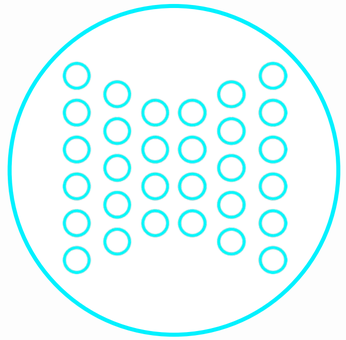
About me :

- ENS -> MVA -> FAIR (engineer) -> FAIR (PhD 3rd year)

About my PhD :

- Interested in sign matrices and tensors (graphs / multi-graphs)
- Observe a few entries, predict the remaining edges
- Factorization methods are simple and efficient for these problems
- When is it needed to go beyond factorization methods ?
- How ?

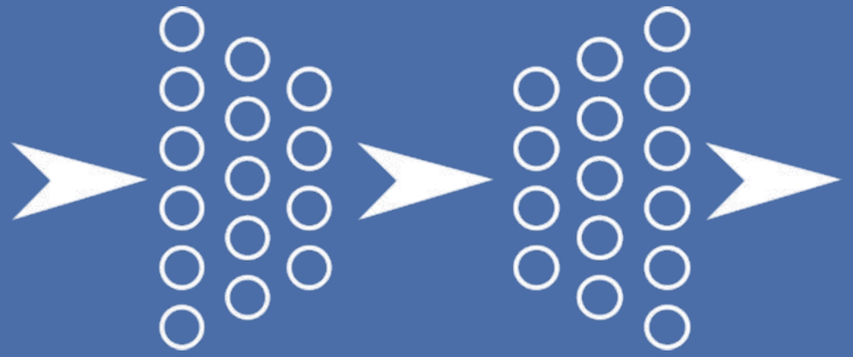


Introduction to Neural Networks

- Artificial Neuron
- Why go deep ?
- Neural Networks

Examples of Inputs / Outputs

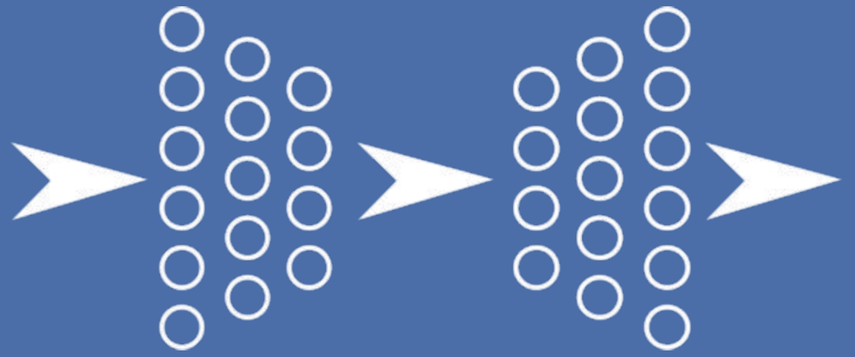
Image In, Class out



Poodle

Examples of Inputs / Outputs

Image In, Text out

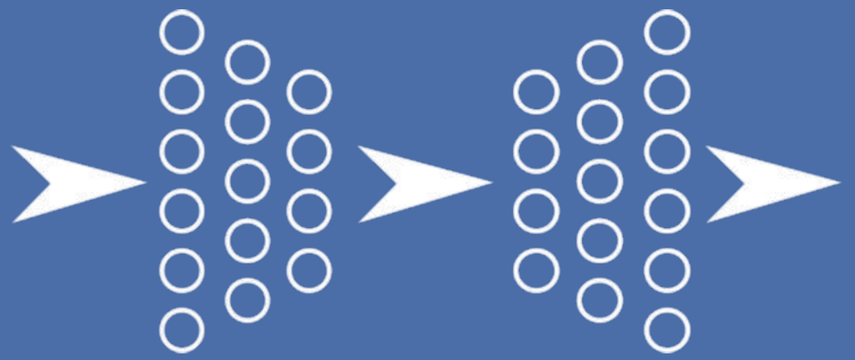


a person on skis
jumping down
part of a hill

Examples of Inputs / Outputs

Text In, Text out

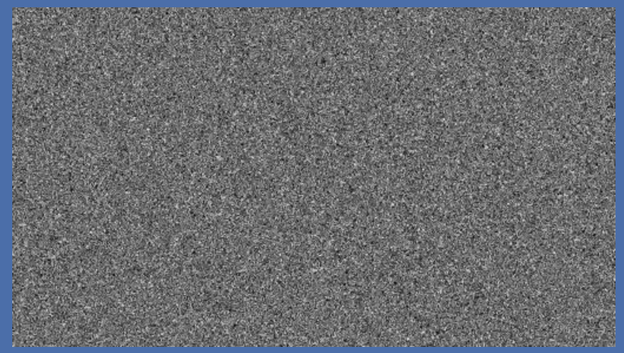
Un skieur sautant au dessus d’un talus

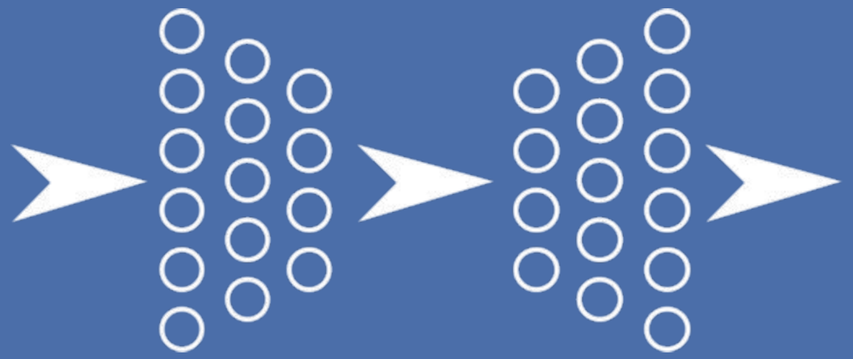


a person on skis jumping down part of a hill

Examples of Inputs / Outputs

Noise In, Face out ?!

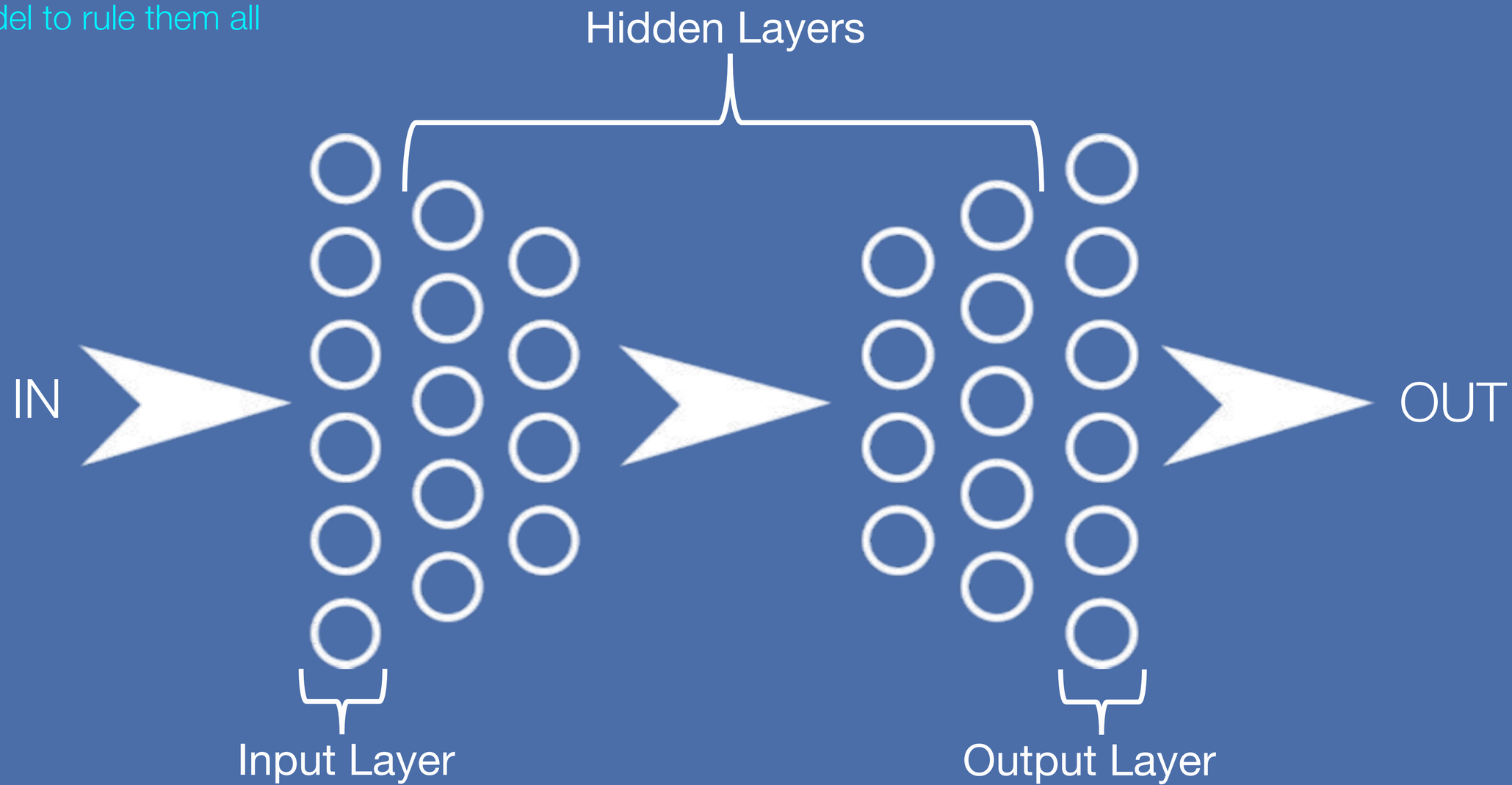




From NVIDIA’s paper at ICLR2018

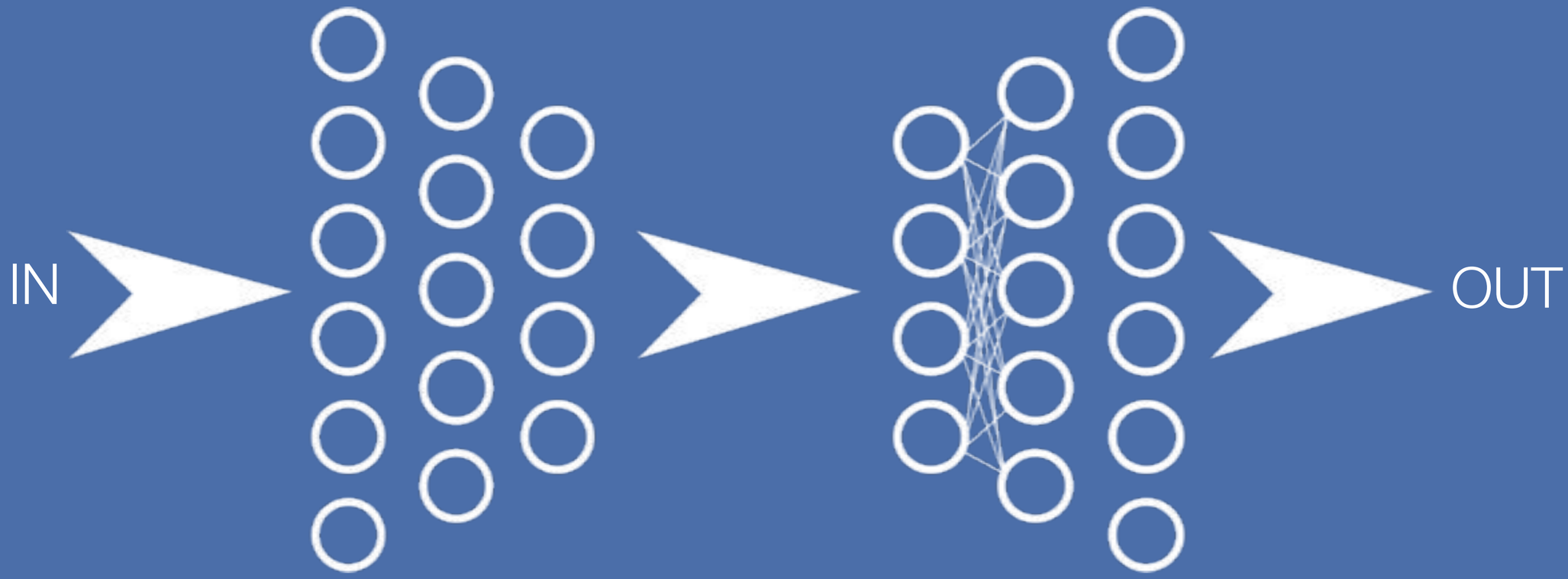
Neural Networks

One model to rule them all



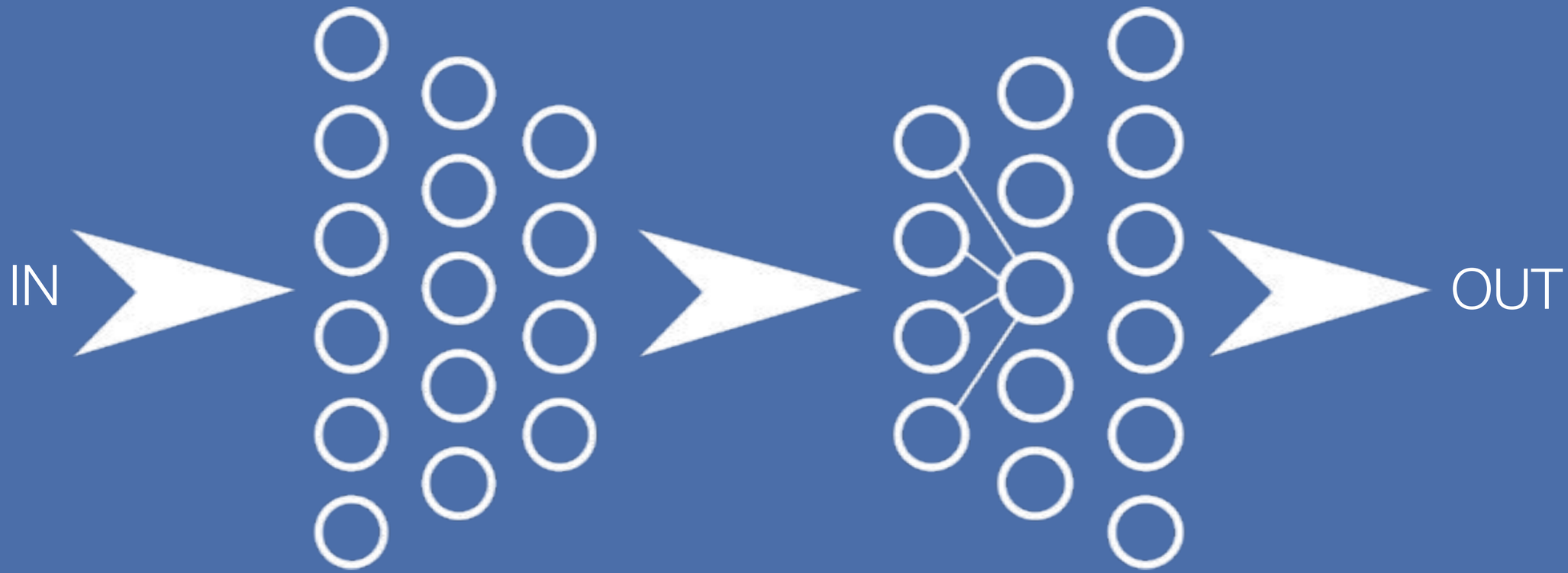
Neural Networks

One model to rule them all



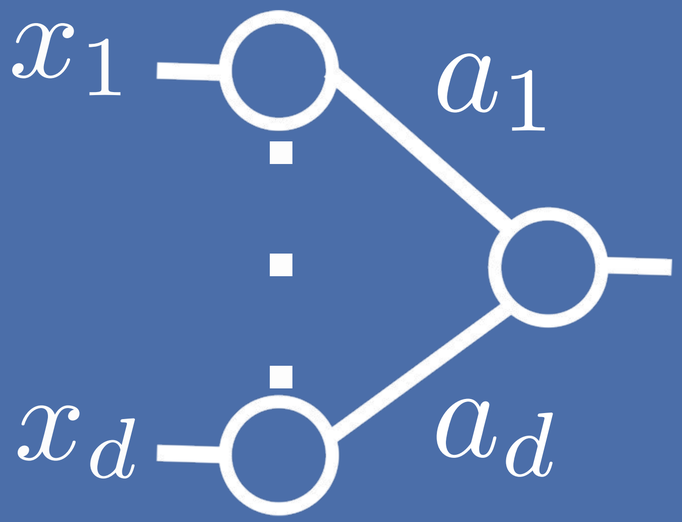
Neural Networks

One model to rule them all



Artificial Neuron

The good ol' Perceptron



$$y = \sigma\big(\overbrace{a_1 x_1 + \ldots + a_d x_d}^{\langle a, x \rangle} + \overbrace{b}^{\text{bias}}\big)$$

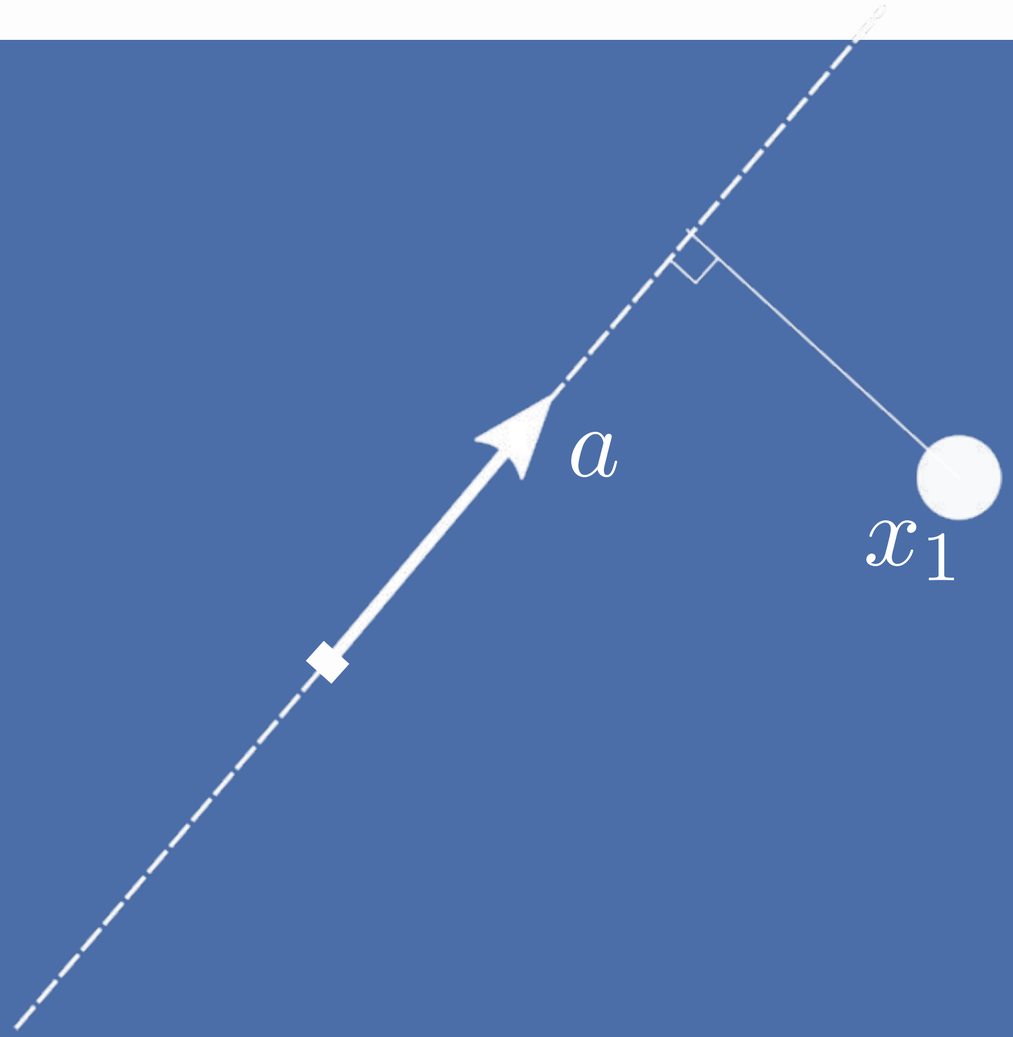
σ: Activation function

Sigmoid

ReLU

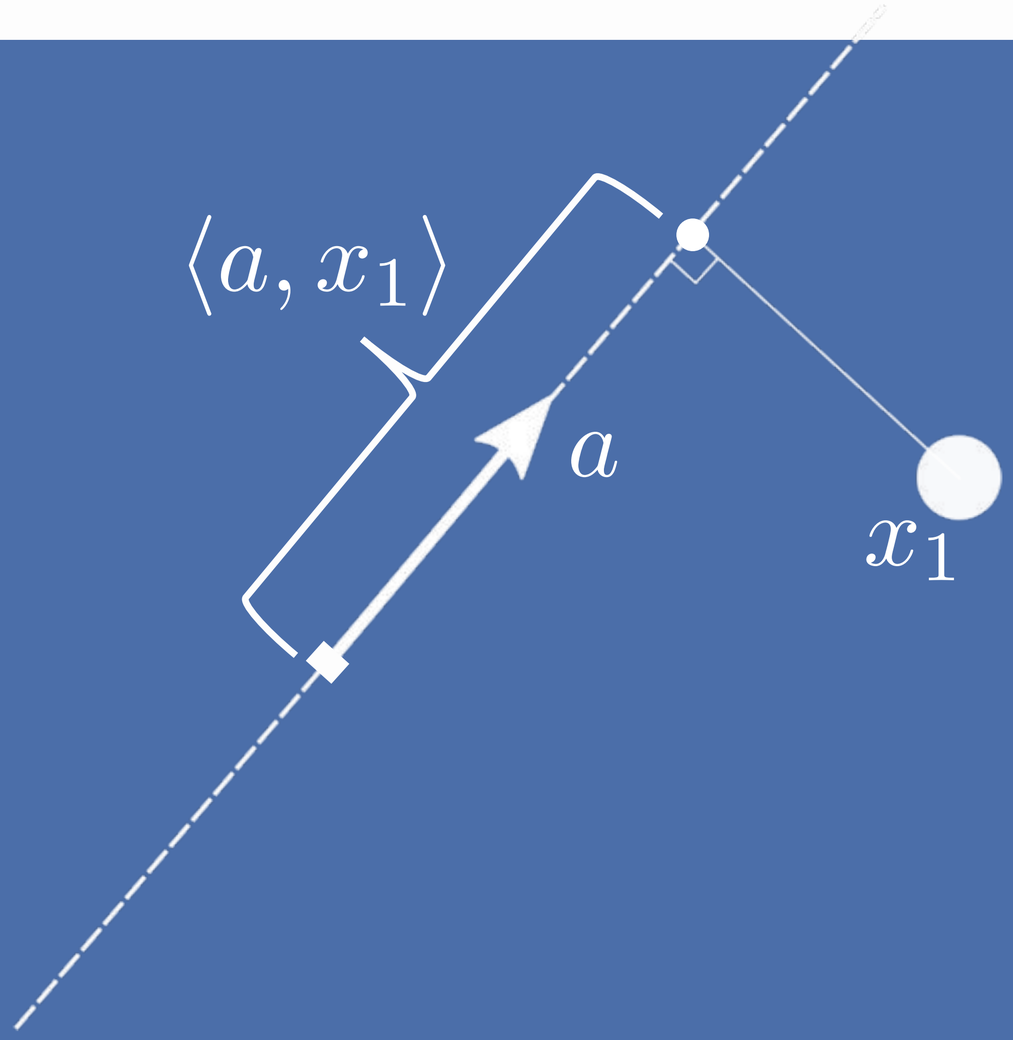
Artificial Neuron

The good ol' Perceptron



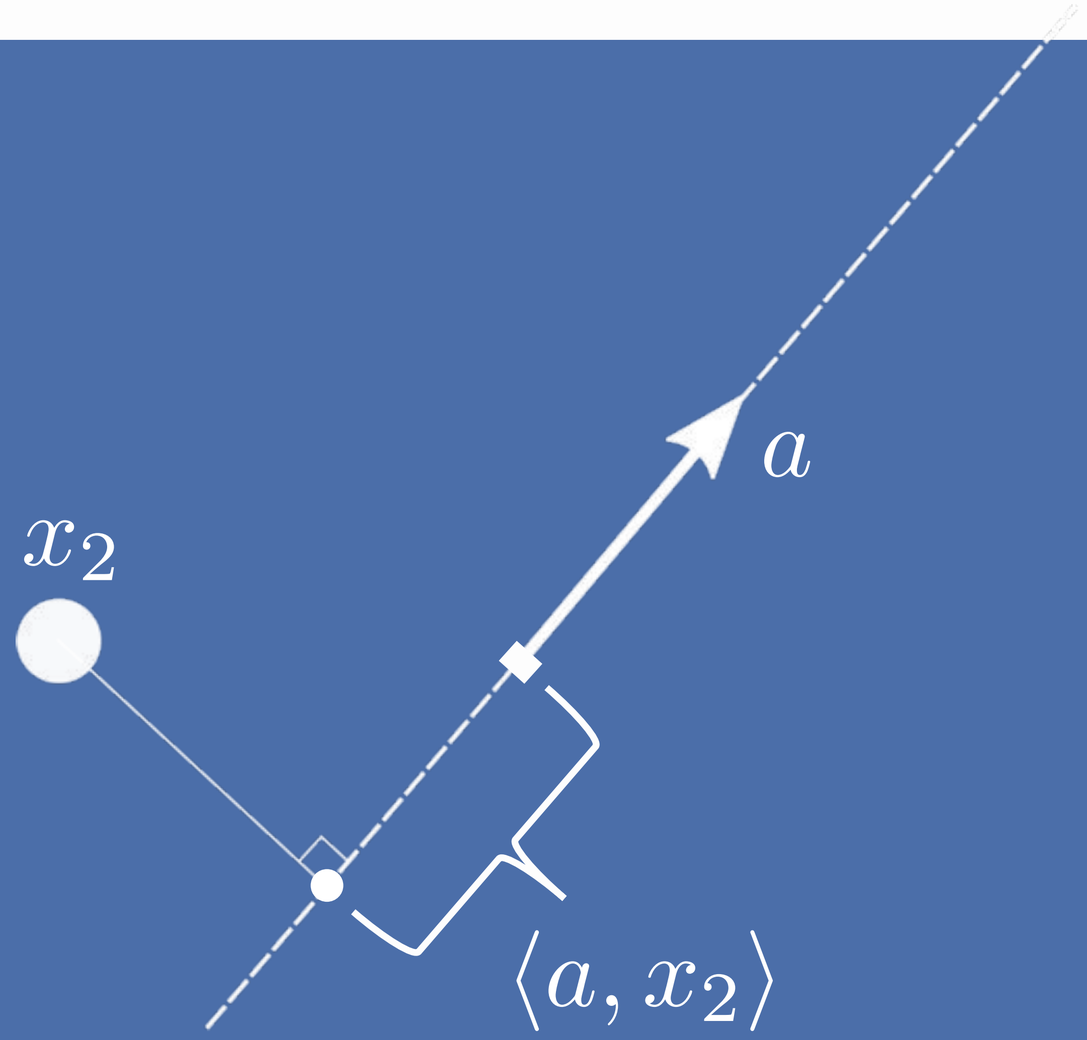
Artificial Neuron

The good ol' Perceptron



Artificial Neuron

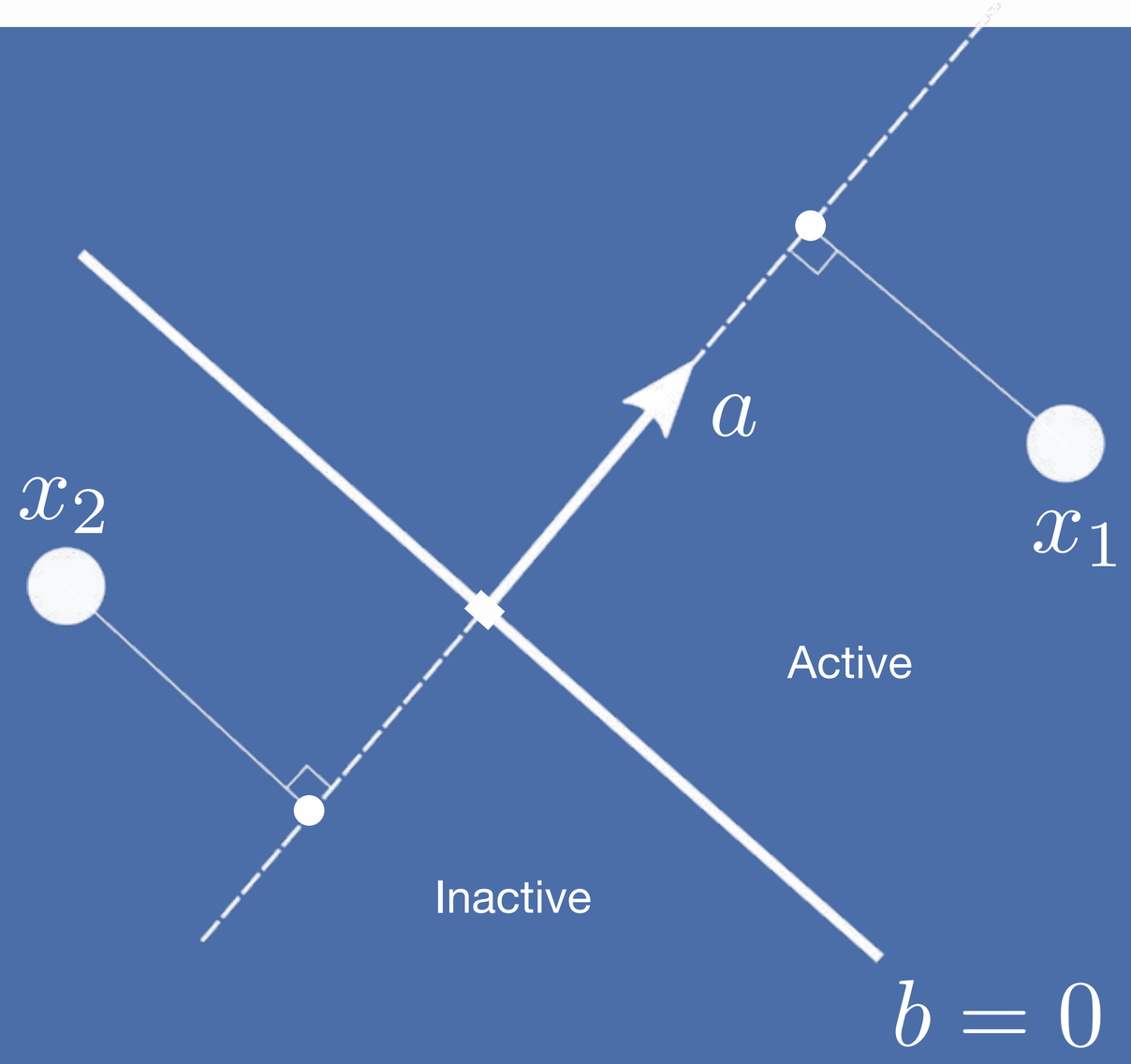
The good ol' Perceptron



Artificial Neuron

The good ol' Perceptron

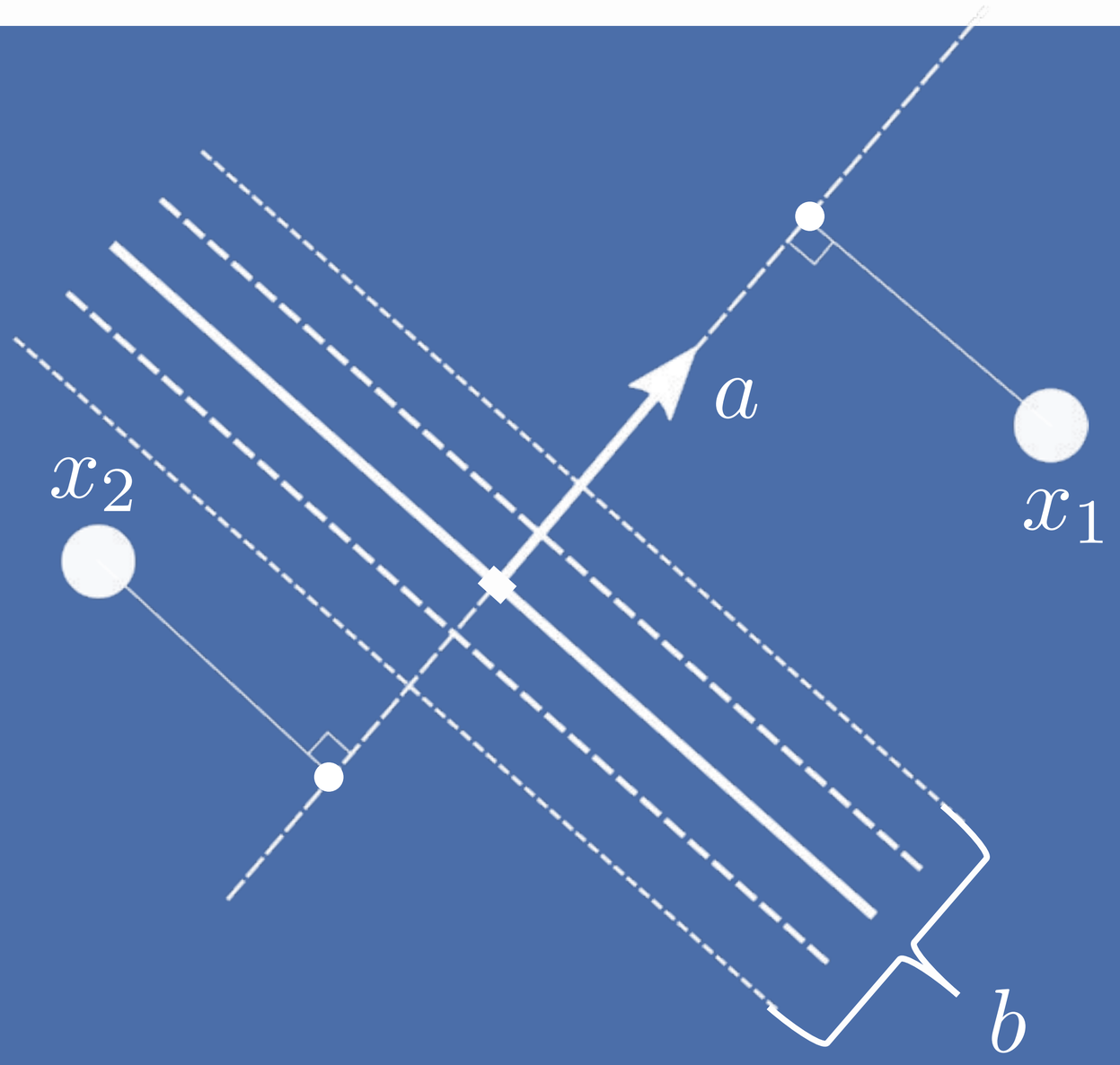
$$y = \sigma(\langle a, x \rangle + b)$$



Artificial Neuron

The good ol' Perceptron

$$y = \sigma(\langle a, x \rangle + b)$$

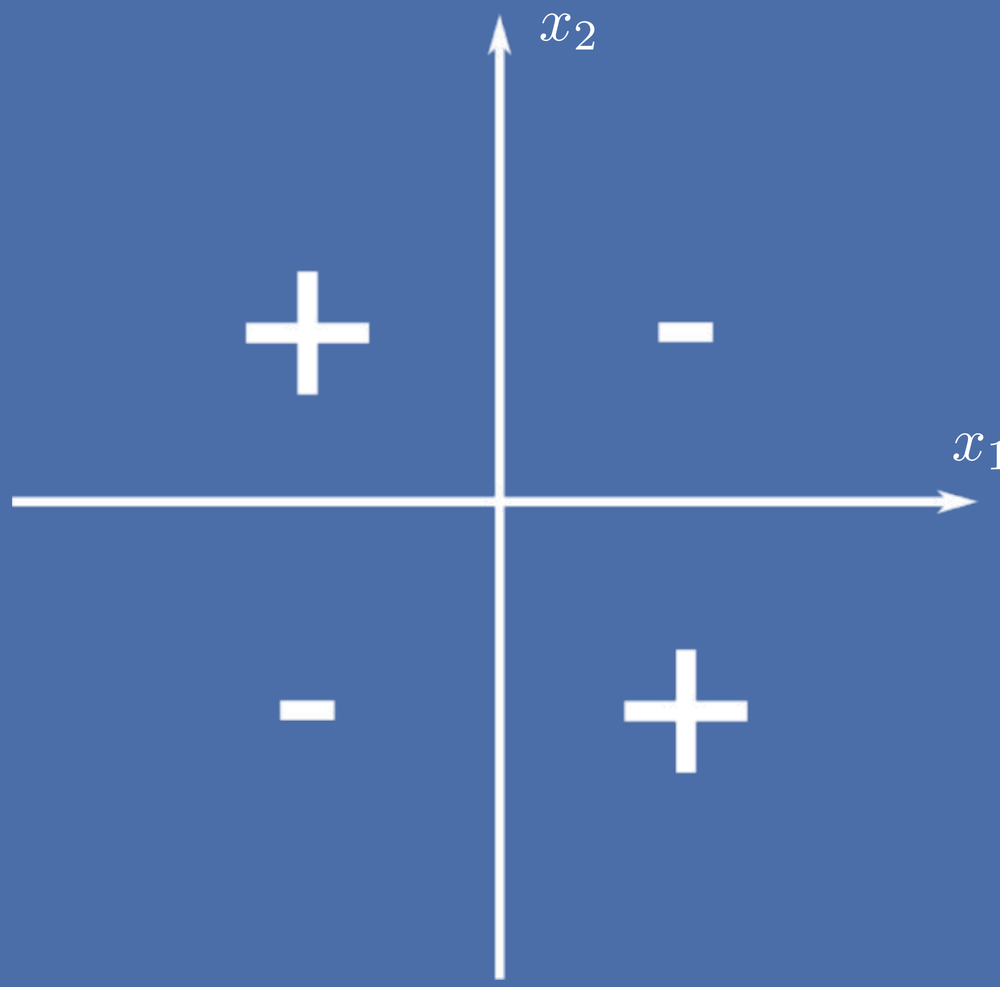


Why deep learning ?

