CS 148 –

Data Science Fundamentals

UCLA Computer Science

Anatomy of a NN

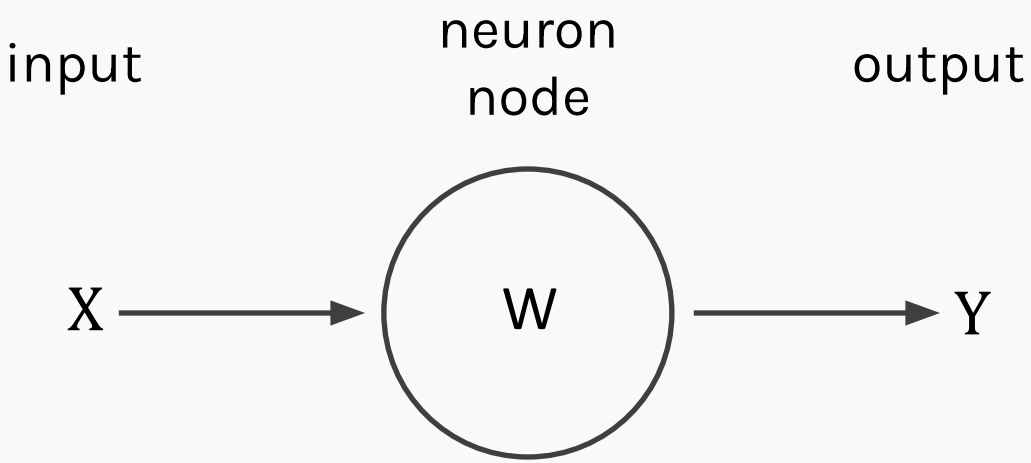
Design choices

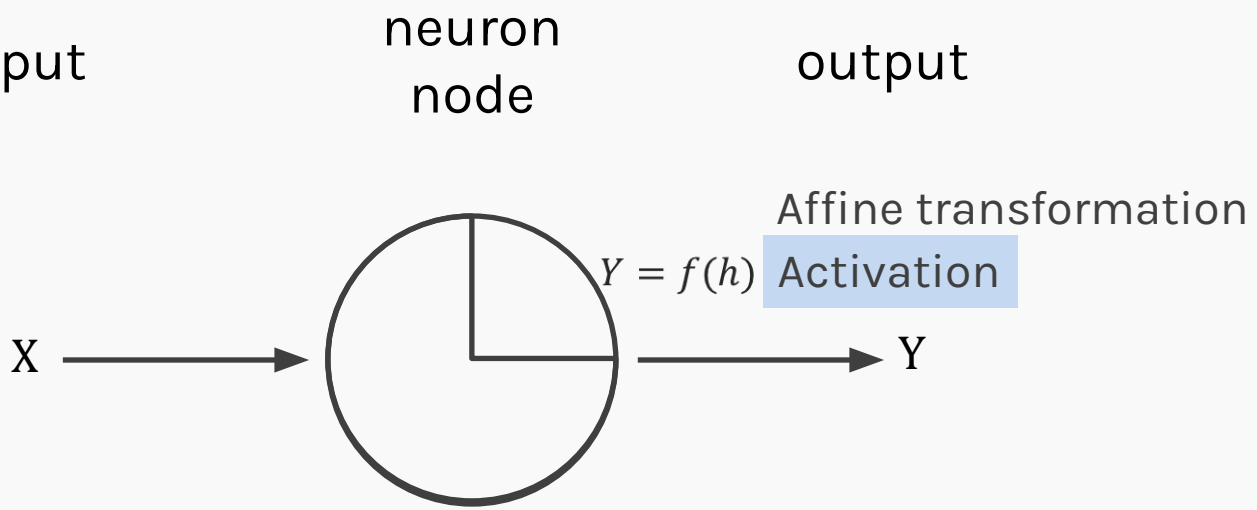
- Activation function
- Loss function
- Output units
- Architecture

Anatomy of a NN

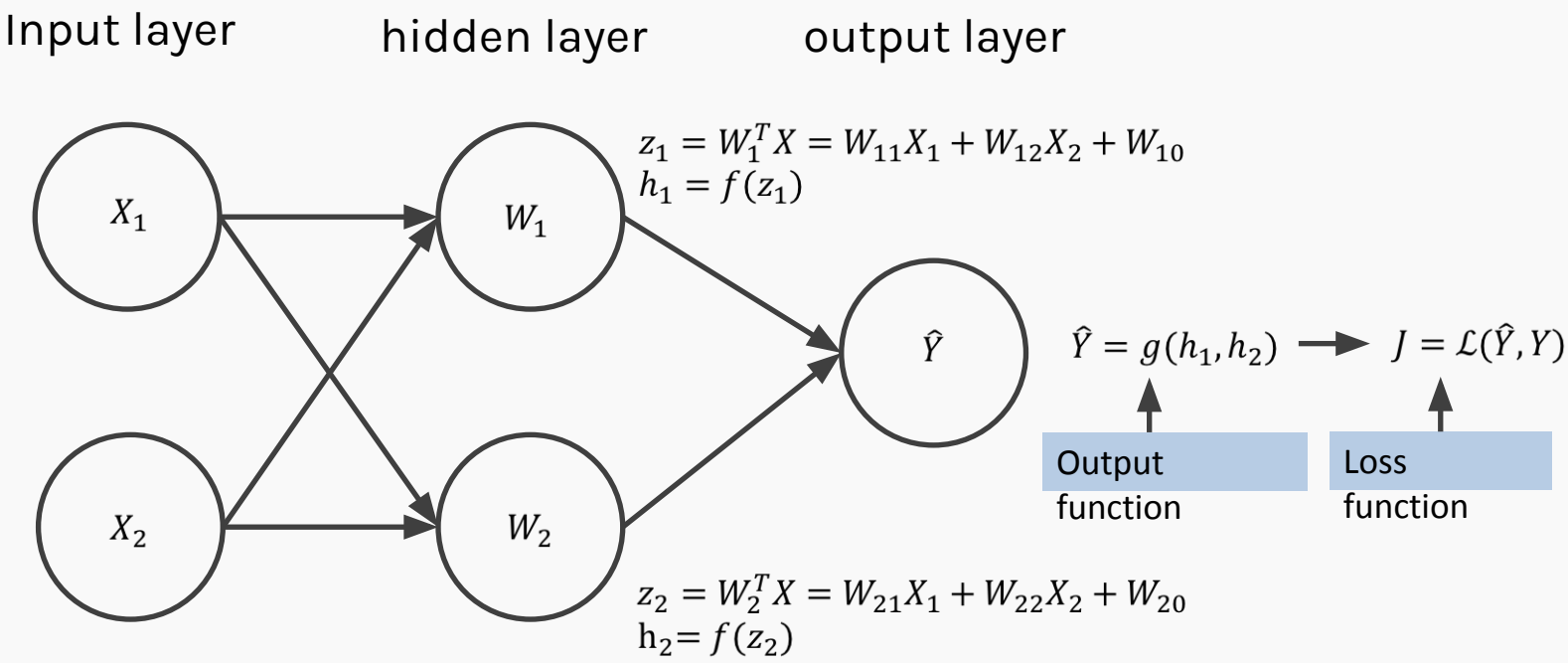
Design choices

- Activation function
- Loss function
- Output units
- Architecture

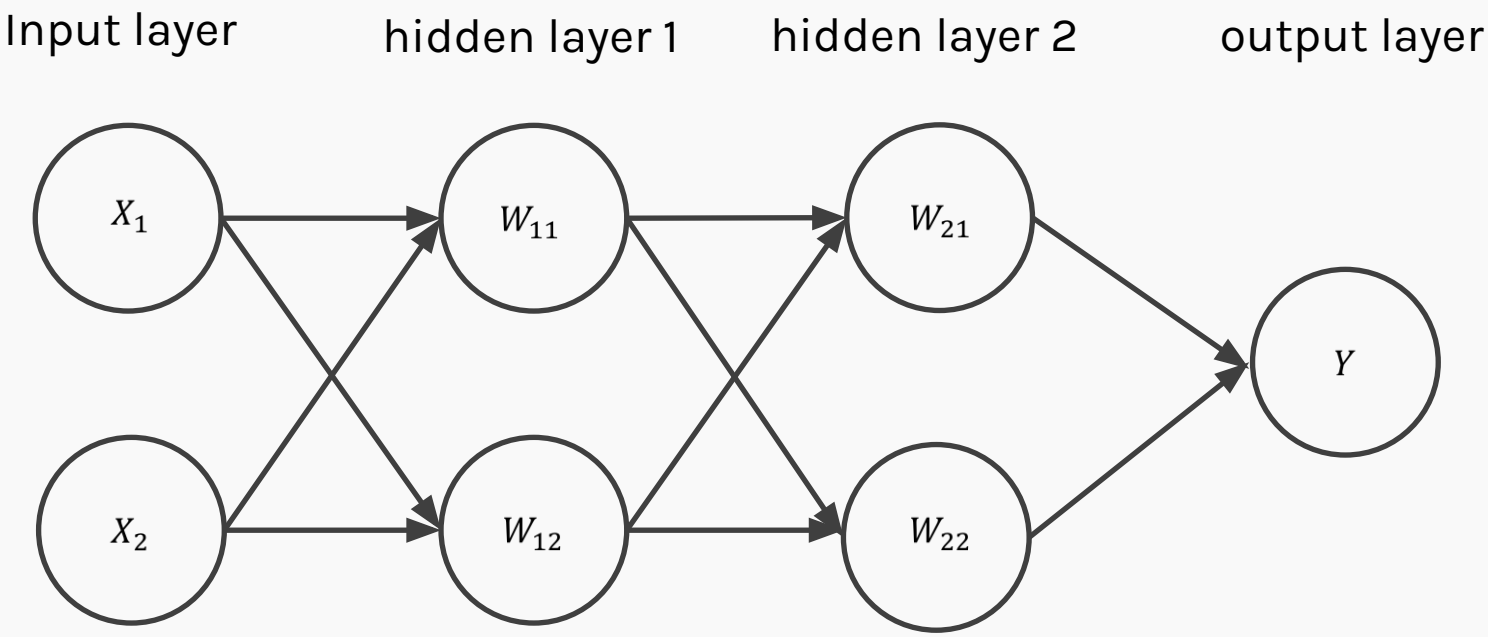

input
neuron
node
output
X
W
Y

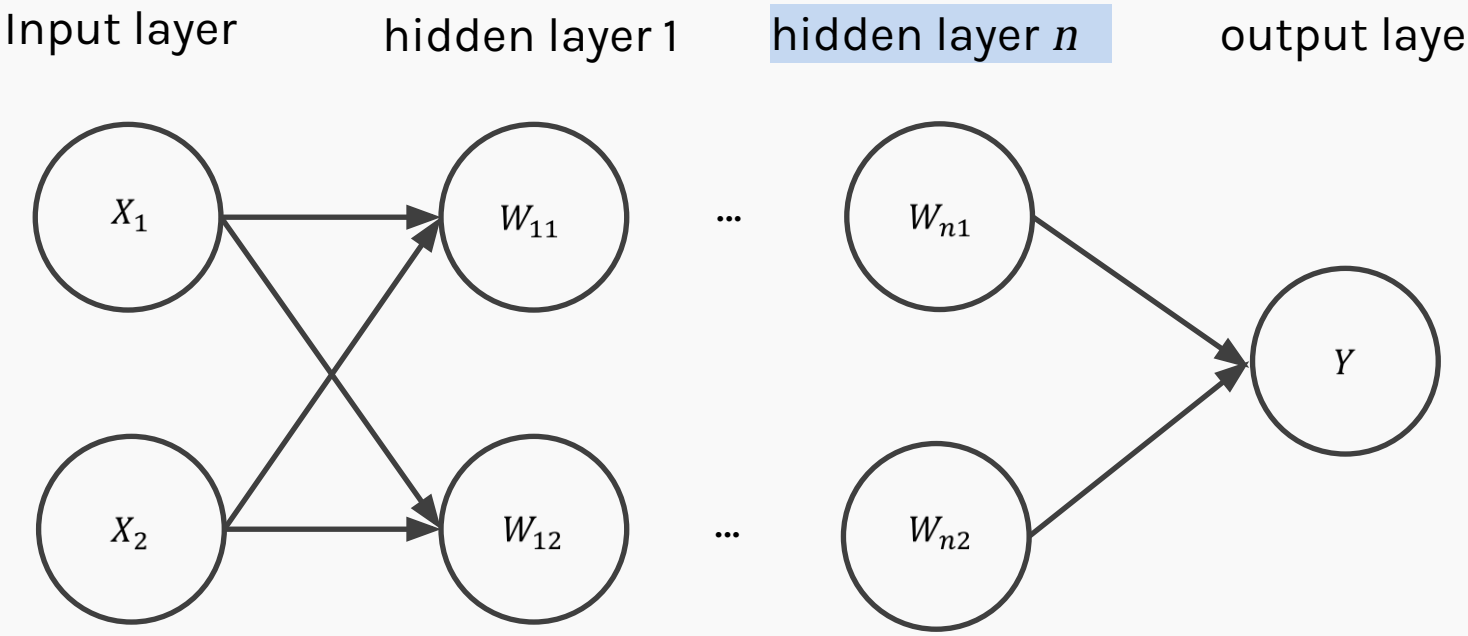


We will talk later about the choice of activation function. So far we have only talked about sigmoid as an activation function but there are other choices.



We will talk later about the choice of the output layer and the loss function. So far we consider sigmoid as the output and log-bernouli.


Input layer
hidden layer 1
hidden layer 2
output layer
X_1
X_2
W_{11}
W_{12}
W_{21}
W_{22}
Y



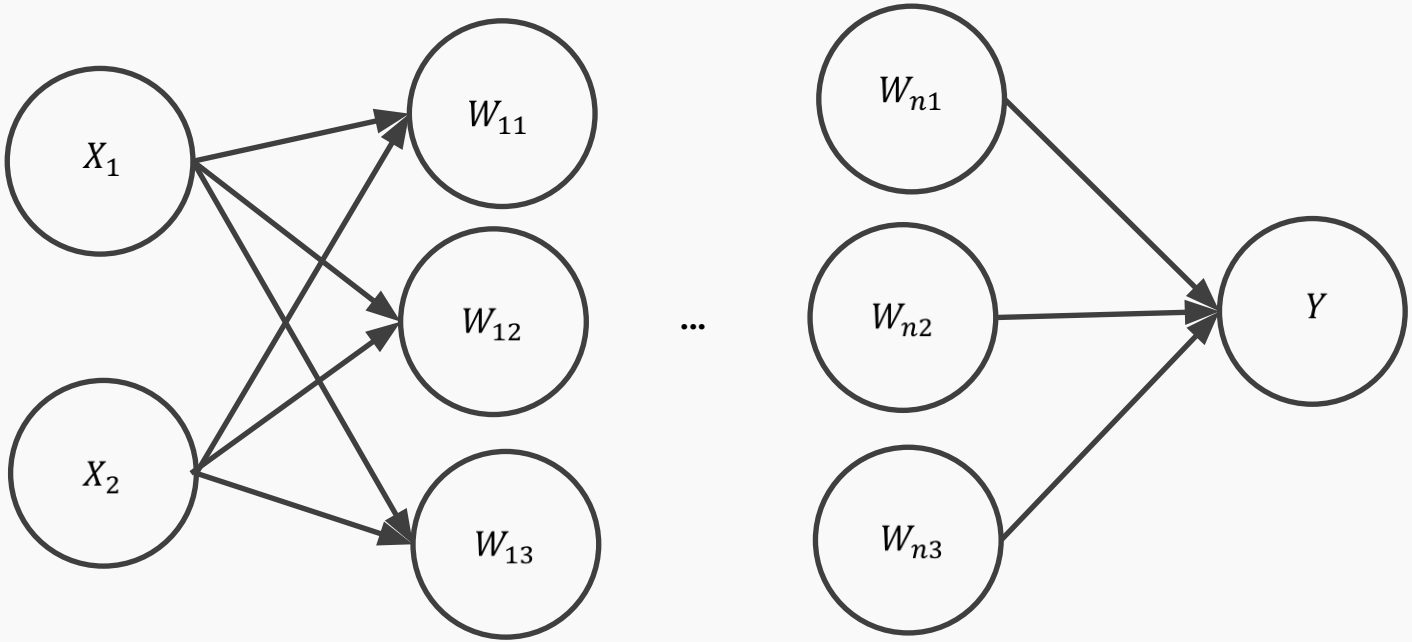
We will talk later about the choice of the number of layers.

