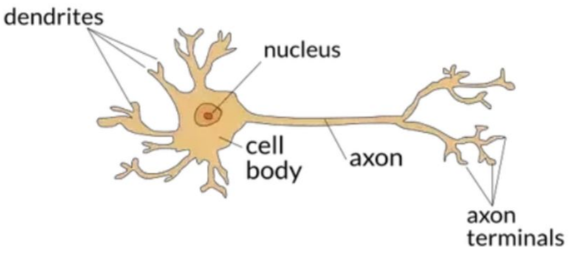
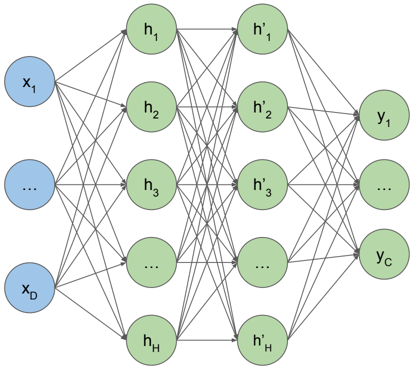
NYU CS-GY 6923
Machine Learning

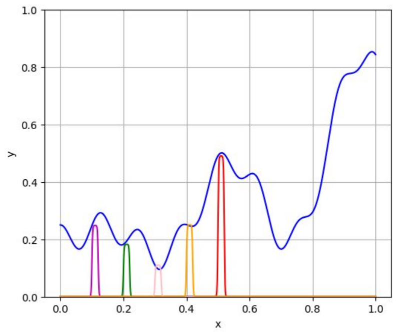
Prof. Pavel Izmailov

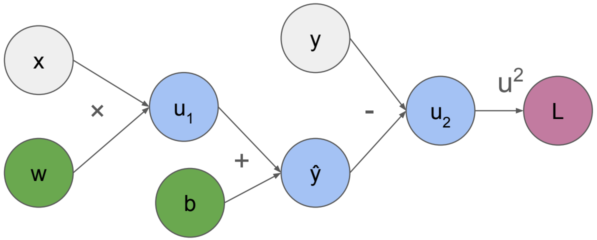
Today

- Neural Networks
- Universal Approximation
- Backpropagation
- Autograd
- Demo

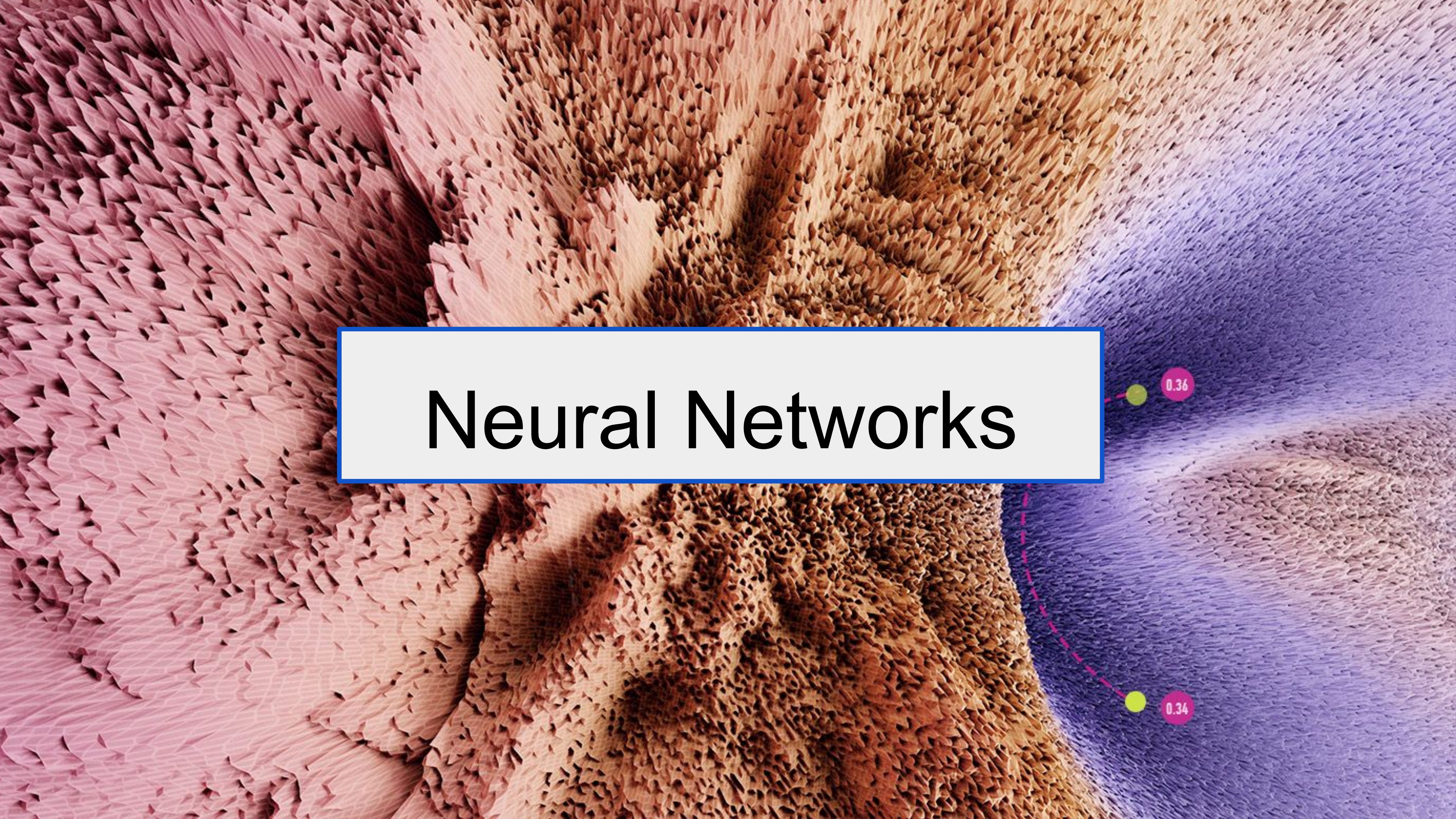




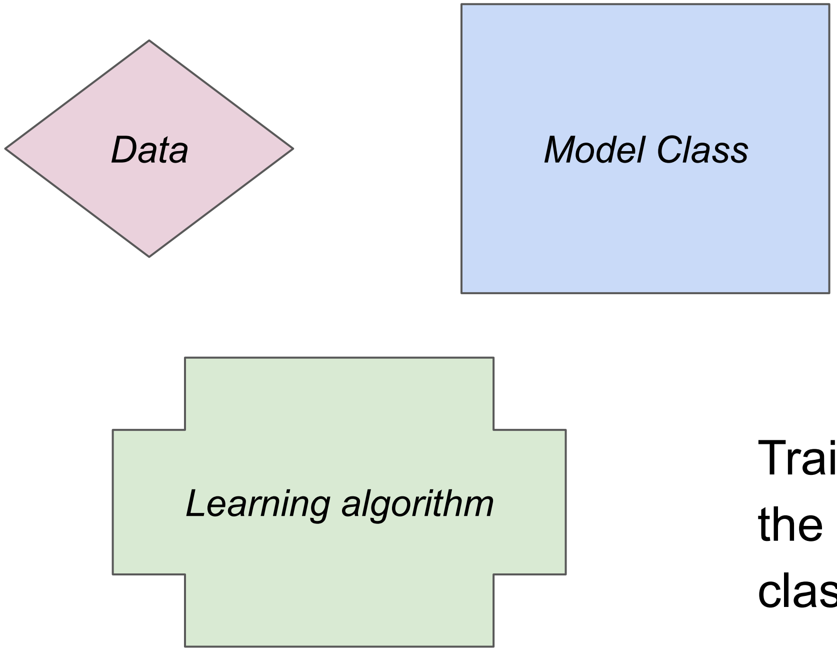




Neural Networks



General Picture of ML

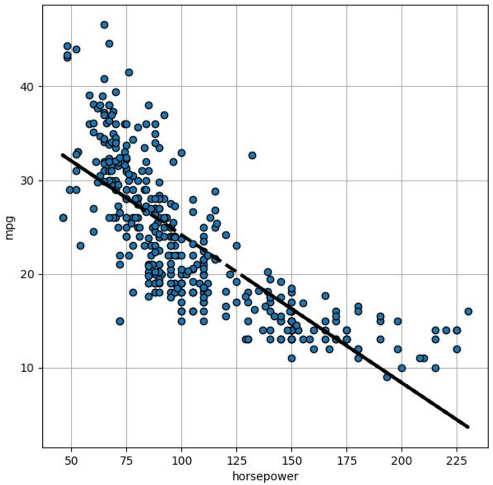


Training algorithm selects the model from a model class using data.

Example: Linear Regression

Data

Model Class:
Ridge Regression



Learning algorithm:
Exact Formula

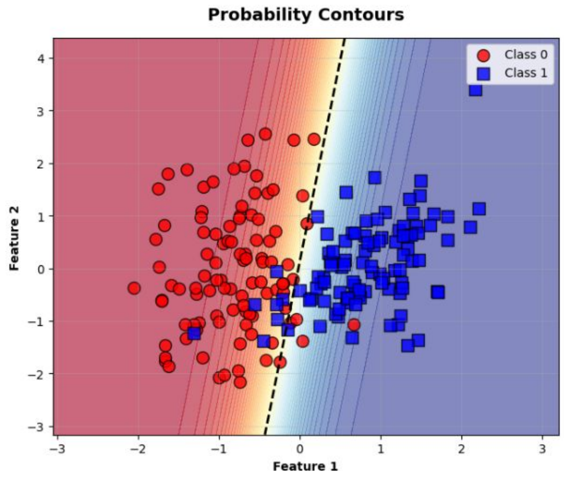
Training algorithm selects the model from a model class using data.

Example: Logistic Regression

Data

Model Class:
Logistic Regression

Learning algorithm:
Gradient Descent

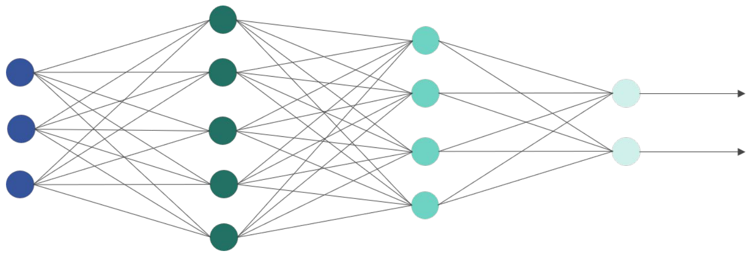


Training algorithm selects the model from a model class using data.

GD-Based Learning

Data

Model Class:
Any Model Differentiable wrt Parameters



Learning algorithm:
Gradient Descent

Training algorithm selects the model from a model class using data.

Gradient Descent is an extremely powerful learning algorithm

GD-Based Learning

Gradient-based learning:

- Define a model: $M(X_i, w)$ producing predictions based on data
  - w represents parameters of the model
  - $M(X_i, w)$ typically needs to be differentiable with respect to *w*
- Define a loss function $L(w)$
  - Typically, $L(w) = \sum_i L(M(X_i, w), y_i)$
- Minimize using gradient descent to find optimal w

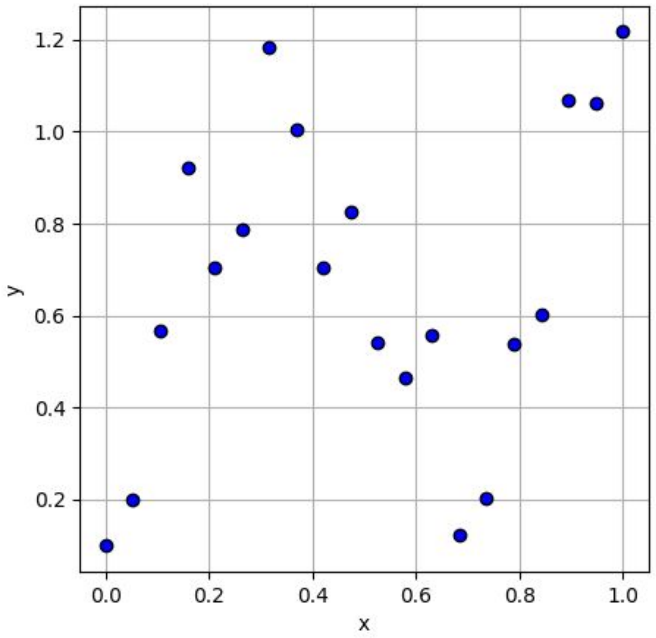
GD-Based Learning

Gradient-based learning:

- Define a model: $M(X_i, w)$ producing predictions based on data
  - w represents parameters of the model
  - $M(X_i, w)$ typically needs to be differentiable with respect to *w*
- Define a loss function $L(w)$
  - Typically, $L(w) = \sum_i L(M(X_i, w), y_i)$
- Minimize using gradient descent to find optimal w

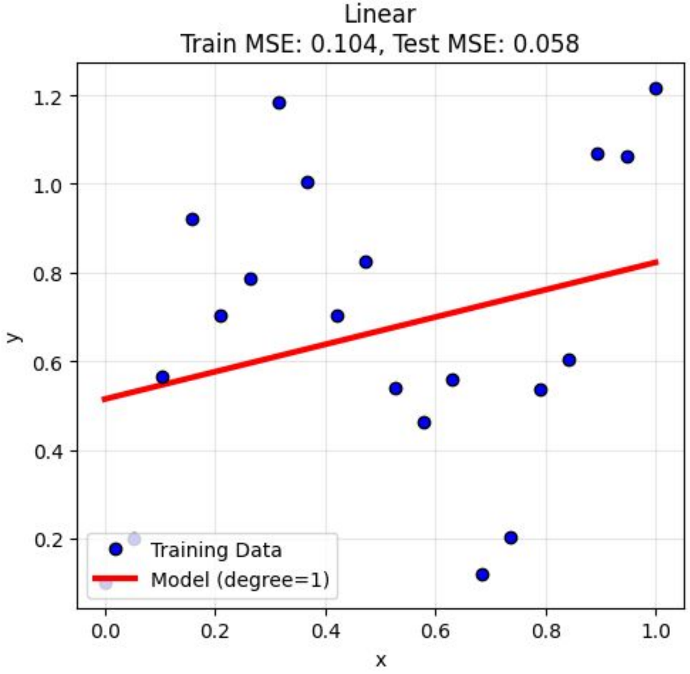
Neural Networks: Motivation

- Non-linear data



Neural Networks: Motivation

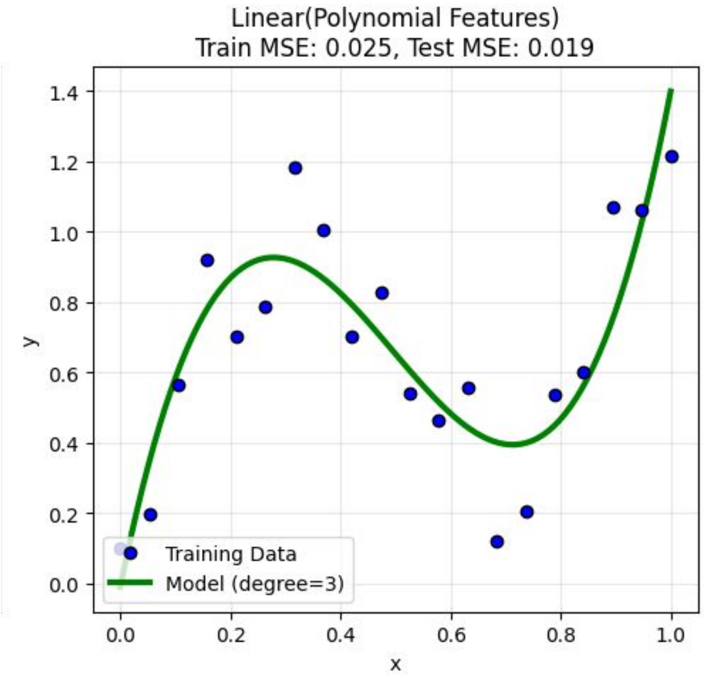
- Non-linear data
- Linear fit doesn’t work



Neural Networks: Motivation

- Non-linear data
- Linear fit doesn't work
- Old solution: use polynomial features

$$(1, x, x^2, \ldots, x^d)$$

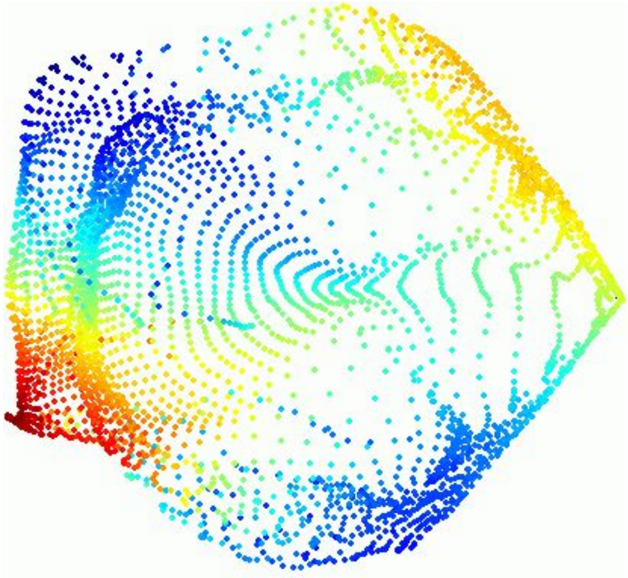


Neural Networks: Motivation

- Non-linear data
- Linear fit doesn’t work
- Old solution: use polynomial features

$$(1, x, x^2, \dots, x^d)$$

- But what if our data is very high-dimensional and we don’t know what features to use?



Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = W_1 x + b_1 \longleftarrow \text{Hidden features}$$

$$\hat{y} = W_2 h + b_2 \longleftarrow \text{Output}$$

$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

- Will this model be able to learn features?

Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = W_1 x + b_1 \longleftarrow \text{Hidden features}$$
$$\hat{y} = W_2 h + b_2 \longleftarrow \text{Output}$$
$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

- Will this model be able to learn features?
  - No, it is equivalent to a linear model!

$$W_2(W_1 x + b_1) + b_2 = (W_2 W_1)x + (b_2 + W_2 b_1) = W' x + b'$$

Feature Engineering → Feature Learning

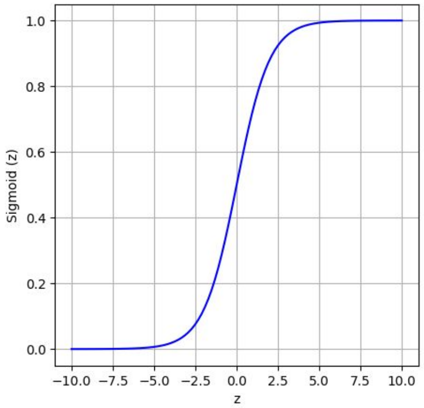
- Let's *learn* the features!

$$\cancel{h = W_1 x + b_1} \longleftarrow \text{Hidden features}$$

$$\hat{y} = W_2 h + b_2 \longleftarrow \text{Output}$$

$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

$$\boxed{h = \sigma(W_1 x + b_1)}$$



- Will this model be able to learn features?
  - No, it is equivalent to a linear model!

$$W_2(W_1 x + b_1) + b_2 = (W_2 W_1)x + (b_2 + W_2 b_1) = W' x + b'$$

- We need a non-linearity!

Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = \sigma(W_1 x + b_1) \longleftarrow \text{Hidden features}$$

$$\hat{y} = W_2 h + b_2 \longleftarrow \text{Output}$$

$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

- What loss can we use?

Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = \sigma(W_1 x + b_1)$$ Hidden features

$$\hat{y} = W_2 h + b_2$$ Output

$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

- What loss can we use? MSE

$$L(W_1, b_1, W_2, b_2) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = \sigma(W_1 x + b_1) \longleftarrow \text{Hidden features}$$

$$\hat{y} = W_2 h + b_2 \longleftarrow \text{Output}$$

$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

- What loss can we use? MSE

$$L(W_1, b_1, W_2, b_2) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- How many parameters?

- Train: minimize loss with Adam / SGD

Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = \sigma(W_1 x + b_1) \longleftarrow \text{Hidden features}$$
$$\hat{y} = W_2 h + b_2 \longleftarrow \text{Output}$$
$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$

- What loss can we use? MSE

$$L(W_1, b_1, W_2, b_2) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

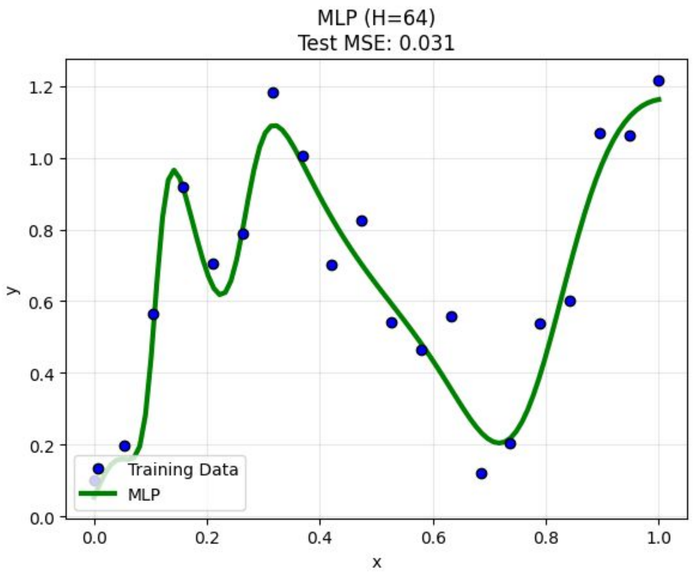
- How many parameters?
DH + H + H + 1

- Train: minimize loss with Adam / SGD

Feature Engineering → Feature Learning

- Let's *learn* the features!

$$h = \sigma(W_1 x + b_1) \quad \longleftarrow \text{Hidden features}$$
$$\hat{y} = W_2 h + b_2 \quad \longleftarrow \text{Output}$$
$$h \in \mathbb{R}^H, x \in \mathbb{R}^D, \hat{y} \in \mathbb{R}$$



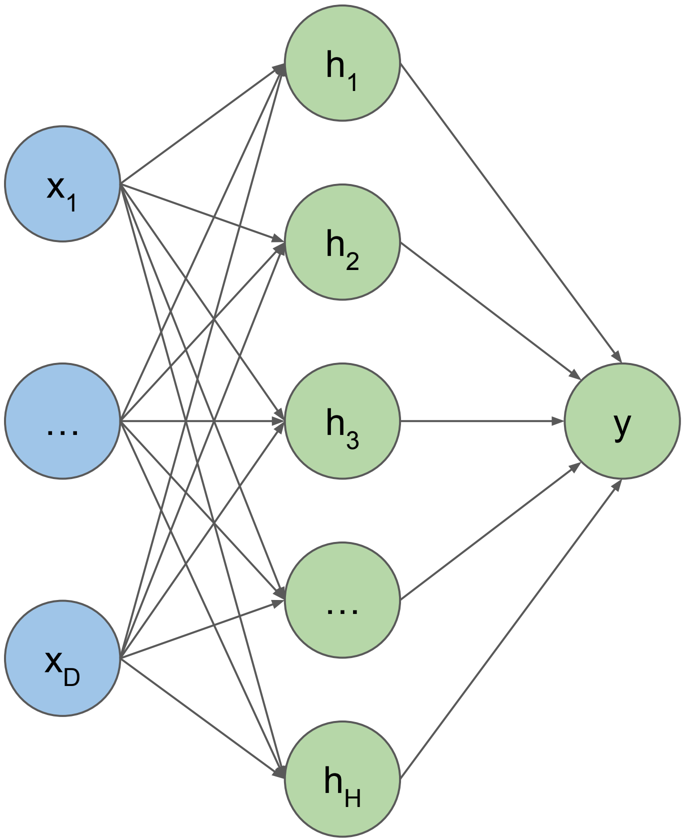
- What loss can we use? MSE

$$L(W_1, b_1, W_2, b_2) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

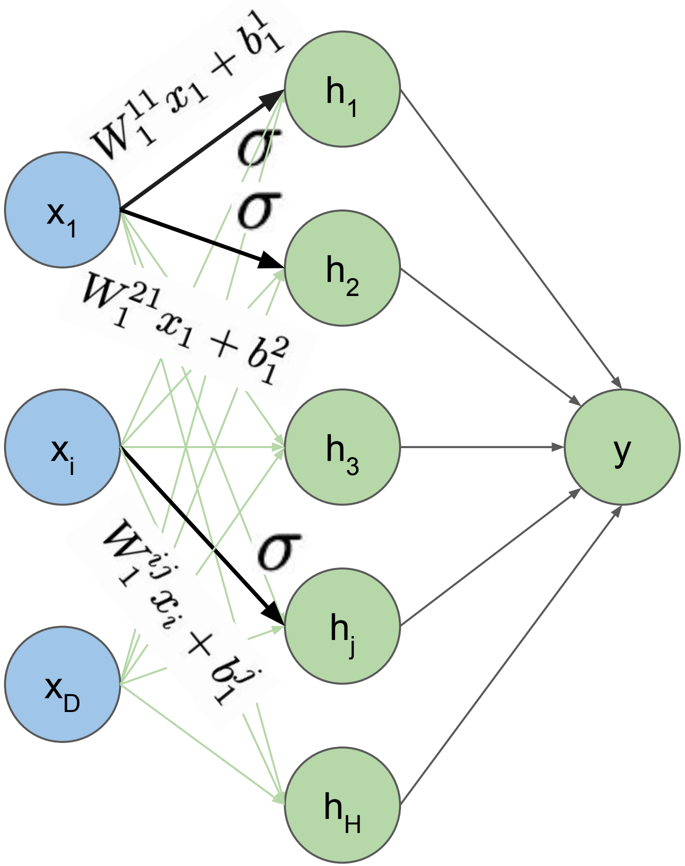
- How many parameters?
  DH + H + H + 1

- Train: minimize loss with Adam / SGD

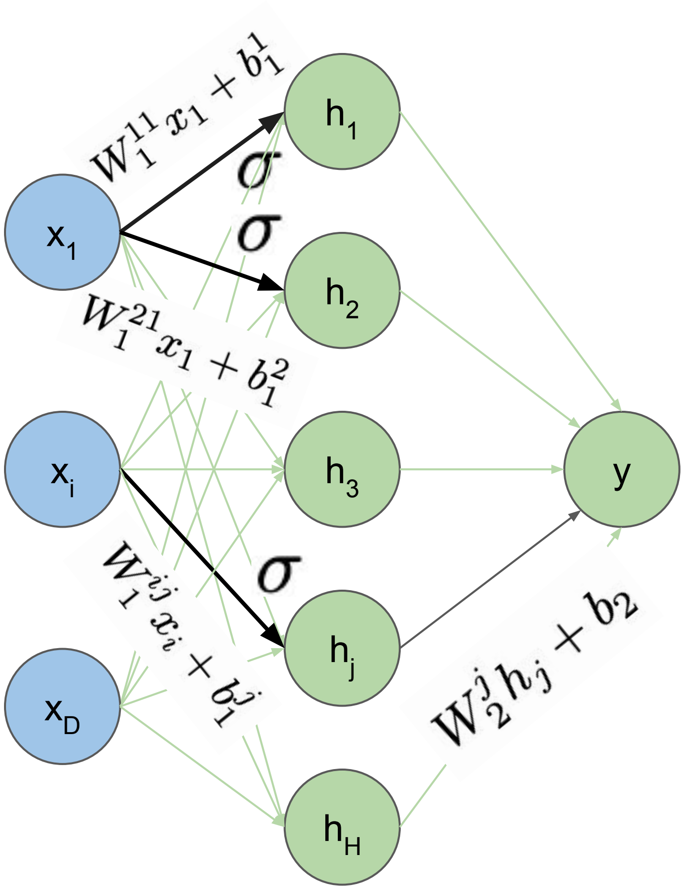
Neural Networks



Neural Networks



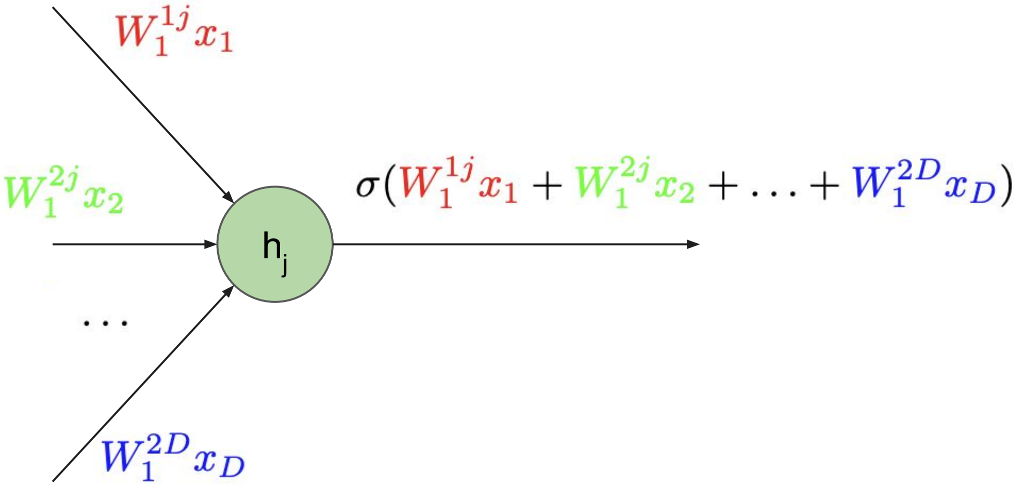
Neural Networks



$$\hat{y} = W_2\sigma(W_1x + b_1) + b_2$$

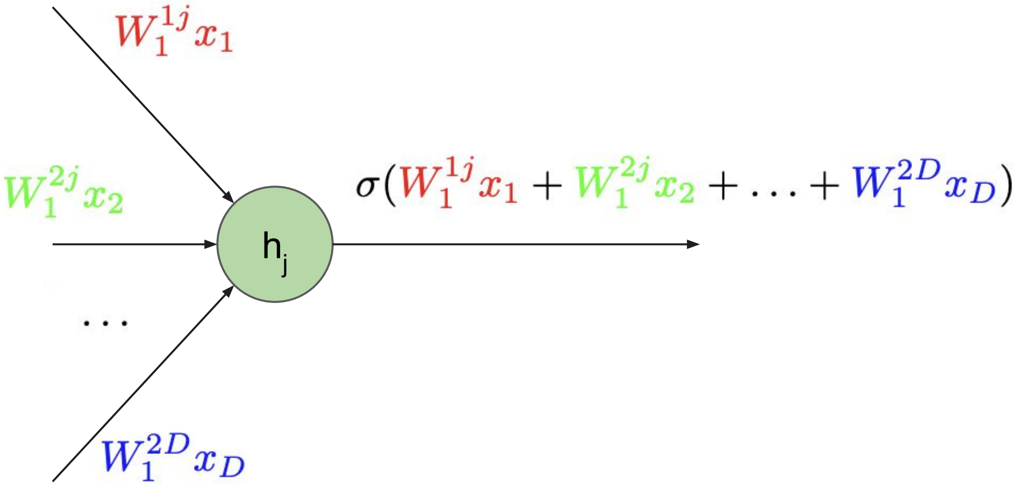
- Linear model on top of learned *hidden features **h***
- Hidden features are computed from input as a linear transform followed by element-wise non-linearity

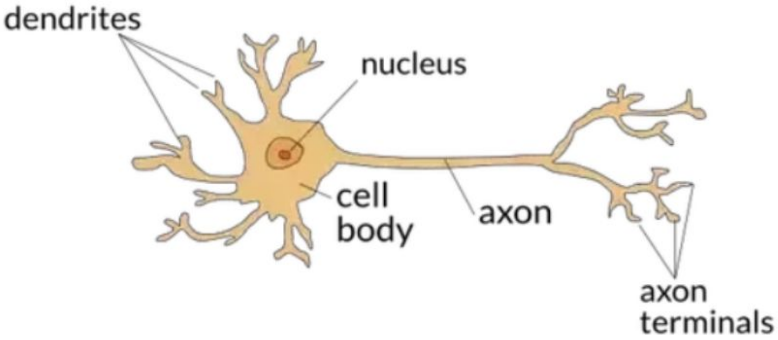
Neurons



- Each node in the network is called a neuron
- Neurons aggregate their inputs and output an *activation*

