Announcements, 2018-01-24

- HW #2 due Monday, 29 Jan 2018, at 11:59pm.

RECAP, 2018-01-22

Classification through linear classifiers and scoring.

$$y^{(i)} = W x^{(i)} + b \qquad (10\times 1)$$

$$(10\times 1) \quad (10\times 3072)(3072\times 1)$$

$$\hat{y}^{(i)} = \begin{bmatrix} 300 \\ 500 \\ -150 \\ \vdots \end{bmatrix}$$

True class 1

Predict class ~~2~~ 10

$$\sum_i \| y^{(i)} - \hat{y}^{(i)} \|^2$$

Turn the scores into a probability

A first thought is to turn the scores into probabilities.

$$\text{softmax}_i(\mathbf{x}) = \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}}$$

$= \Pr(\mathbf{x}$ belongs to class $i)$

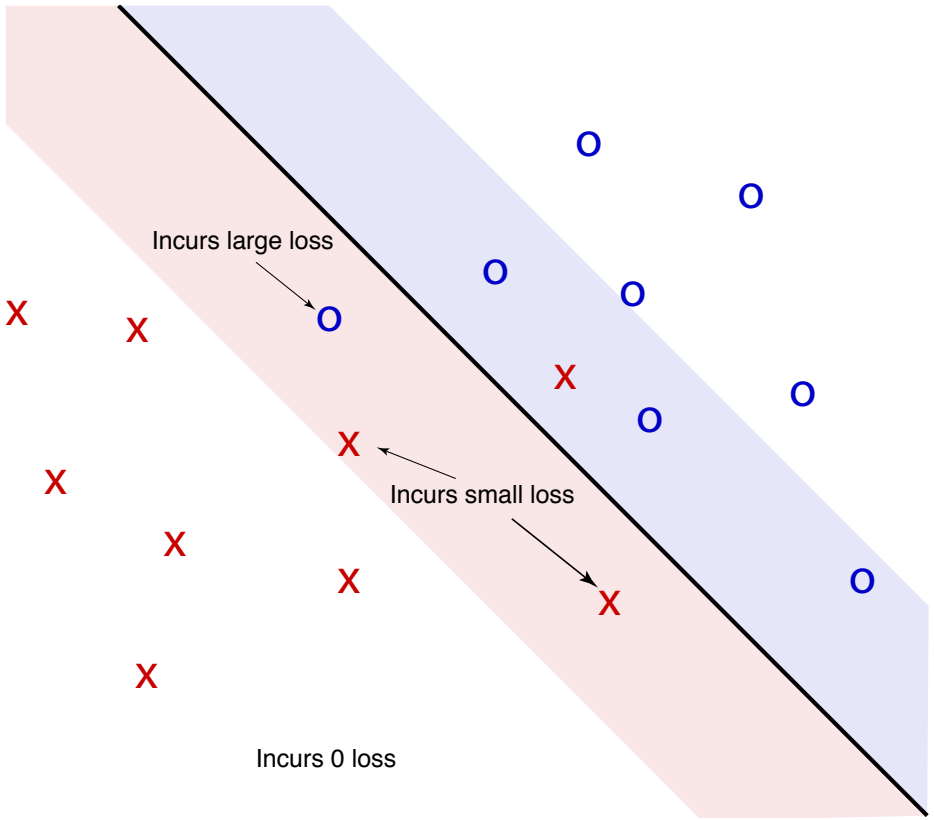
Use the maximum likelihood principle

Turn this into a cost function by using the maximum likelihood principle:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left(\log \sum_{j=1}^{c} e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)$$

Hinge loss intuition

The intuition of the prior slide is that the hinge loss is greatest for misclassifications, and the greater the error in misclassification, the worse the loss. For correct classifications, the loss will be zero only if there is a large enough margin.



Hinge loss

Hinge loss:

$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

0 and 1

$$1 + w_j^T x^{(i)} - w_{y^{(i)}}^T x^{(i)}$$

if $w_{y^{(i)}}^T x^{(i)} > w_j^T x^{(i)} + 1 \Rightarrow 0$

if $w_{y^{(i)}}^T x^{(i)} > w_j^T x^{(i)} \Rightarrow$ small loss

if $w_j^T x^{(i)} > w_{y^{(i)}}^T x^{(i)} \Rightarrow$ large loss

$$\begin{bmatrix} 0 \\ - \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} \\ \\ \end{bmatrix}$$

SVM loss

SVM loss:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

Two reasonable loss functions

Softmax:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left(\log \sum_{j=1}^{c} e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)$$

SVM:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

Parameters? W, b

$$\begin{bmatrix} a_1(x) \\ a_2(x) \\ \vdots \\ a_c(x) \end{bmatrix} = \underbrace{\begin{bmatrix} — & w_1^T & — \\ — & w_2^T & — \\ & \vdots & \\ — & w_c^T & — \end{bmatrix}}_{W} x + \underbrace{b}_{b}$$

Two reasonable loss functions

Softmax:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left(\log \sum_{j=1}^{c} e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)$$

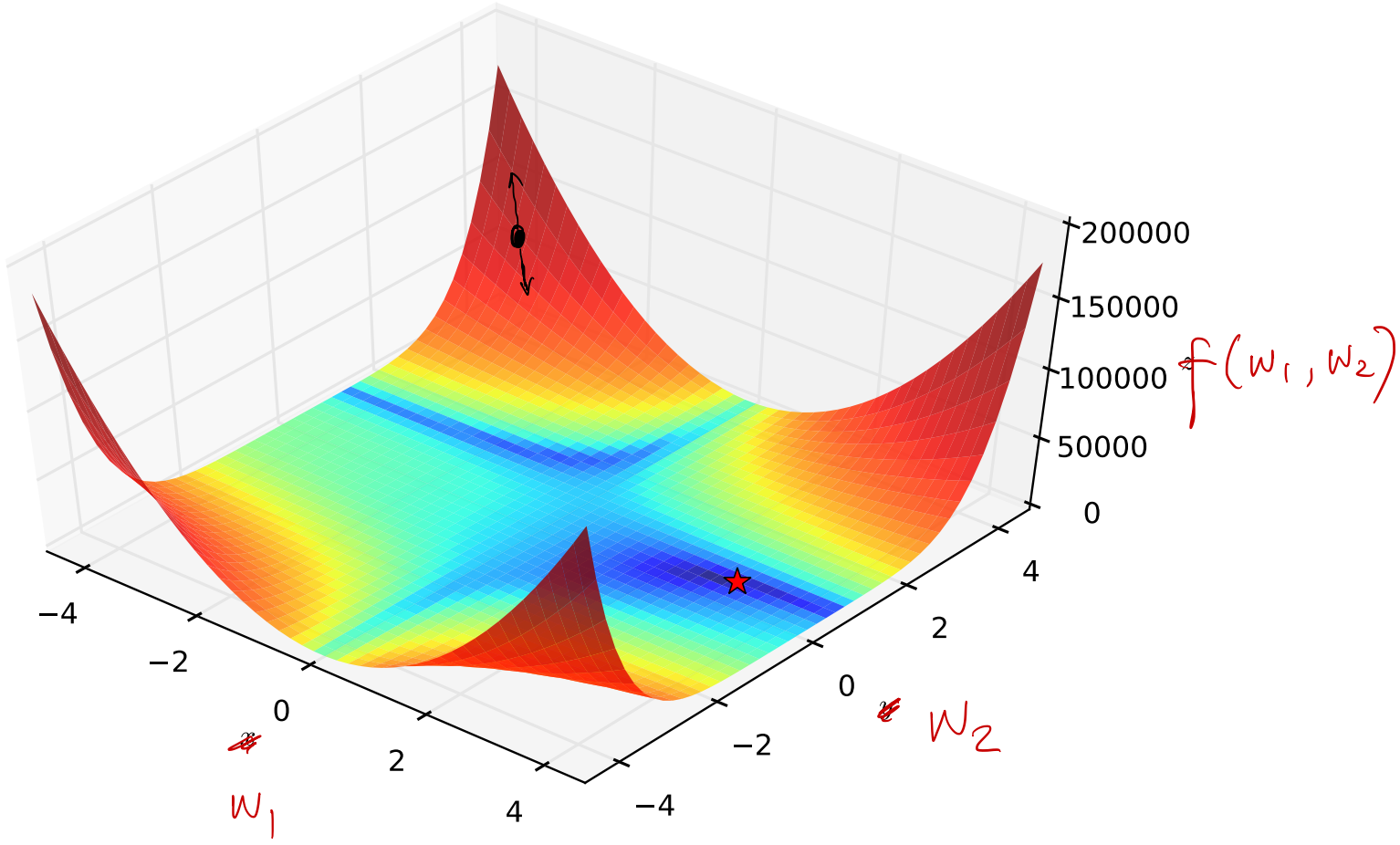
SVM:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

Parameters: $\mathbf{W} \in \mathbb{R}^{c \times n}, \mathbf{b} \in \mathbb{R}^{c}$

Big question: how do we find these parameters?

Finding the optimal weights through gradient descent

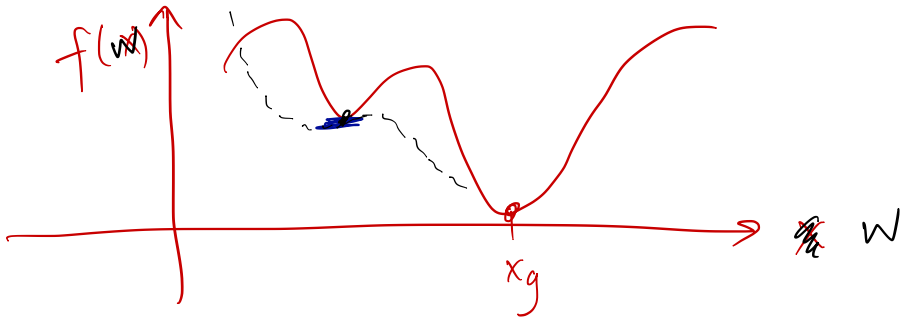


Finding the optimal weights through gradient descent

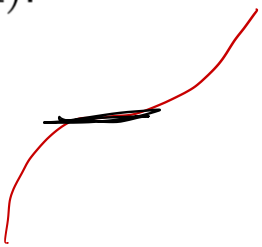
- Our goal in machine learning is to optimize an objective function, $f(x)$. (Without loss of generality, we'll consider minimizing $f(x)$. This is equivalent to maximizing $-f(x)$.)
- From basic calculus, we recall that the derivative of a function, $\frac{df(x)}{dx}$ tells us the slope of $f(x)$ at point x.
  - For small enough ϵ, $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$.
  - This tells us how to reduce (or increase) $f(\cdot)$ for small enough steps.
  - Recall that when $f'(x) = 0$, we are at a stationary point or critical point. This may be a local or global minimum, a local or global maximum, or a saddle point of the function.
- In this class we will consider cases where we would like to maximize f w.r.t. vectors and matrices, e.g., $f(\mathbf{x})$ and $f(\mathbf{X})$.
- Further, often $f(\cdot)$ contains a nonlinearity or non-differentiable function. In these cases, we can't simply set $f'(\cdot) = 0$, because this does not admit a closed-form solution.
- However, we can iteratively approach an critical point via gradient descent.

Finding the optimal weights through gradient descent

Terminology



- A **global minimum** is the point, $\mathbf{x}_g$, that achieves the absolute lowest value of $f(\mathbf{x})$. i.e., $f(\mathbf{x}) \geq f(\mathbf{x}_g)$ for all $\mathbf{x}$.
- A **local minimum** is a point, $\mathbf{x}_\ell$, that is a critical point of $f(\mathbf{x})$ and is lower than its neighboring points. However, $f(\mathbf{x}_\ell) > f(\mathbf{x}_g)$.
- Analogous definitions hold for the **global maximum** and **local maximum**.
- A **saddle point** are critical point of $f(\mathbf{x})$ that are not local maxima or minima. Concretely, neighboring points are both greater than and less than $f(\mathbf{x})$.



