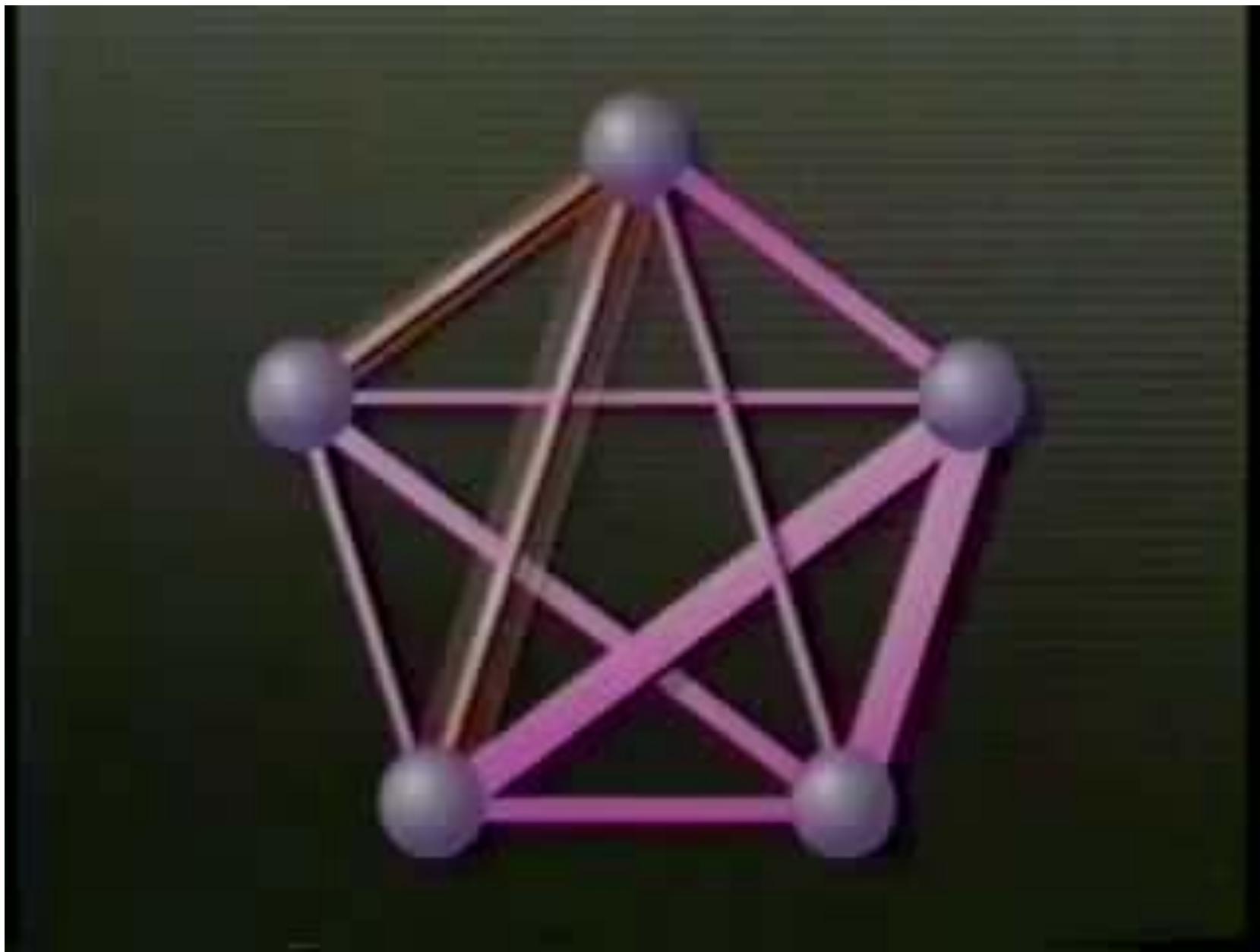


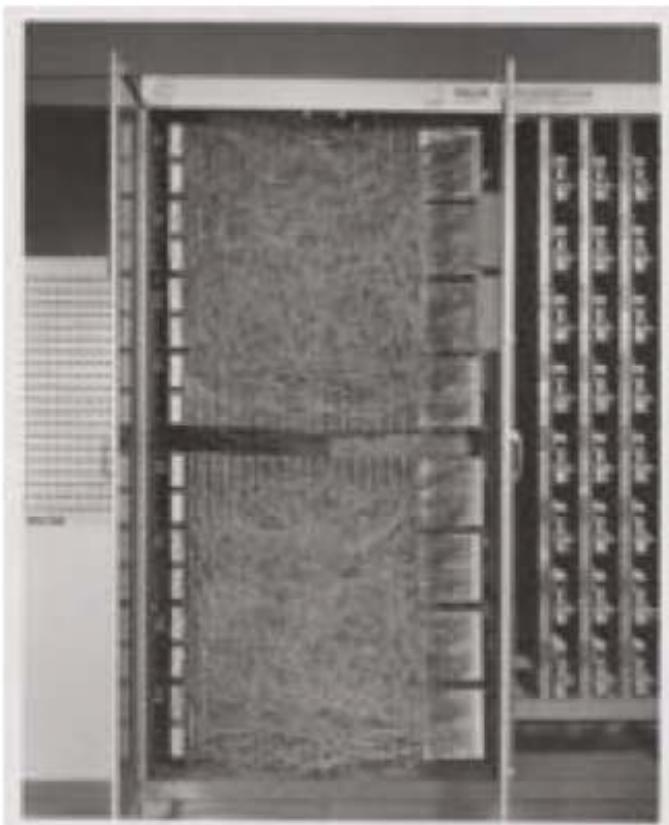
## PART II: FOUNDATIONS OF “SHALLOW” NEURAL NETWORKS



# Rosenblatt Perceptron

FIRST IMPLEMENTATION OF NEURAL NETWORK [Rosenblatt, 1957!]

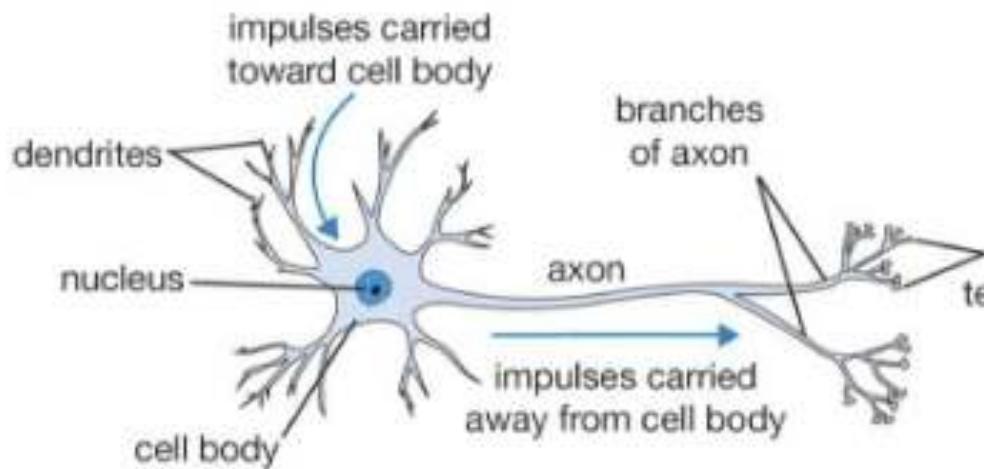
INTENDED TO BE A MACHINE (NOT AN ALGORITHM)



it had an array of 400 photocells, randomly connected to the "neurons".

Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors

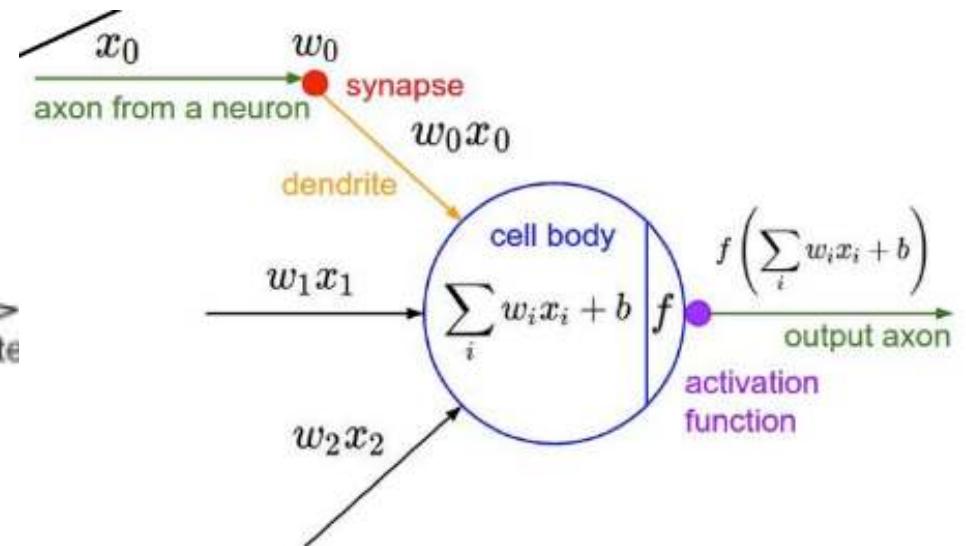
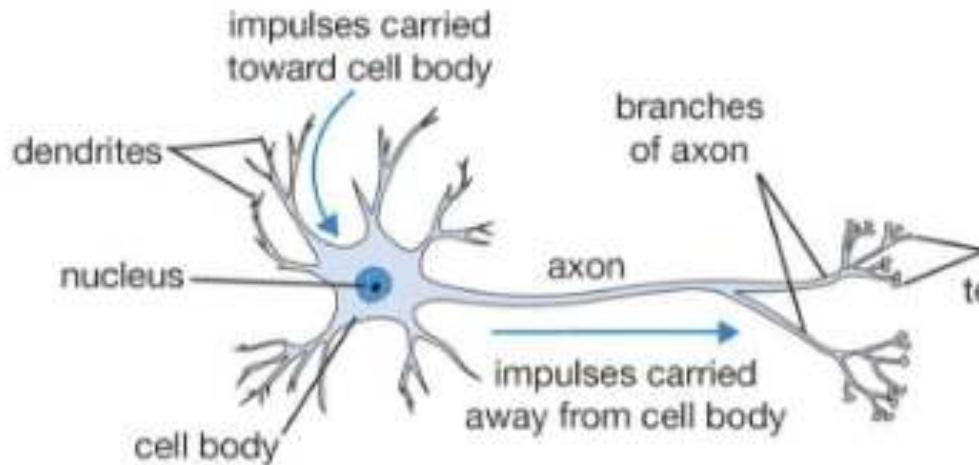
# THE NEURON



INSPIRED BY NEURO - SCIENCE?

Credit: Karpathy

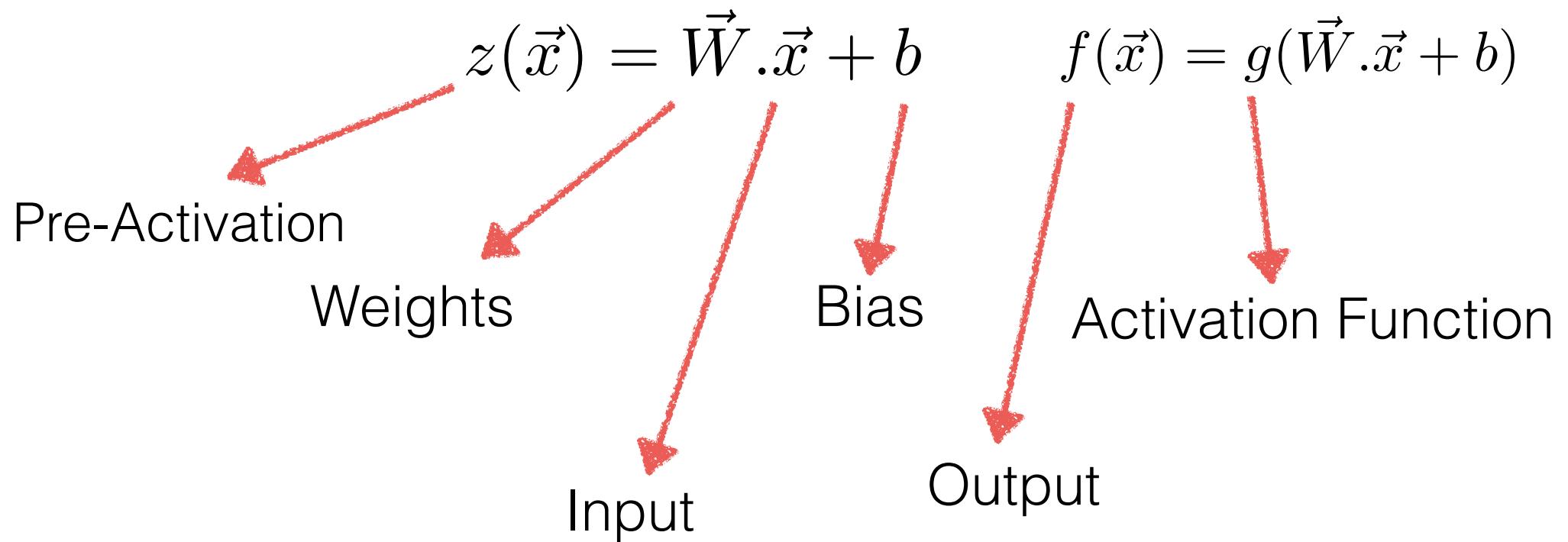
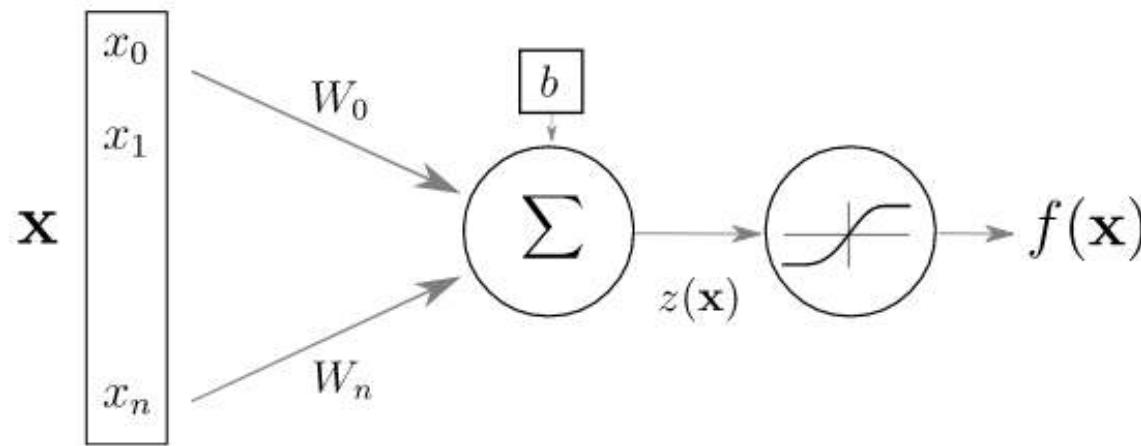
# THE NEURON



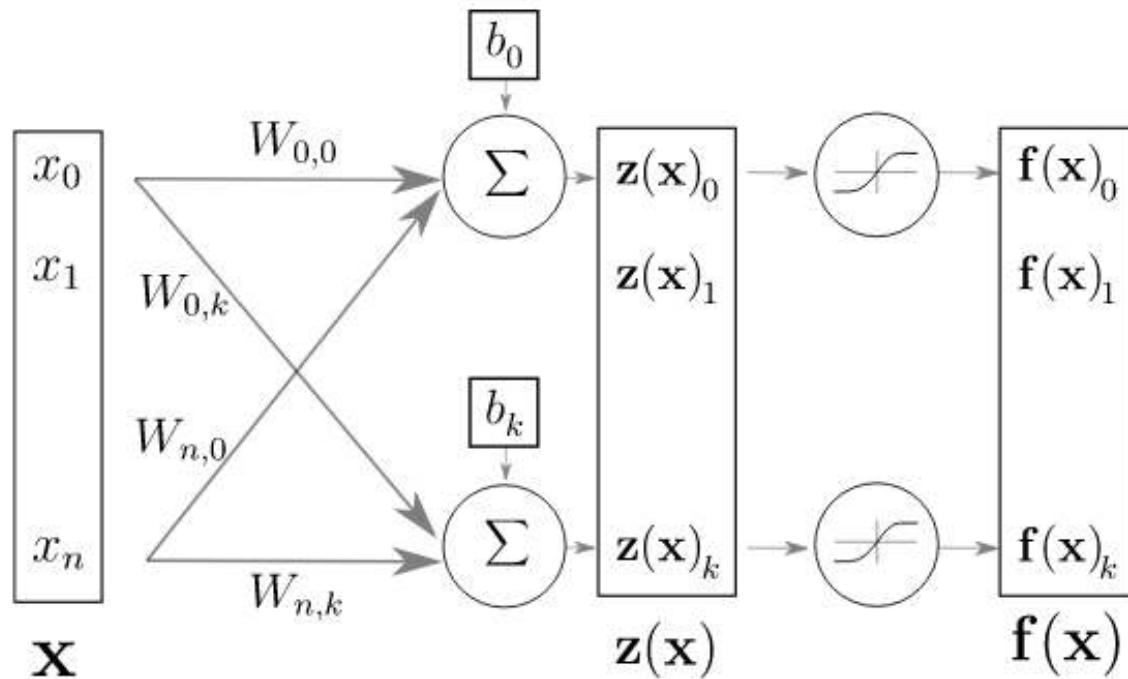
INSPIRED BY NEURO - SCIENCE?

Credit: Karpathy

# TODAY'S ARTIFICIAL NEURON



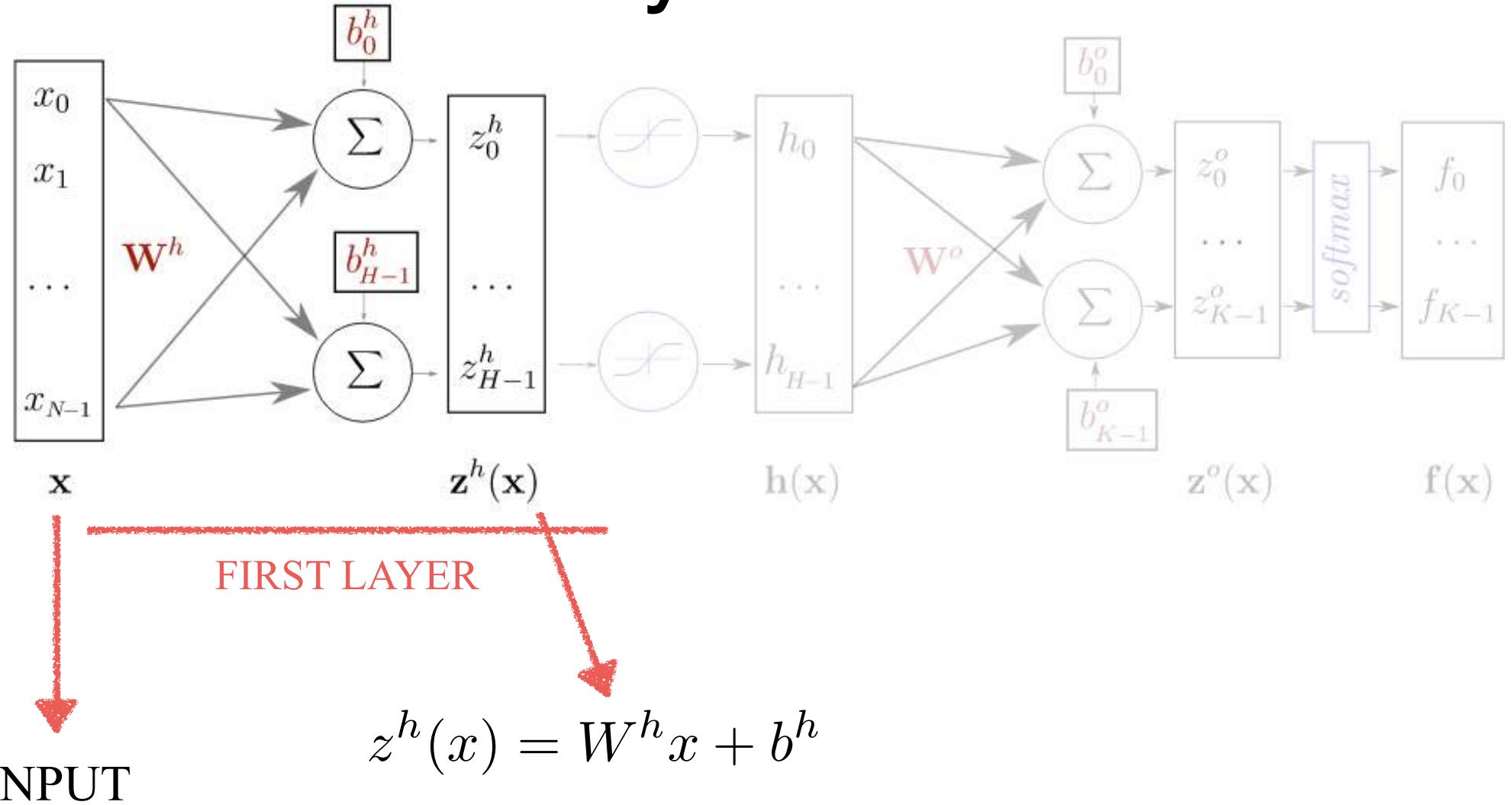
# LAYER OF NEURONS



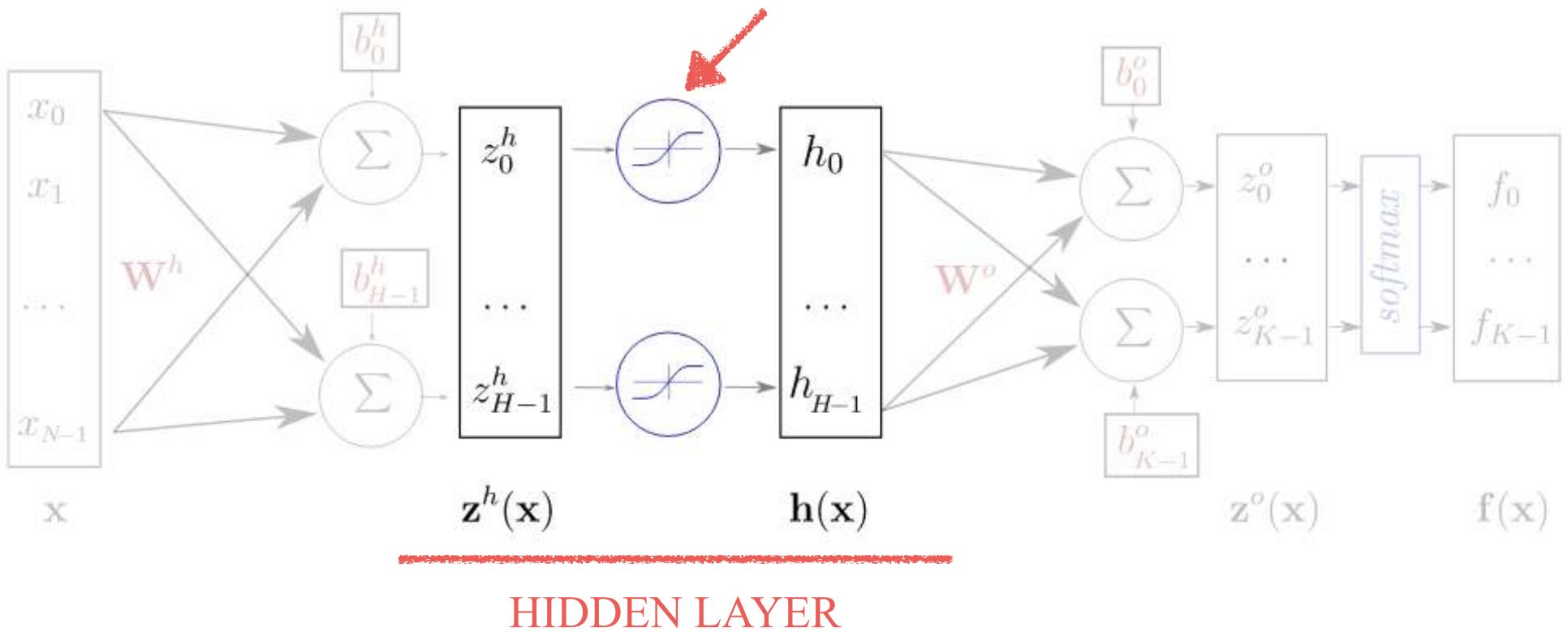
$$f(\vec{x}) = g(\mathbf{W} \cdot \vec{x} + \vec{b})$$

SAME IDEA. NOW **W** becomes a matrix and **b** a vector

# Hidden Layers of Neurons

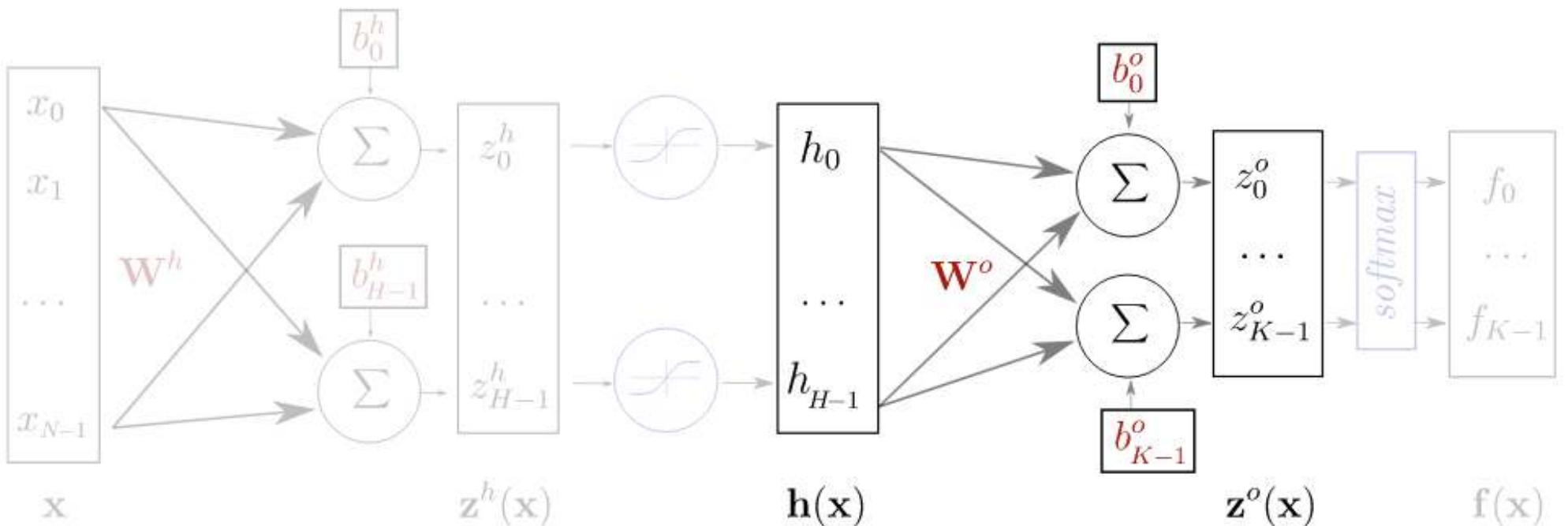


## ACTIVATION FUNCTION



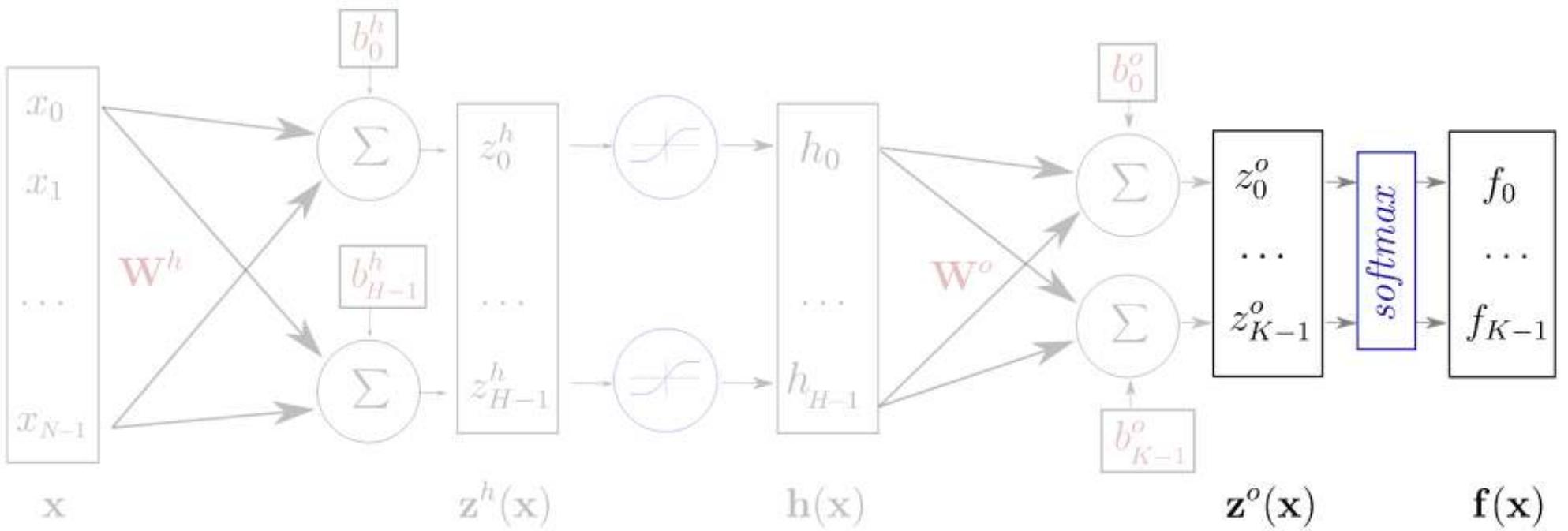
HIDDEN LAYER

$$h(x) = g(z^h(x)) = g(W^h x + b^h)$$



## OUTPUT LAYER

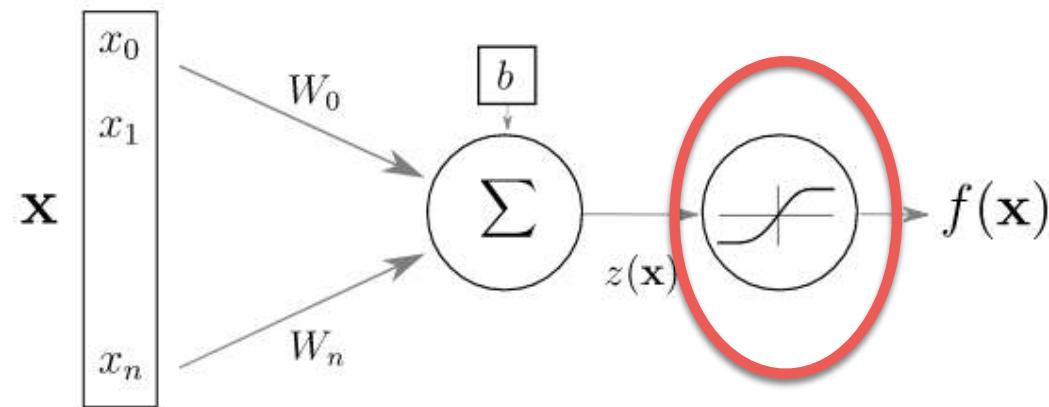
$$z^0(\mathbf{x}) = W^0 h(\mathbf{x}) + b^0$$



