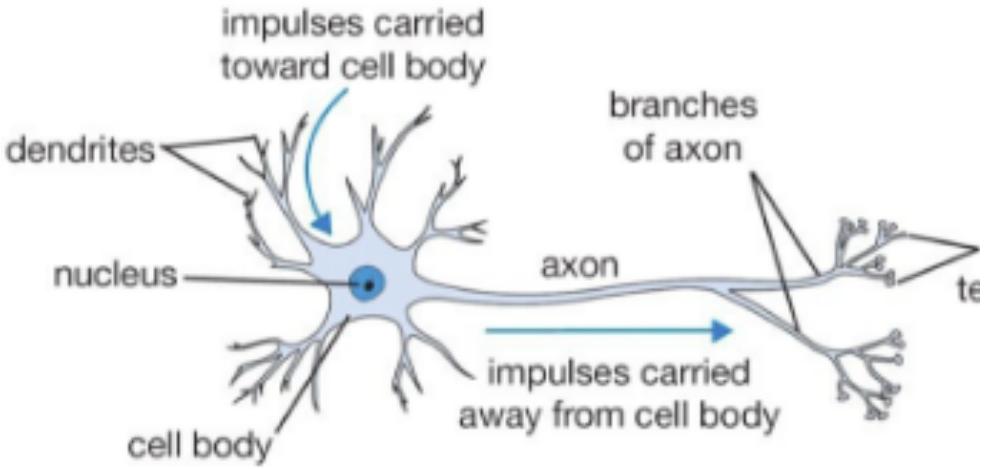
PART II: FOUNDATIONS OF "SHALLOW" NEURAL NETWORKS

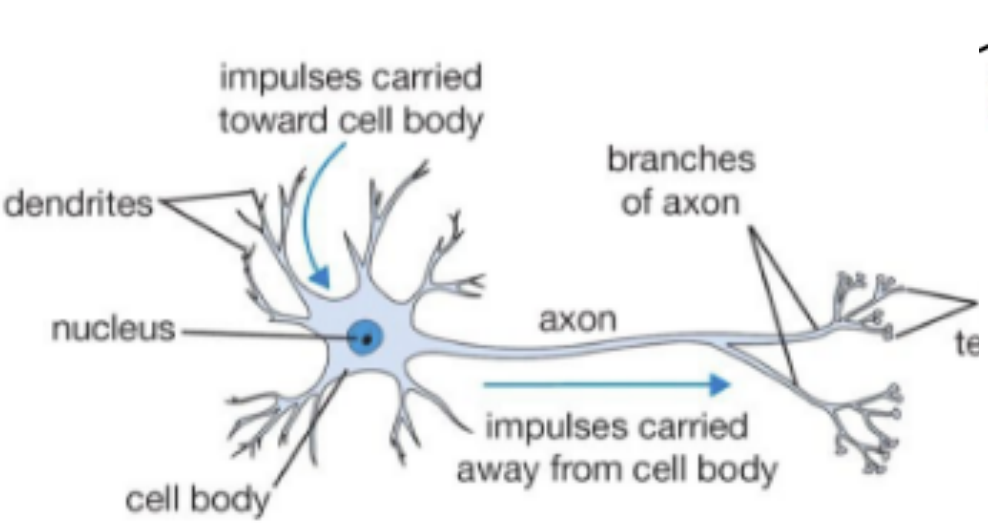
THE NEURON

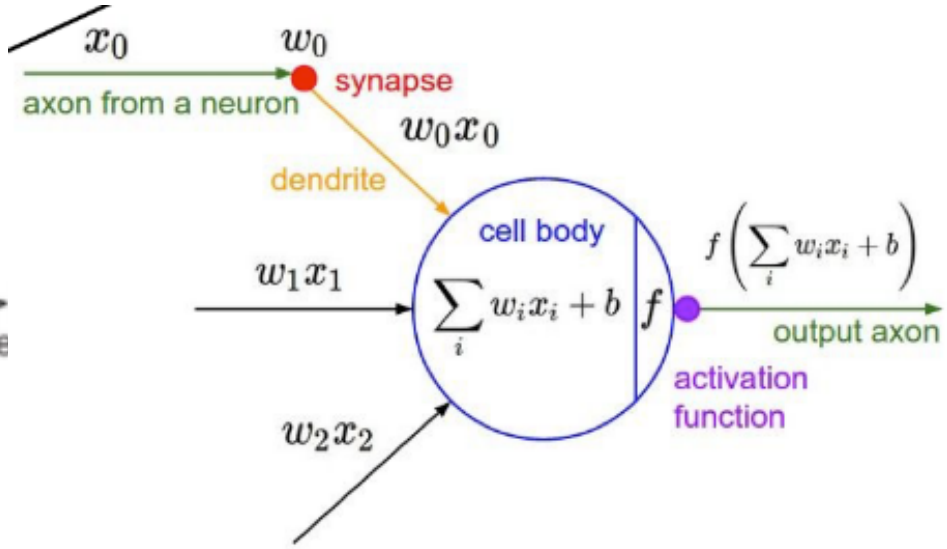


INSPIRED BY NEURO - SCIENCE?

Credit: Karpathy

THE NEURON





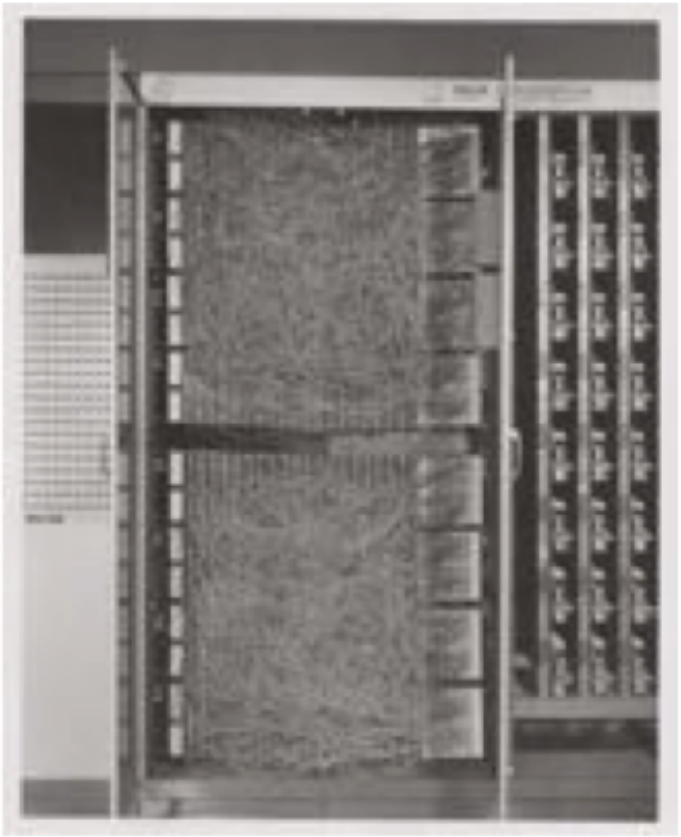
INSPIRED BY NEURO - SCIENCE?

Credit: Karpathy

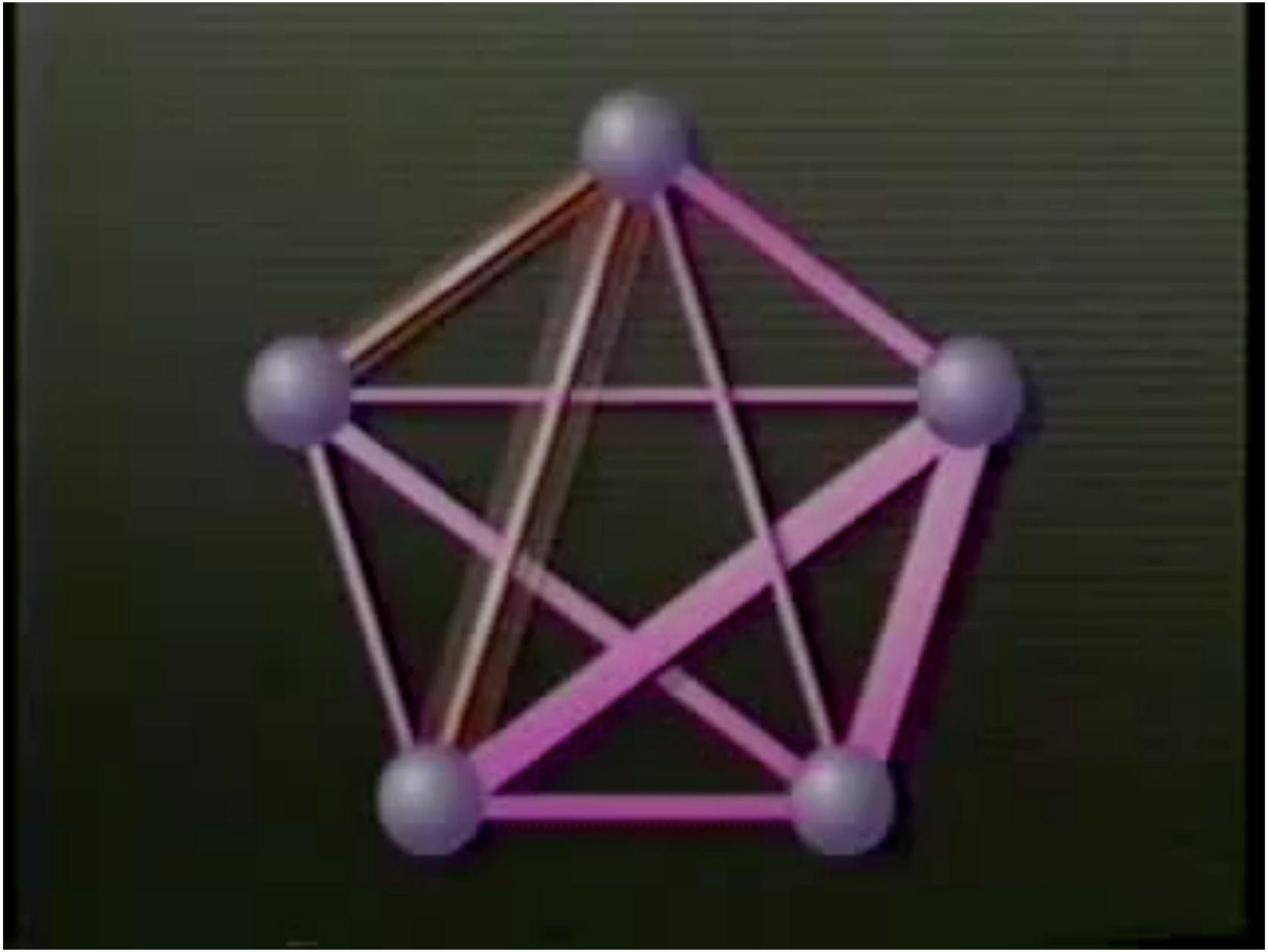
Rosenblatt Perceptron

FIRST IMPLEMENTATION OF NEURAL NETWORK [Rosenblatt, **1957!**]

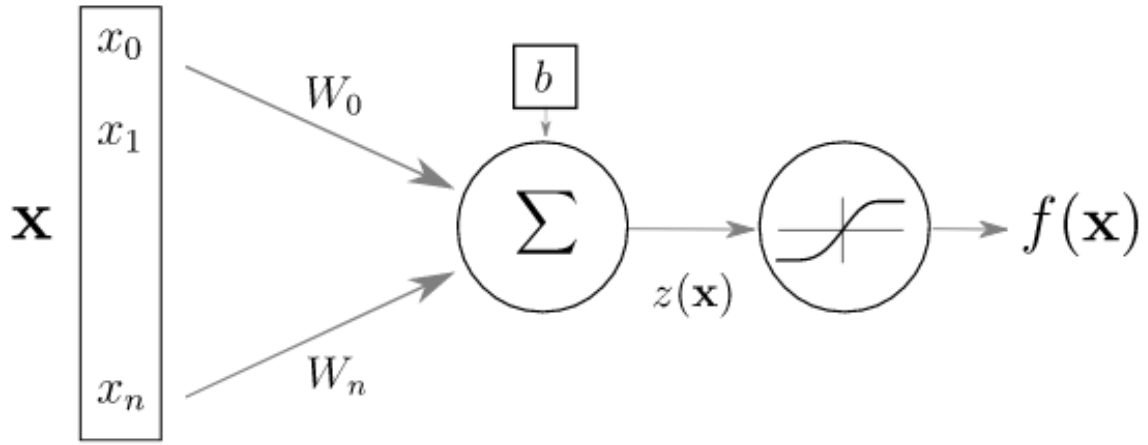
INTENDED TO BE A MACHINE (NOT AN ALGORITHM)