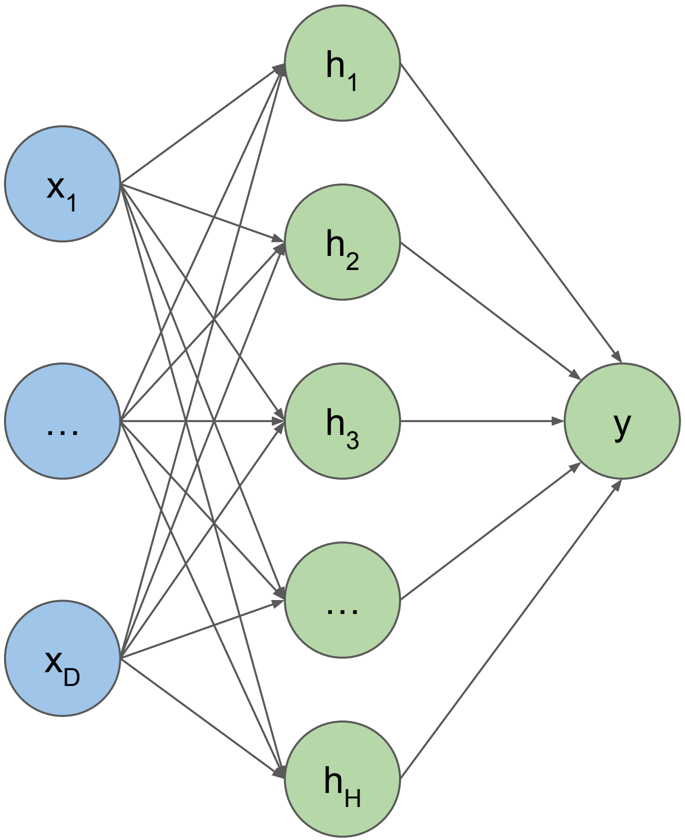
Neurons



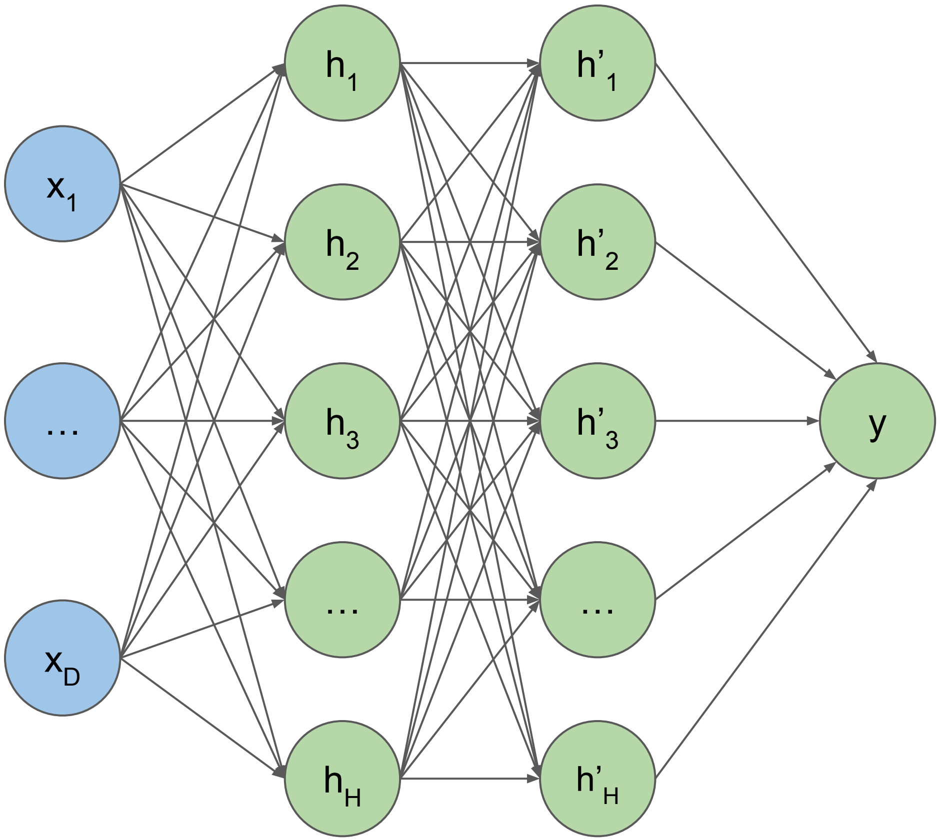


- Each node in the network is called a neuron
- Neurons aggregate their inputs and output an *activation*
- Rough analogy with a biological neuron

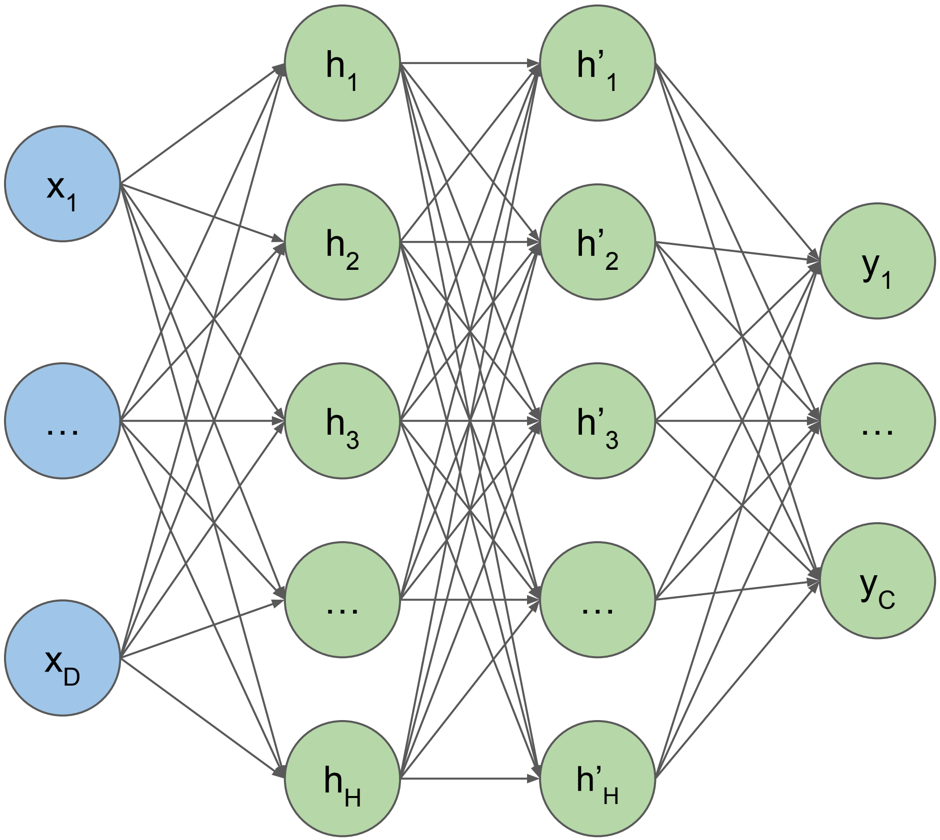
Neural Networks



Neural Networks: 2 hidden layers

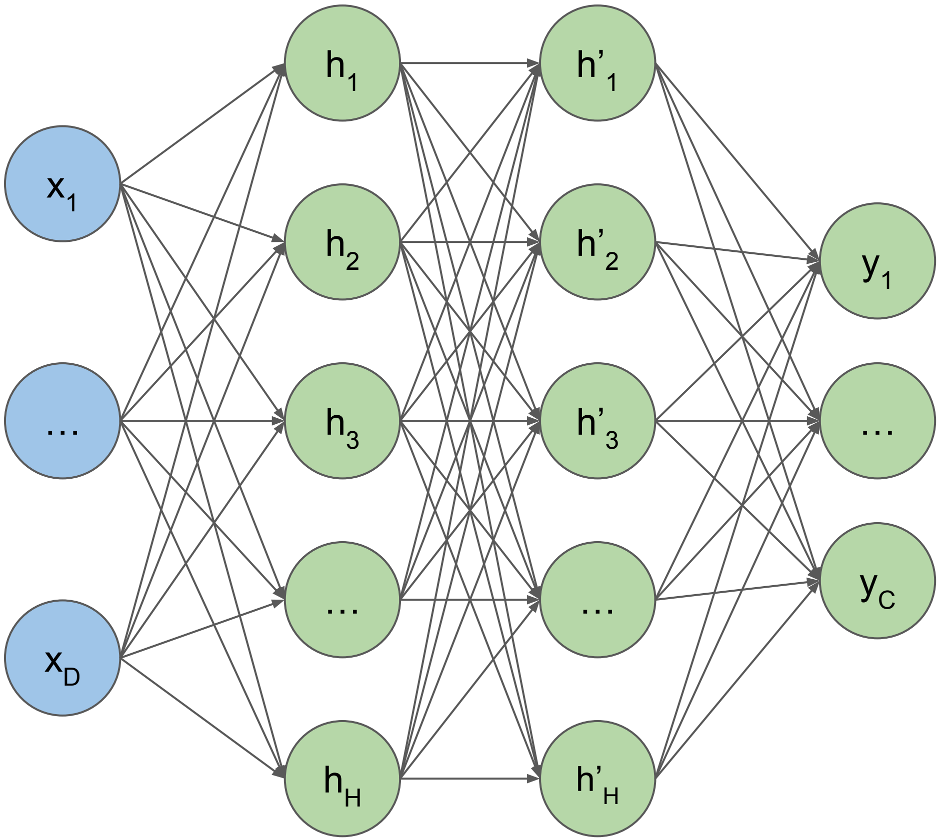


Neural Networks: multiple outputs



- How many parameters does this model have as a function of ***D, H, C***?

Neural Networks: multiple outputs



- How many parameters does this model have as a function of ***D, H, C***?

DH + H + H H + H + H C + C

Universal Approximation Theorem

See http://neuralnetworksanddeeplearning.com/, chapter 4

Universal Approximation Theorem

Universal approximation theorem—Let $C(X, \mathbb{R}^m)$ denote the set of continuous functions from a subset X of a Euclidean $\mathbb{R}^n$ space to a Euclidean space $\mathbb{R}^m$. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Note that $(\sigma \circ x)_i = \sigma(x_i)$, so $\sigma \circ x$ denotes σ applied to each component of x.

Then σ is not polynomial if and only if for every $n \in \mathbb{N}$, $m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n$, $f \in C(K, \mathbb{R}^m), \varepsilon > 0$ there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon$$

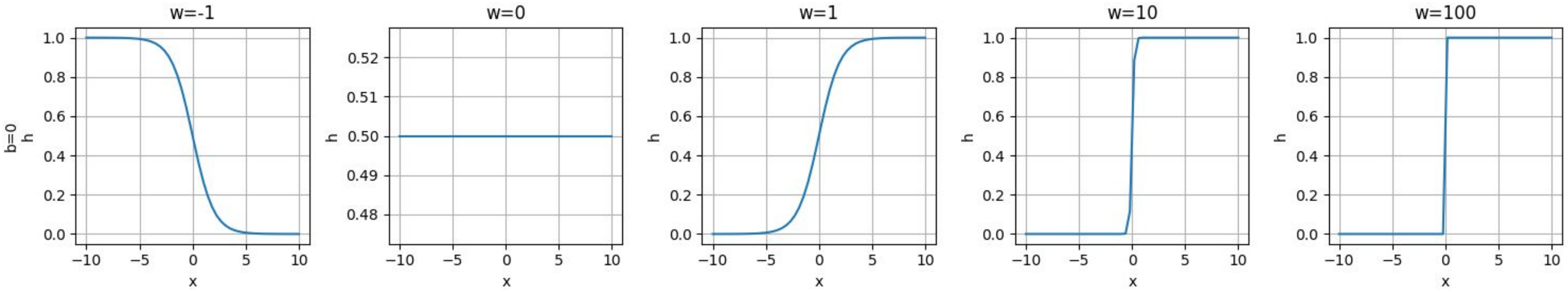
where $g(x) = C \cdot (\sigma \circ (A \cdot x + b))$

Informal version: a neural network with sufficient width with a single hidden layer can approximate any continuous function up to arbitrary precision.

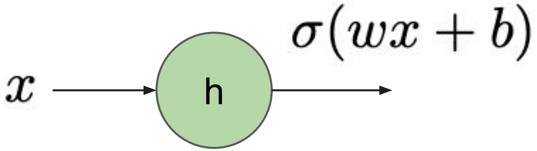
Universal Approximation Theorem

Intuitive proof sketch following Michael Nielsen's book:

$$h = \sigma(wx + b)$$



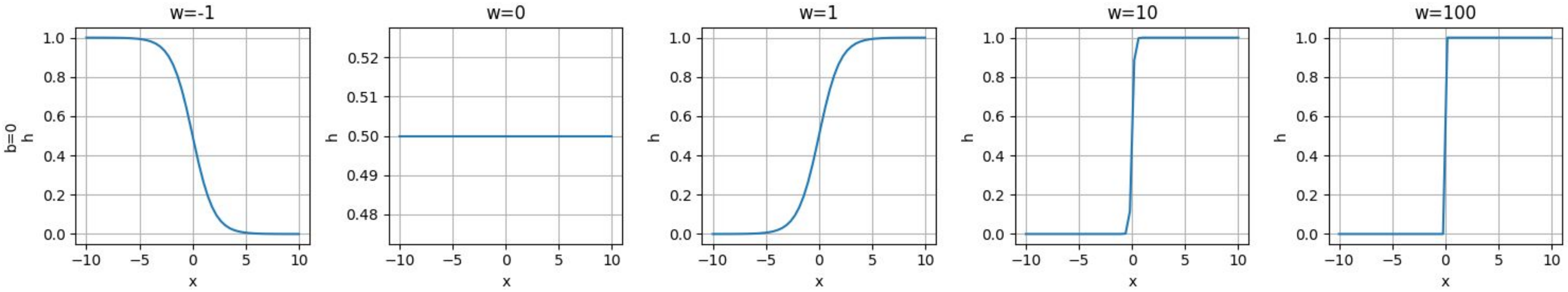
Let's look at the activations of a single neuron and change **w**



Universal Approximation Theorem

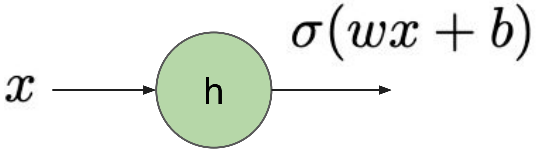
Intuitive proof sketch following Michael Nielsen's book:

$$h = \sigma(wx + b)$$



Let's look at the activations of a single neuron and change **w**

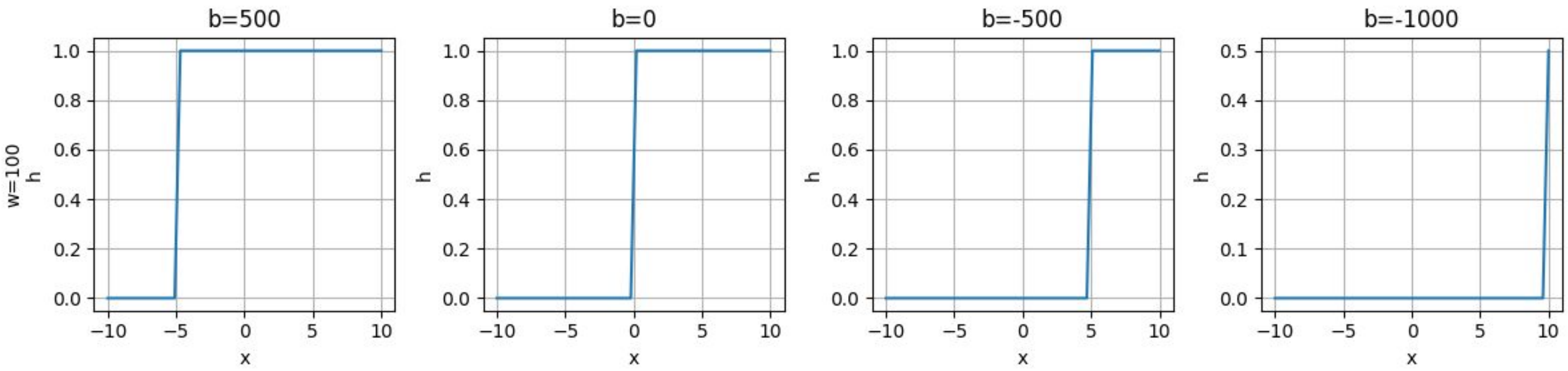
- Larger w ⇒ closer to step function



Universal Approximation Theorem

Intuitive proof sketch following Michael Nielsen's book:

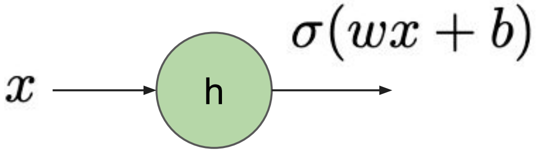
$$h = \sigma(wx + b)$$



Now, let's fix **w** and change **b**.

