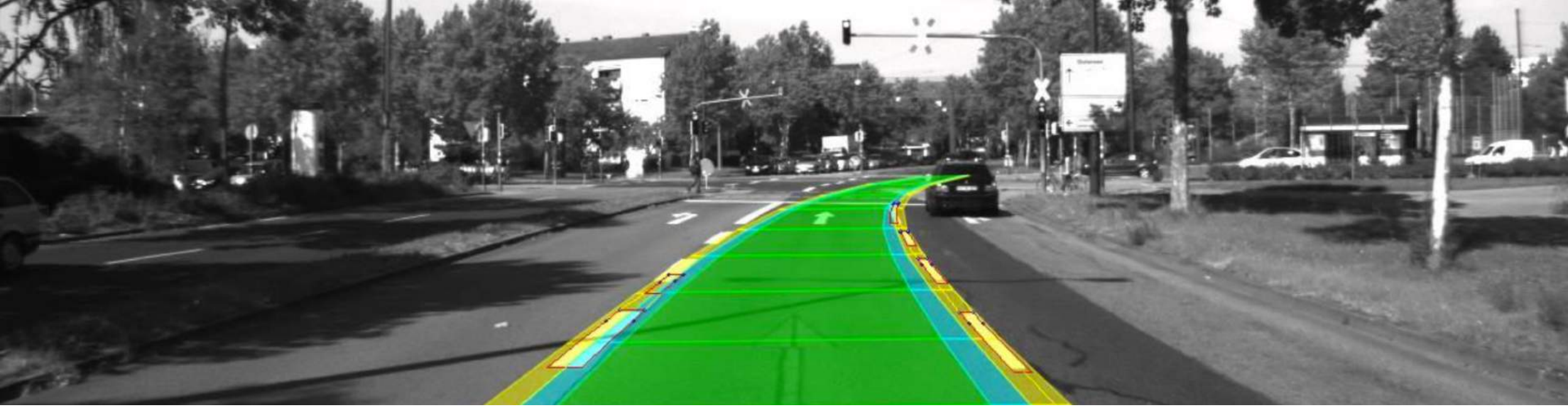


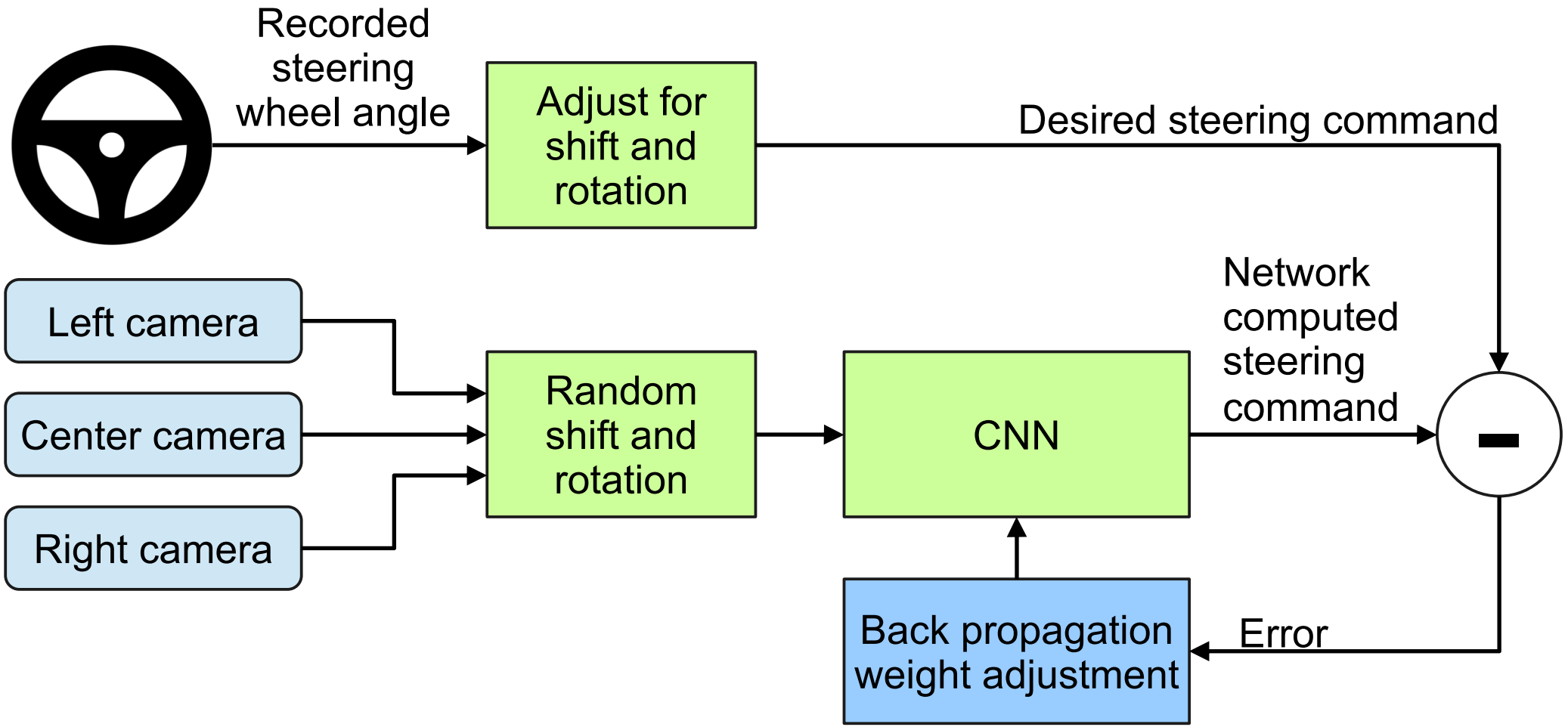
Module 3: Automotive CPS Data driven modeling

Principles of Modeling for Cyber-Physical Systems

Instructor: Madhur Behl

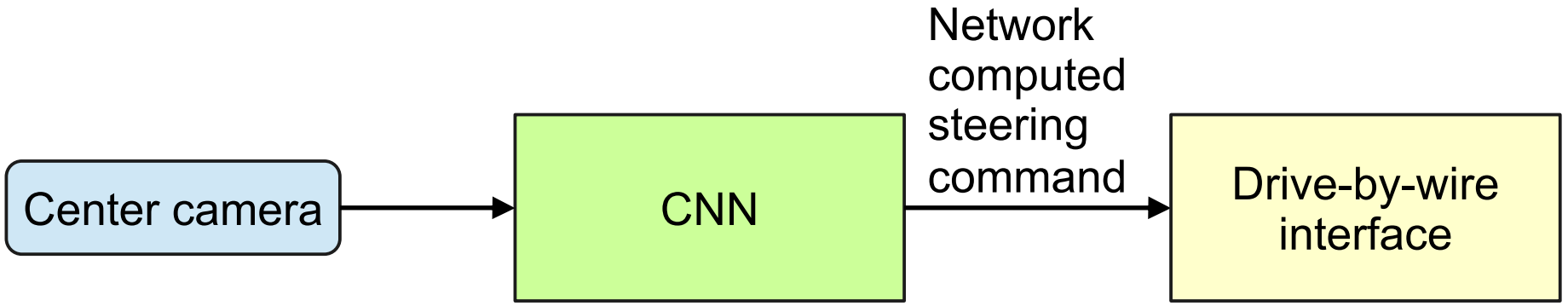
Slides credits:
Brandon Rohrer

TRAINING THE NEURAL NETWORK



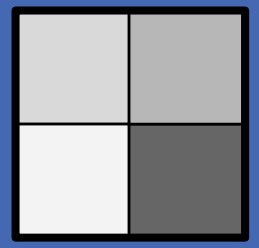
DRIVING

With a single front-facing camera

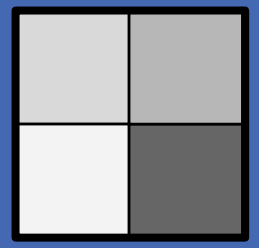


How neural networks work

A four pixel image



Categorize images



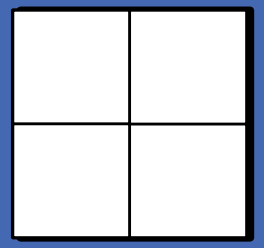
solid

vertical

diagonal

horizontal

Categorize images



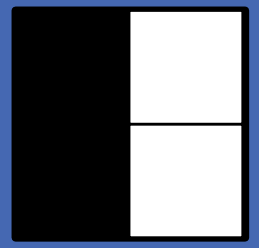
solid

vertical

diagonal

horizontal

Categorize images



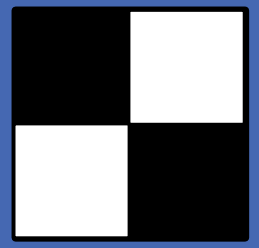
solid

vertical

diagonal

horizontal

Categorize images



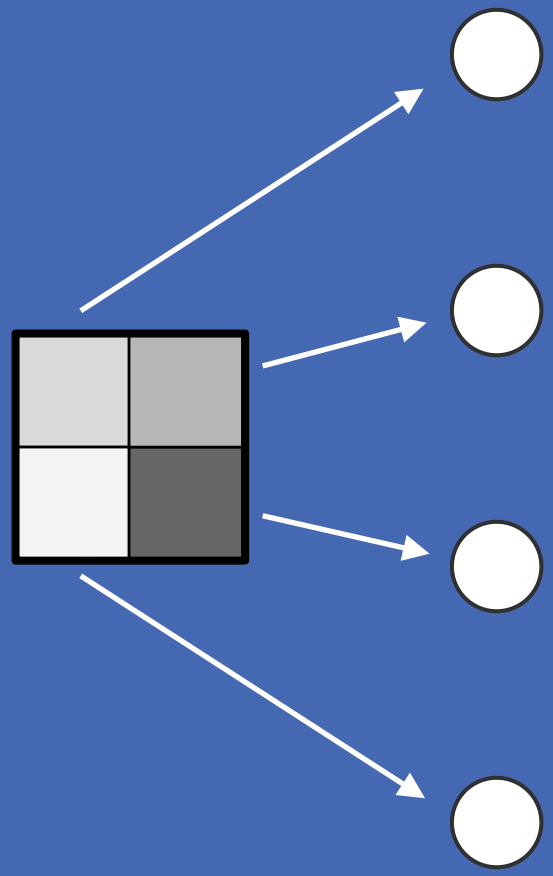
solid

vertical

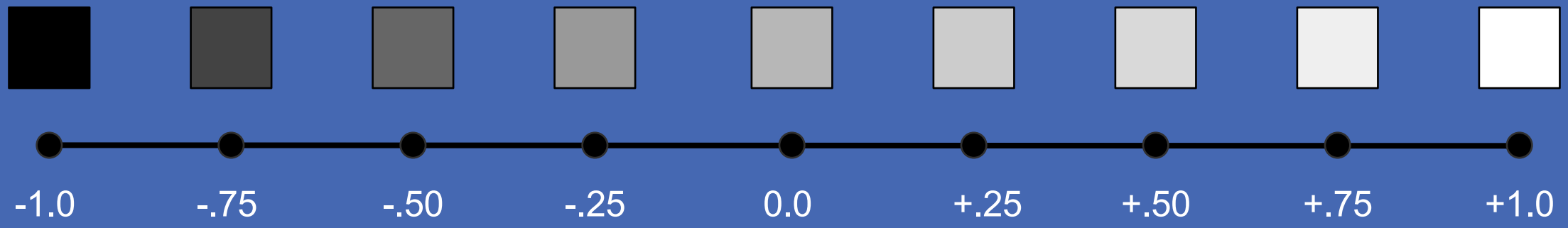
diagonal

horizontal

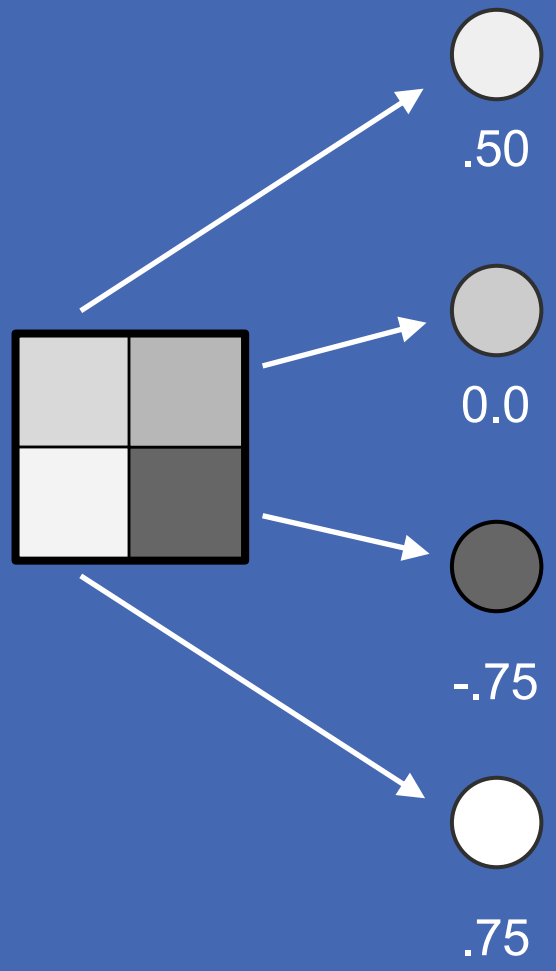
Input neurons



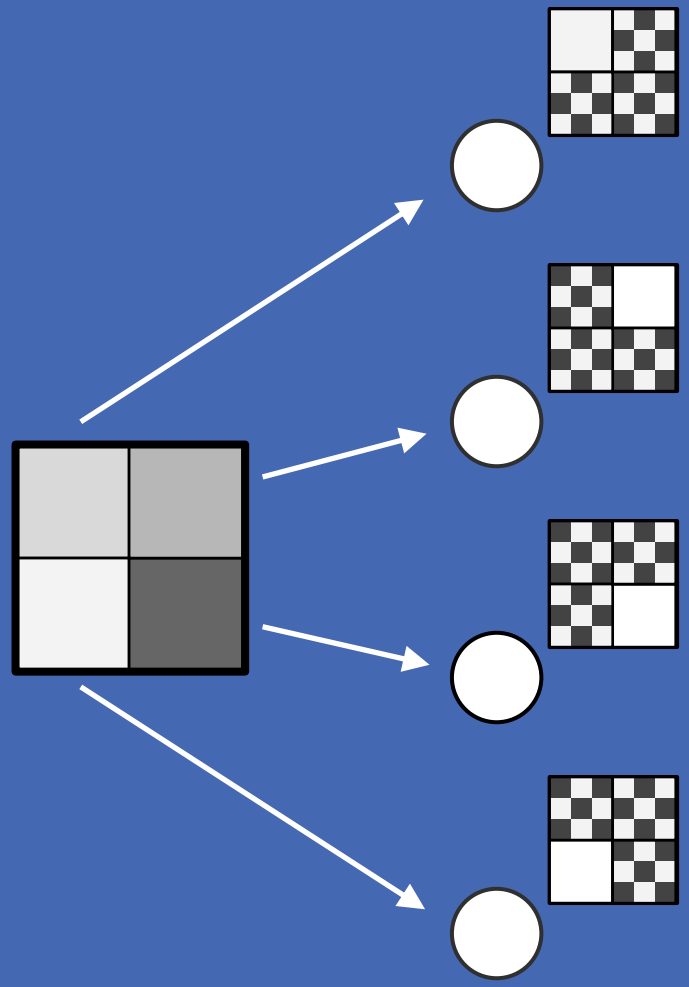
Pixel brightness



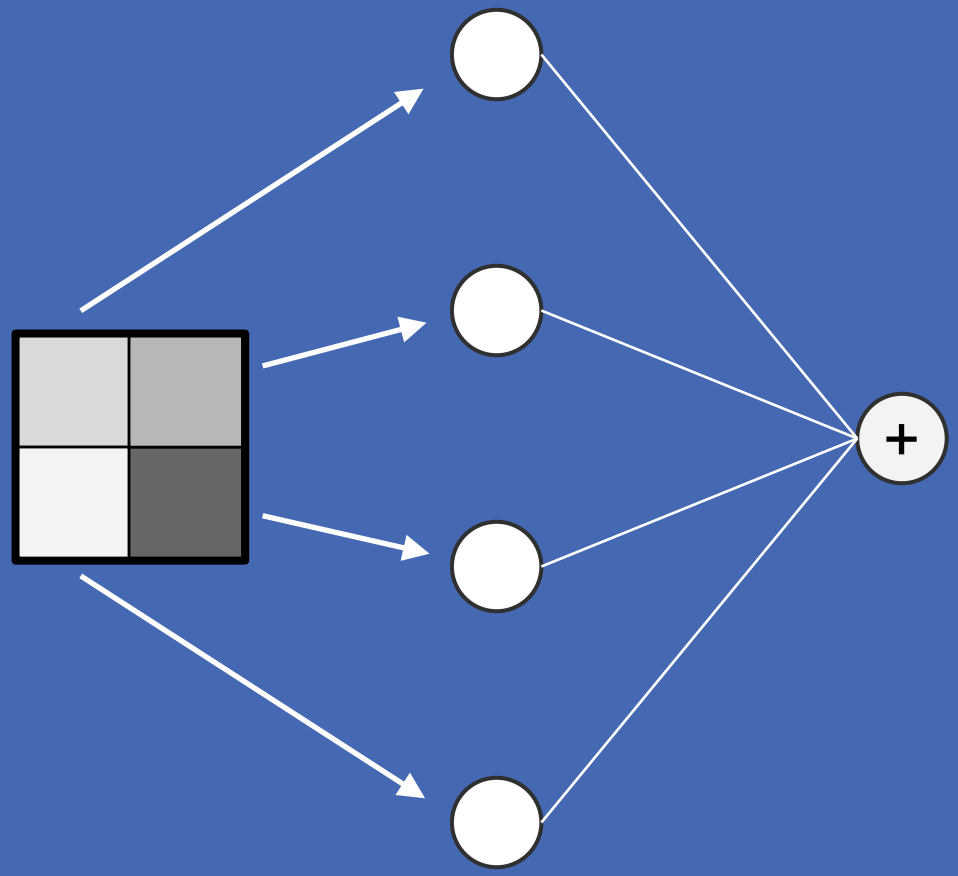
Input vector



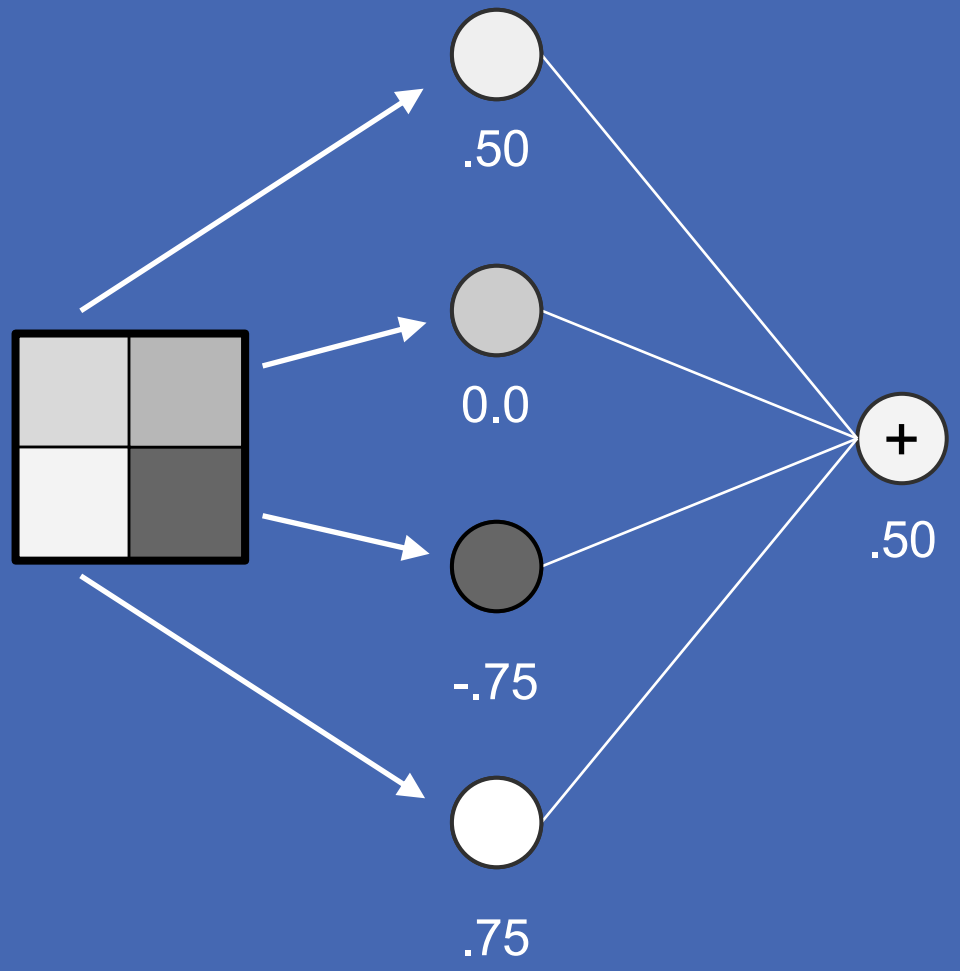
Receptive fields



A neuron



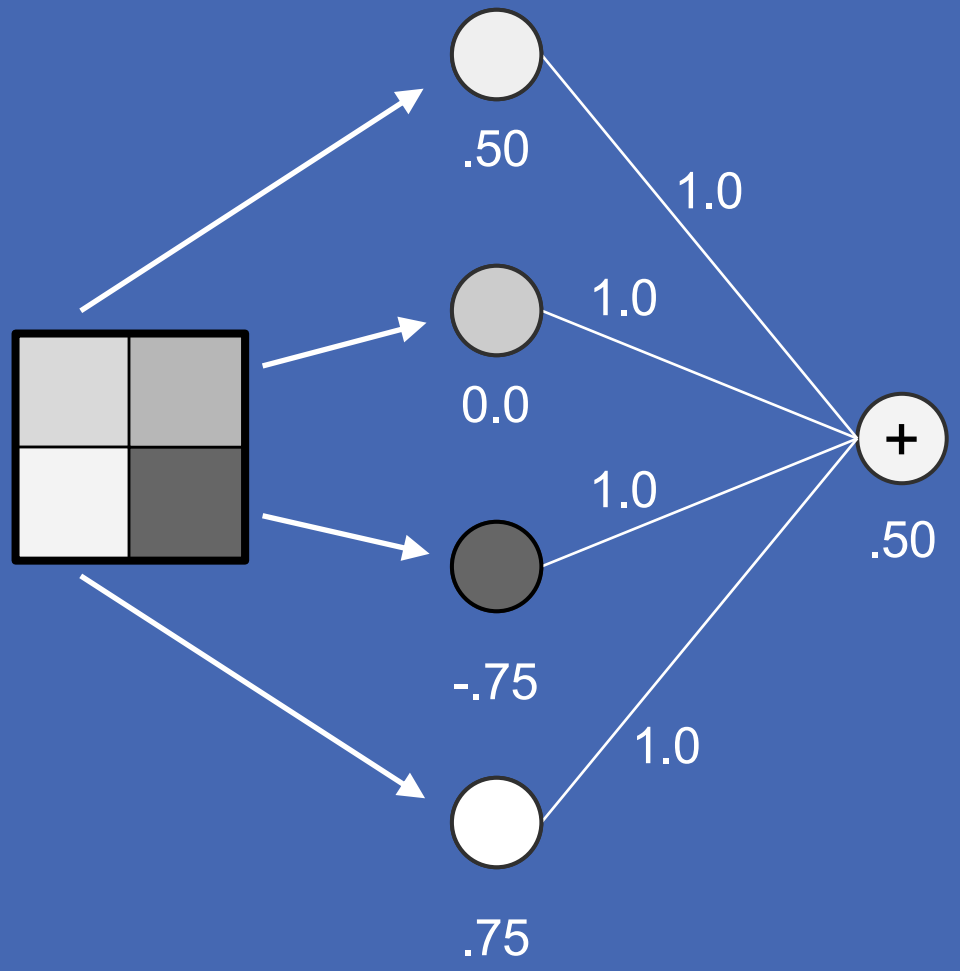
Sum all the inputs



.50
0.00
-.75
\+ .75

.50

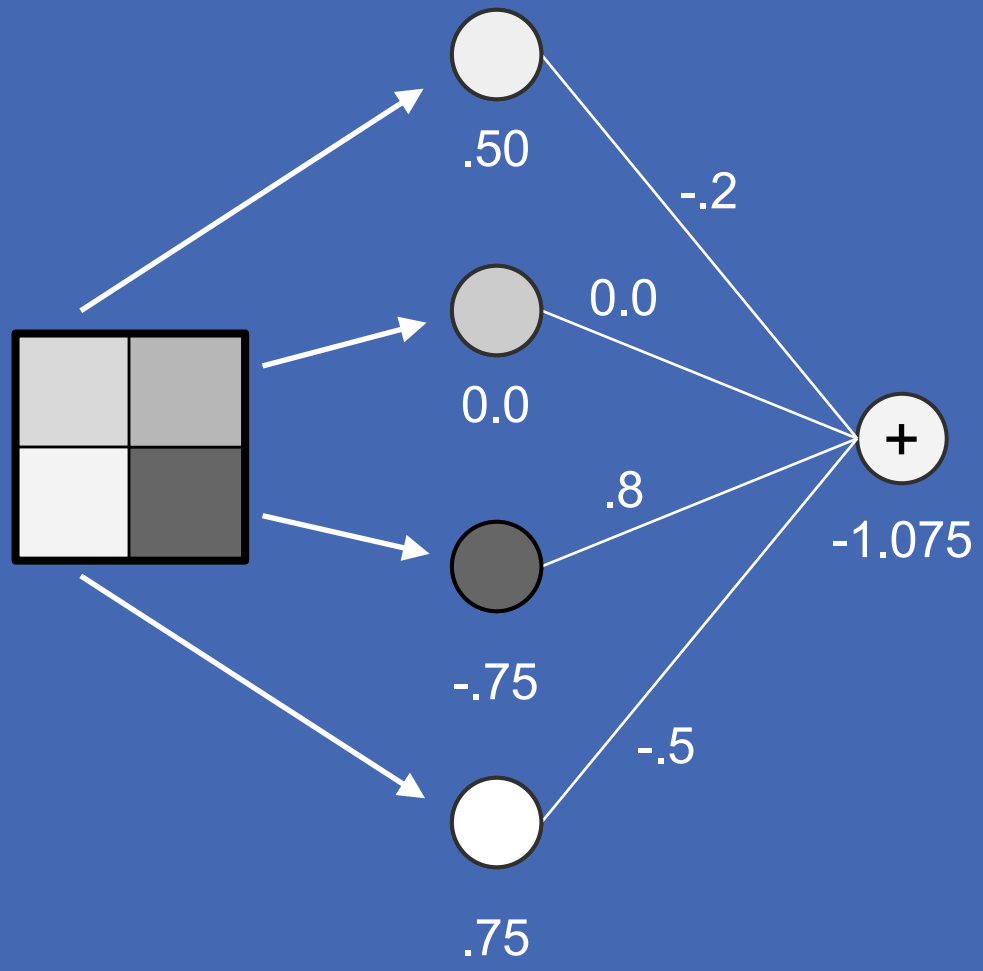
Weights



.50 x 1.0
0.00 x 1.0
-.75 x 1.0
\+ .75 x 1.0

.50

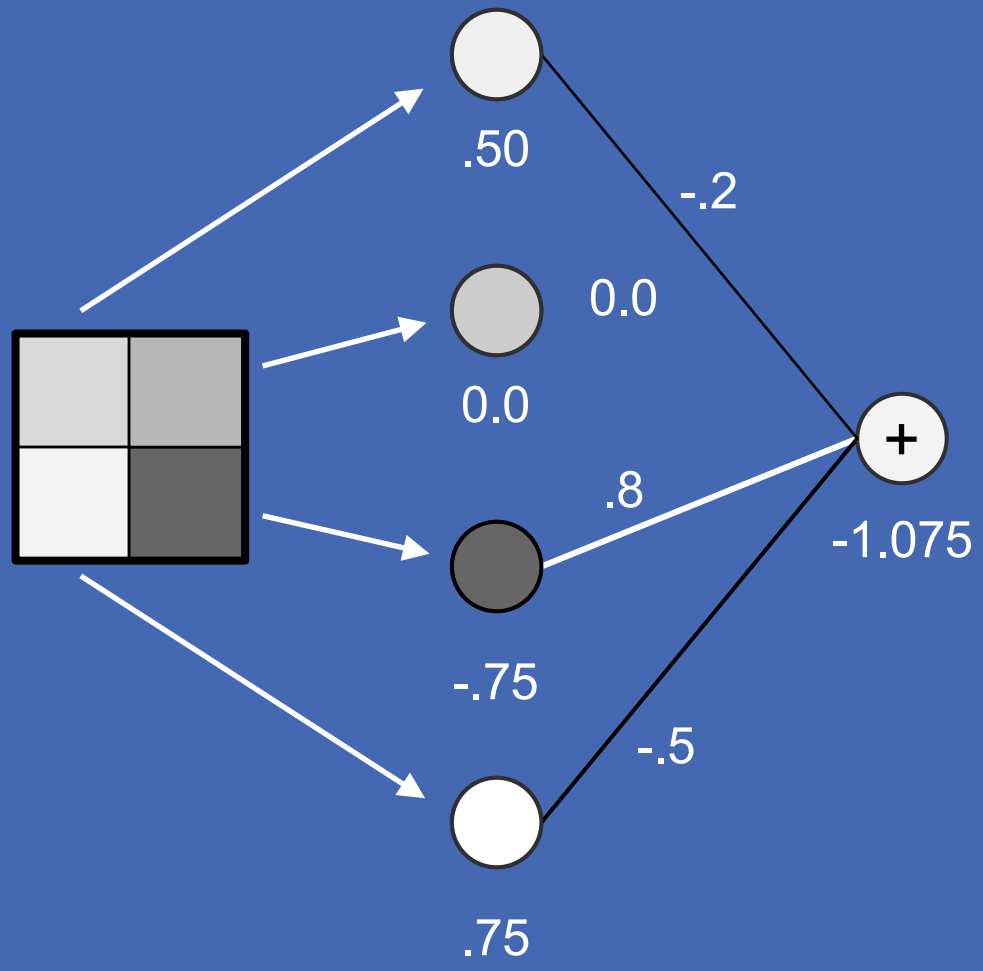
Weights



.50 x -.2
0.00 x 0.0
-.75 x .8
\+ .75 x -.5

-1.075

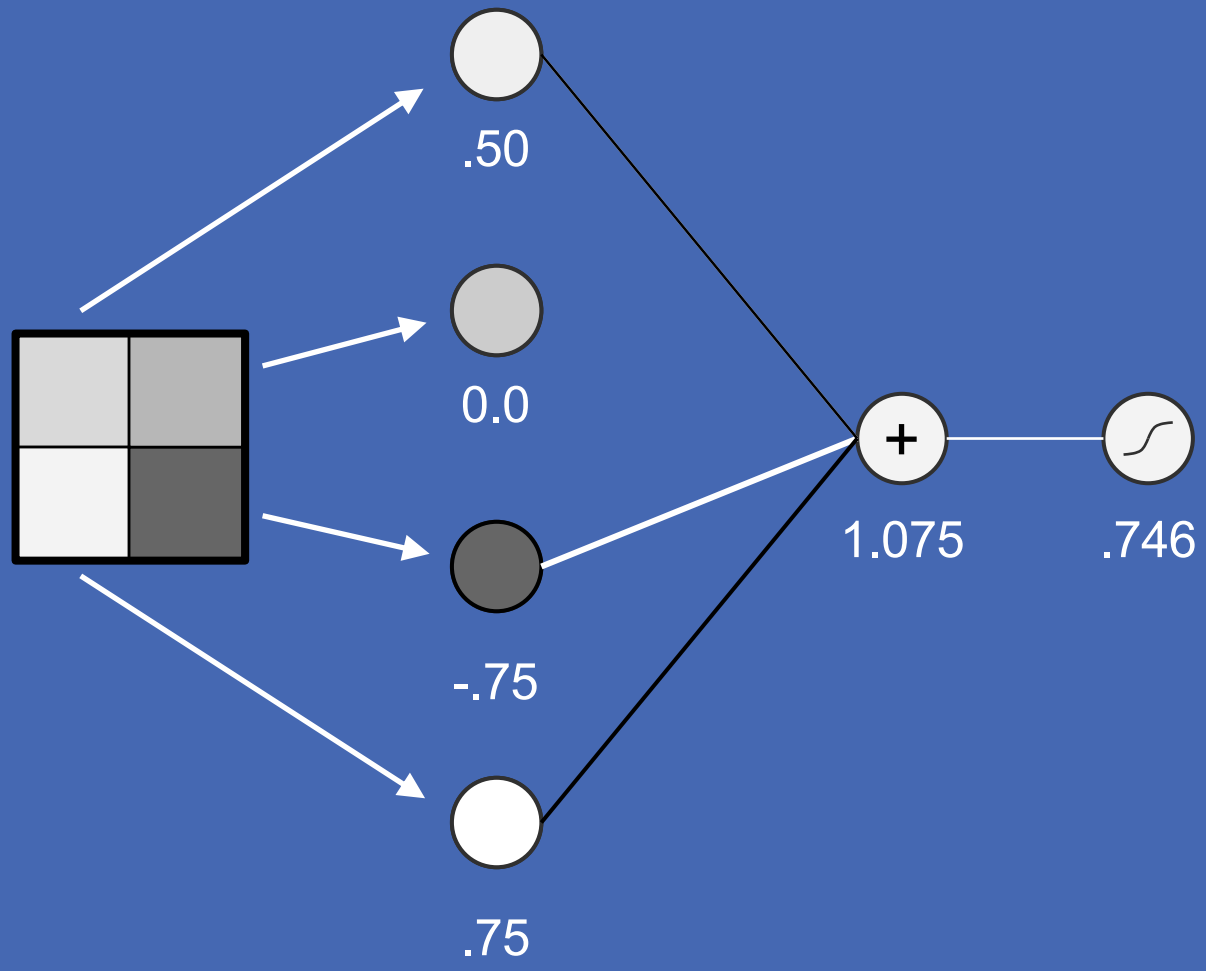
Weights



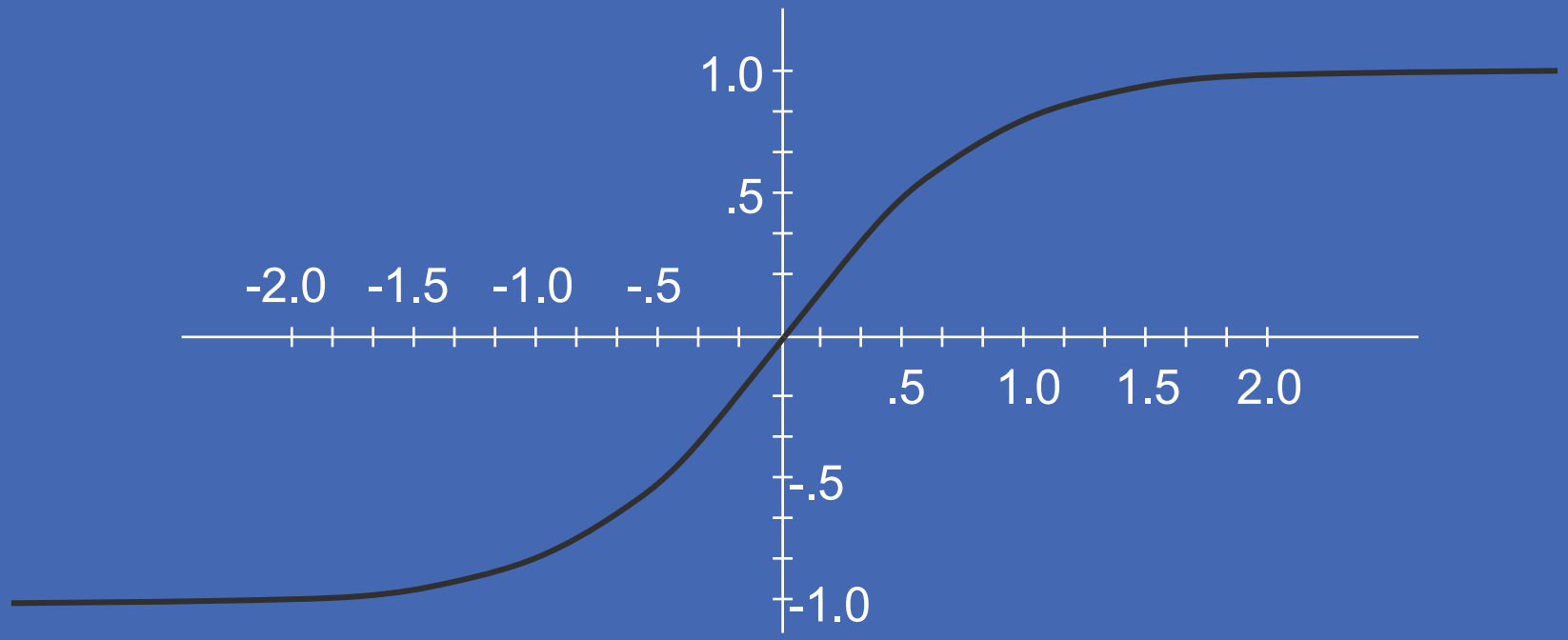
.50 x -.2
0.00 x 0.0
-.75 x .8
\+ .75 x -.5

