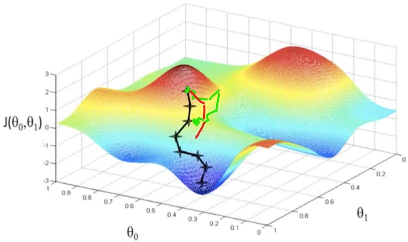
Last session

- Intro.
- Machine Learning.
- Regression and classification.
- Hyper-parameters.
- Overfitting and datasets.
- Perceptron.
- Gradient descent.

Outline

Gradient Descent, GD



- Random initialization.
- Forward pass.
- Error estimation.
- Gradient computation.
- Backward pass.

Q: remember the notion of "taking a step in the direction of steepest descent"?
A: The direction of the step is computed considering all of the parameters at once, i.e., we compute first the gradient (all partial derivatives), and then update all the weights.

GD, pseudocode

Algorithm 1 Gradient Descent

Initialize Ω randomly.
for each epoch **do**
  for each sample **do**
    $\hat{y}_i = f(x_i; \Omega)$
    $E_i = l(y_i, \hat{y}_i)$
    $\Omega = \Omega - \eta \nabla_\Omega E_i$
  end for
end for

where, $\Omega = \{\omega_i\}$, $l(\cdot)$ is a loss function, and $\nabla_\Omega E_i$ is the gradient of the error with respect to the set Ω of parameters.

Q: can you imagine a drawback of this approach?

A: The model will end up being *biased* towards the last seen

Stochastic GD

Stochastic GD (SGD) tries to compensate for the bias of the last seen training samples.

Each epoch, randomly shuffle the order of samples.

Algorithm 2 Stochastic Gradient Descent

Initialize Ω randomly.
for each epoch **do**
    $\{X, y\} = shuffle(\{X, y\})$
    for each sample **do**
        $\hat{y}_i = f(x_i; \Omega)$
        $E_i = l(y_i, \hat{y}_i)$
        $\Omega = \Omega - \eta \nabla_\Omega E_i$
    end for
end for

Batch GD

Algorithm 3 Batch Gradient Descent

```
Initialize Ω randomly.
Define a number of batches.
for each epoch do
    {X, y} = shuffle({X, y})
    for each batch do
        {X_B, y_B} = next N training pairs
        ŷ_B = f(X_B; Ω)
        E_B = (1/N) Σ_{n=1}^{N} l(y_{B_n}, ŷ_{B_n})
        Ω = Ω − η ∇_Ω E_B
    end for
end for
```

1. Initialize Ω randomly.
2. Define a number of batches.
3. **for each** epoch **do**
4. $\quad \{X, y\} = shuffle(\{X, y\})$
5. $\quad$ **for each** batch **do**
6. $\quad\quad \{X_B, y_B\} =$ next N training pairs
7. $\quad\quad \hat{y}_B = f(X_B; \Omega)$
8. $\quad\quad E_B = \frac{1}{N} \sum_{n=1}^{N} l(y_{B_n}, \hat{y}_{B_n})$
9. $\quad\quad \Omega = \Omega - \eta \nabla_\Omega E_B$
10. $\quad$ **end for**
11. **end for**

Q: What advantage would it have to use BGD instead of SGD?

BGD

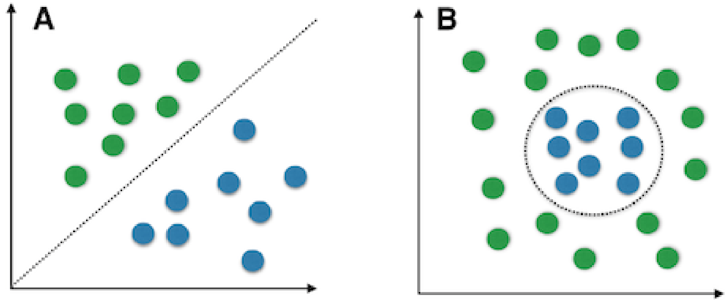
a.k.a., mini-batch gradient descent.

