Lecture 4: Softmax, gradient descent, and neural networks

Announcements:

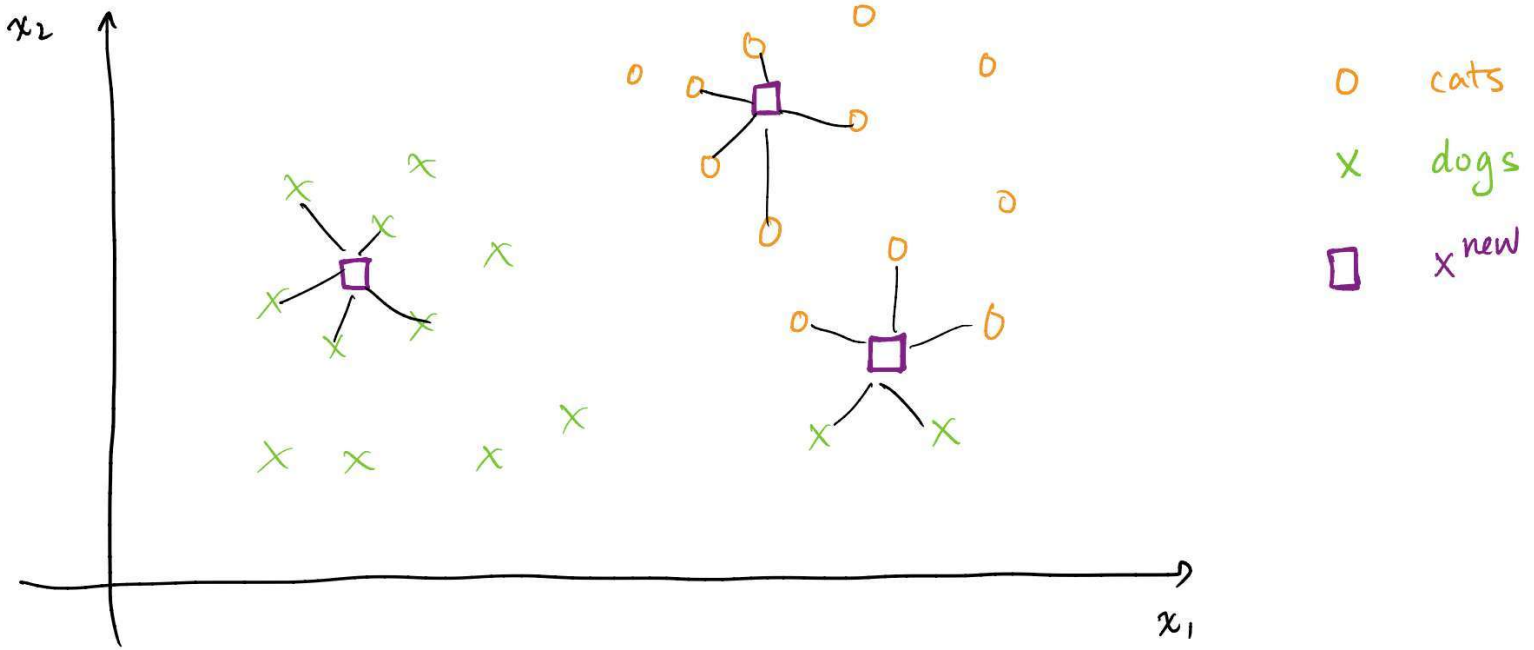
- HW #1 is due Monday Jan 20, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. The graders will grade your work from these notebooks. In the future, you will also have to print out code from any .py files that were modified.

k-nearest neighbors

$k = 5$

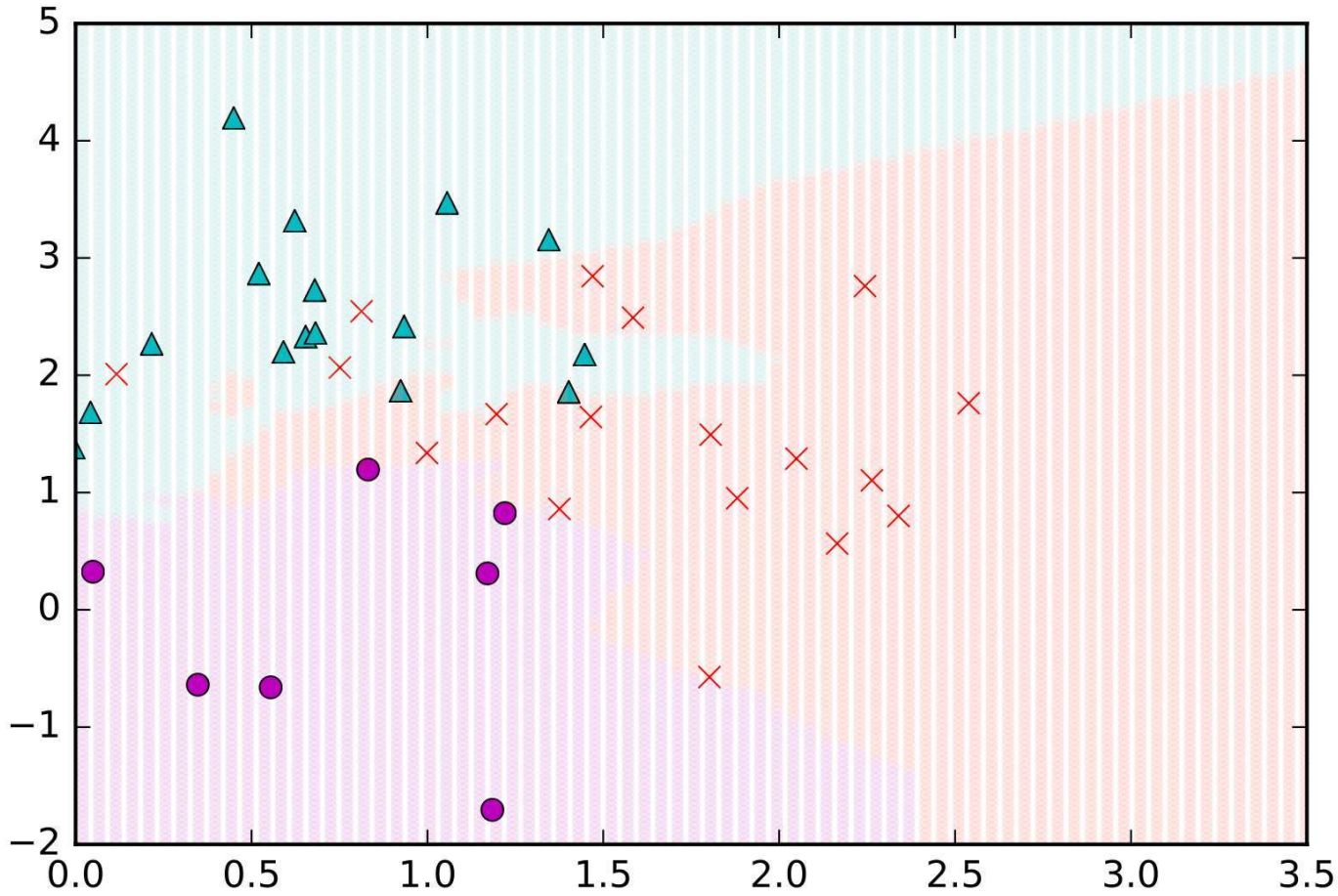
Intuitively, k-nearest neighbors says to find the k closest points (or nearest neighbors) in the training set, according to an appropriate metric. Each of its k nearest neighbors then vote according to what class it is in, and $\mathbf{x}^{\text{new}}$ is assigned to be the class with the most votes.

$x^{(i)} \in \mathbb{R}^2$



k-nearest neighbors

What a solution looks like for k=3 neighbors:



k-nearest neighbors

Why might k-nearest neighbors not be a good idea for image classification?

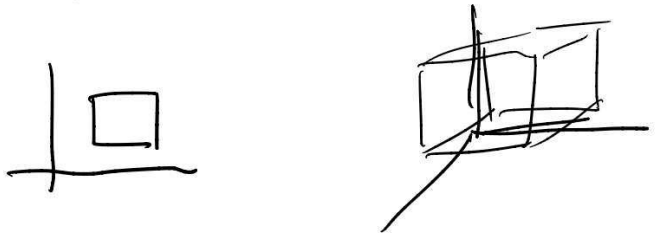
Euclidean distance is not a good metric for semantic similarity of images.

(all 3 images have same L2 distance to the one on the left)

Credit: CS231n, Stanford University

k-nearest neighbors

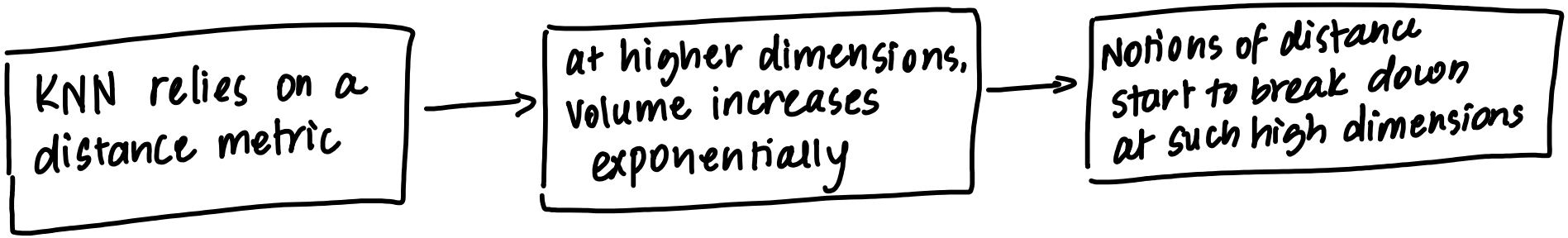
Why might k-nearest neighbors not be a good idea for image classification?



Curse of dimensionality:

- Images are very high-dimensional vectors, e.g., each CIFAR-10 image is a 3072 dimensional vector (and these are small images).

- Notions of "distance" become less intuitive in higher dimensions.
  - Distances in some dimensions matter more than others.
  - In higher-dimensional space, the volume increases exponentially.
  - This leaves a lot of empty space — and so the nearest neighbors may not be so near.

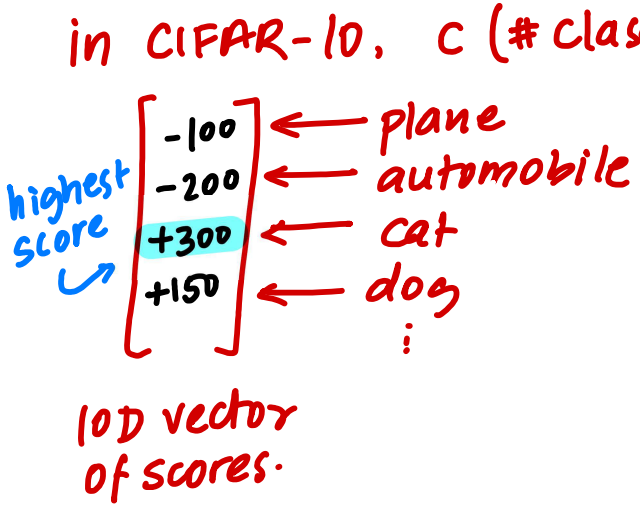


Classifiers based on linear classification

Perhaps a better way would be to develop a "score" for an image coming from each class, and then pick the class that achieves the highest score.

Linear classifiers are a major building block for neural networks. In particular, each layer of a neural network is composed of a linear classifier, followed by a nonlinear function.

The softmax classifier is the most common classifier at the end of a neural network.



Classifiers based on linear classification

Example 2: Consider a matrix, $\mathbf{W}$, defined as:

Each row of W captures what the "average" item looks like for that class →

Row vector

$$\begin{bmatrix} - \mathbf{w}_1^T - \\ \vdots \\ - \mathbf{w}_c^T - \end{bmatrix} \qquad i \in \{1, 2, \ldots, 10\}$$

Then, $\mathbf{W} \in \mathbb{R}^{c \times N}$. Let $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{b}$ is a vector of bias terms. Then $\mathbf{y} \in \mathbb{R}^c$ is a vector of scores, with its ith element corresponding to the score of $\mathbf{x}$ being in class i. The chosen class corresponds to the index of the highest score in $\mathbf{y}$.

$$y = \begin{bmatrix} w_1^T x + b_1 \\ w_2^T x + b_2 \\ \vdots \\ w_{10}^T x + b_{10} \end{bmatrix} = \begin{bmatrix} — w_1^T — \\ — w_2^T — \\ \vdots \\ — w_{10}^T — \end{bmatrix} \begin{bmatrix} x \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{10} \end{bmatrix}$$

W (10 × 3072) · x (3072 × 1) + b (10 × 1)

Score of image x belonging to class 2

CIFAR-10 (width × height × 3)

$x \in \mathbb{R}^{3072}$

$y \in \mathbb{R}^{10}$

$W \in \mathbb{R}^{10 \times 3072}$

$b \in \mathbb{R}^{10}$ → bias vector.