Input layer

hidden layer 1,
3 nodes

hidden layer n
3 nodes

output layer

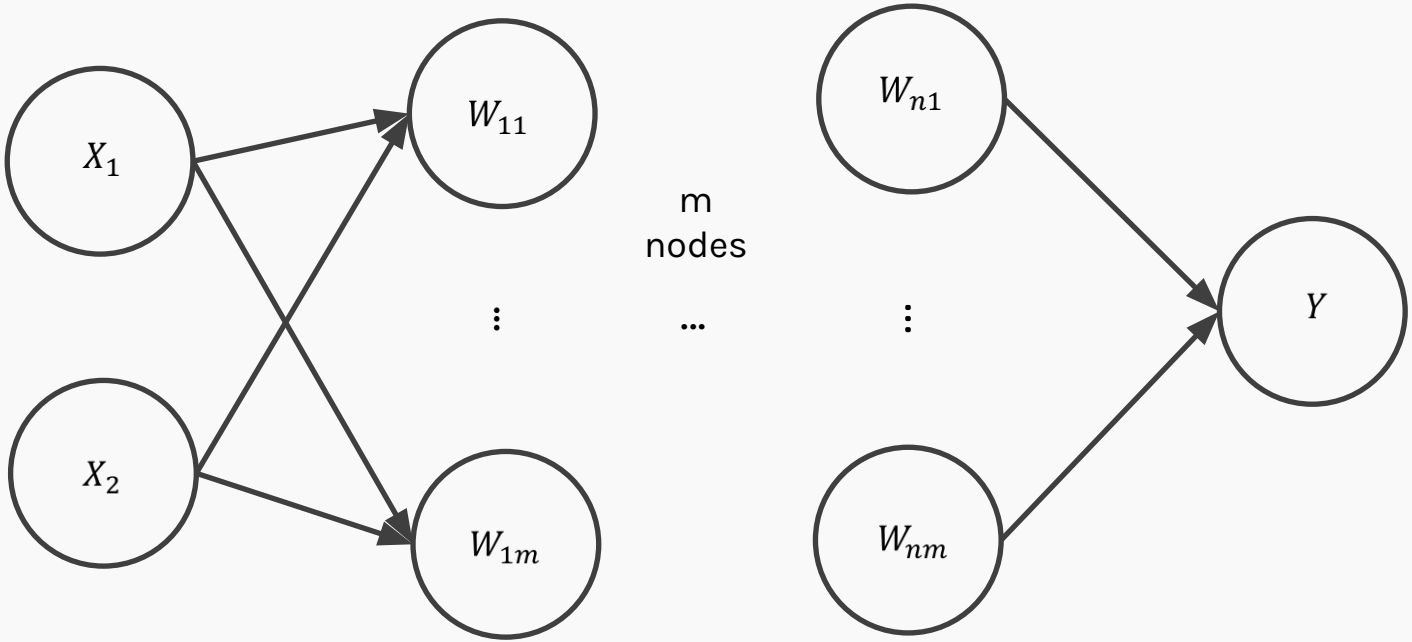


Input layer

hidden layer 1, m nodes

hidden layer n

output layer



We will talk later about the choice of the number of nodes.

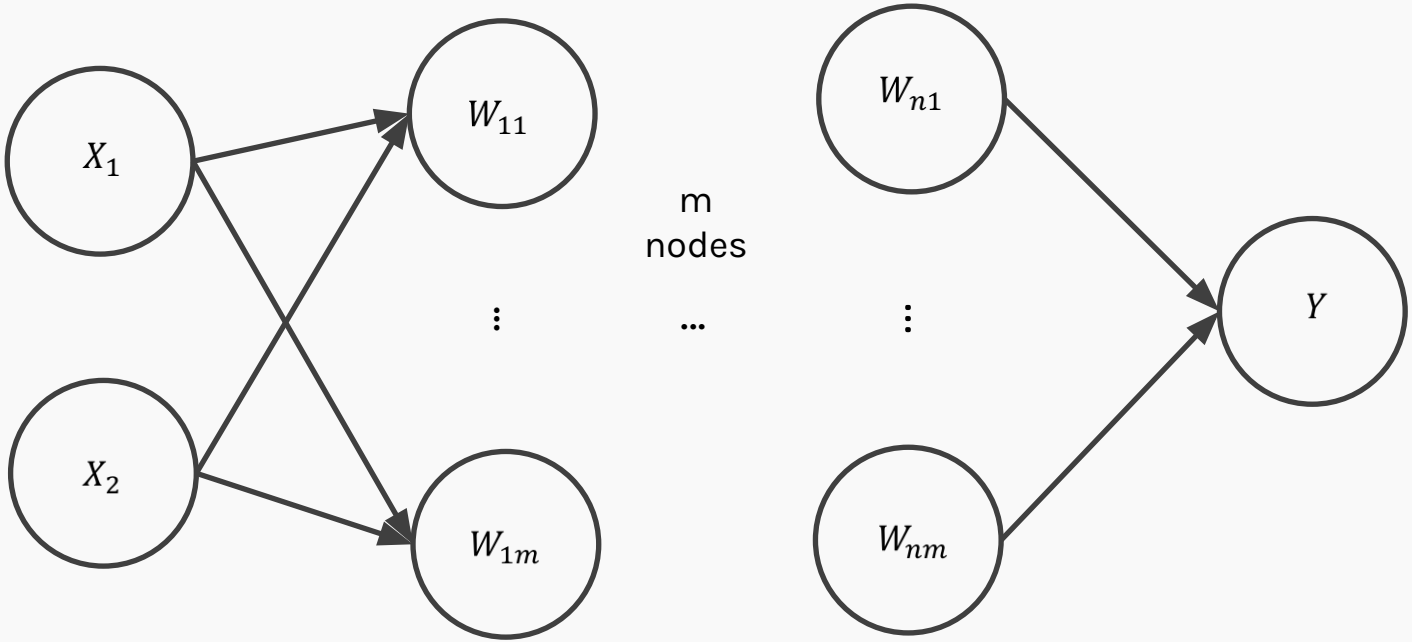
Input layer

hidden layer 1, m nodes

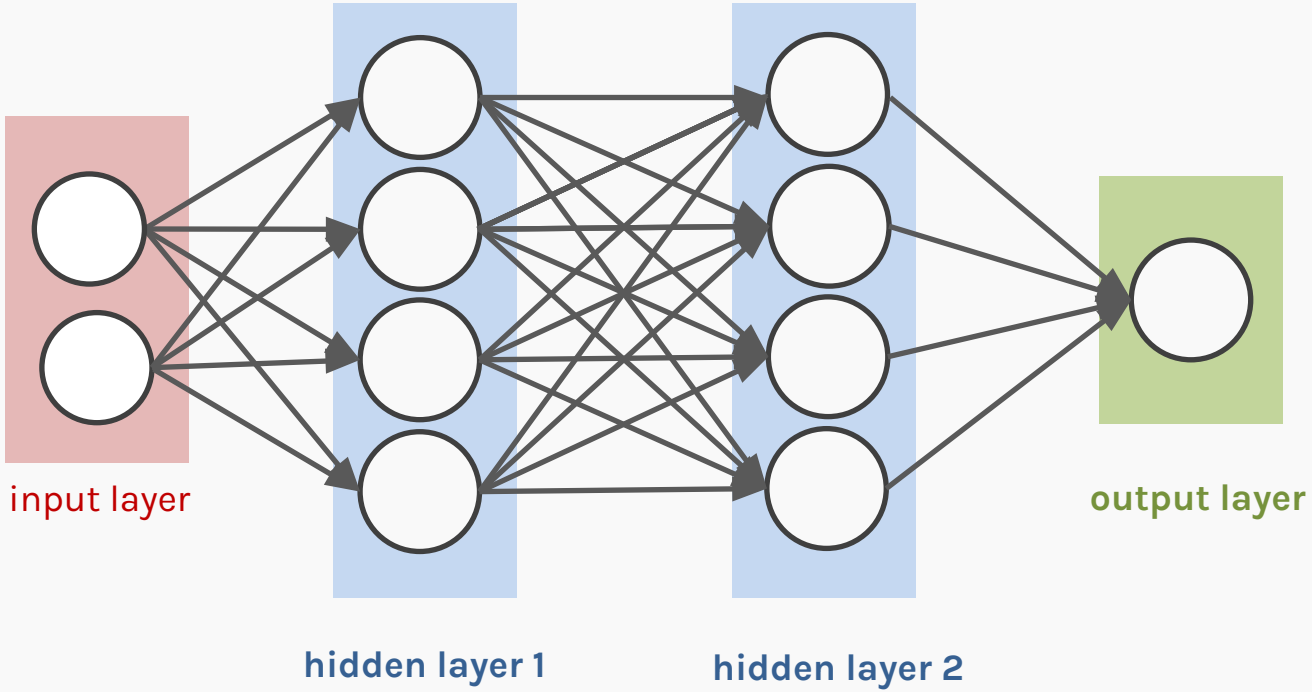
hidden layer n

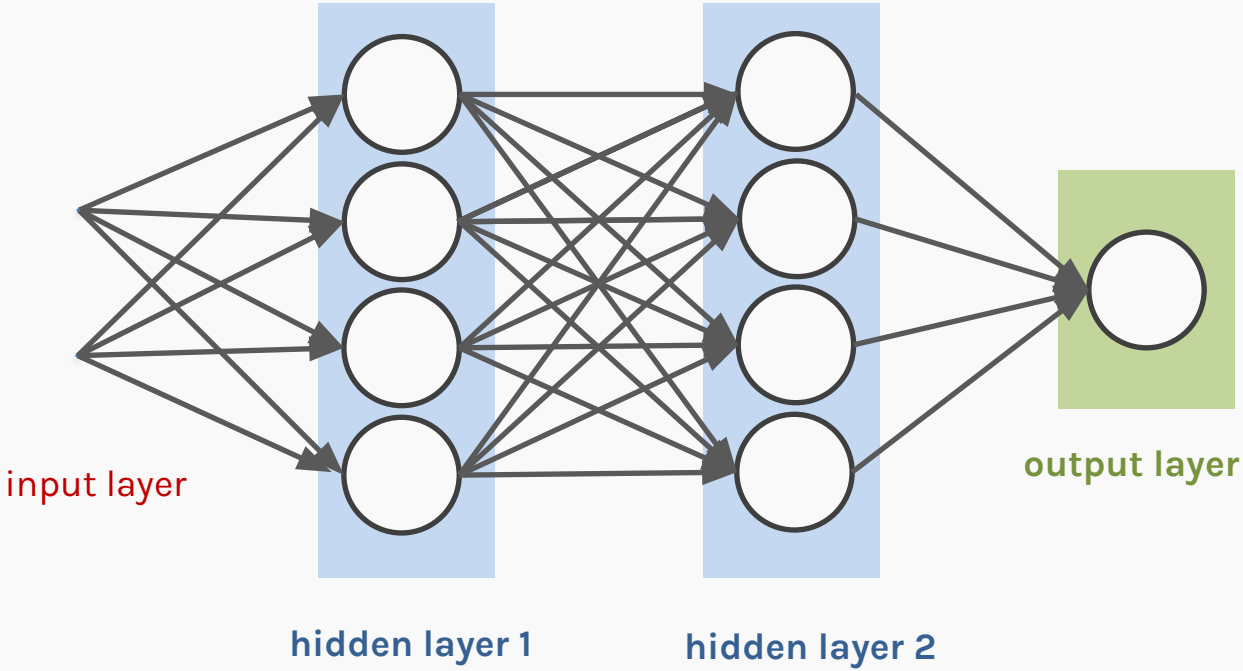
output layer

Number of inputs d

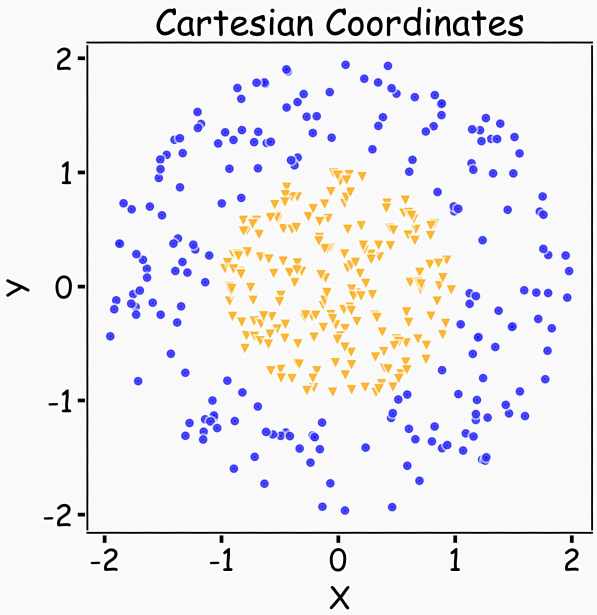


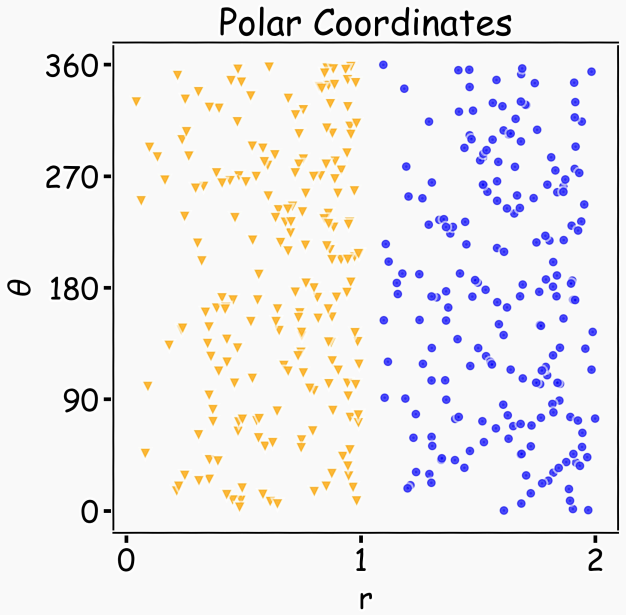
Number of inputs is specified by the data

