Lecture 06

Deep Learning 01

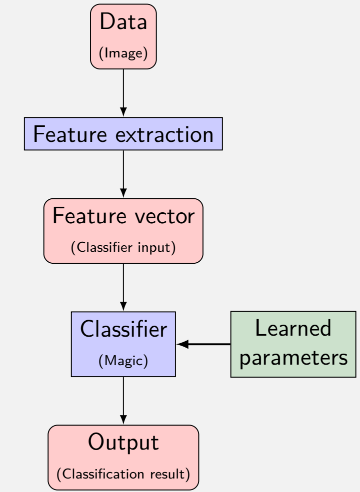
Shallow Neural Networks

2026-03-27

Sébastien Valade

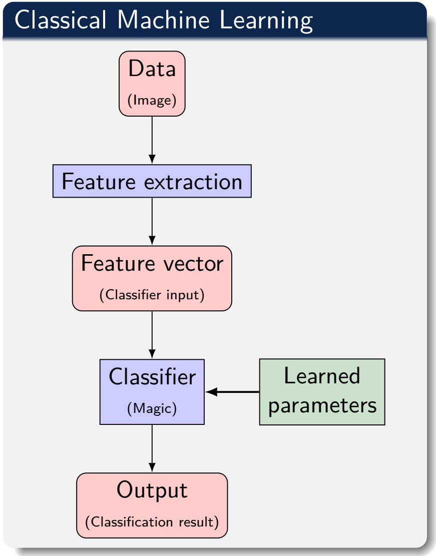
From *classical learning* to *modern learning*

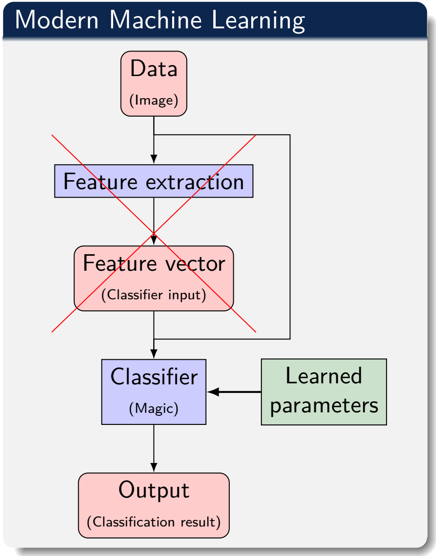
Classical Machine Learning



Introduction

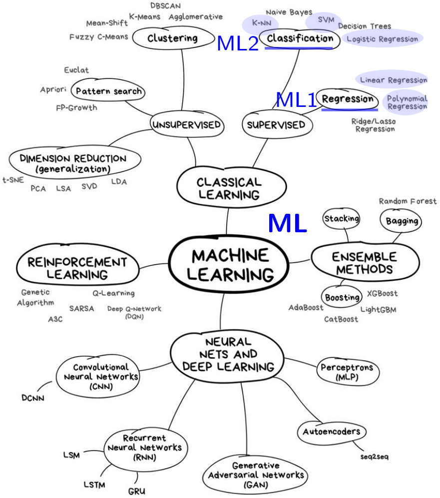
From *classical learning* to *modern learning*





Introduction

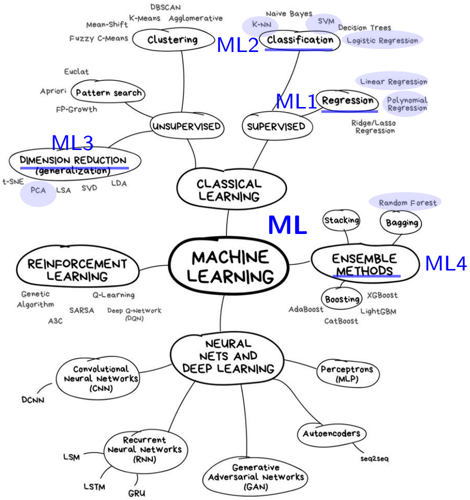
Last weeks: Classical Learning *(ML)*



source

Introduction

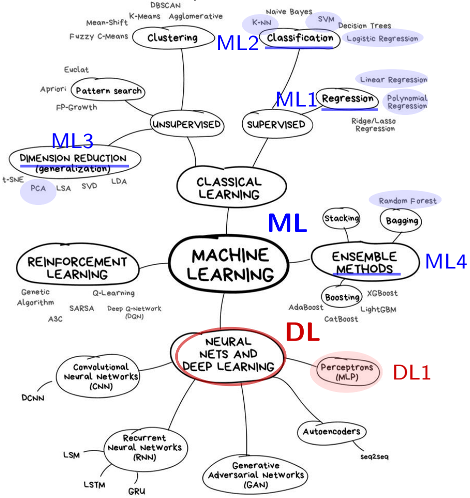
Last weeks: Classical Learning *(ML)*



source

Introduction

Last weeks: Classical Learning *(ML)*
This week: Deep Learning *(Part-1: shallow nets)* - DL1



source

Introduction

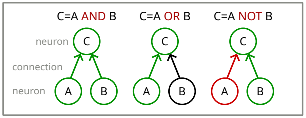
Brief history of Neural Networks

AI Golden Age — AI Winter — AI Golden Age — AI Winter — AI Golden Age

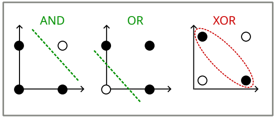
- 1943 **Artificial Neuron AN** *McCulloch-Pitts*
- 1958 **Perceptron** *Rosenblatt*
- 1959 **Adaptive Linear Neuron ADALINE** *Widrow-Hoff*
- 1969 **XOR problem** *Minsky-Papert*
- 1986 **Multi-Layer Perceptron MLP (backpropagation)** *Rumelhart-Hinton-Williams*
- 1995 **SVM** *Vapnik-Cortes*
- 1998 **Convolutional Neural Networks CNN** *Lecun*
- 2012 **AlexNet** *Krizhevsky et al.*
- 2014 **GAN**
- 2017 **Transformer** *(attention)* *Vaswani et al.*
- 2020 **LLM** *(GPT-3)*
- 2022 **RLHF** *(ChatGPT)*
- 2023 **MFM** *(GPT-4)*

1940 1950 1960 1970 1980 1990 2000 2010 2020 2030

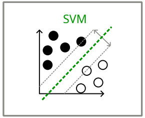
1943 - **Artificial Neurons**



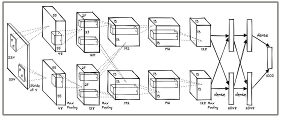
1969 - **XOR problem**



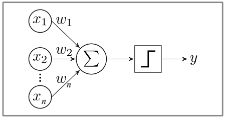
1995 - **Support Vector Machines (SVM)**



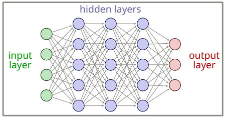
2012 - **AlexNet**



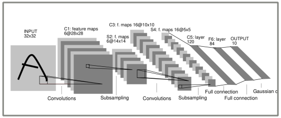
1958-1959 - **Perceptron & Adaline**



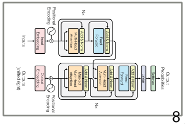
1986 - **Multi-Layer Perceptrons (MLP)**



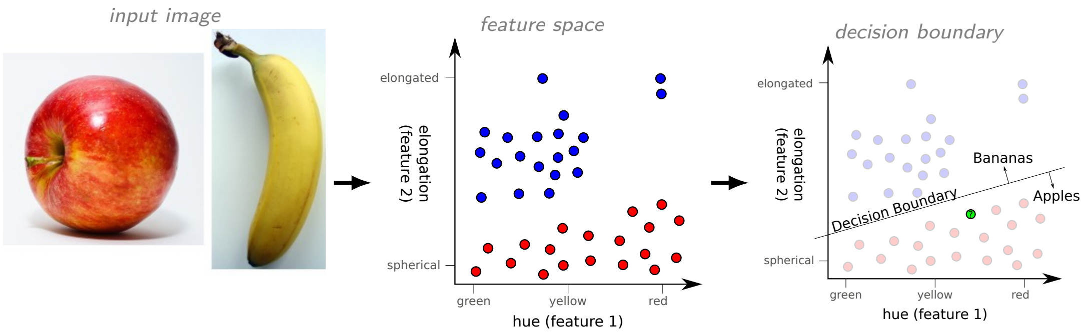
1998 - **Convolutional Neural Networks (CNN)**



2017 - **Transformers**



Recall our toy example from last week: *classify fruit images into either bananas or apples*



Perceptron classifier

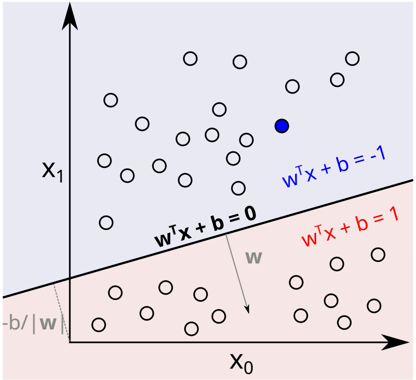
⇒ algorithm which classifies data based on linear decision boundary

⇒ **perceptron**:

binary classification (single sample)

$$\boxed{\begin{aligned}\hat{y} &= \text{sign}(\mathbf{w}^T\mathbf{x} + b)\\ &= \text{sign}(w_0 x_0 + w_1 x_1 + b)\end{aligned}}$$