-1.075

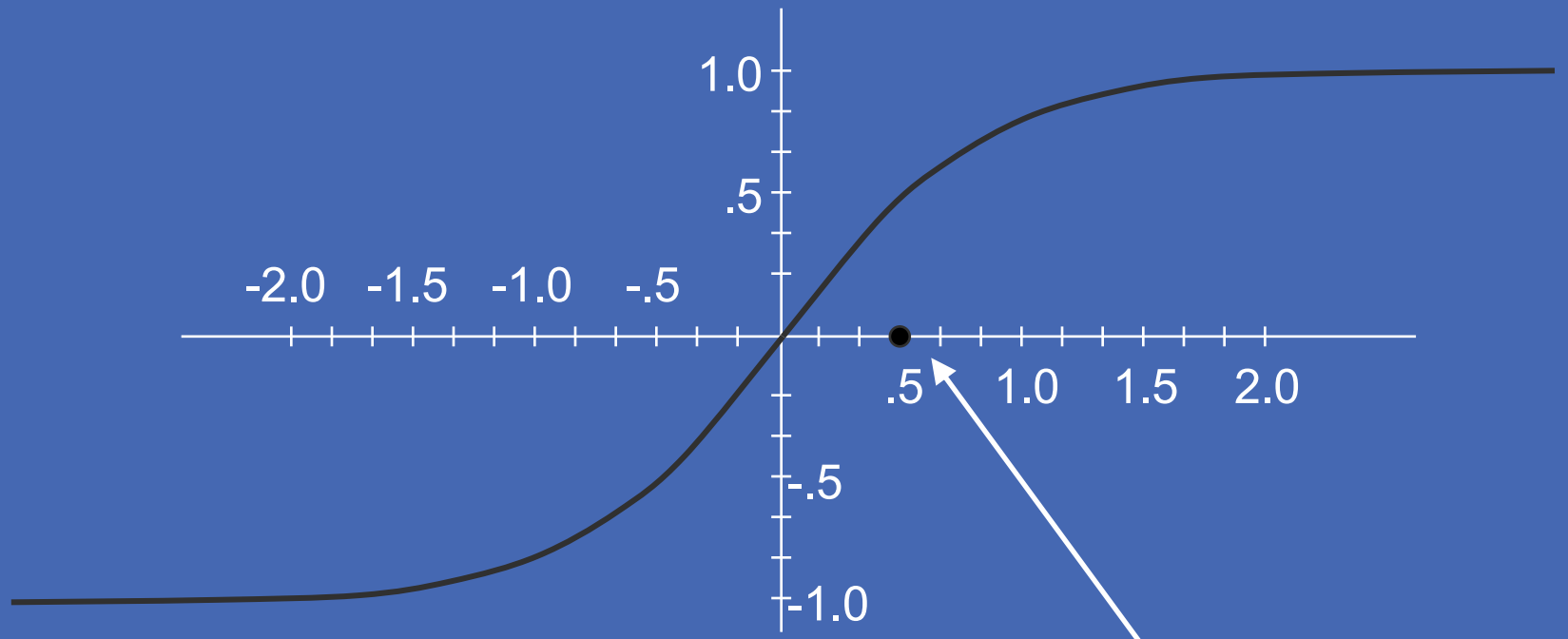
Squash the result



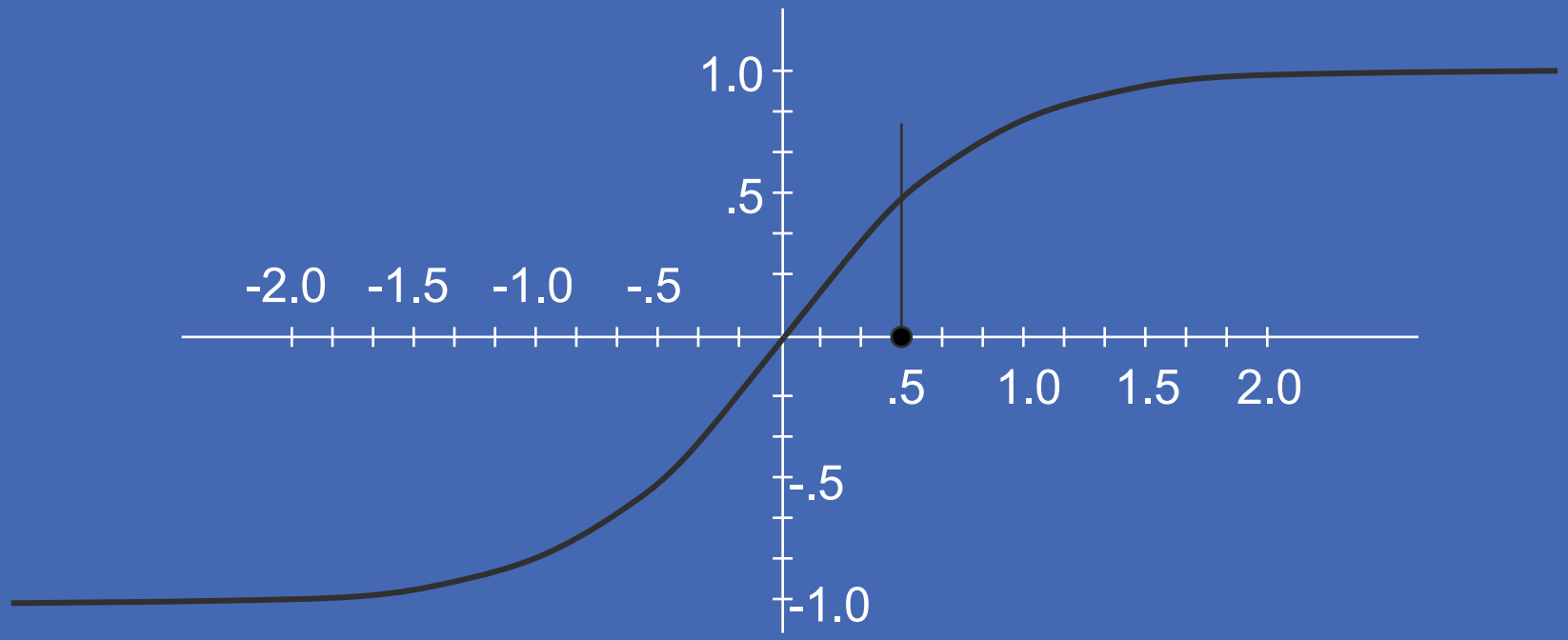
Sigmoid squashing function



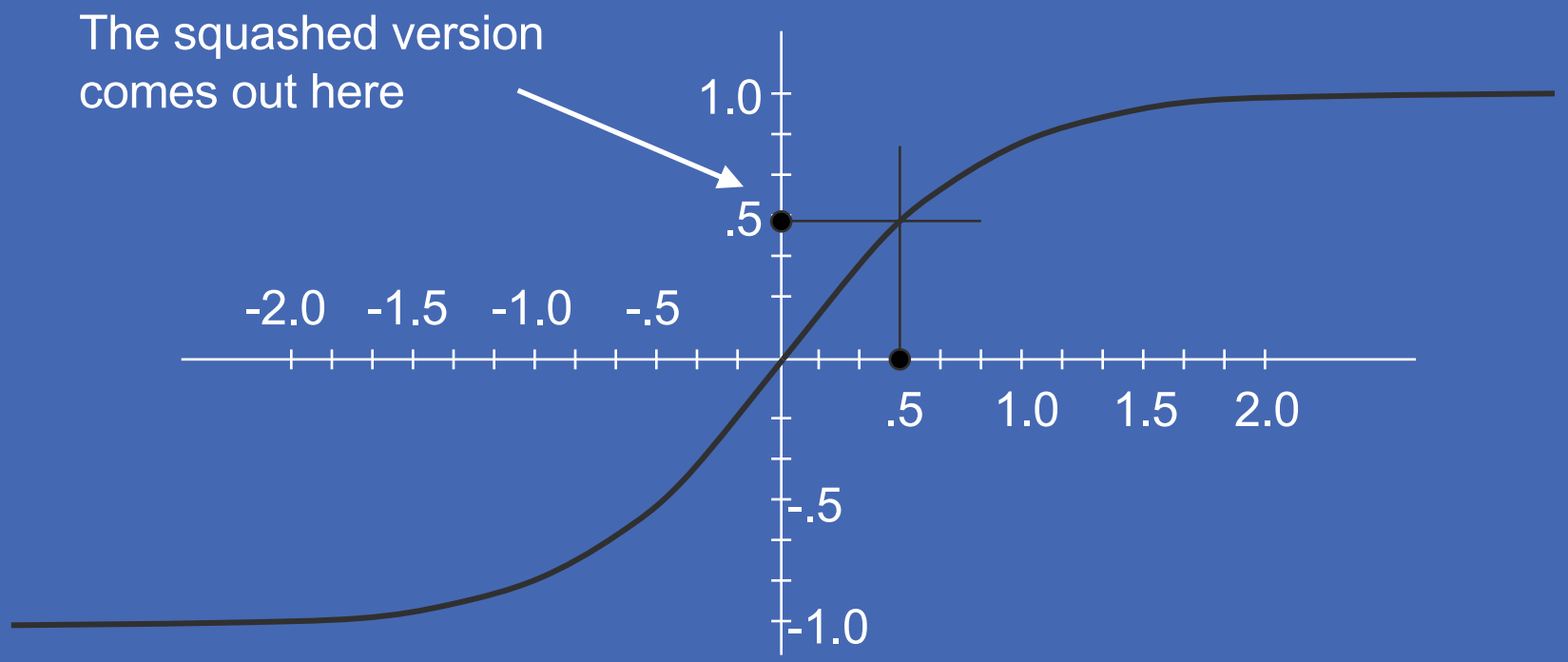
Sigmoid squashing function



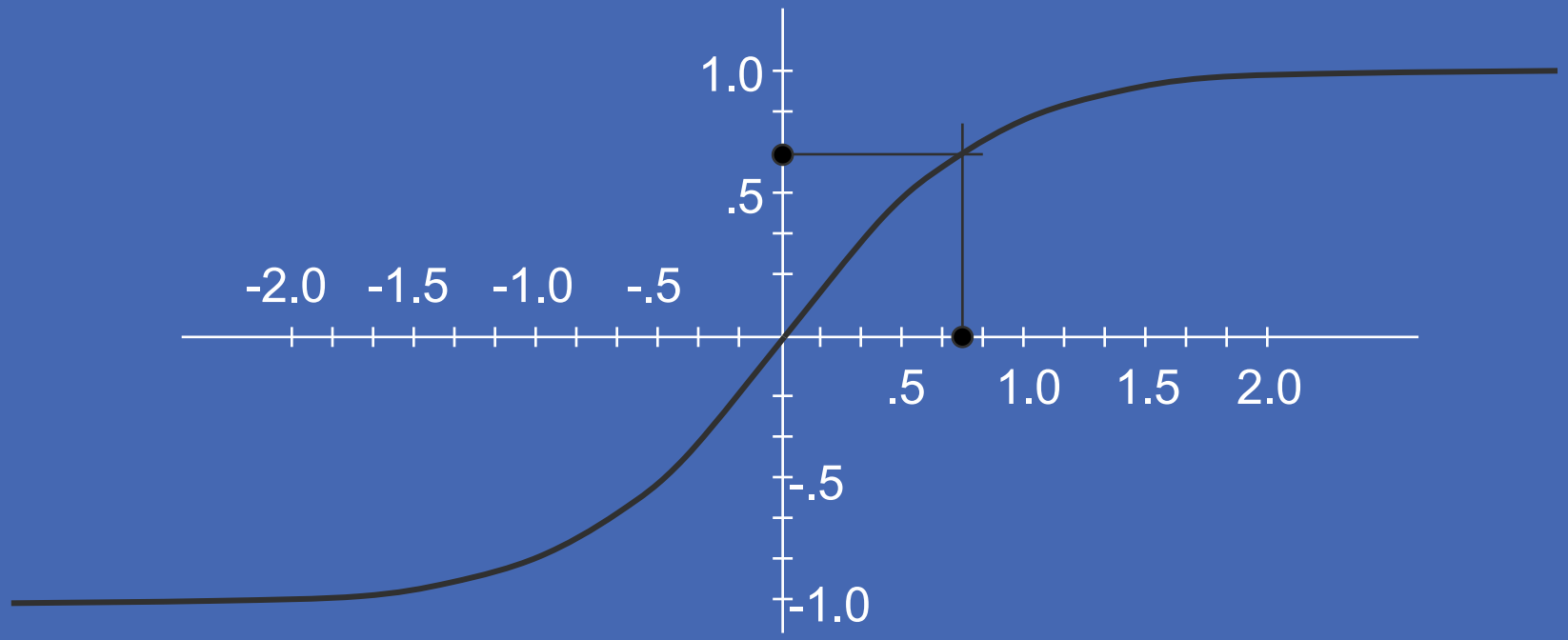
Sigmoid squashing function



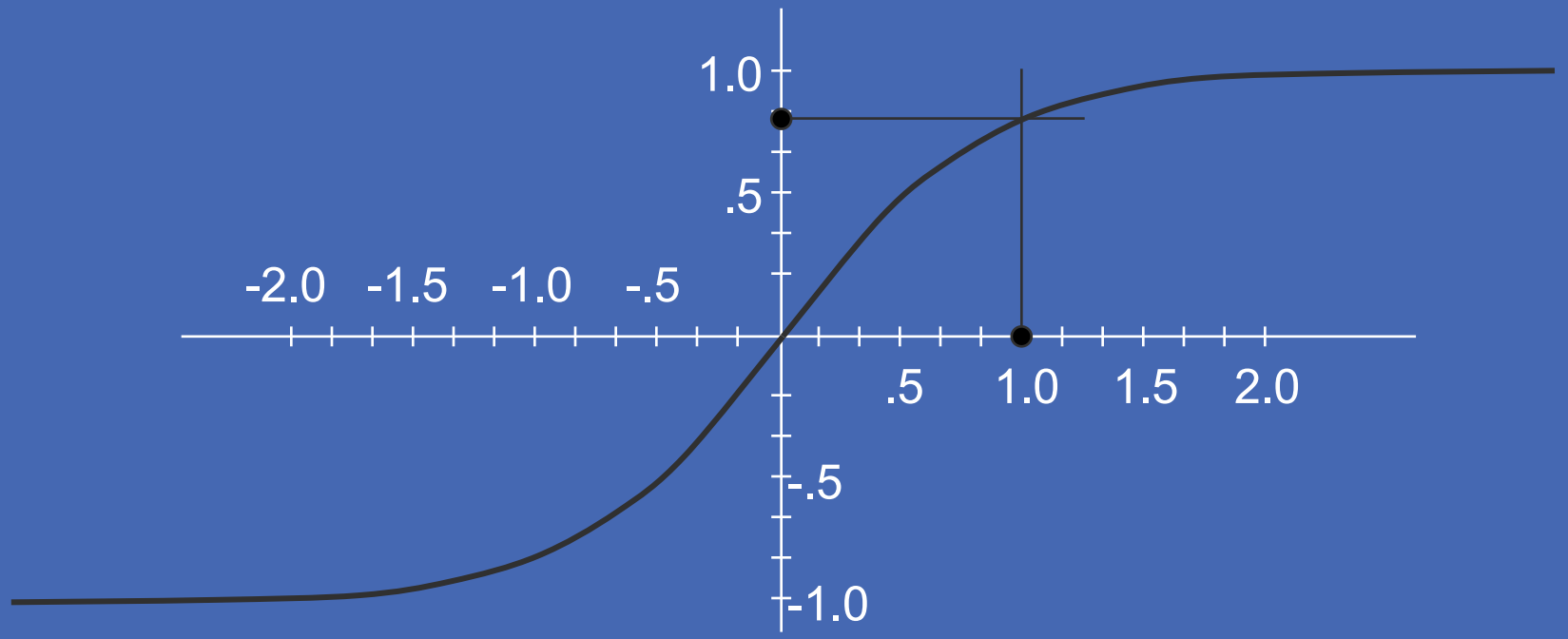
Sigmoid squashing function



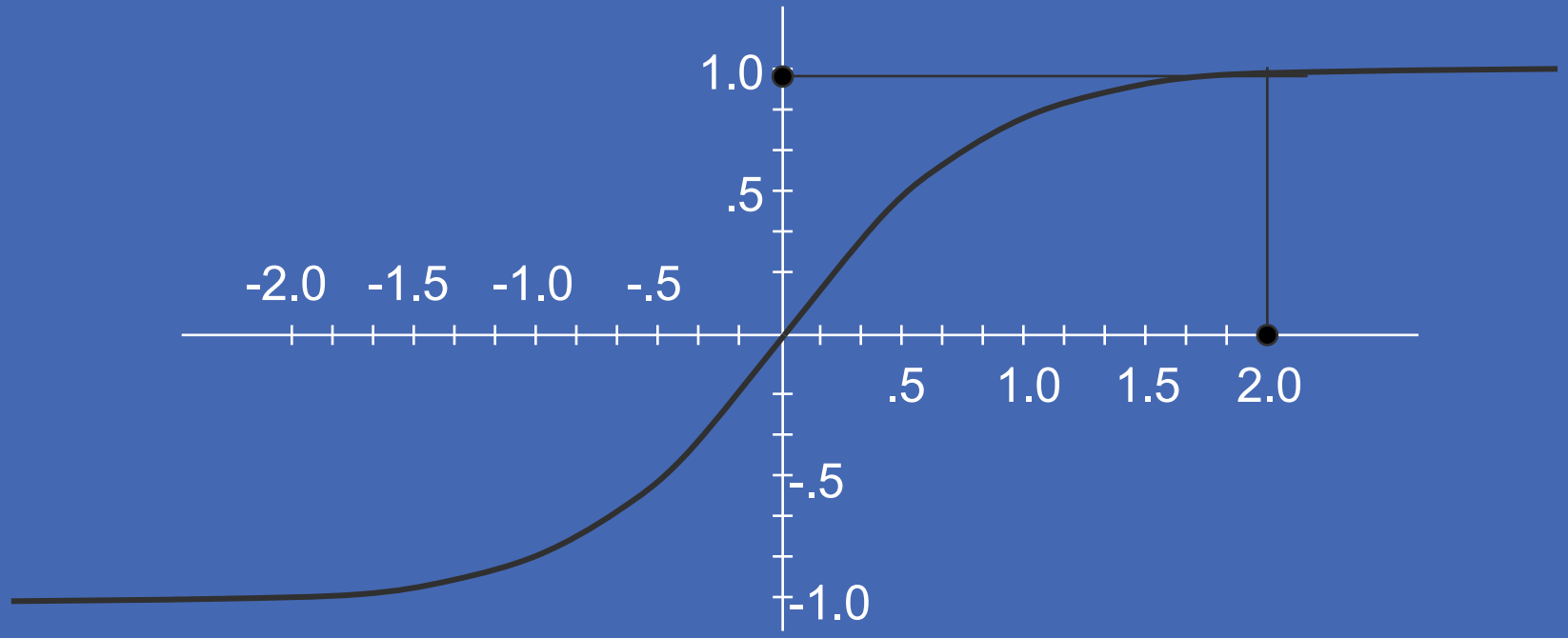
Sigmoid squashing function



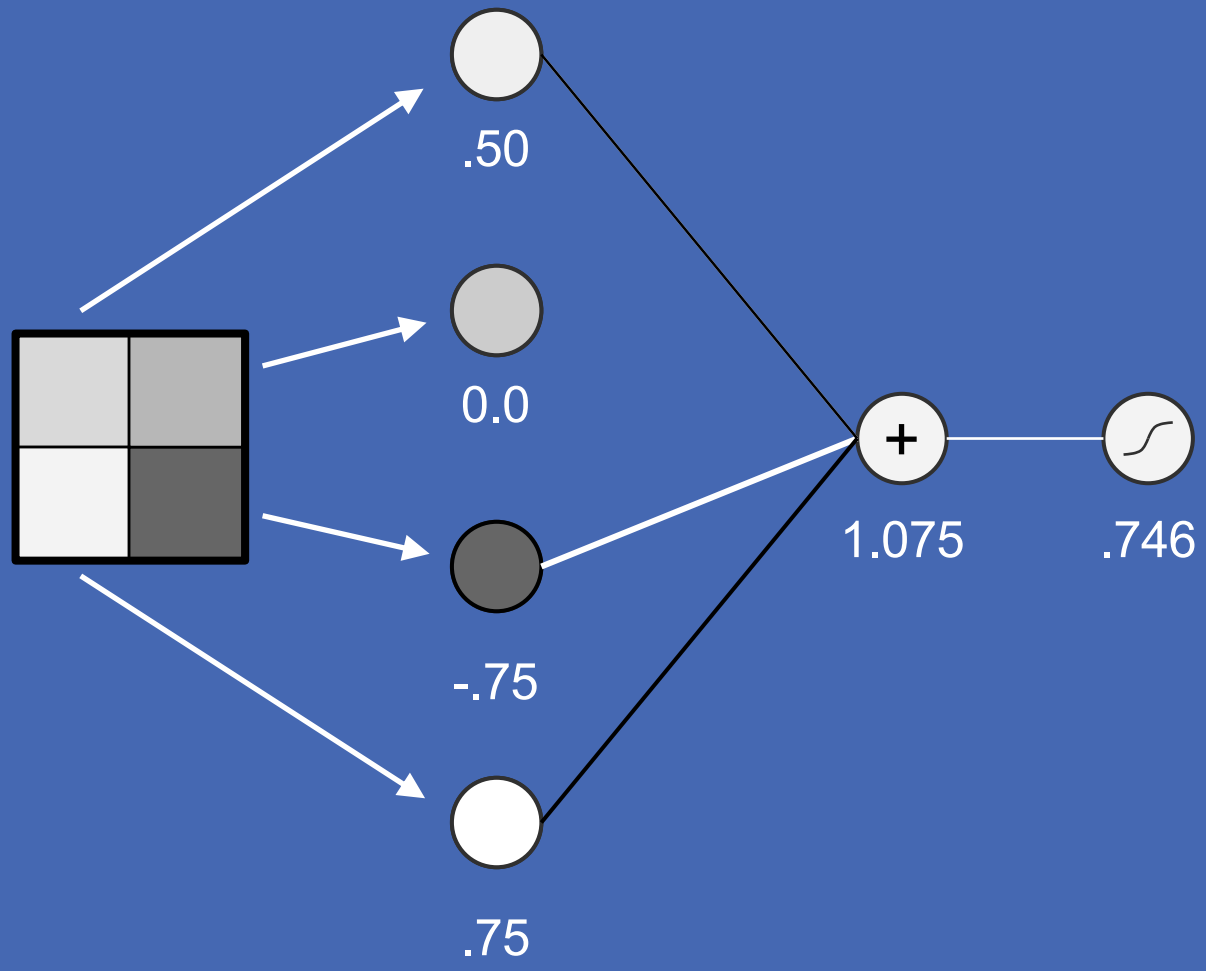
Sigmoid squashing function



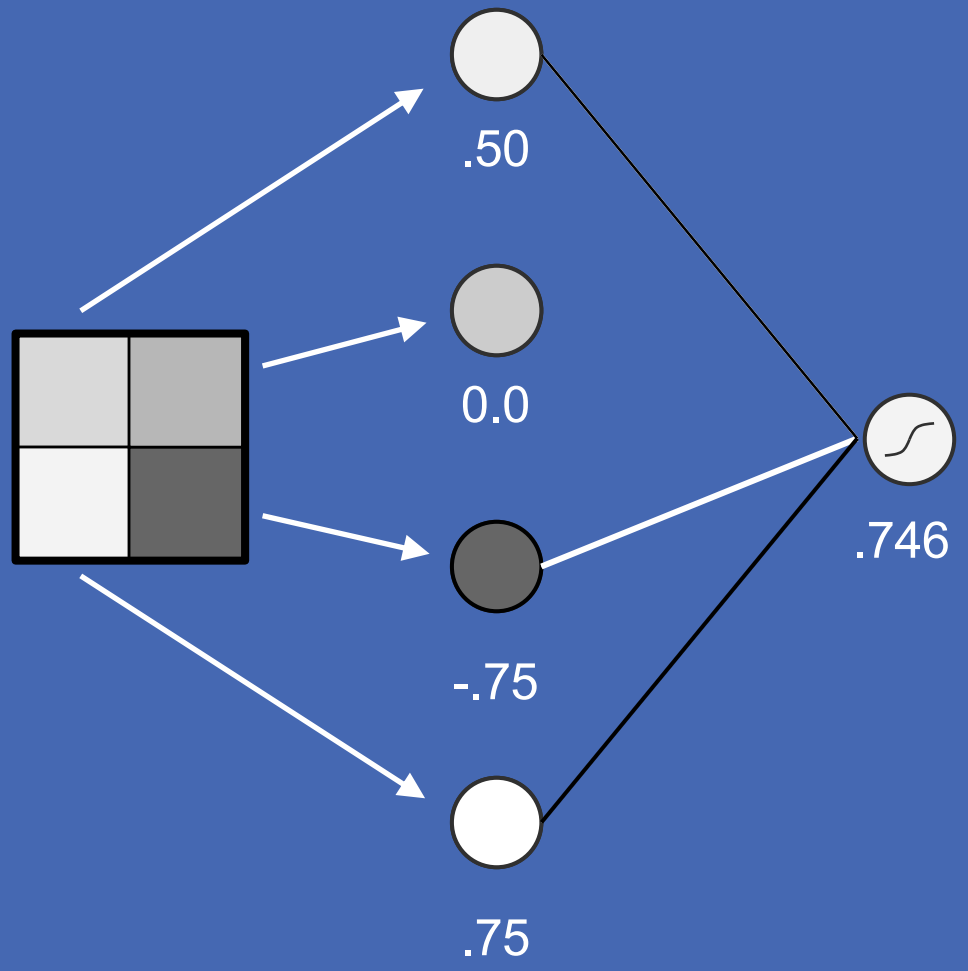
No matter what you start with, the answer stays between -1 and 1.



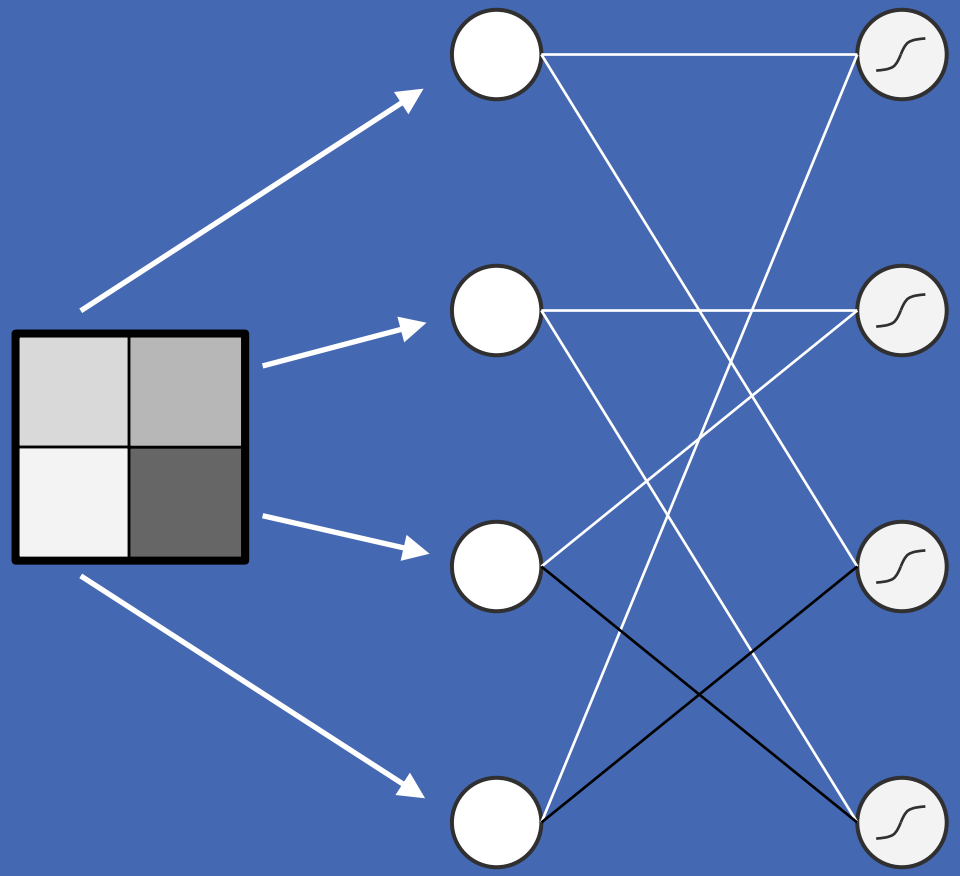
Squash the result



Weighted sum-and-squash neuron

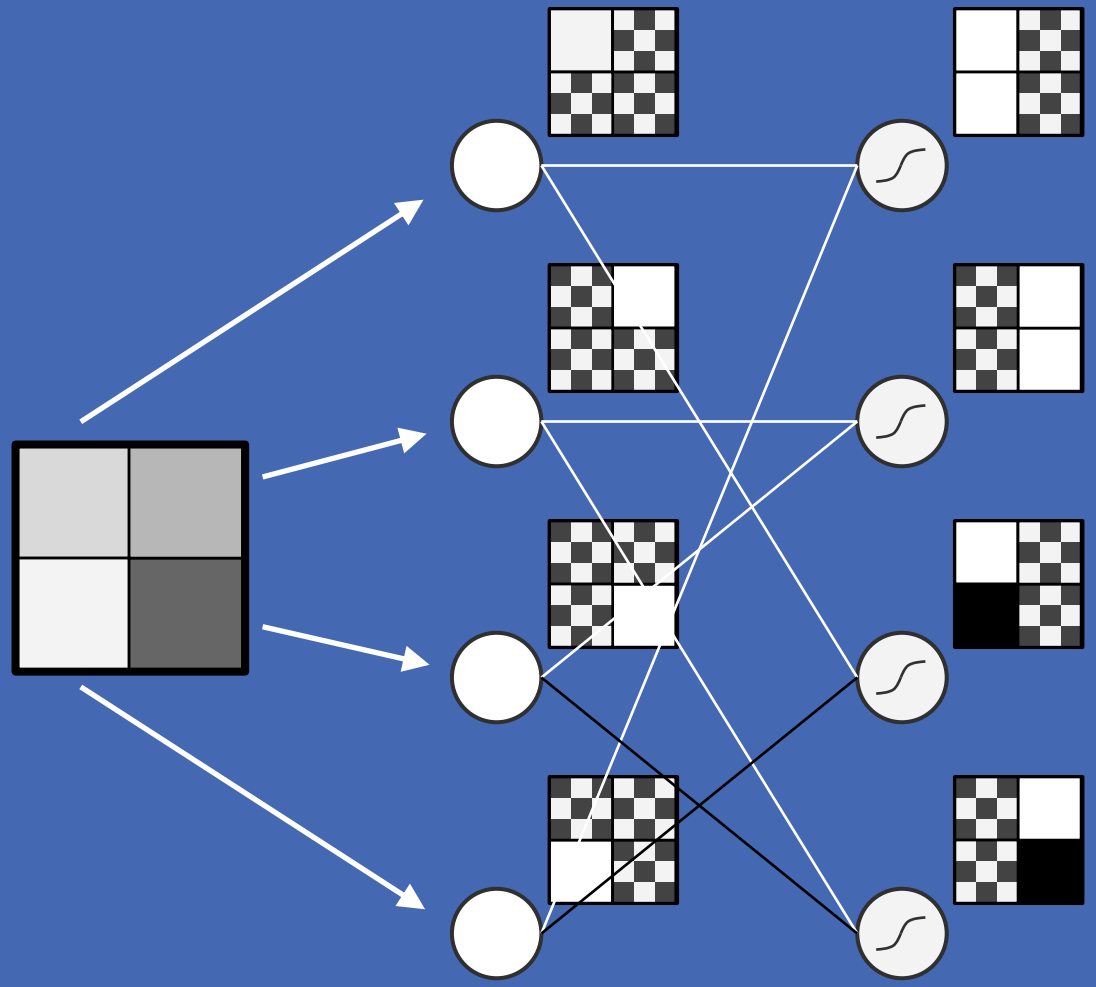


Make lots of neurons, identical except for weights

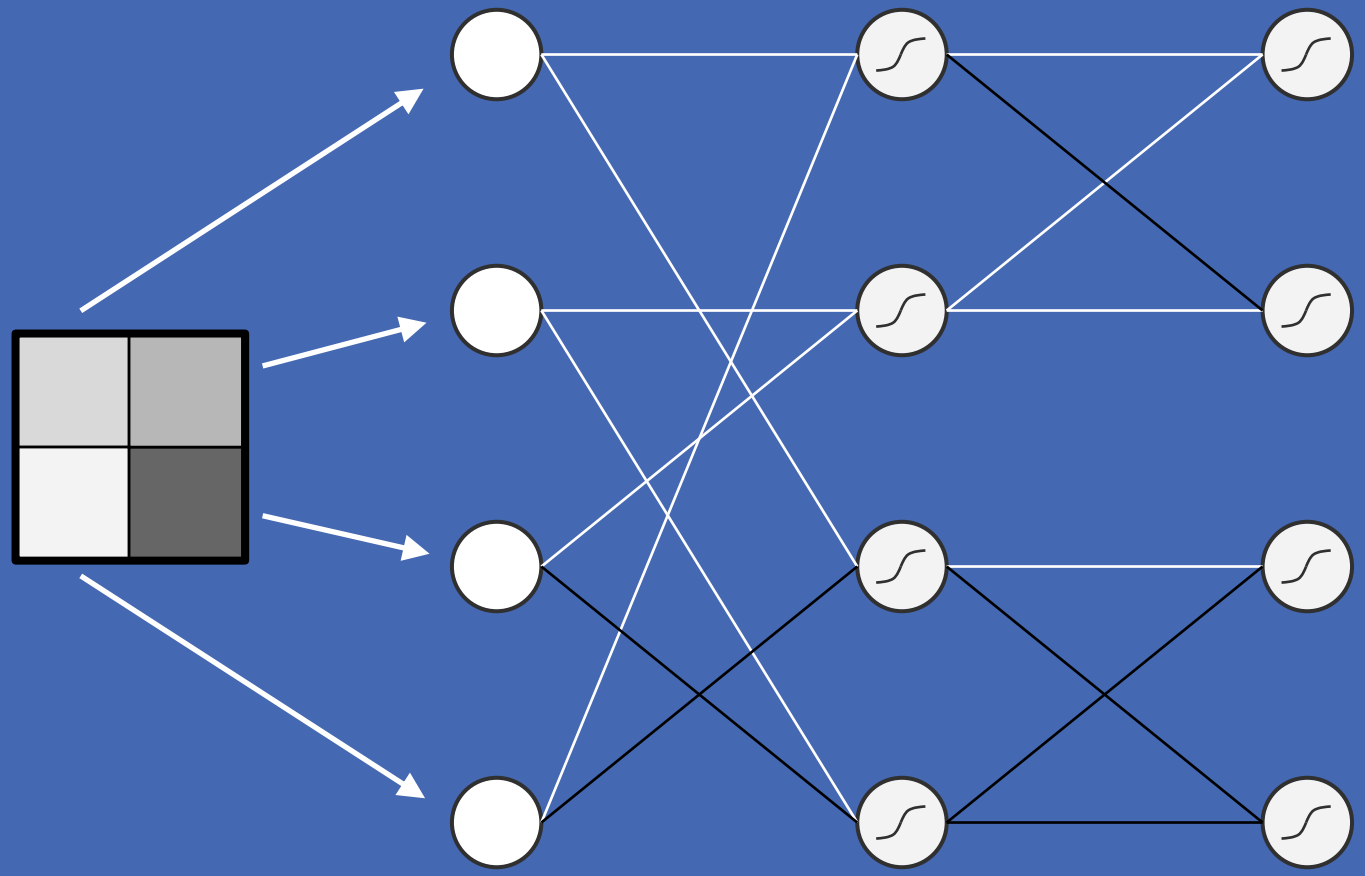


To keep our picture clear, weights will either be
1.0 (white)
-1.0 (black) or
0.0 (missing)

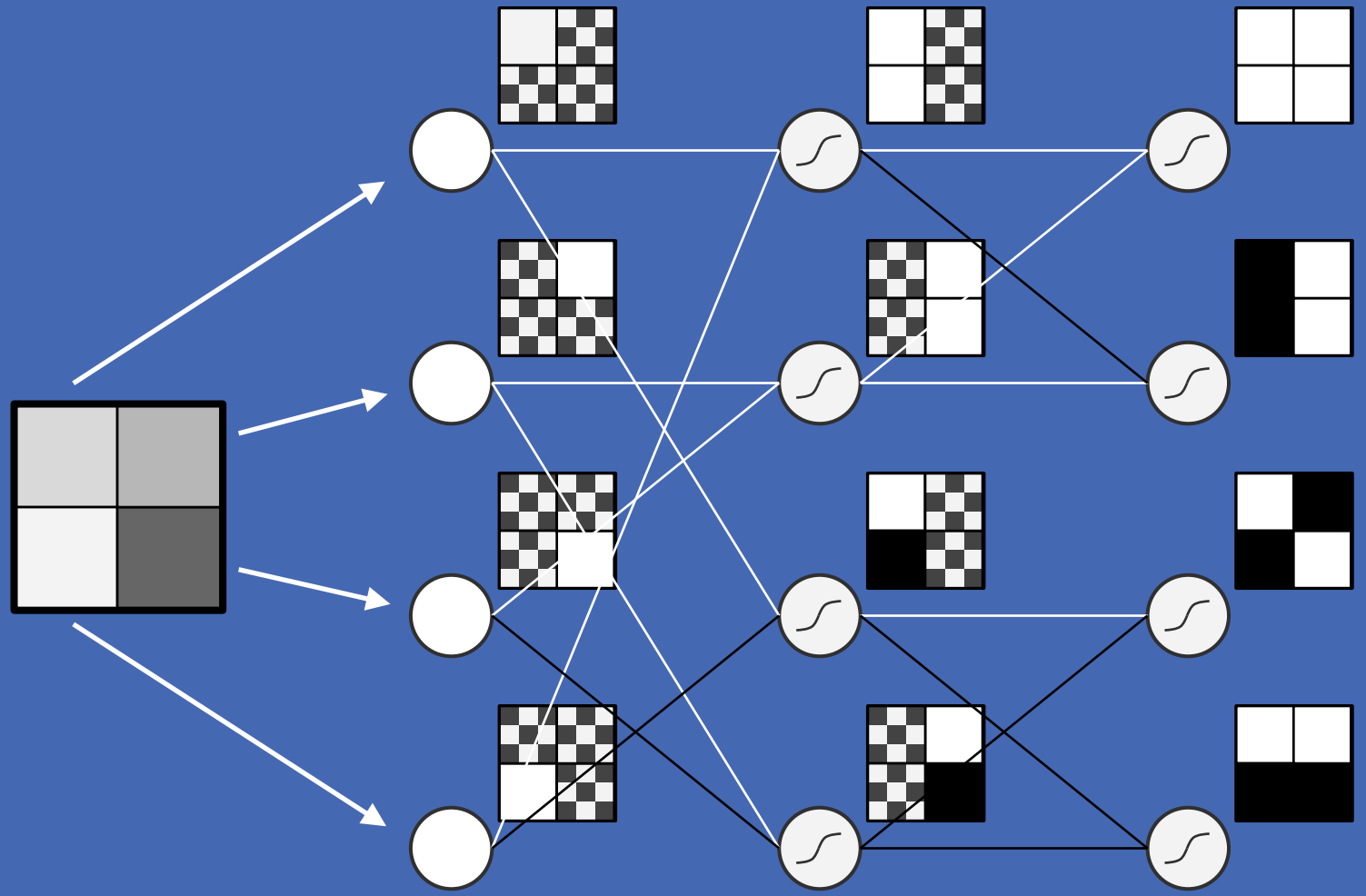
Receptive fields get more complex



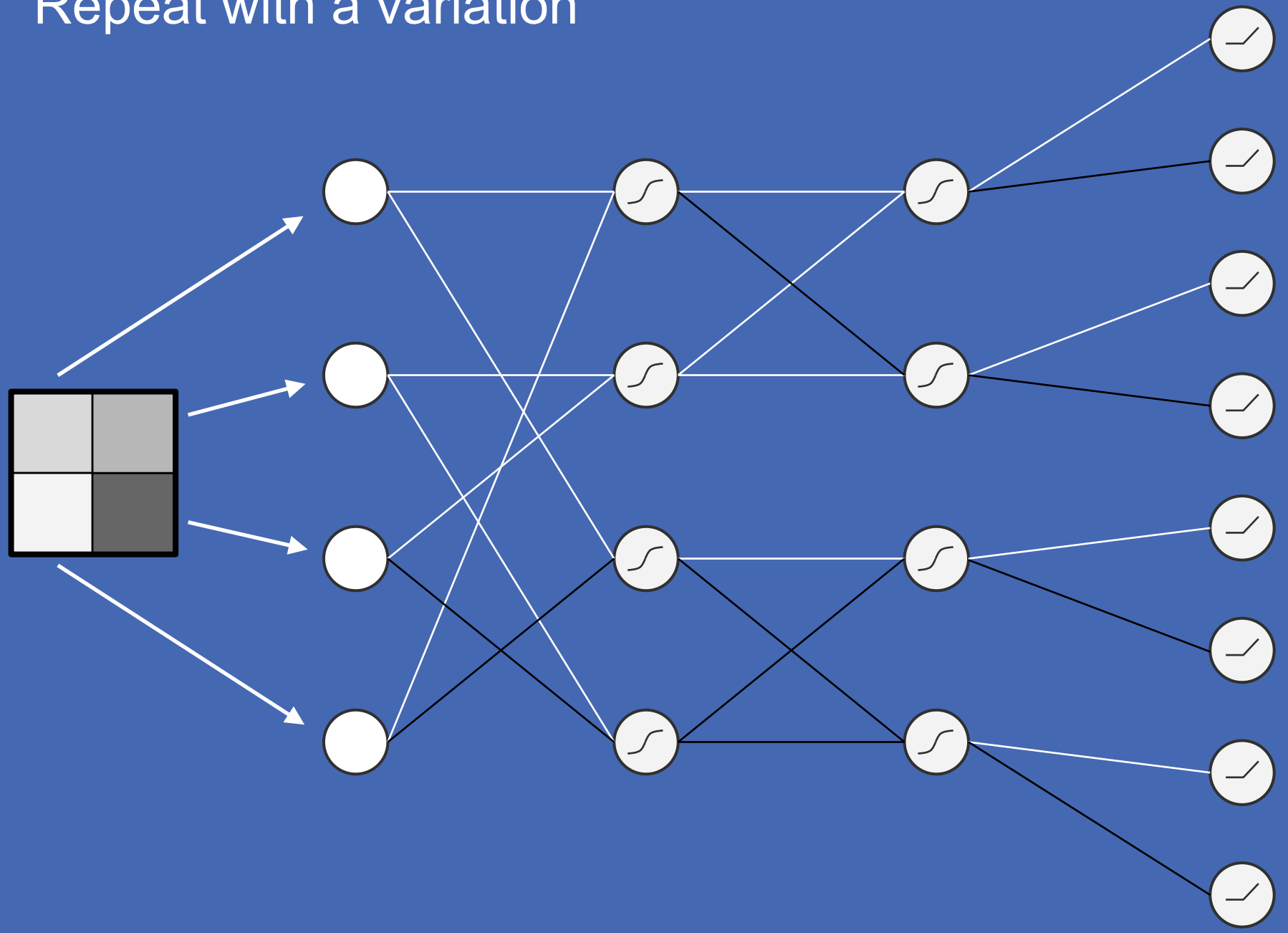
Repeat for additional layers



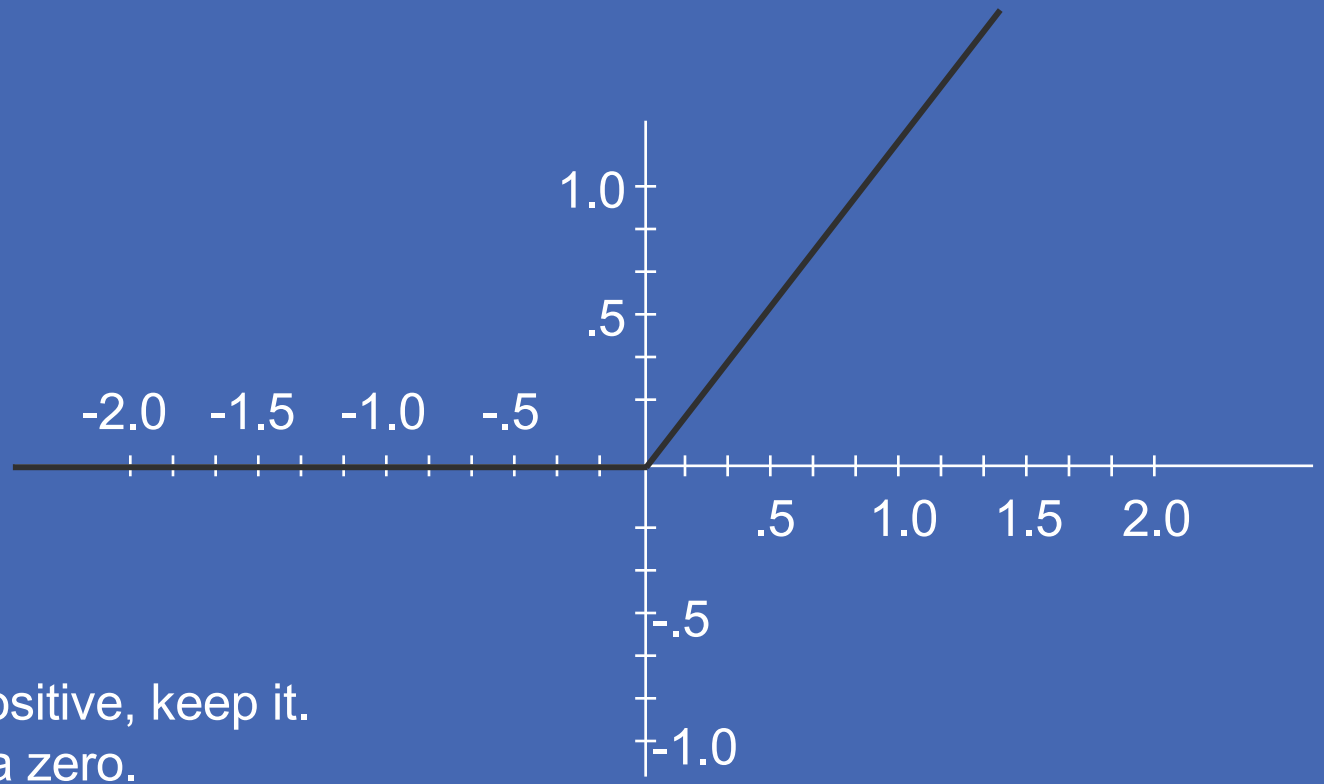
Receptive fields get still more complex



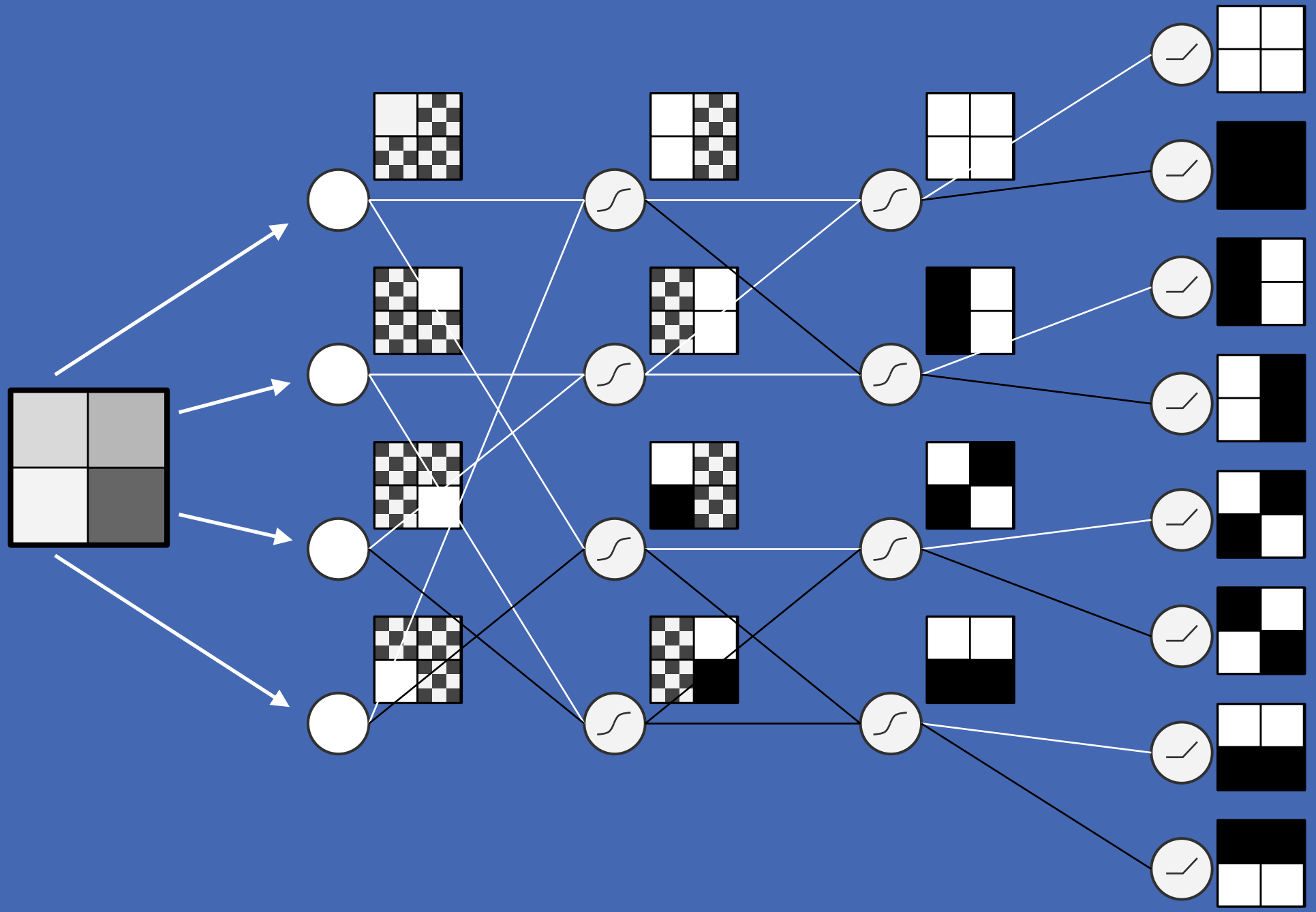
Repeat with a variation



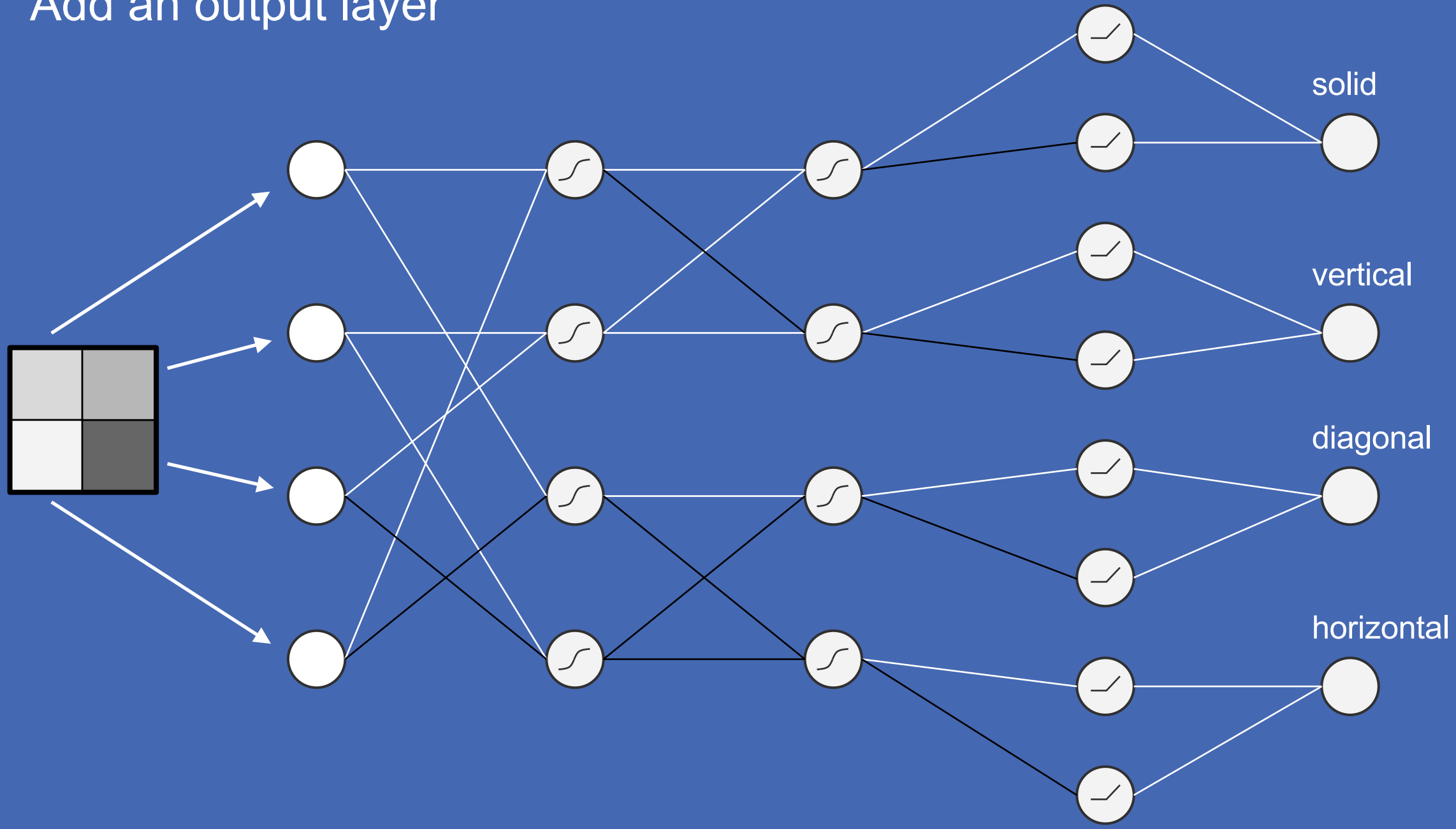
Rectified linear units (ReLUs)