PREDICTION LAYER

$$f(\mathbf{x}) = \text{softmax}(\mathbf{z}^0)$$

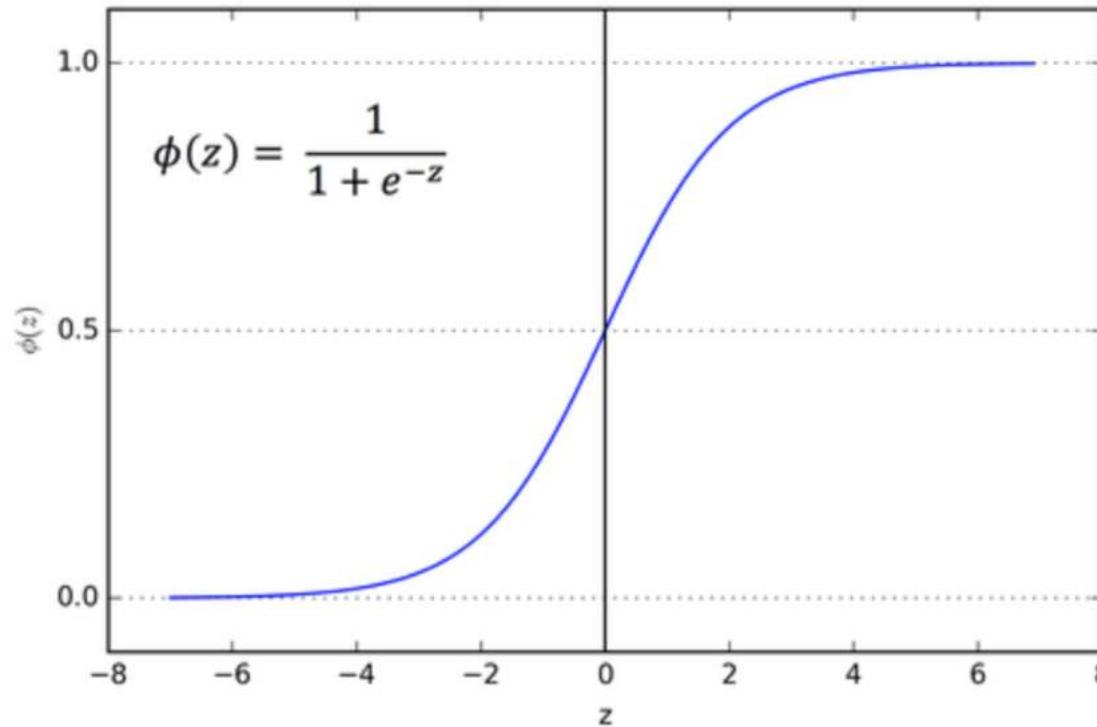
# 1. ACTIVATION FUNCTIONS



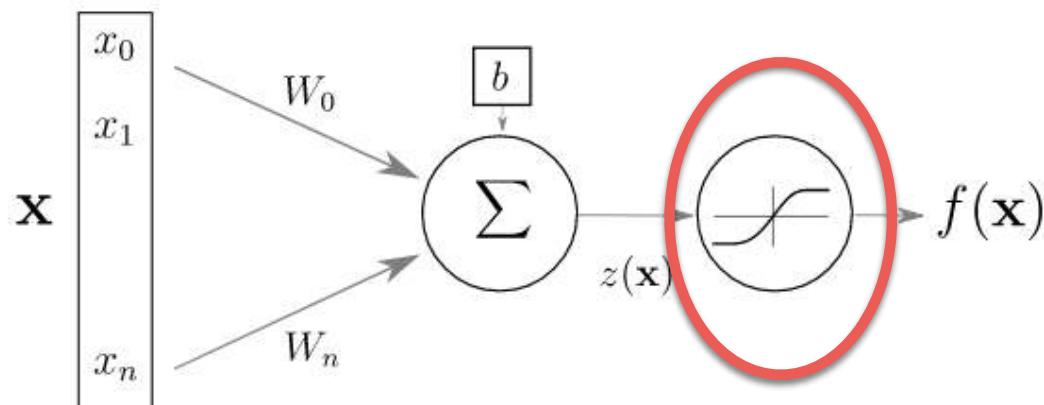
NON TRAINABLE

ADD NON LINEARITIES TO THE PROCESS

# 1. ACTIVATION FUNCTIONS

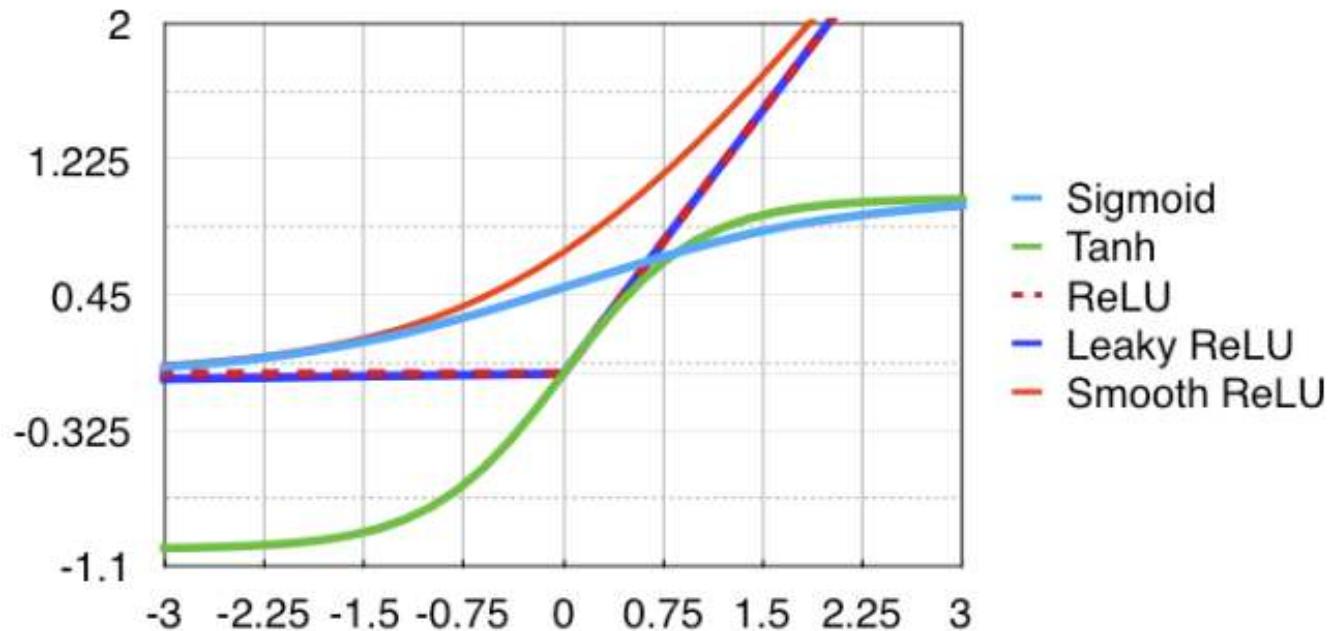


the sigmoid function



ADD NON LINEARITIES TO THE PROCESS

# ACTIVATION FUNCTIONS



Sigmoid:  $f(x) = \frac{1}{1 + e^{-x}}$

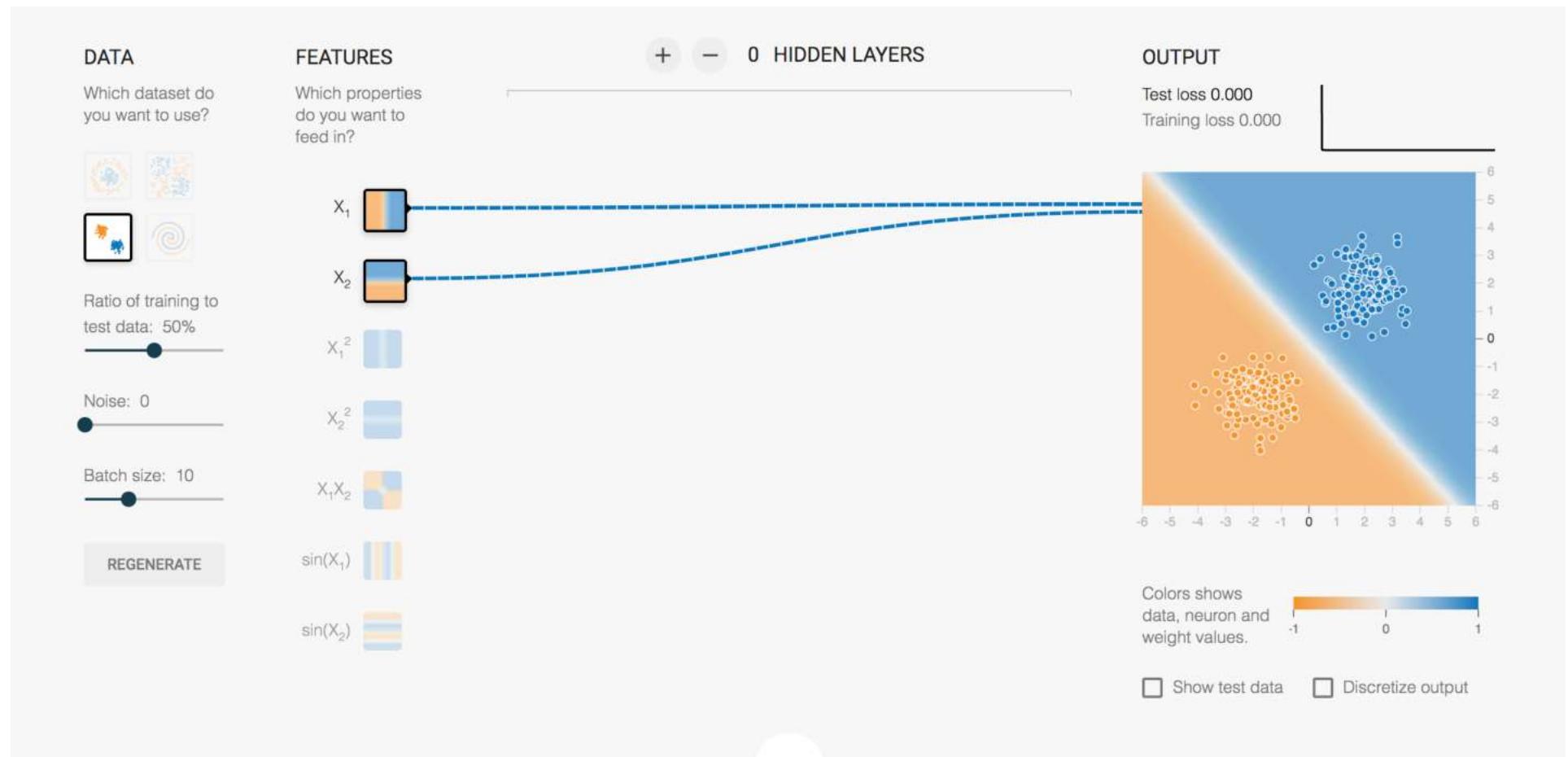
Tanh:  $f(x) = \tanh(x)$

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

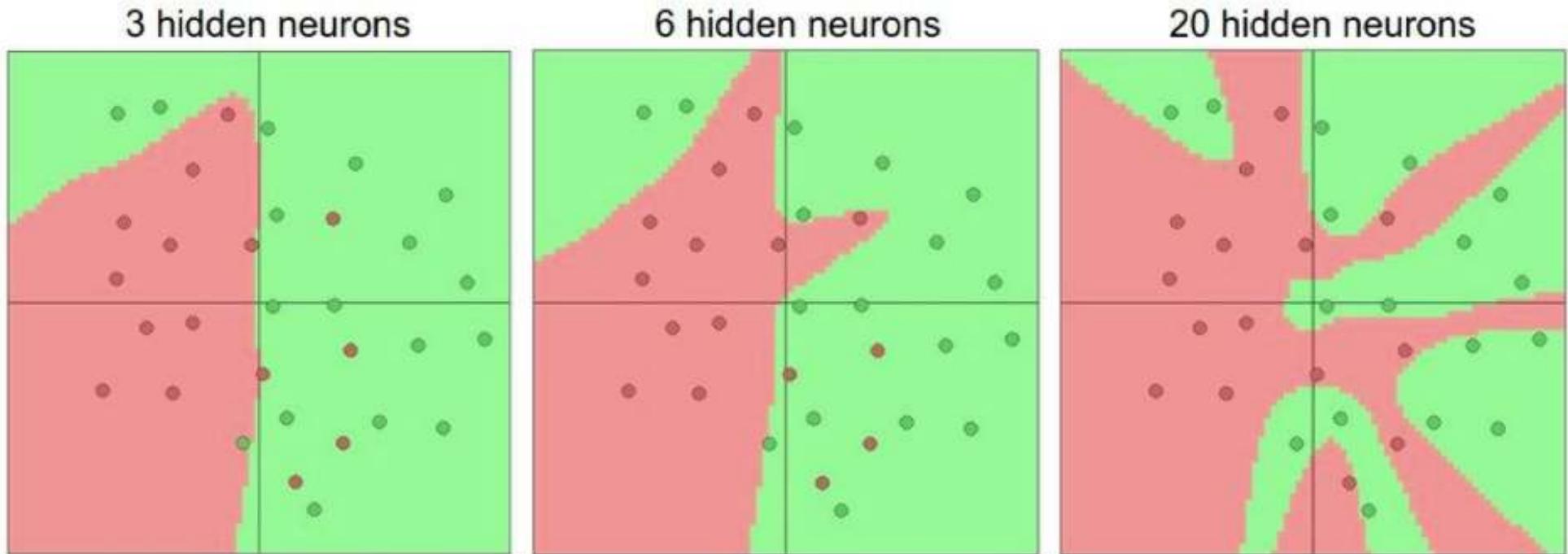
Soft ReLU:  $f(x) = \log(1 + e^x)$

Leaky ReLU:  $f(x) = \epsilon x + (1 - \epsilon) \max(0, x)$

<https://playground.tensorflow.org/>



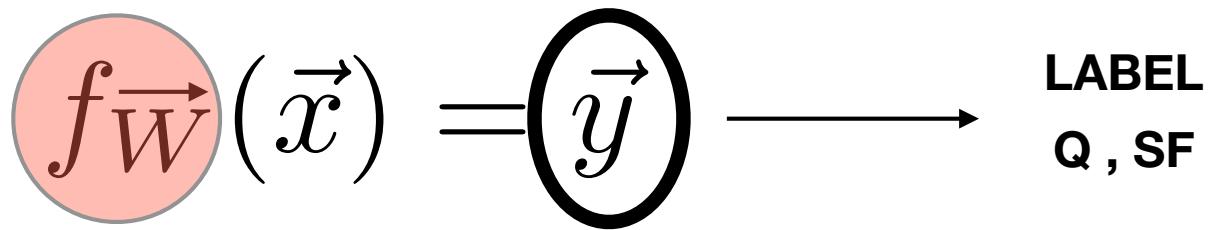
# HIDDEN LAYERS ALLOW INCREASING COMPLEXITY



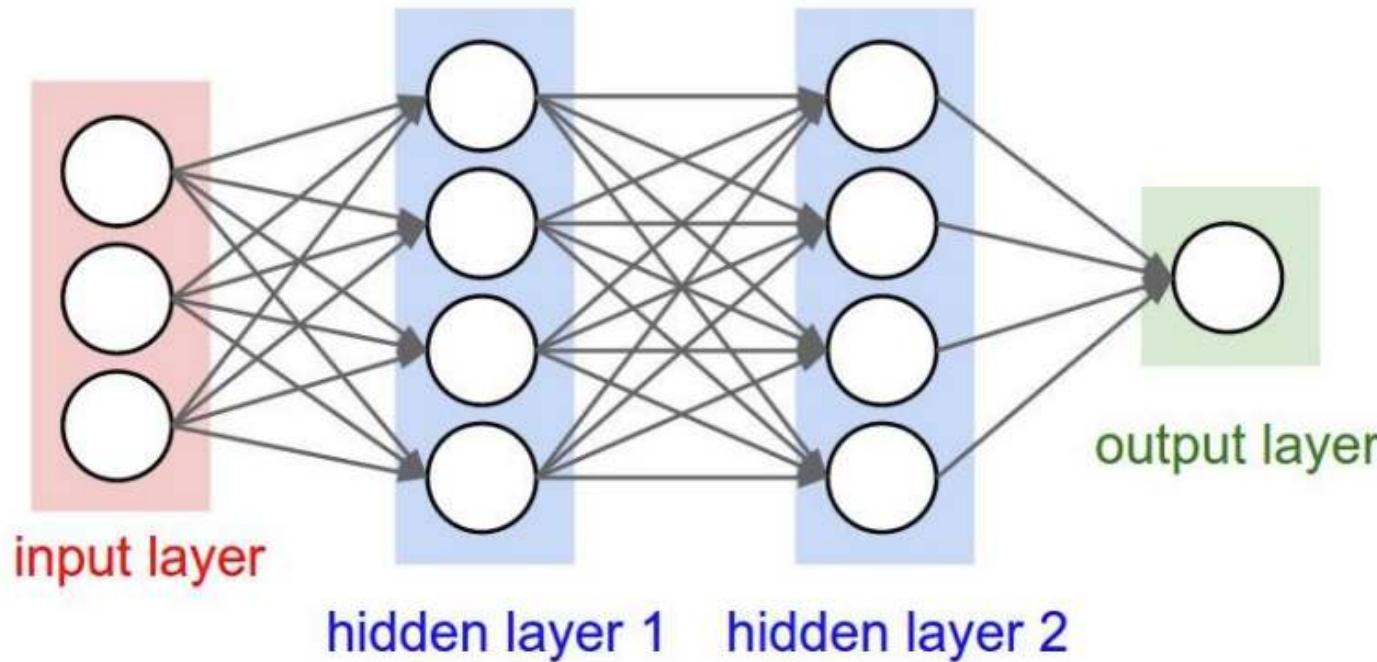
More complex functions allow increasing complexity

Credit: Karpathy

**"CLASSICAL"  
MACHINE LEARNING**



**REPLACE THIS BY A GENERAL  
NON LINEAR FUNCTION WITH SOME PARAMETERS W**



$$f_w(x) = g_3(W_3g_2(W_2g_1(W_1(x)))) \xleftarrow{\text{NETWORK FUNCTION}}$$

# UNIVERSAL APPROXIMATION THEOREM

FOR ANY CONTINUOS FUNCTION FOR A HYPERCUBE  $[0, 1]^d$  TO REAL NUMBERS, AND EVERY POSITIVE EPSILON,  
THERE EXISTS A SIGMOID BASED 1-HIDDEN LAYER  
NEURAL NETWORK THAT OBTAINS AT MOST EPSILON  
ERROR IN FUNCTIONAL SPACE

Cybenko+89

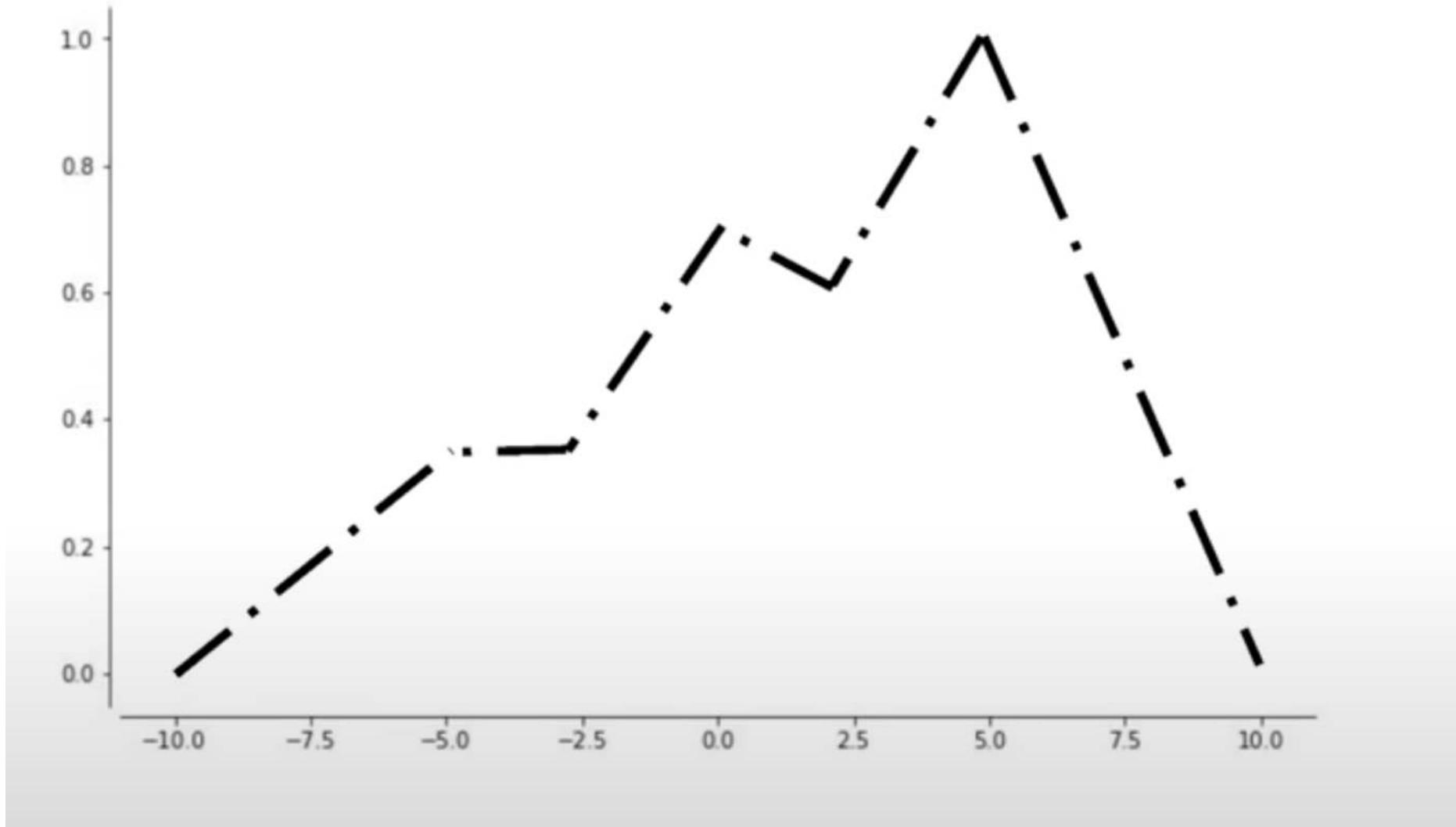
**“BIG ENOUGH NETWORK CAN APPROXIMATE,  
BUT NOT REPRESENT ANY SMOOTH FUNCTION.  
THE MATH DEMONSTRATION IMPLIES SHOWING  
THAT NETWORS ARE DENSE IN THE SPACE OF  
TARGET FUNCTIONS”**

# UNIVERSAL APPROXIMATION THEOREM

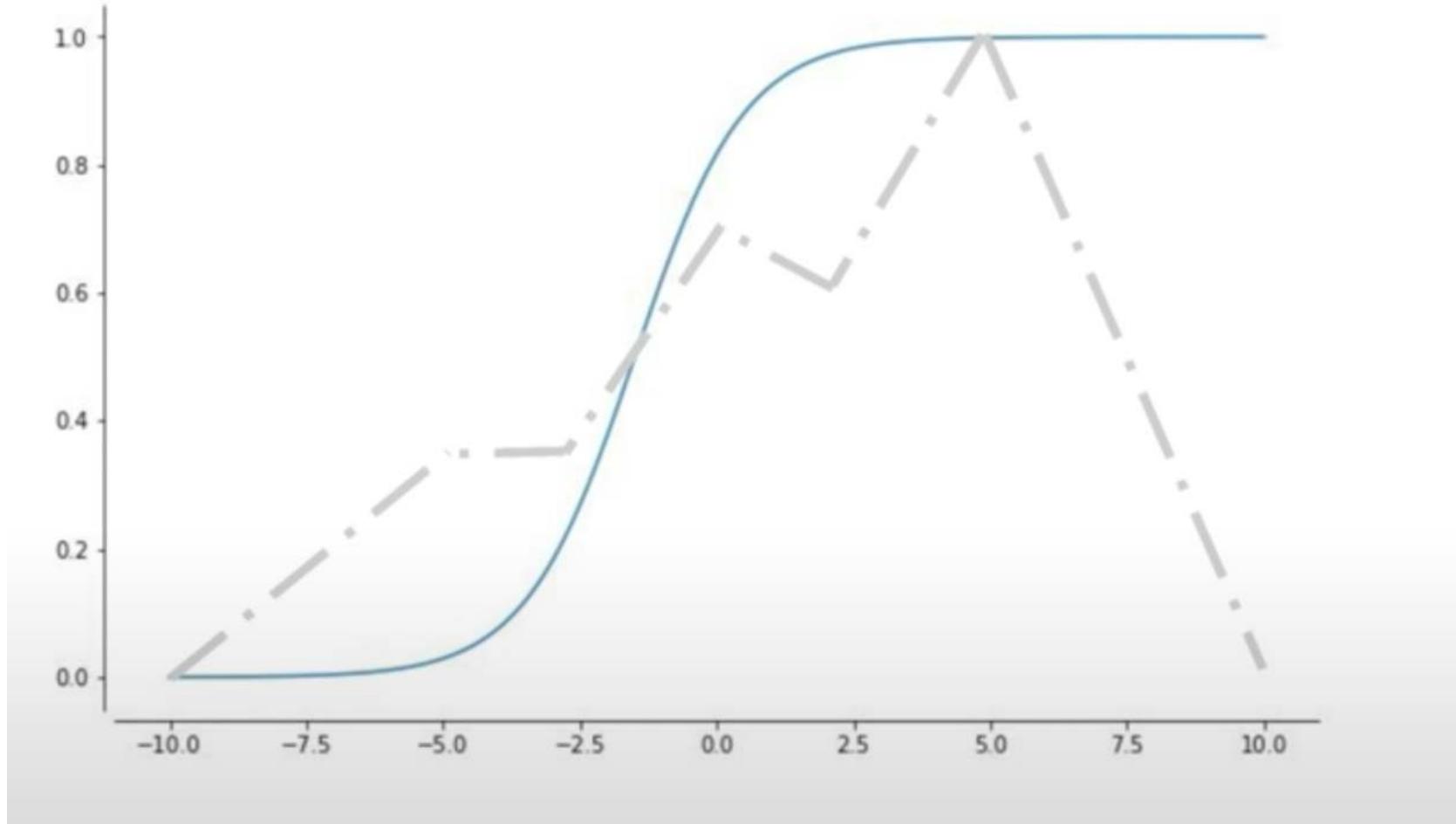
FOR ANY CONTINUOS FUNCTION FOR A HYPERCUBE  $[0,1]^d$  TO REAL NUMBERS, NON-CONSTANT, BOUNDED AND CONTINUOUS ACTIVATION FUNCTION  $f$ , AND EVERY POSITIVE EPSILON, THERE EXISTS A 1-HIDDEN LAYER NEURAL NETWORK USING  $f$  THAT OBTAINS AT MOST EPSILON ERROR IN FUNCTIONAL SPACE

Horváth+91

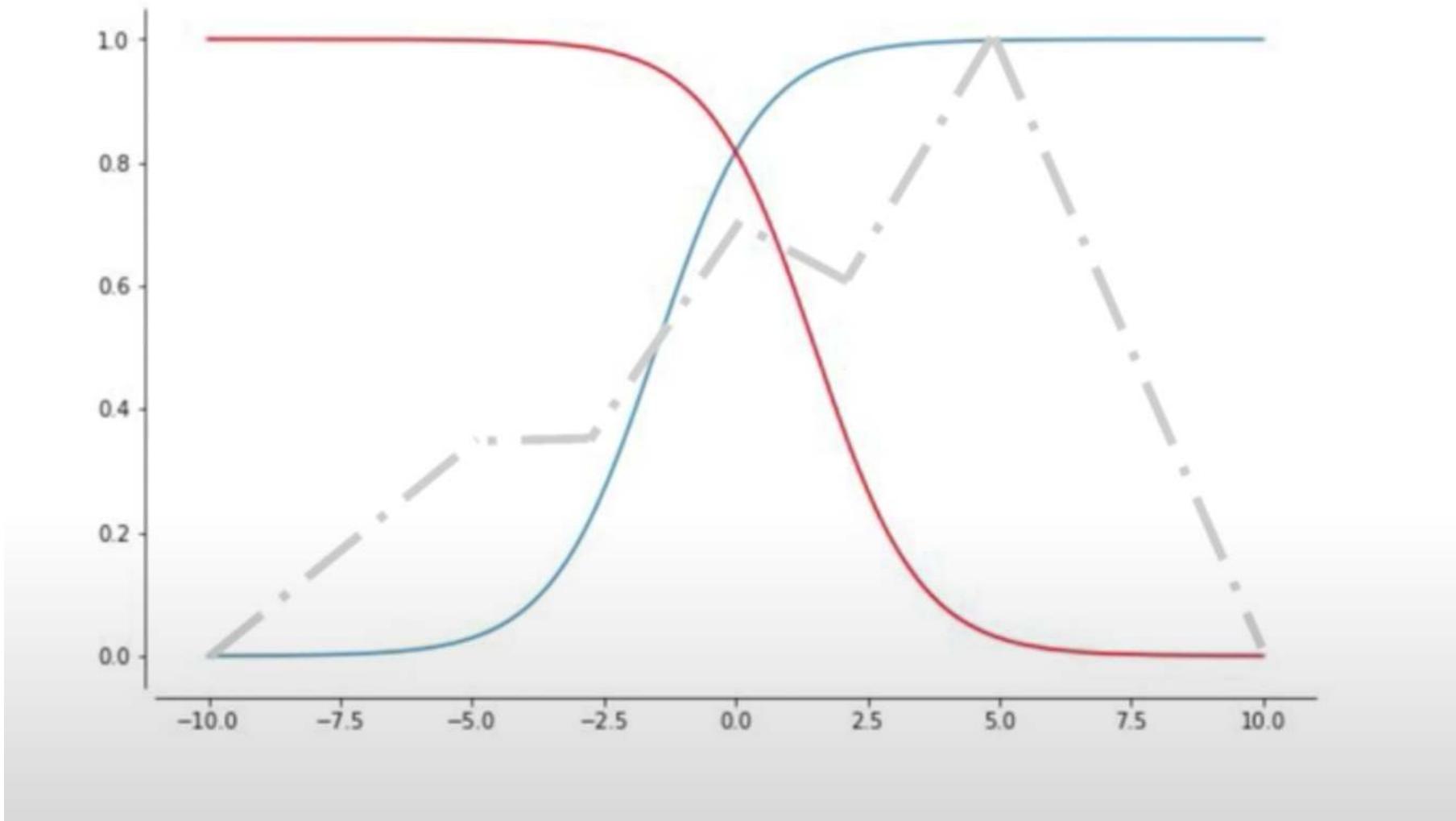
**“BIG ENOUGH NETWORK CAN APPROXIMATE,  
BUT NOT REPRESENT ANY SMOOTH FUNCTION.  
THE MATH DEMONSTRATION IMPLIES SHOWING  
THAT NETWORKS ARE DENSE IN THE SPACE OF  
TARGET FUNCTIONS”**



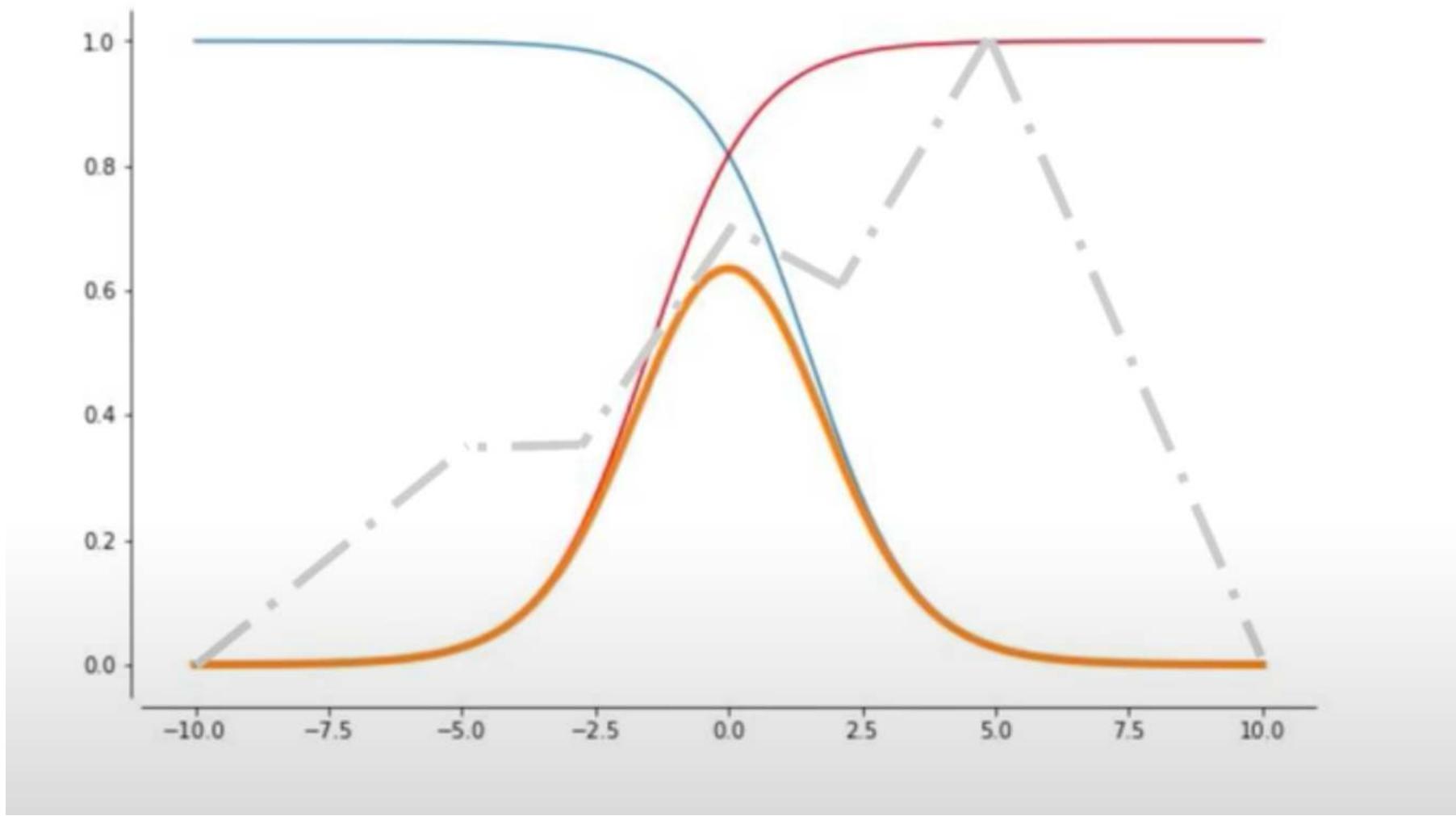
(deepmind  
@Czarnecki)



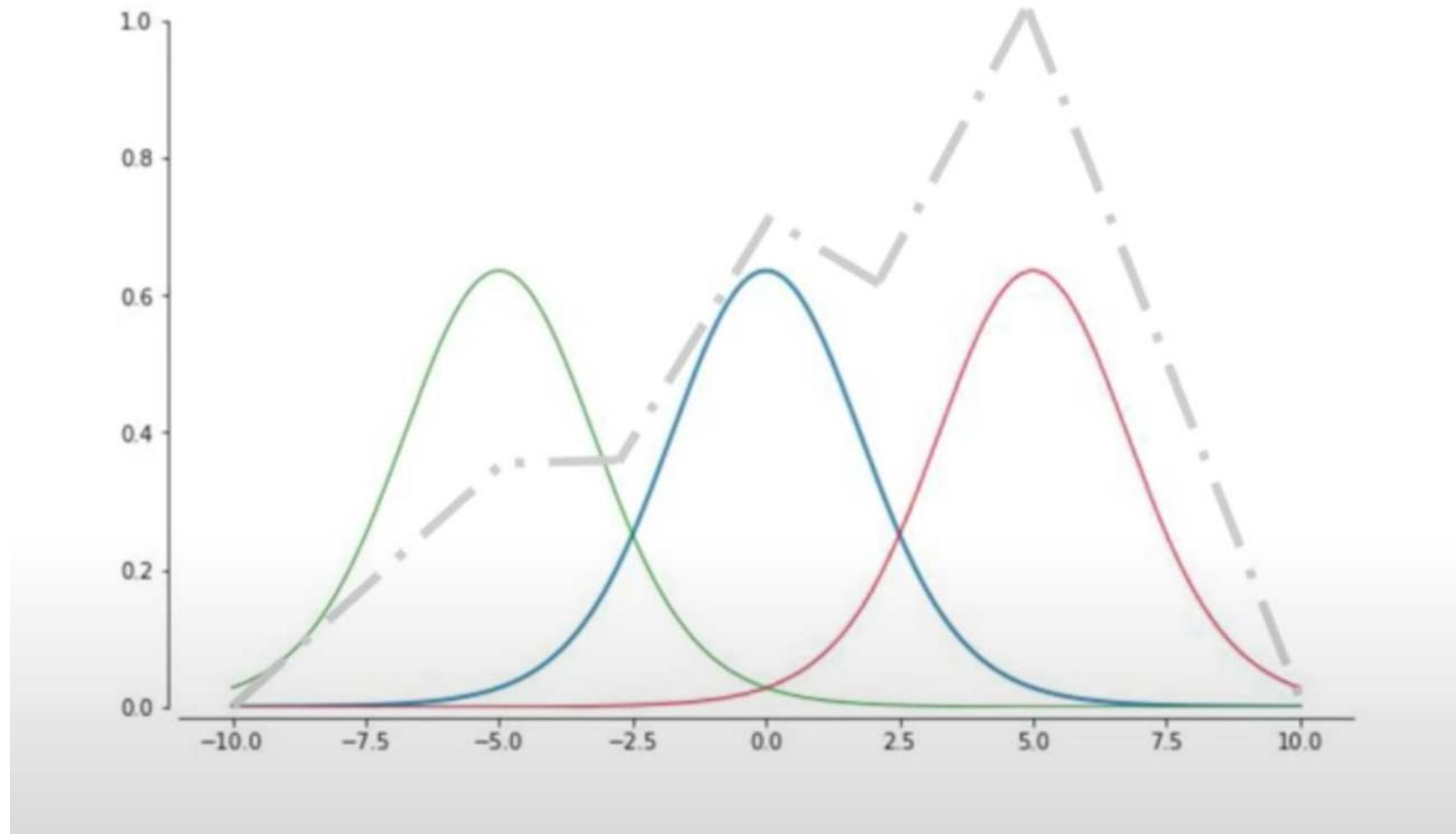
(deepmind  
@Czarnecki)



