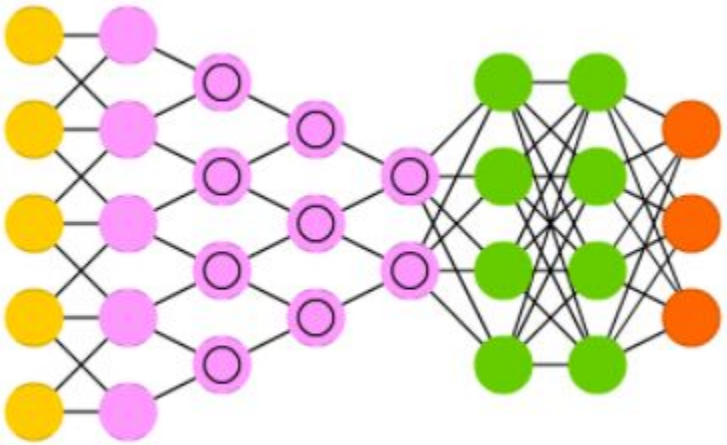
An intro to Neural Networks



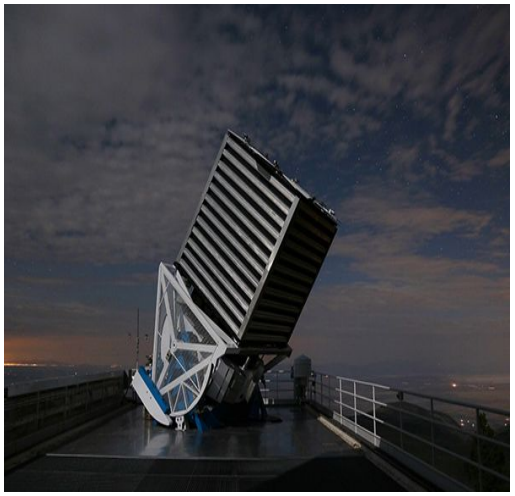
(1)

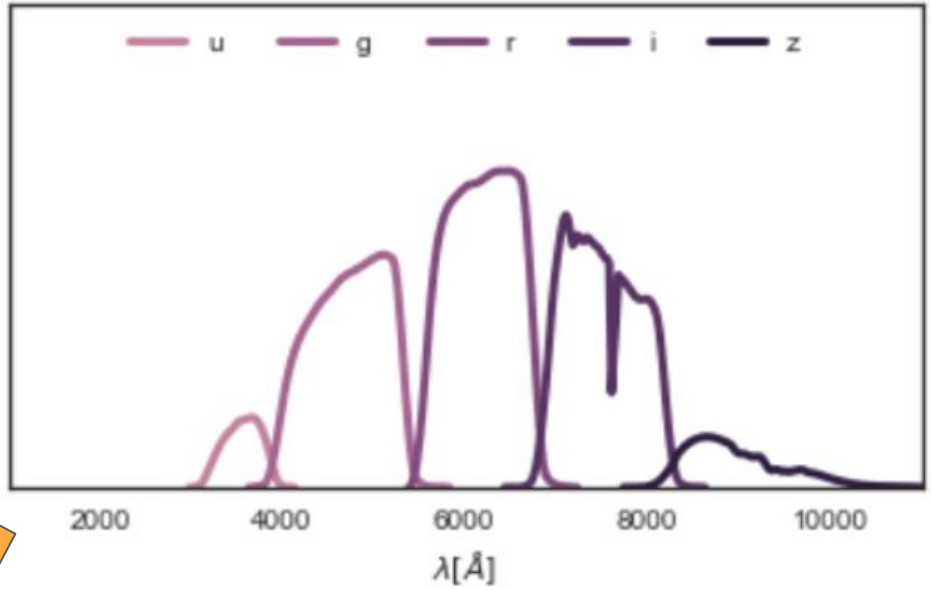
(1) A Graphic demonstrating Deep convolutional Neural nets taken from the Neural Networks Zoo by Fjodor van Veen

Let's start with a simple problem

We have a dataset with a set of features
(*photometric colors*)
And labels
(*quasar or star*)

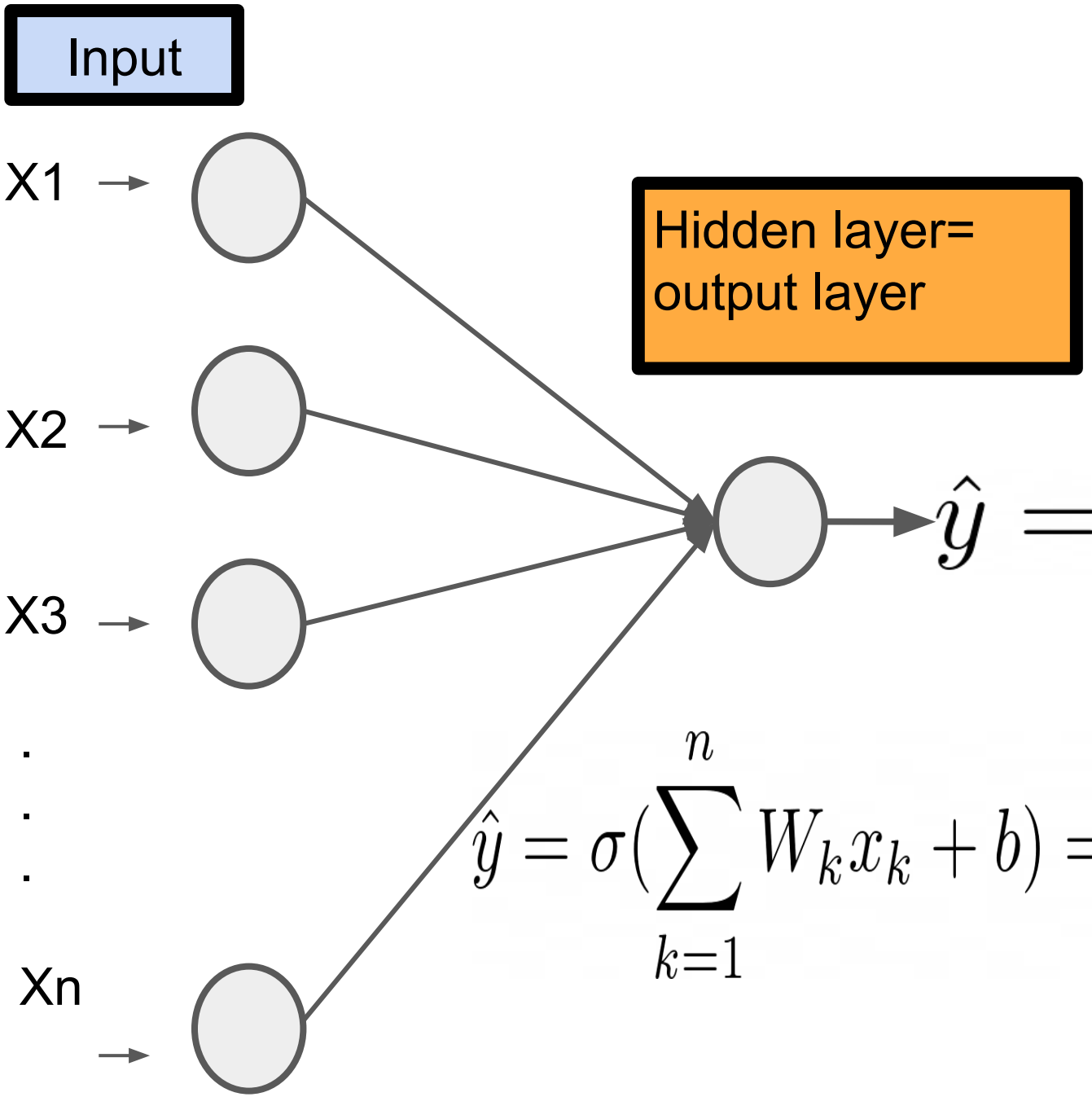
We want to predict the label of an unlabeled example given its features



