https://introml.mit.edu/

# 6.390 Intro to Machine Learning

## Lecture 6: Neural Networks

Shen Shen

March 8, 2024

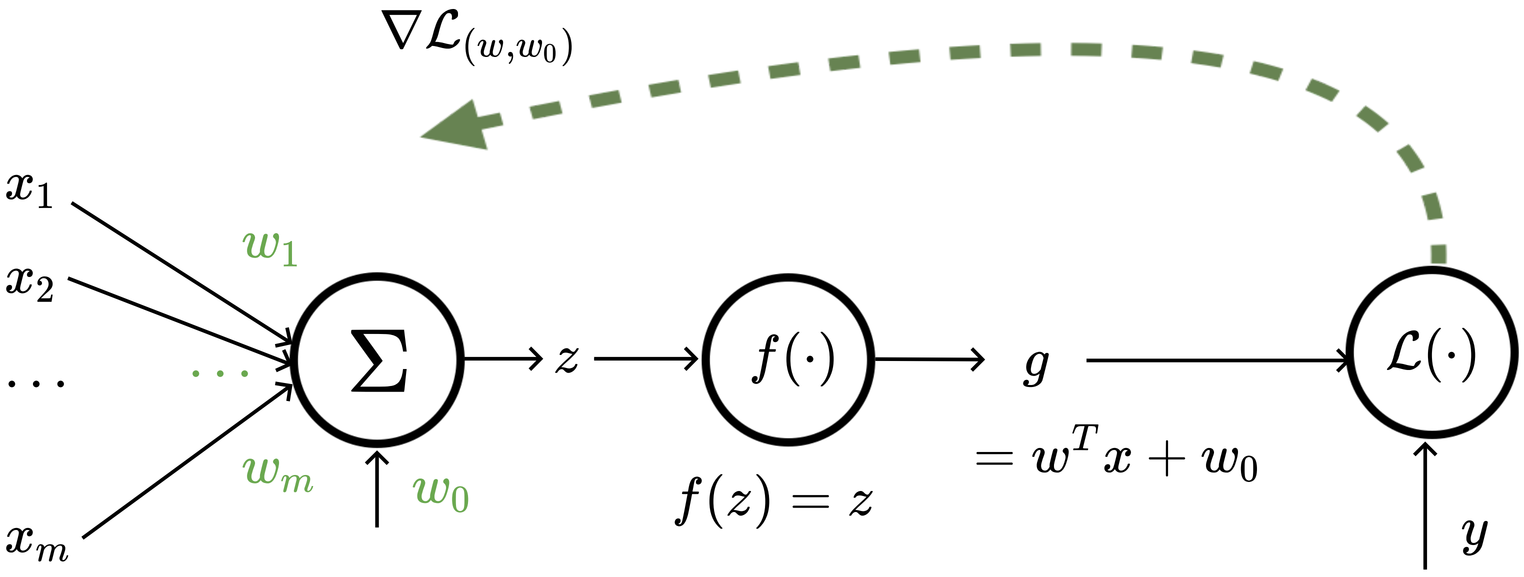
(👉 the Live Slides)

(many slides adapted from Phillip Isola and Tamara Broderick)

# Outline

- Recap and neural networks motivation
- Neural Networks
  - A single neuron
  - A single layer
  - Many layers
  - Design choices (activation functions, loss functions choices)
- Forward pass
- Backward pass (back-propogation)

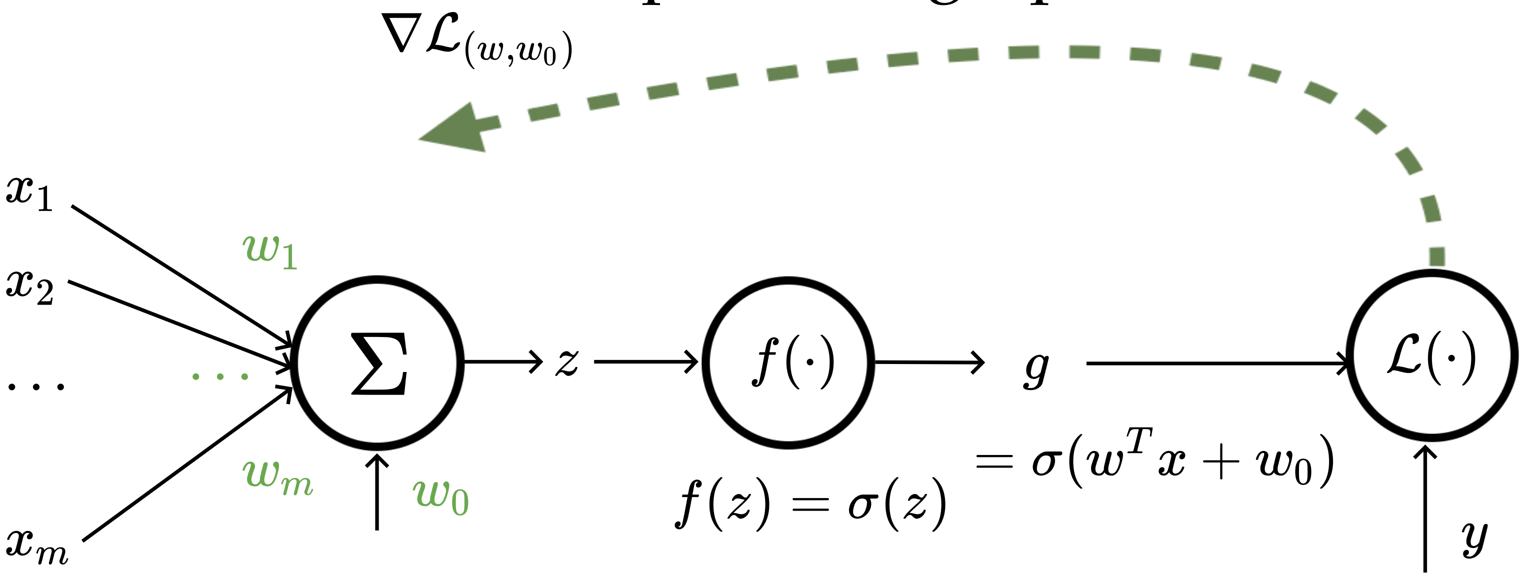
# e.g. linear regression represented as a computation graph




learnable parameters (weights)

- Each data point incurs a loss of $(w^T x^{(i)} + w_0 - y^{(i)})^2$
- Repeat for each data point, sum up the individual losses
- Gradient of the total loss gives us the "signal" on how to optimize for $w, w_0$

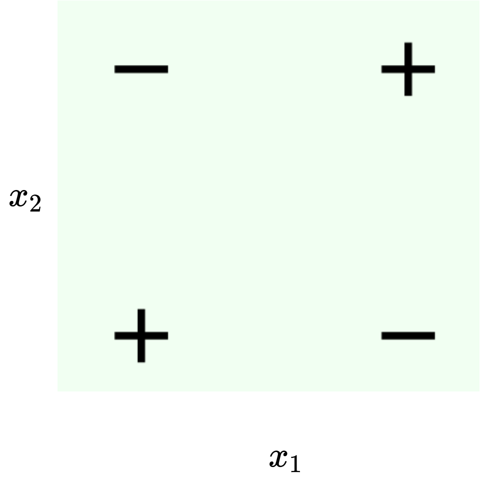
# e.g. linear logistic regression (linear classification) represented as a computation graph




learnable parameters (weights)

- Each data point incurs a loss of $-\left(y^{(i)} \log g^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - g^{(i)}\right)\right)$
- Repeat for each data point, sum up the individual losses
- Gradient of the total loss gives us the "signal" on how to optimize for $w, w_0$

We saw that, one way of getting complex input-output behavior is to leverage nonlinear transformations

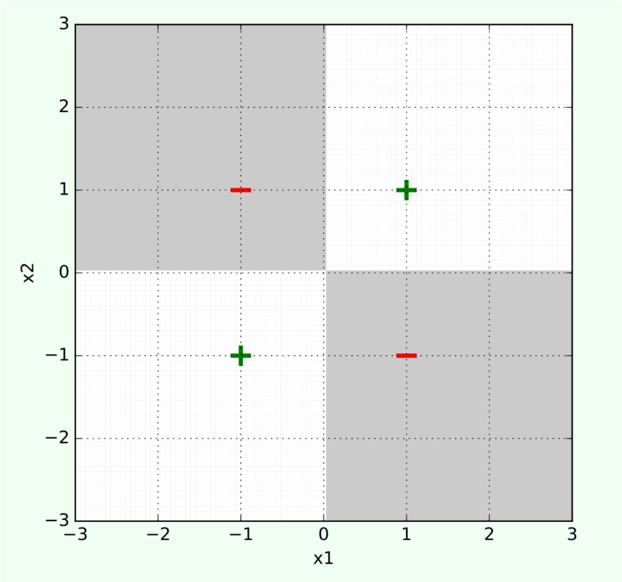



transform

$$\phi\left(\left[x_1, x_2\right]^\top\right) = \left[1, x_1, x_2, x_1^2, x_1 x_2, x_2^2\right]^\top$$

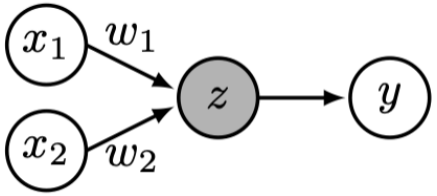
e.g. use for decision boundary

$$\underline{\mathrm{sign}(0 + 0x_1 + 0x_2 + 0x_1^2 + 4x_1x_2 + 0x_2^2 + 0)}$$



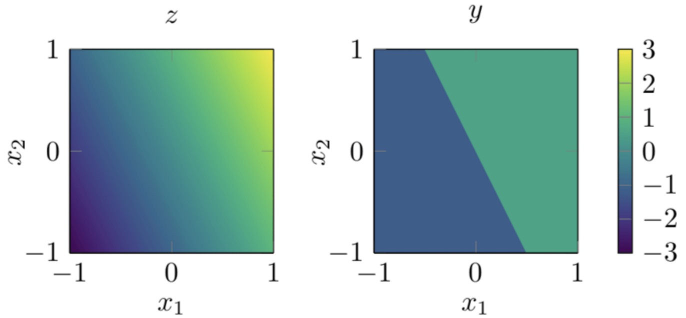

👆 importantly, linear in $\phi$, non-linear in $x$

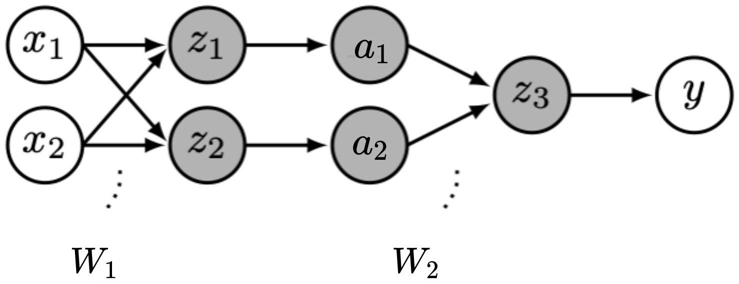
# Today (2nd cool idea): "stacking" helps too!