- $\hat{y} \in \{-1|1\}$: predicted label → *banana | apple*
- $\mathbf{x} \in \mathbb{R}^2$: feature vector → *[hue, elongation]*
- $\mathbf{w} \in \mathbb{R}^2$: **weight vector** → needs to be learned
- $b \in \mathbb{R}$: **bias** → needs to be learned
- *sign*: sign function returning the sign of a real number



Perceptron classifier

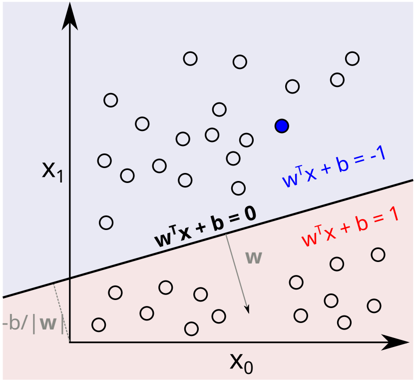
⇒ algorithm which classifies data based on linear decision boundary

⇒ **perceptron**:

binary classification (single sample)

$$\boxed{\begin{aligned}\hat{y} &= \text{sign}(\mathbf{w}^T\mathbf{x} + b) \\ &= \text{sign}(w_0x_0 + w_1x_1 + b)\end{aligned}}$$

- $\hat{y} \in \{-1|1\}$: predicted label → *banana | apple*
- $\mathbf{x} \in \mathbb{R}^2$: feature vector → *[hue, elongation]*
- $\mathbf{w} \in \mathbb{R}^2$: **weight vector** → needs to be learned
- $b \in \mathbb{R}$: **bias** → needs to be learned
- *sign*: sign function returning the sign of a real number



Example:

→ given sample i: $\mathbf{x}_i = [2, -1]$

→ given weights and bias: $\mathbf{w} = [-4, 3]$, $b = 0.5$

→ predicted label: $\hat{y}_i = \text{sign}(-4 \cdot 2 + 3 \cdot (-1) + 0.5) = \text{sign}(-10.5) = -1$

Perceptron classifier

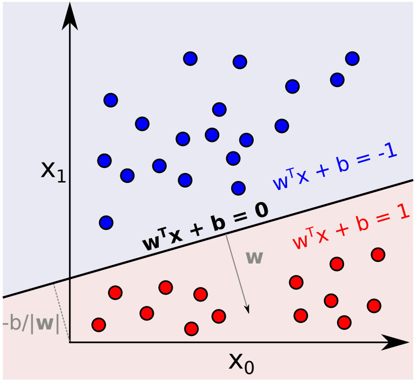
⇒ algorithm which classifies data based on linear decision boundary

⇒ **perceptron**:

binary classification (dataset)

$$\boxed{\begin{aligned}\hat{\mathbf{y}} &= \text{sign}(X\mathbf{w} + b) \\ &= \text{sign}\left(\begin{bmatrix} x_{1,0} & x_{1,1} \\ \vdots & \vdots \\ x_{n,0} & x_{n,1}\end{bmatrix}\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} + b\right)\end{aligned}}$$

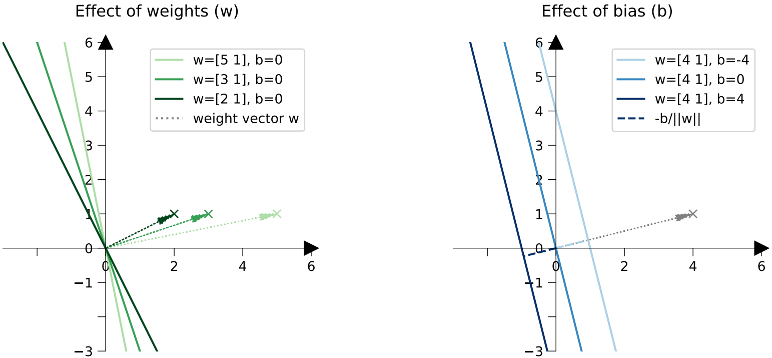
- $\hat{\mathbf{y}} \in \{-1|1\}^n$: predicted labels vector
- $X \in \mathbb{R}^{n\times 2}$: feature matrix → *[hue, elongation]*
- $\mathbf{w} \in \mathbb{R}^2$: **weight vector** → needs to be learned
- $b \in \mathbb{R}$: **bias** → needs to be learned
- *sign*: applied element-wise



Perceptron classifier

$\Rightarrow$ influence of **weight vector w** & **bias b** on decision boundary:

- weight vector **w**: determines the *normal vector* to the decision boundary
- bias b: shifts the decision boundary in the direction of the weight vector
  ($b > 0$ shifts the boundary in the negative direction of the weight vector, and vice versa)



Perceptron

$\Rightarrow$ the perceptron computes predictions in two steps:

1. computes a linear function of the **input vector** $\mathbf{x}$ and the associated **weight vector** $\mathbf{w}$, plus a **bias term** b
   $\Rightarrow$ *weighted sum (linear combination) = sum of all inputs multiplied by their weights, plus bias term*

$$\begin{aligned} z &= \mathbf{w}^T\mathbf{x} + b \\ &= w_0x_0 + w_1x_1 + b \end{aligned}$$

where:

- $\mathbf{x} = [x_0, x_1]^T$ = input vector $\rightarrow$ *[hue, elongation]*
- $\mathbf{w} = [w_0, w_1]^T$ = weight vector $\rightarrow$ *1 per weight per feature*
- b = bias term

Perceptron

⇒ the perceptron computes predictions in two steps:

1. computes a linear function of the **input vector x** and the associated **weight vector w**, plus a **bias term** b
   ⇒ *weighted sum (linear combination) = sum of all inputs multiplied by their weights, plus bias term*

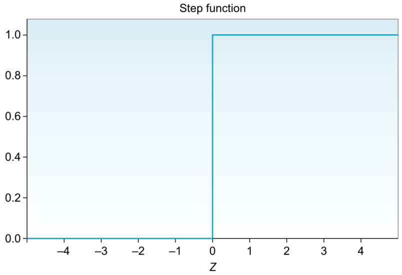
$$z = \mathbf{w}^T\mathbf{x} + b$$
$$= w_0x_0 + w_1x_1 + b$$

where:

- $\mathbf{x} = [x_0, x_1]^T$ = input vector → *[hue, elongation]*
- $\mathbf{w} = [w_0, w_1]^T$ = weight vector → *1 per weight per feature*
- b = bias term

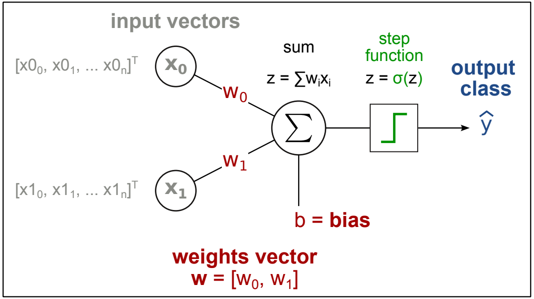
2. applies a **step function**, typically the *modified sign function* (here "wrongfully" denoted as σ)

$$\hat{y} = \sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$



Perceptron

⇒ the two steps can be represented graphically as follows:



Single sample:

$$\begin{aligned}\hat{y} &= \sigma(\mathbf{w}^T\mathbf{x} + b)\\ &= \sigma\left(w_0x_0 + w_1x_1 + b\right)\end{aligned}$$

Dataset of n samples:

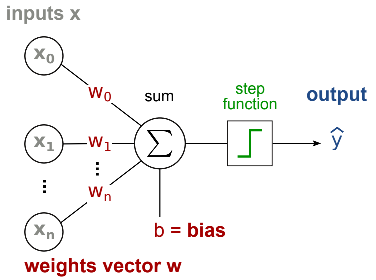
$$\begin{aligned}\hat{\mathbf{y}} &= \sigma(\mathbf{X}\mathbf{w} + b)\\ &= \sigma\left(\begin{pmatrix} x_{1,0} & x_{1,1}\\ \vdots & \vdots\\ x_{n,0} & x_{n,1}\end{pmatrix}\begin{pmatrix} w_0\\ w_1\end{pmatrix} + b\right)\\ &= \sigma\left(\begin{pmatrix} w_0x_{1,0} + w_1x_{1,1} + b\\ \vdots\\ w_0x_{n,0} + w_1x_{n,1} + b\end{pmatrix}\right)\\ &= \begin{pmatrix} \sigma\left(w_0x_{1,0} + w_1x_{1,1} + b\right)\\ \vdots\\ \sigma\left(w_0x_{n,0} + w_1x_{n,1} + b\right)\end{pmatrix}\end{aligned}$$

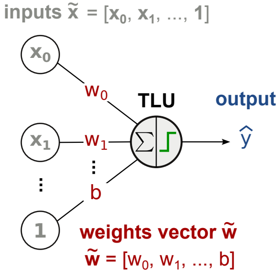
Perceptron

⇒ another representation is with the weight vector $\tilde{\mathbf{w}}$ and augmented input vector $\tilde{\mathbf{x}}$:

→ *the bias term b is learned as a weight* $w_n = b$, *and the input vector* x *is augmented with a vector of values* $x_n = 1$

⇒ this representation is that of an **artificial neuron** called the **threshold logic unit (TLU)**:

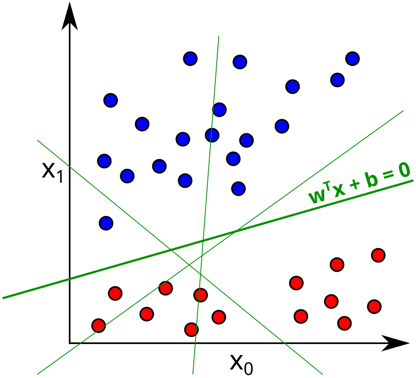




$$\hat{y} = \sigma(\mathbf{w}^T x + b) = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

$$\hat{y} = \sigma(\tilde{\mathbf{w}}^T \tilde{x}) = \sigma\left(\sum_{i=1}^{n} \tilde{w}_i \tilde{x}_i\right)$$

Perceptron



⇒ how is the perceptron trained?

i.e., how are the weights **w** *and bias b learned?*

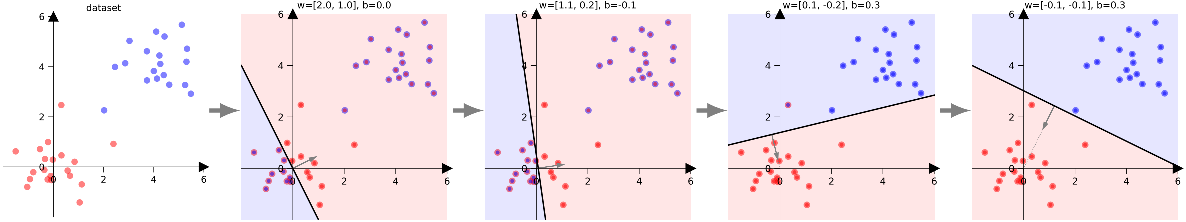
Perceptron learning algorithm

⇒ there is no analytical solution to the perceptron learning problem

⇒ **iterative algorithm** to learn the weight vector $\mathbf{w}$ and bias b that *minimize the classification error*:

> `initialize` $\tilde{\mathbf{w}}$ with random values
> `for` each training samples $\tilde{\mathbf{x}}_\mathbf{i}$:
>   `predict` $\hat{y}_i = \text{sign}(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}}_\mathbf{i})$
>   `update` $\tilde{\mathbf{w}}$ if $\hat{y}_i \neq y_i$:
> $\tilde{\mathbf{w}} = \tilde{\mathbf{w}} + \eta \cdot (y_i - \hat{y}_i) \cdot \tilde{\mathbf{x}}_\mathbf{i}$ where η is the learning rate
>   where $\mathbf{w} = \tilde{\mathbf{w}}[:\text{-}1]$, $b = \tilde{\mathbf{w}}[\text{-}1]$

⇒ Illustration of the convergence of the perceptron learning algorithm:



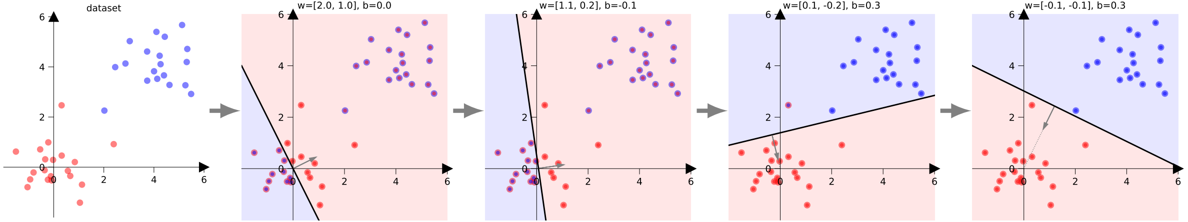
Perceptron learning algorithm

⇒ there is no analytical solution to the perceptron learning problem

⇒ **iterative algorithm** to learn the weight vector $\mathbf{w}$ and bias b that *minimize the classification error*:

initialize $\tilde{\mathbf{w}}$ with random values
for each training samples $\tilde{\mathbf{x}}_\mathbf{i}$:
  predict $\hat{y}_i = \text{sign}(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}}_\mathbf{i})$
  update $\tilde{\mathbf{w}}$ if $\hat{y}_i \neq y_i$:
    $\boxed{\tilde{\mathbf{w}} = \tilde{\mathbf{w}} + \eta \cdot (y_i - \hat{y}_i) \cdot \tilde{\mathbf{x}}_\mathbf{i}}$ where η is the learning rate — where does this come from?
    where $\mathbf{w} = \mathbf{w}$[: -1], $b = \mathbf{w}$[-1]

⇒ Illustration of the convergence of the perceptron learning algorithm:



Gradient descent of the Loss function (for the perceptron)

$\Rightarrow$ learning the weights $\tilde{\mathbf{w}}$ means modifying them such that predicted labels $\hat{y}$ get closer to true labels y

Gradient descent of the Loss function (for the perceptron)

$\Rightarrow$ learning the weights $\tilde{\mathbf{w}}$ means modifying them such that predicted labels $\hat{y}$ get closer to true labels y

$\Rightarrow$ **loss function $\mathcal{L}$** = measure of the difference between predicted and true labels

L2 loss = mean squared error (MSE) = $\boxed{\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2}$

Gradient descent of the Loss function (for the perceptron)

⇒ learning the weights $\tilde{\mathbf{w}}$ means modifying them such that predicted labels $\hat{y}$ get closer to true labels y

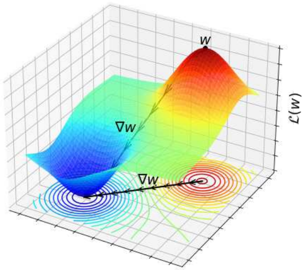
⇒ **loss function $\mathcal{L}$** = measure of the difference between predicted and true labels

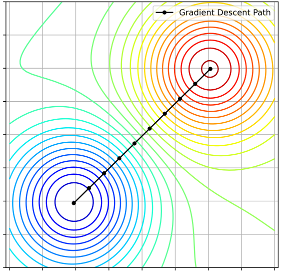
L2 loss = mean squared error (MSE) = $\boxed{\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2}$

⇒ **gradient descent**: starting from a random point on the function $\mathcal{L}(w)$, move in the direction of the steepest descending gradient with a step size η (learning rate)

update rule: $\boxed{w = w - \eta \nabla w}$ where $\nabla w = \frac{d\mathcal{L}}{dw}$ = *gradient of the loss function $\mathcal{L}$ with respect to w*

Note: for simplicity $\tilde{w}$ is here annotated w





Gradient descent of the Loss function

$\Rightarrow$ computing the derivative $\frac{d\mathcal{L}}{dw}$:

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}(y-\hat{y})^2 \\
\frac{d\mathcal{L}}{dw} &= \frac{d}{dw}\left\{\frac{1}{2}(y-\hat{y})^2\right\} \\
&= \frac{1}{2}\cdot 2\cdot(y-\hat{y})\cdot\frac{d}{dw}(y-\hat{y})^{2-1} \quad \text{applying the power rule, where } \frac{d}{dw}(u^n) = n\cdot u^{n-1}\cdot\frac{du}{dw} \\
&= (y-\hat{y})\cdot\frac{d}{dw}(y-w\cdot x) \quad \text{substituting } \hat{y} = w\cdot x \\
&= (y-\hat{y})\cdot-\frac{d}{dw}(w\cdot x) \quad \text{considering } \frac{dy}{dw} = 0 \text{ since } y \text{ is constant} \\
\nabla w &= -(y-\hat{y})\cdot x \quad \text{considering that } \frac{d(w\cdot x)}{dw} = x\text{, since } x \text{ is treated as a constant with respect to } w
\end{aligned}
$$

$\Rightarrow$ the weight update rule for gradient descent becomes: $\boxed{w = w - \eta\nabla w = w + \eta\cdot(y-\hat{y})\cdot x}$

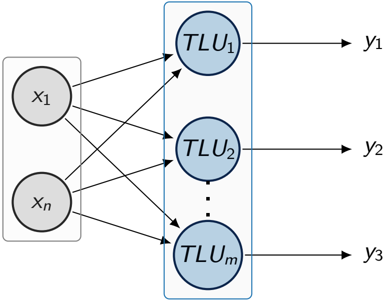
Reminder: common derivation rules

Rule	Function	Derivative
Constant Rule	$f(x) = c$	$f'(x) = 0$
Power Rule	$f(x) = x^n$	$f'(x) = n \cdot x^{n-1}$
Generalized Power Rule	$f(x) = u(x)^n$	$f'(x) = n \cdot u(x)^{n-1} \cdot u'(x)$
Sum Rule	$f(x) = u(x) + v(x)$	$f'(x) = u'(x) + v'(x)$
Difference Rule	$f(x) = u(x) - v(x)$	$f'(x) = u'(x) - v'(x)$
Product Rule	$f(x) = u(x) \cdot v(x)$	$f'(x) = u'(x) \cdot v(x) + u(x) \cdot v'(x)$
Quotient Rule	$f(x) = \frac{u(x)}{v(x)}$	$f'(x) = \frac{u'(x) \cdot v(x) - u(x) \cdot v'(x)}{[v(x)]^2}$
Chain Rule	$f(x) = u(v(x))$	$f'(x) = u'(v(x)) \cdot v'(x)$
Exponential Rule	$f(x) = e^x$	$f'(x) = e^x$
Exponential with Constant Base	$f(x) = a^x$	$f'(x) = a^x \cdot \ln(a)$
Logarithmic Rule	$f(x) = \ln(x)$	$f'(x) = \frac{1}{x}$
Sine Function	$f(x) = \sin(x)$	$f'(x) = \cos(x)$
Cosine Function	$f(x) = \cos(x)$	$f'(x) = -\sin(x)$
Tangent Function	$f(x) = \tan(x)$	$f'(x) = \sec^2(x) = \frac{1}{\cos^2(x)}$

