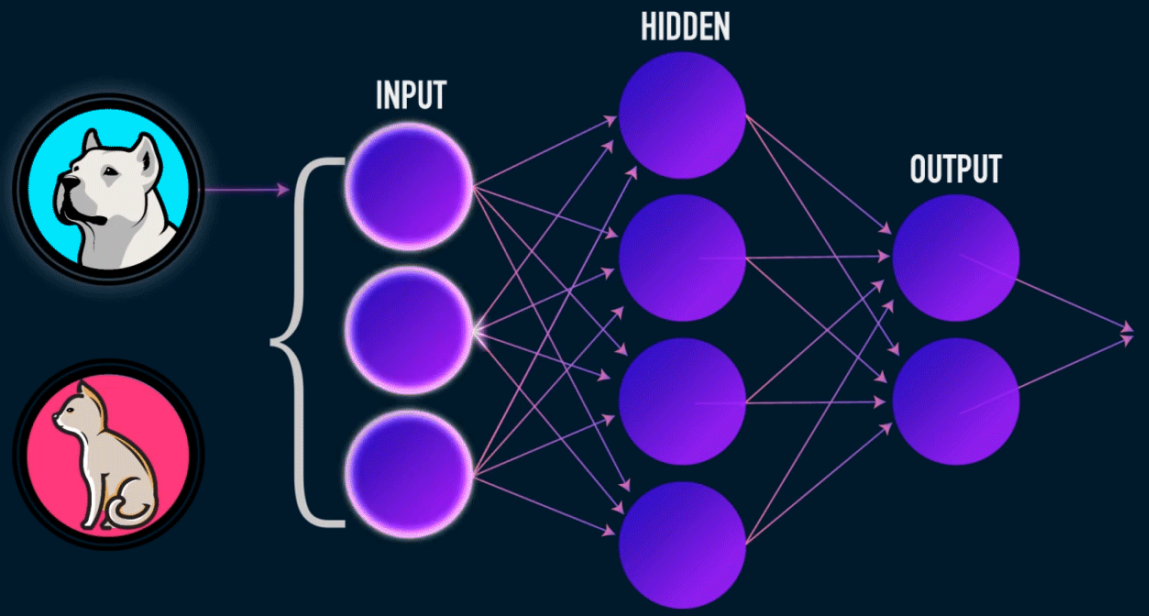
Lecture 20:
Neural Networks for NLP

Zubin Pahuja

zpahuja2@illinois.edu



Today's Lecture

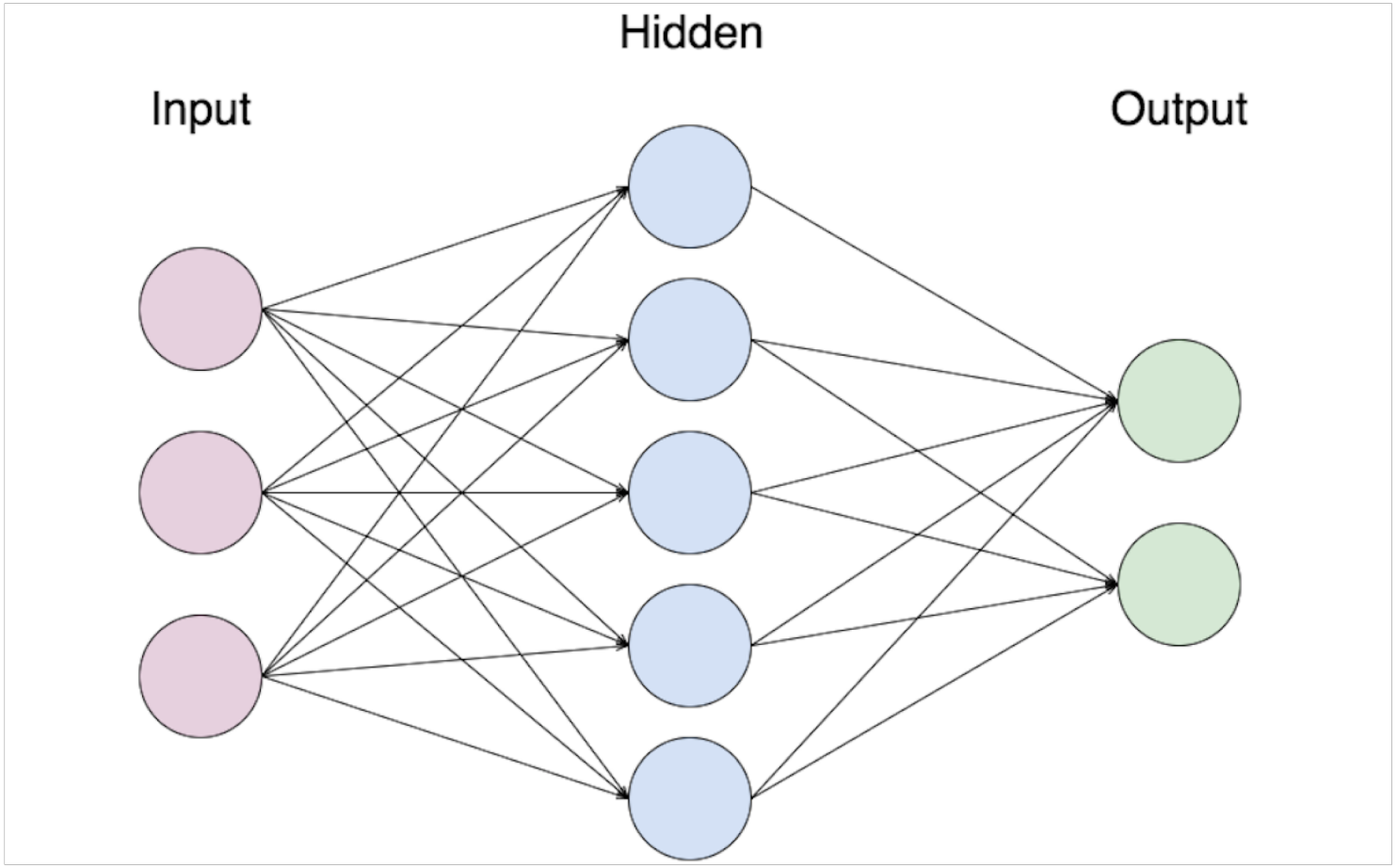
- **Feed-forward neural networks** as **classifiers**
  - simple architecture in which computation proceeds from one layer to the next
- Application to **language modeling**
  - assigning probabilities to word sequences and predicting upcoming words

Supervised Learning

Two kinds of prediction problems:

