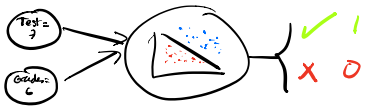
Equation of the hyperplane: $w_1 x_1 + w_2 x_2 + w_n x_n + b = 0$

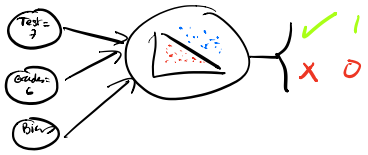
$[w][x] + [b] = 0$

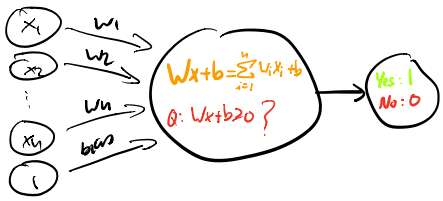
dim: $(1 \times n)$ $(n \cdot 1)$ $(1 \cdot 1)$ ↳ single output

Perceptrons:

Perceptron: Building block for our equation

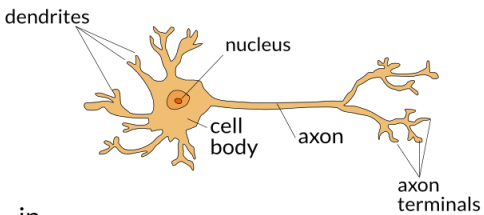


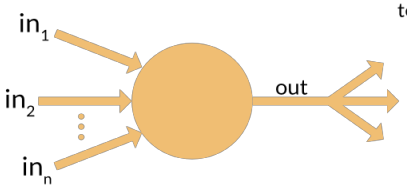




Why Neural Networks?

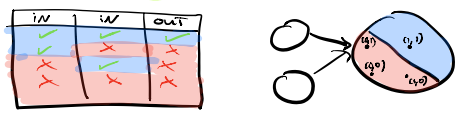
A perceptron is similar to a neuron in the brain



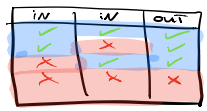


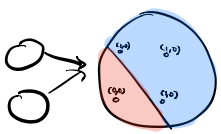
Perceptrons as logical operators

And operator



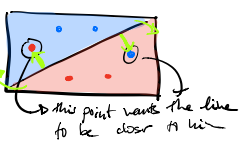
Or operator





Perceptron Trick

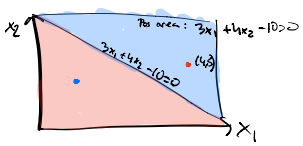
Starting at random and gradually improving



Line: $3x_1 + 4x_2 - 10 = 0$

Point $(4, 5)$, $1/\text{learning rate}$

−	3	4	−10
	4	5	1
	−1	−1	−11



Line: $3x_1 + 4x_2 - 10 = 0$

Point $(4, 5)$, $1/\text{learning rate}$

Learning Rate: 0.1

−	3	4	−10
	0.4	0.5	0.1
	2.6	3.5	−10.1

Makes it come closer but not all the way

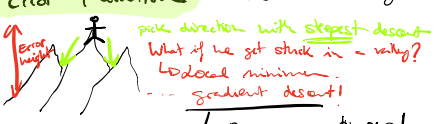
↳ SMALL STEPS

Perceptron Algorithm

1. Start with random weights: $w_1 \dots w_n, b$
2. For every misclassified point $(x_1 \dots x_n)$:
   - 2.1 If prediction = 0:
     - For $i = 1 \dots n$ (learning rate)
       - Change w_i to $w_i + \alpha x_i$
     - Change b to $b + \alpha$
   - 2.2 If prediction = 1:
     - For $i = 1 \dots n$ (learning rate)
       - Change w_i to $w_i - \alpha x_i$
     - Change b to $b - \alpha$
3. Repeat until small error or # iterations is reached

Error Function

Tells us how far away we are from the solution



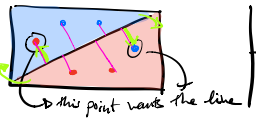
pick direction with steepest descent

What if we get stuck in a valley?

↳ local minima

... gradient descent!

