Understanding Neural Networks

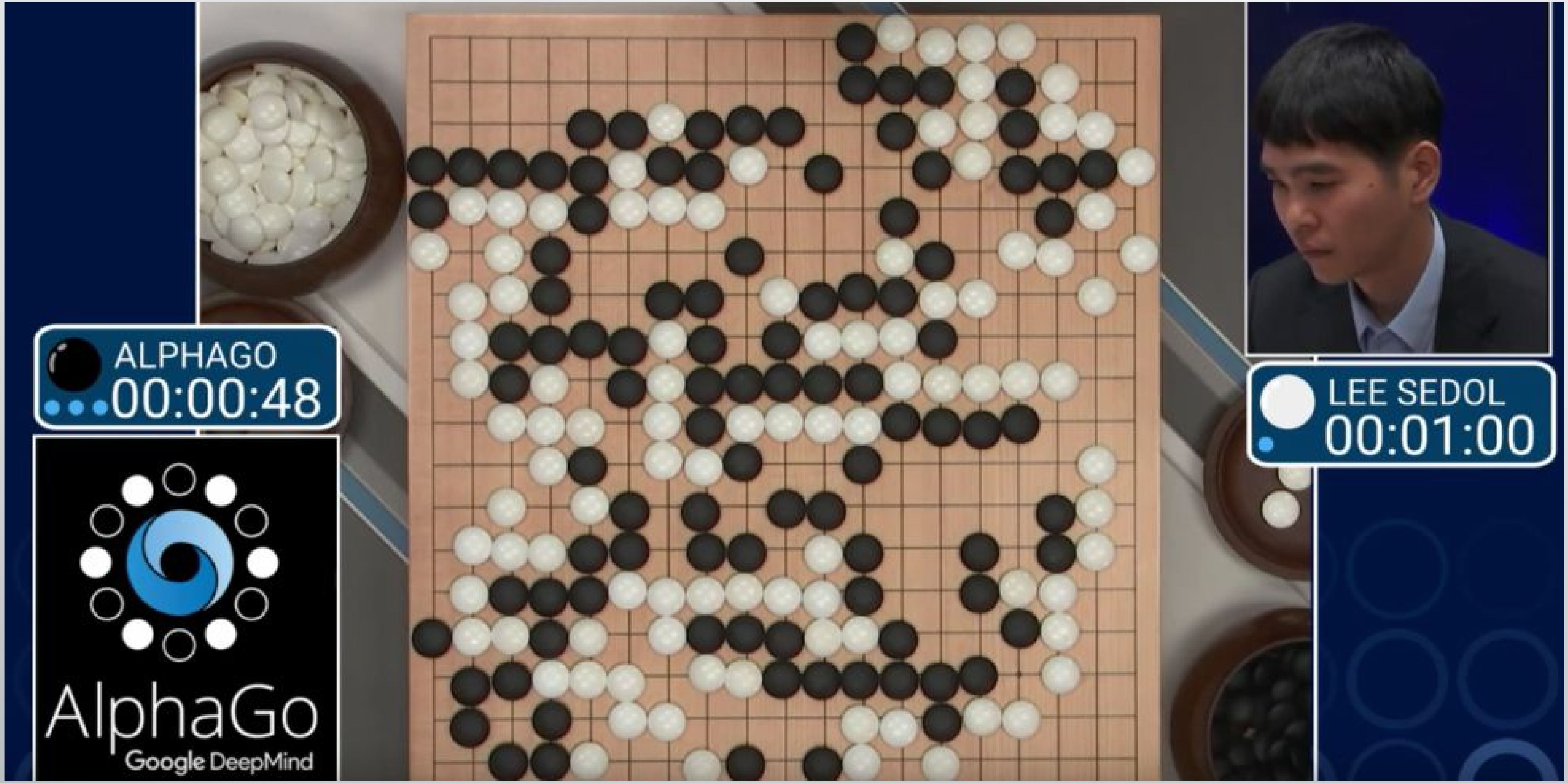
Giles Strong

15/03/17

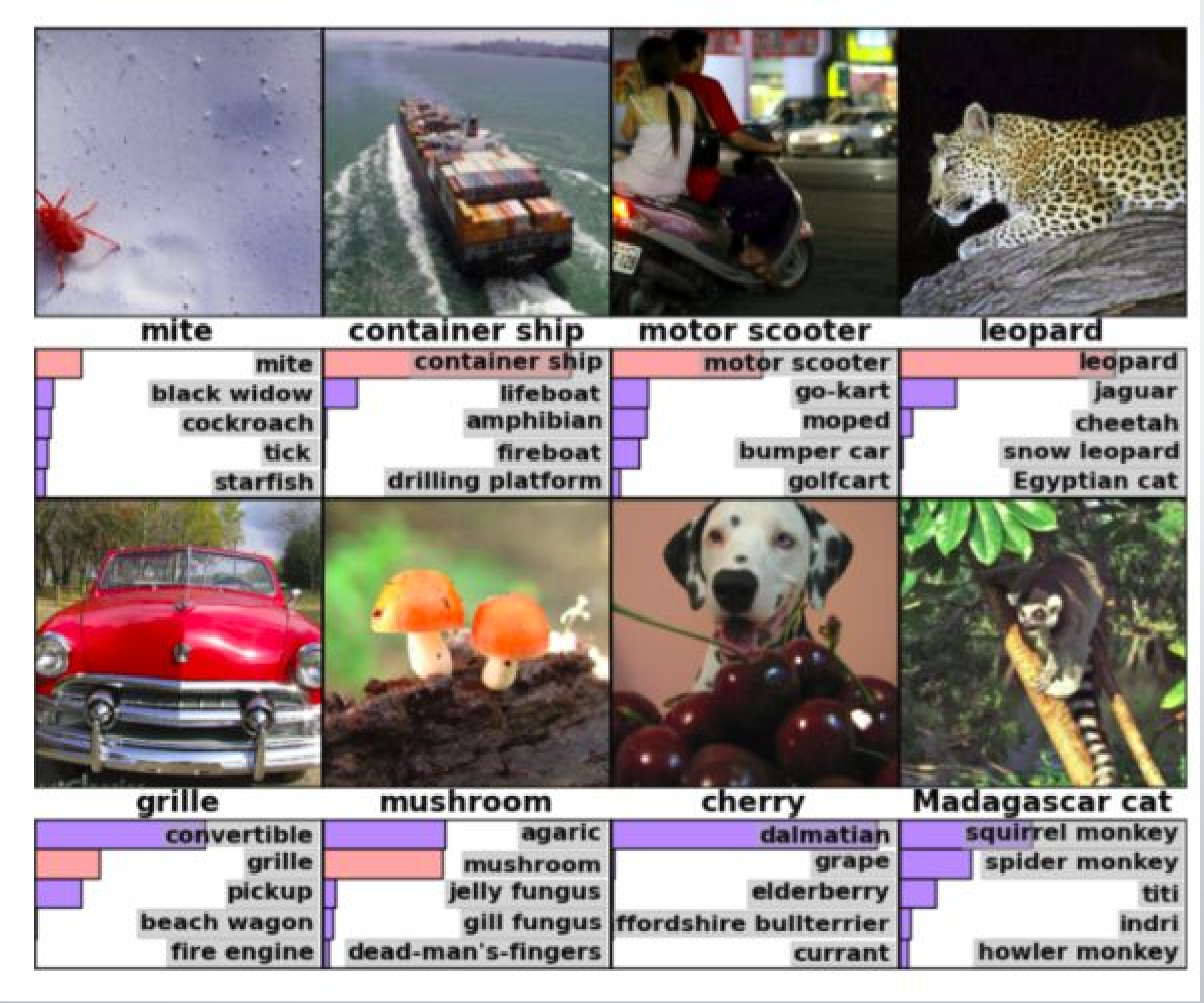
AEMPP I Evaluation Seminar

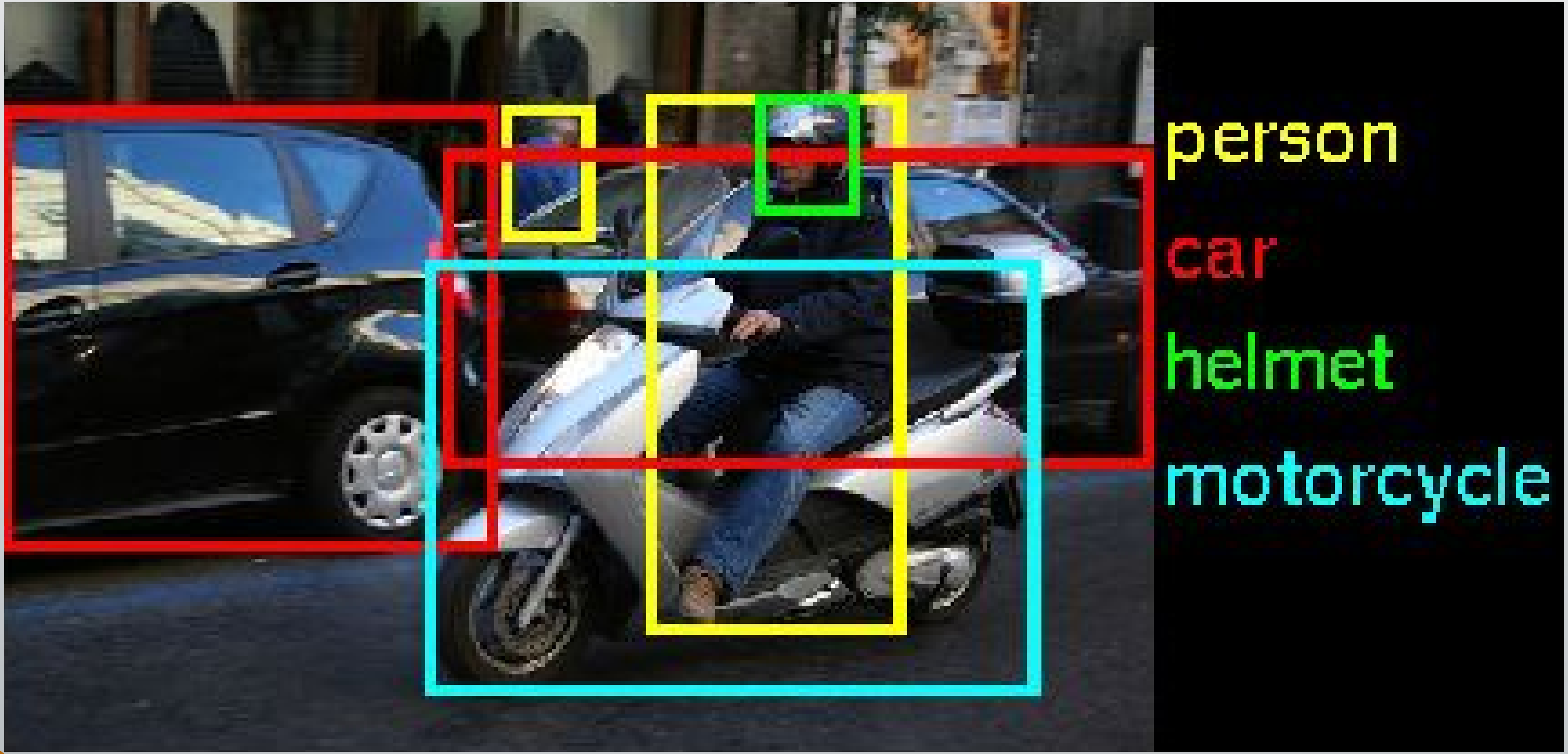
Seminar Questions

- What are artificial neural networks?
- How do they work?
- How can we improve them?
- Why use them in the first place?



ALPHAGO
00:00:48
LEE SEDOL
00:01:00
AlphaGo
Google DeepMind


mite
mite
black widow
cockroach
tick
starfish
container ship
container ship
lifeboat
amphibian
fireboat
drilling platform
motor scooter
motor scooter
go-kart
moped
bumper car
golfcart
leopard
leopard
jaguar
cheetah
snow leopard
Egyptian cat
grille
convertible
grille
pickup
beach wagon
fire engine
mushroom
agaric
mushroom
jelly fungus
gill fungus
dead-man's-fingers
cherry
dalmatian
grape
elderberry
ffordshire bullterrier
currant
Madagascar cat
squirrel monkey
spider monkey
titi
indri
howler monkey

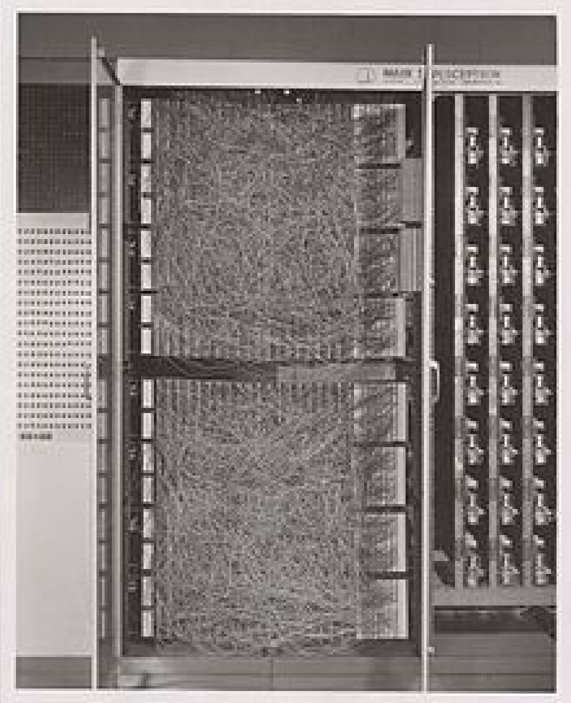

person
car
helmet
motorcycle

Mark I Perceptron – Rosenblatt, 1957

- First machine to run the *single-layer perceptron* algorithm

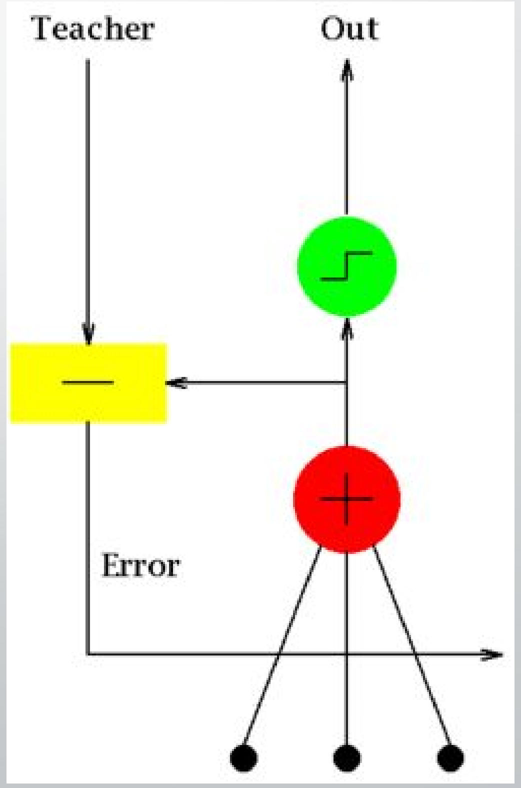
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Weights (*w*) set using potentiometers
- Used for image recognition, but didn't live up to expectations; couldn't learn properly



ADALINE and MADALINE - Widrow and Hoff 1960

- (Multi-layer) perceptron machine
- Still hardware-based
- Used a slightly more advanced algorithm to learn the correct weights
- Still failed to perform as well as expected



Back propagation – 1960-1986

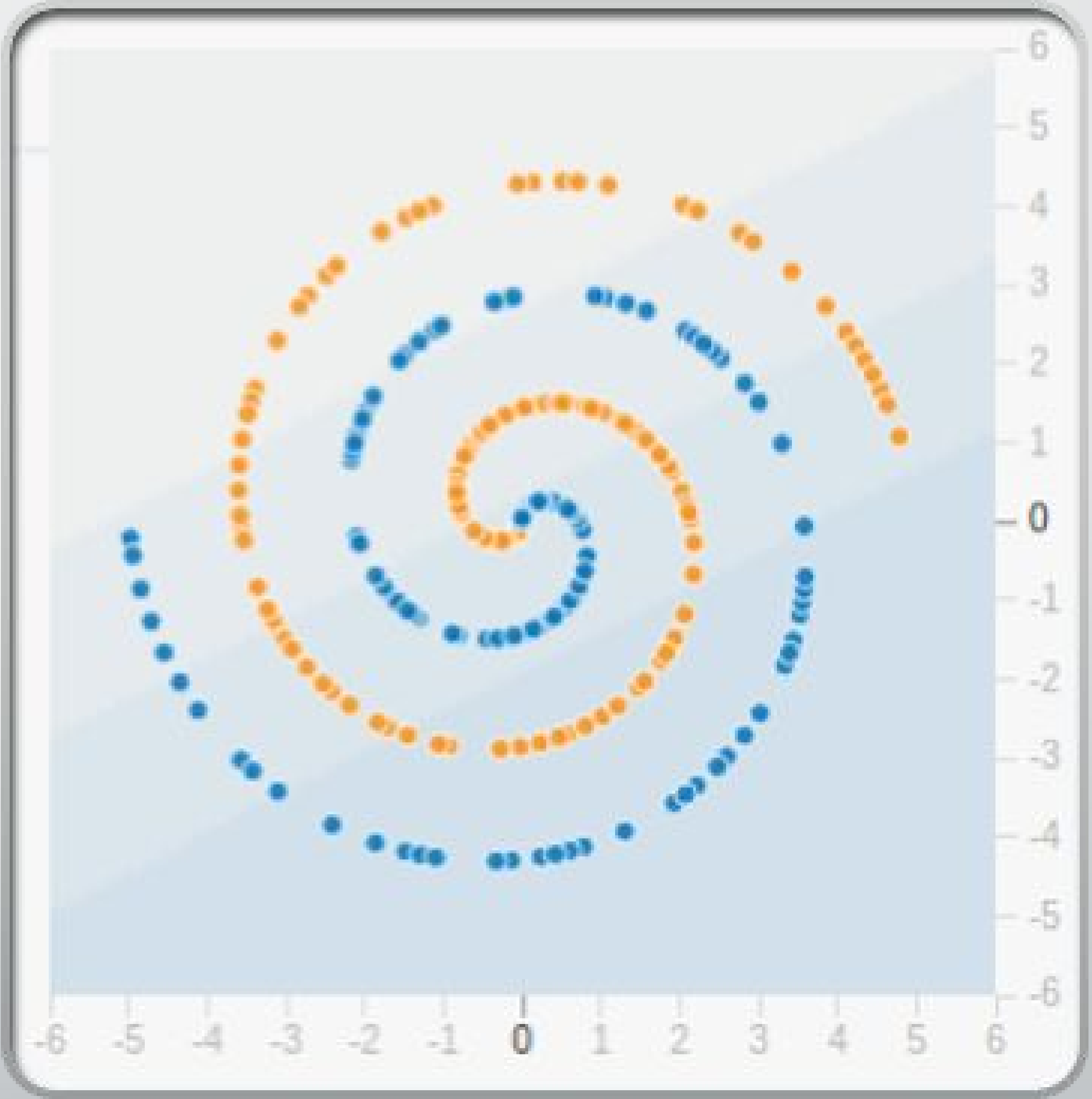
- Weight-learning based on chain-rule differentiation
- Basics, Keely 1960 and Bryson 1962
- First applied to ANNs in 1982 by Werbos
- Shown to be useful in multi-layer ANNs by Rumelhart, Hinton, and Williams in 1986
- However, ANNs still underperformed, and were limited in size; training would get stuck
- Interest in ANNs diminishes

Neural Network Renaissance - 2006

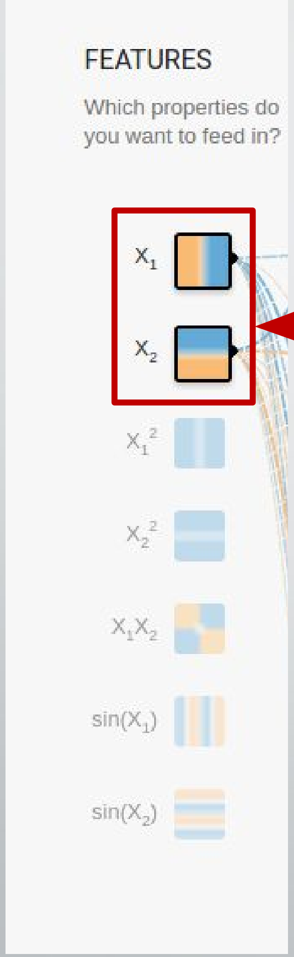
- Hinton and Salakhutdinov develop a layer-by-layer pre-training method
- Allowed backpropagation to work for deep neural-networks
- In 2010 deep neural-networks begin outperforming other methods in speech recognition [Acero, Dahl, Deng, and Yu, 2010]
- Reinvigorated research in NNs

Example

- Say we want to predict the class (orange or blue) of points according to their position
- We want to draw decision boundaries in our feature space



Overview of a neural network



2 input features:
X and Y coordinates

Overview of a neural network

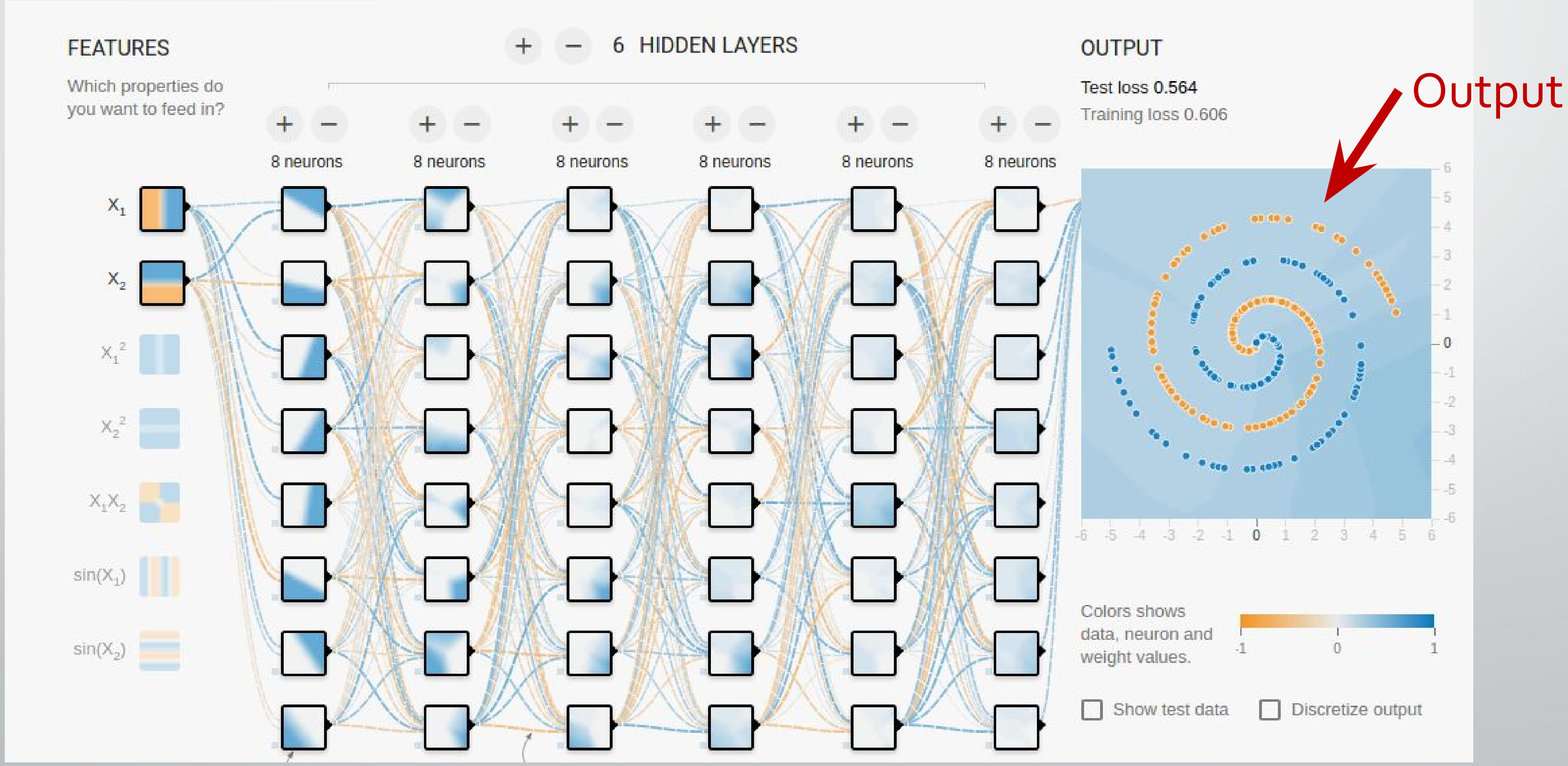
Neuron – applies mathematical transformation