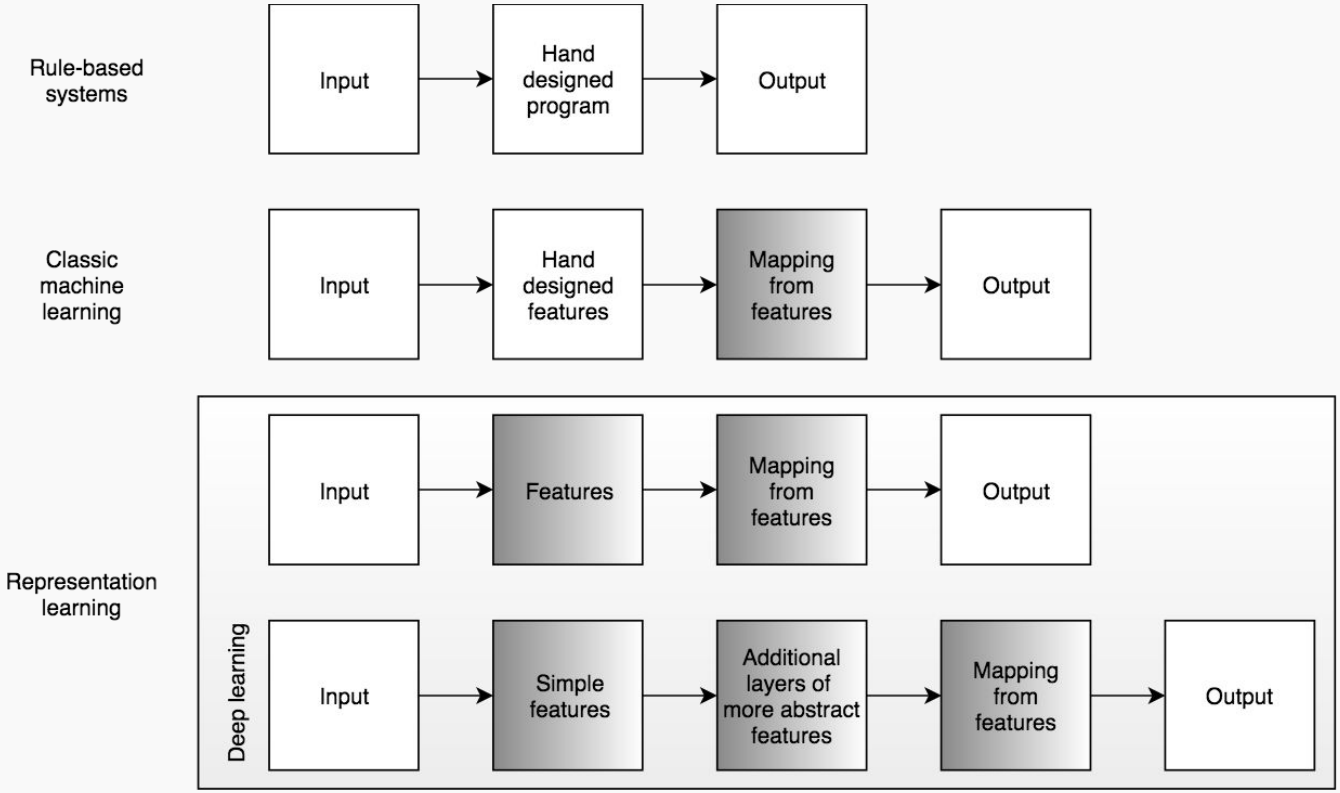

input layer
hidden layer 1
hidden layer 2
output layer


input layer
hidden layer 1
hidden layer 2
output layer

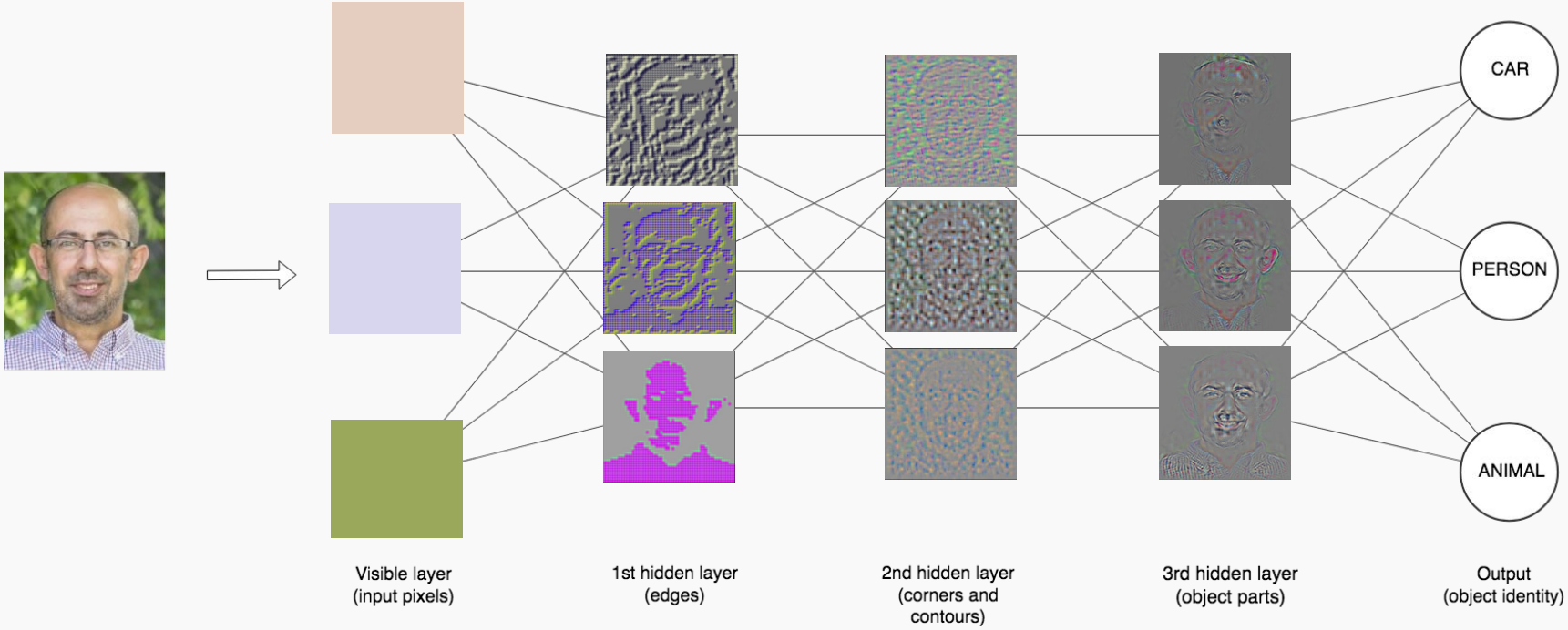
Representation matters!





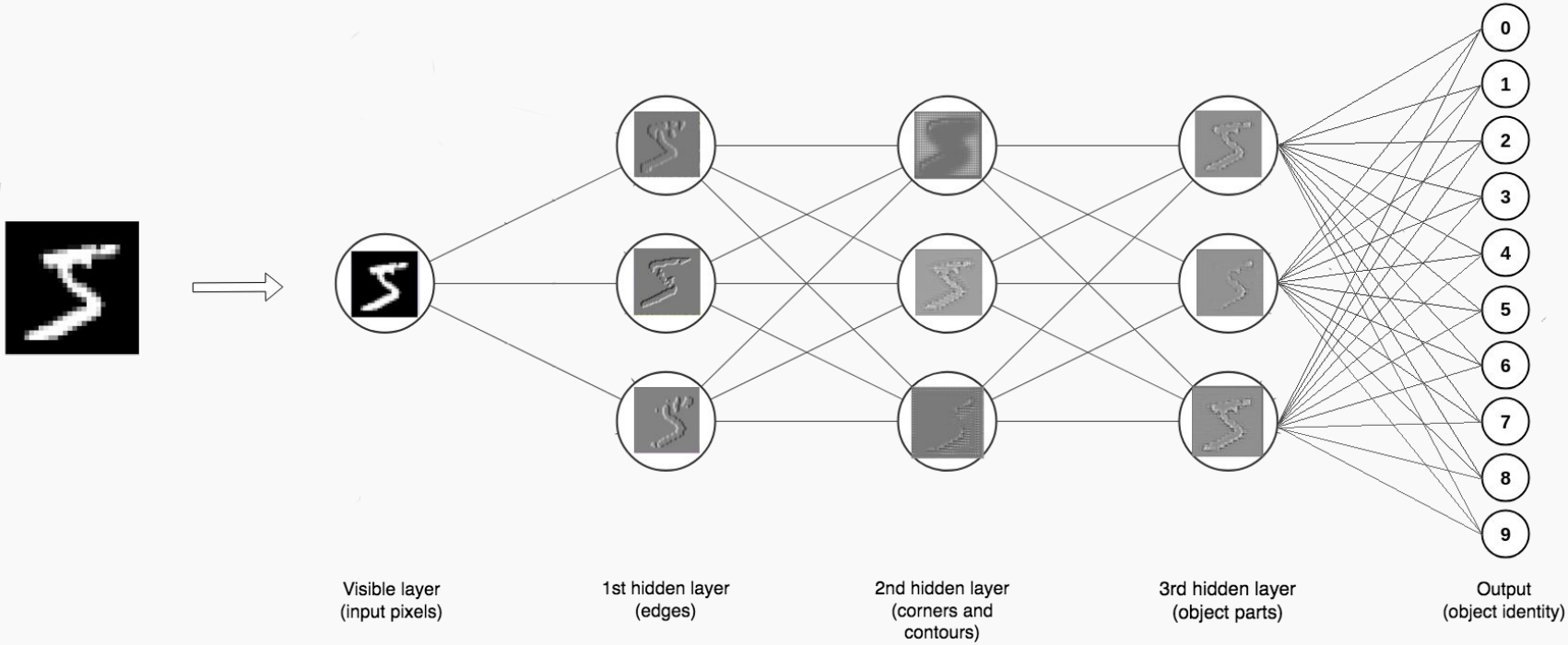

Rule-based systems
Input
Hand designed program
Output
Classic machine learning
Input
Hand designed features
Mapping from features
Output
Representation learning
Input
Features
Mapping from features
Output
Deep learning
Input
Simple features
Additional layers of more abstract features
Mapping from features
Output

Depth = Repeated Compositions

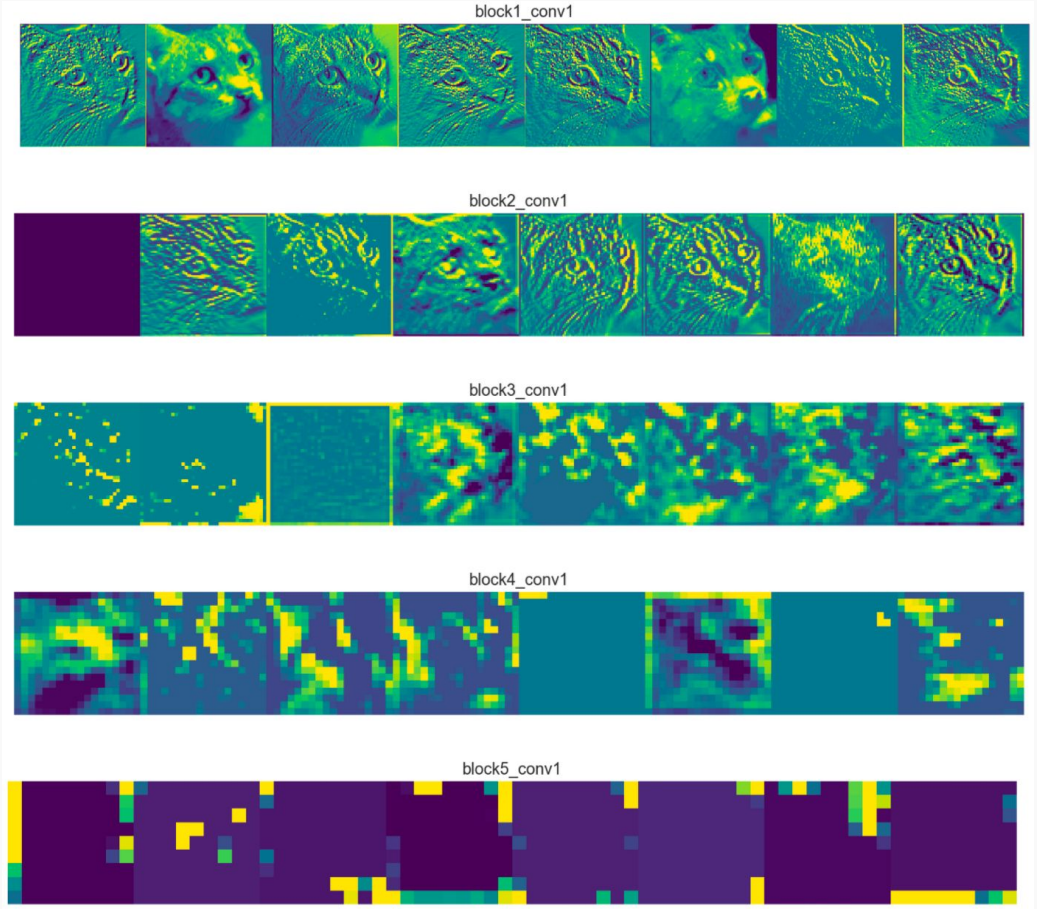


Handwritten digit recognition: MNIST data

Depth = Repeated Compositions



Depth = Repeated Compositions



Linear models:

- Can be fit efficiently (via convex optimization)
- Limited model capacity

Alternative:

$$f(x) = w^T \phi(x)$$

Where ϕ is a *non-linear transform*

Manually engineer Φ

- 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

- Directly learn ϕ

$$f(x;\theta) = W^T \phi(x;\theta)$$

- $\phi(x;\theta)$ is an automatically-learned **representation** of x
- For **deep networks**, ϕ is the function learned by the **hidden layers** of the network
- θ are the learned weights

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

Anatomy of a NN

Design choices

- Activation function
- Loss function
- Output units
- Architecture

Anatomy of a NN

Design choices

- **Activation function**
- Loss function
- Output units
- Architecture

$$h = f(W^T X + b)$$

The activation function should:

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

Common choices are

- sigmoid
- tanh
- ReLU, leaky ReLU, Generalized ReLU, MaxOut
- softplus
- swish

$$h = f(W^T X + b)$$

The activation function should:

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

Common choices are

- sigmoid
- tanh
- ReLU, leaky ReLU, Generalized ReLU, MaxOut
- softplus
- swish

$$h = f(W^T X + b)$$

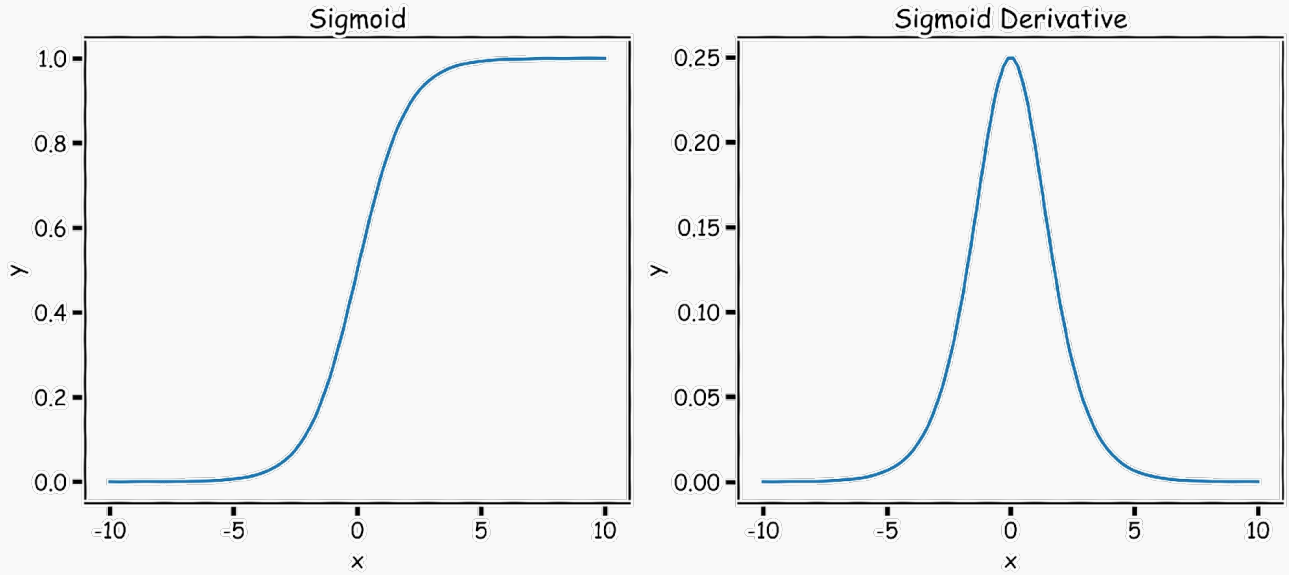
The activation function should:

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

Common choices are

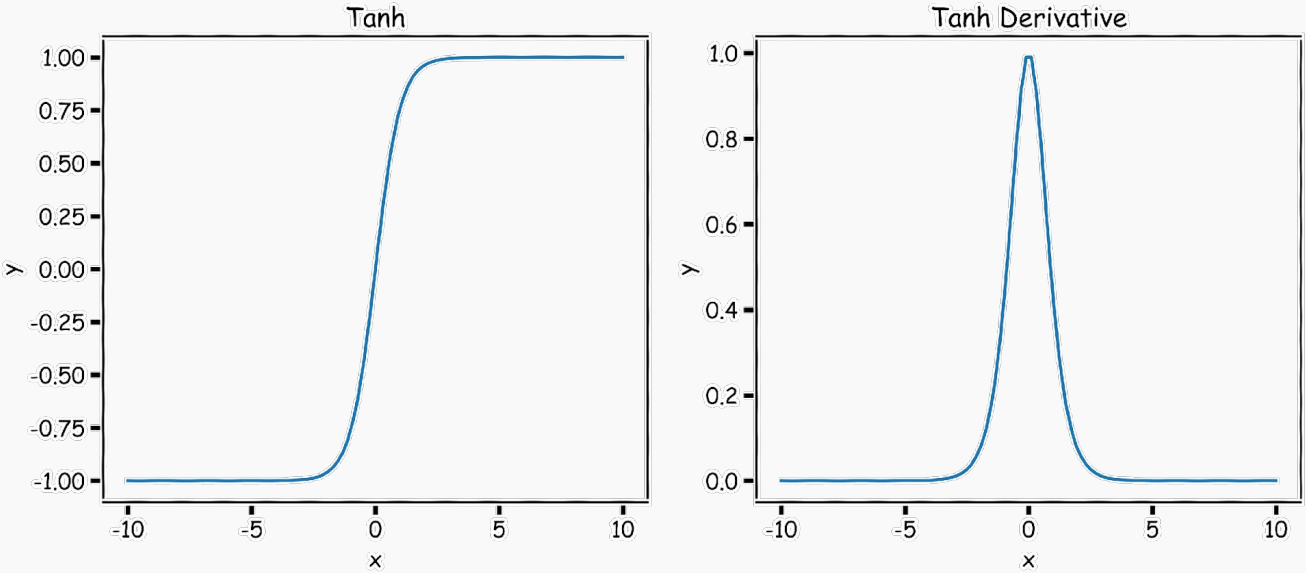
- sigmoid
- tanh
- ReLU, leaky ReLU, Generalized ReLU, MaxOut
- softplus
- swish