$(c = 10)$ → #classes

summary →

$$x^{(i)} \xrightarrow{Wx^{(i)} + b} \hat{y}^{(i)} = \begin{bmatrix} 300 \\ 500 \\ -150 \\ \vdots \end{bmatrix}$$

The ith image, $x^{(i)}$, belongs to class (2) (automobile)

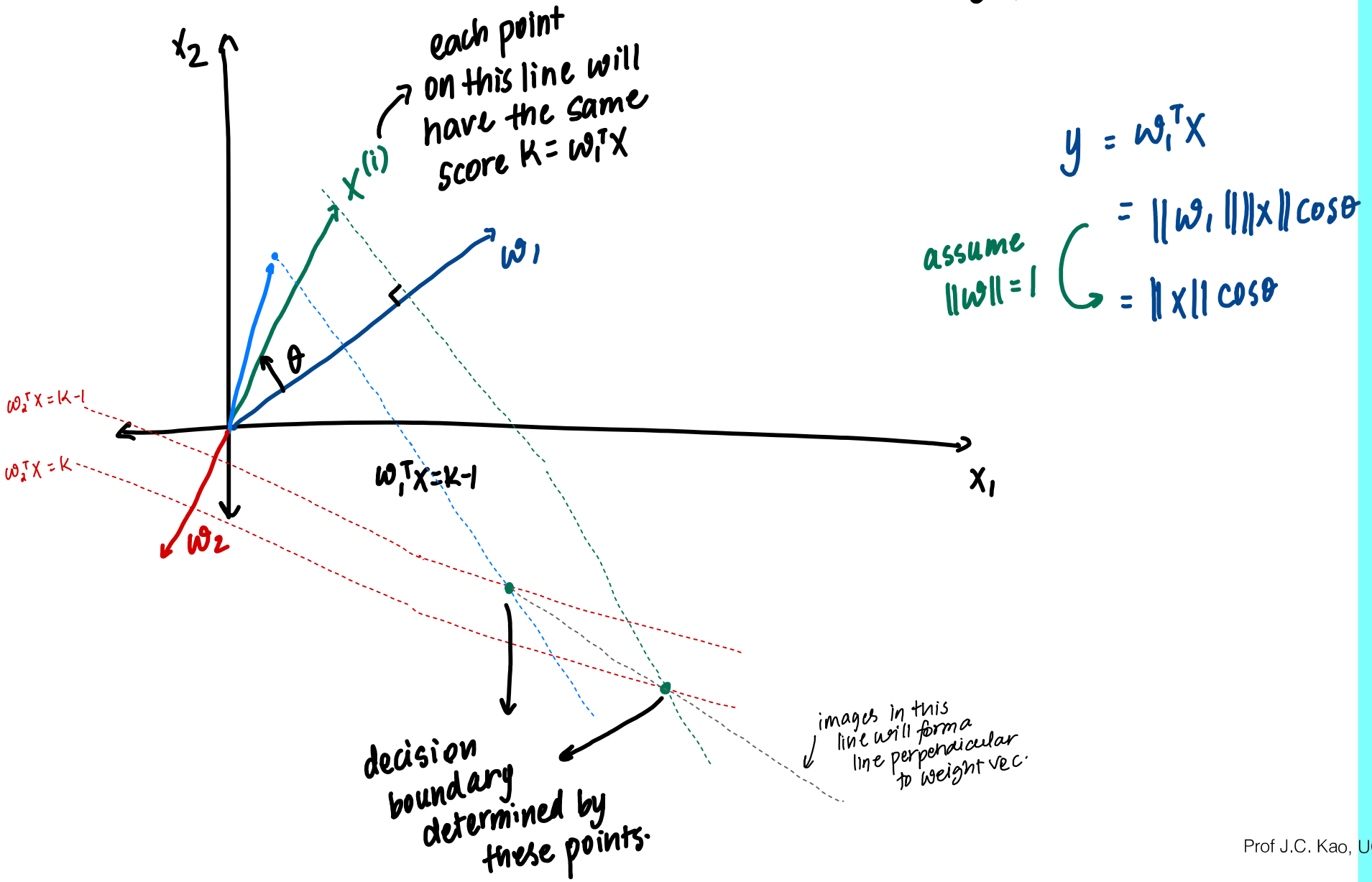
Classifiers based on linear classification

Each row of **W** can be thought of as a template.

Classifiers based on linear classification

What is a linear classifier doing?

$x \in \mathbb{R}^2 \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



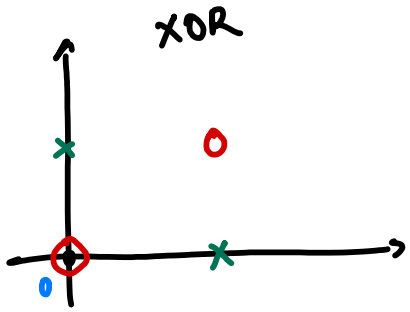
$y = w_1^T x$

$= \|w_1\| \|x\| \cos\theta$

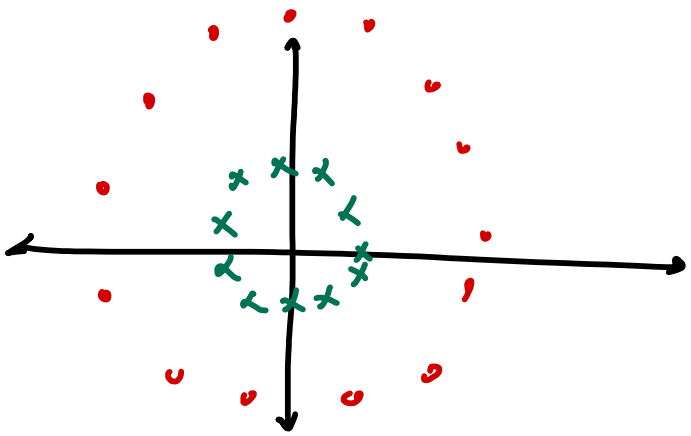
assume $\|w\| = 1$ $\quad = \|x\| \cos\theta$

Classifiers based on linear classification

Where might linear classifiers fail?

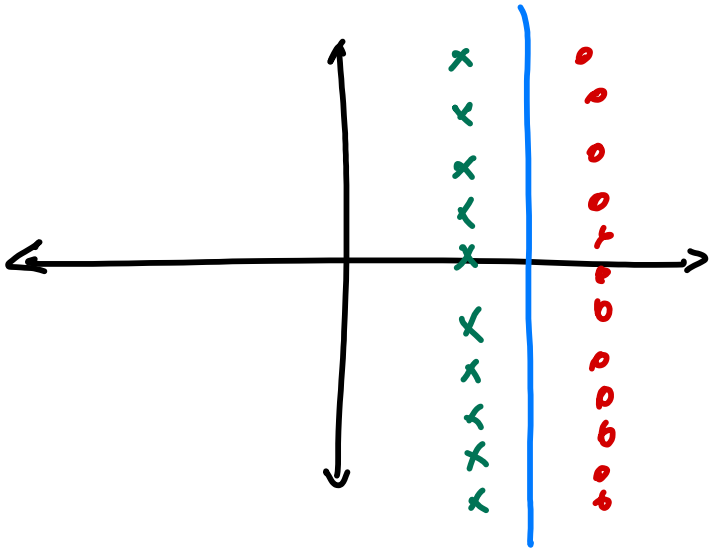


can't draw a straight line to separate two datapoints.



Generally, image data is not linearly separable. But by performing change of variables, the data could become linearly separable.

⇓ Change of variables to polar coordinates.



Classifiers based on linear classification

Next, how do we take the scores we receive (which are analog in value) and turn them into an appropriate **loss function** for us to optimize, so we can learn **W** and **b** appropriately?

Other types of optimization

We've talked about examples where we want to *minimize* a mean-square error or distance metric.

Another metric that we may want to maximize is the *probability of having observed the data*. In this framework, the data is modeled to have some distribution with parameters. We choose the parameters to maximize the probability of having observed our training data.

DATA: H T H H T T H T → Want to come up with a mathematically rigorous way of finding P(heads) given this data

MODEL: $x^{(i)} = \begin{cases} H & \text{w.p. } \theta \\ T & \text{w.p. } 1-\theta \end{cases}$

LIKELIHOOD:

model 1: $\theta = 1$

model 2: $\theta = 0.75$

model 3: $\theta = 0.5$

$1 \cdot 0 \cdot 1 \cdot 1 \cdot 0 \cdot 0 \cdot 1 \cdot 0 = 0$ ← probability of getting the data sequence given the model.

$(0.75)(0.25)(0.75)^2(0.25)^2(0.75)(0.25) = 0.00124$

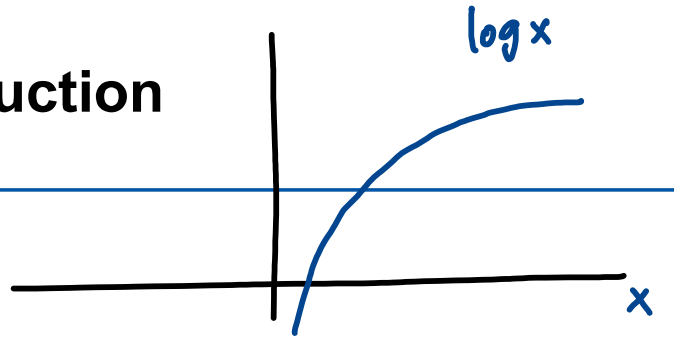
$(0.5)^4(0.5)^4 = 0.0039$

} Between models 2 and 3, model 3 is more likely.

↓

want to find θ that makes this likelihood as high as possible

Maximum-likelihood introduction



ℓ : likelihood $= \theta^4(1-\theta)^4$

$\frac{\partial \text{likelihood}}{\partial \theta} = 0$ and solve for θ.

$\Longrightarrow$ in practice $\frac{\partial \log(\text{likelihood})}{\partial \theta} = 0$

$\boxed{\theta = \frac{1}{2}}$

Chain rule for probability

Notation:

$P(A=a) = P_A(a) = p(a)$

$P(B=a) = P_B(b) = p(b)$

$P(A=a, B=b) = P_{A,B}(a,b) = p(a,b)$

$\Pr(A=a, B=b) = P(A=a)\,P(B=b \text{ given } A=a)$

$p(a,b) = p(a)p(b|a)$

$= p(b)p(a|b)$

$p(a,b,c) = p(c)\cdot p(a|c)\cdot p(b|a,c)$

$= p(a)\cdot p(b|a)\cdot p(c|a,b)$

$= p(a,c)\cdot p(b|a,c)$

Chain rule for probability

$$P(b,c \mid d,e) = \frac{?}{P(d) \cdot P(e \mid d)}$$

$$? = P(b,c \mid d,e) \cdot P(d) \cdot P(e \mid d)$$

$$? = P(b,c,d,e)$$

$$P(d,e) \cdot ? = \frac{P(a,b,c,d,e)}{P(a \mid b,c,d,e)}$$

$$P(a,b,c,d,e) = P(d,e) \cdot P(a \mid b,c,d,e) \cdot ?$$

$$= P(d,e) \cdot P(b,c \mid d,e) \cdot P(a \mid b,c,d,e)$$

$$P(b,c \mid d,e) = ?$$

Turn the scores into a probability

A first thought is to turn the scores into probabilities.

Softmax function

There are several instances when the scores should be normalized. This occurs, for example, in instances where the scores should be interpreted as probabilities. In this scenario, it is appropriate to apply the *softmax* function to the scores.

The softmax function transforms the class score, $\mathrm{softmax}_i(\mathbf{x})$, so that:

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

$a_i(x) = y_i = w_i^T x + b_i$

Note: $\sum_c \mathrm{softmax}_c(x) = 1$

and $0 \le \mathrm{softmax}_i(x) \le 1$

for $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + b_i$ and c being the number of classes.

Two classes: $i=1$, $i=2$

$a_1(x) = w_1^T x + b_1$

$a_2(x) = w_2^T x + b_2$

$$\mathrm{Softmax}_1(x) = \frac{e^{w_1^T x + b_1}}{e^{w_1^T x + b_1} + e^{w_2^T x + b_2}}$$

$$\mathrm{Softmax}_2(x) = \frac{e^{w_2^T x + b_2}}{e^{w_1^T x + b_1} + e^{w_2^T x + b_2}}$$

Softmax classifier

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

for $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + b_i$ and c being the number of classes.

If we let $\theta = \{\mathbf{w}_j, b_j\}_{j=1,\ldots,c}$, then $\text{softmax}_i(\mathbf{x})$ can be interpreted as the probability that $\mathbf{x}$ belongs to class i. That is,

$$\Pr(y^{(j)} = i | \mathbf{x}^{(j)}, \theta) = \text{softmax}_i(\mathbf{x}^{(j)})$$

$\mathbf{x}^{(j)}$: jth image (Example j)

$\Pr(y^{(j)} = i | \mathbf{x}^{(j)}, \theta)$: Probability that image $x^{(j)}$ belongs to class (i)

$P(x^{(1)}, y^{(1)}, x^{(2)}, y^{(2)} | \theta)$

$= P(x^{(1)}, y^{(1)} | \theta) \cdot P(x^{(2)}, y^{(2)} | \theta, x^{(1)}, y^{(1)})$

Assumption: conditional indep.

$= P(x^{(1)}, y^{(1)} | \theta) \cdot P(x^{(2)}, y^{(2)} | \theta)$

Softmax classifier

Although we know the softmax function, how do we specify the *objective* to be optimized with respect to θ?

One intuitive heuristic is that we should choose the parameters, θ, so as to maximize the likelihood of having seen the data. Assuming the samples, $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$ are iid, this corresponds to maximizing:

image dog image cat

w/b

$$p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, y^{(1)}, \ldots, y^{(m)} | \theta) = \prod_{i=1}^{m} p(\mathbf{x}^{(i)}, y^{(i)} | \theta)$$

$$= \prod_{i=1}^{m} p(\mathbf{x}^{(i)} | \theta) p(y^{(i)} | \mathbf{x}^{(i)}, \theta)$$

Softmax

Softmax classifier

$p(x^{(i)}|\theta) = P(x^{(i)})$
↳ Because $x^{(i)}$ is indep. of θ

Why can we remove this?

