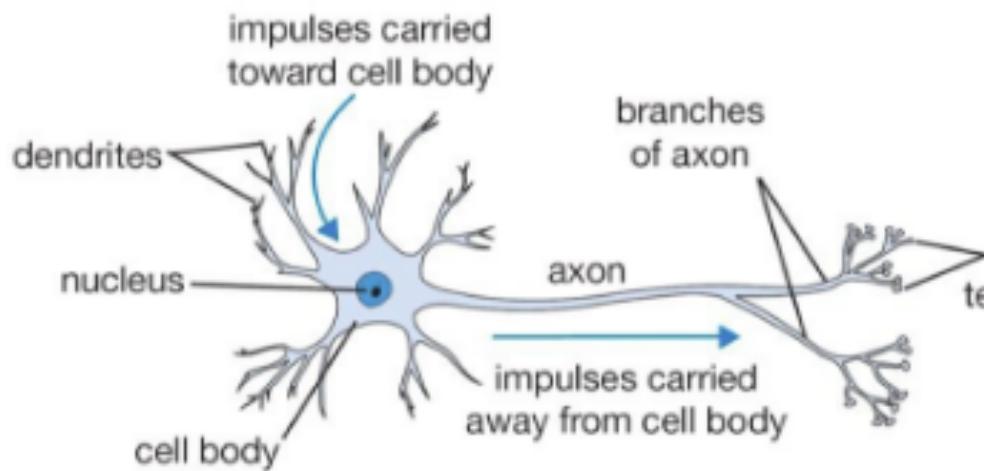


PART II: FOUNDATIONS OF “SHALLOW” NEURAL NETWORKS

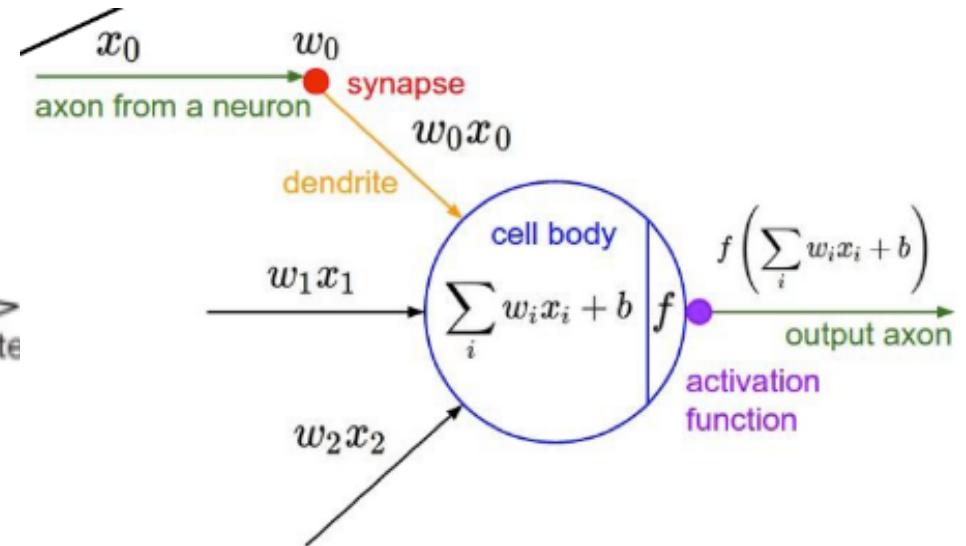
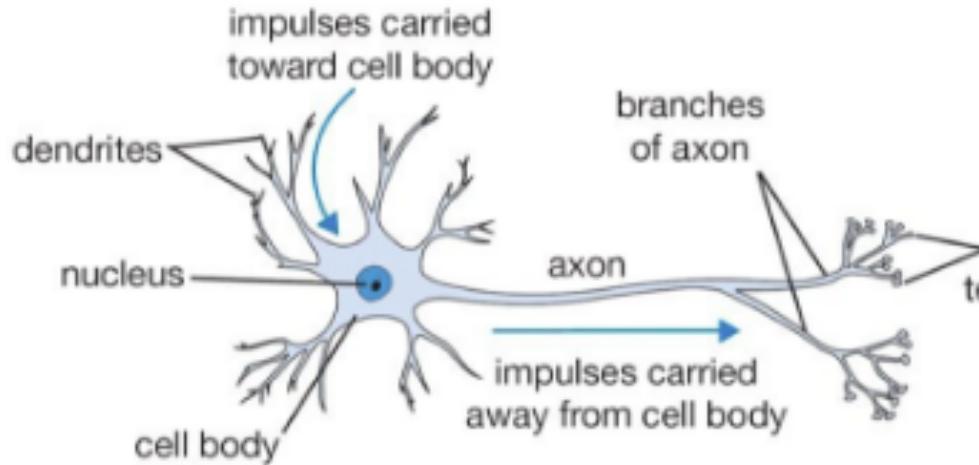
THE NEURON



INSPIRED BY NEURO - SCIENCE?

Credit: Karpathy

THE NEURON



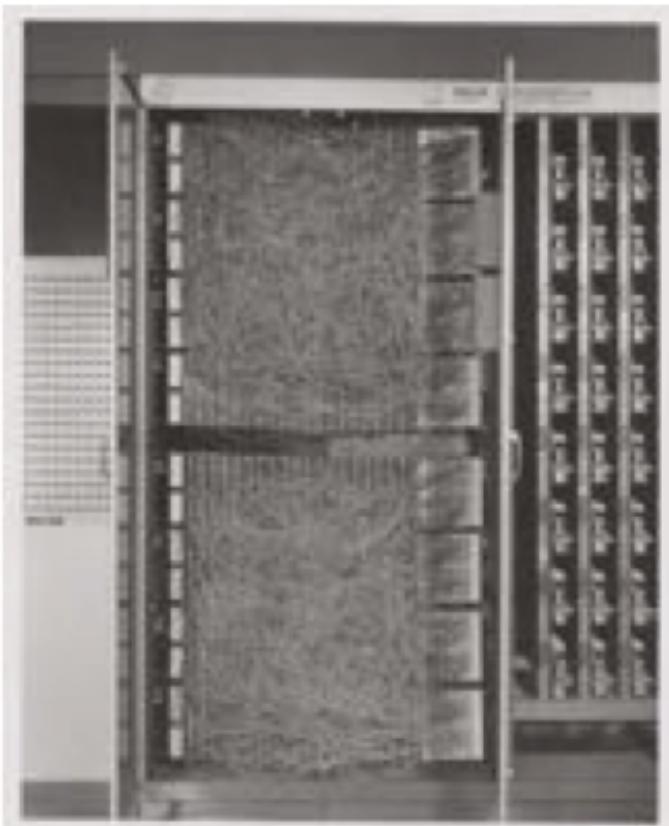
INSPIRED BY NEURO - SCIENCE?

Credit: Karpathy

Rosenblatt Perceptron

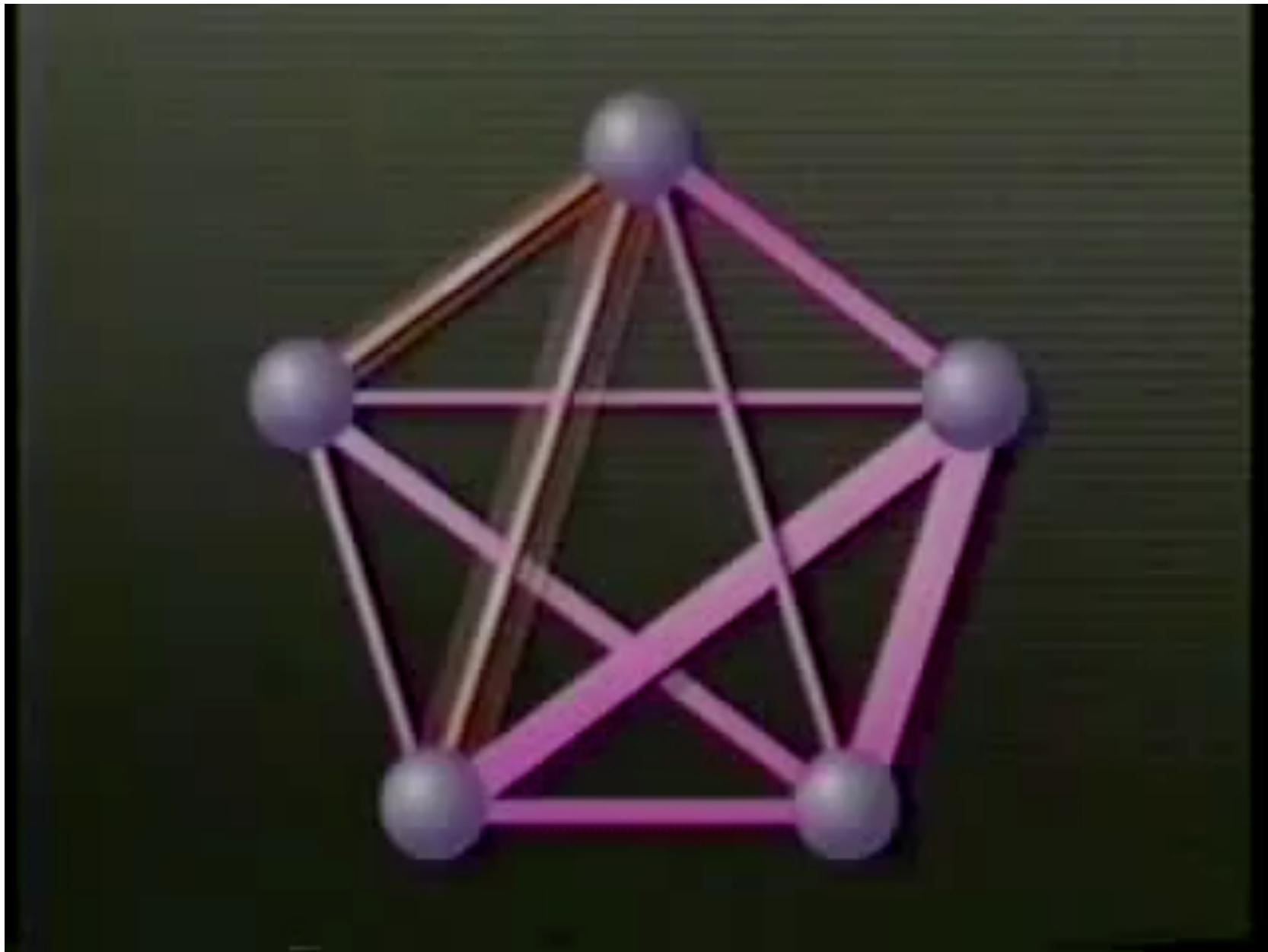
FIRST IMPLEMENTATION OF NEURAL NETWORK [Rosenblatt, 1957!]

INTENDED TO BE A MACHINE (NOT AN ALGORITHM)

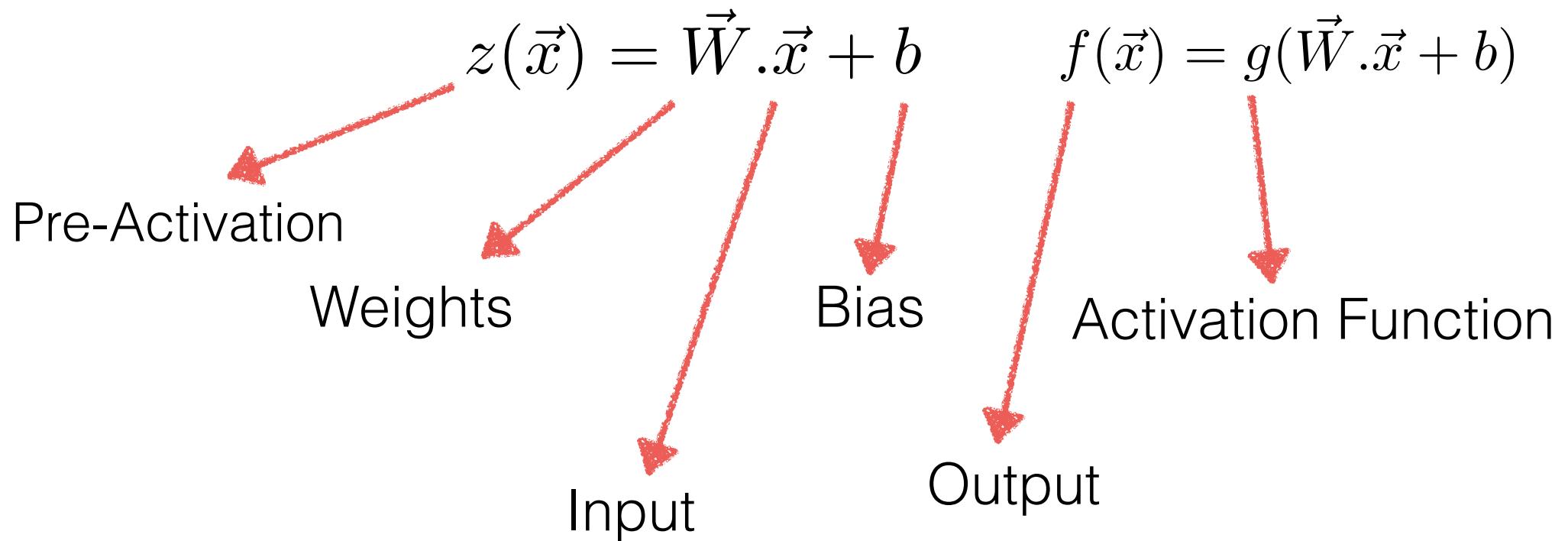
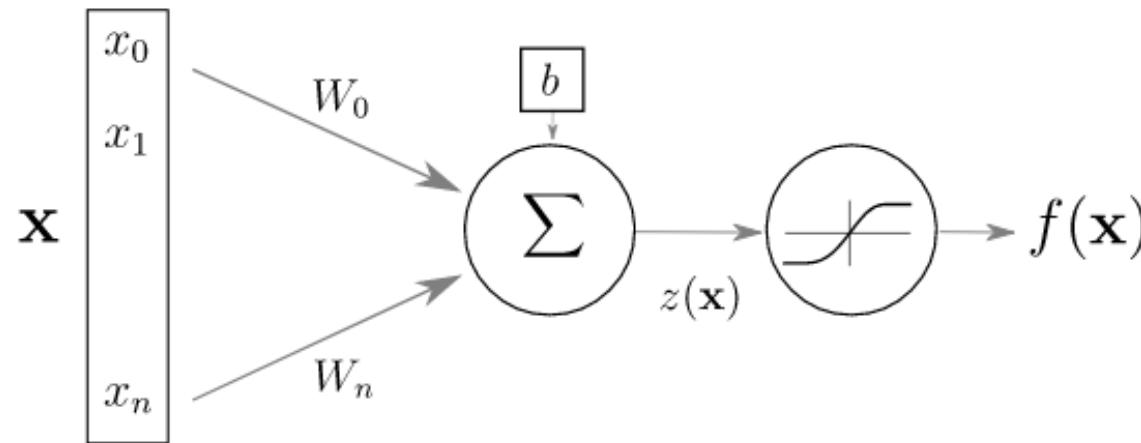


it had an array of 400 photocells, randomly connected to the "neurons".

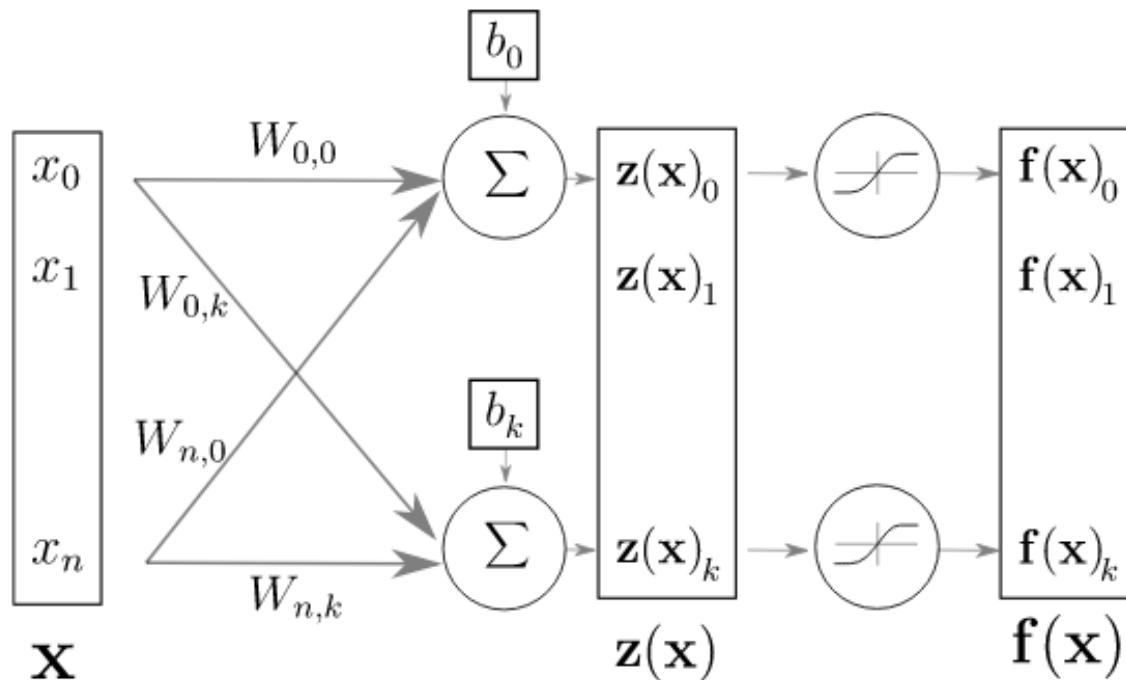
Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors



TODAY'S ARTIFICIAL NEURON



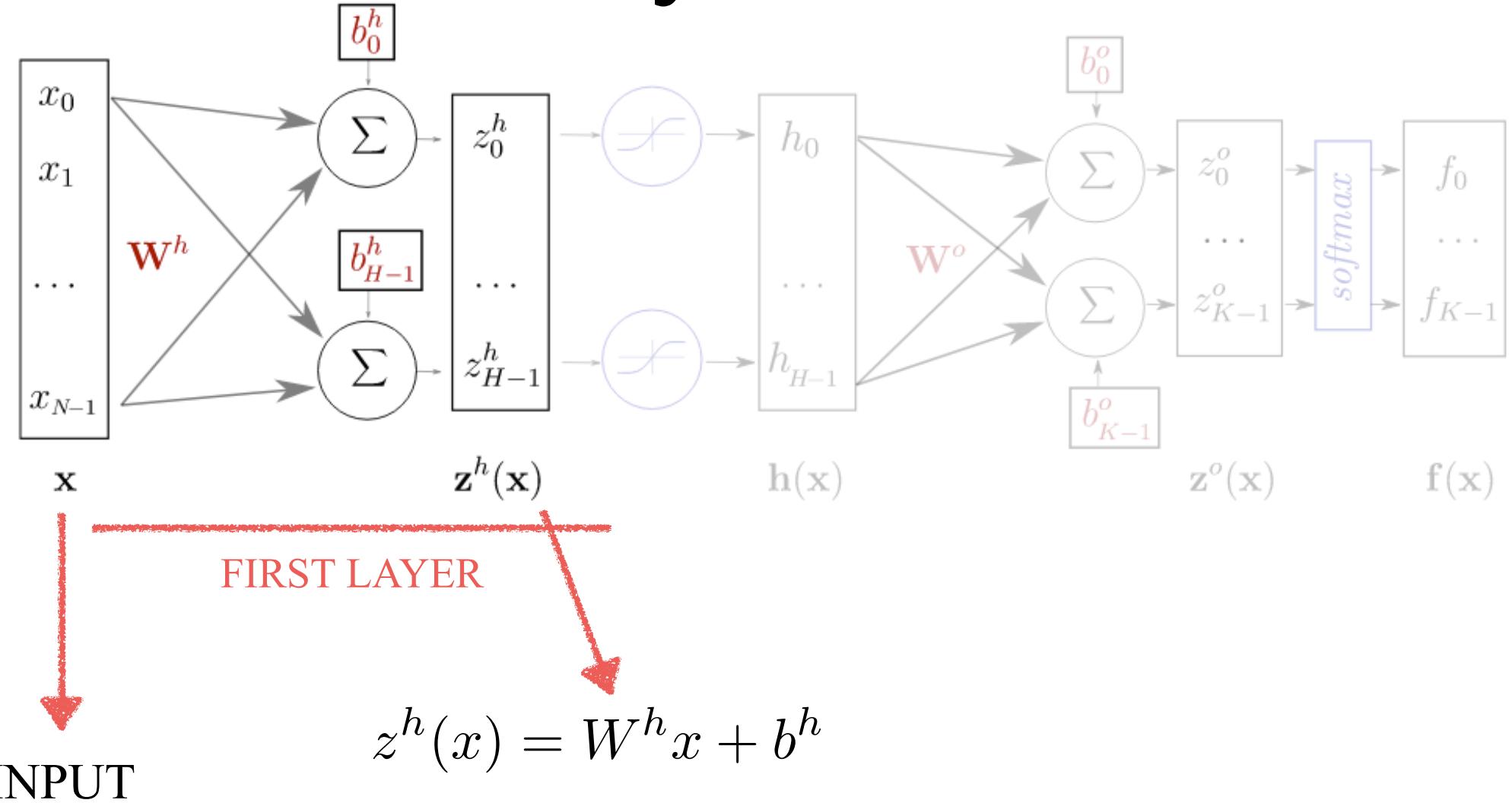
LAYER OF NEURONS



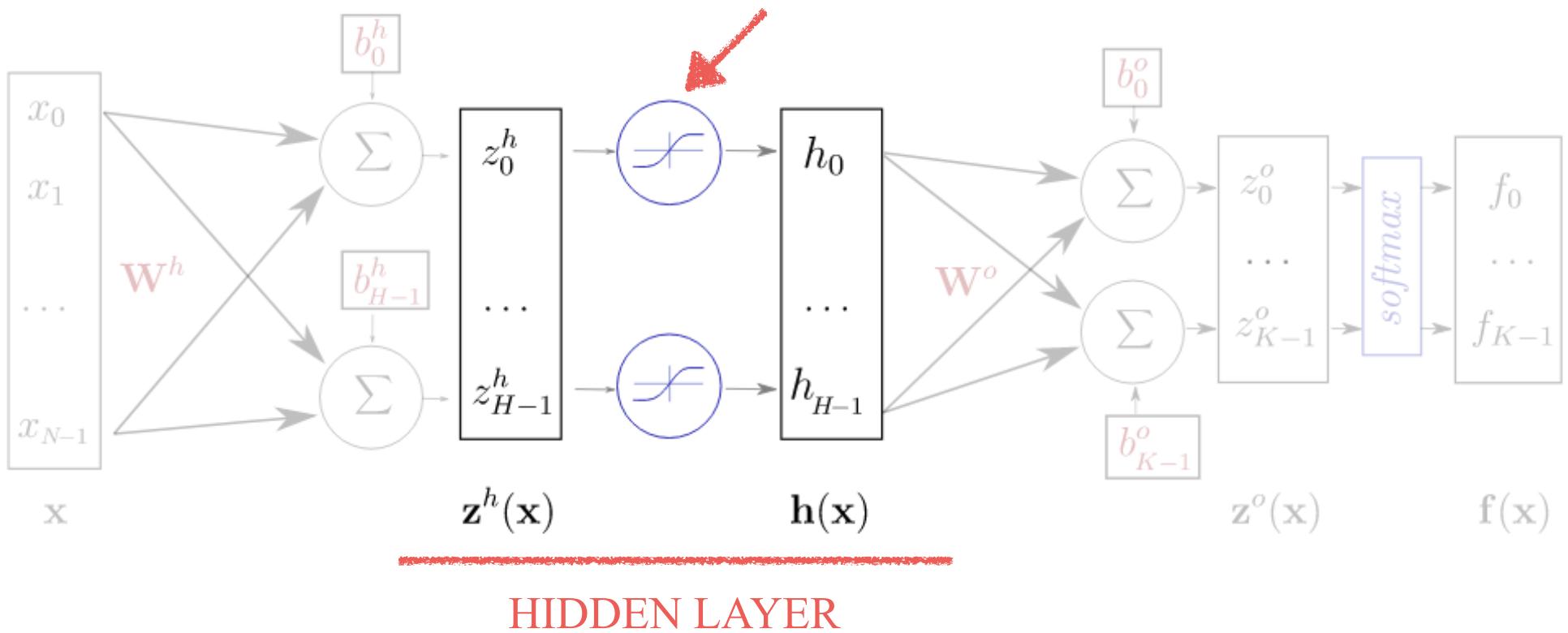
$$f(\vec{x}) = g(\mathbf{W} \cdot \vec{x} + \vec{b})$$

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

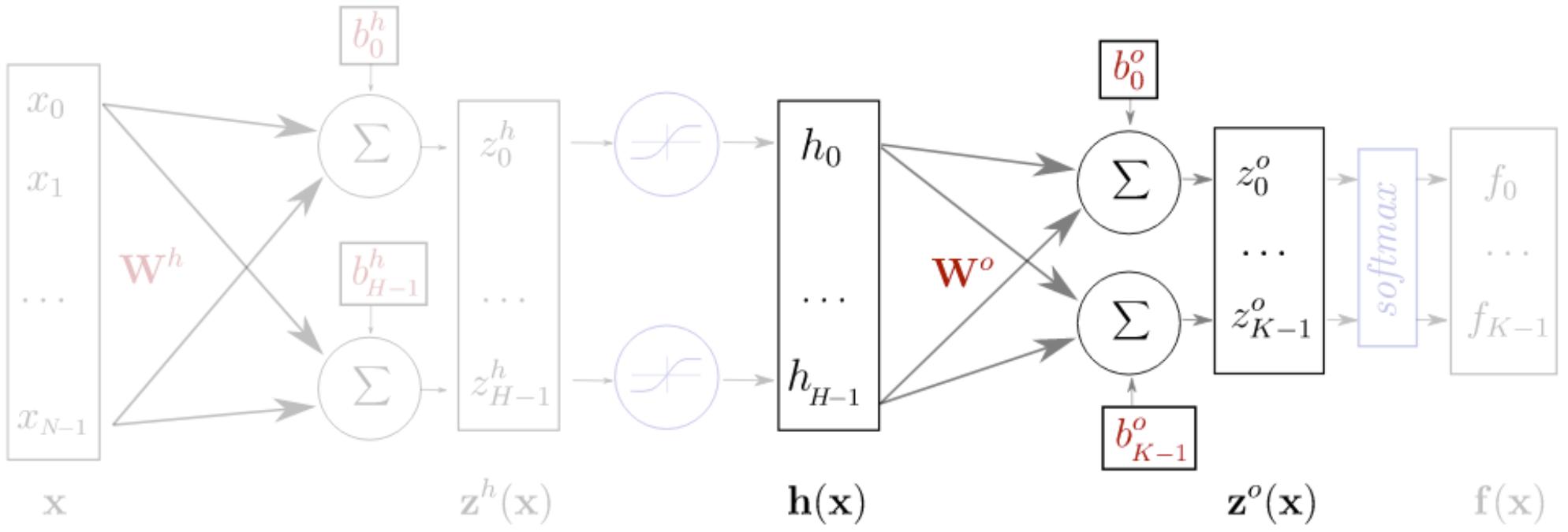
Hidden Layers of Neurons



ACTIVATION FUNCTION

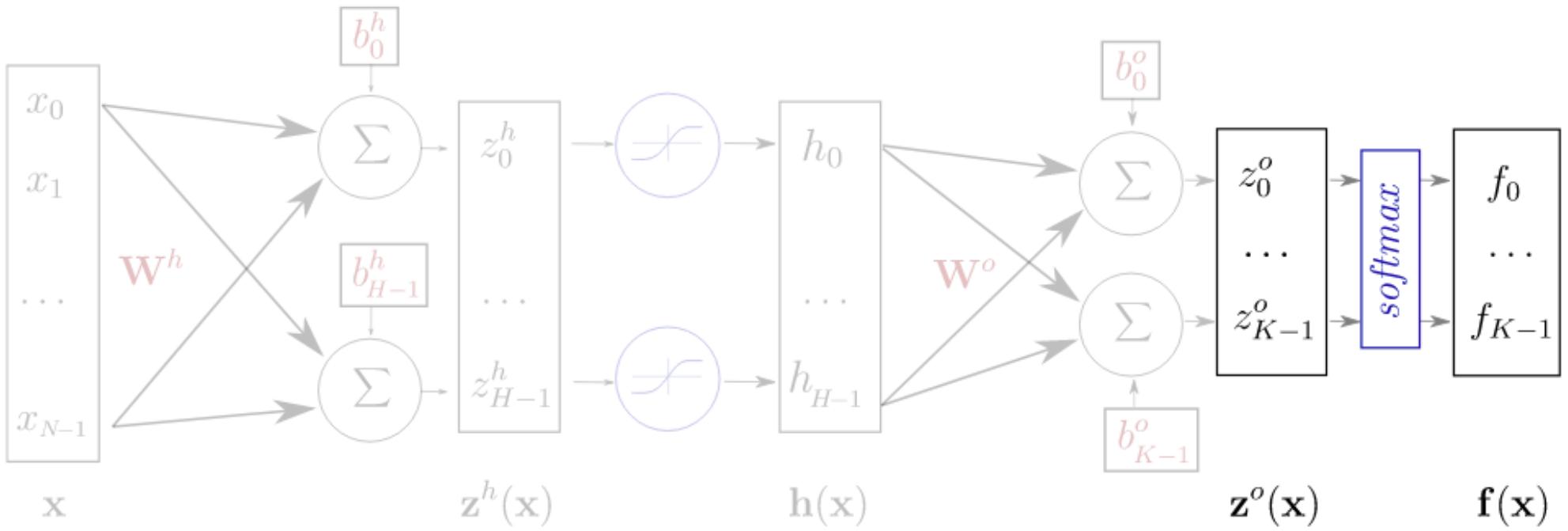


$$h(x) = g(z^h(x)) = g(W^h x + b^h)$$



OUTPUT LAYER

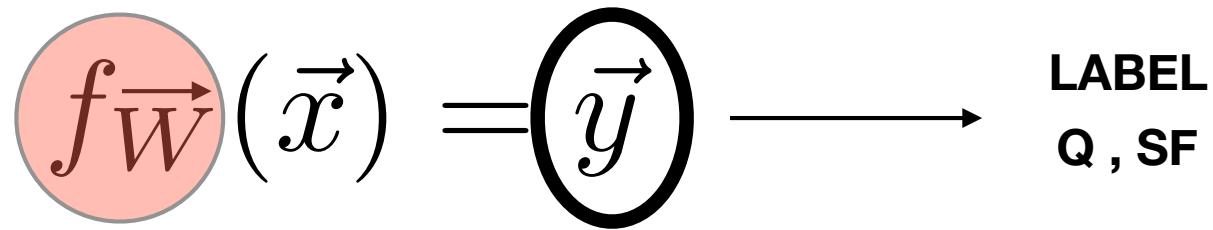
$$z^0(\mathbf{x}) = W^0 h(\mathbf{x}) + b^0$$



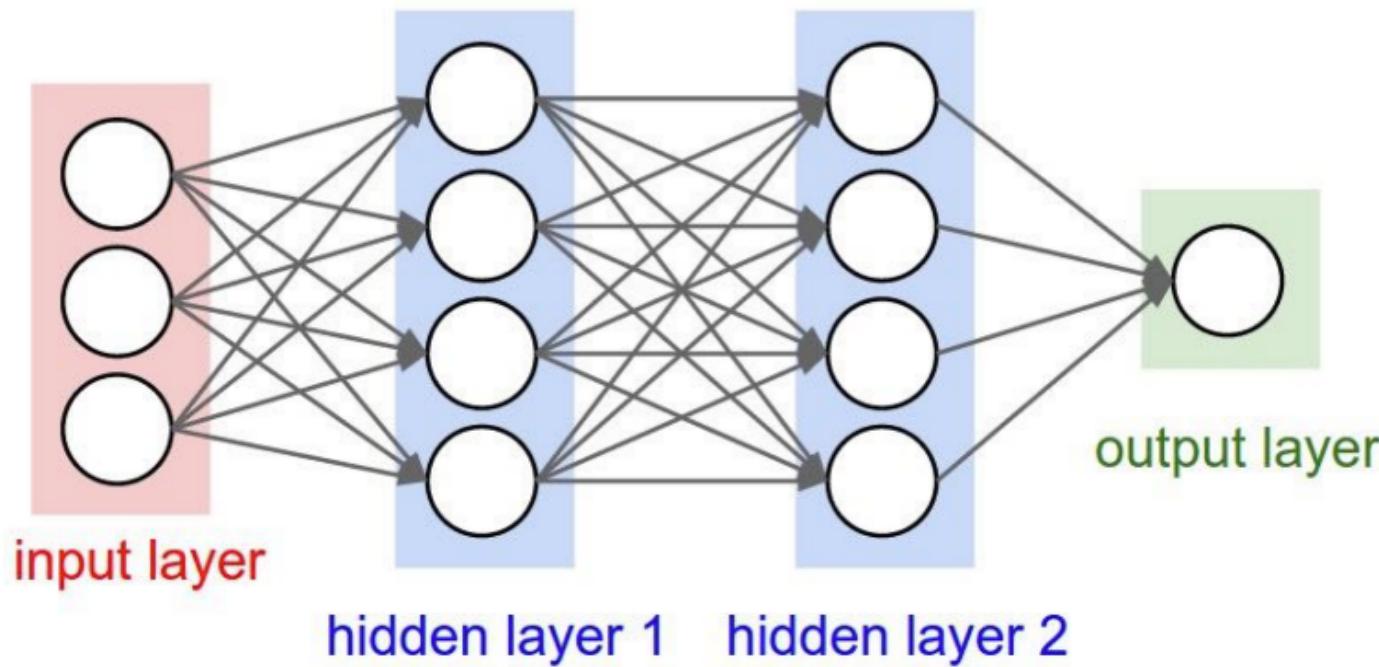
PREDICTION LAYER

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

**"CLASSICAL"
MACHINE LEARNING**



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

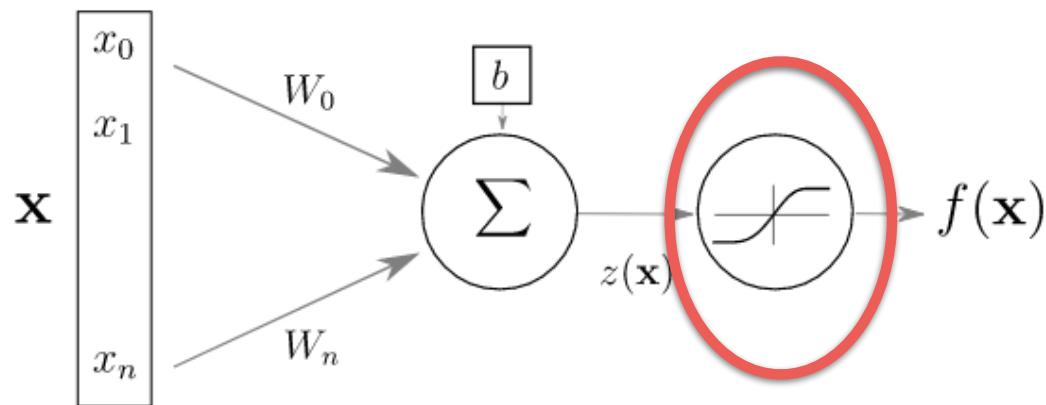


$$p = g_3(W_3g_2(W_2g_1(W_1\vec{x}_0))) \xleftarrow{\text{NETWORK FUNCTION}}$$

SO LET'S GO DEEPER AND DEEPER!

