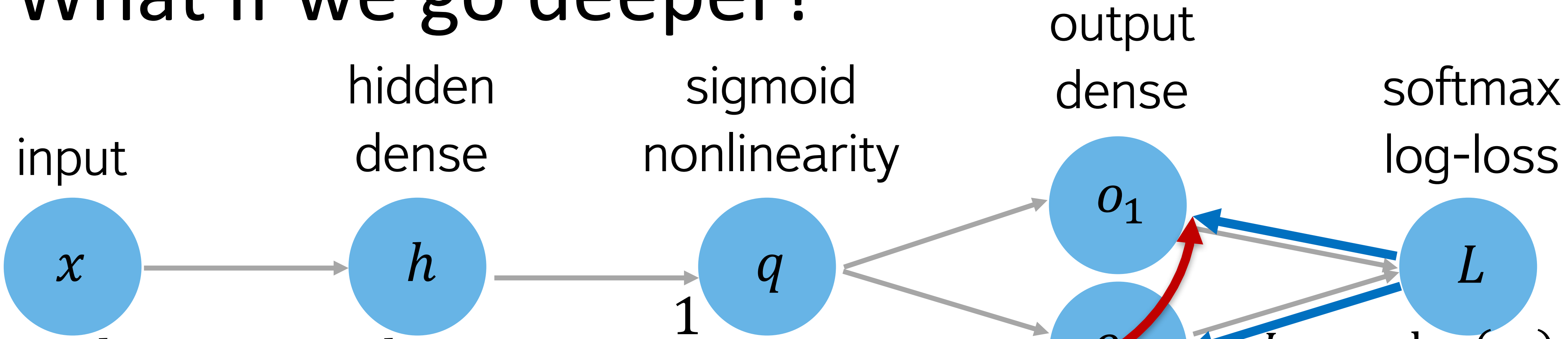
h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9	0.48	1.38
1.86	0.65	-1.14	-1.14	-1.14	0.71	1.61
2.32	0.23	-0.68	-0.68	-0.68	0.84	1.75

What if we go deeper?



- Parameters:
  - Weight w and bias b
  - Weights v_1, v_2

What if we go deeper?



$$h = x \cdot w + b \qquad q = \frac{1}{1 + e^{-h}}$$

$$o_i = q \cdot v_i$$

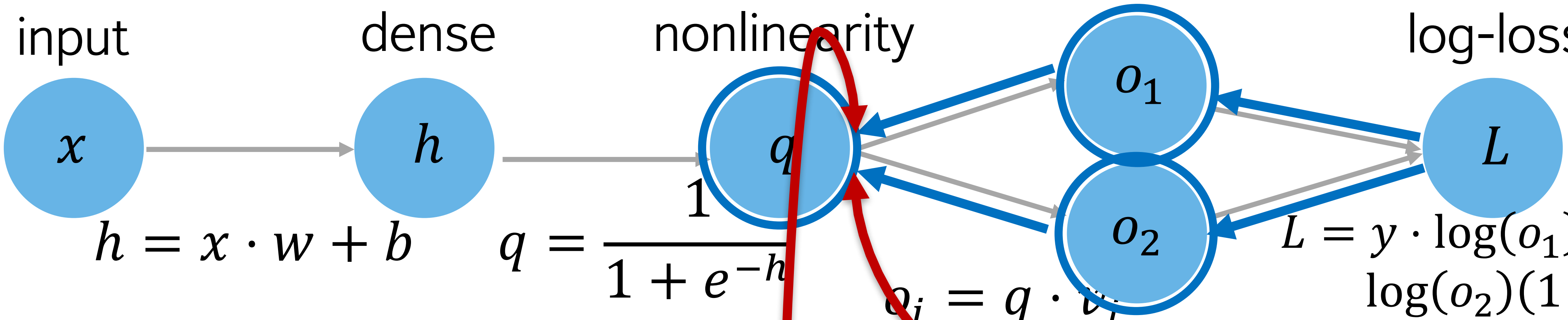
$$L = y \cdot \log(o_1) + \log(o_2)(1 - y) - \log(e^{o_1} + e^{o_2})$$

- Parameters:
  - Weight w and bias b
  - Weights v_1, v_2

$$\boxed{\frac{dL}{do_1}} = \frac{y}{o_1} - \frac{e^{o_1}}{e^{o_2} + e^{o_1}}$$

$$\boxed{\frac{dL}{do_2}} = \frac{1 - y}{o_2} - \frac{e^{o_2}}{e^{o_2} + e^{o_1}}$$

What if we go deeper?