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



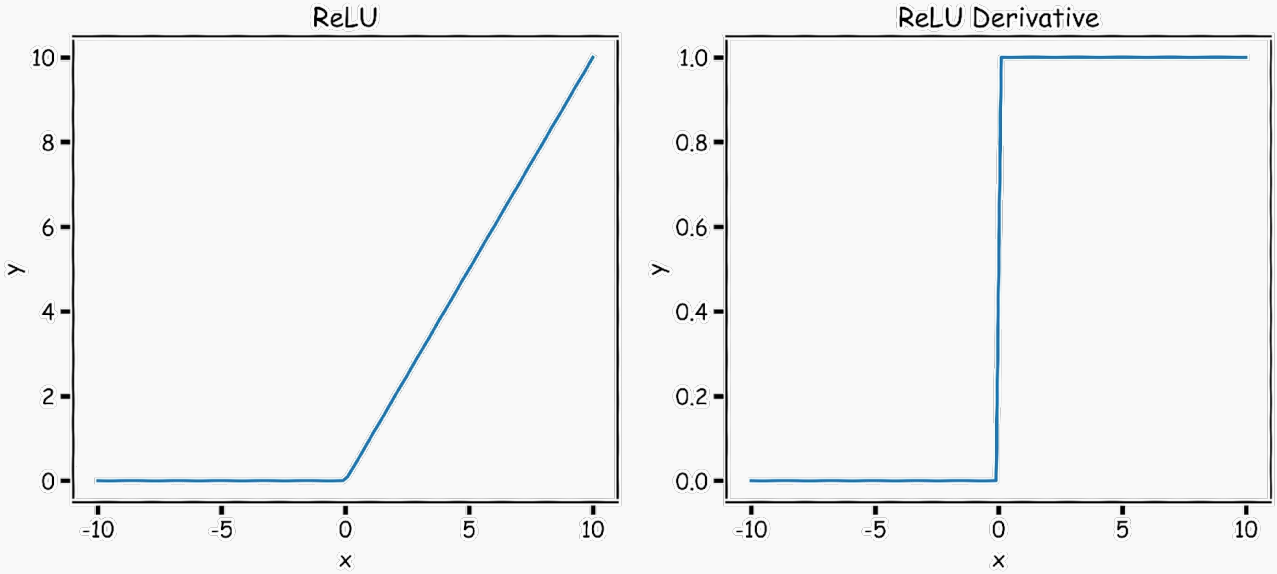
Derivative is **zero** for much of the domain. This leads to “vanishing gradients” in backpropagation.

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



Same problem of “vanishing gradients” as sigmoid.

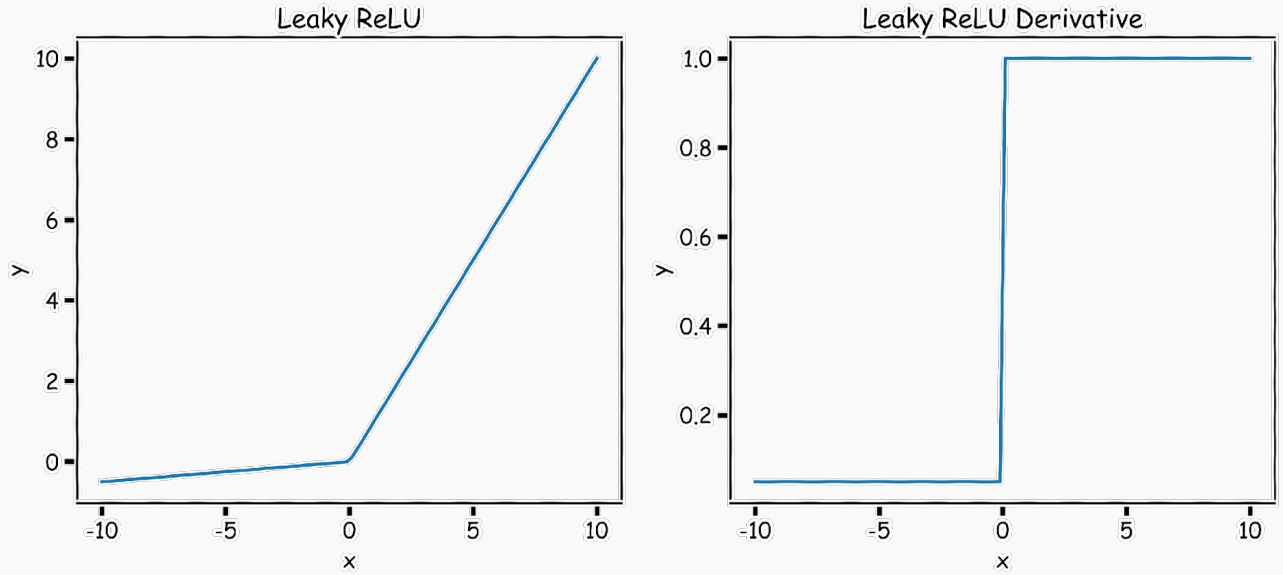
$$y = \max(0, x)$$



Two major advantages:

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

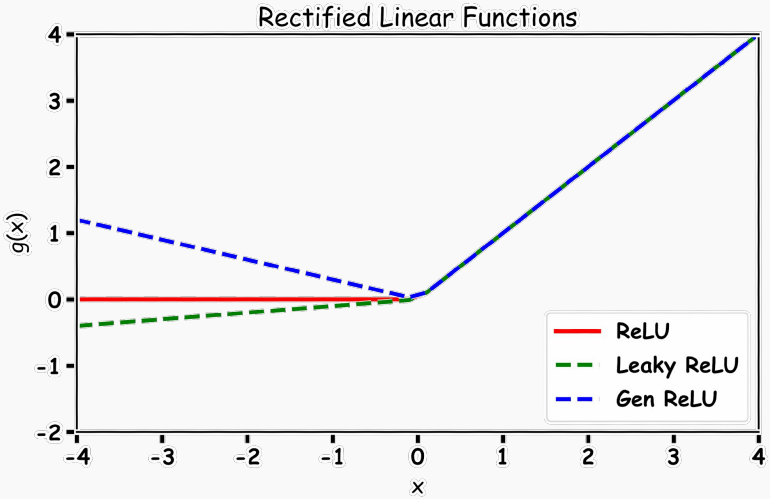
$$y = \max(0, x) + \alpha \min(0,1)$$
where α takes a small value



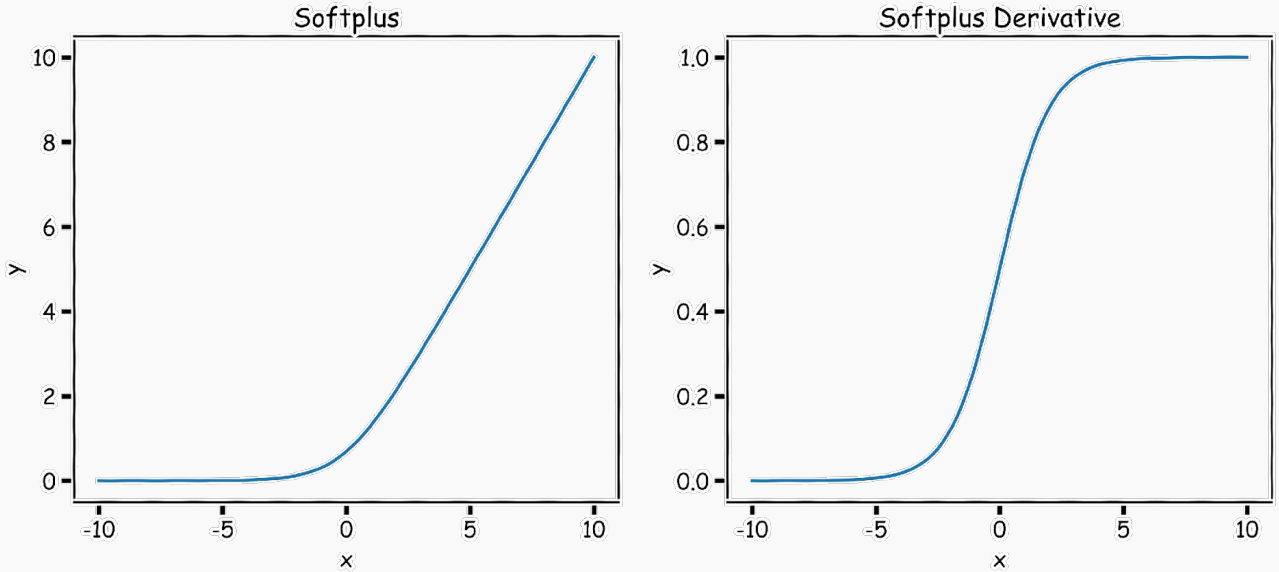
- Tries to fix “dying ReLU” problem: derivative is non-zero everywhere.
- Some people report success with this form of activation function, but the results are not always consistent

Generalization: For $\alpha_i > 0$

$$g(x_i, \alpha) = \max\{a, x_i\} + \alpha \min\{0, x_i\}$$



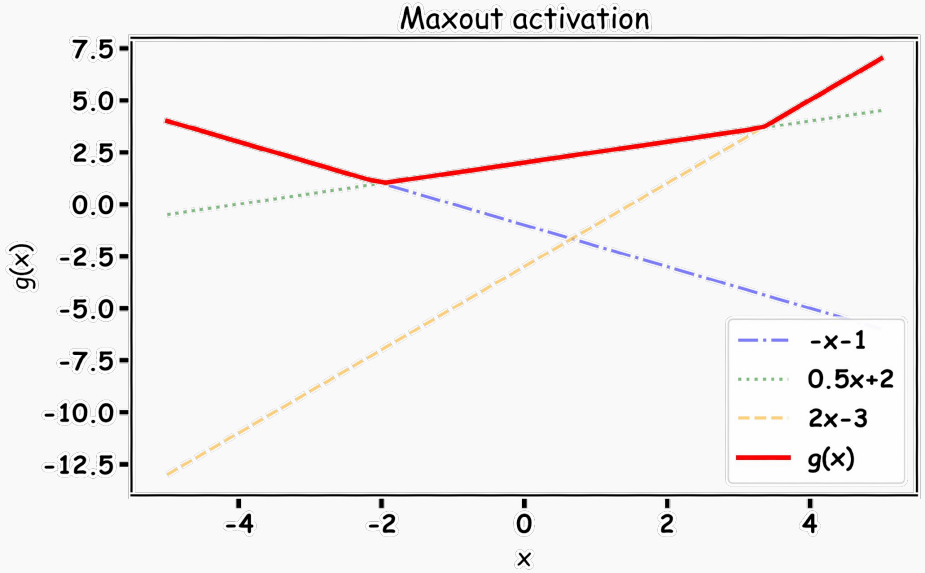
$$y = \log(1 + e^x)$$



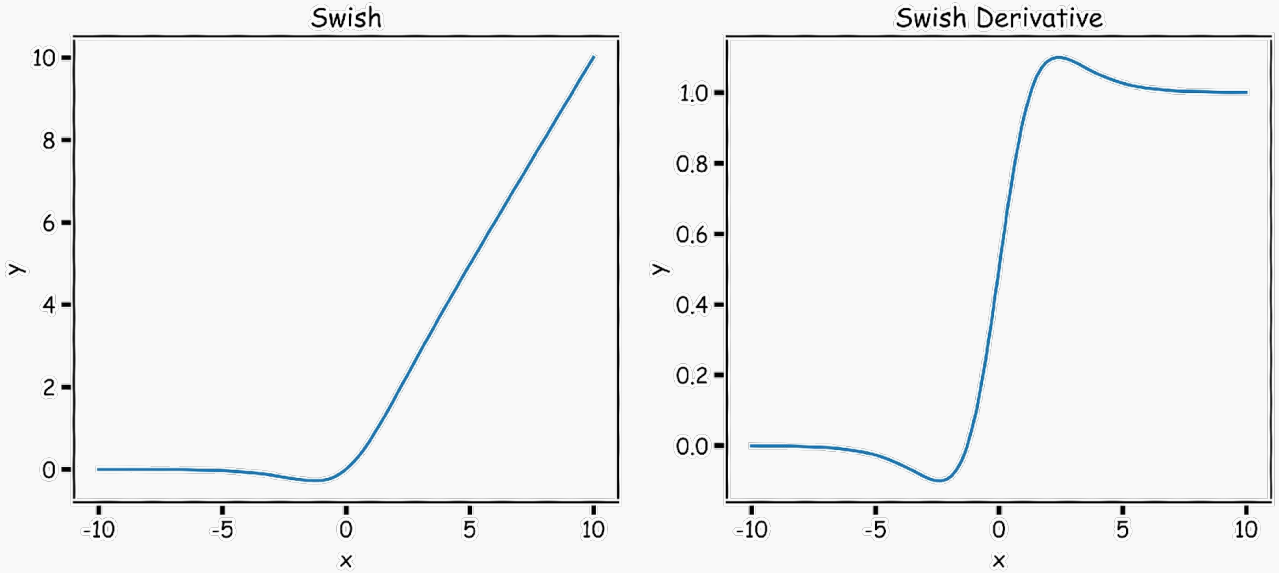
The logistic sigmoid function is a smooth approximation of the derivative of the rectifier

Max of k linear functions. Directly learn the activation function.

$$g(x) = \max_{i \in \{1,\ldots,k\}} \alpha_i x_i + \beta$$



$$g(x) = x\sigma(\beta x)$$



Currently, the most successful and widely-used activation function is the ReLU. Swish tends to work better than ReLU on deeper models across a number of challenging datasets.

Anatomy of a NN

Design choices

- Activation function
- **Loss function**
- Output units
- Architecture

Likelihood for a given point:

$$p(y_i|W;x_i)$$

Assume independency, likelihood for all measurements:

$$L(W;X,Y) = p(Y|W;X) = \prod_i p(y_i|W;x_i)$$

Maximize the likelihood, or equivalently maximize the log-likelihood:

$$\log L(W;X,Y) = \sum_i \log p(y_i|W;x_i)$$

Turn this into a loss function:

$$\mathcal{L}(W;X,Y) = -\log L(W;X,Y)$$

Do not need to design separate loss functions if we follow this simple procedure

Examples:

- Distribution is **Normal** then likelihood is:

$$p(y_i \mid W; x_i) = \frac{1}{\sqrt{\{2\pi^2\sigma\}}} e^{-\frac{\left(y_i - \hat{y}_i\right)^2}{2\sigma\hat{}2}}$$

$$\mathscr{L}(W; X, Y) = \sum_i \left(y_i - \hat{y}_i\right)^2$$

MSE

- Distribution is **Bernouli** then likelihood is:

$$p(y_i \mid W; x_i) = p_i^{y_i}(1 - p_i)^{1-y_i}$$

$$\mathscr{L}(W; X, Y) = -\sum_i \left[y_i \log p_i + (1 - y_i)\log(1 - p_i)\right]$$

Cross-Entropy

Activation function

Loss function

Output units

Architecture

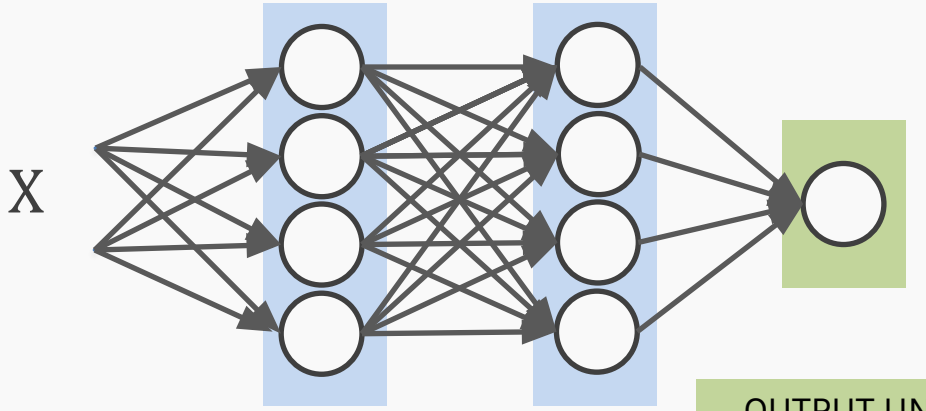
Optimizer

Output Type	Output Distribution	Output layer	Loss Function
Binary			

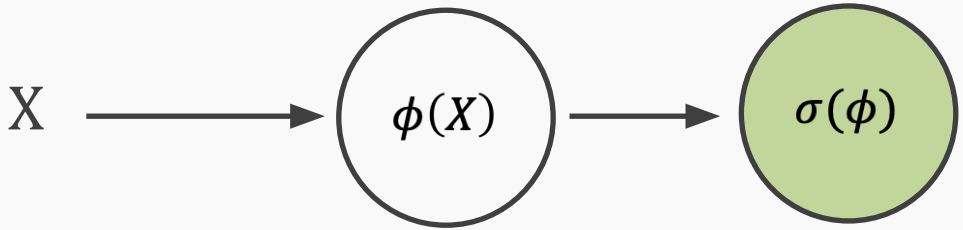
Output Type	Output Distribution	Output layer	Loss Function
Binary	Bernoulli		

