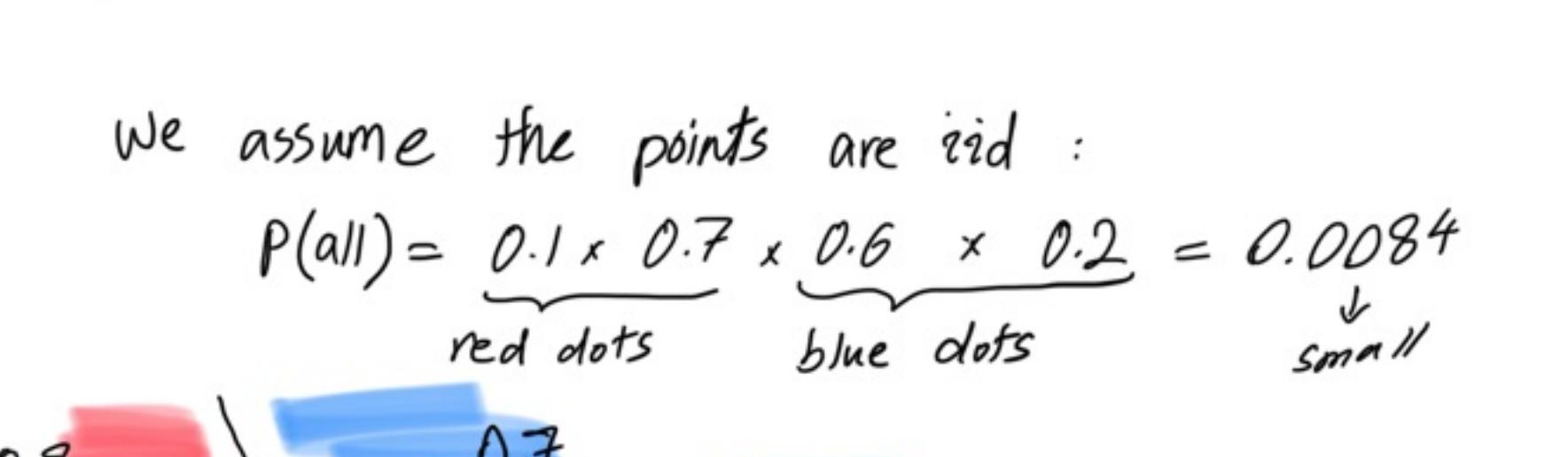


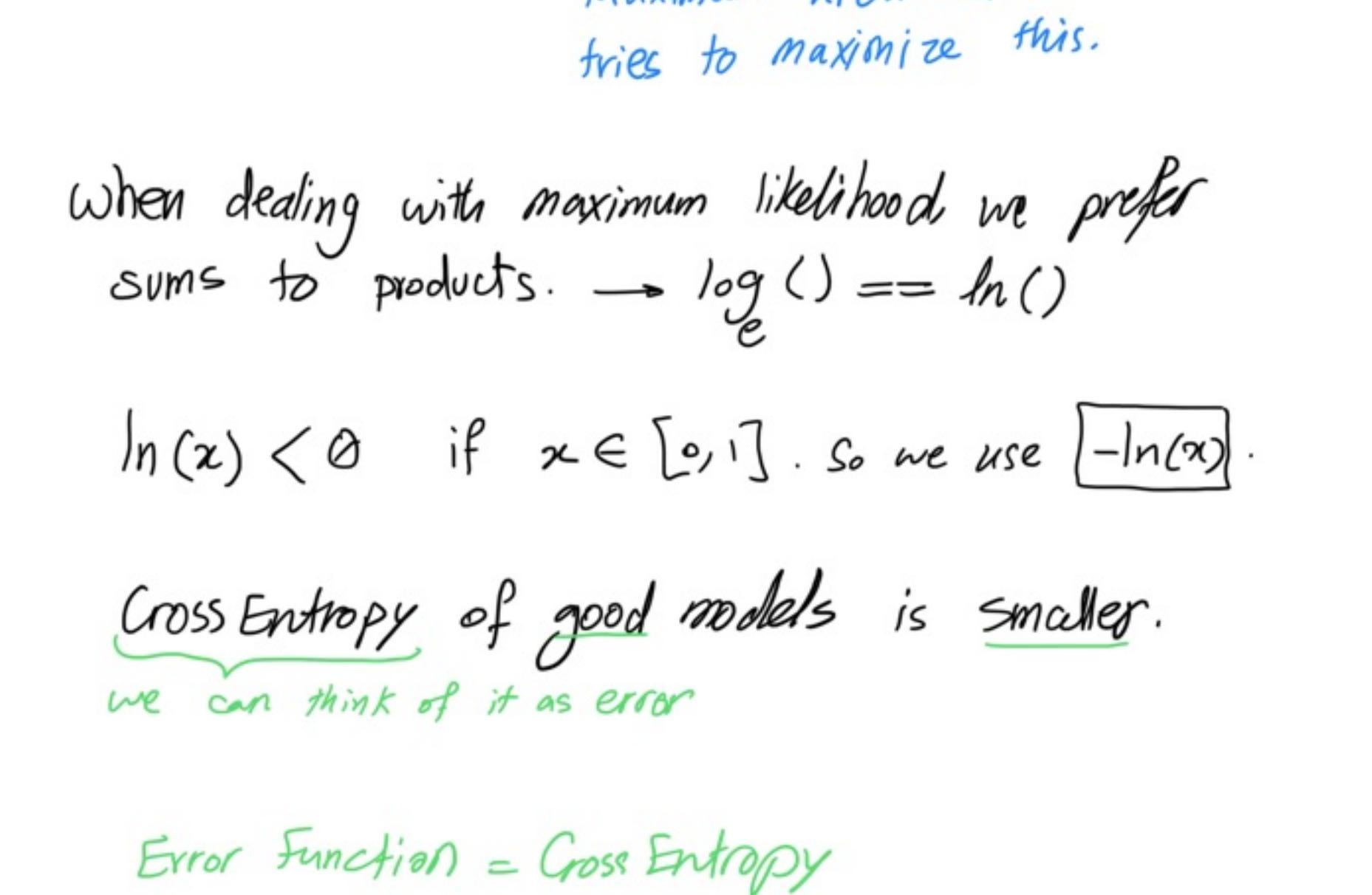
Maximum Likelihood

Pick the model that gives the existing labels the highest probability.
So we try to maximize this probability.

example:



We assume the points are iid :



When dealing with maximum likelihood we prefer sums to products. $\rightarrow \log() = \ln()$

$\ln(x) < 0$ if $x \in [0, 1]$. So we use $[-\ln(x)]$.

Cross Entropy of good models is smaller.
we can think of it as error

Error Function = Cross Entropy

So we now change the goal to minimizing cross entropy.

