# An Introduction to Deep Learning

James Nesbit

June 20, 2019

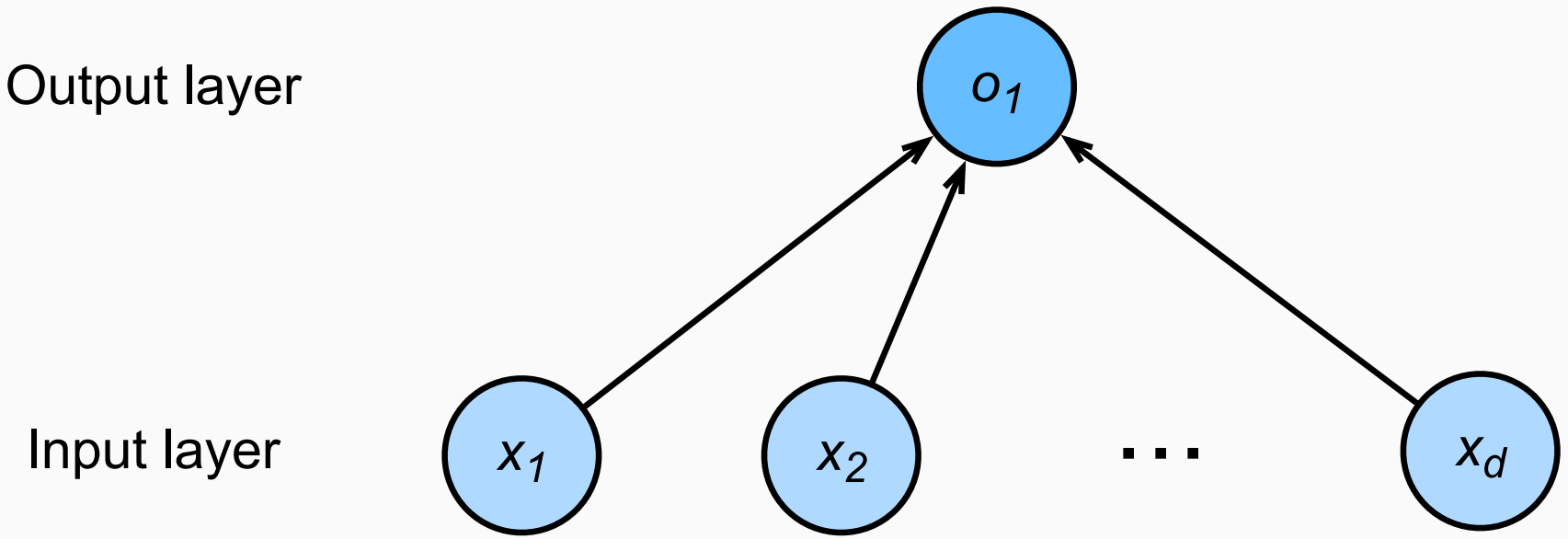
# Why Do We Care?

- Neural networks are state of the art
  - Computer vision
  - Natural language
  - Machine translation
  - Speech recognition
  - Control problems (self-driving cars, Go, Dota2)
  - Time series forecasting
- Not yet state of the art with 'structured data' (compared to trees)

# Linear Regression

- Suppose we have $n$ observations $(\mathbf{x}^{(i)}, y^{(i)})$, $i = 1, \ldots, n$
- Let $\mathbf{x}^{(i)}$ denote our **inputs** (covariates) with **feature dimension** $d$ $(\mathbf{x}^{(i)} \in \mathbb{R}^d)$
- Let $y^{(i)}$ denote our **labels** (responses)

# Network Architecture



# Network Architecture

- The **input layer** is a single point of covariate data, $\mathbf{x} \in \mathbb{R}^d$
- The **output layer** is a linear combination of the connected nodes in the input, with **weights w** and a **bias** $b$

$$\begin{aligned} o_1 &= b + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 \\ &= \mathbf{w}^\mathsf{T} \mathbf{x} \end{aligned}$$

- The model outputs a prediction

$$\hat{y} = \sigma_y(o_1)$$

Where in our case the **activation function** for the output layer is linear, i.e. $\sigma_y(x) = x$

# Forward Propagation

- Thus the model is

$$o_1 = \mathbf{w}^\mathsf{T}\mathbf{x}$$
$$\hat{y} = \sigma_y(o_1)$$

- The act of computing the prediction $\hat{y}$, is known as **forward propagation**, and the networks are often called **feedforward neural networks**
  - In contrast with Recurrent Neural Networks which have **feedback connections**

# Loss Function

- In addition to network architecture we also need a loss function
- The loss function for observation $i$ is given by $\ell^{(i)}(\mathbf{w}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$
- The loss of the entire dataset is

$$L(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell^{(i)}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(\mathbf{w}^\intercal x^{(i)} - y^{(i)})^2$$

- Our goal is to choose $\mathbf{w}$ to minimize the loss across all training samples.

