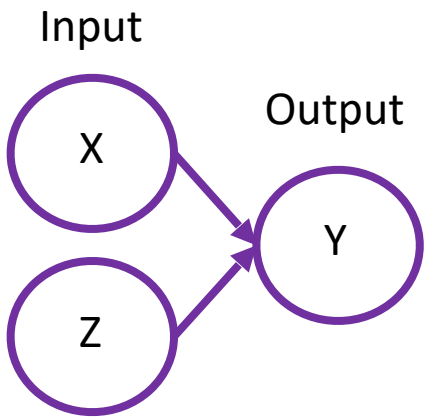
Deep Learning

- We trained a simple linear regression model that had two inputs and one output



That's a Neural Network with no depth

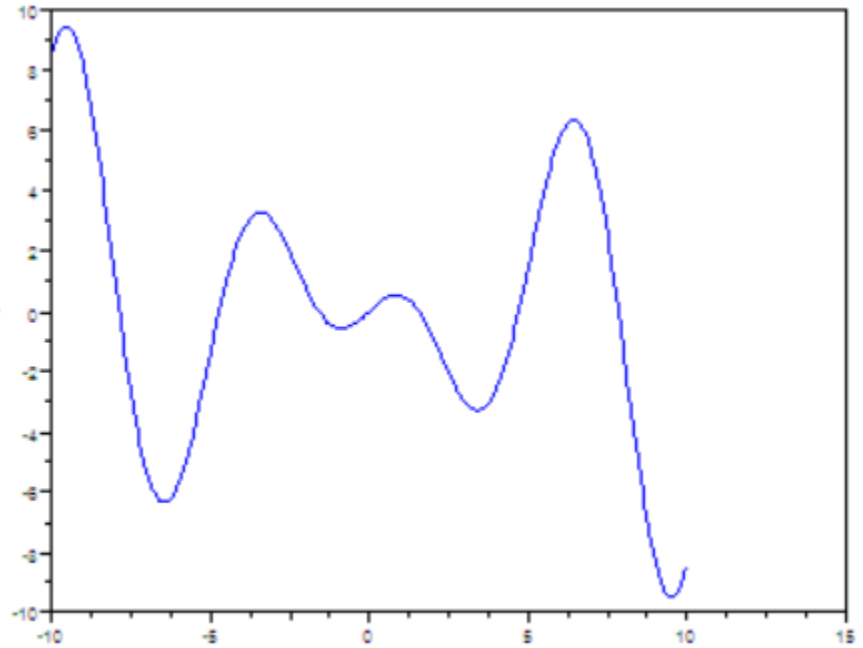
- We used a **Linear Model** to learn the function

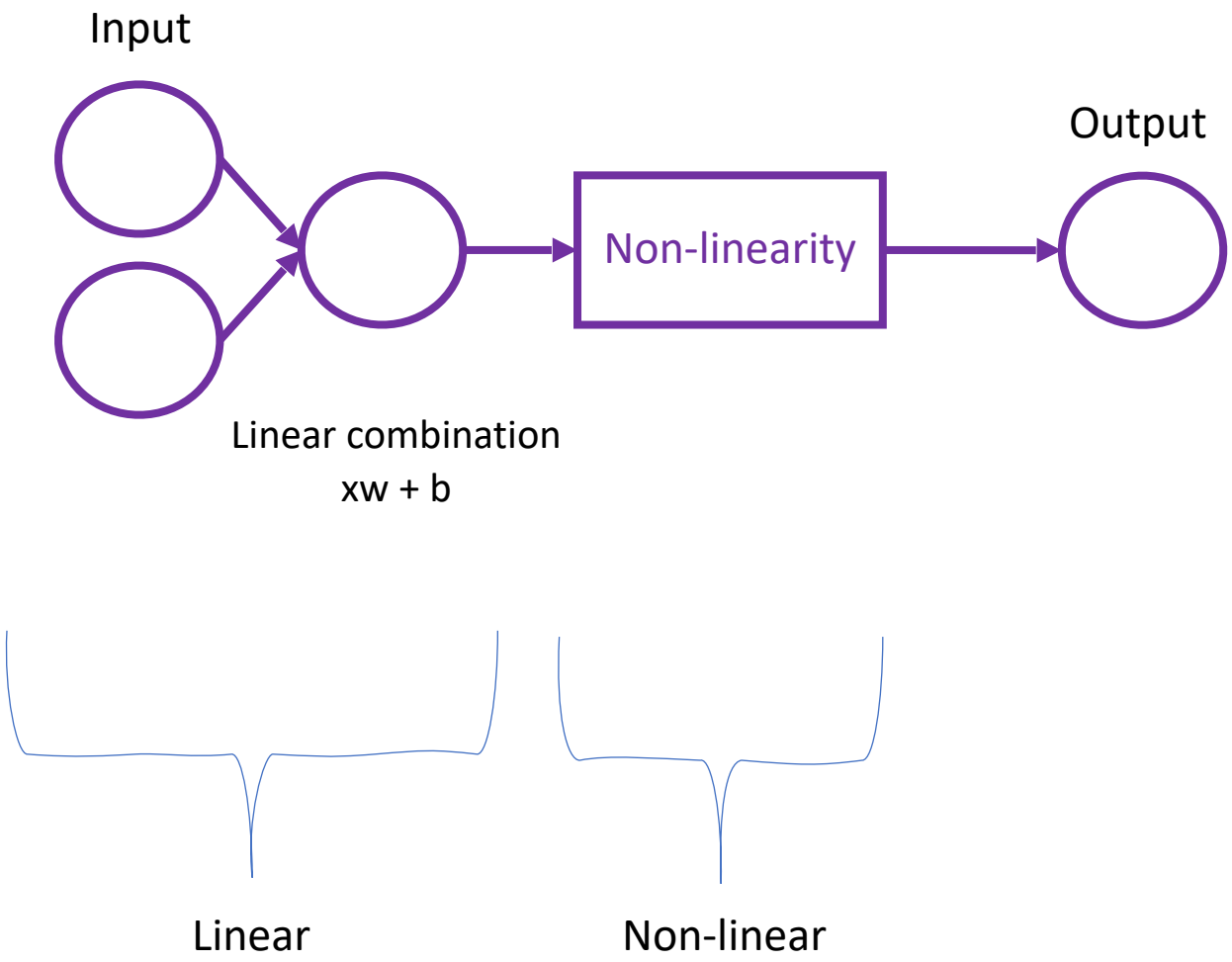
  However, most real-life dependencies cannot be modeled with a simple linear combination. We need better models!

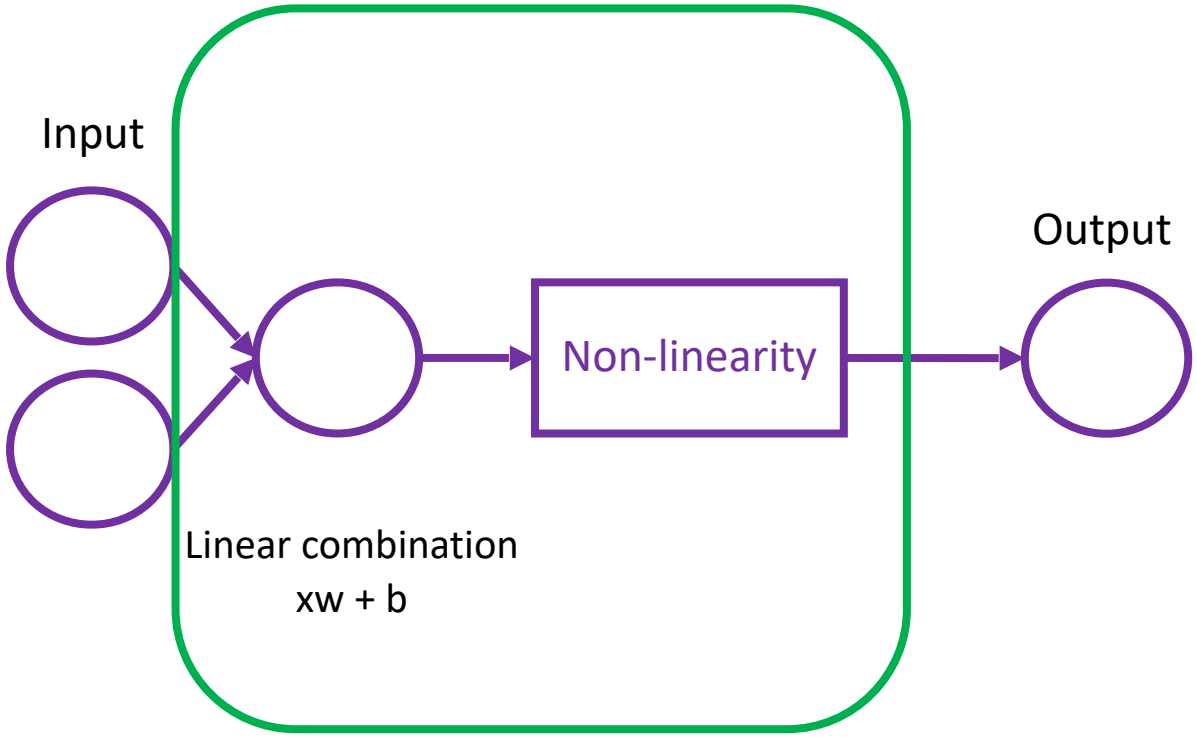
- We can raise the bar using both linear and non-linear operations

  Mixing linear combinations and non-linearities allow us to model arbitrary functions

  Or in other words functions with strange unconventional shapes like this one:



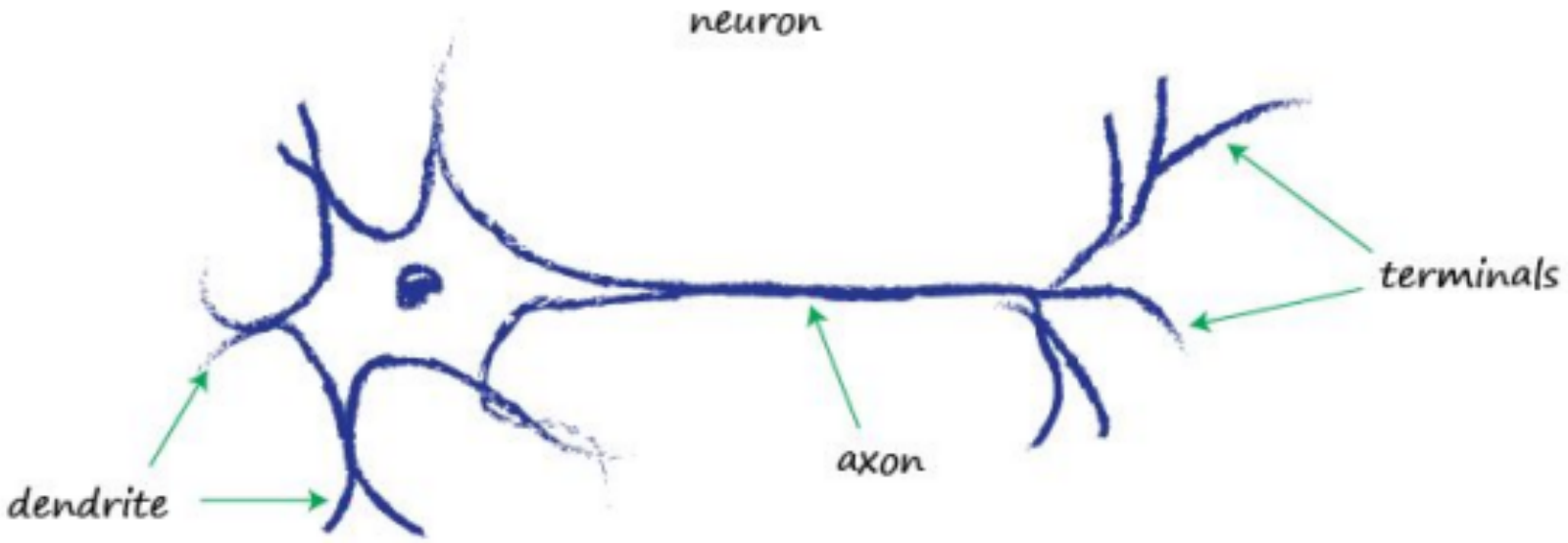

Input
Output
Non-linearity
Linear combination
xw + b
Linear
Non-linear



- The initial linear combination and the added non-linearity form a **layer**

  A **layer** is the building block of neural networks

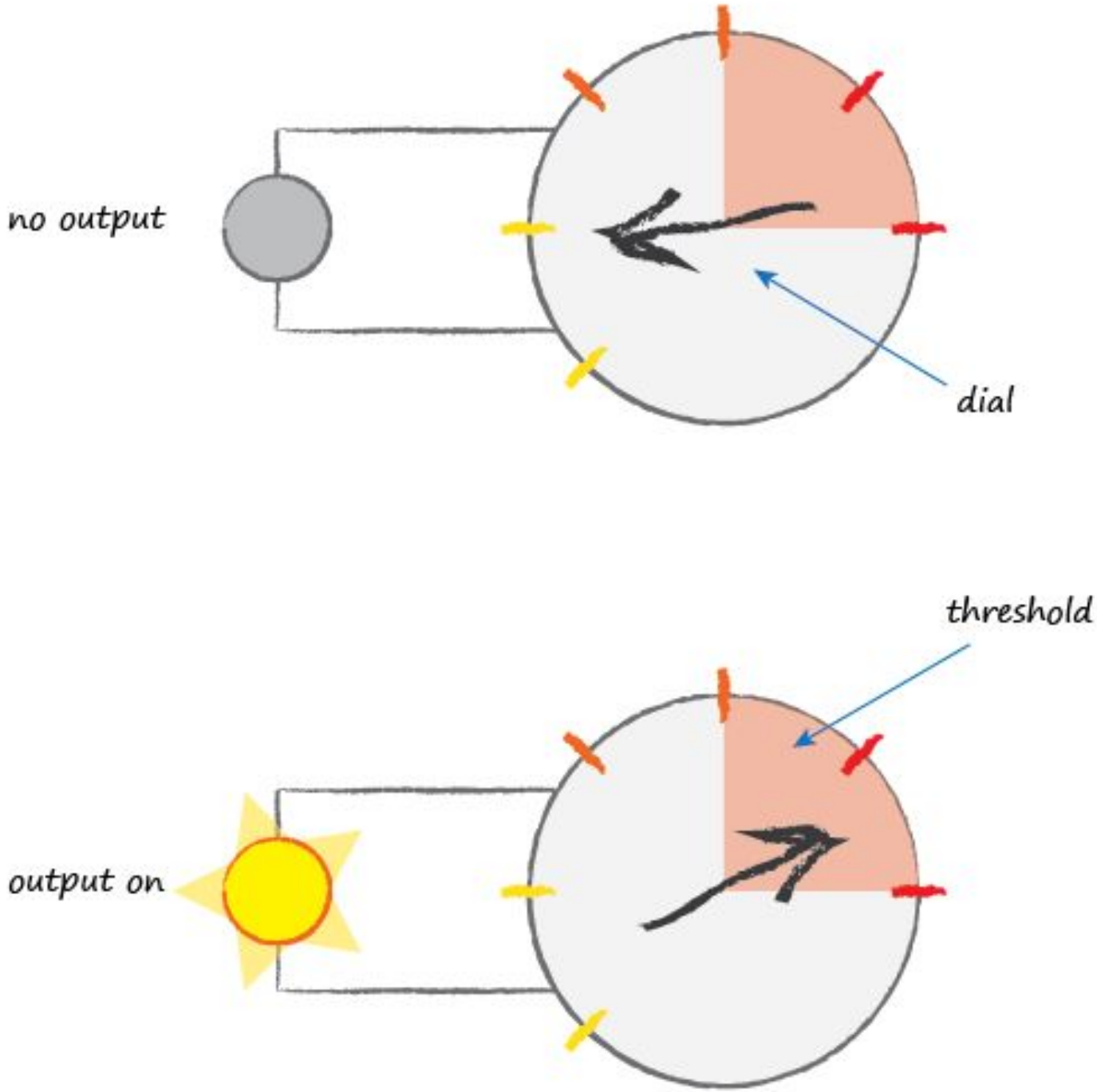
- When we have more than one layer, we are talking about a **deep neural network**



➢ A biological neuron doesn't produce an output that is simply a simple linear function of the input

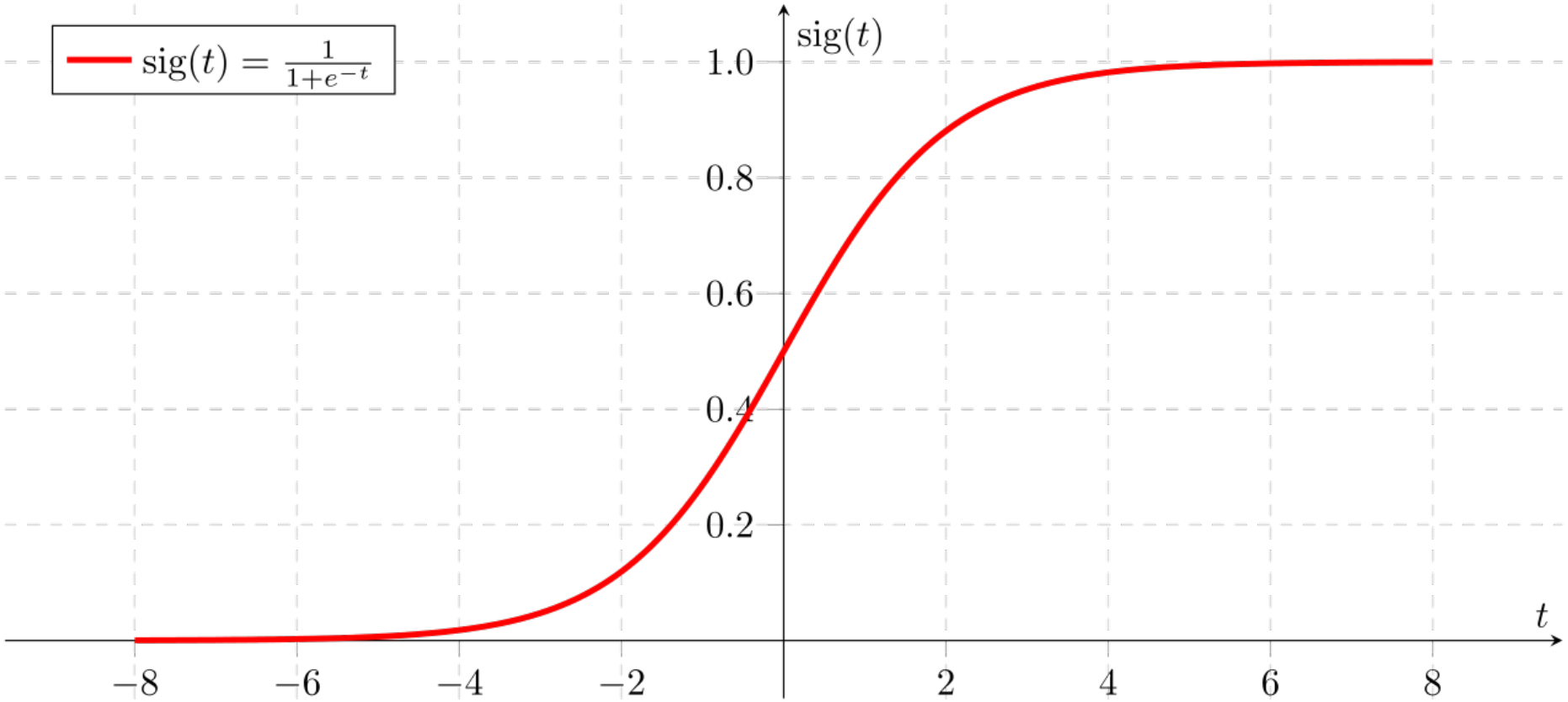
- Observations suggest that <u>neurons don't react readily</u>, but instead **suppress the input until it has grown so large that it triggers an output**

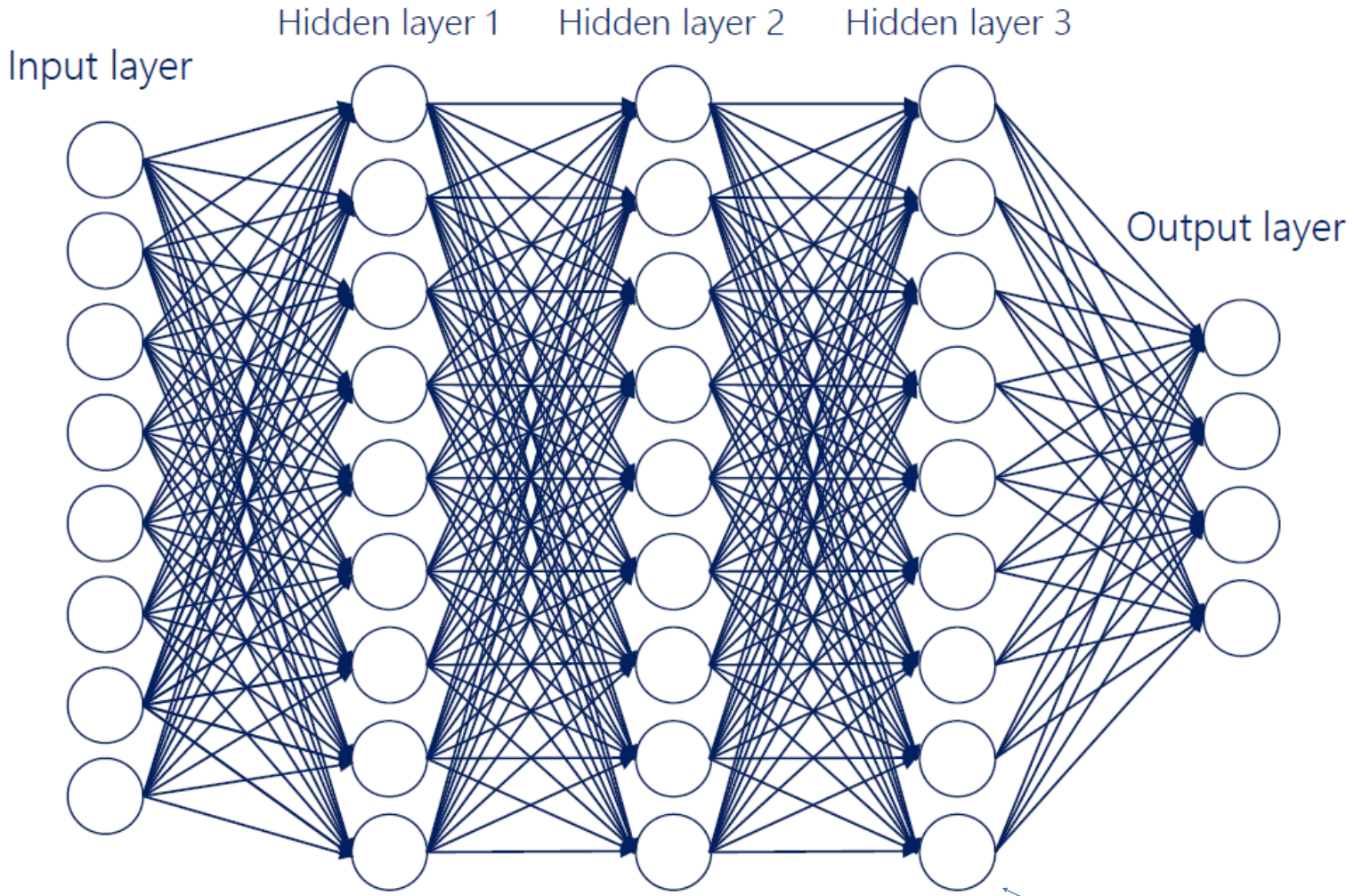
  You can think of this as a **threshold** that must be reached before any output is produced


no output
dial
threshold
output on

The Sigmoid Function

A classic non-linear function is the **Sigmoid function** ($\sigma(t) or\ sig(t)$)





hidden unit or **hidden node**

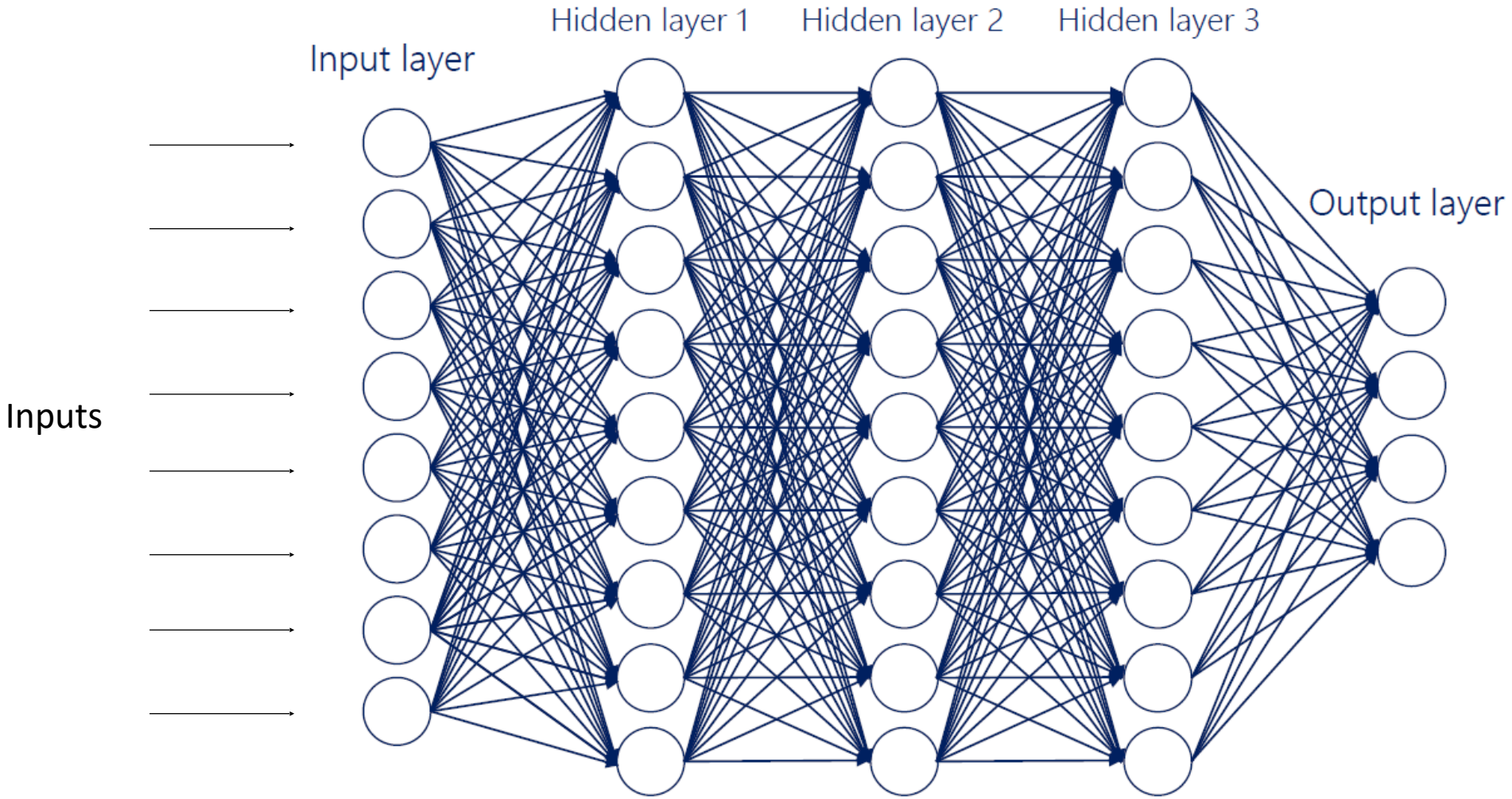
In mathematical terms, if h is the tensor related to the hidden layer, each hidden unit is an element of that **tensor**