Probabilities -

	G	R	B
$P(\text{gift})$	0.8	0.7	0.1
$P(\text{no gift})$	0.2	0.3	0.9

most likely event?

choosing the largest

prob. in each column.

$$0.8 \times 0.7 \times 0.9$$

$$\text{Cross Entropy} = - \sum_{i=1}^m y_i \ln(P_i) + (1-y_i) \ln(1-P_i)$$

it can tell us

how close two

$$\text{CE}[(1, 1, \theta), (0.8, 0.7, 0.1)] = 0.69$$

$$\text{CE}[(0, 0, 1), (0.8, 0.7, 0.1)] = 5.12$$

likely to

happen

Multi-class Entropy:

$$\text{CE} = - \sum_{i=1}^n \sum_{j=1}^m y_{ij} \ln(P_{ij})$$

probability of i and j

$\rightarrow 0/1$ to indicate which

event we are talking about.



$$\left\{ \begin{array}{l} \text{if } y=1 \rightarrow P(\text{blue}) = \hat{y} \rightarrow \text{Error} = -\ln \hat{y} \\ \text{if } y=0 \rightarrow P(\text{red}) = 1 - P(\text{blue}) = 1 - \hat{y} \rightarrow \text{Error} = -\ln(1 - \hat{y}) \end{array} \right.$$

$$\text{Error function: } \frac{-1}{m} \sum_{i=1}^m (1-y_i)(\ln(1-\hat{y}_i)) + y_i \ln \hat{y}_i$$

binary

same as:

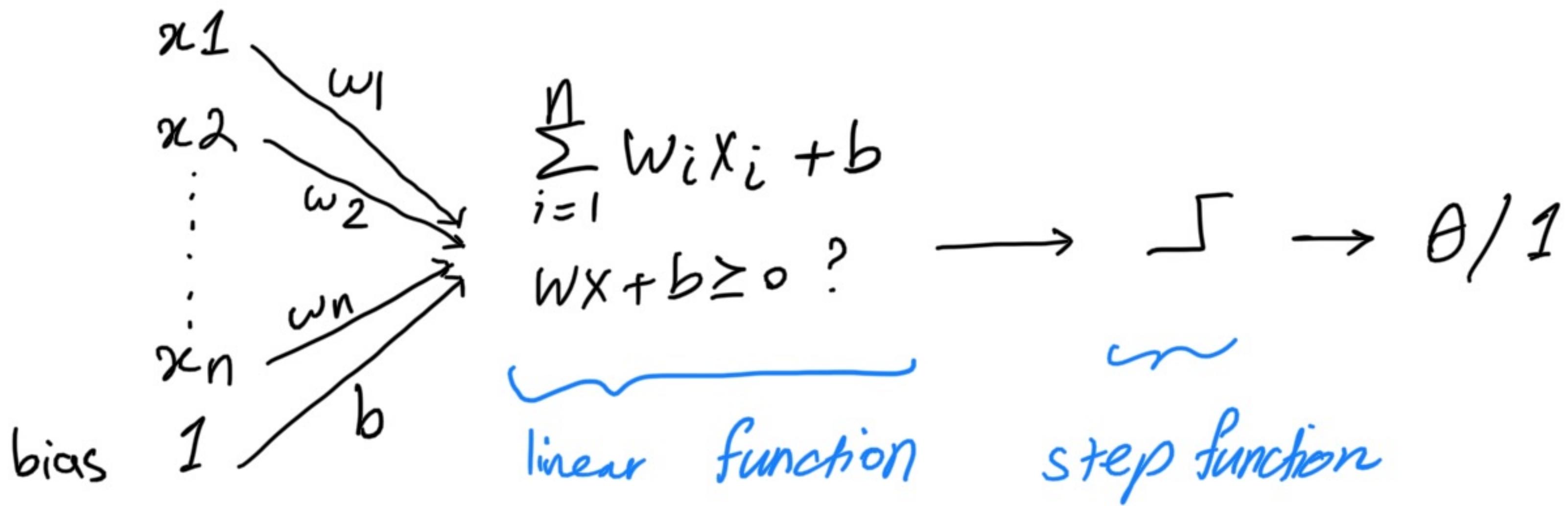
classification:

$$E(w, b) = \frac{-1}{m} \sum_{i=1}^m (1-y_i) \ln(1 - \sigma(wx_i + b)) + y_i \ln \sigma(wx_i + b)$$

Multi-class classification:

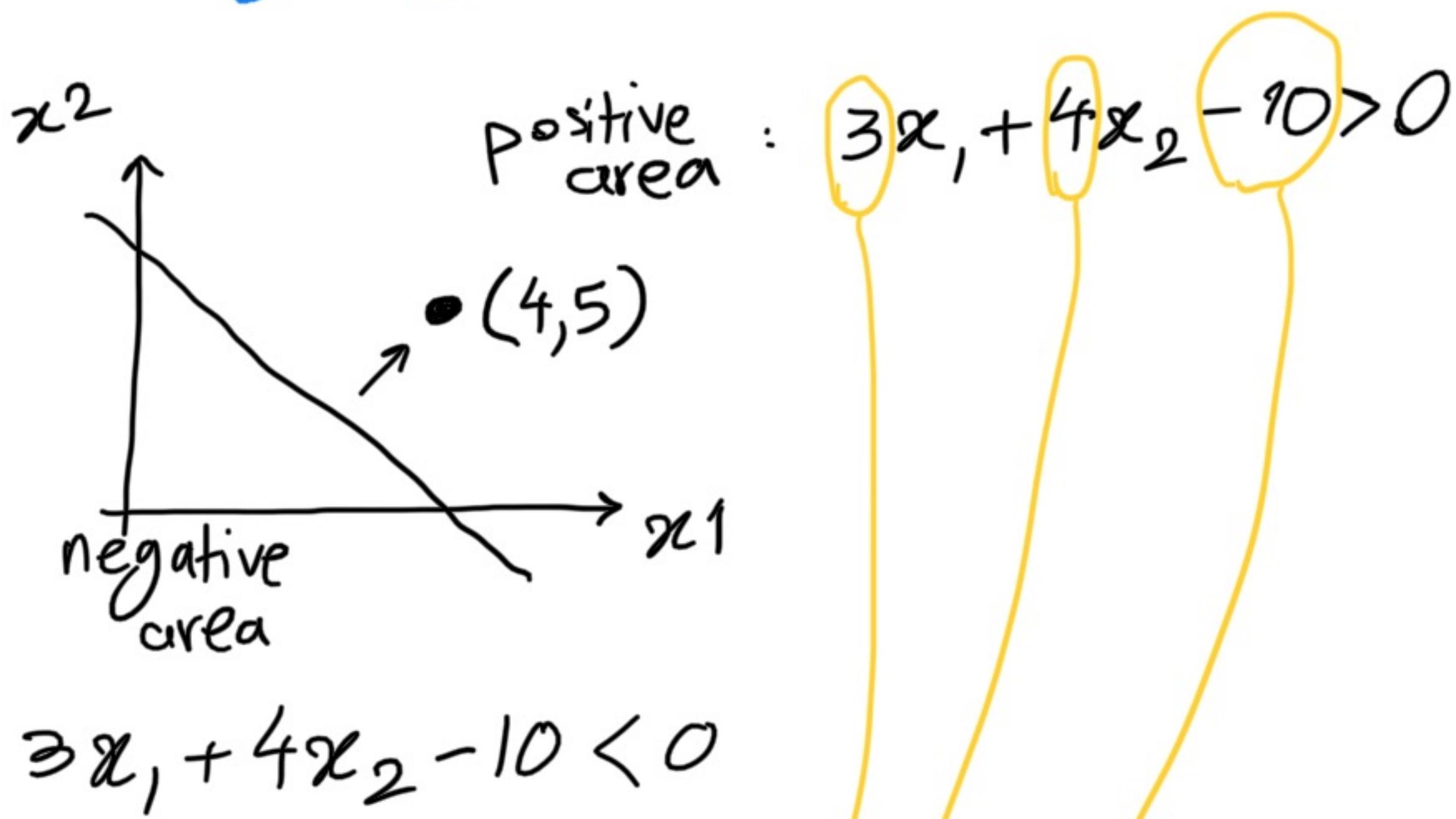
$$\frac{-1}{m} \sum_{i=1}^m \sum_{j=1}^n y_{ij} \ln \hat{y}_{ij}$$

Perceptron



Perceptron Algorithm Goal : split data

move this until we get a good split
 lines that are misclassified, want the line to come closer.



line should make small steps towards the line.
 But we don't want huge steps.

