Introduction to Basics in Neural Network

using python.

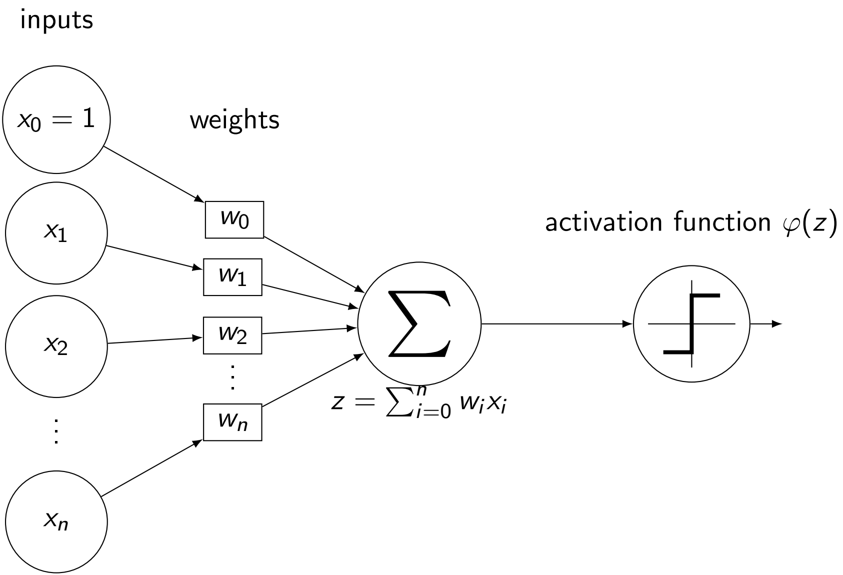
Iván Andrés Trujilllo Abella

Background classification problem

Fisher(1936) proposes the linear Discriminant, the problem consist in predict a binary outcome according to a set of features, for instance find the variables that could predict the bankruptcy.
The main objective is construct a $z = \mathbf{w^t x}$ score whose indicate the probability of belonging to a class.

Perceptron

Rossenblant(1958) We can define a vector input $\mathbf{x}$ and a vector of weights $\mathbf{w}$ and a activation function φ that take as input the inner product of both vectors defined previously $\varphi(\mathbf{w}^t\mathbf{x})$.


inputs
$x_0 = 1$
x_1
x_2
x_n
weights
w_0
w_1
w_2
w_n
$\sum$
$z = \sum_{i=0}^{n} w_i x_i$
activation function $\varphi(z)$

Activation function

$\varphi()$ could be defined as the sigmoid function.

$$\begin{aligned} p(y=1) &= \varphi(z) \\ p(y=0) &= 1-\varphi(z) \end{aligned} \tag{1}$$

Percepton works with a step function:

$$\varphi(z) = \begin{cases} -1 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

Returning a binary outcome.

What set of w values we must choose

Cost function

The cost function could be defined in a soft or a hard way.

$$J(w)_{hard} = \sum_{i=1}^{n} \max(-y_i \hat{y}_i, 0) \tag{2}$$

J only count the number of mismatches. However this function not is differentiable.

$$J(w)_{soft} = \sum_{i=1}^{n} \max(-y_i z_i, 0) \tag{3}$$

if $y_i z_i < 0$ then lost function > 0
if $y_i z_i > 0$ the lost function $= 0$.

Insights

The update of weights is according to the data bias or mistakes, however when the model match to the class then

$$\Delta w_i = (y_i - \hat{y}_i) = 0 \tag{4}$$

where y_i is the real observed data, and $\hat{y}_i$ is the predicted class.
when the $y_i = -1$ and $\hat{y} = 1$ then $\Delta w = -2$, in otherwise $y_i = 1$ and $\hat{y}_i = -1$ then $\Delta w = 2$. in summary:

$$\Delta w_i = \begin{cases} 0 & y_i = \hat{y}_i \\ -2 & y_i < \hat{y}_i \\ 2 & y_i > \hat{y}_i \end{cases}$$

insights

Then when there are mistakes

$$\varphi(w_{i+1}^t \mathbf{x_i}) = \varphi((w_i + \Delta w_i)^t \mathbf{x_i}) = y_i \tag{5}$$

This mean that weights for the vector of features of the sample i are update to predict the correct class.

$$w_{i+1} = w_i + \eta \Delta w_i \mathbf{x_i} \tag{6}$$

Perceptron algorithm

```
initialize w:
for each x in sample :
estimate y(x)
w = w + update(w)
```

LAB

Perceptron implementation from scratch

Perceptron from scratch(click here)

sklearn

It is open source library, integrated with scipy and numpy. It is one of the most popular machine learning library on Github.