$h = x \cdot w + b$ $\quad q = \dfrac{1}{1 + e^{-h}}$ $\quad o_i = q \cdot v_i$

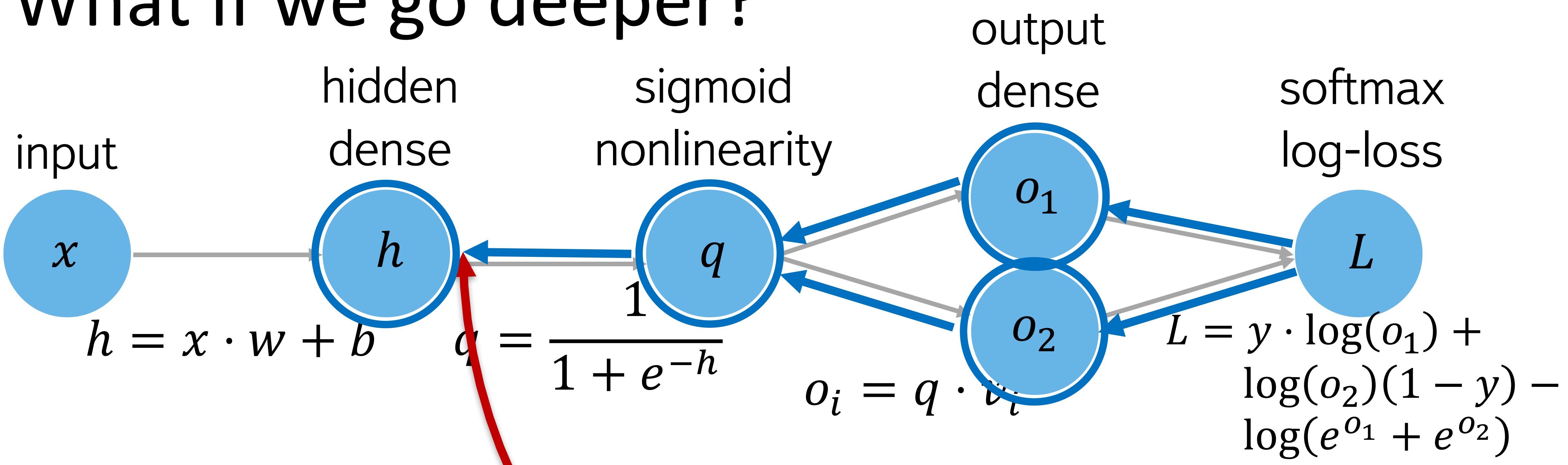
$L = y \cdot \log(o_1) + \log(o_2)(1 - y) - \log(e^{o_1} + e^{o_2})$

- Parameters:
  - Weight w and bias b
  - Weights v_1, v_2

$$\frac{\partial L}{\partial q} = v_1 \cdot \frac{\partial L}{\partial o_1} + v_2 \cdot \frac{\partial L}{\partial o_2}$$

$$\boxed{\frac{\partial L}{\partial v_1}} = \frac{\partial L}{\partial o_1} \cdot q \qquad \boxed{\frac{\partial L}{\partial v_2}} = \frac{\partial L}{\partial o_2} \cdot q$$

What if we go deeper?



- Parameters:
  - Weight w and bias b
  - Weights v_1, v_2

$$\boxed{\frac{\partial L}{\partial h}} = \frac{\partial L}{\partial q} \frac{e^{-q}}{(1 + e^{-q})^2}$$

Backpropagation: the algorithm

- Chain rule can be evaluated numerically!
- Compute the network output and the loss value
- Compute "dLoss" /"dActivation_of_output_layer"
- For each layer, starting from the last:
  - Compute “dActivation” / “dLayer_parameters”, “dActivation” / “dLayer_input”
  - Multiply it by “dLoss” / “dActivation”, get “dLoss” /…
- Make optimization step for the parameters