Multiclass classifier

⇒ a perceptron can be composed of one or more TLUs (**neurons**) organized in a single layer, where every TLU is connected to every input: such a layer is called a **fully connected layer**, a.k.a. a **dense layer**

⇒ when the output layer contains several TLUs, the perceptron becomes a multilabel classifier

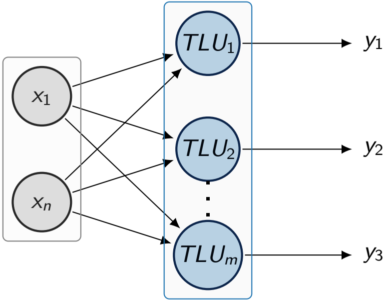


⇒ the perceptron is one of the simplest **Artificial Neural Network (ANN)** architecture

Multiclass classifier

⇒ a perceptron can be composed of one or more TLUs (**neurons**) organized in a single layer, where every TLU is connected to every input: such a layer is called a **fully connected layer**, a.k.a. a **dense layer**

⇒ when the output layer contains several TLUs, the perceptron becomes a multilabel classifier



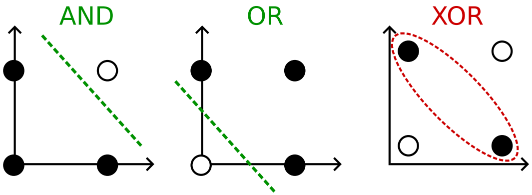
⇒ the perceptron is one of the simplest **Artificial Neural Network (ANN)** architecture

Note: a single-layer perceptron with multiple TLUs as output units acts as multiple linear classifiers in parallel, so the model is still linear.

Limitation of the perceptron classifier

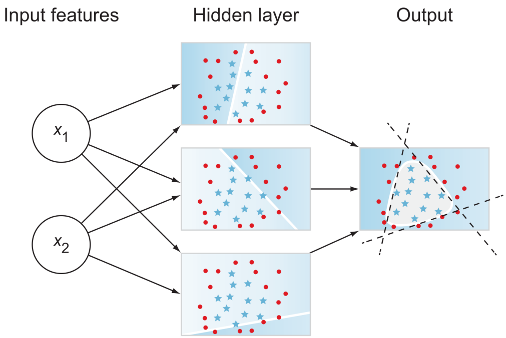
$\Rightarrow$ the single-layer perceptron is a *linear classifier*, so it cannot deal with even trivial *non-linear classifications*

EX: the **XOR problem** (*exclusive OR*)



Limitation of the perceptron classifier

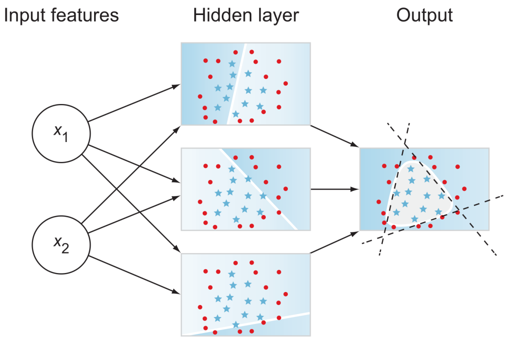
$\Rightarrow$ the limitations of perceptrons can be eliminated by stacking multiple perceptrons into several layers:



(from Elgendy 2020)

Limitation of the perceptron classifier

$\Rightarrow$ the limitations of perceptrons can be eliminated by stacking multiple perceptrons into several layers:

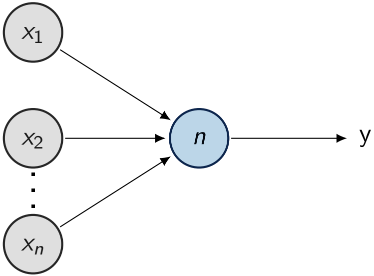


(from Elgendy 2020)

$\Rightarrow$ the resulting ANN is called a **multilayer perceptron (MLP)**

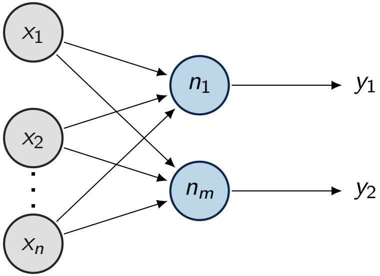
From single to multilayer perceptron

output layer with 1 neuron



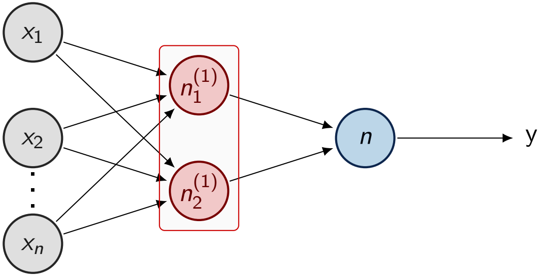
From single to multilayer perceptron

output layer with 2 neurons



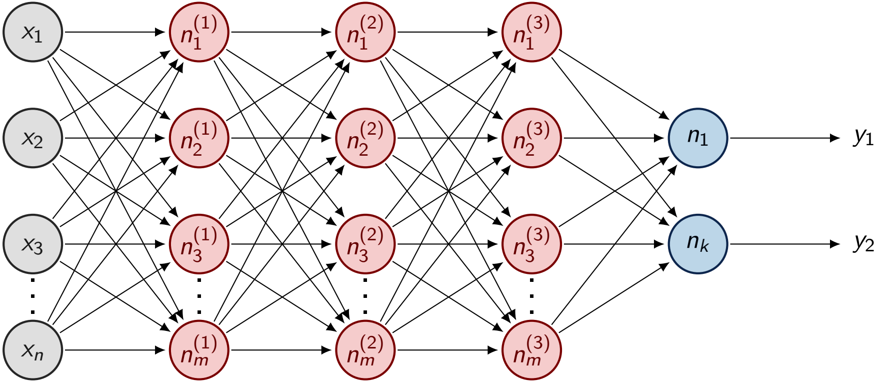
From single to multilayer perceptron

1 hidden layer with 2 neurons + output layer with 1 neuron (= 2 fully connected layers)



From single to multilayer perceptron

3 hidden layers (m neurons) + output layer (2 neurons) (= 4 fully connected layers)



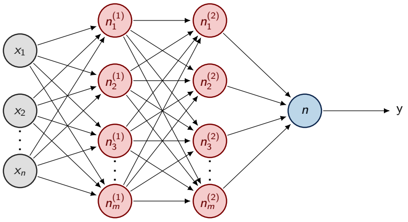
From single to multilayer perceptron

⇒ a **multilayer perceptron** (MLP) is also called a **feedforward neural network**, because information flows from input to output

⇒ a MLP consists of a concatenation of multiple **fully connected (dense) layers**, i.e. each neuron in one layer is connected to every neuron in the next layer

⇒ a MLP is a concatenation of multiple functions, the entire network can be written out as a long equation:

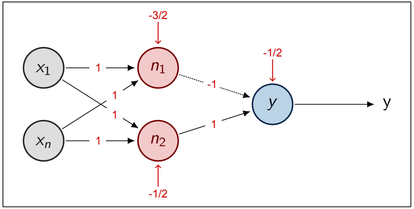
$$\hat{y} = f^{(out)}(w^{out} \cdot f^2(w^2 \cdot f^1(w^1 \cdot x)\ )\ )$$



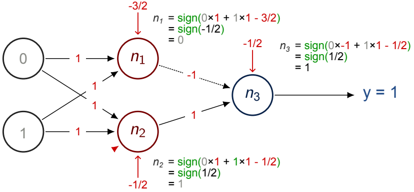

```
# Forward-pass calculations of a 3-layer neural network
f = lambda x: 1.0 / (1.0 + np.exp(-x)) # activation function (sigmoid)
x = np.random.randn(3, 1)    # input vector (3x1)
h1 = f(np.dot(w1, x) + b1)   # first hidden layer activations
h2 = f(np.dot(w2, h1) + b2)  # second hidden layer activations
out = np.dot(w3, h2) + b3    # output neuron
```

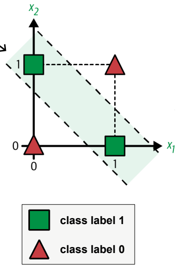
From single to multilayer perceptron

$\Rightarrow$ 1 hidden layer to solve the XOR problem *(example taken from book "Geron 2022")*



EX: classification of datapoint (x_1=0, x_2=1)

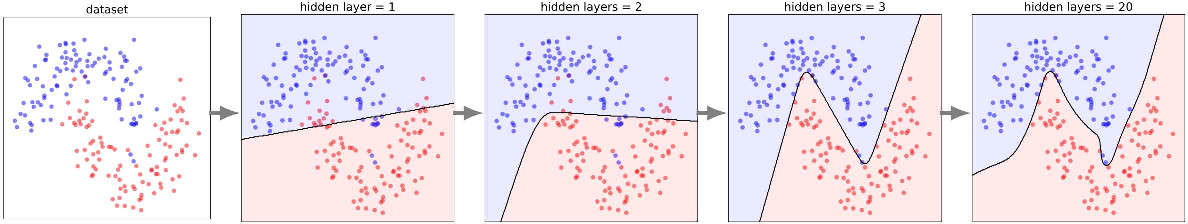




Effects of the number of hidden layers?

⇒ increasing the number of hidden layers allows the network to *learn more complex functions*

⇒ no need to worry about feature engineering, the *hidden layer of the network learn the features*!



Effects of the number of nodes in layers?