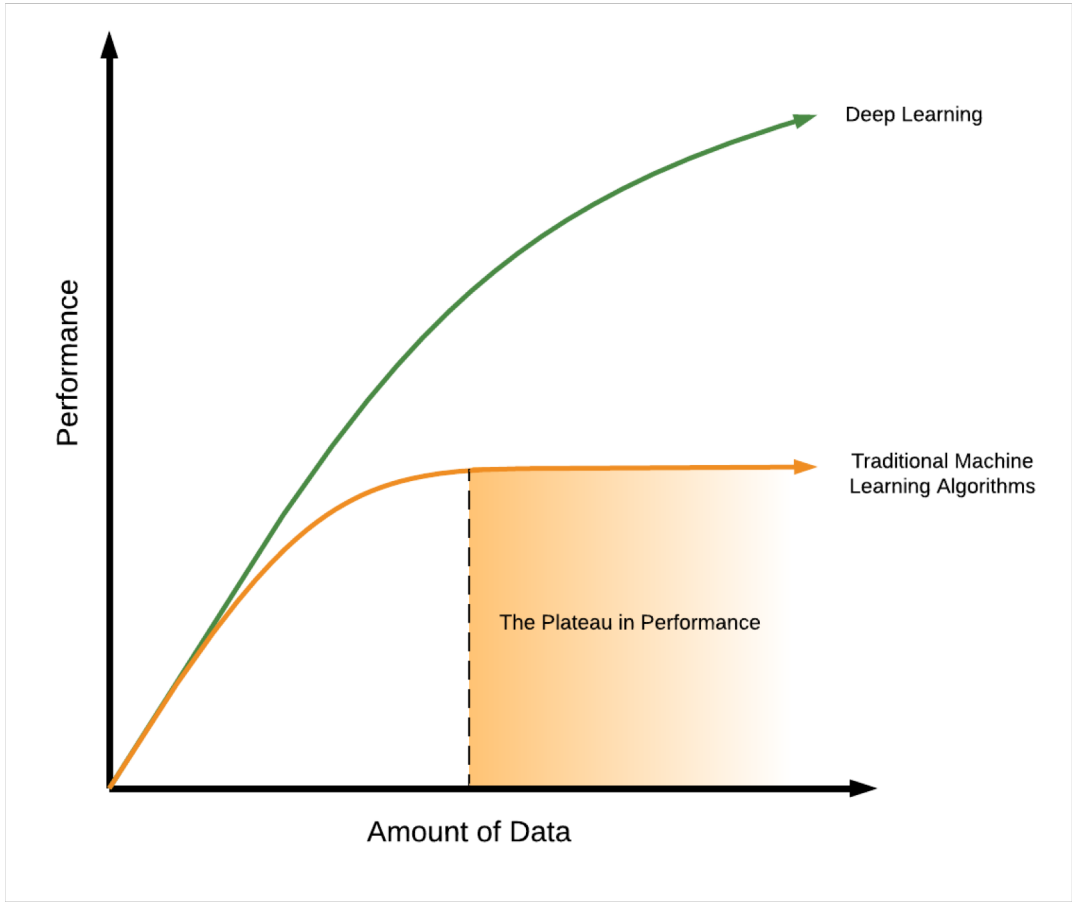
- **Regression**
  - predict results with *continuous* output
  - e.g. price of a house from its size, number of bedrooms, zip code, etc.
- **Classification**
  - predict results in a *discrete* output
  - e.g. whether user will click on an ad

What's a Neural Network?



Why is deep learning taking off?

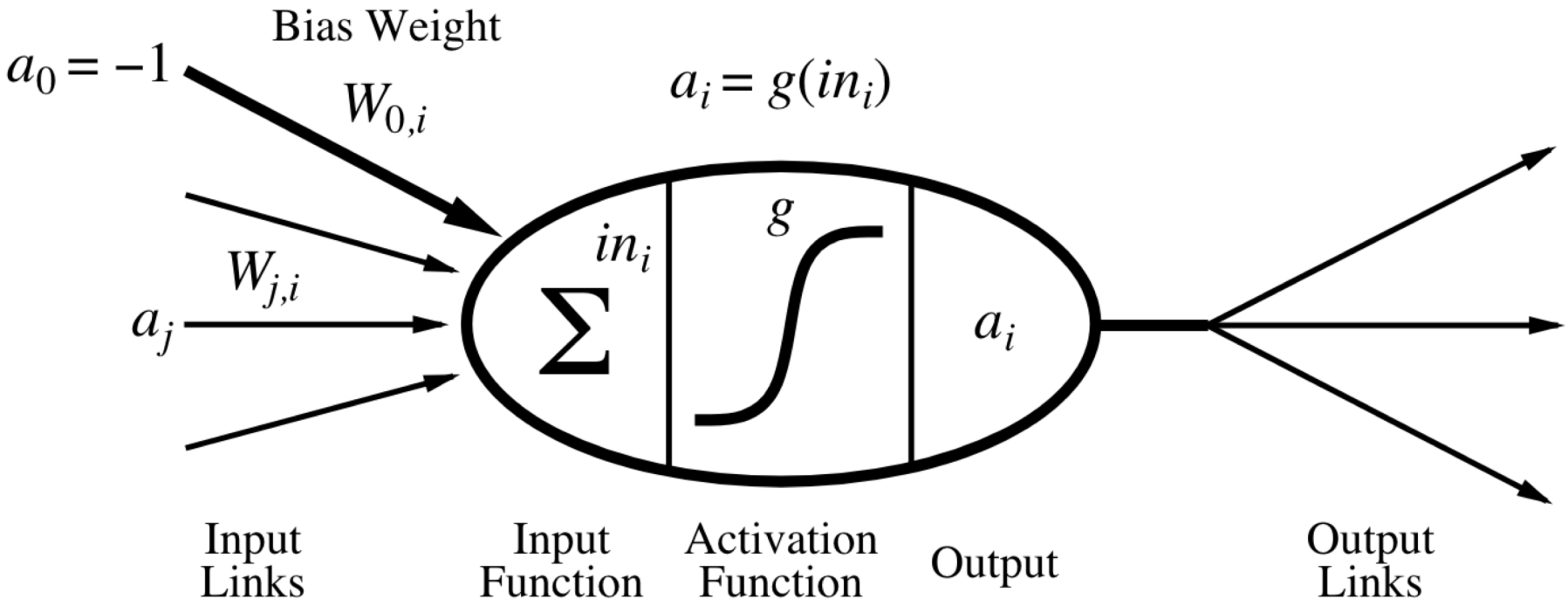
- Unprecedented amount of data
  - performance of traditional learning algorithms such as SVM, logistic regression plateaus
- Faster computation
  - GPU acceleration
  - algorithms that train faster and deeper
    - using ReLU over sigmoid activation
    - gradient descent optimizers, like Adam
- End-to-end learning
  - model directly converts input data into output prediction bypassing intermediate steps in a traditional pipeline



They are called *neural* because their origins lie in

McCulloch-Pitts Neuron

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$



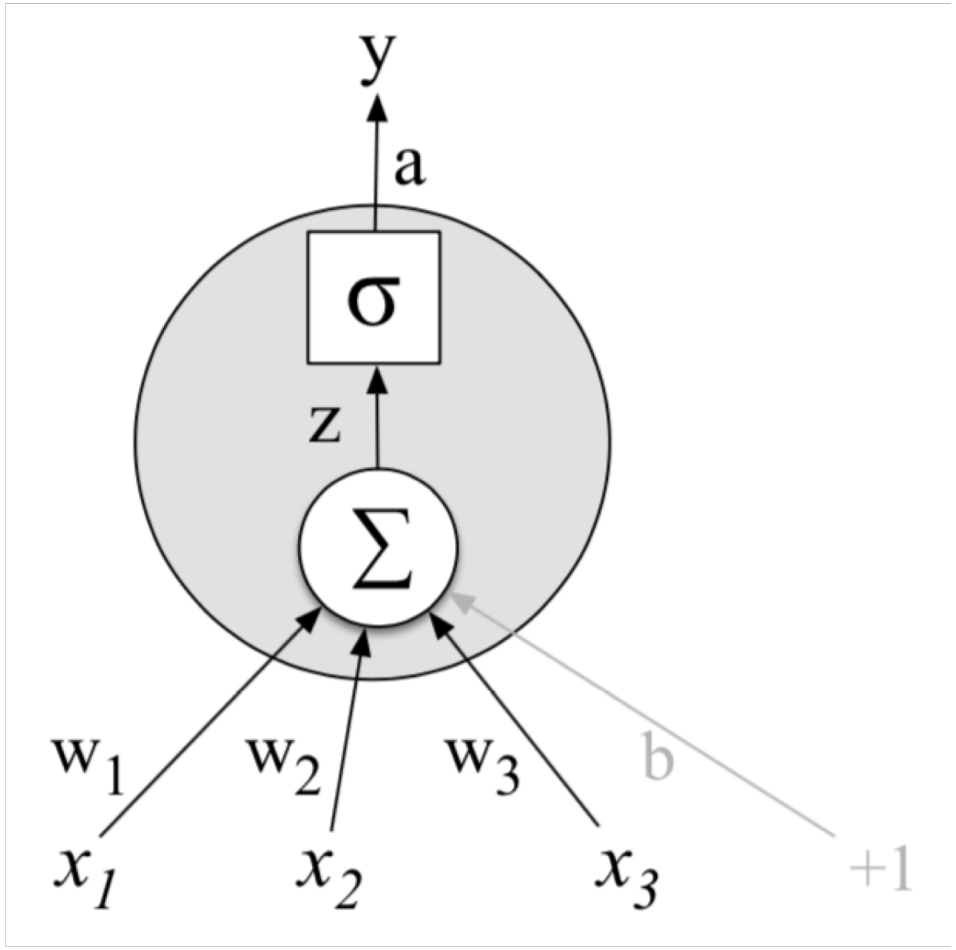
But the modern use in language processing no longer draws on these early biological inspirations

