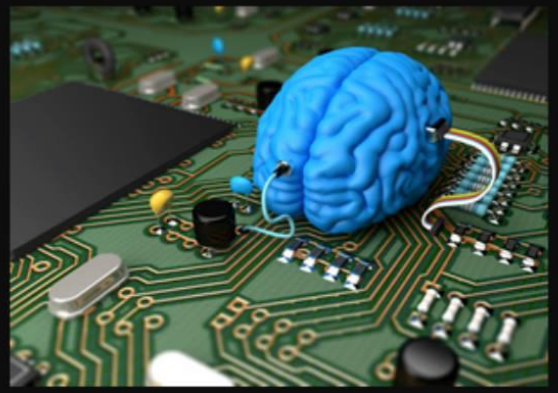
ENCS5341
Machine Learning and Data Science

Neural Networks

Deep Learning

What society thinks I do



What my friends think I do

What other computer scientists think I do

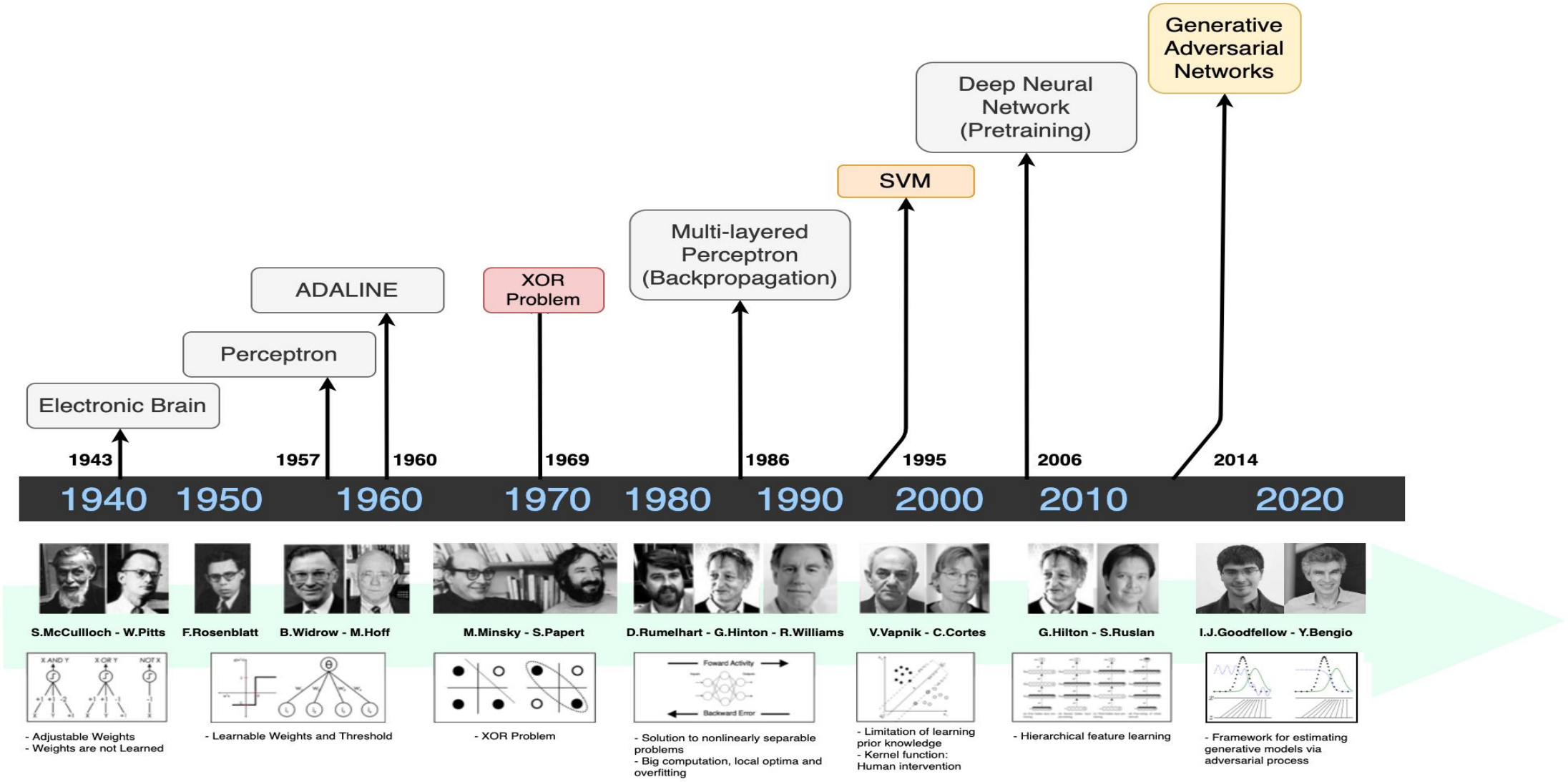
What mathematicians think I do

What I think I do

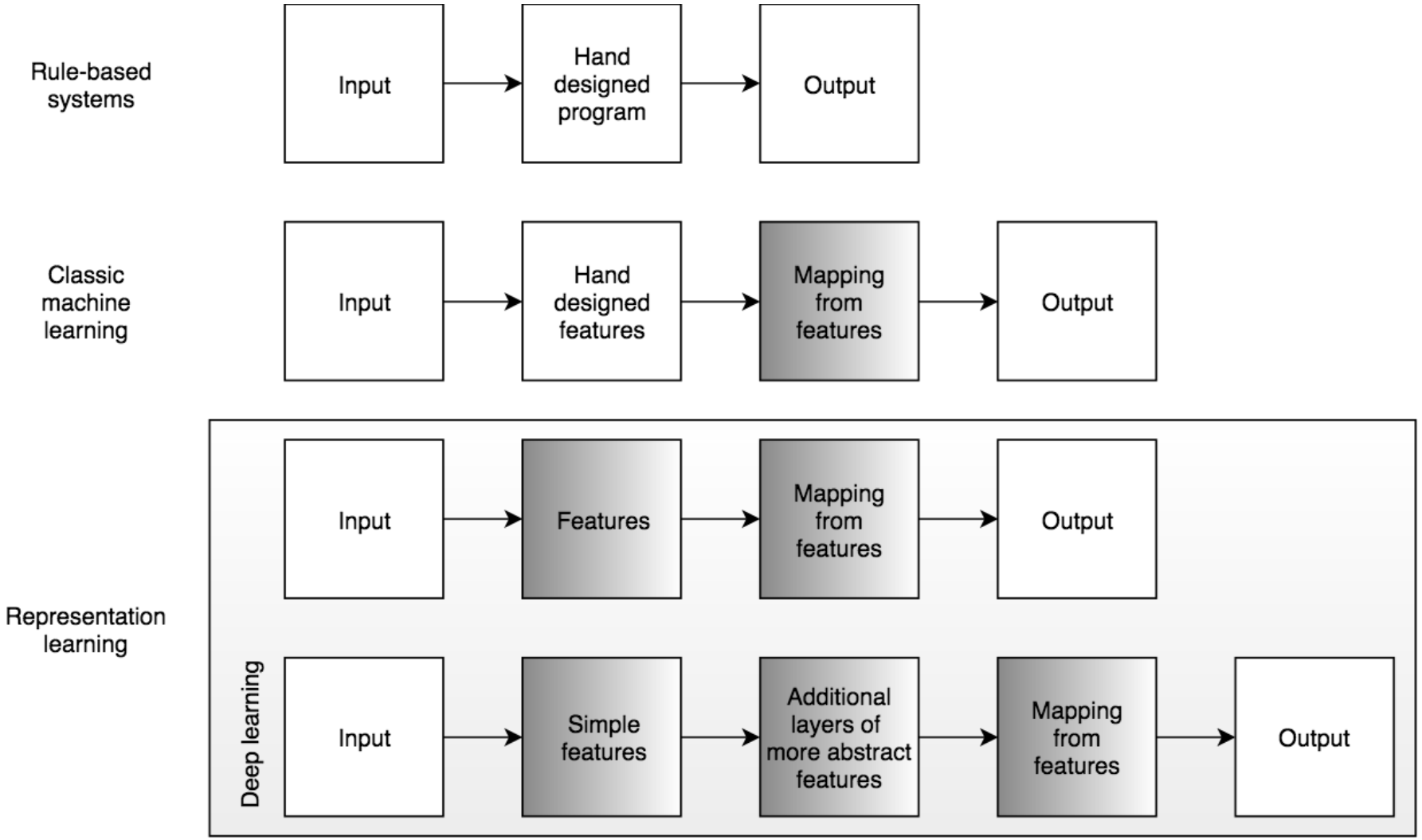
```
import keras
```

What I actually do

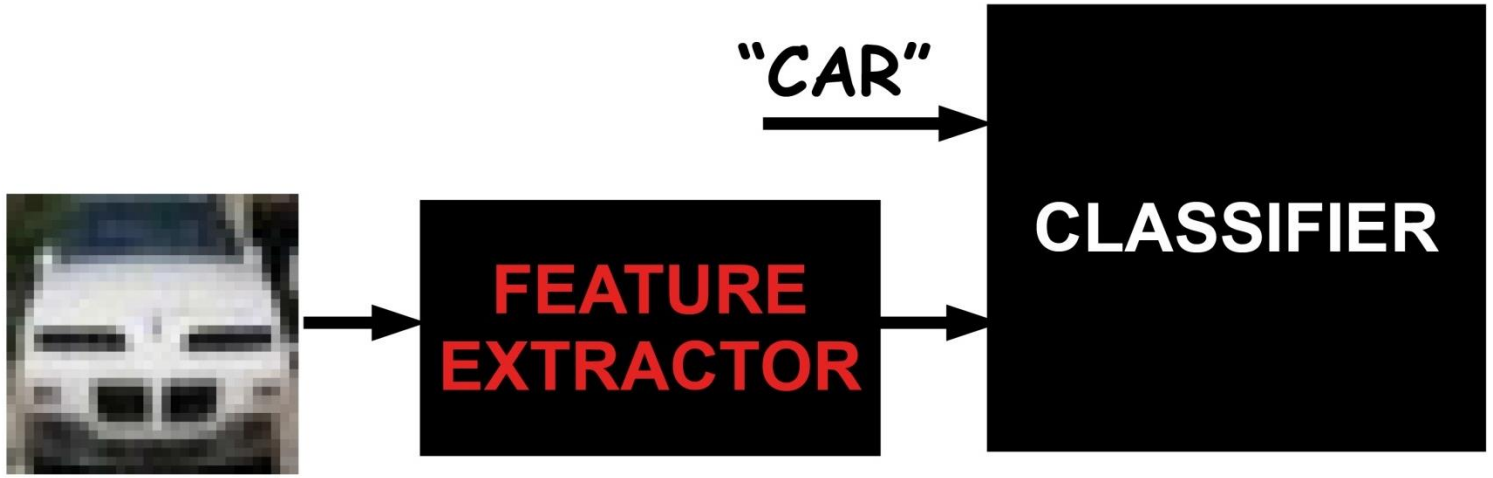
Historical Trends



Introduction: Learning Multiple Components

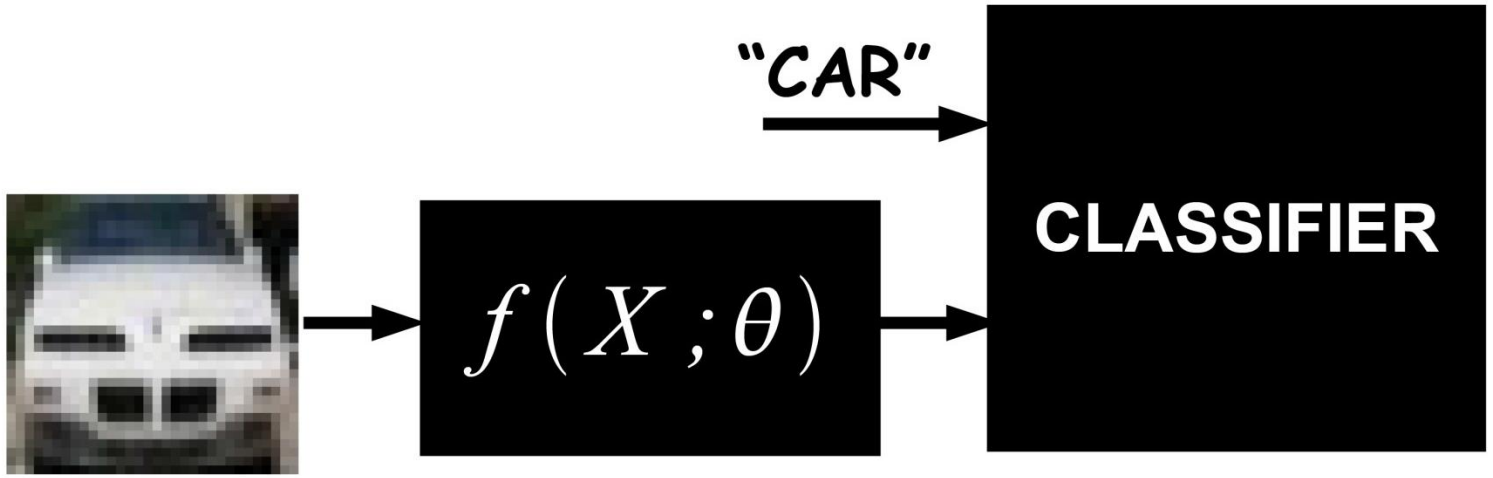


Introduction: Building an Object Recognition System



IDEA: Use data to optimize features for the given task.

Introduction: Building an Object Recognition System



What we want: Use parameterized function such that
a) features are computed efficiently
b) features can be trained efficiently

Introduction: Building an Object Recognition System

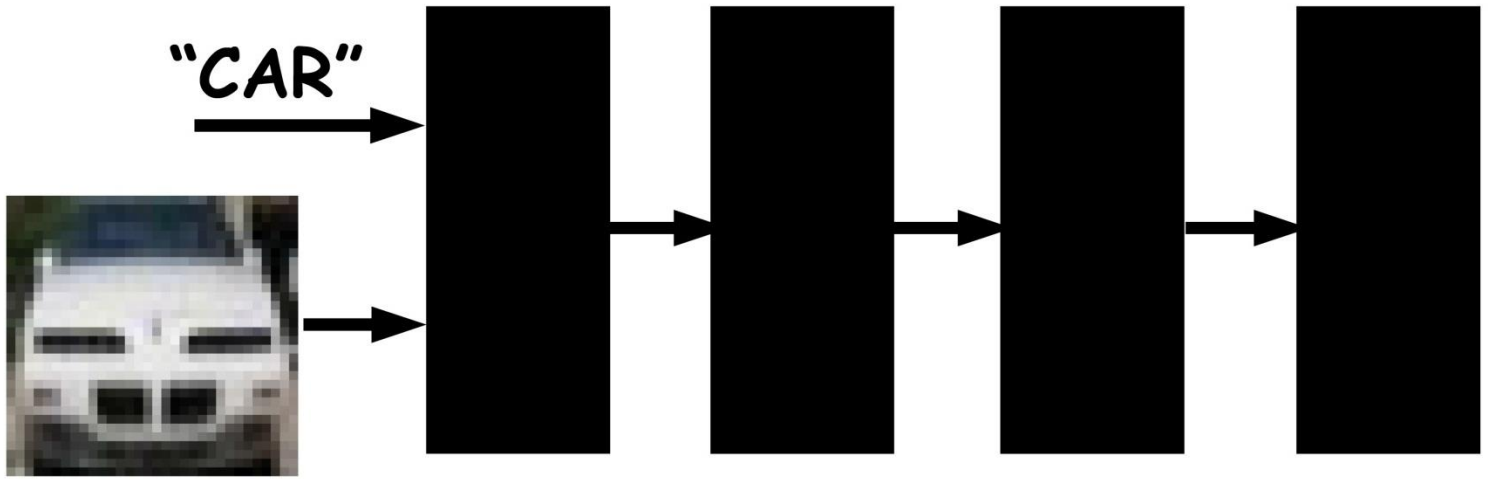
- Everything becomes adaptive.
- No distiction between feature extractor and classifier.
- Big non-linear system trained from raw pixels to labels.

Introduction: Building an Object Recognition System

Q: How can we build such a highly non-linear system?

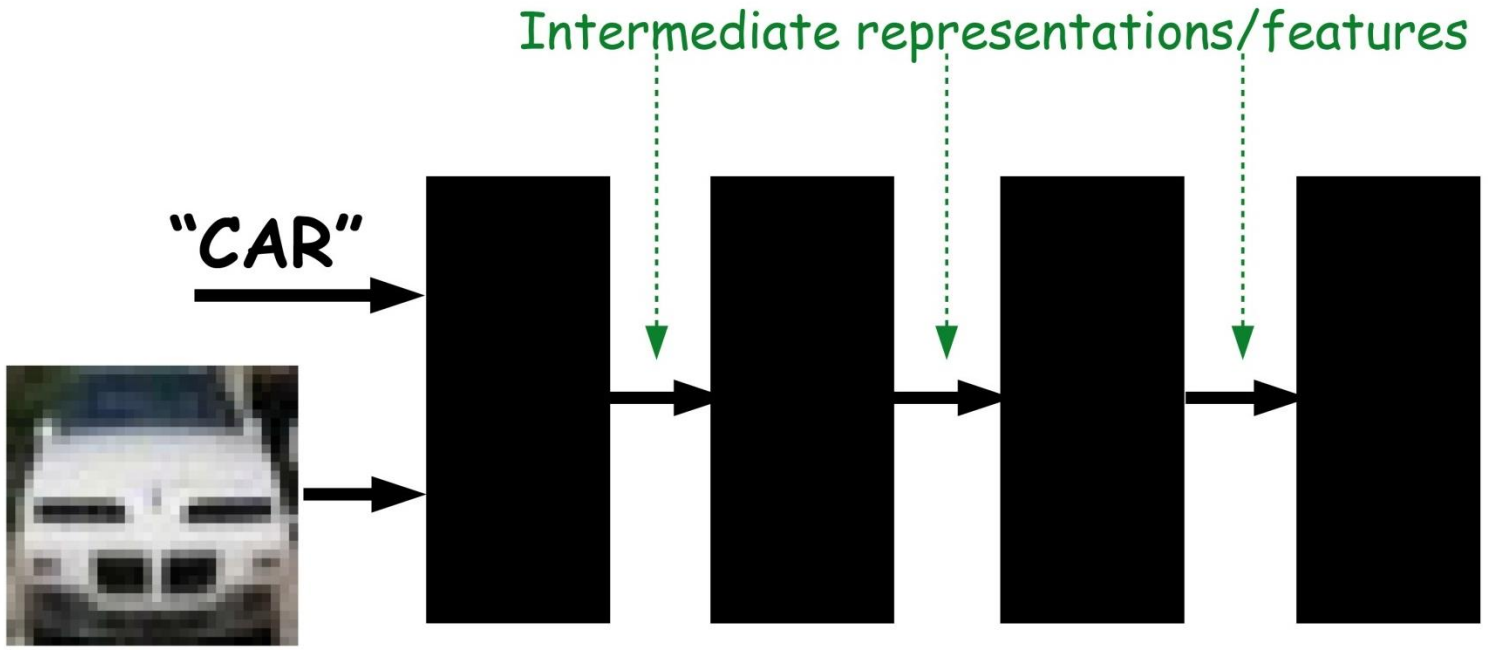
A: By combining simple building blocks we can make more and more complex systems.

Intuition Behind Deep Neural Nets



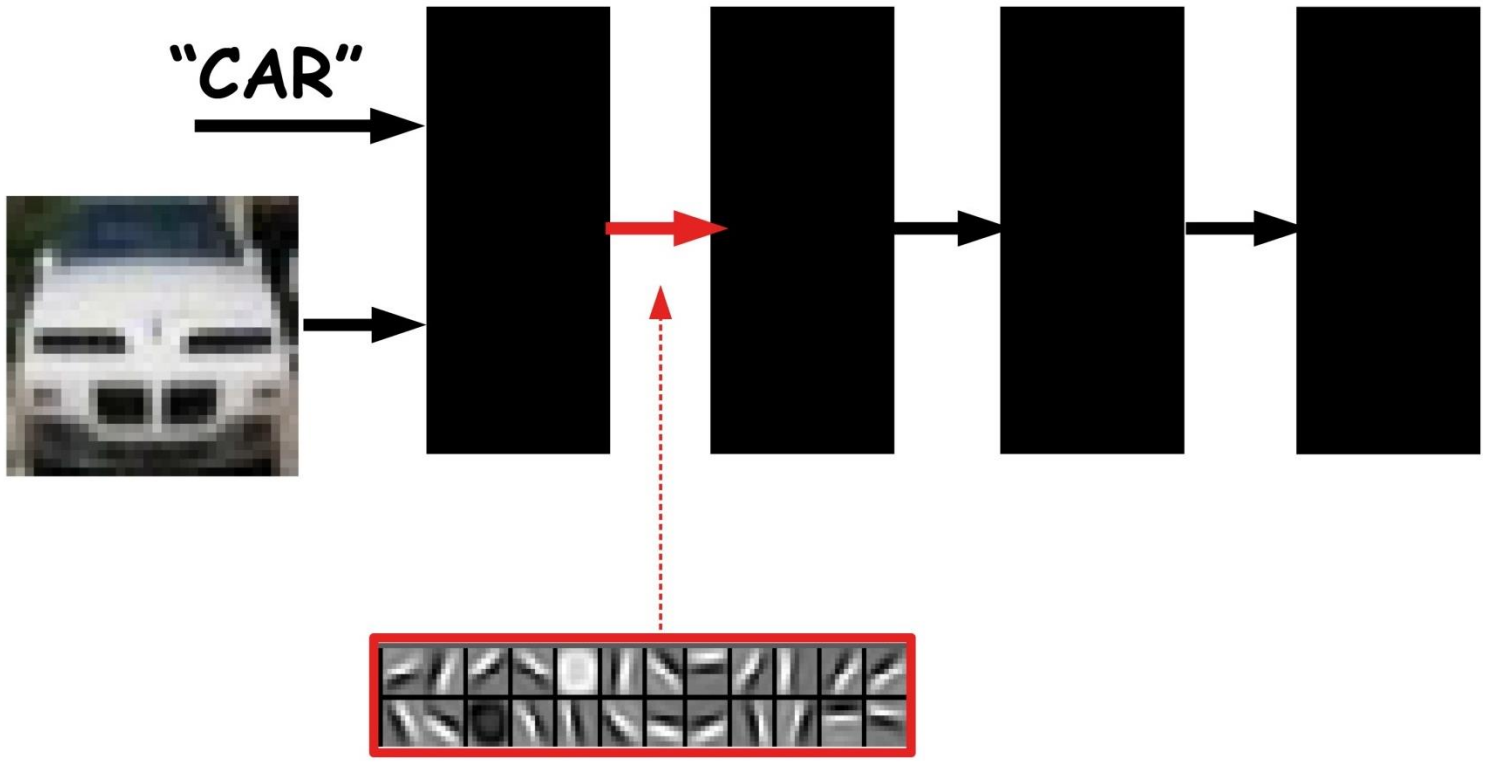
NOTE: Each black box can have trainable parameters.
Their composition makes a highly non-linear system.

Intuition Behind Deep Neural Nets



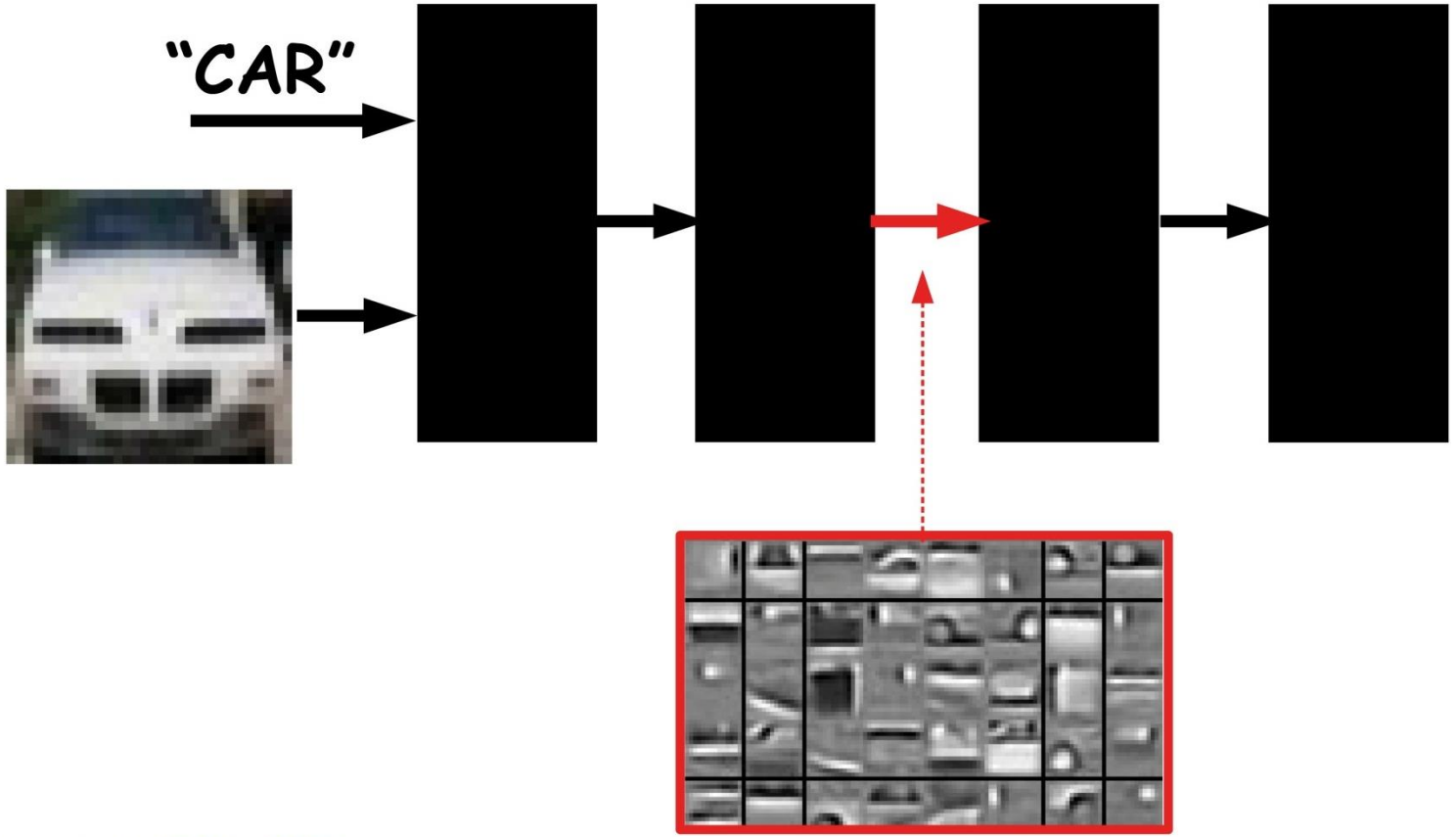
NOTE: System produces a hierarchy of features.

Intuition Behind Deep Neural Nets



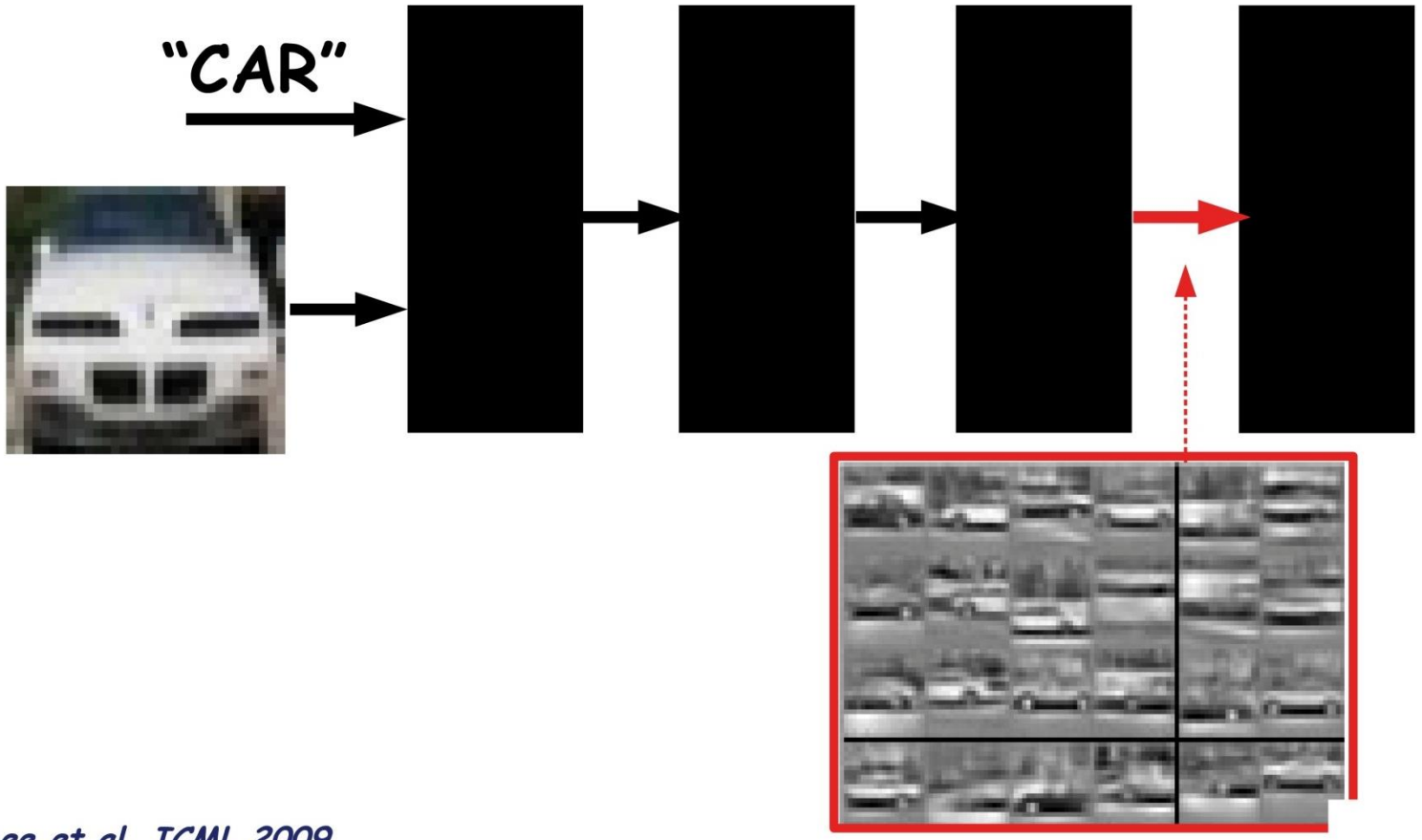
Lee et al. "Convolutional DBN's for scalable unsup. learning..." ICML 2009

Intuition Behind Deep Neural Nets



Lee et al. ICML 2009

Intuition Behind Deep Neural Nets



Lee et al. ICML 2009

Marc'Aurelio Ranzat

Key Ideas of Neural Nets

IDEA # 1

Learn features from data

IDEA # 2

Use differentiable functions that produce features efficiently

IDEA # 3

End-to-end learning:
no distinction between feature extractor and classifier

IDEA # 4

"Deep" architectures:
cascade of simpler non-linear modules

Perceptrons

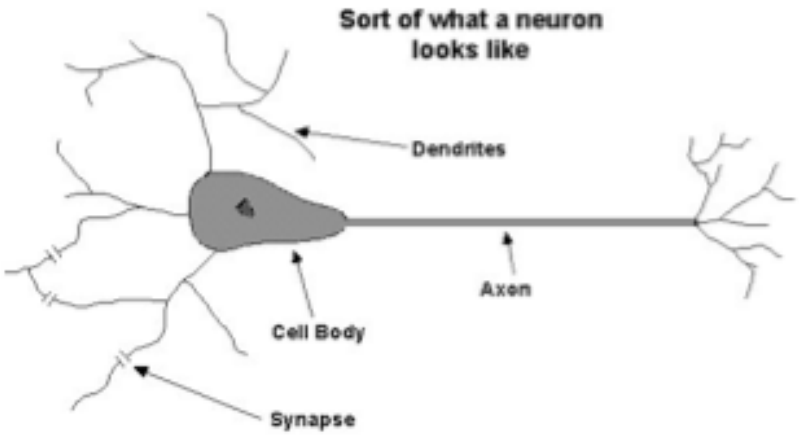
Perceptrons

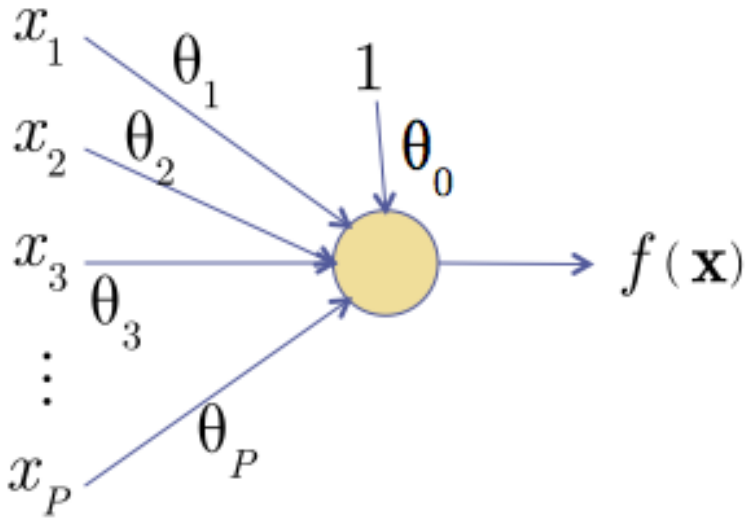
Leading to

- Neural Networks
- aka Multilayer Perceptron Networks
- aka Deep Learning

Neuron inspires Regression

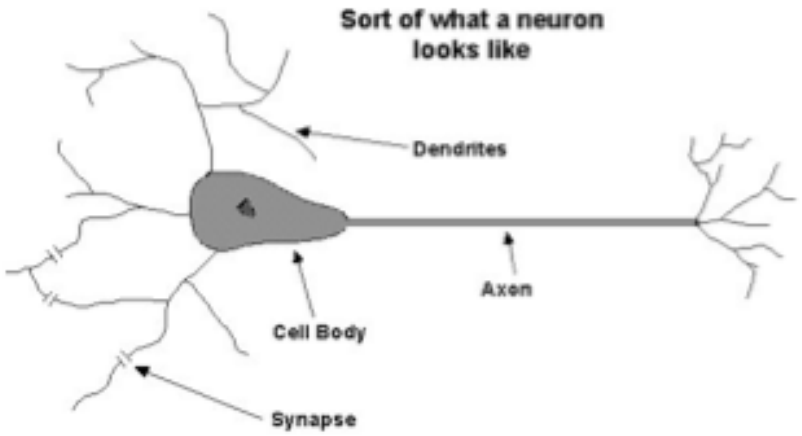
- Graphical Representation of linear regression
- McCullough-Pitts Neuron
- Note: Not a graphical model