- Approximates E with the average error of a batch of samples.
- Fewer updates, i.e., faster optimization process.

- Batch size $bs = 1$ boils down to regular GD.
- Common batch sizes: $16, 32, 64, 128, 256$.

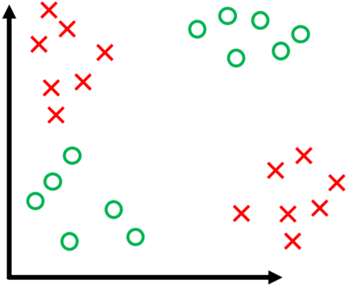
Outline

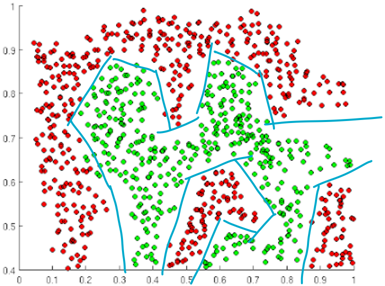
Linearity



Non-linear Separability

Most real-world problems are non-linearly separable.





Q: How do we do in these cases in machine learning?

Non-linear transformations

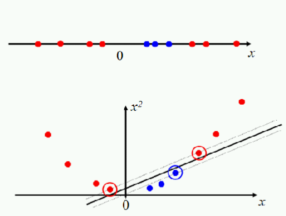
$x_4 = x_1^2 + x_2^2 + x_1 x_2$

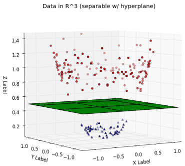
$x_5 = x_1^3 + x_2^3 + x_3 x_4$

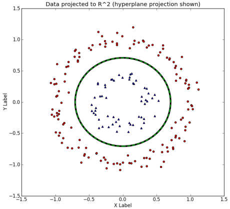
Feature engineering.

Examples:

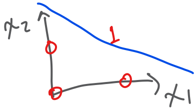
- New feature, $x_2 = (x_1)^2$.
- New feature, $x_3 = x_2 * x_1$.
- Feature selection: use only a subset of features.







Example I



$\hat{y} = W^T X + b$

$b = 1 = x_0$

$w_0 =$

Logical AND: can be solved with a single perceptron.

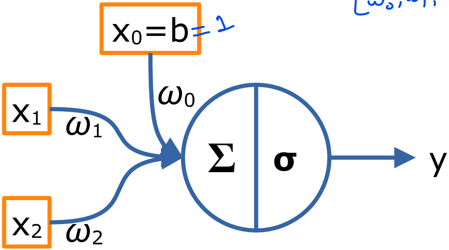
$\hat{y} = W^T X \rightarrow [x_0, x_1, x_2]$

$[w_0, w_1, w_2]$

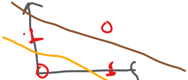
x_0	x_1	x_2	y
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Solved using:

ω_0	ω_1	ω_2
-1.5	1	1



Example II

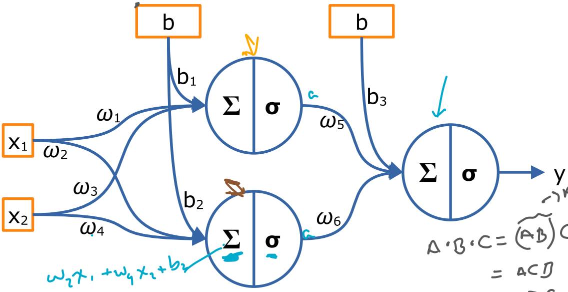


$s = w^T x = \sum_i w_i x_i$

$a = \sigma(s) = \frac{1}{1+e^{-s}}$

Logical XOR: not solved with a single perceptron. Let's cascade non-linear activation functions: multi-layer perceptron (MLP).

x_0	x_1	x_2	y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



$\rightarrow AB = D$

$A \cdot B \cdot C = (AB)C$

$= ACB$

$= DC$

Solved using:

b_1	b_2	b_3	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
-10	30	-30	20	-20	20	20	-20	20

Q: What if we omit the non-linearity activation functions?