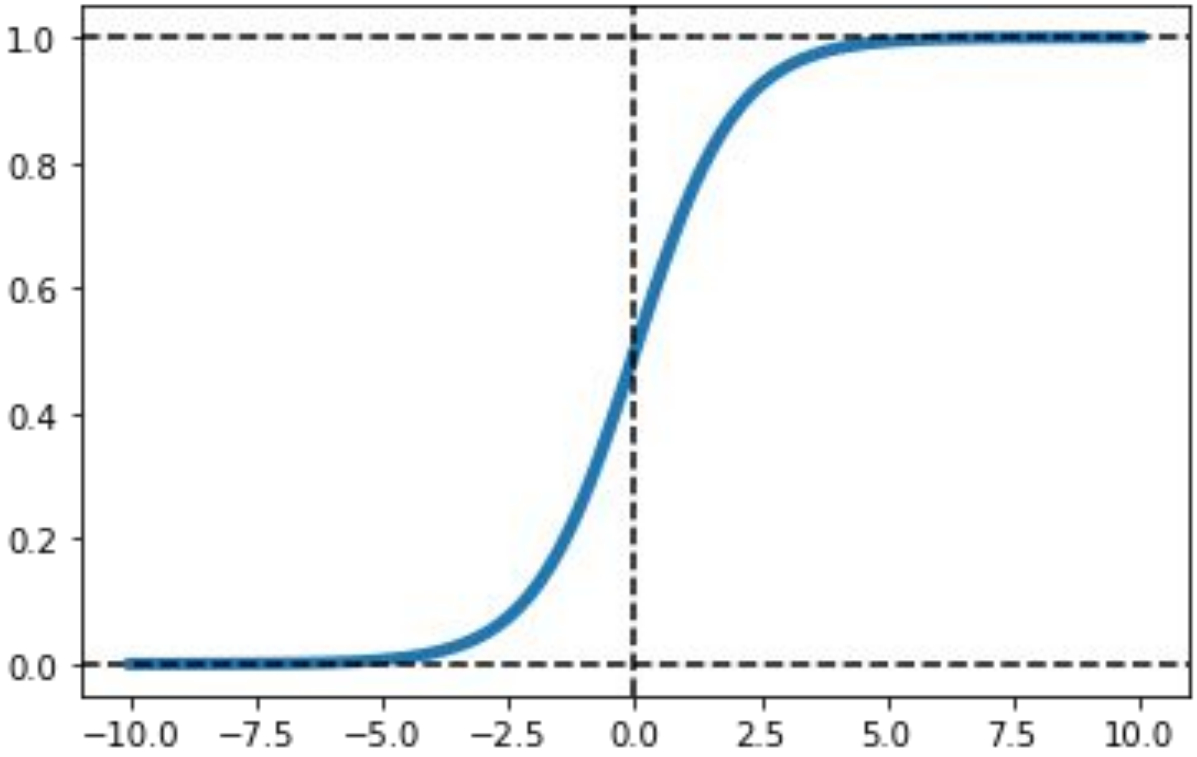


u	g	r	i	z

Label: Star or Quasar?

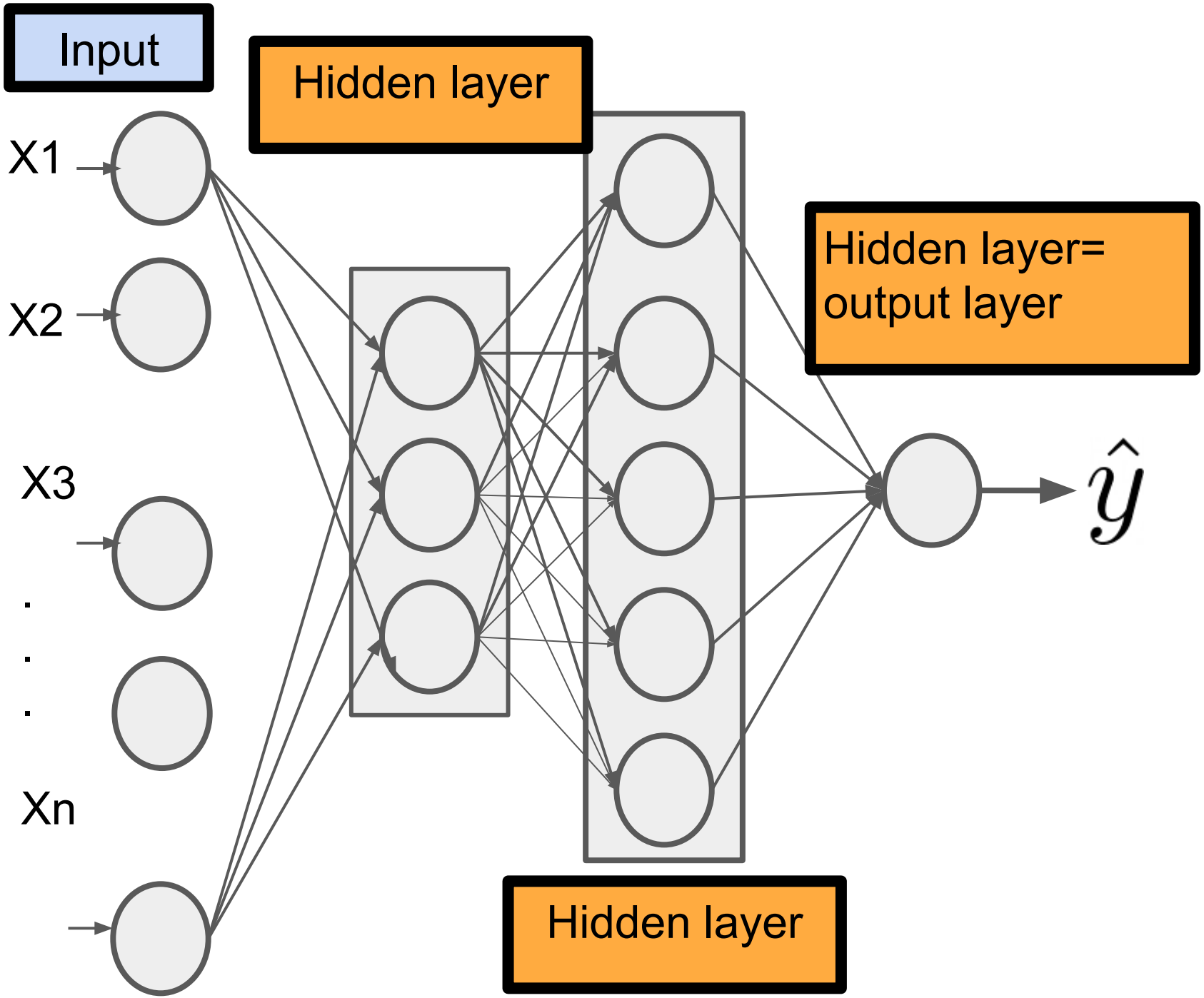

Input
X1
X2
X3
Xn
Hidden layer=
output layer
$\hat{y} = p(C_1|x)$
$\hat{y} = \sigma(\sum_{k=1}^{n} W_k x_k + b) = \sigma(W^T x + b)$

Remember the sigmoid activation function from the previous lecture



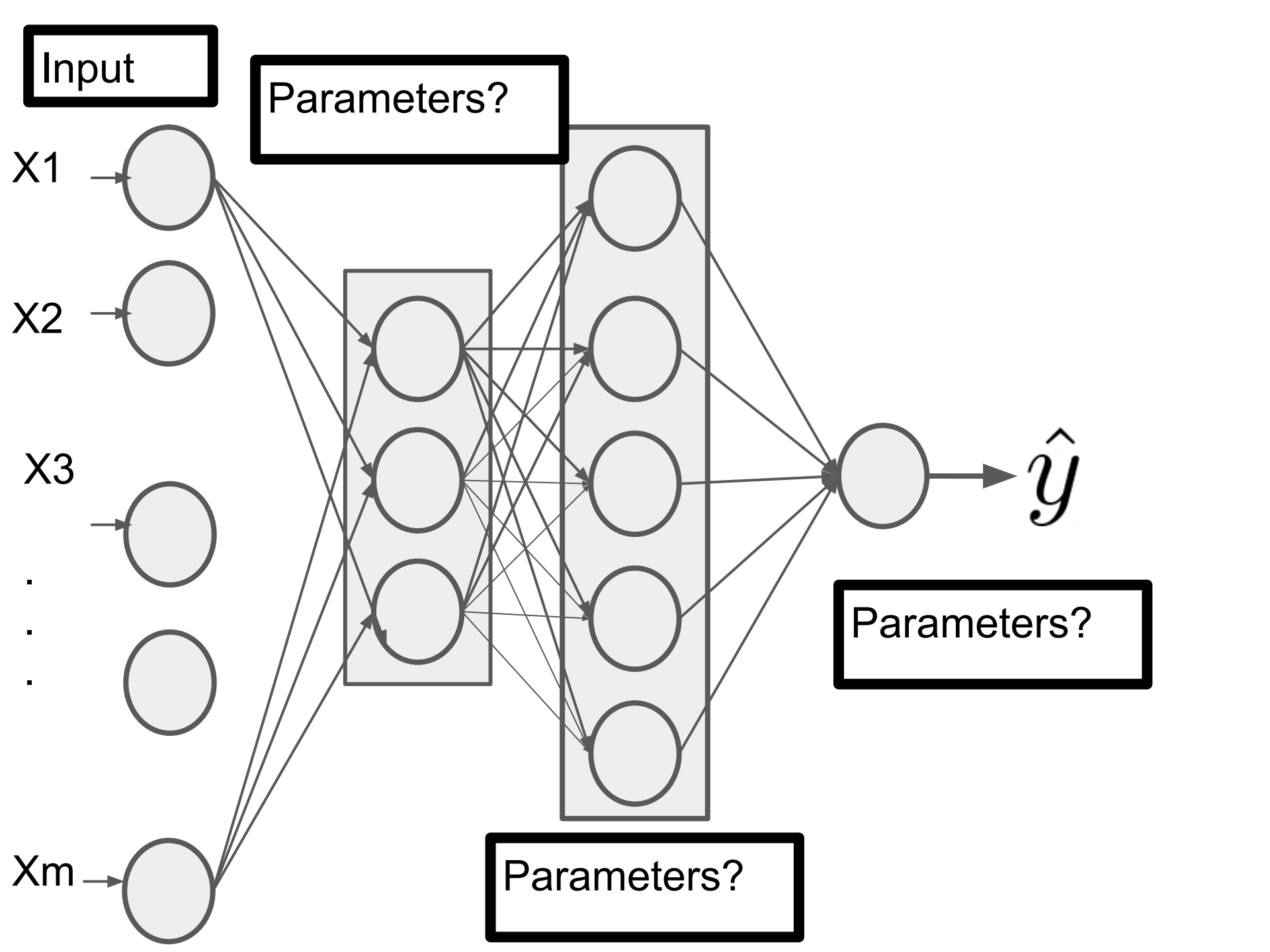
Now let’s go deeper and consider a multi-layer perceptron (MLP)