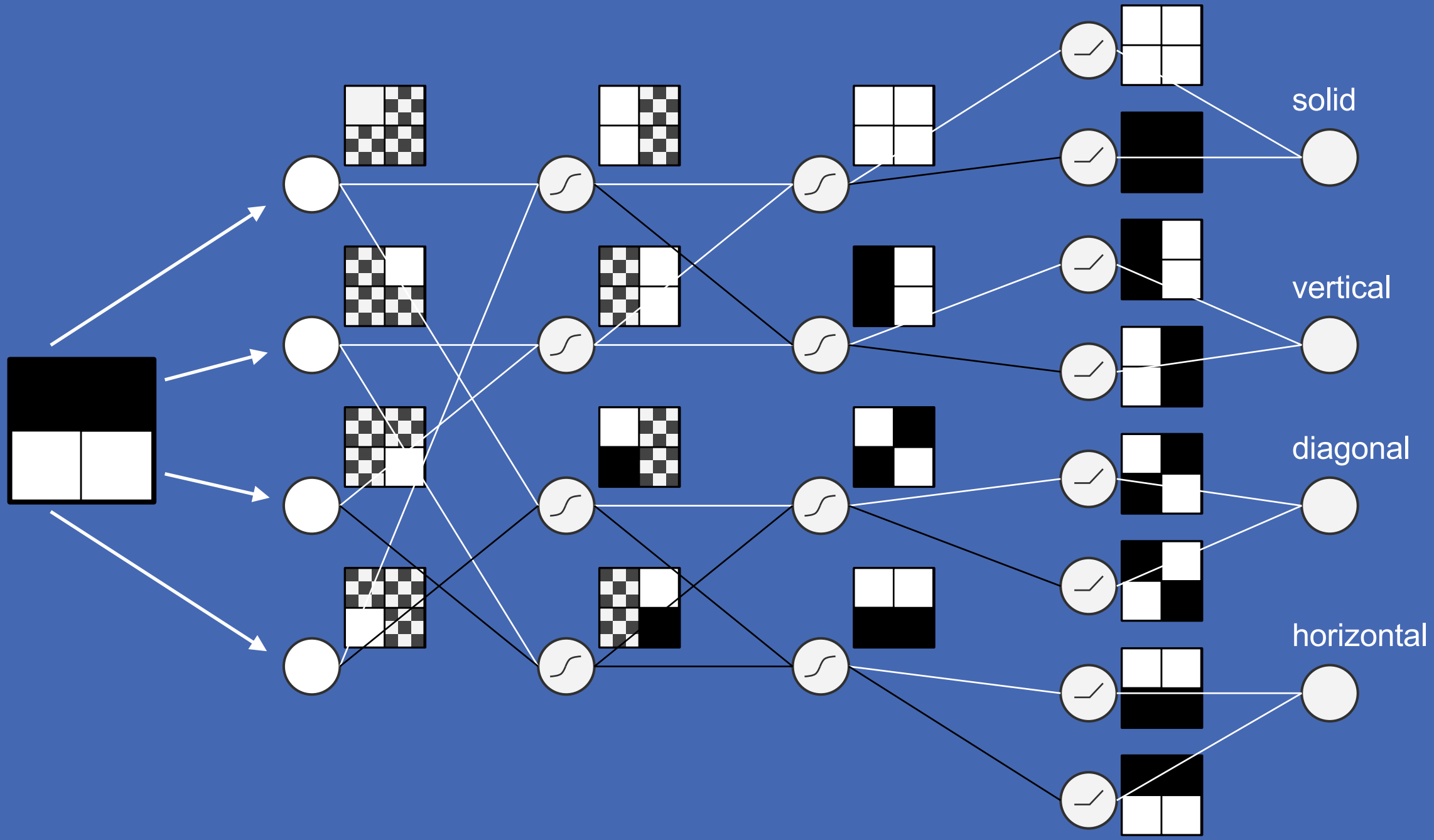


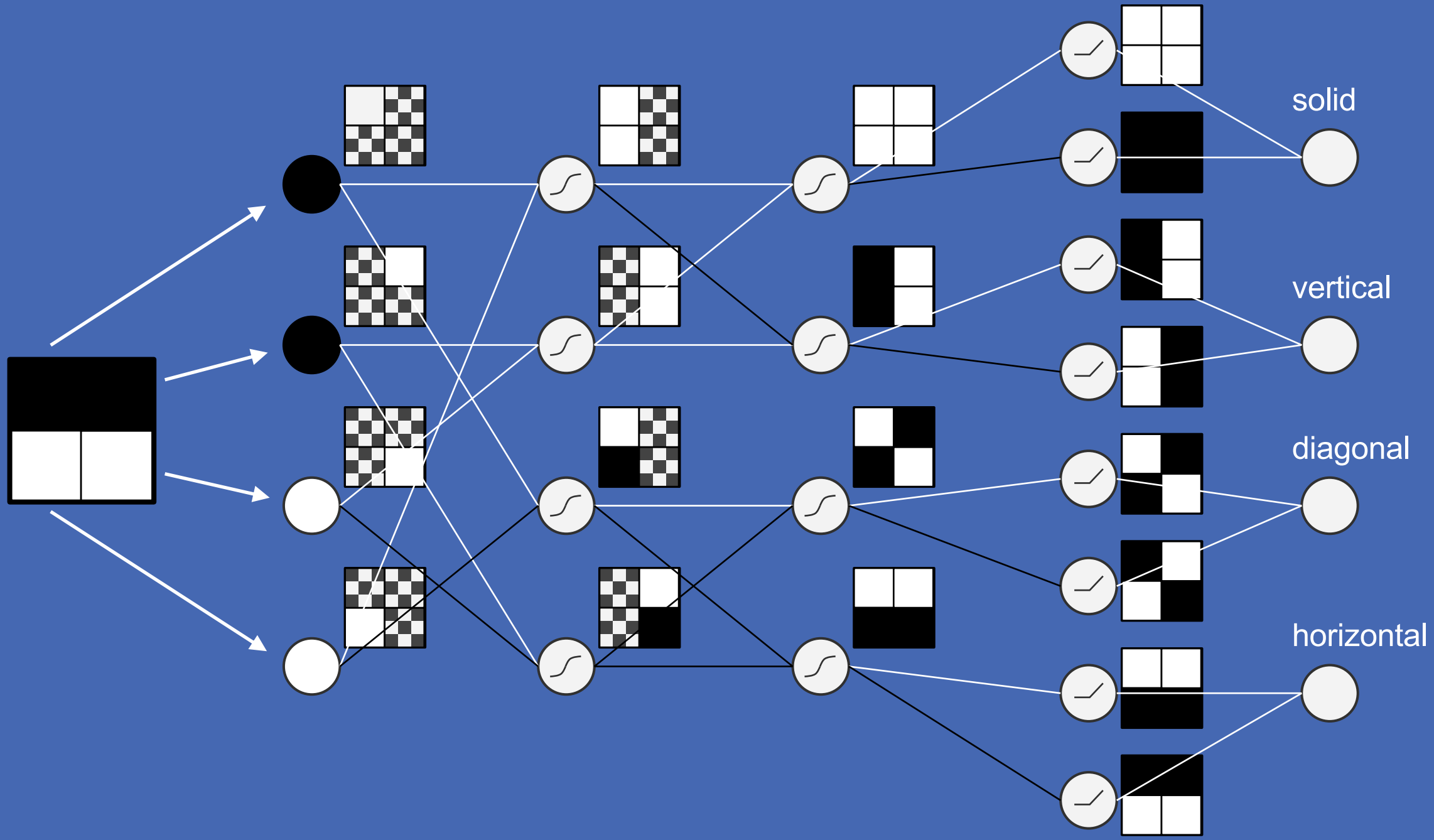
If your number is positive, keep it.
Otherwise you get a zero.

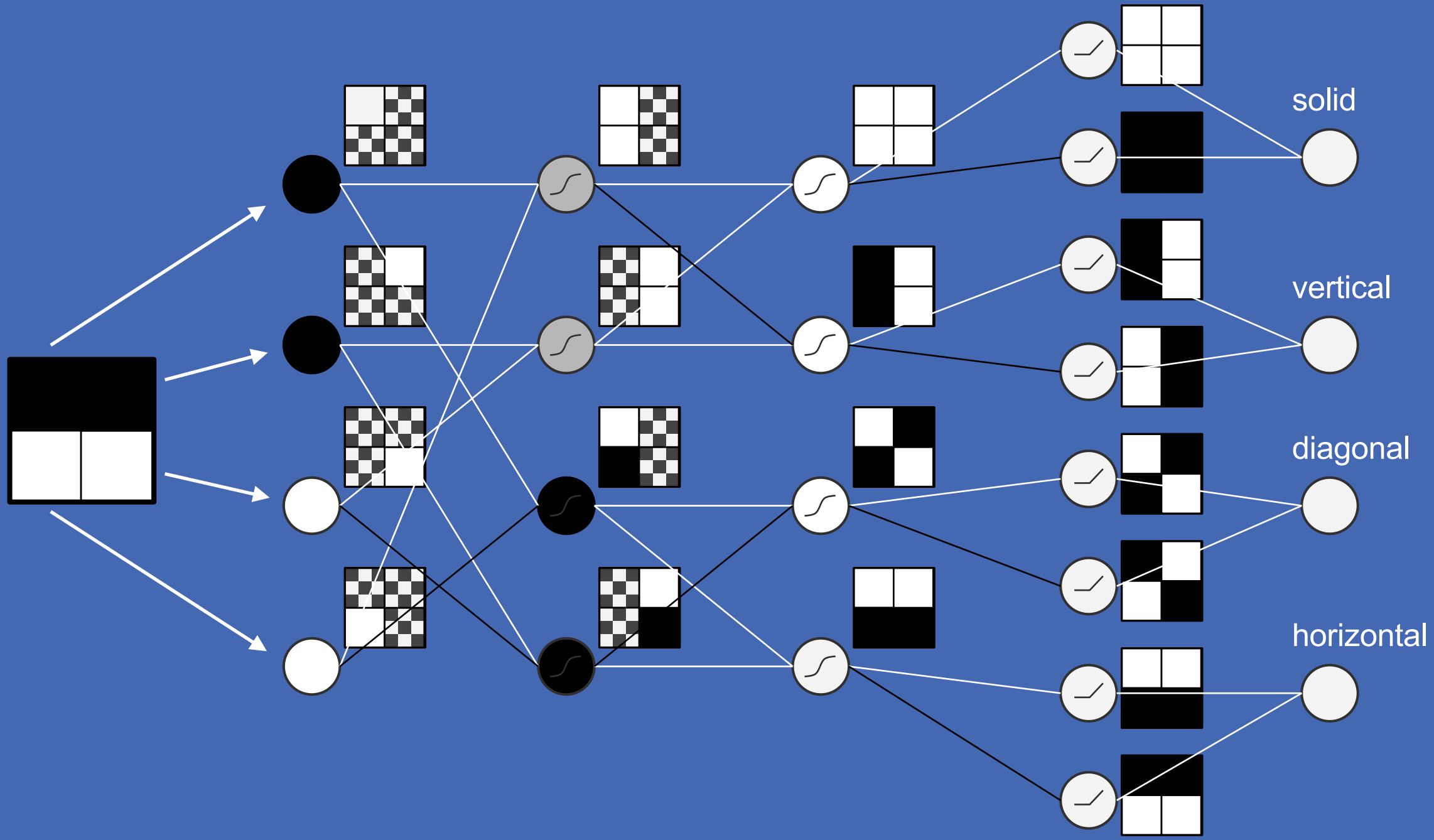


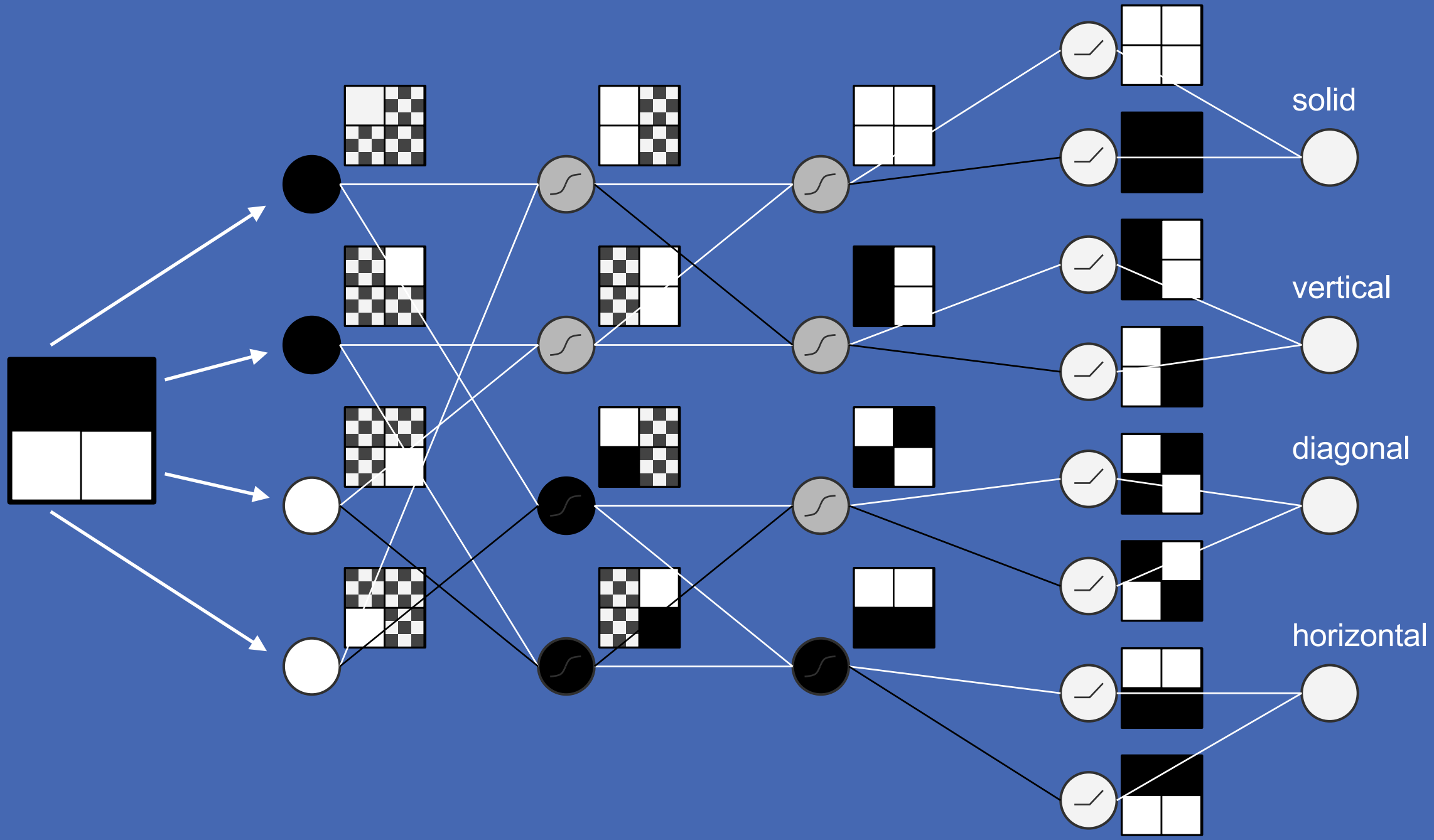
Add an output layer

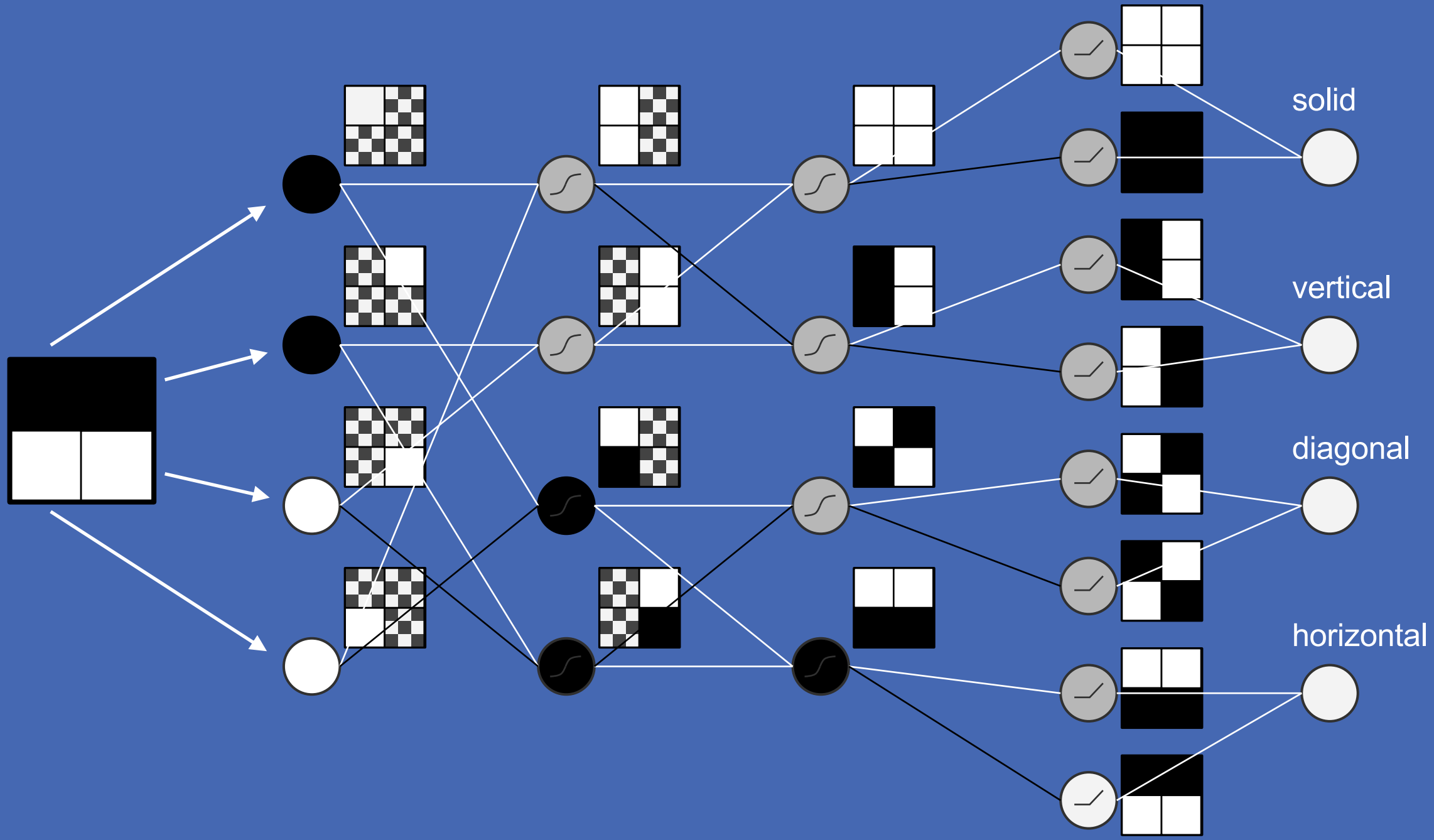


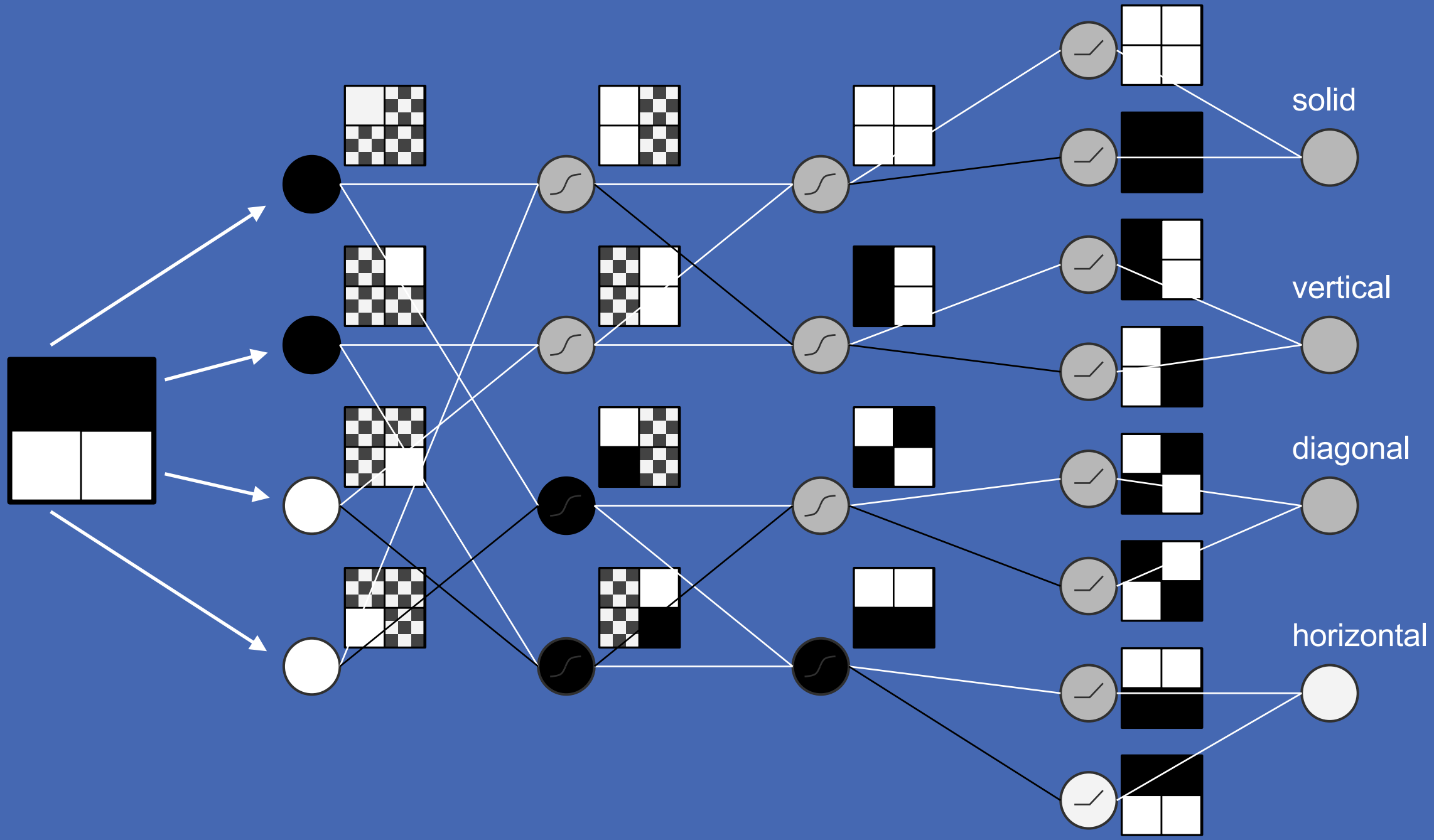

solid
vertical
diagonal
horizontal

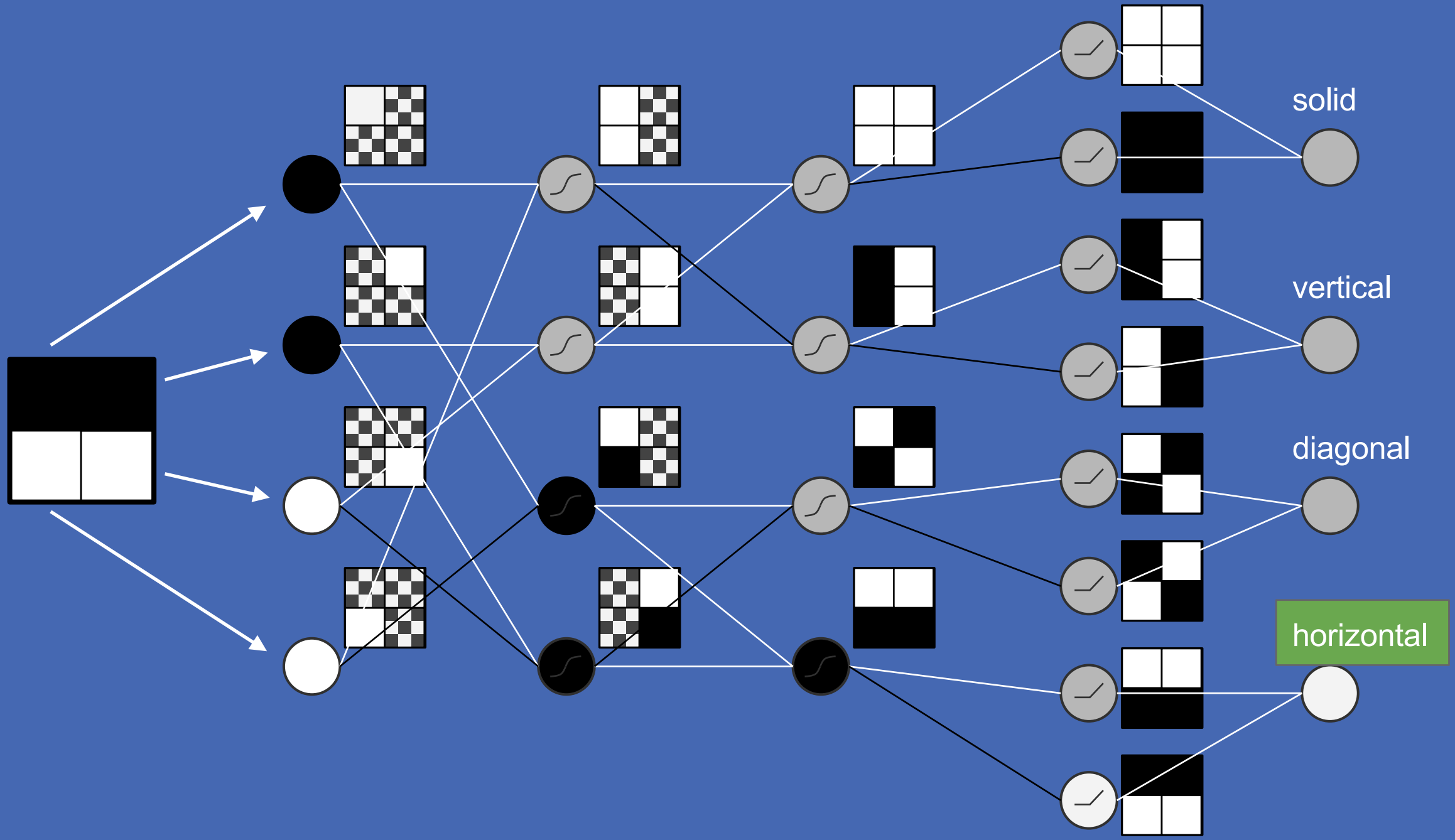

solid
vertical
diagonal
horizontal


solid
vertical
diagonal
horizontal

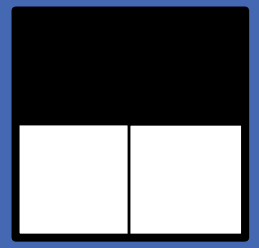

solid
vertical
diagonal
horizontal


solid
vertical
diagonal
horizontal


solid
vertical
diagonal
horizontal


solid
vertical
diagonal
horizontal

Errors



truth

0.

0.

0.

1.

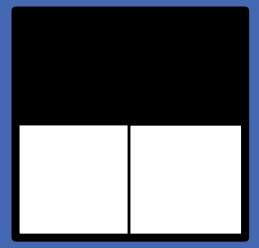
solid

vertical

diagonal

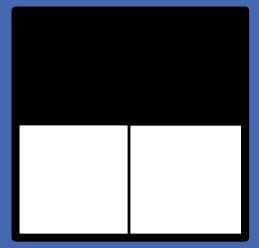
horizontal

Errors



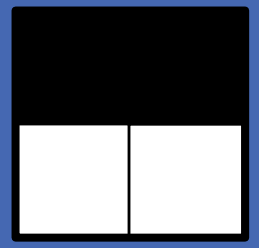
truth	answer	solid
0.	.5	
0.	.75	vertical
0.	-.25	diagonal
1.	-.75	horizontal

Errors



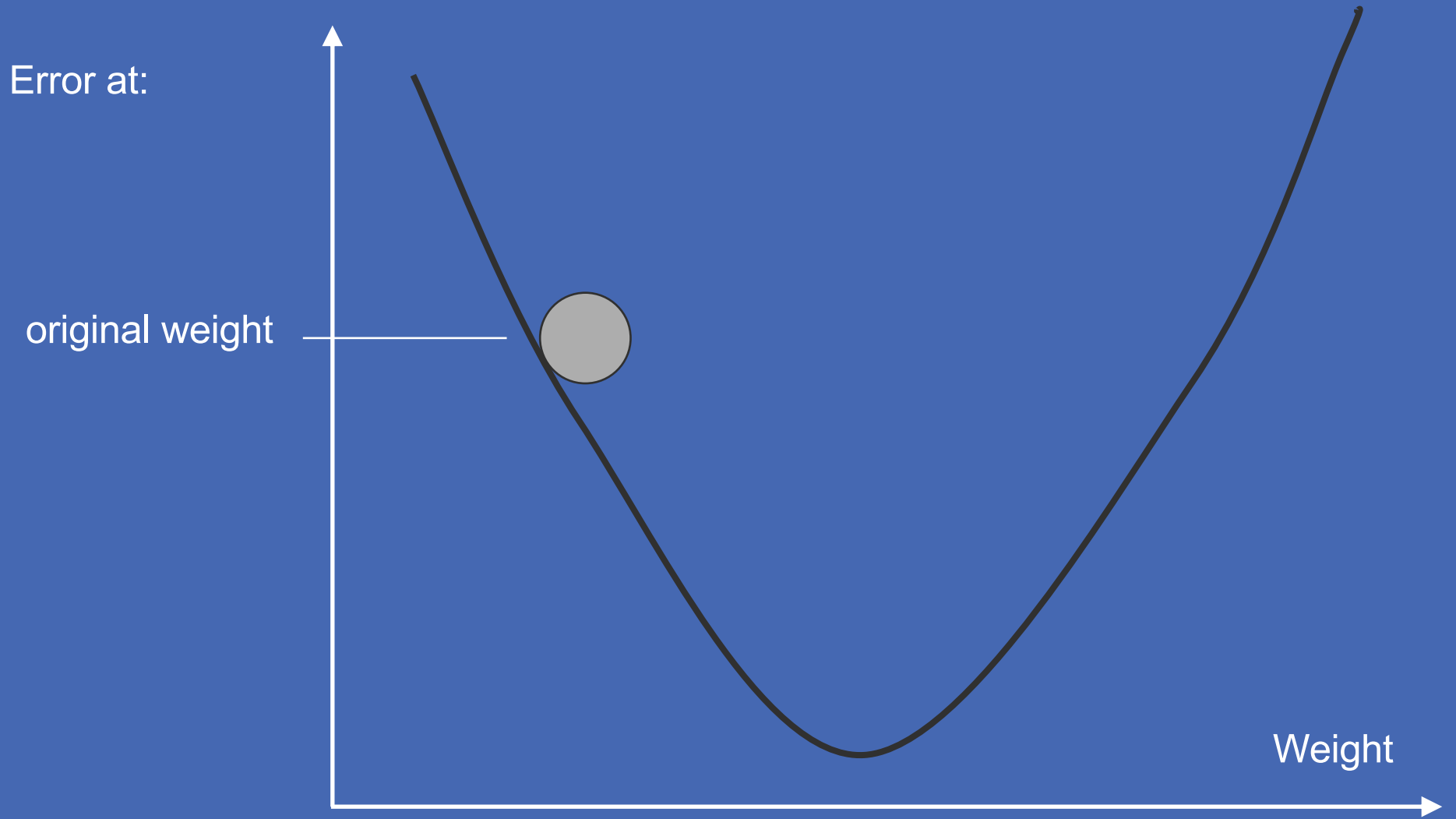
error	truth	answer	solid
.5	0.	.5	
.75	0.	.75	vertical
.25	0.	-.25	diagonal
1.75	1.	-.75	horizontal

Errors

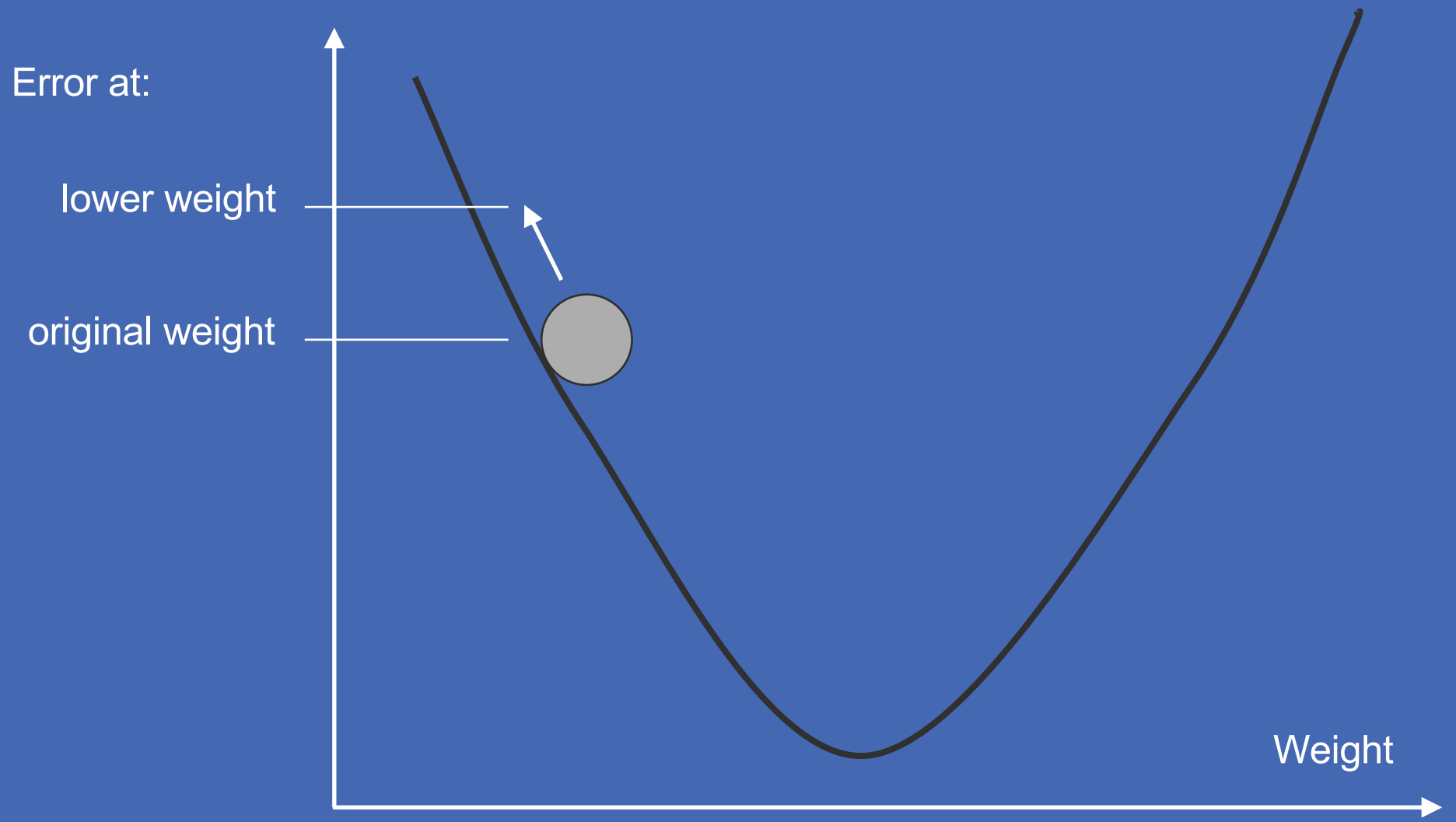


	error	truth	answer	
solid	.5	0.	.5	
vertical	.75	0.	.75	
diagonal	.25	0.	-.25	
horizontal	1.75	1.	-.75	
total	3.25			