That's why we use learning rate, e.g., 0.1.

if point was in negative area

point is in pos. area

$$+ (4 \quad 5 \quad 1) \times 0.1$$

New line : $2.6x_1 + 3.5x_2 - 9.1 = 0$

Perceptron Algorithm

1. Start with random weights w_1, \dots, w_n, b
2. for every misclassified point (x_1, \dots, x_n) :
 - if prediction == 0:
 - for $i=1 \dots n$: update $w_i = w_i + \alpha x_i$
 - update b to $b + \alpha$
 - if prediction == 1:
 - for $i=1 \dots n$: update $w_i = w_i - \alpha x_i$
 - update b to $b - \alpha$

* Perceptron algorithm has to be generalized for non-linear stuff.

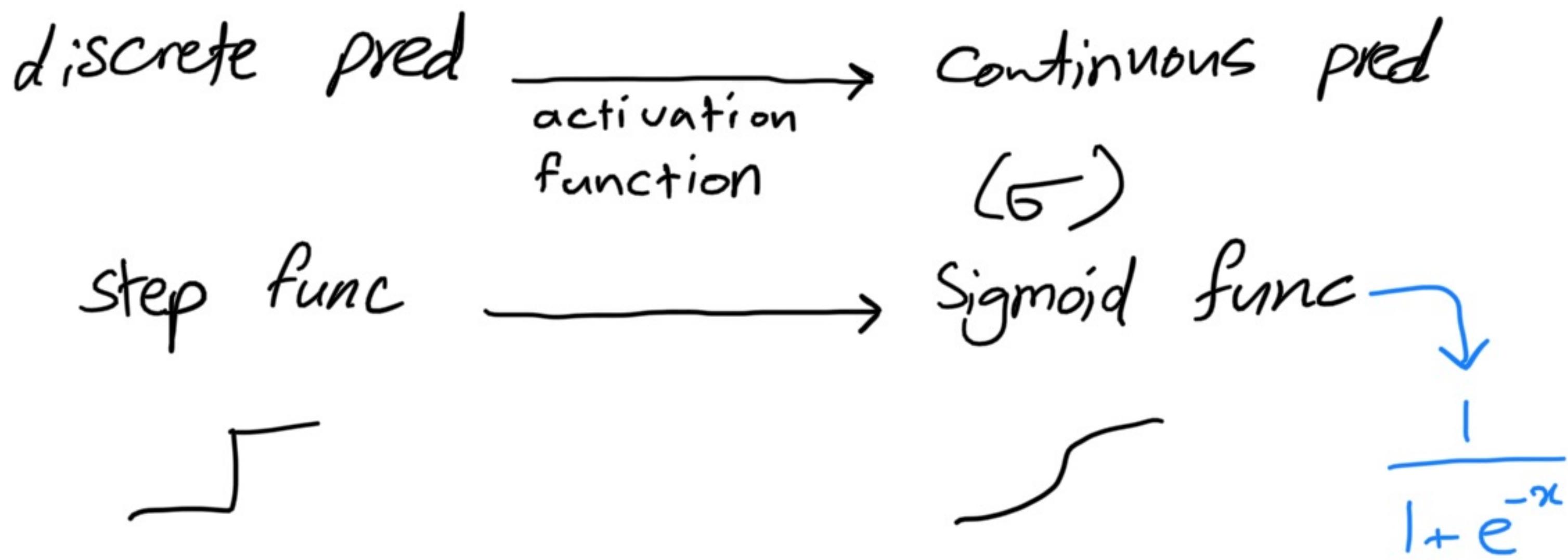
* Error function = Distance : how far are we from the solution?

Gradient Descent :

minimizing the error.

error function must be differentiable
continuous

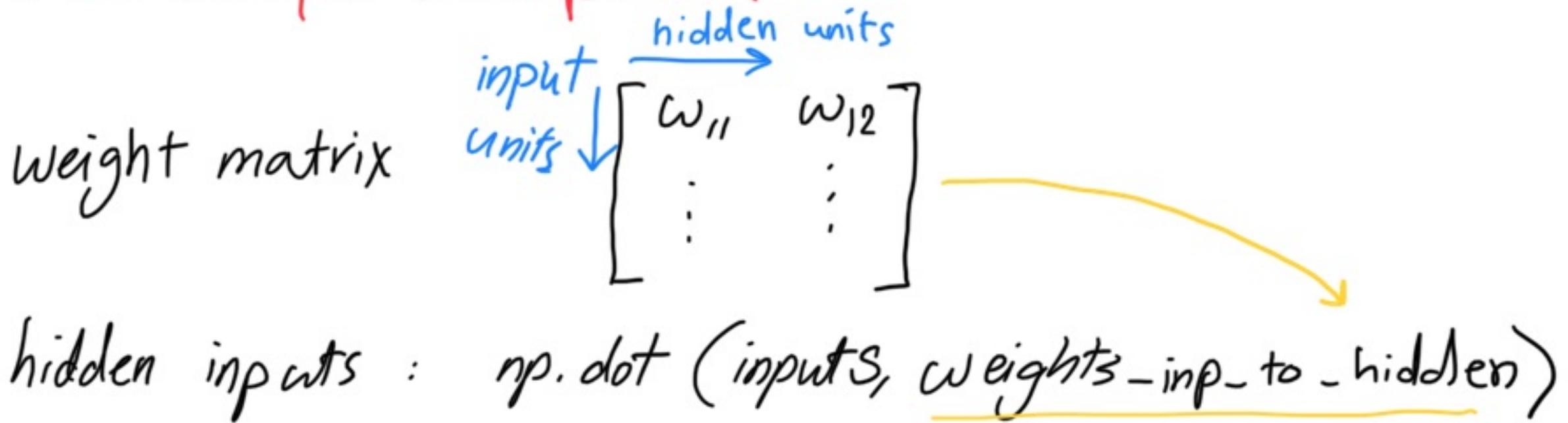
So, how do we go from discrete to continuous?



$wx+b$ gives us \hat{y} . we have to apply $\sigma()$ to \hat{y} .

when the score is zero, probability is 50%.

Multi layer Perceptron



Some numpy tips :