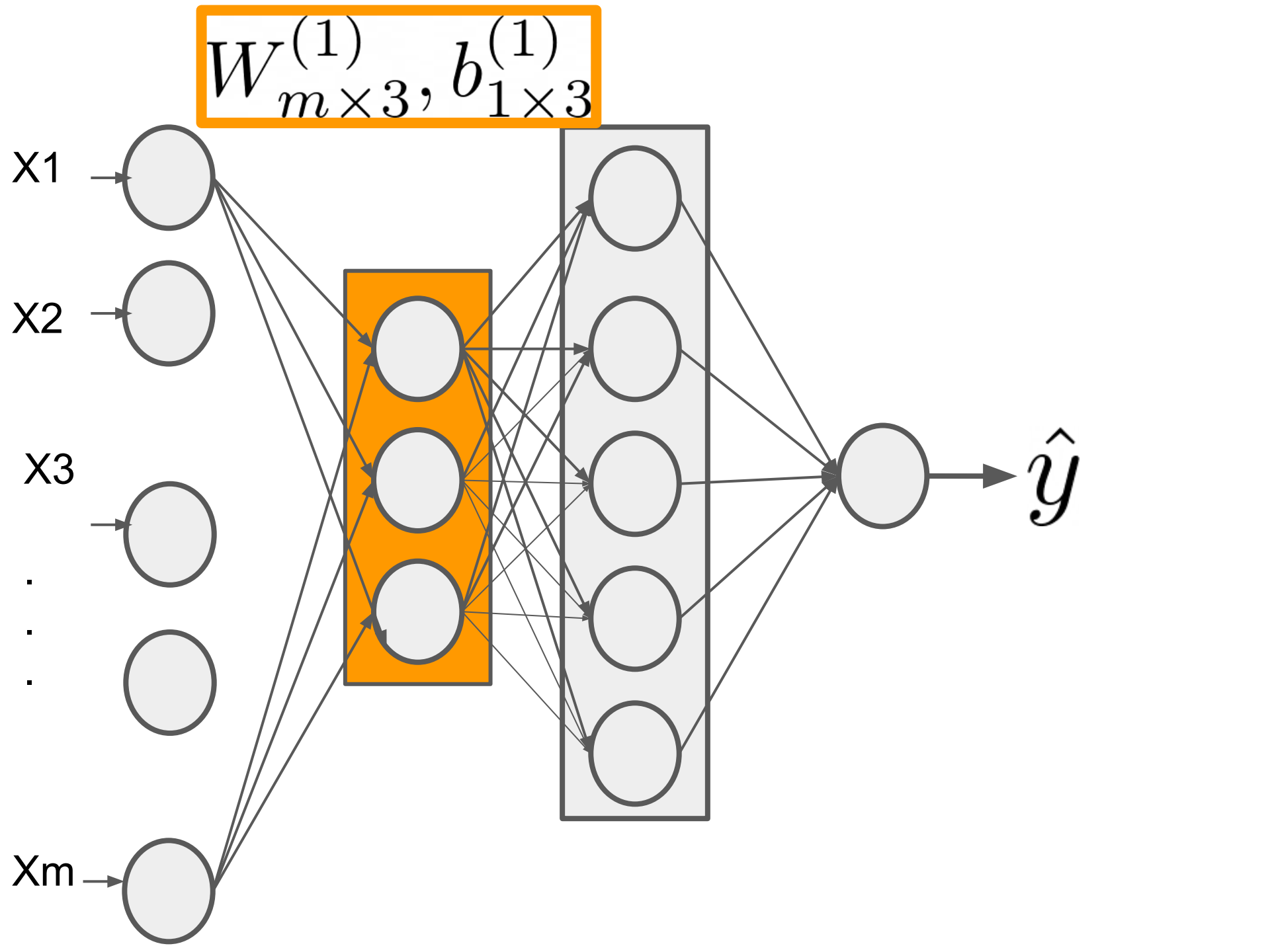
P.s. depending on where you look at, there are other names for this in the literature as well