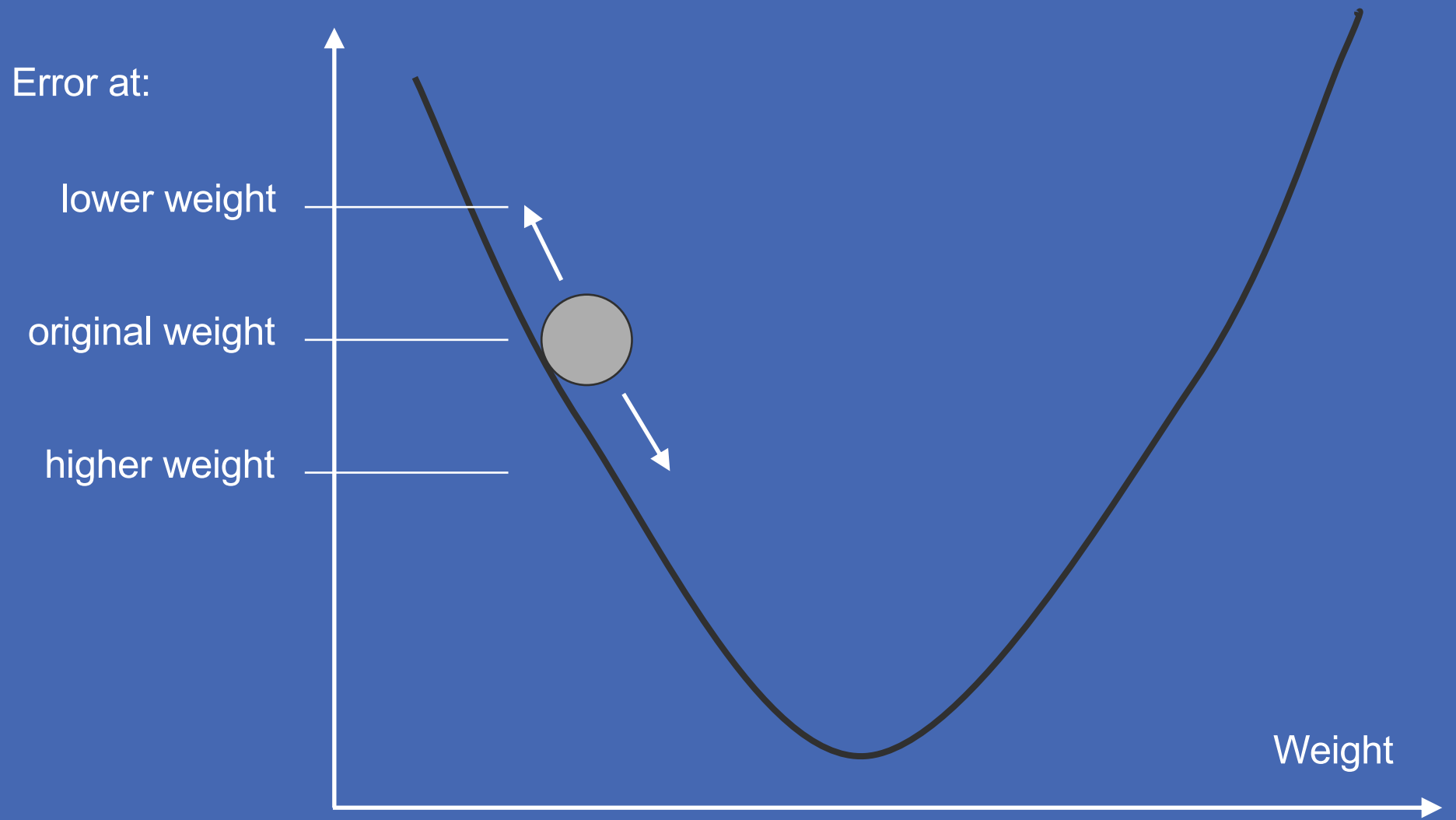
Learn all the weights: Gradient descent



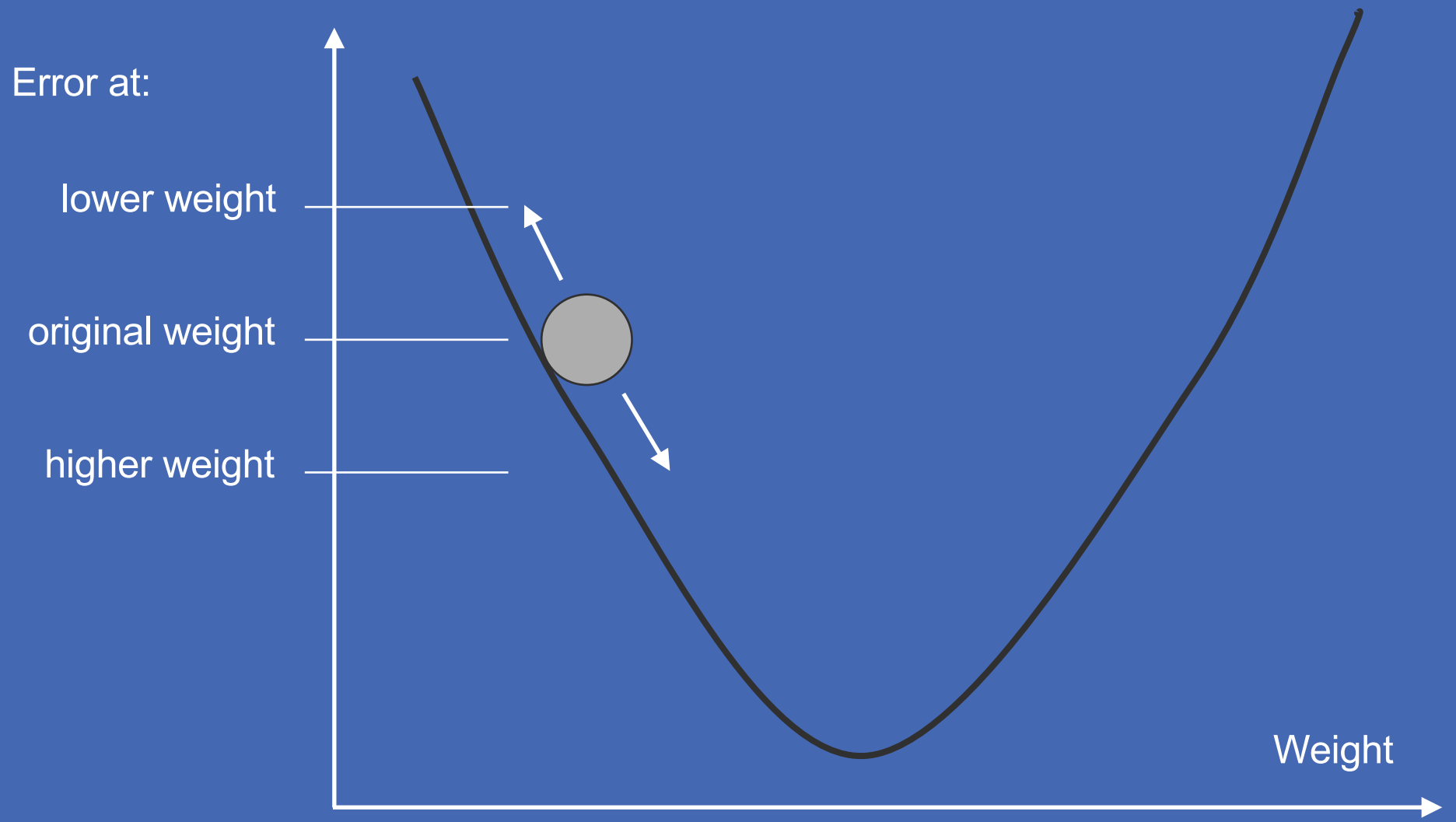
Learn all the weights: Gradient descent



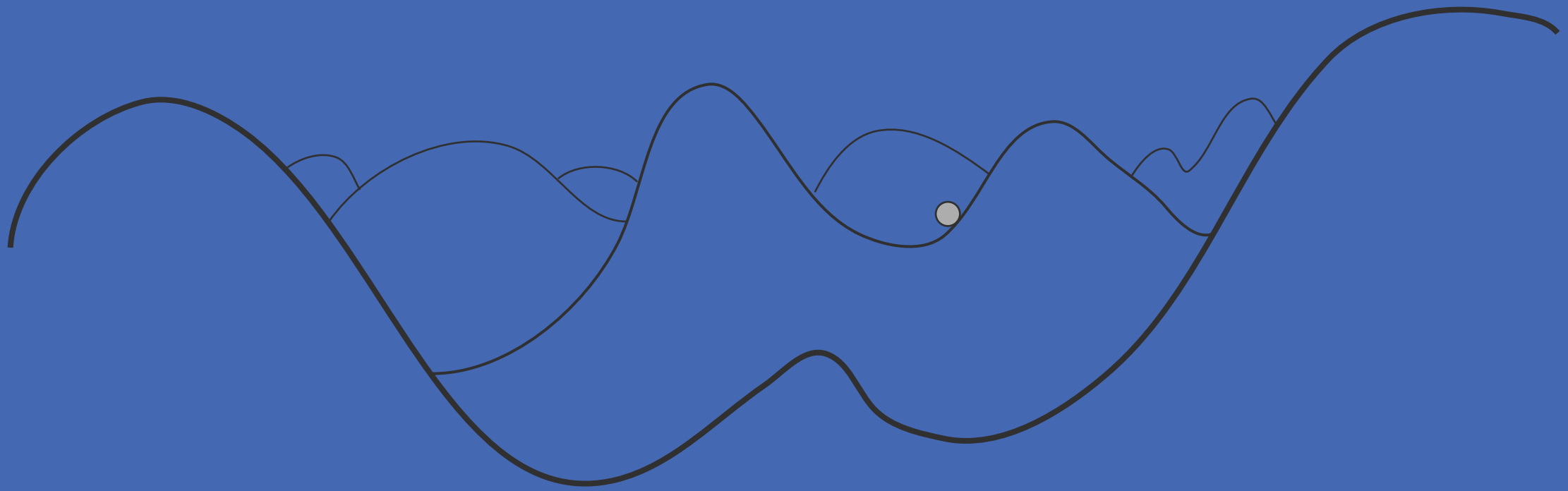
Learn all the weights: Gradient descent



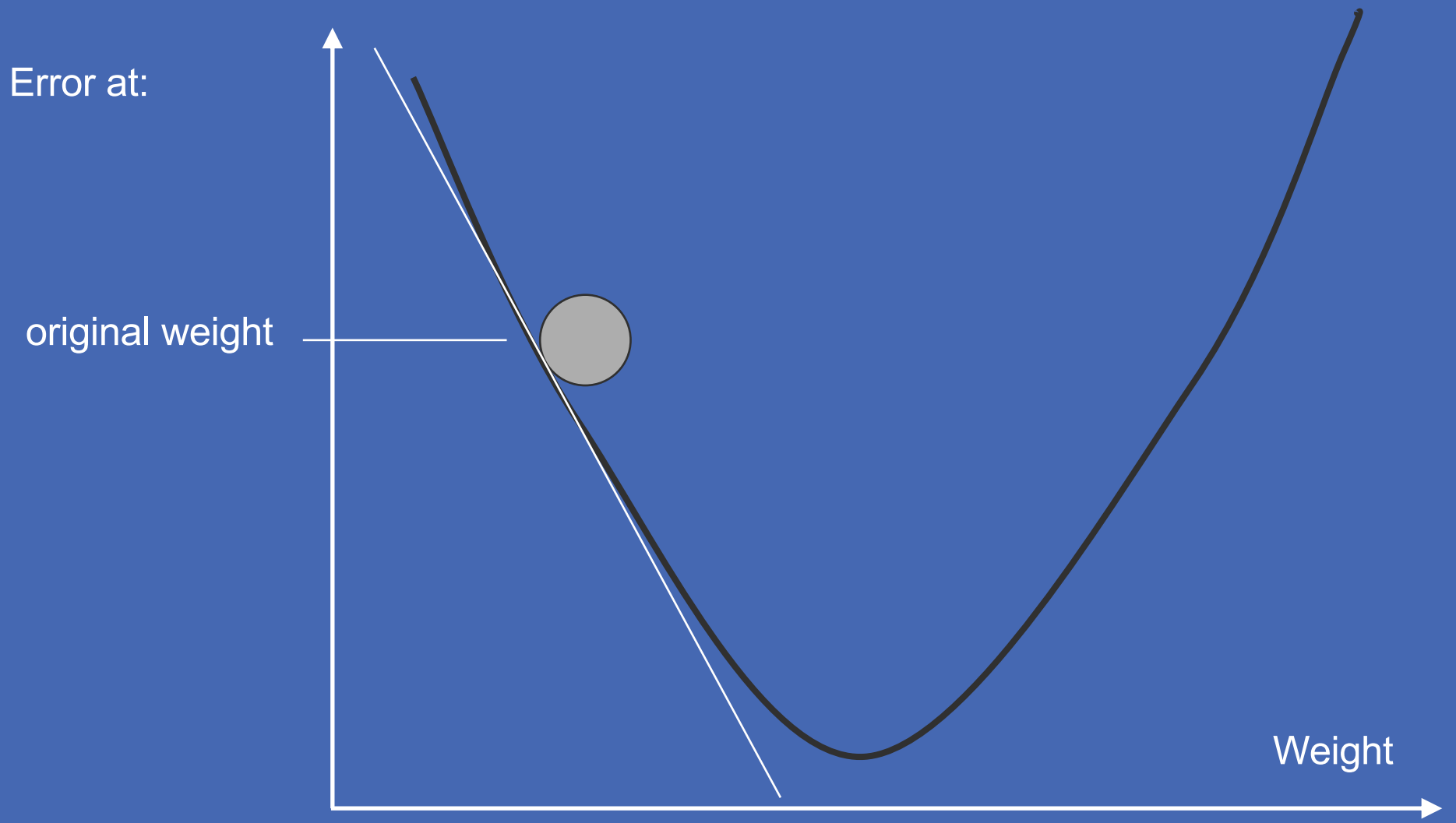
Numerically calculating the gradient is expensive



Numerically calculating the gradient is very expensive

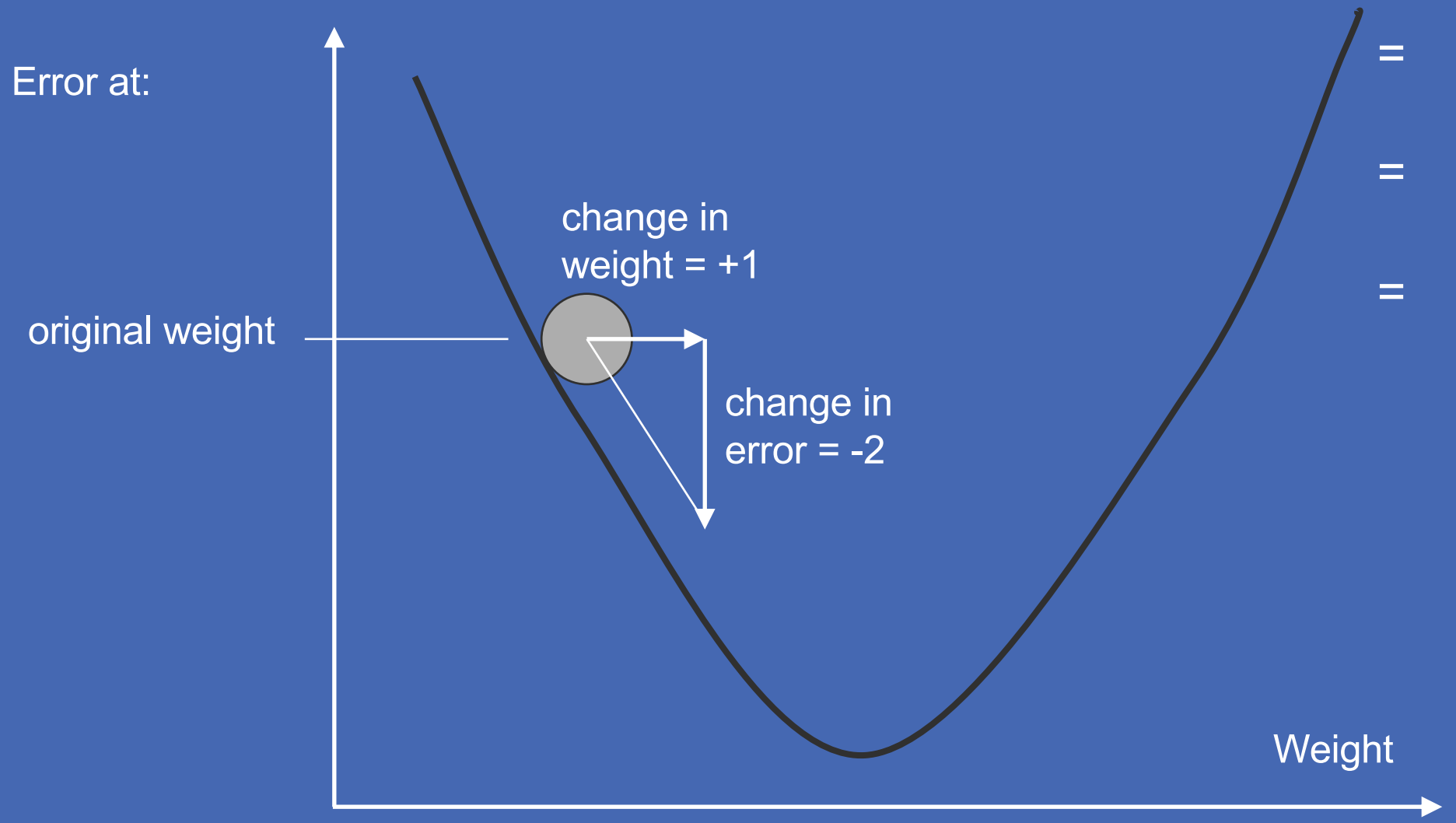


Calculate the gradient (slope) directly



Slope

$$\text{slope} = \frac{\text{change in error}}{\text{change in weight}}$$



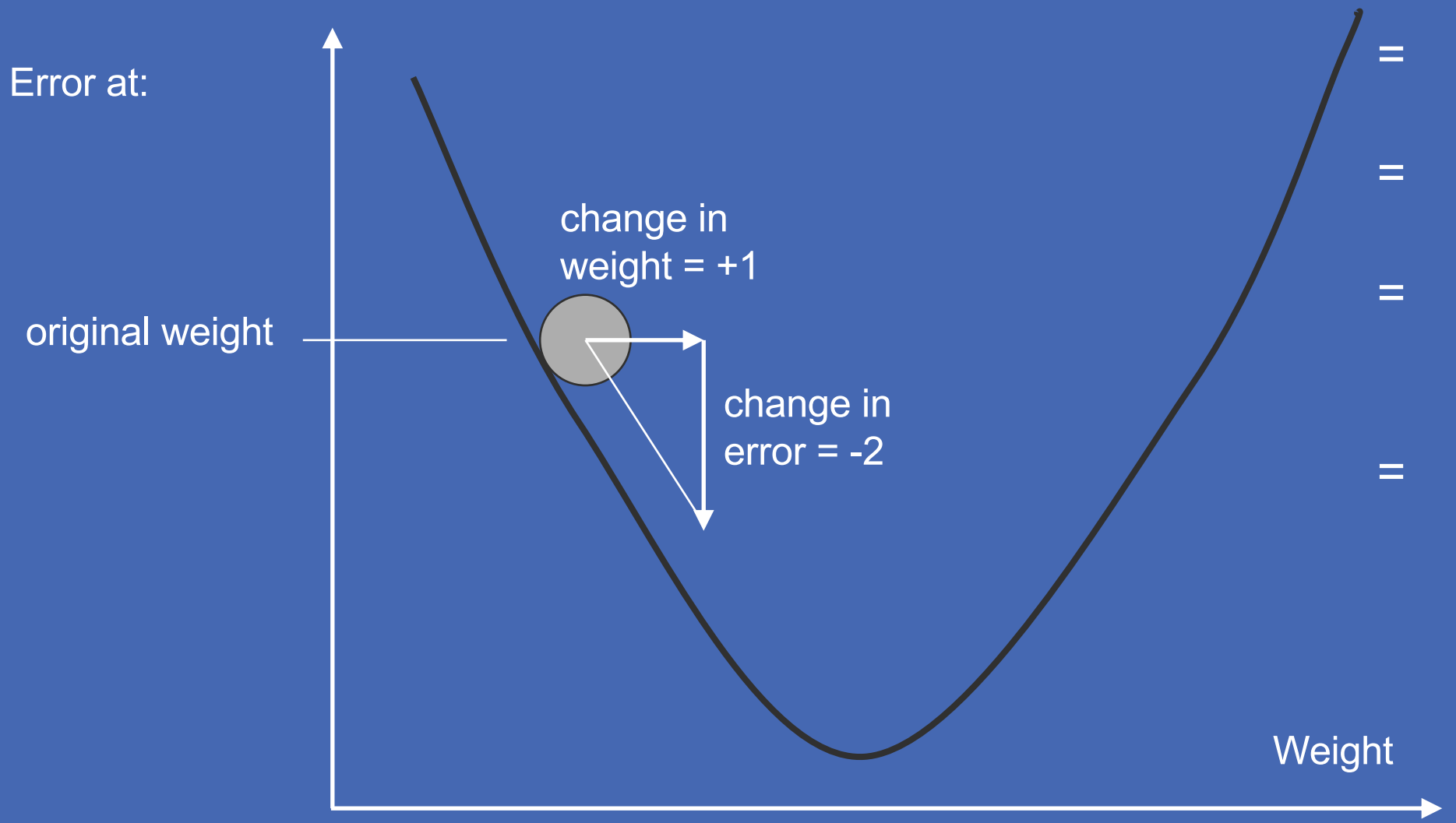
$$= \frac{\Delta \text{ error}}{\Delta \text{ weight}}$$

$$= \frac{d(\text{error})}{d(\text{weight})}$$

$$= \frac{\partial e}{\partial w}$$

Slope

$$\text{slope} = \frac{\text{change in error}}{\text{change in weight}}$$



$$= \frac{\Delta \text{ error}}{\Delta \text{ weight}}$$

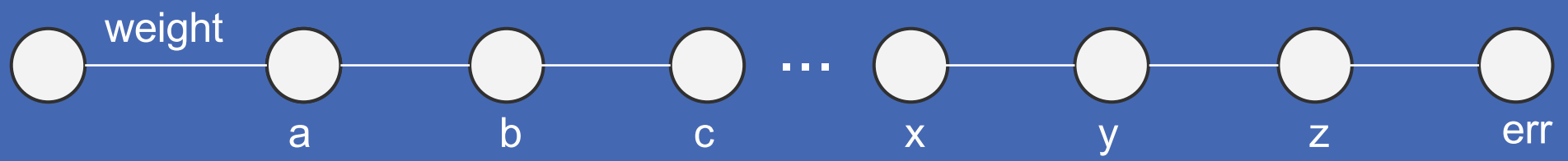
$$= \frac{d(\text{error})}{d(\text{weight})}$$

$$= \frac{\partial e}{\partial w}$$

$$= \frac{-2}{+1} = -2$$

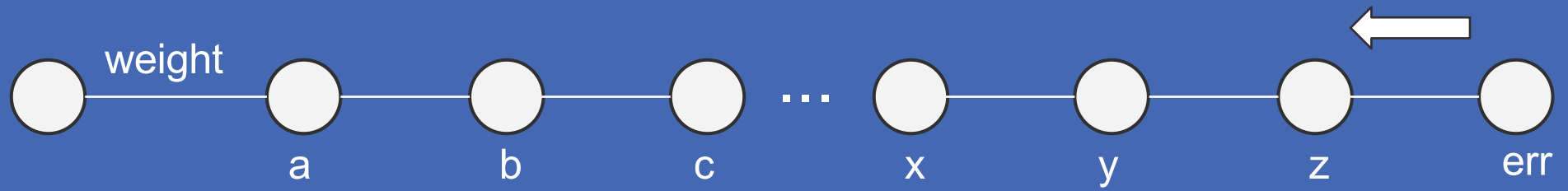
Chaining

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \ldots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



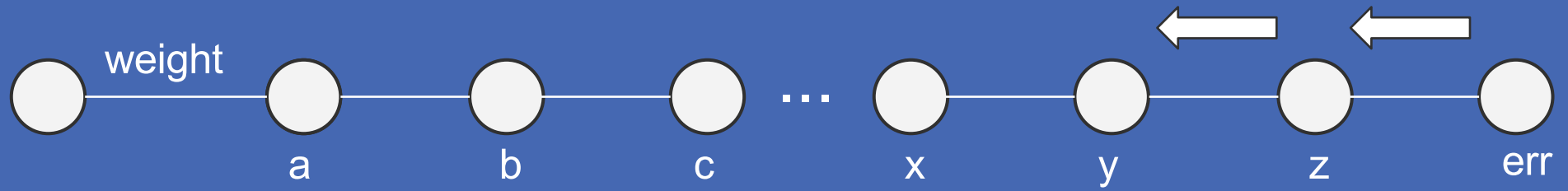
Backpropagation

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \ldots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



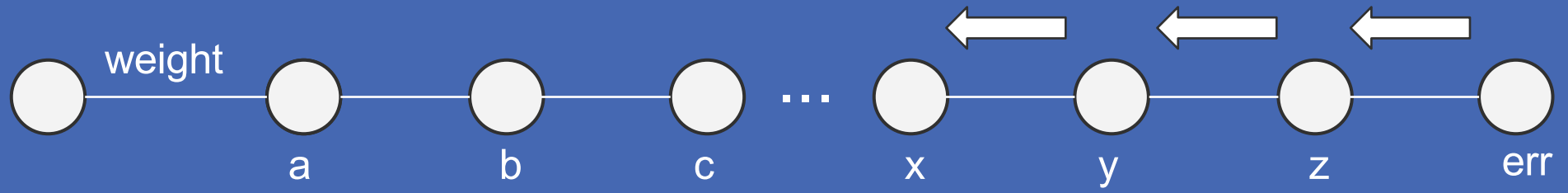
Backpropagation

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \ldots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



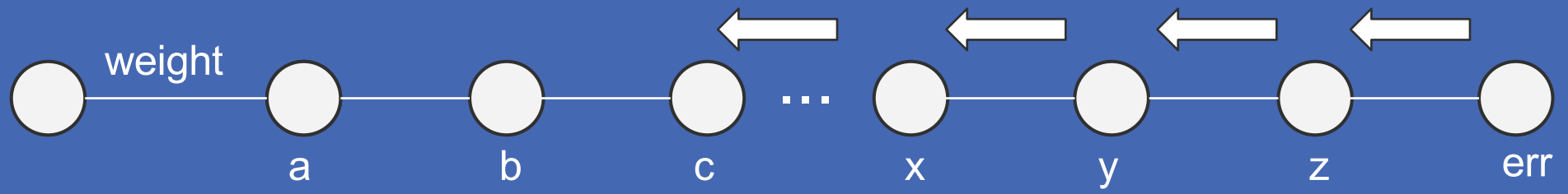
Backpropagation

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \dots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



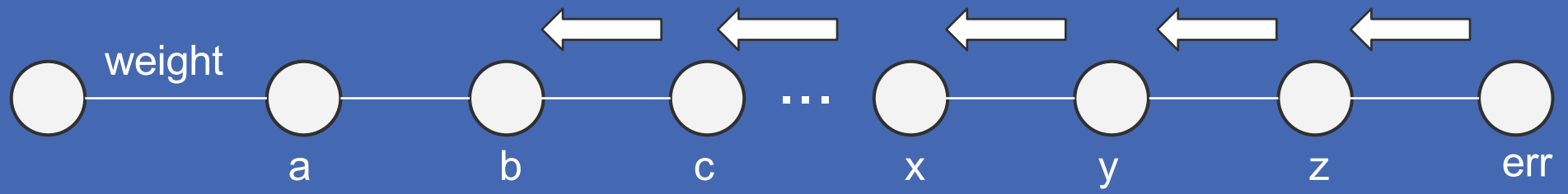
Backpropagation

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \ldots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



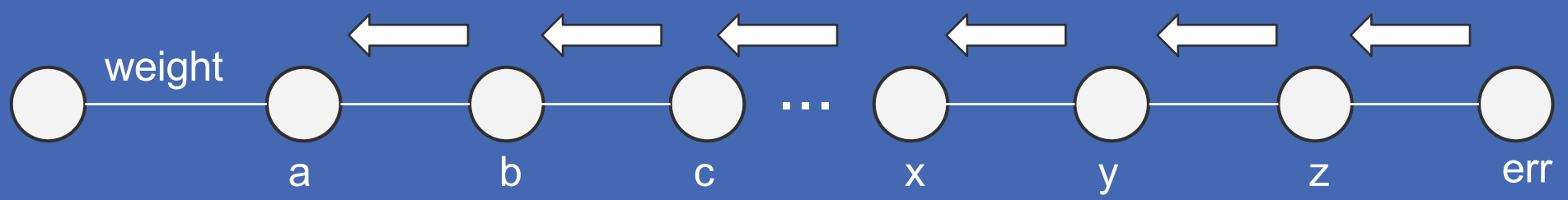
Backpropagation

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \ldots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



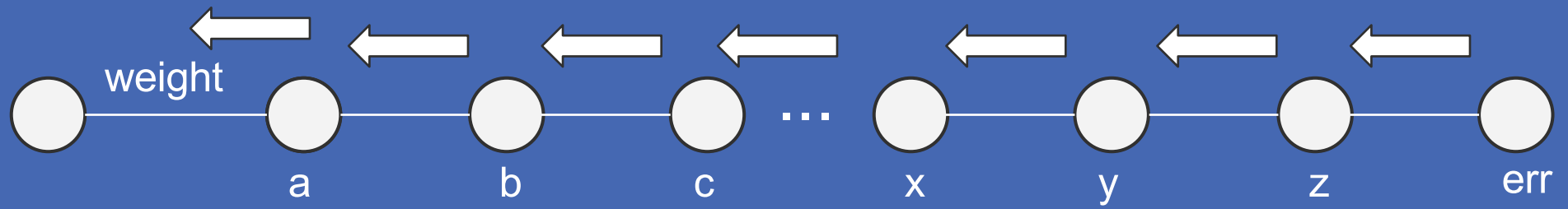
Backpropagation

$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \ldots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$

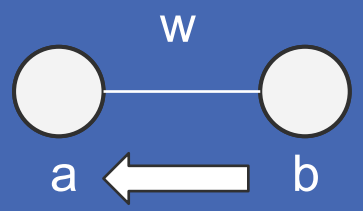


Backpropagation

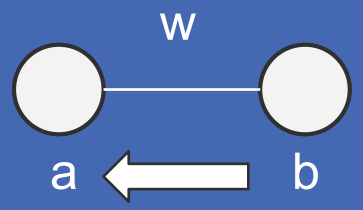
$$\frac{\partial err}{\partial weight} = \frac{\partial a}{\partial weight} * \frac{\partial b}{\partial a} * \frac{\partial c}{\partial b} * \frac{\partial d}{\partial c} * \dots * \frac{\partial y}{\partial x} * \frac{\partial z}{\partial y} * \frac{\partial err}{\partial z}$$



Backpropagation challenge: weights

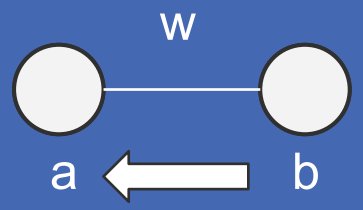


Backpropagation challenge: weights



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

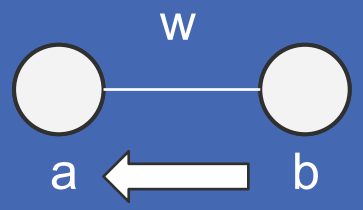
Backpropagation challenge: weights



$b = wa$

$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

Backpropagation challenge: weights

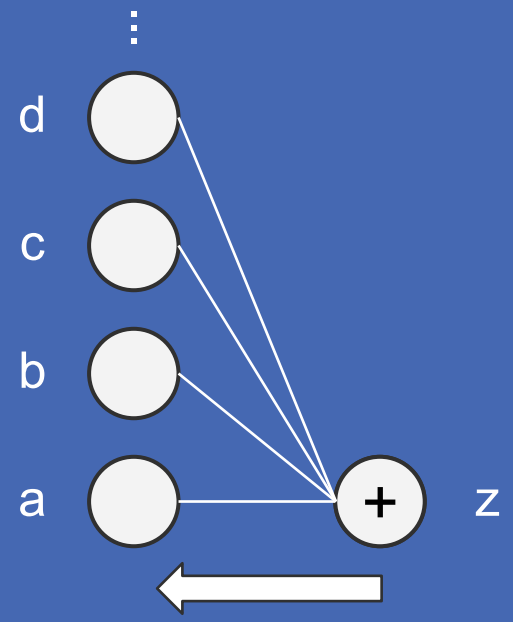


$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

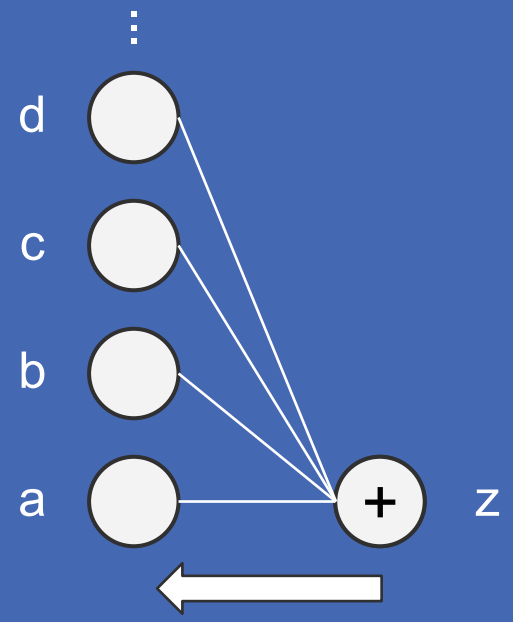
$$b = wa$$

$$\frac{\partial b}{\partial a} = w$$

Backpropagation challenge: sums

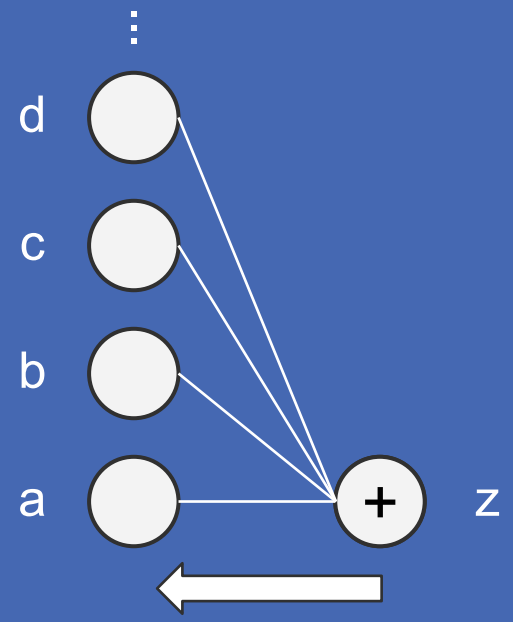


Backpropagation challenge: sums



$$\frac{\partial err}{\partial a} = \frac{\partial z}{\partial a} * \frac{\partial err}{\partial z}$$

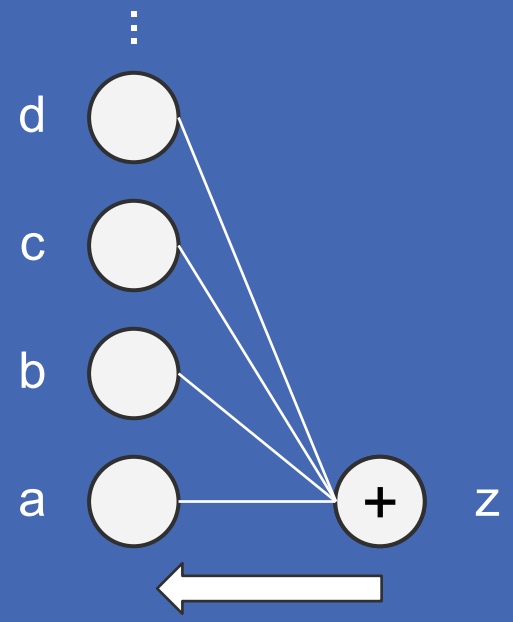
Backpropagation challenge: sums



$$z = a + b + c + d + \ldots$$

$$\frac{\partial err}{\partial a} = \frac{\partial z}{\partial a} * \frac{\partial err}{\partial z}$$

Backpropagation challenge: sums

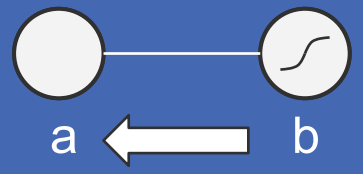


$$\frac{\partial err}{\partial a} = \frac{\partial z}{\partial a} * \frac{\partial err}{\partial z}$$

$$z = a + b + c + d + ...$$

$$\frac{\partial z}{\partial a} = 1$$

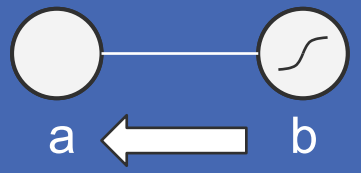
Backpropagation challenge: sigmoid



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

Backpropagation challenge: sigmoid

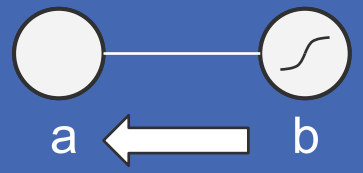
$$b = \frac{1}{1 + e^{-a}}$$



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

Backpropagation challenge: sigmoid

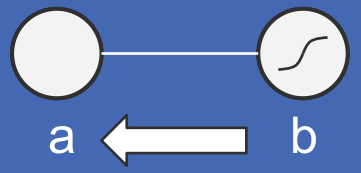
$$b = \frac{1}{1 + e^{-a}} = \sigma(a)$$



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

Backpropagation challenge: sigmoid

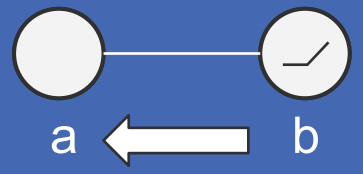
$$b = \frac{1}{1 + e^{-a}} = \sigma(a)$$



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

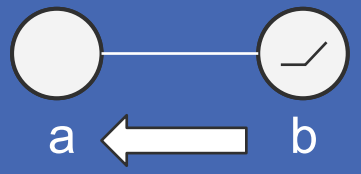
$$\frac{\partial b}{\partial a} = \sigma(a) * (1 - \sigma(a))$$

Backpropagation challenge: ReLU



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

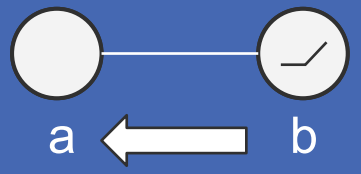
Backpropagation challenge: ReLU



$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

$$b = a, \quad a > 0$$
$$= 0, \quad \text{otherwise}$$

Backpropagation challenge: ReLU

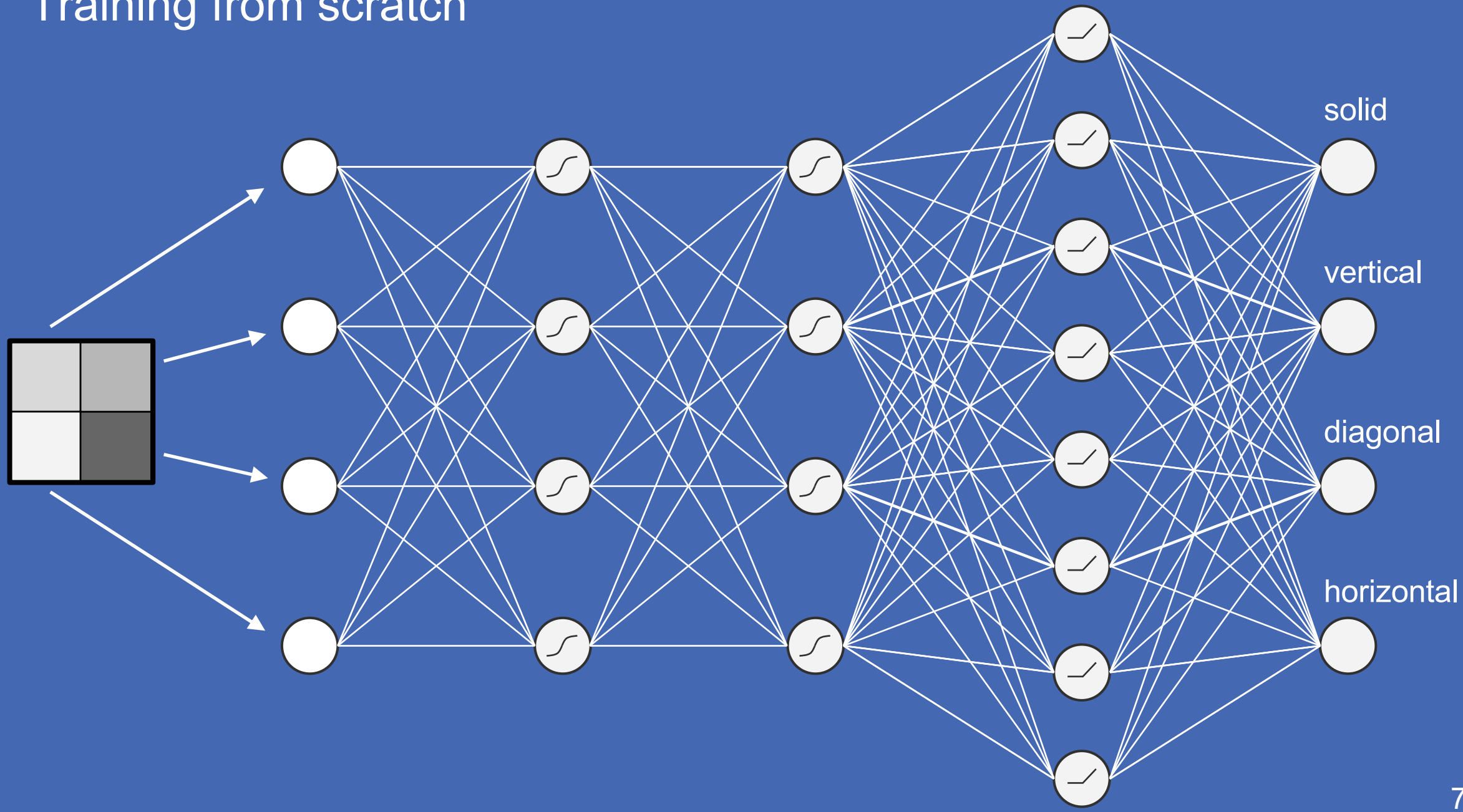


$$\frac{\partial err}{\partial a} = \frac{\partial b}{\partial a} * \frac{\partial err}{\partial b}$$

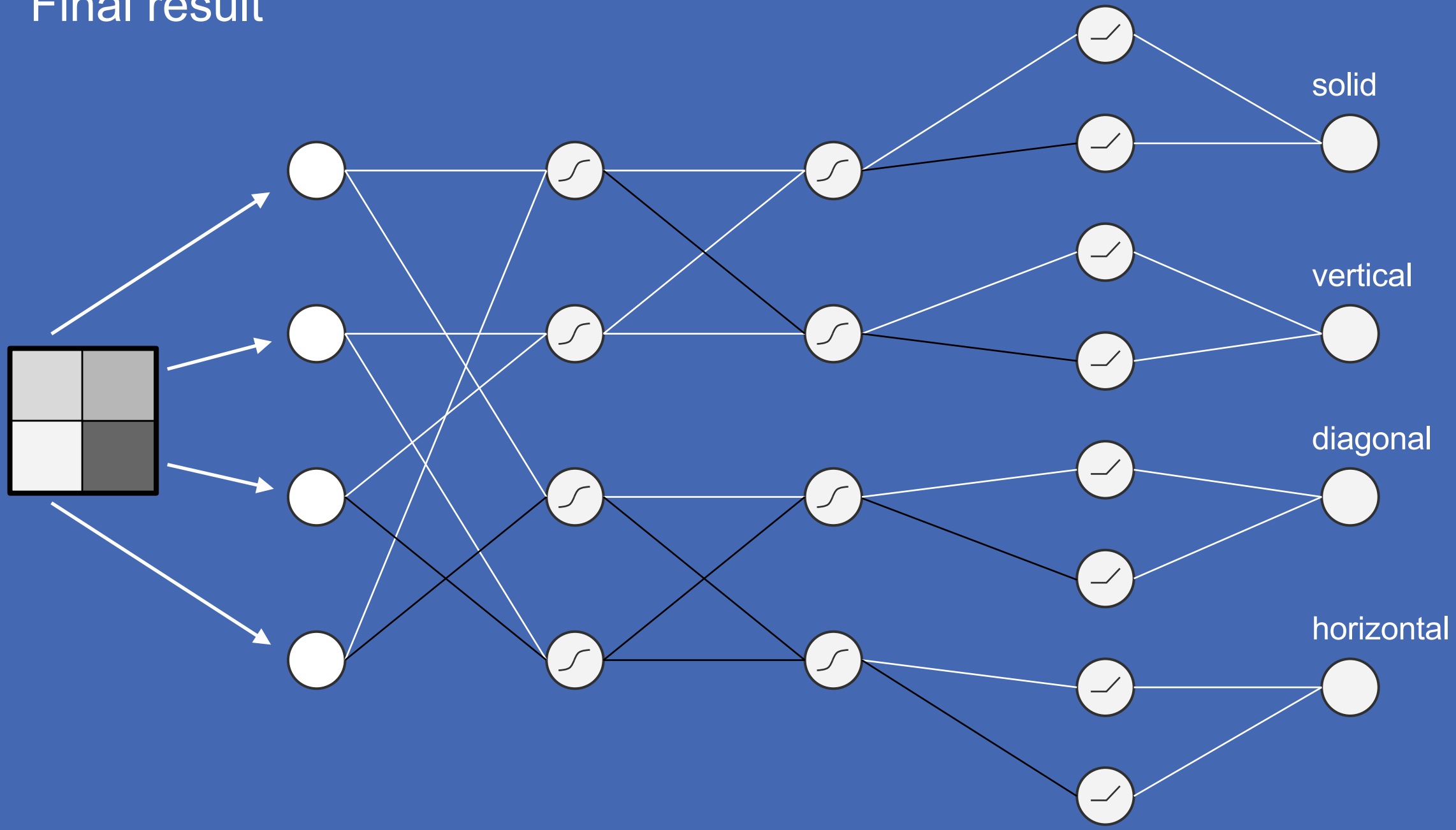
$$b = a, \quad a > 0$$
$$= 0, \quad \text{otherwise}$$

$$\frac{\partial b}{\partial a} = 1, \quad a > 0$$
$$0, \quad \text{otherwise}$$

Training from scratch



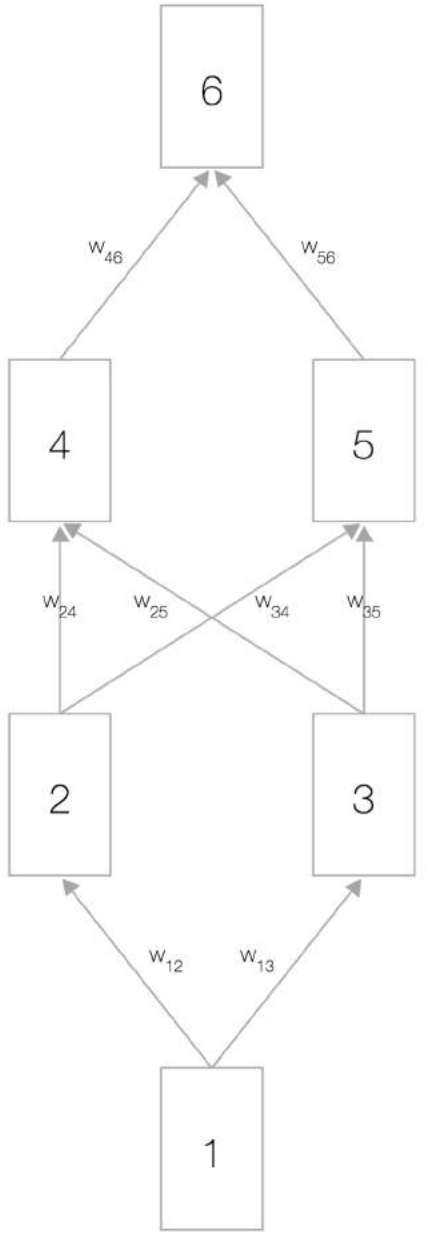
Final result



Simple neural network

On the right, you see a neural network with one input, one output node and two hidden layers of two nodes each.