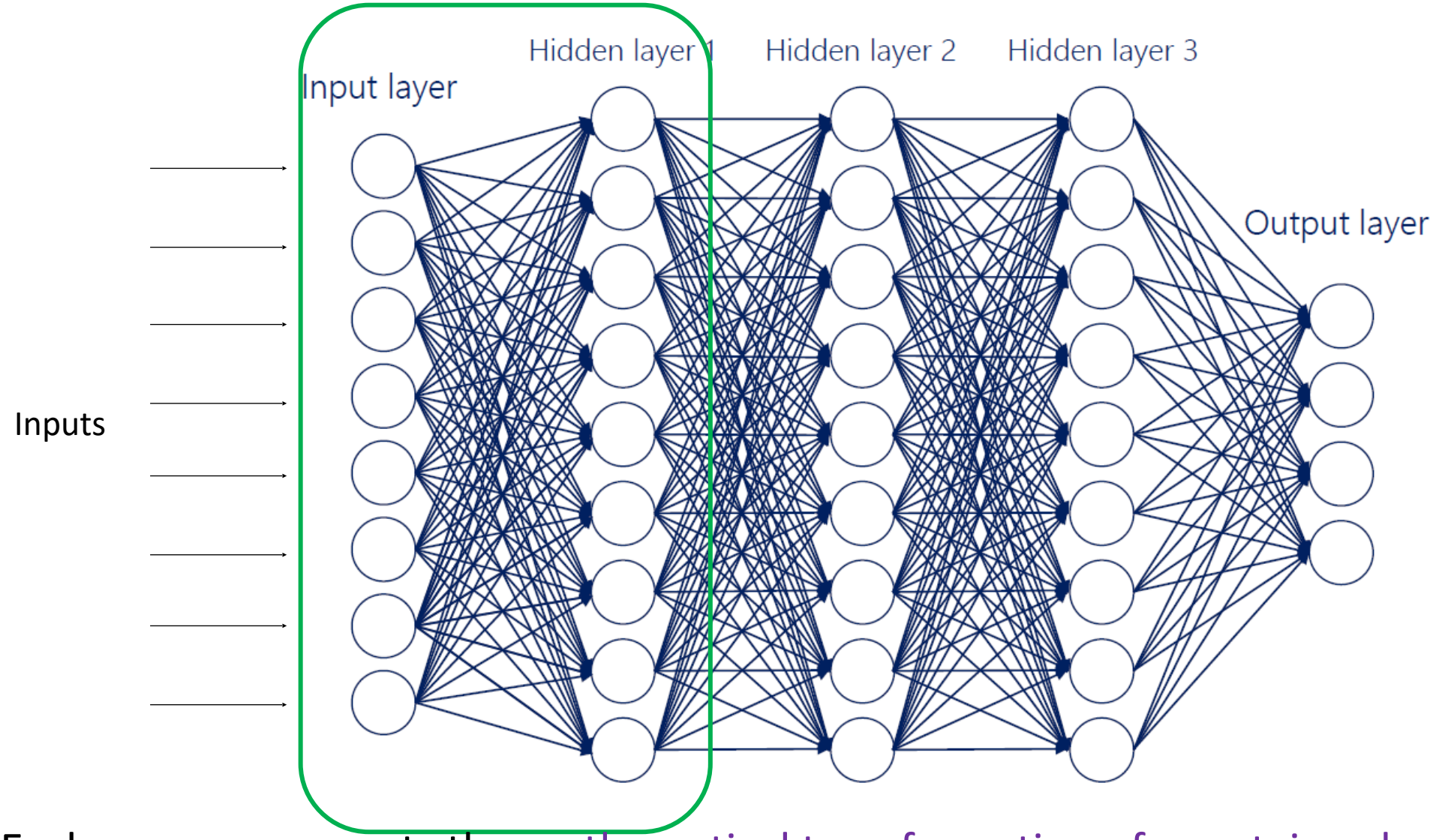
Hyperparameters

- **Width**: number of hidden units per layer
- **Depth**: number of hidden layers in the “deepnet”
- Learning rate

These values (an some more…) are called **hyperparameters**

- They differ from parameters (like weights and biases) because they are not found by optimization, but they are pre-set

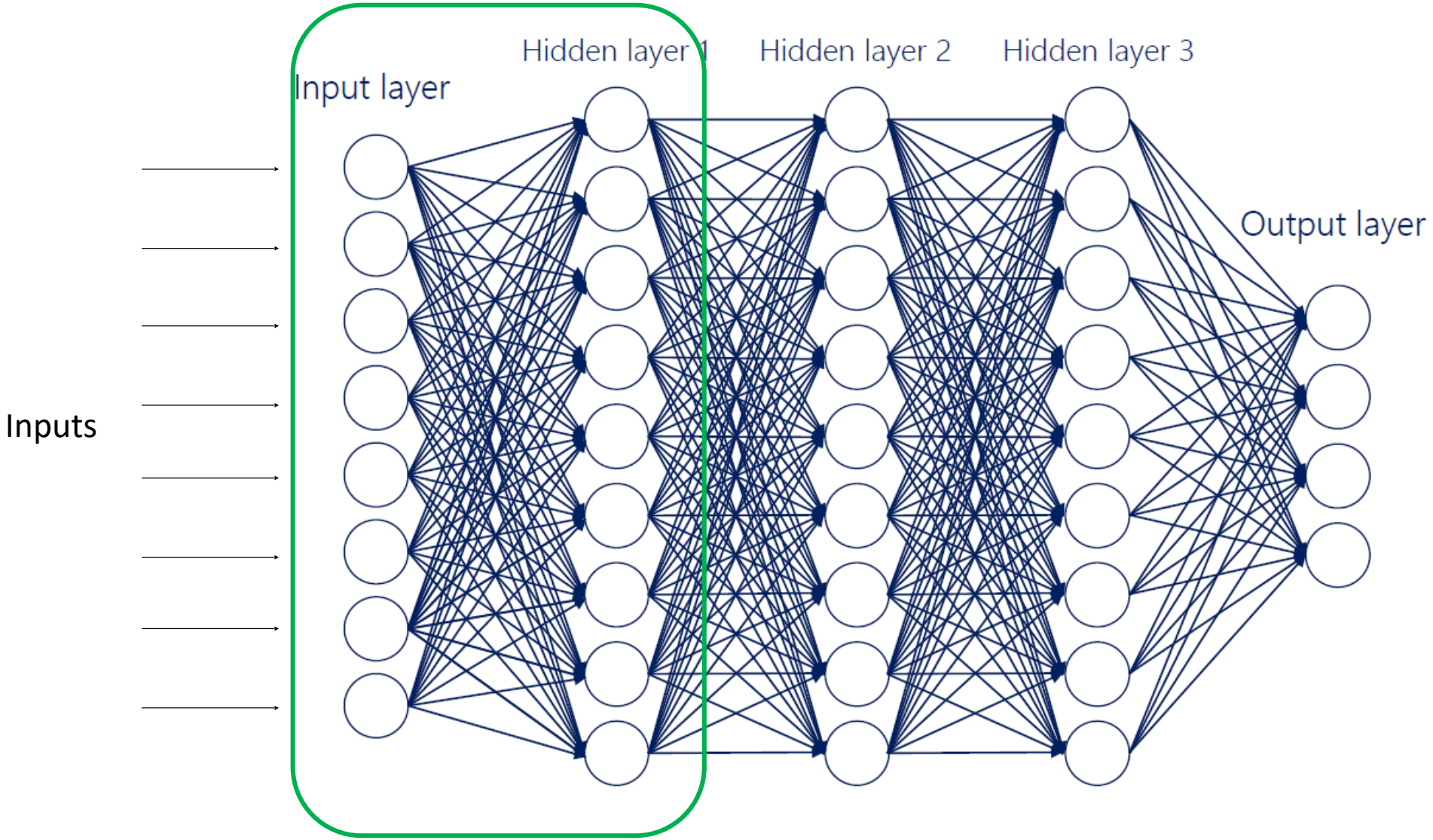

Input layer
Hidden layer 1
Hidden layer 2
Hidden layer 3
Output layer
Inputs



- Each arrow represents the mathematical transformation of a certain value

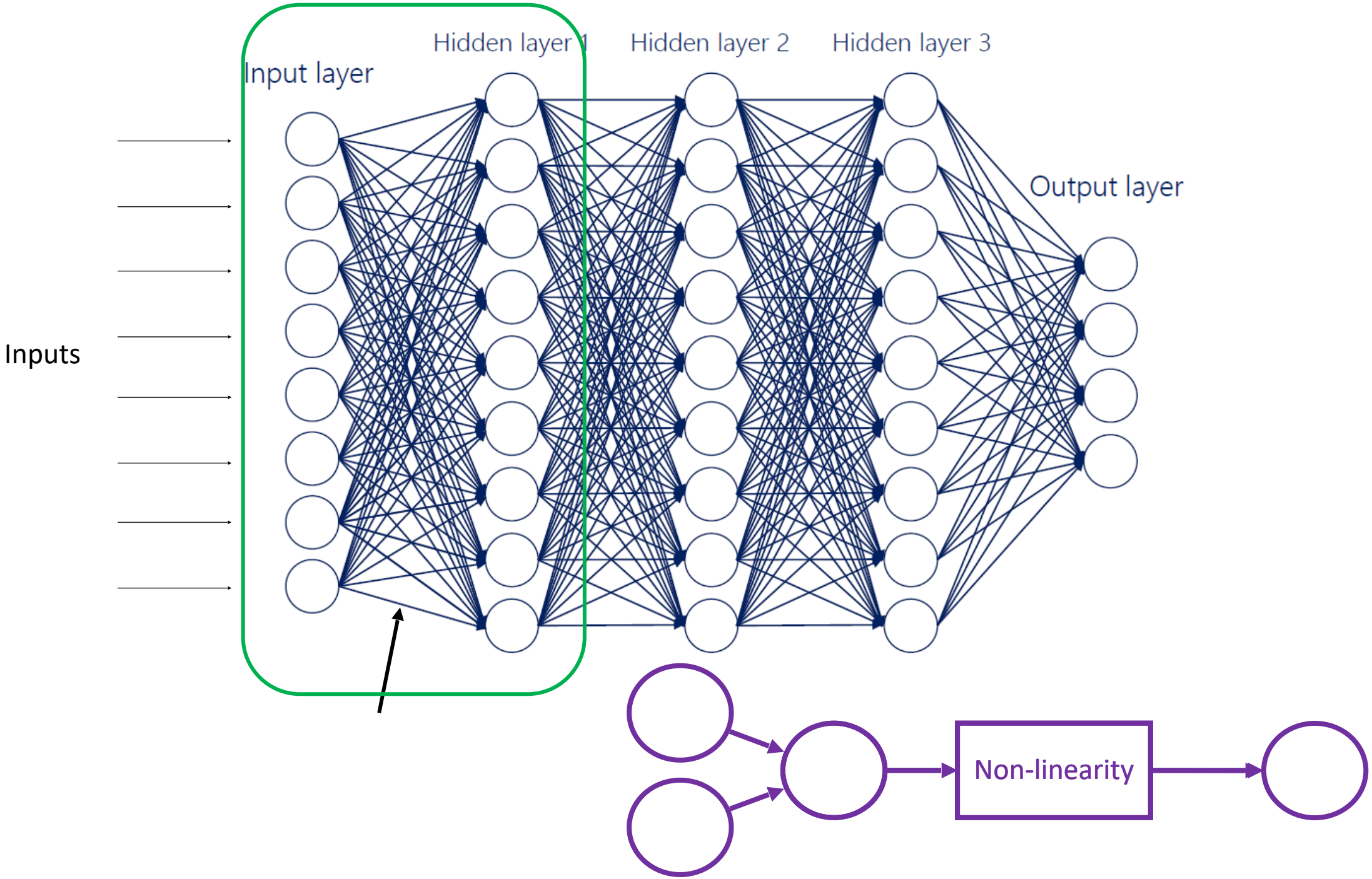
  A logic is added in combination with a non-linearity

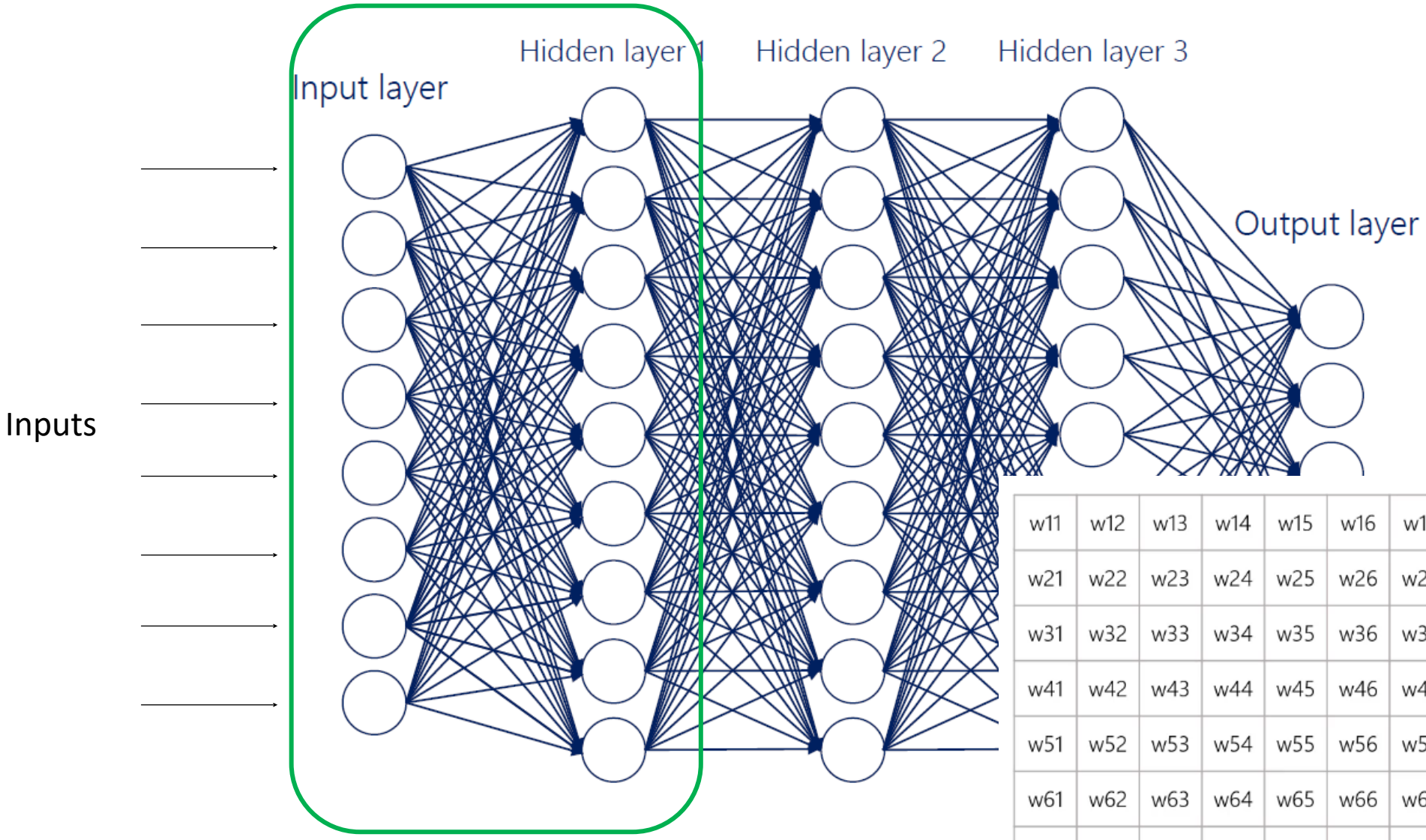
  - Note that the non-linearity doesn't change the shape of the expression it only changes its linearity



We need **linear combination**: **xw + b**

➢ We need weights!

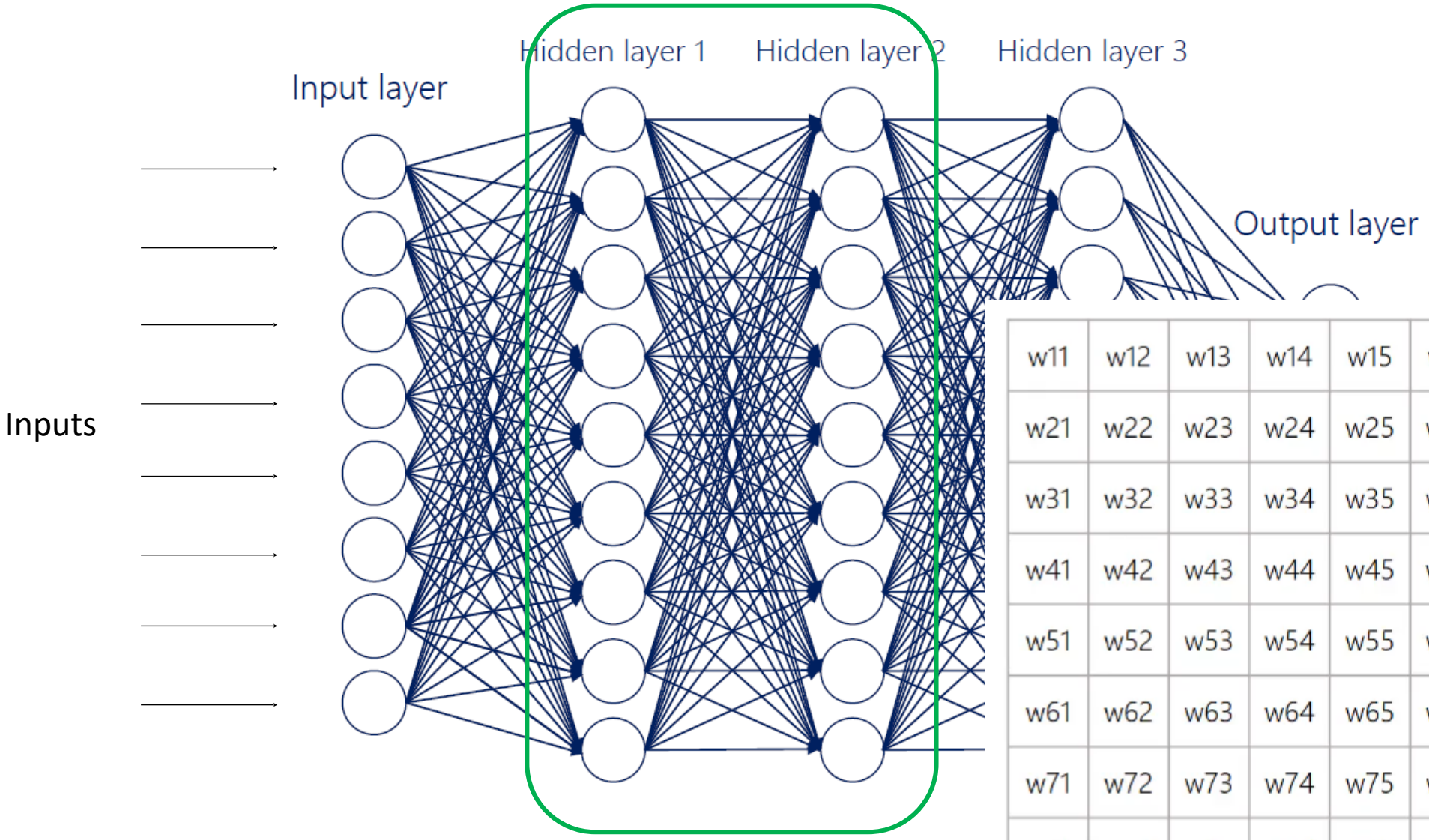

Input layer
Hidden layer 1
Hidden layer 2
Hidden layer 3
Output layer
Inputs
Non-linearity



w11	w12	w13	w14	w15	w16	w17	w18	w19
w21	w22	w23	w24	w25	w26	w27	w28	w29
w31	w32	w33	w34	w35	w36	w37	w38	w39
w41	w42	w43	w44	w45	w46	w47	w48	w49
w51	w52	w53	w54	w55	w56	w57	w58	w59
w61	w62	w63	w64	w65	w66	w67	w68	w69
w71	w72	w73	w74	w75	w76	w77	w78	w79
w81	w82	w83	w84	w85	w86	w87	w88	w89

- We have one weight per each arrow
- Each input node connects to all the hidden nodes in the first hidden layer

8 x 9 = 72



w11	w12	w13	w14	w15	w16	w17	w18	w19
w21	w22	w23	w24	w25	w26	w27	w28	w29
w31	w32	w33	w34	w35	w36	w37	w38	w39
w41	w42	w43	w44	w45	w46	w47	w48	w49
w51	w52	w53	w54	w55	w56	w57	w58	w59
w61	w62	w63	w64	w65	w66	w67	w68	w69
w71	w72	w73	w74	w75	w76	w77	w78	w79
w81	w82	w83	w84	w85	w86	w87	w88	w89
w91	w92	w93	w94	w95	w96	w97	w98	w99

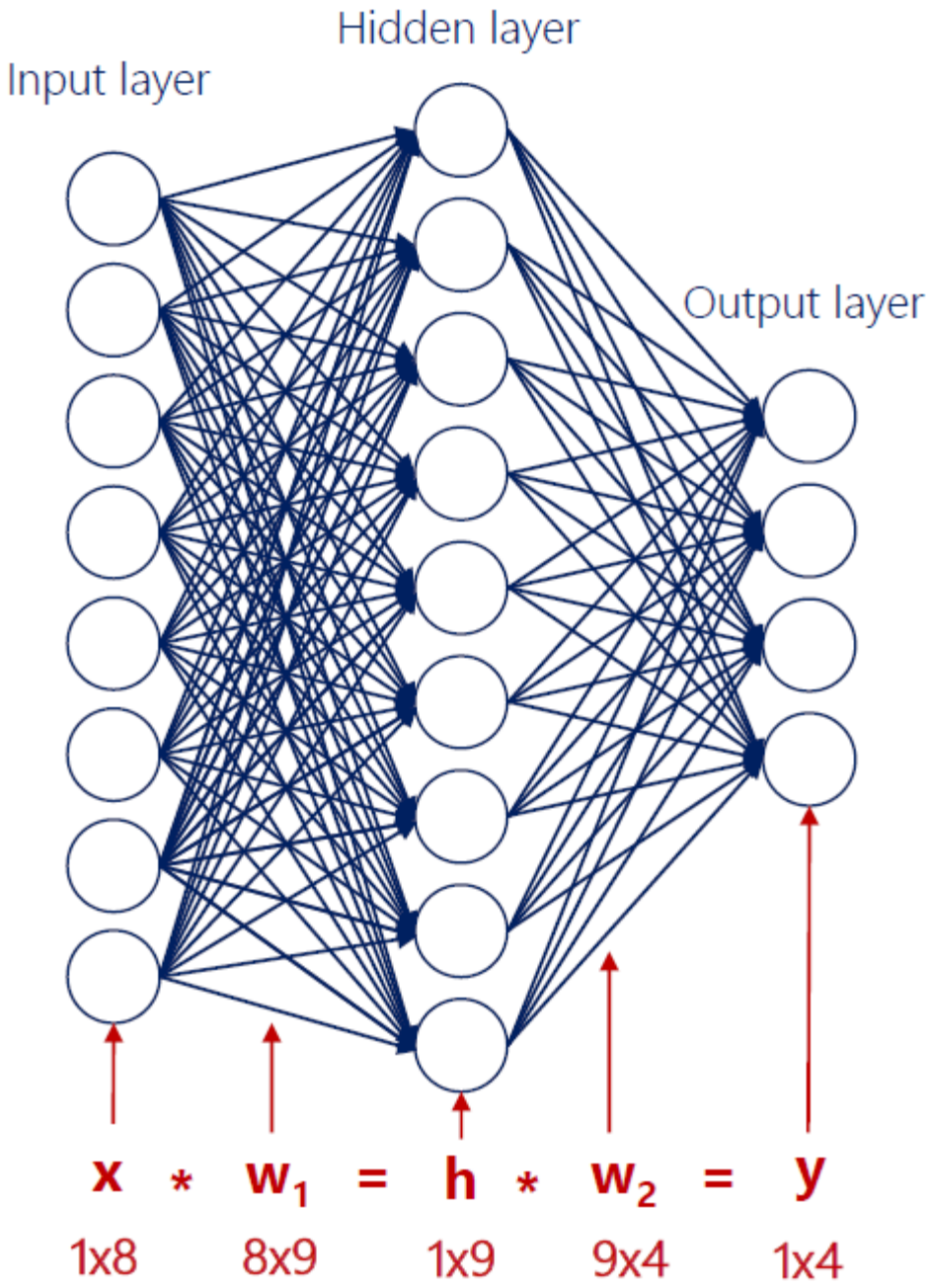
9 x 9 = 81

- Now we connect, following the same logic, the Hidden layer 1 to the Hidden layer 2
- Now we have 81 weights

Is non-linearity required?

- The non-linearity is used to stack together the layers

- What would happen if we didn’t rely on it?