for creating a column vector: `data[:, None]`

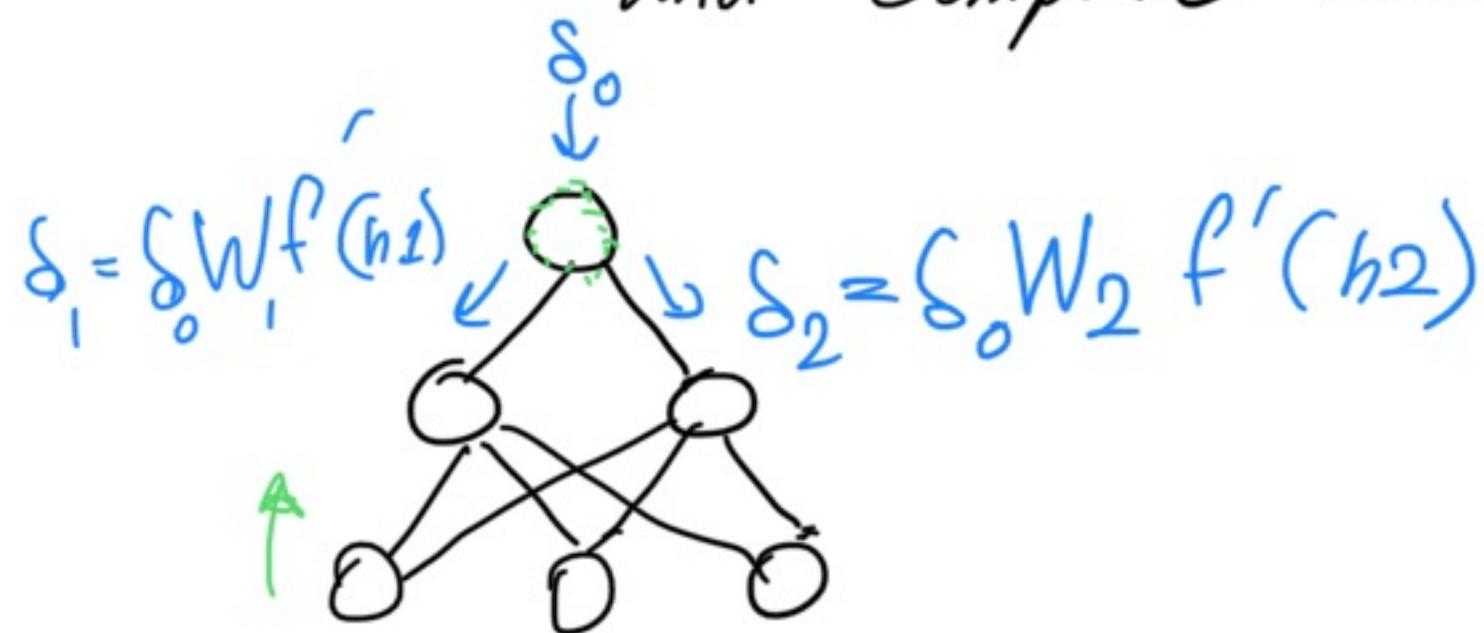
for transposing a matrix: `data.T`

* Backpropagation

same as forward but we assume

the output layer is the input

and compute each layer's error.



Squared Errors

$$E = \frac{1}{2} \sum_m \sum_j [y_j^m - \hat{y}_j^m]^2$$

Annotations:

- m (circled) points to "data points".
- j (circled) points to "output units".
- y_j^m points to "true value".
- \hat{y}_j^m points to "prediction".
- The entire term $[y_j^m - \hat{y}_j^m]^2$ depends on w_{ij} 's.

(SSE)
sum of squared errors

Gradient is another word for slope or rate of change.

Gradient descent can get into "local minima".

One way to solve is "momentum".

error term : $\delta = (y - \hat{y}) f'(h)$

↳ activation function

$$w_i = w_i + \eta \delta x_i$$

$$h = \sum w_i x_i$$

Mean Squared Error (MSE)

$$E = \frac{1}{2m} \sum_m (y^m - \hat{y}^m)^2$$

Multi-class Classification

We use softmax instead of sigmoid. e.g.,

class1 → score=2

$$\frac{2}{(2+1+\theta)}$$

class2 → score=1

$$\frac{1}{(2+1+\theta)}$$

class3 → score=0

$$\frac{0}{(2+1+\theta)}$$

but it doesn't work
for negative scores.



to solve the issue, we
make all the score positive.

instead of $\frac{2}{(2+1+\theta)}$ we have $\frac{e^2}{(e^2 + e^1 + e^0)}$.

Softmax function

it converts score to probability.

$$\text{probability(class } i) = \frac{e^{z_i}}{\sum_n e^{z_j}}$$

$z_1 \dots z_N$ are scores.

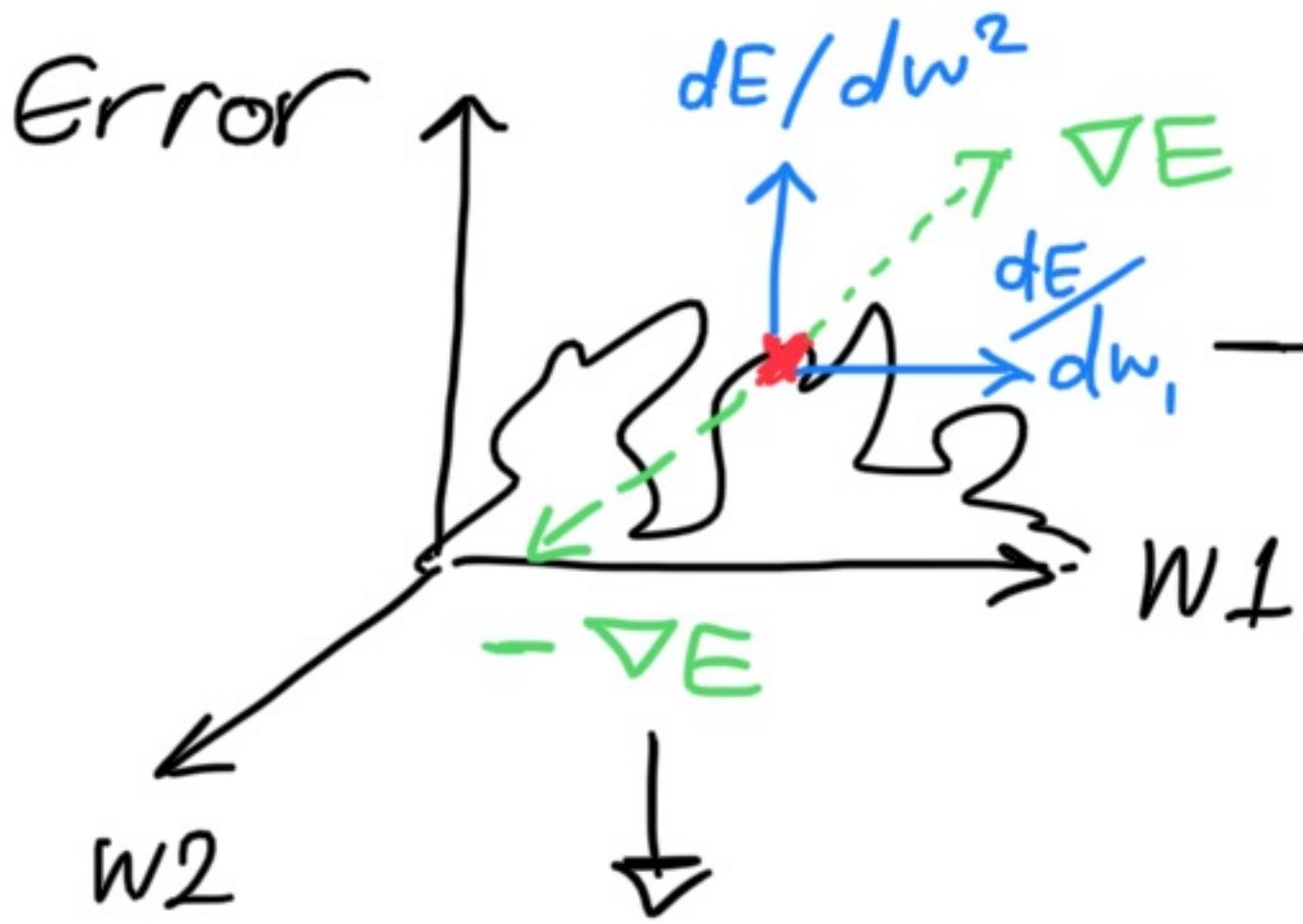
One-hot encoding:

if 3 classes →

c ₀	c ₁	c ₂
0	0	1

 when label is c2.

Gradient Descent



minimizing the error means reaching axis w_1 as much as we can.

$-\nabla E$ tells us how to decrease E the most.