hnique called automatic differentiation. In the

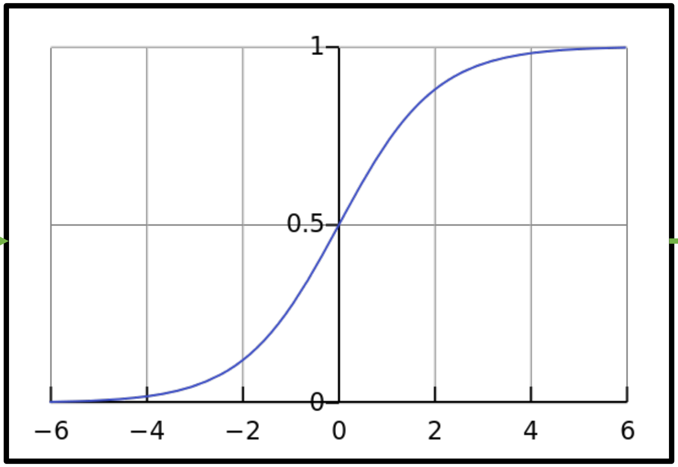
Intermediate conclusion

- You can have any crazy layer as long as you can compute its gradient
- **In fact:** no need to compute the gradients by hand!
  - There are frameworks for that (e.g. theano, tensorflow, and pytorch)

Tricks for training deep neural models

Sigmoid activation

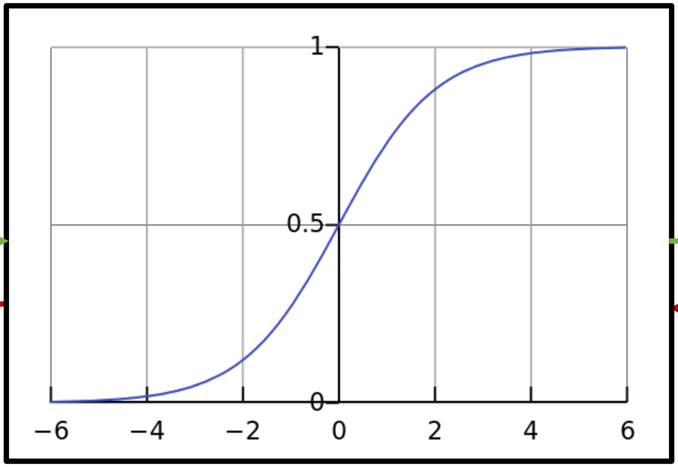
x



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

Sigmoid activation

x



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

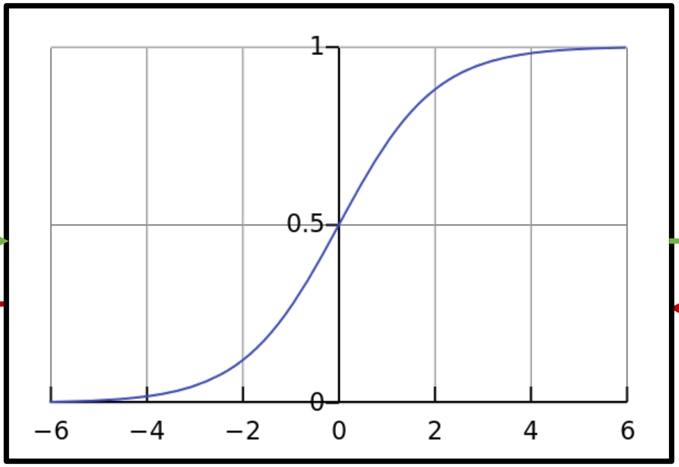
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial x}$$

$$\frac{\partial L}{\partial \sigma}$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Sigmoid activation

x



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

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial x}$$

$$\frac{\partial L}{\partial \sigma}$$

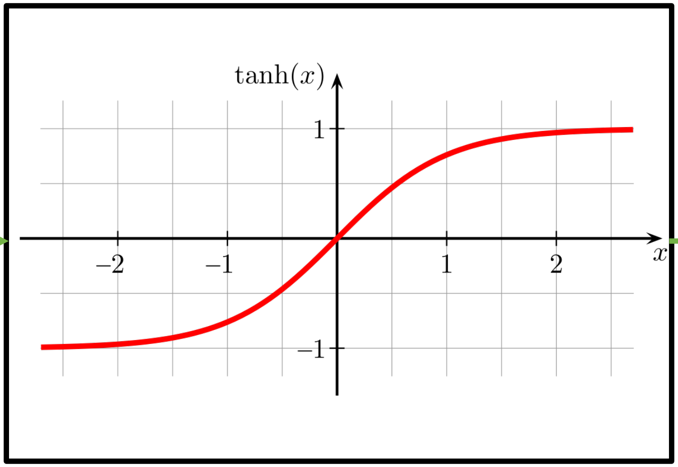
$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x))$$