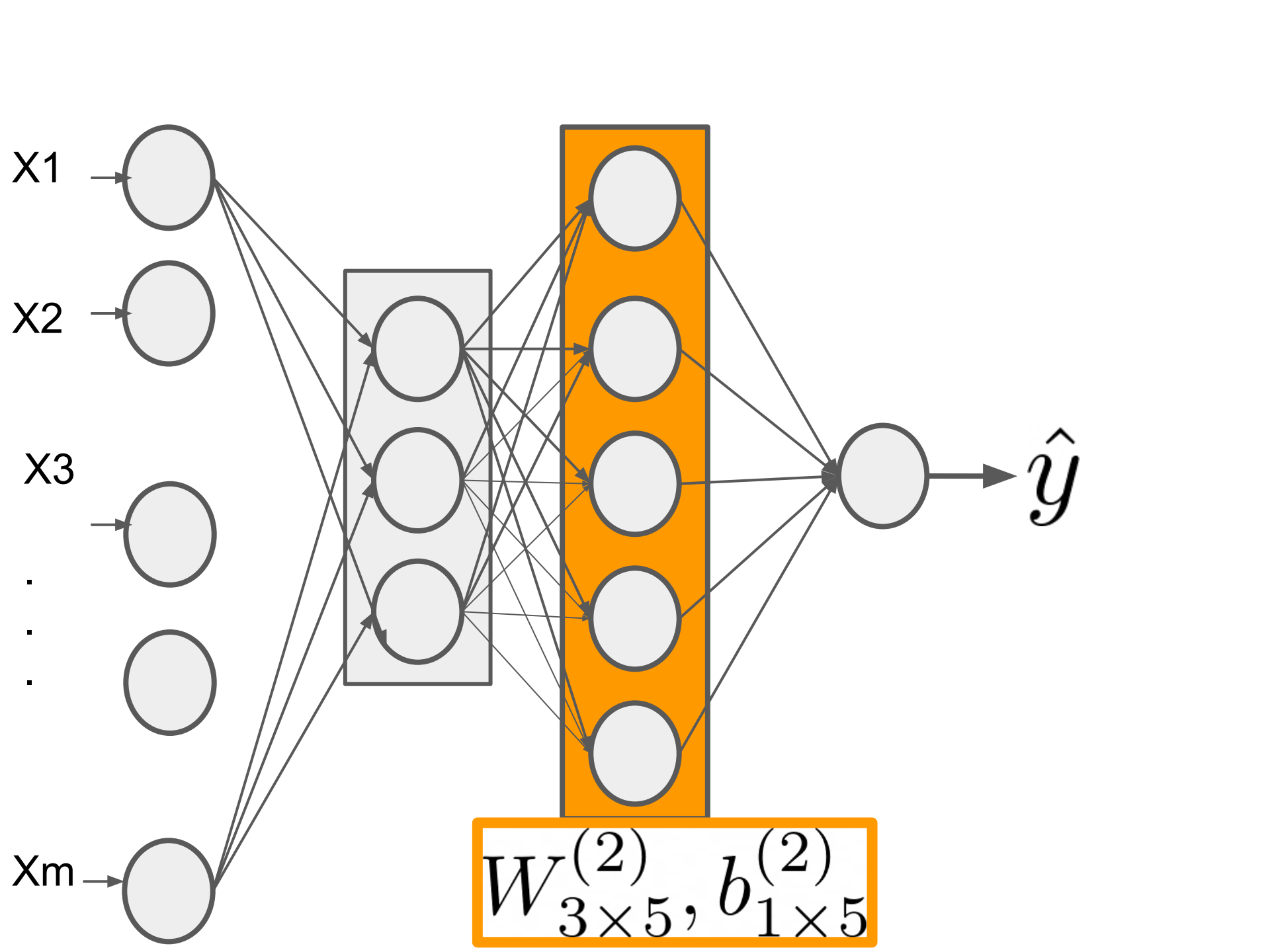

Input
Hidden layer
X1
X2
X3
Xn
Hidden layer= output layer
$\hat{y}$
Hidden layer

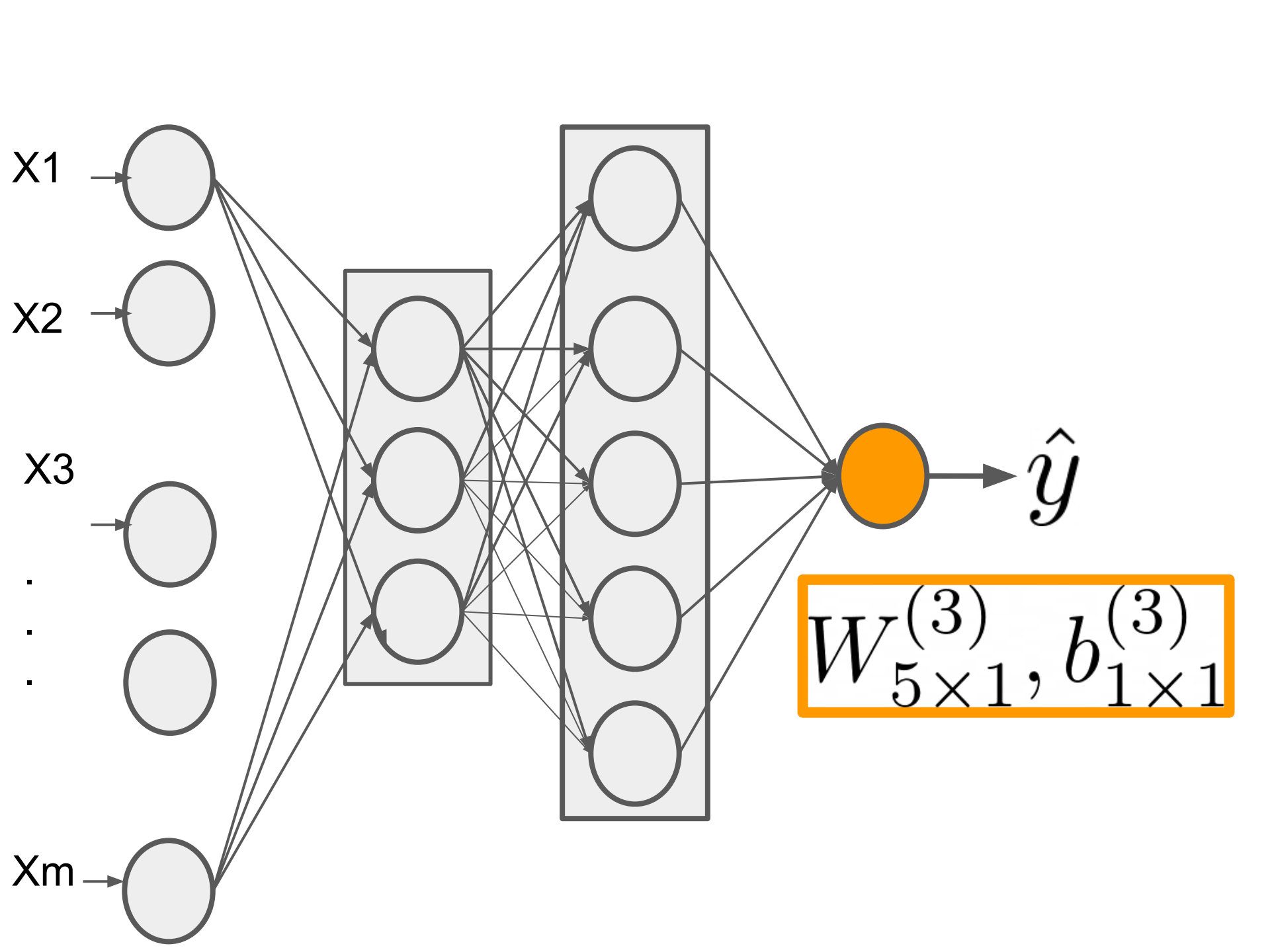
Quiz: What are the free parameter of this model?

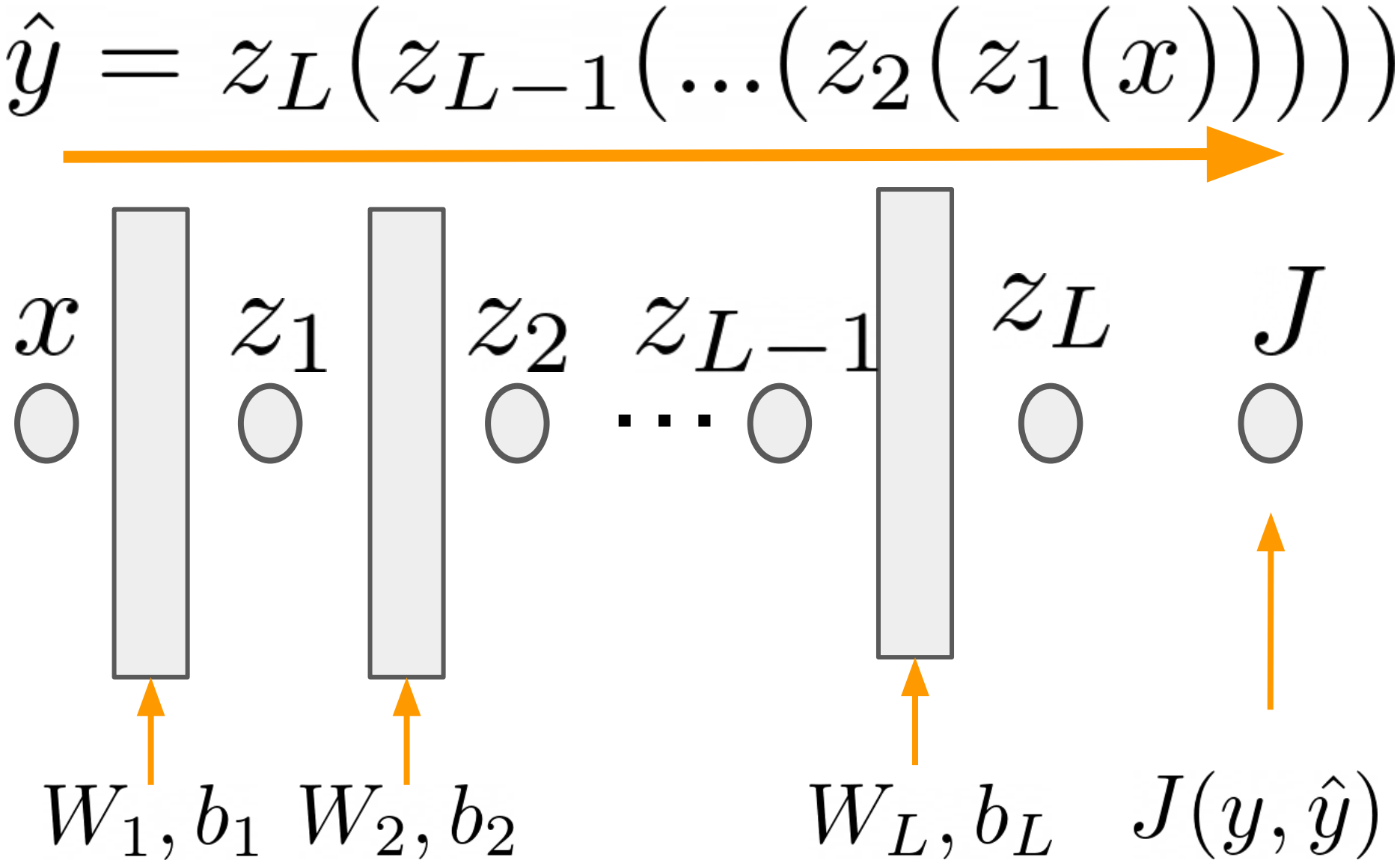
Reminder: the mathematical operation in each single neuron can be summarized as:
Multiplication ⇒ Addition ⇒ Nonlinear activation function


Input
Parameters?
X1
X2
X3
Xm
$\hat{y}$
Parameters?
Parameters?