Nodes in neighboring layers are connected with weights w_{ij}, which are the network parameters.



Output layer

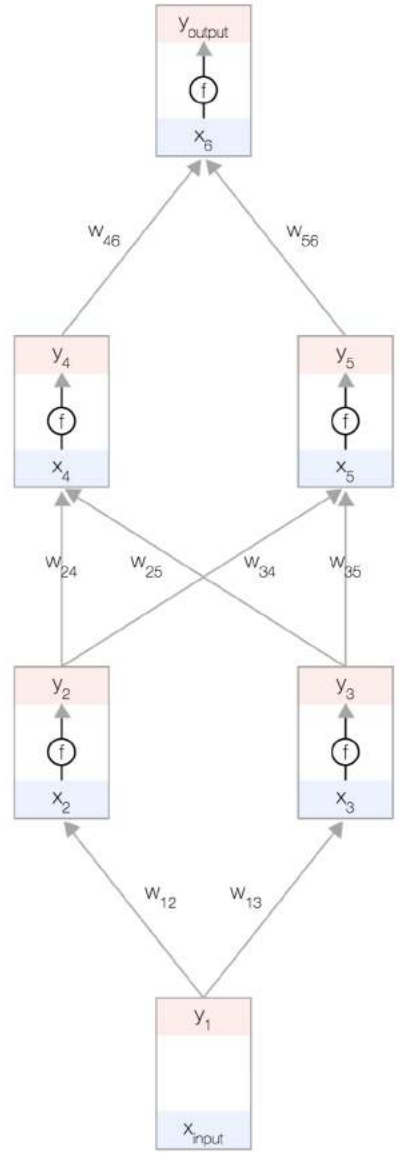
Hidden layers

Input layer

Activation function

Each node has a total input x, an activation function $f(x)$ and an output $y = f(x)$. $f(x)$ has to be a non-linear function, otherwise the neural network will only be able to learn linear models.

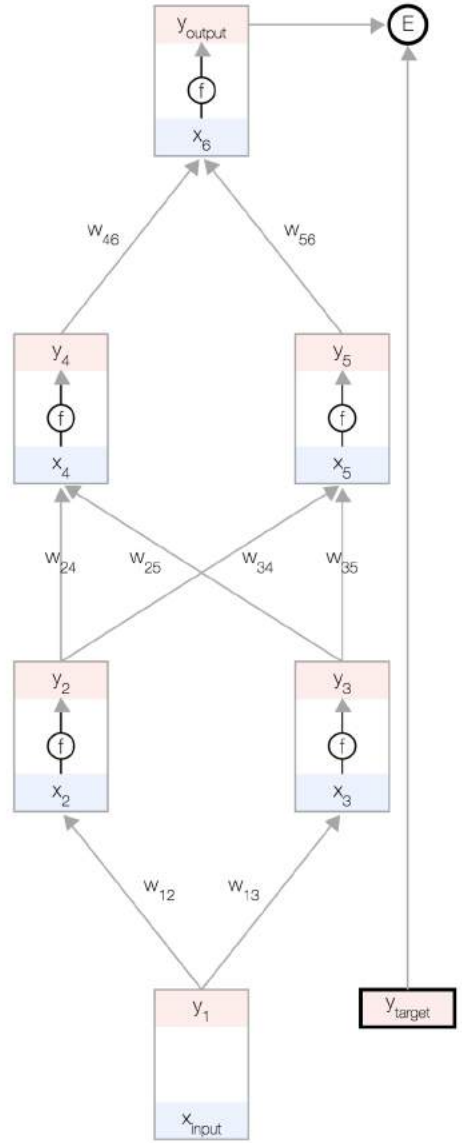
A commonly used activation function is the Sigmoid function: $f(x) = \frac{1}{1+e^{-x}}$.



Error function

The goal is to learn the weights of the network automatically from data such that the predicted output y_{output} is close to the target y_{target} for all inputs x_{input}.

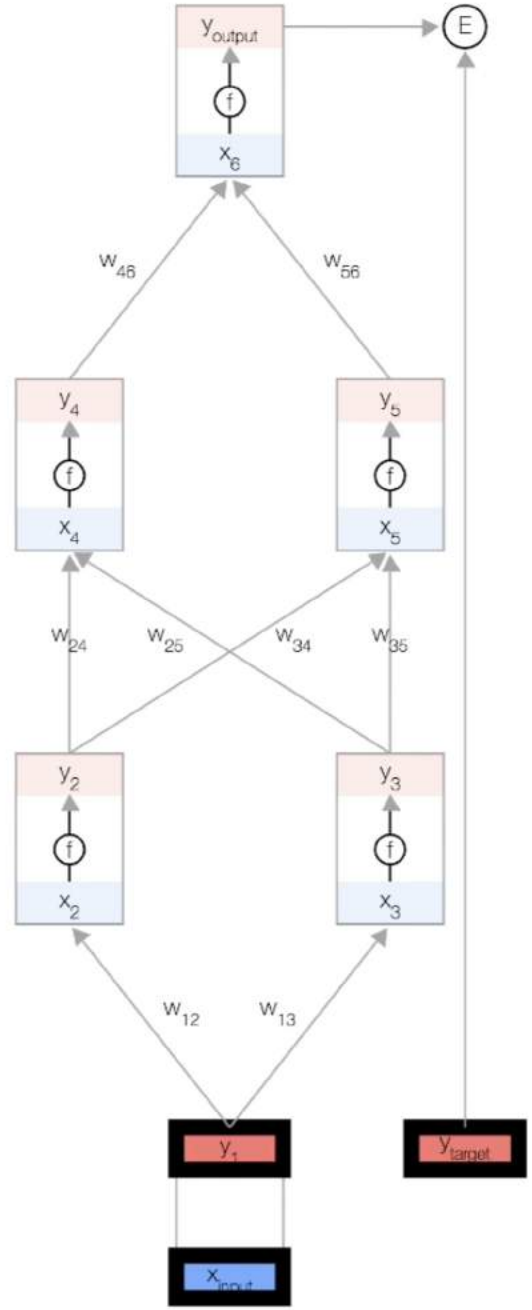
To measure how far we are from the goal, we use an error function E. A commonly used error functon is $E(y_{output}, y_{target}) = \frac{1}{2}(y_{output} - y_{target})^2$.



Forward propagation

We begin by taking an input example (x_{input}, y_{target}) and updating the input layer of the network.

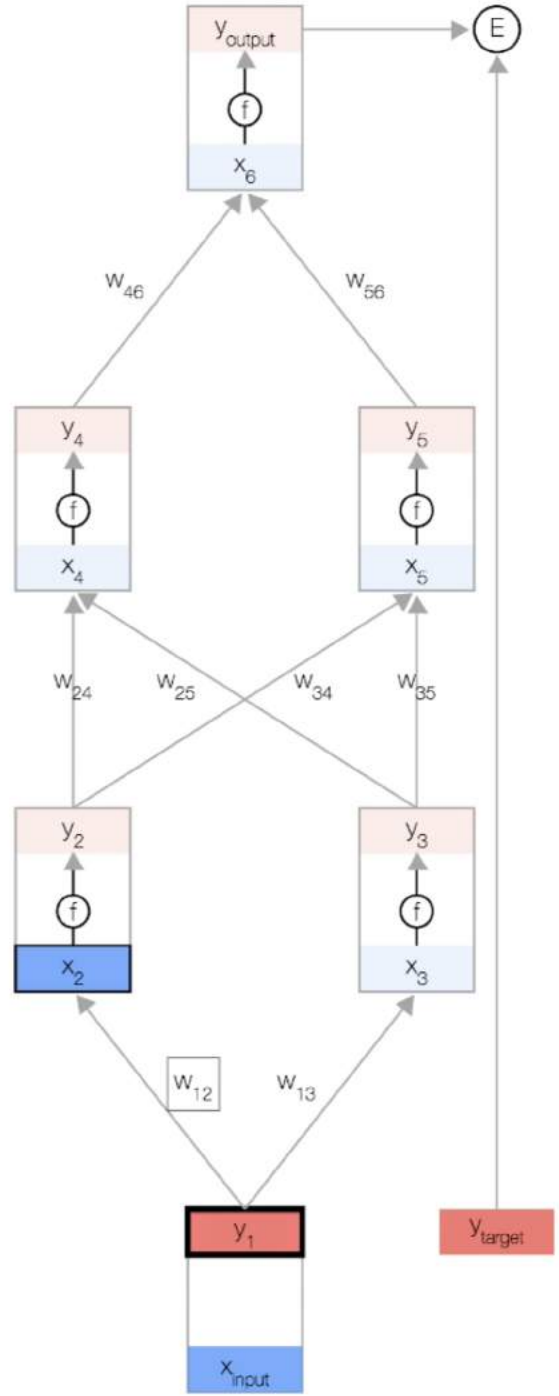
For consistency, we consider the input to be like any other node but without an activation function so its output is equal to its input, i.e. $y_1 = x_{input}$.



Forward propagation

Now, we update the first hidden layer. We take the output y of the nodes in the previous layer and use the weights to compute the input x of the nodes in the next layer.

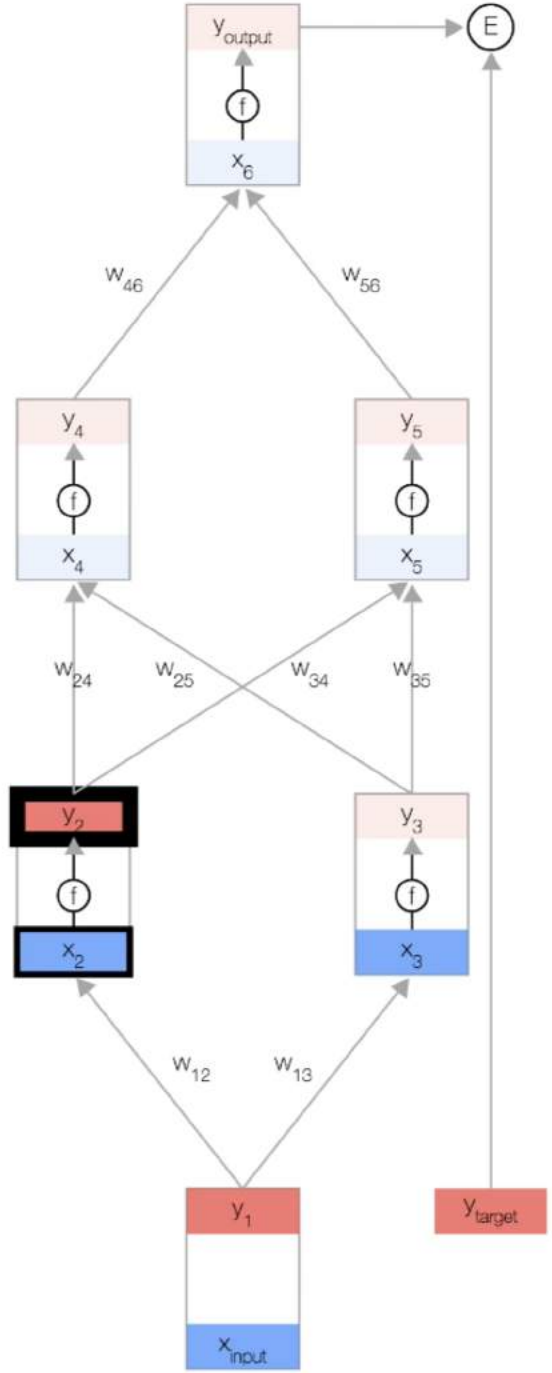
$$x_j = \sum_{i \in in(j)} w_{ij} y_i + b_j$$



Forward propagation

Then we update the output of the nodes in the first hidden layer. For this we use the activation function, $f(x)$.

$$y = f(x)$$

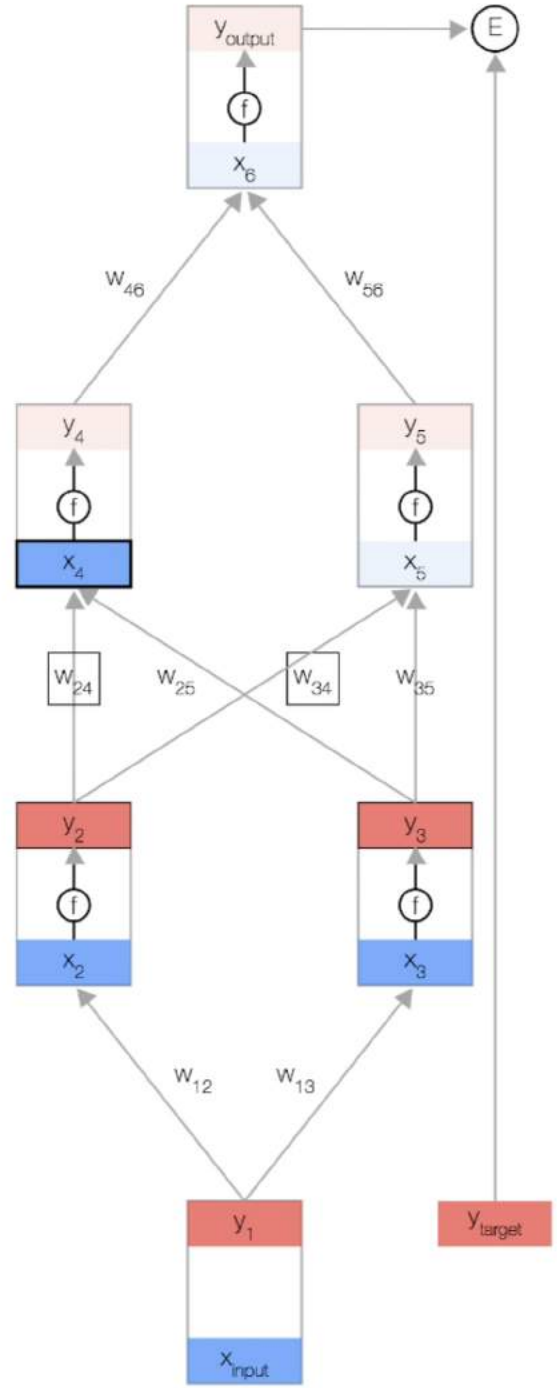


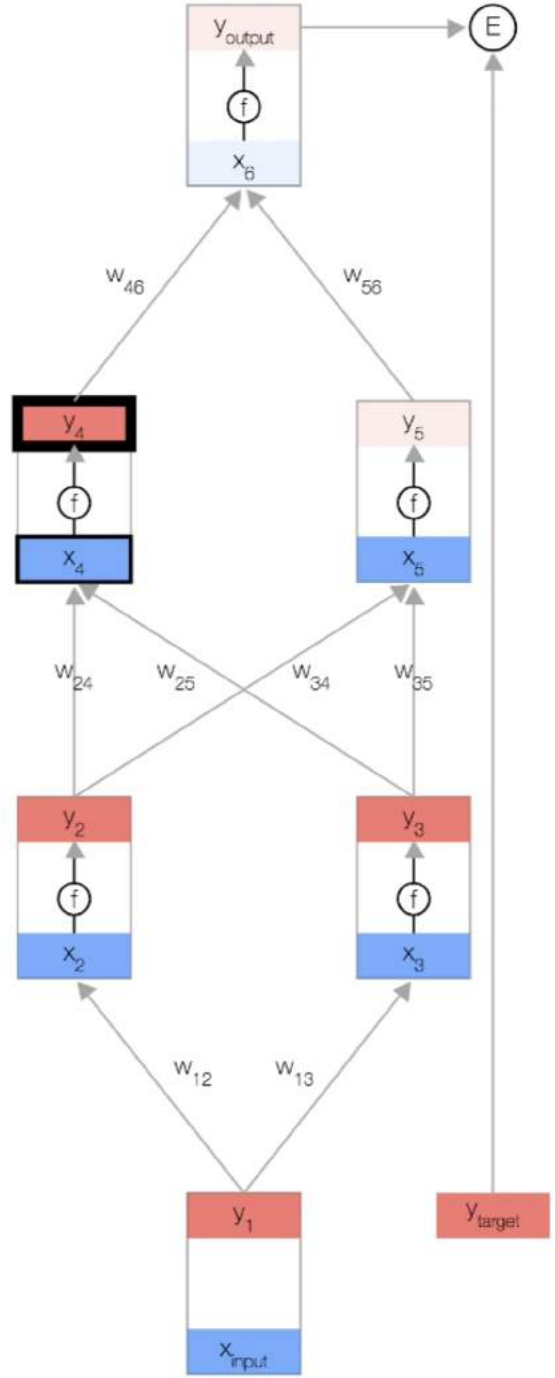
Forward propagation

Using these 2 formulas we propagate for the rest of the network and get the final output of the network.

$$y = f(x)$$

$$x_j = \sum_{i \in in(j)} w_{ij} y_i + b_j$$




y_{output}
E
f
x_6
w_{46}
w_{56}
y_4
y_5
x_4
x_5
w_{24}
w_{25}
w_{34}
w_{35}
y_2
y_3
x_2
x_3
w_{12}
w_{13}
y_1
y_{target}
x_{input}

Error derivative

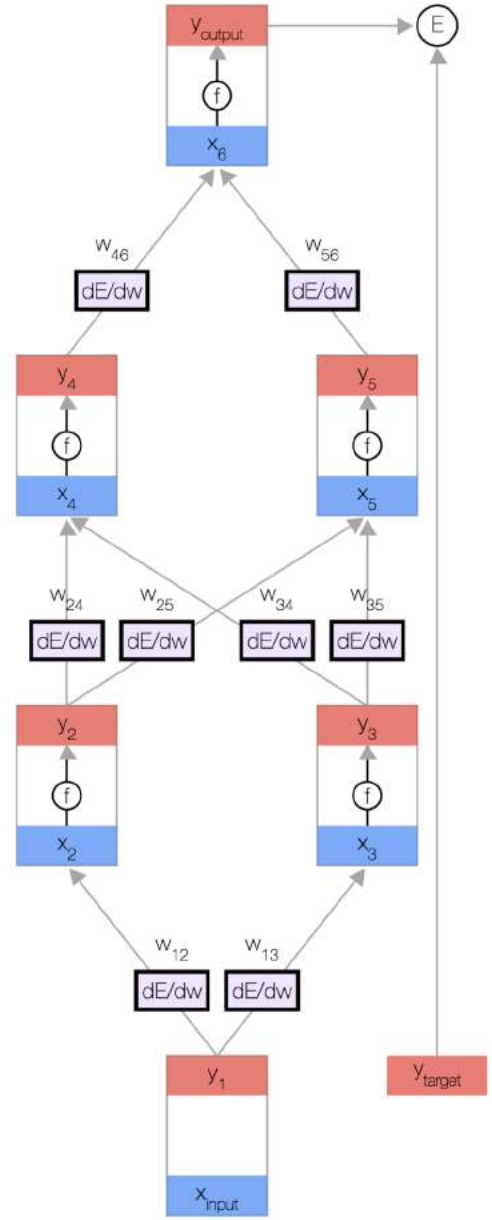
The backpropagation algorithm decides how much to update each weight of the network after comparing the predicted output with the desired output for a particular example. For this, we need to compute how the error changes with respect to each weight $\frac{dE}{dw_{ij}}$.

Once we have the error derivatives, we can update the weights using a simple update rule:

$$w_{ij} = w_{ij} - \alpha \frac{dE}{dw_{ij}}$$

where α is a positive constant, referred to as the learning rate, which we need to fine-tune empirically.

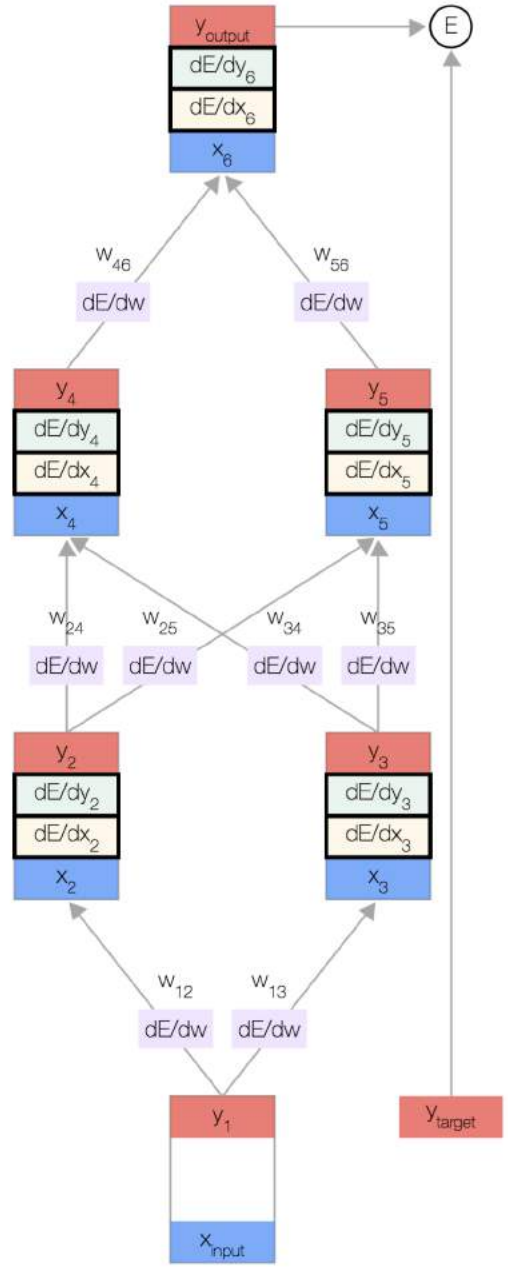
[Note] The update rule is very simple: if the error goes down when the weight increases ($\frac{dE}{dw_{ij}} < 0$), then increase the weight, otherwise if the error goes up when the weight increases ($\frac{dE}{dw_{ij}} > 0$), then decrease the weight.



Additional derivatives

To help compute $\frac{dE}{dw_{ij}}$, we additionally store for each node two more derivatives: how the error changes with:

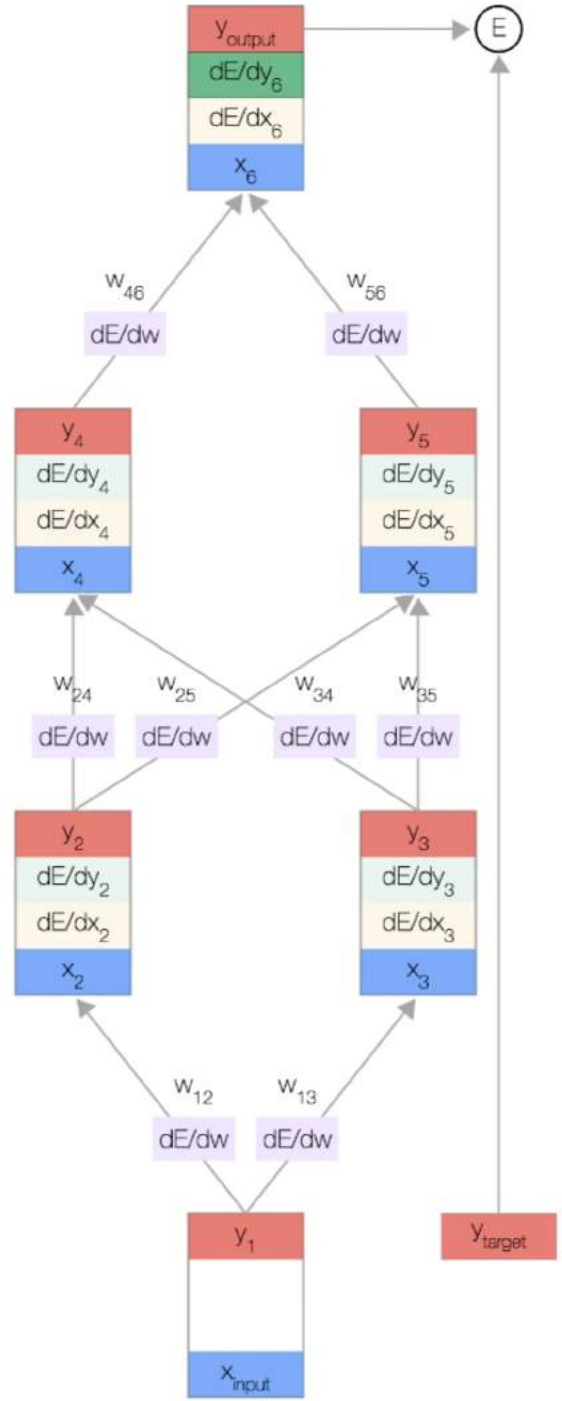
- the total input of the node $\frac{dE}{dx}$ and
- the output of the node $\frac{dE}{dy}$.



Back propagation

Let's begin backpropagating the error derivatives. Since we have the predicted output of this particular input example, we can compute how the error changes with that output. Given our error function $E = \frac{1}{2}(y_{output} - y_{target})^2$ we have:

$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target}$$

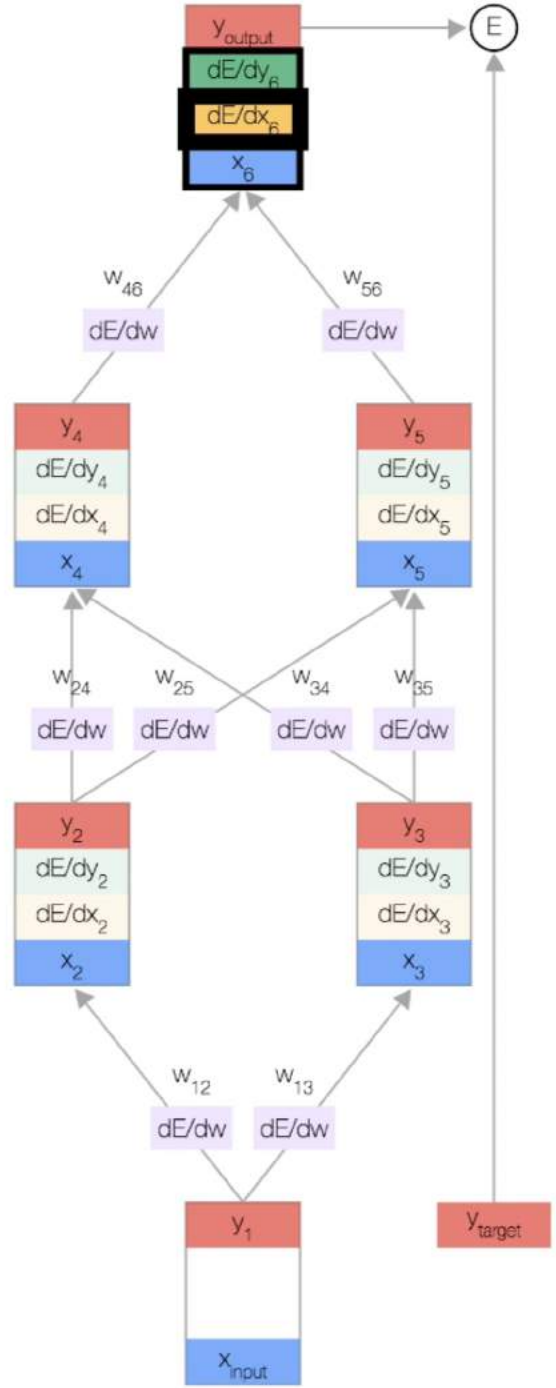


Back propagation

Now that we have $\frac{dE}{dy}$ we can get $\frac{dE}{dx}$ using the chain rule.

$$\frac{\partial E}{\partial x} = \frac{dy}{dx}\frac{\partial E}{\partial y} = \frac{d}{dx}f(x)\frac{\partial E}{\partial y}$$

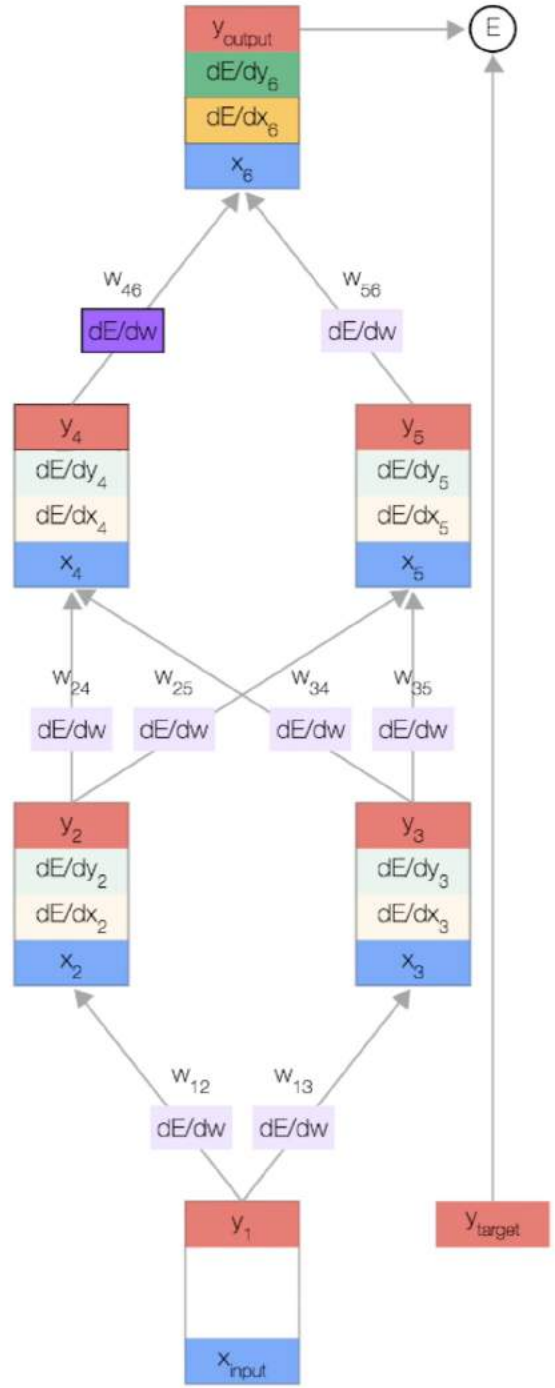
where $\frac{d}{dx}f(x) = f(x)(1 - f(x))$ when $f(x)$ is the Sigmoid activation function.



Back propagation

As soon as we have the error derivative with respect to the total input of a node, we can get the error derivative with respect to the weights coming into that node.

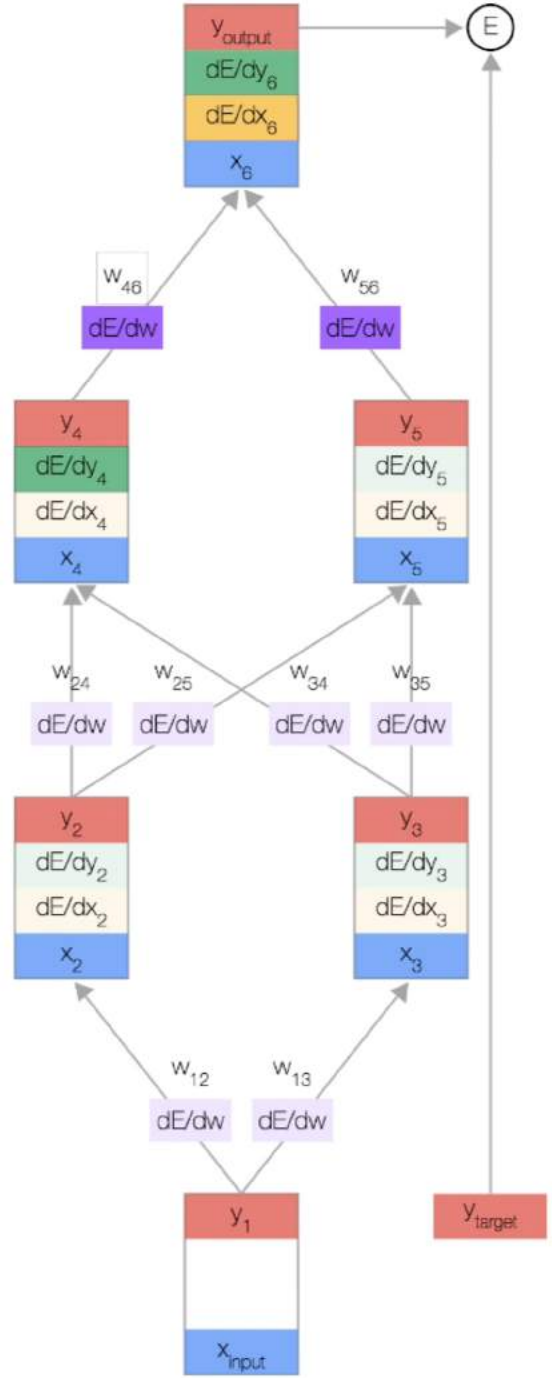
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}$$



Back propagation

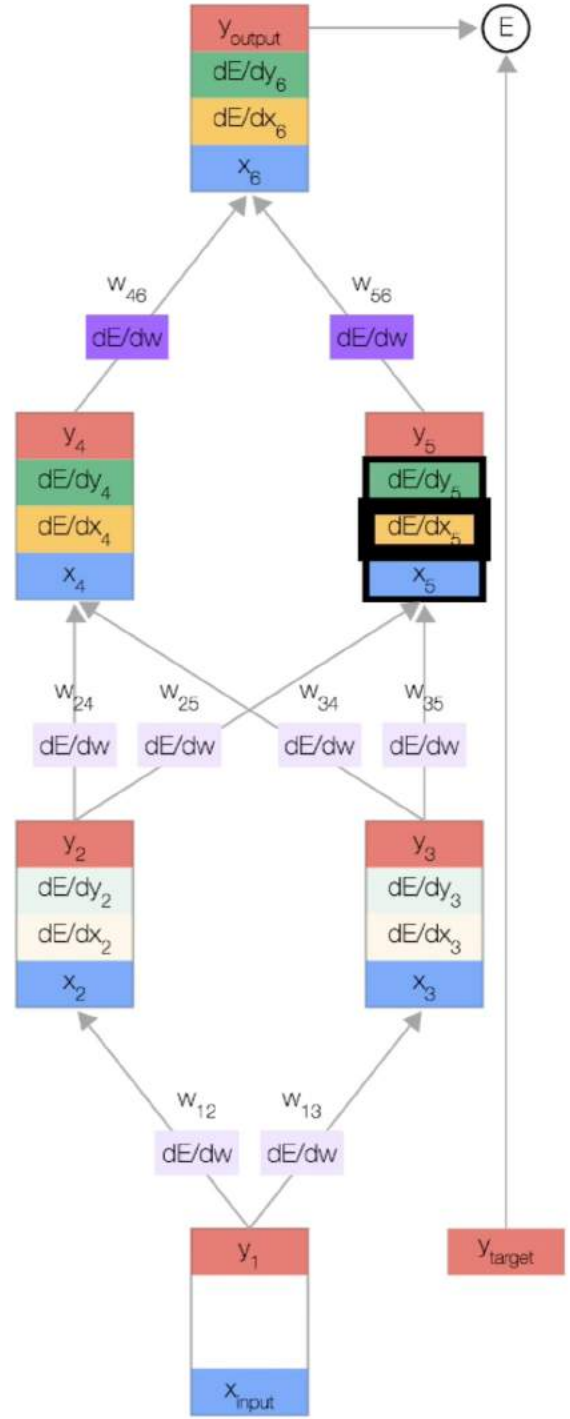
And using the chain rule, we can also get $\frac{dE}{dy}$ from the previous layer. We have made a full circle.

$$\frac{\partial E}{\partial y_i} = \sum_{j \in out(i)} \frac{\partial x_j}{\partial y_i} \frac{\partial E}{\partial x_j} = \sum_{j \in out(i)} w_{ij} \frac{\partial E}{\partial x_j}$$



Back propagation

All that is left to do is repeat the previous three formulas until we have computed all the error derivatives.



