ME 536

Week 12: How deep is your love* *to learn* ?

* old times were very funny... what do you think today will look like in 45 years?

Learning:

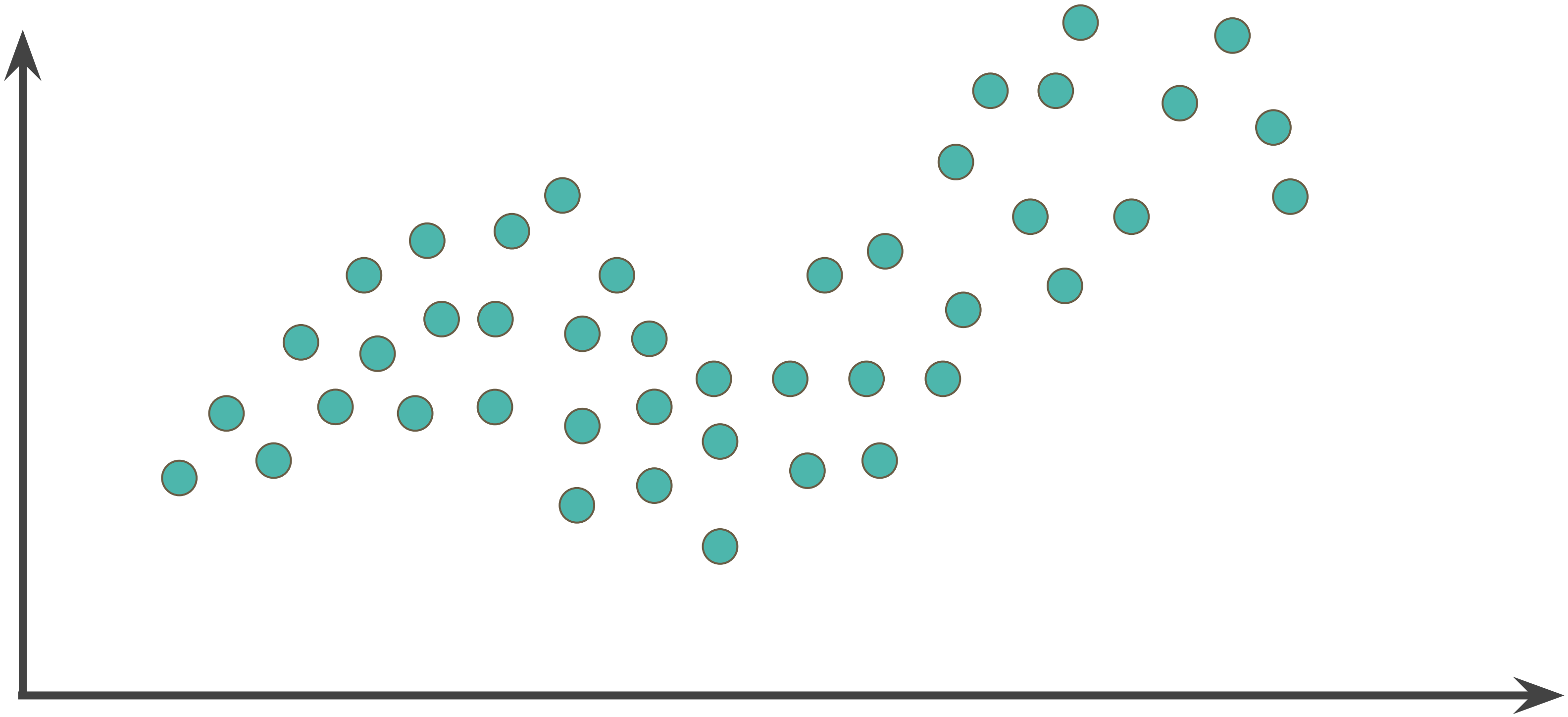
What to learn ?

How to learn ?

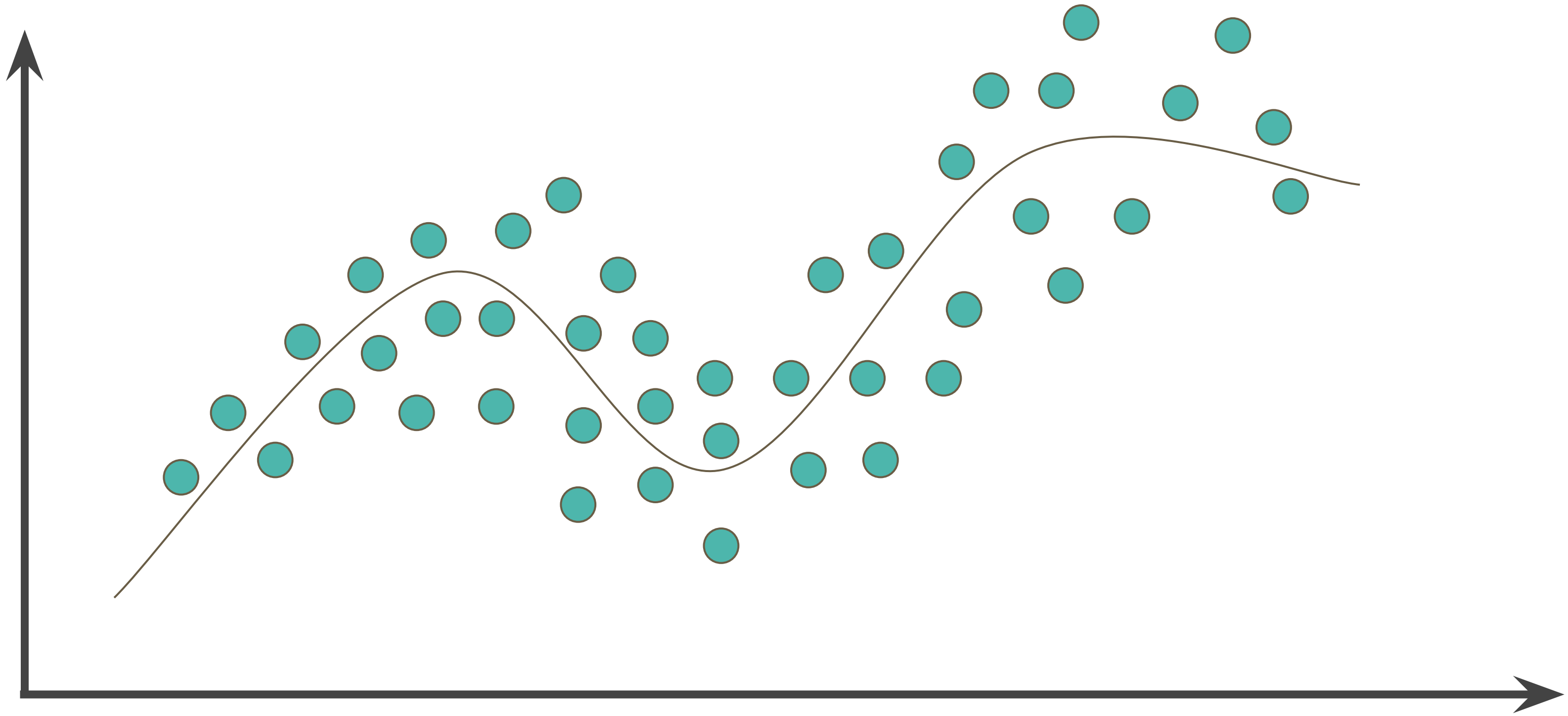
Learn once or life-long learning ?

...

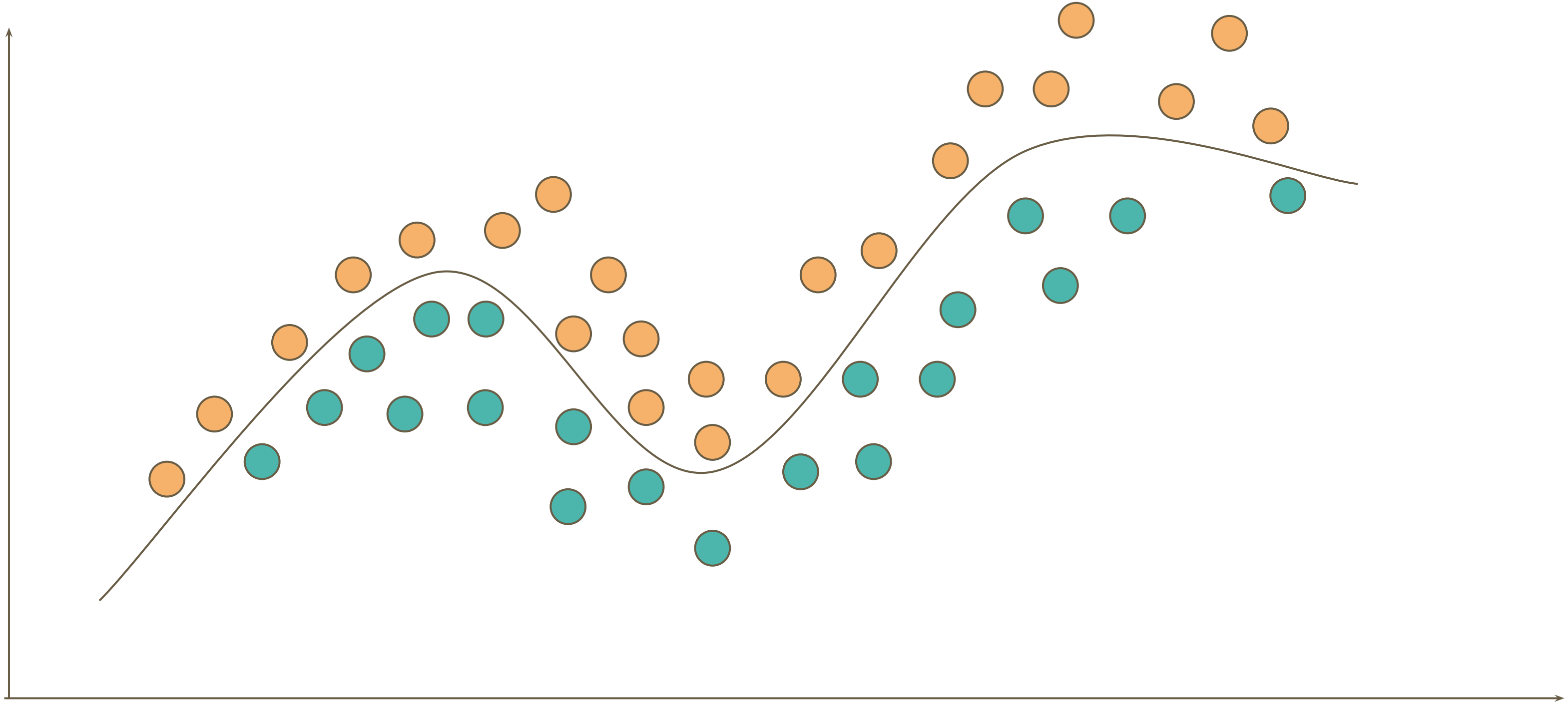
DATA



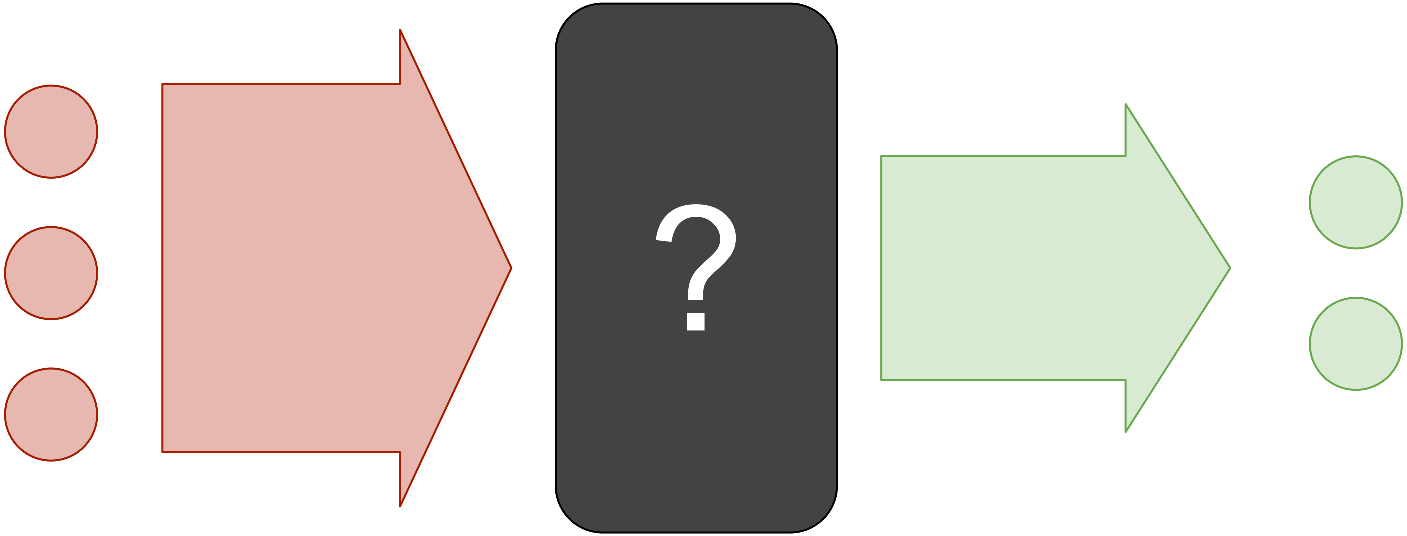
Regression



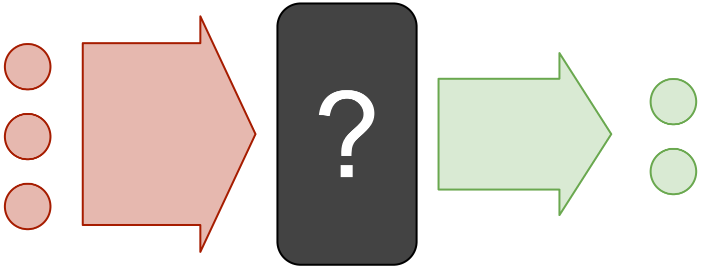
or Classification?