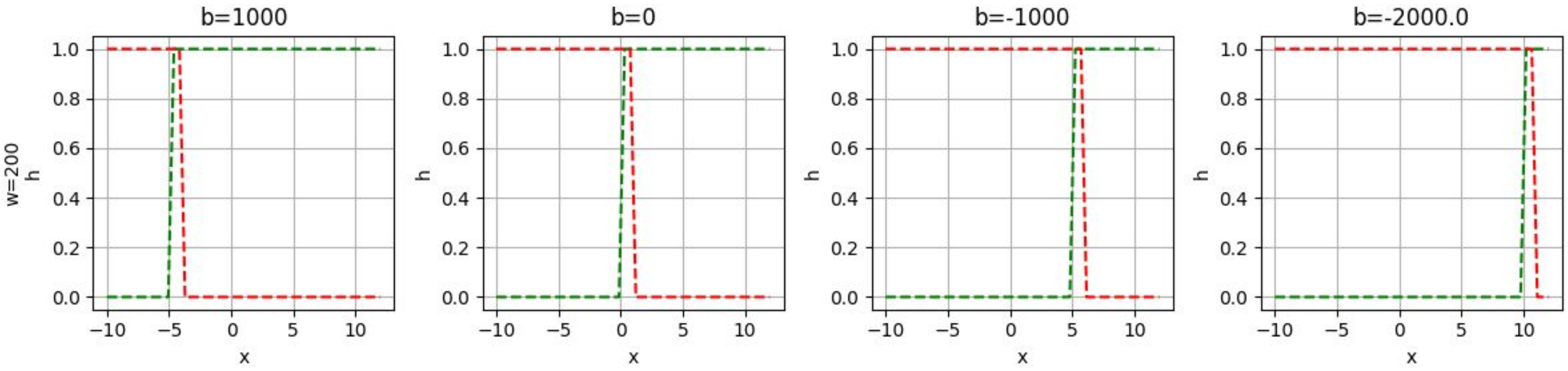
$$\sigma(w\hat{x} + b) = 0.5 \Leftrightarrow b = -w\hat{x}.$$

- **b** shifts the curve horizontally



Universal Approximation Theorem

Intuitive proof sketch following [Michael Nielsen's book]:



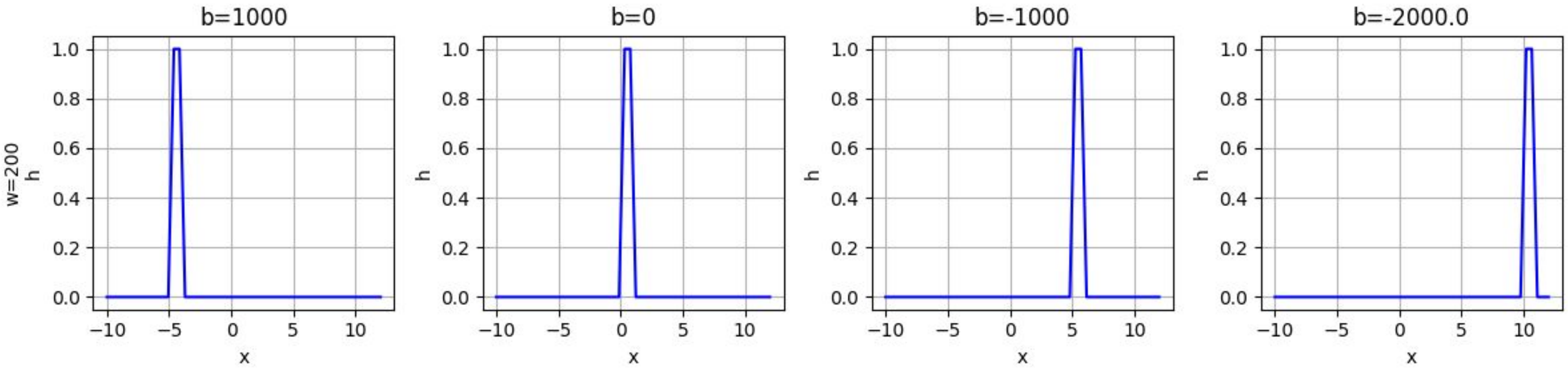
Now, let's put two of these together, but pointing in the opposite directions!

$$\sigma(wx - w\hat{x}) + \sigma(-wx + w(\hat{x} + \epsilon)) - 1$$

- We get at bump at a desired position!

Universal Approximation Theorem

Intuitive proof sketch following Michael Nielsen's book:



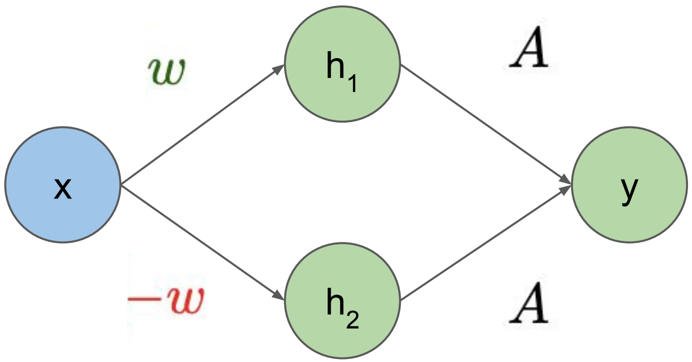
Now, let's put two of these together, but pointing in the opposite directions!

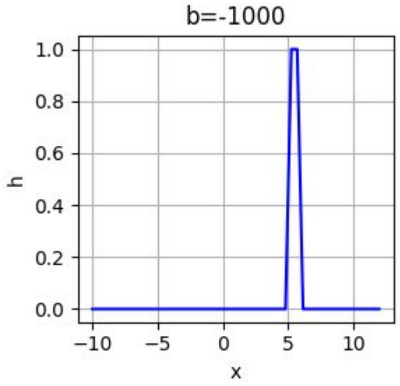
$$\sigma(wx - w\hat{x}) + \sigma(-wx + w(\hat{x} + \epsilon)) - 1$$

- We get at bump at a desired position!

Universal Approximation Theorem

Intuitive proof sketch following [Michael Nielsen's book](#):

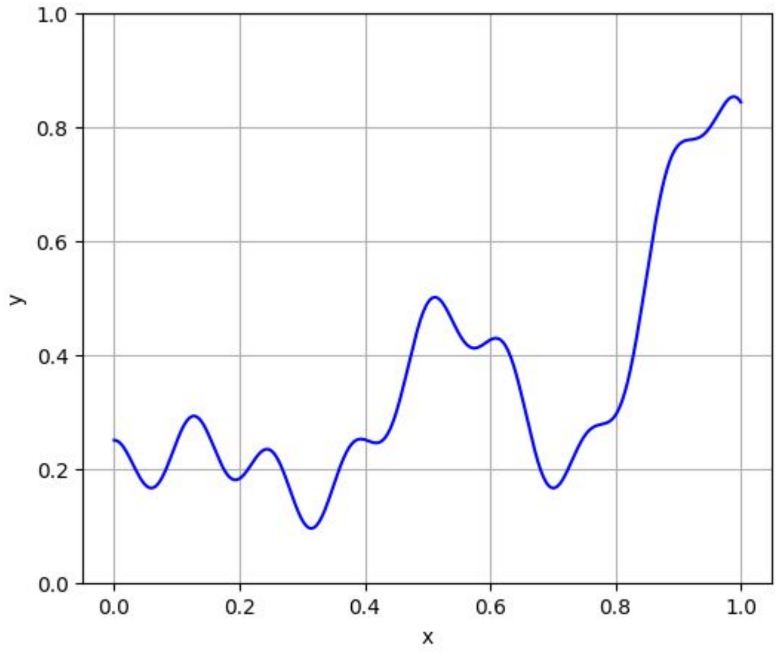




- We can create a bump with a neural net with two hidden units

$$A \cdot (\sigma(wx - w\hat{x}) + \sigma(-wx + w(\hat{x} + \epsilon)) - 1)$$

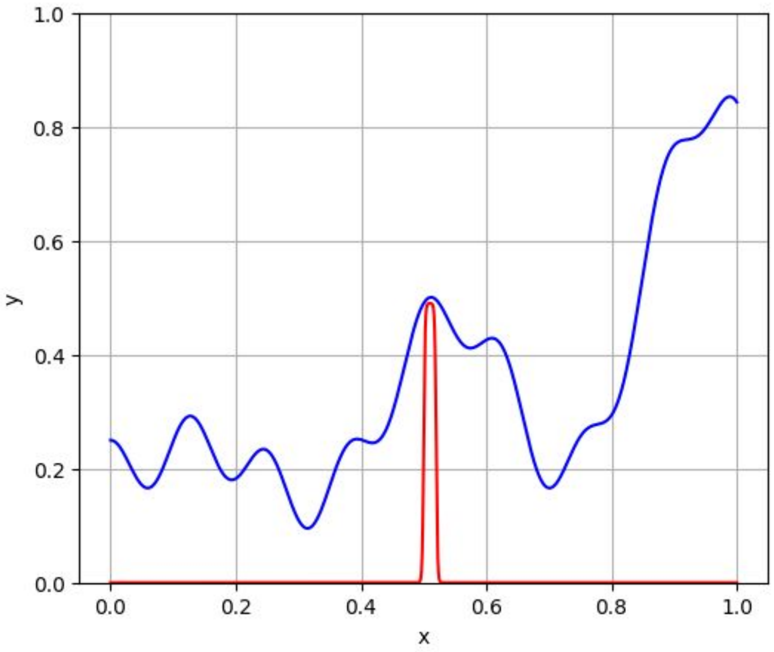
Universal Approximation Theorem



Let's approximate a function *f* with a collection of bumps.

$$\sum_{j=1}^{M} f(\hat{x}_j) \cdot (\sigma(wx - w\hat{x}_j) + \sigma(-wx + w(\hat{x}_j + \epsilon)) - 1)$$

Universal Approximation Theorem

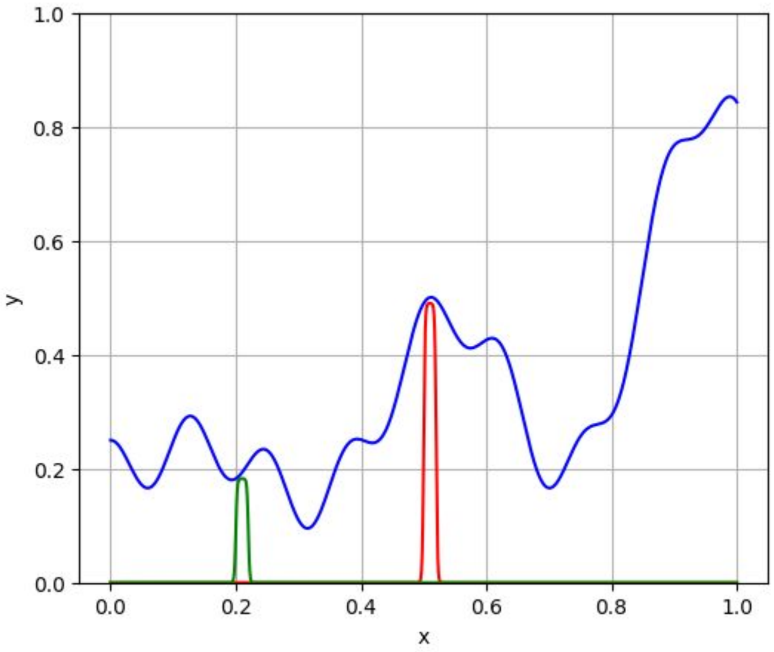


Let's approximate a function *f* with a collection of bumps.

- Start with a bump at 0.5

$$\sum_{j=1}^{M} f(\hat{x}_j) \cdot (\sigma(wx - w\hat{x}_j) + \sigma(-wx + w(\hat{x}_j + \epsilon)) - 1)$$

Universal Approximation Theorem

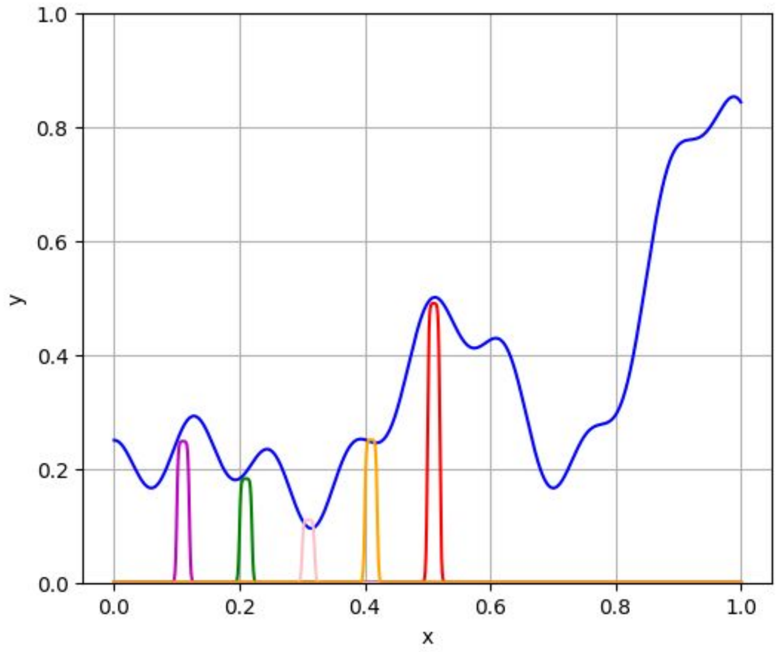


Let's approximate a function *f* with a collection of bumps.

- Start with a bump at 0.5
- Add another one at 0.2
-

$$\sum_{j=1}^{M} f(\hat{x}_j) \cdot \left(\sigma(wx - w\hat{x}_j) + \sigma(-wx + w(\hat{x}_j + \epsilon)) - 1\right)$$

Universal Approximation Theorem

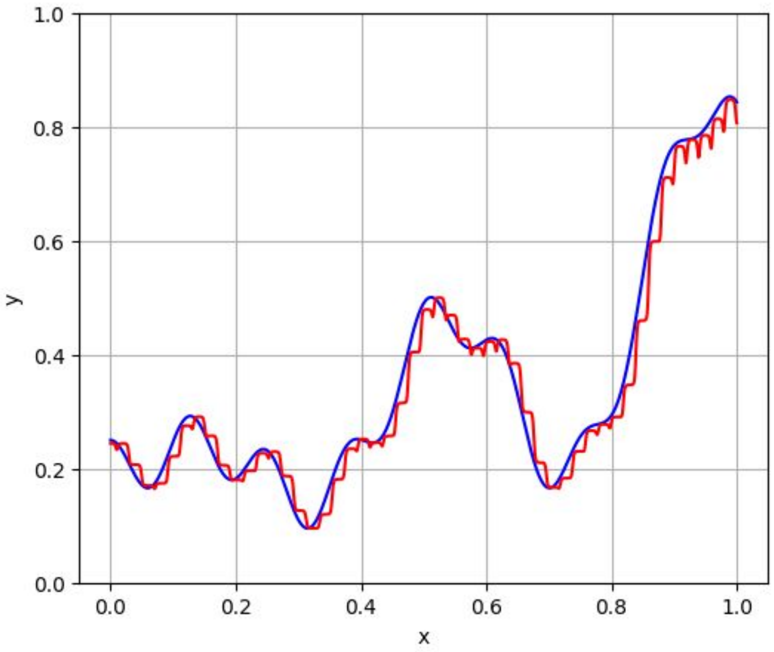


Let's approximate a function *f* with a collection of bumps.

- Start with a bump at 0.5
- Add another one at 0.2
- Keep going…

$$\sum_{j=1}^{M} f(\hat{x}_j) \cdot \left(\sigma(wx - w\hat{x}_j) + \sigma(-wx + w(\hat{x}_j + \epsilon)) - 1\right)$$

Universal Approximation Theorem

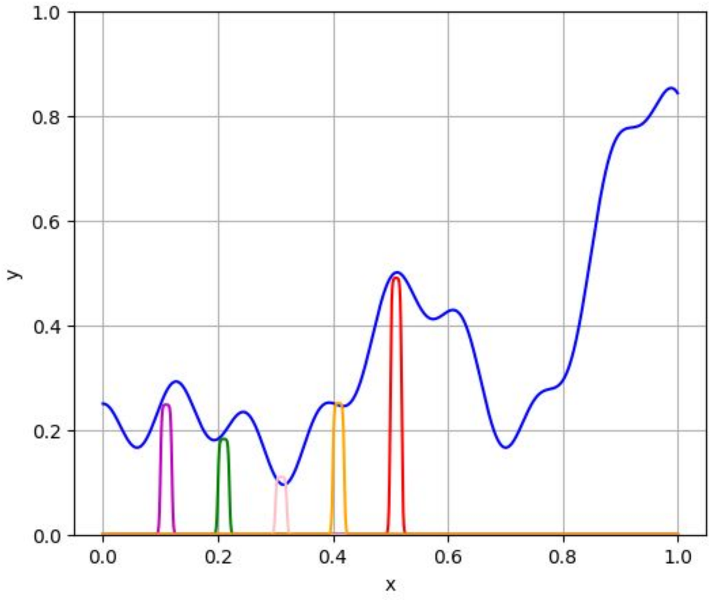


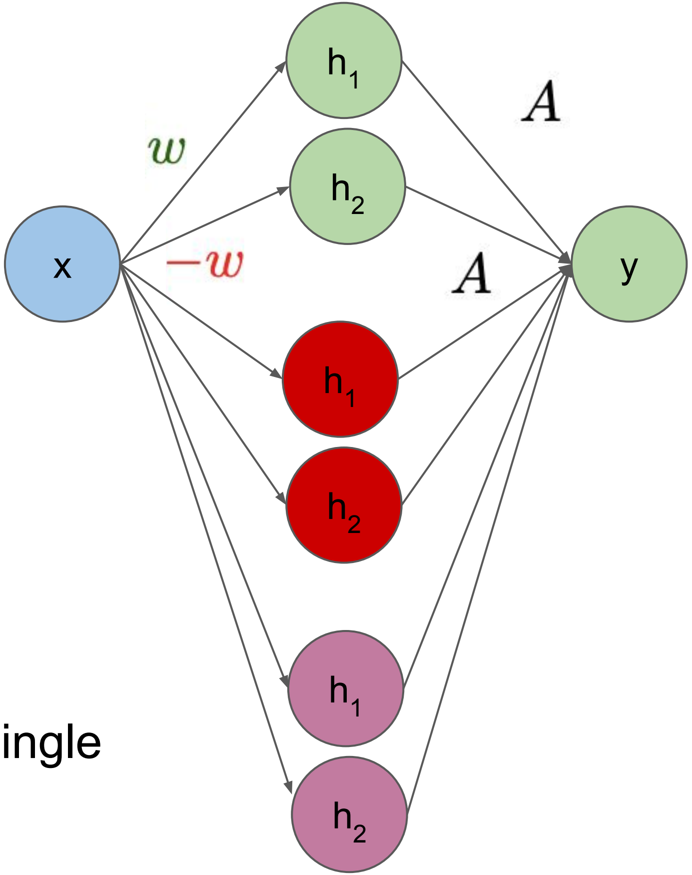
Let's approximate a function *f* with a collection of bumps.