We need to go deeper

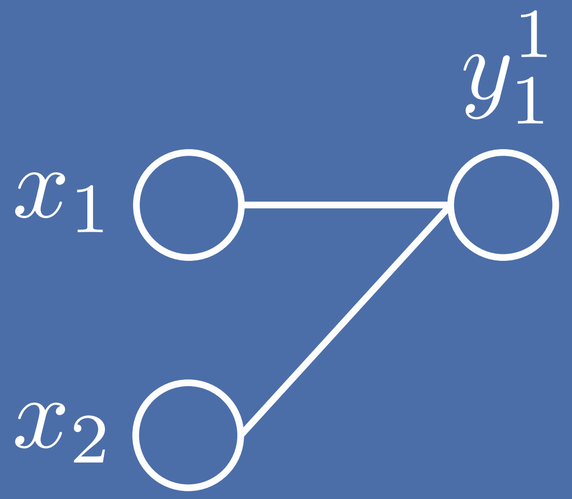
x_1 ○

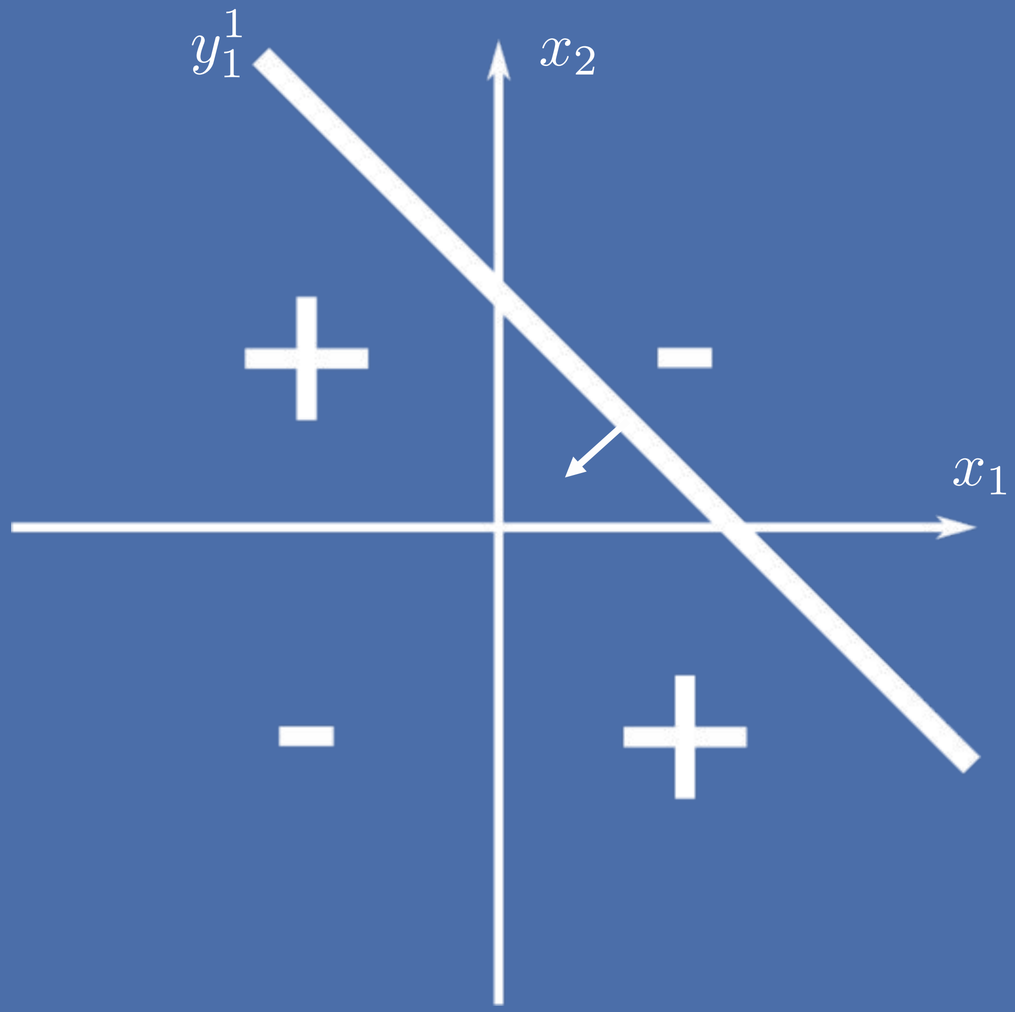
x_2 ○



Why deep learning ?

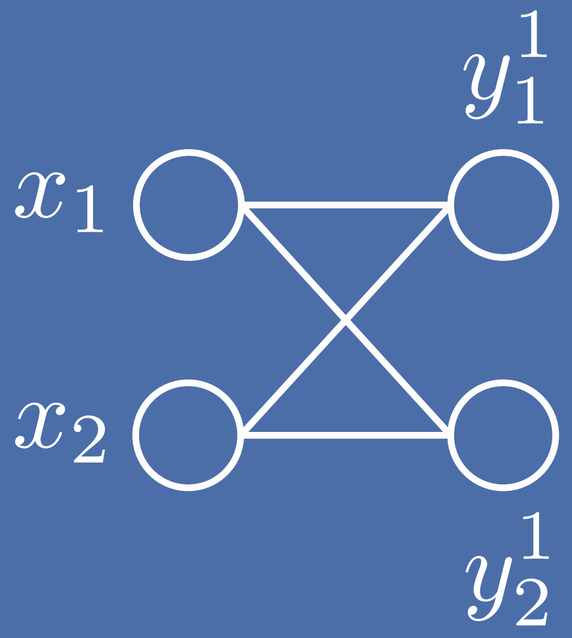
We need to go deeper

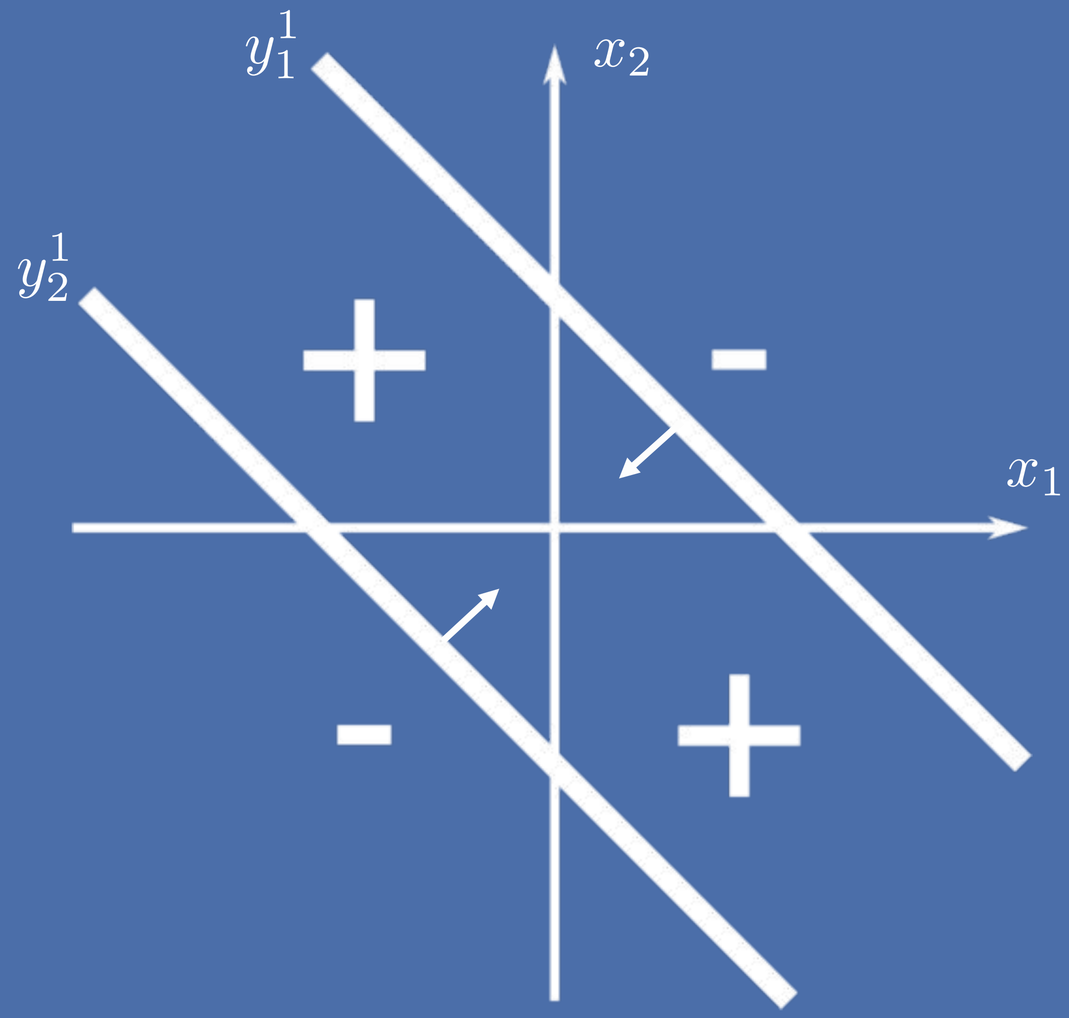




Why deep learning ?

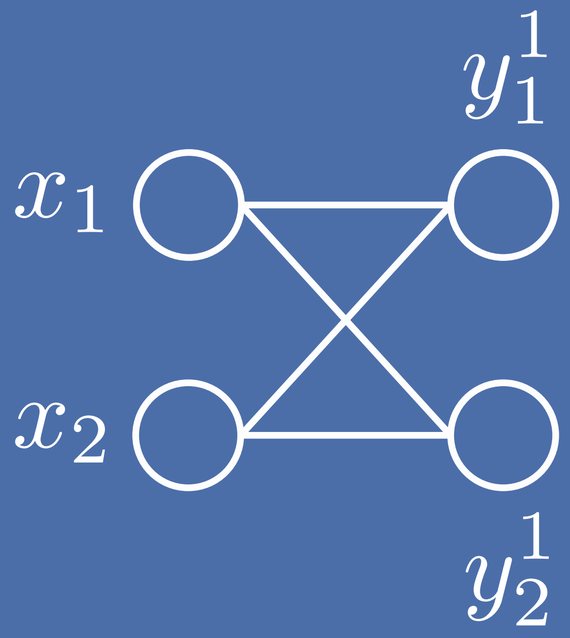
We need to go deeper

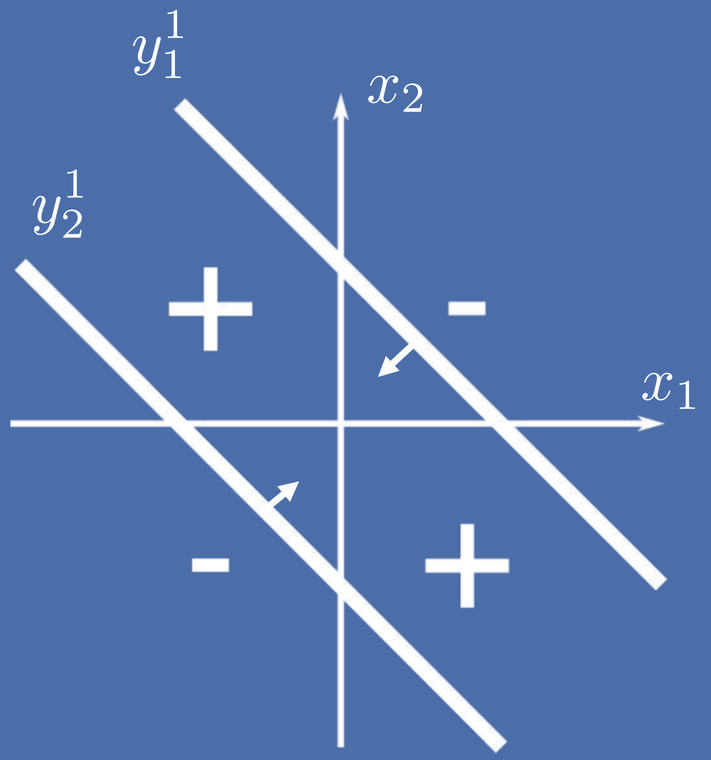


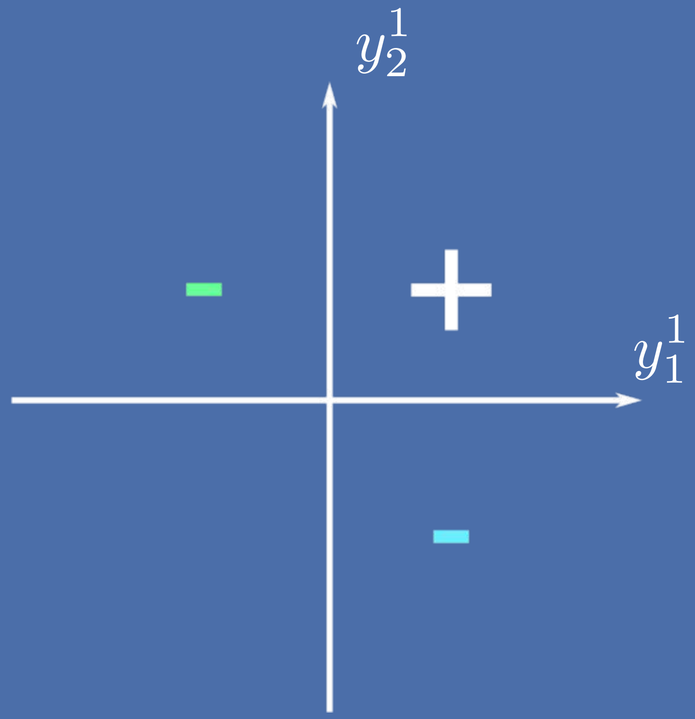


Why deep learning ?

We need to go deeper

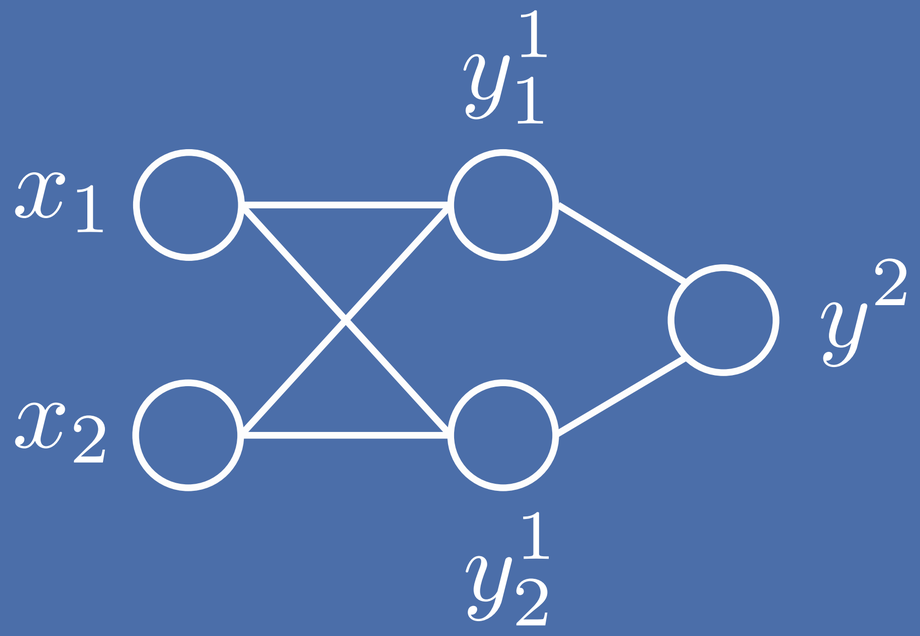


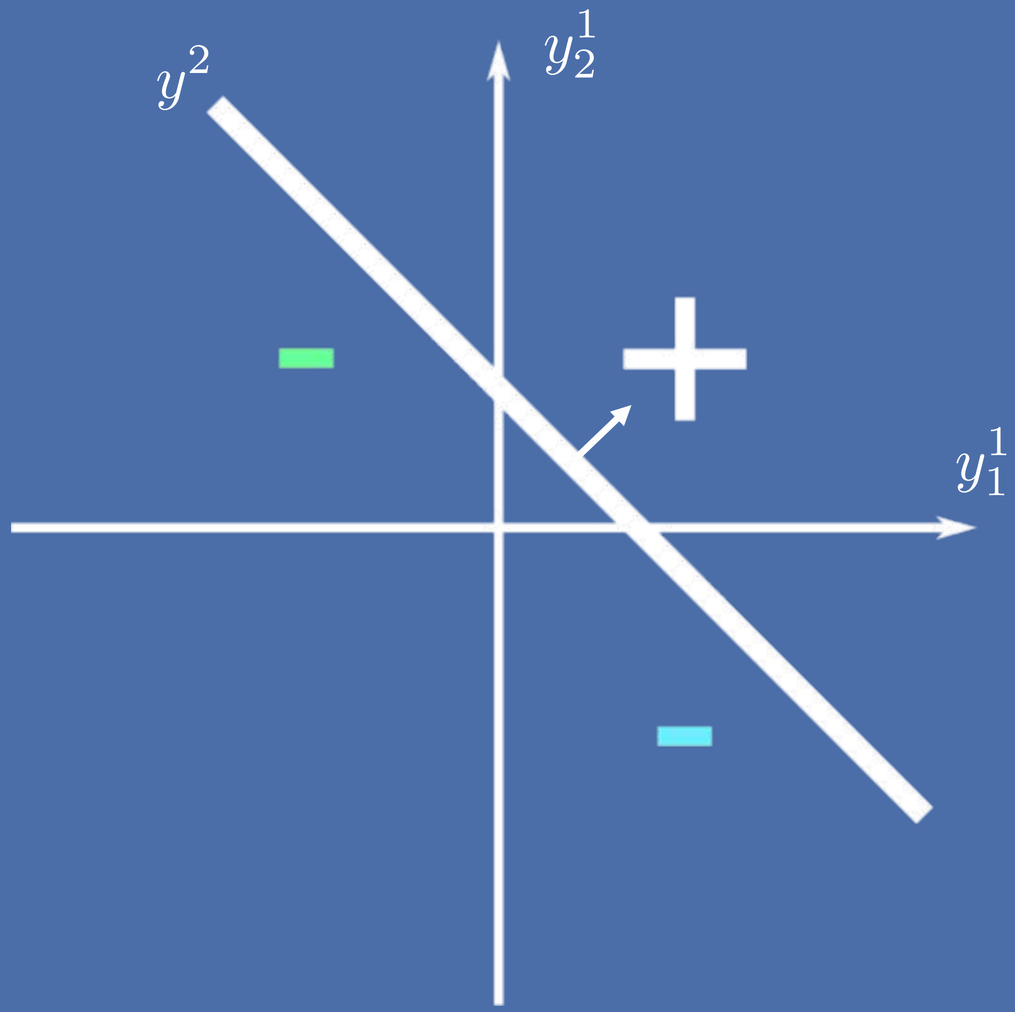




Why deep learning ?

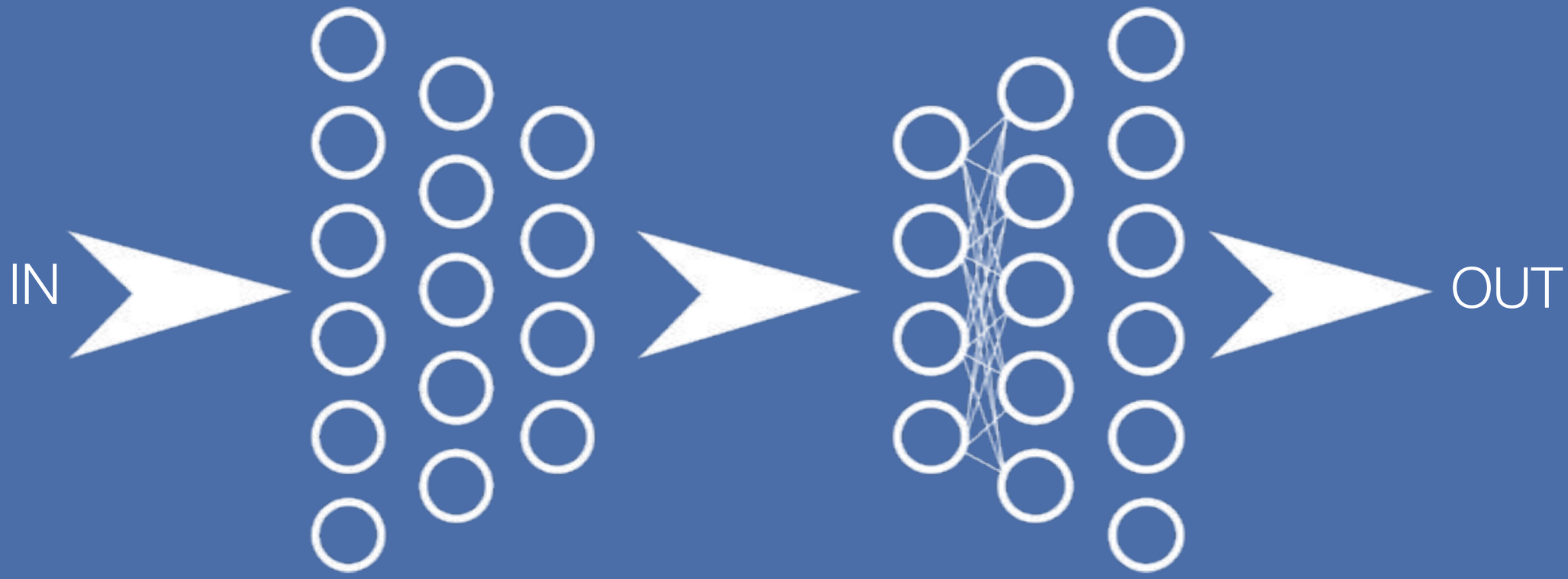
We need to go deeper





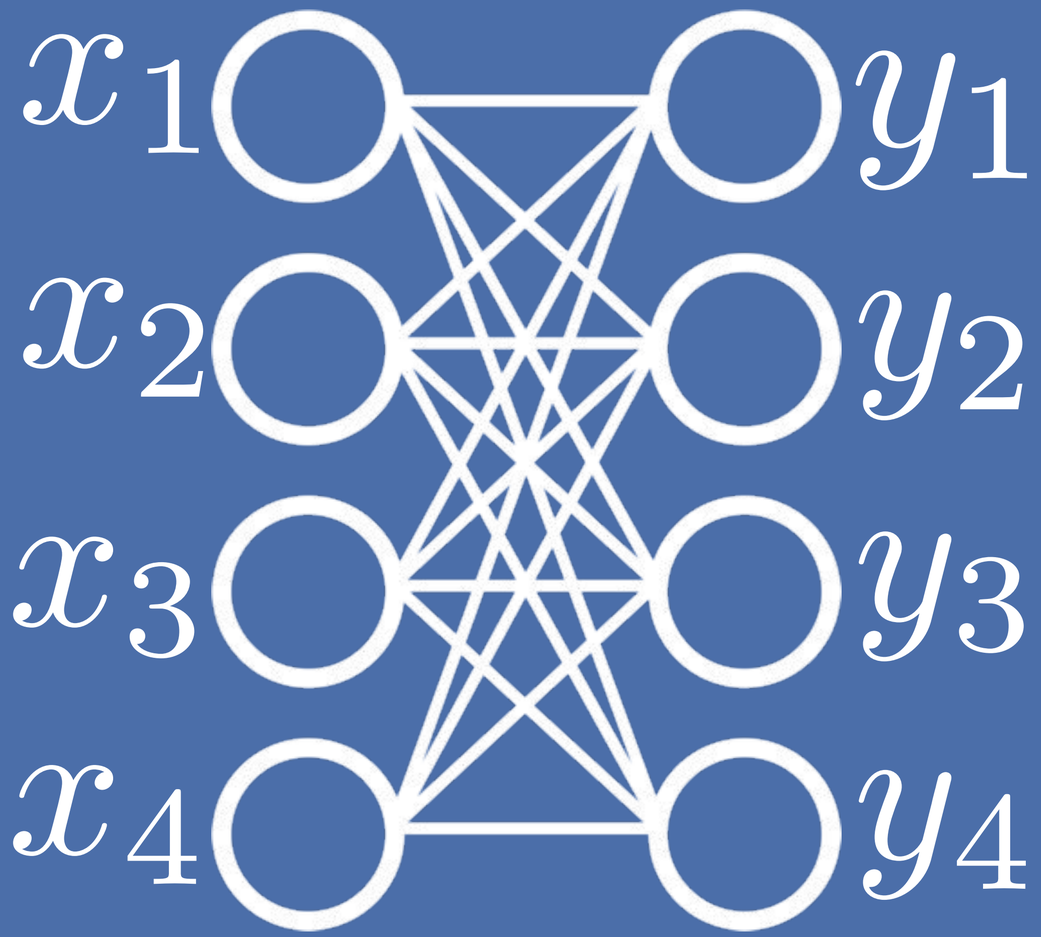
Hidden Layer

And in the darkness bind them



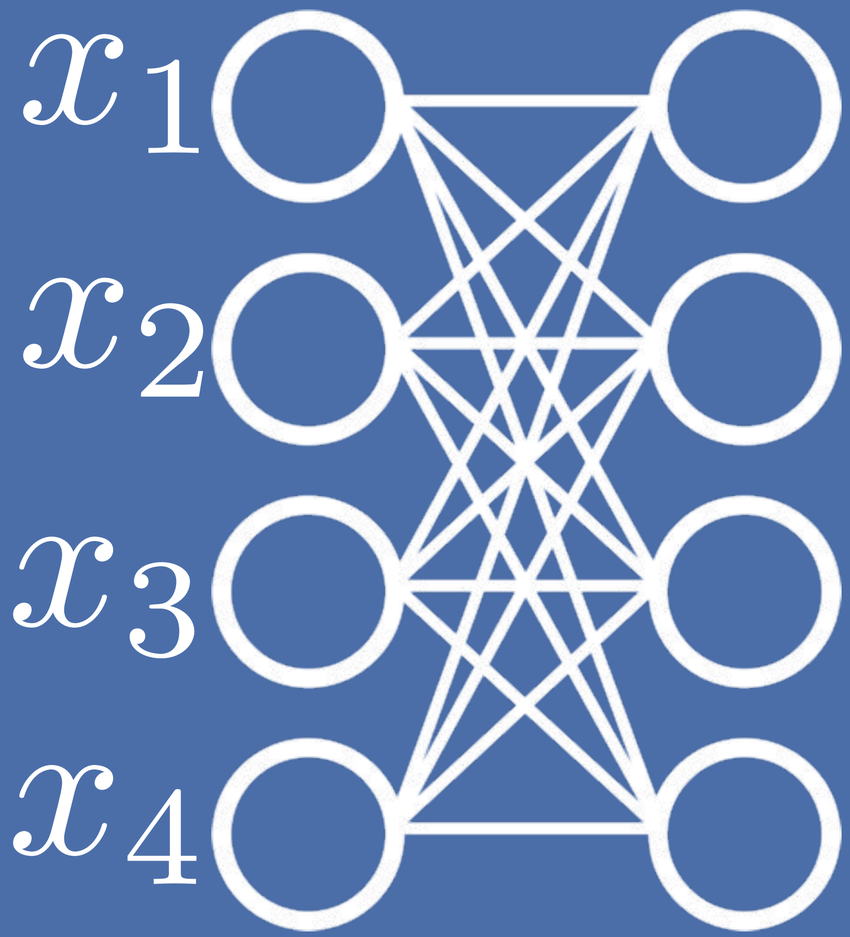
Hidden Layer

Set of neurons with shared inputs



Hidden Layer

Set of neurons with shared inputs



$$y_1 = \sigma(\langle a_1, x \rangle + b_1)$$

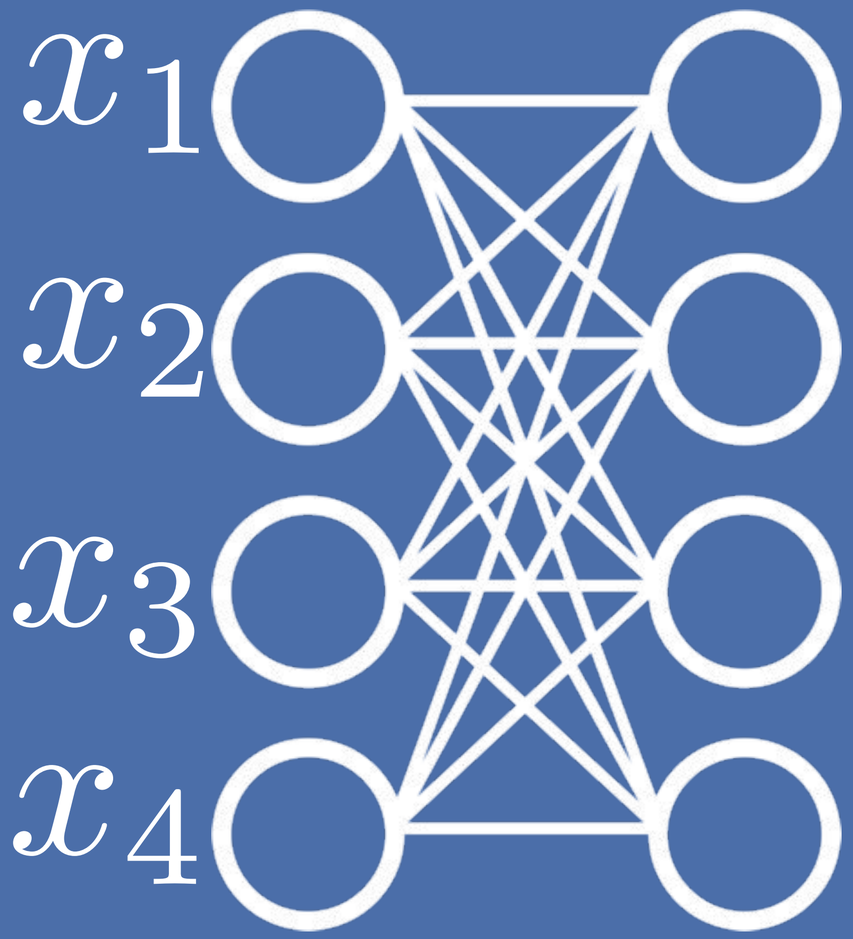
$$y_2 = \sigma(\langle a_2, x \rangle + b_2)$$

$$y_3 = \sigma(\langle a_3, x \rangle + b_3)$$

$$y_4 = \sigma(\langle a_4, x \rangle + b_4)$$

Hidden Layer

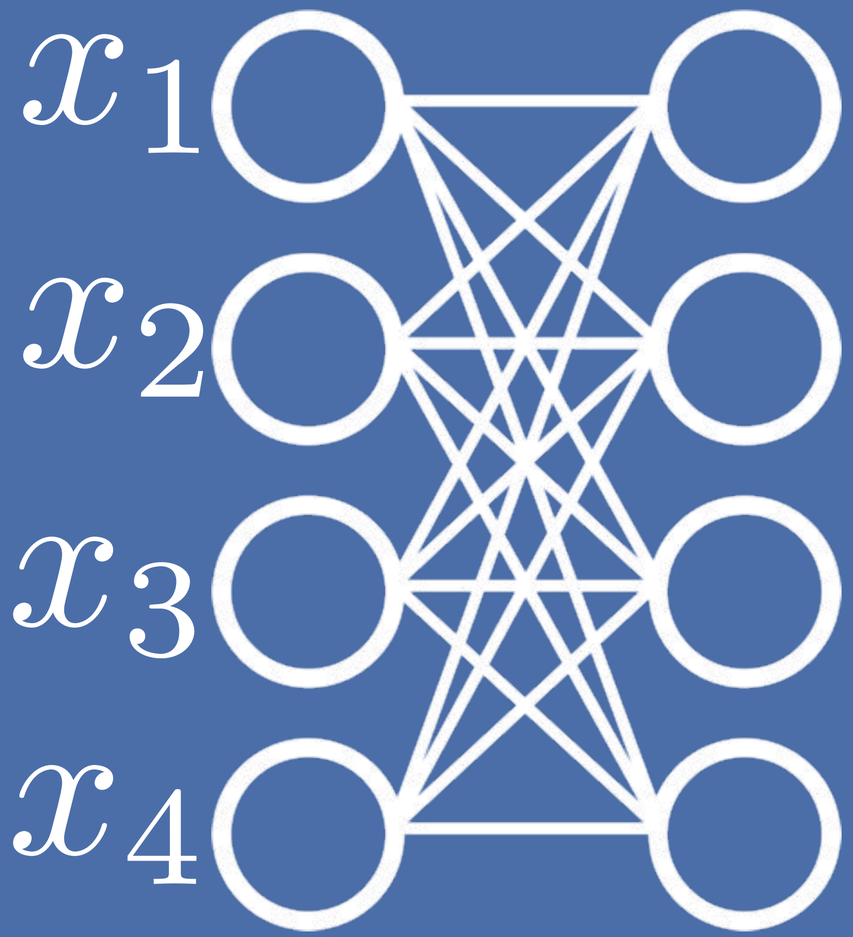
Set of neurons with shared inputs



$$\begin{matrix} & & & A & & b \\ y_1 & = & \sigma(& \langle \begin{bmatrix} a_1 \end{bmatrix}, x \rangle & + \begin{bmatrix} b_1 \end{bmatrix}) \\ y_2 & = & \sigma(& \langle \begin{bmatrix} a_2 \end{bmatrix}, x \rangle & + \begin{bmatrix} b_2 \end{bmatrix}) \\ y_3 & = & \sigma(& \langle \begin{bmatrix} a_3 \end{bmatrix}, x \rangle & + \begin{bmatrix} b_3 \end{bmatrix}) \\ y_4 & = & \sigma(& \langle \begin{bmatrix} a_4 \end{bmatrix}, x \rangle & + \begin{bmatrix} b_4 \end{bmatrix}) \end{matrix}$$

