# EECS 498-004: Introduction to Natural Language Processing

Instructor: Prof. Lu Wang

Computer Science and Engineering

University of Michigan

https://web.eecs.umich.edu/~wangluxy/

# Thanks for your hard work and feedbacks on homework!

- We will strive for clarity!
- Things that I want to stress:
    - This course will **not rely on automated grading**. Several considerations:
        - **Coding flexibility**: Results may vary due to different choices of tools, e.g. sentence segmenters, tokenizers, etc (we're happy to make recommendations, but will not put constraints on the choice)
        - Results may also differ among submissions due to different machines (and configurations) used
        - **Personalized comments**: IAs will run your code and grade based on logics, and comment accordingly.
    - **Readings are required** (i.e. not optional).
        - Some notations are different in 3rd edition of the textbook, but shouldn't affect understanding.

# Quick polls on programming assignments

- 1. HW2: Currently we have two programming questions in HW2 (one for HMM with Viterbi implementation, the other for feedforward neural networks using existing tools). **Q**: moving the neural network question to next homework (i.e. HW3)?

- 2. HW 3&4: **Q**: reducing programming assignments (less questions) in the future homeworks?

# Outline

- Logistic Regression
- Feedforward Neural Networks
- Recurrent Neural Networks

[Some slides are borrowed from Dan Jurafsky, Hugo Larochelle, and Chris Manning]

# Logistic Regression (LogReg)

- Generative and Discriminative Classifiers
- Classification in LogReg (test)
- An example with sentiment analysis
- Learning in LogReg (training)

# Logistic Regression (LogReg)

- Generative and Discriminative Classifiers
- Classification in LogReg (test)
- An example with sentiment analysis
- Learning in LogReg (training)

# Generative and Discriminative Classifiers

- Naïve Bayes is a **generative** classifier

by contrast:

- Logistic regression is a **discriminative** classifier

# Generative and Discriminative Classifiers

Suppose we're distinguishing cat from dog images

imagenet

imagenet

# Generative Classifier:

- Build a model of what's in a cat image
  - Knows about whiskers, ears, eyes
  - Assigns a probability to any image:
    - how cat-y is this image?

Also build a model for dog images

Now given a new image:

**Run both models and see which one fits better**

# Discriminative Classifier

Just try to distinguish dogs from cats

Oh look, dogs have collars!

Let's ignore everything else

# Finding the correct class c from a document d in Generative vs Discriminative Classifiers

- Naive Bayes

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \ \overbrace{P(d|c)}^{\text{likelihood}} \ \overbrace{P(c)}^{\text{prior}}$$

- Logistic Regression

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \ \overset{\text{posterior}}{P(c|d)}$$

# Components of a probabilistic machine learning classifier

Given *m* input/output pairs $(x^{(i)}, y^{(i)})$:

1. A **feature representation** of the input. For each input observation $x(i)$, a vector of features $[x_1, x_2, \ldots, x_n]$. Feature *i* for input $x^{(j)}$ is $x_i$, or sometimes $f_i(x)$.
2. A **classification function** that computes *y*, the estimated class, via $p(y|x)$, like the **sigmoid** or **softmax** functions.
3. An objective function for learning, like **cross-entropy loss**.
4. An algorithm for optimizing the objective function: **stochastic gradient descent**.

# The two phases of Logistic Regression

- **Training**: we learn weights $w$ and $b$ using **stochastic gradient descent** and **cross-entropy loss**.

- **Test**: Given a test example $x$ we compute $p(y|x)$ using learned weights (or parameters), and return whichever label ($y = 1$ or $y = 0$) is higher probability

# Logistic Regression (LogReg)

- Generative and Discriminative Classifiers
- ➡ Classification in LogReg (test)
- An example with sentiment analysis
- Learning in LogReg (training)

# Binary Classification in Logistic Regression

- Given a series of input/output pairs:
  - $(x^{(i)}, y^{(i)})$
- For each observation $x^{(i)}$
  - We represent $x^{(i)}$ by a **feature vector** $[x_1, x_2, \ldots, x_n]$
  - We compute an output: a predicted class $\hat{y}^{(i)} \in \{0,1\}$

# Features in logistic regression

- For feature $x_i$, weight $w_i$ tells "how important is $x_i$"
    - $x_i$ ="review contains '`awesome`'": $w_i$ = `+10`
    - $x_j$ ="review contains '`abysmal`'": $w_j$ = `−10`
    - $x_k$ ="review contains '`mediocre`'": $w_k$ = `−2`

# Logistic Regression for one observation x

- Input observation: vector $x = [x_1, x_2, \ldots, x_n]$
- Weights: one per feature: $W = [w_1, w_2, \ldots, w_n]$
  - Sometimes we call the weights $\theta$
- Output: a predicted class $\hat{y} \in \{0,1\}$

# How to do classification

- For each feature $x_i$, weight $w_i$ tells us the importance of $x_i$
  - Plus we'll have a bias b
- We'll sum up all the weighted features and the bias

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

$$\Longrightarrow z = w \cdot x + b$$

- If this sum is high, we say y=1; if low, then y=0

# But we want a probabilistic classifier

- We need to formalize “sum is high”.
- We’d like a principled classifier that gives us a probability, just like Naive Bayes did
- Concretely, we want a model that can tell us:

  $p(y=1|x)$

  $p(y=0|x)$

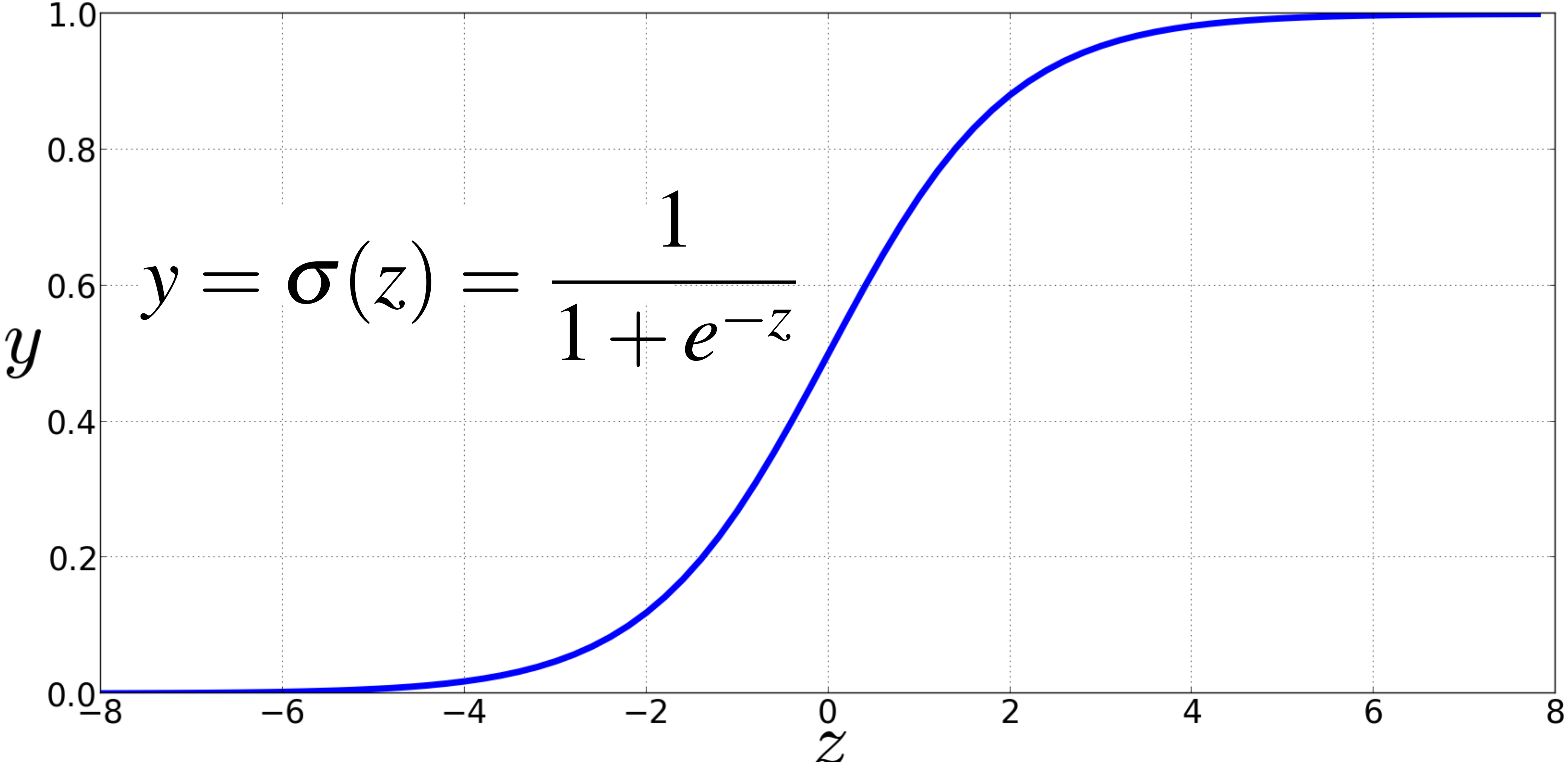
# The problem: z isn't a probability, it's just a number!

$$z = w \cdot x + b$$

- Solution: use a function of z that goes from 0 to 1

$$y = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)}$$

# The very useful **sigmoid** (or logistic) function



$$y = \sigma(z) = \frac{1}{1+e^{-z}}$$

# Idea of logistic regression

- We’ll compute $w \cdot x + b$
- And then we’ll pass it through the sigmoid function:

$$\sigma(w \cdot x + b)$$

- And we’ll just treat it as a probability

# Making probabilities with sigmoids

$$
\begin{aligned}
P(y=1) &= \sigma(w \cdot x + b) \\
&= \frac{1}{1+\exp\left(-(w \cdot x + b)\right)}
\end{aligned}
$$

$$
\begin{aligned}
P(y=0) &= 1 - \sigma(w \cdot x + b) \\
&= 1 - \frac{1}{1+\exp\left(-(w \cdot x + b)\right)} \\
&= \frac{\exp\left(-(w \cdot x + b)\right)}{1+\exp\left(-(w \cdot x + b)\right)}
\end{aligned}
$$

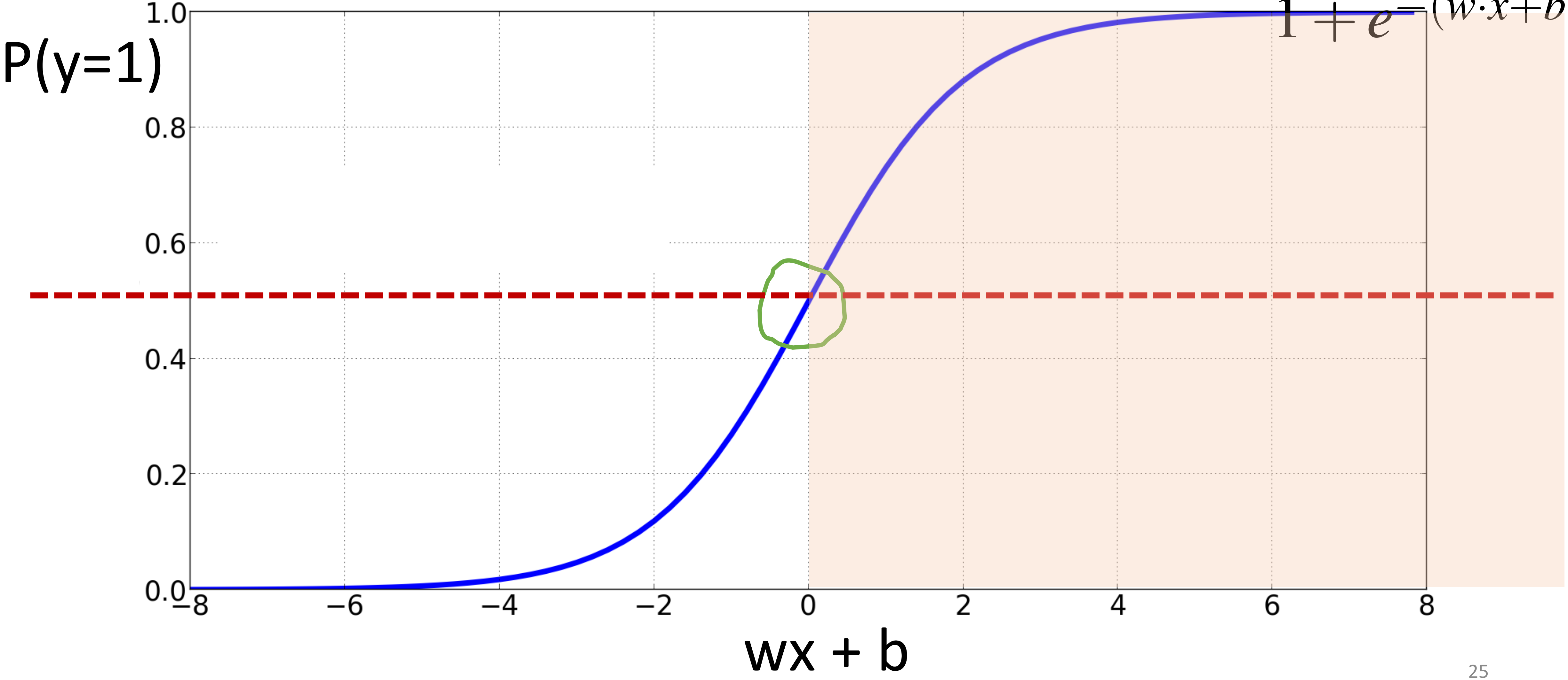
# Turning a probability into a classifier

$$\hat{y} = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

0.5 here is called the **decision boundary**

# The probabilistic classifier

$$P(y=1) = \sigma(w \cdot x + b) = \frac{1}{1+e^{-(w \cdot x+b)}}$$



# Turning a probability into a classifier

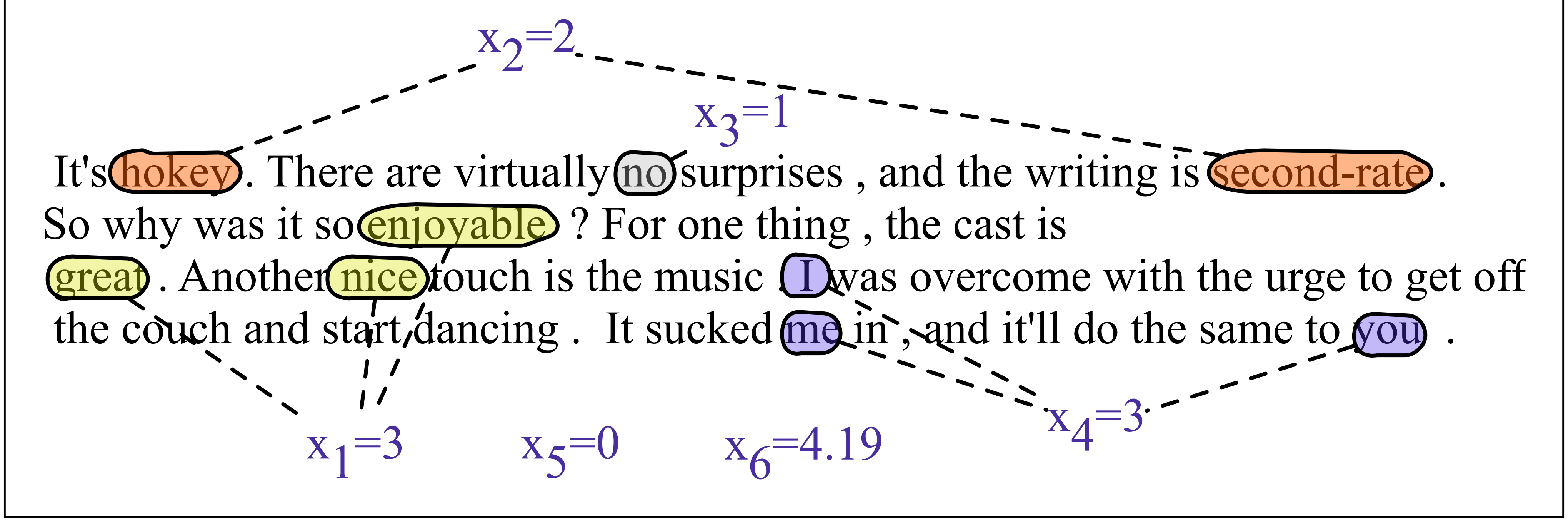
$$\hat{y} = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} \text{if } \mathrm{w}\cdot\mathrm{x}+\mathrm{b} > 0 \\ \text{if } \mathrm{w}\cdot\mathrm{x}+\mathrm{b} \leq 0 \end{array}$$

# Logistic Regression (LogReg)

- Generative and Discriminative Classifiers
- Classification in LogReg (test)
- ➡ An example with sentiment analysis
- Learning in LogReg (training)

# Sentiment example: does y=1 or y=0?

- It's hokey . There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable ? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .



It's hokey. There are virtually no surprises , and the writing is second-rate. So why was it so enjoyable ? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

| Feature | Definition | Value of the Feature |
|---|---|---|
| $x_1$ | $\text{count(positive lexicon)} \in \text{doc)}$ | 3 |
| $x_2$ | $\text{count(negative lexicon)} \in \text{doc)}$ | 2 |
| $x_3$ | $\begin{cases} 1 & \text{if “no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$ | 1 |
| $x_4$ | $\text{count}(\text{1st and 2nd pronouns} \in \text{doc})$ | 3 |
| $x_5$ | $\begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$ | 0 |
| $x_6$ | $\log(\text{word count of doc})$ | $\ln(66) = 4.19$ |

# Classifying sentiment for input x

| Feature | Definition | Value of the Feature |
|---|---|---|
| $x_1$ | $\text{count(positive lexicon)} \in \text{doc})$ | $3$ |
| $x_2$ | $\text{count(negative lexicon)} \in \text{doc})$ | $2$ |
| $x_3$ | $\begin{cases} 1 & \text{if ``no''} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$ | $1$ |
| $x_4$ | $\text{count}(\text{1st and 2nd pronouns} \in \text{doc})$ | $3$ |
| $x_5$ | $\begin{cases} 1 & \text{if ``!''} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$ | $0$ |
| $x_6$ | $\log(\text{word count of doc})$ | $\ln(66) = 4.19$ |

Suppose $w = [2.5, -5.0, -1.2, 0.5, 2.0, 0.7]$ $b = 0.1$

# Classifying sentiment for input x

Suppose $w = [2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \quad b = 0.1$

$$
\begin{aligned}
p(+|x) = P(Y = 1|x) &= \sigma(w \cdot x + b) \\
&= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\
&= \sigma(.833) \\
&= 0.70
\end{aligned}
$$

$$
\begin{aligned}
p(-|x) = P(Y = 0|x) &= 1 - \sigma(w \cdot x + b) \\
&= 0.30
\end{aligned}
$$

# Classification in (**binary**) logistic regression: summary

- Given:
  - a set of classes: (+ sentiment,- sentiment)
  - a vector **x** of features `[`$x_1$`, `$x_2$`, …, `$x_n$`]`
    - $x_1$= count( "awesome")
    - $x_2$ = log(number of words in review)
  - A vector **w** of weights `[`$w_1$`, `$w_2$`, …, `$w_n$`]`
    - $w_i$ for each feature $f_i$

$$
\begin{aligned}
P(y=1) &= \sigma(w \cdot x + b) \\
&= \frac{1}{1+e^{-(w \cdot x + b)}}
\end{aligned}
$$

# Logistic Regression (LogReg)

- Generative and Discriminative Classifiers
- Classification in LogReg (test)
- An example with sentiment analysis
- ➡ Learning in LogReg (training)

# Wait, where did the W's come from?

- Supervised classification: we know the correct label *y* (either 0 or 1) for each *x*.
- What the system produces is an estimate, $\hat{y}$
- We want to set *w* and *b* to minimize the **distance** between our estimate $\hat{y}^{(i)}$ and the true $y^{(i)}$.

# Wait, where did the W's come from?

- Supervised classification: we know the correct label *y* (either 0 or 1) for each *x*.
- What the system produces is an estimate, $\hat{y}$
- We want to set *w* and *b* to minimize the **distance** between our estimate $\hat{y}^{(i)}$ and the true $y^{(i)}$.
  - We need a distance estimator: a **loss function** or a **cost function**
  - We need an **optimization algorithm** to update *w* and *b* to minimize the loss.

# Learning components

- A loss function:
  - **cross-entropy loss**
- An optimization algorithm:
  - **stochastic gradient descent** (not covered in the lecture)

# The distance between $\hat{y}$ and y

We want to know how far is the classifier output:

$\hat{y} = \sigma(w \cdot x + b)$

from the true output:

y [= either 0 or 1]

We'll call this difference:

$L(\hat{y}, y)$ = how much $\hat{y}$ differs from the true $y$

# Cross-entropy loss

- We choose the parameters *w*,*b* that maximize the probability of the true *y* labels in the training data given the observations *x*

# Deriving cross-entropy loss for a single observation x

- **Goal**: maximize probability of the correct label *p*(*y*|*x*)
- Since there are only 2 discrete outcomes (0 or 1) we can express the probability *p*(*y*|*x*) from our classifier (the thing we want to maximize) as

$$p(y|x) \;=\; \hat{y}^{y}\,(1-\hat{y})^{1-y}$$

- if the gold-standard label y=1, this simplifies to $\hat{y}$ (*the predicted probability for x having a label of 1*)
- if the gold-standard label y=0, this simplifies to 1– $\hat{y}$

# Deriving cross-entropy loss for a single observation x

**Goal**: maximize probability of the correct label $p(y|x)$

Maximize:

$$p(y|x) = \hat{y}^{y}(1-\hat{y})^{1-y}$$

- Now take the log of both sides (mathematically handy)

Maximize:

$$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^{y}(1-\hat{y})^{1-y}\right]\\ &= y\log\hat{y}+(1-y)\log(1-\hat{y})\end{aligned}$$

- Whatever values maximize log p(y|x) will also maximize p(y|x)

# Deriving cross-entropy loss for a single observation x

**Goal**: maximize probability of the correct label $p(y|x)$

Maximize:
$$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^{y}\,(1-\hat{y})^{1-y}\right]\\ &= y\log\hat{y}+(1-y)\log(1-\hat{y})\end{aligned}$$

- Now flip sign to turn this into a **loss**: something to **minimize**

**Cross-entropy loss**

$$L_{\mathrm{CE}}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y}+(1-y)\log(1-\hat{y})]$$

- Or, plugging in definition of $\hat{y}$:

$$L_{\mathrm{CE}}(\hat{y},y) \;=\; -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))]$$

# Let's see if this works for our sentiment example

- We want loss to be:

  smaller if the model's prediction is close to the correct label

  bigger if model is confused

- Let's first suppose the true label of this is y=1 (positive)

It's hokey . There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable ? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

# Let's see if this works for our sentiment example

- True value is y=1. How well is our model doing?

$$
\begin{aligned}
p(+|x) = P(Y=1|x) &= \sigma(w \cdot x + b) \\
&= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\
&= \sigma(.833) \\
&= 0.70
\end{aligned}
$$

- Pretty well! What's the loss?

$$
\begin{aligned}
L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(w \cdot x + b) + (1-y) \log (1 - \sigma(w \cdot x + b))] \\
&= -[\log \sigma(w \cdot x + b)] \\
&= -\log(.70) \\
&= .36
\end{aligned}
$$

# Let's see if this works for our sentiment example

- Suppose true value instead was y=0.

$$
\begin{aligned}
p(-|x) = P(Y=0|x) &= 1-\sigma(w\cdot x+b) \\
&= 0.30
\end{aligned}
$$

- What's the loss?

$$
\begin{aligned}
L_{\mathrm{CE}}(\hat{y},y) &= -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \\
&= -[\log(1-\sigma(w\cdot x+b))] \\
&= -\log(.30) \\
&= 1.2
\end{aligned}
$$

# Let's see if this works for our sentiment example

- The loss when model was right (if true y=1)

$$\begin{aligned} L_{\mathrm{CE}}(\hat{y},y) &= -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \\ &= -[\log\sigma(w\cdot x+b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

- Is lower than the loss when model was wrong (if true y=0):

$$\begin{aligned} L_{\mathrm{CE}}(\hat{y},y) &= -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \\ &= -[\log(1-\sigma(w\cdot x+b))] \\ &= -\log(.30) \\ &= 1.2 \end{aligned}$$

- Sure enough, loss was bigger when model was wrong!

# Our goal: minimize the loss

- Let's make explicit that the loss function in parameterized by weights $\theta$=(w,b)
- We want the weights that minimize the loss, averaged over all examples:

$$\hat{\theta} \;=\; \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)};\theta), y^{(i)})$$

# Problem of Overfitting

- A model that perfectly match the training data has a problem.
- It will also **overfit** to the data, e.g.,
  - A random word that perfectly predicts *y* (it happens to only occur in one class) will get a very high weight.
  - Failing to generalize to a test set without this word.
- A good model should be able to **generalize.**

# Problem of Overfitting

+

This movie drew me in, and it'll do the same to you.

–

I can't tell you how much I hated this movie. It sucked.

Useful or harmless features

X1 = "this"
X2 = "movie
X3 = "hated"
X4 = "drew me in"

4gram features that just "memorize" training set and might cause problems

X5 = "the same to you"
X6 = "tell you how much"

# Problem of Overfitting

- 4-gram model on tiny data will just memorize the data
  - 100% accuracy on the training set
- But it will be surprised by the novel 4-grams in the test data
  - Low accuracy on test set
- Models that are too powerful can **overfit** the data
  - Fitting the details of the training data so exactly that the model doesn't generalize well to the test set
    - How to avoid overfitting?
      - Regularization in logistic regression
      - Dropout in neural networks

# Regularization

- A solution for overfitting
- Add a regularization term *R*(θ) to the loss function (for now written as maximizing log probability rather than minimizing loss)

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta)$$

- Idea: choose an *R*(θ) that penalizes large weights
  - fitting the data well with lots of big weights not as good as fitting the data a little less well, with small weights

# L2 Regularization

- The sum of the squares of the weights: **L2 norm** $||\theta||_2$
    - i.e., the square of the **Euclidean distance** of $\theta$ to the origin.

- L2 regularized objective function:

$$R(\theta) \;=\; ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2$$

$$\hat{\theta} \;=\; \underset{\theta}{\operatorname{argmax}} \left[ \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} \theta_j^2$$

# L1 Regularization

- The sum of the absolute value of the weights: **L1 norm** $||\theta||_1$

- L1 regularized objective function:

$$R(\theta) \ = \ ||\theta||_1 = \sum_{i=1}^{n} |\theta_i|$$

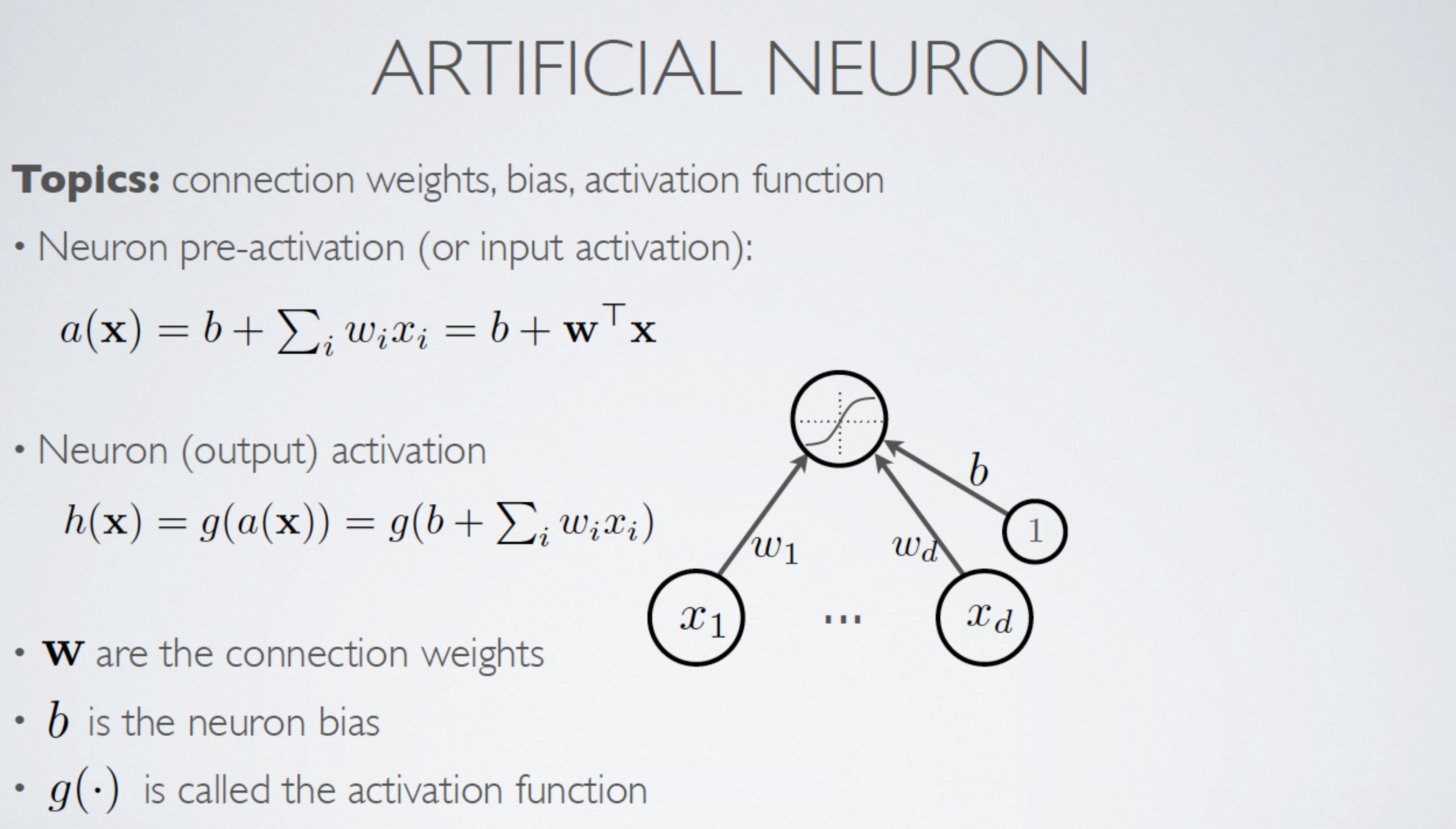
$$\hat{\theta} \ = \ \underset{\theta}{\operatorname{argmax}} \left[ \sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j|$$

# Outline

- Logistic Regression
- ➡ Feedforward Neural Networks
- Recurrent Neural Networks

# Neural Network Learning

- Learning approach based on modeling adaptation in biological neural systems.
- Perceptron: Initial algorithm for learning simple neural networks (single layer) developed in the 1950's.
- Backpropagation: More complex algorithm for learning multi-layer neural networks developed in the 1980's. (not required for this class)



# ARTIFICIAL NEURON

**Topics:** connection weights, bias, activation function

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron (output) activation

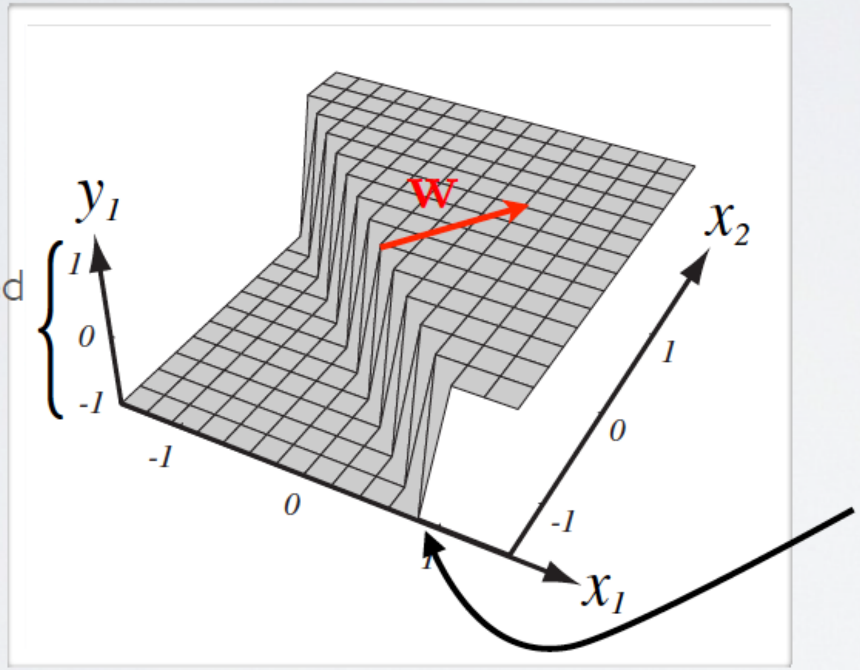
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

- $\mathbf{w}$ are the connection weights
- $b$ is the neuron bias
- $g(\cdot)$ is called the activation function

# ARTIFICIAL NEURON

**Topics:** connection weights, bias, activation function

range determined by $g(\cdot)$



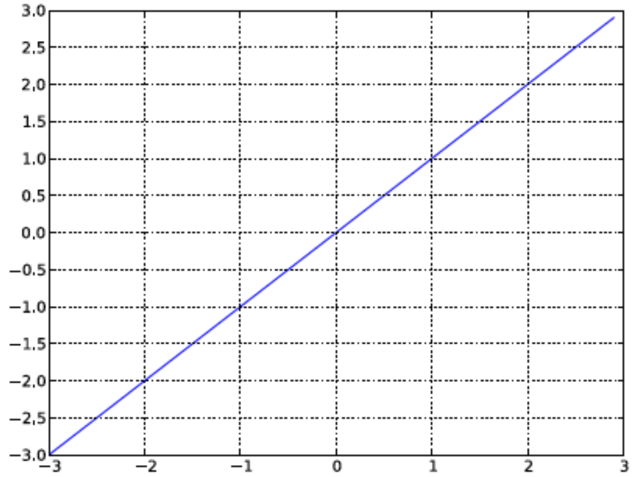
bias $b$ only changes the position of the riff

(from Pascal Vincent's slides)

# ACTIVATION FUNCTION

**Topics:** linear activation function

- Performs no input squashing
- Not very interesting...