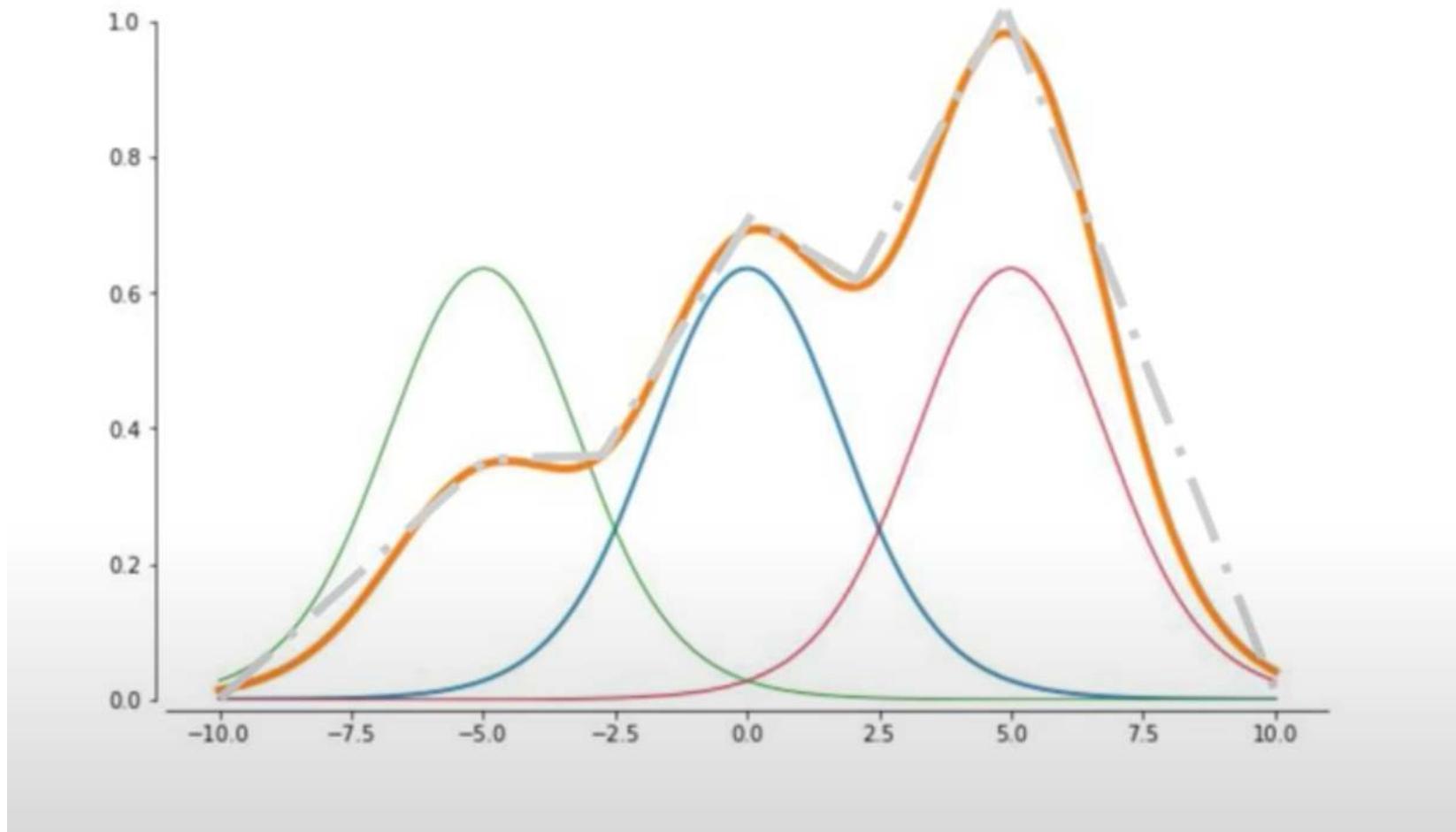
(deepmind  
@Czarnecki)



(deepmind  
@Czarnecki)



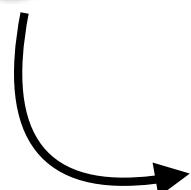
(deepmind  
@Czarnecki)



(deepmind  
@Czarnecki)

# Optimisation of Neural Networks

$$f_W(\vec{x}) = \vec{y}$$



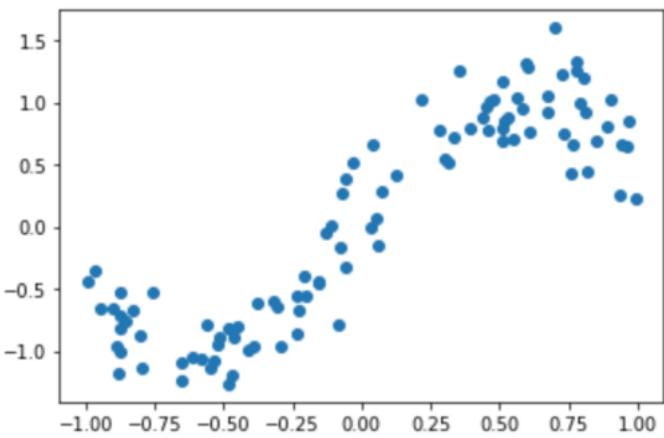
how to find the weights?

# LOSS FUNCTIONS

WE FIRST NEED TO DEFINE THE OBJECTIVE TO MINIMISE:  
THIS IS CALLED THE **LOSS FUNCTION**

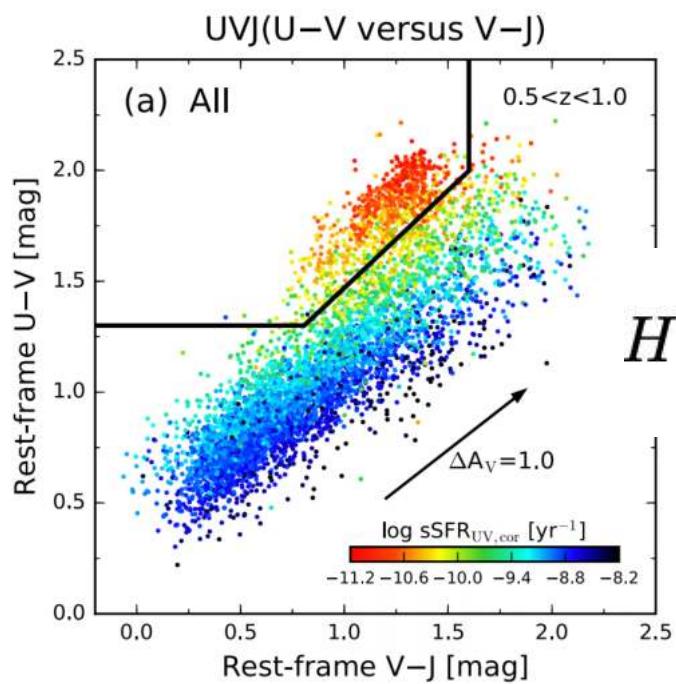
THE CHOICE OF THE LOSS FUNCTION IS CRITICAL AND  
WILL DETERMINE THE BEHAVIOUR OF THE ALGORITHM

IT IS IMPORTANT TO CHOOSE THE LOSS FUNCTION FOR  
YOUR PROBLEM



## Regression

$$MSE = \frac{1}{N} \sum [y_i - f_w(x_i)]^2$$



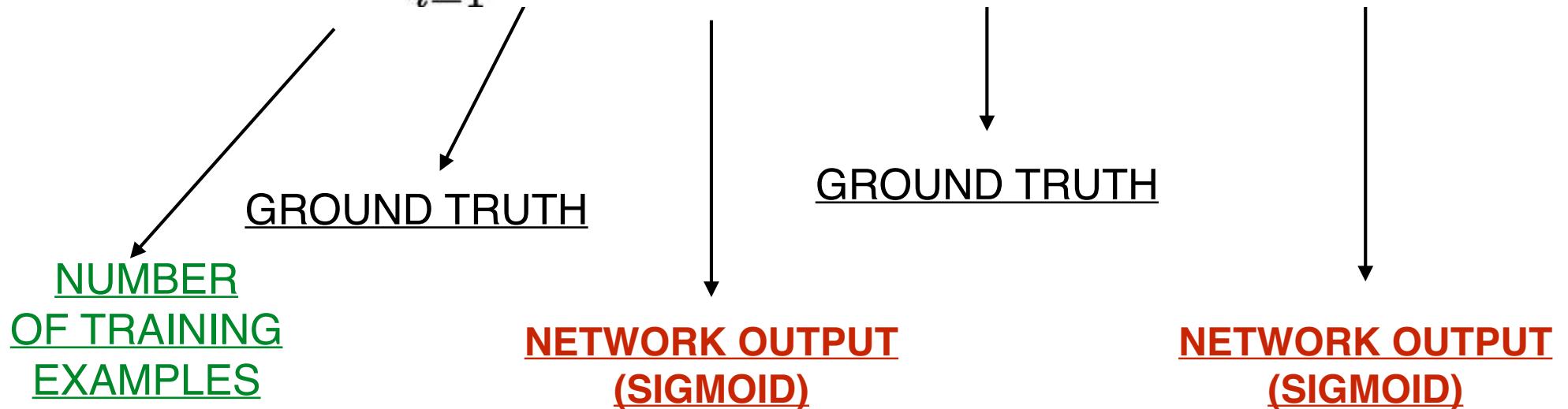
## Classification

$$H = -\frac{1}{N} \sum_{i=1}^N y_i \log(f_w(x_i)) + (1 - y_i) \log(1 - f_w(x_i))$$

# BINARY CLASSIFICATION LOSS

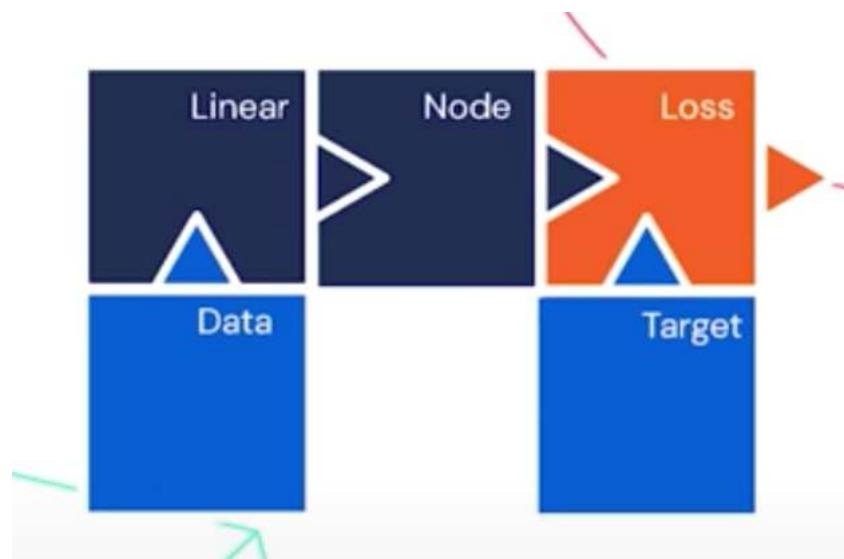
We typically use the binary cross-entropy loss:

$$H = -\frac{1}{N} \sum_{i=1}^N y_i \log(f_w(x_i)) + (1 - y_i) \log(1 - f_w(x_i))$$



OK, SO NOW WE HAVE THE 3 MAIN INGREDIENTS THAT COMPOSE AN ANN:

1. LINEAR NEURON OR UNIT
2. NON LINEARITIES (ACTIVATION FUNCTION)
3. LOSS FUNCTION



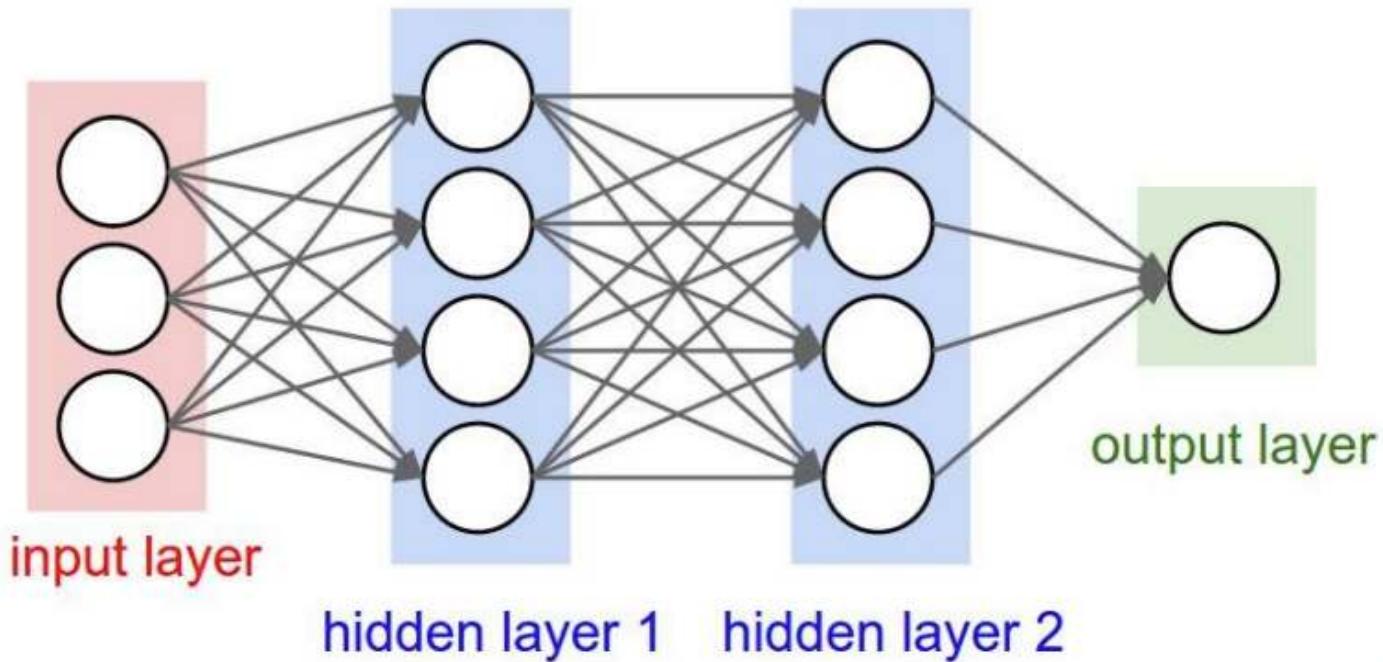
(deepmind  
@Czarnecki)

LET'S SEE WHAT WE CAN DO WITH  
THIS SIMPLE MODEL!

**SO NOW HOW DO WE  
OPTIMIZE THIS?**

# OPTIMIZATION

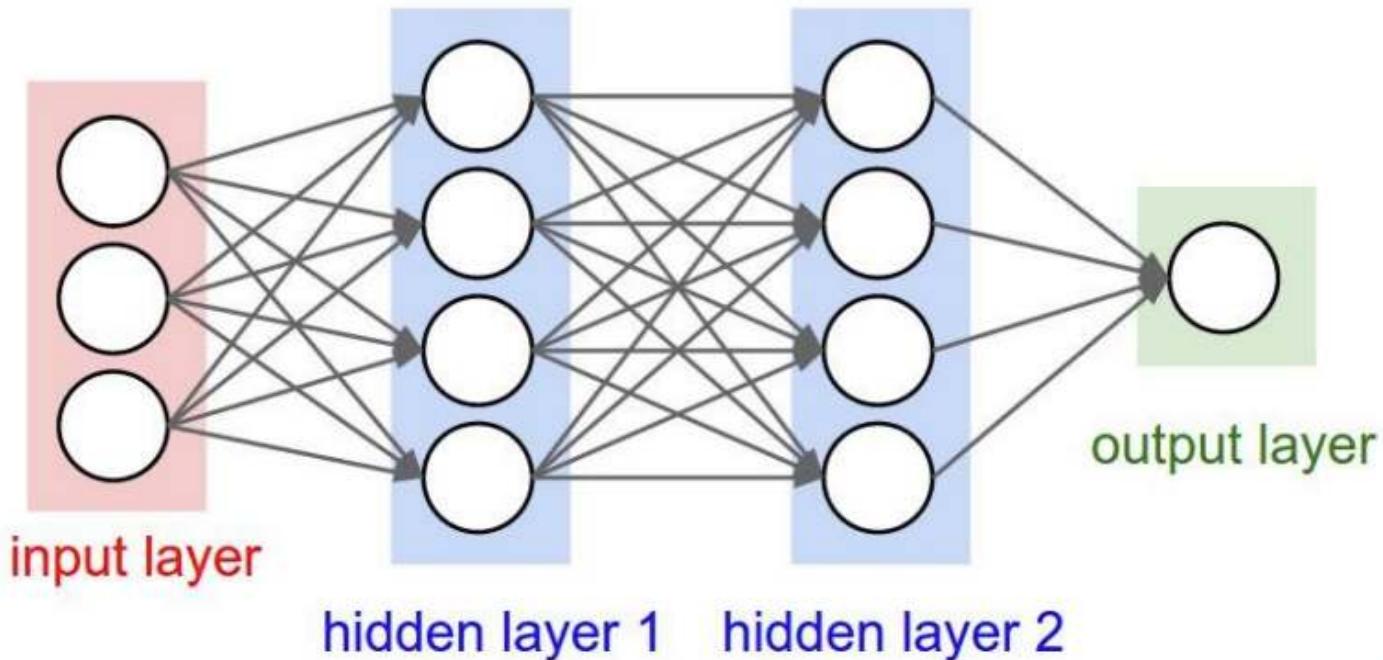
[OR HOW TO FIND THE WEIGHTS?]



$$f_w(x) = g_3(W_3g_2(W_2g_1(W_1(x)))) \xleftarrow{\text{NETWORK FUNCTION}}$$

# OPTIMIZATION

[OR HOW TO FIND THE WEIGHTS?]



$$f_w(x) = g_3(W_3g_2(W_2g_1(W_1(x))))$$

$$MSE = \frac{1}{N} \sum [y_i - f_w(x_i)]^2$$
$$H = -\frac{1}{N} \sum_{i=1}^N y_i \log(f_w(x_i)) + (1 - y_i) \log(1 - f_w(x_i))$$

WE SIMPLY WANT TO MINIMIZE THE LOSS FUNCTION WITH  
RESPECT TO THE WEIGHTS, i.e. FIND THE WEIGHTS THAT  
GENERATE THE MINIMUM LOSS

WE SIMPLY WANT TO MINIMIZE THE LOSS FUNCTION WITH  
RESPECT TO THE WEIGHTS, i.e. FIND THE WEIGHTS THAT  
GENERATE THE MINIMUM LOSS

WE THEN USE STANDARD MINIMIZATION ALGORITHMS  
THAT YOU ALL KNOW...

# FOR EXAMPLE....

Gradient Descent

$$W_{t+1} = W_t - \lambda_t \nabla f(W_t)$$



[gradient]

Newton

$$W_{t+1} = W_t - \lambda [Hf(W_t)]^{-1} \nabla f(W_t)$$



[hessian]

NEWTON CONVERGES FASTER...

# FOR EXAMPLE....

## Gradient Descent

$$W_{t+1} = W_t - \lambda_t \nabla f(W_t)$$



[gradient]

## Newton

$$W_{t+1} = W_t - \lambda [Hf(W_t)]^{-1} \nabla f(W_t)$$



[hessian]

NEWTON CONVERGES FASTER...

**BUT NEEDS THE HESSIAN**

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial W_1^2} & \frac{\partial^2 f}{\partial W_1 \partial W_2} & \dots & \frac{\partial^2 f}{\partial W_1 \partial W_n} \\ \frac{\partial^2 f}{\partial W_2 \partial W_1} & \frac{\partial^2 f}{\partial W_2^2} & \dots & \frac{\partial^2 f}{\partial W_2 \partial W_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial W_n \partial W_1} & \frac{\partial^2 f}{\partial W_n \partial W_2} & \dots & \frac{\partial^2 f}{\partial W_n^2} \end{bmatrix}$$

# FOR EXAMPLE....

## Gradient Descent

$$W_{t+1} = W_t - \lambda_t \nabla f(W_t)$$



[gradient]

## Newton

$$W_{t+1} = W_t - \lambda [Hf(W_t)]^{-1} \nabla f(W_t)$$

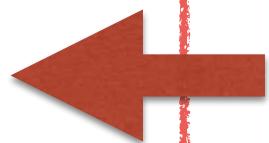


[hessian]

NEWTON CONVERGES FASTER...

**BUT NEEDS THE HESSIAN**

MOST USED BY FAR....



$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial W_1^2} & \frac{\partial^2 f}{\partial W_1 \partial W_2} & \dots & \frac{\partial^2 f}{\partial W_1 \partial W_n} \\ \frac{\partial^2 f}{\partial W_2 \partial W_1} & \frac{\partial^2 f}{\partial W_2^2} & \dots & \frac{\partial^2 f}{\partial W_2 \partial W_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial W_n \partial W_1} & \frac{\partial^2 f}{\partial W_n \partial W_2} & \dots & \frac{\partial^2 f}{\partial W_n^2} \end{bmatrix}$$

# FOR EXAMPLE....

## Gradient Descent

$$W_{t+1} = W_t - \lambda_t \nabla f(W_t)$$



[gradient]

EVERYTHING RELIES  
ON COMPUTING THE GRADIENT

MOST USED BY FAR....

## Newton

$$W_{t+1} = W_t - \lambda [Hf(W_t)]^{-1} \nabla f(W_t)$$



[hessian]

NEWTON CONVERGES FASTER...

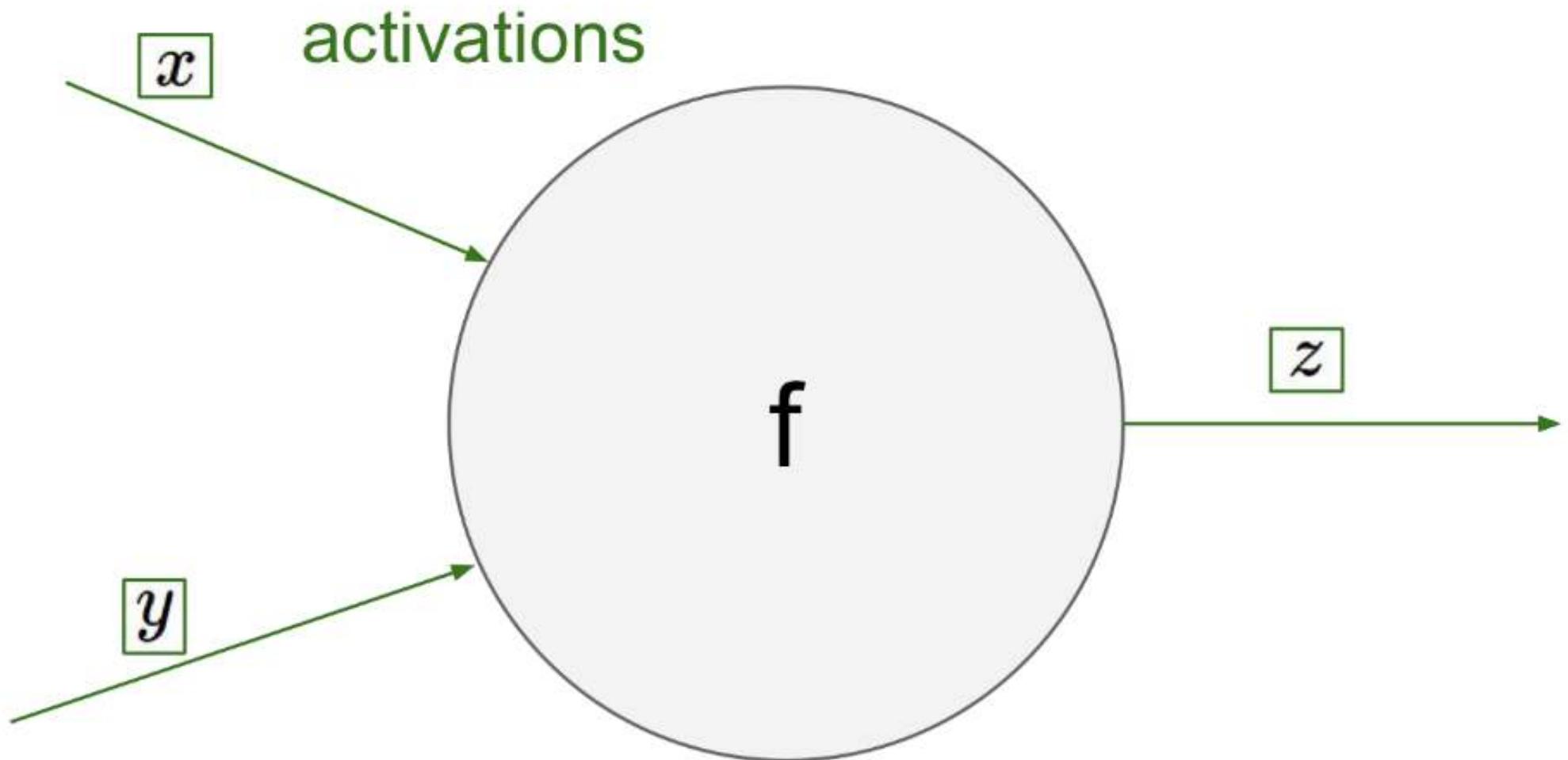
BUT NEEDS THE HESSIAN

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial W_1^2} & \frac{\partial^2 f}{\partial W_1 \partial W_2} & \dots & \frac{\partial^2 f}{\partial W_1 \partial W_n} \\ \frac{\partial^2 f}{\partial W_2 \partial W_1} & \frac{\partial^2 f}{\partial W_2^2} & \dots & \frac{\partial^2 f}{\partial W_2 \partial W_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial W_n \partial W_1} & \frac{\partial^2 f}{\partial W_n \partial W_2} & \dots & \frac{\partial^2 f}{\partial W_n^2} \end{bmatrix}$$

NICE, BUT I NEED TO COMPUTE THE  
GRADIENT AT EVERY ITERATION OF  
AN ARBITRARY COMPLEX FUNCTION!

# BACKPROPAGATION

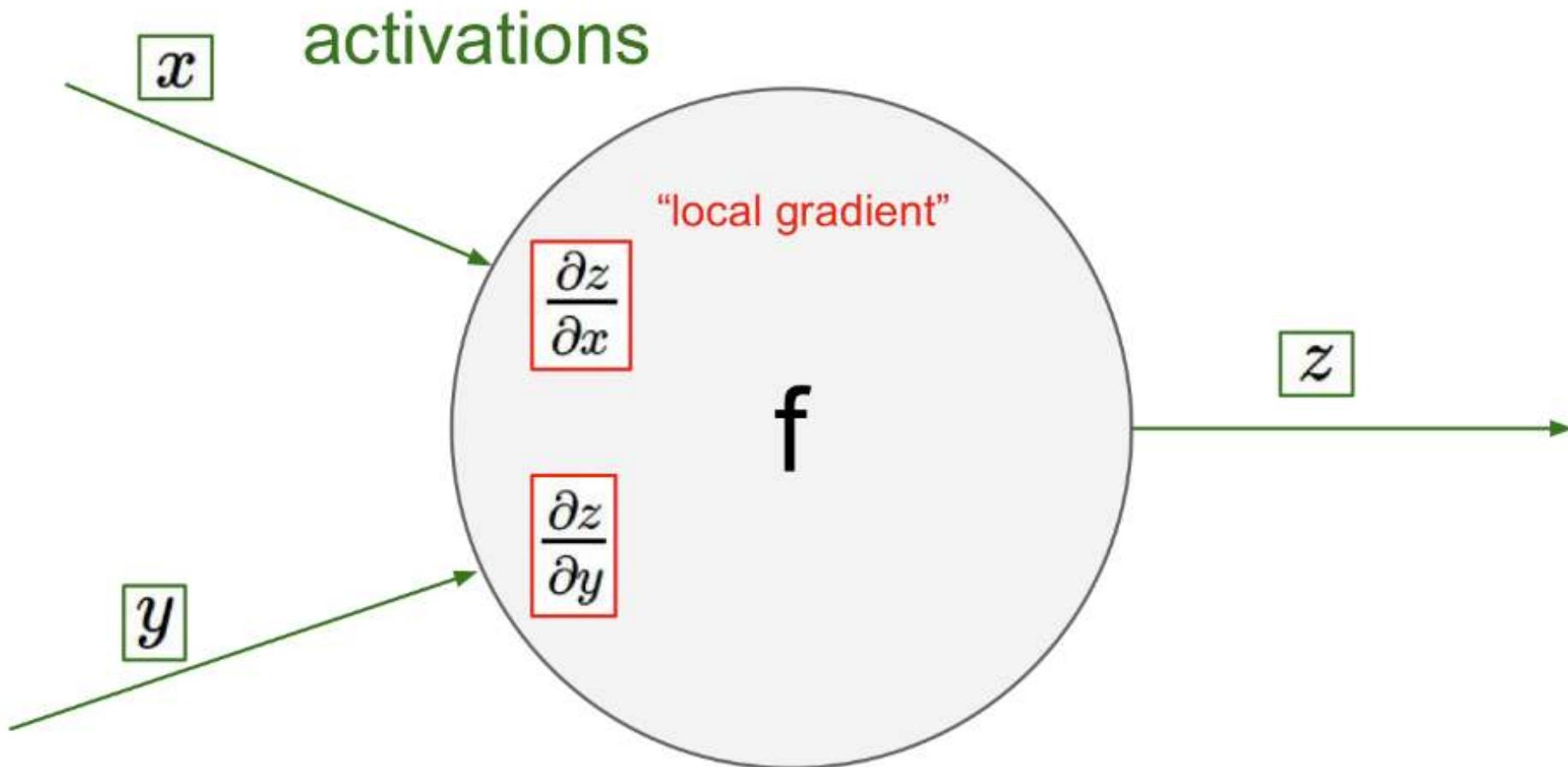
[AT THE NEURON LEVEL]



Credit: A. Karpathy

# BACKPROPAGATION

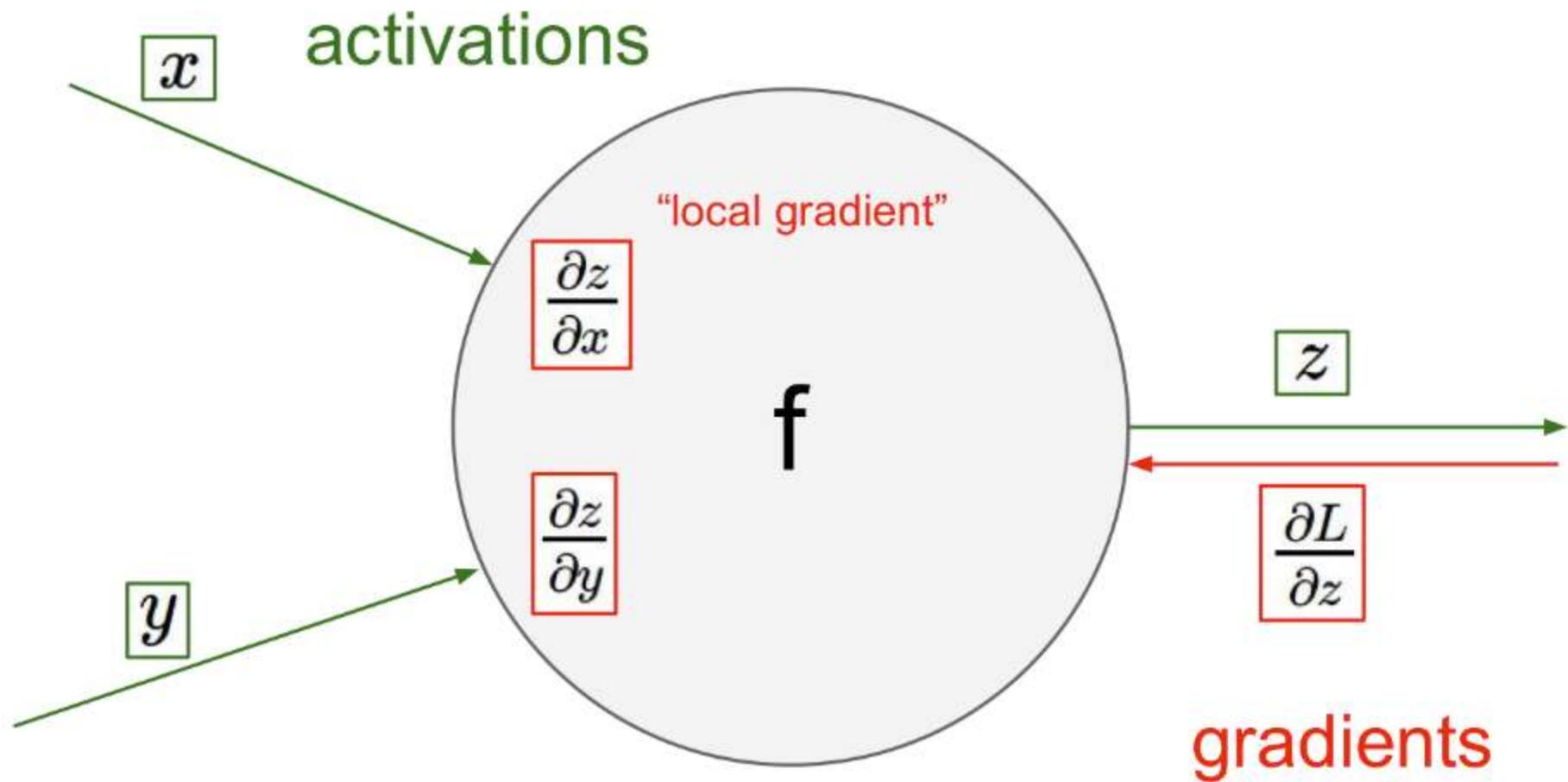
[AT THE NEURON LEVEL]