Hidden Layer

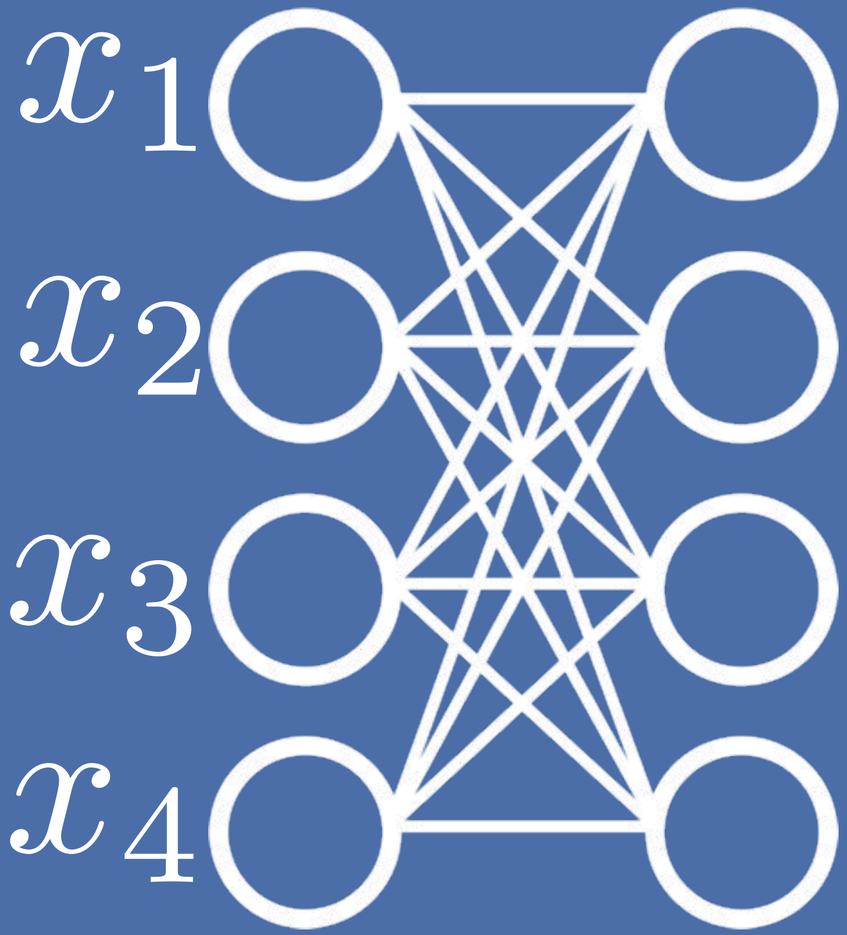
Set of neurons with shared inputs

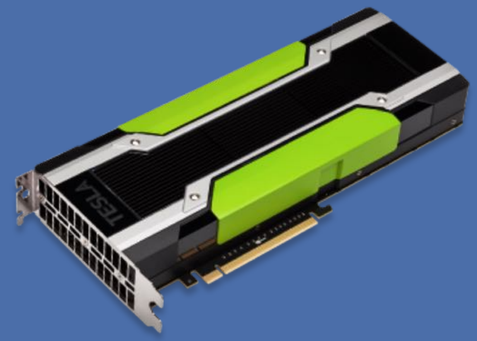


$$y = \sigma(Ax + b)$$

Hidden Layer

Set of neurons with shared inputs

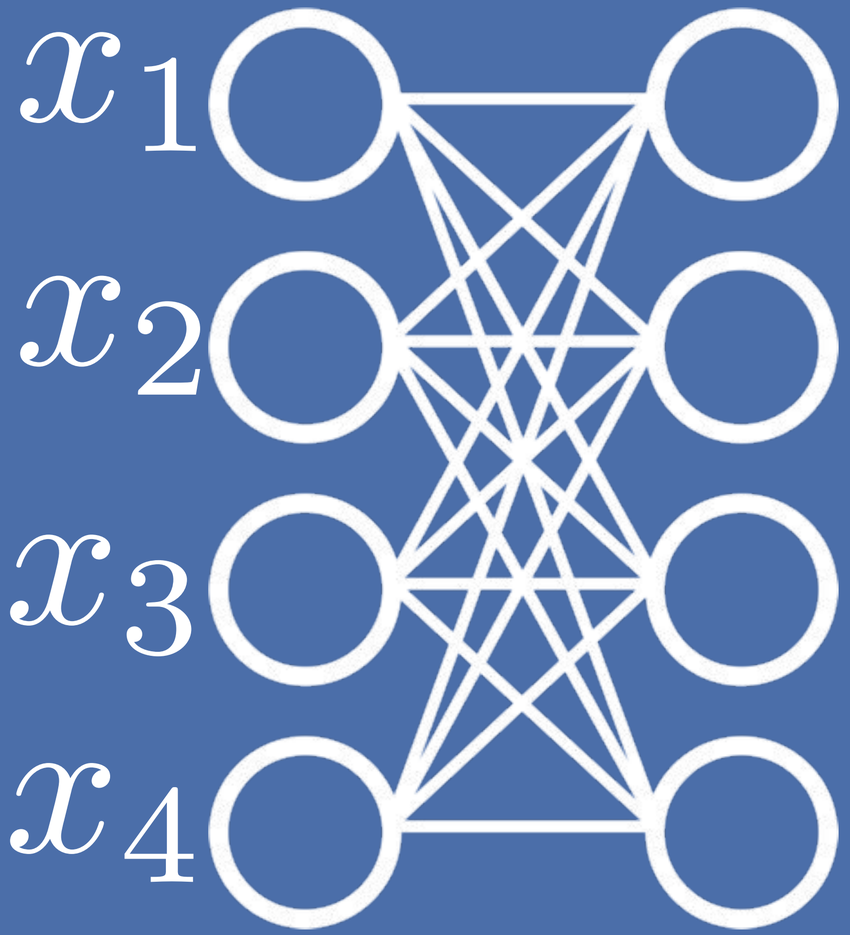




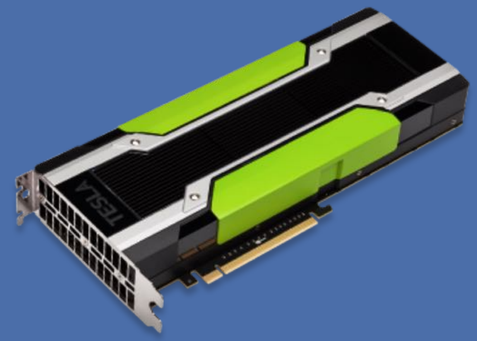
$$y = \sigma\left(\left[Ax + b\right]\right)$$

Hidden Layer

Set of neurons with shared inputs



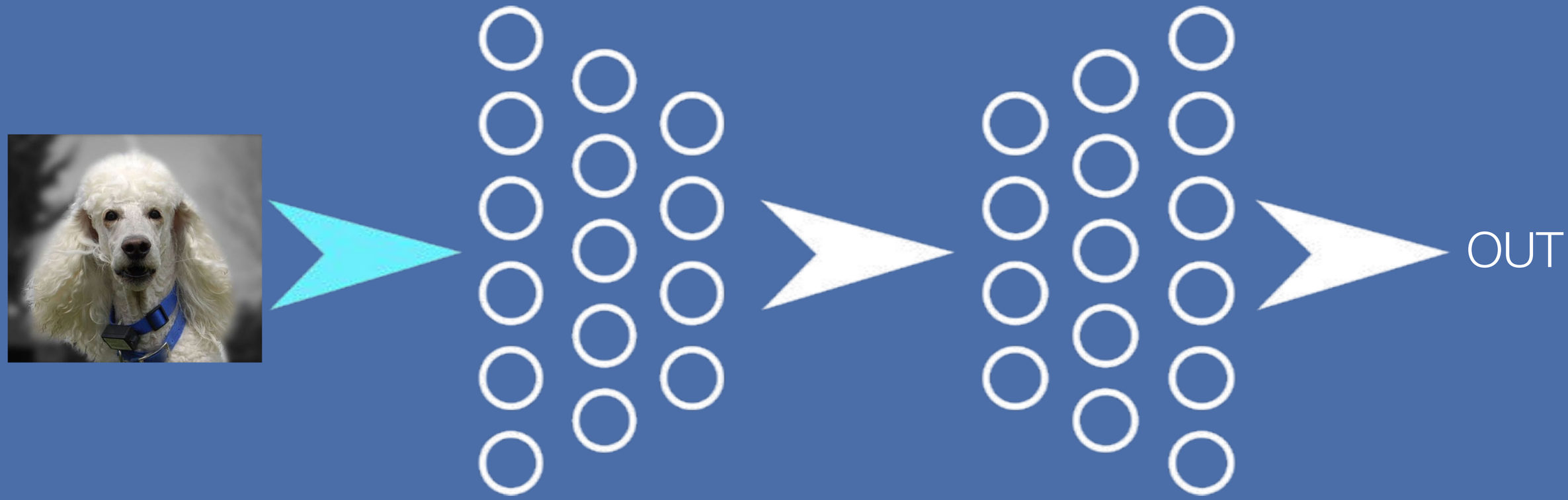
> 100x speed-up



$$y = \sigma\left(\left[Ax + b\right]\right)$$

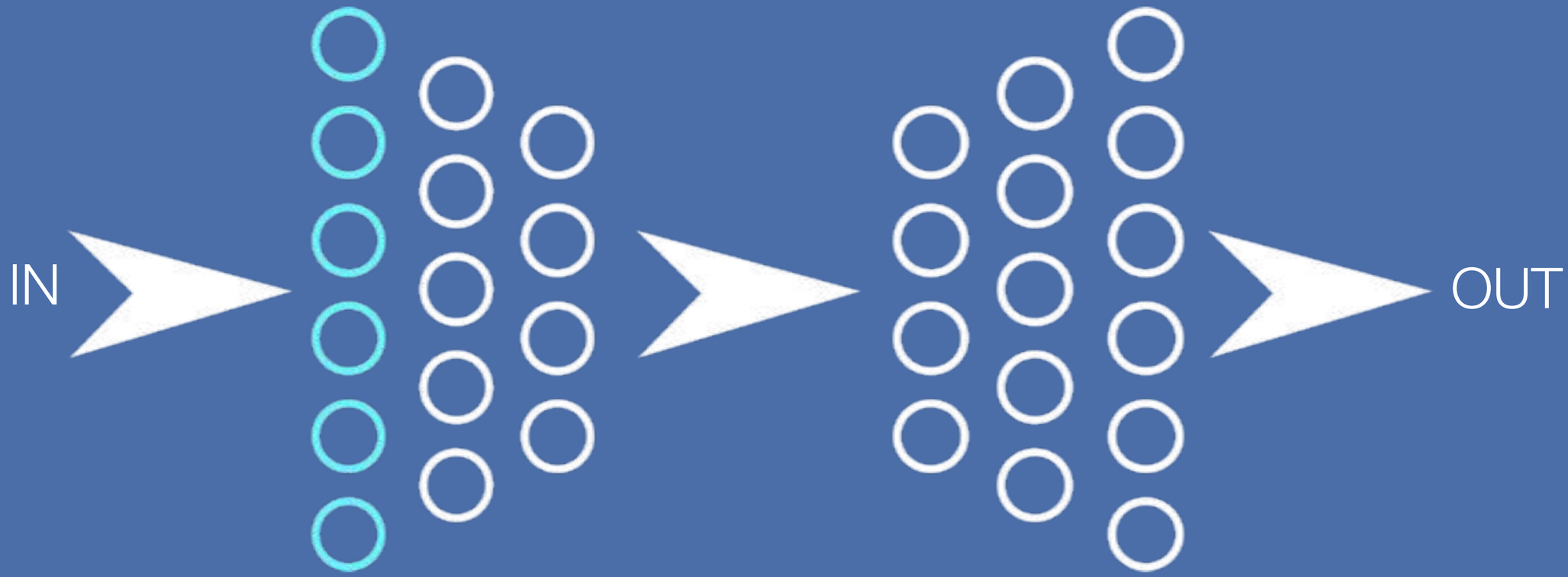
Forward Pass

Hmm what's this picture …



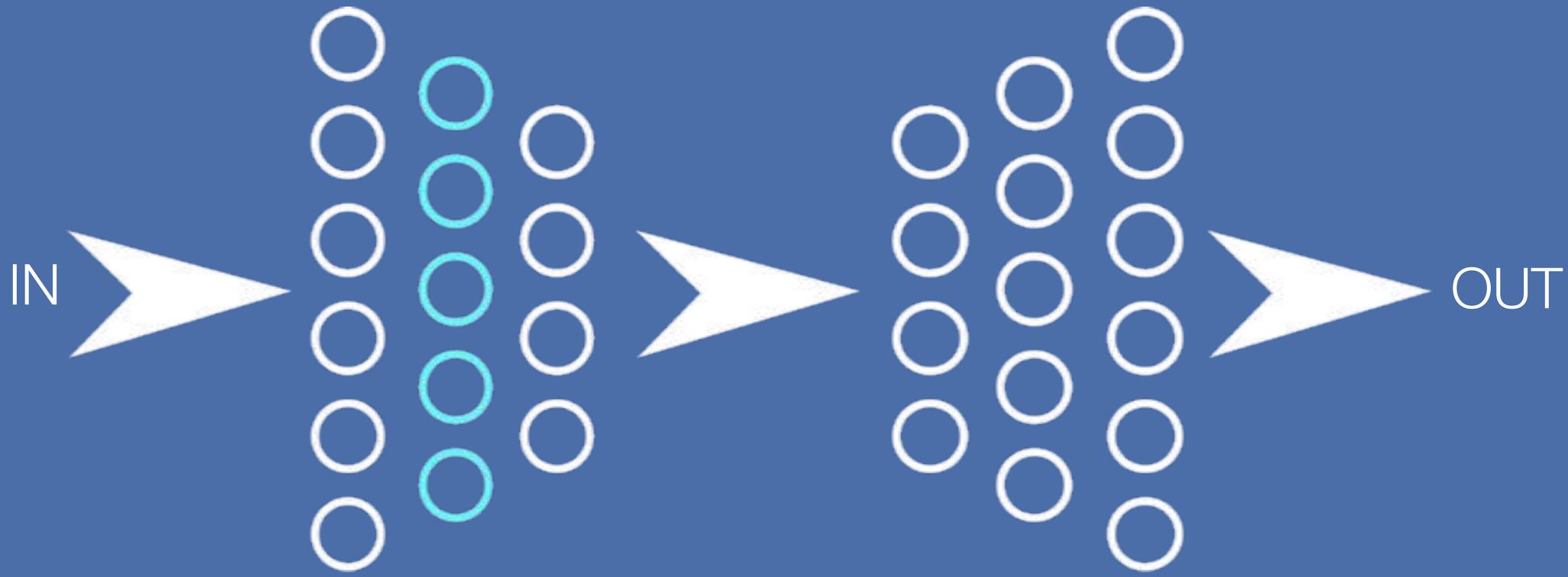
Forward Pass

Wait for it …



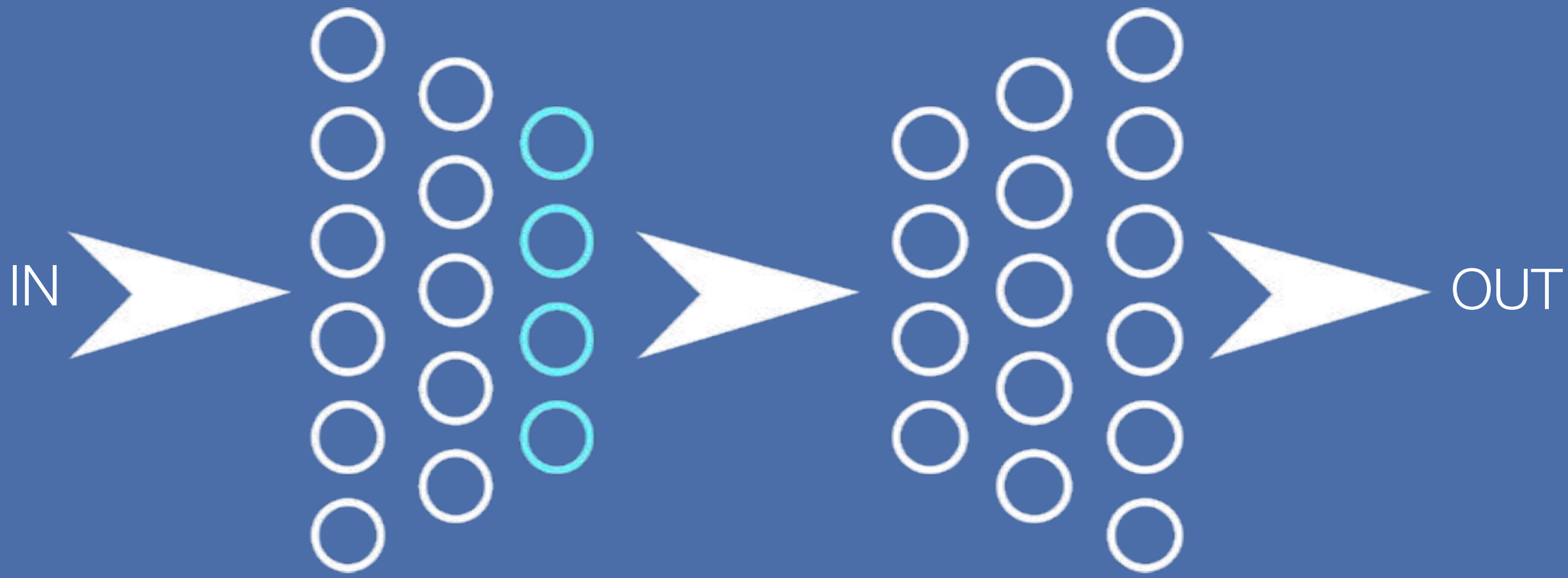
Forward Pass

...



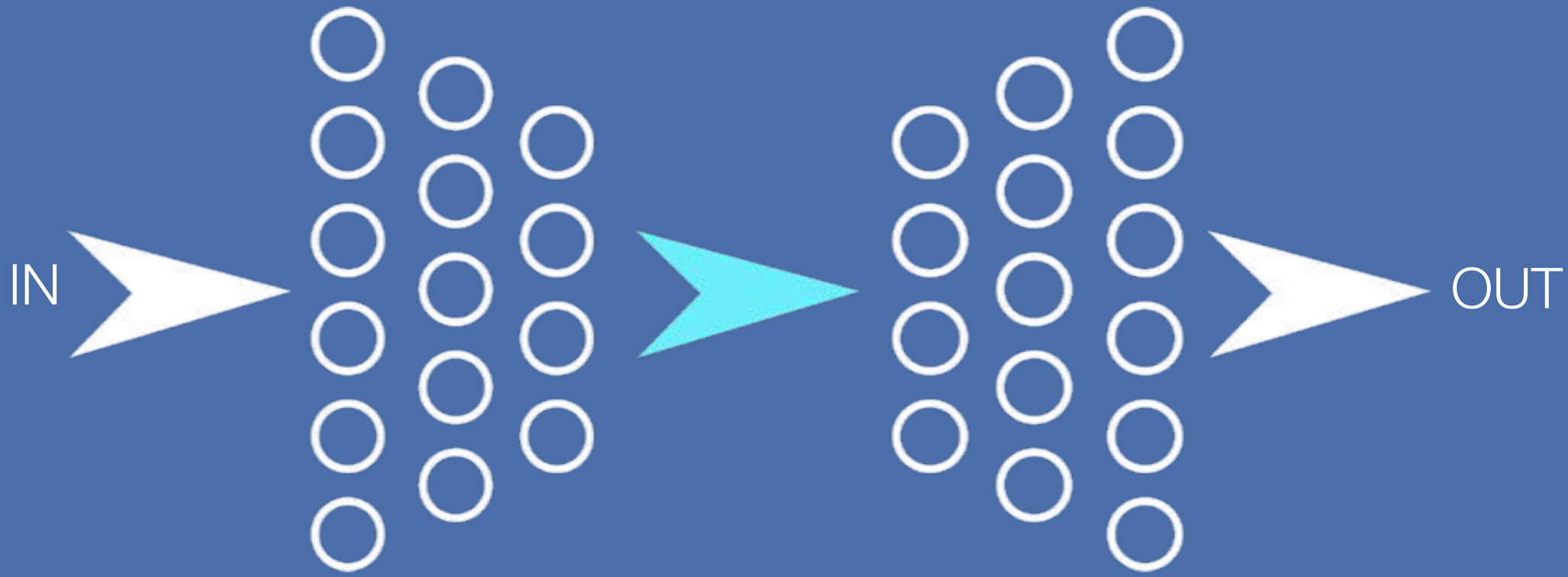
Forward Pass

...



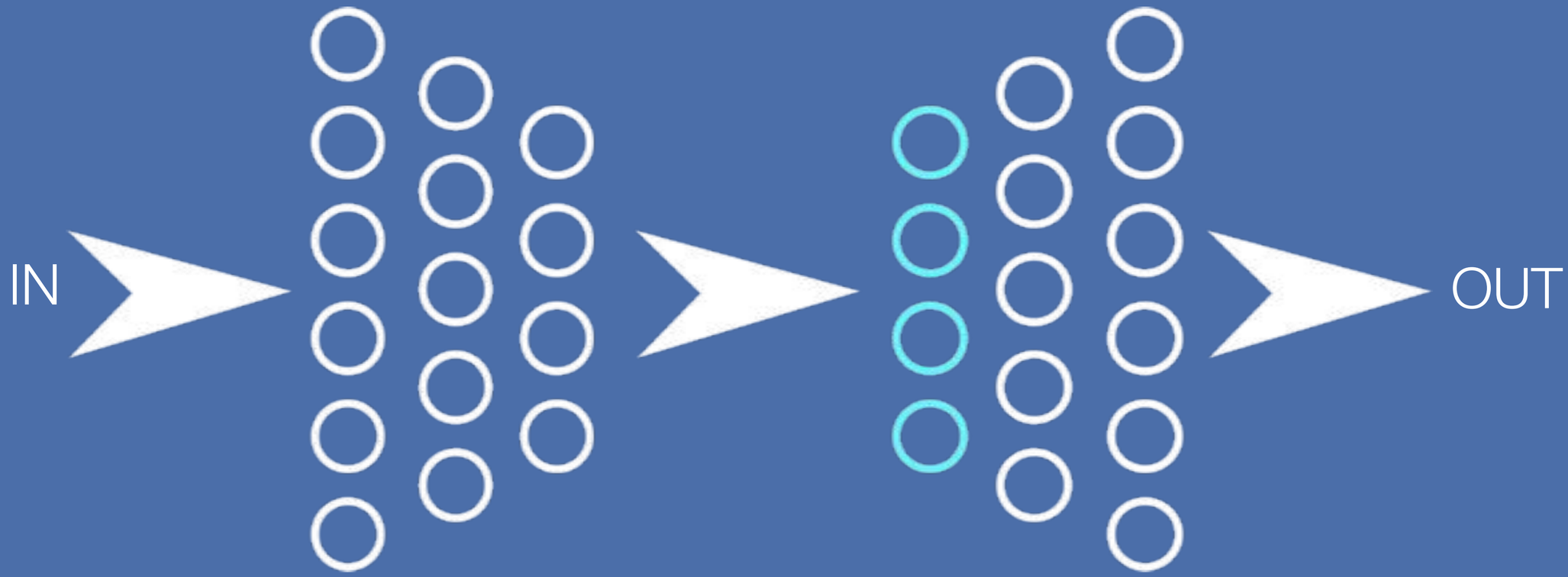
Forward Pass

...



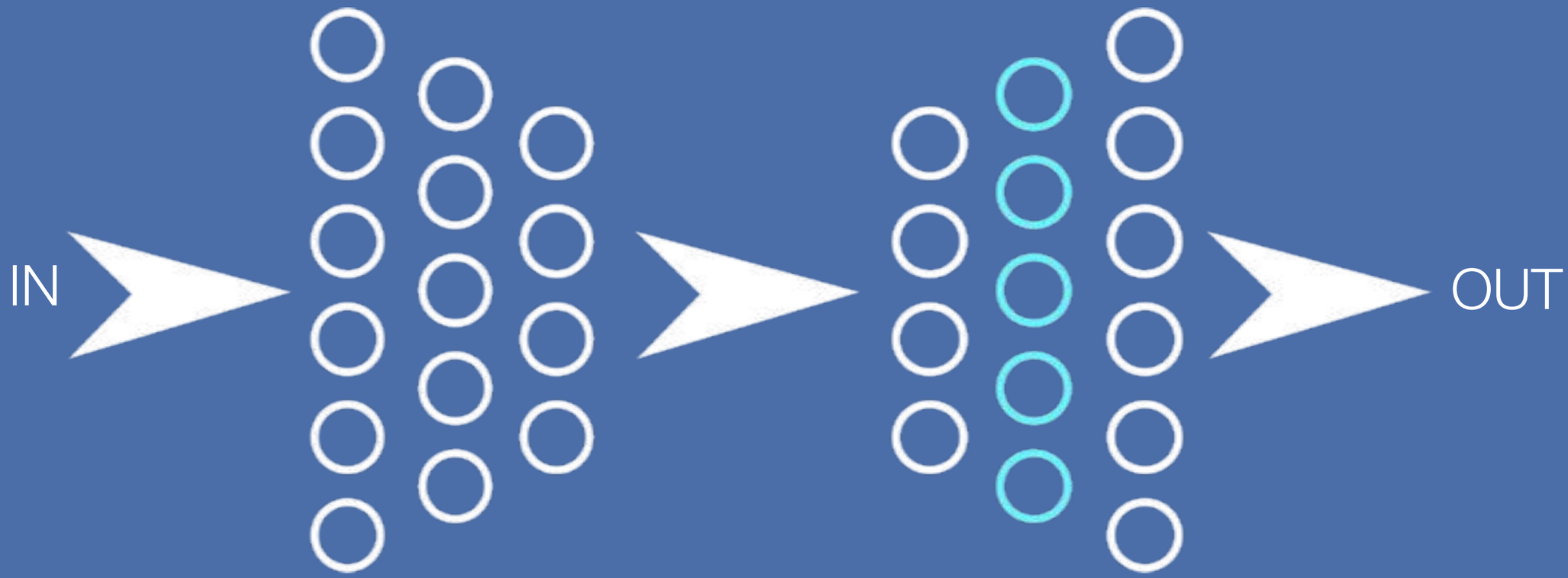
Forward Pass

...



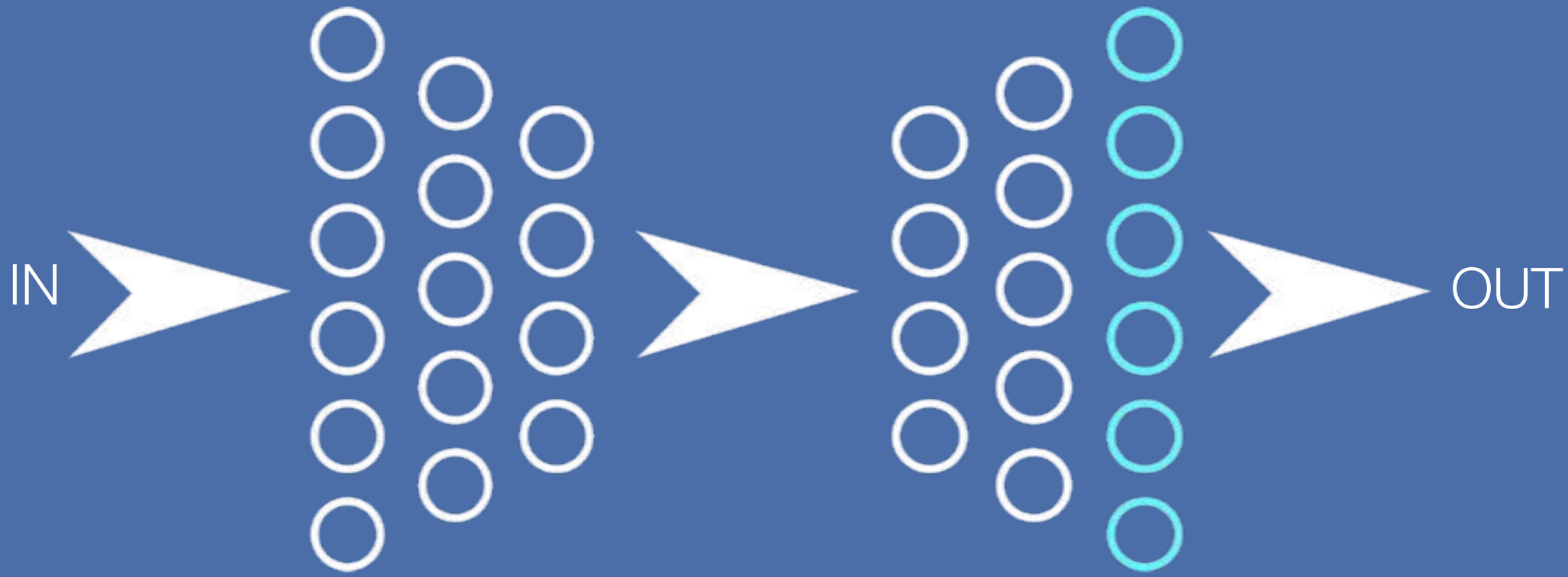
Forward Pass

...



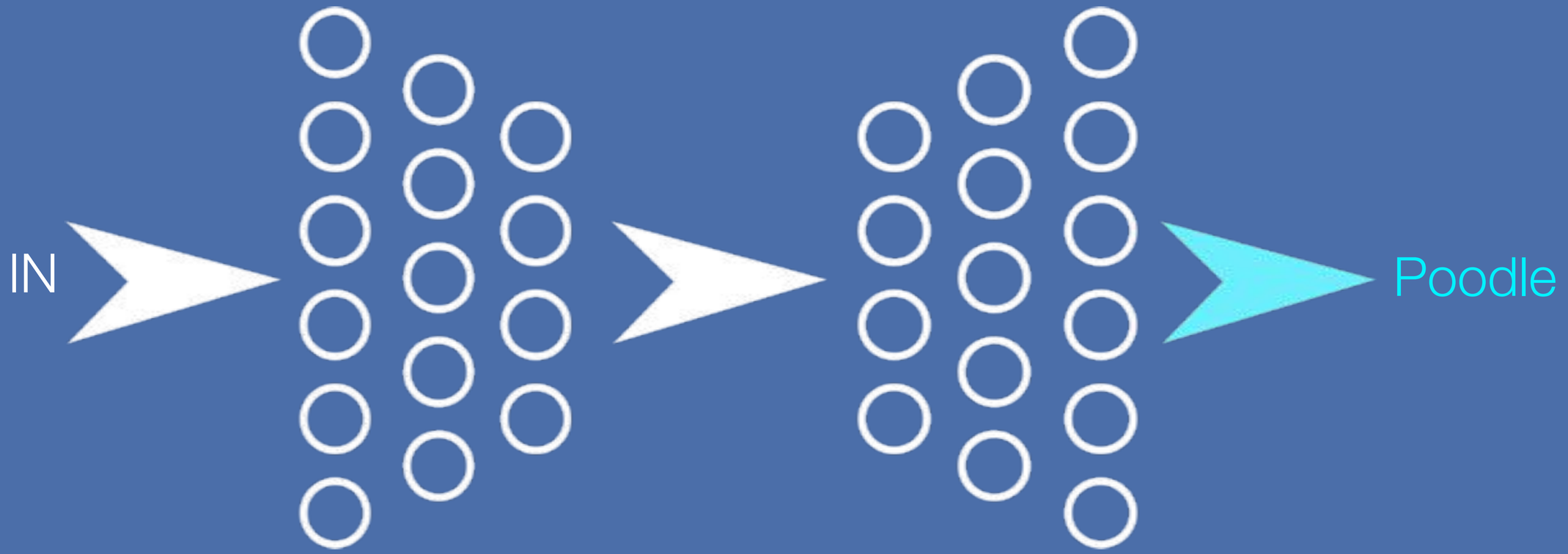
Forward Pass

...



Forward Pass

It’s a Poodle !