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



$$z(\vec{x}) = \vec{W}.\vec{x} + b \qquad f(\vec{x}) = g(\vec{W}.\vec{x} + b)$$

Pre-Activation

Weights

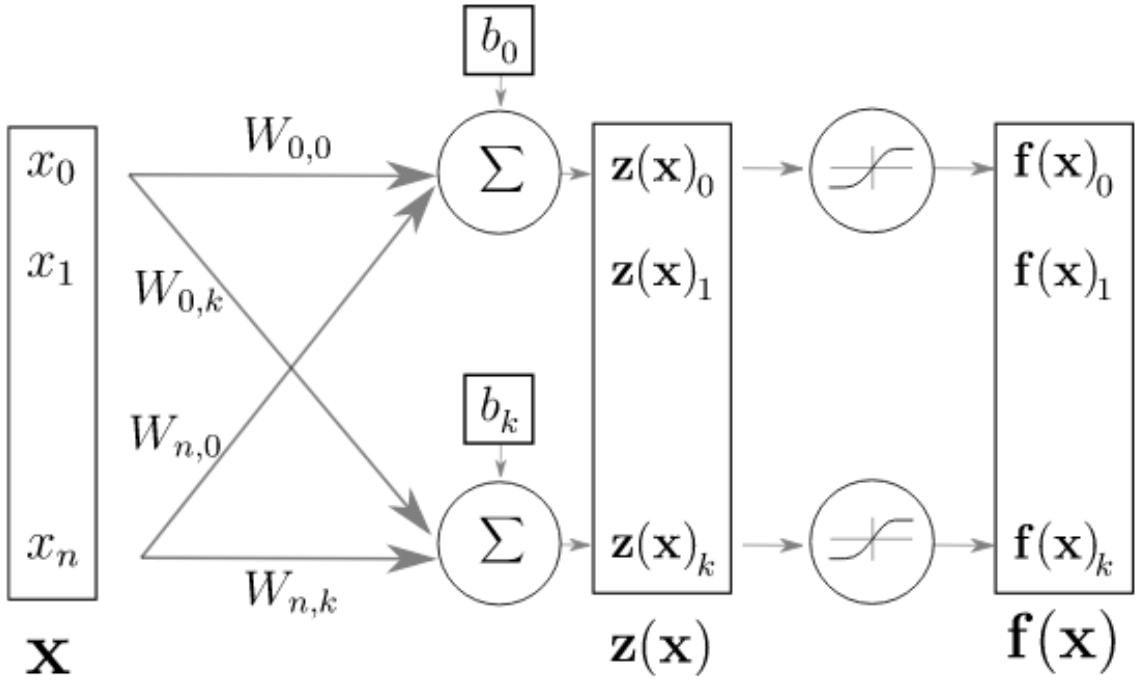
Input

Bias

Output

Activation Function

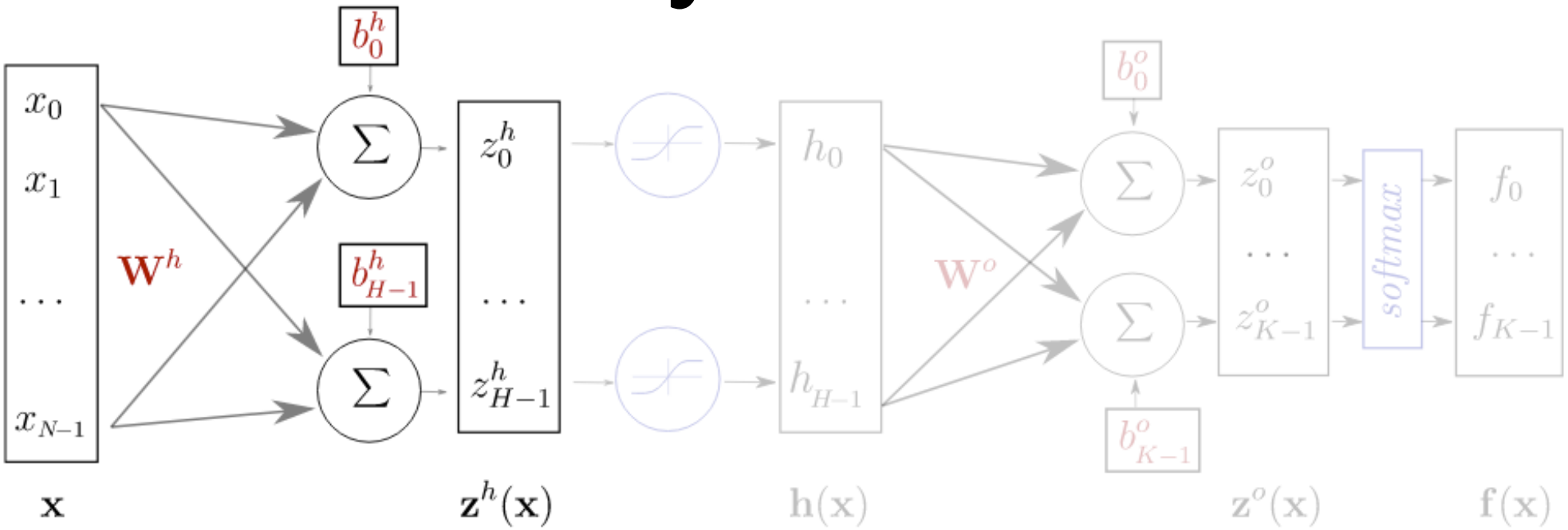
LAYER OF NEURONS



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

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

Hidden Layers of Neurons

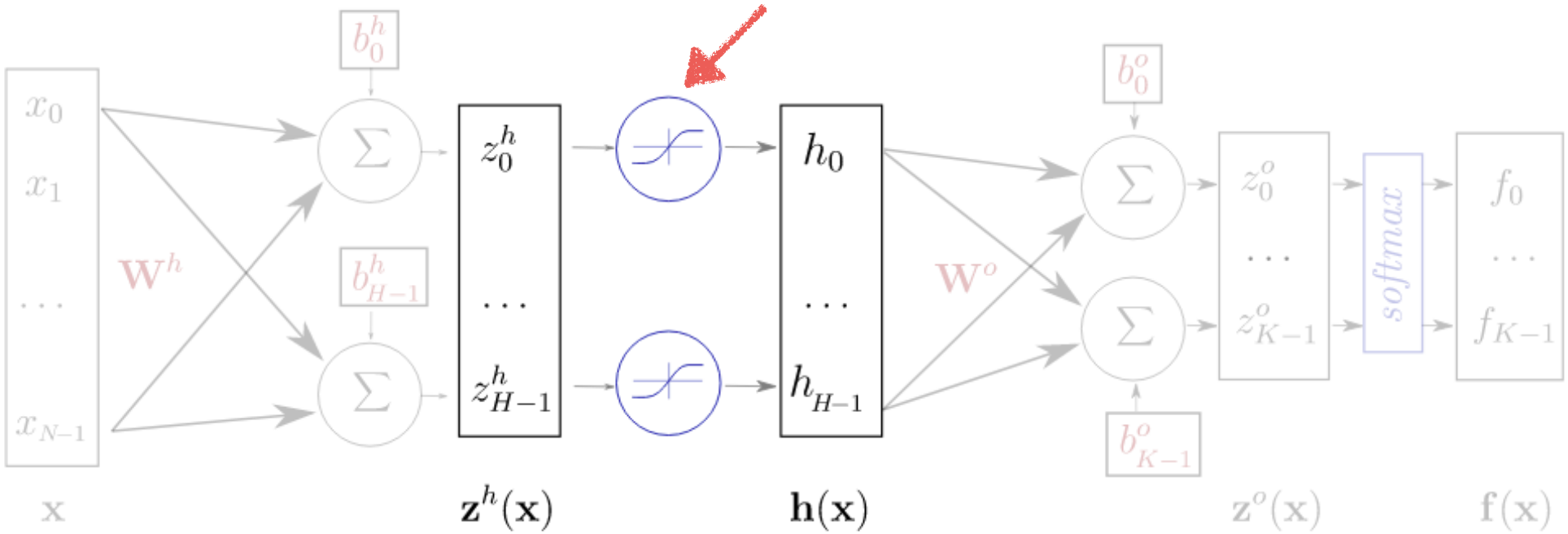


FIRST LAYER

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

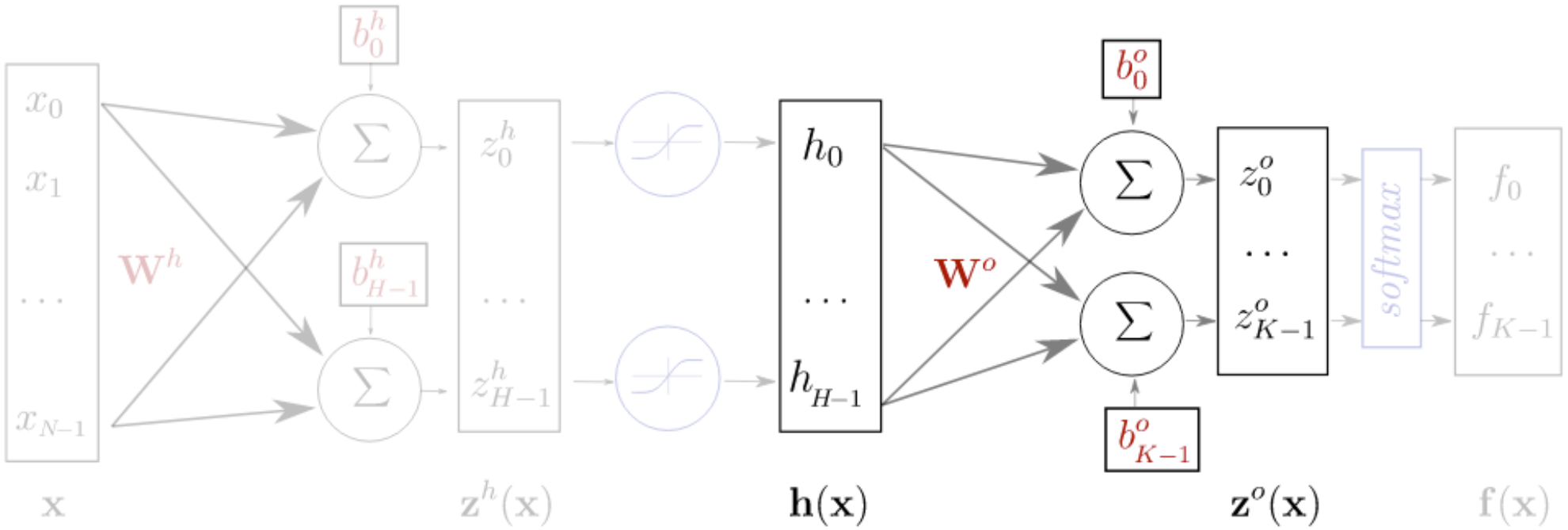
INPUT

ACTIVATION FUNCTION



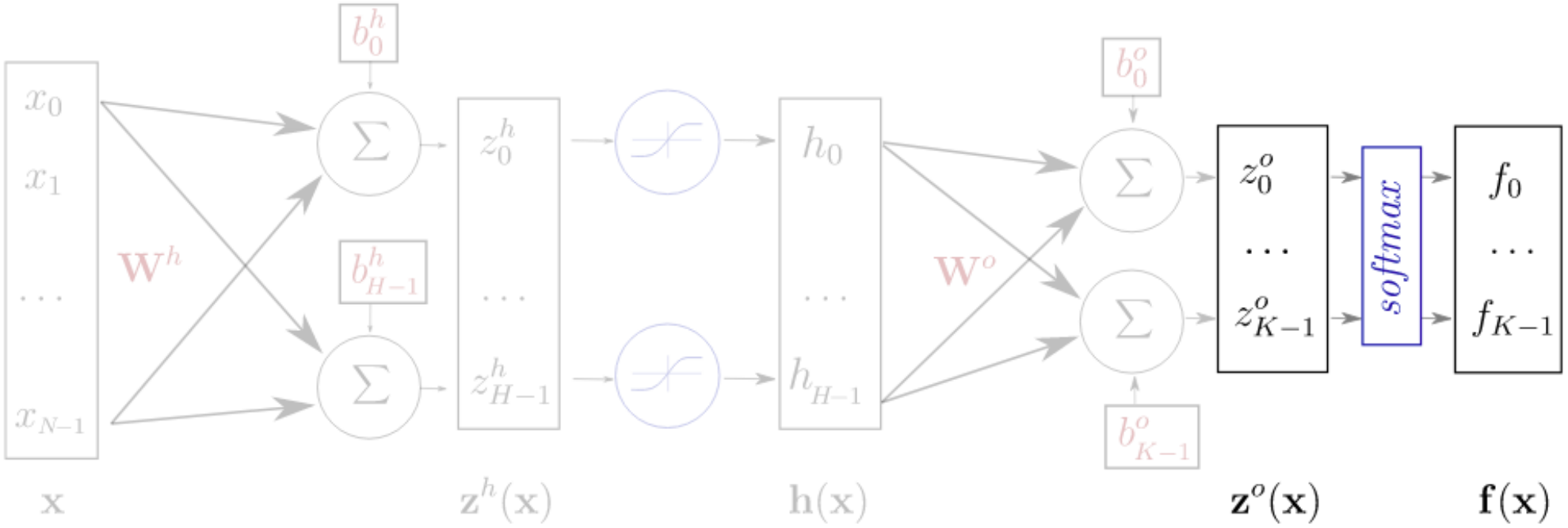
HIDDEN LAYER

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



OUTPUT LAYER

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



PREDICTION LAYER

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

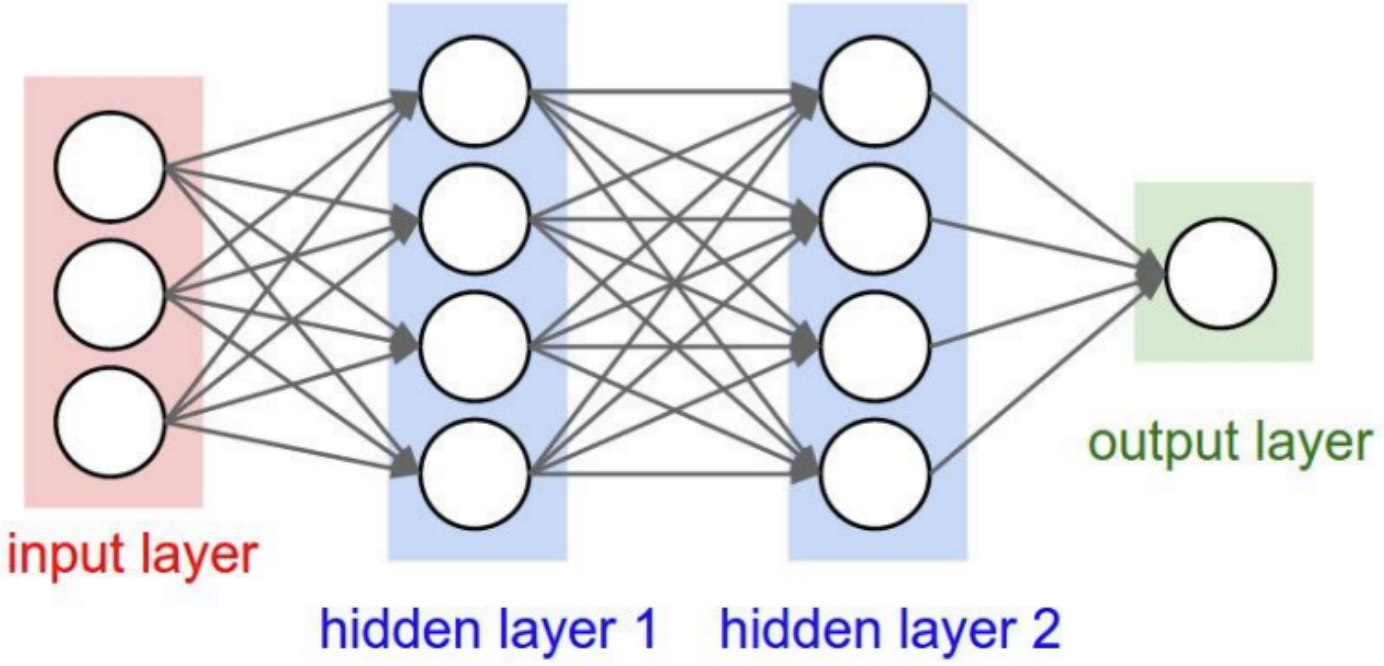
“CLASSICAL” MACHINE LEARNING

$$f_{\vec{W}}(\vec{x}) = \vec{y}$$

LABEL

Q , SF

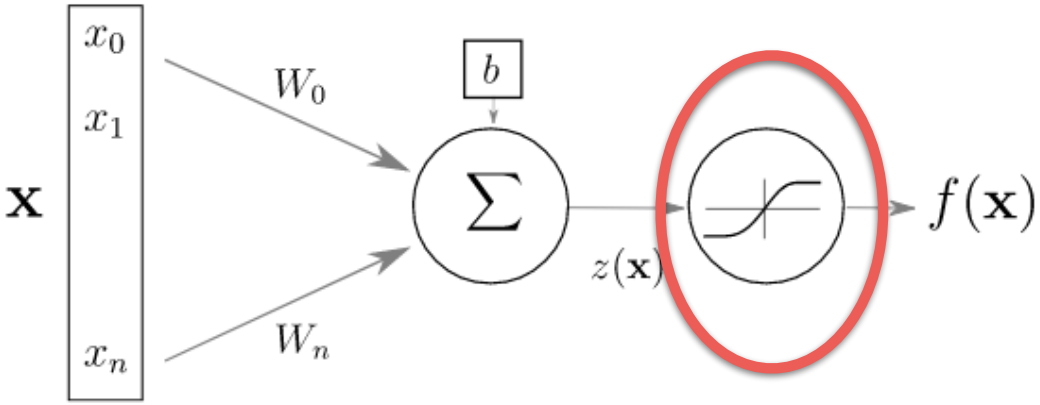
REPLACE THIS BY A GENERAL NON LINEAR FUNCTION WITH SOME PARAMETERS W



$$p = g_3(W_3 g_2(W_2 g_1(W_1 \vec{x}_0)))$$

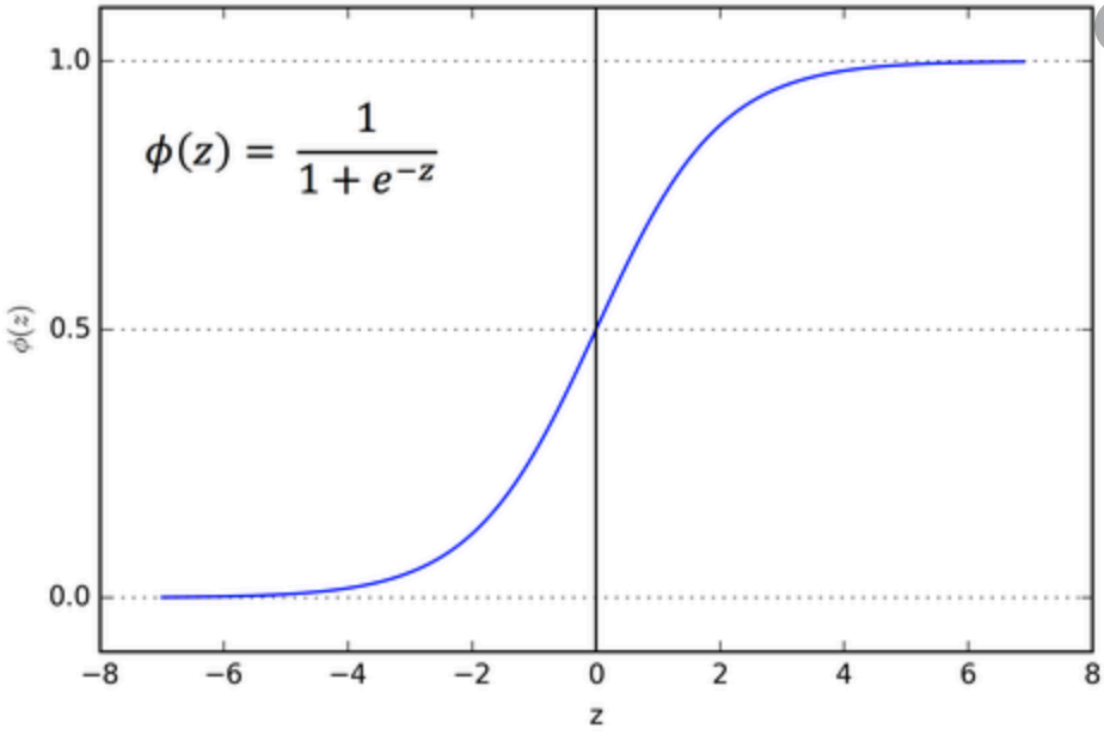
NETWORK FUNCTION

ACTIVATION FUNCTIONS?

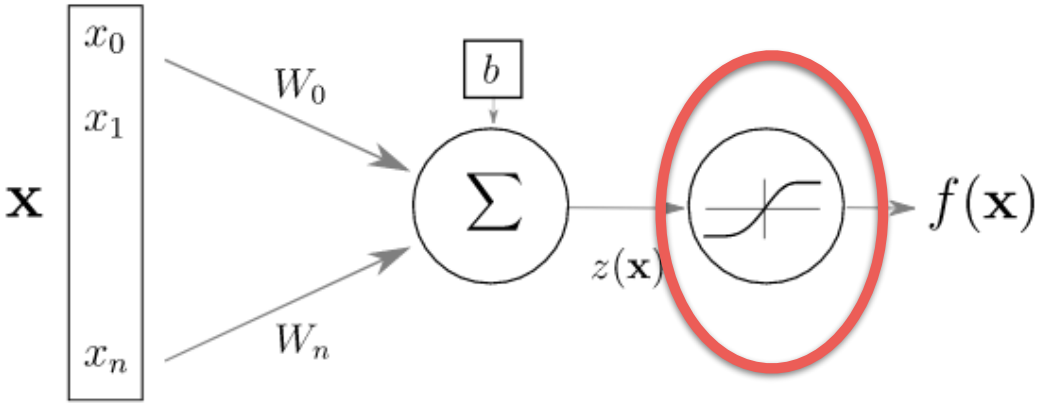


ADD NON LINEARITIES TO THE PROCESS

ACTIVATION FUNCTIONS?

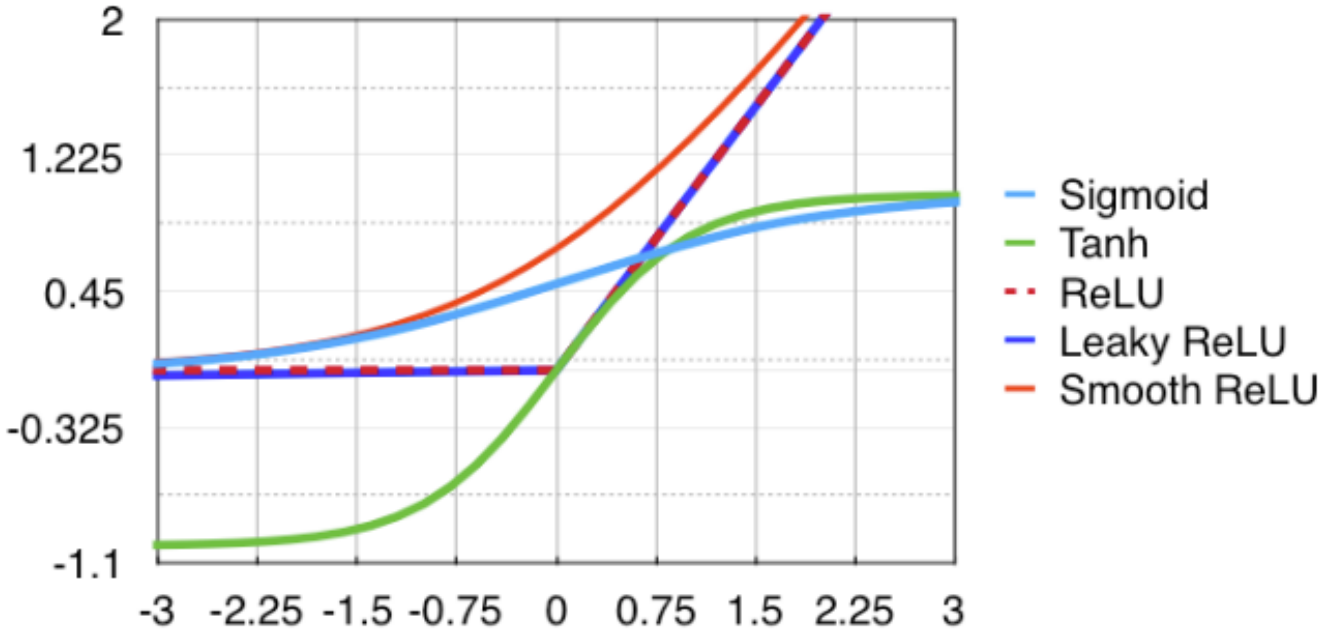


the sigmoid function



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

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:

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

NUMBER OF TRAINING EXAMPLES

GROUND TRUTH

NETWORK OUTPUT (SIGMOID)

GROUND TRUTH

NETWORK OUTPUT (SIGMOID)

BINARY CLASSIFICATION LOSS

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

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

NUMBER OF TRAINING EXAMPLES

GROUND TRUTH

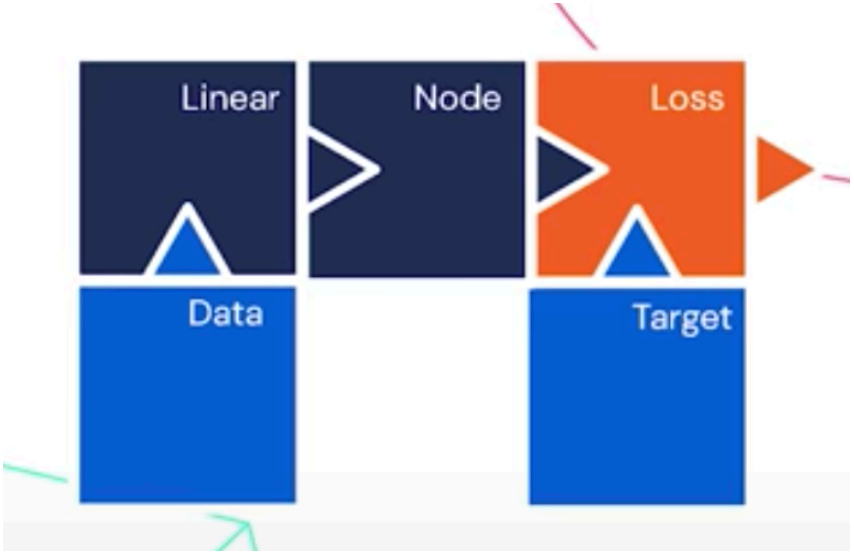
NETWORK OUTPUT (SIGMOID)

GROUND TRUTH

NETWORK OUTPUT (SIGMOID)

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!

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

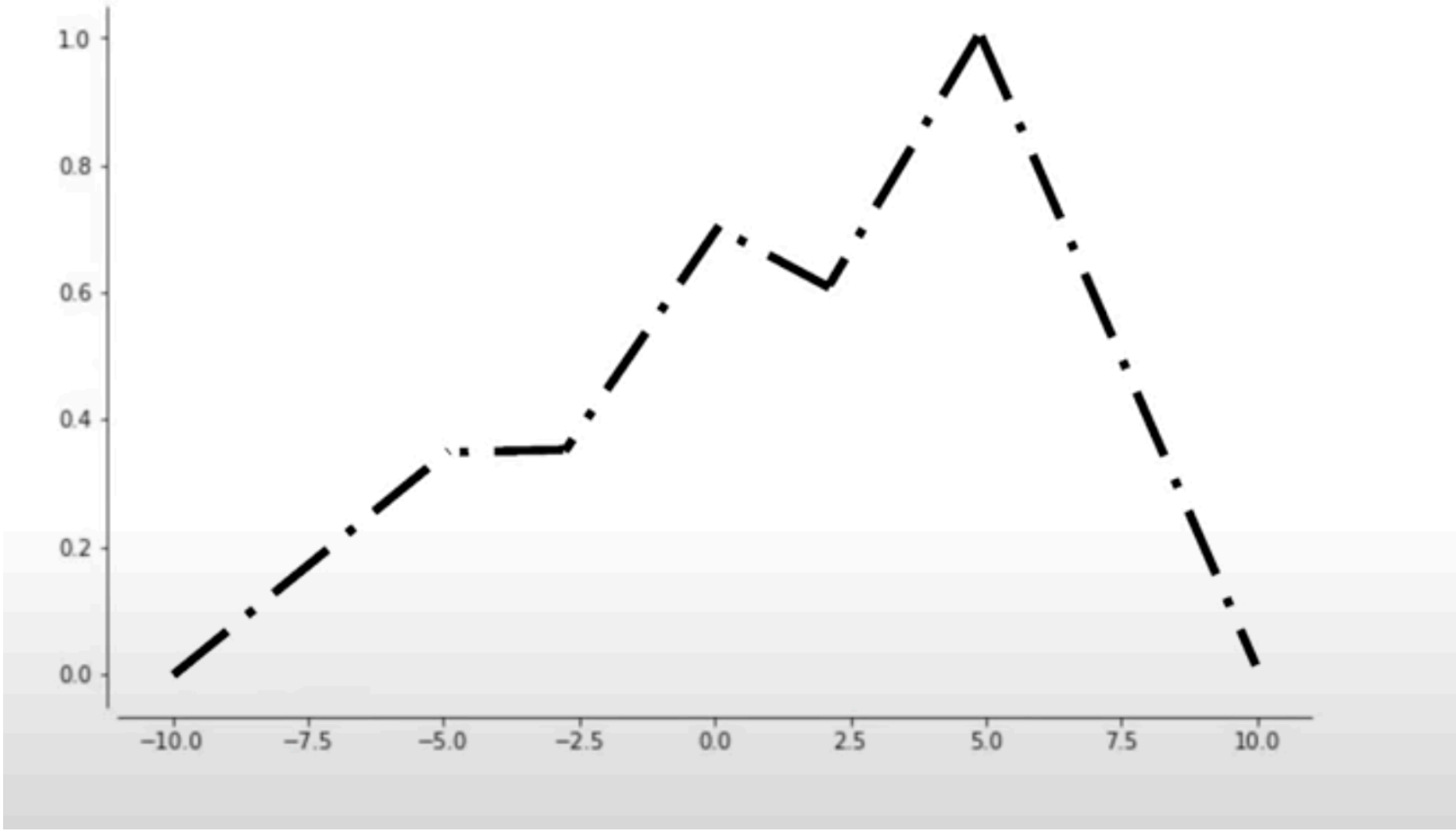
"BIG ENOUGH NETWORK CAN APPROXIMATE, BUT NOT REPRESENT ANY SMOOTH FUNCTION. THE MATH DEMONSTRATION IMPLIES SHOWING THAT NETWORS ARE DENSE IN THE SPACE OF TARGET FUNCTIONS"

UNIVERSAL APPROXIMATION THEOREM

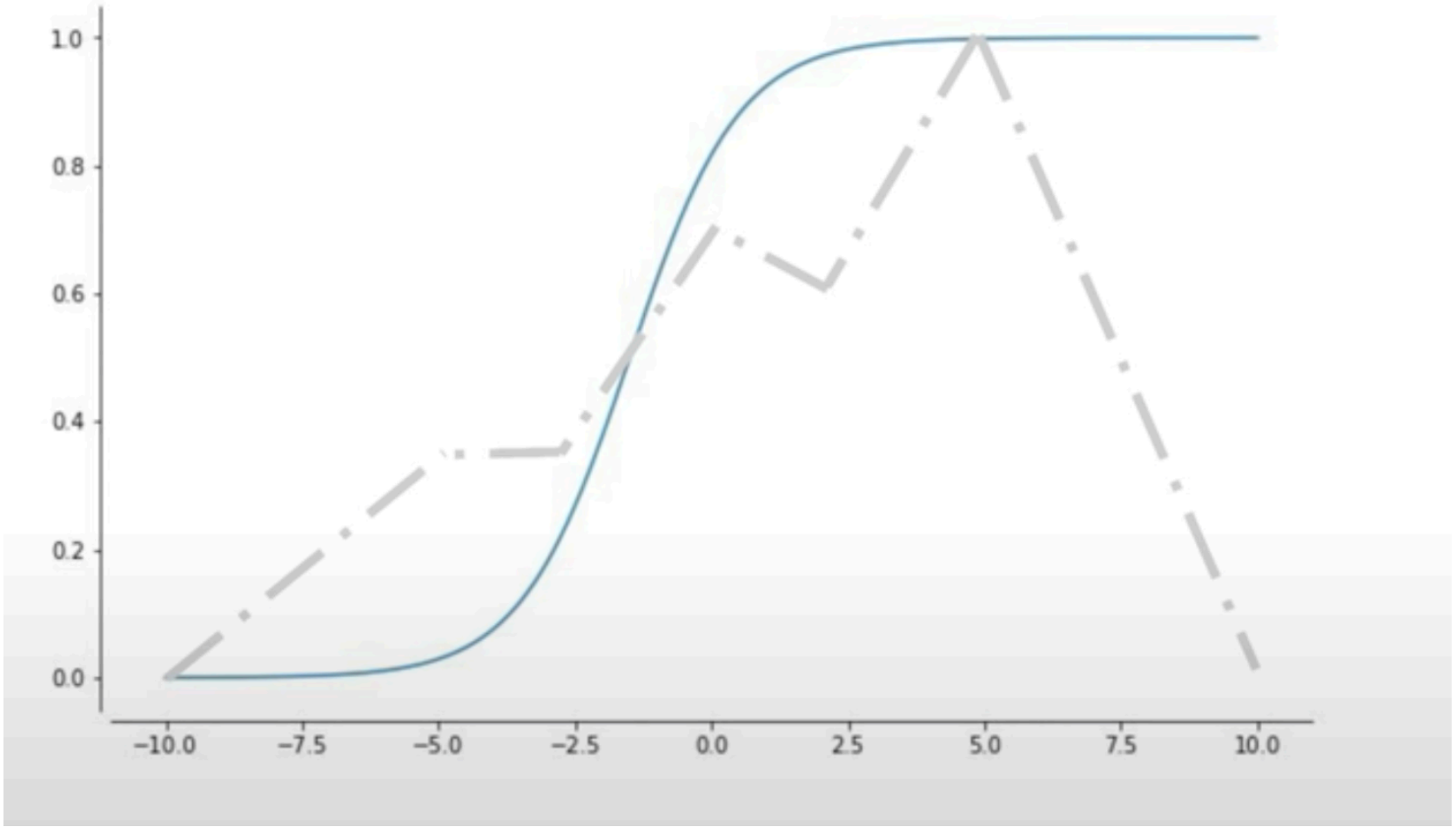
FOR ANY CONTINUOS FUNCTION FOR A HYPERCUBE [0,1]d TO REAL NUMBERS, NON-CONSTANT, BOUNDED AND CONTINUOUS ACTIVATION FUNCTION f, AND EVERY POSITIVE EPSILON, THERE EXISTS A1-HIDDEN LAYER NEURAL NETWORK USING f THAT OBTAINES AT MOST EPSILON ERROR IN FUNCTIONAL SPACE