Finding the optimal weights through gradient descent

Gradient

Recall the gradient, $\nabla_{\mathbf{x}} f(\mathbf{x})$, is a vector whose ith element is the partial derivative of $f(\mathbf{x})$ w.r.t. x_i, the ith element of $\mathbf{x}$. Concretely, for $\mathbf{x} \in \mathbb{R}^n$,

$$\Delta x = \begin{bmatrix} \delta x_1 \\ \delta x_2 \\ \vdots \\ \delta x_n \end{bmatrix} \qquad \nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial (x_n)} \end{bmatrix}$$

- The gradient tells us how a small change in $\Delta\mathbf{x}$ affects $f(\mathbf{x})$ through

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \Delta\mathbf{x}^T \nabla_{\mathbf{x}} f(\mathbf{x})$$

- The directional derivative of $f(\mathbf{x})$ in the direction of the unit vector $\mathbf{u}$ is given by:

$$\mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x})$$

Finding the optimal weights through gradient descent

Arriving at gradient descent

- To minimize $f(\mathbf{x})$, we want to find the direction in which $f(\mathbf{x})$ decreases the fastest. To do so, we find the direction $\mathbf{u}$ which minimizes the directional derivative.

$$\begin{aligned}\min_{\mathbf{u},\|\mathbf{u}\|=1} \mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x}) &= \min_{\mathbf{u},\|\mathbf{u}\|=1} \|\mathbf{u}\| \, \|\nabla_{\mathbf{x}} f(\mathbf{x})\| \cos\theta \\ &= \min_{\mathbf{u}} \|\nabla_{\mathbf{x}} f(\mathbf{x})\| \cos(\theta)\end{aligned}$$

  where θ is the angle between the vectors $\mathbf{u}$ and $\nabla_{\mathbf{x}} f(\mathbf{x})$.

- This quantity is minimized for $\mathbf{u}$ pointing in the opposite direction of the gradient, so that $\cos(\theta) = -1$.
- Hence, we arrive at gradient descent. To update $\mathbf{x}$ so as to minimize $f(\mathbf{x})$, we repeatedly calculate:

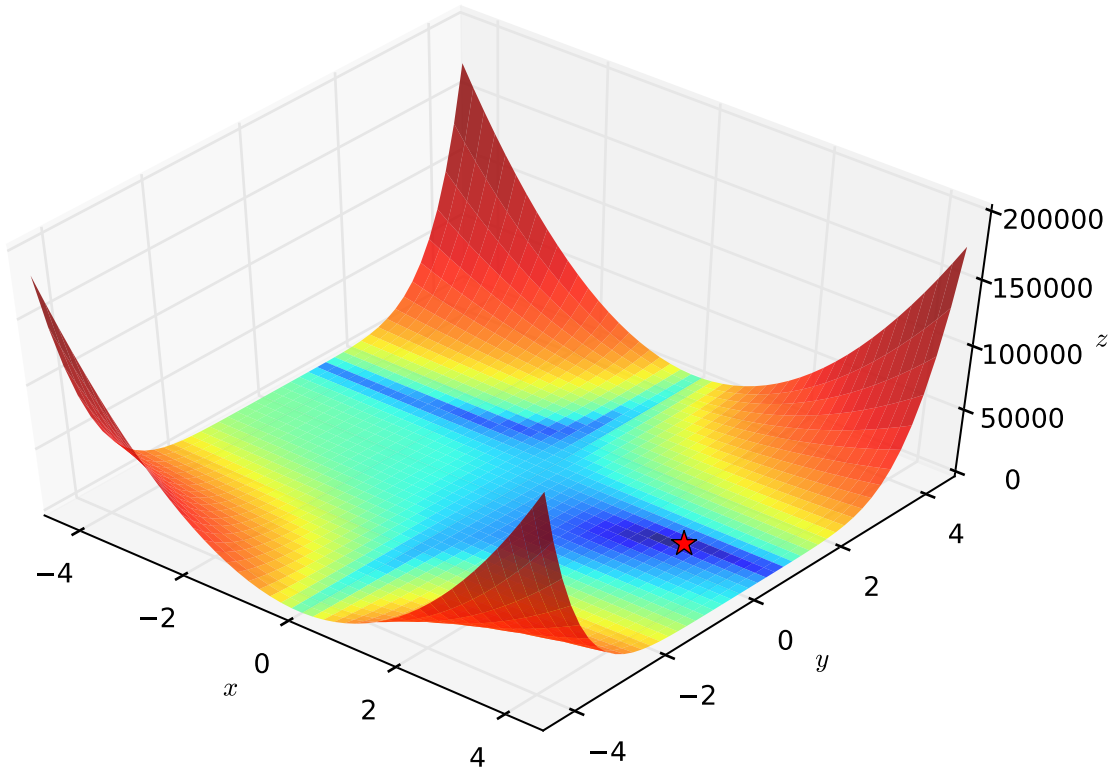
$$\mathbf{x} := \mathbf{x} - \epsilon \nabla_x f(\mathbf{x})$$

- ϵ is typically called the *learning rate*. It can change over iterations. Setting the value of ϵ appropriately is an important part of deep learning.

Finding the optimal weights through gradient descent

Example:

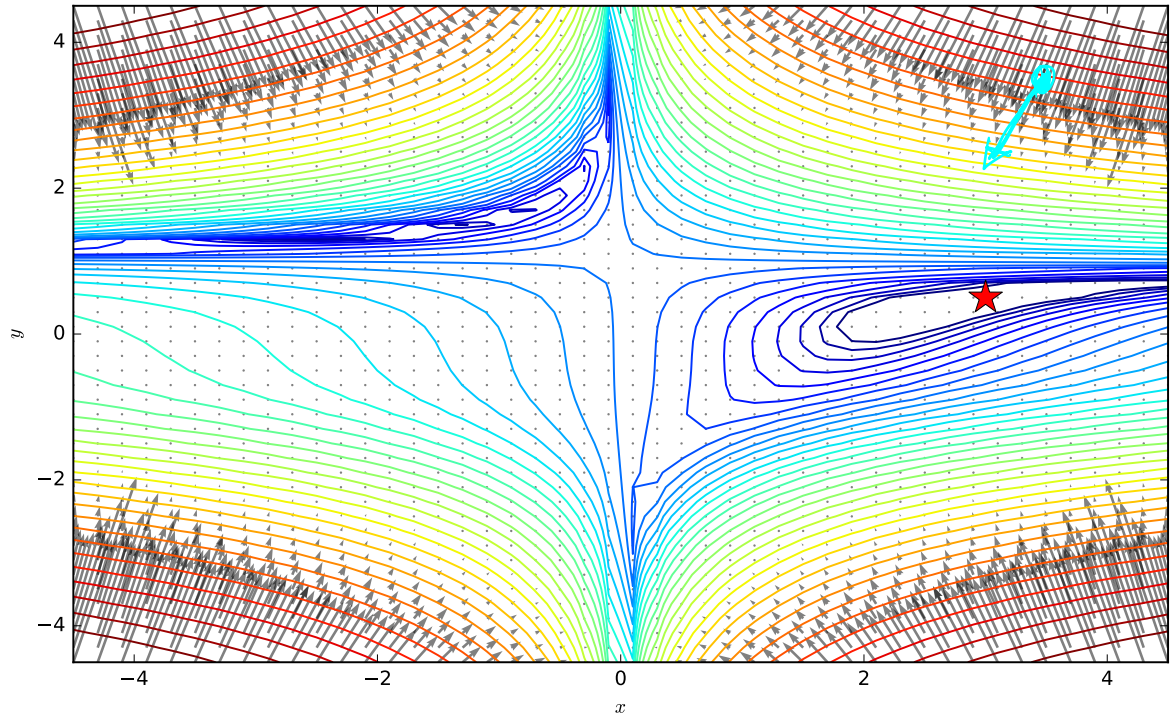
Animations thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/



Finding the optimal weights through gradient descent

Example:

Animations thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/



Finding the optimal weights through gradient descent

```
def gd(func, x0, eps=1e-4, tol=1e-3):
    last_diff = np.Inf
    x = x0
    path = [np.copy(x0)]
    costs = [func(x0)[0]]
    grads = []
    i = 1
    hit_max = False

    while last_diff > tol:
        cost, g = func(x) # returns the cost and the gradient
        x -= eps*g # gradient step
        last_diff = np.linalg.norm(x - path[-1]) # stopping criterion

        i += 1
        if i > max_iters:
            hit_max = True
            break
        path.append(np.copy(x))
        costs.append(cost)
        grads.append(g)

    return path, costs, grads, hit_max
```

svm.loss_and_grad()

x = x - eps · g

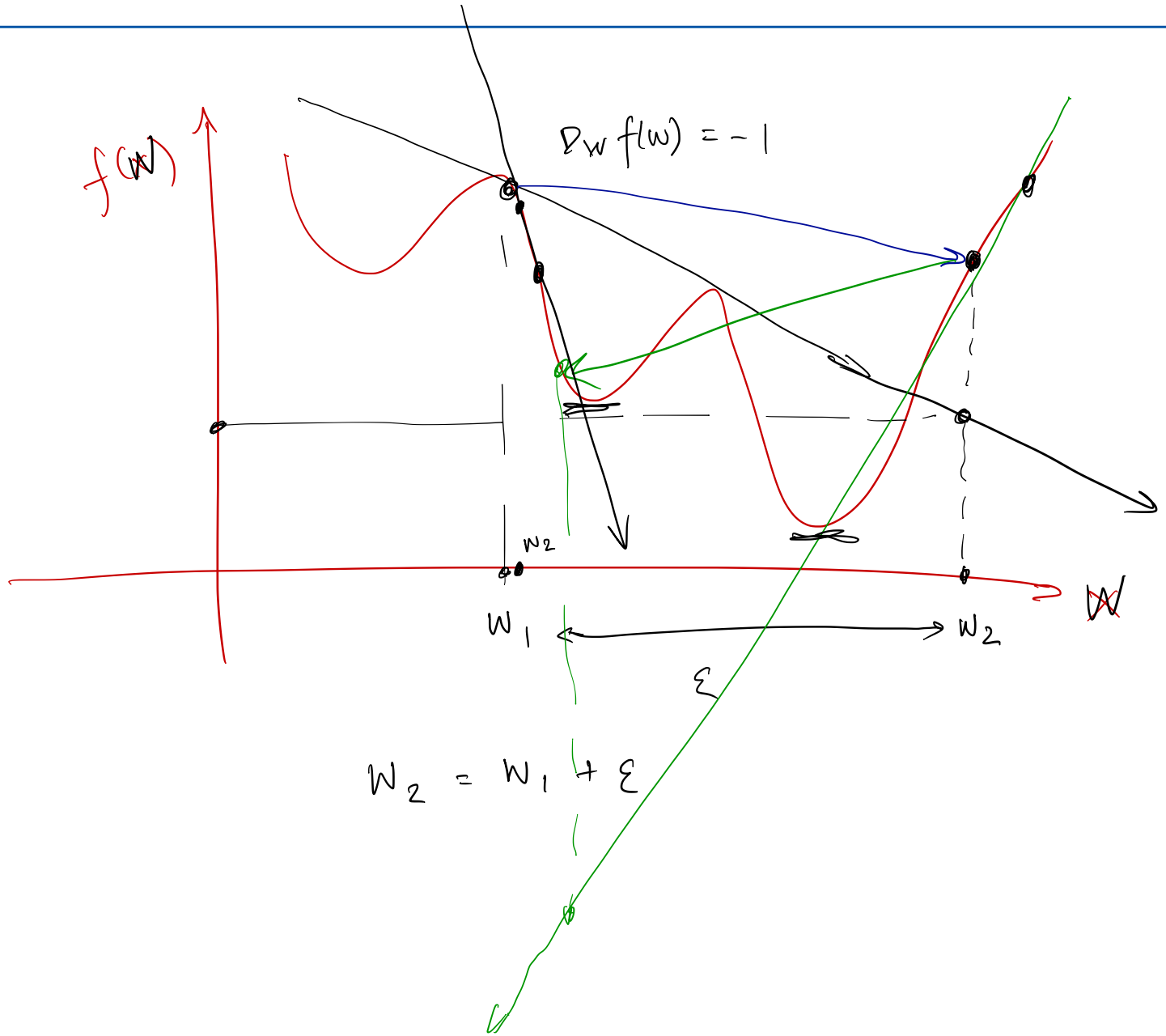
Finding the optimal weights through gradient descent

http://seas.ucla.edu/~kao/opt_anim/1gd.mp4
http://seas.ucla.edu/~kao/opt_anim/2gd.mp4

Why not use a numerical gradient?