Recap

The story so far

- Artificial Neuron
- Neural Network

Recap
The story so far

- Artificial Neuron -> Thresholds its input, one is not enough
- Neural Network

Recap
The story so far

- Artificial Neuron -> Thresholds its input, one is not enough
- Neural Network -> Computes its output with the Forward pass

Learning

”It’s all downhill from here”

- Learning from data
- Stochastic Gradient Descent
- Back Propagation

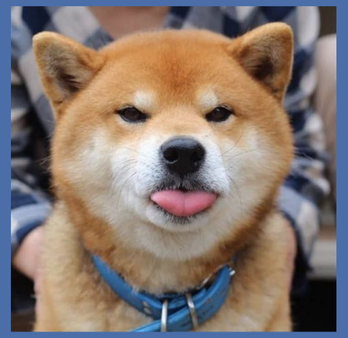
Learning From Data

Step 1 – Get Data

A dataset is a set of (input, output)

Poodle

Shiba Inu



Shiba Inu

Not Dog

Learning From Data

Step 2 – Get More

A dataset is a set of (input, output), lots of them

Loss Function

Define your goal

Loss function

$$\ell(f(x), y)$$

Loss Function

Define your goal

Loss function

Model prediction

$$\ell(f(x), y)$$

Loss Function

Define your goal

Loss function

Model prediction

Ground Truth (or label)

$$\ell(f(x), y)$$

Loss Function

Define your goal

Loss function

Model prediction

Ground Truth (or label)

$$\ell(f(x), y)$$

Poodle

Loss Function

Define your goal

The loss defines how
much you pay
for answering ”Shiba”
instead of ”Poodle”

Loss function

Model prediction

Ground Truth (or label)

$$\ell(f(x), y)$$

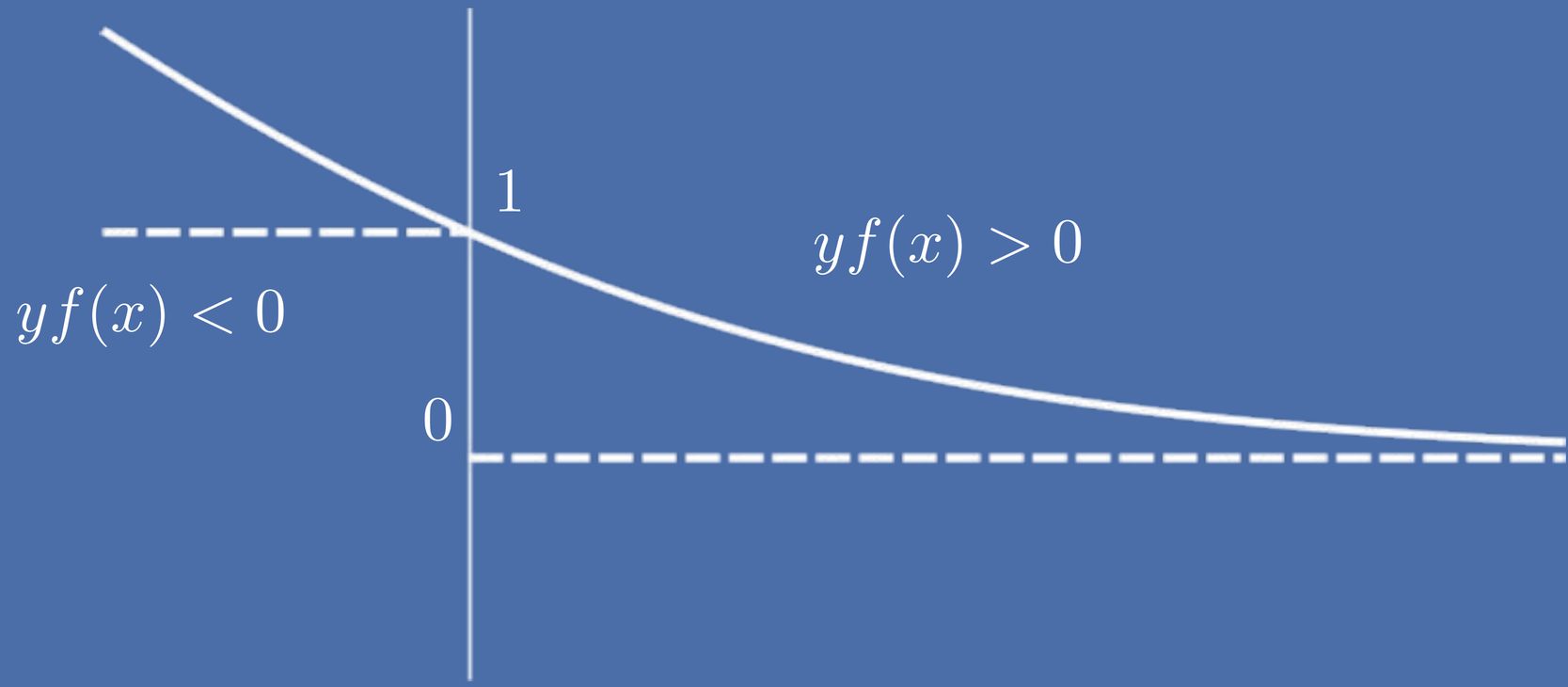
Poodle

Loss Function

Some examples

Classification

- Goal : assign class +1 / -1
- Loss : based on sign agreement $yf(x)$

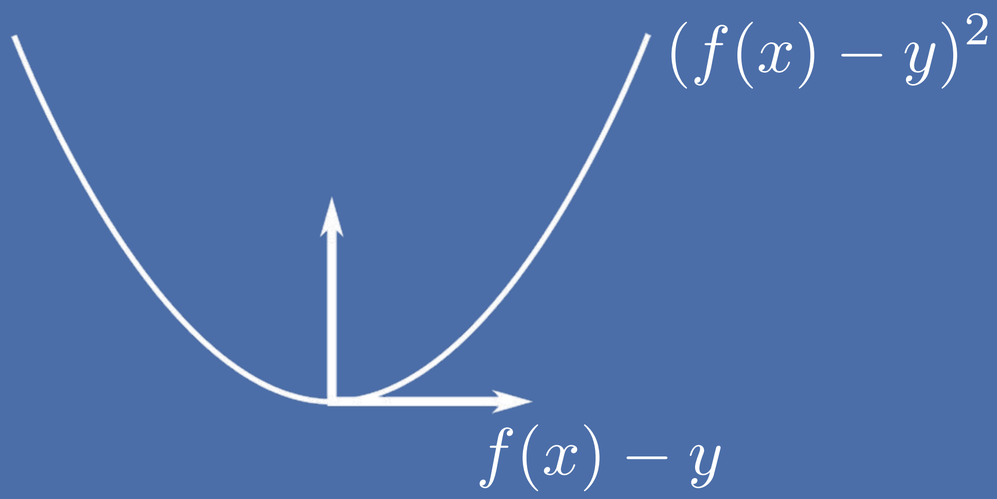


Loss Function

Some examples

Regression

- Goal : learn continuous value y
- Loss : based on distance $(f(x) - y)^2$



Loss : Application-dependent measure of error

Training Loss

Let's get to minimizing

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

Training Loss

Let's get to minimizing

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

Model Parameters

Training Loss

Let's get to minimizing

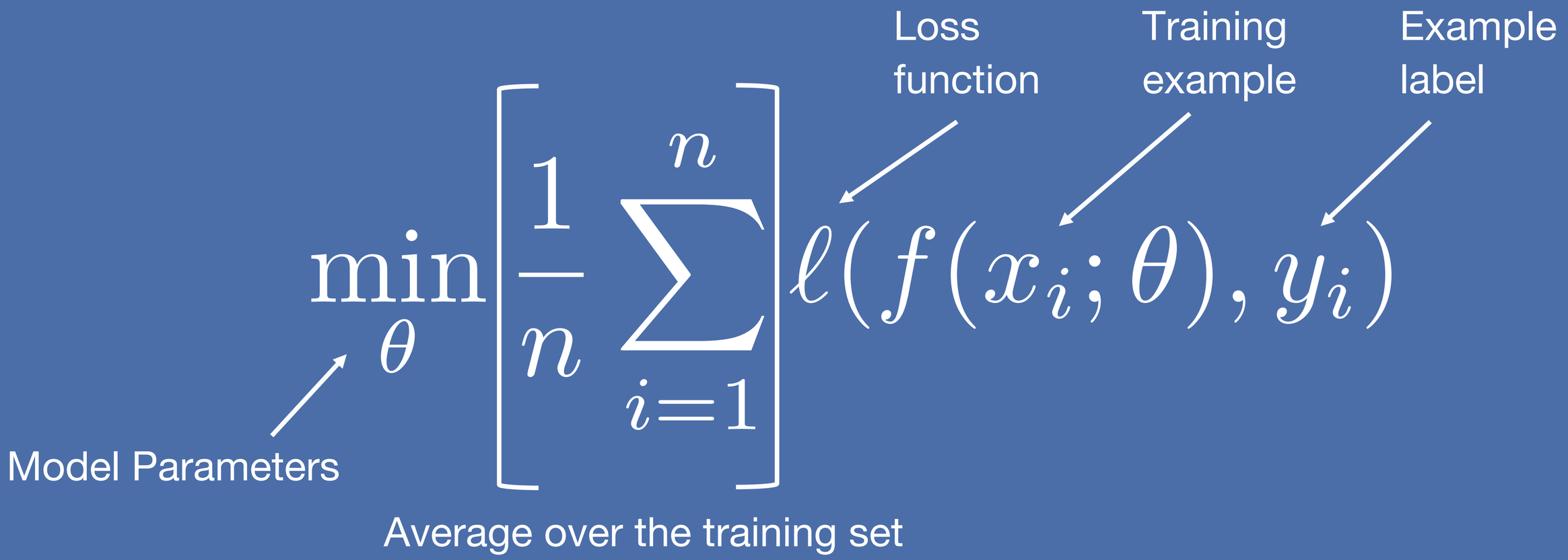
$$\min_{\theta} \left[\frac{1}{n} \sum_{i=1}^{n} \right] \ell(f(x_i; \theta), y_i)$$

Model Parameters

Average over the training set

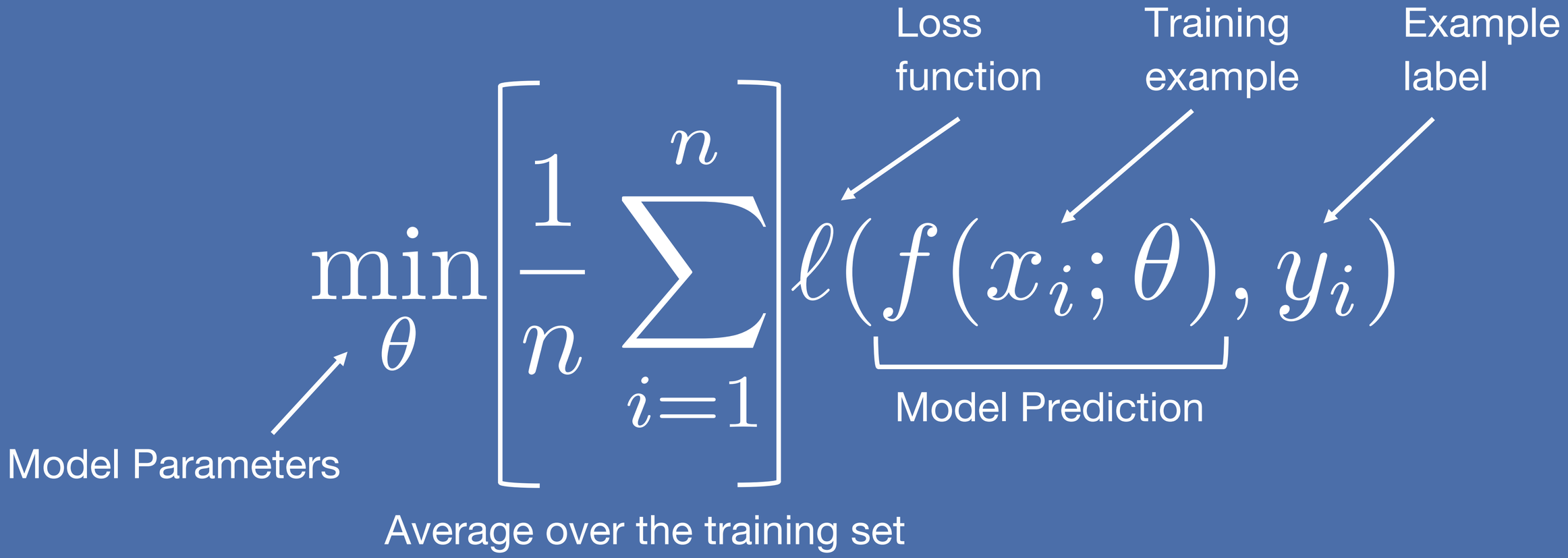
Training Loss

Let's get to minimizing



Training Loss

Let's get to minimizing



Training Loss

Let's get to minimizing

Loss function

poodle

$$\min_{\theta} \left[\frac{1}{n} \sum_{i=1}^{n} \right] \ell\big(f(x_i;\theta), y_i\big)$$

Model Parameters

Average over the training set

Training Loss

Let's get to minimizing

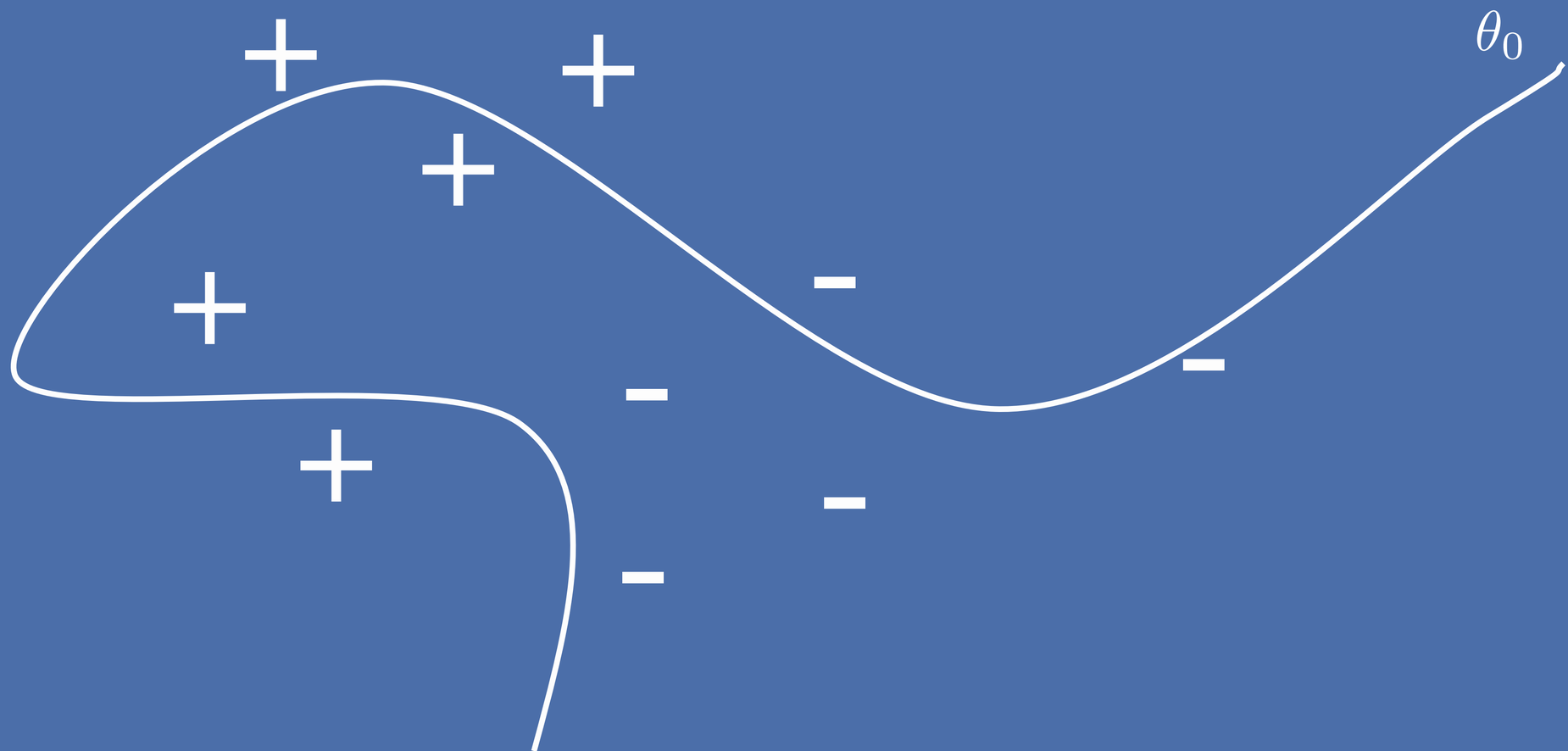
Find the parameters that minimize the average loss on the training set

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

An iterative process

Step 1 – be random

Your neural net start with random parameters θ_0

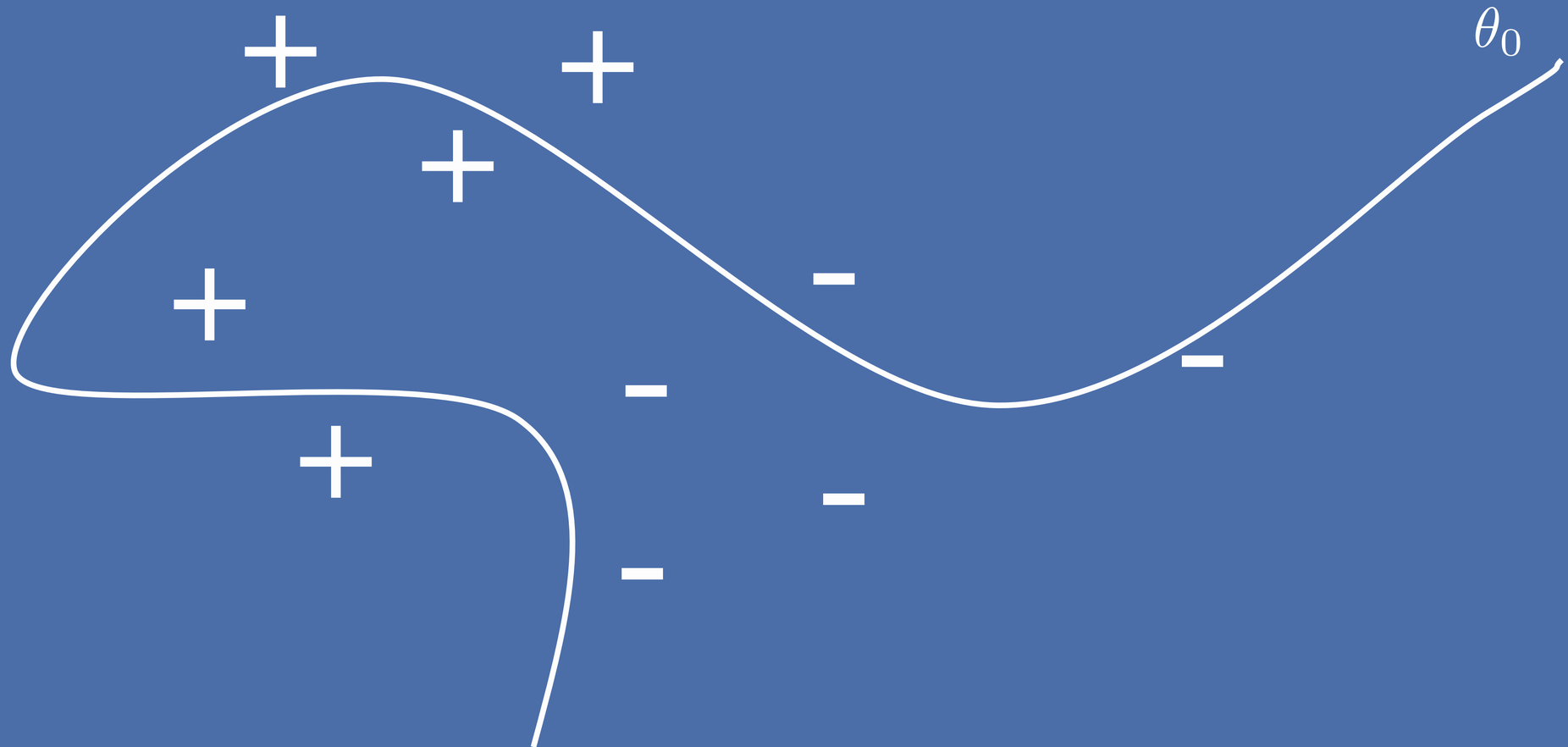


An iterative process

Step 1 – be random

$$\frac{1}{n}\sum_{i=1}^{n}\ell(f(x_i;\theta),y_i)$$

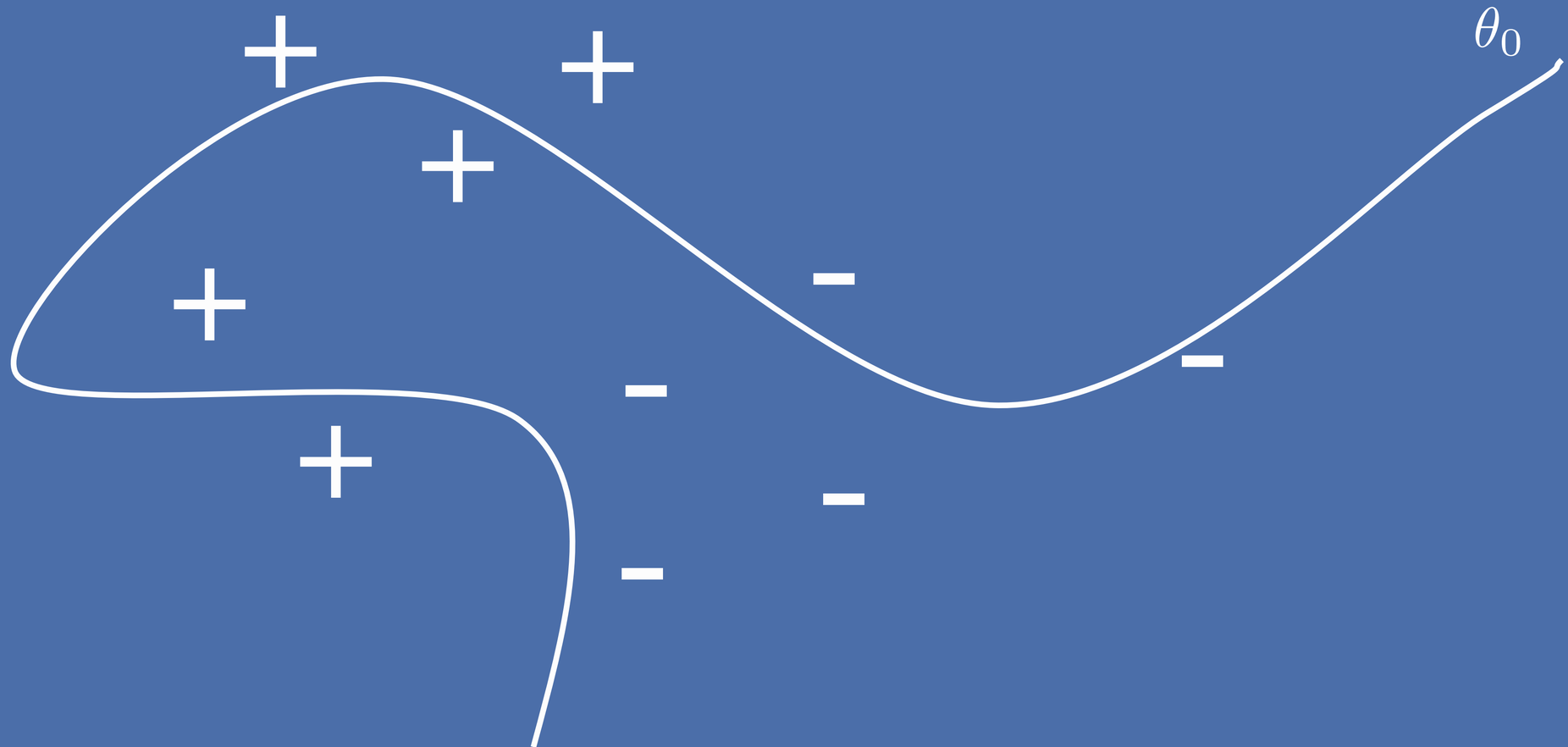
Your neural net start with random parameters θ_0



An iterative process

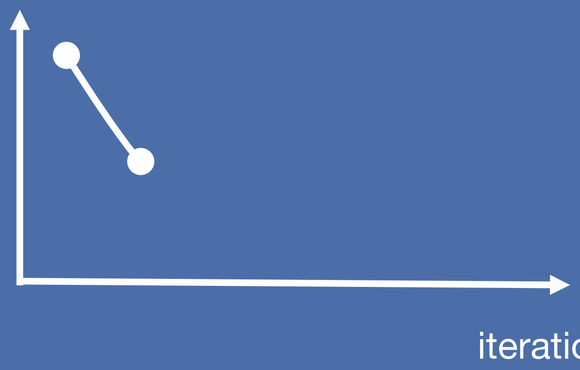
Step 1 – be random

Your neural net start with random parameters θ_0

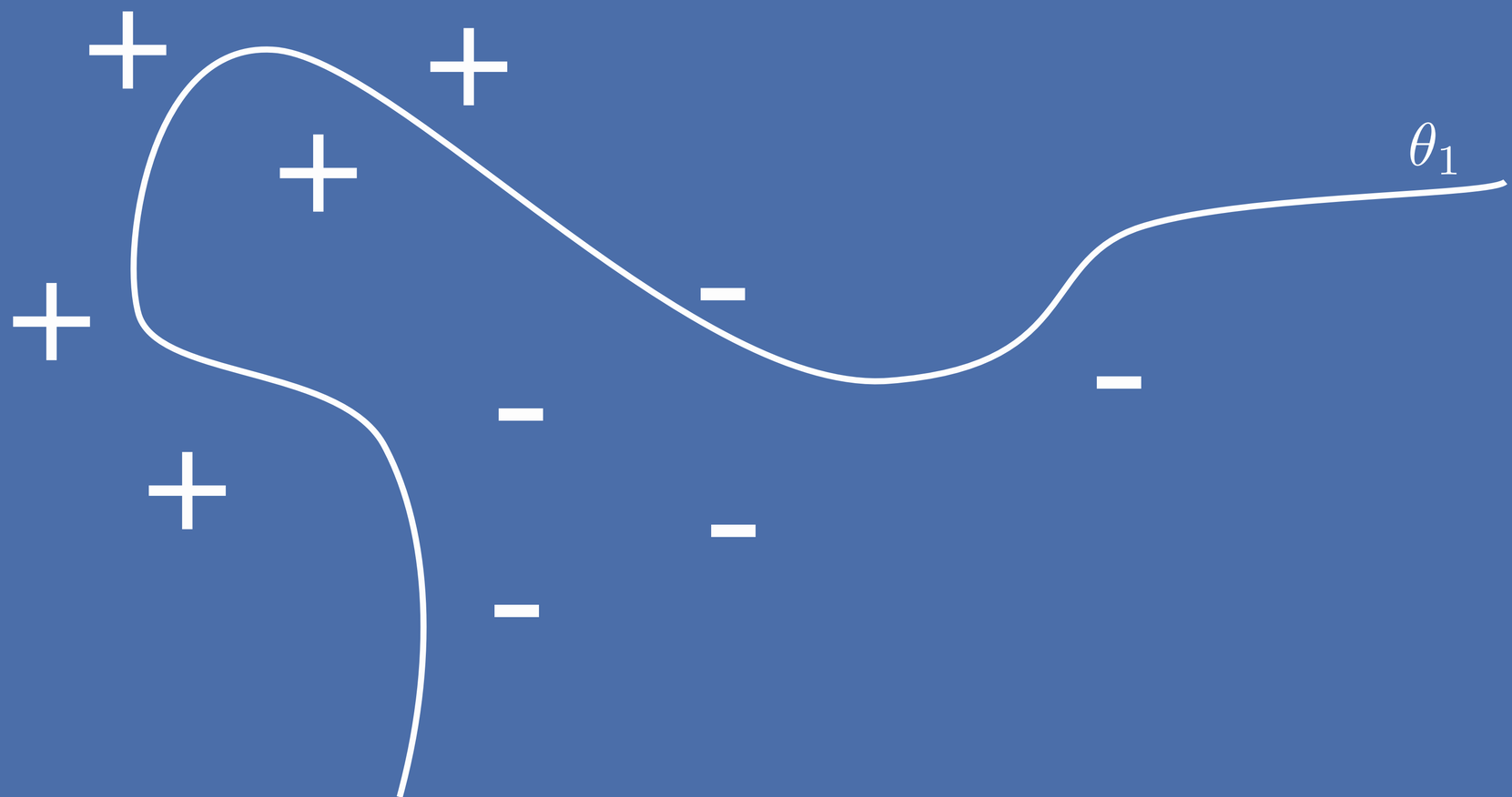


An iterative process

Step 2 – be less random



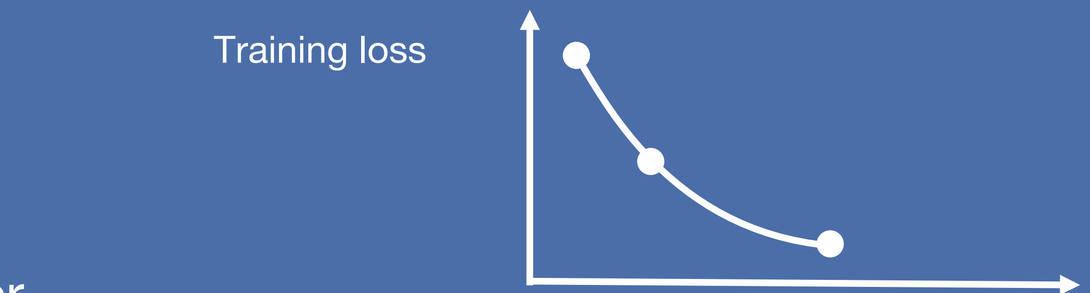
Then you'll find θ_1 that performs a bit better