- Sigmoid neurons can saturate and lead to vanishing gradients
- Not zero-centered
- e^x is computationally expensive

ry values - no meaningful solutio

Tanh activation

x



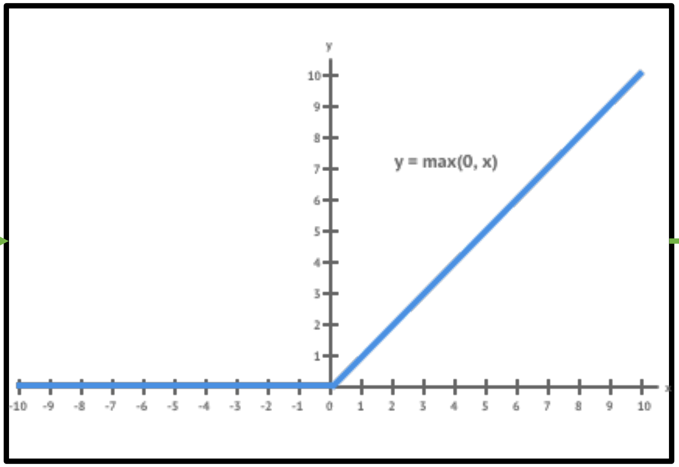
$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- Zero-centered
- But still pretty much like sigmoid

aturates and has vanishing gra

ReLU activation

x



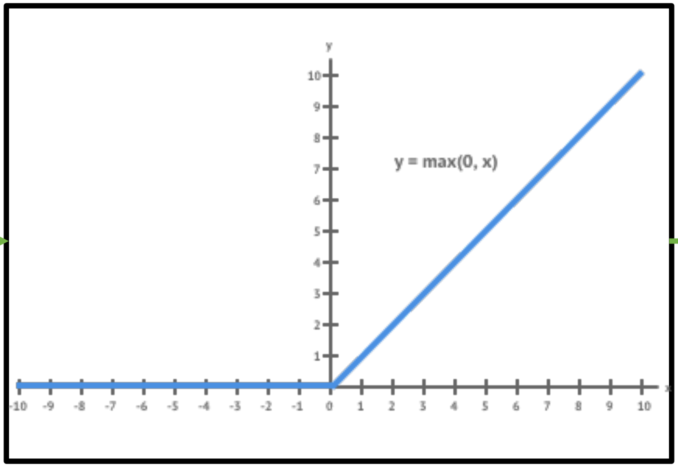
$f(x) = \max(0, x)$

e property. Eliminates bias towa

- Fast to compute
- Gradients do not vanish for $x > 0$
- Provides faster convergence in practice!

