Deep Learning (CS324)

3. Deep networks & backpropagation*

Prof. Jianguo Zhang

SUSTech

*Based on http://www.deeplearningbook.org chapters 5 and 6

Learning machines

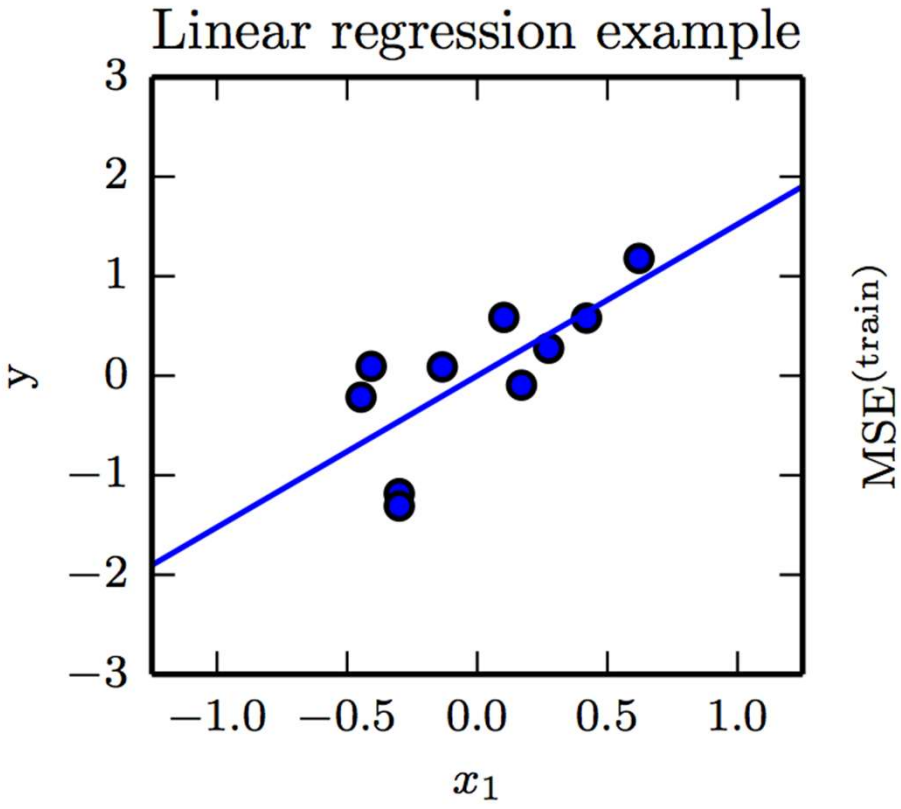
- *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."* (Mitchell, 1997)

Learning machines

- Task **T**: e.g., classification, regression, density estimation, etc.
- Performance measure **P**: e.g., accuracy, error rate
- Experience **E**: supervised, unsupervised, reinforcement learning

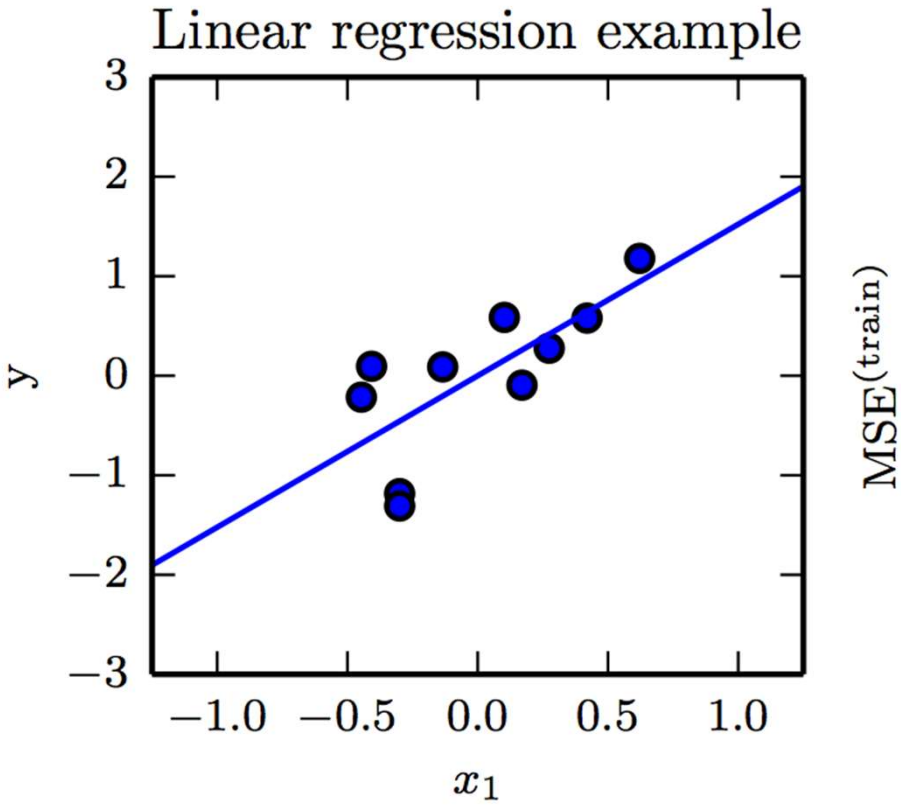
https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets

Example: Linear regression



$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

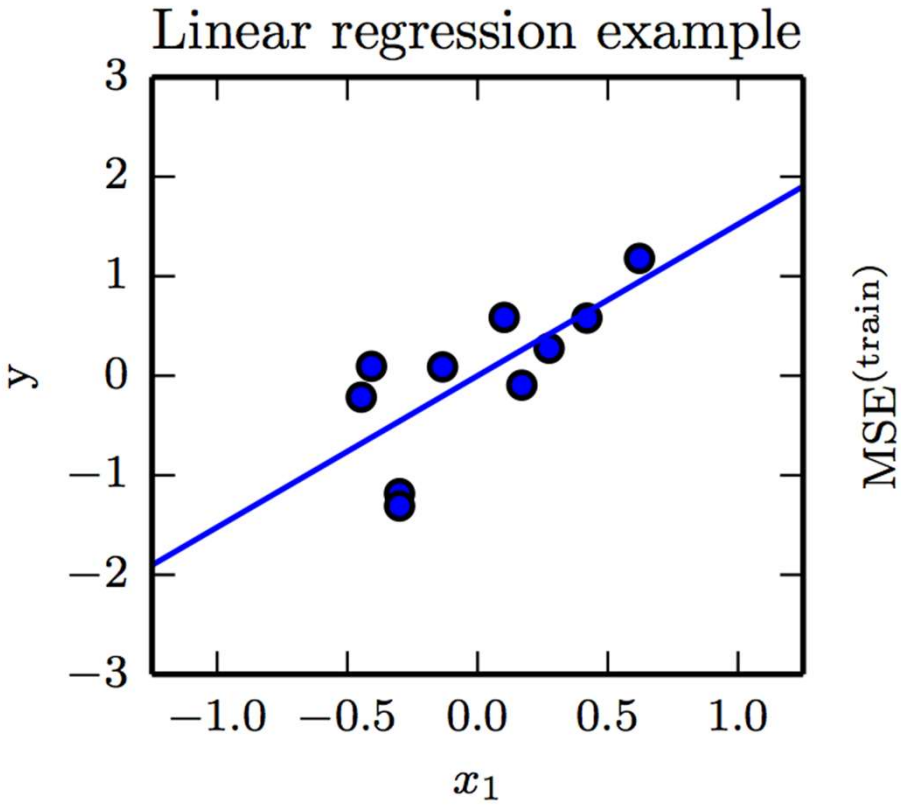
Example: Linear regression



$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

Input

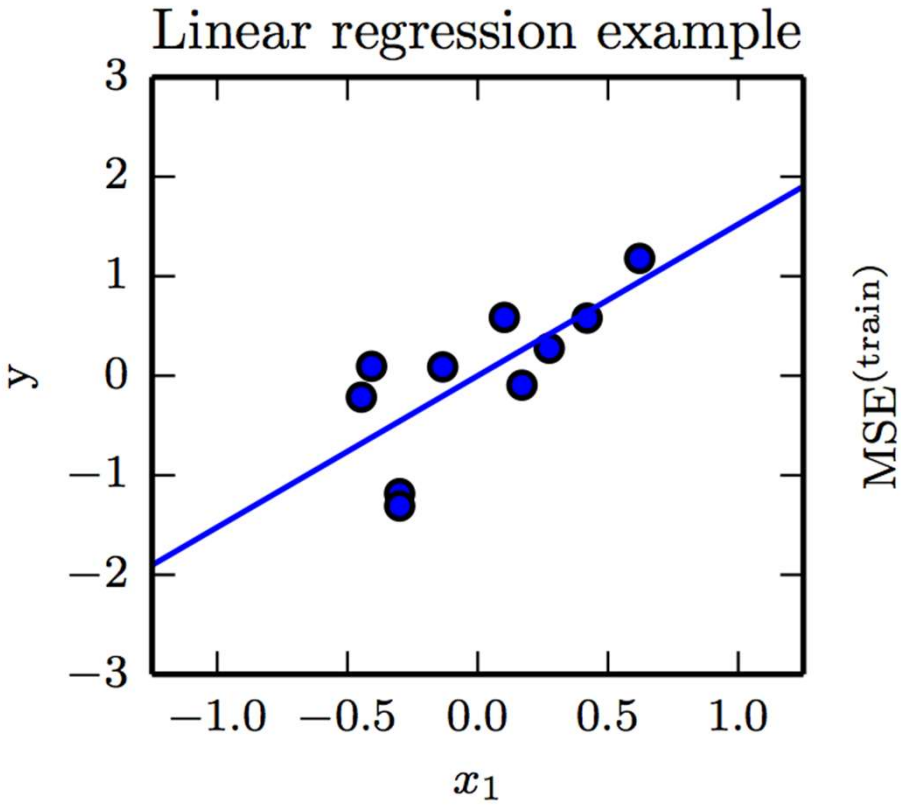
Example: Linear regression



$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

Prediction

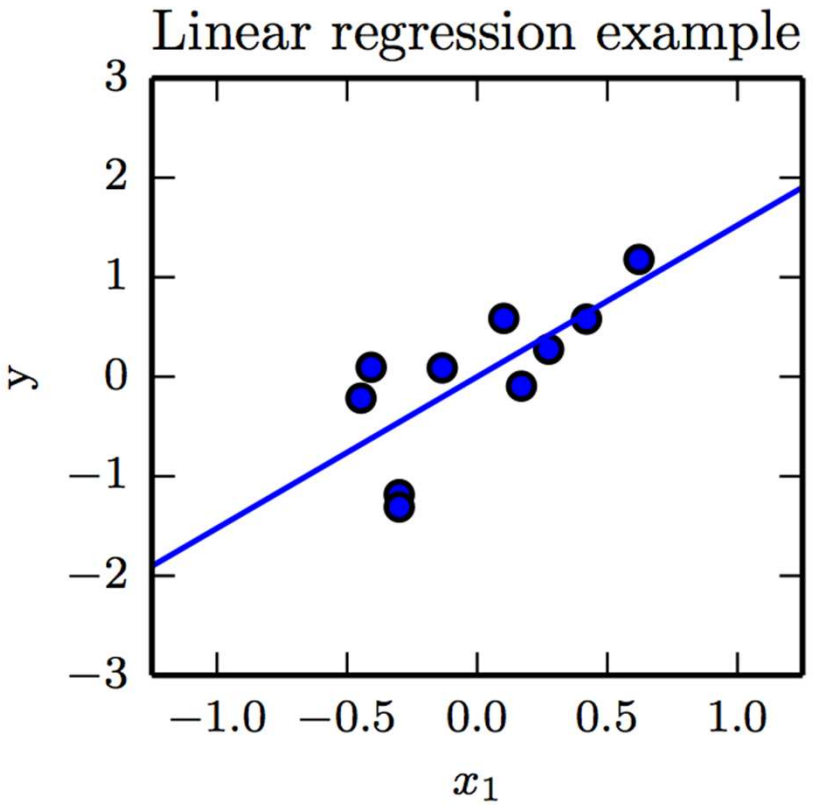
Example: Linear regression

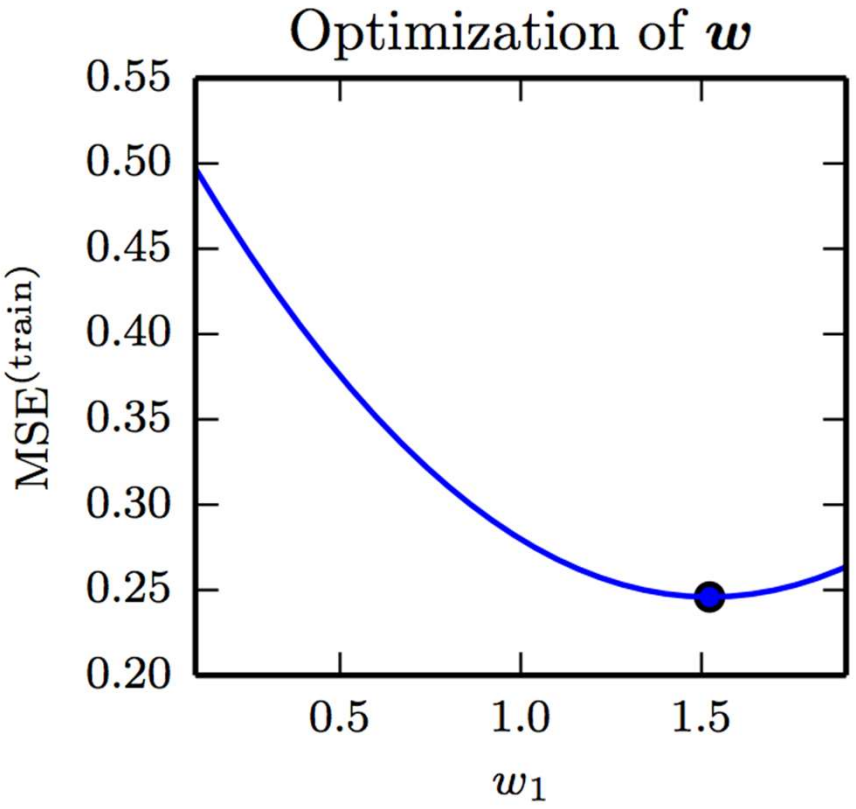


$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

Learnable parameters

Example: Linear regression

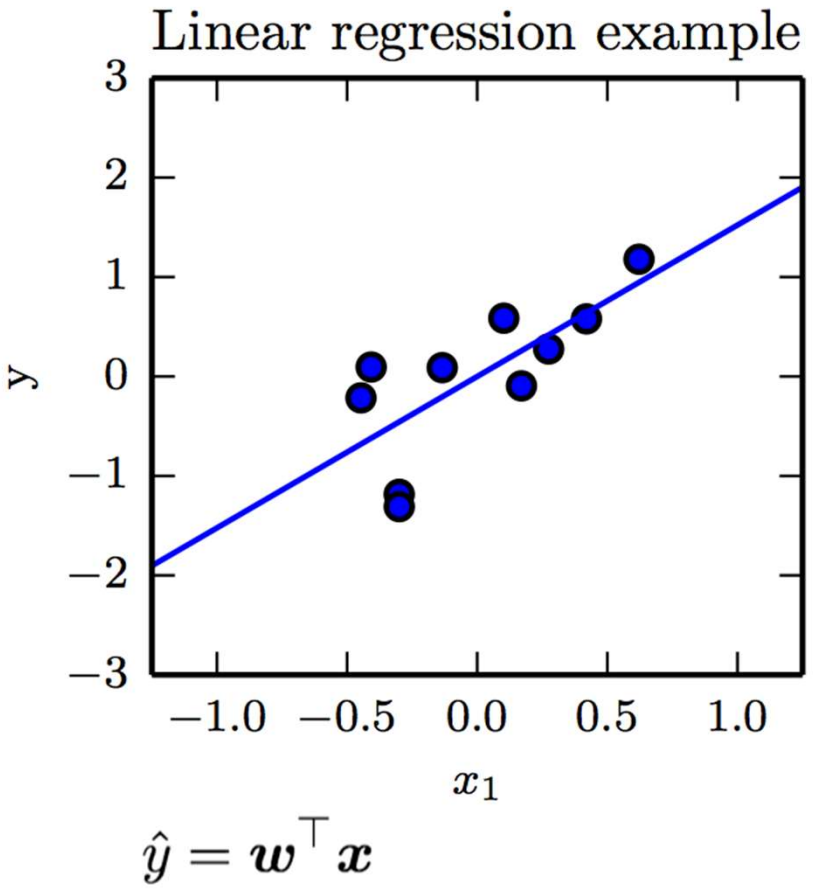




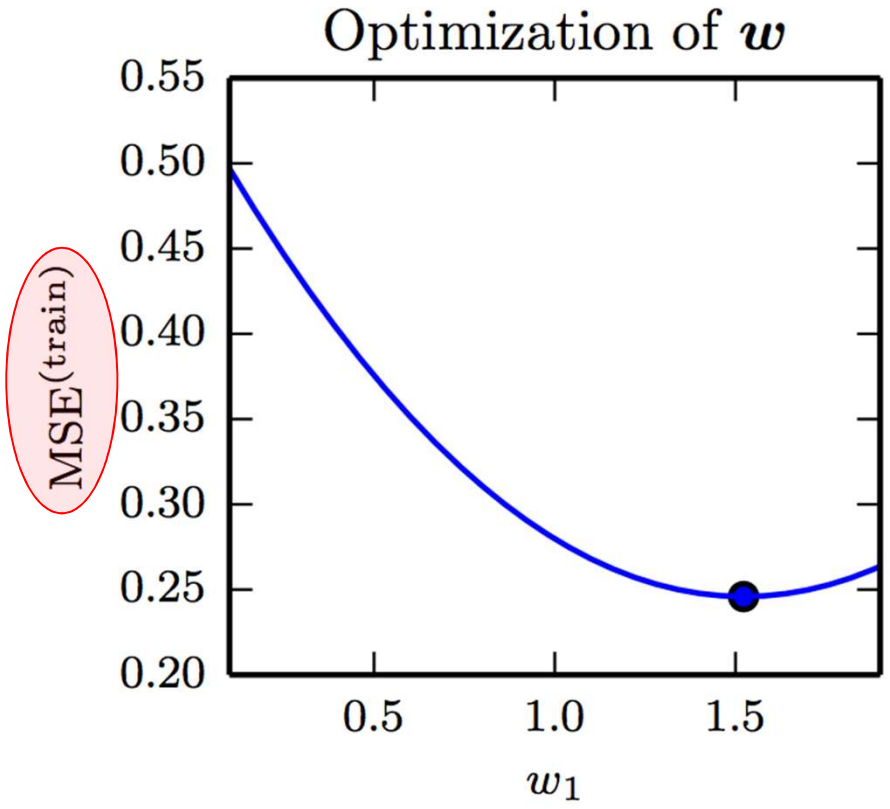
$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

Learnable parameters

Example: Linear regression



$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$



$$\mathrm{MSE}_{\mathrm{train}} = \frac{1}{m}\sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2,$$

Mean squared error: our performance measure **P**

Example: Linear regression

- Mean squared error: our performance measure **P**

Training error: the error/loss computed on the training set. (In this case the MSE on the training set)

Test error/loss: the error/loss on the test set

Generalization error/loss: the gab between the training error and test error. (**Note that it only makes sense only when the training error is small. There is no need to talk about GE when the training error is large**)