- Linear model (ignoring the biases) between the input and the hidden layer:

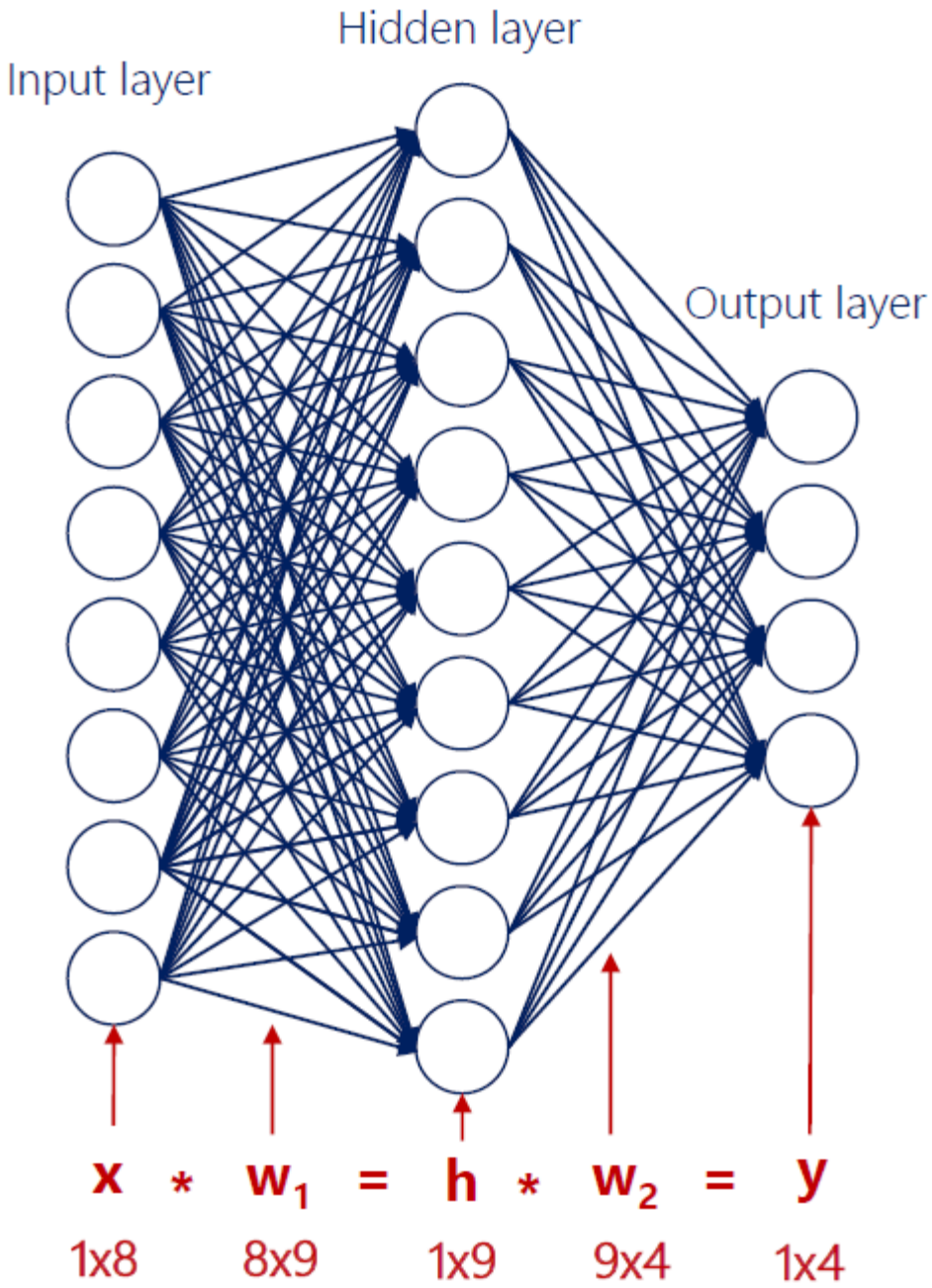
$$h = x * w_1$$

- Linear model (ignoring the biases) between the hidden and output layer:

$$y = h * w_2$$

$$y = x * w_1 * w_2$$

Now, because $\mathbf{w_1}$ is [8x9] and $\mathbf{w_2}$ is [9x4], we can combine them in $\mathbf{w}'$ that becomes [8x4]



- Linear model (ignoring the biases) between the input and the hidden layer:

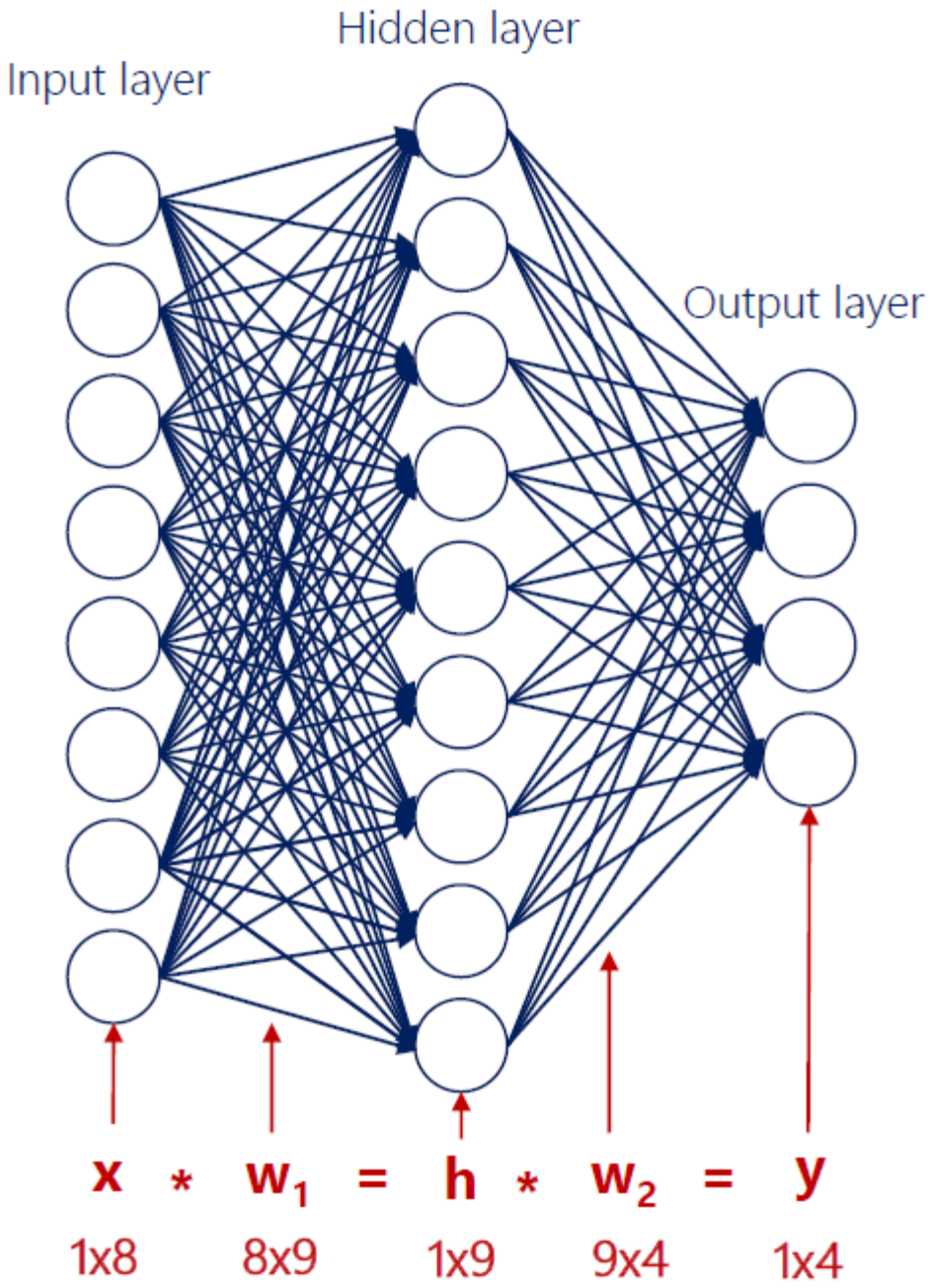
$$h = x * w_1$$

- Linear model (ignoring the biases) between the hidden and output layer:

$$y = h * w_2$$

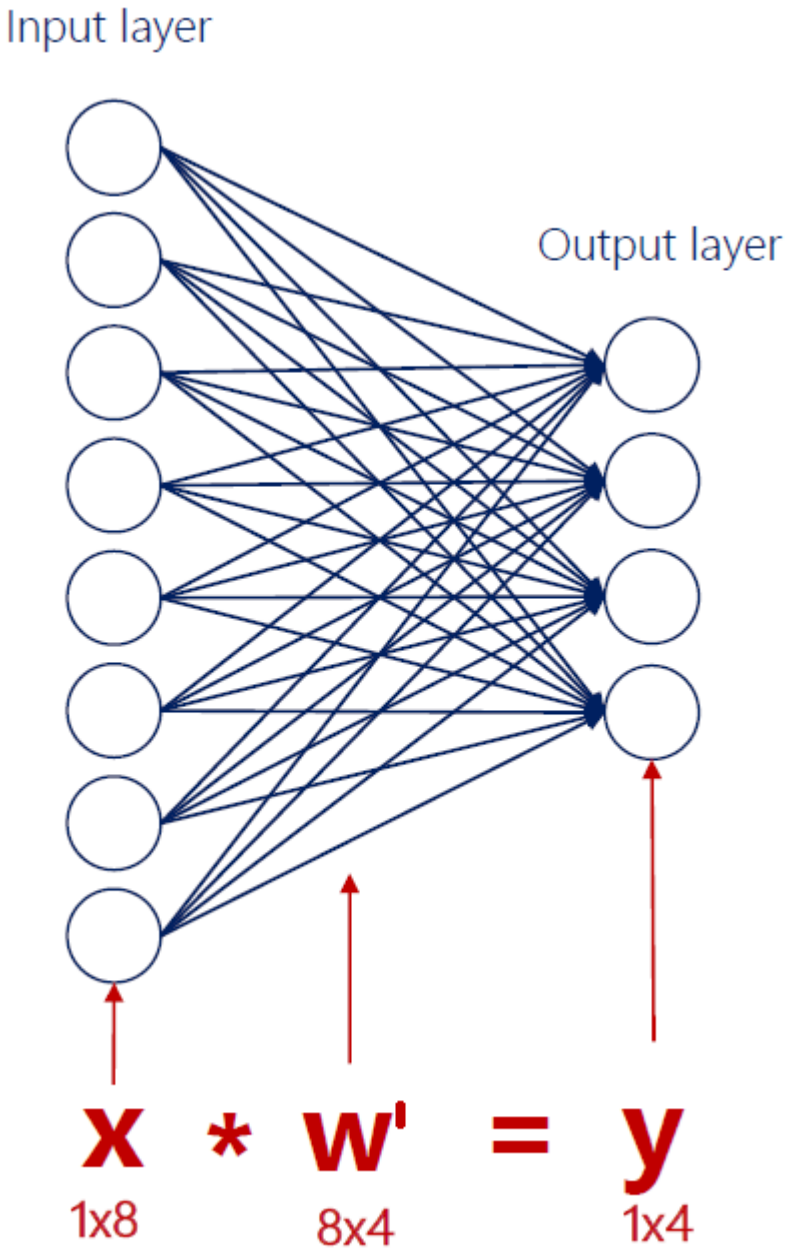
$$y = x * w_1 * w_2$$

$$y = x * w'$$



$$y = x * w'$$

- The two consecutive linear transformation became a single one



$$y = x * w'$$

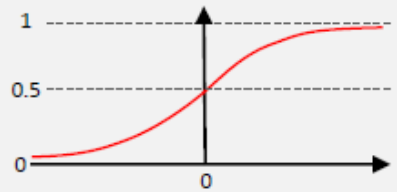
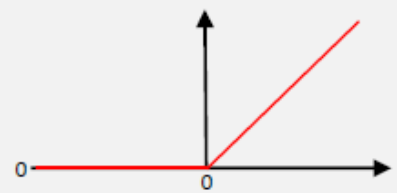
- The two consecutive linear transformation became a single one
- No matter how many hidden layers we have, they would always stack together
- The network returns to be just a simple linear model, incapacitated to solve complex problems

Activation Functions

- The non-linearity is an **Activation Function**

  It tells when the linear model has to “swap”

- These type of functions are **monotonic**, **continuous** and **differentiable**

Name	Formula	Derivative	Graph	Range
sigmoid (logistic function)	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1-\sigma(a))$		(0,1)
TanH (hyperbolic tangent)	$\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^{a}+e^{-a})^2}$		(-1,1)
ReLu (rectified linear unit)	relu(a) = max(0,a)	$\frac{\partial \text{ relu}(a)}{\partial a} = \begin{cases} 0, if\ a \le 0 \\ 1, if\ a > 0 \end{cases}$		(0,∞)
softmax	$\sigma_i(\boldsymbol{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_i(\boldsymbol{a})}{\partial a_j} = \sigma_i(\boldsymbol{a})\left(\delta_{ij} - \sigma_j(\boldsymbol{a})\right)$ Where δ_{ij} is 1 if i=j, 0 otherwise		(0,1)

Let's focus on the **softmax** function

- This function accepts in input an entire vector
- Divide the exponential of each term by the sum of the exp. of the entire vector

  It basically **takes in consideration all the set of values**
- The output of this function would fit a probability distribution

  You basically compute an average of each value, so the sum of outputs is "node stochastic"

softmax	$\sigma_i(\boldsymbol{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_i(\boldsymbol{a})}{\partial a_j} = \sigma_i(\boldsymbol{a})\left(\delta_{ij} - \sigma_j(\boldsymbol{a})\right)$ Where δ_{ij} is 1 if i=j, 0 otherwise		(0,1)

Softmax

- The **softmax** activation transforms a bunch of arbitrarily large or small numbers into a valid **probability distribution**

- While other activation functions get an input value and transform it, regardless of the other elements, the **softmax** considers the information about the **whole set of numbers** we have

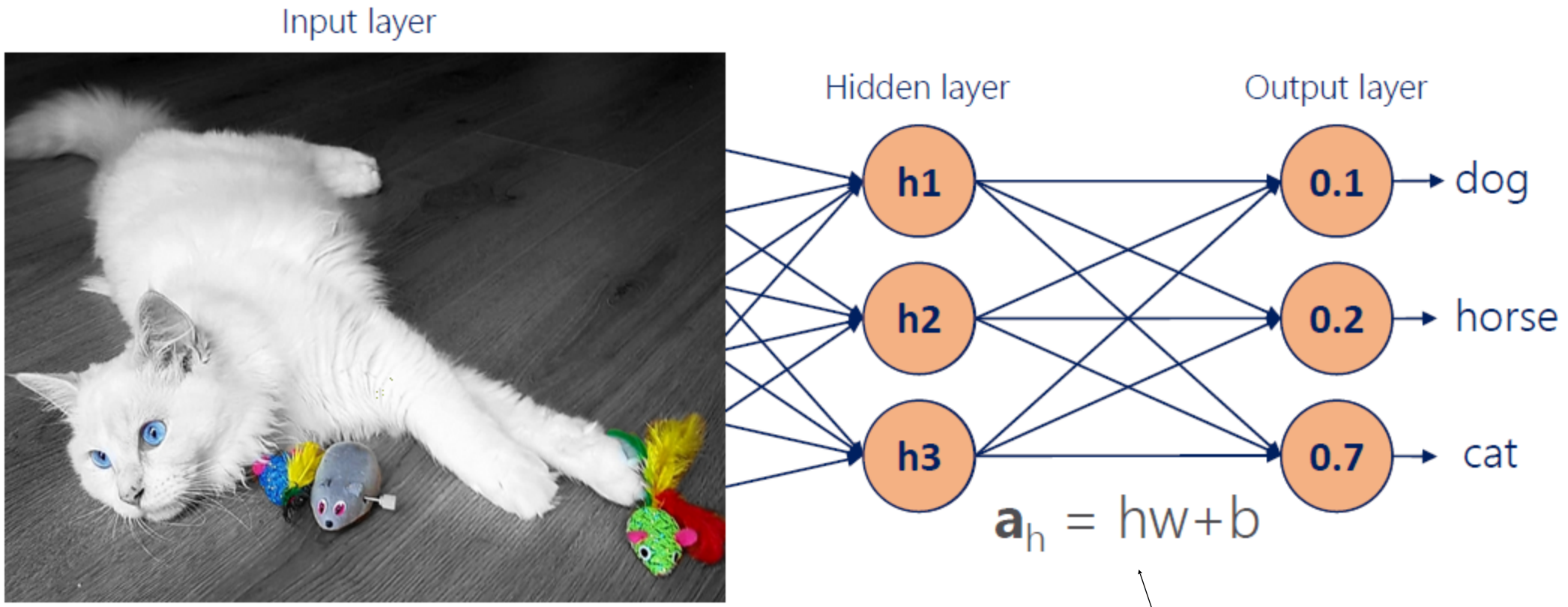
- The values that **softmax** outputs are in the range from 0 to 1 and their sum is exactly 1 (like probabilities)

Softmax

- The property of the **softmax** to output probabilities is so useful that it is often used as the activation function for the **output layer** for **classification** problems

- However, when the **softmax** is used prior to that (as the activation of a hidden layer), the results are not as satisfactory

  That's because a lot of the information about the variability of the data is lost



This is the Linear combination formula on top of which we apply the activation function

Backpropagation

- The **training** process consists of updating parameters through the gradient descent for optimizing the **objective function** in supervised learning

- The process of optimization consisted of minimizing the loss

  The updates were directly related to the partial derivatives of the loss and indirectly related to the errors (or **deltas**)

$$w_{i+1} = w_i - \eta \nabla_w L(y, t)$$
$$b_{i+1} = b_i - \eta \nabla_b L(y, t)$$

Backpropagation

- We defined deltas as the difference between output and corresponding target

- The delta, in this case, becomes:

$$\delta_{i,\ output} = y_i - t_i$$

However, for the hidden layers is not so intuitive

$$\delta_{i,\ hidden} = ?$$

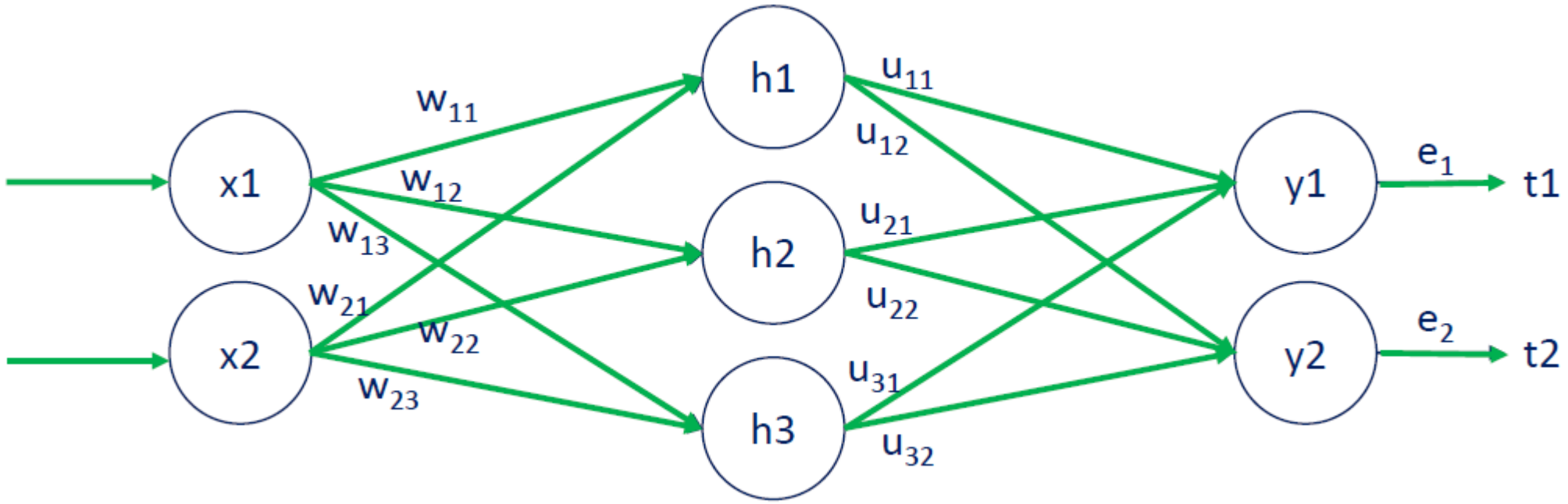
Backpropagation

- Computing the deltas in the hidden layers is obtained through the **backpropagation of errors**

- Before we get there, let's see **forward propagation**

- **Forward propagation** is the process of pushing inputs through the net

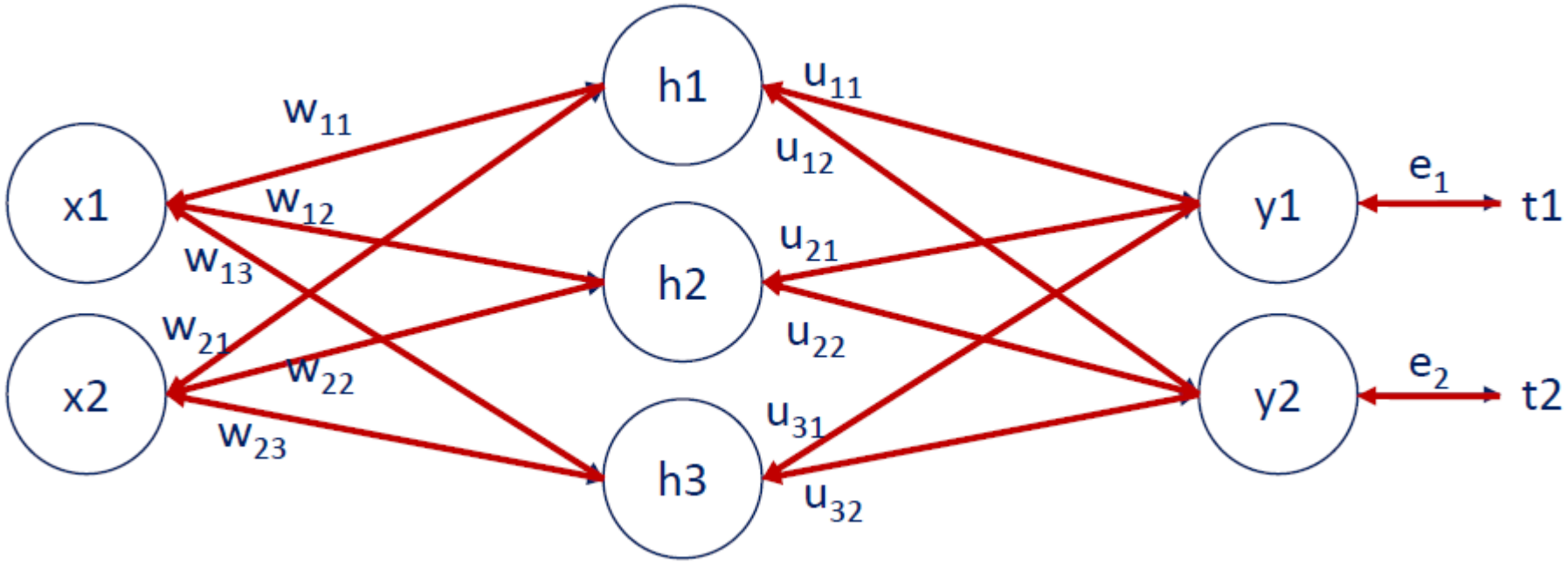
- At the end of each epoch the obtained outputs are compared to the targets to form the errors

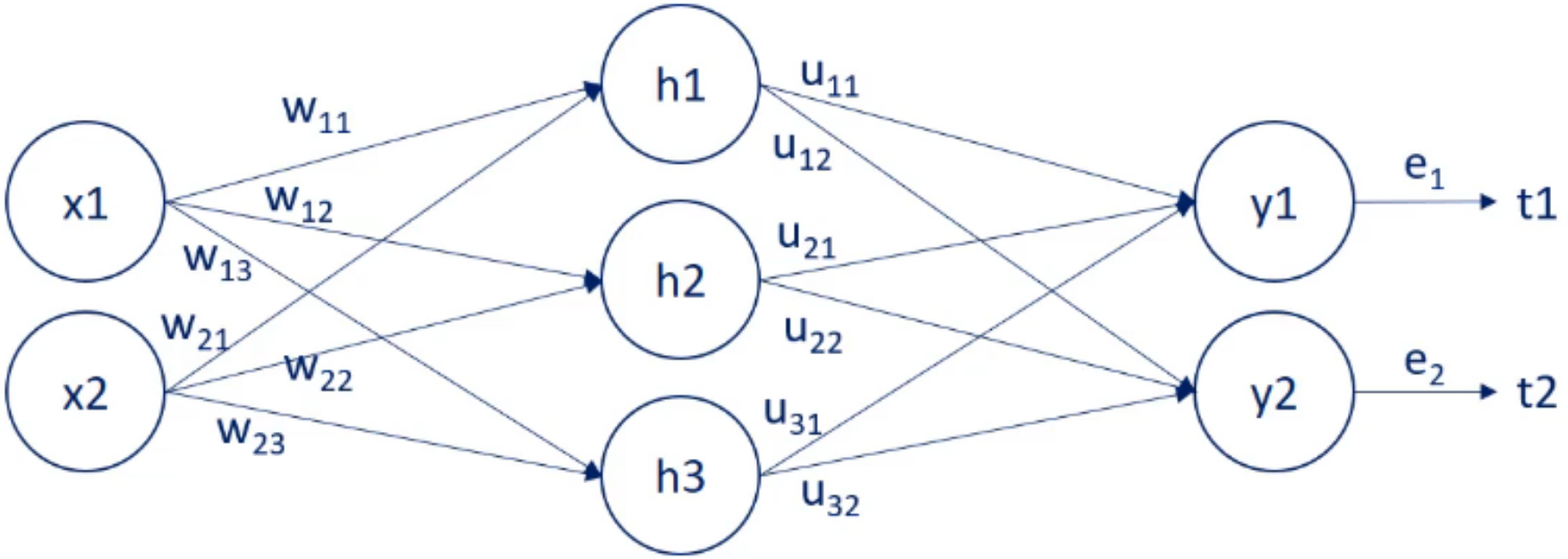


The u_{ii} you see are the weights after a specific hidden layer

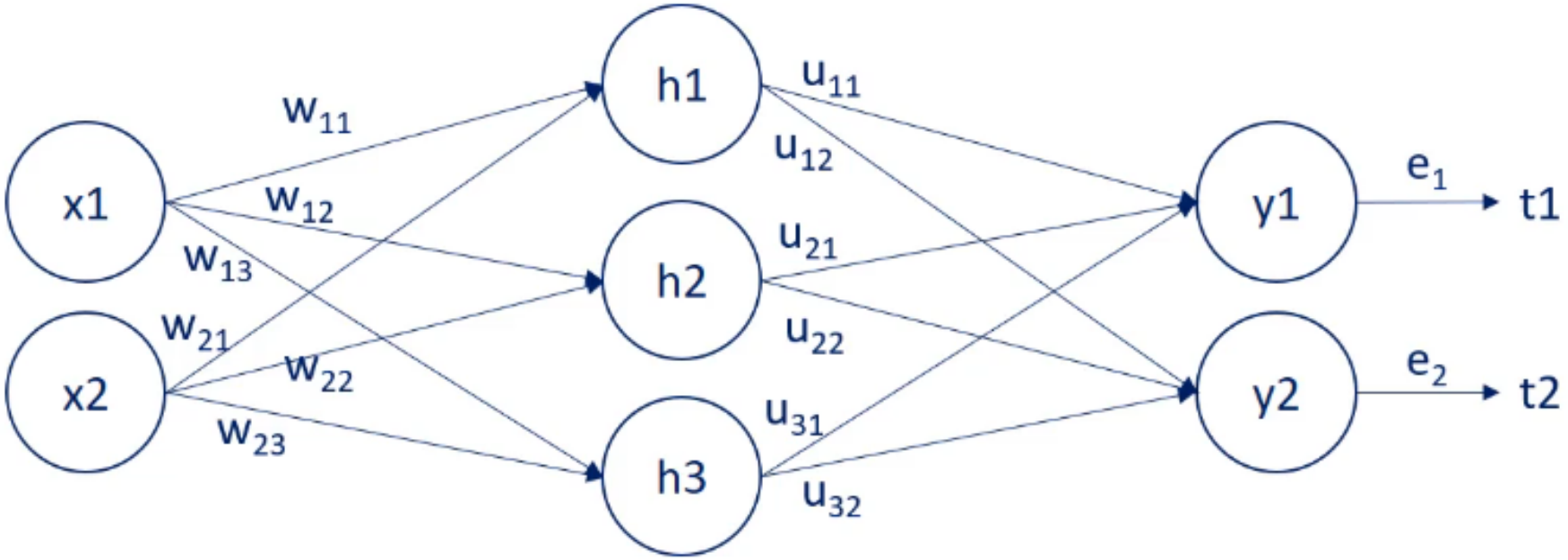
Backpropagation

- Then, we back propagate through partial derivatives and change each parameter so errors at the next epoch are minimized





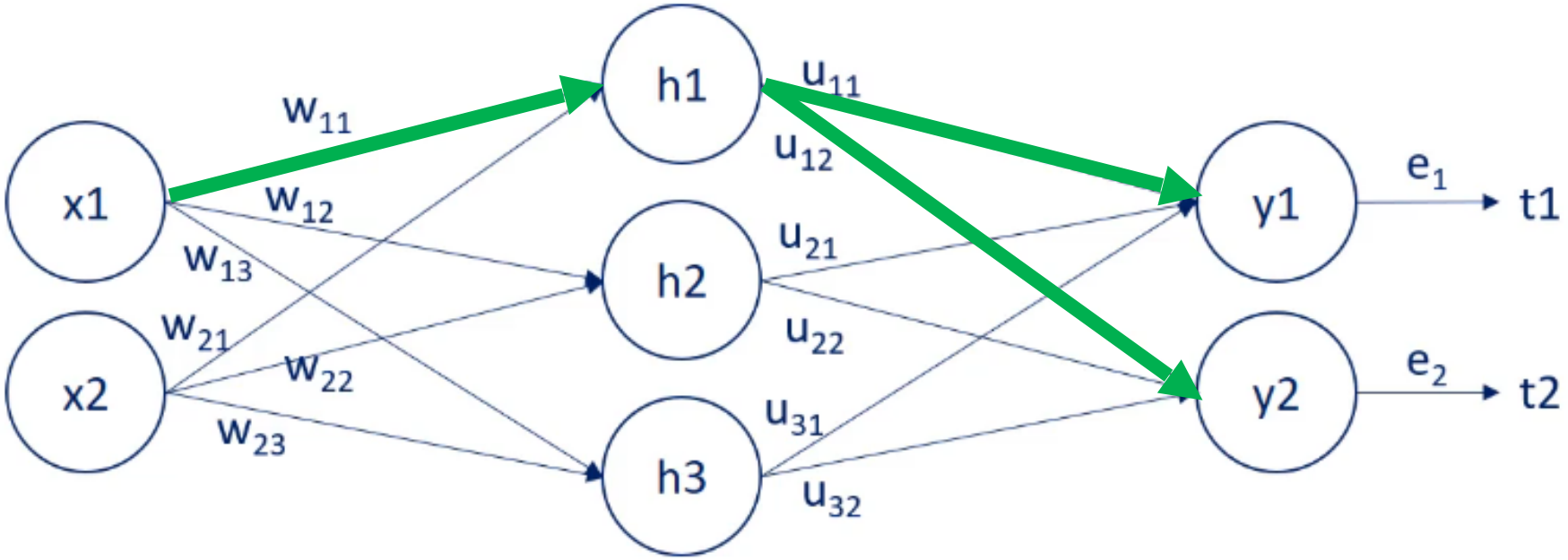
- We have e_1 that is the error on the output y_1 in relation to the target t_1
- Which weights are contributing to this error?
- The weights are (follow the arrows): u_{11}, u_{21} and u_{31}
- Let's focus on u_{11}