#classes

$$\arg\max_{\theta} \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta) p(y^{(i)}|\mathbf{x}^{(i)}, \theta) = \arg\max_{\theta} \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}, \theta)$$

θ: W, b

$$= \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \log\left(\text{softmax}_{y^{(i)}}(x^{(i)})\right)$$

$$= \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \log\left[\frac{e^{w_{y^{(i)}}^T x^{(i)} + b_{y^{(i)}}}}{\sum_{j=1}^{c} e^{w_j^T x^{(i)} + b_j}}\right]$$

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

$a_{y^{(i)}}(x^{(i)})$: Score of $x^{(i)}$ belonging to the class that came with the data ($y^{(i)}$)

$\frac{1}{m}$: scaling factor

$$= \underset{\theta}{\text{argmin}} \frac{1}{m} \sum_{i=1}^{m} \left[\log\left(\sum_{j=1}^{c} e^{a_j(x^{(i)})}\right) - a_{y^{(i)}}(x^{(i)})\right]$$

LOSS FUNCTION (cross entropy loss)

i^{th} example: $x^{(i)}$, $y^{(i)} = 2$

$(i+1)^{th}$ example: $x^{(i+1)}$, $y^{(i+1)} = 4$

Notation:

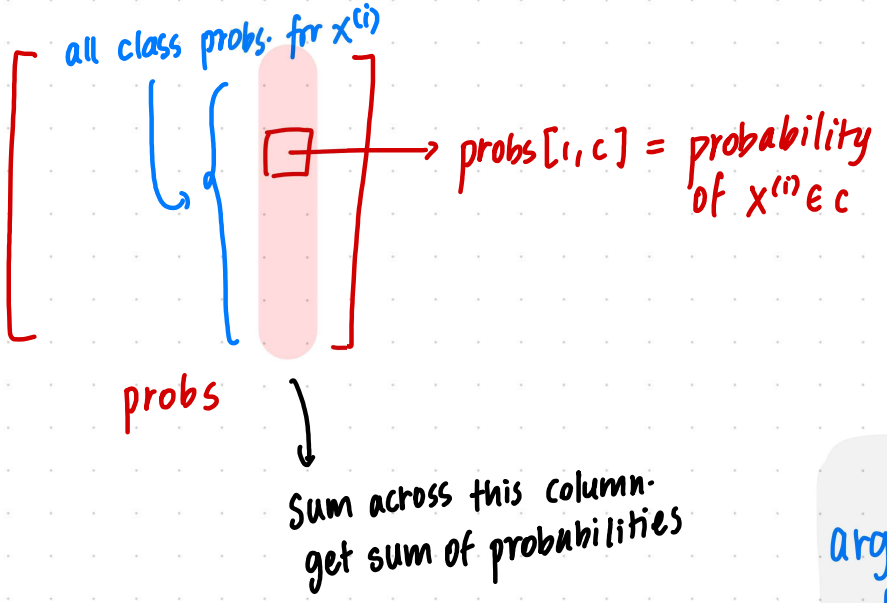
$$a_j(x^{(i)}) = w_j^T x^{(i)} + b_j$$

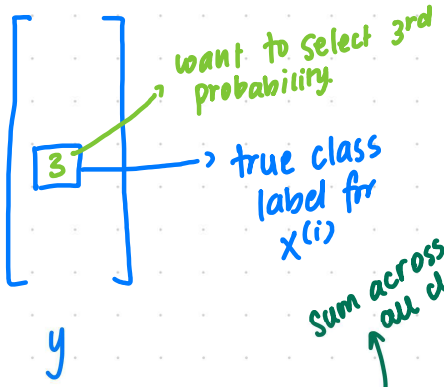
$$\underset{\theta}{\text{argmax}}\, l = \underset{\theta}{\text{argmax}} \log(l)$$

$$\log\left(\prod_{i=1}^{m} P(y^{(i)}|x^{(i)}, \theta)\right) = \sum_{i=1}^{m} \log\left[P(y^{(i)}|x^{(i)}, \theta)\right]$$

$$\underset{\theta}{\text{argmax}}\, f(\theta) = \underset{\theta}{\text{argmin}} - f(\theta)$$

ROUGH WORK - HW 3





$$a_j(x^{(i)}) = w_j^T x^{(i)} + b_j \longrightarrow \text{Scores.}$$

$$a = w^T x^{(i)} + b \longrightarrow \text{scores.}$$

$$\underset{\theta}{\text{argmin}} \; \frac{1}{m} \sum_{i=1}^{m} \left[\log \left(\sum_{j=1}^{c} e^{a_j(x^{(i)})} \right) - a_y^{(i)} x^{(i)} \right]$$

- Sum across all classes.
- Score for sample $x^{(i)}$
- average across samples
- log of sum
- e^{scores}
- difference
- score of true label for sample $x^{(i)}$

- When score for correct probability is highest, then diff will be very small.

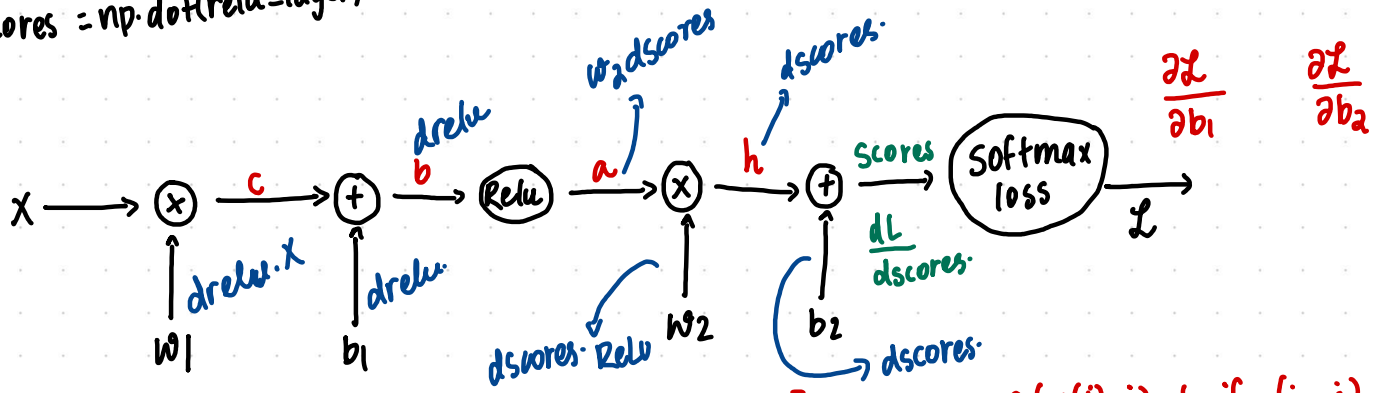
Eqns

```
layer_1 = X @ W1.T + b1
relu = np.maximum(layer1, 0)
scores = np.dot(relu_layer, W2.T) + b2
```

$X \in \mathbb{R}^{N \times D}$
$W_1 \in \mathbb{R}^{H,D}$
$b_1 \in \mathbb{R}^{H}$
$W_2 \in \mathbb{R}^{C,H}$
$b_2 \in \mathbb{R}^{C}$

Want to find

$\frac{\partial L}{\partial W_1}$ $\frac{\partial L}{\partial W_2}$ $\frac{\partial L}{\partial b_1}$ $\frac{\partial L}{\partial b_2}$



$$\frac{\partial L}{\partial W_2} = \sum_{j=1}^{m} \left[\delta(y^{(j)}, i)\, x^{(j)} - \text{softmax}_i(x^{(j)}) x^{(j)} \right]$$ where $\delta(y^{(j)}, i) = 1$ if $(i = j)$

$$\frac{\partial L}{\partial b_2} = \sum_{j=1}^{m} \left[\delta(y^{(j)}, i) - \text{softmax}_i(x^{(j)}) \right]$$ where $\delta(y^{(j)}, i) = 1$ if $(i = j)$

} computed in HW 2

$$\frac{\partial L}{\partial W_2} = \frac{\partial h}{\partial W_2} \cdot \frac{\partial L}{\partial h}$$

$\frac{\partial h}{\partial W_2}$ → ReLU. dscores

probability that x belongs to class i

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

Softmax classifier

Now we have our softmax loss function.

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

Note, we haven't figured out yet how to get the optimal parameters.

(That'll be later.)

cross entropy loss: tries to maximise likelihood of data.

How do we choose $\{w, b\}$ such that loss is as small as possible?

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2

Score check: $-a_{y^{(i)}}(\mathbf{x}^{(i)}) + \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x}^{(i)}))$

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2

Cat: -2.1 + log(exp(2.1) + exp(3.4) + exp(-2.0)) = 1.54

Softmax loss for cat.

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	1.54		

Car: -5.1 + log(exp(0.2) + exp(5.1) + exp(1.7)) = 0.04

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	1.54	0.04	

Bird: 1.2 + log(exp(2.3) + exp(3.1) + exp(-1.2)) = 4.68

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	1.54	0.04	4.68

$$-a_{y^{(i)}}(\mathbf{x}^{(i)}) + \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x}^{(i)}))$$

Softmax classifier

$e^{100} \gg e^{20} \gg e^{-1}$

$e^{100} + e^{20} + e^{-1} \approx e^{100}$

A few additional notes on the softmax classifier:

Softmax classifier: intuition

When optimizing likelihoods, we typically work with the "log likelihood." When applying the softmax, we interpret its output as the probability of a class.

$$\begin{aligned} \log \Pr(y = y^{(i)} | \mathbf{x}) &= \log \text{softmax}_i(\mathbf{x}) \\ &= a_{y^{(i)}}(\mathbf{x}) - \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x})) \end{aligned}$$

3 classes

Score: 100 (e^{100}), 20 (e^{20}), -1 (e^{-1})

When maximizing this, the term $a_{y^{(i)}}(\mathbf{x})$ is made larger, and the term $\log \sum_j \exp(a_j(\mathbf{x}))$ is made smaller. The latter term can be approximated by $\max_j a_j(\mathbf{x})$. (Why?)

We consider two scenarios:

- If $a_{y^{(i)}}(\mathbf{x})$ produces the largest score, then the log likelihood is approximately 0.
- If $a_j(\mathbf{x})$ produces the largest score for $j \neq y^{(i)}$ then $a_{y^{(i)}}(\mathbf{x}) - a_j(\mathbf{x})$ is negative, and thus the log likelihood is negative.

Softmax classifier

A few additional notes on the softmax classifier:

Overflow of softmax

A potential problem when implementing a softmax classifier is overflow.

- If $a_i(\mathbf{x}) \gg 0$, then $e^{a_i(\mathbf{x})}$ may be very large, and numerically overflow and / or result to numerical impression.
- Thus, it is standard practice to normalize the softmax function as follows:

score vector

$$\begin{bmatrix} 1000 \\ 1200 \\ 1150 \end{bmatrix} \xrightarrow{-1200} \begin{bmatrix} -200 \\ 0 \\ -50 \end{bmatrix}$$

$$\begin{aligned} \text{softmax}_i(\mathbf{x}) &= \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}} \\ &= \frac{ke^{a_i(\mathbf{x})}}{k\sum_{j=1}^{c} e^{a_j(\mathbf{x})}} \\ &= \frac{e^{a_i(\mathbf{x})+\log k}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})+\log k}} \end{aligned}$$

- A sensible choice of k is so that $\log k = -\max_i a_i(\mathbf{x})$, making the maximal argument of the exponent 0.

Lecture 5: Gradient Descent, Neural Networks, Backpropagation

Announcements:

- HW #2 is due Monday Jan 27, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

Classifiers based on linear classification

Example 2: Consider a matrix, $\mathbf{W}$, defined as:

$$\begin{bmatrix} - \mathbf{w}_1^T - \\ \vdots \\ - \mathbf{w}_c^T - \end{bmatrix}$$

$i \in \{1, 2, \ldots, 10\}$

Then, $\mathbf{W} \in \mathbb{R}^{c \times N}$. Let $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{b}$ is a vector of bias terms. Then $\mathbf{y} \in \mathbb{R}^c$ is a vector of scores, with its ith element corresponding to the score of $\mathbf{x}$ being in class i. The chosen class corresponds to the index of the highest score in $\mathbf{y}$.

$$y = \begin{bmatrix} w_1^T x + b_1 \\ w_2^T x + b_2 \\ \vdots \\ w_{10}^T x + b_{10} \end{bmatrix}_{10 \times 1} = \underbrace{\begin{bmatrix} — w_1^T — \\ — w_2^T — \\ \vdots \\ — w_{10}^T — \end{bmatrix}}_{W,\ 10 \times 3072} \begin{bmatrix} x \\ \end{bmatrix}_{3072 \times 1} + \begin{bmatrix} b \\ \end{bmatrix}_{10 \times 1}$$

Score of image x belonging to class 2.

CIFAR-10

$x \in \mathbb{R}^{3072}$

$y \in \mathbb{R}^{10}$ $\quad c = 10$

$W \in \mathbb{R}^{10 \times 3072}$

$b \in \mathbb{R}^{10}$

$x^{(i)} \xrightarrow{Wx^{(i)} + b} \hat{y}^{(i)} = \begin{bmatrix} 300 \\ 500 \\ -150 \\ \vdots \\ -23 \end{bmatrix}$

The ith image, $x^{(i)}$, belongs to class 2.

Softmax classifier

$\arg\max_\theta \ell = \arg\max_\theta \log(\ell)$

$\log\left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)\right) = \sum_{i=1}^{m} \log\left(p(y^{(i)}|x^{(i)},\theta)\right)$

$p(x^{(i)}|\theta) = p(x^{(i)})$ (w, b)

$\log(a \cdot b) = \log a + \log b$

$$\arg\max_\theta \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta) p(y^{(i)}|\mathbf{x}^{(i)},\theta) = \arg\max_\theta \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)},\theta)$$

(w, b)

$$= \arg\max_\theta \sum_{i=1}^{m} \log \text{softmax}_{y^{(i)}}(x^{(i)})$$

$$= \arg\max_\theta \sum_{i=1}^{m} \log \left[\frac{e^{w_{y^{(i)}}^T x^{(i)} + b_{y^{(i)}}}}{\sum_{j=1}^{c} e^{w_j^T x^{(i)} + b_j}} \right]$$

i^{th} ex: $x^{(i)}$, $y^{(i)} = 2$

$i+1^{th}$ ex: $x^{(i+1)}$, $y^{(i+1)} = 4$

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

$a_j(x^{(i)}) = w_j^T x^{(i)} + b_j$

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

LOSS FUNCTION

$\arg\max_\theta f(\theta) = \arg\min_\theta -f(\theta)$

Softmax loss function

Softmax:

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

Parameters?