An iterative process

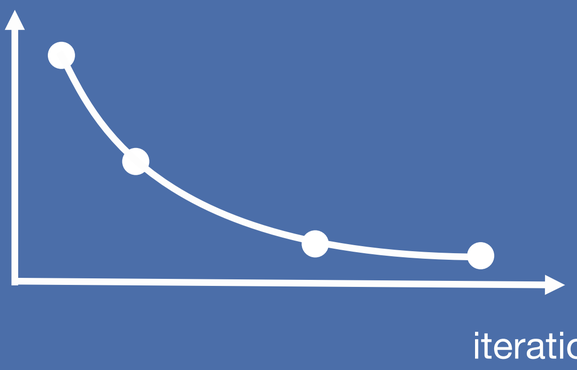
Step 3 – be even less random

Then you'll find θ_2 that performs even better

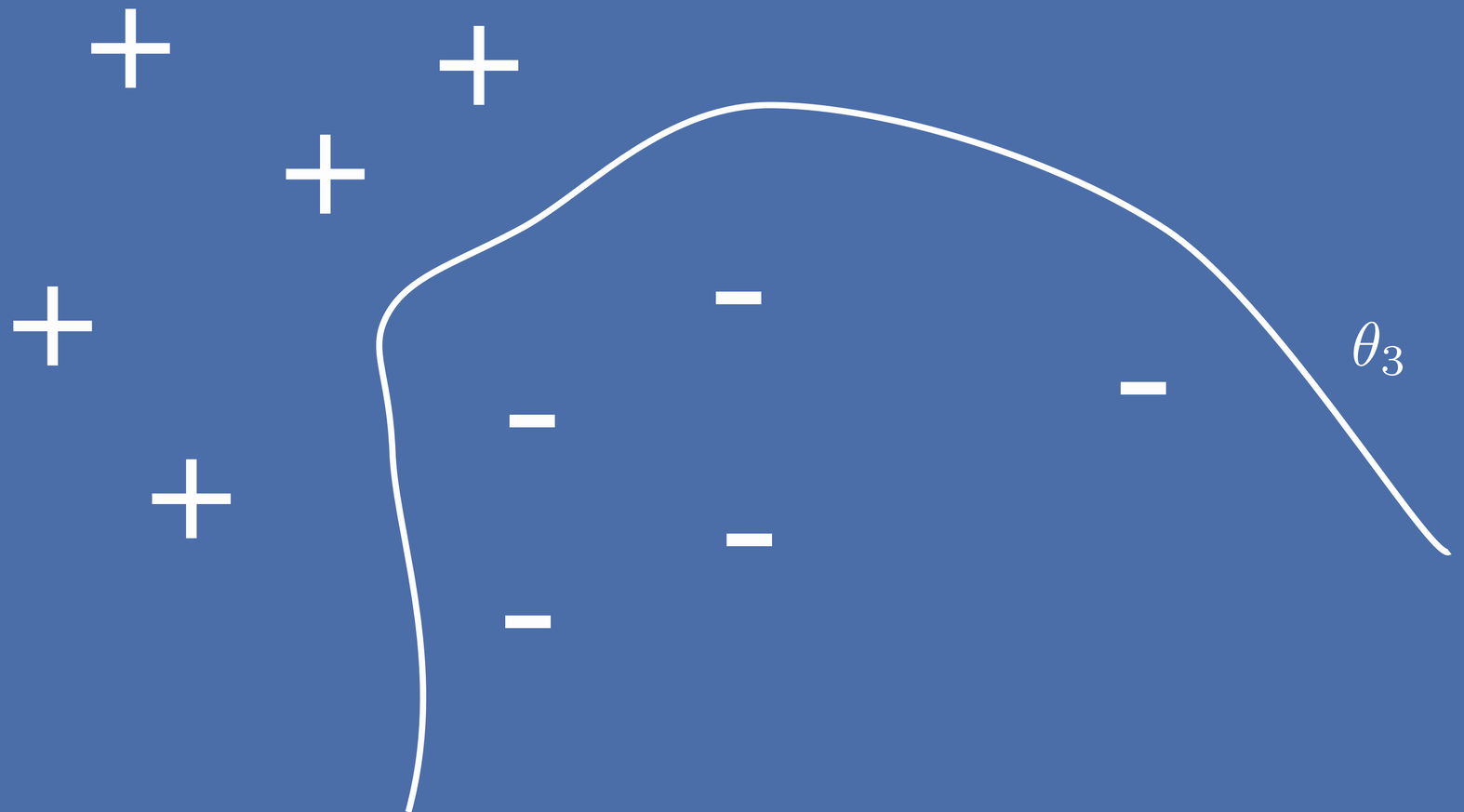


An iterative process

Step 4 – profit



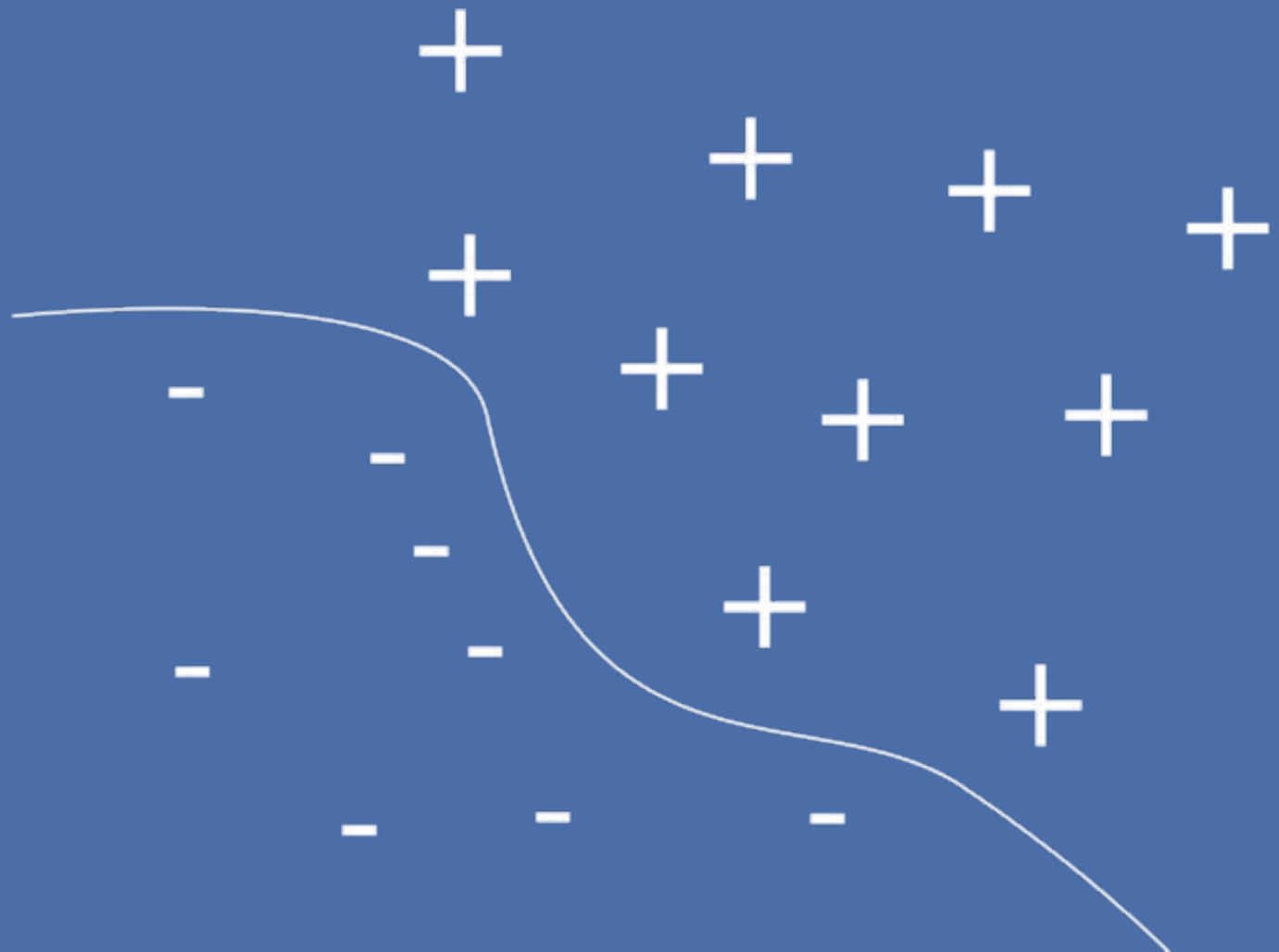
And finally a θ_3



Learning From Data

Avoiding Overfitting

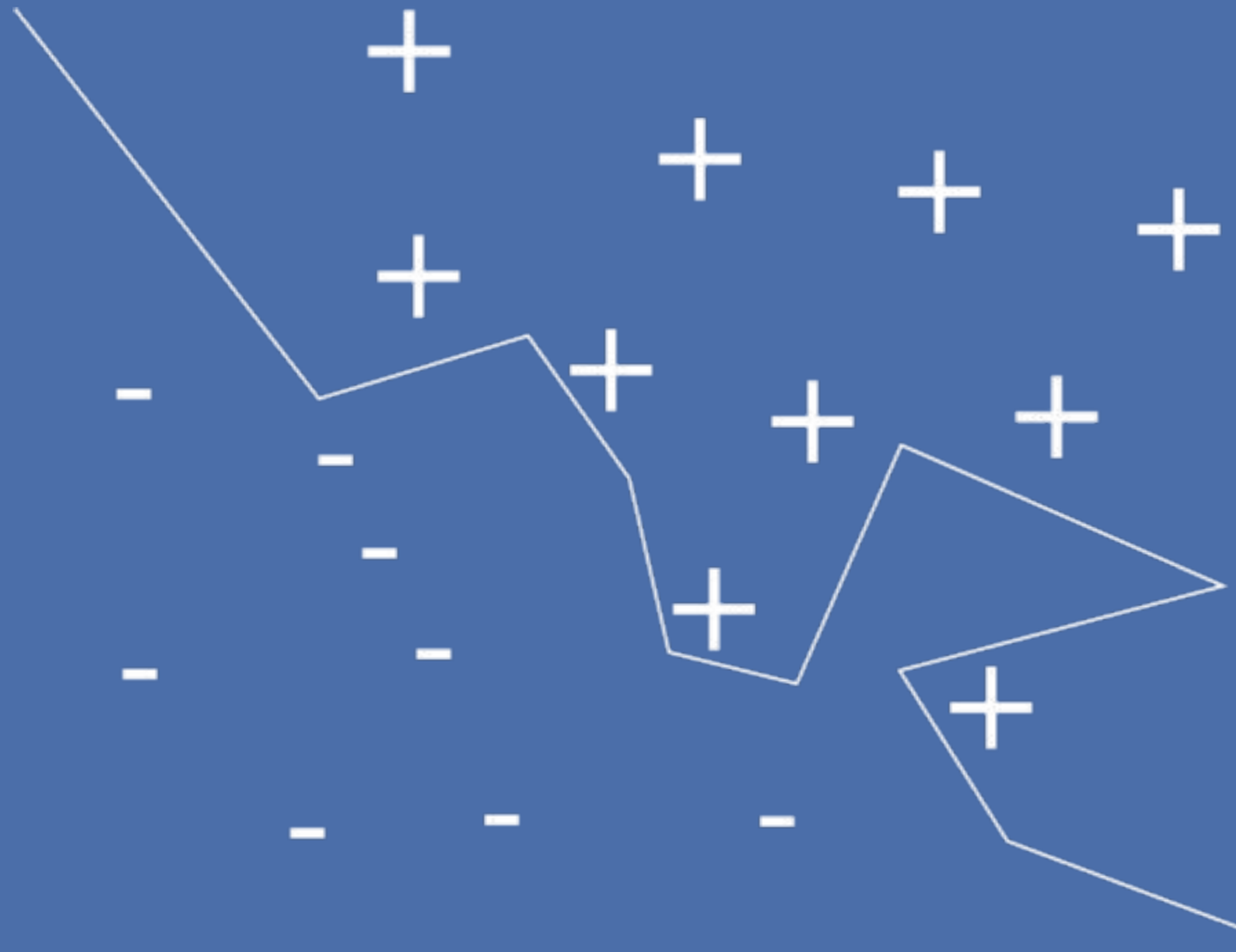
This classifier has zero error



Learning From Data

Avoiding Overfitting

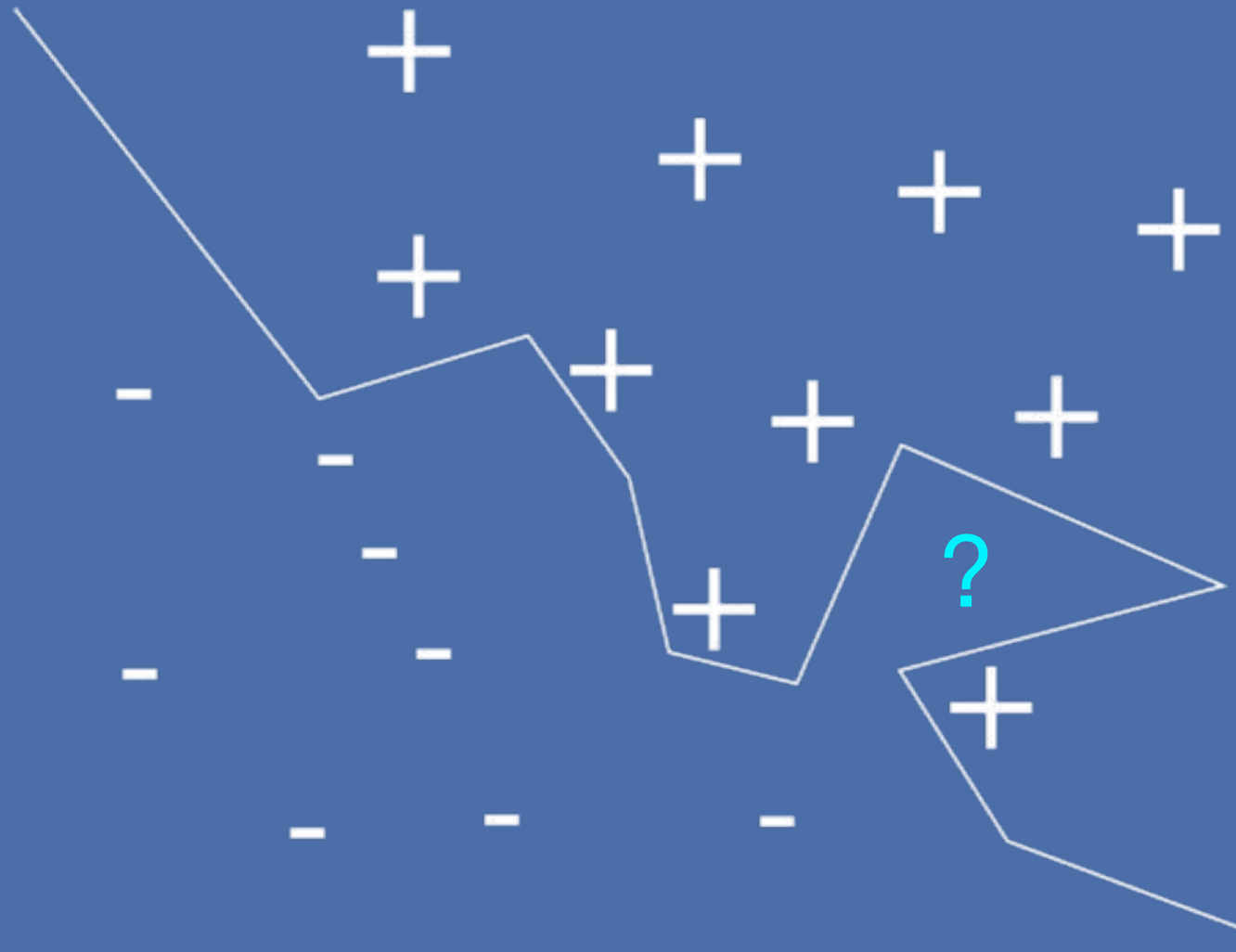
This one too. How do we pick one ?



Learning From Data

Avoiding Overfitting

This one too. How do we pick one ?

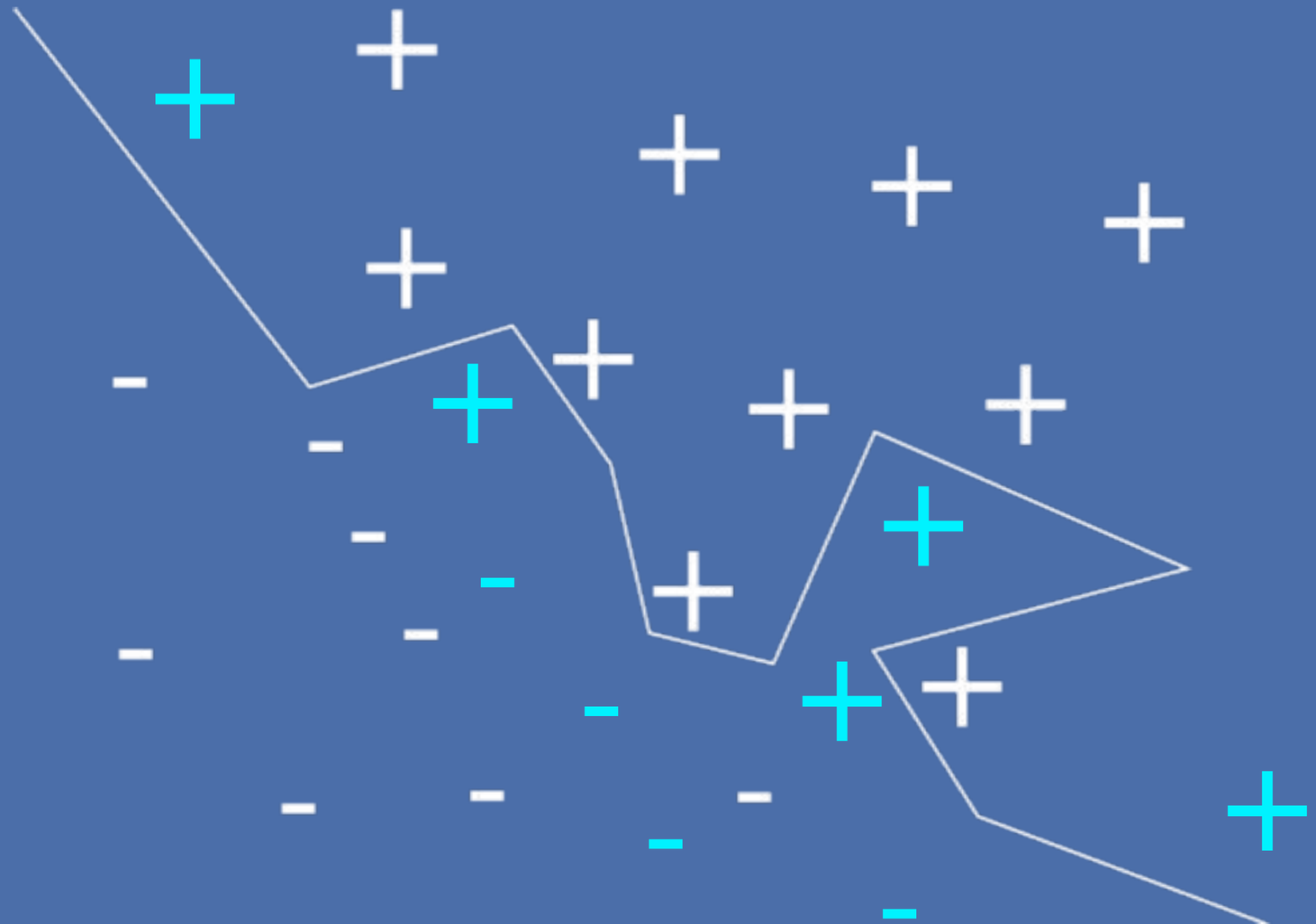


Learning From Data

Avoiding Overfitting

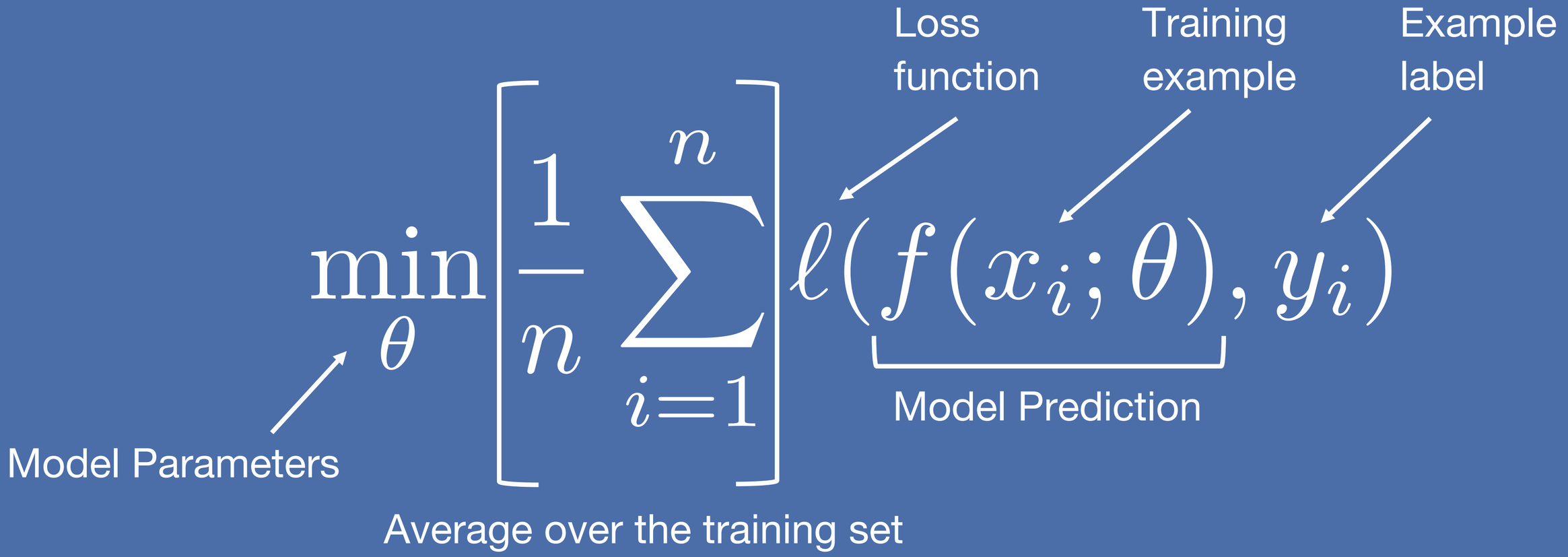
This one too. How do we pick one ?

Solution : Training and Validation Set



Gradient Descent

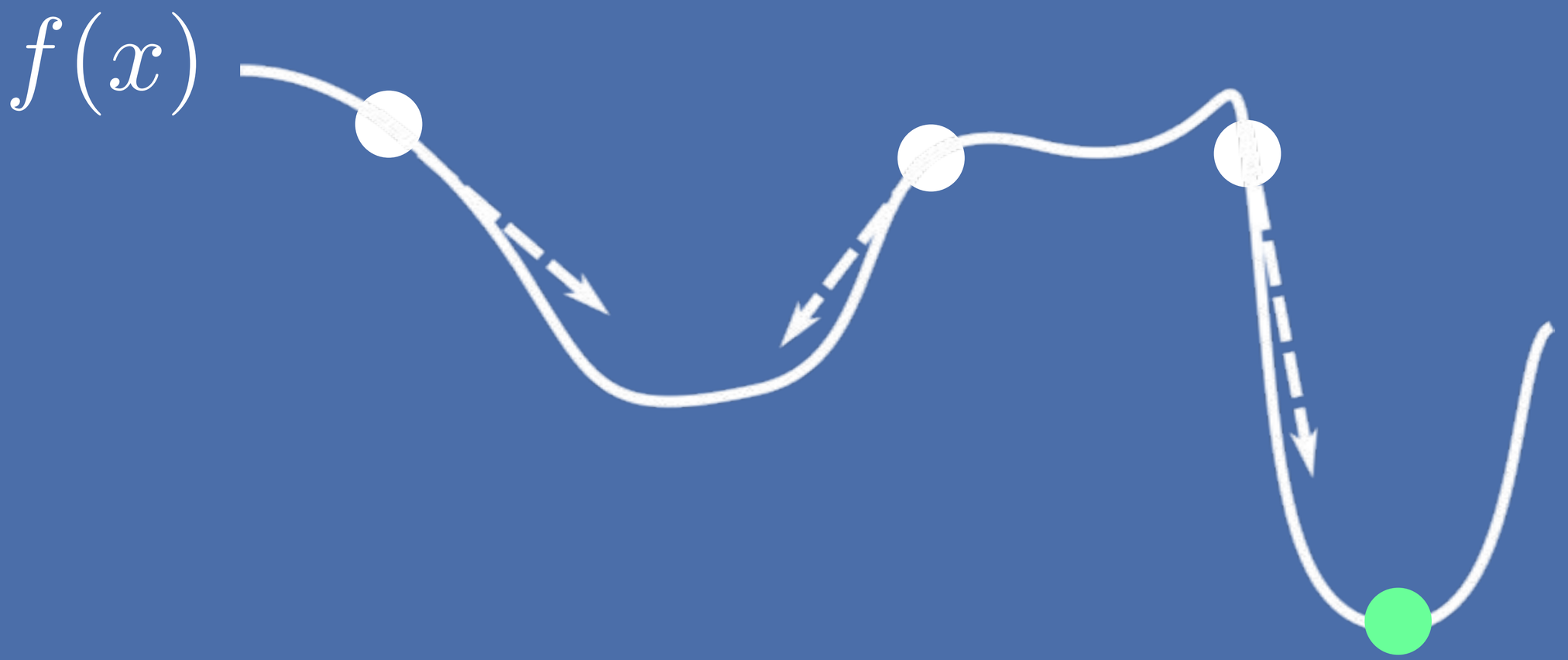
Back to minimizing



Gradient Descent

Back to minimizing

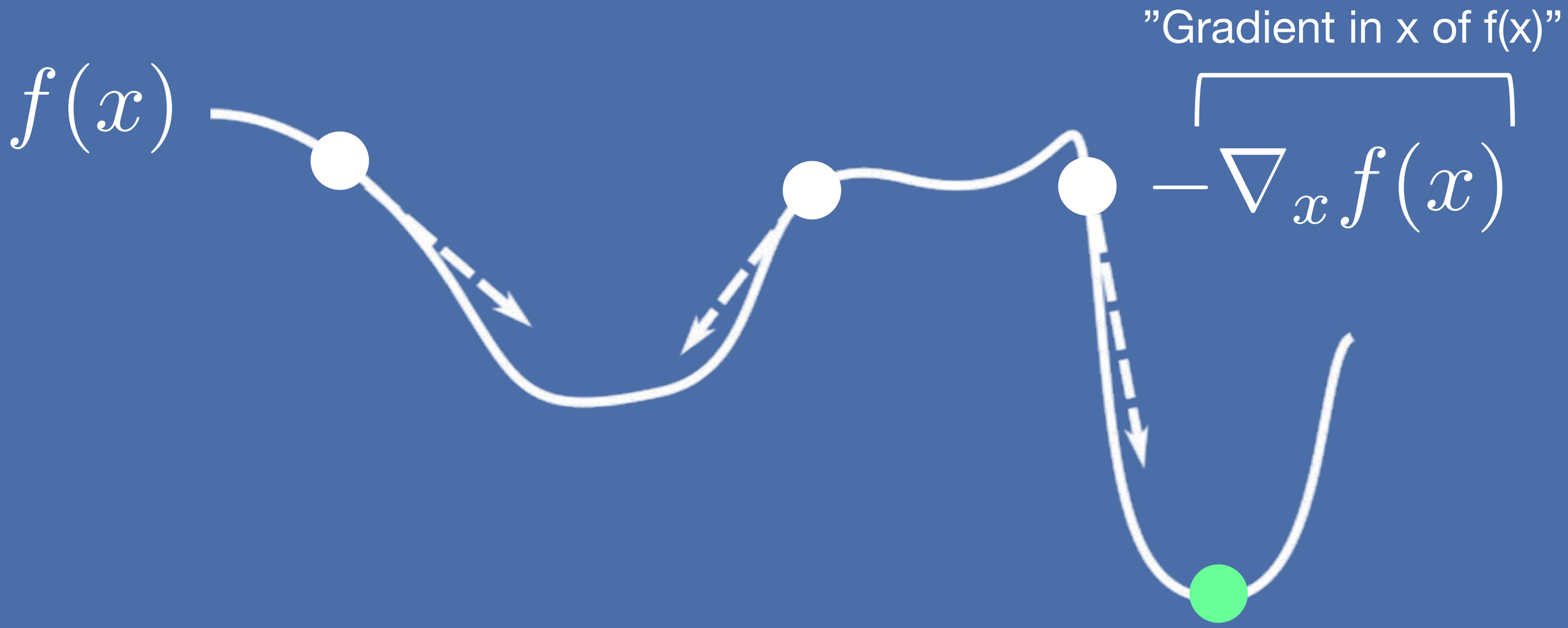
Skier’s approach to minimization : Steepest Descent