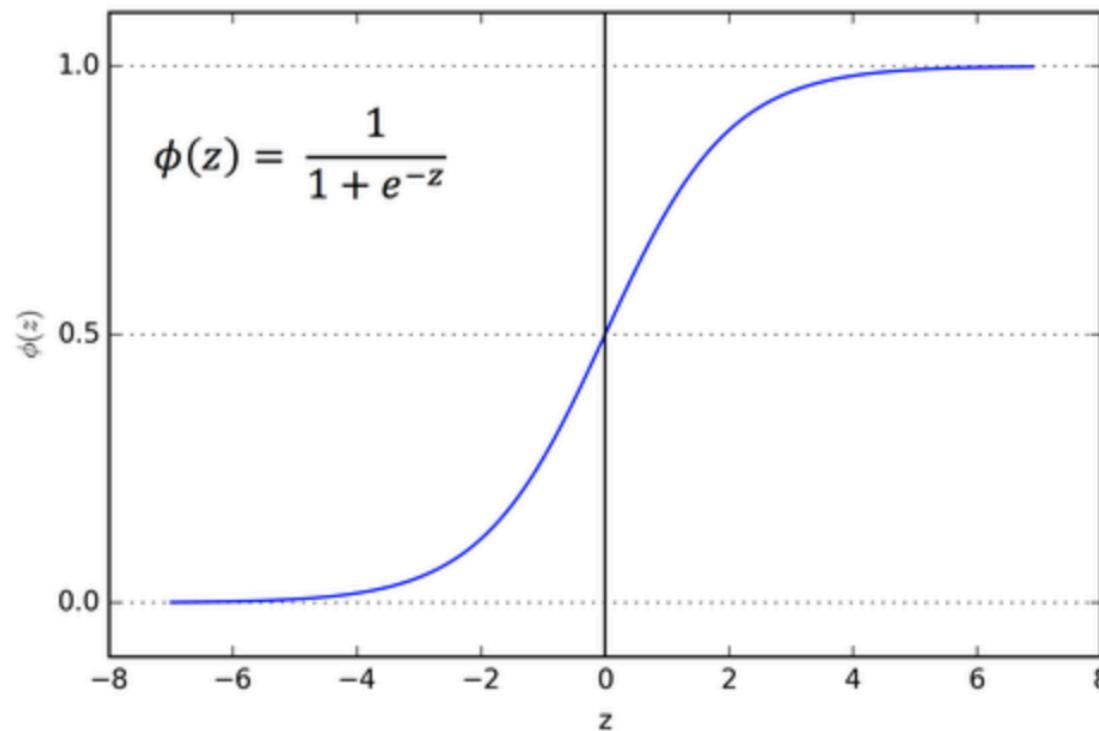
LET'S FIRST EXAMINE IN MORE DETAIL HOW SIMPLE
“SHALLOW” NETWORKS WORK

ACTIVATION FUNCTIONS?

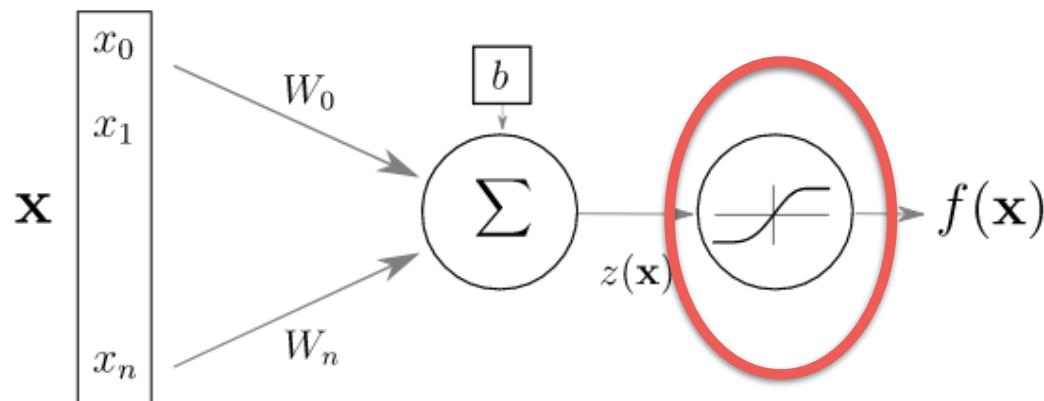


ADD NON LINEARITIES TO THE PROCESS

ACTIVATION FUNCTIONS?

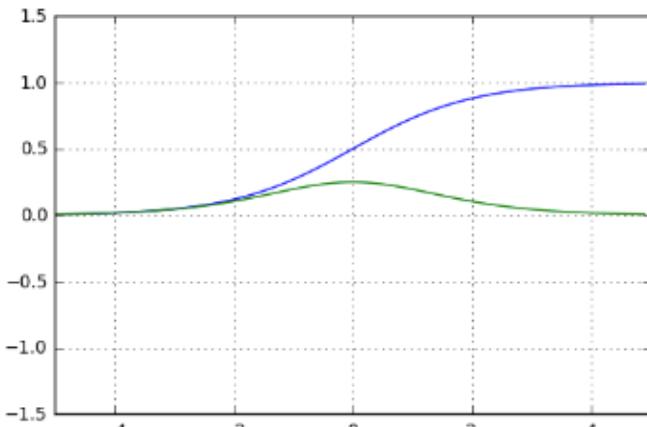


the sigmoid function



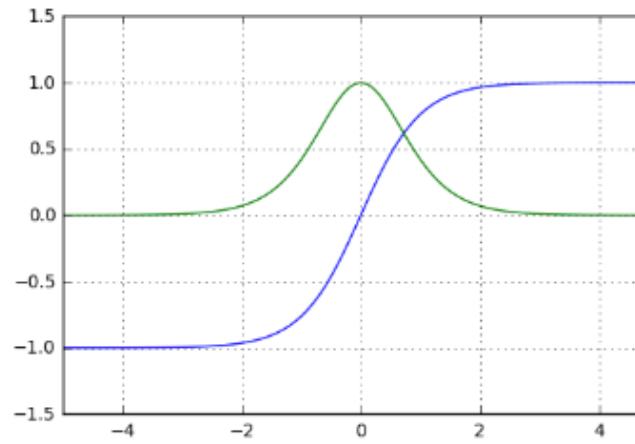
ADD NON LINEARITIES TO THE PROCESS

ACTIVATION FUNCTIONS



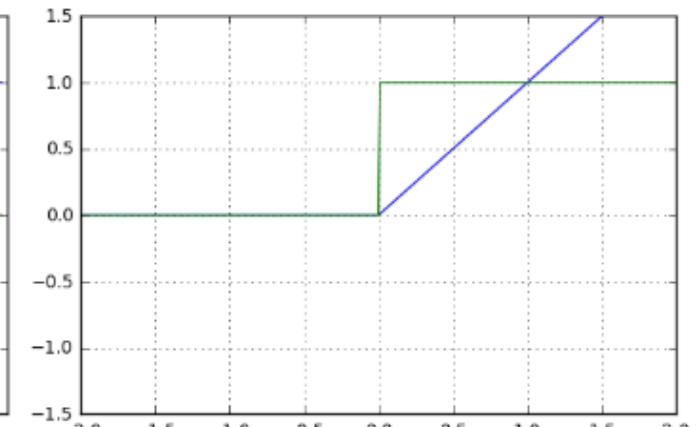
$$\text{sigm}(x) = \frac{1}{1 + e^{-x}}$$

$$\text{sigm}'(x) = \text{sigm}(x)(1 - \text{sigm}(x))$$



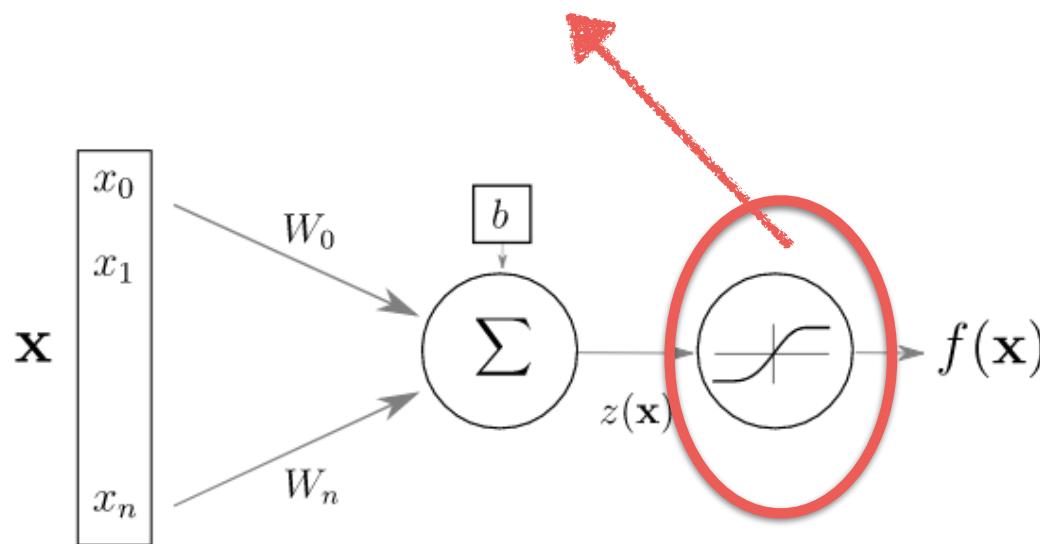
$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$\tanh'(x) = 1 - \tanh(x)^2$$



$$\text{relu}(x) = \max(0, x)$$

$$\text{relu}'(x) = 1_{x>0}$$



AND REMEMBER WE NEED A LOSS FUNCTION ...

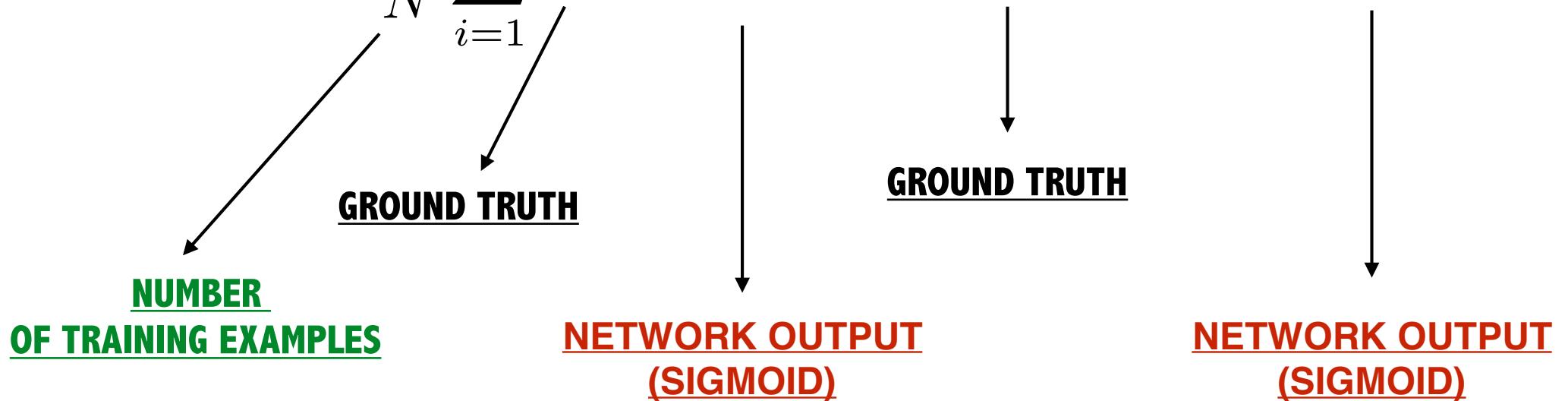
AND REMEMBER WE NEED A LOSS FUNCTION ...

LET'S SART WITH A SIMPLE CLASSIFICATION PROBLEM

BINARY CLASSIFICATION LOSS

WE TYPICALLY USE THE BINARY CROSS-ENTROPY LOSS:

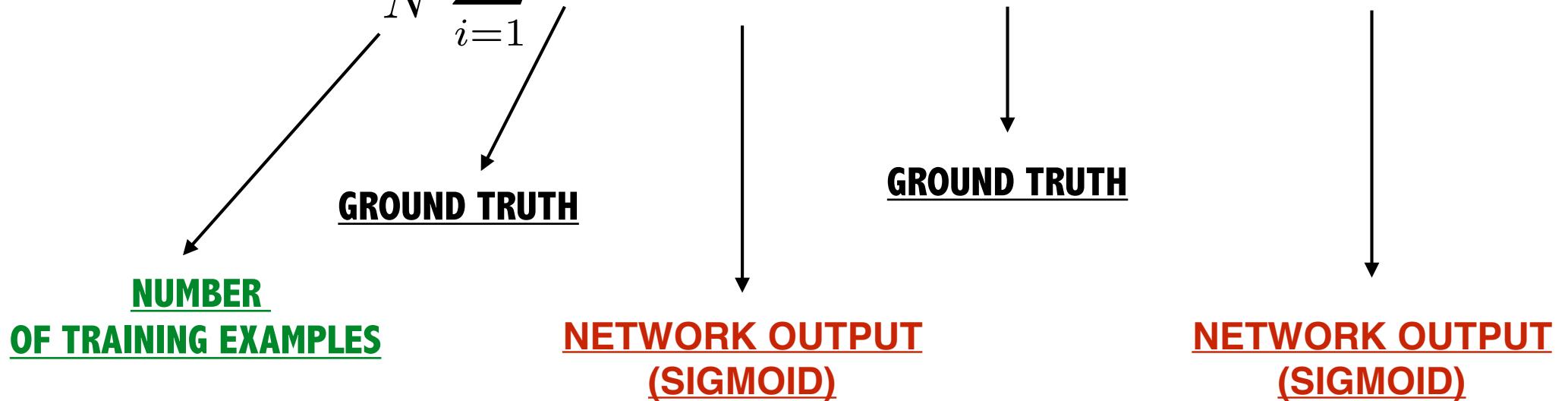
$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i))$$



BINARY CLASSIFICATION LOSS

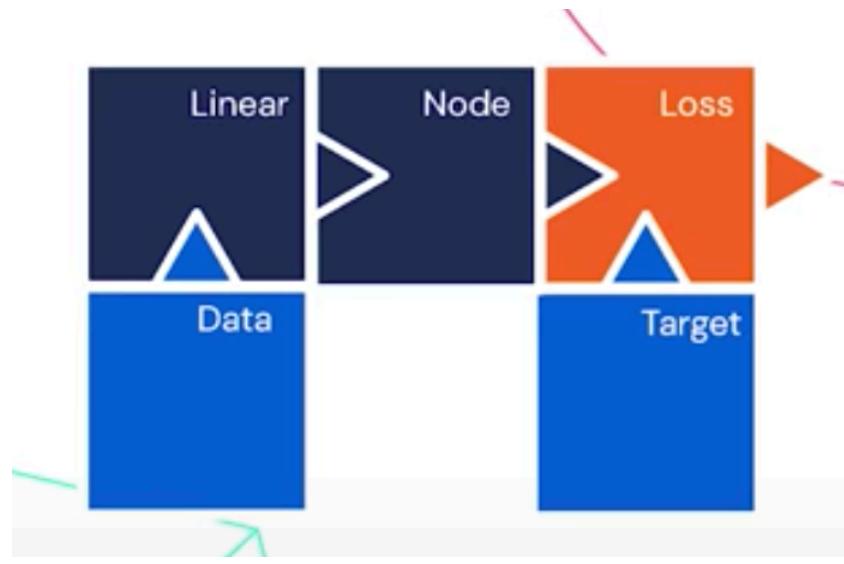
WE MINIMIZE THE CROSS ENTROPY BETWEEN THE TRUE $[q(y)]$ AND PREDICTED $[p(y)]$ DISTRIBUTIONS

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



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

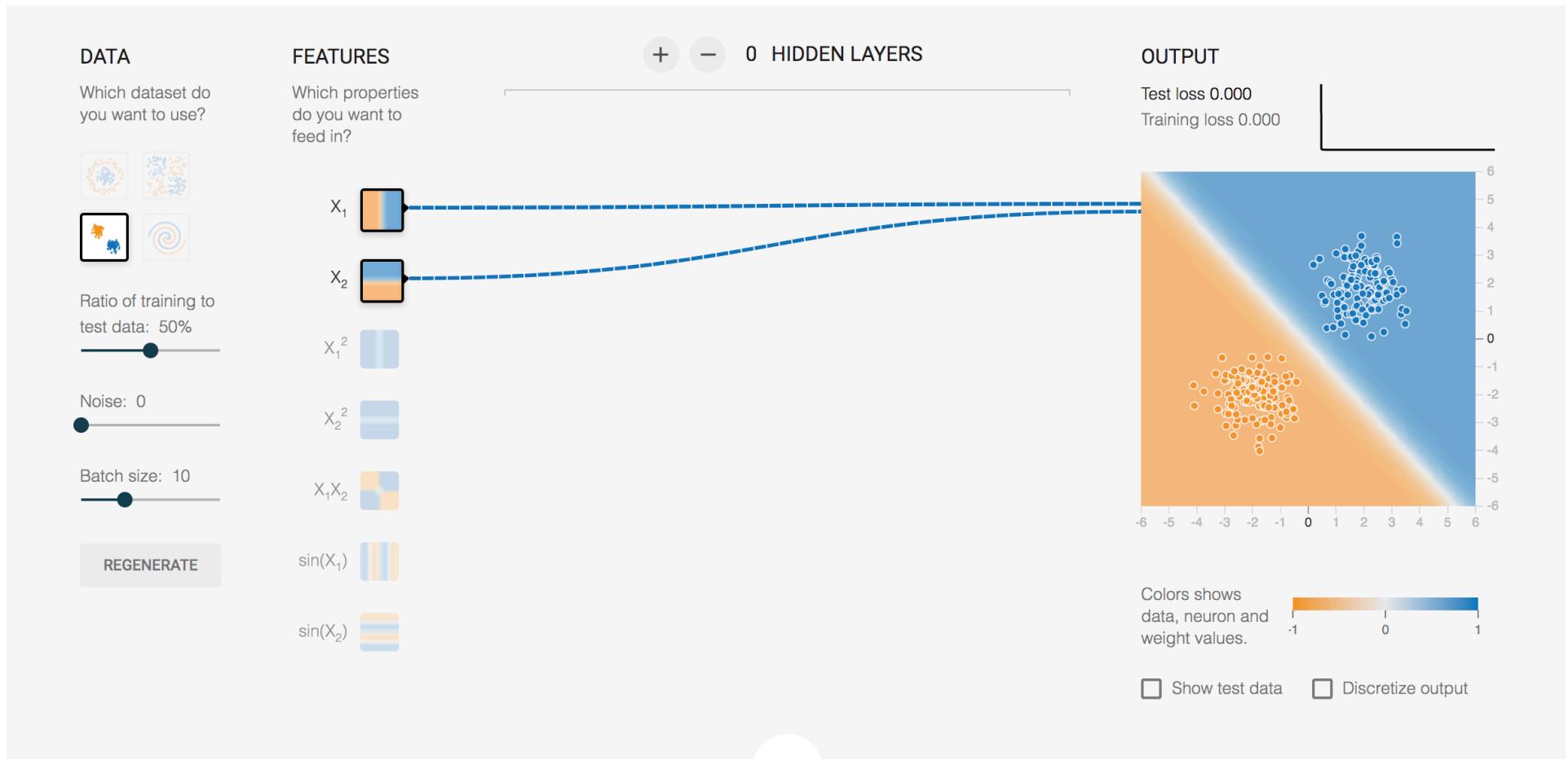
1. LINEAR NEURON OR UNIT
2. NON LINEARITIES (ACTIVATION FUNCTION)
3. LOSS FUNCTION



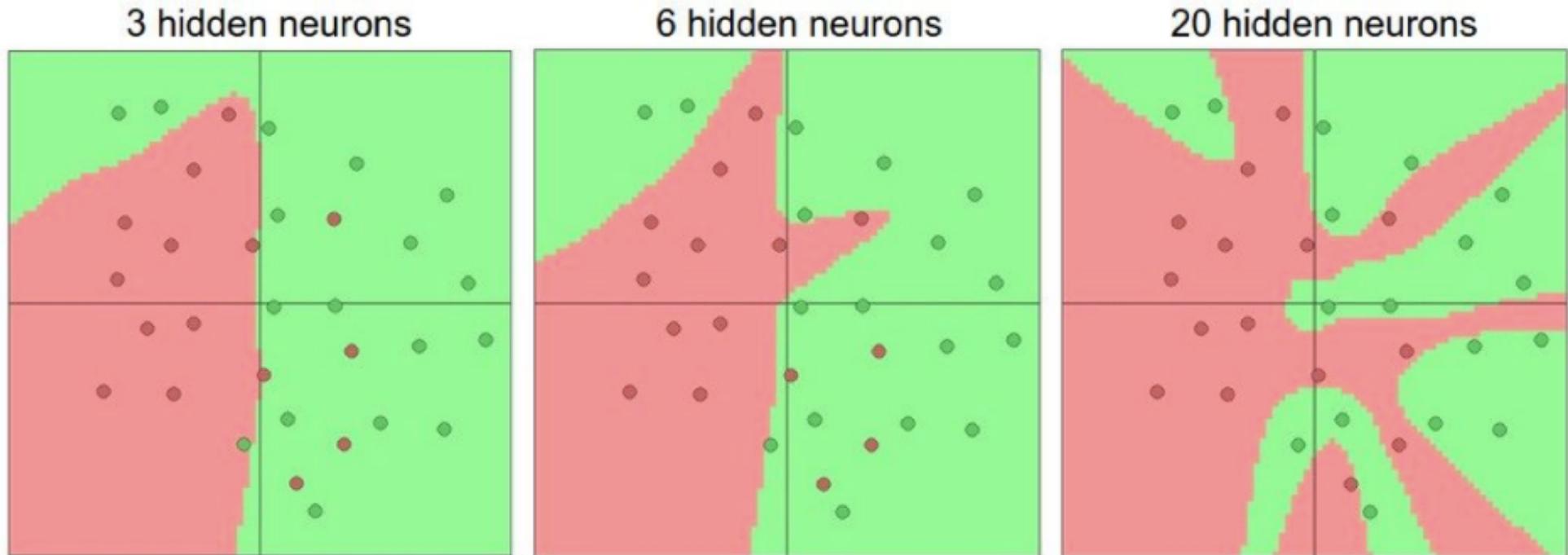
(deepmind
@Czarnecki)

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

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



HIDDEN LAYERS ALLOW INCREASING COMPLEXITY



More complex functions allow increasing complexity

Credit: Karpathy

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 OBTAINES AT MOST EPSILON ERROR IN FUNCTIONAL SPACE

Cybenko+89

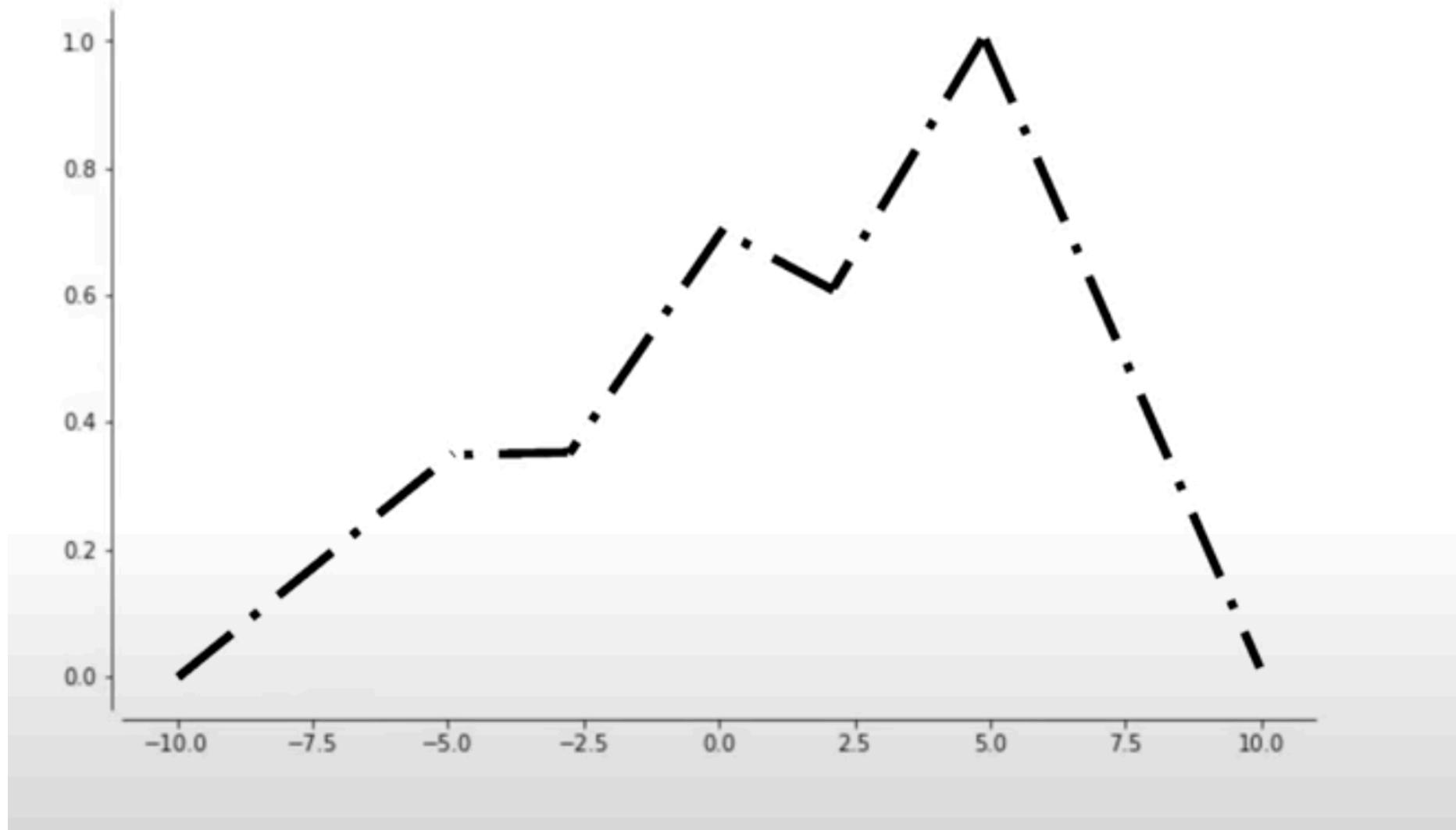
“BIG ENOUGH NETWORK CAN APPROXIMATE, BUT NOT REPRESENT ANY SMOOTH FUNCTION. THE MATH DEMONSTRATION IMPLIES SHOWING THAT NETWORS ARE DENSE IN THE SPACE OF TARGET FUNCTIONS”