Horvik+91

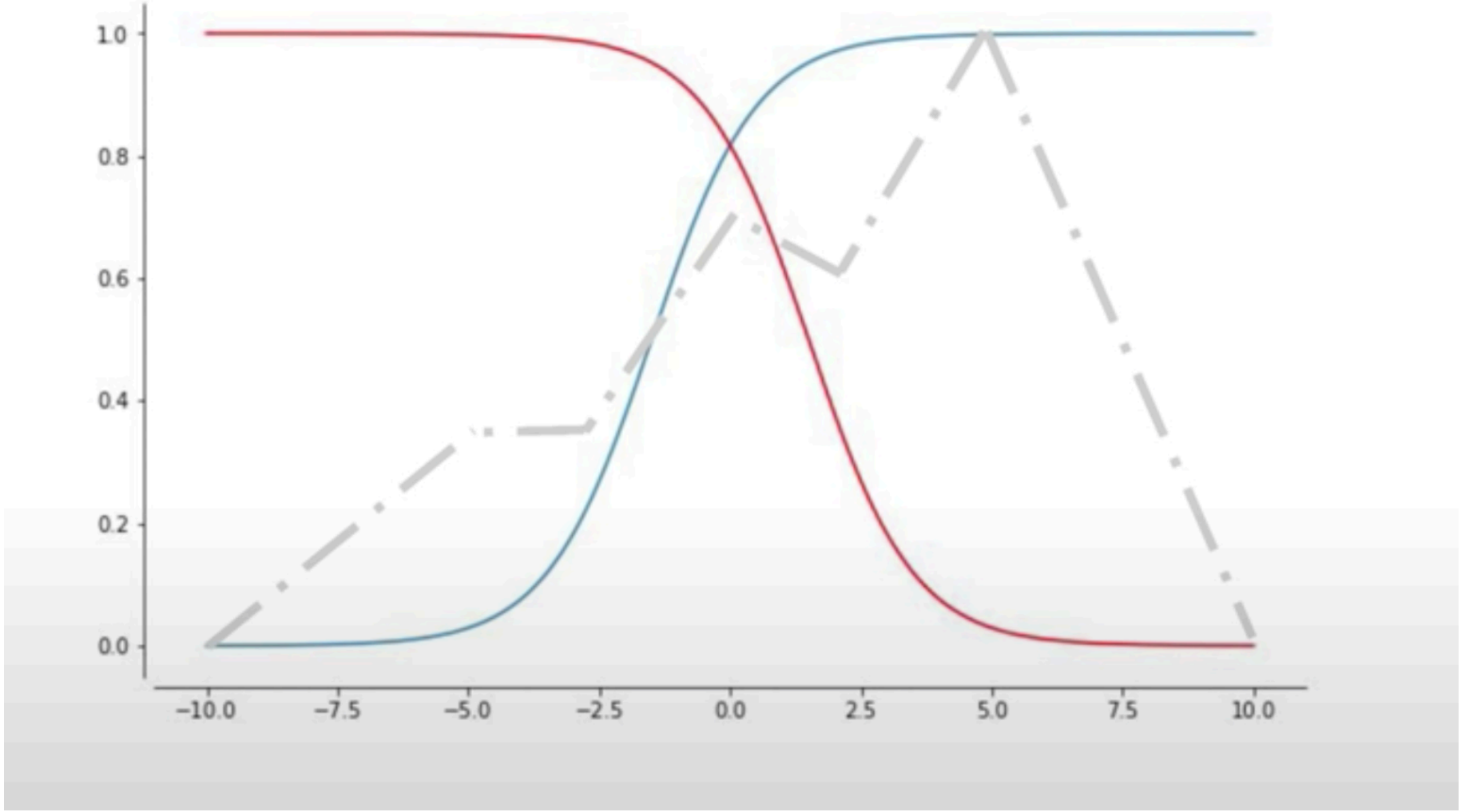
"BIG ENOUGH NETWORK CAN APPROXIMATE, BUT NOT REPRESENT ANY SMOOTH FUNCTION. THE MATH DEMONSTRATION IMPLIES SHOWING THAT NETWORS ARE DENSE IN THE SPACE OF TARGET FUNCTIONS"



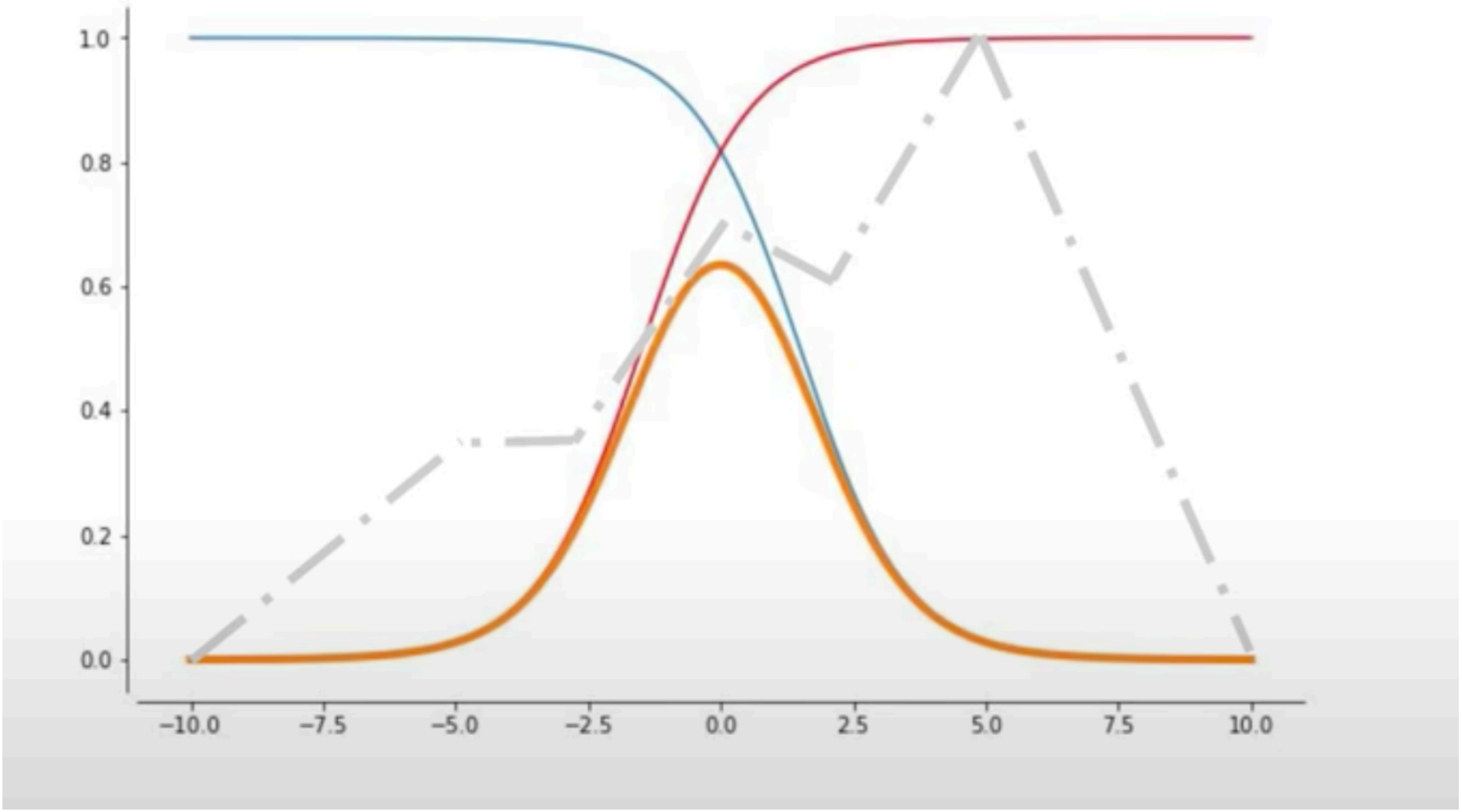
(deepmind @Czarnecki)



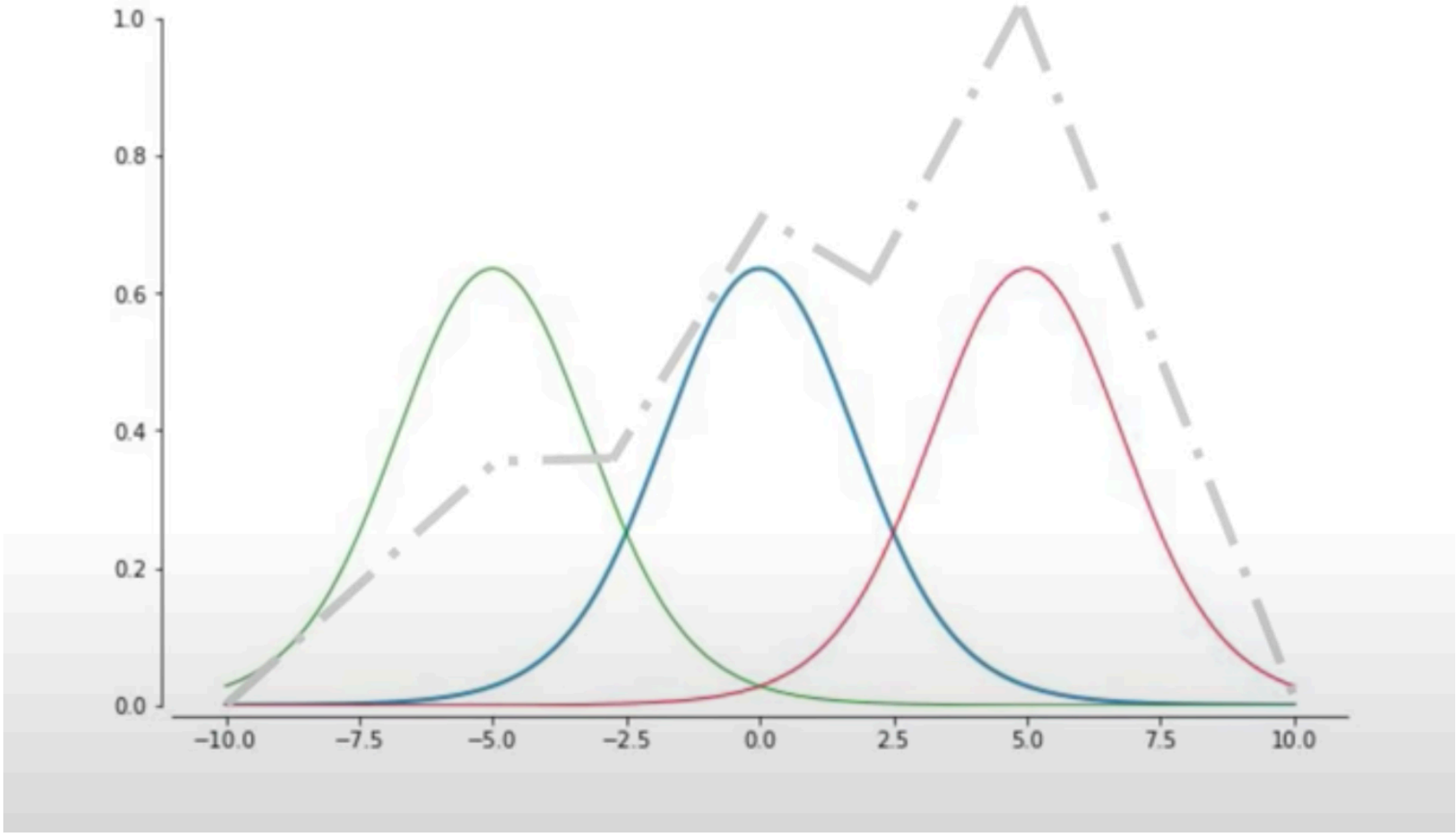
(deepmind @Czarnecki)



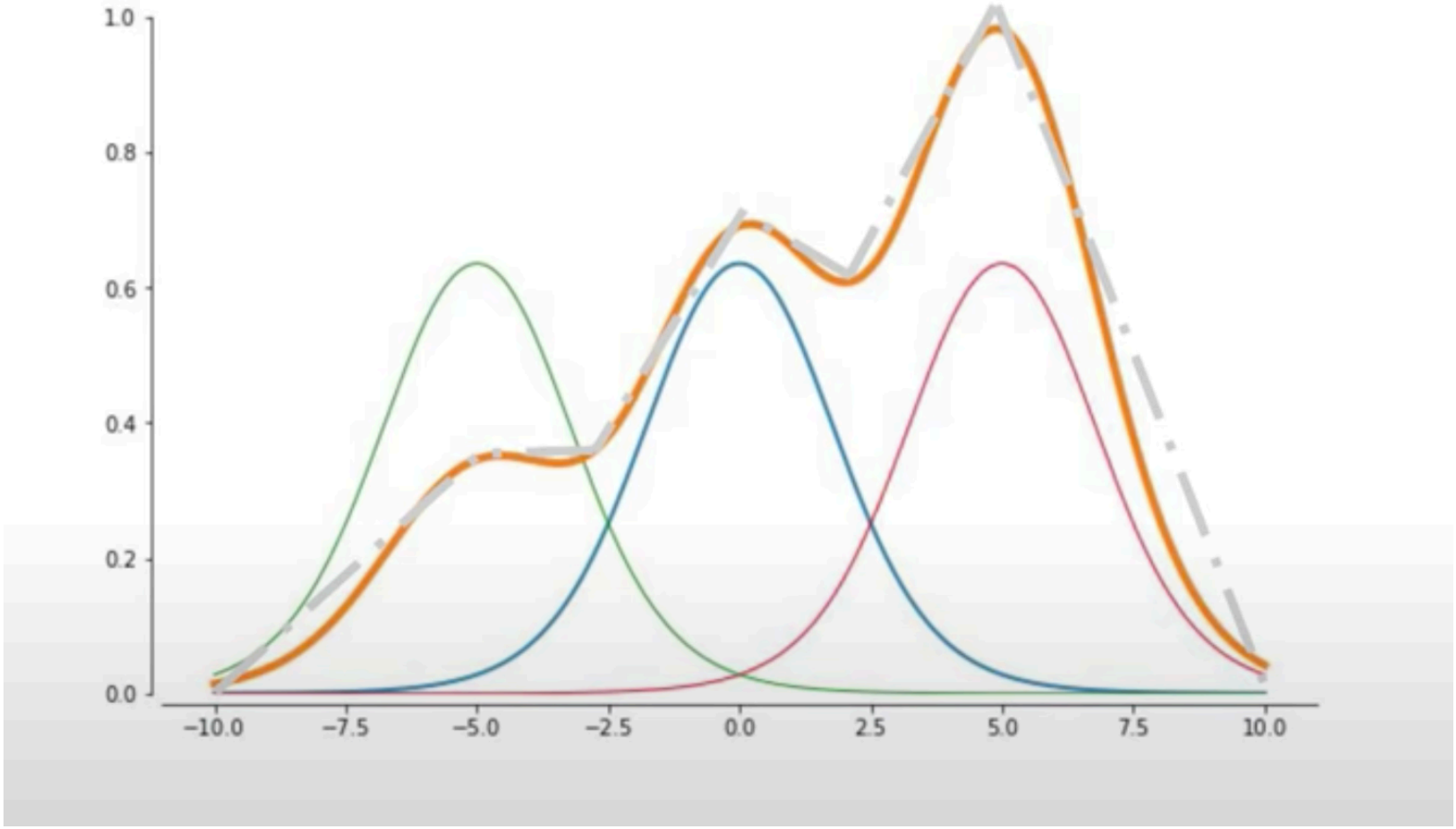
(deepmind
@Czarnecki)



(deepmind
@Czarnecki)

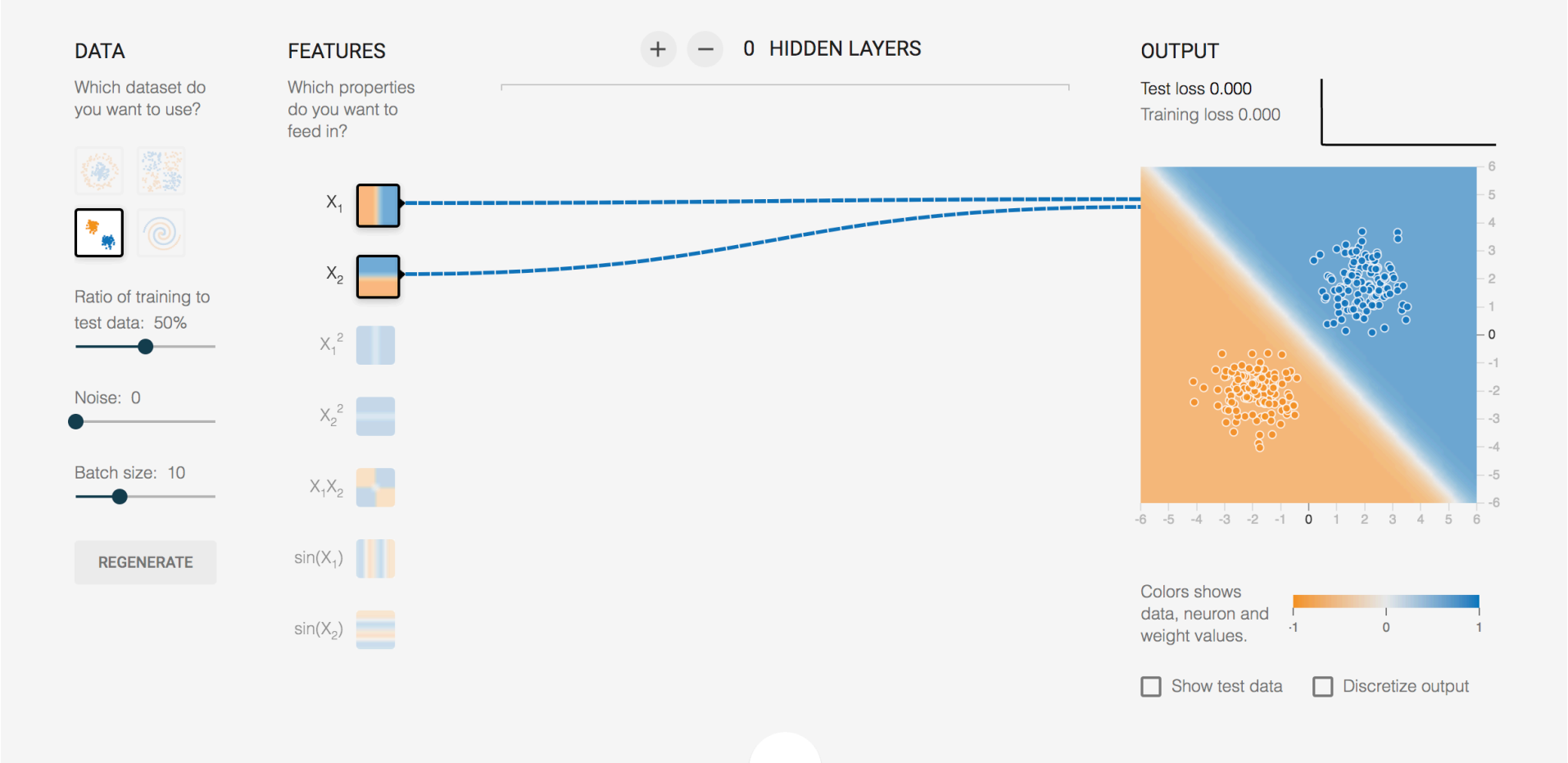


(deepmind
@Czarnecki)

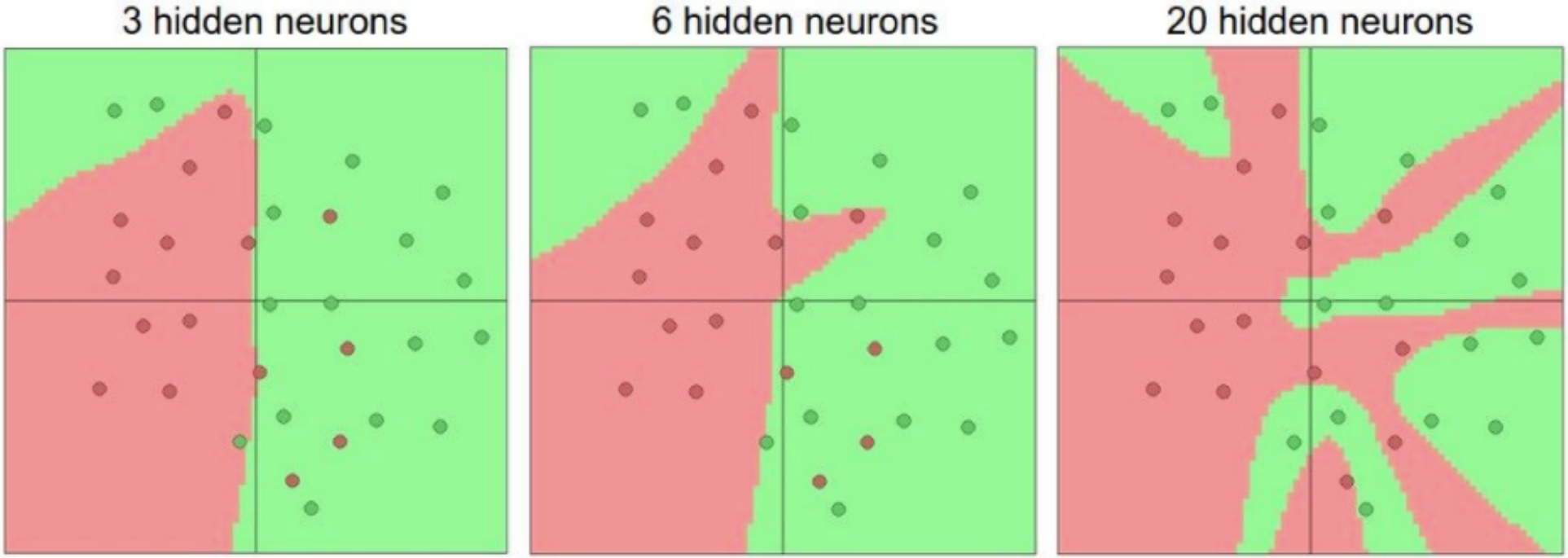


(deepmind
@Czarnecki)

https://
playground.tensorflow.org/



HIDDEN LAYERS ALLOW INCREASING COMPLEXITY



More complex functions allow increasing complexity

Credit: Karpathy

SO LET’S GO DEEPER AND DEEPER!