A black box: In general

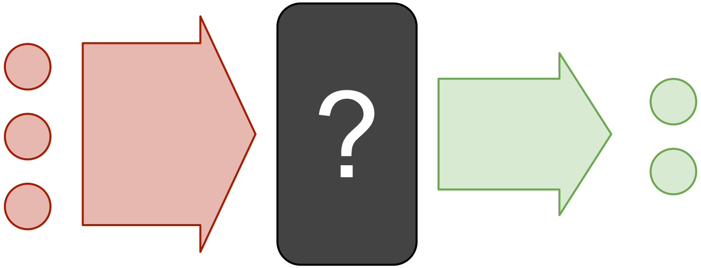


A black box: In general



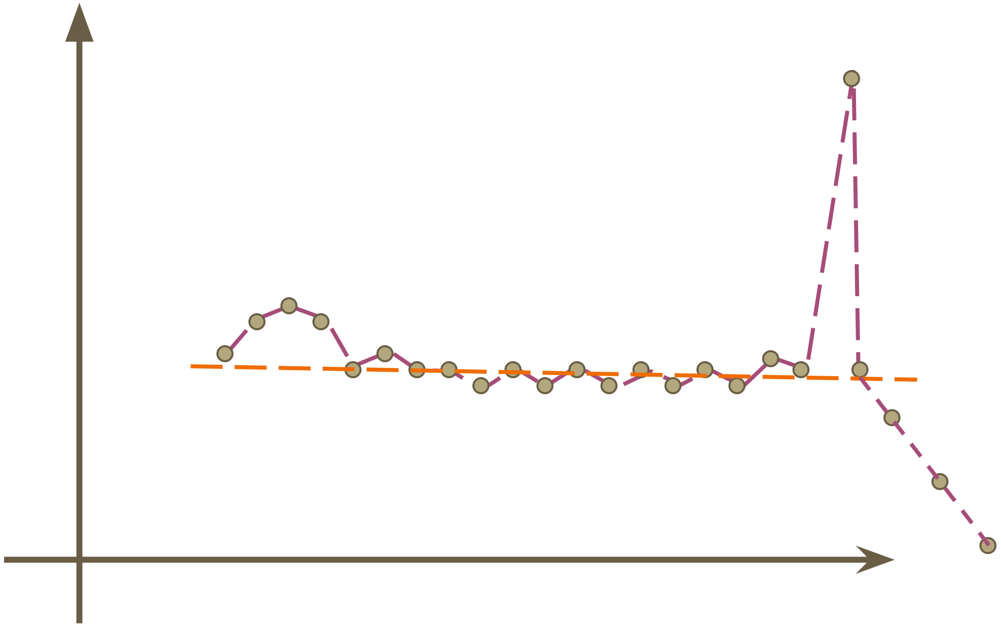
- Data needed
- Type of model available ?
  - What if model is not known at all?
  - What if the model is hard to derive?
- What if all inputs do not propagate at the same speed to output?
- What if data is available?
  - What if whole input spectrum is not covered in data?
- ...
- Does nature help?

A black box: In general



While blackbox is identified:

- Can it generalize?
- Over- / under-fitting ... /learning



If this is from the step response of a second order system? Why use an ANN?

Rules of thumb

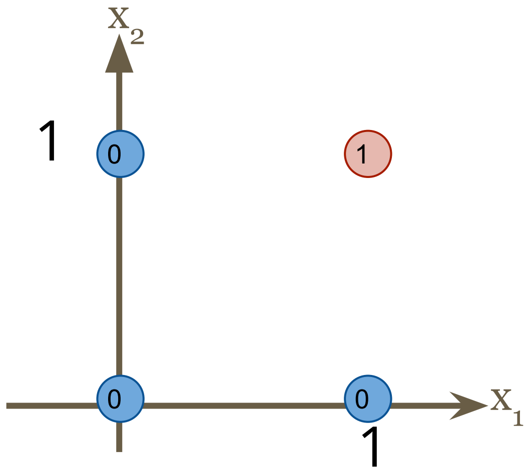
- Do not expect **miracles** from ANN and do not **blindly** use them.
- Direct use of ANNs without prior analysis might be **more costly** than expected.
- If you have **a good understanding of the model** why not identify the parameters and use the model?
- Analysis of why **ANNs misbehave** is tricky
- **Best ANN topology** to start with is **not** necessarily **known** given the problem.
- If you have partial prior **understanding of your model**, try to inject into the ANN if possible.

A simple case: logic AND

&	x_1 = 0	x_1 = 1
x_2 = 0	0	0
x_2 = 1	0	1

Just a line is enough...

Isn't it?



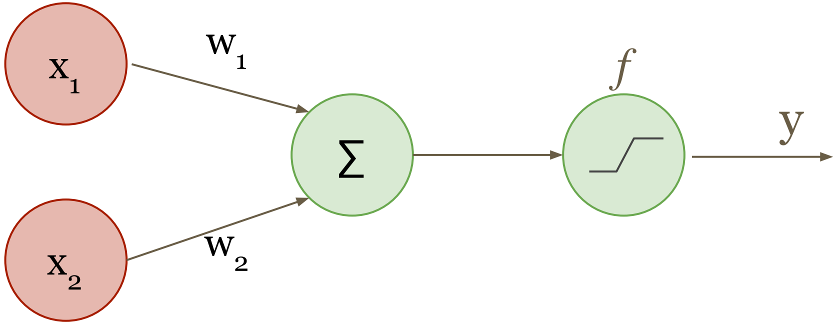
A simple case: perceptron w/o bias

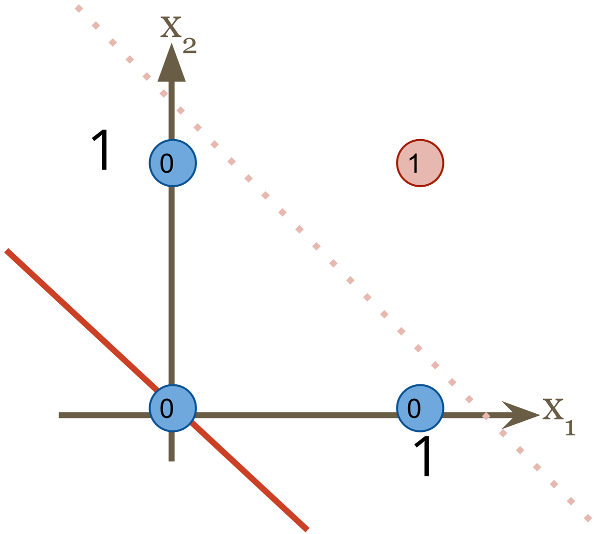
&	x_1 = 0	x_1 = 1
x_2 = 0	0	0
x_2 = 1	0	1

Just a line is enough...

Isn't it? or how about a scalar field?

$$y = f(x_1 w_1 + x_2 w_2)$$





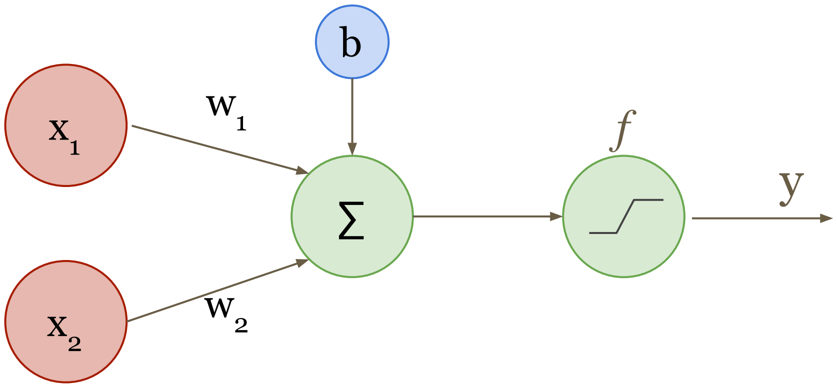
A simple case: perceptron

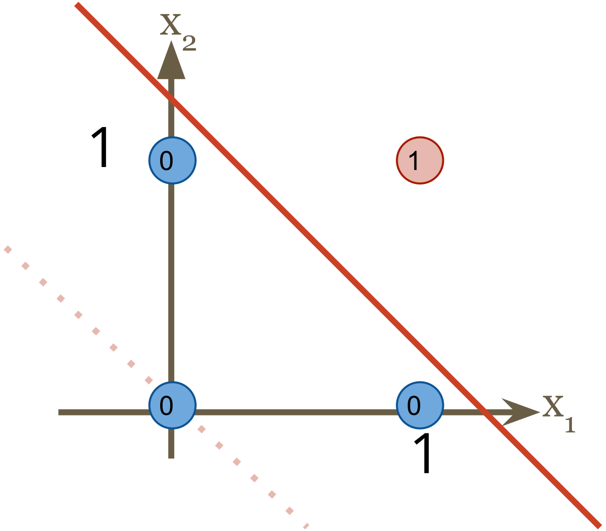
&	x_1 = 0	x_1 = 1
x_2 = 0	0	0
x_2 = 1	0	1

Just a line is enough...

Isn't it? or how about a scalar field?

$y = f(x_1 w_1 + x_2 w_2 + b)$ *where is the y axis?*





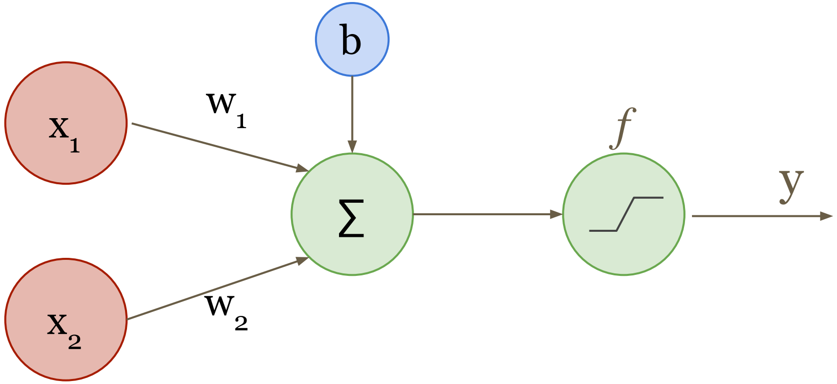
A simple case: More compact form - dimension free

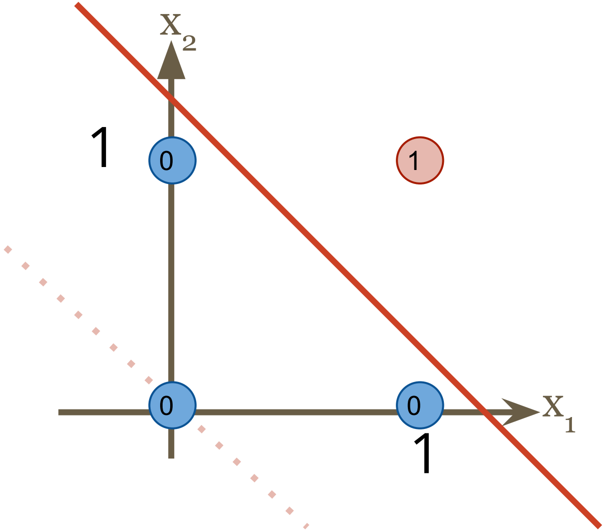
&	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0	1

$y = f(\, x_1 w_1 + x_2 w_2 + b)$

Let $\mathbf{x}^T = [x_1 \;\; x_2]$, $\mathbf{w}^T = [w_1 \;\; w_2]$ - *dimension of* $\mathbf{x}$, $\mathbf{w}$ *does not matter*

$y = f(\, \mathbf{x}^T \mathbf{w} + b)$





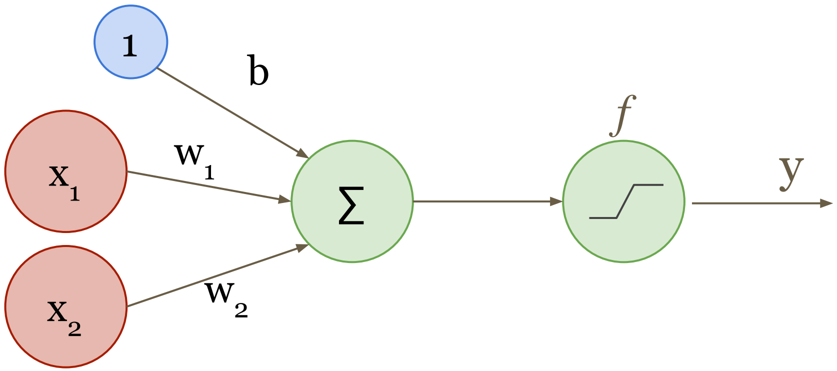
A simple case: Alternative form

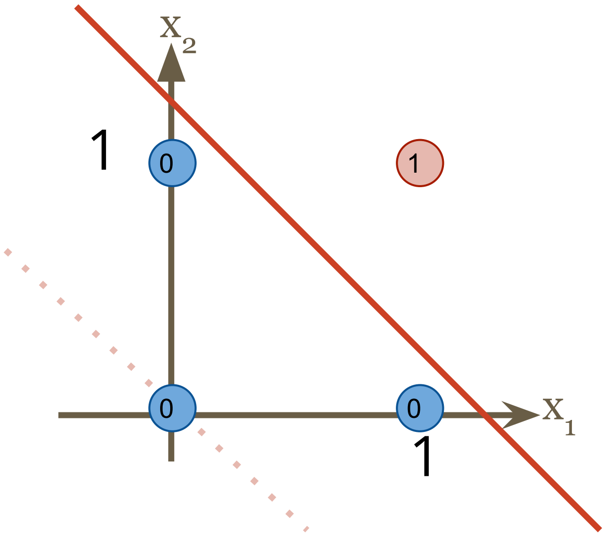
& (x_2 \ x_1)	0	1
0	0	0
1	0	1

$$y = f(x_1 w_1 + x_2 w_2 + b)$$

Let $\mathbf{x}^T = [1\ x_1\ \ x_2]$, $\mathbf{w}^T = [b\ w_1\ \ w_2]$

$$y = f(\mathbf{x}^T \mathbf{w})$$





A simple case: How to initialize w_i?