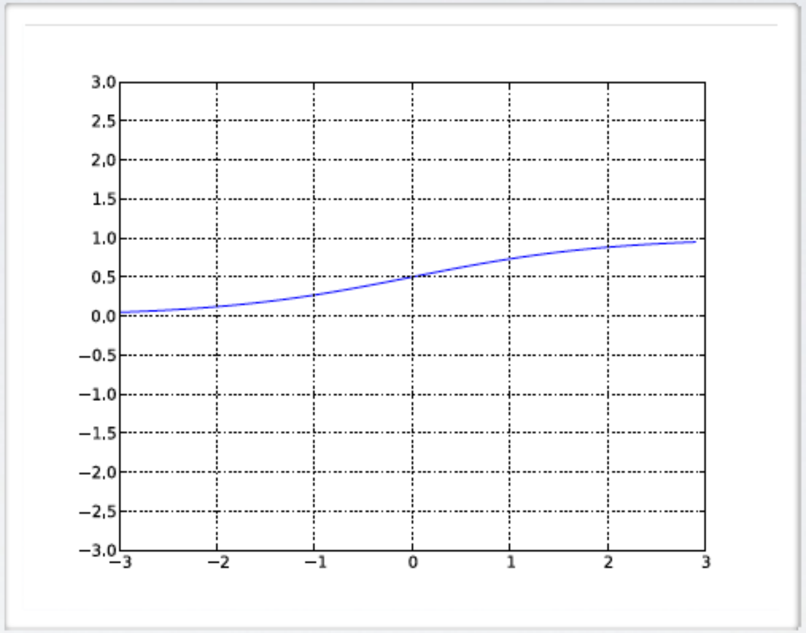


$$g(a) = a$$

# ACTIVATION FUNCTION

**Topics:** sigmoid activation function

- Squashes the neuron's pre-activation between 0 and 1
- Always positive
- Bounded
- Strictly increasing

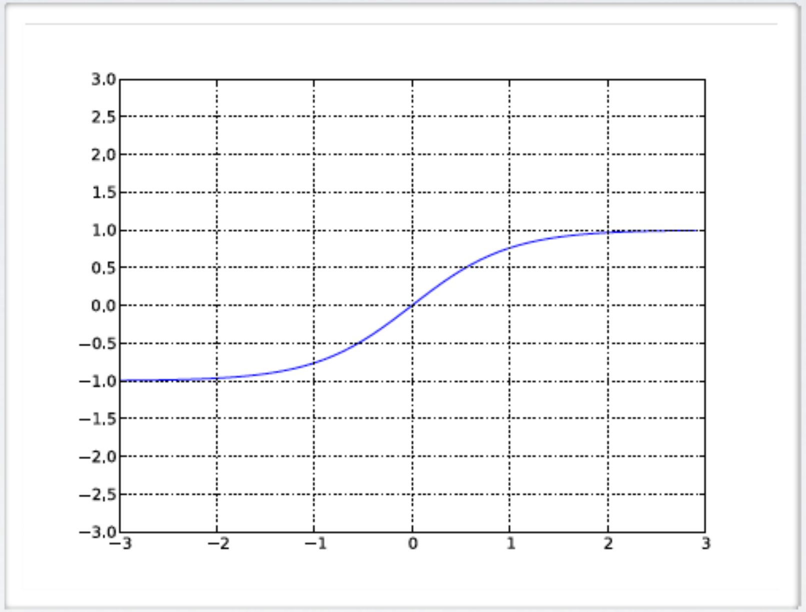


$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

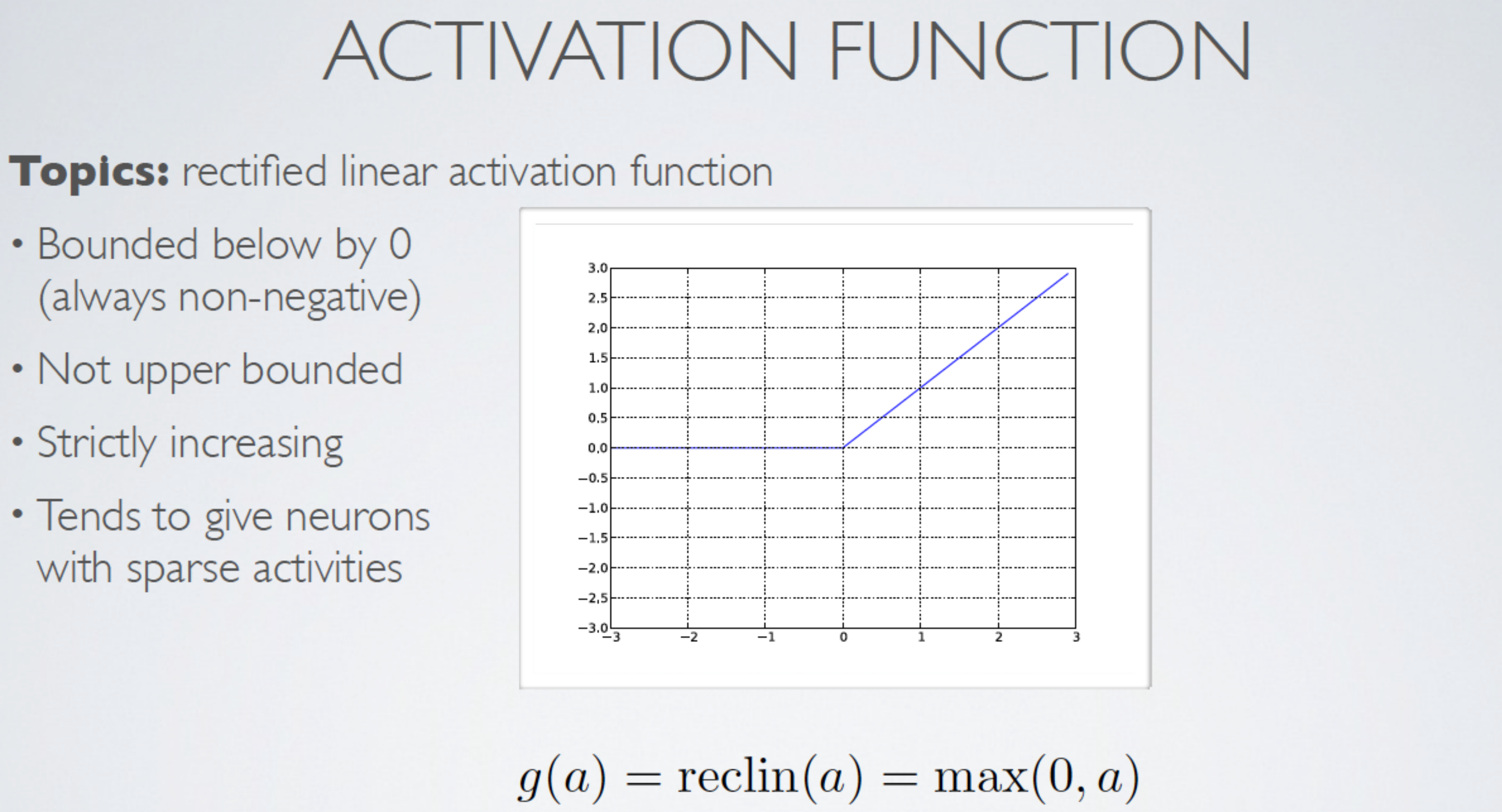
# ACTIVATION FUNCTION

**Topics:** hyperbolic tangent ("tanh") activation function

- Squashes the neuron's pre-activation between -1 and 1
- Can be positive or negative
- Bounded
- Strictly increasing



$$g(a) = \tanh(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = \frac{\exp(2a)-1}{\exp(2a)+1}$$

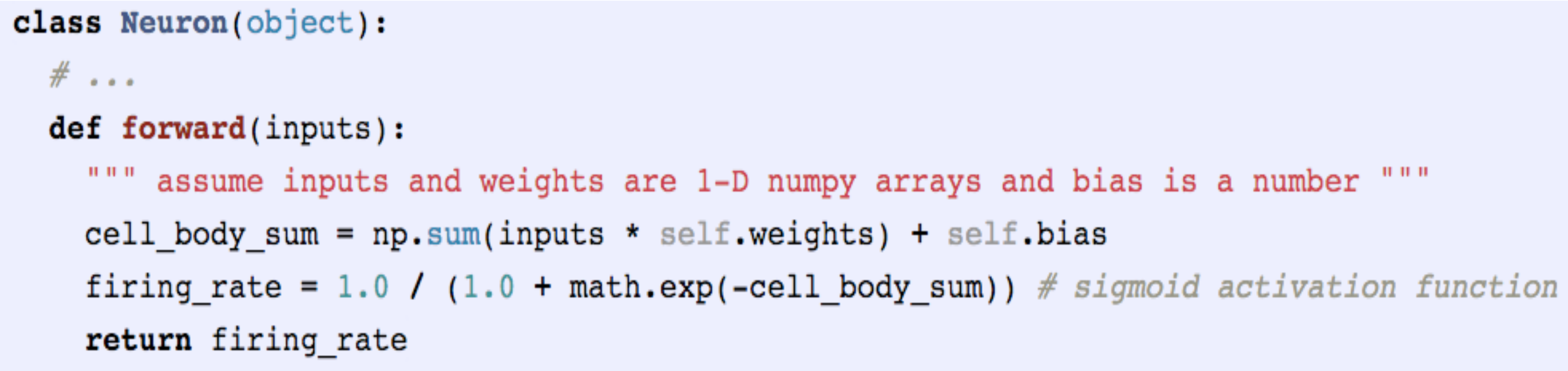


# ACTIVATION FUNCTION

**Topics:** rectified linear activation function

- Bounded below by 0 (always non-negative)
- Not upper bounded
- Strictly increasing
- Tends to give neurons with sparse activities