Credit: A. Karpathy

# BACKPROPAGATION

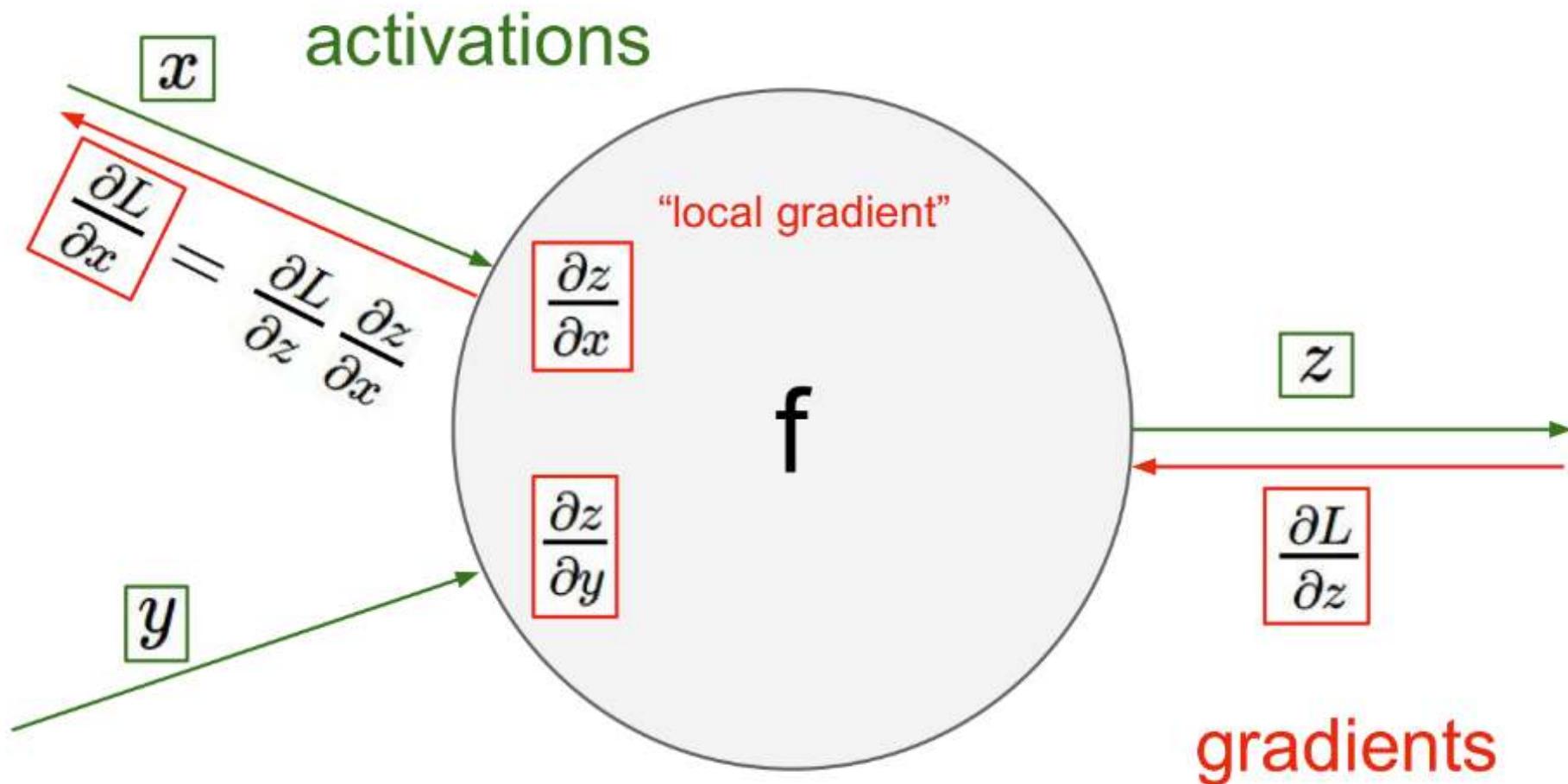
[AT THE NEURON LEVEL]



Credit: A. Karpathy

# BACKPROPAGATION

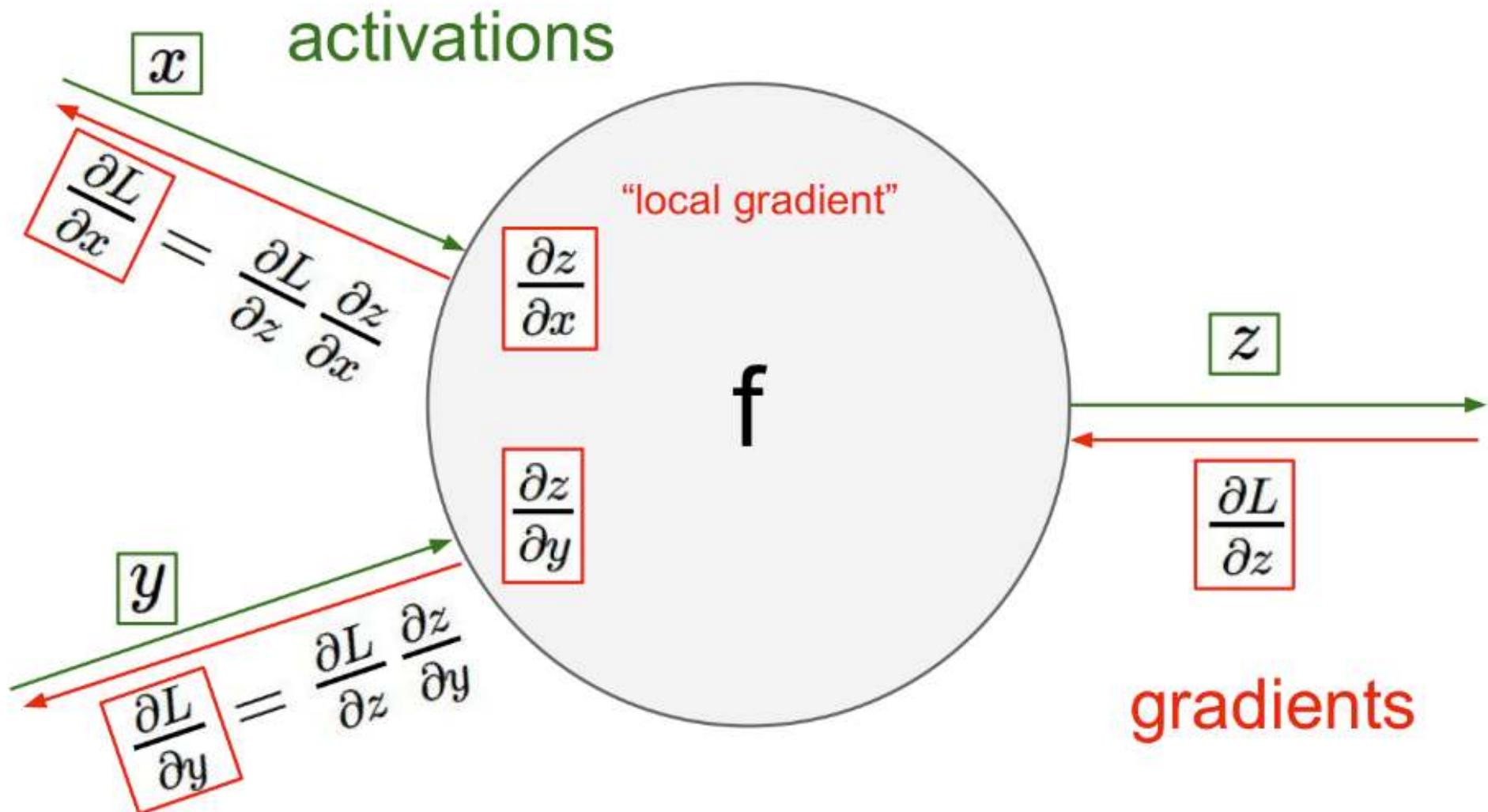
[AT THE NEURON LEVEL]



Credit: A. Karpathy

# BACKPROPAGATION

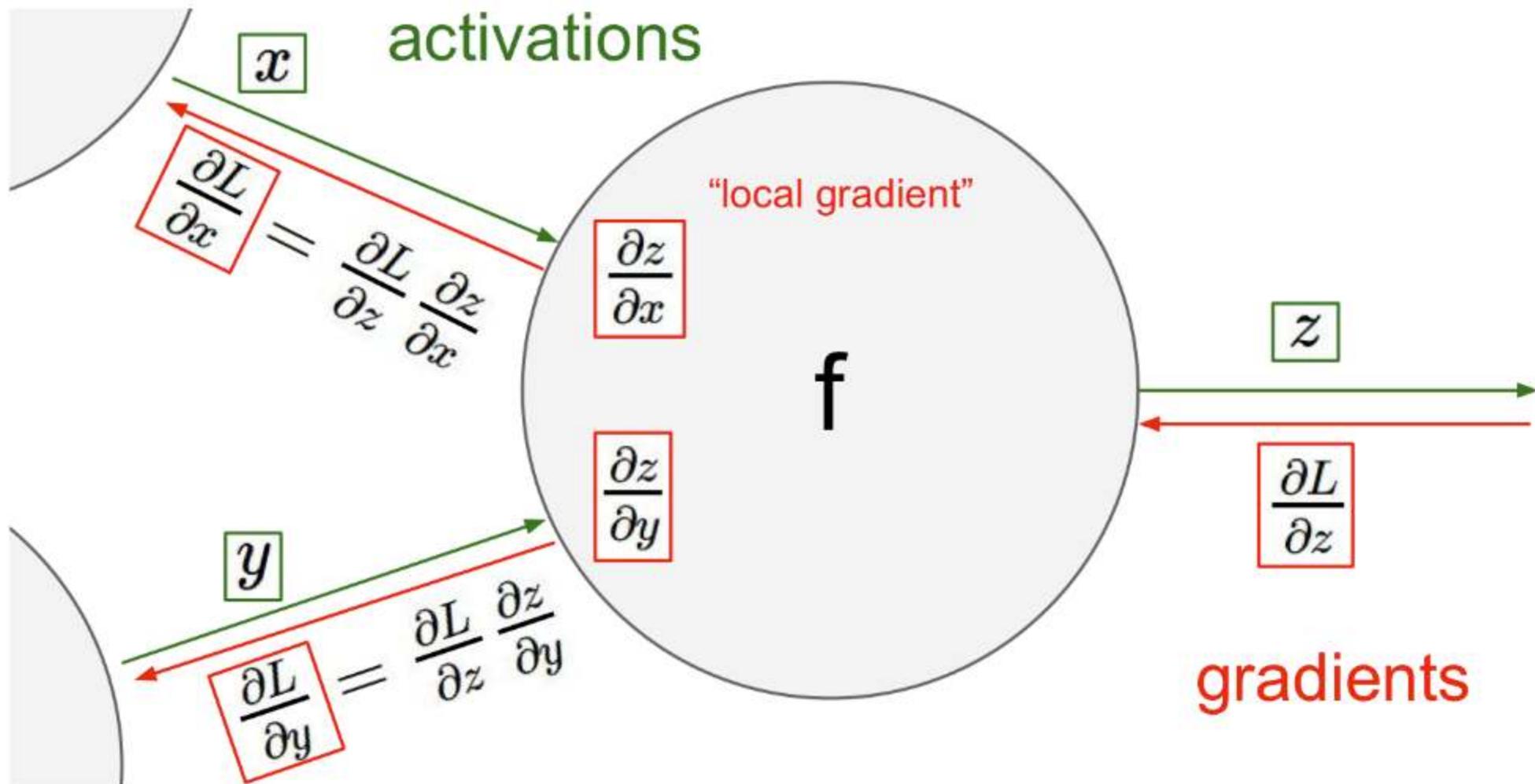
[AT THE NEURON LEVEL]



Credit: A. Karpathy

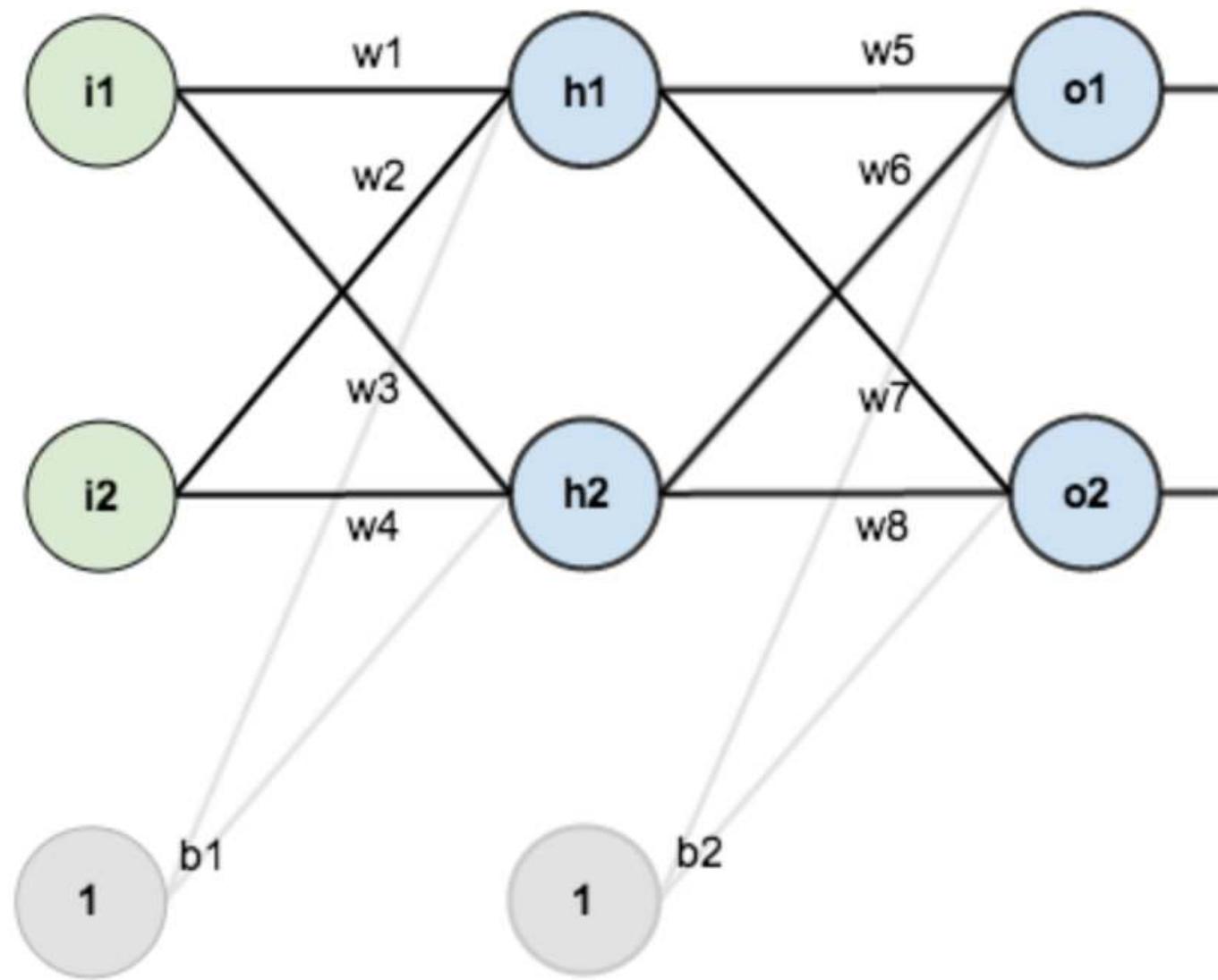
# BACKPROPAGATION

[AT THE NEURON LEVEL]

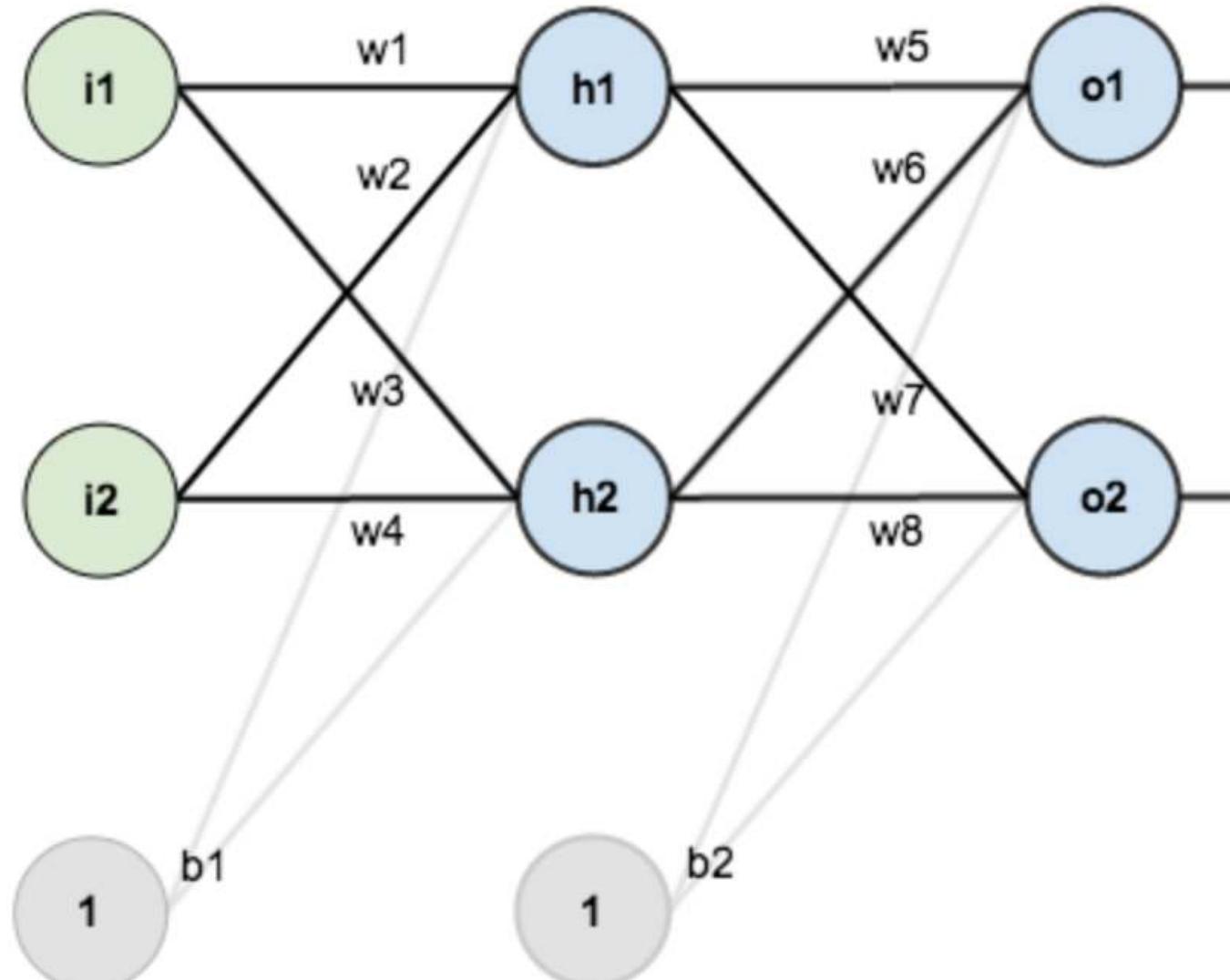


Credit: A. Karpathy

**LET'S FOLLOW A NETWORK  
WHILE IT LEARNS...**

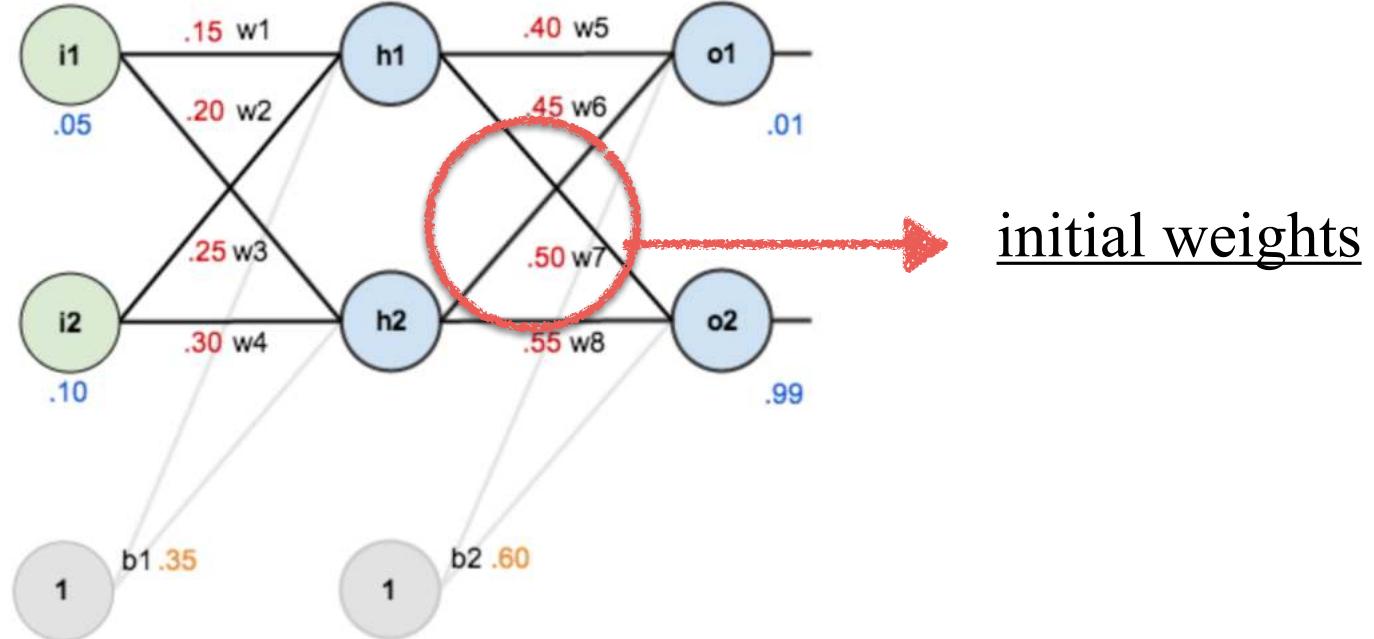


EXAMPLE TAKEN FROM HERE



LET'S ASSUME A VERY SIMPLE TRAINING SET:  
 $X=(0.05, 0.10) \rightarrow Y=(0.01, 0.99)$

EXAMPLE TAKEN FROM HERE

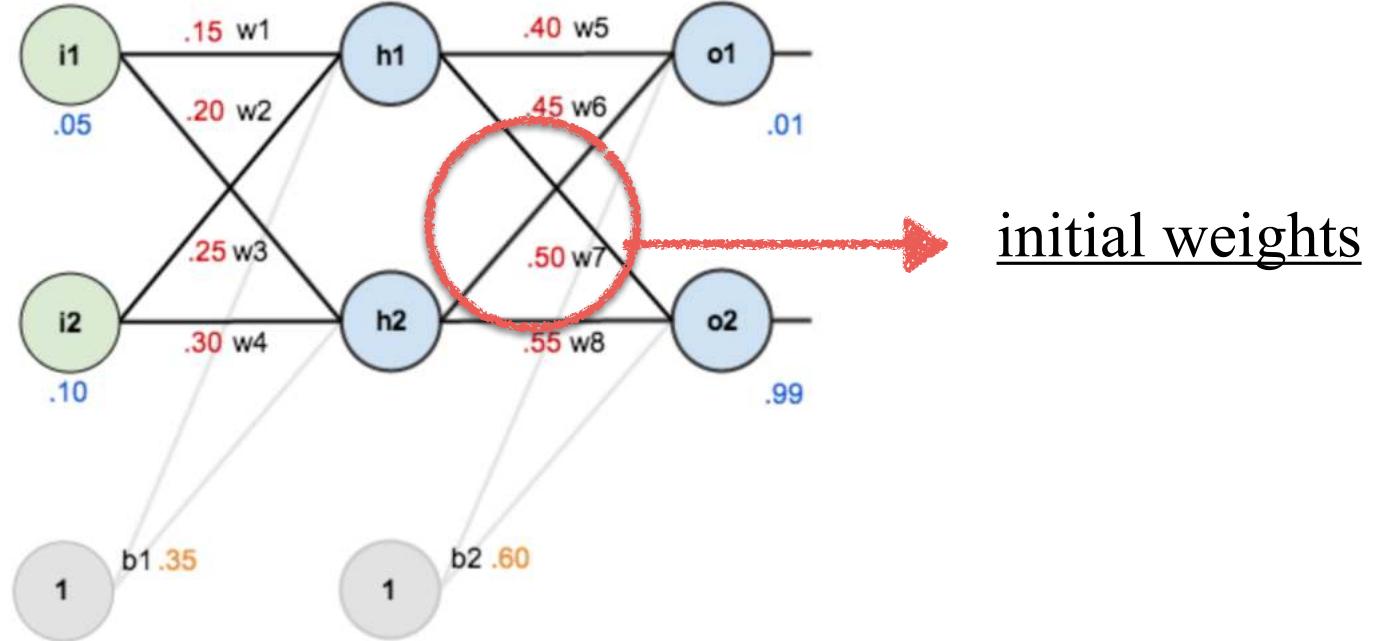


## 1. THE FORWARD PASS

$$in_{h1} = w_1 i_1 + w_2 i_2 + b_1$$

$$in_{h1} = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 = 0.3775$$

[with initial weights]



## 1. THE FORWARD PASS

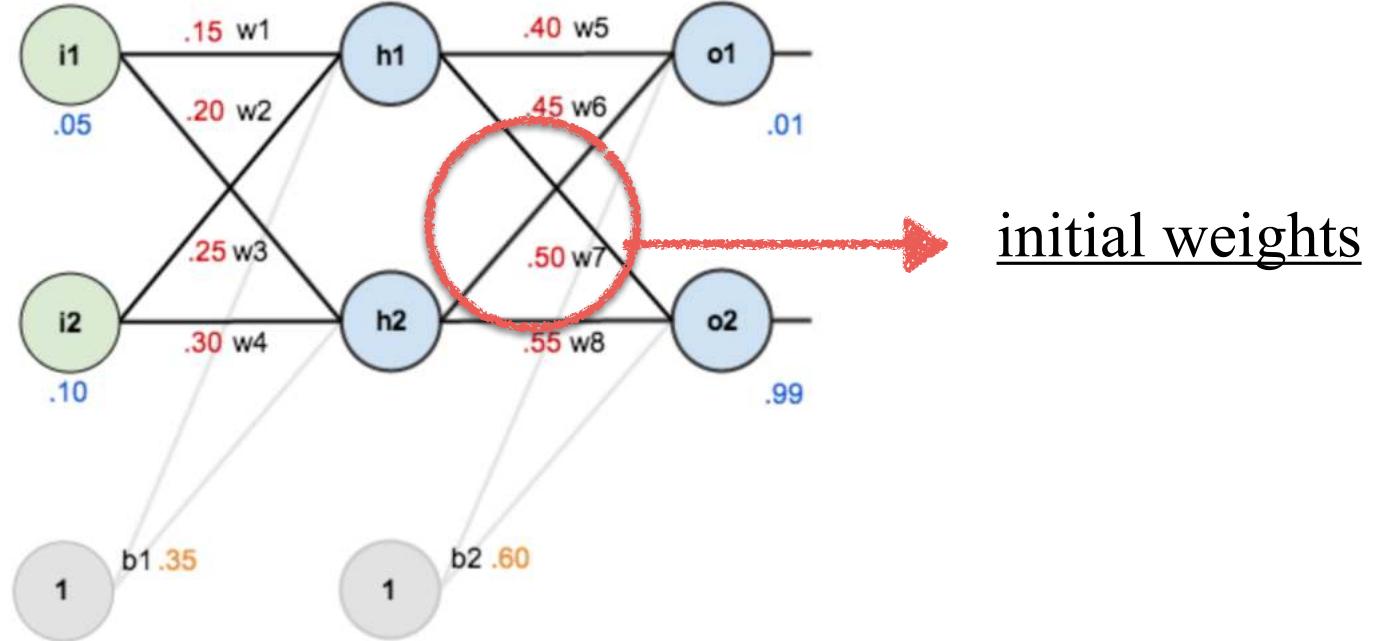
$$in_{h1} = w_1 i_1 + w_2 i_2 + b_1$$

$$in_{h1} = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 = 0.3775$$

[with initial weights]

$$out_{h1} = \frac{1}{1 + e^{-in_{h1}}} = 0.5932$$

[after the activation function]



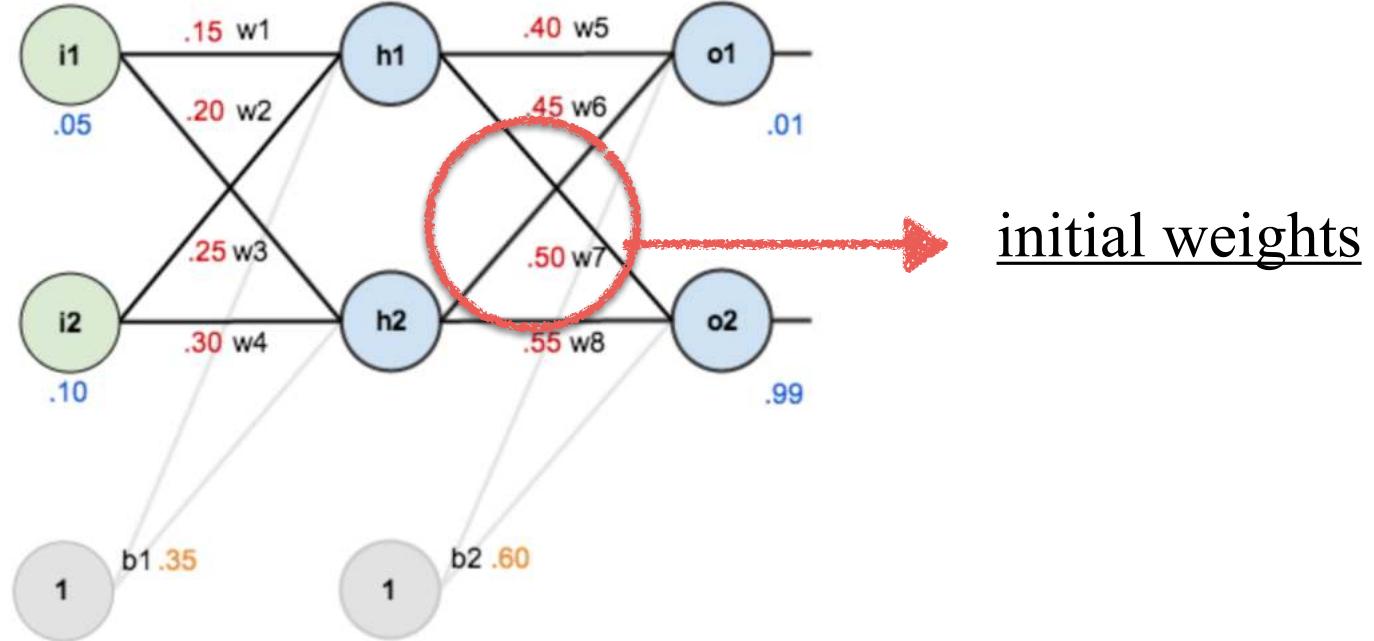
## 1. THE FORWARD PASS

WE CONTINUE TO  $o_1$

$$in_{o1} = w_5 out_{h1} + w_6 out_{h2} + b_2$$

$$in_{o1} = 0.4 \times 0.593 + 0.45 \times 0.596 + 0.6 = 1.105$$

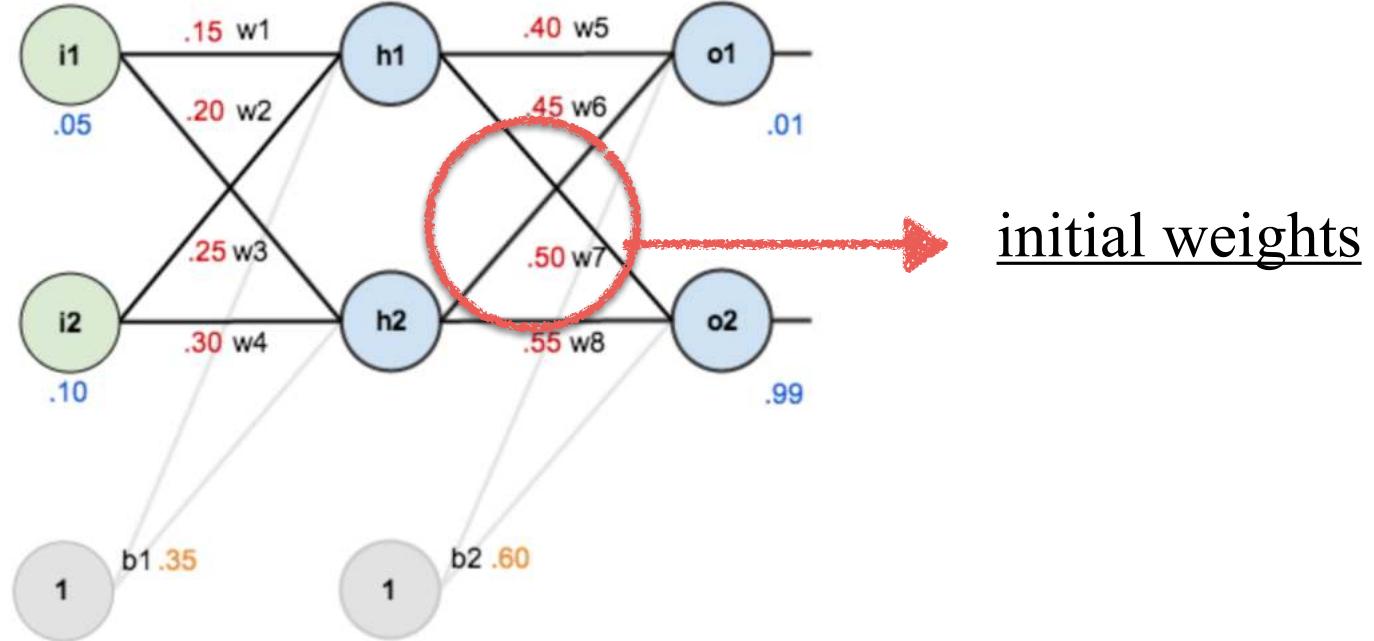
$$out_{o1} = \frac{1}{1 + e^{-1.105}} = 0.751$$