Neural Units

- Building blocks of a neural network
- Given a set of inputs $x_1 ... x_n$, a unit has a set of corresponding **weights** $w_1 ... w_n$ and a **bias** b, so the weighted sum z can be represented as:

$$z = b + \sum_i w_i x_i$$

or, $z = w \cdot x + b$ using **dot-product**

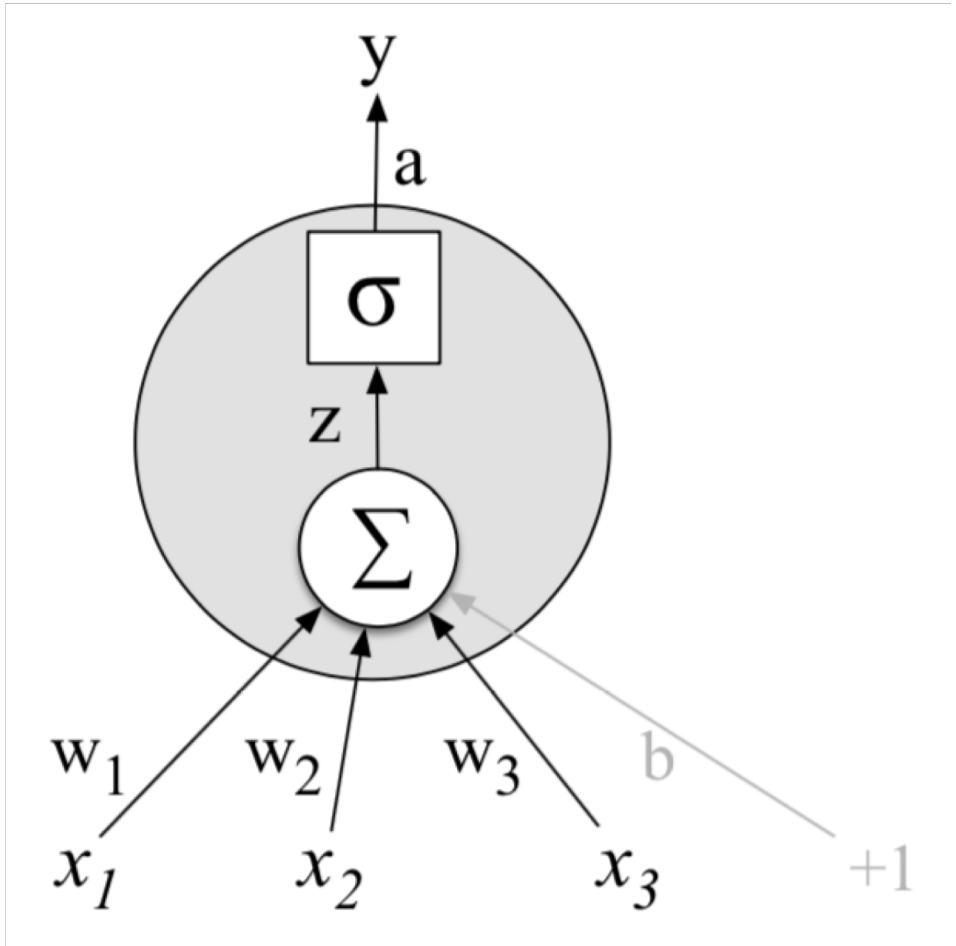


Neural Units

- Apply non-linear function *f* (or *g*) to *z* to compute **activation** *a*:

$$y = a = f(z)$$

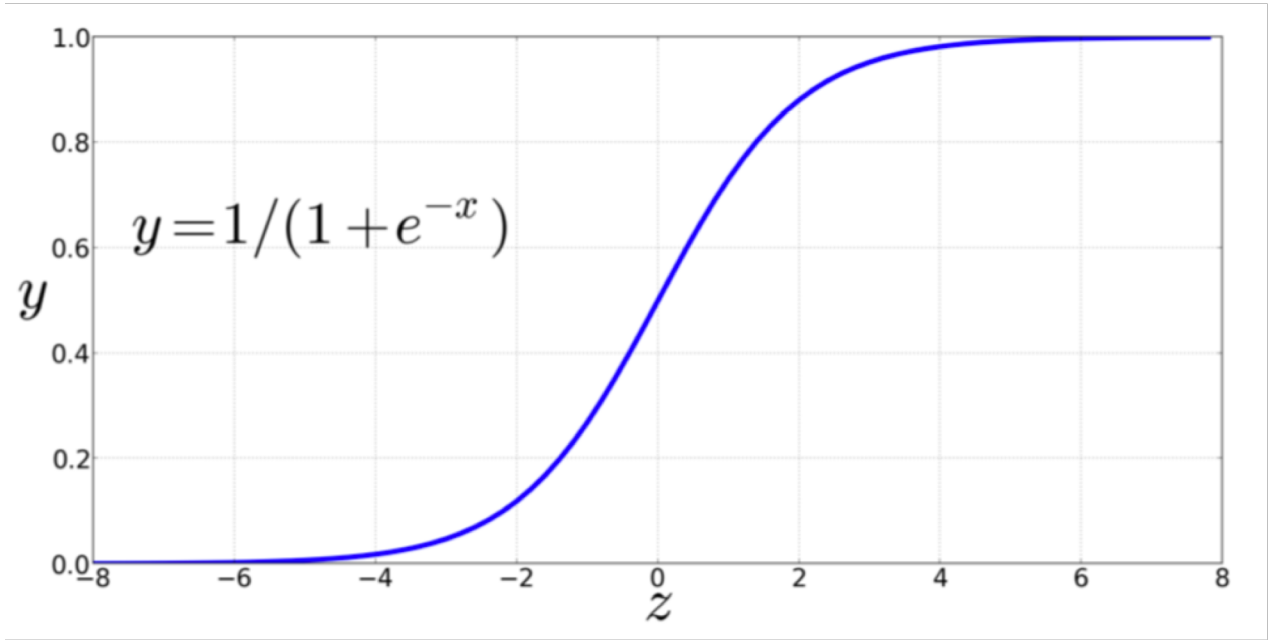
  - since we are modeling a single unit, the activation is also the final output *y*