$z = w^T x$

$y = \text{sign}(z)$



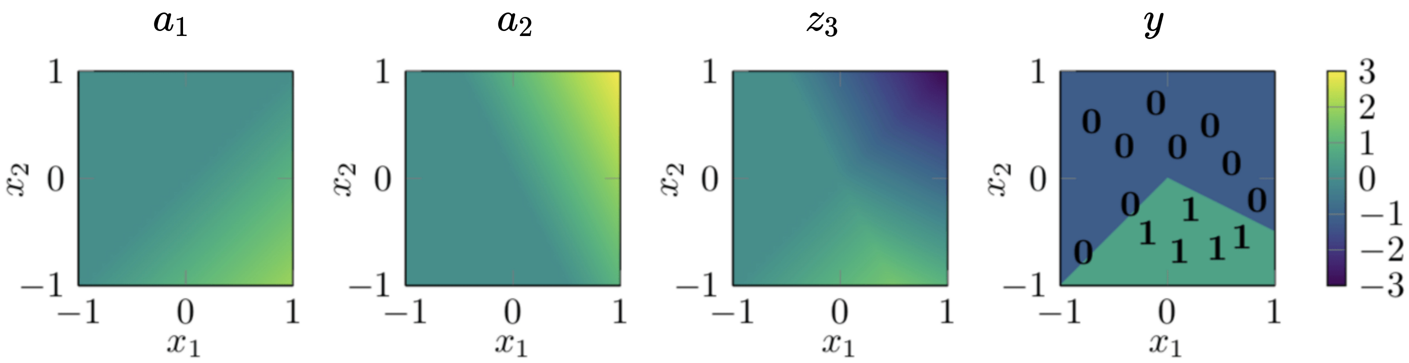




$\mathbf{z} = \mathbf{x}^T \mathbf{W}_1$

$\mathbf{a} = \text{sign}(\mathbf{z})$

$z_3 = \mathbf{a}^T \mathbf{W}_2$

$y = \text{sign}\left(z_3\right)$



So, two epiphanies:

- nonlinearity empowers linear tools
- stacking helps

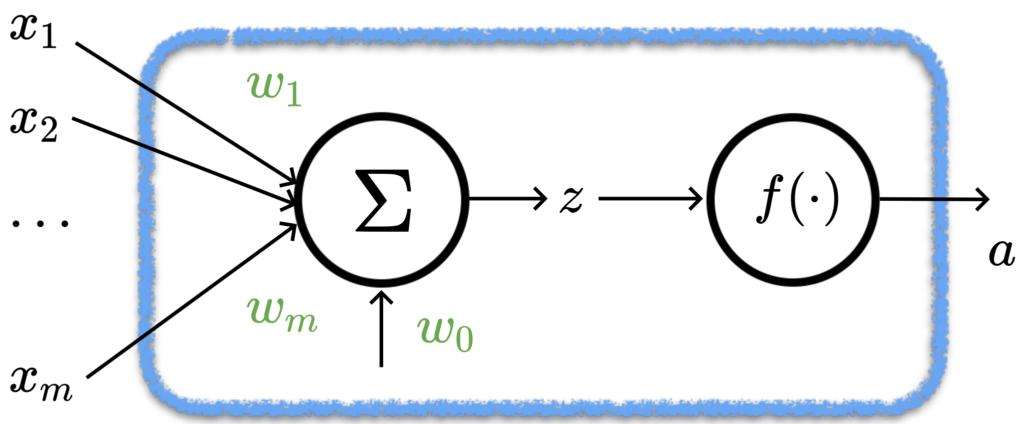

(👋 heads-up: all neural network graphs focus on a single data point for simple illustration.)

# Outline

- Recap and neural networks motivation
- Neural Networks
  - A single neuron
  - A single layer
  - Many layers
  - Design choices (activation functions, loss functions choices)
- Forward pass
- Backward pass (back-propogation)

# A single neuron is

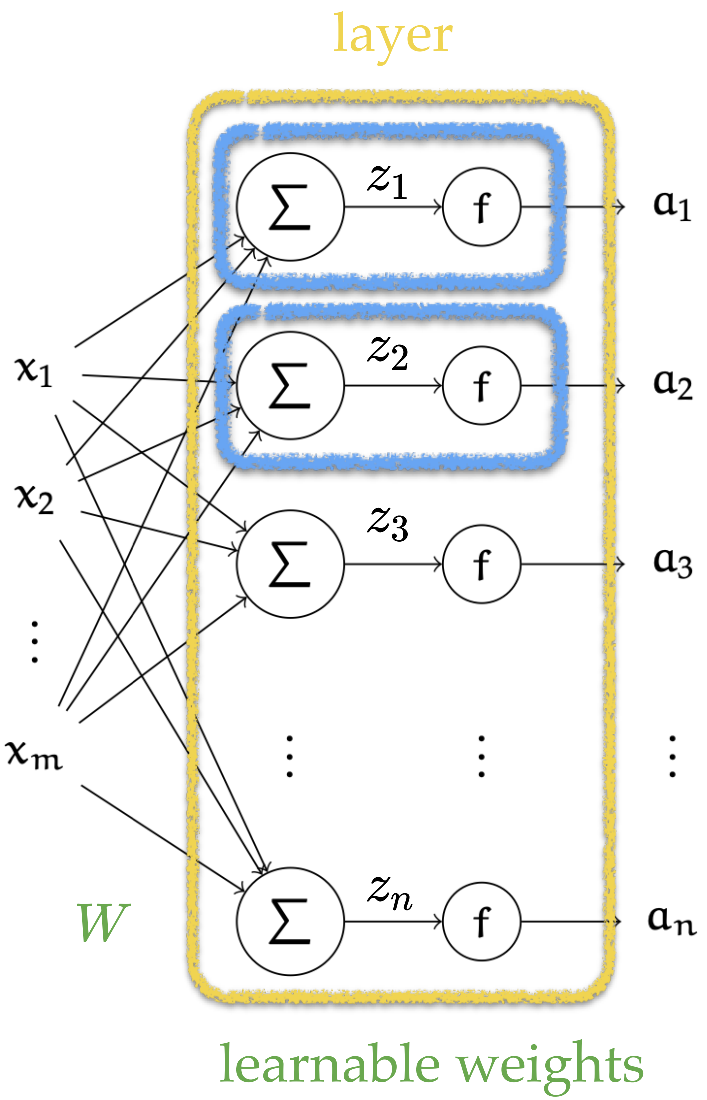
- the basic operating "unit" in a neural network.
- the basic "node" when a neural network is viewed as computational graph.




- $x$: $m$-dimensional input (a single data point)
- $w$: weights (i.e. parameters)
- $z$: pre-activation **scalar** output
- $f$: activation function
- $a$: post-activation **scalar** output

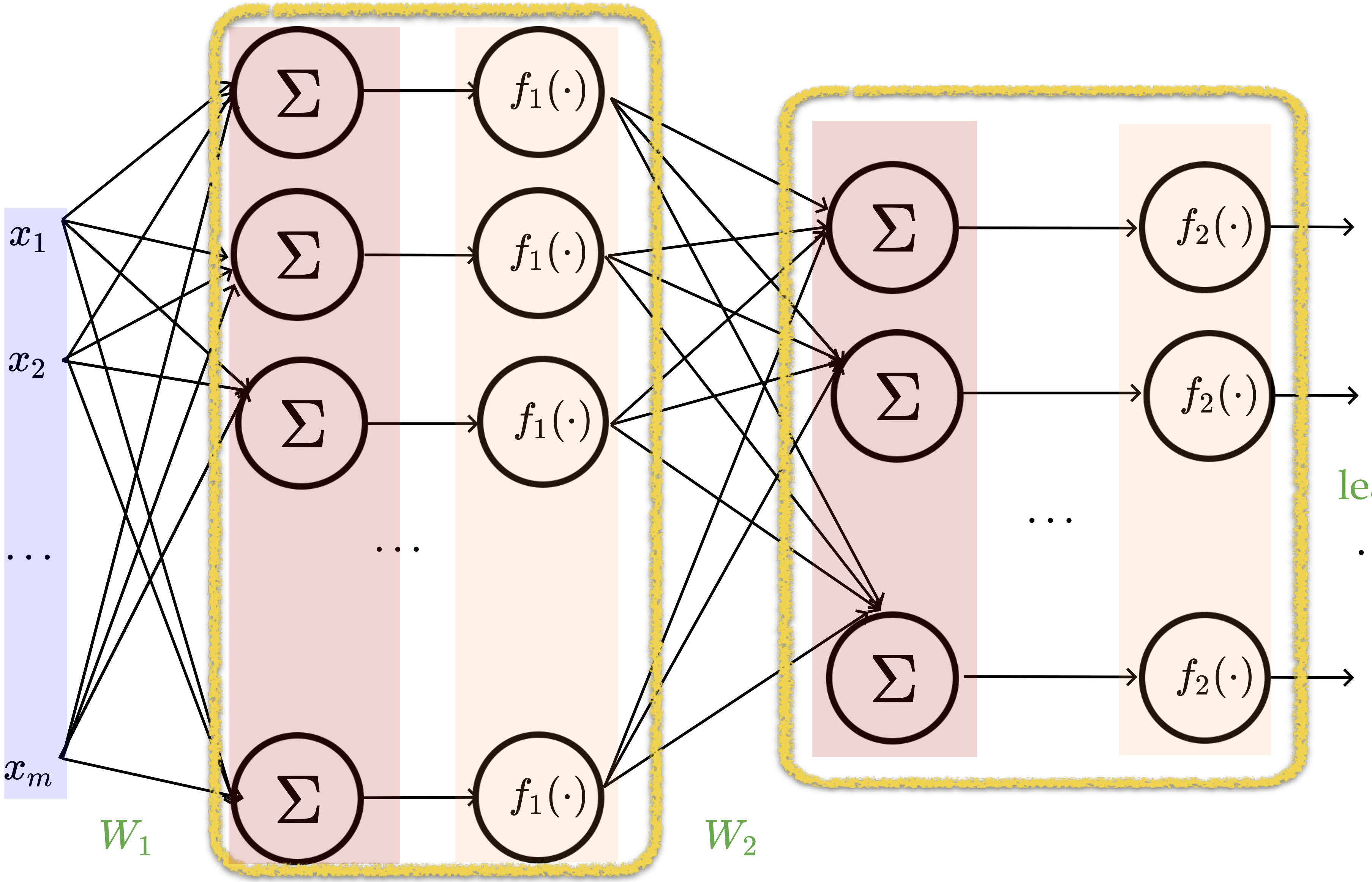
- $\boxed{\text{neuron}}$, a function, maps a vector input $x \in \mathbb{R}^m$ to a scalar output
- inside the neuron, circles do function evaluation/computation
- $f$: we engineers choose
- $w$: learnable parameters