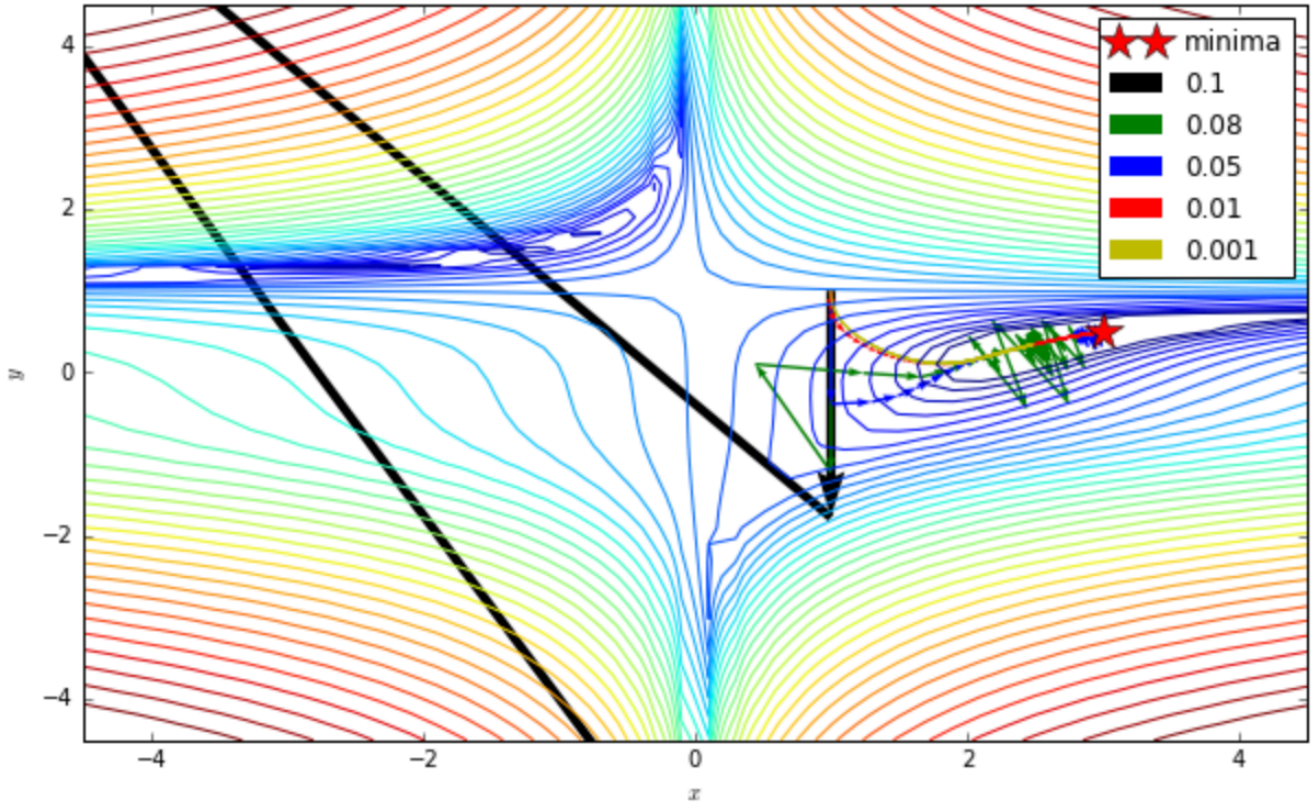
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

How do I pick a right step size?



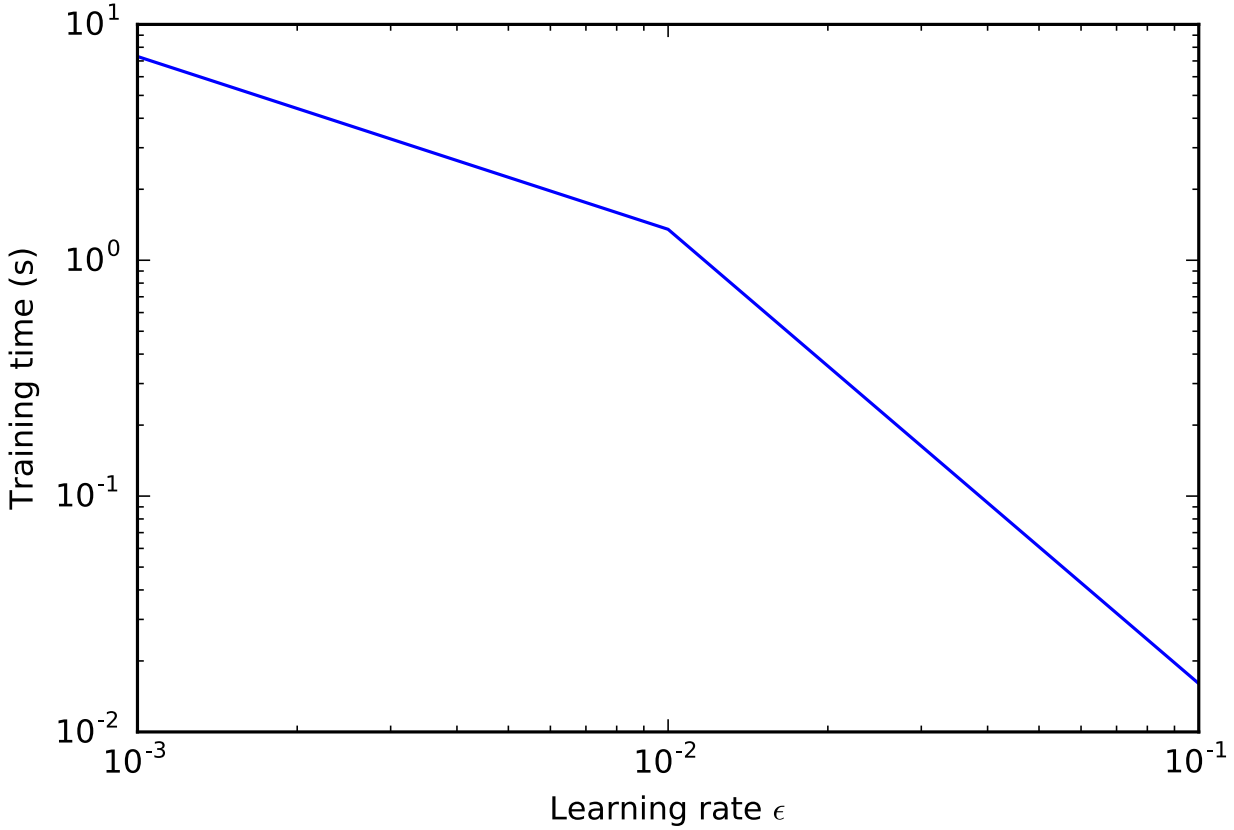
Finding the optimal weights through gradient descent

Varying the learning rate:

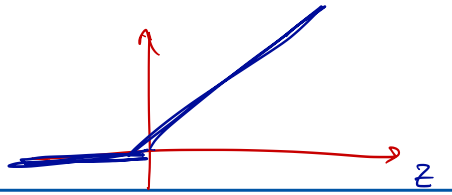


Finding the optimal weights through gradient descent

Why not always use smaller learning rates?



Hinge loss (sub)gradient



$$\frac{1}{m}\sum_{i=1}^{m}\sum_{j\neq y^{(i)}}\max\left(0, \underbrace{1+w_j^T x^{(i)} - w_{y^{(i)}}^T x^{(i)}}_{z_j}\right)$$

L_i

$$\nabla_x \left(w_j^T x^{(i)}\right) = x^{(i)}$$

$$\nabla_{w_j} L_i = \begin{cases} 0 & \text{if } z_j \leq 0 \\ x^{(i)} & \text{if } z_j > 0 \end{cases} = \mathbb{I}(z_j > 0)\, x^{(i)}$$

$$\nabla_{w_{y^{(i)}}} L_i = -\sum_{j\neq y^{(i)}} \mathbb{I}(z_j > 0)\, x^{(i)}$$

Hinge loss (sub)gradient

Finding the optimal weights through gradient descent

How does this example differ from what we will really encounter?

- In the illustrated gradient descent, we know the function f() exactly, and thus at every point in space, we can calculate the gradient at that point exactly.
- In optimization, we differentiate the cost function f() with respect to the parameters.
  - The gradient of f() w.r.t. parameters is a function of the training data!
  - Hence, we can think of each data point as providing a noisy estimate of the gradient at that point.

Finding the optimal weights through gradient descent

To get a more robust estimate of the gradient, we would use as many data samples as possible.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)$$

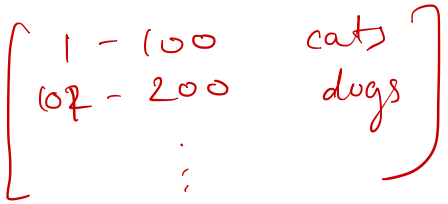
and its gradient is:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta) \\ &= \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta) \\ &\approx \mathbb{E}\left[\nabla_\theta \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)\right] \end{aligned}$$

Finding the optimal weights through gradient descent

However, it's expensive to have to calculate the gradient by using *every example* in the training set.

To this end, we may want to get a noisier estimate of the gradient with fewer examples.



Batch vs minibatch (cont)

Calculating the gradient exactly is expensive, because it requires evaluating the model on all m examples in the dataset. This leads to an important distinction.

- Batch algorithm: uses all m examples in the training set to calculate the gradient.
- Minibatch algorithm: approximates the gradient by calculating it using k training examples, where $m > k > 1$.
- Stochastic algorithm: approximates the gradient by calculating it over one example.

It is typical in deep learning to use minibatch gradient descent. Note that some may also use minibatch and stochastic gradient descent interchangeably.

Finding the optimal weights through gradient descent

You'll do this in the HW. More on this later in the optimization lecture…

- And a lot more to be said about optimization.
- First order vs second order methods
- Momentum
- Adaptive gradients.
- … all of these will become quite important when we get to neural networks. We'll cover these in an optimization lecture.