Gradient Descent

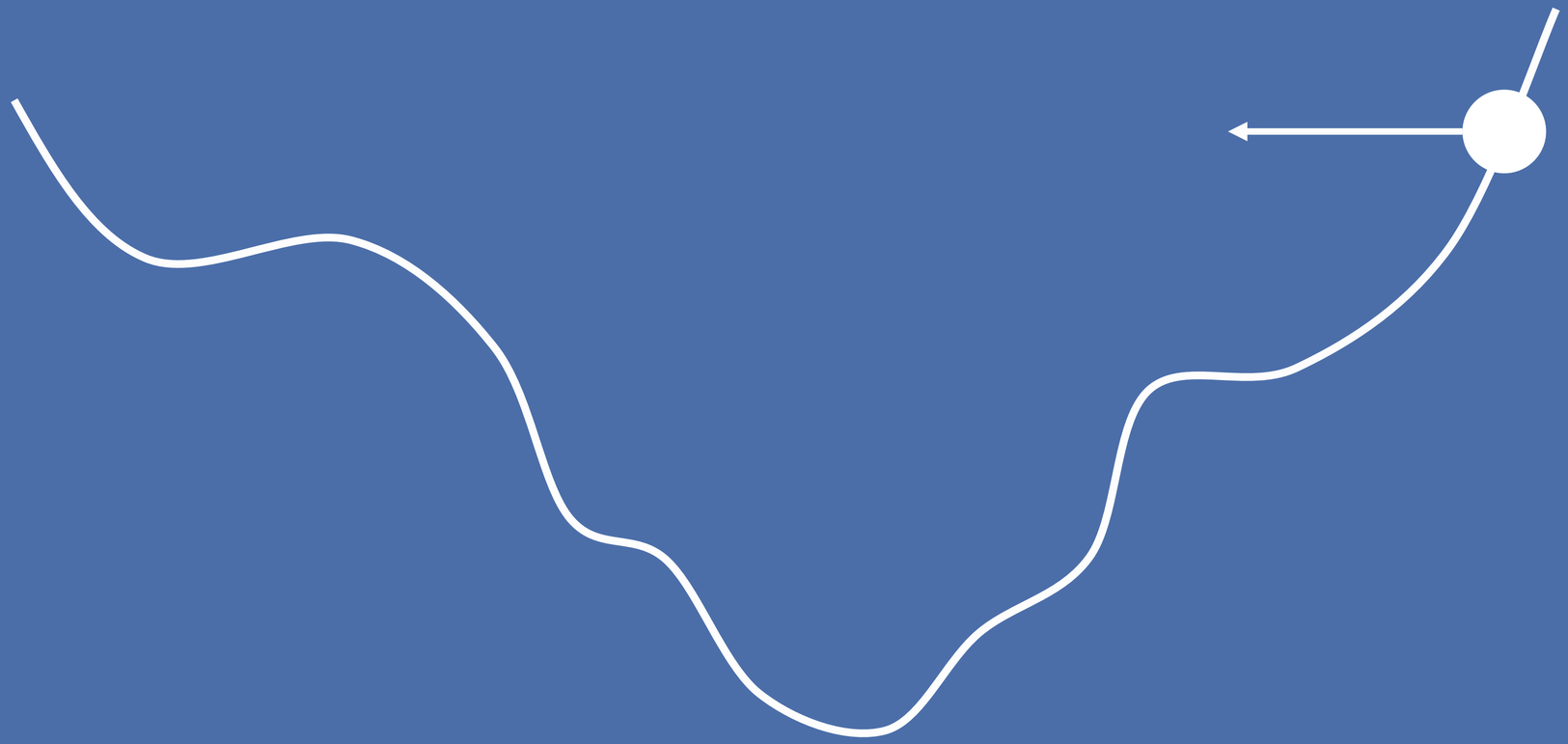
Back to minimizing

Skier’s approach to minimization : Steepest Descent



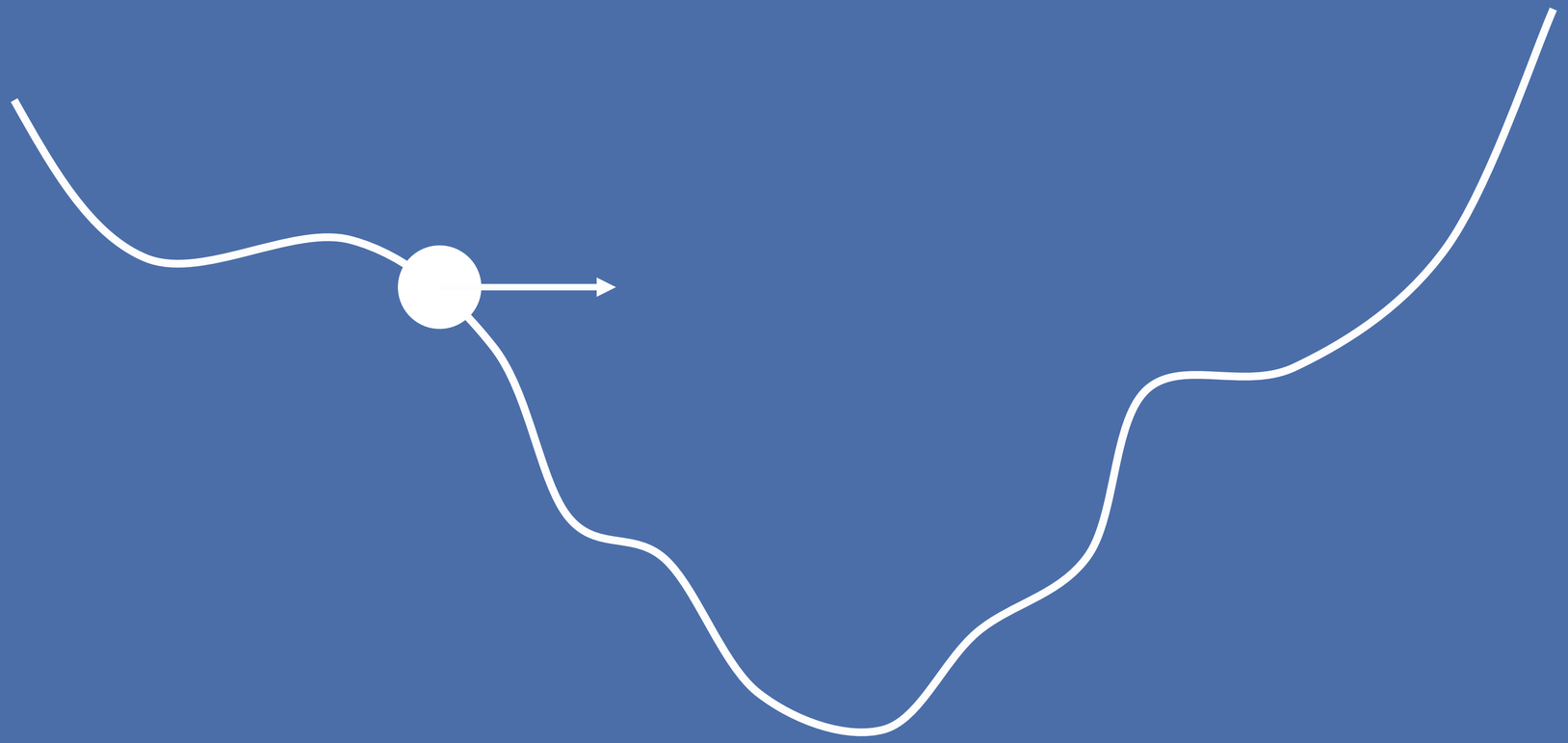
Gradient Descent

Back to minimizing



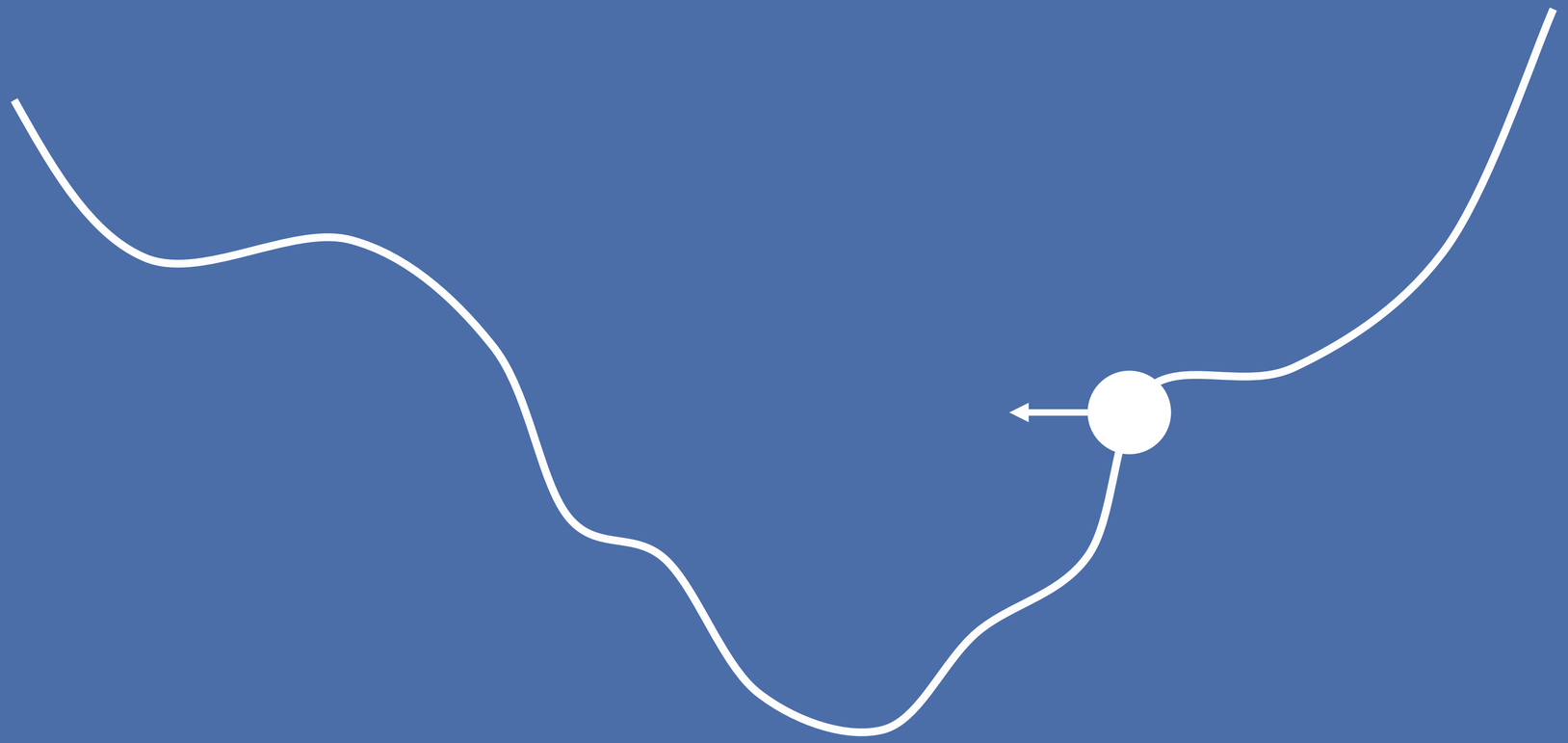
Gradient Descent

Back to minimizing



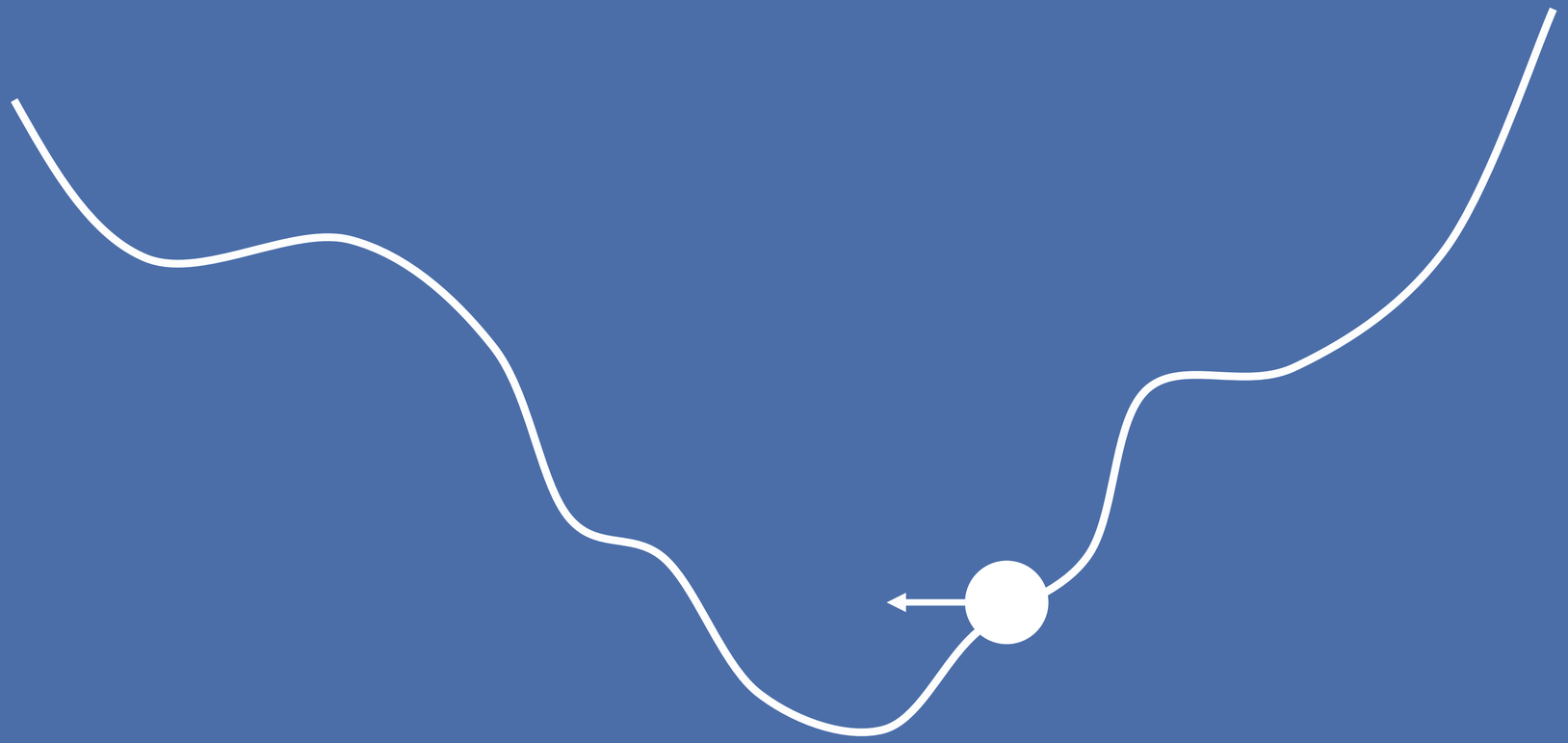
Gradient Descent

Back to minimizing



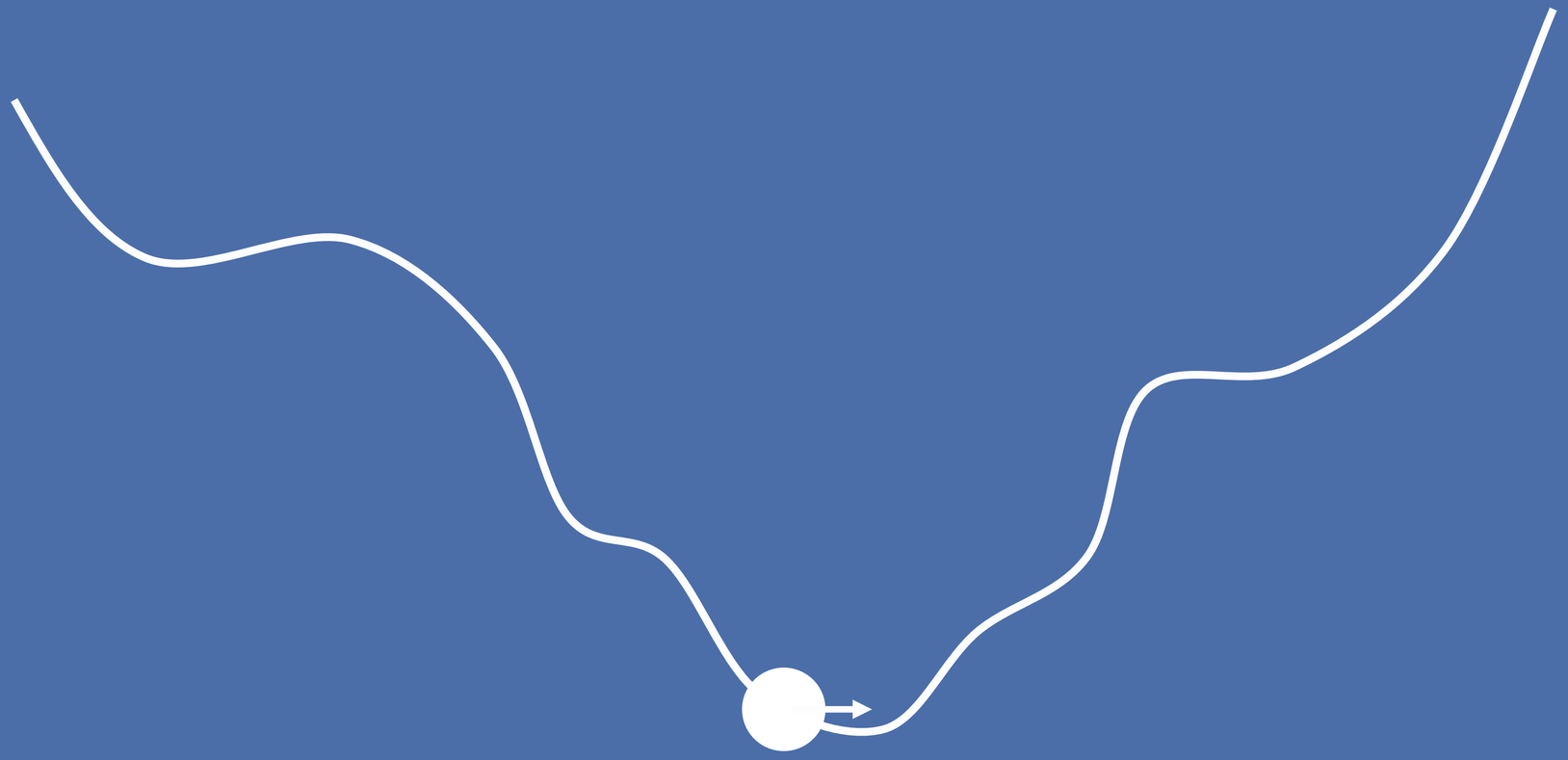
Gradient Descent

Back to minimizing



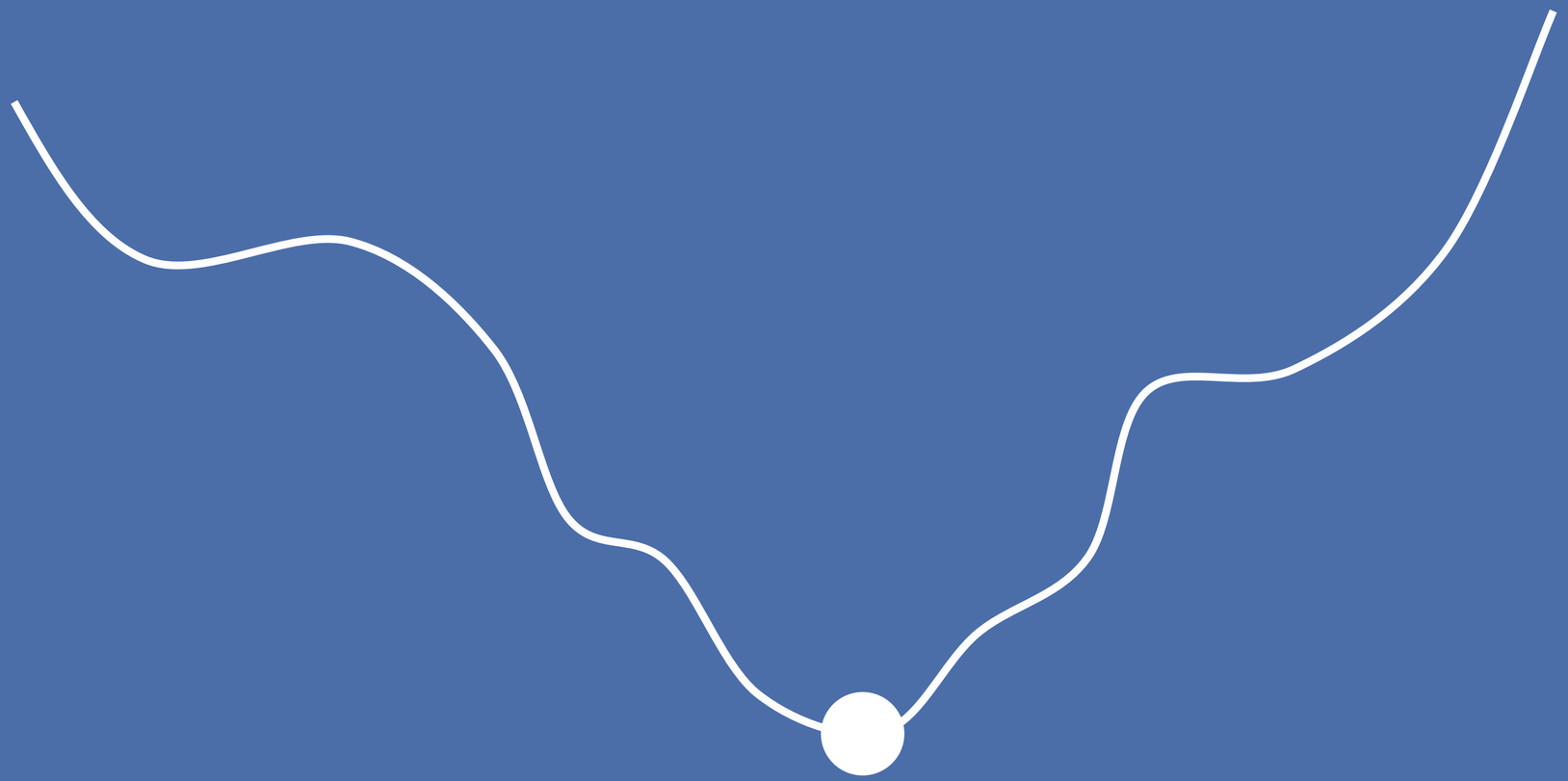
Gradient Descent

Back to minimizing



Gradient Descent

Back to minimizing



Gradient Descent

Back to minimizing

Problem : How do we compute the gradient ?

Large

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

Complicated function (neural net)

Gradient Descent

Back to minimizing

Problem : How do we compute the gradient ?

Large -> Stochastic Gradient Descent

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

Complicated function (neural net) -> BackProp

Gradient Descent

Back to minimizing

Problem : How do we compute the gradient ?

Large -> Stochastic Gradient Descent

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

Stochastic Gradient Descent

Killing n

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \underbrace{\ell(f(x_i;\theta), y_i)}_{\text{One function}}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(f(x_i;\theta), y_i)$$

$$\approx \nabla_\theta \ell(f(x_j;\theta), y_j)$$

Stochastic Gradient Descent

Killing n

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i;\theta), y_i)$$

The Gradient of the Average

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(f(x_i;\theta), y_i)$$

$$\approx \nabla_\theta \ell(f(x_j;\theta), y_j)$$

Stochastic Gradient Descent

Killing n

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i;\theta), y_i)$$

The Gradient of the Average

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(f(x_i;\theta), y_i)$$

= The Average of the Gradients

$$\approx \nabla_\theta \ell(f(x_j;\theta), y_j)$$

Stochastic Gradient Descent

Killing n

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i;\theta), y_i)$$

The Gradient of the Average

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(f(x_i;\theta), y_i)$$

= The Average of the Gradients

$$\approx \nabla_\theta \ell(f(x_j;\theta), y_j)$$

In expectation, for uniform j

Stochastic Gradient Descent

Killing n

For some number of iterations :

Pick some random example (x_j, y_j)

$$\theta_{n+1} \leftarrow \theta_n - \eta \nabla_\theta \ell(f(x_j; \theta_n), y_j)$$

Gradient Step

Learning-rate

Back Propagation

Computing the gradient

Problem : How do we compute the gradient ?

$$\nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \theta), y_i)$$

Complicated function (neural net) -> BackProp

BackProp
Computing the gradient

Problem : How do we compute the gradient ?

$$f_i(x) = \sigma(A_i x + b_i) \longleftarrow$$ Hidden Layer i

BackProp
Computing the gradient

Problem : How do we compute the gradient ?

$$f_i(x) = \sigma(A_i x + b_i)$$ ← Hidden Layer i

$$f = f_h(f_{h-1}(f_{h-2}(\ldots))) = (f_h \circ f_{h-1} \circ \ldots \circ f_1)(x)$$

Complete Neural Network

BackProp
Computing the gradient

Problem : How do we compute the gradient ?

$$f_i(x) = \sigma(A_i x + b_i)$$ ← Hidden Layer i

$$f = f_h(f_{h-1}(f_{h-2}(\ldots))) = (f_h \circ f_{h-1} \circ \ldots \circ f_1)(x)$$

Complete Neural Network

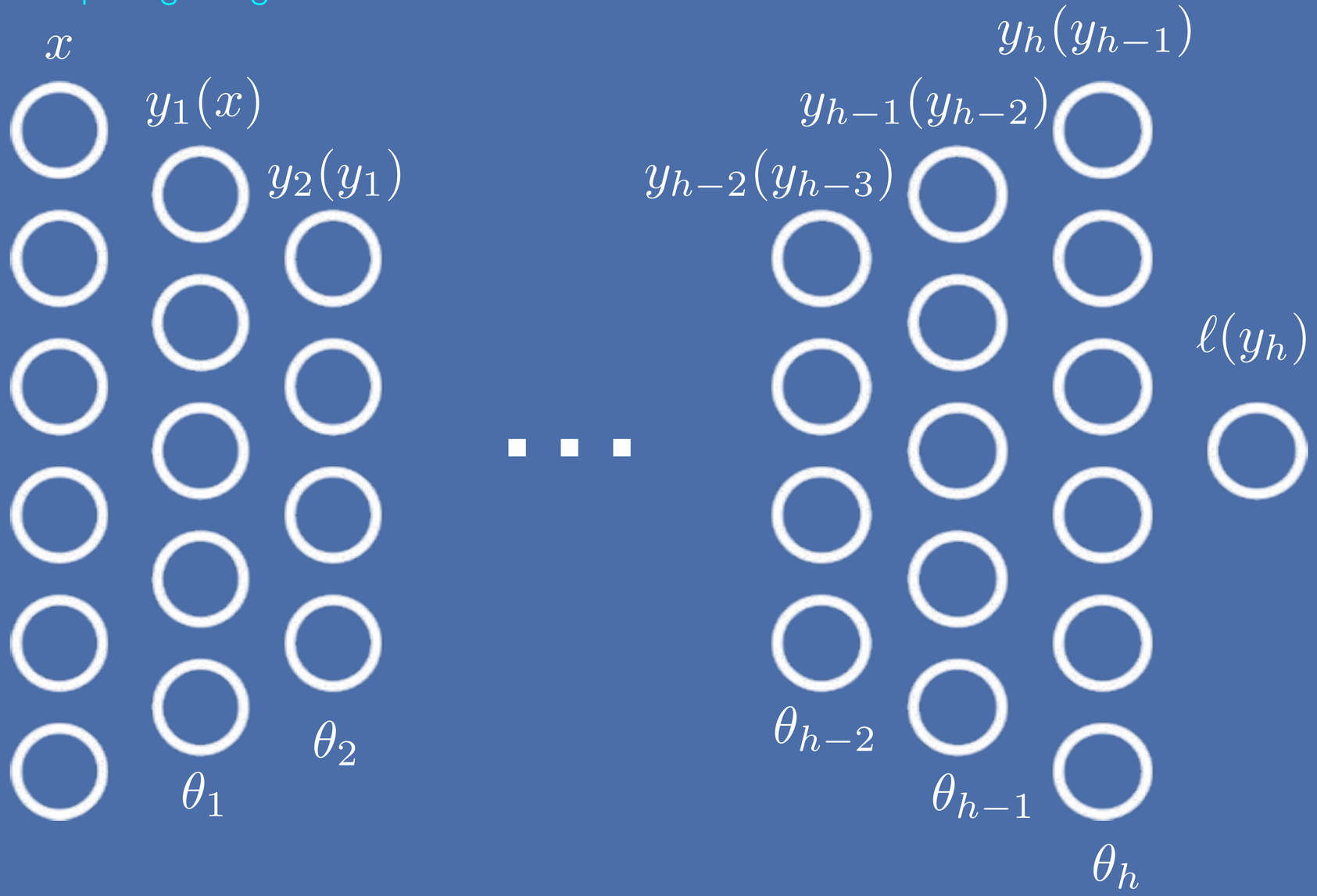
$$\nabla_\theta \ell(f(x_i;\theta), y_i) \; ??$$

BackProp
Computing the gradient

Chain-rule : $$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial x}$$

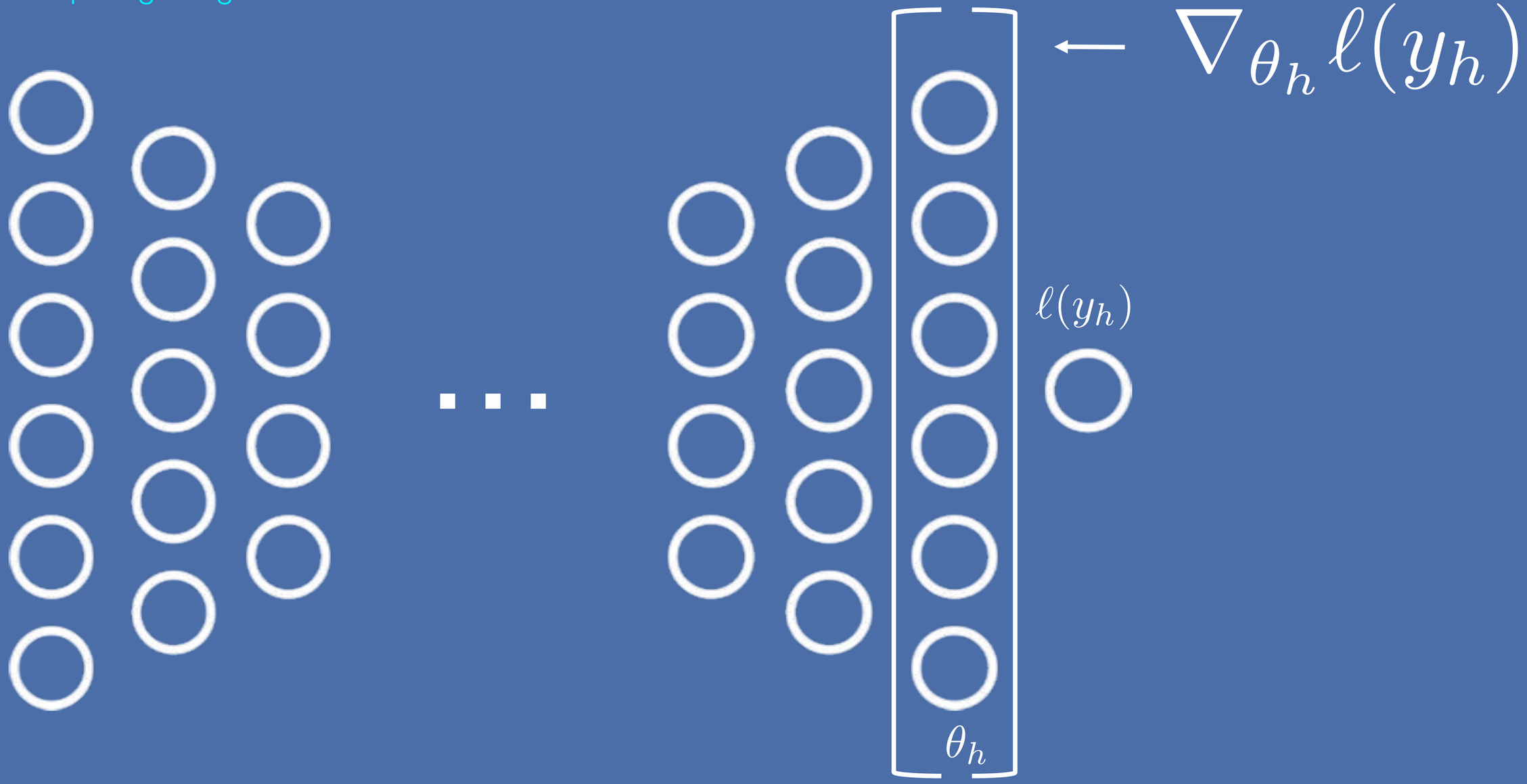
BackProp

Computing the gradient



BackProp

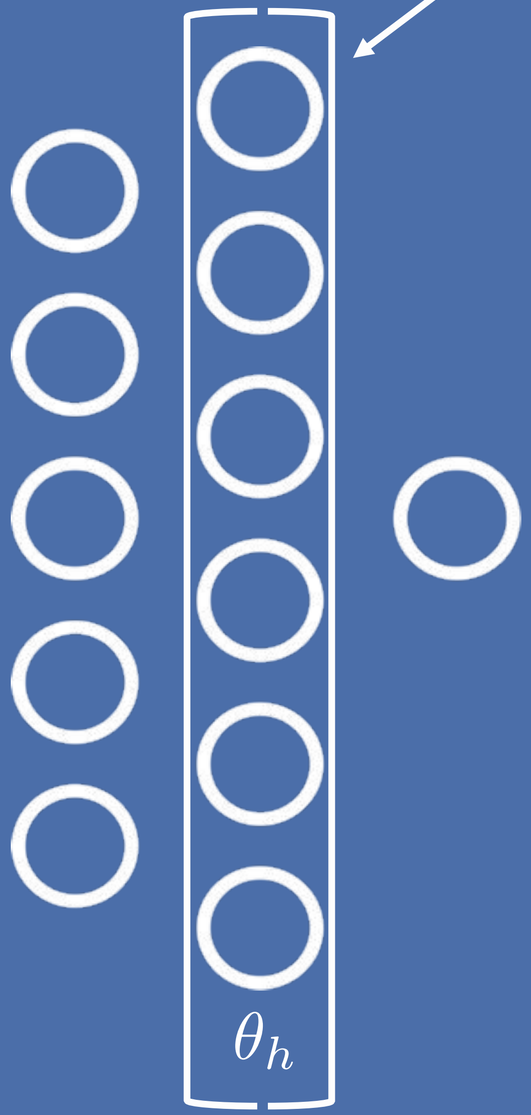
Computing the gradient



BackProp

Computing the gradient

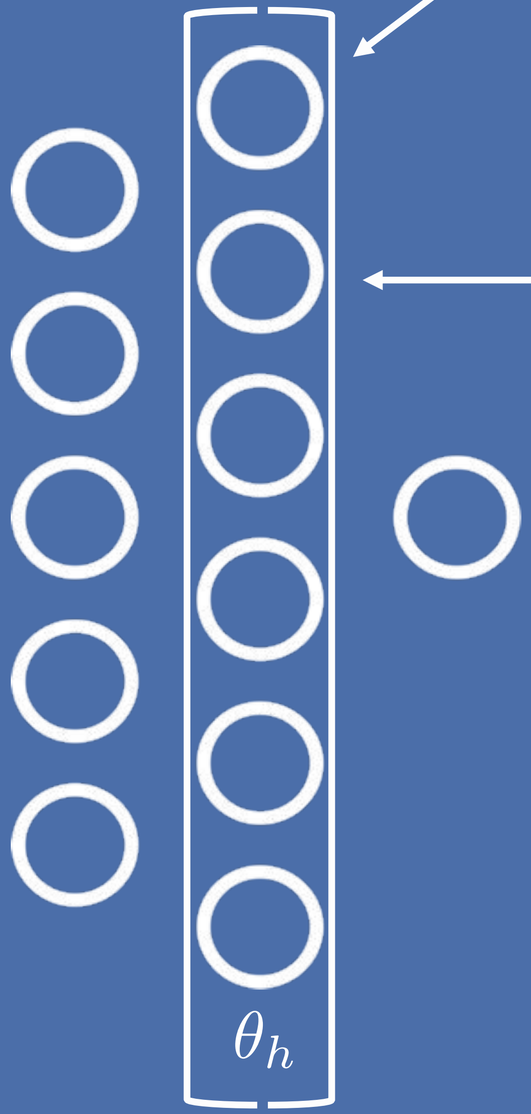
$$\nabla_{\theta_h} \ell(y_h)$$



BackProp

Computing the gradient

$$\nabla_{\theta_h} \ell(y_h)$$

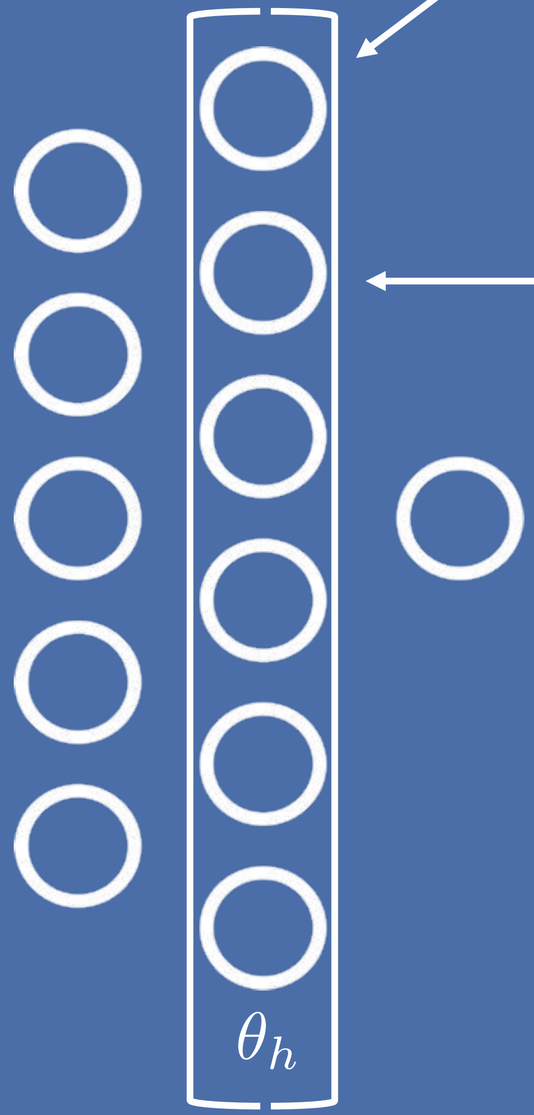


$$\frac{\partial \ell(y_h)}{\partial \theta_{h,i}} = \underbrace{\frac{\partial \ell(y_h)}{\partial y_h} \frac{\partial y_h}{\partial \theta_{h,i}}}_{\text{Chain-Rule}}$$

BackProp

Computing the gradient

$$\nabla_{\theta_h} \ell(y_h)$$



$$\frac{\partial \ell(y_h)}{\partial \theta_{h,i}} = \frac{\partial \ell(y_h)}{\partial y_h} \frac{\partial y_h}{\partial \theta_{h,i}}$$

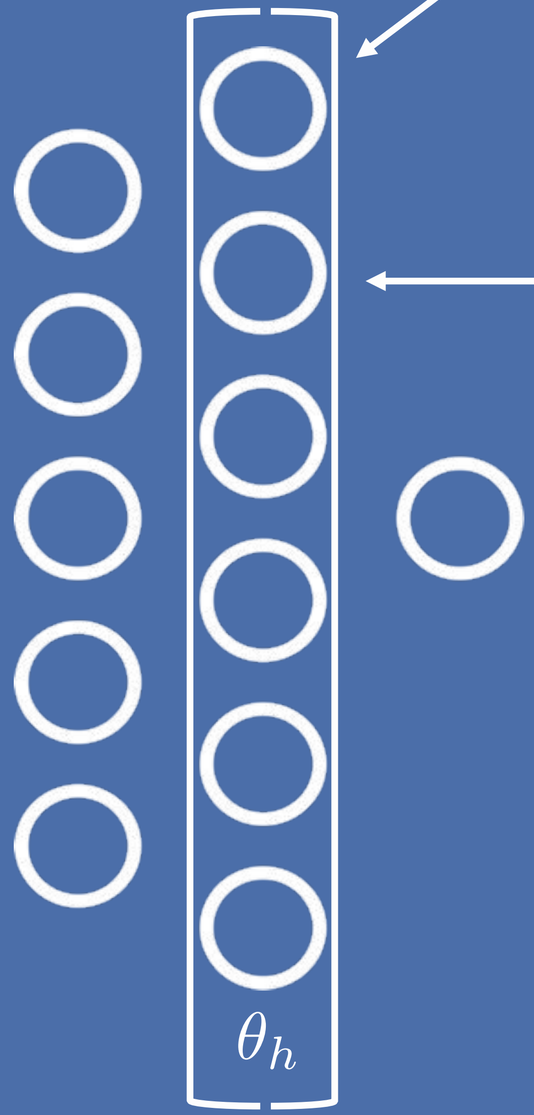
Doesn't depend on current layer

Only depends on ℓ

BackProp

Computing the gradient

$$\nabla_{\theta_h} \ell(y_h)$$



$$\frac{\partial \ell(y_h)}{\partial \theta_{h,i}} = \frac{\partial \ell(y_h)}{\partial y_h} \frac{\partial y_h}{\partial \theta_{h,i}}$$

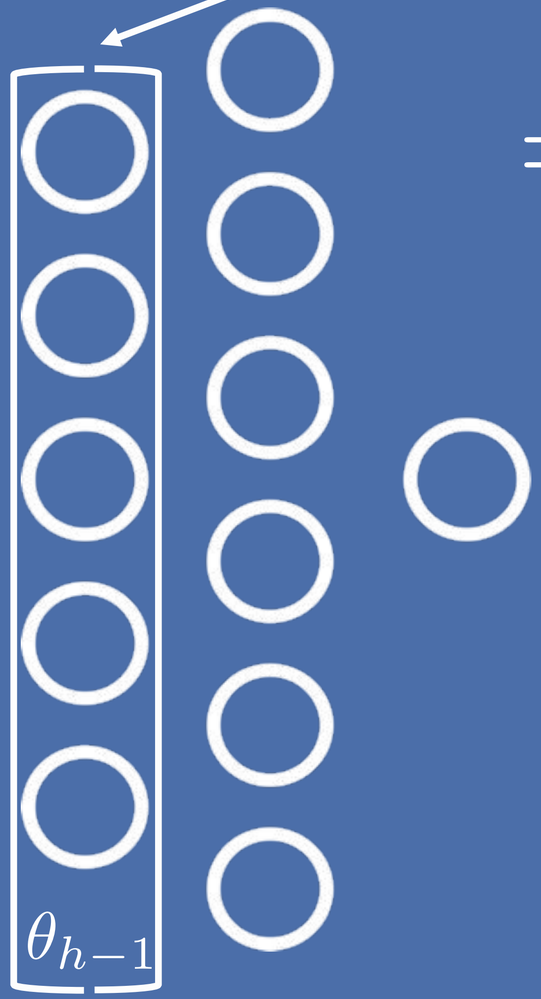
Only depends on current layer

BackProp

Computing the gradient

$$\nabla_{\theta_{h-1}} \ell(y_h)$$

$$= \Phi_{h-1}(\theta_{h-1}, \nabla_{y_{h-1}} \ell(y_h))$$



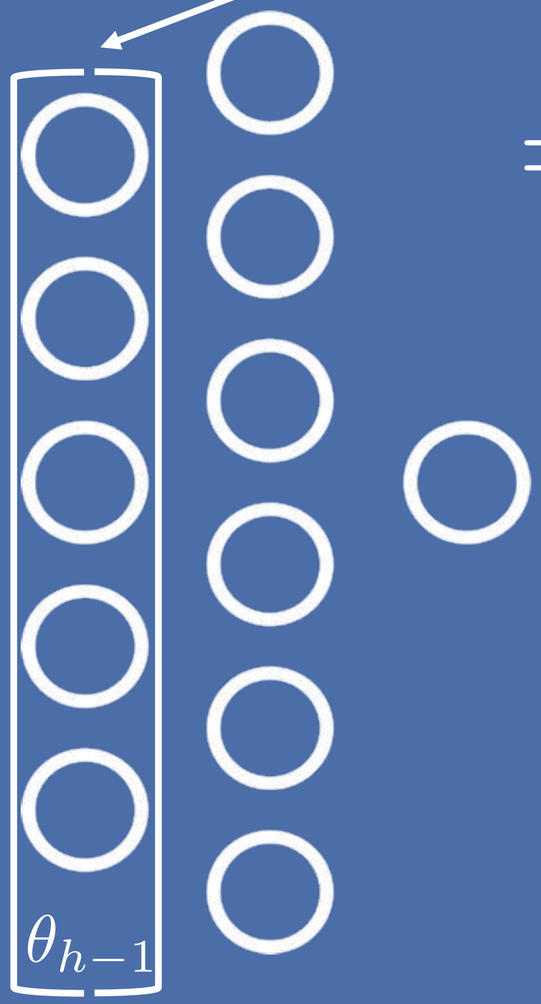
BackProp

Computing the gradient

$$\nabla_{\theta_{h-1}} \ell(y_h)$$

$$= \underbrace{\Phi_{h-1}}(\theta_{h-1}, \nabla_{y_{h-1}} \ell(y_h))$$

Depends on current layer's structure

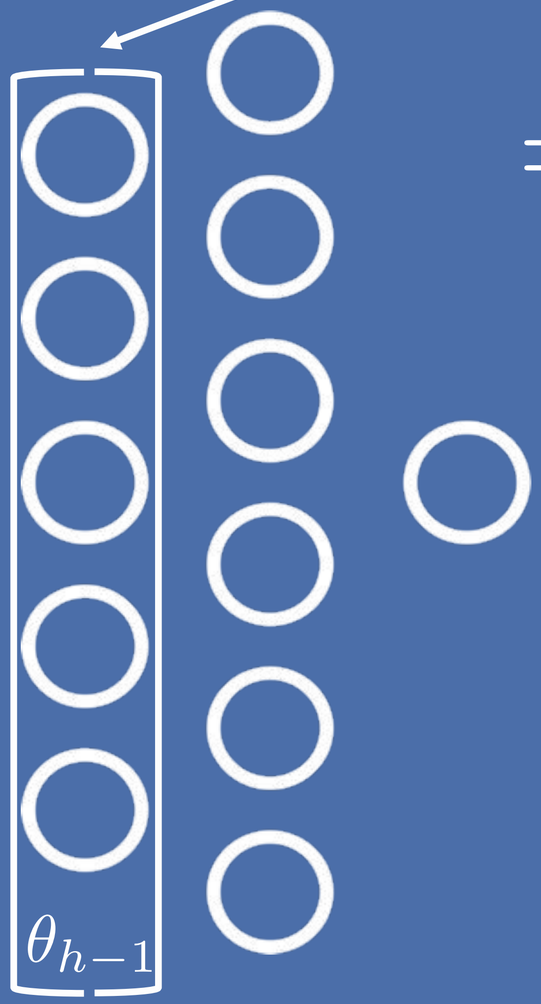


BackProp

Computing the gradient

$$\nabla_{\theta_{h-1}} \ell(y_h)$$

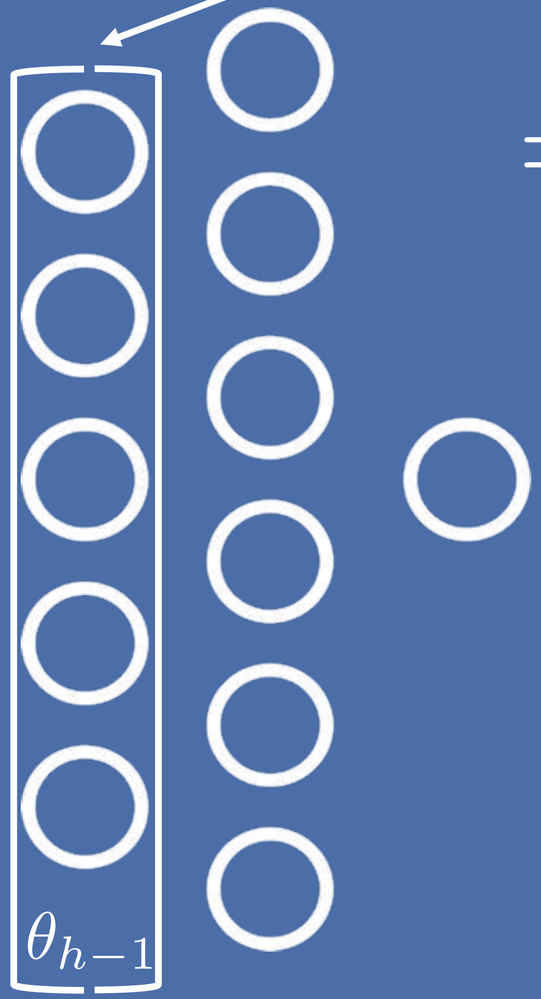
$$= \underbrace{\Phi_{h-1}}_{\text{Depends on current layer's structure}}(\underbrace{\theta_{h-1}}_{\text{Known}}, \nabla_{y_{h-1}} \ell(y_h))$$