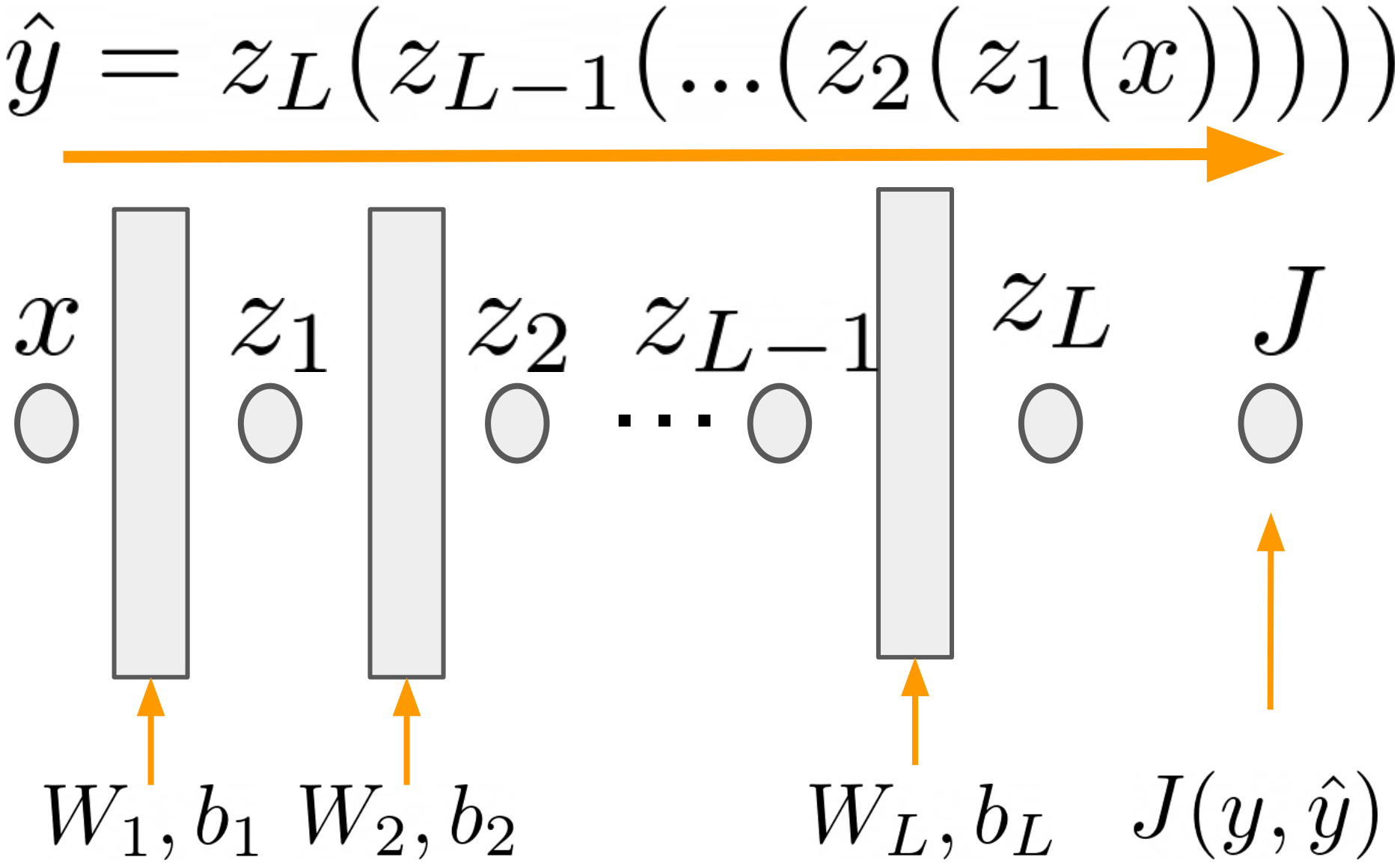

$W^{(1)}_{m\times 3}, b^{(1)}_{1\times 3}$
X1
X2
X3
.
.
.
Xm
$\hat{y}$


X1
X2
X3
Xm
$\hat{y}$
$W^{(2)}_{3\times5}, b^{(2)}_{1\times5}$


X1
X2
X3
Xm
$\hat{y}$
$W^{(3)}_{5\times1}, b^{(3)}_{1\times1}$


$\hat{y} = z_L(z_{L-1}(\ldots(z_2(z_1(x)))))$
x
z_1
z_2
z_{L-1}
z_L
J
W_1, b_1
W_2, b_2
W_L, b_L
$J(y, \hat{y})$

The parameters are found by optimizing the loss function J


$\hat{y} = z_L(z_{L-1}(\ldots(z_2(z_1(x)))))$
x
z_1
z_2
z_{L-1}
z_L
J
W_1, b_1
W_2, b_2
W_L, b_L
$J(y, \hat{y})$

Question: How are the derivatives of the cost function with respect to the model parameters computed?

Answer: with the ***Chain Rule***!

$$\frac{\partial J}{\partial \theta_{L-1}} = \frac{\partial J}{\partial z_L} \cdot \frac{\partial z_L}{\partial \theta_{L-1}}$$

$$\frac{\partial J}{\partial \theta_{L-2}} = \frac{\partial J}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \frac{\partial z_{L-1}}{\partial \theta_{L-2}}$$

$$\vdots$$

$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \cdots \frac{\partial z_{l+2}}{\partial z_{l+1}} \frac{\partial z_{l+1}}{\partial \theta_l}$$

This procedure is called:

The BackPropagation!

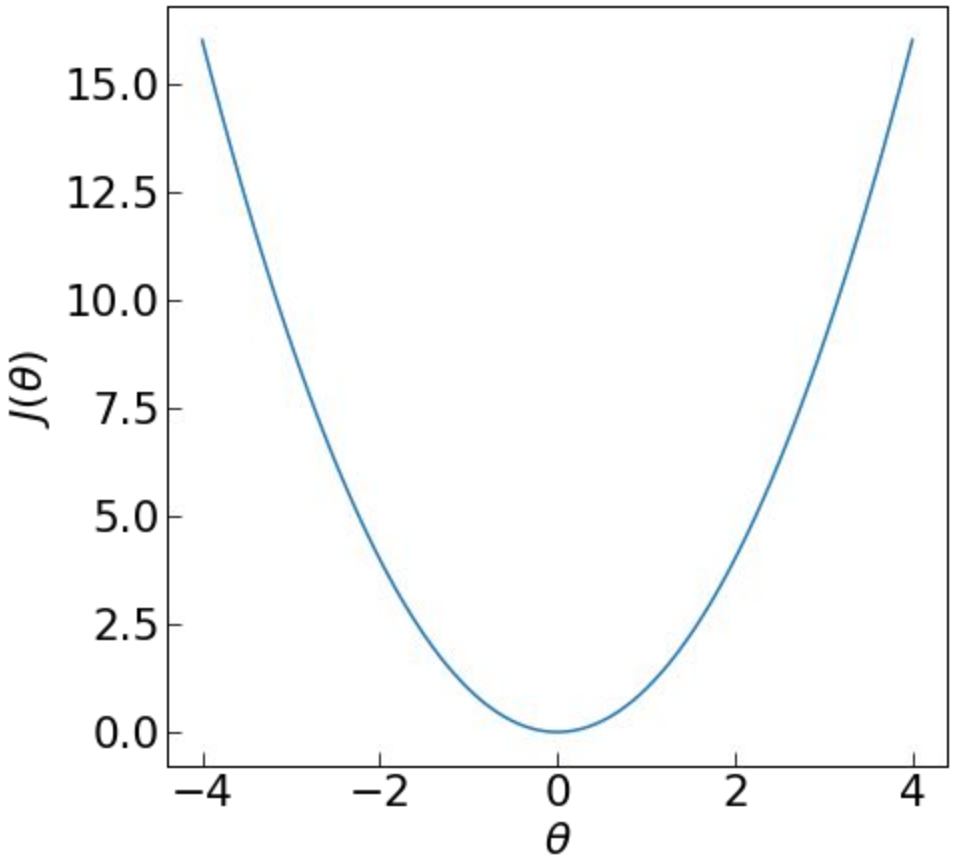
Once we have the derivatives we can use the ***Gradient descent*** algorithms to optimize the cost function

