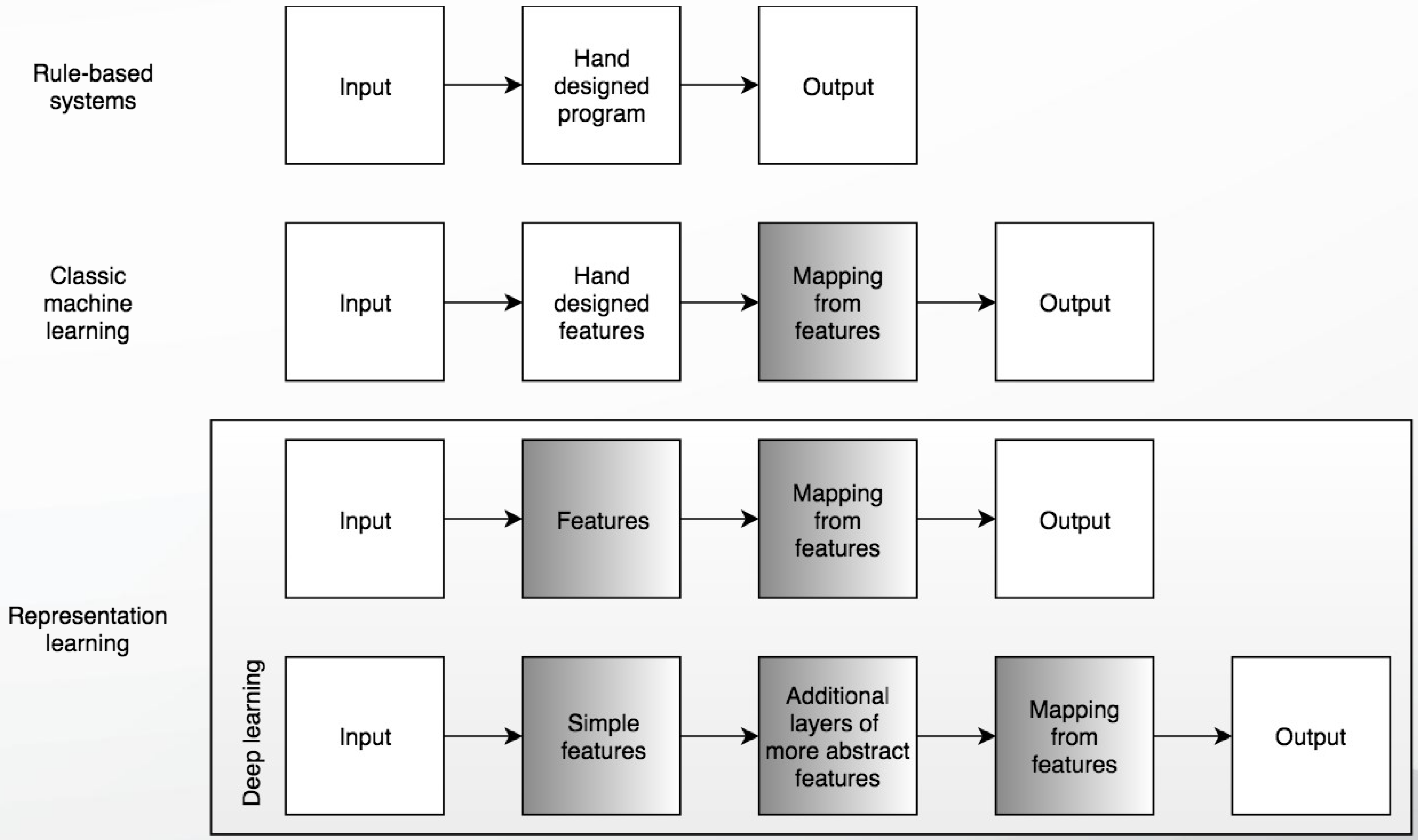
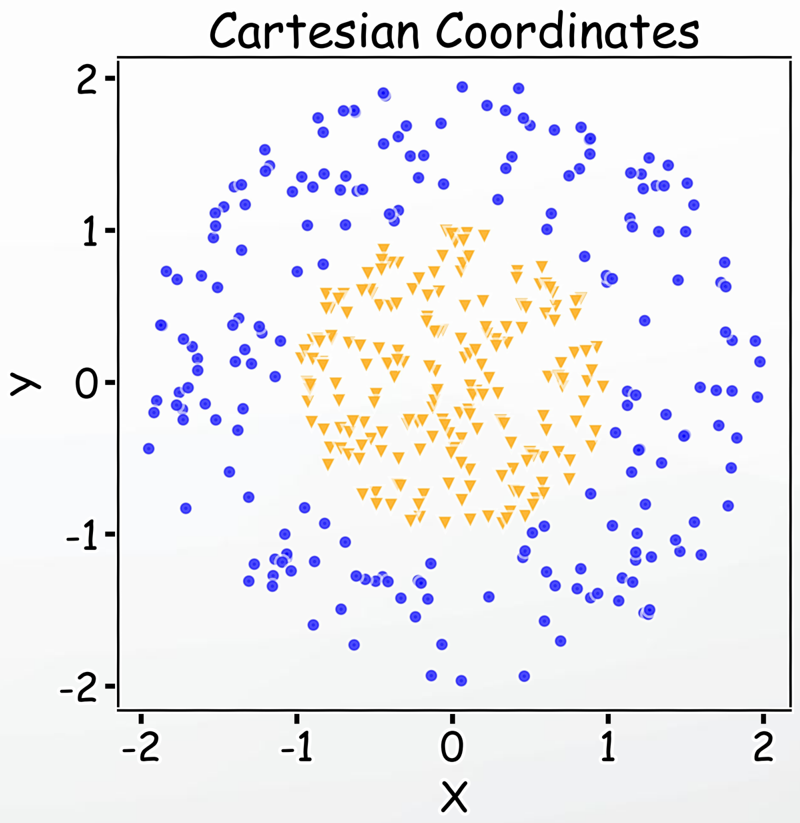
Artificial Neural Networks

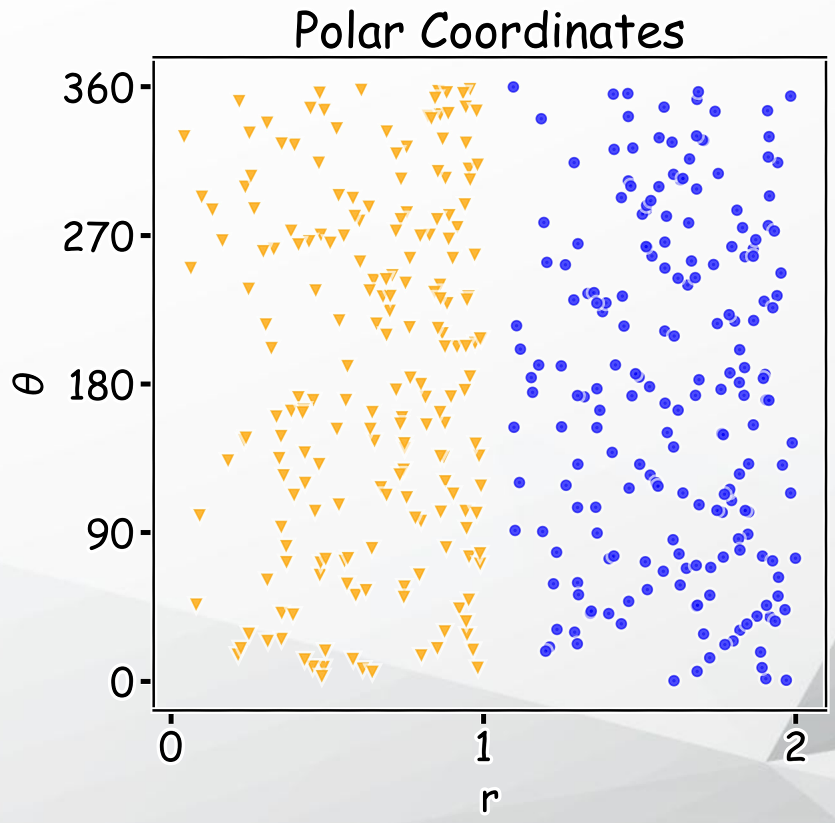
Learning multiple components



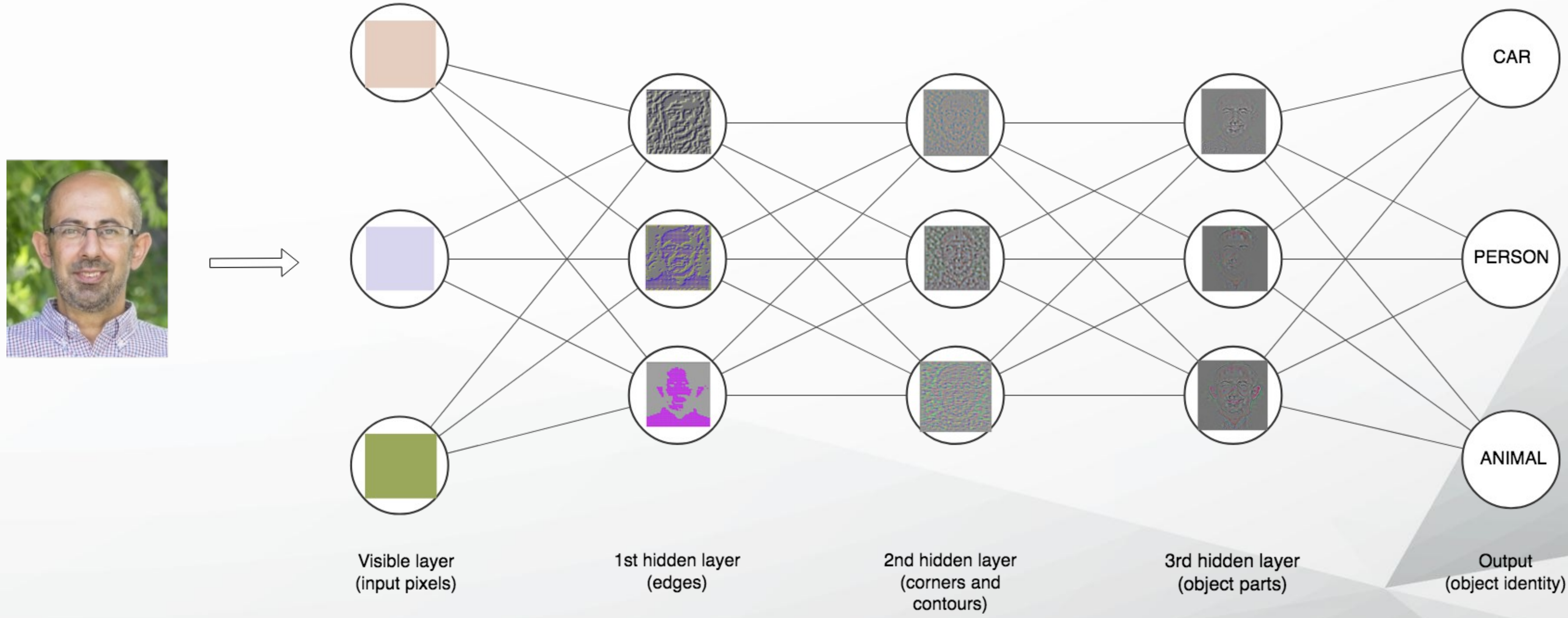
Representation

- First - Representation matters!





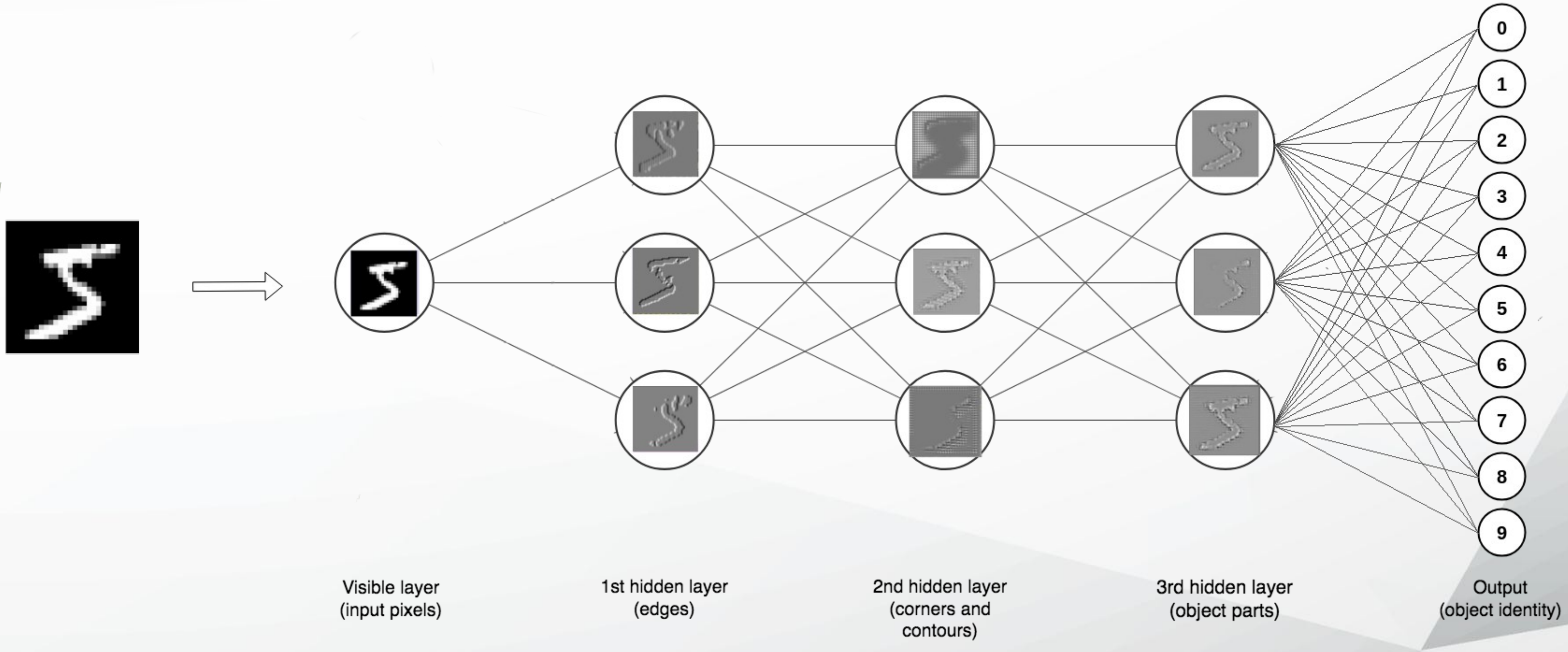
Depth – repeated compositions


CAR
PERSON
ANIMAL
Visible layer (input pixels)
1st hidden layer (edges)
2nd hidden layer (corners and contours)
3rd hidden layer (object parts)
Output (object identity)