## **1. THE FORWARD PASS**

AND THE SAME FOR  $o_2$

$$out_{o2} = 0.7729$$

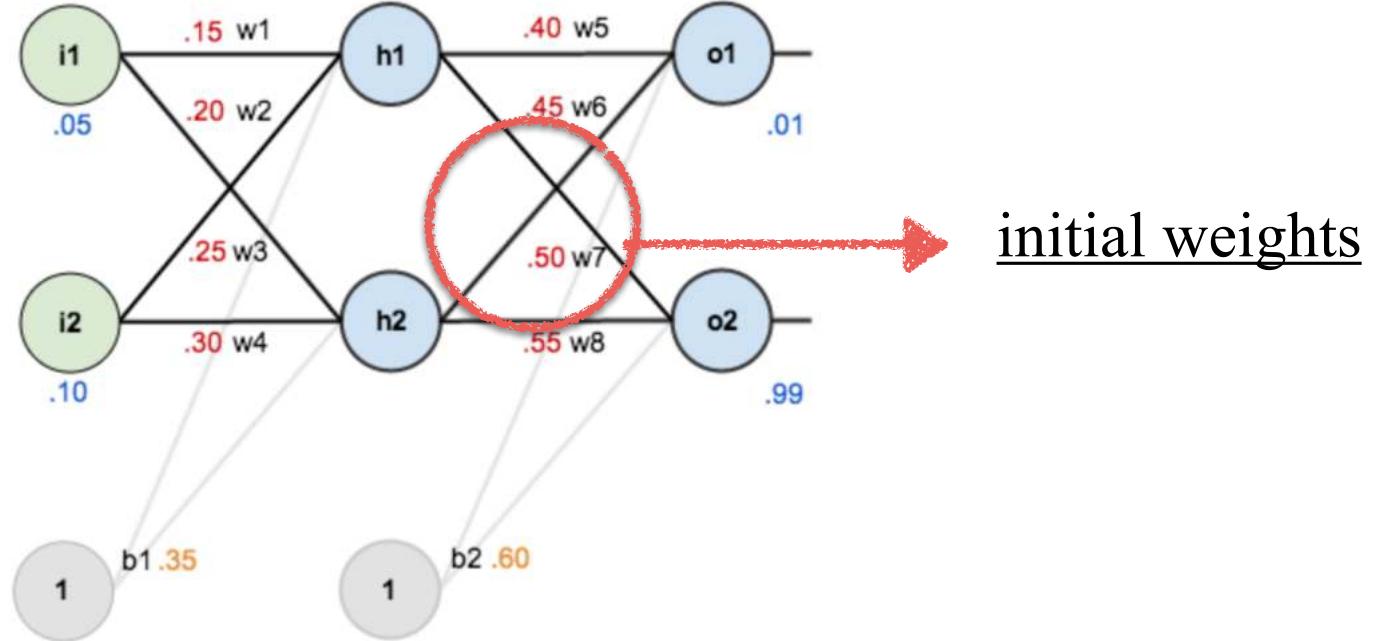


## 2. THE LOSS FUNCTION

$$L_{total} = \sum 0.5(target - output)^2$$

$$L_{o1} = 0.5(target_{o1} - output_{o1})^2 = 0.5 \times (0.01 - 0.751)^2 = 0.274$$

$$L_{o2} = 0.023$$



## 2. THE LOSS FUNCTION

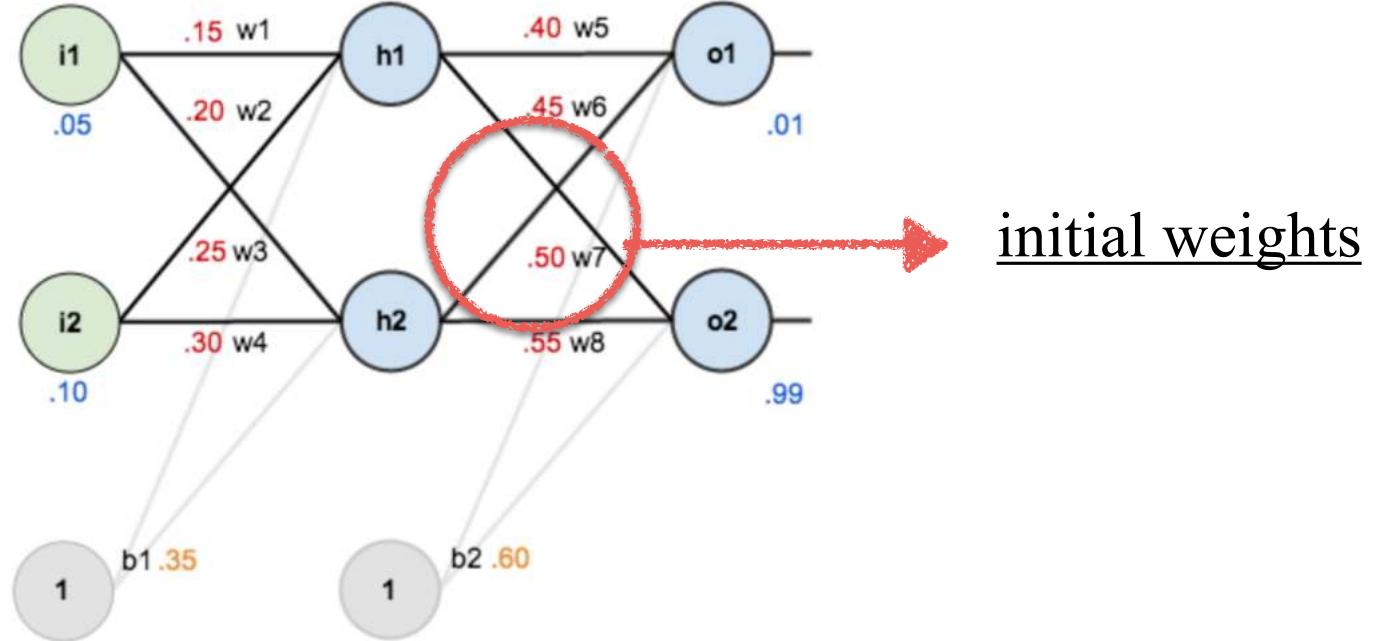
$$L_{total} = \sum 0.5(target - output)^2$$

$$L_{o1} = 0.5(target_{o1} - output_{o1})^2 = 0.5 \times (0.01 - 0.751)^2 = 0.274$$

$$L_{o2} = 0.023$$



$$L_{total} = L_{o1} + L_{o2} = 0.298$$



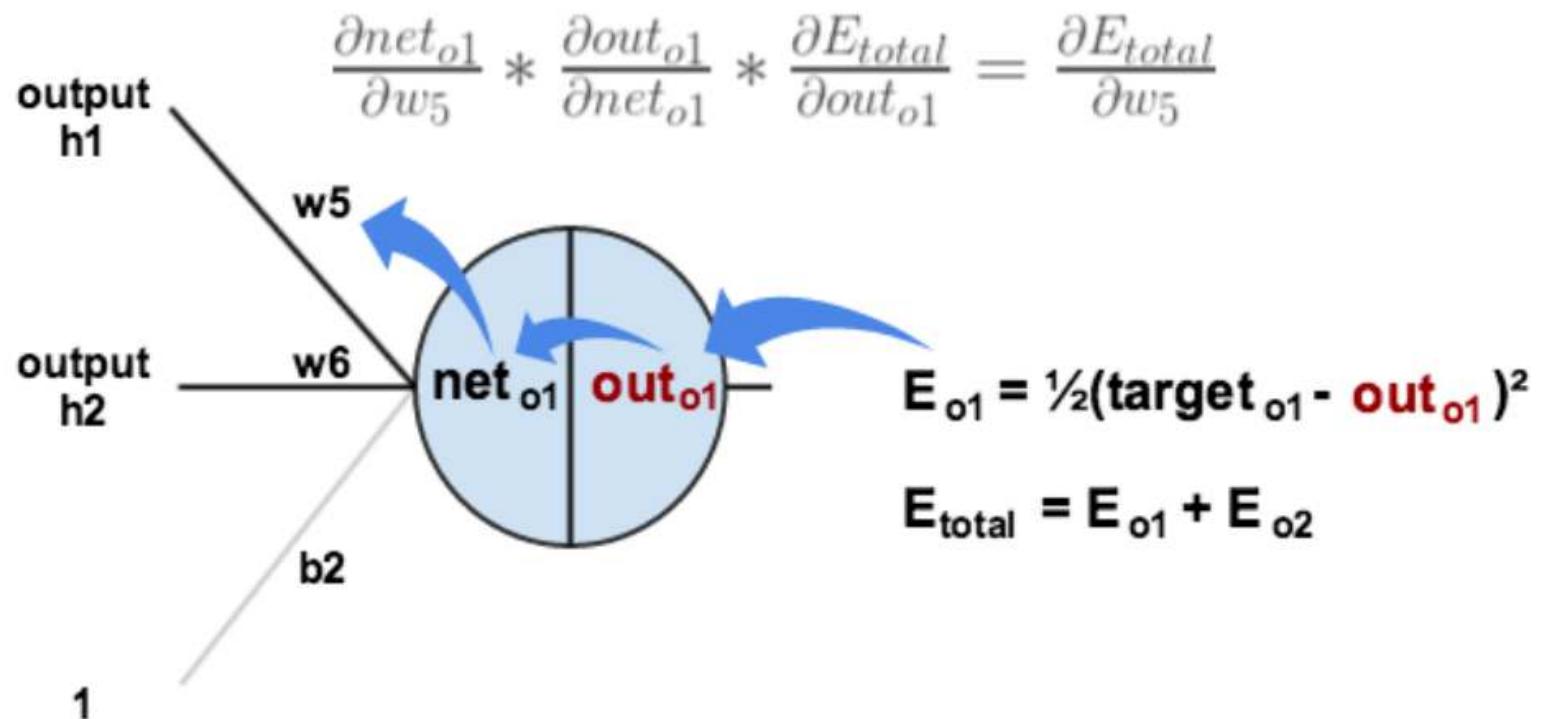
### 3. THE BACKWARD PASS

FOR W5

WE WANT:

$$\frac{\partial L_{total}}{\partial w_5}$$

[gradient of loss function]



### 3. THE BACKWARD PASS

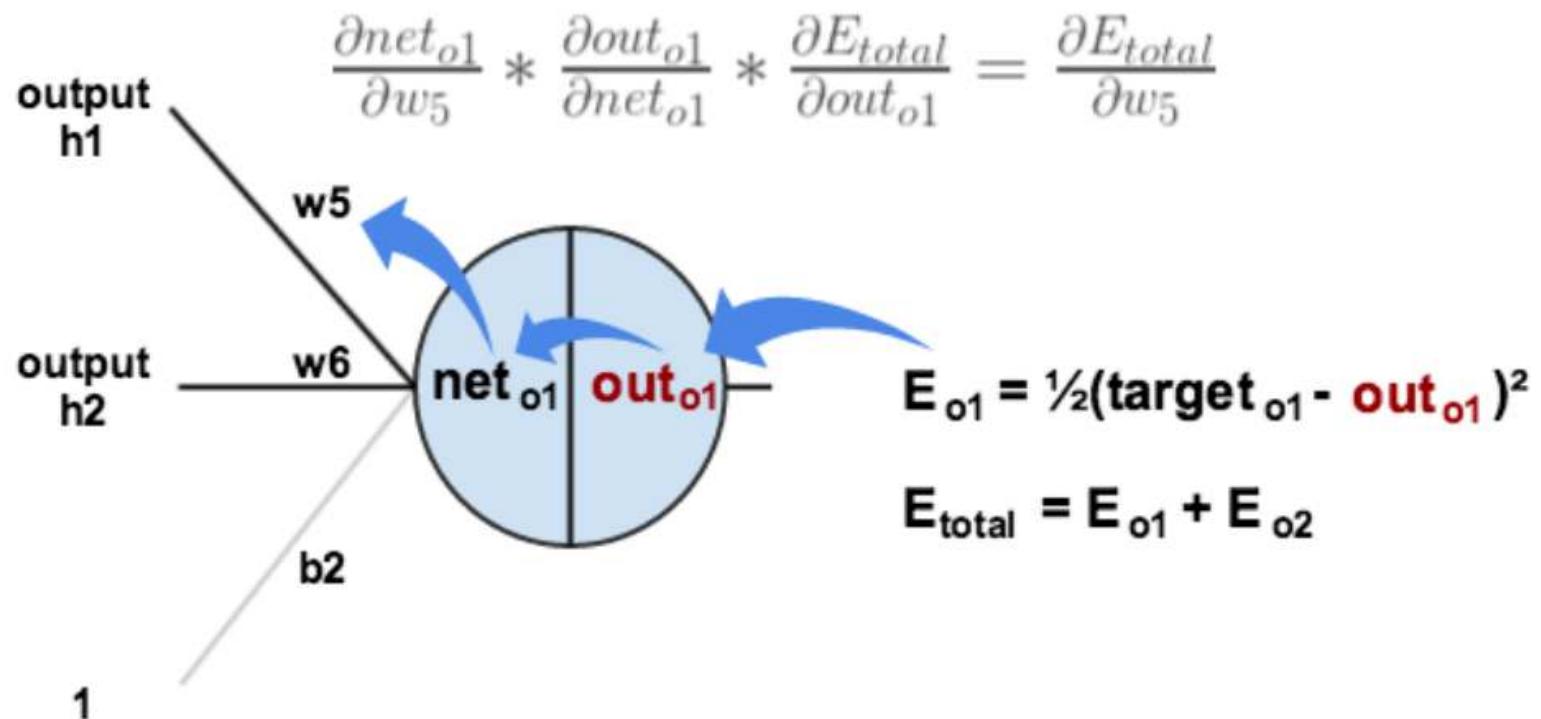
FOR  $w_5$

WE WANT:

$$\frac{\partial L_{total}}{\partial w_5} \quad [\text{gradient of loss function}]$$

WE APPLY THE CHAIN RULE:

$$\frac{\partial L_{total}}{\partial w_5} = \frac{\partial L_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial in_{o1}} \times \frac{\partial in_{o1}}{\partial w_5}$$

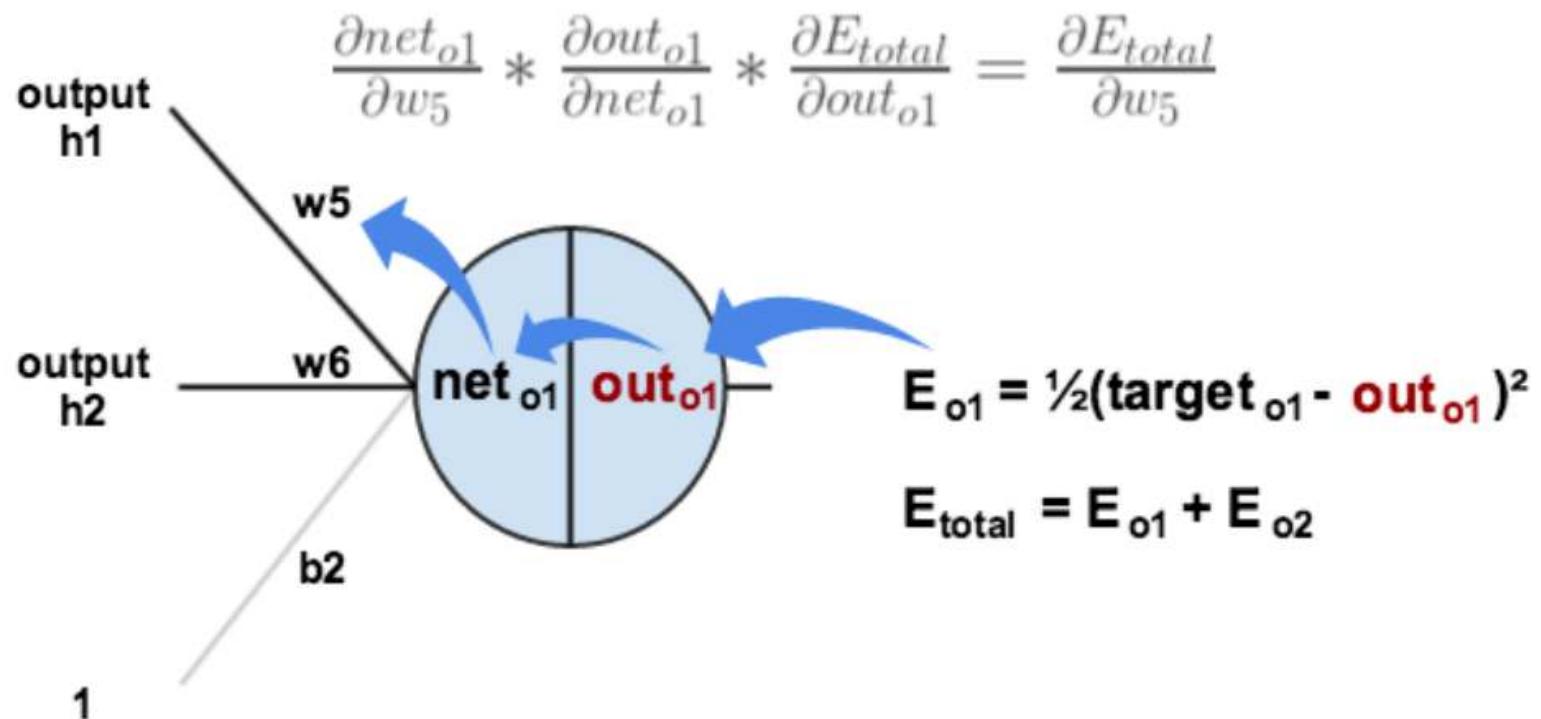


### 3. THE BACKWARD PASS

$$\frac{\partial L_{total}}{\partial w_5} = \frac{\partial L_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial in_{o1}} \times \frac{\partial in_{o1}}{\partial w_5}$$

$$L_{total} = 0.5(\text{target}_{o1} - out_{o1})^2 + 0.5(\text{target}_{o2} - out_{o2})^2$$

$$\frac{\partial L_{total}}{\partial out_{o1}} = 2 \times 0.5(\text{target}_{o1} - out_{o1}) \times (-1) = 0.741$$

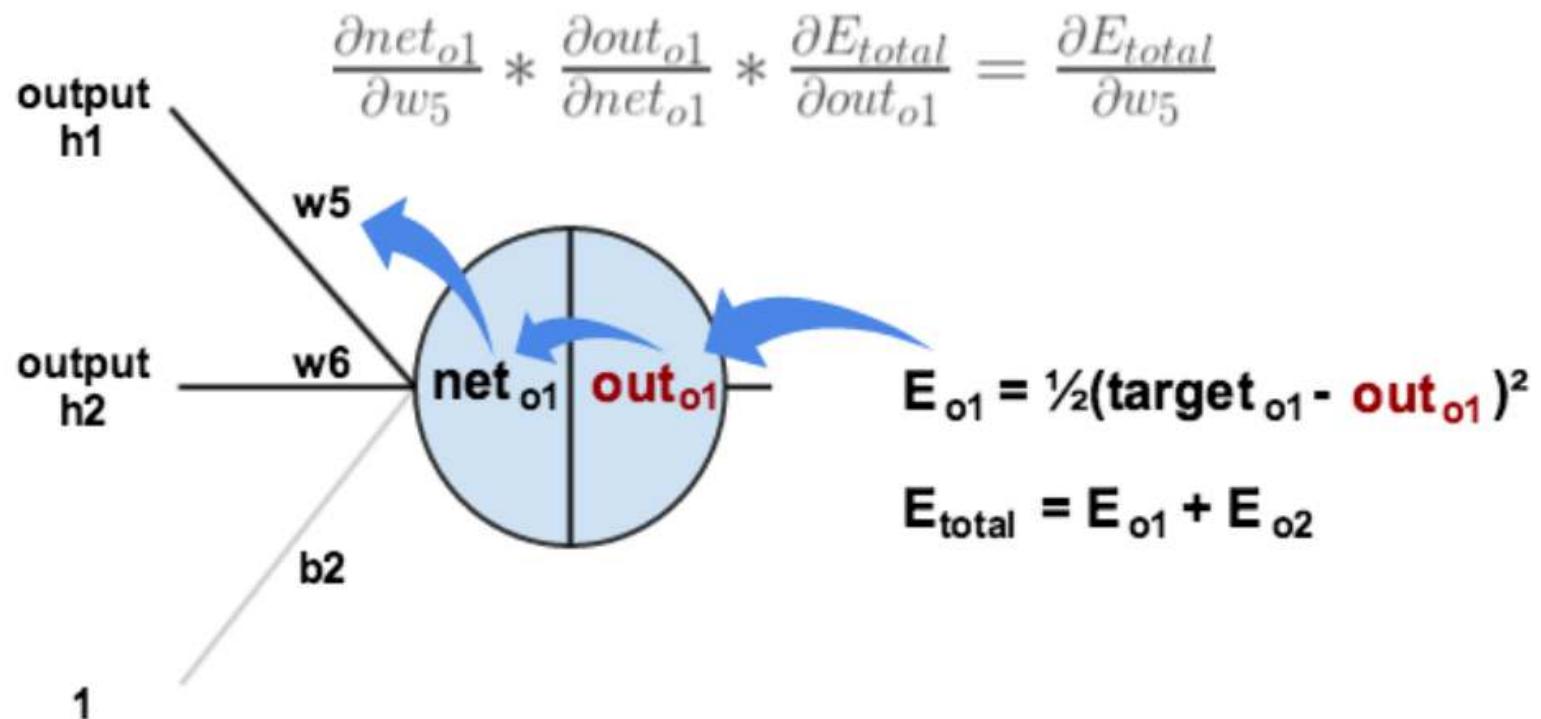


### 3. THE BACKWARD PASS

$$\frac{\partial L_{total}}{\partial w_5} = \frac{\partial L_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial in_{o1}} \times \frac{\partial in_{o1}}{\partial w_5}$$

$$out_{o1} = \frac{1}{1 + e^{-in_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial in_{o1}} = out_{o1} \times (1 - out_{o1}) = 0.186$$

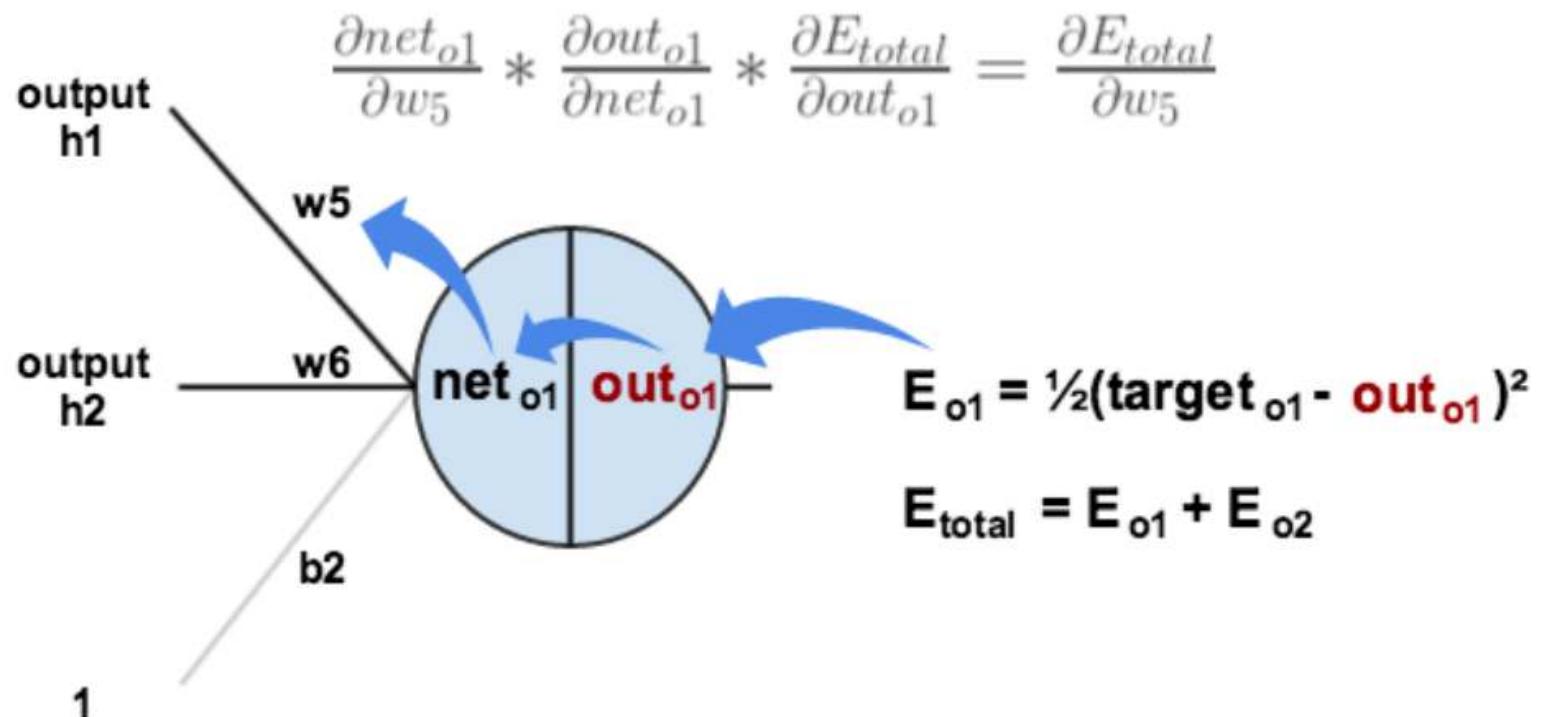


### 3. THE BACKWARD PASS

$$\frac{\partial L_{total}}{\partial w_5} = \frac{\partial L_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial in_{o1}} \times \frac{\partial in_{o1}}{\partial w_5}$$

$$in_{o1} = w_5 \times out_{h1} + w_6 \times out_{h2} + b_2$$

$$\frac{\partial in_{o1}}{\partial w_5} = out_{h1} \times w_5^{1-1} = out_{h1} = 0.593$$

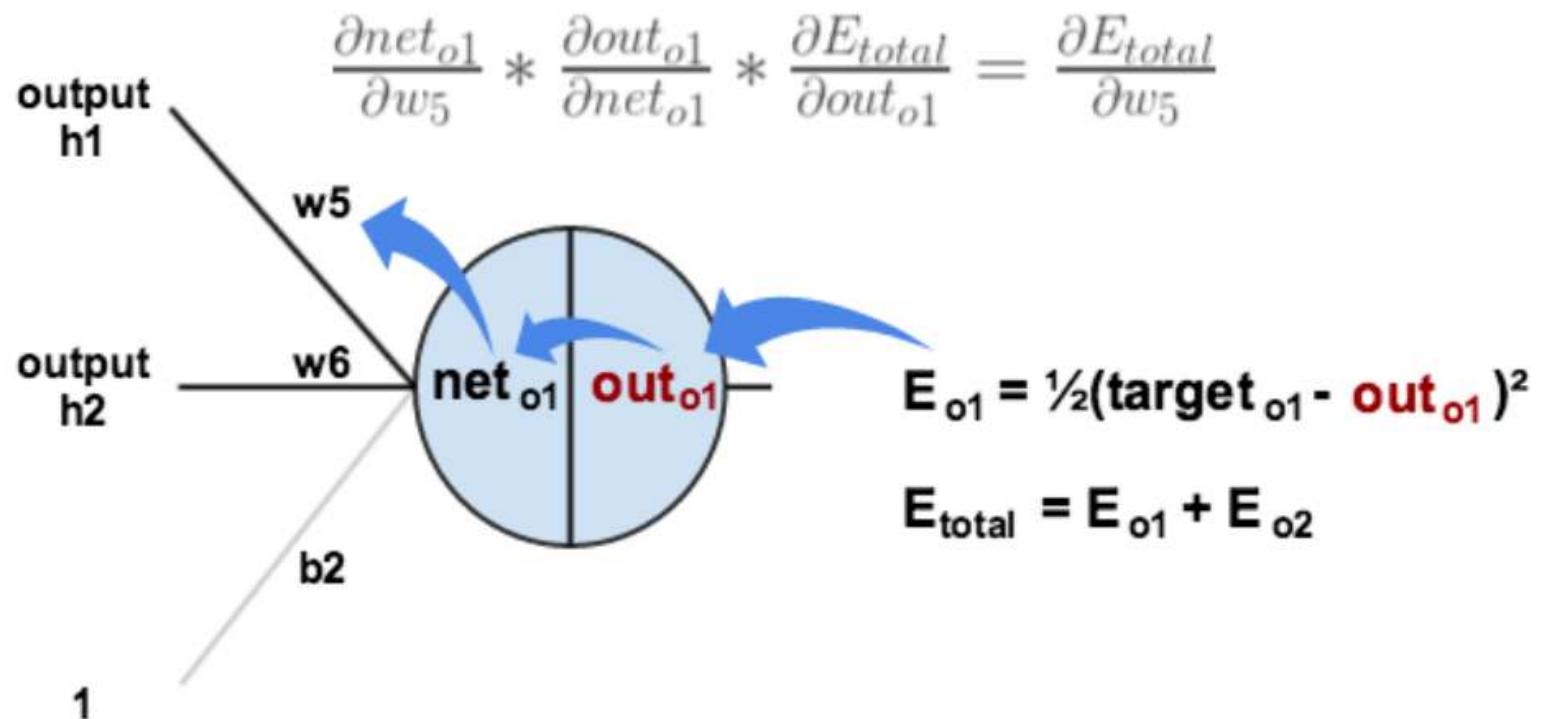


### 3. THE BACKWARD PASS

ALL TOGETHER:

$$\frac{\partial L_{total}}{\partial w_5} = \frac{\partial L_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial in_{o1}} \times \frac{\partial in_{o1}}{\partial w_5}$$

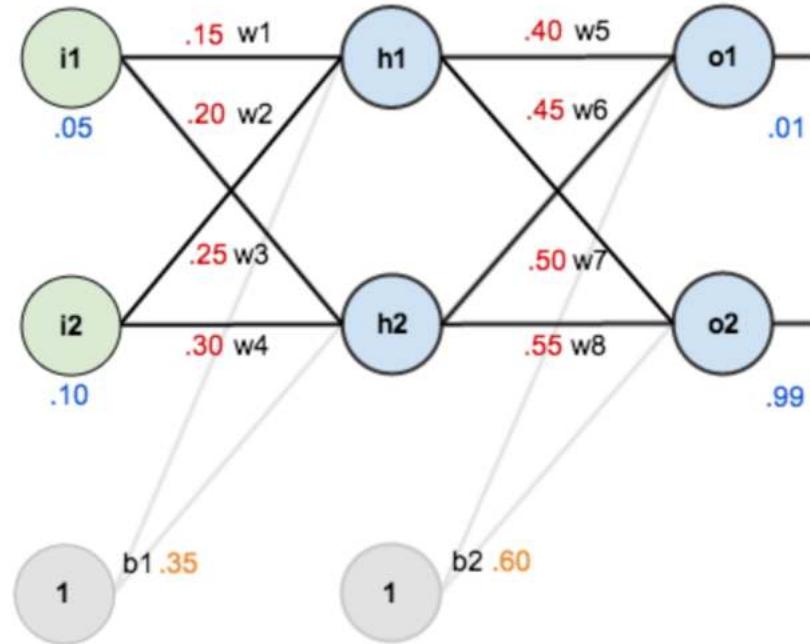
$$\frac{\partial L_{total}}{\partial w_5} = 0.741 \times 0.186 \times 0.593 = 0.082$$



## 4. UPDATE WEIGHTS WITH GRADIENT AND LEARNING RATE

$$w_5^{t+1} = w_5 - \lambda \times \frac{\partial L_{total}}{\partial w_5}$$

$$w_5^{t+1} = 0.4 - 0.5 \times 0.082 = 0.358$$



**THIS IS REPEATED FOR THE OTHER WEIGHTS  
OF THE OUTPUT LAYER**

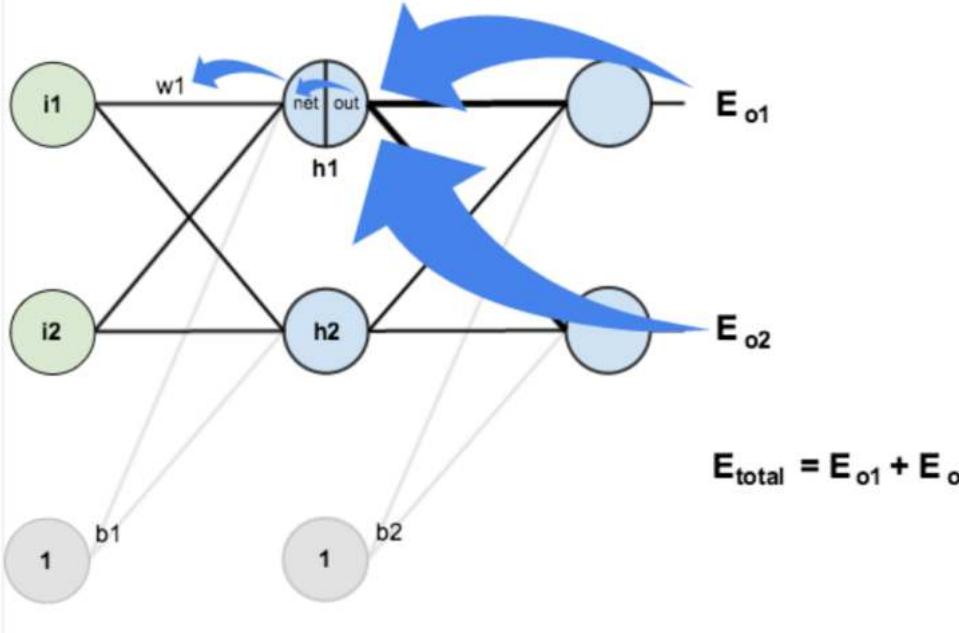
$$w_6^{t+1} = 0.408$$

$$w_7^{t+1} = 0.511$$

$$w_8^{t+1} = 0.561$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



**AND BACK-PROPAGATED TO THE HIDDEN LAYERS**

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT  
EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING  
THE STEEPEST DESCENT

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT  
EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING  
THE STEEPEST DESCENT

THE CHOICES OF THE LEARNING RATES AND THE  
FREQUENCY OF WEIGHT UPDATES ARE IMPORTANT HYPER  
PARAMETERS TO MAKE NNs MORE ROBUST

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT  
EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING  
THE STEEPEST DESCENT