ReLU activation

x

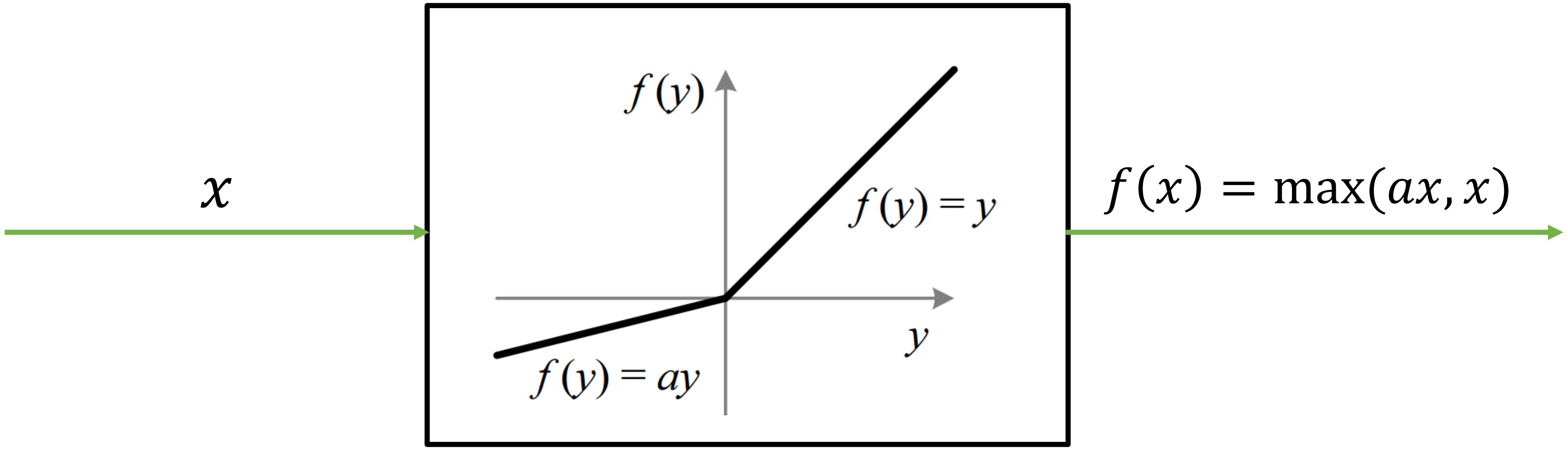


$f(x) = \max(0, x)$

- Fast to compute.
- Gradients do not vanish for $x>0$.
- Provides faster convergence in practice!
- Not zero-centered.
- Can die: if not activated, never updates!

ative values will basically turn-off th

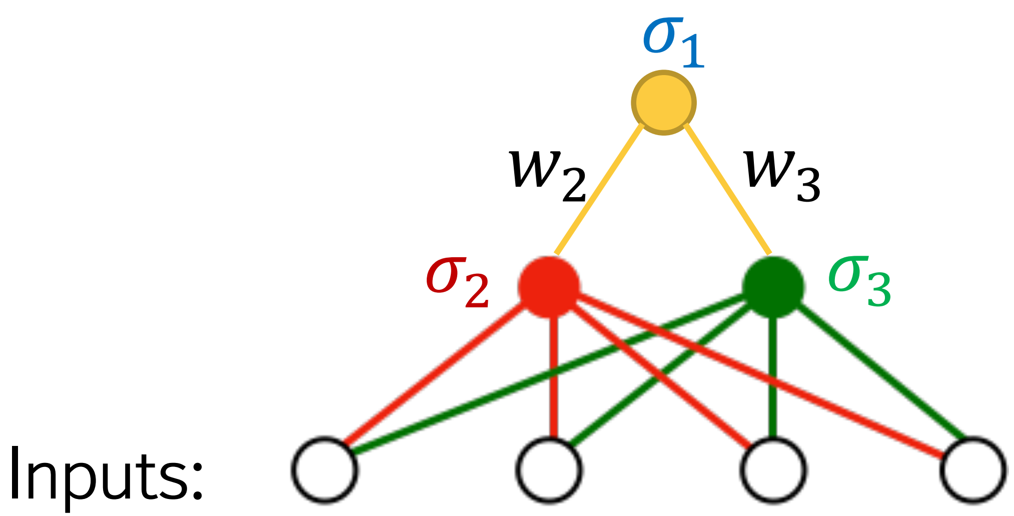
Leaky ReLU activation



- Will not die!
- $a \neq 1$

***Usually 10^-2

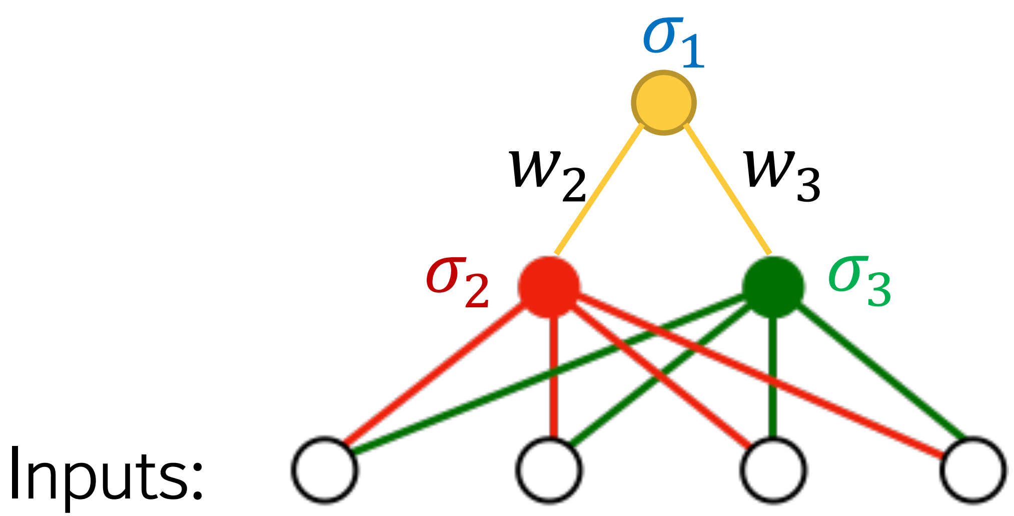
Weights initializations



$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \sigma_1} \sigma_1 (1 - \sigma_1) \sigma_2$$
$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \sigma_1} \sigma_1 (1 - \sigma_1) \sigma_3$$

- Maybe start with all zeros?