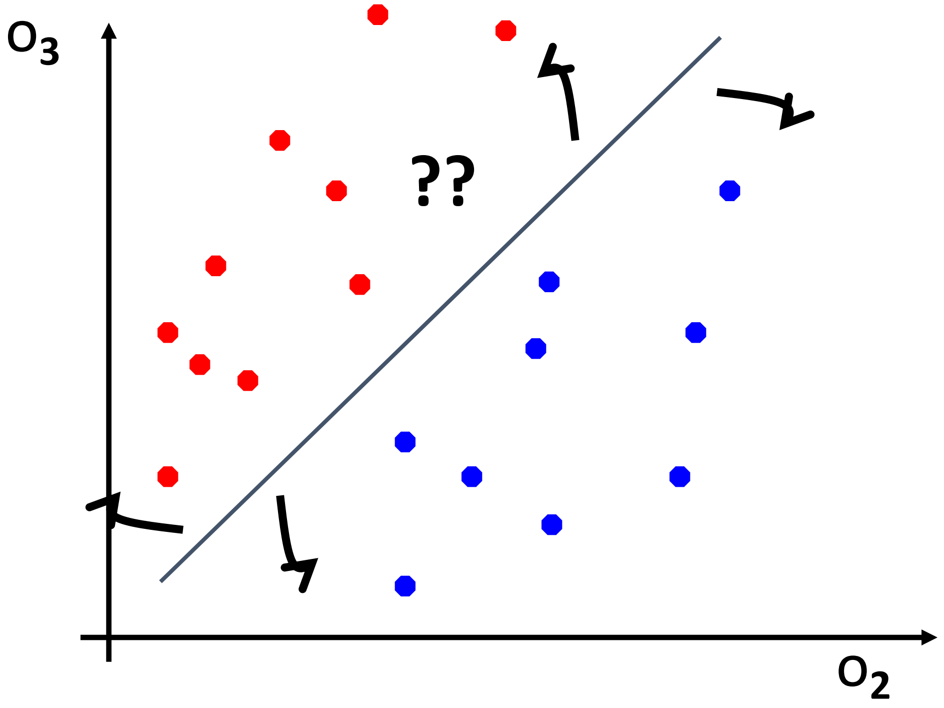
$$g(a) = \mathrm{reclin}(a) = \max(0, a)$$



```python
class Neuron(object):
  # ...
  def forward(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

# Linear Separator

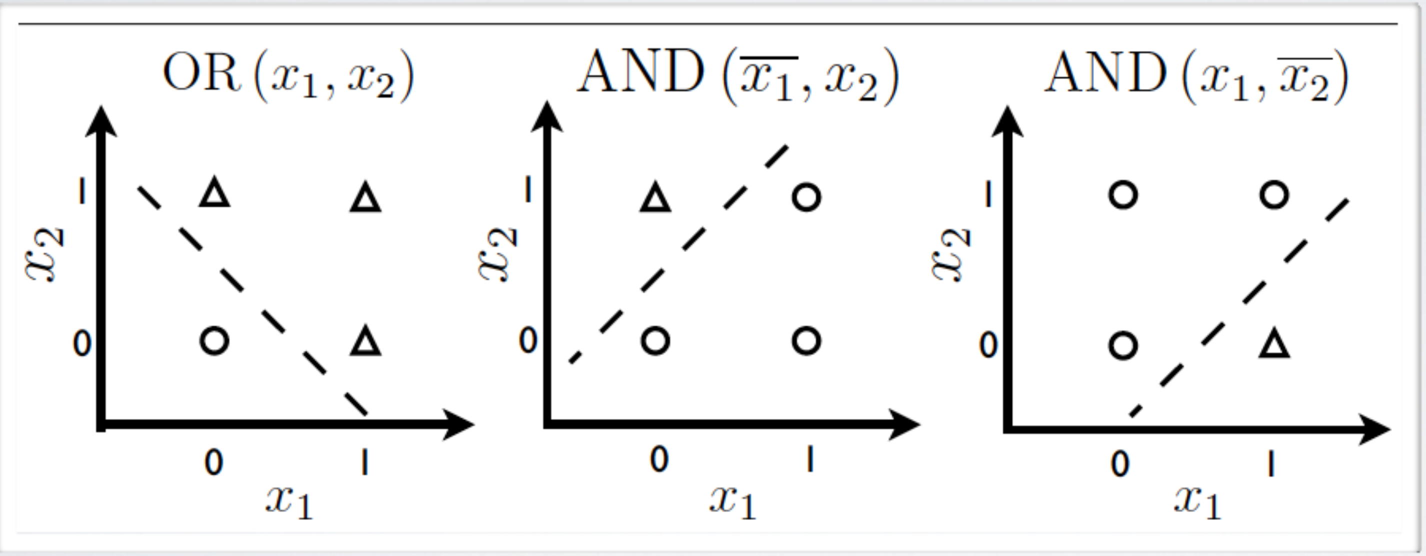
- Since one-layer neuron (aka perceptron) uses linear threshold function, it is searching for a linear separator that discriminates the classes.



# ARTIFICIAL NEURON

**Topics:** capacity of single neuron

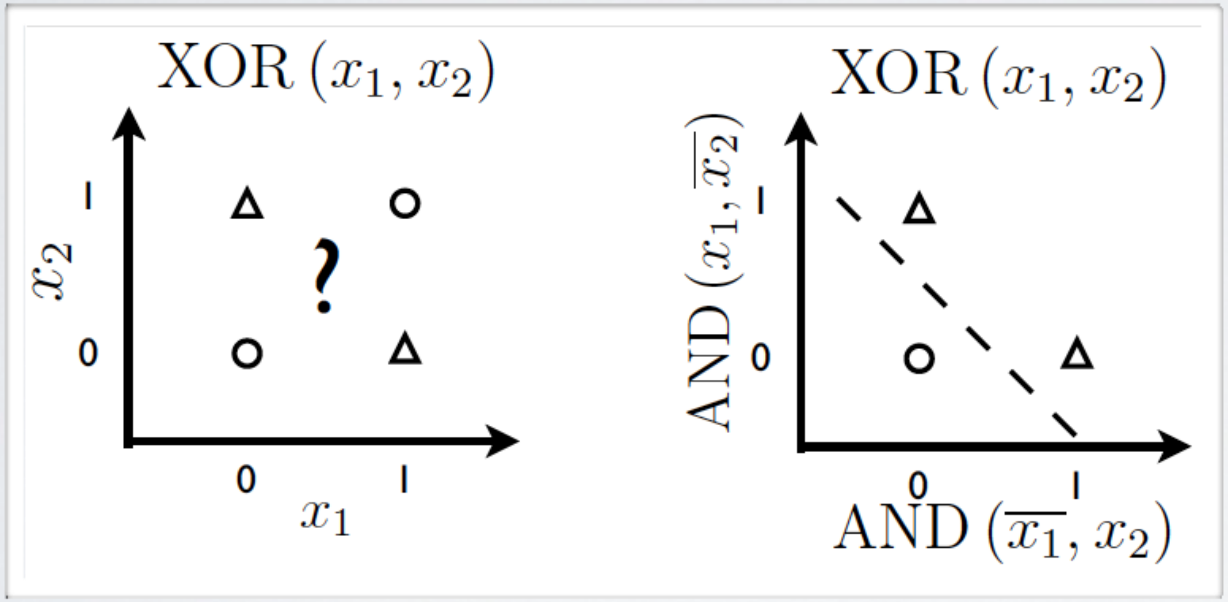
- Can solve linearly separable problems



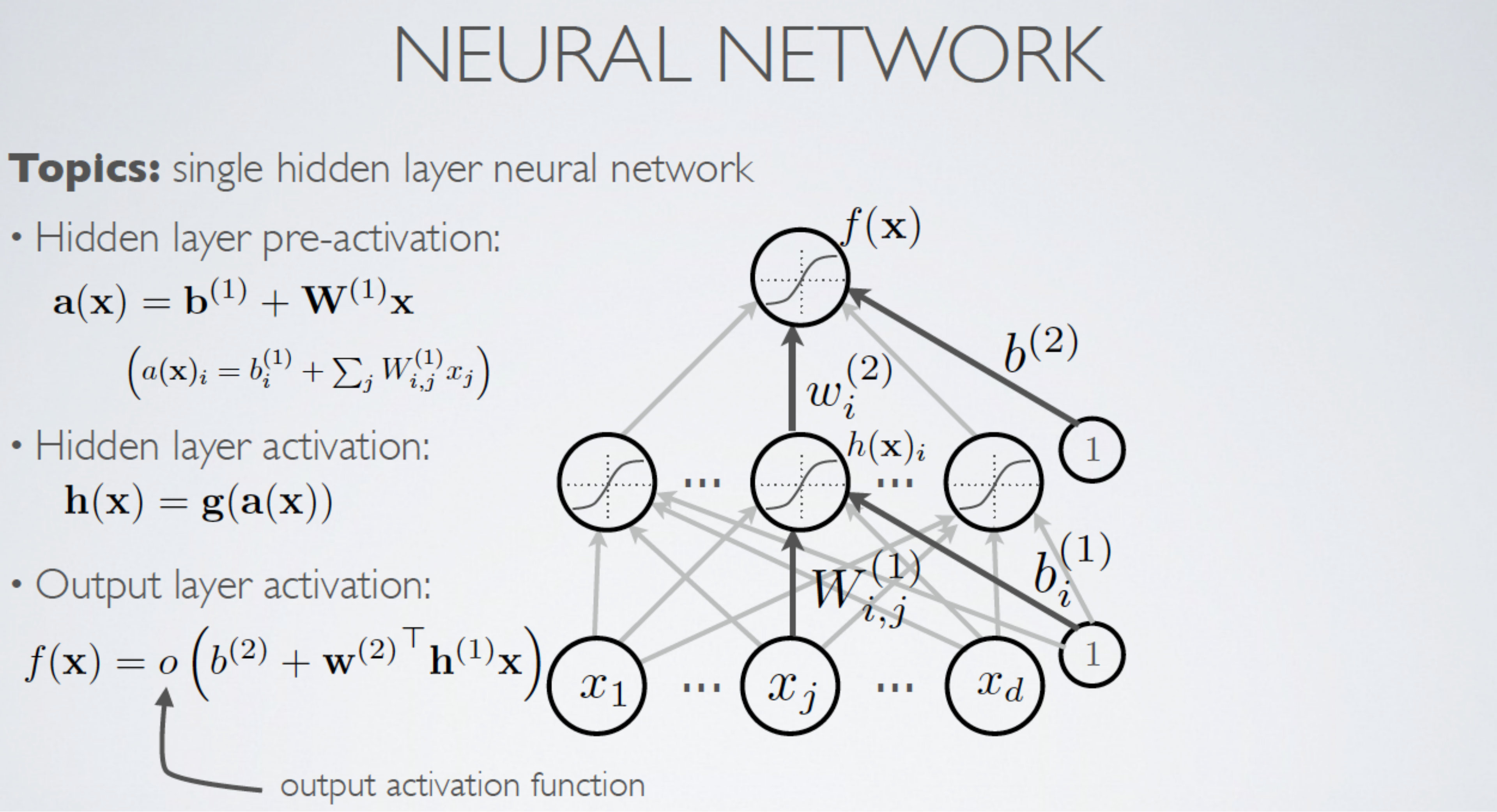
# ARTIFICIAL NEURON

**Topics:** capacity of single neuron

- Can't solve non linearly separable problems...



- ... unless the input is transformed in a better representation



# NEURAL NETWORK

**Topics:** single hidden layer neural network

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

$$\left(a(\mathbf{x})_i = b_i^{(1)} + \textstyle\sum_j W_{i,j}^{(1)} x_j\right)$$

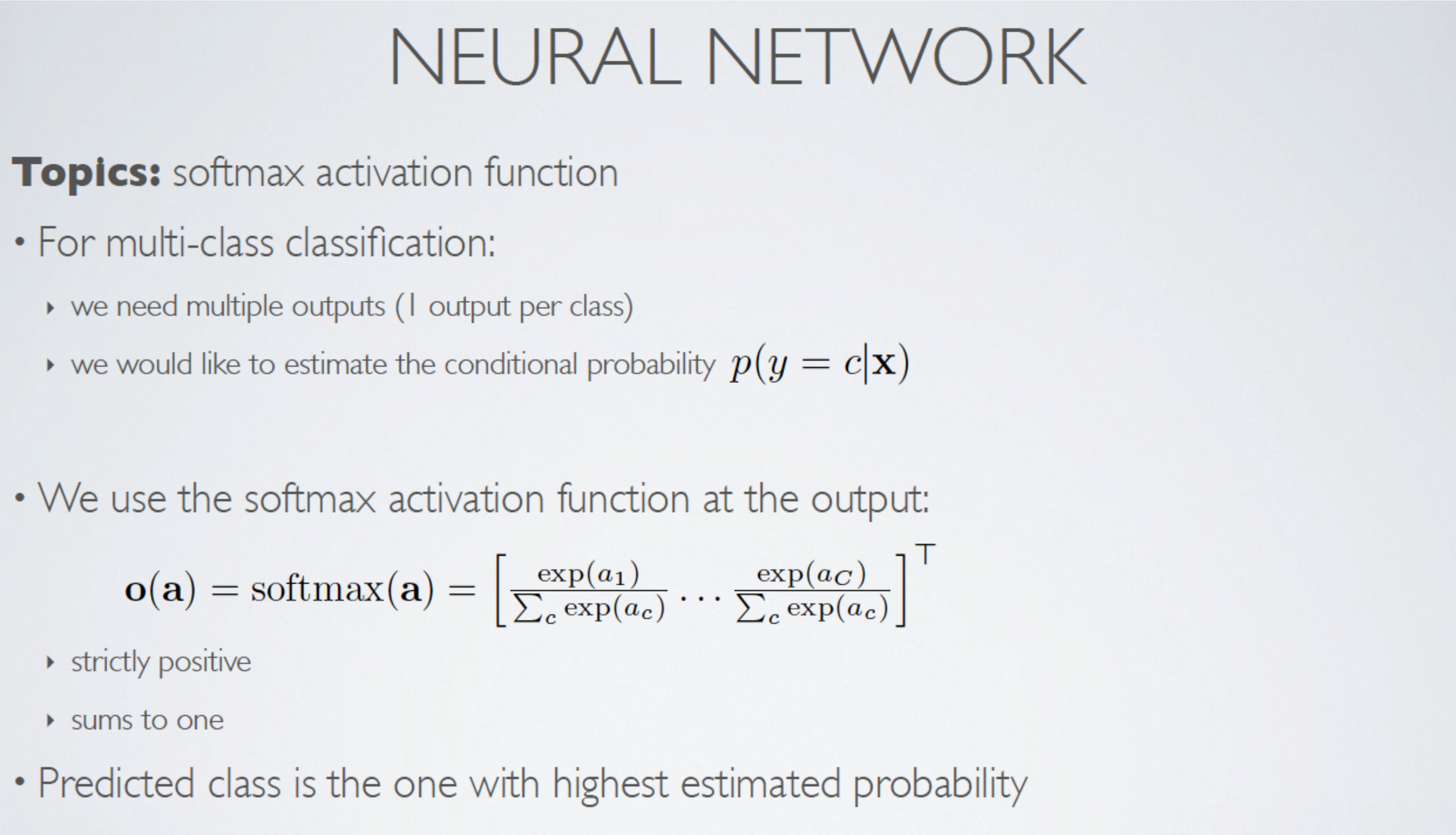
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o\left(b^{(2)} + \mathbf{w}^{(2)^\top}\mathbf{h}^{(1)}\mathbf{x}\right)$$

output activation function



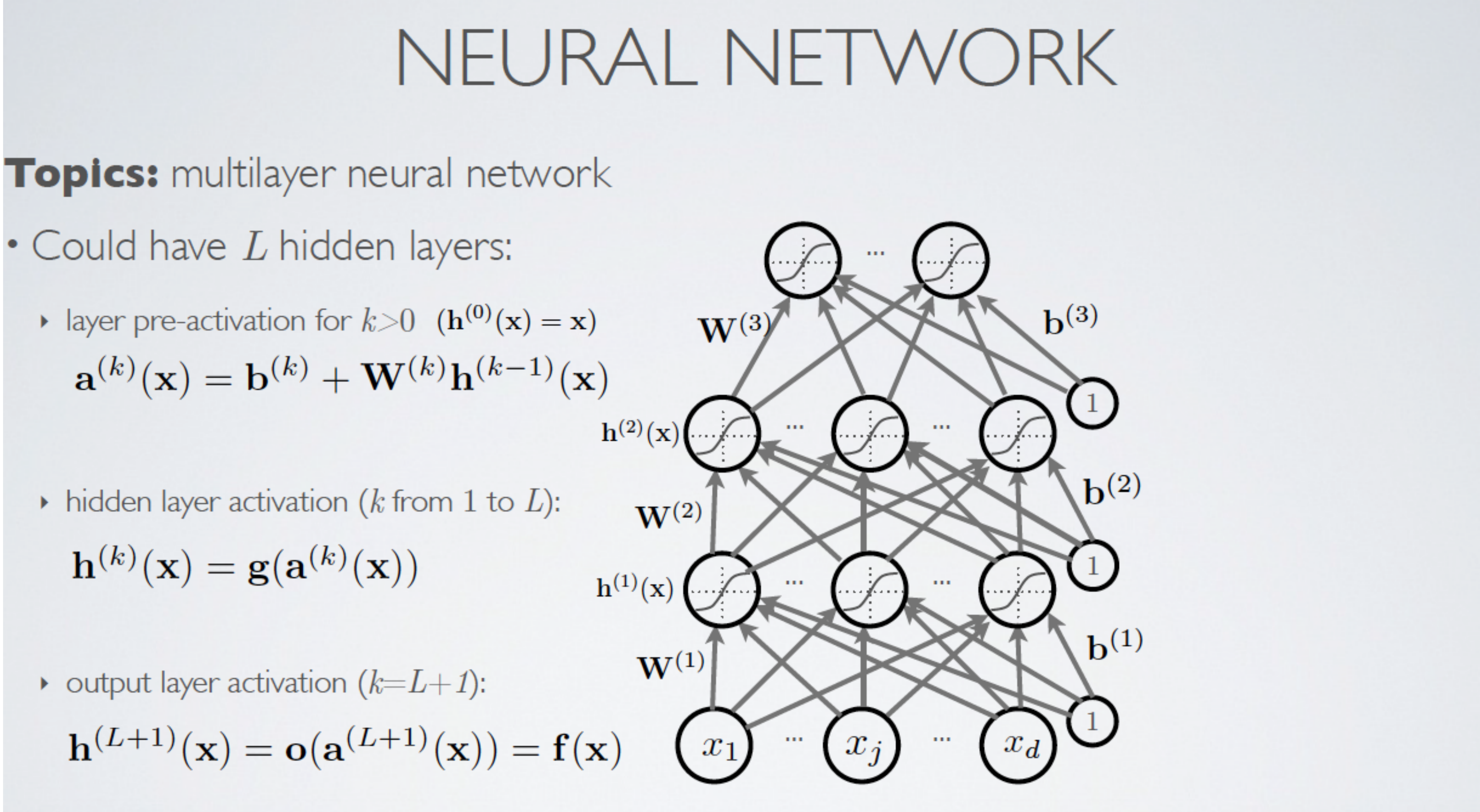
# NEURAL NETWORK

**Topics:** softmax activation function

- For multi-class classification:
  - we need multiple outputs (1 output per class)
  - we would like to estimate the conditional probability $p(y = c|\mathbf{x})$
- We use the softmax activation function at the output:

$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)} \ldots \frac{\exp(a_C)}{\sum_c \exp(a_c)}\right]^\top$$

  - strictly positive
  - sums to one