$\theta = \{ w, b \}$

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

To do so, we use the technique of 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 $\mathbf{x} \in \mathbb{R}^n$,

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

- 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:

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

- The directional derivative tells us the slope of f in the direction $\mathbf{u}$.

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:

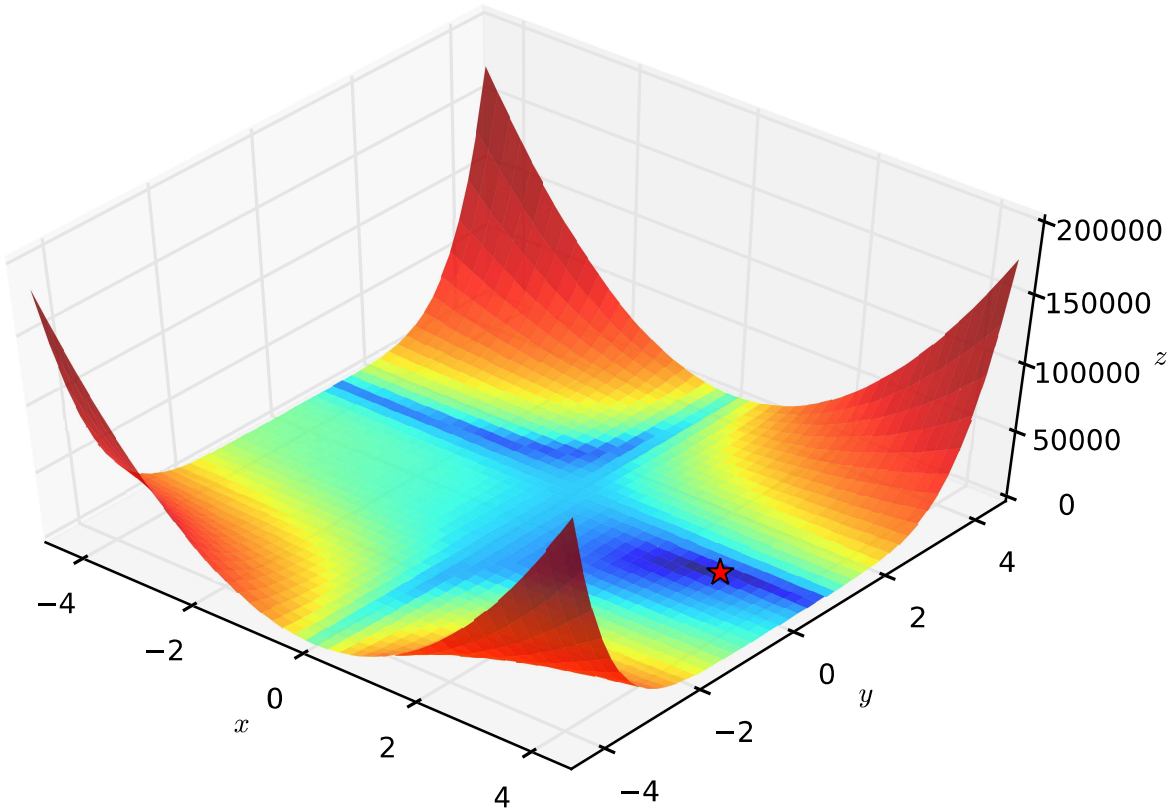
$$\mathbf{x} := \mathbf{x} - \epsilon \nabla_x f(\mathbf{x})$$

- 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:

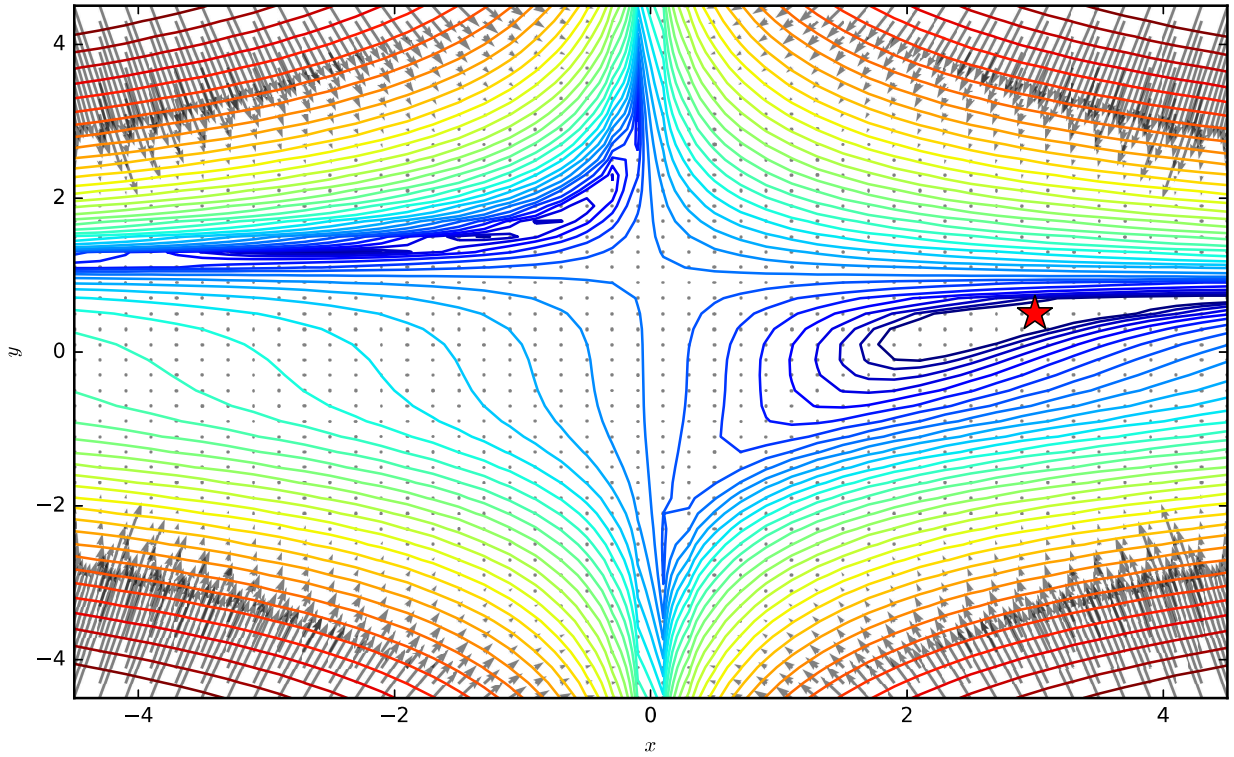
Animations thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/



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

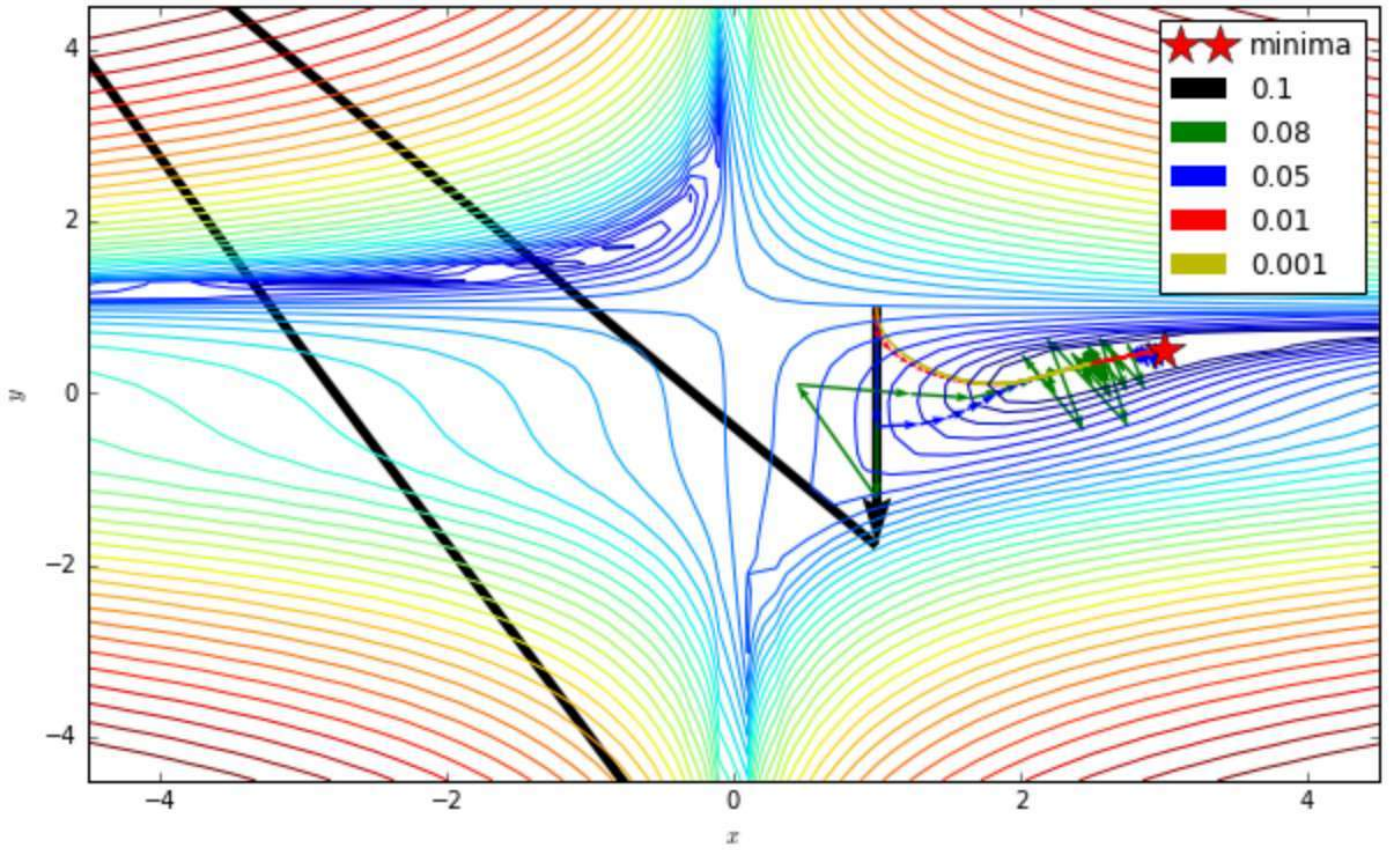
```
def gd(func, x0, eps=1e-4, tol=1e-3):
    last_diff = np.Inf
    x = x0
    path = [np.copy(x0)]
    costs = [func(x0)[0]]
    grads = []
    i = 1
    hit_max = False

    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
```

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

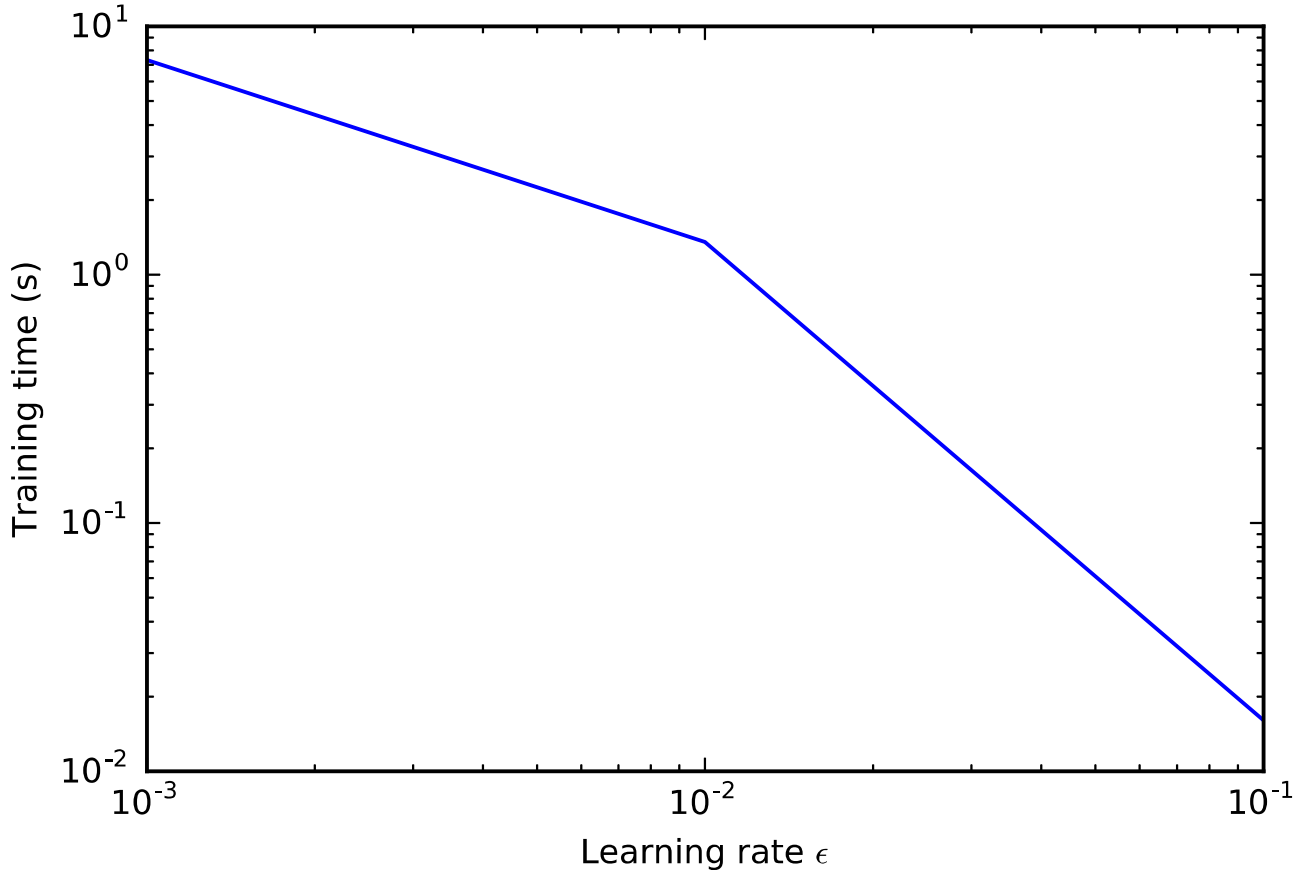
Finding the optimal weights through gradient descent

How do I pick the right step size?