THE CHOICES OF THE INITIAL WEIGHTS AND THE  
LEARNING RATES ARE IMPORTANT

# LEARNING RATES

$$W_{t+1} = W_t - \lambda \nabla f(W_t)$$



THERE ARE DIFFERENT WAYS  
TO UPDATE THE LEARNING RATE

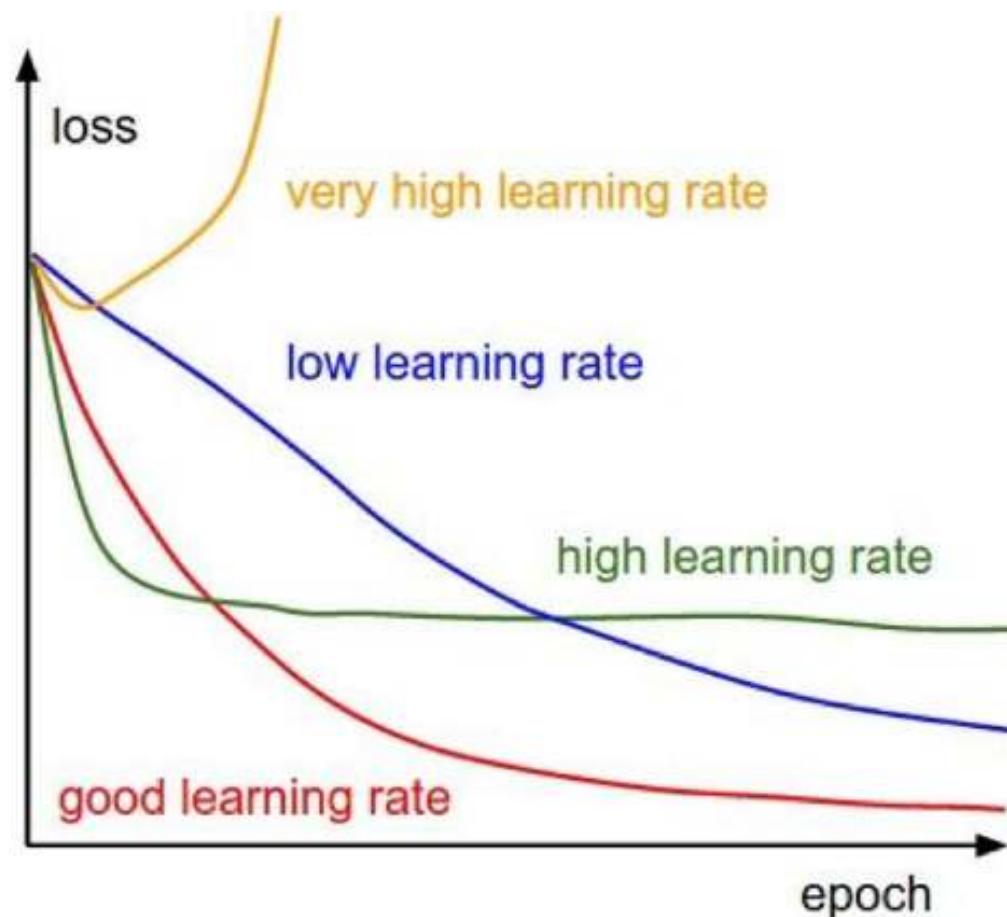
Credit:

ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT  
EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING  
THE STEEPEST DESCENT

THE CHOICES OF THE INITIAL WEIGHTS AND THE  
LEARNING RATES ARE IMPORTANT

WE WILL TALK ABOUT  
THIS LATER

# LEARNING RATES



Credit:

# LEARNING RATES

$$W_{t+1} = W_t - \lambda \nabla f(W_t)$$



THERE ARE DIFFERENT WAYS  
TO UPDATE THE LEARNING RATE

## ADAGRAD:

THE LEARNING RATE IS SCALED DEPENDING ON THE HISTORY OF PREVIOUS GRADIENTS

$$W_{t+1} = W_t - \frac{\lambda}{\sqrt{G_t + \epsilon}} \nabla f(W_t)$$

G IS A MATRIX CONTAINING ALL PREVIOUS GRADIENTS. WHEN THE GRADIENT BECOMES LARGE THE LEARNING RATE IS DECREASED AND VICE VERSA.

$$G_{t+1} = G_t + (\nabla f)^2$$

# LEARNING RATES

$$W_{t+1} = W_t - \lambda \nabla f(W_t)$$



THERE ARE DIFFERENT WAYS  
TO UPDATE THE LEARNING RATE

## RMSPROP:

THE LEARNING RATE IS SCALED DEPENDING ON THE HISTORY OF PREVIOUS GRADIENTS

$$W_{t+1} = W_t - \frac{\lambda}{\sqrt{G_t + \epsilon}} \nabla f(W_t)$$

SAME AS ADAGRAD BUT G IS CALCULATED BY EXPONENTIALLY DECAYING AVERAGE

$$G_{t+1} = \lambda G_t + (1 - \lambda)(\nabla f)^2$$

## **ADAM [Adaptive moment estimator]:**

SAME IDEA, USING FIRST AND SECOND ORDER  
MOMENTUMS

$$G_{t+1} = \beta_2 G_t + (1 - \beta_2)(\nabla f)^2 \quad M_{t+1} = \beta_1 M_t + (1 - \beta_1)(\nabla f)$$

$$W_{t+1} = W_t - \frac{\lambda}{\sqrt{\hat{G}_t + \epsilon}} \hat{M}_t$$

with:  $\hat{M}_{t+1} = \frac{M_t}{1 - \beta_1}$        $\hat{G}_{t+1} = \frac{G_t}{1 - \beta_2}$

## **ADAM [Adaptive moment estimator]:**

SAME IDEA, USING FIRST AND SECOND ORDER  
MOMENTUMS

$$G_{t+1} = \beta_2 G_t + (1 - \beta_2)(\nabla f)^2 \quad M_{t+1} = \beta_1 M_t + (1 - \beta_1)(\nabla f)$$

ONLY FOR YOUR  
RECORDS

$$\sqrt{G_t + \epsilon}$$

with:  $\hat{M}_{t+1} = \frac{M_t}{1 - \beta_1}$        $\hat{G}_{t+1} = \frac{G_t}{1 - \beta_2}$

# IN KERAS:

## RMSprop

[source]

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSProp optimizer.

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

This optimizer is usually a good choice for recurrent neural networks.

### Arguments

- `lr`: float  $\geq 0$ . Learning rate.
- `rho`: float  $\geq 0$ .
- `epsilon`: float  $\geq 0$ . Fuzz factor. If `None`, defaults to `K.epsilon()`.
- `decay`: float  $\geq 0$ . Learning rate decay over each update.

### References

- [rmsprop](#): Divide the gradient by a running average of its recent magnitude

# IN KERAS:

## Adam

[source]

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Adam optimizer.

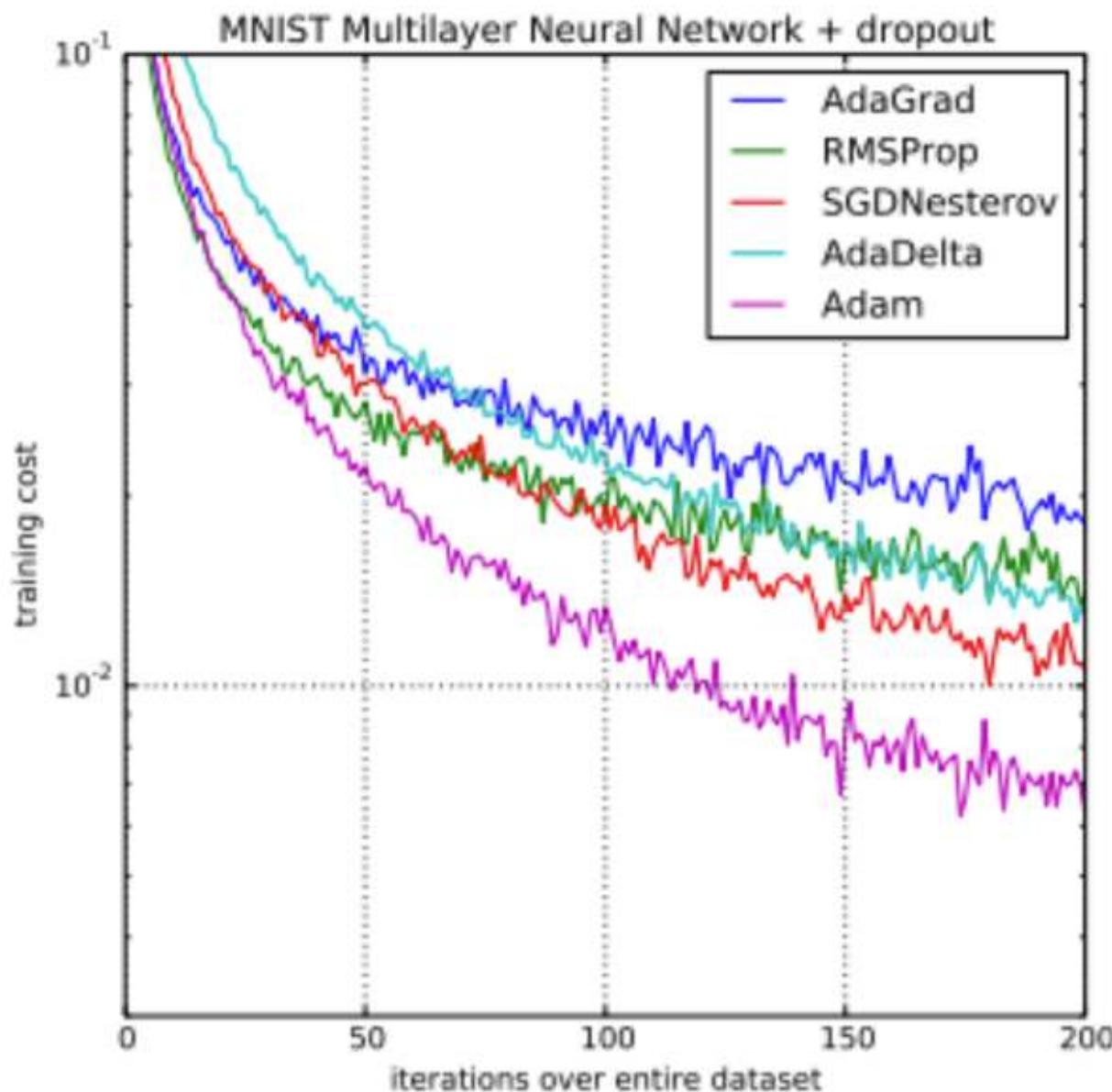
Default parameters follow those provided in the original paper.

### Arguments

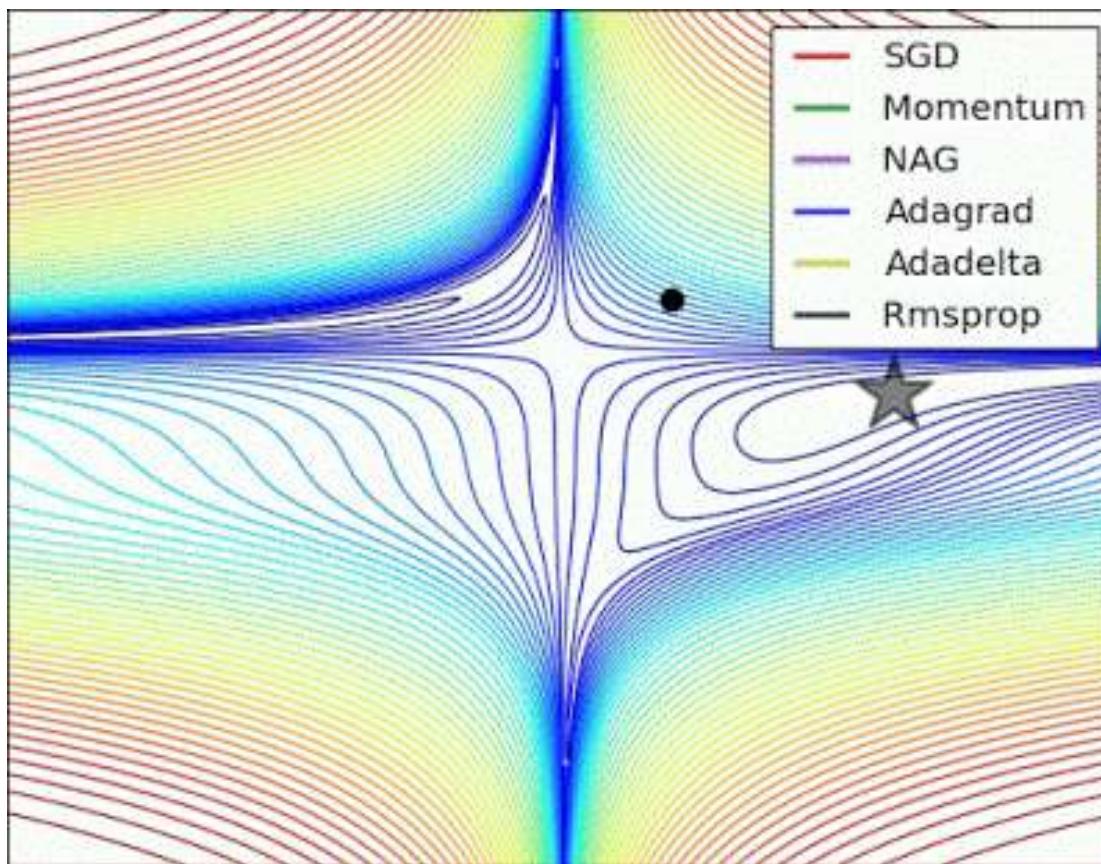
- **lr**: float  $\geq 0$ . Learning rate.
- **beta\_1**: float,  $0 < \text{beta} < 1$ . Generally close to 1.
- **beta\_2**: float,  $0 < \text{beta} < 1$ . Generally close to 1.
- **epsilon**: float  $\geq 0$ . Fuzz factor. If `None`, defaults to `K.epsilon()`.
- **decay**: float  $\geq 0$ . Learning rate decay over each update.
- **amsgrad**: boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

### References

- [Adam - A Method for Stochastic Optimization](#)
- [On the Convergence of Adam and Beyond](#)



Credit



Credit

# BATCH GRADIENT DESCENT

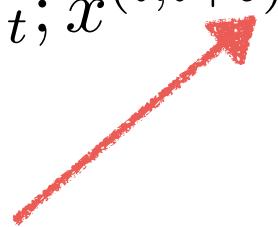
LOCAL MINIMA CAN ALSO BE AVOIDED BY COMPUTING THE  
GRADIENT IN SMALL BATCHES INSTEAD OF OVER THE FULL  
DATASET

# BATCH GRADIENT DESCENT

LOCAL MINIMA CAN ALSO BE AVOIDED BY COMPUTING THE GRADIENT IN SMALL BATCHES INSTEAD OF OVER THE FULL DATASET

## MINI-BATCH GRADIENT DESCENT

$$V_{t+1/num} = W_t - \lambda_t \nabla f(W_t; x^{(i,i+b)}, y^{(i,i+b)})$$



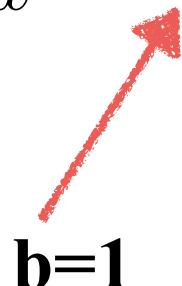
THE GRADIENT IS COMPUTED OVER A BATCH OF SIZE B

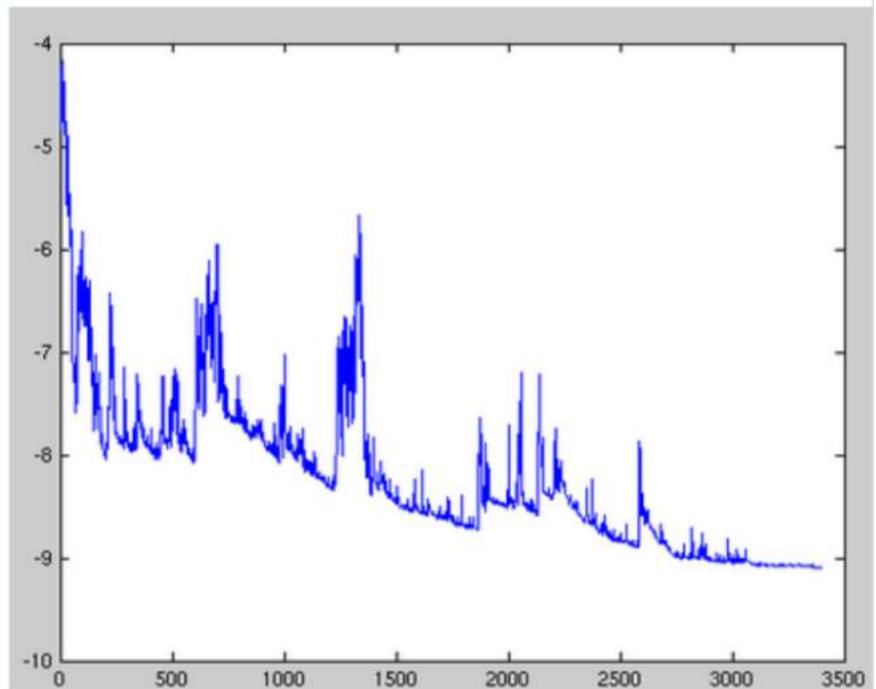
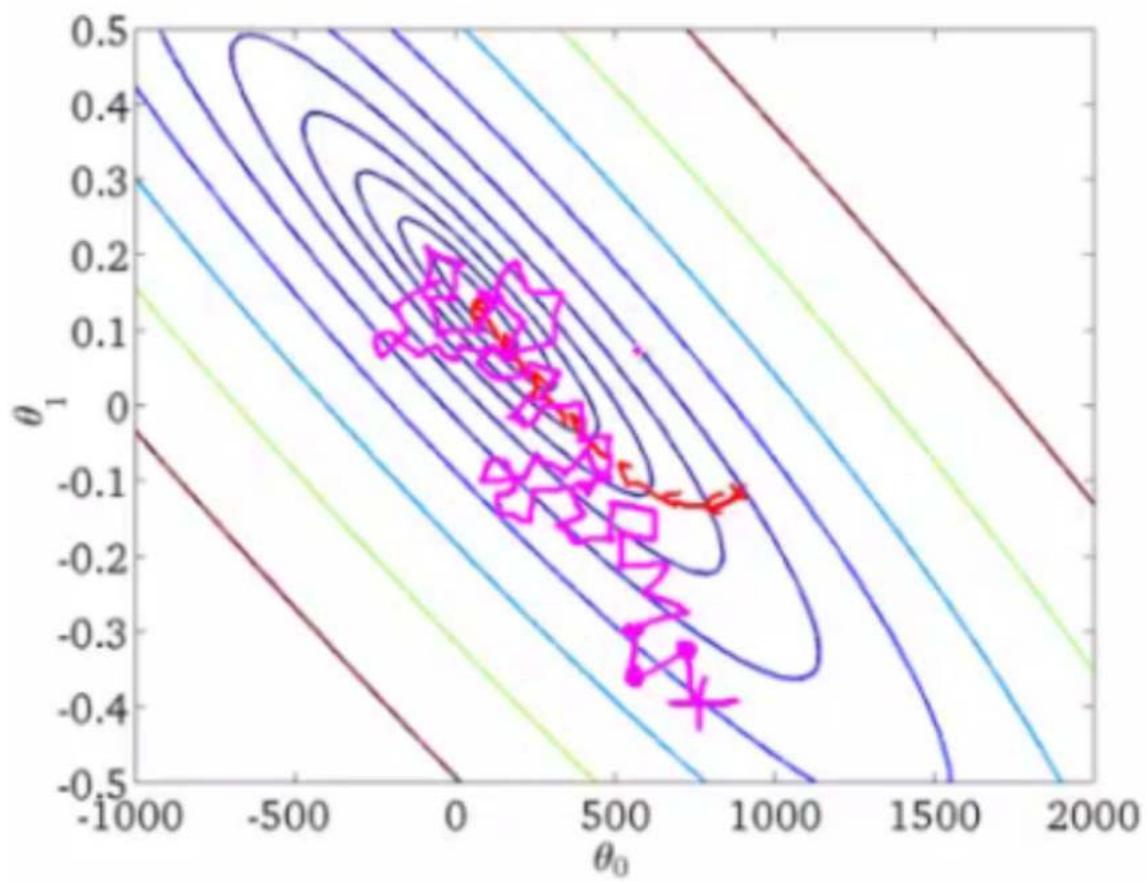
# STOCHASTIC GRADIENT DESCENT

THE EXTREME CASE IS TO COMPUTE THE GRADIENT ON EVERY TRAINING EXAMPLE.

## STOCHASTIC GRADIENT DESCENT

$$V_{t+1/num} = W_t - \lambda_t \nabla f(W_t; x^{(i,i+b)}, y^{(i,i+b)})$$

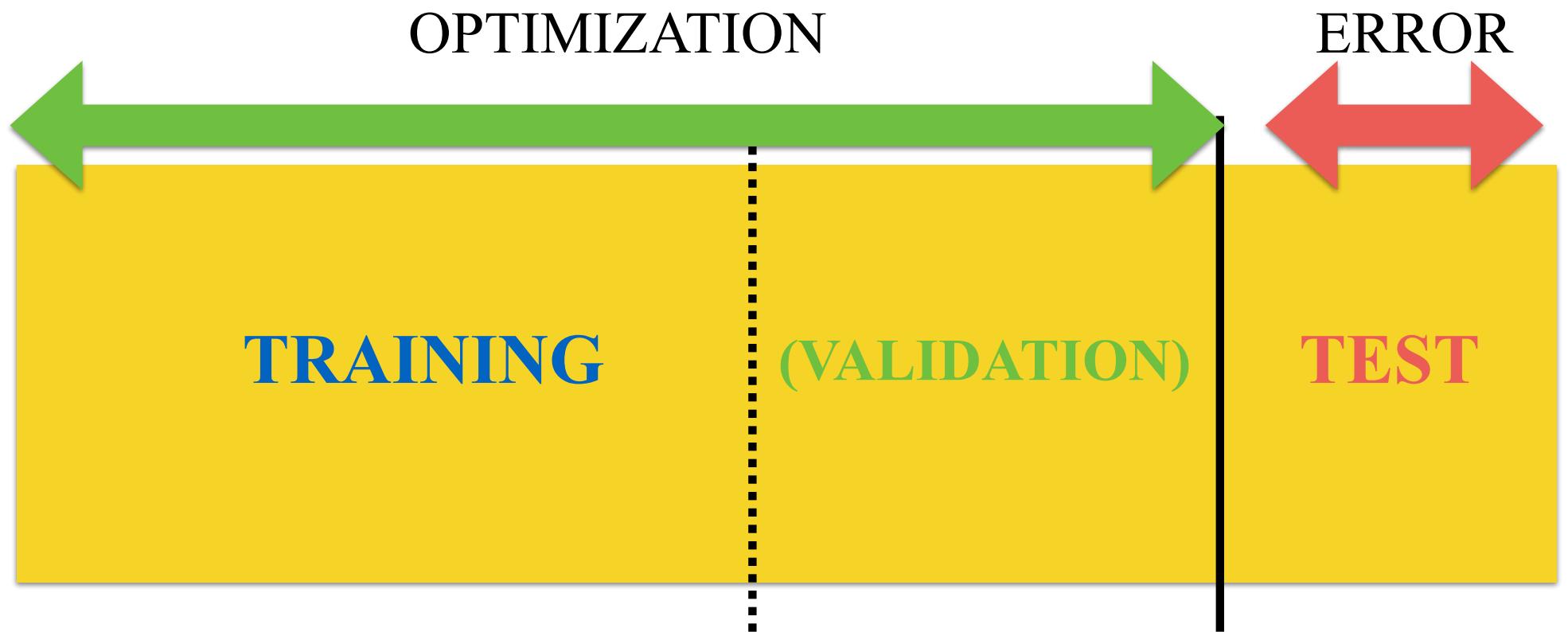




Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

Credit

# IN PRACTICE



training set: use to train the classifier

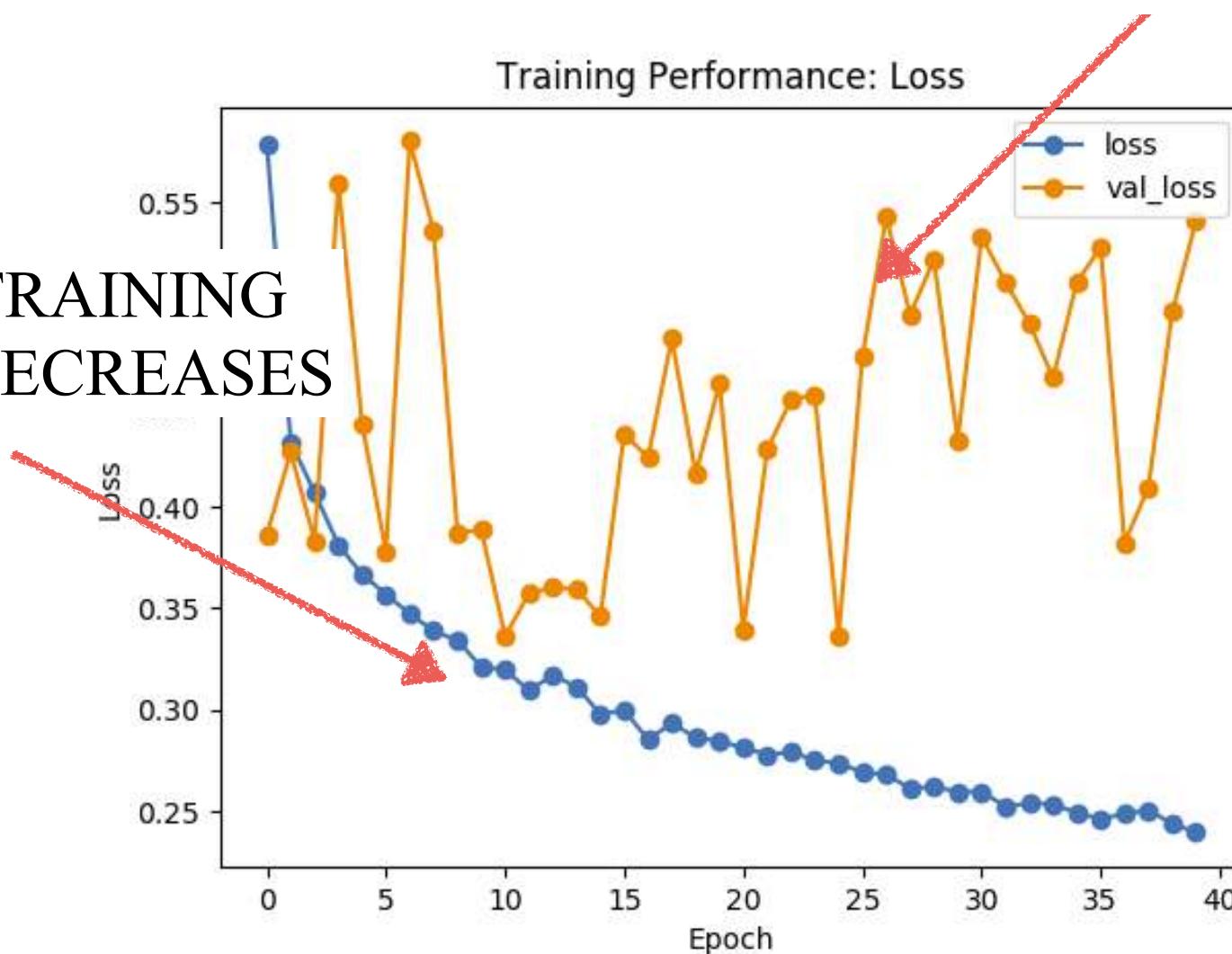
validation set: use to monitor performance in real time - check  
for overfitting

test set: use to train the classifier

# OVER-FITTING

THE TEST STAYS CONSTANT  
OR INCREASES

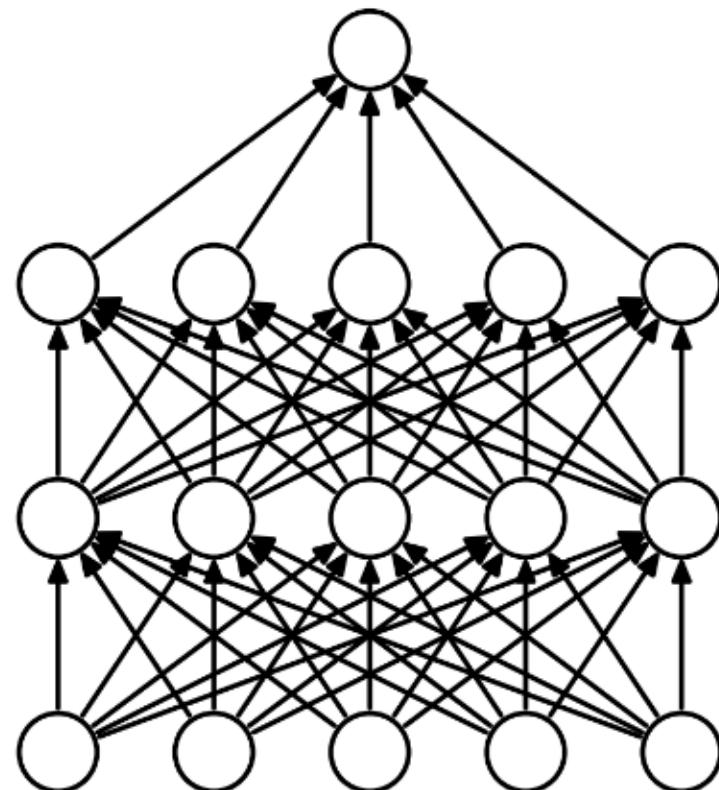
THE TRAINING  
LOSS DECREASES



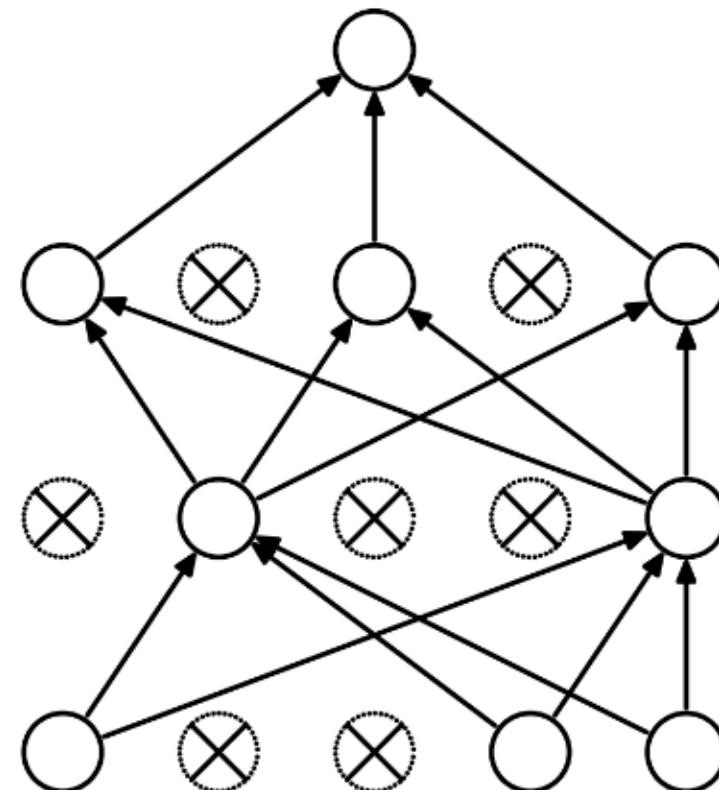
# DROPOUT

[Hinton+12]

- THE IDEA IS TO REMOVE NEURONS RANDOMLY DURING THE TRAINING
- ALL NEURONS ARE PUT BACK DURING THE TEST PHASE



(a) Standard Neural Net



(b) After applying dropout.

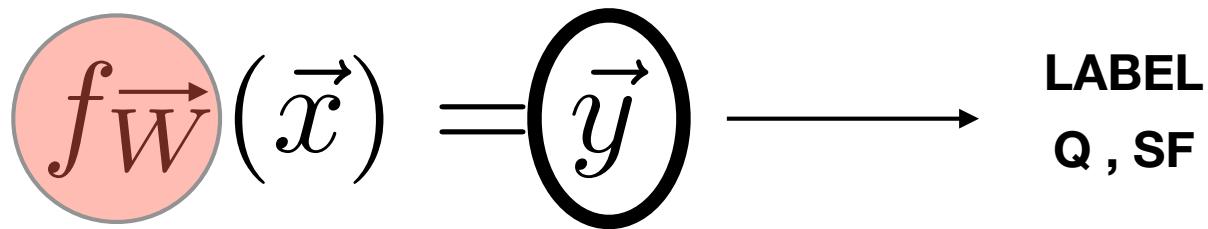
# DROPOUT

## WHY DOES IT WORK?

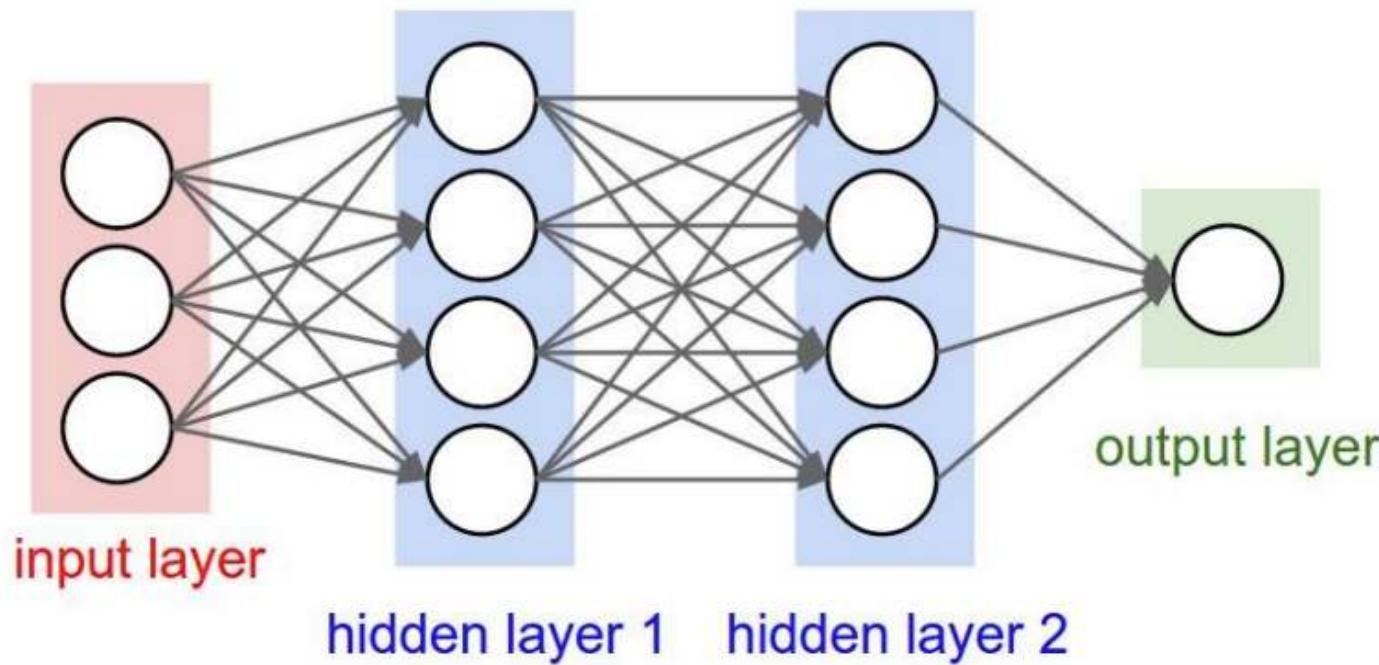
1. SINCE NEURONS ARE REMOVED RANDOMLY, IT AVOIDS CO-ADAPTATION AMONG THEMSELVES

2. DIFFERENT SETS OF NEURONS WHICH ARE SWITCHED OFF, REPRESENT A DIFFERENT ARCHITECTURE AND ALL THESE DIFFERENT ARCHITECTURES ARE TRAINED IN PARALLEL. FOR  $N$  NEURONS ATTACHED TO DROPOUT, THE NUMBER OF SUBSET ARCHITECTURES FORMED IS  $2^N$ . SO IT AMOUNTS TO PREDICTION BEING AVERAGED OVER THESE ENSEMBLES OF MODELS.