- Start with a bump at 0.5
- Add another one at 0.2
- Keep going…
- 100 sigmoid pairs!

$$\sum_{j=1}^{M} f(\hat{x}_j) \cdot (\sigma(wx - w\hat{x}_j) + \sigma(-wx + w(\hat{x}_j + \epsilon)) - 1)$$

Universal Approximation Theorem



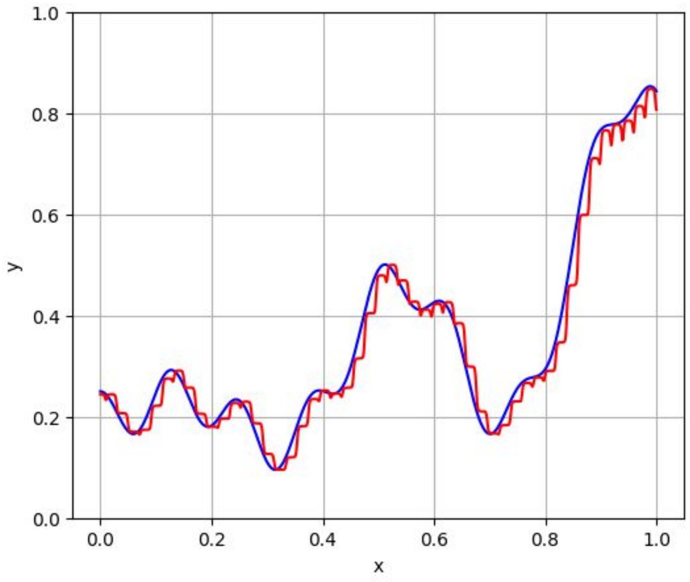


Putting the bumps together, we still get a single layer network.

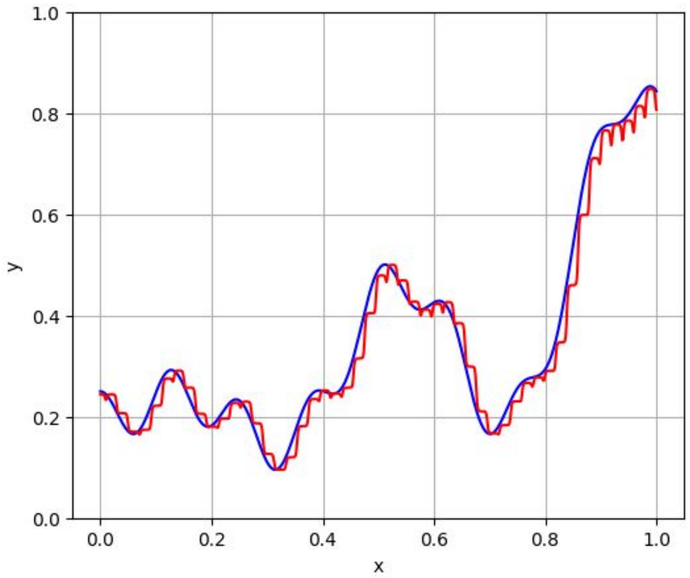
Universal Approximation Theorem

The theorem is still true if

- Multiple inputs
- Multiple outputs
- Almost arbitrary non-linearity



Universal Approximation Theorem



The theorem is still true if

- Multiple inputs
- Multiple outputs
- Almost arbitrary non-linearity

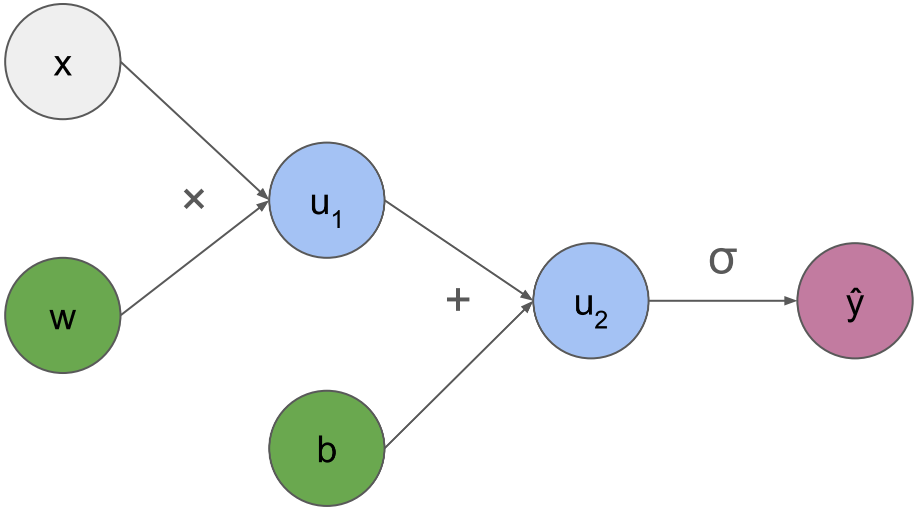
Is this how neural networks approximate functions?

- No, this is a theoretical construction; models are more parameter-efficient
- In practice, depth makes a difference

Backpropagation

See https://www.deeplearningbook.org/, chapter 6.5

Computation Graphs

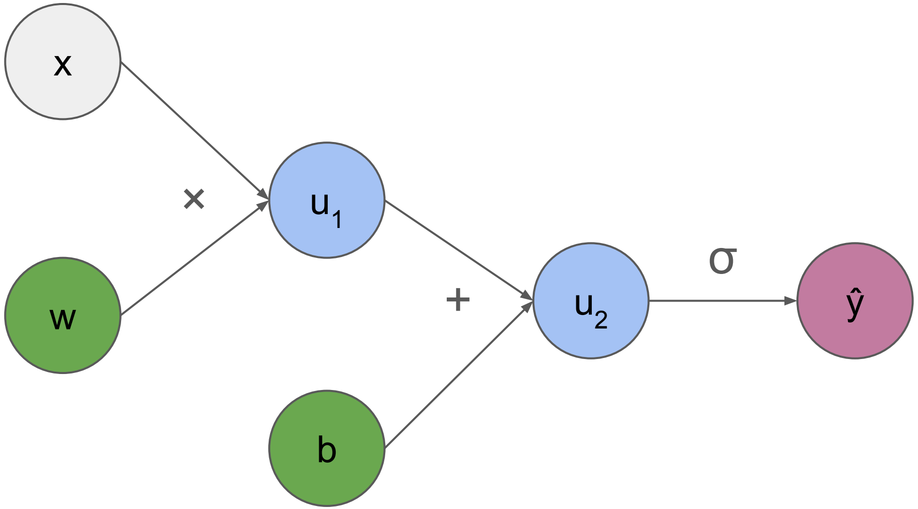


We will represent computations as graphs

- Each nodes represents a variable: input, output or intermediate
- All non-input variables are results of atomic operations on other variables
- What could the computational graph on the left do?

See https://www.deeplearningbook.org/, chapter 6.5

Computation Graphs



Notice that we introduced auxiliary variables that we don't normally have.

We will represent computations as graphs

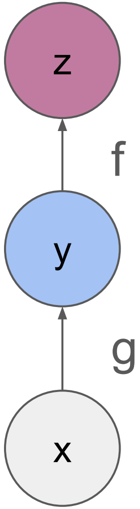
- Each nodes represents a variable: input, output or intermediate
- All non-input variables are results of atomic operations on other variables
- What could the computational graph on the left do?
  Logistic regression prediction

Chain Rule

$$z = f(g(x))$$
$$y = g(x)$$
$$z = f(y)$$

$$\frac{dz}{dx} =$$

All variables are scalars

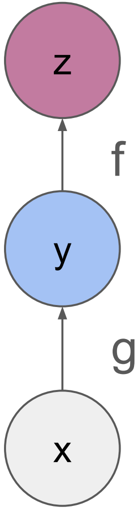


Chain Rule

$$z = f(g(x))$$
$$y = g(x)$$
$$z = f(y)$$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

All variables are scalars

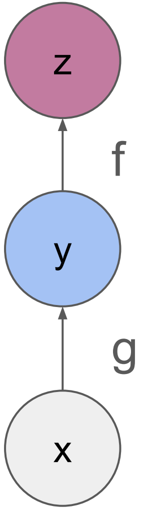


Chain Rule

$$z = f(g(x))$$
$$y = g(x)$$
$$z = f(y)$$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

All variables are scalars

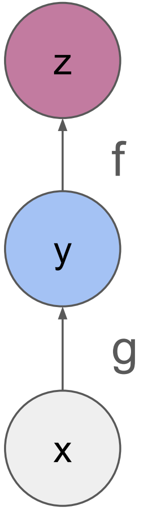


$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i}$$

If y and x are vectors

Chain Rule

$$z = f(g(x))$$
$$y = g(x)$$
$$z = f(y)$$



$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

All variables are scalars

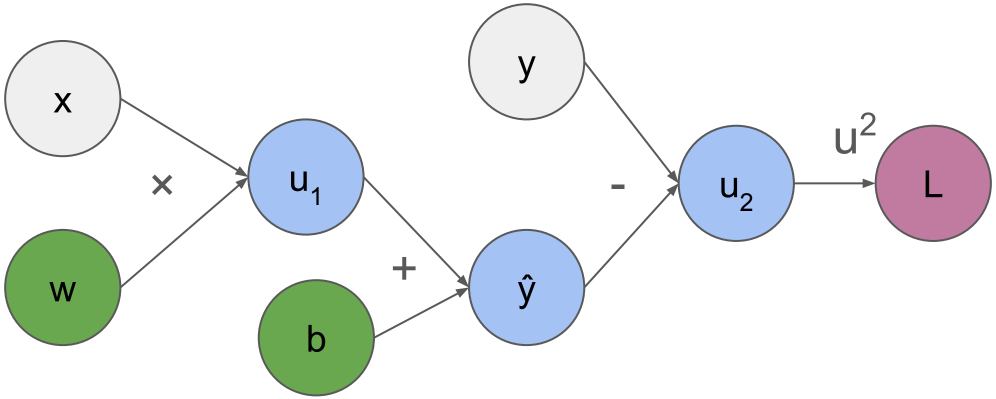
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i}$$

If y and x are vectors

Jacobian matrix

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

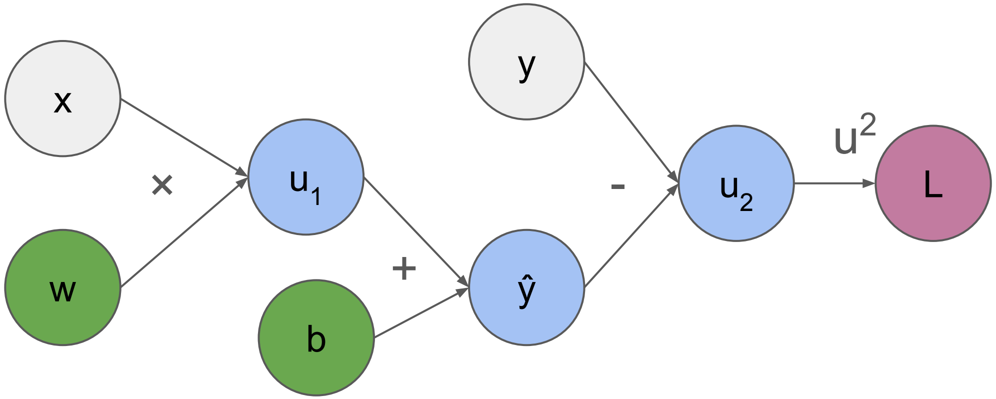
Backpropagation example: linear regression



Forward pass: evaluate the computation graph

$$u_1 = xw$$
$$\hat{y} = u_1 + b$$
$$u_2 = y - \hat{y}$$
$$L = u_2^2$$

Backpropagation example: linear regression



Backward pass: reverse the order of computation

Forward pass: evaluate the computation graph

$$u_1 = xw$$

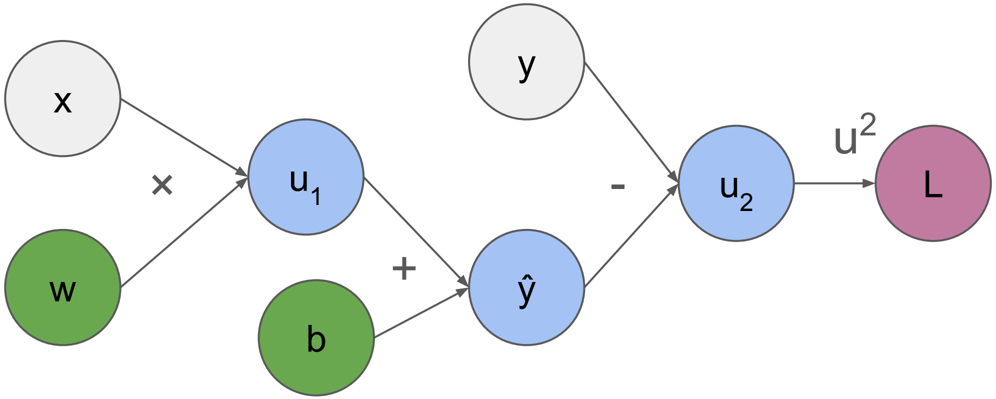
$$\hat{y} = u_1 + b$$

$$u_2 = y - \hat{y}$$

$$L = u_2^2$$

$$\frac{\partial L}{\partial u_2} = 2u_2$$

Backpropagation example: linear regression



Backward pass: reverse the order of computation

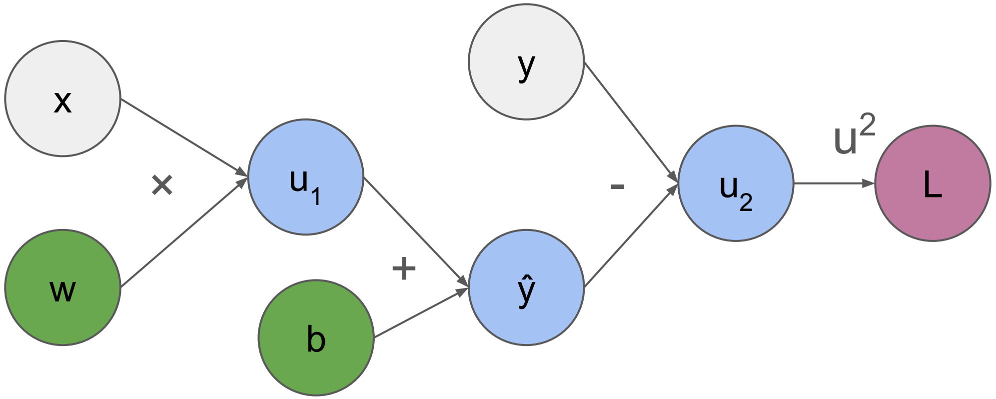
Forward pass: evaluate the computation graph

$$u_1 = xw$$
$$\hat{y} = u_1 + b$$
$$u_2 = y - \hat{y}$$
$$L = u_2^2$$

$$\frac{\partial L}{\partial u_2} = 2u_2$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial u_2} \cdot \frac{\partial u_2}{\partial \hat{y}} = -2u_2$$

Backpropagation example: linear regression



Backward pass: reverse the order of computation

Forward pass: evaluate the computation graph

$$u_1 = xw$$
$$\hat{y} = u_1 + b$$
$$u_2 = y - \hat{y}$$
$$L = u_2^2$$

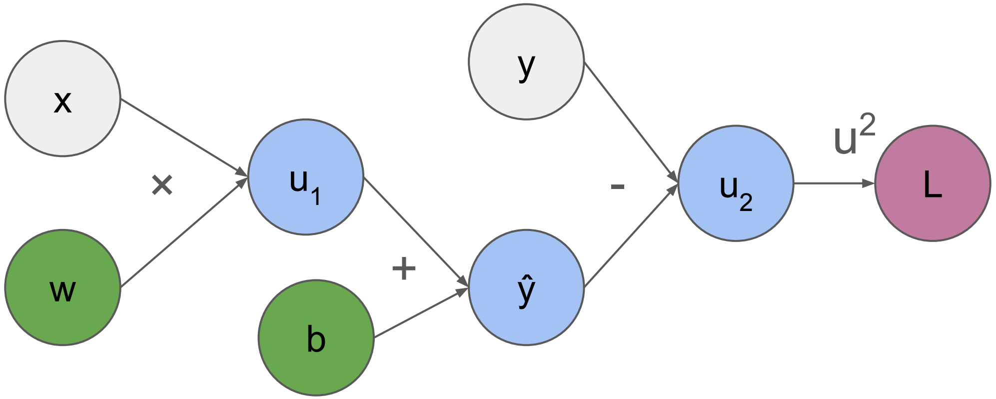
$$\frac{\partial L}{\partial u_2} = 2u_2$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial u_2} \cdot \frac{\partial u_2}{\partial \hat{y}} = -2u_2$$

$$\frac{\partial L}{\partial u_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial u_1} = -2u_2$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = -2u_2$$