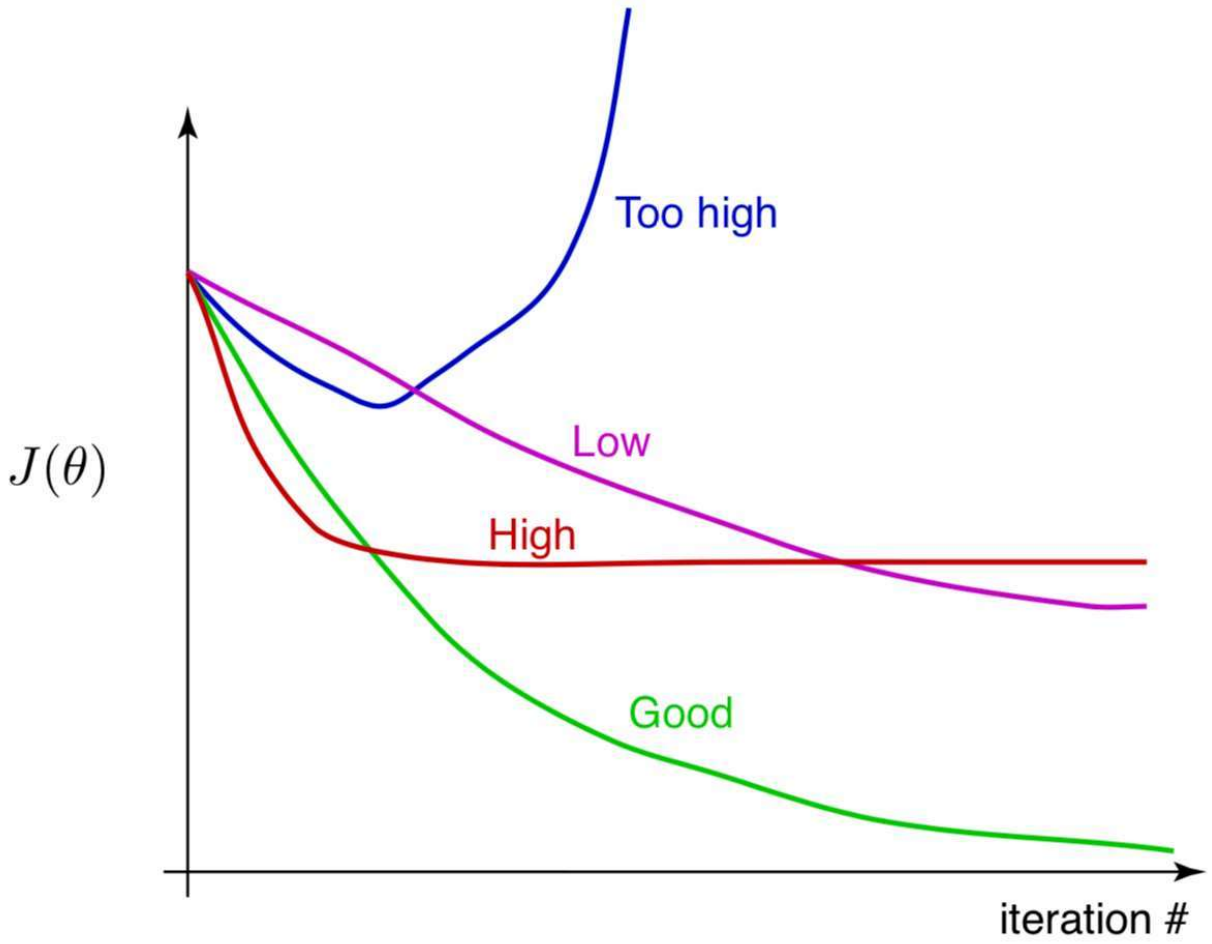
Finding the optimal weights through gradient descent

Why not always use smaller learning rates?



Interpreting the cost

The cost function can be very informative as to how to adjust your step sizes for gradient descent.



Why not use a numerical gradient?

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

Finding the optimal weights through gradient descent

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

- In this example, 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

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

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

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.

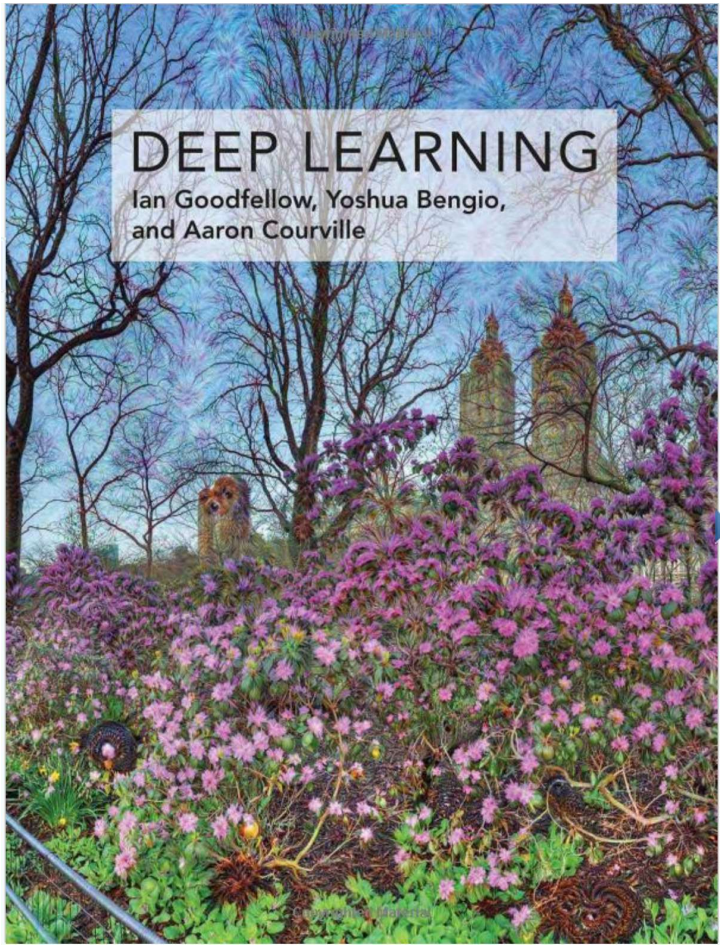
Lecture 5: Neural networks

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

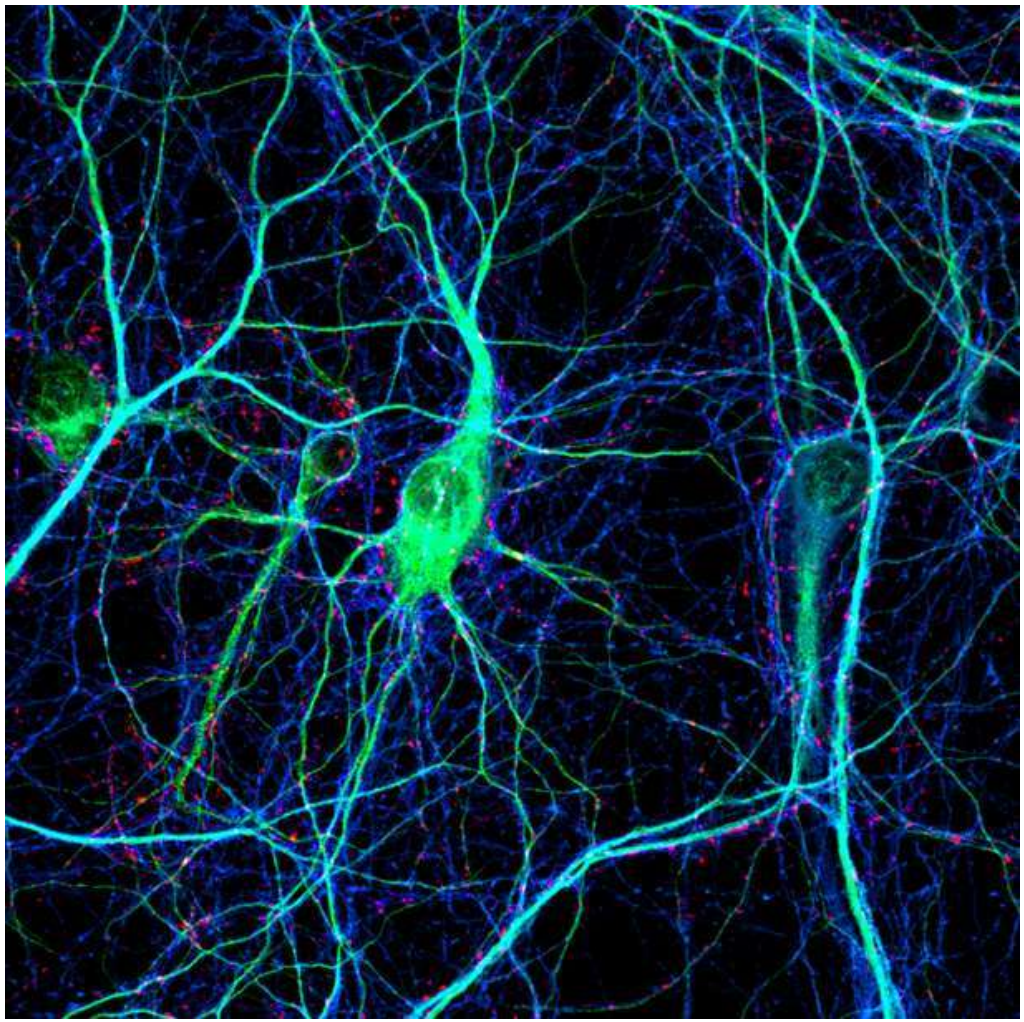
Announcements

Reading:

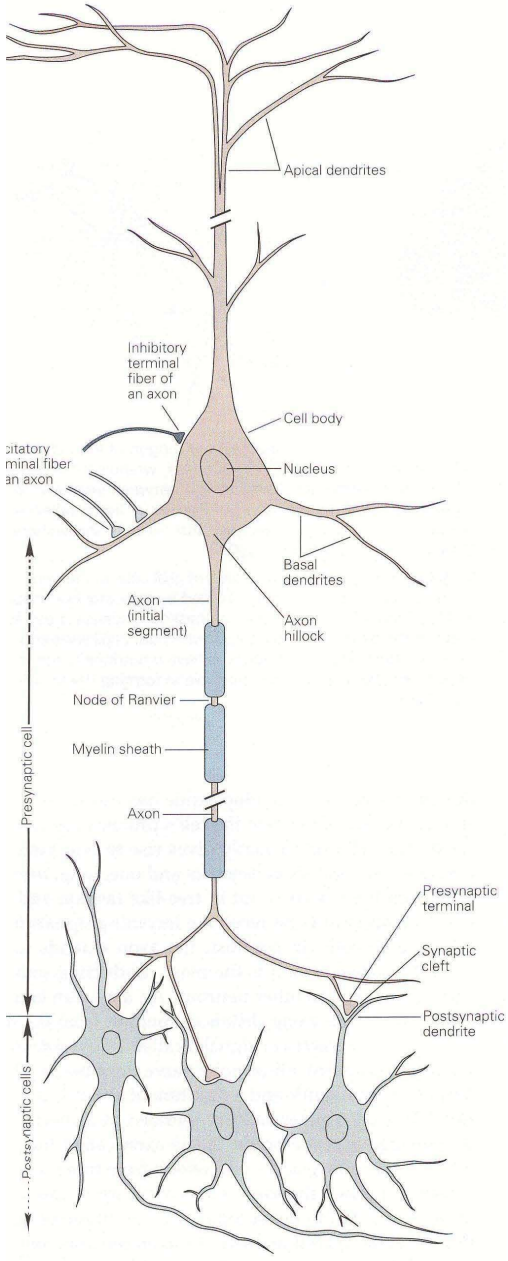
Deep Learning, 6 (intro), 6.1, 6.2, 6.3, 6.4



Inspiration from neuroscience



Inspiration from neuroscience



Neurons are the main signaling units of the nervous system.

- Neurons have four regions:
  1) Cell body (soma) – metabolic center, with nucleus, etc.
  2) Dendrites – tree like structure for receiving **input** signals.
  3) Axon – single, long, tubular structure for sending **output** signals.
  4) Presynaptic terminals – sites of communication to next neurons.