Convolutional Neural Networks

End-to-end learning

Self-Driving Cars

Madhur Behl
Principles of Modeling for Cyber-Physical Systems

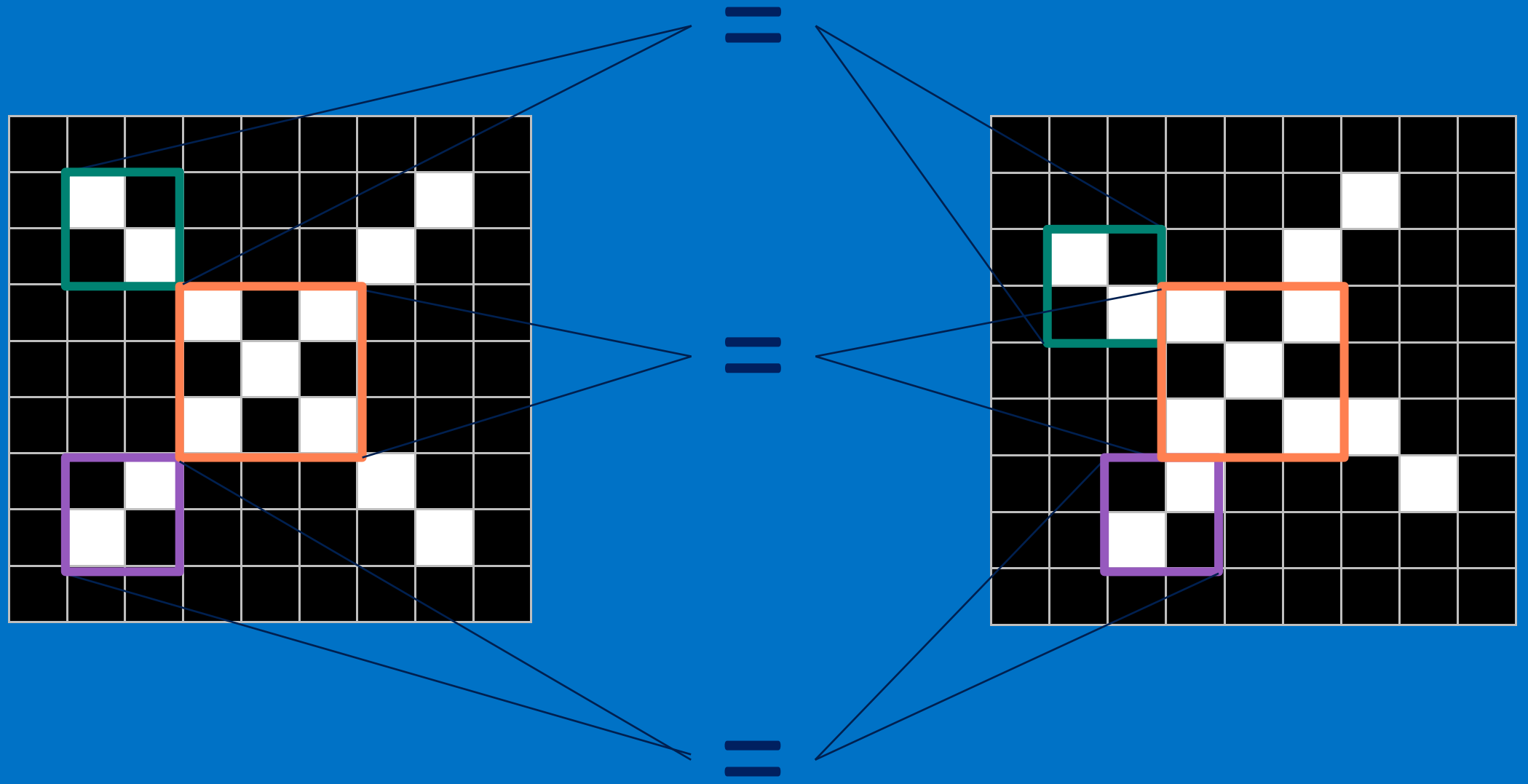
Slides courtesy Brandon Rohrer

A toy ConvNet: X's and O's

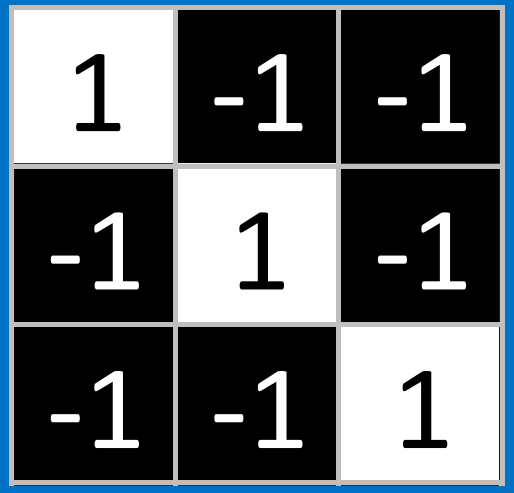
Says whether a picture is of an X or an O

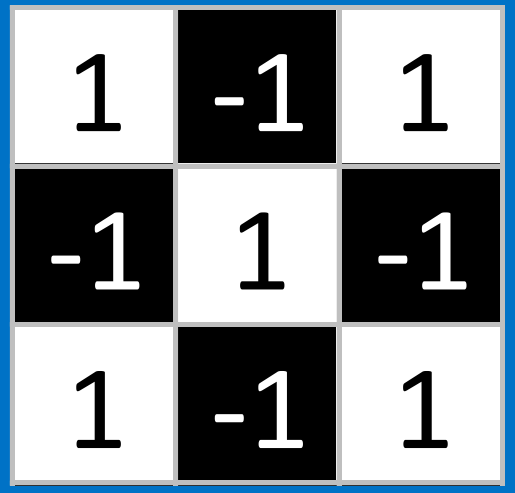
A two-dimensional
array of pixels

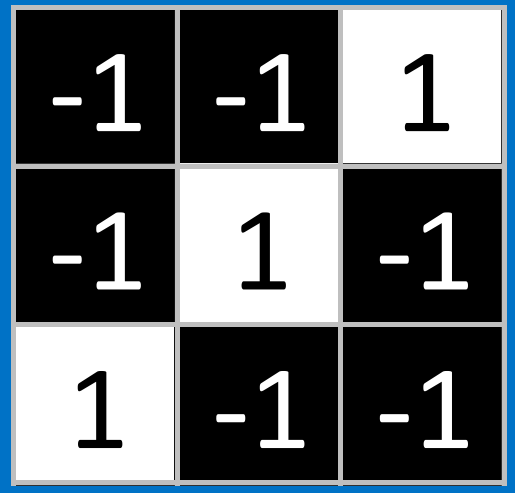
ConvNets match pieces of the image



Features match pieces of the image



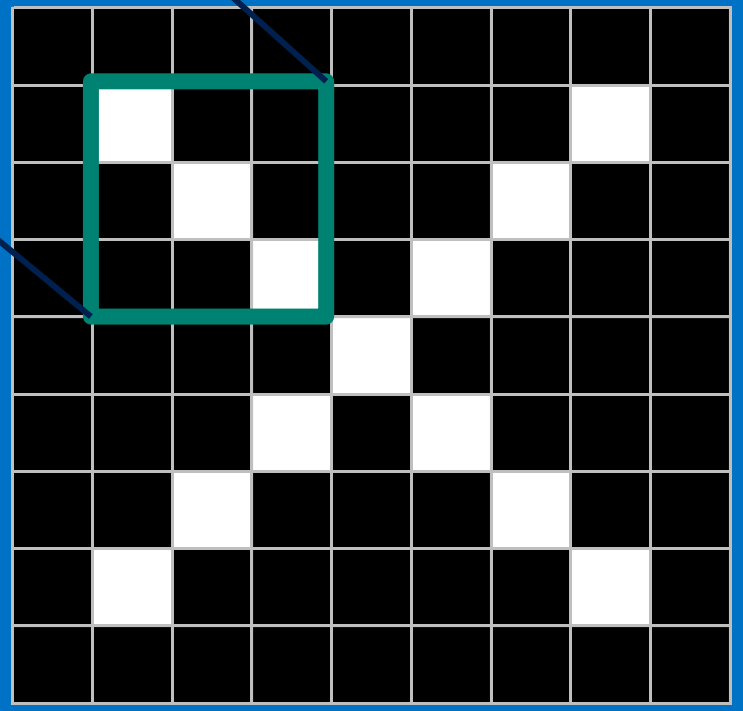




1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

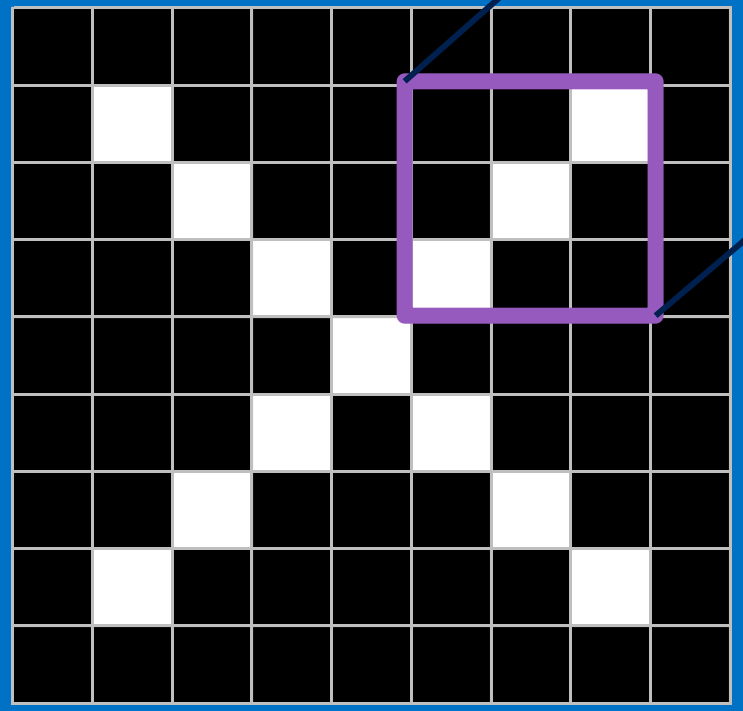
-1	-1	1
-1	1	-1
1	-1	-1



1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

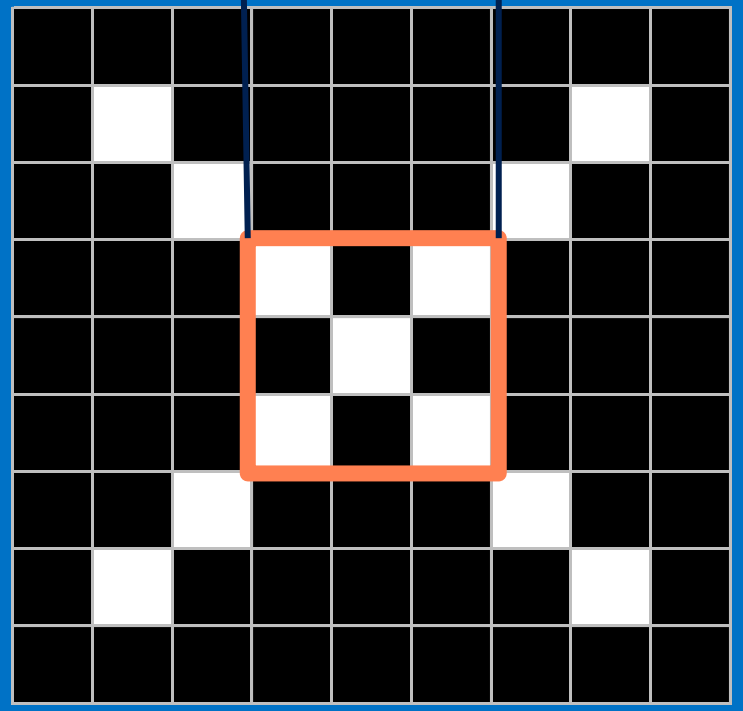
-1	-1	1
-1	1	-1
1	-1	-1



1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

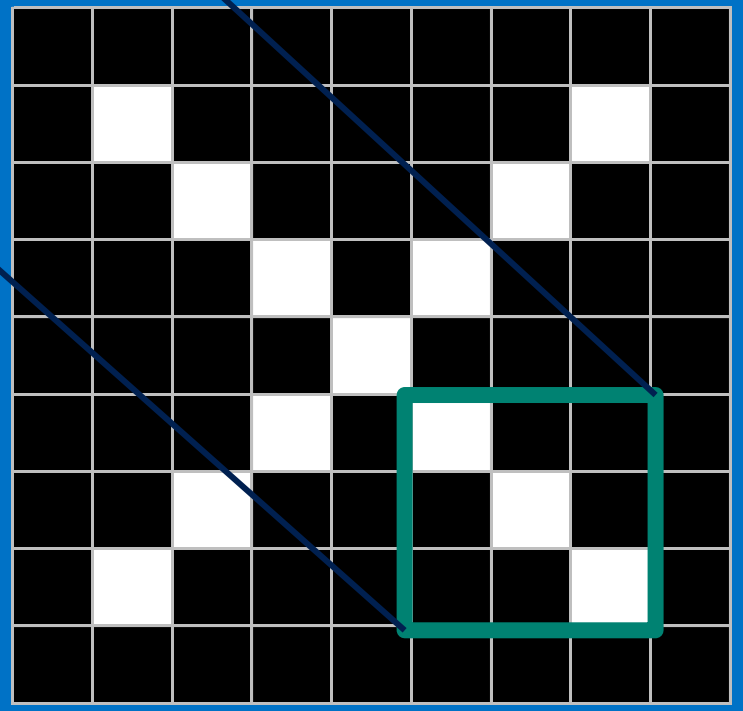
-1	-1	1
-1	1	-1
1	-1	-1



1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

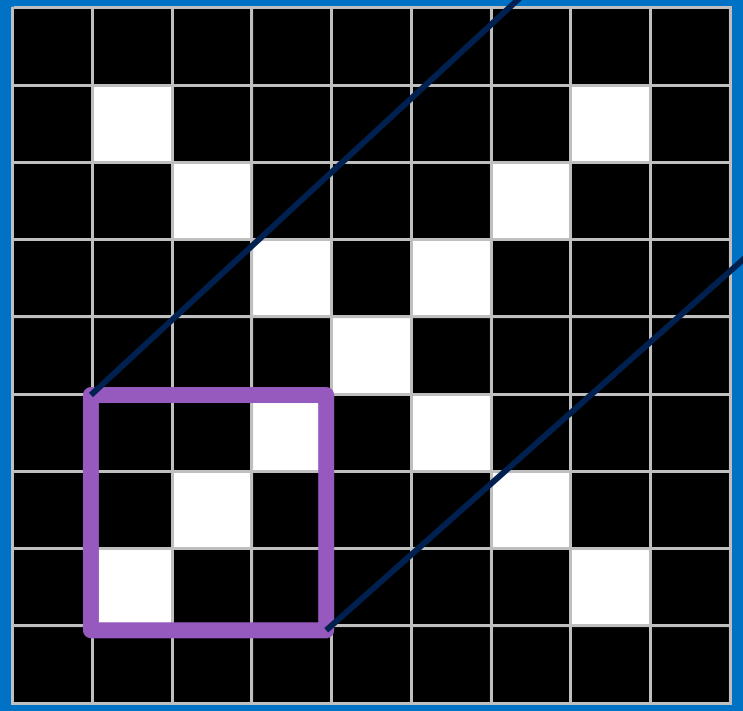
-1	-1	1
-1	1	-1
1	-1	-1



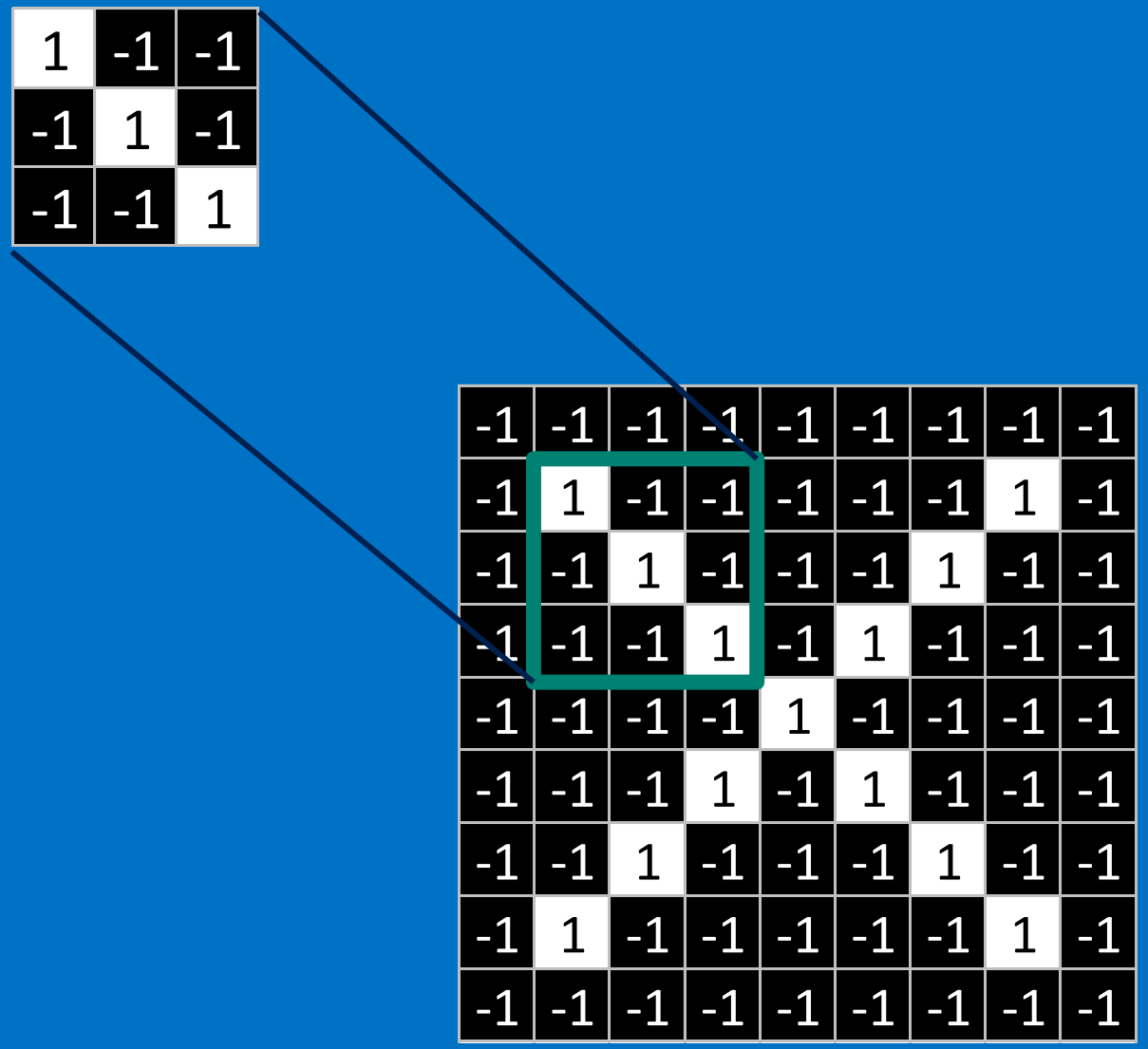
1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

-1	-1	1
-1	1	-1
1	-1	-1



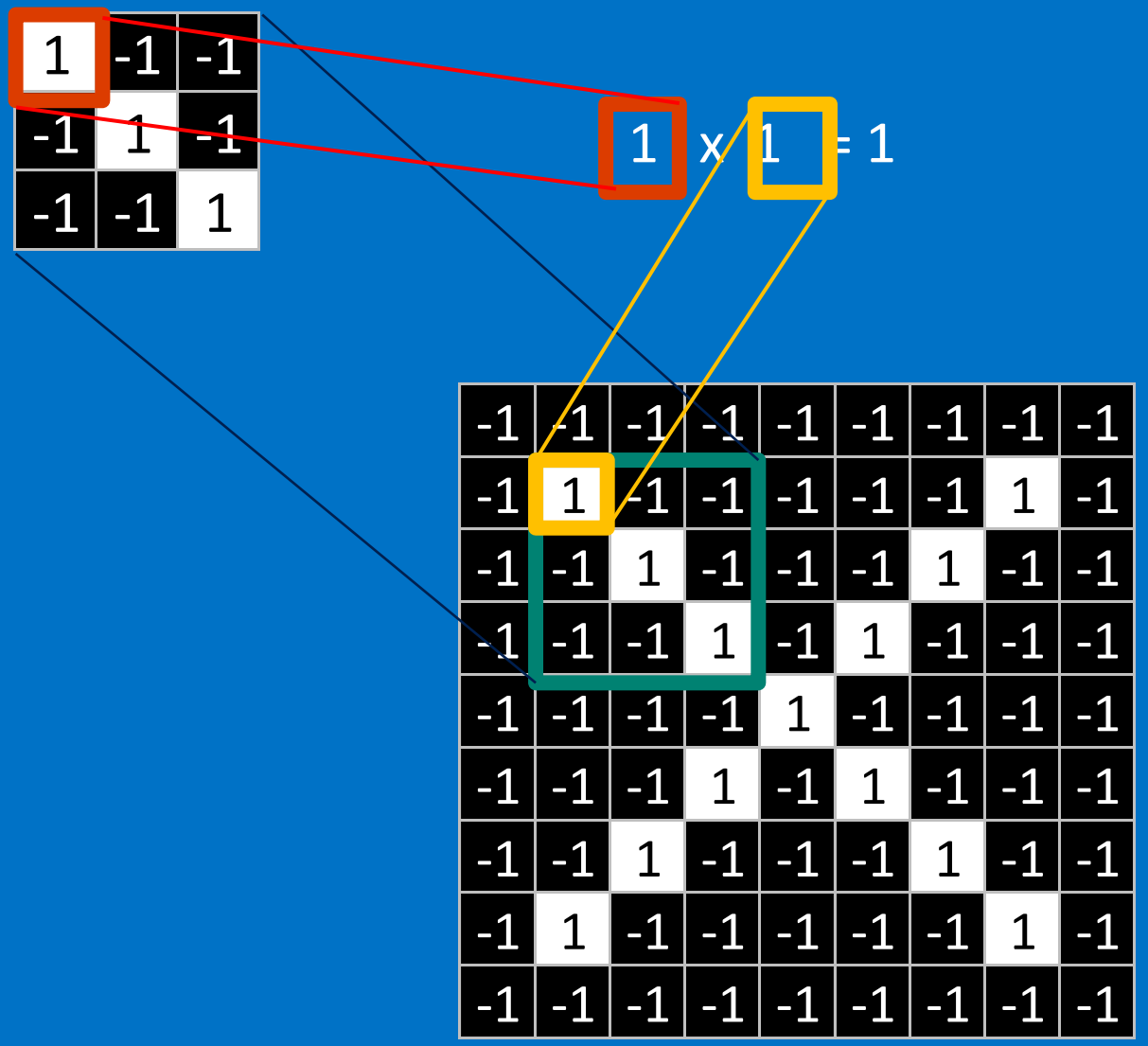
Filtering: The math behind the match



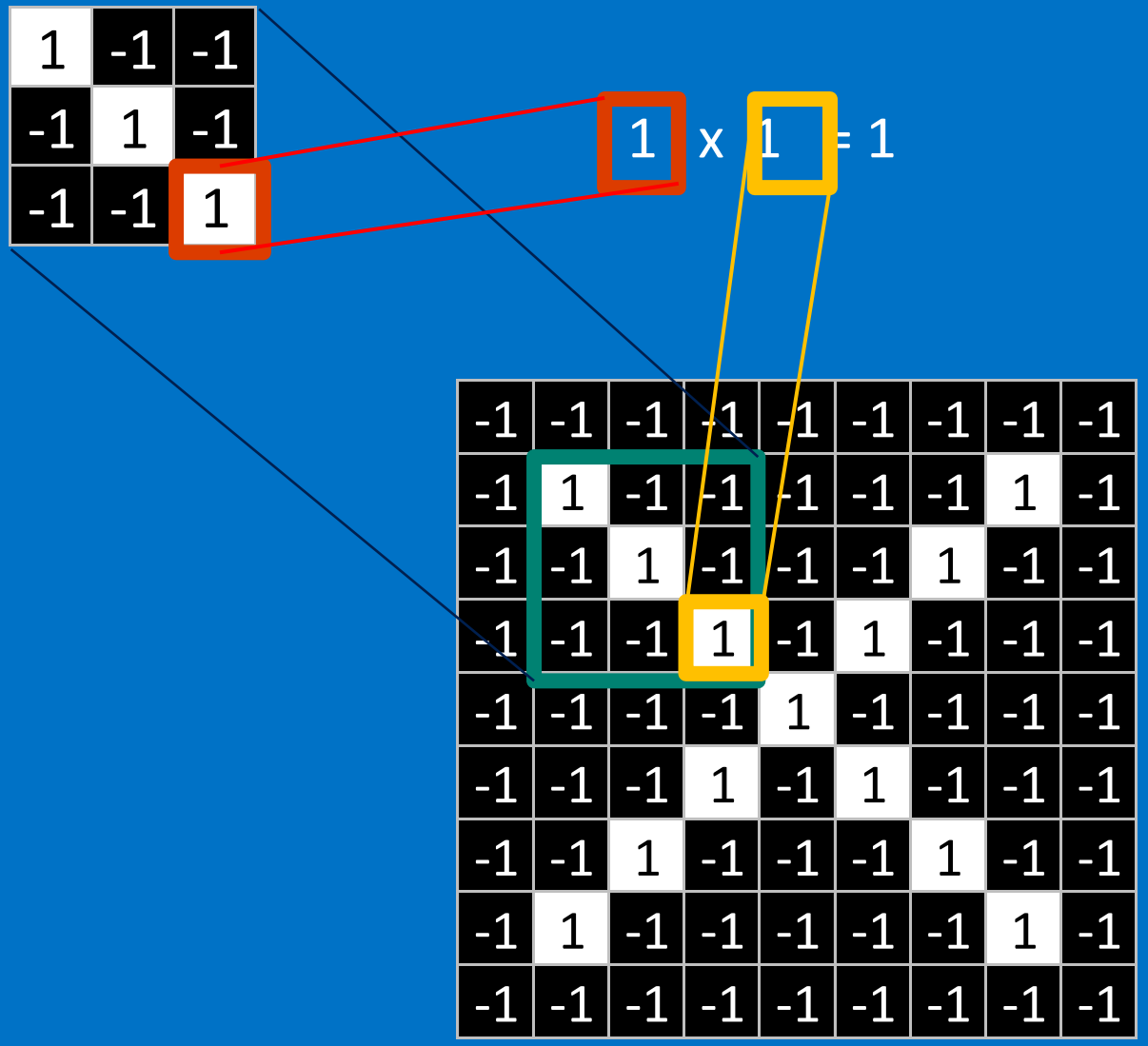
Filtering: The math behind the match

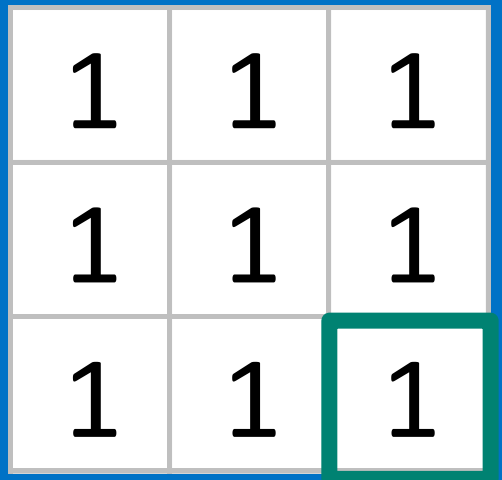
1. Line up the feature and the image patch.
2. Multiply each image pixel by the corresponding feature pixel.
3. Add them up.
4. Divide by the total number of pixels in the feature.

Filtering: The math behind the match

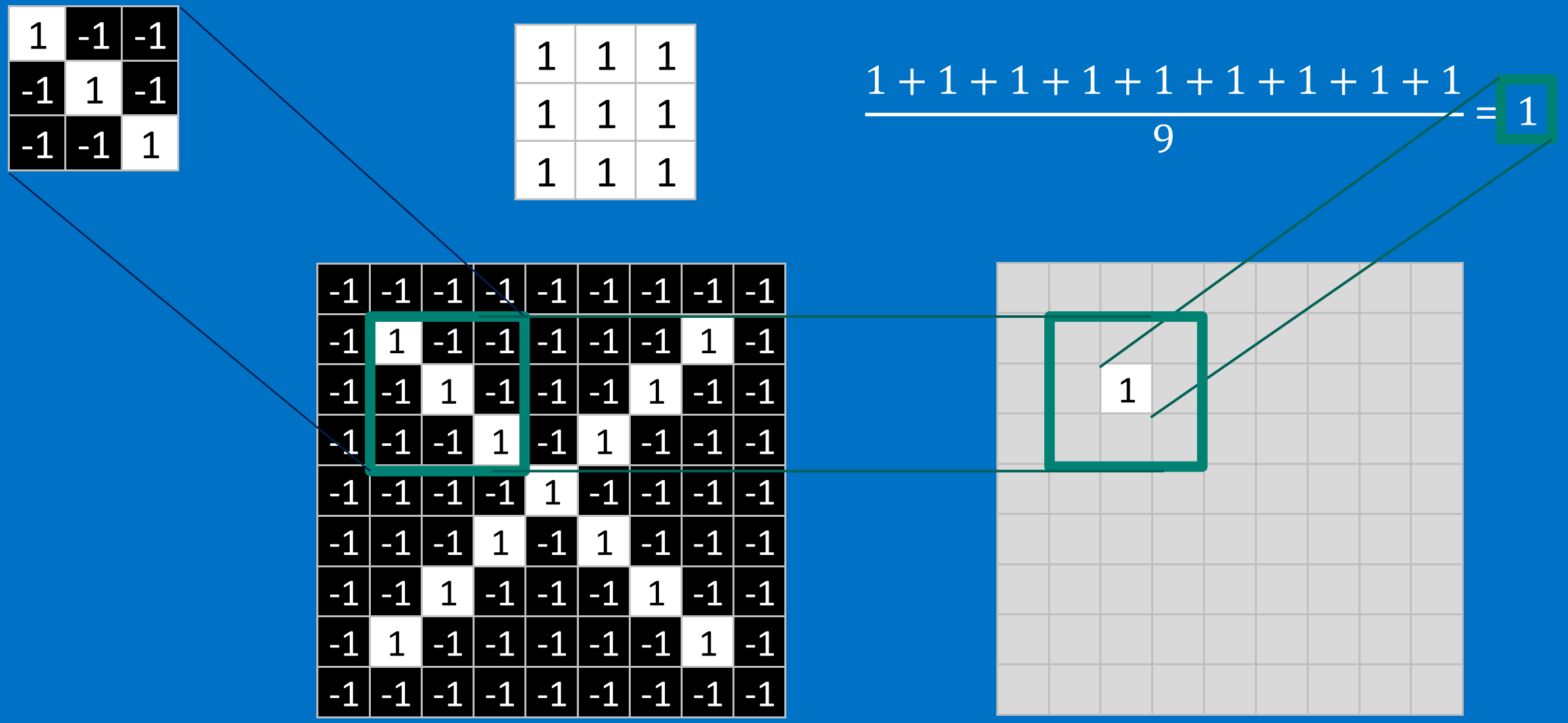


Filtering: The math behind the match





Filtering: The math behind the match

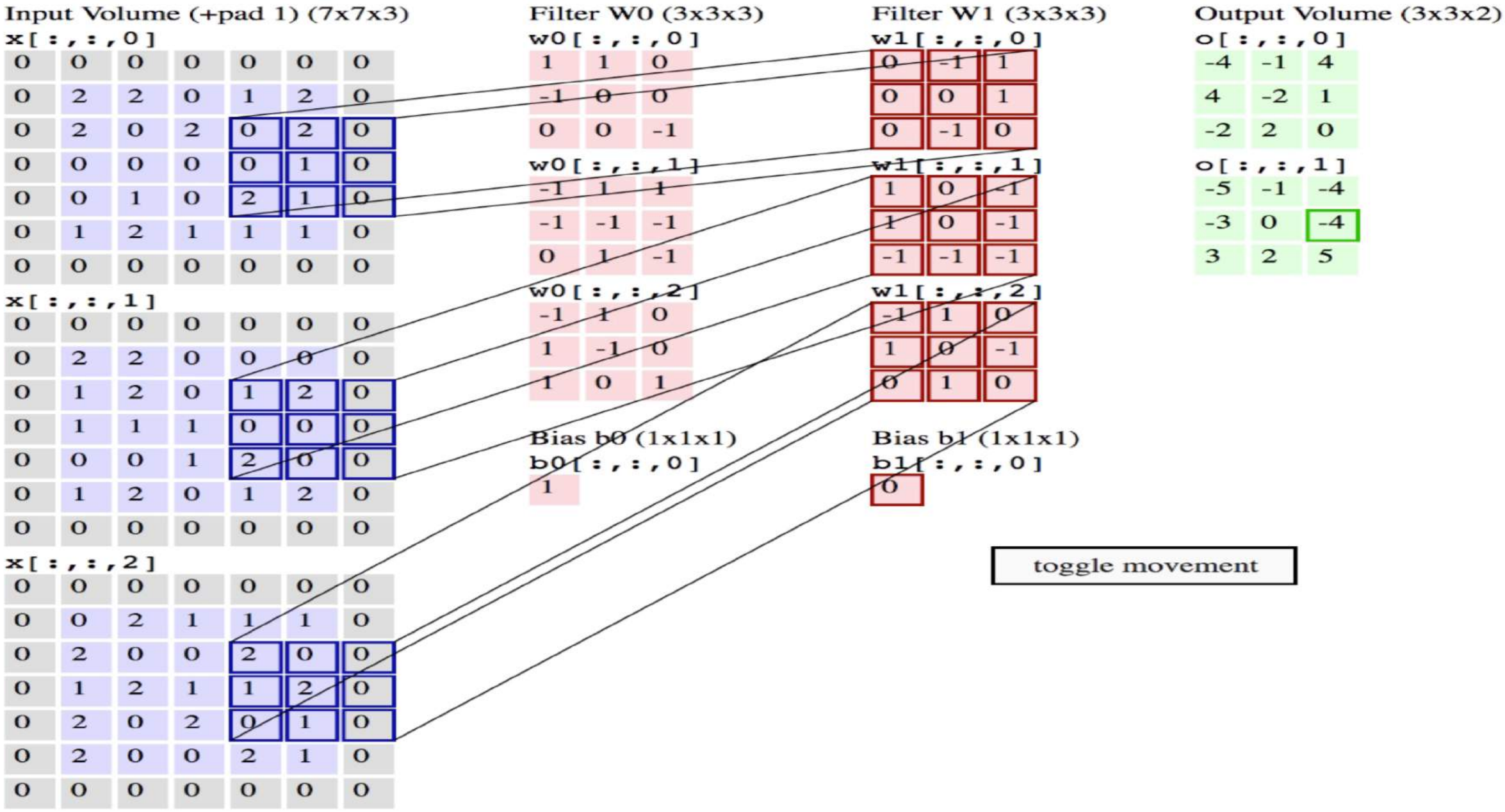


Convolution: Trying every possible match

1	-1	-1
-1	1	-1
-1	-1	1

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 2 0 1 2 0
0 2 0 2 0 2 0
0 0 0 0 0 1 0
0 0 1 0 2 1 0
0 1 2 1 1 1 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 2 2 0 0 0 0
0 1 2 0 1 2 0
0 1 1 1 0 0 0
0 0 0 1 2 0 0
0 1 2 0 1 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 2 1 1 1 0
0 2 0 0 2 0 0
0 1 2 1 1 2 0
0 2 0 2 0 1 0
0 2 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 1 0
-1 0 0
0 0 -1
w0[:,:,1]
-1 1 1
-1 -1 -1
0 1 -1
w0[:,:,2]
-1 1 0
1 -1 0
1 0 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
0 -1 1
0 0 1
0 -1 0
w1[:,:,1]
1 0 -1
1 0 -1
-1 -1 -1
w1[:,:,2]
-1 1 0
1 0 -1
0 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
-4 -1 4
4 -2 1
-2 2 0
o[:,:,1]
-5 -1 -4
-3 0 -4
3 2 5
toggle movement

Convolution: Trying every possible match

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

$\otimes$

1	-1	-1
-1	1	-1
-1	-1	1

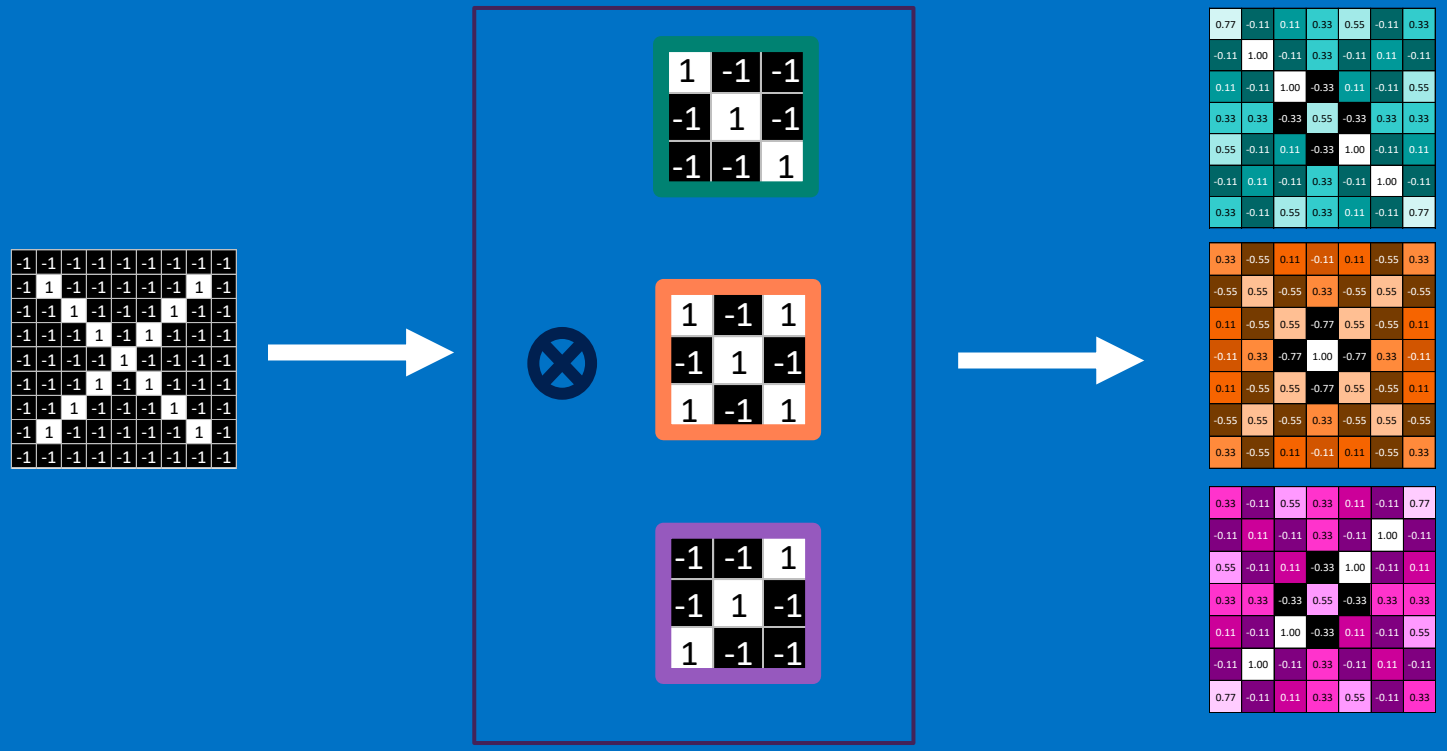
$=$

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77


-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 1 -1 -1 -1 -1 -1 1 -1
-1 -1 1 -1 -1 -1 1 -1 -1
-1 -1 -1 1 -1 1 -1 -1 -1
-1 -1 -1 -1 1 -1 -1 -1 -1
-1 -1 -1 1 -1 1 -1 -1 -1
-1 -1 1 -1 -1 -1 1 -1 -1
-1 1 -1 -1 -1 -1 -1 1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
1 -1 -1
-1 1 -1
-1 -1 1
=
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1 -1 1
-1 1 -1
1 -1 1
=
0.33 -0.55 0.11 -0.11 0.11 -0.55 0.33
-0.55 0.55 -0.55 0.33 -0.55 0.55 -0.55
0.11 -0.55 0.55 -0.77 0.55 -0.55 0.11
-0.11 0.33 -0.77 1.00 -0.77 0.33 -0.11
0.11 -0.55 0.55 -0.77 0.55 -0.55 0.11
-0.55 0.55 -0.55 0.33 -0.55 0.55 -0.55
0.33 -0.55 0.11 -0.11 0.11 -0.55 0.33
-1 -1 1
-1 1 -1
1 -1 -1
=
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33

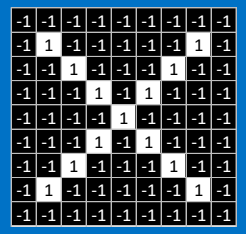
Convolution layer

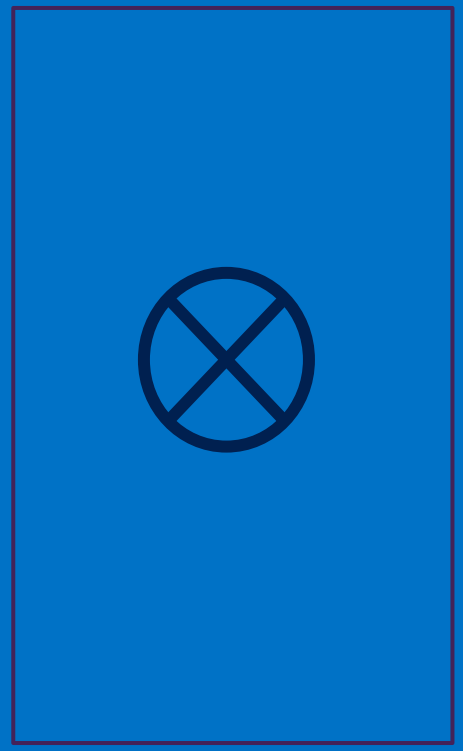
One image becomes a stack of filtered images

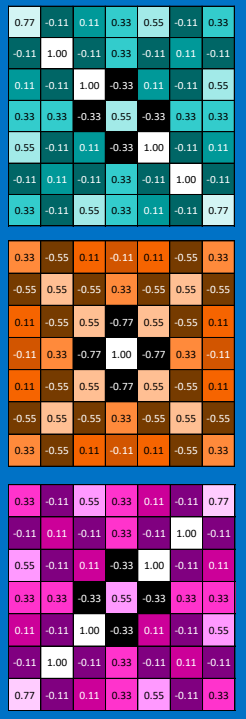


Convolution layer

One image becomes a stack of filtered images



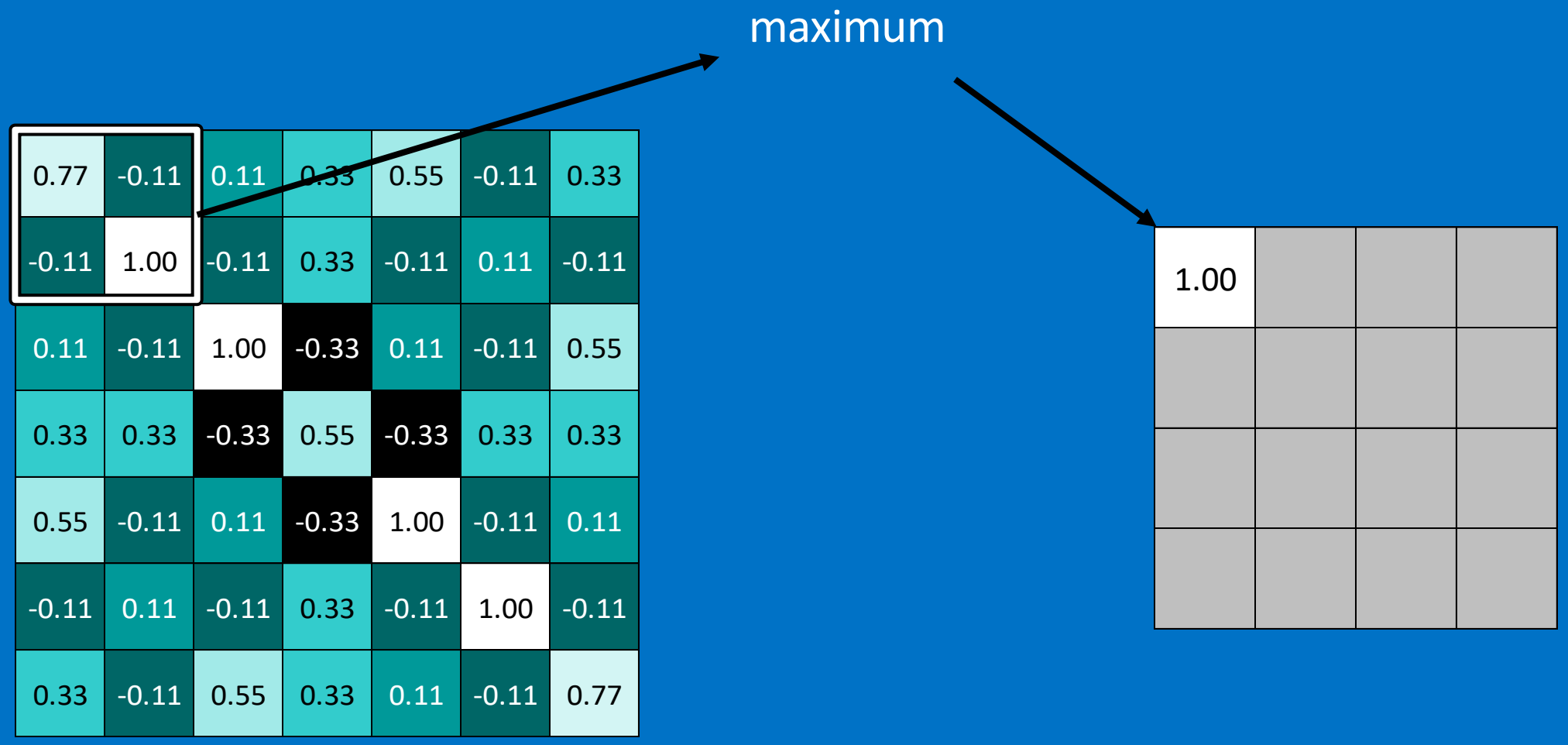




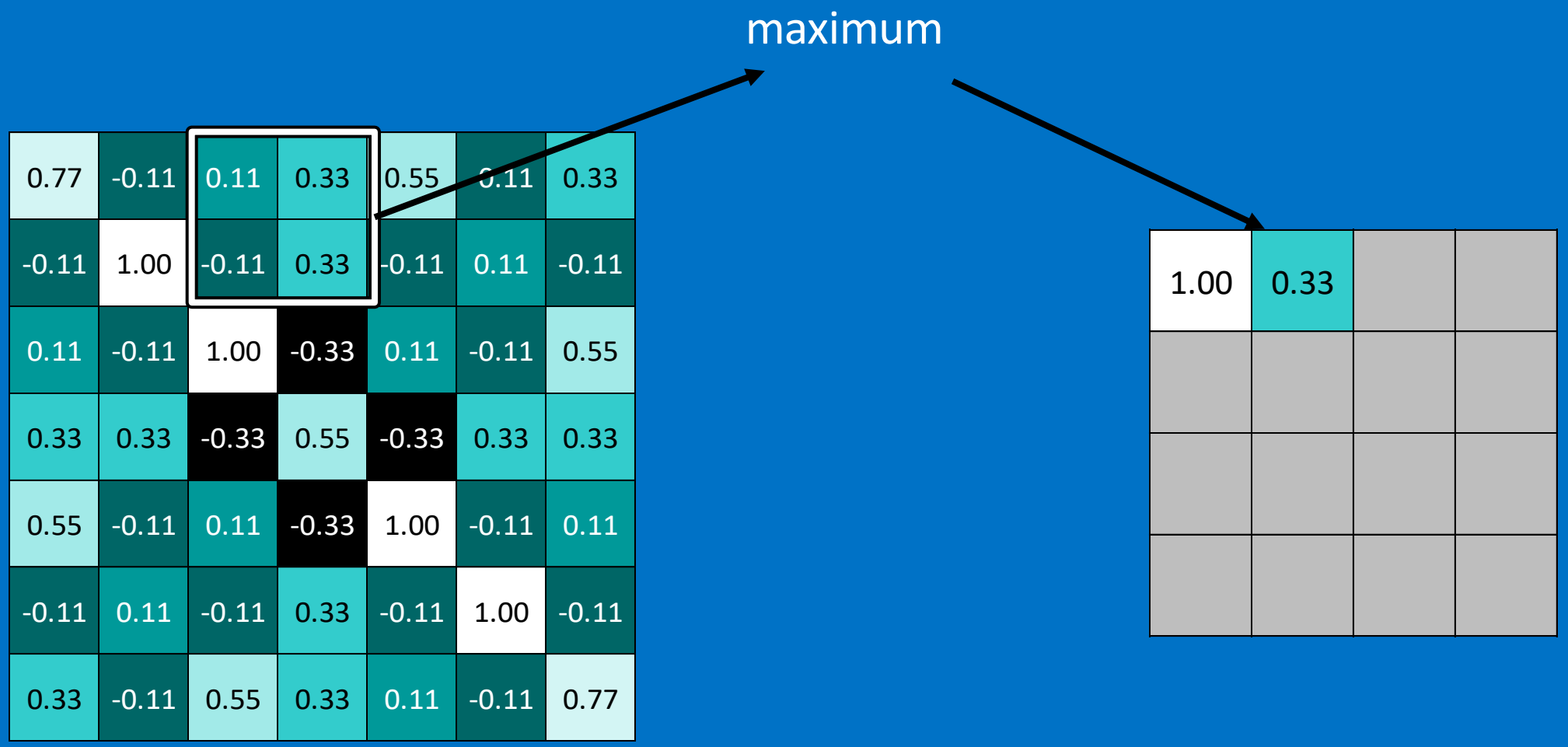
Pooling: Shrinking the image stack

1. Pick a window size (usually 2 or 3).
2. Pick a stride (usually 2).
3. Walk your window across your filtered images.
4. From each window, take the maximum value.