- Classification (Neural networks Support Vector Machine)
- Decision trees
- Cluster
- Regression

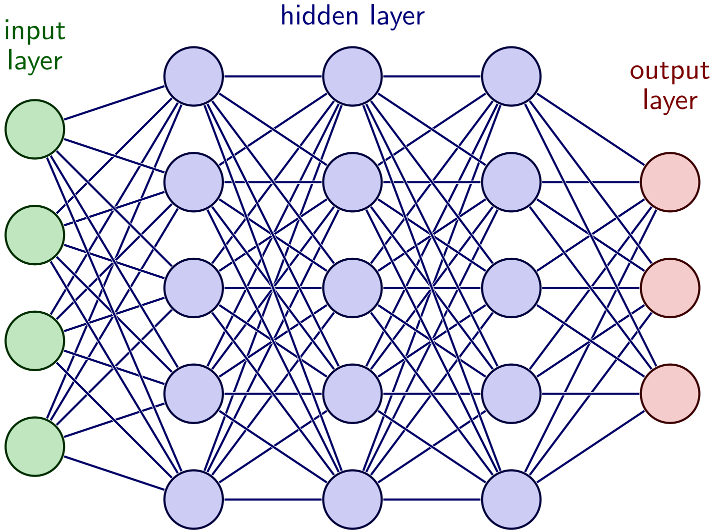
sklearn

```
from skelearn.linear_model import Perceptron
model = Perceptron(penalty=None , max_iter=1000, eta0=0.4,
    random_state=1)
model.fit(X,y)
model.score(X,y) # Print the number of matches
from sklearn.metrics import confusion_matrix
confusion_matrix(y, model.predict(X))
print(model.coef_)
```

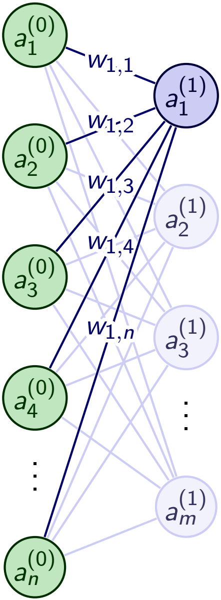
Add more layers!

Perceptron learning algorithm doesn't work

Perceptron could learn of mistakes, but is not desgined to handle more layers!



Source:Tikz



$$
\begin{aligned}
&= \sigma\left(w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right) \\
&= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)
\end{aligned}
$$

$$
\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ w_{2,0} & w_{2,1} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \ldots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]
$$

$$
a^{(1)} = \sigma\left(\mathbf{W}^{(0)}a^{(0)} + \mathbf{b}^{(0)}\right)
$$

Source: Tizk

Notation

$w^L_{j,k}$ weigth that connect the j nueron with k neuron in the L layer.

$w_{destination,origin}$

$w^L_{j,k}$ is the weight of k neuron in $L-1$ layer with j neuron at L layer.

b_j

b^L_j is the bias in L layer for j neuron.

a_j

a^L_j is the activation of the j neuron in L layer.

$$a^L_j = \sigma(z^L_j) = \sigma(\sum_k w^L_{j,k} a^{L-1}_k + b^L_j) \tag{7}$$

$$a_j^L = \sigma(z_j^L) = \sigma(\sum_k w_{j,k}^L a_k^{L-1} + b_j^L) \quad (8)$$

$\sum_k$ indicates that the activation of the j neuron rely on in all weights in the layer L.