# Gradient Descent

- Starting at an initial guess $\mathbf{w}_0$, and consider the sequence $\mathbf{w}_0, \mathbf{w}_1, \ldots$ such that

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \nabla L(\mathbf{w}_n)$$

- For sufficiently small $\gamma_n \in \mathbb{R}_+$, we will have $L(\mathbf{x}_0) > L(\mathbf{x}_1) > \ldots$, and the sequence will converge to a local minima
    - If $L$ is convex, then a local minima will also be a global minima (note that convexity is not something that we will often be able to leverage)

# Stochastic Gradient Descent

- We can rewrite the equation in the previous slide as

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \sum_{i=1}^{n} \partial_{\mathbf{w}} \ell^i(\mathbf{w_n}).$$

- Now consider randomly sampling a single $j \in \{1, \ldots, n\}$ and update $\mathbf{w}_{n+1}$ using only this observation

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \partial_{\mathbf{w}} \ell^j(\mathbf{w}_n),$$

- This method is known as **stochastic gradient descent** (SGD)

# Mini-batch Gradient Descent

- An another alternative is to sample (without replacement) a mini-batch $\mathcal{B}$ of observations

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \sum_{k=1}^{|\mathcal{B}|} \partial_{\mathbf{w}} \ell^k(\mathbf{w}_n),$$

- Known as **mini-batch gradient descent** (MBGD).

# Which Method to Use

- Stochastic gradient descent is the preferred method because it uses information 'more efficiently'
  - Imagine that the data was 10 copies of the same observations
- Requires many more iterations, but fewer computations
- Noisy nature of updates may help us in nonconvex problems
- Mini-batch has flavours of both, but allows for vectorization (particularly using GPUs)
  - For this reason mini-batch is the method of choice

# Gradient Computation

- Gradients are not computed using closed equations
- Instead we use **automatic differentiation**
- Every computer program executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.)
- We can use the chain rule, to accurately compute derivatives
- Note this is distinct from
  - Numerical differentiation (method of finite differences)
  - Symbolic differentiation

# Softmax Regression

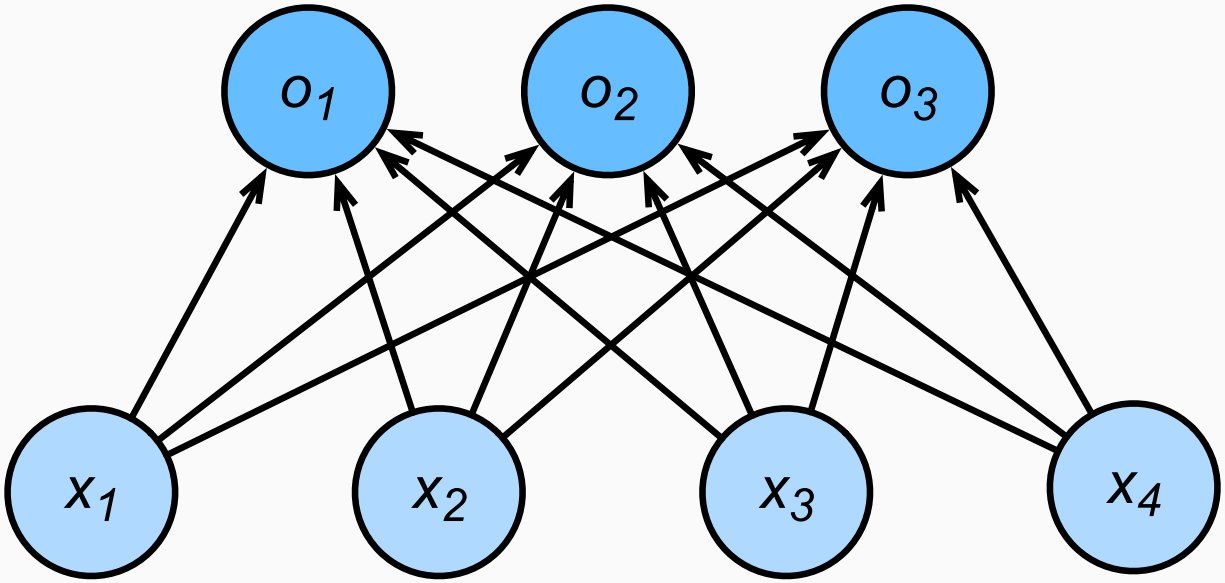
- Also known as logistic regression
  - Note that ML people make a distinct between regression ($y$ is continuous) and **classification** ($y$ is discrete)
- Let's suppose our labels have 3 categories, encoded as **one hot encoded** vectors

$$y \in \{(1,0,0),(0,1,0),(0,0,1)\}$$

# Network Architecture

Output layer

Input layer



# Network Architecture

- Note that each node in the input layer is connected to each node in the output layer
  - We would refer to the output layer as a **fully connected layer**
- The output layer is

$$o_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1,$$
$$o_2 = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2,$$
$$o_3 = x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.$$

or $\mathbf{o} = \mathbf{W}\mathbf{x}$

# Network Architecture

- To have the model output probabilities, we will use the **softmax** activation function for output layer:

$$\text{softmax}(o_i) = \frac{\exp(o_i)}{\sum_{j=1}^{3} \exp(o_j)}.$$

- To summarize

$$\mathbf{o} = \mathbf{W}\mathbf{x}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}).$$