MLP

Consecutive linear operations are equivalent to a single linear operations, i.e., DL is enable by the use of non-linear activation functions.

- We end up with: input, hidden, and output layers.
- Intermediate representations correspond feature engineering.
- However, features are learned rather than engineered.
- End-to-end process.
- Information abstraction increases with depth.
- Inspired on human brain?

Nonlinearities

In general, the more difficult the problem looks, the more chances are it is non-linearly separable. Therefore, the deeper the model must be.

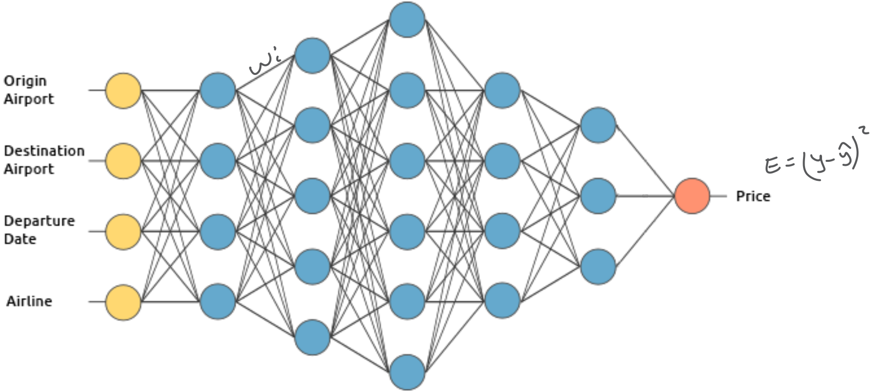
Q: But how do we do it?

Outline

GD and depth

$\frac{\partial E}{\partial w_i}$

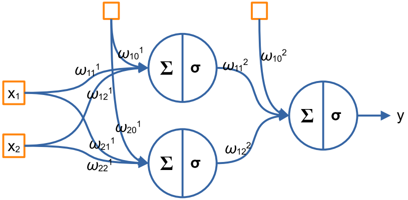
Q: How do we perform GD on deep networks?



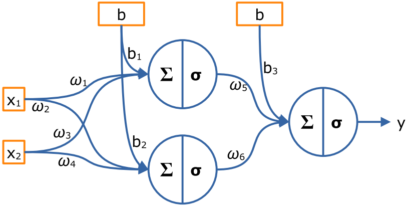
A: Use backpropagation (*backprop*): gradient descent + chain rule.

Notation I

Common notation



Simplified notation



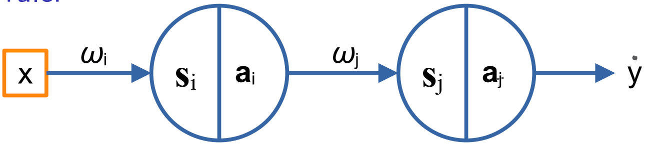
Notation II

Let's also use:

- $s = \sum_i \omega_i x_i + b$, linear combination.
- $a = \sigma(s)$, non-linear activation.

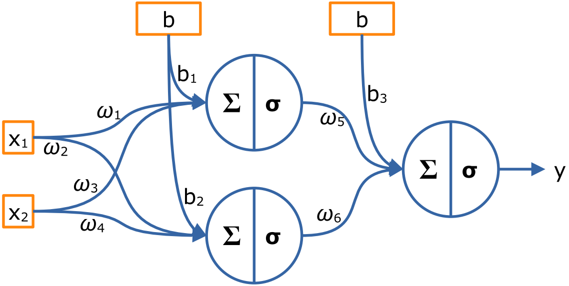
$E = (y - \hat{y})$

Chain rule:



$$\frac{\partial E}{\partial \omega_i} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial a_j}\frac{\partial a_j}{\partial s_j}\frac{\partial s_j}{\partial a_i}\frac{\partial a_i}{\partial s_i}\frac{\partial s_i}{\partial \omega_i}.$$

Example



1 training pair $\{\mathbf{x} = [0.05, 0.1], \qquad y = 1\}$.