Depth – repeated compositions

MNIST – dataset of hand written digits

Depth – repeated compositions



Common types of Neural Network Architectures

- Feedforward Neural networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Long Short-term Memory networks
- Autoencoders
- Generative Adversarial Networks

Feed-forward Neural Networks

- Simplest form of ANN:
  - the perceptrons are arranged in layers
    - the first layer is taking the inputs
    - the last layer is producing the outputs
    - between them there are hidden layers
- The data flow goes in one direction:
  - each perceptron is connected with every perceptron on the next layer
  - there is no connection between the perceptrons in the same layer

Features of feed-forward ANNs

- recall that a single perceptron can classify points into two regions that are linearly separable

Consider a more complex network

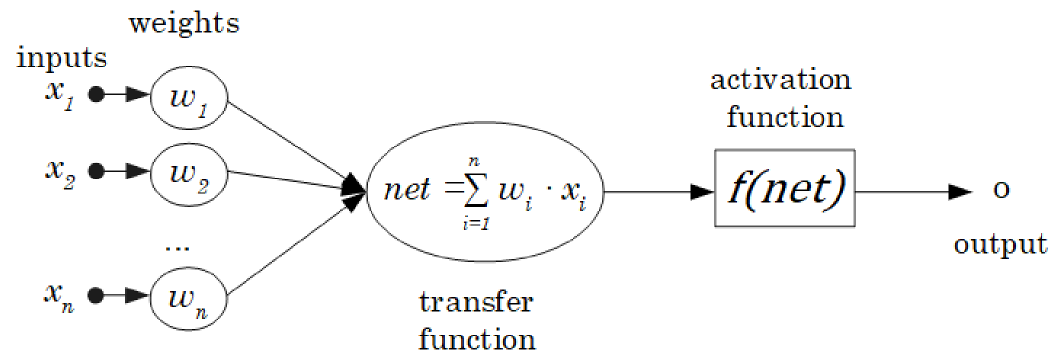
- with more perceptrons that are independent of each other in the hidden layer, the points are classified in more pairs of linearly separable regions, each of it having a unique line separating the region.

- by varying the number of nodes in the hidden layer, the number of layers, and the number of input and output nodes, one can classification of points in arbitrary dimension into an arbitrary number of groups

- hence feed-forward networks are commonly used for classification.

Artificial neural network – structure

a node's structure

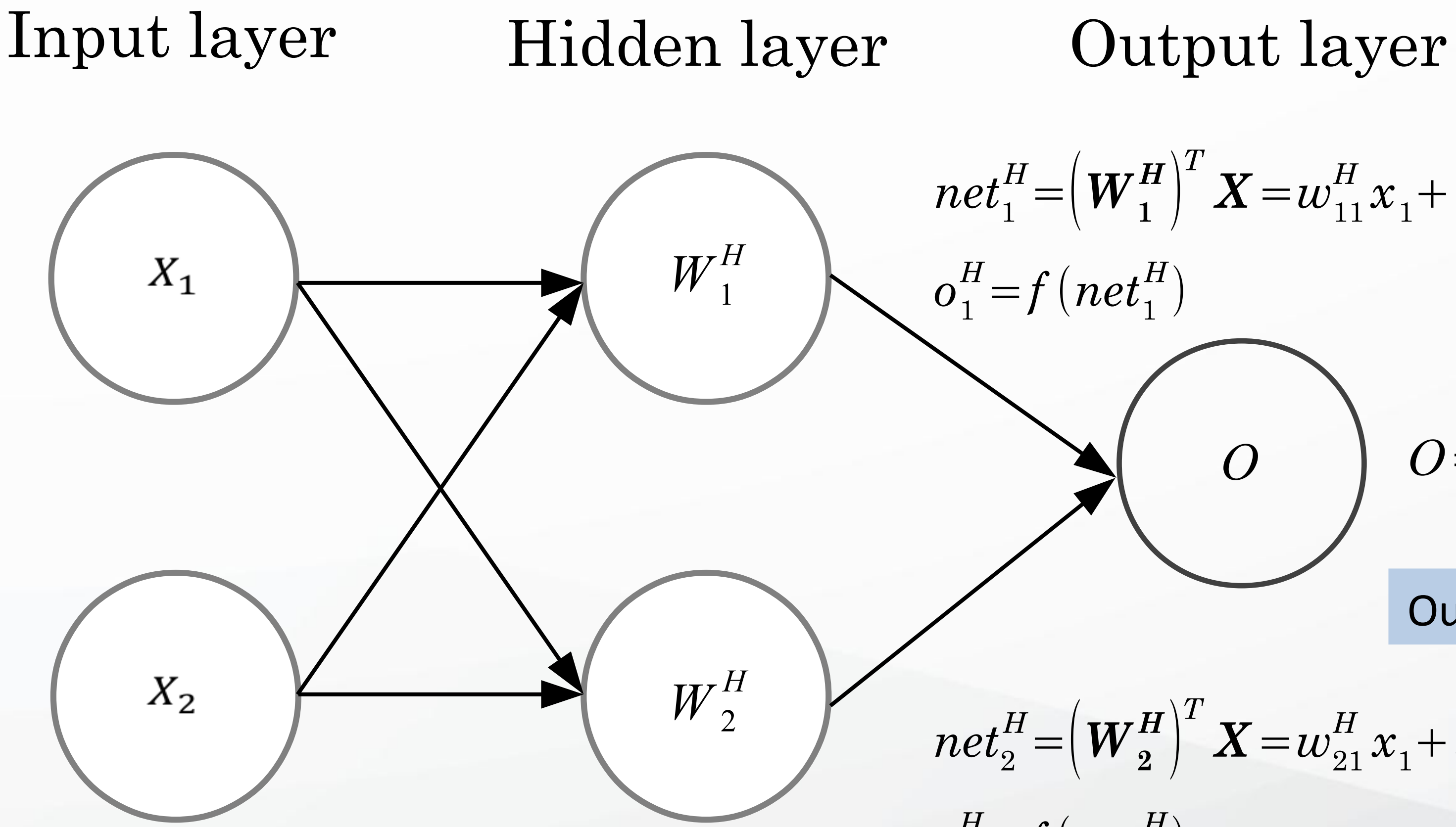


$$\boldsymbol{X} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}$$

$$\boldsymbol{W} = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix}$$

$$\boldsymbol{o} = f\left(\boldsymbol{X}^T \boldsymbol{W}\right)$$

Artificial neural network – structure



$$net_1^H = \left(\boldsymbol{W_1^H}\right)^T \boldsymbol{X} = w_{11}^H x_1 + w_{12}^H x_2$$

$$o_1^H = f\left(net_1^H\right)$$

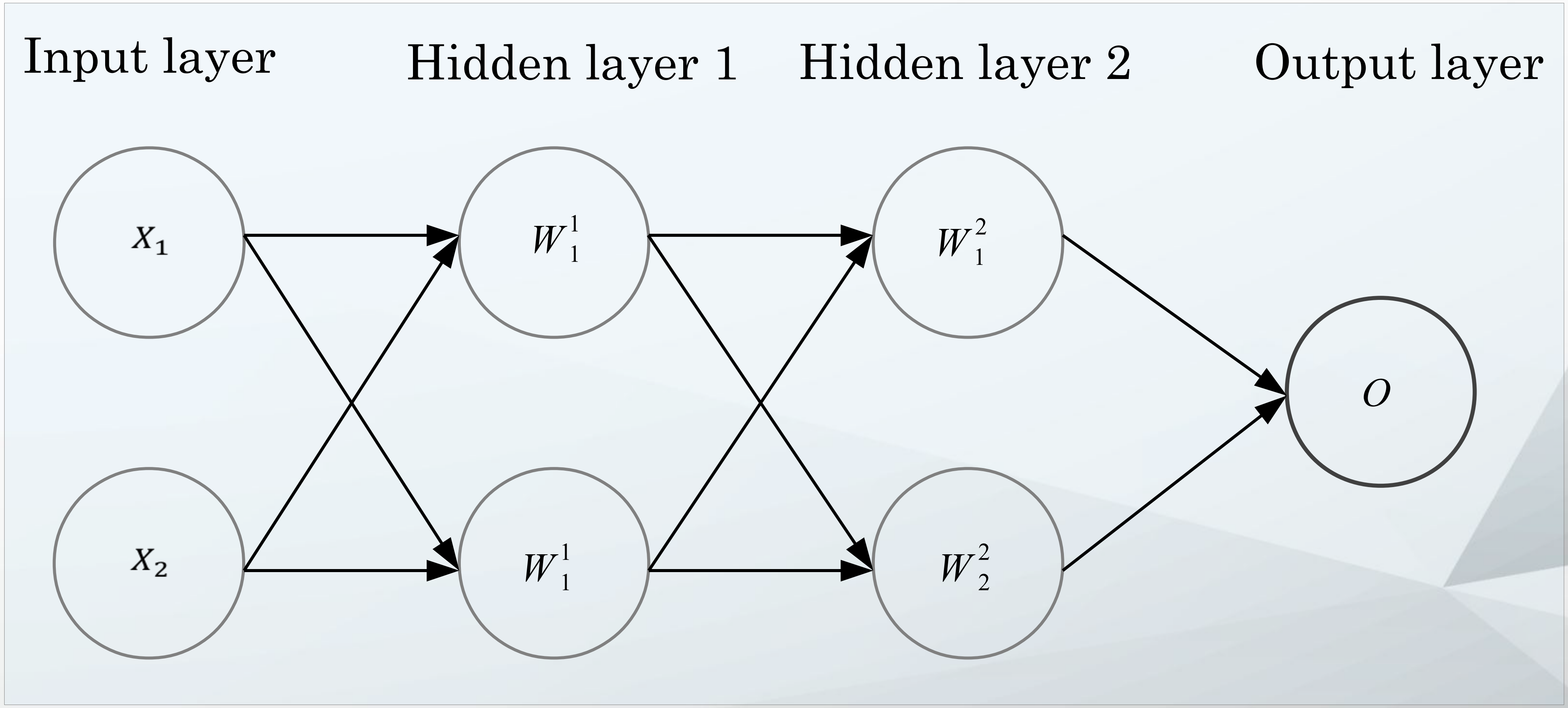
$$net_2^H = \left(\boldsymbol{W_2^H}\right)^T \boldsymbol{X} = w_{21}^H x_1 + w_{22}^H x_2$$

$$o_2^H = f\left(net_2^H\right)$$

$$\begin{tikzcd} O = g\left(o_1^H, o_2^H\right) \arrow[r] & Error = E(O, t) \\ \text{Output function} \arrow[u] & \text{Loss function} \arrow[u] \end{tikzcd}$$

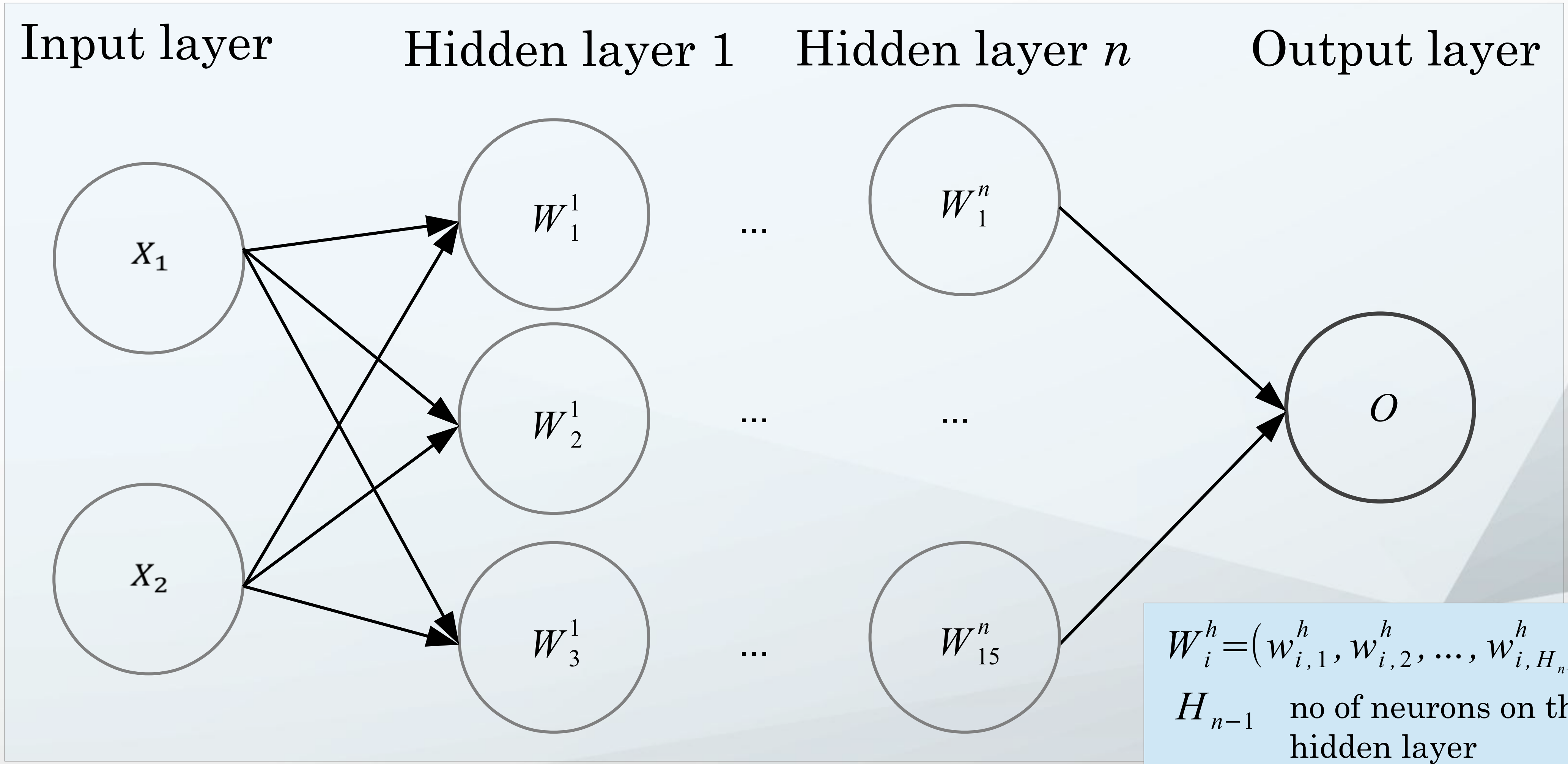
Artificial neural network – structure

Example of a **2 : 2 : 2 : 1** architecture with *2* hidden layers



Artificial neural network – structure

Example of a **2:3: … :15:1** architecture with *n* hidden layers



$W_i^h = (w_{i,1}^h, w_{i,2}^h, \ldots, w_{i,H_{n-1}}^h)$

H_{n-1} no of neurons on the *n -1* hidden layer

Design choices for feed-forward ANNs

Activation function

Loss function

Output units

Architecture

Linearity versus non-linearity

- Linear models:
  - Can be fit efficiently (via convex optimization)
  - Limited model capacity
- Alternative:

$$f(\boldsymbol{x}) = W^T \phi(\boldsymbol{x})$$

- Where $\phi(\boldsymbol{x})$ is a *non-linear transform*

Activation functions for ANNs

The activation function should:

- Provide **non-linearity**
- Ensure **gradients remain large** through hidden unit

Common choices are

- Sigmoid
- Relu, leaky ReLU, Generalized ReLU, MaxOut
- Softplus
- Tanh
- Swish

Traditional ANNs (like feed-forward)

- Manually engineer
  - Domain specific, enormous human effort
- Generic transform
  - Maps to a higher-dimensional space
  - Kernel methods: e.g. RBF kernels
  - Over fitting: does not generalize well to test set
  - Cannot encode enough prior information

Deep Learning

- Directly learn ϕ

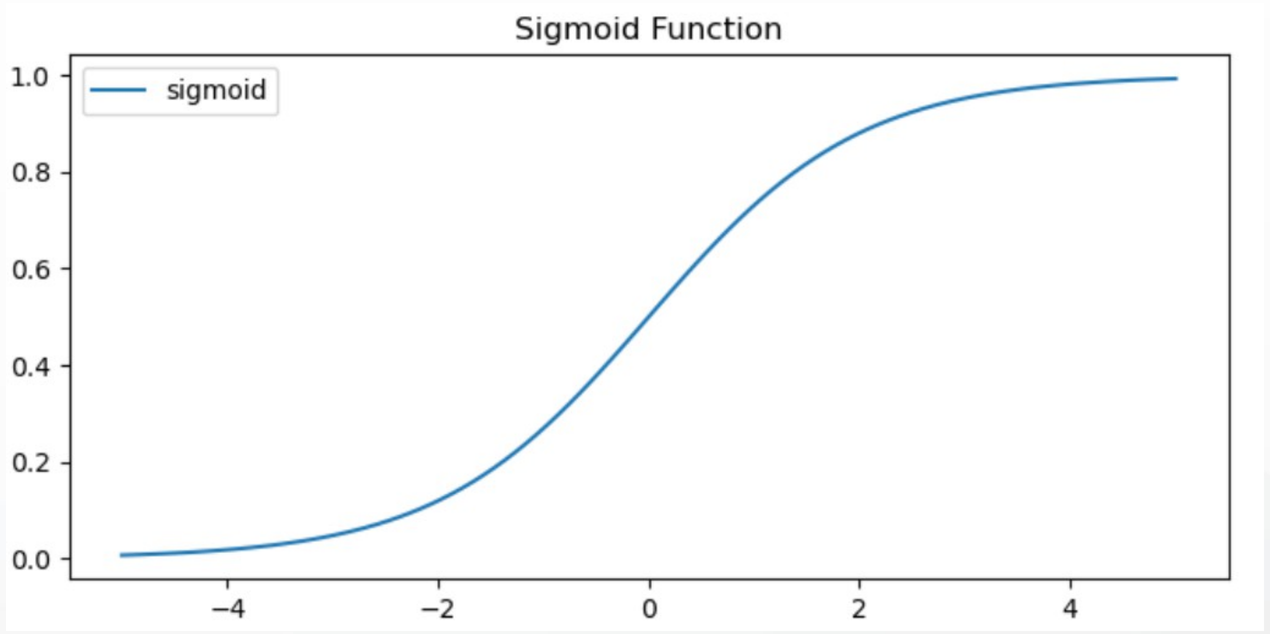
$$f(\boldsymbol{x}, \eta) = W^T \phi(\boldsymbol{x}, \eta)$$

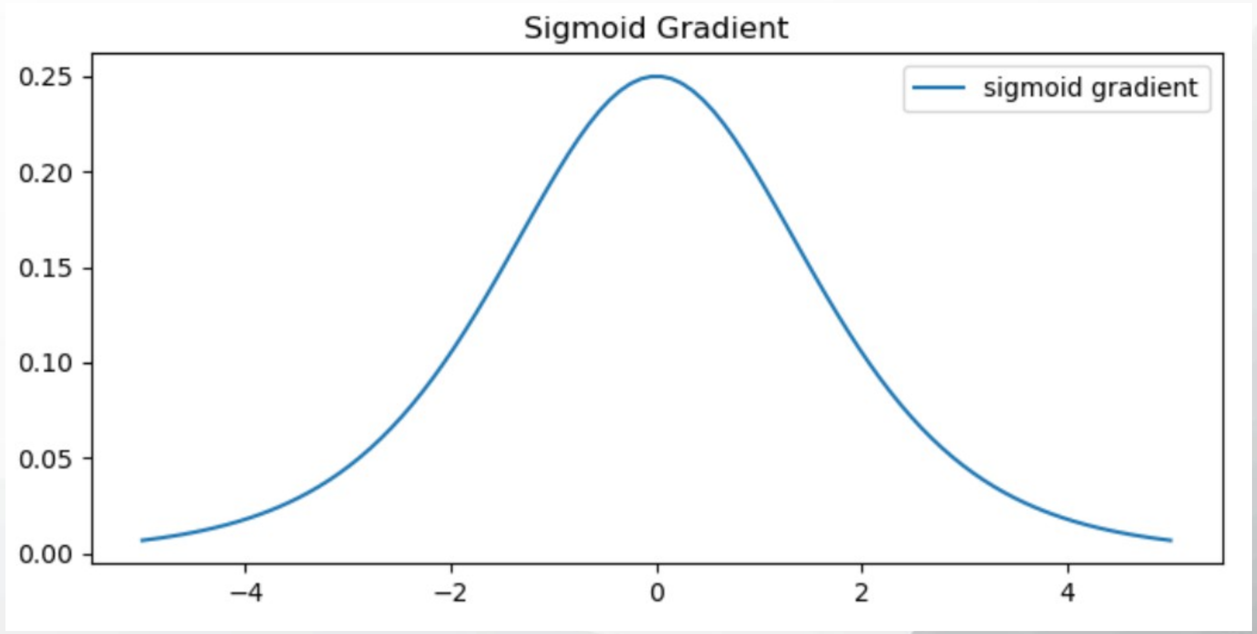
  - $\phi(\boldsymbol{x}, \eta)$ is an automatically-learned **representation** of x
- For **deep networks**, ϕ is the function learned by the **hidden layers** of the network
- η are the learned weights

  Non-convex optimization
- Can encode prior beliefs, generalizes well

Sigmoid (aka Logistic)

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

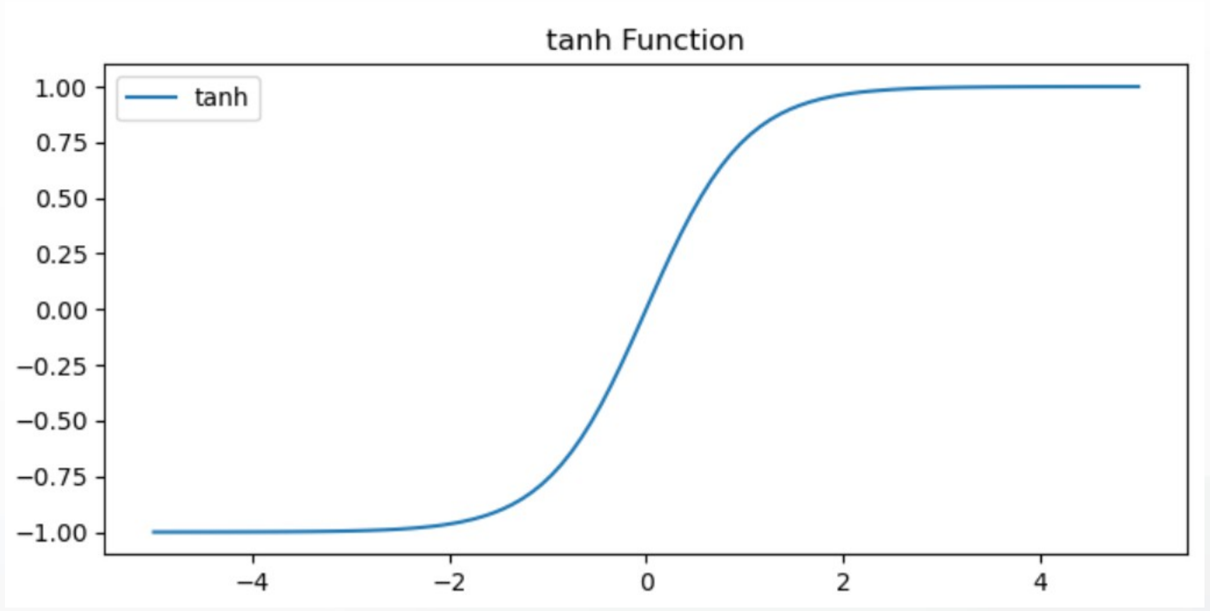


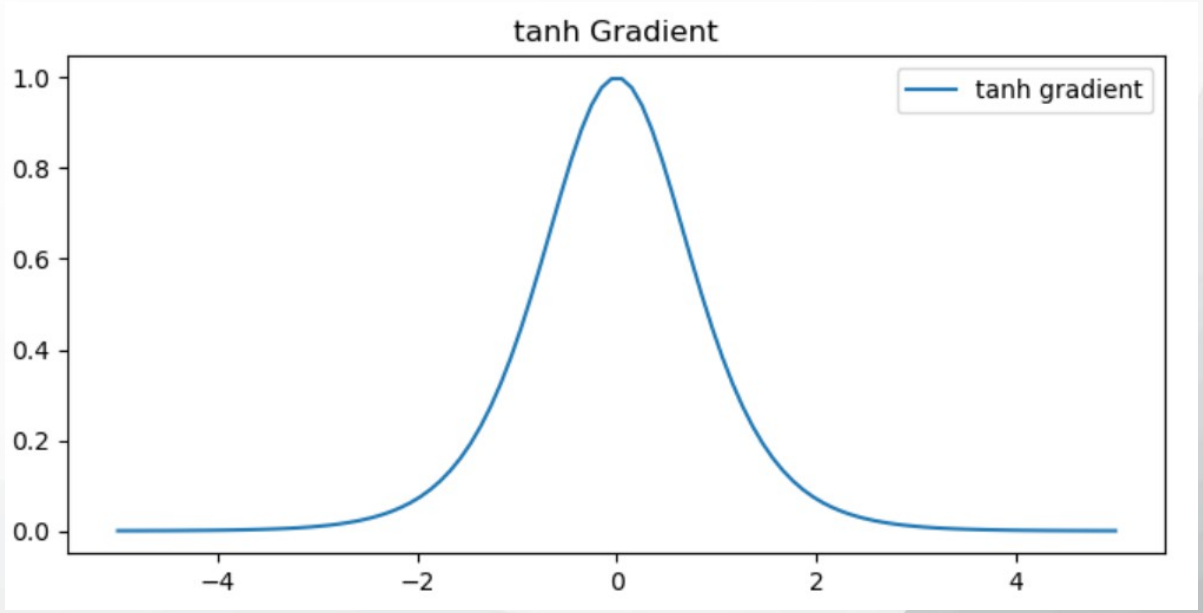


Derivative is **zero** for much of the domain. This leads to “vanishing gradients” in backpropagation.

Hyperbolic tangent (aka tanh)