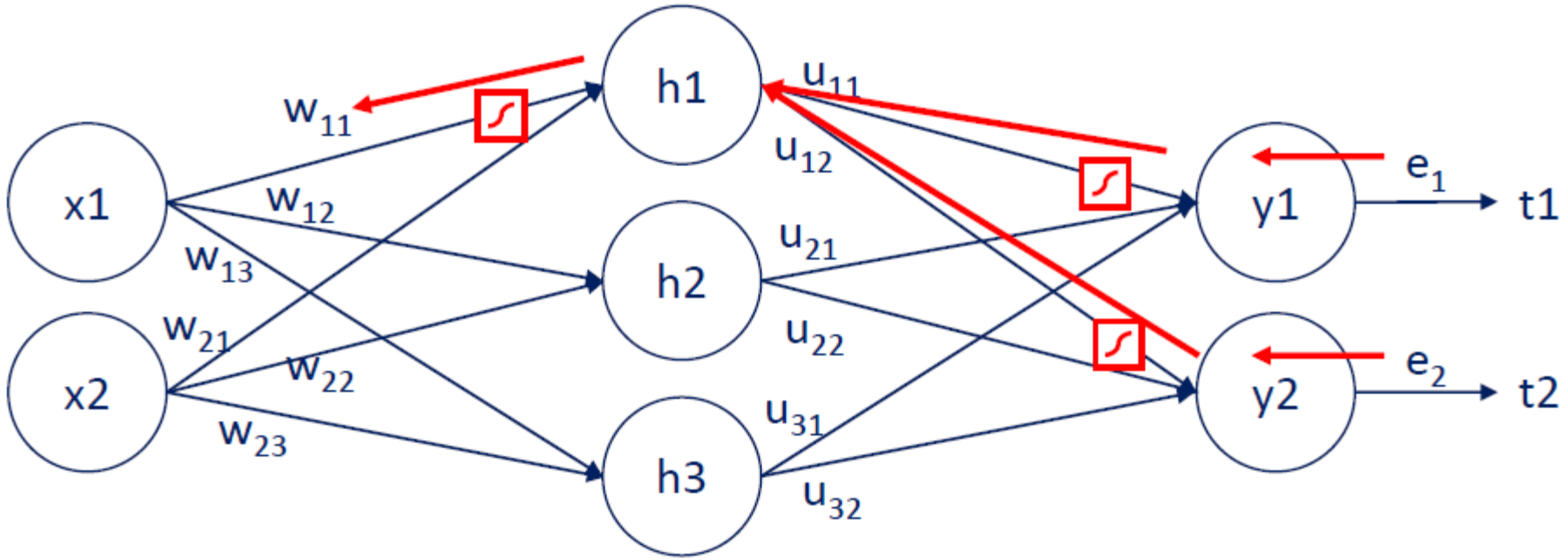
- Each u_{ii} contributes to a single error
- So, we find its derivative and update the coefficient

  This is the same thing we have done applying the gradient descent before

$$w_{i+1} = w_i - \eta \nabla_w L(y, t) \quad \Rightarrow \quad u_{11_{i+1}} = u_{11_i} - \eta \nabla_{u_{11}} L(y_1, t_1)$$



- Now, think about w_{11}
- This weight is required to predict $\mathrm{h1}$
- Then, we used $h1$ to calculate the weights u_{11} and u_{12}
- Finally, w_{11} plays a role in determining the two errors e_1 and e_2



- So, we back propagate to measure the influence on the errors of each weight w_{ii} with the information given by the corresponding u_{ii}

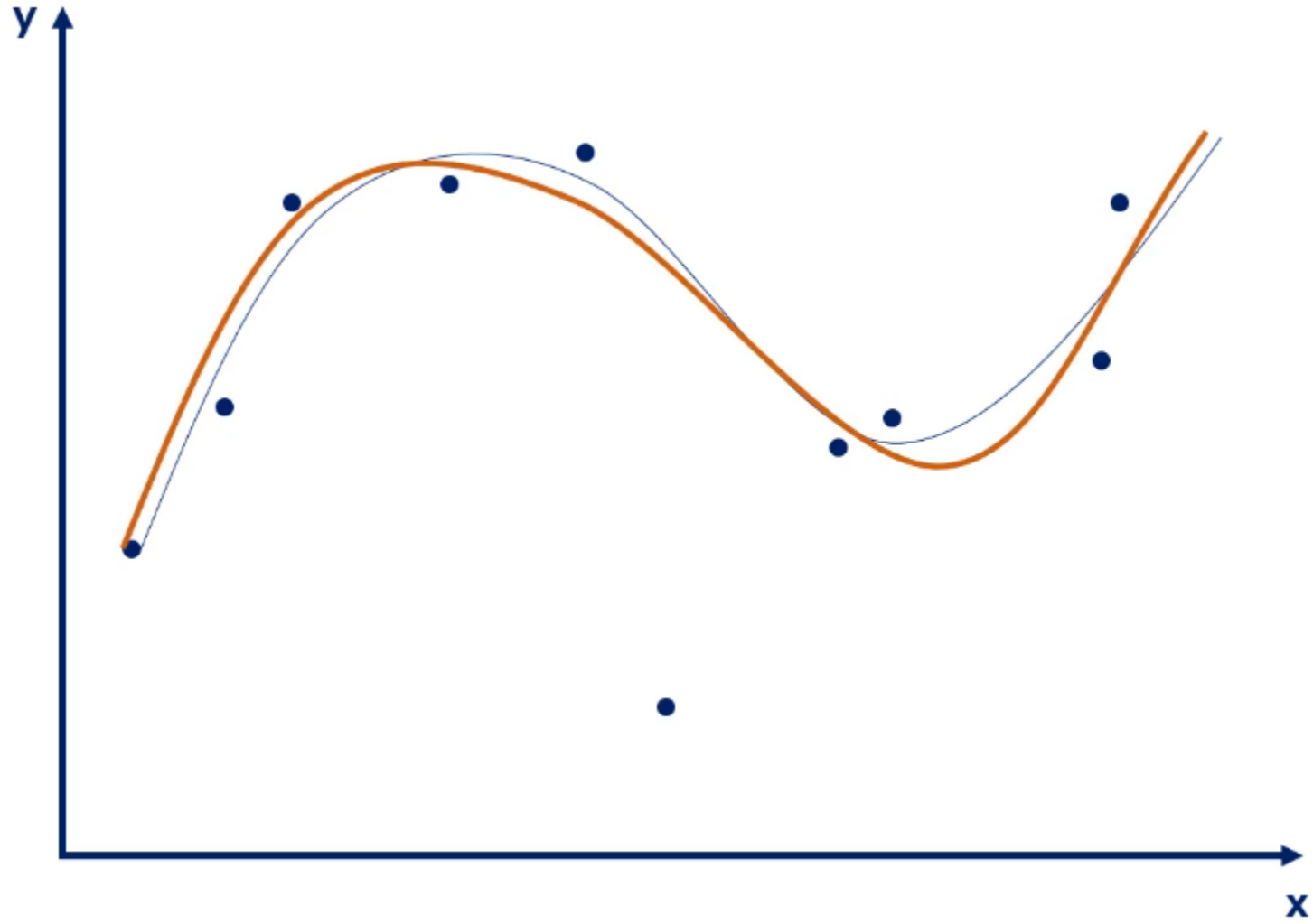
1. Essentially, through back propagation, the algorithm identifies which weights lead to which errors
2. Then, it adjusts the weights that have a bigger contribution to the errors by more than the weights with a smaller contribution.

Overfitting and underfitting

Overfitting and underfitting

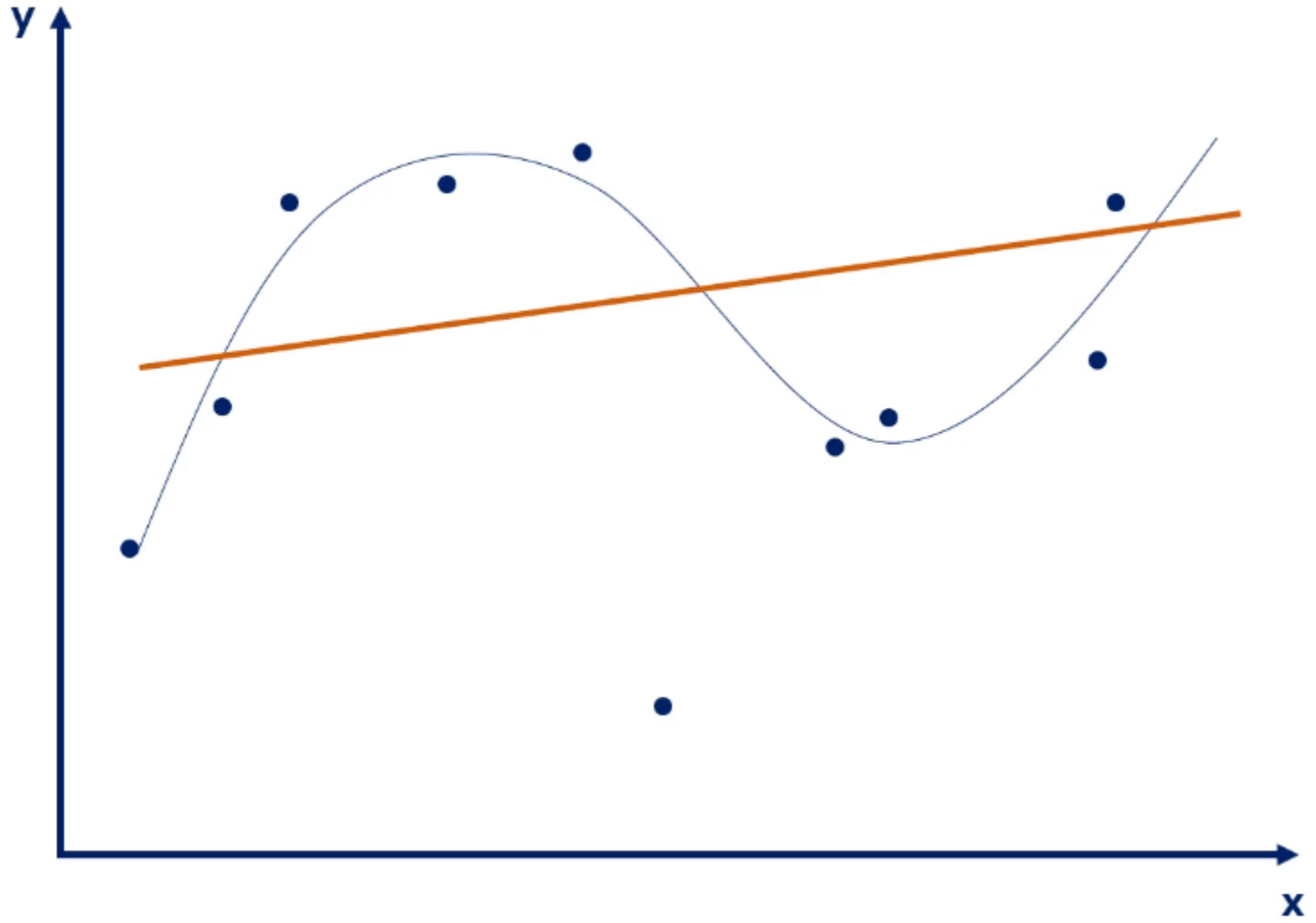
- **Overfitting** indicates that our training has focused on the particular training set so much that has missed the point
  - Detecting overfitting is very tough

- **Underfitting**, on the other hand, indicates that the model has not captured the underlying logic of the data
  - Basically, the model doesn’t know what to do and the answer is far from the expectations



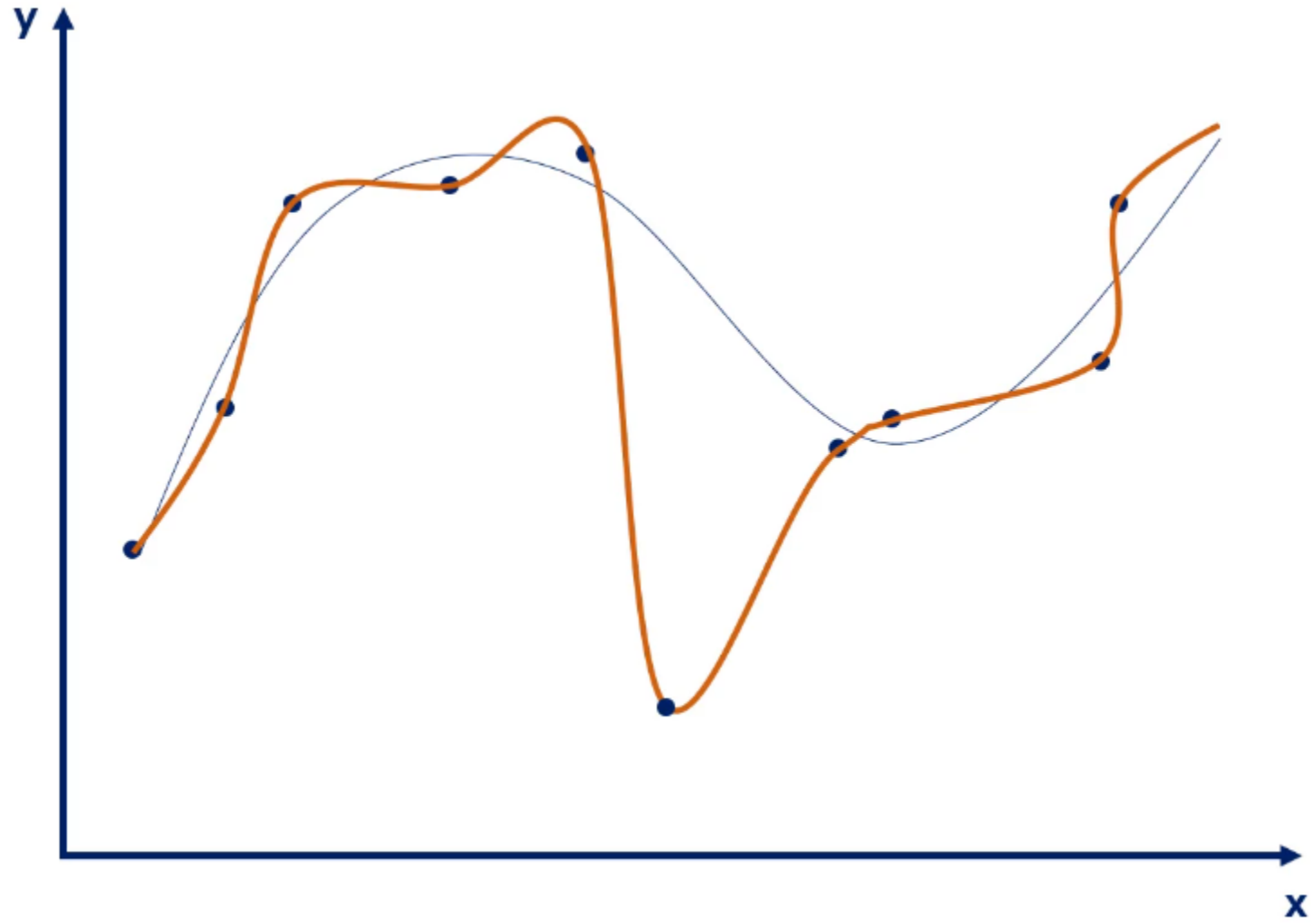
- This is an example of a good model

It’s not perfect, of course, but is very close to the actual relationship



- This is an example of **underfitting**

Accuracy is low and the loss function returns very high values



- This is an example of **overfitting**

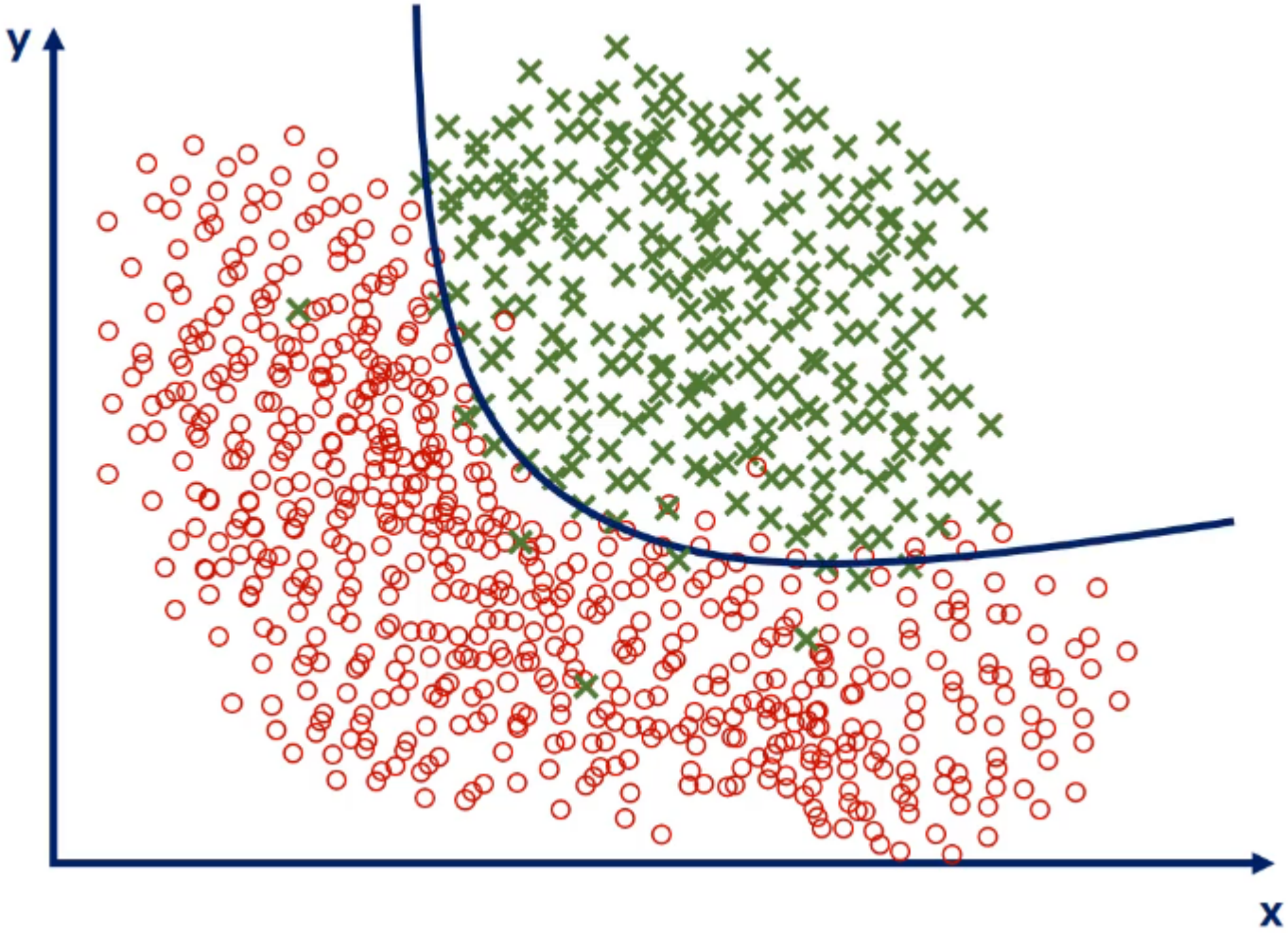
All the random noise is captured in the model

The underlying relationship is “not understood”

NOTE: with an overfitting model you get low Losses and high accuracy!!

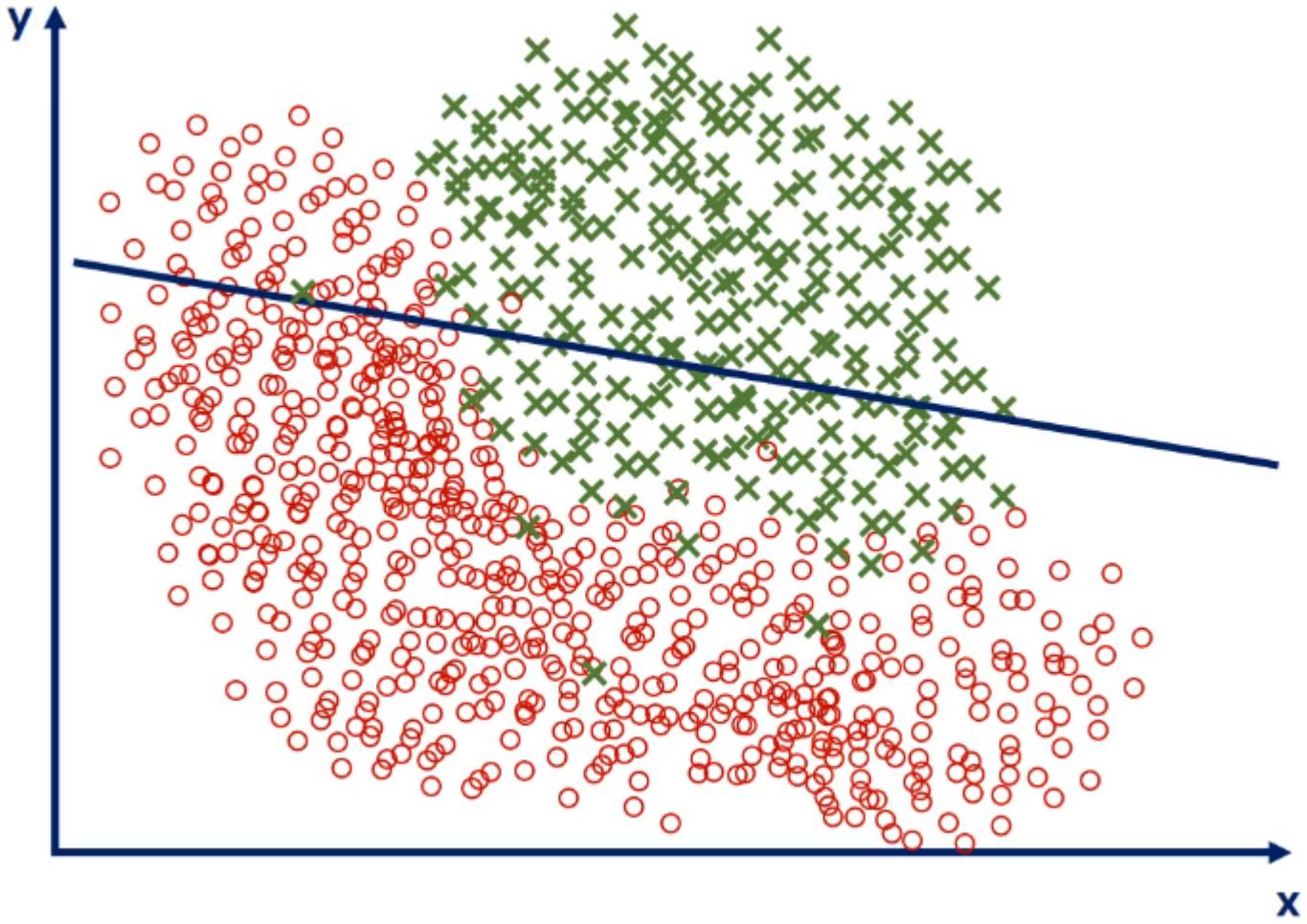
Overfitting and underfitting

- Let’s see a classification example now



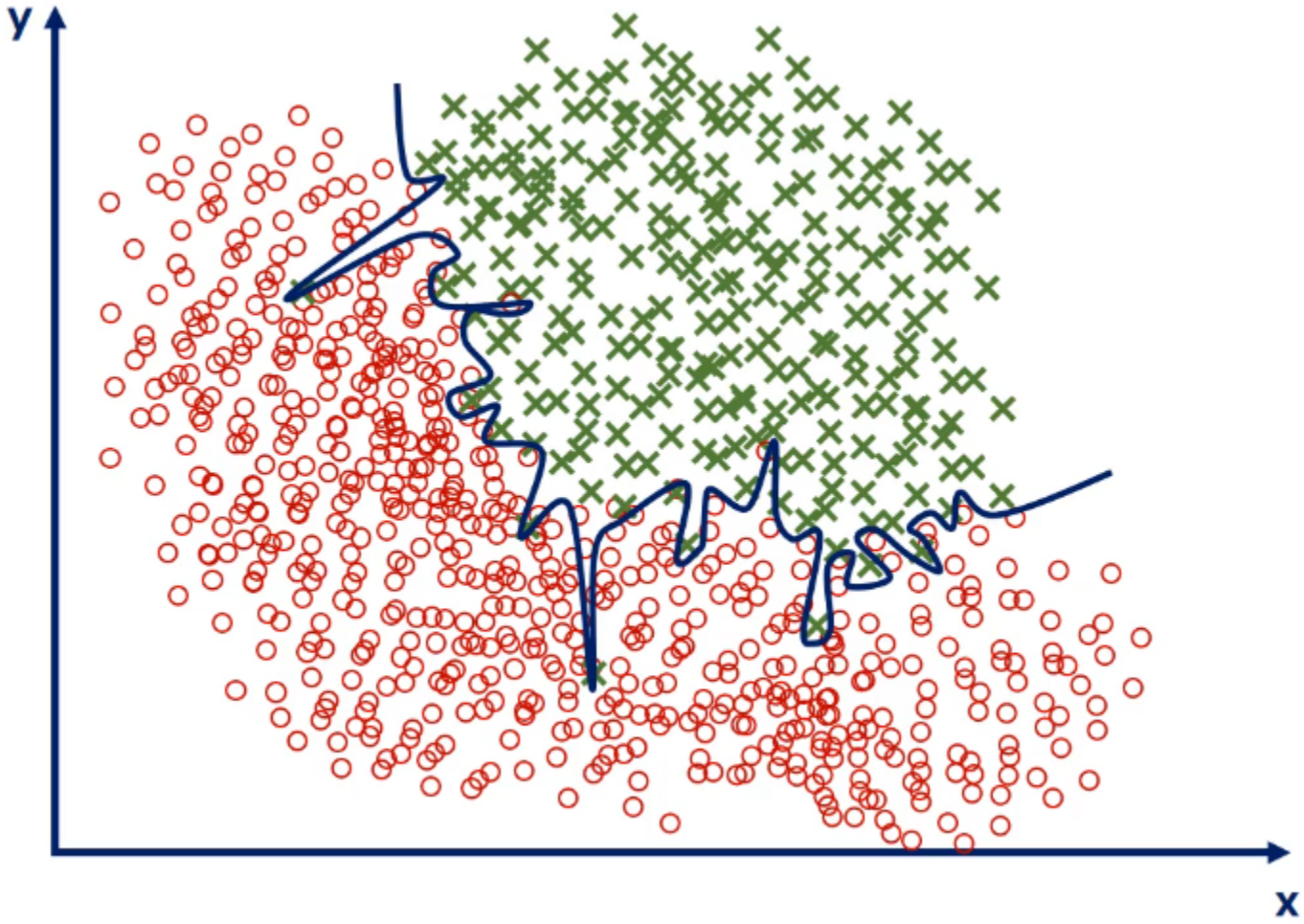
- This is an example of a good model

We have a quadratic function that classifies with few errors



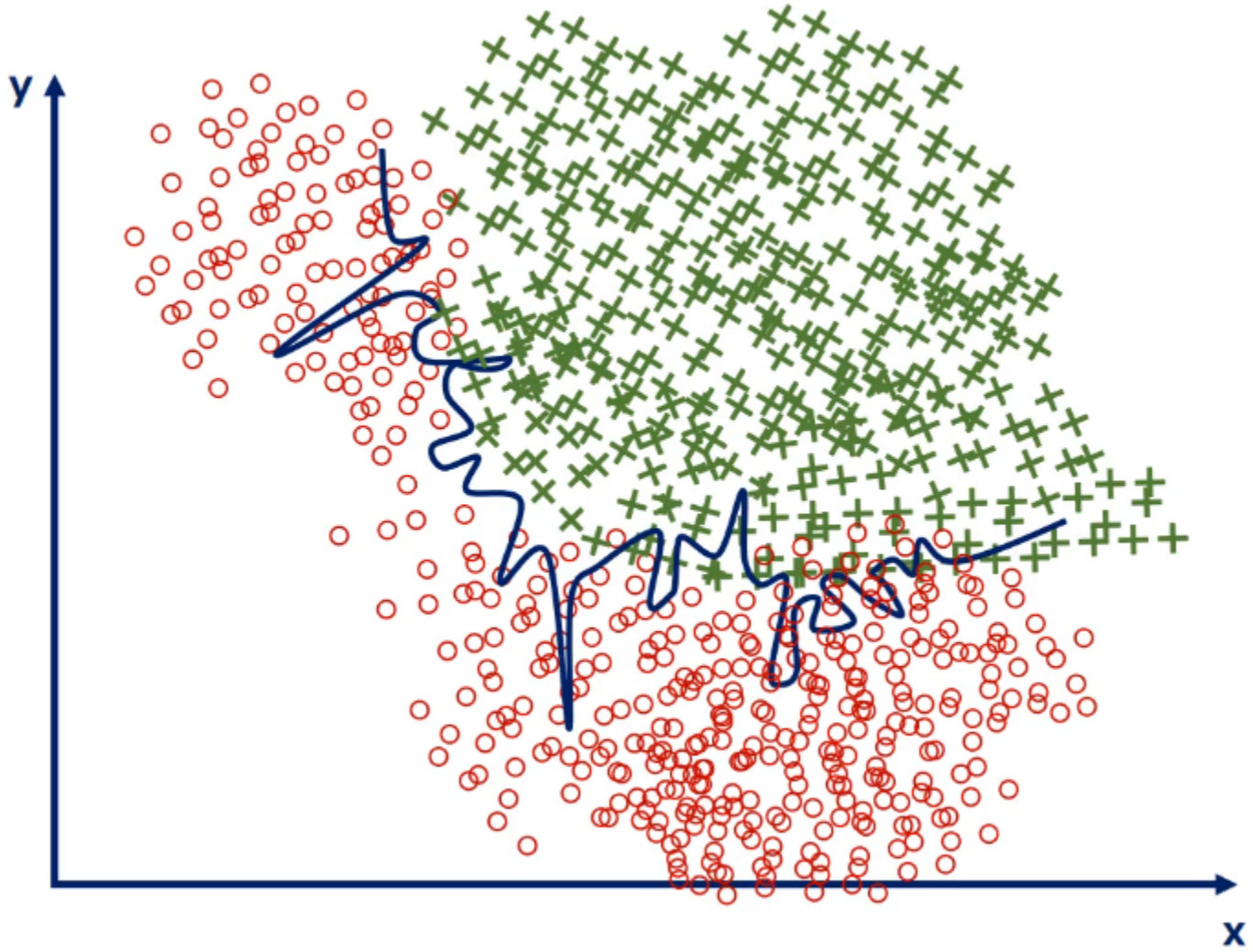
- This is an example of **underfitting**