- The loss function is the **cross-entropy loss**

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{n} y_j \log(\hat{y}_j)$$

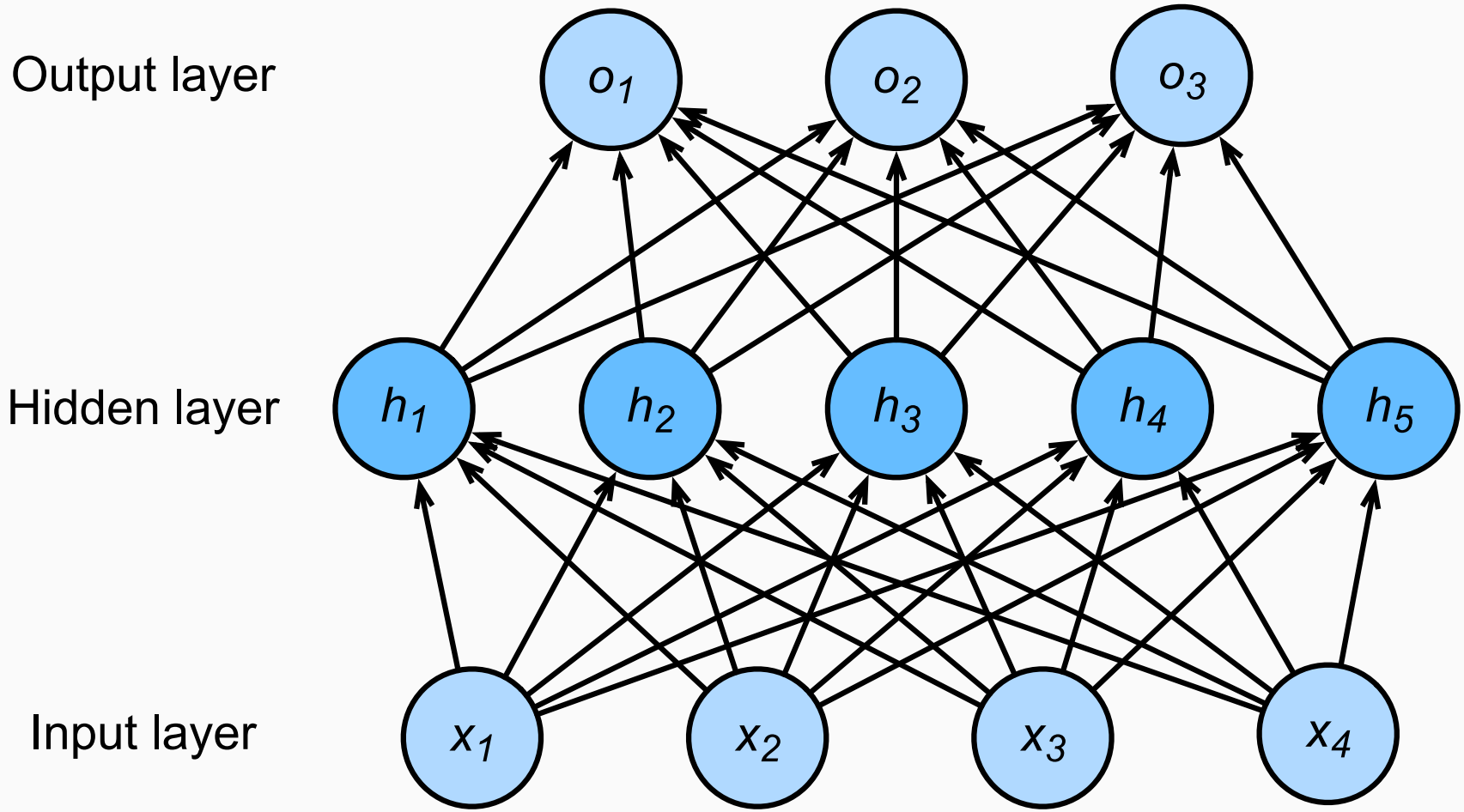
# Deep Networks

- Now let's start to make our model 'deep'
- The model can be written as

$$\mathbf{h} = \mathbf{W}_1\mathbf{x}$$
$$\mathbf{o} = \mathbf{W}_2\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$$

- **h** is referred to as a **hidden layer**

# Network Architecture



# The Need for Activation Functions

- As written, this model doesn't improve at all on the softmax regression as

$$\mathbf{o} = \mathbf{W}_2\mathbf{h} = \mathbf{W}_2\mathbf{W}_1\mathbf{x} = \mathbf{W}x,$$

  Where $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$
- In order to get any benefit from adding in a hidden layer, we add an activation function on our hidden units $\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x})$.

# Generalizing the MLP

- This model is know as the **multilayer perceptron** (MLP)
- Is easy to generalize

$$
\begin{aligned}
\mathbf{h}_1 &= \sigma_1(\mathbf{W}_1\mathbf{x}), \\
\mathbf{h}_j &= \sigma_j(\mathbf{W}_j\mathbf{h}_{j-1}), \text{ for } j = 2, \ldots, L \\
\mathbf{o} &= \mathbf{W}_{L+1}\mathbf{h}_L \\
\hat{\mathbf{y}} &= \sigma_y(\mathbf{o})
\end{aligned}
$$

# Modern Implementations

- Neural networks had an initial burst of research in the late 80s and 90s
  - They typically made one or two hidden layers, which today would be referred to you as a **shallow** neural network
  - Deep neural network didn't perform very well
- ResNet, a convolutional deep neural network used for image recognition, has $1,202$ layers

# What Has Changed

- More data
  - Lots of parameters require lots of data
- More computing power, in particular the use of GPUs
  - We need lots of iterations of gradient descent to achieve a model that fits well
- Regularization techniques
  - Need to prevent overfitting of the model
- Overcoming the numerical issues with computing gradients
  - Known as the vanishing (exploding) gradient problem