UNIVERSAL APPROXIMATION THEOREM

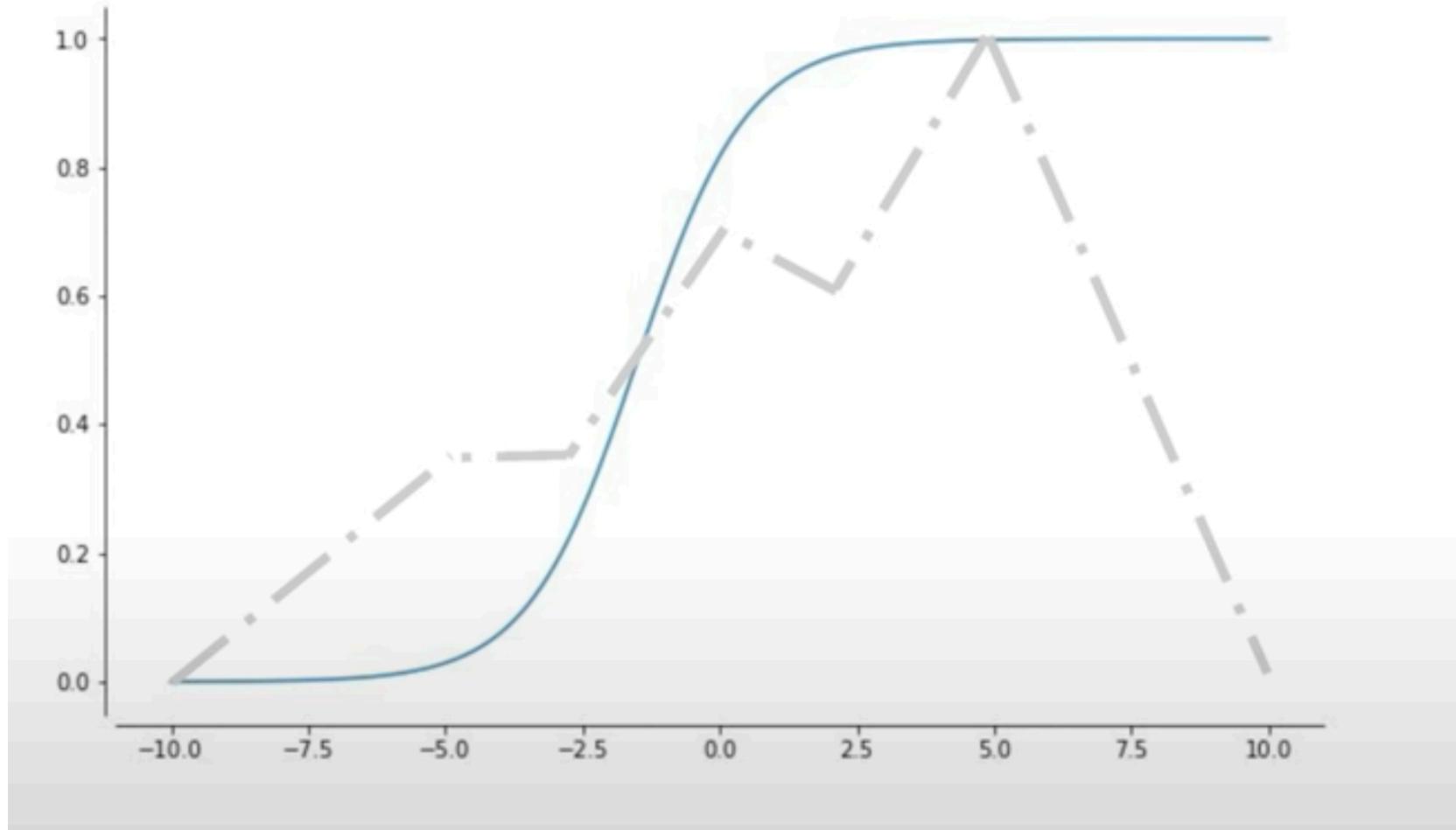
FOR ANY CONTINUOS FUNCTION FOR A HYPERCUBE $[0,1]^d$ TO REAL NUMBERS, NON-CONSTANT, BOUNDED AND CONTINUOUS ACTIVATION FUNCTION f , AND EVERY POSITIVE EPSILON, THERE EXISTS A 1-HIDDEN LAYER NEURAL NETWORK USING f THAT OBTAINES AT MOST EPSILON ERROR IN FUNCTIONAL SPACE

Horvik+91

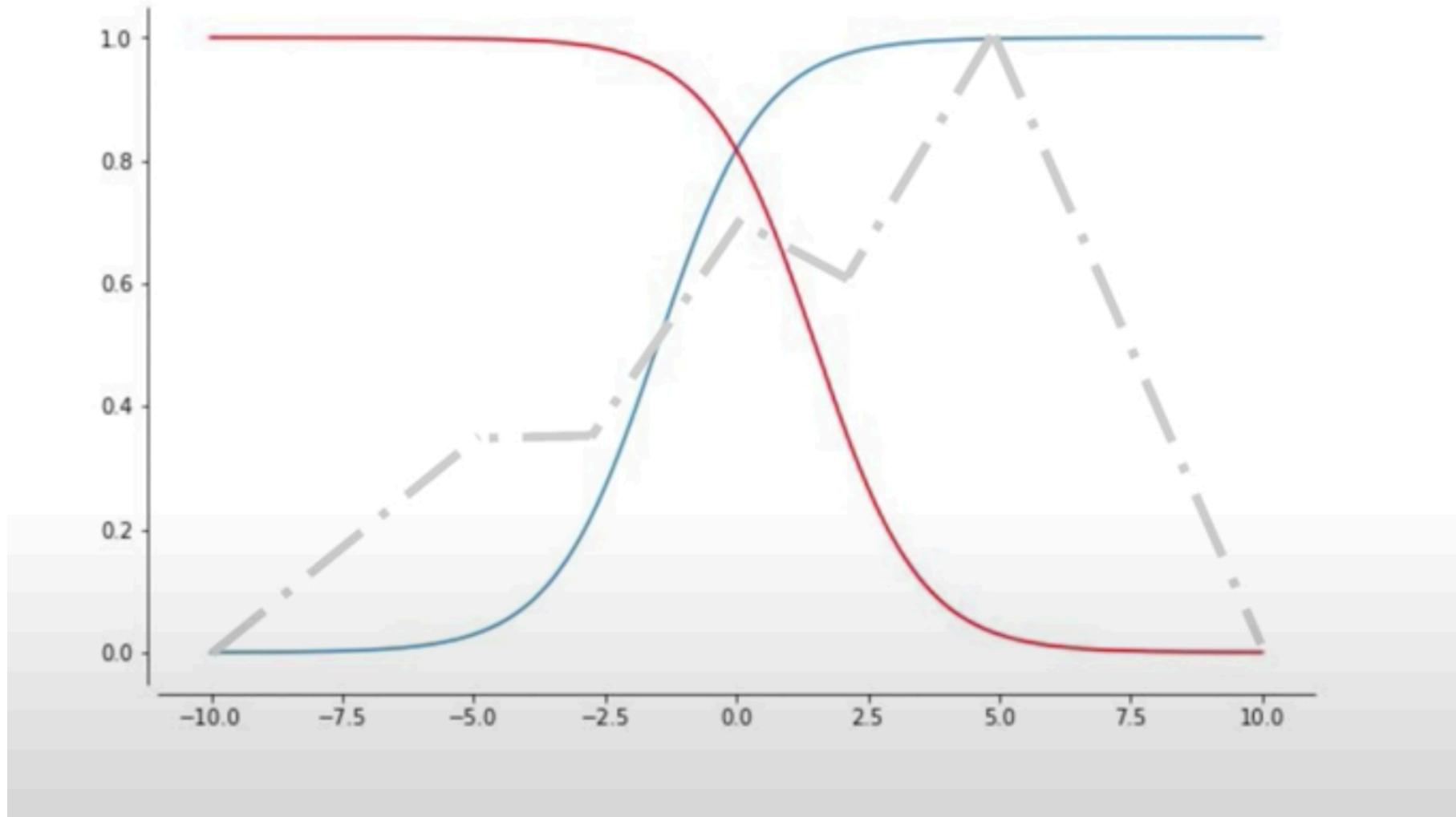
“BIG ENOUGH NETWORK CAN APPROXIMATE, BUT NOT REPRESENT ANY SMOOTH FUNCTION. THE MATH DEMONSTRATION IMPLIES SHOWING THAT NETWORS ARE DENSE IN THE SPACE OF TARGET FUNCTIONS”



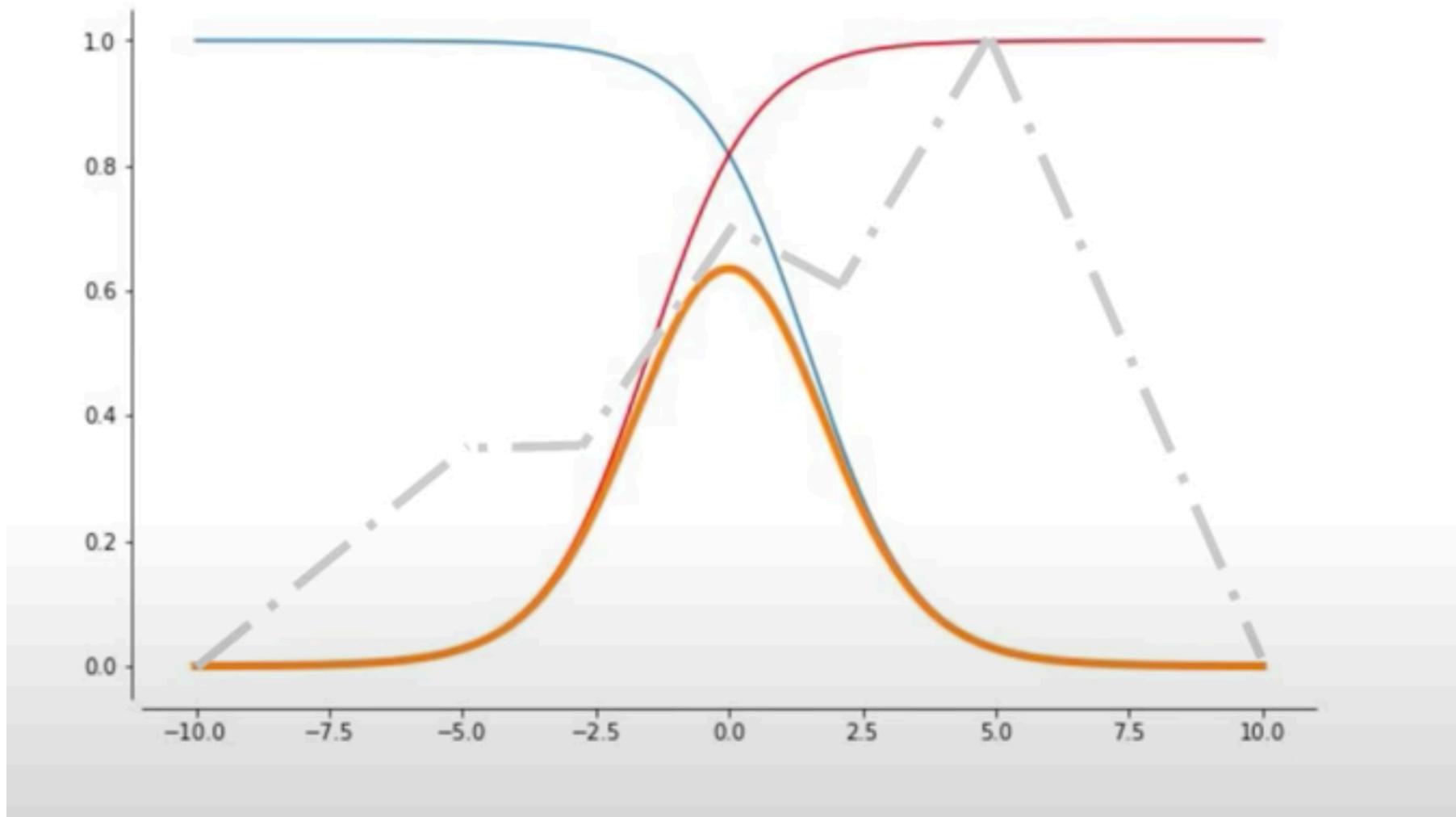
(deepmind
@Czarnecki)



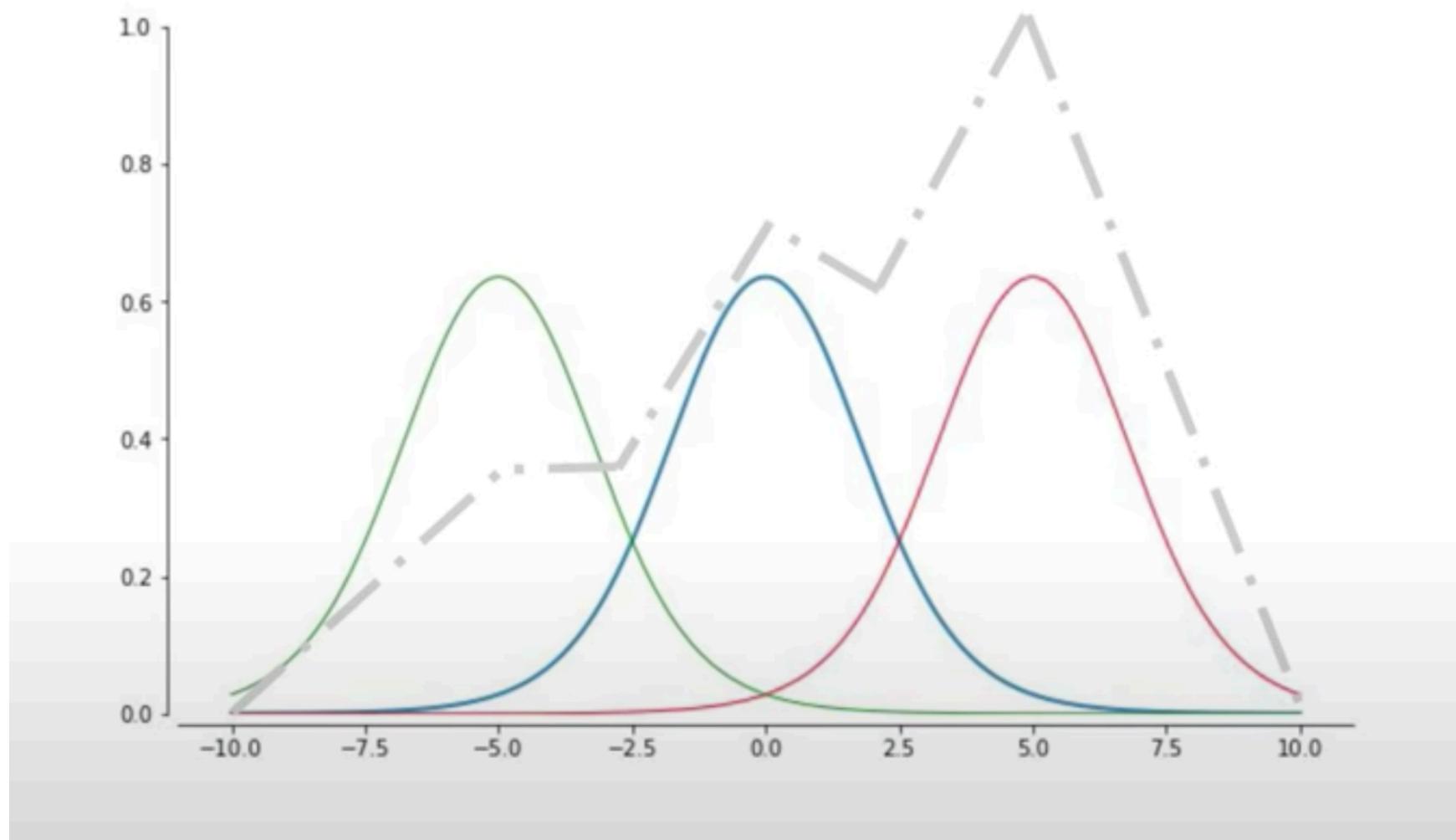
(deepmind
@Czarnecki)



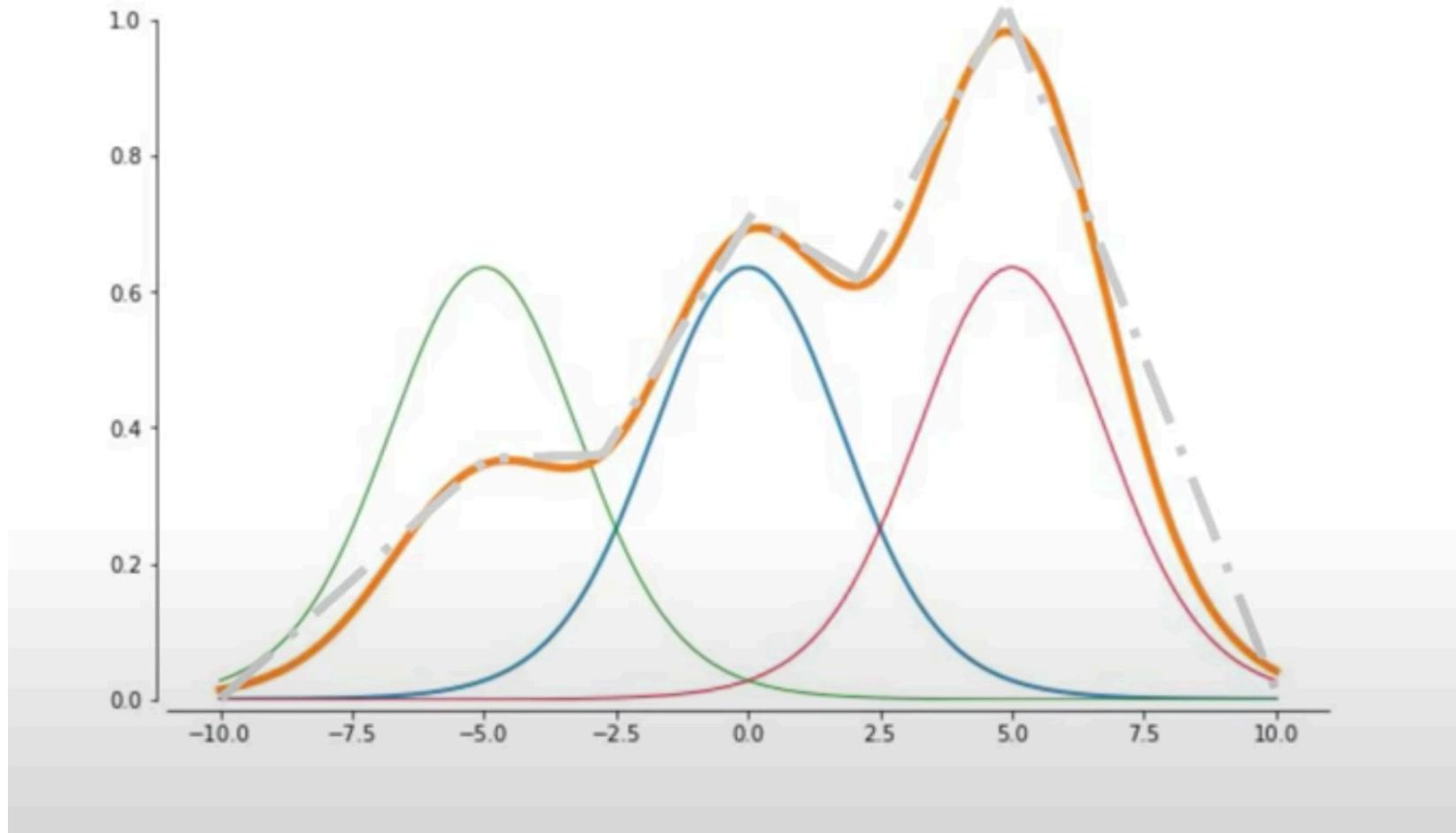
(deepmind
@Czarnecki)



(deepmind
@Czarnecki)

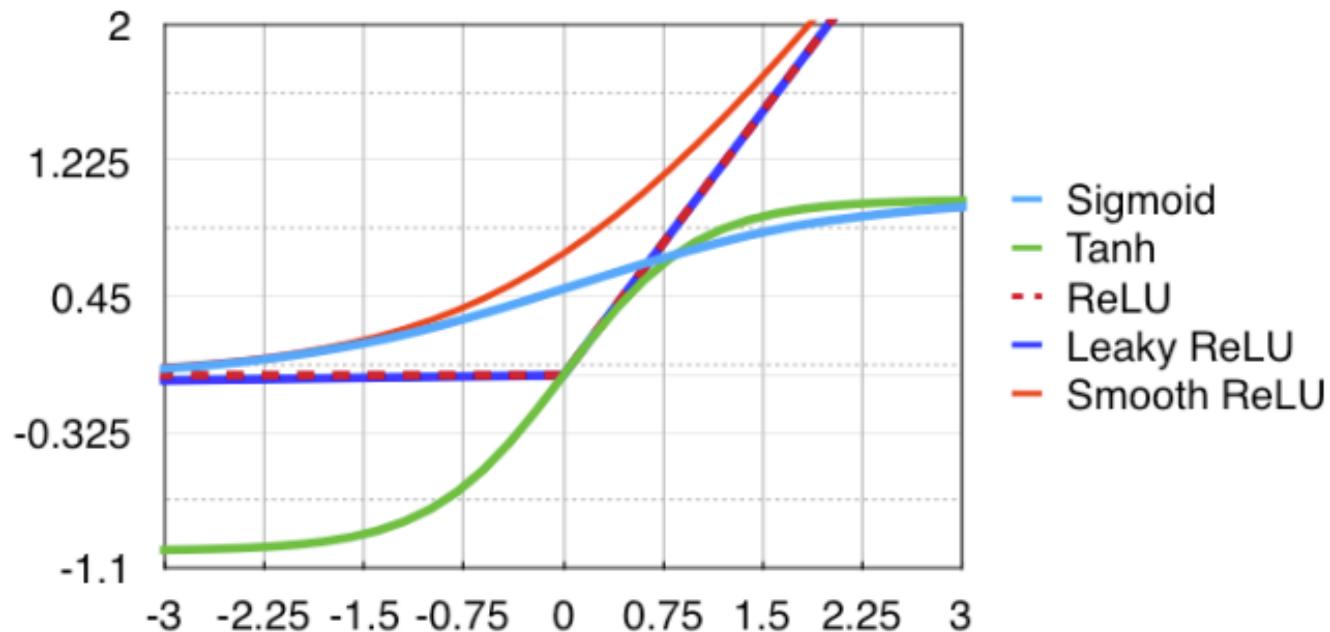


(deepmind
@Czarnecki)



(deepmind
@Czarnecki)

ACTIVATION FUNCTIONS



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

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

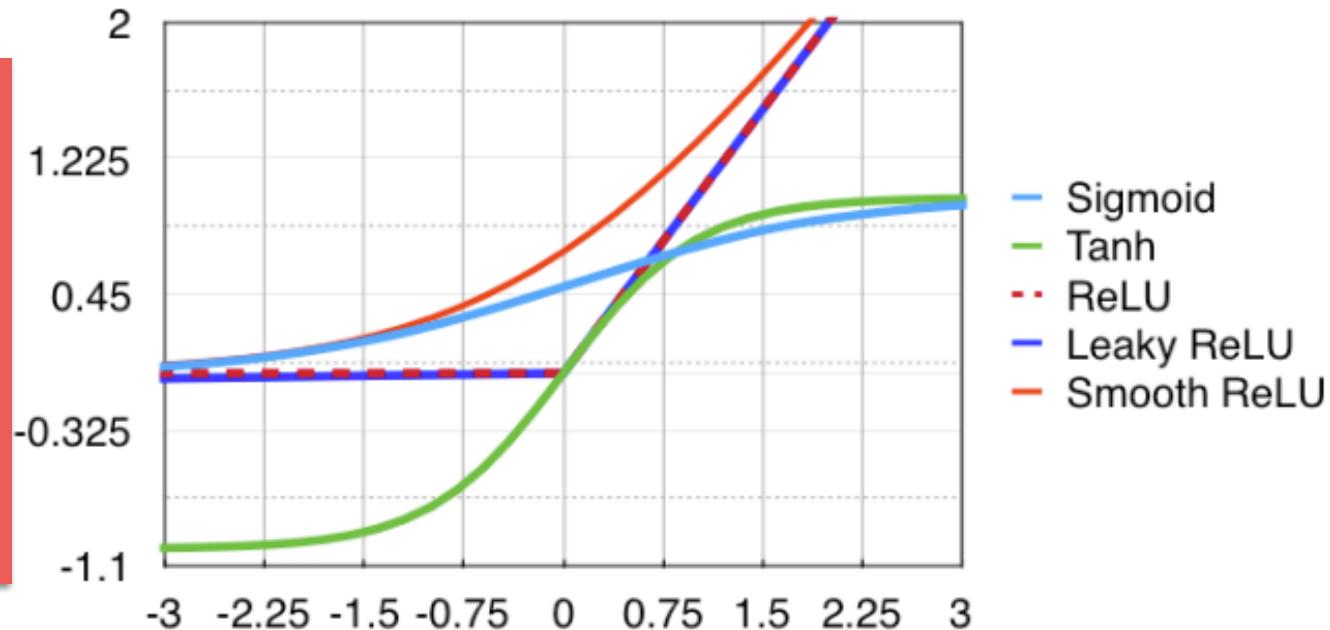
ReLU: $f(x) = \max(0, x)$

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

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

ACTIVATION FUNCTIONS

+
**MANY
OTHERS!**



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

SOFTMAX

A generalization of the SIGMOID ACTIVATION

$$\text{softmax}(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{i=1}^n e^{x_i}}$$

THE OUTPUT IS NORMALIZED BETWEEN 0 AND 1

THE COMPONENTS ADD TO 1

CAN BE INTERPRETED AS A PROBABILITY

$$p(Y = c | X = \mathbf{x}) = \text{softmax}(z(\mathbf{x}))_c$$

SO LET'S GO DEEPER AND DEEPER!

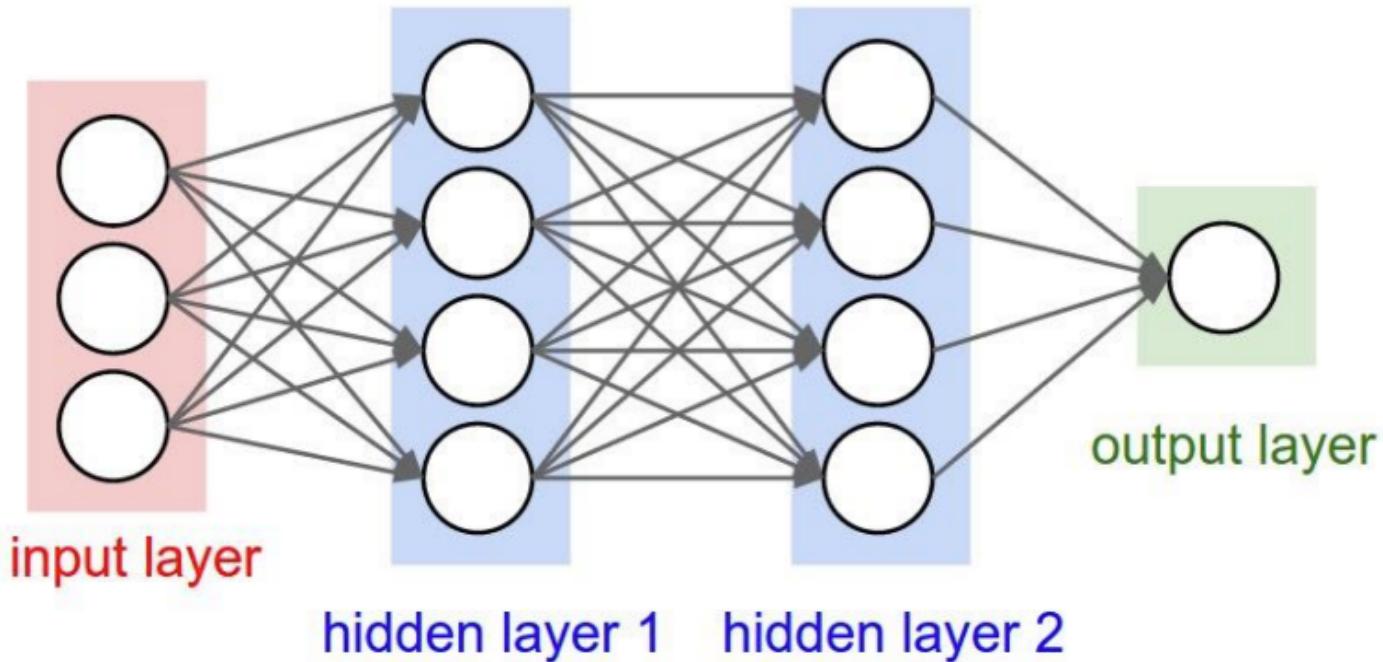
YES BUT...

NOT SO STRAIGHTFORWARD, DEEPER MEANS MORE
WEIGHTS, MORE DIFFICULT OPTIMIZATION, RISK OF
OVERFITTING...

**OK, SO NOW LET'S FIND THE
WEIGHTS**

OPTIMIZATION

[OR HOW TO FIND THE WEIGHTS?]



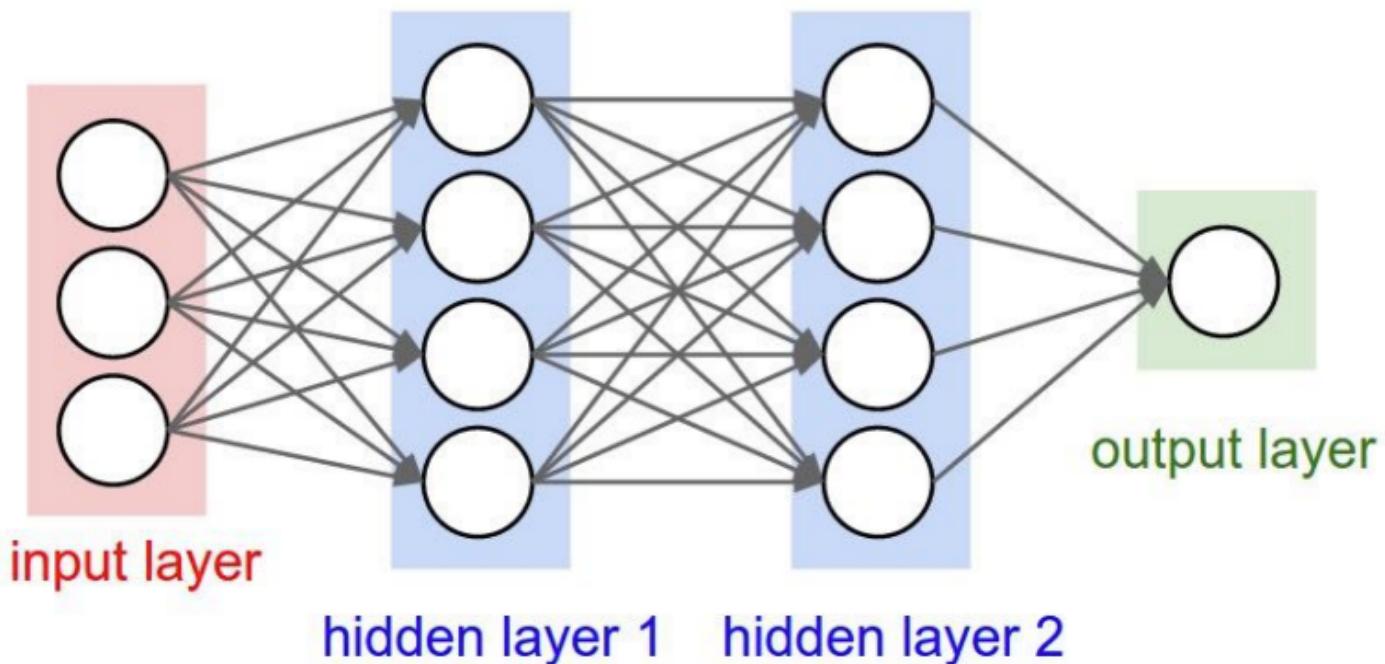
$$p = g_3(W_3g_2(W_2g_1(W_1\vec{x}_0)))$$



NETWORK
FUNCTION

OPTIMIZATION

[OR HOW TO FIND THE WEIGHTS?]



$$p = g_3(W_3g_2(W_2g_1(W_1\vec{x}_0)))$$

$$\frac{1}{N} \sum_{i=1}^N (y_i - p_i)^2$$

←
LOSS
FUNCTION

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

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

FOR EXAMPLE....

Gradient Descent

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



[gradient]

Newton

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

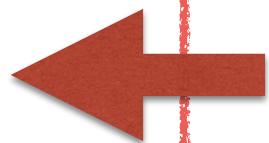


[hessian]

NEWTON CONVERGES FASTER...

BUT NEEDS THE HESSIAN

MOST USED BY FAR....



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

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...