Layer of neurons

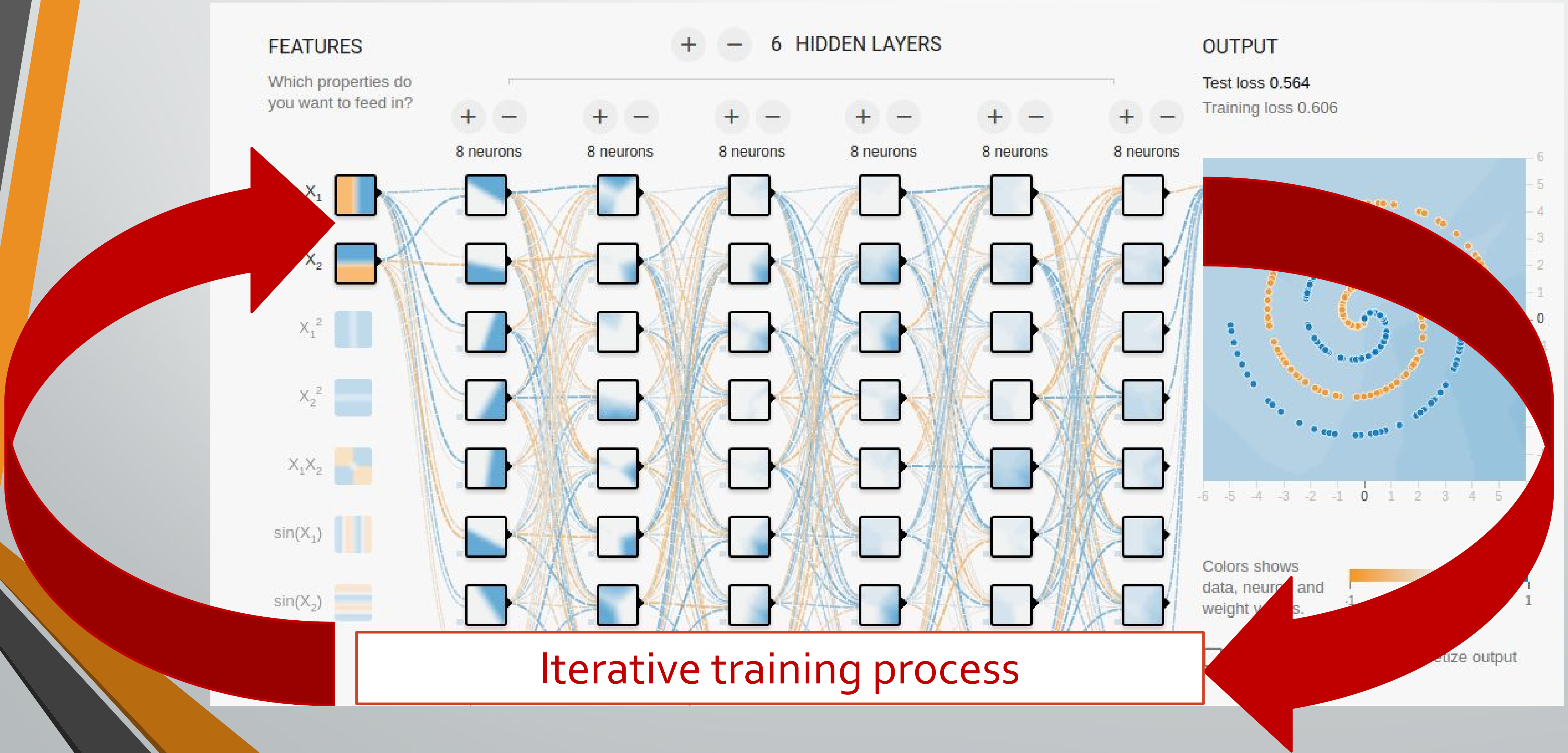
Overview of a neural network

Add additional layers

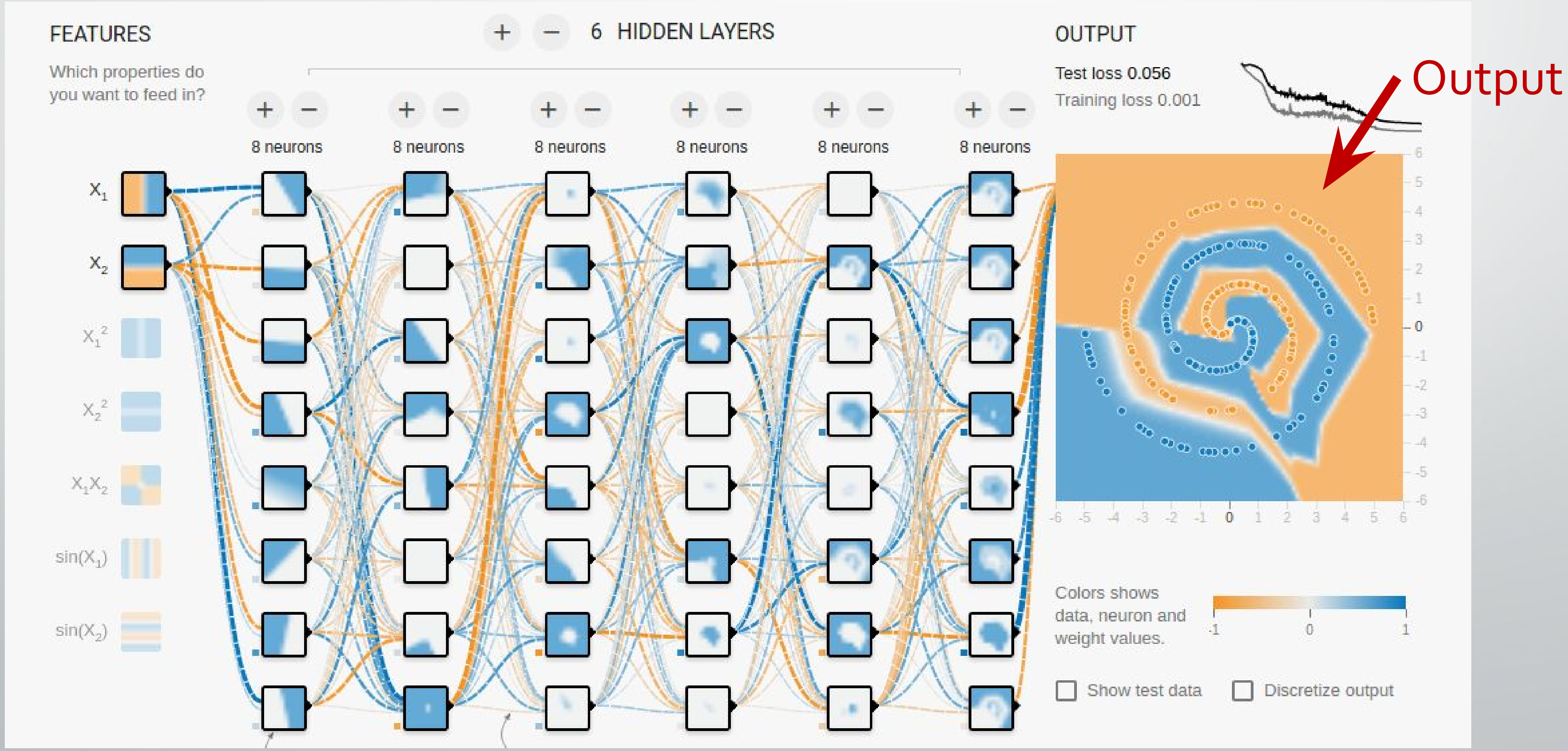
Overview of a neural network



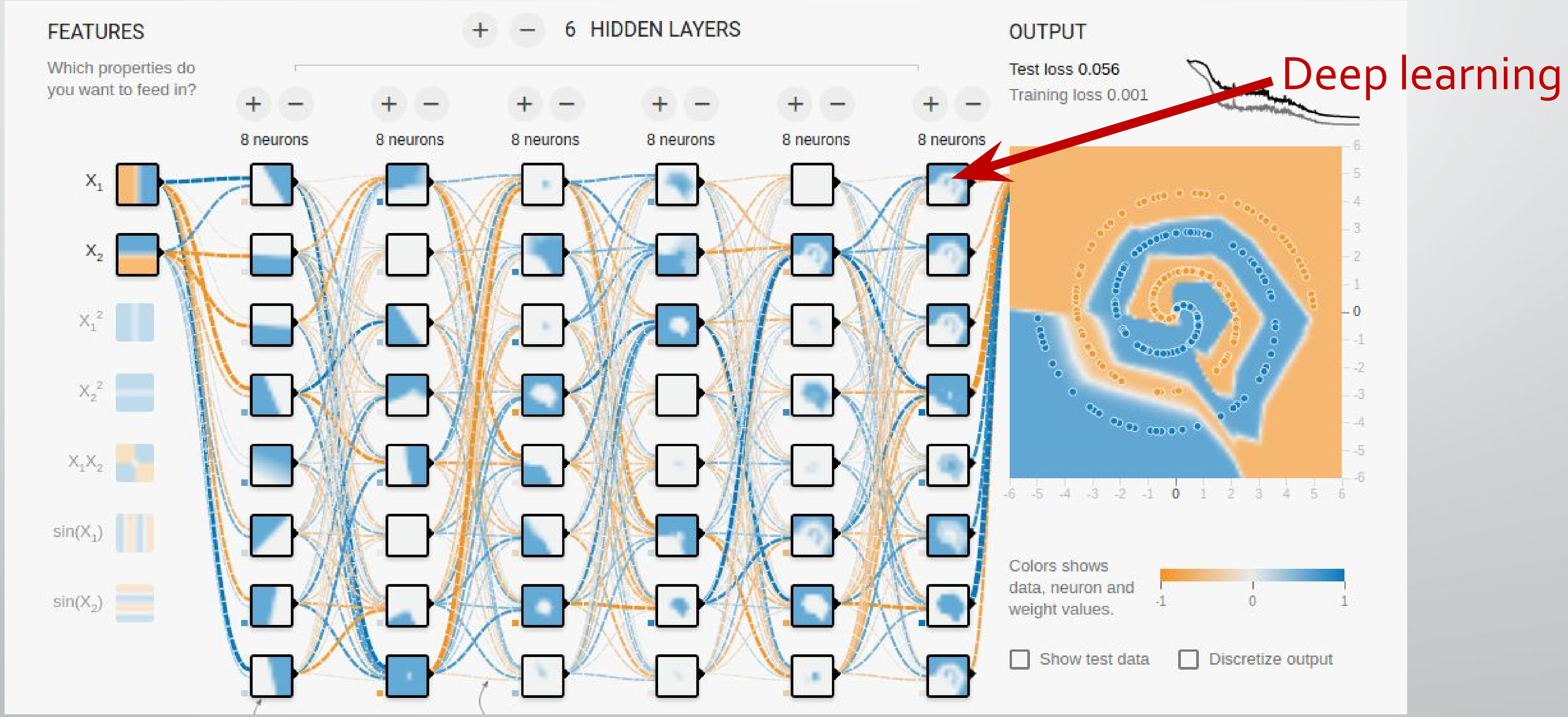
Overview of a neural network



Overview of a neural network



Overview of a neural network



Main components of a neural network

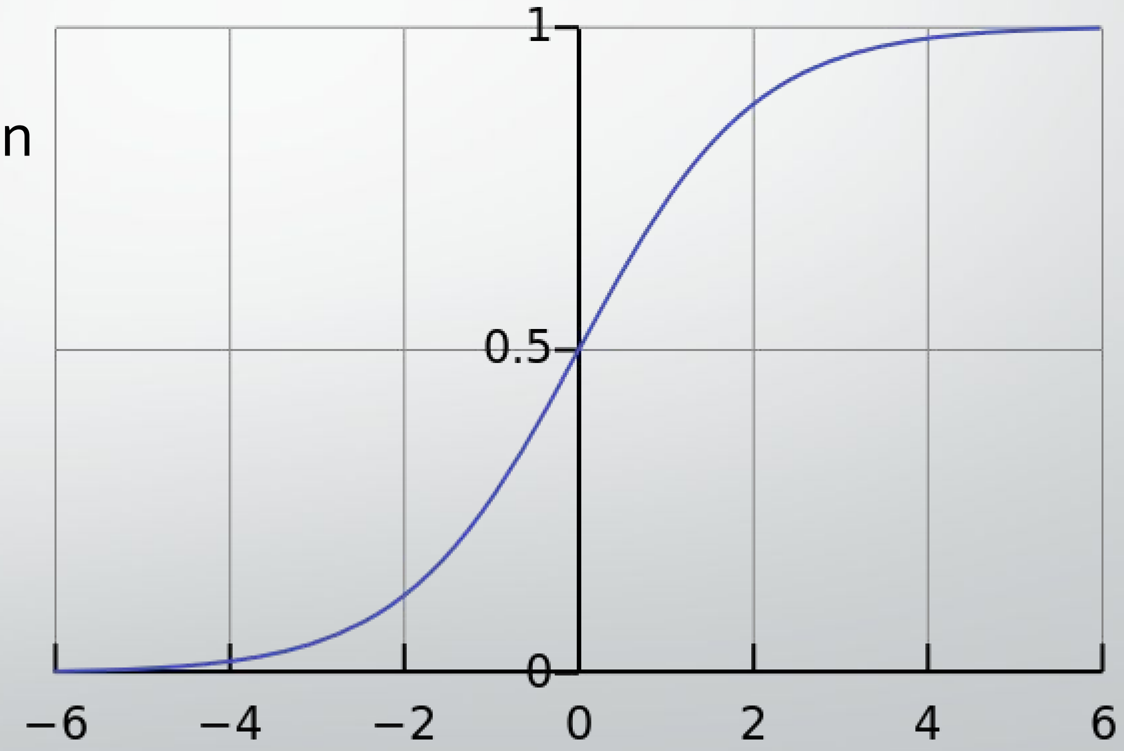
1. Neurons
2. The network
3. Training

What is a neuron?

- Quite simply, it is a mathematical transformation:
- It takes vector of inputs $\underline{x}$
- Weights each input element
- Applies an *activation function*, e.g sigmoid: $$f = \frac{1}{1 + e^{-\sum_i w_i x_i}}$$
- And passes its output forwards in the network

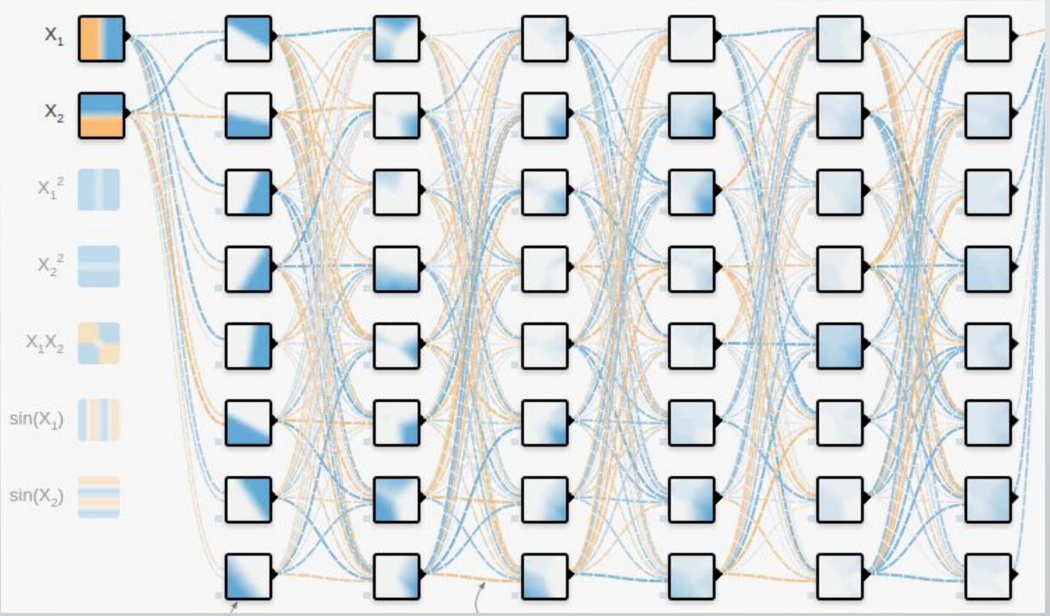
What is a neuron?

- The function applied by the neuron can be any continuous mathematical function of the inputs
- However there are several 'standard' ones which are used
- Sigmoid was a common choice



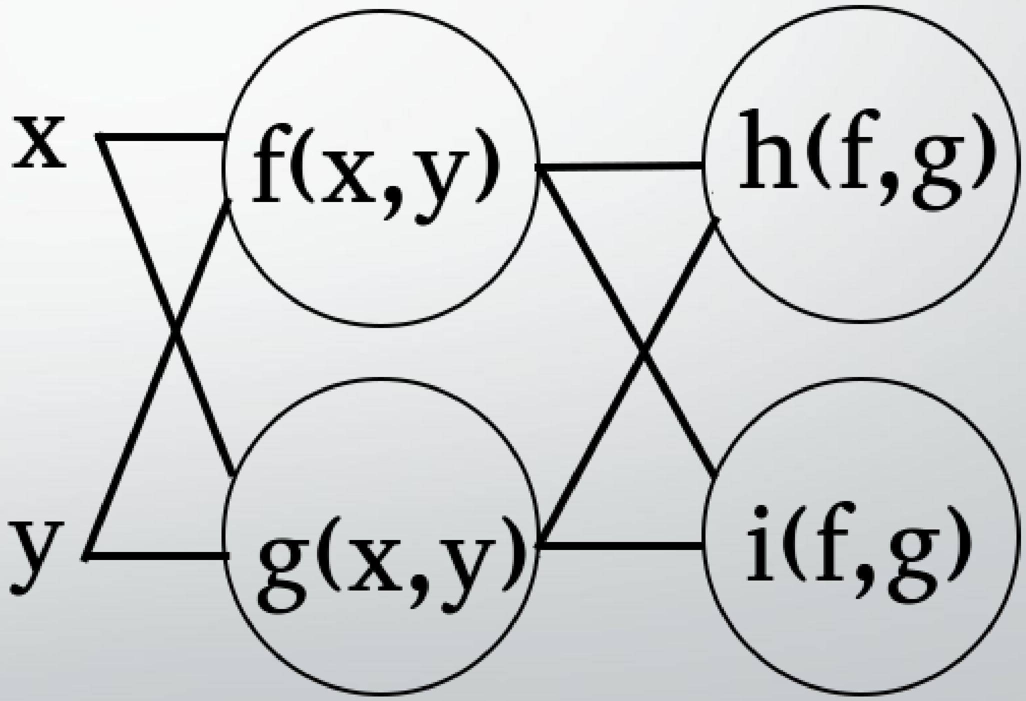
Constructing a network

- As seen earlier, a network is simply many layers of neurons



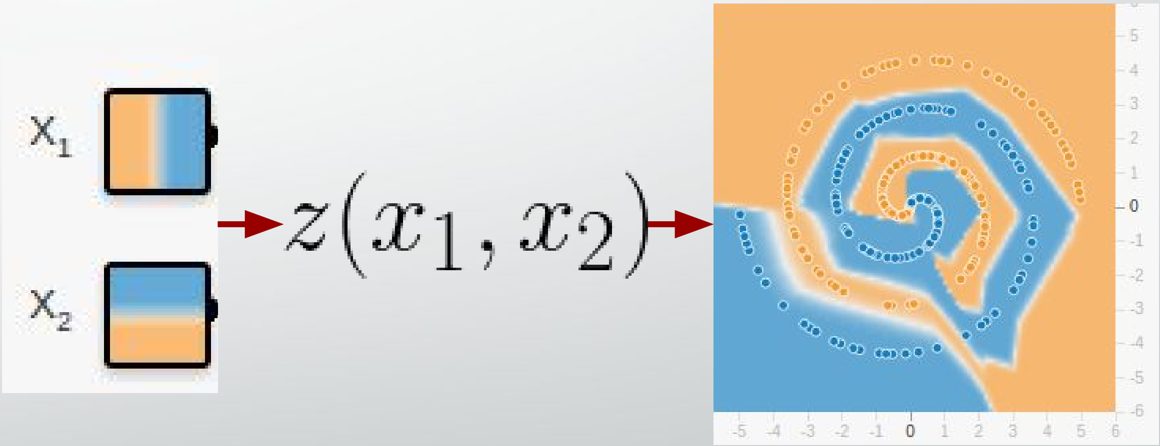
Constructing a network

- A single neuron applies a basic function to the inputs
- By connecting layers of neurons together, more complex functions can be constructed



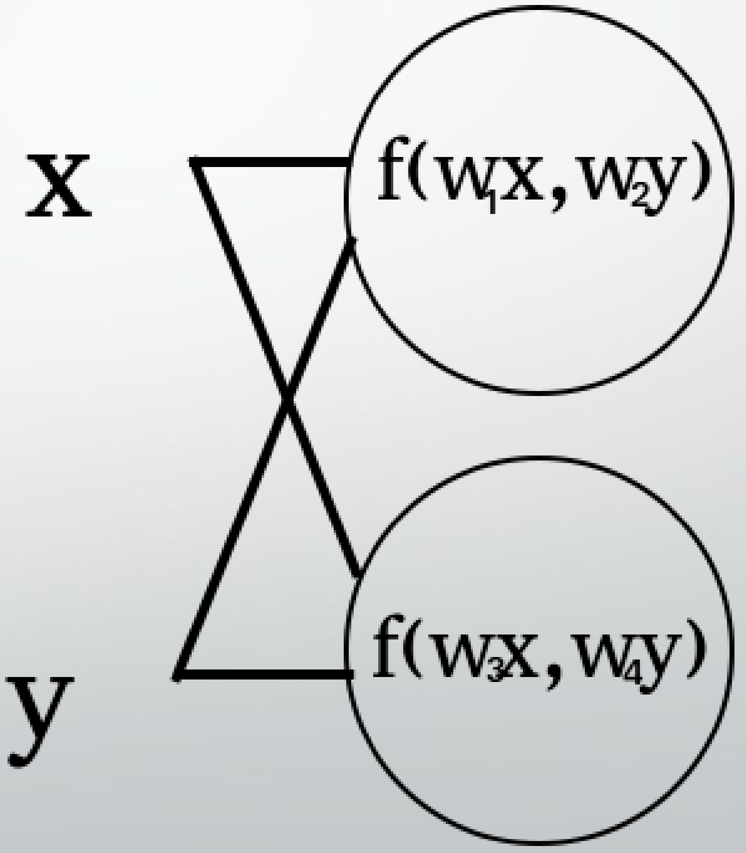
Constructing a network

- The aim is to learn a function which maps the inputs to the desired outputs



Constructing a network

- Each neuron applies the same basic function
- But the weights each neuron applies can be different
- ∴ create the map by altering the weights



Towards training

- How do we alter the weights?
- Could test random settings, but unlikely to arrive at good settings for anything but tiny networks
- Need to alter the weights intelligently, i.e. train the network
- To do this, we need to quantify the performance of the network

Quantifying performance - Loss

- This measure of performance is called a loss function
- It quantises the difference between the network's prediction for a data point, and the actual value of the data point

Quantifying performance - Loss

- One example is the mean squared-error:

Average over data points

Difference between prediction and truth

$$MSE = \boxed{\frac{1}{N}\sum_{n=1}^{N}} \boxed{(\hat{y_n} - y_n)}^2$$

Quantifying performance - Loss

- For classification, the cross-entropy is better:

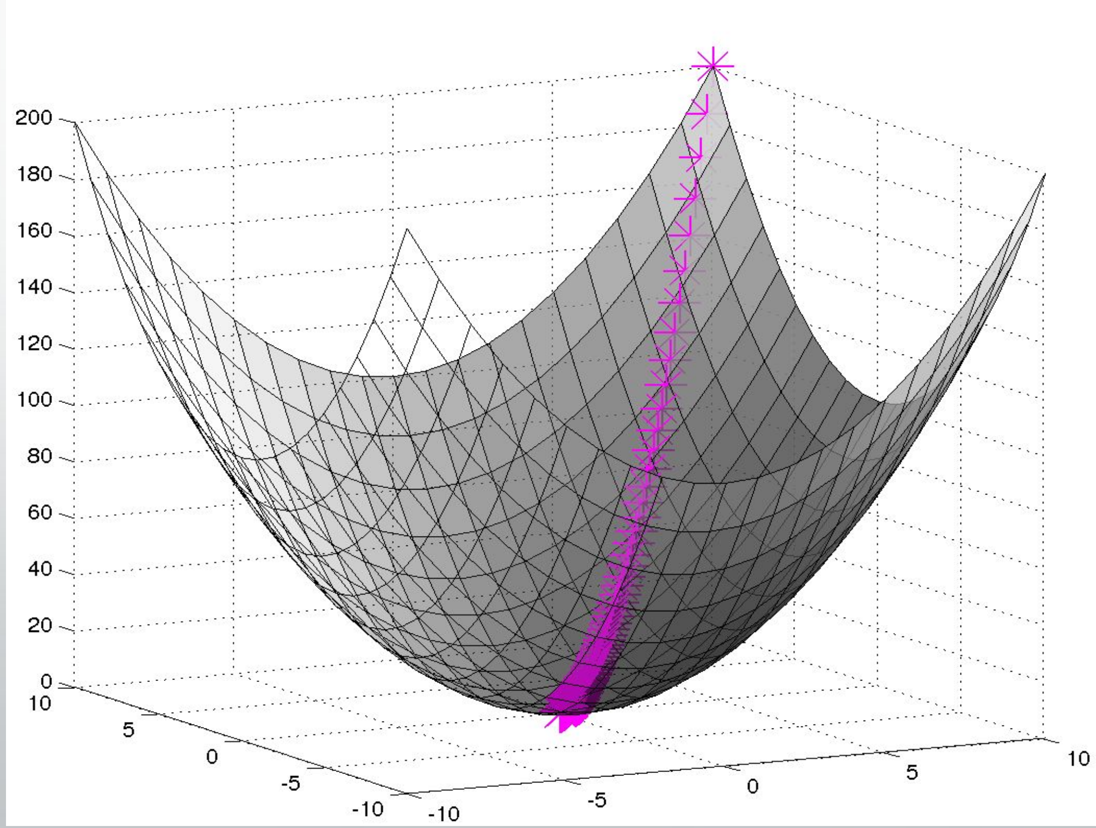
$$CE = -\underbrace{\frac{1}{N}\sum_{n=1}^{N}}_{\text{Average over data points}}\Big[\underbrace{y_n \log \hat{y_n} + (1 - y_n) \log (1 - \hat{y_n})}_{\text{Difference between prediction and truth}}\Big]$$

Network optimisation

- Armed with a quantified measure of performance
- Our aim now is to minimise the loss function ⇒ an optimisation problem
- Lots of advanced algorithms exist: Genetic, Metropolis-Hastings, *et cetera*
- But the parameter space is huge! ⇒long convergence time

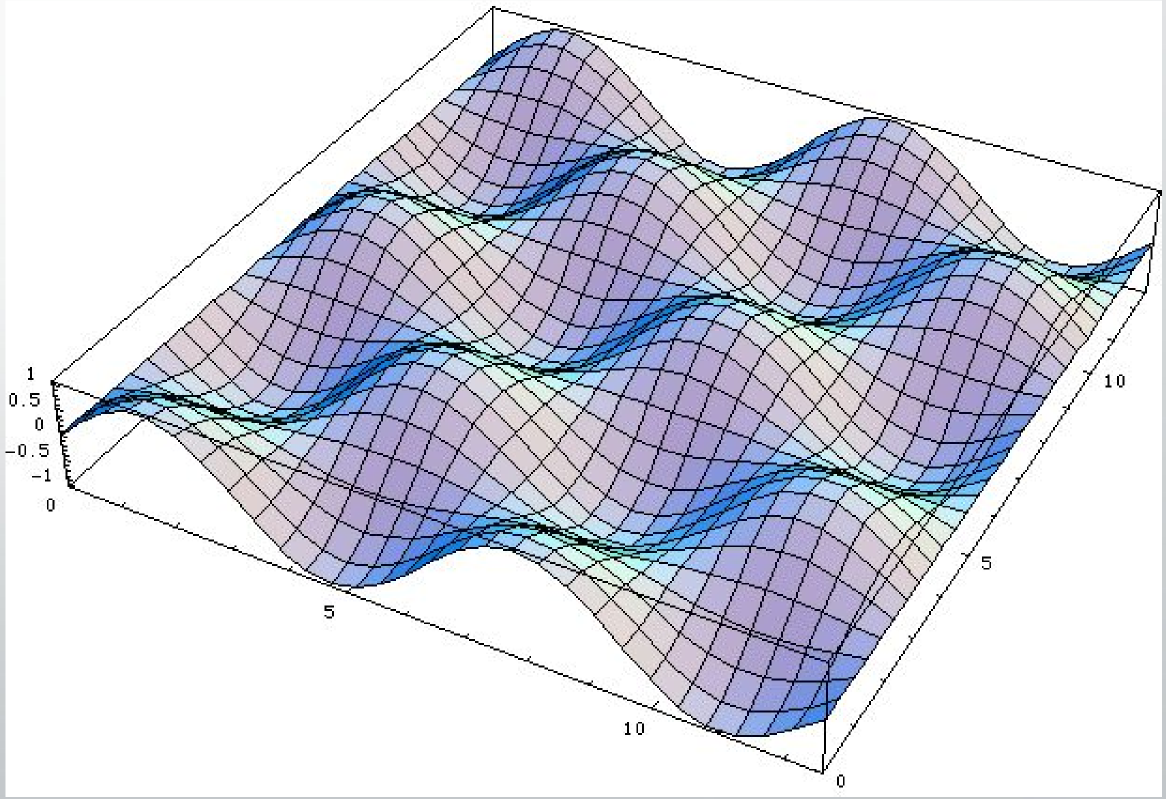
Network optimisation

- Turns out, the gradient descent algorithm works just fine



Network optimisation

- The loss function contains many local minima
- But each is about as optimal as the others
- We simply need to reach to bottom of a high-dimensional bowl
- We do this by moving down the gradient



Gradient evaluation - numerical method

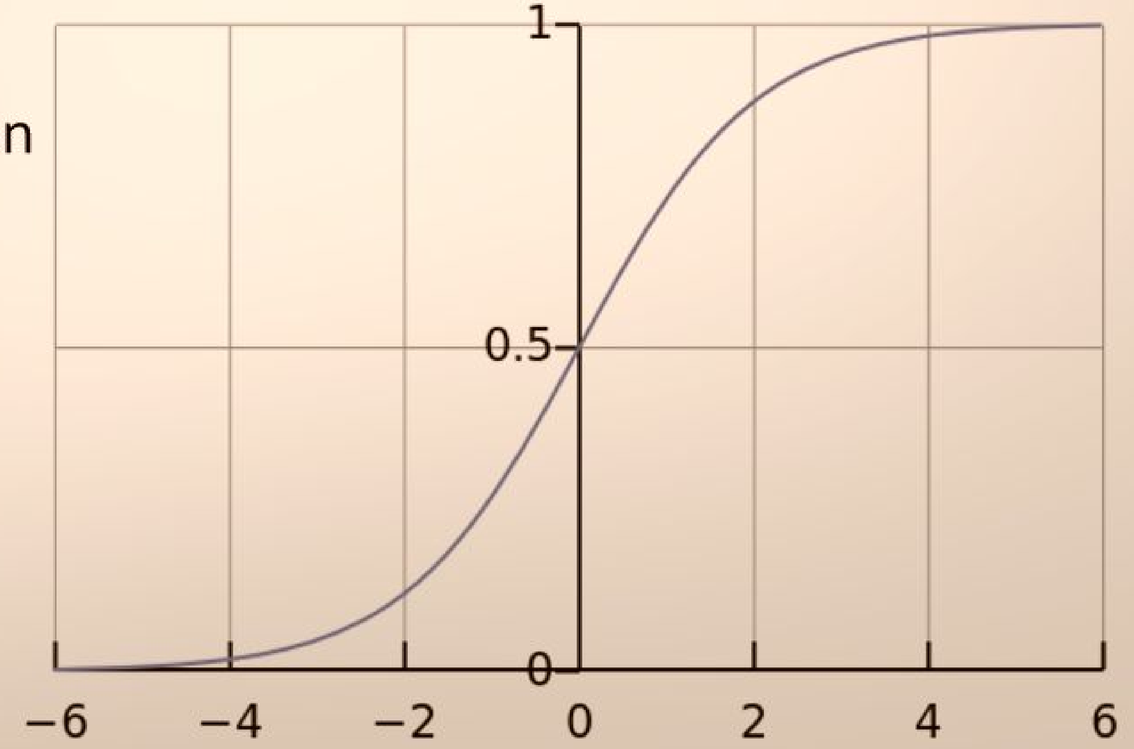
- In order to move down the slope, we first need to know the gradient of the loss function at a given point: $\nabla \mathcal{L}$
- This can be estimated numerically by varying each weight in the network by a small amount, *h*, and seeing how the output changes :

$$\frac{\partial f\left(x,y\right)}{\partial x} \approx \frac{f\left(x+h,y\right)-f\left(x,y\right)}{h}$$

- This works, but is time-consuming to compute: we can hundreds of thousands of weights to evaluate!