&	x_1 = 0	x_1 = 1
x_2 = 0	0	0
x_2 = 1	0	1

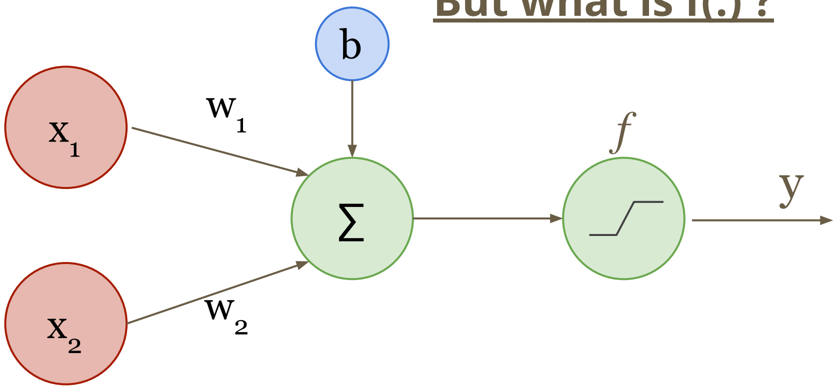
Random sounds good in general,

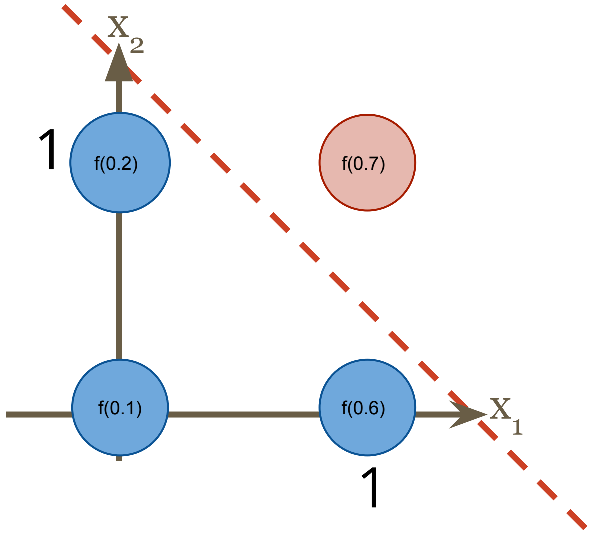
so let: $\mathbf{w}^T = [0.5 \;\; 0.1]$, $b = 0.1$

$y = f(\,\mathbf{x}^T\mathbf{w} + b) = f(\,0.5x_1 + 0.1\,x_2 + 0.1)$

<u>What should f(.) return?</u>

<u>But what is f(.) ?</u>





A simple case: How lucky can we get?

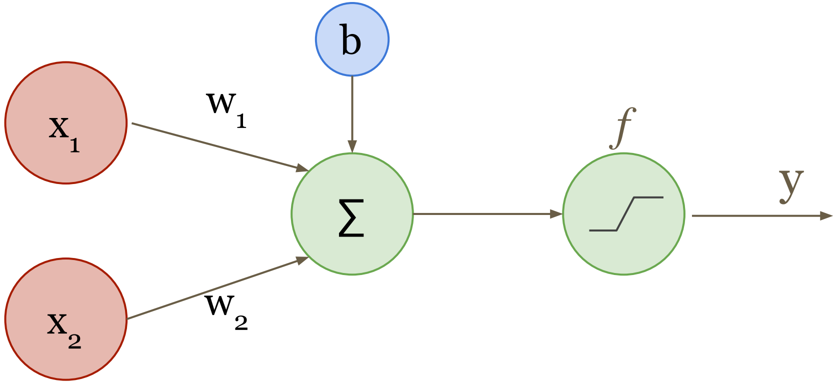
	&	x_1 = 0	x_1 = 1
x_2	0	0	0
	1	0	1

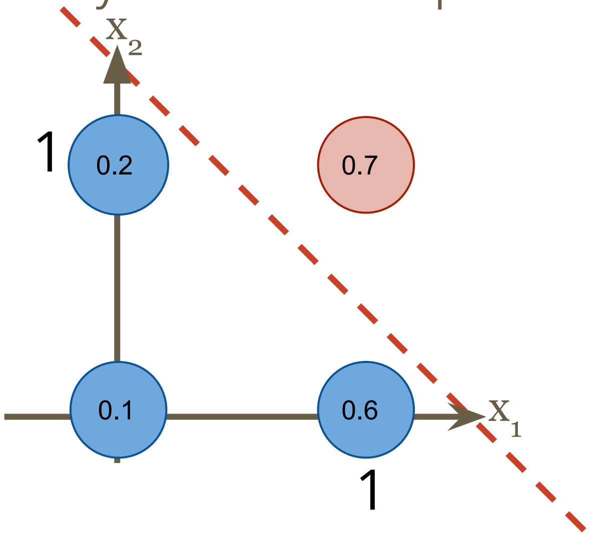
Random sounds good in general, so let: $\mathbf{w}^T = [0.5 \;\; 0.1]$, $b = 0.1$

$y = f(\mathbf{x}^T \mathbf{w} + b) = f(0.5x_1 + 0.1 x_2 + 0.1)$

What if $f(x) = x$ $\qquad y = 0.5\, x_1 + 0.1\, x_2 + 0.1$

Will simple case work here? May be with a bit of post-work?





A simple case: Which f ?

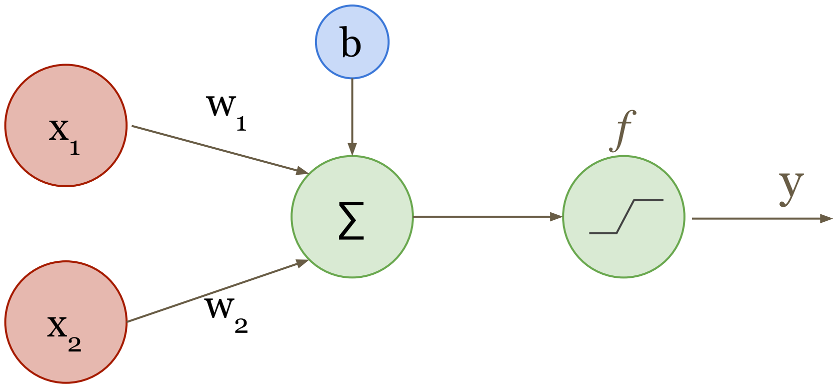
&	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0	1

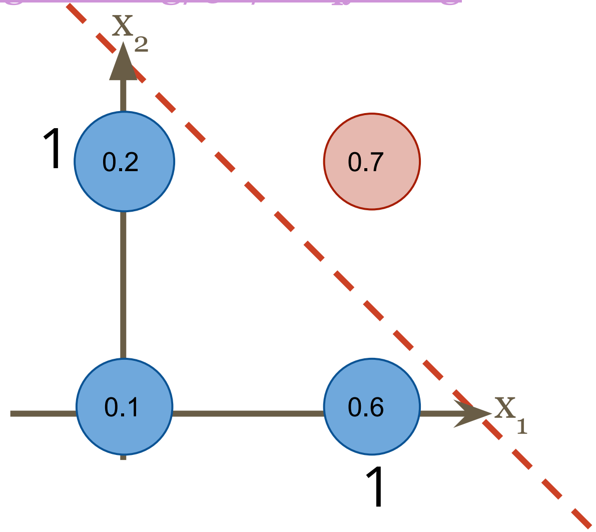
Let: $\mathbf{w}^T = [0.5 \;\; 0.1]$, $b = 0.1$

$y = f(\mathbf{x}^T \mathbf{w} + b) = f(0.5x_1 + 0.1x_2 + 0.1)$,

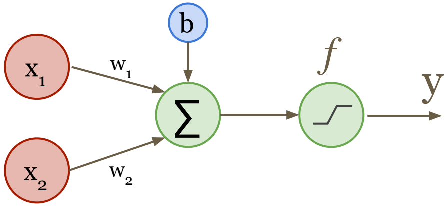
simple case $f(x) = x$, $\qquad y = 0.5x_1 + 0.1x_2 + 0.1$

Check out https://www.geogebra.org/3d/dbqykxwg





In general: Which f ?



Check out: https://www.geogebra.org/calculator/kzexwpwz

Sigmoid

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

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

Tangent Hyperbolic

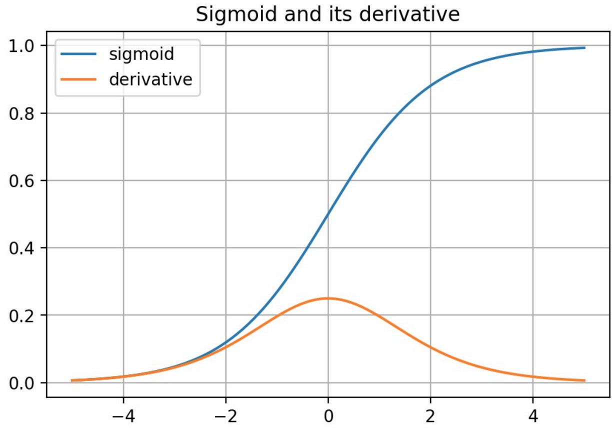
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

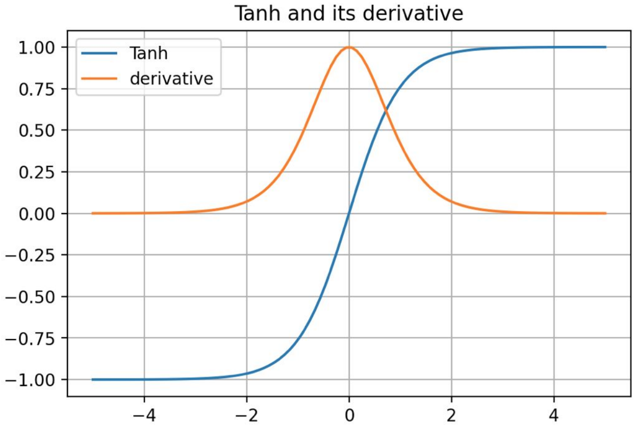
$$f'(x) = 1 - f(x)^2$$

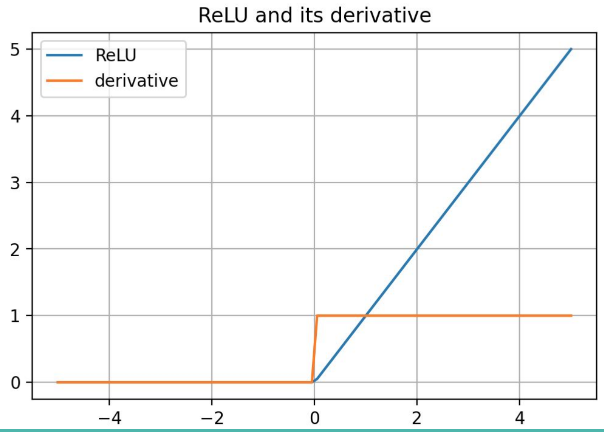
Rectified Linear Unit: a.k.a **ReLU**

$$f(x) = max(0, x)$$

$$f'(x) = \begin{cases} 1, & for x > 0 \\ 0, & otherwise \end{cases}$$







A simple case: Which f is ?

&	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0	1

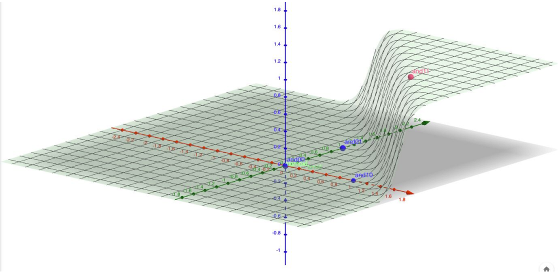
Let: $\mathbf{w}^T = [0.5 \;\; 0.1]$, $b = 0.1$

$y = f(\mathbf{x}^T \mathbf{w} + b) = f(0.5x_1 + 0.1\, x_2 + 0.1)$,

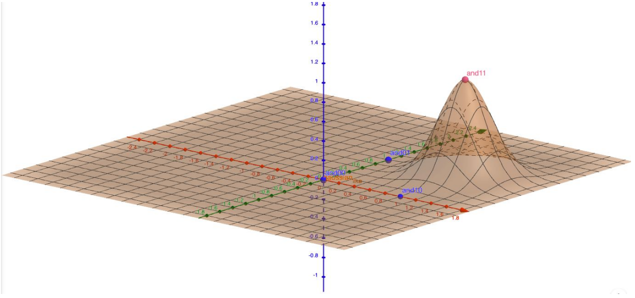
sigmoid case $f(x) = (1+e^{-x})^{-1}$,

Check out https://www.geogebra.org/3d/dbqykxwg

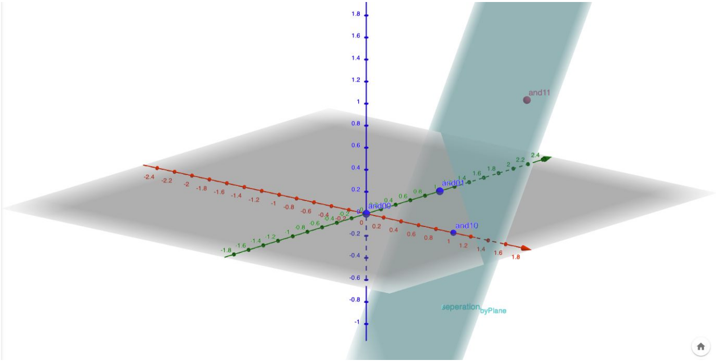
Sigmoid



Gaussian



Plane



A simple case: How to generate training data?