In models:

$$\hat{y} = \sigma(wx + b) \quad \text{Bad}$$

$$\hat{y} = \sigma(w_1x_1 + \dots + w_nx_n + b)$$

$$\nabla E = \frac{\partial E}{\partial w_1} + \dots + \frac{\partial E}{\partial w_n} + \frac{\partial E}{\partial b}, \quad \alpha = 0.1 \quad \text{learning rate}$$

$$w'_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i} \quad \text{we want to take small steps.}$$

$$b' \leftarrow b - \alpha \frac{\partial E}{\partial b}$$

$$\Rightarrow \hat{y}' = \sigma(w'x + b') \quad \leftarrow \text{better}$$

replacing

$$\Rightarrow E \quad \nabla E = -(y - \hat{y})(x_1, \dots, x_{n+1})$$

gradient at point (x_1, \dots, x_n) label y

$$\Rightarrow w'_i \leftarrow w_i + \alpha(y - \hat{y})x_i$$

$$b' \leftarrow b + \alpha(y - \hat{y})$$

Gradient Descent Algorithm:

1. Start with random weights w_1, \dots, w_n, b

2. For every point (x_1, \dots, x_n) :

For $i = 1 \dots n$:

$$\text{update } \hat{w}_i \leftarrow w_i - \alpha(\hat{y} - y)x_i$$

$$\text{update } b' \leftarrow b - \alpha(\hat{y} - y)$$

3. Repeat until error is small \rightarrow #epochs

« Summary »

sigmoid activation function $\sigma(x) = \frac{1}{1 + e^{-x}}$

output (prediction) formula $\hat{y} = \sigma(w_1x_1 + w_2x_2 + b)$

Error function

$$\text{Error}(y, \hat{y}) = -y \log \hat{y} - (1-y) \log(1-\hat{y})$$

Updating weights

$$w_i \rightarrow w_i + \alpha(y - \hat{y})x_i$$

$$b \rightarrow b + \alpha(y - \hat{y})$$

Perceptron vs. Gradient Descent

in GDA: change w_i to $w_i + \alpha(y - \hat{y})x_i$ every time

in PA: only update if misclassified

In fact, they are basically the same!

in GDA: both of come closer & go further away

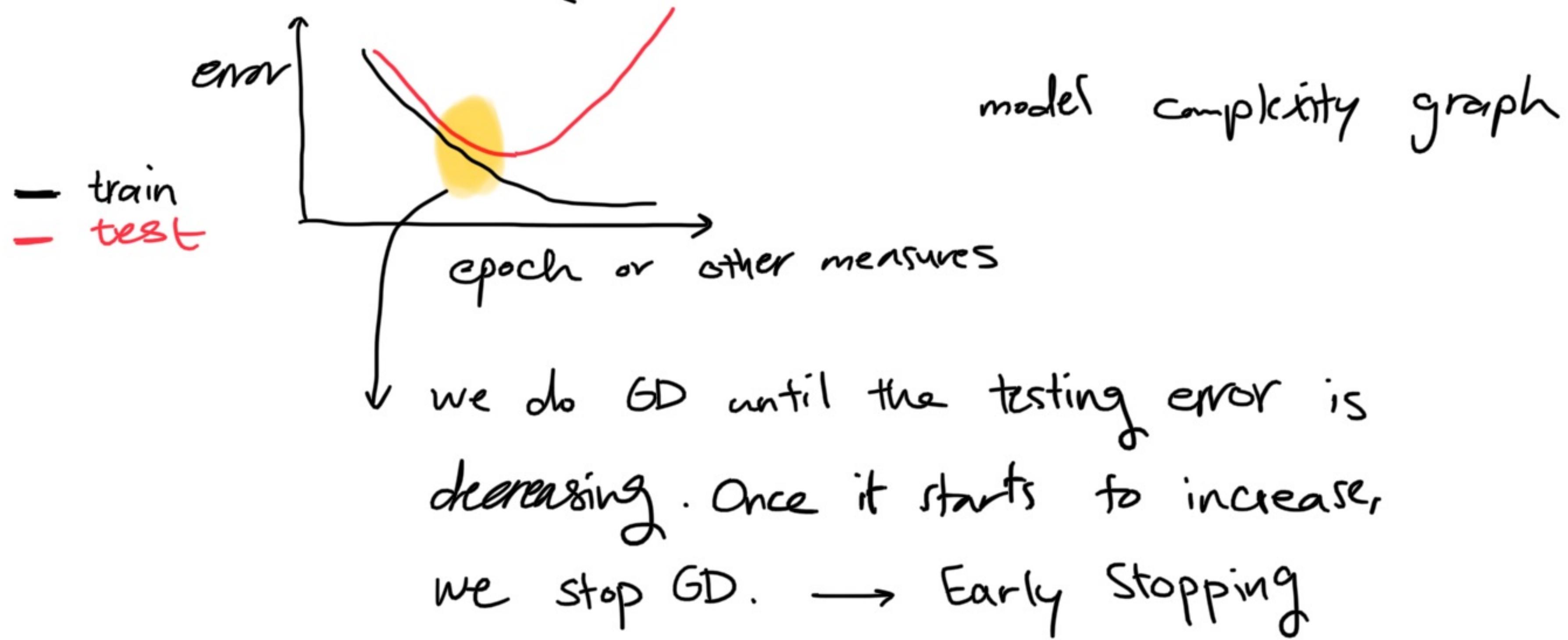
in PA: only come closer! or do nothing

Some keywords

underfitting = error due to bias (high bias)

overfitting = error due to variance (high variance)

$$\text{indicator} = \begin{cases} \text{test error: large} \\ \text{train error: tiny} \end{cases} \rightarrow \text{overfit}$$



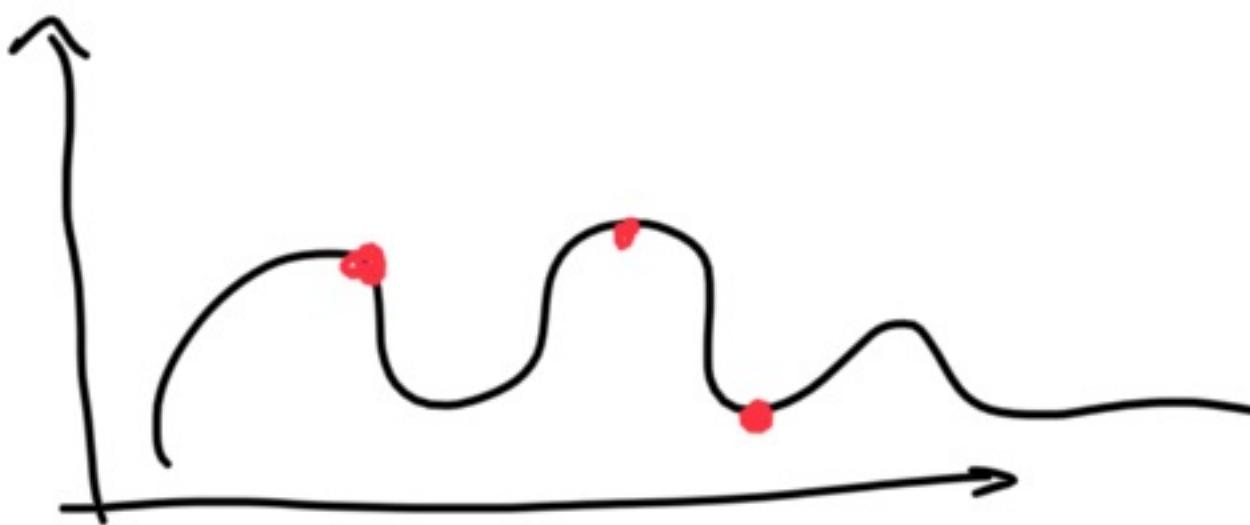
Batch Gradient Descent → for all the data

Stochastic Gradient Descent → make small batches of data

Learning Rate Decay:

rule of thumb: if the model is not working increase the rate.