Backpropagation example: linear regression



Backward pass: reverse the order of computation

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u_1} \cdot \frac{\partial u_1}{\partial w} = -2u_2 \cdot x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = -2u_2$$

$$\frac{\partial L}{\partial u_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial u_1} = -2u_2$$

Forward pass: evaluate the computation graph

$$u_1 = xw$$

$$\hat{y} = u_1 + b$$

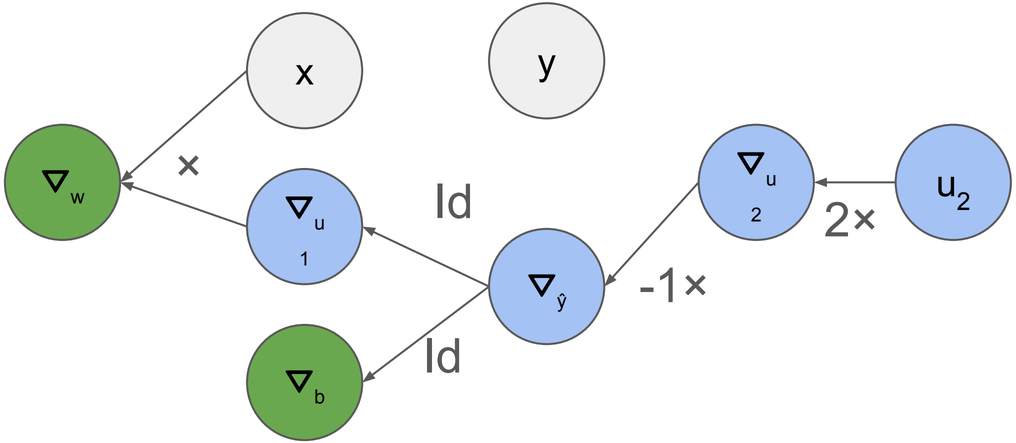
$$u_2 = y - \hat{y}$$

$$L = u_2^2$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial u_2} \cdot \frac{\partial u_2}{\partial \hat{y}} = -2u_2$$

$$\frac{\partial L}{\partial u_2} = 2u_2$$

Backpropagation example: linear regression



Backward pass: reverse the order of computation

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u_1} \cdot \frac{\partial u_1}{\partial w} = -2u_2 \cdot x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = -2u_2$$

$$\frac{\partial L}{\partial u_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial u_1} = -2u_2$$

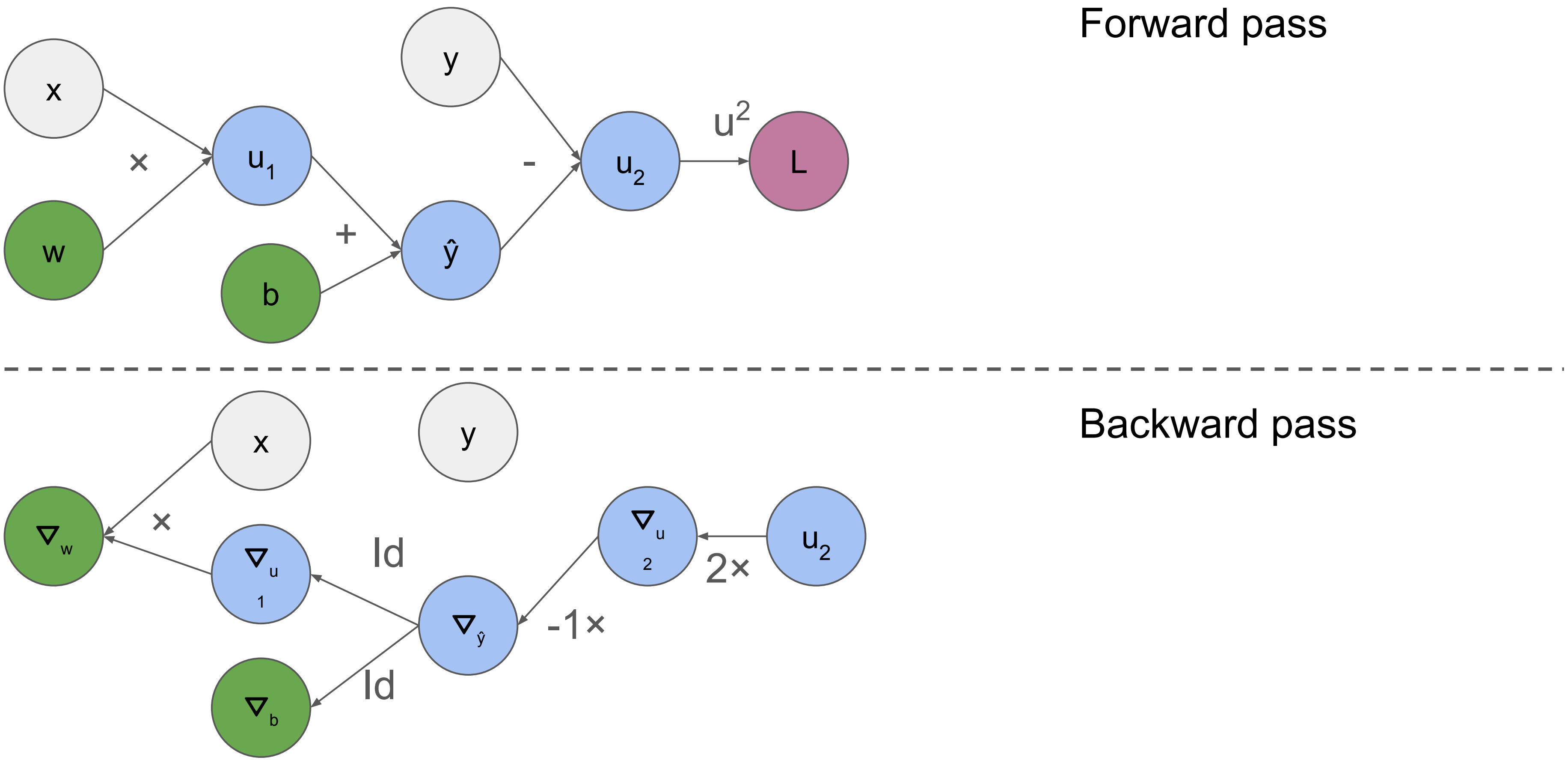
$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial u_2} \cdot \frac{\partial u_2}{\partial \hat{y}} = -2u_2$$

$$\frac{\partial L}{\partial u_2} = 2u_2$$

Forward pass: evaluate the computation graph

$$u_1 = xw$$
$$\hat{y} = u_1 + b$$
$$u_2 = y - \hat{y}$$
$$L = u_2^2$$

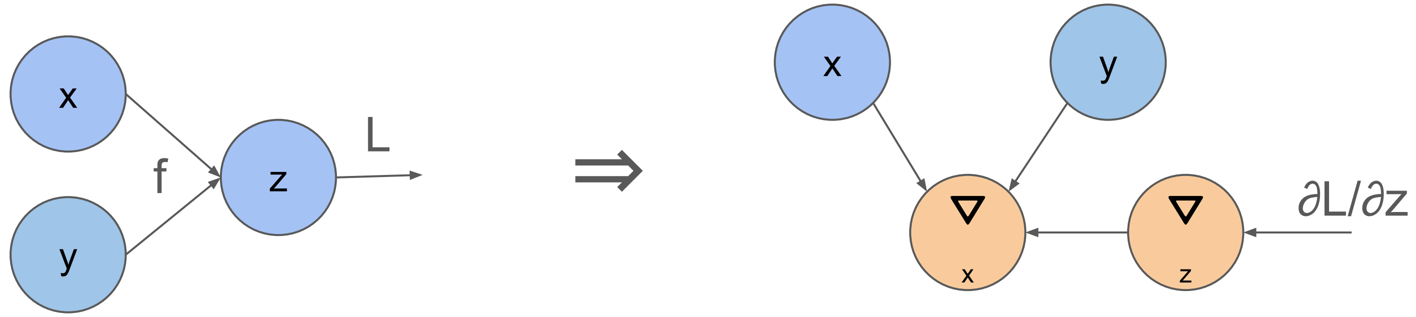
Backpropagation example: linear regression



Backward computation graph

We can always construct the backward graph algorithmically

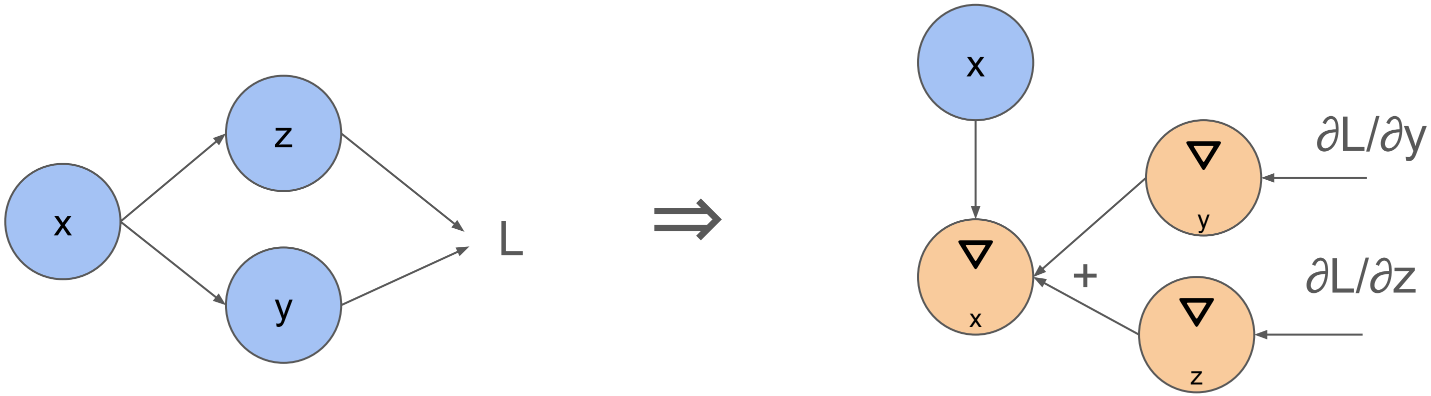
- Nodes:
  - Keep all the nodes from the forward pass
  - Add a gradient node for each of the forward nodes
- Edges:
  - Backward of the original edges + a few rules



Backward computation graph

We can always construct the backward graph algorithmically

- Nodes:
  - Keep all the nodes from the forward pass
  - Add a gradient node for each of the forward nodes
- Edges:
  - Backward of the original edges + a few rules



Backward computation graph

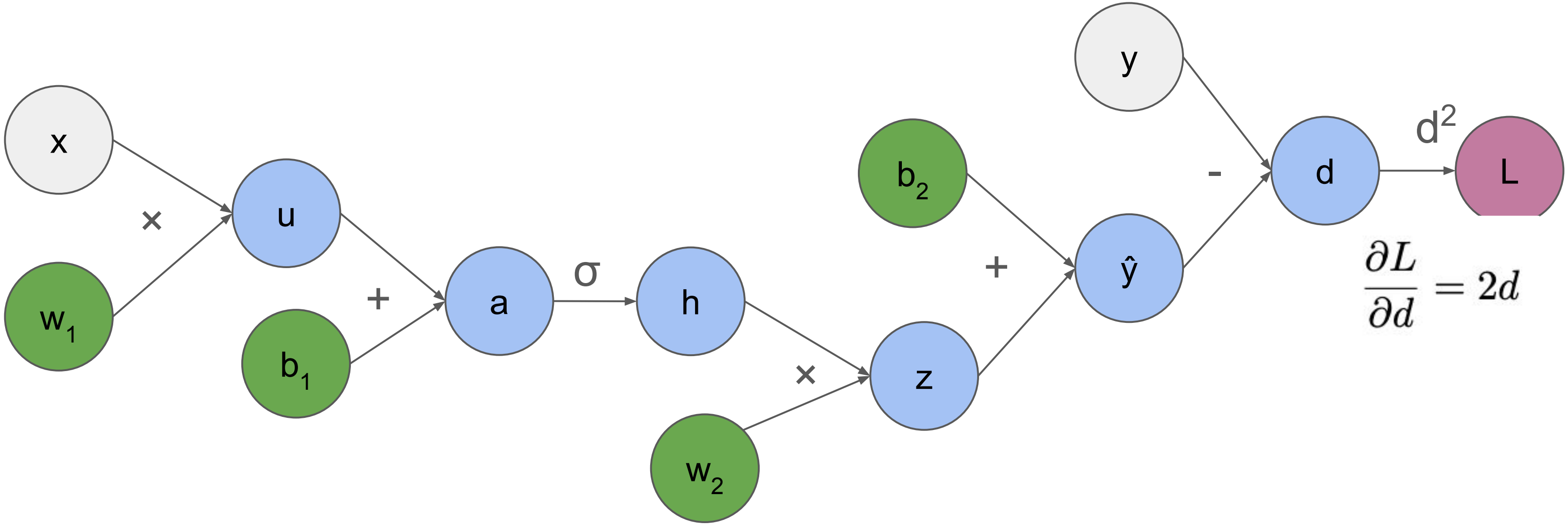
We can always construct the backward graph algorithmically

- Nodes:
  - Keep all the nodes from the forward pass
  - Add a gradient node for each of the forward nodes
- Edges:
  - Backward of the original edges + a few rules

In general, backward pass:

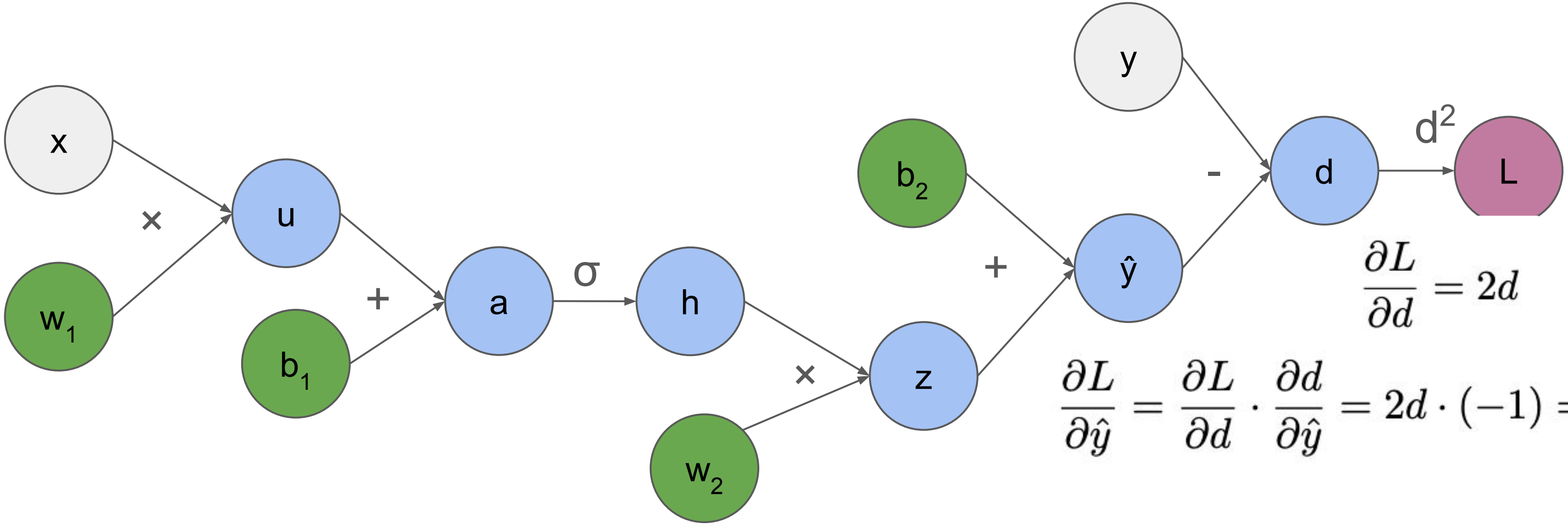
- 2x the nodes, 2x memory
- Need to keep around all intermediate values
- Backward cost ≤ c * forward cost, c is 2-4 for NNs

Backprop: MLP (scalar)



$$\frac{\partial L}{\partial d} = 2d$$

Backprop: MLP (scalar)