A linear model would classify some observations correctly



- This is an example of **overfitting**

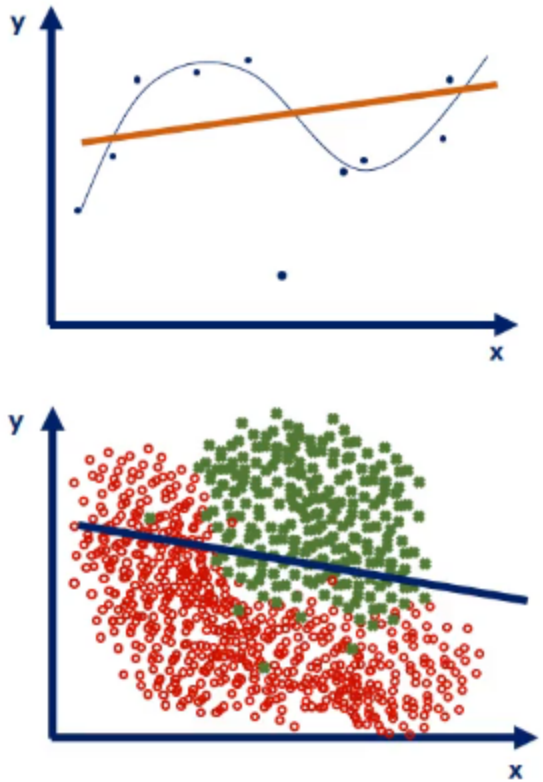
It’s perfect!! But…



- This is an example of **overfitting**

When the observations change… it’s not so perfect anymore

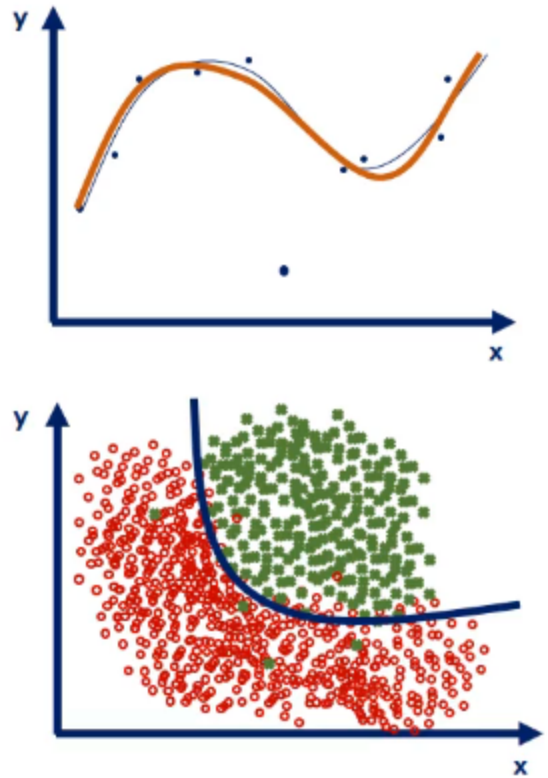
An **underfitted** model



Doesn't capture any logic

- High loss
- Low accuracy

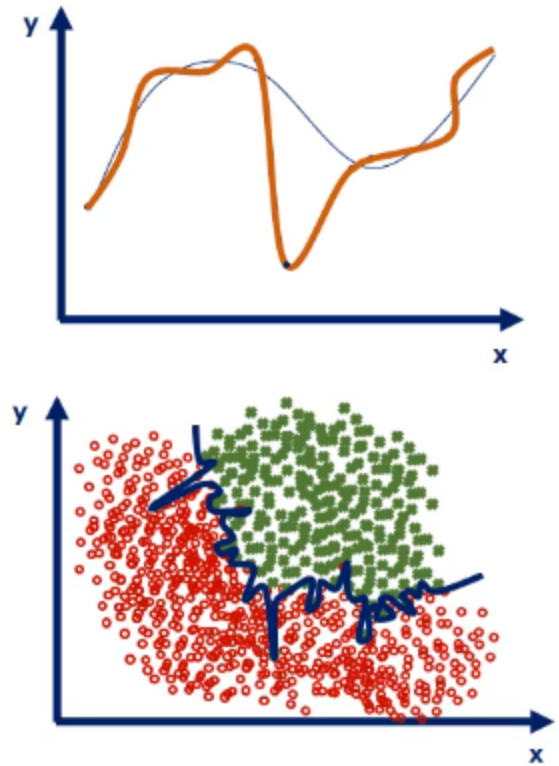
A **good** model



Captures the underlying logic of the dataset

- Low loss
- High accuracy

An **overfitted** model



Captures all the noise, thus "missed the point"

- Low loss
- Low accuracy

Overfitting and underfitting

What is a well-trained model, then?

➢It’s a tradeoff between under and overfitting

- This fine balance is often called the **bias variance tradeoff**

Training, Validation and Test

To avoid overfitting, we can split our dataset in three parts:

- **Training**
- **Validation**
- **Test**

- There is no fixed rule, but the split is often:
  - 80% - 10% - 10%
  - 70% - 20% - 10%

- It also depends on the type and size of the dataset

Training, Validation and Test

To avoid overfitting, we can split our dataset in three parts:

- **Training**
- **Validation**
- **Test**

1. We train on the Training set
2. We test the model against the Validation set with just forward propagation (after every epoch)
3. We get the Loss function results
   - This **Validation Loss** should be close to the **Training Loss**
   - Note that Validation and Training sets belong to the same dataset, so they represent the same relationship and have the same dependencies

Training, Validation and Test

To avoid overfitting, we can split our dataset in three parts:

- **Training**
- **Validation**
- **Test**

4. Finally, we test the model against the Test set

   The accuracy that we got here is the actual accuracy of the model

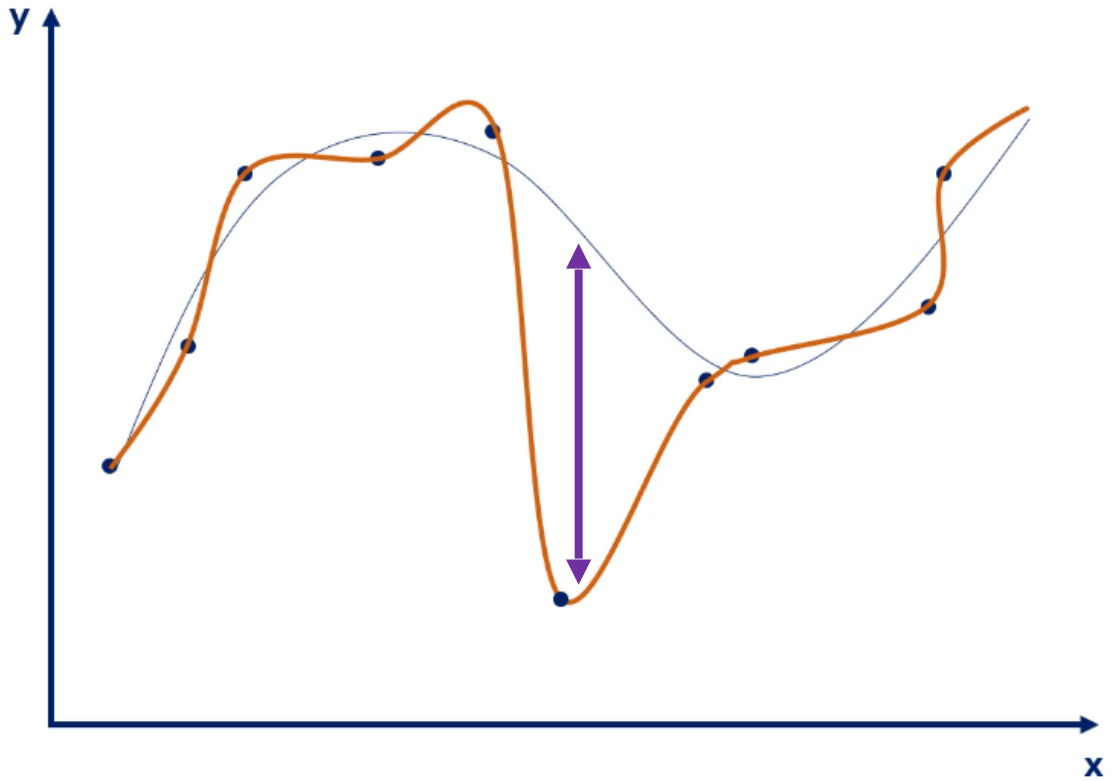
Training, Validation and Test

To avoid overfitting, we can split our dataset in three parts:

- **Training**
- **Validation**
- **Test**

- We should be able to see that the Training Loss gets closer and closer to 0
- However, if the Validation Loss starts to increase (at any point), then we have a problem… an overfitting problem!
  - Basically, the model has become very good at predicting the training set, but is getting far from the actual relationship

```
Epoch 1. Training loss: 0.341. Validation loss: 0.192
Epoch 2. Training loss: 0.164. Validation loss: 0.143
Epoch 3. Training loss: 0.115. Validation loss: 0.106
Epoch 4. Training loss: 0.086. Validation loss: 0.093
Epoch 5. Training loss: 0.068. Validation loss: 0.087
Epoch 6. Training loss: 0.054. Validation loss: 0.084
Epoch 7. Training loss: 0.042. Validation loss: 0.074
Epoch 8. Training loss: 0.033. Validation loss: 0.077
Epoch 9. Training loss: 0.029. Validation loss: 0.090
Epoch 10. Training loss: 0.022. Validation loss: 0.072
Epoch 11. Training loss: 0.018. Validation loss: 0.074
Epoch 12. Training loss: 0.014. Validation loss: 0.075
Epoch 13. Training loss: 0.011. Validation loss: 0.081
Epoch 14. Training loss: 0.009. Validation loss: 0.085
Epoch 15. Training loss: 0.007. Validation loss: 0.085
Epoch 16. Training loss: 0.009. Validation loss: 0.091
Epoch 17. Training loss: 0.008. Validation loss: 0.099
Epoch 18. Training loss: 0.006. Validation loss: 0.100
Epoch 19. Training loss: 0.005. Validation loss: 0.092
Epoch 20. Training loss: 0.002. Validation loss: 0.084
Epoch 21. Training loss: 0.002. Validation loss: 0.092
Epoch 22. Training loss: 0.016. Validation loss: 0.085
Epoch 23. Training loss: 0.005. Validation loss: 0.094
Epoch 24. Training loss: 0.001. Validation loss: 0.085
Epoch 25. Training loss: 0.001. Validation loss: 0.088
Epoch 26. Training loss: 0.000. Validation loss: 0.087
Epoch 27. Training loss: 0.000. Validation loss: 0.089
Epoch 28. Training loss: 0.000. Validation loss: 0.091
Epoch 29. Training loss: 0.000. Validation loss: 0.092
Epoch 30. Training loss: 0.000. Validation loss: 0.092
```

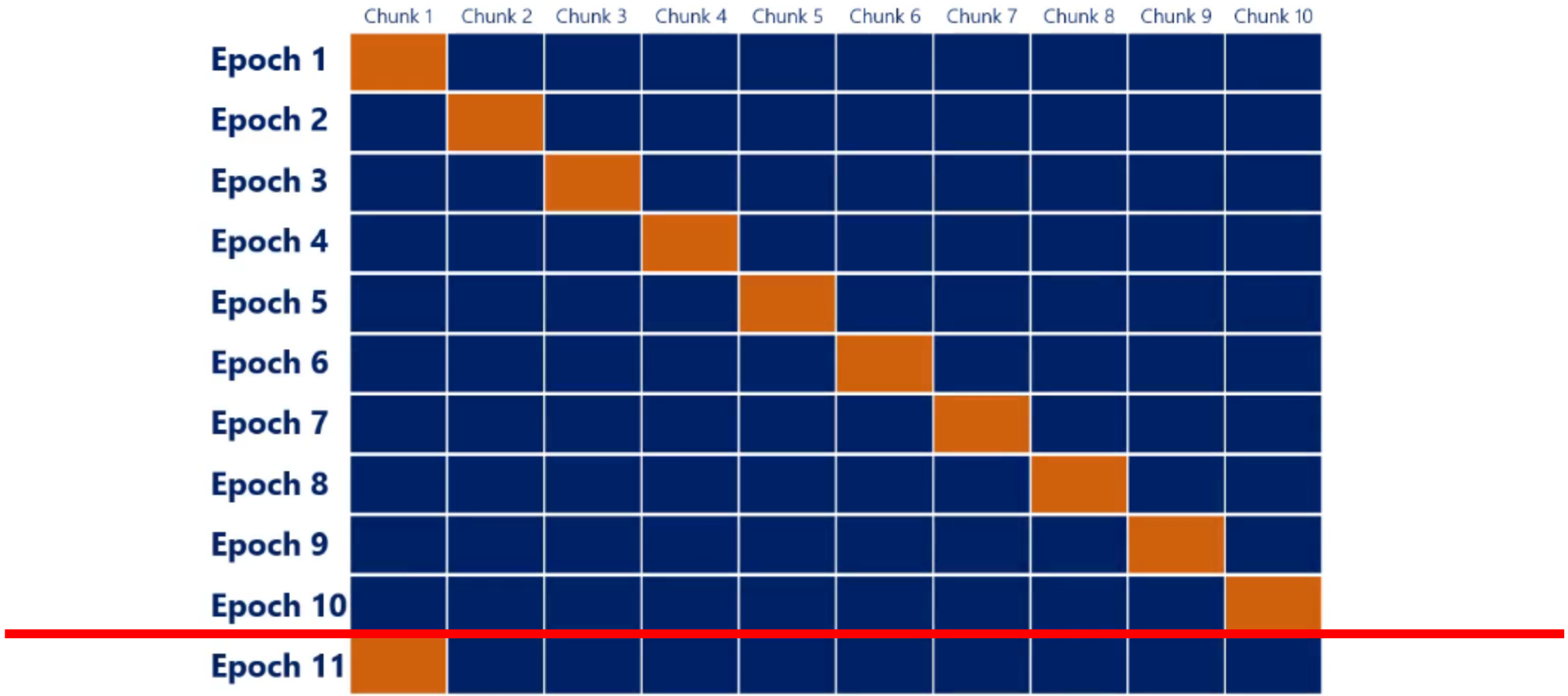



```
Epoch 1. Training loss: 0.341. Validation loss: 0.192
Epoch 2. Training loss: 0.164. Validation loss: 0.143
Epoch 3. Training loss: 0.115. Validation loss: 0.106
Epoch 4. Training loss: 0.086. Validation loss: 0.093
Epoch 5. Training loss: 0.068. Validation loss: 0.087
Epoch 6. Training loss: 0.054. Validation loss: 0.084
Epoch 7. Training loss: 0.042. Validation loss: 0.074
Epoch 8. Training loss: 0.033. Validation loss: 0.077
Epoch 9. Training loss: 0.029. Validation loss: 0.090
Epoch 10. Training loss: 0.022. Validation loss: 0.072
Epoch 11. Training loss: 0.018. Validation loss: 0.074
Epoch 12. Training loss: 0.014. Validation loss: 0.075
Epoch 13. Training loss: 0.011. Validation loss: 0.081
Epoch 14. Training loss: 0.009. Validation loss: 0.085
Epoch 15. Training loss: 0.007. Validation loss: 0.085
Epoch 16. Training loss: 0.009. Validation loss: 0.091
Epoch 17. Training loss: 0.008. Validation loss: 0.099
Epoch 18. Training loss: 0.006. Validation loss: 0.100
Epoch 19. Training loss: 0.005. Validation loss: 0.092
Epoch 20. Training loss: 0.002. Validation loss: 0.084
Epoch 21. Training loss: 0.002. Validation loss: 0.092
Epoch 22. Training loss: 0.016. Validation loss: 0.085
Epoch 23. Training loss: 0.005. Validation loss: 0.094
Epoch 24. Training loss: 0.001. Validation loss: 0.085
Epoch 25. Training loss: 0.001. Validation loss: 0.088
Epoch 26. Training loss: 0.000. Validation loss: 0.087
Epoch 27. Training loss: 0.000. Validation loss: 0.089
Epoch 28. Training loss: 0.000. Validation loss: 0.091
Epoch 29. Training loss: 0.000. Validation loss: 0.092
Epoch 30. Training loss: 0.000. Validation loss: 0.092
```

N-fold Cross Validation

- When the dataset is not large enough, it would not be possible to split it in three parts.
- To overcome this issue, we can train the model using **N-fold Cross Validation**

1. We split the dataset in N sets and use N-1 of them to train, while just 1 to validate
2. We repeat this for each epoch, constantly changing the validation set


Chunk 1
Chunk 2
Chunk 3
Chunk 4
Chunk 5
Chunk 6
Chunk 7
Chunk 8
Chunk 9
Chunk 10
Epoch 1
Epoch 2
Epoch 3
Epoch 4
Epoch 5
Epoch 6
Epoch 7
Epoch 8
Epoch 9
Epoch 10
Epoch 11
Validation
Training

N-fold Cross Validation

- Note that this would not necessarily remove any possibility of overfitting
  - In fact, overfitting may still be possible
- You want to keep a Test set to test the model even further
- However, this approach allows to build a relatively functional model even if the dataset is not large enough

Early Stopping and Initialization

Early Stopping

➢If we keep minimizing the **Loss function**, we end up overfitting

We need to stop the Training process before it overfits the model

We have seen code where we fixed the number of epochs

This is a naïve solution that does not work against complex cases

Early Stopping

- The best approach is to combine these 2 rules:
  - Stop when the **validation loss** starts increasing, or
  - Stop when the **training loss** becomes very small

Example:

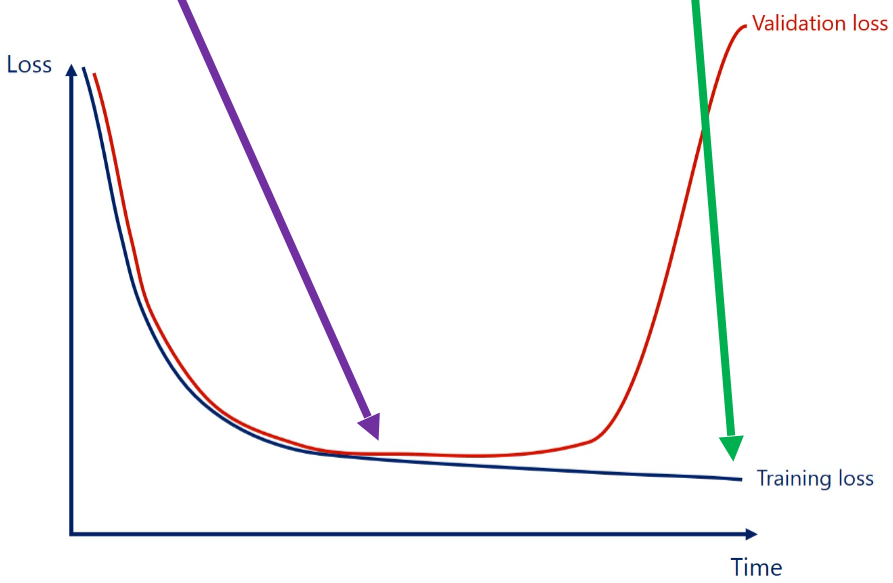
```
if (V_{i+1} > V_i) or (X_{i+1} − X_i < 0.001))
    stop()
```

Early Stopping

Example:

if $(V_{i+1} > V_i)$ or $(X_{i+1} - X_i < 0.001))$

stop()



V_i is the Validation loss at epoch i

X_i is the Training loss at epoch i

Initialization

- Initialization is the process in which we set the initial values of the weights

- An inappropriate initialization would cause an unoptimizable model

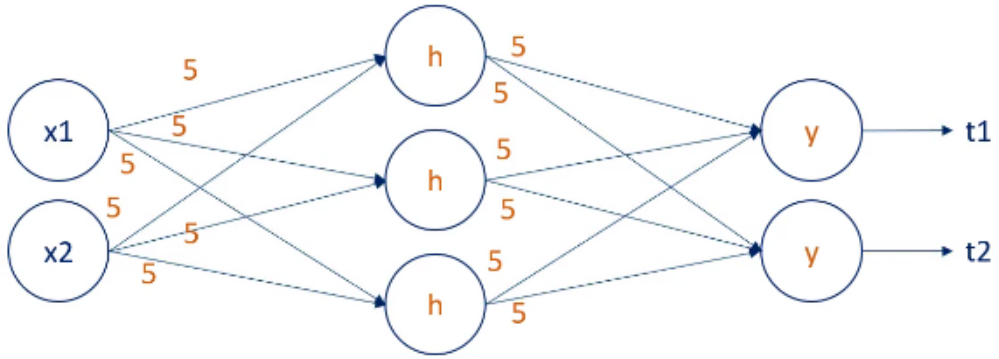
Initialization

❑First, the weights cannot be all equal

- If they are all equal, the **backpropagation** would not be able to recognize the different nodes and would upgrade all of them in the same way

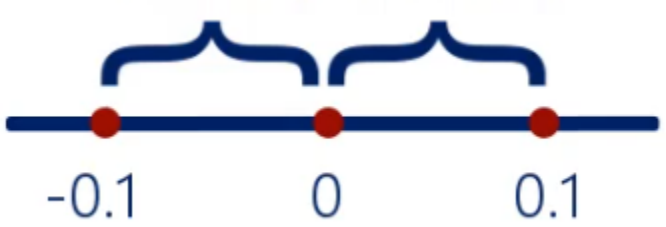
  Basically, making the weights useless

- Even the outputs would all be the same

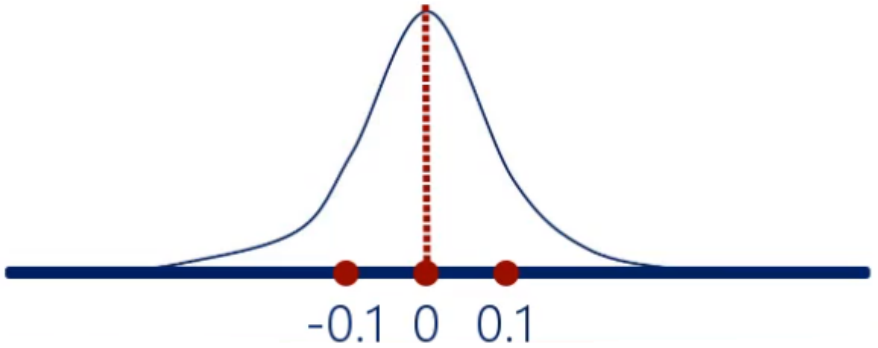


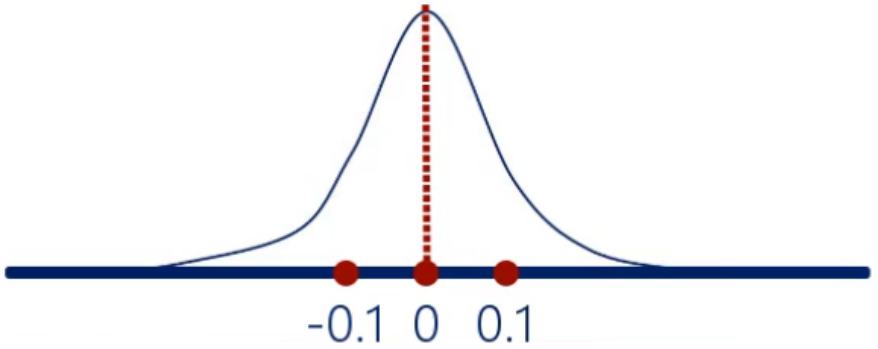
- In our code, we have chosen the weights generated from a uniform random function, with very small range

  Each value has the same probability to be chosen



- Another way would be to use a normal initializer (normal distribution)