Activation Functions: Sigmoid

- **Sigmoid (σ)**
  - maps output into the range [0,1]
  - differentiable

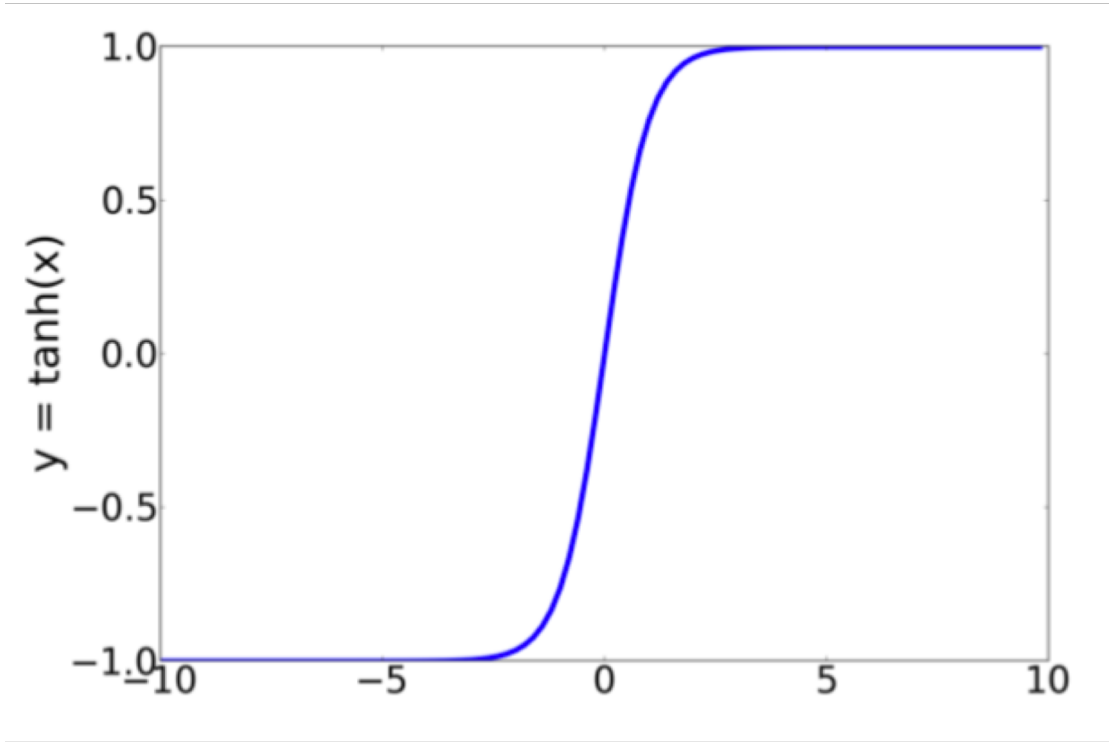
$$y = \sigma(z) = \frac{1}{1+e^{-z}}$$



Activation Functions: Tanh

- **Tanh**
  - maps output into the range [-1, 1]
  - better than sigmoid
  - smoothly differentiable and maps outlier values towards the mean

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

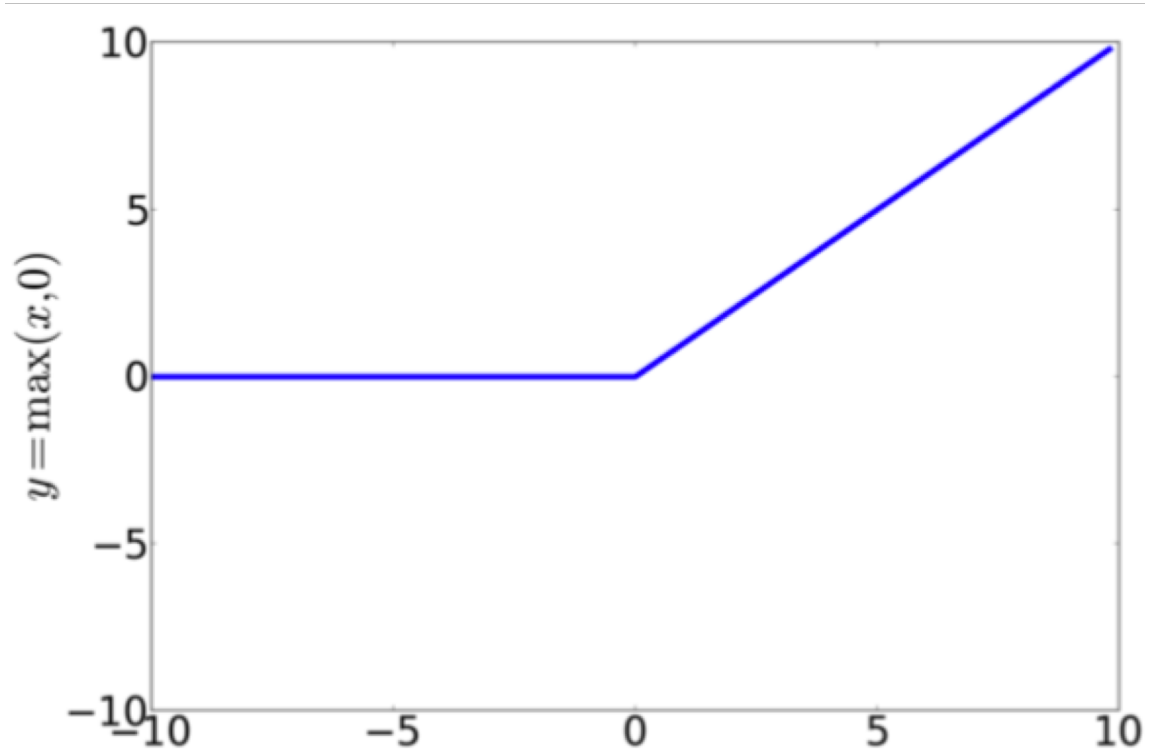


Activation Functions: ReLU

- **Rectified Linear Unit (ReLU)**

$$y = max(x, 0)$$

  - High values of *z* in sigmoid/ tanh result in values of *y* that are close to 1 which causes problems for learning



XOR Problem

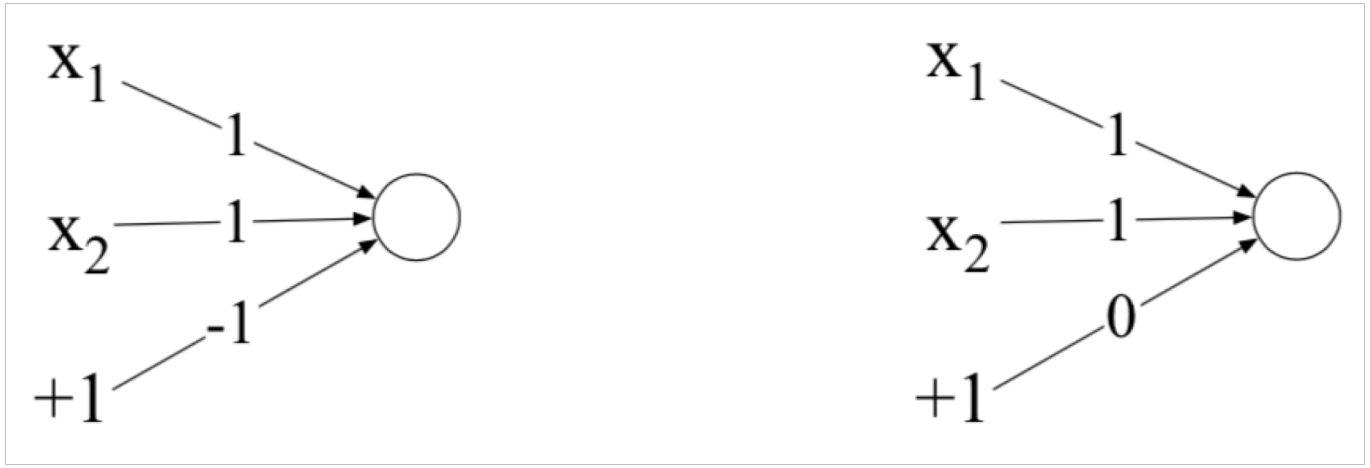
- Minsky-Papert proved **perceptron** can't compute **XOR** logical operation