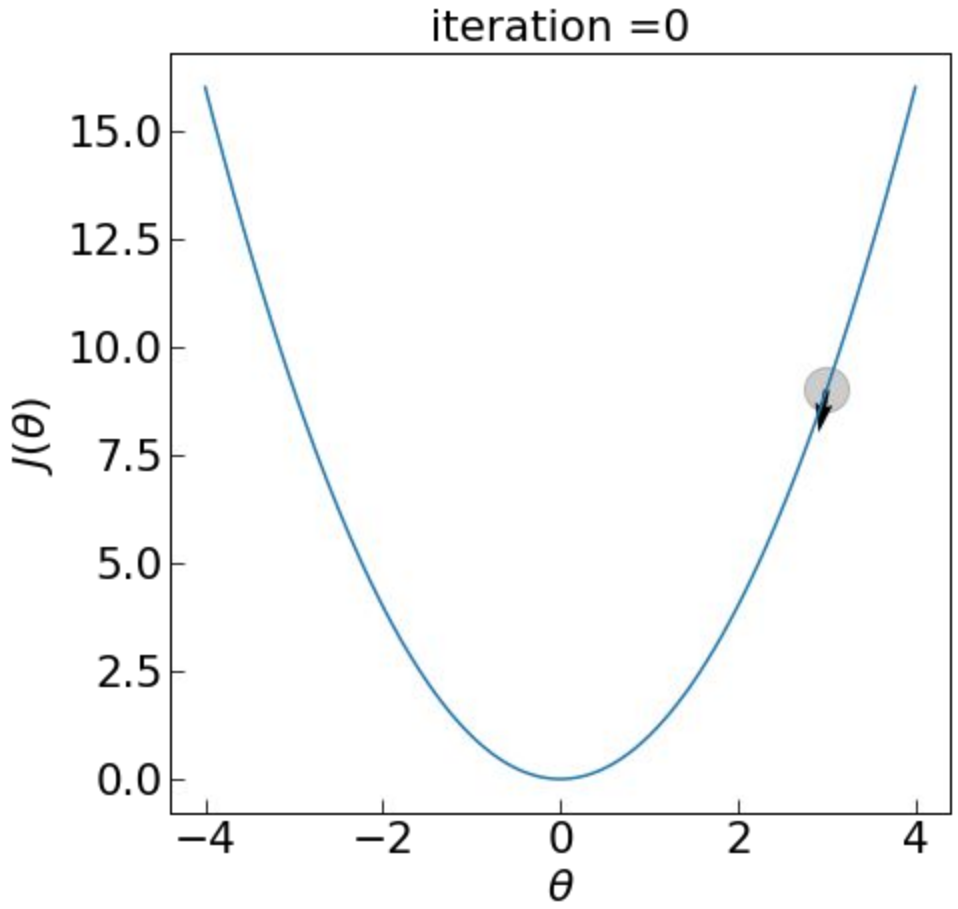
Parameter update in neural nets

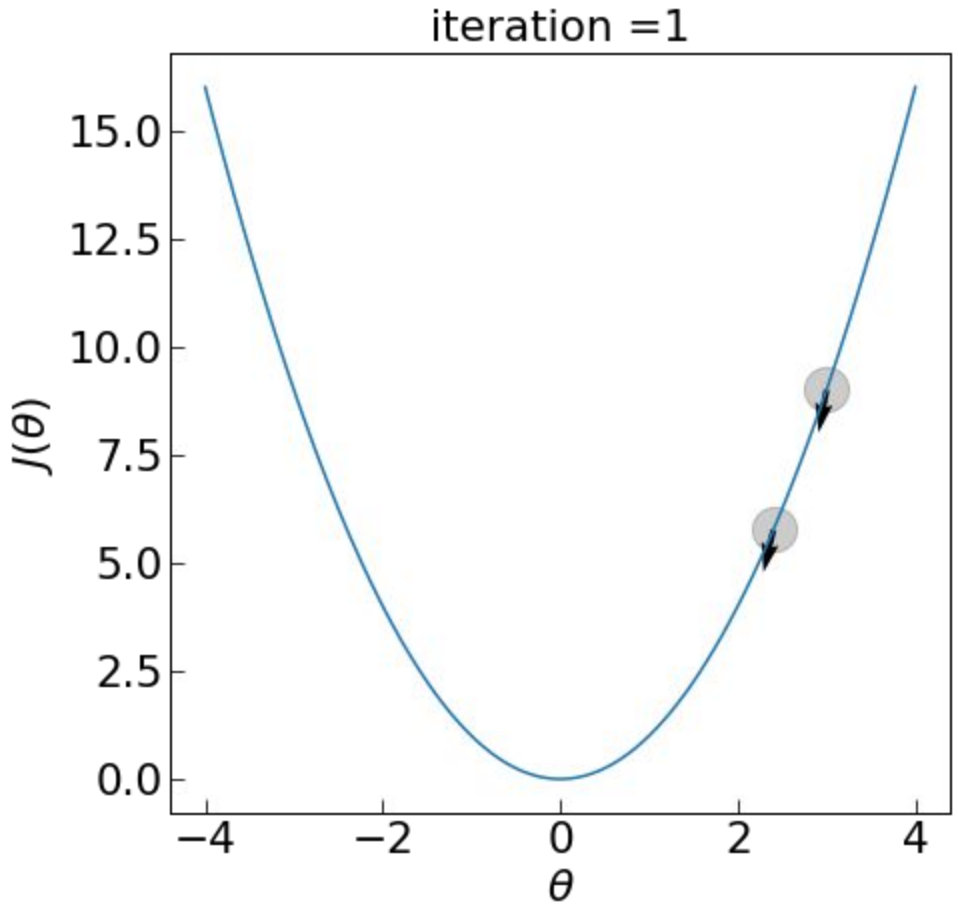
$$\theta(t+1) = \theta(t) - \underbrace{lr}_{\text{learning rate}} \frac{\partial J}{\partial \theta(t)}$$

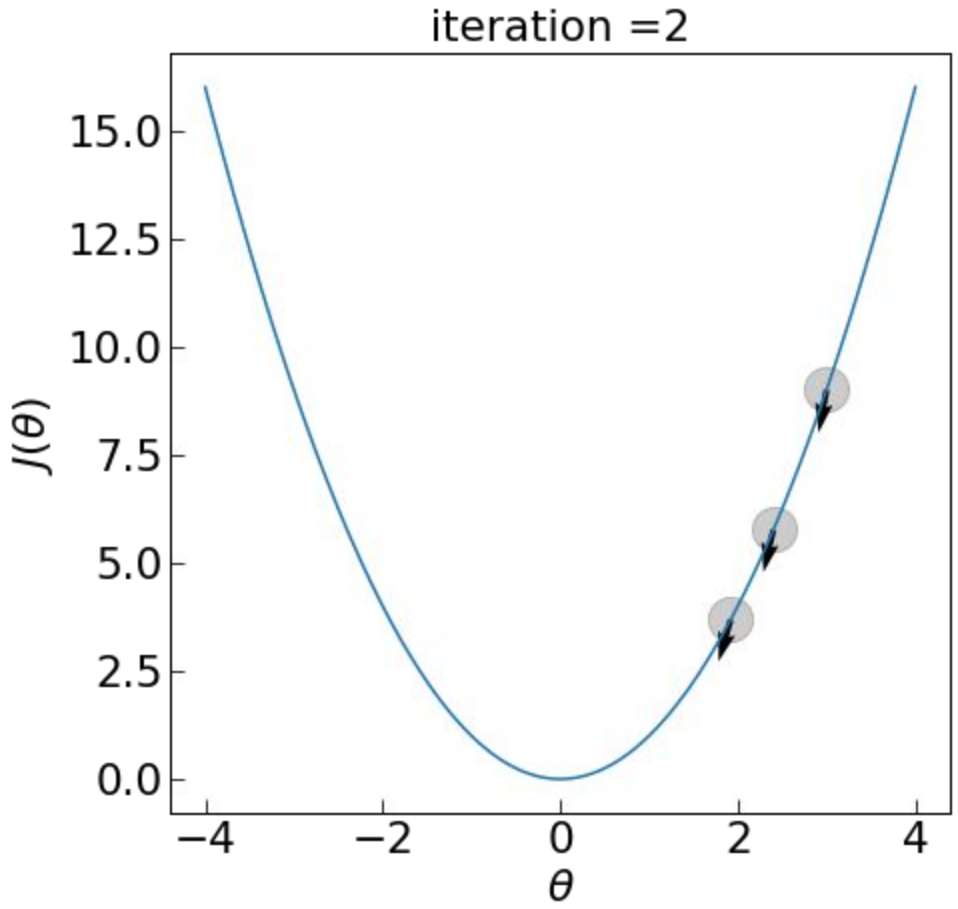
How do we set the learning rate?