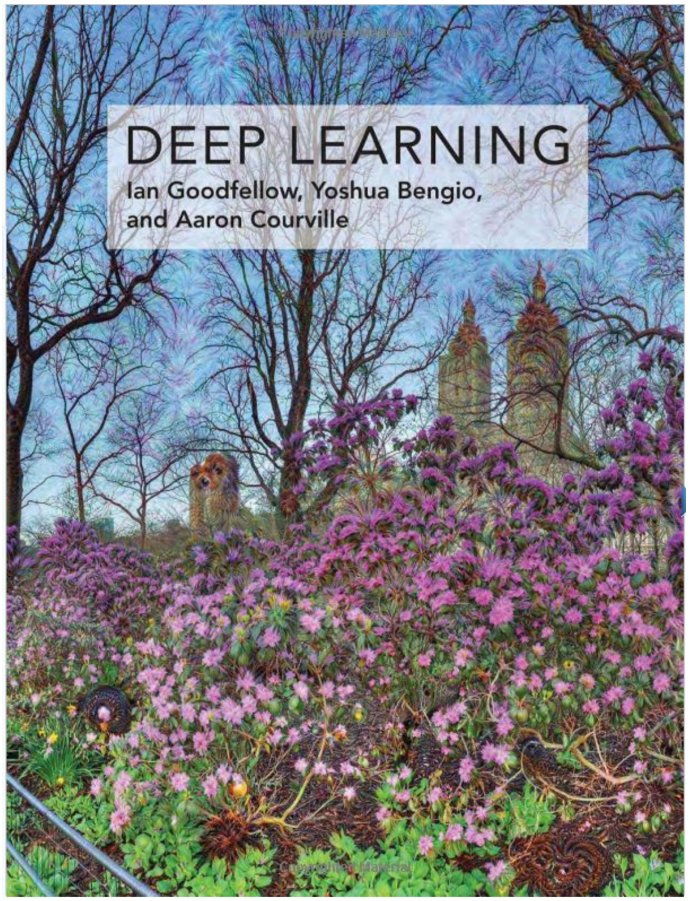
Lecture 4: Neural networks

In this lecture, we'll introduce the neural network architecture, parameters, and its inspiration from biological neurons.

Announcements, 2018-01-22

Reading:

Deep Learning, 6 (intro), 6.1, 6.2, 6.3, 6.4

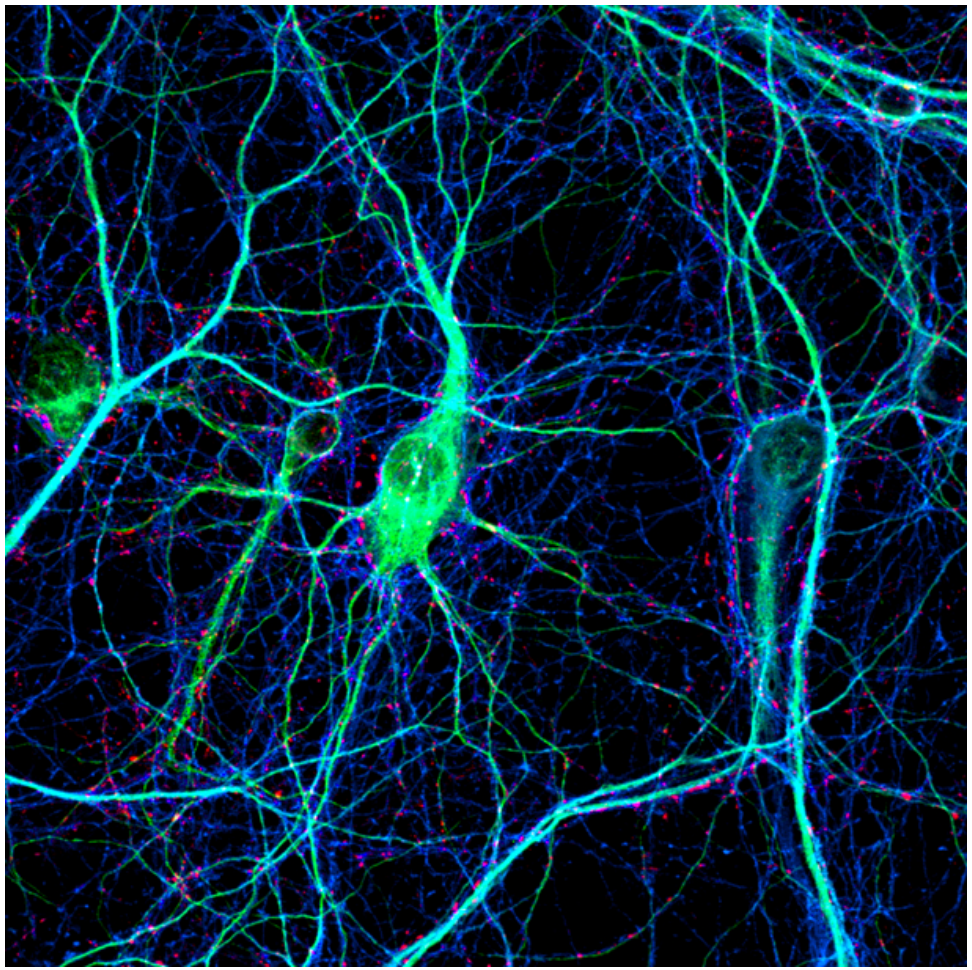


Inspiration from neuroscience

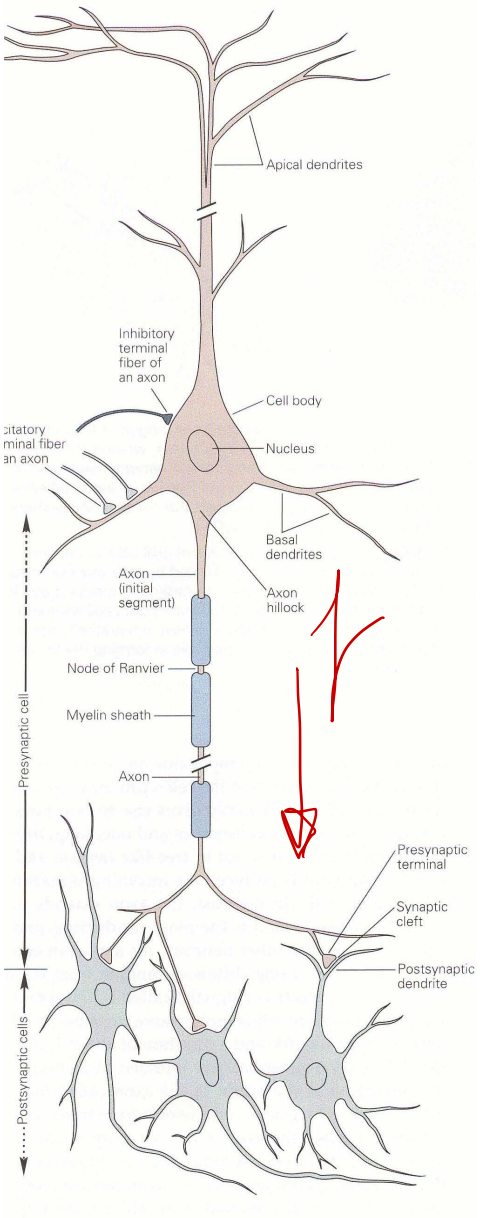
Neuroscience

Neural networks, as their name suggests, take their inspiration from neurons. However, the connection is very loose. There are many known aspects of neural signaling that are not incorporated into artificial neural networks. That said, it is accurate to state that the basic computing principle for artificial neural networks is inspired by neuroscience.

Inspiration from neuroscience



Inspiration from neuroscience



Neurons are the main signaling units of the nervous system.

- Neurons have:
  1. Dendrites – tree like structure for receiving **input** signals.
  2. The axon hillock, an integration center — for summing propagated input signals.
  3. Axon – single, long, tubular structure for sending **output** signals.

- Axons (the **output)** convey signals to other neurons:
  - Conveys electrical signals long distances (0.1mm – 3 m).
  - Conveys action potentials (~100 mV, ~1 ms pulses).
  - Action potentials initiate at the axon hillock.
  - Propagate w/o distortion or failure at 1-100 m/s.

Inspiration from neuroscience

Neurons are diverse (unlike in neural networks)

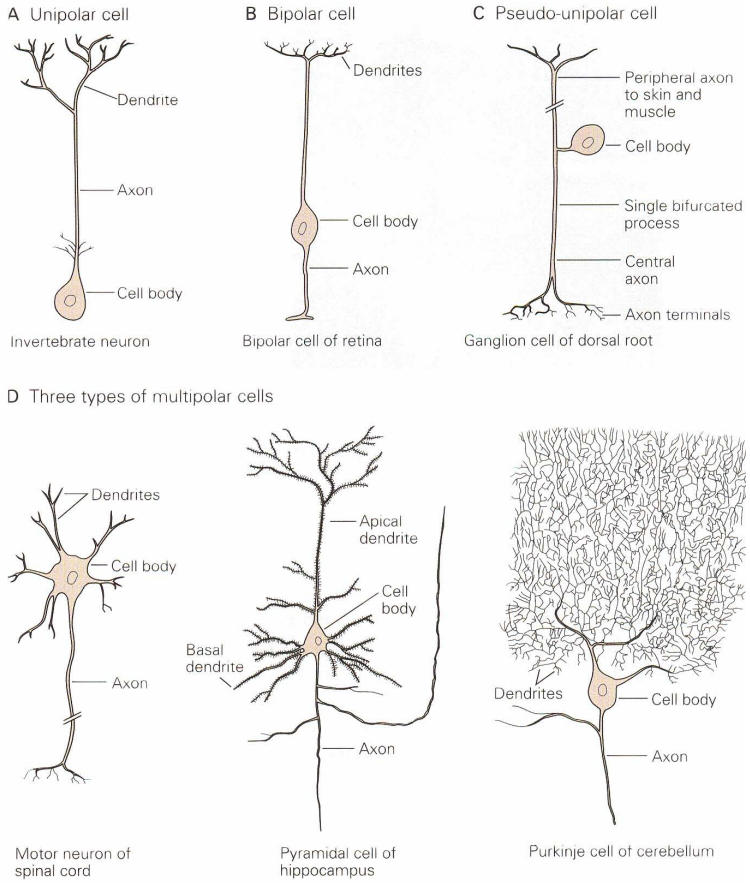


Figure 2-4 Neurons can be classified as unipolar, bipolar, or multipolar according to the number of processes that originate from the cell body.

A. Unipolar cells have a single process, with different segments serving as receptive surfaces or releasing terminals. Unipolar cells are characteristic of the invertebrate nervous system.

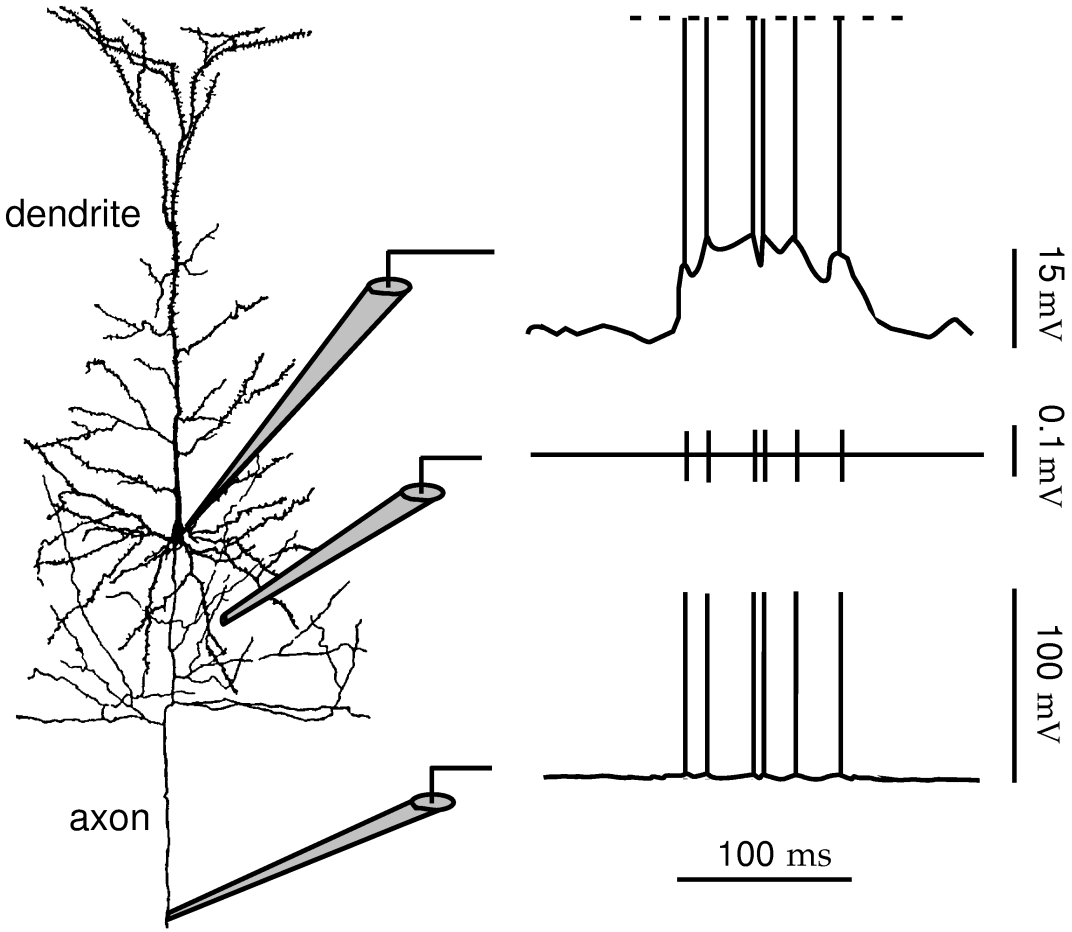
B. Bipolar cells have two processes that are functionally specialized: the dendrite carries information to the cell, and the axon transmits information to other cells.