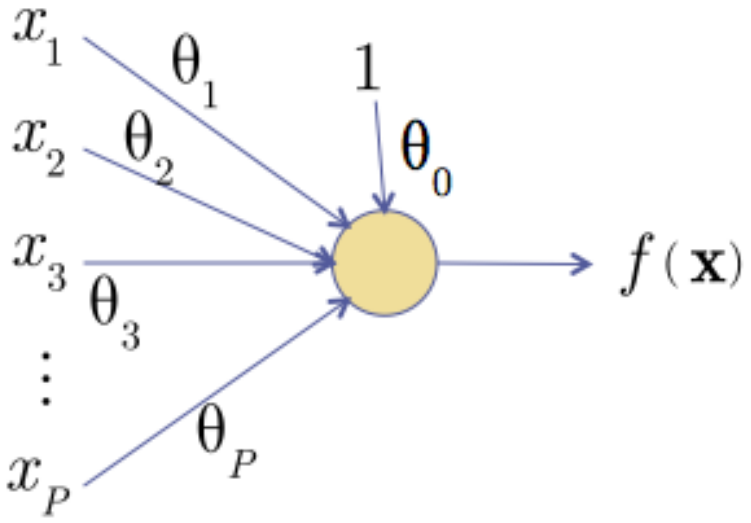




Neuron inspires Regression

- Edges multiply the signal x_i by some weight θ_i.
- Nodes sum inputs: $$f(x, \theta) = \sum_{d=1}^{D} \theta_d x^d + \theta_0$$
- Equivalent to Linear Regression

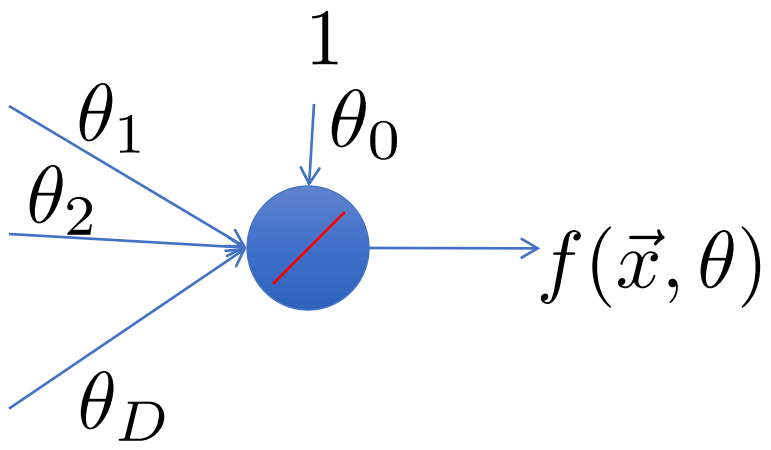




Activation function

- How do we construct the neural output

$$f(\vec{x}, \theta) = \theta^T \vec{x}$$

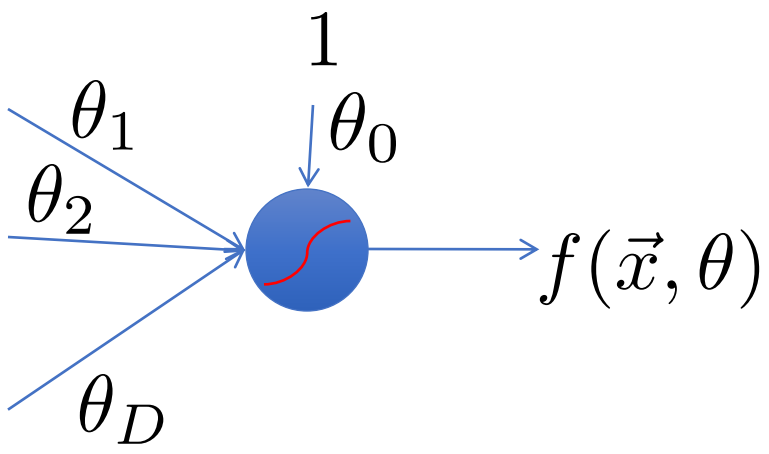


Linear Neuron

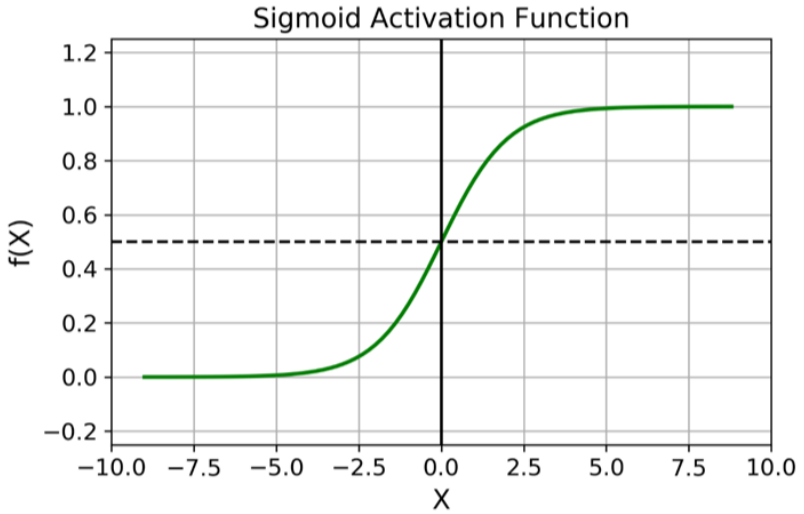
Activation function

- Sigmoid function or Squashing function

$$f(\vec{x},\theta) = g(\theta^T \vec{x}) \qquad g(z) = (1 + exp(-z))^{-1}$$



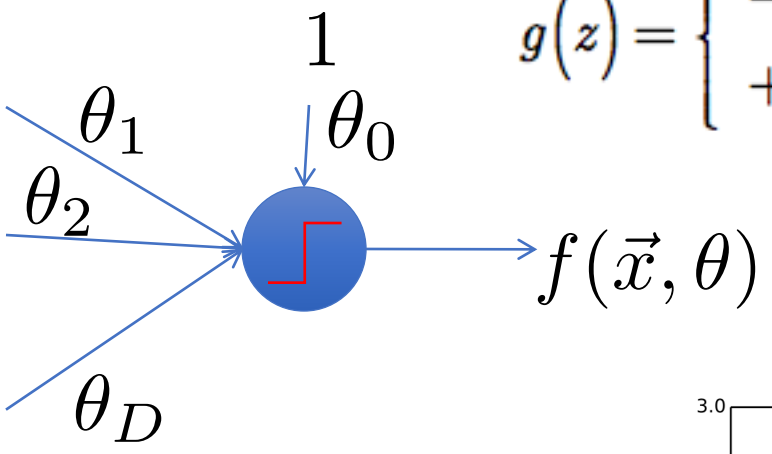
Logistic Neuron



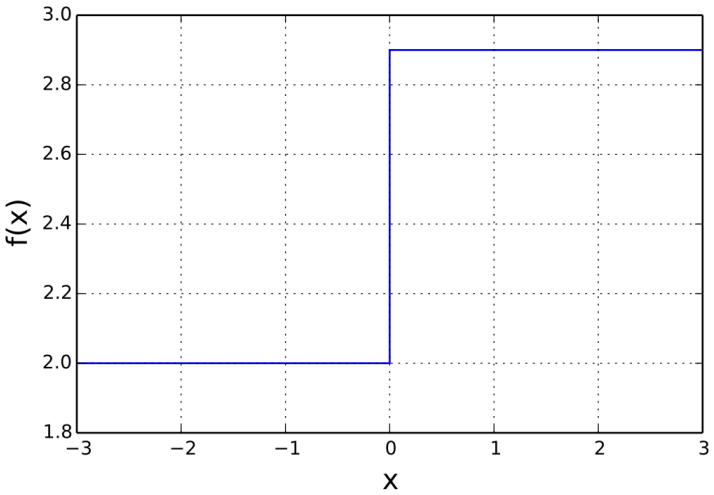
Perceptron

- Classification squashing function

$$g(z) = \begin{cases} -1 \text{ when } z < 0 \\ +1 \text{ when } z \geq 0 \end{cases}$$



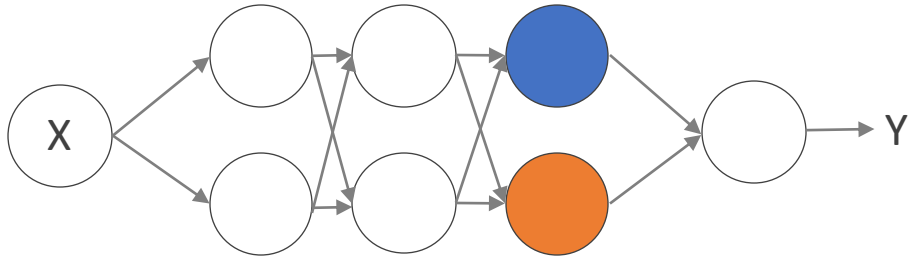
Perceptron

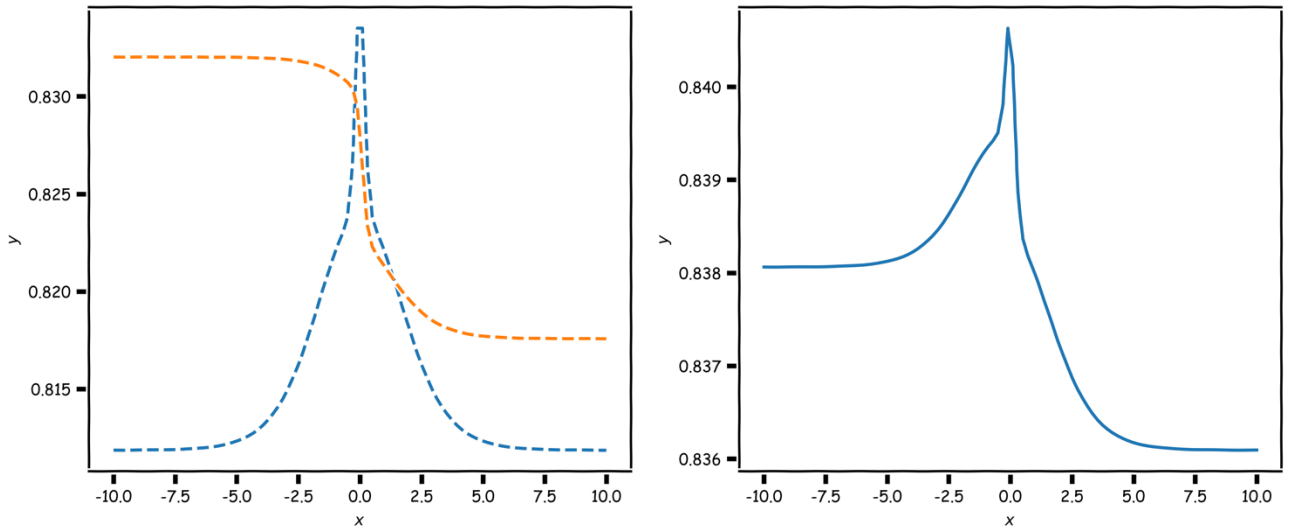


The central idea behind Neural Networks

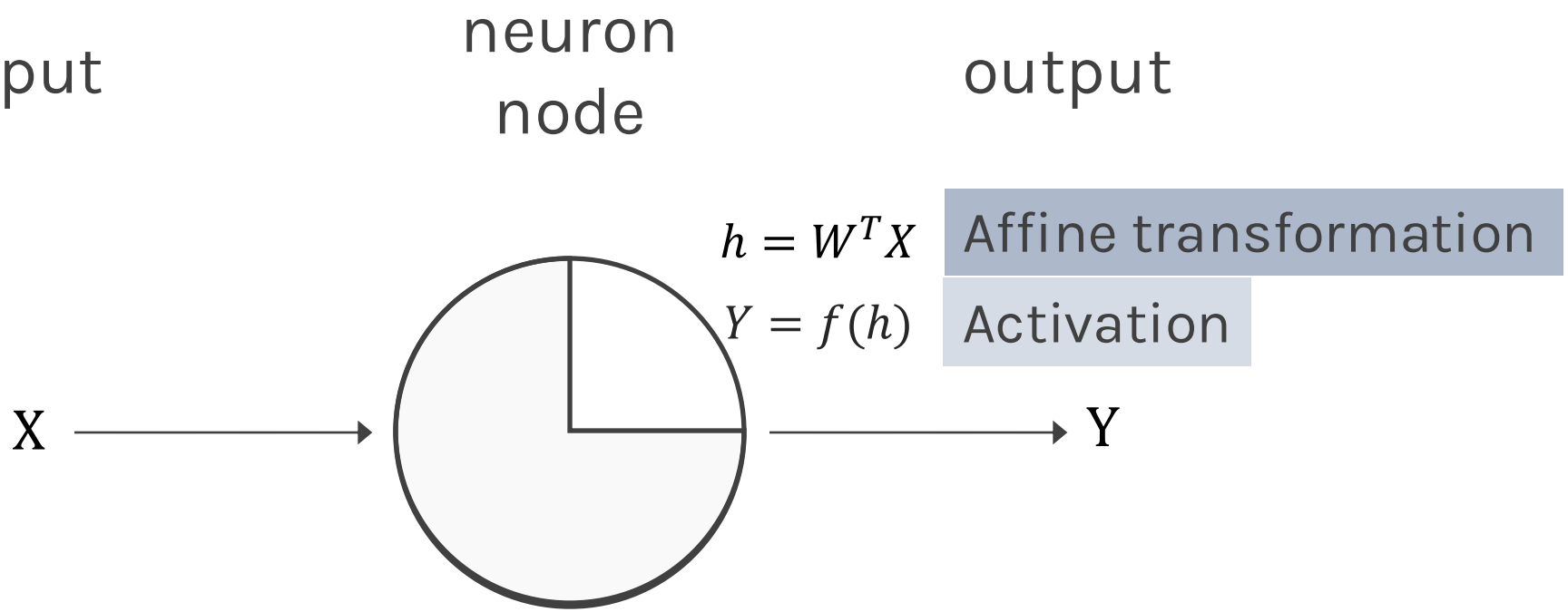
Building a model that is both complex and simple to manipulate by composing multiple functions together.

Example:



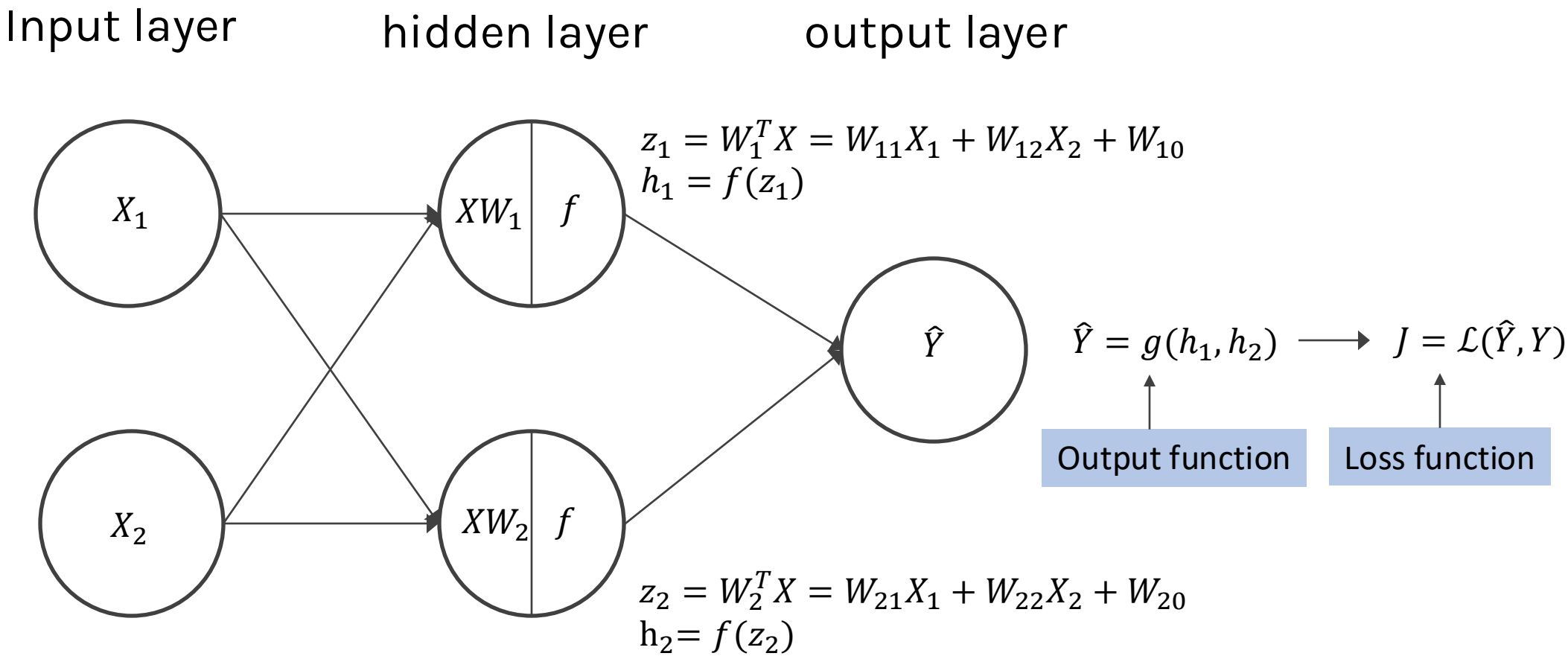


Anatomy of artificial neural network (ANN)



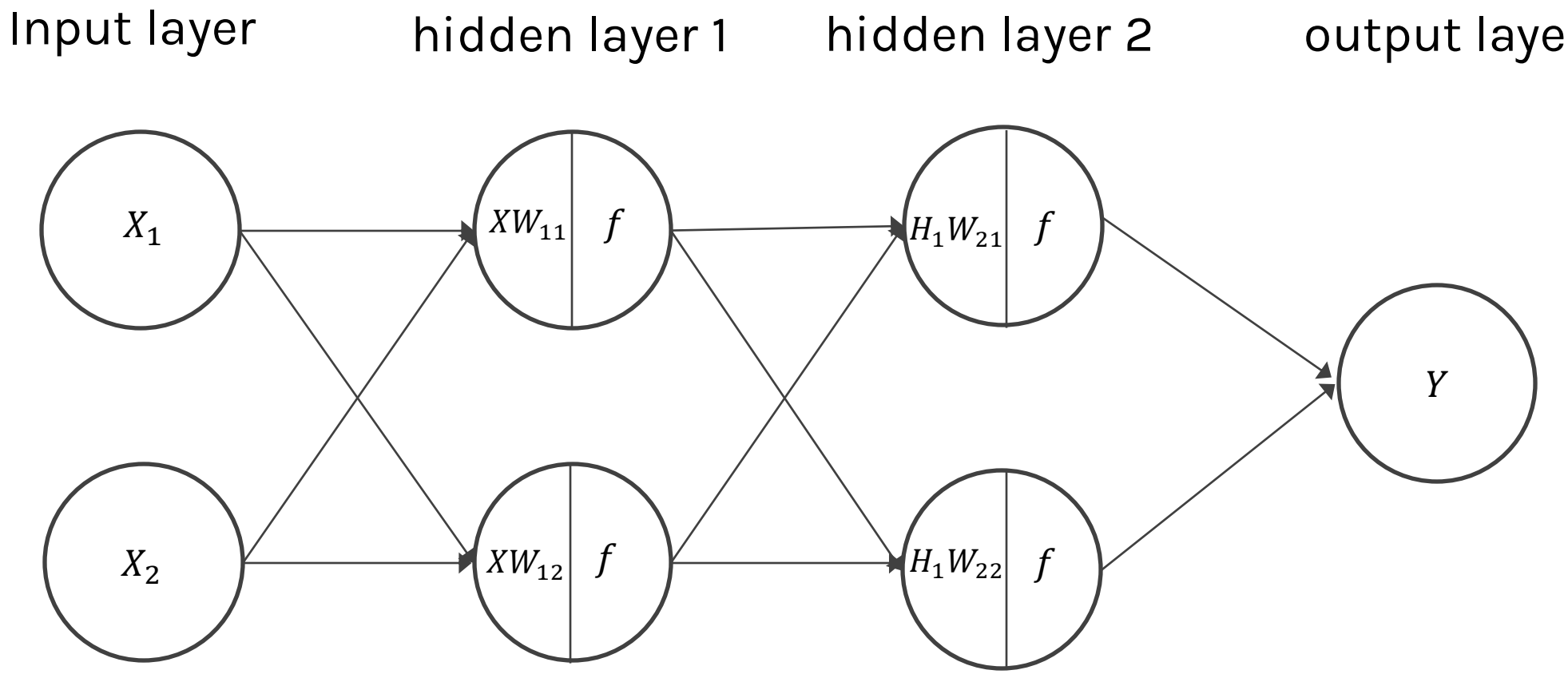
We will talk later about the choice of activation function.

Anatomy of artificial neural network (ANN)

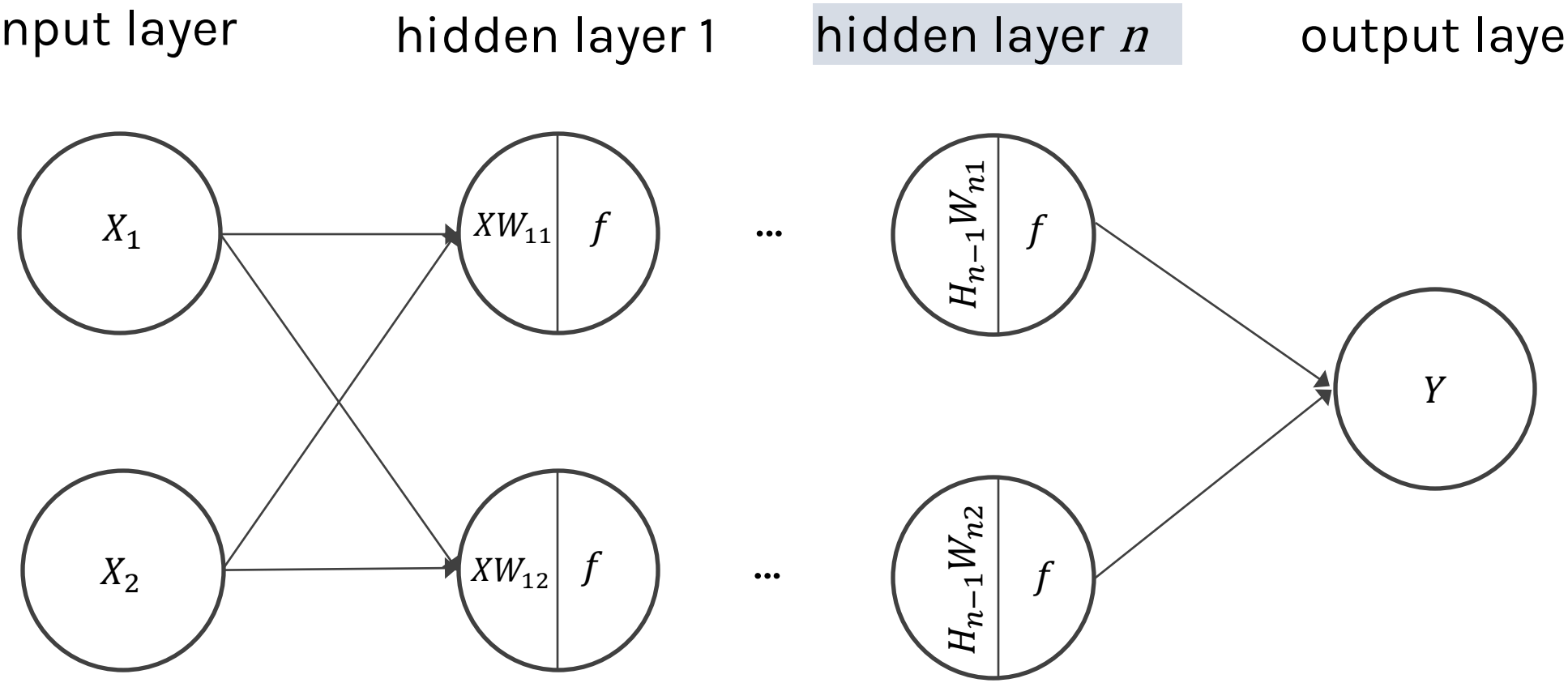


We will talk later about the choice of the output layer and the loss function.

Anatomy of artificial neural network (ANN)



Anatomy of artificial neural network (ANN)



Anatomy of artificial neural network (ANN)

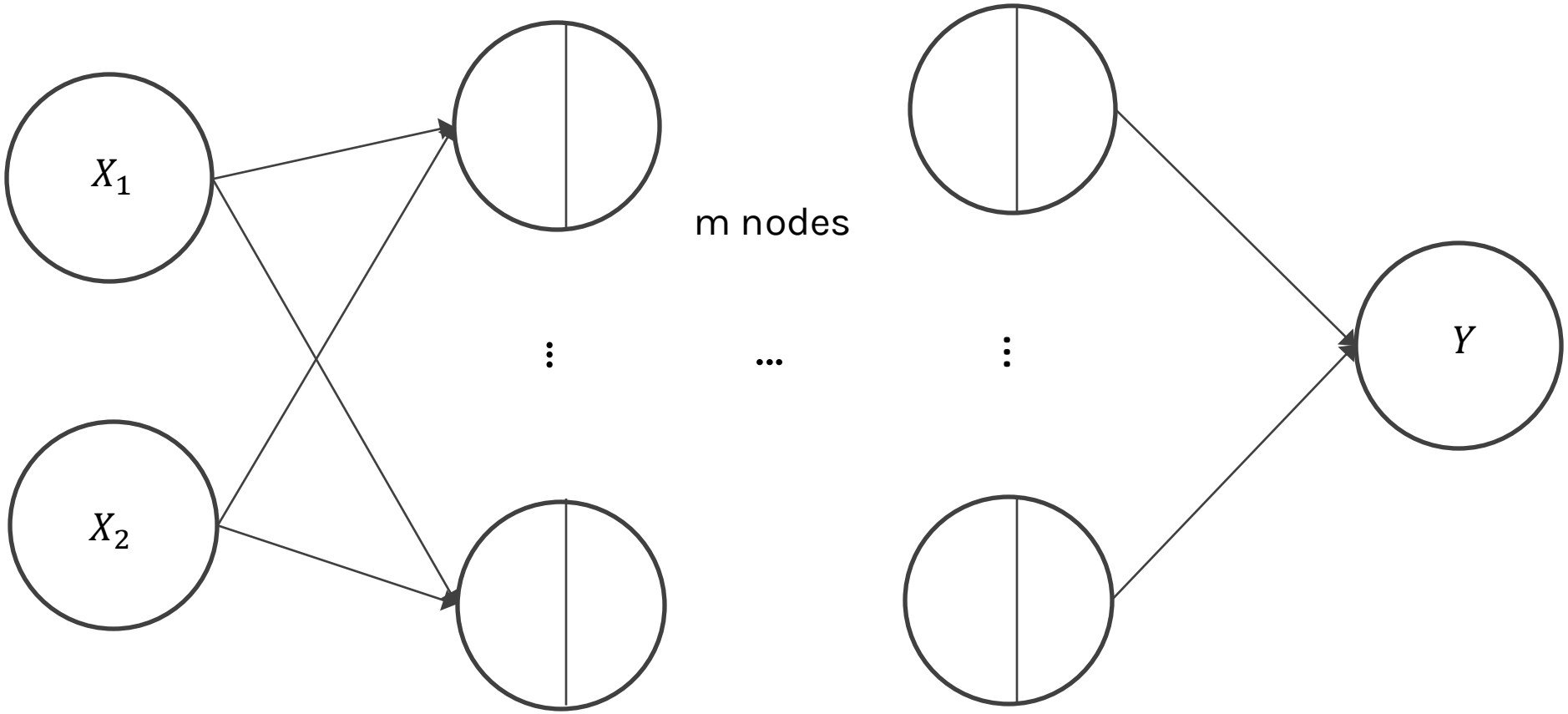
Input layer

hidden layer 1,

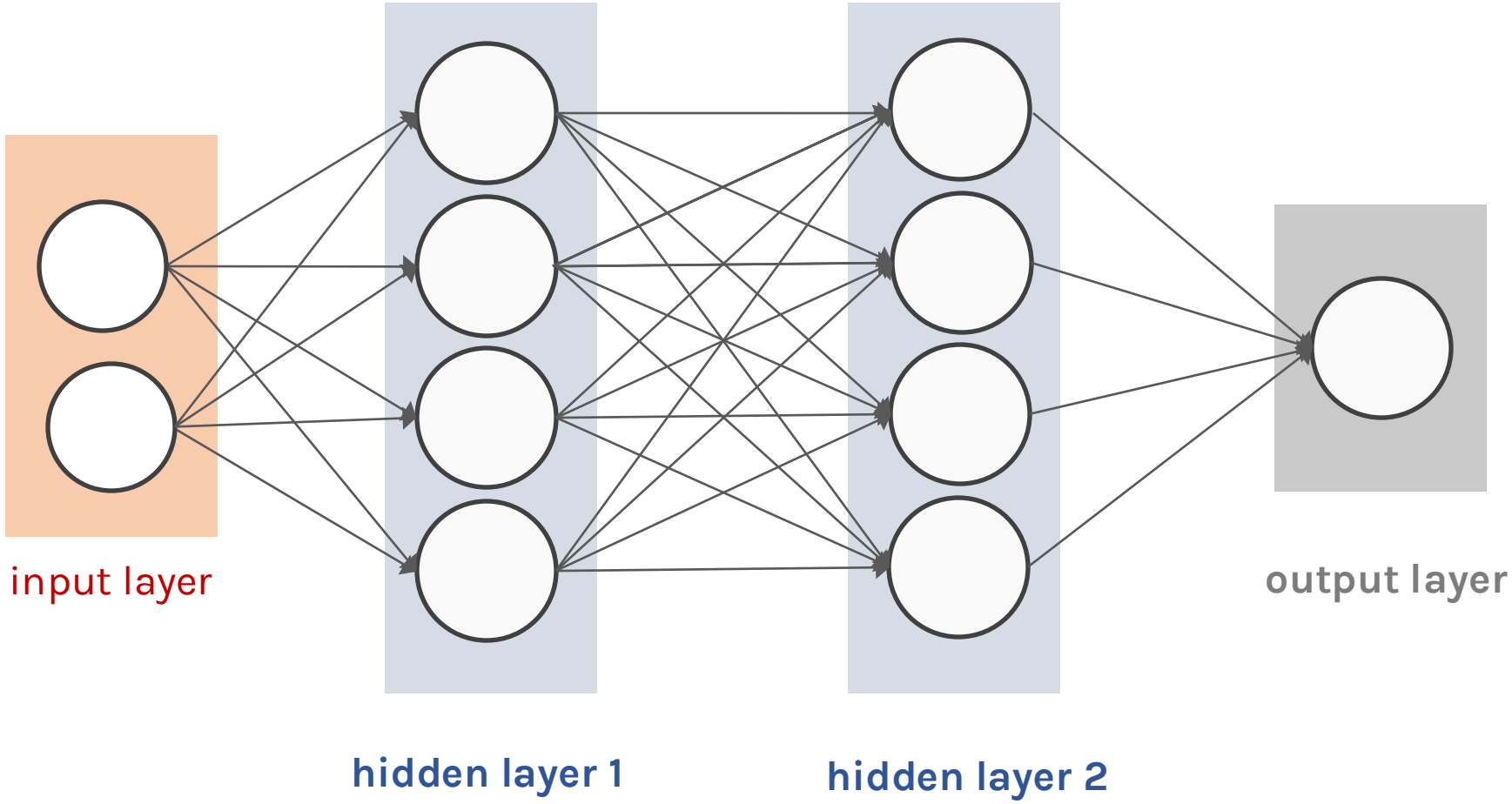
m nodes

hidden layer n

output layer



Anatomy of artificial neural network (ANN)