- Predicted class is the one with highest estimated probability



# NEURAL NETWORK

**Topics:** multilayer neural network

- Could have $L$ hidden layers:
  - layer pre-activation for $k{>}0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

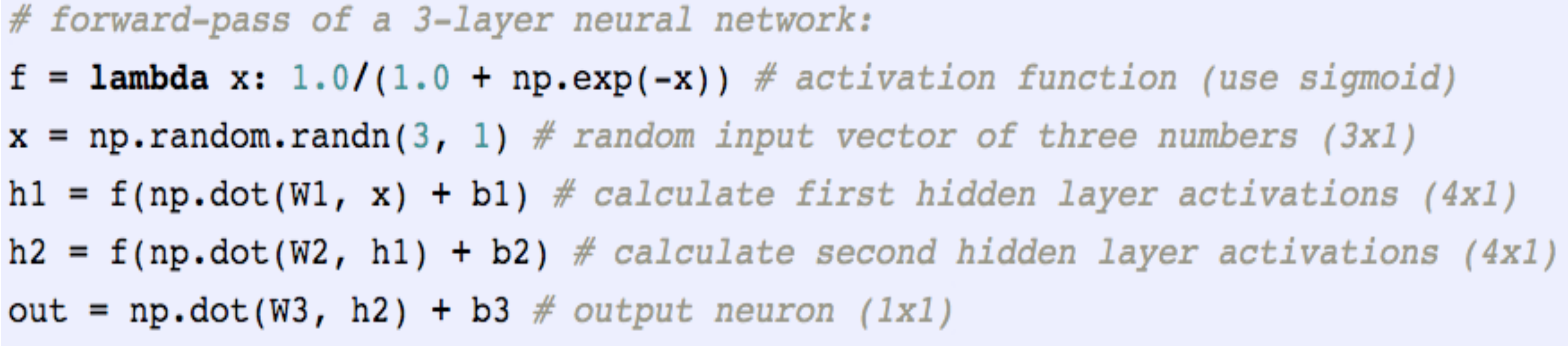
    $$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation ($k$ from 1 to $L$):

    $$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

  - output layer activation ($k{=}L{+}1$):

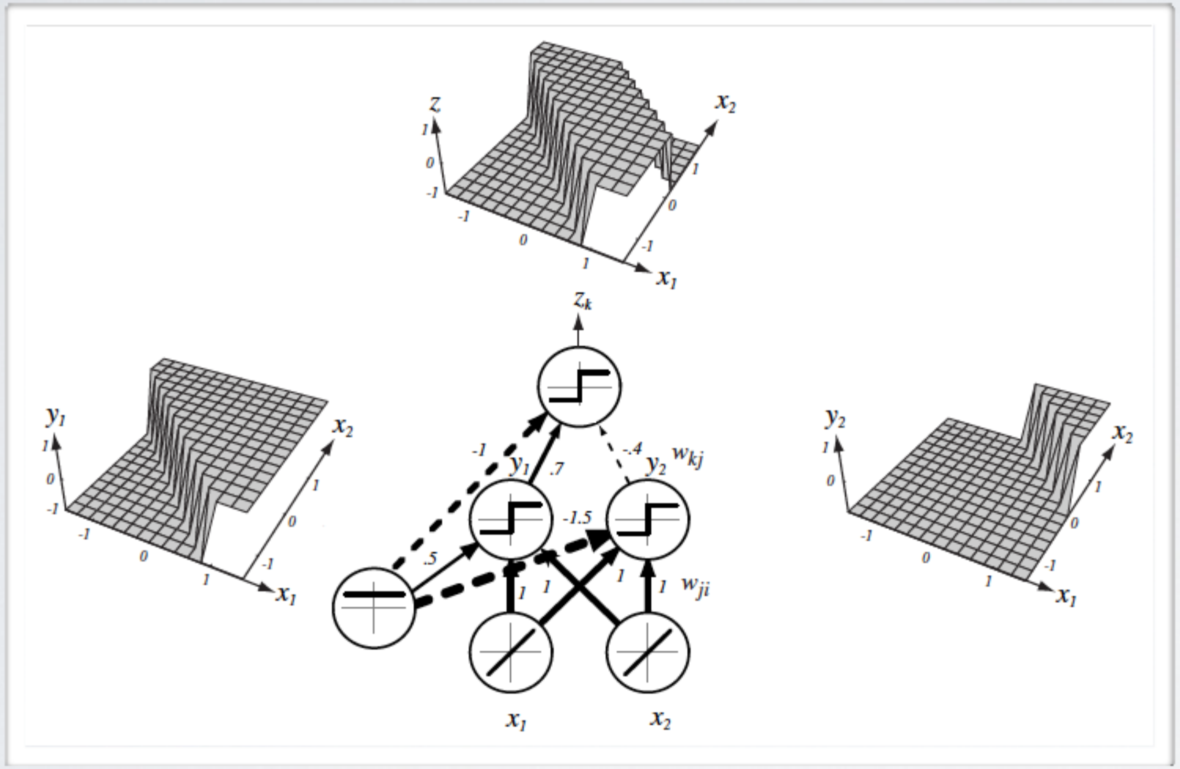
    $$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

# CAPACITY OF NEURAL NETWORK

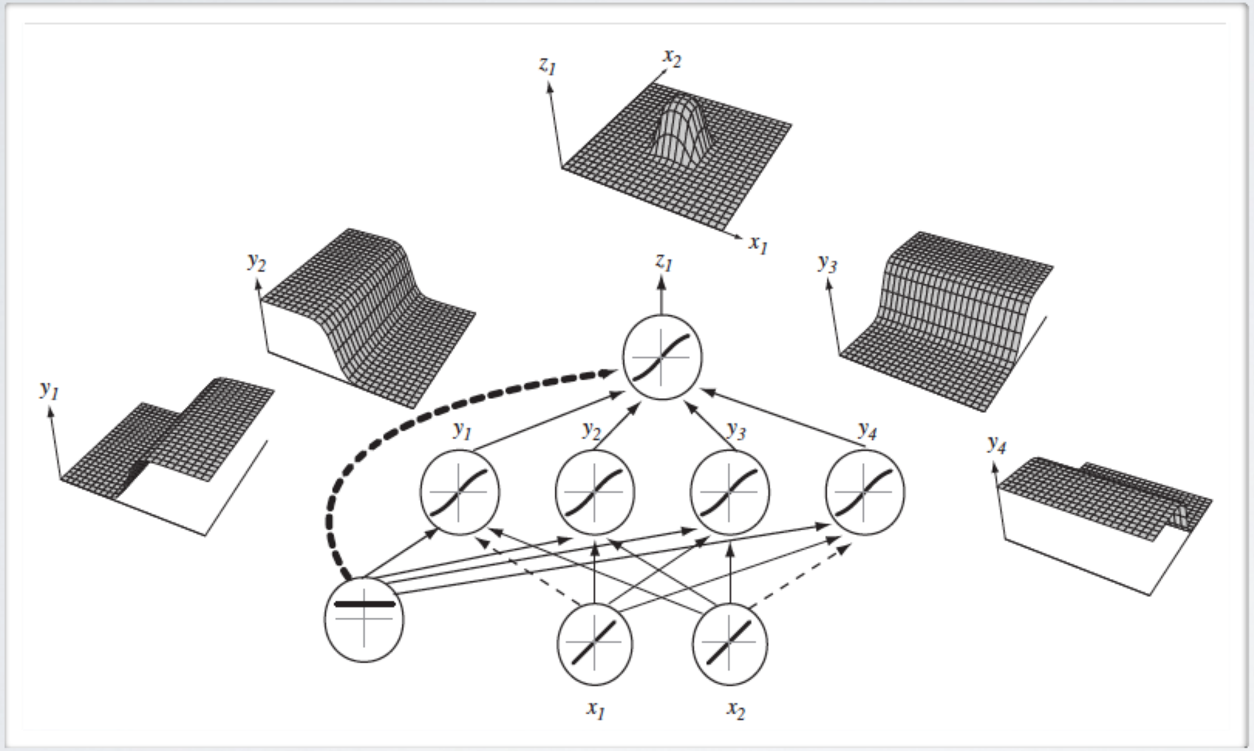
**Topics:** single hidden layer neural network



(from Pascal Vincent's slides)

# CAPACITY OF NEURAL NETWORK

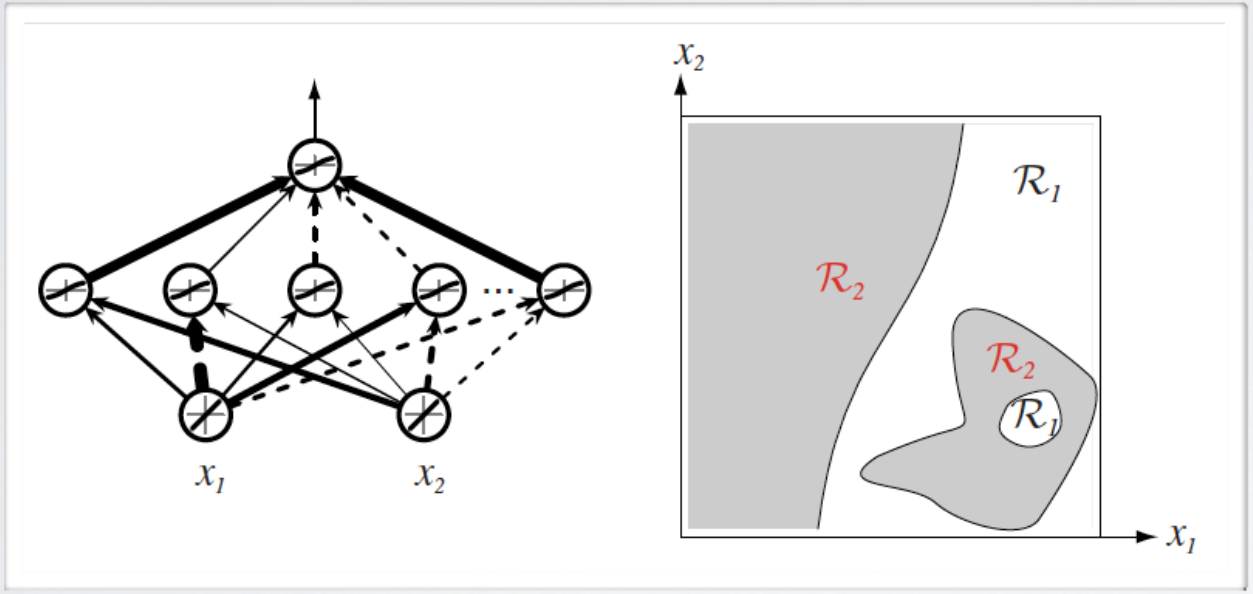
**Topics:** single hidden layer neural network



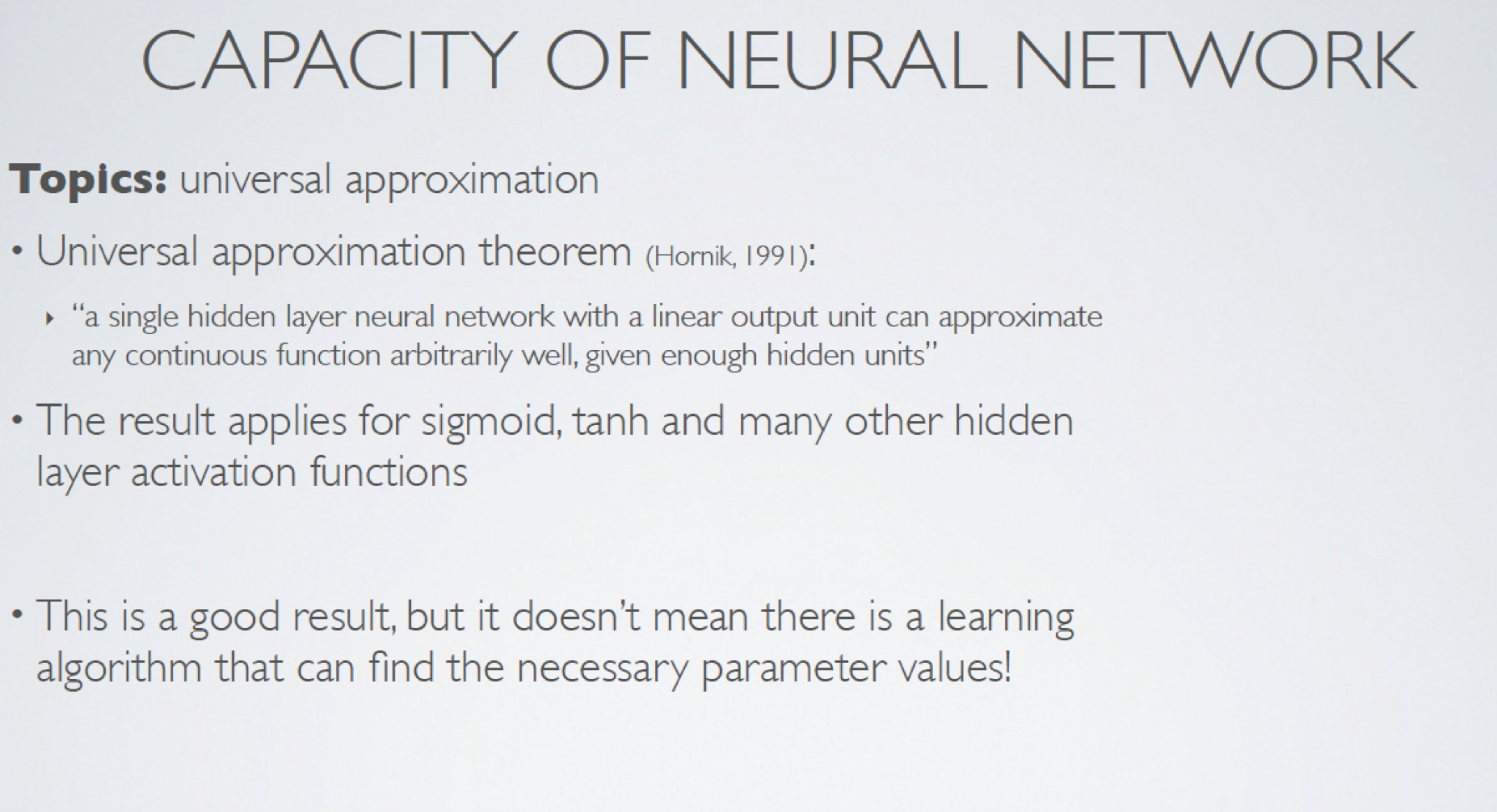
(from Pascal Vincent's slides)

# CAPACITY OF NEURAL NETWORK

**Topics:** single hidden layer neural network



(from Pascal Vincent's slides)

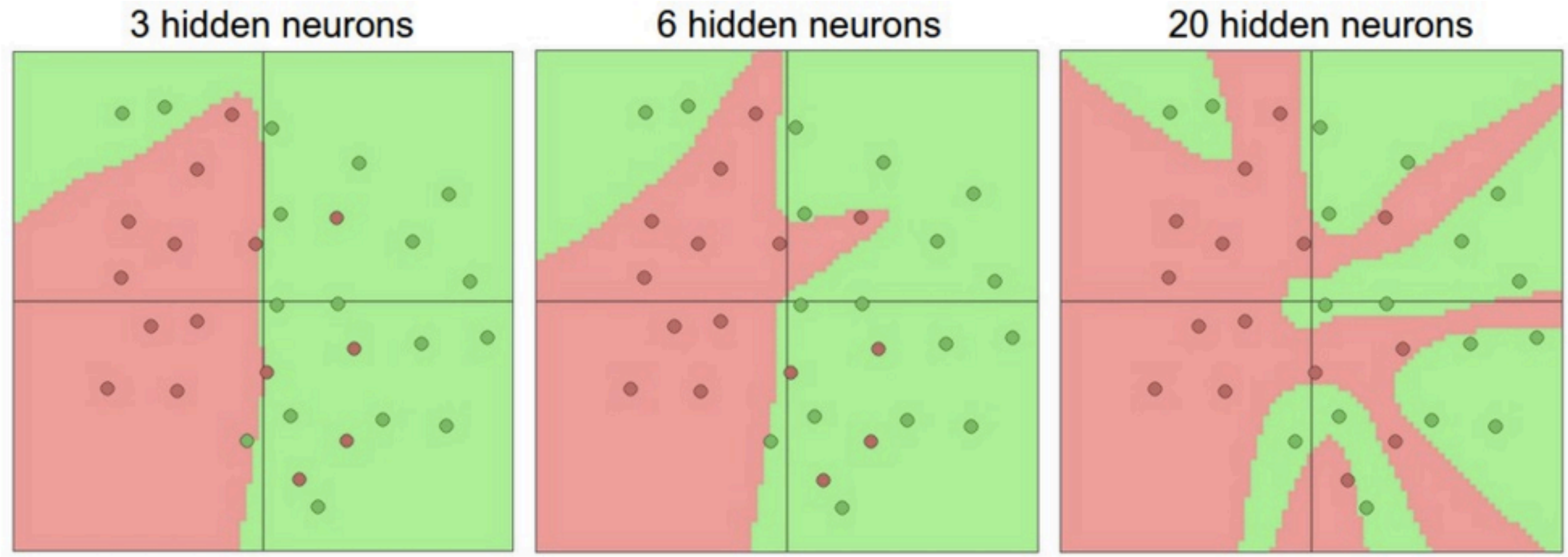


# CAPACITY OF NEURAL NETWORK

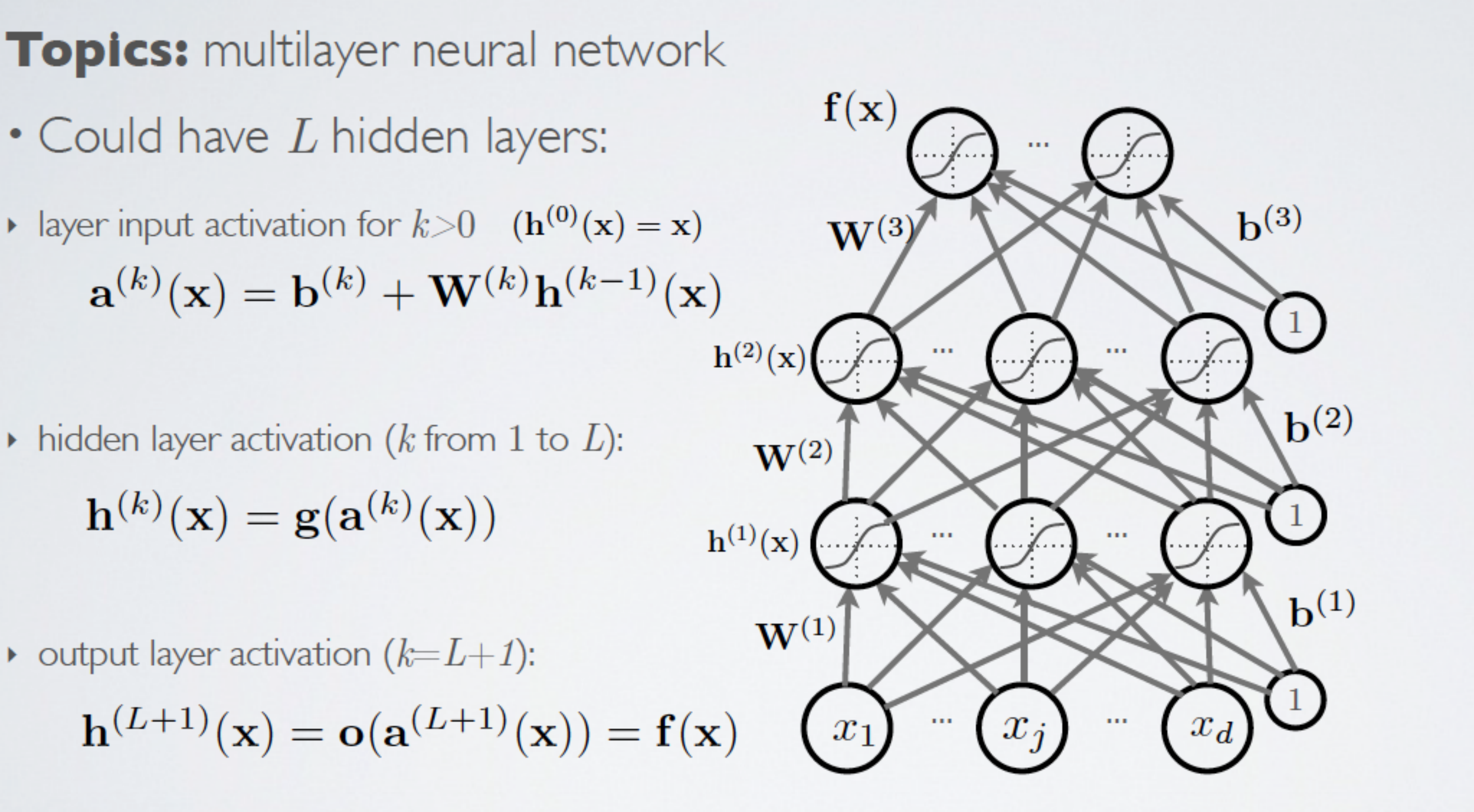
**Topics:** universal approximation

- Universal approximation theorem (Hornik, 1991):
  - "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"
- The result applies for sigmoid, tanh and many other hidden layer activation functions

- This is a good result, but it doesn't mean there is a learning algorithm that can find the necessary parameter values!


3 hidden neurons
6 hidden neurons
20 hidden neurons

# How to train a neural network?



**Topics:** multilayer neural network

- Could have $L$ hidden layers:
  - layer input activation for $k>0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

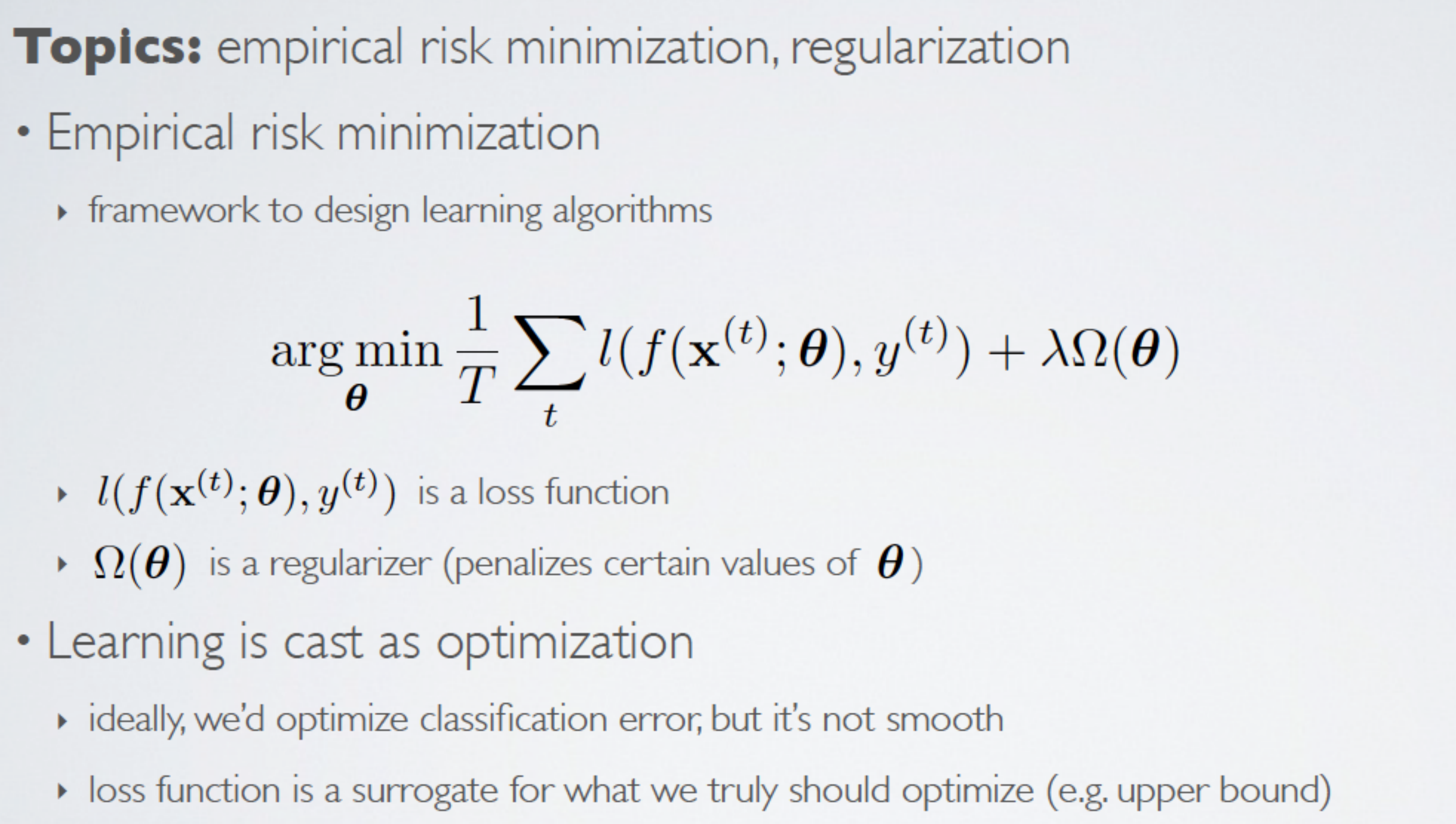
  - hidden layer activation ($k$ from 1 to $L$):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

  - output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

# Empirical Risk Minimization and Regularization



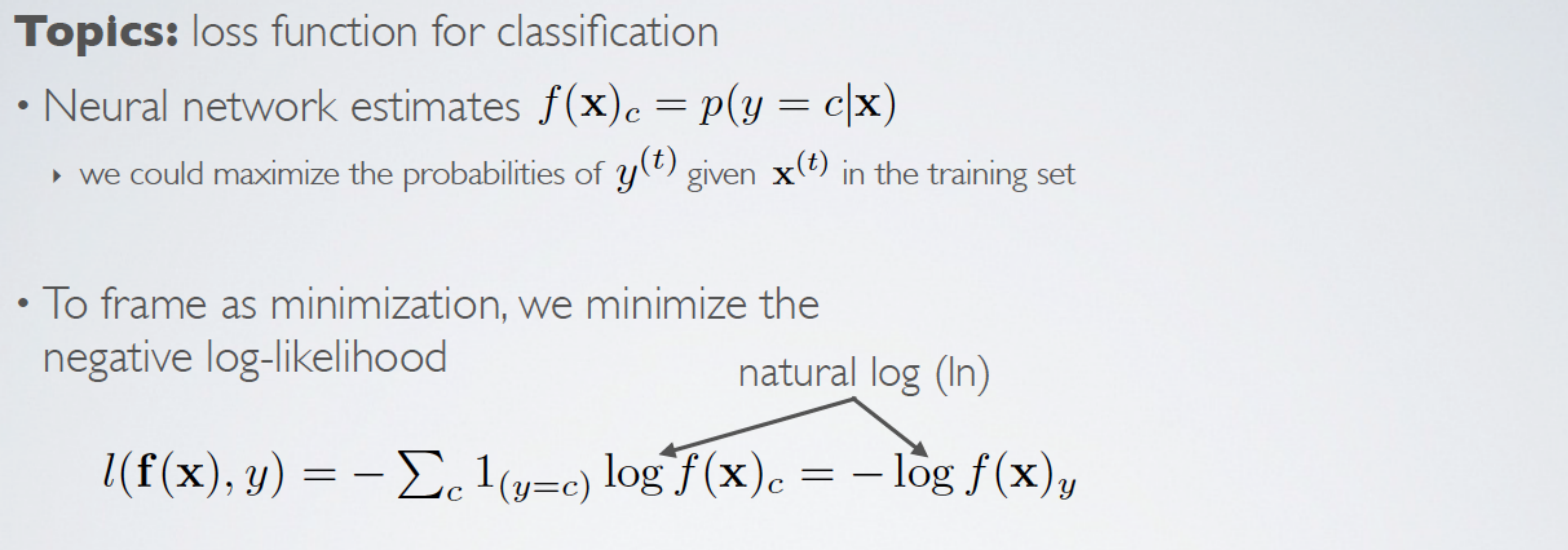
**Topics:** empirical risk minimization, regularization

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) + \lambda \Omega(\boldsymbol{\theta})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function
  - $\Omega(\boldsymbol{\theta})$ is a regularizer (penalizes certain values of $\boldsymbol{\theta}$)
- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

# Loss Function

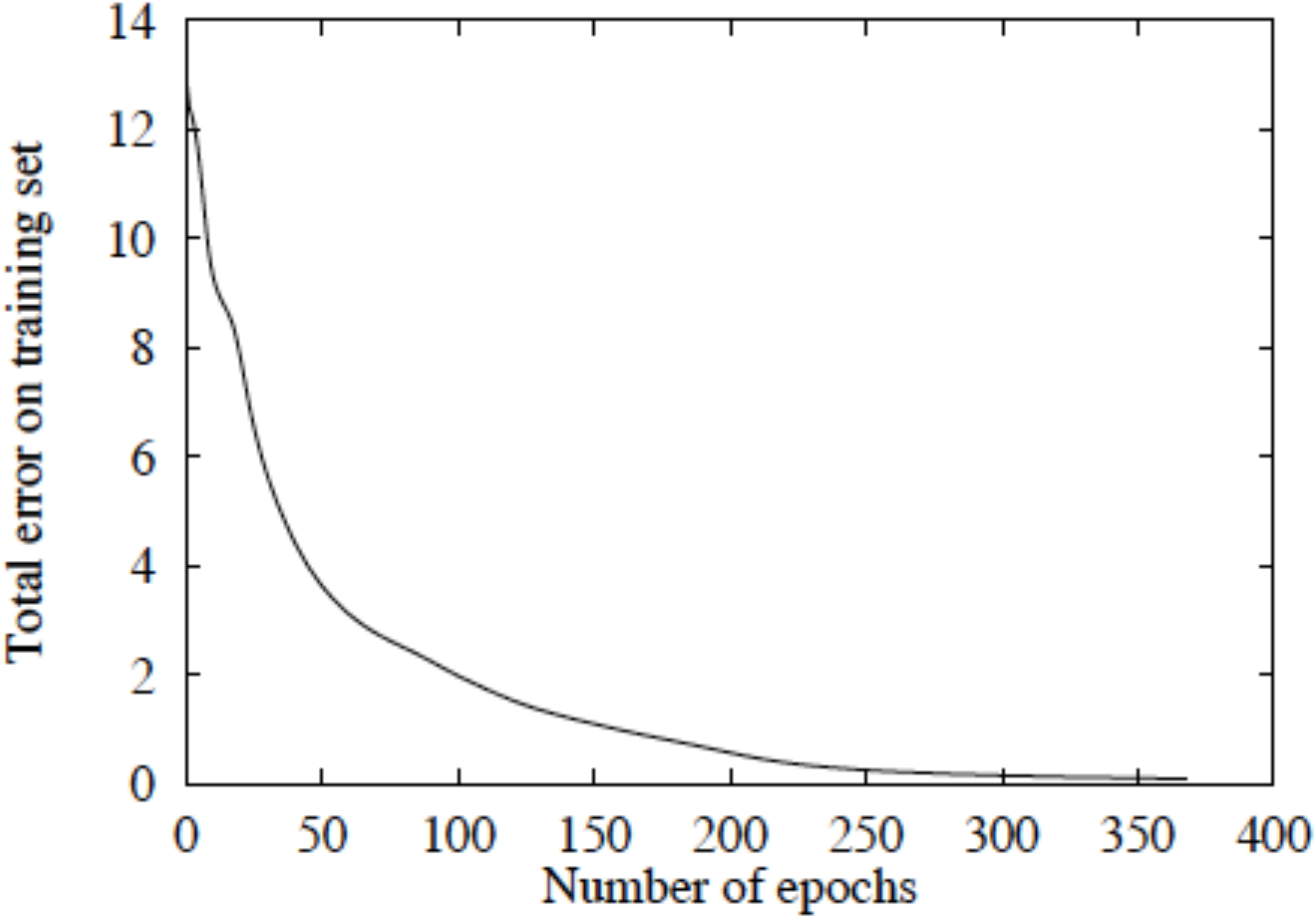


**Topics:** loss function for classification

- Neural network estimates $f(\mathbf{x})_c = p(y = c|\mathbf{x})$
  - we could maximize the probabilities of $y^{(t)}$ given $\mathbf{x}^{(t)}$ in the training set
- To frame as minimization, we minimize the negative log-likelihood

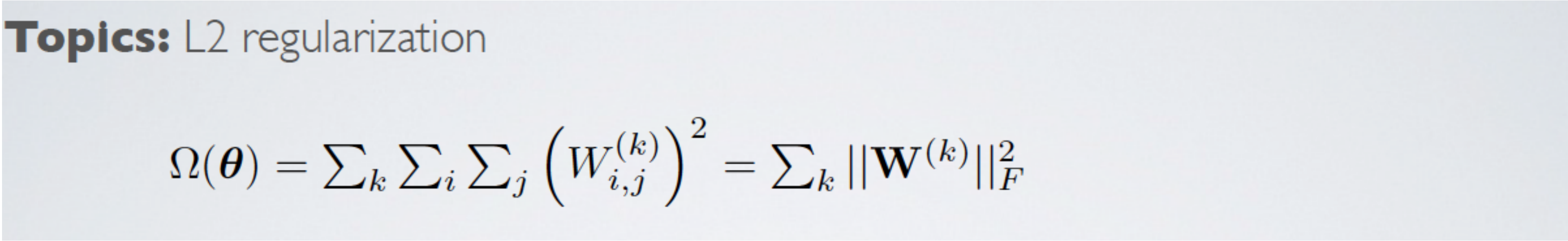
natural log (ln)

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y$$



[figure from Greg Mori’s slides]

# Regularization

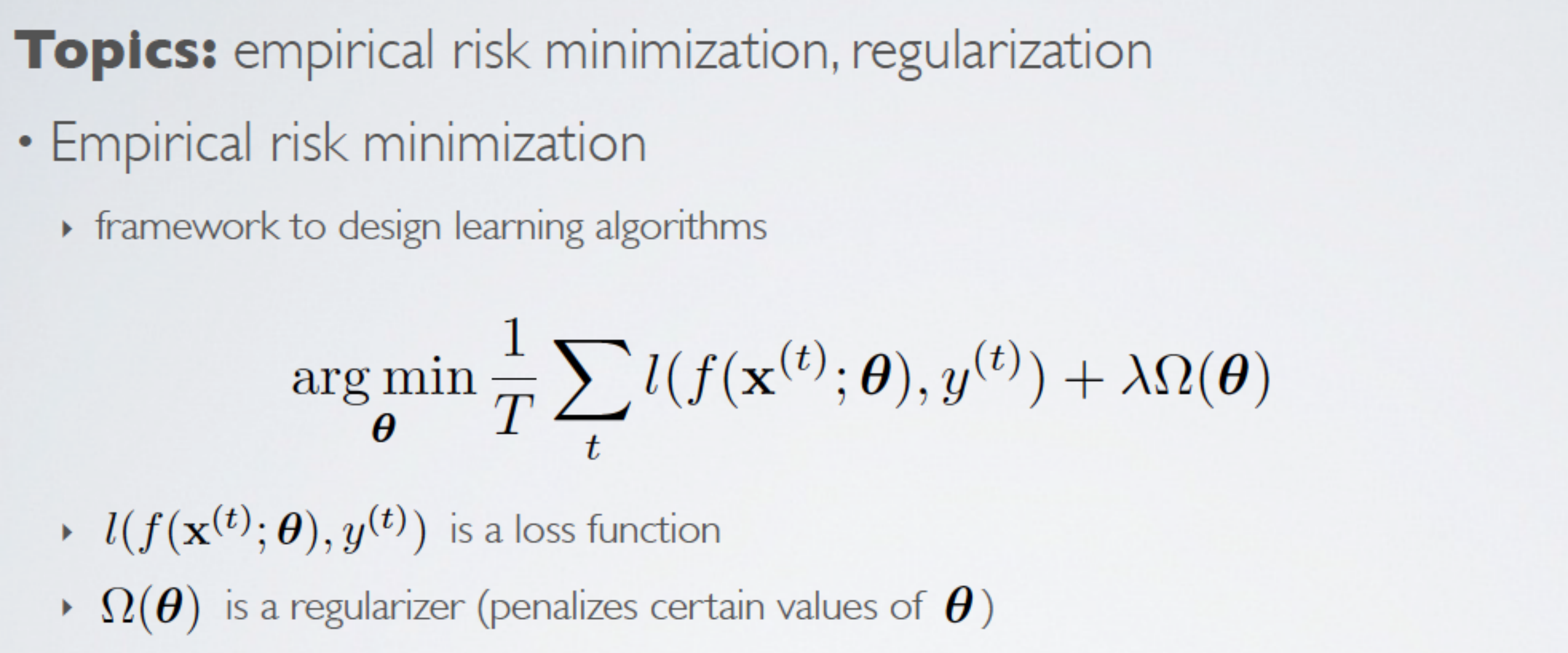


**Topics:** L2 regularization

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)}\right)^2 = \sum_k ||\mathbf{W}^{(k)}||_F^2$$

It’s call squared Frobenius norm when **W** is a matrix

# Empirical Risk Minimization

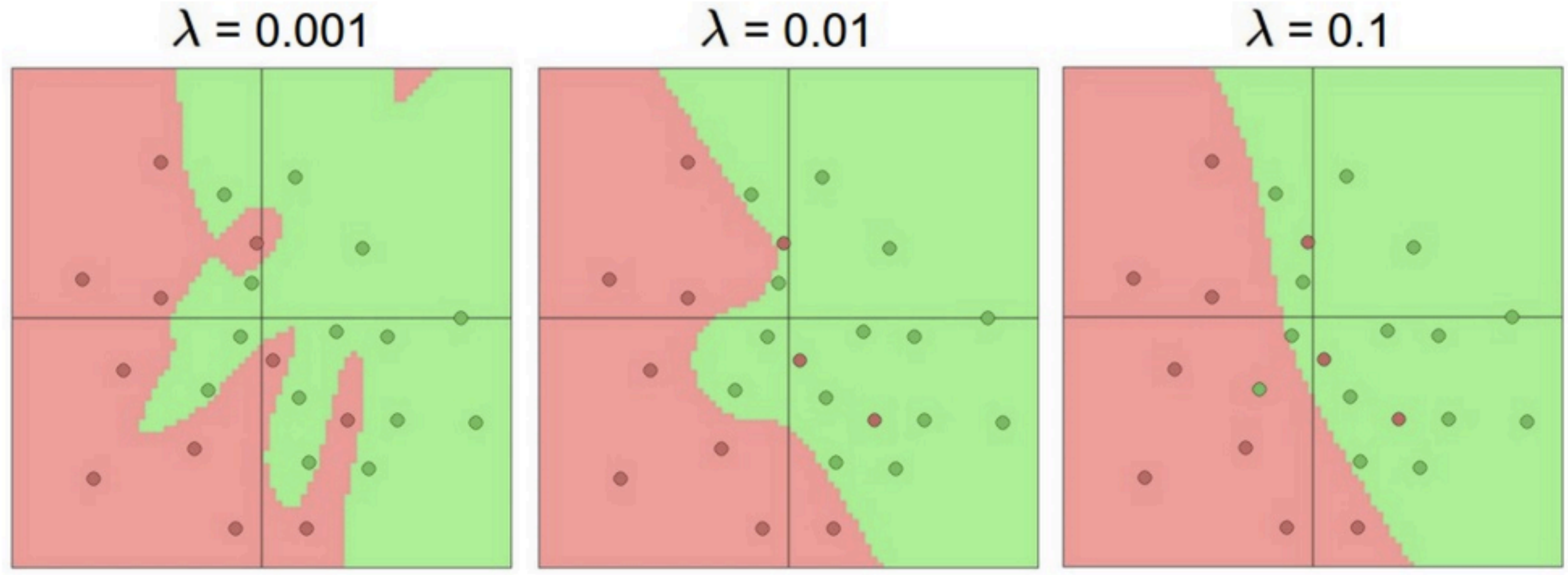


**Topics:** empirical risk minimization, regularization

- Empirical risk minimization