Weights initializations



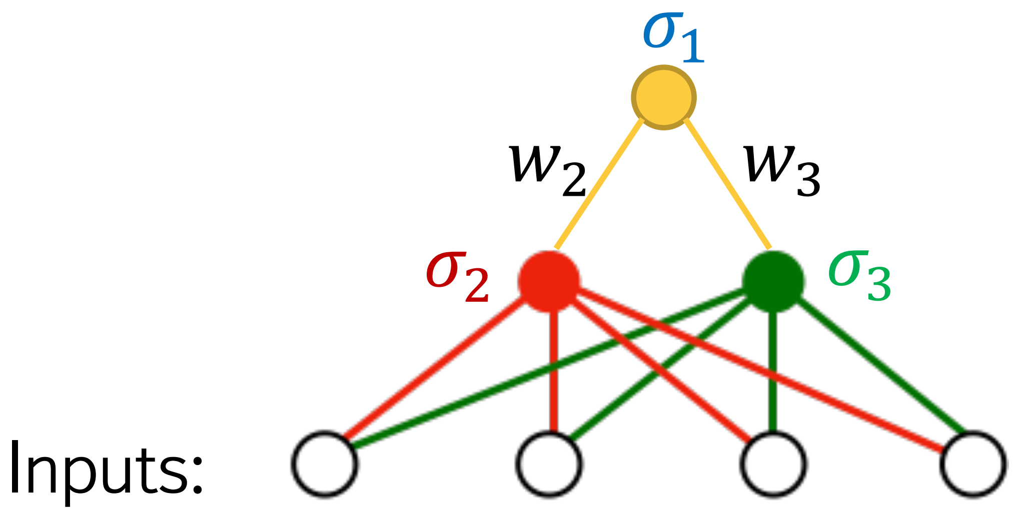
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \sigma_1} \sigma_1 (1 - \sigma_1) \sigma_2$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \sigma_1} \sigma_1 (1 - \sigma_1) \sigma_3$$

σ_2 and σ_3 will always get the same updates!

- Maybe start with all zeros?

Weights initializations



$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \sigma_1}\sigma_1(1-\sigma_1)\sigma_2$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \sigma_1}\sigma_1(1-\sigma_1)\sigma_3$$

σ_2 and σ_3 will always get the same updates!

- ~~Maybe start with all zeros?~~
- Need to break symmetry!
- Maybe start with small random numbers then?
- But how small? $0.03 \cdot \mathcal{N}(0,1)$?

Weights initializations

- Linear models work best when inputs are normalized.
- Neuron is a linear combination of inputs + activation.
- Neuron output will be used by consecutive layers.

Weights initializations

- Let's look at the neuron output **before activation:** $\sum_{i=1}^{n} x_i w_i$.
- If $E(x_i) = E(w_i) = 0$ and we generate weights independently from inputs, then $E(\sum_{i=1}^{n} x_i w_i) = 0$.
- But variance can grow with consecutive layers.
- Empirically this hurts convergence for deep networks!

Weights initializations

- Let's look at the variance of $\sum_{i=1}^{n} x_i w_i$:

Weights initializations

- Let's look at the variance of $\sum_{i=1}^{n} x_i w_i$: i.i.d. w_i and mostly uncorrelated x_i

$Var(\sum_{i=1}^{n} x_i w_i) =$

$= \sum_{i=1}^{n} Var(x_i w_i) =$

Weights initializations

- Let's look at the variance of $\sum_{i=1}^{n} x_i w_i$: i.i.d. w_i and mostly uncorrelated x_i

$$Var(\sum_{i=1}^{n} x_i w_i) =$$

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$

independent factors w_i and x_i

$$= \sum_{i=1}^{n} \begin{pmatrix} [E(x_i)]^2 Var(w_i) \\ +[E(w_i)]^2 Var(x_i) \\ +Var(x_i)Var(w_i) \end{pmatrix} =$$