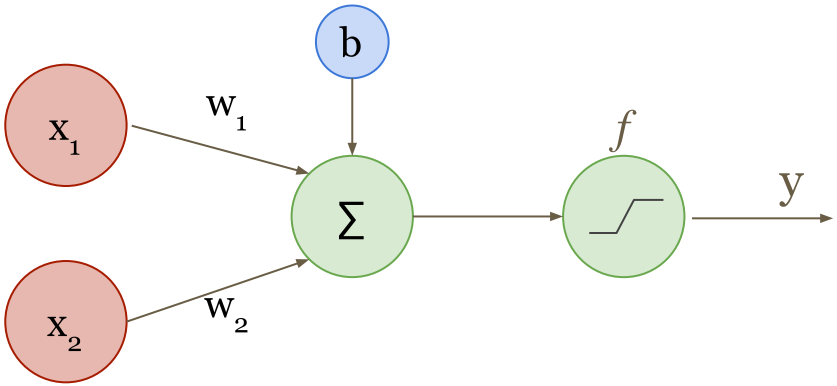
&	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0	1

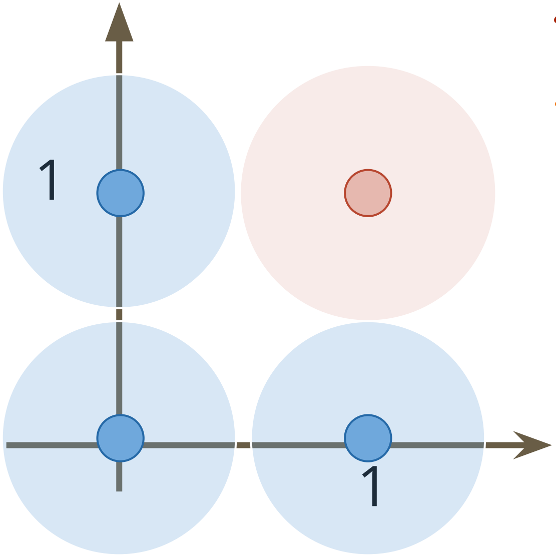
$$y = f(\, x_1 w_1 + x_2 w_2 + b)$$

Let $\mathbf{x}^T = [x_1 \;\; x_2]$, $\mathbf{w}^T = [w_1 \;\; w_2]$

$$y = f(\, \mathbf{x}^T \mathbf{w} + b)$$

x_1	x_2	y_t
0	0	0
0	1	0
1	0	0
1	1	1
?	?	?
…	…	…





A simple case: How to generate training data?

x_1

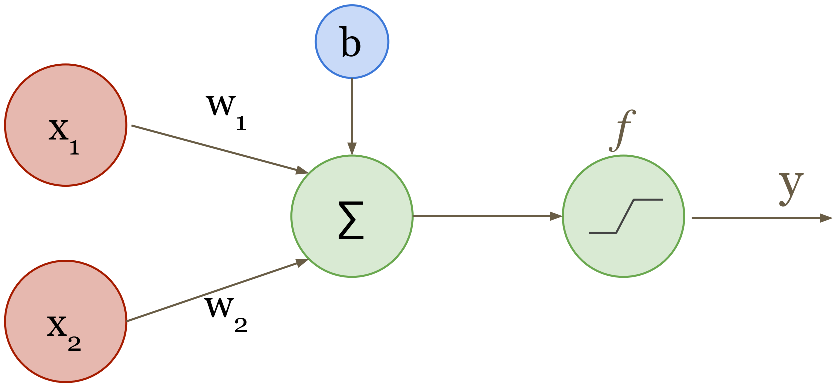
	&	0	1
x_2	0	0	0
	1	0	1

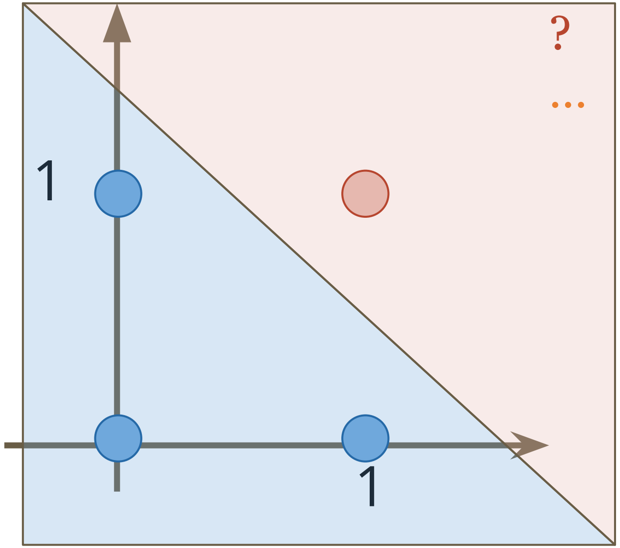
$$y = f(\, x_1 w_1 + x_2 w_2 + b)$$

Let $\mathbf{x}^T = [x_1 \;\; x_2]$, $\mathbf{w}^T = [w_1 \;\; w_2]$

$$y = f(\, \mathbf{x}^T \mathbf{w} + b)$$

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{y_t}$
0	0	0
0	1	0
1	0	0
1	1	1
?	?	?
...	...	...





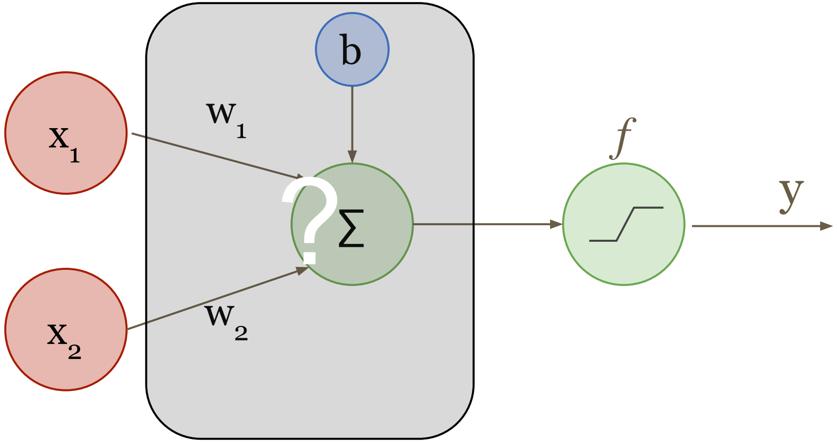
A *not so* simple case: when one line ain't enough

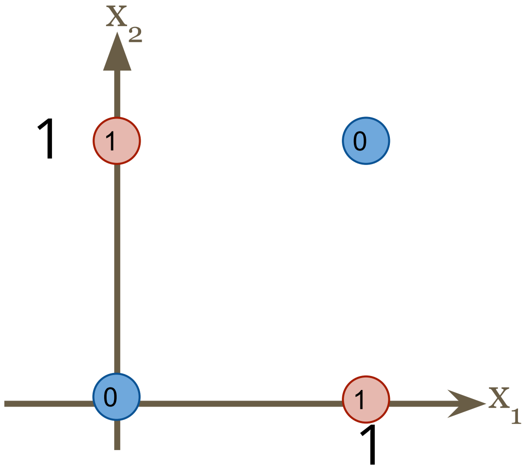
XOR	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	1
$x_2 = 1$	1	0

$$y = f(\, x_1 w_1 + x_2 w_2 + b)$$

Let $\mathbf{x}^T = [x_1 \;\; x_2]$, $\mathbf{w}^T = [w_1 \;\; w_2]$

$$y = f(\, \mathbf{x}^T \mathbf{w} + b)$$





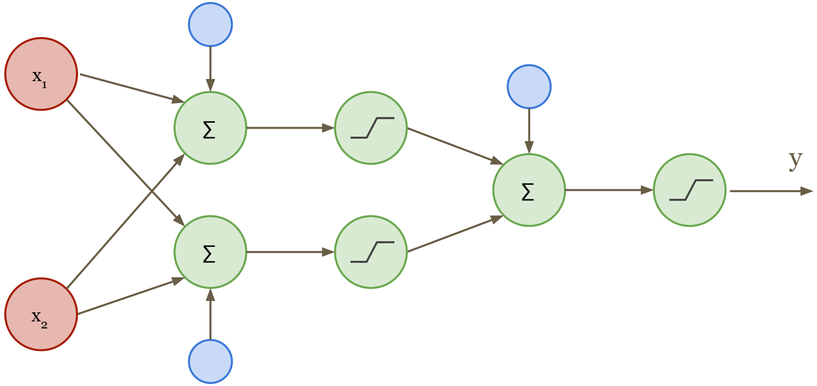
A *not so* simple case: when one line ain't enough

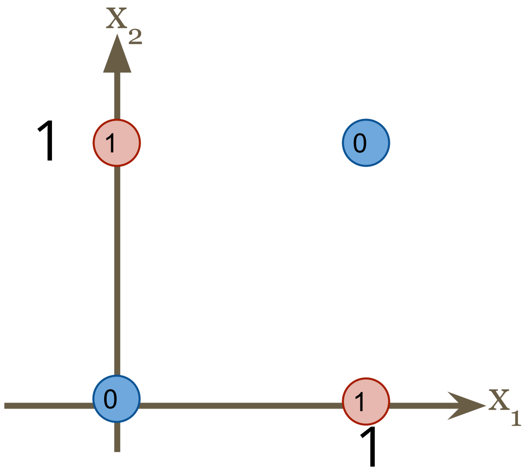
XOR	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	1
$x_2 = 1$	1	0

$$y = f(\, x_1 w_1 + x_2 w_2 + b)$$

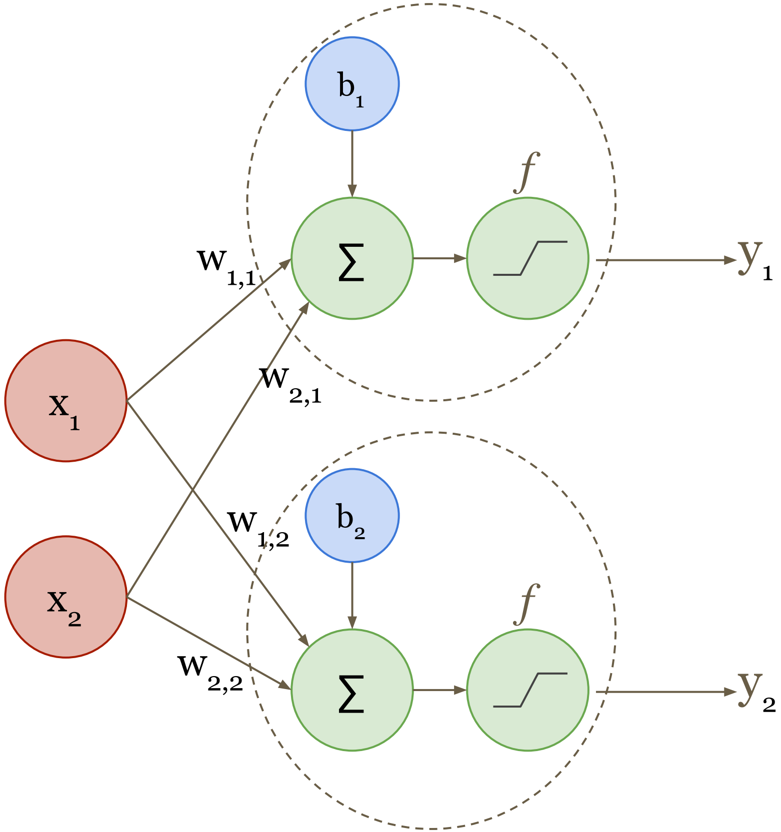
Let $\mathbf{x}^T = [x_1 \;\; x_2]$, $\mathbf{w}^T = [w_1 \;\; w_2]$

$$y = f(\, \mathbf{x}^T \mathbf{w} + b)$$





A more general case: MIMO - 2x2



Let:

$$\mathbf{x}^T = [x_1 \ \ x_2]$$

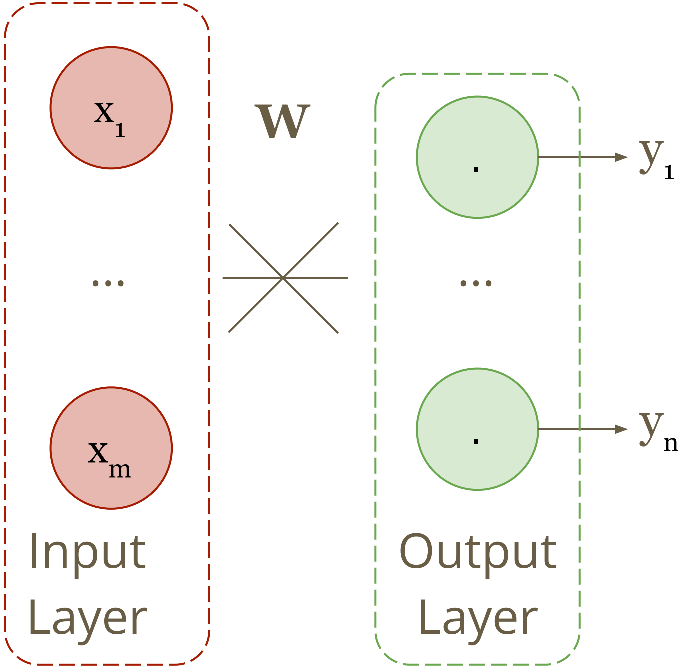
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$$

$$\mathbf{b} = [b_1 \ \ b_2]$$

$$\mathbf{y} = [y_1 \ \ y_2]$$

$$\mathbf{y} = f(\,\mathbf{x}^T\,\mathbf{W} + \mathbf{b})$$

A more general case: Input & Output Layers - MIMO



Let:

$$\mathbf{x}^T = [x_1 \dots x_m]$$

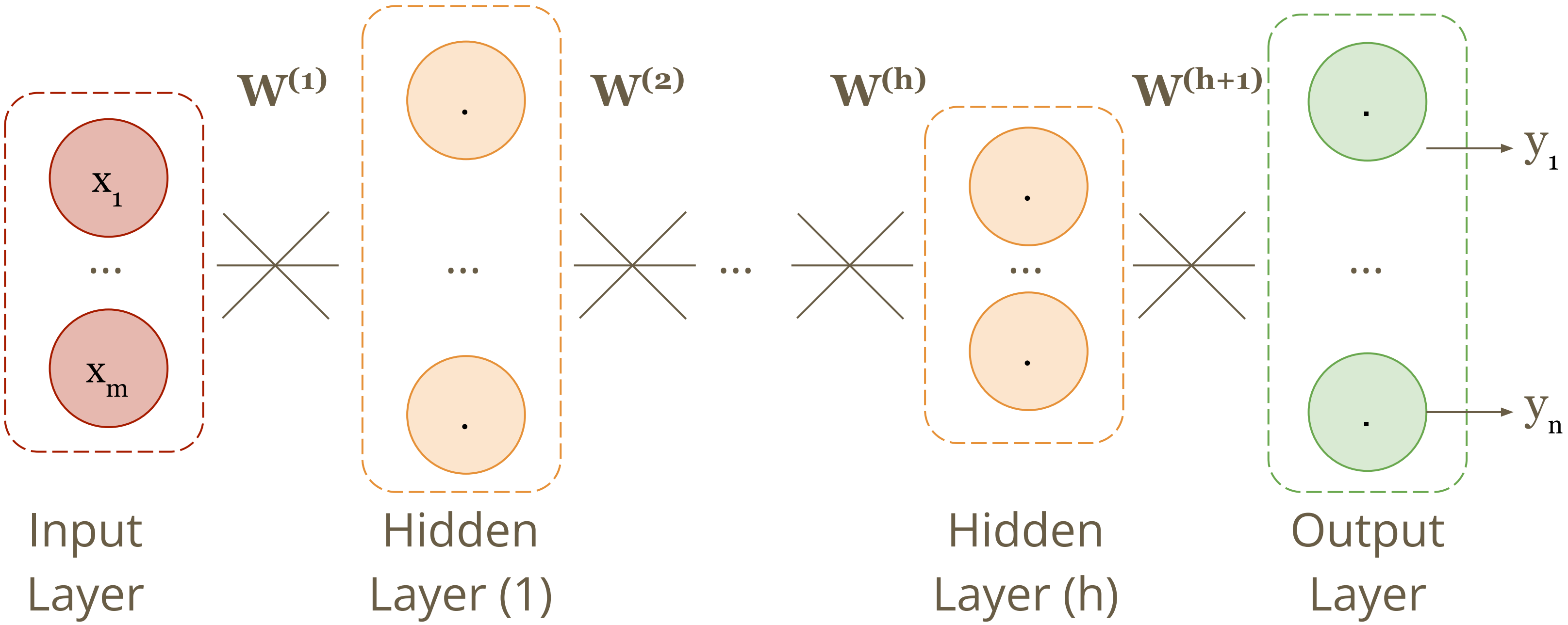
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & \dots & w_{1,n} \\ & \dots & \\ w_{m,1} & \dots & w_{m,n} \end{bmatrix}$$