$J()$ Cost function

$$J(a^L(z^L(w^L, a^{L-1}, b^L) \quad (9)$$

Matrix representation

$$\begin{bmatrix} z_1^L \\ z_2^L \\ \vdots \\ z_n^L \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,n} \end{bmatrix} \begin{bmatrix} a_1^{L-1} \\ a_2^{L-1} \\ \vdots \\ a_n^{L-1} \end{bmatrix} + \begin{bmatrix} b_1^L \\ b_2^L \\ \vdots \\ b_n^L \end{bmatrix} \tag{10}$$

Rememberthat $a_1^L = \sigma(z_1^L)$

$$\frac{\partial J}{\partial a_m^{L-1}} = \sum_i^n \frac{\partial z_i^L}{\partial a_m^{L-1}} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial J}{\partial a_i^L} \tag{11}$$

Forward

Example

A foward with a neural network with the following architecture $(3, 3, 2, 1)$.

$$
\begin{aligned}
z^2 &= \begin{pmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^2 \\ b_2^2 \\ b_3^3 \end{pmatrix} \\
&= \begin{pmatrix} \sum_{i=1}^{3}(w_{1i}^2 x_i) + b_1^2 \\ \sum_{i=1}^{3}(w_{2i}^2 x_i) + b_2^2 \\ \sum_{i=1}^{3}(w_{3i}^2 x_i) + b_3^2 \end{pmatrix} = \begin{pmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \end{pmatrix}
\end{aligned}
\tag{12}
$$

Forward

$$a^2 = \sigma\left(\begin{pmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \end{pmatrix}\right) = \begin{pmatrix} \sigma(z_1^2) \\ \sigma(z_2^2) \\ \sigma(z_3^2) \end{pmatrix} \tag{13}$$

Now we've already calculated the *first propagation*.

$$z^3 = \begin{pmatrix} w_{11}^3 & w_{12}^3 & w_{13}^3 \\ w_{21}^3 & w_{22}^3 & w_{23}^3 \end{pmatrix} \begin{pmatrix} \sigma(z_1^2) \\ \sigma(z_2^2) \\ \sigma(z_3^2) \end{pmatrix} + \begin{pmatrix} b_1^3 \\ b_2^3 \end{pmatrix} \tag{14}$$

Forward

$$z^3 = \begin{pmatrix} \sum_{i=1}^{3} \left(w_{1i}^3 \sigma(z_i^2) \right) + b_1^3 \\ \sum_{i=1}^{3} \left(w_{1i}^3 \sigma(z_i^2) \right) + b_2^3 \end{pmatrix} = \begin{pmatrix} z_1^3 \\ z_2^3 \end{pmatrix} \tag{15}$$

The *Activation will be:*

$$a^3 = \begin{pmatrix} \sigma(z_1^3) \\ \sigma(z_2^3) \end{pmatrix} \tag{16}$$

finally:

...

$$a^4 = \sigma \left(\begin{pmatrix} w_{11}^4 & w_{12}^4 \end{pmatrix} \begin{pmatrix} \sigma(z_1^3) \\ \sigma(z_2^3) \end{pmatrix} + \begin{pmatrix} b_1^4 \end{pmatrix} \right) \tag{17}$$

Summary

...

w^L is the matrix with neurons that connect the $(L-1)$ layer with (L) layer, $s(L)$ indicates *number of neurons* in L layer.

$$\underset{s(L)\times s(L-1)}{W^L} = \begin{pmatrix} w^L_{11} & w^L_{12} & \dots & w^L_{1s(L-1)} \\ w^L_{21} & w^L_{22} & \dots & w^L_{2s(L-1)} \\ \vdots & \vdots & \dots & \vdots \\ w^L_{s(L)1} & w^L_{s(L)2} & \dots & w^L_{s(L)s(L-1)} \end{pmatrix} \tag{18}$$

...

$$z^L_k = \sum_{i=1}^{s(L-1)} w^L_{ki} a^{L-1}_k + b^L_k \tag{19}$$

...

$$\sigma(z^L) = \sigma\left(\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_{s(L)} \end{pmatrix}\right) = \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_{s(L)}) \end{pmatrix} \quad (20)$$