# Exploding Gradient Problem

- Consider a deep network with $d$ layers

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}_t)$$
$$\mathbf{o} = f_d \circ f_{d-1} \circ \dots f_1(\mathbf{x})$$

- The gradient of the output layer with respect to weights in $\text{t}^{\text{th}}$ hidden layer

$$\partial_{\mathbf{W}_t}\mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}}\mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \dots \cdot \underbrace{\partial_{\mathbf{h}^t}\mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t}\mathbf{h}^t}_{:=\mathbf{v}_t}.$$

- This idea of errors being pushed back from the network is known as **backpropagation**
  - But its just the chain rule. See Rob Tibshirani's glossary: machine learning vs statistics

# Exploding Gradient Problem

- $\partial_{\mathbf{h}^t} \mathbf{h}^t$ is a vector
- $\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d \dots \partial_{\mathbf{h}^{t+1}} \mathbf{h}^{t+1}$ are matrices
- If matrices have small (large) eigenvalues, then the product will vanish (explode)
    - This is particularly true for early layers, whose gradients are the product of many matrices
- This presents an issue for gradient descent algorithms which will either update very slowly or diverge (in the cast where gradients become large)

# Activation Functions

- The vanishing gradient problem can be caused by or ameliorated by the choice of activation functions that are used
- We will focus on the three most popular activation functions
  - Sigmoid
  - Hyperbolic tangent (tanh)
  - Recitifed Linear Unit (ReLU)

# Sigmoid

- The univariate logitistic / softmax function, defined as

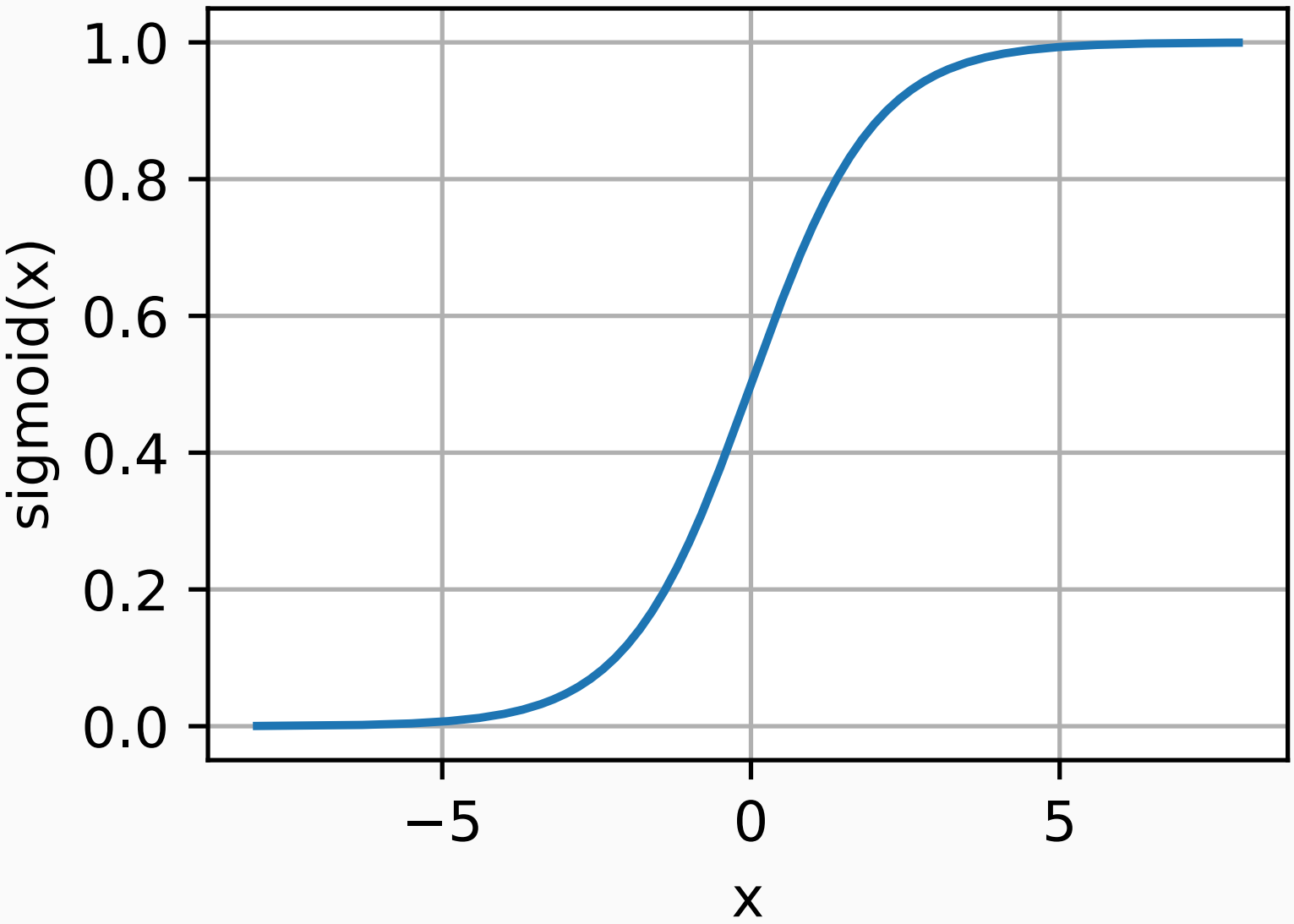
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