↳ Requires continuous predictions

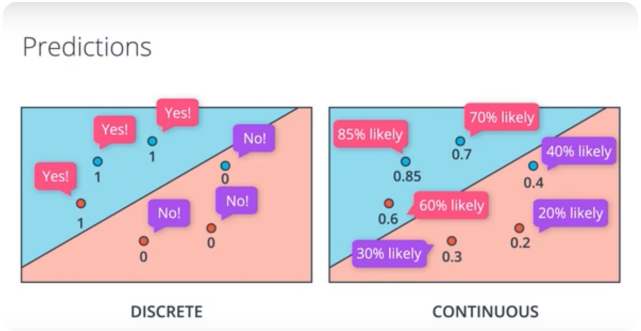


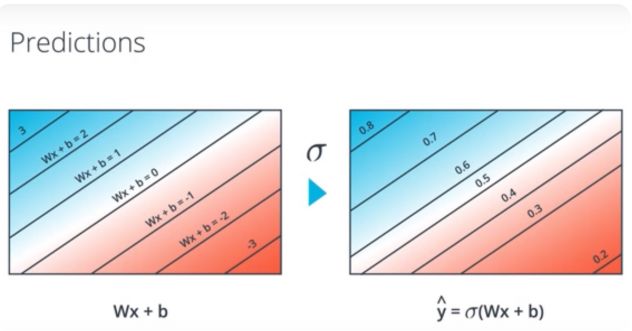
How do we tell the computer how far it is from a perfect solution?

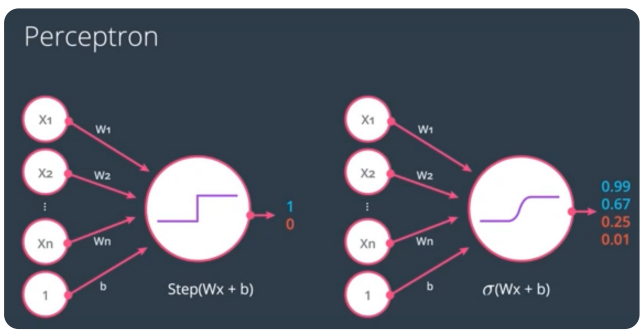
↳ count mistakes (or distance to line) due to small steps

Discrete vs Continuous

In order to do gradient descent, we need to have a continuous output from the perceptron.
We then need to use a different activation function (like σ) in order to transform the output to continuous.
The sigmoid is also really useful to calculate the probabilities since all outputs range from $[0, 1]$ ⇝ Same requirements for probability.







Softmax

Classification Problem

P(gift) = 0.8

P(no gift) = 0.2

Score(gift) = linear Funct

P(gift) = ∝ Score

But what if we had more options? Ex: Is it a cat? Is it a dog? Is it a bear?

P(duck) = 0.67 (2) P(beaver) = 0.24 (1) P(walrus) = 0.09 (0)

They all should add to 1

(2, 1, 0) → Normalize $\frac{2}{2+1+0} = \frac{2}{3}$, $\frac{1}{2+1+0} = \frac{1}{3}$, $\frac{0}{2+1+0} = 0$ → Problem: What happens if scores are negative? Doesn't work!

We first need to turn all scores into positive! → e^x

(2, 1, 0) → e^x → (7.39, 2.72, 1) → Softmax → $\frac{e^2}{e^2+e^1+e^0} = 0.67$, $\frac{e^1}{e^2+e^1+e^0} = 0.24$, $\frac{e^0}{e^2+e^1+e^0} = 0.09$ adds to 1.

Softmax definition

$$P(\text{class } i) = \frac{e^{z_i}}{e^{z_1} + \dots + e^{z_n}}$$

One Hot encoding for categorical data

Animal	Duck?	Beaver?	Walrus?
Duck	1	0	0
Beaver	0	1	0
Walrus	0	0	1

Maximum likelihood and Maximizing Probability

Which model is more accurate? The one that gives the highest probabilities to the events

Maximizing the probability ∝ minimizing error — We calculate the error

If we had many numbers, product scales really bad. Sums are better in changing over time