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

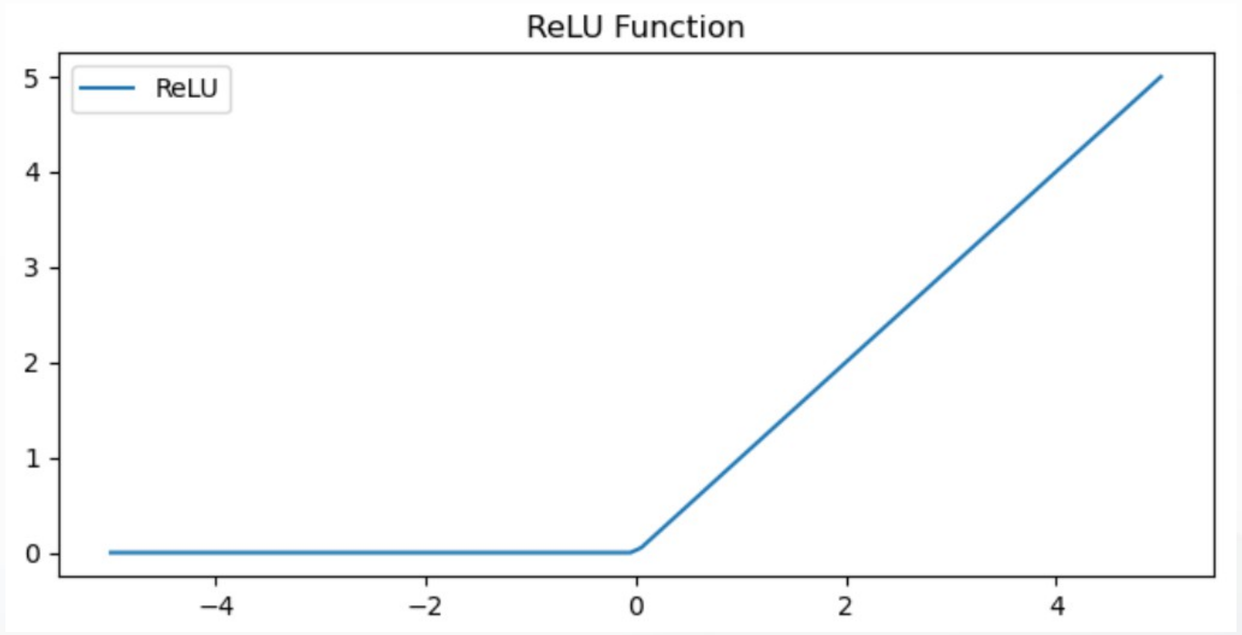


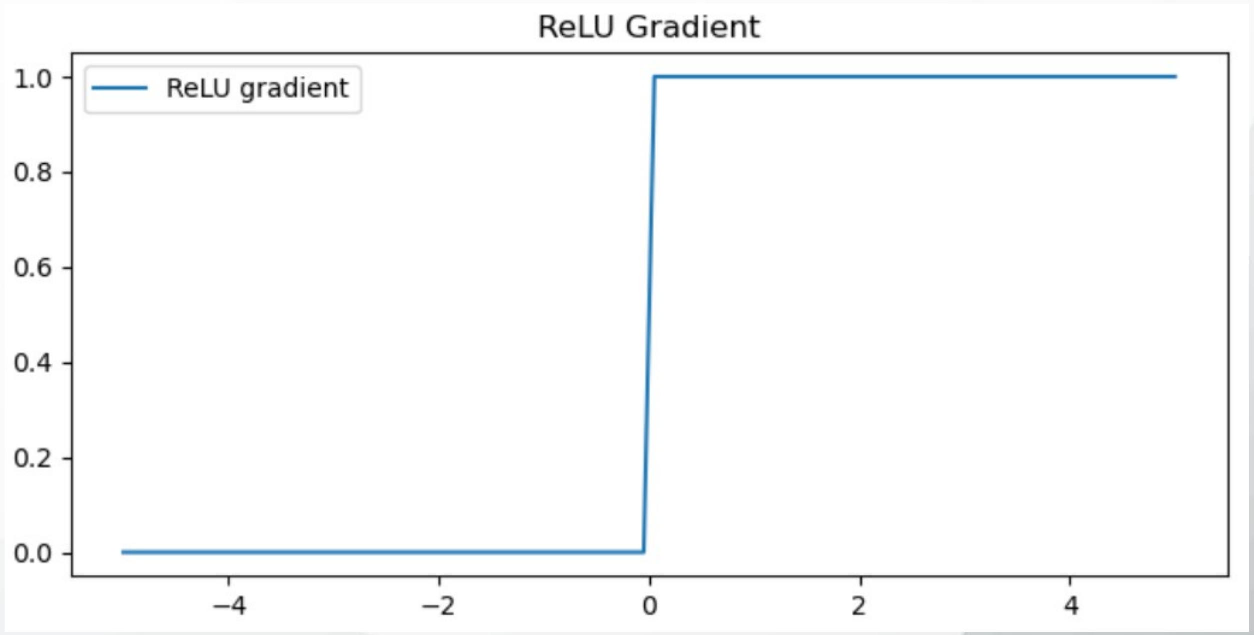


Derivative is also **zero** for much of the domain.

Rectified Linear Unit (ReLU)

$$y = max(0, x)$$





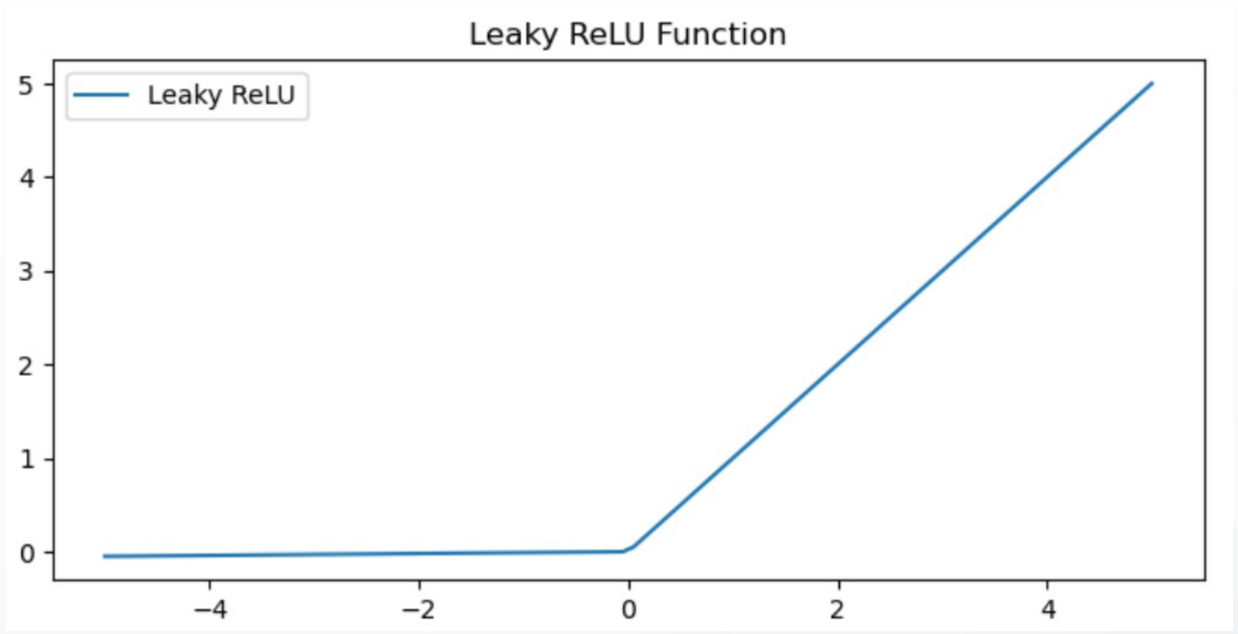
Two major advantages:

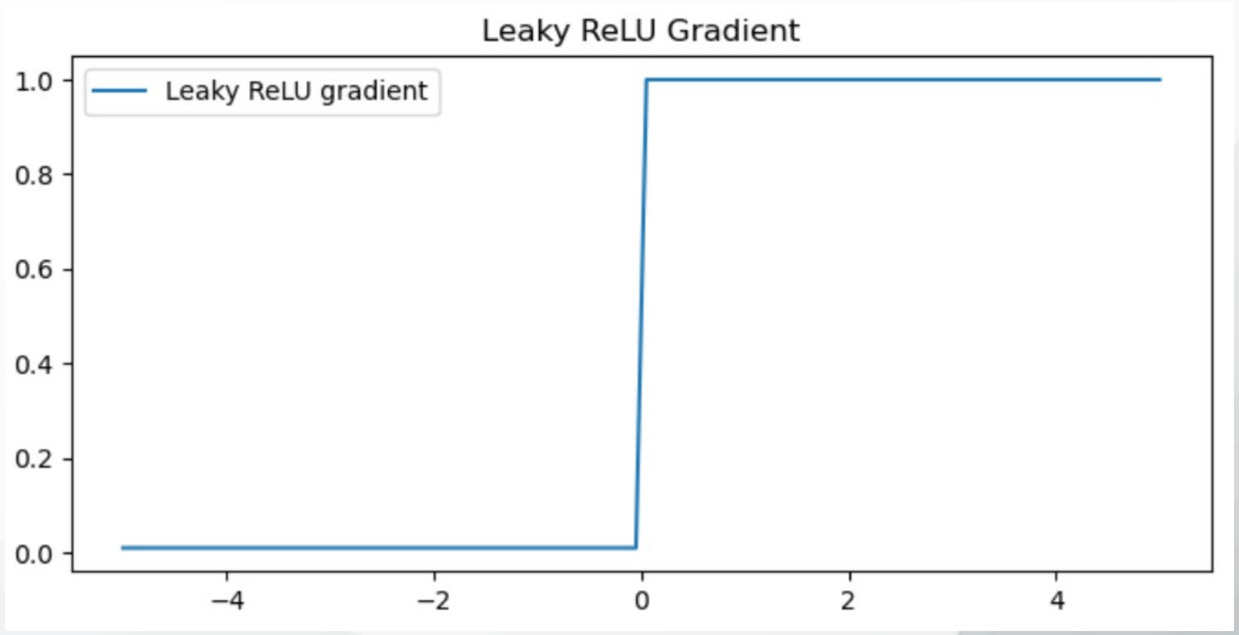
1. No vanishing gradient when $x > 0$
2. Provides sparsity (regularization) since $y = 0$ when $x < 0$

Leaky ReLU

$$y = max(0, x) + min(\alpha x, 0)$$

α *is a small positive value*

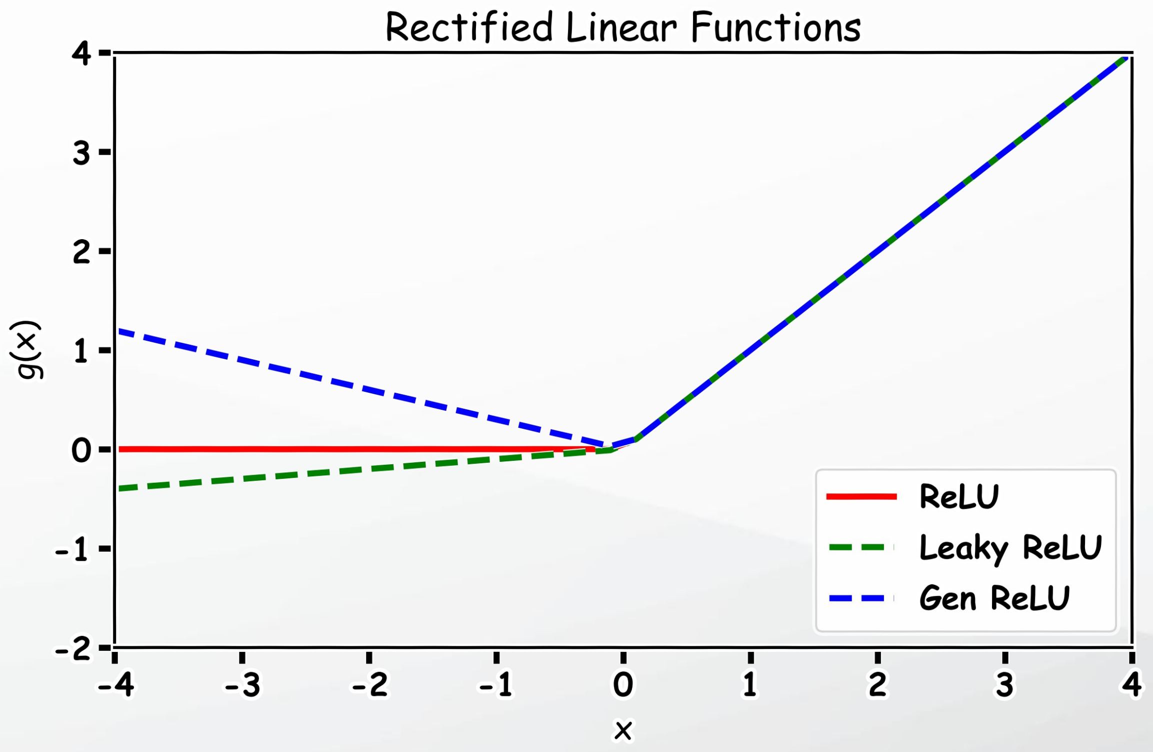




- Tries to fix "dying ReLU" problem: derivative is non-zero everywhere.
- Some people report success with this form of activation function, but the results are not always consistent