- 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)

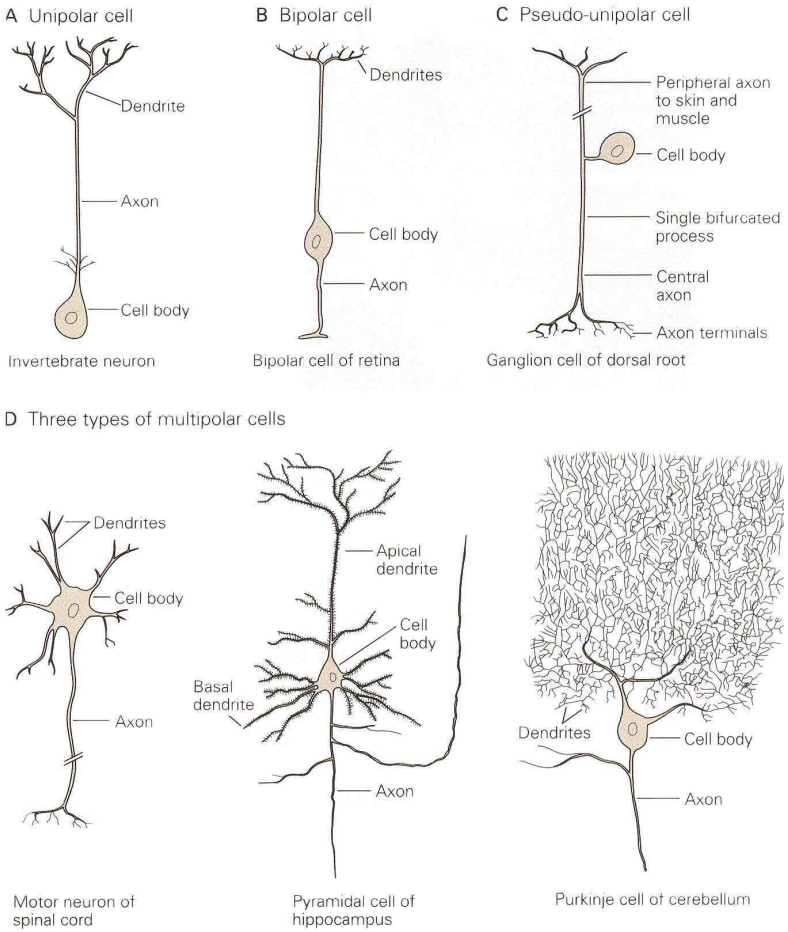
Figure 2-4 Neurons can be classified as unipolar, bipolar, or multipolar according to the number of processes that originate from the cell body.

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.

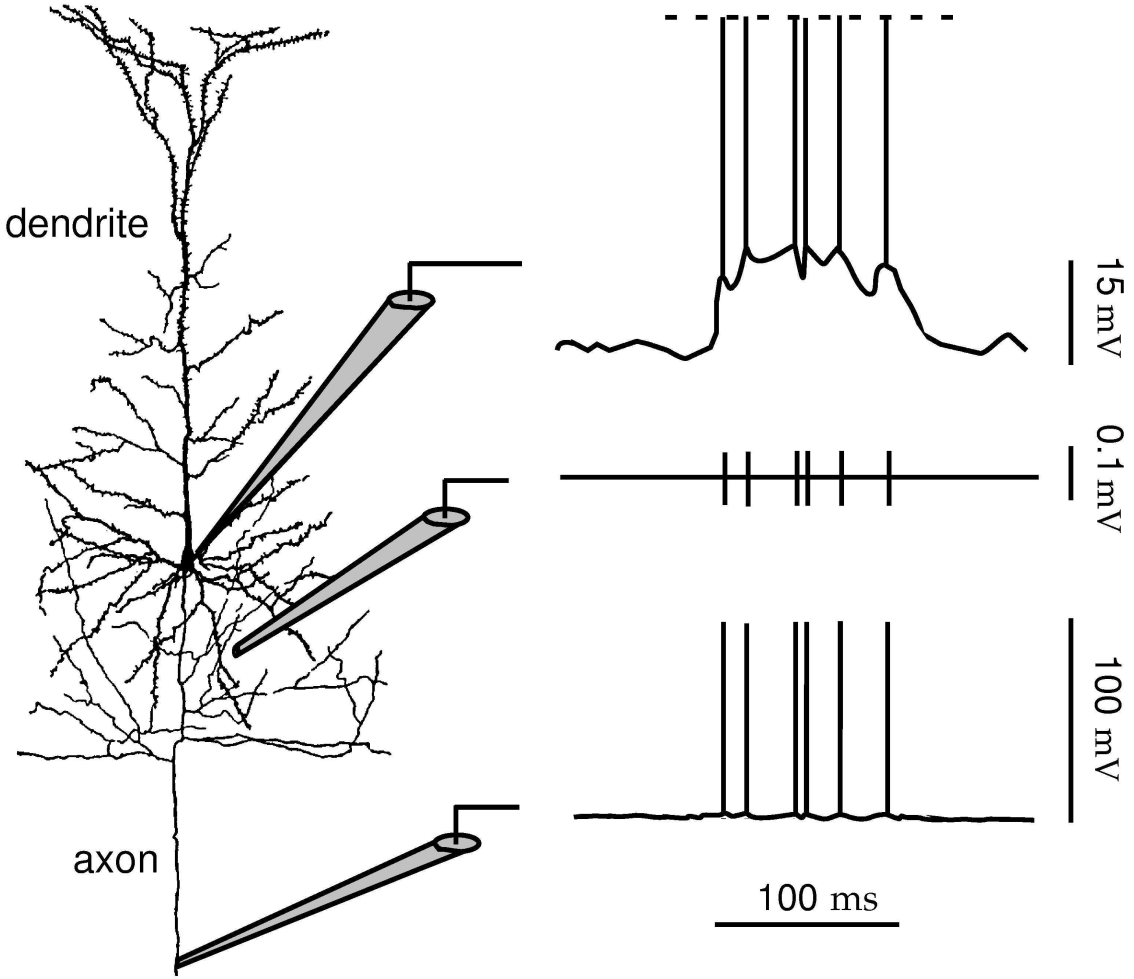
C. Certain neurons that carry sensory information, such as information about touch or stretch, to the spinal cord belong to a subclass of bipolar cells designated as pseudo-unipolar. As such cells develop, the two processes of the embryonic bipolar cell become fused and emerge from the cell body as a 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.)



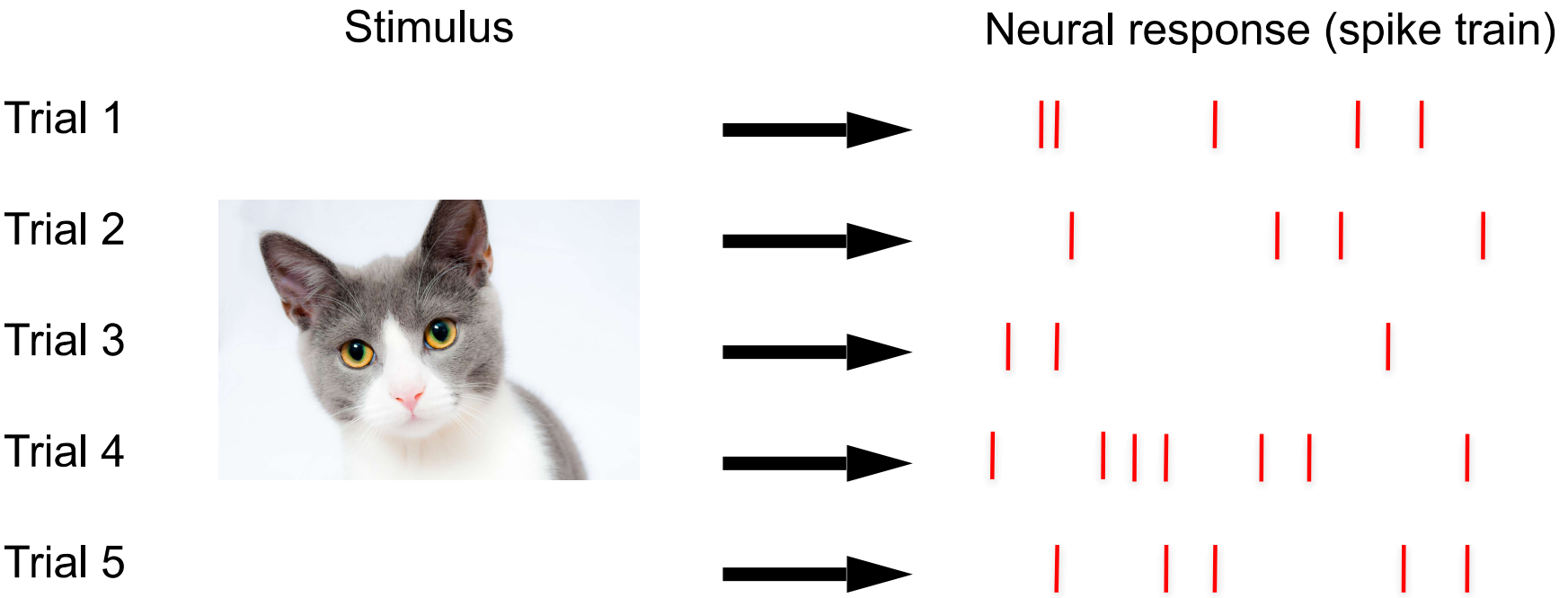
Inspiration from neuroscience

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



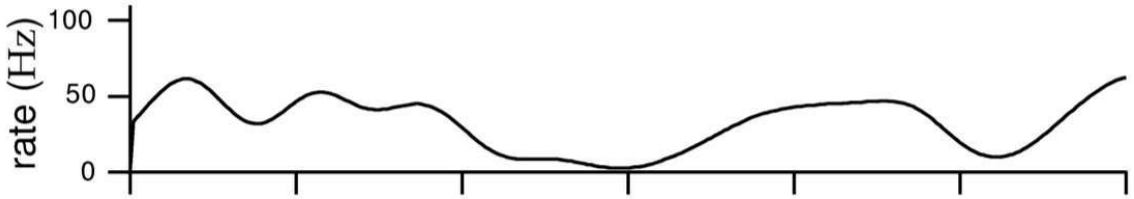
Inspiration from neuroscience

… And the spikes are probabilistic.



Inspiration from neuroscience

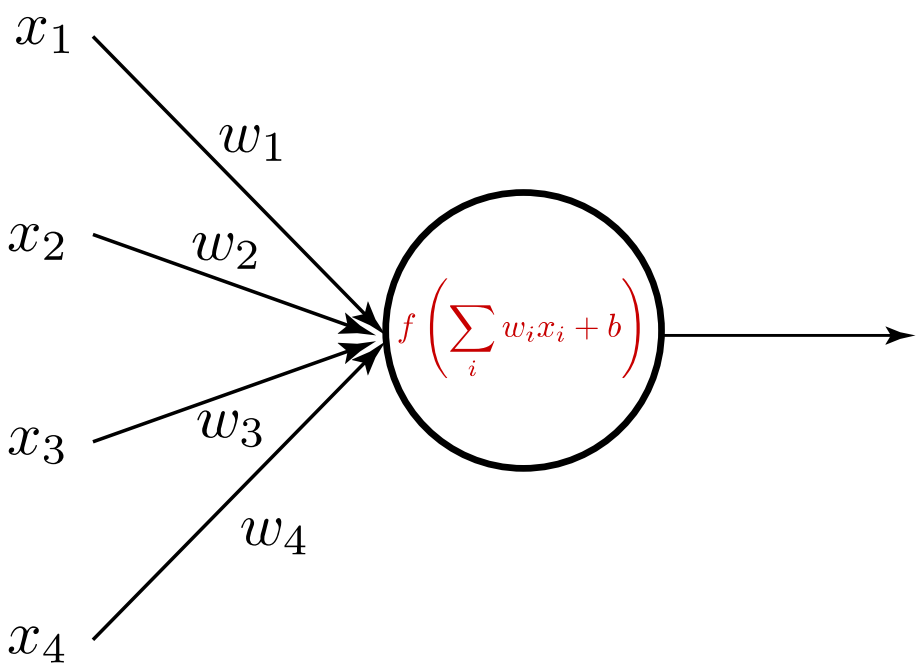
The spikes reflect an underlying rate.



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?

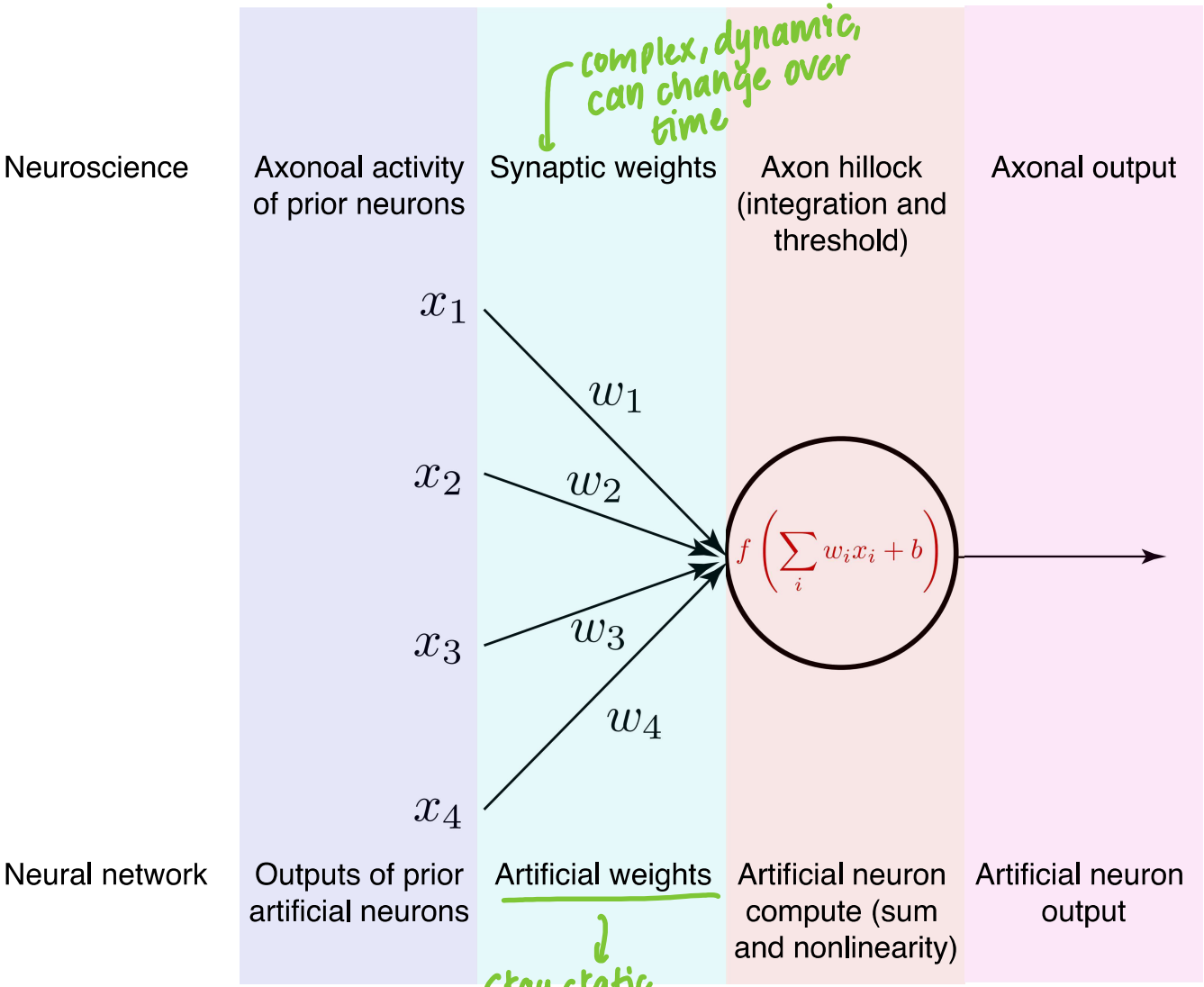
The artificial neuron (cont.)