- With gradient

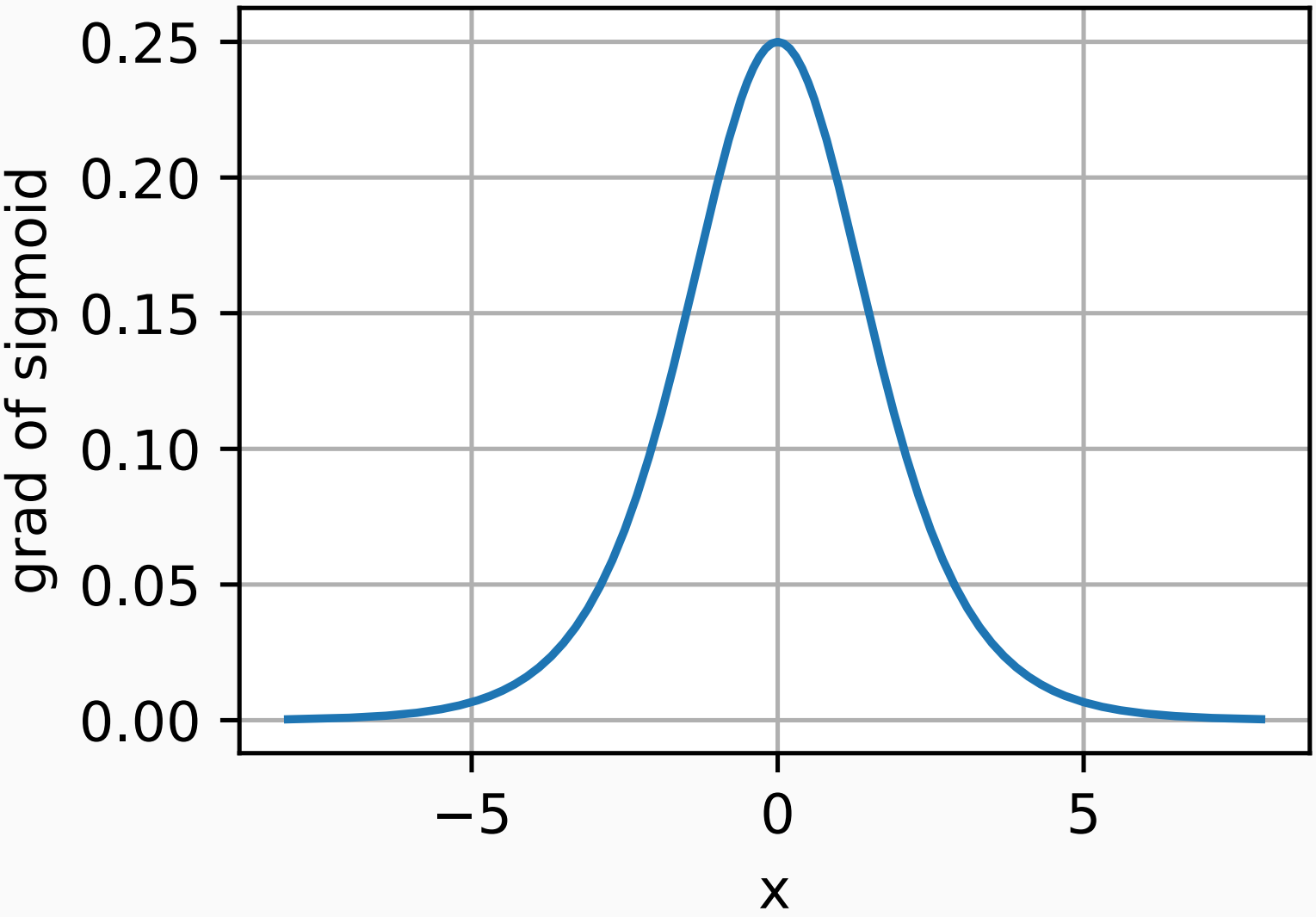
$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2}$$

- Smooth thresholding rule, squashes inputs into $(0, 1)$
- Gradients vanish when $x$ is not in a neighborhood of 0
    - If all units take these values, we would call the network **saturated**