Design choices: Activation function

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

The activation function should:

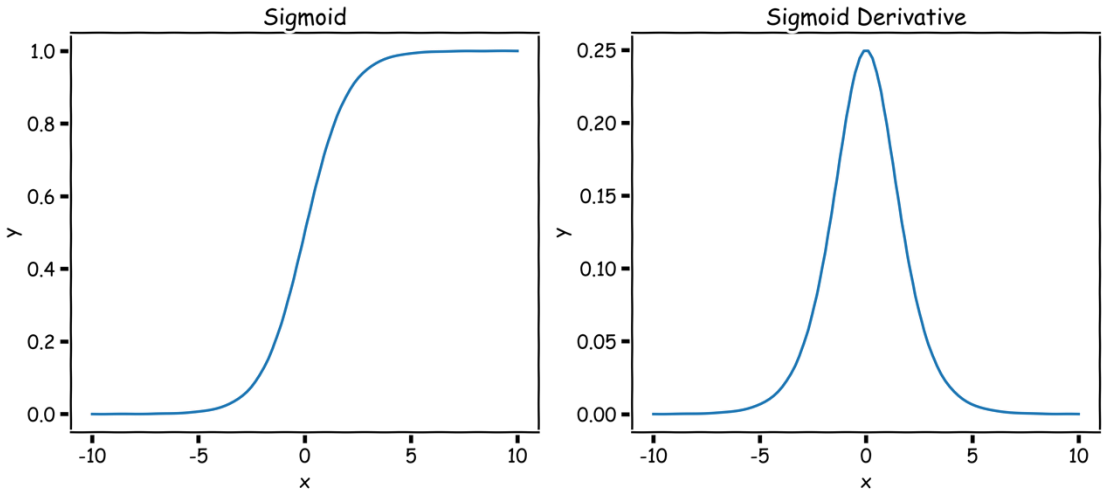
- Provide non-linearity
- Ensure gradients remain large through hidden unit

Common choices are

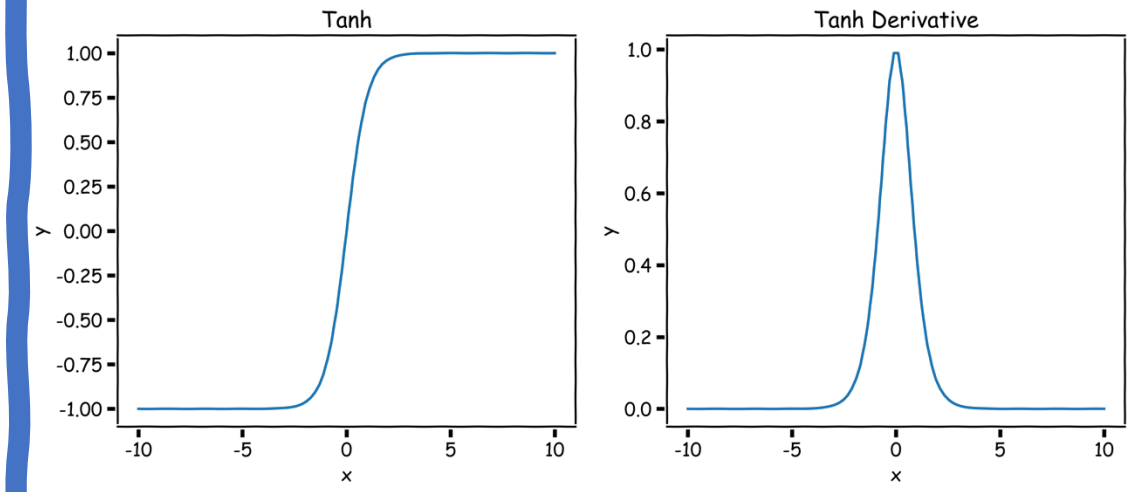
- sigmoid, tanh
- ReLU, leaky ReLU, Generalized ReLU
- softplus
- swish

Sigmoid, $\sigma()$ (aka logistic) and tanh

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



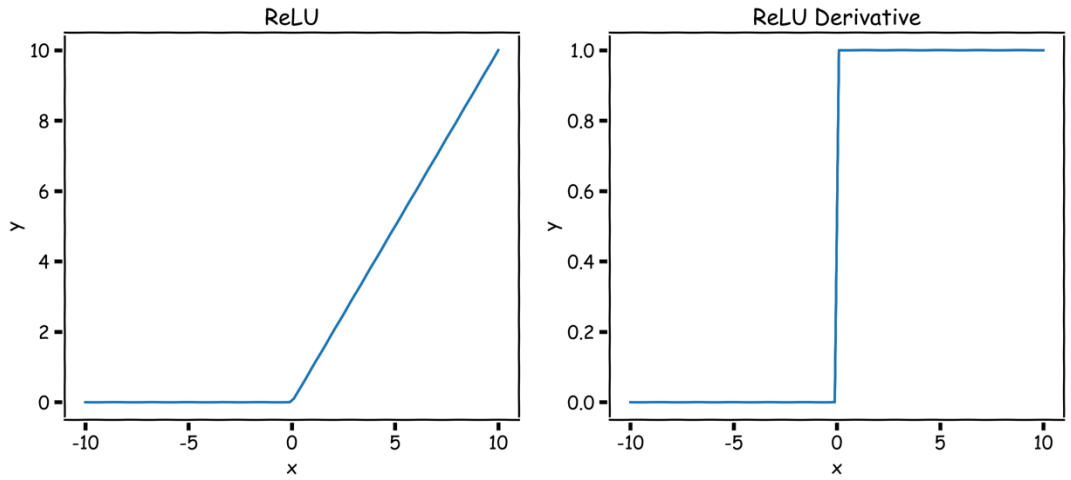
$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



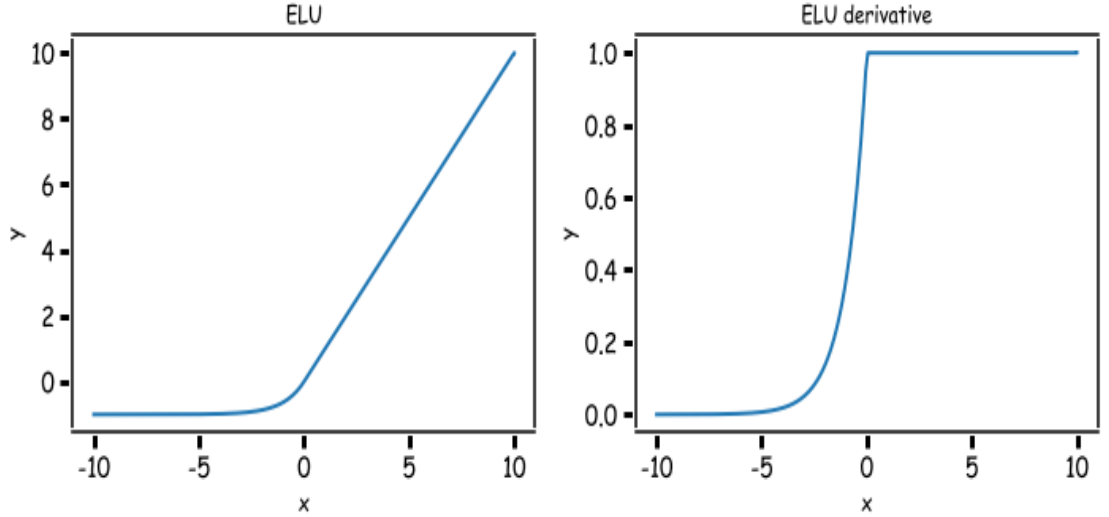
Derivative is **zero** for much of the domain. This leads to "vanishing gradients" in backpropagation.

Rectified Linear Unit, ReLU(), Exponential ReLU (ELU)

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



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



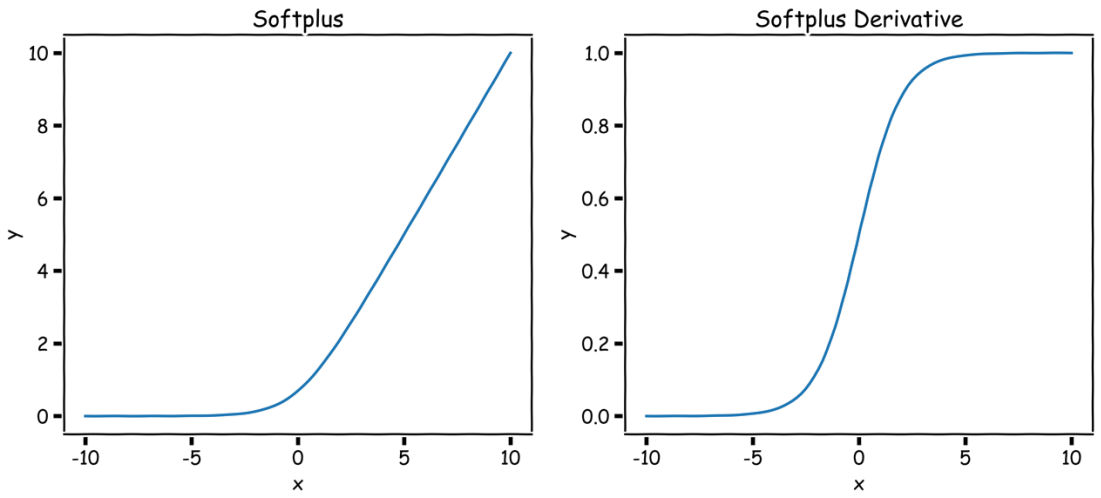
Two major advantages:

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

No vanishing gradients and easy to calculate.

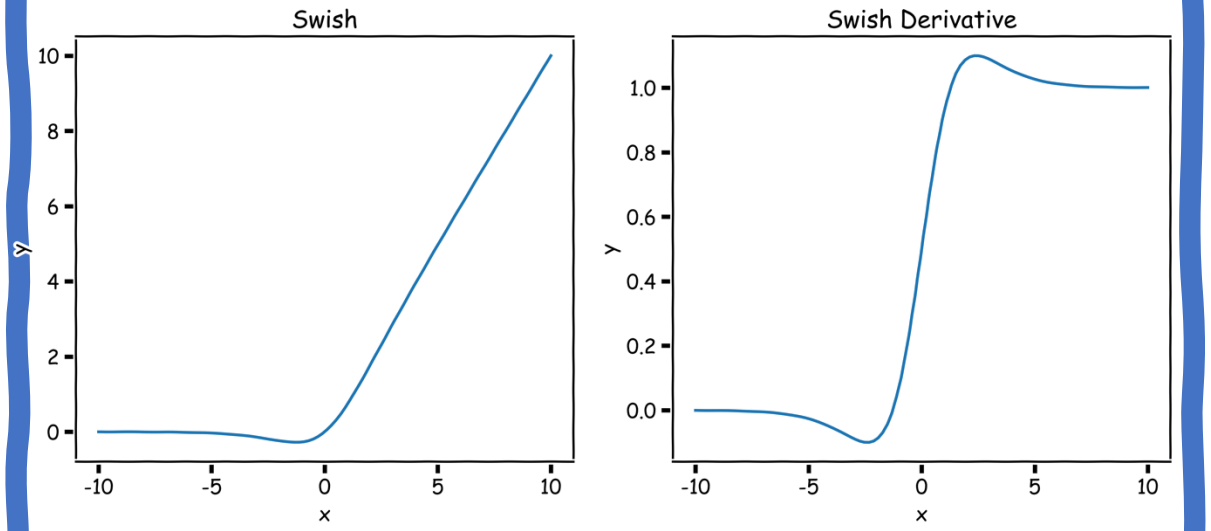
Softplus and Swish

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



The derivative of the softplus is the sigmoid logistic function, which is a smooth approximation of the derivative of the rectifier. So the derivative of the softplus is continuous.

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



Swish tends to work better than ReLU on deeper models across a number of challenging datasets.

TL;DR (too long; didn’t read)

Use ReLU or leaky ReLU or ELU to start.

If you are concerned for a few points improvement in performance, experiment with swish and softplus and treat them as hyperparameters. In other words choose the activation that gives you the best validation loss.

Loss Function

Probabilistic modeling

Likelihood for a given measurement:

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

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 minimizing the –ve log-likelihood:

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

Loss Function

Do not need to design separate loss functions if we follow the probabilistic modeling approach, i.e. minimize the -ve likelihood function.

Examples:

- Distribution is **Normal** then -ve log-likelihood is **MSE** :

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

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

- Distribution is **Bernouli** then -ve log-likelihood is **Binary Cross-Entropy**:

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

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

Loss Function

- For y that follow **Multinouli,** then likelihood is:

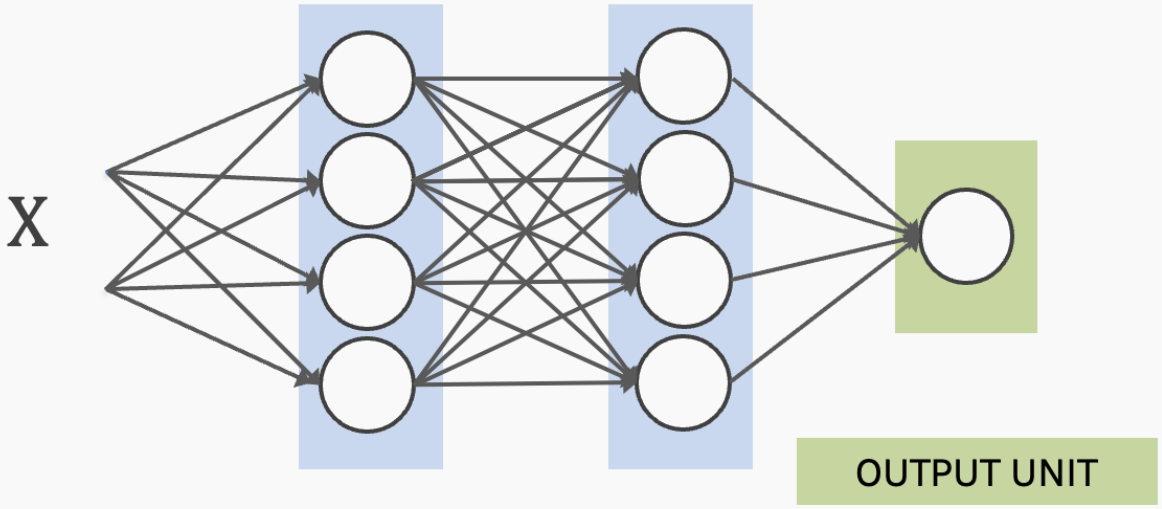
$$p(y_i|W;x_i) = \prod_k p\,(y_i = k)^{\mathbb{I}(y_i=k)}$$

Cross-Entropy

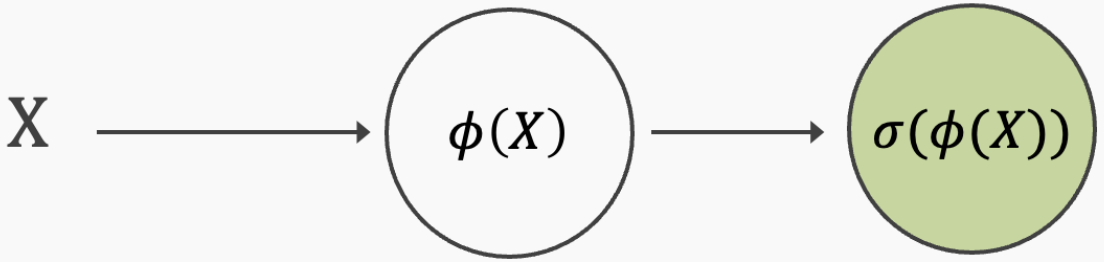
$$\mathcal{L}(W;X,Y) = -\sum_i \sum_k \mathbb{I}(y_i = k) \log p(y_i = k)$$

where k is the class and $\mathbb{I}(y_i = k)$ is the indicator function.

Output unit for binary classification



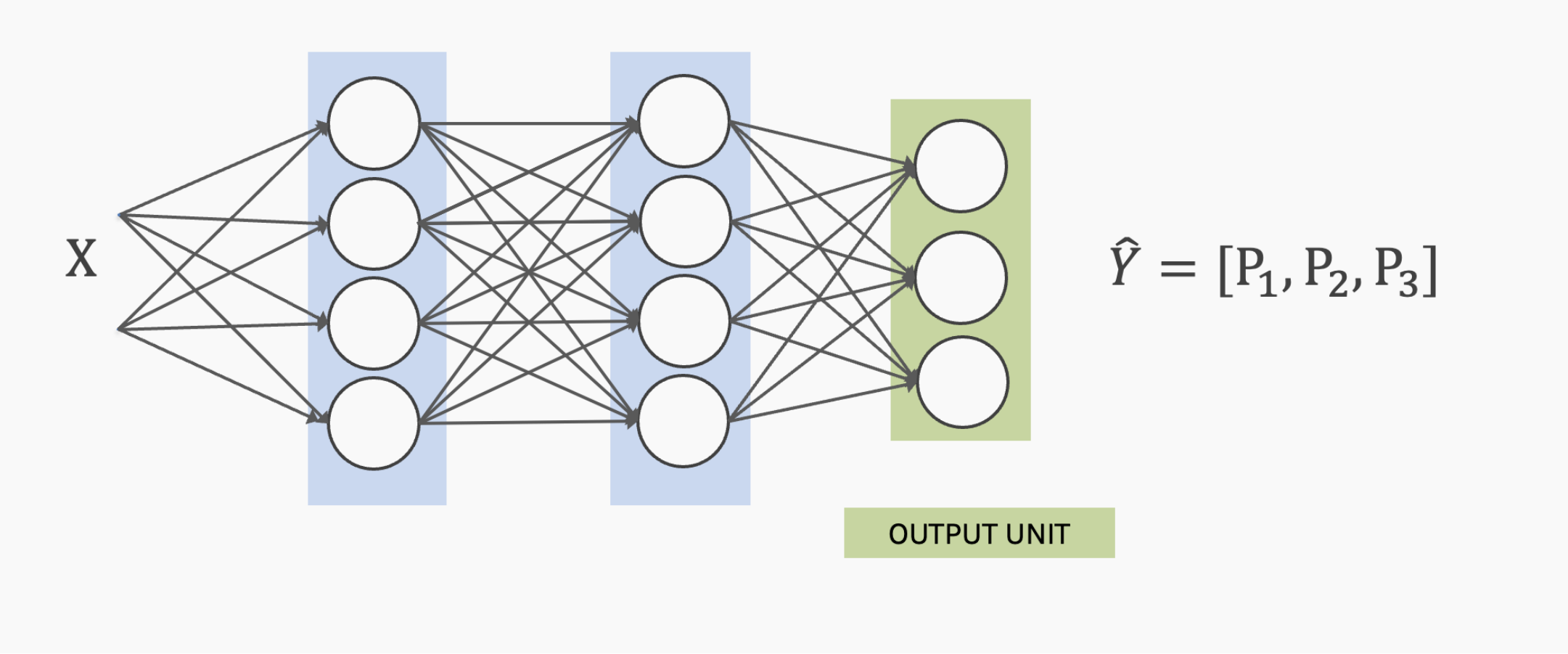
$\hat{Y} = \mathrm{P}(\mathrm{y} = 1)$



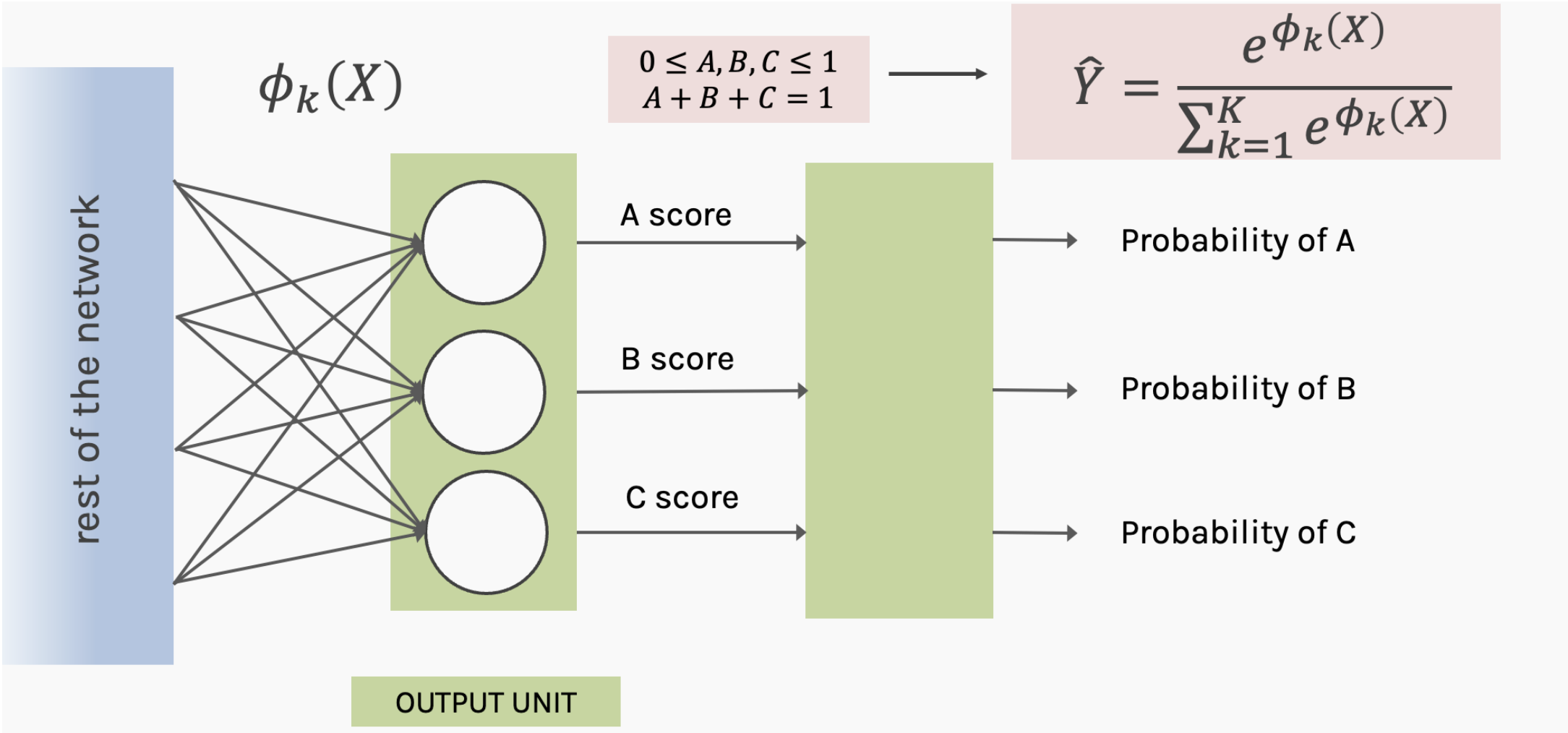
$\hat{Y} = \mathrm{P}(\mathrm{y} = 1)$

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

Output unit for multi-class classification



SoftMax


$\phi_k(X)$
$0 \le A, B, C \le 1$
$A + B + C = 1$
$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^{K} e^{\phi_k(X)}}$
rest of the network
A score
B score
C score
Probability of A
Probability of B
Probability of C
OUTPUT UNIT

SoftMax and Cross-Entropy Loss

Cross-Entropy (Softmax)

Softmax $L_i = -\log(\frac{e^{s_{y_i}}}{\sum_i e^{s_j}})$

Score function $s = f(x_i, W)$

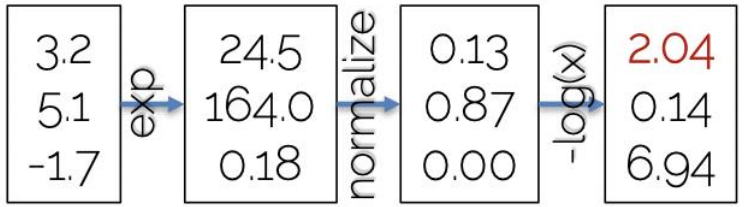
e.g., $f(x_i, W) = W \cdot [x_0, x_1, \dots, x_N]^T$

Suppose: 3 training examples and 3 classes



scores				
	cat	3.2	1.3	2.2
	chair	5.1	4.9	2.5
	"car"	-1.7	2.0	-3.1
Loss		2.04	0.079	6.156

Given a function with weights W,
Training pairs $[x_i; y_i]$ (input and labels)