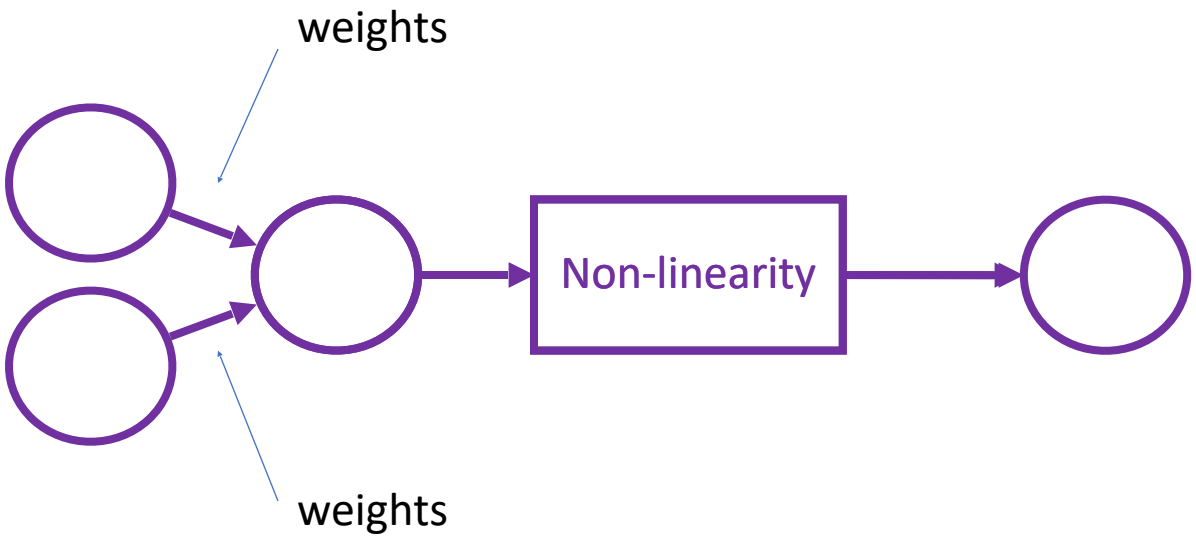


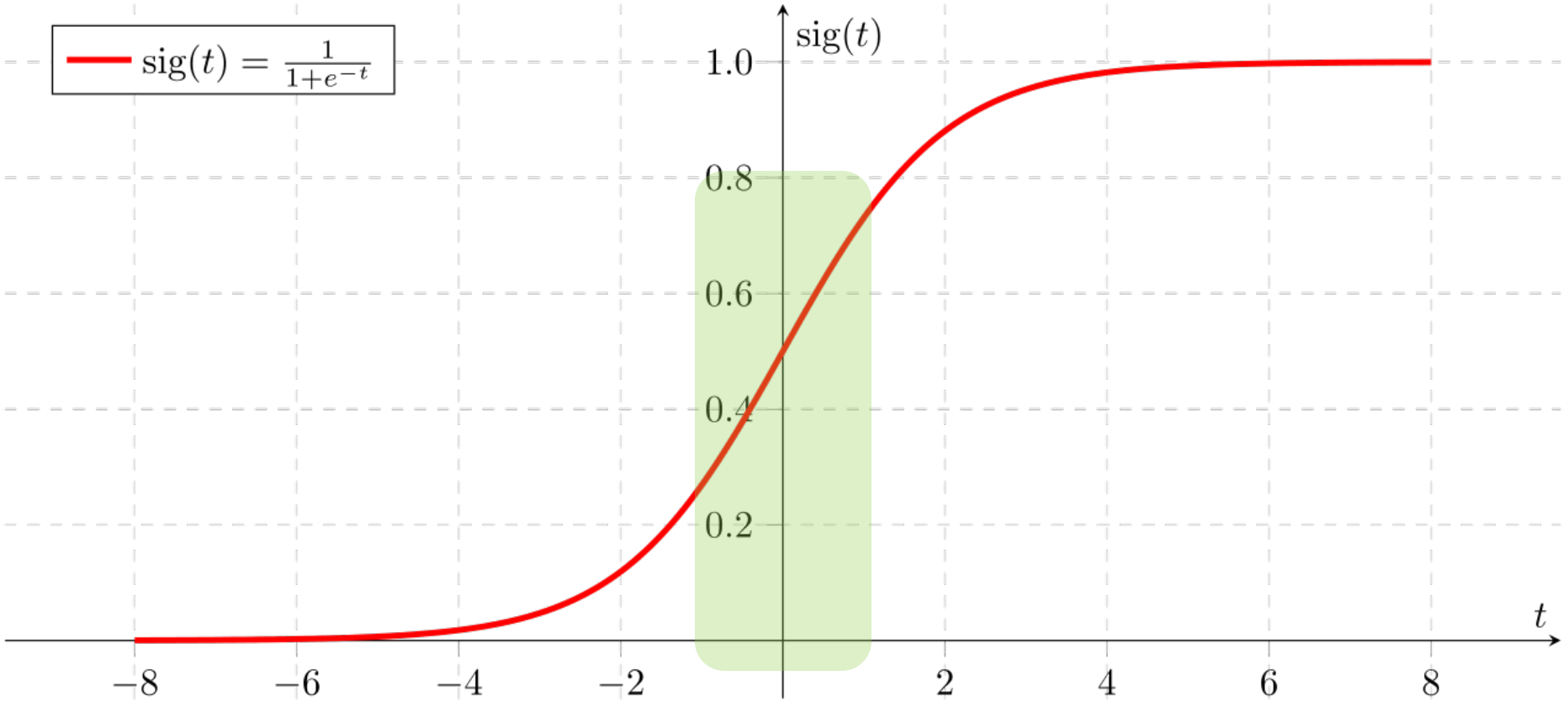
- In a normal distribution, the values closer to 0 have more probability to be chosen

  Note that we still count only the values between -0.1 and 0.1

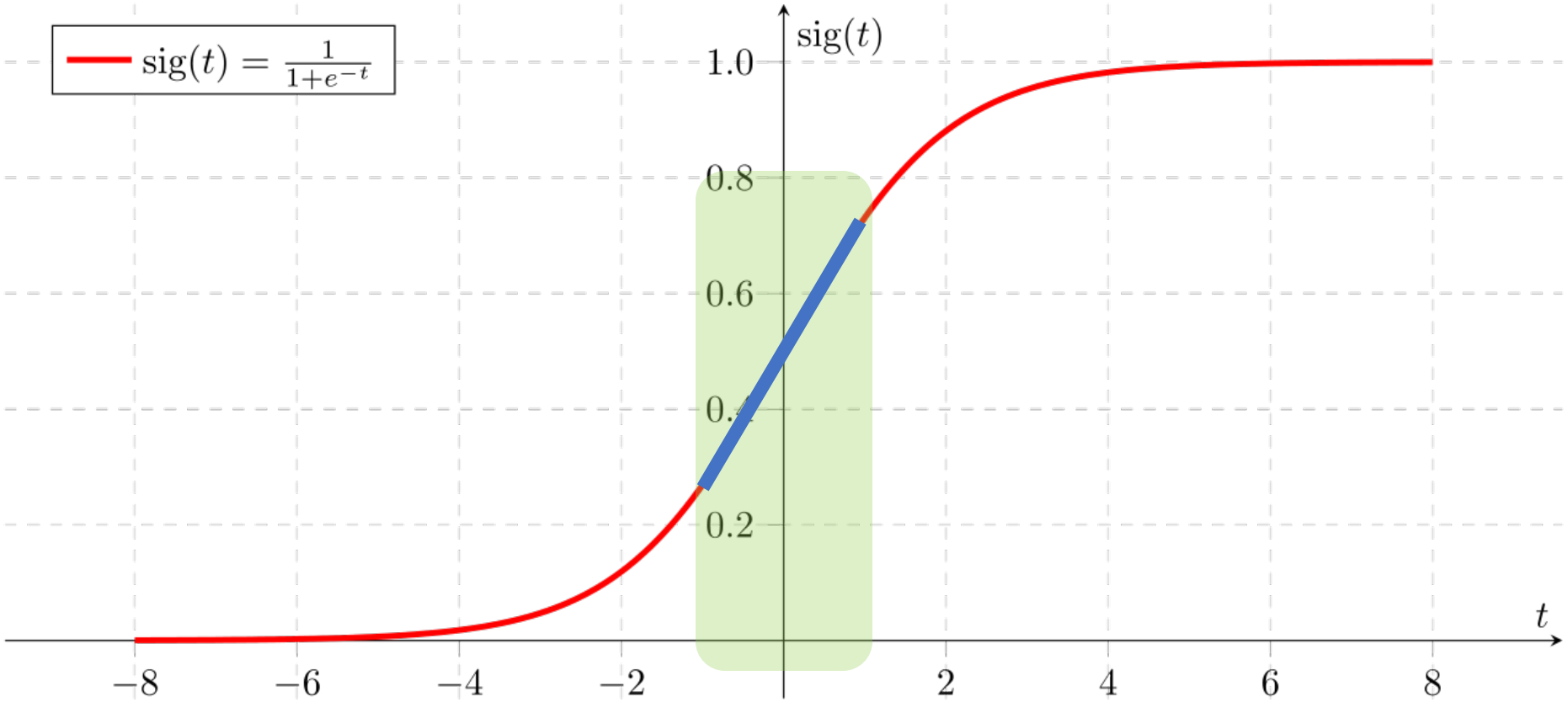
There is a problem, though…

- The initial values for the weights are used for the **Linear Combination**
- Then, the obtained results are “activated”…



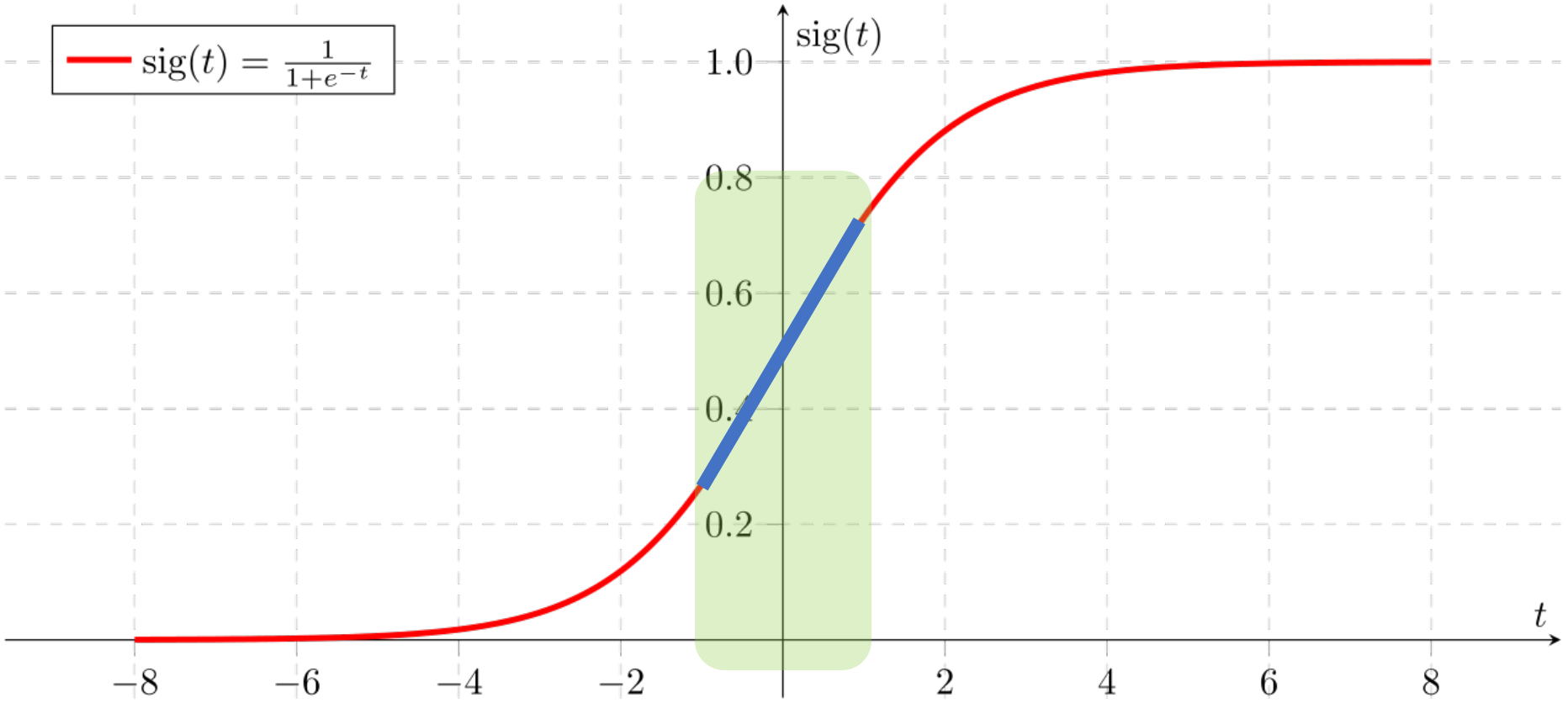


- Following both methods (normal and uniform distribution) and after being “activated”, the initial values of the weights would be enclosed in the area showed in green above



- That's bad, because, in that specific area, the sigmoid is almost linear!

  But we know that non-linearity is essential for DNN



- The weights have to be initialized in a reasonable range, so to have a good variance along the linear combinations

Xavier Glorot initialization

- Also known as just **Xavier/Glorot initialization**
- Splits in **Normal** and **Uniform Xavier initialization**
- This is *state-of-the-art* and the default initializer used by **TensorFlow**

Xavier Glorot initialization

Uniform Xavier initialization:

- The uniform range is equal to:

$$[-x, x] \text{ where } \mathrm{x} = \sqrt{\frac{6}{\#inputs + \#outputs}}$$

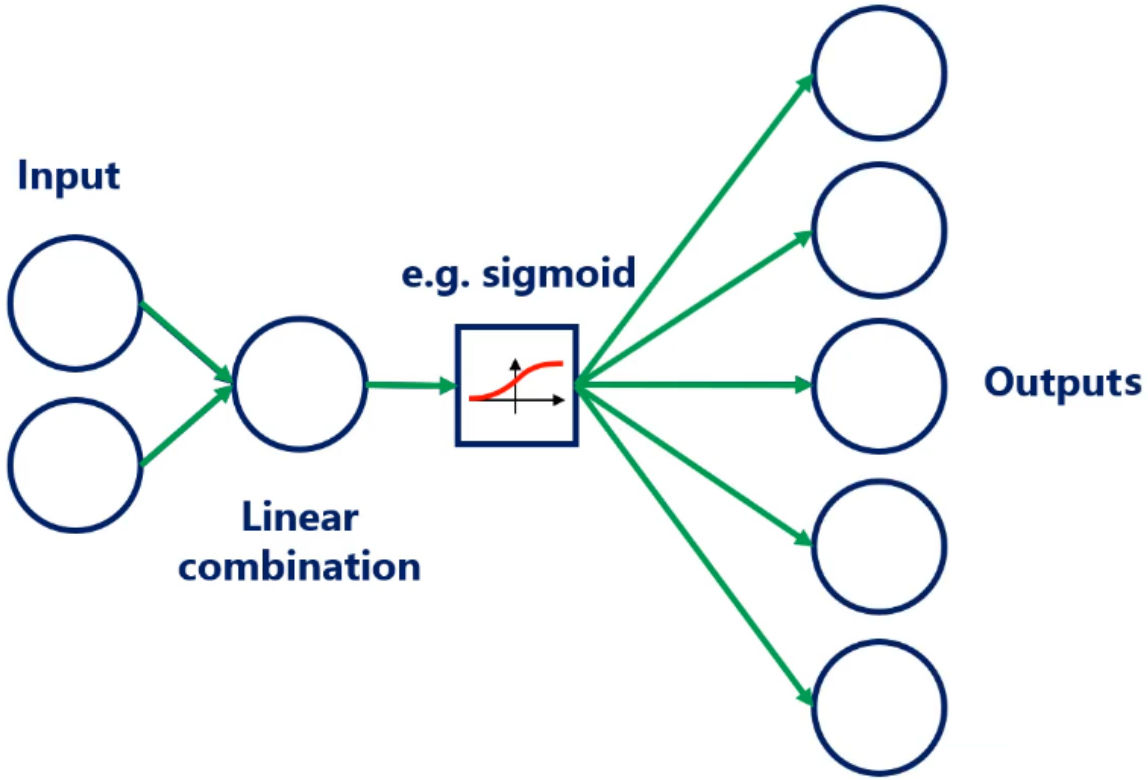
Normal Xavier initialization:

- Normal distribution with mean in 0 and standard deviation:

$$\sigma = \sqrt{\frac{2}{\#inputs + \#outputs}}$$

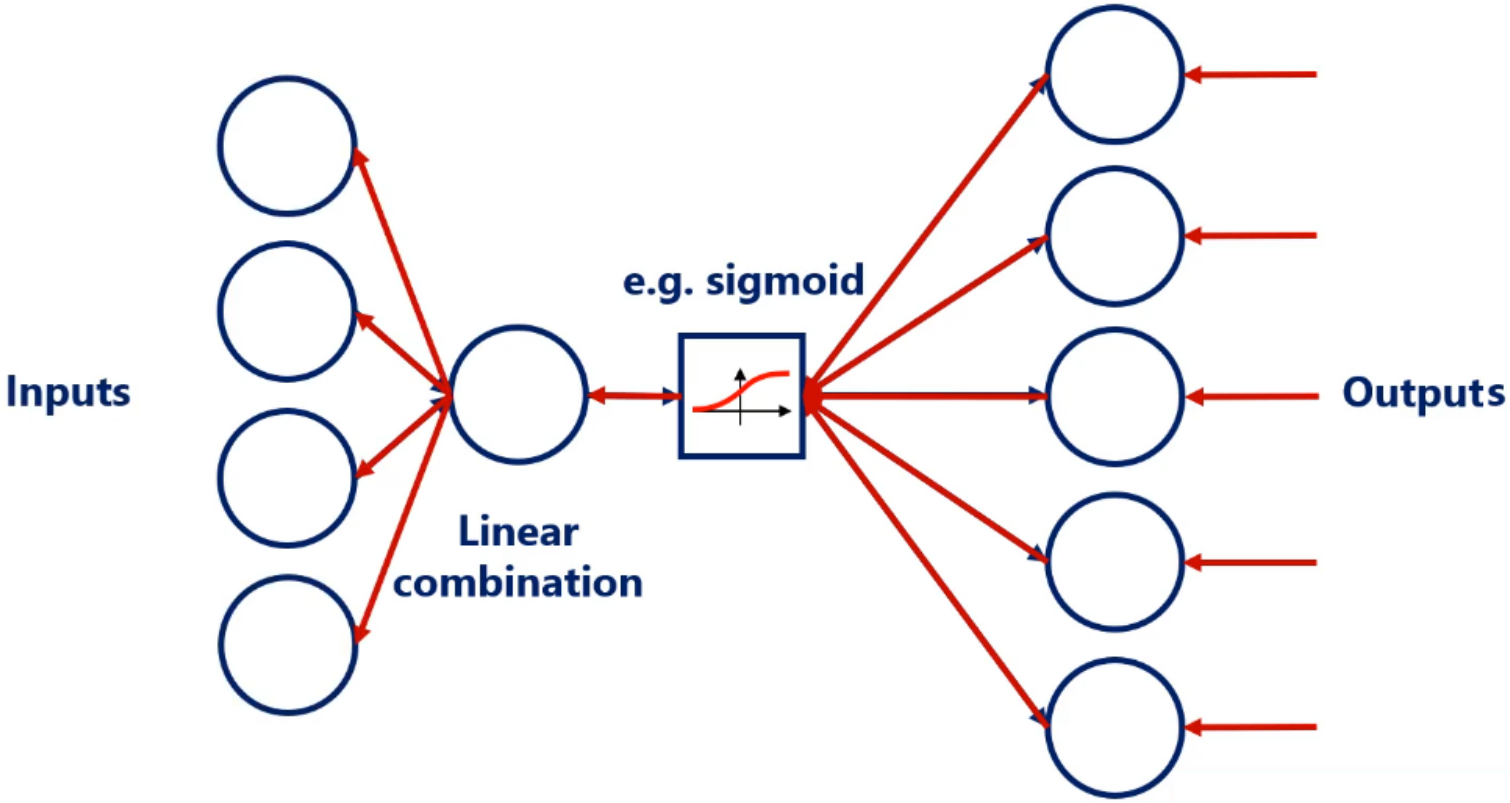
- Why do we focus so much on $\#inputs + \#outputs$?

The $\#outputs$ helps, because the more outputs we have, the more spread the data should be



- Why do we focus so much on $\#inputs + \#outputs$?

The $\#inputs$ helps too, because when we backpropagate we have the same problem but “vice versa”



Optimization

Which algorithms to use to obtain the best model?

Stochastic Gradient Descent

- The **Gradient Descent** that we introduced in the NumPy lecture has a couple of major drawbacks:
  - It iterates over the whole training set before updating the weights
  - Also, each update is very small

- However, we also saw that TensorFlow uses **Stochastic Gradient Descent**

  This should be a clue that SGD is better, because Google is always right!

Stochastic Gradient Descent

- The **SGD** works in the exact same way as the GD, but instead of updating the weights once per epoch **it updates the weights in real time inside a single epoch**

  Each of the batches per epoch would contribute to the upgrading

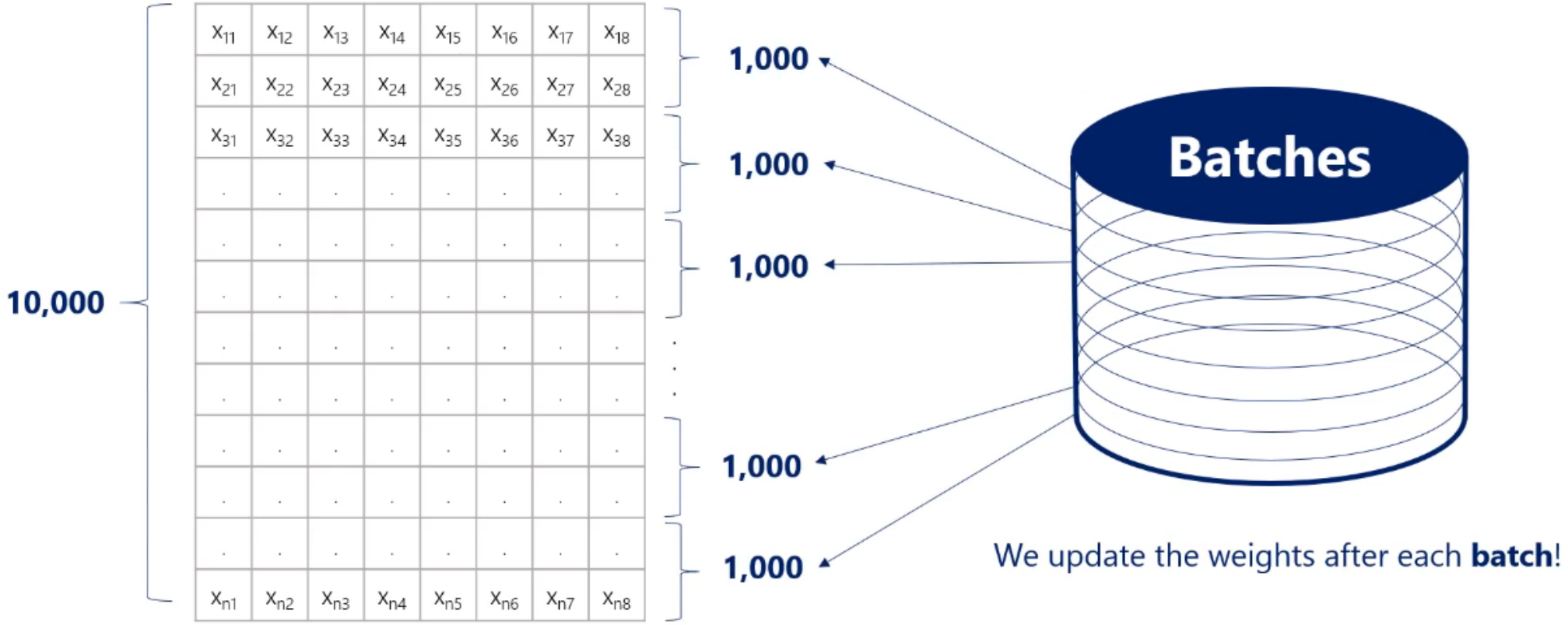
- The stochastic gradient descent is closely related to the concept of **batching**

  Batching is the process of splitting data into **n** batches (aka minibatches)

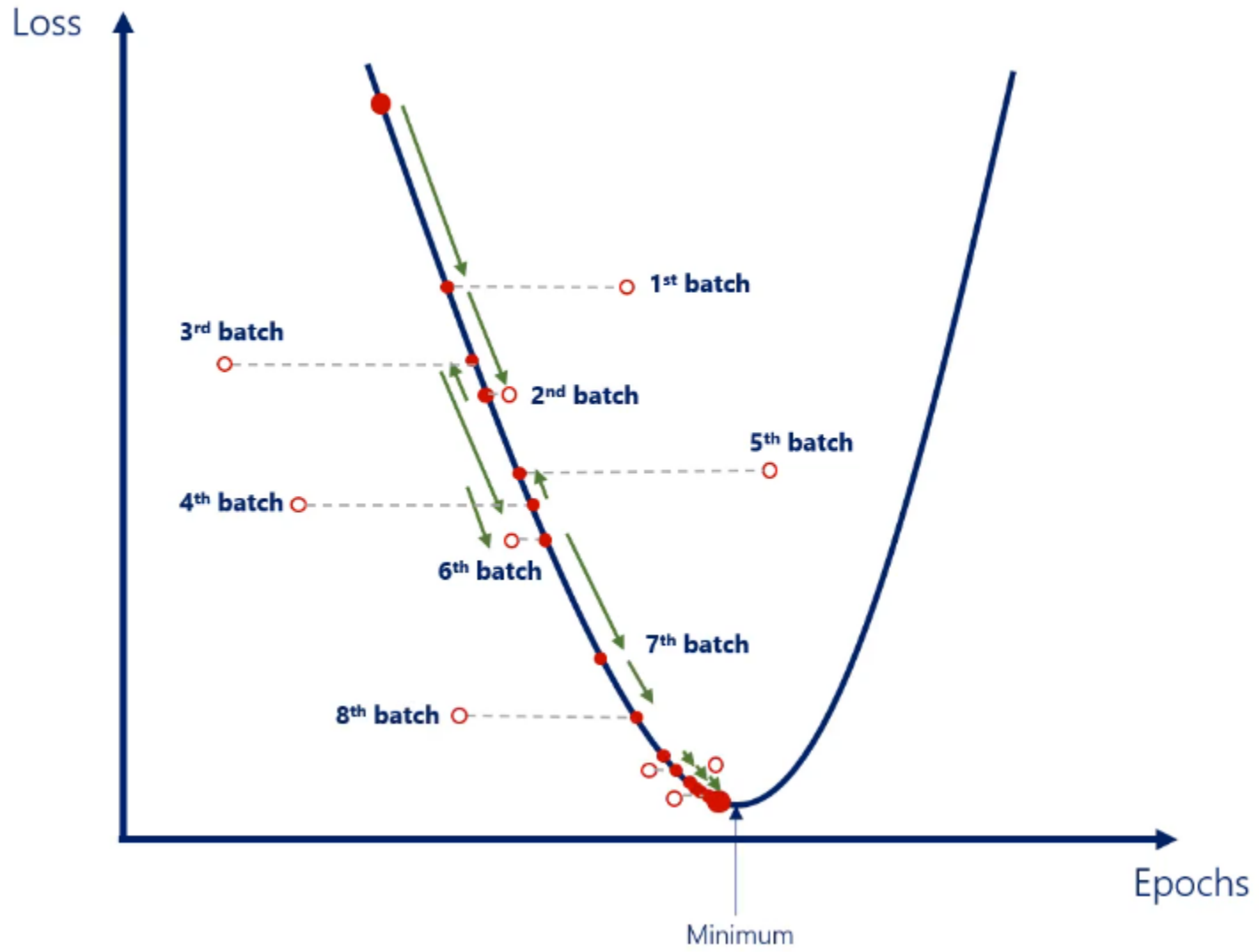
  ➢We update the weights after every batch instead of every epoch

Real SGD actually updates after every input, this is instead called **Mini-batch GD**

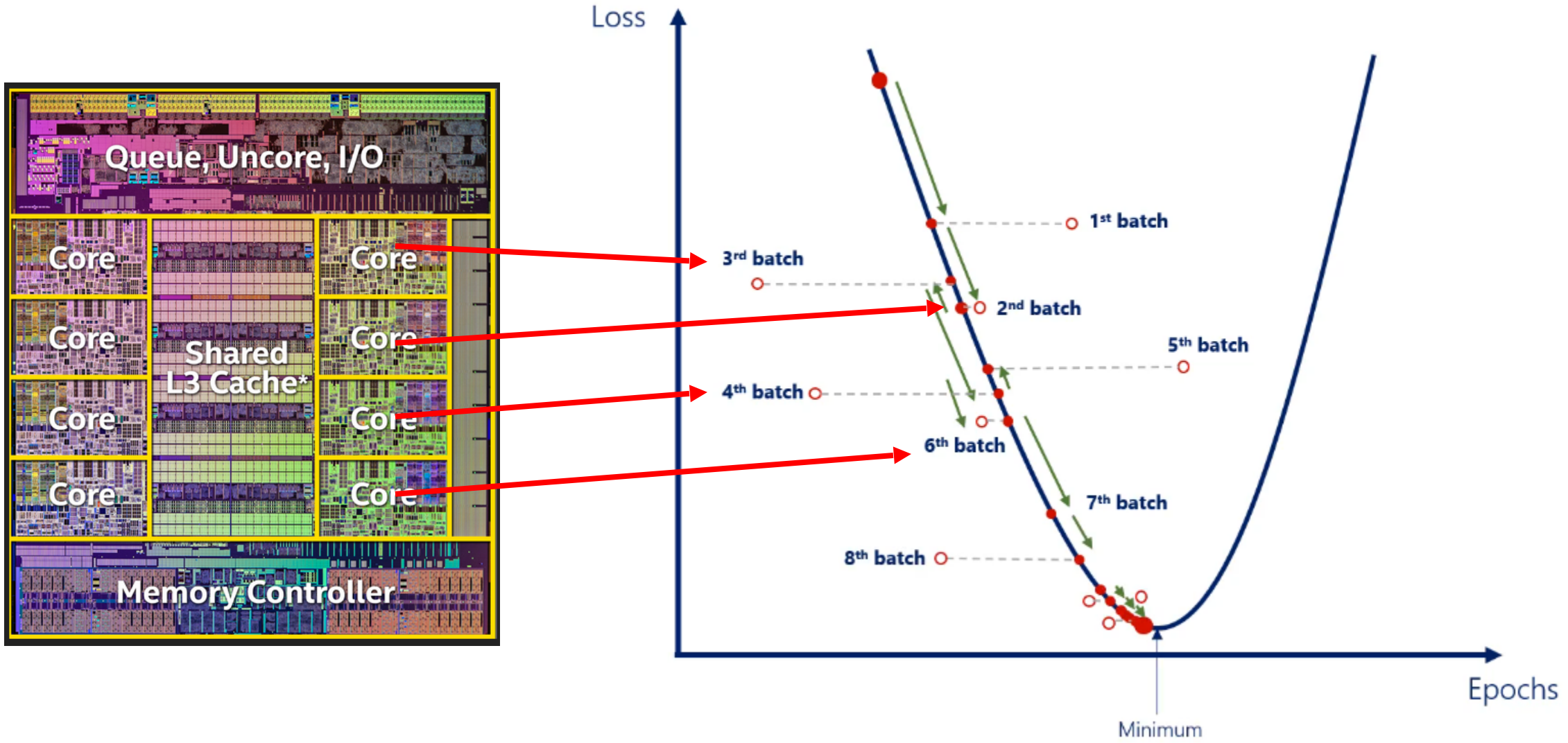
Mini-batch Gradient Descent



- Here we assume to have 10,000 training points and to split them in 10 batches
- So, we have 1,000 points per batch and 10 batches per epoch



- It is virtually the same algorithm as GD, but faster in getting closer to the min
- However, this speed would cost in accuracy, but overall is worth it



- Multicore architecture can work on separate batches

Local minimum and momentum

- We can reach a minimum, but the minimum may be a **local minimum** instead of the **global minimum**

- We need to improve our SGD adding **momentum**

  We had this formula to compute the new weight:

$$w_{i+1} = w_i - \eta \nabla_w L(y, t)$$

- We need to understand how fast we are reaching the minimum

  The faster, the higher the momentum

Local minimum and momentum

- We check how fast we were descending a moment ago

$$w_{i+1} = w_i - \eta \nabla_w L(y, t) - \eta \nabla_w L(y, t-1)$$

α

However, we add an α to adjust how much the previous Loss should count

This parameter is an **hyperparameter** and a usual value is $\alpha = 0.9$

$$w_{i+1} = w_i - \eta \nabla_w L(y, t) - \alpha \eta \nabla_w L(y, t-1)$$

Playing around with the Learning rate

- Another **hyperparameter** that we want to tune correctly is the learning rate η

It must be “small enough” and “big enough”. But how much?