Now the Function cost, is evaluated in

$$J(a^{\mathbb{L}}). \quad (21)$$

We can uses MLE over bernoulli distribution.

$$\quad (22)$$

Summary

...

$$\underset{s(L)\times 1}{z^L} = \underset{s(L)\times s(L-1)}{W^L} \quad \underset{s(L-1)\times 1}{a^{L-1}} + \underset{s(L)\times 1}{b^L} \tag{23}$$

Forward propagation

1. $z^2 = w^2 x + b^2$
2. $a^2 = \sigma(z^2)$
3. $z^3 = w^3 a^2 + b^3$
4. $a^3 = \sigma(z^3)$
5. $z^4 = w^4 a^3 + b^4$
6. $a^4 = \sigma(z^4)$

Forward propagation pseudocode

Algorithm 1: Forward propagation

Data: Feature vector
Result: Gradient vector
$a^1 \leftarrow x$ (feature vector);
$W^2 \leftarrow random$;
for $L = 2$ *to* $\mathbb{L}$ **do**
    $z^L = w^L a^{(L-1)} + b^L$;
    $a^L = \sigma(z^L)$;
end

Backpropagation

The idea behind the chain rule is fundamental here:

...

$$E \longleftarrow L \longleftarrow (L-1) \longleftarrow (L-2), ..., l, ..., \longleftarrow 1 \tag{24}$$

for instance; the weights and biases of $(L-2)$ layer affect the $(L-1)$ layer and so on. Namely, a composed function $E = f(L(L-1(L-2(...))))$.

Chain rule example

$$\begin{aligned} \frac{\partial E}{\partial (L-2)} &= \frac{\partial (L-1)}{\partial (L-2)} \frac{\partial L}{\partial (L-1)} \frac{\partial E}{\partial L} \\ &= \frac{\partial (L-1)}{\partial (L-2)} \frac{\partial E}{\partial (L-1)} \end{aligned} \tag{25}$$

Key

...

$$z_k^L = \sum_{i=1}^{s(L-1)} W_{ki}^L a_i^{L-1} + b_k^L \tag{26}$$

...

$$a_k^L = \sigma(z_k^L) \tag{27}$$

...

$$\begin{aligned} z_k^{L+1} &= \sum_{i=1}^{s(L)} w_{ki}^L a_i^L + b_k^{L+1} \\ &= \sum_{i=1}^{s(L)} W_{ki}^L \sigma(z_i^L) + b_k^{L+1} \end{aligned} \tag{28}$$

Key

...

$$\frac{\partial z_k^L}{\partial a_i^{L-1}} = w_{ki}^L \tag{29}$$

...

$$\frac{\partial z_k^L}{\partial w_{ki}} = a_i^{L-1} \tag{30}$$

...

$$\frac{\partial z_k^{L+1}}{\partial z_i^L} = \frac{\partial z_k^{L+1}}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} = \frac{\partial z_k^{L+1}}{\partial \sigma(z_i^L)}\frac{\partial \sigma(z_i^L)}{\partial z_i^L} = w_{ki}^{L+1}\sigma'(z_i^L) \tag{31}$$

Key

Error in last layer($\mathbb{L}$)

$$\frac{\partial E}{\partial z_i^{\mathbb{L}}} = \delta_i^{\mathbb{L}} \quad = \frac{\partial a_i^{\mathbb{L}}}{\partial z_i^{\mathbb{L}}} \frac{\partial E}{\partial a_i^{\mathbb{L}}} \tag{32}$$

Error in any layer(L)

$$\begin{aligned} \frac{\partial E}{\partial z_i^{L}} = \delta_i^{L} &= \sum_{j=1}^{s(L)} \frac{\partial z_j^{L+1}}{\partial z_i^{L}} \frac{\partial E}{\partial z_j^{L+1}} \\ &= \sum_{j=1}^{s(L+1)} w_{ji}^{L+1} \sigma'(z_i^{L}) \delta_j^{L+1} \end{aligned} \tag{33}$$

Local gradient

...