C. Certain neurons that carry sensory information, such as information about touch or stretch, to the spinal cord belong to a subclass of bipolar cells designated as pseudo-unipolar. As such cells develop, the two processes of the embryonic bipolar cell become fused and emerge from the cell body as a single process. This outgrowth then splits into two processes, *both* of which function as axons, one going to peripheral skin or muscle, the other going to the central spinal cord.

D. Multipolar cells have an axon and many dendrites. They are the most common type of neuron in the mammalian nervous system. Three examples illustrate the large diversity of these cells. Spinal motor neurons (left) innervate skeletal muscle fibers. Pyramidal cells (middle) have a roughly triangular cell body; dendrites emerge from both the apex (the apical dendrite) and the base (the basal dendrites). Pyramidal cells are found in the hippocampus and throughout the cerebral cortex. Purkinje cells of the cerebellum (right) are characterized by the rich and extensive dendritic tree in one plane. Such a structure permits enormous synaptic input. (Adapted from Ramón y Cajal 1933.)

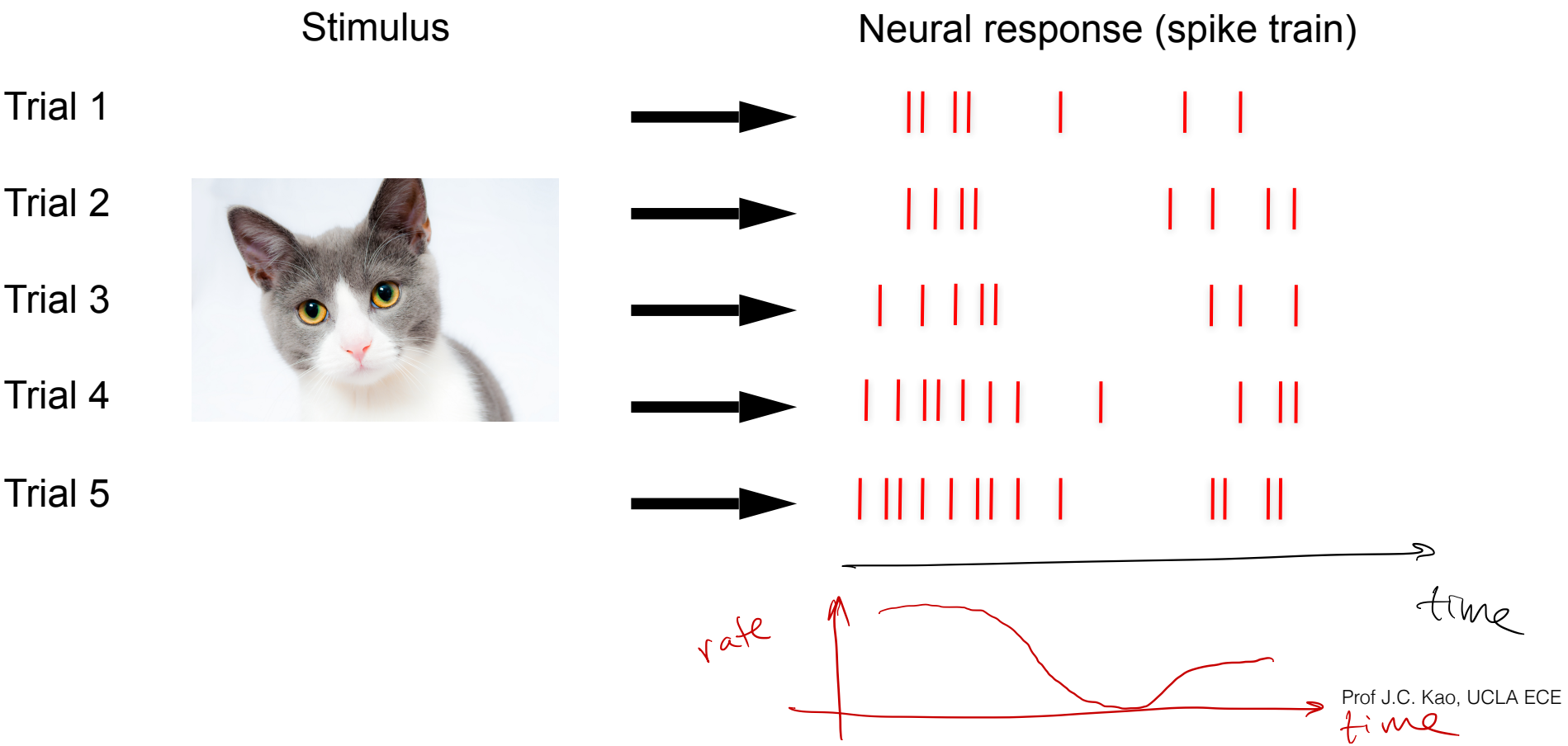
Inspiration from neuroscience

Neurons fundamentally communicate through all-or-nothing spikes (not analog values!):



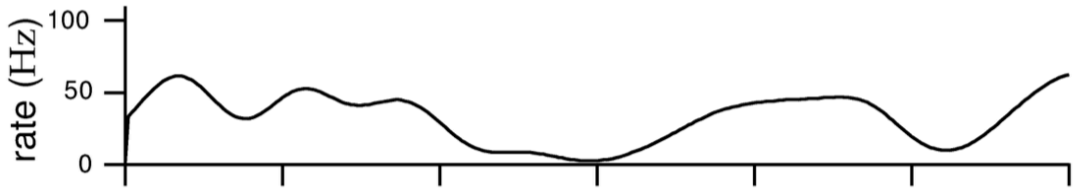
Inspiration from neuroscience

… And the spikes are probabilistic.



Inspiration from neuroscience

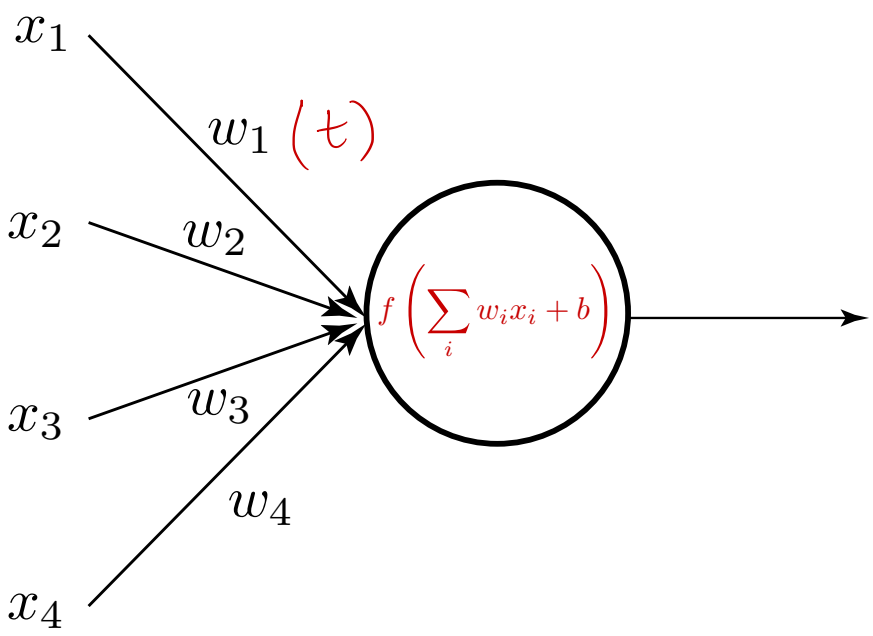
The spikes reflect an underlying rate.



This rate is what the neural networks are “encoding.”

Inspiration from neuroscience

How does the artificial neuron compare to the real neuron?



Inspiration from neuroscience

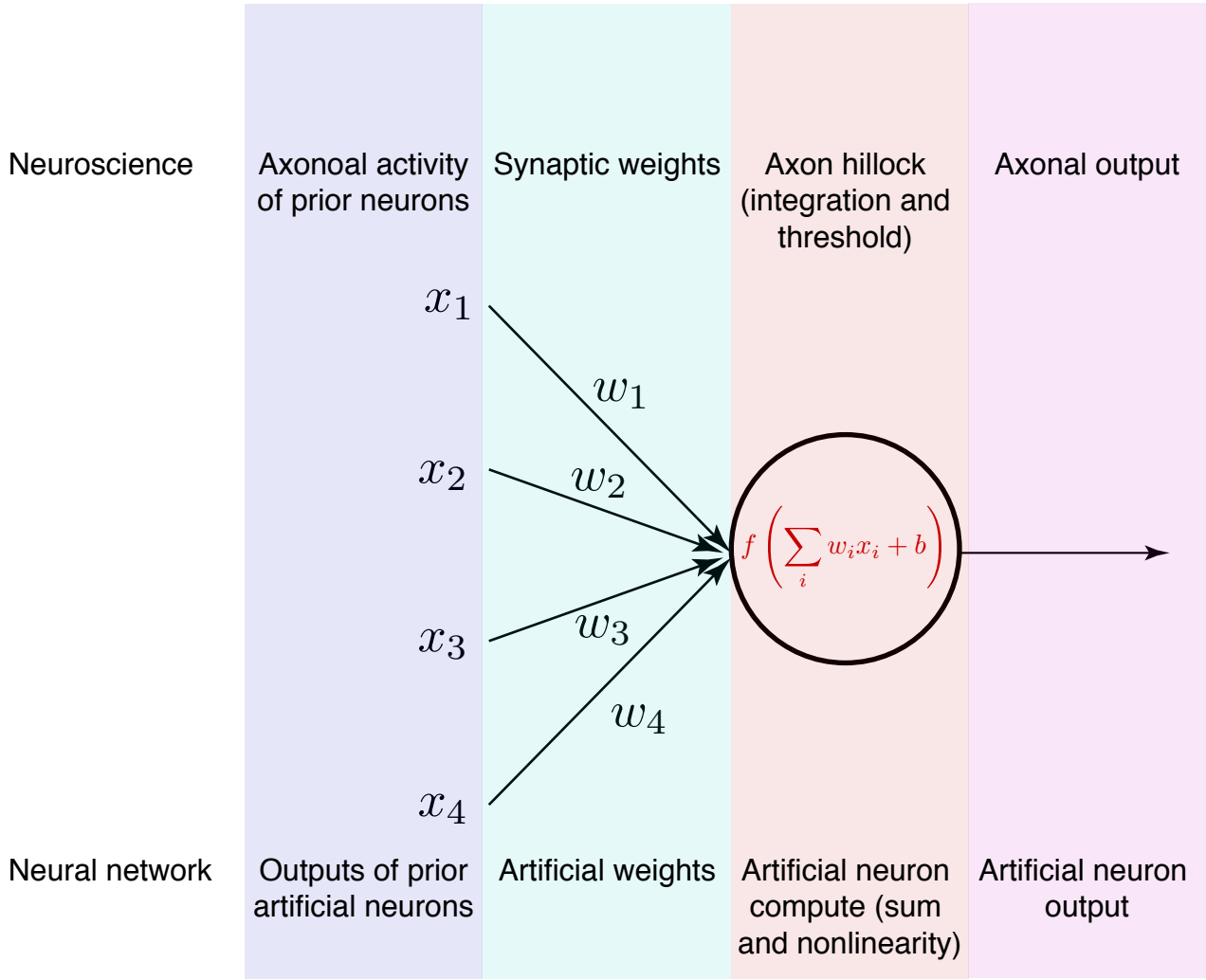
How does the artificial neuron compare to the real neuron?