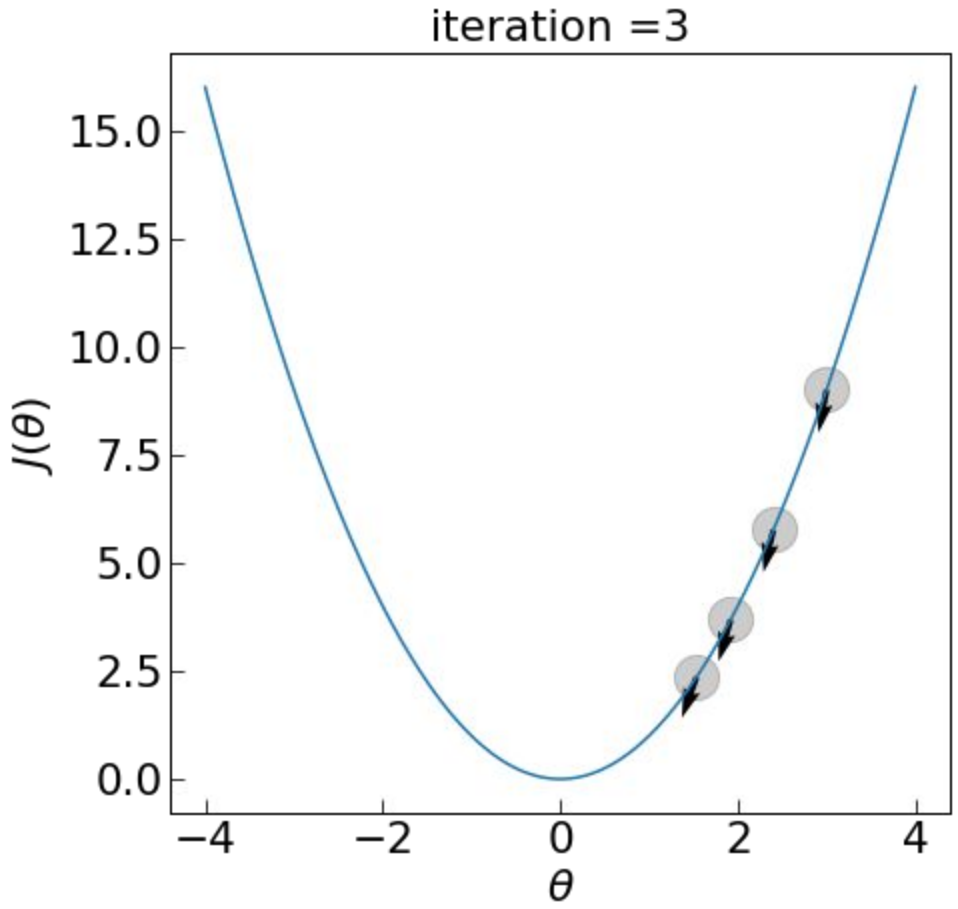
How does ***Gradient descent*** work?


15.0
12.5
10.0
7.5
5.0
2.5
0.0
$J(\theta)$
−4
−2
0
2
4
θ


iteration =0
15.0
12.5
10.0
7.5
5.0
2.5
0.0
$J(\theta)$
−4
−2
0
2
4
θ


iteration =1
15.0
12.5
10.0
7.5
5.0
2.5
0.0
$J(\theta)$
−4
−2
0
2
4
θ

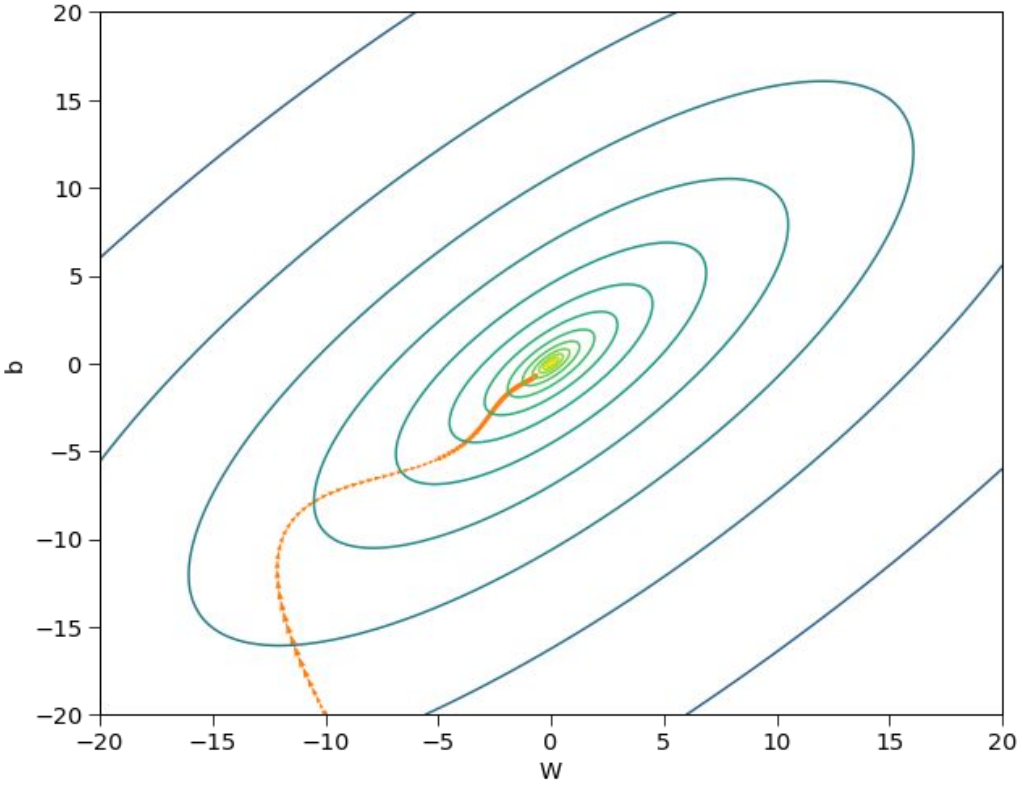

iteration =2
15.0
12.5
10.0
7.5
5.0
2.5
0.0
$J(\theta)$
−4
−2
0
2
4
θ


iteration =3
15.0
12.5
10.0
7.5
5.0
2.5
0.0
$J(\theta)$
−4
−2
0
2
4
θ

How does ***Gradient descent*** finds the path of maximal descent towards the minima

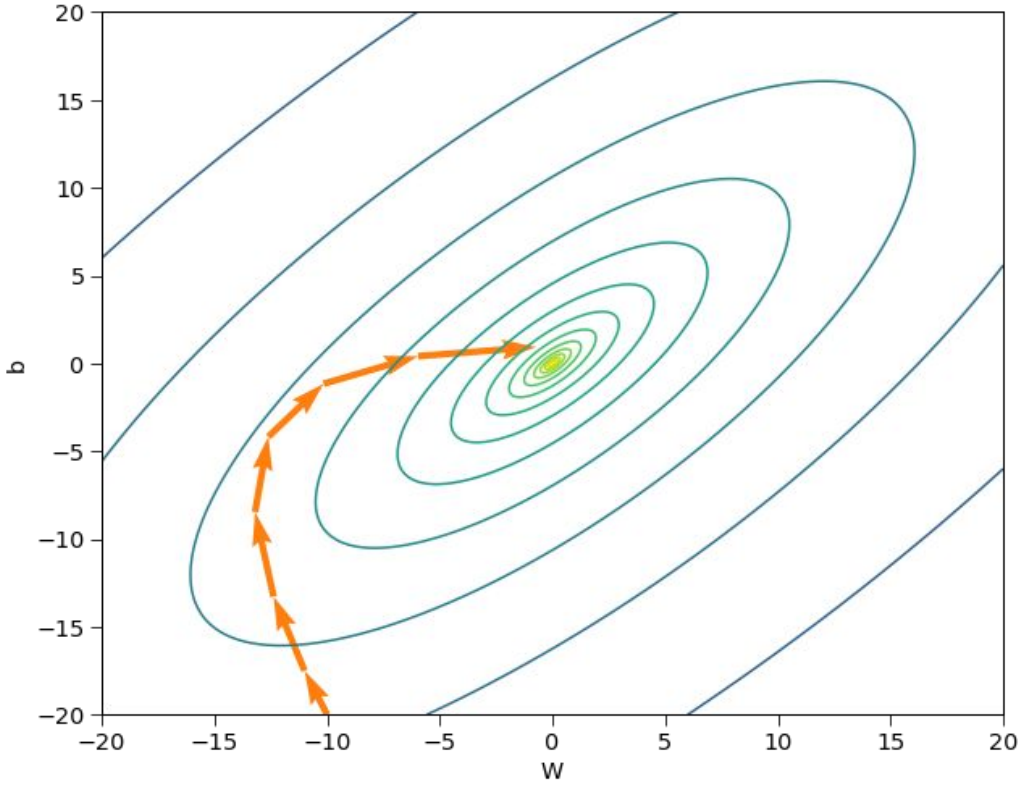
What ***learning rate*** should I use?

Low learning rate



If the learning rate is too low, it takes a long time for the model to make progress and move away from regions of the parameter space with high cost function

High learning rate



A high learning rate will cause oscillations in the cost function

How do we set the learning rate?