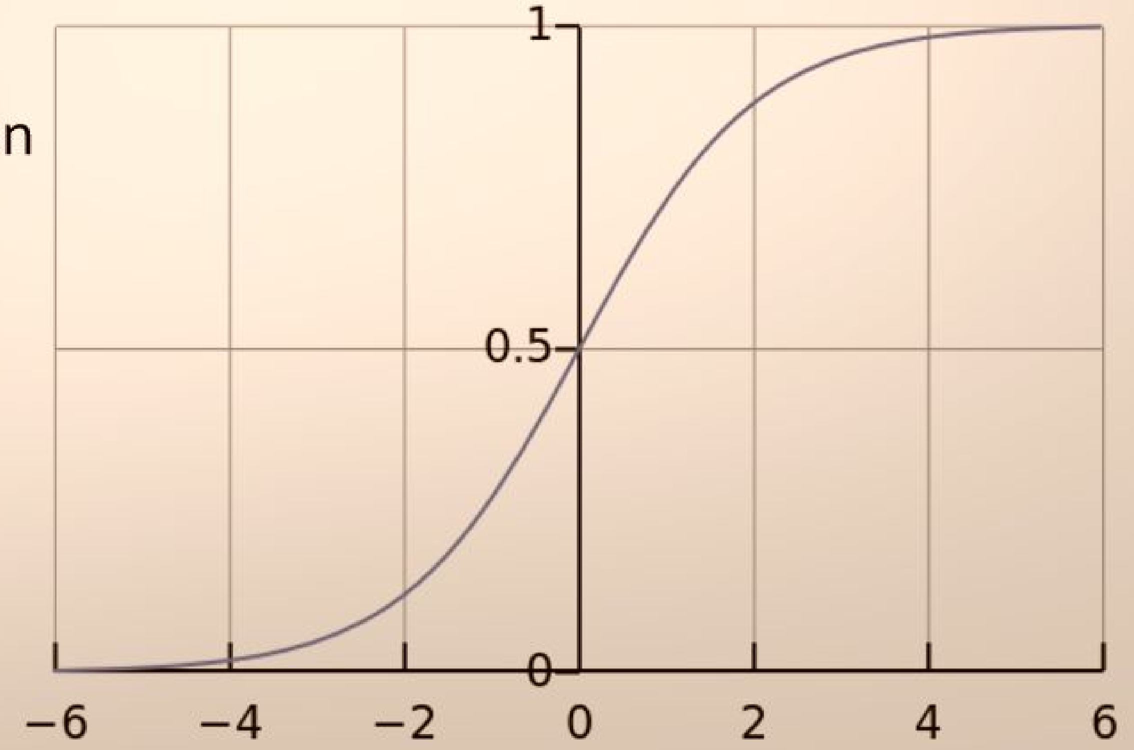
What is a neuron?

- The function applied by the neuron can be any continuous mathematical function of the inputs
- However there are several ‘standard’ ones which are used
- Sigmoid was a common choice



What is a neuron?

- The function applied by the neuron can be any continuous mathematical function of the inputs
- However there are several 'standard' ones which are used
- Sigmoid was a common choice



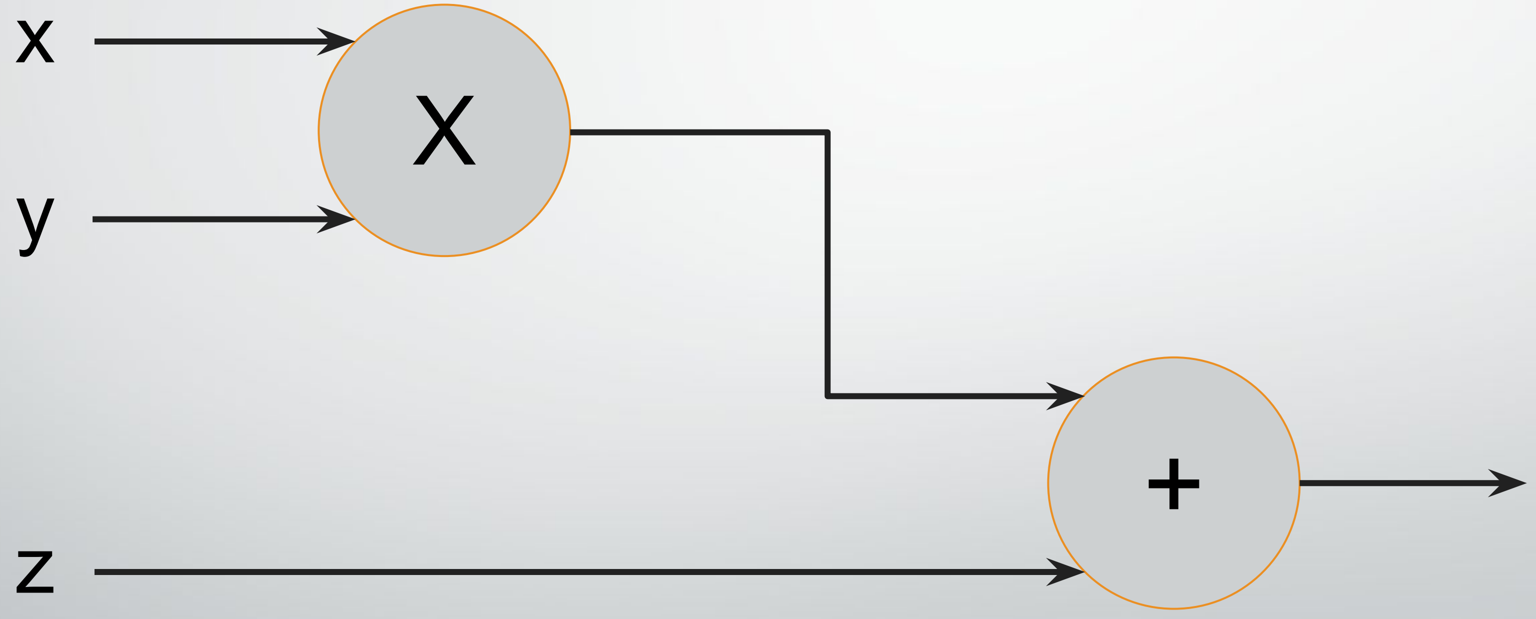
Gradient evaluation - Analytical method

- Because each neuron applies a continuous function, the entire network is differentiable
- We can compute the gradient analytically !

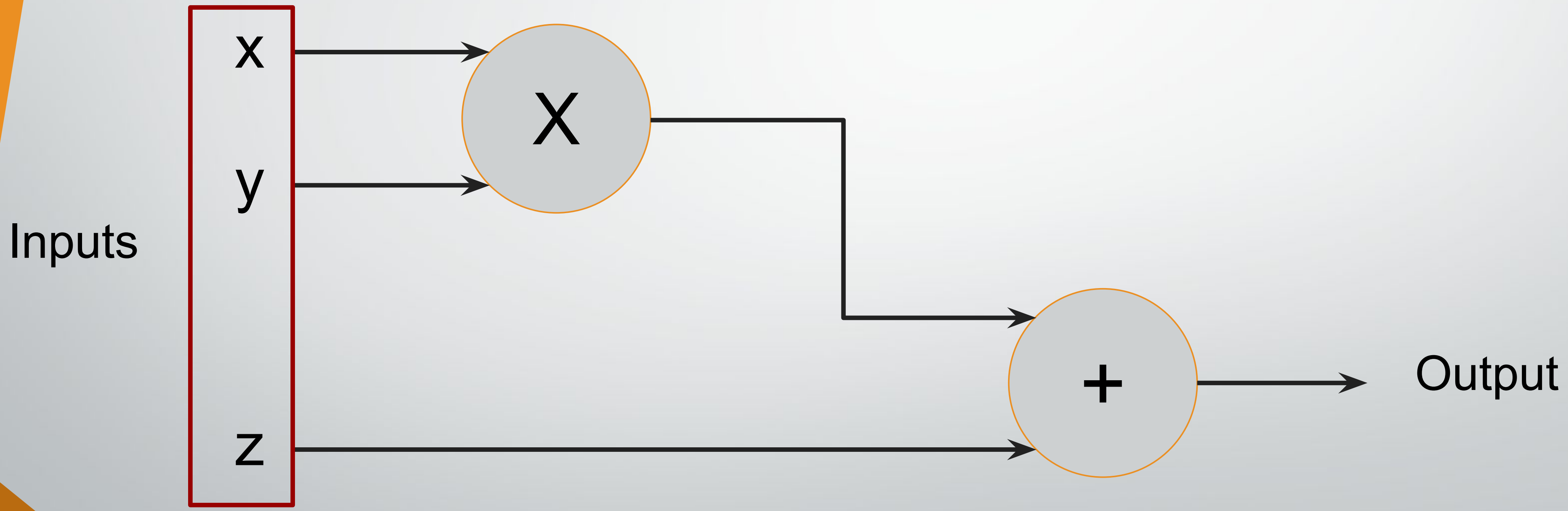
Enter back-propagation

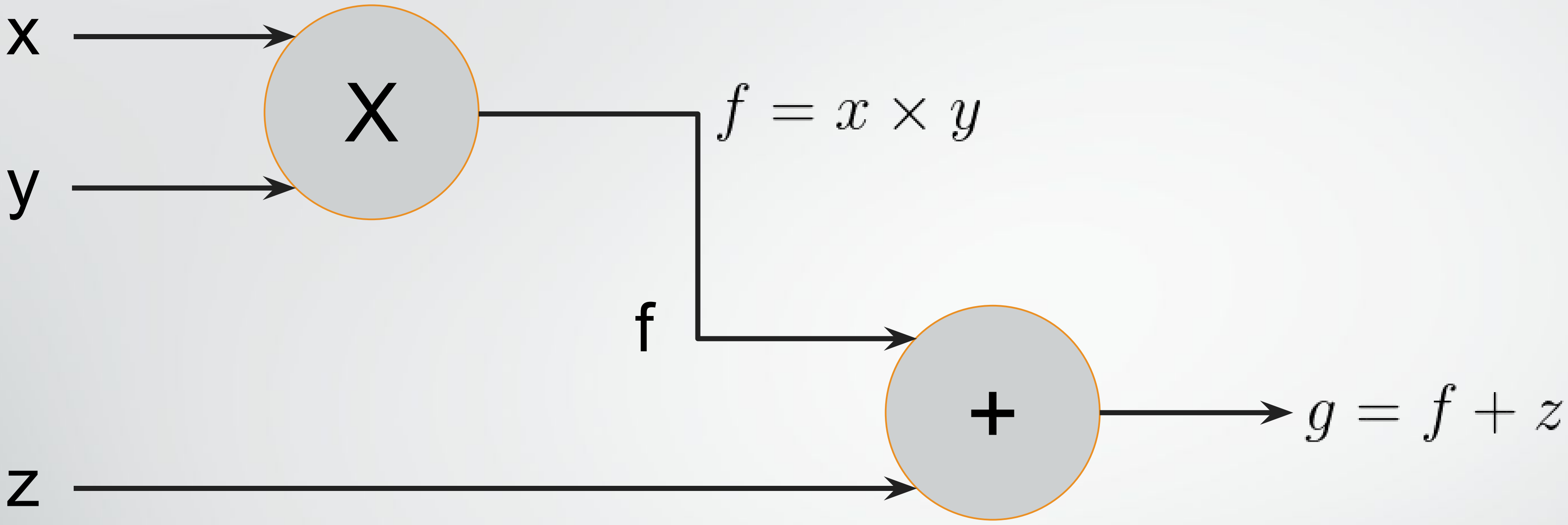
- Essentially, this method of analytical evaluation is a two-step process
- First we do a *forward pass* of a data point, to evaluate the loss
- Next we do a *backwards pass* through the network of the gradient of loss at that parameter point
- The neurons then know exactly how they effect the loss function and can be adjusted accordingly
- This is called *back-propagation*

Simple example

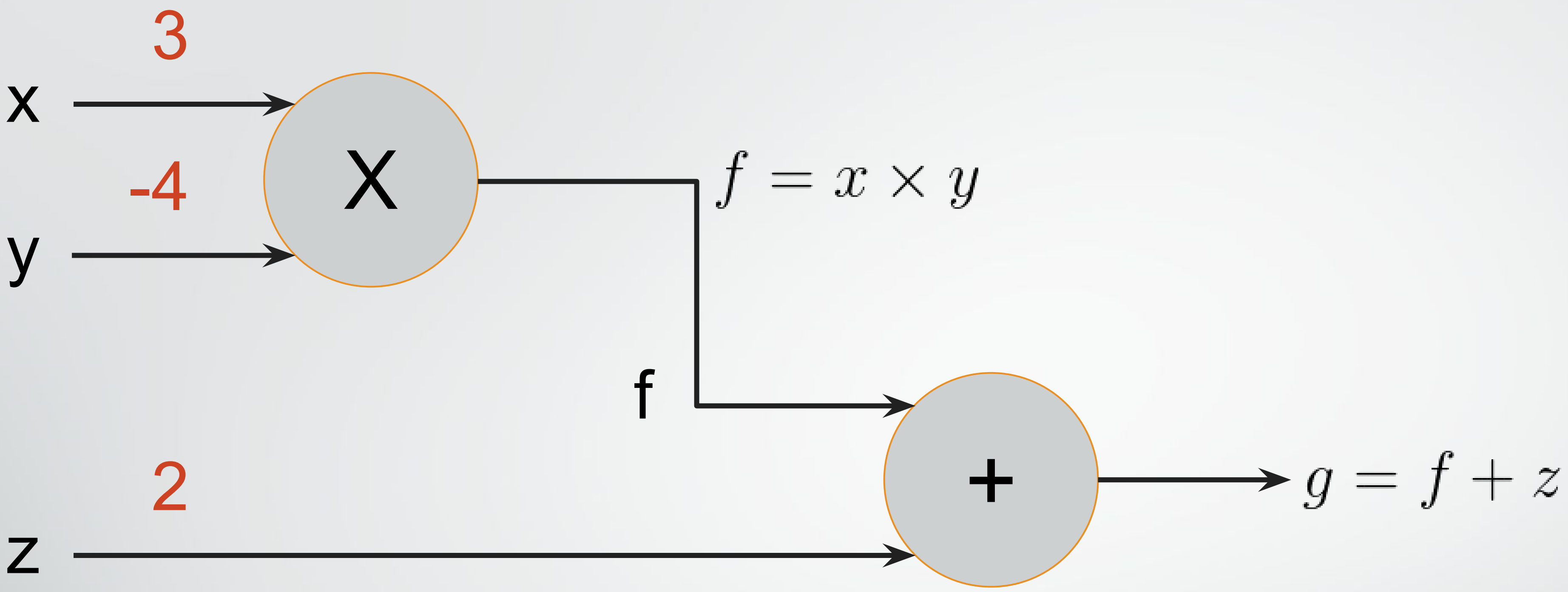


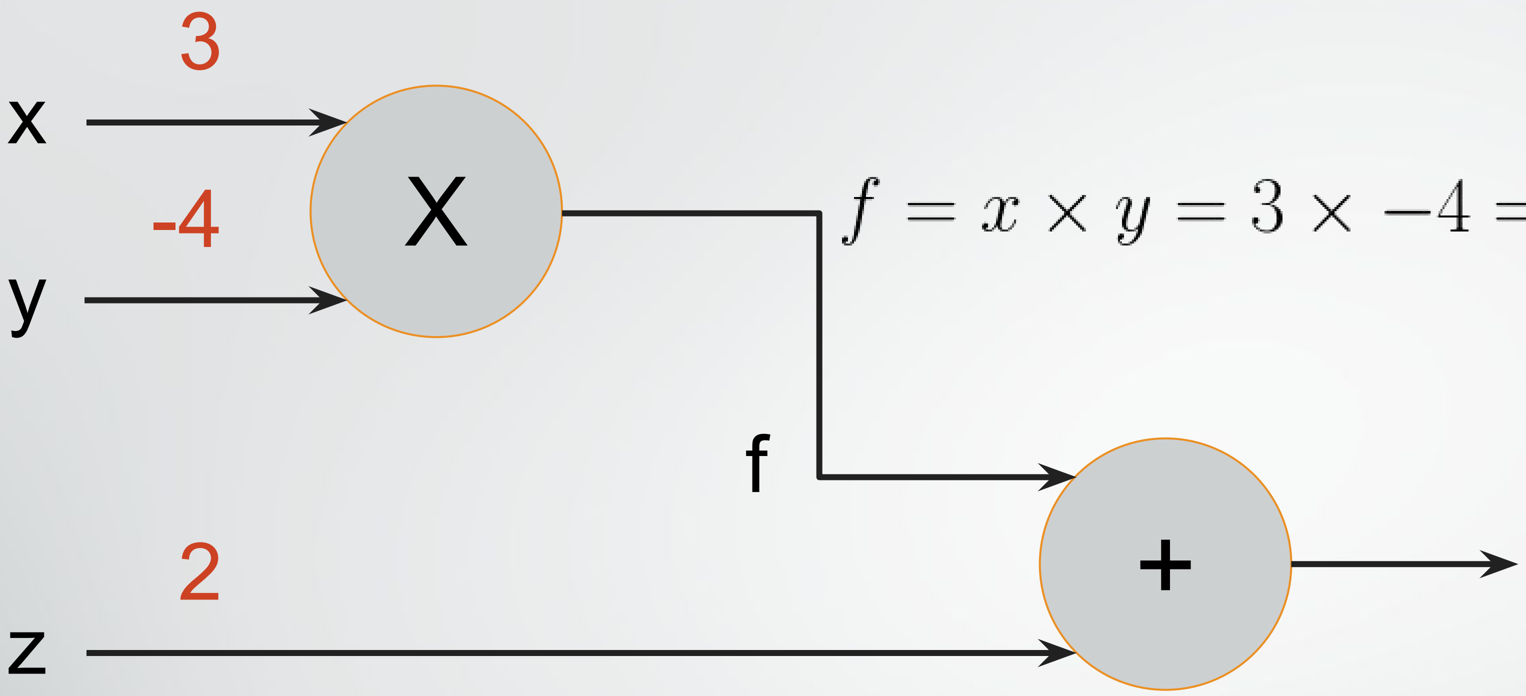
Simple example





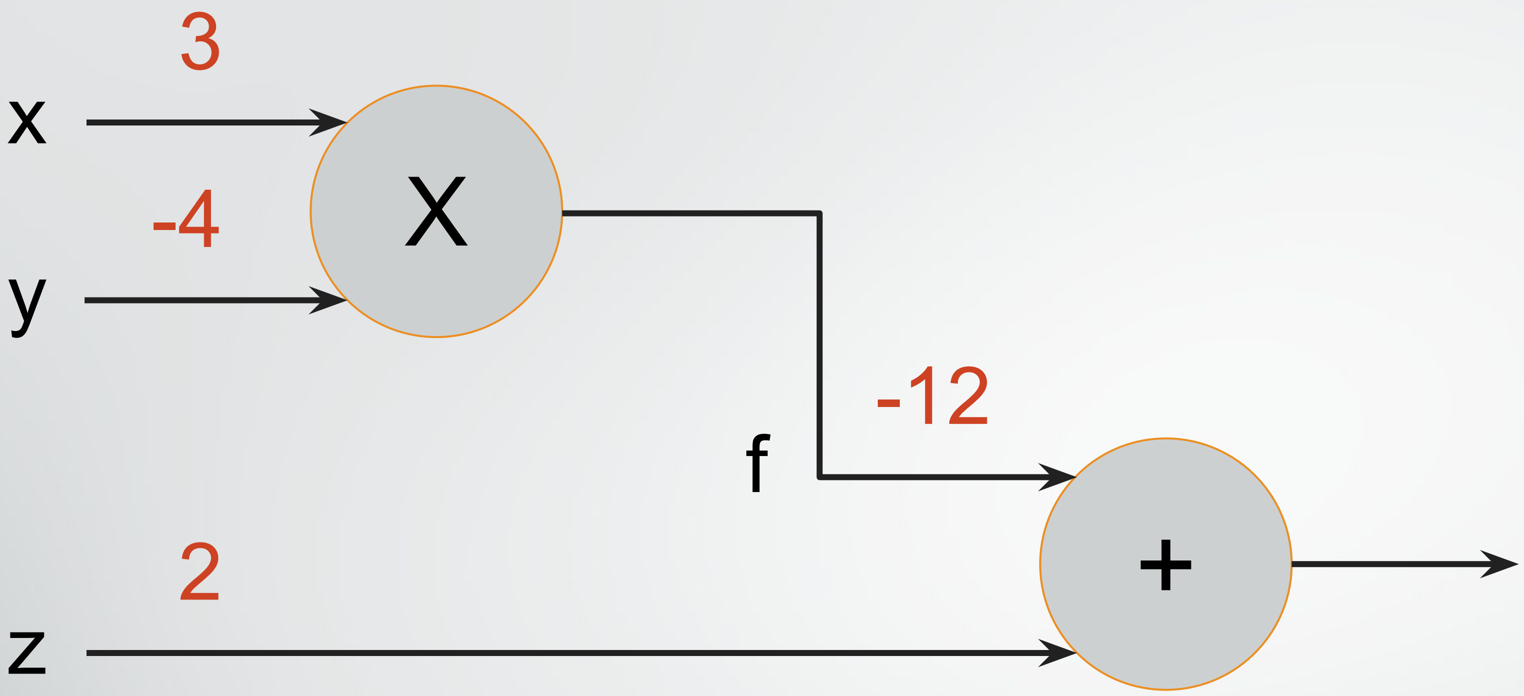
- Aim is to make decrease the value of *g(x,y,x)*
- Say we have an example data point: *x*=3, *y*=-4, *z*=2
- Let's do a forward pass through our network