Underfitting and overfitting

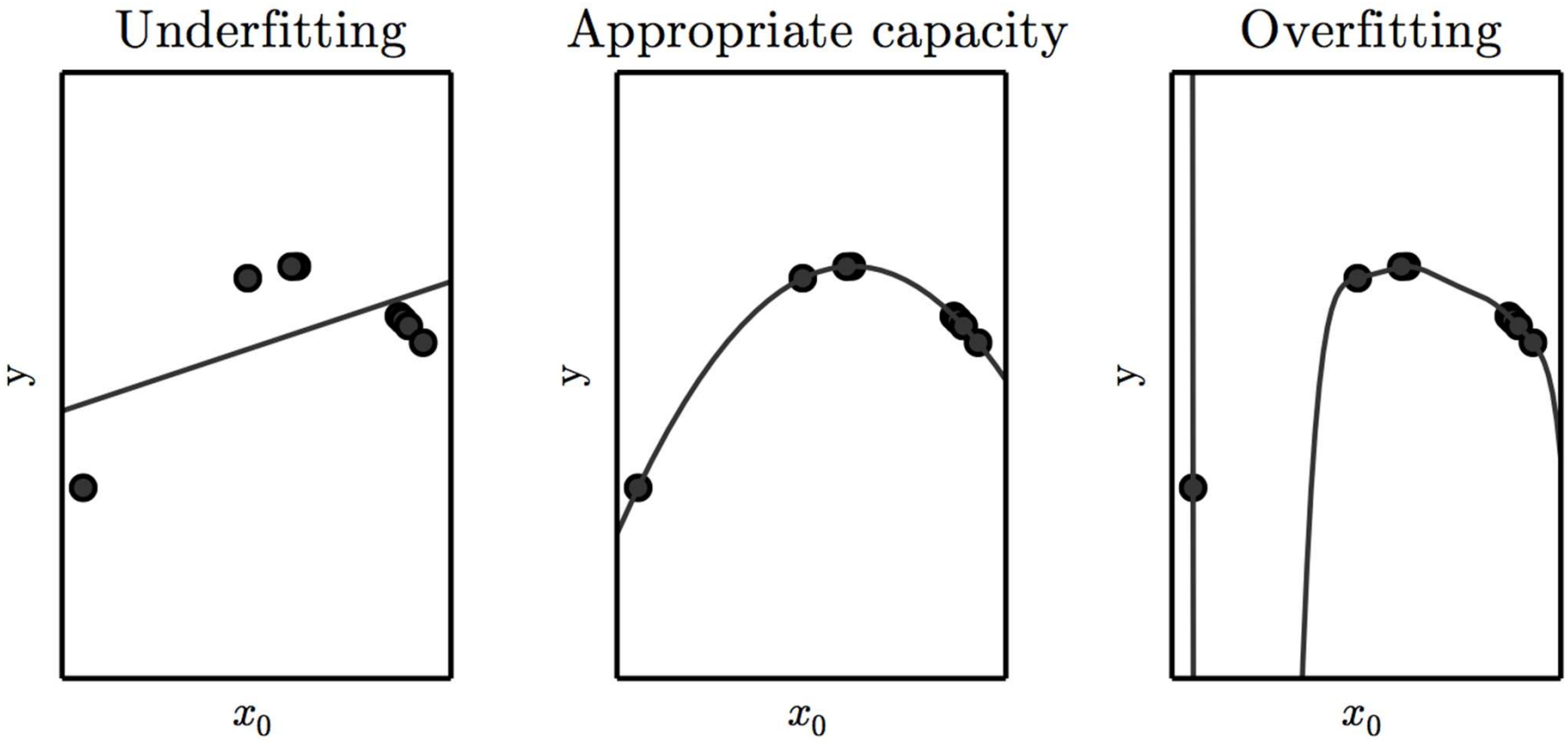


Figure 5.2

Model Capacity: the ability of a model to fit a variety of functions

Generalisation and capacity

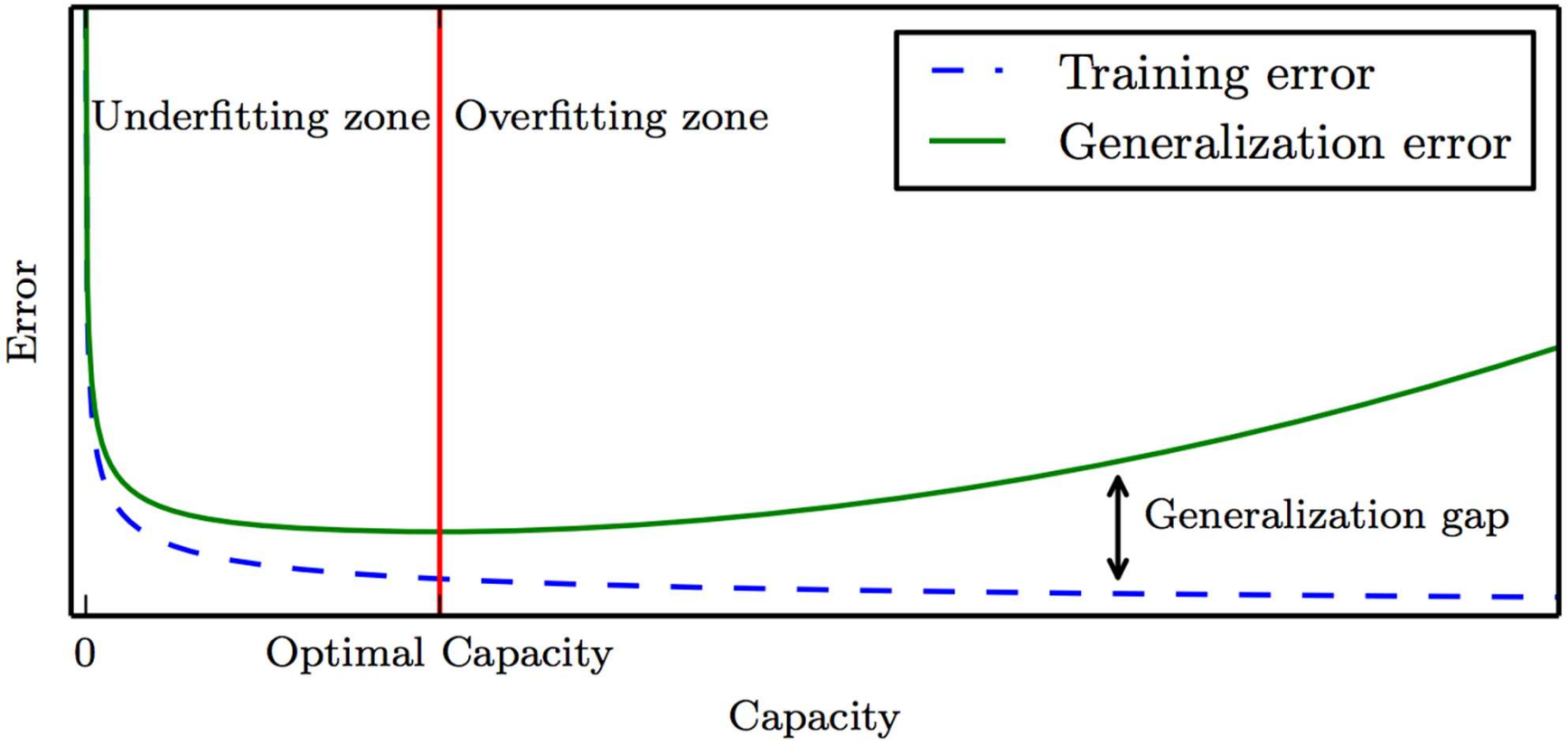
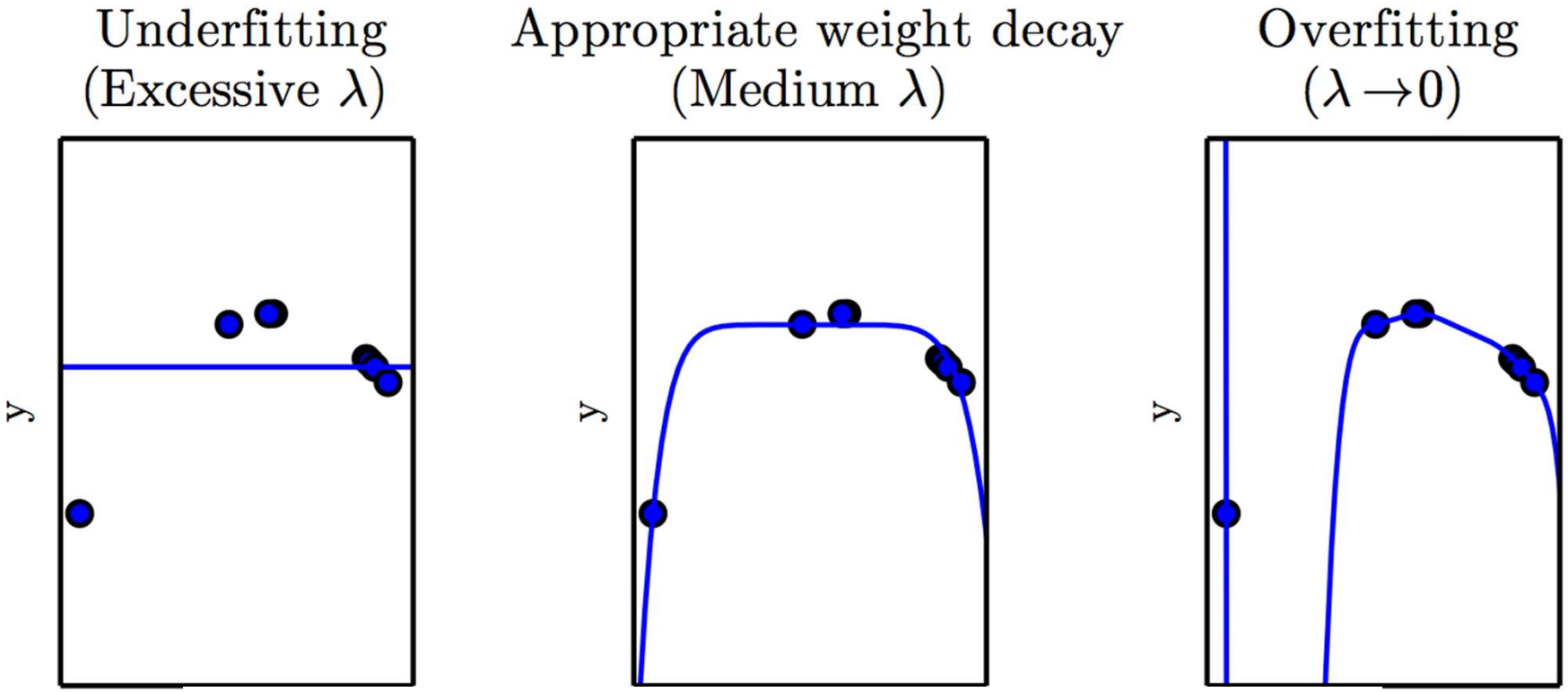


Figure 5.3

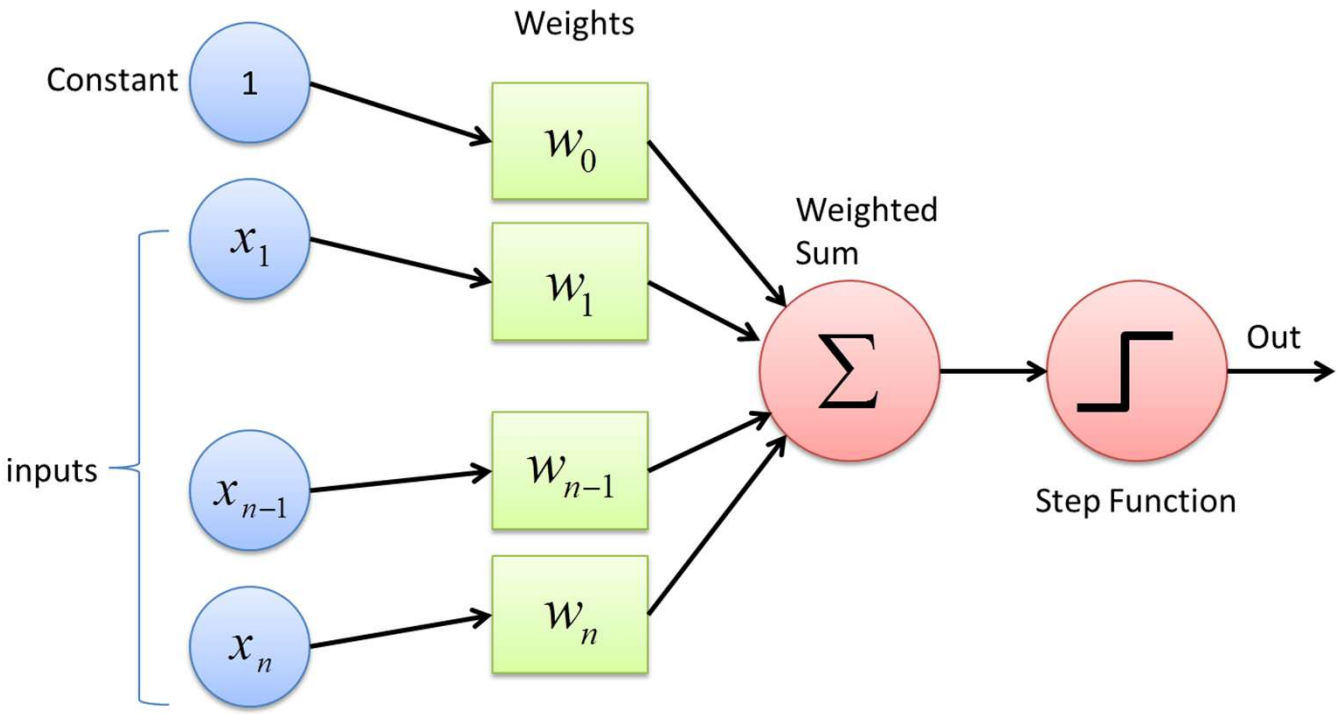
Hyperparameters & validation set

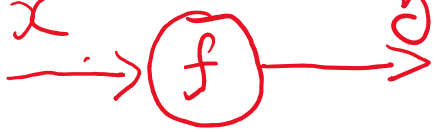


$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w},$$

Figure 5.5 – example polynomial regression with hyper-paras

Perceptron

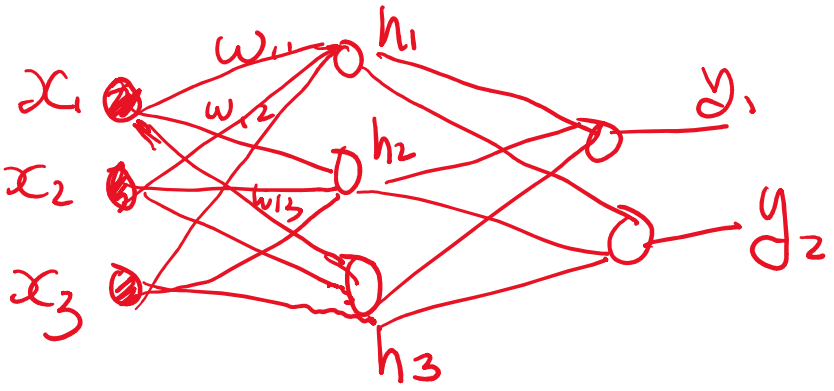

Weights
Constant
1
w_0
x_1
w_1
Weighted
Sum
Σ
Out
inputs
x_{n-1}
w_{n-1}
x_n
w_n
Step Function