These are some common approaches:

1) Linearly decay the learning rate until some iteration and leave it constant afterwards

2) Run the optimization for say 100~200 iterations with many learning rates and see which learning rate yields a lower cost function

3) Cross-validation

Stochastic Gradient Descent

$$\{x_b\}_{b=1}^{B=\text{Batchsize}} \leftarrow \text{training examples}$$

$$\frac{\partial J}{\partial \theta(t)} \simeq \frac{1}{B} \sum_{b=1}^{B} \frac{\partial J_b}{\partial \theta(t)}$$

$$\theta(t+1) = \theta(t) - lr \frac{\partial J}{\partial \theta(t)}$$

Hyperparameters of the Stochastic gradient descent

$$\underbrace{B}_{\text{Batch size}}, \underbrace{lr}_{\text{learning rate}}$$

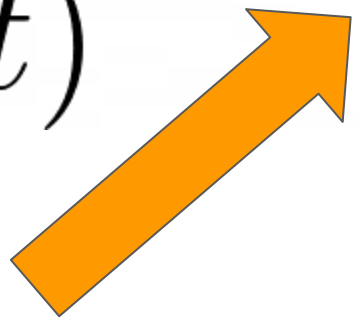
Improvements to gradient descent

Momentum

Adds an extra term to the gradient descent update

This extra term is called: ***velocity***

$$\theta_{t+1} = \theta_t - lr \frac{dJ}{d\theta(t)} + \gamma v$$



Accumulation of gradients from the past

Momentum

$$\theta_{t+1} = \theta_t - lr \frac{dJ}{d\theta(t)} + \gamma v$$

The parameter gamma is now an extra hyperparameter

Typical value = 0.9
Larger values wrt the learning rate will cause the past gradients to affect the current update more

Adagrad (Duchi et al., 2011):

For each model parameter we have one learning rate

Per-parameter learning rates are decreased according to the previous gradients:

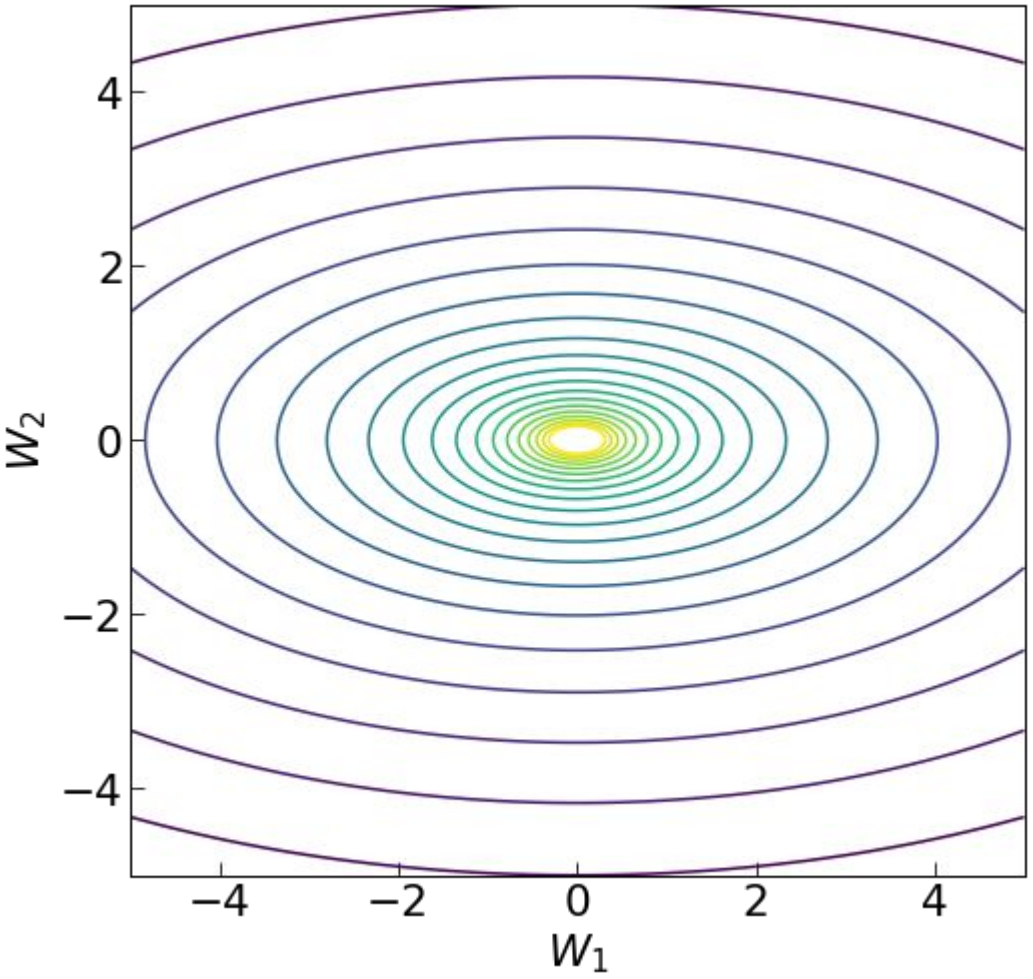
$$lr(\theta_a, t) \propto \frac{1}{\sqrt{\sum_{t'=1}^{t-1} ||dJ/d\theta_a|_{t'}||^2}}$$

If the gradients are large along one direction, learning rate is decreased more and vice-versa

Adagrad

Question:

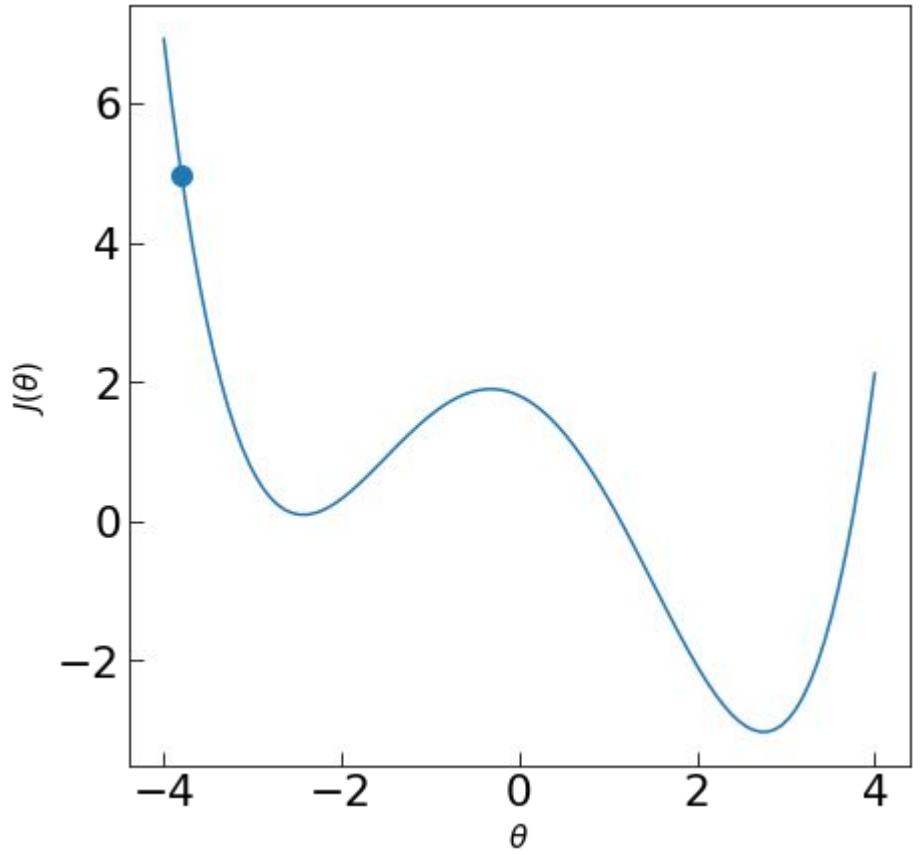
In which direction does the learning rate decrease faster?



Adagrad

Con:

By the time the parameter reaches the vicinity of the local minimum, the learning rate might have been decreased so much that it may get stuck



RMSprop (Hinton, 2012)

Similar to AdaGrad

But with a different way of decaying the rate

The contribution of past epochs to the average gradient exponentially drops

Gradients from the recent iteration contribute more than the gradients from the past

Adam (Kingma and Ba 2014)

(roughly speaking) Combines RMSprop and Momentum

Similarity to Momentum: past gradients are used to estimate the first moment of the gradient for the update

Similarity to RMSprop: past gradients are used to estimate the second moment of the gradient needed to rescale the learning rate

Adam (Kingma and Ba 2014)

Fairly robust algorithm for optimization

There are some hyperparameters that we discuss in the coding session

Now it’s time to code!
Up next: Overfitting!