3
x
-4
y
X
$f = x \times y$
f
2
z
+
$g = f + z$

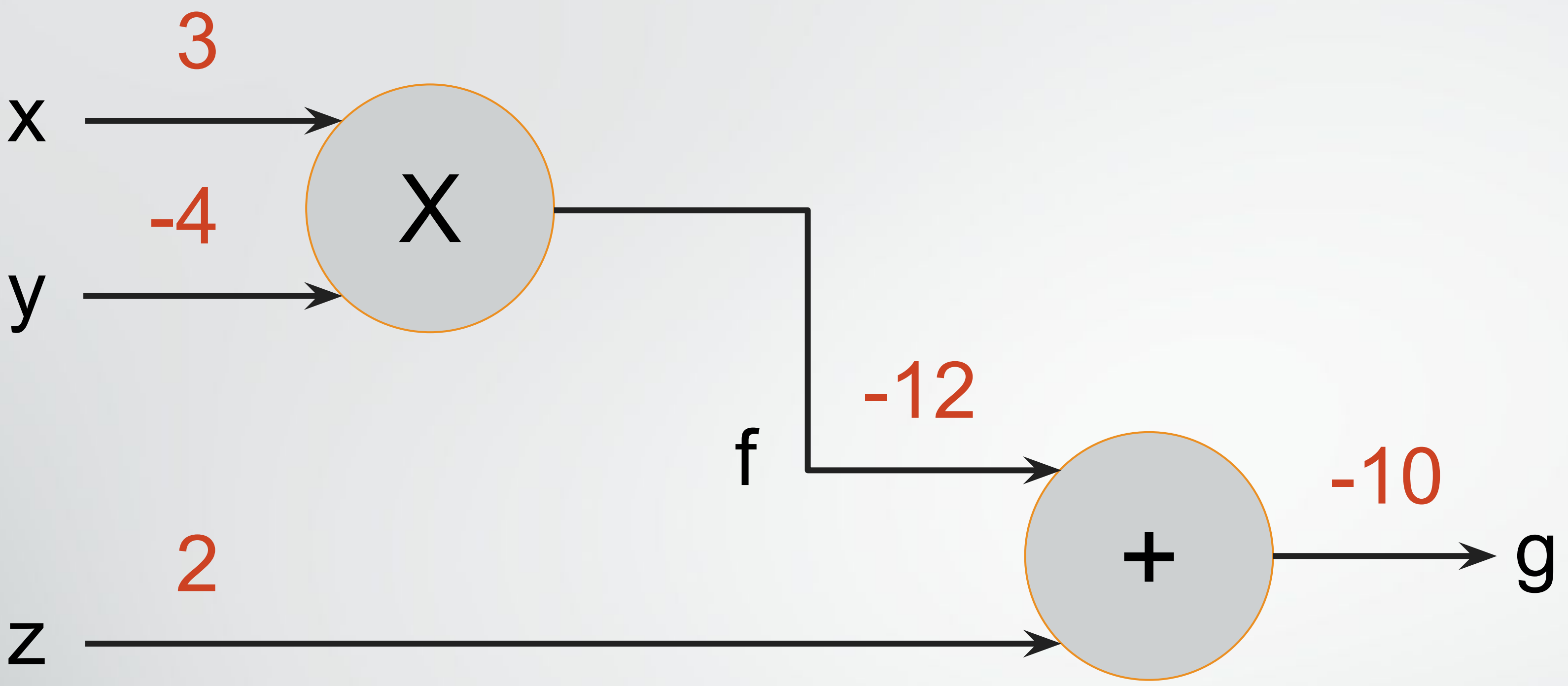


$$f = x \times y = 3 \times -4 = -12$$

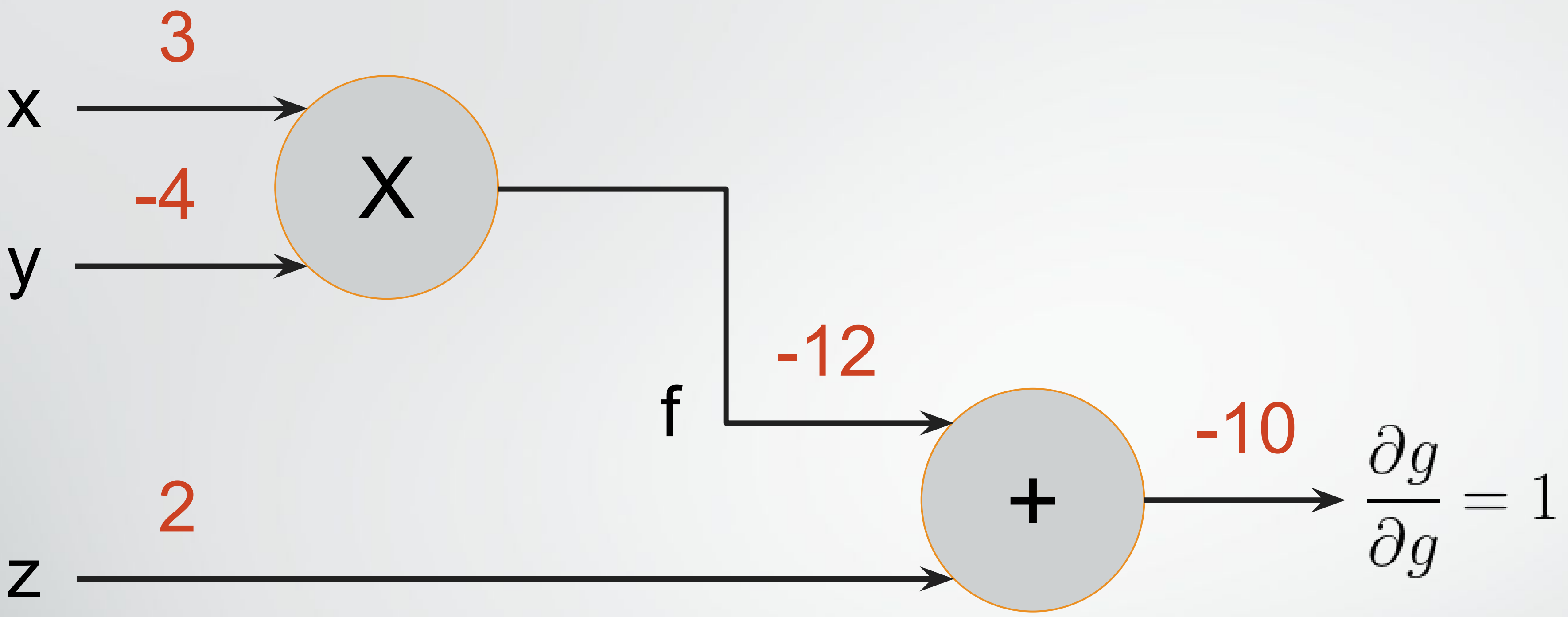
$$g = f + z$$



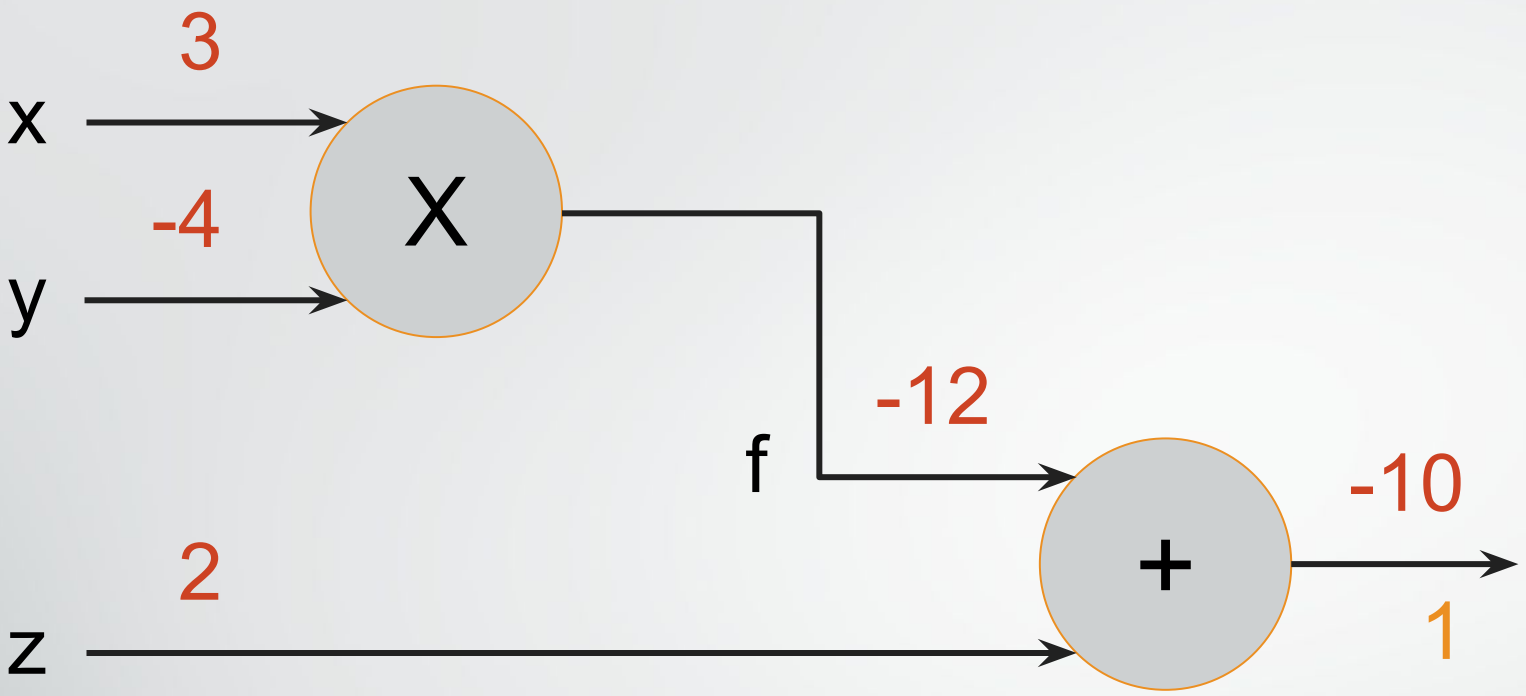
$$g = f + z$$
$$= -12 + 2 = -10$$



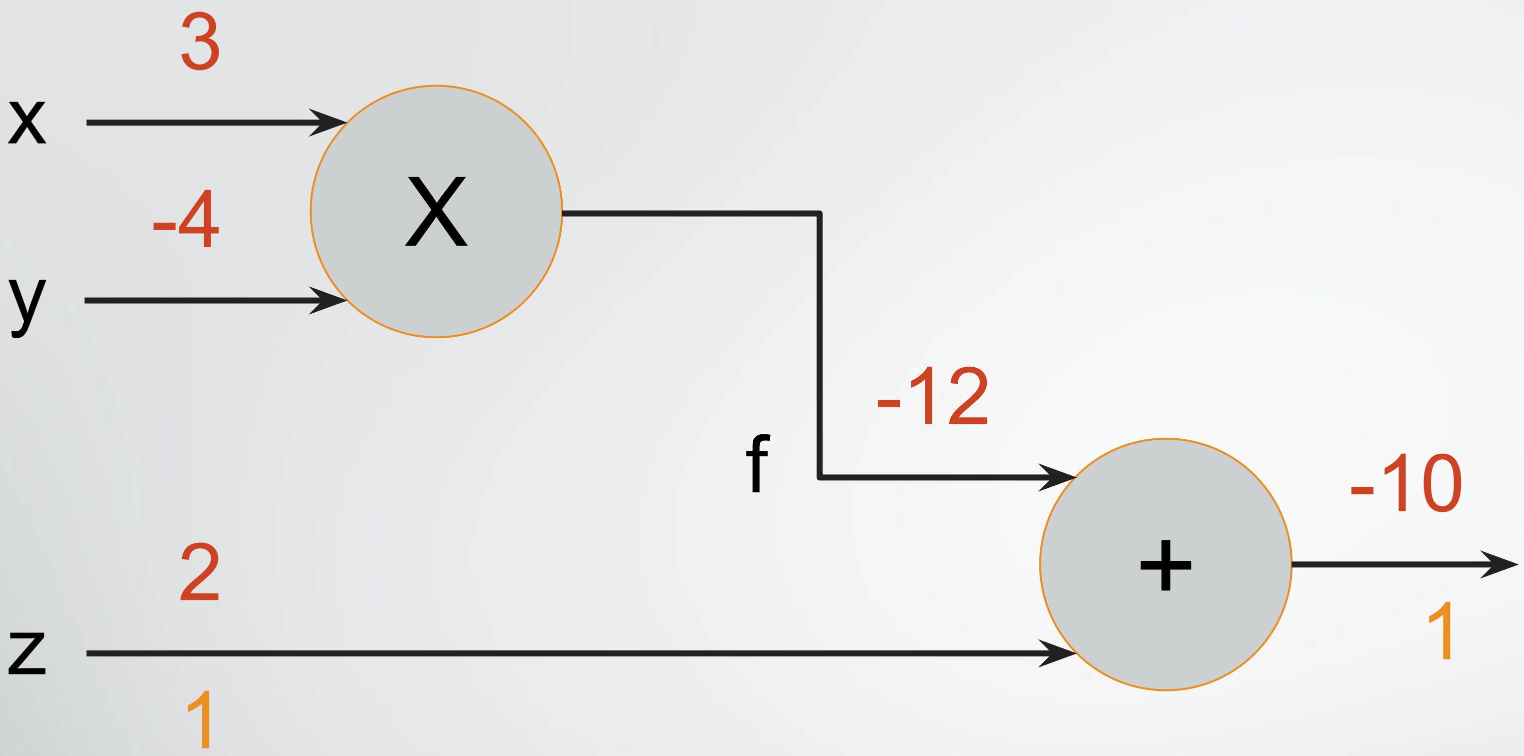
- So for our test point, the output is -10
- Now let's back-propagate the gradient
- This will tell us how we should alter the inputs in order to decrease the output



- The output's effect on itself, just one

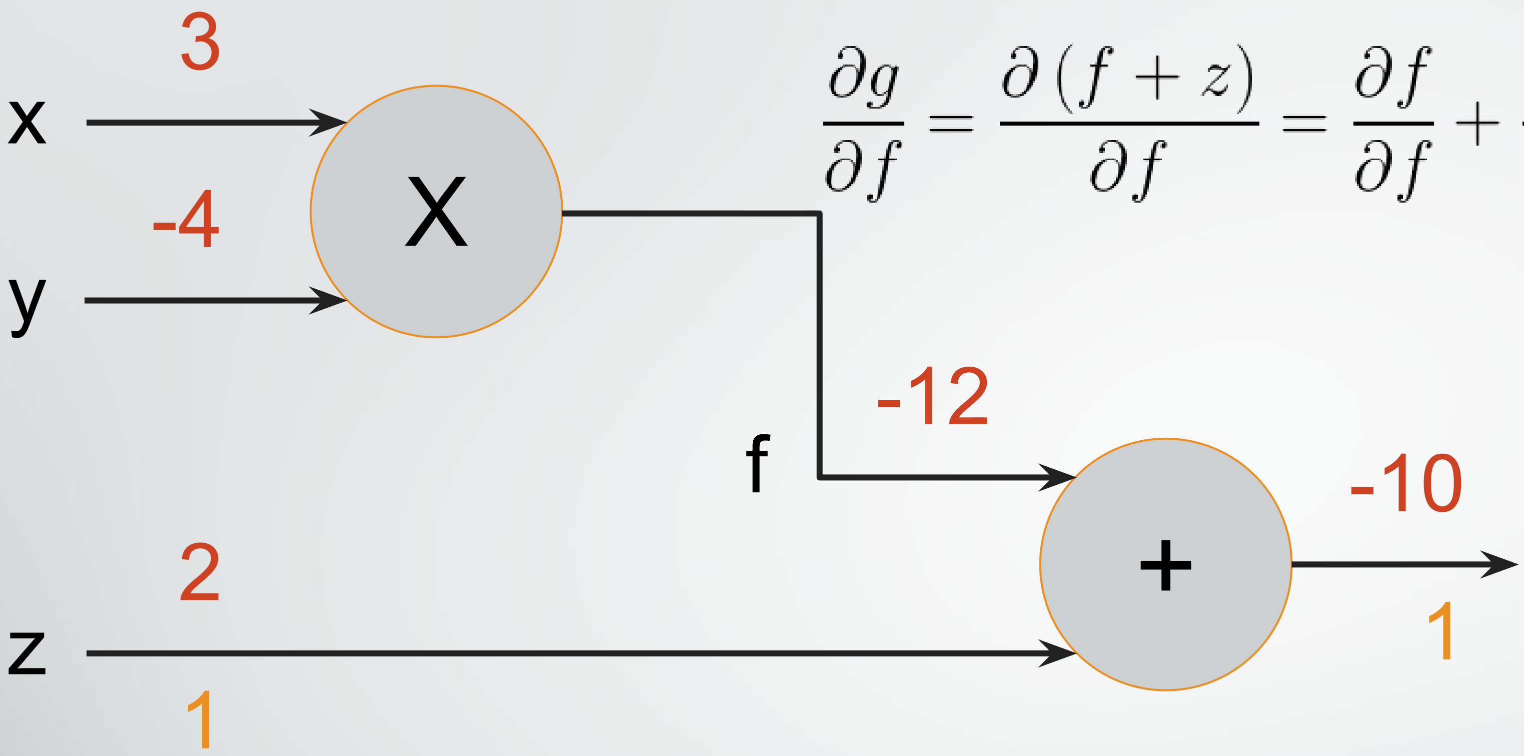


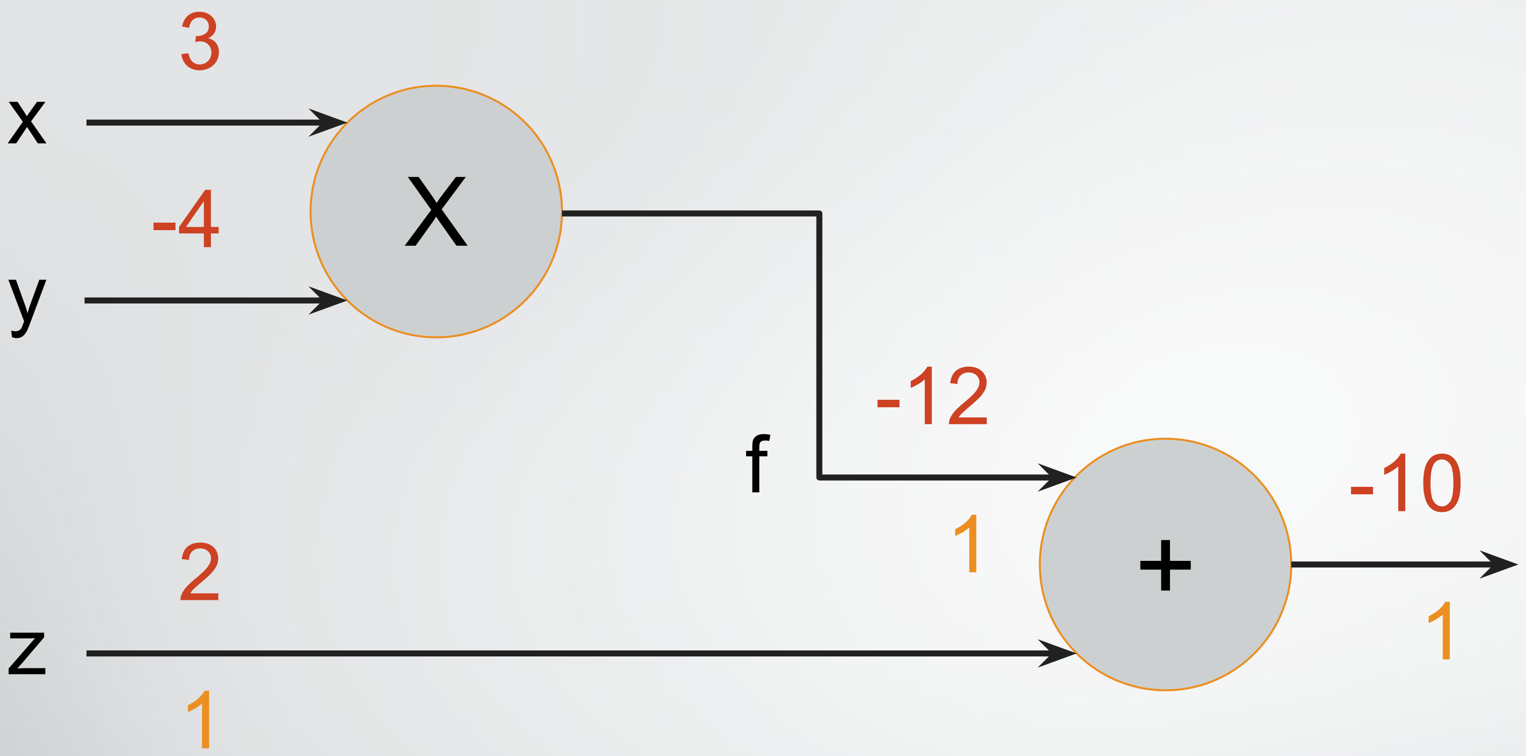
$$\frac{\partial g}{\partial z} = \frac{\partial\left(f+z\right)}{\partial z} = \frac{\partial f\left(x,y\right)}{\partial z} + \frac{\partial z}{\partial z} = 1$$



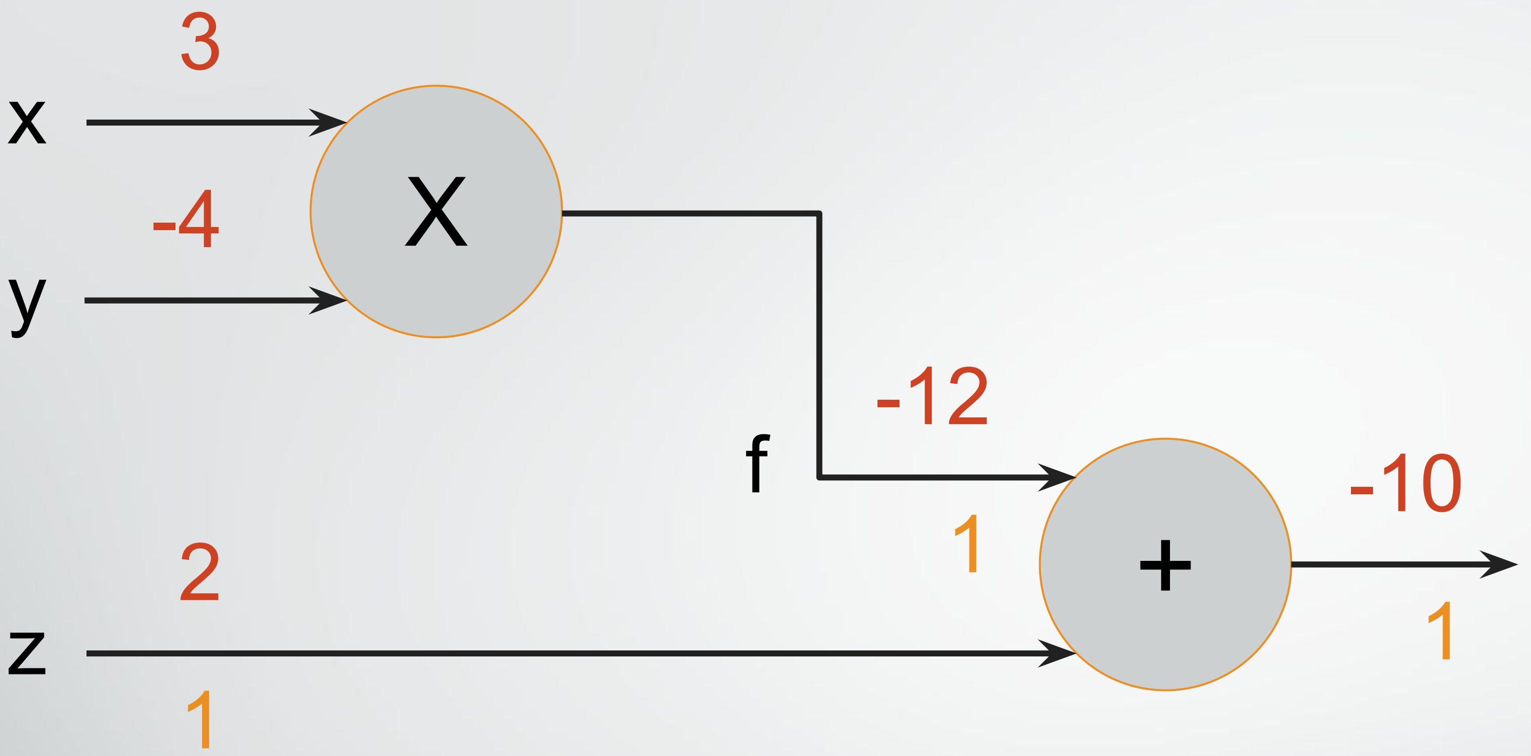
- Input *z* exerts a force of 1 on the output

$$\frac{\partial g}{\partial f} = \frac{\partial (f+z)}{\partial f} = \frac{\partial f}{\partial f} + \frac{\partial z}{\partial f} = 1$$

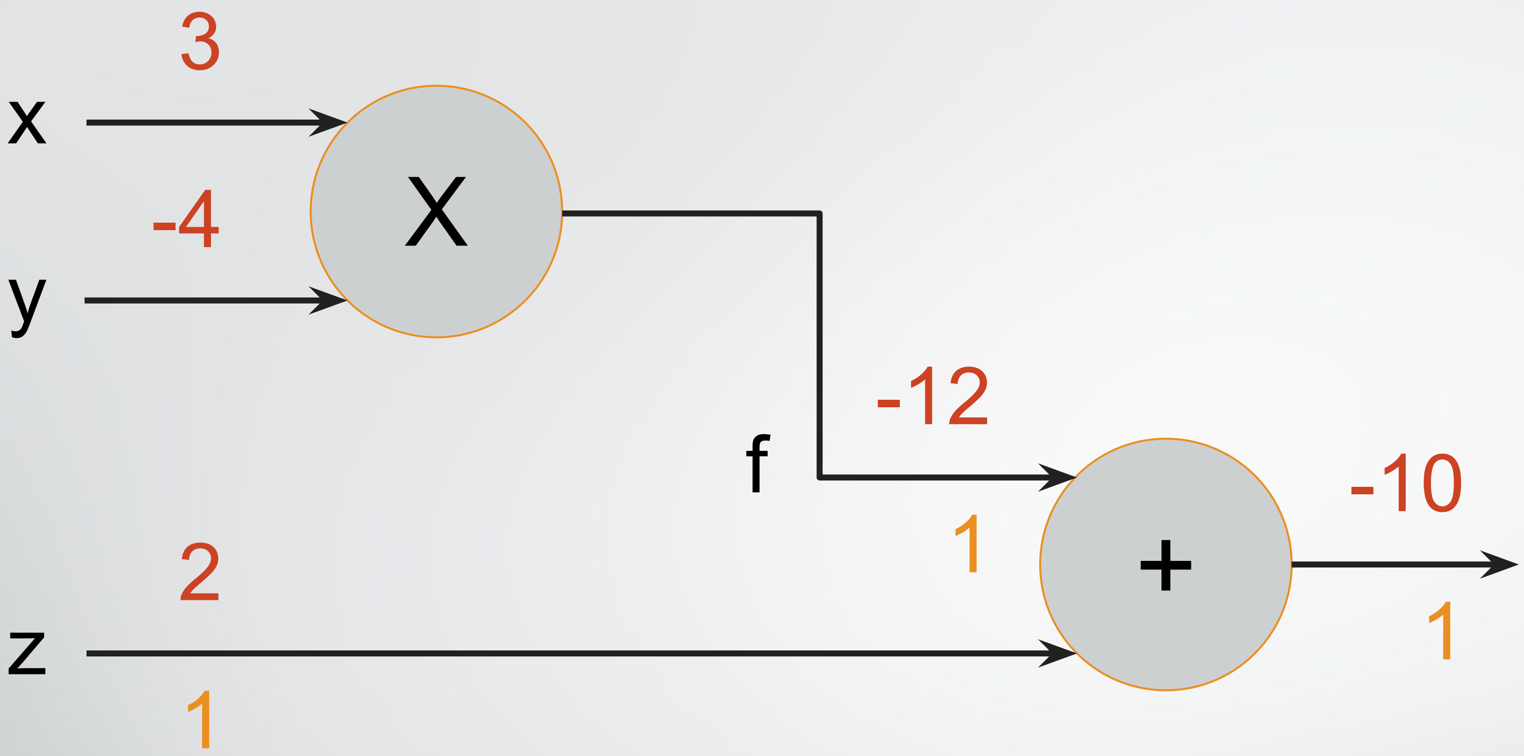




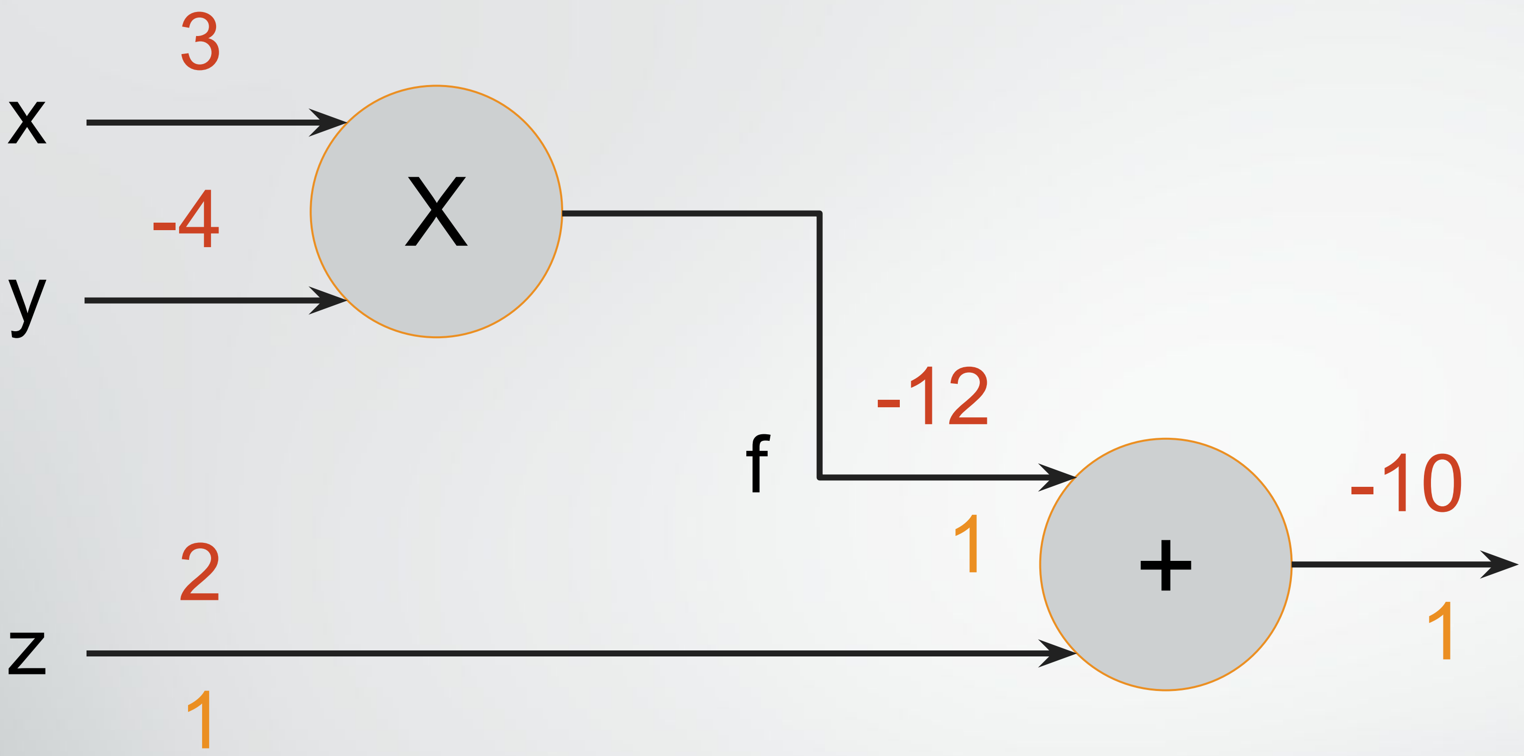
- As does the value of *f(x,y)*



- Now we want to evaluate the effect of *x* on *g*: $\dfrac{\partial g}{\partial x}$

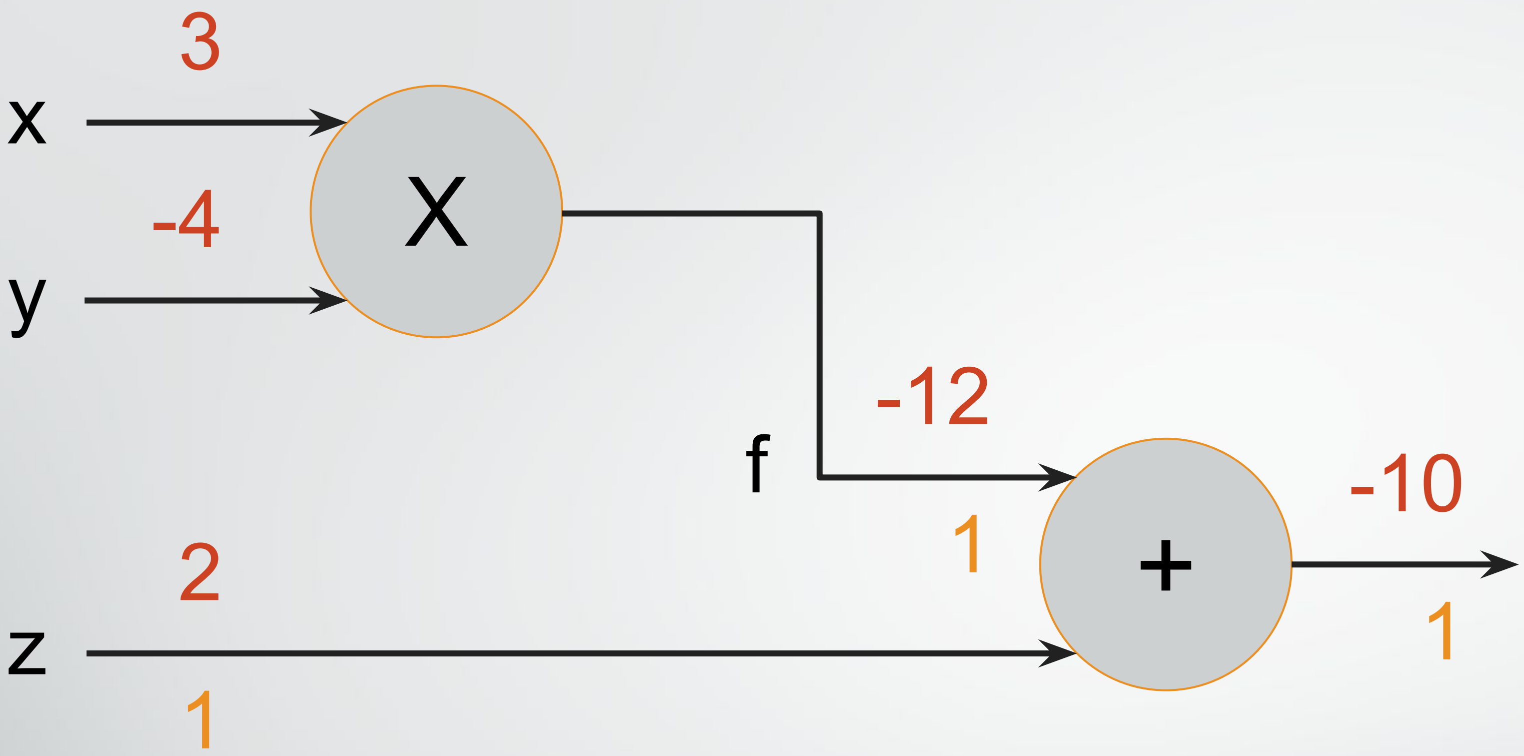


- Let's use the chain-rule: $\frac{\partial g}{\partial x} = \frac{\partial g}{\partial f} \times \frac{\partial f}{\partial x}$



- Let's use the chain-rule: $\frac{\partial g}{\partial x} = \boxed{\frac{\partial g}{\partial f}} \times \frac{\partial f}{\partial x}$