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

XOR Problem

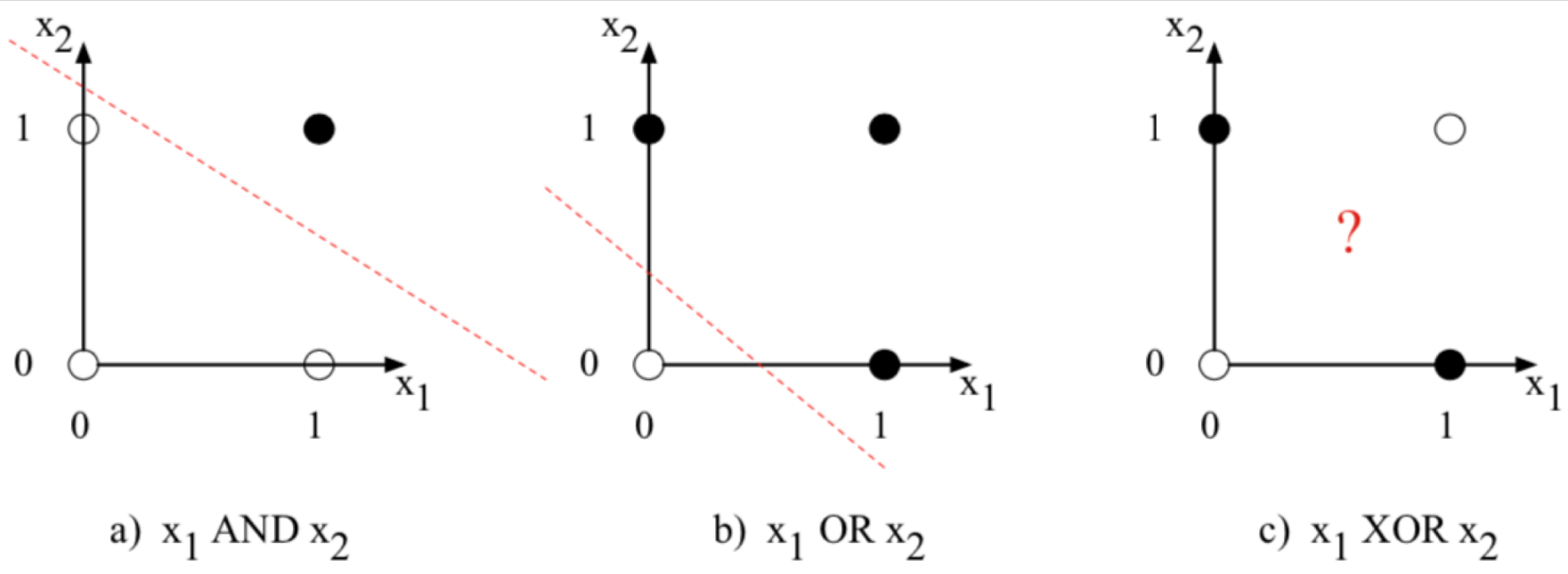
- Perceptron can compute the logical **AND** and **OR** functions easily



- But it's not possible to build a perceptron to compute logical **XOR**!

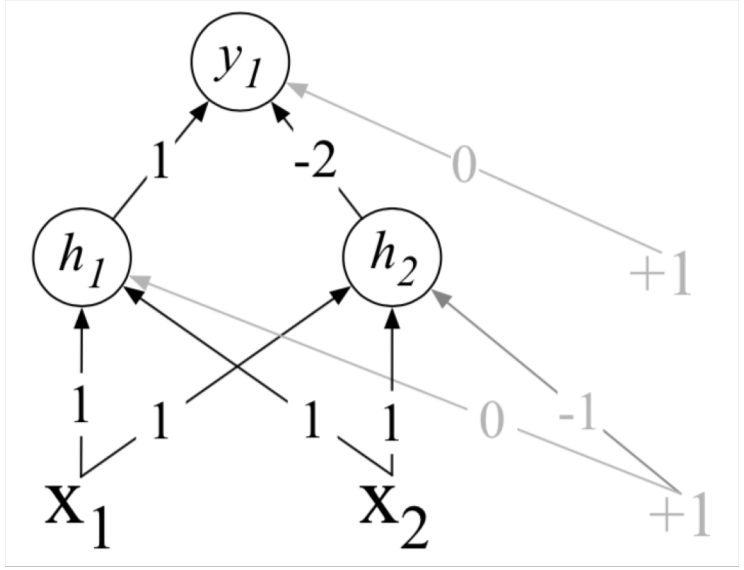
XOR Problem

- Perceptron is a **linear classifier** but XOR is not *linearly separable*
  - for a 2D input x_0 and x_1, the perceptron equation: $w_1x_1 + w_2x_2 + b = 0$ is the equation of a line



XOR Problem: Solution

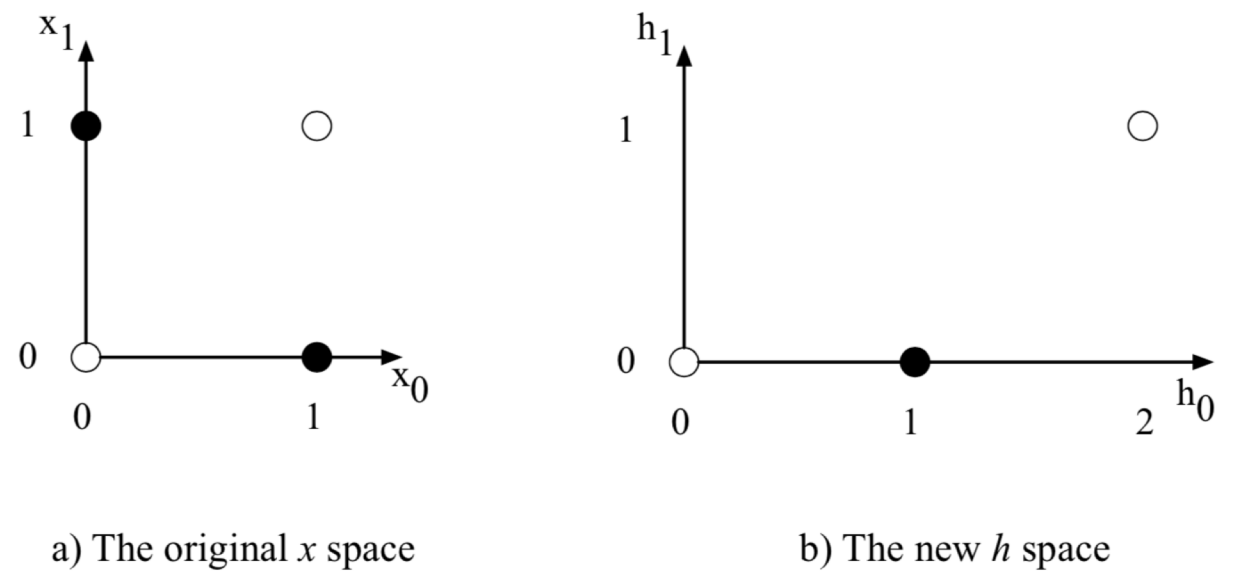
- XOR function can be computed using two layers of **ReLU**-based units



- XOR problem demonstrates need for **multi-layer** networks

XOR Problem: Solution

- Hidden layer forms a linearly separable representation for the input



In this example, we stipulated the weights but in real applications, the weights for neural networks are learned automatically using the error **back-propagation** algorithm

Why do we need non-linear activation functions?

- Network of simple linear (perceptron) units cannot solve XOR problem
  - a network formed by many layers of purely linear units can always be reduced to a single layer of linear units

$$a^{[1]} = z^{[1]} = W^{[1]} \cdot x + b^{[1]}$$

$$a^{[2]} = z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]}$$

$$= W^{[2]} \cdot (W^{[1]} \cdot x + b^{[1]}) + b^{[2]}$$

$$= (W^{[2]} \cdot W^{[1]}) \cdot x + (W^{[2]} \cdot b^{[1]} + b^{[2]})$$

$$= W' \cdot x + b'$$

  … no more expressive than logistic regression!
  - we've already shown that a single unit cannot solve the XOR problem