$$\begin{aligned}\frac{\partial E}{\partial w_{ik}^{L}} &= \frac{\partial z_i^L}{\partial w_{ik}^L}\frac{\partial E}{\partial z_i^L} \\ &= \frac{\partial z_i^L}{\partial w_{ik}^L}\delta_i^L \\ &= a_k^{L-1}\delta_i^L\end{aligned} \tag{34}$$

Vectorized

∇ Gradient

$$\nabla_{\boldsymbol{a}} C = \begin{pmatrix} \frac{\partial C}{\partial a_1} \\ \frac{\partial C}{\partial a_2} \\ \vdots \\ \frac{\partial C}{a_n} \end{pmatrix} \tag{35}$$

$\odot$ Hadamard product (Element-Wise product)

$$(A \odot B)_{ij}) = A_{ij}.B_{ij} \tag{36}$$

Multiply each element of A and B in the same index position ij.

Vectorized errors

$\delta_i^{\mathbb{L}}$

$$
\begin{aligned}
\delta_1^{\mathbb{L}} &= \sigma'(z_1^{\mathbb{L}})E'(a_1^{\mathbb{L}}) \\
\delta_2^{\mathbb{L}} &= \sigma'(z_2^{\mathbb{L}})E'(a_2^{\mathbb{L}}) \\
&\vdots \\
\delta_{s(\mathbb{L})}^{\mathbb{L}} &= \sigma'(z_{s(\mathbb{L})}^{\mathbb{L}})E'(a_{s(\mathbb{L})}^{\mathbb{L}})
\end{aligned}
\tag{37}
$$

...

$$
\underset{s(\mathbb{L})\times 1}{\delta^{\mathbb{L}}} = \underset{s(\mathbb{L})\times 1}{\sigma'(z^{\mathbb{L}})} \odot \underset{s(\mathbb{L})\times 1}{\nabla_{\boldsymbol{a}^{\mathbb{L}}} E}
\tag{38}
$$

Vectorized errors

According to previous results $\delta_i^L = \sum_{j=1}^{s(L+1)} w_{ji}^{L+1}\delta_j^{L+1}\sigma'(z_i^L)$

...

$$\begin{pmatrix} \delta_1^L \\ \delta_2^L \\ \vdots \\ \delta_{s(L)}^L \end{pmatrix} = \begin{pmatrix} w_{11}^{L+1} & w_{21}^{L+1} & \dots & w_{s(L+1)1}^{L+1} \\ w_{12}^{L+1} & w_{22}^{L+1} & \dots & w_{s(L+1)2}^{L+1} \\ \vdots & \vdots & \vdots & \vdots \\ w_{1s(L)}^{L+1} & w_{2s(L)}^{L+1} & \dots & w_{s(L+1)S(L)}^{L+1} \end{pmatrix} \begin{pmatrix} \delta_1^{L+1} \\ \delta_2^{L+1} \\ \vdots \\ \delta_{s(L+1)}^{L+1} \end{pmatrix} \odot \begin{pmatrix} \sigma'(z_1^L) \\ \sigma'(z_2^L) \\ \vdots \\ \sigma'(z_{s(L)}^L) \end{pmatrix}$$

...

$$\underset{s(L)\times 1}{\delta^L} = \underset{s(L)\times s(L+1)}{\left(w^{L+1}\right)^T} \underset{s(L+1)\times 1}{\delta^{L+1}} \odot \underset{s(L)\times 1}{\sigma'(z^L)} \tag{39}$$

Backprop

Algorithm 2: Backward Propagation

Data: Initial error

Result: Error vectors

$\delta^{\mathbb{L}} \leftarrow \sigma'(z^{\mathbb{L}}) \odot \nabla_{\boldsymbol{a}^{\mathbb{L}}} E$ (Initial error) ;

for $L = \mathbb{L} - 2$ *to* 2 **do**

$\quad \delta^L = (w^{L+1})^T \delta^{L+1} \odot \sigma'(z^L)$

end

Algorithm 3: Back propagationa and gradient descendent

Data: Initial error

Result: Error vectors

$W^L \quad \forall L \leftarrow$ *random* (initial random solution) ;

$a^1 \leftarrow x$ (Feature vector) ;

for *n to N (patterns)* **do**

  for $L = 2$ *to* $\mathbb{L}$ **do**

    $z^L = w^L a^{(L-1)} + b^L$;