$$\mathbf{b} = [b_1 \dots b_n]$$

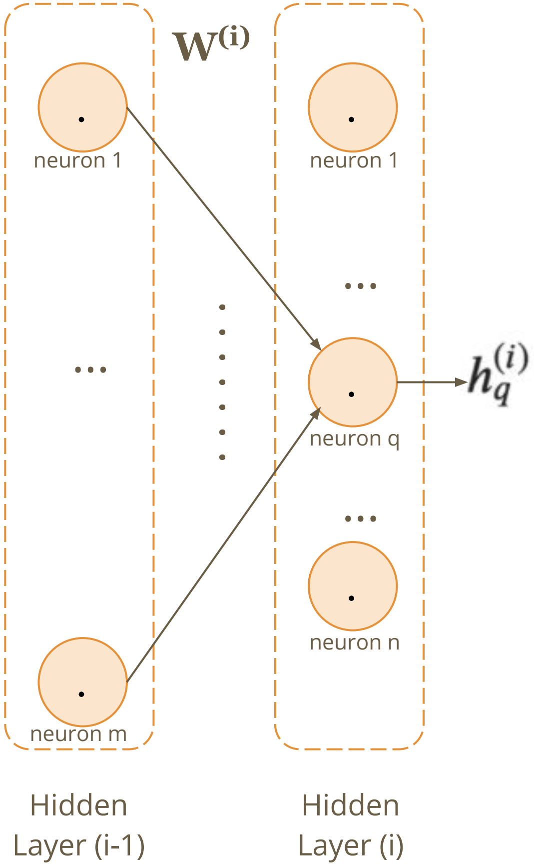
$$\mathbf{y} = [y_1 \dots y_n]$$

$$\mathbf{y} = f(\,\mathbf{x}^T\,\mathbf{W} + \mathbf{b})$$

Most general case: Shallow & Deep & Deeper



Most general case: Output of *any* neuron



Let hidden layers $(i-1)$, i have m, n neurons respectively.

$h_q^{(i)}$ is the output of the q^{th} neuron at the i^{th} hidden layer.

Weight $w_{p,q}^{(i)}$ is between the q^{th} neuron at hidden layer i and p^{th} neuron at the previous layer.

Bias for neuron q at hidden layer i is $b_q^{(i)}$.

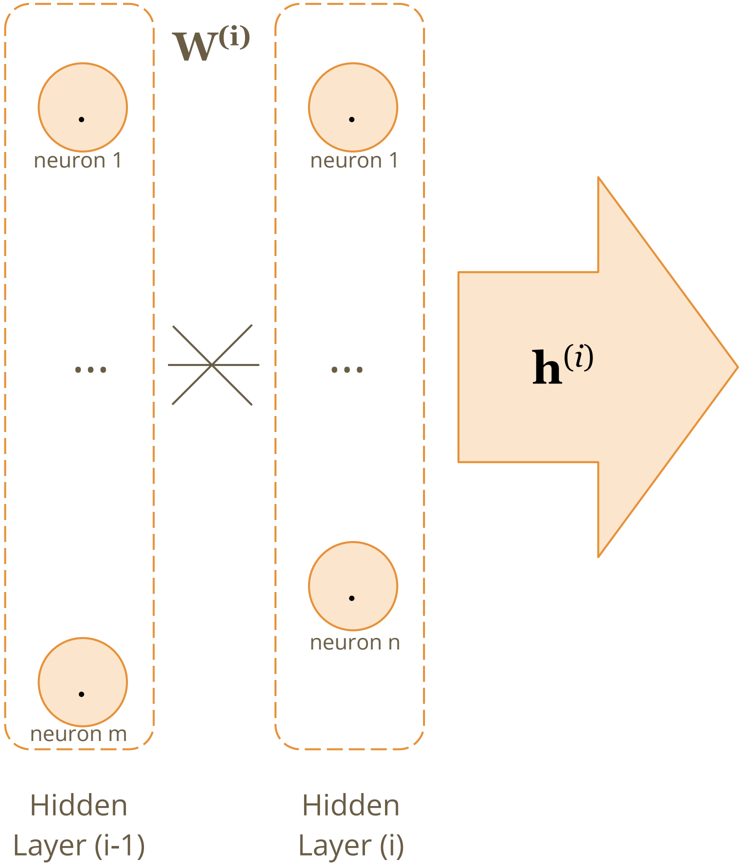
Then $h_q^{(i)}$ can be written as:

$$h_q^{(i)} = f\left(\sum_{j=1}^{m} \left[h_j^{(i-1)} w_{j,q}^{(i)} + b_q^{(i)} \right] \right)$$

where,

$f(\cdot)$ is the activation function.

Most general case: Output of *any* layer



Let $\mathbf{h}^{(i)}$ be the output of layer i, then:

$$\mathbf{h}^{(i)} = f\left(\mathbf{h}^{(i-1)^T}\mathbf{W}^{(i)} + \mathbf{b}^{(i)}\right)$$

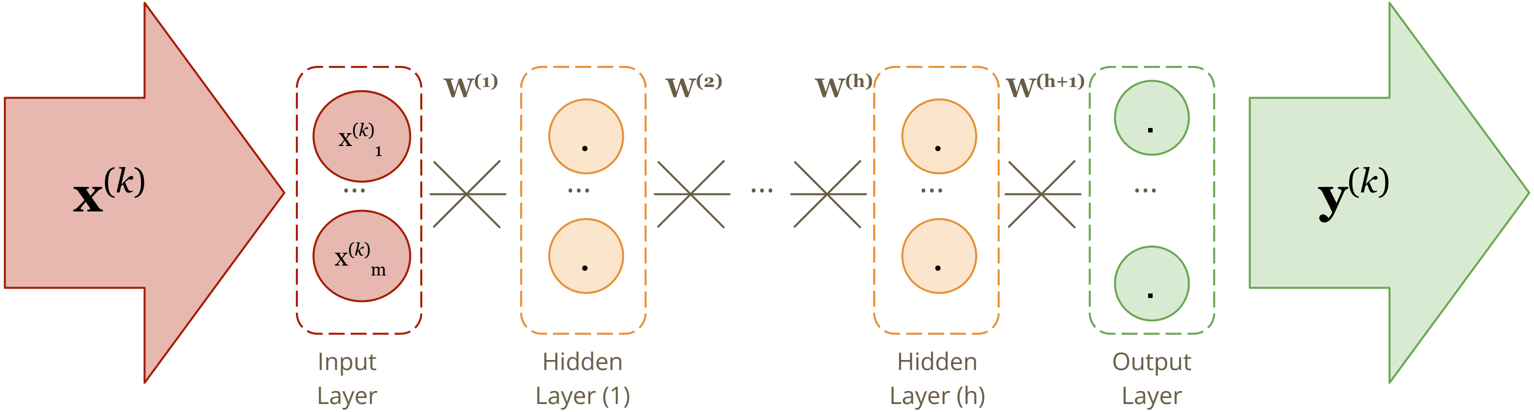
where,

$\mathbf{b}^{(i)} = \left[b_1^{(i)}, \ldots, b_n^{(i)} \right]$ and,

hidden

$\mathbf{W}_{mxn}^{(i)}$ is the weight matrix between layers $(i-1)$ and i.

Most general case: Output of *the network*



Note that, in general there will be several inputs, so let there be d many data points $\mathbf{x}^{(k)}$ as input and $\mathbf{y}^{(k)}$ is the corresponding network *prediction*/output, where $k = 1, \ldots, d$.

$$\mathbf{y}^{(k)} = f\Big(f(\ldots f(f(\mathbf{x}^{(k)^T}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})^T\mathbf{W}^{(2)} + \mathbf{b}^{(2)}\Big)^T \ldots\Big)^T\mathbf{W}^{(h+1)} + \mathbf{b}^{(h+1)}\Big)$$

In more general terms,

$$\mathbf{y}^{(k)} = F(\mathbf{x}^{(k)}, \mathbf{W}),$$

where $\mathbf{W} = \left\{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(h+1)}\right\}$ i.e. it represents the set of all $\mathbf{W}^{(i)}$s.

A sample case: How to train? When to train?

&	x_1 = 0	x_1 = 1
x_2 = 0	0	0
x_2 = 1	0	1

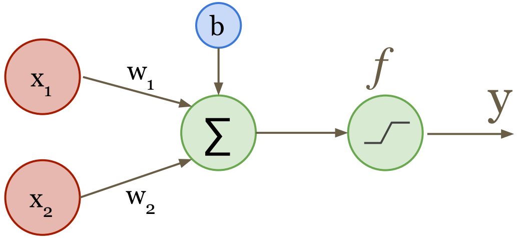
Initialize: $\mathbf{w}^T = [0.5 \;\; 0.1]$, $b = 0.1$

$y = f(\mathbf{x}^T \mathbf{w} + b) = f(0.5x_1 + 0.1x_2 + 0.1)$, **simple case** $\rightarrow f(x) = x$

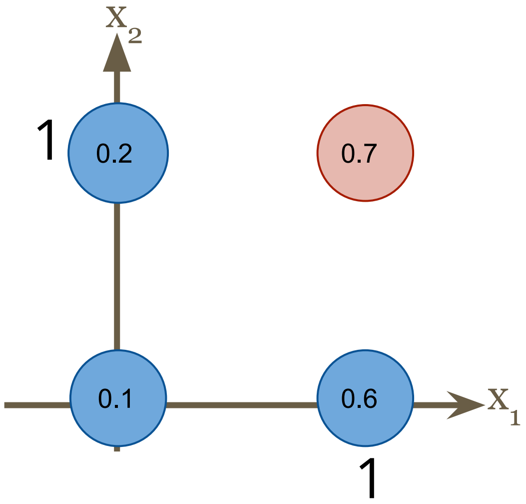
$y = 0.5x_1 + 0.1x_2 + 0.1$,

where y_t is the **true value**, i.e. ground truth

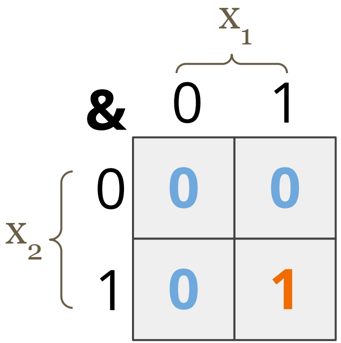
How and **when** to update w_i ?

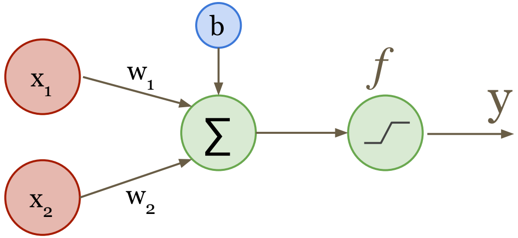


x_1	x_2	y_t	y	error
0	0	0	0.1	0.1
0	1	0	0.2	0.2
1	0	0	0.6	0.6
1	1	1	0.7	0.3



A sample case: What to minimize at the end? J ?





Initialize: $\mathbf{w}^T = [0.5 \;\; 0.1]$, $b = 0.1$

$y = f(\mathbf{x}^T \mathbf{w} + b) = f(0.5x_1 + 0.1\,x_2 + 0.1)$, simple case $f(x) = x$

$y = 0.5\,x_1 + 0.1\,x_2 + 0.1$

x_1	x_2	y_t	y	err
0	0	0	0.1	0.1
0	1	0	0.2	0.2
1	0	0	0.6	0.6
1	1	1	0.7	0.3

Loss function: $L(y_t, y)$

$L_i(y_t, y)$ Loss for *input i*

Total loss in this case:

$\boldsymbol{J} = L_1 + \ldots + L_4$

Where $\boldsymbol{J}$ is the *cost function*

where $\mathbf{y}_t$, $\mathbf{y}$ are the vectors representing the **ground truth** and the **network output**, *i.e. network prediction* respectively.

Tensor Flow: Loss functions...

`class BinaryCrossentropy` : Computes the cross-entropy loss between true labels and predicted labels.

`class CategoricalCrossentropy` : Computes the crossentropy loss between the labels and predictions.

`class CategoricalHinge` : Computes the categorical hinge loss between `y_true` and `y_pred` .

`class CosineSimilarity` : Computes the cosine similarity between labels and predictions.

`class Hinge` : Computes the hinge loss between `y_true` and `y_pred` .

`class Huber` : Computes the Huber loss between `y_true` and `y_pred` .

`class KLDivergence` : Computes Kullback-Leibler divergence loss between `y_true` and `y_pred` .

`class LogCosh` : Computes the logarithm of the hyperbolic cosine of the prediction error.

`class Loss` : Loss base class.

`class MeanAbsoluteError` : Computes the mean of absolute difference between labels and predictions.

`class MeanAbsolutePercentageError` : Computes the mean absolute percentage error between `y_true` and `y_pred` .

`class MeanSquaredError` : Computes the mean of squares of errors between labels and predictions.

`class MeanSquaredLogarithmicError` : Computes the mean squared logarithmic error between `y_true` and `y_pred` .

`class Poisson` : Computes the Poisson loss between `y_true` and `y_pred` .

`class Reduction` : Types of loss reduction.

`class SparseCategoricalCrossentropy` : Computes the crossentropy loss between the labels and predictions.

`class SquaredHinge` : Computes the squared hinge loss between `y_true` and `y_pred` .