We can then use logarithms since $\log(ab) = \log(a) + \log(b)$

Cross Entropy

$0.6 \cdot 0.2 \cdot 0.1 \cdot 0.7 \rightarrow \ln(0.6) + \ln(0.2) + \ln(0.1) + \ln(0.7)$] they are all negative numbers

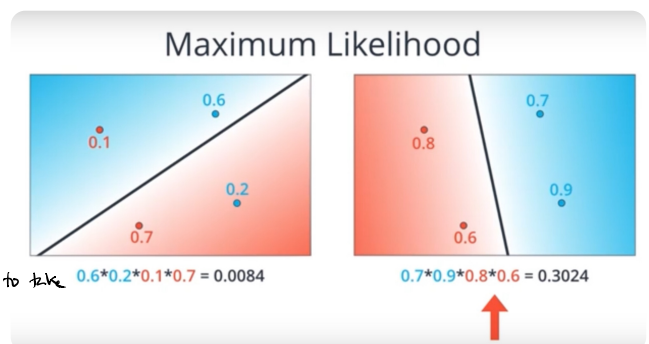
We then need to take the negatives

A good model will give us a low cross entropy (high probabilities will give smaller numbers)

The misclassified points will be the biggest indicators of high entropy

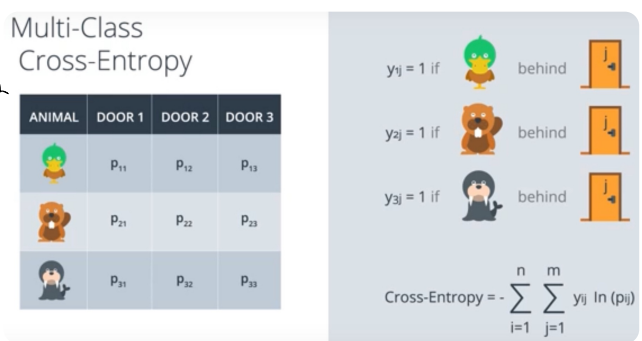
∝ errors at each point.

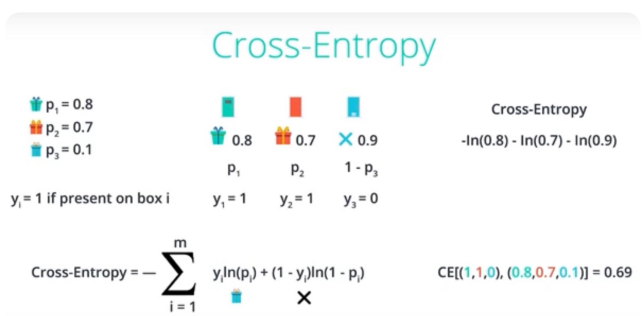
Maximizing probability ∝ Minimizing cross entropy