# A single layer is

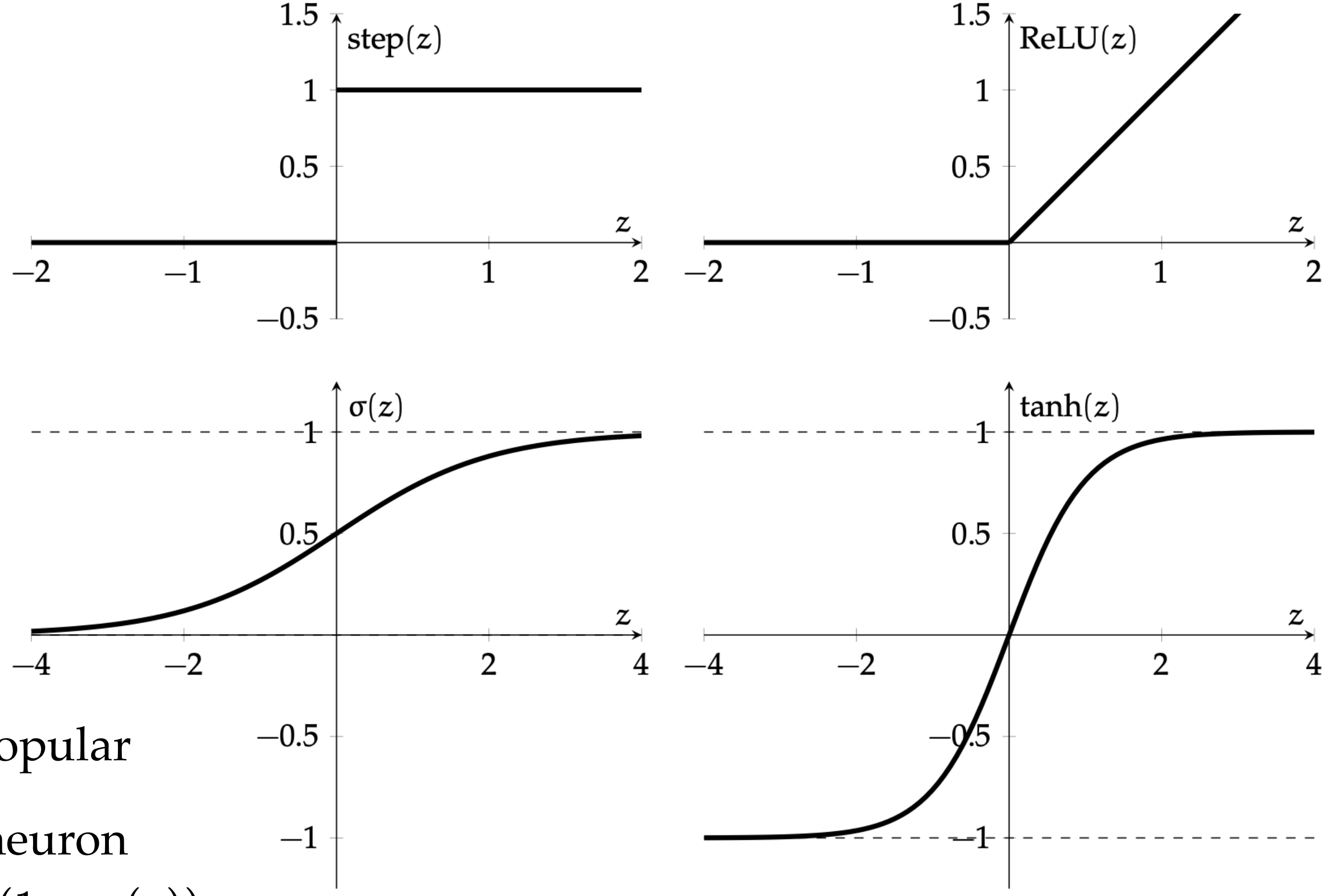



- made of many individual neurons.
- (# of neurons) = (layer output dimension).
- typically, all neurons in one layer use the same activation $f$ (if not; uglier/messier algebra)
- typically, no "cross-wire" between neurons. e.g. $z_1$ doesn't influence $a_2$. in other words, a layer has the same activation applied element-wise. (softmax is an exception to this, details later.)
- typically, fully connected. i.e. there's an edge connecting $x_i$ to $z_j$, for all $i \in \{1, 2, 3, \ldots, m\}; j \in \{1, 2, \ldots, n\}$. in other words, all $x_i$ influence all $a_j$.

# A (feed-forward) neural network is



# Activation function $f$ choices



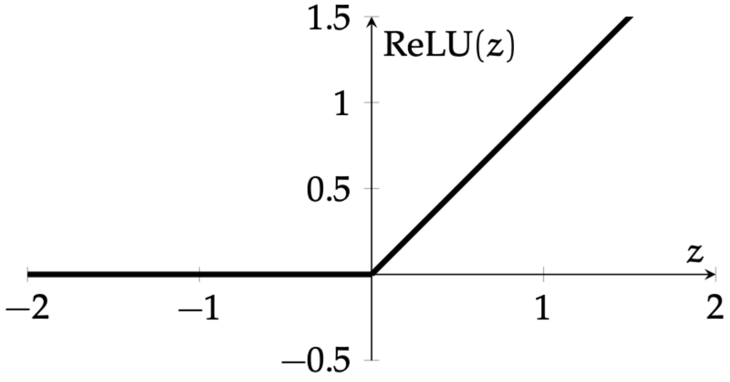

$\sigma$ used to be popular

- firing rate of neuron
- $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$

# ReLU is the de-facto activation choice nowadays

$$\mathrm{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

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

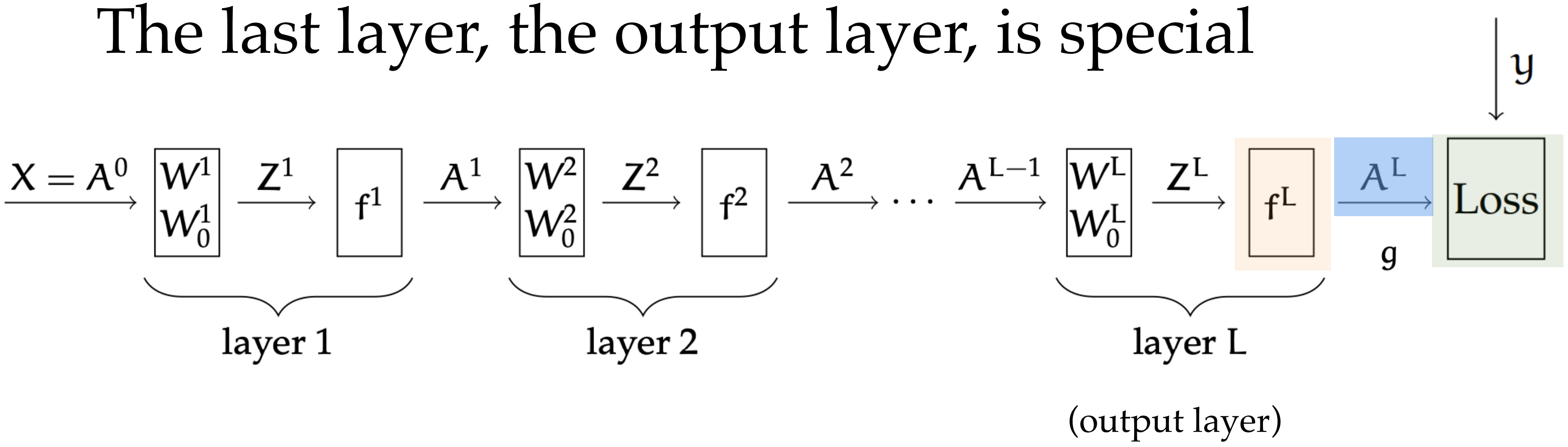



- **Default** choice in **hidden** layers.
- Pro: **very** efficient to implement, choose to let the gradient be:

$$\frac{\partial \mathrm{ReLU}(z)}{\partial z} := \begin{cases} 0, & \text{if} \quad z < 0 \\ 1, & \text{if} \quad \text{otherwise} \end{cases}$$

- Drawback: if strongly in negative region, unit can be "dead" (no gradient).
- Inspired variants like elu, leaky-relu.