Output Type	Output Distribution	Output layer	Loss Function
Binary	Bernoulli		Binary Cross Entropy

Output Type	Output Distribution	Output layer	Loss Function
Binary	Bernoulli	?	Binary Cross Entropy



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



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

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

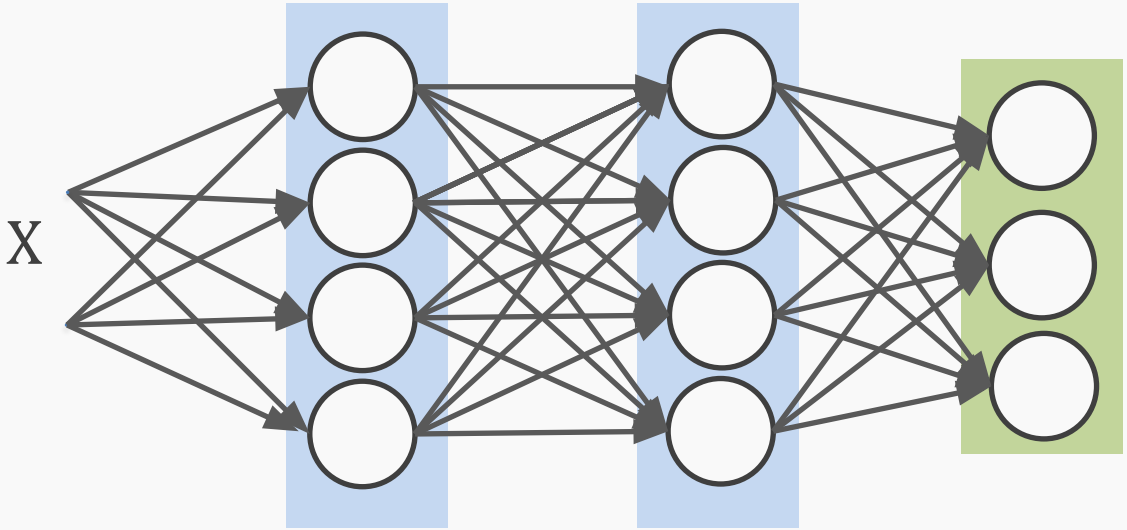
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete			

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli		

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli		Cross Entropy

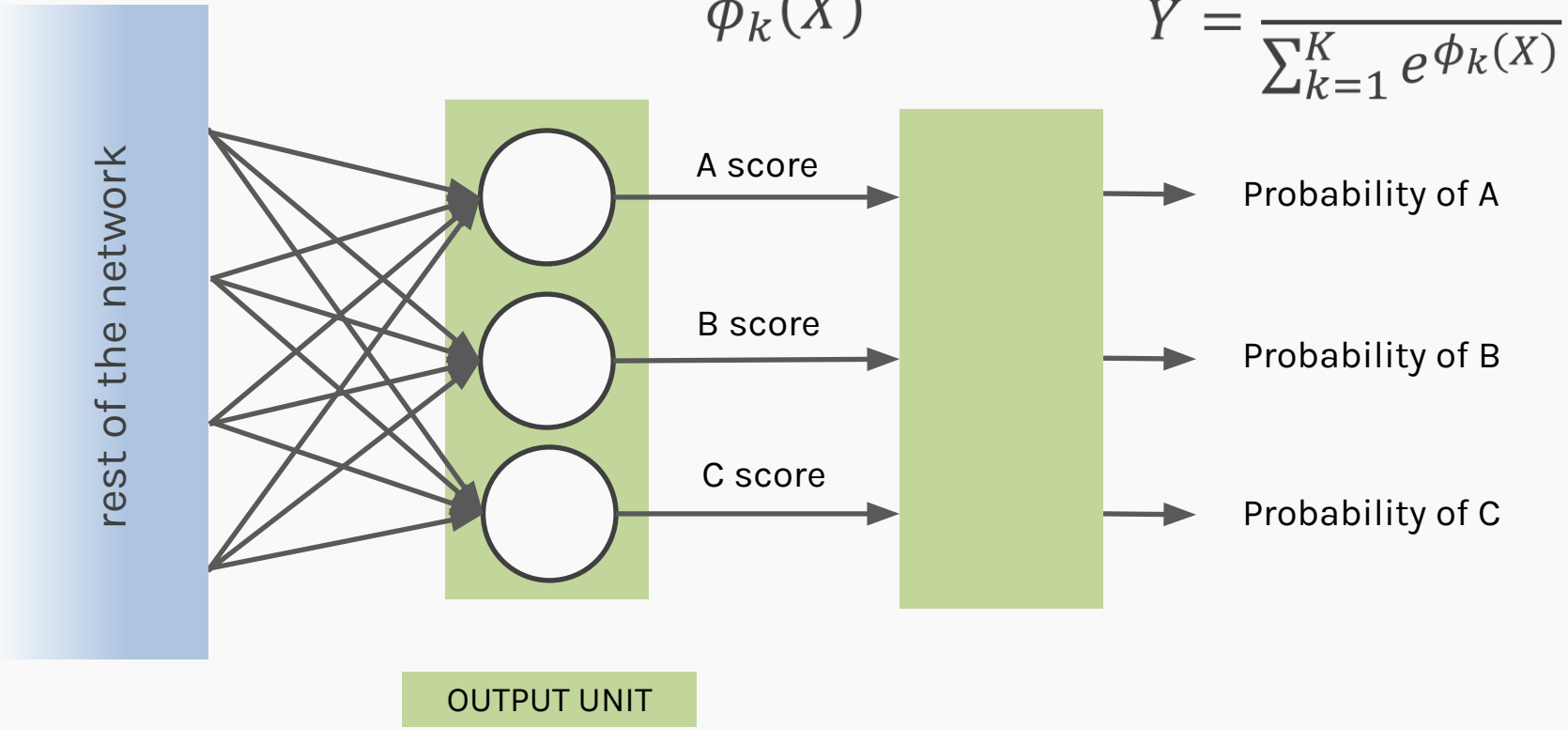
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	?	Cross Entropy



$$\hat{Y} = [P_1, P_2, P_3]$$

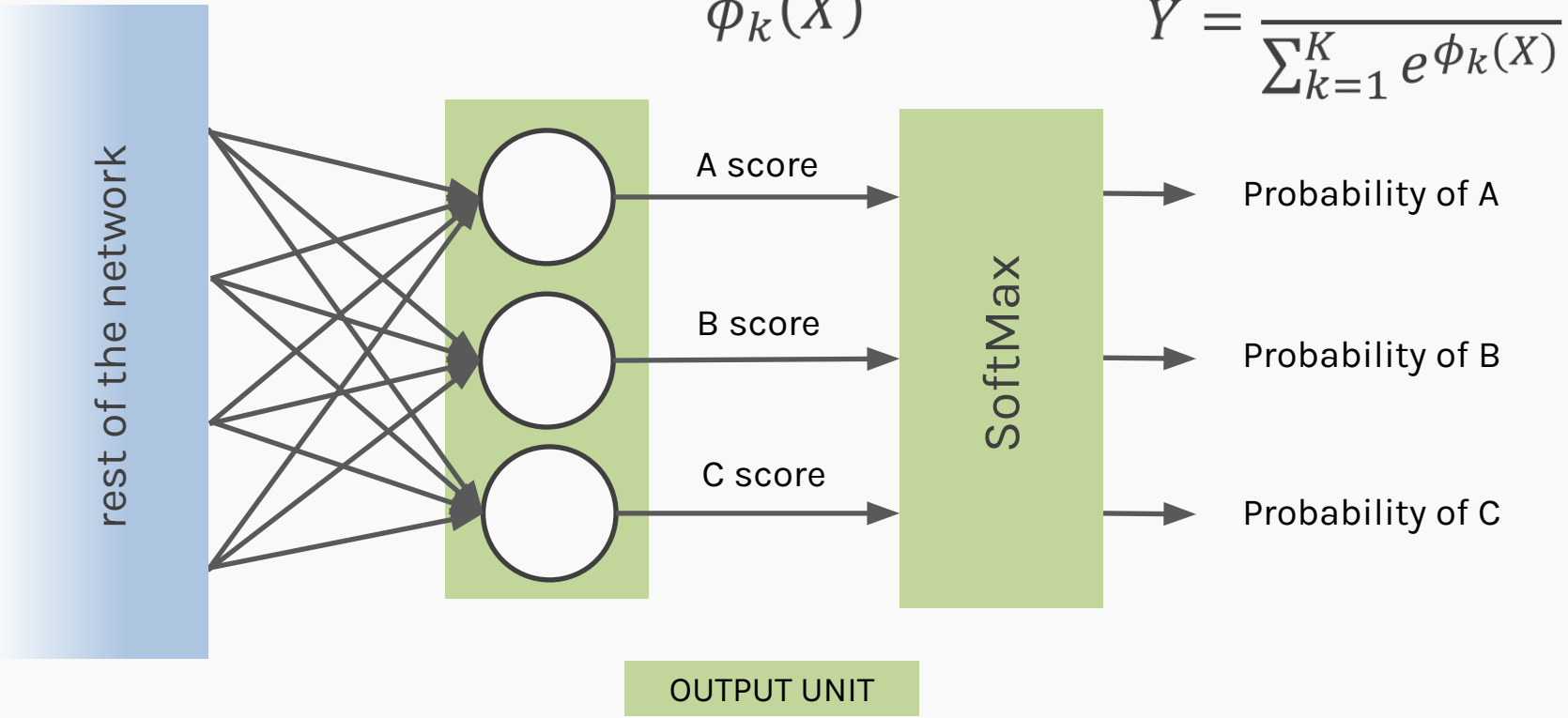
$$\phi_k(X)$$

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



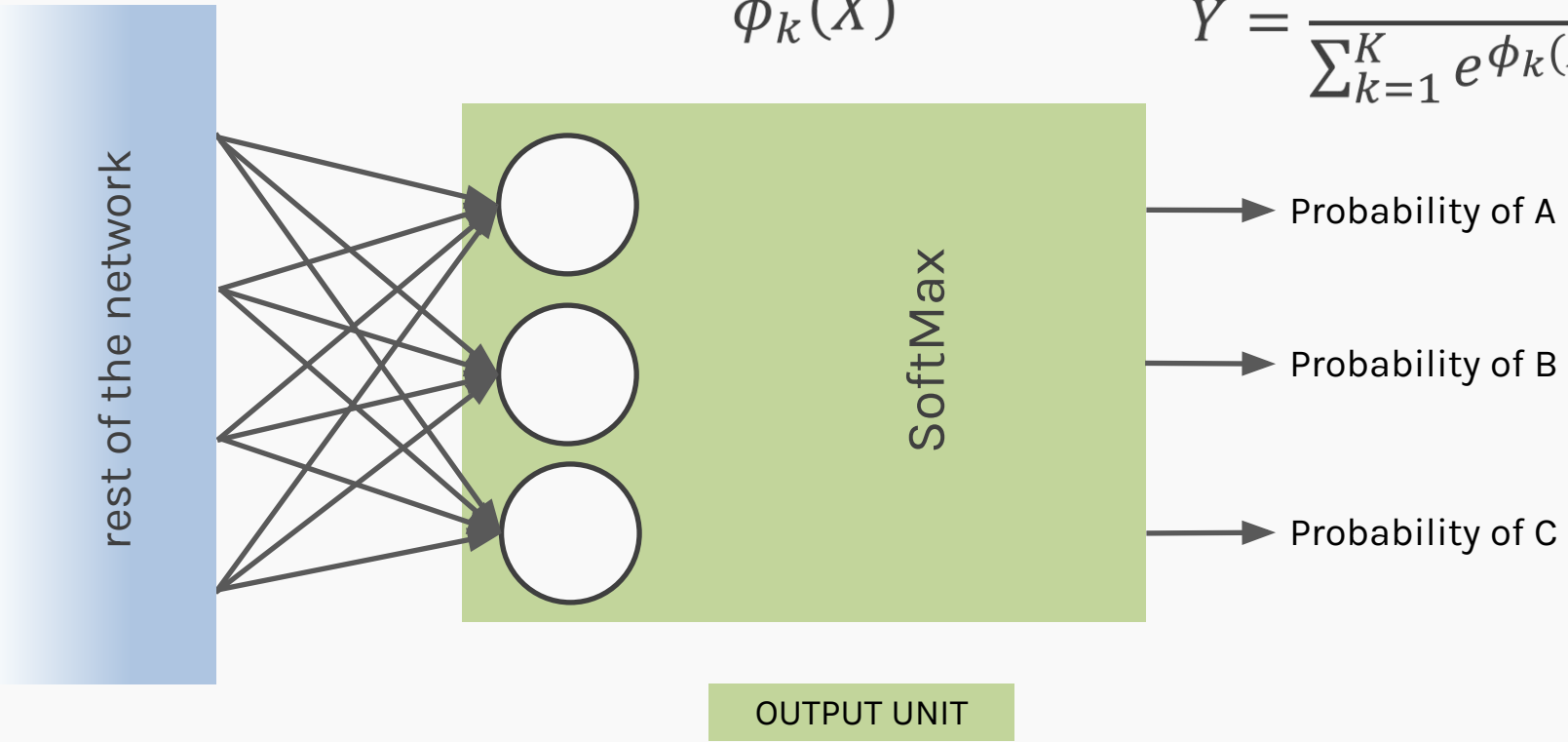
$$\phi_k(X)$$

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



$$\phi_k(X)$$

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



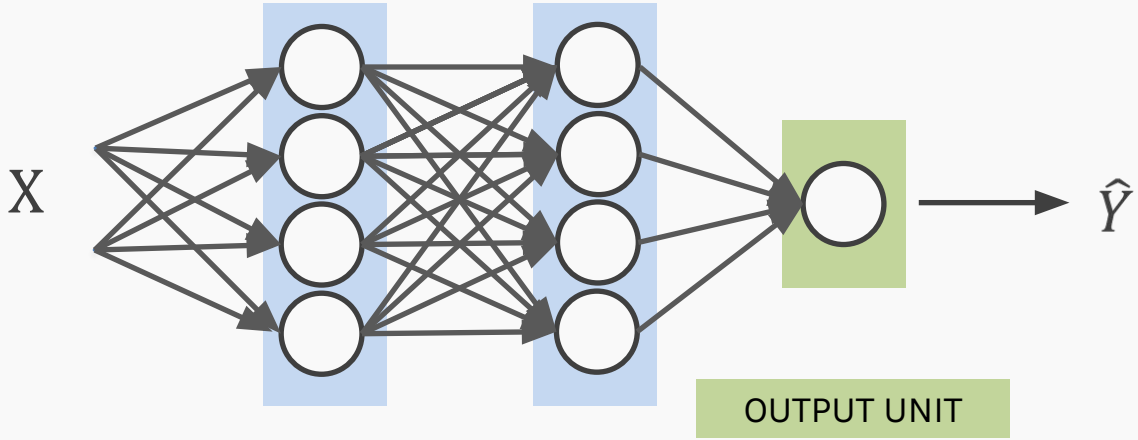
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy

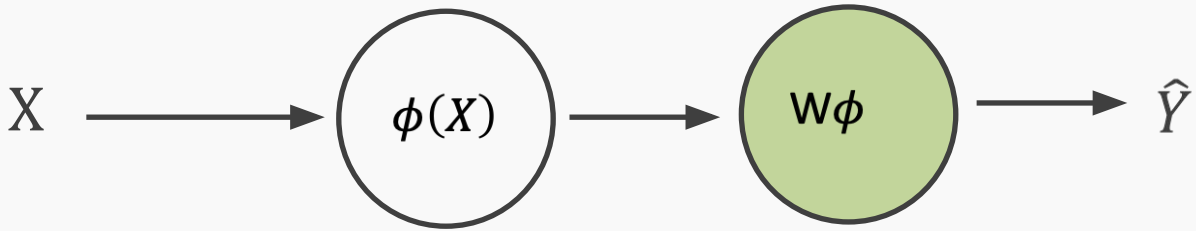
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous			