The artificial neuron (cont.)

- The incoming signals, a vector $\mathbf{x} \in \mathbb{R}^N$, reflects the output of N neurons that are connected to the current artificial neuron.
- The incoming signals, $\mathbf{x}$, are pointwise multiplied by a vector, $\mathbf{w} \in \mathbb{R}^N$. That is, we calculate $w_i x_i$ for $i = 1, \ldots, N$. This computation reflects dendritic processing.
- The "dendritic-processed" signals are then summed, i.e., we calculate $\sum_i w_i x_i + b$. This computation reflects integration at the axon hillock (the first "Node of Ranvier") where action potentials are generated if the integrated signal is large enough.
- The output of the artificial neuron is then a nonlinearly transformation of the integrated signal, i.e., $f(\sum_i w_i x_i + b)$. Rather than reflecting whether an action potential was generated or not (which is a noisy process), this nonlinear output is typically treated as the *rate* of the neuron. The higher the rate, the more likely the neuron is to fire action potentials.

Inspiration from neuroscience

How does the artificial neuron compare to the real neuron?



Inspiration from neuroscience

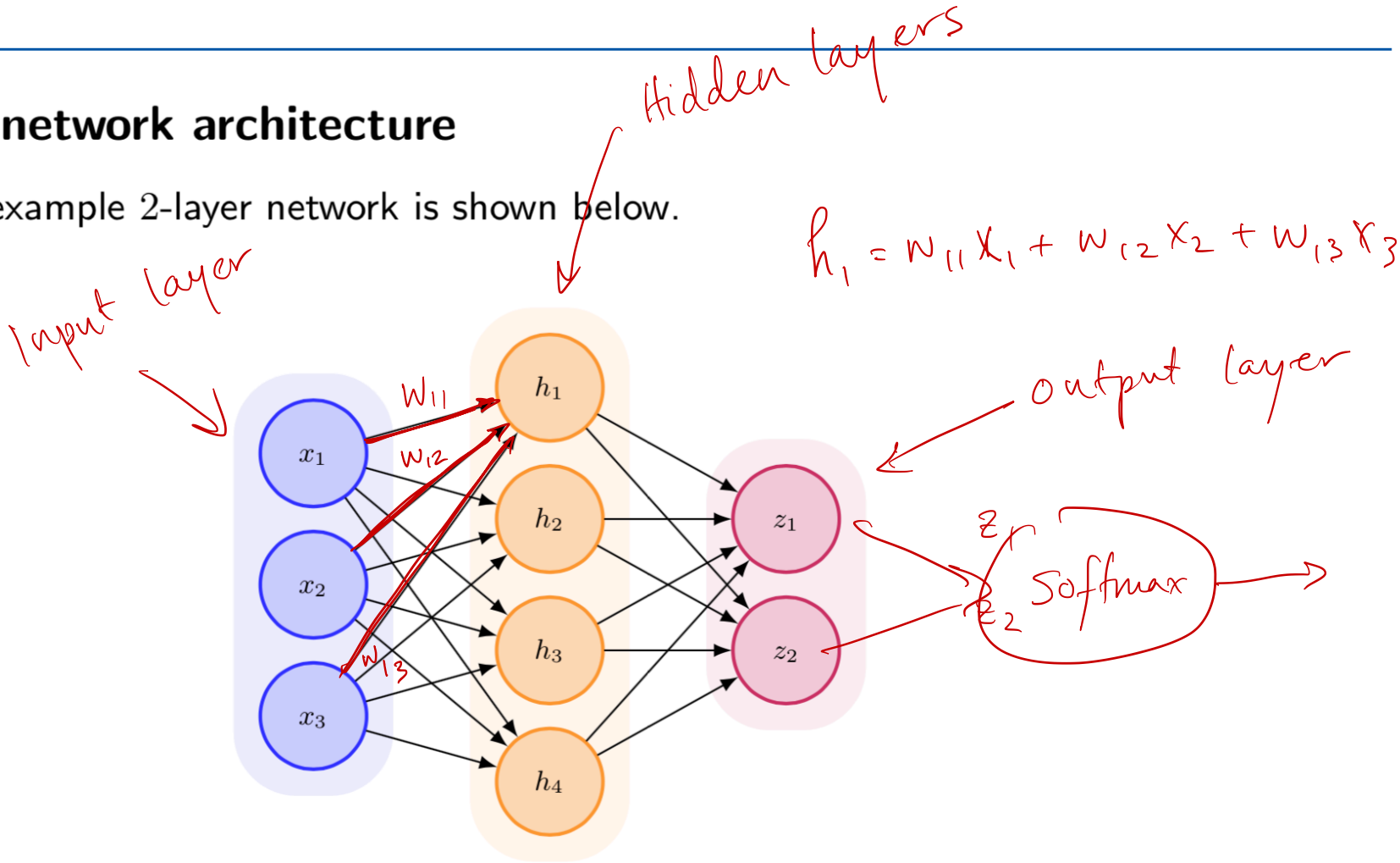
Caution when comparing to biology

These computing analogies are not precise, with large approximations. Limitations in the analogy include:

- Synaptic transmission is probabilistic, nonlinear, and dynamic.
- Dendritic integration is probabilistic and may be nonlinear.
- Dendritic computation has associated spatiotemporal decay.
- Integration is subject to biological constraints; for example, ion channels (which change the voltage of the cell) undergo refractory periods when they do not open until hyperpolarization.
- Different neurons may have different action potential thresholds depending on the density of sodium-gated ion channels.
- Feedforward and convolutional neural networks have no recurrent connections.
- Many different cell types.
- Neurons have specific dynamics that can be modulated by e.g., calcium concentration.
- And so many more...

Neural network architecture

An example 2-layer network is shown below.



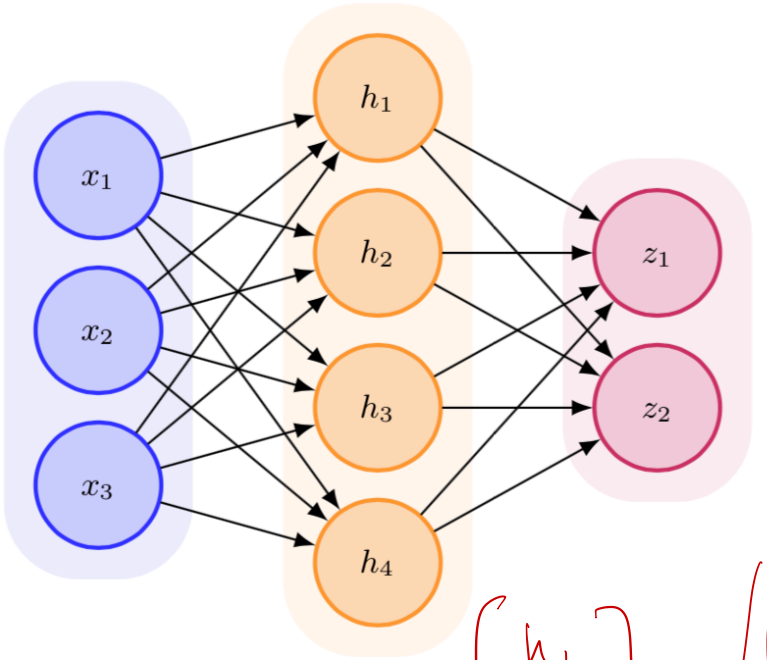
Here, the three dimensional inputs ($\mathbf{x} \in \mathbb{R}^3$) are processed into a four dimensional intermediate representation ($\mathbf{h} \in \mathbb{R}^4$), which are then transormed into the two dimensional outputs ($\mathbf{z} \in \mathbb{R}^2$).

Nomenclature

Some naming conventions.

- We call the first layer of a neural network the "input layer." We typically represent this with the variable $\mathbf{x}$.
- We call the last layer the "output layer." We typically represent this with the variable $\mathbf{z}$. (Note: why not $\mathbf{y}$ to match our prior nomenclature for the supervised outputs? Because the output of the network may be a processed version of $\mathbf{z}$, e.g., $\mathrm{softmax}(\mathbf{z})$.)
- We call the intermediate layers the "hidden layers." We typically represent this with the variable $\mathbf{h}$.
- When we specify that a network has N layers, this does not include the input layer.

Neural networks



$W_1 \in \mathbb{R}^{4\times 3}$

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = f\left(\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ \vdots & & \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \right)$$

First layer: $\mathbf{h} = f\left(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1\right)$

Second (output layer): $\mathbf{z} = f\left(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2\right)$

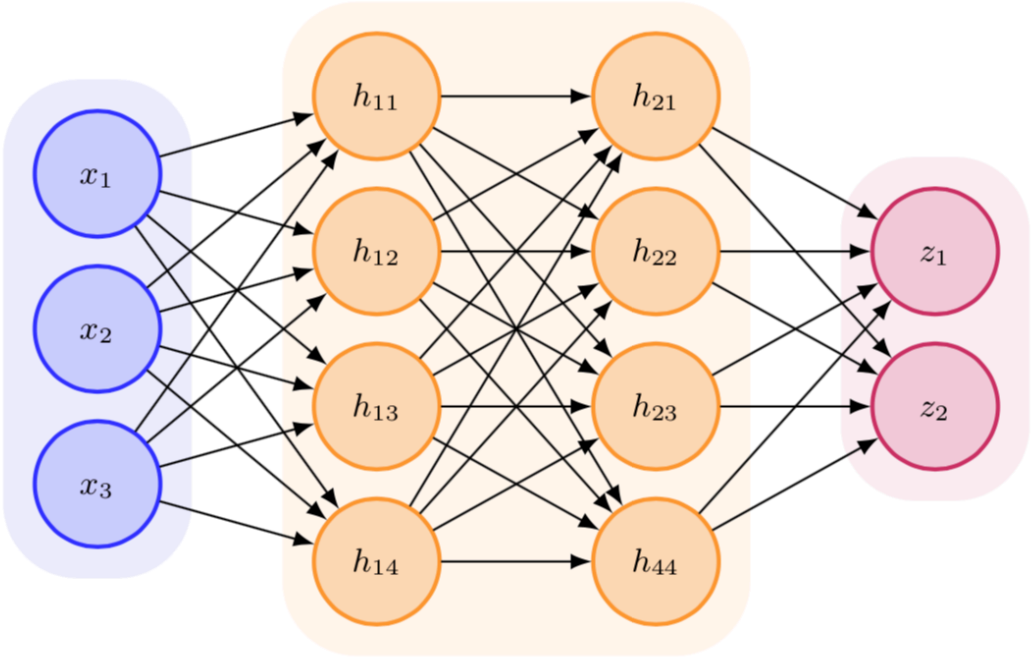
This network has 6 neurons (not counting the input). It has (3x4) + (4x2) = 20 weights, and 4+2 = 6 biases for a total of 26 learnable parameters.