$\Rightarrow$ number of nodes in each layer:

- input layer: number determined by the *data dimensionality*
  - *EX: previous examples $x_1, x_2 \Rightarrow$ 2 input nodes*
  - *EX: MNIST handwritten digits dataset = 28x28 pixel images $\Rightarrow$ 784 input nodes*
- output layer: number determined by the *number of classes* to classify
  - *EX: binary classification $\Rightarrow$ 1 output node*
  - *EX: MNIST handwritten digits dataset $\Rightarrow$ 10 output nodes*
- hidden layer: number determined by the *complexity of the function to learn*
  - *EX: XOR problem $\Rightarrow$ 2 hidden nodes is enough*
  - *EX: MNIST handwritten digits dataset $\Rightarrow$ more nodes & hidden layers can capture more subtleties and improve performance*

  Nota Bene: higher network dimensionality (more nodes & layers) $\Leftrightarrow$ more computation and prone to overfitting!

Effects of the number of nodes in layers?

$\Rightarrow$ number of nodes in each layer:

- **input layer**: number determined by the *data dimensionality*
  - *EX: previous examples* $x_1, x_2 \Rightarrow$ *2 input nodes*
  - *EX: MNIST handwritten digits dataset = 28x28 pixel images* $\Rightarrow$ *784 input nodes*
- **output layer**: number determined by the *number of classes* to classify
  - *EX: binary classification* $\Rightarrow$ *1 output node*
  - *EX: MNIST handwritten digits dataset* $\Rightarrow$ *10 output nodes*
- **hidden layer**: number determined by the *complexity of the function to learn*
  - *EX: XOR problem* $\Rightarrow$ *2 hidden nodes is enough*
  - *EX: MNIST handwritten digits dataset* $\Rightarrow$ *more nodes & hidden layers can capture more subtleties and improve performance*

  Nota Bene: higher network dimensionality (more nodes & layers) $\Leftrightarrow$ *more computation and prone to overfitting!*

Effects of the number of nodes in layers?

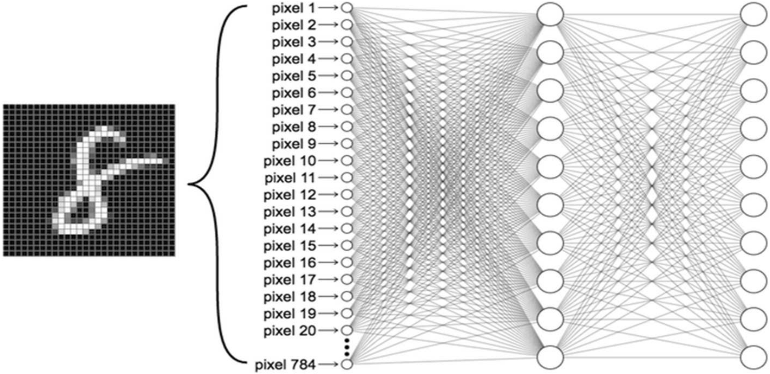
$\Rightarrow$ number of nodes in each layer:

- input layer: number determined by the *data dimensionality*
  - *EX: previous examples* $x_1, x_2 \Rightarrow$ *2 input nodes*
  - *EX: MNIST handwritten digits dataset = 28x28 pixel images* $\Rightarrow$ *784 input nodes*
- **output layer**: number determined by the *number of classes* to classify
  - *EX: binary classification* $\Rightarrow$ *1 output node*
  - *EX: MNIST handwritten digits dataset* $\Rightarrow$ *10 output nodes*
- **hidden layer**: number determined by the *complexity of the function to learn*
  - *EX: XOR problem* $\Rightarrow$ *2 hidden nodes is enough*
  - *EX: MNIST handwritten digits dataset* $\Rightarrow$ *more nodes & hidden layers can capture more subtleties and improve performance*

    Nota Bene: higher network dimensionality (more nodes & layers) $\Leftrightarrow$ *more computation and prone to overfitting!*

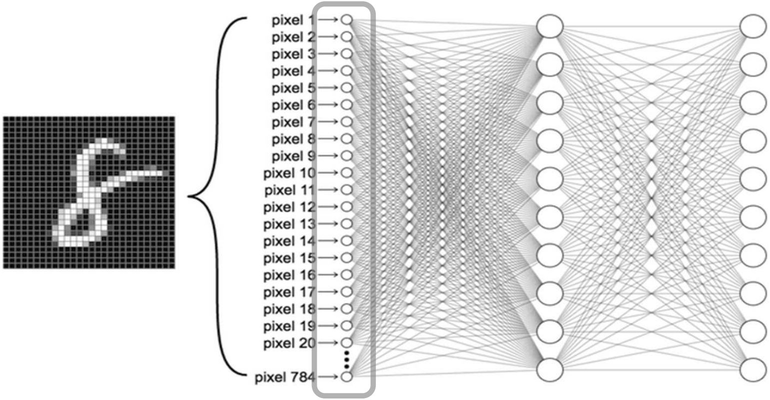
Effects of the number of nodes in layers?

⇒ classification of the MNIST dataset with a MLP



Effects of the number of nodes in layers?

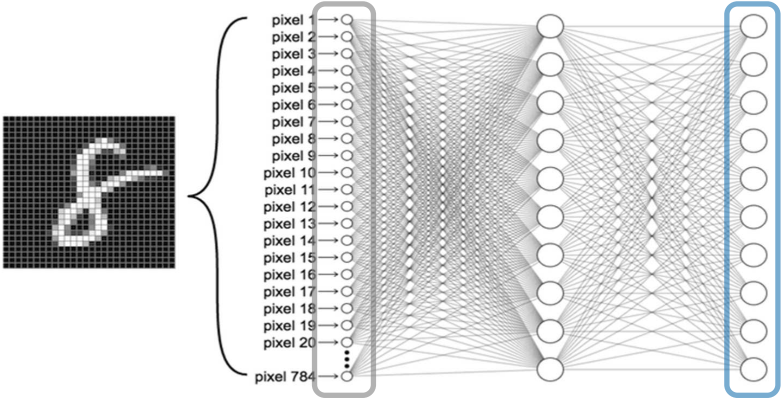
⇒ classification of the MNIST dataset with a MLP



input layer
28x28 pixels ⇒ 784 neurons

Effects of the number of nodes in layers?

⇒ classification of the MNIST dataset with a MLP

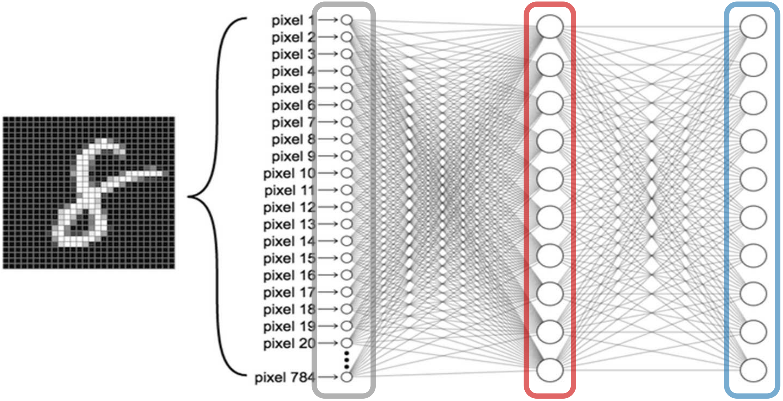


input layer
28x28 pixels ⇒ 784 neurons

output layer
10 digits ⇒ 10 neurons
softmax layer

Effects of the number of nodes in layers?

$\Rightarrow$ classification of the MNIST dataset with a MLP



input layer
28x28 pixels $\Rightarrow$ 784 neurons

hidden layer*

output layer
10 digits $\Rightarrow$ 10 neurons
softmax layer

* *hidden layer: would require more neurons to have proper performance*

$\Rightarrow$ **each node has weights and biases**

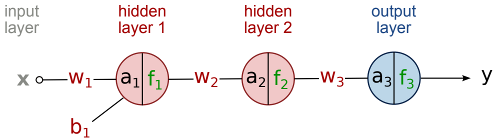
$\Rightarrow$ how do we train an MLP?

i.e., how do we learn all weights and biases across layers?