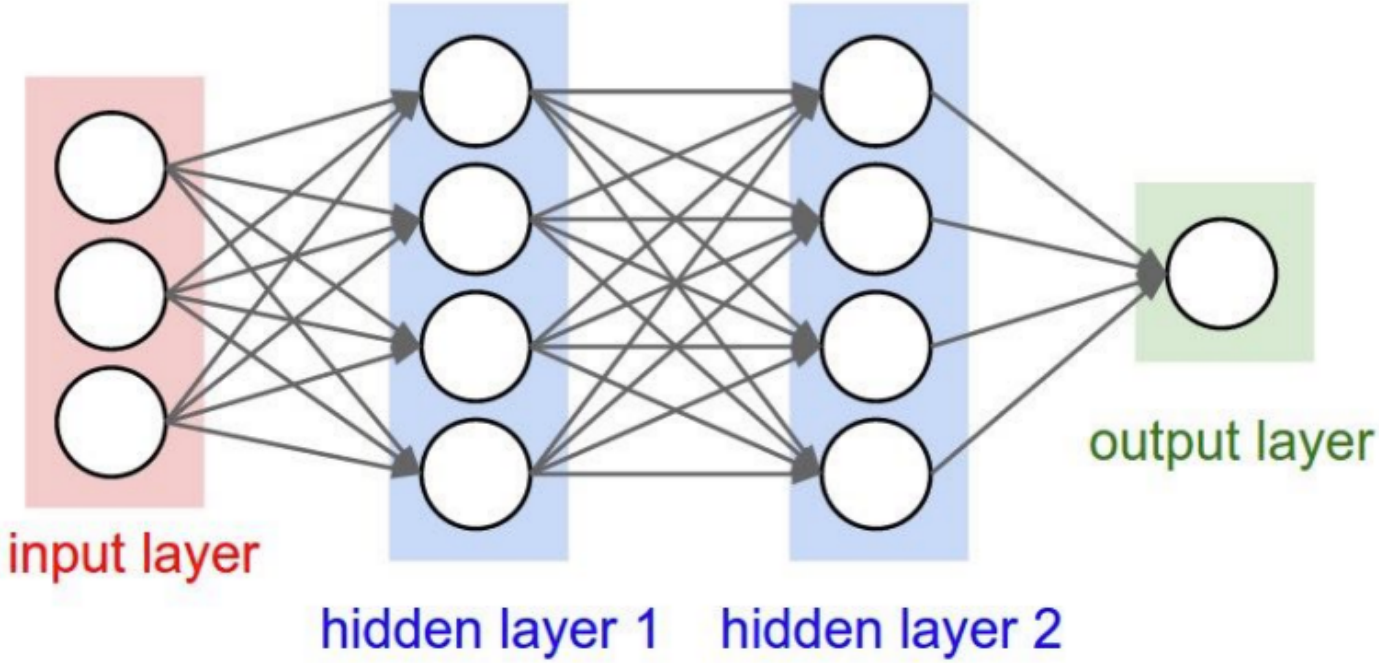
YES BUT…

NOT SO STRAIGHTFORWARD, DEEPER MEANS MORE WEIGHTS … **DO WE KNOW HOW TO MINIMIZE THIS?**

OK, SO NOW LET'S FIND THE WEIGHTS

OPTIMIZATION

[OR HOW TO FIND THE WEIGHTS?]

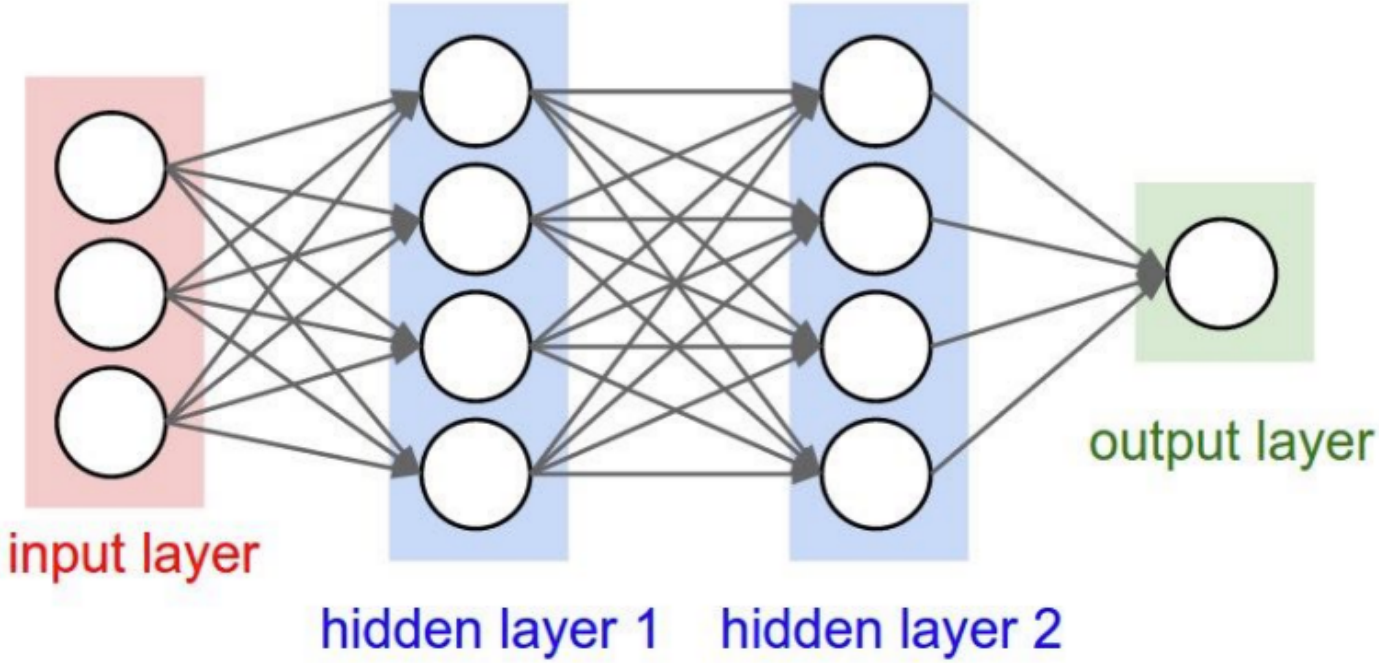


$$p = g_3(W_3 g_2(W_2 g_1(W_1 \vec{x_0})))$$

NETWORK FUNCTION

OPTIMIZATION

[OR HOW TO FIND THE WEIGHTS?]



$$p = g_3(W_3 g_2(W_2 g_1(W_1 \vec{x_0})))$$

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

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

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

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….

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

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…

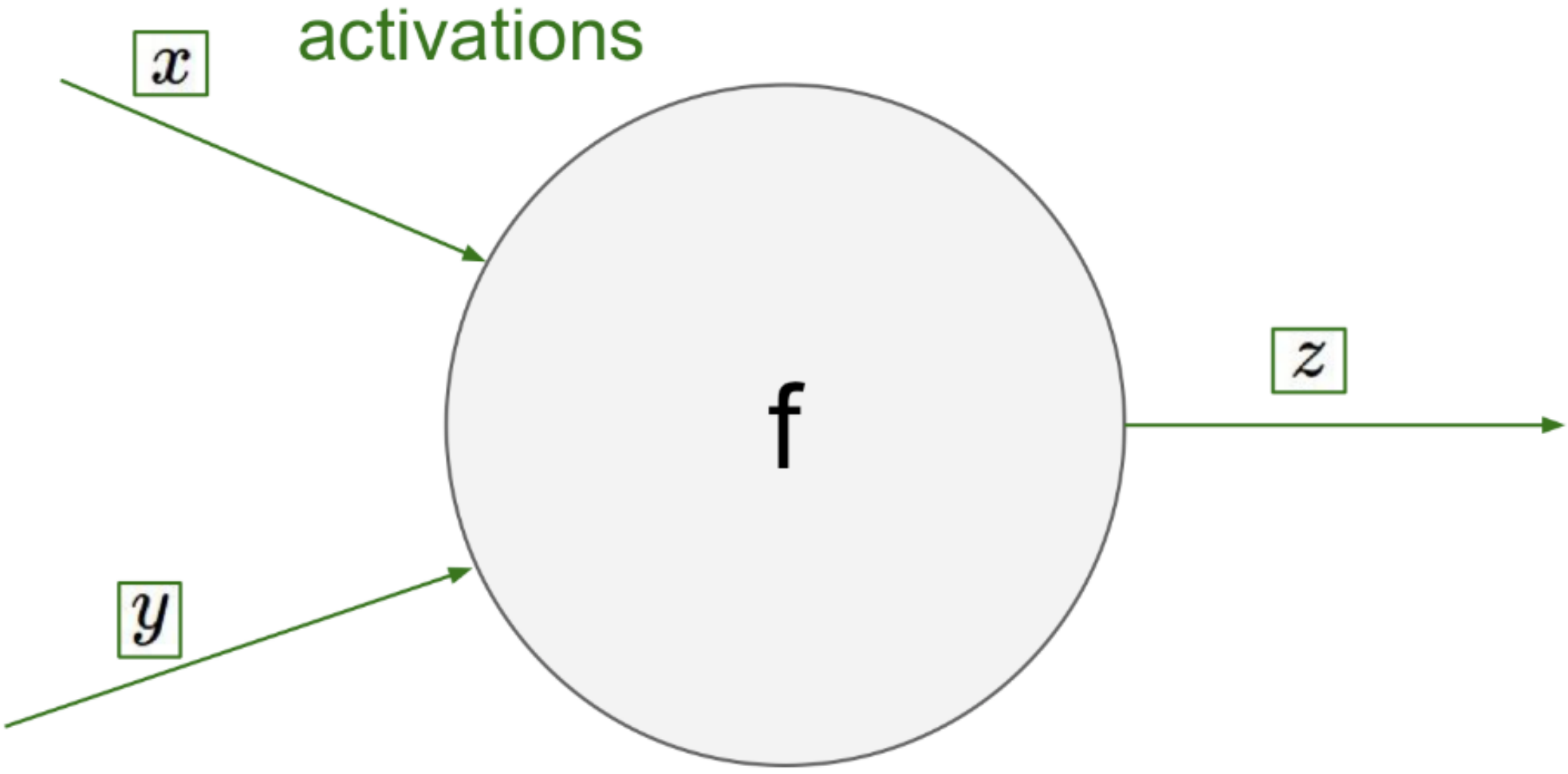
BUT NEEDS THE HESSIAN

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

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

BACKPROPAGATION

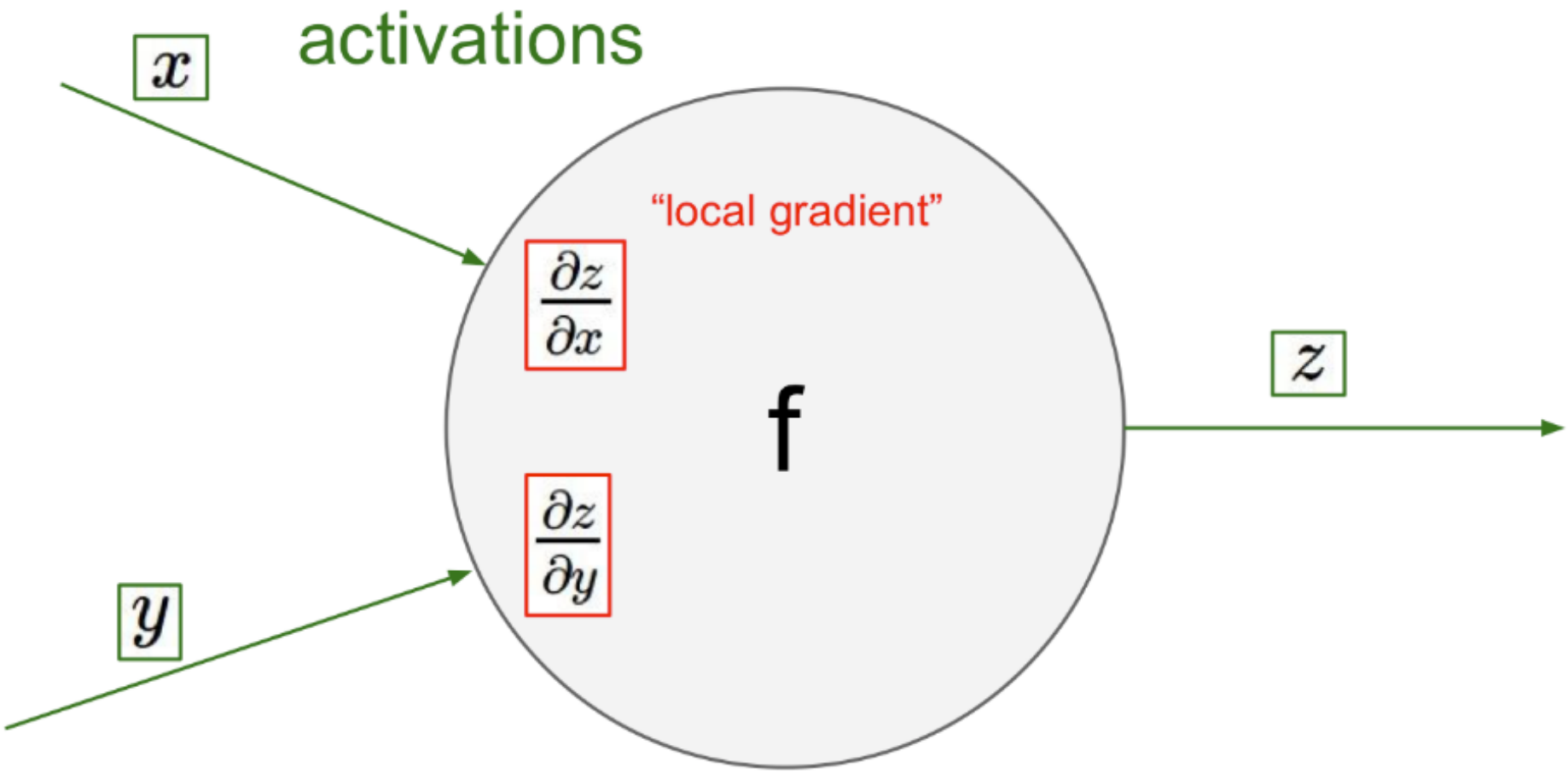
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

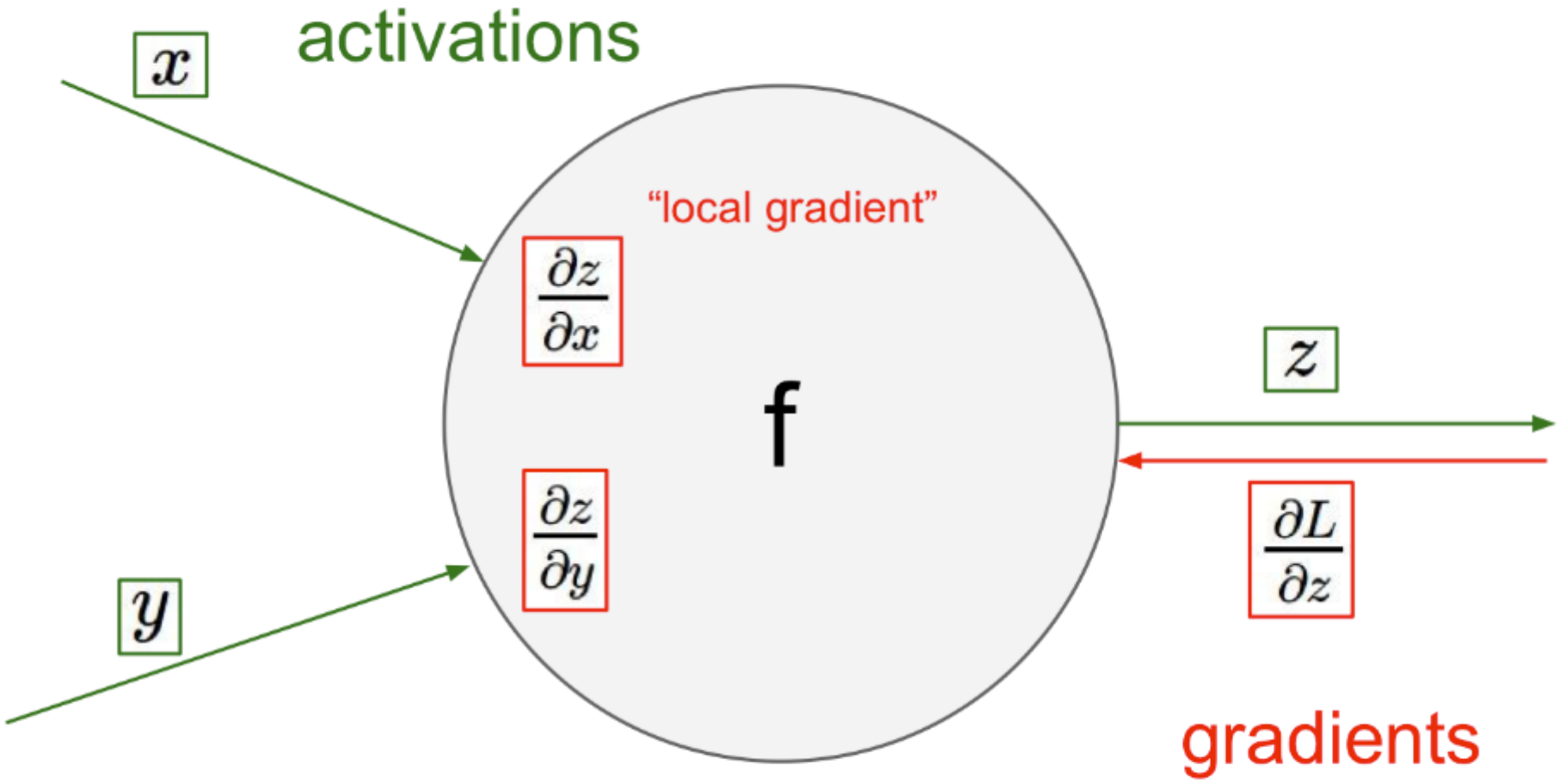
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

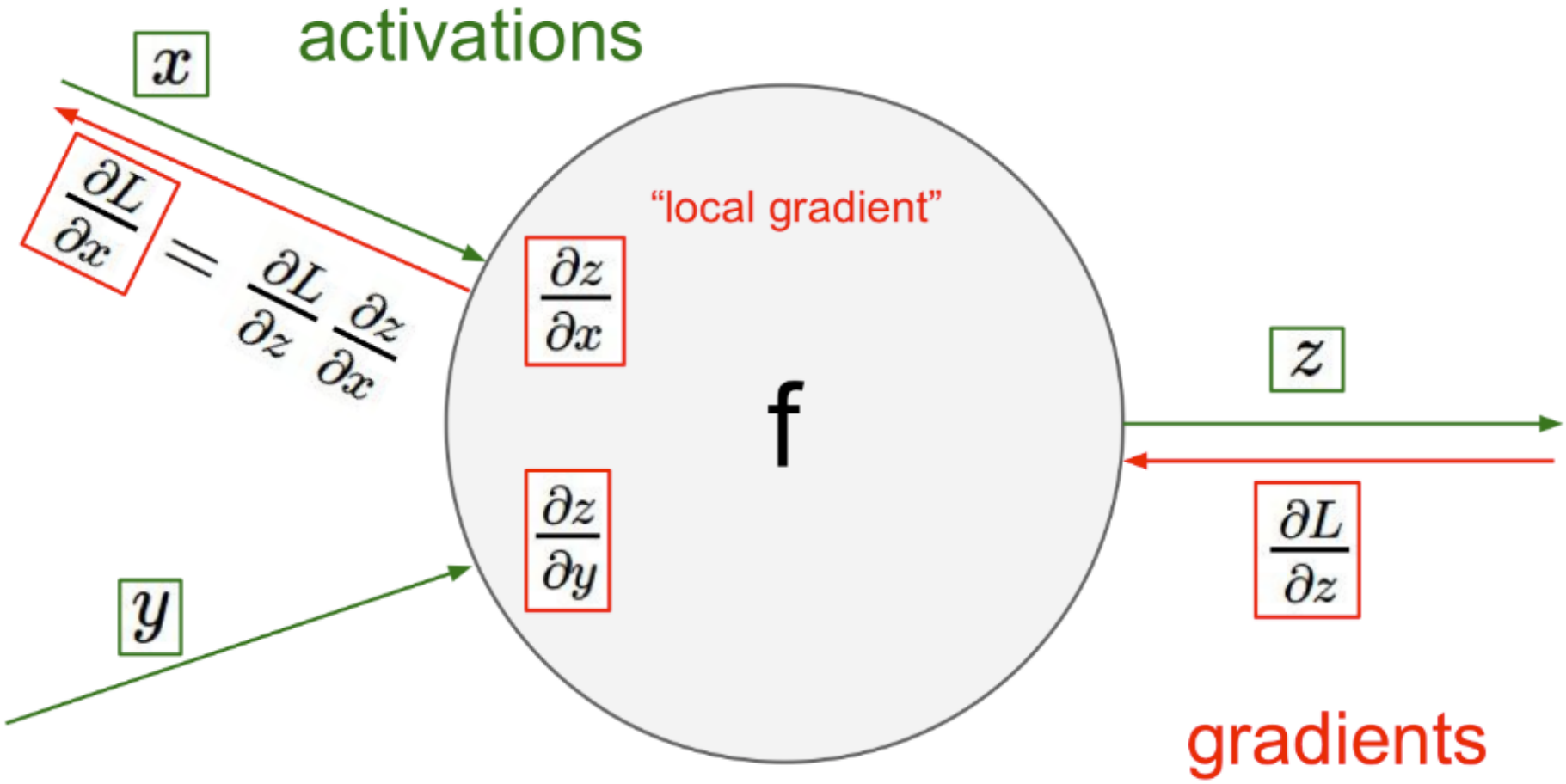
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

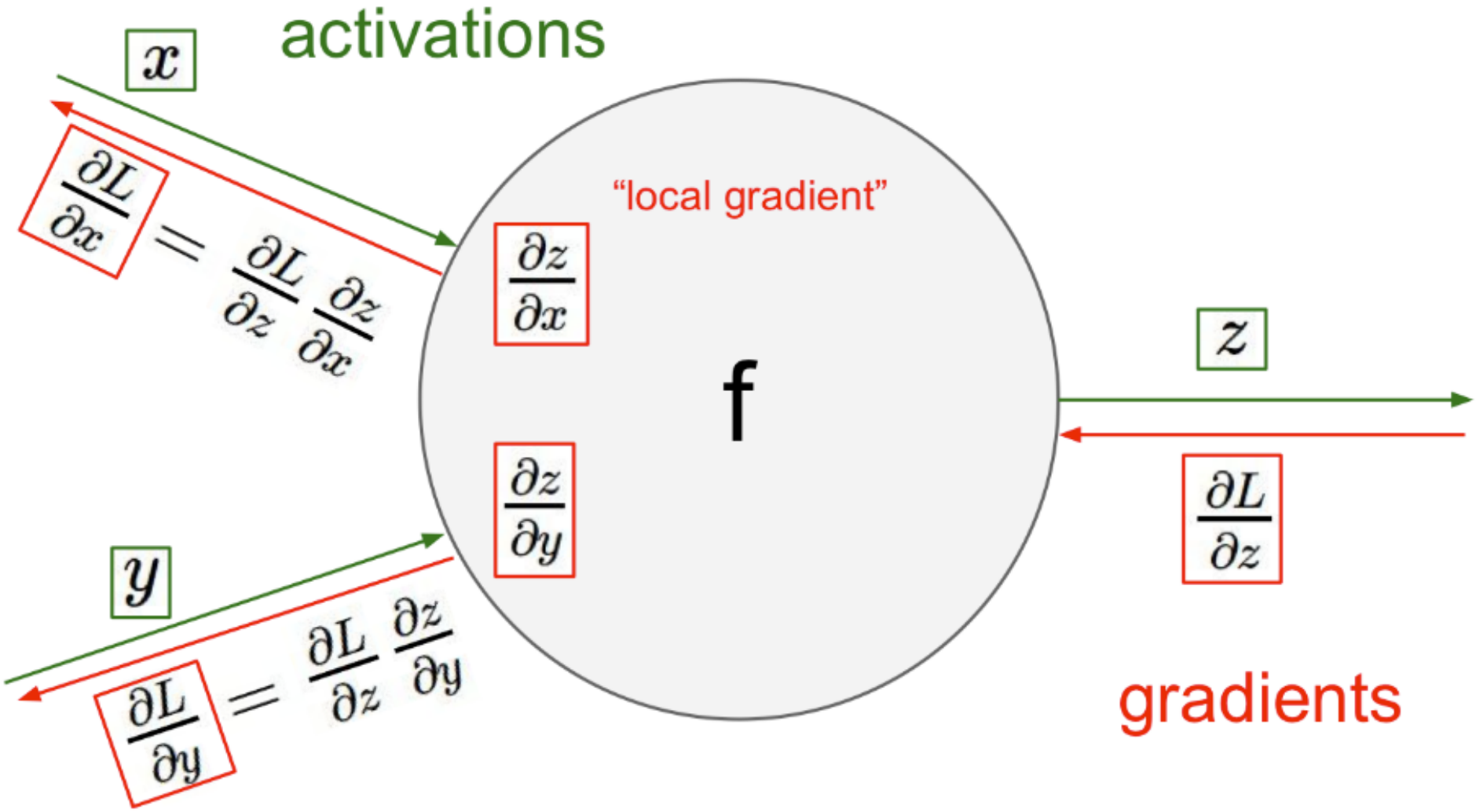
[AT THE NEURON LEVEL]