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian		

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

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	?	MSE





$$X \Longrightarrow \phi(X) \Longrightarrow \hat{Y} = W\phi(X)$$

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

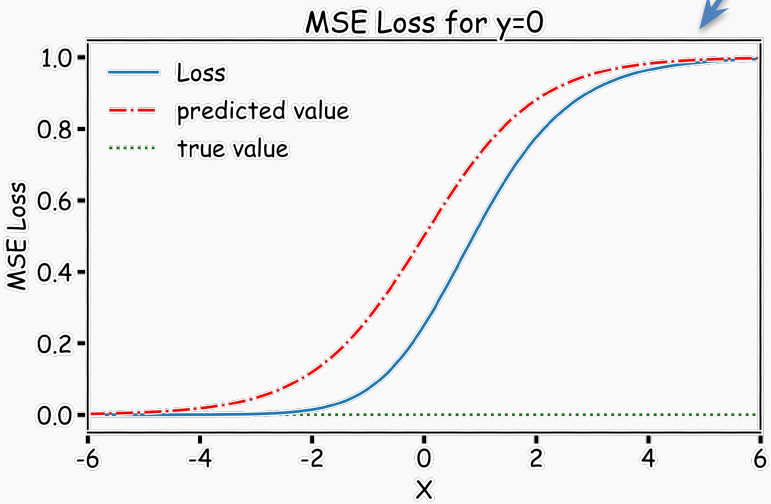
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	

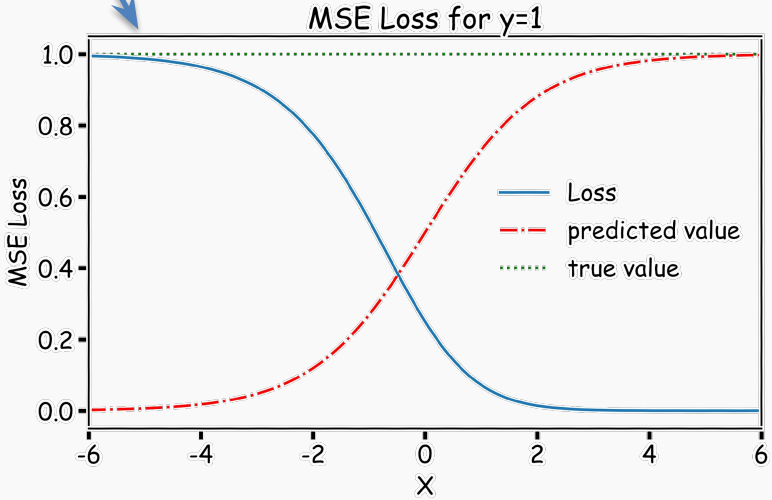
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	GANS

Example: sigmoid output + squared loss

$$L_{sq} = (y - \hat{y})^2 = \big(y - \sigma(x)\big)^2$$

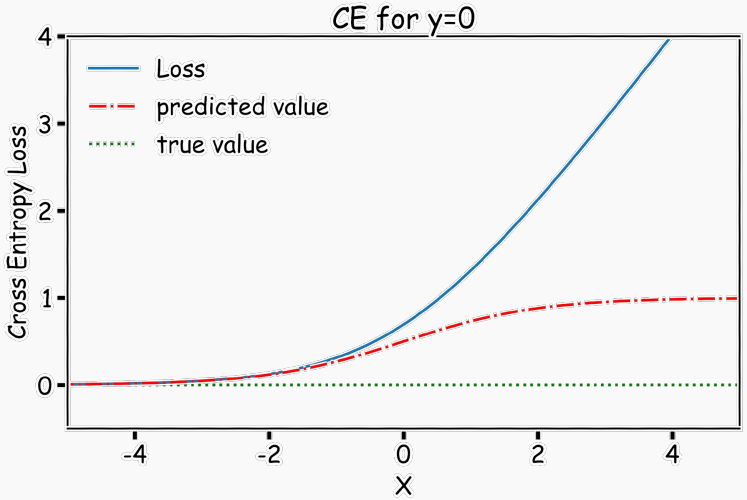
Flat surfaces

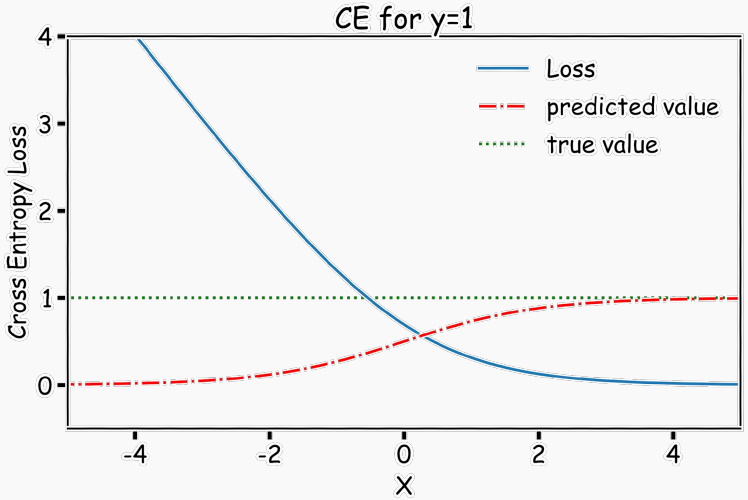




Example: sigmoid output + cross-entropy loss

$$L_{ce}(y, \hat{y}) = -\{\, y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \}$$