Using backpropagation to learn MLP

$\Rightarrow$ recall that the *feedforward* operations of a MLP is a concatenation of multiple functions:

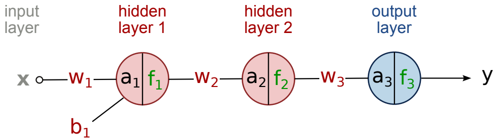
$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)\ )\ )$$



Using backpropagation to learn MLP

⇒ recall that the *feedforward* operations of a MLP is a concatenation of multiple functions:

$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)\ )\ )$$



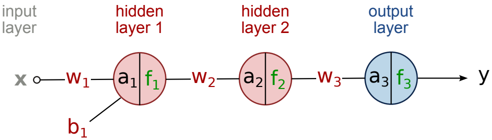
⇒ to learn the MLP $y = f_{MLP}(x; \theta)$ means that the MLP parameters $\theta = \{f, w, b\}$ need to be optimized to best fit the training dataset $\{x,\ y\}$, where:

- f = node activation functions *(some activation functions have parameters, e.g. the Leaky ReLU $f(x) = max(\alpha x, x)$)*
- w = node weights
- b = node biases

Using backpropagation to learn MLP

⇒ recall that the *feedforward* operations of a MLP is a concatenation of multiple functions:

$$y = f_3(w_3 \cdot f_2(w_2 \cdot f_1(w_1 \cdot x + b_1)\ )\ )$$



⇒ to learn the MLP $y = f_{MLP}(x;\theta)$ means that the MLP parameters $\theta = \{f, w, b\}$ need to be optimized to best fit the training dataset $\{x,\ y\}$, where:

- f = node activation functions *(some activation functions have parameters, e.g. the Leaky ReLU $f(x) = max(\alpha x, x)$)*
- w = node weights
- b = node biases

⇒ to do so we need to compute the partial derivatives of the loss function $\mathcal{L}$ with respect to θ:

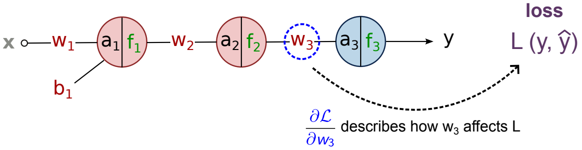
$$\frac{\partial \mathcal{L}}{\partial \theta} = \left[\frac{\partial \mathcal{L}}{\partial w_3}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}\right]$$

Using backpropagation to learn MLP

Let's compute the partial derivatives $\left[\frac{\partial \mathcal{L}}{\partial w_3}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}\right]$:

1. compute $\frac{\partial \mathcal{L}}{\partial w_3}$:

   $\Rightarrow$ the partial derivative $\frac{\partial \mathcal{L}}{\partial w_3}$ describes how w_3 will affect the Loss function L

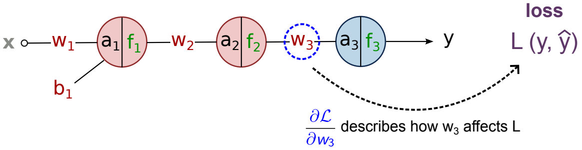


Using backpropagation to learn MLP

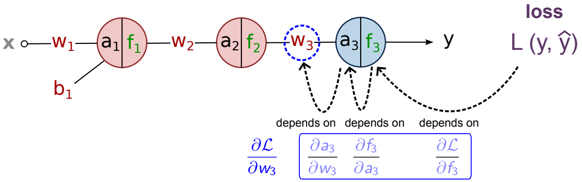
Let's compute the partial derivatives $\left[\frac{\partial \mathcal{L}}{\partial w_3}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}\right]$:

1. compute $\frac{\partial \mathcal{L}}{\partial w_3}$:

   $\Rightarrow$ the partial derivative $\frac{\partial \mathcal{L}}{\partial w_3}$ describes how w_3 will affect the Loss function L



   $\Rightarrow$ according to the chain rule: $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3}$



Using backpropagation to learn MLP

Let's compute the partial derivatives $\left[\frac{\partial \mathcal{L}}{\partial w_3}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}\right]$:

1. compute $\frac{\partial \mathcal{L}}{\partial w_3}$ *(continued)*:

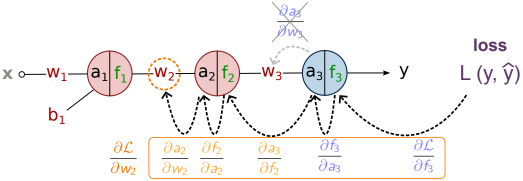
   $\Rightarrow$ we can compute $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3}$ as follows:

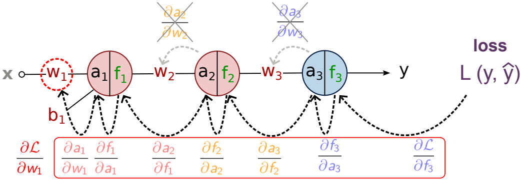
$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_3} &= \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3} \\
&= (y-\hat{y})\frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial w_3} \quad \text{considering } \mathcal{L} = \text{L2 loss} = \frac{1}{2}(y-\hat{y})^2 \Rightarrow \frac{\partial \mathcal{L}}{\partial y} = (y-\hat{y}) \\
&= (y-\hat{y})f_3(1-f_3)\frac{\partial a_3}{\partial w_3} \quad \text{considering } f_3 \text{ is a sigmoid function whose derivative is } f' = f(x)(1-f(x)) \\
&= (y-\hat{y})f_3(1-f_3)f_2 \quad \text{considering } a_3 = w_3 \cdot f_2
\end{aligned}$$

Using backpropagation to learn MLP

Let's compute the partial derivatives $\left[\frac{\partial \mathcal{L}}{\partial w_3}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}\right]$:

1. already computed: $\frac{\partial \mathcal{L}}{\partial w_3} = (y - \hat{y}) f_3 (1 - f_3) f_2$
2. compute: $\frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}$ by re-using already computed derivates (backpropagate!)





$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial w_3}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial f_2}\frac{\partial f_2}{\partial a_2}\frac{\partial a_2}{\partial w_2}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial f_2}\frac{\partial f_2}{\partial a_2}\frac{\partial a_2}{\partial f_1}\frac{\partial f_1}{\partial a_1}\frac{\partial a_1}{\partial w_1}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial f_3}\frac{\partial f_3}{\partial a_3}\frac{\partial a_3}{\partial f_2}\frac{\partial f_2}{\partial a_2}\frac{\partial a_2}{\partial f_1}\frac{\partial f_1}{\partial a_1}\frac{\partial a_1}{\partial b}$$

Summary of the procedure to train the MLP

$\Rightarrow$ for each training sample $\{x_i, y_i\}$:

1. Predict
   - forward pass: compute the output of the network
   $$\hat{y}_i = f_{MLP}(x_i; \theta)$$
   - compute Loss $\mathcal{L}$: compare the predicted output with the true output
   $$\mathcal{L} = \text{loss}(\hat{y}_i, y_i)$$
2. Update weights
   - backpropagation: compute the gradients of the loss function with respect to the network parameters θ
   $$\frac{\partial \mathcal{L}}{\partial \theta} = \left[\frac{\partial \mathcal{L}}{\partial w_3}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b}\right]$$
   - update weights: use the gradients to update the weights of the network
   $$w_3 = w_3 - \eta \nabla w_3 \quad \text{where } \nabla w_3 = \frac{\partial \mathcal{L}}{\partial w_3}$$
   $$w_2 = w_2 - \eta \nabla w_2 \quad \text{where } \nabla w_2 = \frac{\partial \mathcal{L}}{\partial w_2}$$
   $$w_1 = w_1 - \eta \nabla w_1 \quad \text{where } \nabla w_1 = \frac{\partial \mathcal{L}}{\partial w_1}$$
   $$b = b - \eta \nabla b \quad \text{where } \nabla b = \frac{\partial \mathcal{L}}{\partial b}$$

Choice of activation functions

⇒ recall that the perceptron used the **Step** function

⇒ issue: derivative is 0 almost everywhere, so gradients vanish and gradient descent can't learn the weights

⇒ other activation functions have been proposed for backpropagation to work:

- **Sigmoid / Logistic**

$$f(\mathbf{x})_i = \frac{1}{1+e^{-\mathbf{x}_i}}$$

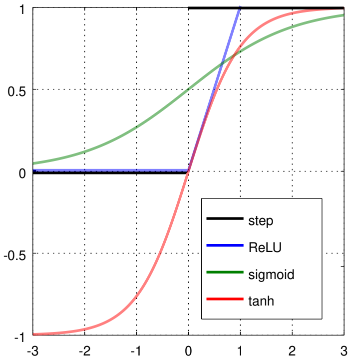
- **TanH**

$$f(\mathbf{x})_i = tanh(\mathbf{x}_i) = \frac{e^{\mathbf{x}_i} - e^{-\mathbf{x}_i}}{e^{\mathbf{x}_i} + e^{-\mathbf{x}_i}}$$

- **ReLU** (Rectified Linear Unit)

$$f(\mathbf{x})_i = max(\mathbf{x}_i, 0)$$

Note: sigmoid/tanh are older (saturate); ReLU is now preferred (speed and stronger gradients)



Choice of Loss functions (what we minimize)

⇒ the loss $\mathcal{L}$ measures the mismatch between predictions $\hat{y}$ and targets y

⇒ choice depends on the task type:

- **Regression** (continuous target):
  - L2 loss → Mean Squared Error (MSE)
  - L1 loss → Mean Absolute Error (MAE)
- **Binary classification**:
  - Hinge loss (used in SVM)
  - Binary Cross-Entropy (BCE) (used in neural networks)
- **Multiclass classification**:
  - Categorical Cross-Entropy
  - Sparse Categorical Cross-Entropy

⇒ in practice, we often prefer differentiable losses to enable gradient-based optimization

Choice of Gradient Descent (how we minimize)

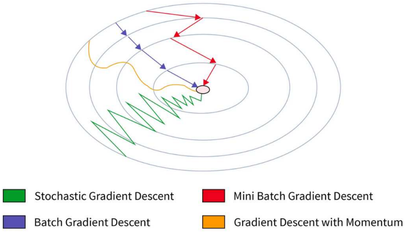
$\Rightarrow$ the loss $\mathcal{L}$ is minimized using iterative updates

$\Rightarrow$ several algorithms exist which calculate gradient $\nabla\mathcal{L}$ differently:

- **Batch Gradient Descent**
  - uses the full dataset to compute $\nabla\mathcal{L}$
  - stable but computationally expensive
- **Stochastic Gradient Descent (SGD)**
  - updates using one sample at a time
  - faster, but noisy updates
- **Mini-batch Gradient Descent**
  - uses small batches of data
  - good trade-off between speed and stability

$\Rightarrow$ variants modify the update rule:

- **Momentum**
  - accumulates past gradients
  - smoother, faster convergence
- **Adam**
  - adaptive learning rates + momentum
  - widely used in deep learning



source

MLP from scratch

$\Rightarrow$ several deep learning frameworks exist to create complex neural networks with different architectures (discussed later)

$\Rightarrow$ however it is useful to build a simple MLP from scratch to understand the fundamentals

$\Rightarrow$ see the great tutorial by Shan-Hung Wu at NTHU: Neural Networks from Scratch

MLP playground

⇒ build your own MLP using the interactive web platform developped by Google: playground.tensorflow.org

⇒ see the effects of training in real time!