Credit: A. Karpathy

BACKPROPAGATION

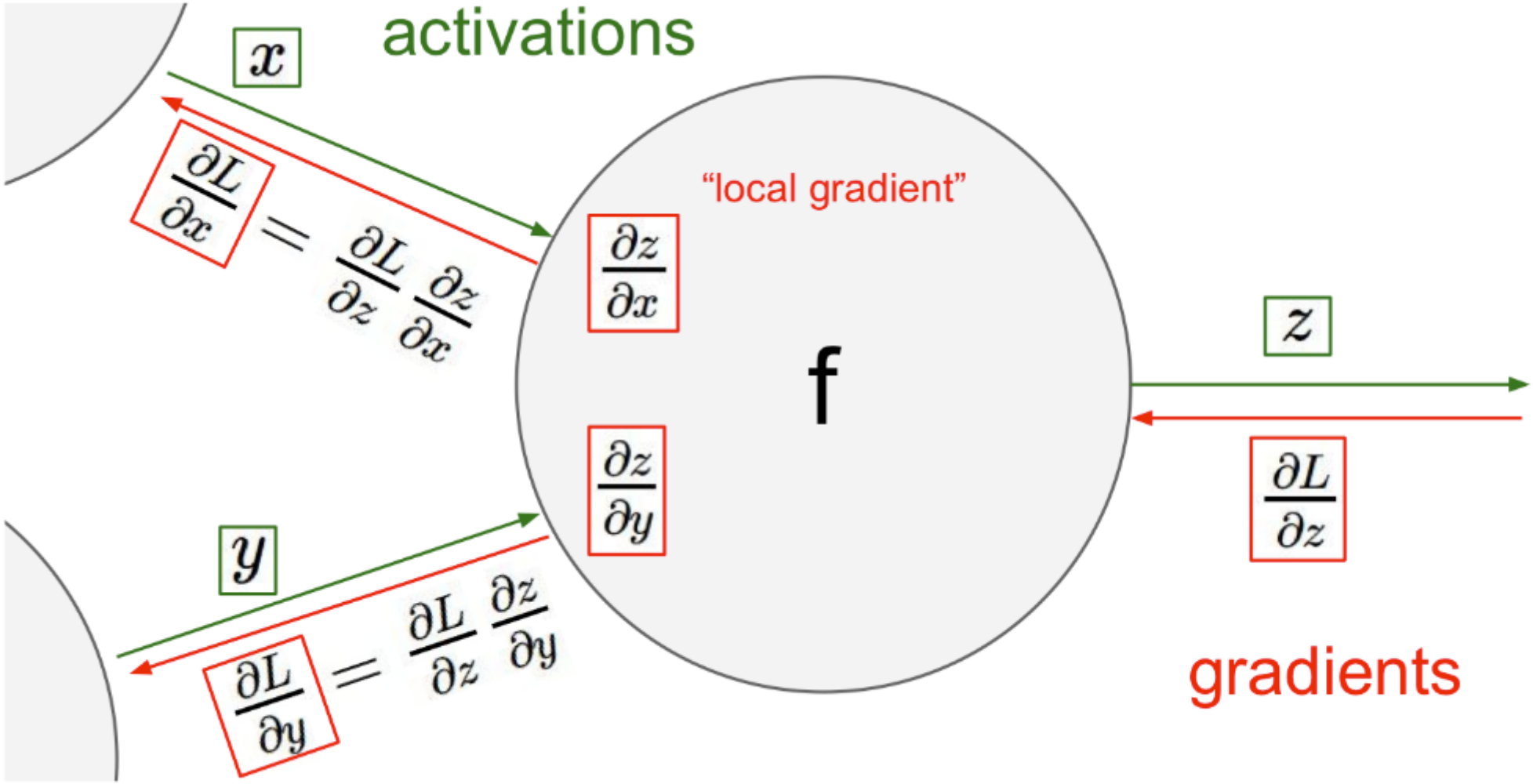
[AT THE NEURON LEVEL]



Credit: A. Karpathy

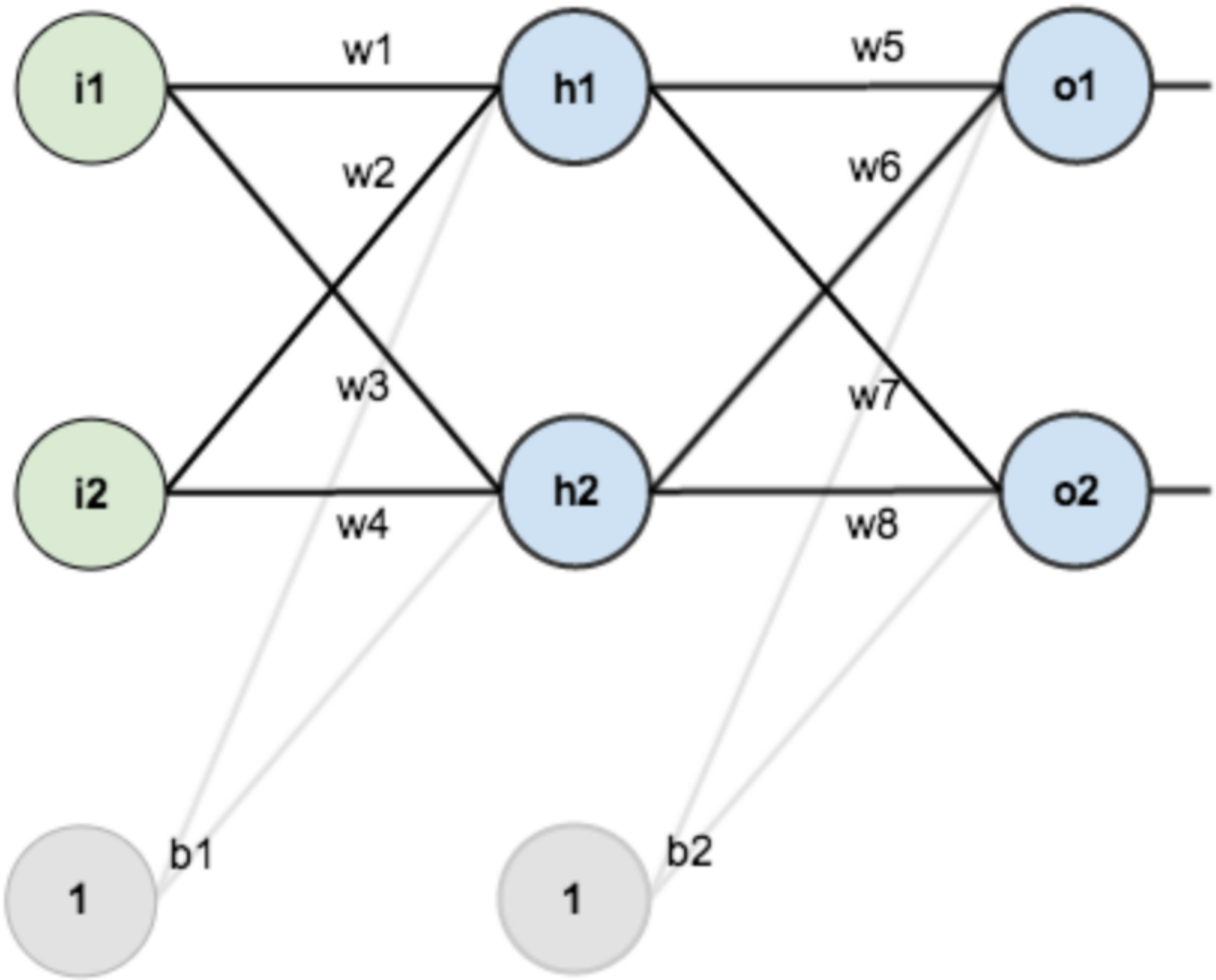
BACKPROPAGATION

[AT THE NEURON LEVEL]

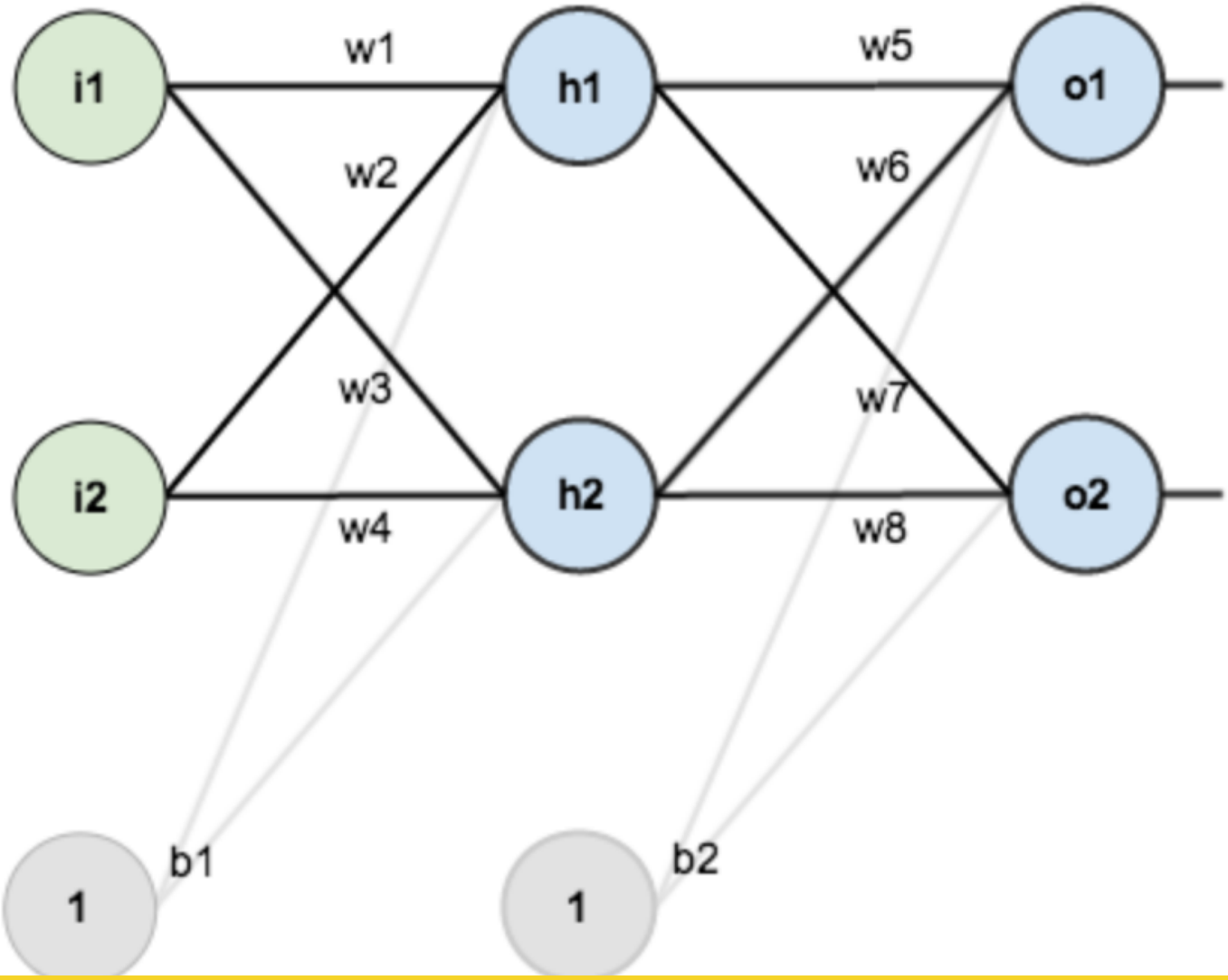


Credit: A. Karpathy

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

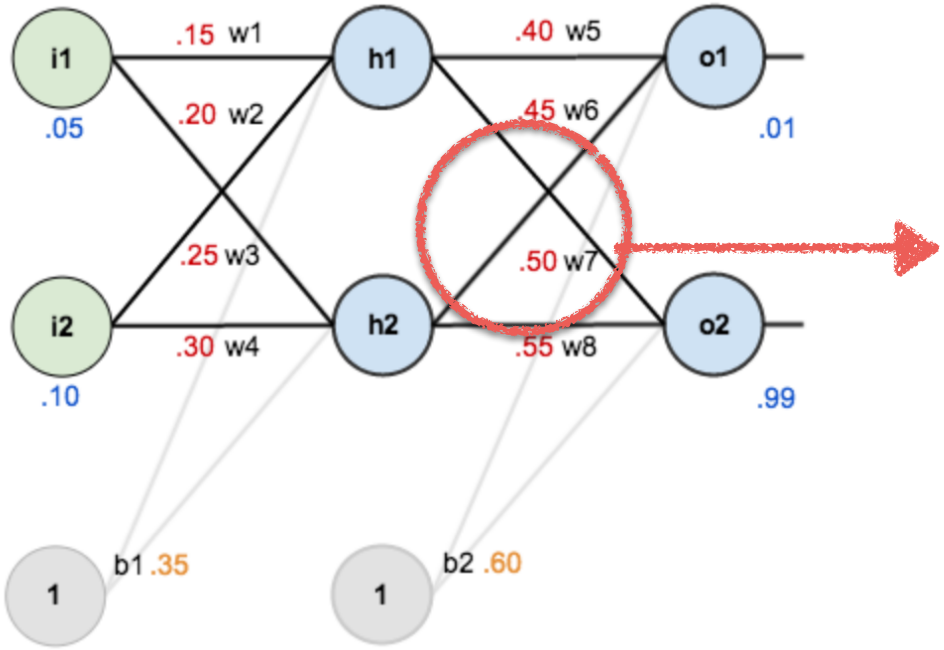


EXAMPLE TAKEN FROM HERE



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

EXAMPLE TAKEN FROM HERE



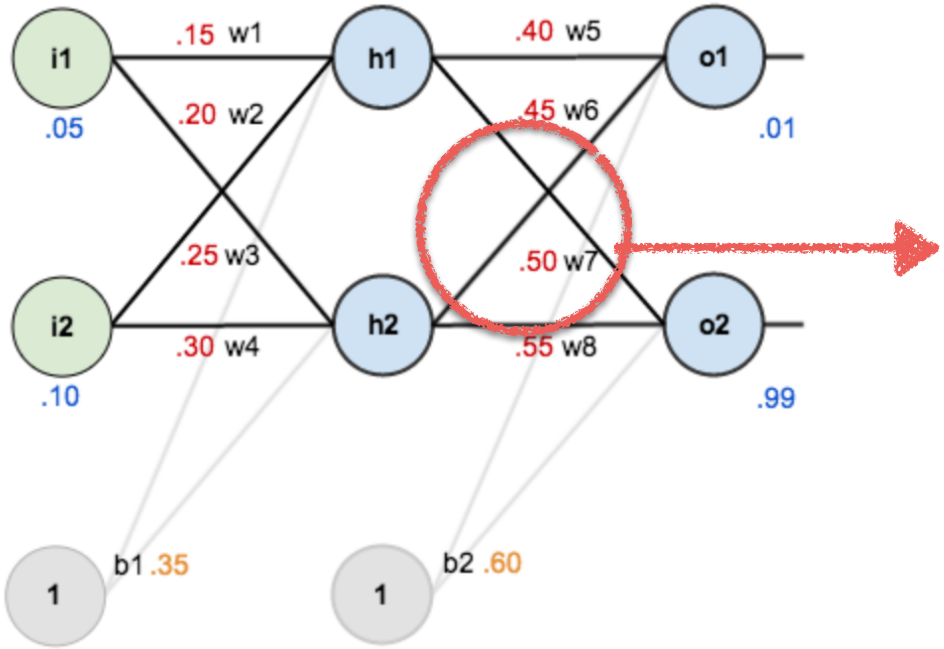
initial weights

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]



initial weights

1. THE FORWARD PASS

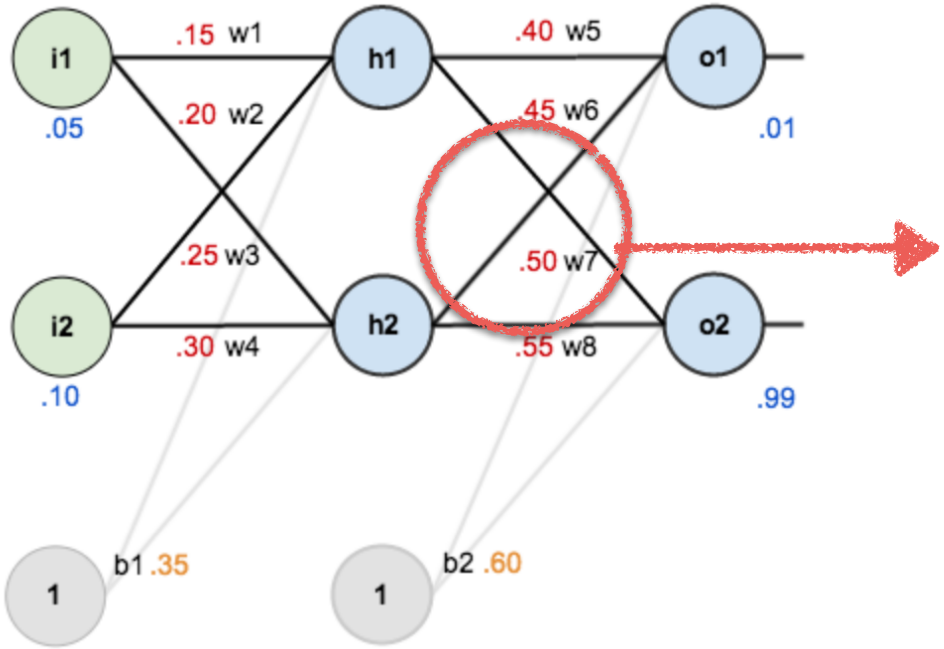
$$in_{h1} = w_1 i_1 + w_2 i_2 + b_1$$

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

[with initial weights]

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

[after the activation function]



initial weights

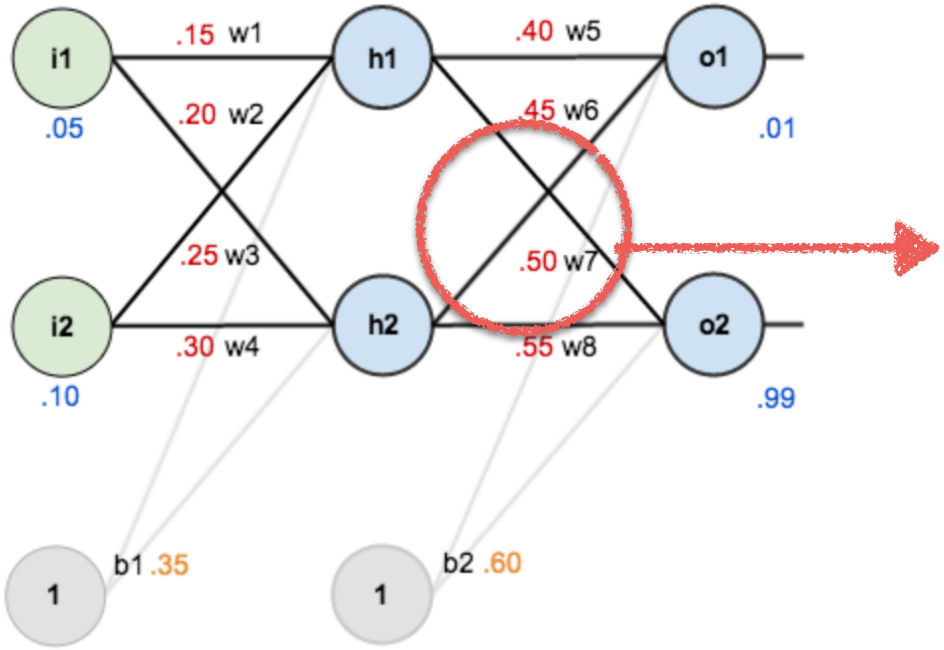
1. THE FORWARD PASS

WE CONTINUE TO o1

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

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

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

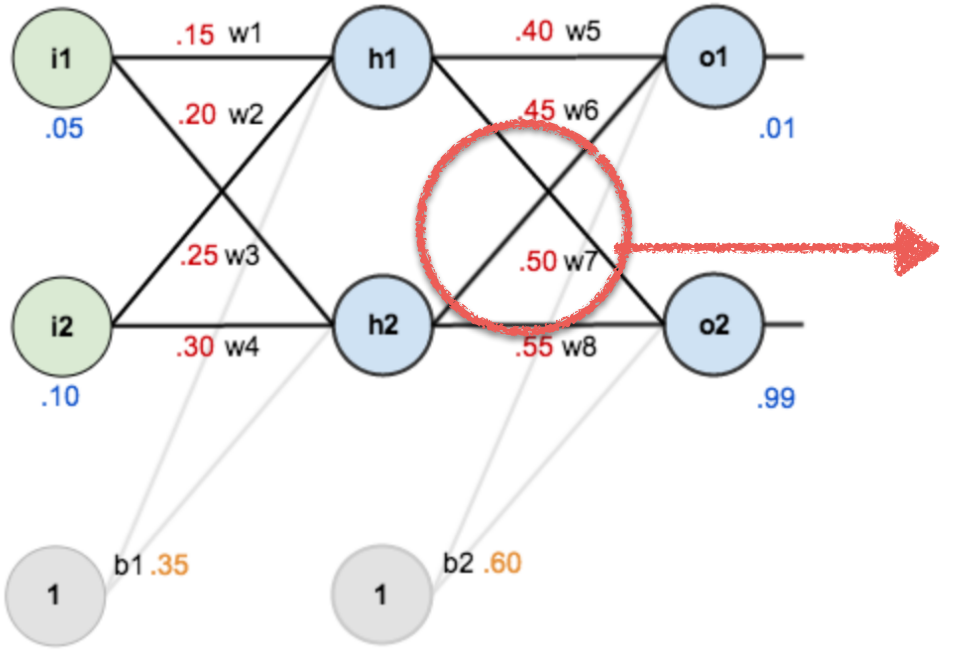


initial weights

1. THE FORWARD PASS

AND THE SAME FOR o2

$$out_{o2} = 0.7729$$



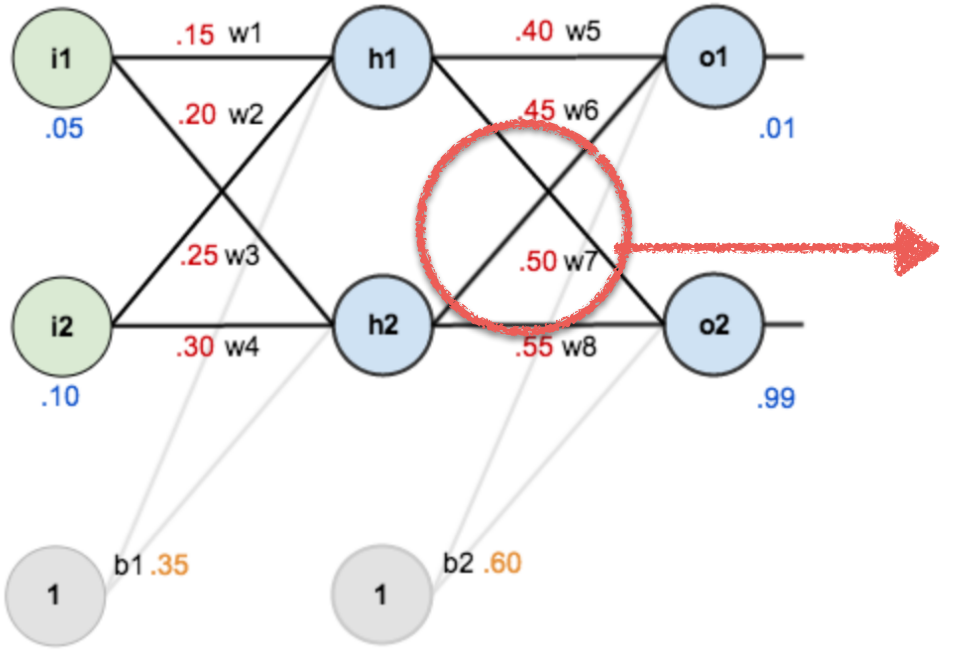
initial weights

2. THE LOSS FUNCTION

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

$$L_{o1} = 0.5(target_{o1} - output_{01})^2 = 0.5 \times (0.01 - 0.751)^2 = 0.274$$