$y = f(x)$

$y = x$

$\frac{dy}{dx} = 1$



Deeper

Computation Graph

① $z = \sigma(b)$

② $b = wx$

$z(x) = \sigma(wx)$

If $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial b} \cdot \frac{\partial b}{\partial x} = \sigma(b) \cdot (1 - \sigma(y)) \cdot w$

$\sigma(y) = y^{-1}$

$y = 1 + e^{-x}$

$\frac{d\sigma}{dx} = \frac{d\sigma}{dy} \cdot \frac{dy}{dx} = +1 \cdot y^{-2} \cdot e^{-x} \cdot (+1)$

$= y^{-2} e^{-x} = \frac{+e^{-x}}{(1+e^{-x})^2}$

$= \left(\frac{1+e^{-x}}{1+e^{-x}} \cdot \frac{+1}{1+e^{-x}}\right) \cdot \frac{1}{1+e^{-x}}$

$= \sigma(x)(1 - \sigma(x))$

Perceptron

Given a training set D = $\{(\mathbf{x}_i, y_i)\}$, $\mathbf{x}_i \in \Re^n$, $y_i \in \{-1,1\}$

1. Initialize $\mathbf{w} = 0 \in \Re^n$
2. For epoch = 1 … T:
   1. Shuffle the data
   2. For each training example $(\mathbf{x}_i, y_i) \in$ D:
      - If $y_i \mathbf{w}^T \mathbf{x}_i \leq 0$, update $\mathbf{w} \leftarrow \mathbf{w} + r\, y_i\, \mathbf{x}_i$
3. Return $\mathbf{w}$

Prediction: sgn($\mathbf{w}^T\mathbf{x}$)

Perceptron

Given a training set $D = \{(\mathbf{x}_i, y_i)\}$, $\mathbf{x}_i \in \Re^n$, $y_i \in \{-1,1\}$

1. Initialize $\mathbf{w} = 0 \in \Re^n$
2. For epoch = 1 … T:
   1. Shuffle the data
   2. For each training example $(\mathbf{x}_i, y_i) \in D$:
      - If $y_i \mathbf{w}^T \mathbf{x}_i \leq 0$, update $\mathbf{w} \leftarrow \mathbf{w} + r\, y_i \mathbf{x}_i$
3. Return $\mathbf{w}$

Prediction: $\text{sgn}(\mathbf{w}^T \mathbf{x})$

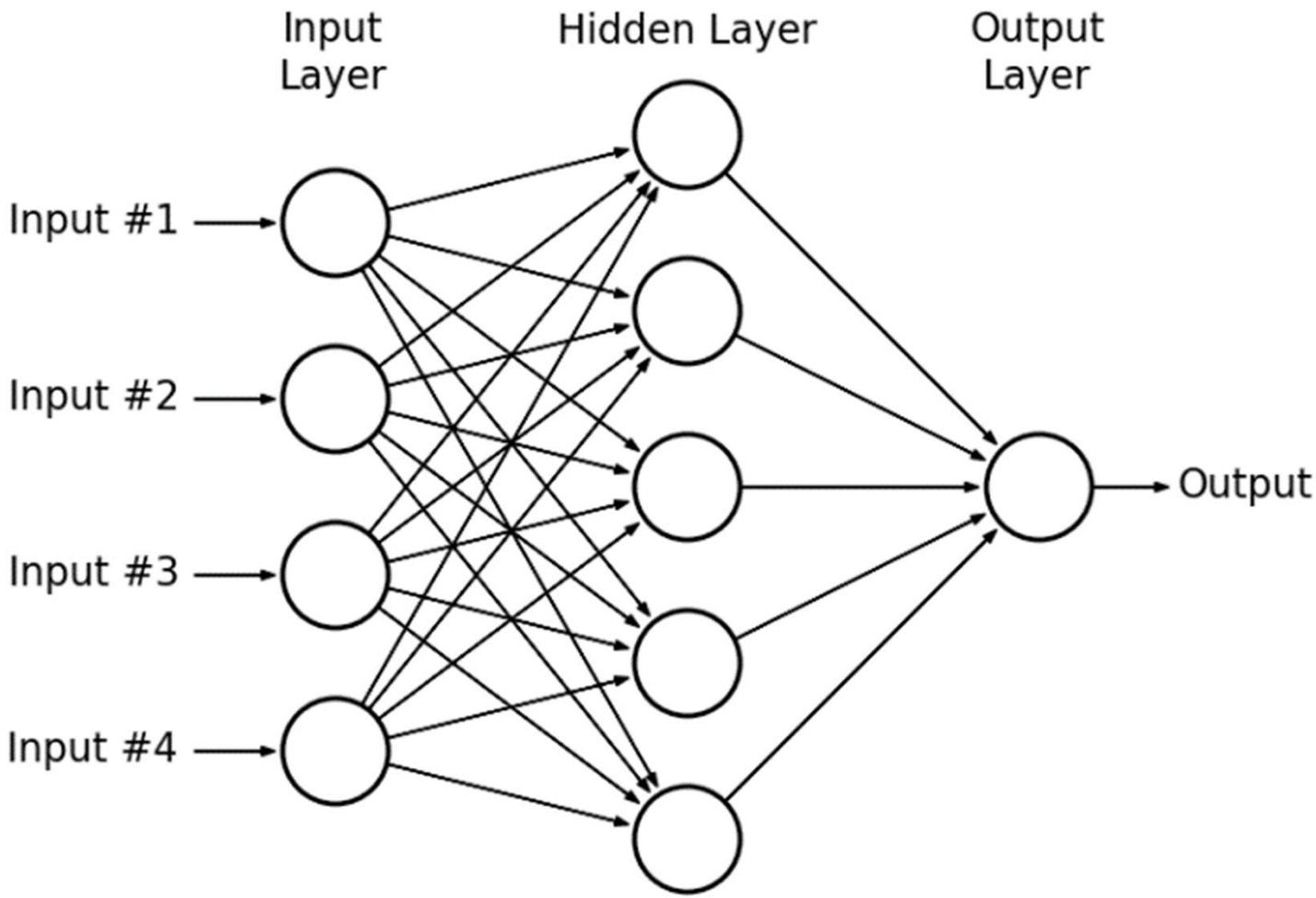
Going deep

- We've seen that the perceptron is just a linear classifier so it cannot solve non-linear problems, e.g., XOR
- Solution? Apply non-linear transformation to input so that data is linearly separable in the new space
  - Kernel methods: how to choose the kernel?
  - Deep learning: let's learn the transformation!

Going deep

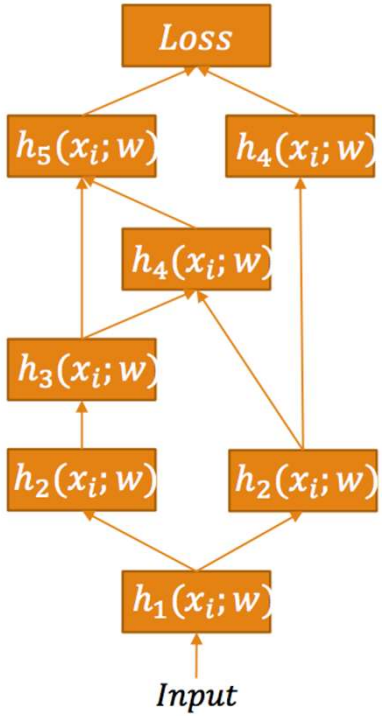
- We've seen that the perceptron is just a linear classifier so it cannot solve non-linear problems
- Solution? Apply non-linear transformation to input so that data is linearly separable in the new space
  - Kernel methods: how to choose the kernel?
  - Deep learning: let's learn the transformation!
    - Non-convex optimisation problems require gradient learning

Going deep – Single Hidden Layer



Deep neural networks

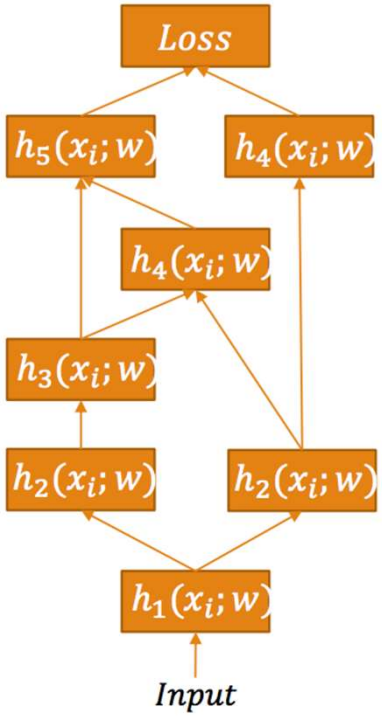
- So what is deep learning?
  A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimised with (stochastic) gradient descent



$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

Deep neural networks

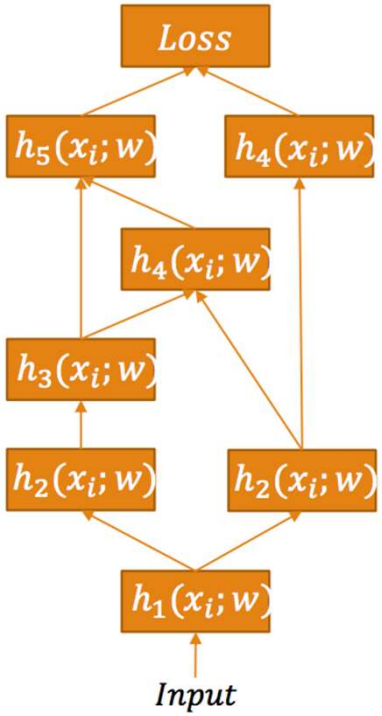
- Each module is a function with trainable parameters **w**
- Functions should be at least first-order differentiable (almost) everywhere
- Store module input for easier backpropagation



$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

Deep neural networks

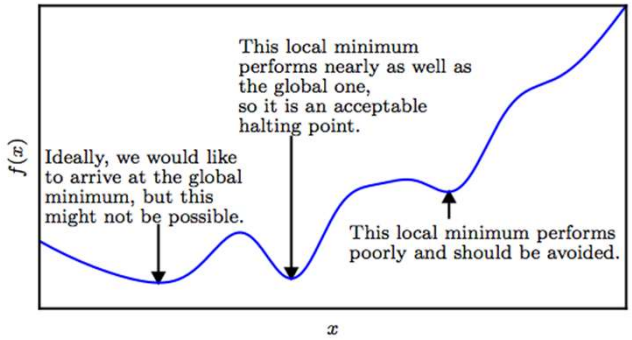
- Output of one module in input of next module in the forward direction, forming a directed acyclic graph
- But there can also be loops (recurrent connections in RNNs)

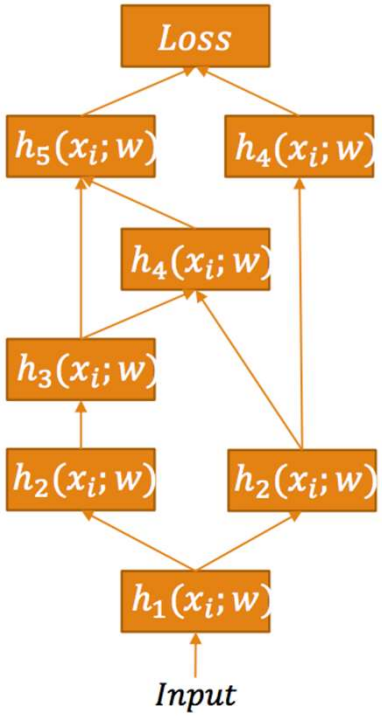


$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

Gradient-based learning

- How to find the optimal weights (i.e., those that minimise the loss)?
  - Gradient-based learning