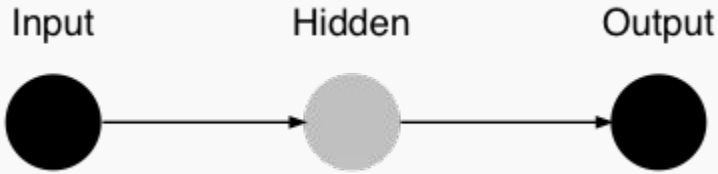
Activation function

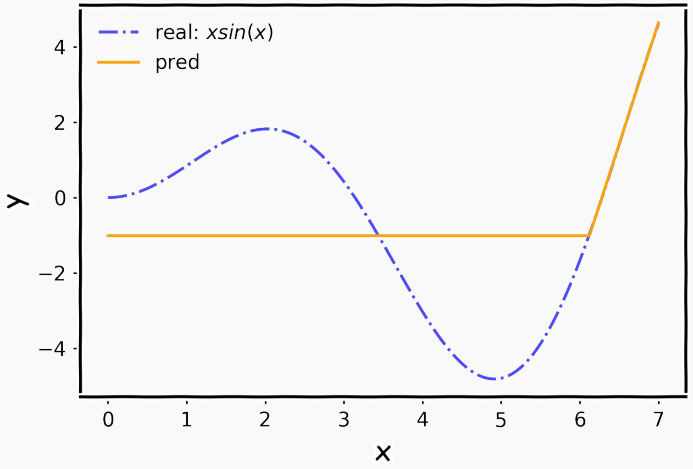
Loss function

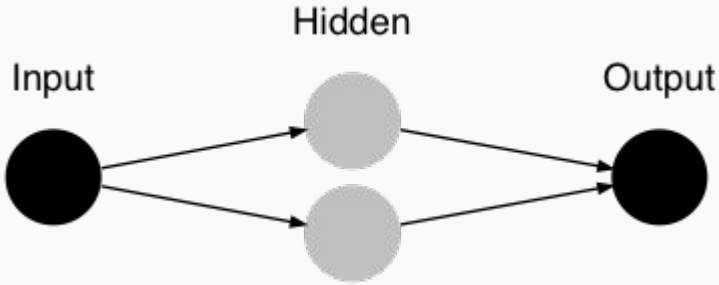
Output units

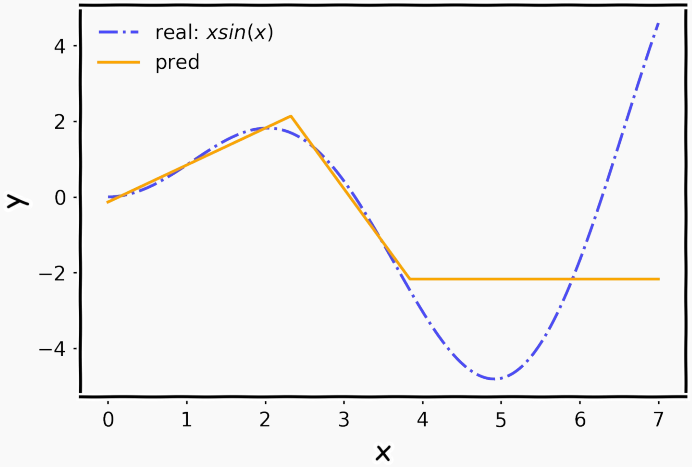
Architecture

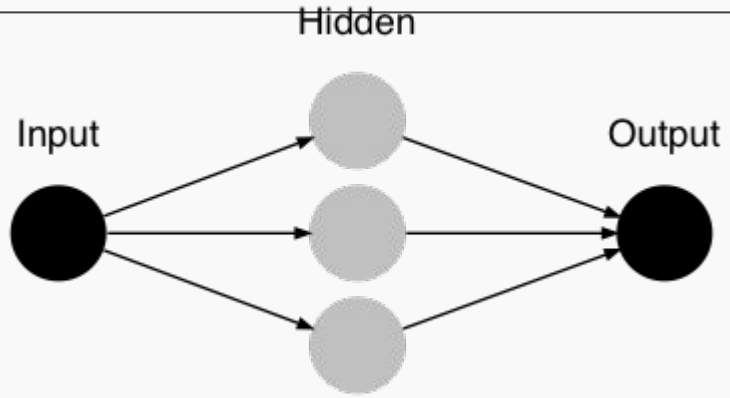
Optimizer

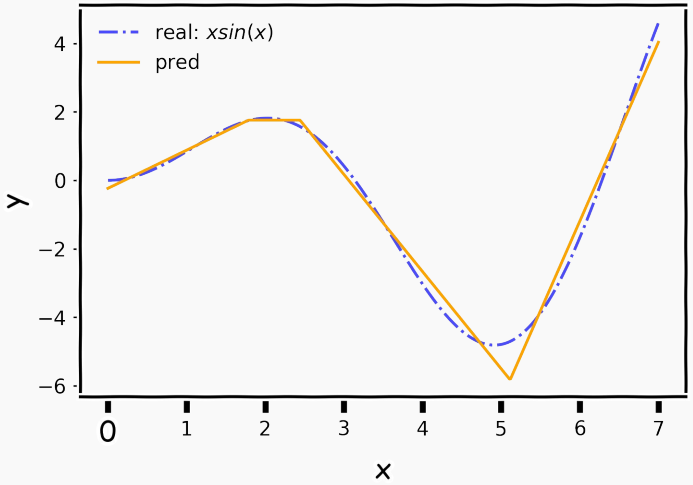

Input
Hidden
Output

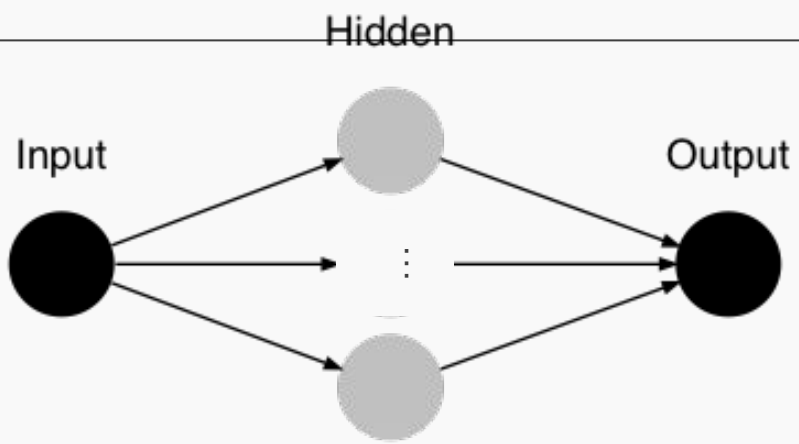

real: xsin(x)
pred
4
2
0
−2
−4
y
0
1
2
3
4
5
6
7
x

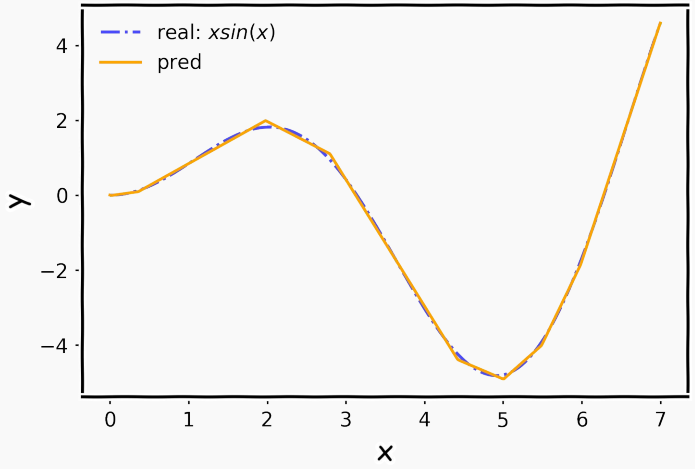

Hidden
Input
Output

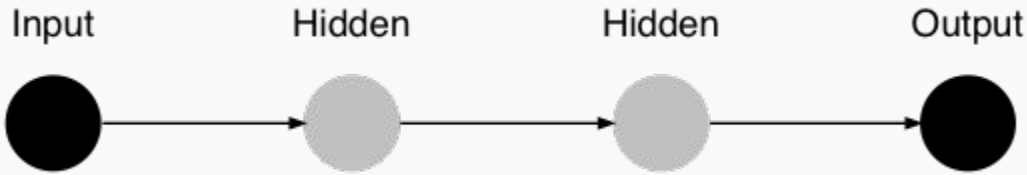

real: xsin(x)
pred
4
2
0
−2
−4
y
0
1
2
3
4
5
6
7
x

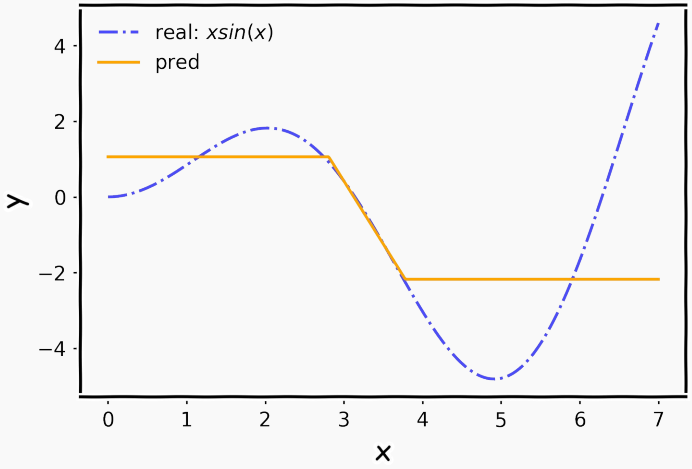

Hidden
Input
Output

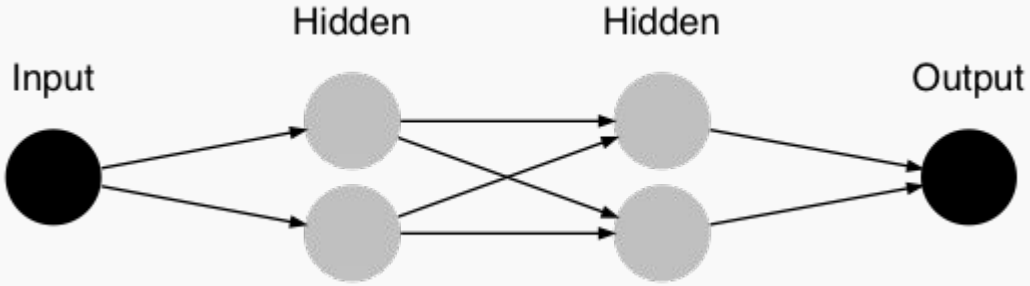

real: xsin(x)
pred
y
x
4
2
0
−2
−4
−6
0
1
2
3
4
5
6
7

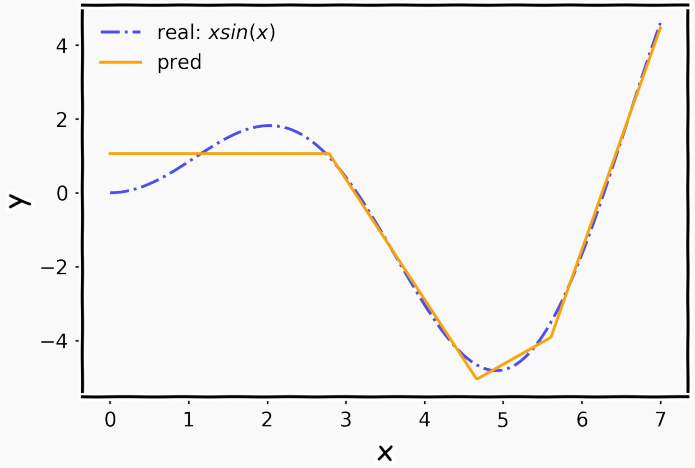

Hidden
Input
Output

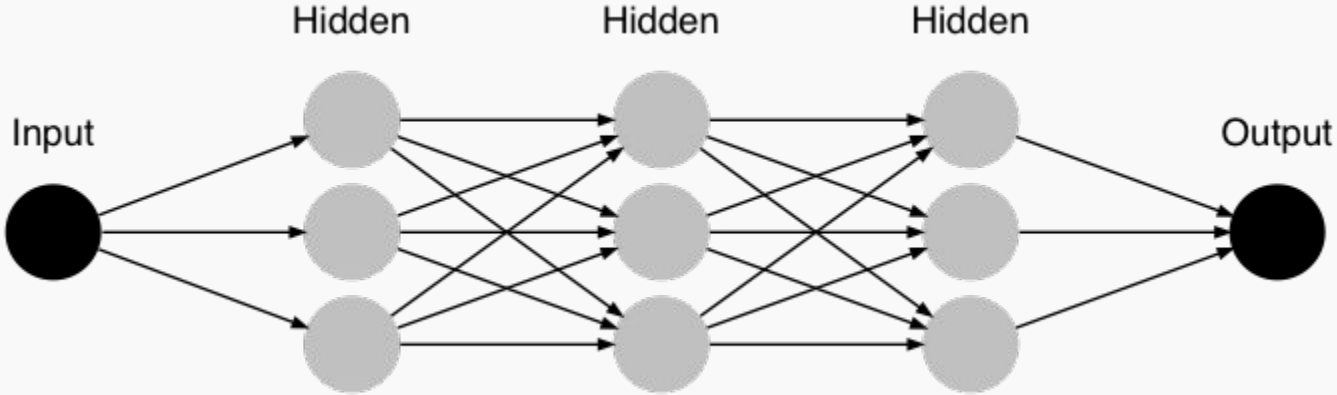

real: xsin(x)
pred
y
x
4
2
0
−2
−4
0
1
2
3
4
5
6
7

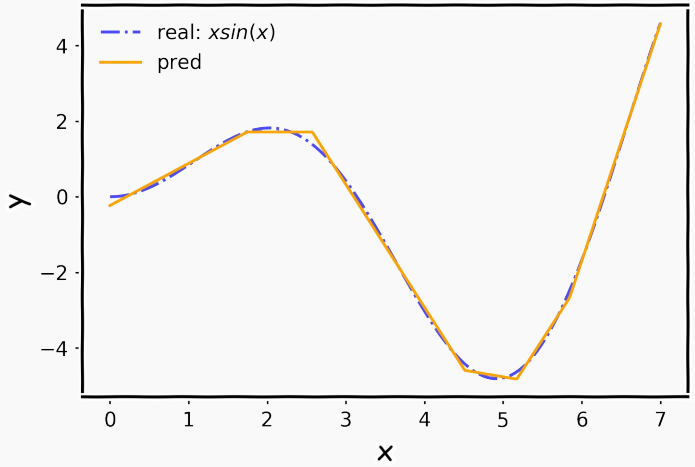

Input
Hidden
Hidden
Output


real: xsin(x)
pred
y
x
4
2
0
−2
−4
0
1
2
3
4
5
6
7

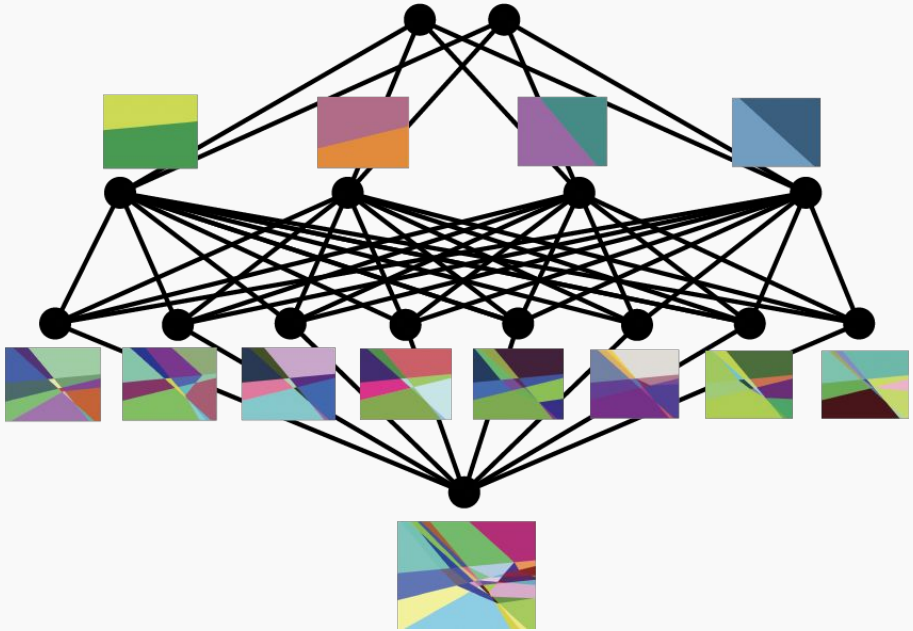

Input
Hidden
Hidden
Output


real: xsin(x)
pred
y
x
4
2
0
−2
−4
0
1
2
3
4
5
6
7

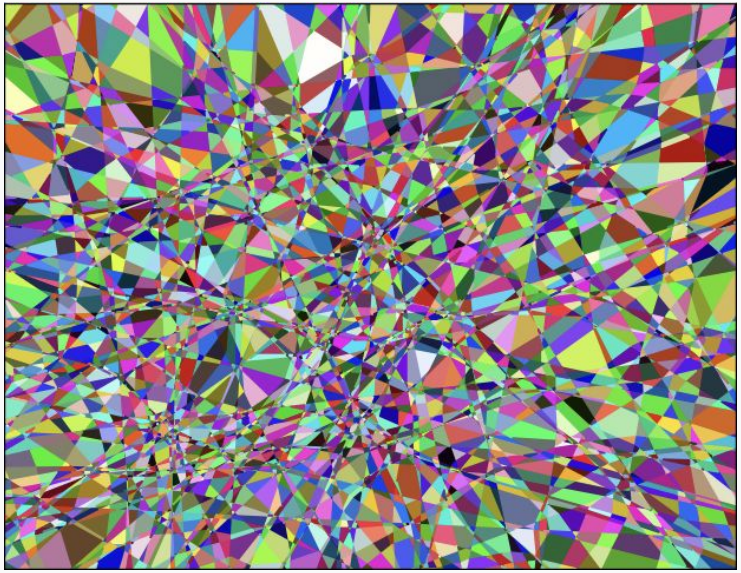

Hidden
Hidden
Hidden
Input
Output


real: xsin(x)
pred
y
x
4
2
0
−2
−4
0
1
2
3
4
5
6
7

Evolution of linear regions within a ReLU network for 2-dimensional input.

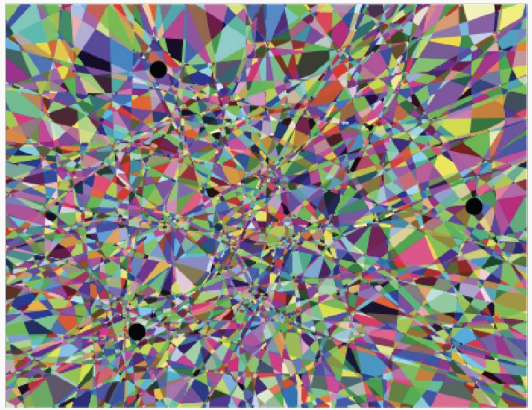


How many linear regions?

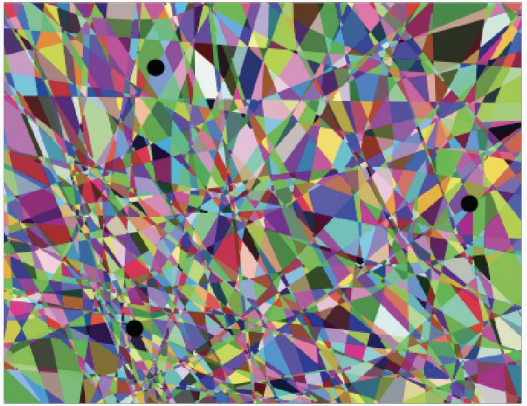


A two- dimensional slice through the 784-dimensional input space of vectorized MNIST, as represented by a fully-connected ReLU network with three hidden layers of width 64 each. Colors denote different linear regions of the piecewise linear network.

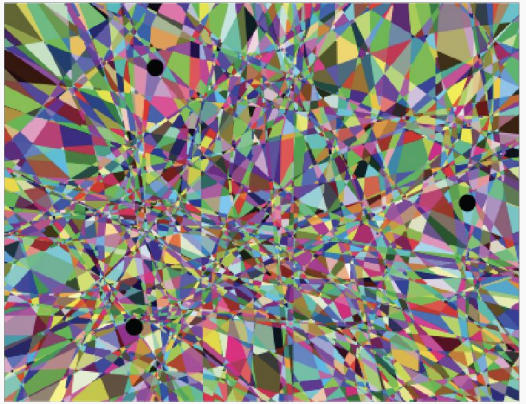
Epoch 0: 9744 regions



Epoch 1: 4196 regions



Epoch 20: 8541 regions



Linear regions that intersect a 2D plane through input space for a network of depth 3 and width 64 trained on MNIST. Black dots indicate the positions of the three MNIST training examples defining the plane.

Think of a Neural Network as function approximation.

$$Y = f(x) + \epsilon$$

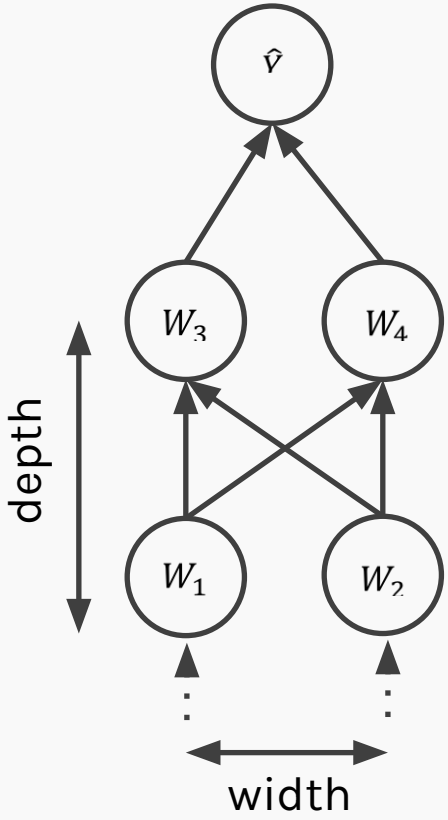
$$Y = \hat{f}(x) + \epsilon$$

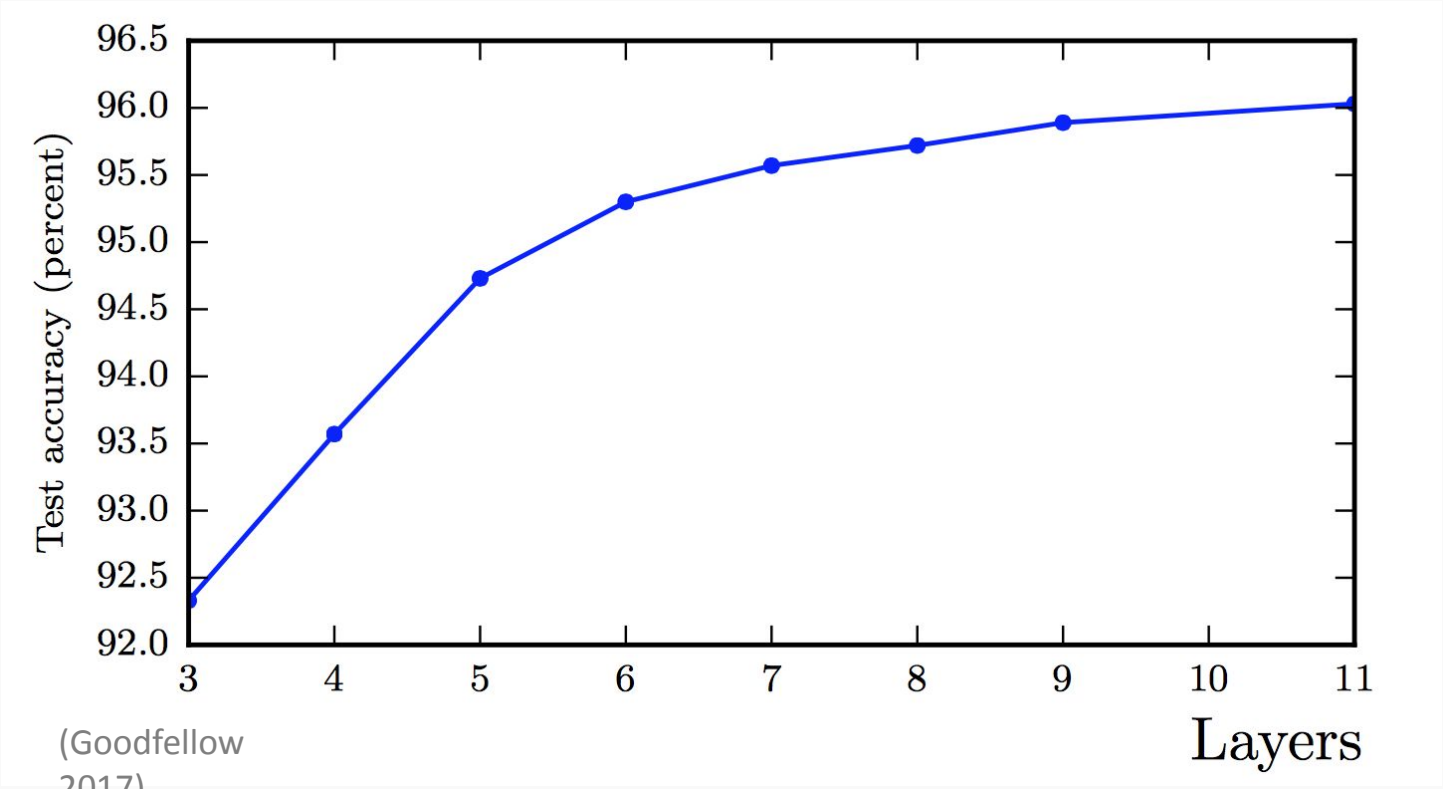
NN: $\Longrightarrow \hat{f}(x)$

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

So why deeper?

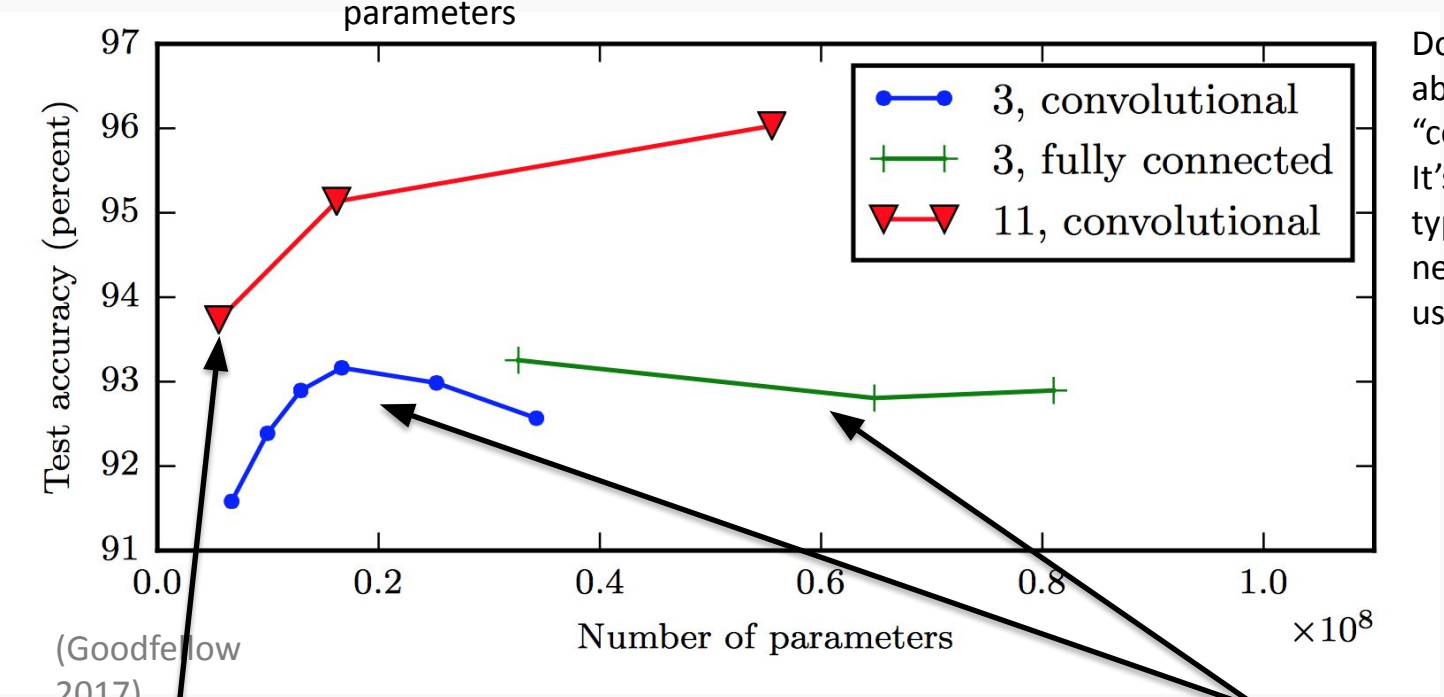
- Shallow net may need (exponentially) more width





(Goodfellow 2017)

Depth helps, and it's not just because of more parameters



(Goodfellow 2017)

Don't worry about this word "convolutional". It's just a special type of neural network, often used for images.