Check out: https://www.tensorflow.org/api_docs/python/tf/keras/losses

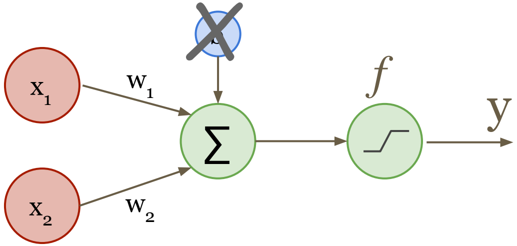
pyTorch Flow: Loss functions…

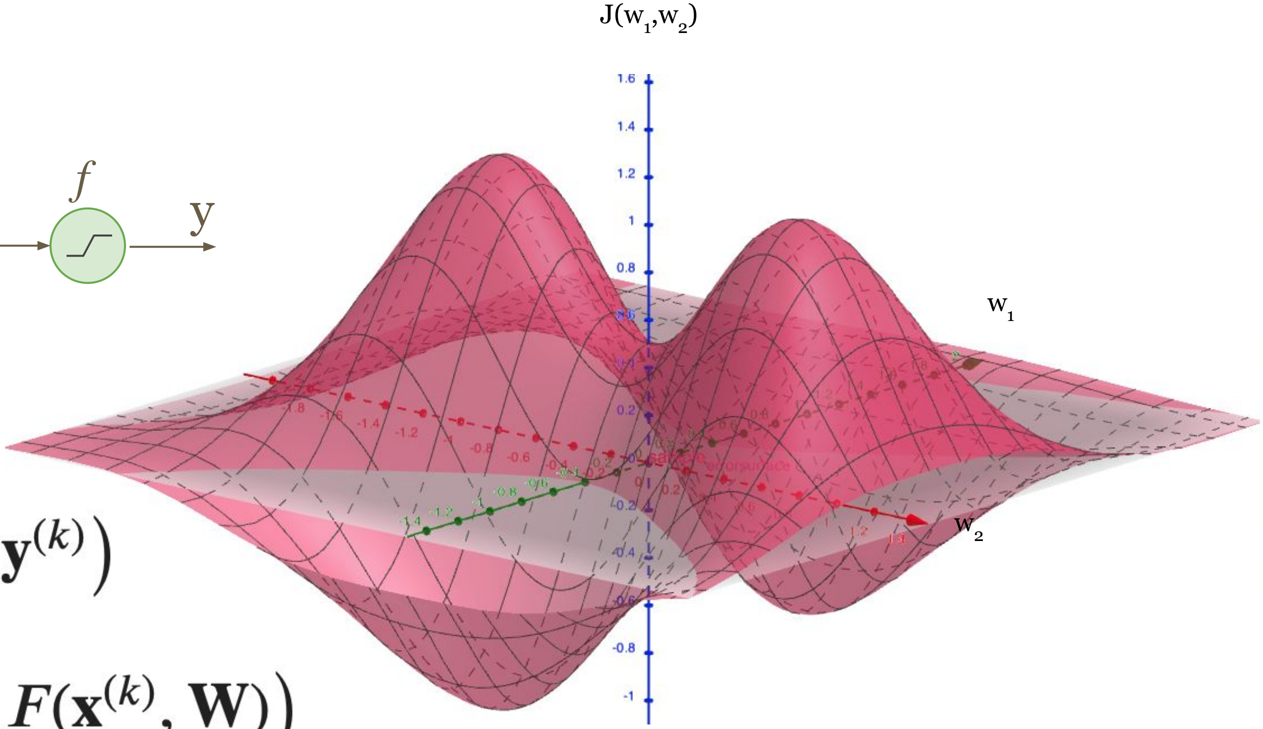
nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
nn.CrossEntropyLoss	This criterion computes the cross entropy loss between input and target.
nn.CTCLoss	The Connectionist Temporal Classification loss.
nn.NLLLoss	The negative log likelihood loss.
nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
nn.KLDivLoss	The Kullback-Leibler divergence loss measure
nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
nn.BCEWithLogitsLoss	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
nn.MarginRankingLoss	Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch *Tensors*, and a label 1D mini-batch tensor y (containing 1 or -1).

nn.HingeEmbeddingLoss	Measures the loss given an input tensor x and a labels tensor y (containing 1 or -1).
nn.MultiLabelMarginLoss	Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 2D *Tensor* of target class indices).
nn.HuberLoss	Creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise.
nn.SmoothL1Loss	Creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise.
nn.SoftMarginLoss	Creates a criterion that optimizes a two-class classification logistic loss between input tensor x and target tensor y (containing 1 or -1).
nn.MultiLabelSoftMarginLoss	Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input x and target y of size (N, C).
nn.CosineEmbeddingLoss	Creates a criterion that measures the loss given input tensors x_1, x_2 and a *Tensor* label y with values 1 or -1.
nn.MultiMarginLoss	Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 1D tensor of target class indices, $0 \leq y \leq \text{x.size}(1) - 1$):
nn.TripletMarginLoss	Creates a criterion that measures the triplet loss given an input tensors $x1, x2, x3$ and a margin with a value greater than 0.
nn.TripletMarginWithDistanceLoss	Creates a criterion that measures the triplet loss given input tensors a, p, and n (representing anchor, positive, and negative examples, respectively), and a nonnegative, real-valued function ("distance function") used to compute the relationship between the anchor and positive example ("positive distance") and the anchor and negative example ("negative distance").

Check out: https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Surface & Cost Function: A *simple* hypothetical case

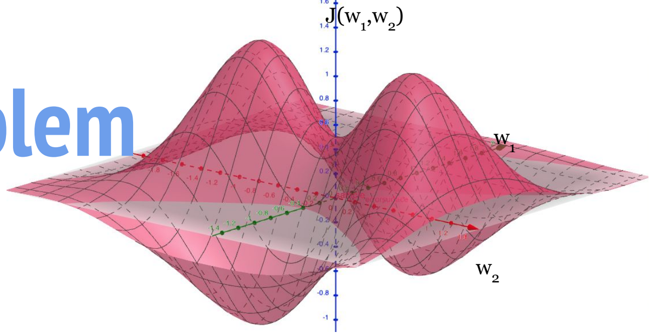


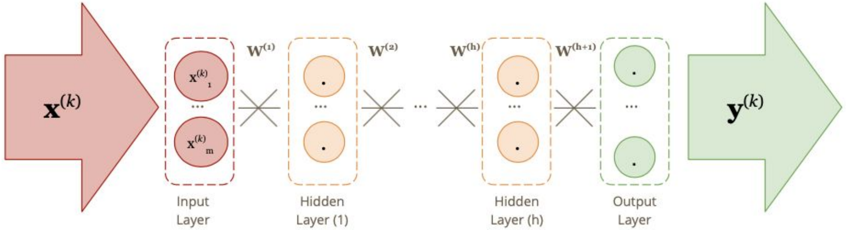


$$J(\mathbf{W}) = \frac{1}{d}\sum_{k=1}^{d} L\left(\mathbf{y}_t^{(k)}, \mathbf{y}^{(k)}\right)$$

$$= \frac{1}{d}\sum_{k=1}^{d} L\left(\mathbf{y}_t^{(k)}, F(\mathbf{x}^{(k)}, \mathbf{W})\right)$$

Loss Surface Minima: A search problem





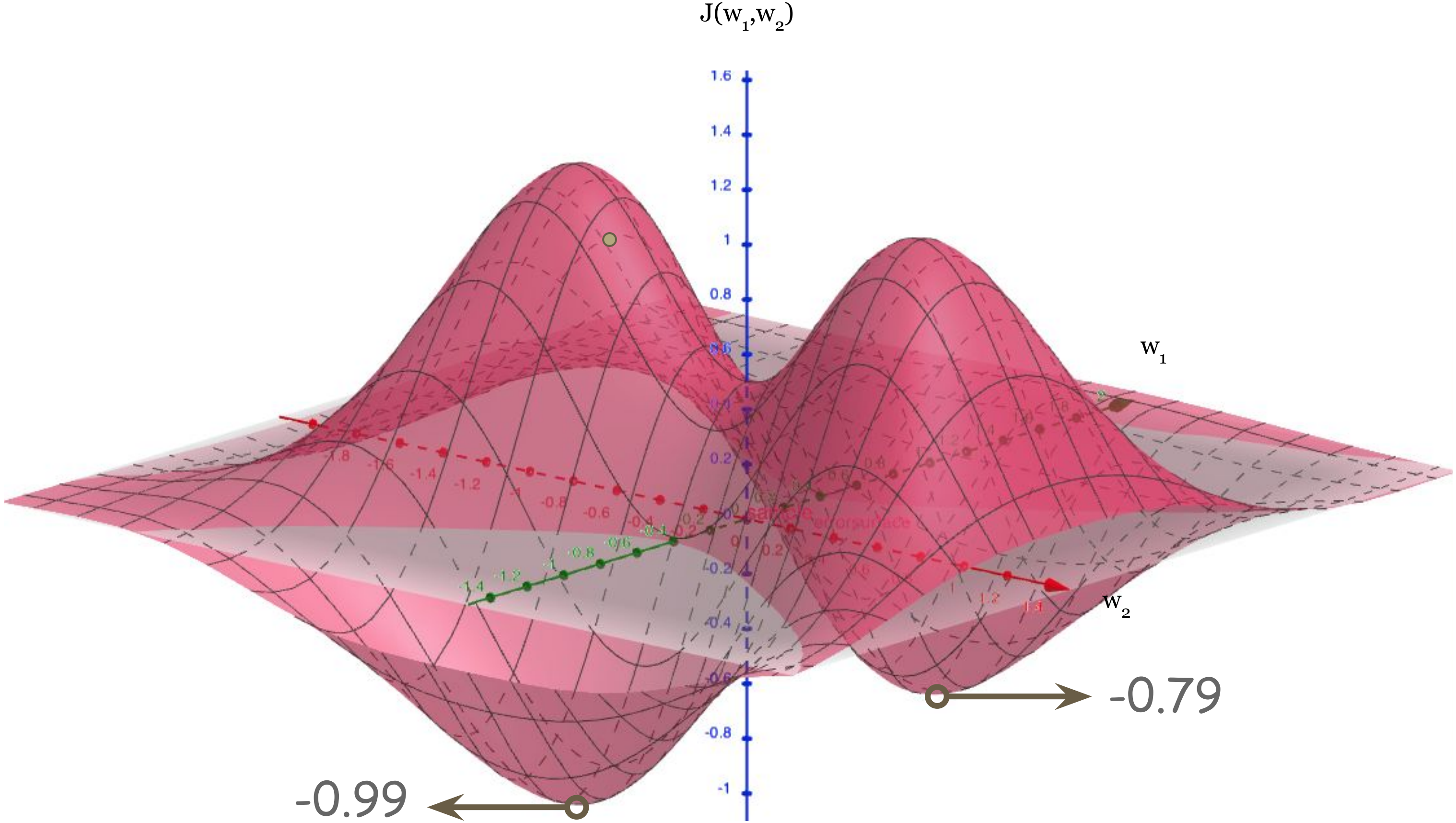
For d data points, cost function can be written as:

$$J(\mathbf{W}) = \frac{1}{d}\sum_{k=1}^{d} L\left(\mathbf{y}_t^{(k)}, \mathbf{y}^{(k)}\right) = \frac{1}{d}\sum_{k=1}^{d} L\left(\mathbf{y}_t^{(k)}, F(\mathbf{x}^{(k)}, \mathbf{W})\right)$$

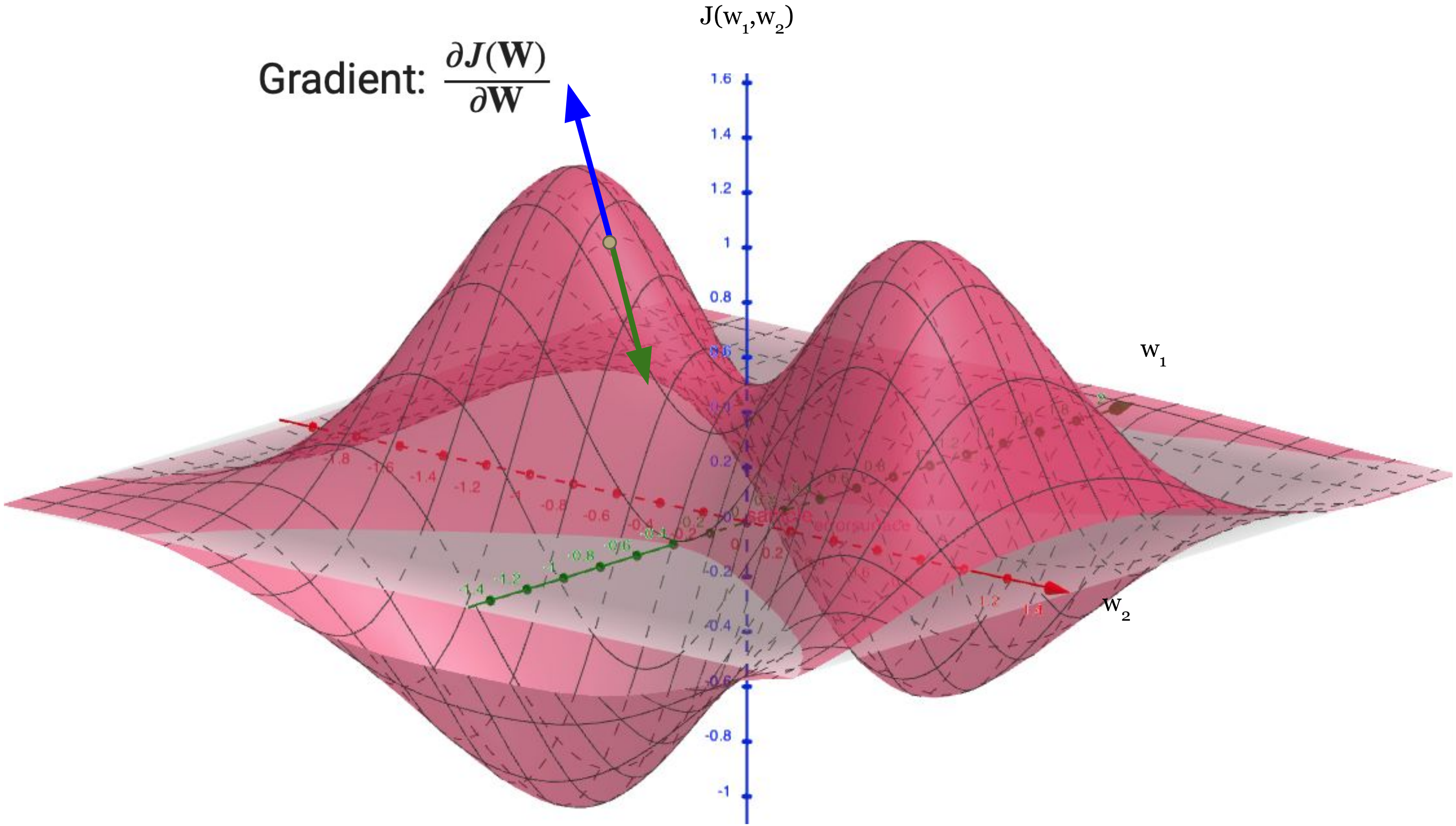
Best set of weight matrices given the selected cost function:

$$W^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

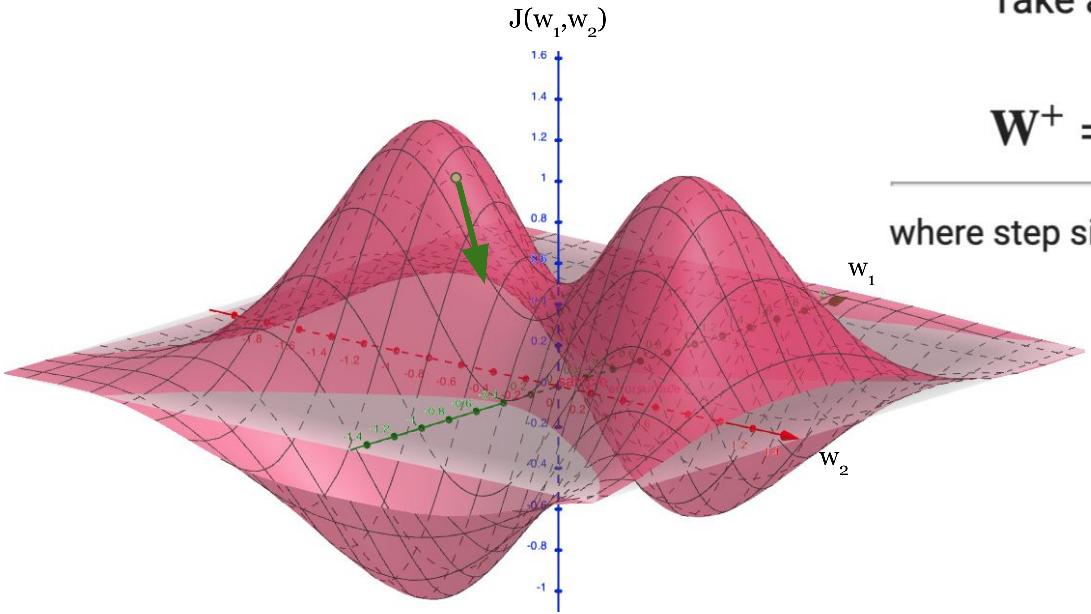
Loss Surface Gradient: Slide to Minima but which?