Initialize with:

ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	b_1	b_2	b_3
0.15	0.20	0.25	0.30	0.50	0.55	0.35	0.35	0.60

And let's use $\eta = 0.1$.

Example cont.

$t \to y(t)$

$$\hat{y}(0) = 0.773.$$

After one update:

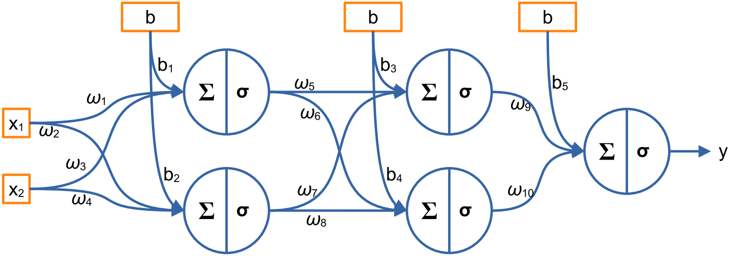
ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
0.15002	0.20003	0.25005	0.30005	0.50237	0.55238

b_1	b_2	b_3
0.35048	0.35053	0.60398

$$\hat{y}(1) = 0.7742.$$

Notice: The impact of backprop is proportional to the depth of the layer: weights in shallow layers are update more softly with respect to those in deeper layers.

Multiple connections



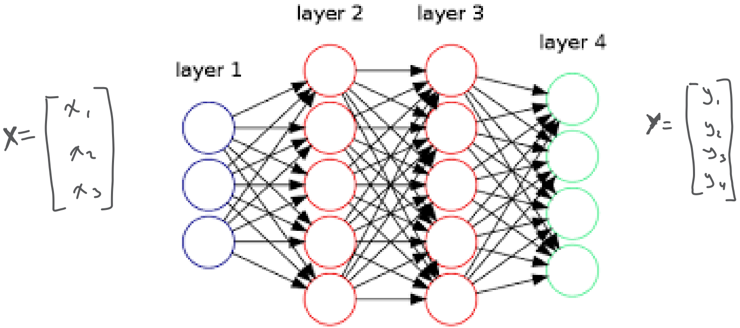
$$\frac{\partial E}{\partial \omega_5} = \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_3}\frac{\partial a_3}{\partial s_3}\frac{\partial s_3}{\partial \omega_5}.$$

$$\frac{\partial E}{\partial \omega_1} = \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_3}\frac{\partial a_3}{\partial s_3}\frac{\partial s_3}{\partial a_1}\frac{\partial a_1}{\partial s_1}\frac{\partial s_1}{\partial \omega_1} + \frac{\partial E}{\partial a_5}\frac{\partial a_5}{\partial s_5}\frac{\partial s_5}{\partial a_4}\frac{\partial a_4}{\partial s_4}\frac{\partial s_4}{\partial a_1}\frac{\partial a_1}{\partial s_1}\frac{\partial s_1}{\partial \omega_1}.$$

Outline

Multi-variate regression

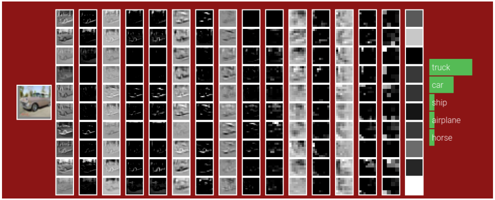
- One output perceptron works well for uni-variate regression, i.e., $\hat{y}$ is a scalar.
- More perceptrons can be used for a multi-variate problem, i.e., $\hat{\mathbf{y}}$ is a vector.