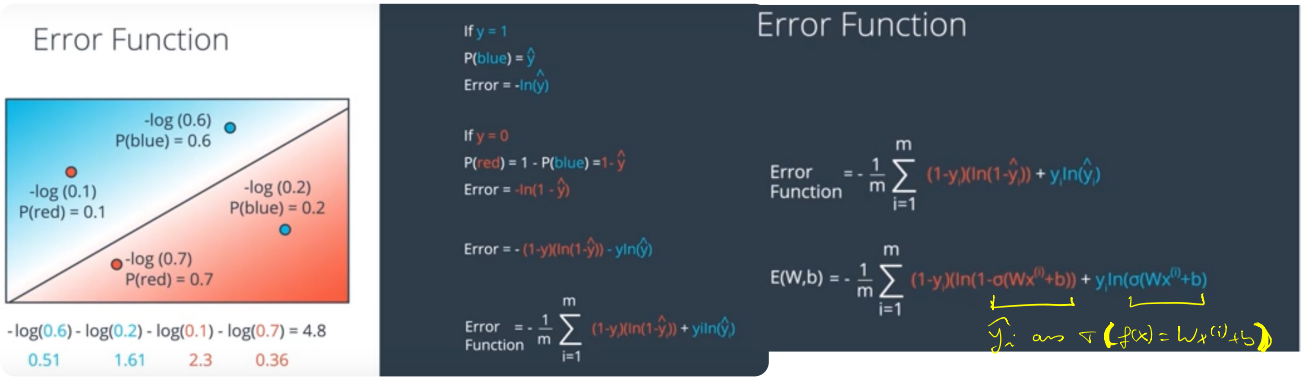
Multiclass Cross Entropy

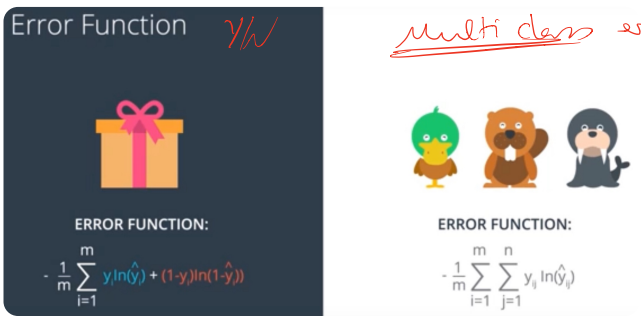
Essentially a generalization of the cross entropy of several classes (beyond Yes p_i no $(1-p_i)$)





Error Function



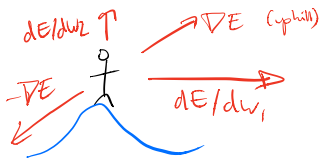


Multi class error

ERROR FUNCTION:

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

Gradient Descent



$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \dots \frac{\partial E}{\partial w_n}, \frac{\partial E}{\partial b}\right)$$

$\alpha = 0.1$ (learning rate)

$$w_i' = w_i - \alpha \cdot \frac{\partial E}{\partial w_i}$$

$$b' = b - \alpha \cdot \frac{\partial E}{\partial b}$$

$$\hat{y} = \sigma(W'x + b')$$ ← Better than W

Logistic Regression Algorithm

1. Start with random weights: $w_1, \dots, w_n, b$
2. For every point $(x_1, \dots, x_n)$:
   - 2.1. For $i = 1 \dots n$
     - 2.2.1 Update $w_i' \leftarrow w_i - \alpha(\hat{y} - y)x_i$
     - 2.2.2 Update $b' \leftarrow b - \alpha(\hat{y} - y)$
3. Repeat until error is small or ~ fixed # of times (epochs)

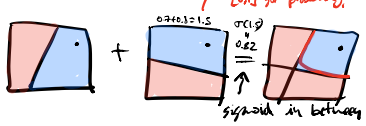
Very similar to the perceptron!

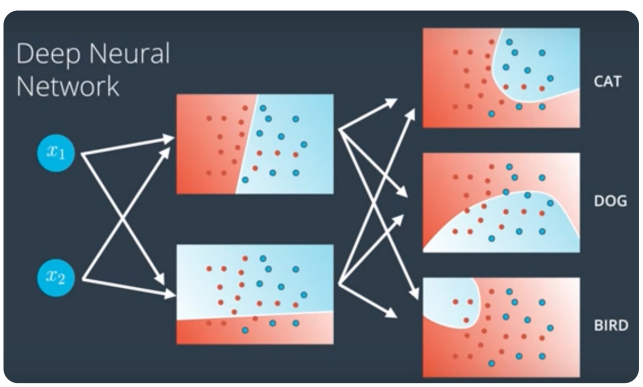
Non Linear Models

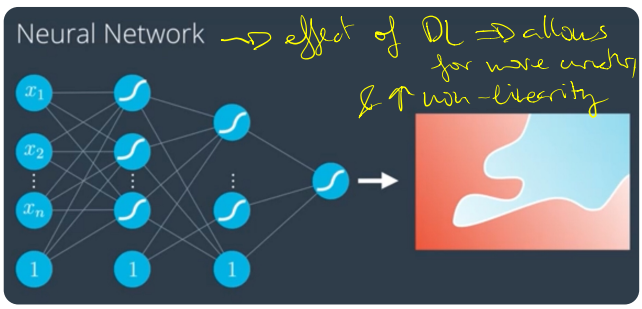
This can be achieved via Neural Networks:

Neural Network Architecture

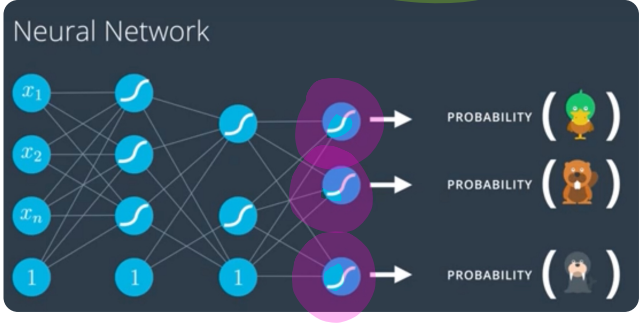
Non-linear models can be achieved via the combination of two non linear models







Multi Class Classification



No need to create a separate NN for each animal, we just need to add more nodes in the output layer

Feed Forward

Process in which neural networks turn an input into an output

$\hat{y} = \sigma(Wx + b)$

How to do it for multilayer perceptrons?
Do same error function, with more complicated boundary

Back propagation

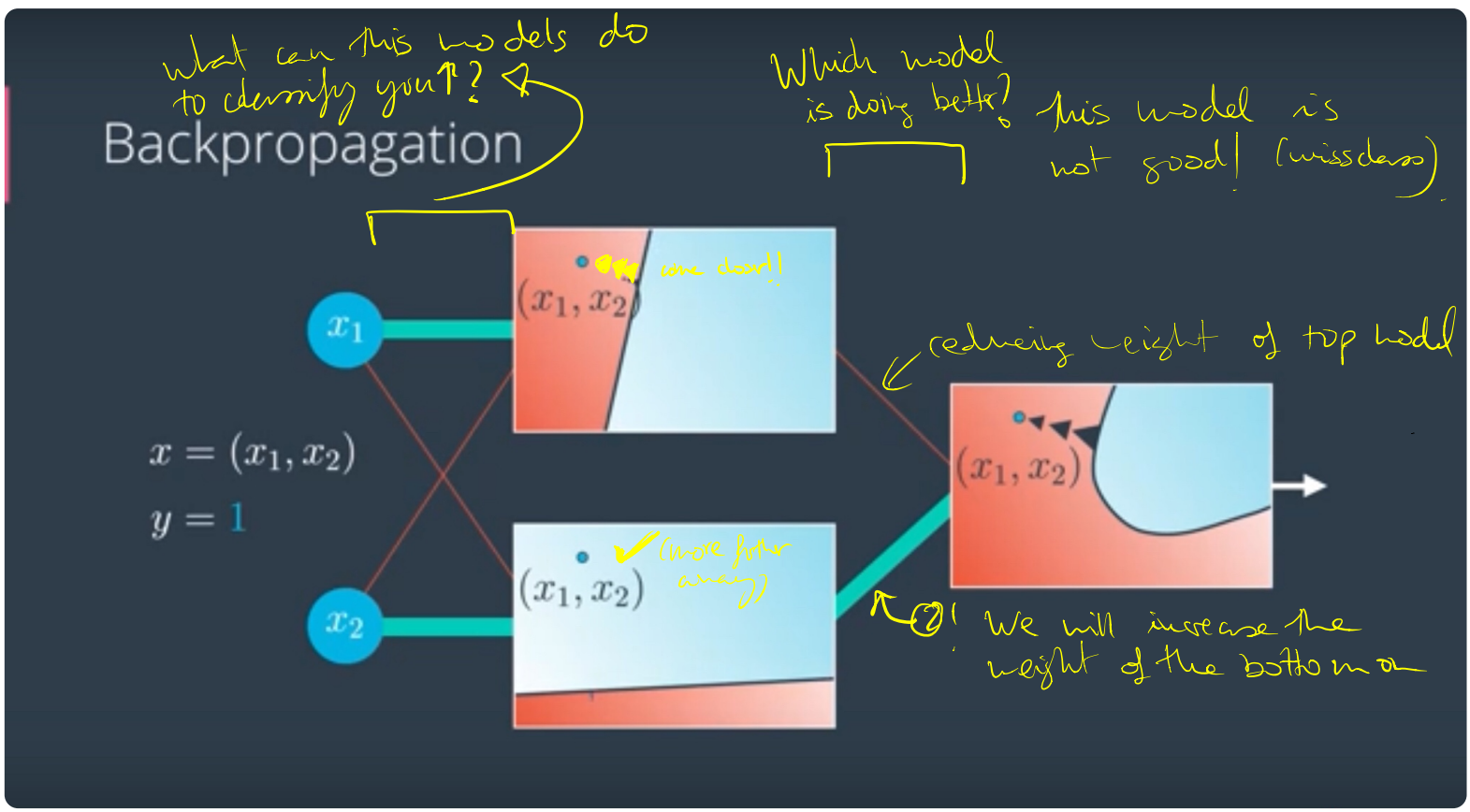
→ Feed forward: Perceptron plots a point and returns the probability of it being blue