Several frameworks exist:

- Tensor Flow
  - developped by Google

Several frameworks exist:

- Tensor Flow
  - developped by Google
  - includes the high-level API Keras library (TF version ≥ 2)

Several frameworks exist:

- Tensor Flow
  - developped by Google
  - includes the high-level API Keras library (TF version $\geq$2)
- PyTorch
  - developed by Facebook

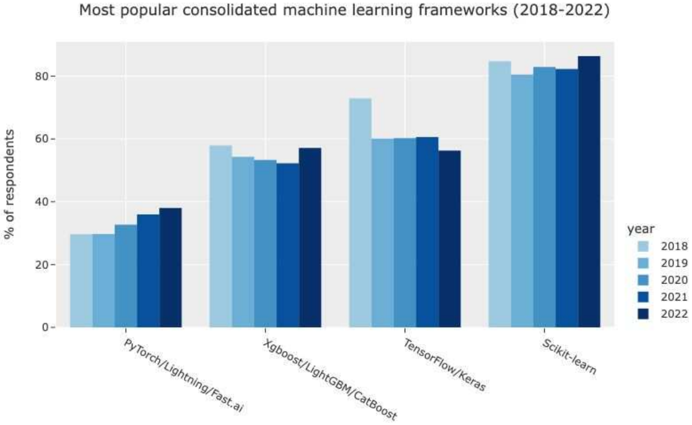
Several frameworks exist:

- Tensor Flow
  - developped by Google
  - includes the high-level API Keras library (TF version $\geq$2)
- PyTorch
  - developed by Facebook
  - based on the Torch framework (Lua)

Several frameworks exist:

- Tensor Flow
  - developped by Google
  - includes the high-level API Keras library (TF version $\geq$2)
- PyTorch
  - developed by Facebook
  - based on the Torch framework (Lua)
- MindSpore, Caffe, MXNet, Theano, etc.

Popularity of the main frameworks (from Kaggle 2022[1])



[1] 2022 - Kaggle Data Science & ML Survey (link)

Differences between PyTorch & Tensor Flow

⇒ some time back, both frameworks had major differences in terms of design, paradigm, syntax, etc.

⇒ nowadays, the practical differences are much smaller

PyTorch

```python
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2d(in_channels=1,
                            out_channels=32,
                            kernel_size=3)
        self.flatten = Flatten()
        self.d1 = Linear(21632, 128)
        self.d2 = Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.flatten(x)
        x = F.relu(self.d1(x))
        x = self.d2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

TensorFlow

```python
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2D(filters=32,
                            kernel_size=3,
                            activation='relu')
        self.flatten = Flatten()
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(10)

    def call(self, x):
        x = self.conv1(x)
        x = self.flatten(x)
        x = self.d1(x)
        output = self.d2(x)
        return output
```

(from Shan-Hung Wu, NTHU CS565600)

Differences between PyTorch & Tensor Flow

⇒ some time back, both frameworks had major differences in terms of design, paradigm, syntax, etc.

⇒ nowadays, the practical differences are much smaller

PyTorch

```python
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2d(in_channels=1,
                            out_channels=32,
                            kernel_size=3)
        self.flatten = Flatten()
        self.d1 = Linear(21632, 128)
        self.d2 = Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.flatten(x)
        x = F.relu(self.d1(x))
        x = self.d2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

TensorFlow

```python
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2D(filters=32,
                            kernel_size=3,
                            activation='relu')
        self.flatten = Flatten()
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(10)

    def call(self, x):
        x = self.conv1(x)
        x = self.flatten(x)
        x = self.d1(x)
        output = self.d2(x)
        return output
```

(from Shan-Hung Wu, NTHU CS565600)

⇒ we'll be using mostly TensorFlow since it is integrated in Google Colab

Installing Tensor Flow with Anaconda (instructions):

⇒ for those working in local environment, follow these steps to install Tensor Flow in a "conda environment"

1. Create environment and install Tensor Flow package & dependencies inside

```
$ conda env list                  # optional: list existing environments
$ conda create -n tf tensorflow   # create environment called "tf" & install CPU-only TensorFlow
```

2. Activate the created environment

```
$ conda activate tf
```

3. Install additional packages in the active environment

```
$ conda install jupyter matplotlib pandas scikit-learn
```

4. Launch Jupyter from the active environment, import Tensor Flow, and you're good to go!

```
$ jupyter notebook
```

```python
# Create a new notebook with Python 3 kernel
import tensorflow as tf
```

4.4. installing Tensor Flow

Nota Bene

Two distinct versions of TF exist, depending on whether it should run on CPU (Central Processing Unit), or GPU (Graphics Processing Unit)

$\Rightarrow$ CPU-only TensorFlow (recommended for beginners)

```
$ conda create -n tf tensorflow
```

$\Rightarrow$ GPU TensorFlow

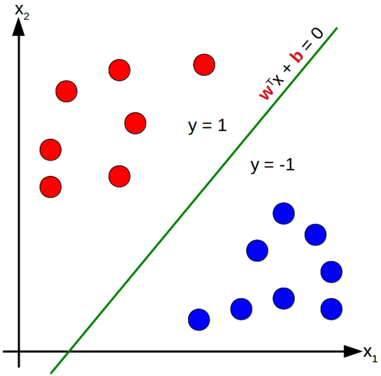
```
$ conda create -n tf-gpu tensorflow-gpu
```

$\Rightarrow$ GPU will be much faster, but more expensive, and trickier to setup (requires CUDA)

Toy example: linear classification task using scikit-learn and tensor flow

perceptron: $\boxed{\hat{y} = \text{sign}(\mathbf{w}^T \mathbf{x} + b)}$

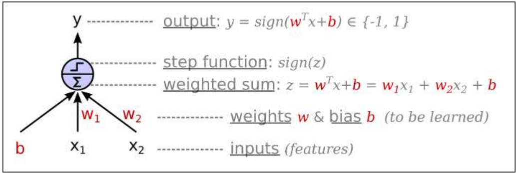
- $\hat{y} \in \{-1|1\}$: predicted label $\rightarrow$ *banana | apple*
- $\mathbf{x} \in \mathbb{R}^2$: feature vector $\rightarrow$ *[hue, elongation]*
- $\mathbf{w} \in \mathbb{R}^2$: **weight vector** $\rightarrow$ needs to be learned
- $b \in \mathbb{R}$: **bias** $\rightarrow$ needs to be learned
- *sign*: sign function returning the sign of a real number



Toy example: linear classification task using scikit-learn and tensor flow

perceptron: $\boxed{\hat{y} = \text{sign}(\mathbf{w}^T\mathbf{x} + b)}$

- $\hat{y} \in \{-1|1\}$: predicted label $\rightarrow$ *banana | apple*
- $\mathbf{x} \in \mathbb{R}^2$: feature vector $\rightarrow$ *[hue, elongation]*
- $\mathbf{w} \in \mathbb{R}^2$: **weight vector** $\rightarrow$ needs to be learned
- $b \in \mathbb{R}$: **bias** $\rightarrow$ needs to be learned
- *sign*: sign function returning the sign of a real number



Solution with Scikit-Learn: Perceptron classifier

```
from sklearn import datasets
from sklearn import linear_model
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

```
# Load data
iris = datasets.load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype('int') # Iris setosa?
```

1.1 Load data

```
# Preprocess data
X, y = shuffle(X, y, random_state=0)
scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train = X[:75]
y_train = y[:75]
X_test = X[75:]
y_test = y[75:]
```

1.2 Preprocess data
- shuffle
- scale
- split into train/test

```
# Select model
clf = linear_model.Perceptron()
```

2. Select model

```
# Train model
clf.fit(X_train, y_train)

print('weights:', clf.coef_)
print('bias:', clf.intercept_)
```

3. Train model

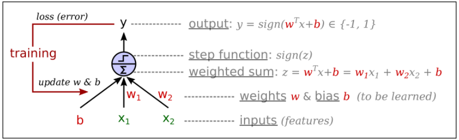
```
# Evaluate
y_pred = clf.predict(X_train)
accuracy_score(y_train, y_pred)
```

4. Evaluate model

```
# Predict from model
y_pred = clf.predict([[2, 0.5]])
```

5. Predict from model

```
# Plot data + linear classifier
#plt.scatter(X[:,0], X[:,1], c=y)
plt.scatter(X_train[:,0], X_train[:,1], c=y_train)
plt.scatter(X_test[:,0], X_test[:,1], c=y_test, alpha=.25)

weights = clf.coef_[0]
bias = clf.intercept_
slope = -weights[0] / weights[1]
yintercept = -bias / weights[1]
_x = np.linspace(-2,2)
_y = slope*_x + yintercept
plt.plot(_x, _y, '-r')
```



4.5. from ML (sklearn) to DL (tensorflow)

Solution with Scikit-Learn: Perceptron classifier