Random restart to solve local minima



fixing Vanishing Gradient problem:

1. Changing the activation function. tanh, ReLU

Momentum to solve local minima:

β : momentum $\in (0, 1)$

$\text{step}(t) = \text{step}(t) + \beta \text{step}(t-1) + \beta^2 \text{step}(t-2) \dots$

↳ intuitively it gets us out of local minima.

Dropout

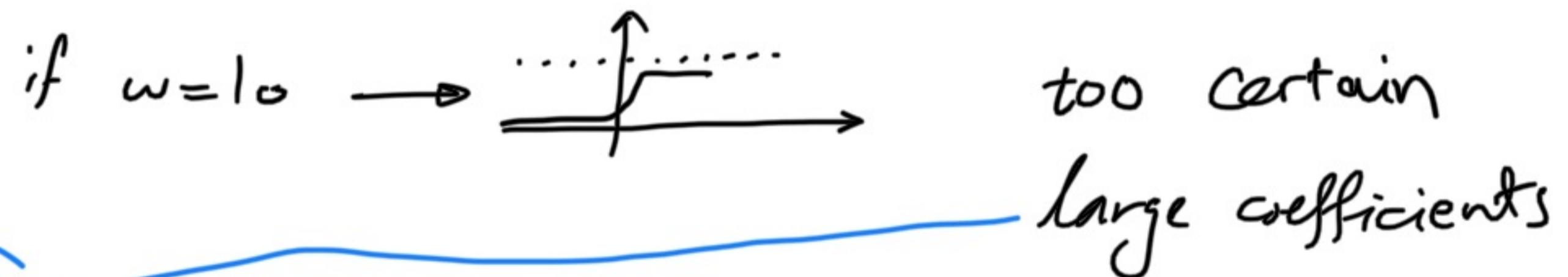
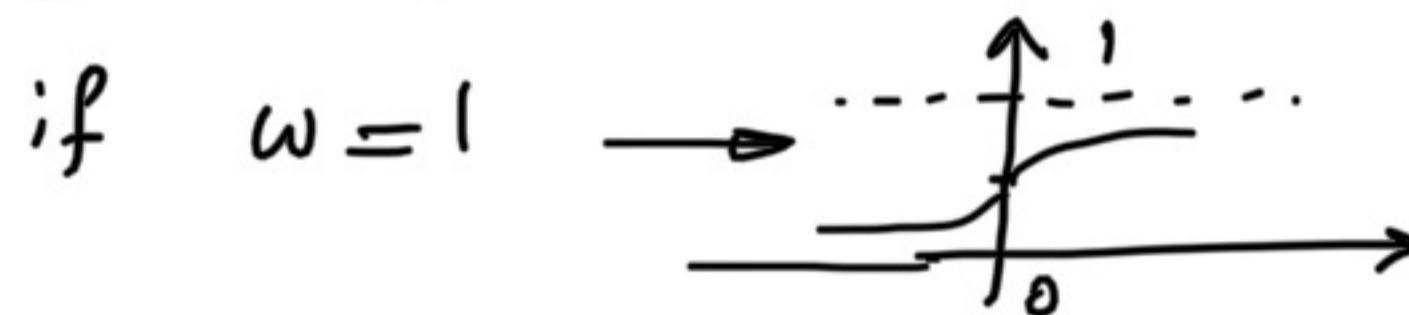
Some
✓
Sometimes, weights in NN are too high that other parts of the model does not train at all.

the dropout strategy randomly turns off some nodes during the training

dropout (0.2) → 0.2 = probability that each node will be turned off.

Regularization

example : $\hat{y} = \sigma(w_1x_1 + w_2x_2 + b)$ prediction



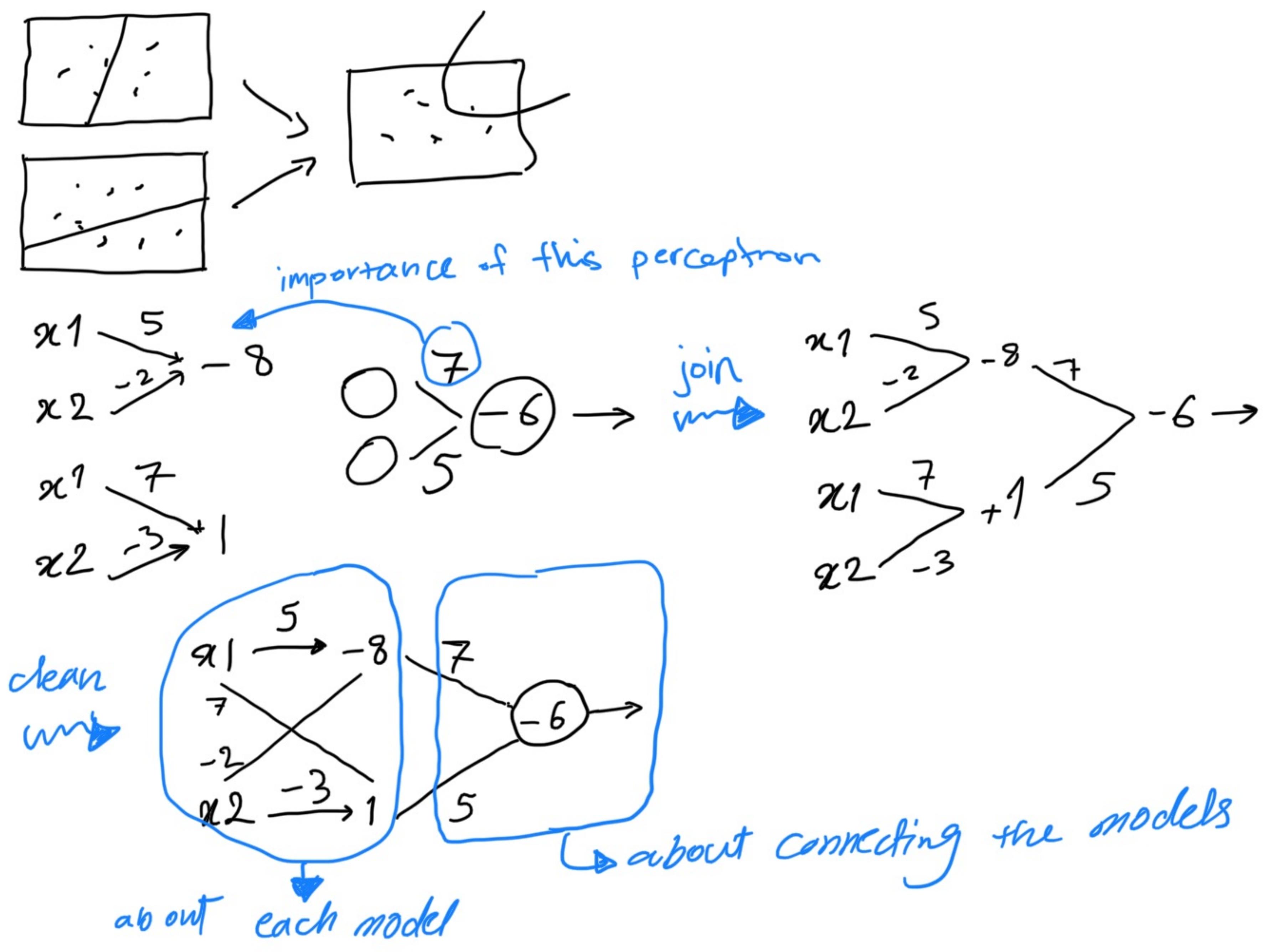
solution

In regularization, we want to penalize large weights.