# Sigmoid



# Sigmoid



# Tanh

- The hyperbolic tangent defined as

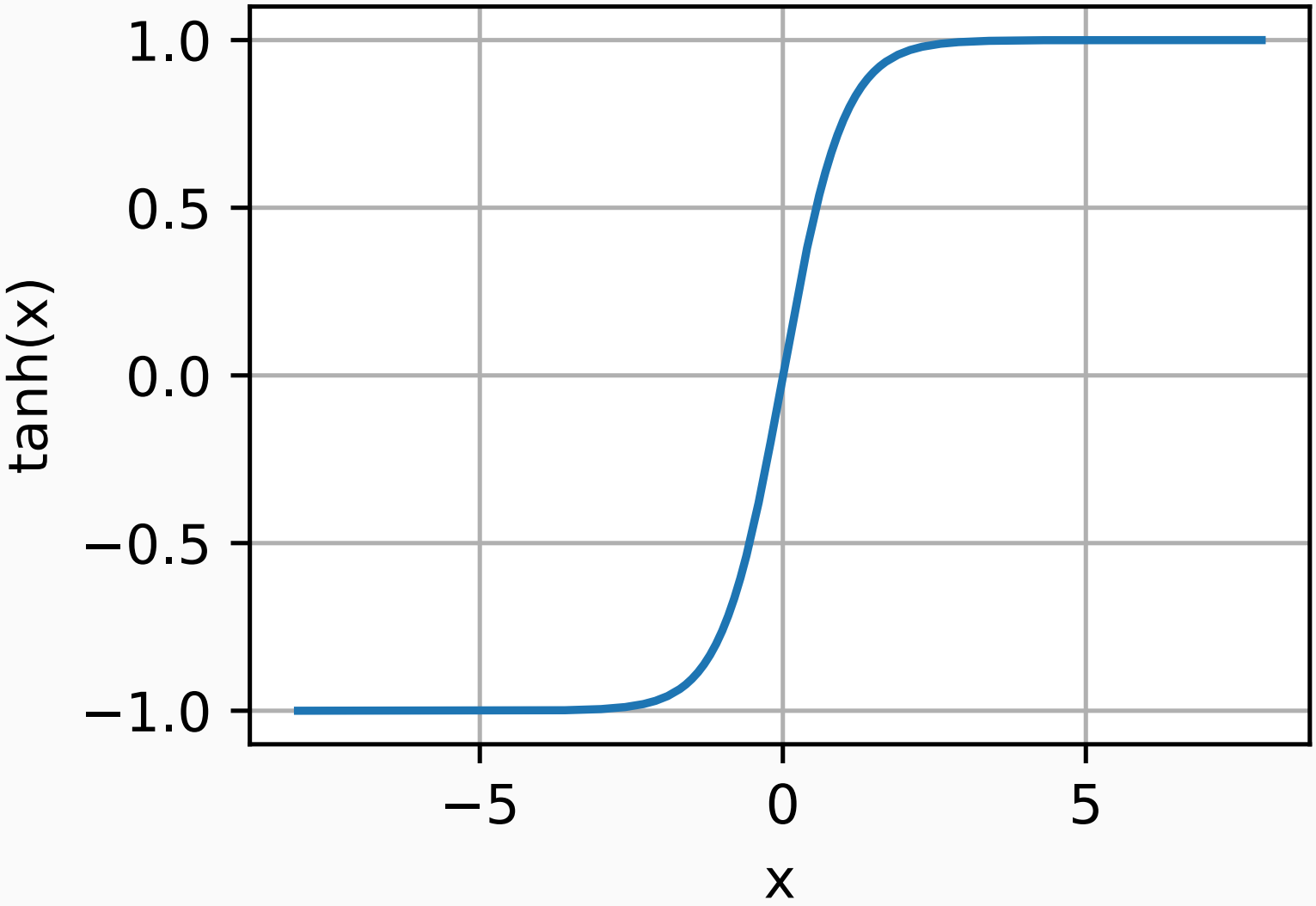
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.(-x).$$

- With gradient

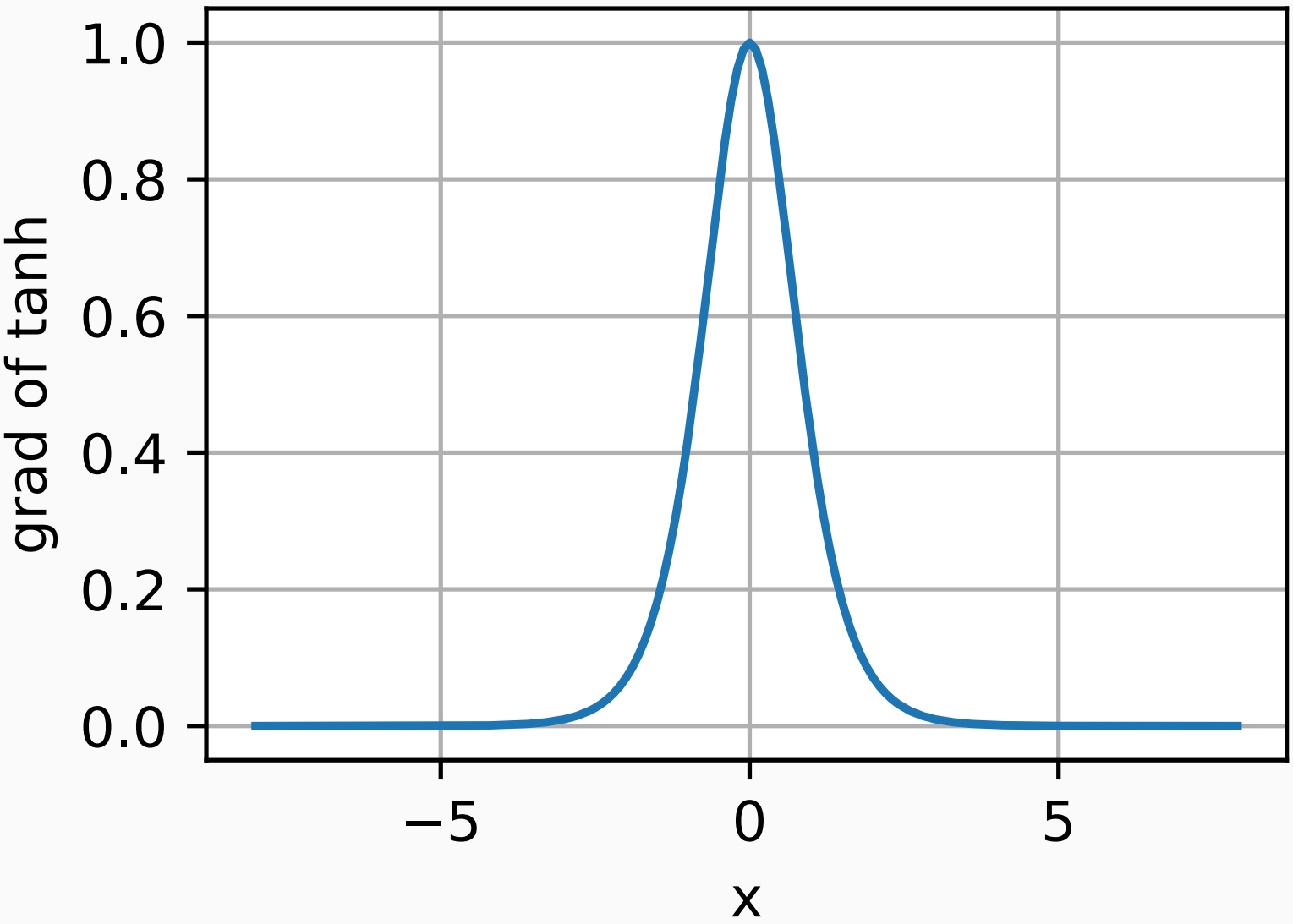
$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