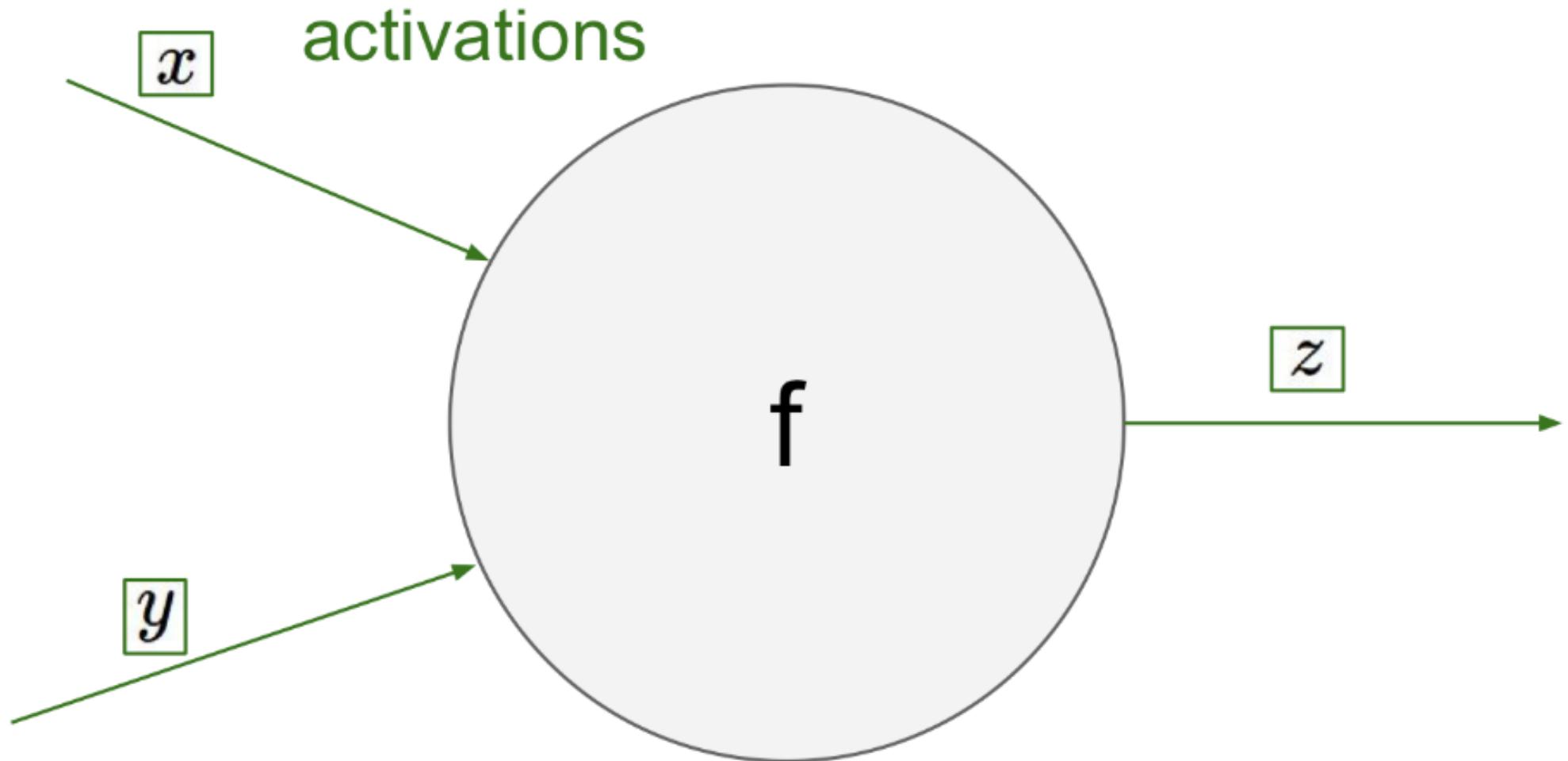
BUT NEEDS THE HESSIAN

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

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

BACKPROPAGATION

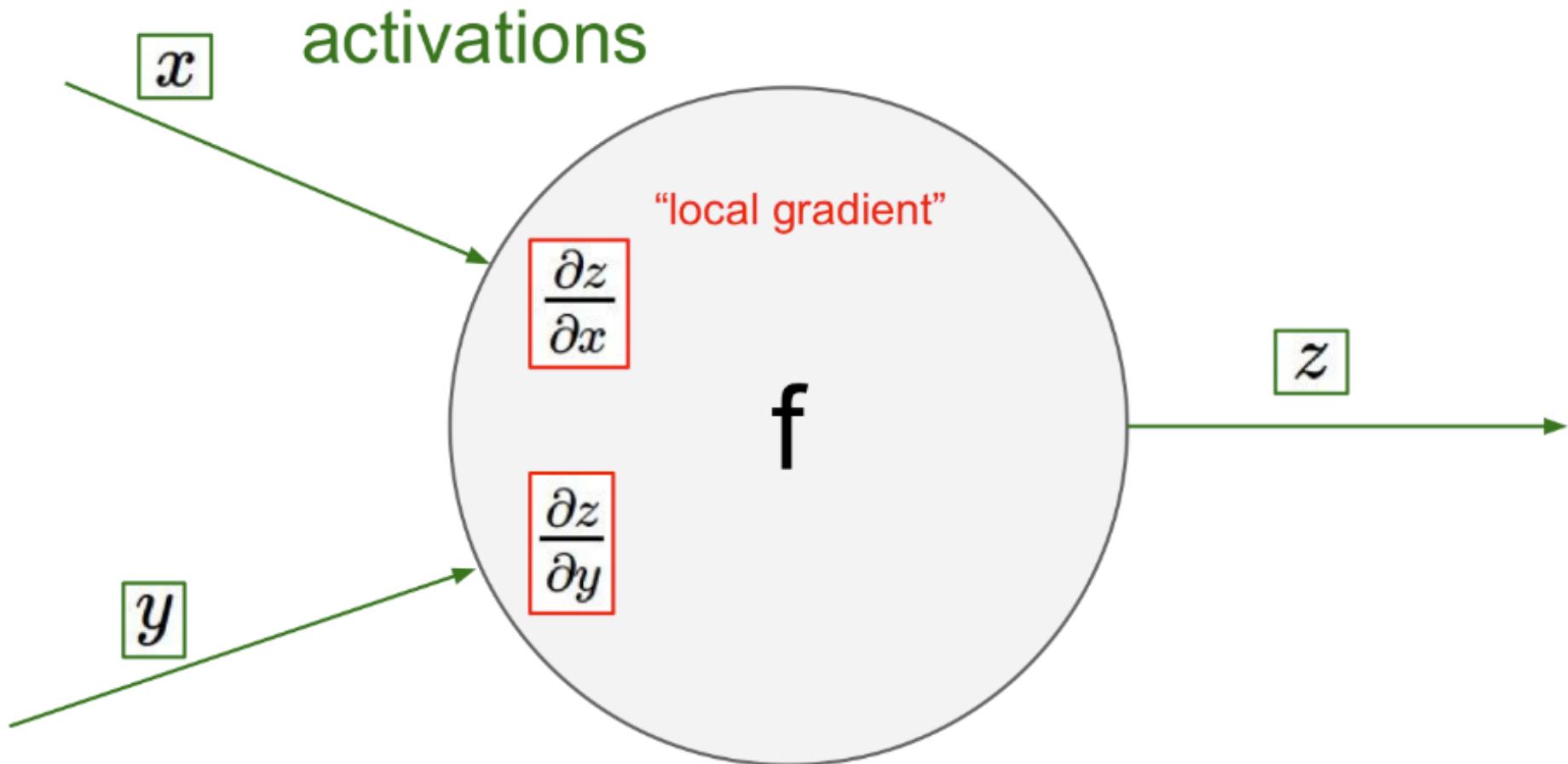
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

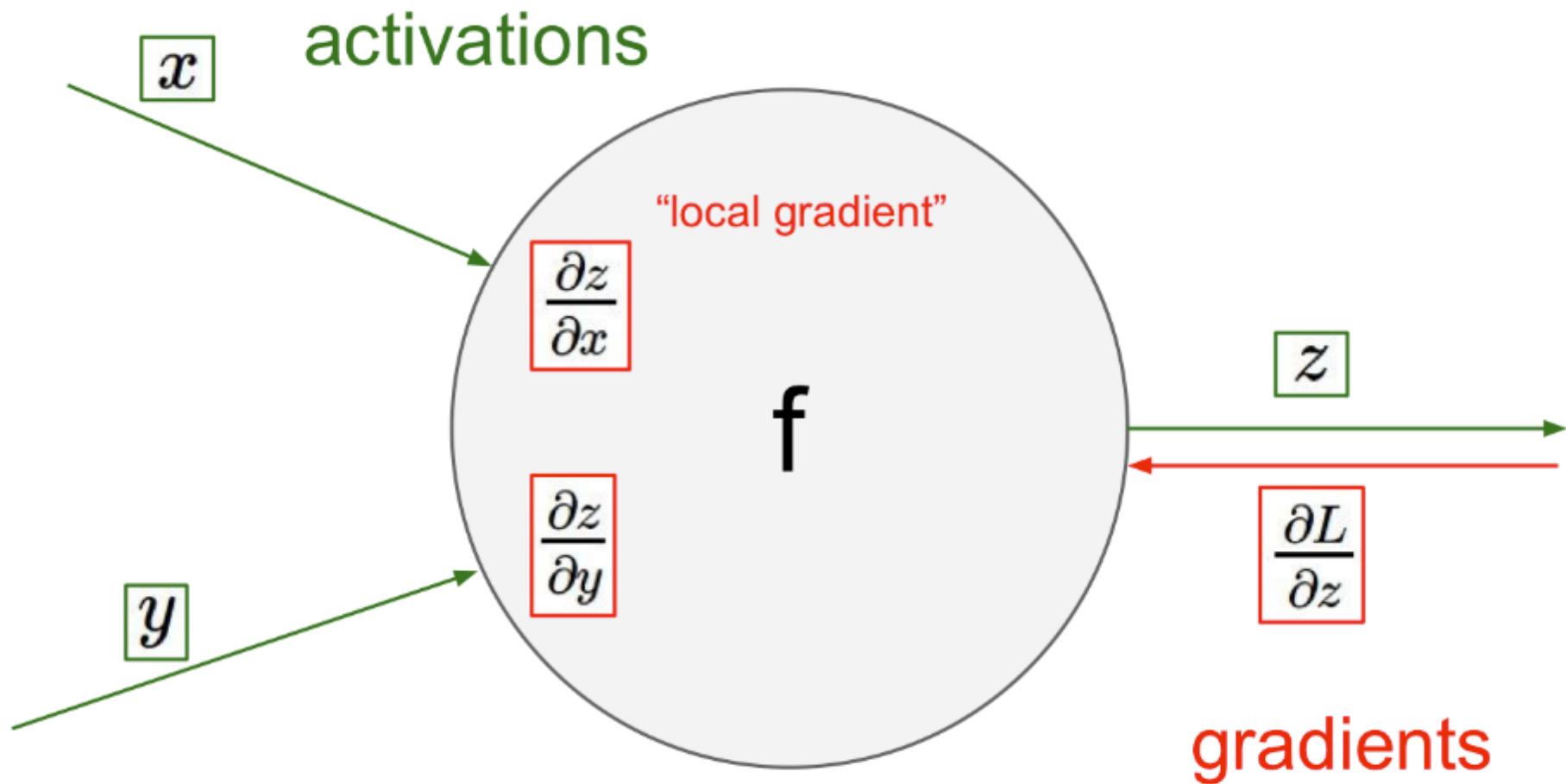
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

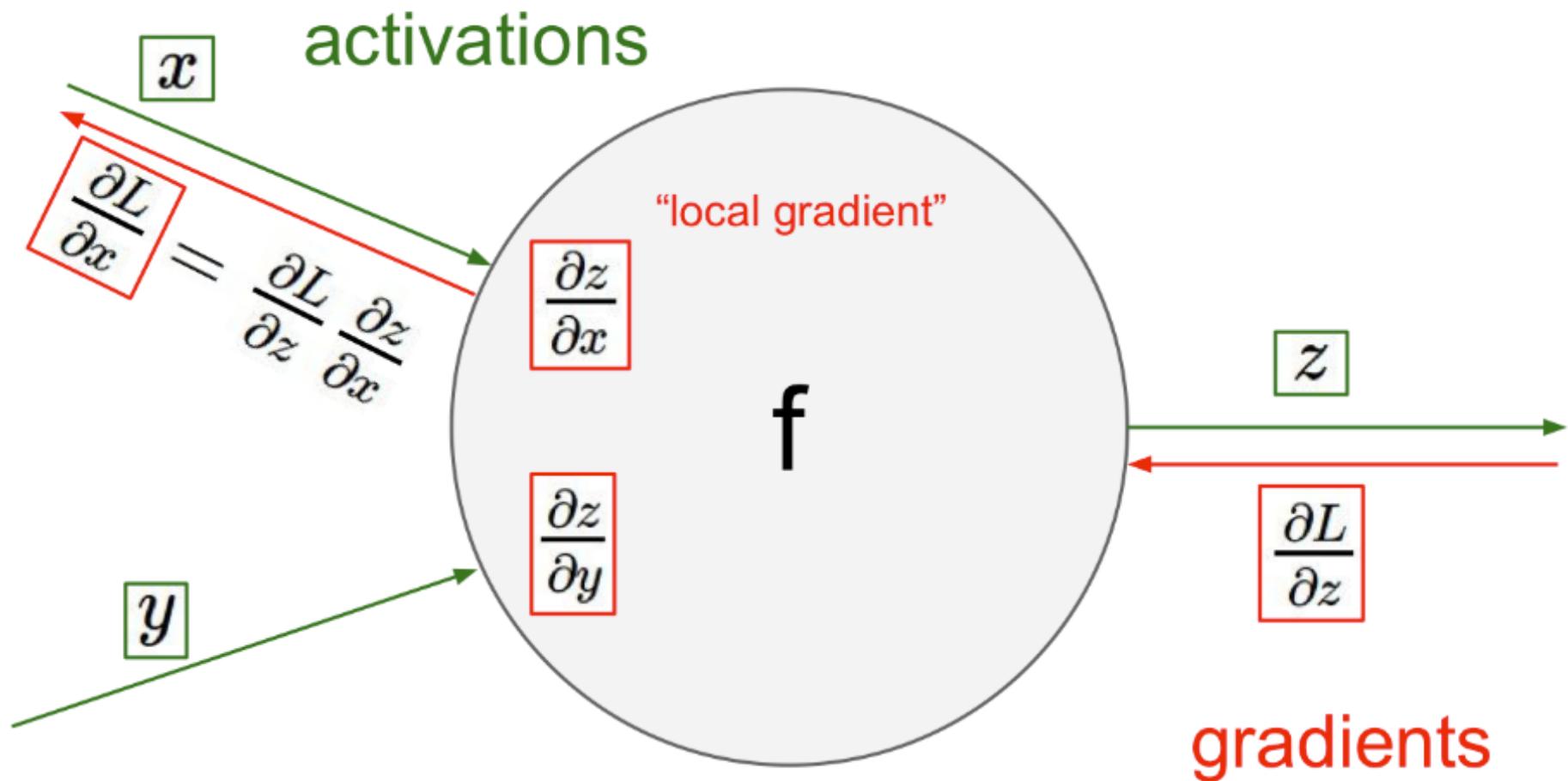
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

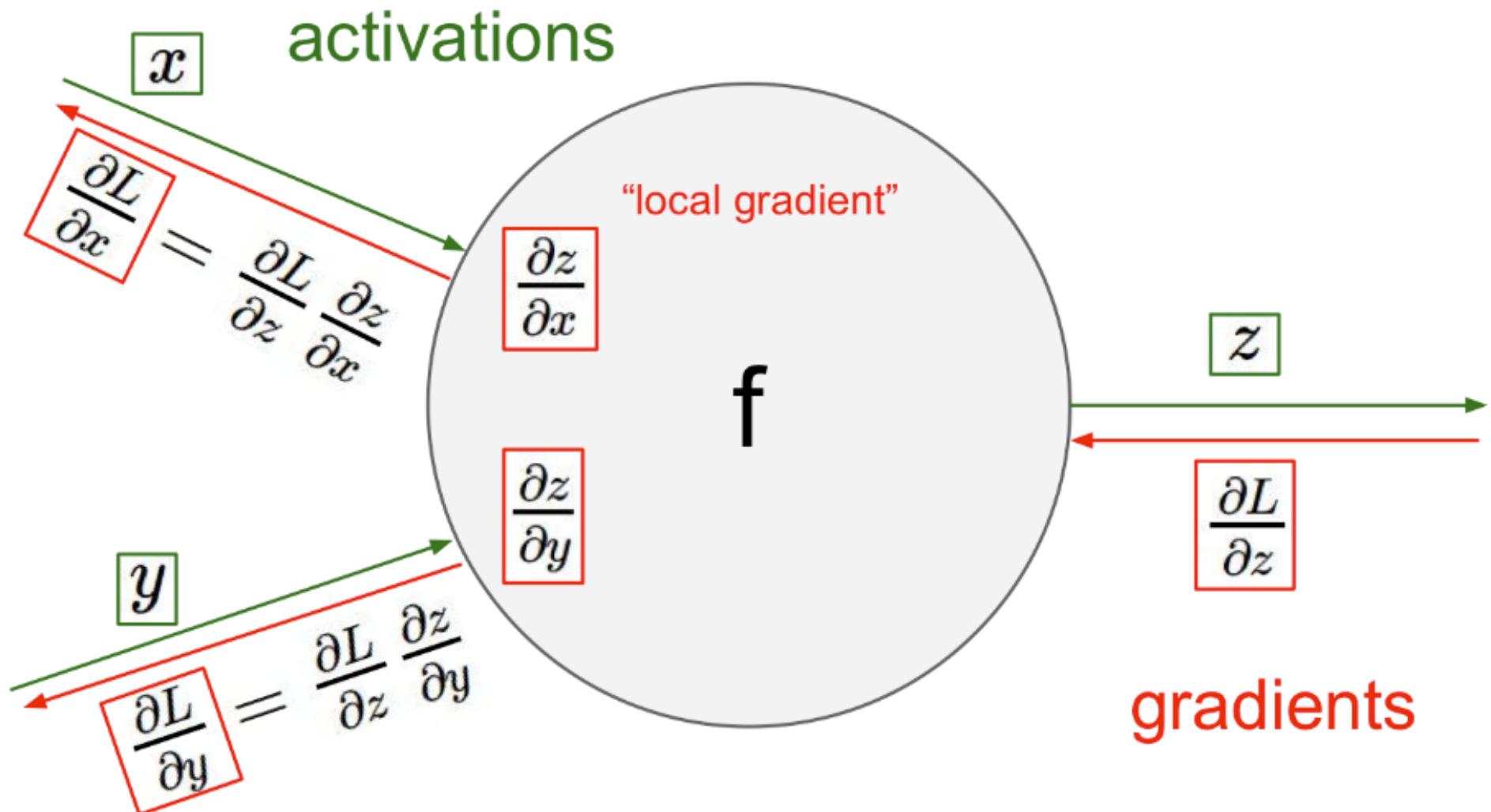
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

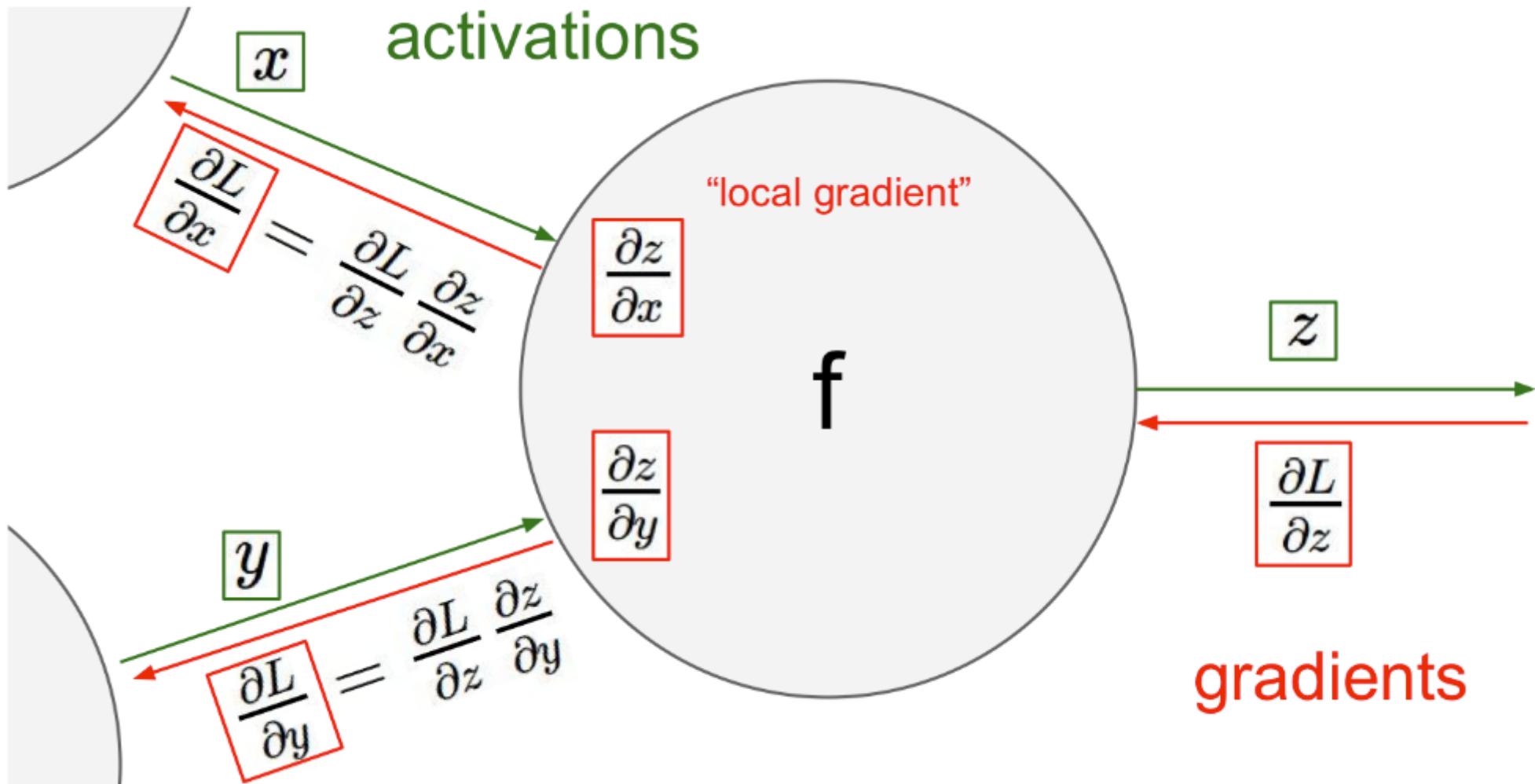
[AT THE NEURON LEVEL]



Credit: A. Karpathy

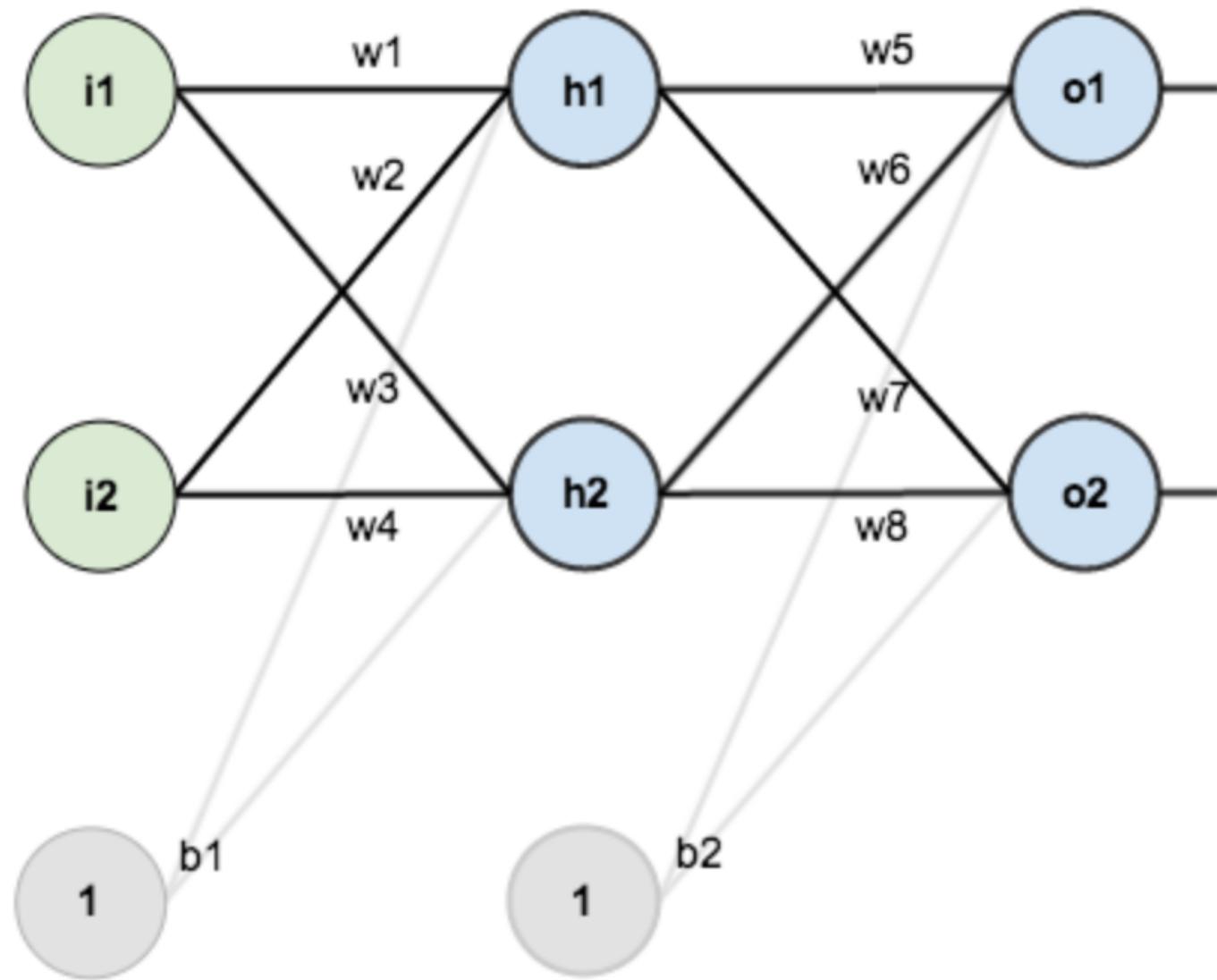
BACKPROPAGATION

[AT THE NEURON LEVEL]

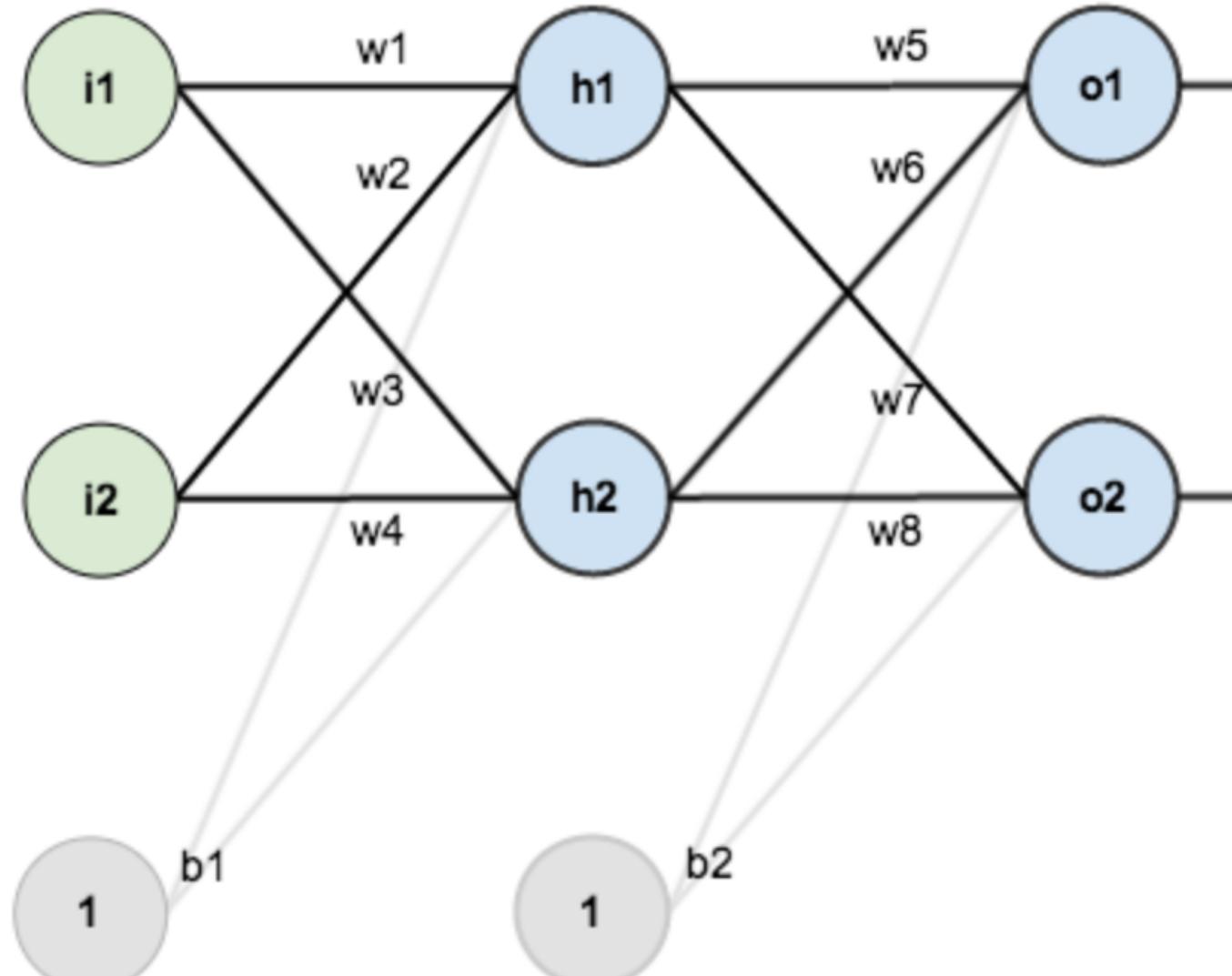


Credit: A. Karpathy

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

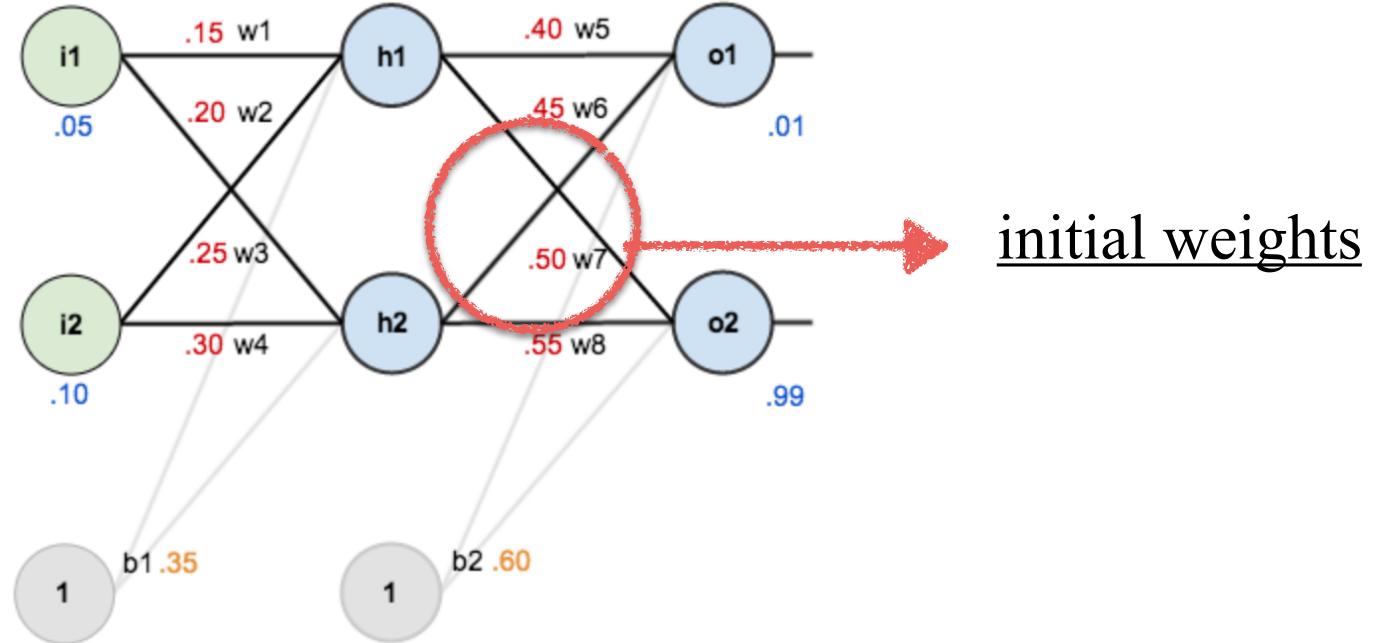


EXAMPLE TAKEN FROM HERE



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

EXAMPLE TAKEN FROM HERE

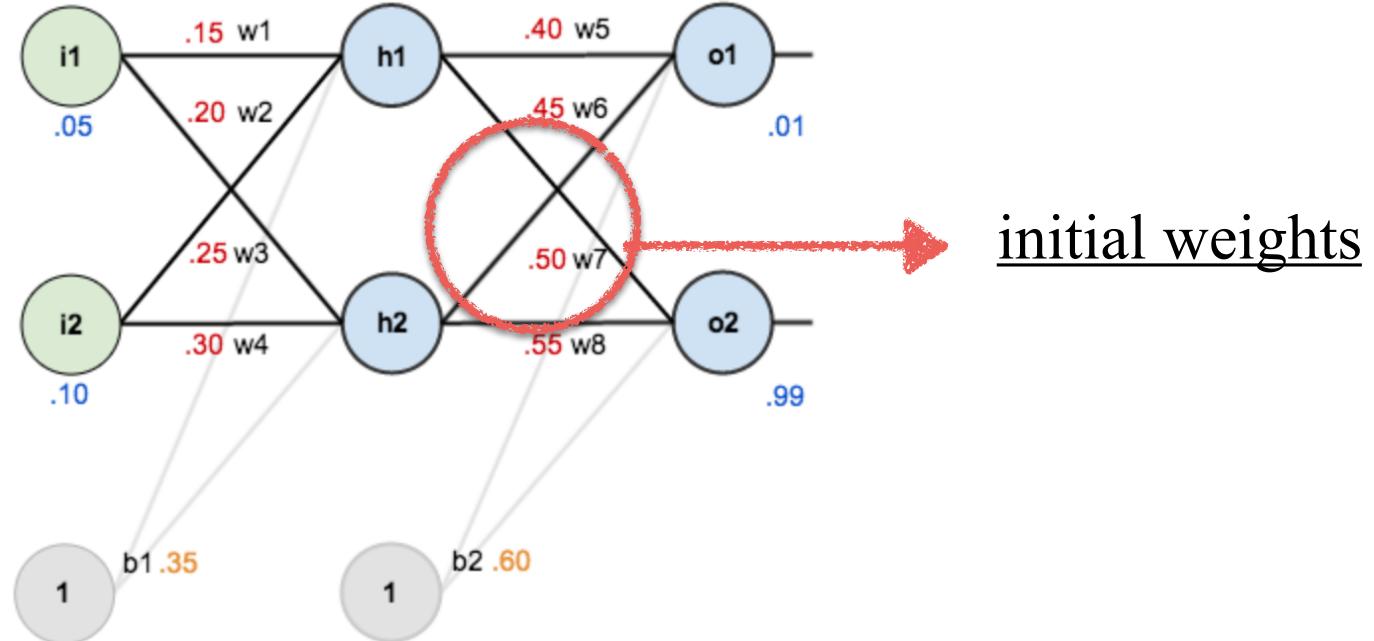


1. THE FORWARD PASS

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

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

[with initial weights]