Weights initializations

- Let's look at the variance of $\sum_{i=1}^{n} x_i w_i$: i.i.d. w_i and mostly uncorrelated x_i

$$Var(\sum_{i=1}^{n} x_i w_i) =$$

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$

independent factors w_i and x_i

$$= \sum_{i=1}^{n} \begin{pmatrix} [E(x_i)]^2 Var(w_i) \\ +[E(w_i)]^2 Var(x_i) \\ +Var(x_i)Var(w_i) \end{pmatrix} =$$

w_i and x_i have 0 mean

$$= \sum_{i=1}^{n} Var(x_i)Var(w_i) = Var(x)[\boldsymbol{n\,Var(w)}]$$

Weights initializations

- Let's look at the variance of $\sum_{i=1}^{n} x_i w_i$: i.i.d. w_i and mostly uncorrelated x_i

$$Var(\sum_{i=1}^{n} x_i w_i) =$$

$$= \sum_{i=1}^{n} Var(x_i w_i) =$$ independent factors w_i and x_i

$$= \sum_{i=1}^{n} \begin{pmatrix} [E(x_i)]^2 Var(w_i) \\ +[E(w_i)]^2 Var(x_i) \\ +Var(x_i) Var(w_i) \end{pmatrix} =$$ w_i and x_i have 0 mean

$$= \sum_{i=1}^{n} Var(x_i) Var(w_i) = Var(x)[\boldsymbol{n\,Var(w)}]$$

We want this to be 1

Weights initializations

- Let's use the fact that $Var(aw) = a^2Var(w)$.
- For $[\boldsymbol{n}\, \boldsymbol{Var}(\boldsymbol{aw})]$ to be 1 we need to multiply $\mathcal{N}(0,1)$ weights ($Var(w) = 1$) by $a = 1/\sqrt{n}$.
- Xavier initialization (Glorot et al.) multiplies weights by $\sqrt{2}/\sqrt{n_{in} + n_{out}}$.
- Initialization for ReLU neurons (He et al.) uses multiplication by $\sqrt{2}/\sqrt{n_{in}}$.

Batch normalization

- We know how to initialize our network to constrain variance.
- But what if it grows during backpropagation?
- Batch normalization controls mean and variance of outputs before activations.

Batch normalization

- Let's normalize h_i – neuron output before activation:

$$h_i = \boldsymbol{\gamma_i} \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \boldsymbol{\beta_i}$$

→ 0 mean, unit variance

Batch normalization

- Let's normalize h_i – neuron output before activation:

$$h_i = \boldsymbol{\gamma_i} \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \boldsymbol{\beta_i}$$

→ 0 mean, unit variance

- Where do μ_i and σ_i^2 come from? We can estimate them having **a current training batch!**

Batch normalization

- Let's normalize h_i – neuron output before activation:

$$h_i = \boldsymbol{\gamma_i} \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \boldsymbol{\beta_i}$$

→ 0 mean, unit variance

- Where do μ_i and σ_i^2 come from? We can estimate them having **a current training batch!**
- During testing we will use an exponential moving average over batches:

$$0 < \alpha < 1 \qquad \begin{aligned} \mu_i &= \alpha \cdot \mathbf{mean_{batch}} + (1 - \alpha) \cdot \mu_i \\ \sigma_i^2 &= \alpha \cdot \mathbf{variance_{batch}} + (1 - \alpha) \cdot \sigma_i^2 \end{aligned}$$

Batch normalization

- Let's normalize h_i – neuron output before activation:

$$h_i = \boxed{\gamma_i} \boxed{\frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}} + \boxed{\beta_i}$$

Train using backpropagation!

0 mean, unit variance

- Where do μ_i and σ_i^2 come from? We can estimate them having **a current training batch!**
- During testing we will use an exponential moving average over batches:

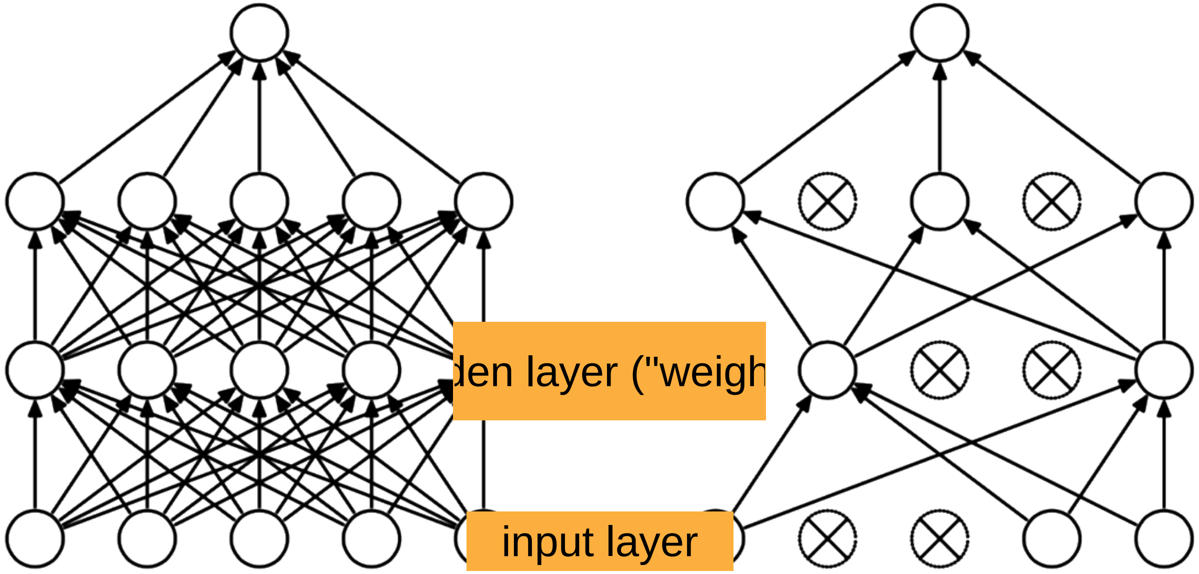
$$0 < \alpha < 1 \qquad \begin{aligned} \mu_i &= \alpha \cdot \mathbf{mean_{batch}} + (1-\alpha) \cdot \mu_i \\ \sigma_i^2 &= \alpha \cdot \mathbf{variance_{batch}} + (1-\alpha) \cdot \sigma_i^2 \end{aligned}$$

Dropout

- A **regularization** technique to reduce overfitting.
- We keep neurons active (non-zero) with probability p.
- This way we sample the network during training and change only a subset of its parameters on every iteration.

(hidden layer)--> single fea

value (each dot is different weight - eg. di



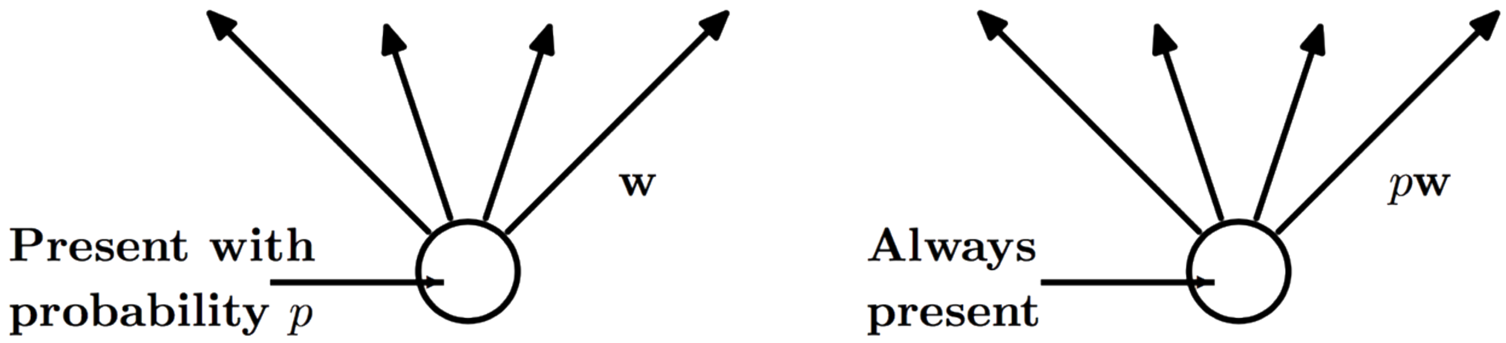
(a) Standard Neural Net (b) After applying dropout.

Nitish Srivastava, http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

Dropout

- During testing all neurons are present but their outputs are multiplied by p to maintain the scale of inputs:

Expected input weight: $p \cdot w + (1 - p) \cdot 0$



training testing

Nitish Srivastava, http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

- The authors of dropout say it is similar to having an ensemble of exponentially large number of smaller networks.

Takeaways

- Use ReLU activation
- Use He et al. initialization
- Try to add batchnorm or dropout
- Try to augment your training data