# The last layer, the output layer, is special



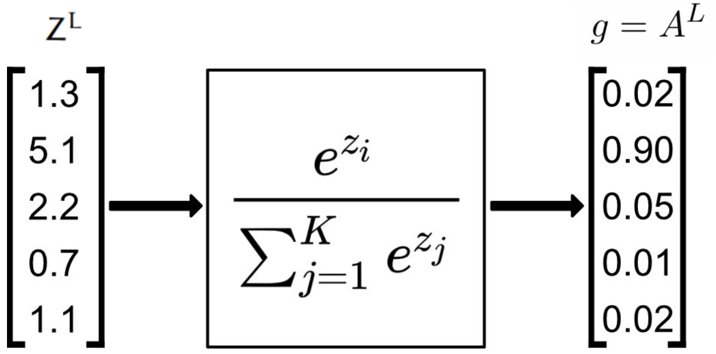

- activation and loss depends on problem at hand
- we've seen e.g. regression (one unit in last layer, squared loss).

---

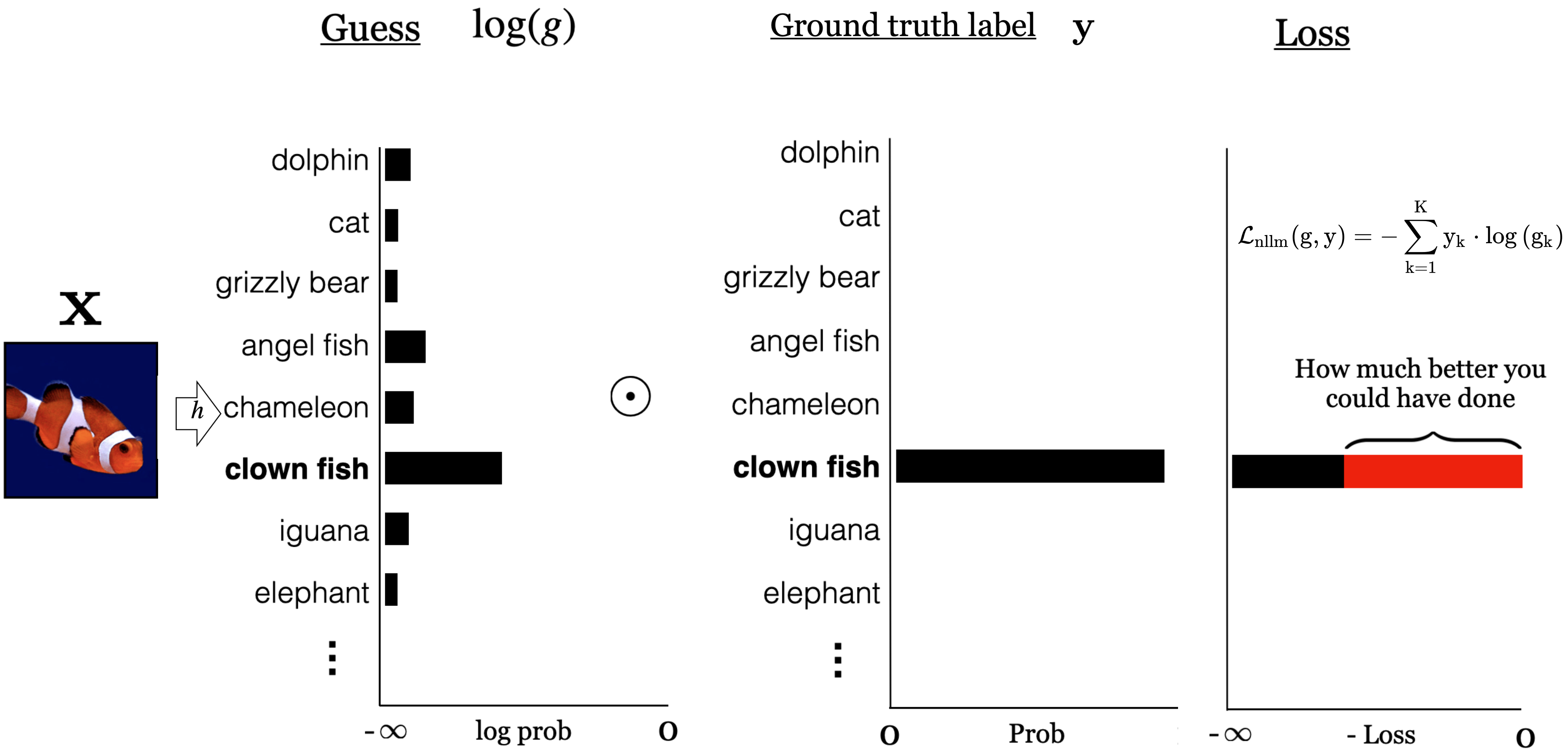
More complicated example: predict **one** class out of $K$ possibilities

then last layer: $K$ nuerons, softmax activation

e.g., say $K = 5$ classes

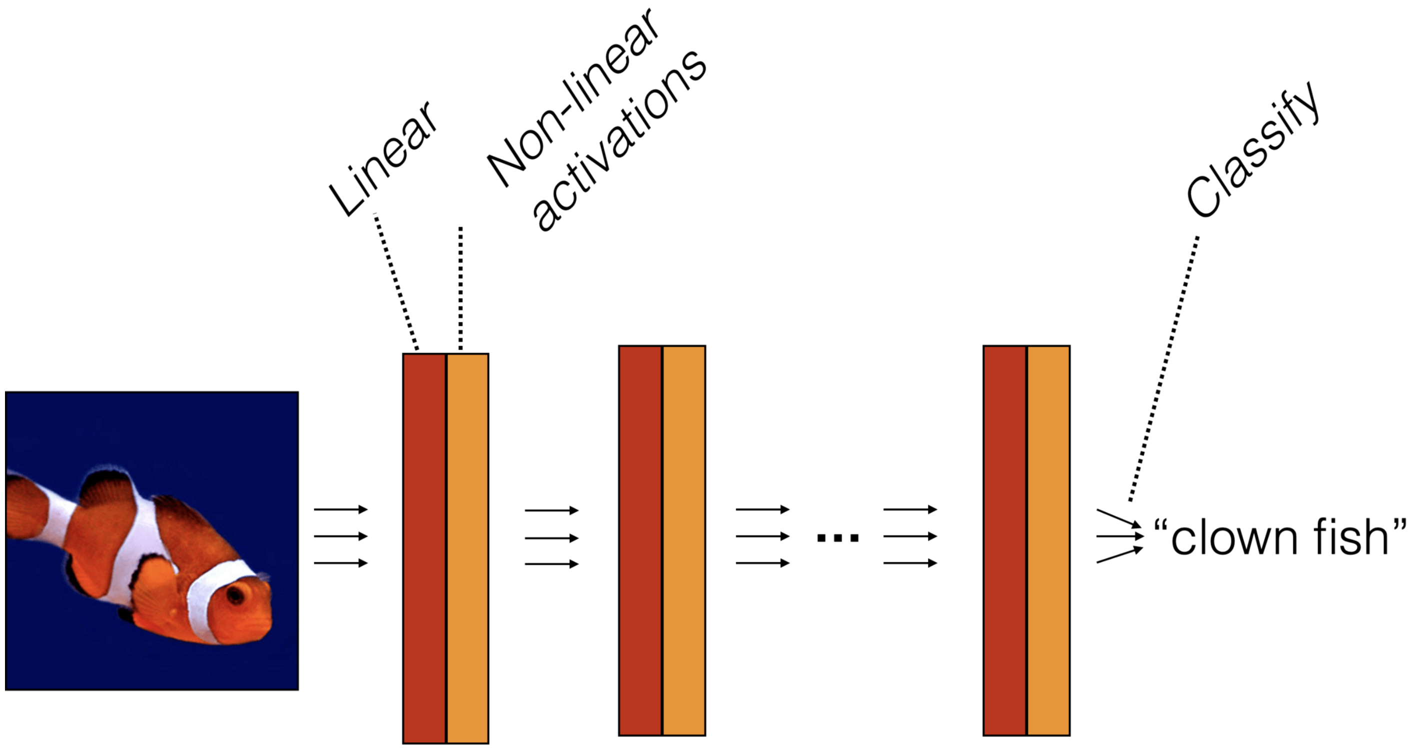



$$g = A^L = f^L(Z^L) = \text{softmax}(Z^L) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_K)/\sum_i \exp(z_i) \end{bmatrix}$$


Guess log(g)
Ground truth label y
Loss
x
h
dolphin
cat
grizzly bear
angel fish
chameleon
clown fish
iguana
elephant
⋮
- ∞
log prob
0
dolphin
cat
grizzly bear
angel fish
chameleon
clown fish
iguana
elephant
⋮
0
Prob
$\mathcal{L}_{\mathrm{nllm}}(\mathrm{g}, \mathrm{y}) = -\sum_{\mathrm{k}=1}^{\mathrm{K}} \mathrm{y}_{\mathrm{k}} \cdot \log(\mathrm{g}_{\mathrm{k}})$
How much better you could have done
- ∞
- Loss
0

# Outline

- Recap and neural networks motivation
- Neural Networks
  - A single neuron
  - A single layer
  - Many layers
  - Design choices (activation functions, loss functions choices)
- Forward pass
- Backward pass (back-propogation)