Feed-Forward Neural Networks

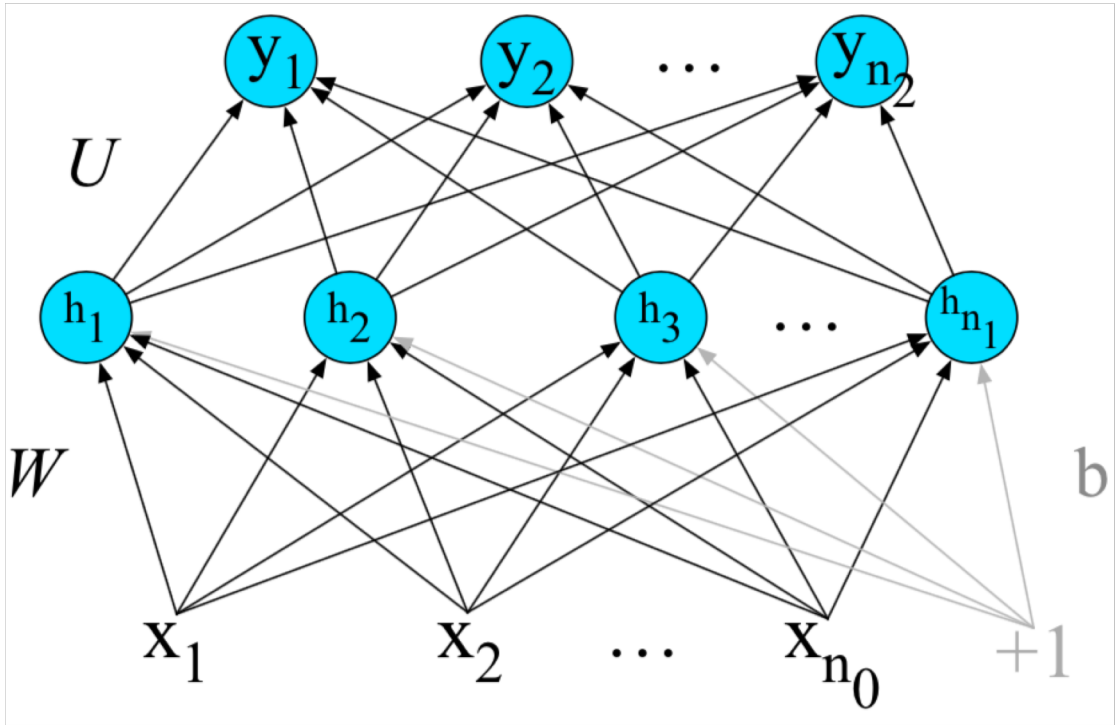
a.k.a. multi-layer perceptron (MLP), though it's a misnomer

- Each layer is **fully-connected**
- Represent parameters for hidden layer by combining weight vector w_i and bias b_i for each unit i into a single weight matrix $\boldsymbol{W}$ and a single bias vector $\boldsymbol{b}$ for the whole layer

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$h = a^{[1]} = \sigma(z^{[1]})$$

where $W \in \mathbb{R}^{n_1 \times n_0}$ and $b, h \in \mathbb{R}^{n_1}$



Feed-Forward Neural Networks

- Output could be real-valued number (for regression), or a probability distribution across the output nodes (for multinomial classification)

$$z^{[2]} = W^{[2]}h + b^{[2]}, \text{ such that } z^{[2]} \in \mathbb{R}^{n_2}, W^{[2]} \in \mathbb{R}^{n_2 \times n_1}$$

- We apply **softmax** function to encode $z^{[2]}$ as a probability distribution

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \leq i \leq d$$

- So a neural network is like *logistic regression* over induced feature *representations* from prior layers of the network rather than forming features using feature templates

Recap: 2-layer Feed-Forward Neural Network

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$
$$a^{[1]} = h = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$
$$\hat{y} \quad = a^{[2]}$$

We use $a^{[0]}$ to stand for input x, $\hat{y}$ for predicted output, y for ground truth output and $\mathrm{g}(\cdot)$ for activation function. $g^{[2]}$ might be *softmax* for multinomial classification or *sigmoid* for binary classification, while ReLU or tanh might be activation function $g(\cdot)$ at the internal layers.

N-layer Feed-Forward Neural Network

for *i* **in** 1..n:

$$z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$$

$$a^{[i]} = g^{[i]}(z^{[i]})$$

$$\hat{y} = a^{[n]}$$

Training Neural Nets: Loss Function

- Models the distance between the system output and the gold output
- Same as logistic regression, the **cross-entropy loss**
  - for binary classification

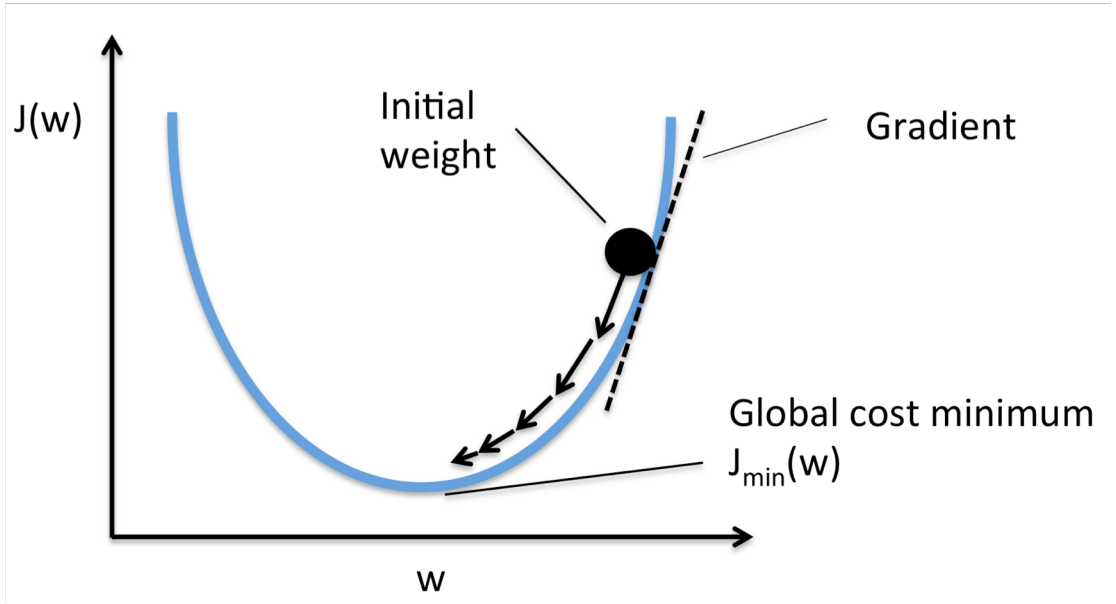
$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

  - for multinomial classification

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

Training Neural Nets: Gradient Descent

- To find parameters that minimize loss function, we use **gradient descent**
- But it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer
  - we use **error back-propagation** to partial out loss over intermediate layers
  - builds on notion of **computation graphs**

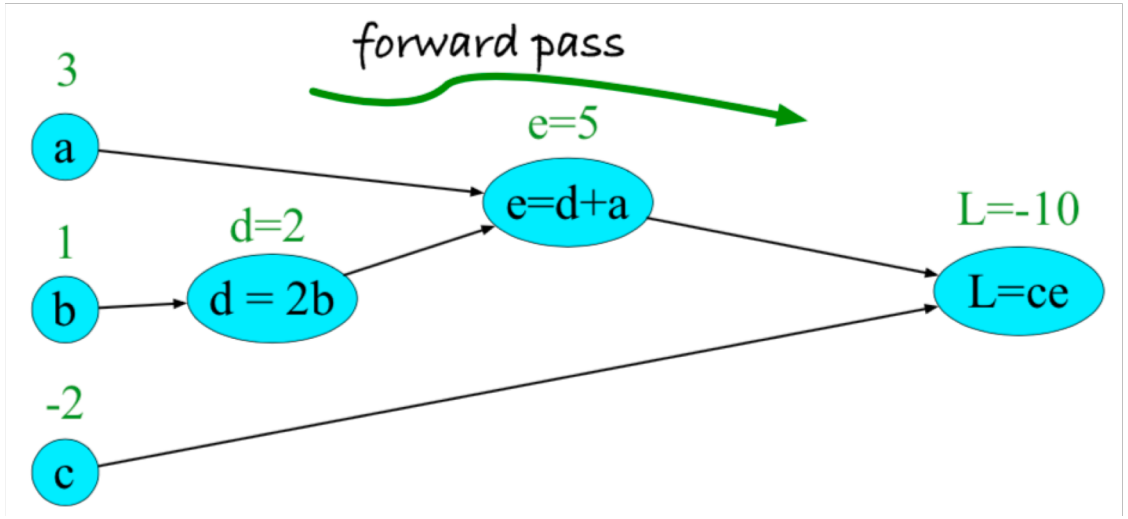


Training Neural Nets: Computation Graphs

Computation is broken down into separate operations, each of which is modeled as a node in a graph