Loss Surface Gradient: Steepest Descent to Minima



Loss Surface Gradient: Steep*est* Descent to Minima



Gradient Descent Algorithm:

- Initialize network: $\mathbf{W}$, random is a good choice
- Loop until not worth it:

  Take a step in $-$gradient direction to update $\mathbf{W}$ as:

$$\mathbf{W}^{+} = \boxed{\mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}}$$

---

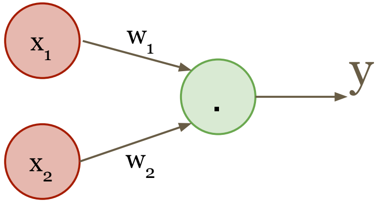
where step size η is referred to as the learning rate.

Alternative methods result in **different optimizers**

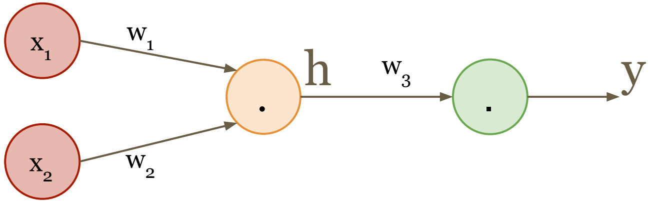
How to update w_i: Backpropagation

$$J(\mathbf{W}) = \frac{1}{d}\sum_{k=1}^{d} L\left(\mathbf{y}_t^{(k)}, \mathbf{y}^{(k)}\right)$$
$$= \frac{1}{d}\sum_{k=1}^{d} L\left(\mathbf{y}_t^{(k)}, F(\mathbf{x}^{(k)}, \mathbf{W})\right)$$

What is the effect of w_1 on $\mathbf{J}$:



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial y} \cdot \frac{\partial y}{\partial w_1}$$



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial y} \cdot \frac{\partial y}{\partial h} \cdot \frac{\partial h}{\partial w_1}$$

Tensor Flow: Optimizers… Gradient Descent

`class Adadelta` : Optimizer that implements the Adadelta algorithm.

`class Adagrad` : Optimizer that implements the Adagrad algorithm.

`class Adam` : Optimizer that implements the Adam algorithm.

`class Adamax` : Optimizer that implements the Adamax algorithm.

`class Ftrl` : Optimizer that implements the FTRL algorithm.

`class Nadam` : Optimizer that implements the NAdam algorithm.

`class Optimizer` : Base class for Keras optimizers.

`class RMSprop` : Optimizer that implements the RMSprop algorithm.

`class SGD` : Gradient descent (with momentum) optimizer.

Check out: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

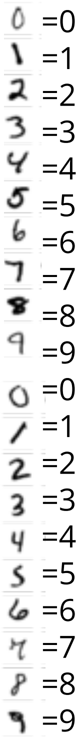
pyTorch Flow: Optimizers... Gradient Descent

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	Implements lazy version of Adam algorithm suitable for sparse tensors.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.

`LBFGS`	Implements L-BFGS algorithm, heavily inspired by minFunc.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.
`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

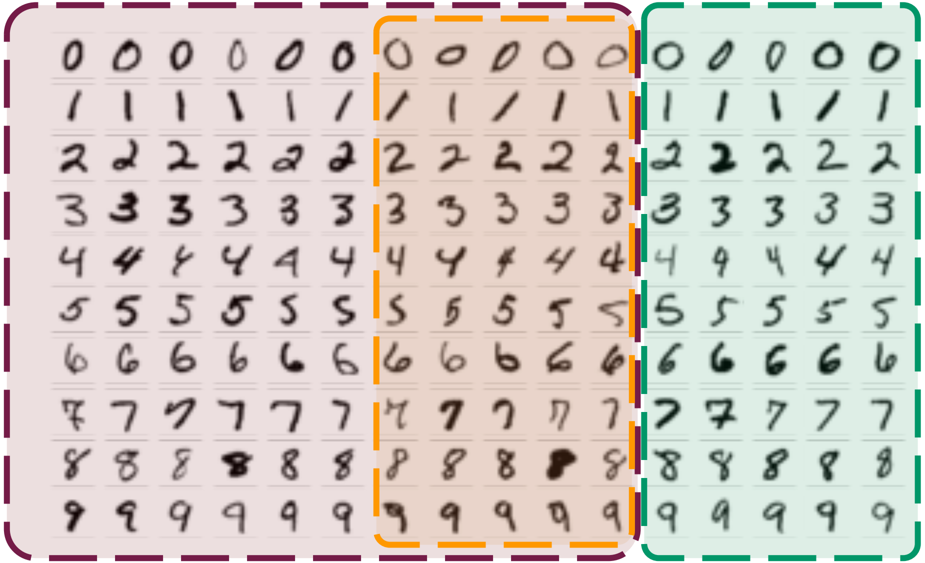
Check out: https://pytorch.org/docs/stable/optim.html

Big Picture: Getting started - Labelled Data



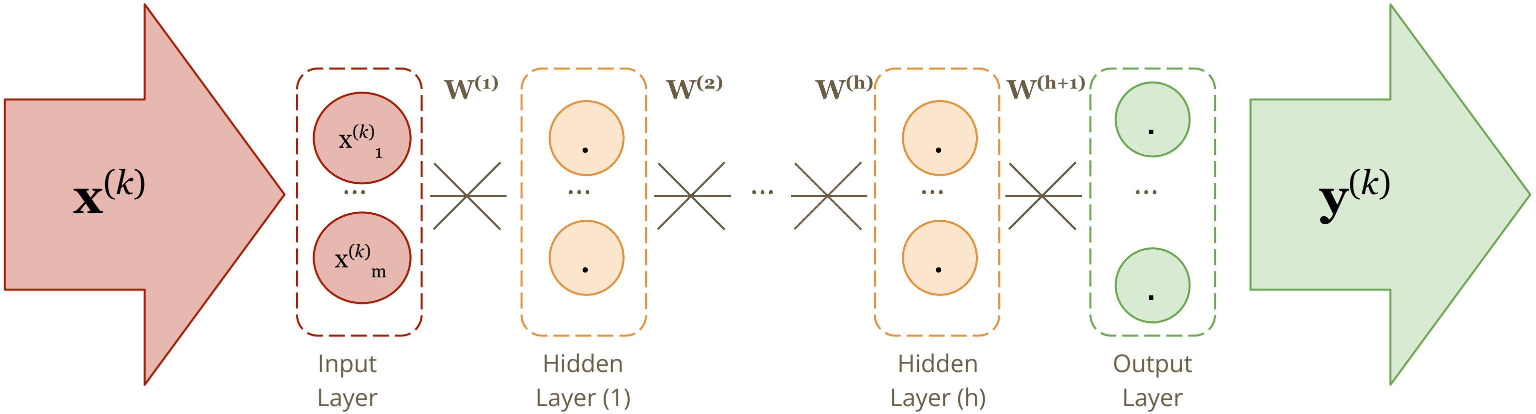
Labelled data: If not available you get the honor to label 70K of them! Enjoy...

Big Picture: Getting started - Divide Data



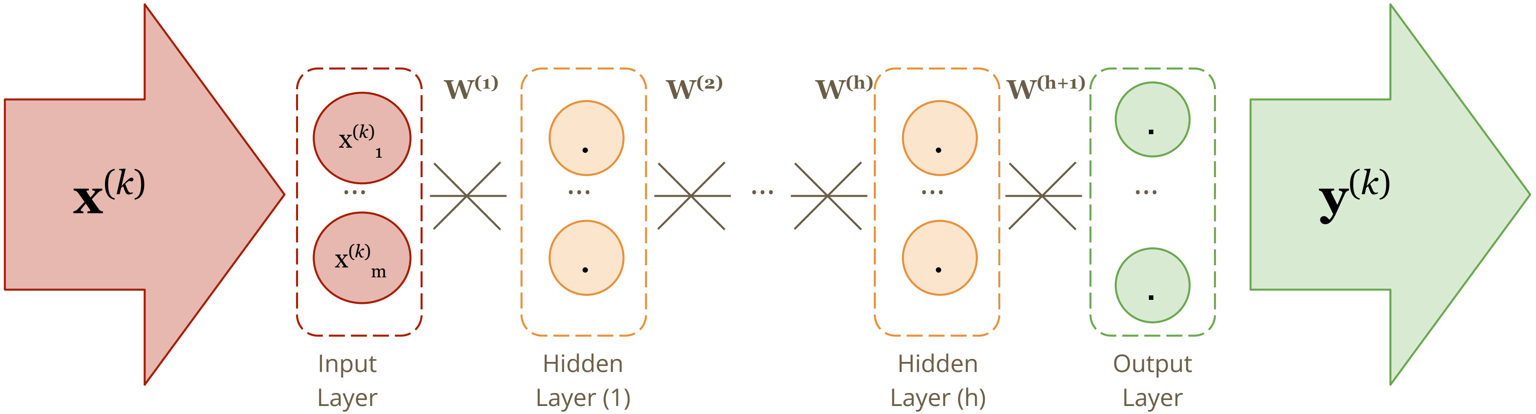
Data → **Training** data, **Validation** data, **Test** data

Big Picture: Getting started - Train where?



Hyperparameters: network topology, number of layers, number of neurons etc, *regularization* (dropout, early stopping, data augmentation, etc), optimizer, activation function, ...

Big Picture: Regularization ...



Fight against: complex solutions lead to **overfitting** / overlearning / memorizing data

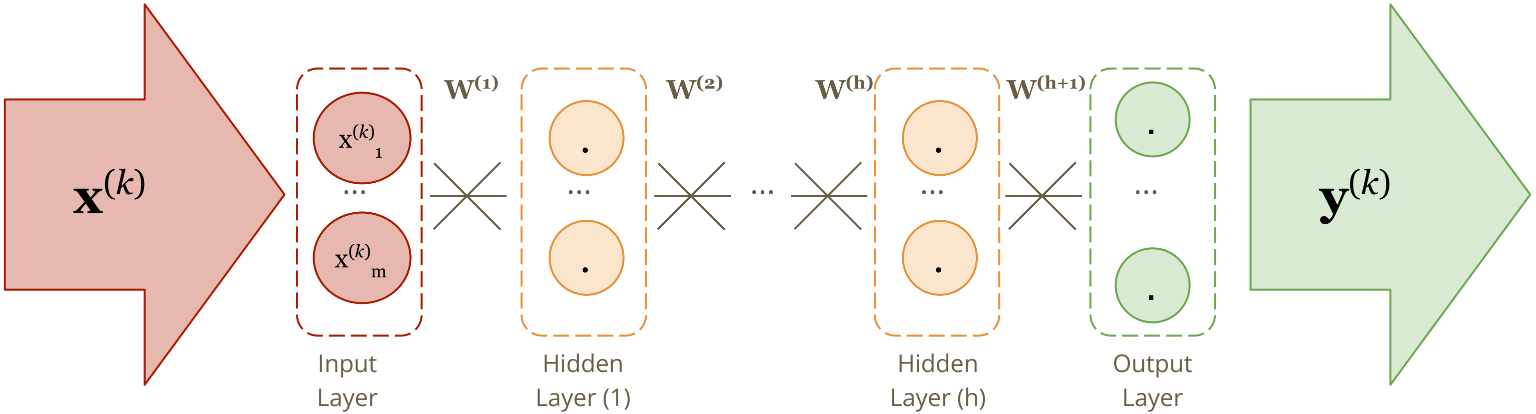
Dropout: Probabilistically pop some of the neurons in each iteration

Data Augmentation: shift, scale, rotate, add noise, etc. to generate variations

Regularization term: add a term to the cost function

Early Stopping: Stop when validation error stops improving

Big Picture: When to update?