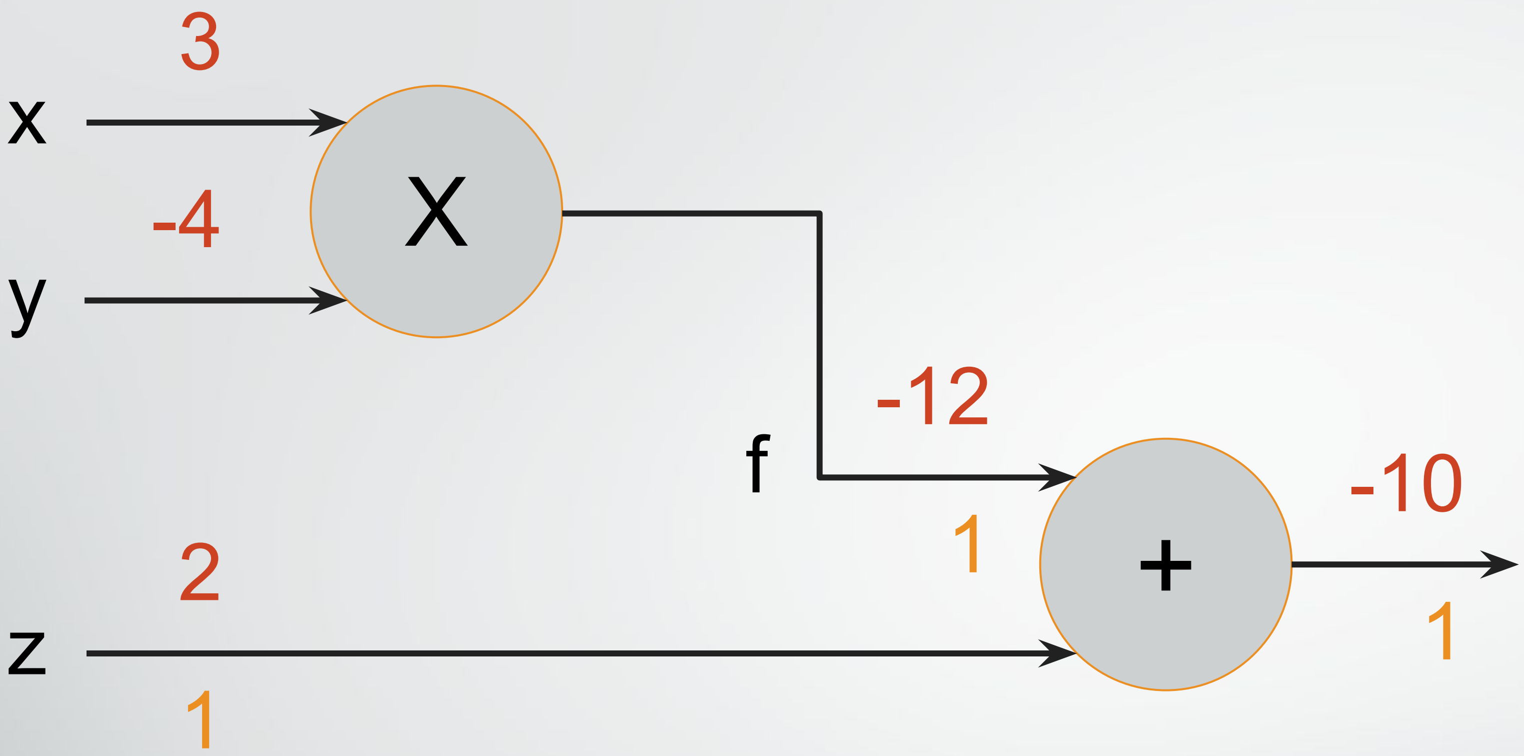
We know this already



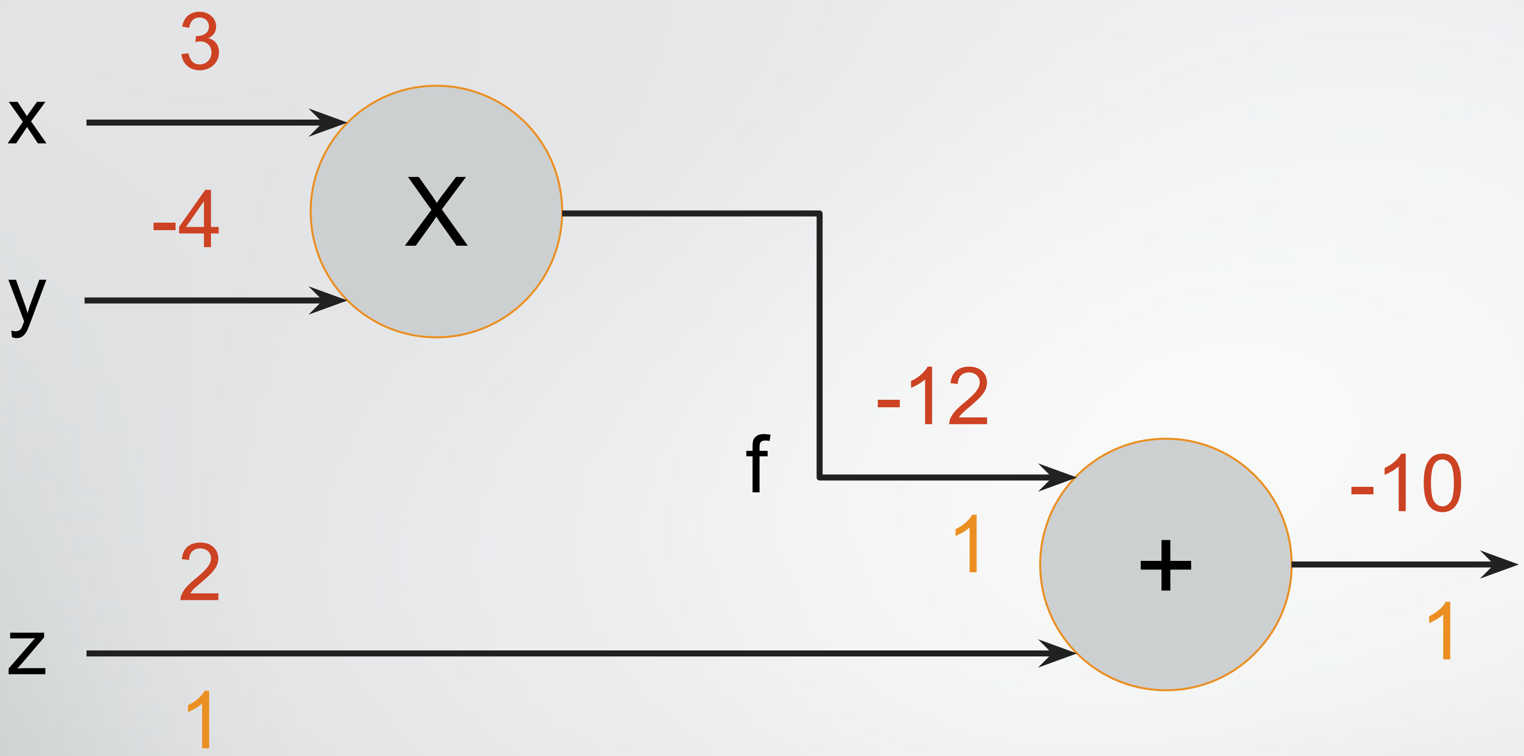
- Let’s use the chain-rule: $\frac{\partial g}{\partial x} = \boxed{\frac{\partial g}{\partial f}} \times \boxed{\frac{\partial f}{\partial x}}$

We know this already

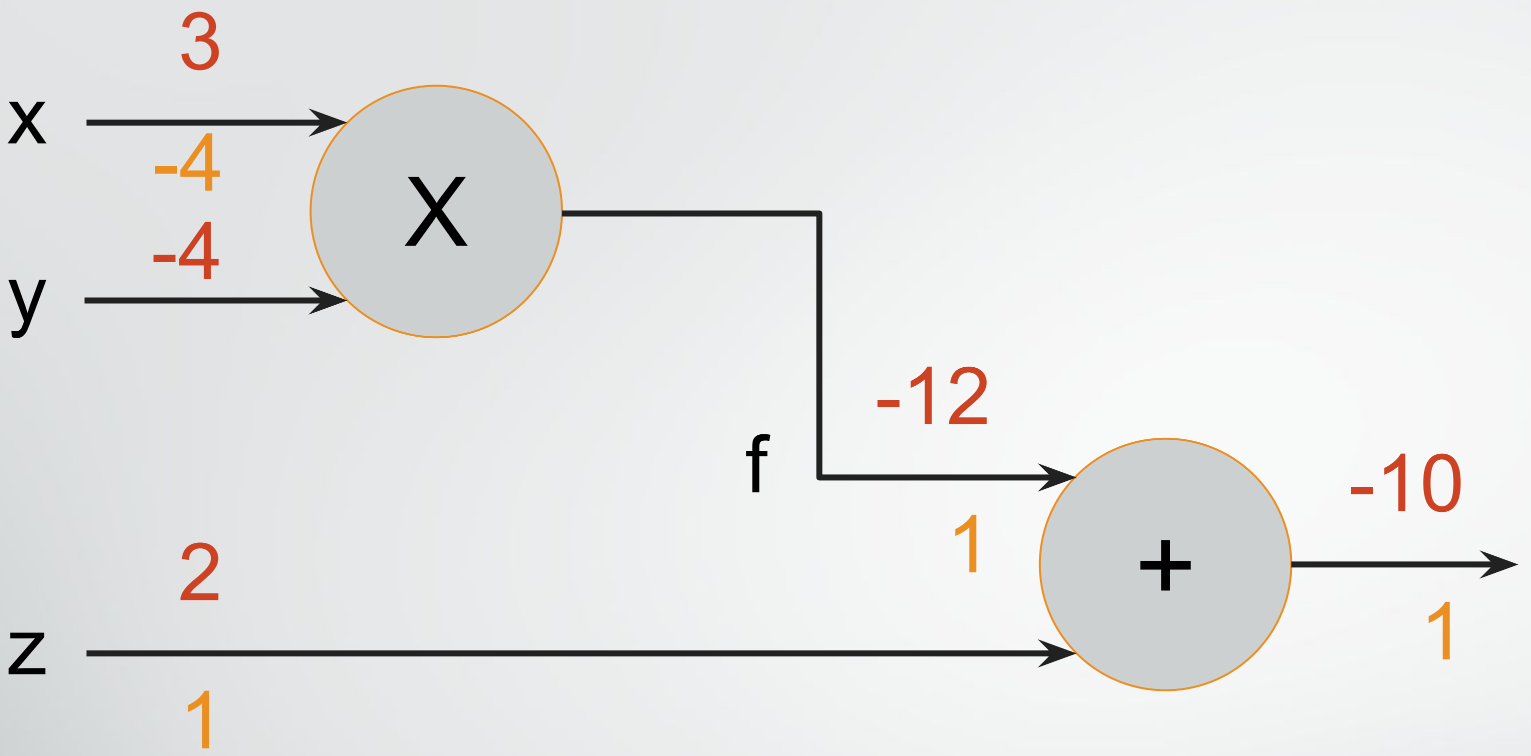
And we can evaluate this



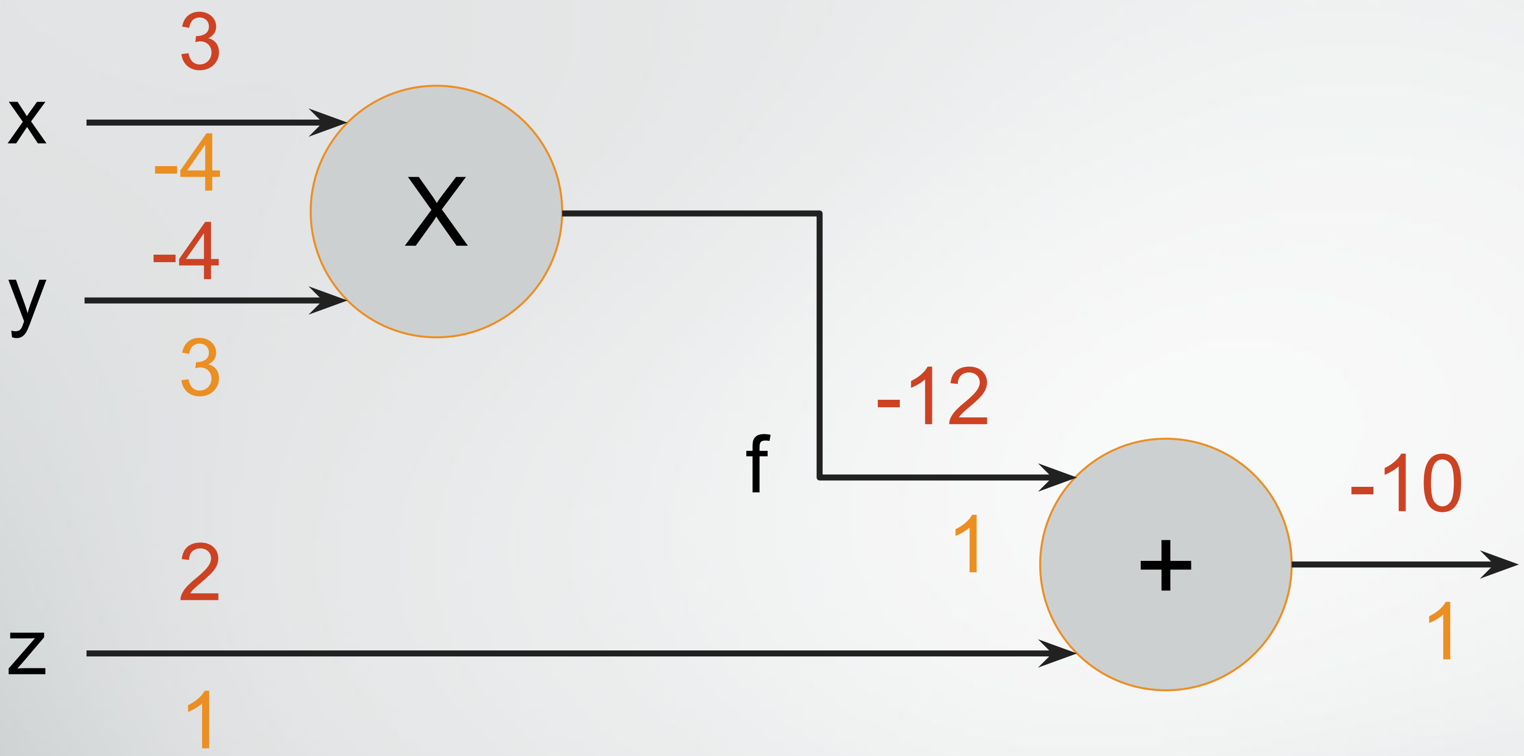
$$\frac{\partial f}{\partial x} = \frac{\partial (xy)}{\partial x} = y$$



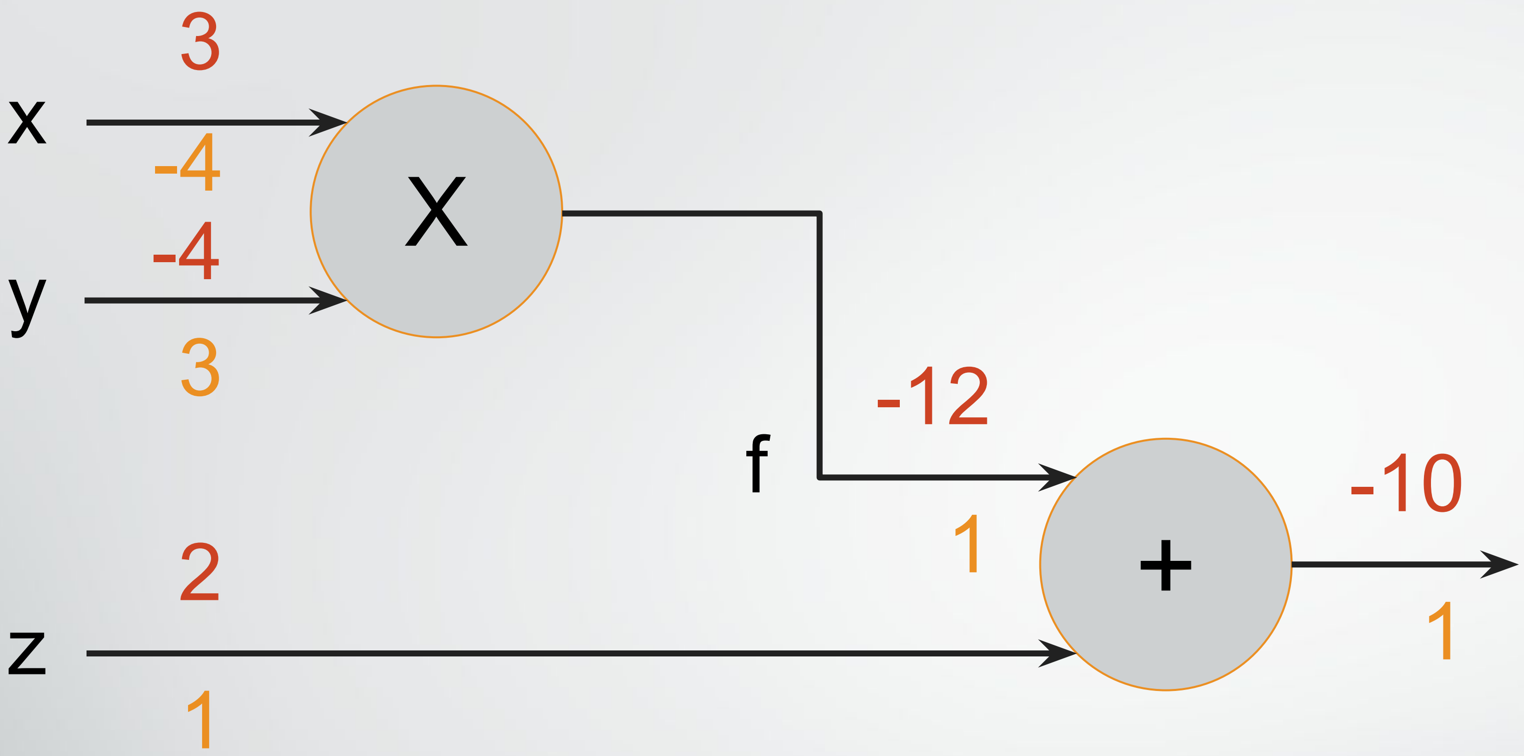
$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial f} \times \frac{\partial f}{\partial x} = 1 \times y = -4$$



- Similarly: $\frac{\partial g}{\partial y} = \frac{\partial g}{\partial f} \times \frac{\partial (xy)}{\partial y} = 1 \times x = 3$



- So, we now know each variable's effect on the output
- Now let's take one step down the gradient
- We'll use a step size (μ) of 0.1

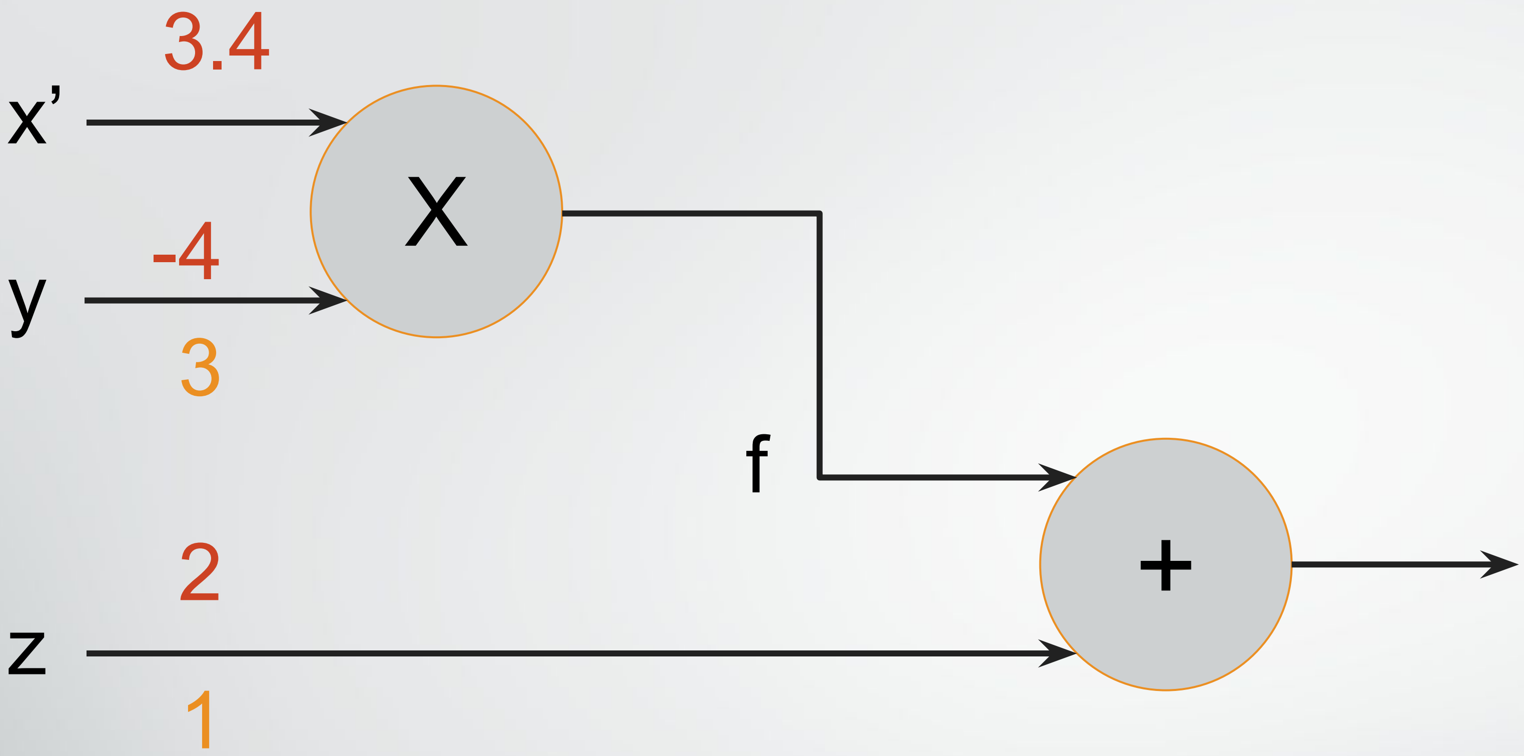


New value

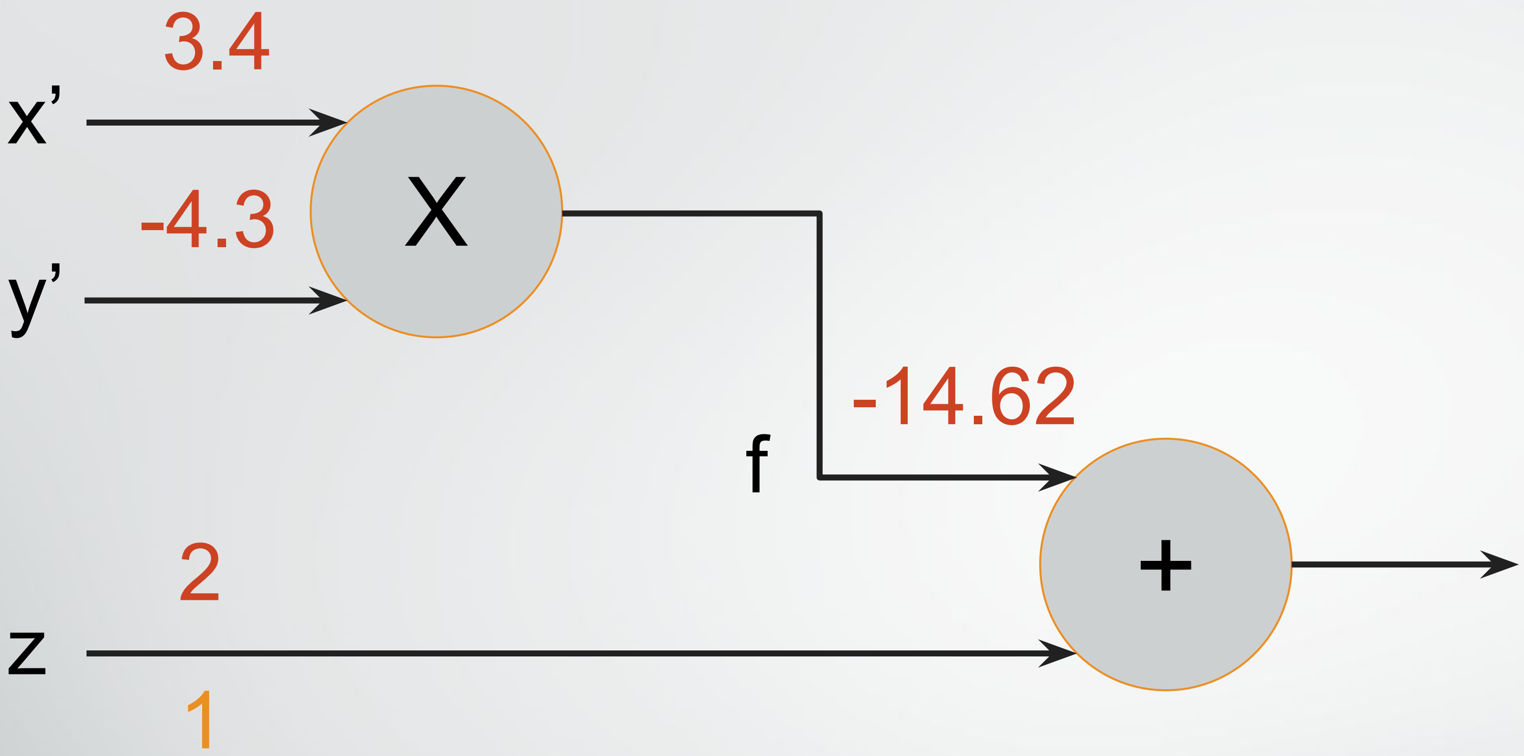
Move down the gradient

$$\boxed{x'} = x \boxed{-} \left(\frac{\partial g}{\partial x} \times \mu \right)$$

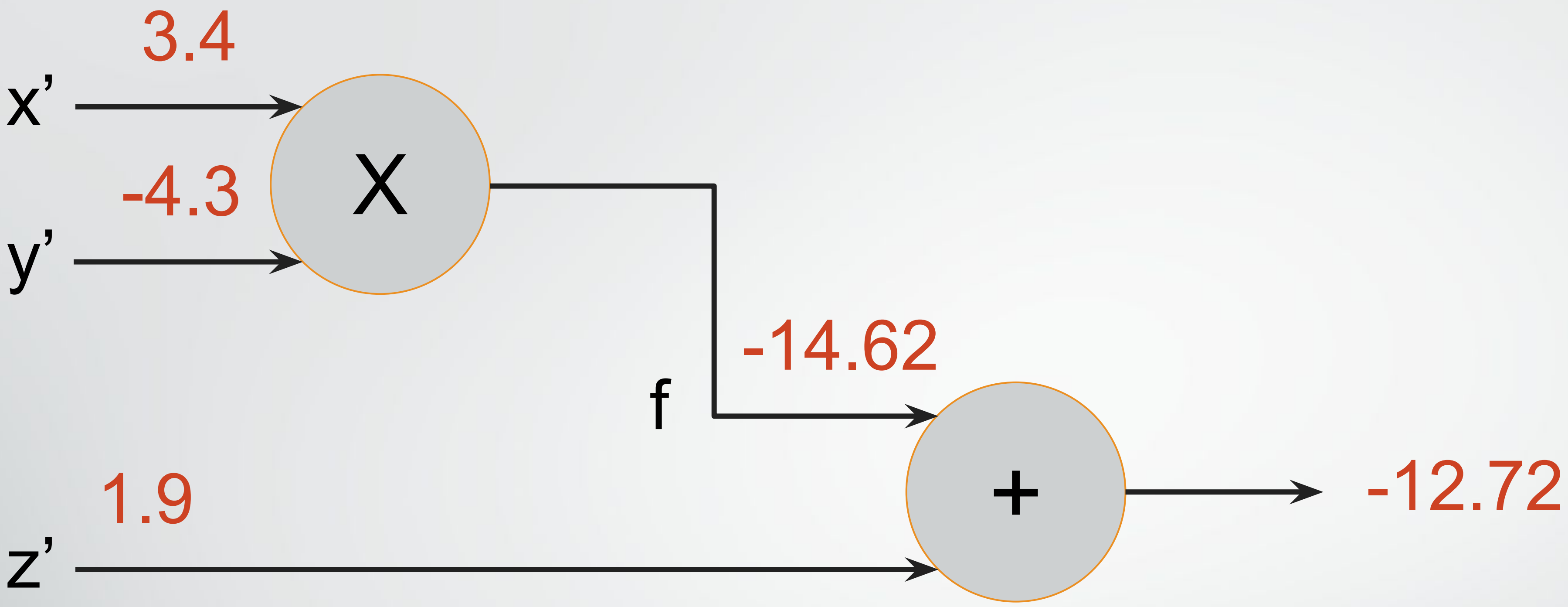
$$= 3 - (-4 \times 0.1) = 3.4$$



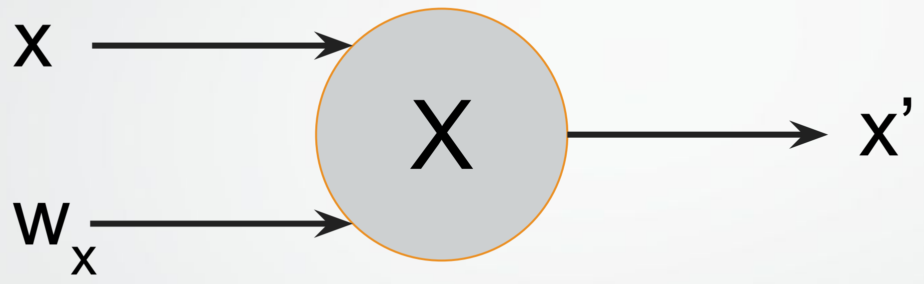
$$y' = y - \left(\frac{\partial g}{\partial y} \times \mu\right)$$
$$= -4 - (3 \times 0.1) = -4.3$$



$$z' = z - \left(\frac{\partial g}{\partial z} \times \mu\right)$$
$$= 2 - (1 \times 0.1) = 1.9$$