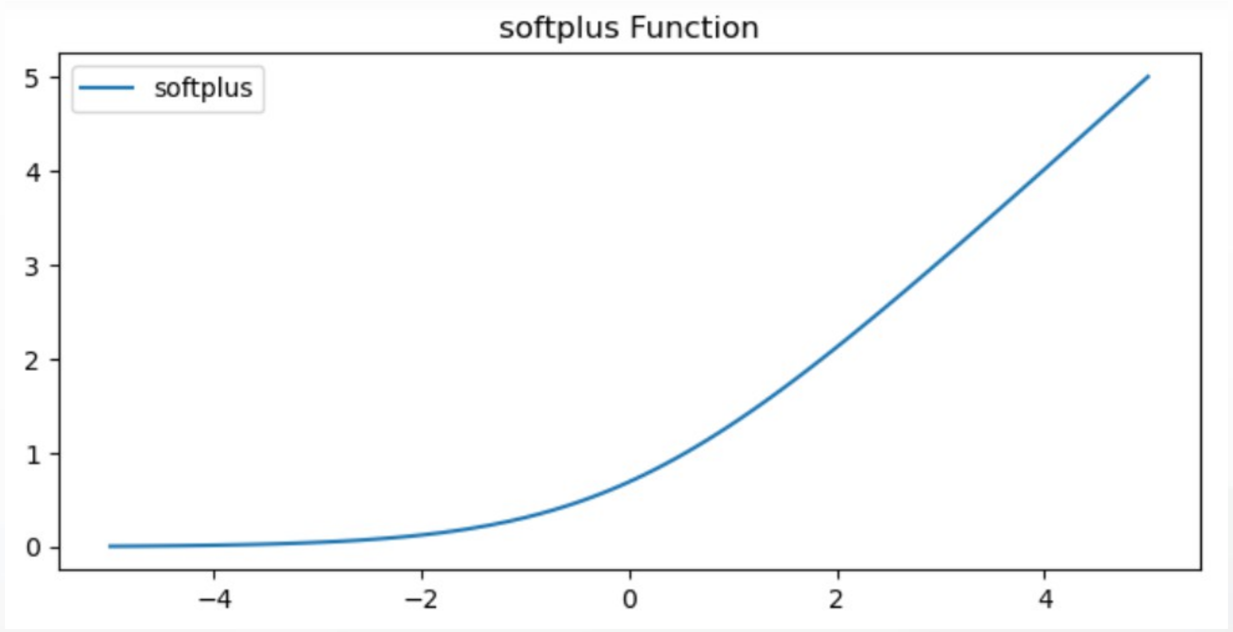
Generalized ReLU

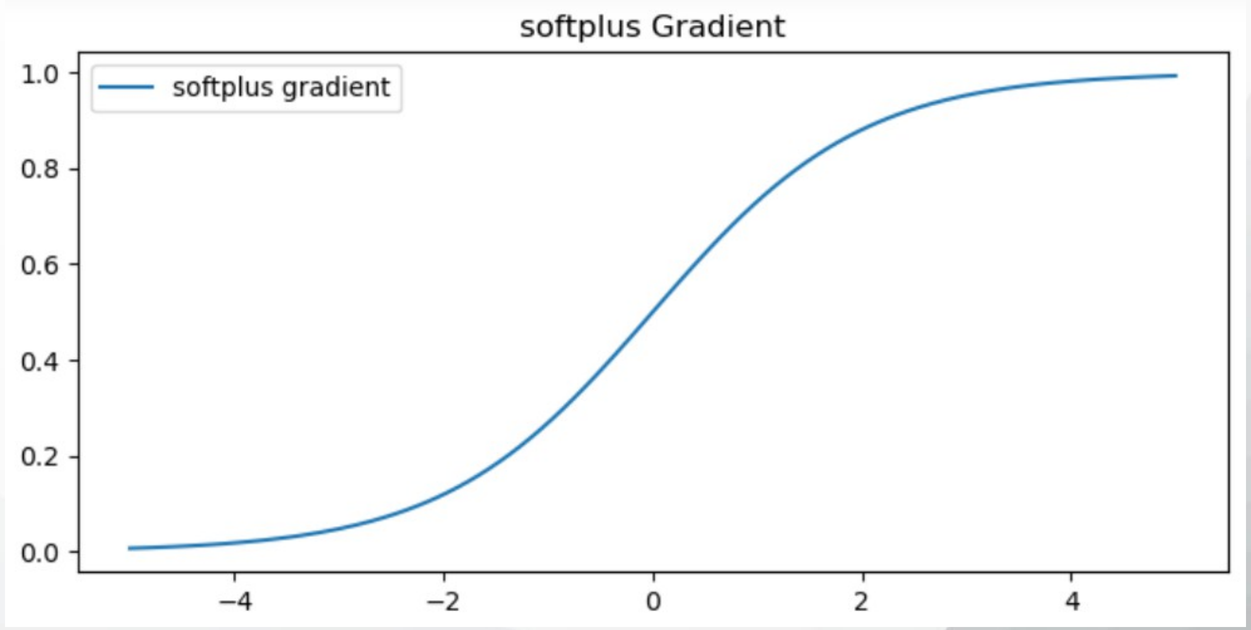
$$y = max(0, x) + \alpha\, min(x, 0)$$



Softplus function

$$y = \log\left(1 + e^{x}\right)$$

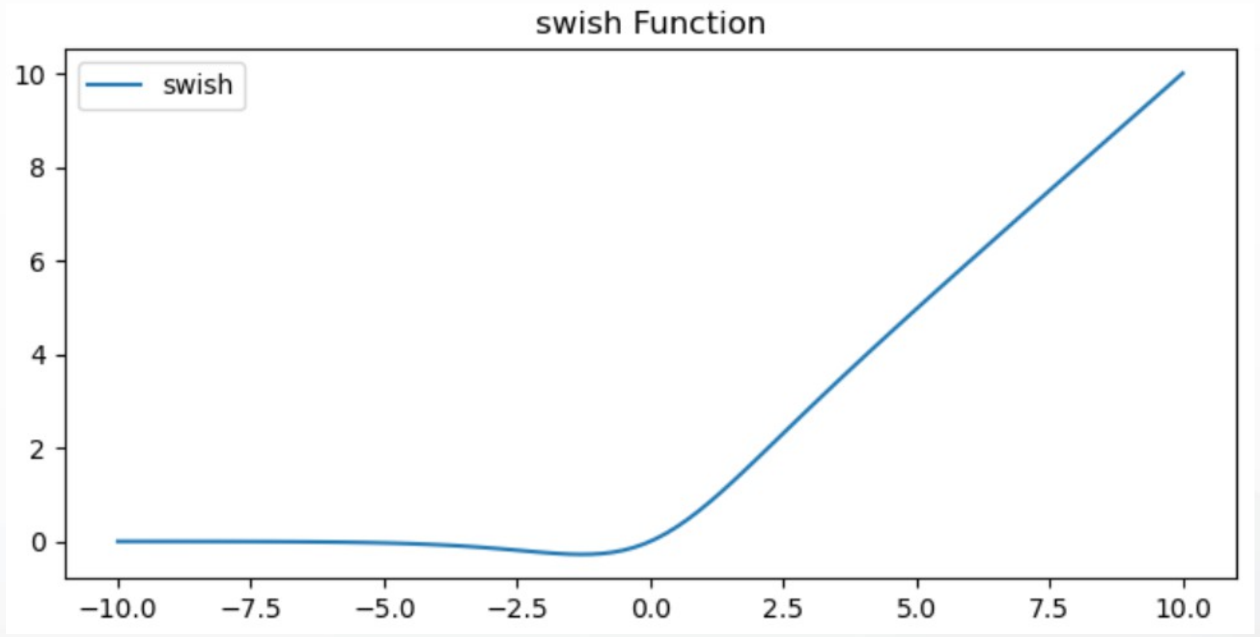


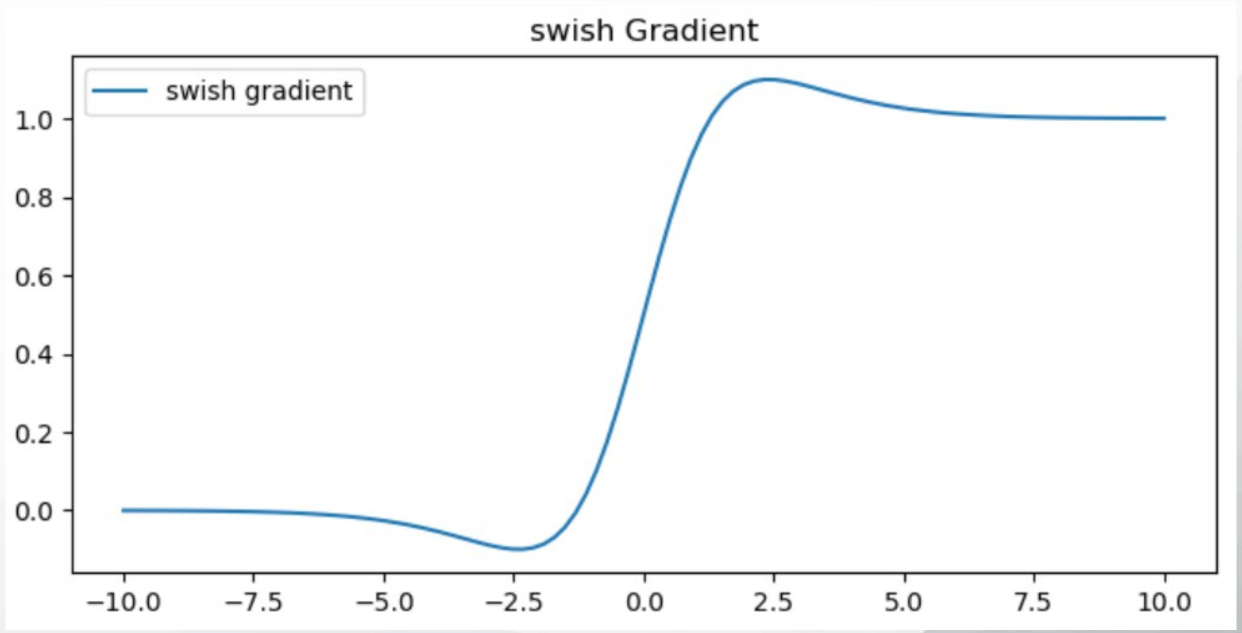


Outputs produced by sigmoid and tanh functions have upper and lower limits whereas softplus function produces outputs in scale of *(0, +∞)*.

Swish function

$$y = x\ sigmoid(x)$$





Loss functions for ANNs

- Likelihood for a given point
- Assume independence – likelihood for all measurements
- Maximize the likelihood, or equivalently maximize the log-likelihood
- Turn all this into a loss function

Common loss functions for ANNs

Mean Absolute Error Loss

Mean Squared Error Loss

Negative Log-Likelihood Loss

Cross-Entropy Loss

Hinge Embedding Loss

…

Mean Absolute Error Loss (MAE)

also called L1 Loss

- computes the average of the sum of absolute differences between actual values and predicted values.

$$loss(x, y) = |x - y|$$

- *x* represents the actual value and *y* the predicted value.

used for:

- regression problems
  - especially when the distribution of the target variable has outliers, such as small or big values that are a great distance from the mean value.

Mean Squared Error Loss (MSE)

also called L2 Loss

- computes the average of the squared differences between actual values and predicted values

$$loss(x, y) = (x - y)^2$$

- *x* represents the actual value and *y* the predicted value.

MSE is the default loss function for most regression problems.

Cross-Entropy Loss

Is the difference between two probability distributions for a provided set of occurrences or random variables.

In the discrete setting, given two probability distributions p and q, their cross-entropy is defined as

$$H(p,q) = -\sum_{x \in X} p(x) \log(q(x))$$

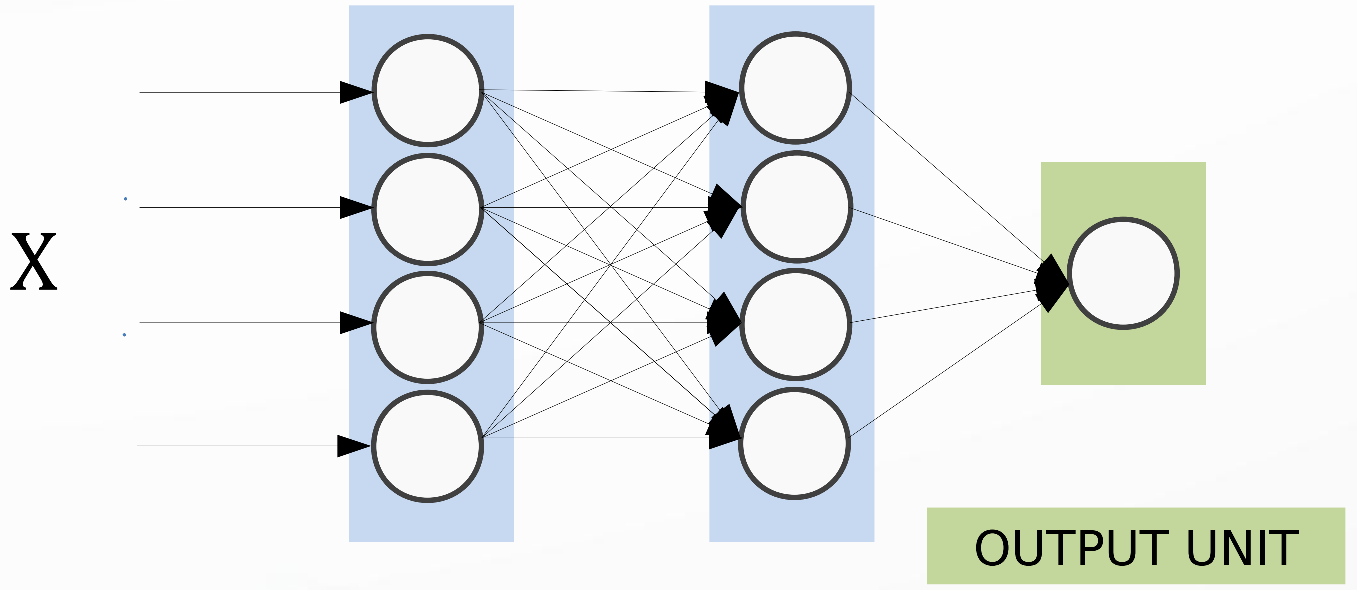
To compute the loss we apply the cross-entropy operator between the desired output and to the real output of our model after we used the softmax operator on the output.