**"CLASSICAL"  
MACHINE LEARNING**

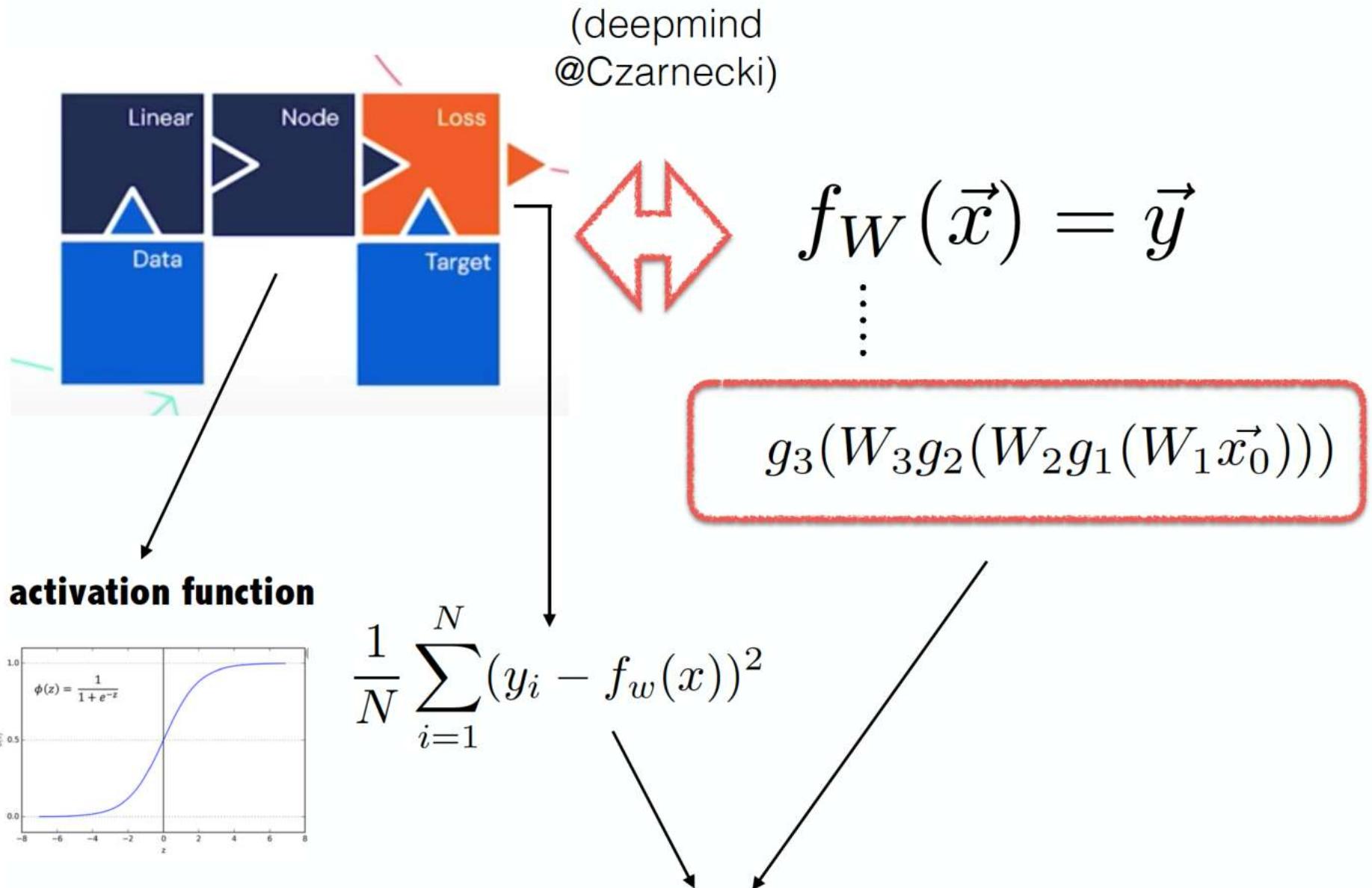


**REPLACE THIS BY A GENERAL  
NON LINEAR FUNCTION WITH SOME PARAMETERS W**



$$p = g_3(W_3g_2(W_2g_1(W_1\vec{x}_0))) \xleftarrow{\text{NETWORK FUNCTION}}$$

# RECAP:



**Minimized through gradient descent (backpropagation)**

# Neural Networks as statistical models

$$f_W(\vec{x}) = \vec{y} \quad \textcolor{red}{\leftrightarrow} \quad p(y|x; W)$$

# NEURAL NETWORKS AS STATISTICAL MODELS

Where does it come from?

$$H = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(f_w(x_i)) + (1 - y_i) \cdot \log(1 - f_w(x_i))$$

GROUND TRUTH

GROUND TRUTH

NETWORK OUTPUT



# NEURAL NETWORKS AS STATISTICAL MODELS

Where does it come from?

$$H = -\frac{1}{N} \sum_{i=1}^N y_i \log(f_w(x_i)) + (1 - y_i) \log(1 - f_w(x_i))$$

GROUND TRUTH

GROUND TRUTH

NETWORK OUTPUT  
(SOFTMAX)

NETWORK OUTPUT  
(SOFTMAX)

# BINARY CLASSIFICATION LOSS

WE TYPICALLY USE THE BINARY CROSS-ENTROPY LOSS:

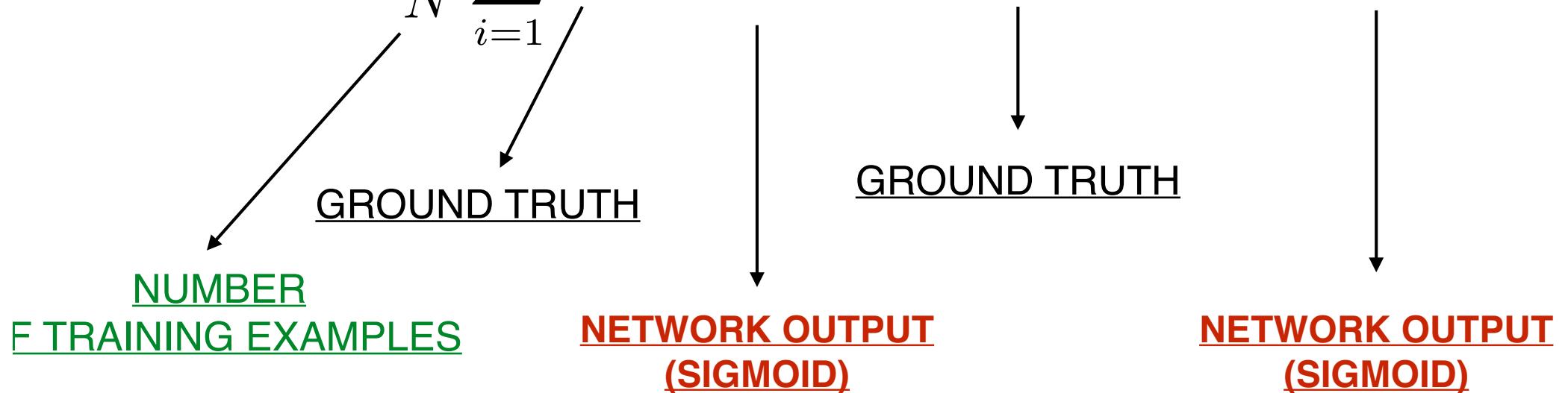
$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i))$$

The diagram illustrates the components of the Binary Cross-Entropy Loss formula. It shows three main parts: 'NUMBER OF TRAINING EXAMPLES' (green), 'GROUND TRUTH' (black), and 'NETWORK OUTPUT (SIGMOID)' (red). Arrows point from each part to its corresponding term in the formula. The first term,  $\frac{1}{N}$ , has arrows pointing from 'NUMBER OF TRAINING EXAMPLES' and 'GROUND TRUTH'. The second term,  $\sum_{i=1}^N$ , has an arrow pointing from 'GROUND TRUTH'. The third term,  $y_i \log(p(y_i))$ , has an arrow pointing from 'GROUND TRUTH'. The fourth term,  $(1 - y_i) \log(1 - p(y_i))$ , has an arrow pointing from 'NETWORK OUTPUT (SIGMOID)'.

# BINARY CLASSIFICATION LOSS

WE MINIMIZE THE CROSS ENTROPY BETWEEN THE TRUE [q(y)] AND PREDICTED [p(y)] DISTRIBUTIONS

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i))$$



# NEURAL NETWORKS AS STATISTICAL MODELS

Where does it come from?

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

The diagram illustrates the derivation of the cross-entropy loss function. It starts with the formula:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

Arrows point from the terms to their corresponding components:

- An arrow points from  $y_i \cdot \log(p(y_i))$  to GROUND TRUTH.
- An arrow points from  $(1 - y_i) \cdot \log(1 - p(y_i))$  to GROUND TRUTH.
- An arrow points from  $\log(p(y_i))$  to NETWORK OUTPUT (SOFTMAX).
- An arrow points from  $\log(1 - p(y_i))$  to NETWORK OUTPUT (SOFTMAX).

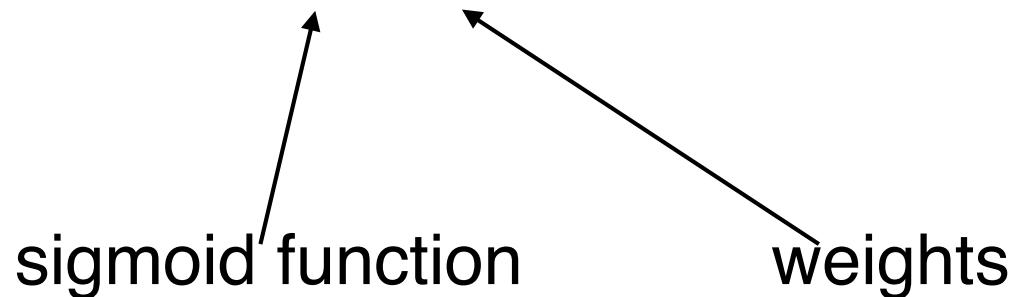
we have a set of independent realizations:

$$(x_i, y_i); i = 1, \dots, N$$

and want to find the underlying probability distribution of y given x:

Since y can only take 0 / 1 values, we assume it can be parametrized with a Bernouilli distribution:

$$P(y_i = 0|x_i) = 1 - \text{sigm}(f_w(x_i)) \quad P(y_i = 1|x_i) = \text{sigm}(f_w(x_i))$$



So the goal is to find the values of w (weights) that generate a Bernouilli distribution for y given a set of N independent observations

## This can be achieved via Maximum Likelihood estimation:

The likelihood of a given observation under the Bernouilli assumption can be written as:

$$L(w; x_i, y_i) = [\text{sigm}(f_w(x_i))]^{y_i} [1 - \text{sigm}(f_w(x_i))]^{1-y_i}$$

which is equal to  $[\text{sigm}(f_w(x_i))]$  if  $y=1$  and  $[1 - \text{sigm}(f_w(x_i))]$  if  $y=0$

And the likelihood of the entire sample is the product of the likelihoods:

$$L(w; x_i, y_i) = \prod_{i=1}^N [\text{sigm}(f_w(x_i))]^{y_i} [1 - \text{sigm}(f_w(x_i))]^{1-y_i}$$

So, the log-likelihood:

$$l(w; x_i, y_i) = \sum_{i=1}^N y_i \times \log[\text{sigm}(f_w(x_i))] + (1 - y_i) \times \log[1 - \text{sigm}(f_w(x_i))]$$

THEREFORE EQUIVALENT TO THE CROSS-ENTROPY LOSS:

$$l(w; x_i, y_i) = \sum_{i=1}^N y_i \times \log[\text{sigm}(f_w(x_i))] + (1 - y_i) \times \log[1 - \text{sigm}(f_w(x_i))]$$

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i))$$

WHEN YOU DO A BINARY CLASSIFICATION WITH NEURAL NETWORKS YOU ARE SIMPLY FINDING THE WEIGHTS OF YOUR MODEL THAT MAXIMIZE THE LIKELIHOOD OF A BERNOULLI DISTRIBUTION

NOTE THAT SINCE  $f_w$  IS FIXED (THE ARCHITECTURE), WE FIND A SOLUTION AMONG A POSSIBLE SET OF SOLUTIONS

# REGRESSIONS

$$MSE = \frac{1}{N} \sum [y_i - f_w(x_i)]^2$$

WHEN USING THE MEAN SQUARED ERROR, WE ARE ASSUMING THAT THE OUTPUT POINTS FOLLOW A NORMAL DISTRIBUTION. OUR ESTIMATOR IS THEREFORE THE MEAN OF THE NORMAL PDF THAT MAXIMIZES THE LIKELIHOOD (MAXIMUM LIKELIHOOD ESTIMATION).

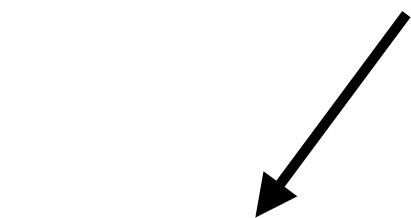
$$L(w, \sigma^2; y, X) = (2\pi\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - f_w(x_i))^2\right)$$

A diagram illustrating the components of the likelihood function. An arrow points from the word "weights" to the term  $f_w(x_i)$  in the equation. Another arrow points from the phrase "neural network" to the same term  $f_w(x_i)$ .

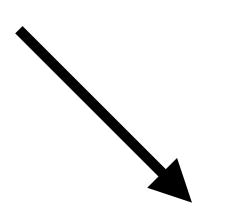
The standard deviation is assumed to be equal to one.

$$D = (x^{(i)}, y^{(i)})$$

$$p(D|w) = \prod_i p(y^{(i)}|x^{(i)}, W)$$



Crossentropy

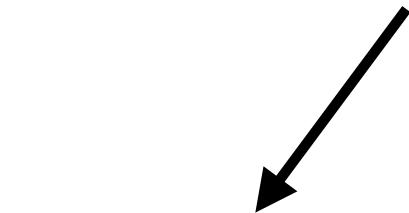


MSE

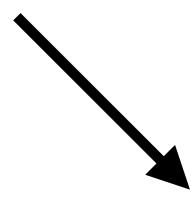
**A standard NN performs  
Maximum Likelihood  
Estimation  
of the weights w**

$$D = (x^{(i)}, y^{(i)})$$

$$p(D|w) = \prod_i p(y^{(i)}|x^{(i)}, W)$$



Crossentropy



MSE

**A standard NN performs  
Maximum Likelihood  
Estimation  
of the weights w**

If we assume a prior distribution for the weights  $p(w)$ :

$$p(w|D) \propto p(D|w).p(w) \quad \text{MAP: Maximum A Posteriori}$$



Posterior distribution over the weights

$$D = (x^{(i)}, y^{(i)})$$

$$p(D|w) = \prod_i p(y^{(i)}|x^{(i)}, W)$$

A standard NN performs  
Maximum Likelihood  
Estimation  
of the weights w

These estimations provide point estimates. How can we estimate uncertainties with a Neural Network?

If we assume a prior distribution for the weights  $p(w)$ :

$$p(w|D) \propto p(D|w).p(w) \quad \text{MAP: Maximum A Posteriori}$$

Posterior distribution over the weights

For regressions: let's do some “**standard**” bayesian inference

goal of inference  
*posterior*

$$p(\theta | X)$$

For regressions: let's do some “**standard**” bayesian inference

goal of inference

$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

*posterior*      *likelihood*    *prior*

we can evaluate the posterior using **Bayes rule**

For regressions: let's do some “**standard**” bayesian inference

goal of inference

$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

*posterior*      *likelihood*    *prior*

**prior** specified by prior experiments

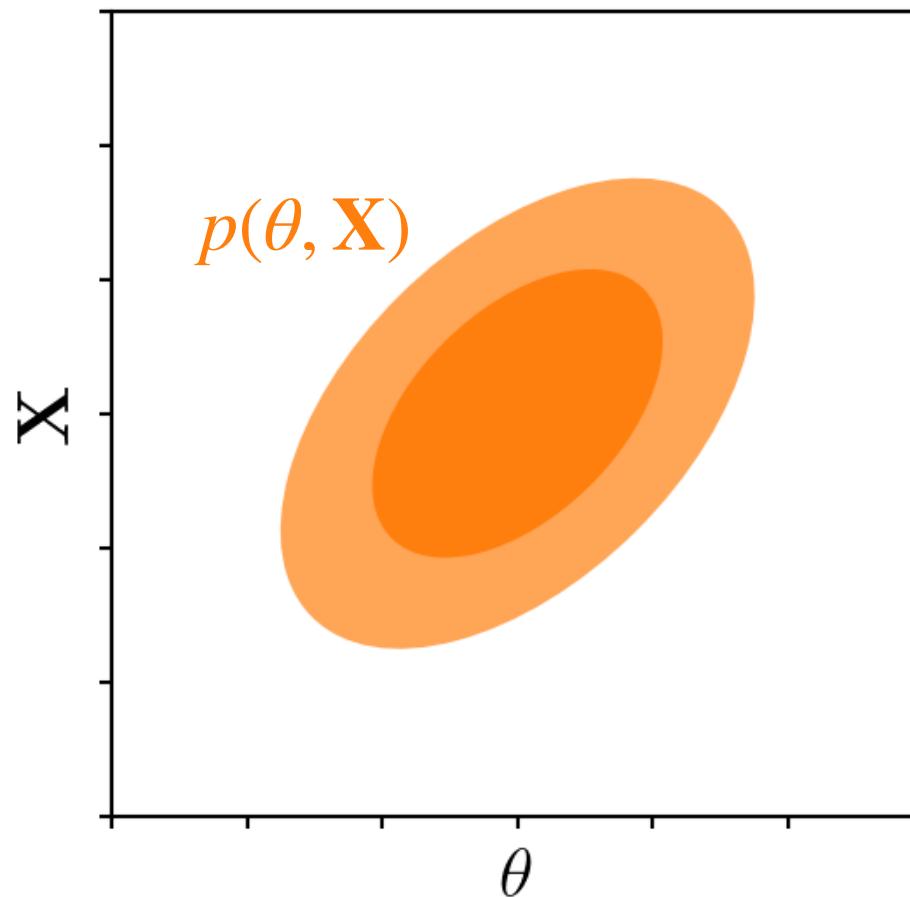
For regressions: let's do some “**standard**” bayesian inference

goal of inference

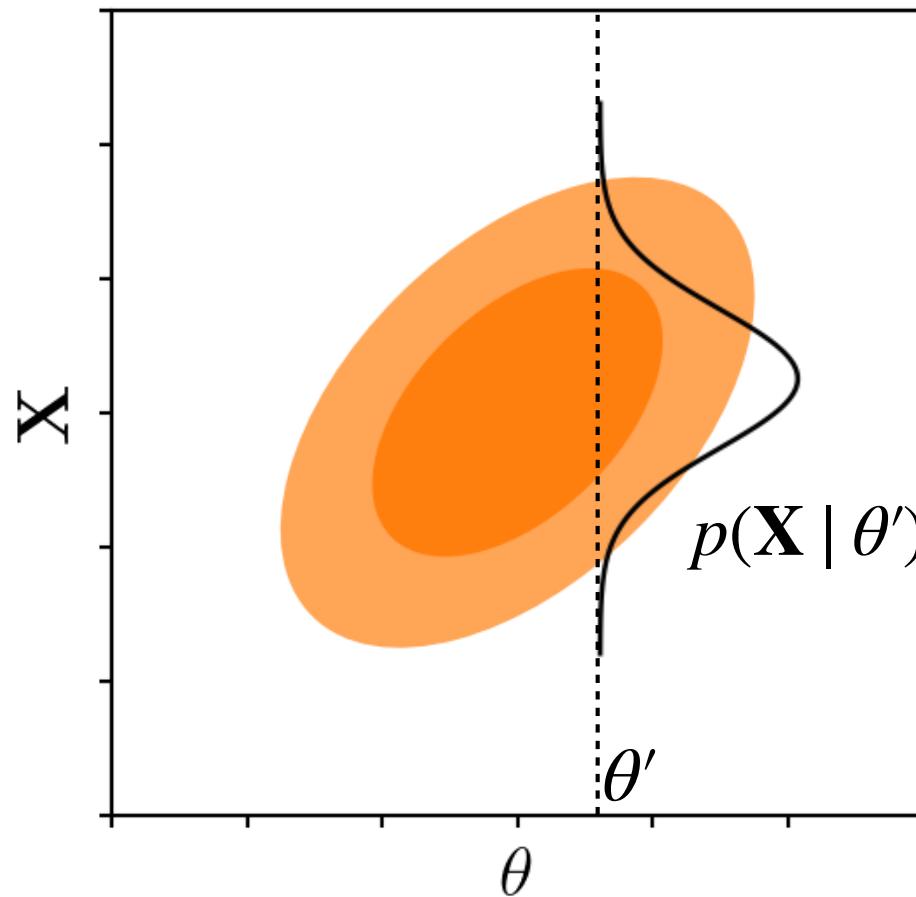
$$p(\theta | X) = \frac{likelihood \ prior}{p(X)}$$
$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

**likelihood** typically assumed to be multivariate Gaussian

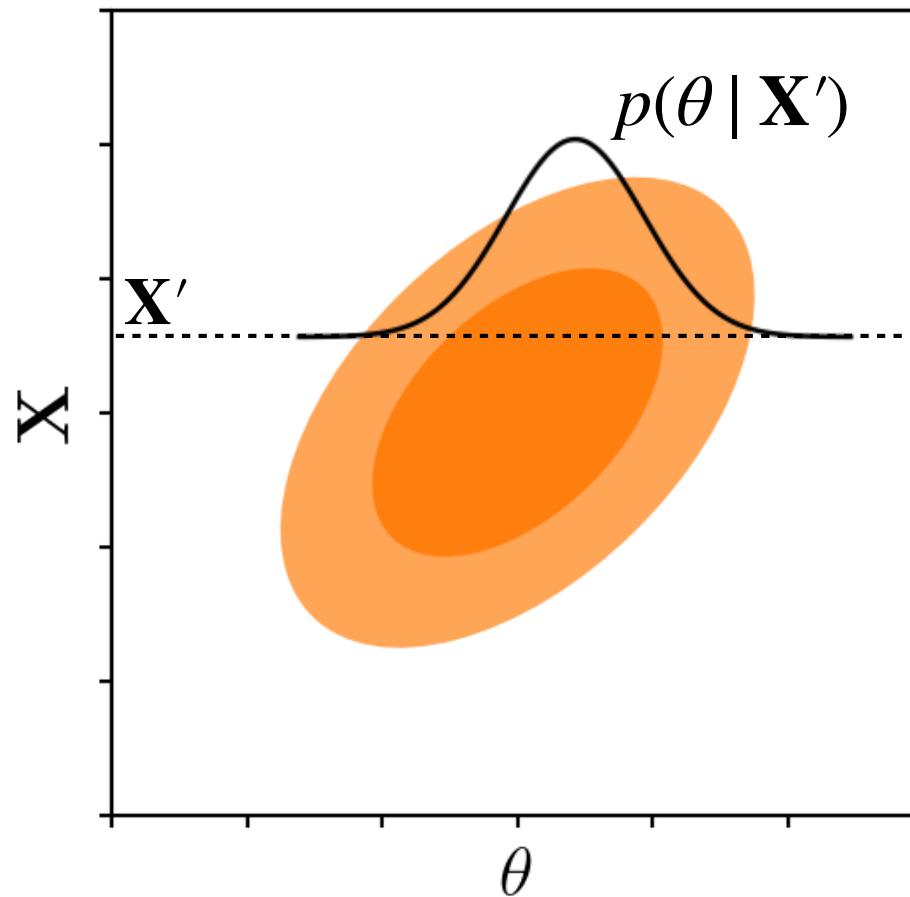
consider an experiment with *1D observation  $\mathbf{X}$*  and *1D parameter  $\theta$*



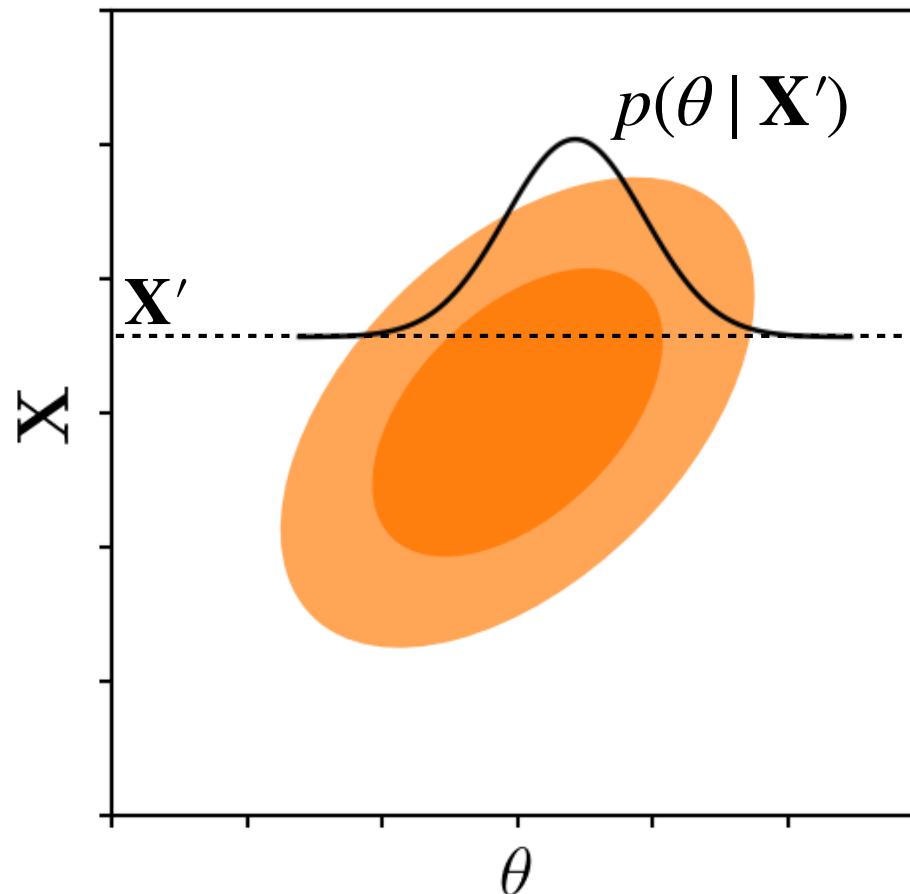
$p(\theta, \mathbf{X})$  includes both the **likelihood**  $p(\mathbf{X} | \theta')$  and posterior  $p(\theta | \mathbf{X}')$



$p(\theta, \mathbf{X})$  includes both the likelihood  $p(\mathbf{X} | \theta')$  and **posterior**  $p(\theta | \mathbf{X}')$



$p(\theta, \mathbf{X})$  includes both the likelihood  $p(\mathbf{X} | \theta')$  and posterior  $p(\theta | \mathbf{X}')$



$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

$$p(\theta, X) = p(X | \theta) p(\theta) = p(\theta | X) p(X) \longrightarrow \text{Bayes rule}$$

For regressions: let's do some “standard” bayesian inference

goal of inference

$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

*posterior*      *likelihood prior*

$\ln p(X | \theta) = \ln \mathcal{L} = (X - m(\theta))^T \mathbf{C}^{-1} (X - m(\theta))$

Mean of gaussians

covariance matrix

model

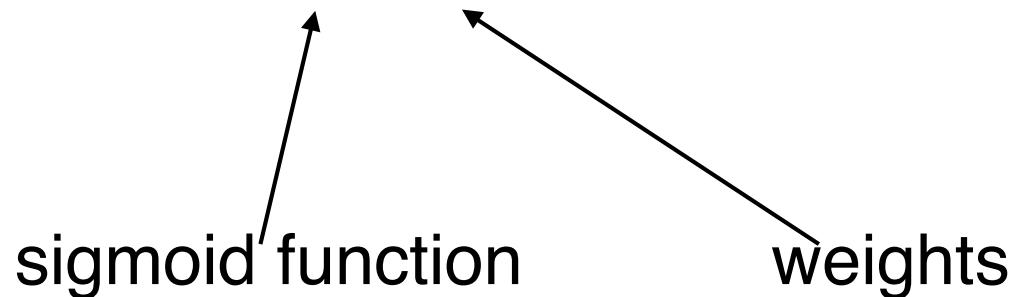
we have a set of independent realizations:

$$(x_i, y_i); i = 1, \dots, N$$

and want to find the underlying probability distribution of y given x:

Since y can only take 0 / 1 values, we assume it can be parametrized with a Bernouilli distribution:

$$P(y_i = 0|x_i) = 1 - \text{sigm}(f_w(x_i)) \quad P(y_i = 1|x_i) = \text{sigm}(f_w(x_i))$$



So the goal is to find the values of w (weights) that generate a Bernouilli distribution for y given a set of N independent observations

## This can be achieved via Maximum Likelihood estimation:

The likelihood of a given observation under the Bernouilli assumption can be written as:

$$L(w; x_i, y_i) = [\text{sigm}(f_w(x_i))]^{y_i} [1 - \text{sigm}(f_w(x_i))]^{1-y_i}$$

which is equal to  $[\text{sigm}(f_w(x_i))]$  if  $y=1$  and  $[1 - \text{sigm}(f_w(x_i))]$  if  $y=0$

And the likelihood of the entire sample is the product of the likelihoods:

$$L(w; x_i, y_i) = \prod_{i=1}^N [\text{sigm}(f_w(x_i))]^{y_i} [1 - \text{sigm}(f_w(x_i))]^{1-y_i}$$

So, the log-likelihood:

$$l(w; x_i, y_i) = \sum_{i=1}^N y_i \times \log[\text{sigm}(f_w(x_i))] + (1 - y_i) \times \log[1 - \text{sigm}(f_w(x_i))]$$

THEREFORE EQUIVALENT TO THE CROSS-ENTROPY LOSS:

$$l(w; x_i, y_i) = \sum_{i=1}^N y_i \times \log[\text{sigm}(f_w(x_i))] + (1 - y_i) \times \log[1 - \text{sigm}(f_w(x_i))]$$

$$H = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(f_w(x_i)) + (1 - y_i) \cdot \log(1 - f_w(x_i))$$

WHEN YOU DO A BINARY CLASSIFICATION WITH NEURAL NETWORKS YOU ARE SIMPLY FINDING THE WEIGHTS OF YOUR MODEL THAT MAXIMIZE THE LIKELIHOOD OF A BERNOULLI DISTRIBUTION

ASSUMES THAT THE POSTERIOR  $p(y|x)$  CAN BE MODELLED BY A BERNOULLI DISTRIBUTION

For regressions: let's do some “standard” bayesian inference

goal of inference

<i>posterior</i>	<i>likelihood</i>	<i>prior</i>
------------------	-------------------	--------------

$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

$\ln p(X | \theta) = \ln \mathcal{L} = (X - m(\theta))^T \mathbf{C}^{-1} (X - m(\theta))$

Mean of gaussians

↑

model

covariance matrix

For regressions: let's do some “**standard**” bayesian inference

goal of inference  
*posterior*      *likelihood prior*

$$p(\theta | X) = \frac{p(X | \theta) p(\theta)}{p(X)}$$

$\ln p(X | \theta) = \ln \mathcal{L} = (X - m(\theta))^T \mathbf{C}^{-1} (X - m(\theta))$

Mean of gaussians  
model  
covariance matrix

If all Gaussians are assumed to be independent and with standard deviation = 1, then:

$$\mathbf{C}^{-1} = \mathbf{I} \quad \ln p(X | \theta) = (X - m(\theta))^2 = MSE$$

# REGRESSIONS

$$L(w, \sigma^2; y, X) = (2\pi\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - f_w(x_i))^2\right)$$


weights

neural network

WHEN USING THE MEAN SQUARED ERROR, WE ARE ASSUMING THAT THE POSTERIOR  $P(y|x)$  IS A NORMAL DISTRIBUTION.

OUR ESTIMATOR IS THEREFORE THE MEAN OF THE NORMAL PDF THAT MAXIMIZES THE LIKELIHOOD (MAXIMUM LIKELIHOOD ESTIMATION).