Neural network architecture 2

Fully Connected Networks

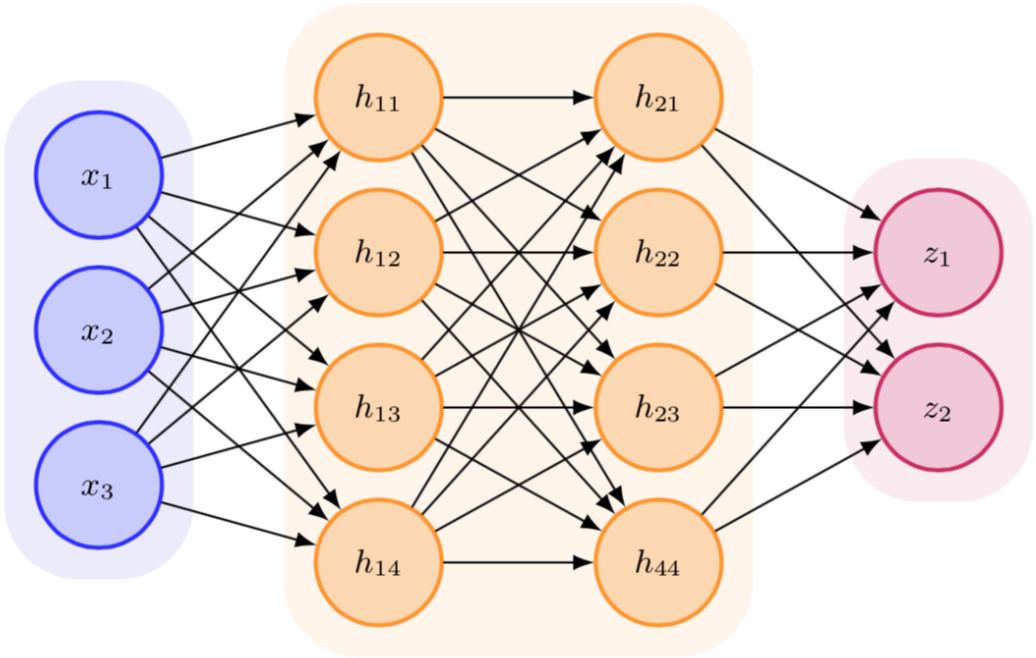
FC

An example 3-layer network is shown below.



Here, h_{ij} denotes the jth element of $\mathbf{h}_i$. There are many considerations in architecture design, which we will later discuss.

Neural networks

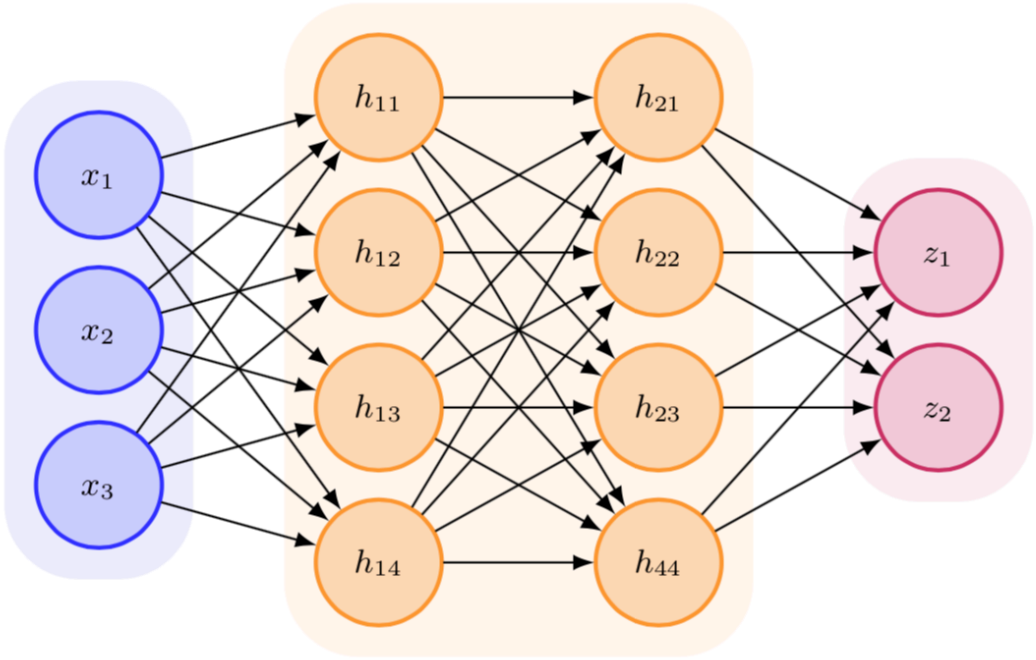


First layer: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

Second layer: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$

Third (output layer): $\mathbf{z} = f(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$

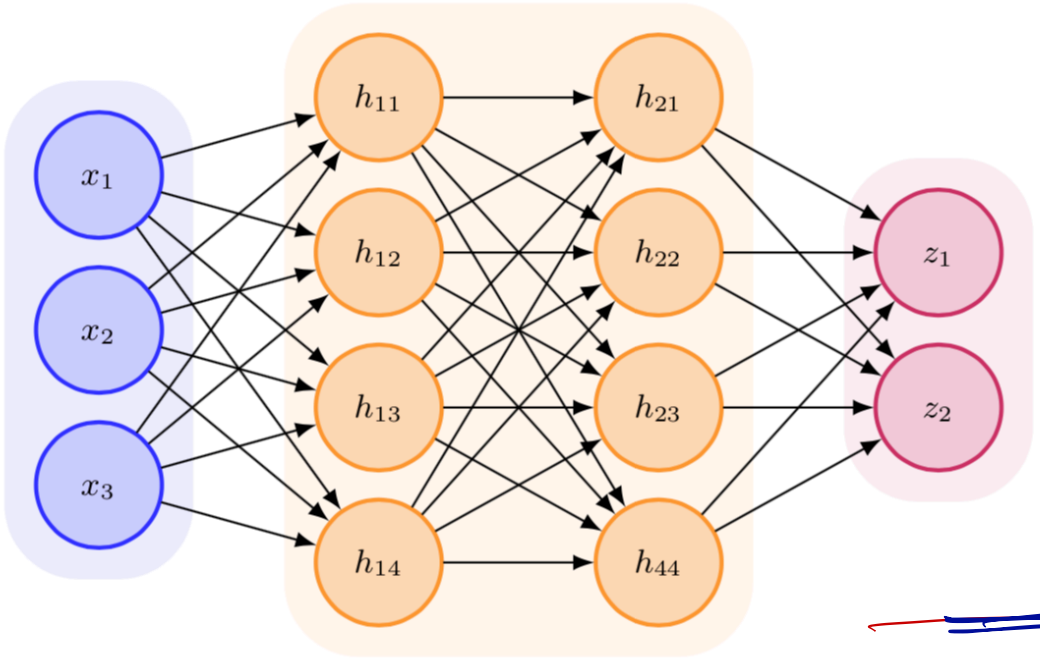
Neural networks

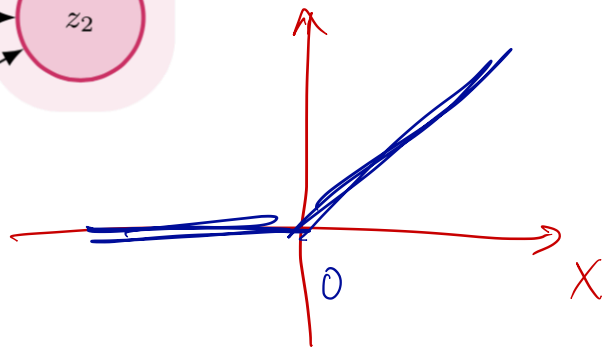


This network has 10 neurons (not counting the input). It has (3x4) + (4x4) + (4x2) = 36 weights, and 4+4+2= 10 biases for a total of 46 learnable parameters.

Perspective: Convolutional neural networks typically have on the order of hundreds of millions of parameters.

Neural networks

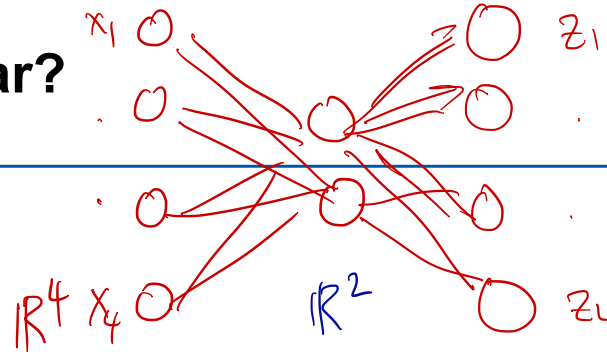





```
# Define the activation function
f = lambda x: x * (x > 0)
# Forward pass of a 3-layer network
h1 = f(np.dot(W1, x) + b1)
h2 = f(np.dot(W2, h1) + b2)
z = f(np.dot(W3, h2) + b3)
```

```
def f(x):
    return x * (x > 0)
```

What if f() is linear?



Neural networks

The above figure suggests the following equation for a neural network.

$\|z_i - x_i\|^2$

- Layer 1: $\mathbf{h}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$
- Layer 2: $\mathbf{h}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

$z = W_N h_{N-1} + b_N$

$= W_N (W_{N-1} h_{N-2} + b_{N-1}) + b_N$

Any composition of linear functions can be reduced to a single linear function. Here, $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where

$$\mathbf{W} = \mathbf{W}_N \cdots \mathbf{W}_2\mathbf{W}_1$$

and

$$\mathbf{b} = \mathbf{b}_N + \mathbf{W}_N\mathbf{b}_{N-1} + \cdots + \mathbf{W}_N \cdots \mathbf{W}_3\mathbf{b}_2 + \mathbf{W}_N \cdots \mathbf{W}_2\mathbf{b}_1$$

$h_1 = \begin{bmatrix} h_{11} \\ h_{12} \\ \vdots \\ h_{1N} \end{bmatrix}$

Introducing nonlinearity

To increase the network capacity, we can make it nonlinear. We do this by introducing a nonlinearity, $f(\cdot)$, at the output of each artificial neuron.

- Layer 1: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
- Layer 2: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

$f(h_1) = \begin{bmatrix} f(h_{11}) \\ f(h_{12}) \\ \vdots \\ f(h_{1N}) \end{bmatrix}$

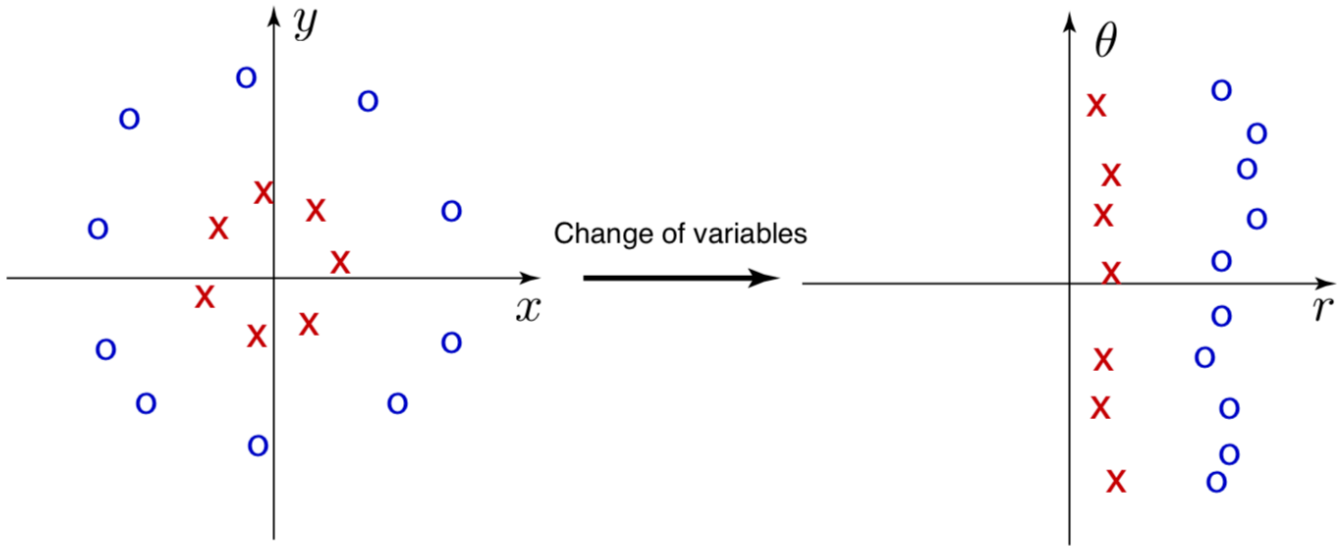
A few notes:

- These equations describe a *feedforward neural network*, also called a *multilayer perceptron*.
- $f(\cdot)$ is typically called an *activation function* and is applied elementwise on its input.
- The activation function does not typically act on the output layer, $\mathbf{z}$, as these are meant to be interpreted as scores. Instead, separate "output activations" are used to process $\mathbf{z}$. While these output activations may be the same as the activation function, they are typically different. For example, it may comprise a softmax or SVM classifier.