BackProp

Computing the gradient

$$\nabla_{\theta_{h-1}} \ell(y_h)$$

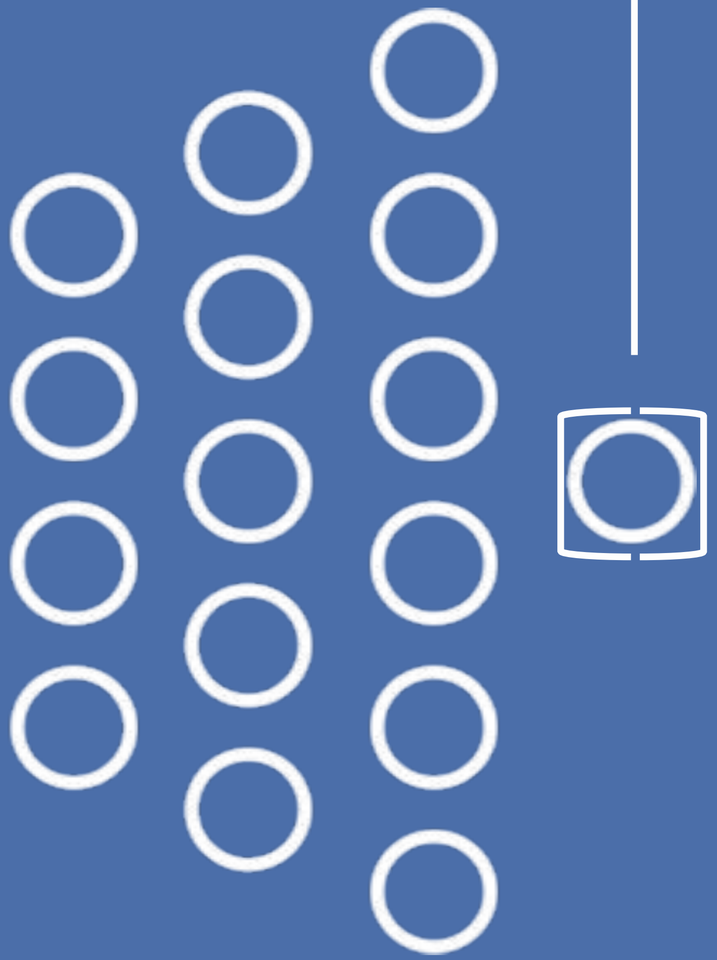


$$= \underbrace{\Phi_{h-1}}_{\text{Depends on current layer's structure}}\left(\underbrace{\theta_{h-1}}_{\text{Known}}, \underbrace{\nabla_{y_{h-1}} \ell(y_h)}_{\text{Already computed}}\right)$$

BackProp

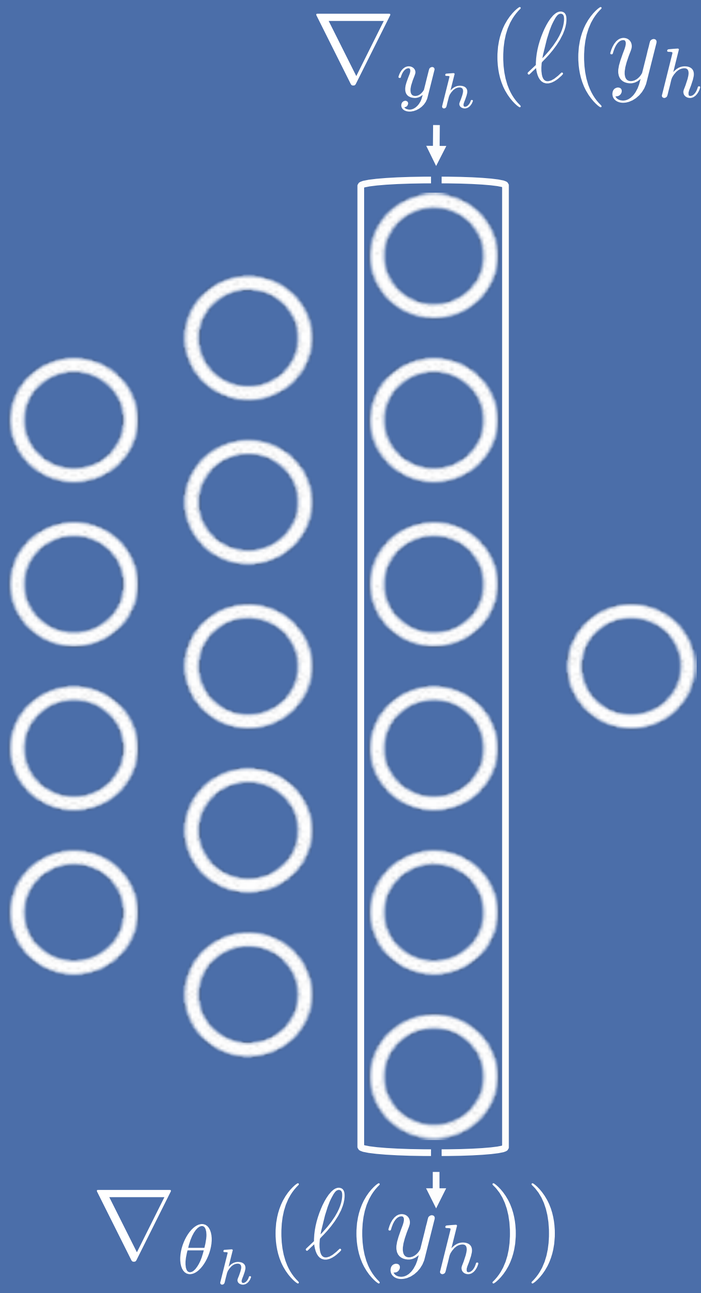
It's Backwards !

$$\nabla_{y_h}\left(\ell(y_h)\right)$$



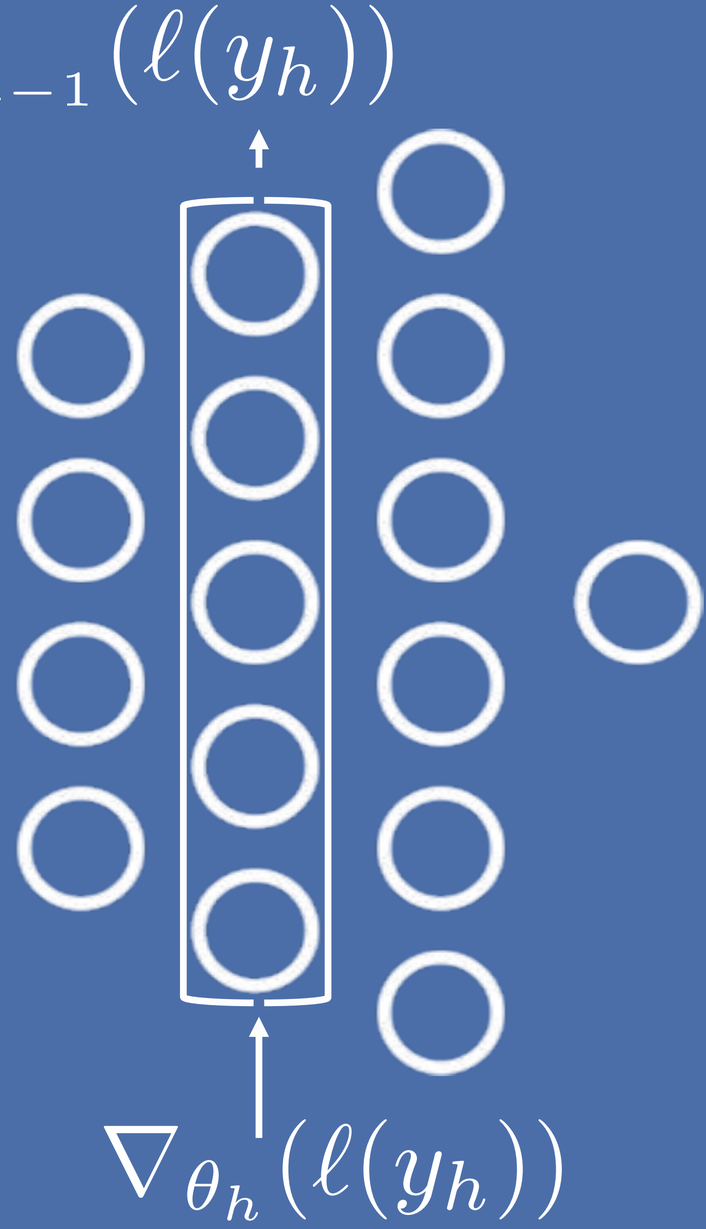
BackProp

It's Backwards !



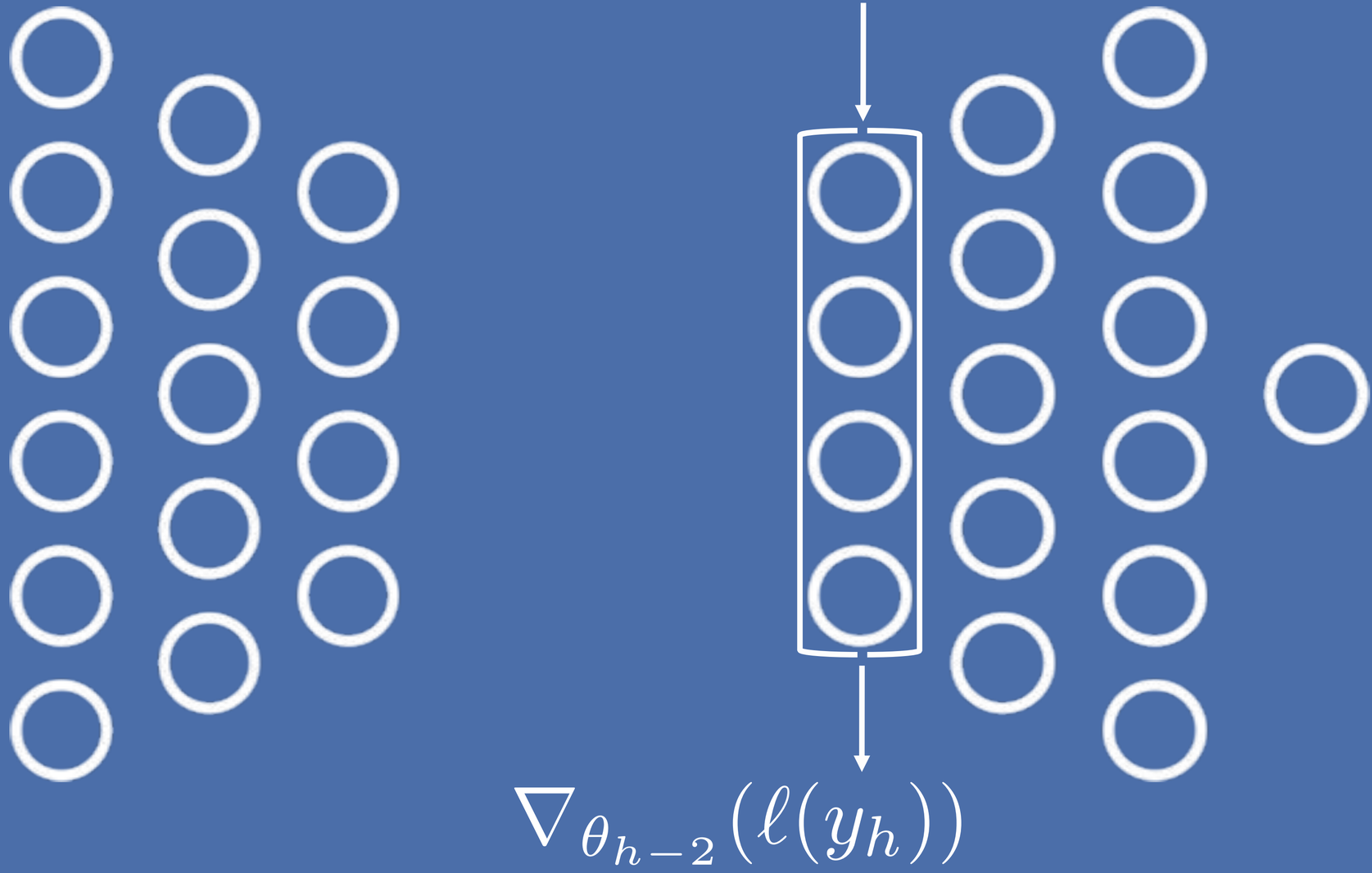
BackProp

It's Backwards !



BackProp

It's Backwards !



BackProp

It's Backwards !

More precisely, let's look at $o = f(x; \theta)$

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{i=1}^{m} \frac{\partial o_i}{\partial \theta_j} \frac{\partial \mathcal{L}}{\partial o_i}$$

Next layer gives us $\nabla_o \mathcal{L}(y)$

We use the chain rule to compute :

- The gradient of our parameters : $\nabla_\theta \mathcal{L}(y) = J_\theta(f) \nabla_o \mathcal{L}(y)$

- The gradient of the inputs : $\nabla_x \mathcal{L}(y) = J_x(f) \nabla_o \mathcal{L}(y)$

Return the gradient of the inputs

Recap

The Story so far

- Training / Testing set
- Loss
- (Stochastic) Gradient Descent
- Back-Propagation

Recap
The Story so far

- Training / Testing set -> Required to avoid overfitting
- Loss
- (Stochastic) Gradient Descent
- Back-Propagation

Recap
The Story so far

- Training / Testing set -> Required to avoid overfitting
- Loss -> Defines the problem you’re solving
- (Stochastic) Gradient Descent
- Back-Propagation

Recap
The Story so far

- Training / Testing set -> Required to avoid overfitting
- Loss -> Defines the problem you’re solving
- (Stochastic) Gradient Descent -> Minimize by taking successive steps
- Back-Propagation

Recap
The Story so far

- Training / Testing set -> Required to avoid overfitting
- Loss -> Defines the problem you're solving
- (Stochastic) Gradient Descent -> Minimize by taking successive steps
- Back-Propagation -> A trick to compute the gradient

Recap

The Story so far

Pick a random example (x_j, y_j)

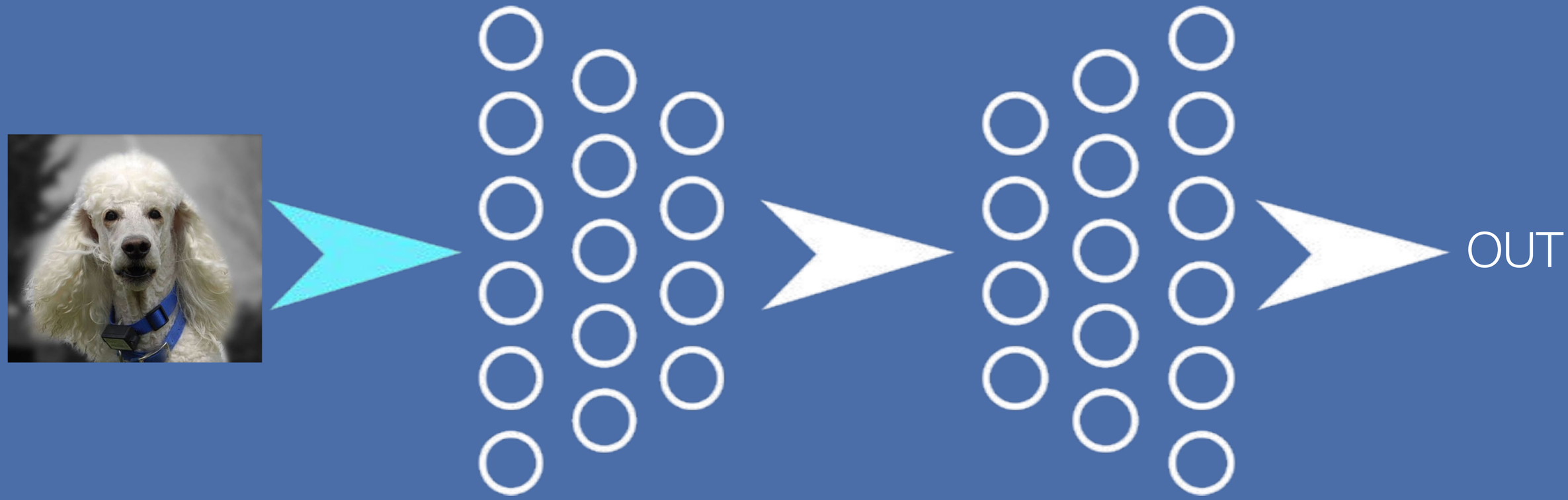
Recap
The Story so far

Pick a random example

poodle

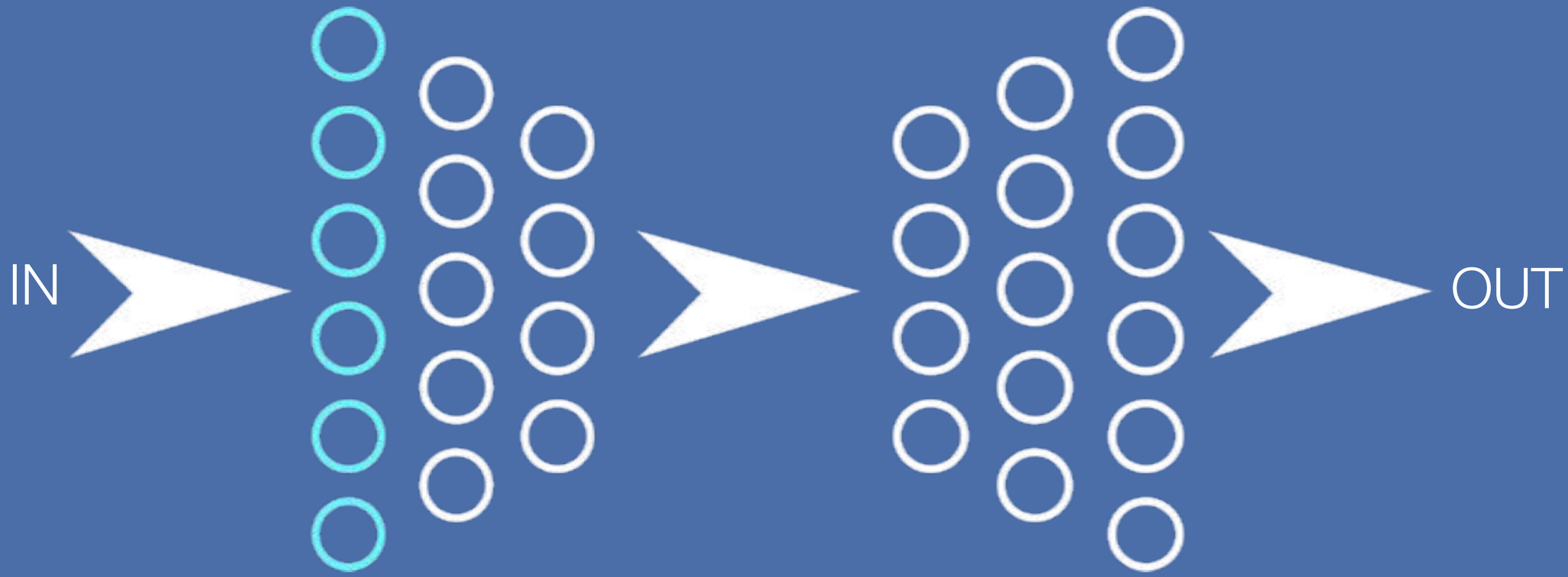
Forward Pass

Hmm what's this picture …



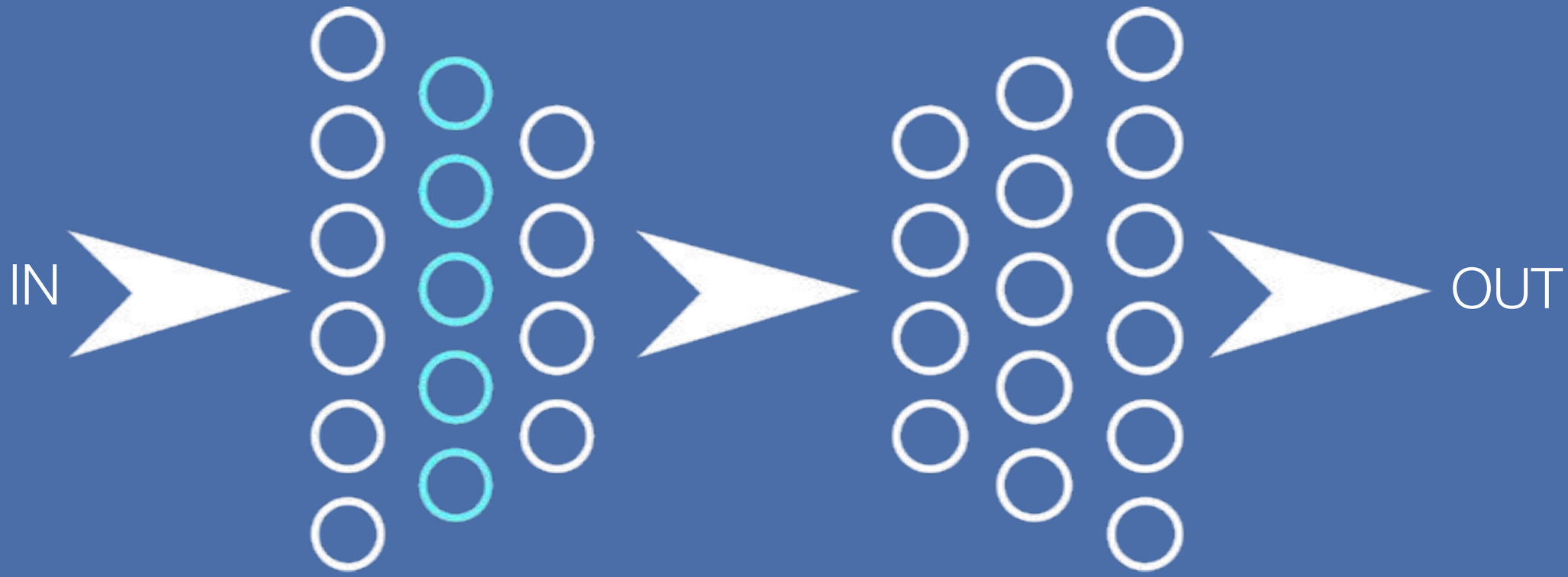
Forward Pass

Wait for it …



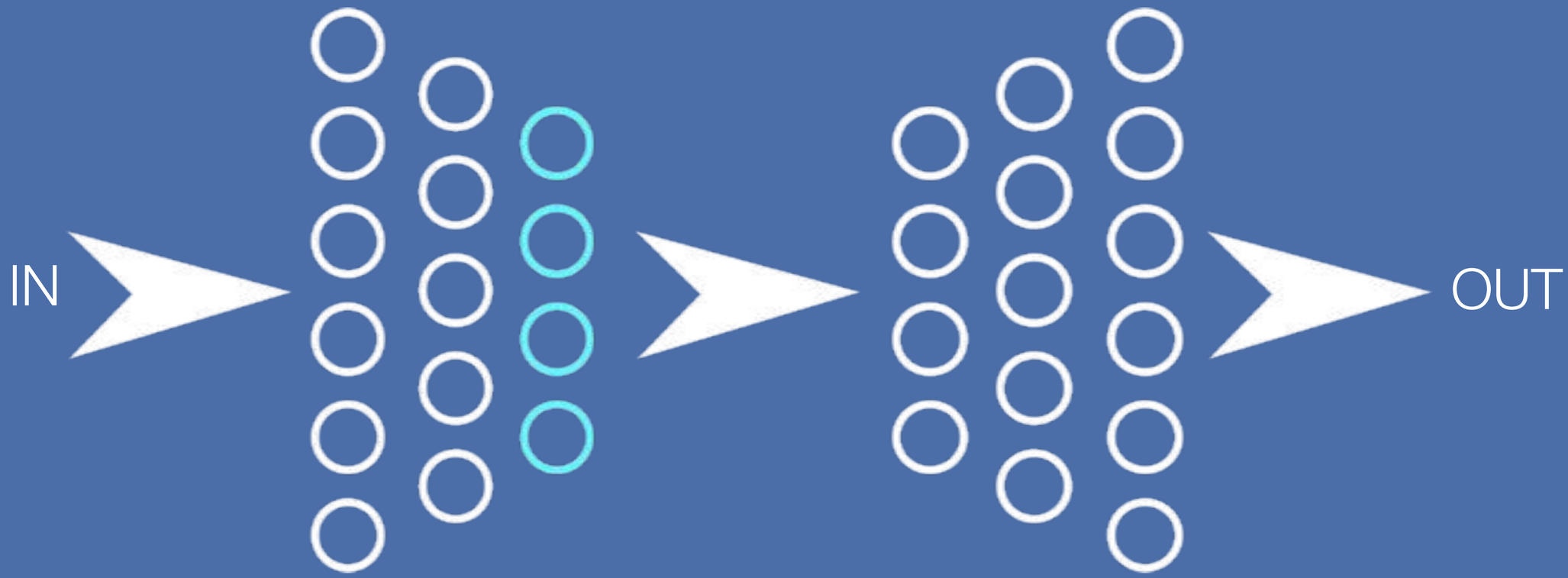
Forward Pass

...



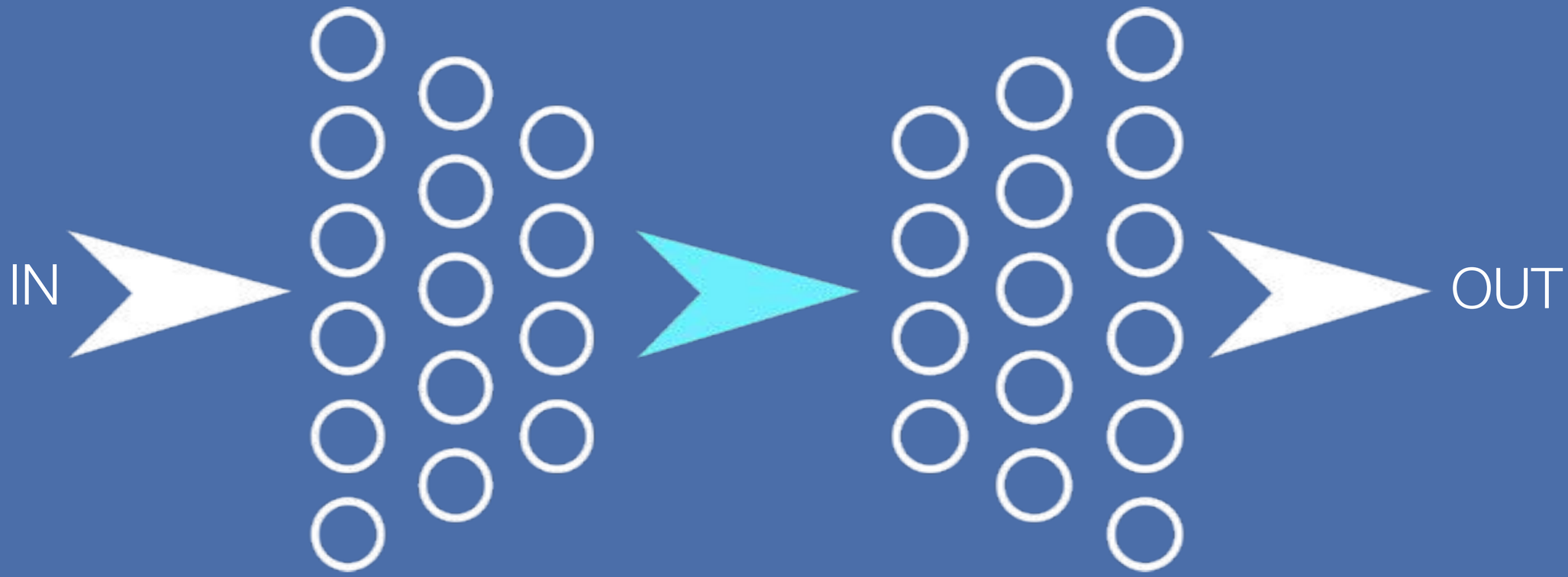
Forward Pass

...



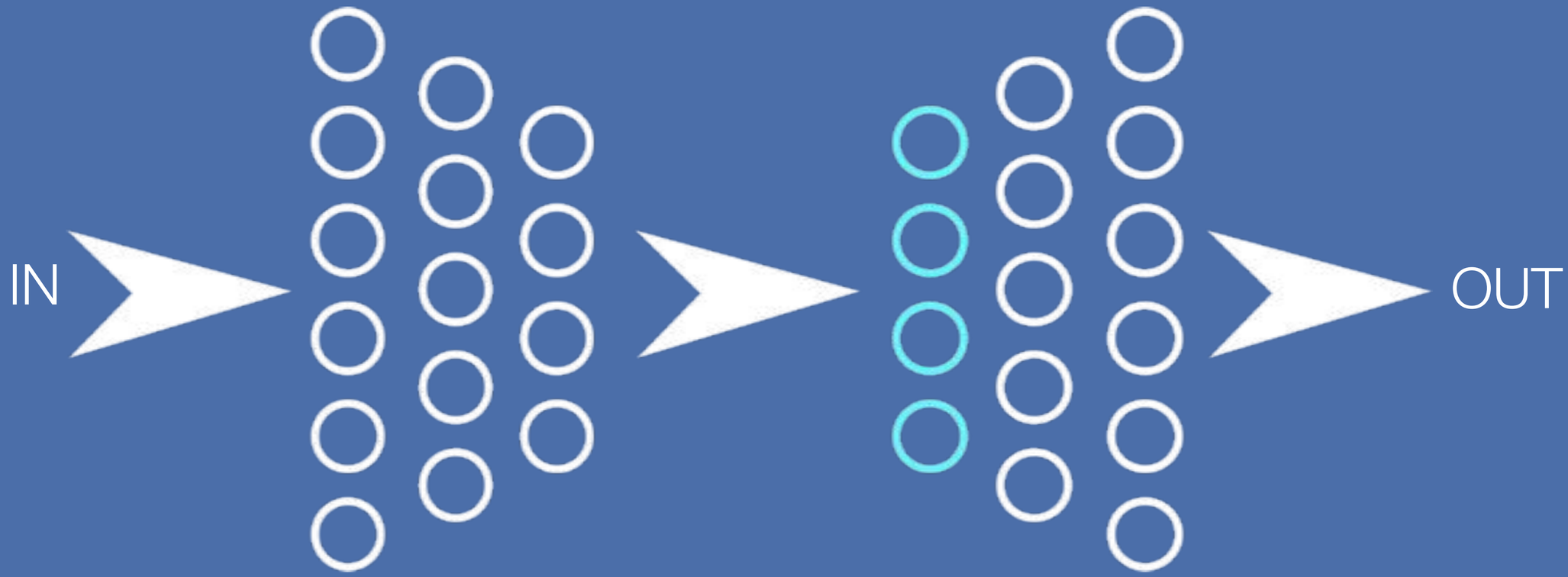
Forward Pass

...



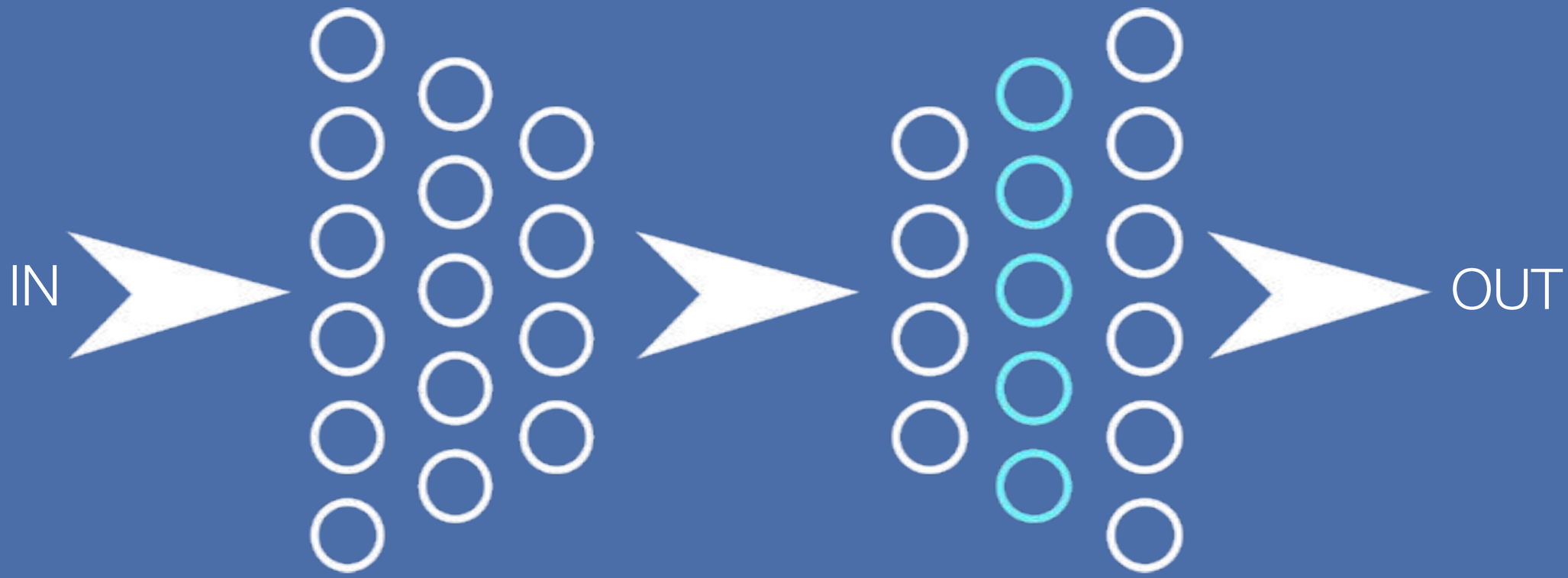
Forward Pass

...