$$L_{o2} = 0.023$$



initial weights

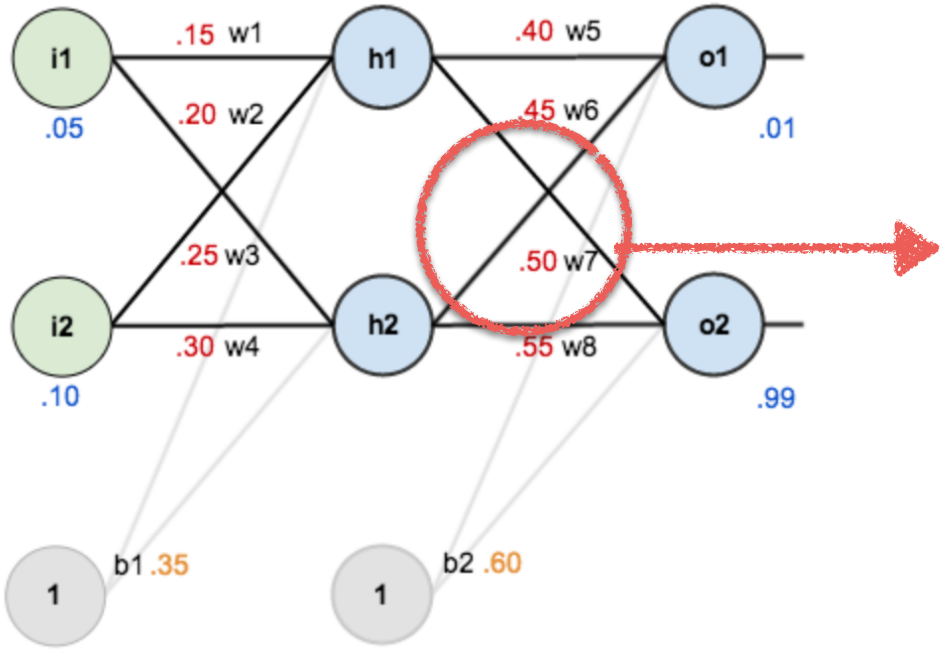
2. THE LOSS FUNCTION

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

$$L_{o1} = 0.5(target_{o1} - output_{01})^2 = 0.5 \times (0.01 - 0.751)^2 = 0.274$$

$$L_{o2} = 0.023$$

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



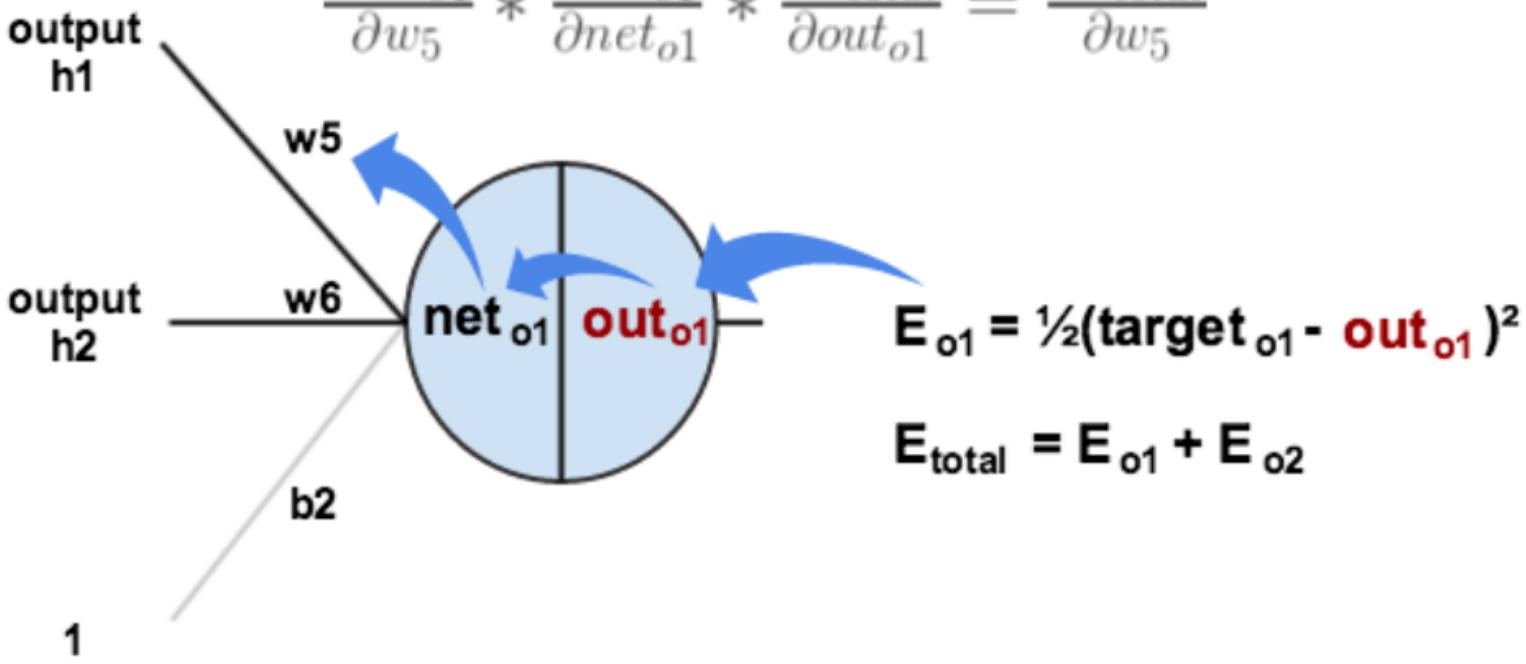
initial weights

3. THE BACKWARD PASS

FOR W5

WE WANT: $\frac{\partial L_{total}}{\partial w_5}$ [gradient of loss function]

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



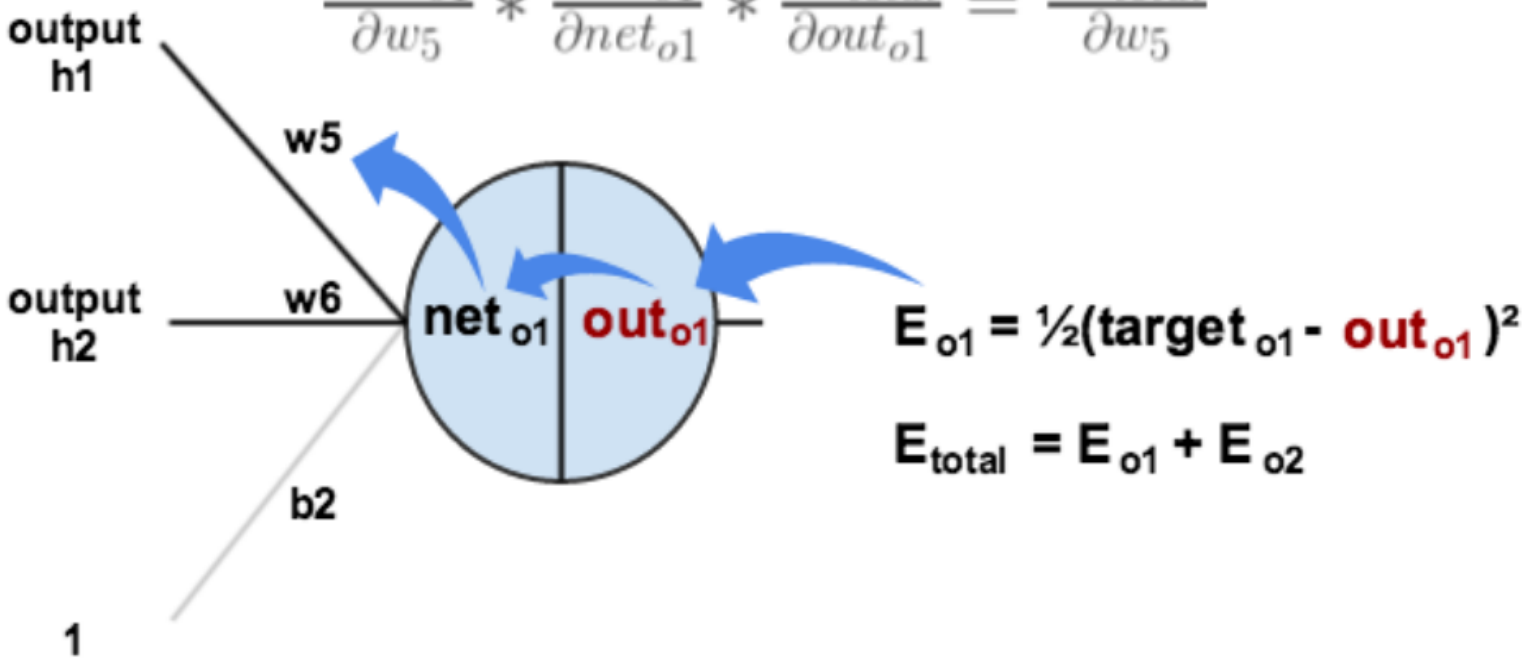
3. THE BACKWARD PASS

FOR W5 WE WANT: $\frac{\partial L_{total}}{\partial w_5}$ [gradient of loss function]

WE APPLY THE CHAIN RULE:

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

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



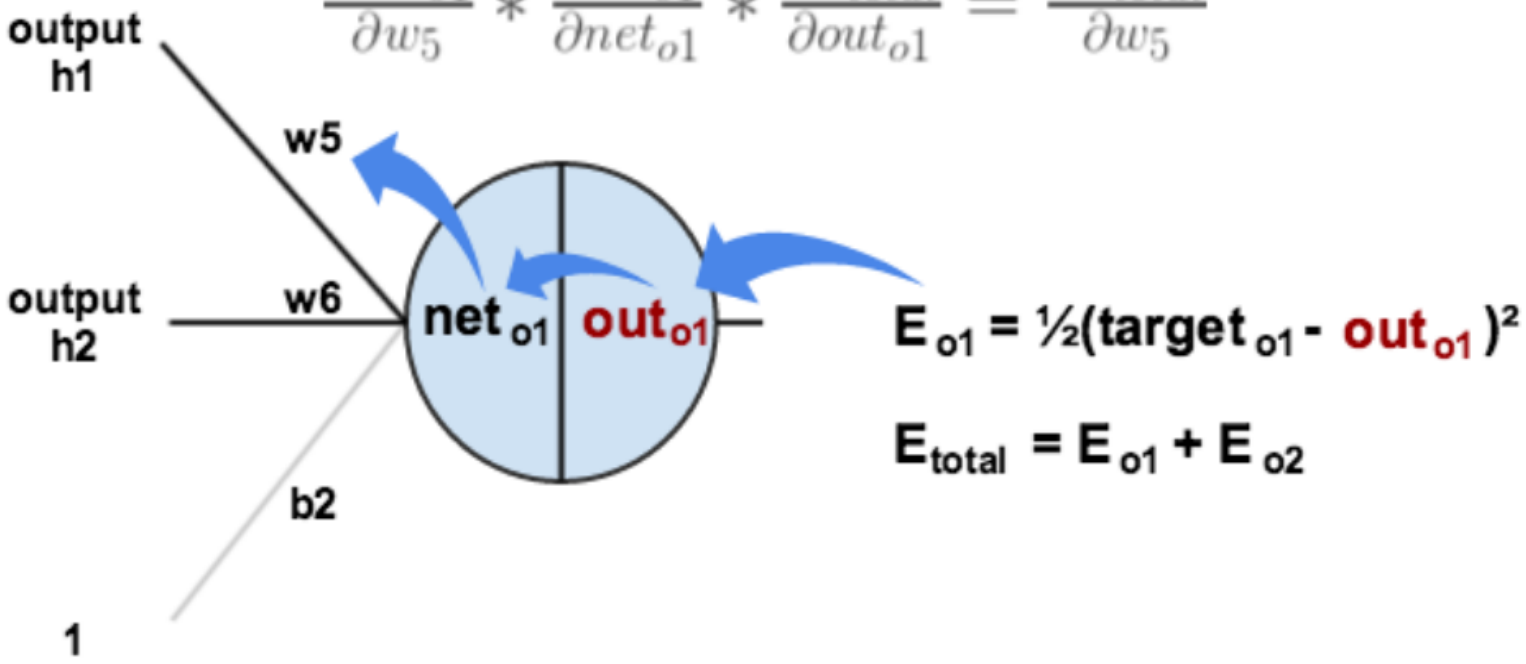
3. THE BACKWARD PASS

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

$$L_{total} = 0.5(target_{o1} - out_{o1})^2 + 0.5(target_{o2} - out_{o2})^2$$

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

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



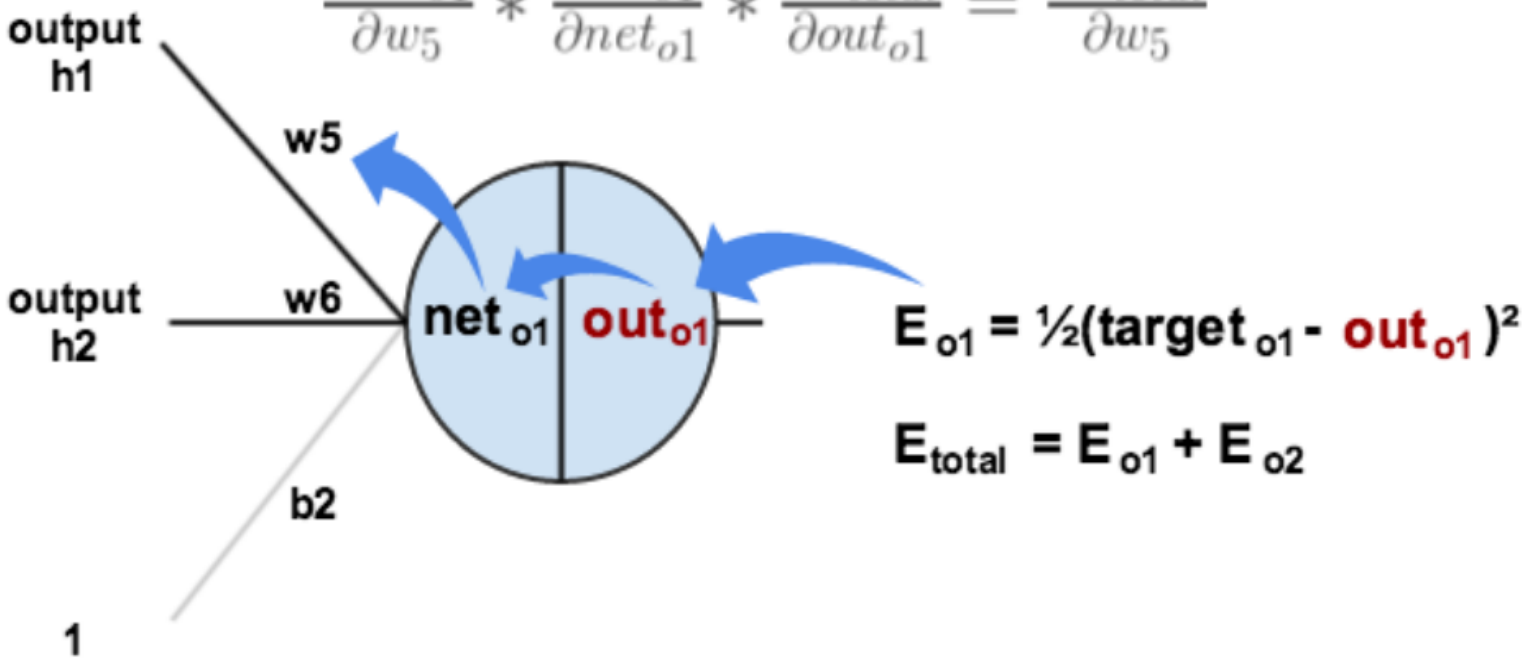
3. THE BACKWARD PASS

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

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

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

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



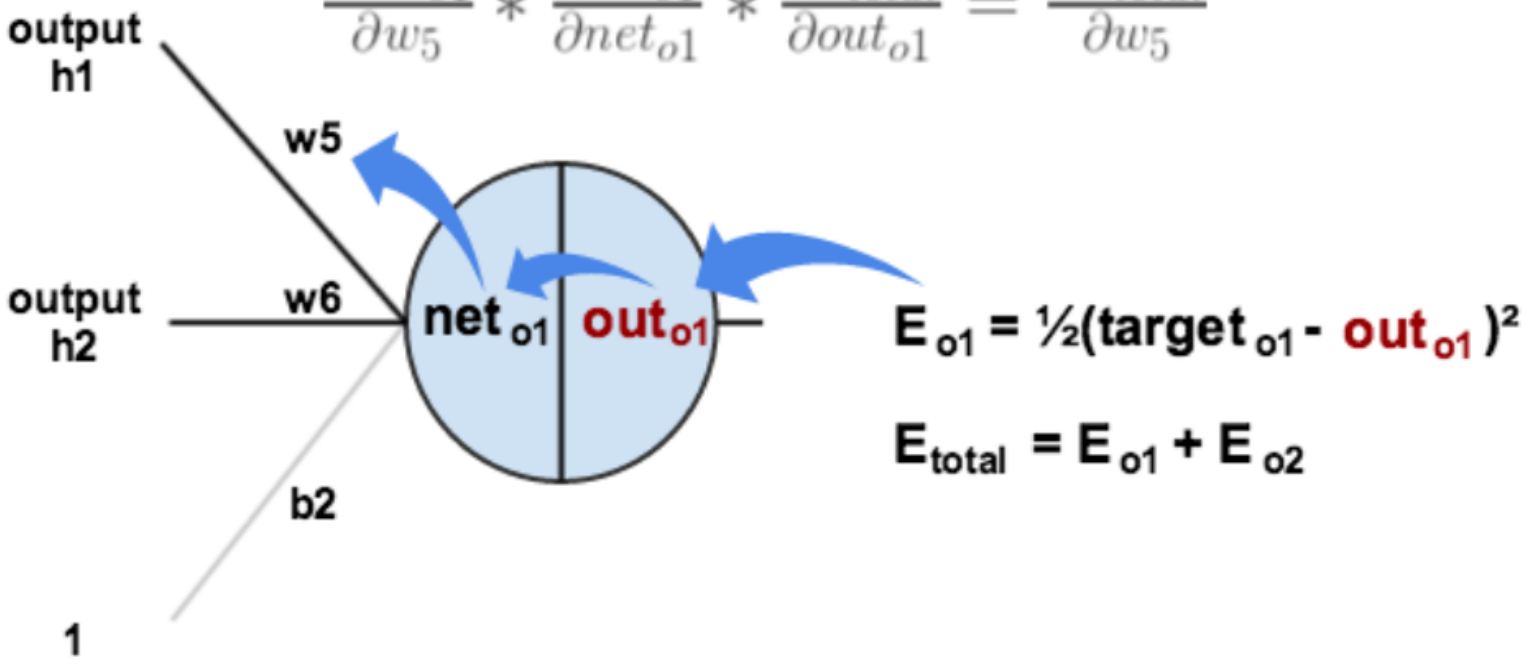
3. THE BACKWARD PASS

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

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

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

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



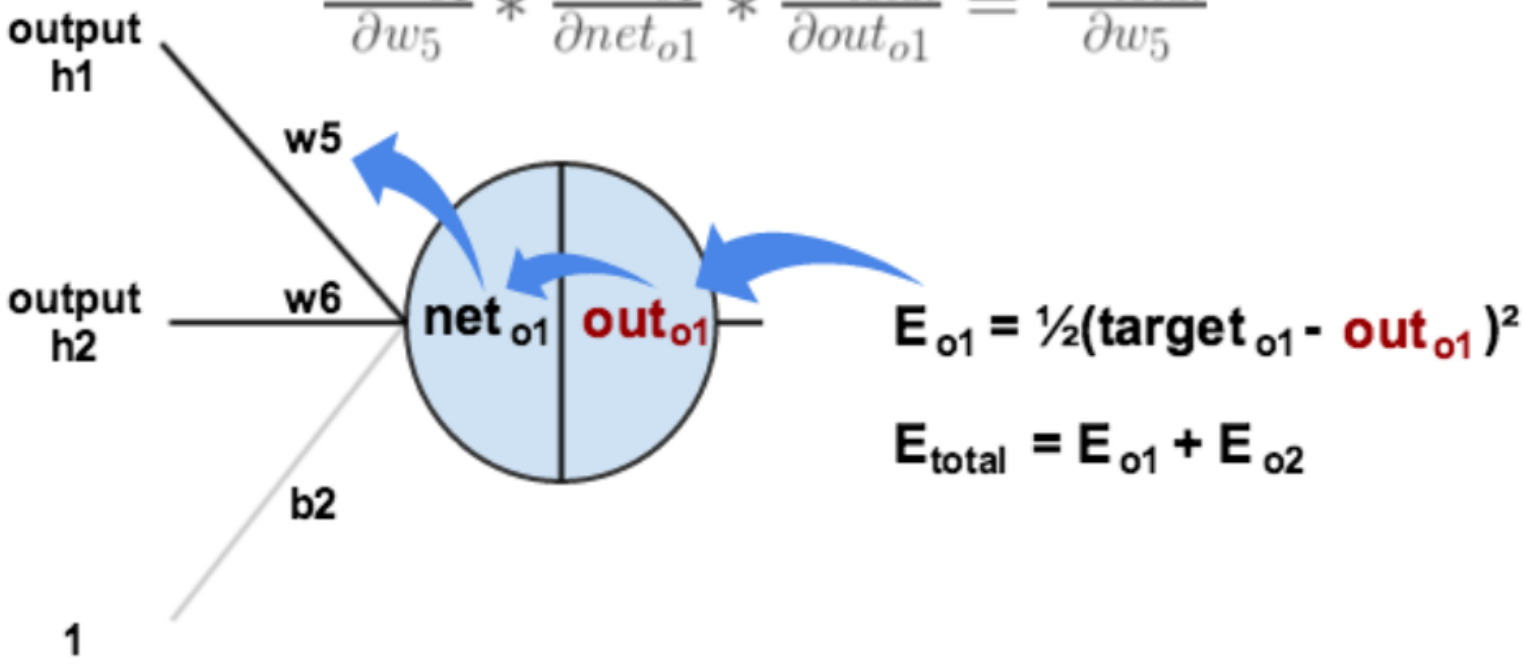
3. THE BACKWARD PASS

ALL TOGETHER:

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

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

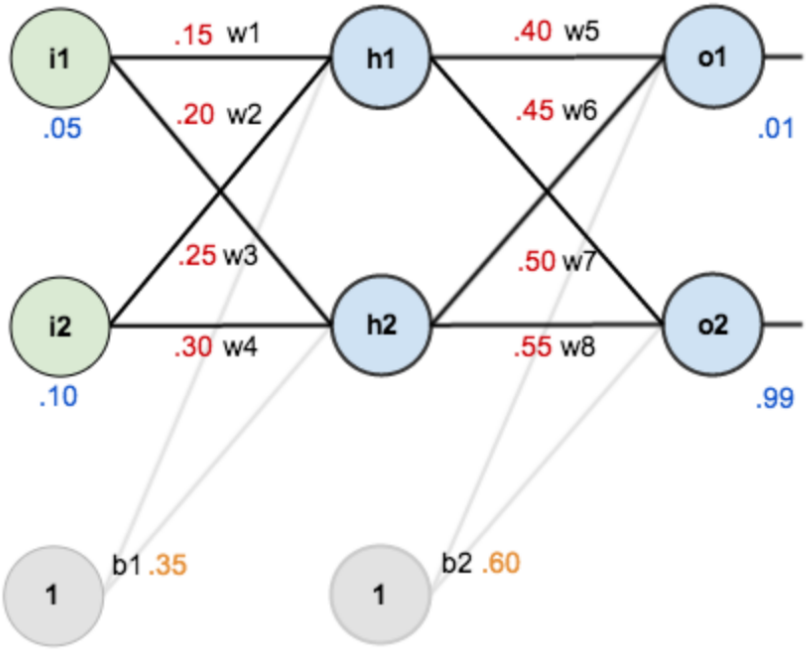
$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



4. UPDATE WEIGHTS WITH GRADIENT AND LEARNING RATE

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

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

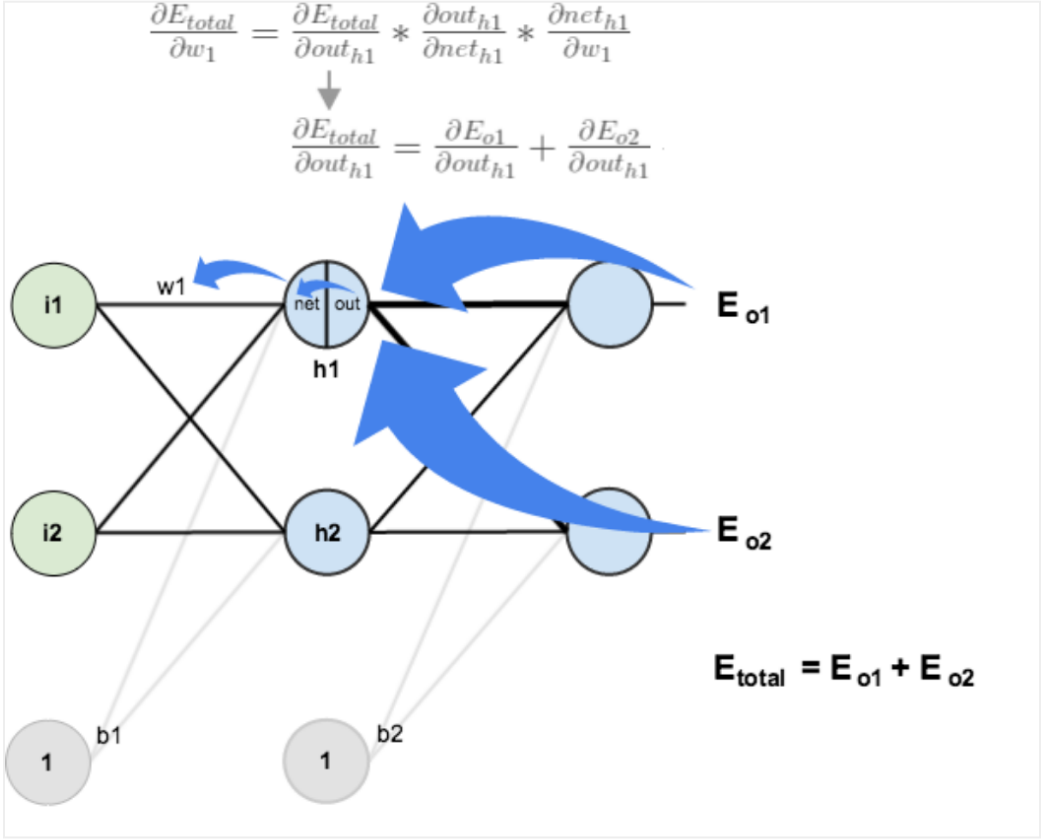


THIS IS REPEATED FOR THE OTHER WEIGHTS OF THE OUTPUT LAYER

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

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

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

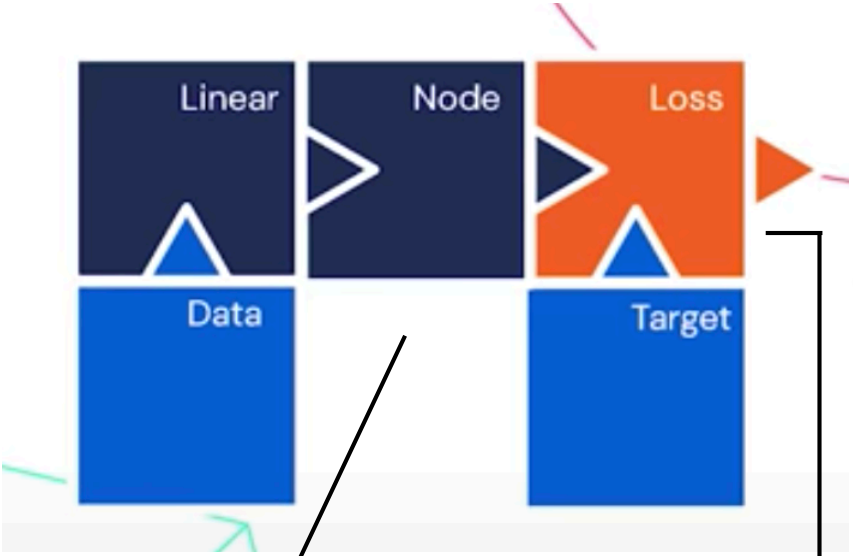


AND BACK-PROPAGATED TO THE HIDDEN LAYERS

IN ORDER TO KNOW MORE ABOUT BACKPROPAGATION…

CLICK HERE

RECAP



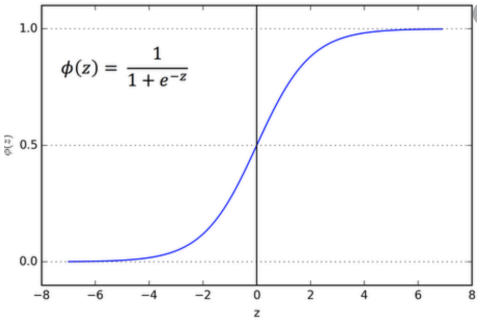
(deepmind
@Czarnecki)

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

$$\vdots$$

$$g_3(W_3 g_2(W_2 g_1(W_1 \vec{x_0})))$$

activation function



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

Minimized through gradient descent (backpropagation)

NEURAL NETWORKS AS STATISTICAL MODELS

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

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

GROUND TRUTH

NETWORK OUTPUT (SOFTMAX)

GROUND TRUTH

NETWORK OUTPUT (SOFTMAX)

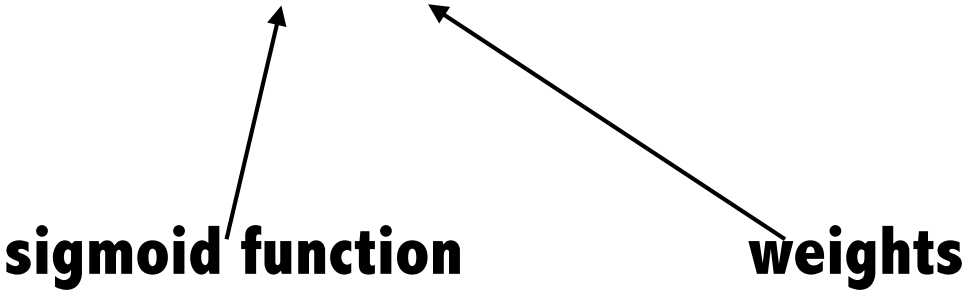
we have a set of independent realizations:

$$(x_i, y_i); i = 1, ..., N$$

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:

$$P(y_i = 0|x_i) = 1 - sigm(f_w(x_i)) \qquad P(y_i = 1|x_i) = sigm(f_w(x_i))$$



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:

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

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

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

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

So, the log-likelihood:

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

THEREFORE EQUIVALENT TO THE CROSS-ENTROPY LOSS:

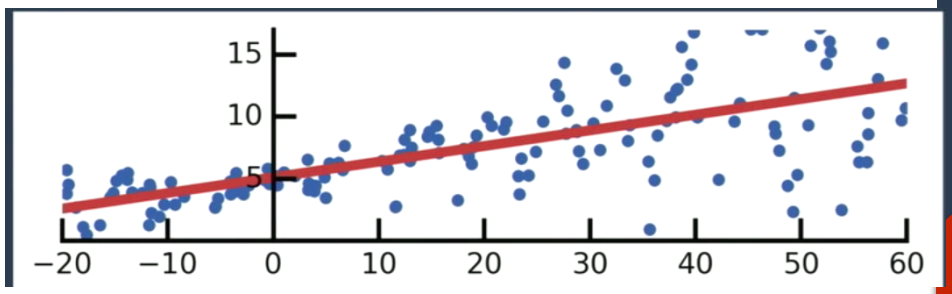
$$l(w; x_i, y_i) = \sum_{i=1}^{N} y_i \times \log[sigm(f_w(x_i))] + (1 - y_i) \times \log[1 - sigm(f_w(x_i))]$$

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

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

NOTE THAT SINCE fw 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 WITH STANDARD DEVIATION EQUAL TO UNITY. OUR ESTIMATOR IS THEREFORE THE MEAN OF THE NORMAL PDF THAT MAXIMIZES THE LIKELIHOOD.

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

weights

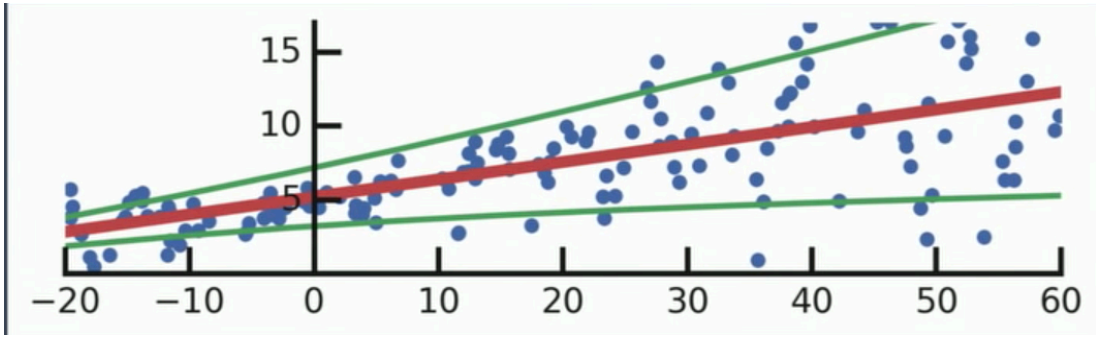
neural network

SO, IN A NUTSHELL, WE ARE DOING BAYESIAN VARIATIONAL INFERENCE ASSUMING THEREFORE A PDF FOR OUR OUTPUT VARIABLE (BERNOUILLI 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

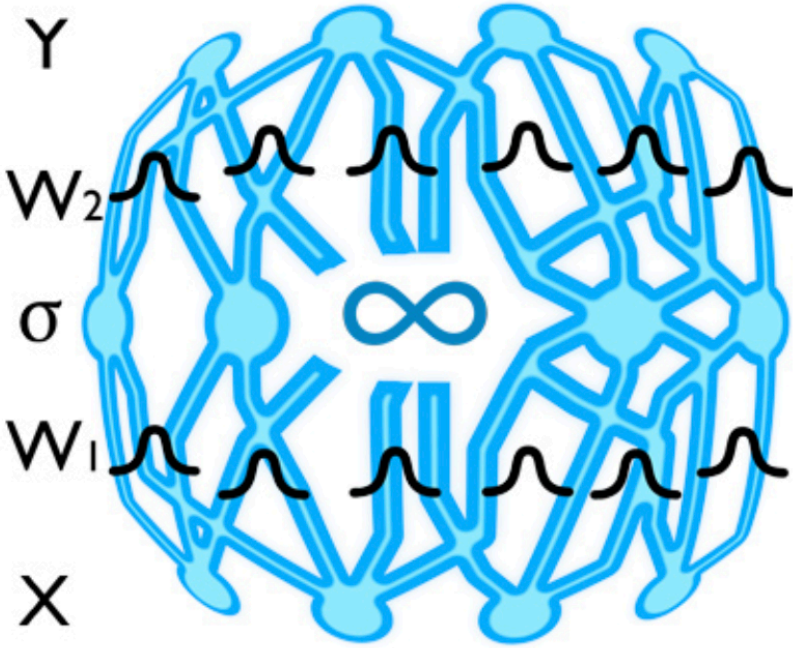
DEV SUMMIT 2019

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

} Our wish.