The hidden layers as learned features

A perspective on feature learning

One area of machine learning is very interested in finding *features* of the data that are then good for use as the input data to a classifier (like a SVM). Why might this be important?



The intermediate layers of the neural network (i.e., $\mathbf{h}_1, \mathbf{h}_2$, etc.) are features that the later layers then use for decoding. If the performance of the neural network is well, these features are good features.

Importantly, these features don't have to be handcrafted.

Example: XOR

Consider a system that produces training data that follows the $\mathrm{xor}(\cdot)$ function. The xor function accepts a 2-dimensional vector $\mathbf{x}$ with components x_1 and x_2 and returns 1 if $x_1 \neq x_2$. Concretely,

x_1	x_2	$\mathrm{xor}(\mathbf{x})$
0	0	0
0	1	1
1	0	1
1	1	0

$$J(\theta) = \frac{1}{2}\sum_{\mathbf{x}}(g(\mathbf{x}) - y(\mathbf{x}))^2$$

(Note, we wouldn't know $\mathrm{xor}(\mathbf{x})$, but we would have samples of corresponding inputs and outputs from training data. Hence, it may be better to simply replace $\mathrm{xor}(\mathbf{x})$ with $y(\mathbf{x})$ representing training examples.)

Example: XOR

Consider first a linear approximation of xor, via $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$. Then,

$$\begin{aligned}\frac{\partial J(\mathbf{w}, b)}{\partial \mathbf{w}} &= \sum_{\mathbf{x}} (\mathbf{w}^T\mathbf{x} + b - y(\mathbf{x}))\mathbf{x} \\ \frac{\partial J(\mathbf{w}, b)}{\partial b} &= \sum_{\mathbf{x}} (\mathbf{w}^T\mathbf{x} + b - y(\mathbf{x}))\end{aligned}$$

Equating these to 0, we arrive at:

$$(w_1 + b - 1)\begin{bmatrix} 1 \\ 0 \end{bmatrix} + (w_2 + b - 1)\begin{bmatrix} 0 \\ 1 \end{bmatrix} + (w_1 + w_2 + b)\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

These two equations can be simplified as:

$$\begin{aligned}(w_1 + b - 1) + (w_1 + w_2 + b) &= 0 \\ (w_2 + b - 1) + (w_1 + w_2 + b) &= 0\end{aligned}$$

These equations are symmetric, implying $w_1 = w_2 = w$. This means:

$$3w + 2b - 1 = 0 \implies b = \frac{1 - 3w}{2}$$

Example: XOR

Now let's consider using a two-layer neural network, with the following equation:

$$g(\mathbf{x}) = \mathbf{w}^T \max(0, \mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$

We haven't yet discussed how to optimize these parameters, but the point here is to show that by introducing a simple nonlinearity like $f(x) = \max(0, x)$, we can now solve the $\mathrm{xor}(\cdot)$ problem. Consider the solution:

$$\begin{aligned} \mathbf{W} &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ \mathbf{c} &= [0, -1]^T \\ \mathbf{w} &= [1, -2]^T \end{aligned}$$

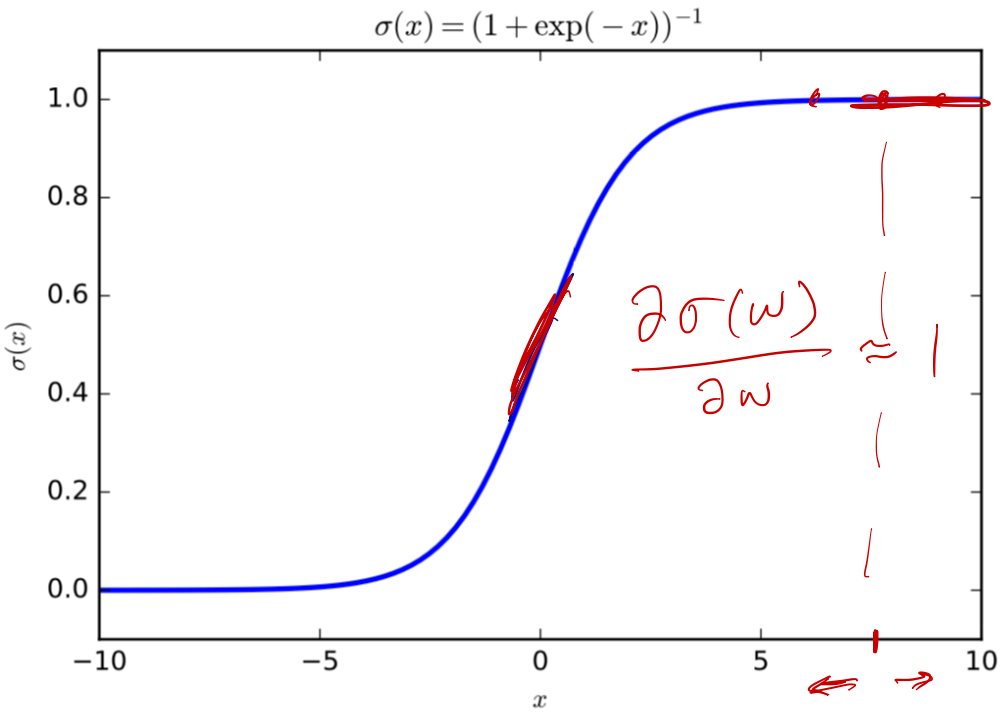
What nonlinearity to use?

There are a variety of activation functions. We'll discuss some more commonly encountered ones.

Sigmoid unit

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

```
f = lambda x: 1.0 / (1.0 + np.exp(-x))
```



$\frac{\partial \sigma(w)}{\partial w} \approx 0$

$\frac{\partial \sigma(w)}{\partial w} \approx 1$

Its derivative is:

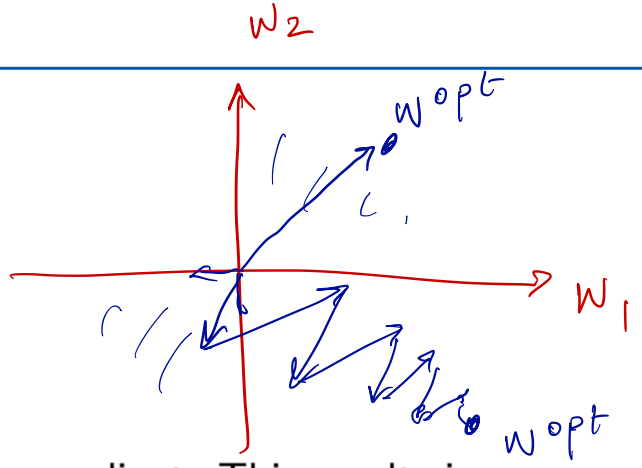
$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Sigmoid unit

"One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm." (Goodfellow et al., p. 173)

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \sigma(w)} \frac{\partial \sigma(w)}{\partial w}$$

Sigmoid unit



Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.

Cons:

- At extremes, the unit *saturates* and thus has zero gradient. This results in no learning with gradient descent.
- The sigmoid unit is not zero-centered; rather its outputs are centered at 0.5.

Consider $f(\mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$. Defining $z = \mathbf{w}^T\mathbf{x} + b$, the derivative with respect to $\mathbf{w}$, the parameters, is:

$$\frac{\partial f(\mathbf{w})}{\partial \mathbf{w}} = \sigma(z)(1 - \sigma(z))\mathbf{x}$$

$\frac{\partial \mathcal{L}}{\partial f} \frac{\partial f}{\partial w} = \frac{\partial \mathcal{L}}{\partial w}$

If $\mathbf{x} \geq 0$ (e.g., if the input units all had a sigmoidal output), then the gradient has all positive entries. Let's say we had some gradient, $\frac{\partial \mathcal{L}}{\partial f}$, which can be positive or negative. Then $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ will have all positive or negative entries.

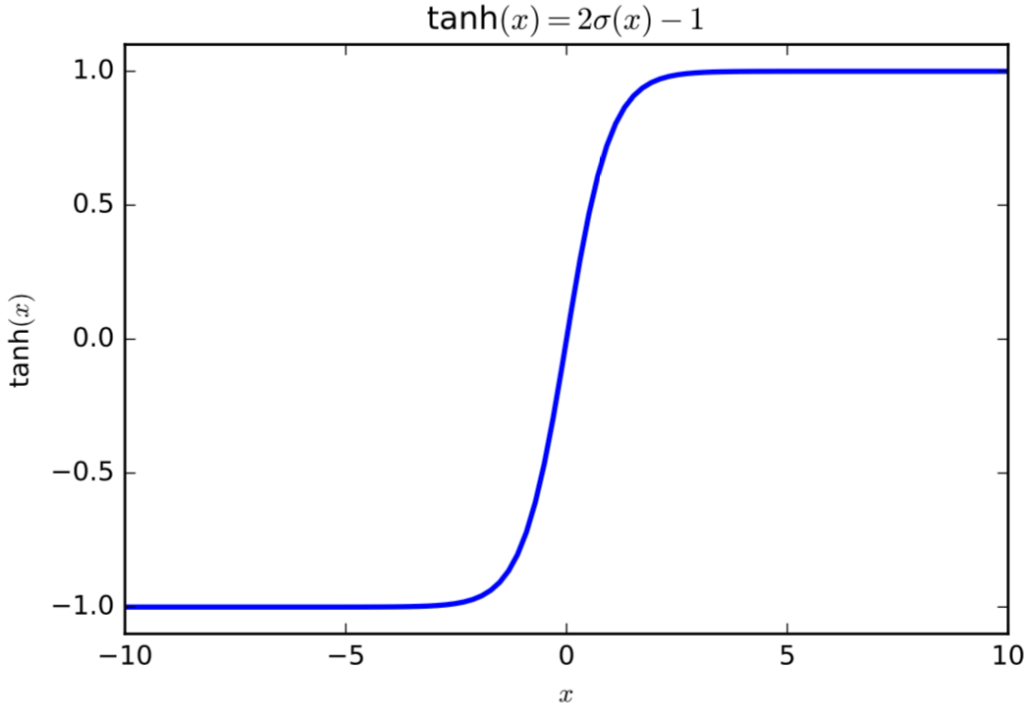
This can result in zig-zagging during gradient descent.

Hyperbolic tangent

Hyperbolic tangent, $\tanh(x) = 2\sigma(x) - 1$

The hyperbolic tangent is a zero-centered sigmoid-looking activation.

```
f = lambda x: np.tanh(x)
```



Its derivative is:

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