- We can follow, what is called, a **learning rate schedule**

Playing around with the Learning rate

1. We start with a high learning rate
   - So, to reach the minimum faster
2. After a given number of epochs, we lower it
   - To avoid oscillation
3. When close to the last epoch, we lower it even further!
   - To obtain a more precise answer

Playing around with the Learning rate

Example:

1. For 5 epochs we choose $\eta = 0.1$
2. For the next 80 epochs, we choose $\eta = 0.01$
3. For the remaining epochs, we choose $\eta = 0.001$

However, this doesn't really work so well in real life

Playing around with the Learning rate

- A better approach is to have the **learning rate** increasing exponential at each epoch

1. We start with $\boldsymbol{\eta_0 = 0.1}$ in the first epoch
2. We increment $\boldsymbol{\eta}$ at each epoch following the formula:

$$\eta = \eta_0 e^{-\frac{n}{c}}$$

Where $\boldsymbol{n}$ is the current epoch and $\boldsymbol{c}$ is a constant

Playing around with the Learning rate

$$\eta = \eta_0 e^{-\frac{n}{c}}$$

- A common constant value is $c = 20$

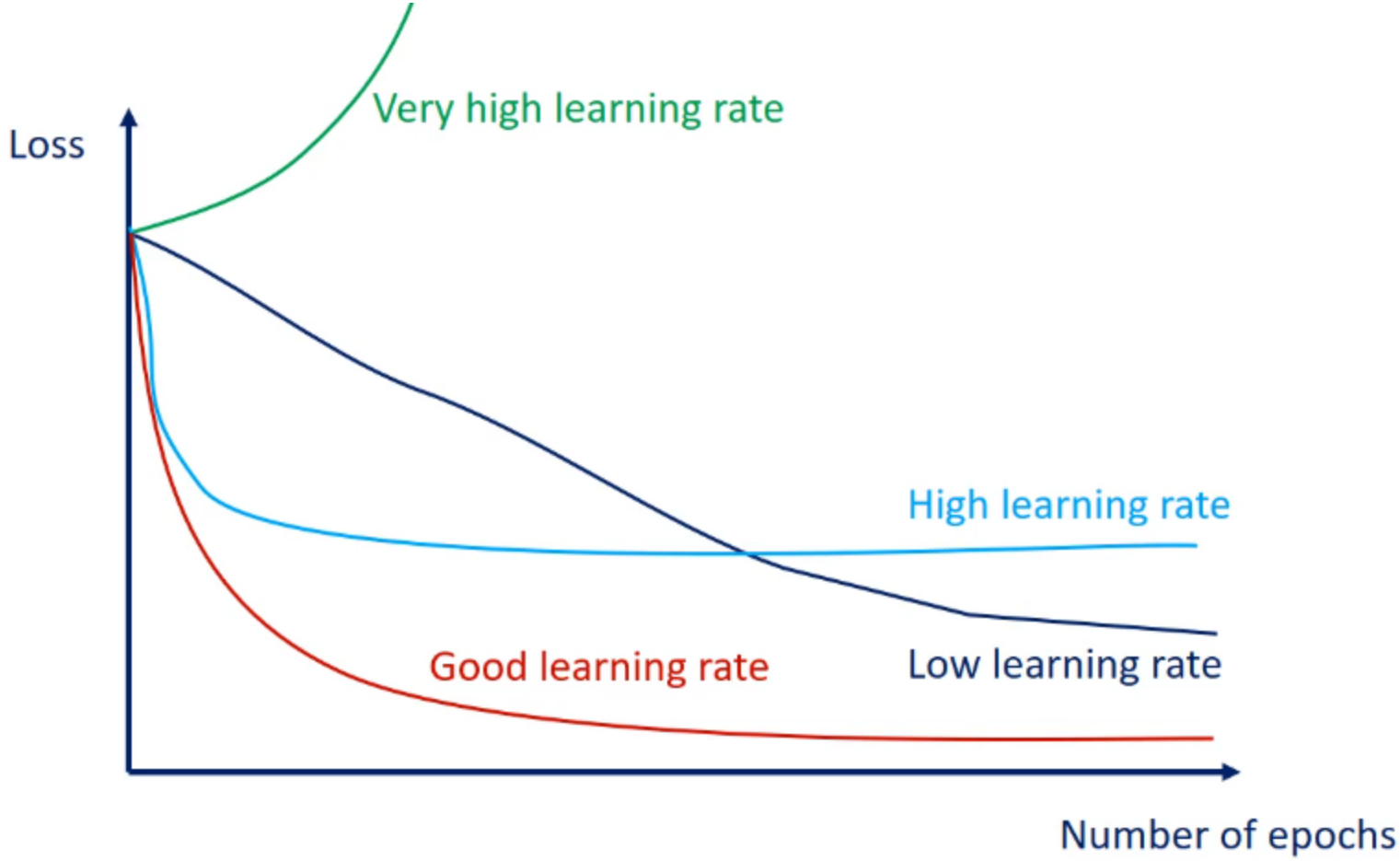
  However, there is a rule of thumb for which:

  100 epochs $\rightarrow 50 < c < 500$

  1000 epochs $\rightarrow 500 < c < 5000$

- This value is another **hyperparameter**.

  You need to tune it, but usually is not very influential. Nonetheless, every problem is different…



➢ We want to obtain <u>a curve that looks like the red one</u> in the picture

- We can plot the **Loss function** to see if it is close to that curve

  Note that, eventually, the Low learning rate (dark blue) would converge too, but much slower than the Good one (red)

Hyperparameters

Let’s recap the hyperparameters:

- Width
- Depth
- Learning rate (η)
- Batch size
- Momentum coefficient (α)
- Decay coefficient (c)

Adaptive Learning rate schedule

We will discuss 3 Adaptive Learning rate schedule:

- **AdaGrad** (Adaptive Gradient algorithm)
- **RMSProp** (Root Mean Square Propagation)
- **ADAM** (Adaptive Moment estimation)

AdaGrad

- AdaGrad dynamically varies the learning rate at each update and for every weight individually

  The original rule was:

$$w_{i+1} = w_i - \eta \nabla_w L(y, t)$$

  That is:

$$w(t+1) - w(t) = -\eta \frac{\partial L}{\partial w}(t)$$

$$\Delta \boldsymbol{w} = -\boldsymbol{\eta} \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}}(\boldsymbol{t})$$

  The change is equal to the partial derivative of the Loss with respect to w

AdaGrad

$$\Delta \boldsymbol{w} = -\eta \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w}}(t)$$

- The **Adaptive Gradient** expression is in a way similar:

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \in} * \frac{\partial L}{\partial w}(t)$$

The index **i** indicates that the update rule is individual for each weight

AdaGrad

$$\Delta \boldsymbol{w}_i(t) = -\frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{G}_i(t)} + \in} * \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w}}(t)$$

Where $G_i(t)$ is:

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

- Note that $G_i(t)$ is a **monotonous increasing function**

  It would constantly increase

AdaGrad

This is **monotonous decreasing**

$$\Delta \boldsymbol{w}_i(t) = -\frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{G}_i(t)} + \in} * \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w}}(t)$$

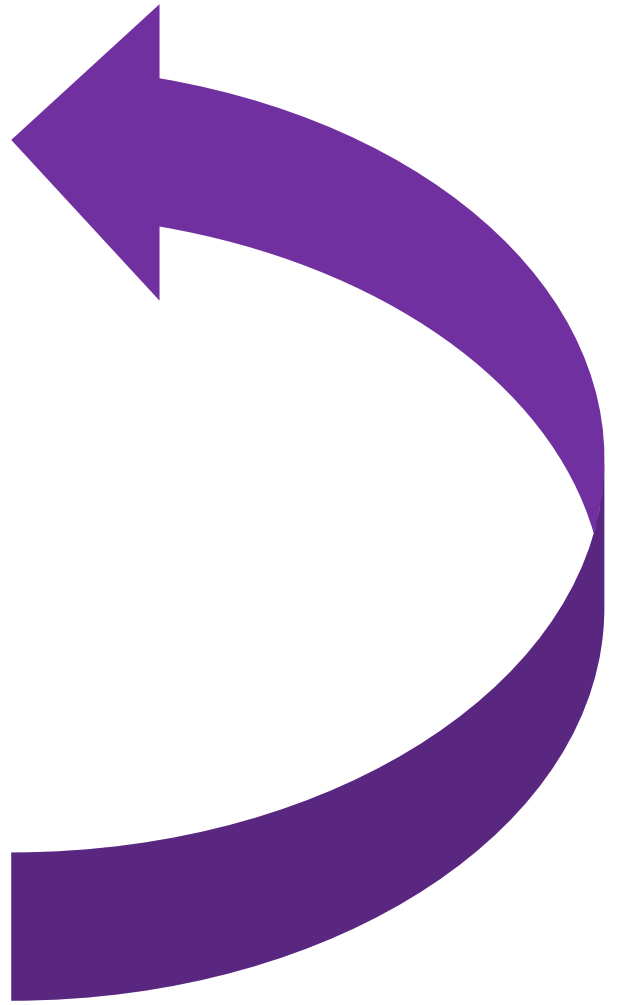
Where $G_i(t)$ is:

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

- Note that $G_i(t)$ is a **monotonous increasing function**

  It would constantly increase



AdaGrad

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \in} * \frac{\partial L}{\partial w}(t)$$

The value ϵ is a small number that avoid a division by 0 if $G_i(t)$ is zero

- This learning rate scheduler is adaptive because is based on the training itself
  - $G_i(t)$ depends on the Loss

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

AdaGrad

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \in}} * \frac{\partial L}{\partial w}(t)$$

- Also, every individual weight in the whole network keeps track of its own function to normalize its own steps

  The adaptation is per weight

- It's an important observation because different weights do not reach their optimal values simultaneously

RMSProp

$$\Delta \boldsymbol{w}_i(t) = -\frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{G}_i(t)} + \in} * \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w}}(t)$$

- **Root Mean Square Propagation** has the same update rule as Adaptive Gradient algorithm. However, $G_i(t)$ is different

$$G_i(t) = \beta G_i(t-1) + \left(1-\beta\right)\left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

with beginning point $G_i(0) = 0$

RMSProp

$$\Delta \boldsymbol{w}_i(t) = -\frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{G}_i(t)} + \in} * \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w}}(t)$$

$$G_i(t) = \beta G_i(t-1) + \left(1-\beta\right)\left(\frac{\partial L}{\partial w_i}(t)\right)^2$$

- The (other) hyperparameter $\boldsymbol{\beta}$ is a value in between 0 and 1, that usually is around ~0.9
- This makes $G_i(t)$ monotonously decreasing and, so, $\boldsymbol{\eta}$ monotonously increasing

  Empirical evidence shows that in this way the rate adapts much more efficiently

ADAM

- With **Adaptive Moment estimation** we combine the **learning rate schedules** and the **momentum**

  Proposed in 2015, became the most advanced optimizer, being **very fast and efficient**

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \in}} * \frac{\partial L}{\partial w}(t)$$

$$w_{i+1} = w_i - \eta \nabla_w L(y, t) - \alpha \eta \nabla_w L(y, t-1)$$

ADAM

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t)} + \in} * \frac{\partial L}{\partial w}(t)$$

$$w_{i+1} = w_i - \eta \nabla_w L(y, t) - \alpha \eta \nabla_w L(y, t-1)$$

$$\Delta \boldsymbol{w_i}(\boldsymbol{t}) = -\frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{G_i}(\boldsymbol{t})} + \boldsymbol{\epsilon}} * \boldsymbol{M_i}(\boldsymbol{t})$$

$$M_i(t) = \alpha M_i(t-1) + (1-\alpha)\frac{\partial L}{\partial w_i}(t)$$

$$M_i(0) = 0$$

Preprocessing

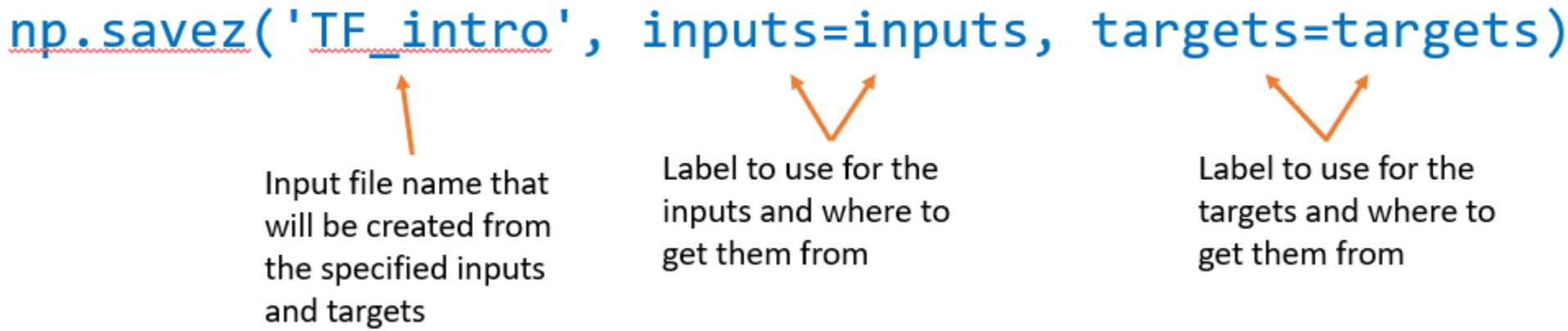
Preprocessing

- Preprocessing refers to any manipulation we apply to the dataset before running it through the model
  - Up to now, we have assumed that we had already preprocessed our data in a way suitable for training

Preprocessing

- We have actually seen a basic preprocessing when we converted our dataset in an **.npz** file for TensorFlow

```
np.savez('TF_intro', inputs=inputs, targets=targets)
```



- However, this is basically just reordering the information

Preprocessing

- More broadly, we consider preprocessing as actual **data transformation**

Let’s see some preprocessing techniques

Feature Scaling

- **Feature Scaling** (aka **Standardization**) is the process of transforming the data we are working with into a standard scale

- A classic approach is computing, per each value, a standardized variable:

$$\frac{x - \mu}{\sigma}$$

- The goal is to obtain a distribution with mean (μ) equal to 0, and standard deviation (σ) equal to 1

Feature Scaling

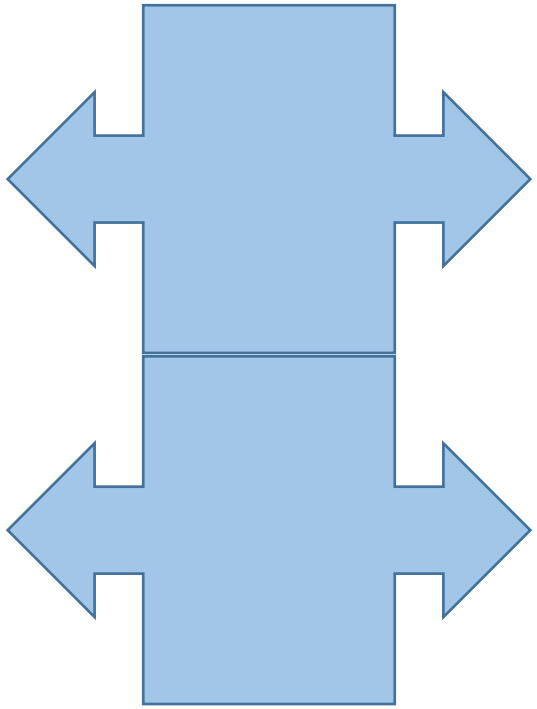
$$\frac{x - \mu}{\sigma}$$

Example:

Data: (1.3, 1.34, 1.25)

Mean: 1.296

Std: 0.045



(0.07, 0.96, -1.03)

Data: (110k, 98.7k, 135k)

Mean: 114,566.6

Std: 18575.88

(-0.25, -0.85, 1.1)

Feature Scaling

- Standardizing makes it easier to compare scores
  - In particular when those scores were measured on different scales

Principal Components Analysis

Another preprocessing method is **Principal Components Analysis** (**PCA**)

- It is a dimensioned reduction technique used when working with several variables referring to the same bigger concept or latent variable

- There are more, and they are problem specific

Preprocessing for Classification

There are two major techniques to preprocess data meant for classification problems (e.g., images)

- **One-hot encoding**

  Vectors where only one bit is set, and each vector corresponds to a sample

  Ex: (001, 010, 100) for three objects

  - This is a very good approach, because the samples are uncorrelated and unequivocal

- **Binary encoding**

  Encoding using the minimum number of bits to index all the samples

  Ex: (01, 10, 11) for three objects

  - This is not a perfect approach, because it could create dependences and unjustified correlations among the samples (order, for example)

Preprocessing for Classification

One-hot encoding introduces a problem, though.

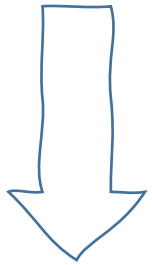
- If we have thousands of different types of samples, we would have vectors with thousands of elements!

- Instead, with **binary encoding** they would contain just a bunch of bits (say, 16)

➢Thus, it is better to use **binary encoding** when the categories of samples are a lot!

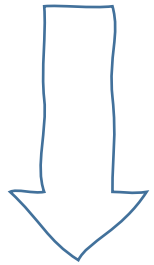
Preprocessing for Classification

ONE-HOT



Few categories

BINARY



Many categories

“Hello World”

MNIST example

MNIST

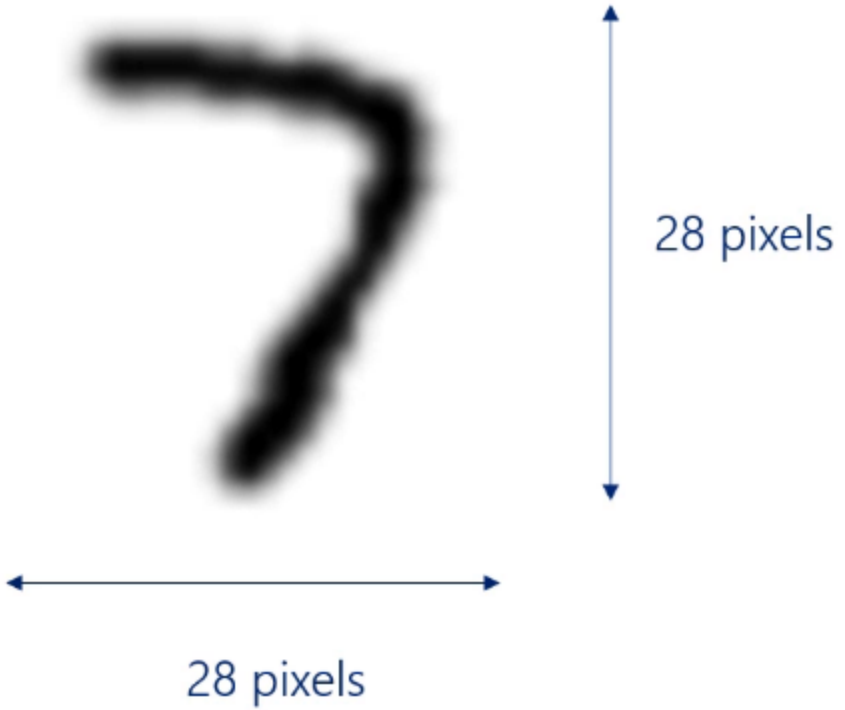
- The MNIST dataset consists of around 70,000 images of handwritten digits
- Since we have 10 digits, there are 10 classes from 0 to 9
- Our objective is to build an algorithm that takes as input an image and then correctly determines which number is shown in that image
- It is considered the “Hello World” of ML

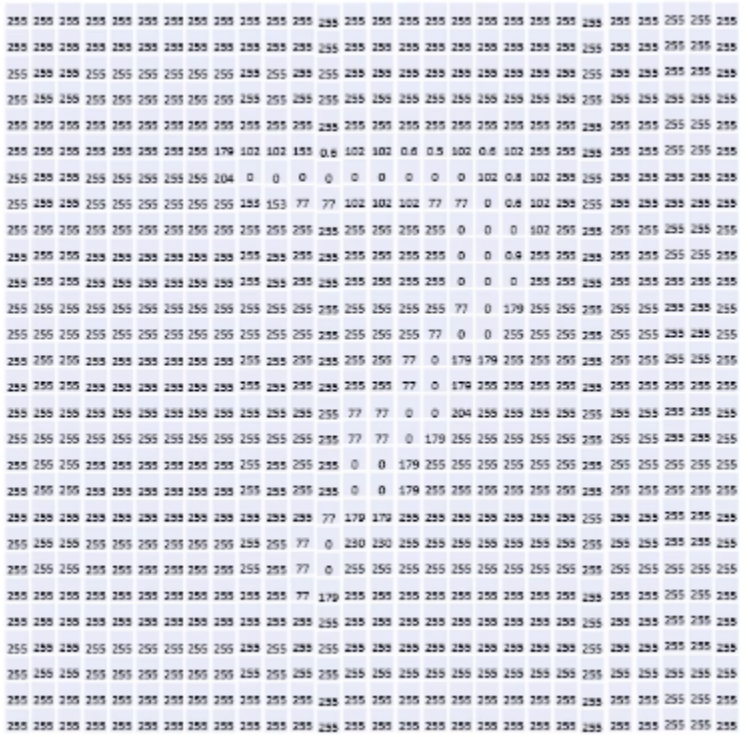
MNIST

- Here a sample of the images in the MNIST dataset:

MNIST

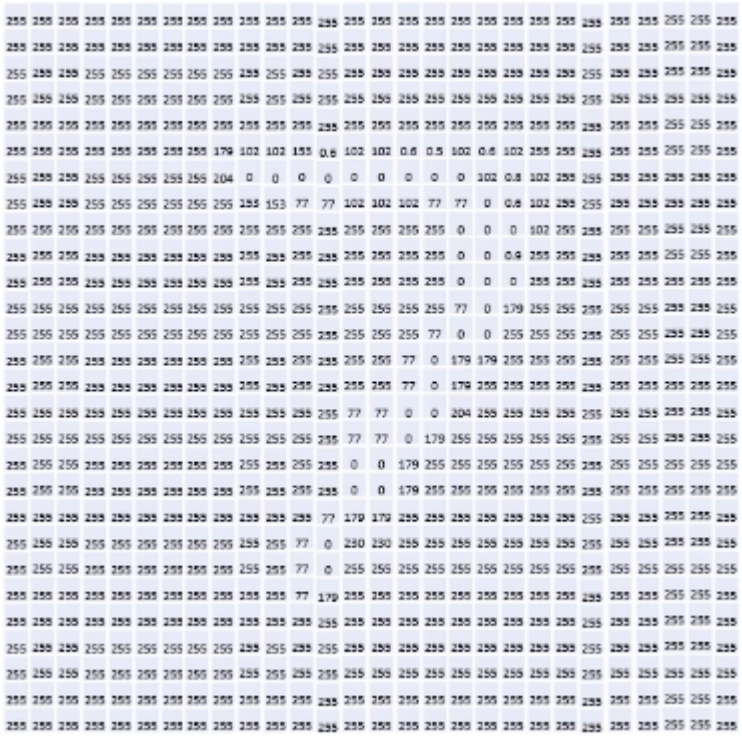
- Each photo looks like this, and they are 28x28 pixels
- Each pixel is a value from 0 to 255 indicating the greyscale
  0 for purely black and 255 for purely white