    $a^L = \sigma(z^L)$;

  end

  $E, \delta^{\mathbb{L}} \leftarrow J(a^L), \sigma'(z^{\mathbb{L}}) \odot \nabla_{\boldsymbol{a}^{\mathbb{L}}} E$ (Initial error)

  for $L = \mathbb{L} - 2$ *to* 2 **do**

    $\delta^L = (w^{L+1})^T \delta^{L+1} \odot \sigma'(z^L)$

  end

  for $L = \mathbb{L}$ *to* 2 **do**

    $w^L, b^L = w^L - \alpha\delta^L (a^{L-1})^T, b^L - \alpha\delta^L$

  end

end

Error minimization

Is not a restricted optimization problem!

$$\min L(\vec{w}, y) \tag{40}$$

you are not secure that is a **convex problem!** therefore *First Order Condition*(FCO) not is a solution, but could be very useful as mechanism to implement a guiaded search!.

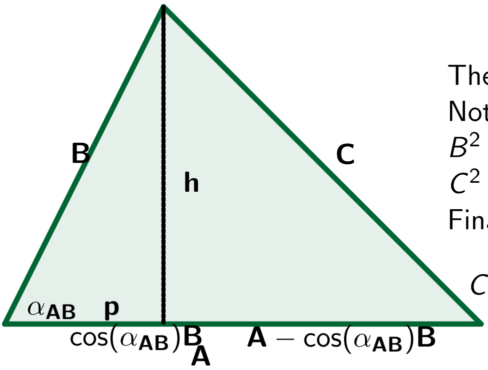
Surface problem

illustration

There a lot of directions!

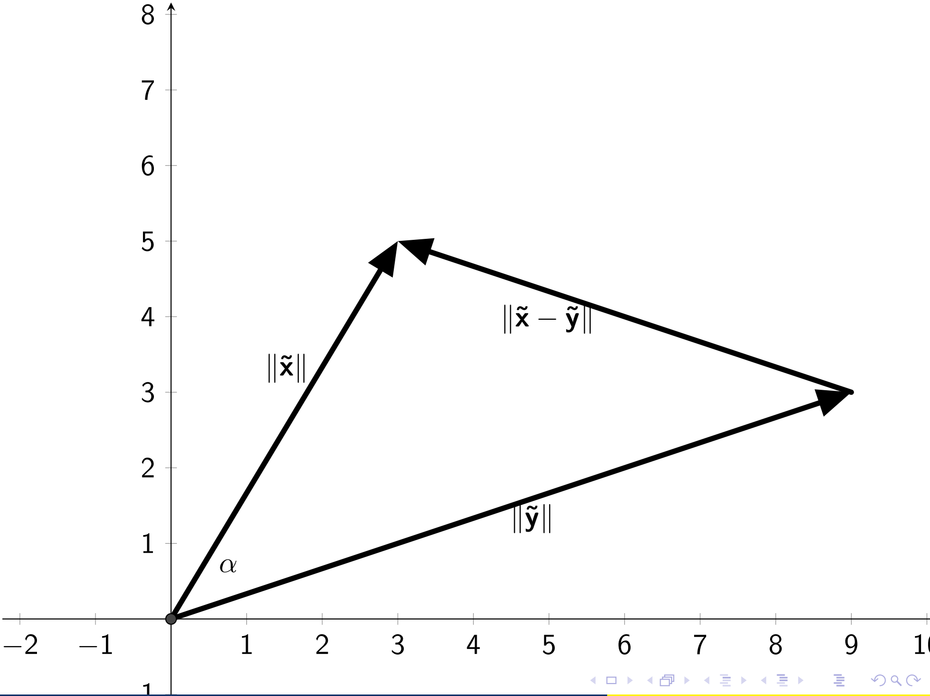
Cosine law

Generalization of pythogoream theorem



The idea is related C with B and A. Note that $cos(\alpha_{AB}) = \frac{p}{B}$. Therefore $B^2 = (\cos(\alpha_{AB})B)^2 + h^2$, and $C^2 = (A - \cos(\alpha_{AB})B)^2 + h^2$. Finally $C^2 - B^2$.