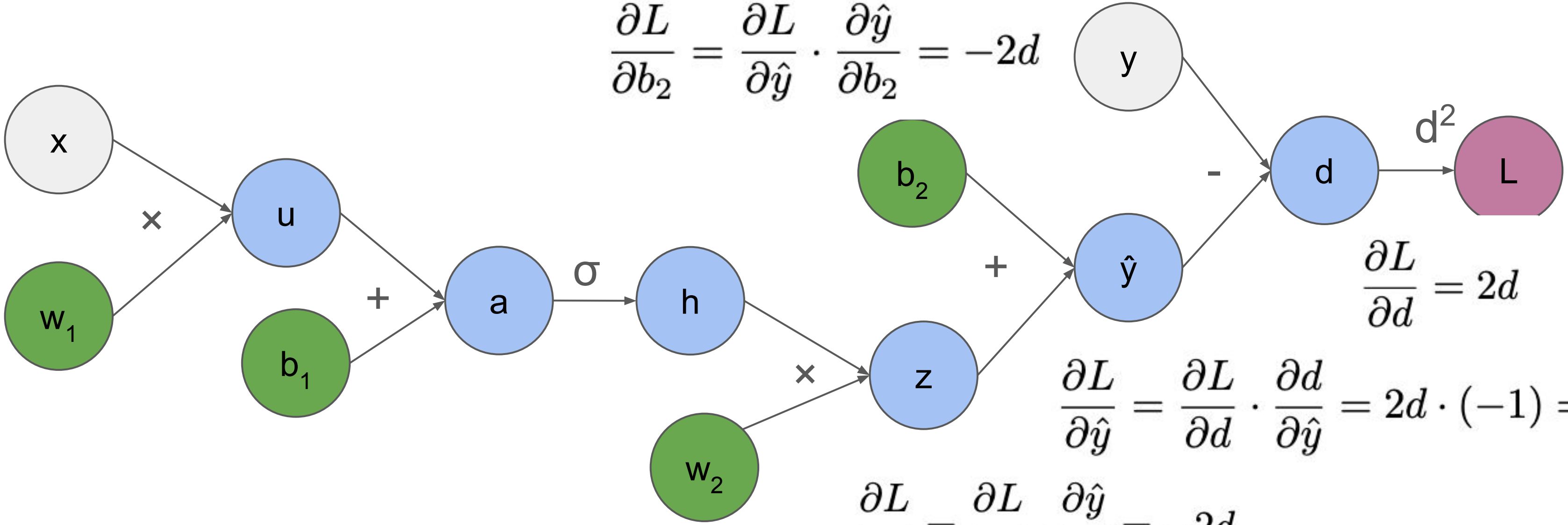


$$\frac{\partial L}{\partial d} = 2d$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial \hat{y}} = 2d \cdot (-1) = -2d$$

Backprop: MLP (scalar)

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_2} = -2d$$

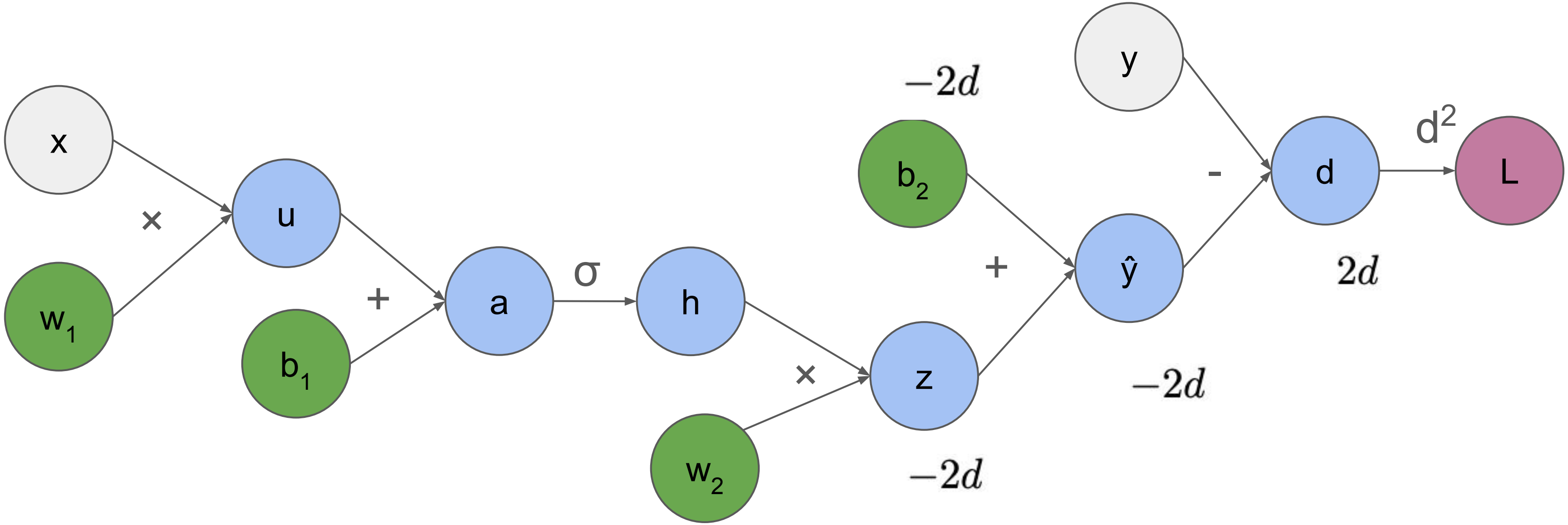


$$\frac{\partial L}{\partial d} = 2d$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial \hat{y}} = 2d \cdot (-1) = -2d$$

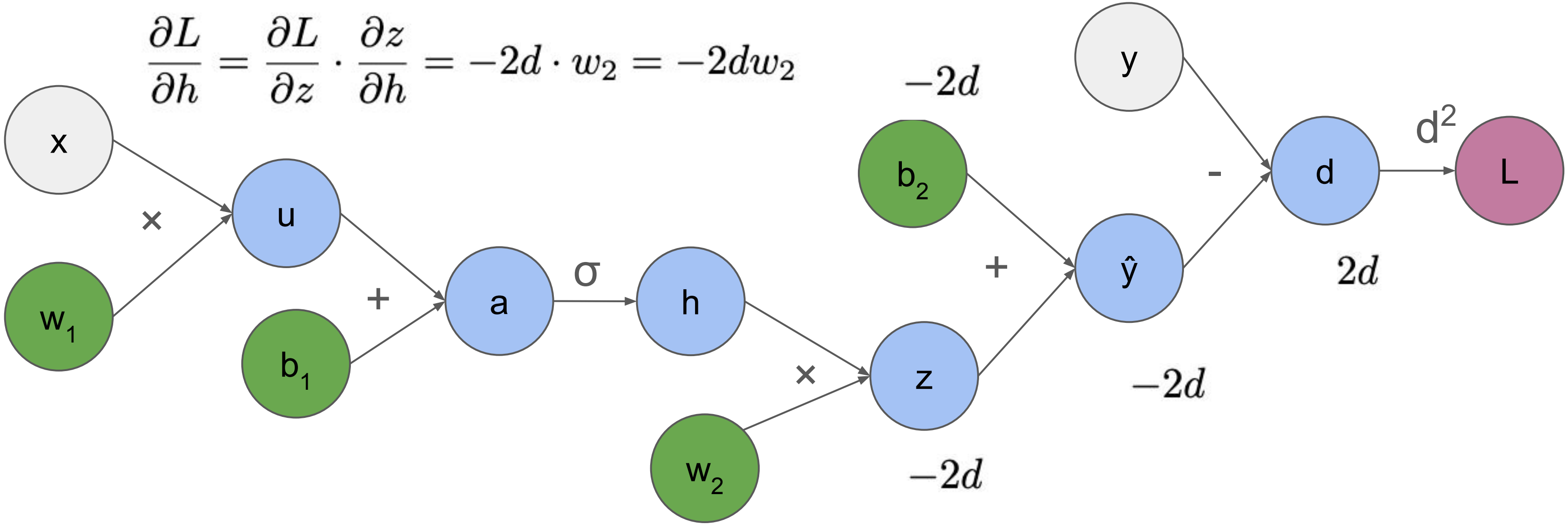
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = -2d$$

Backprop: MLP (scalar)



Backprop: MLP (scalar)

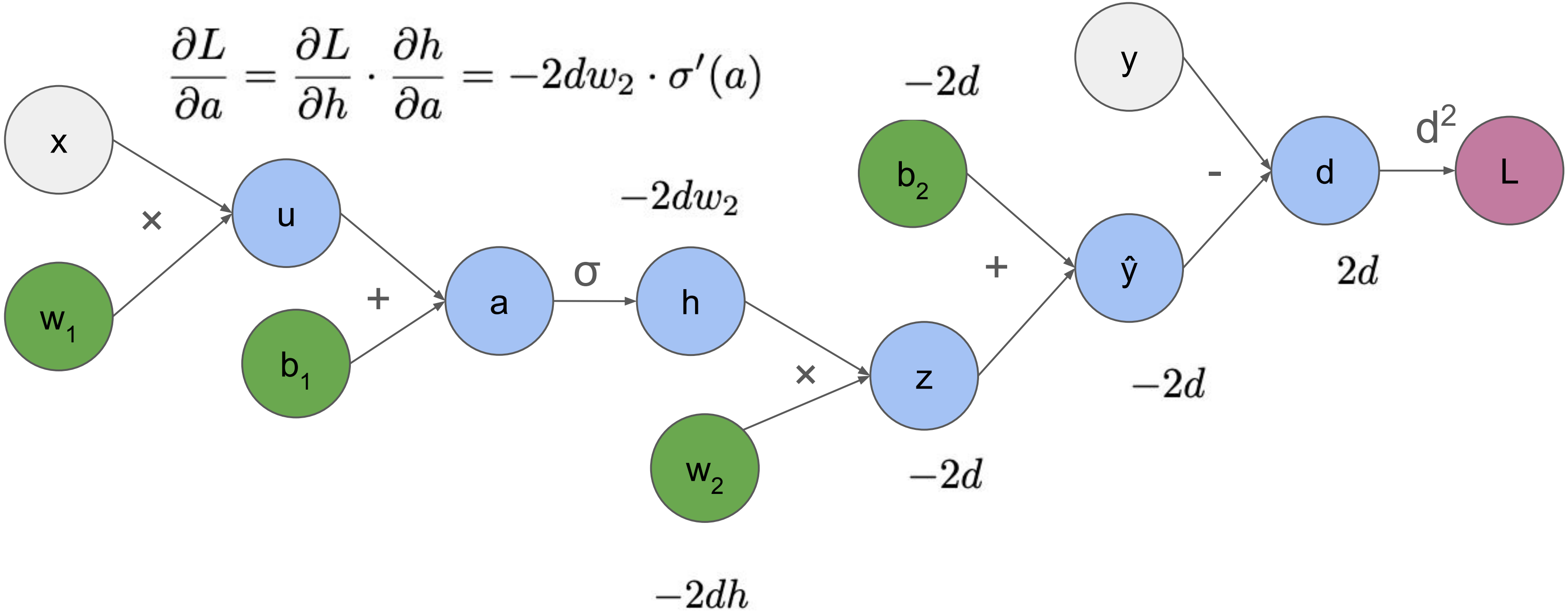
$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial h} = -2d \cdot w_2 = -2dw_2$$



$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_2} = -2d \cdot h = -2dh$$

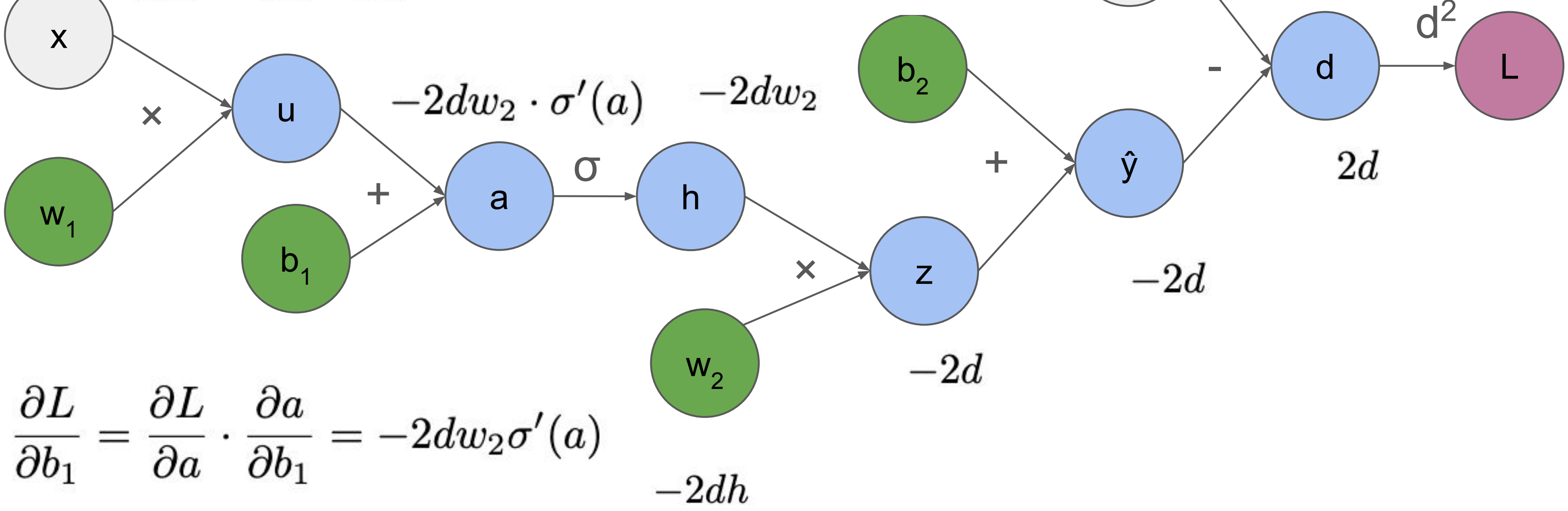
Backprop: MLP (scalar)

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial a} = -2dw_2 \cdot \sigma'(a)$$



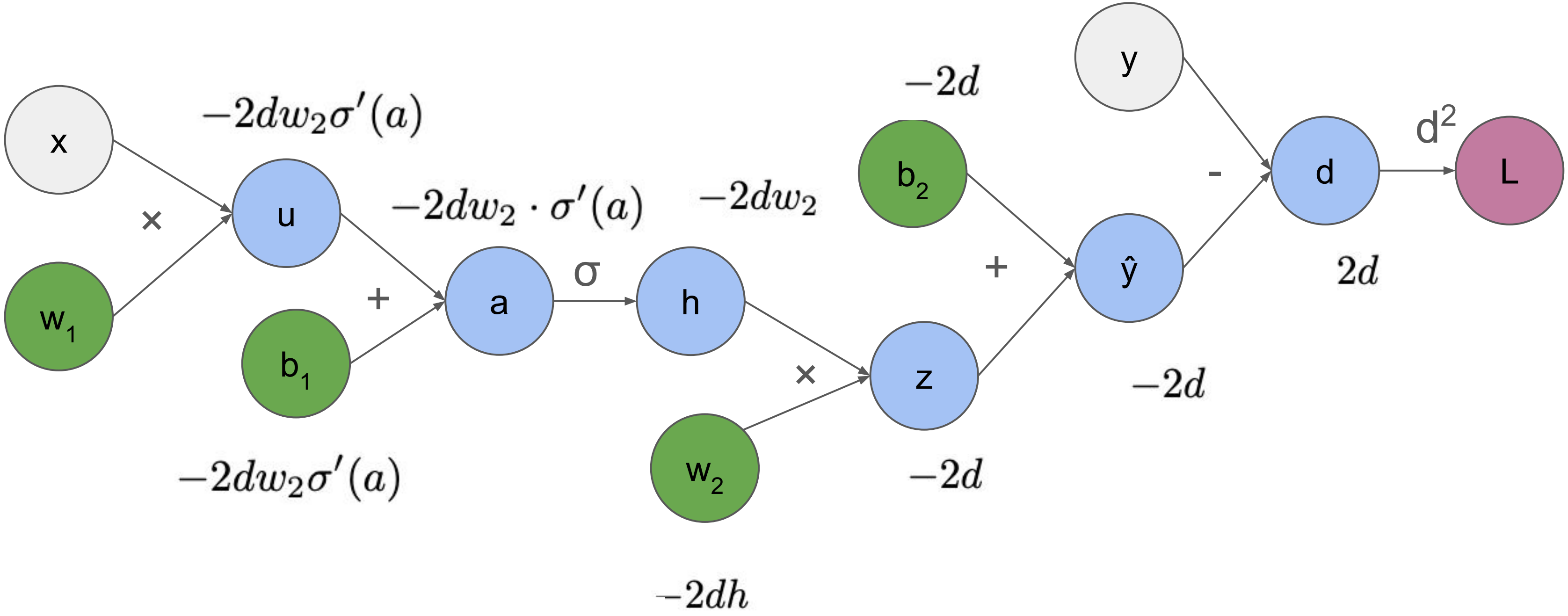
Backprop: MLP (scalar)

$$\frac{\partial L}{\partial u} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial u} = -2dw_2\sigma'(a)$$