Multi-class classification

- One output perceptron works well for binary classification problems, i.e., $\hat{y}$ is a scalar indicating the probability of the input belonging to the positive class.
- More perceptrons can be used for a multi-class classification problem, i.e., $\hat{\mathbf{y}}$ is a vector indicating the probability of belonging to each possible class.
- In this case, the ground-truth is a one-hot encoding vector. E.g., $\mathbf{y} = [0, 0, 1, 0, 0]$.

$\hat{y} = [0.001, 0.002, 0.96, 0.01, 0.02]$

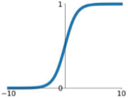


Outline

Comparison

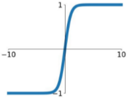
Sigmoid

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



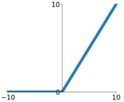
tanh

$\tanh(x)$



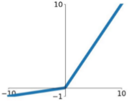
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

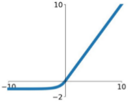


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Linear

$$a = \sum_{n=0}^{N} x_n \omega_n.$$

Derivative:

$$\frac{\partial a}{\partial \omega_i} = x_i.$$

Sigmoid

$$a = \sigma(s) = \frac{1}{1 + e^{-s}}.$$

Derivative:

$$\sigma'(s) = \sigma(s)(1 - \sigma(s)).$$

Tanh

$$a = \frac{e^s - e^{-s}}{e^s + e^{-s}}.$$

Derivative:

$$a' = 1 - a^2.$$

ReLU

Logistic functions suffer of the so-called *vanishing gradient* issue. Rectified Linear Units (ReLU) are an alternative:

$$a = \begin{cases} 0, & s < 0, \\ s, & s \geq 0. \end{cases}$$

Derivative:

$$a' = \begin{cases} 0, & s < 0, \\ 1, & s \geq 0. \end{cases}$$

They are also more efficient computationally speaking.

Leaky ReLU

$$a = \begin{cases} \alpha s, & s < 0, \\ s, & s \geq 0, \end{cases}$$

with $0 < \alpha < 1$.

Derivative:

$$a' = \begin{cases} \alpha, & s < 0, \\ 1, & s \geq 0. \end{cases}$$

ELU: Exponential Linear Unit

$$a = \begin{cases} \alpha(e^s - 1), & s < 0, \\ s, & s \geq 0, \end{cases}$$

with $0 < \alpha < 1$.

Derivative:

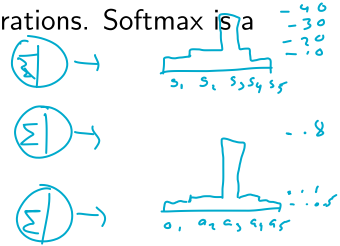
$$a' = \begin{cases} a + \alpha, & s < 0, \\ 1, & s \geq 0. \end{cases}$$

$y = [0\ 0\ 0\ 1\ 0\ 0]$ $\hat{y} = [0.01, 0.01, 0.01, 0.95, 0.01, 0.01]$

Softmax

All previous functions are element-wise operations. Softmax is a vector normalizer.

$$a_i = \frac{e^{s_i}}{\sum_j e^{s_j}}.$$



Derivative:

$$a_i' = a_i(1 - a_i).$$

$\sum_i a_i = 1$

Used to exaggerate the most probable of the elements of the vector. Useful in output layers for multi-class classification problems.

Common use scenarios

Activation	Use
ReLU (or variants)	All hidden layers in all scenarios.
Sigmoid (tanh)	Output layer for binary classification.
Sigmoid	Output layer for regression with $0 \leq y \leq 1$.
Tanh	Output layer for regression with $-1 \leq y \leq 1$.
Linear	Output layer for unbounded regression.
Softmax	Output layer for multi-class classification.