Used in classifications

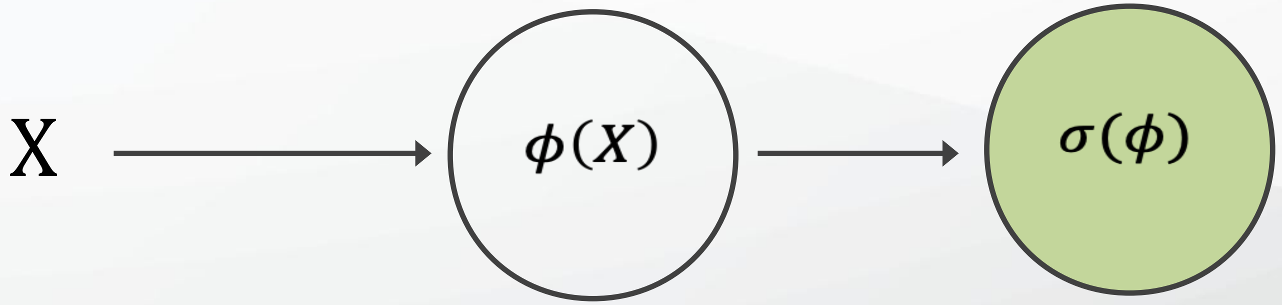
Output functions

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	GANS

Output unit for binary classification



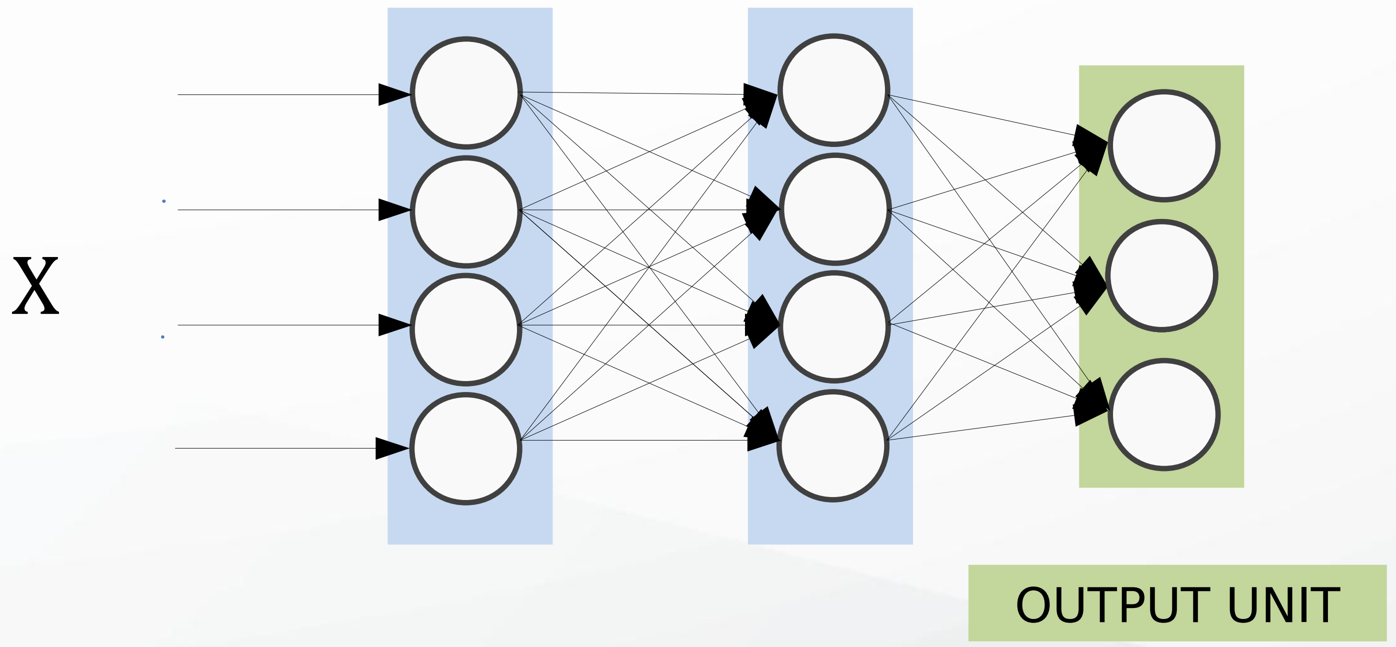
$\hat{Y} = \mathrm{P}(y = 1)$



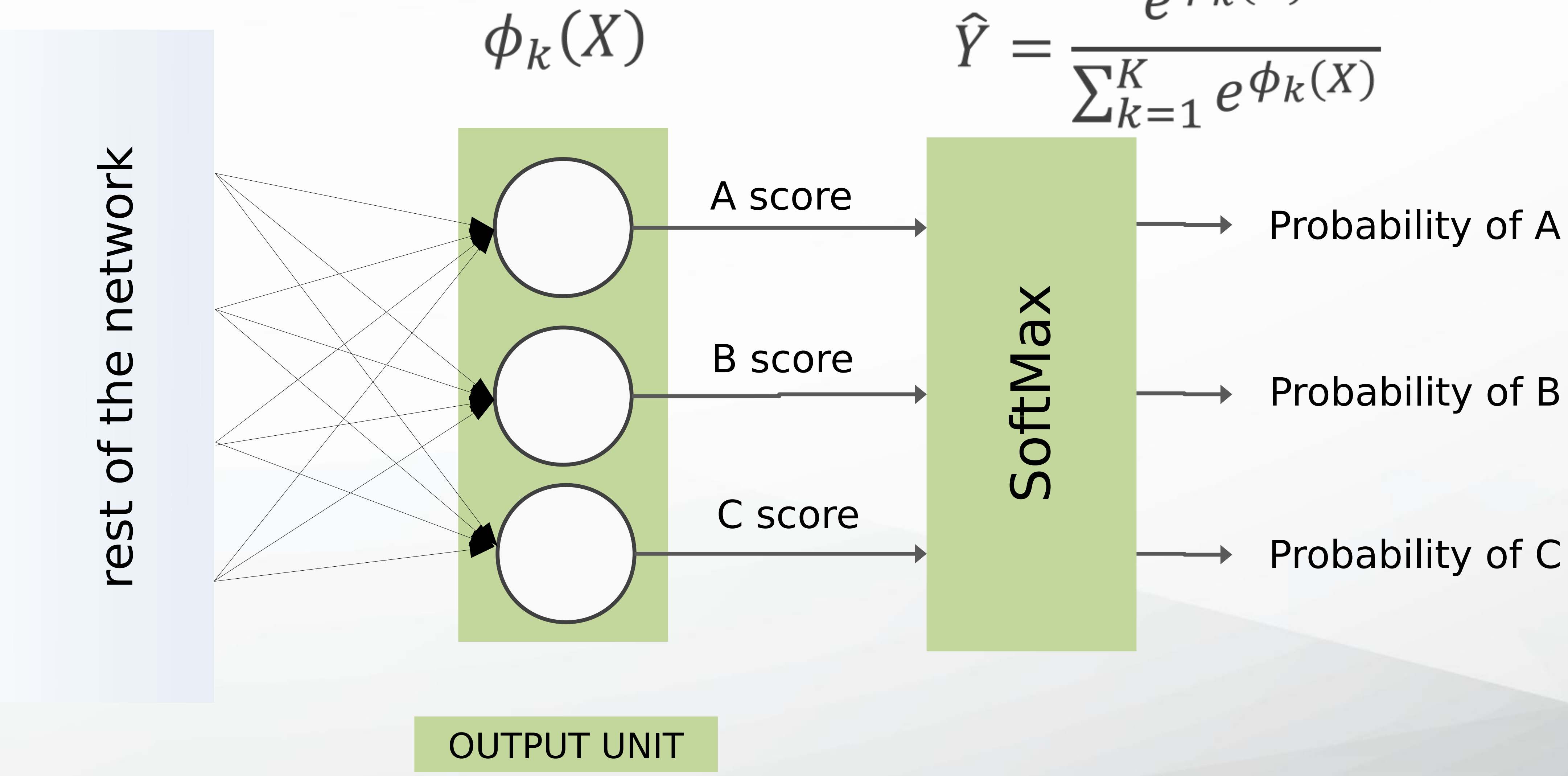
$\hat{Y} = \mathrm{P}(y = 1)$

$$X \Longrightarrow \phi(X) \Longrightarrow P(y = 1) = \frac{1}{1 + e^{-\phi(X)}}$$