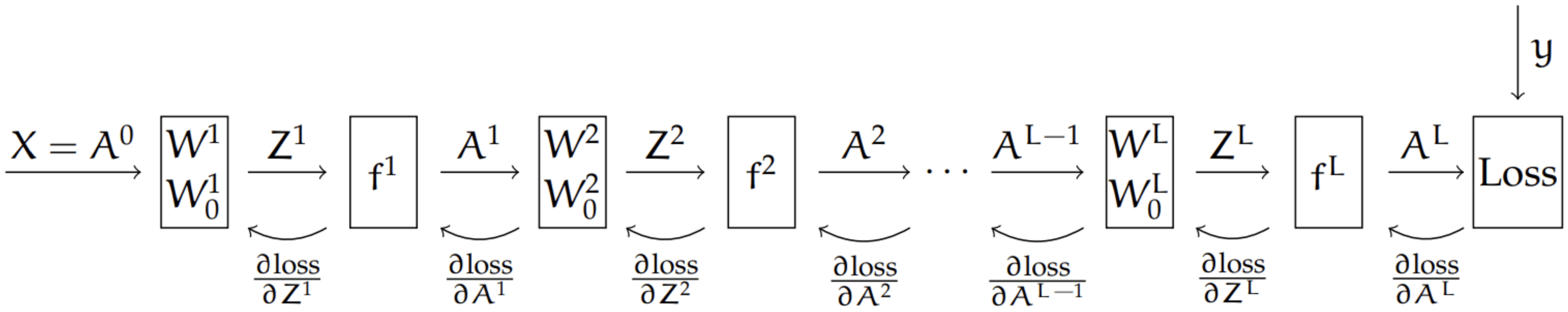



$$f_L\left(\ldots f_2\left(f_1\left(\mathbf{x}^{(i)}, \mathbf{W}_1\right), \mathbf{W}_2\right), \ldots \mathbf{W}_L\right)$$

How do we optimize

$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}\left(f_L\left(\ldots f_2\left(f_1\left(\mathbf{x}^{(i)}, \mathbf{W}_1\right), \mathbf{W}_2\right), \ldots \mathbf{W}_L\right), \mathbf{y}^{(i)}\right)$ though?

Forward propagation to obtain the output (model's guess)




Backpropagation to obtain gradients with respect to the loss

# Backprop = gradient descent & the chain rule

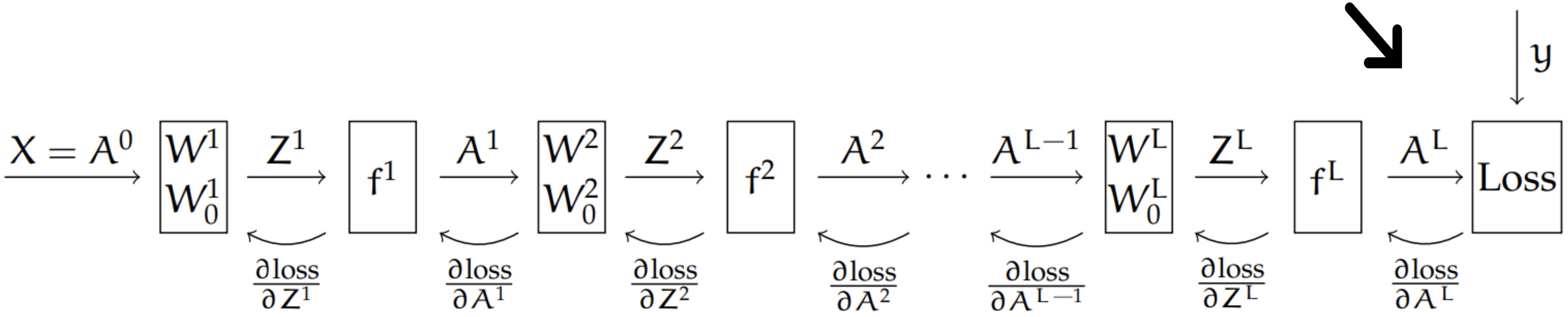
Recall that, the chain rule says:

For the composed function: $h(\mathbf{x}) = f(g(\mathbf{x}))$, its derivative is: $h'(\mathbf{x}) = f'(g(\mathbf{x}))g'(\mathbf{x})$

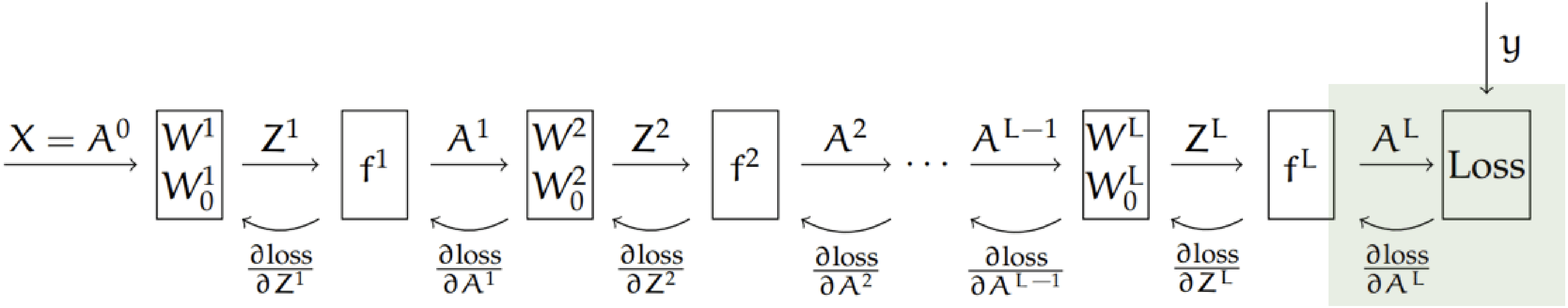
---

Here, our loss depends on the final output,

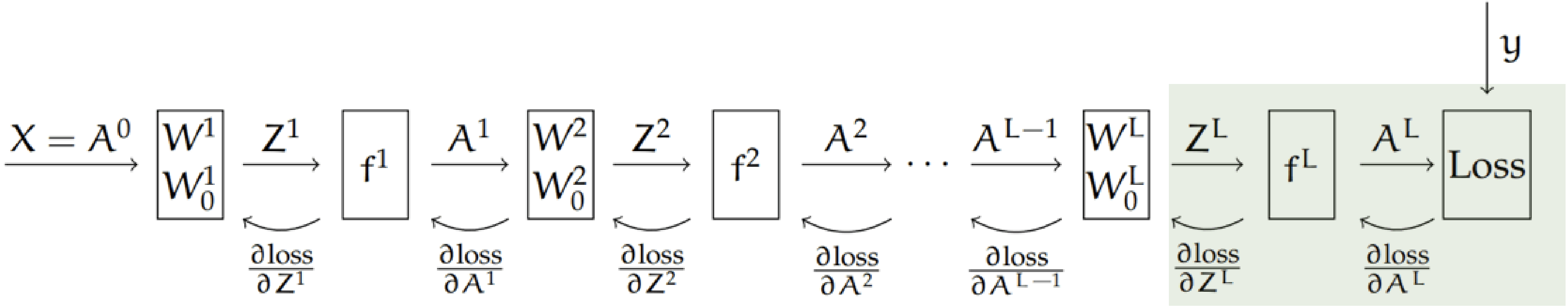
and the final output $A^L$ comes from a chain of composition of functions



# Backprop = gradient descent & the chain rule



# Backprop = gradient descent & the chain rule



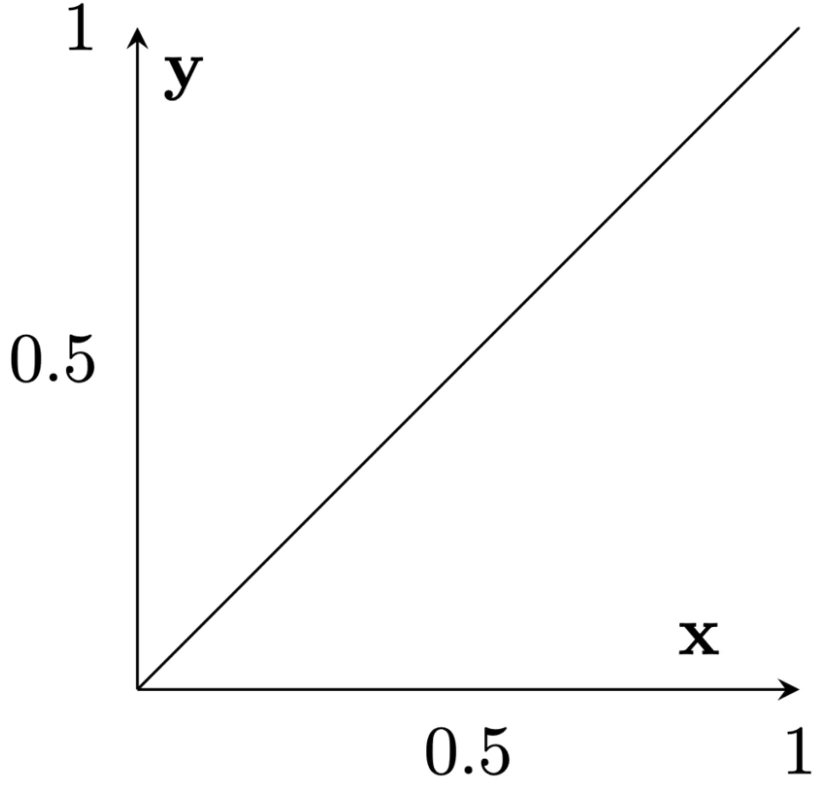

$$\underset{n^\ell \mathrm{x}1}{\frac{\partial \text{loss}}{\partial Z^{(\ell)}}} = \underset{n^\ell \mathrm{x} n^\ell}{\frac{\partial A^{(\ell)}}{\partial Z^{(\ell)}}} \underset{n^\ell \mathrm{x} n^{\ell+1}}{\frac{\partial Z^{(\ell+1)}}{\partial A^{(\ell)}}} \underset{n^{\ell+1} \mathrm{x} n^{\ell+1}}{\frac{\partial A^{(\ell+1)}}{\partial Z^{(\ell+1)}}} \cdots \underset{n^{L-1} \mathrm{x} n^{L-1}}{\frac{\partial A^{(L-1)}}{\partial Z^{(L-1)}}} \underset{n^{L-1} \mathrm{x} n^{L}}{\frac{\partial Z^{(L)}}{\partial A^{(L-1)}}} \underset{n^{L} \mathrm{x} n^{L}}{\frac{\partial A^{(L)}}{\partial Z^{(L)}}} \underset{n^{L} \mathrm{x}1}{\frac{\partial \text{loss}}{\partial A^{(L)}}}$$