- Having updated our inputs, we find that the output has decreased by 2.72

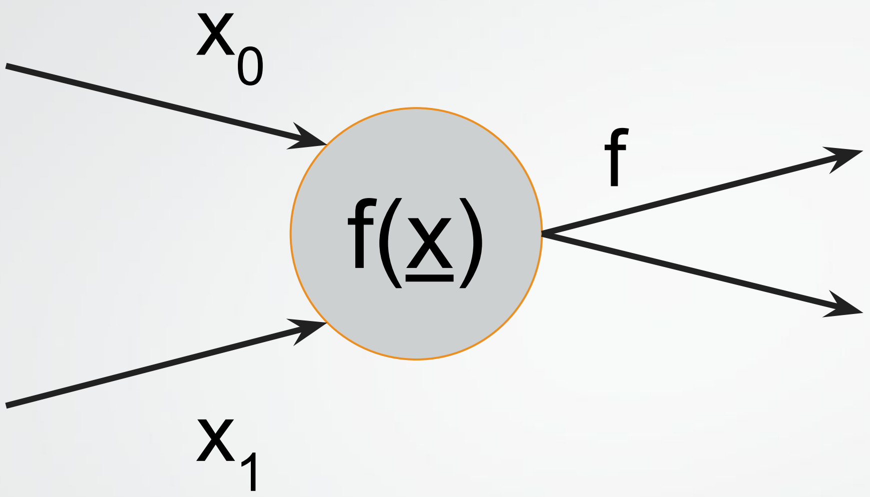


- In actual implementation we can’t change our input data
- Instead we weight the incoming signals
- This is just another ‘sub-neuron’
- Meaning we can back-propagate the gradient into it

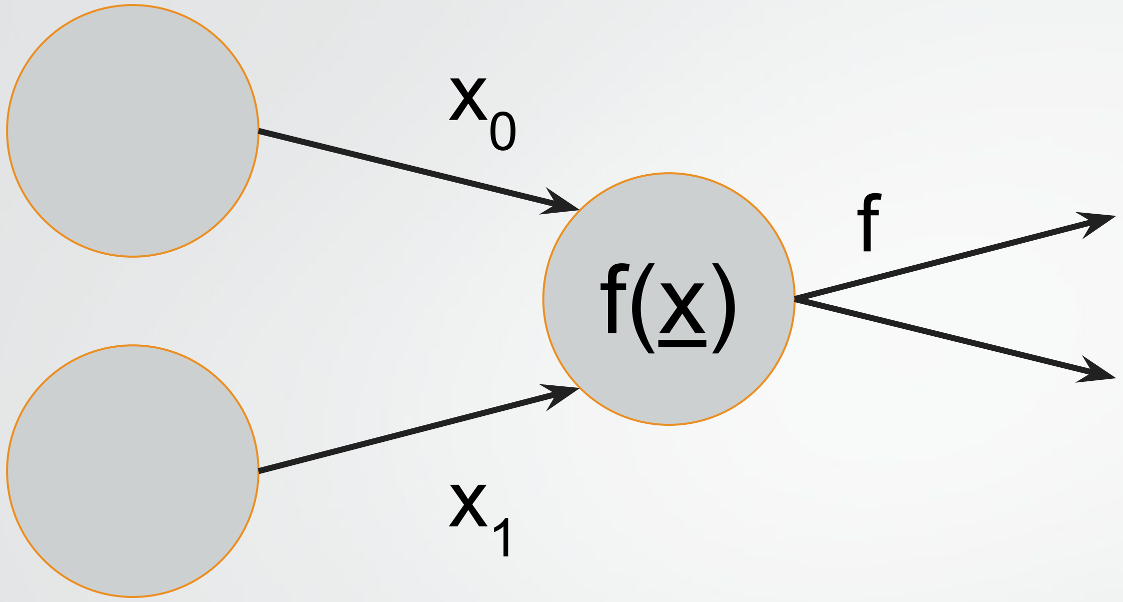
- Let’s generalise and recap

$$f(\underline{x})$$

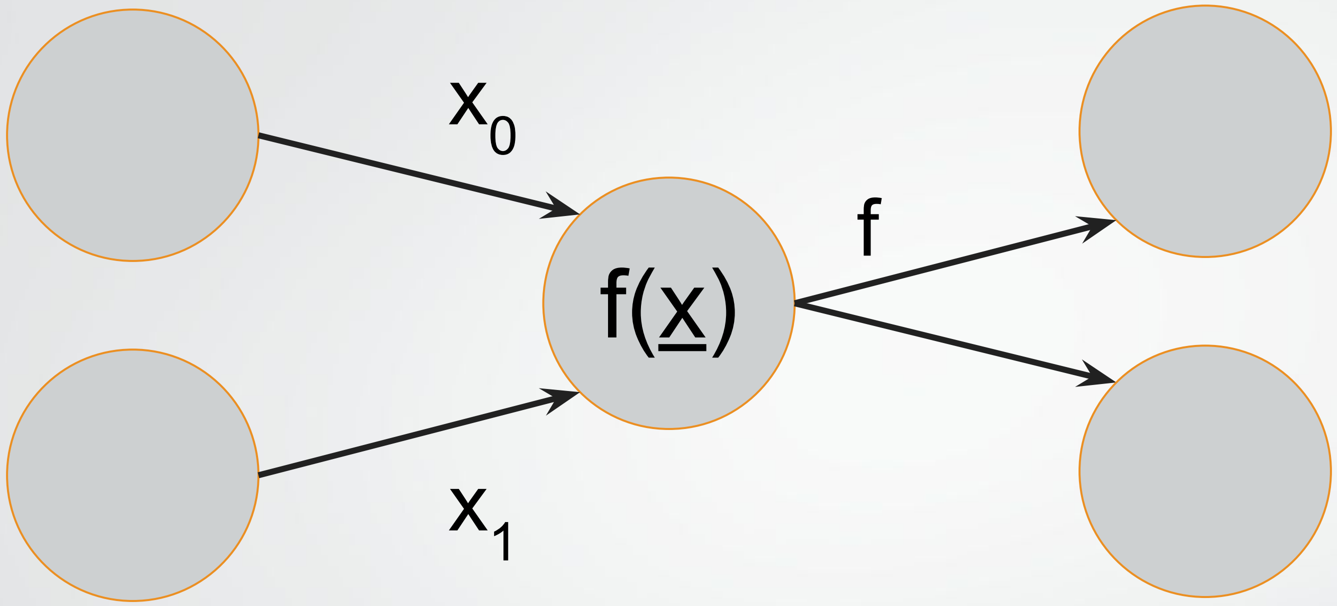
- Let’s generalise and recap
- We have a neuron in a network



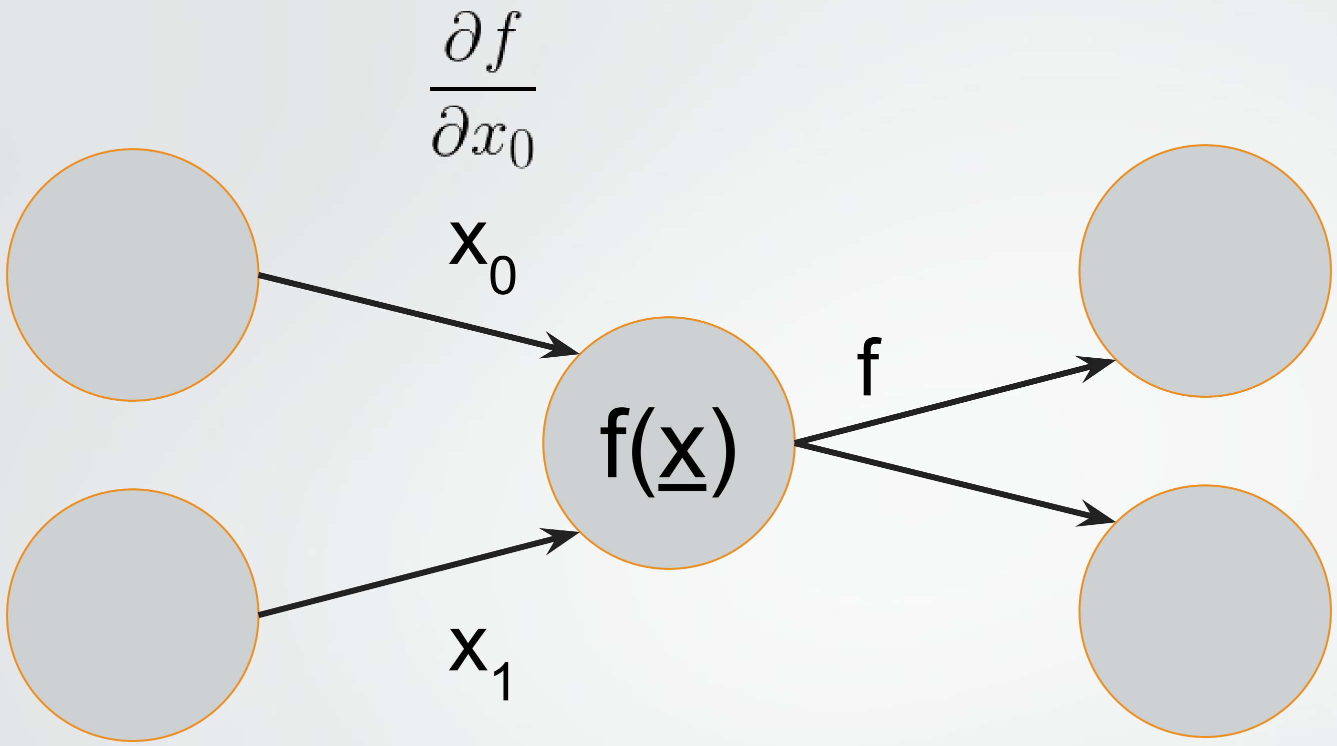
- Let's generalise and recap
- We have a neuron in a network
- It receives inputs, applies a function, and produces an output



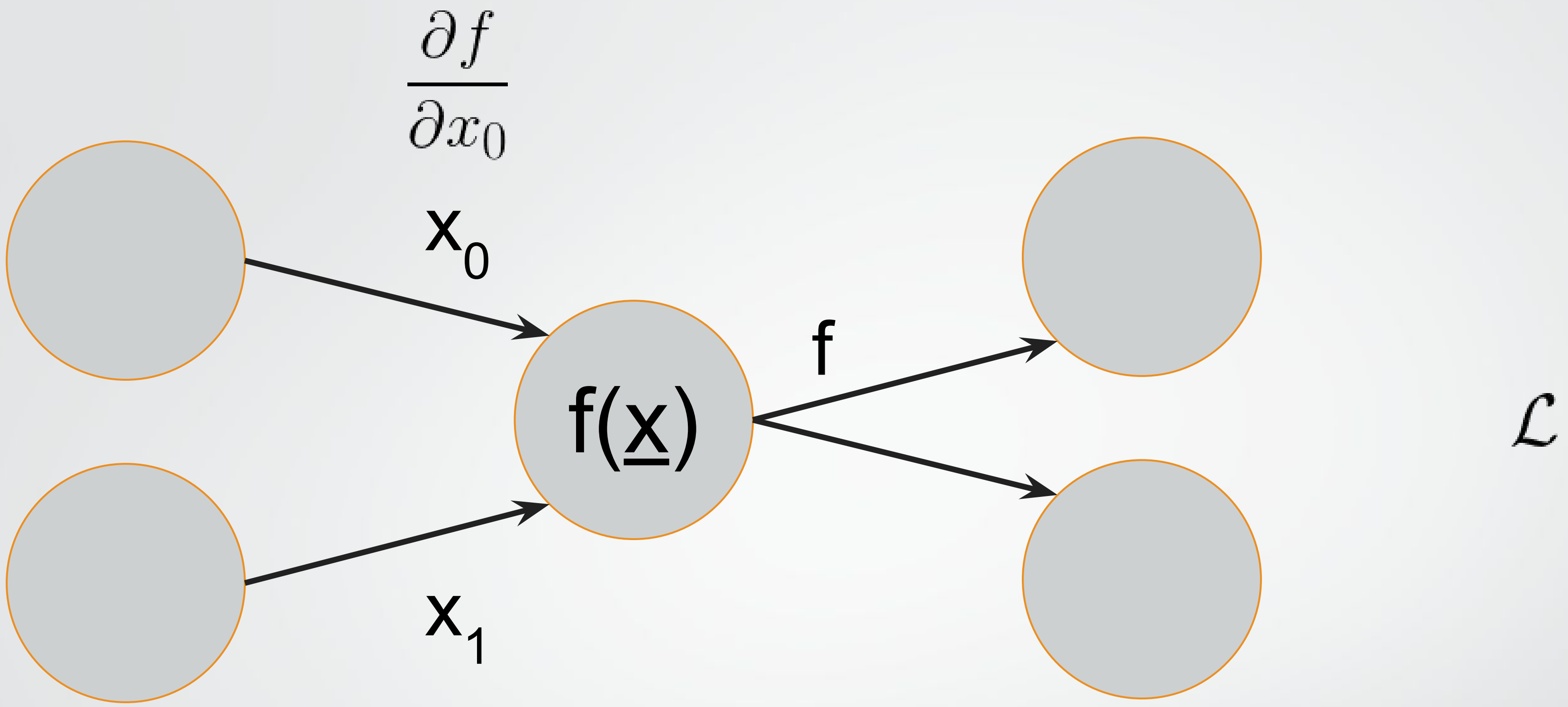
- We have a neuron in a network
- It receives inputs, applies a function, and produces an output
- These inputs come from neurons in the previous layer



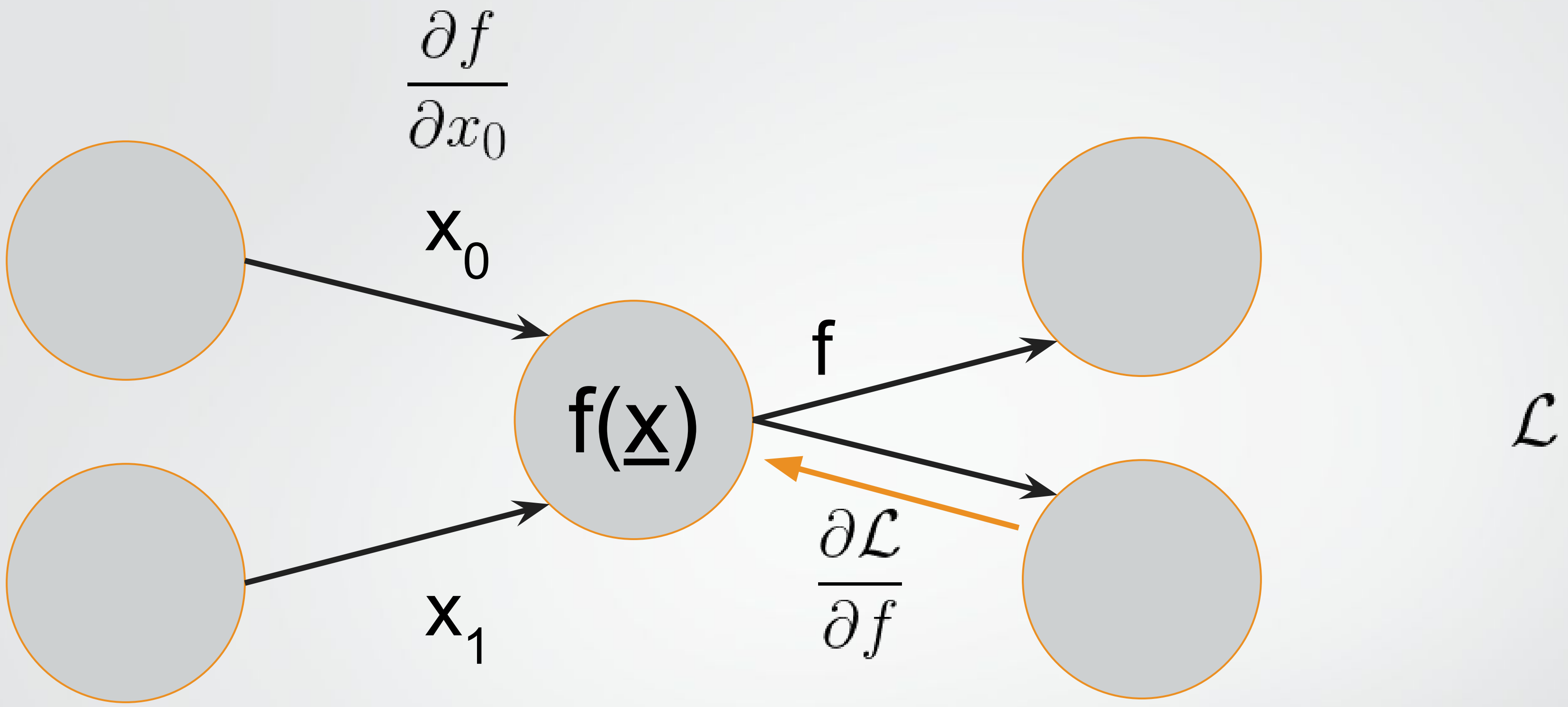
- It receives inputs, applies a function, and produces an output
- These inputs come from neurons in the previous layer
- And the outputs are passed to the next layer



- These inputs come from neurons in the previous layer
- And the outputs are passed to the next layer
- At the same time as calculating its output, the neuron can also compute its *local gradients*

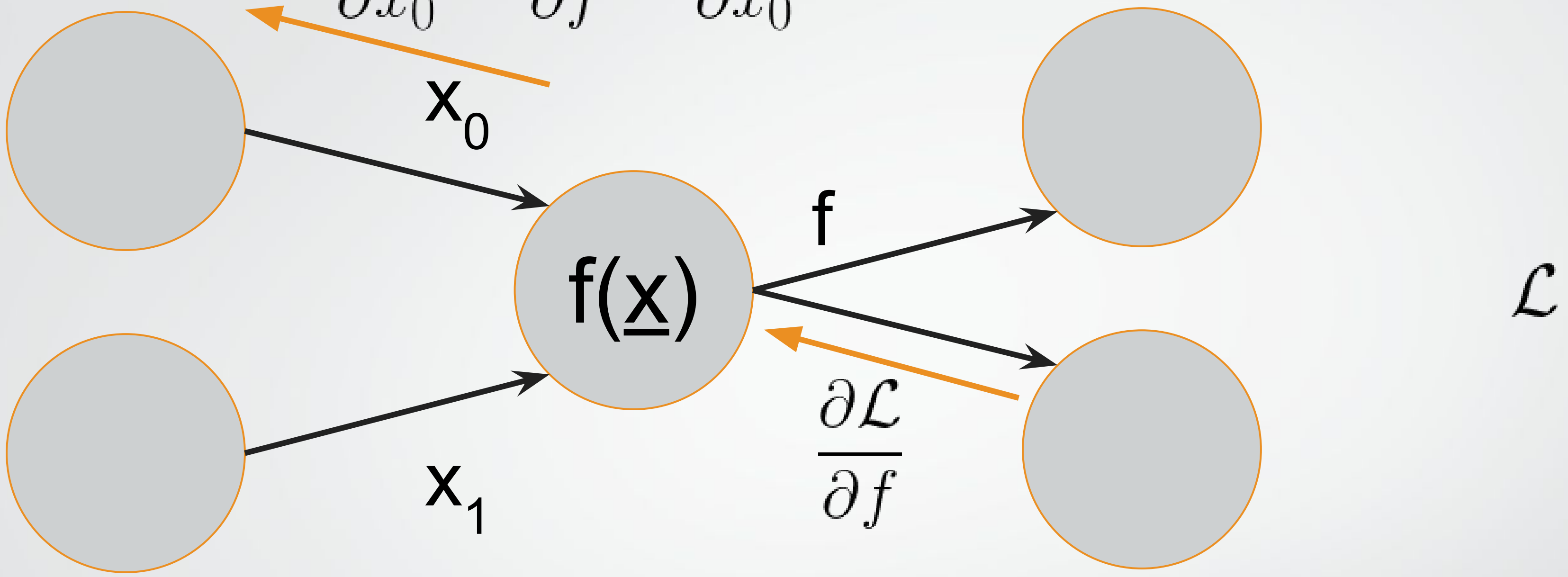


- And the outputs are passed to the next layer
- At the same time as calculating its output, the neuron can also compute its *local gradients*
- Eventually the loss function gets computed

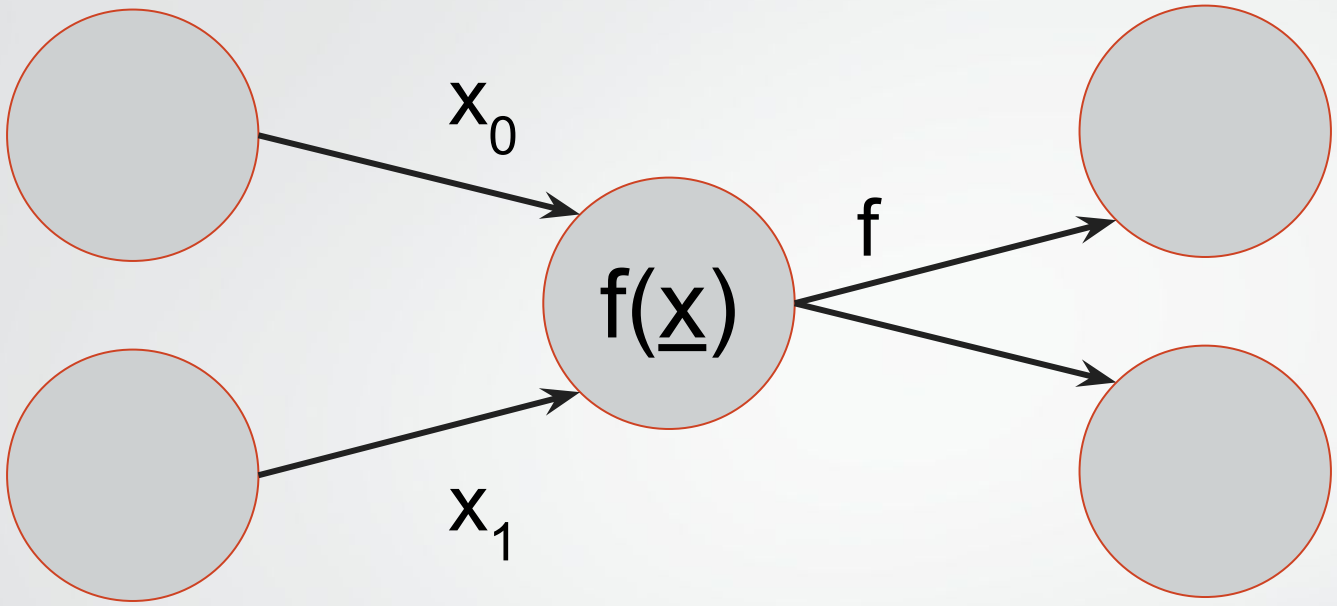


- Eventually the loss function gets computed
- The gradient of the loss eventually gets back-propagated to our neuron
- The neuron sees the effect of its output on the loss

$$\frac{\partial \mathcal{L}}{\partial x_0} = \frac{\partial \mathcal{L}}{\partial f} \times \frac{\partial f}{\partial x_0}$$



- The neuron sees the effect of its output on the loss
- Having already calculated its local gradients, the neuron simply times this by the incoming gradient (chain-rule)
- The new gradient propagates on to the next layer



- Having already calculated its local gradients, the neuron simply times this by the incoming gradient (chain-rule)
- The new gradient propagates on to the next layer
- Having calculated all the analytic gradients we can update the weights by stepping down the gradient

Back propagation – 1960-1986

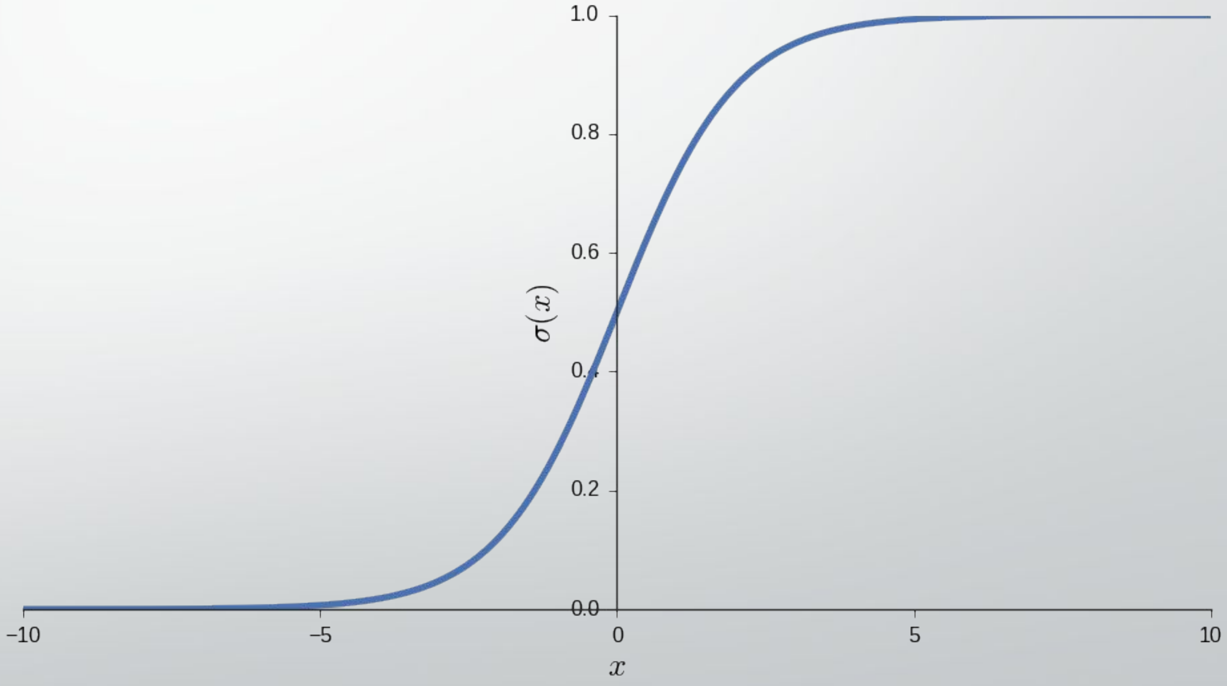
- Weight-learning based on chain-rule differentiation
- Basics, Keely 1960 and Bryson 1962
- First applied to ANNs in 1982 by Werbos
- Shown to be useful in multi-layer ANNs by Rumelhart, Hinton, and Williams in 1986
- However, ANNs still underperformed, and were limited in size; training would get stuck
- Interest in ANNs diminishes

Problems

- Even with back-propagation, NNs would get stuck during training
- Why did it take another 28 years for them to become useful?