$$C^2 = A^2 + B^2 - 2A\cos(\alpha_{AB}) \quad (41)$$


$\|\tilde{\mathbf{x}}\|$
$\|\tilde{\mathbf{x}} - \tilde{\mathbf{y}}\|$
$\|\tilde{\mathbf{y}}\|$
α

Norm

The length or norm of a vector $\vec{x} \in R^n$ expressed as $\|\vec{x}\|$ it also important and will be defined using the pitagoras theorem. The norm of a vector is defined as $\sqrt{\sum x_i^2}$. According to the definition of inner product we can said that $\sqrt{\vec{x}.\vec{x}} = \|\vec{x}\|$ or $\vec{x}.\vec{x} = \|\vec{x}\|^2$.

Ortogonality proof

$$\begin{aligned}\|\vec{x}-\vec{y}\|^2 =& (\vec{x}-\vec{y}).(\vec{x}-\vec{y}) \\ =& \|\vec{x}\|^2+\|\vec{y}\|^2-2\vec{x}.\vec{y}\end{aligned} \tag{42}$$

Now by cosine law we have that $\|\vec{x}-\vec{y}\|^2 = \|\vec{x}\|^2+\|\vec{y}\|^2-2\|\vec{x}\|\|\vec{y}\|\cos\alpha$.
Now if we equate the expresions

$$\vec{x}.\vec{y} = \|\vec{x}\|\|\vec{y}\|\cos\alpha \tag{43}$$

$$\cos\alpha = \frac{\vec{x}.\vec{y}}{\|\vec{x}\|\|\vec{y}\|} \tag{44}$$

Derivate the folling $\cos(\frac{\pi}{2}) = \cos(90) = 0$ (Uses cartesian representation).

Ortogonality proof

In cartesian product we can said that:

$$\cos(\alpha) = \frac{x}{\sqrt{x^2 + y^2}} \tag{45}$$

in the case of $\cos(90) = \frac{0}{y}$. In this case $\vec{x}.\vec{y} = 0$ because are ortogonal.

Directional derivative

if we have $z = f(x, y)$ We have a lot of directions to descendent we could descendt in the direction to the vector:

$$\vec{v} = \begin{bmatrix} \delta_1 & \delta_2 \end{bmatrix} \tag{46}$$

Whichs means δ_1 steps in the x direction and δ_2 steps in y direction;

$$\frac{df(x, y)}{d\vec{v}} = \frac{\partial f(x, y)}{\partial x}\delta_1 + \frac{\partial f(x, y)}{\partial y}\delta_2 \tag{47}$$

Directional derivative

We call gradient to

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \end{bmatrix} \tag{48}$$

therefore

$$\frac{df(x, y)}{d\vec{v}} = \vec{v}.\nabla f(x, y) \tag{49}$$

Here there are a important "remark" that $\vec{v}$ is scalable for any $k \in \mathbb{R}$ to avoid this constrain $\|\vec{v}\| = 1$.

Optimization problem

find out $\vec{v'}$ that max $\frac{df(x,y)}{d\vec{v}}$, remember that $\arg\max_x f(x) = \{x \mid \forall x' : f(x') \leq f(x)\}$, therefore

$$
\begin{aligned}
\vec{v'} &= \underset{\|\vec{v}\|=1}{\arg\max}\, \vec{v}.\nabla f(x,y) \\
\vec{v'} &= \underset{\|v\|=1}{\arg\max}\, \|\vec{v}\| \|\nabla f(x,y)\| \cos(\alpha)
\end{aligned}
\tag{50}
$$

Nothe that $\nabla f(x,y)$ not is function of $\vec{v}$ therefore the problem is to find the vector whose angle produce the major $\cos(\alpha)$.

Optimization problem

$$\vec{v'} = \arg\max_{\|\vec{v}\|} \cos(\alpha) \tag{51}$$

remeber that $cos : \mathbb{R} \rightarrow [-1, 1]$ and $\cos(0) = 1$ therefore $\vec{v'}$ is parallel to the vector $\nabla f(x, y)$.