(

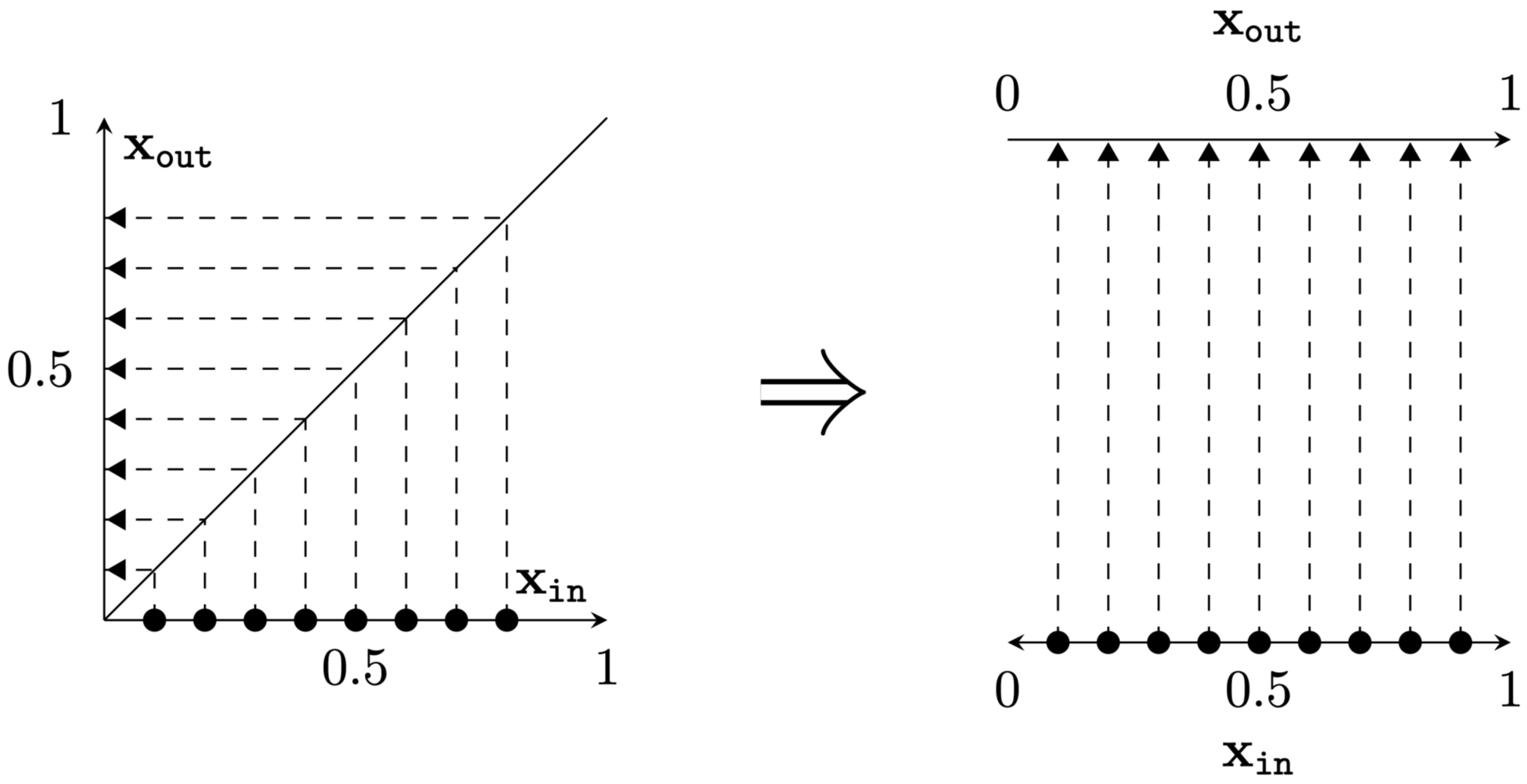
(The demo won't embed in PDF. But the direct link below works.)

https://playground.tensorflow.org/

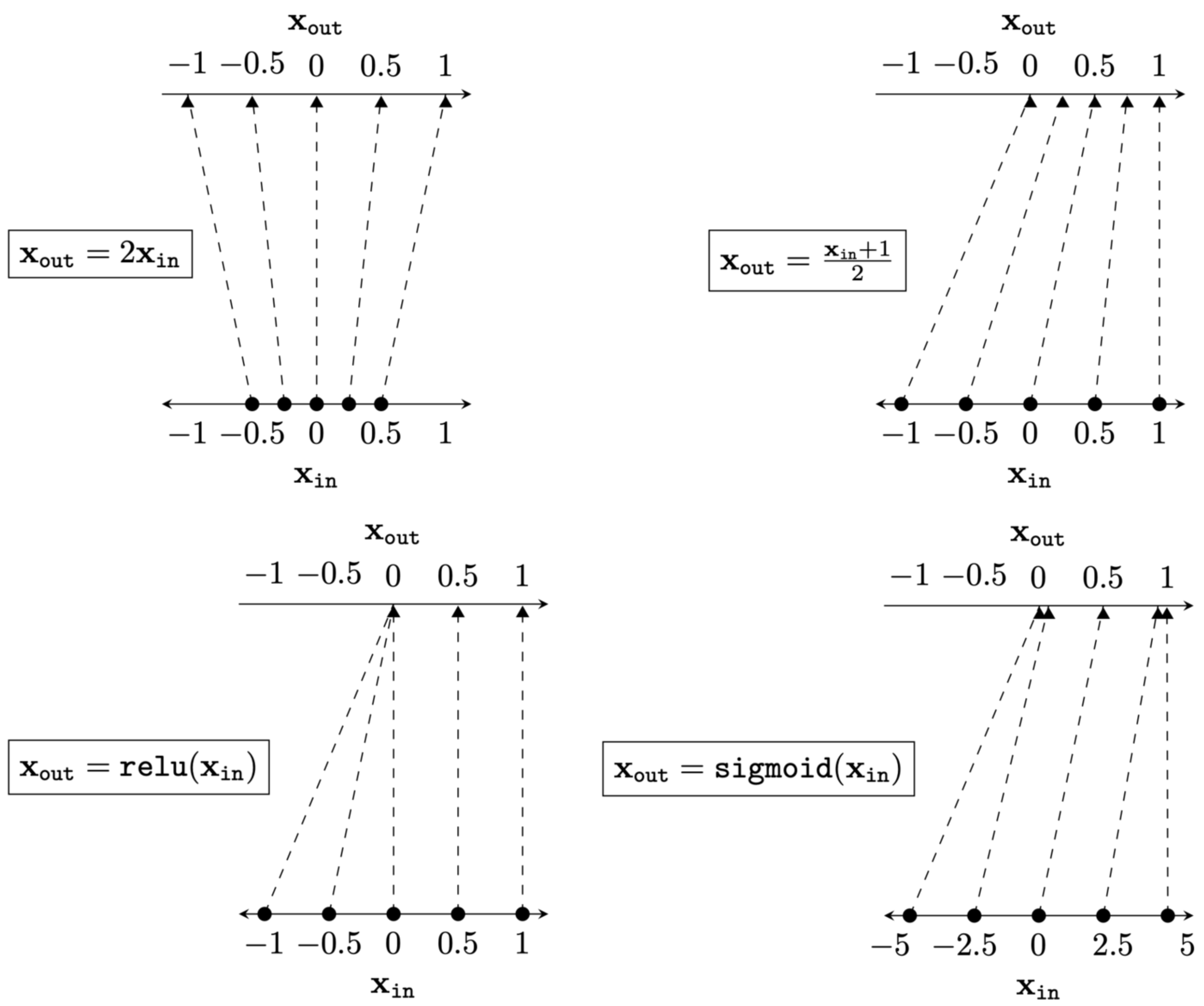
# Two different ways to represent a function