Consider $L(a, b, c) = c(a + 2b)$

$$
\begin{aligned}
d &= 2 * b \\
e &= a + d \\
L &= c * e
\end{aligned}
$$



Training Neural Nets: Backward Differentiation

- Uses **chain rule** from calculus

  For $f(x) = u(v(x))$, we have $\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$

- For our function $L = c(a + 2b)$, we need the derivatives:

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

- Requires the intermediate derivatives:

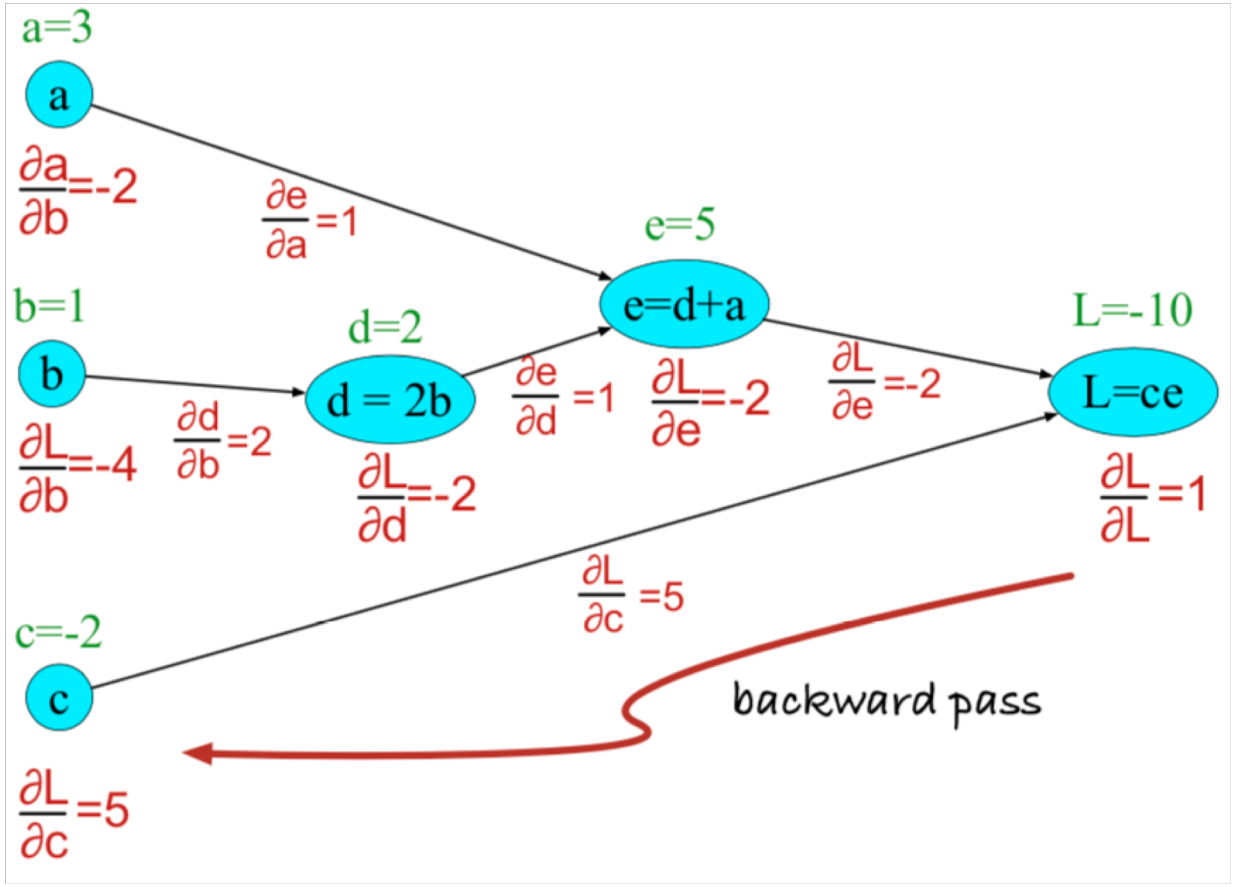
$$\begin{aligned} L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\ e = a + d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\ d = 2b &: \quad \frac{\partial d}{\partial b} = 2 \end{aligned}$$

Training Neural Nets: Backward Pass

- Compute from *right* to *left*
- For each node:
  1. compute local partial derivative with respect to the parent
  2. multiply it by the partial that is passed down from the parent
  3. then pass it to the child
- Also requires derivatives of activation functions

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-z)$$

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z)$$



Training Neural Nets: Best Practices

- **Non-convex** optimization problem
  1. initialize weights with small random numbers, preferably gaussians
  2. regularize to prevent over-fitting, e.g. **dropout**
- Optimization techniques for gradient descent
  - momentum, RMSProp, Adam, etc.

Parameters vs Hyperparameters

- **Parameters** are learned by gradient descent
  - e.g. weights matrix *W* and biases *b*
- **Hyperparameters** are set prior to learning
  - e.g. learning rate, mini-batch size, model architecture (number of layers, number of hidden units per layer, choice of activation functions), regularization technique
  - require to be tuned

Neural Language Models

Predicting upcoming words from prior word context

Neural Language Models

- Feed-forward neural LM is a standard feedforward network that takes as input at time *t a representation of some number of previous words* (w_{t-1},w_{t-2}…) and outputs *probability distribution* over possible next words
- Advantages
  - don't need smoothing
  - can handle much longer histories
  - generalize over context of similar words
  - higher predictive accuracy
- Uses include machine translation, dialog, language generation

Embeddings

- Mapping from words in vocabulary *V* to vectors of real numbers *e*
- Each word may be represented as **one hot-vector** of length |V|

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

- Concatenate each of N context vectors for preceding words
- Long, sparse, hard to generalize. Can we learn a concise representation?

Embeddings

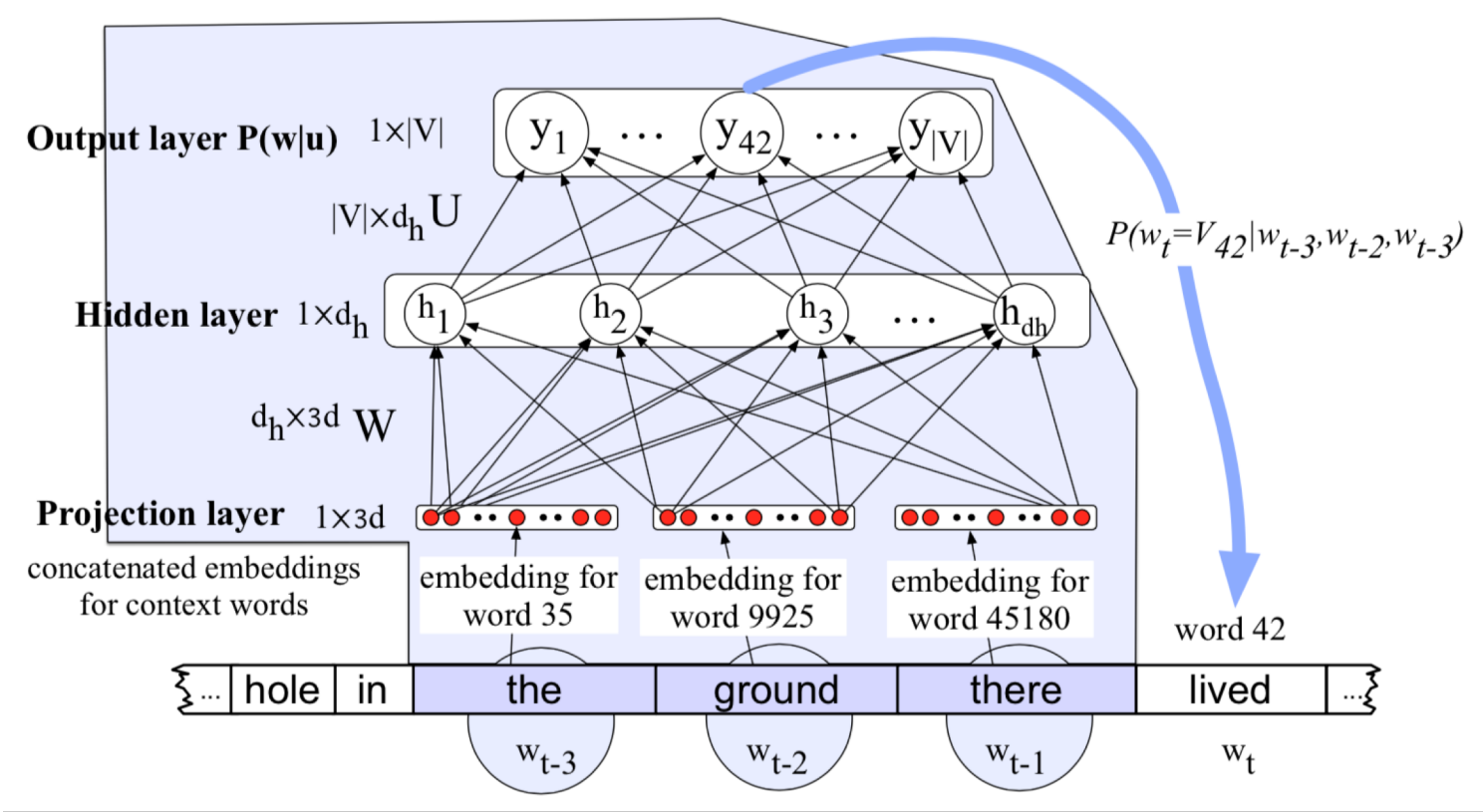
- Allow neural n-gram LM to generalize to unseen data better