1. THE FORWARD PASS

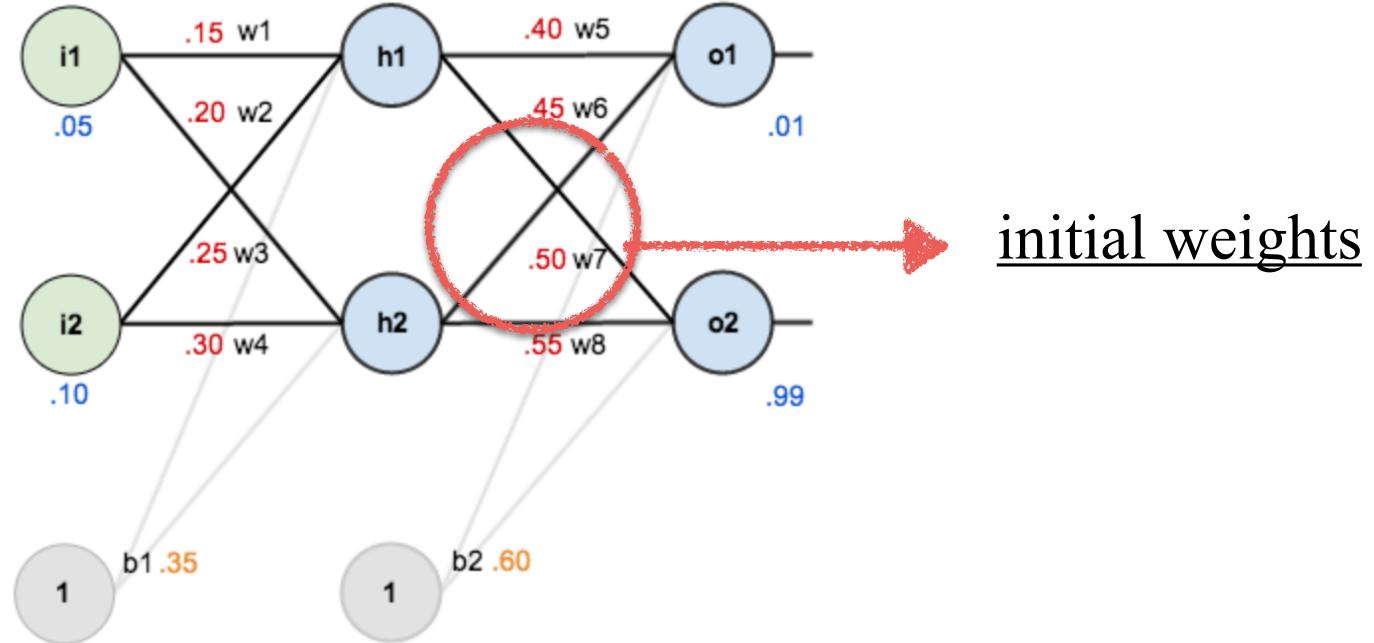
$$in_{h1} = w_1 i_1 + w_2 i_2 + b_1$$

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

[with initial weights]

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

[after the activation function]



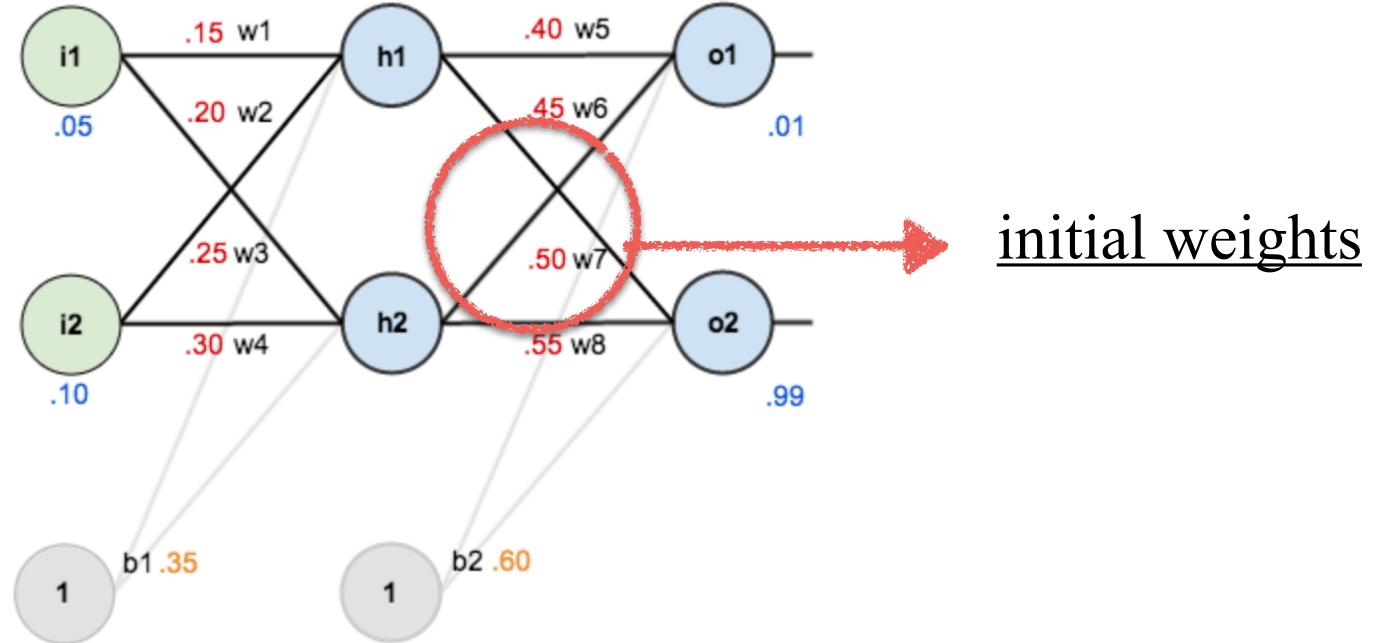
1. THE FORWARD PASS

WE CONTINUE TO o_1

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

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

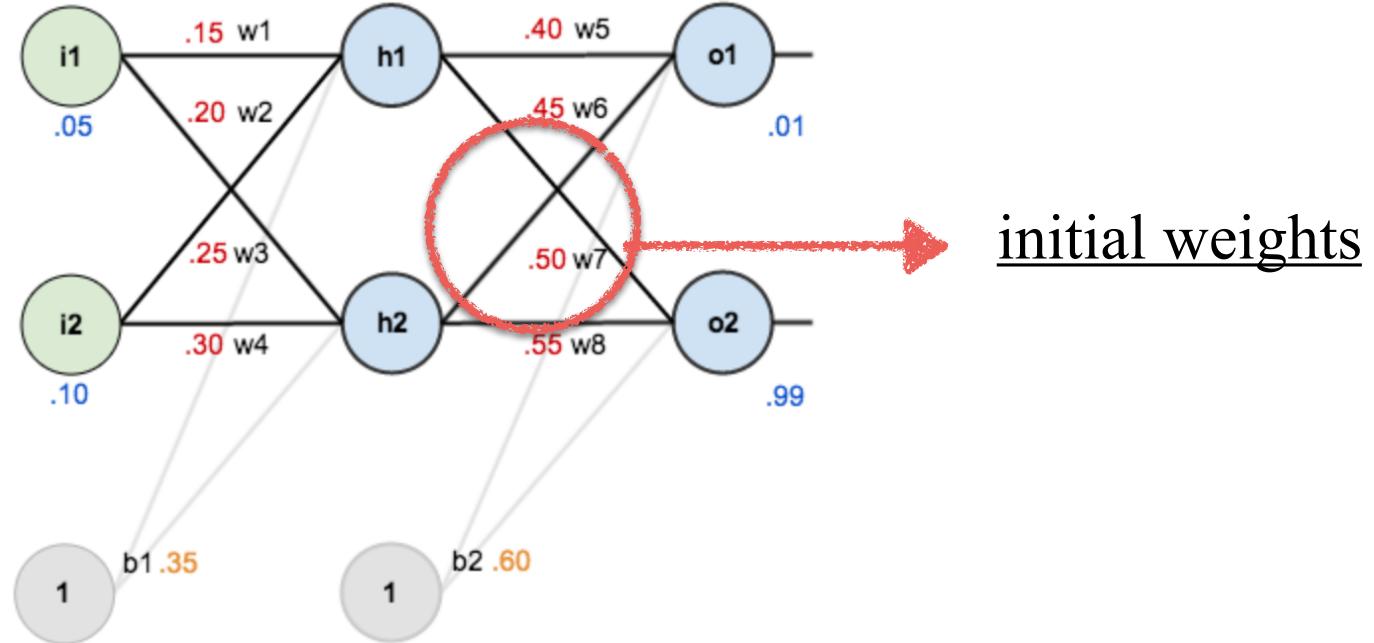
$$out_{o1} = \frac{1}{1 + e^{-1.105}} = 0.751$$



1. THE FORWARD PASS

AND THE SAME FOR o_2

$$out_{o2} = 0.7729$$

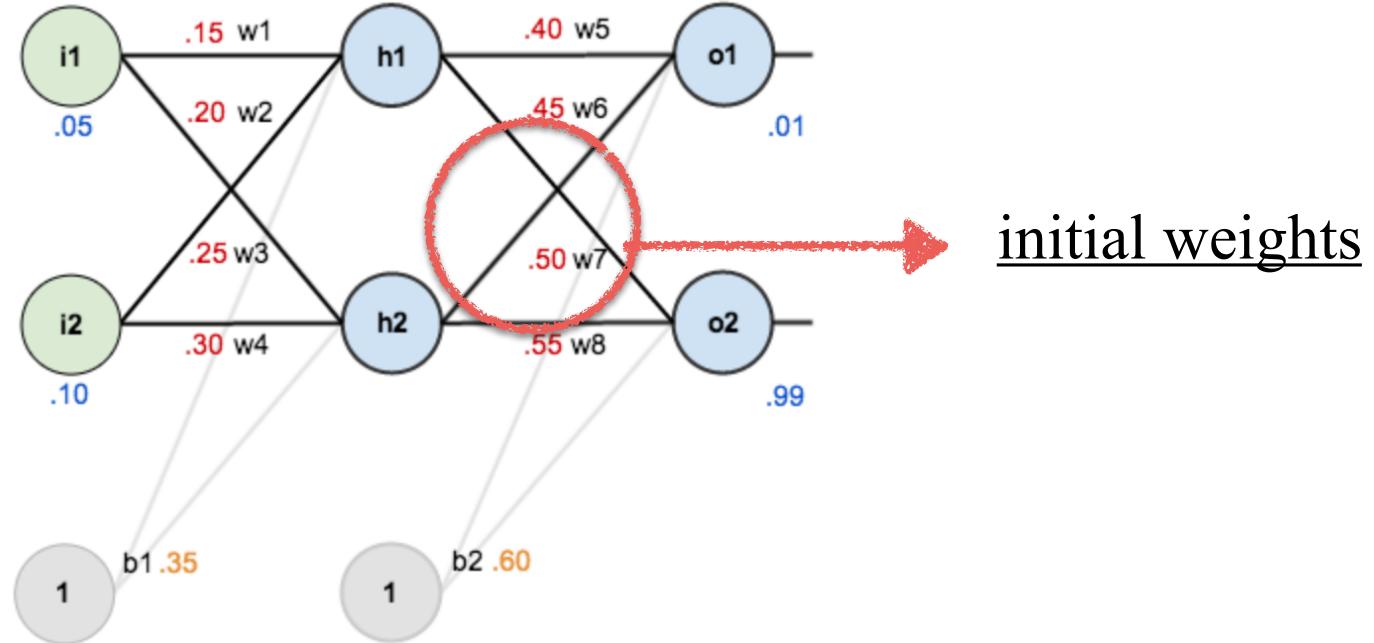


2. THE LOSS FUNCTION

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

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

$$L_{o2} = 0.023$$

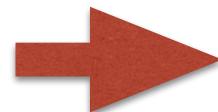


2. THE LOSS FUNCTION

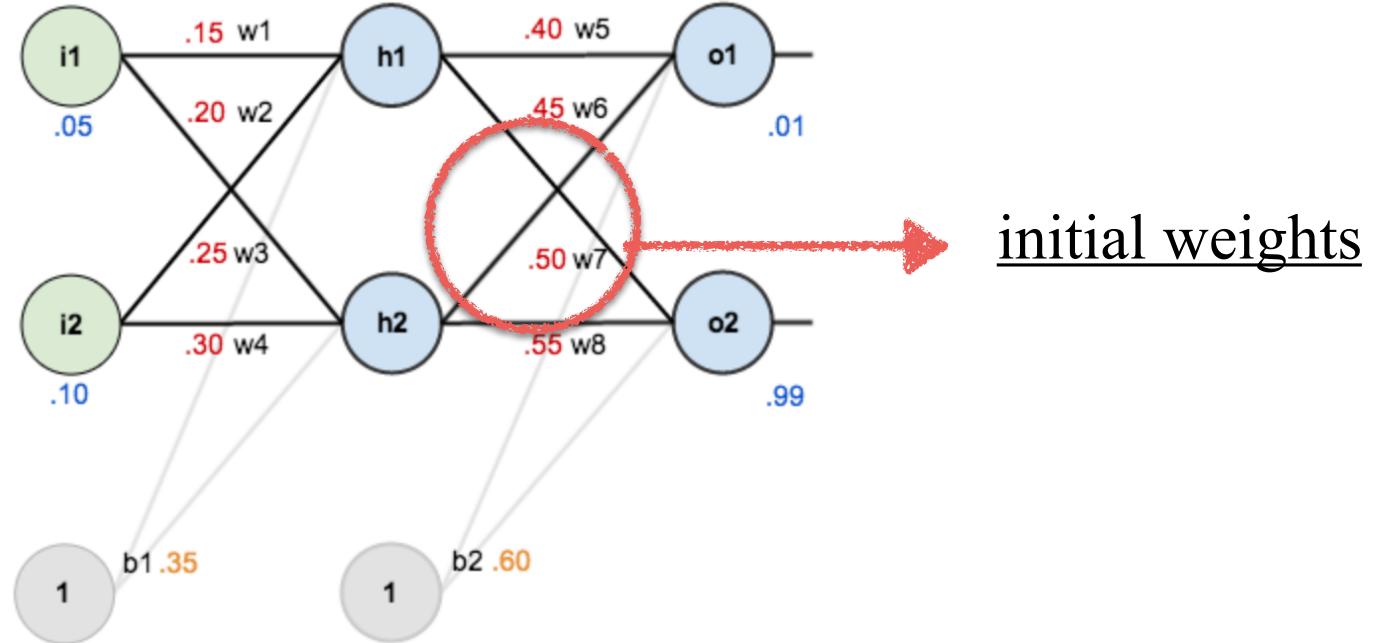
$$L_{total} = \sum 0.5(target - output)^2$$

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

$$L_{o2} = 0.023$$



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



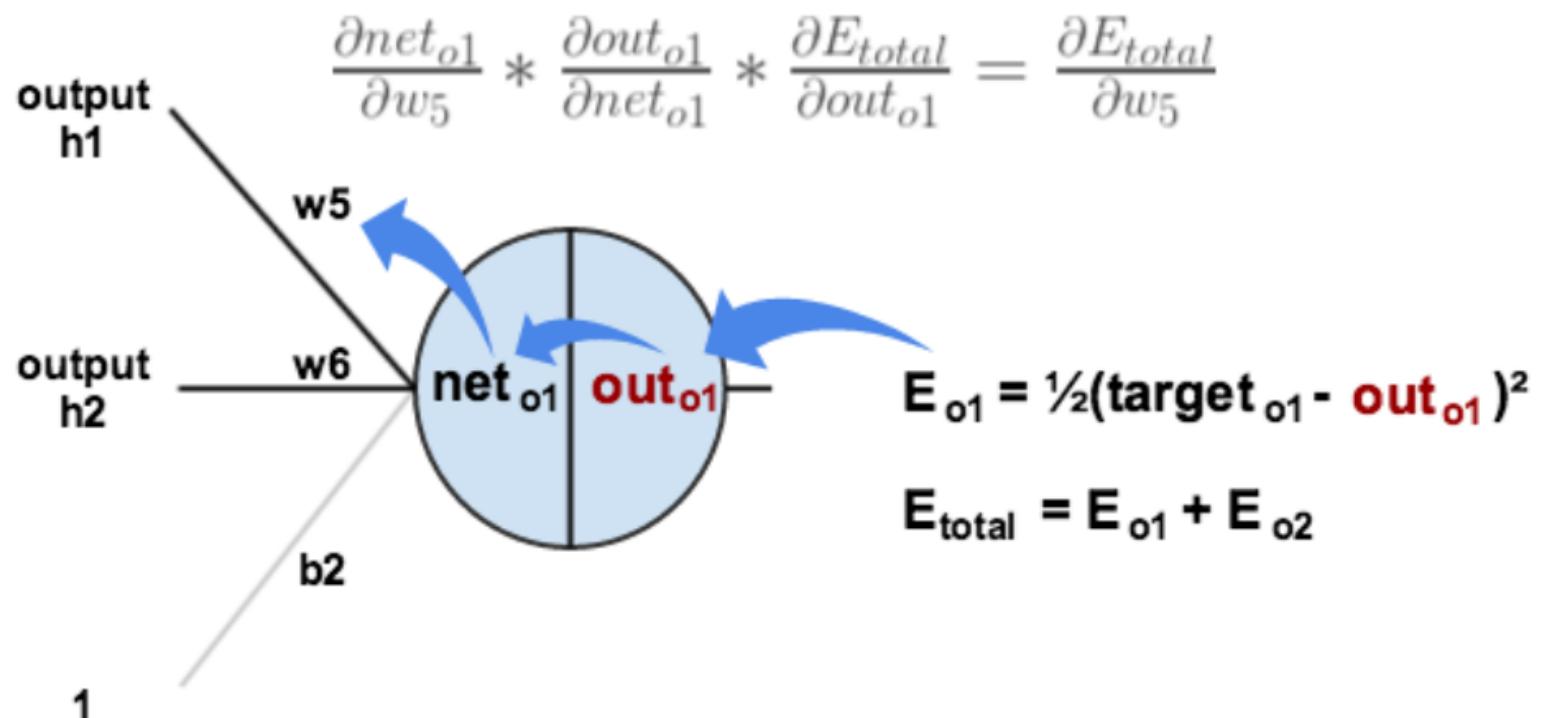
3. THE BACKWARD PASS

FOR *W*₅

WE WANT:

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

[gradient of loss function]



3. THE BACKWARD PASS

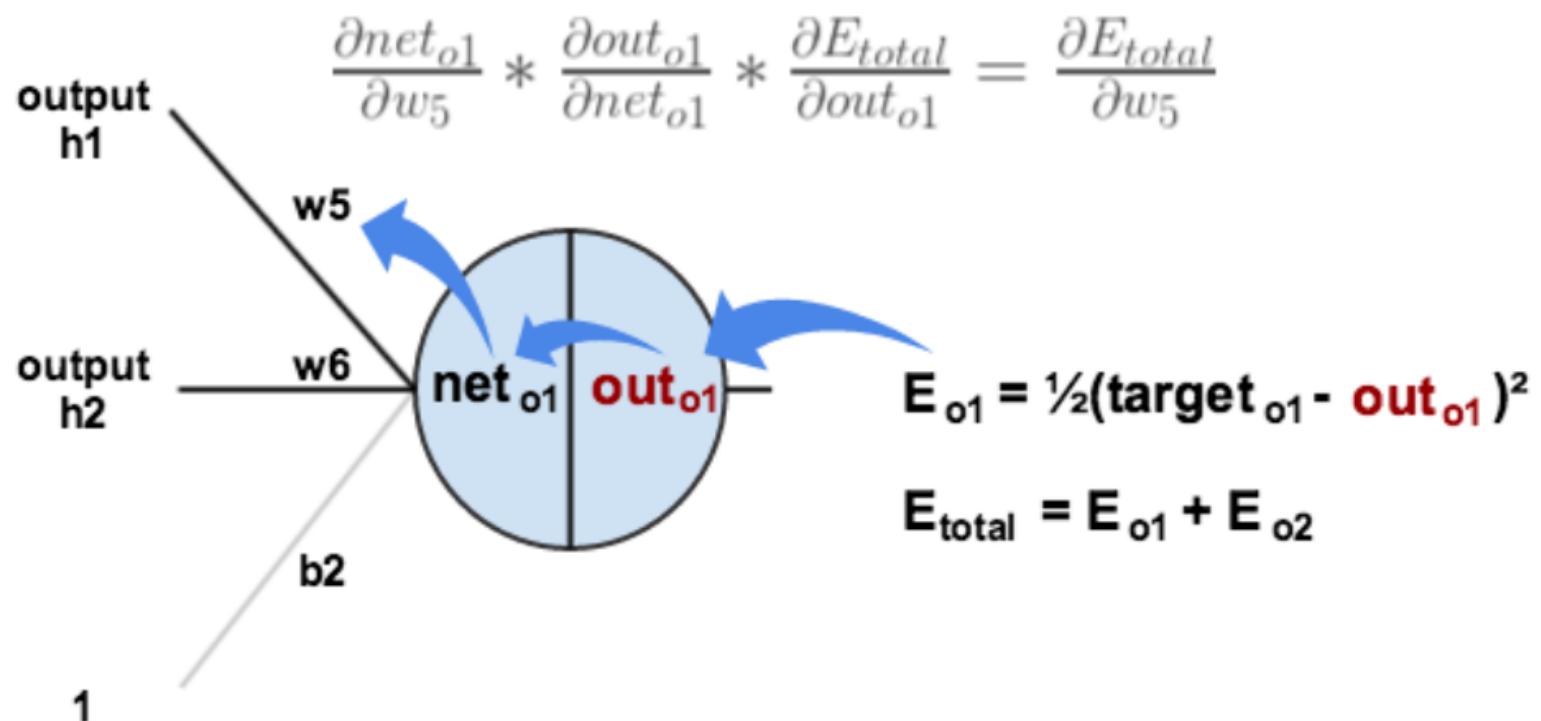
FOR w_5

WE WANT:

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

WE APPLY THE CHAIN RULE:

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

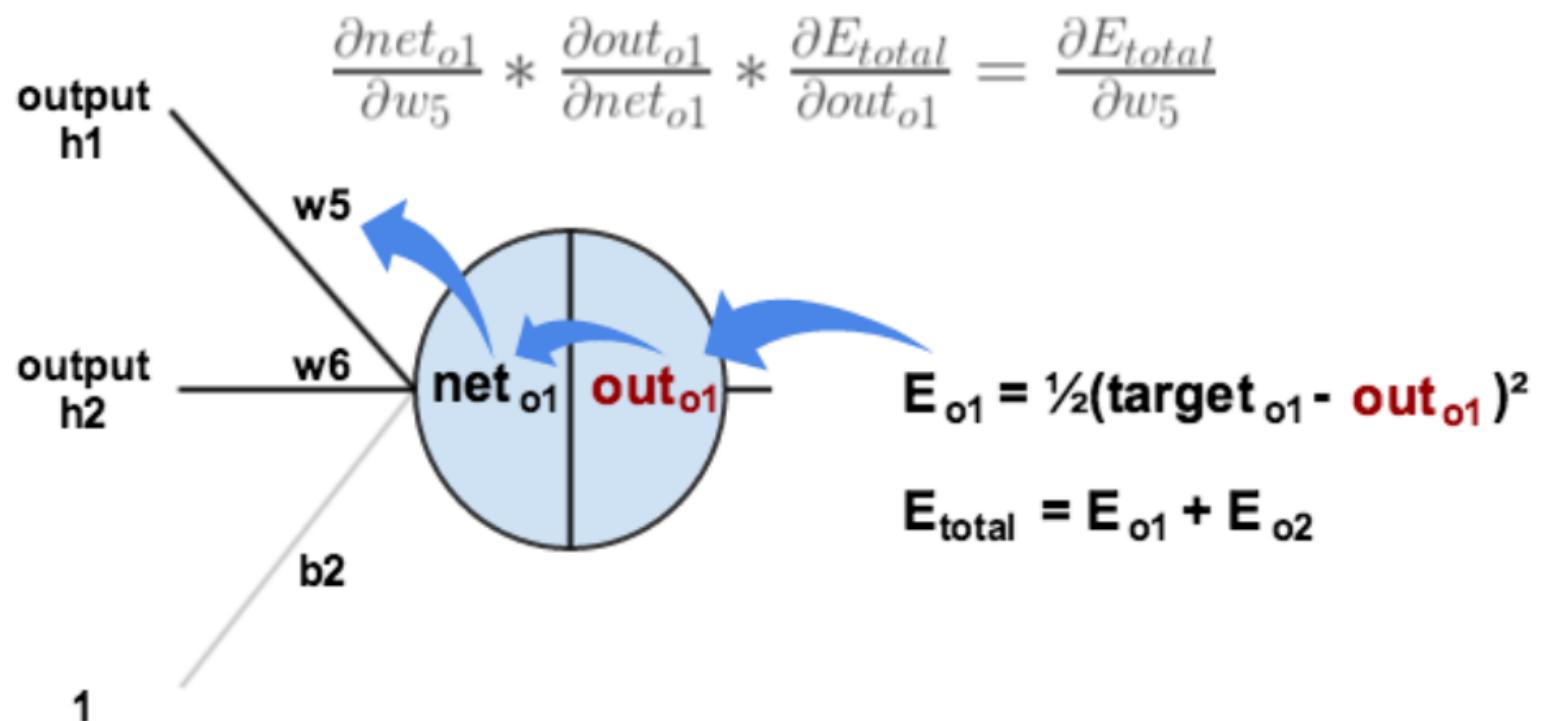


3. THE BACKWARD PASS

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

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

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

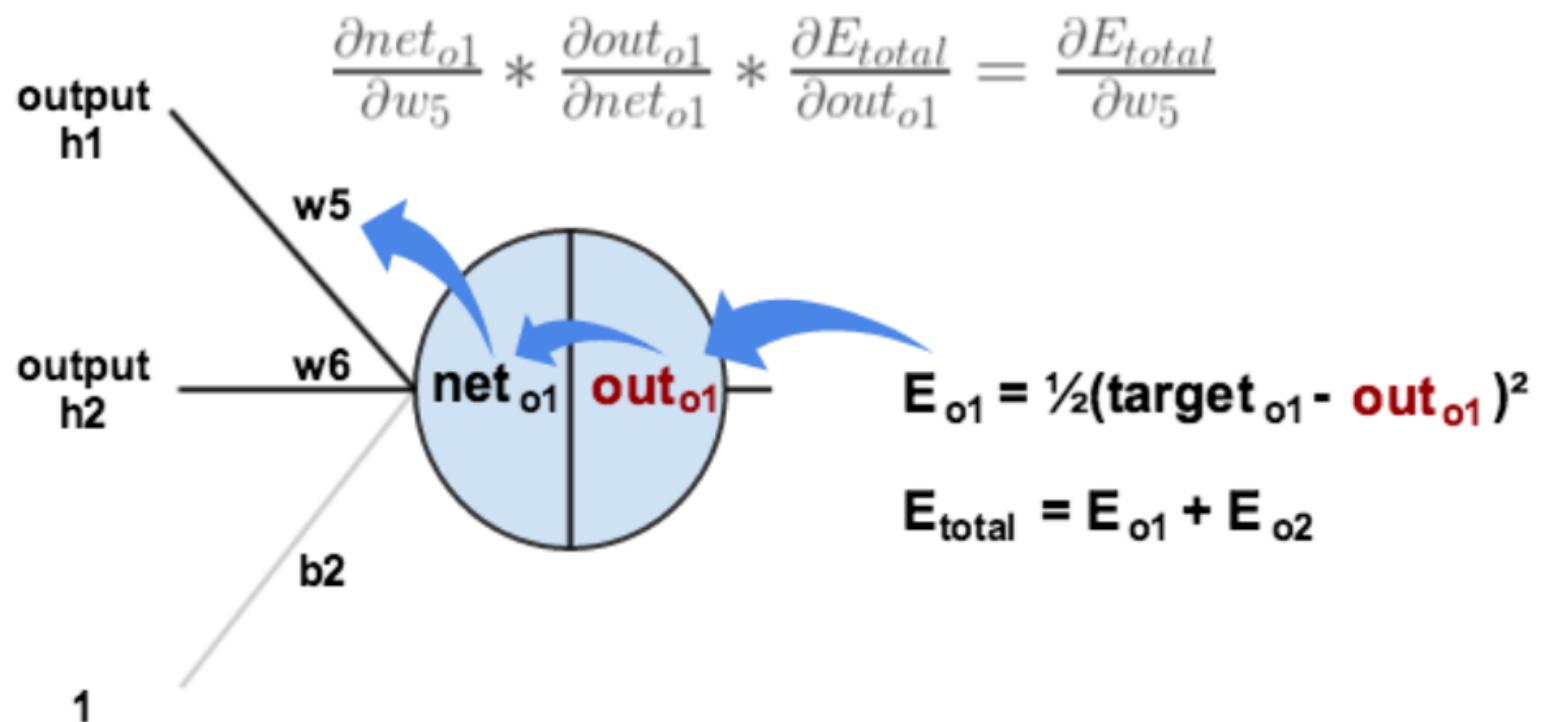


3. THE BACKWARD PASS

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

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

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

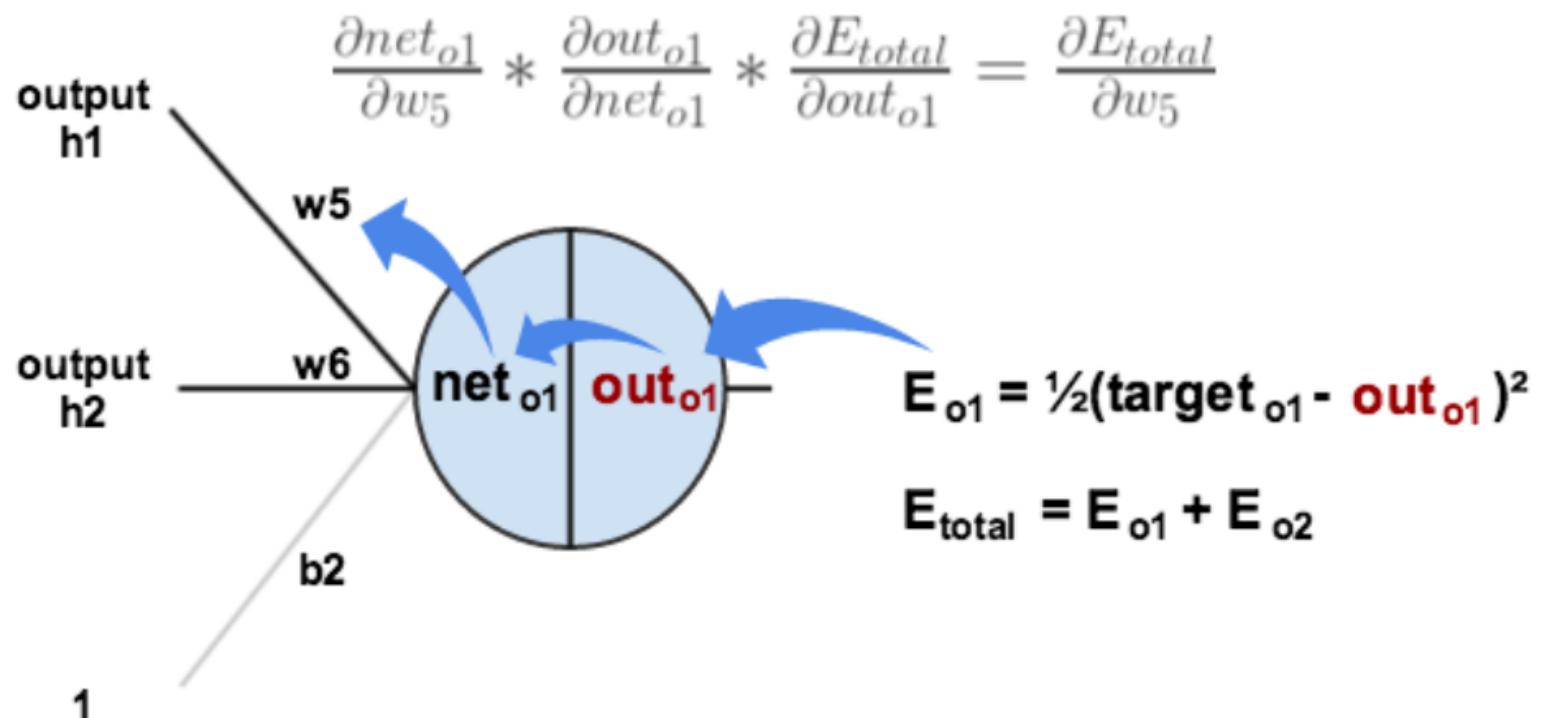


3. THE BACKWARD PASS

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

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

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

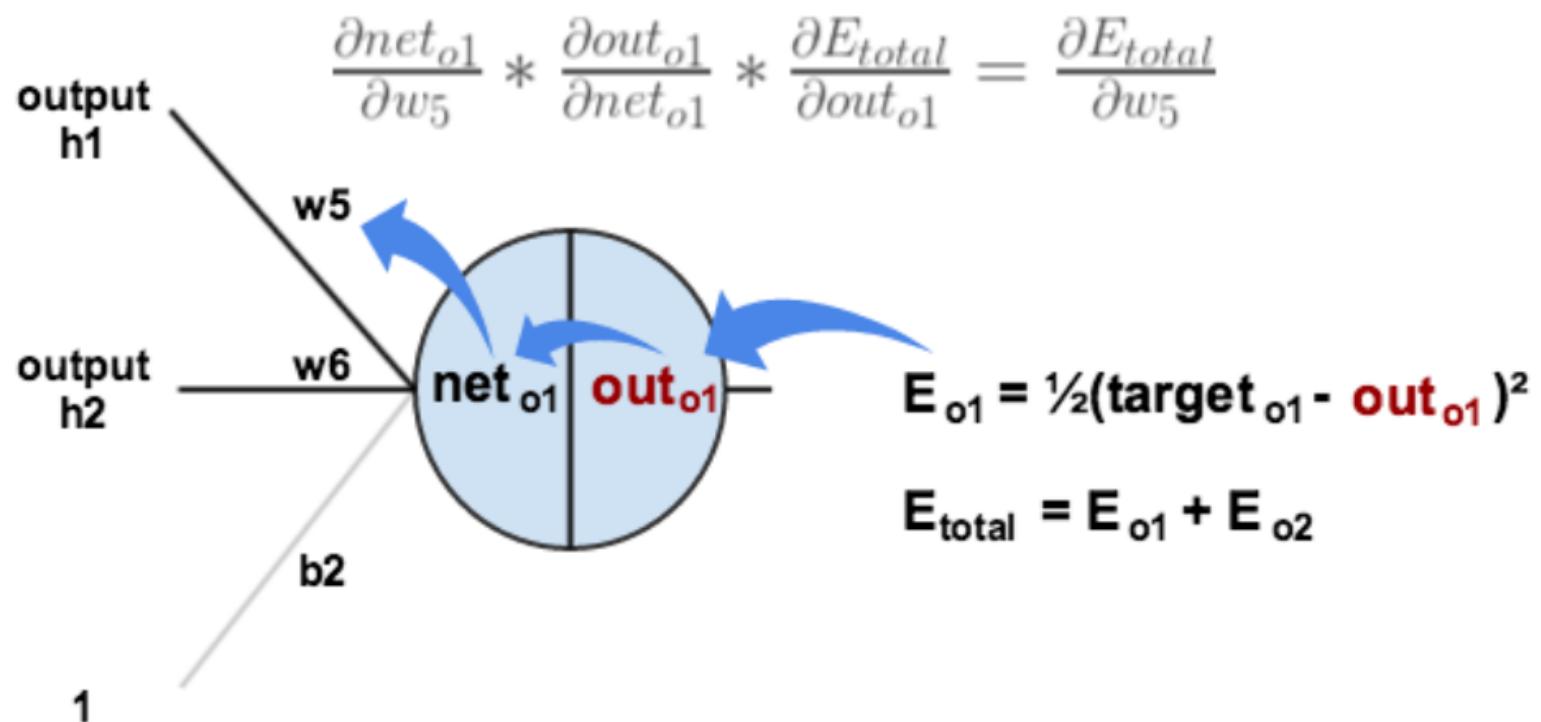


3. THE BACKWARD PASS

ALL TOGETHER:

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

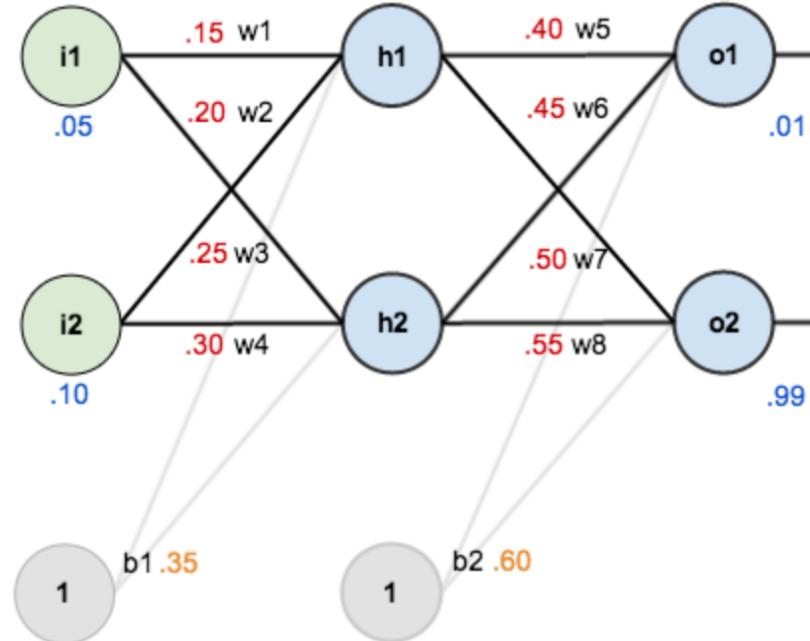
$$\frac{\partial L_{total}}{\partial w_5} = 0.741 \times 0.186 \times 0.593 = 0.082$$



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**

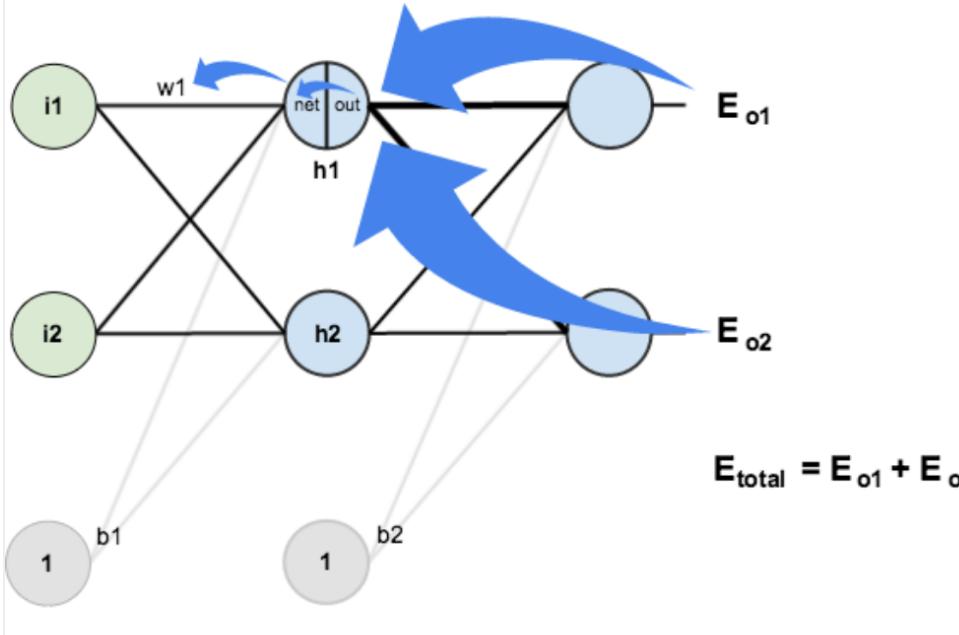
$$w_6^{t+1} = 0.408$$

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

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

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

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

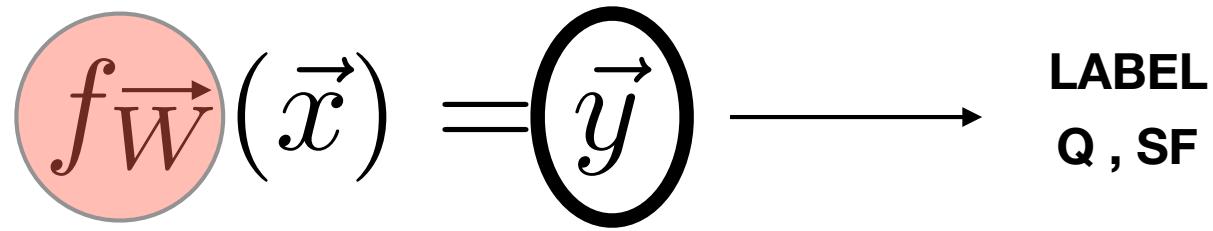


AND BACK-PROPAGATED TO THE HIDDEN LAYERS

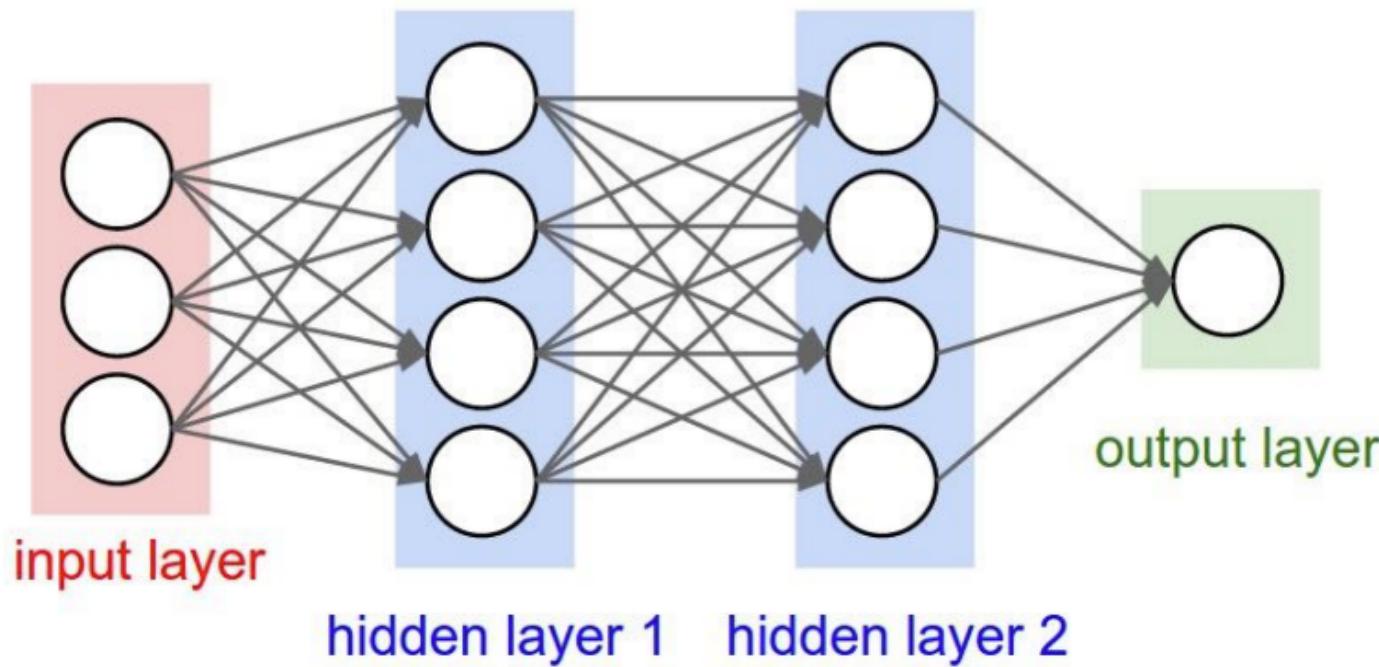
IN ORDER TO KNOW MORE
ABOUT BACKPROPAGATION...

[CLICK HERE](#)

**"CLASSICAL"
MACHINE LEARNING**



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



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

LET'S TALK LOSS FUNCTIONS....

THE CHOICE OF THE LOSS FUNCTION IS CRITICAL AND WILL DETERMINE THE BEHAVIOR OF THE ALGORITHM
(REMEMBER: LOSS IS WHAT IS MINIMIZED BY THE OPTIMIZATION ALGORITHM)

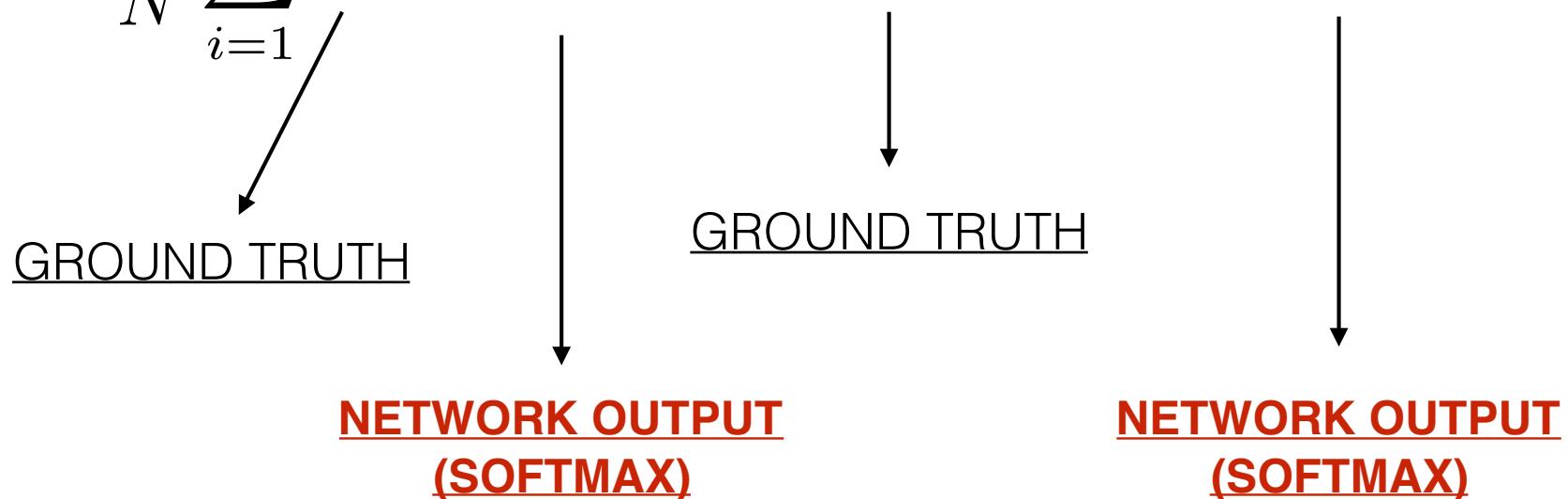
SOME ALGORITHMS SUCH AS RFs HAVE LESS FLEXIBILITY.
ONE ADVANTAGE OF NNs IS THAT THE LOSS FUNCTION CAN BE CHANGED VERY EASILY.

IT IS IMPORTANT TO CHOOSE THE LOSS FUNCTION FOR YOUR PROBLEM

NEURAL NETWORKS AS STATISTICAL MODELS

Let's have a look at the “binary cross-entropy loss”. Where does it come from?

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



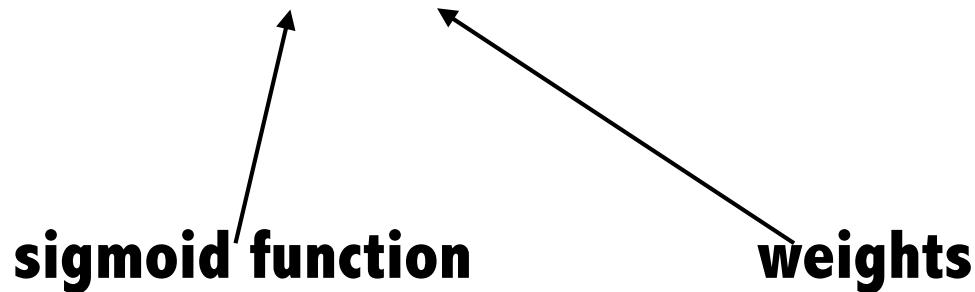
we have a set of independent realizations:

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

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 Bernoulli 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 Bernoulli 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:

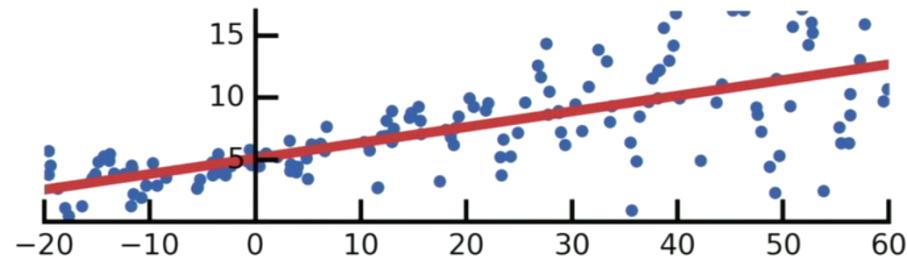
$$l(w; x_i, y_i) = \sum_{i=1}^N y_i \times \log[\text{sigm}(f_w(x_i))] + (1 - y_i) \times \log[1 - \text{sigm}(f_w(x_i))]$$

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

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

WHAT ABOUT REGRESSIONS?



```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(hidden_units, ...),
    tf.keras.layers.Dense(1),
])
model.compile(loss=tf.keras.mean_squared_error)
```

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.

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

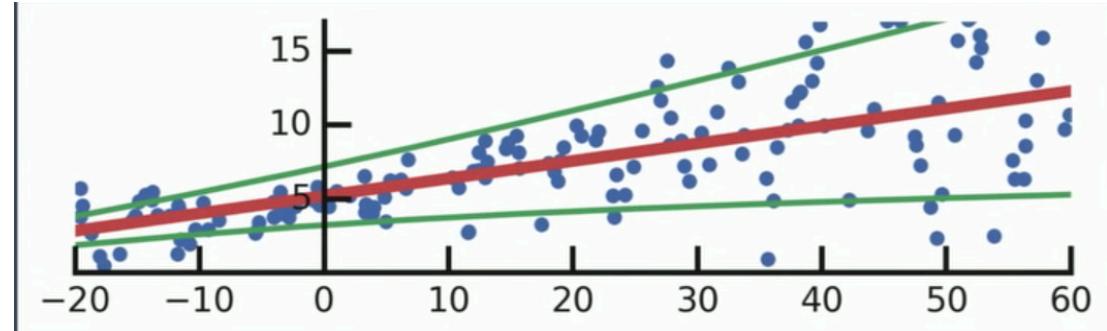
The diagram consists of two black arrows. One arrow originates from the word "weights" at the bottom left and points upwards towards the term w in the equation. The other arrow originates from the words "neural network" at the bottom right and points upwards towards the term $f_w(x_i)$.

**SO, IN A NUTSHELL, WE ARE DOING BAYESIAN VARIATIONAL INFERENCE
ASSUMING THEREFORE A PDF FOR OUR OUTPUT VARIABLE (BERNOULLI
FOR CLASSIFICATION, NORMAL FOR REGRESSION)**

Tensorflow probability allows now to generalize this to any distribution:

Guess what! You can.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(hidden_units, ...),
    tf.keras.layers.Dense(1),
    tfp.layers.DistributionLambda(lambda t:
        tfd.Normal(loc=t[...], 0,
                   scale=1)),
```



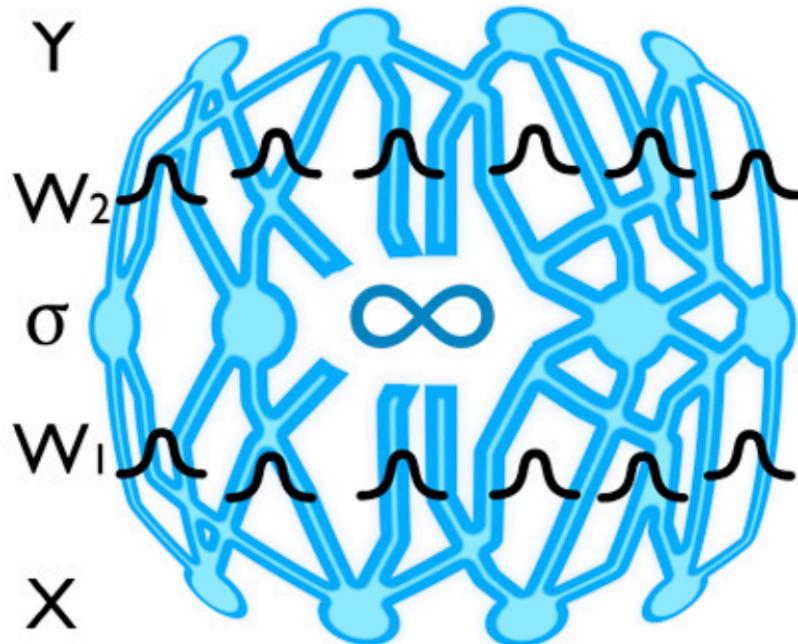
TensorFlow

DEV SUMMIT 2019

```
model.compile(loss=tf.keras.mean_squared_error)
model.compile(loss=lambda y, rv_y: -rv_y.log_prob(y)) } Our wish.
yhat = model(x_tst) # A Distribution, not a Tensor!
```

SEE THIS GREAT TALK: <https://www.youtube.com/watch?v=BrwKURU-wpk>

BUT REMEMBER, WE ARE CHOOSING ONE MODEL AND ONE VALUE FOR THE WEIGHTS



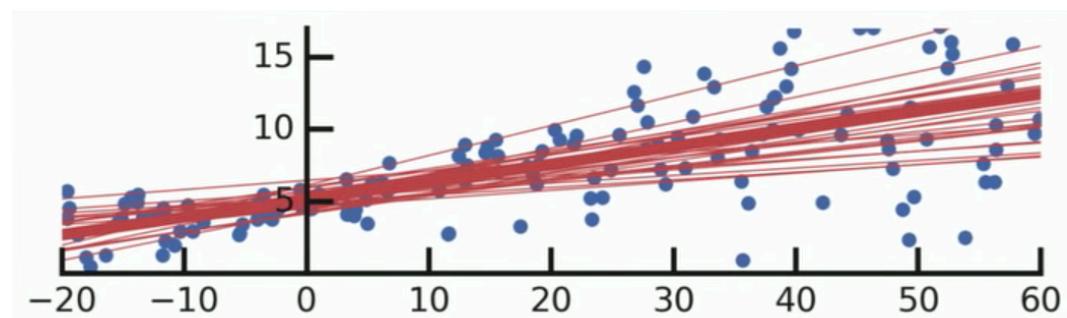
IDEALLY, WE WOULD LIKE TO MARGINALIZE OVER ALL MODELS AND OVER ALL POSSIBLE WEIGHT VALUES

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

CAPTURING “MODEL UNCERTAINTY”