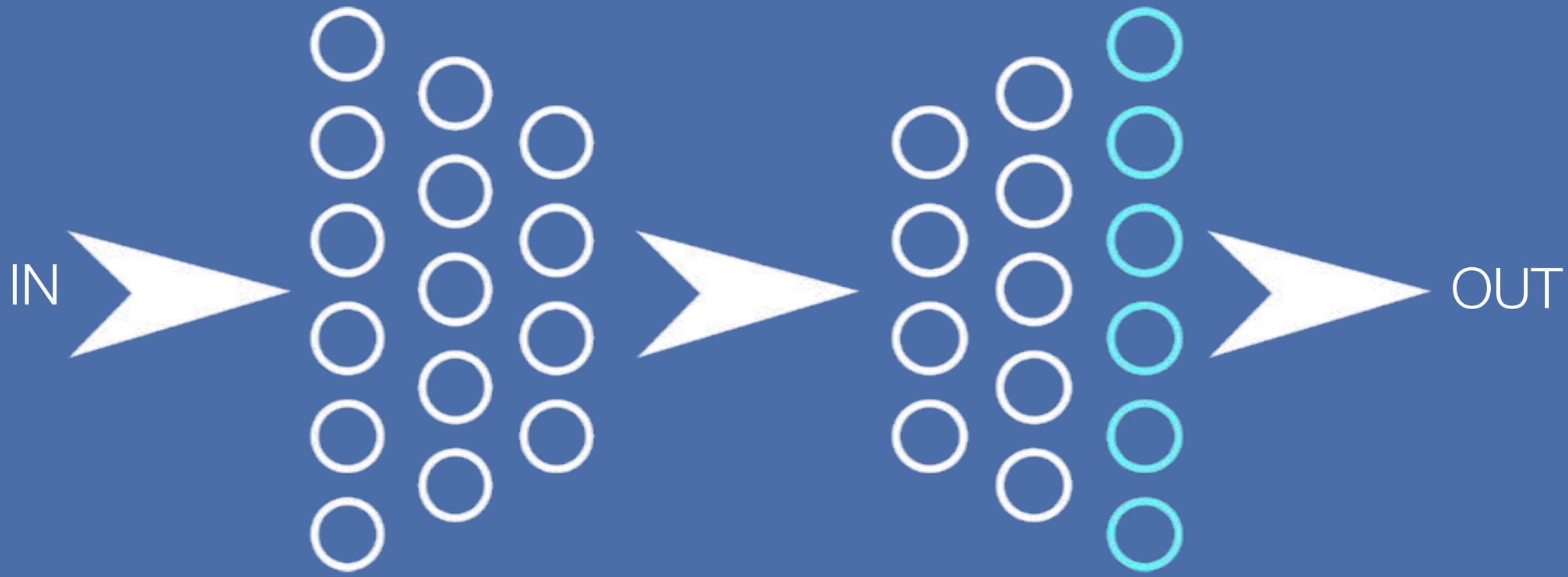
Forward Pass

...



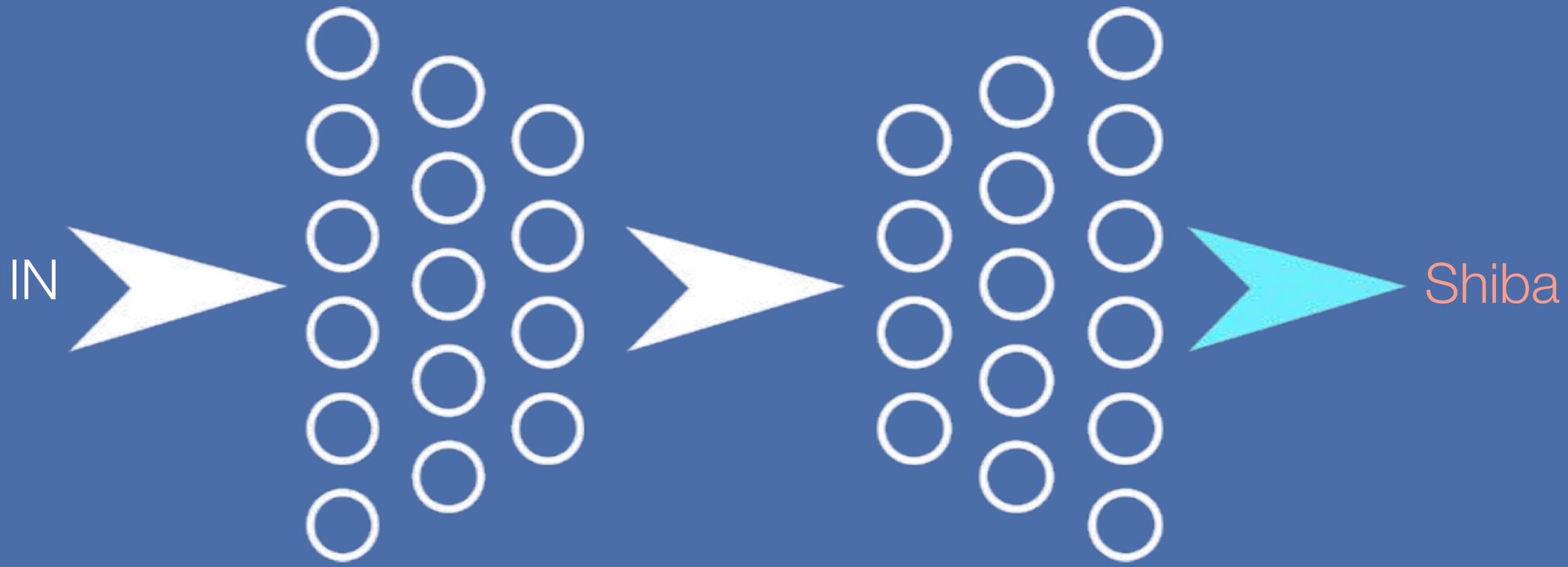
Forward Pass

...



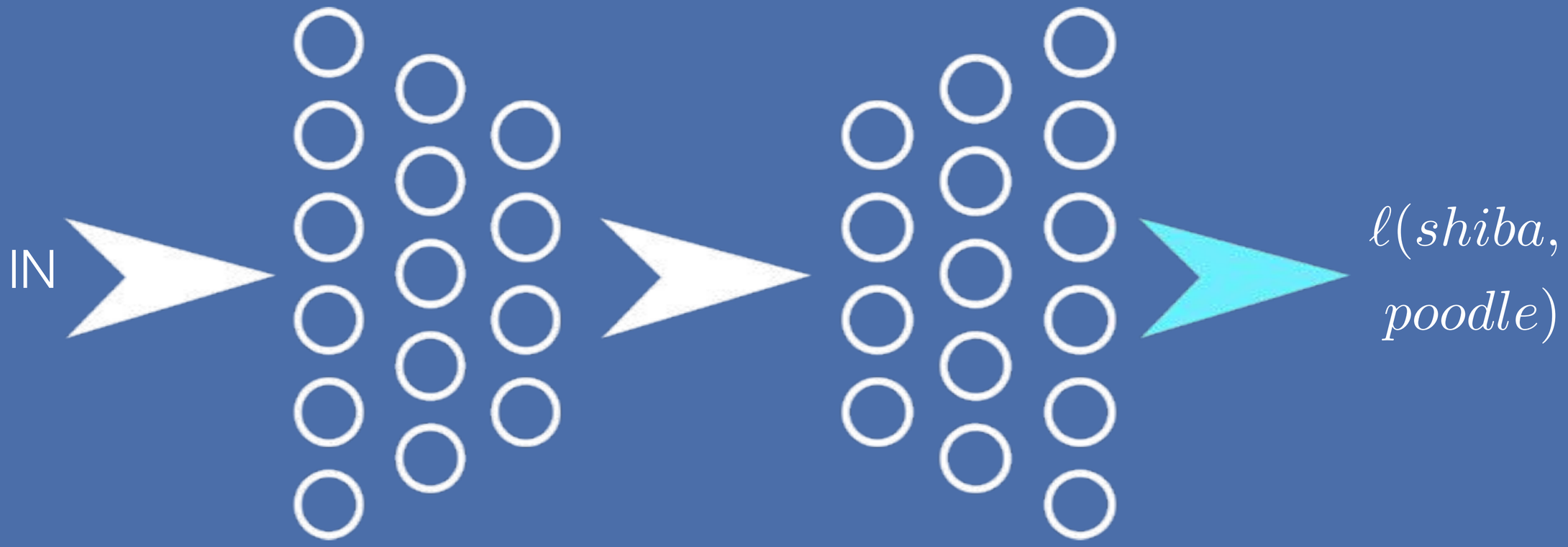
Forward Pass

It’s a Shiba !



Forward Pass

Wait, was it a shiba ?



BackProp

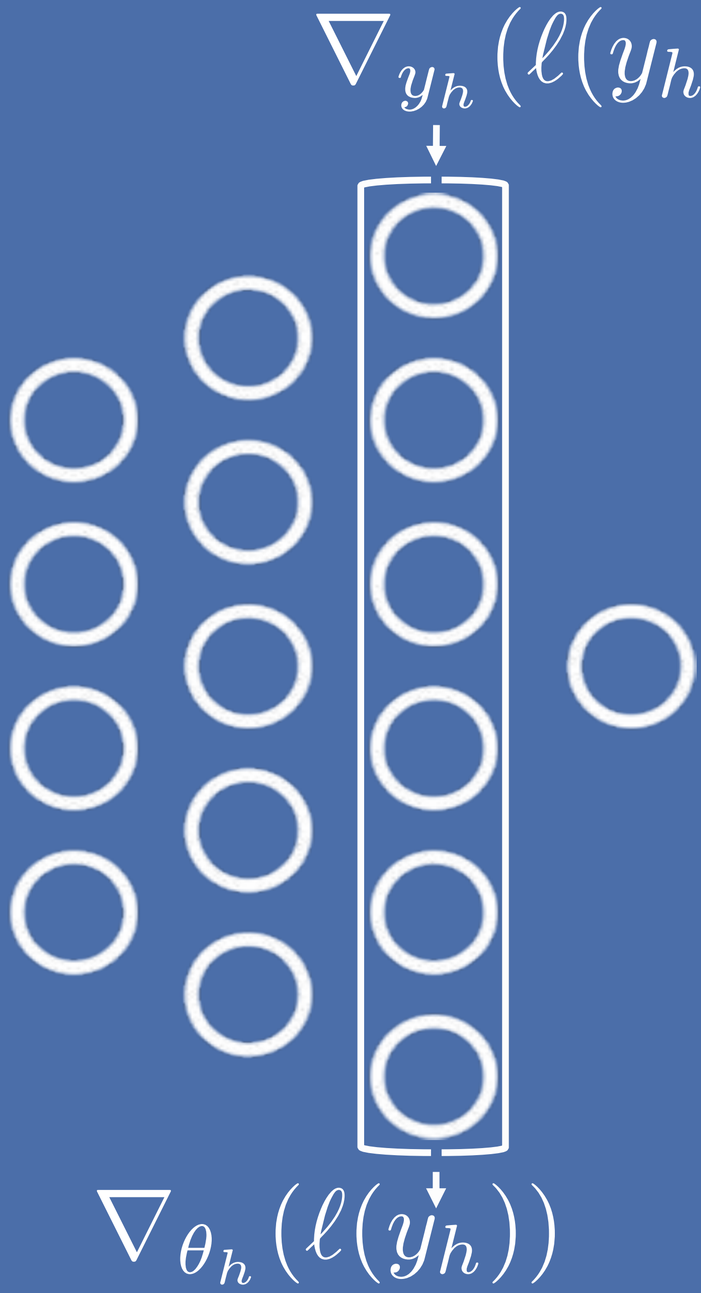
It was a poodle !

$\nabla_{y_h}(\ell(y_h))$

$\ell(shiba, poodle)$

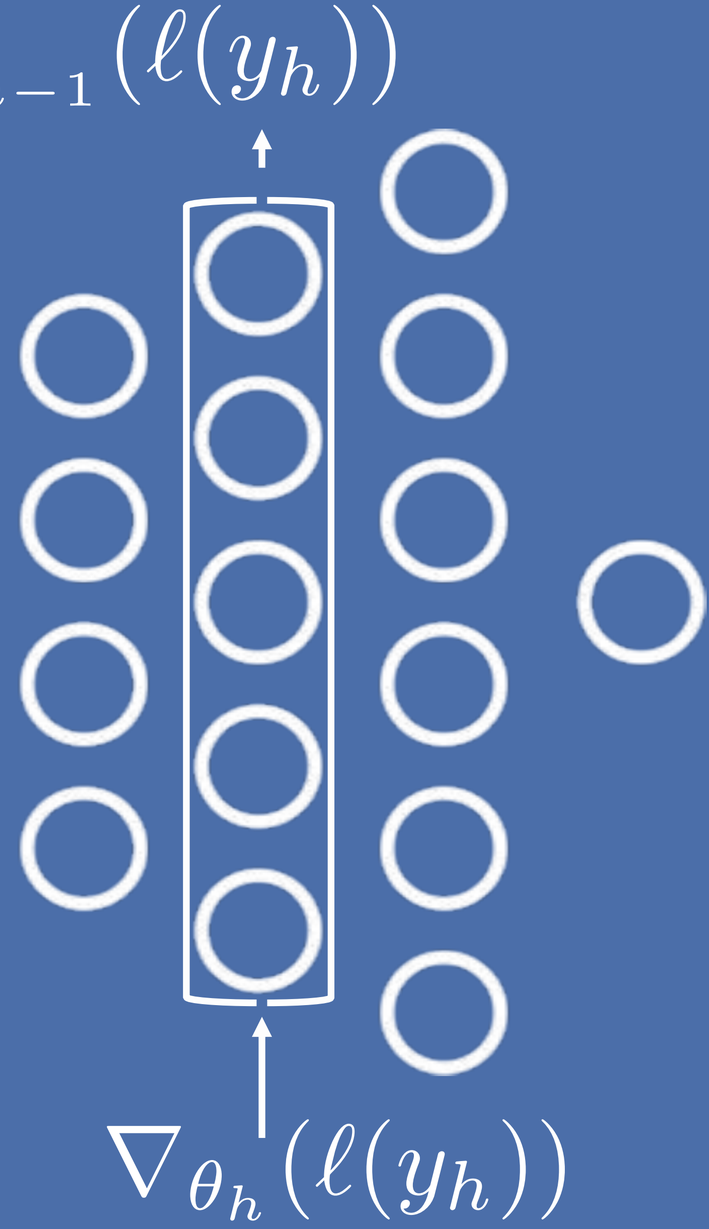
BackProp

It was a poodle !



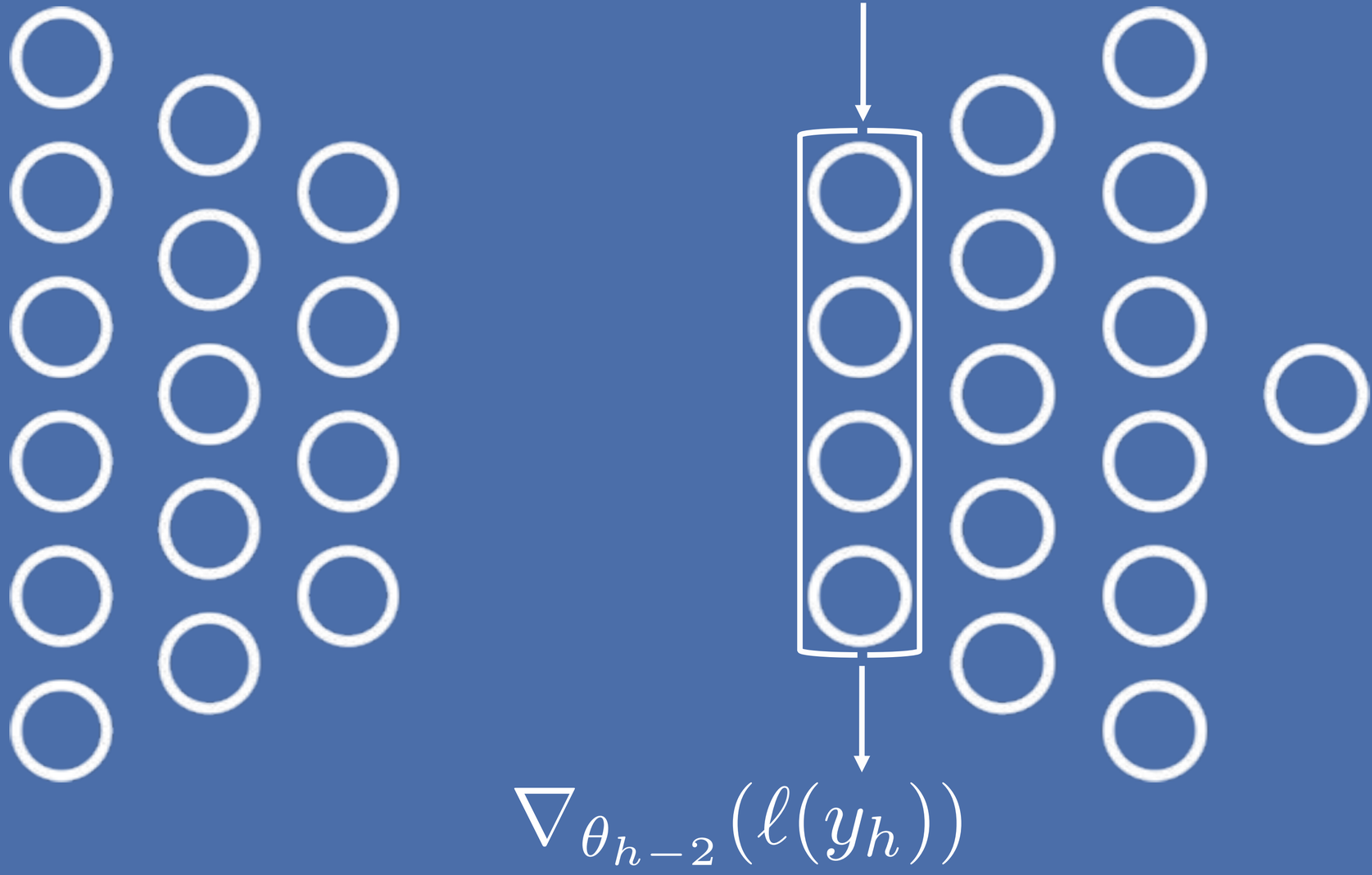
BackProp

It was a poodle !



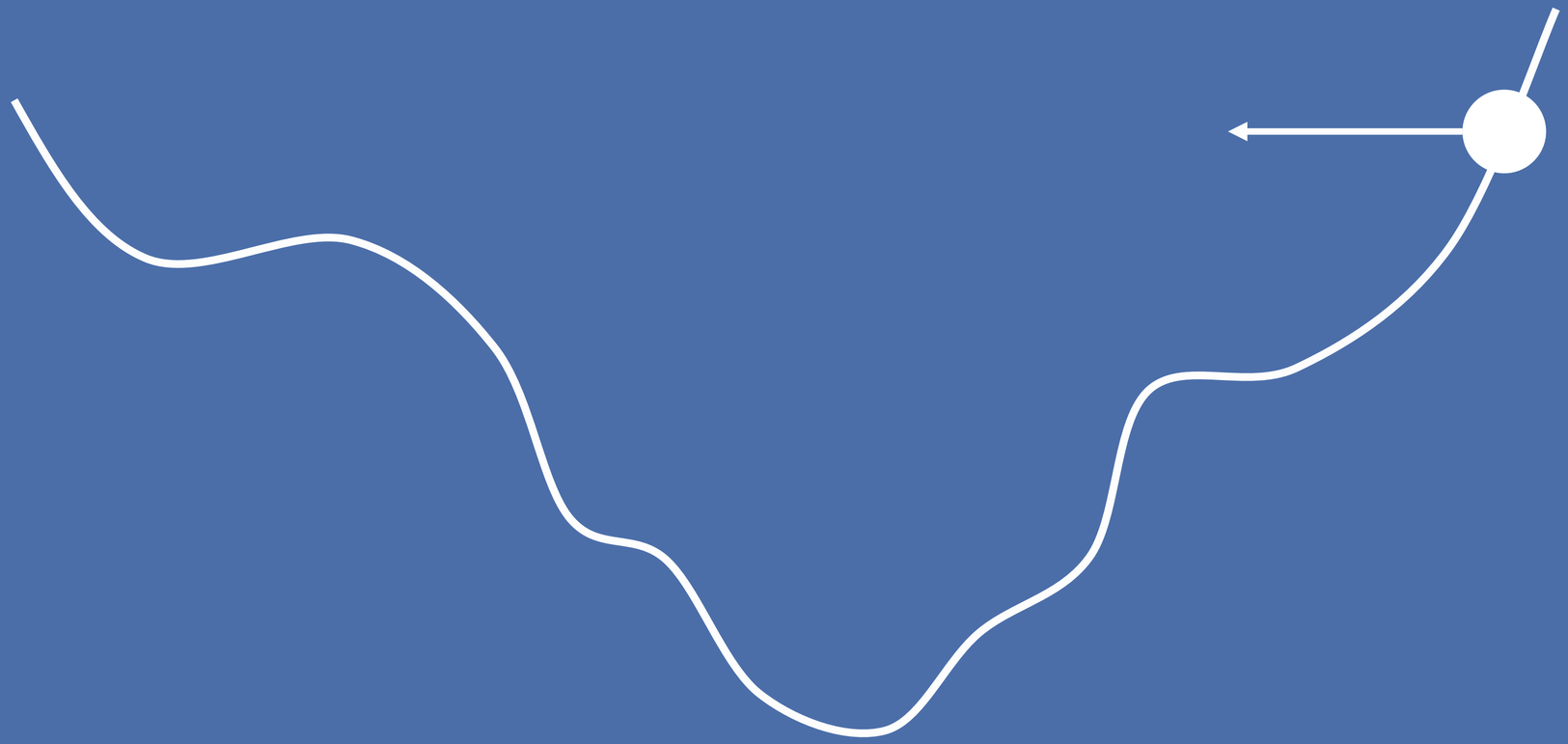
BackProp

It was a poodle !



Gradient Descent

Take a step !



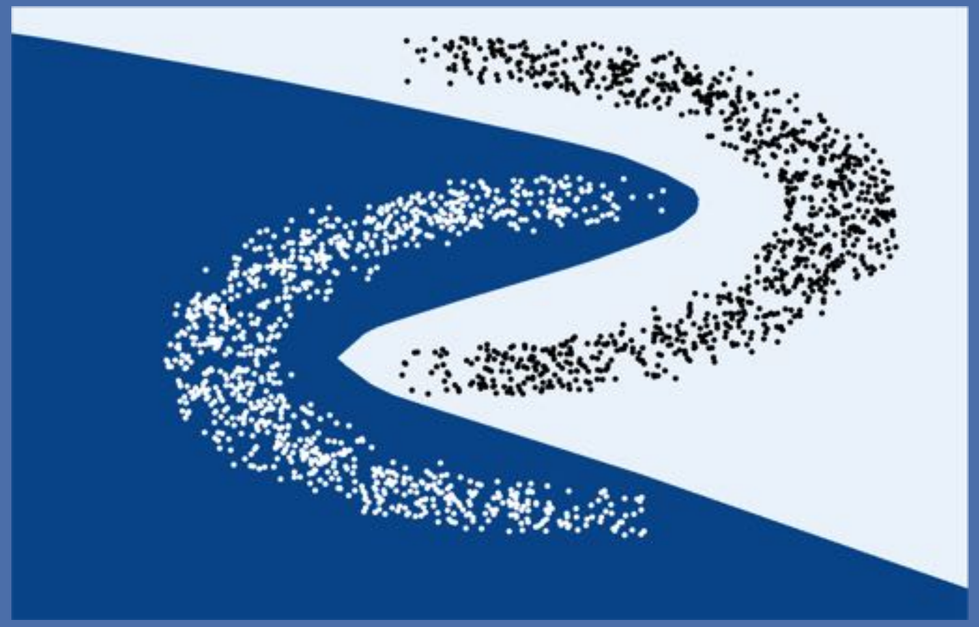
In Practice

”How I learned to stop caring and love the Backprop”

- PyTorch
- Picking Hyper-Parameters

In PyTorch

It's so easy …



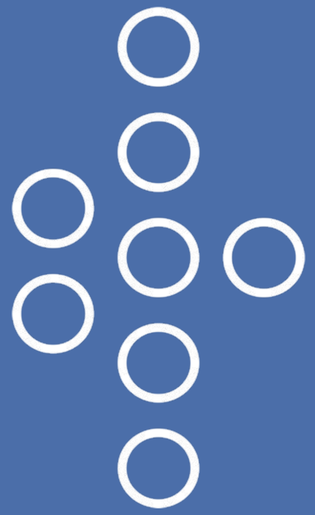
```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

In PyTorch

It's so easy …



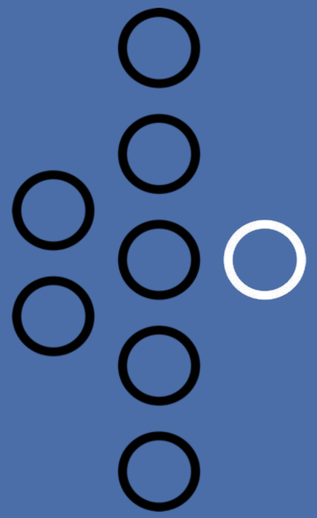
```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

In PyTorch

It's so easy …



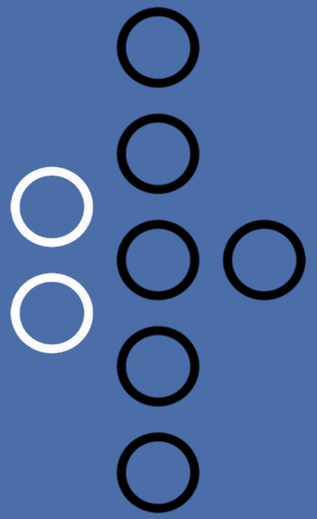
```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

In PyTorch

It's so easy …



```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

Stochastic Gradient Descent

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

Set all stored gradients to zero ←

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

Forward pass (output from input) ←

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

Compute the loss ←

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

BackProp ←

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

Gradient Step ←

In PyTorch

It's so easy …

Wait what ?
Is that it ?

```
import torch
import torch.nn as nn
import torch.optim as optim

h = 50
net = nn.Sequential(
    nn.Linear(2, h),
    nn.ReLU(),
    nn.Linear(h, 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(net.parameters(), lr=1)

for i in range(100):
    optimizer.zero_grad()
    output = net(X[i])
    loss = nn.BCELoss(output, Y[i])
    loss.backward()
    optimizer.step()
```

In PyTorch

It's so easy …

Define Model Parameters

```
import torch
import torch.nn as nn

class Linear(nn.Module):
    def __init__(self):
        super(Linear, self).__init__()
        self.A = nn.Parameter(torch.randn(hidden_size, input_size))
        self.b = nn.Parameter(torch.randn(hidden_size, 1))

    def forward(self, x):
        x = torch.mm(self.A, torch.t(x)) + self.b
        return x
```

In PyTorch

It's so easy …

```
import torch
import torch.nn as nn

class Linear(nn.Module):
    def __init__(self):
        super(Linear, self).__init__()
        self.A = nn.Parameter(torch.randn(hidden_size, input_size))
        self.b = nn.Parameter(torch.randn(hidden_size, 1))

    def forward(self, x):
        x = torch.mm(self.A, torch.t(x)) + self.b
        return x
```

Define Forward Pass

In PyTorch

It's so easy …

For simple forwards,
the backward pass will be
added automatically !

```
import torch
import torch.nn as nn

class Linear(nn.Module):
    def __init__(self):
        super(Linear, self).__init__()
        self.A = nn.Parameter(torch.randn(hidden_size, input_size))
        self.b = nn.Parameter(torch.randn(hidden_size, 1))

    def forward(self, x):
        x = torch.mm(self.A, torch.t(x)) + self.b
        return x
```

Hyper-Parameters

AKA Those that shall not be learned

How do we pick :

- Number of Layers
- Number of Hidden Units
- Activation Function
- Loss
- Learning Rate
- …

Hyper-Parameters

AKA Those that shall not be learned

How do we pick :

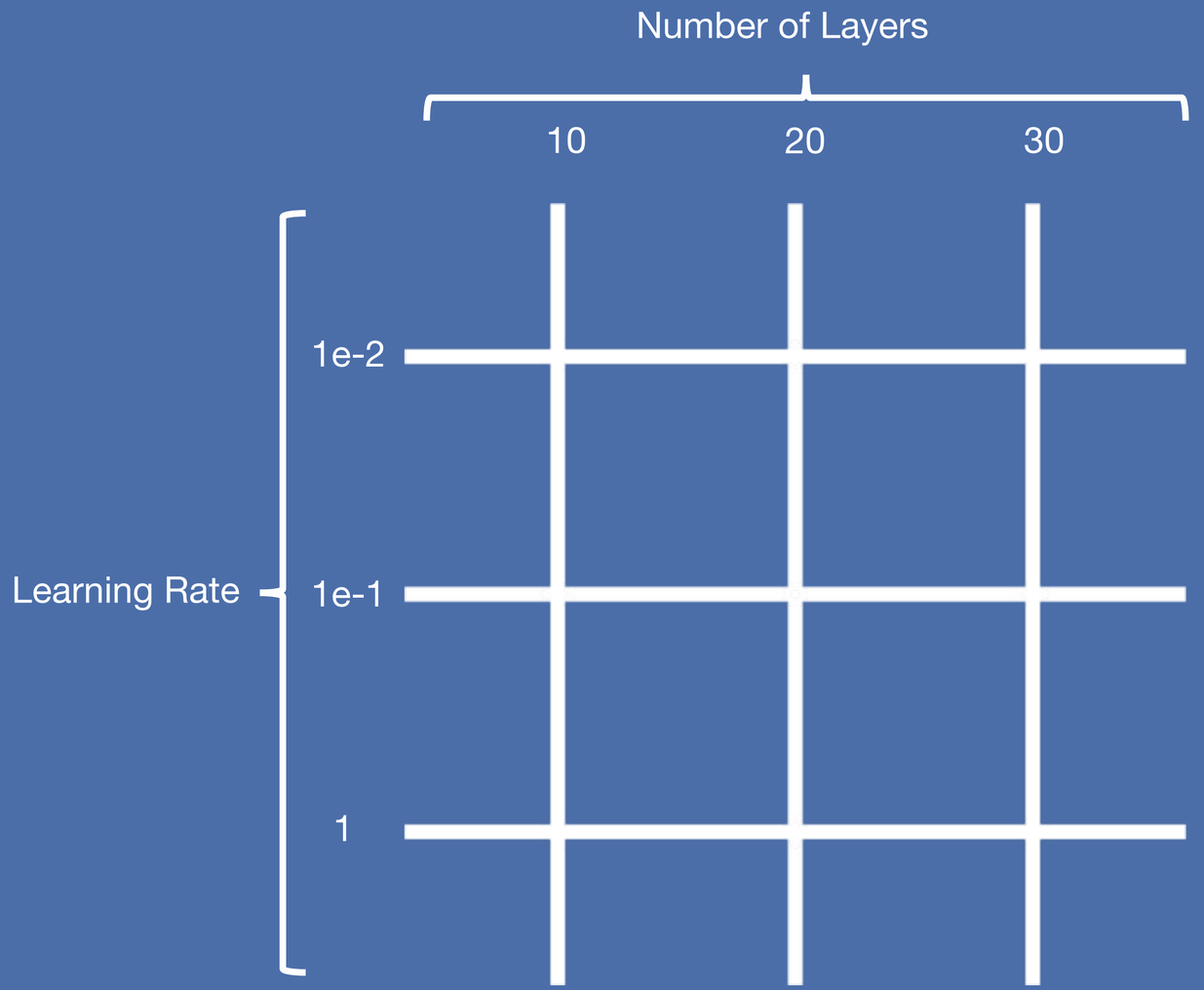
- Number of Layers
- Number of Hidden Units
- Activation Function
- Loss
- Learning Rate
- …

We don’t know !

Grid Search

AKA Science !

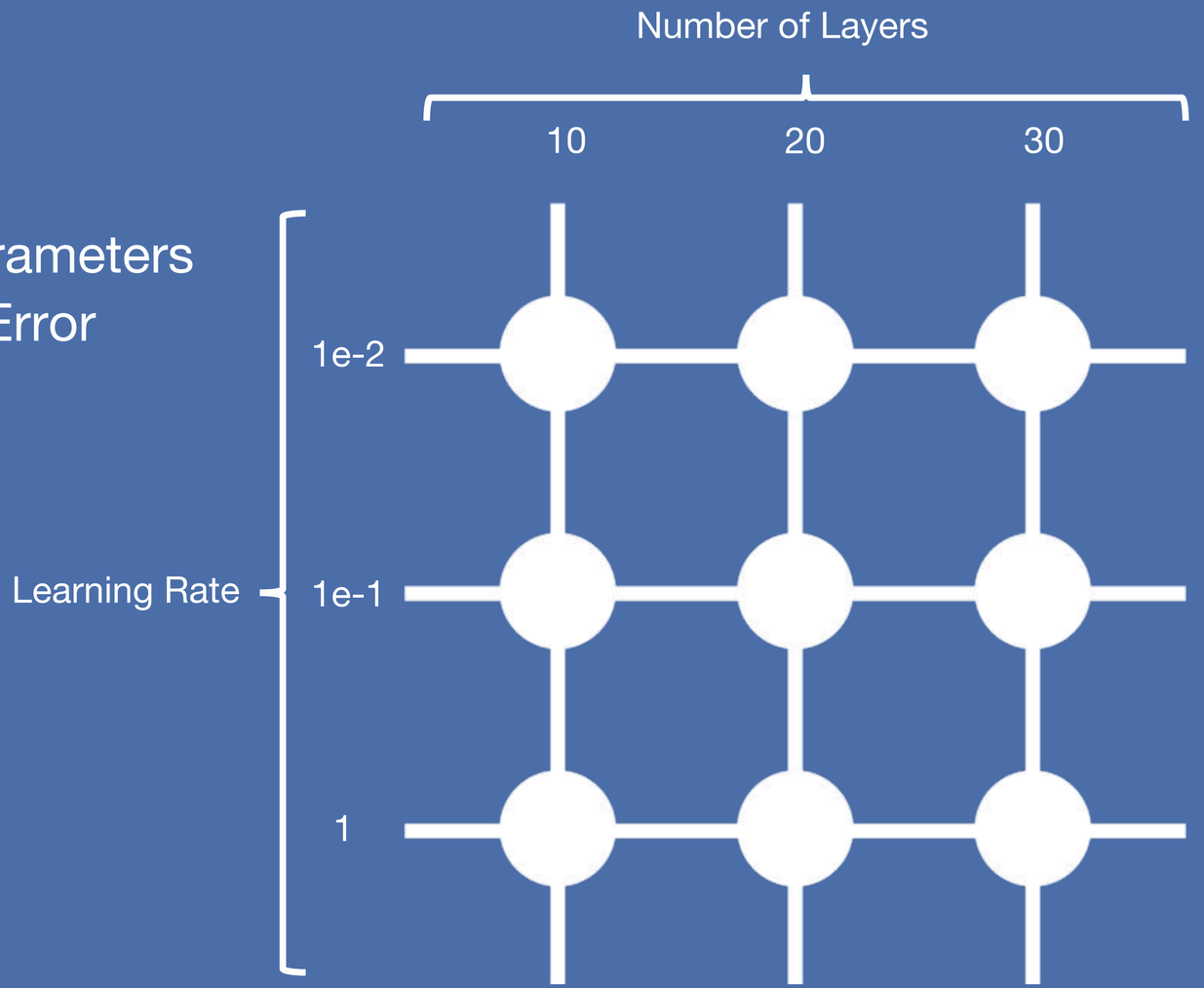
We use Grid-Searches :



Grid Search
AKA Science !

We use Grid-Searches :

- Evaluate for set of hyper-parameters
- Select Based on Validation Error



Recap

The Story so far

- Pytorch
- Grid-Search

Recap

The Story so far

- Pytorch -> Allows us not to care too much about Backprop
- Grid-Search

Recap

The Story so far

- Pytorch -> Allows us not to care too much about Backprop
- Grid-Search -> How we set Hyper-Parameters we cannot learn