  - framework to design learning algorithms

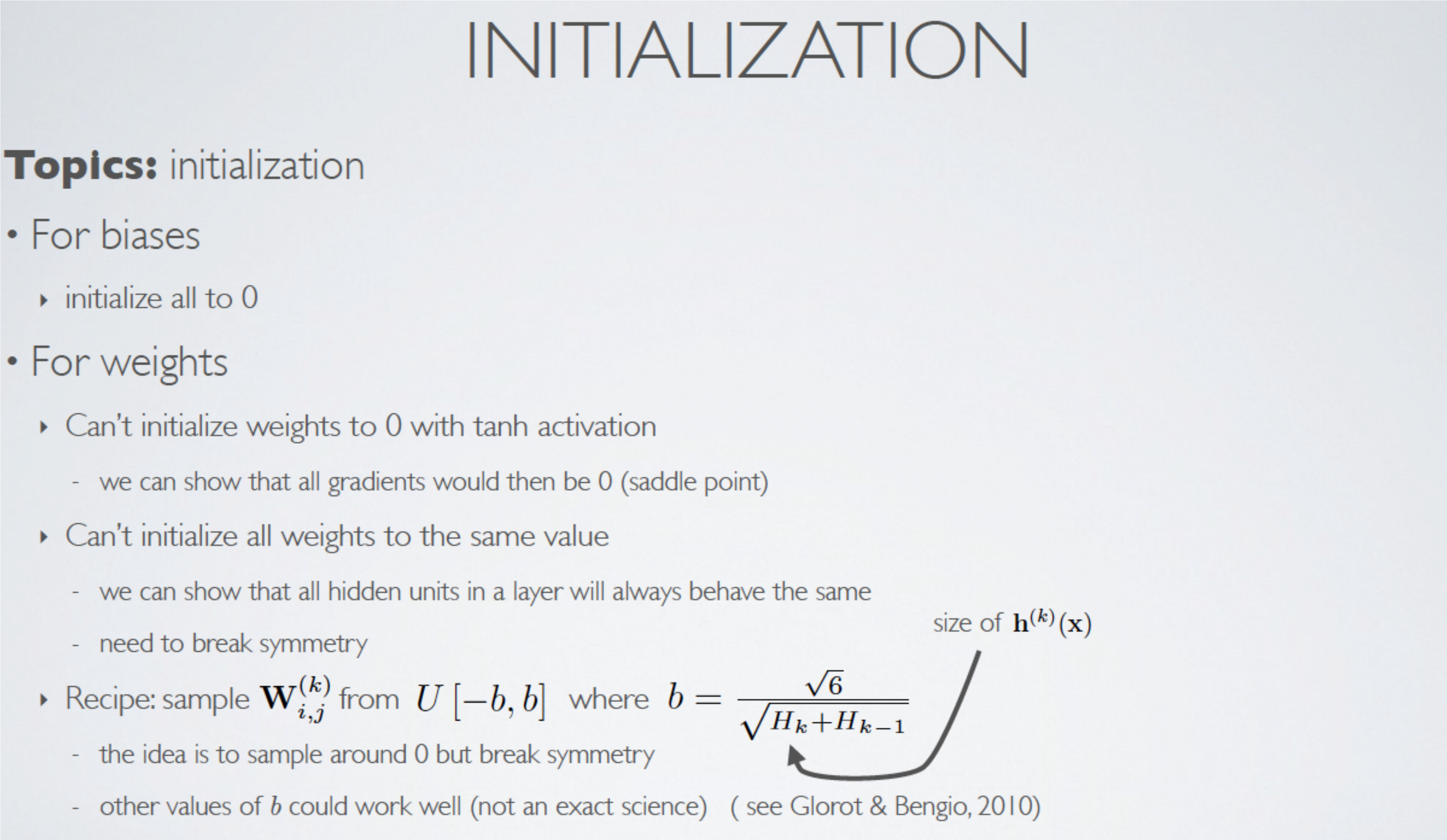
$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) + \lambda \Omega(\boldsymbol{\theta})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function
  - $\Omega(\boldsymbol{\theta})$ is a regularizer (penalizes certain values of $\boldsymbol{\theta}$)



$\lambda = 0.001$ $\lambda = 0.01$ $\lambda = 0.1$

[http://cs231n.github.io/neural-networks-1/]



# INITIALIZATION

**Topics:** initialization

- For biases
  - initialize all to 0
- For weights
  - Can't initialize weights to 0 with tanh activation
    - we can show that all gradients would then be 0 (saddle point)
  - Can't initialize all weights to the same value
    - we can show that all hidden units in a layer will always behave the same
    - need to break symmetry
  - Recipe: sample $\mathbf{W}_{i,j}^{(k)}$ from $U\left[-b, b\right]$ where $b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$
    - the idea is to sample around 0 but break symmetry
    - other values of $b$ could work well (not an exact science) ( see Glorot & Bengio, 2010)

size of $\mathbf{h}^{(k)}(\mathbf{x})$

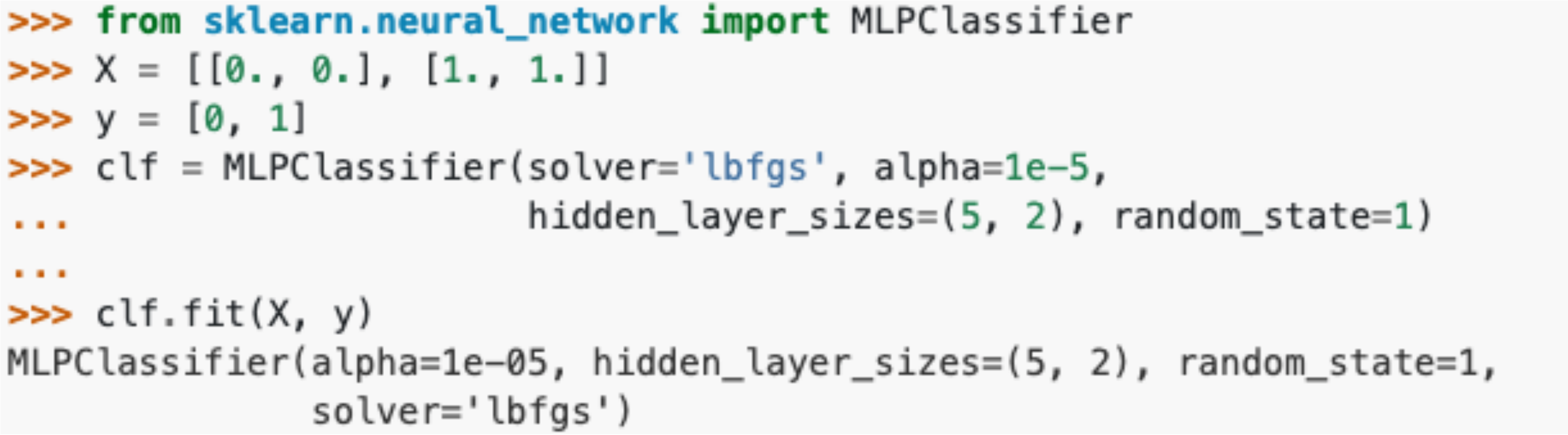
# Learning the Parameters (weights and bias)

- Backpropagation (BP) algorithm (not required for this course)
- Further reading on BP:
  - https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd
  - https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

# Popular Tools

- scikit-learn: https://scikit-learn.org/
- PyTorch: https://pytorch.org/
- Tensorflow: https://www.tensorflow.org/

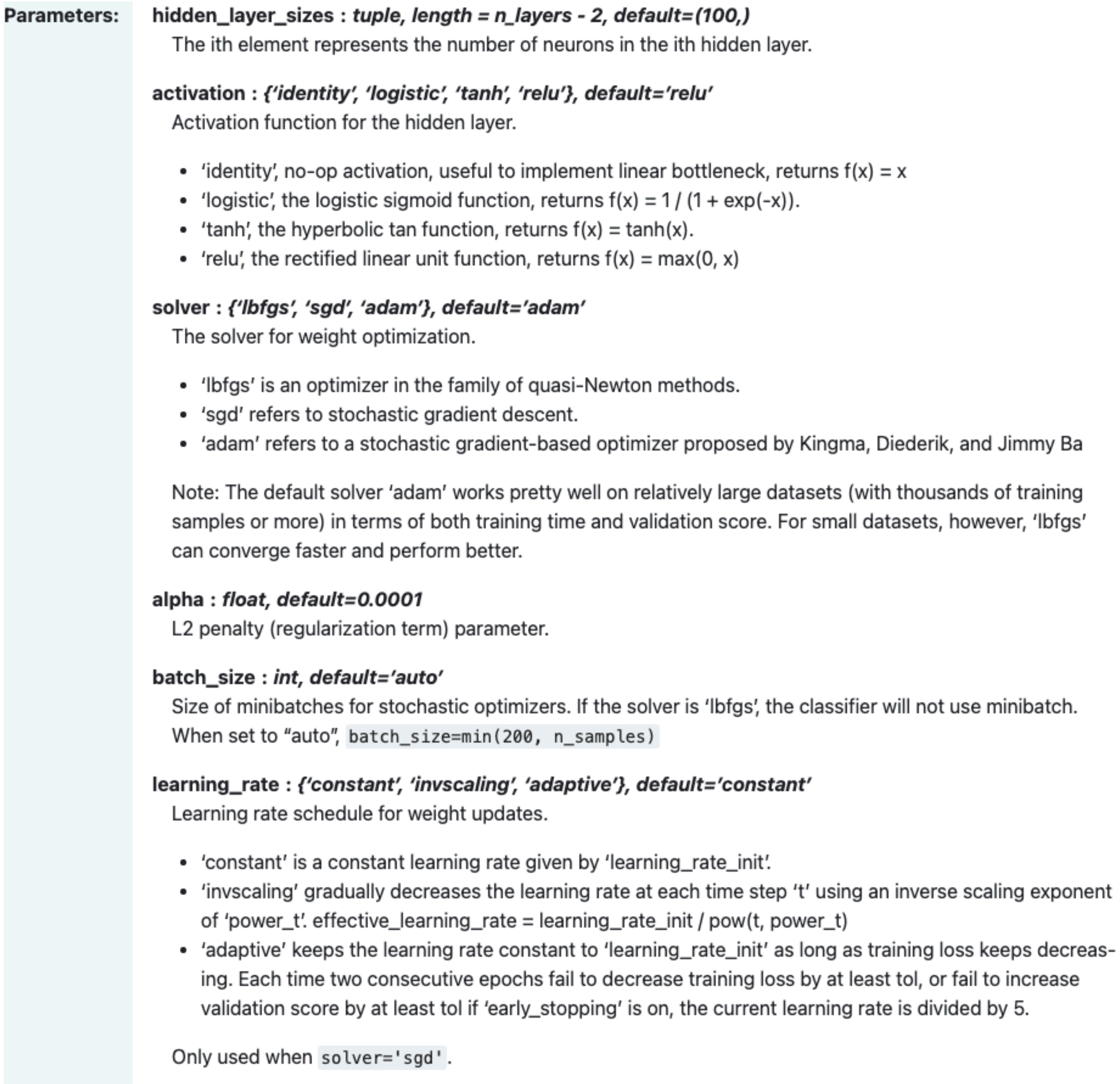
# scikit-learn MLPClassifier