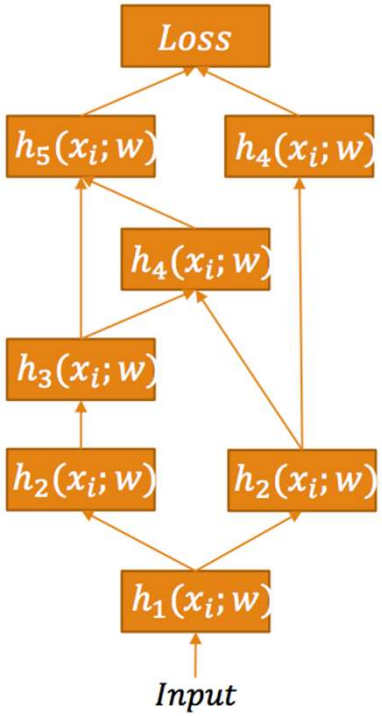




$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

Gradient-based learning

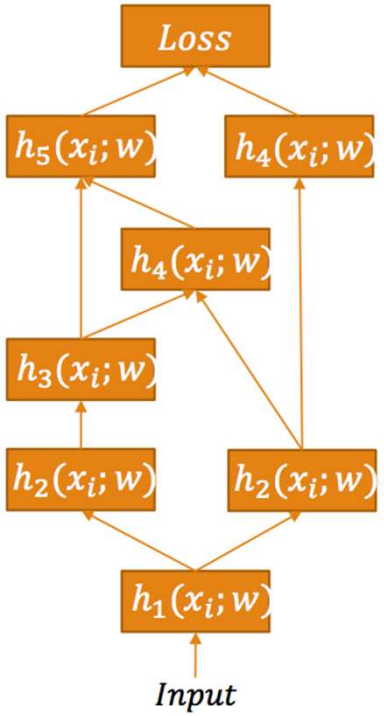
- But we're computing the gradient of a very complex function!
- …and here's where we need backpropagation



$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

Deep neural networks

- But first, let's have a closer look at the ingredients to build a deep network
  - 1) Cost functions
  - 2) Output units
  - 3) Hidden units
  - 4) Architecture



$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

Cost functions: cross entropy

- Similar to other parametric models, we usually train using maximum likelihood, so cost function is the negative log-likelihood, i.e., **cross-entropy** between training data and model distribution

$$L(\tilde{y}) = -\sum_{i=1}^{C} y_i \log \tilde{y}_i \qquad \frac{\partial L}{\partial \tilde{y}_i} = -\frac{y_i}{\tilde{y}_i}$$

- We like logs! When applies on output functions that tend to saturate (so, small gradient) because of exp: log is the antidote we need

Cost functions: euclidean loss

- Commonly used for regression tasks

$$L(\tilde{y}) = 0.5||y - \tilde{y}||^2$$

- Gradient:

$$\frac{\partial L}{\partial \tilde{y}} = \tilde{y} - y$$

Output units: linear

- Affine transformation

$$\tilde{y} = wx + b$$

- Often used to produce the mean of a conditional Gaussian distribution
- No activation saturation, strong and stable gradient

$$\frac{\partial \tilde{y}}{\partial w} = x$$

Output unit: sigmoid

- If we want to output a binary variable, the usual maximum likelihood approach is to output a Bernoulli distribution
- In simple terms, we want our output unit to produce a probability, i.e., a number in [0,1]

$$\tilde{y} = \sigma(wx + b)$$

Output unit: sigmoid

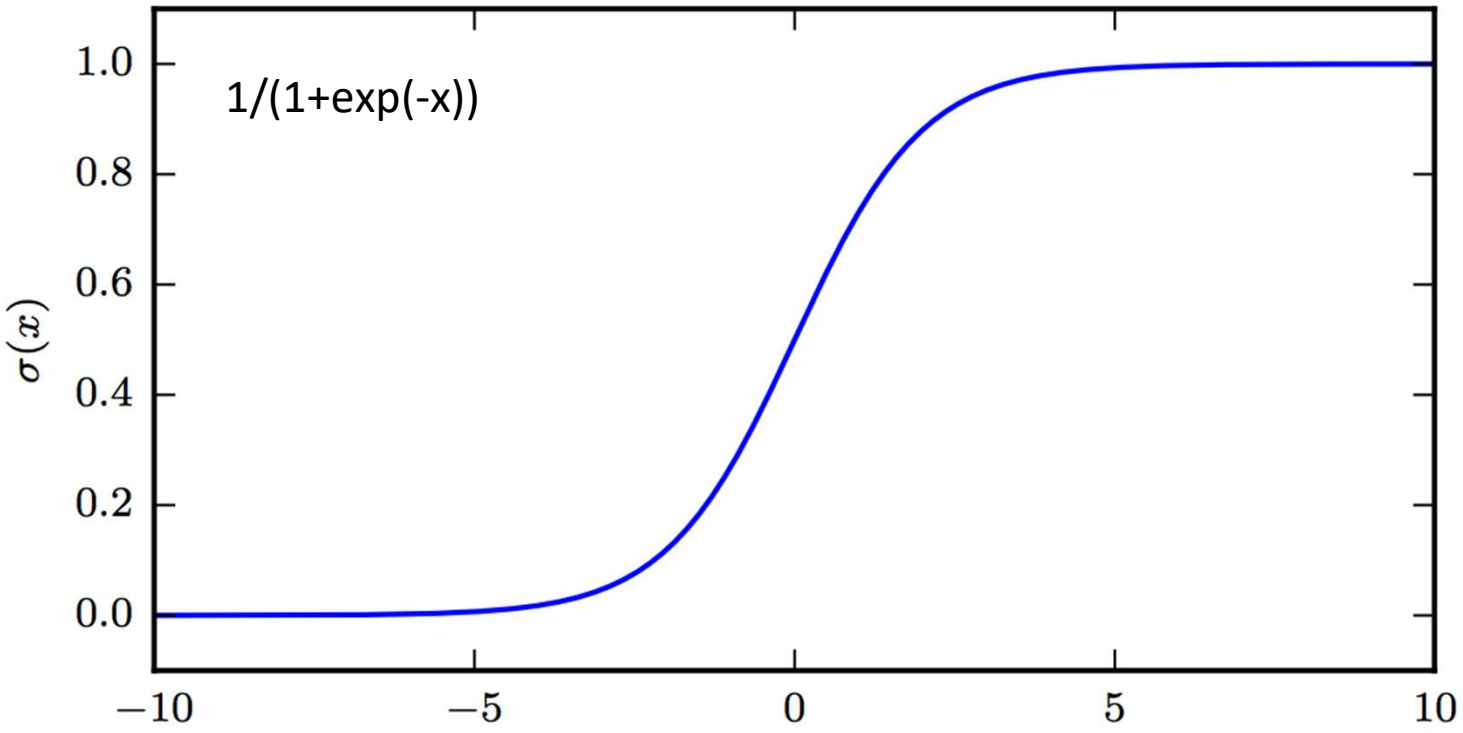


Figure 3.3: The logistic sigmoid function.

Output unit: sigmoid

- If we want to output a binary variable, the usual maximum likelihood approach is to output a Bernoulli distribution
- In simple terms, we want our output unit to produce a probability, i.e., a number in [0,1]

$$\tilde{y} = \sigma(wx + b)$$

- Gradient

$$\frac{\partial \tilde{y}}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Output unit: softmax

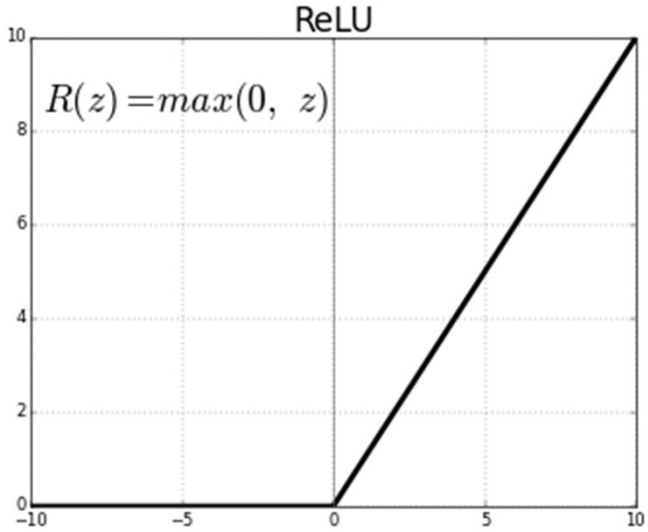
- If we want to output to be a probability distribution over *C* classes, we use a softmax output unit

$$\tilde{y}_i = \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Hidden units: ReLU

- Most popular choice
- Activation

$$a = h(x) = \max(x, 0)$$

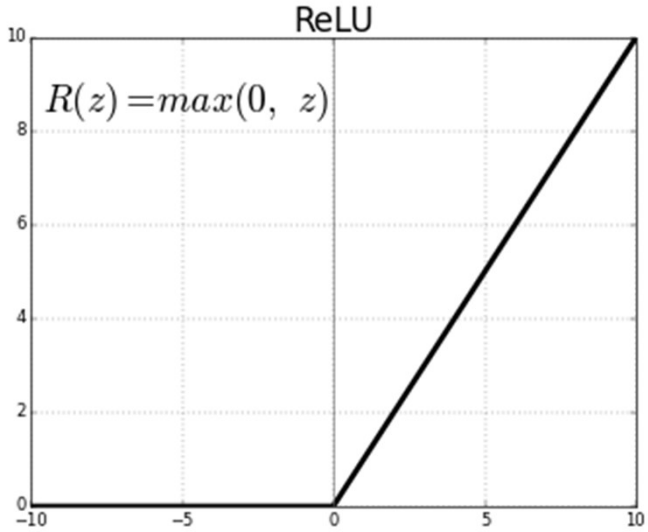


- Gradient

$$\frac{\partial a}{\partial x} = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases}$$

Hidden units: ReLU

- Strong and fast to compute gradient
- Not differentiable at 0 but no big deal
- But ReLU units can "die". More about this when we discuss vanishing gradients…



See https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#Variants for variants

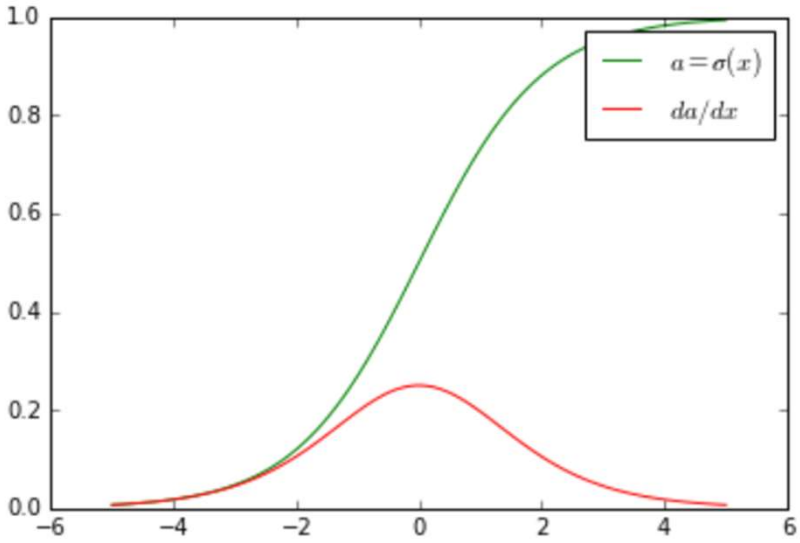
Hidden units: sigmoid

- Activation

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

- Gradient

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



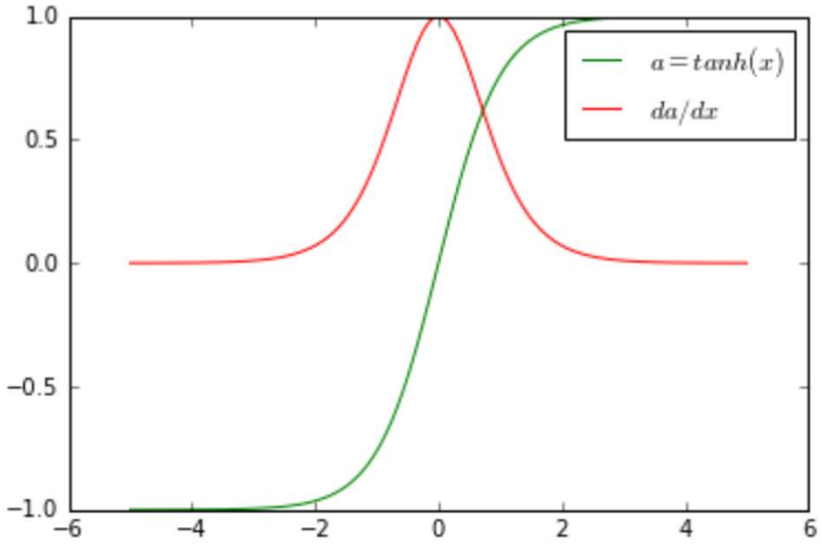
Hidden units: tanh

- Activation

$$a = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Gradient

$$\frac{\partial a}{\partial x} = 1 - \tanh^2(x)$$



Hidden units: sigmoid and tanh

- Both very popular before the introduction of ReLU
- But they saturate very easily (zero gradient)
  - This can be interpreted as the unit being overconfident (i.e., stops to learn)
- Also, small gradients, can be problematic especially when we multiply together several small gradients (see chain rule later…)
- Between the two, *tanh* better as a hidden unit