```
model = tf.keras.Sequential([
    tfp.layers.DenseVariational(hidden_units, ...), },
    tfp.layers.DenseVariational(1), },
    tfp.layers.DistributionLambda(lambda t:
        tfd.Normal(loc=t[...], 0),
        scale=1)),
])
```

} Bayesian Weights
} Linear Regression



AND YOU STILL ARE LEFT WITH THE CHOICE OF THE MODEL ARCHITECTURE:

Introduction to the Keras Tuner

[Run in Google Colab](#)[View source on GitHub](#)[Download notebook](#)

Overview

The Keras Tuner is a library that helps you pick the optimal set of hyperparameters for your TensorFlow program. The process of selecting the right set of hyperparameters for your machine learning (ML) application is called *hyperparameter tuning* or *hypertuning*.

Hyperparameters are the variables that govern the training process and the topology of an ML model. These variables remain constant over the training process and directly impact the performance of your ML program. Hyperparameters are of two types:

1. **Model hyperparameters** which influence model selection such as the number and width of hidden layers
2. **Algorithm hyperparameters** which influence the speed and quality of the learning algorithm such as the learning rate for Stochastic Gradient Descent (SGD) and the number of nearest neighbors for a k Nearest Neighbors (KNN) classifier

In this tutorial, you will use the Keras Tuner to perform hypertuning for an image classification application.

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 INITIAL WEIGHTS AND THE
LEARNING RATES ARE IMPORTANT

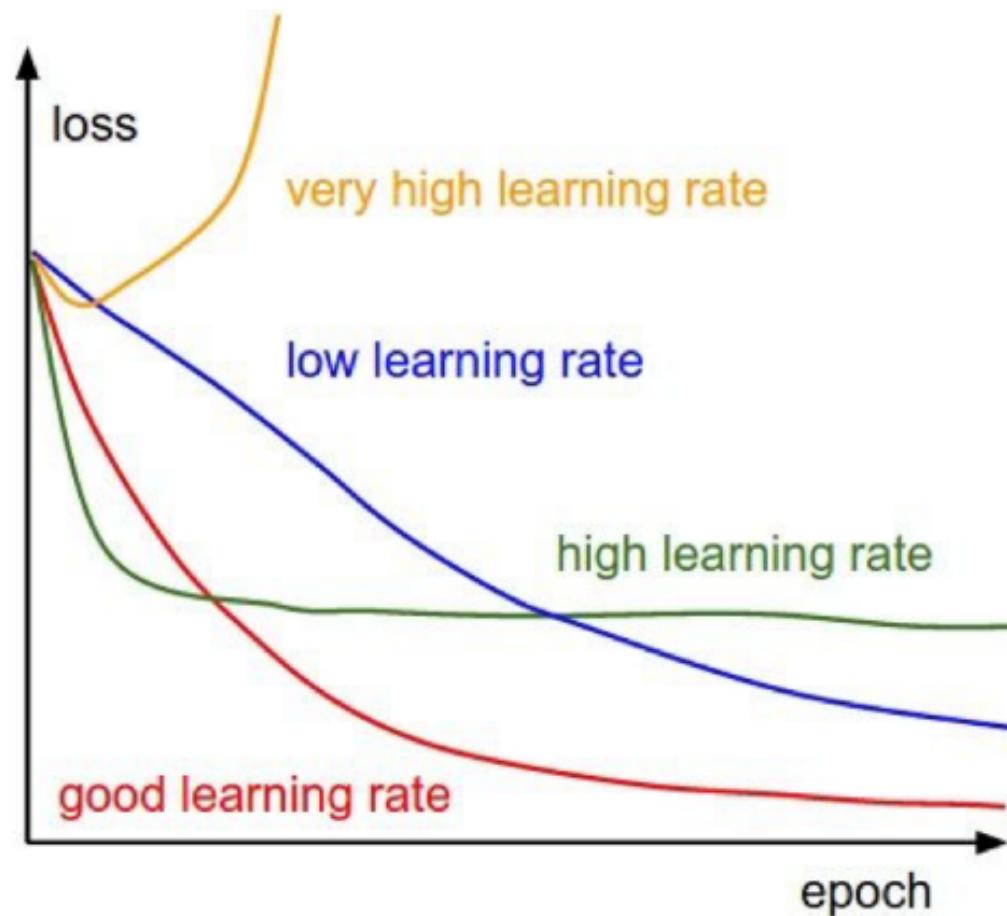
ONE KEY PROBLEM WITH GRADIENT DESCENT IS THAT IT
EASILY CONVERGES TO LOCAL MINIMA BY FOLLOWING
THE STEEPEST DESCENT

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

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.

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

Credit:

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

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

Credit:

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 ≥ 0 . Learning rate.
- `rho`: float ≥ 0 .
- `epsilon`: float ≥ 0 . Fuzz factor. If `None`, defaults to `K.epsilon()`.
- `decay`: float ≥ 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.

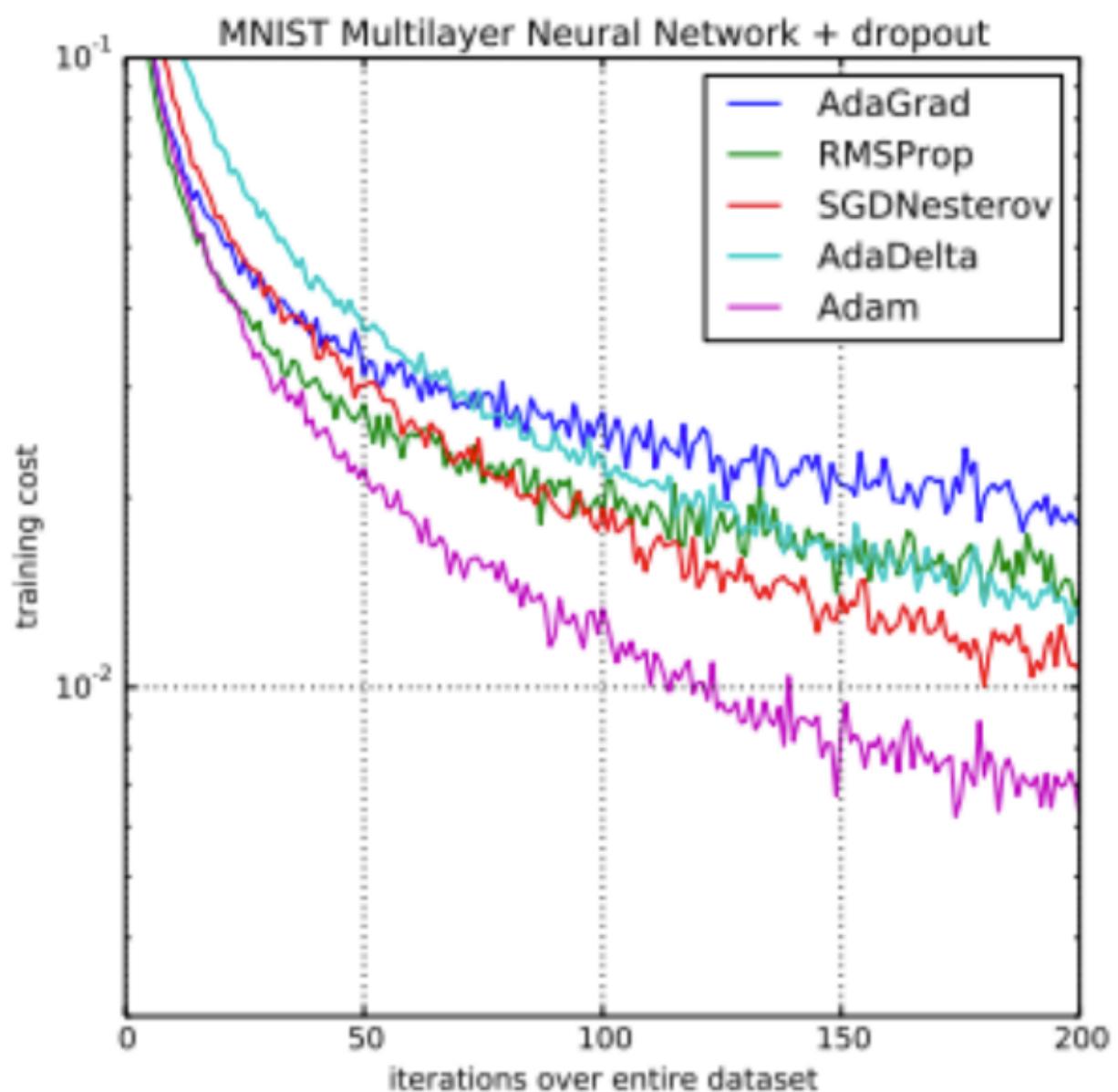
Default parameters follow those provided in the original paper.

Arguments

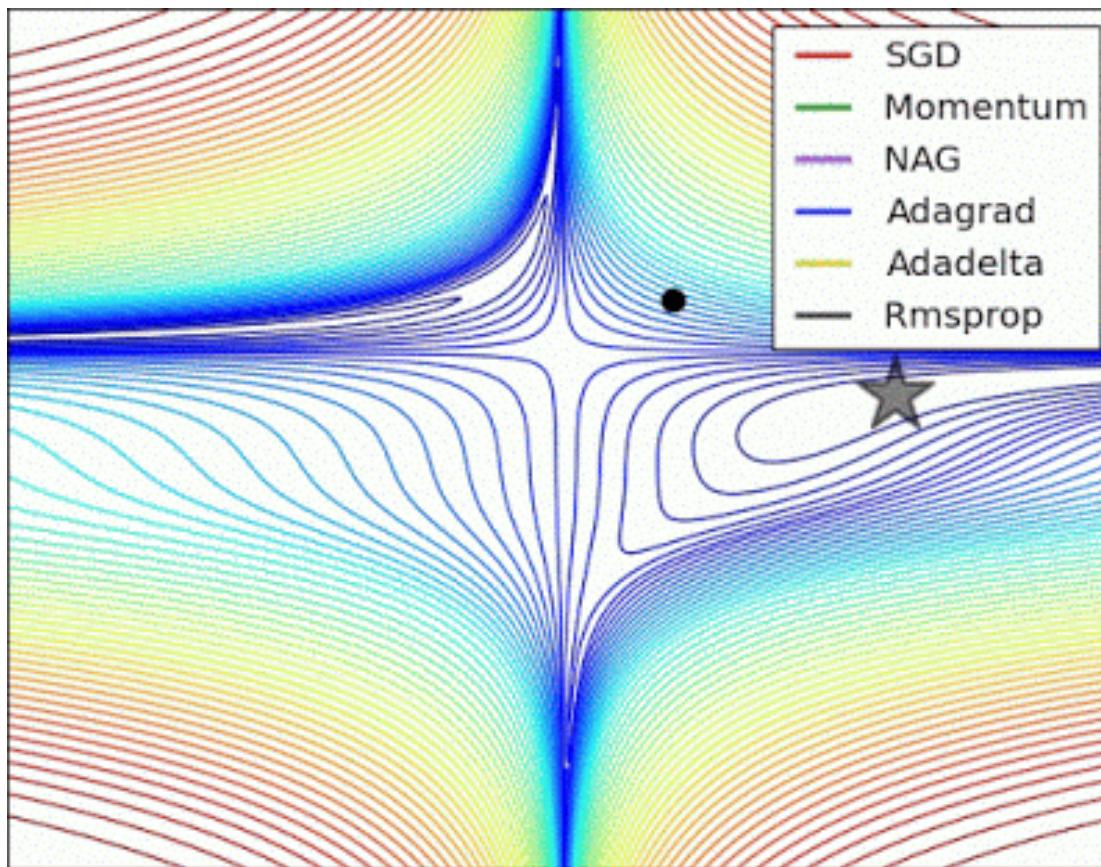
- `lr`: float ≥ 0 . Learning rate.
- `beta_1`: float, $0 < \beta_1 < 1$. Generally close to 1.
- `beta_2`: float, $0 < \beta_2 < 1$. Generally close to 1.
- `epsilon`: float ≥ 0 . Fuzz factor. If `None`, defaults to `K.epsilon()`.
- `decay`: float ≥ 0 . Learning rate decay over each update.
- `amsgrad`: 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

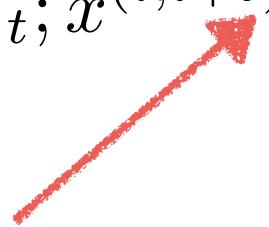
LOCAL MINIMA CAN ALSO BE AVOIDED BY COMPUTING THE
GRADIENT IN SMALL BATCHES INSTEAD OF OVER THE FULL
DATASET

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

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



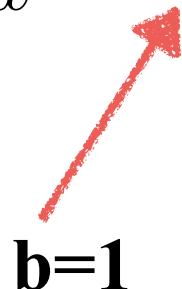
THE GRADIENT IS COMPUTED OVER A BATCH OF SIZE B

STOCHASTIC GRADIENT DESCENT

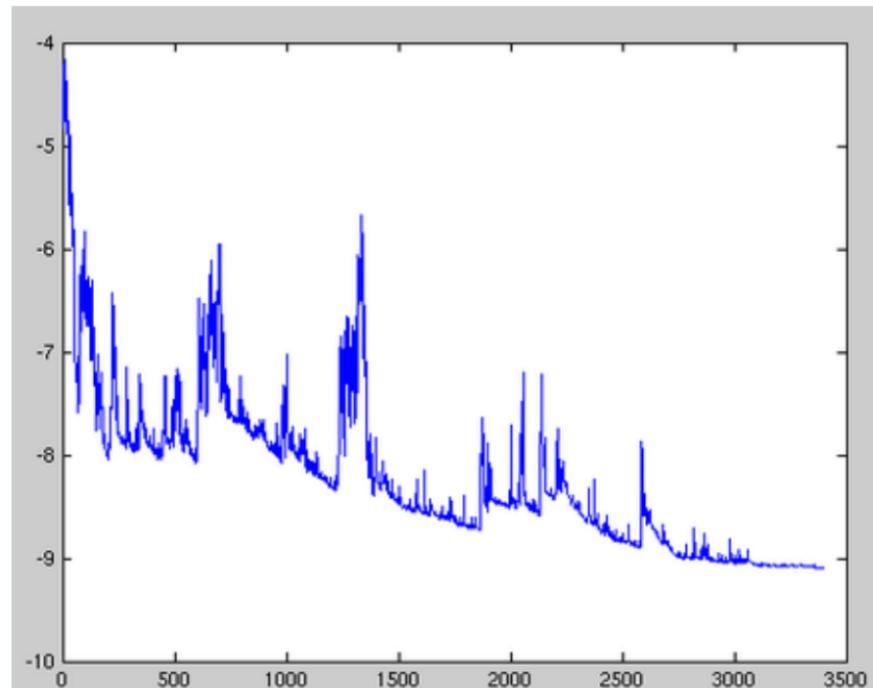
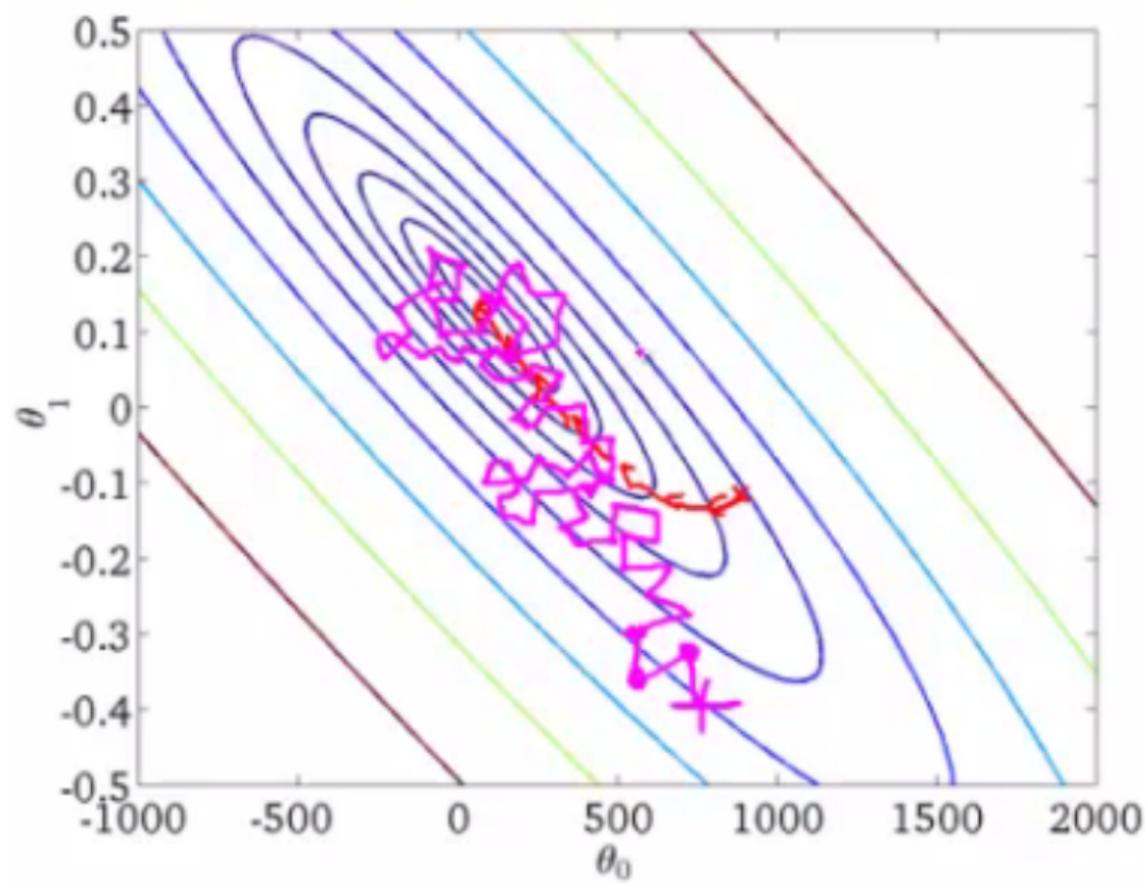
THE EXTREME CASE IS TO COMPUTE THE GRADIENT ON EVERY TRAINING EXAMPLE.

STOCHASTIC GRADIENT DESCENT

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



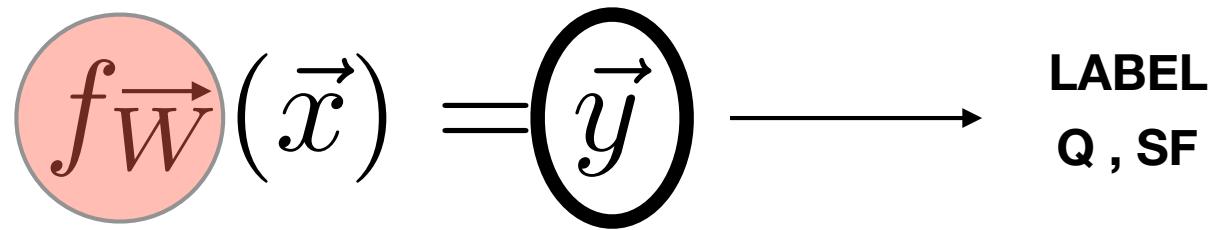
b=1



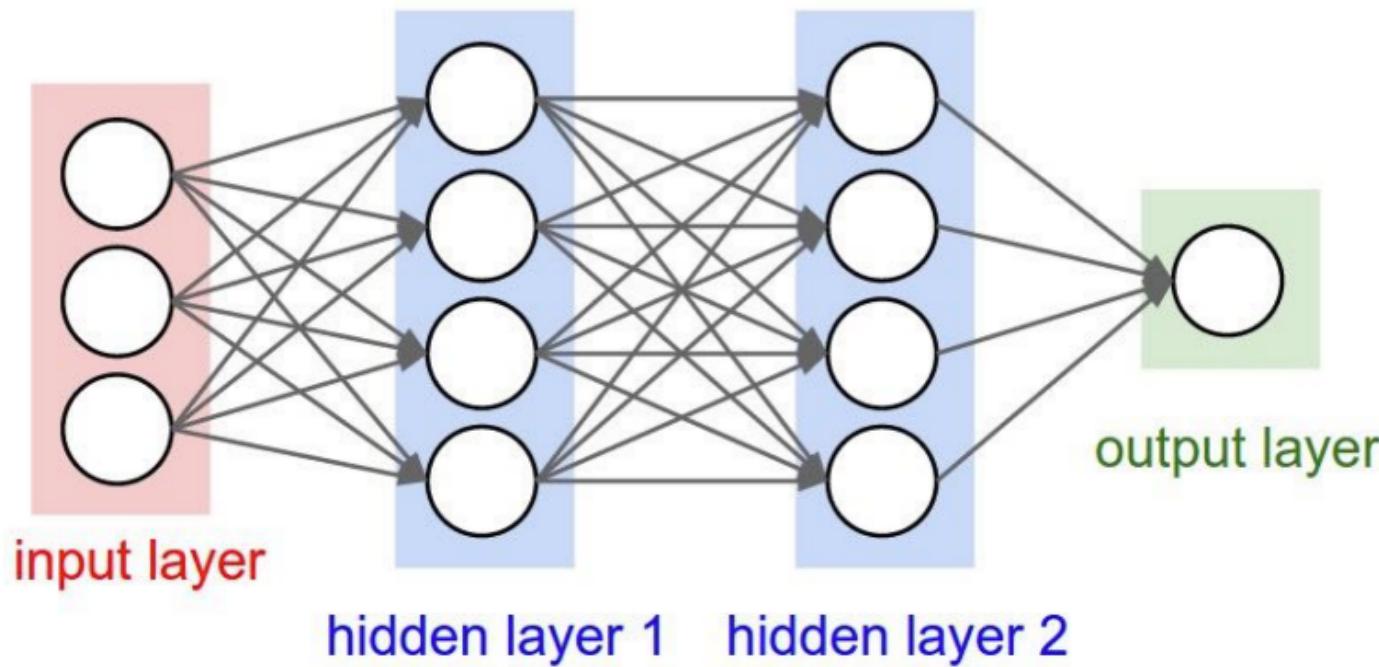
Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

Credit

**"CLASSICAL"
MACHINE LEARNING**



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



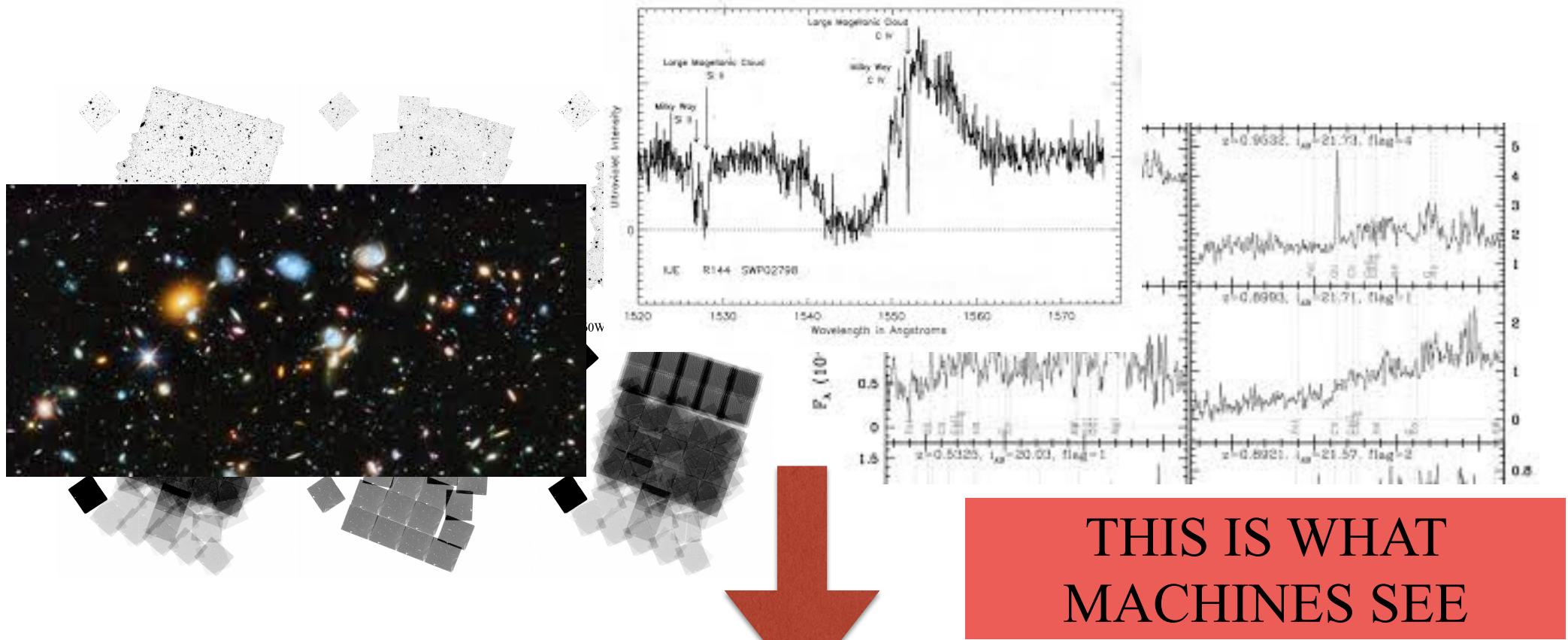
$$p = g_3(W_3g_2(W_2g_1(W_1\vec{x}_0))) \xleftarrow{\text{NETWORK FUNCTION}}$$

CAN WE GO DEEP NOW?

CAN WE GO DEEP NOW?

ALMOST THERE...LET'S THINK FOR A
MOMENT ABOUT WHAT WE PUT AS
INPUT...

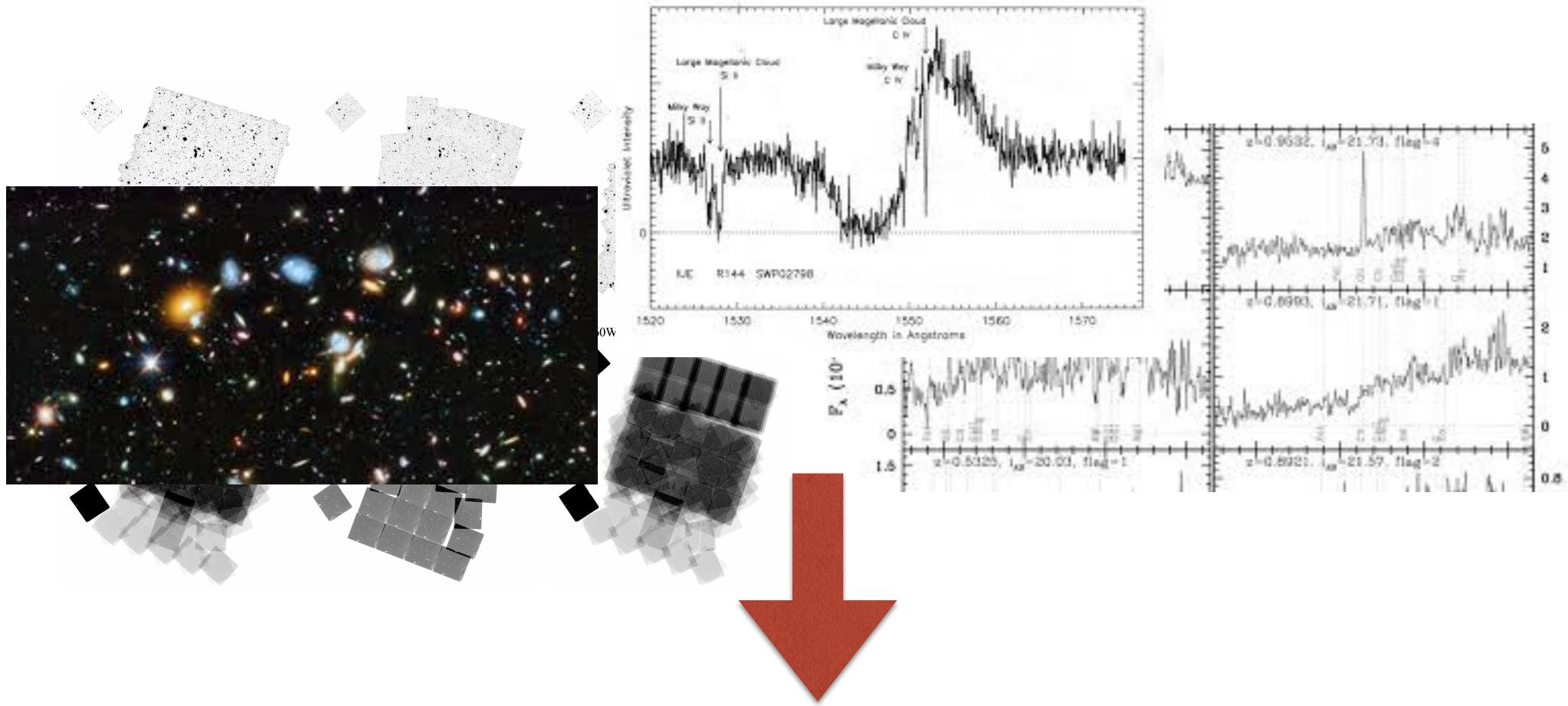
What do we put as input?



THIS IS WHAT MACHINES SEE

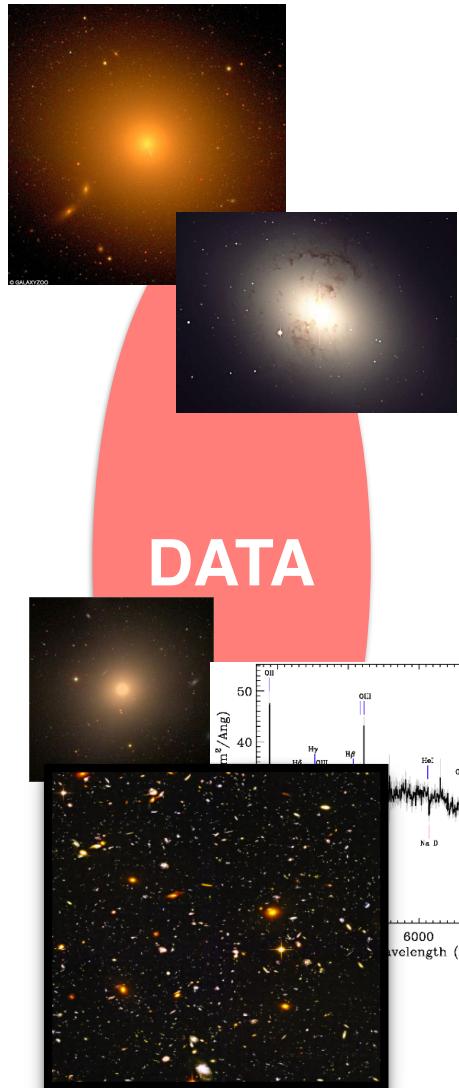
A magnifying glass is positioned over a grid of binary digits (0s and 1s) on a light blue background. The digits are arranged in a repeating pattern. The magnifying glass is focused on a specific group of four binary digits: 0101. A red arrow points from the top right towards the magnifying glass. The background consists of a large number of binary digits, creating a digital or data processing theme.

What do we put as input?



PRE-PROCESS DATA TO EXTRACT MEANINGFUL INFORMATION

THIS IS GENERALLY CALLED **FEATURE EXTRACTION**



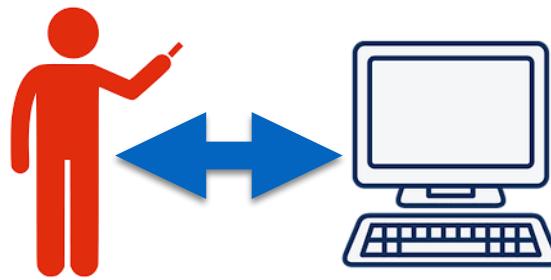
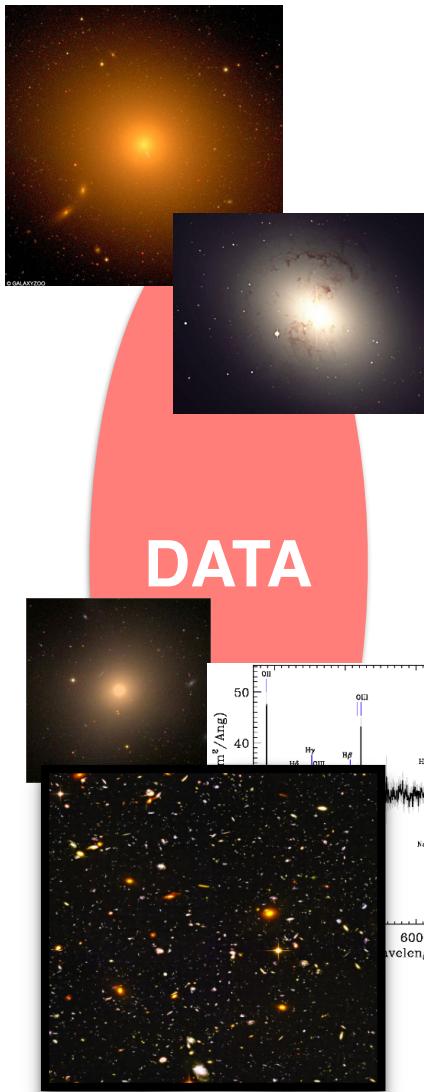