Output unit for multi class classification



Softmax

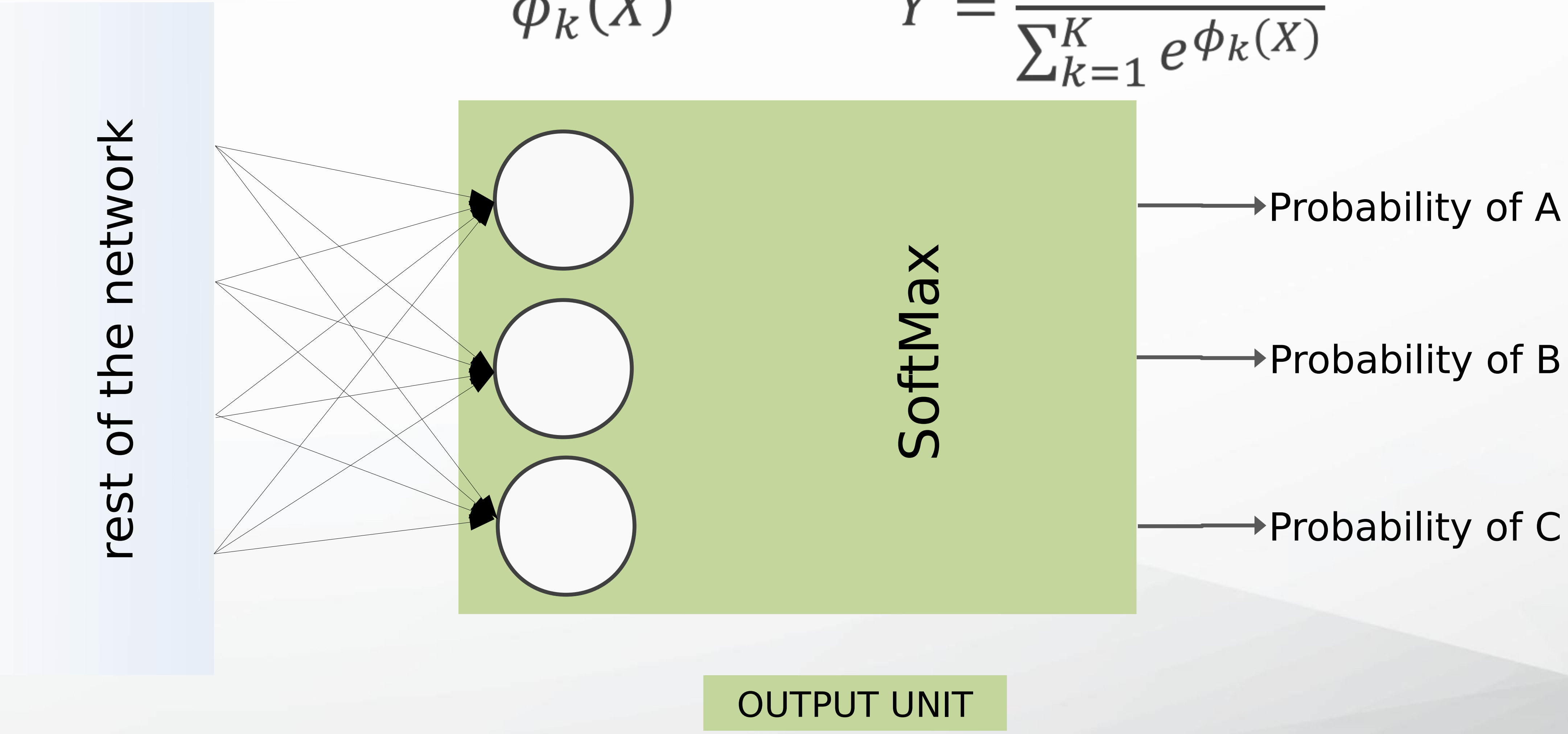


$$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^{K} e^{\phi_k(X)}}$$

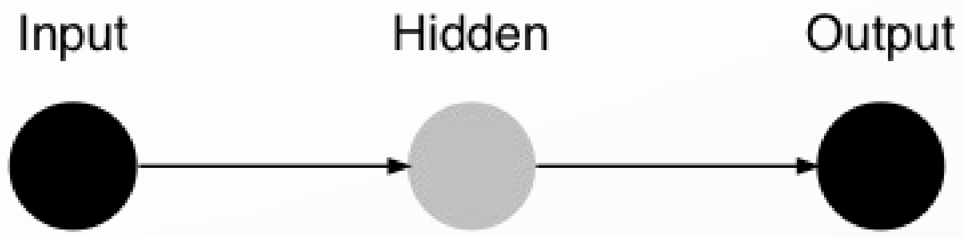
Softmax

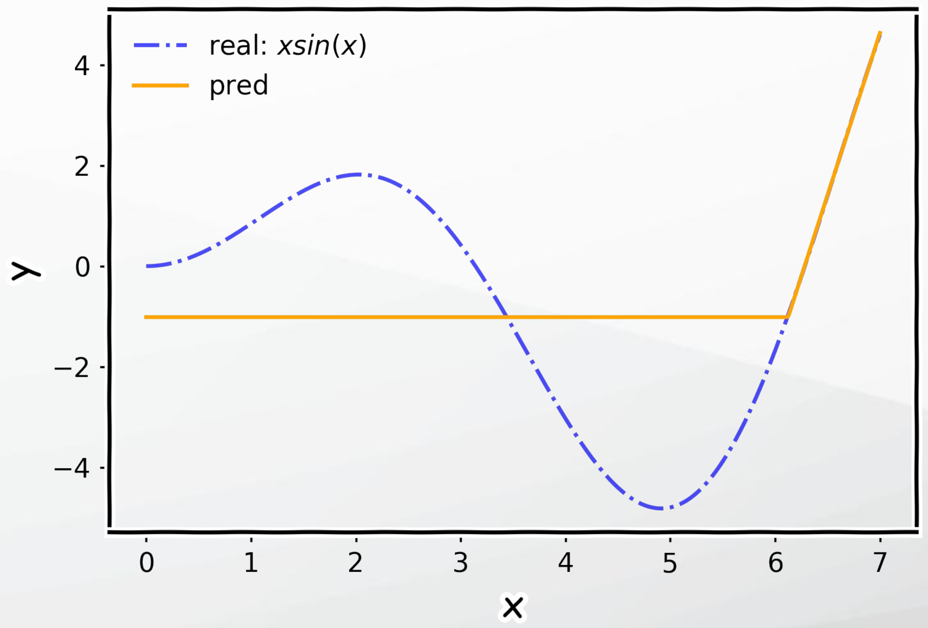
$\phi_k(X)$

$$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^{K} e^{\phi_k(X)}}$$

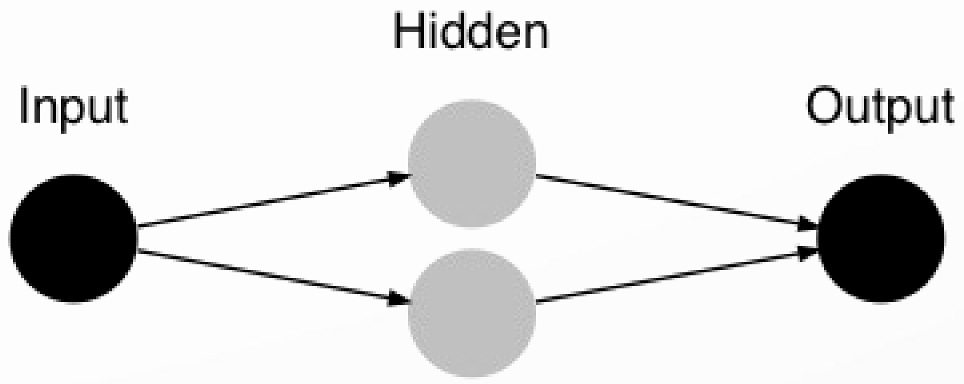


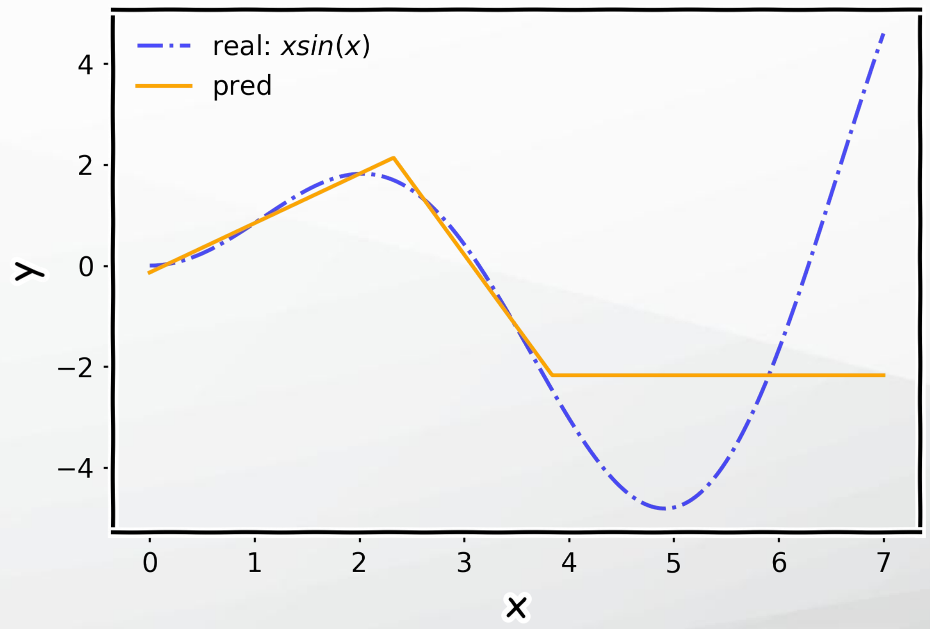
Architecture – performance example



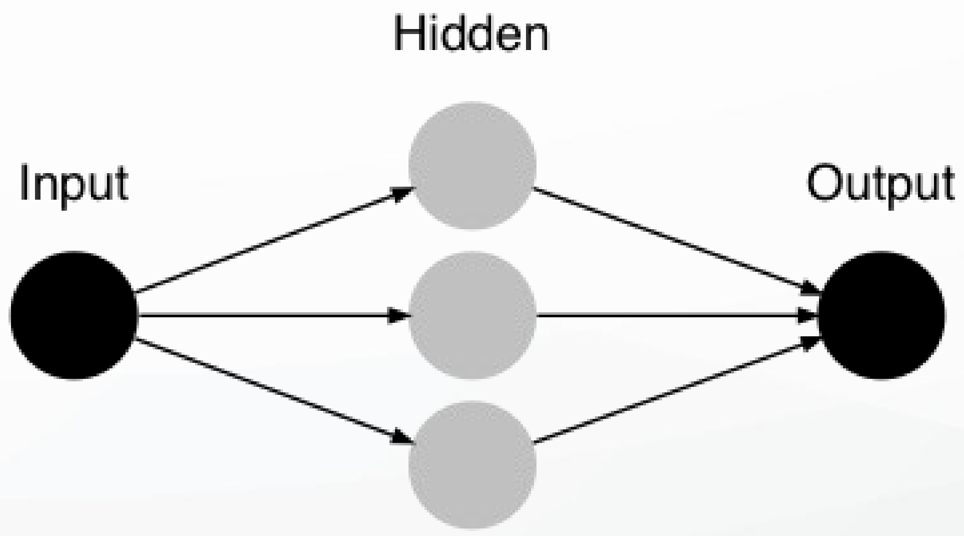


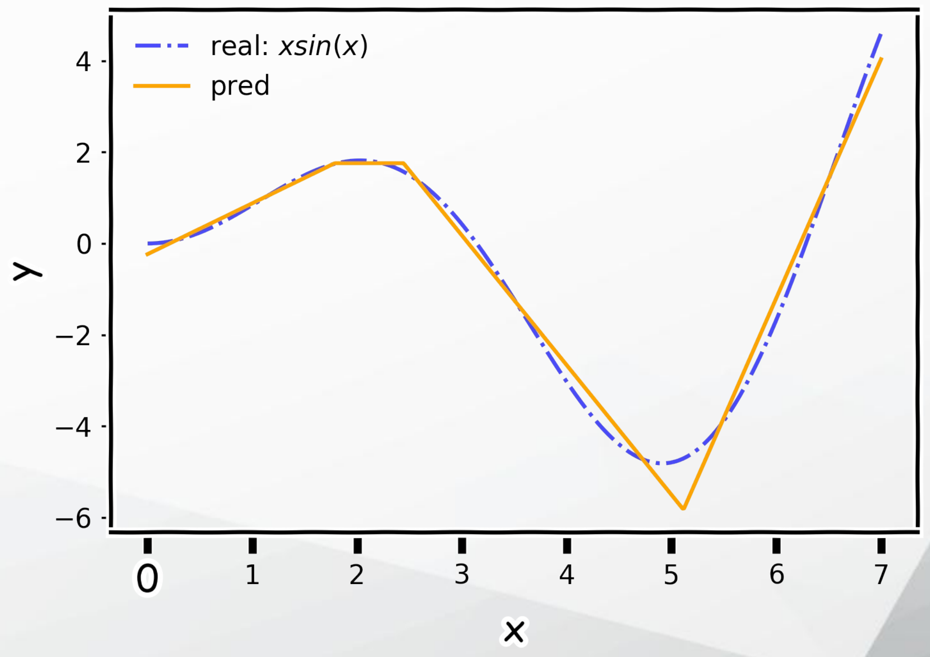
Architecture – performance example



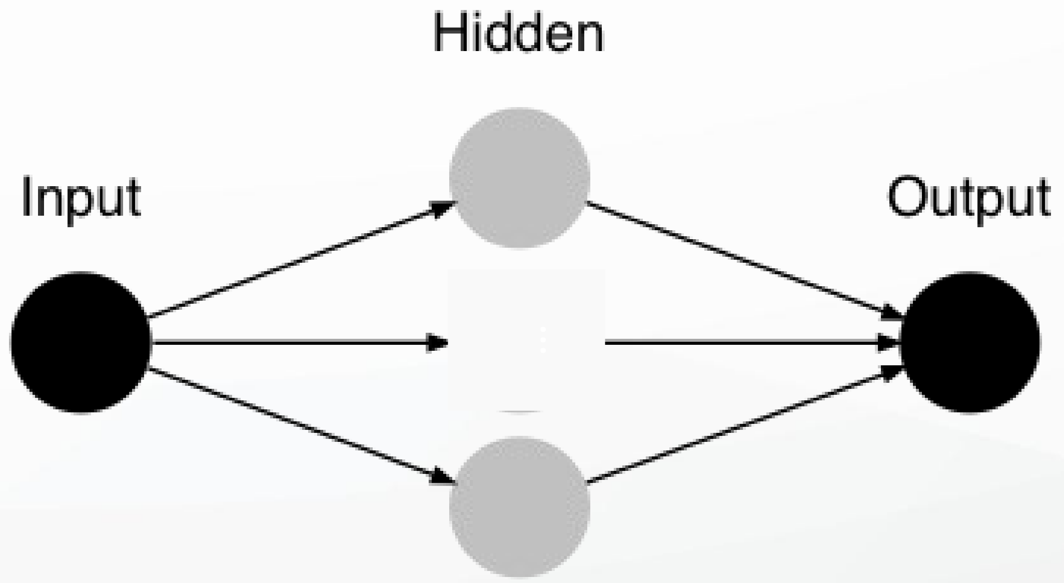


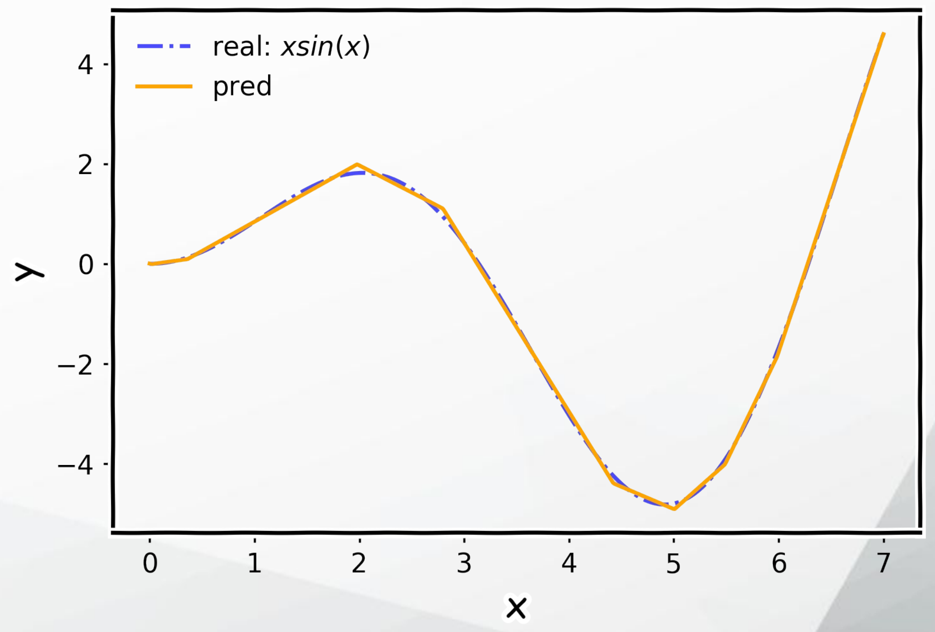
Architecture – performance example



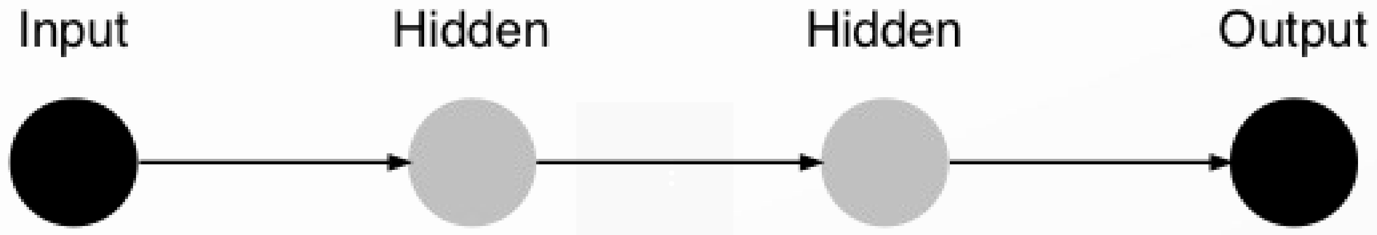


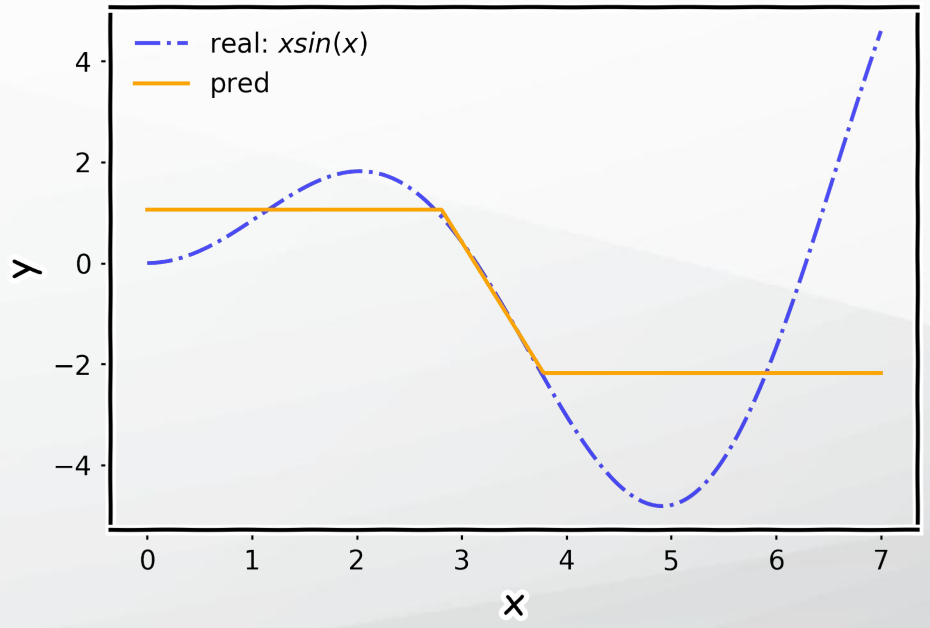
Architecture – performance example



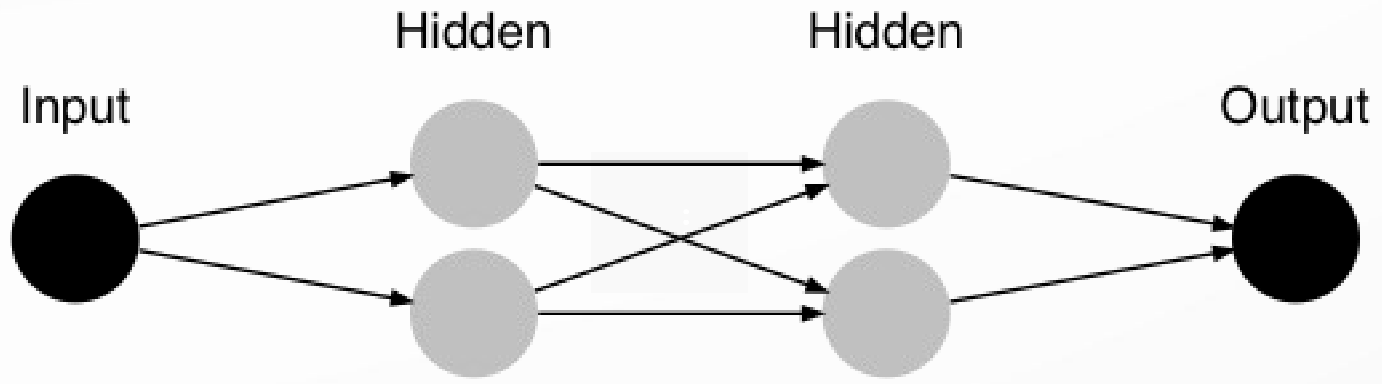


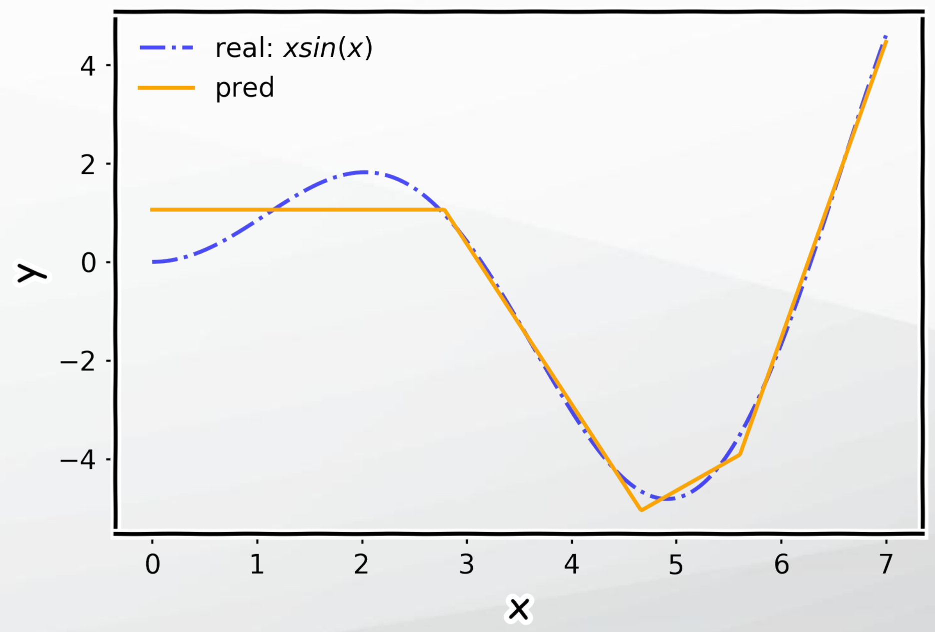
Architecture – performance example



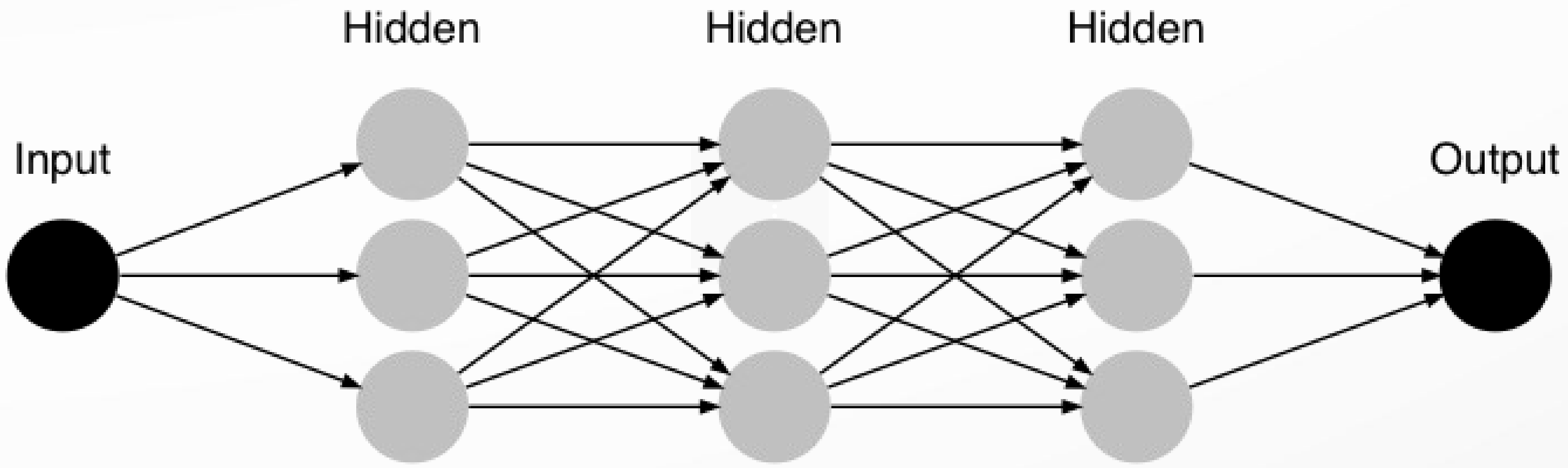


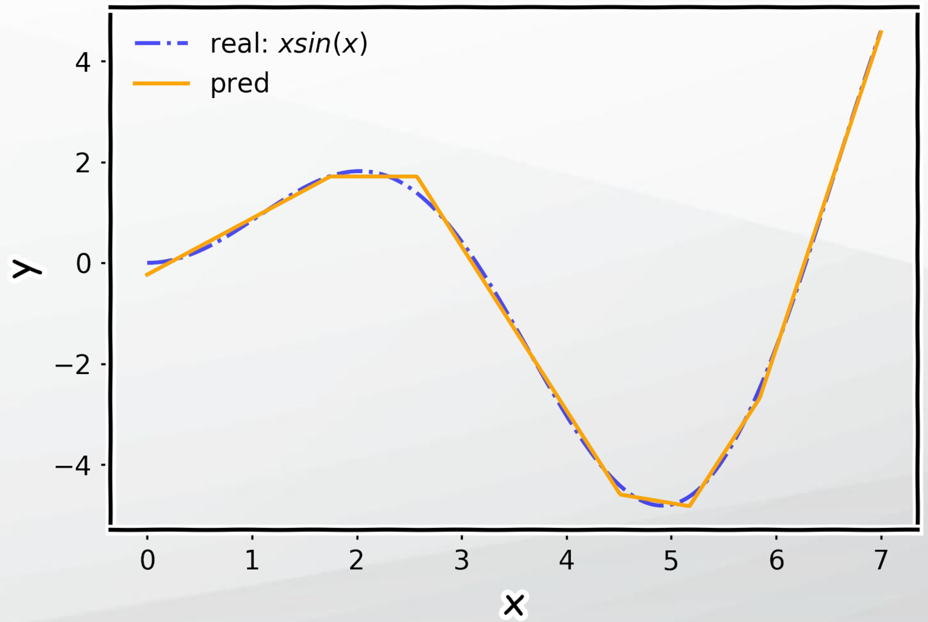
Architecture – performance example





Architecture – performance example





Universal Approximation Theorem

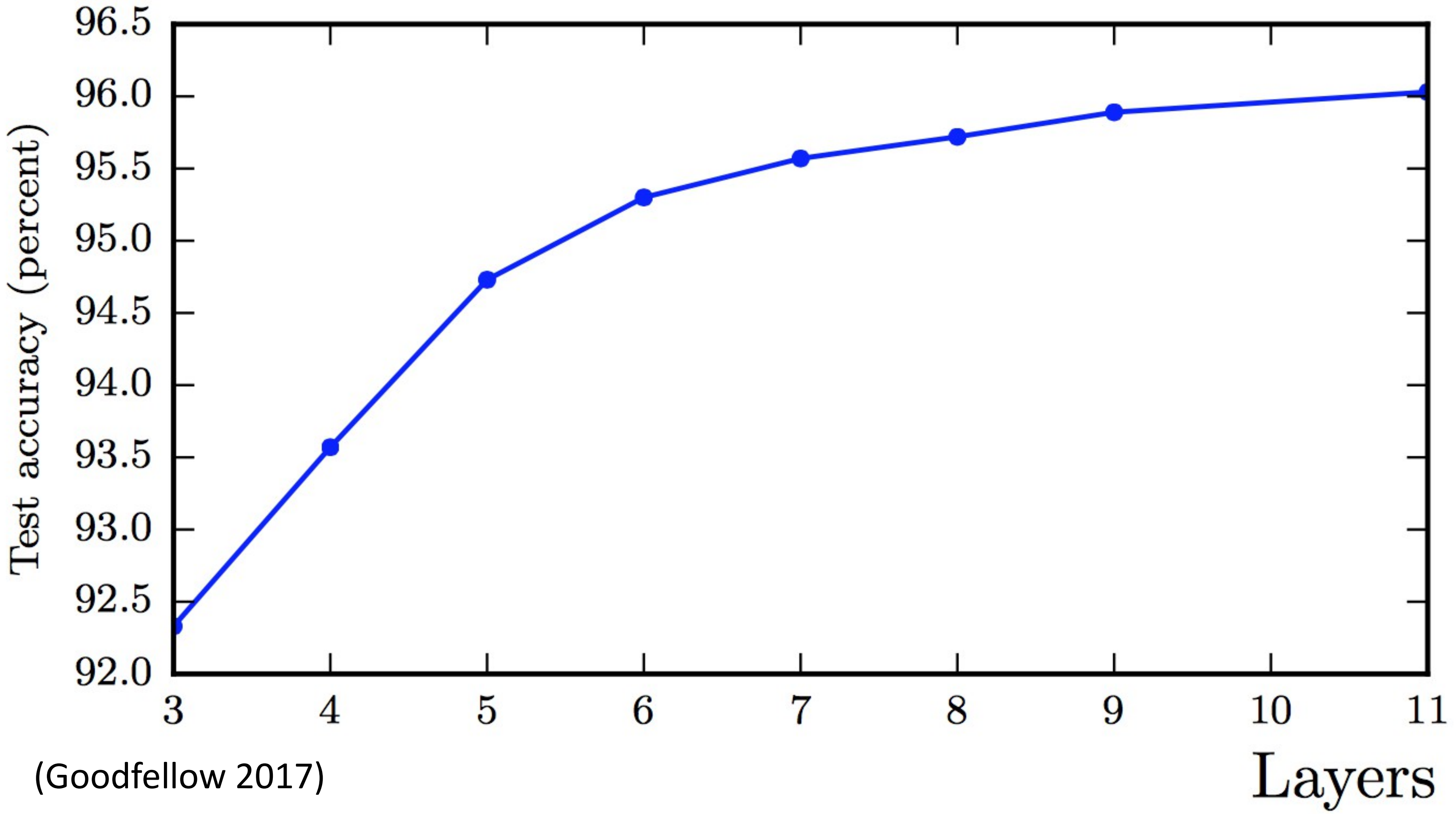
Think of a Neural Network as function approximation.

- NN:

  One hidden layer is enough to *represent* an approximation of any function to an arbitrary degree of accuracy

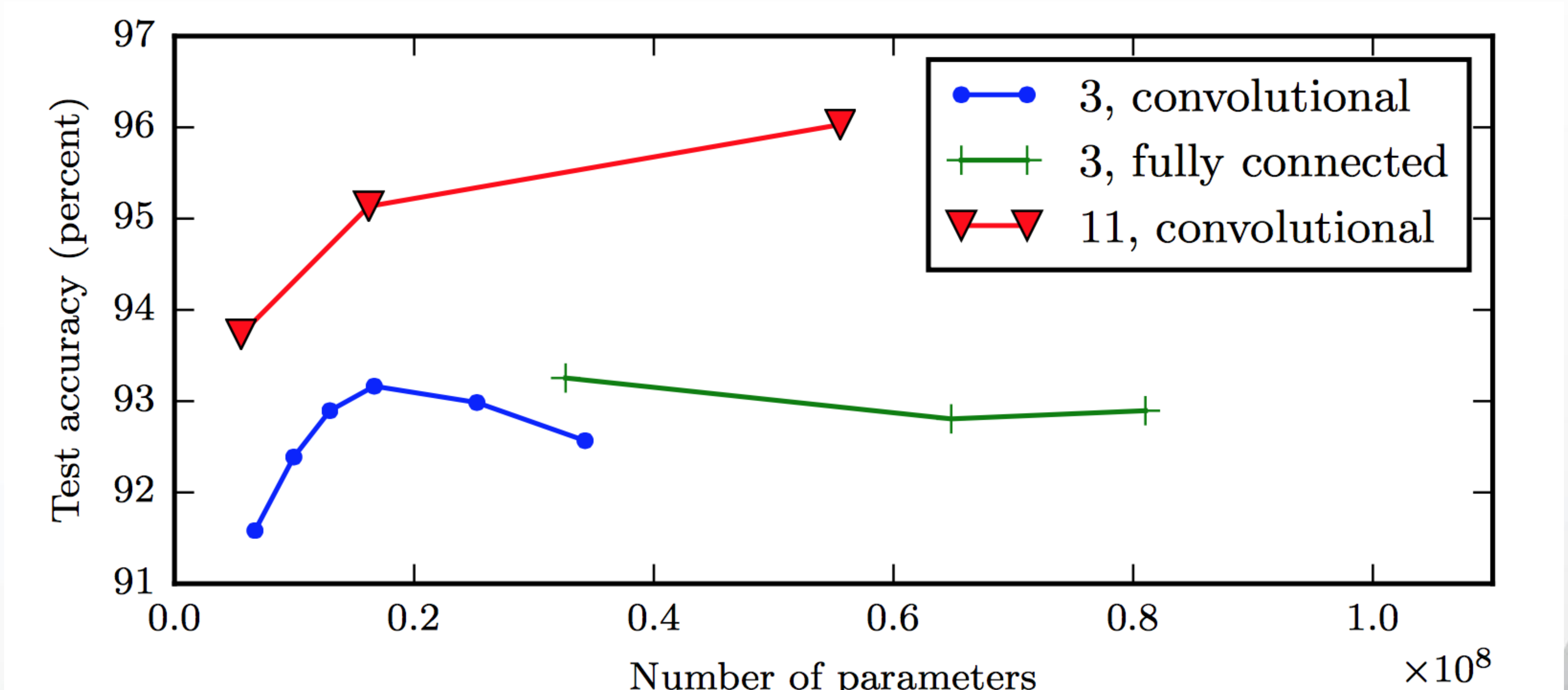
- So why deeper?
  - Shallow net may need (exponentially) more width
  - Shallow net may overfit more

Better generalization with depth



(Goodfellow 2017)

Overfitting in shallow nets



(Goodfellow 2017)

The **11-layer net** generalizes better on the test set when controlling for number of parameters.

The 3-layer nets perform worse on the test set, even with similar number of total parameters.

Training a feed-forward ANN

```
Initialisation of network weights $w^1_{1,1}, w^1_{1,2}, w^1_{1,3}, \ldots, w^n_{1,1}, \ldots, w^n_{H_{n-1}, H_n}, w^O_{1,1}, w^O_{1,2}, \ldots, w^O_{r, H_n}$
While not stop condition
    For each train example $(x^d, t^d)$, where $d = 1, 2, \ldots, n$
        Activate the network and determine the output $o^d$ :
            forward propagate the information and determine the output of each neuron
        Modify the weights:
            establish and backward propagate the error:
              • establish the errors of neurons from the output layer $e_r, r=1, \ldots, m$
              • backward propagate the errors in the entire network → distribute the errors
                on all connections of the network
              • modify the weights
EndWhile
```

Back-propagation of errors in an ANN

- One of the first algorithms for fine tuning of the weights in an ANN
- Very popular

Advantages:

- A gradient descent method
- does not require normalization of input vectors (normalization could improve performance)

Limitations:

- is not guaranteed to find the global minimum of the error function, but only a local minimum
- it has trouble crossing plateaus in the error function landscape.
- requires the derivatives of activation functions to be known at network design time.

Computing the derivatives of error

We consider an error function for the model $E(X, W, t)$

Compute the derivative of this function with respect to every weight $w^l_{j,k}$ (the weight from layer *l,* between neuron *j* from layer *l* and neuron *k* from layer *l-1*).

In general it is:

$$\frac{\partial E}{\partial w^l_{j,k}} = \frac{\partial E}{\partial o^l_j}\frac{\partial o^l_j}{\partial w^l_{j,k}} = \frac{\partial E}{\partial o^l_j}\frac{\partial o^l_j}{\partial net^l_j(X^l)}\frac{\partial net^l_j(X^l)}{\partial w^l_{j,k}}$$

Observe that in the last derivative we have a sum, but only one term depends on the weight so:

$$\frac{\partial net^l_j(X^l)}{\partial w^l_{j,k}} = \frac{\partial}{\partial w^l_{j,k}}\left(\sum_{q=1}^{n_{l-1}} x^l_q w^l_{j,q}\right) = \frac{\partial(x^l_k w^l_{j,k})}{\partial w^l_{j,k}} = x^l_k = o^{l-1}_k$$

Here: $X^l = (x^l_{1,} x^l_{2,} \ldots, x^l_{n_l-1})$ is the input for layer l (aka the output o^{l-1} from the previous layer)

Computing the derivatives of error

For the first hidden layer we have a different situation: the input for this layer is the actual input in the network.

If we evaluate further the partial derivative we get:

$$\frac{\partial o_j^l}{\partial net_j^l(X^l)} = \frac{\partial \phi(net_j^l(X^l))}{\partial net_j^l(X^l)}$$

which is the partial derivative of the activation function ϕ - hence the importance of the derivative

In practice we begin from the last layer because the partial derivatives for this layer are straight forward, and we move backward from layer to layer until we reach the first hidden one.

Thank You!