Problem 1: Activation function

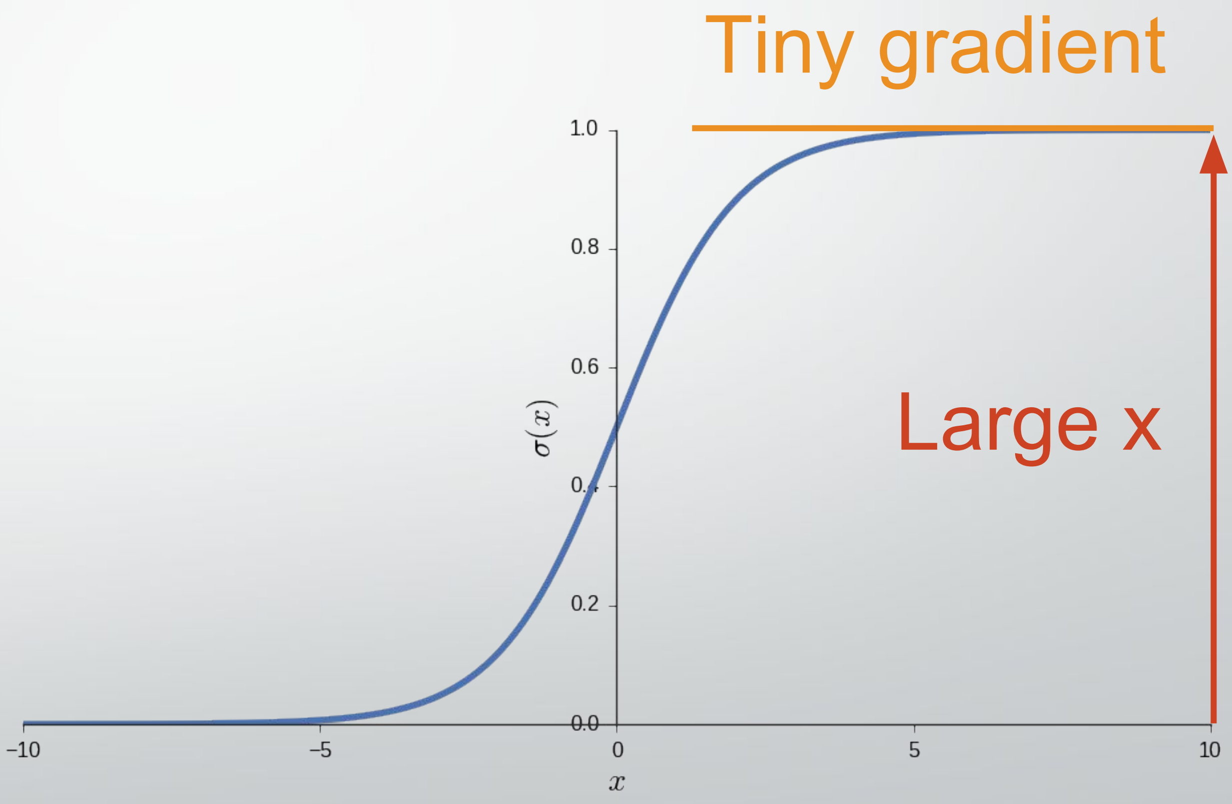
- The sigmoid function was used because it was smooth between the bounds of zero and one
- Early ‘connectionist’ interpretations of NNs likened it to the firing rate of a biological neuron
- But it has several problems…



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

1: It can kill gradients during back-prop

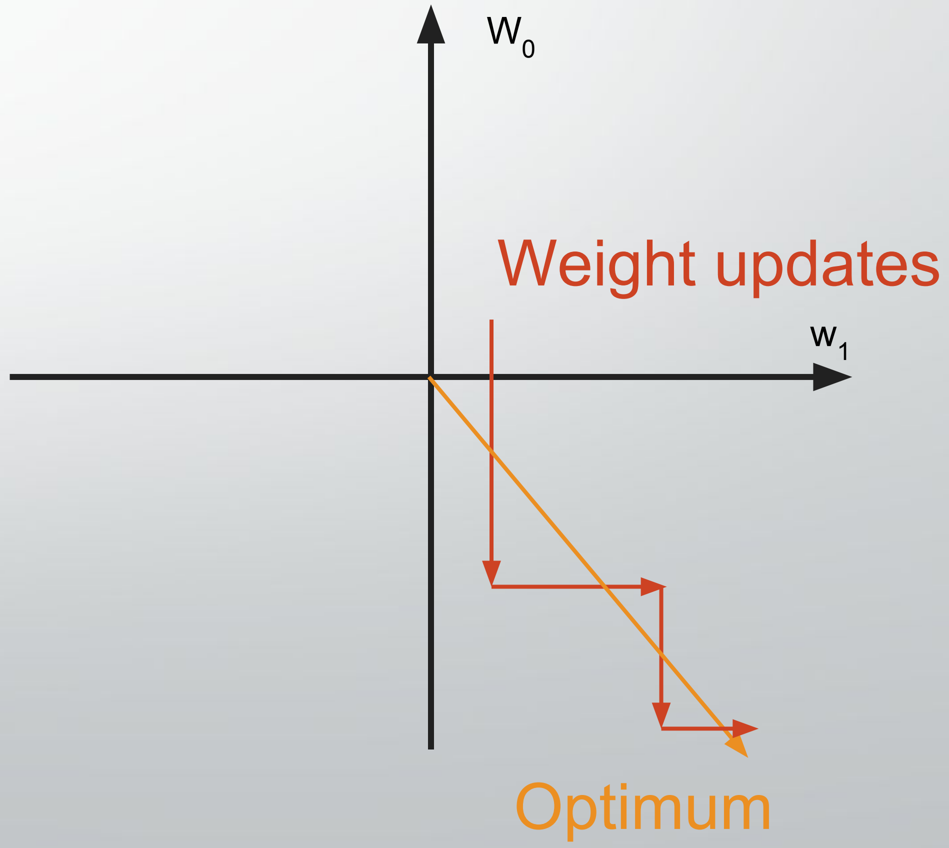
- When |x| is large, the local gradient drops close to zero
- The saturated neuron effectively passes zero loss-gradient back to previous layers
- This stops them from updating their weights



$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \sigma}{\partial x} \times \frac{\partial \mathcal{L}}{\partial \sigma} = 0 \times \frac{\partial \mathcal{L}}{\partial \sigma} = 0$$

2: The outputs are not zero-centred

- Outputs are always positive
- Gradients propagated to the weights are therefore either always positive or always negative
- If the optimum set of weights is a mixture of positive and negative weights, then this can only be reached by zigzagging towards the optimum position

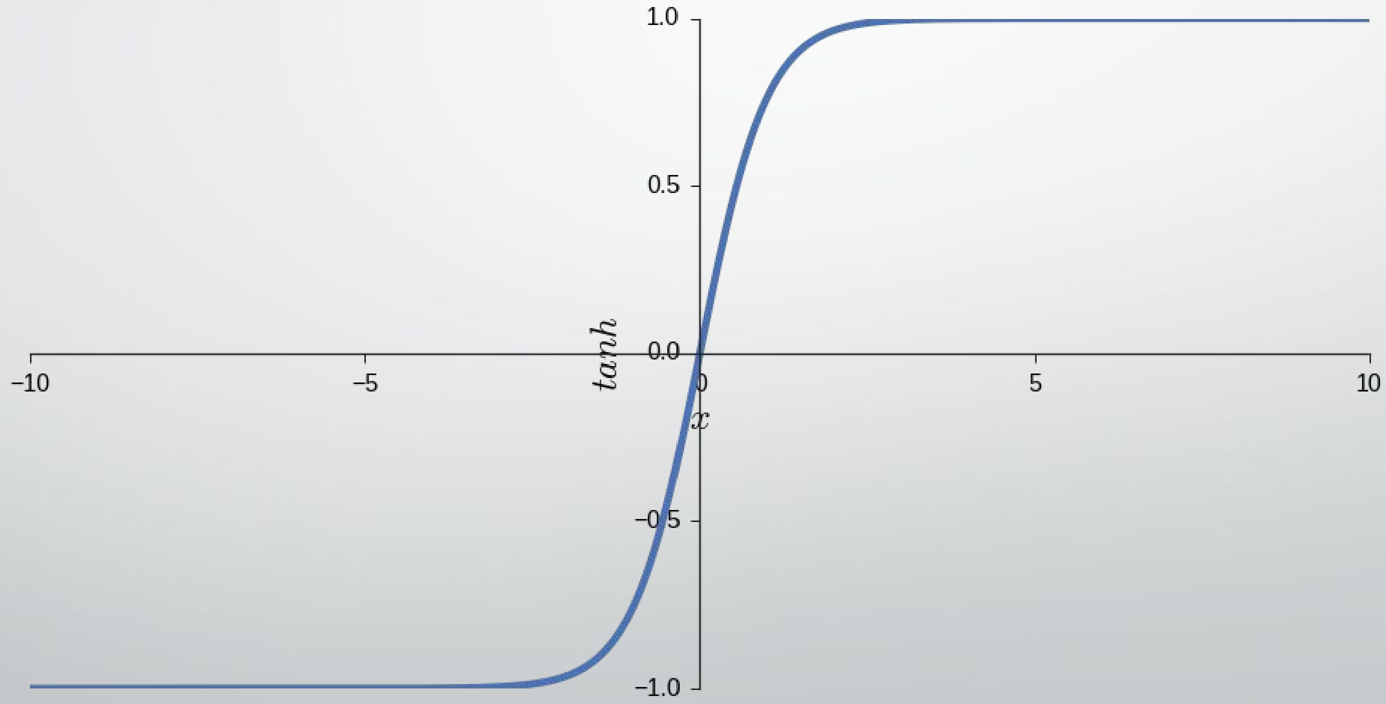


3: Expensive to compute

- The sigmoid function contains the exponential function
- This requires a lot of CPU time to compute, compared to other functions
- Only a slight slowdown, but a slowdown nonetheless
- Especially once networks start to get large

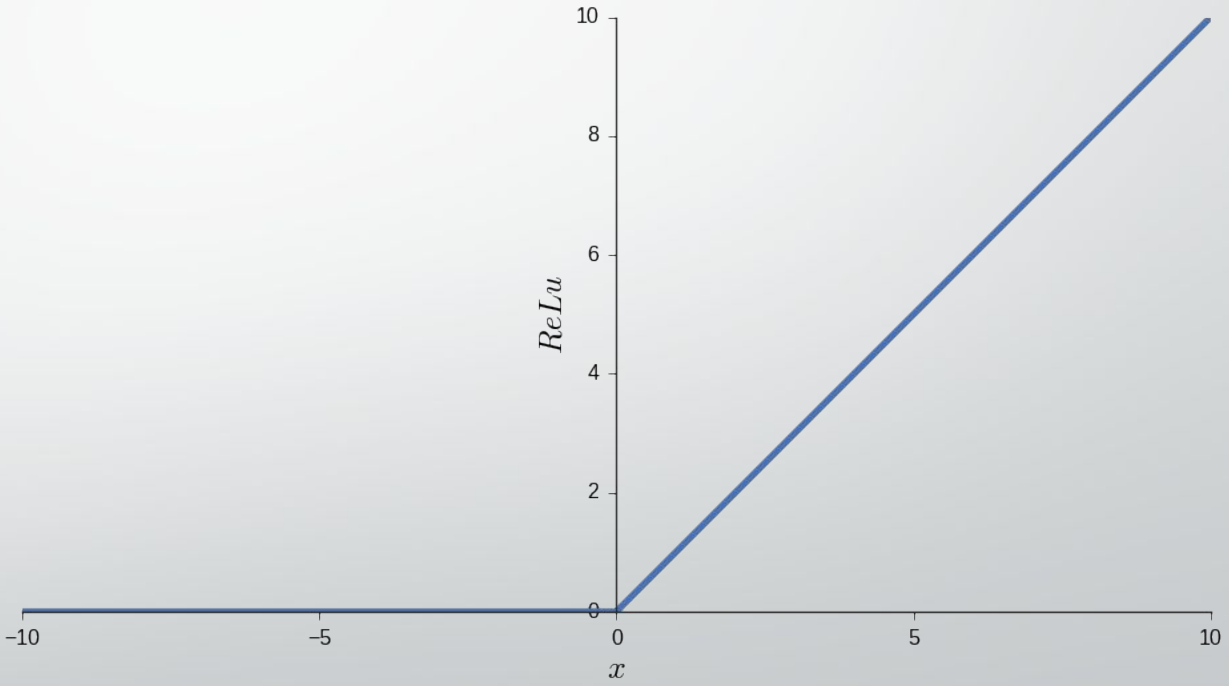
$$\sigma(x) = \frac{1}{1 + \boxed{e}^{-x}}$$

An improvement: tanh



The solution

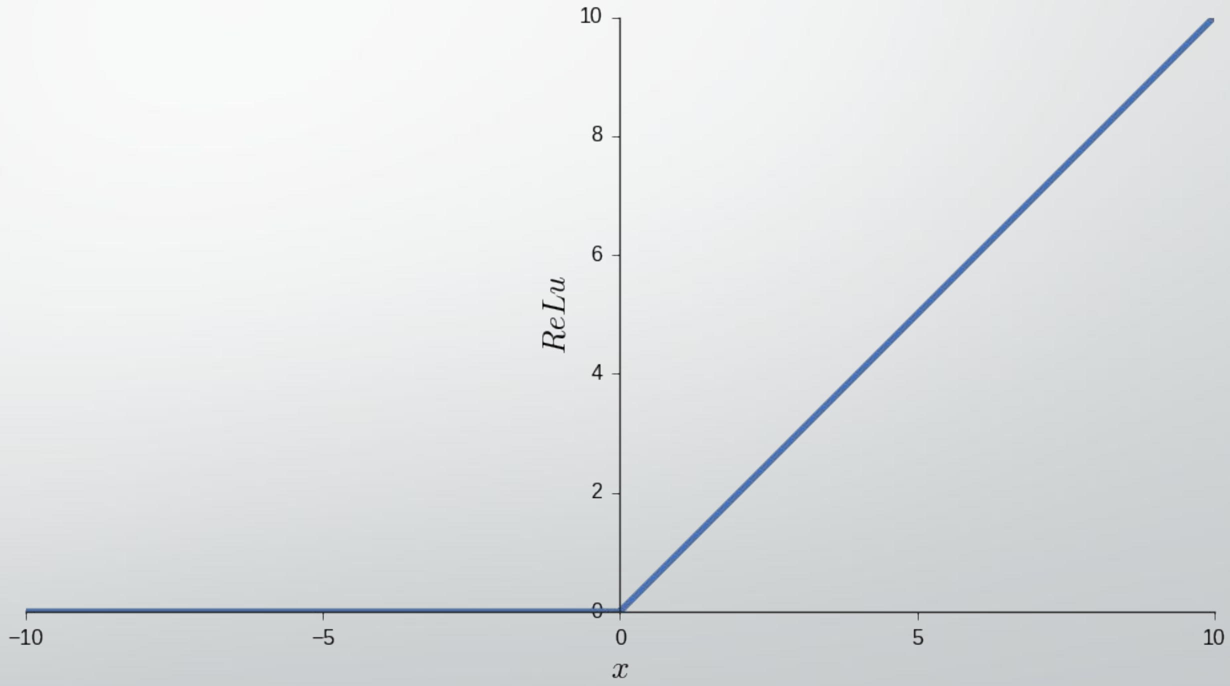
- Use a rectified linear-unit as the activation function
- Introduced in 2000 by Hahnloser et al.
- Gradient never saturates in positive region
- Easy to compute
- Shown to converge 6 times more quickly than tanh; Hinton, Krizhevsky, and Sutskever 2012



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

The solution

- Still non-zero centred
- Still kills gradients in negative region
- Depending on initialisation of weights, can sometime never activate (dead ReLu)



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

Problem 2: Initialisation

- How exactly do we initialise the weights in a network?
- Could set them all to the same value; they'd all respond the same way
- We need something 'symmetry breaking'

Problem 2: Initialisation

- Default was to sample a Gaussian distribution and times by some factor
- If the factor were too large then the neurons would saturate (for sigmoid and tanh); gradients go to zero, nothing trains
- If the factor is too small, the output of the network becomes zero
- Factor must be set carefully by hand

The solution

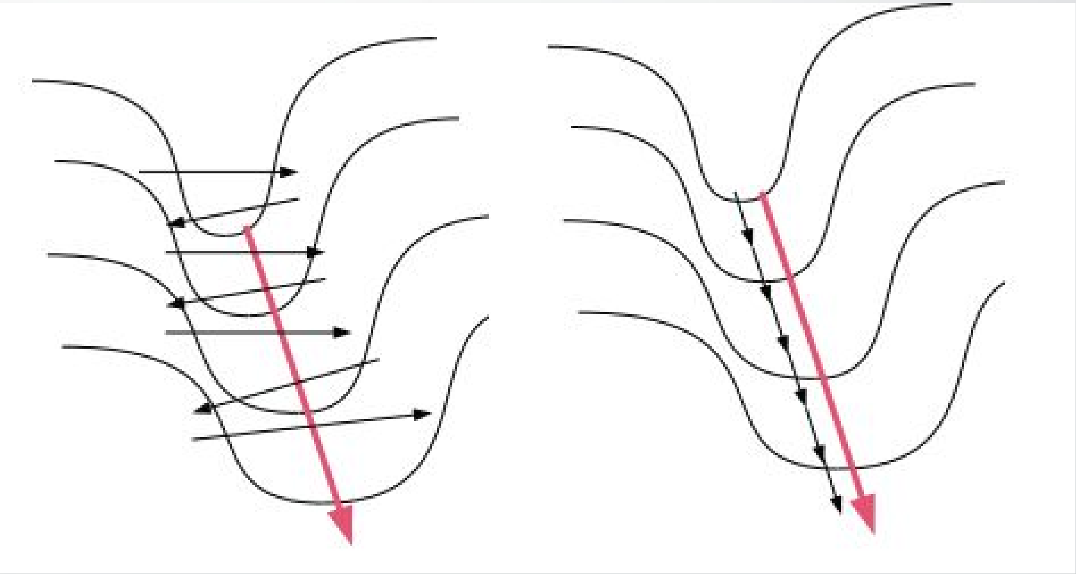
- Mathematically sensible solution proposed by Bengio and Glorot in 2010: Xavier initialisation
- Applies a factor of $N_{\text{in}}^{-\frac{1}{2}}$
- For neurons with few inputs, the weights are higher
- For neurons with many inputs, the weights are lower
- Similar levels of outputs throughout the network

The solution

- Was derived without accounting for the activation function
- Works well for sigmoid and tanh
- Doesn’t work for ReLu; results in lots of dead neurons
- Instead, an extra factor of two must be added: $\left(\frac{N_{\text{in}}}{2}\right)^{-\frac{1}{2}}$
- He et al, 2015

Problem 3: Convergence time

- Gradient descent is able to optimise the weights
- However, it can easily slow down in narrowly sloping 'valleys'



How GD moves   Ideal moves

Standard gradient descent

Current parameters

- The standard update is: $\boxed{\theta_{t+1}} = \boxed{\theta_t} - \boxed{\mu}\,\boxed{\nabla \mathcal{L}(\theta_t)}$

New parameters

Gradient at current parameters

Learning rate

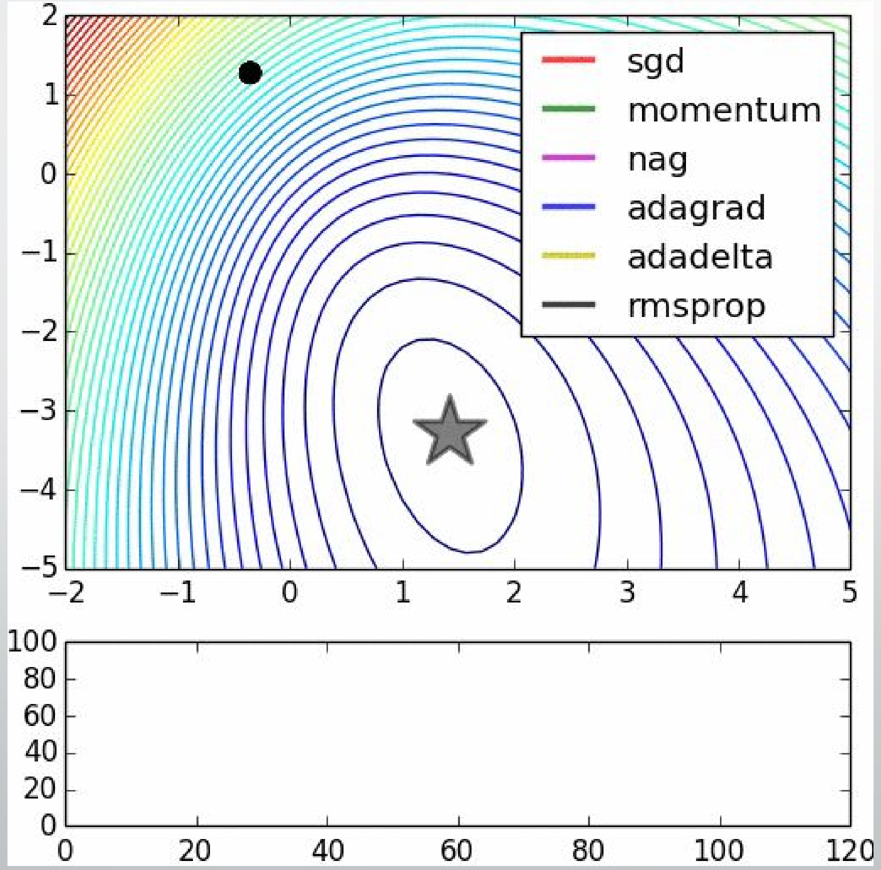
Solution 1: Add momentum

- Instead, allow velocity to accumulate:
- Should help move quickly down shallow slopes

Friction

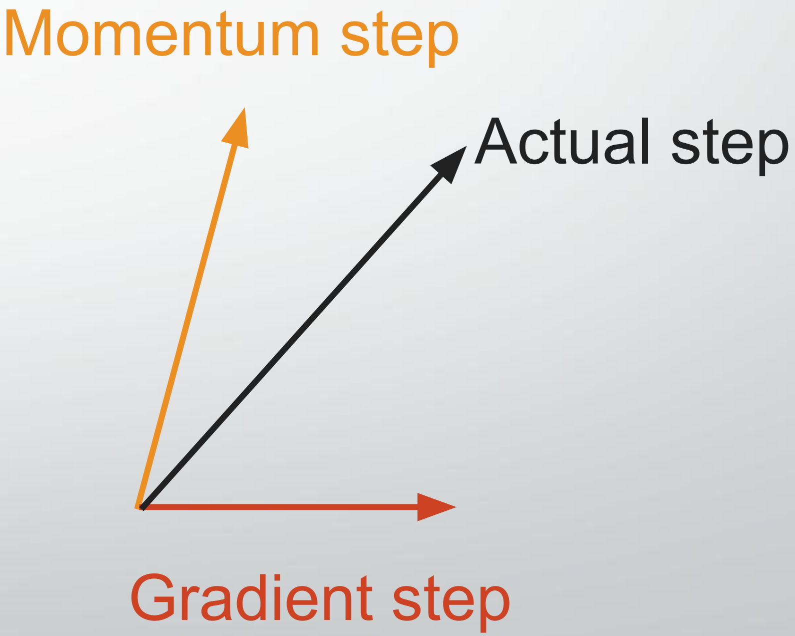
$$v_{t+1} = \boxed{\alpha} v_t - \mu \nabla \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

Solution 1: Add momentum



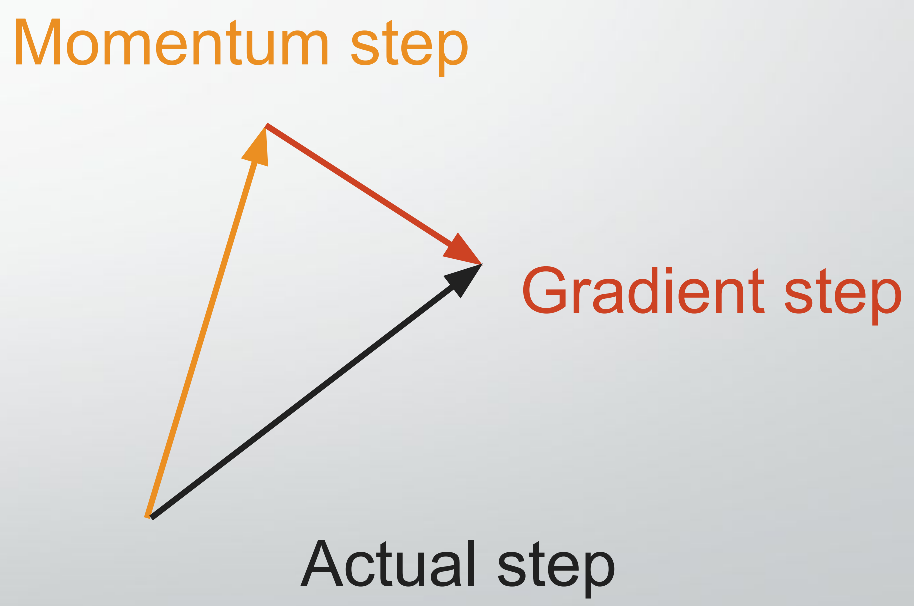
Solution 2: Make momentum 'smart'

- We saw a large speed up in convergence with momentum
- But the method also overshot the target
- The momentum update consists of a momentum step, and a gradient step



Solution 2: Make momentum 'smart'

- Since we know we'll make the momentum step
- Let's make it first before evaluating the gradient
- Then we'll be evaluating the gradient at the position after the momentum step



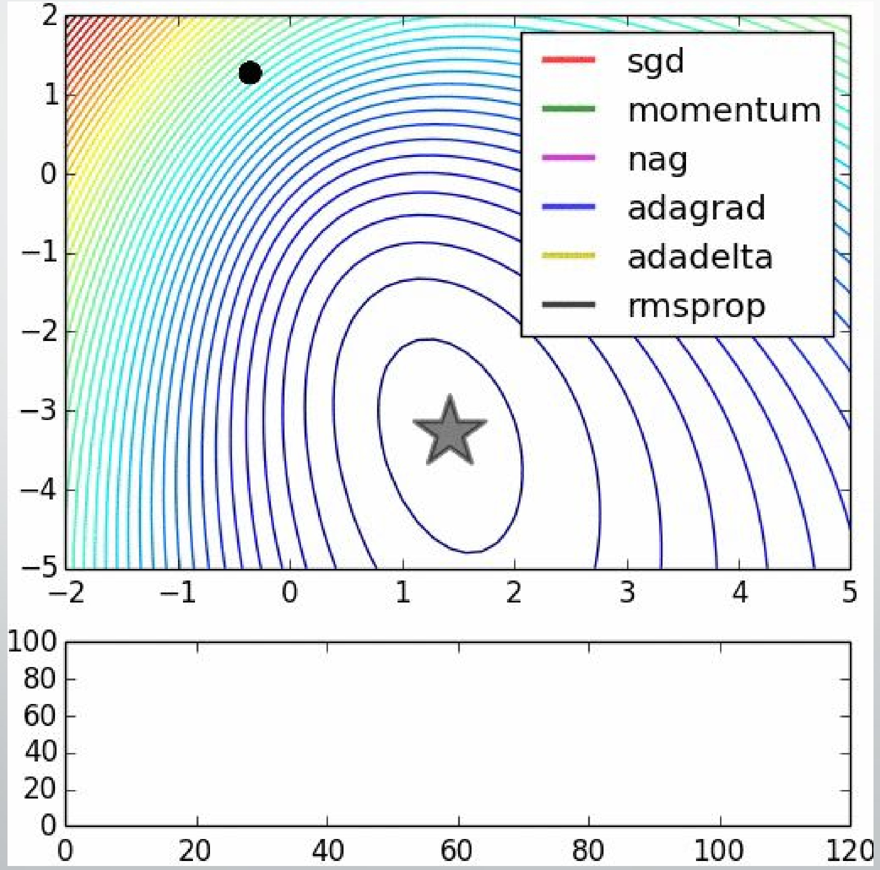
Solution 2: Make momentum 'smart'

- This one-step-lookahead allows for reduced overshooting
- Allows for quicker convergence
- Referred to as Nesterov momentum

$$v_{t+1} = \alpha v_t - \mu \boxed{\nabla \mathcal{L}(\theta_t + \alpha v_t)}$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

Evaluate gradient after momentum step

Solution 2: Make momentum 'smart'



Solution 3: Adapt the learning rate

- For steep gradients we want a small learning rate
- For shallow ones, a high learning rate
- Let's give each parameter its own learning rate
- And scale them according to past gradients
- ADAGRAD; Duchi, Hazan, and Singha 2011

$$\mu_{\theta_i} = \frac{\mu_0}{\sqrt{\boxed{\sum_{n=0}^{t} \nabla \mathcal{L}(\theta_{i,n})^2}}}$$

Square sum of past gradients

Solution 3: Adapt the learning rate

- Over time, the learning rate will drop to zero
- Not so good for deep networks
- Let's allow the store of past gradients to decay
- Effectively keeping a moving average of past gradients
- RMSProp; Hinton & Tieleman, 2012

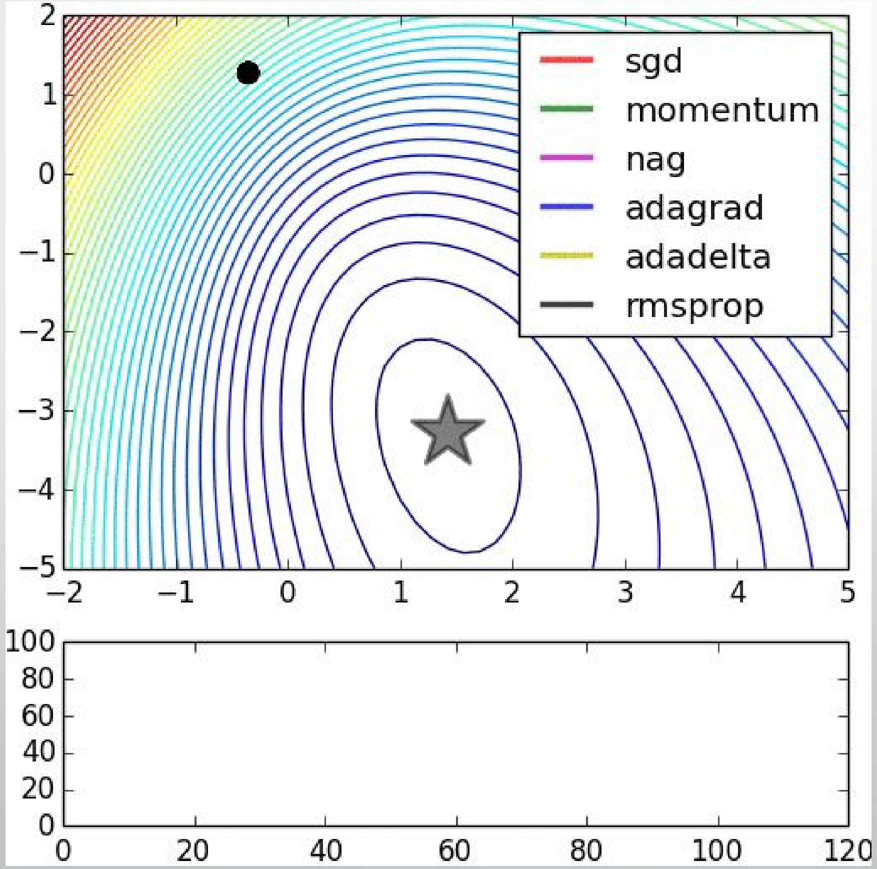
Decay rate

$$a_{i,t+1} = \lambda a_{i,t} + (1 - \boxed{\lambda}) \nabla \mathcal{L}(\theta_{i,t})^2$$

$$\mu_{i,t+1} = \frac{\mu_0}{\sqrt{\boxed{a_{i,t+1}}}}$$

Leaking store of past gradients

Solution 3: Adapt the learning rate



Final step: Combine them

- Both methods of adding Nesterov momentum and adapting the learning rate are seen to offer improvements
- No reason why they can't be combined
- This is called NADAM; Dozat 2015

Improvements - Batch normalisation

- Initialisation methods assume unit-Gaussian inputs
- Sometimes this is not the case: data isn't pre-processed, signals become non-Gaussian
- Means that the initialisation isn't always optimal