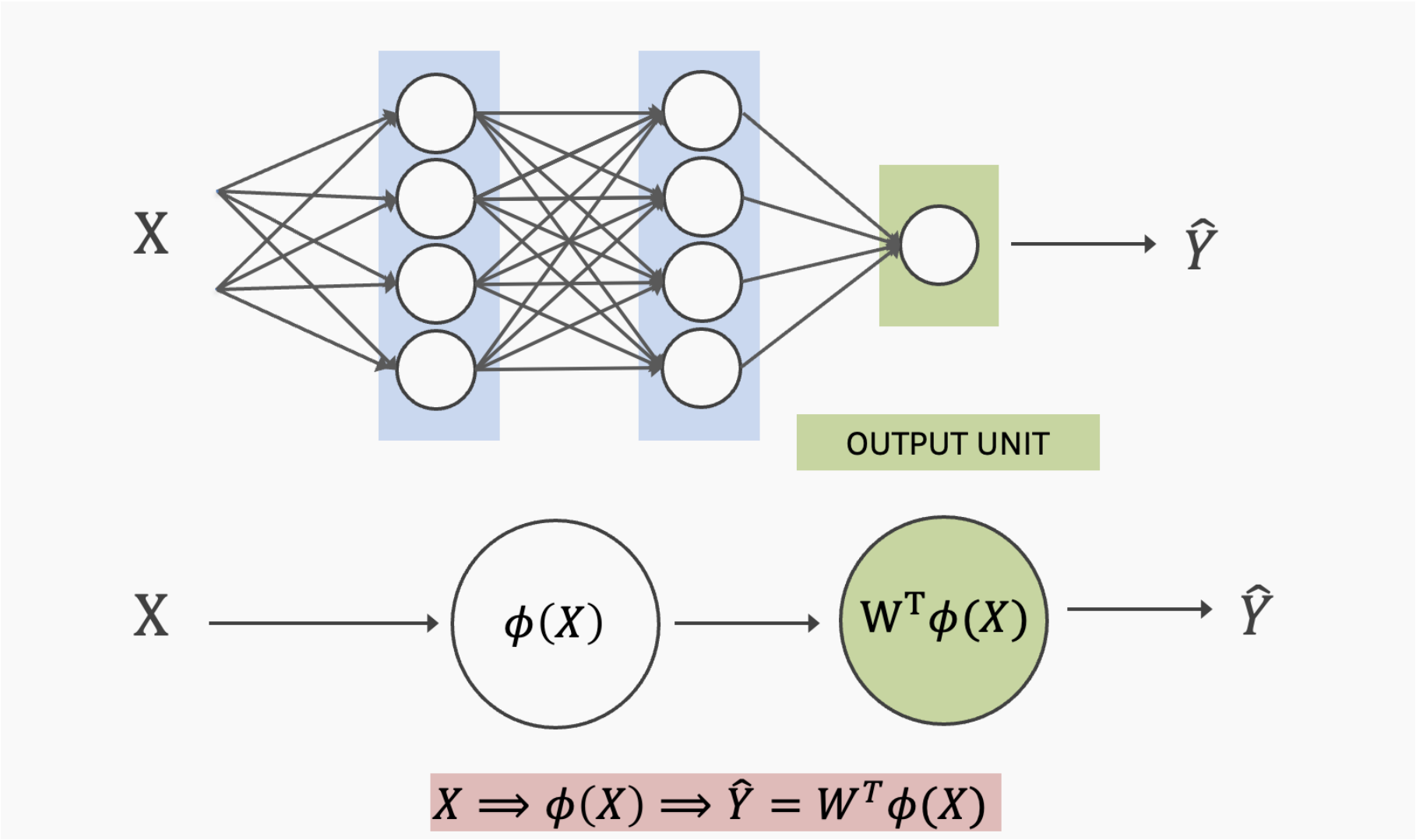
$$L = \frac{1}{N}\sum_{i=1}^{N} L_i =$$

$$= \frac{L_1 + L_2 + L_3}{3} =$$

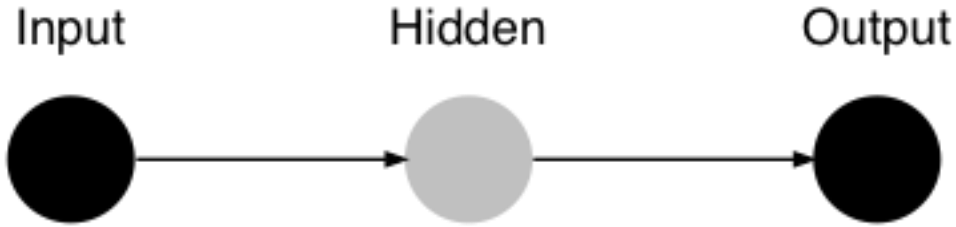
$$= \frac{2.04 + 0.079 + 6.156}{3} =$$

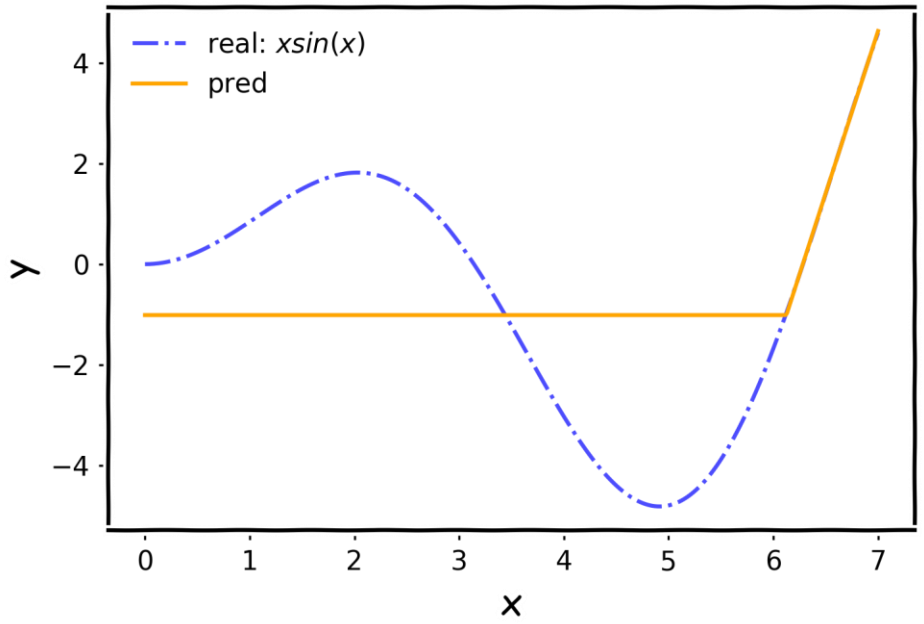
$$= \mathbf{2.76}$$

Output unit for regression

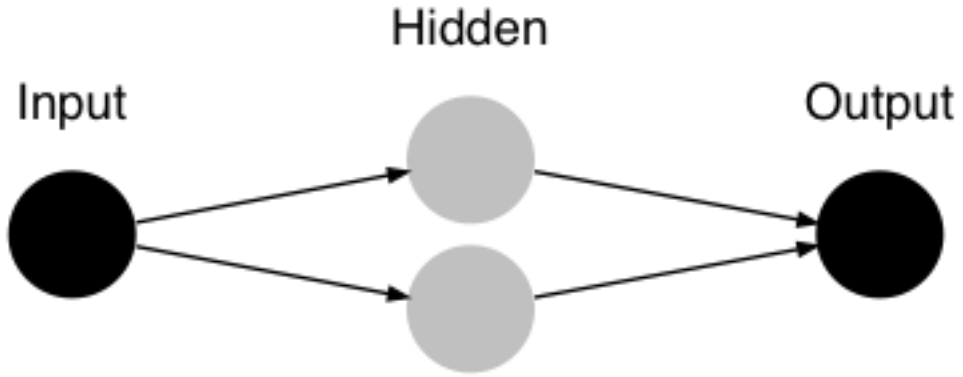


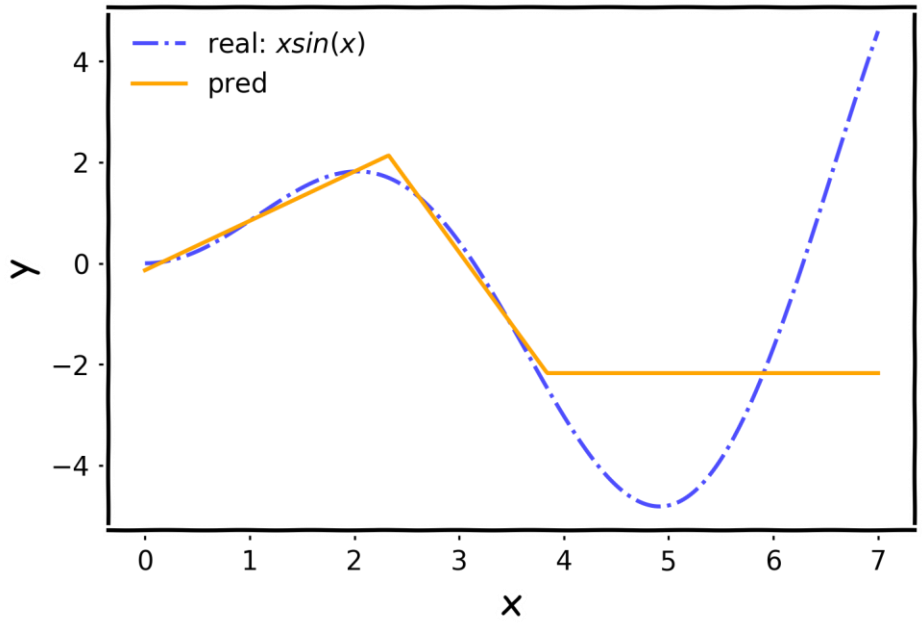
Number of nodes



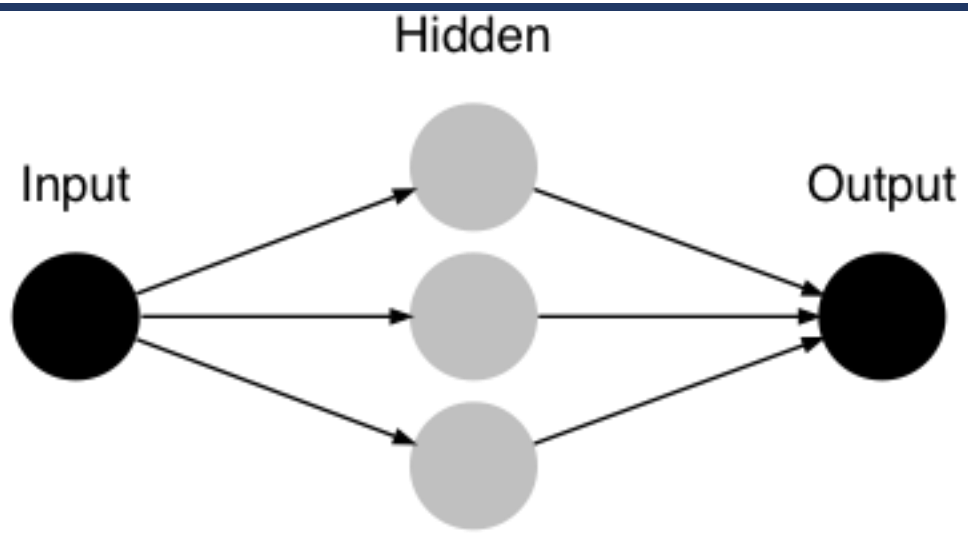


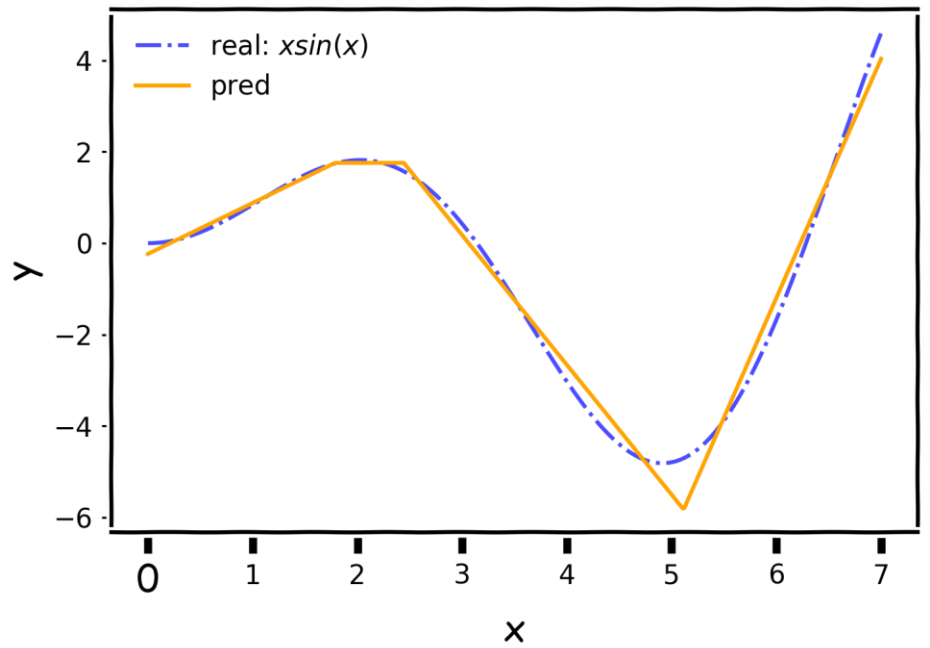
Number of nodes



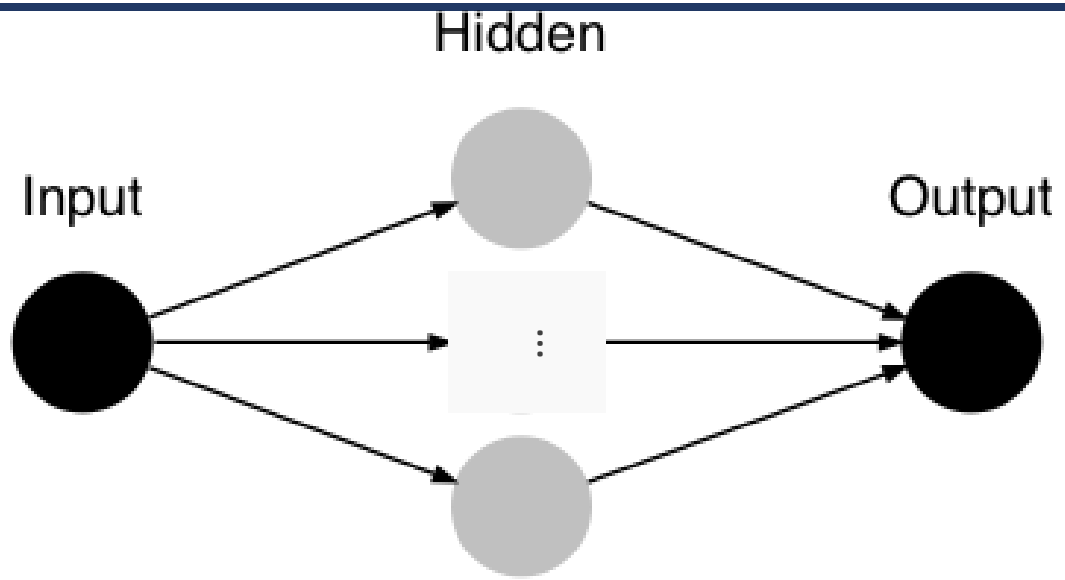


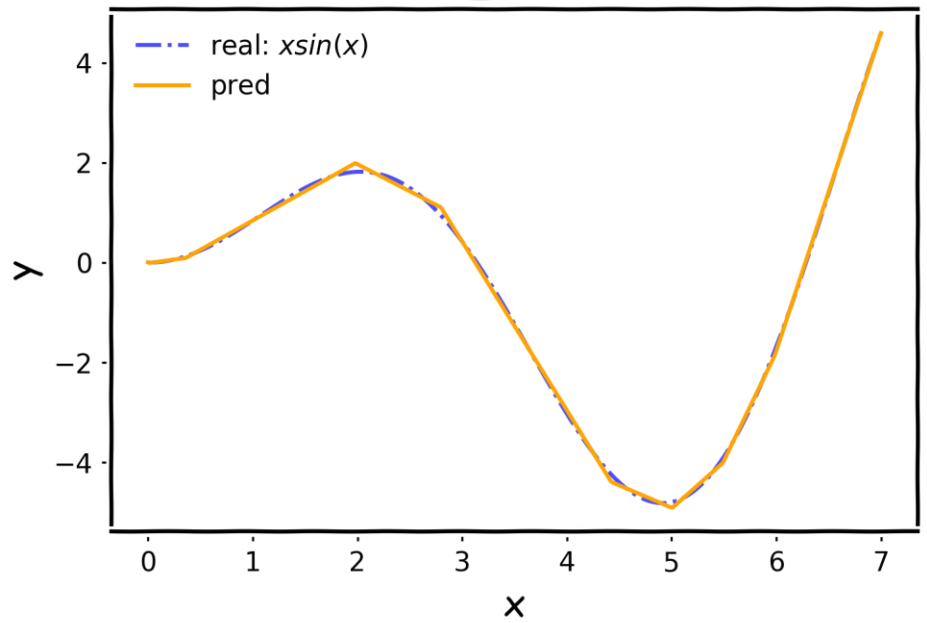
Number of nodes



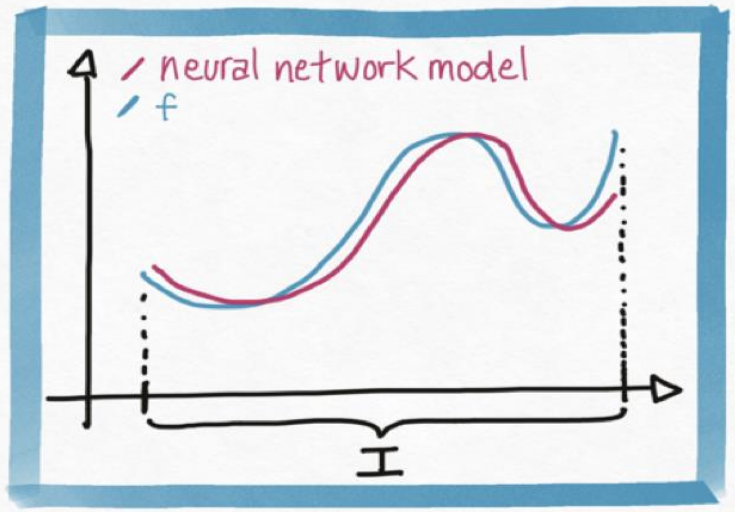


Number of nodes





Neural Networks as Universal Approximators



We have seen that neural networks can represent complex functions, but are there limitations on what a neural network can express?

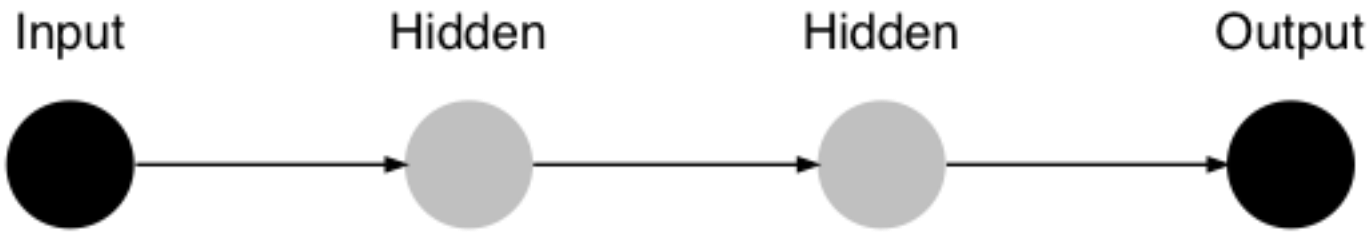
Theorem:

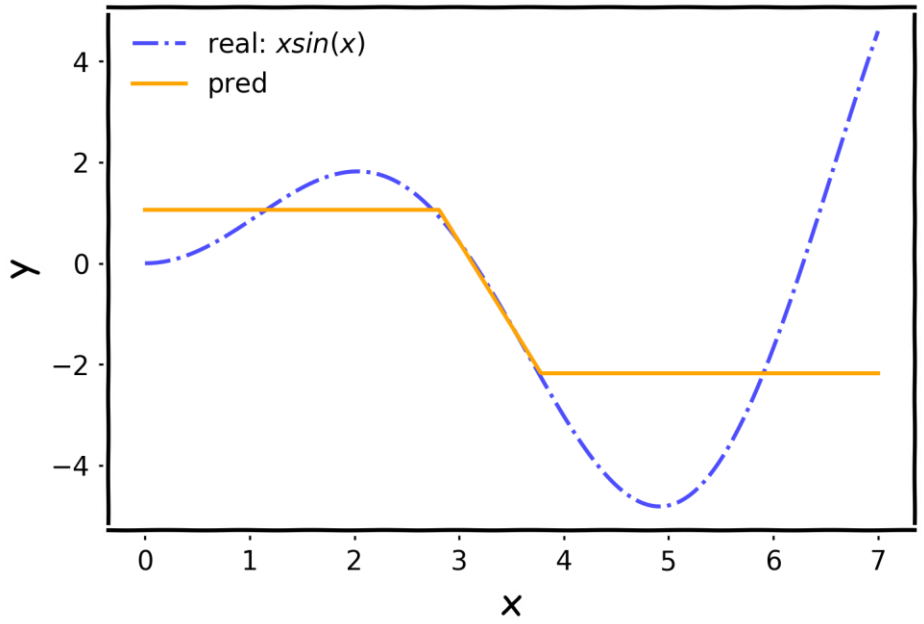
For any continuous function f defined on a bounded domain, we can find a neural network that approximates f with an arbitrary degree of accuracy.

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

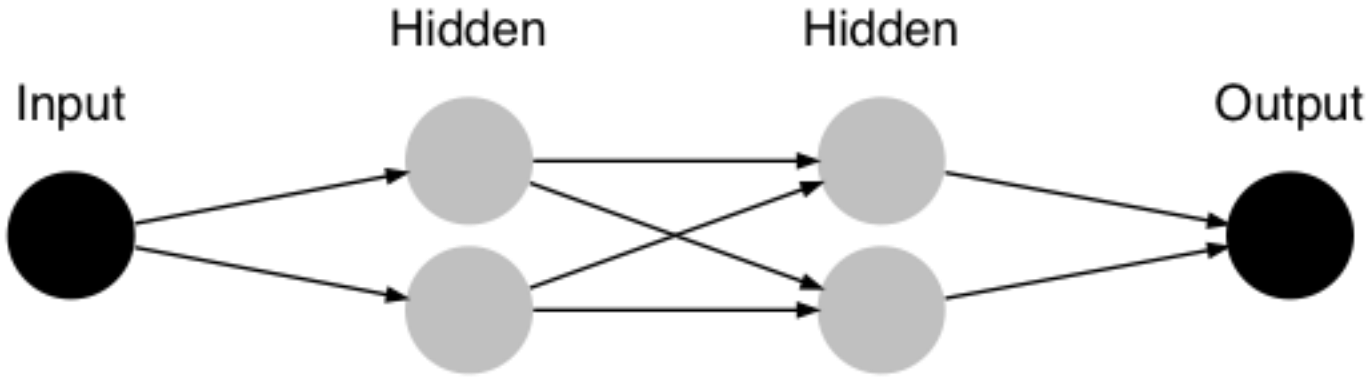
So why deeper?

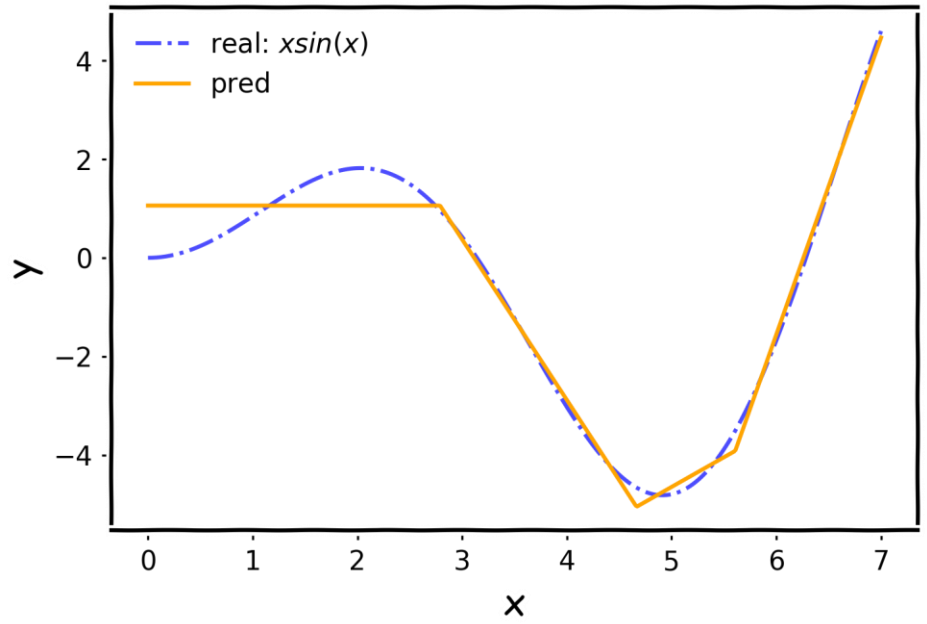
Layers


Input
Hidden
Hidden
Output

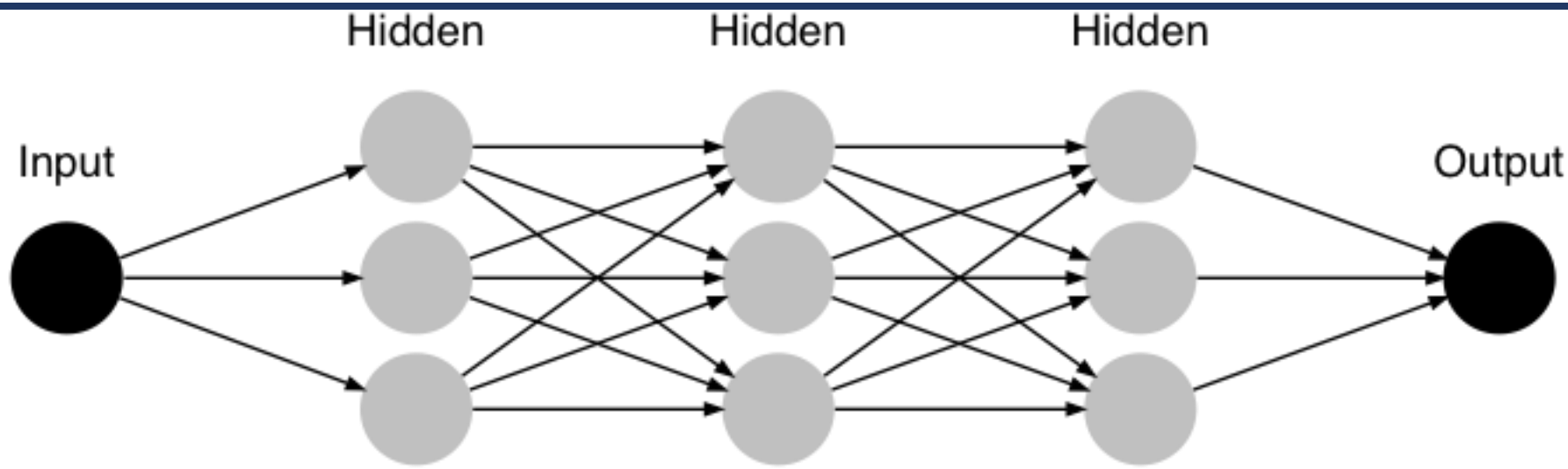

real: xsin(x)
pred
4
2
0
−2
−4
y
0
1
2
3
4
5
6
7
x

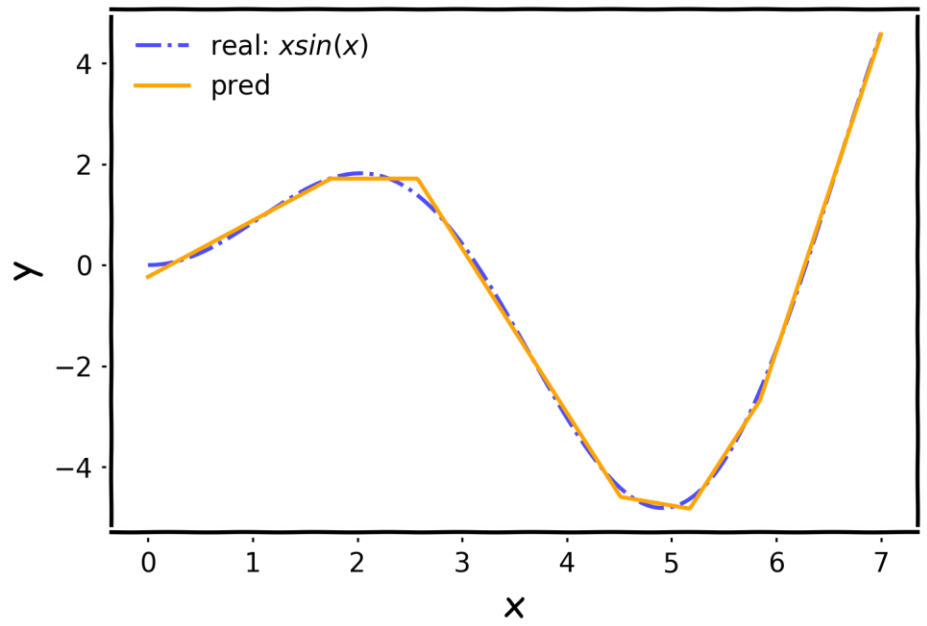
Layers


Input
Hidden
Hidden
Output


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

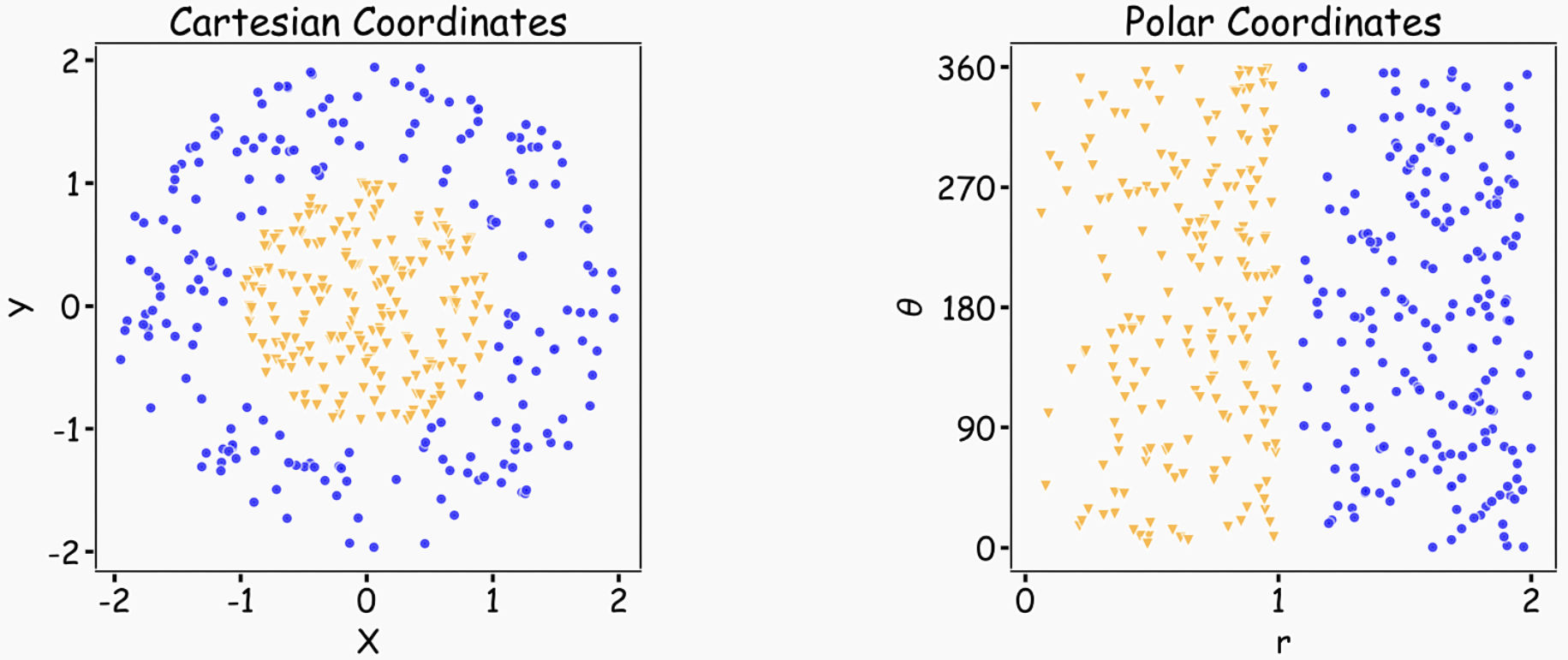
Layers


Hidden
Hidden
Hidden
Input
Output

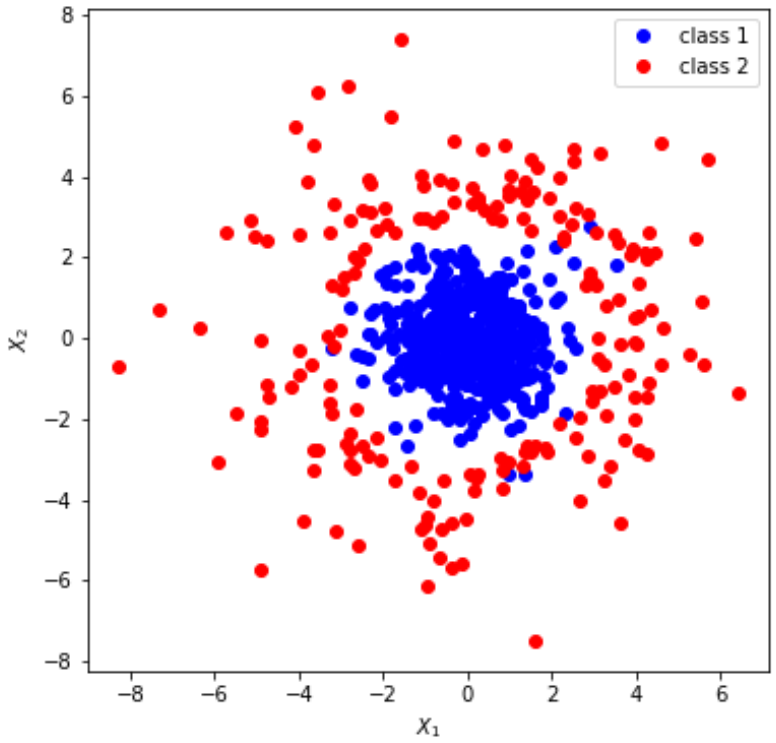

real: xsin(x)
pred
4
2
0
−2
−4
y
0
1
2
3
4
5
6
7
x

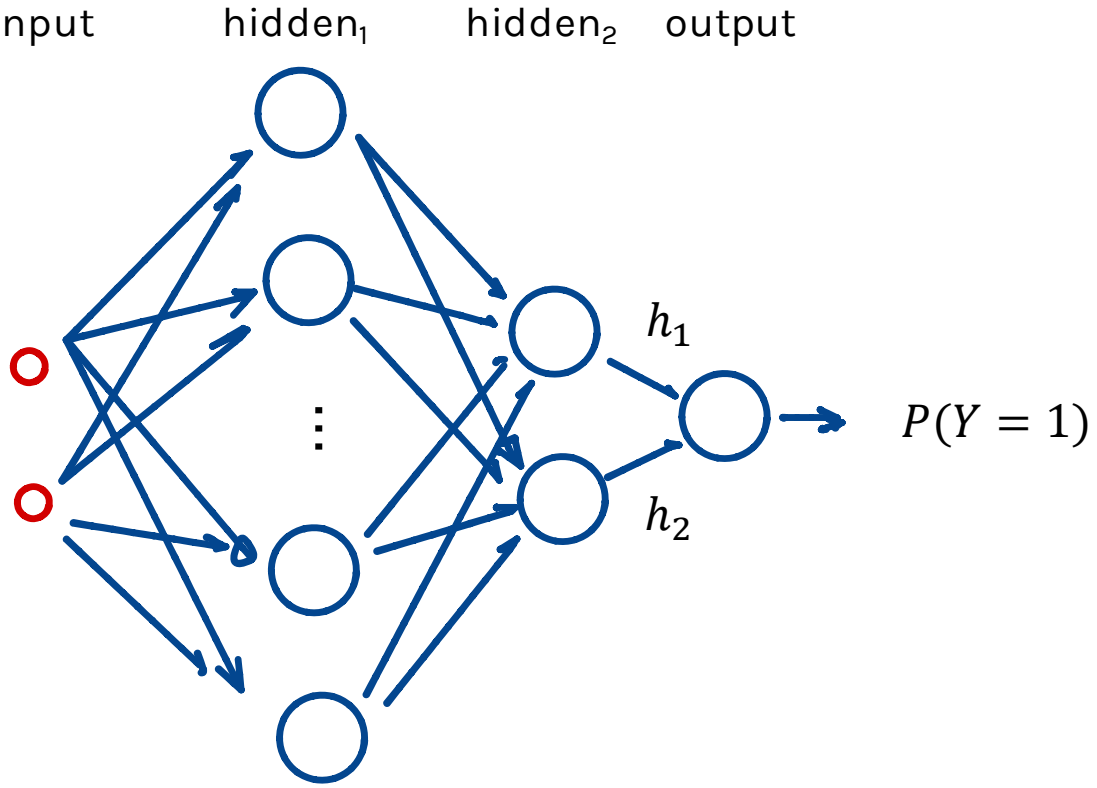
Why layers?

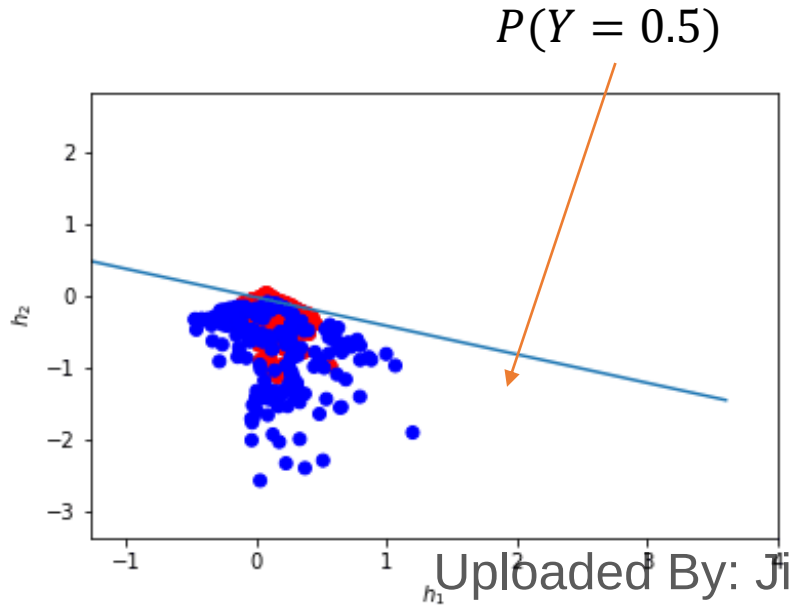
Representation matters!



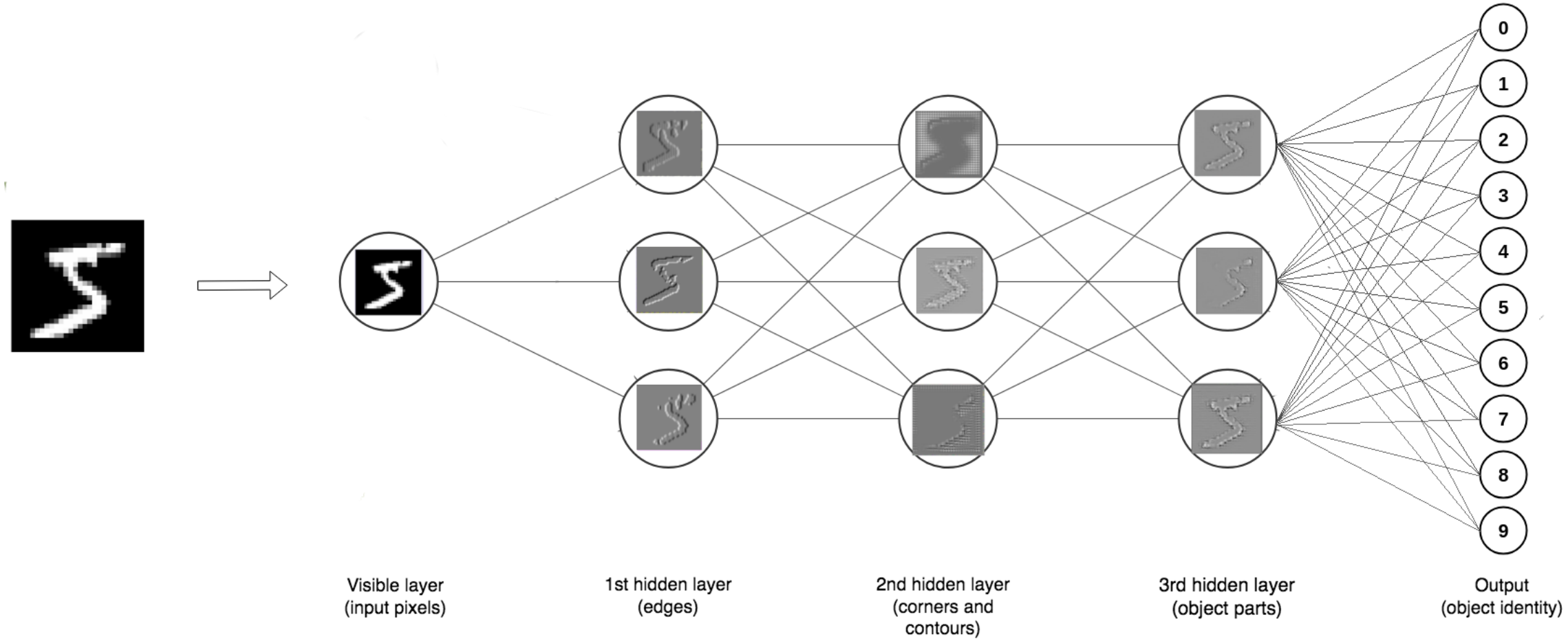
Neural networks can **learn useful representations** for the problem. This is another reason why they can be so powerful!