- 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.

High level analogy only.

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...

Inspiration from neuroscience

Caution when comparing to biology

On the prior list, several of these bullet points constitute entire research areas. E.g., several labs work specifically on studying the details of synaptic transmission.

Big picture: though neural networks are inspired by biology, they approximate biological computation at a fairly crude level. These networks ought not be thought of as models of the brain, although recent work (including my research group's work) has used them as a means to propose mechanistic insight into neurophysiological computation.

Nomenclature

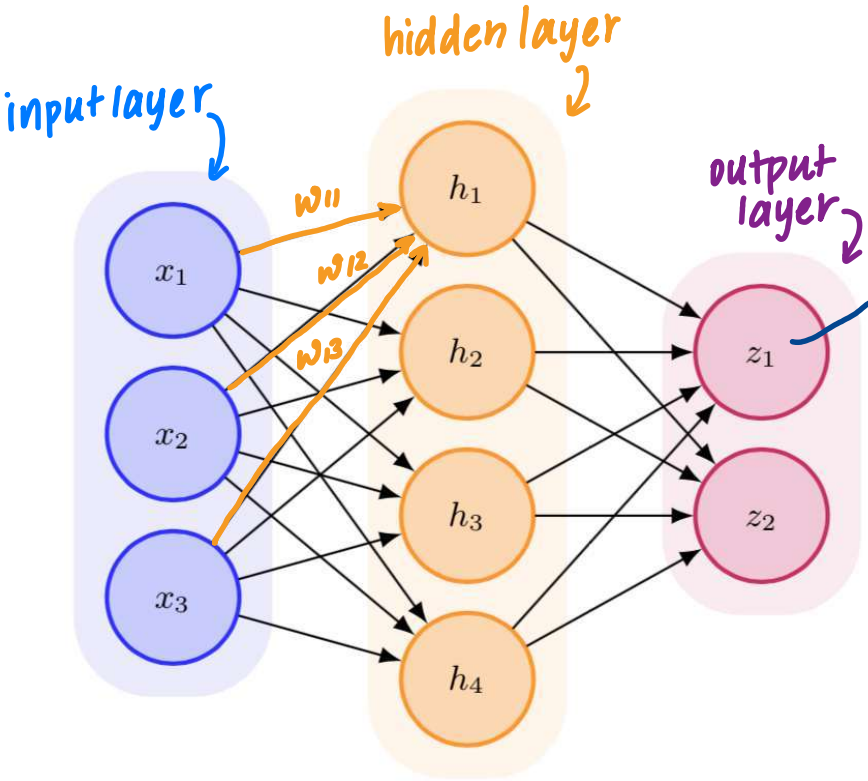
Some naming conventions.

- 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 network architecture

layers in a neural network = hidden layer + output layer

An example 2-layer network is shown below.



also called "logits"

scores that are used in a softmax classifier

Activity of 1-layer:

$h = f(Wx + b)$

$h_1 = f(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1)$

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = f\left(\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ & \vdots & \\ w_{41} & w_{42} & w_{43} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

applied element wise to the vector.

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 ($\mathbf{z} \in \mathbb{R}^2$).

Neural networks

Parameters: weights & biases

$h = f(W_1 x + b_1)$

(4x1) (4x3) (3x1) (4x1)

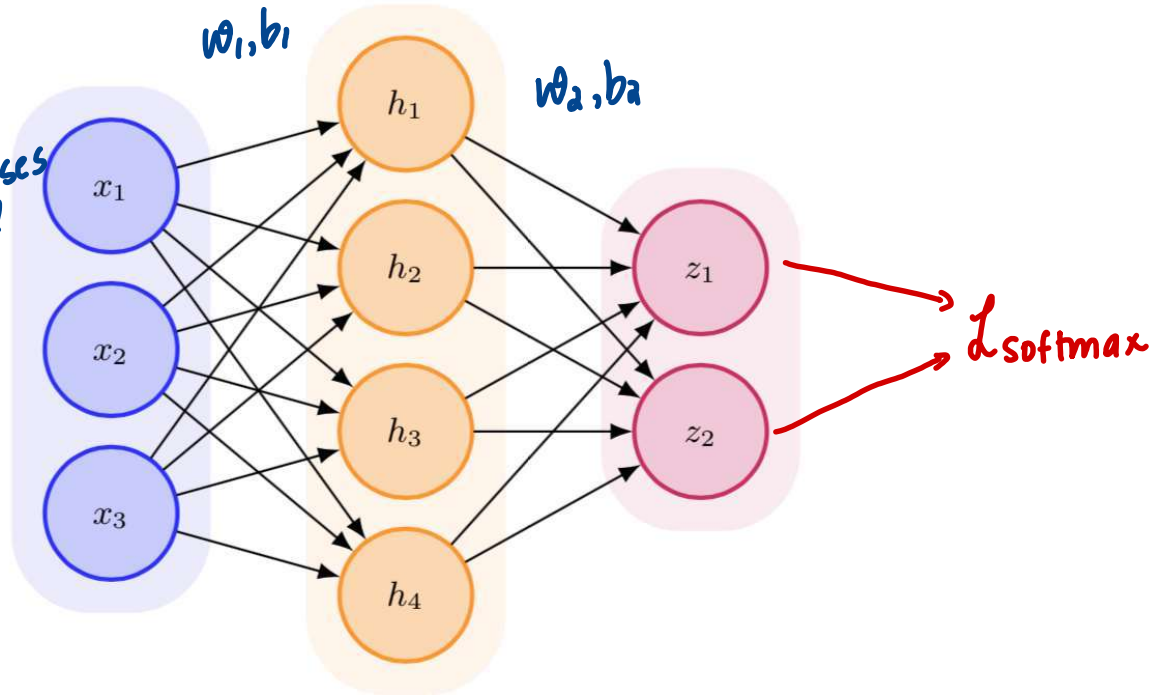
weights: 4x3 + biases: 4x1 = 16

$z = W_2 h + b_2$

(2x1) (2x4) (4x1) (2x1)

weights: 2x4 + biases: 2x1 = 10

Total learnable parameters:
= # weights + # biases
= (12+8) + (4+2) = 26



nonlinear activation function.

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

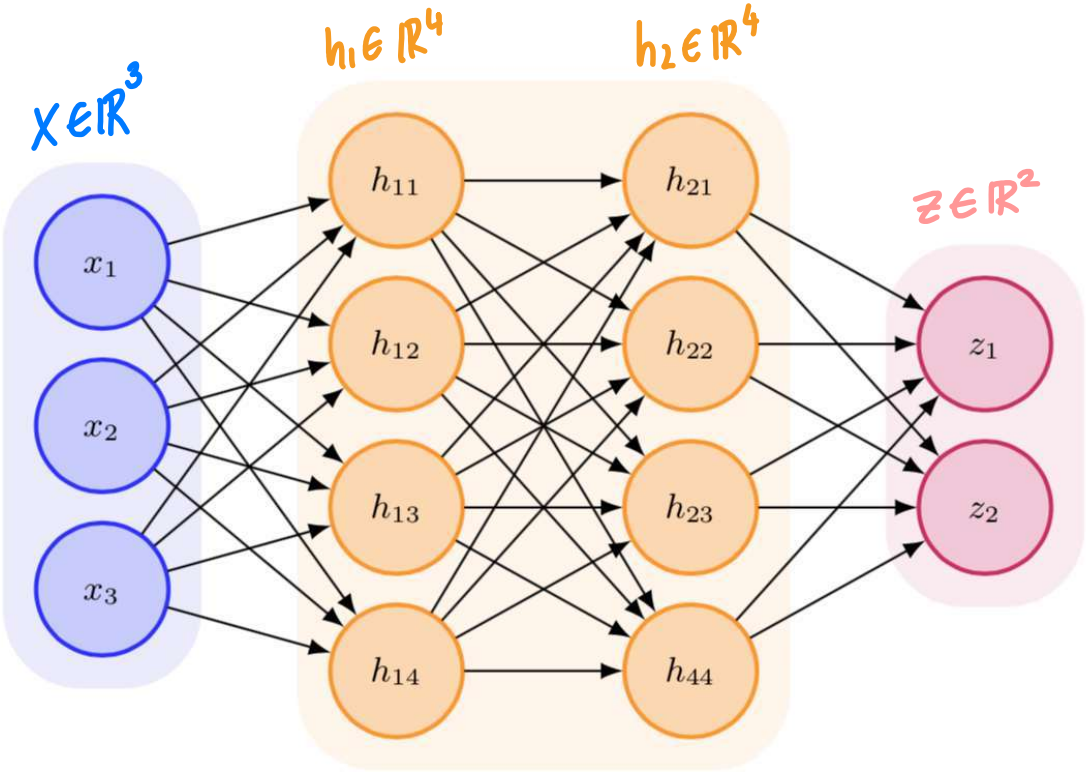
Second (output layer): $\mathbf{z} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$

output layer usually does not have an f, because output layer is a softmax classifier.

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.

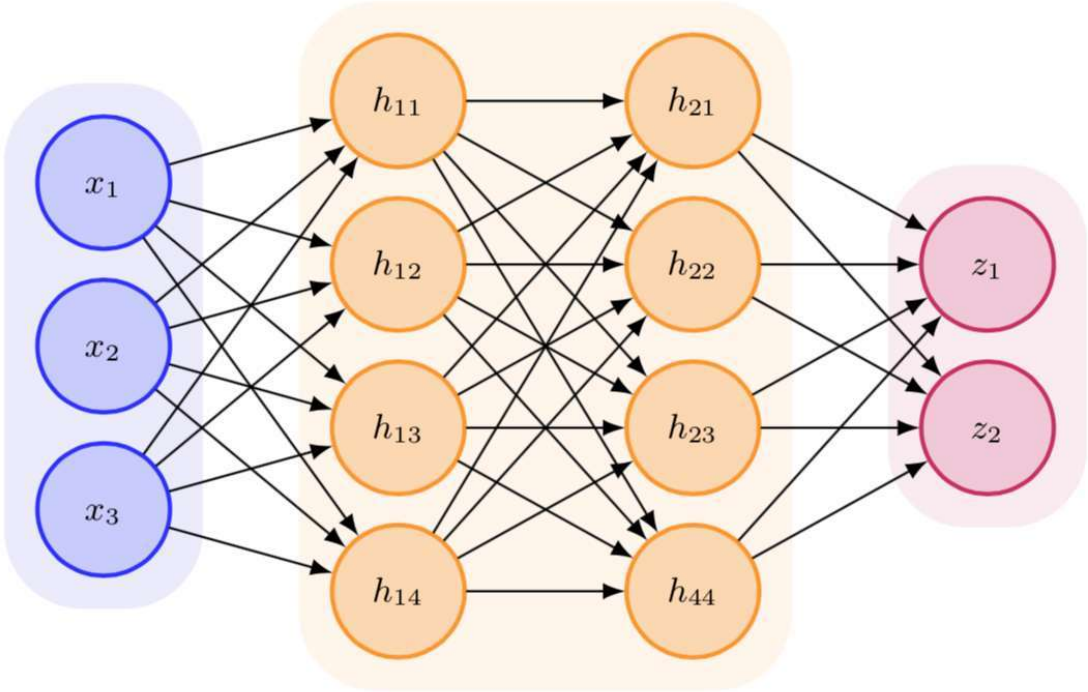
Neural network architecture 2

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



nonlinearity ↓ weights ↓ biases

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): $\mathbf{z} = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3$

softmax classifier (linear)

f does not have to be the same across all hidden layers, but in practice it usually is to reduce the # design choices to be made.