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



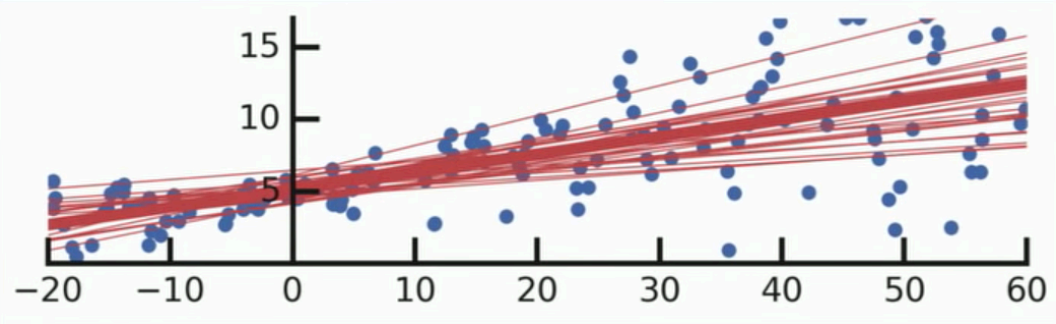
CAPTURING "MODEL UNCERTAINTY"

```
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

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

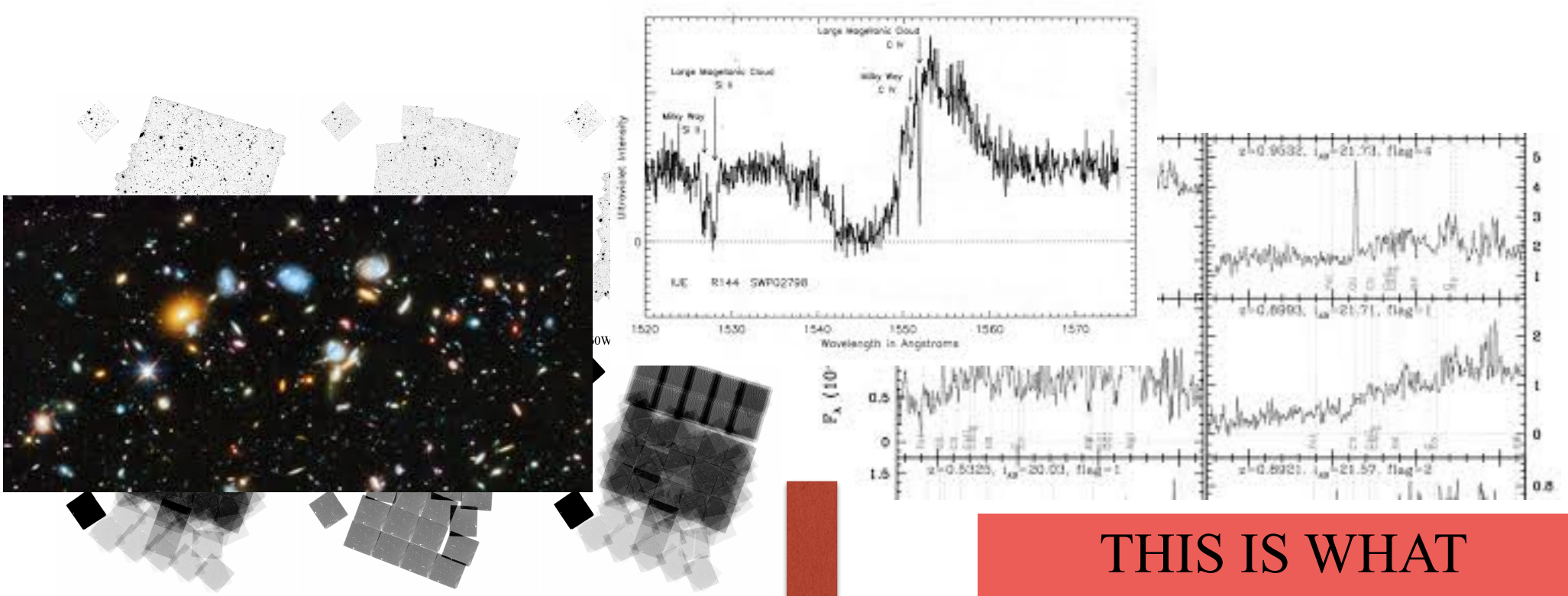


BNNs ADD A PRIOR DISTRIBUTION TO EACH WEIGHT - HARD TO TRAIN

CAN WE GO DEEP NOW?

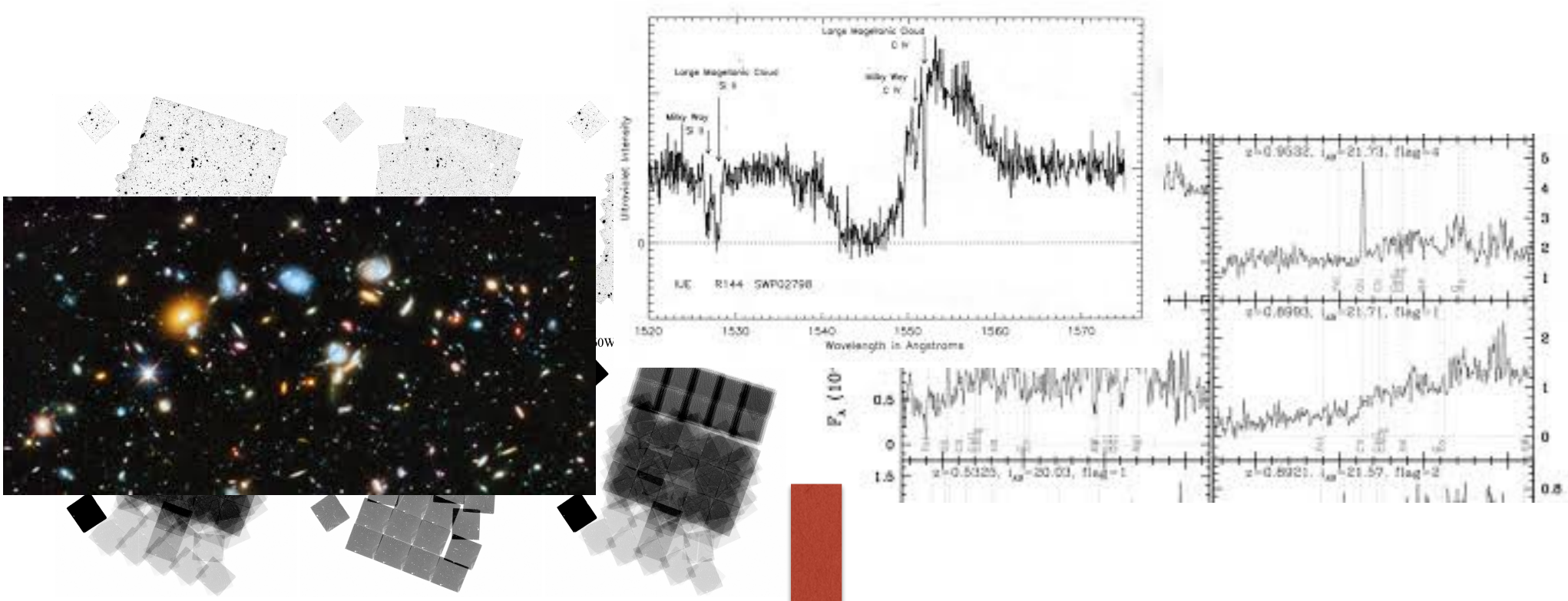
ALMOST THERE…LET'S THINK FOR A MOMENT ABOUT WHAT WE PUT AS INPUT…

What do we put as input?



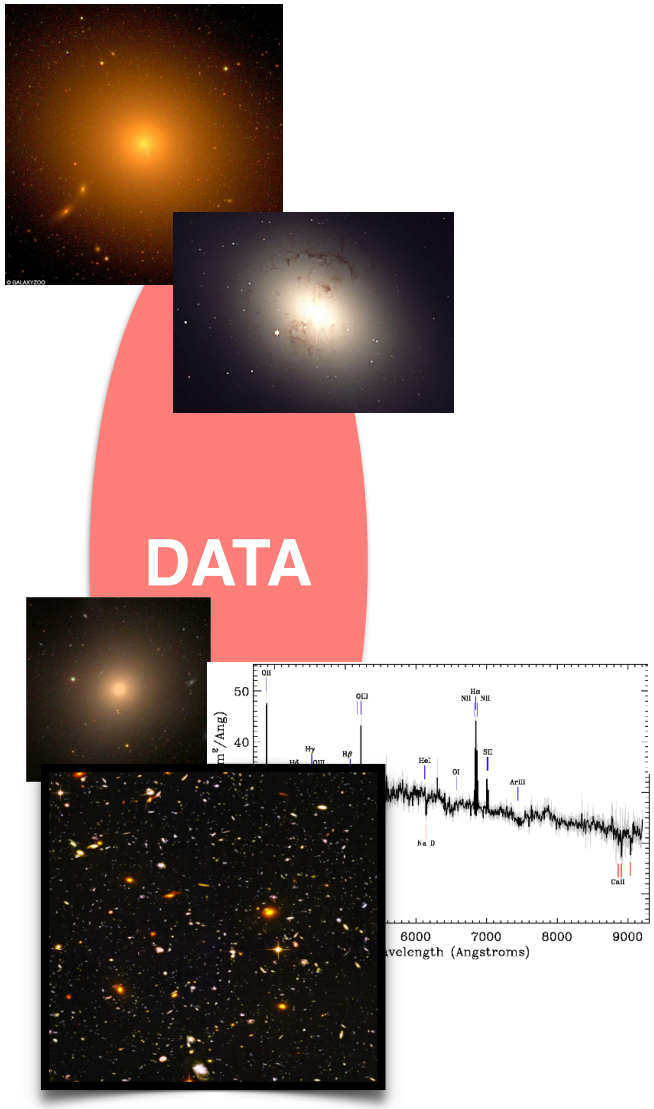
THIS IS WHAT MACHINES SEE

What do we put as input?



PRE-PROCESS DATA TO EXTRACT MEANINGFUL INFORMATION

THIS IS GENERALLY CALLED **FEATURE EXTRACTION**



Galaxy!

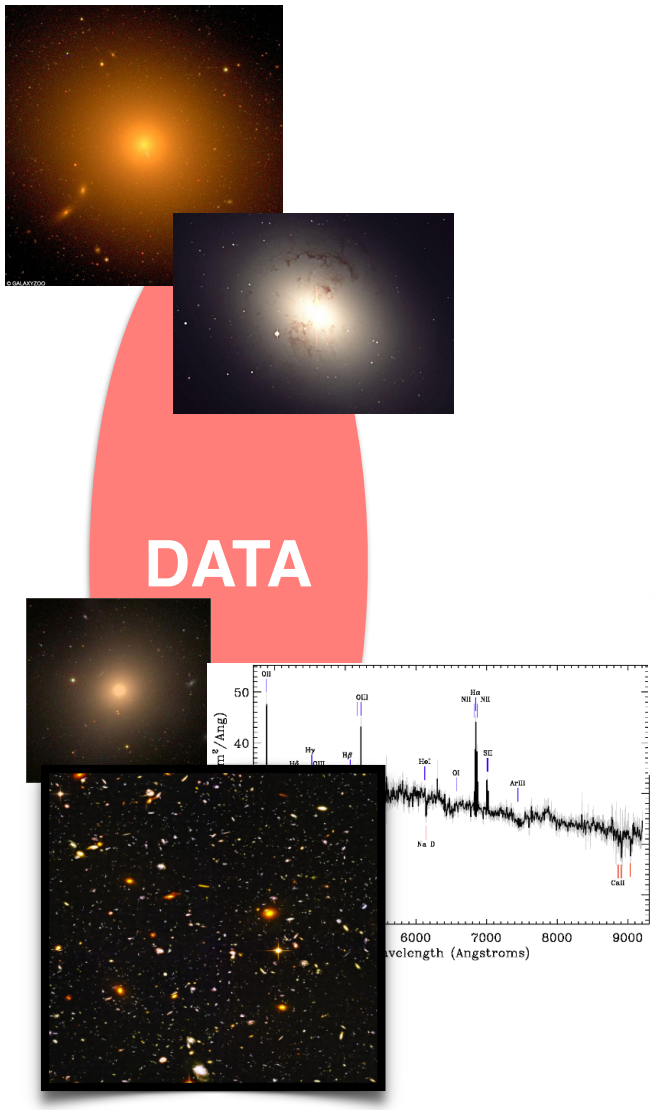
Star!

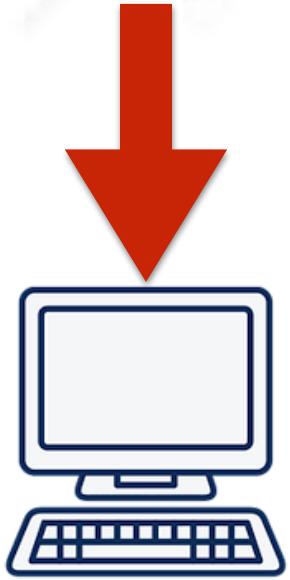
Merger!

Clump!

AGN!

Knowledge of Physics





Galaxy!

Star!

Merger!

Clump!

AGN!

$$f_{\vec{W}}(\vec{x}) = \vec{y}$$

LABEL

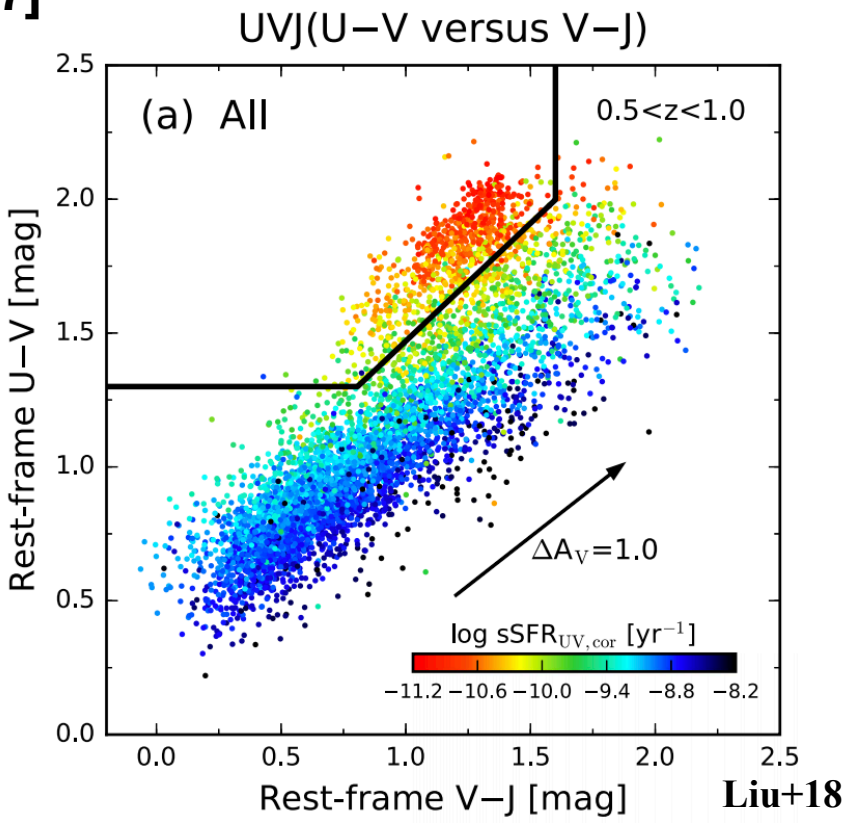
Q(0) , SF(1)

NETWORK FUNCTION

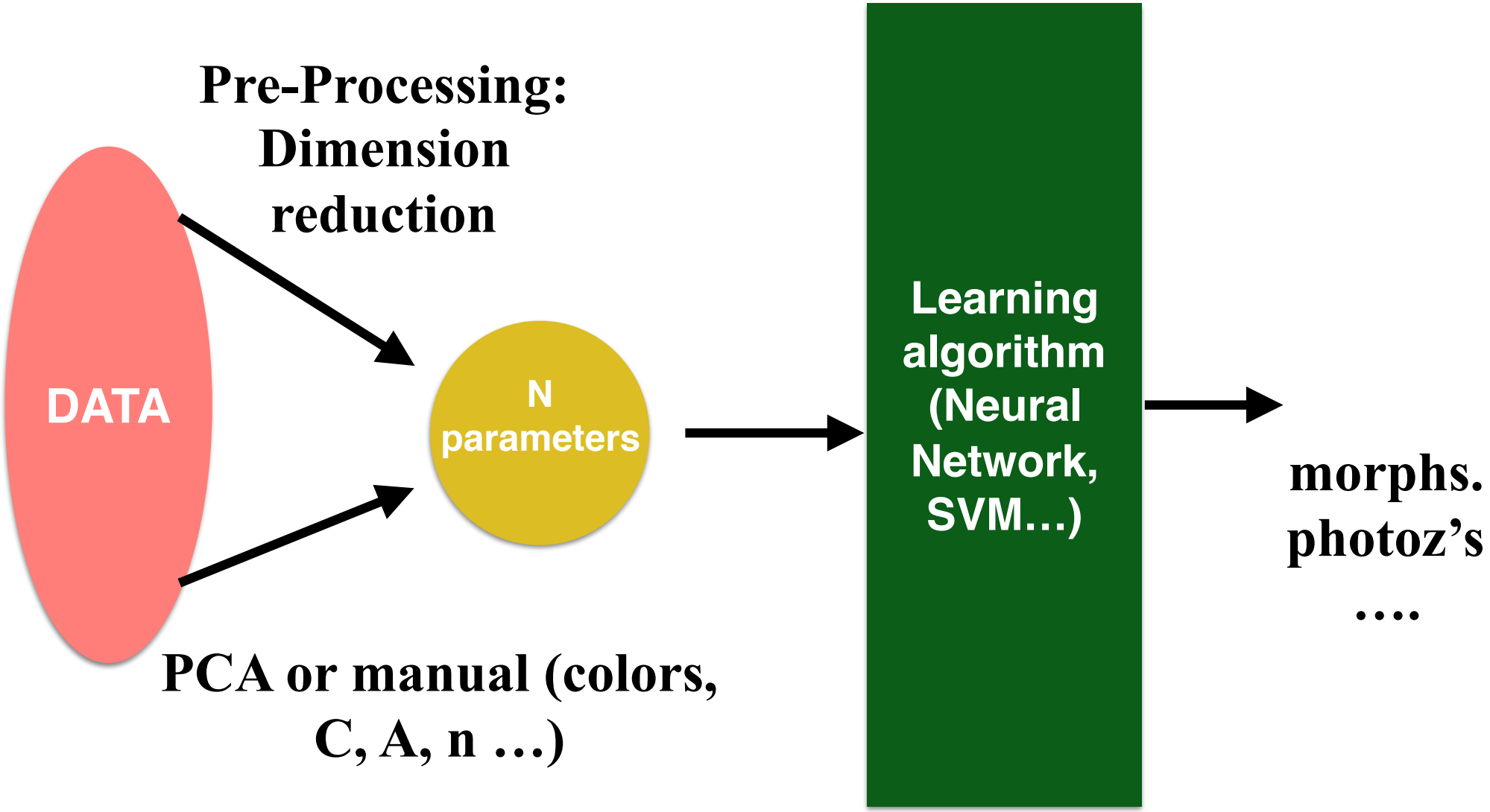
(U-V, V-J) FEATURES

sgn[(u-v)-0.8*(v-j)-0.7]

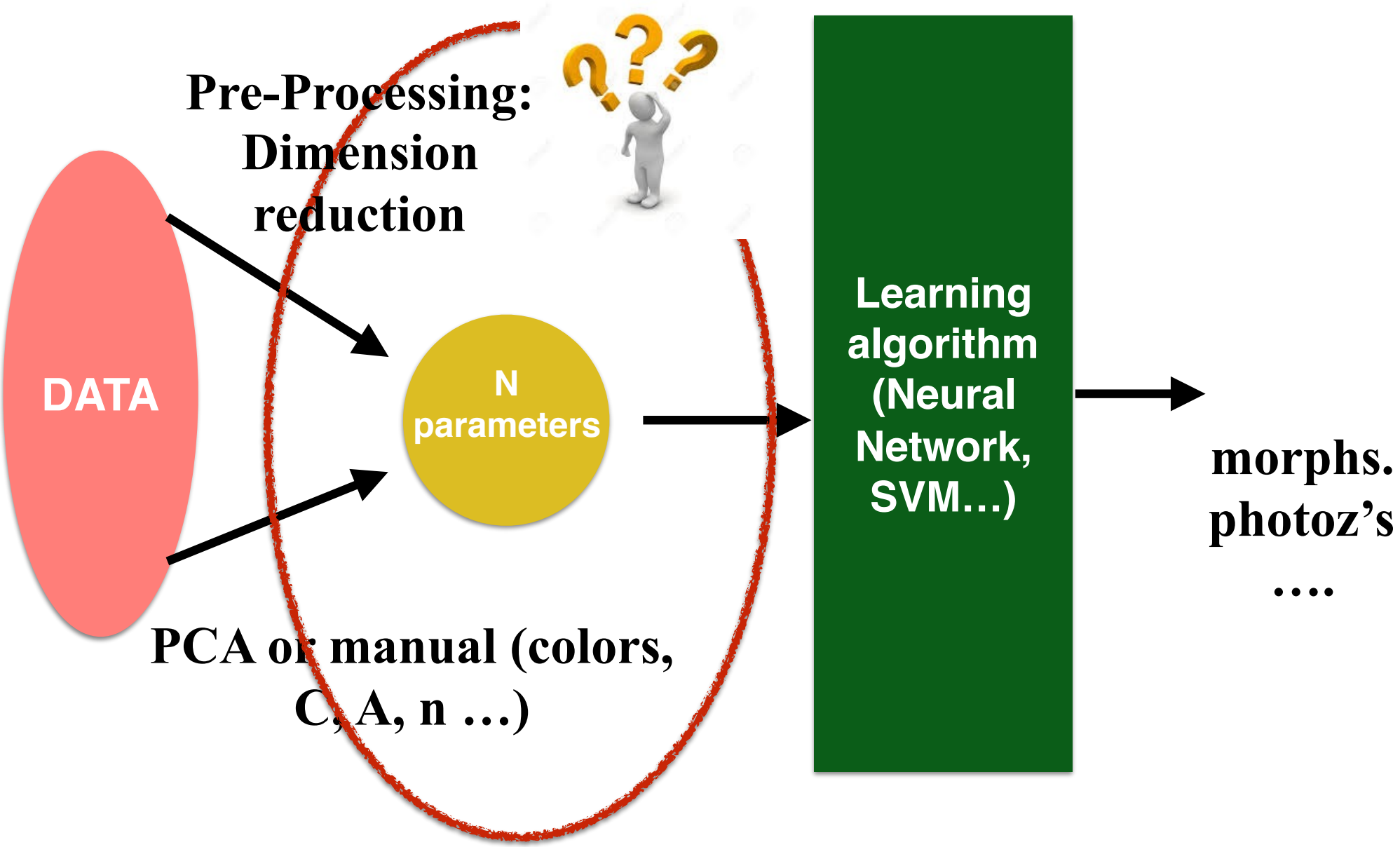
WEIGHTS



THE "CLASSICAL" APPROACH

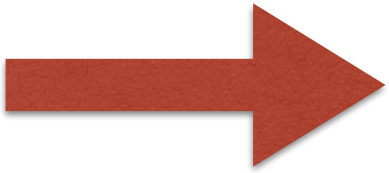


"CLASSICAL" MACHINE LEARNING



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

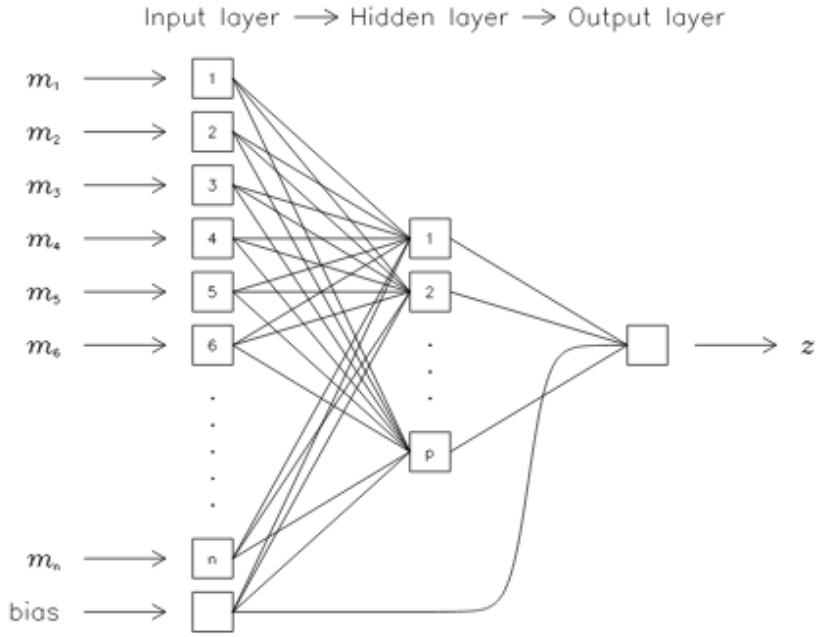
SDSS

g

r

i

z



Collister+08

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

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

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

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

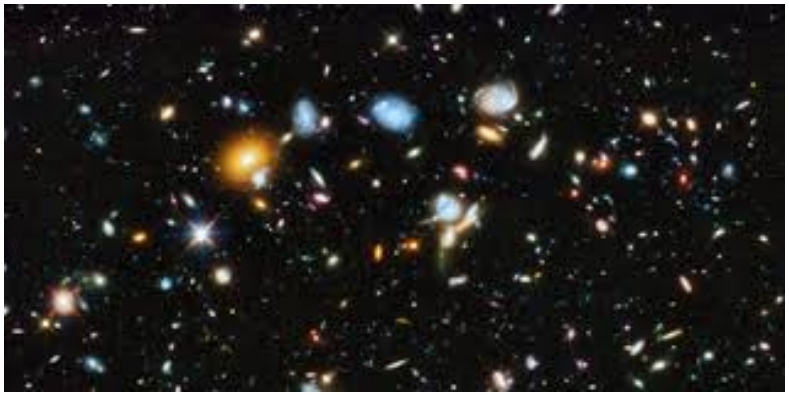
LARGE DIMENSION SIGNALS SUCH AS IMAGES OR SPECTRA WOULD REQUIRE TREMENDOUSLY LARGE MODELS

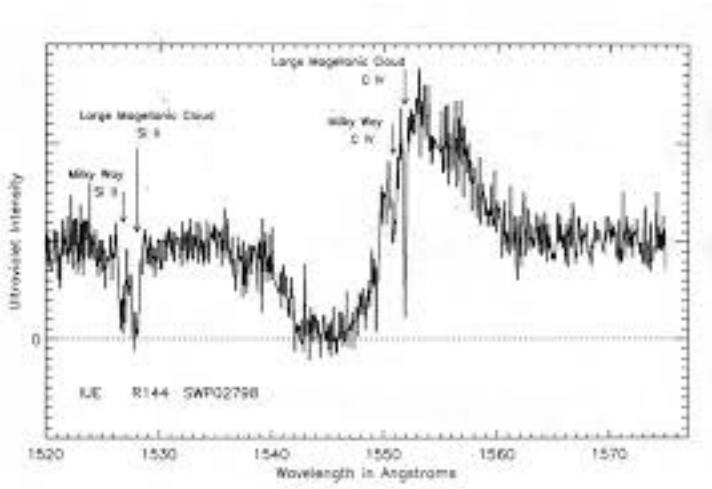
A 512x512 image as input of a fully connected layer producing output of same size:

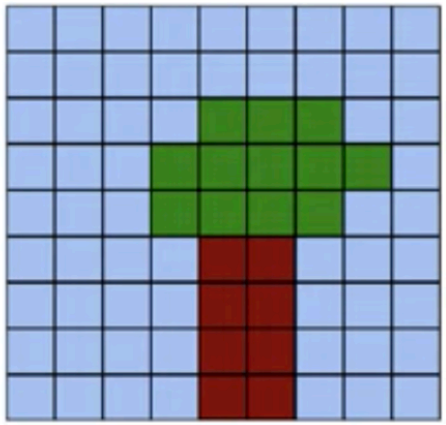
$$(512 \times 512)^2 = 7e10$$

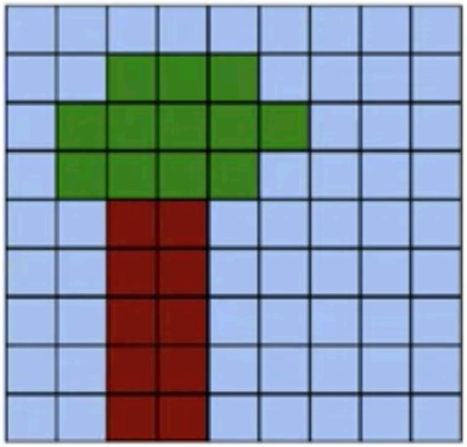
BUT

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

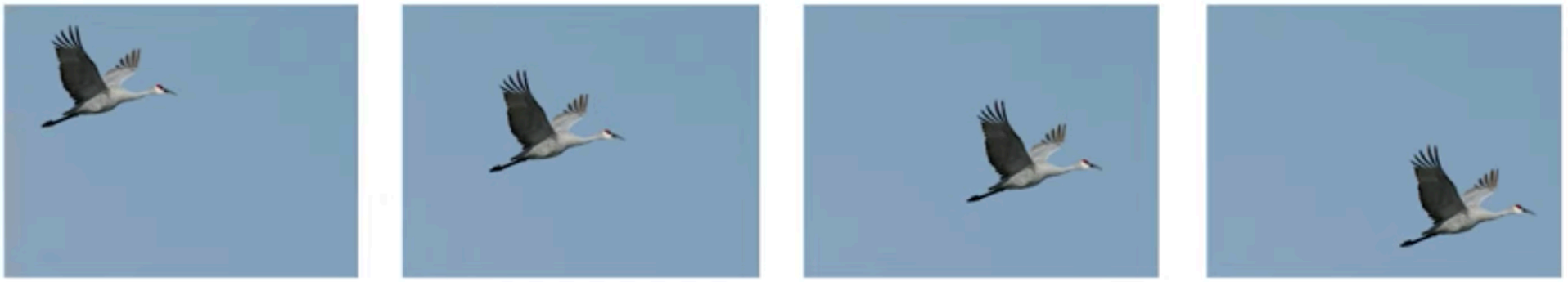








(Dielemann@Deepmind)



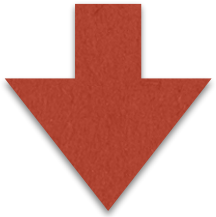
TWO BASIC PROPERTIES OF IMAGING DATA (BUT ALSO SPECTROSCOPY IN SOME SENSE) ARE **LOCALITY** **TRANSLATION INVARIANCE**

locality: nearby pixels are more strongly correlated

translational invariance: meaningful patterns can appear anywhere in the image

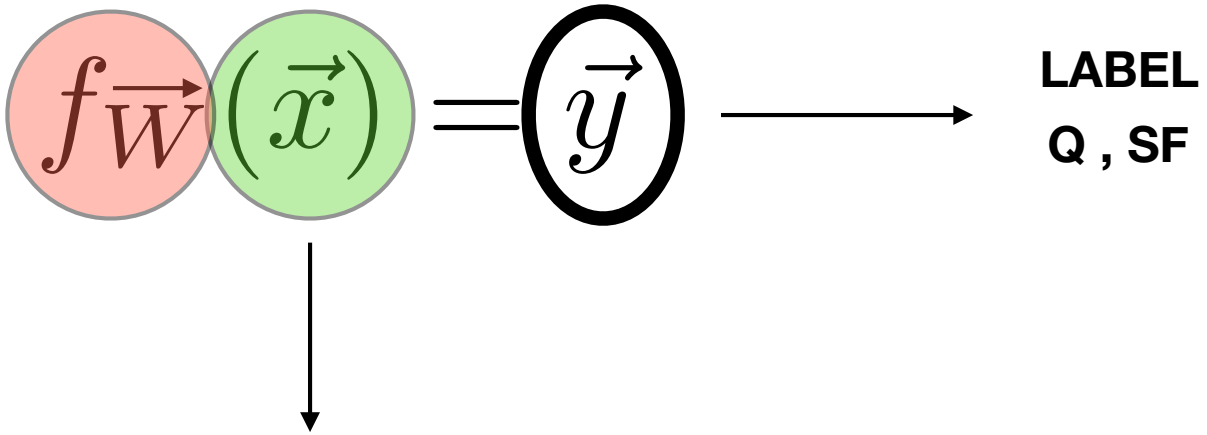
(Dielemann@Deepmind)

FEEDING INDIVIDUAL RESOLUTION ELEMENTS IS NOT VERY EFFICIENT SINCE IT LOOSES 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….