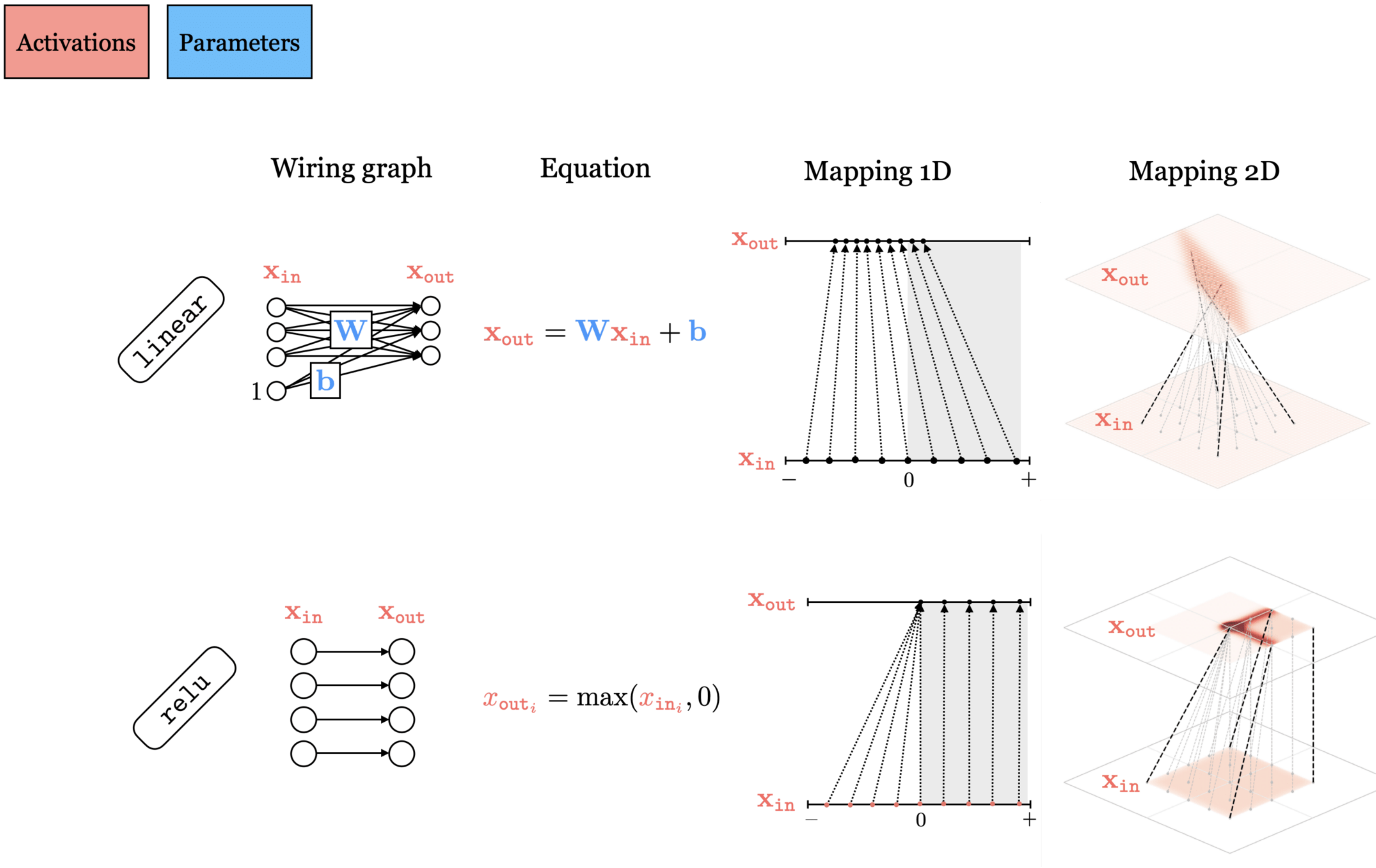


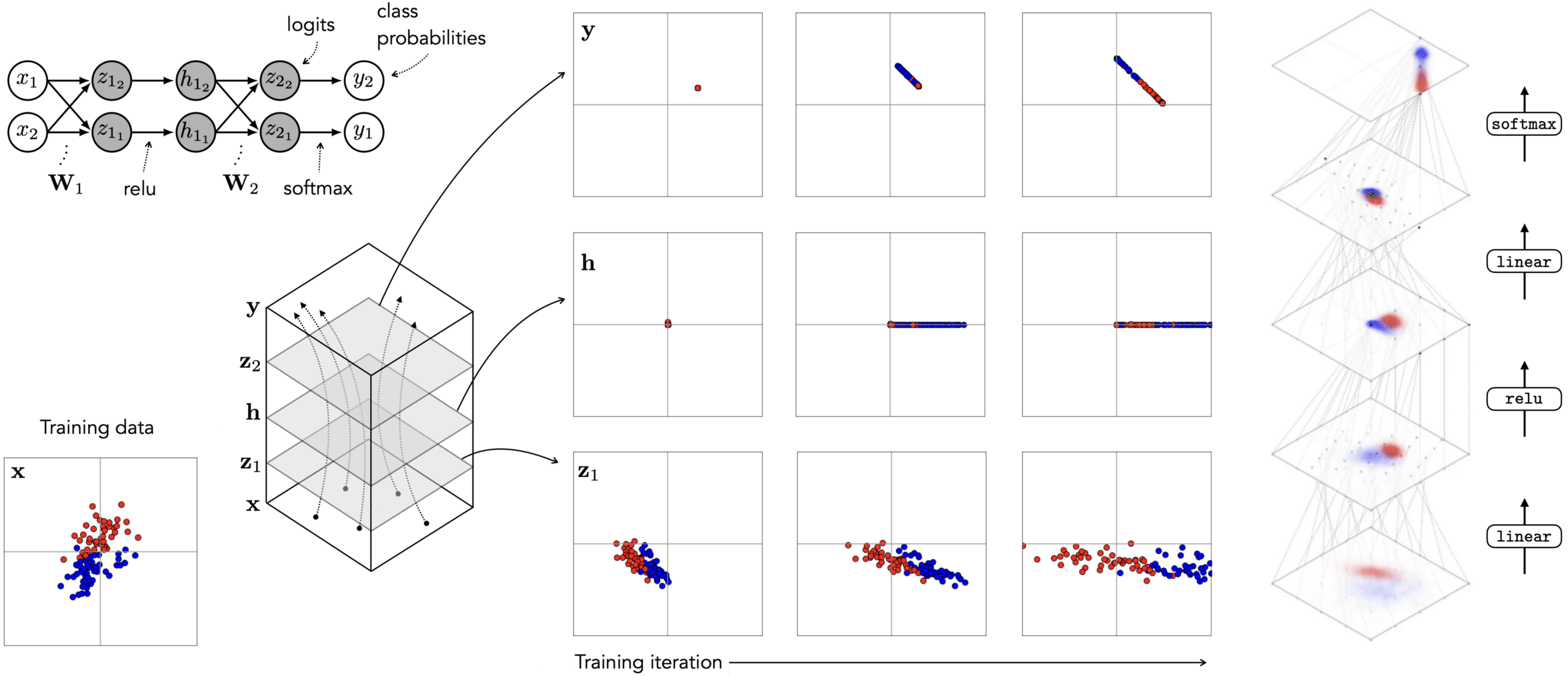
# Two different ways to represent a function



# Data transformations for a variety of neural net layers

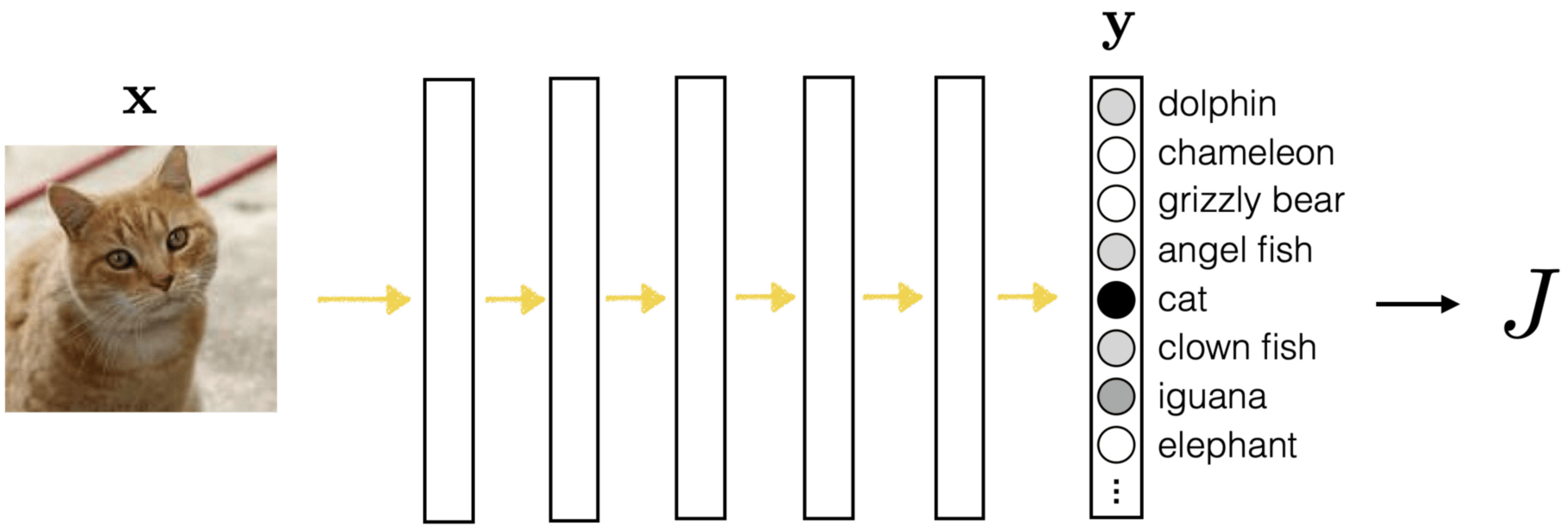



Activations
Parameters
Wiring graph
Equation
Mapping 1D
Mapping 2D
linear
$\mathbf{x}_{\texttt{in}}$
$\mathbf{x}_{\texttt{out}}$
$\mathbf{W}$
$\mathbf{b}$
1
$\mathbf{x}_{\texttt{out}} = \mathbf{W}\mathbf{x}_{\texttt{in}} + \mathbf{b}$
−
0
+
relu
$x_{\texttt{out}_i} = \max(x_{\texttt{in}_i}, 0)$