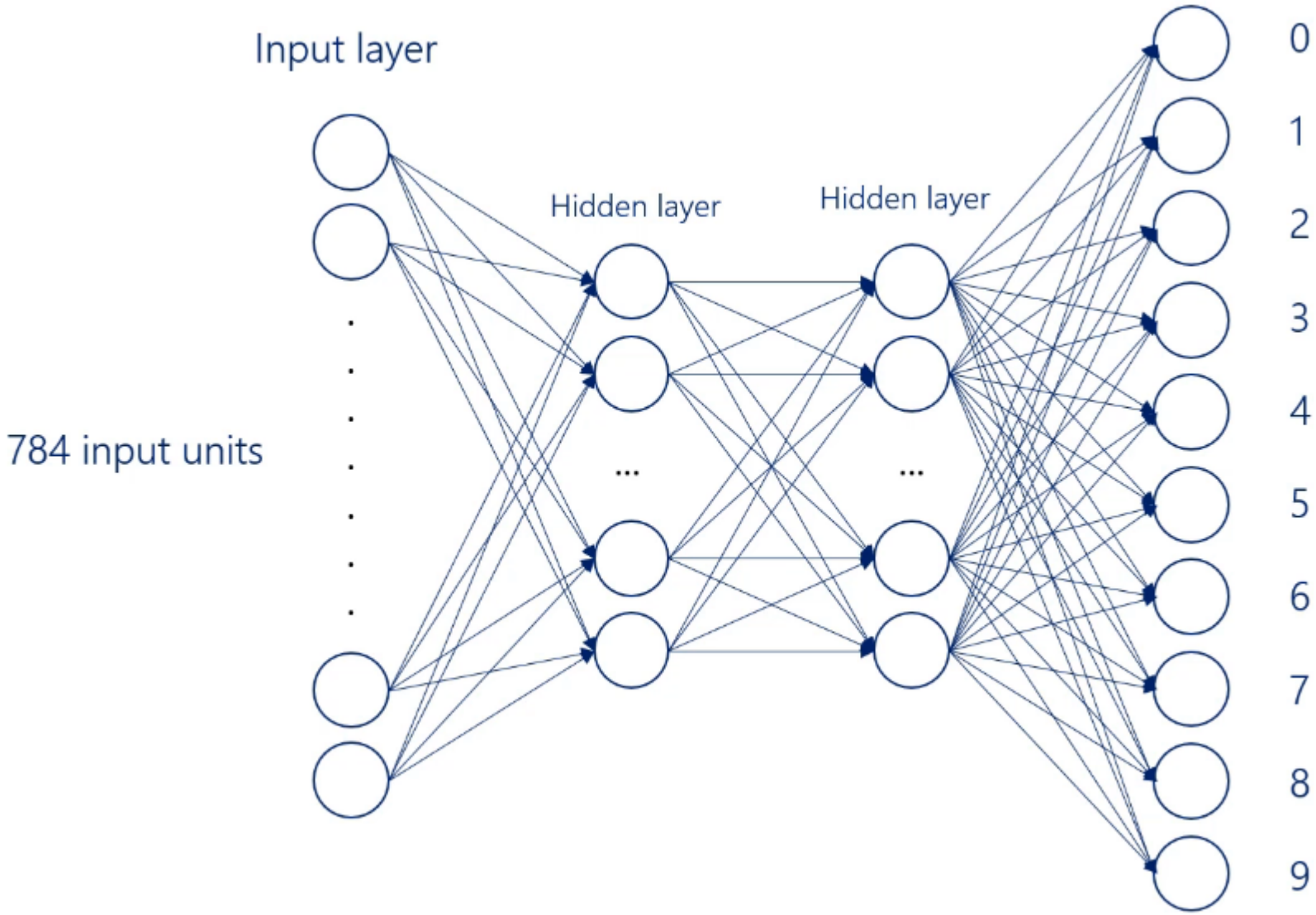




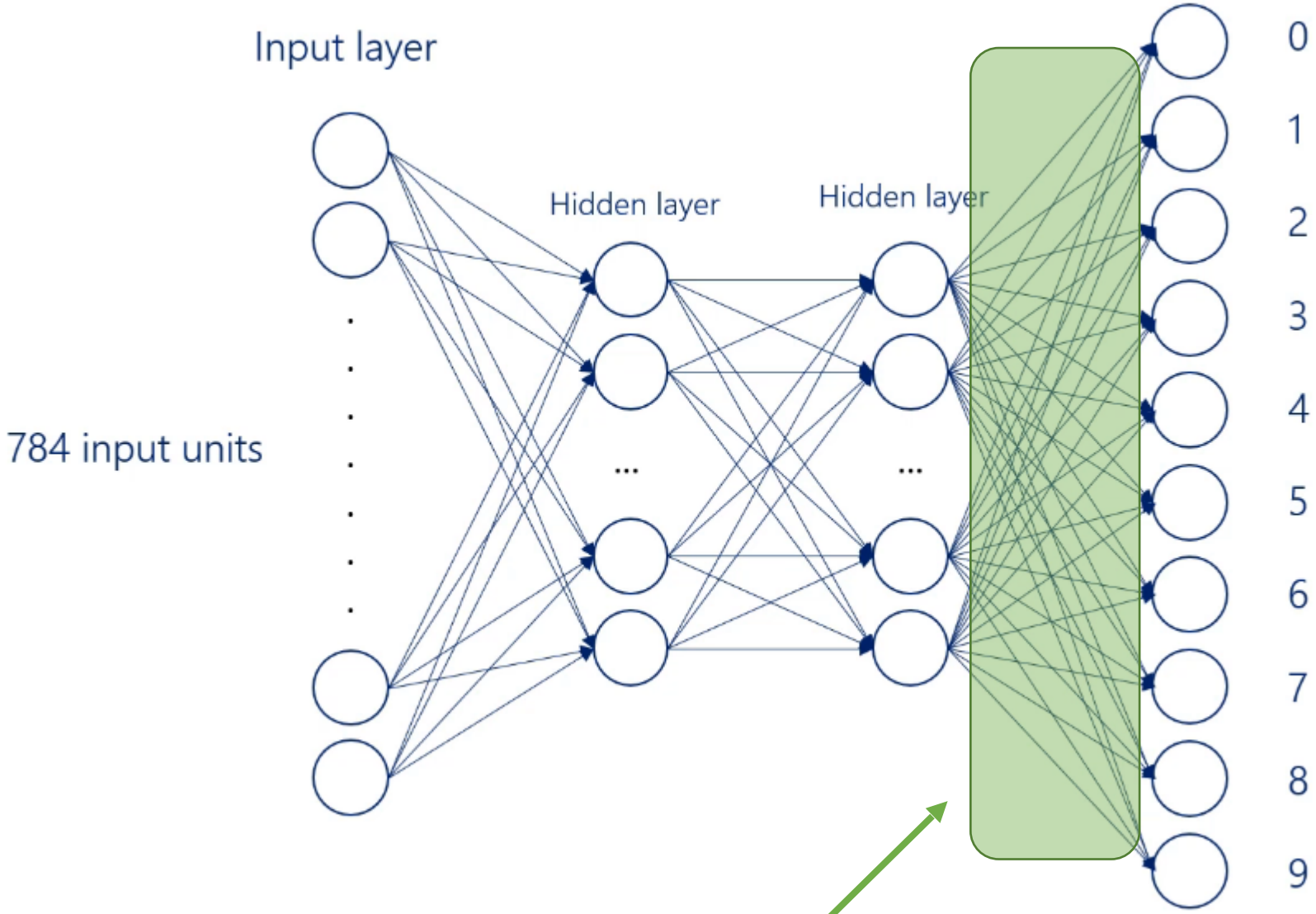
MNIST

- Reading the image by columns, we have a vector of 784x1 values (28x28)





Note that the output nodes would form a hot-vector



Since we need the probability of a digit being correctly labeled, we use a **Softmax** activation function for the output layer

What to do:

1. We must prepare our data and preprocess it

   We will create **training**, **validation** and **test datasets** as well as select the batch size

2. We must outline the model and choose the activation functions we want to employ

3. We must set the appropriate advanced optimizers and the loss function

4. We will make it learn (back propagating)

   At each epoch we will validate

5. We will test the accuracy of the model against the test dataset

Import the packages

- TensorFlow includes a data provider for MNIST

  It comes with the `tensorflow-datasets` module, therefore, please install the package using one of these two commands (Anaconda prompt):

  ```
  pip install tensorflow-datasets
  conda install tensorflow-datasets
  ```

- Also, run this command:

  ```
  pip install ipywidgets
  ```

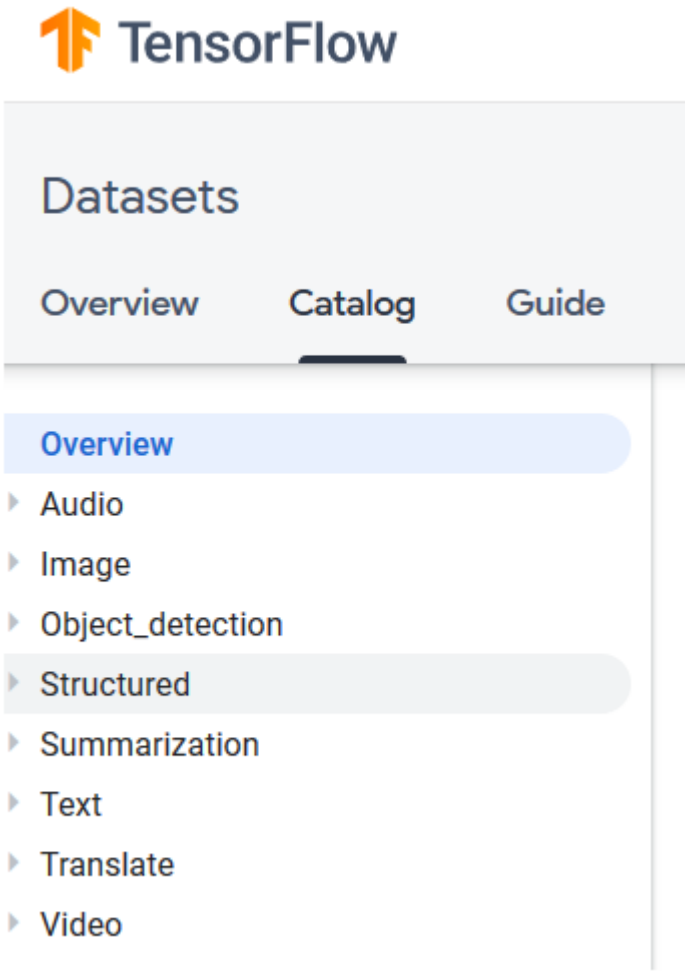
Import the packages

```
import numpy as np
import tensorflow as tf

import tensorflow_datasets as tfds
```

Loading and preprocessing of the data

- More datasets can be found here:

https://www.tensorflow.org/datasets/catalog/overview



Loading and preprocessing of the data

`tfds.load` loads a dataset (or downloads and then loads if that's the first time you use it)

```
mnist_dataset, mnist_info = tfds.load(name='mnist', with_info=True, as_supervised=True)
```

`name='mnist'` is the dataset to load/download

`with_info=True` will also provide us with a tuple containing information about the version, features, number of samples

We will use this information a bit below and we will store it in `mnist_info`

`as_supervised=True` will load the dataset in a 2-tuple structure (input, target)

Alternatively, `as_supervised=False`, would return a dictionary
But obviously, we prefer to have our inputs and targets separated

Loading and preprocessing of the data

```
mnist_dataset, mnist_info = tfds.load(name='mnist', with_info=True, as_supervised=True)
```

- The `with_info` parameter set to `True` will store the additional info in the variable `mnist_info`

The code without the parameter would look like this:

```
mnist_dataset = tfds.load(name='mnist', as_supervised=True)
```

Training, Validation and Testing set

- Once we have loaded the dataset, we can easily extract the training and testing dataset with the built references

```
mnist_train, mnist_test = mnist_dataset['train'], mnist_dataset['test']
```

- Note that there is no Validation set, in fact, TensorFlow datasets have usually only the Training and Test sets

Split	Examples
'test'	10,000
'train'	60,000

Training, Validation and Testing set

- We start by defining the number of validation samples as a % of the train samples

  This is also where we make use of `mnist_info`

  We don't have to count the observations ourselves!

```
num_validation_samples = 0.1 * mnist_info.splits['train'].num_examples
```

Let's cast this number to an integer, because as a float it could cause an error along the way

```
num_validation_samples = tf.cast(num_validation_samples, tf.int64)
```

Training, Validation and Testing set

- Let's also store the number of test samples in a dedicated variable instead of using the `mnist_info` one

```
num_test_samples = mnist_info.splits['test'].num_examples
num_test_samples = tf.cast(num_test_samples, tf.int64)
```

Training, Validation and Testing set

Preprocessing:

Normally, we would like to scale our data in some way to make the result more numerically stable

- In this case we will simply prefer to have inputs between 0 and 1

Training, Validation and Testing set

Preprocessing:

- Let's define a function called: `scale`, that will take an MNIST image and its label

```
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255.
    return image, label
```

We make sure the value is a float, and since the possible values for the inputs are 0 to 255 (256 different shades of grey), we divide each element by 255, so we would get the desired result -> all elements will be between 0 and 1

Training, Validation and Testing set

- The method `.map()` allows us to apply a custom transformation to a given dataset
- We have already decided that we will get the validation data from `mnist_train`, so, finally, we scale and batch the test data too

  We scale it so it has the same magnitude as the train and validation

  There is no need to shuffle the test data, because we won't be training on it
  - There would be a single batch, equal to the size of the test data

```
scaled_train_and_validation_data = mnist_train.map(scale)
test_data = mnist_test.map(scale)
```

```
import numpy as np
import tensorflow as tf

import tensorflow_datasets as tfds
```

```
mnist_dataset, mnist_info = tfds.load(name='mnist', with_info=True, as_supervised=True)
mnist_train, mnist_test = mnist_dataset['train'], mnist_dataset['test']

num_validation_samples = 0.1 * mnist_info.splits['train'].num_examples
num_validation_samples = tf.cast(num_validation_samples, tf.int64)

num_test_samples = mnist_info.splits['test'].num_examples
num_test_samples = tf.cast(num_test_samples, tf.int64)

def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255.
    return image, label

scaled_train_and_validation_data = mnist_train.map(scale)
test_data = mnist_test.map(scale)
```

Shuffling the data

Why is shuffling a necessary step?

- We don’t know if the targets (labels) are shuffled or they are sorted

- It could happen that entire batches could contain only one type of target

  We need to spread them as much randomly as possible

Target	Batch Number
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	1
0	2
0	2
0	2
0	2
0	2
0	2
1	2
1	2
1	2
1	2
1	2
1	3
1	3
1	3
1	3
1	3
1	3
1	3
1	3
1	3
1	3
1	3

Shuffling the data

We will now shuffle the data and prepare the Validation set

- Luckily for us, there is a `shuffle` method readily available and we just need to specify the buffer size

```
BUFFER_SIZE = 10000
shuffled_train_and_validation_data = scaled_train_and_validation_data.shuffle(BUFFER_SIZE)
```

- The `BUFFER_SIZE` parameter is here for cases when we're dealing with enormous datasets
  We would not be able to shuffle the whole dataset at once because we could not fit it all in memory
  - Instead, TF would only store `BUFFER_SIZE` samples in memory at a time and shuffles them

    - If `BUFFER_SIZE`=1 => no shuffling will actually happen
    - If `BUFFER_SIZE` >= num samples => shuffling is uniform and at once
    - `BUFFER_SIZE` in between 1 and the num samples, then we would have a computational optimization to approximate uniform shuffling

Shuffling the data

- Once we have scaled and shuffled the data, we can proceed to actually extracting the train and validation

  Our validation data would be equal to 10% of the training set, which we've already calculated

```
validation_data = shuffled_train_and_validation_data.take(num_validation_samples)
train_data = shuffled_train_and_validation_data.skip(num_validation_samples)
```

  - We use the `take()` method to take that many samples
  - Similarly, the `train_data` is everything else, so we `skip` as many samples as there are in the validation dataset

Shuffling the data

- We can also take advantage of the occasion to batch the train data

  This would be very helpful when we train, as we would be able to iterate over the different batches

```
BATCH_SIZE = 100
train_data = train_data.batch(BATCH_SIZE)
validation_data = validation_data.batch(num_validation_samples)
test_data = test_data.batch(num_test_samples)
validation_inputs, validation_targets = next(iter(validation_data))
```

.batch would obviously split the data in batches

Shuffling the data

- We can also take advantage of the occasion to batch the train data

  This would be very helpful when we train, as we would be able to iterate over the different batches

```
BATCH_SIZE = 100
train_data = train_data.batch(BATCH_SIZE)
validation_data = validation_data.batch(num_validation_samples)
test_data = test_data.batch(num_test_samples)
validation_inputs, validation_targets = next(iter(validation_data))
```

The Validation set doesn't need to be split in batches because is not costly to do a forward step
However, **we need to batch it anyway**, because the model expects data in batches
So, we split it in a single batch that contains all the Validation samples

Shuffling the data

- We can also take advantage of the occasion to batch the train data

  This would be very helpful when we train, as we would be able to iterate over the different batches

```
BATCH_SIZE = 100
train_data = train_data.batch(BATCH_SIZE)
validation_data = validation_data.batch(num_validation_samples)
test_data = test_data.batch(num_test_samples)
validation_inputs, validation_targets = next(iter(validation_data))
```

Even the Test set doesn't need to be split in batches
So, we'll follow the same approach as for the Validation set

Shuffling the data

- We can also take advantage of the occasion to batch the train data

  This would be very helpful when we train, as we would be able to iterate over the different batches

```
BATCH_SIZE = 100
train_data = train_data.batch(BATCH_SIZE)
validation_data = validation_data.batch(num_validation_samples)
test_data = test_data.batch(num_test_samples)
validation_inputs, validation_targets = next(iter(validation_data))
```

`iter` creates an **iterator**, and `next` takes the next batch

In fact, with `as_supervized=True` we've got a 2-tuple structure

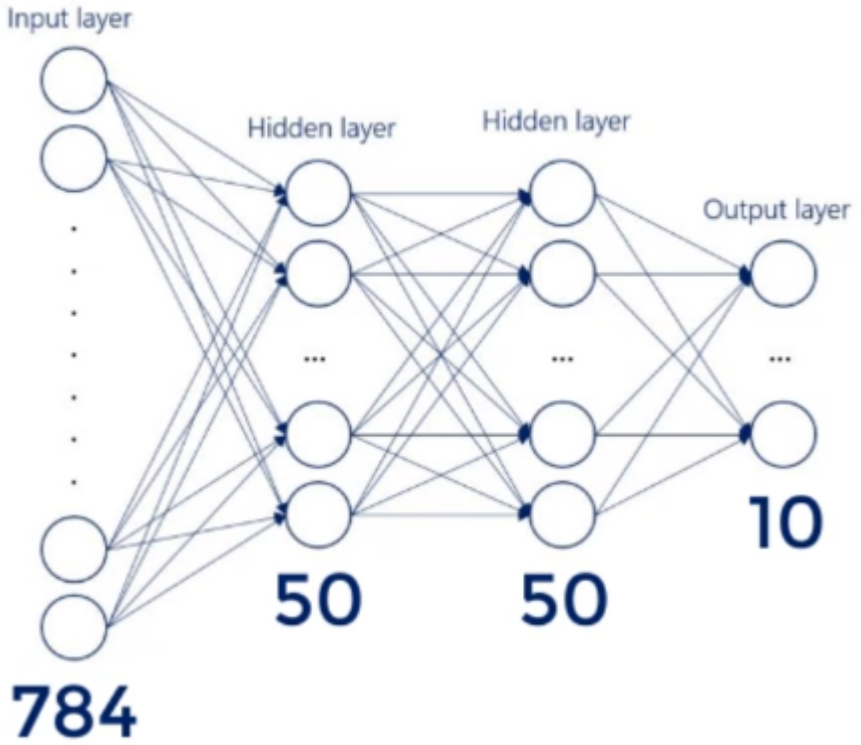
Because it's only one batch, it would load the inputs and the targets

```
mnist_dataset, mnist_info = tfds.load(name='mnist', with_info=True, as_supervised=True)
mnist_train, mnist_test = mnist_dataset['train'], mnist_dataset['test']

num_validation_samples = 0.1 * mnist_info.splits['train'].num_examples
num_validation_samples = tf.cast(num_validation_samples, tf.int64)

num_test_samples = mnist_info.splits['test'].num_examples
num_test_samples = tf.cast(num_test_samples, tf.int64)

def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255.
    return image, label

scaled_train_and_validation_data = mnist_train.map(scale)
test_data = mnist_test.map(scale)

BUFFER_SIZE = 10000
shuffled_train_and_validation_data = scaled_train_and_validation_data.shuffle(BUFFER_SIZE)

validation_data = shuffled_train_and_validation_data.take(num_validation_samples)
train_data = shuffled_train_and_validation_data.skip(num_validation_samples)

BATCH_SIZE = 100
train_data = train_data.batch(BATCH_SIZE)
validation_data = validation_data.batch(num_validation_samples)
test_data = test_data.batch(num_test_samples)
validation_inputs, validation_targets = next(iter(validation_data))
```

The Model

- We want to build something that looks like this.

  The Width and Depth may not be optimal… more testing should be done to increase the accuracy



The Model

```
input_size = 784
output_size = 10
hidden_layer_size = 50

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

- Use same hidden layer size for both hidden layers
  - Not a necessity, you can have different widths and could be better

The Model

```
input_size = 784
output_size = 10
hidden_layer_size = 50

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

- The first layer (the input layer)

  Each observation is 28x28x1 pixels, therefore it is a tensor of rank 3

The Model

```
input_size = 784
output_size = 10
hidden_layer_size = 50

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

- Since we don't know CNNs yet, we don't know how to feed such input into our net, so we must flatten the images into a vector
- There is a convenient method `Flatten` that simply takes our 28x28x1 tensor and orders it into a 784x1 vector

  This allows us to actually create a feed forward neural network

The Model

```
input_size = 784
output_size = 10
hidden_layer_size = 50

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

- The method `tf.keras.layers.Dense` is basically implementing:

```
output = activation(dot(input, weight) + bias)
```

It takes several arguments, but the most important ones for us are the `hidden_layer_size` and the `activation` function

The Model

```
input_size = 784
output_size = 10
hidden_layer_size = 50

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

- We use **relu** (**rectified linear unit**) this time, but you can experiment with something else:

  https://www.tensorflow.org/api_docs/python/tf/keras/activations

The Model

```
input_size = 784
output_size = 10
hidden_layer_size = 50

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

- The final layer is no different, we just make sure to activate it with **softmax**

Optimization Algorithm

We define:

- The optimizer
- The loss function
- The metrics

Optimization Algorithm

We define:

- The optimizer - **Adam**, because we know it is the best choice
- The loss function - **?**
- The metrics - **Accuracy**

- We'd like to employ a **loss** that is used for classifiers, and **cross-entropy** would normally be our first choice

Optimization Algorithm

- There are three built in variations of a **cross entropy loss**:

1. `binary_crossentropy`

   Logically binary cross entropy refers to the case where we've got binary encoding so we won't be choosing this one

2. `categorical_crossentropy`
3. `sparse_categorical_crossentropy`

   `categorical_crossentropy` and `sparse_categorical_crossentropy` are equivalent with the difference that `sparse_categorical_crossentropy` **applies one-hot encoding to the data**

   - Is our data one hot encoded? Well, no… but it will! So, we choose this one

Optimization Algorithm

We define:

- The optimizer - **Adam**, because we know is the best choice
- The loss function - `sparse_categorical_crossentropy`
- The metrics - **Accuracy**

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Training

- We fit the model, specifying the training data, the total number of epochs and the validation data that we just created ourselves in the format: (inputs,targets)

```
NUM_EPOCHS = 5
model.fit(train_data, epochs=NUM_EPOCHS, validation_data=(validation_inputs, validation_targets),
          validation_steps=10, verbose =2)
```

Let’s recap what is going to happen:

1. The training loss will be set to zero

2. The algorithm will iterate over a preset number of batches, all extracted from the training set

   Essentially the whole training set will be utilized but in batches

3. Therefore, the weights and biases will be updated, as many times as there are batches, at the end of each epoch

4. We'll get a value for the loss function indicating how the training is going

5. Moreover, we'll also see a training accuracy thanks to the last argument we added (`verbose=2`)

6. Then, at the end of the epoch, the algorithm will forward propagate the whole validation data set in a single batch through the optimized model and calculate the validation accuracy

- When we reach the maximum number of epochs the training will be over

Training

- We fit the model, specifying the training data, the total number of epochs and the validation data that we just created ourselves in the format: (inputs,targets). Plus, the validation steps to do.

```
NUM_EPOCHS = 5
model.fit(train_data, epochs=NUM_EPOCHS, validation_data=(validation_inputs, validation_targets),
          validation_steps=10, verbose =2)
```

```
Epoch 1/5
540/540 - 4s - loss: 0.4176 - accuracy: 0.8800 - val_loss: 0.2130 - val_accuracy: 0.9388
Epoch 2/5
540/540 - 4s - loss: 0.1795 - accuracy: 0.9466 - val_loss: 0.1424 - val_accuracy: 0.9585
Epoch 3/5
540/540 - 4s - loss: 0.1365 - accuracy: 0.9601 - val_loss: 0.1227 - val_accuracy: 0.9635
Epoch 4/5
540/540 - 4s - loss: 0.1112 - accuracy: 0.9676 - val_loss: 0.1010 - val_accuracy: 0.9705
Epoch 5/5
540/540 - 4s - loss: 0.0937 - accuracy: 0.9722 - val_loss: 0.0864 - val_accuracy: 0.9732
```

```
NUM_EPOCHS = 5
model.fit(train_data, epochs=NUM_EPOCHS, validation_data=(validation_inputs, validation_targets),
          validation_steps=10, verbose =2)
```

```
Epoch 1/5
540/540 - 4s - loss: 0.4176 - accuracy: 0.8800 - val_loss: 0.2130 - val_accuracy: 0.9388
Epoch 2/5
540/540 - 4s - loss: 0.1795 - accuracy: 0.9466 - val_loss: 0.1424 - val_accuracy: 0.9585
Epoch 3/5
540/540 - 4s - loss: 0.1365 - accuracy: 0.9601 - val_loss: 0.1227 - val_accuracy: 0.9635
Epoch 4/5
540/540 - 4s - loss: 0.1112 - accuracy: 0.9676 - val_loss: 0.1010 - val_accuracy: 0.9705
Epoch 5/5
540/540 - 4s - loss: 0.0937 - accuracy: 0.9722 - val_loss: 0.0864 - val_accuracy: 0.9732
```

Number of batches

Time required by the epoch to conclude

The training loss
It doesn't change very much, because we already updated 540 weights and biases in the first epoch only

```
NUM_EPOCHS = 5
model.fit(train_data, epochs=NUM_EPOCHS, validation_data=(validation_inputs, validation_targets),
          validation_steps=10, verbose =2)
```

```
Epoch 1/5
540/540 - 4s - loss: 0.4176 - accuracy: 0.8800 - val_loss: 0.2130 - val_accuracy: 0.9388
Epoch 2/5
540/540 - 4s - loss: 0.1795 - accuracy: 0.9466 - val_loss: 0.1424 - val_accuracy: 0.9585
Epoch 3/5
540/540 - 4s - loss: 0.1365 - accuracy: 0.9601 - val_loss: 0.1227 - val_accuracy: 0.9635
Epoch 4/5
540/540 - 4s - loss: 0.1112 - accuracy: 0.9676 - val_loss: 0.1010 - val_accuracy: 0.9705
Epoch 5/5
540/540 - 4s - loss: 0.0937 - accuracy: 0.9722 - val_loss: 0.0864 - val_accuracy: 0.9732
```

The accuracy
It shows what percent of the cases in our output were equal to the targets

- Logically it follows the trend of the loss

Loss and accuracy for the validation dataset

```
NUM_EPOCHS = 5
model.fit(train_data, epochs=NUM_EPOCHS, validation_data=(validation_inputs, validation_targets),
          validation_steps=10, verbose =2)
```

```
Epoch 1/5
540/540 - 4s - loss: 0.4176 - accuracy: 0.8800 - val_loss: 0.2130 - val_accuracy: 0.9388
Epoch 2/5
540/540 - 4s - loss: 0.1795 - accuracy: 0.9466 - val_loss: 0.1424 - val_accuracy: 0.9585
Epoch 3/5
540/540 - 4s - loss: 0.1365 - accuracy: 0.9601 - val_loss: 0.1227 - val_accuracy: 0.9635
Epoch 4/5
540/540 - 4s - loss: 0.1112 - accuracy: 0.9676 - val_loss: 0.1010 - val_accuracy: 0.9705
Epoch 5/5
540/540 - 4s - loss: 0.0937 - accuracy: 0.9722 - val_loss: 0.0864 - val_accuracy: 0.9732
```

NOTE:

- The accuracy on the validation set is the **true accuracy** of the model for the epoch!

  This is because the **training accuracy** is the average accuracy across batches, while the **validation accuracy** is that of the whole validation set

- The overall accuracy of our model is then given by the validation accuracy for the last epoch

  For us it's a 97% that is remarkable, but… can we could be able to do better

Testing

- To increase the accuracy, try with 100 hidden nodes

What would be consider good accuracy for this problem?

- Well, it's actually **>99.7%**

- Keep fiddling with the hyperparameters and see if you can get closer to 100%!

Testing

- Okay, but… if you got 99.7% accuracy, is that the accuracy of your model?

Tricky question

- No, because that's validation accuracy!

- The more you fiddle the hyperparameters… the more overfitting to the validation set you get!

Testing

- Here comes the **testing set**!

  A set that our model truly has never seen before

```
test_loss, test_accuracy = model.evaluate(test_data)
```

```
      1/Unknown - 1s 1s/step - loss: 0.0918 - accuracy: 0.9742
```

```
print('Test loss: {0:.2f}. Test accuracy: {1:.2f}%'.format(test_loss, test_accuracy*100.))
```

```
Test loss: 0.09. Test accuracy: 97.42%
```

  - The test accuracy should be close to the validation accuracy