Final notes on units

- Many more types of units out there (hinge loss, dropout, L1-regularisation, etc.)
- We can combine different units to create new ones, or create totally new ones
- ReLU popular choice for hidden units
- Choice of output unit depends on the problem

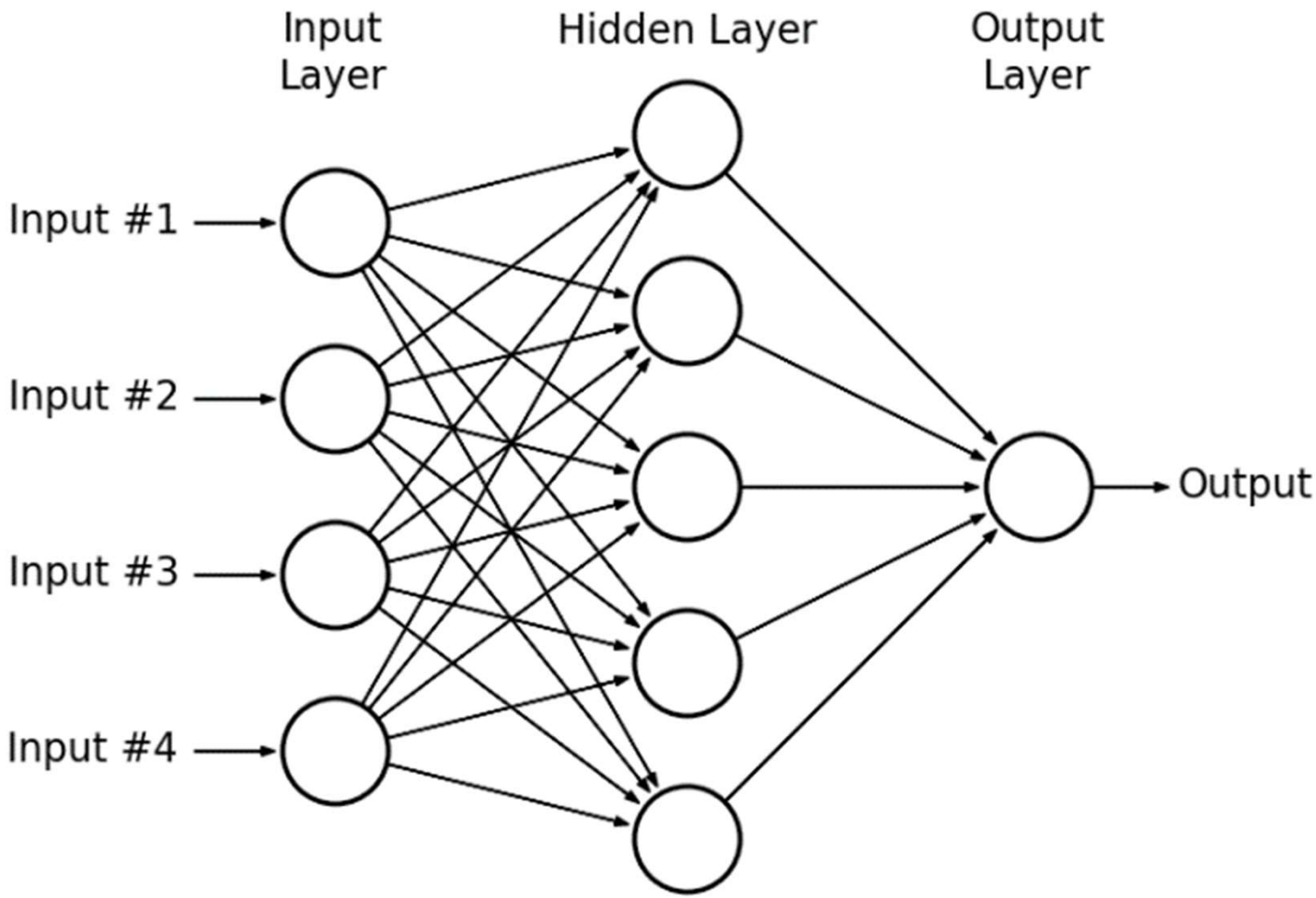
Architecture design

- Usually neural networks are organised into groups of units called **layers**
- Each layer is a function of the layer that precedes it, i.e., the layers form a chain (but not always)

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) \qquad \boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

- How to choose number and width of the layer?
  - The number of layers is usually referred to as **depth**

Architecture design



Architecture design

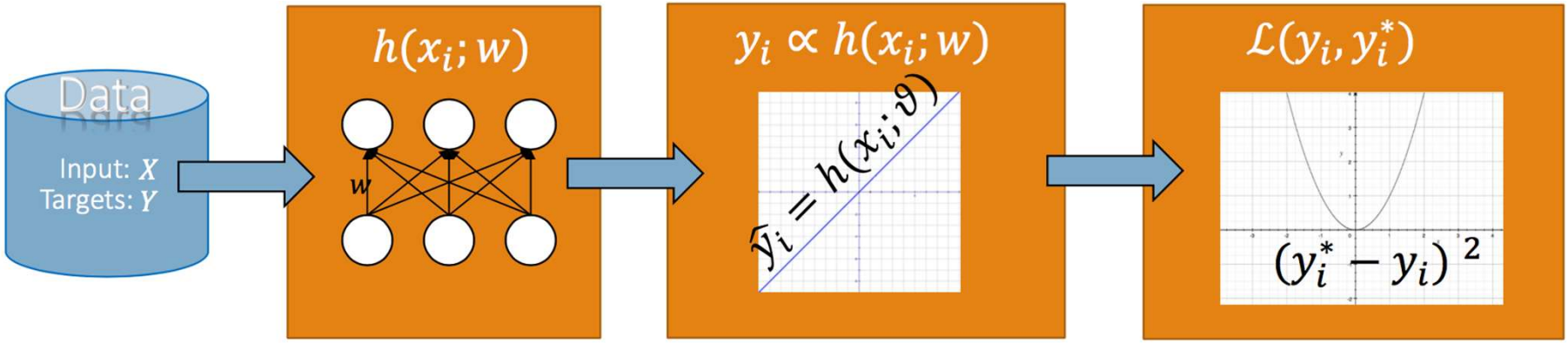
- *How many layers and units?*
  - The **universal approximation theorem** says that, provided it has enough units, a single layers is sufficient to approximate any continuous function on a closed and bounded subset of R^n
  - But the learning can still fail!
    - Overfitting
    - Local minima

Architecture design

- *How many layers and units?*
  - The more layers we have, the fewer units we need. With only one layer, we may need an exponential number of units with respect to the dimension of the input space.
- *How many connections?*
  - Usually a complete set but in certain domains we can drop some connections, e.g., in computer vision using CNNs

Forward and backward. Repeat.

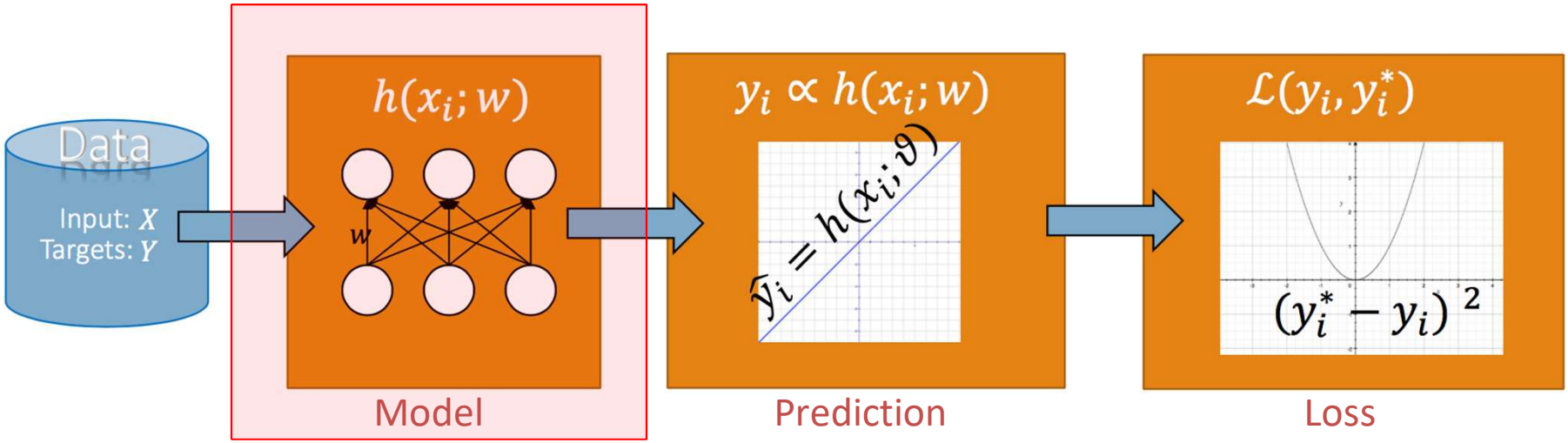
- *How do we train this network of units/functions/modules?*



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

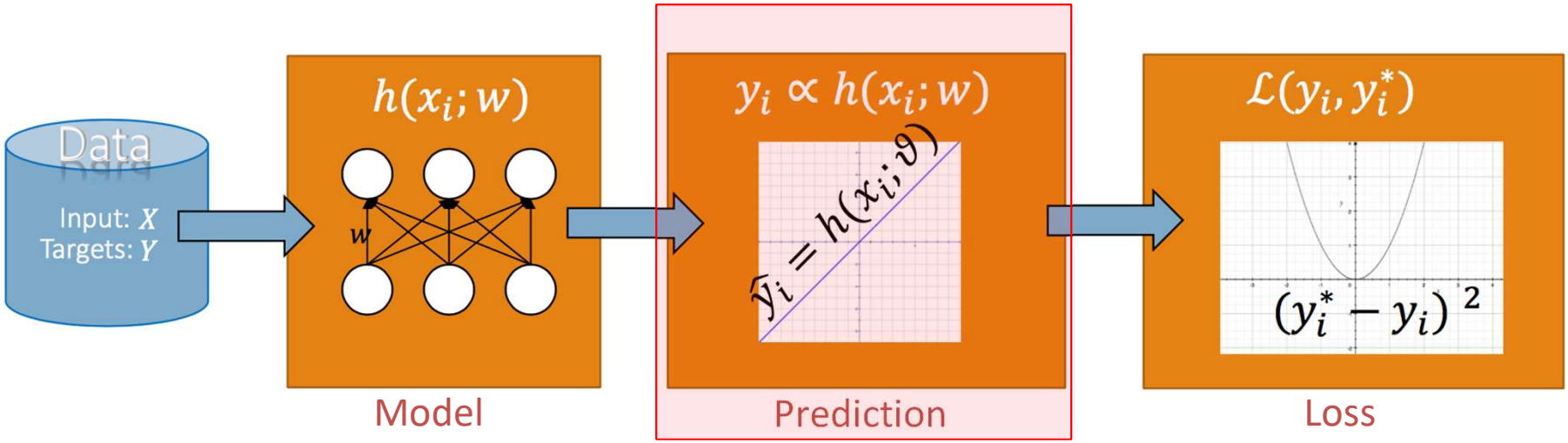
- *How do we train this network of units/functions/modules?*
- We start by defining an architecture and initialising it randomly…