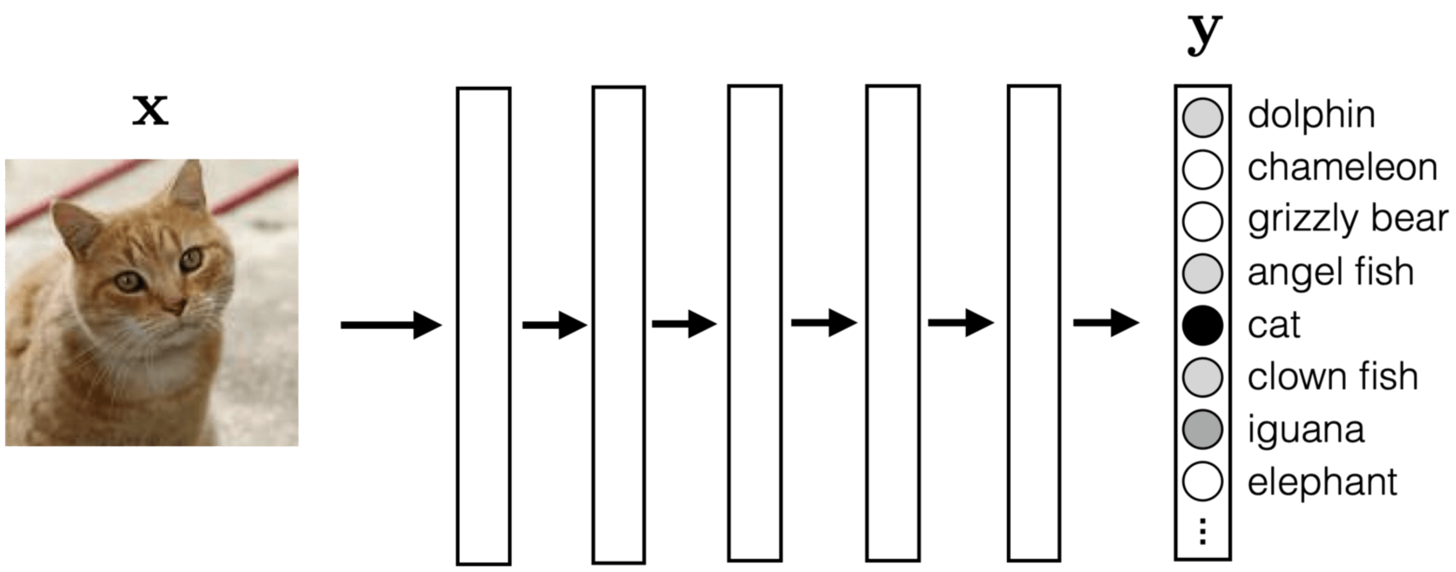
logits
class probabilities
$x_1$
$x_2$
$z_{1_2}$
$z_{1_1}$
$h_{1_2}$
$h_{1_1}$
$z_{2_2}$
$z_{2_1}$
$y_2$
$y_1$
$\mathbf{W}_1$
relu
$\mathbf{W}_2$
softmax
Training data
$\mathbf{x}$
$\mathbf{y}$
$\mathbf{z}_2$
$\mathbf{h}$
$\mathbf{z}_1$
$\mathbf{x}$
$\mathbf{y}$
$\mathbf{h}$
$\mathbf{z}_1$
Training iteration
softmax
linear
relu
linear

# Optimizing parameters versus optimizing inputs



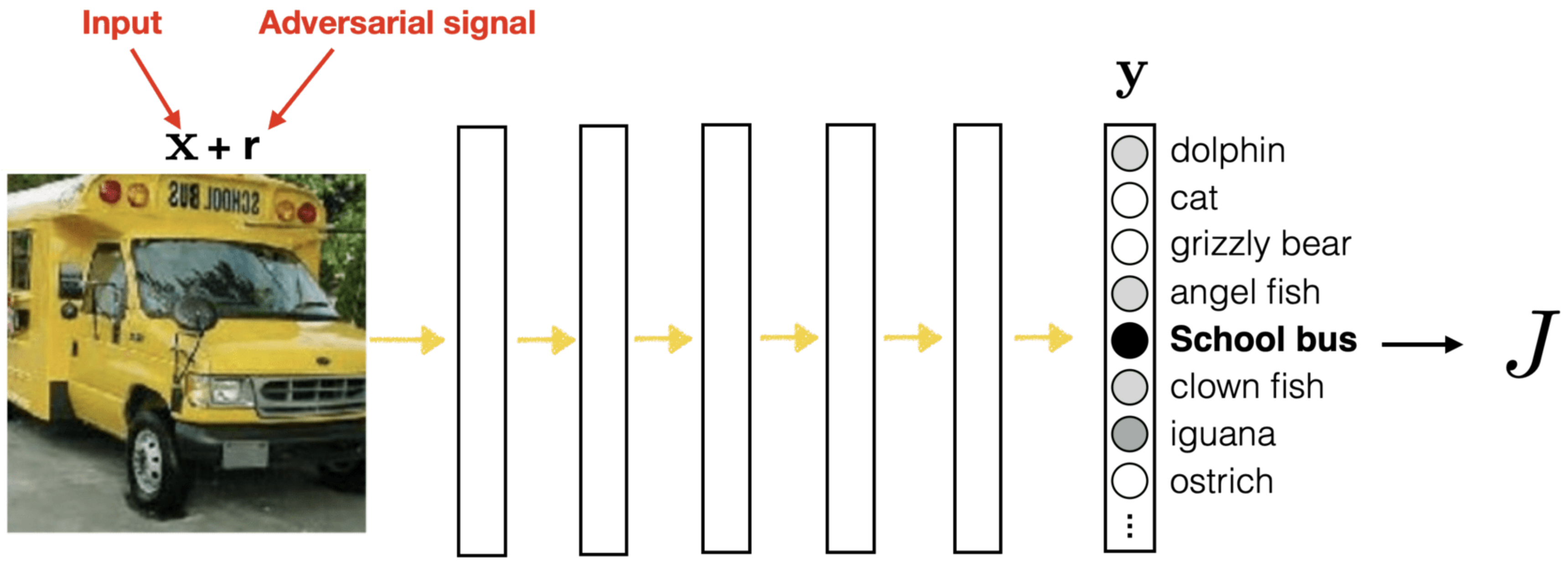

$$\frac{\partial J}{\partial \theta} \leftarrow$$ How much the total cost is increased or decreased by changing the parameters.

# Optimizing parameters versus optimizing inputs




$$\frac{\partial y_j}{\partial \mathbf{x}} \leftarrow$$ How much the “cat” score is increased or decreased by changing the image pixels.

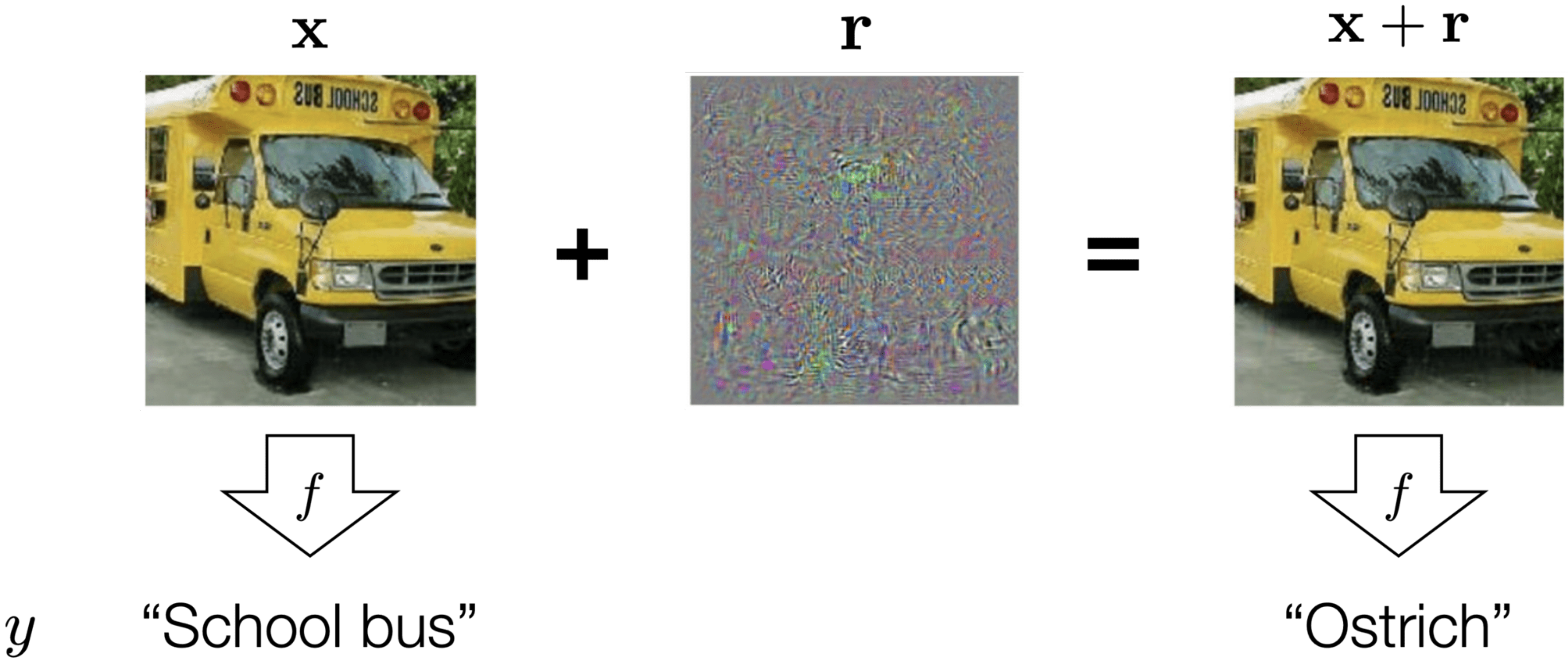
# Adversarial attacks




$$\frac{\partial y_j}{\partial r} \leftarrow$$ What adversarial signal r should we add to change the output label?

[“Intriguing properties of neural networks”, Szegedy et al. 2014]

# Adversarial attacks




$$\arg\max_{\mathbf{r}} p(y = \texttt{ostrich} | \mathbf{x} + \mathbf{r}) \quad \text{subject to} \quad \|\mathbf{r}\| < \epsilon$$

[“Intriguing properties of neural networks”, Szegedy et al. 2014]

)

# Summary

- We saw last week that introducing non-linear transformations of the inputs can substantially increase the power of linear regression and classification hypotheses.
- We also saw that it's kind of difficult to select a good transformation by hand.
- Multi-layer neural networks are a way to make (S)GD find good transformations for us!
- Fundamental idea is easy: specify a hypothesis class and loss function so that d Loss / d theta is well behaved, then do gradient descent.
- Standard feed-forward NNs (sometimes called multi-layer perceptrons which is actually kind of wrong) are organized into layers that alternate between parametrized linear transformations and fixed non-linear transforms (but many other designs are possible!)
- Typical non-linearities include sigmoid, tanh, relu, but mostly people use relu
- Typical output transformations for classification are as we have seen: sigmoid and/or softmax
- There's a systematic way to compute d Loss / d theta via backpropagation

We'd love it for you to share some lecture feedback.

# Thanks!