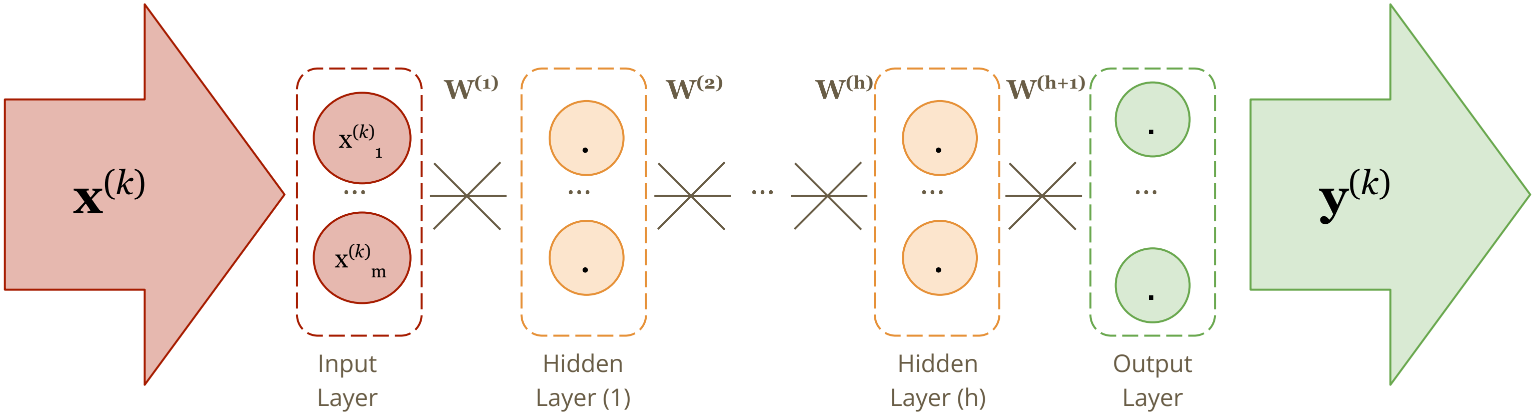


Epoch / Batch: Pass all the data through the network, calculate loss, update weights.

After every data point: Randomly select one - SGD - check this video out

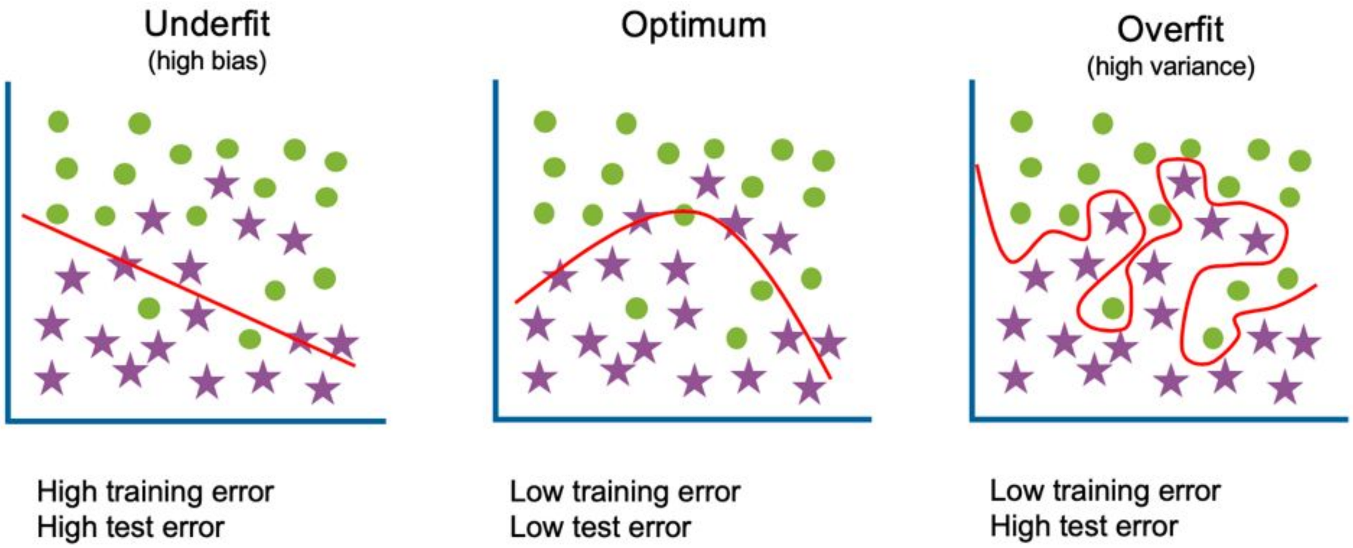
Mini-Batch: Data passed in subsets and weights updated after each batch

Big Picture: Loop until not worth it?



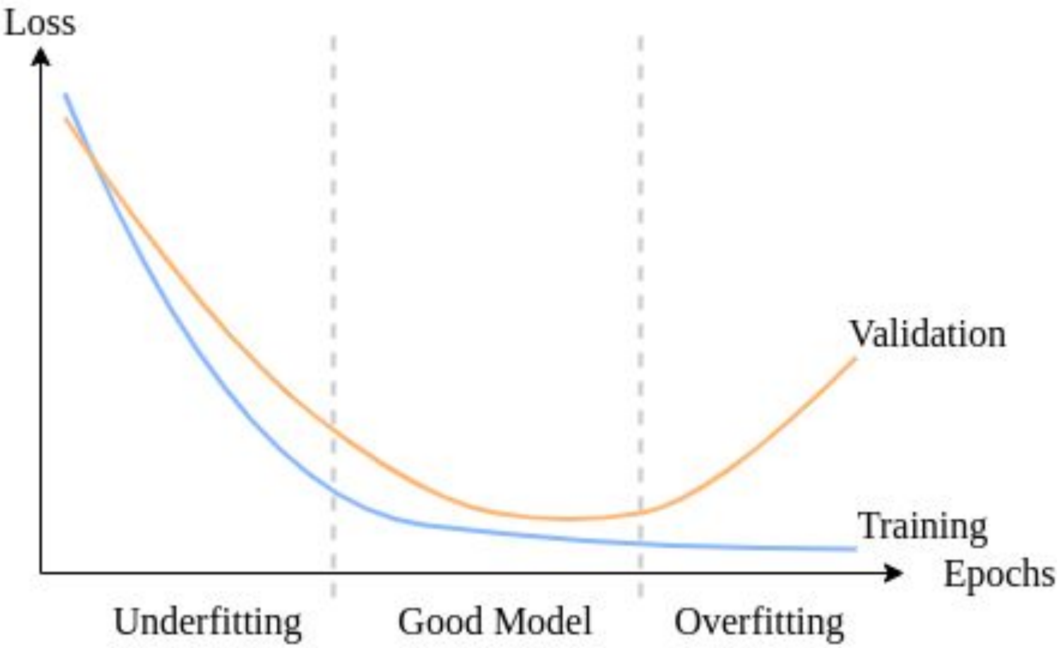
- Error is low enough - *i.e. error is below a preset threshold*
- Got bored - *i.e. maximum epoch limit is reached*
- Validation error started to increase while training decreases - *how is this even possible?*
- ???

Big Picture: Try not to over- or under-fit



https://www.javatpoint.com/overfitting-in-machine-learning

Big Picture: How to avoid overfit?



https://www.baeldung.com/cs/ml-underfitting-overfitting

A simple case: Try backpropagation manually

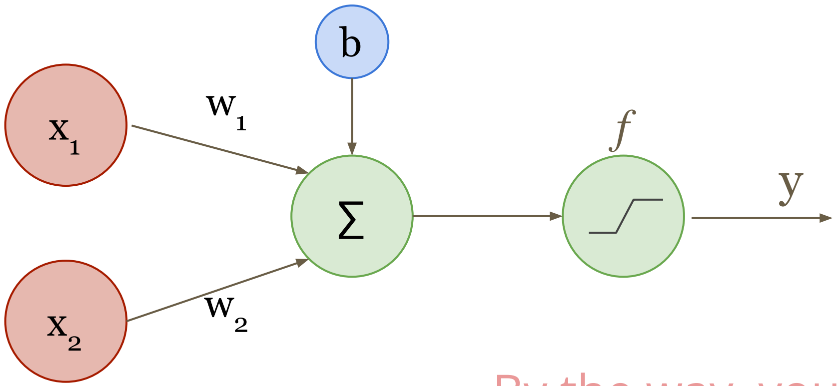
&	x_1 = 0	x_1 = 1
x_2 = 0	0	0
x_2 = 1	0	1

Initialize **w**, b as you like

choose $f(\,.\,)$, loss function and learning rate

Given $y = f(\mathbf{x}^T \mathbf{w} + b)$

Run gradient descent



By the way, you can implement this in numpy

Good news

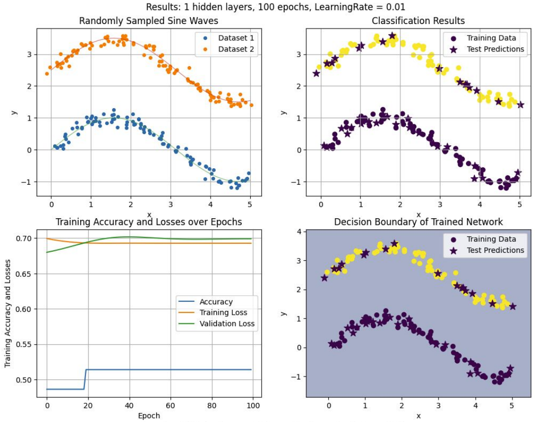
13 lines of code → A Neural Network: A good read, a good practice

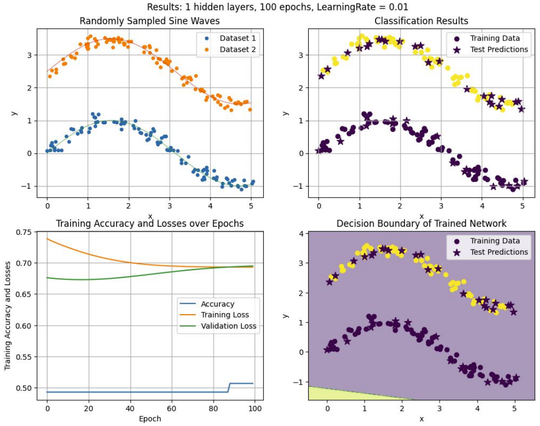
You won't need to code a ANN from scratch:

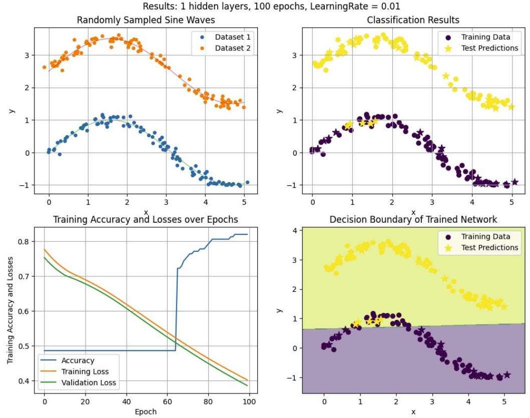
- TF, pyTorch, etc. exit
- LLMs assist

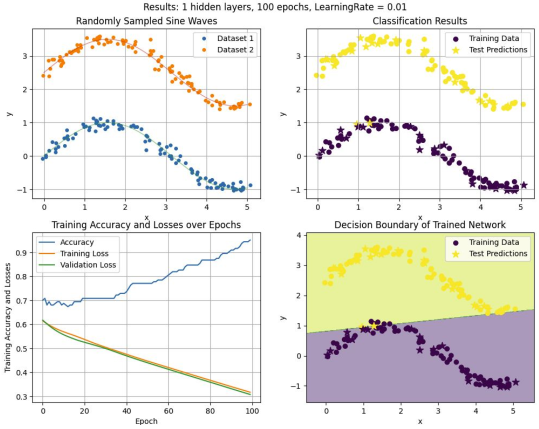
Check out: Tensorflow playground

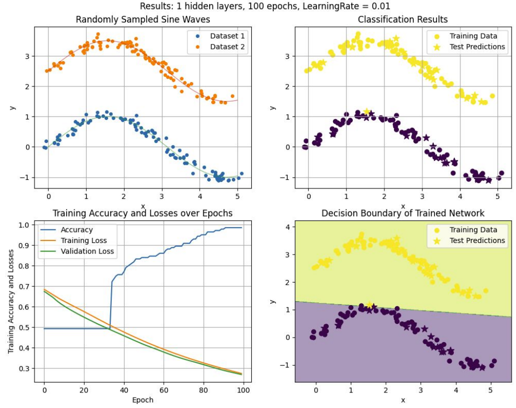
Same conditions, Just re-runs from scratch

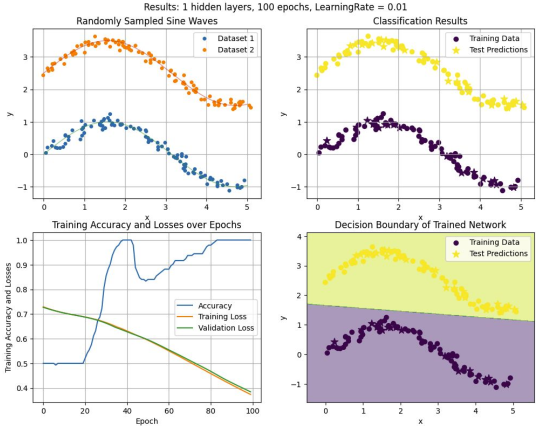






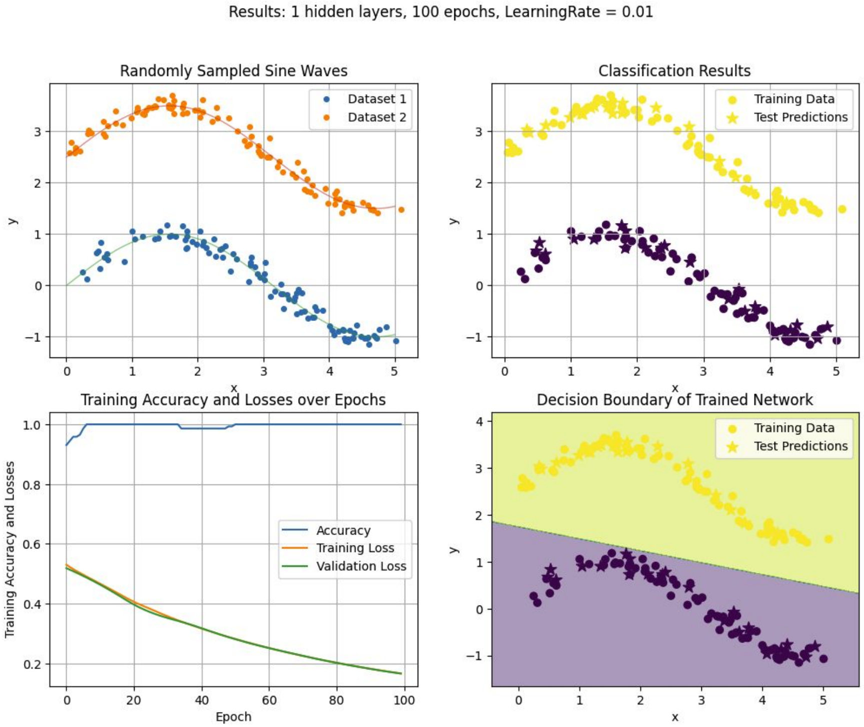






Let’s try together

Check this colab notebook



to be continued...