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.

Principal Components Analysis (PCA)

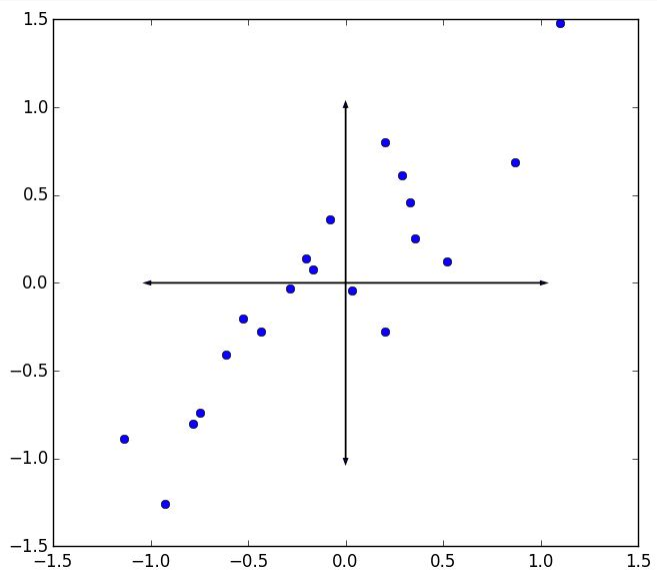
Assume we have n data samples, $x_1,\dots,x_n$, where $x_i \in R^p$. Assume that these observations have been normalized to have zero mean. We can stack these into a matrix:

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix}$$

And the covariance matrix becomes:

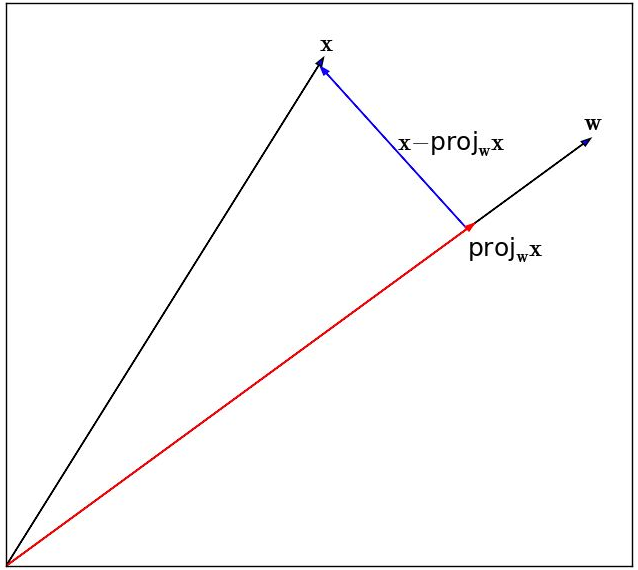
$$X^T X = \sum_j \mathbf{x}_j \mathbf{x}_j^T =: S \in \mathbb{R}^{p \times p}$$

Given a point set, what is the best single direction to project all the data onto?



Recall that the projection of a vector, x, onto the vector w is given by:

$$\text{proj}_{\mathbf{w}}\mathbf{x} = \frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}\mathbf{x}$$



so that the squared error incurred from projecting x onto w is

$$\text{squared error} = ||\mathbf{x} - \frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}\mathbf{x}||^2$$

Thus, we seek a single direction, w, which minimizes:

$$f(\mathbf{w}) = \sum_{i=1}^{n} ||\mathbf{x}_i - \frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}\mathbf{x}_i||^2$$
$$= \sum_i \mathbf{x}_i^T\mathbf{x}_i - \frac{(\mathbf{x}_i^T\mathbf{w})^2}{\mathbf{w}^T\mathbf{w}}$$

Writing the second term a little differently reveals something interesting:

$$f(\mathbf{w}) = \sum_i \mathbf{x}_i^T\mathbf{x}_i - \sum_i \frac{\mathbf{w}^T\mathbf{x}_i\mathbf{x}_i^T\mathbf{w}}{\mathbf{w}^T\mathbf{w}}$$

or bringing the sum inside the second term...

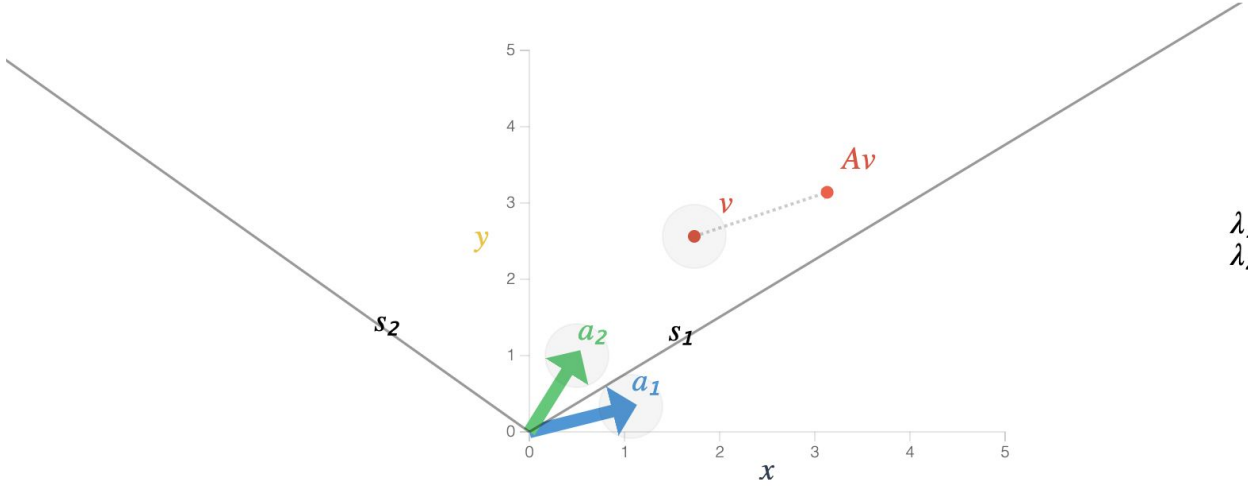
$$f(\mathbf{w}) = \sum_i \mathbf{x}_i^T\mathbf{x}_i - \frac{\mathbf{w}^T S\mathbf{w}}{\mathbf{w}^T\mathbf{w}}$$

$$f(\mathbf{w}) = \sum_i \mathbf{x}_i^T \mathbf{x}_i - \frac{\mathbf{w}^T S \mathbf{w}}{\mathbf{w}^T \mathbf{w}}$$

The second term is the Rayleigh quotient for eigvenvalues of S.

Thus to minimize this function, we should set w to be the eigenvector of S corresponding the largest eigenvalue of S.

let v be a vector (shown as a point) and A be a matrix with 2 columns. If we multiply v by A, then A sends v to a new vector Av.



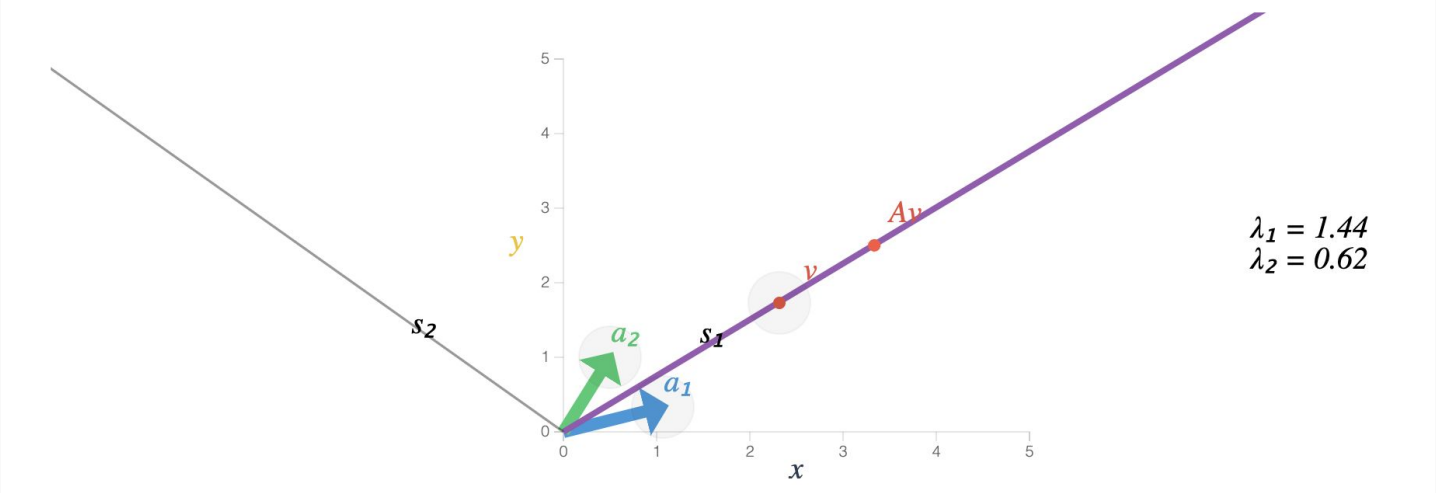
$\lambda_1 = 1.44$
$\lambda_2 = 0.62$

$$A = \begin{bmatrix} a_1,x & a_2,x \\ a_1,y & a_2,y \end{bmatrix} = \begin{bmatrix} 1.07 & 0.50 \\ 0.33 & 1.00 \end{bmatrix}$$

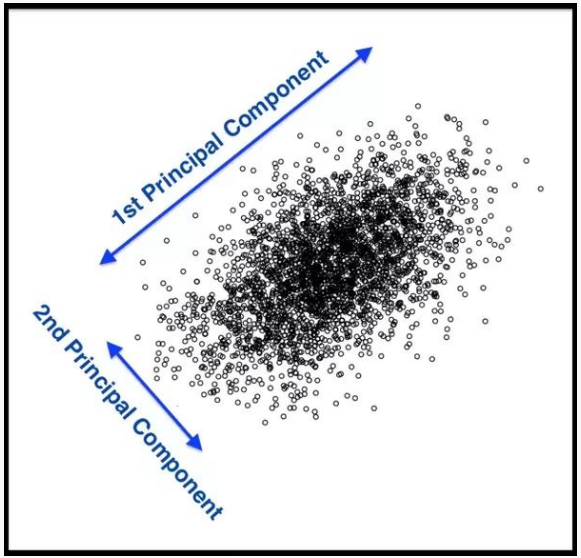
$$v = \begin{bmatrix} 2.32 \\ 1.73 \end{bmatrix}$$

$$Av = \begin{bmatrix} 3.34 \\ 2.50 \end{bmatrix}$$

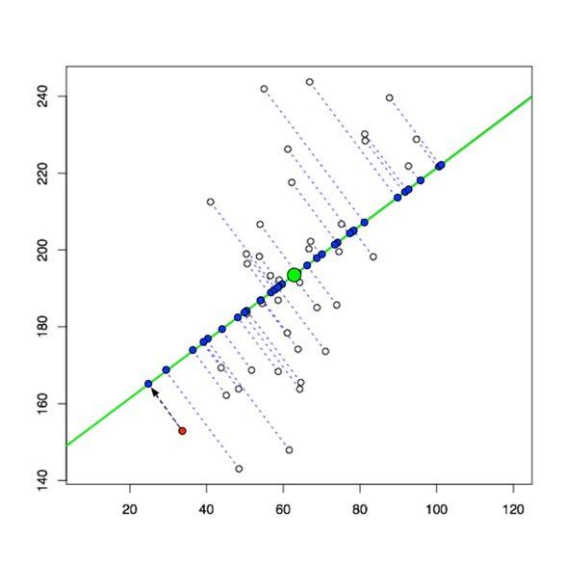
If you can draw a line through the three points (0,0), v and Av, then Av is just v multiplied by a number λ; that is, Av = λv. In this case, we call λ an **eigenvalue** and v an **eigenvector**.



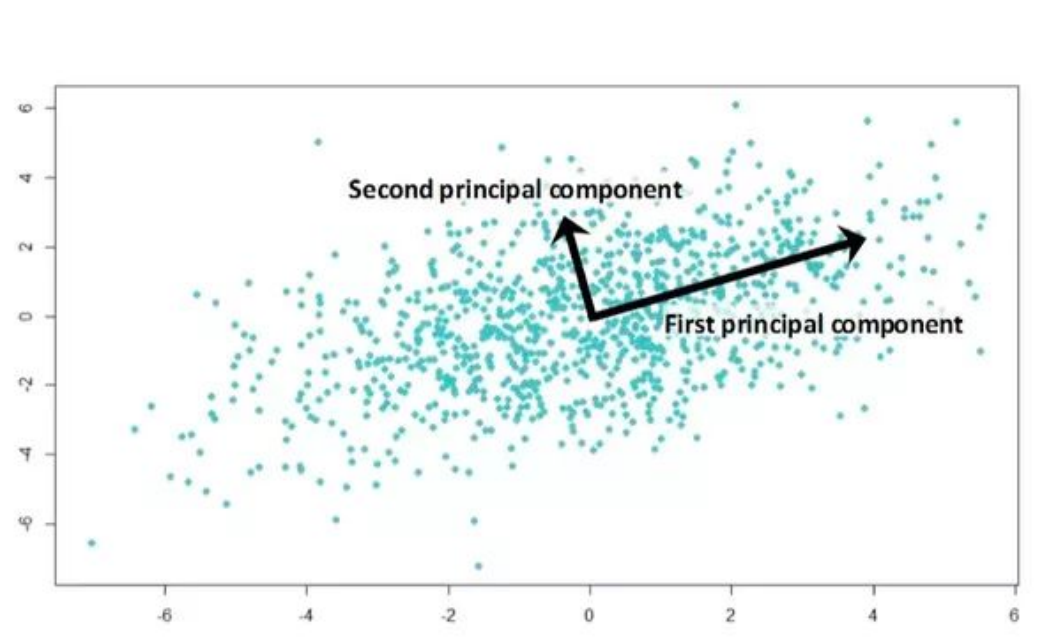
Principal Components Analysis (PCA) is a method to identify a new set of predictors, as linear combinations of the original ones, that captures the `maximum amount' of variance in the observed data.



Transforming our observed data means projecting our dataset onto the space defined by the top *m* PCA components, these components are our new predictors.



Since the mean of the projections is zero, minimizing the residual sum of squares turns out to be equivalent to maximizing the variance of the projections.



Now, lets minimize the residual of the projection on w:

$$\begin{aligned} \|\vec{x_i} - (\vec{w} \cdot \vec{x_i})\vec{w}\|^2 &= \|\vec{x_i}\|^2 - 2(\vec{w} \cdot \vec{x_i})(\vec{w} \cdot \vec{x_i}) + \|\vec{w}\|^2 \\ &= \|\vec{x_i}\|^2 - 2(\vec{w} \cdot \vec{x_i})^2 + 1 \end{aligned}$$

Equivalently, we want to maximize : $\frac{1}{n}\sum_{i=1}^{n}(\vec{w} \cdot \vec{x_i})^2$

The mean of a square is always equal to the square of the mean plus the variance:

$$\frac{1}{n}\sum_{i=1}^{n}(\vec{w} \cdot \vec{x_i})^2 = \left(\frac{1}{n}\vec{x_i} \cdot \vec{w}\right)^2 + \mathrm{Var}\left[\vec{w} \cdot \vec{x_i}\right]$$

Since the mean of the projections is zero, minimizing the residual sum of squares is equivalent to maximizing the variance of the projections

We made it!

Questions?