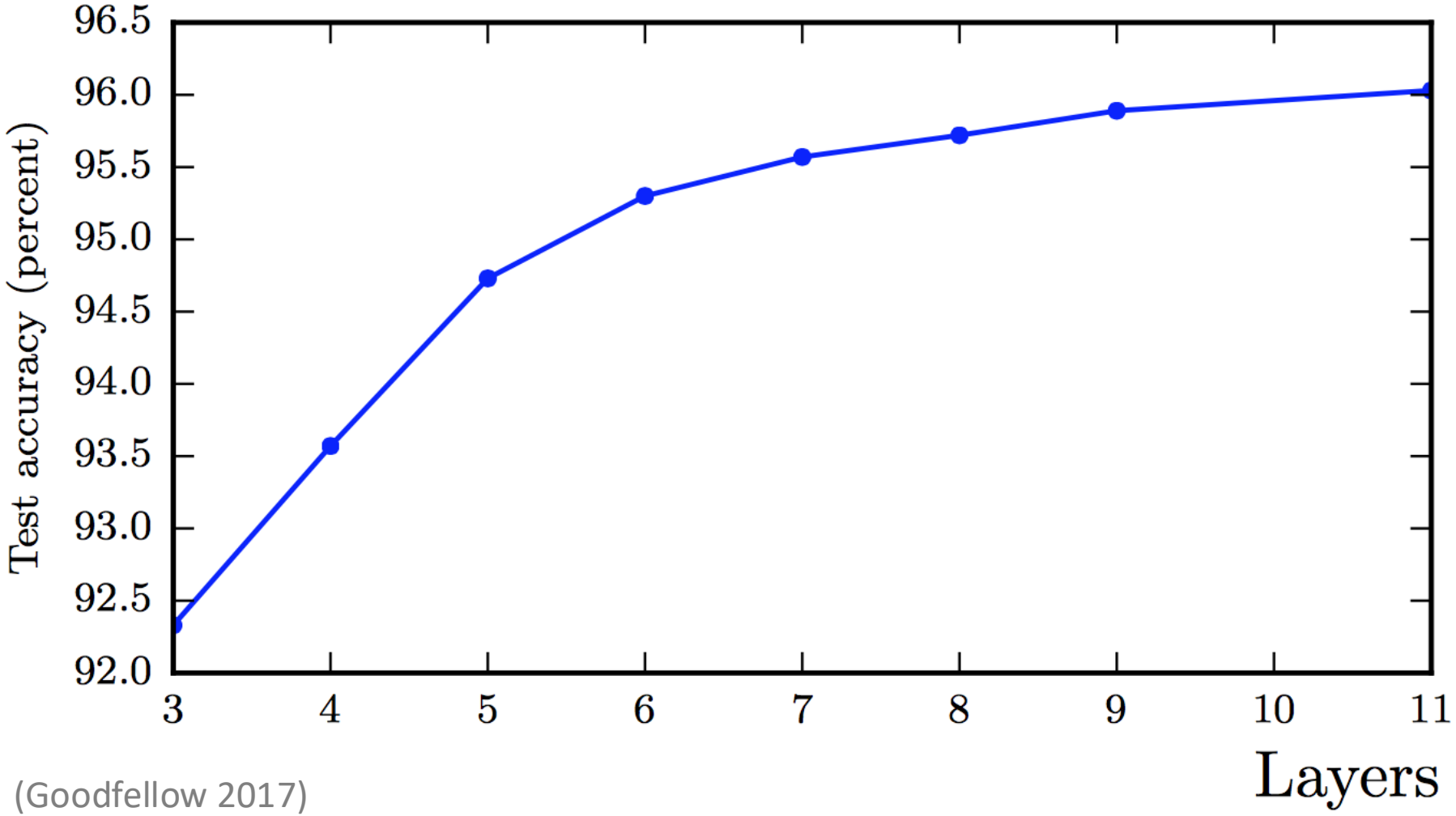




Depth = Repeated Compositions



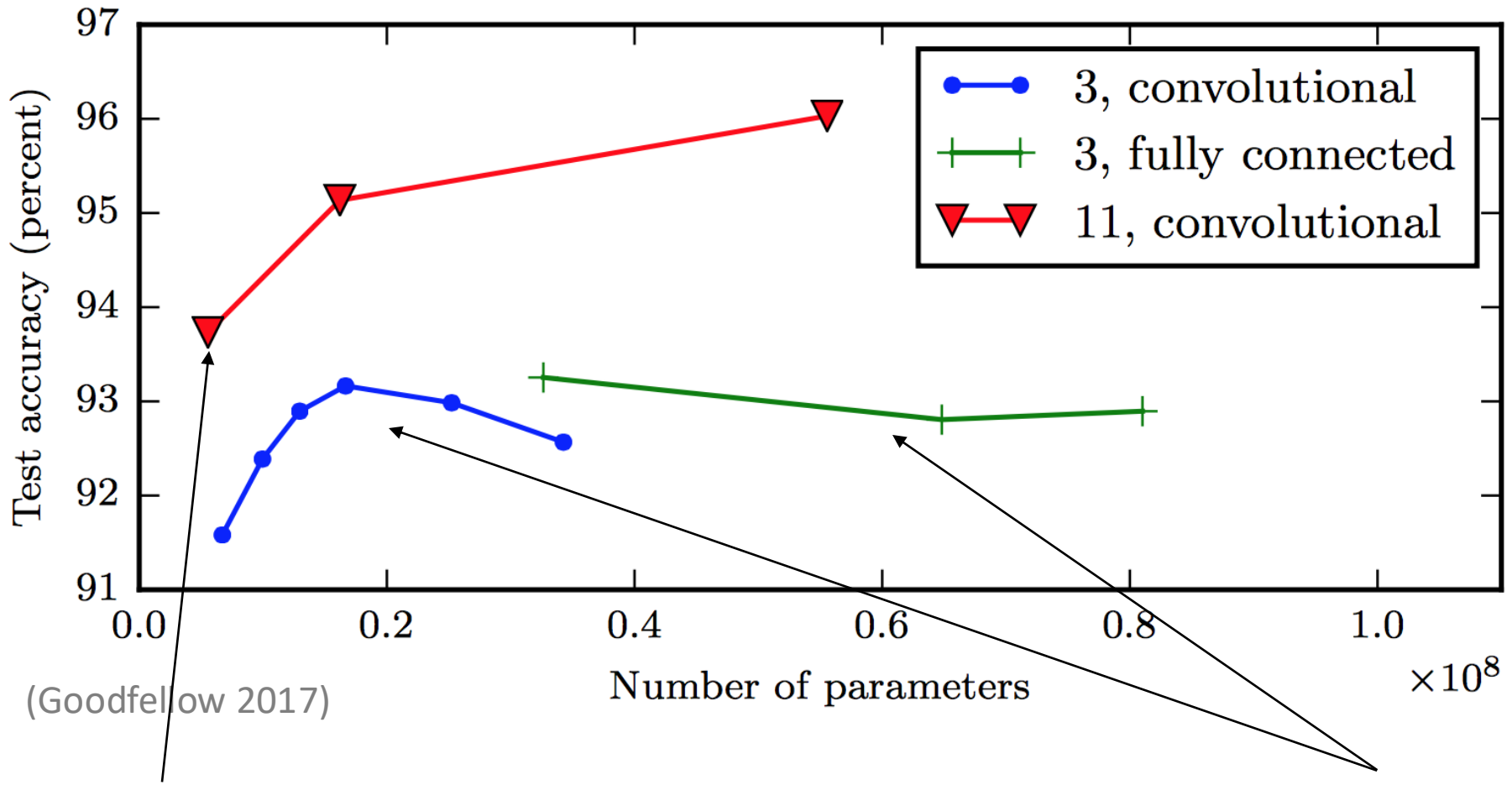
Better Generalization with Depth



(Goodfellow 2017)

Shallow Nets Overfit More

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



(Goodfellow 2017)

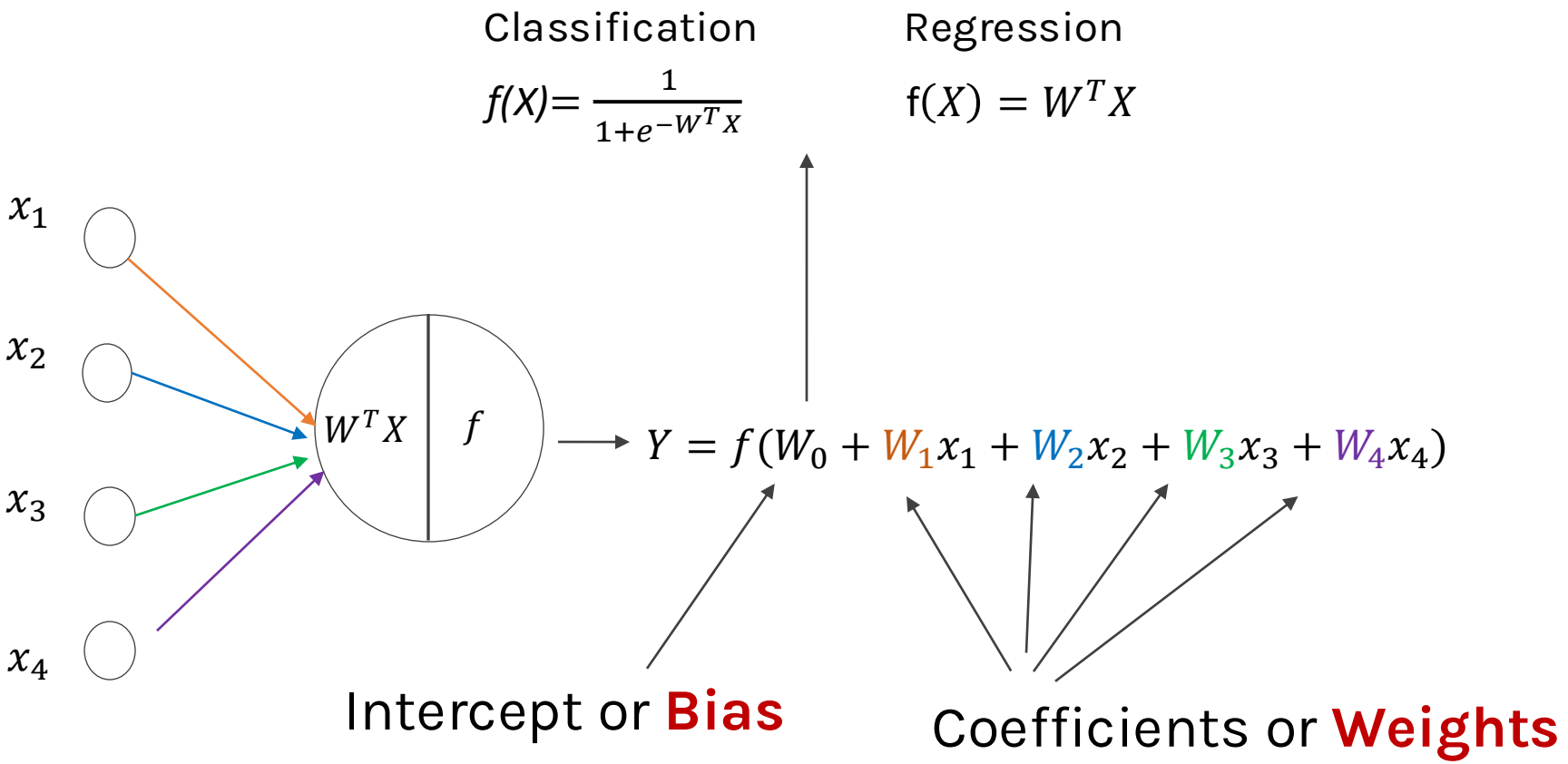
Don't worry about this word "convolutional". It's just a special type of neural network, often used for images.

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.

Learning the coefficients

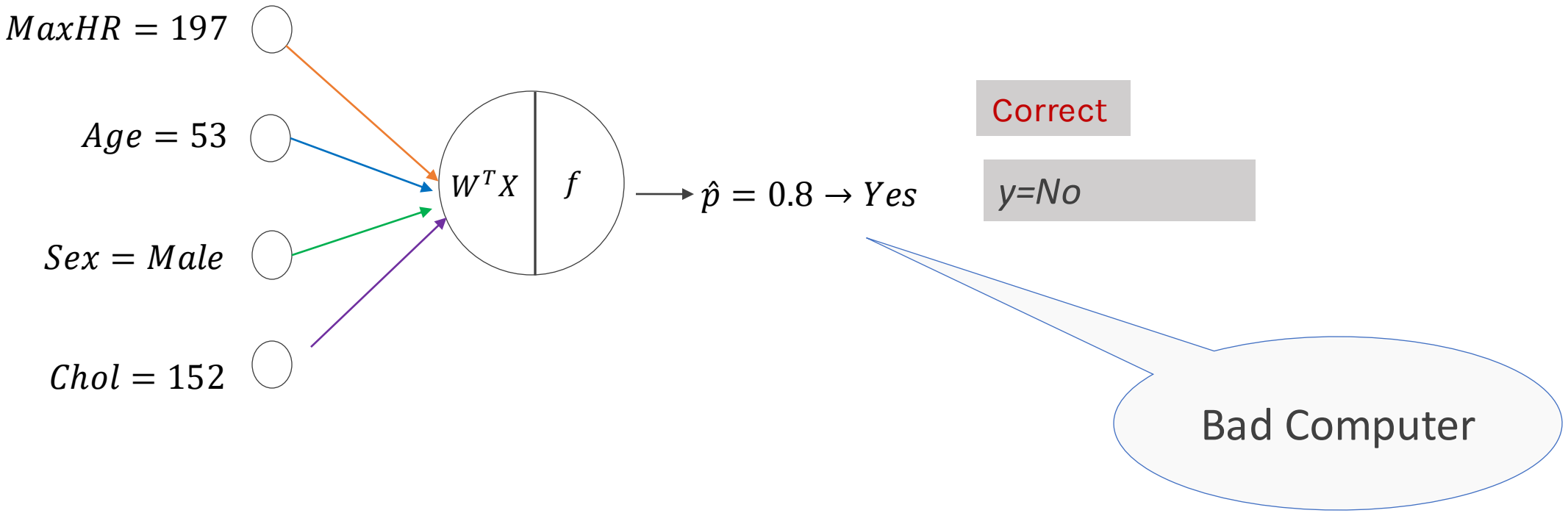
Start with single neuron



But what is the idea?

Start with all randomly selected weights. Most likely it will perform horribly.

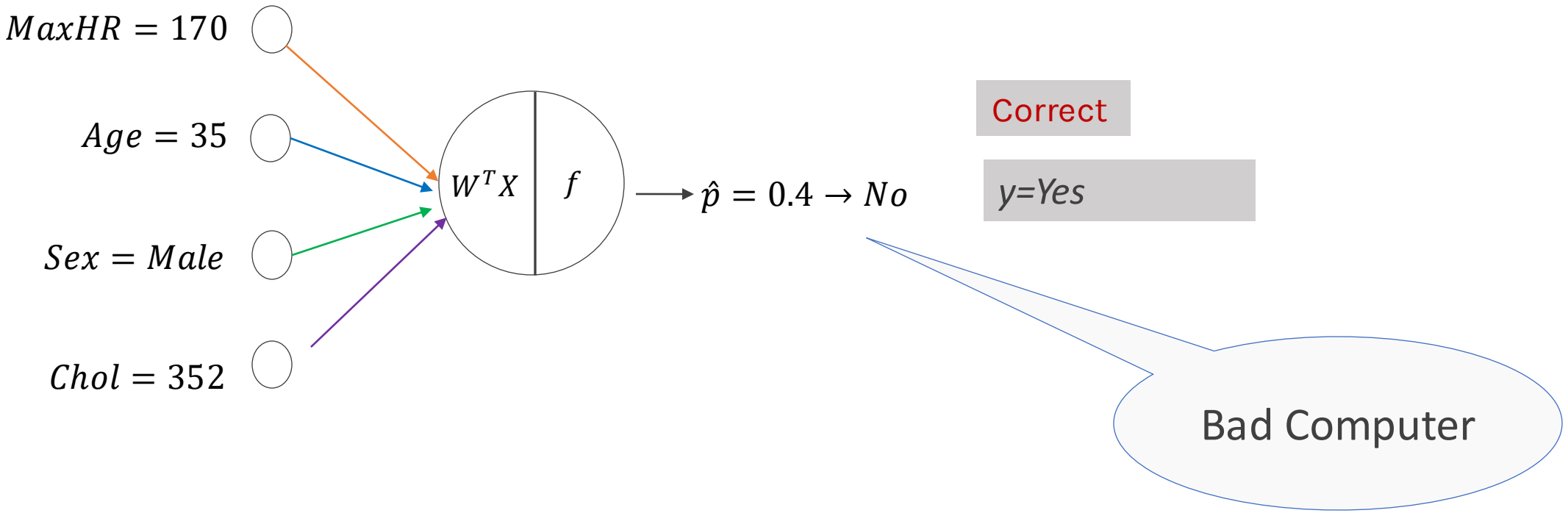
For example, in our heart data, the model will be giving us the wrong answer.



But what is the idea?

Start with all randomly selected weights. Most likely it will perform horribly.

For example, in our heart data, the model will be giving us the wrong answer.



But what is the idea?

- **Loss Function**: Takes all of these results and averages them and tells us how bad or good the computer or those weights are.

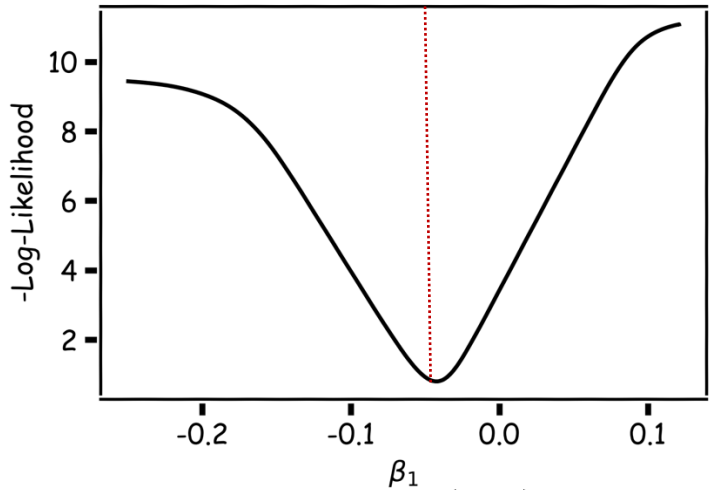
- Telling the computer how bad or good is, does not help.
- You want to tell it how to change those weights so it gets better.

Loss function: $\mathcal{L}(w_0, w_1, w_2, w_3, w_4)$

For now let's only consider a single weight, $\mathcal{L}(w_1)$

Minimizing the Loss function

Ideally we want to know the value of w_1 that gives the minimal $\mathcal{L}(W)$

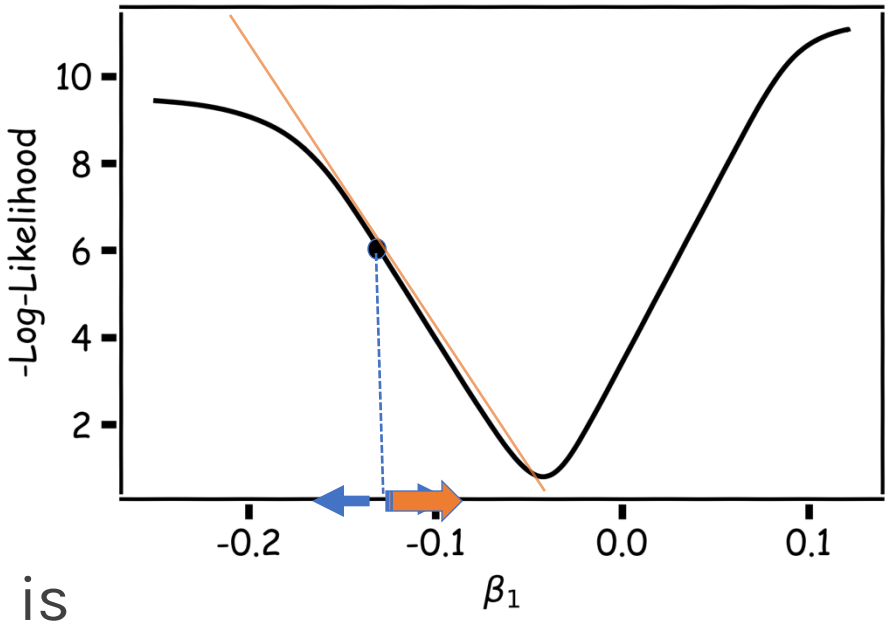


To find the optimal point of a function $\mathcal{L}(W)$

$$\frac{d\mathcal{L}(W)}{dW} = 0 \qquad \text{solve for } W^* = \text{argmin}_W \mathcal{L}(W)$$

And find the W that satisfies that equation. **Sometimes** there is no explicit solution for that.

Estimate of the regression coefficients: gradient descent



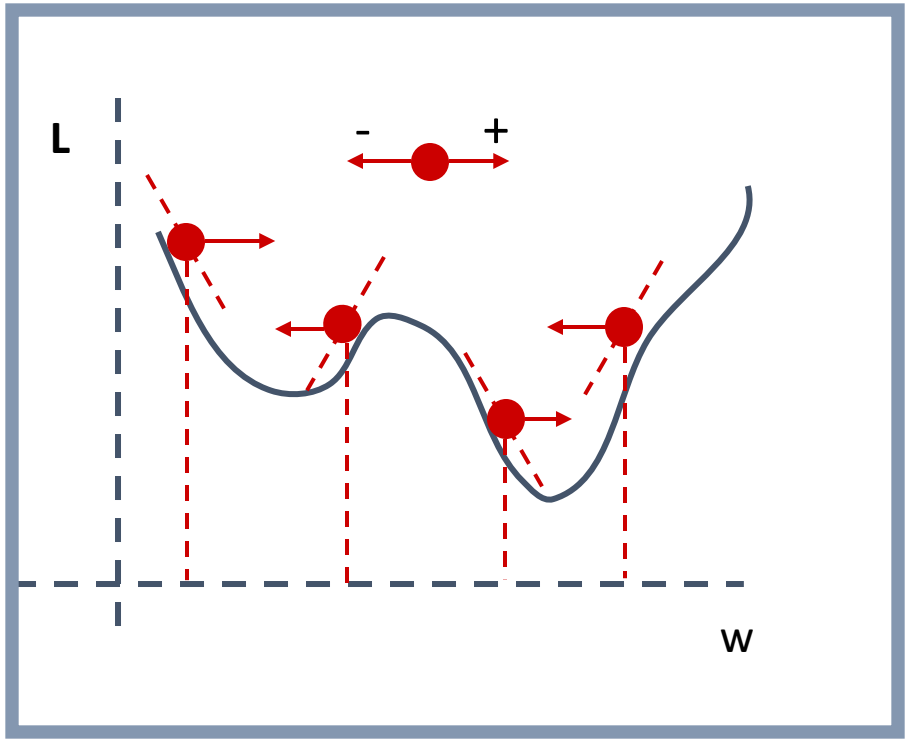
A more flexible method is

- Start from a random point
  1. Determine which direction to go to reduce the loss (left or right)
  2. Compute the slope of the function at this point and step to the right if slope is negative or step to the left if slope is positive
  3. Goto to #1

Gradient Descent

- Algorithm for optimization of first order to finding a minimum of a function.
- It is an iterative method.
- L is decreasing much faster in the direction of the negative derivative.
- The learning rate is controlled by the magnitude of η.

$$w^{(i+1)} = w^{(i)} - \eta \frac{d\mathcal{L}}{dw}$$



Batch and Stochastic Gradient Descent

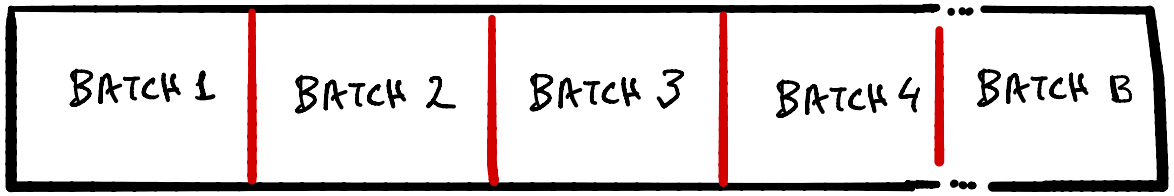
$$\mathcal{L} = -\sum_{i} [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

- Instead of using all the examples for every step, use a subset of them (batch).
- For each iteration *k*, use the following loss function to derive the derivatives:

$$\mathcal{L}^k = -\sum_{i \in b^k} [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

- which is an **approximation** to the full loss function.

DATA



calculate $L_1 \Rightarrow \frac{\partial L_1}{\partial w} \Rightarrow w^* = w - \eta \frac{\partial L_1}{\partial w}$

calculate $L_2 \Rightarrow \frac{\partial L_2}{\partial w} \Rightarrow w^* = w - \eta \frac{\partial L_2}{\partial w}$

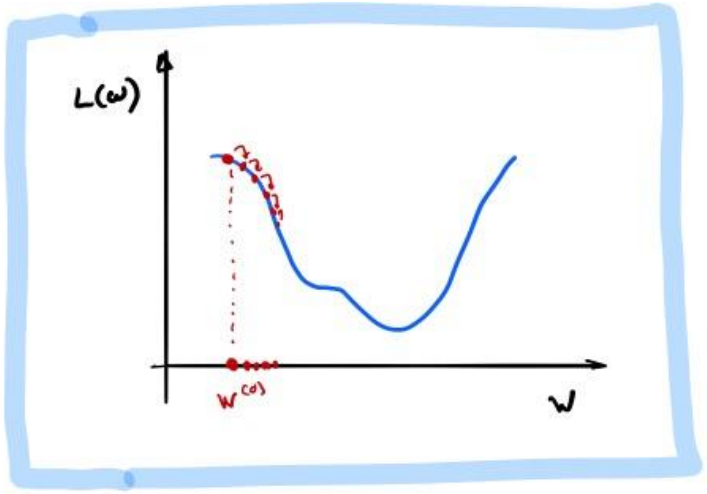
$L_B \Rightarrow \frac{\partial L_B}{\partial w} \Rightarrow w^* = w - \eta \frac{\partial L_B}{\partial w}$

COMPLETE DATA $\Rightarrow$ ONE EPOCH

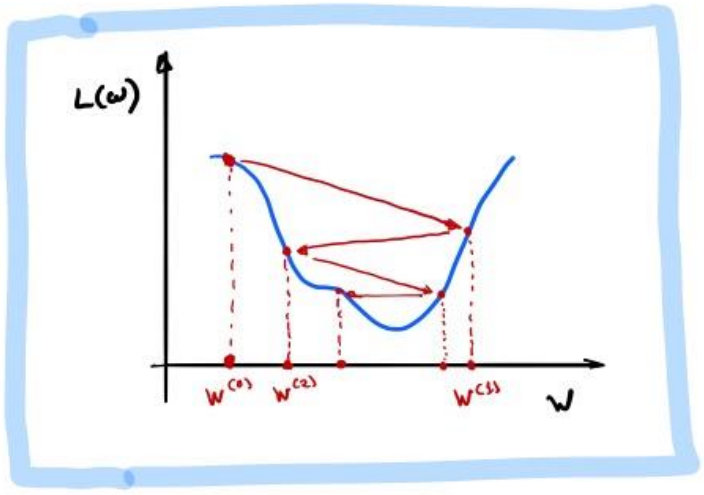
RESHUFFLE DATA AND REPEAT

Learning Rate

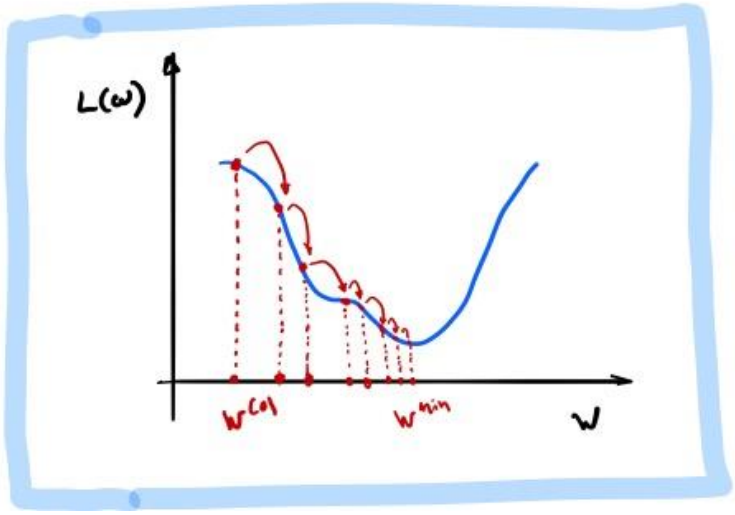
- Our choice of the learning rate has a significant impact on the performance of gradient descent.



When η is too small, the algorithm makes very little progress.

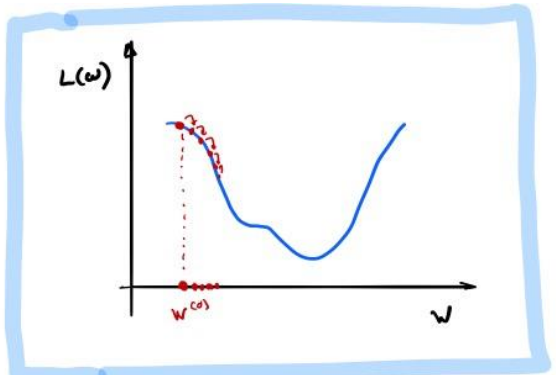


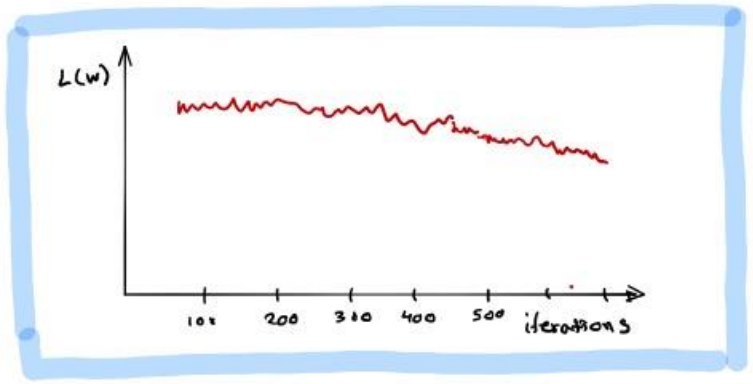
When η is too large, the algorithm may overshoot the minimum and has crazy oscillations.



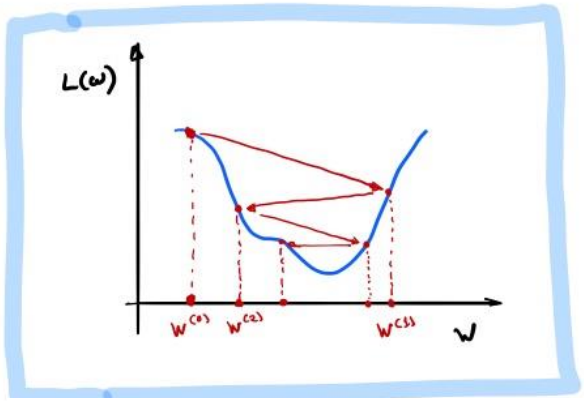
When η is appropriate, the algorithm will find the minimum. The algorithm **converges**!

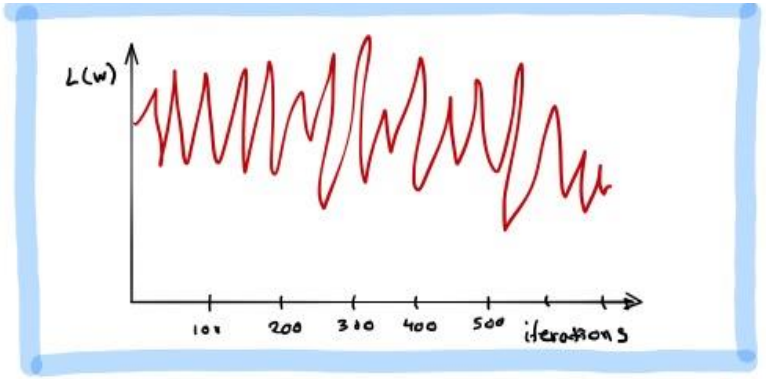
How can we tell when gradient descent is converging? We visualize the loss function at each step of gradient descent. This is called the **trace plot**.