Pooling


maximum
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00

Pooling

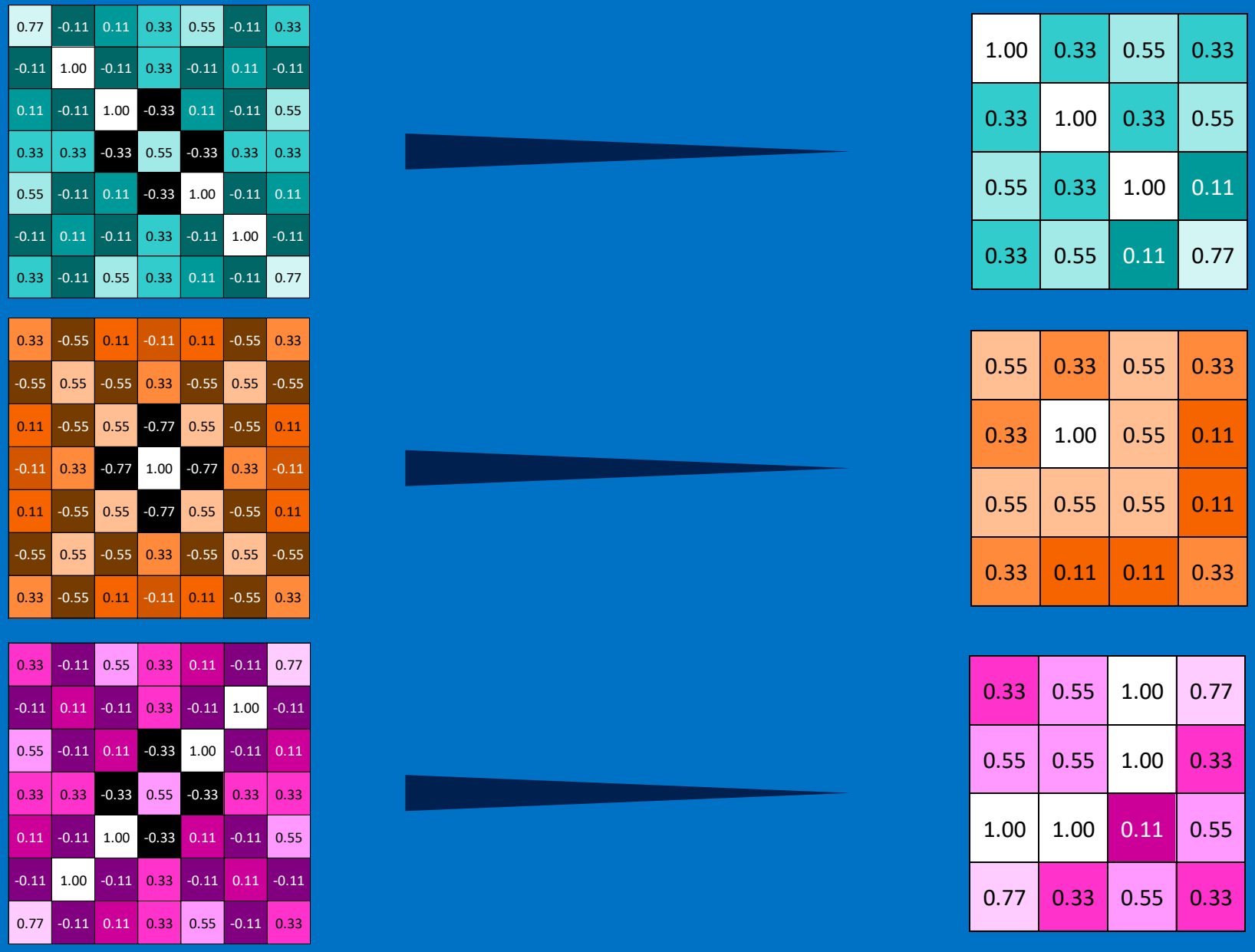

maximum
0.77 -0.11 0.11 0.33 0.55 0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00 0.33

Pooling

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

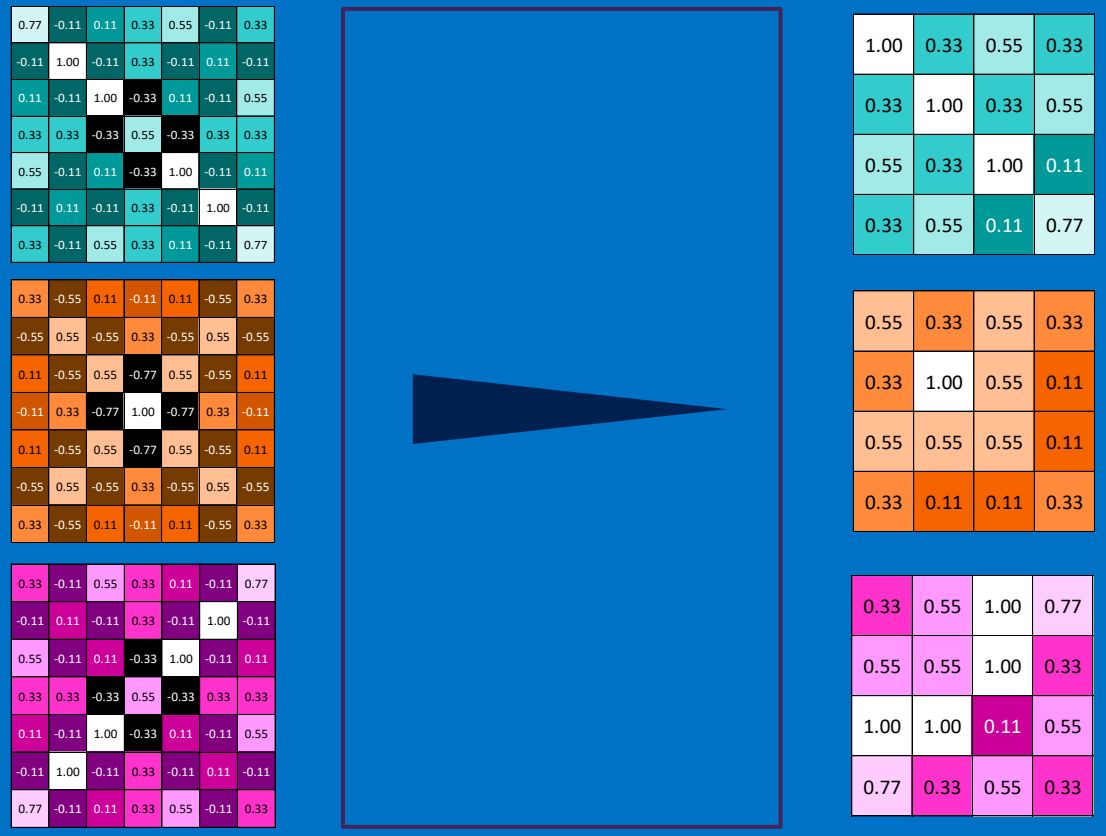
max pooling

1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77


0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00 0.33 0.55 0.33
0.33 1.00 0.33 0.55
0.55 0.33 1.00 0.11
0.33 0.55 0.11 0.77
0.33 -0.55 0.11 -0.11 0.11 -0.55 0.33
-0.55 0.55 -0.55 0.33 -0.55 0.55 -0.55
0.11 -0.55 0.55 -0.77 0.55 -0.55 0.11
-0.11 0.33 -0.77 1.00 -0.77 0.33 -0.11
0.11 -0.55 0.55 -0.77 0.55 -0.55 0.11
-0.55 0.55 -0.55 0.33 -0.55 0.55 -0.55
0.33 -0.55 0.11 -0.11 0.11 -0.55 0.33
0.55 0.33 0.55 0.33
0.33 1.00 0.55 0.11
0.55 0.55 0.55 0.11
0.33 0.11 0.11 0.33
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
0.33 0.55 1.00 0.77
0.55 0.55 1.00 0.33
1.00 1.00 0.11 0.55
0.77 0.33 0.55 0.33

Pooling layer

A stack of images becomes a stack of smaller images.

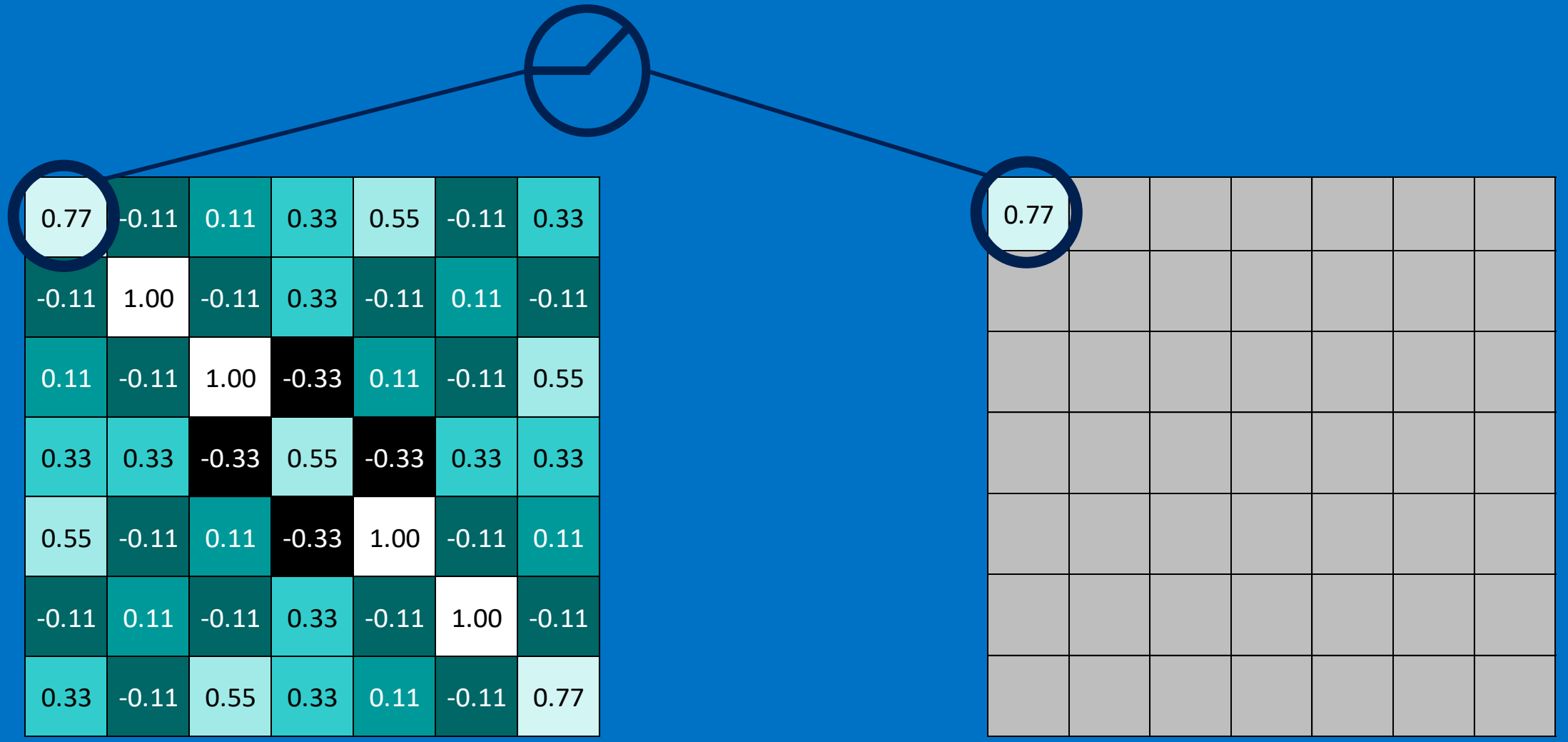


Normalization

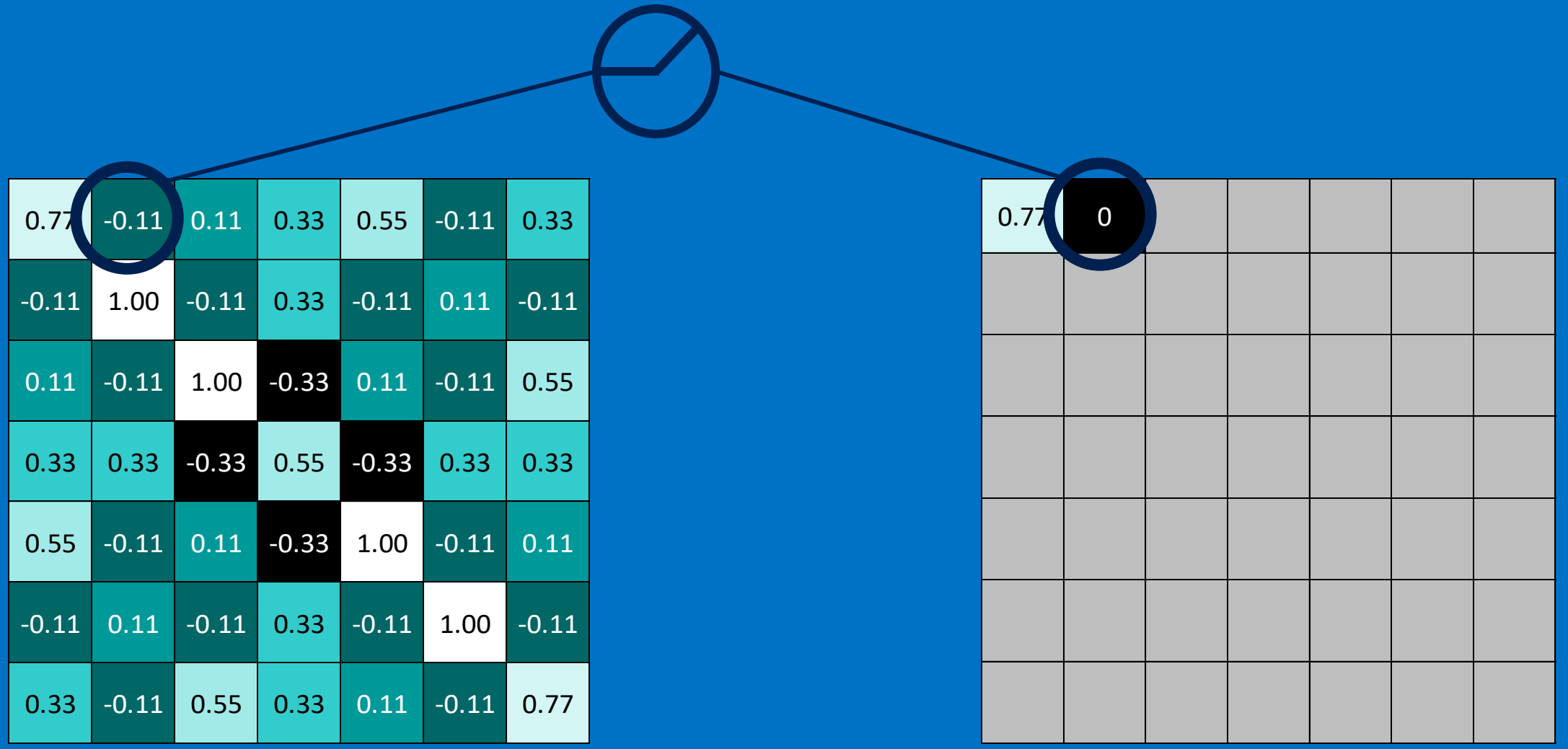
Keep the math from breaking by tweaking each of the values just a bit.

Change everything negative to zero.

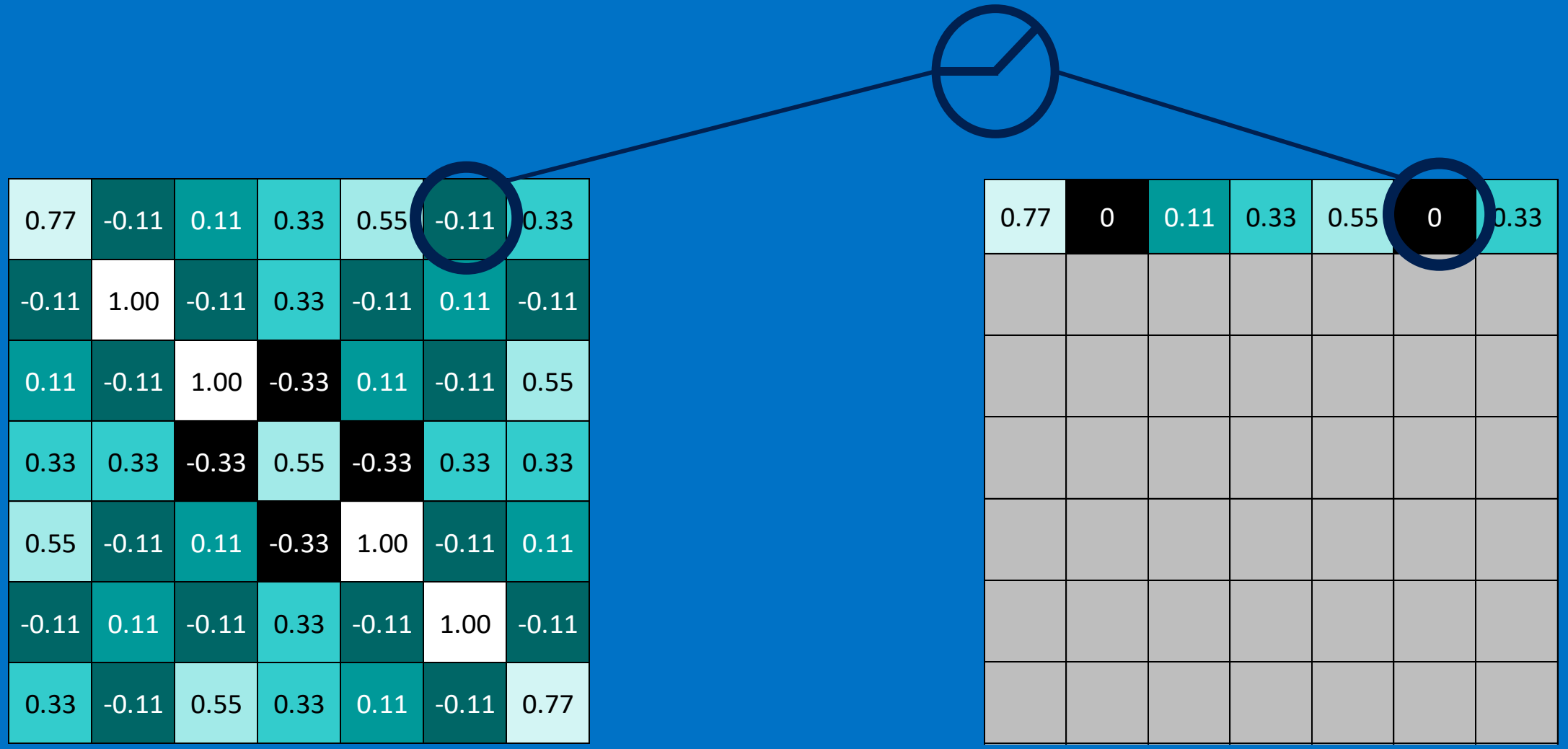
Rectified Linear Units (ReLUs)



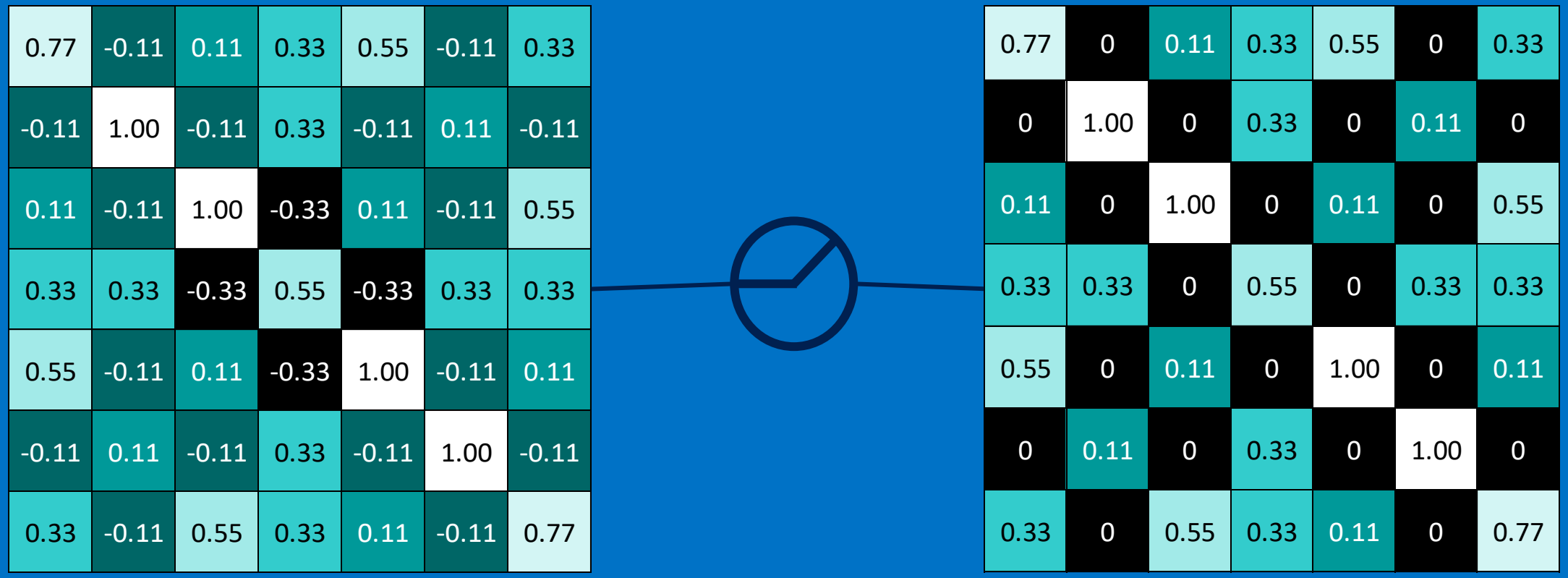
Rectified Linear Units (ReLUs)



Rectified Linear Units (ReLUs)

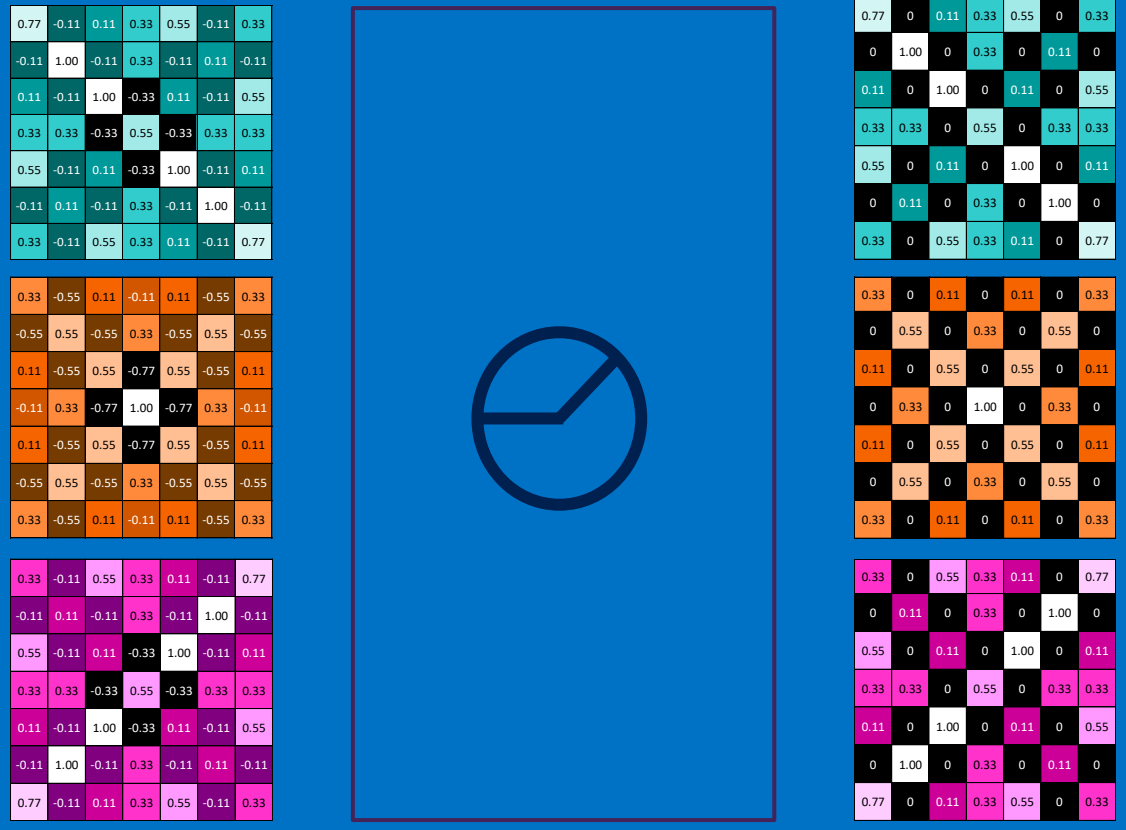


Rectified Linear Units (ReLUs)



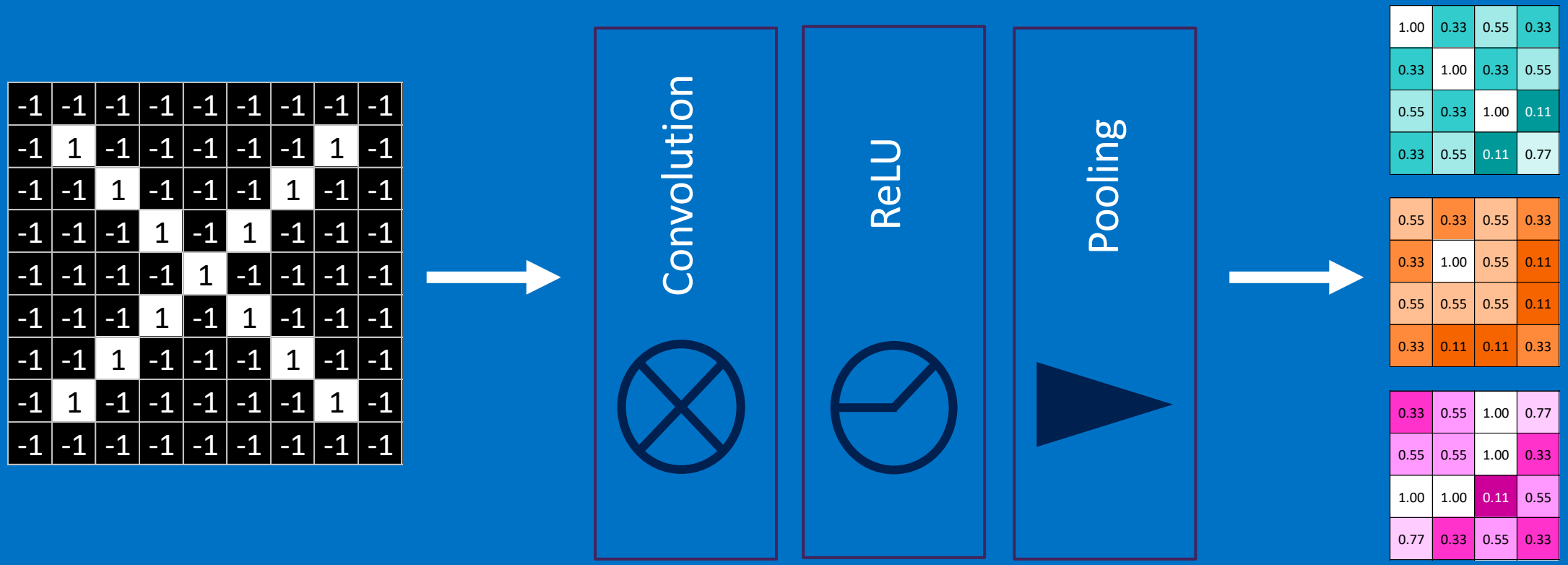
ReLU layer

A stack of images becomes a stack of images with no negative values.



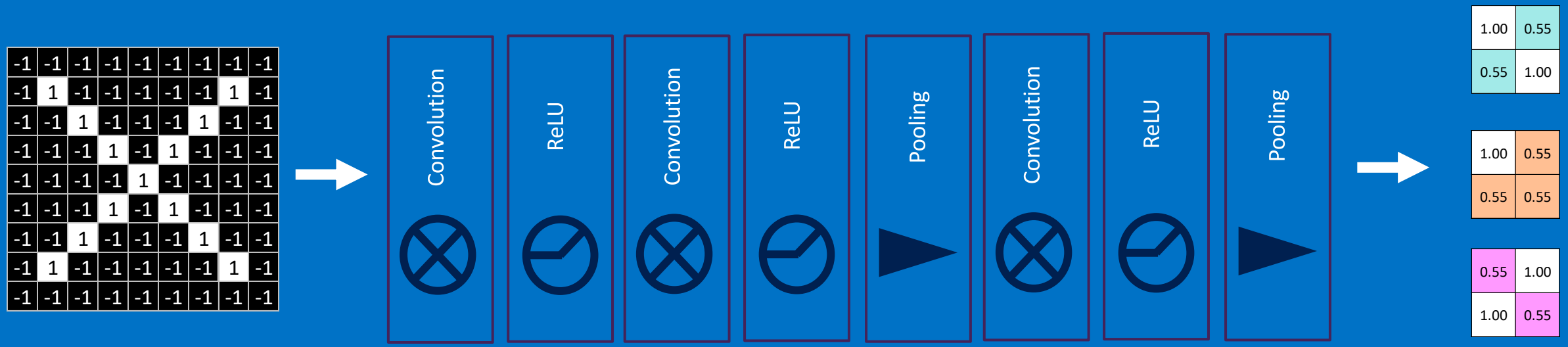
Layers get stacked

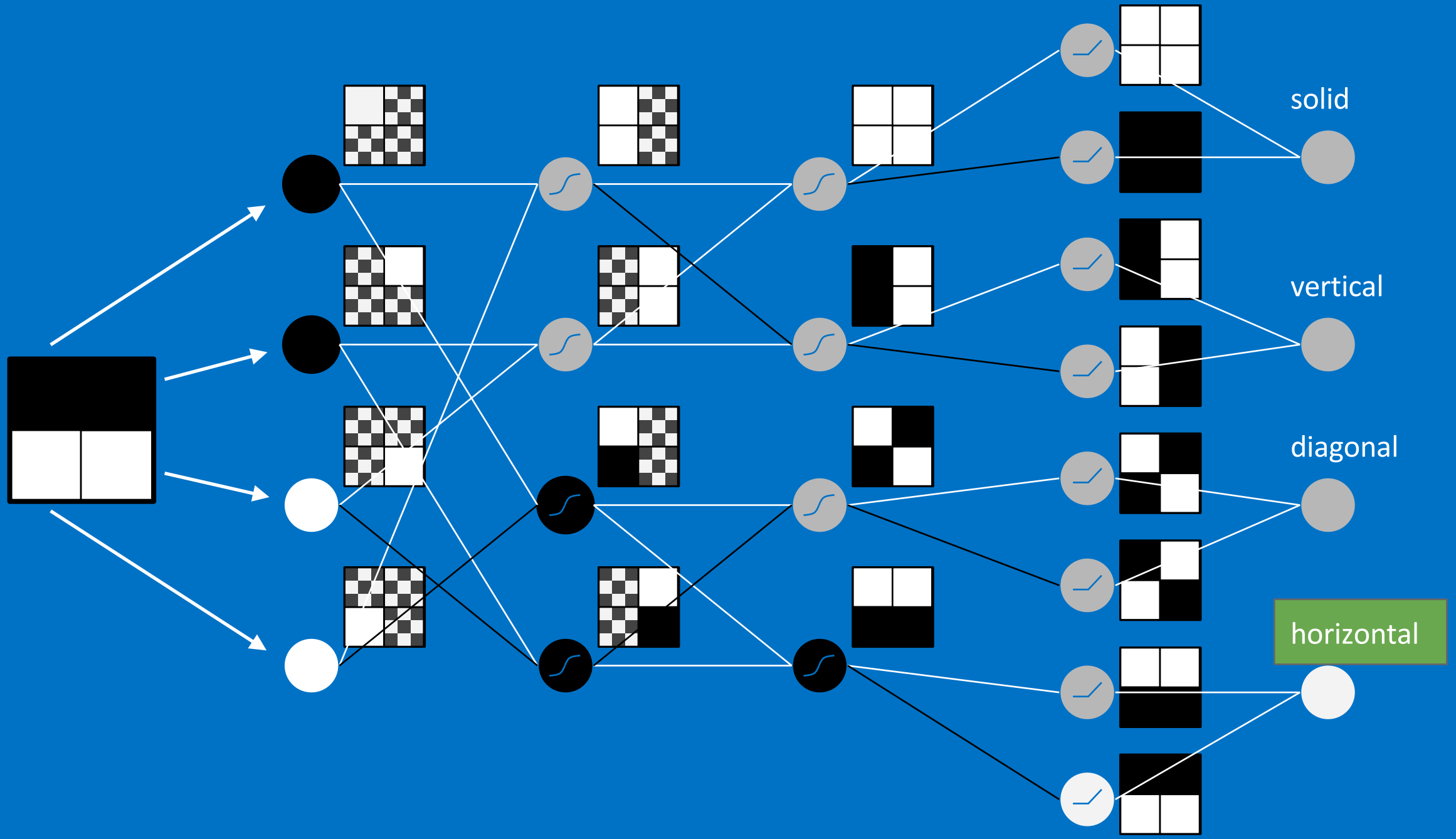
The output of one becomes the input of the next.