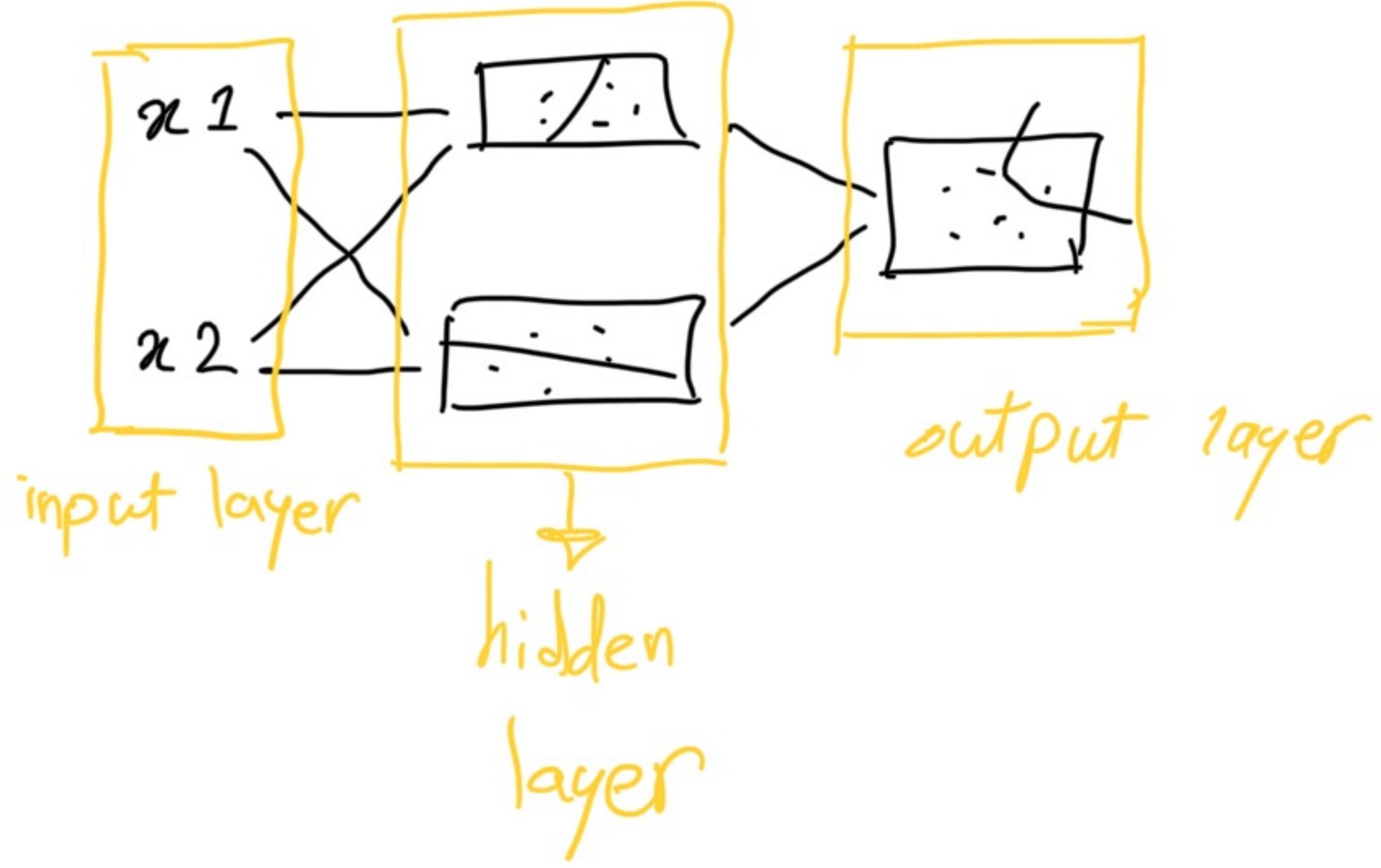
good for
feature selection \leftarrow Error + $\lambda(|w_1| + \dots + |w_n|)$ l1 - regularization

usually good \leftarrow Error + $\lambda(w_1^2 + \dots + w_n^2)$ l2 - regularization
for training models

Neural Network Architecture :



Multiple layers:

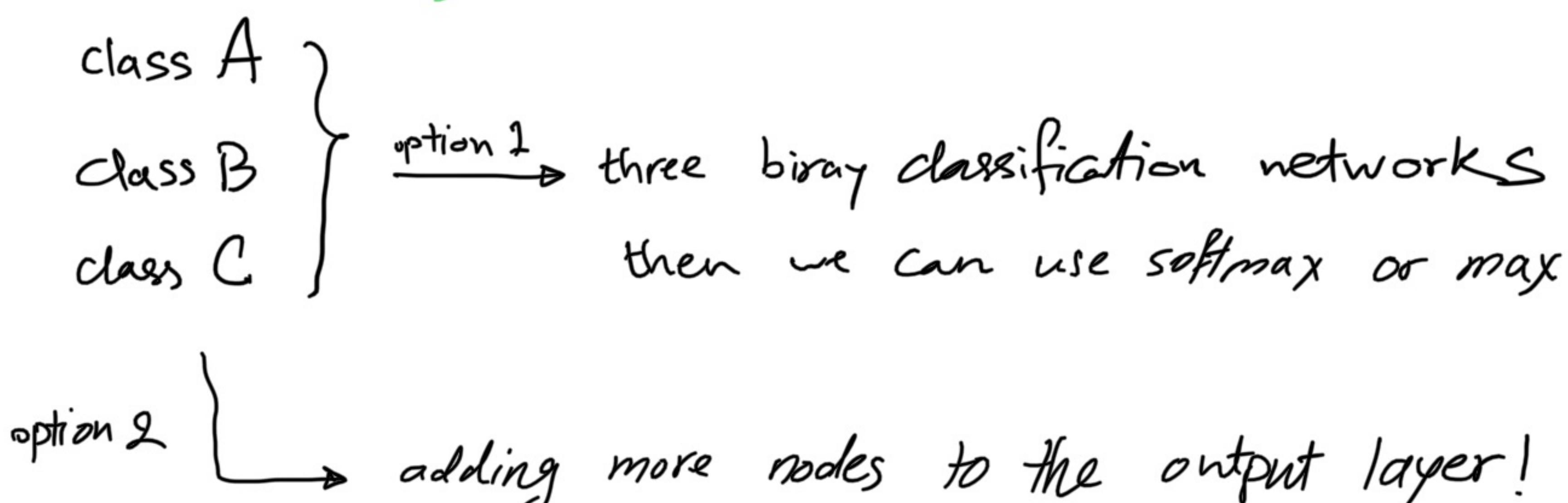


increasing number of input nodes to $n \rightarrow n$ -dim data

Deep Neural Networks:

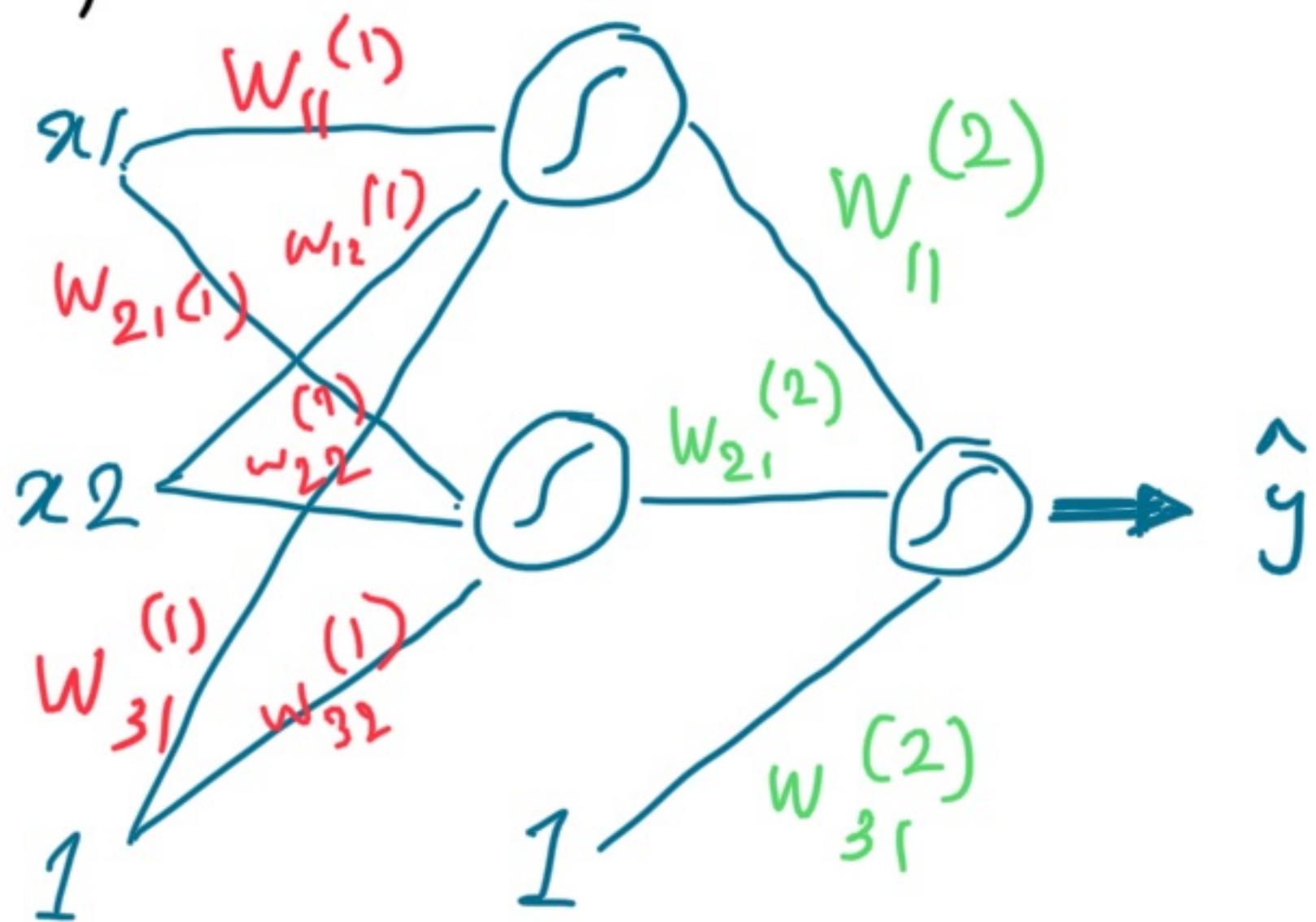


Multiclass classification:



Feed forward

The process nn use to turn input into an output.



$$\sigma \begin{pmatrix} w_{11}^{(2)} \\ w_{21}^{(2)} \\ w_{31}^{(2)} \end{pmatrix} = \sigma \begin{pmatrix} w_{11}^{(1)} & \dots & w_{12}^{(1)} \\ \vdots & & \vdots \\ w_{31}^{(1)} & \dots & w_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \hat{y}$$

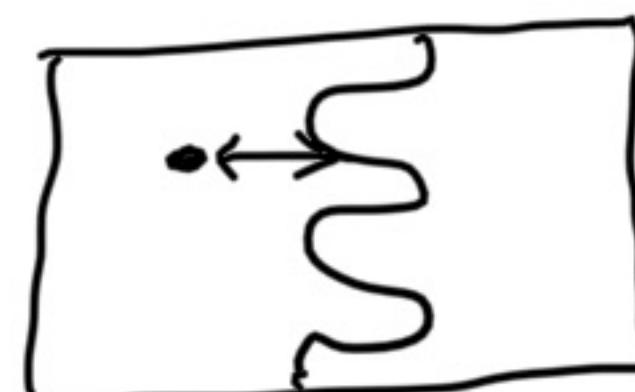
↓
input vector

simply put: $\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)} (\mathbf{x})$

↓
applied to

Error function

Similar to single perceptron.



Back propagation

do feedforward and get the error
go at the opposite direction.

increase the weights that come from better
classification and decrease ones that are worse.

more formally:

$$w_{ij}^{(k)} \leftarrow w_{ij}^{(k)} - \alpha \frac{\partial E}{\partial w_{ij}^{(k)}}$$

Some reminders: chain rule

$$A = f(x)$$

$$B = g \circ f(x)$$

$$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A} \frac{\partial A}{\partial x}$$

$$\sigma'(x)$$

$$= g'(x)(1 - g(x))$$

we now have all the ingredients to train
neural networks!

Sentiment Analysis

classify imdb reviews as POS or NEG.

Project 1:

```
from collections import Counter  
positives = Counter()  
positives[token] += 1 → used like a dict()  
positives.most_common()
```

Project 2:

allocating memory is expensive. It's better to allocate less frequently. `np.zeros((1, no-s))`

Project 3:

- it's useful to monitor "correct so far" samples as the model trains.
- lowering the learning rate might be useful if the network is not learning anything.

Project 4:

- Weight initialization strategy matters!
- one strategy: `np.random.normal`
- a rule of thumb:
 - it's good practice to start weights in range $[-y, y]$ where $y = 1/\sqrt{n}$.
number of inputs to a given layer
- noise versus signal in NN;
value of the inputs of the network, affects weights a lot. If one of them is 18, and the other is 2, the one with value=18 will dominate highly.
- when we tokenize the data, we should look out for meaningless stuff such as punctuation and common / filler words. Because they are not contributing to the prediction.
i.e., noise

Project 5:

- how to improve efficiency of computation?
i.e., what is wasteful in the network?
- ① long input vector: if many of them are zero, we're doing a useless mat mul.
 - ② if we have "1" in the input vector, we should not do mult.

Project 6:

if we strategically reduce vocabulary size, we may improve the accuracy.

Setting hyperparameters

Iterations

- Increase if both train & dev losses are decreasing.
- Stop if dev loss has saturated.
- Decrease if dev loss starts increasing.

Learning Rate

- Increase if train loss is not changing or decreasing fast enough.
- Decrease if train loss keeps fluctuating or starts increasing.

Hidden Nodes

- Increasing it makes harder and longer for the model to train.
- Increase when train loss saturates at a high value.
- Decrease if train loss decreases very slowly or keep fluctuating even after adjusting learning rate.

numpy intro

ndarray → support for math operations

scalars in numpy can be uint8, int8, uint16, ...

example:

{
 | s = np.array(5) → only holds a scalar
 | s.shape → () → 0-dim
 | x = s + 3

{
 | v = np.array([1, 2, 3])
 | v.shape → (3,)
 | v[1] → 2, v[1:] → [2, 3]

{
 | m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
 | m.shape → (3, 3)

reshaping:

v = np.array([1, 2, 3, 4]) (4,)

x = v.reshape(1, 4) (1, 4)

element-wise matrix operations:

$$2 + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2+1 & 2+2 \\ 2+3 & 2+4 \end{bmatrix}$$

x = np.multiply(some_array, 5) = x = some_array * 5

m * = 0 → no matter the dims of m,

all of the elements become zero.

m * m → element-wise multiplication

Matrix product:

np.matmul(A, B)

AB ≠ BA

Matrix transpose:

np.transpose(a) or a.T