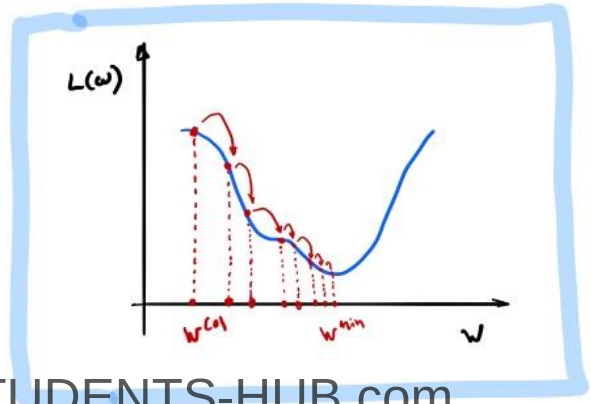


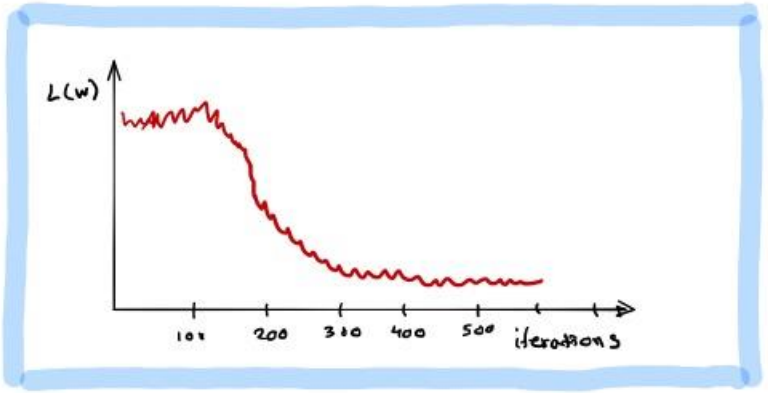
While the loss is decreasing throughout training, it does not look like descent hit the bottom.





Loss is mostly oscillating between values rather than converging.





The loss has decreased significantly during training. Towards the end, the loss stabilizes and it can't decrease further.

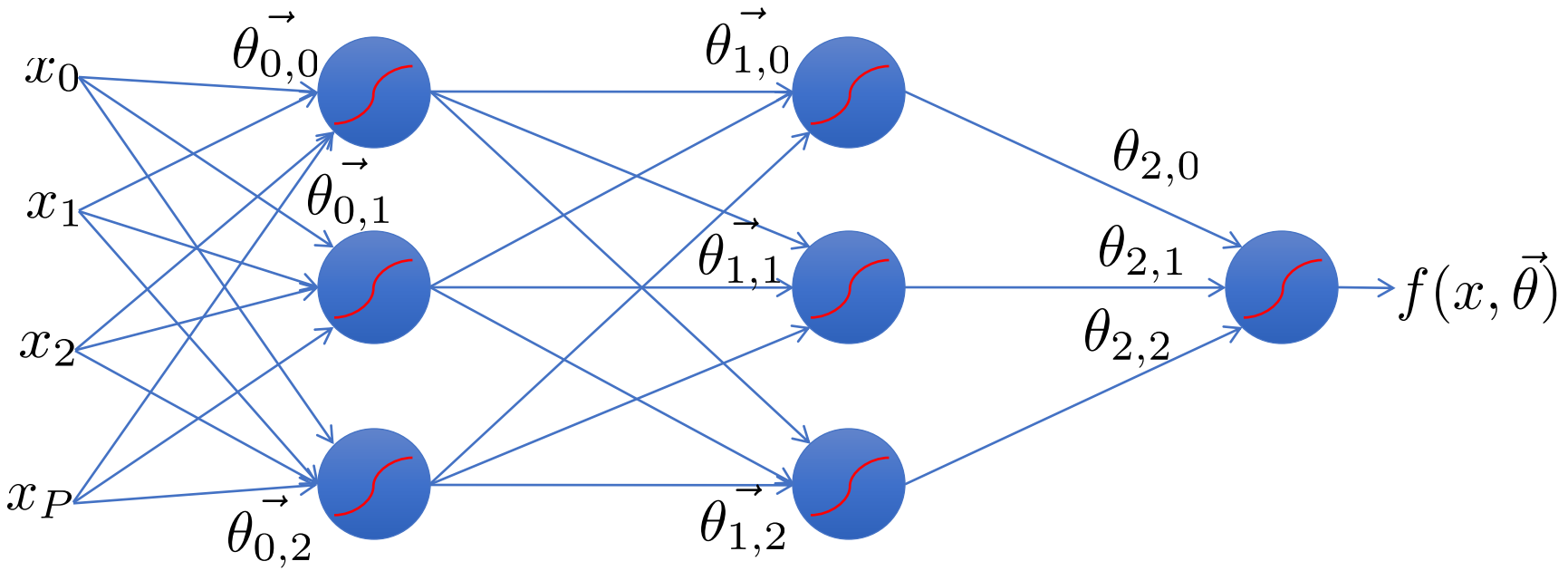
Learning Rate

There are many alternative methods which address how to set or adjust the learning rate, using the derivative or second derivatives and or the momentum.

More on this later.

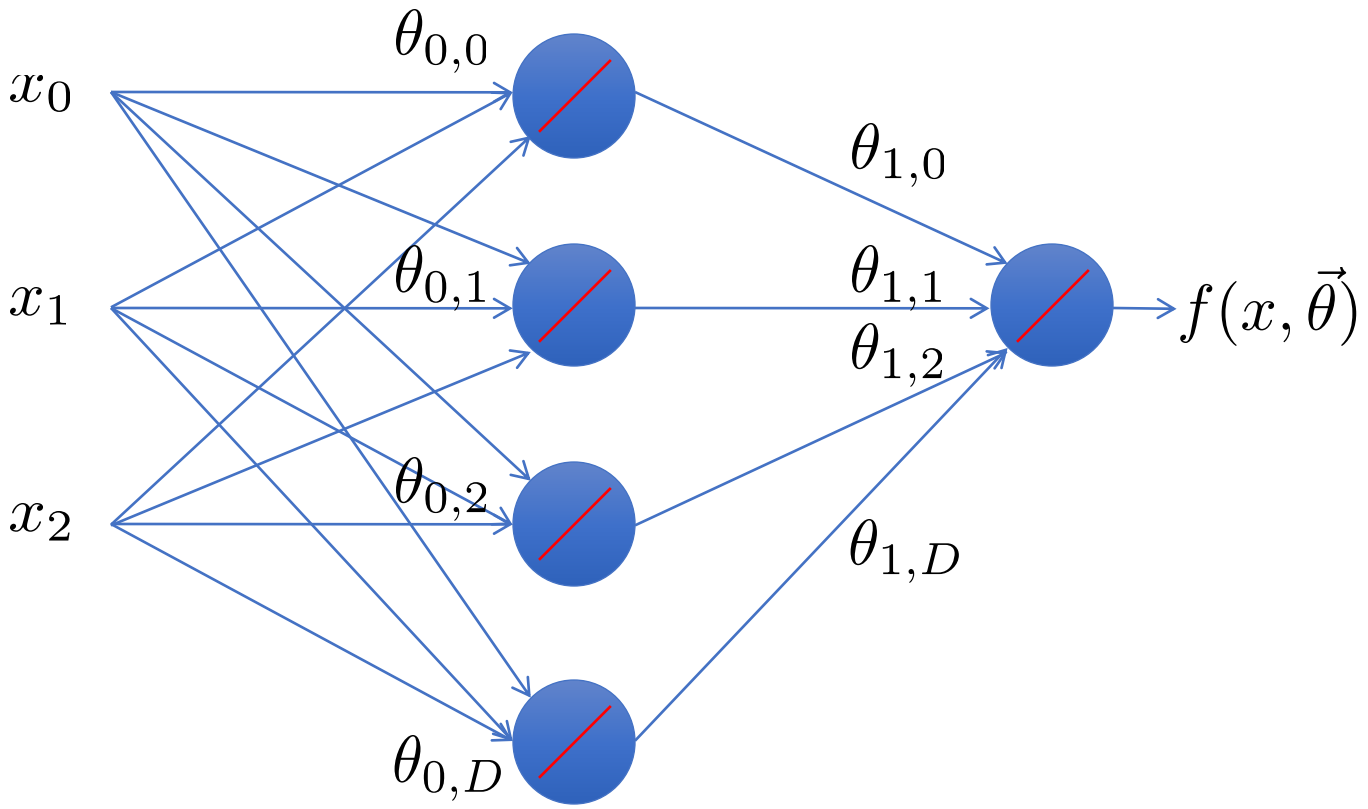
Multilayer Networks

- Stack Neurons together
- The output from one layer is the input to the next
- Each Layer has its own sets of weights



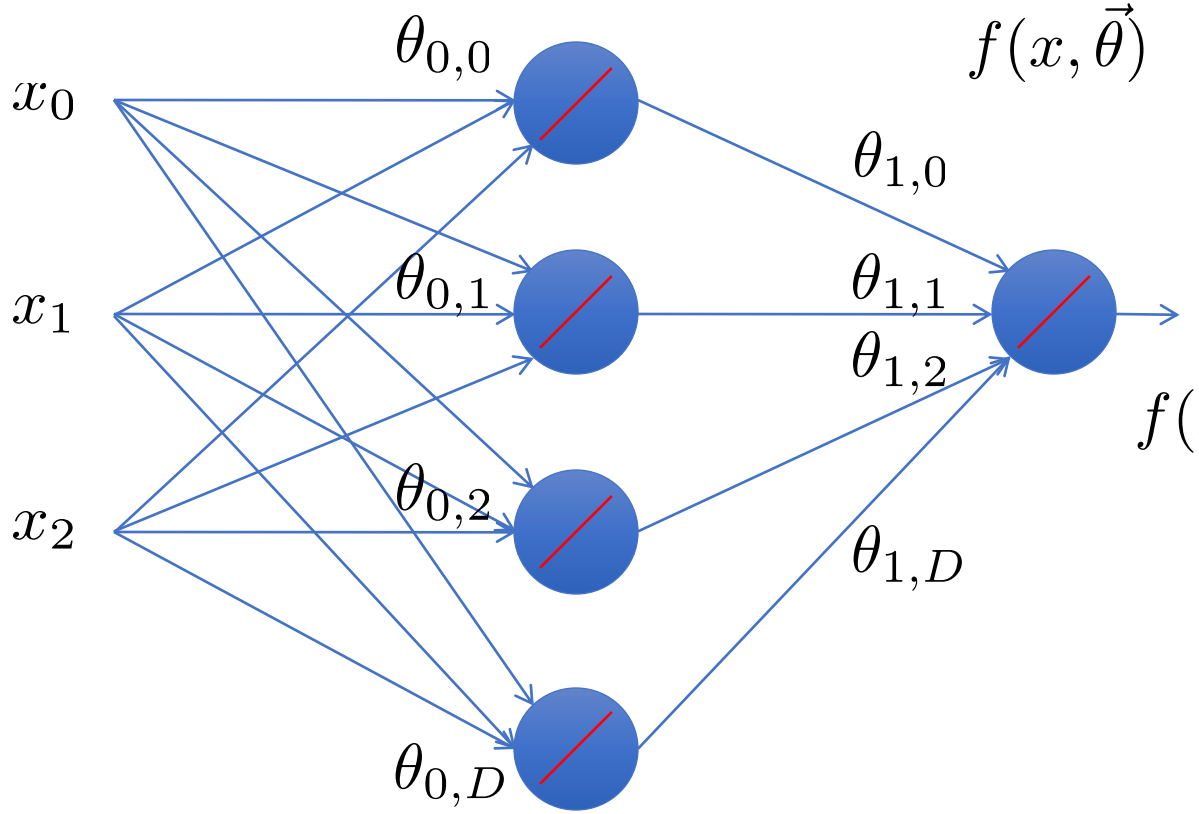
Linear Regression Neural Networks

- What happens when we arrange linear neurons in a multilayer network?



Linear Regression Neural Networks

- Nothing special happens.
- The product of two linear transformations is itself a linear transformation.



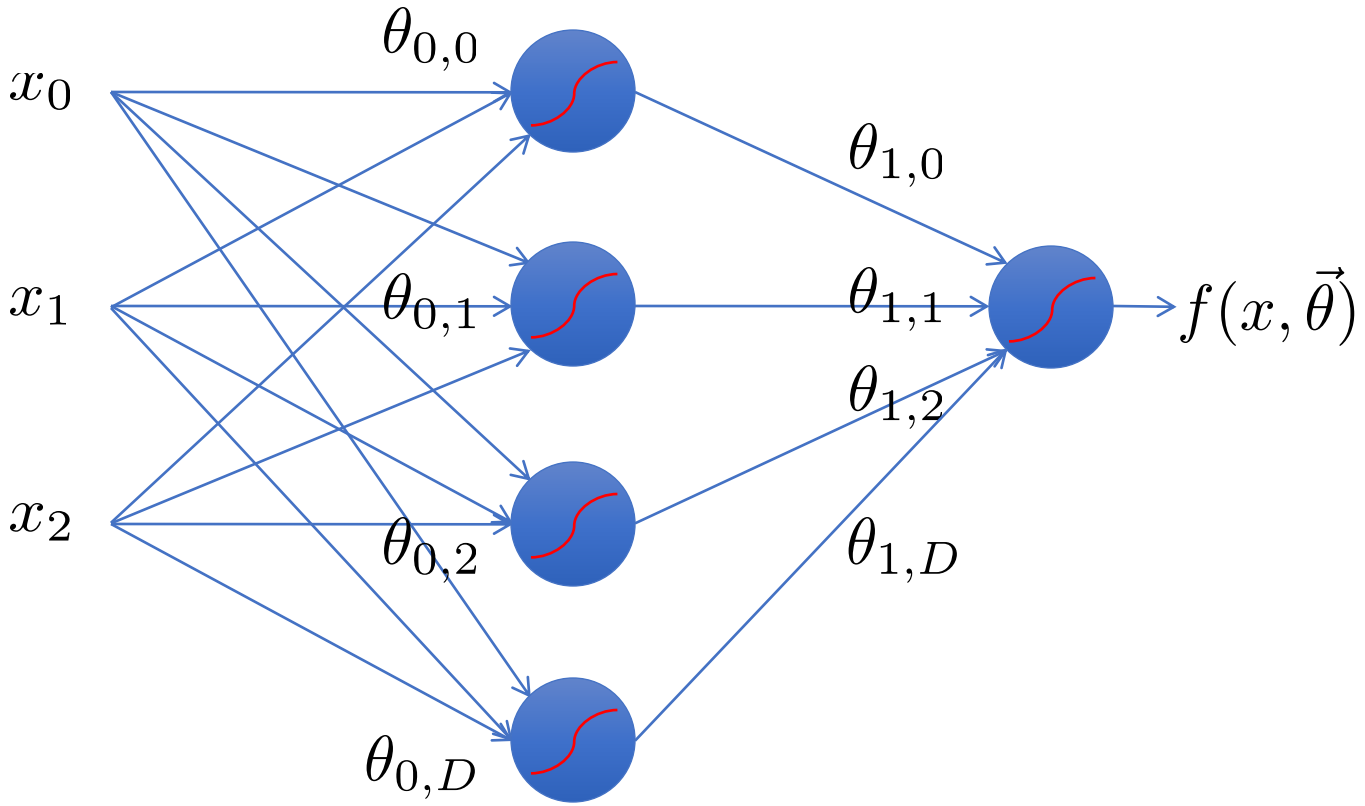
$$f(x,\vec{\theta}) = \sum_{i=0}^{D} \theta_{1,i} \sum_{n=0}^{N-1} \theta_{0,i,n} x_n$$

$$f(x,\vec{\theta}) = \sum_{i=0}^{D} \theta_{1,i} [\theta_{0,i}^T \vec{x}]$$

$$\boxed{f(x,\vec{\theta}) = \sum_{i=0}^{D} [\hat{\theta}_i^T \vec{x}]}$$

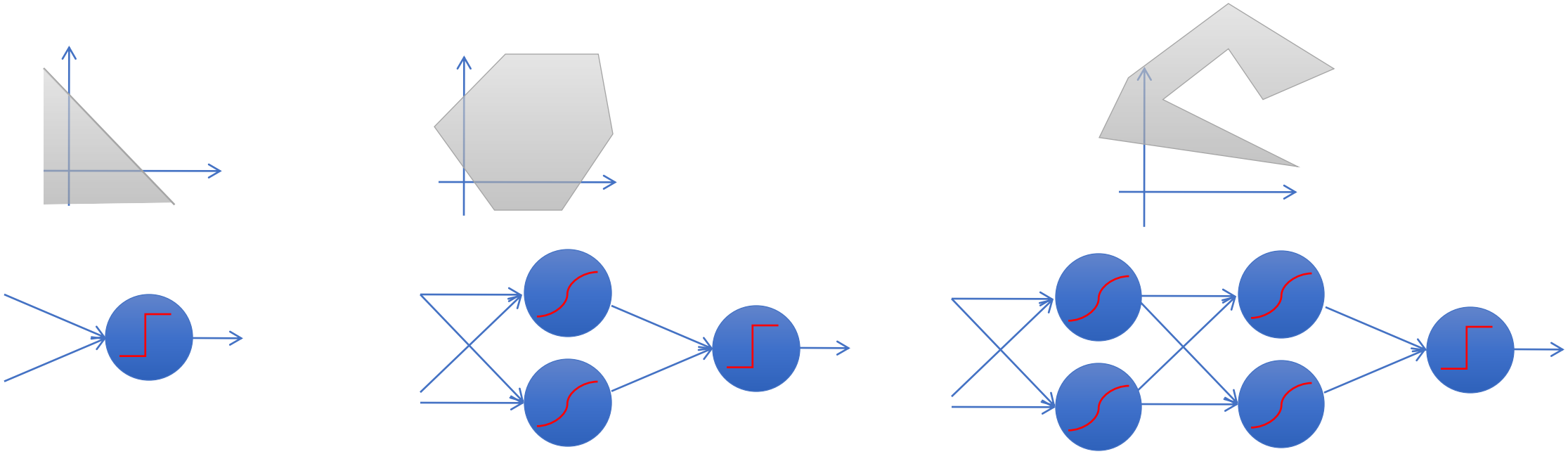
Neural Networks

- We want to introduce non-linearities to the network.
- Non-linearities allow a network to identify complex regions in space



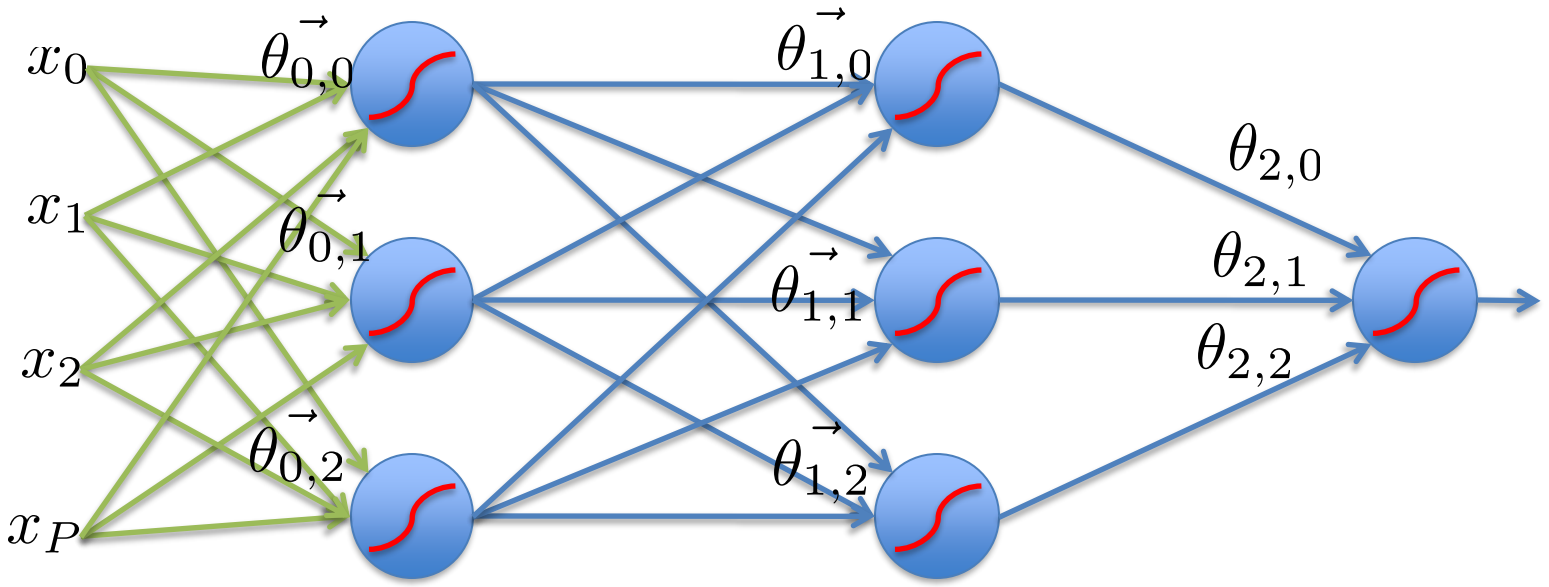
Linear Separability

- More layers can handle more complicated spaces – but require more parameters
- Each node splits the feature space with a hyperplane
- A 2-layer network can represent any convex hull.



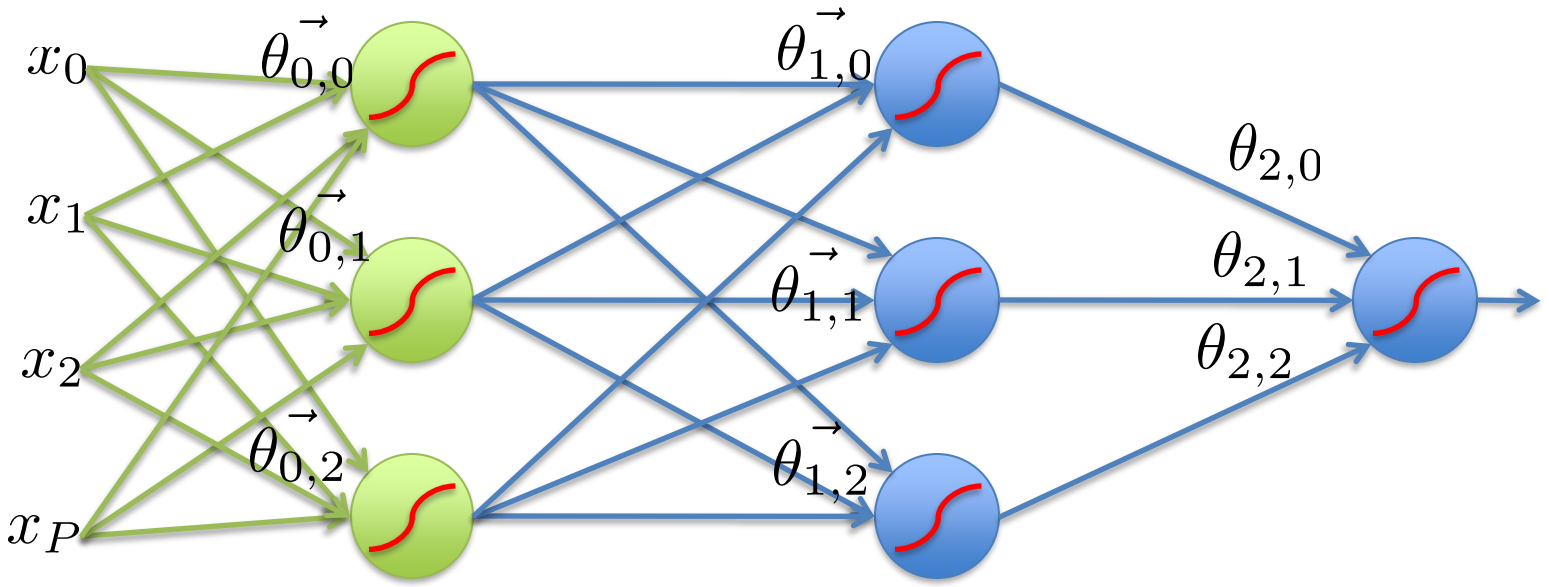
Feed-Forward Networks

- Predictions are fed forward through the network to classify



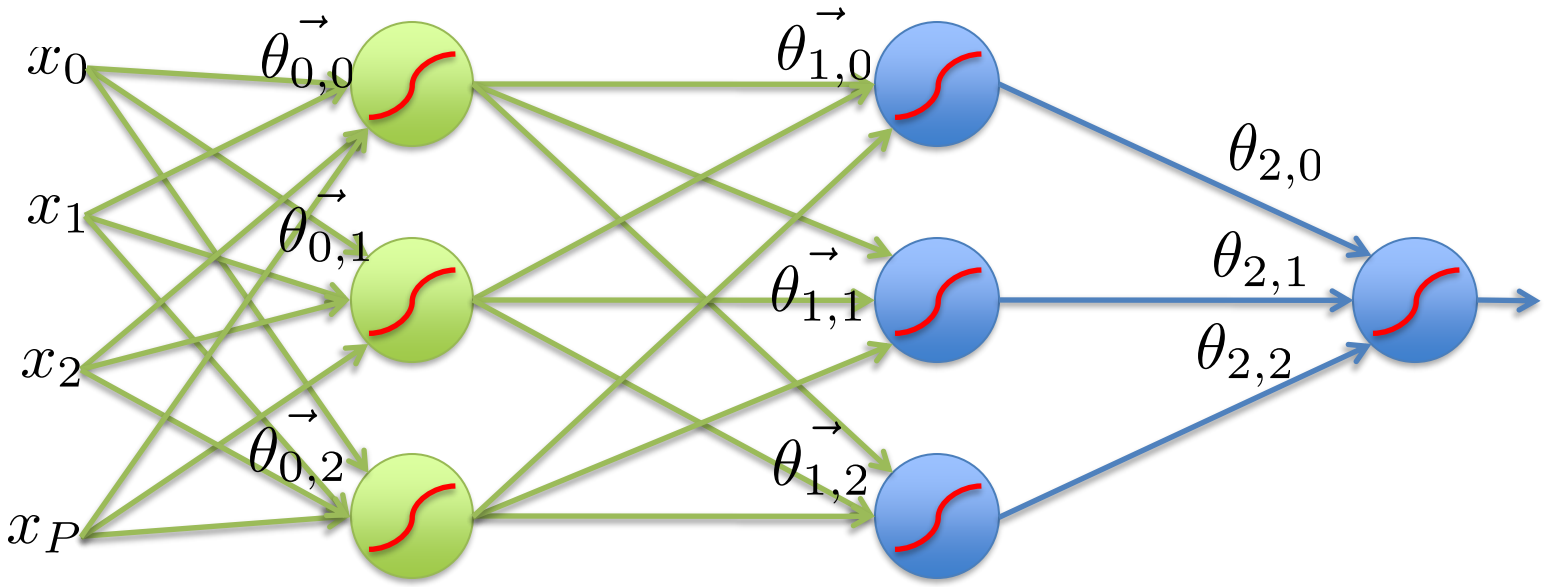
Feed-Forward Networks

- Predictions are fed forward through the network to classify



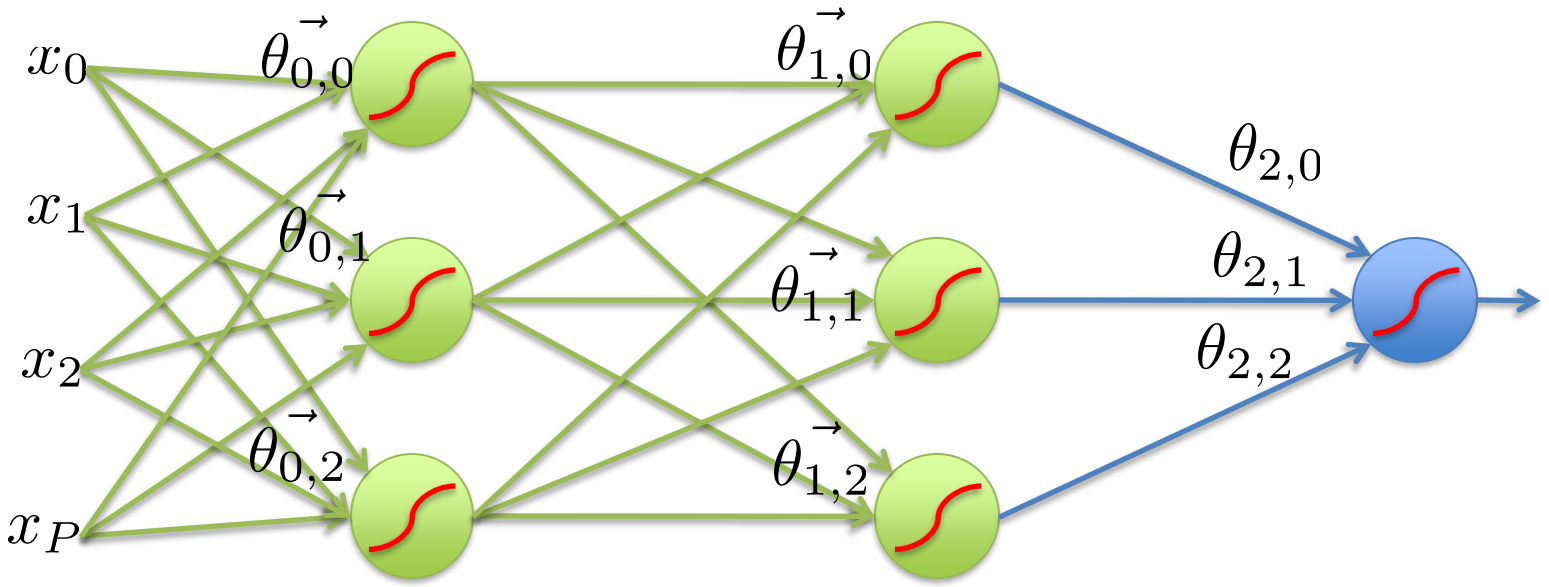
Feed-Forward Networks

- Predictions are fed forward through the network to classify



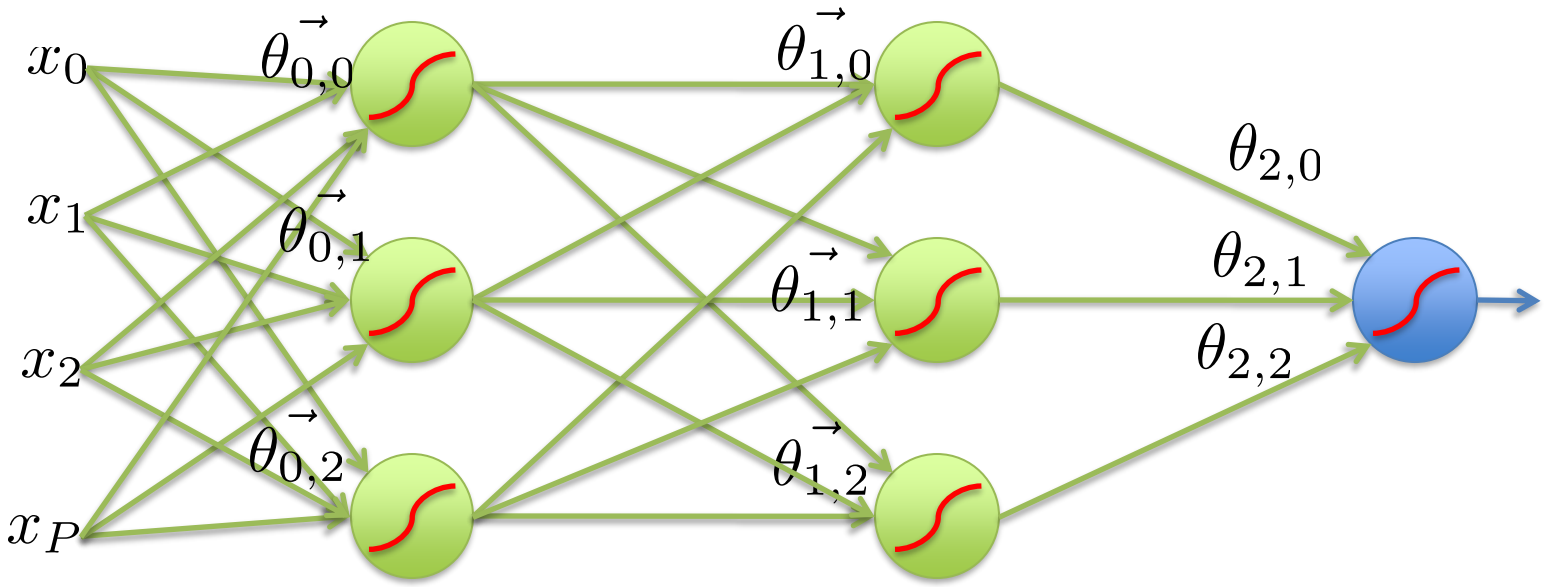
Feed-Forward Networks

- Predictions are fed forward through the network to classify



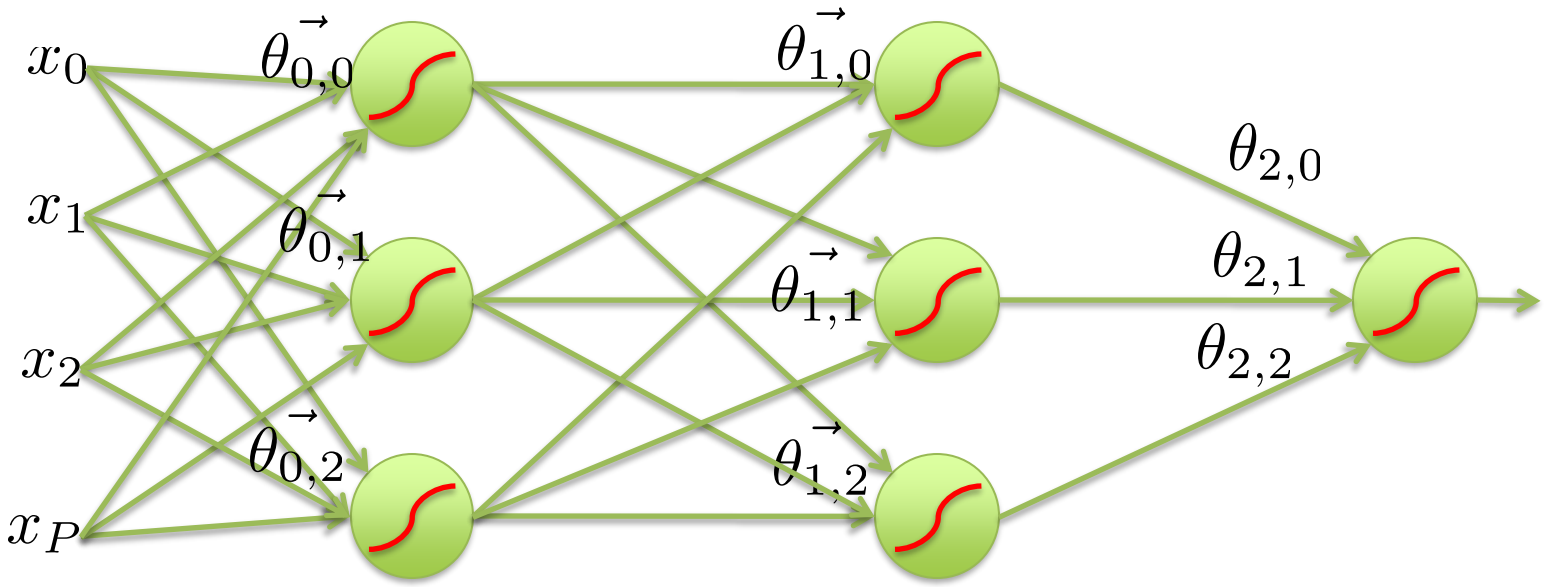
Feed-Forward Networks

- Predictions are fed forward through the network to classify



Feed-Forward Networks

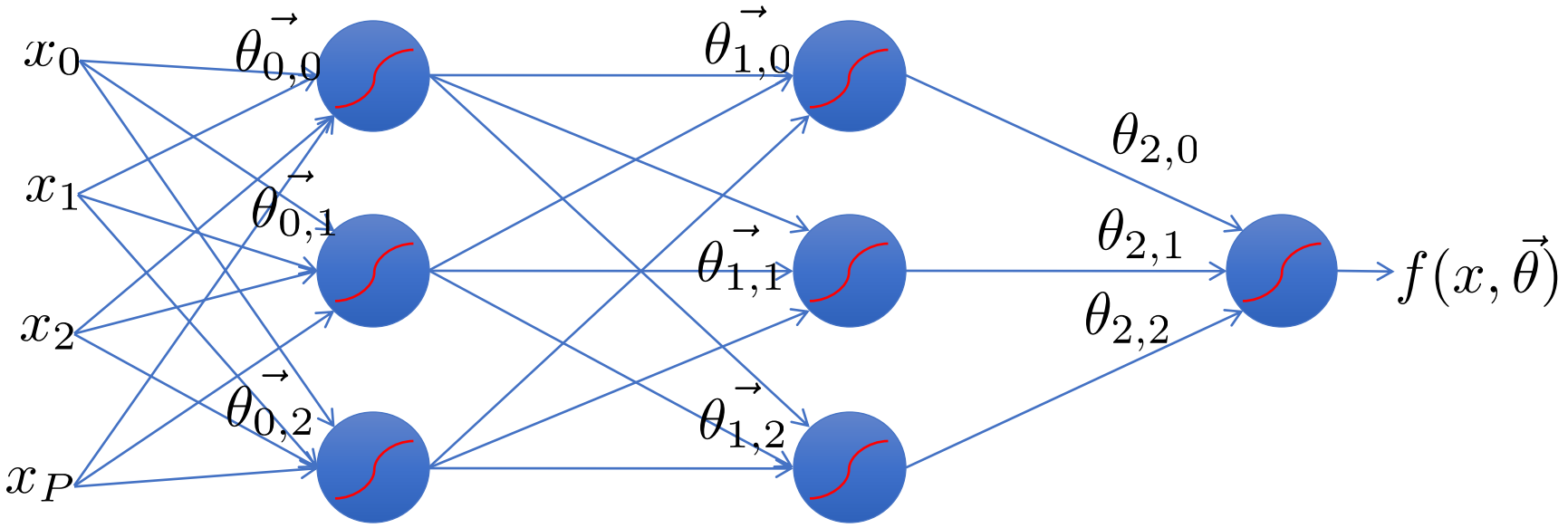
- Predictions are fed forward through the network to classify



Error Backpropagation

- We will do gradient descent on the whole network.