→ Back propagation asks the point: how would you like to change the model?

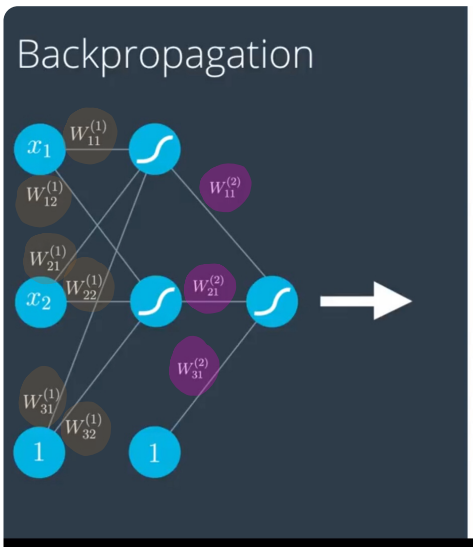
Closer / Far away

How can we do it for Multilayer Perceptron?

→ We obtain new weights → They get closer to the point



Backpropagation Math



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

$$W^{(1)} = \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix}$$

Bias

matrix notation for convenience

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial W_{11}^{(1)}} & \frac{\partial E}{\partial W_{12}^{(1)}} & \frac{\partial E}{\partial W_{11}^{(2)}} \\ \frac{\partial E}{\partial W_{21}^{(1)}} & \frac{\partial E}{\partial W_{22}^{(1)}} & \frac{\partial E}{\partial W_{21}^{(2)}} \\ \frac{\partial E}{\partial W_{31}^{(1)}} & \frac{\partial E}{\partial W_{32}^{(1)}} & \frac{\partial E}{\partial W_{31}^{(2)}} \end{pmatrix}$$

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

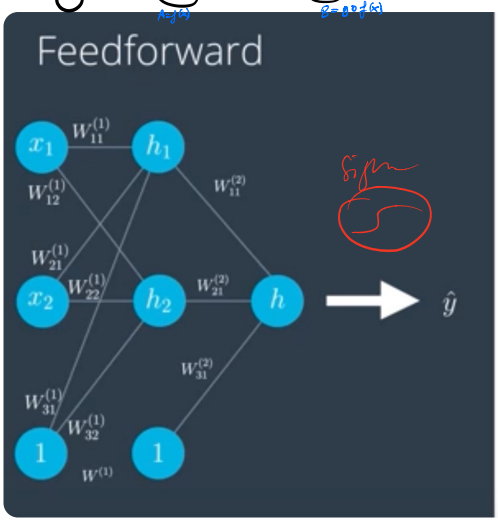
updating each weight

so it will give us new updated weight w_i_j super k prime.

Chain Rule

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

"When composing functions (ex: feedforward), derivatives just multiply"



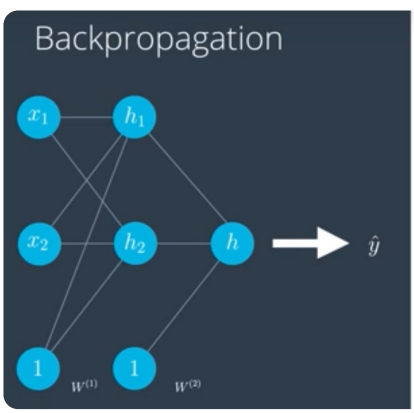
$$h_1 = W_{11}^{(1)} x_1 + W_{21}^{(1)} x_2 + W_{31}^{(1)}$$

$$h_2 = W_{12}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{32}^{(1)}$$

$$h = W_{11}^{(2)} \sigma(h_1) + W_{21}^{(2)} \sigma(h_2) + W_{31}^{(2)}$$

$$\hat{y} = \sigma(h)$$

$\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)} x$ (condensed notation)