```
from sklearn import datasets
from sklearn import linear_model
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

```
# Load data
iris = datasets.load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype('int') # Iris setosa?
```

1.1 Load data

```
# Preprocess data
X, y = shuffle(X, y, random_state=0)
scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train = X[:75]
y_train = y[:75]
X_test = X[75:]
y_test = y[75:]
```

1.2 Preprocess data
- shuffle
- scale
- split into train/test

```
# Select model
clf = linear_model.Perceptron()
```

2. Select model

```
# Train model
clf.fit(X_train, y_train)

print('weights:', clf.coef_)
print('bias:', clf.intercept_)
```

3. Train model

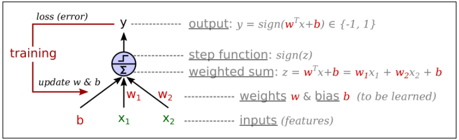
```
# Evaluate
y_pred = clf.predict(X_train)
accuracy_score(y_train, y_pred)
```

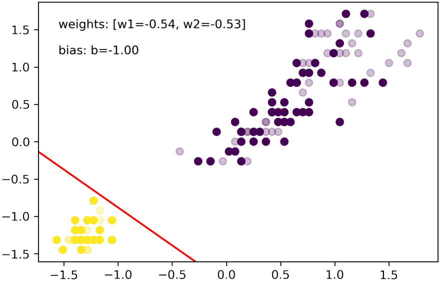
4. Evaluate model

```
# Predict from model
y_pred = clf.predict([[2, 0.5]])
```

5. Predict from model

```
# Plot data + linear classifier
#plt.scatter(X[:,0], X[:,1], c=y)
plt.scatter(X_train[:,0], X_train[:,1], c=y_train)
plt.scatter(X_test[:,0], X_test[:,1], c=y_test, alpha=.25)

weights = clf.coef_[0]
bias = clf.intercept_
slope = -weights[0] / weights[1]
yintercept = -bias / weights[1]
_x = np.linspace(-2,2)
_y = slope*_x + yintercept
plt.plot(_x, _y, '-r')
```





Solution with Tensor Flow - Keras: 1 neuron network

```
import tensorflow as tf
from sklearn import datasets
from sklearn import linear_model
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

```
# Load data
iris = datasets.load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype('int') # Iris setosa?
```

1.1 Load data

```
# Preprocess data
X, y = shuffle(X, y, random_state=0)
scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train = X[:75]
y_train = y[:75]
X_test = X[75:]
y_test = y[75:]
```

1.2 Preprocess data
- shuffle
- scale
- split into train/test

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(2,)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.summary()
```

2.1 Build model
- set layer type/order

```
# Compile model
model.compile(optimizer='sgd',
              loss='BinaryCrossentropy',
              metrics=['accuracy'])
```

2.2 Compile model
- set loss function
- set optimizer
- set metrics

```
# Train model
history = model.fit(X_train, y_train, epochs=50) #, batch_size=10)
```

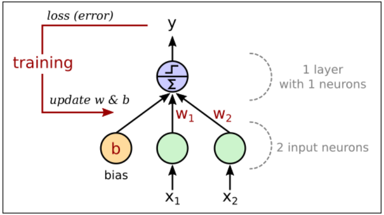
3. Train model
- learn layer parameters (weights/biases)
- plot training history (check for overfitting)

```
# Evaluate model
test_loss, test_acc = model.evaluate(X_test,  y_test, verbose=2)
print('Test accuracy:', test_acc)
```

4. Evaluate model
- evaluate accuracy on test dataset

```
# Predict (data should be preprocessed just like training data)
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
pred = probability_model.predict([[-2, -2]])
print(pred)
```

5. Predict from model
- predict image class using learned model




```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 2)                 0
_________________________________________________________________
dense (Dense)                (None, 1)                 3
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

y

So if we can do the same thing, why switch from **sklearn** to **tensor flow** ?

Tensor Flow is a framework for **Deep Learning**

$\Rightarrow$ can design multi-layered networks, and train them in a very flexible/optimized manner

$\Rightarrow$ can solve much more complex problems, by optimizing several thousands/millions of weights during training!

So if we can do the same thing, why switch from **sklearn** to **tensor flow** ?

Tensor Flow is a framework for **Deep Learning**

⇒ can design multi-layered networks, and train them in a very flexible/optimized manner

⇒ can solve much more complex problems, by optimizing several thousands/millions of weights during training!

"Hello World" example in Keras TensorFlow: MNIST fashion dataset classification task with MLP

```
import tensorflow as tf

# Load data
fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

1.1 Load data
- training dataset
- validation dataset
- test dataset

```
# Preprocess data
X_train, X_test, X_valid = X_train/255.0, X_test/255.0, X_valid/255.0
```

1.2 Preprocess data
- scale pixel intensities to 0-1

```
# Build model (using the Sequential API)
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
model.summary()
```

2.1 Build model
- set layer type/order

```
# Compile model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

2.2 Compile model
- set loss function
- set optimizer
- set metrics

```
# Train model
history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                    epochs=30, # nb of times X_train is seen seen
                    batch_size=32) # nb of images per training instance
print('training instances per epoch = {}'.format(X_train.shape[0] / 32))

# Plot training history
import pandas as pd
pd.DataFrame(history.history).plot()
```

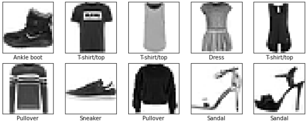
3. Train model
- learn layer parameters (weights/biases)
- plot training history (check for overfitting)

```
# Evaluate model
test_loss, test_acc = model.evaluate(X_test,  y_test)
print('Test accuracy:', test_acc)
```

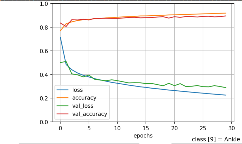
4. Evaluate model
- evaluate accuracy on test dataset

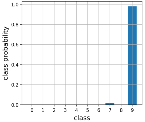
```
# Predict
img = X_test[0,:,:]
img = (np.expand_dims(img,0)) # add image to a batch
y_proba = model.predict(img).round(2)
y_pred = np.argmax(model.predict(img), axis=-1)

plt.bar(range(10), y_proba[0])
plt.imshow(img[0,:,:], cmap='binary')
plt.title('class {} = {}'.format(y_pred, class_names[np.argmax(y_proba)]))
```

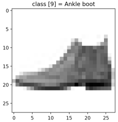
5. Predict from model
- predict image class using learned model




```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```







"Hello World" example in Keras TensorFlow: MNIST fashion dataset classification task with MLP

```
import tensorflow as tf
```

```
# Load data
fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

1.1 Load data
- training dataset
- validation dataset
- test dataset

```
# Preprocess data
X_train, X_test, X_valid = X_train/255.0, X_test/255.0, X_valid/255.0
```

1.2 Preprocess data
- scale pixel intensities to 0–1

```
# Build model (using the Sequential API)
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
model.summary()
```

2.1 Build model
- set layer type/order

```
# Compile model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

2.2 Compile model
- set loss function
- set optimizer
- set metrics

```
# Train model
history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                    epochs=30, # nb of times X_train is seen seen
                    batch_size=32) # nb of images per training instance
print('training instances per epoch = {}'.format(X_train.shape[0] / 32))

# Plot training history
import pandas as pd
pd.DataFrame(history.history).plot()
```

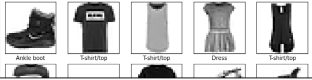
3. Train model
- learn layer parameters (weights/biases)
- plot training history (check for overfitting

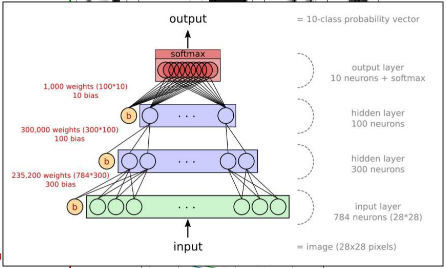
```
# Evaluate model
test_loss, test_acc = model.evaluate(X_test,  y_test)
print('Test accuracy:', test_acc)
```

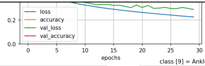
4. Evaluate model
- evaluate accuracy on test dataset

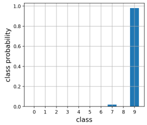
```
# Predict
img = X_test[0,:,:]
img = (np.expand_dims(img,0)) # add image to a batch
y_proba = model.predict(img).round(2)
y_pred = np.argmax(model.predict(img), axis=-1)

plt.bar(range(10), y_proba[0])
plt.imshow(img[0,:,:], cmap='binary')
plt.title('class {} = {}'.format(y_pred, class_names[np.argmax(y_proba)]))
```

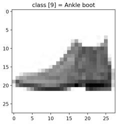
5. Predict from model
- predict image class using learned model











Glossary

Key parameters and definitions (from Google's ML glossary, Chollet 2017, etc.)

- **loss function** (objective function)

  The quantity that will be minimized during training. It represents a measure of success for the task at hand.

- **optimizer**

  Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).

- **accuracy**

  The fraction of predictions that a classification model got right.

- **epoch**

  Each iteration over all the training data.

- **batch_size**

  Number of samples per gradient update.

- **activation function**

  A function (for example, ReLU or sigmoid) that takes in the weighted sum of all of the inputs from the previous layer and then generates and passes an output value (typically nonlinear) to the next layer.

- **softmax**

  A function that provides probabilities for each possible class in a multi-class classification model. The probabilities add up to exactly 1.0. For example, softmax might determine that the probability of a particular image being a dog at 0.9, a cat at 0.08, and a horse at 0.02.