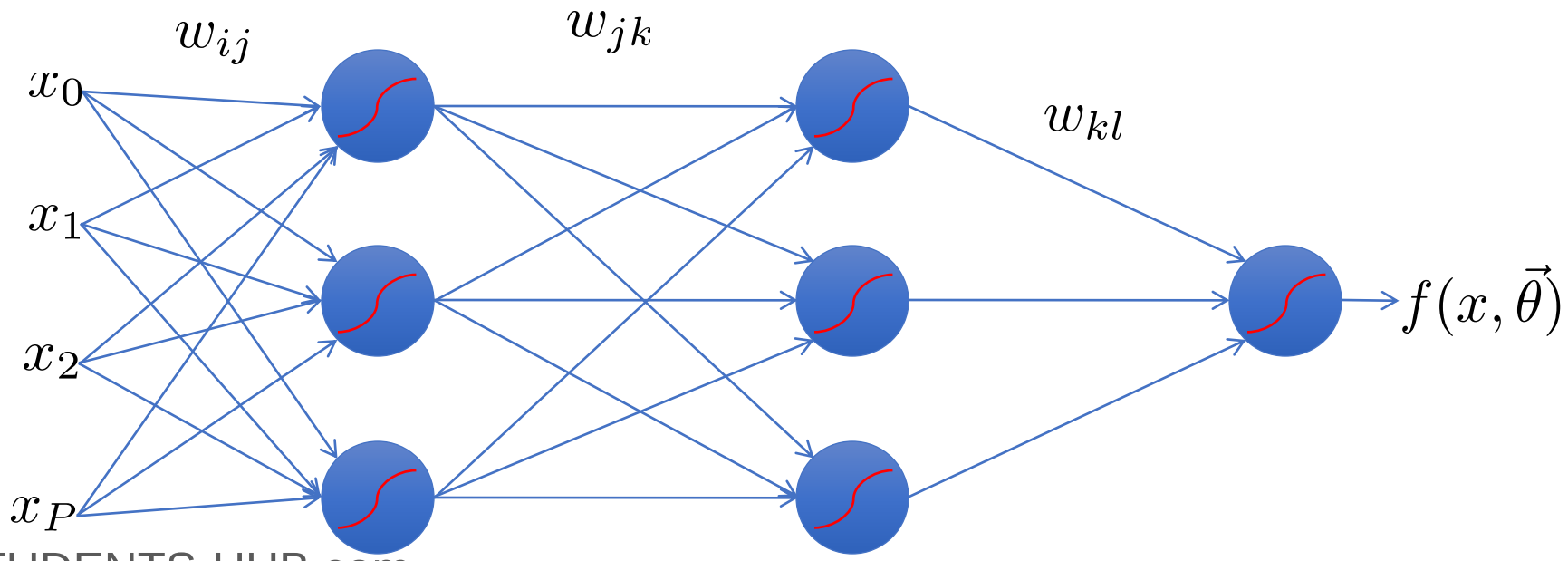
- Training will proceed from the last layer to the first.



Error Backpropagation

- Introduce variables over the neural network

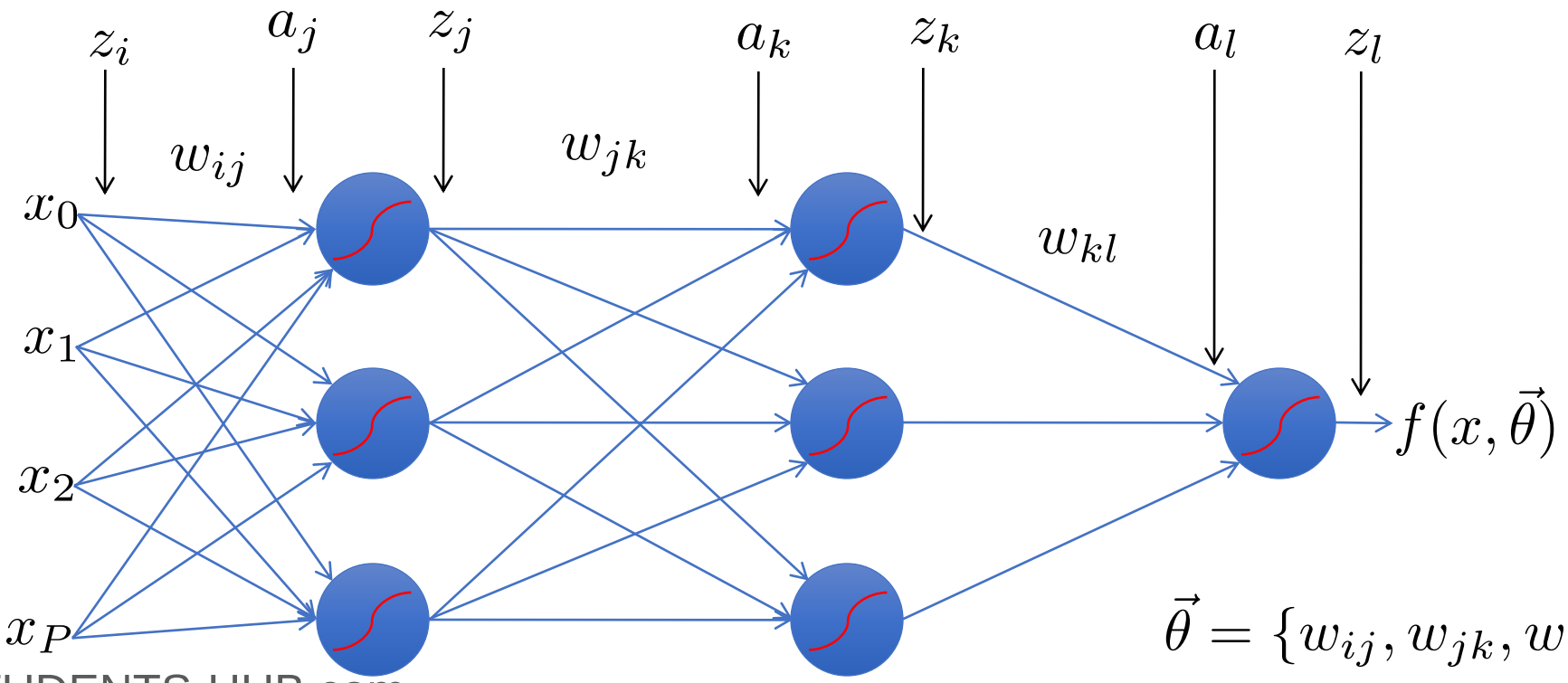
$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$



Error Backpropagation

- Introduce variables over the neural network

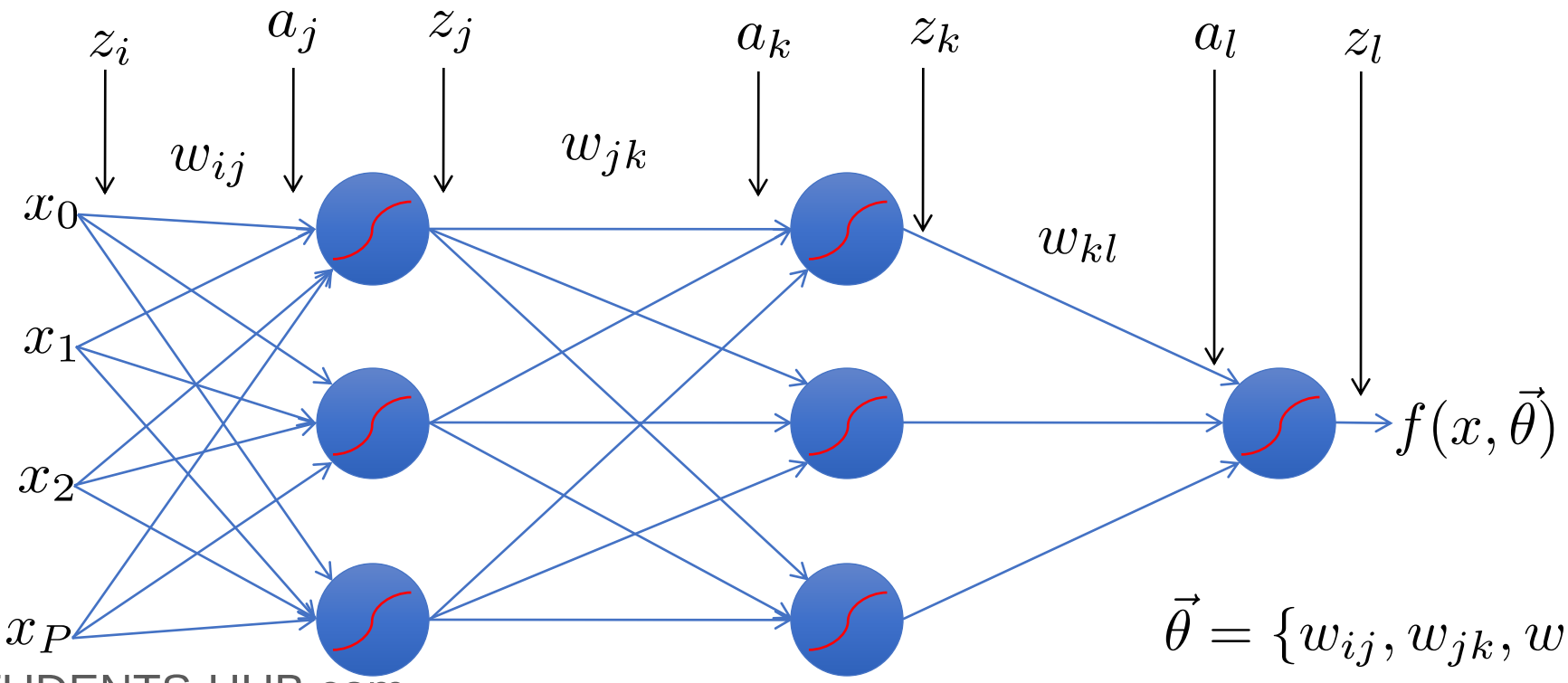
- Distinguish the input and output of each node



Error Backpropagation

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

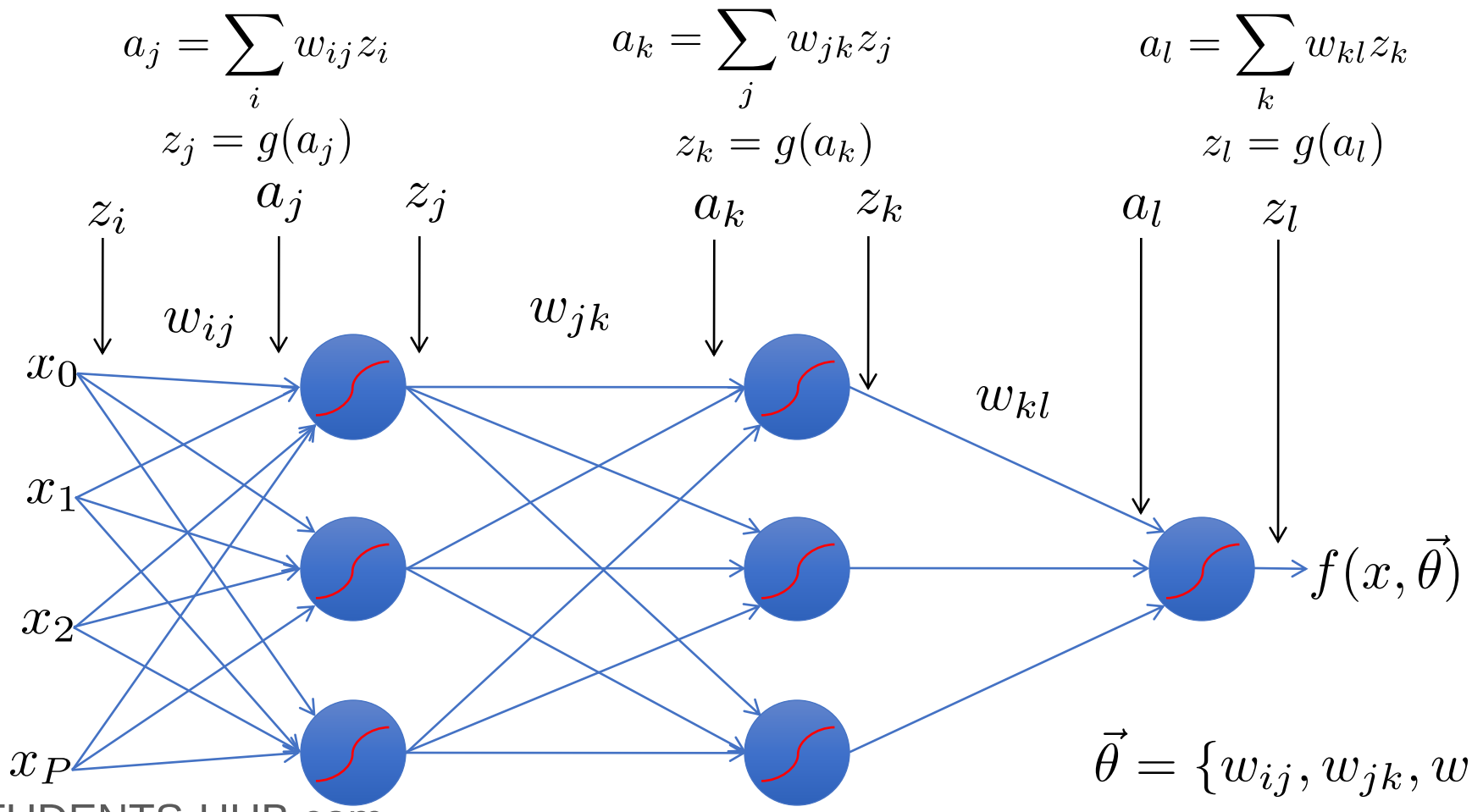
$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Error Backpropagation

- Training: Take the gradient of the last component and iterate backwards

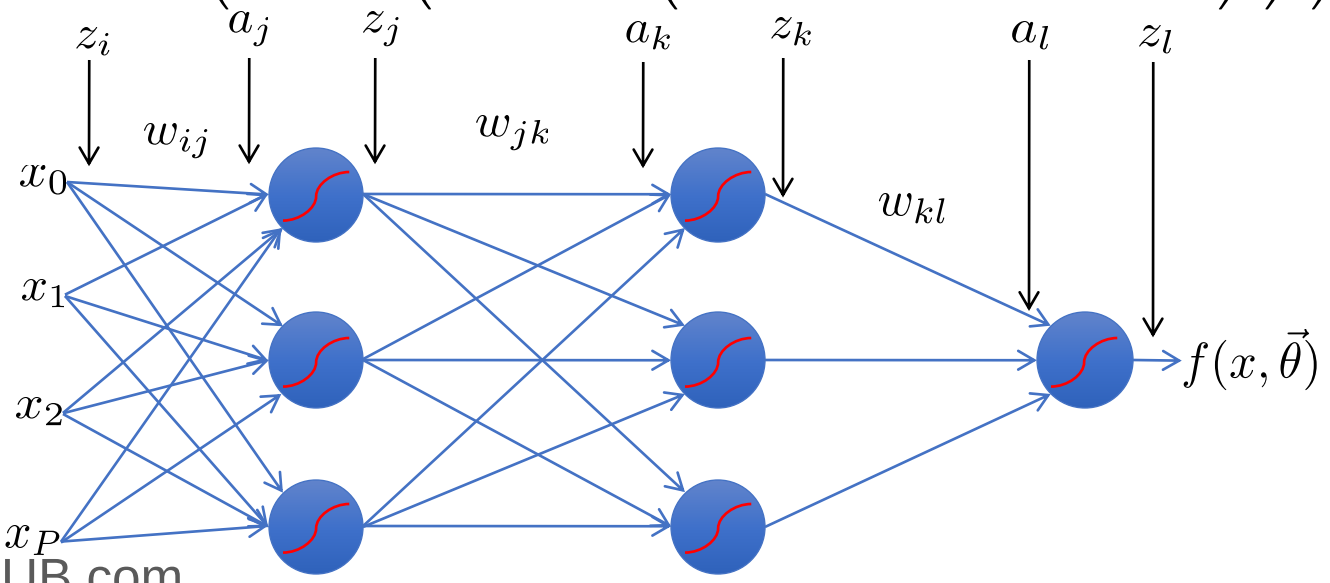


Error Backpropagation

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$ Empirical Risk Function

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl} g\left(\sum_j w_{jk} g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$



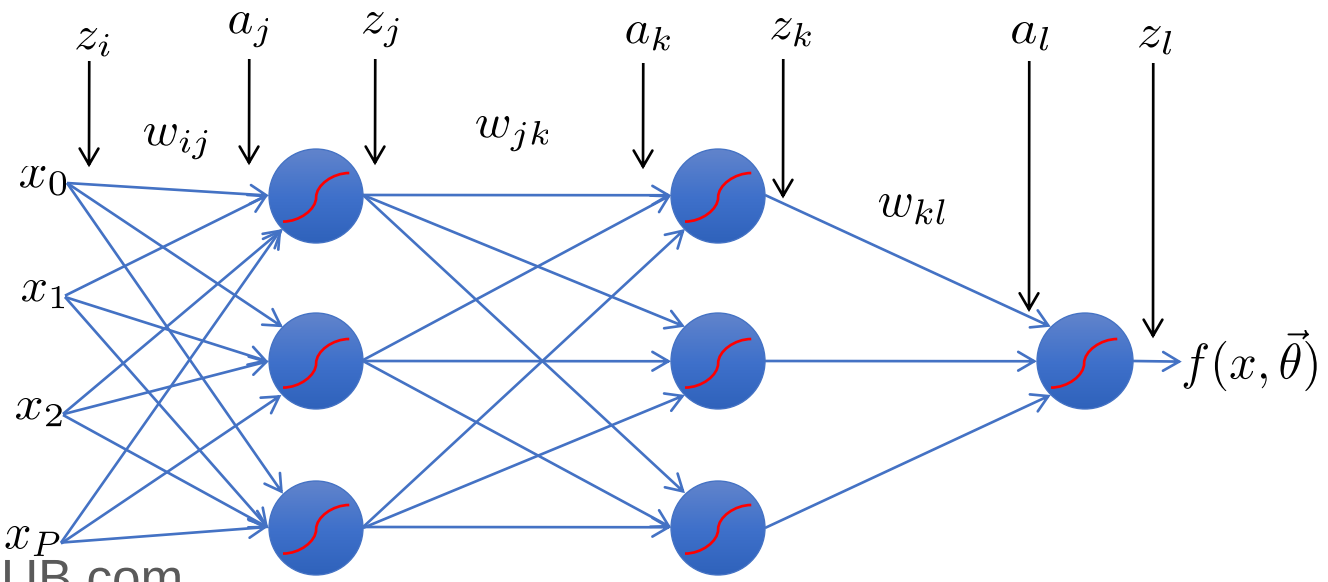
Error Backpropagation

- Optimize last layer weights w_{kl}: $L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



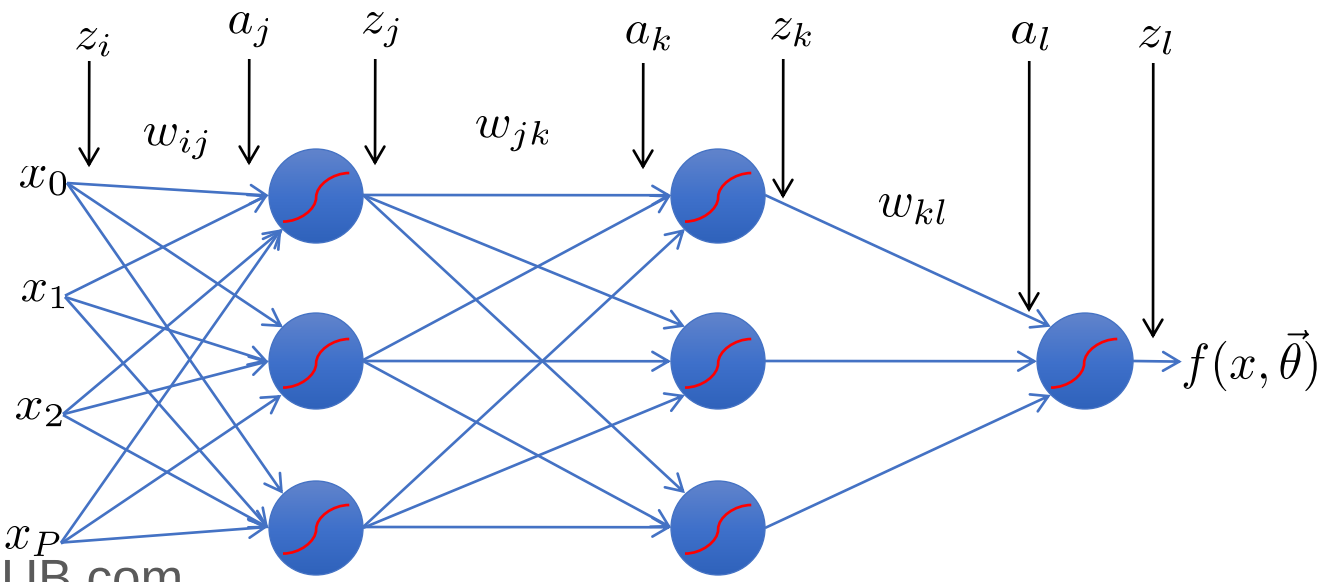
Error Backpropagation

- Optimize last layer weights w_{kl}: $L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$



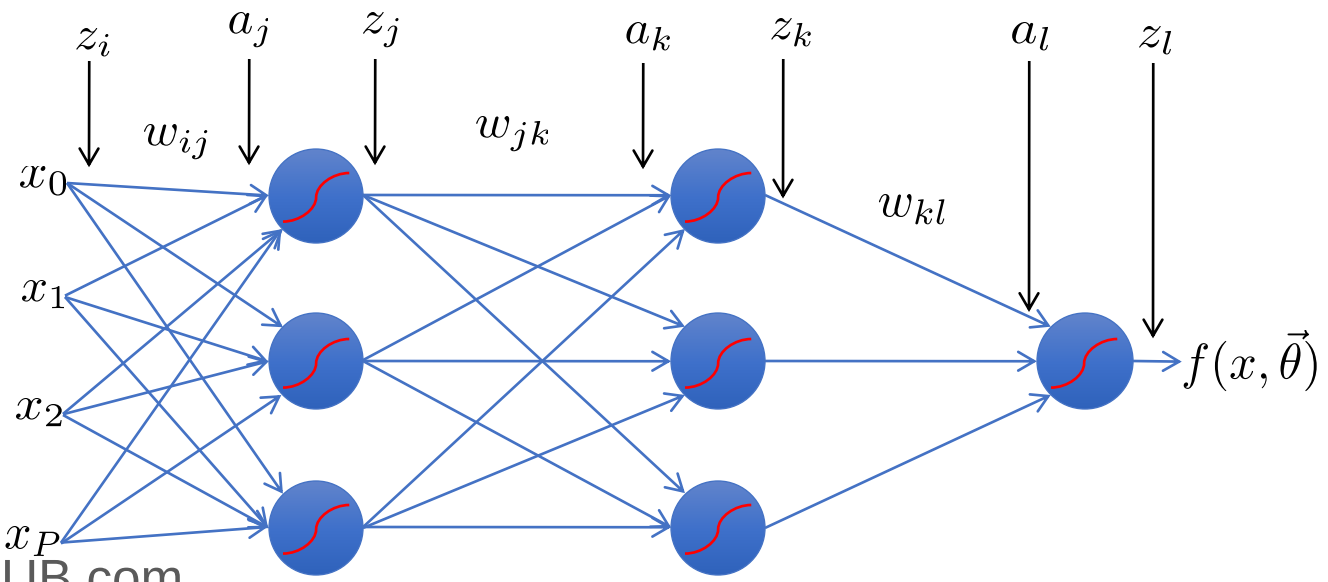
Error Backpropagation

- Optimize last hidden weights w_{jk}: $\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}}\frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule



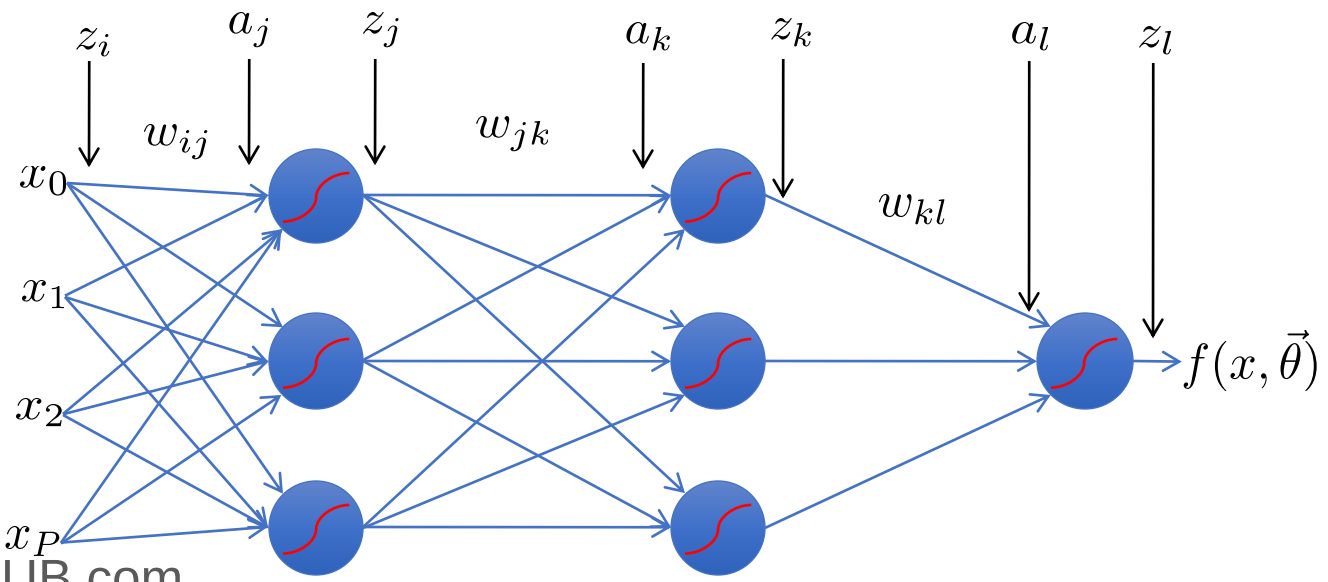
Error Backpropagation

- Optimize last hidden weights w_{jk}: $\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}}\frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right][z_{j,n}]$$



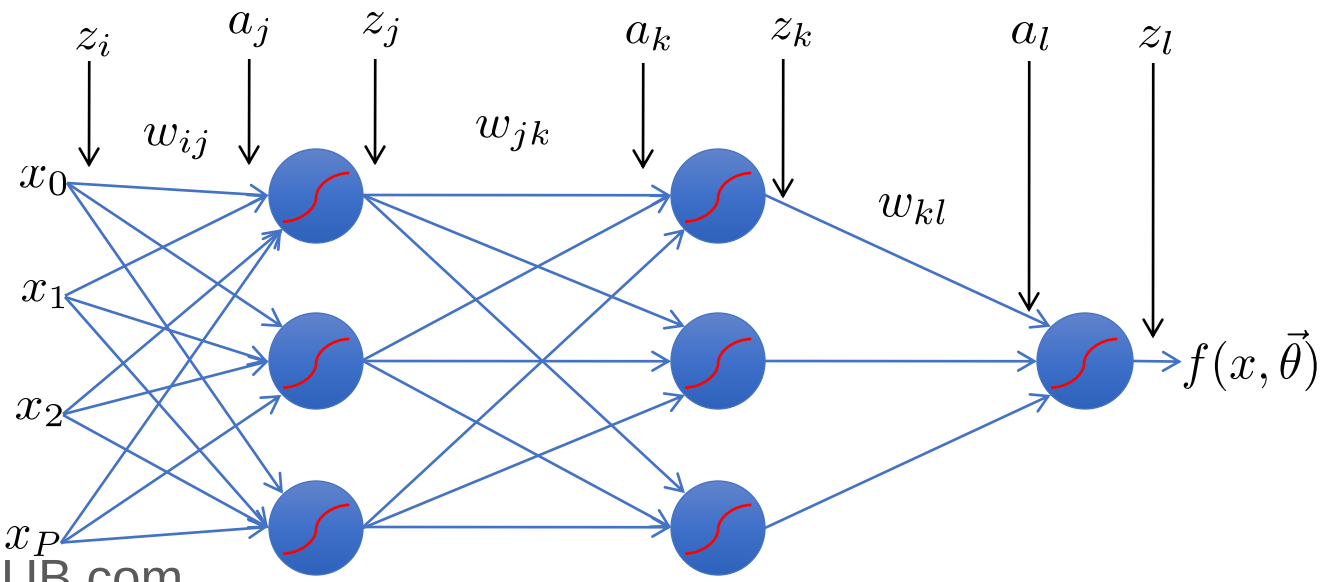
Error Backpropagation

- Optimize last hidden weights w_{jk}: $\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right][z_{j,n}]$$

$$a_l = \sum_k w_{kl} g(a_k)$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l w_{kl} g'(a_{k,n})\right][z_{j,n}] = \frac{1}{N}\sum_n [\delta_{k,n}][z_{j,n}]$$