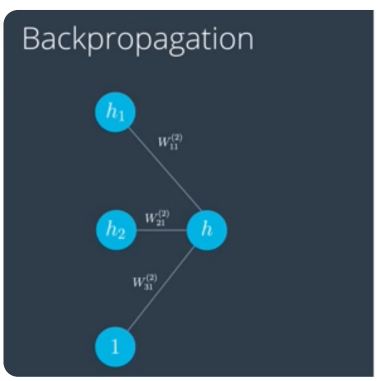


$$E(W) = -\frac{1}{m} \sum_{i=1}^{m} y_i ln(\hat{y}_i) + (1 - y_i) ln(1 - \hat{y}_i)$$

$$E(W) = E(W_{11}^{(1)}, W_{12}^{(1)}, ..., W_{31}^{(2)})$$

$$\nabla E = (\frac{\partial E}{\partial W_{11}^{(1)}}, ..., \frac{\partial E}{\partial W_{31}^{(2)}})$$

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}$$



$$h = W_{11}^{(2)} \sigma(h_1) + W_{21}^{(2)} \sigma(h_2) + W_{31}^{(2)}$$

$$\frac{\partial h}{\partial h_1} = W_{11}^{(2)} \sigma(h_1)[1 - \sigma(h_1)]$$

derivative of σ respect to h_1 $\left(\frac{\partial \sigma}{\partial h_1}\right)$

$$\sigma'(x) = \frac{\partial}{\partial x} \frac{1}{1 + e^{-x}}$$

$$= \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} = \sigma(x) \cdot (1 - \sigma(x))$$

Training Optimization

Underfitting: killing Godzilla w/ a spoon

Overfitting: killing a fly w/ a bazooka.

↳ high variance. Does not generalize well

Train / Test ≈ Study / Exam

Early stopping

→ When testing error starts increasing.

Early quitting on epochs!!

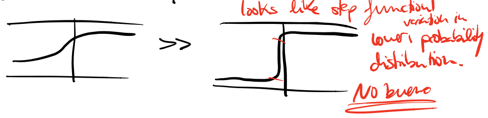
Subtle Overfitting

$10x_1 + 10x_2$ gives less error than $x_1 + x_2$

due to $\sigma(20) = 0.99....$ vs $\sigma(2) = 0.8$

$\sigma(-20) = 0.....01$ vs $\sigma(-2) = 0.12$

But is this better?



Solution: Regularization

LARGE COEFFICIENTS ⟶ OVERFITTING

PENALIZE LARGE WEIGHTS

$$(w_1, \ldots, w_n)$$

L1 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \lambda(|w_1| + \ldots + |w_n|)$

L2 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \lambda(w_1^2 + \ldots + w_n^2)$

L1 good for feature selection

Sparsity (1, 0, 0, 1, 0)

Sparsity (0.5, 0.3, -0.2, 0.4, 0.1)

→ L2 usually better for training models

$0.5^2 + 0.5^2 = 0.5$ vs $1^2 + 0^2 = 1$

L2 prefers smaller coef.

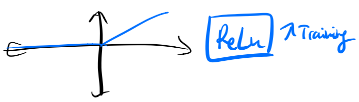
Dropout

Sometimes one part of the network has very large weights and overshadows the other part, which doesn't play any role at all...

In order to solve this we will randomly turn off some of the nodes in the epoch. ⟹ Better generalization.

Vanishing gradient

Occurs due to the slope of the sigmoid



Batch vs Stochastic Gradient Descent

Epoch = Steps

Stochastic gradient descent → smaller sub-steps

In order to do this, we split the data into several batches

Adaptive learning Rate

Better if it decreases as we get closer to solution

Momentum

A way to get over the humps (local minima)

Constant β [0,1]

Step (n) + β step(n-1) + β^2 step(n-2) + ...