**The standard deviation is assumed to be constant (equal to one)**

$$\ln p(X|\theta) = (X - m(\theta))^2 = MSE$$

If you do not care about the posterior, one way to find the parameters of the model theta, is by maximising the likelihood: **maximum likelihood estimation, i.e. minimising the MSE**

When training a NN with an MSE loss, we are finding the parameters of the network w that maximise the likelihood, under a Gaussian assumption.

$$\ln p(X|w) \sim (X - f_w(X))^2 = MSE$$

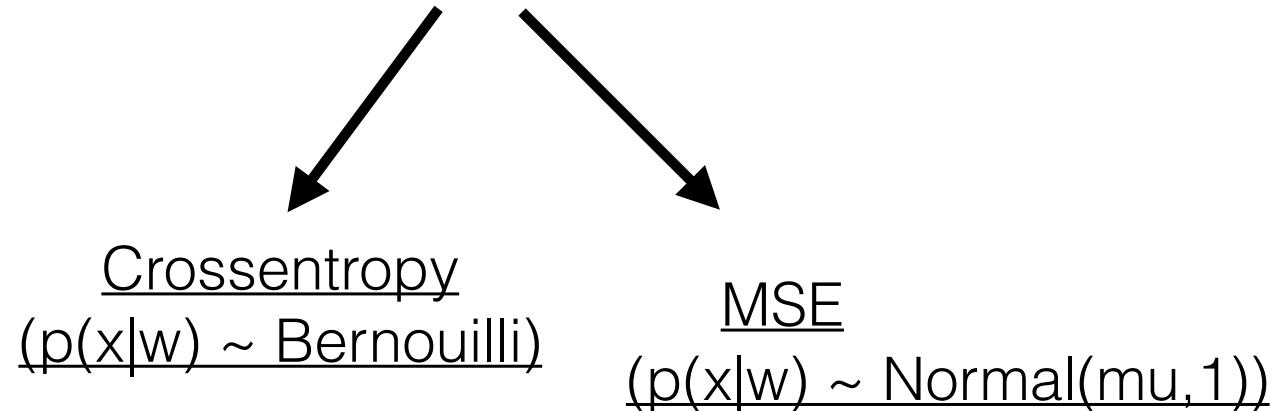


Weights of NN

# A standard NN performs Maximum Likelihood Estimation of the weights W

$$D = (x^{(i)}, y^{(i)})$$

$$p(D|w) = \prod_i p(y^{(i)}|x^{(i)}, W)$$



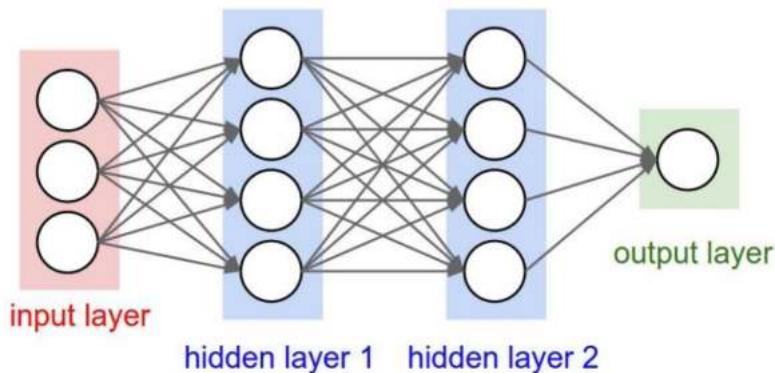
And then, under the “brutal” assumption, that one can marginalise over all possible NN weights:

$$\ln p(X|w) \sim \ln p(X|y)$$

And then, under the “brutal” assumption, that one can marginalise over all possible NN weights:

$$\ln p(X|w) \sim \ln p(X|y)$$

And if you assume even further that the prior distribution of  $y$  is encoded in the training set:



$$\sim p(y|X)$$

REMEMBER, THE COVARIANCE MATRIX IS NOTHING ELSE THAN THAT:

$$K_{\mathbf{XX}} = \begin{bmatrix} E[(X_1 - E[X_1])(X_1 - E[X_1])] & E[(X_1 - E[X_1])(X_2 - E[X_2])] & \cdots & E[(X_1 - E[X_1])(X_n - E[X_n])] \\ E[(X_2 - E[X_2])(X_1 - E[X_1])] & E[(X_2 - E[X_2])(X_2 - E[X_2])] & \cdots & E[(X_2 - E[X_2])(X_n - E[X_n])] \\ \vdots & \vdots & \ddots & \vdots \\ E[(X_n - E[X_n])(X_1 - E[X_1])] & E[(X_n - E[X_n])(X_2 - E[X_2])] & \cdots & E[(X_n - E[X_n])(X_n - E[X_n])] \end{bmatrix}$$

## “standard” bayesian inference in galaxy evolution

$$\ln p(\theta | X) = \ln p(X | \theta) + \ln p(\theta) + \text{const.}$$

sample  $p(\theta | X)$  using monte carlo sampling to estimate posterior  
(e.g. *MCMC, HMC, nested sampling*)

**why simulation-based inference?**

Gaussian likelihood is often **an incorrect assumption**

$$\ln p(X | \theta) = \ln \mathcal{L} = \frac{1}{2} (X - m(\theta))^T \mathbf{C}^{-1} (X - m(\theta))$$

**why simulation-based inference?**

Gaussian likelihood is often **an incorrect assumption**

simulations have **advantages over analytic models**

*e.g. observational systematics, non-linear regimes*

**why simulation-based inference?**

Gaussian likelihood is often **an incorrect assumption**

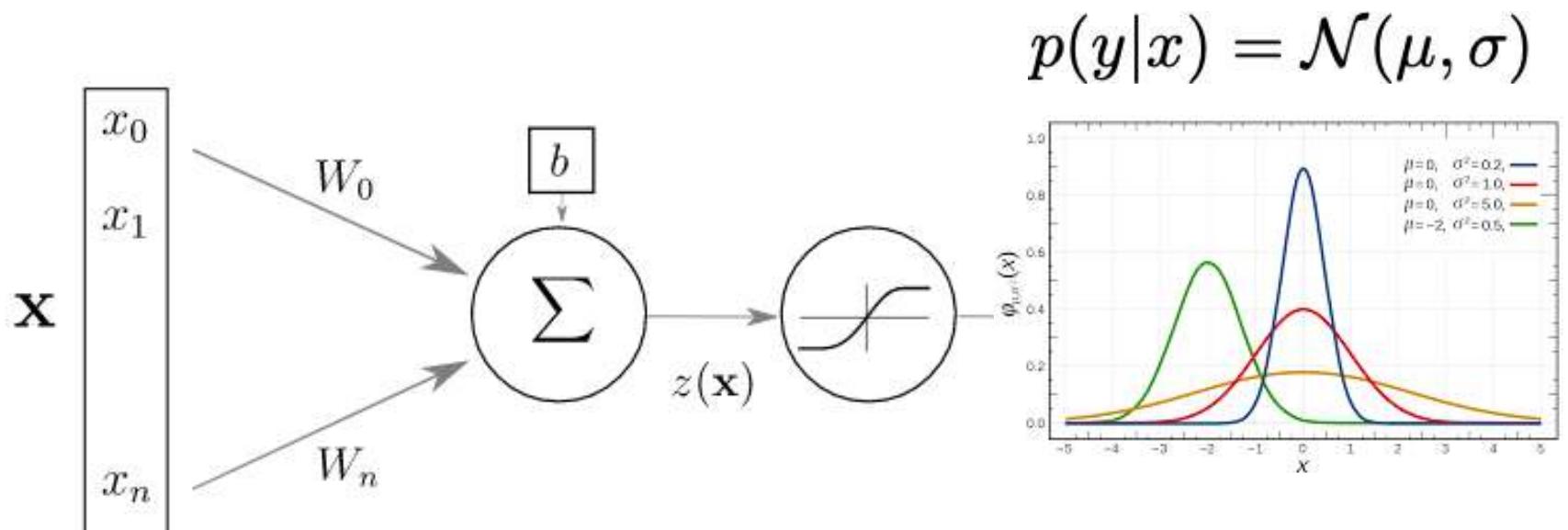
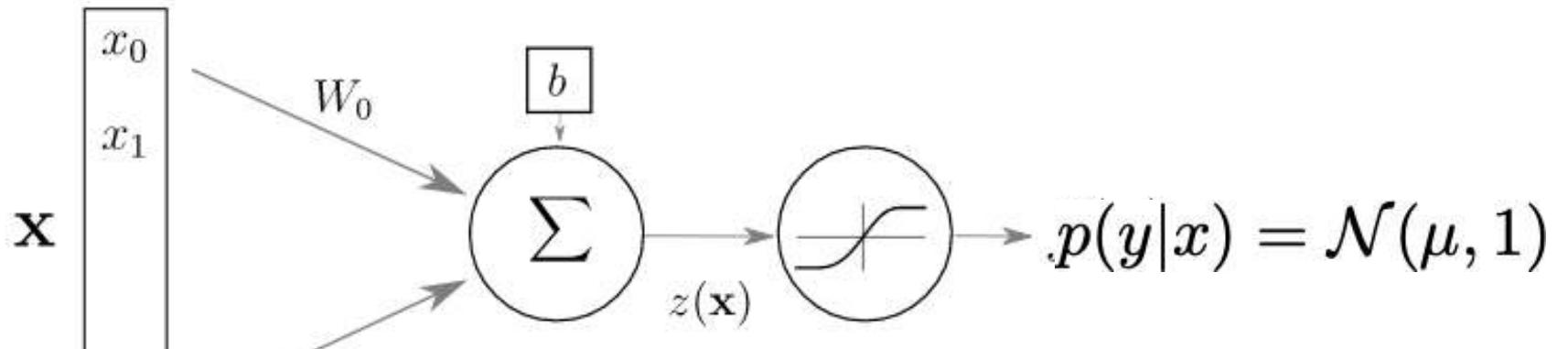
simulations have **advantages over analytic models**

**amortized** inference to infer *billions* of observations

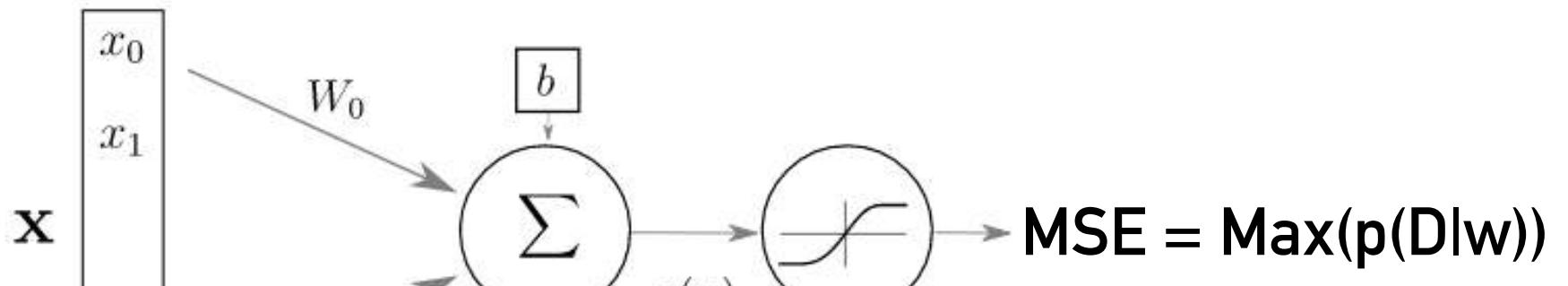
## TWO TYPES OF UNCERTAINTY:

- **Aleatoric (Random)**: Uncertainty coming from the inherent noise in training data. It cannot be reduced if we get more data. Aleatoric uncertainty is covered by the probability distribution used to define the likelihood function:  $p(D|w)$
- **Epistemic (Systematics)**: This kind of uncertainty can be reduced if we get more data. Consequently, epistemic uncertainty is higher in regions of no or little training data and lower in regions of more training data

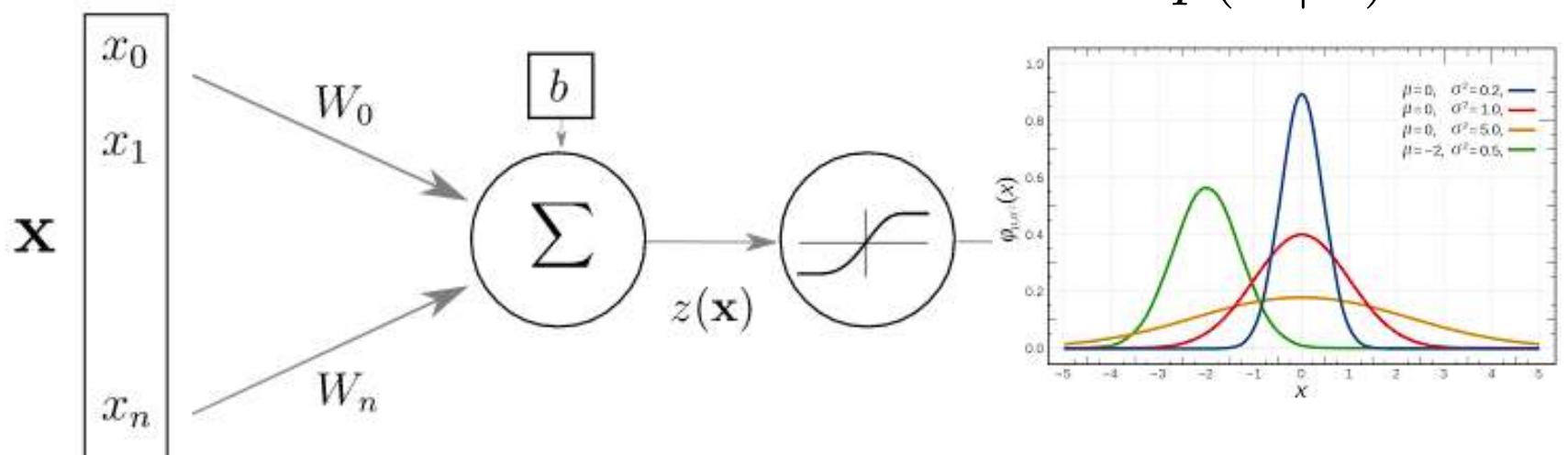
# Mixture Density Network



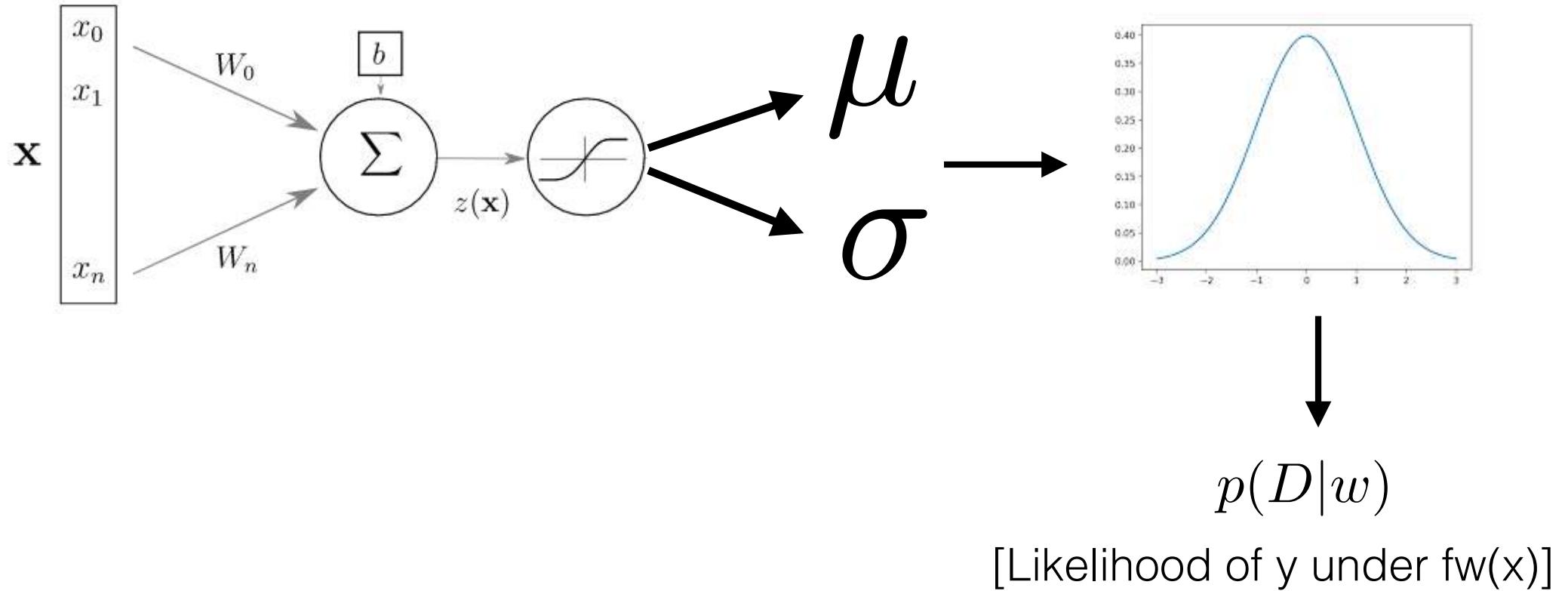
# Mixture Density Network



$$\text{MSE} = \text{Max}(p(D|w))$$

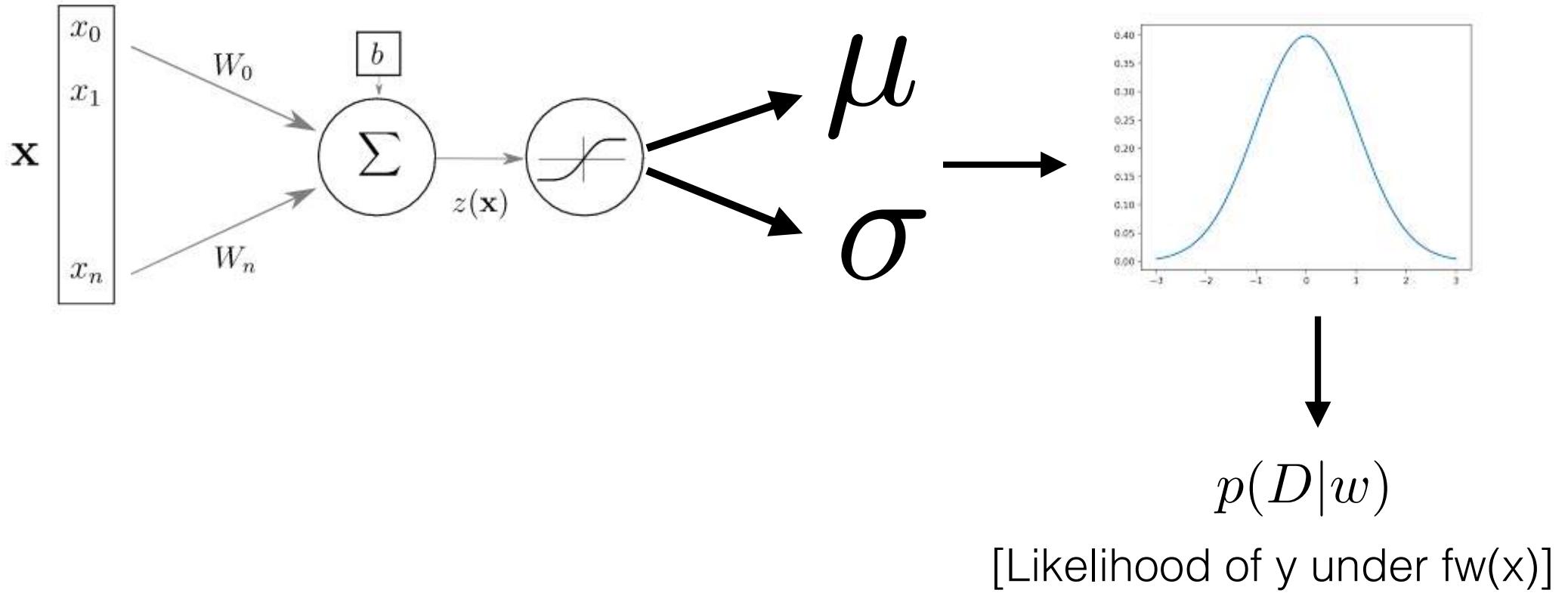


# Mixture Density Network



**Reparametrization trick:** The neural network outputs the parameters of a distribution, on which one can back propagate.

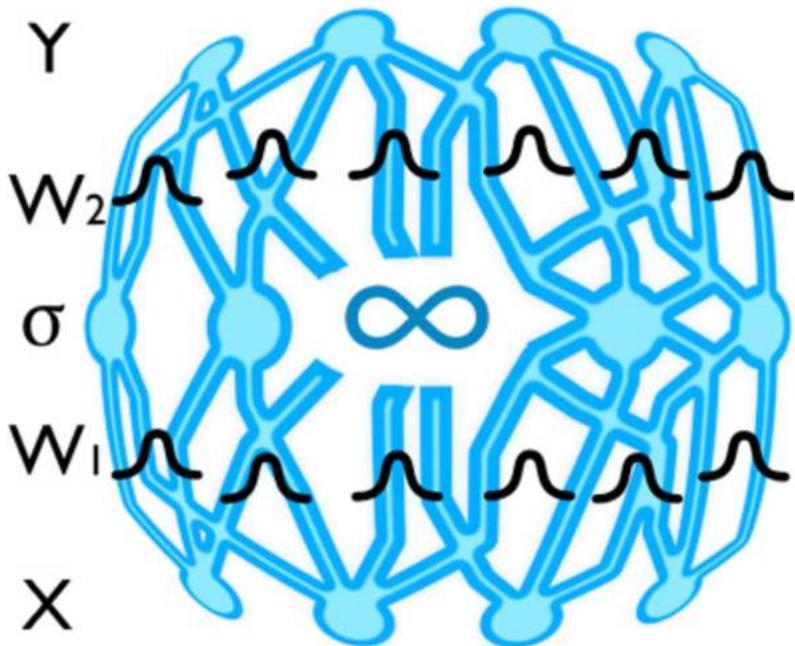
# Probabilistic Neural Network



***The loss function becomes then the max of the (log) likelihood***

## CAPTURING “MODEL UNCERTAINTY”, OR EPISTEMIC UNCERTAINTY

In the previous approach, we are choosing one model among all possible models

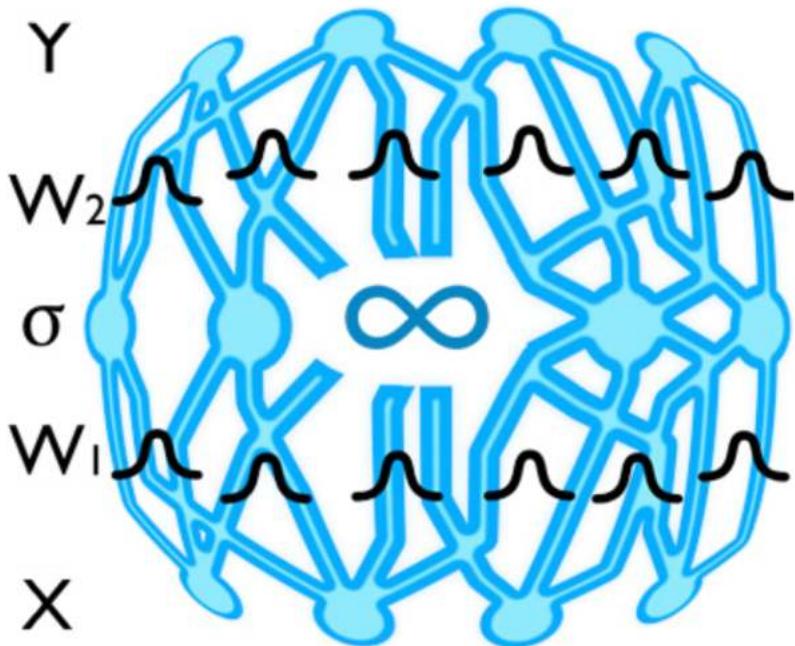


**IDEALLY, WE WOULD LIKE TO MARGINALIZE OVER ALL MODELS AND OVER ALL POSSIBLE WEIGHT VALUES**

$$p(y|x, D) = \int p(y|x, W) \cdot p(W|D) dW$$

## CAPTURING “MODEL UNCERTAINTY”, OR EPISTEMIC UNCERTAINTY

In the previous approach, we are choosing one model among all possible models



**IDEALLY, WE WOULD LIKE TO MARGINALIZE OVER ALL MODELS AND OVER ALL POSSIBLE WEIGHT VALUES**

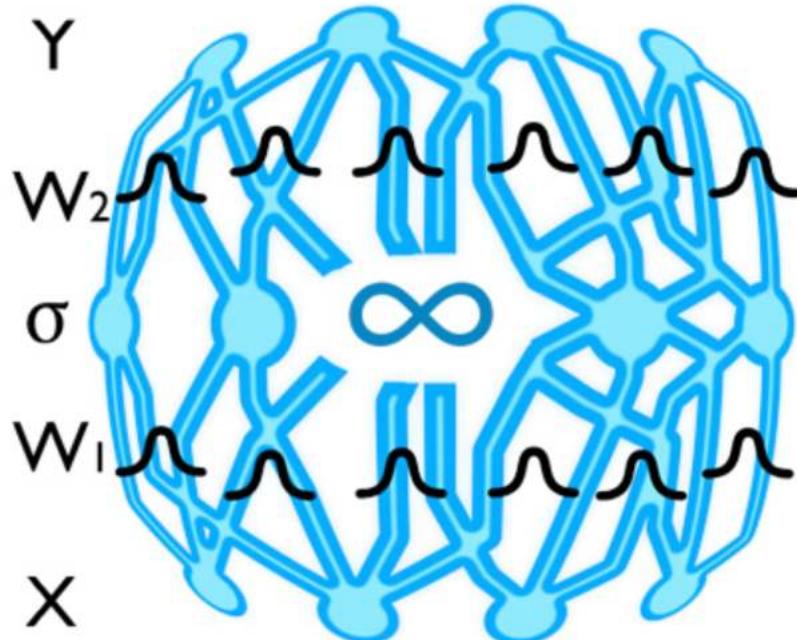
$$p(y|x, D) = \int p(y|x, W) \cdot p(W|D) dW$$



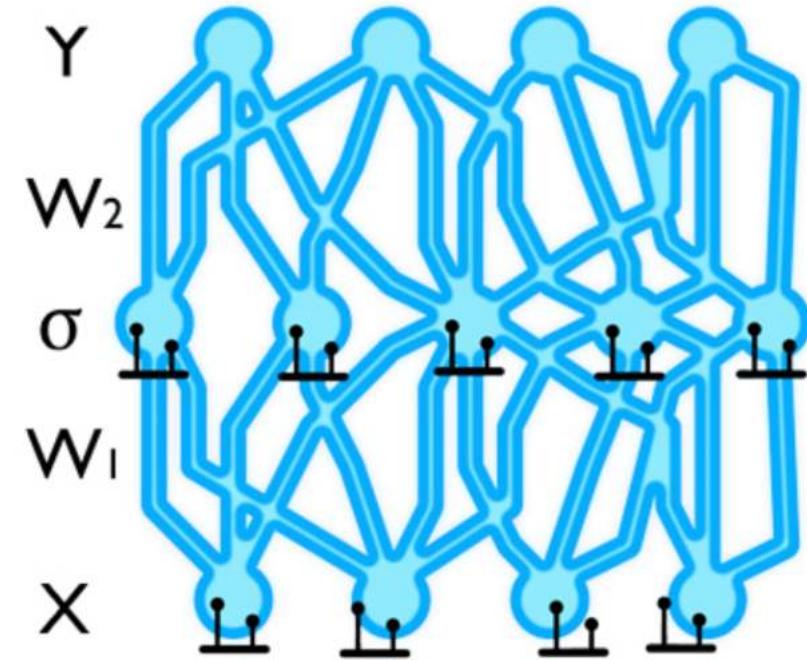
This is typically intractable

# DROPOUT AS EPISTEMIC UNCERTAINTY ESTIMATION

Denker&Lecun91, Neal+95, Graves+11, Kingma+15, Gal+15...



**BNNs ADD A PRIOR DISTRIBUTION TO EACH WEIGHT - HARD TO TRAIN**



**GAL+15 SHOW THAT DROPOUT CAN BE USED TO ESTIMATE UNCERTAINTY**

# DROPOUT AS EPISTEMIC UNCERTAINTY ESTIMATION

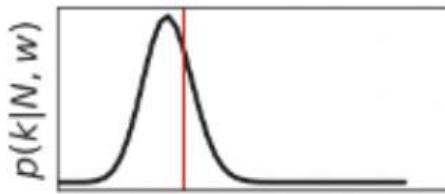
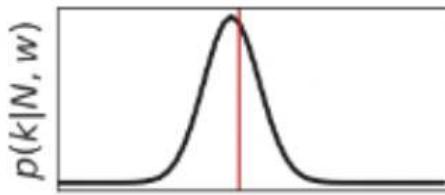
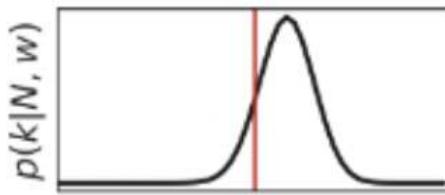
We can treat the many different networks (with different neurons dropped out) as Monte Carlo samples from the space of all available models. This provides mathematical grounds to reason about the model's uncertainty and, as it turns out, often improves its performance.

We simply apply dropout at test time, that's all! Then, instead of one prediction, we get many, one by each model. We can then average them or analyze their distributions.

## Aleatoric

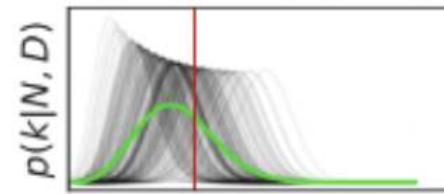
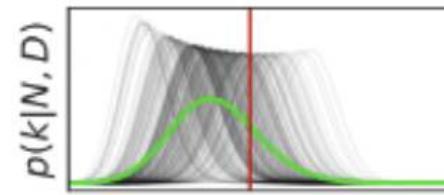
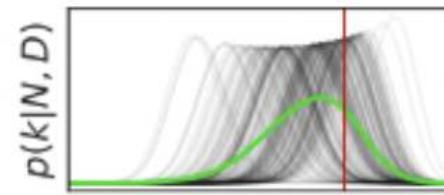
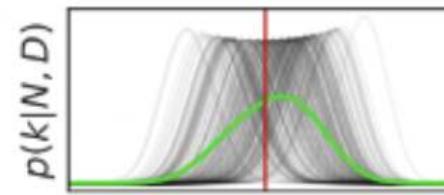


— Posterior  
— Observed

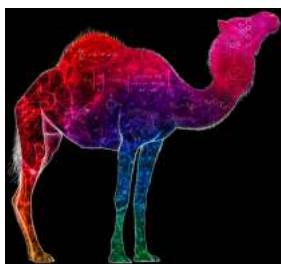
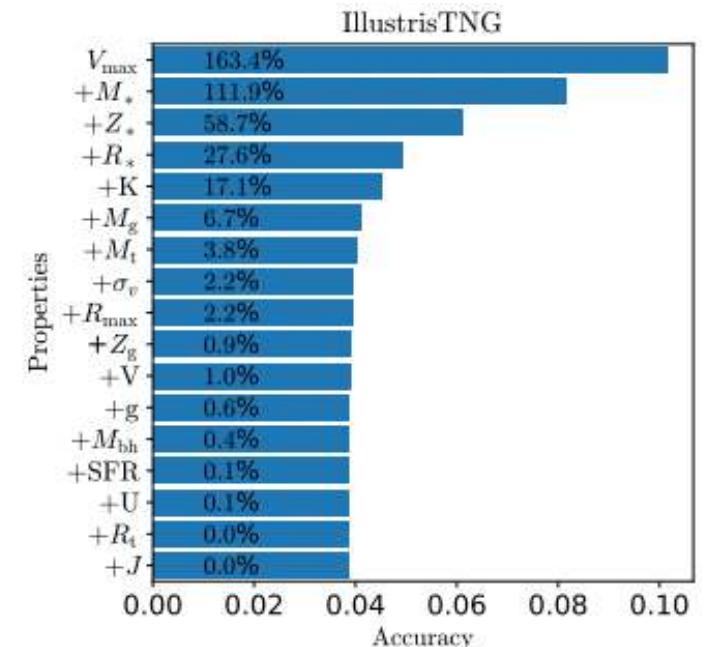
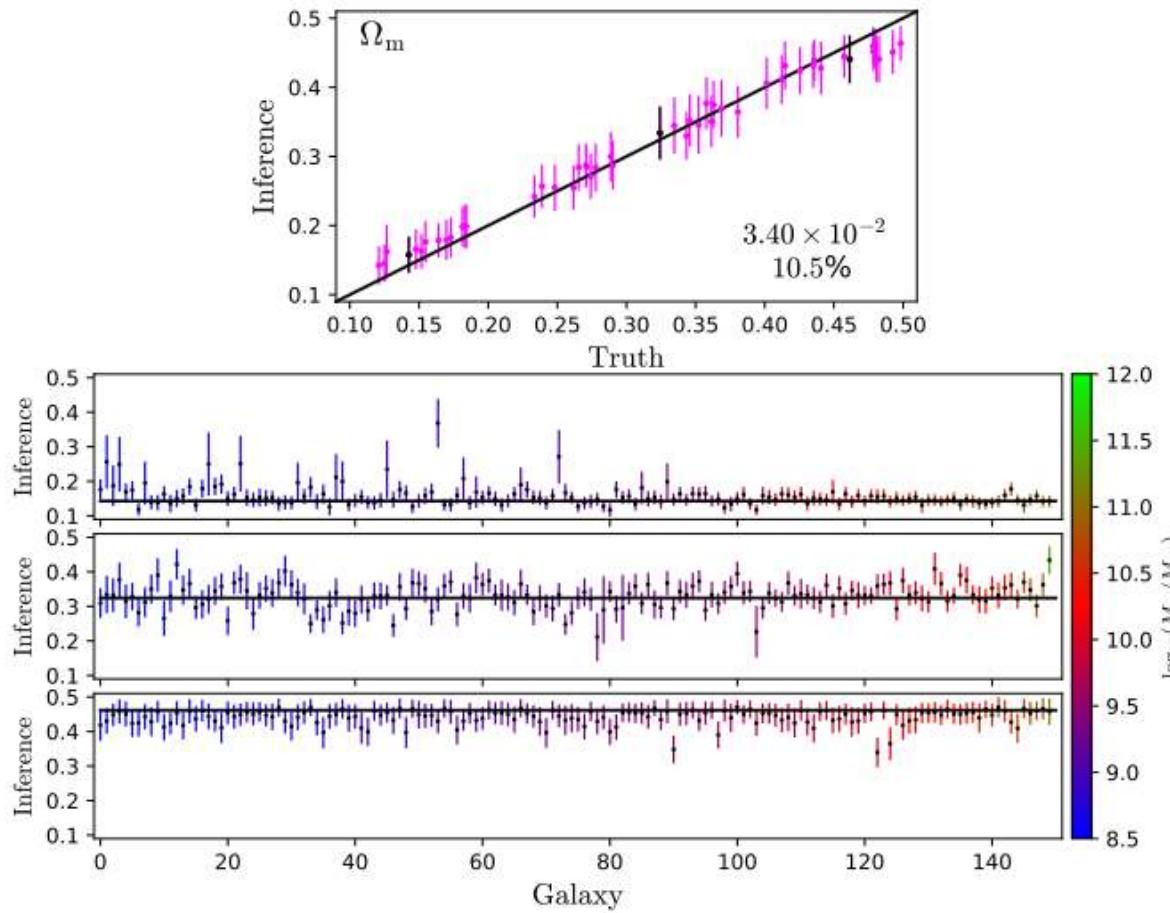


## Aleatoric+Epistemic

— Model Posteriors  
— Posterior  
— Observed



# Cosmological Inference: beyond summary statistics



(Cosmology and Astrophysics with Machine Learning Simulations)

**CAMELS**

**Villaescusa-Navarro+22**

Ravanbakhsh+17, Brehmer+19, Ribli+19, Pan+19, Ntampaka+19, Alexander+20, Arjona+20, Coogan+20, Escamilla-Rivera+20, Hortua+20, Vama+20, Vernardos+20, Wang+20, Mao+20, Arico+20, Villaescusa\\_navarro+20, Singh+20, Park+21, Modi+21, Villaescusa-Navarro+21ab, Moriwaki+21, DeRose+21, Makinen+21, Villaescusa-Navarro+22