Deep stacking

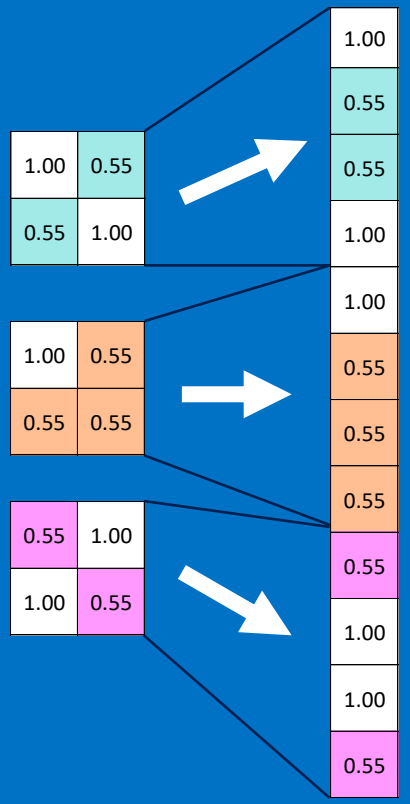
Layers can be repeated several (or many) times.




solid
vertical
diagonal
horizontal

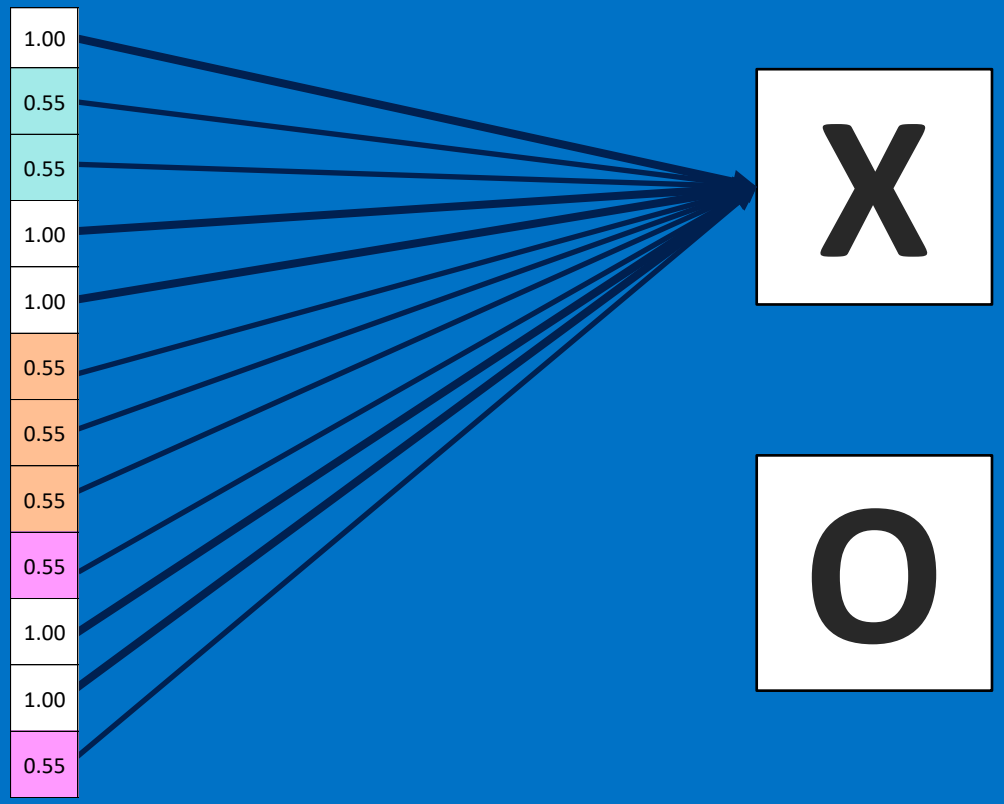
Fully connected layer

Every value gets a vote



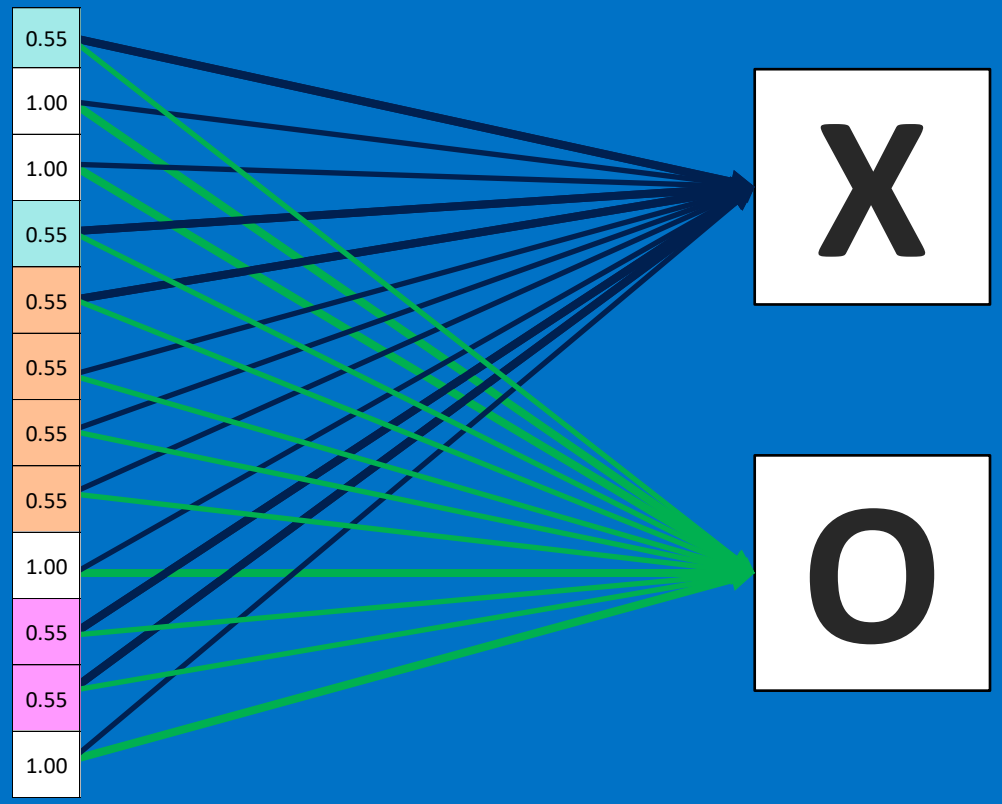
Fully connected layer

Vote depends on how strongly a value predicts X or O



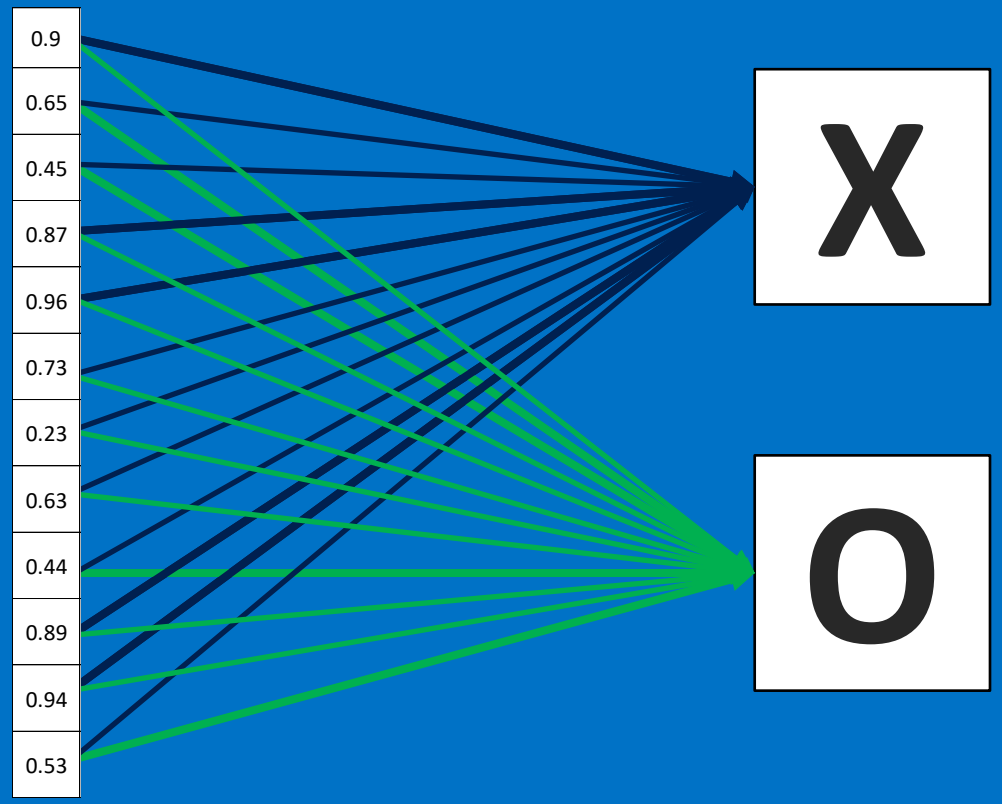
Fully connected layer

Vote depends on how strongly a value predicts X or O



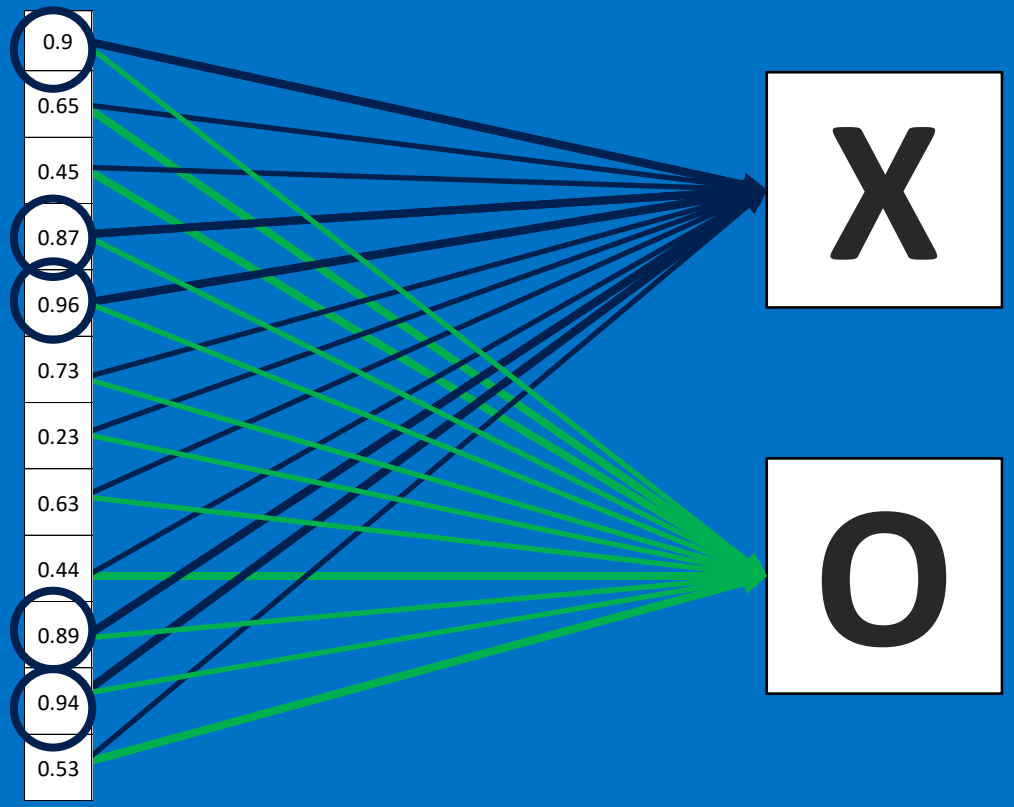
Fully connected layer

Future values vote on X or O



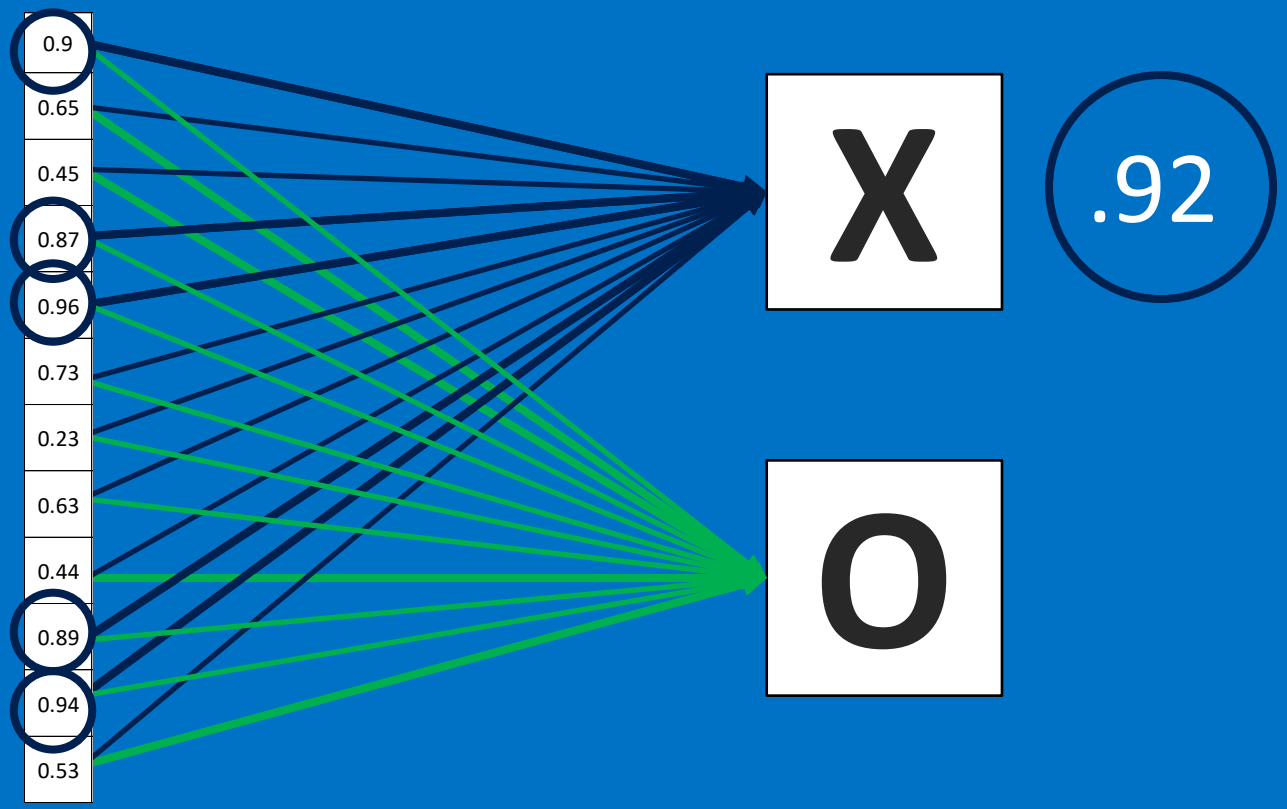
Fully connected layer

Future values vote on X or O



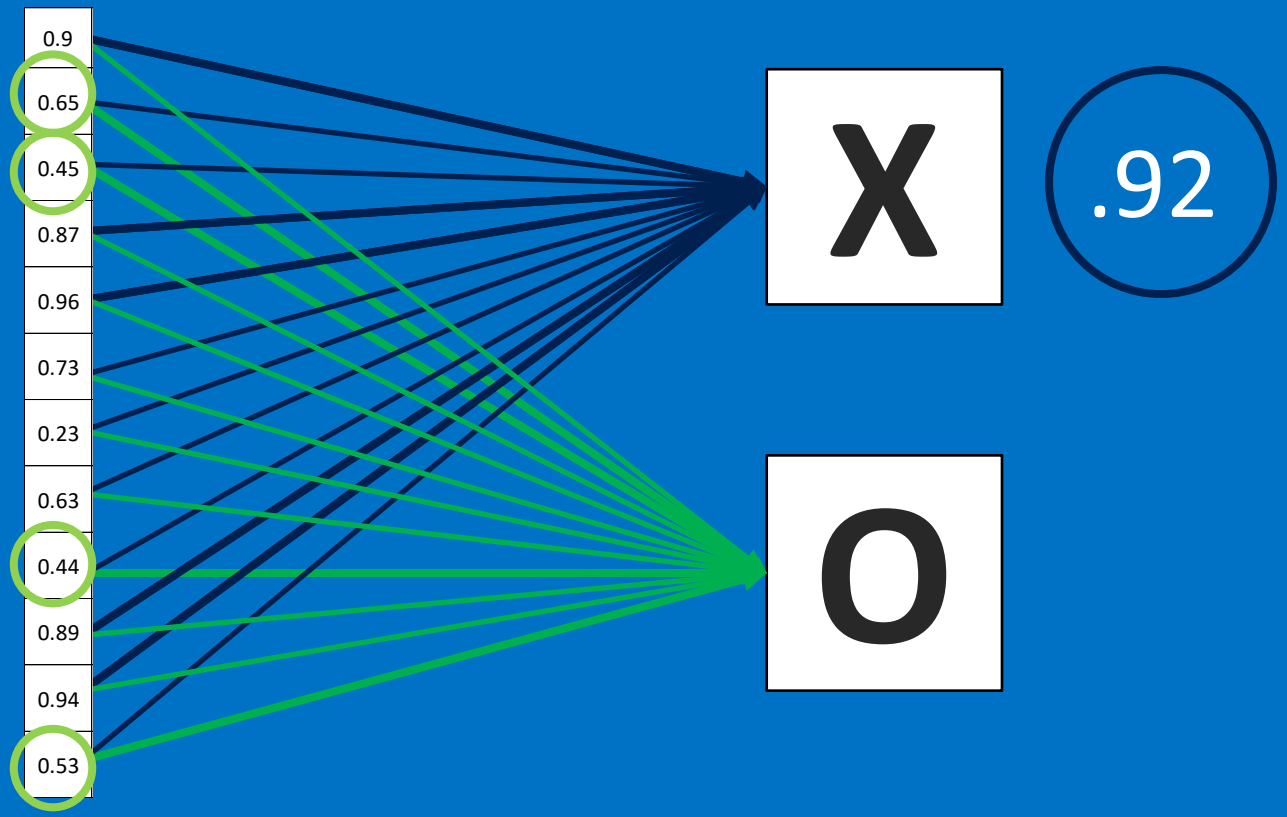
Fully connected layer

Future values vote on X or O



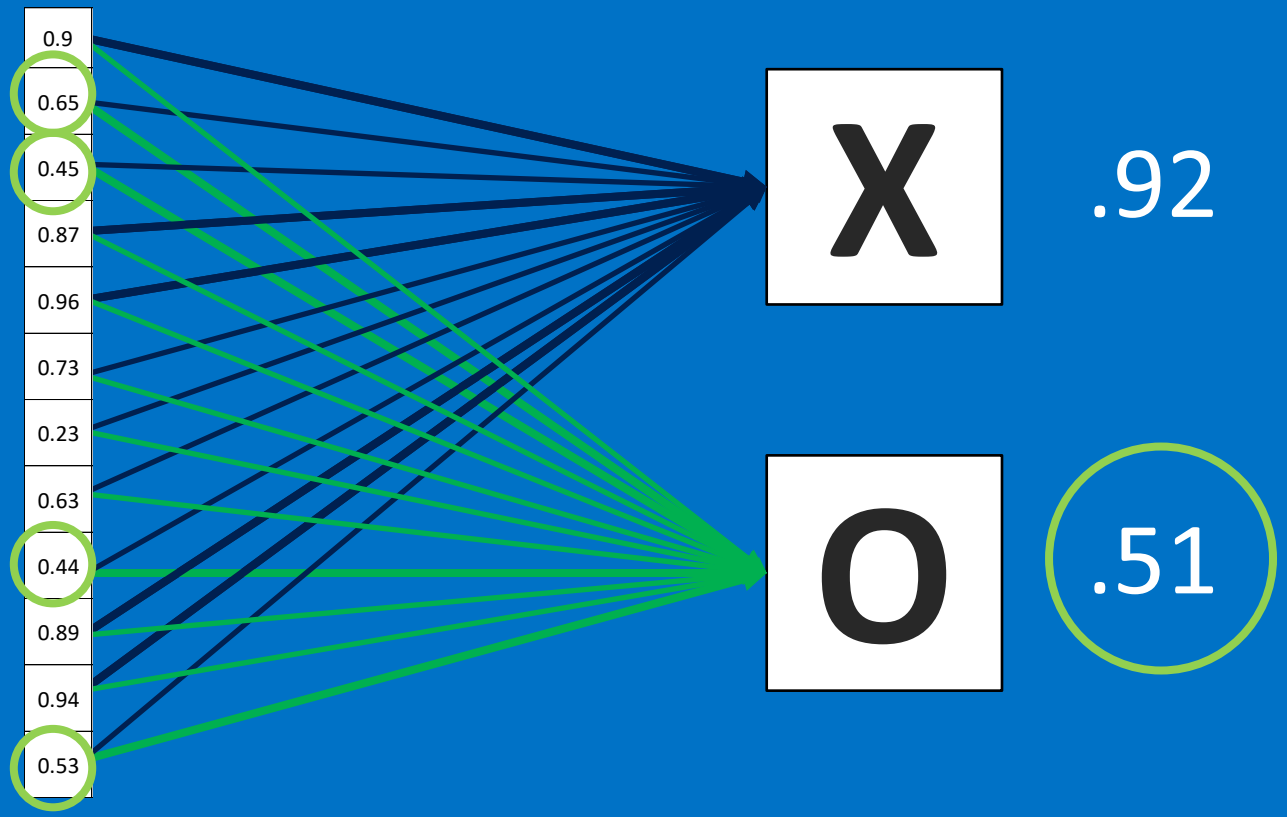
Fully connected layer

Future values vote on X or O



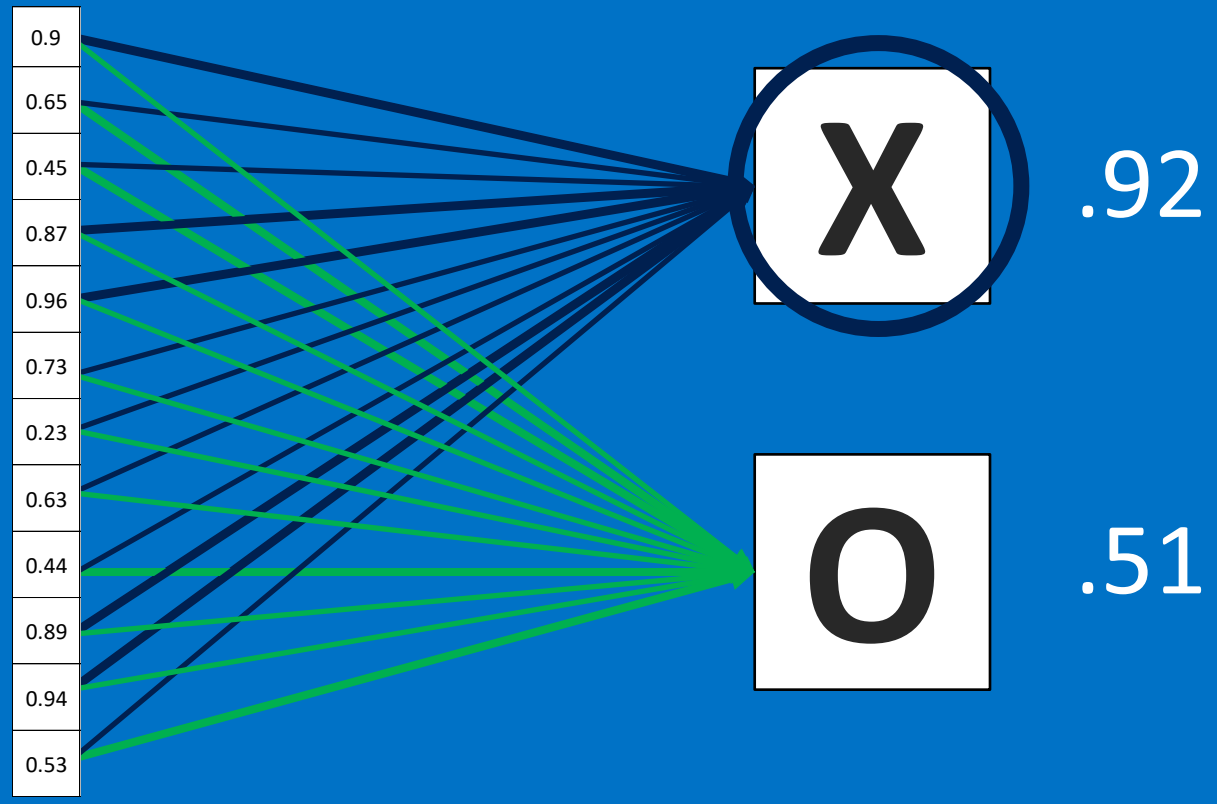
Fully connected layer

Future values vote on X or O



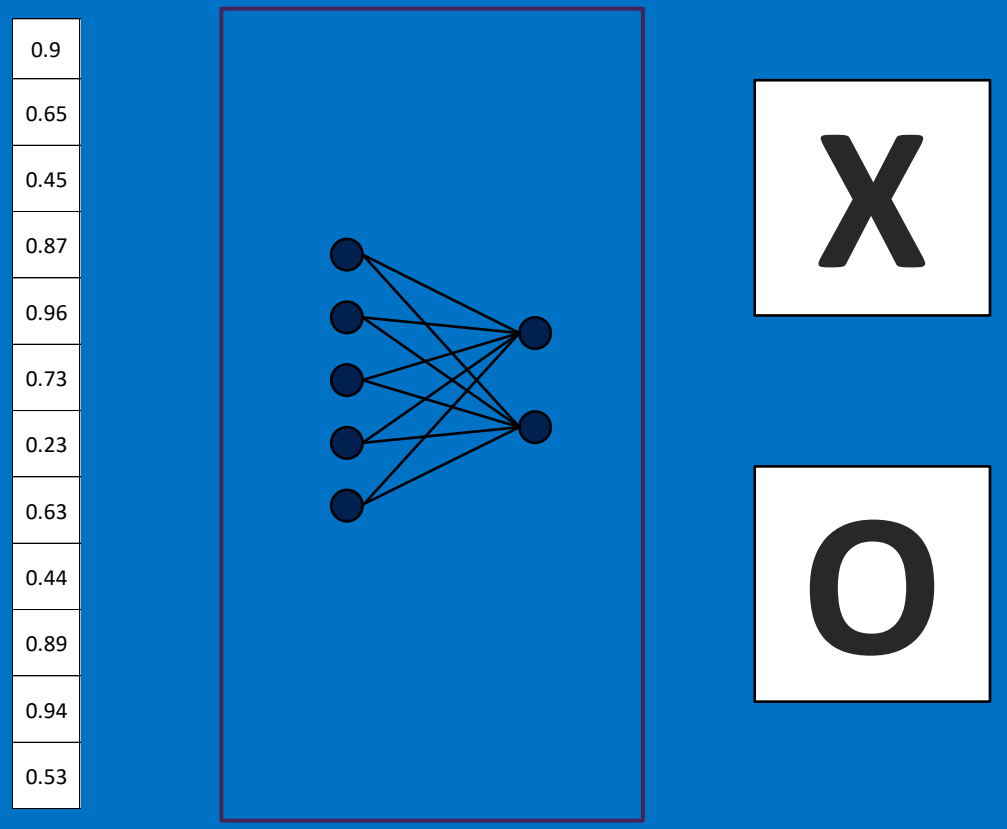
Fully connected layer

Future values vote on X or O



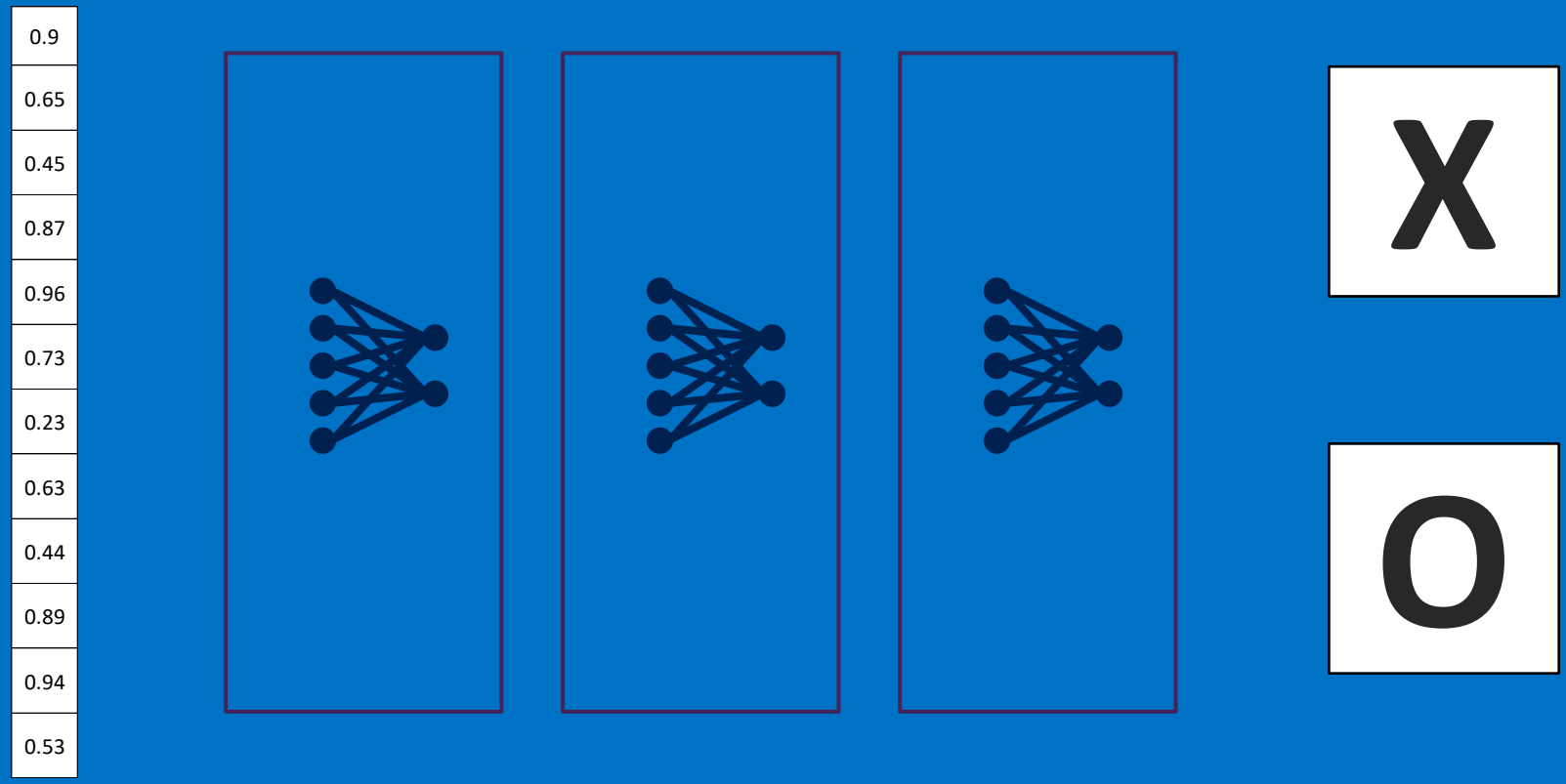
Fully connected layer

A list of feature values becomes a list of votes.



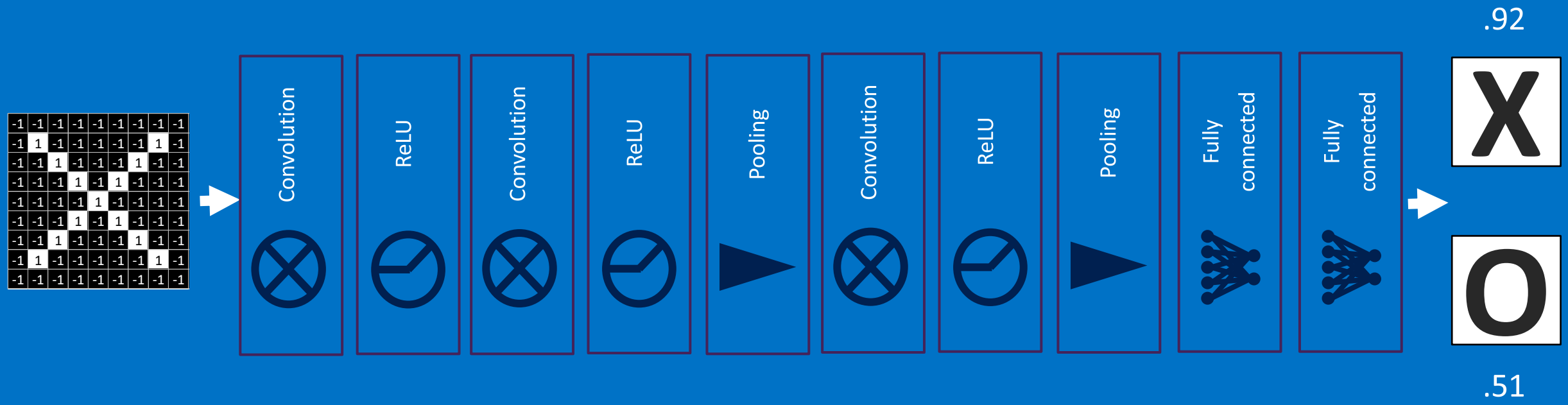
Fully connected layer

These can also be stacked.



Putting it all together

A set of pixels becomes a set of votes.



Learning

Q: Where do all the magic numbers come from?

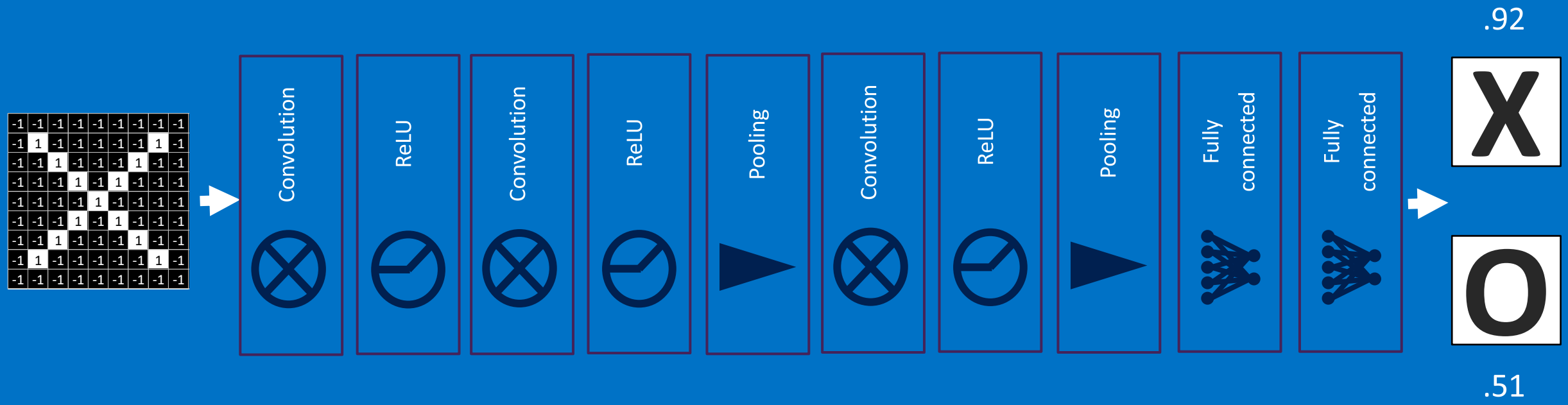
Features in convolutional layers

Voting weights in fully connected layers

A: Backpropagation

Backprop

Error = right answer – actual answer



Hyperparameters (knobs)

- Convolution
  - Number of features
  - Size of features
- Pooling
  - Window size
  - Window stride
- Fully Connected
  - Number of neurons

Architecture

How many of each type of layer?

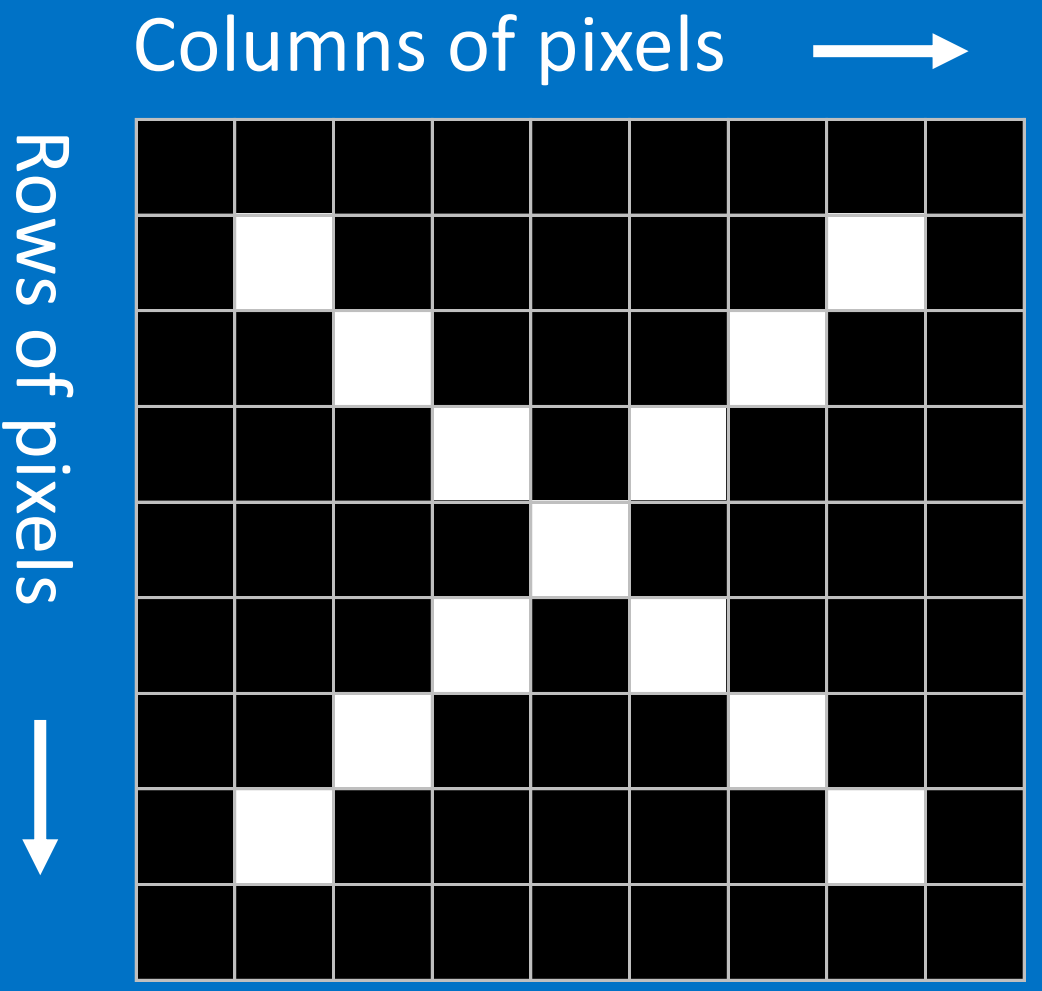
In what order?

Not just images

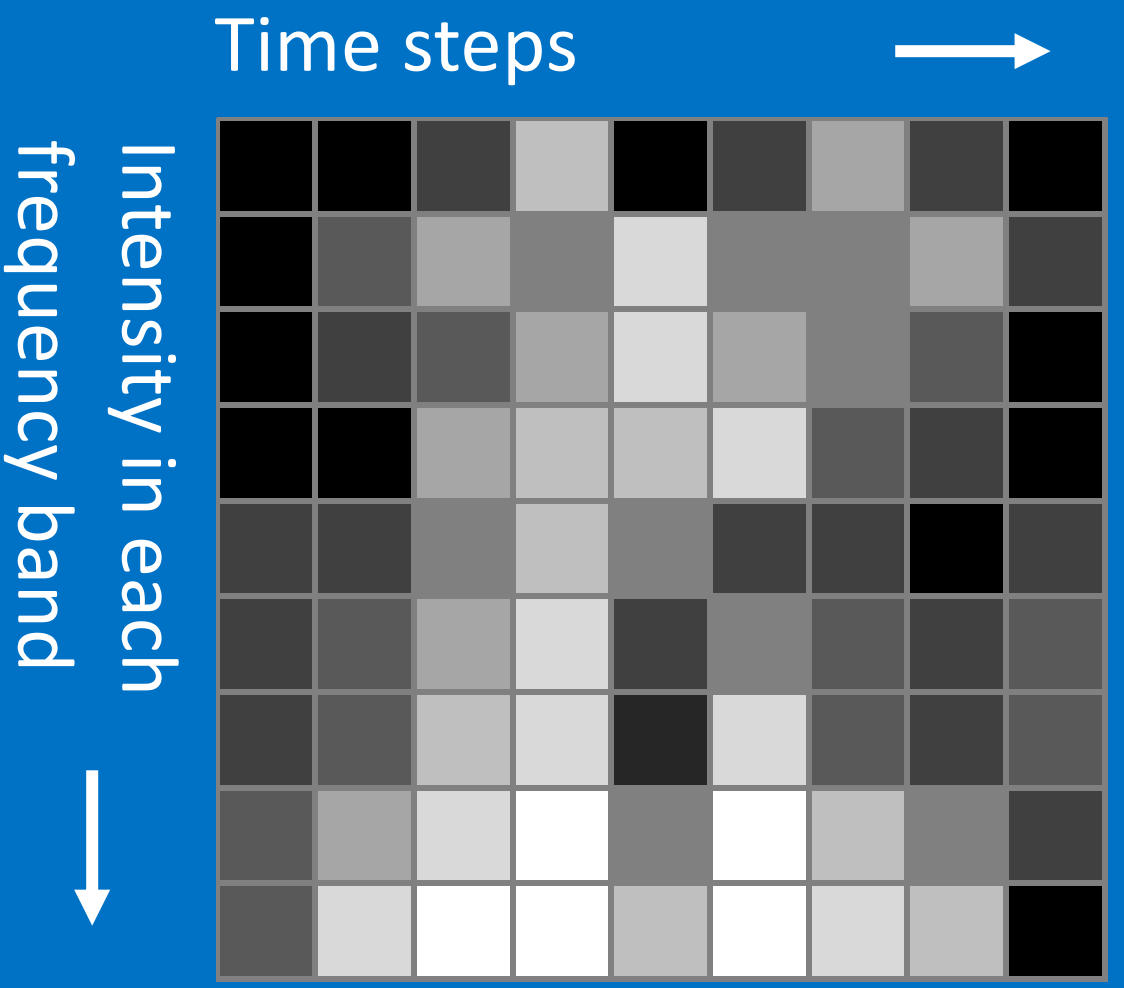
Any 2D (or 3D) data.

Things closer together are more closely related than things far away.

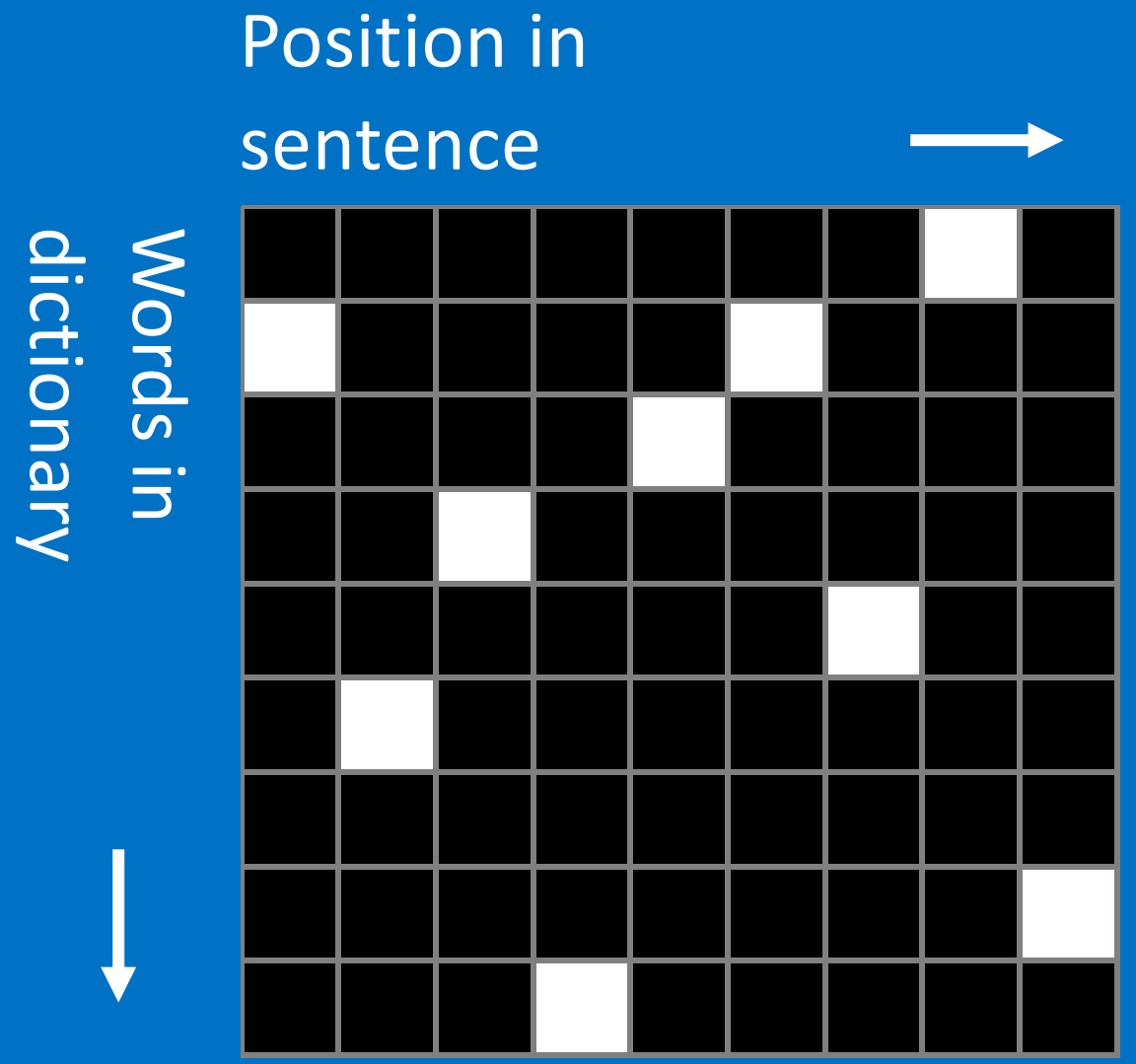
Images



Sound



Text



Limitations

ConvNets only capture local “spatial” patterns in data.

If the data can’t be made to look like an image, ConvNets are less useful.

Customer data

Name, age,
address, email,
purchases,
browsing activity,…

Customers

A	22	1A	a@a	1	aa	a1.a	123	aa1
B	33	2B	b@b	2	bb	b2.b	234	bb2
C	44	3C	c@c	3	cc	c3.c	345	cc3
D	55	4D	d@d	4	dd	d4.d	456	dd4
E	66	5E	e@e	5	ee	e5.e	567	ee5
F	77	6F	f@f	6	ff	f6.f	678	ff6
G	88	7G	g@g	7	gg	g7.g	789	gg7
H	99	8H	h@h	8	hh	h8.h	890	hh8
I	111	9I	i@i	9	ii	i9.i	901	ii9

Rule of thumb

If your data is just as useful after swapping any of your columns with each other, then you can’t use Convolutional Neural Networks.

In a nutshell

ConvNets are great at finding patterns and using them to classify images.