Neural networks

Fully Connected Neural Networks (FCNet)

OR multilayer perceptron. (MLP)

$h_1 = f(W_1 x + b_1)$

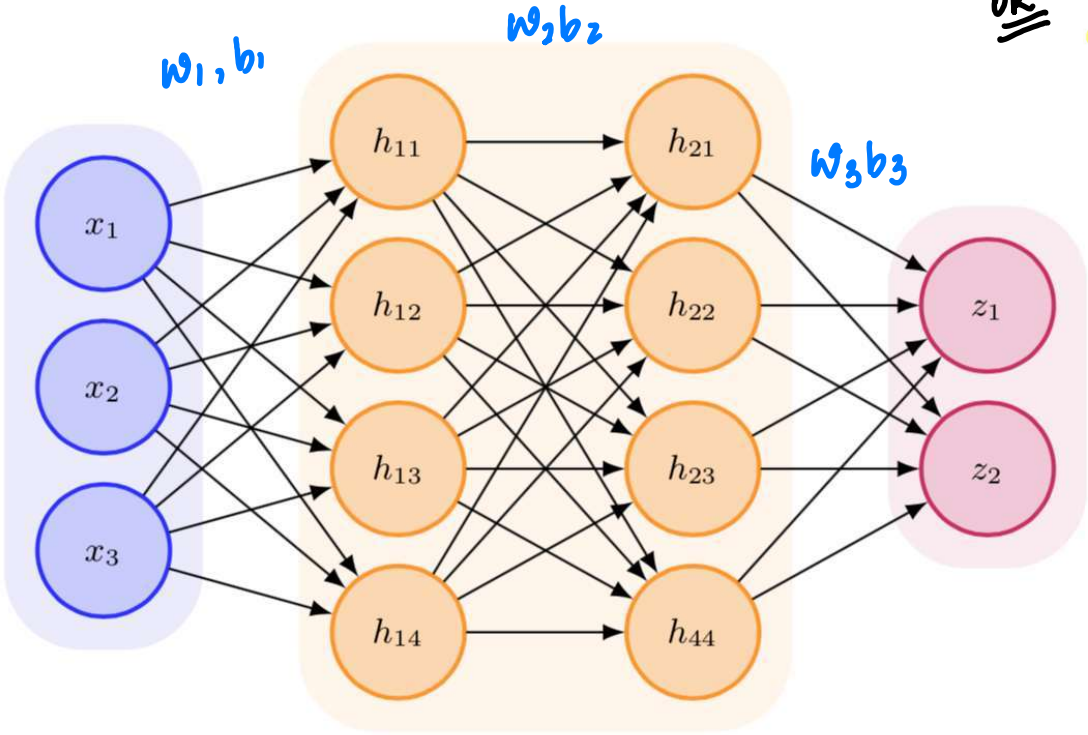
(4x1) (4x3)(3x1) (4x1)

$h_2 = f(W_2 h_1 + b_2)$

(4x1) (4x4)(4x1) (4x1)

$z = W_3 h_2 + b_3$

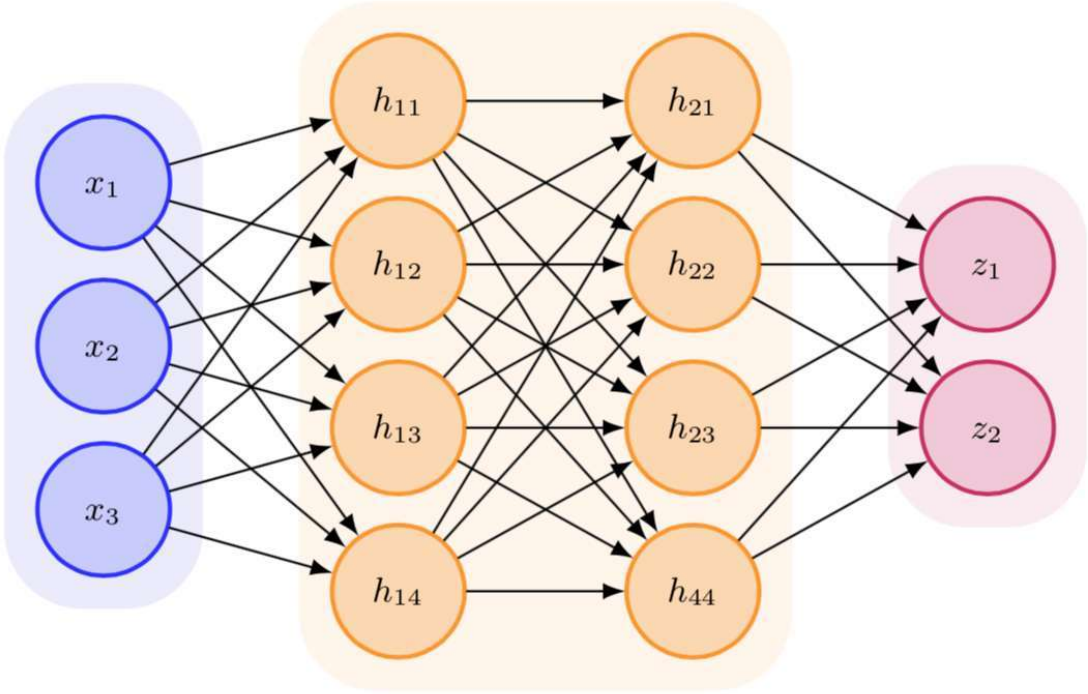
(2x1) (2x4)(4x1) (2x1)



Total # learnable parameters
= (12+4)+(16+4)+(8+2) = 16+20+10 = 46

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.

Neural networks



def f(x):
return x * (x>0)
} implementation of ReLU.

```
# 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 =  np.dot(W3, h2) + b3
```

What if f() is linear?

Neural networks

suppose $f(x) = x$ [identity]
then when you get to z, all you have is softmax classifier.

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

- 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: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

$h_2 = w_2 h_1 + b_2$
$= w_2 (w_1 x + b_1) + b_2$
$= \underbrace{w_2 w_1}_{\hat{w}} x + \underbrace{w_2 b_1 + b_2}_{\hat{b}}$
$= \hat{w}x + \hat{b}$ ← affine function of x.

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

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

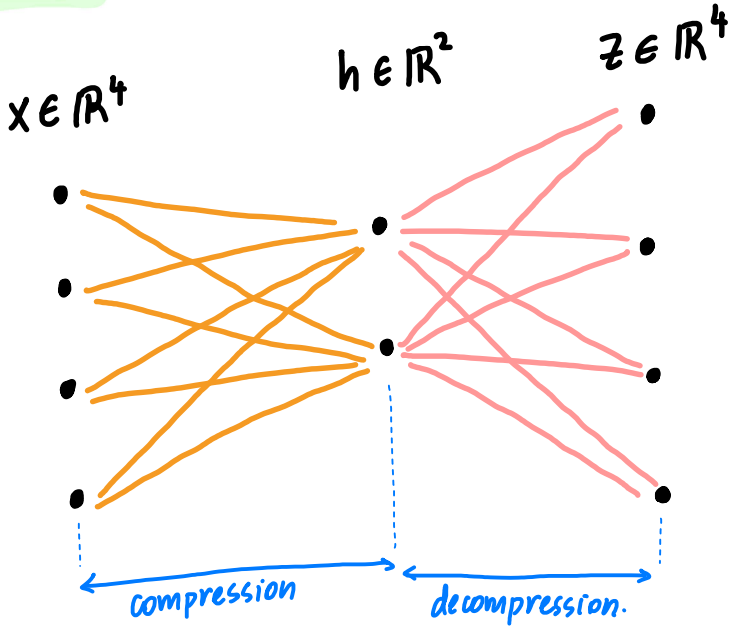
and

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

What if f() is linear?

- This may be useful in some contexts. For example, when $\dim(\mathbf{h}) \ll \dim(\mathbf{x})$, this corresponds to finding a low-rank representation of the inputs.
- However, a system with greater complexity may require a higher capacity model.

Autoencoder [suppose f is linear]



$L = \|z - x\|^2$

① If h is too small, may compress too much and lose info.
② if h is too large, may not compress enough.
③ when f is linear, you are constrained to making linear projections of original data.
④ Nonlinear f (ReLU) could have higher expressive capacity.

- compression and decompression of x.
- useful even if f is linear (linear dimensionality reduction).

autoencoder vs. variational autoencoder:

↓ $h = wx$

↓ $p(h|x)$ introduce some stochasticity into the prob.

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: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

Neural networks are doing a bunch of nonlinear transformations of your data to make it linearly separable at the output.

A few notes:

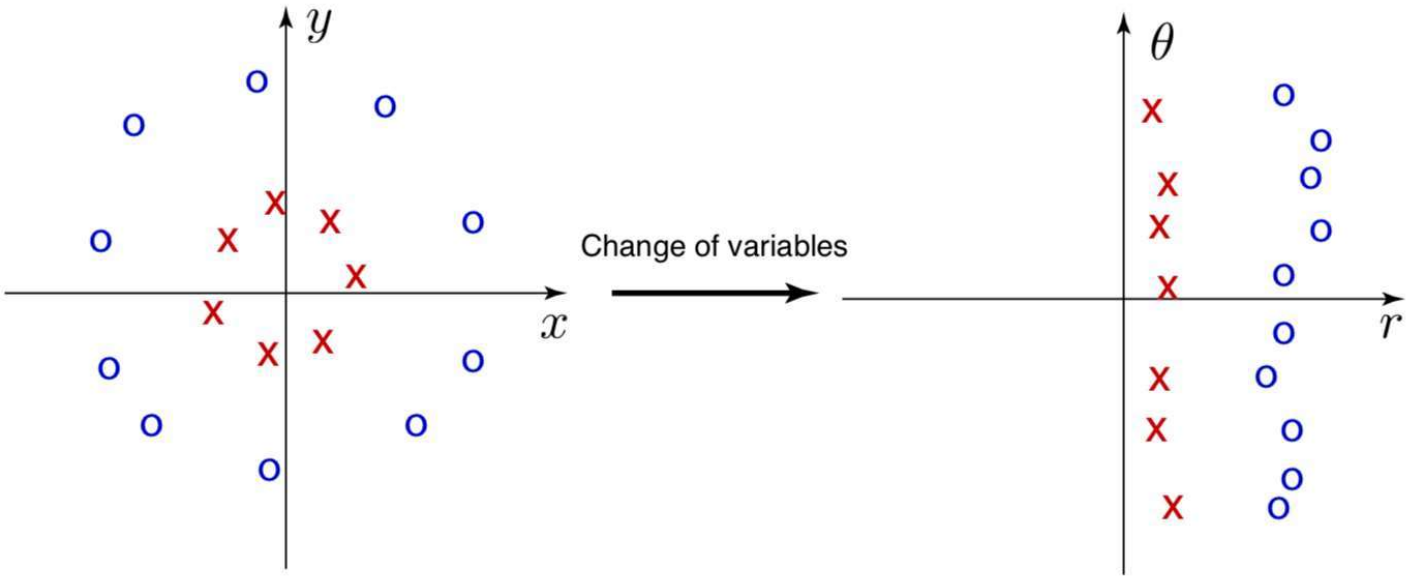
- These equations describe a *feedforward neural network*, also called a *multilayer perceptron*.
- $f(\cdot)$ is typically called an *activation function* and is applied elementwise on its input.
- The activation function does not typically act on the output layer, $\mathbf{z}$, as these are meant to 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?

not linearly separable



linearly separable.

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 $\text{xor}(\cdot)$ 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	$\text{xor}(\mathbf{x})$
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 $\text{xor}(\mathbf{x})$, but we would have samples of corresponding inputs and outputs from training data. Hence, it may be better to simply replace $\text{xor}(\mathbf{x})$ with $y(\mathbf{x})$ representing training examples.)

Review on your own

Example: XOR

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

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

How to get from ① to ②?

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

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:

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

What conclusion?

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 $\text{xor}(\cdot)$ 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

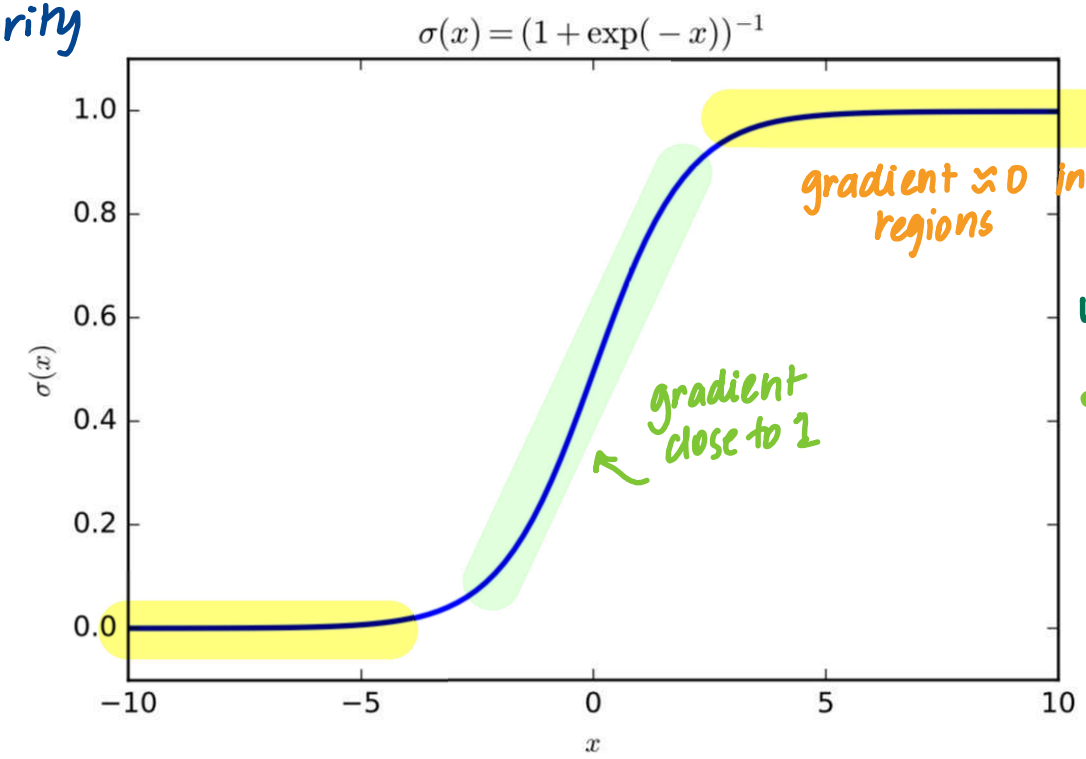
Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

"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)

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

using σ for nonlinearity

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



why is gradient ≈ 0 bad?

$\mathcal{L}(w)$ $\quad \sigma(w^T x + b)$

$w \leftarrow w - \varepsilon \frac{\partial \mathcal{L}}{\partial w}$

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

slope of σ as a function of w

if this is a very small number

Its derivative is:

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