```
>>> from sklearn.neural_network import MLPClassifier
>>> X = [[0., 0.], [1., 1.]]
>>> y = [0, 1]
>>> clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
...                     hidden_layer_sizes=(5, 2), random_state=1)
...
>>> clf.fit(X, y)
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_state=1,
              solver='lbfgs')
```

# scikit-learn MLPClassifier

More parameters that can be indicated/tuned, details at: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html



**Parameters:**

**hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)***
The ith element represents the number of neurons in the ith hidden layer.

**activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'***
Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

**solver : *{'lbfgs', 'sgd', 'adam'}, default='adam'***
The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

**alpha : *float, default=0.0001***
L2 penalty (regularization term) parameter.

**batch_size : *int, default='auto'***
Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`

**learning_rate : *{'constant', 'invscaling', 'adaptive'}, default='constant'***
Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

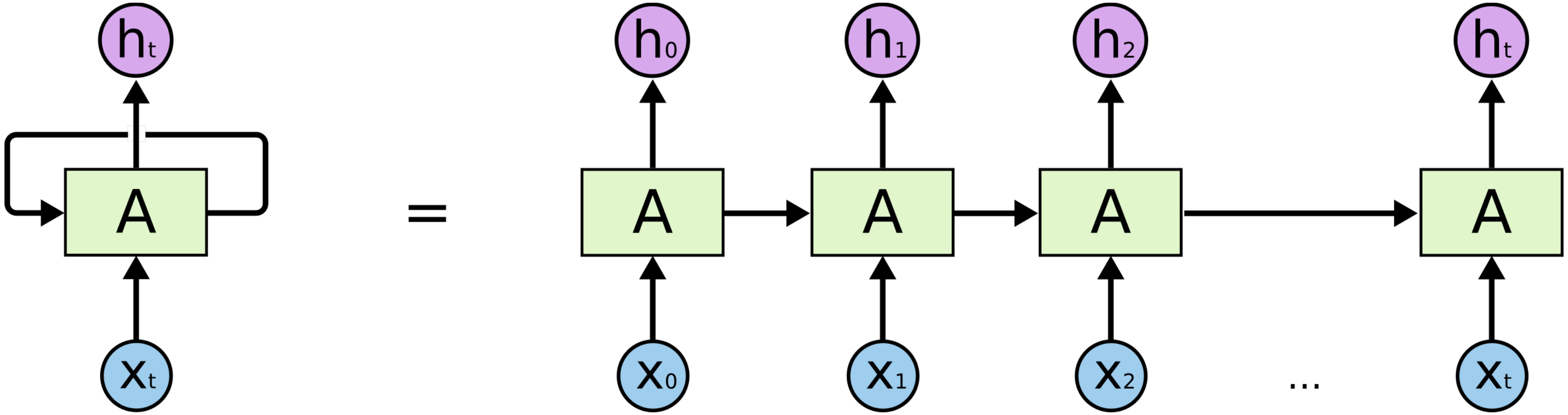
Only used when `solver='sgd'`.

# Outline

- Logistic Regression
- Feedforward Neural Networks
- Recurrent Neural Networks

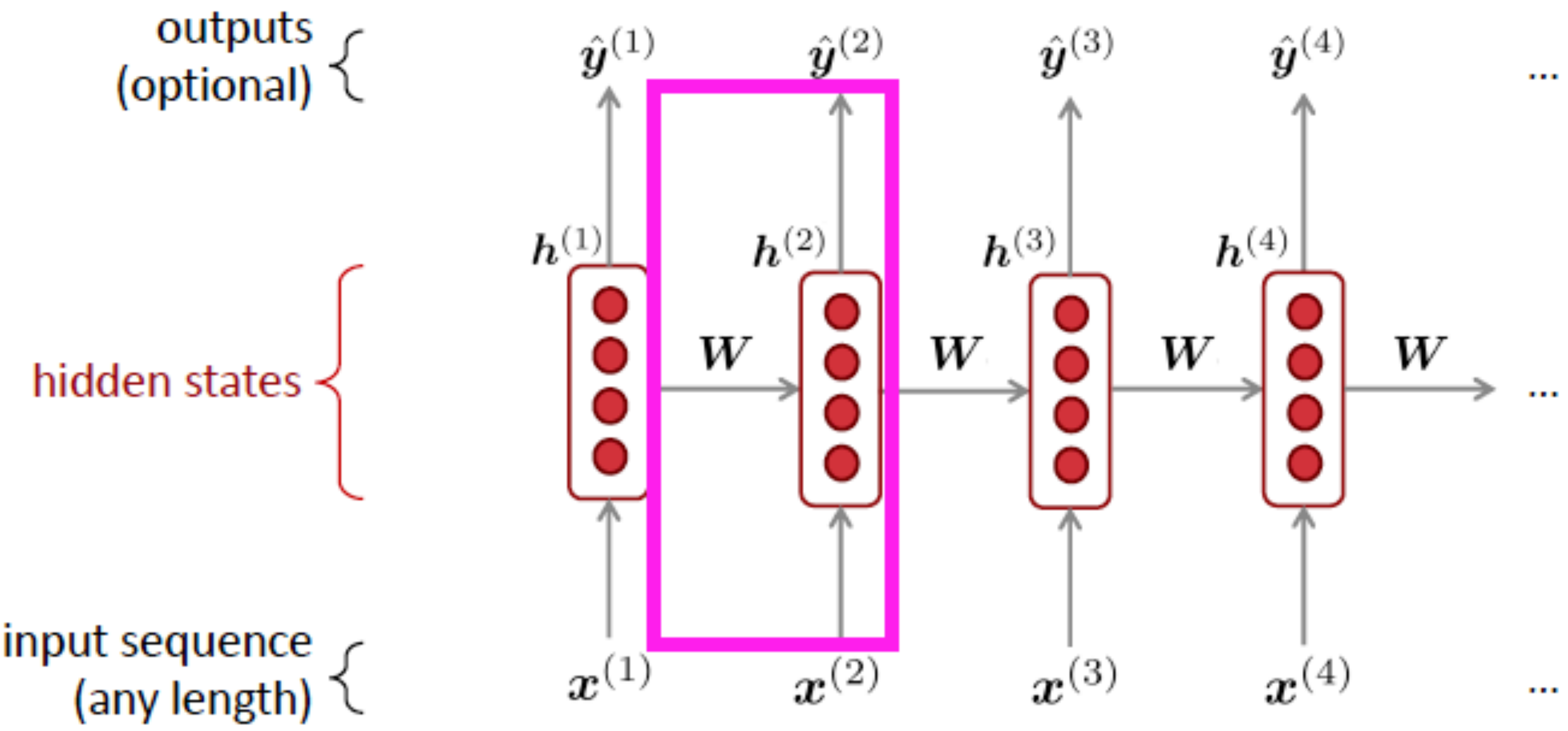
# Long Distance Dependencies

- It is very difficult to train NNs to retain information over many time steps
- This makes it very difficult to handle long-distance dependencies, such as subject-verb agreement.
- E.g. Jane walked into the room. John walked in too. It was late in the day. Jane said hi to _?_



# Recurrent Neural Networks (RNN)

- Core idea: Apply the same weights $W$ repeatedly



# A Simple RNN Language Model

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

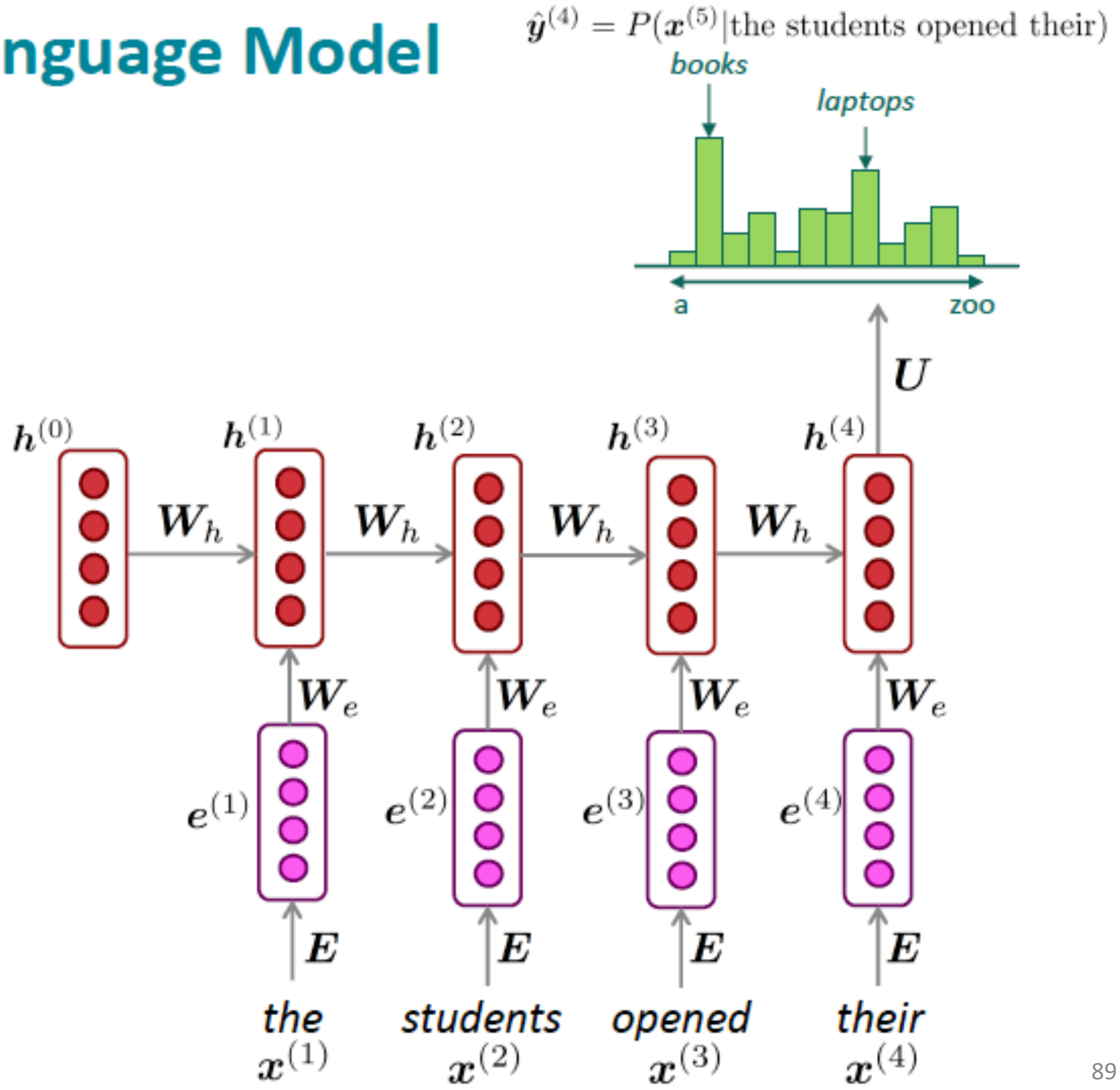
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



$$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$$

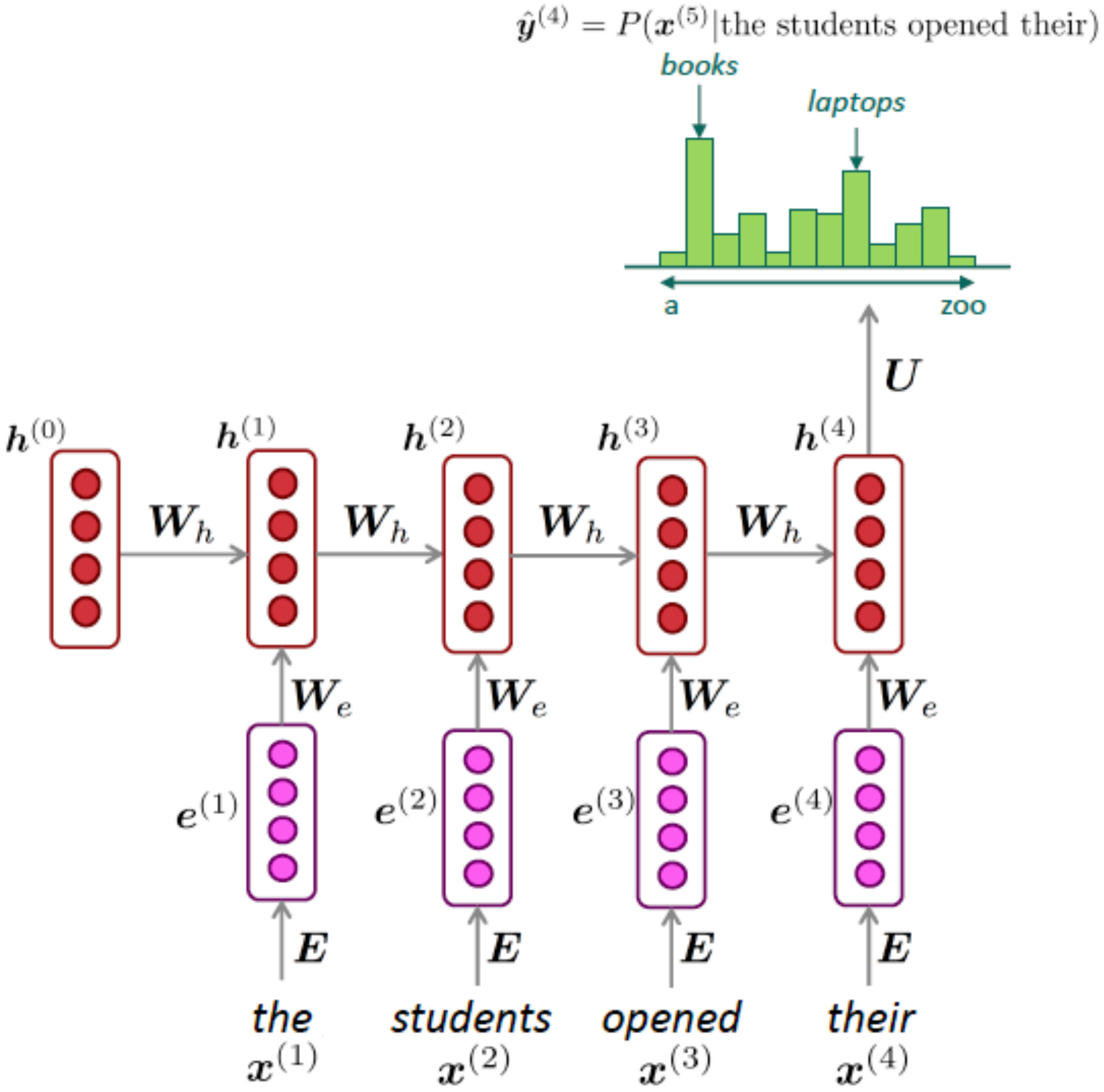
# Pros and Cons

RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

RNN **Disadvantages**:

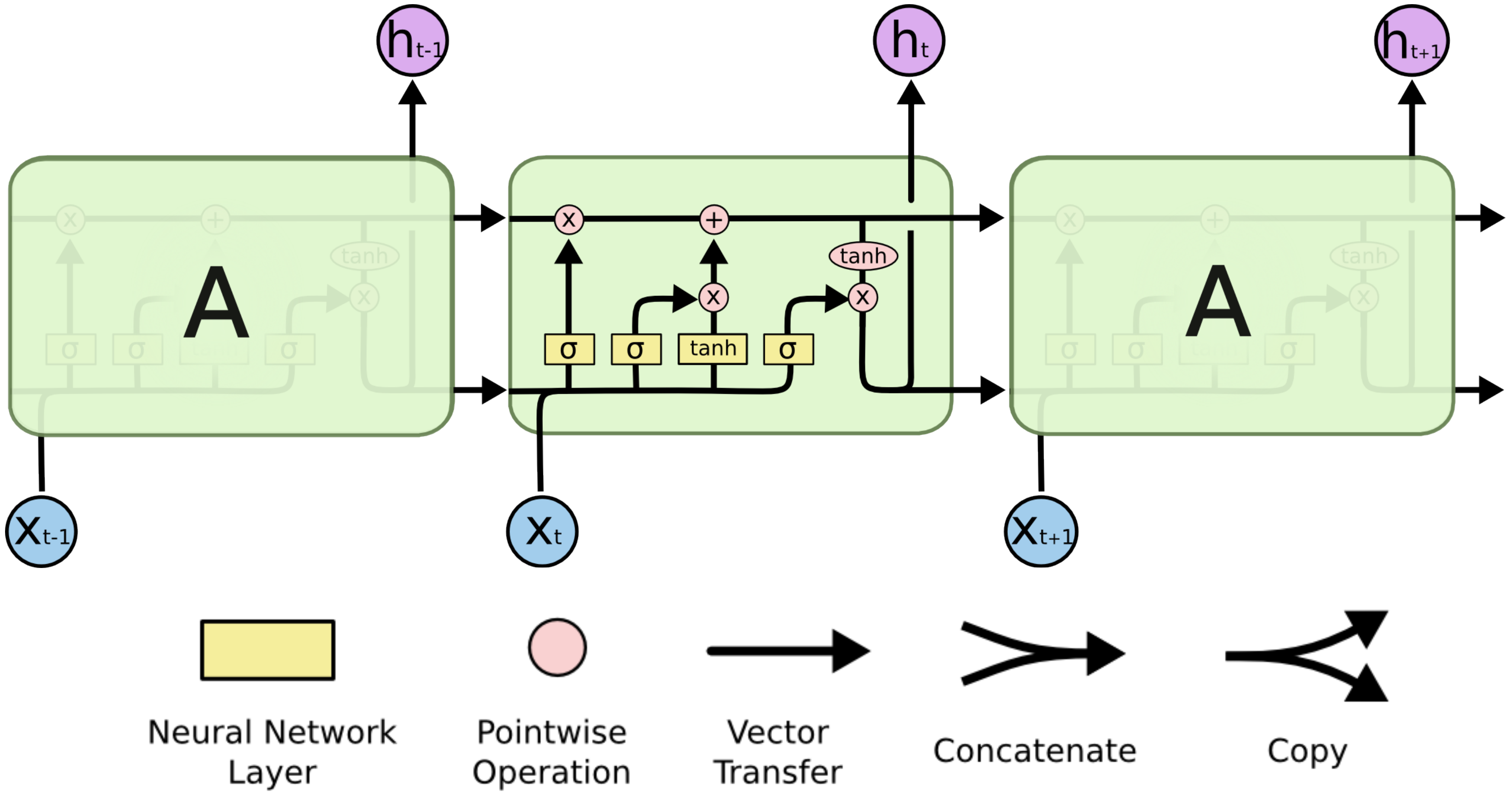
- Recurrent computation is slow
- In practice, difficult to access information from many steps back



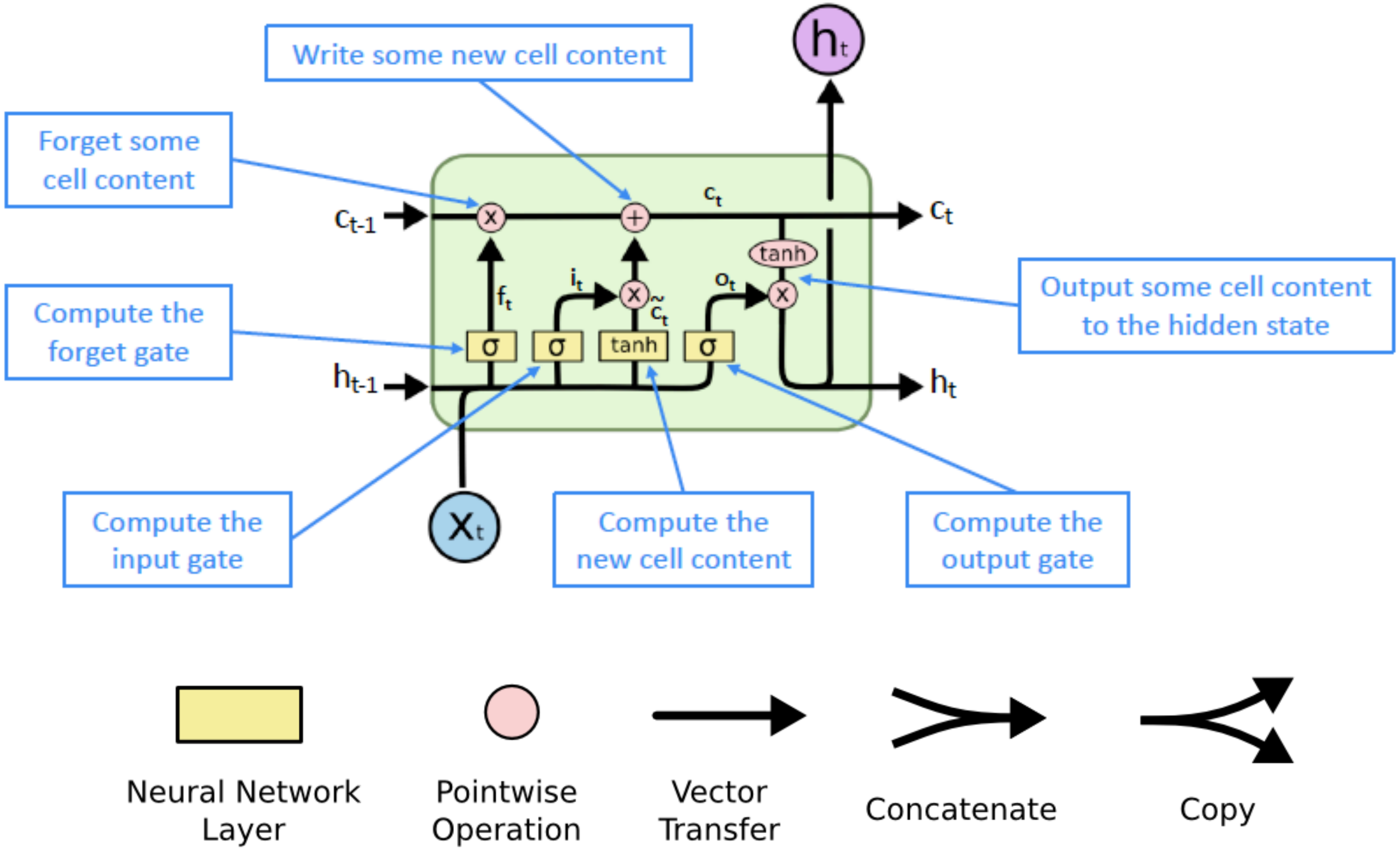
# Long-Short Term Memory Networks (LSTMs)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997

# Long-Short Term Memory Networks (LSTMs)



# Long-Short Term Memory Networks (LSTMs)



Write some new cell content

Forget some cell content

Compute the forget gate

Output some cell content to the hidden state

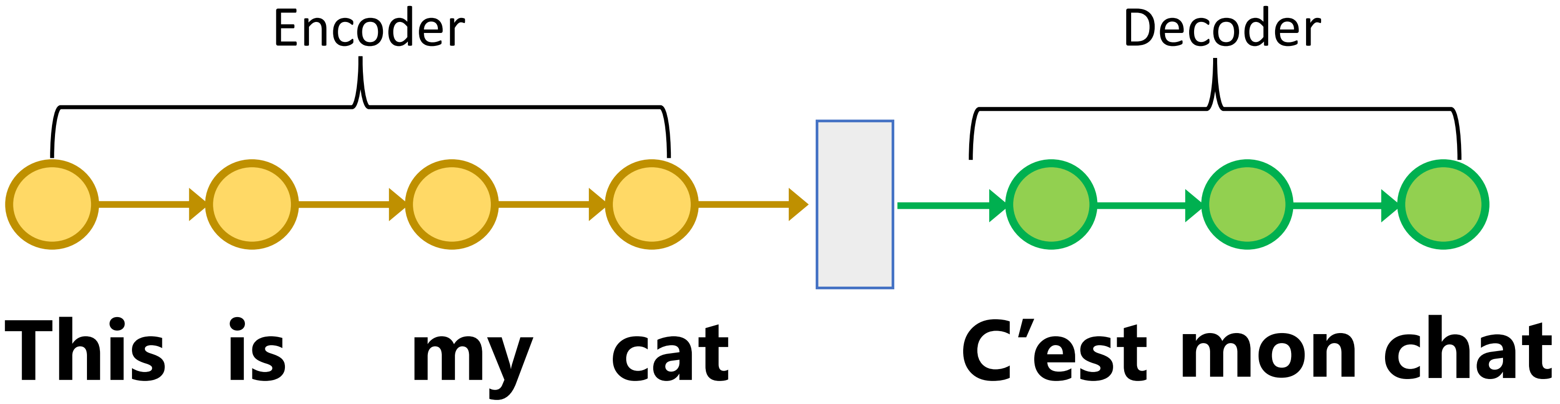
Compute the input gate

Compute the new cell content

Compute the output gate

# Sequence to Sequence

- Encoder/Decoder framework maps one sequence to a "deep vector" then another LSTM maps this vector to an output sequence.



# Successful Applications of LSTMs

- Speech recognition: Language and acoustic modeling
- Sequence labeling
  - POS Tagging
  - NER
  - Phrase Chunking
- Neural syntactic and semantic parsing
- Image captioning
- Sequence to Sequence
  - Machine Translation (Sustkever, Vinyals, & Le, 2014)
  - Summarization
  - Video Captioning (input sequence of CNN frame outputs)