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

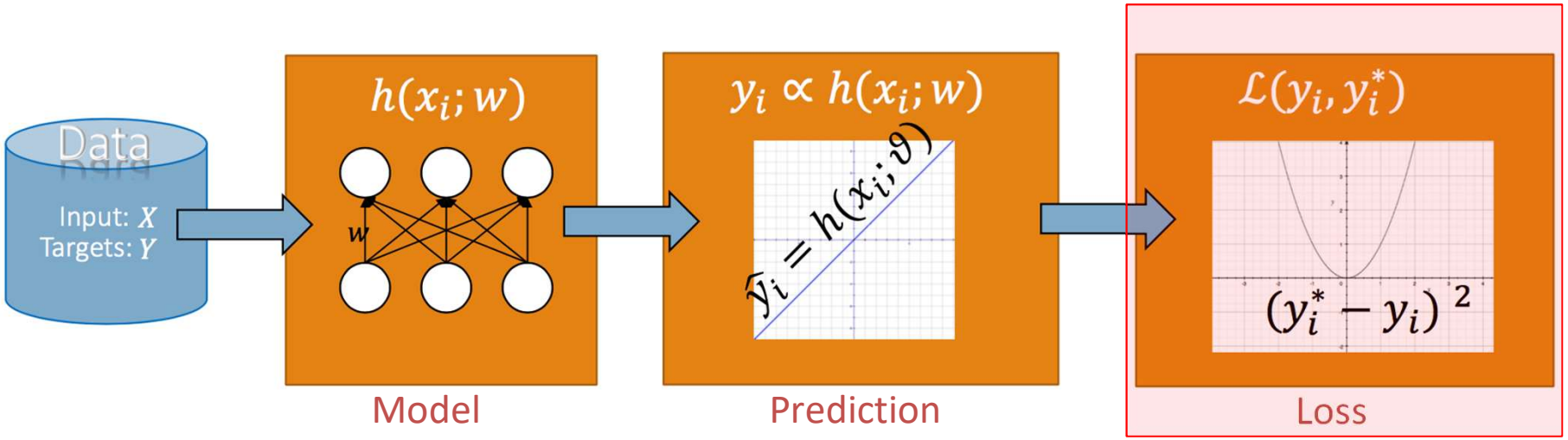
- *How do we train this network of units/functions/modules?*
- Then we feed some data in our model and we make a prediction (probably rather inaccurate)



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

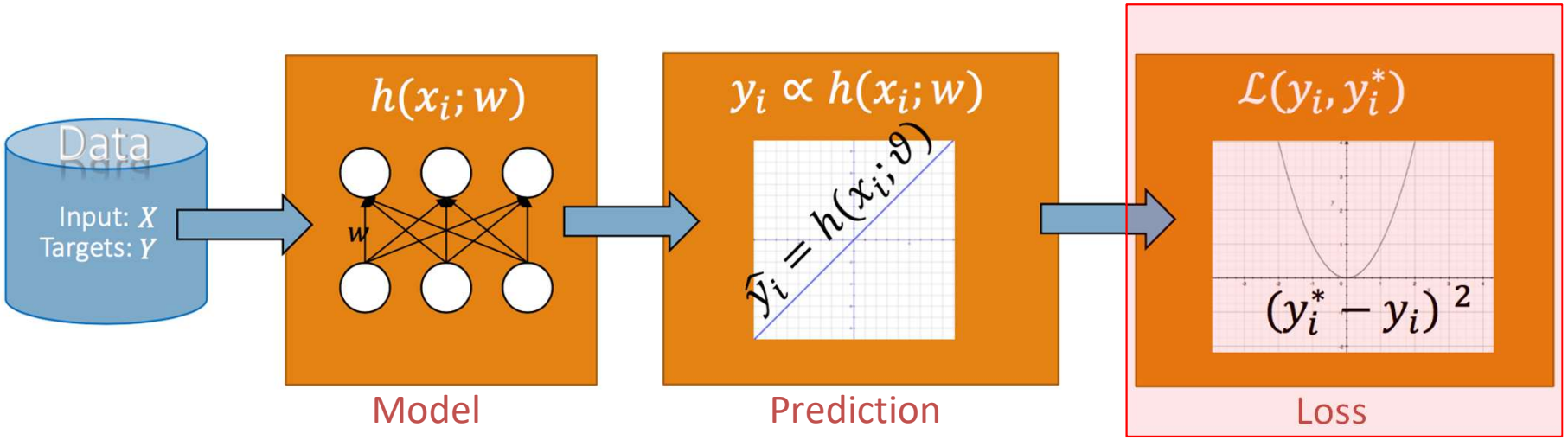
- *How do we train this network of units/functions/modules?*
- Then we evaluate the prediction



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

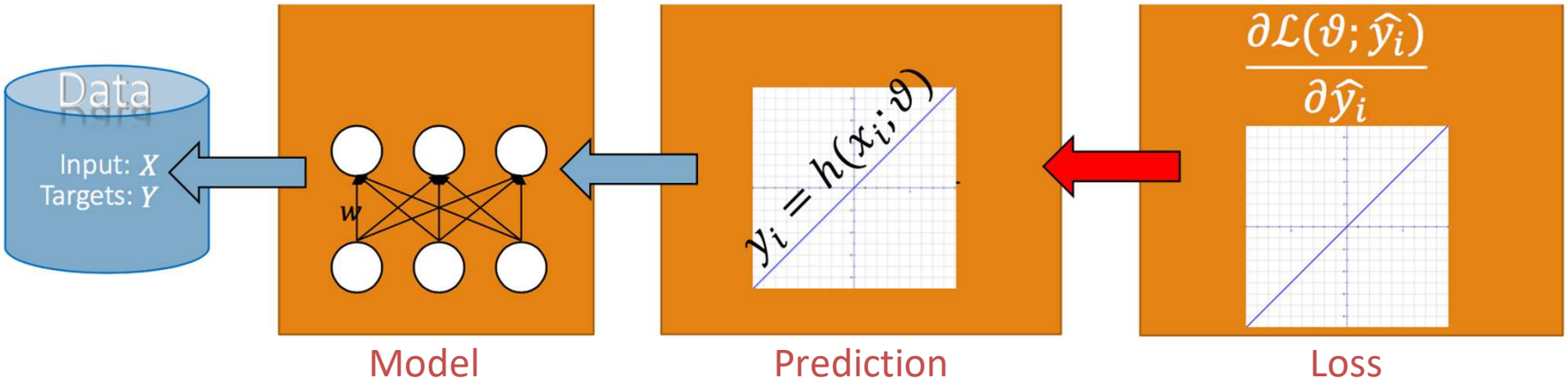
- *How do we train this network of units/functions/modules?*
- This is the **forward propagation** step



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

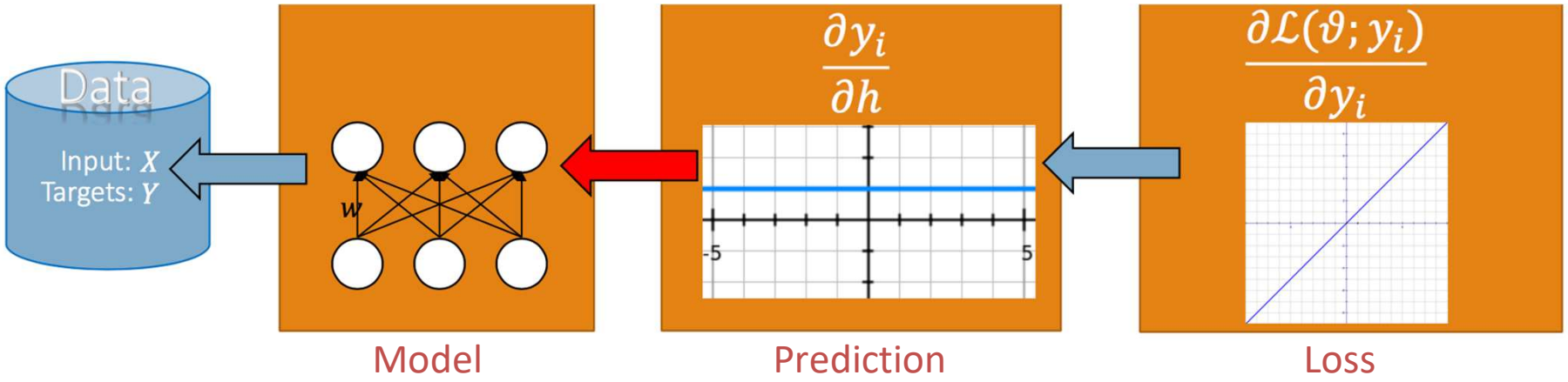
- *How do we train this network of units/functions/modules?*
- Compute the gradient of the loss



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

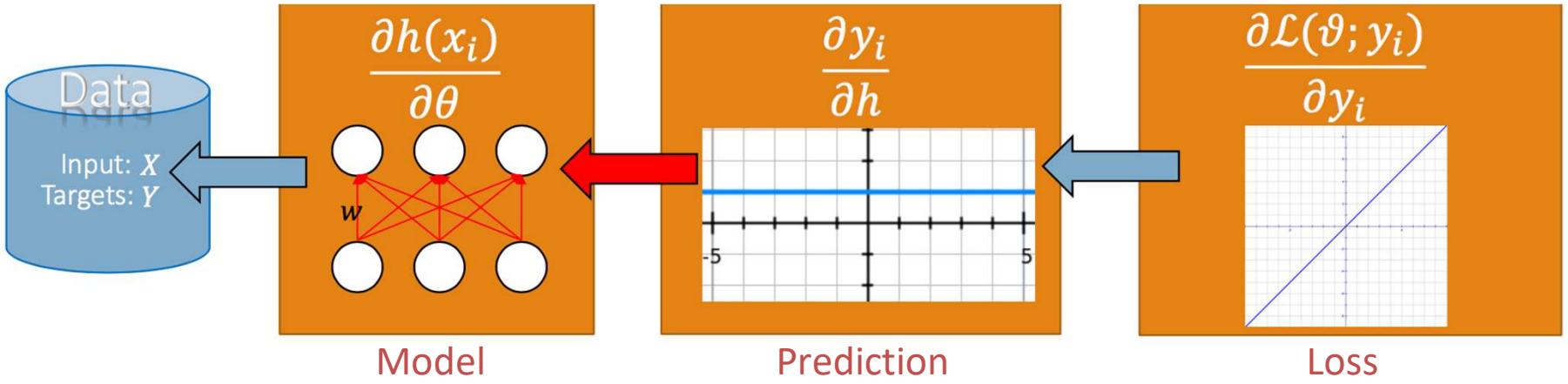
- *How do we train this network of units/functions/modules?*
- Propagate the gradient information and compute gradients at each unit (**chain rule**)