Improvements - Batch normalisation

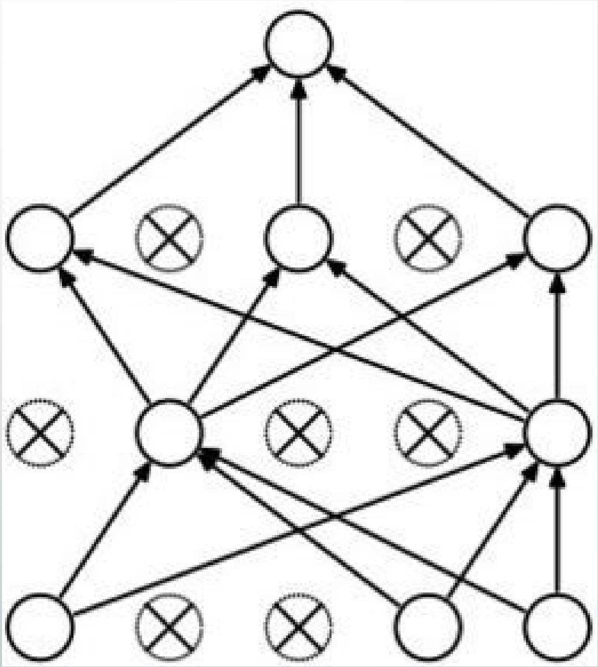
- But if you want unit-Gaussian inputs, then just make them unit-Gaussian!
- Adding a *batch-normalisation* layer on the inputs of a neuron layer will transform signals into $\mathcal{N}(0, 1)$
- Transformation adjusts per *batch* of data
- Batch normalisation; Ioffe and Szegede, 2015
- Leads to much quicker convergence

Improvements - Ensembling

- A single model is unlikely to be optimal for all possible inputs
- By training multiple copies of the same model
- Then combining their predictions
- The ensembled model is likely to be more performant in a wider range of input regions
- Effectively a guaranteed improvement!
- Can experiment with different weighting schemes, combinations of architectures, ML algorithms, *et cetera*

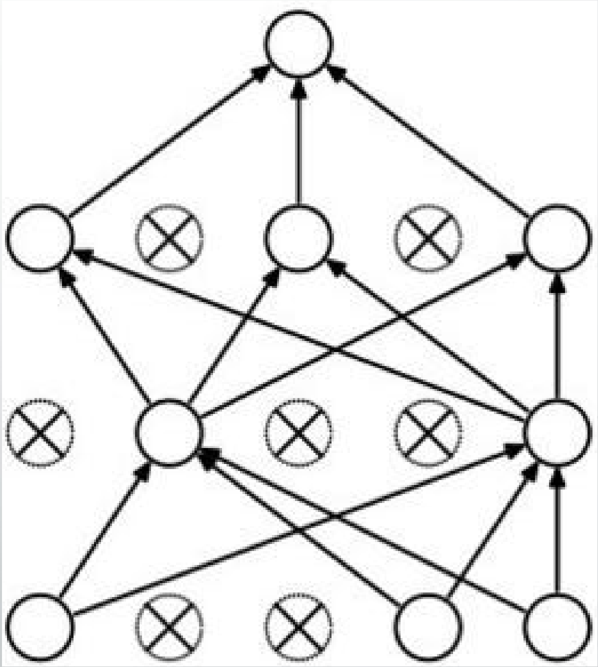
Improvements - Dropout

- Slightly counter-intuitive
- Involves randomly dropping (masking) neurons per training iteration
- Means that during that iteration, the dropped neurons are never used
- Hinton et al, 2014



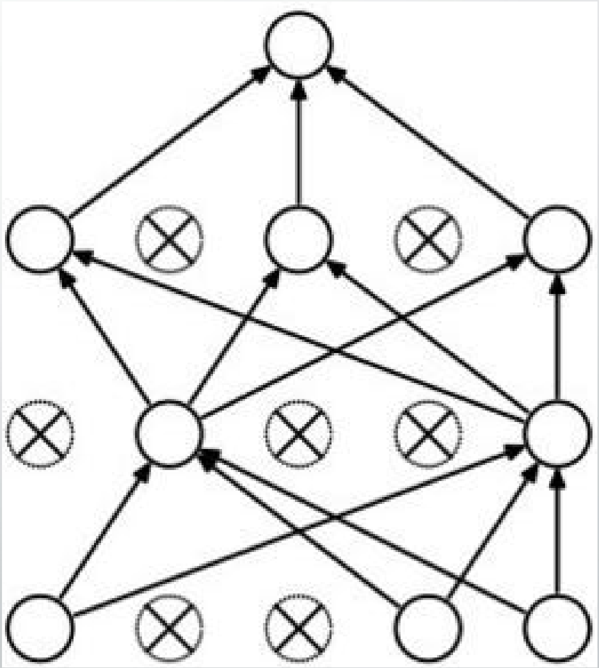
Improvements - Dropout

- Prevents the network from becoming over reliant on certain inputs
- Forces it to generalise to the data
- Effectively trains many sub-networks, i.e. internal ensembling
- Speeds up training (few things to evaluate)



Improvements - Dropout

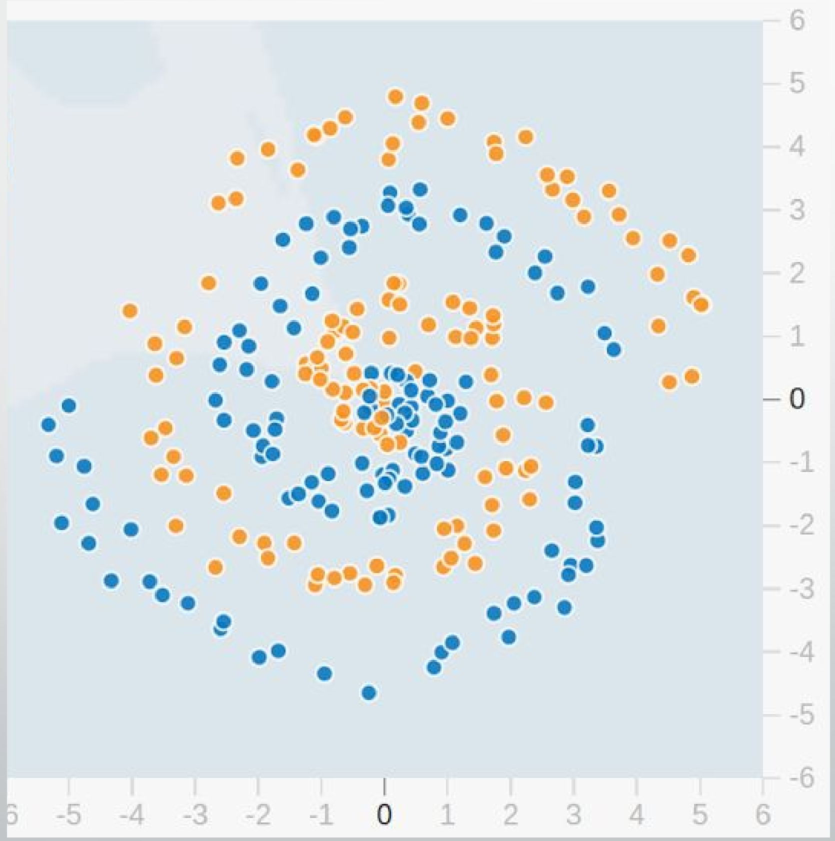
- One subtlety:
- During training perhaps only half the network is used
- During application, all the network is used
- Need to scale outputs during training to maintain similar levels of activation in each regime



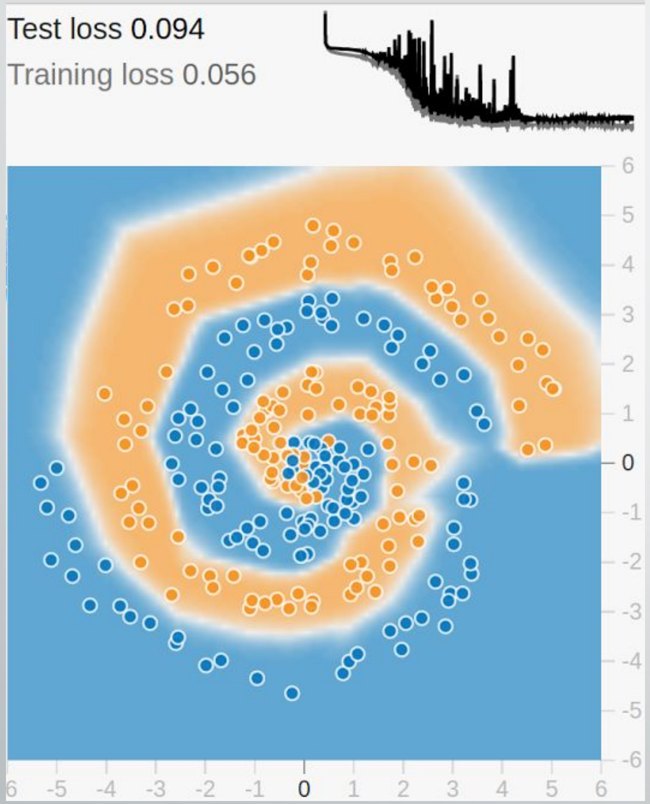
Advantages over other Machine Learning methods

- Direct access to nonlinear responses
- Many previous ML methods have a linear response
- Ensembling them (e.g. random forest; an ensemble of decision trees) could allow for non-linear fitting
- By using a nonlinear activation function, NNs can directly apply nonlinear fitting

Advantages over other Machine Learning methods

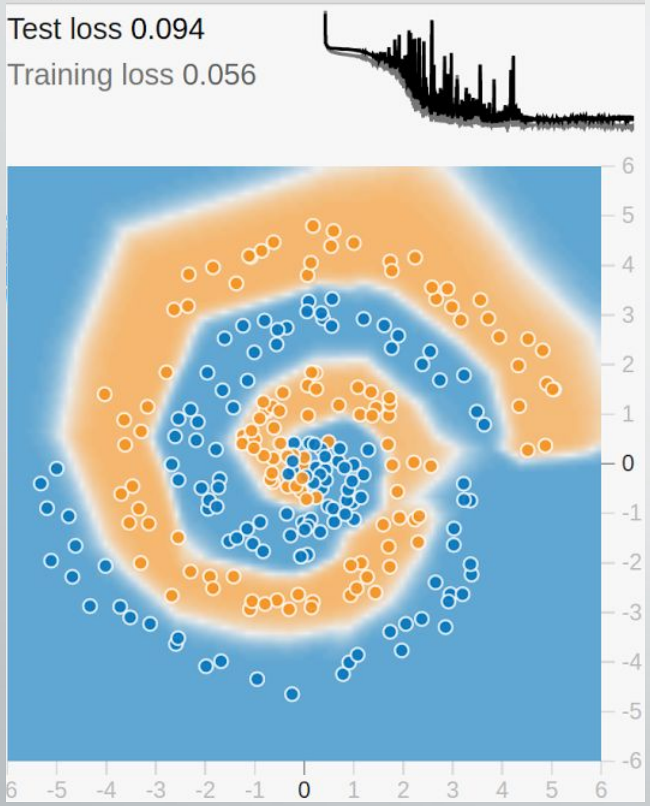


Advantages over other Machine Learning methods

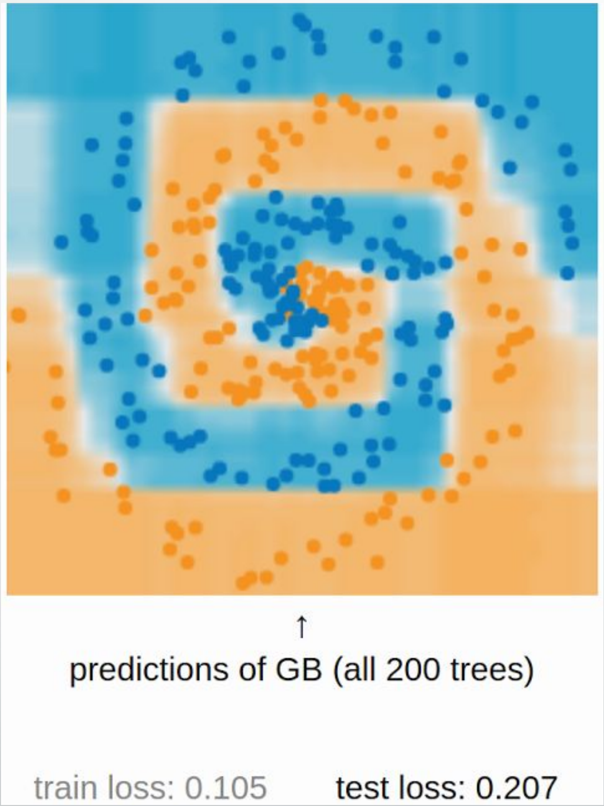


NN

Advantages over other Machine Learning methods

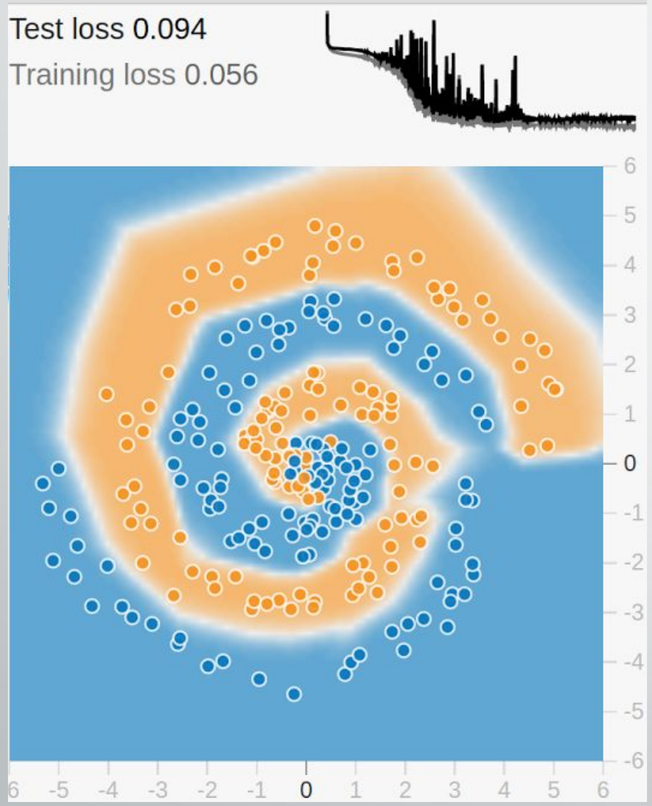


NN

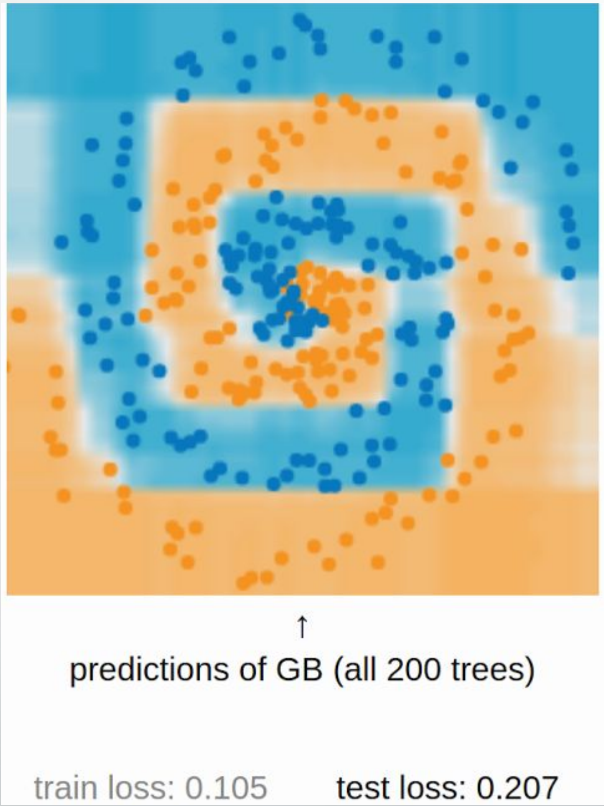


BDT

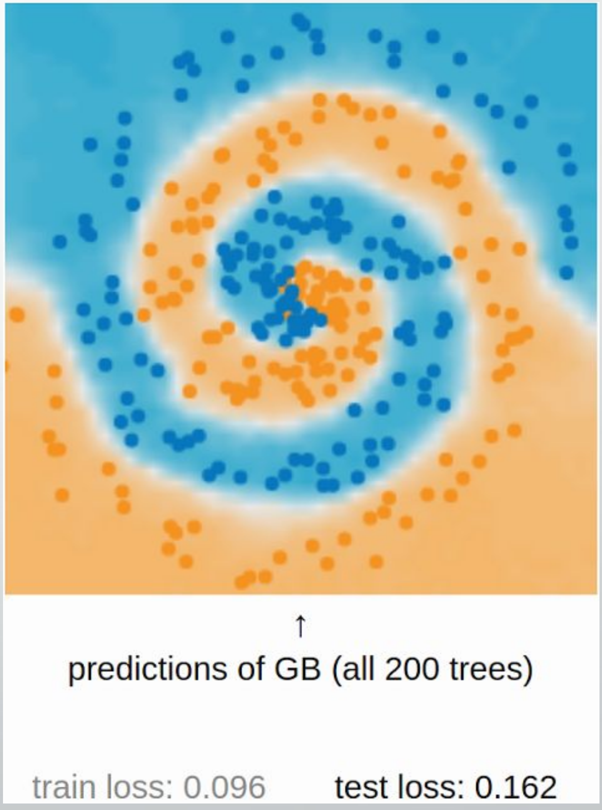
Advantages over other Machine Learning methods



NN

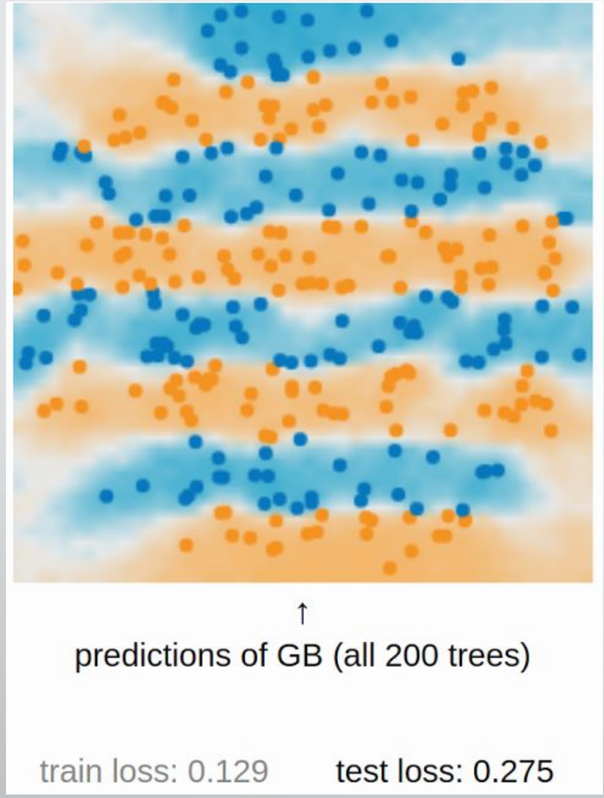


BDT

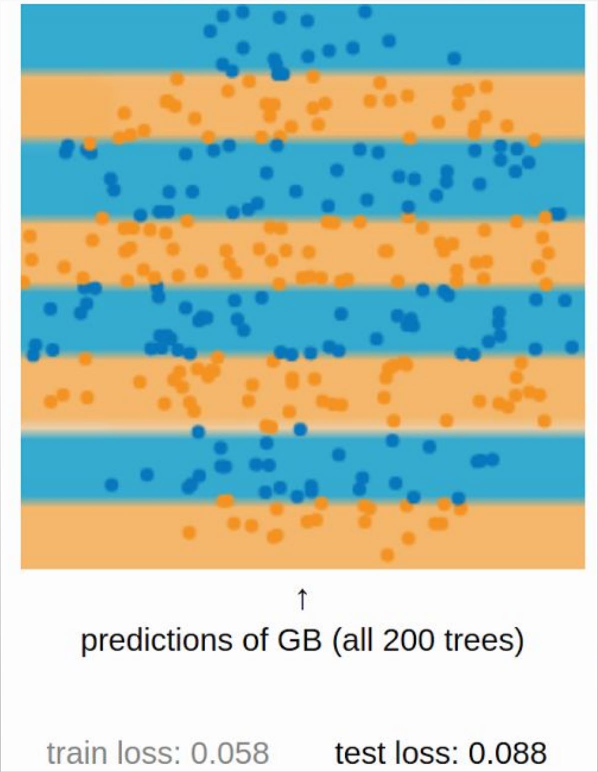


BDT+rotation

Advantages over other Machine Learning methods



BDT+rotation

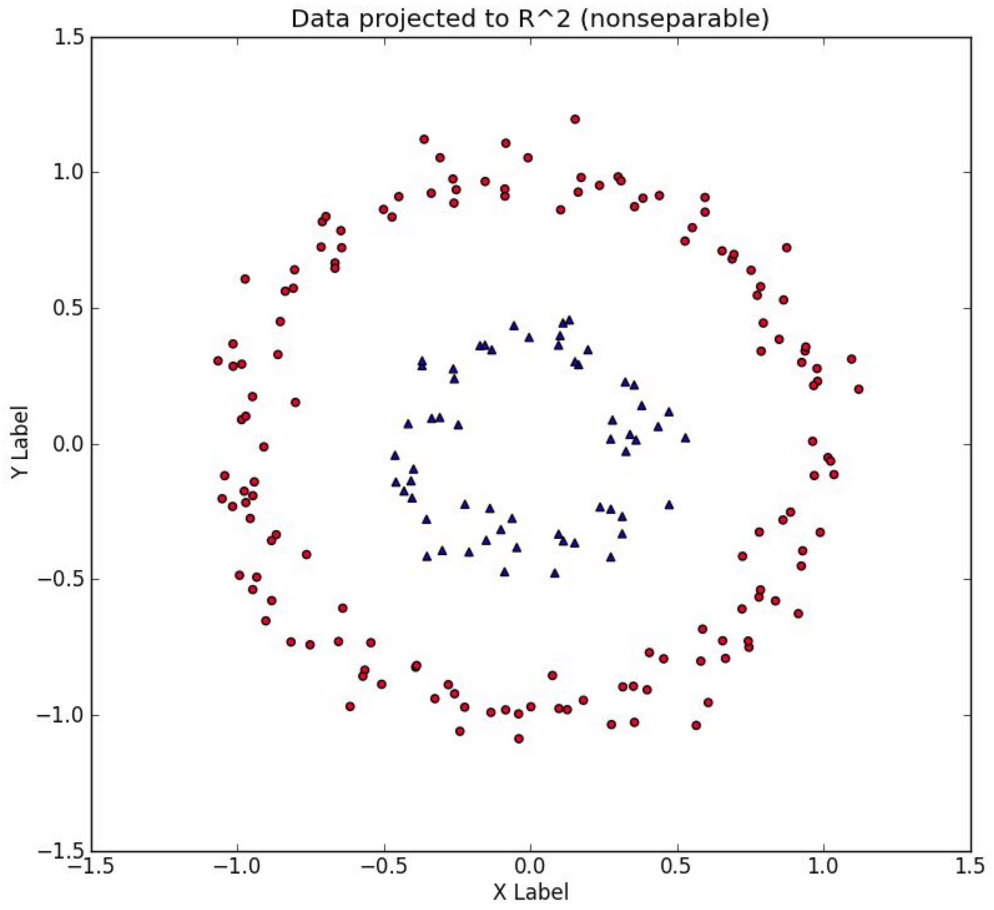


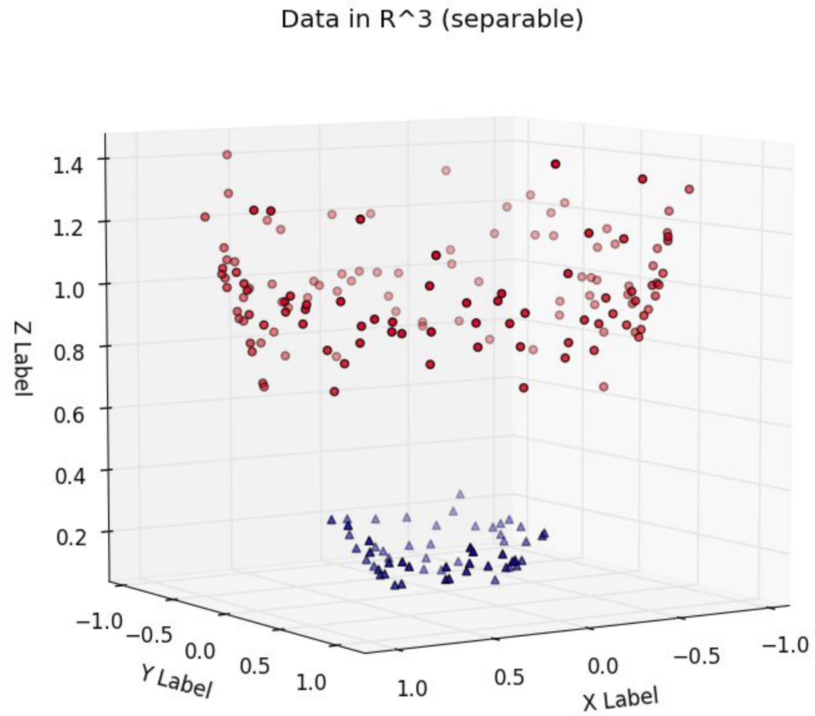
BDT

Automated learning of powerful features

- Power of linear classifiers relies on the ‘kernel trick’
- The application of a kernel function which warps feature-space to make data classes be linearly separable

Automated learning of powerful features

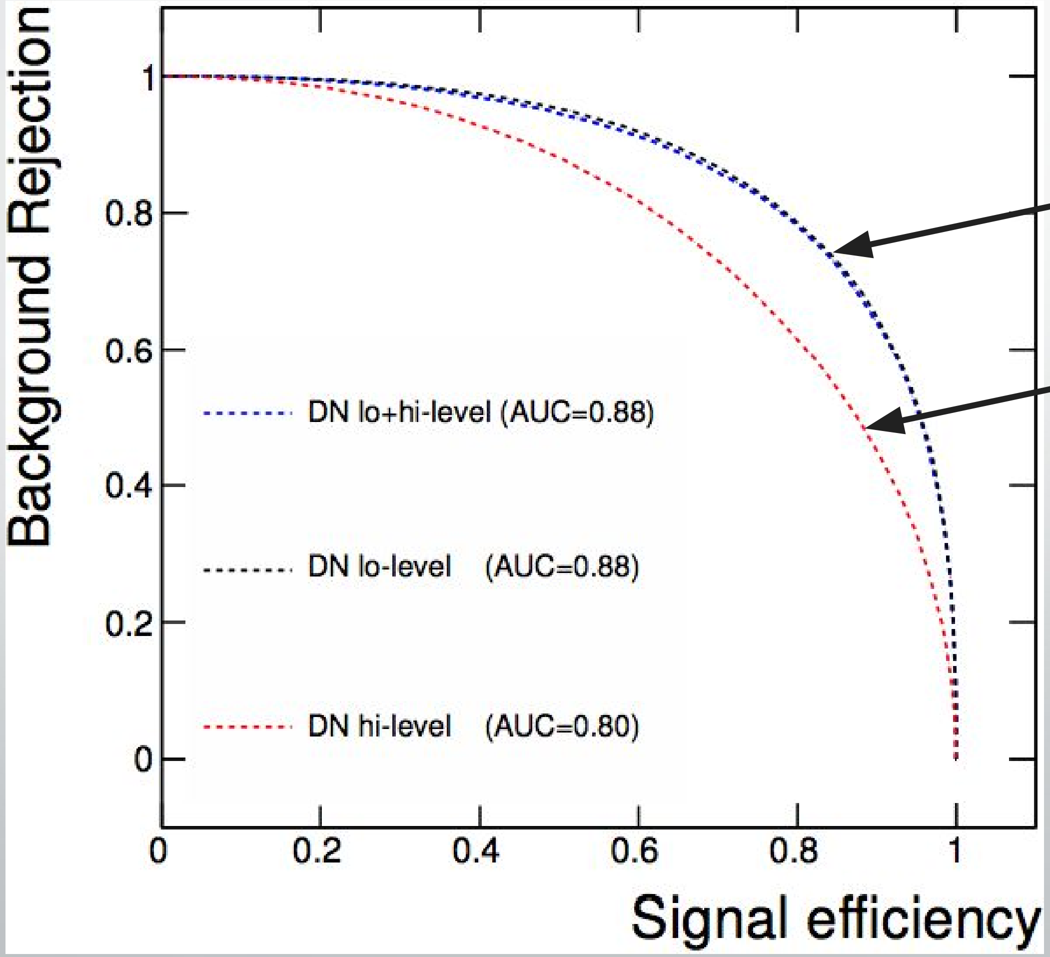




Automated learning of powerful features

- HEP example might be the invariant mass of a resonance
- High-level features which are nonlinear combinations of other features
- For other methods, are best calculated by hand and fed in; feature engineering
- High reliance on *domain knowledge*

Automated learning of powerful features



No improvement when HL features are added

DNN learns its own discriminant features

Whitson, 2015

Summary

- Neural networks are powerful implementations of Machine Learning
- Are able to make use of high-dimensional patterns in data
- Reduced feature engineering
- Must be built with care

Cheat sheet

- Activate with ReLu
- Initialise with He
- Optimise with NADAM
- Use batch-norm in front of every layer
- Try with dropout
- Use an ensemble of NNs
- Don’t withhold low-level information