Spiral!

Emission line!

Merger!

Clump!

AGN!



Spiral!

Emission line!

Merger!

Clump!

AGN!

$$f_W(\vec{x}) = \vec{y} \longrightarrow \text{LABEL}$$

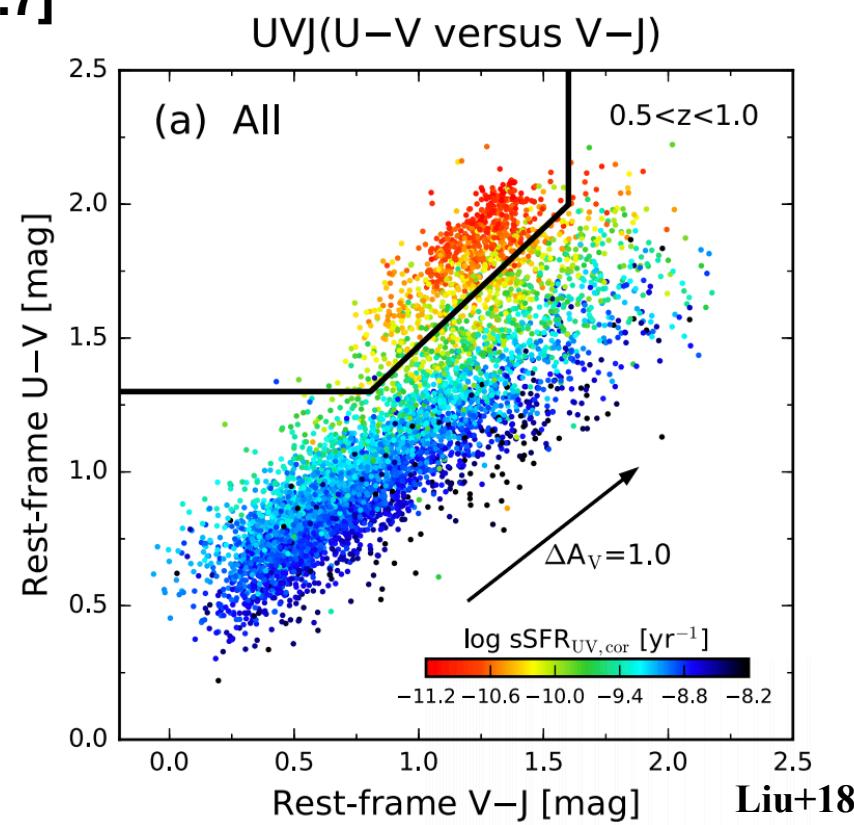
Q(0) , SF(1)

NETWORK FUNCTION

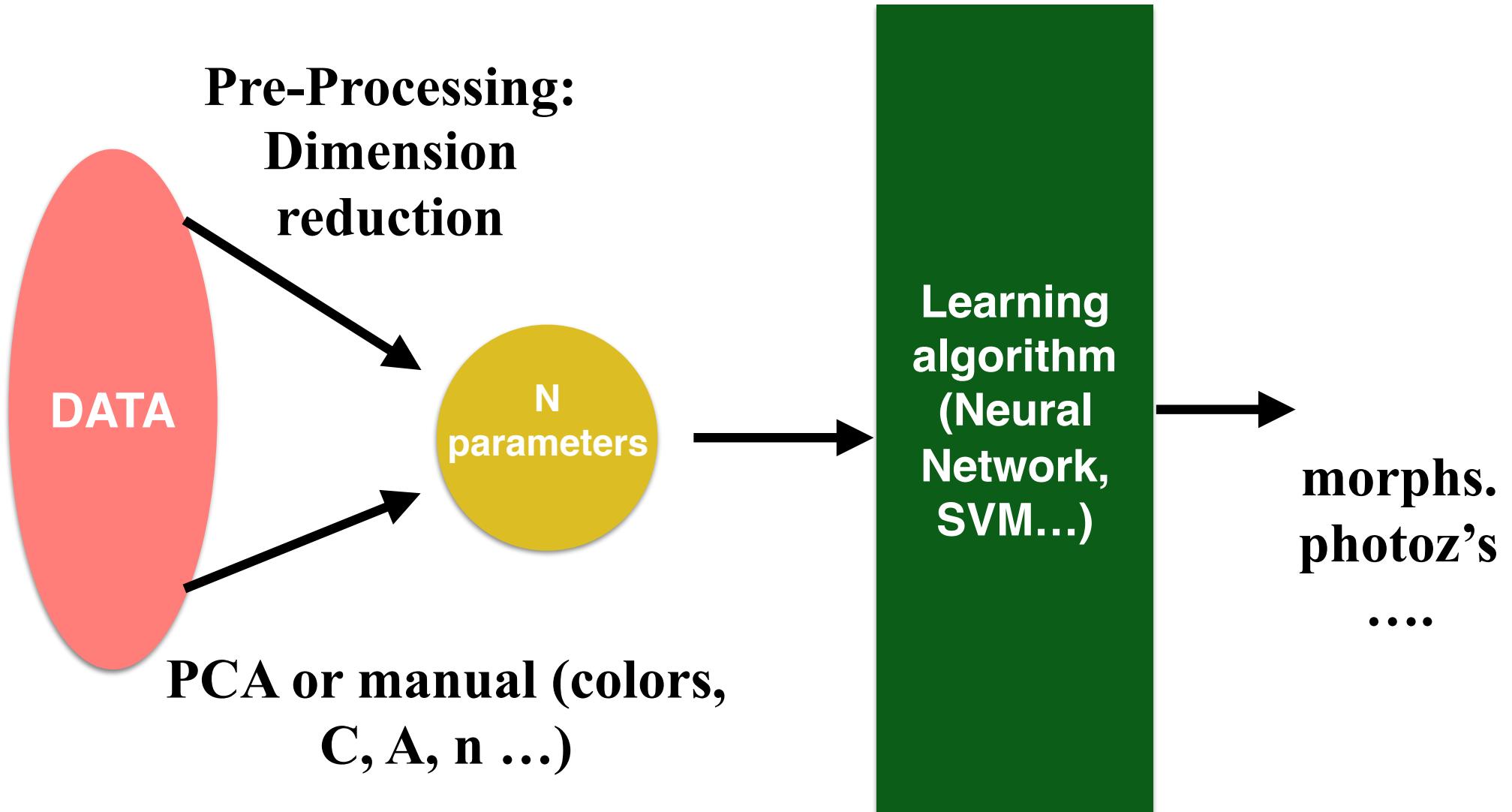
(U-V, V-J) FEATURES

$$\text{sgn}[(u-v)-0.8*(v-j)-0.7]$$

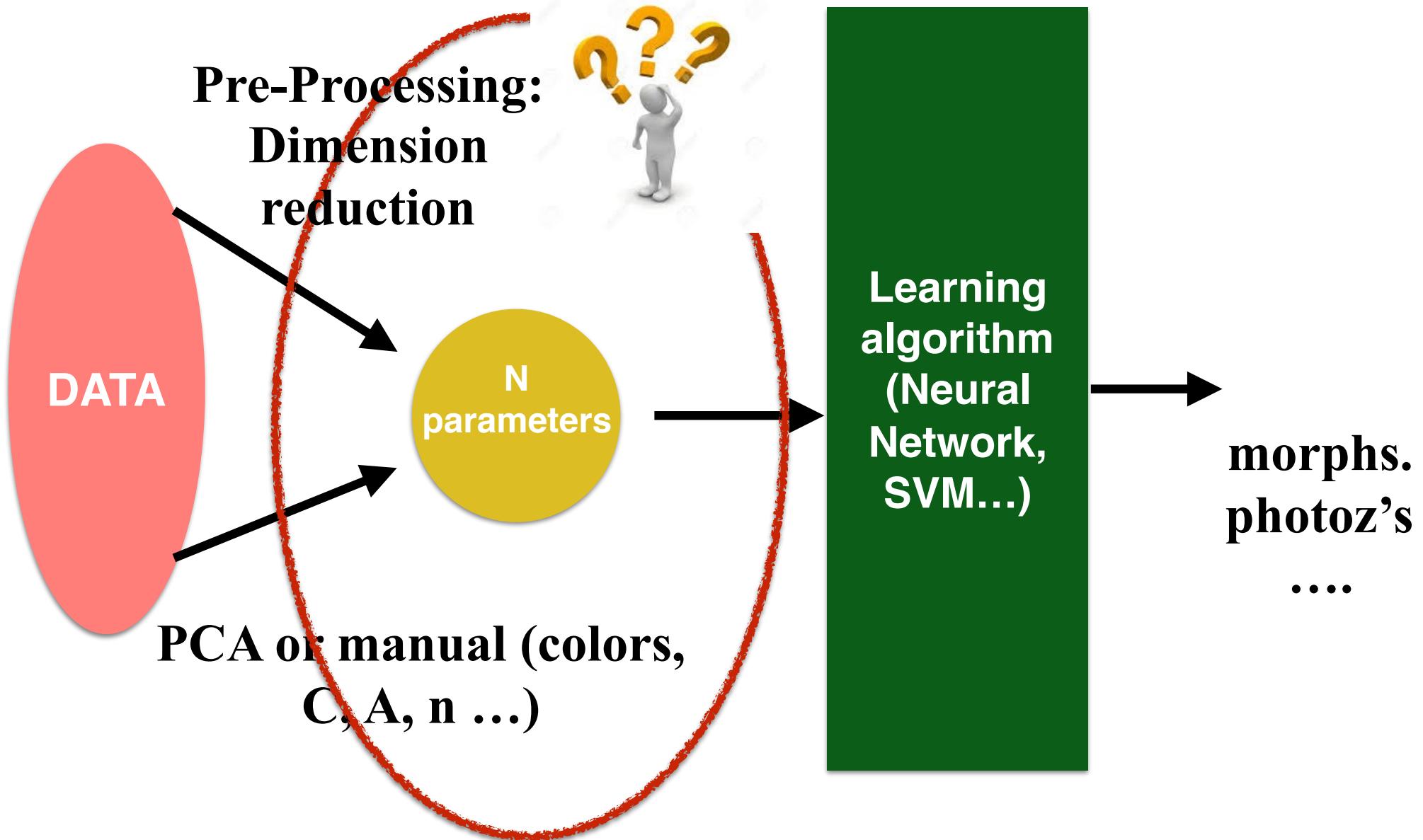
WEIGHTS



THE “CLASSICAL” APPROACH

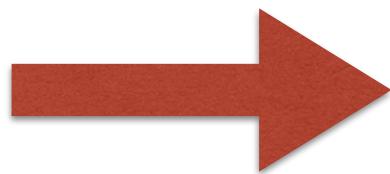


“CLASSICAL” MACHINE LEARNING



In Astronomy

- Colors, Fluxes
- Shape indicators
- Line ratios, spectral features
- Stellar Masses, Velocity Dispersions



Requires specialized software before feeding the machine learning algorithm

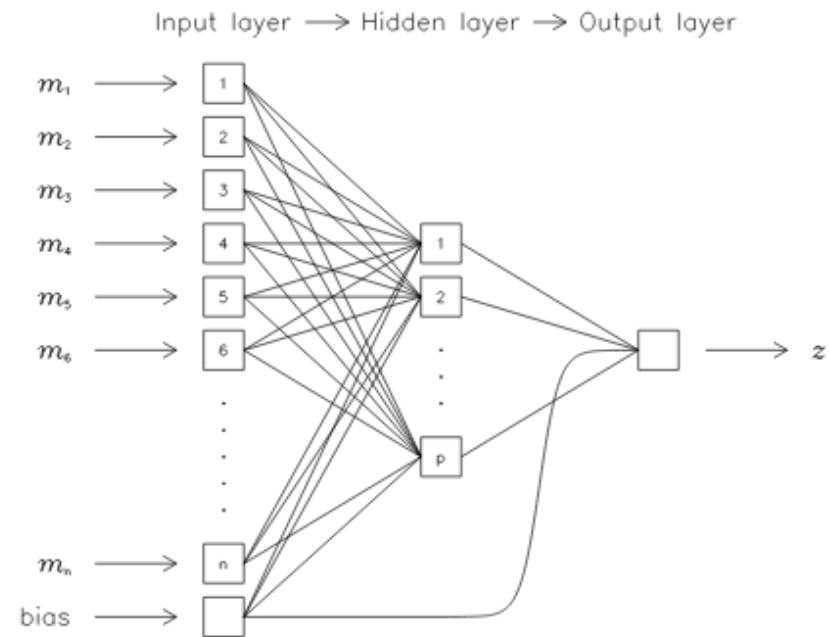
IT IMPLIES A DIMENSIONALITY REDUCTION!

PHOTOMETRIC REDSHIFTS

SDSS

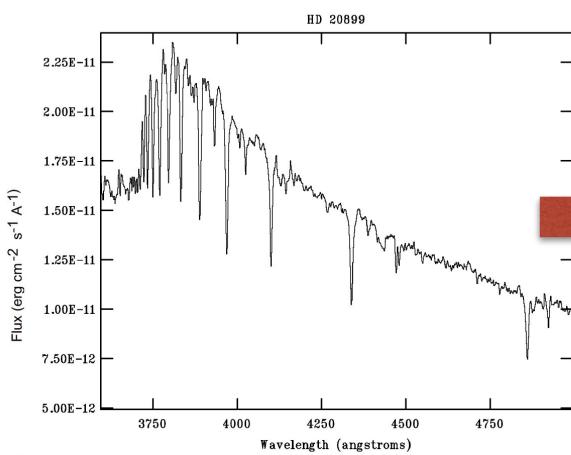


g
r
i
z

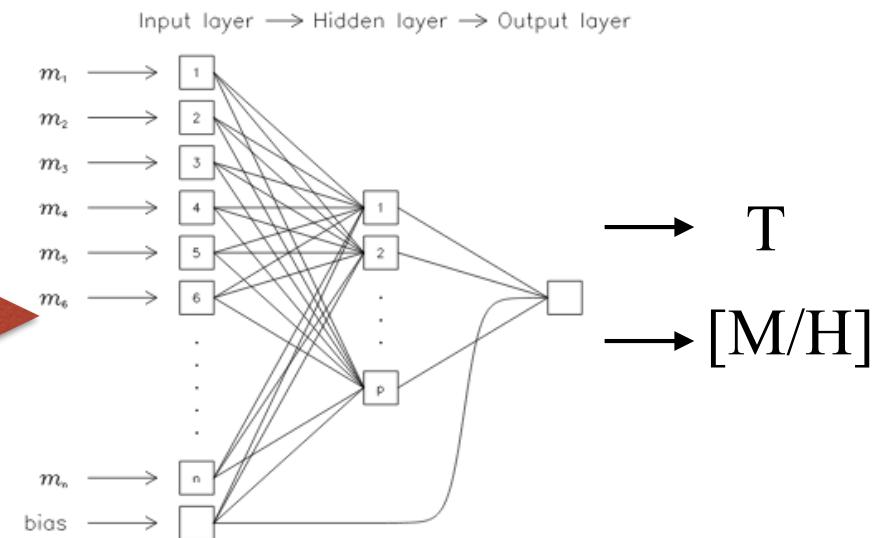


Collister+08

STELLAR PARAMETERS FROM MEDIUM BAND FILTERS



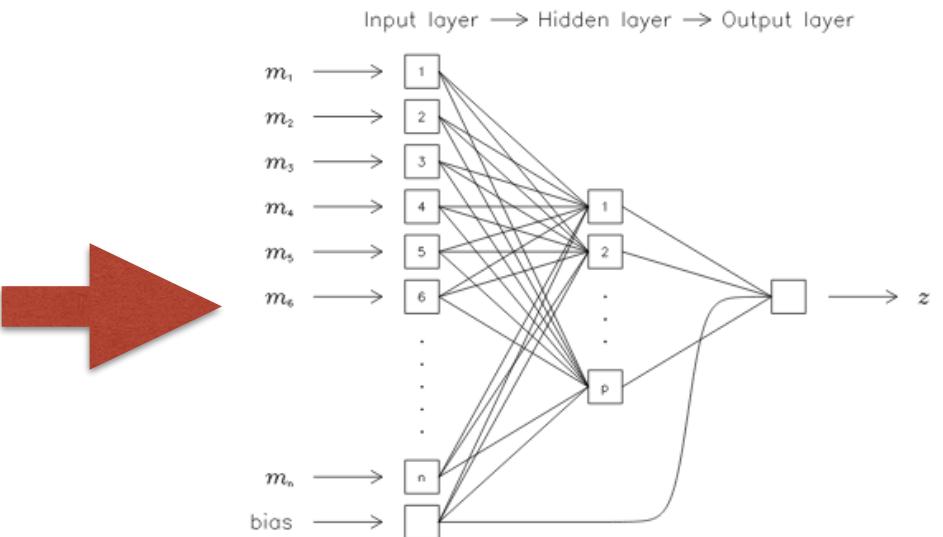
MEDIUM
BAND
FLUXES



Bailer-Jones+00

No.	Symbol	Description	Scale ^a
1	CVD	Central velocity dispersion	~1 kpc
2	M_{bulge}	Bulge stellar mass	0.5–4 kpc
3	R_e	Bulge effective radius	0.5–4 Kpc
4	B/T	Bulge-to-total stellar mass ratio	0.5–8 kpc
5	M_*	Total stellar mass	2–8 kpc
6	M_{disc}	Disc stellar mass	4–10 kpc
7	M_{halo}	Group halo mass	0.1–1 Mpc
8	δ_5	Local density parameter	0.5–3 Mpc

Notes. ^a Approximate 1σ range from centre of galaxy. For photometric quantities half-light radii are used.



HEAVILY PROCESSED DATA

Other general computer vision features [for images!]

- Pixel Concatenation
- Color histograms
- Texture Features
- Histogram of Gradients
- SIFT

FOR MANY YEARS COMPUTER VISION RESEARCHERS HAVE BEEN TRYING TO FIND THE MOST GENERAL FEATURES

Other general computer vision features [for images!]

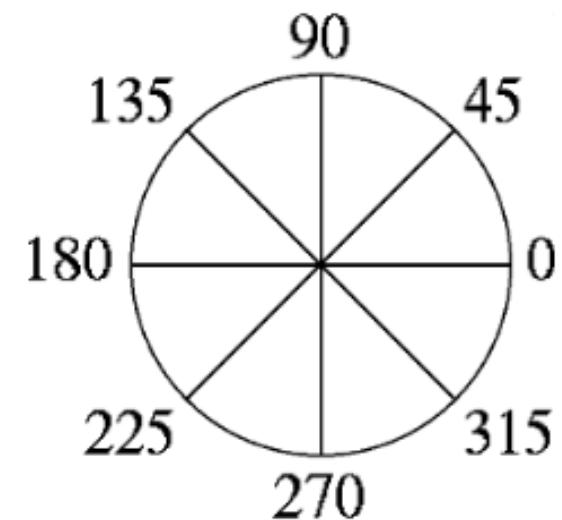
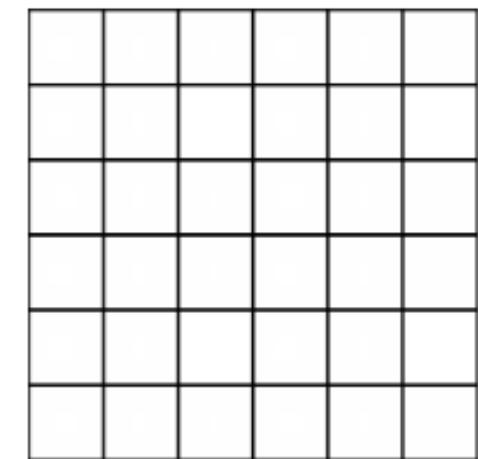
- Pixel Concatenation
- Color histograms
- Texture Features
- Histogram of Gradients
- SIFT

FOR MANY YEARS COMPUTER VISION RESEARCHERS HAVE BEEN TRYING TO FIND THE MOST GENERAL FEATURES

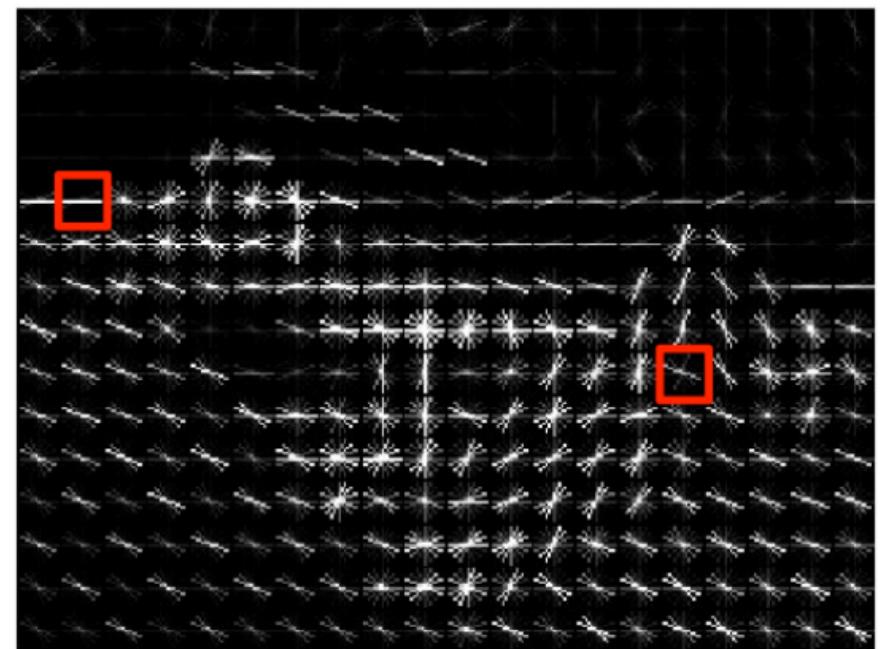
THE BEST CLASSICAL SOLUTION [BEFORE 2012] WHERE BASED ON LOCAL FEATURES

HISTOGRAM OF ORIENTED GRADIENTS (HoG)

1. DIVIDE IMAGE INTO SMALL SPATIAL REGIONS CALLED CELLS
2. COMPUTE INTENSITY GRADIENTS OVER N DIRECTIONS [TYPICALLY 9 FOR IMAGE]
3. COMPUTE WEIGHTED 1-D HISTOGRAM OF ALL DIRECTIONS. A CELL IS REDUCED TO N NUMBERS



HISTOGRAM OF ORIENTED GRADIENTS (HoG)



**EVERYTHING IS IN THE FEATURES....WHAT IF I
IGNORED SOME IMPORTANT FEATURES?**



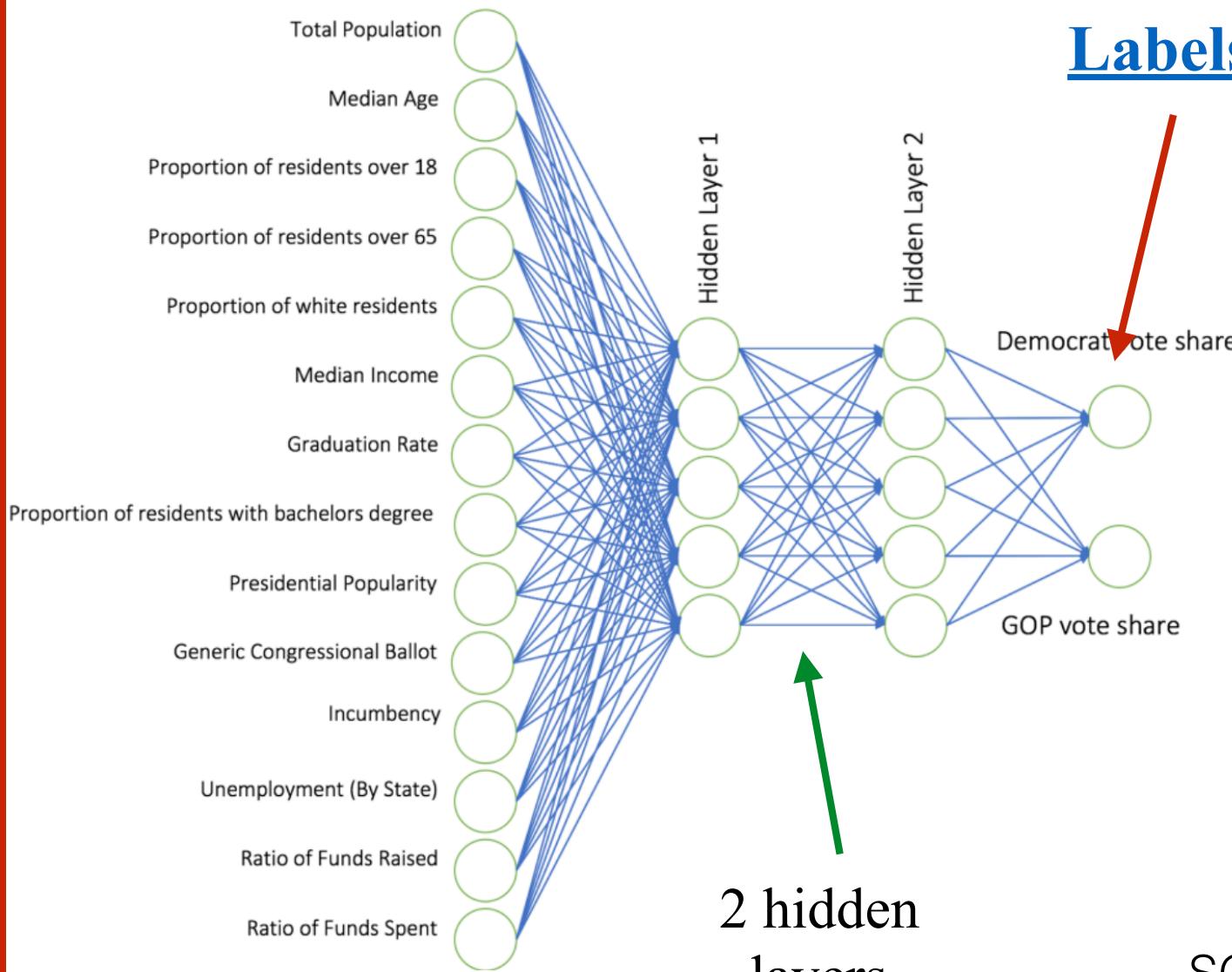
**EVERYTHING IS IN THE FEATURES....WHAT IF I
IGNORED SOME IMPORTANT FEATURES?**



NEURAL NETWORK TO PREDICT RESULTS OF MIDTERM ELECTIONS

$$p = g_3(W_3g_2(W_2g_1(W_1\vec{x}_0)))$$

Features
(x)



SOURCE

NEURAL NETWORK TO PREDICT RESULTS OF MIDTERM ELECTIONS

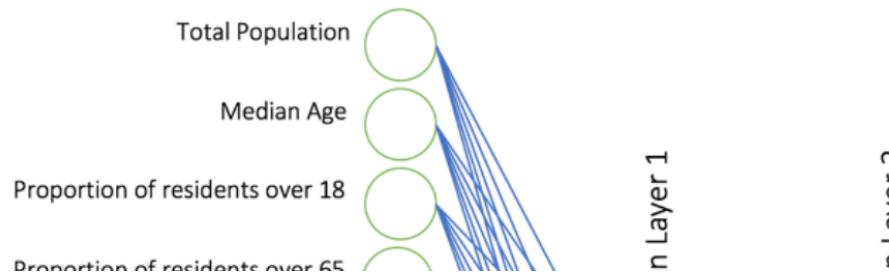
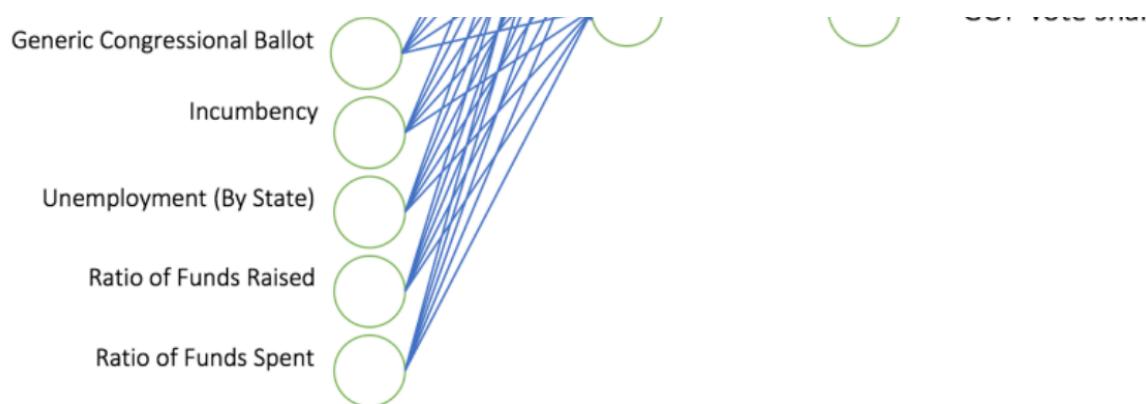


Table 2 – Results of both Models:

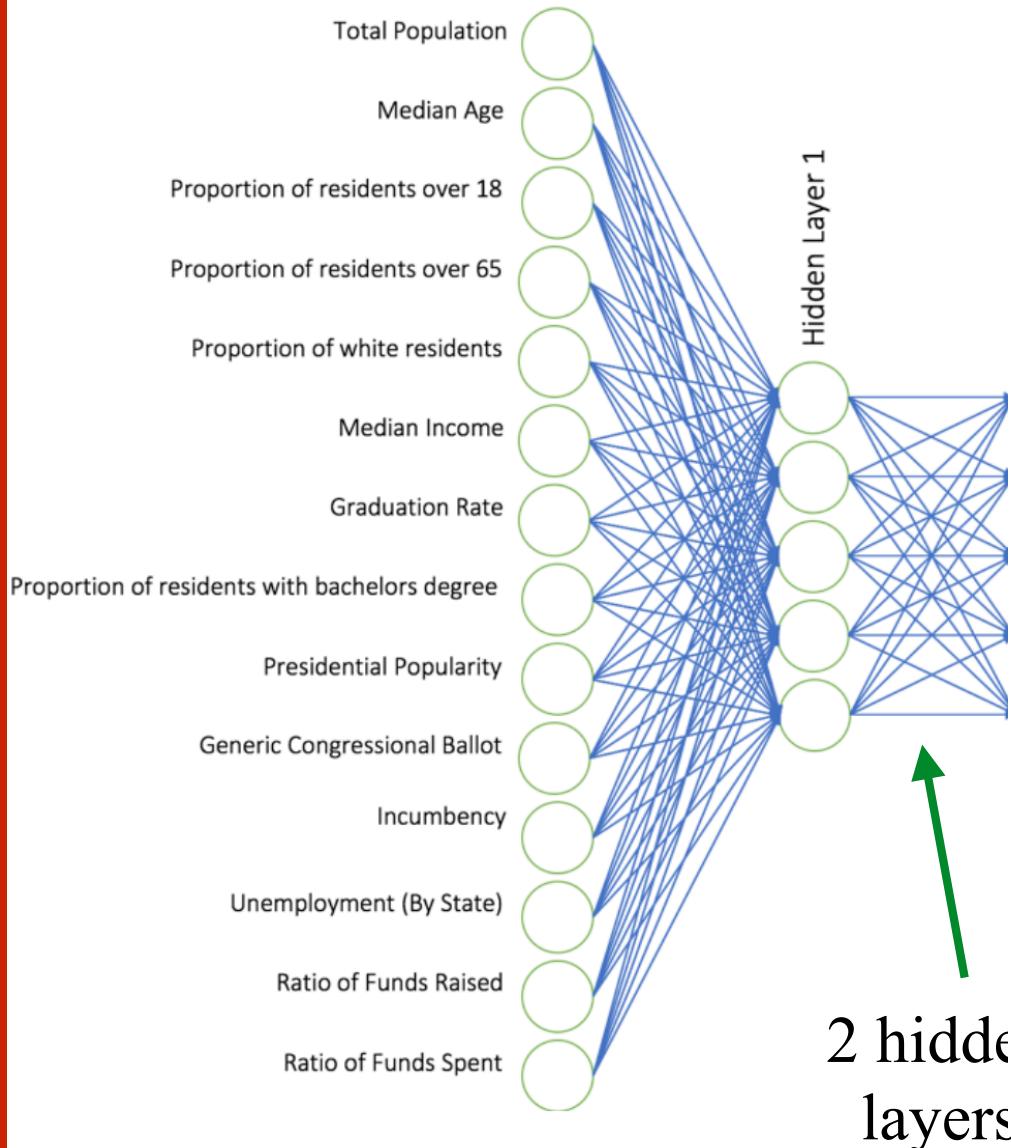
<i>Model not including past elections (Model A)</i>	<i>Model including past elections (Model B)</i>
D +3	D +17
Democrat Seats: 219	Democrat Seats: 226
Republican Seats: 216	Republican Seats: 209
33% chance of Republicans keeping house*	0.3% chance of Republicans keeping house*



232+198

NEURAL NETWORK TO PREDICT RESULTS OF MIDTERM ELECTIONS

Features (x)



Bad Weather, Known to Lower Turnout, Will Greet Many Voters

Rain can decrease voter numbers, which studies show tends to help Republicans. “I hope it rains hard tomorrow,” one Republican candidate said.

10h ago

WHAT ABOUT USING RAW DATA?

ALL INFORMATION IS IN THE INPUT DATA

WHY REDUCING ?

LET THE NETWORK FIND THE INFO

WHAT ABOUT USING RAW DATA?

ALL INFORMATION IS IN THE INPUT DATA

WHY REDUCING ?

LET THE NETWORK FIND THE INFO

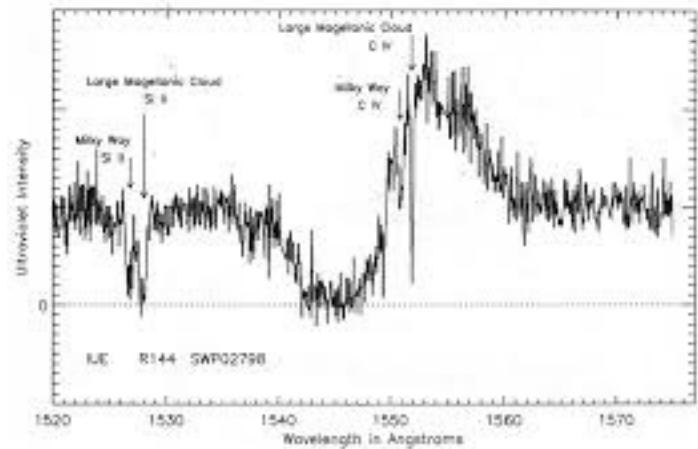
LARGE DIMENSION SIGNALS SUCH AS IMAGES OR SPECTRA WOULD REQUIRE TREMENDOUSLY LARGE MODELS

A 512x512 image as input of a fully connected layer producing output of same size:

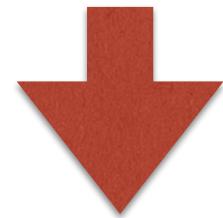
$$(512 \times 512)^2 = 7e10$$

BUT

FEEDING INDIVIDUAL RESOLUTION ELEMENTS IS NOT
VERY EFFICIENT SINCE IT LOOSES ALL INVARIANCE TO
TRANSLATION AND IGNORES CORRELATION IN THE DATA
AT ALL SCALES

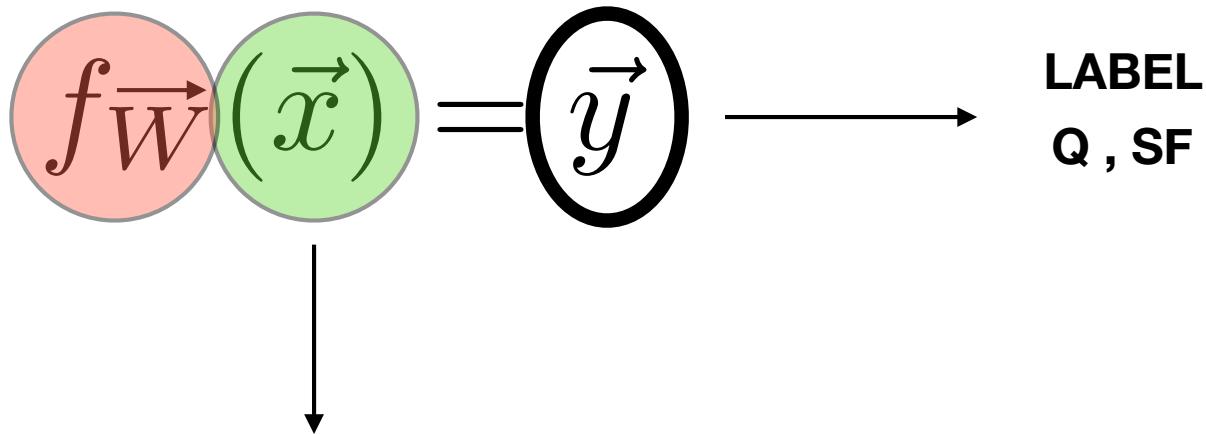


FEEDING INDIVIDUAL RESOLUTION ELEMENTS IS NOT
VERY EFFICIENT SINCE IT LOSES ALL INVARIANCE TO
TRANSLATION



SO?

DEEP LEARNING



LET THE MACHINE FIGURE THIS OUT (“unsupervised feature extraction”)

LET'S GO A STEP FORWARD INTO LOOSING CONTROL....