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

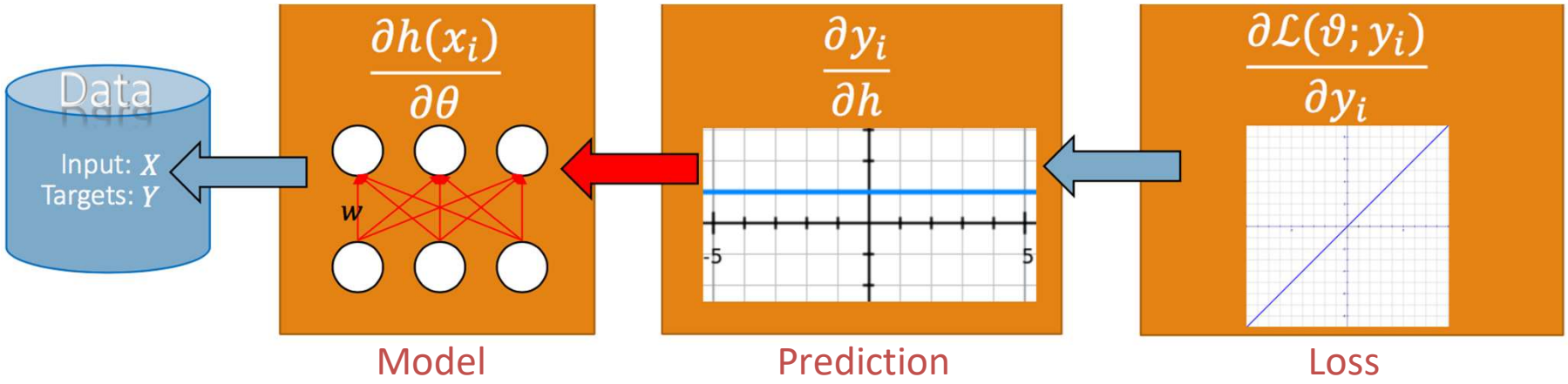
- *How do we train this network of units/functions/modules?*
- Update the model parameters using the compute gradients and the gradient descent rule



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

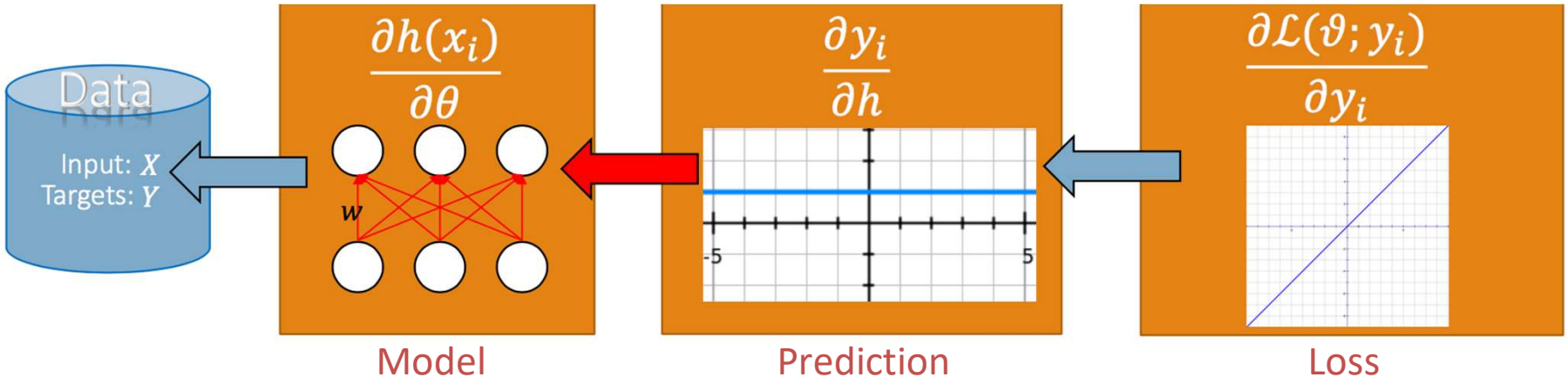
- *How do we train this network of units/functions/modules?*
- These are the **backpropagation** steps



Source: Efstratios Gavves' lecture slides

Forward and backward. Repeat.

- *How do we train this network of units/functions/modules?*
- Repeat **forward propagation** and **backpropagation** until "convergence"

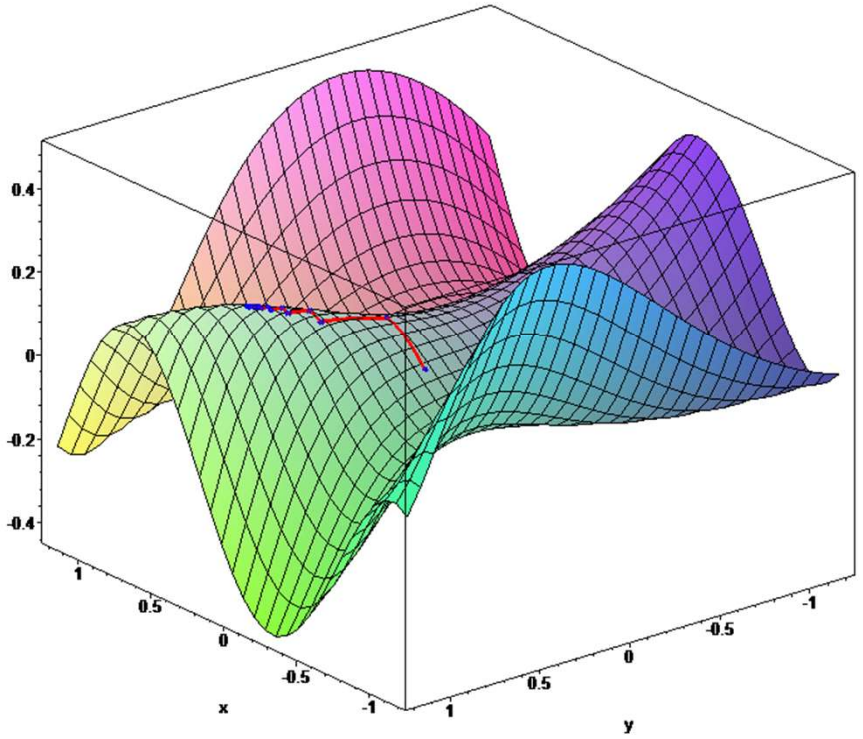


Source: Efstratios Gavves' lecture slides

Gradient descent

- Once we have the gradient we can update the weights using the formula

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$



Gradient descent

- Once we have the gradient we can update the weights using the formula

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_{w^{(t)}} L$$

- But how can we compute the gradient for such a complicated function?

$$\tilde{y} = h_N(h_{N-1}(h_{N-2}(...h_1(x; w_1); w_{N-2}); w_{N-1}); w_N)$$

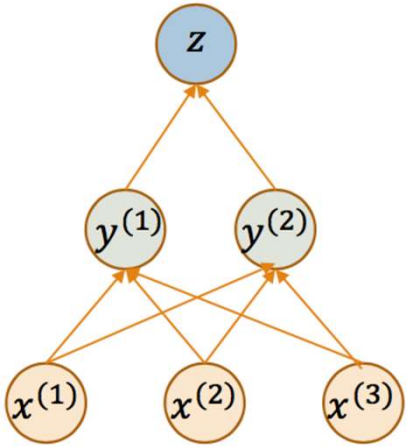
Backpropagation: chain rule

- Relatively straightforward application of the chain rule of calculus

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

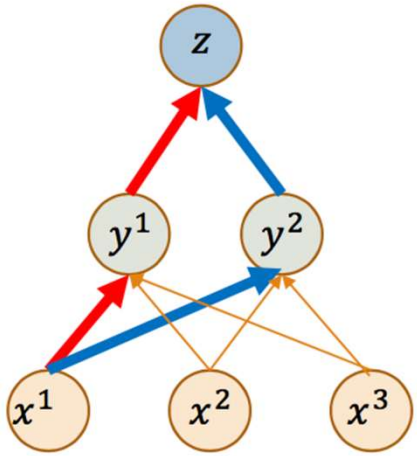
Backpropagation: chain rule

- Relatively straightforward application of the chain rule of calculus



Backpropagation: chain rule

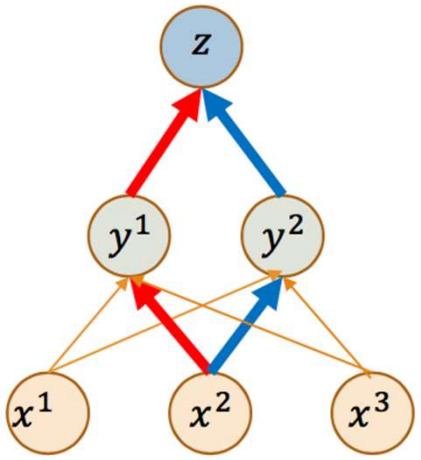
- Relatively straightforward application of the chain rule of calculus



$$\frac{dz}{dx^1} = \frac{dz}{dy^1}\frac{dy^1}{dx^1} + \frac{dz}{dy^2}\frac{dy^2}{dx^1}$$

Backpropagation: chain rule

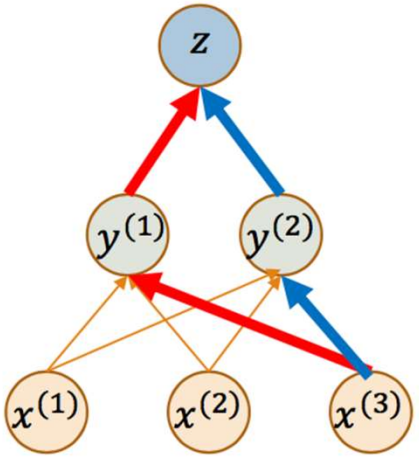
- Relatively straightforward application of the chain rule of calculus



$$\frac{dz}{dx^2} = \frac{dz}{dy^1}\frac{dy^1}{dx^2} + \frac{dz}{dy^2}\frac{dy^2}{dx^2}$$

Backpropagation: chain rule

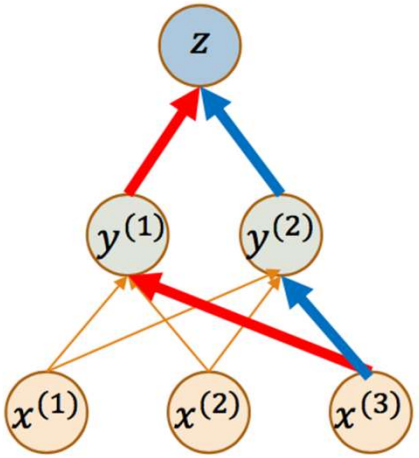
- Relatively straightforward application of the chain rule of calculus



$$\frac{dz}{dx^3} = \frac{dz}{dy^1}\frac{dy^1}{dx^3} + \frac{dz}{dy^2}\frac{dy^2}{dx^3}$$

Backpropagation: chain rule

- Relatively straightforward application of the chain rule of calculus



$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Backpropagation: chain rule

- Relatively straightforward application of the chain rule of calculus
- We're computing the gradient from all possible paths
- In vector notation, we're computing

$$\nabla_x z = \left(\frac{d\mathbf{y}}{d\mathbf{x}}\right)^T \nabla_y z$$

Backpropagation: chain rule

- Relatively straightforward application of the chain rule of calculus
- We're computing the gradient from all possible paths
- In vector notation, we're computing

$$\nabla_x z = \left(\frac{d\mathbf{y}}{d\mathbf{x}}\right)^T \nabla_y z$$

Gradient

Backpropagation: chain rule

- Relatively straightforward application of the chain rule of calculus
- We're computing the gradient from all possible paths
- In vector notation, we're computing

$$\nabla_x z = \left(\frac{d\mathbf{y}}{d\mathbf{x}}\right)^T \nabla_y z$$

Jacobian

https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant

Backpropagation: chain rule

We will continue to talk about Backpropagation in next lecture with more solid examples