“I have to make sure when I get home to feed the cat.”

If we’ve never seen the word “dog” after “feed the”, n-gram LM will predict “cat” given the prefix. But neural LM makes use of similarity of embeddings to assign a reasonably high probability to both *dog* and *cat*

Embeddings

Moving window at time *t* with **pre-trained** embedding vector, say using *word2vec* for each of three previous words w_{t-1}, w_{t-2}, and w_{t-3}, concatenated to produce input

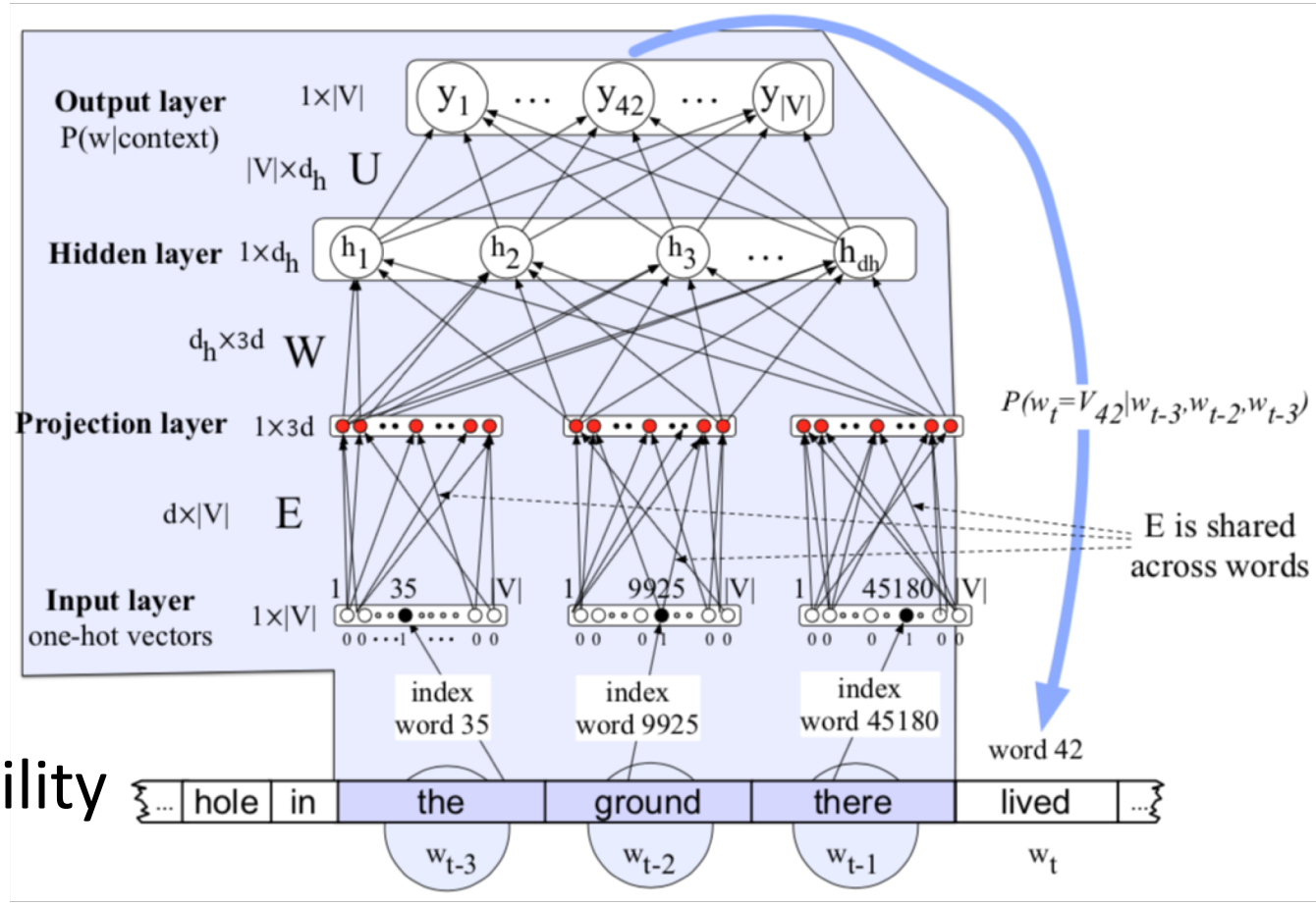


Learning Embeddings for Neural n-gram LM

- Task may place strong constraints on what makes a good *representation*
- To learn embeddings, add an extra layer to the network and propagate errors all the way back to the embedding vectors
- Represent each of *N* previous words as *one hot-vector* of length |V|, and learn an embedding matrix $\mathrm{E} \in \mathbb{R}^{d \times V}$ such that for one-hot column vector x_i for word V_i, the **projection layer** is $Ex_i = e_i$

Learning Embeddings: Forward Pass

$$a^{[0]} = e = (Ex_1, Ex_2, \ldots, Ex_n)$$
$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$\hat{y} \quad = a^{[2]} = g^{[2]}(z^{[2]})$$



Each node *i* in $\hat{y}$ estimates probability $P(w_t\_i \mid w_{t-1}, w_{t-2}, w_{t-3})$

Training the Neural Language Model

- To set all the parameters θ = *E*,*W*,*U*,*b*, we do **gradient descent** using error **back propagation** on the **computation graph** to compute gradient
- Loss Function: **cross-entropy** (negative log likelihood)

$$L = -\log \mathrm{p}\left(w_{t_i} \mid w_{t-1}, w_{t-2}, w_{t-n+1}\right)$$

Training the parameters to minimize loss will result both in an algorithm for language modeling (a word predictor) but also a new set of embeddings E

Summary

- Neural networks are built out of **neural units**, which take weighted sum of inputs and apply a non-linear **activation function** such as sigmoid, tanh, ReLU
- In a fully-connected **feed-forward network**, each unit in layer *i* is connected to each unit in layer *i* + 1, and there are no cycles
- Power of neural networks comes from the ability of early layers to **learn representations** that can be utilized by later layers in the network
- Neural networks are trained by optimization algorithms like **gradient descent** using **error back-propagation** on a **computation graph**
- **Neural language models** use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous *n* words
- Neural language models can use **pretrained** embeddings, or can learn **embeddings** from scratch in the process of language modeling