- Squashes inputs into $(-1, 1)$
- Suffers from same gradient problems as sigmoid

# Tanh



# Tanh



# ReLU

- The rectified linear unit, defined as

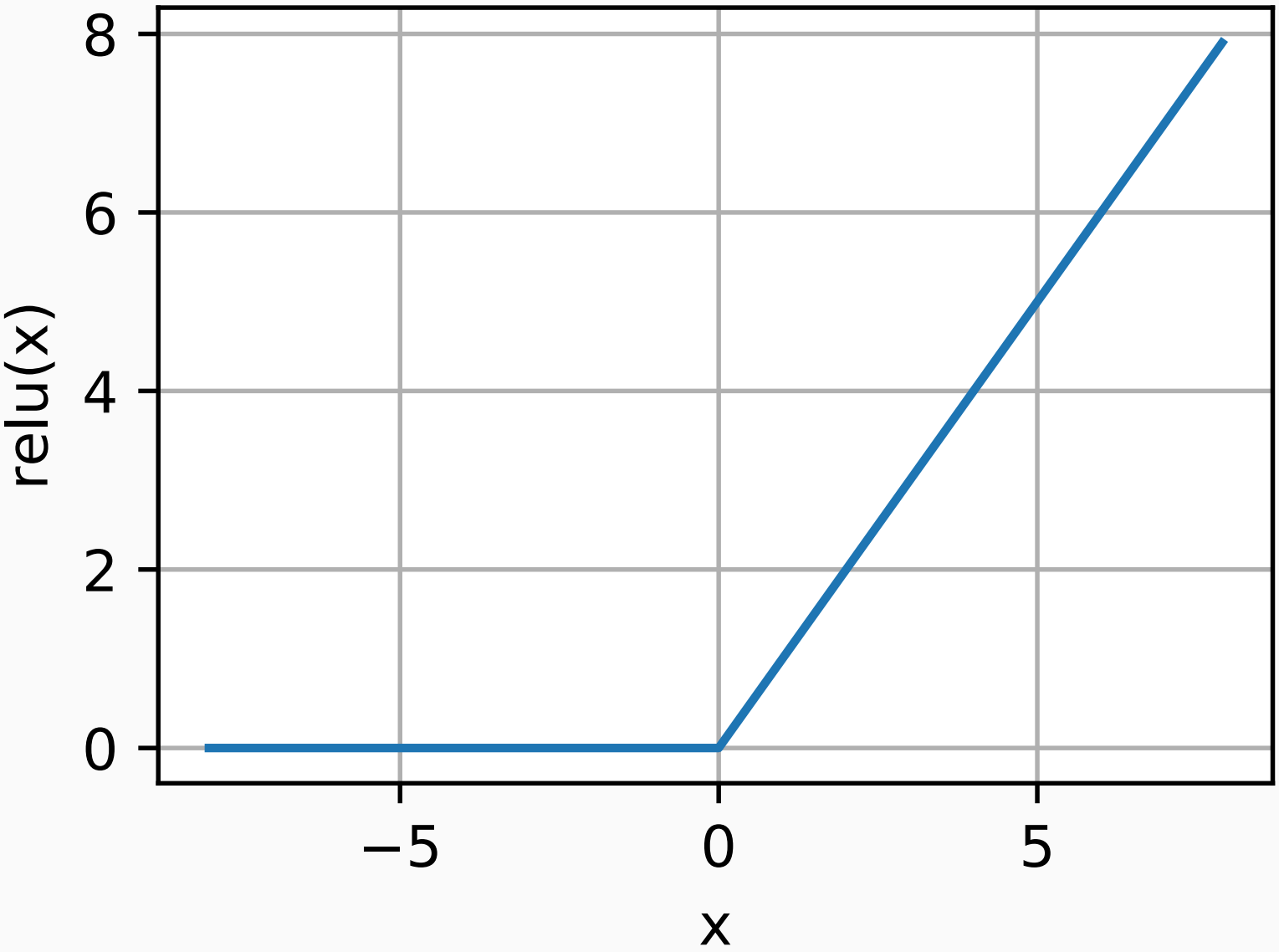
$$\text{ReLU}(x) = \max\{x, 0\}.$$

- With gradient

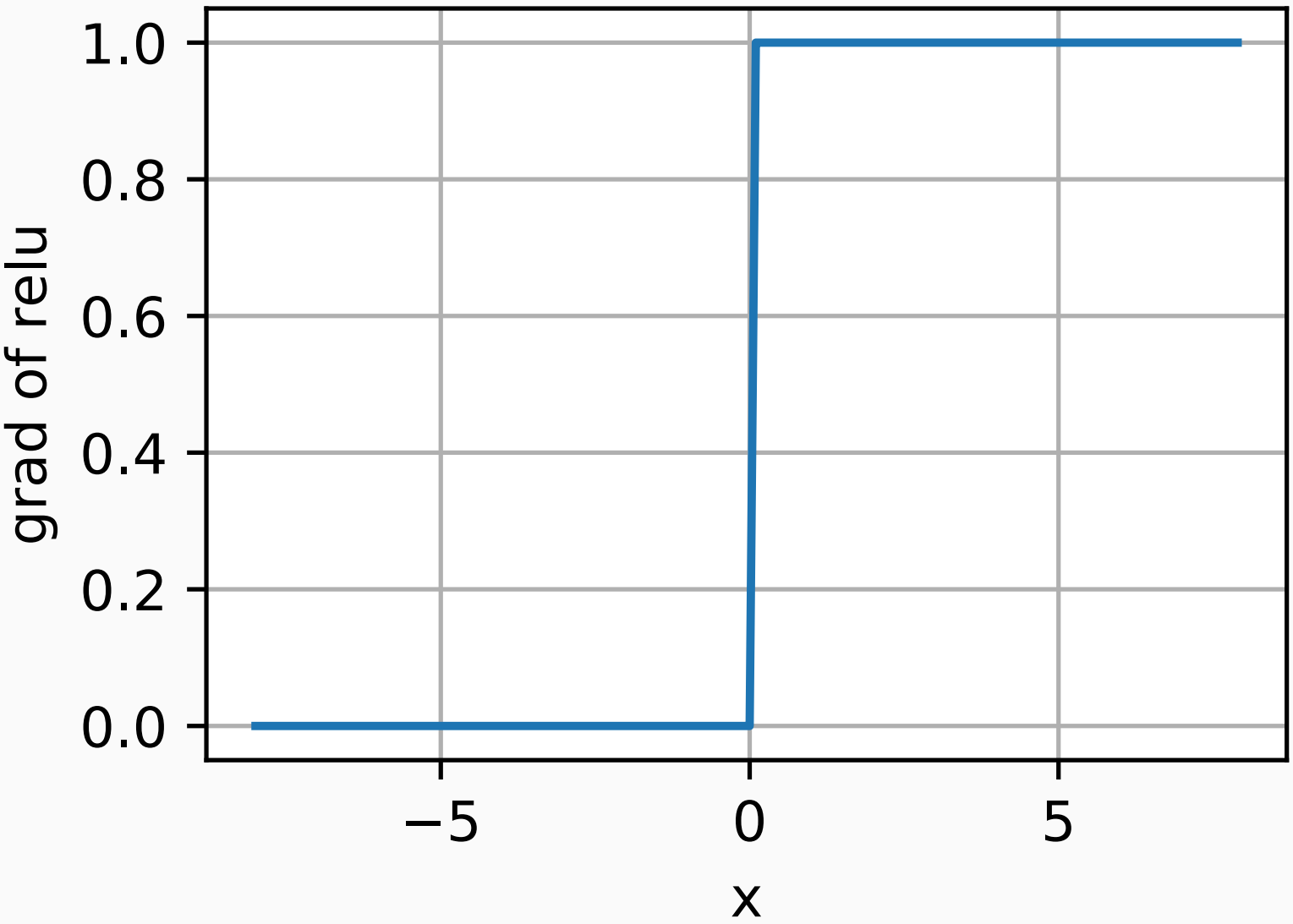
$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Doesn't suffer from saturation problems and gradients are easy to compute
- Induces sparsity when $x < 0$
- Can suffer from **dying ReLU problem**, where unit is turned off ($x < 0$) during training and weights never update

# ReLU



# ReLU



# Some Theory

- Neural networks are a sieve
- Shallow neural networks
  - They are universal approximators (Cybenko 1989, Hornik 1991)
  - Rate results exist for a variety of function spaces / smoothness classes (Chen and White 1999)
- Deep neural networks
  - DNNs achieve minimax optimal rates in Holder smoothness class (Liu, Boukai, and Shang 2019)
  - DNNs are minimax optimal in spaces with hierarchical property (Mhaskar and Poggio 2016)

$$f(x_1, \ldots, x_8) = h_3(h_{21}(h_{11}(x_1, x_2), h_{12}(x_3, x_4)), h_{22}(h_{13}(x_5, x_6), h_{14}(x_7, x_8)))$$