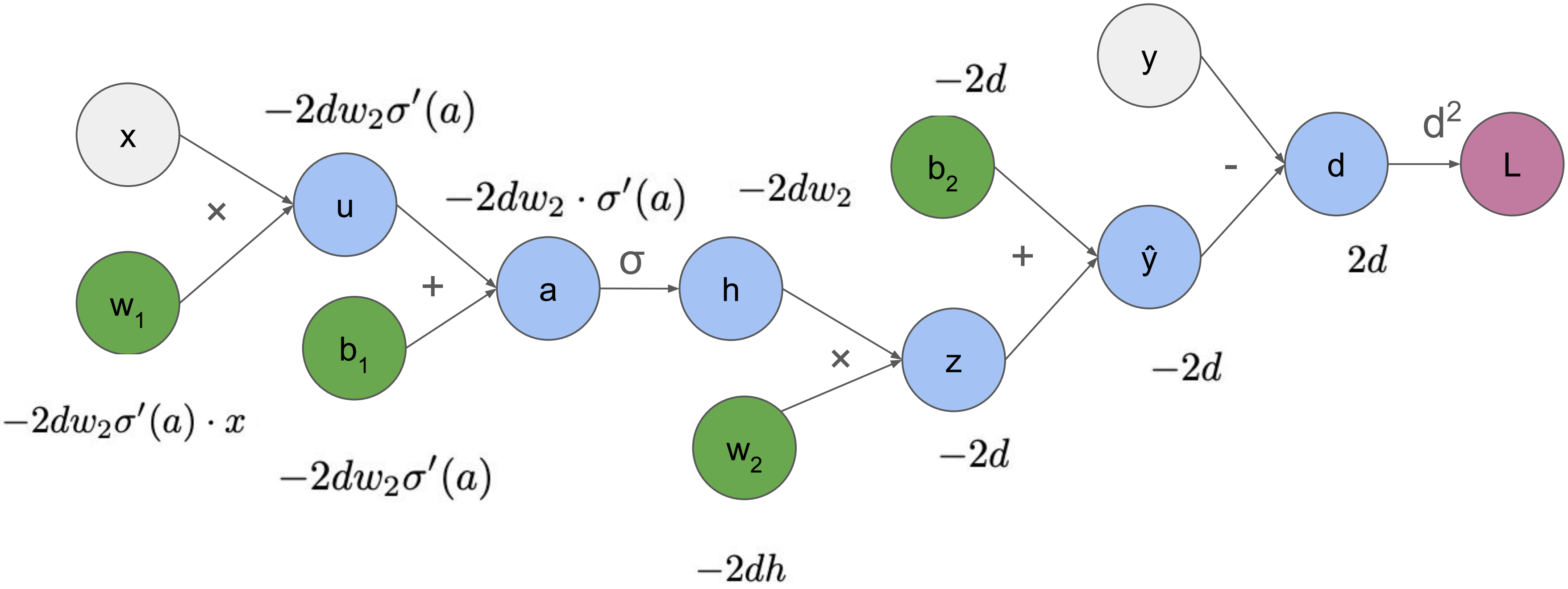
$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial b_1} = -2dw_2\sigma'(a)$$

Backprop: MLP (scalar)



$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial u} \cdot \frac{\partial u}{\partial w_1} = -2dw_2\sigma'(a) \cdot x$$

Backprop: MLP (scalar)



We computed the gradients with respect to all the parameters

- We can now update all the parameters with gradient descent

Backprop: MLP general case

$$x \in \mathbb{R}^{n_{in}},\ W_1 \in \mathbb{R}^{n_h \times n_{in}},\ b_1 \in \mathbb{R}^{n_h}$$
$$h \in \mathbb{R}^{n_h},\ W_2 \in \mathbb{R}^{n_{out} \times n_h},\ b_2 \in \mathbb{R}^{n_{out}}$$
$$y, \hat{y}, d \in \mathbb{R}^{n_{out}},\ L \in \mathbb{R}$$

$$u = W_1 x$$
$$a = u + b_1$$
$$h = \sigma(a)$$
$$z = W_2 h$$
$$\hat{y} = z + b_2$$
$$d = \hat{y} - y$$
$$L = d^T d = \|d\|^2$$

Forward pass

$$\frac{\partial L}{\partial L} = 1$$
$$\frac{\partial L}{\partial d} = 2d$$
$$\frac{\partial L}{\partial \hat{y}} = 2d$$
$$\frac{\partial L}{\partial b_2} = 2d$$
$$\frac{\partial L}{\partial z} = 2d$$

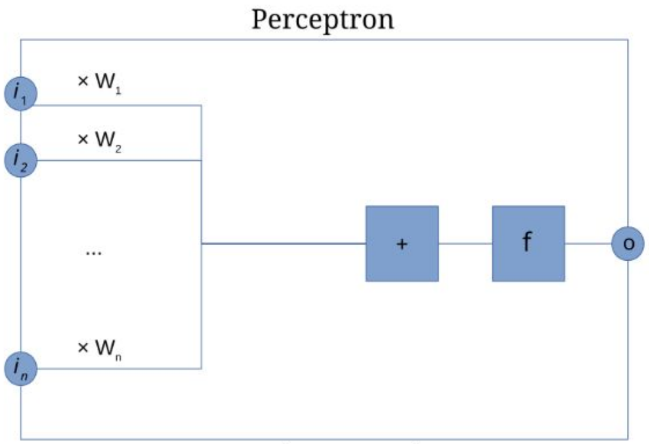
$$\frac{\partial L}{\partial W_2} = 2d\, h^T$$
$$\frac{\partial L}{\partial h} = W_2^T (2d)$$
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial h} \odot \sigma'(a)$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial a}$$
$$\frac{\partial L}{\partial u} = \frac{\partial L}{\partial a}$$
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial u} x^T$$

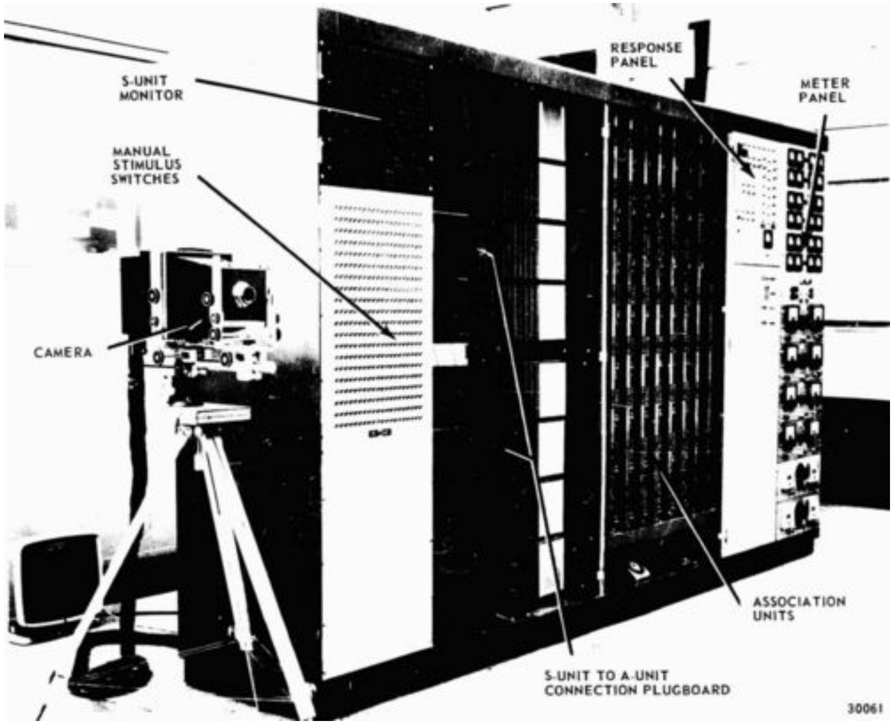
Backward pass

History

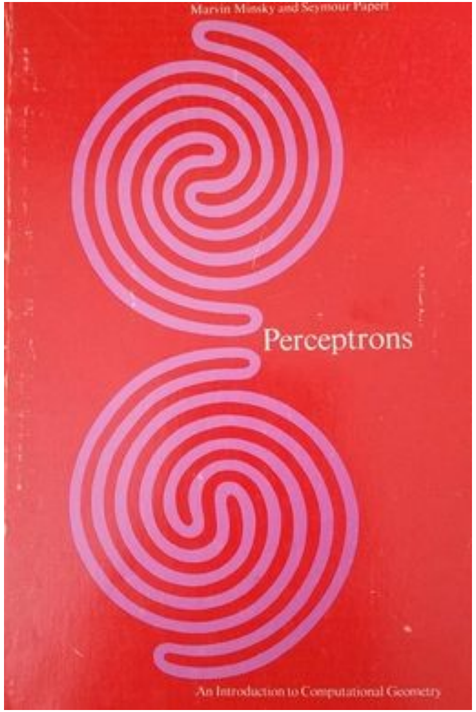
Rosenblatt Perceptron (1957)



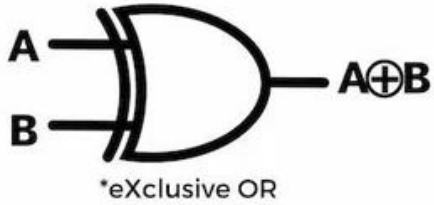
$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$



Marvin Minsky, Seymour Papert (1969)



Showed that perceptrons cannot represent XOR

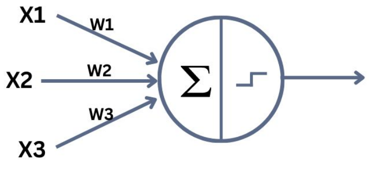


A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

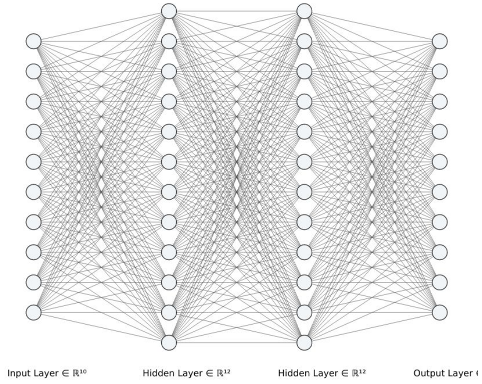
Caused AI winter

Hinton et al: Backpropagation (1986)

Showed how to train Multi-Layer Perceptrons



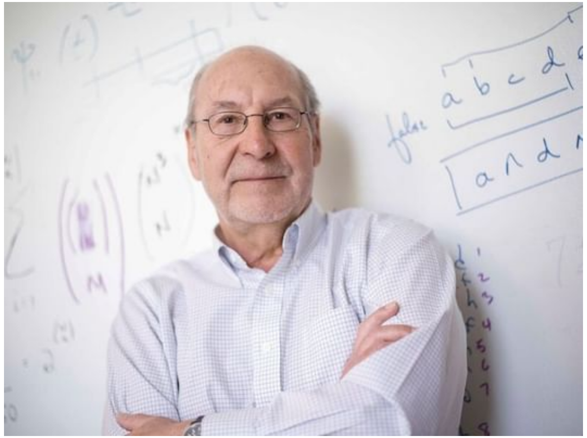
Single-layer perceptron

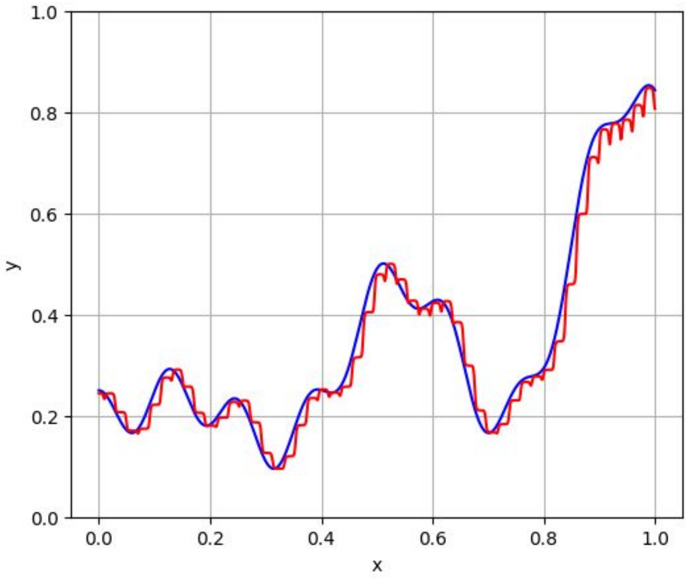


Multi-layer perceptron

Universal approximation theorem: Cybenko (1989)

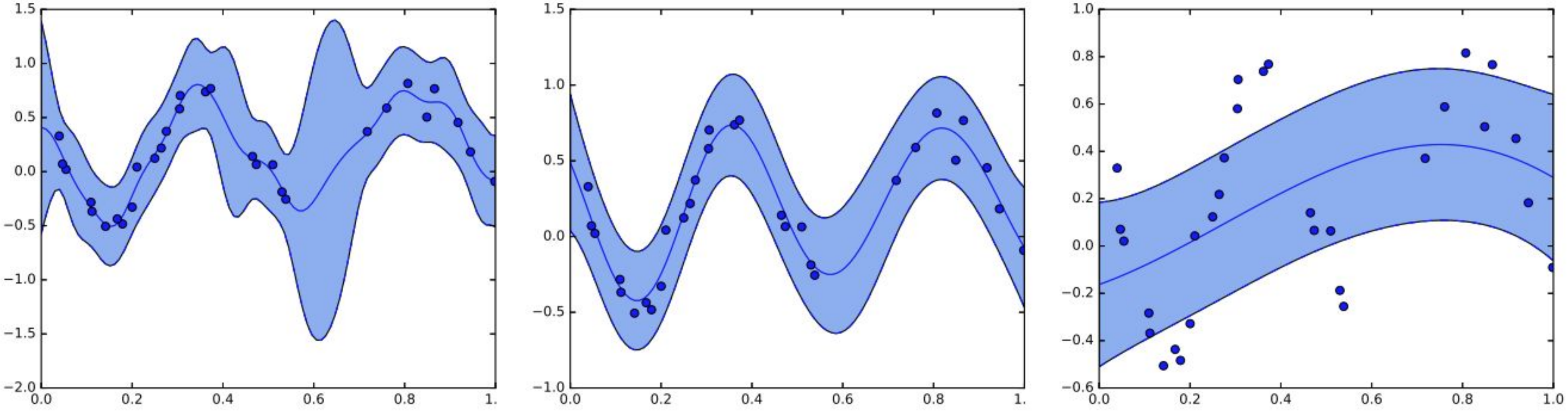
Showed MLPs can approximate arbitrary functions





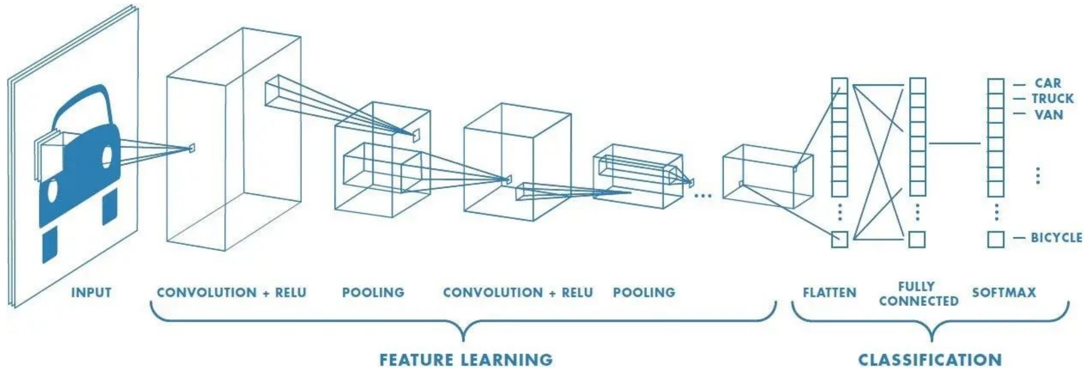
Kernel Methods (1990-2000)

Another winter: kernel methods work better and are more interpretable



ConvNets: Yann Lecun (1989)

Yann invented CNNs which eventually became state-of-the art for image processing



ImageNet Moment (2012)

Ilya Sutskever

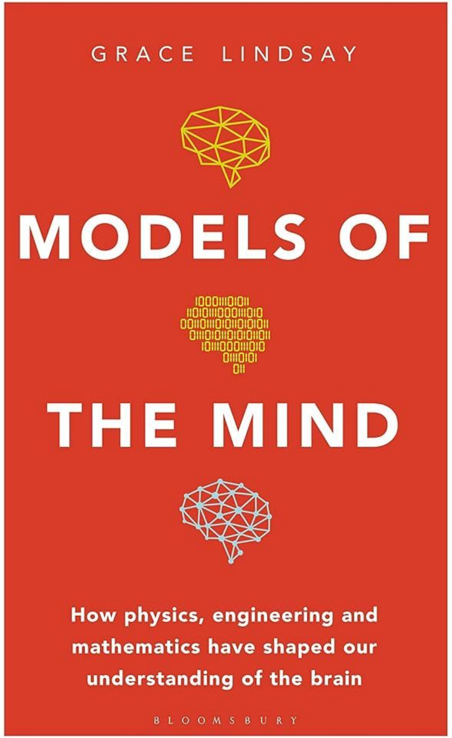
Start of the current era, neural nets outcompete all other methods on an image processing problem.

ImageNet Moment (2012)

Ilya Sutskever

Start of the current era, neural nets outcompete all other methods on an image processing problem.

More history



Great book if you want to learn more about the history of neural networks.