Same direction

The direction of steepest descent ($\vec{v'}$) from the point (x, y) is equal to $\nabla f(x, y)$.

$$\vec{v'} = k \nabla f(x, y) \tag{52}$$

Optimization problem

Now remember that $\|\vec{v'}\| = 1$,

$$\vec{v'} = \frac{\nabla f(x, y)}{\|f(x, y)\|} \tag{53}$$

the directional derivative is:

$$\frac{df(x, y)}{d\vec{v'}} = \vec{v'}.\nabla f(x, y) = \frac{\|\nabla f(x, y)\|^2}{\|\nabla f(x, y)\|} \tag{54}$$

Rate of change is the norm of the same gradient

$$\frac{df(x, y)}{d\vec{v'}} = \|\nabla f(x, y)\| \tag{55}$$

Gradient descendent

Now that we find the way of descendent or minimize the error training, we can uses:

$$\vec{W}_{t+1} = \vec{W}_t - \alpha \nabla f(x, y) \tag{56}$$

Proofs?

Gradient. Hessian, Jacobian

Jacobian

Defined to hold the partial derivatives of **vector valued function**

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^n \tag{57}$$

Example

for $g : \mathbb{R}^2 \to \mathbb{R}^2$

$$g(x, y) = g(f_1(x, y), f_2(x, y)) \tag{58}$$

$$\mathbb{J} = \begin{bmatrix} \left.\frac{\partial f_1}{\partial x}\right|_{x_0, y_0} & \left.\frac{\partial f_1}{\partial y}\right|_{x_0, y_0} \\ \left.\frac{\partial f_2}{\partial x}\right|_{x_0, y_0} & \left.\frac{\partial f_2}{\partial y}\right|_{x_0, y_0} \end{bmatrix} = \begin{bmatrix} \nabla f_1(x_0, y_0) \\ \nabla f_2(x_0, y_0 \end{bmatrix}$$

Gradient, Hessian, Jacobian

Recap

Gradient

We defined the gradient for a scalar valued function $f()$:

$$f : \mathbb{R}^n \rightarrow \mathbb{R} \tag{59}$$

Hessian

The hessian is the derivative of the gradient of a scalar valued function $f()$

$$f : \mathbb{R}^n \rightarrow \mathbb{R} \tag{60}$$

$$\mathbf{H}f(x, y) = \nabla^2 f(x, y) = \begin{bmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{bmatrix} \tag{61}$$

Derivatives

We need derivatives to adjust weights!!

We can compute **forward** but how we propagate the errors to all layers (**backward propagation**)

- calculus
- Numerical (finite differences)
- symbolic differentiation
- Automatic differentiation

Automatic differentiation

Derivative

$$f : D \to I \tag{62}$$

$$df : D \to (D \multimap I) \tag{63}$$

where $\multimap$ means a linear mapping.

This fact coud be seen in the next equation:

$$f(x + \Delta x) \approx f(x) + \frac{df(x)}{dx}\Delta x \tag{64}$$

Linear Map

A linear map, $\mathbb{L} : D \multimap I$ satisfy:

- $\forall x, y : D \quad \mathbb{L} \odot (x + y) = \mathbb{L} \odot x + \mathbb{L} \odot y$
- $\forall k \in \mathbb{R}, x : D \quad k * (\mathbb{L} \odot x) = \mathbb{L} \odot (k * x)$

Matrix properties of linearity

Think in the properties of linear transformation in matrices..

the operators $*, +$ vary according to the objects for instance $(*)$ scalar or dot product.

Chain rule

A composed function:

$$\begin{aligned} f(x) &= g(h(x)) \\ f &= f \circ h \end{aligned} \tag{65}$$

therefore the derivation is also is a composition of linear maps!

Automatic differentiation

AD is descomposition in aritmetic operations $(+, -, *, /)$ and another elementaries as exponential.

Automatic differentiation

There are two ways:

- Forward mode
- Reverse mode

Backpropagation

Backpropagation is therefore a special case of reverse-mode AD for scalar functions.

Antother alternatives

Backpropagation is a learning method, but if try with GA?

Train a Neural Network with GA