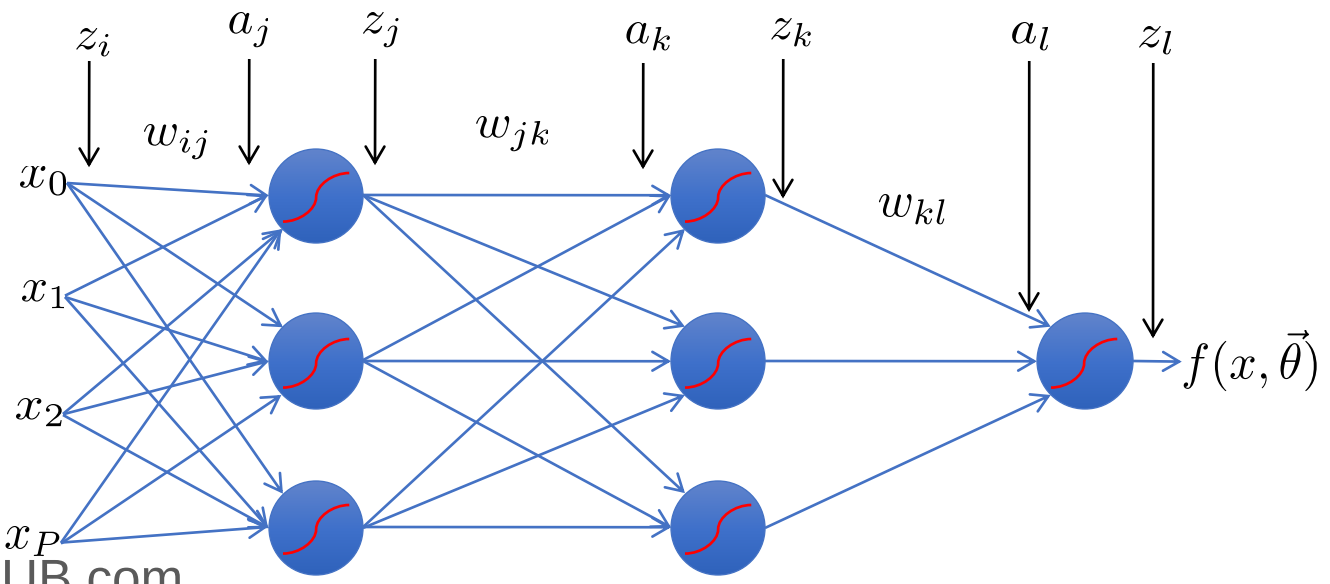
Error Backpropagation

- Repeat for all hidden layers:

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

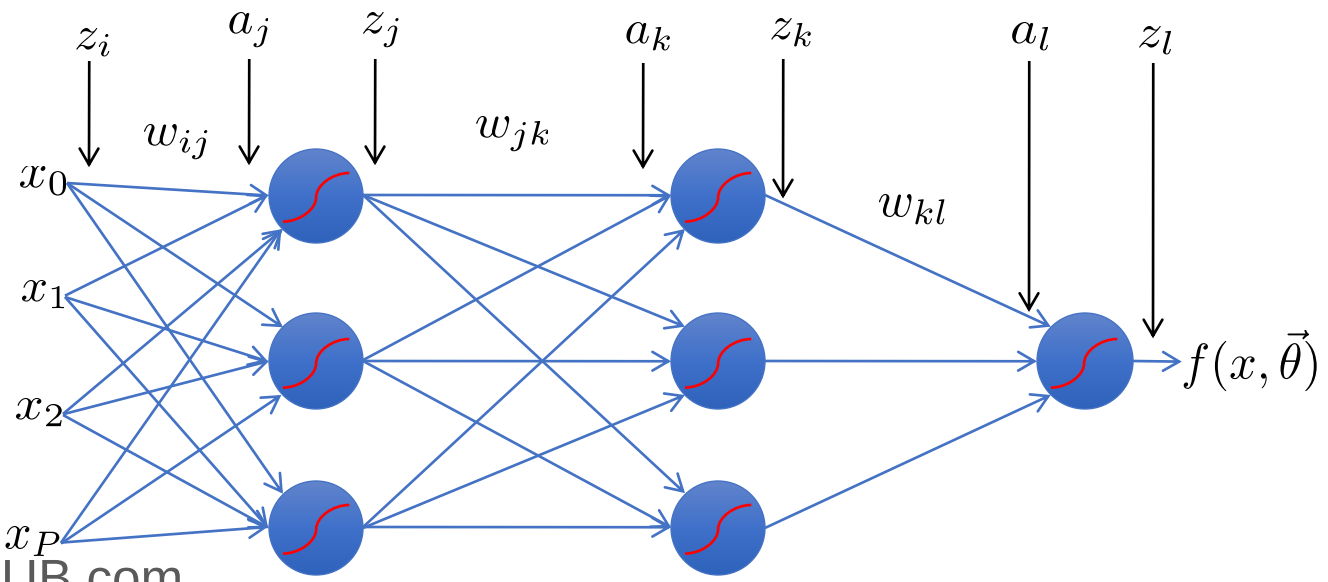
$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



Error Backpropagation

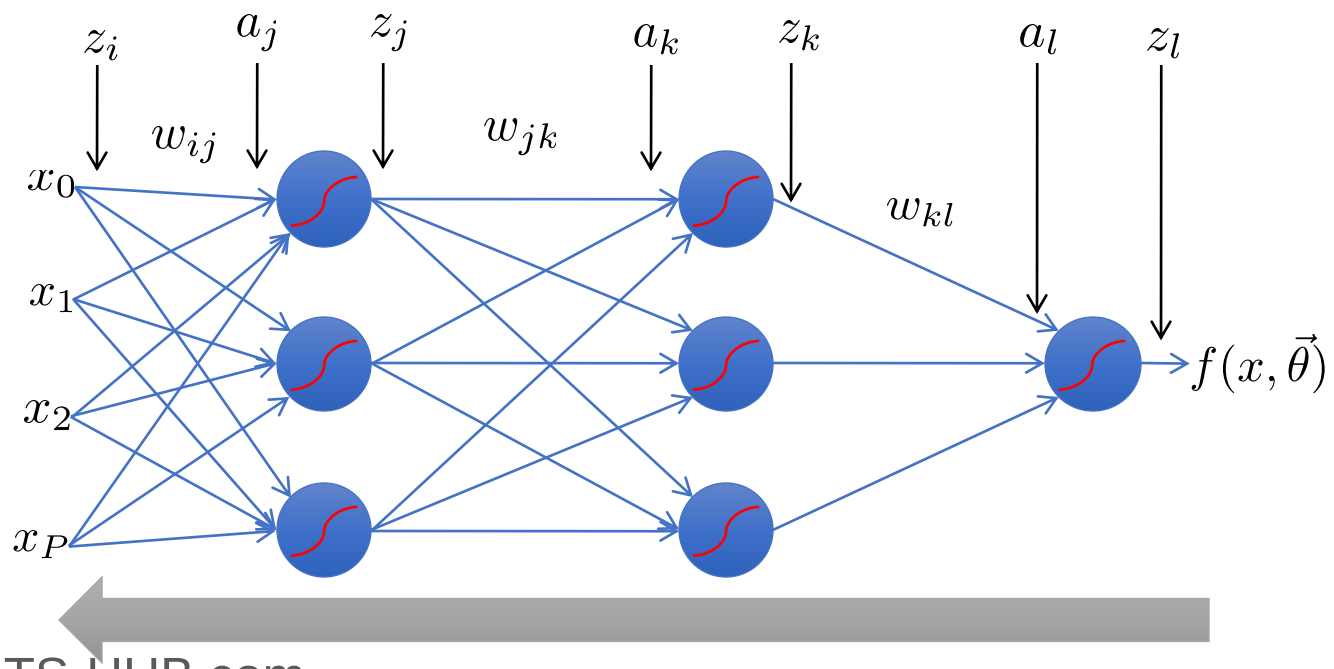
- Now that we have well defined gradients for each parameter, update using Gradient Descent:

$$
\begin{aligned}
w_{ij}^{t+1} &= w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}} \\
w_{jk}^{t+1} &= w_{jk}^{t} - \eta \frac{\partial R}{w_{kl}} \\
w_{kl}^{t+1} &= w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}}
\end{aligned}
$$



Error Back-propagation

- Error backpropagation unravels the multivariate chain rule and solves the gradient for each partial component separately.
- The target values for each layer come from the next layer.
- This feeds the errors back along the network.



Problems with Neural Networks

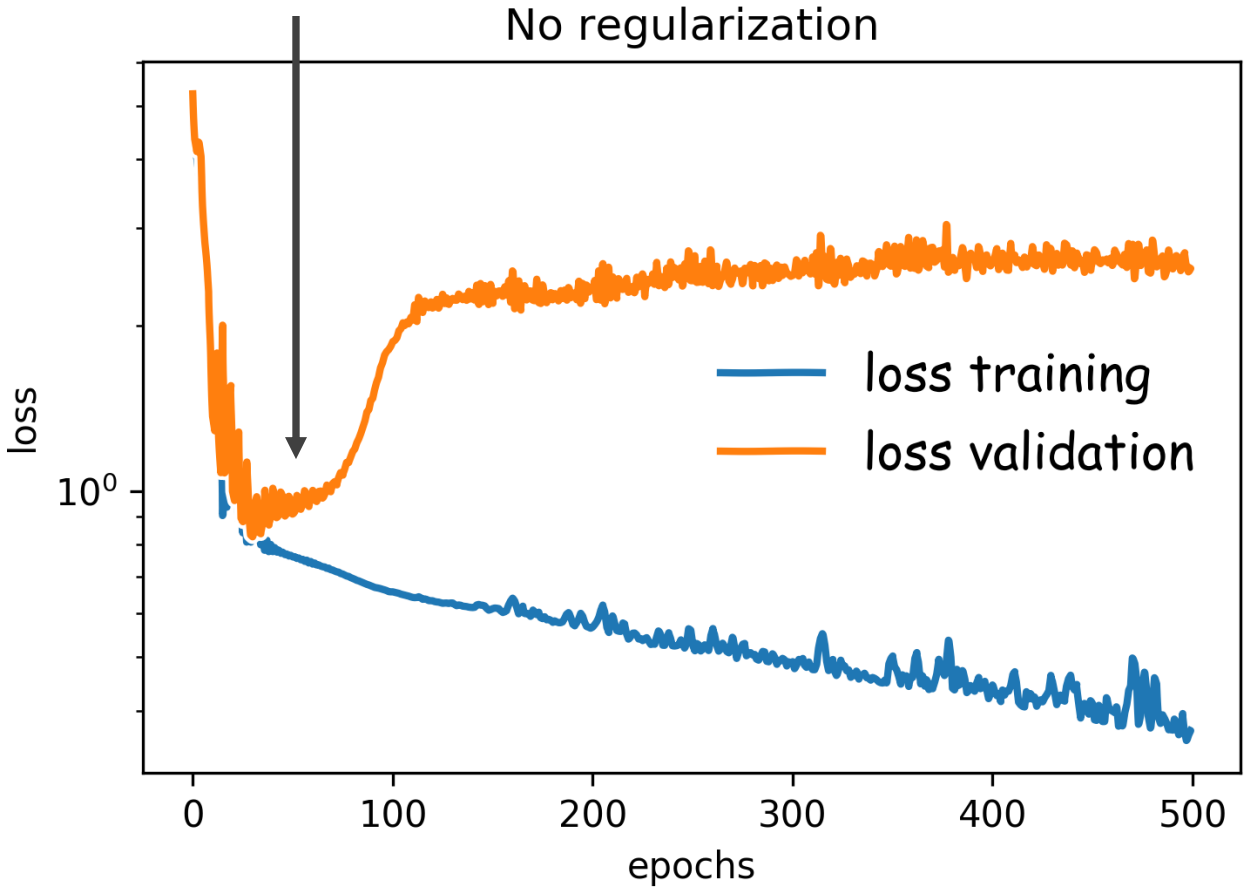
- Interpretation of Hidden Layers
- Overfitting (Many parameters)
- Require very large annotated datasets

Overfitting

- Augment training data by applying transformations on the training data (mirroring, rotation, affine, …)
- Norm Penalties: L_2 and L_1 regularization
- Early Stopping
- Use “dropout” to regularize the weights in the globally connected layers (which contain most of the parameters).
  - Dropout means that half of the hidden units in a layer are randomly removed for each training example.
  - This stops hidden units from relying too much on other hidden units.

Early Stopping

Early stopping: terminate while validation set performance is better. Sometimes is worth waiting a little before stopping. This is called `patience`.

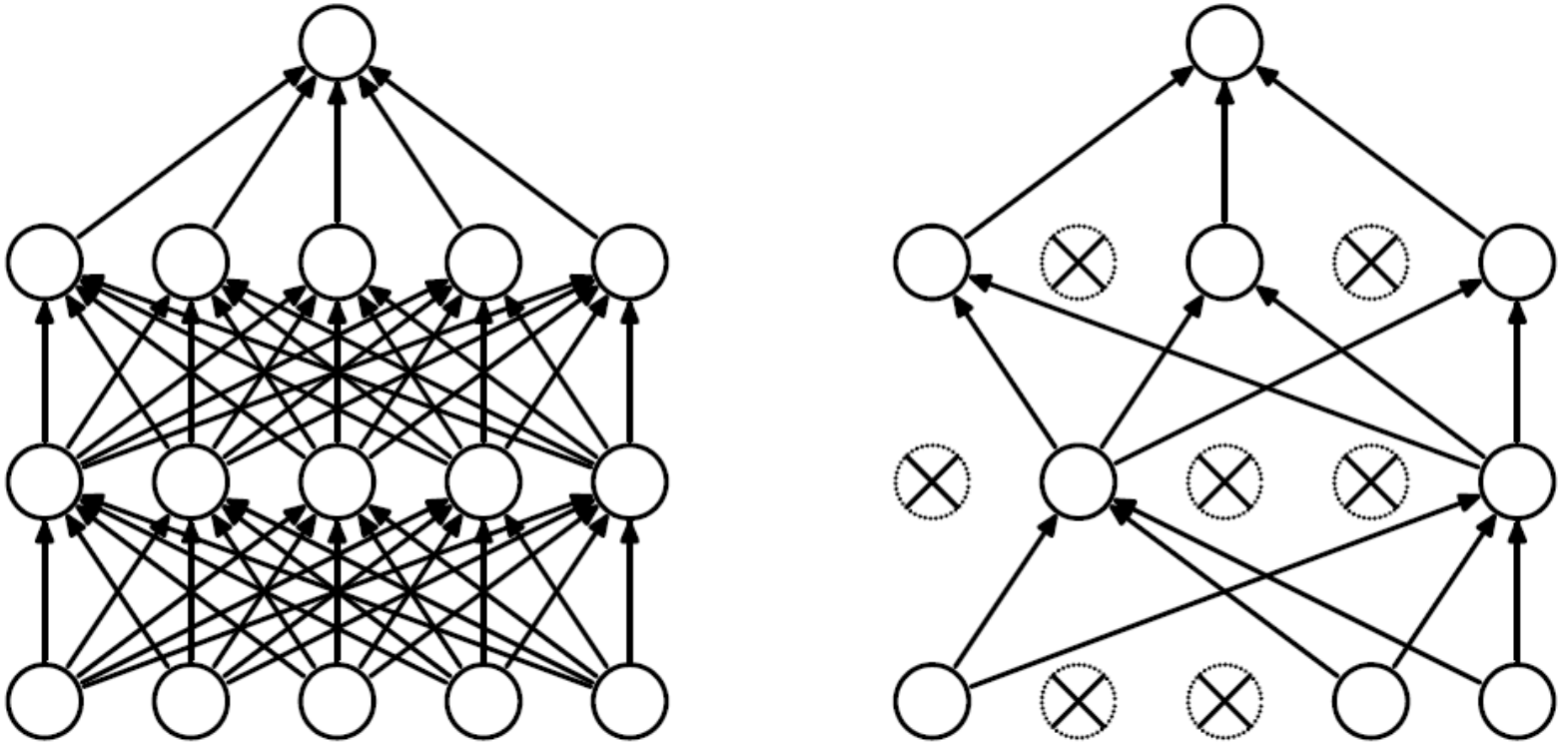


Patience is defined as the number of epochs to wait before early stop if no progress on the validation set.

The patience is often set somewhere between 10 and 100 (10 or 20 is more common), but it really depends on the dataset and network.

Dropout

- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.

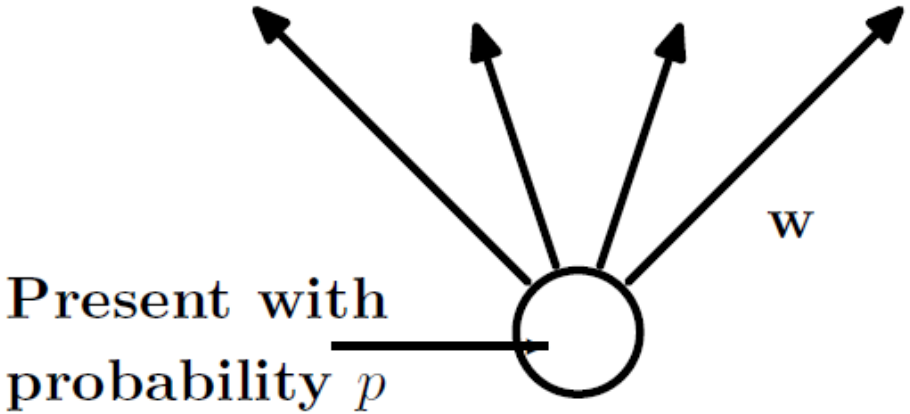


(a) Standard Neural Net

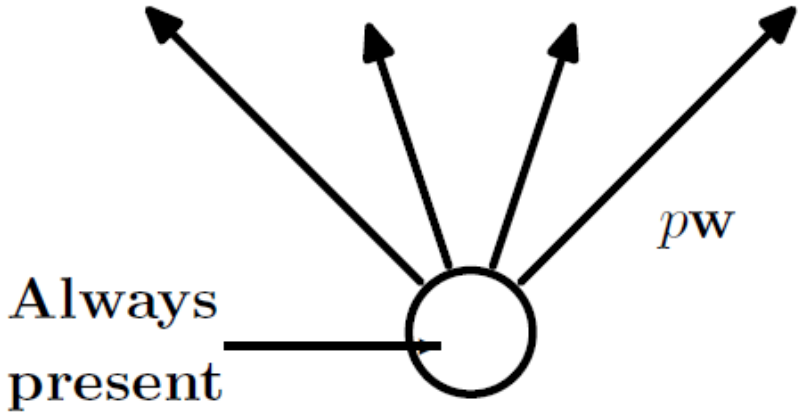
(b) After applying dropout.

Dropout

- At test time, all nodes are used but the weights are scaled.



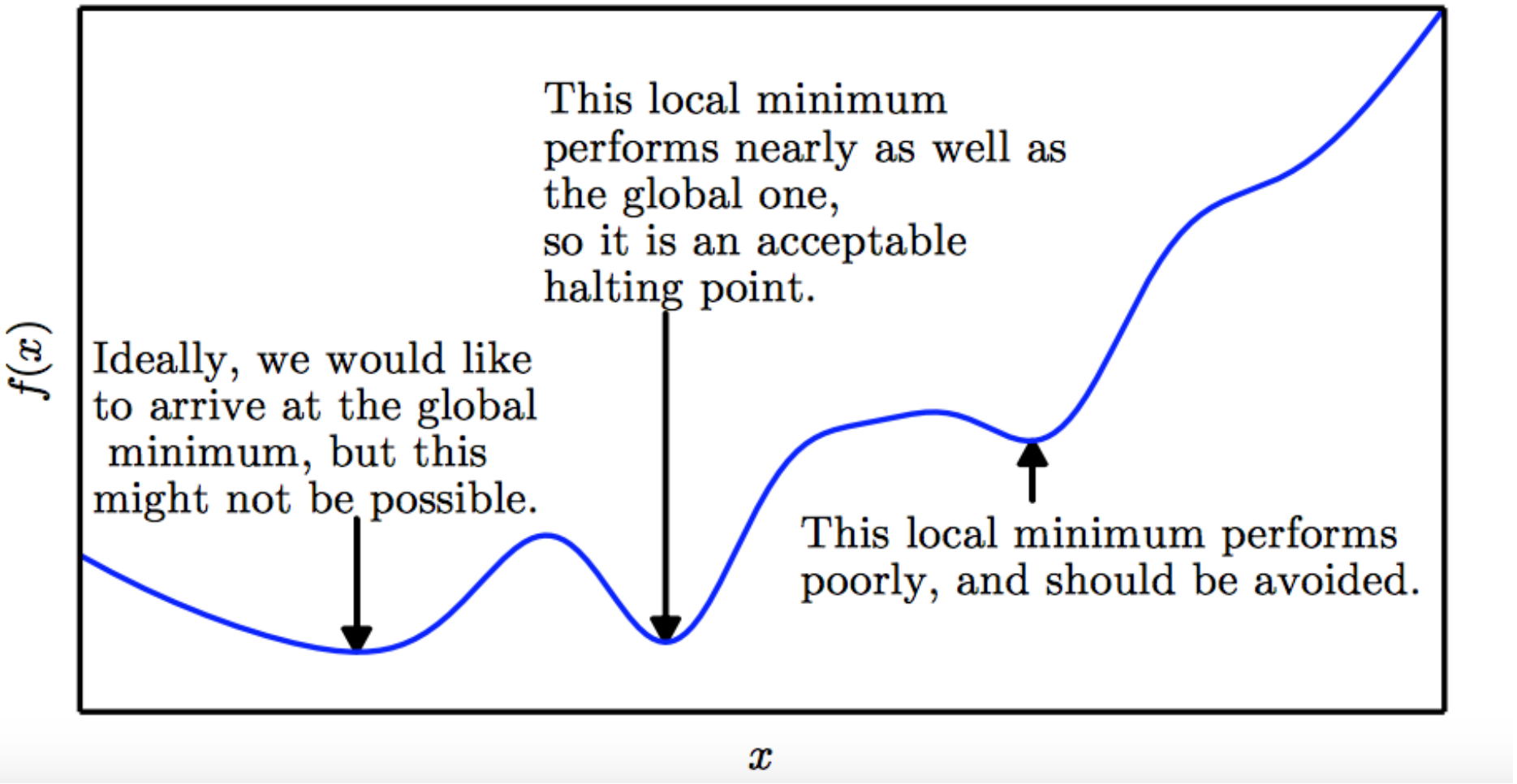
(a) At training time



(b) At test time

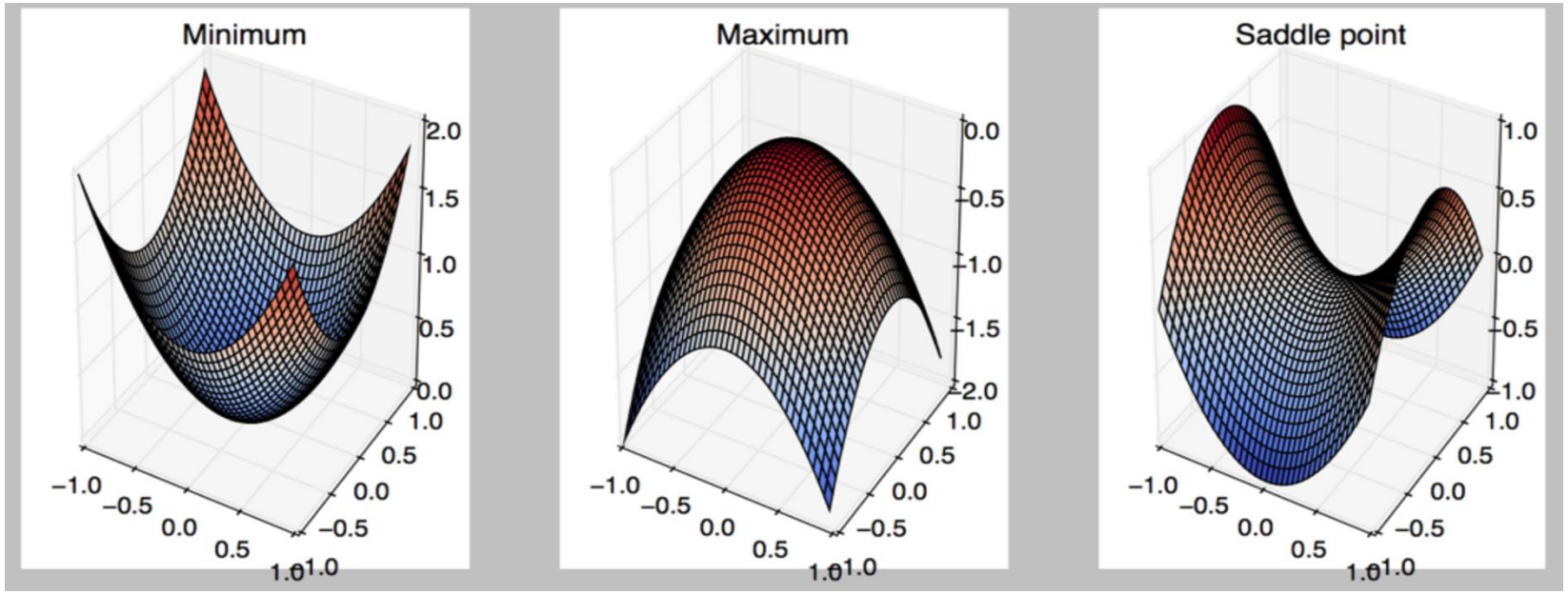
Optimizers

Local Minima



Critical Points

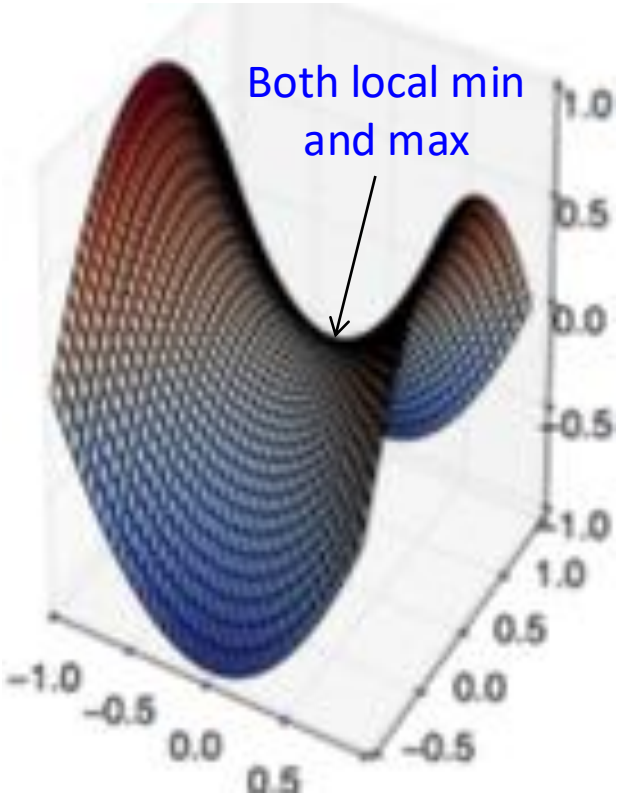
- Points with zero gradient
- 2nd-derivate (Hessian) determines curvature



Local Minima

- Old view: local minima is major problem in neural network training
- Recent view:
  - For sufficiently large neural networks, most local minima incur low cost
  - Not important to find true global minimum

Saddle Points

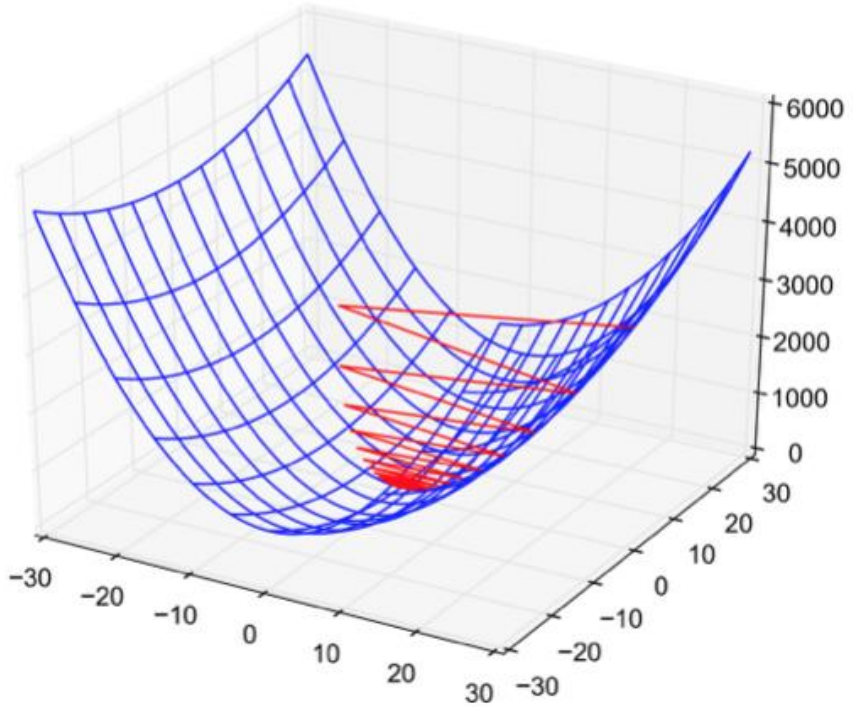


- Recent studies indicate that in high dim, saddle points are more likely than local min
- Gradient can be very small near saddle points

Poor Conditioning

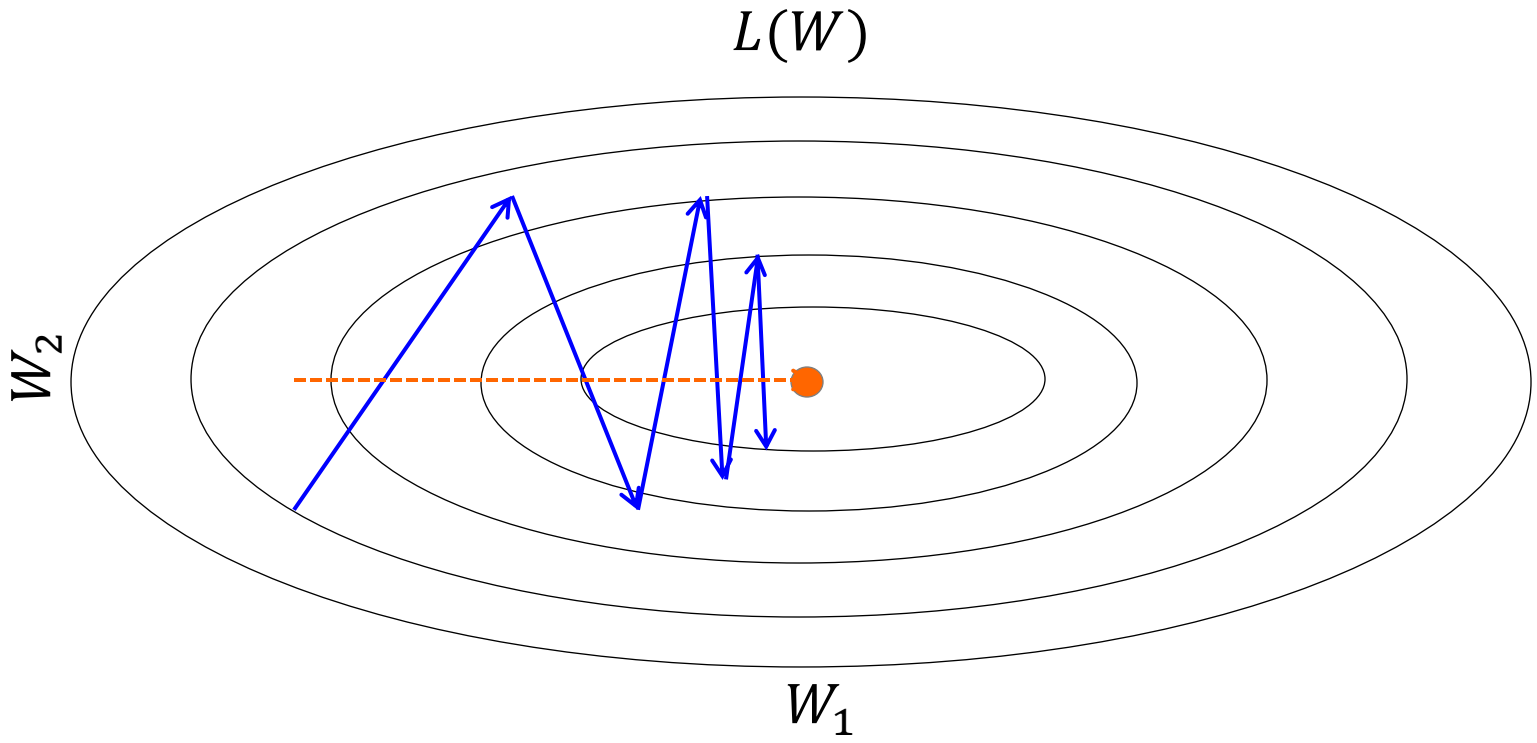
- Poorly conditioned Hessian matrix
  - High curvature: small steps leads to huge increase
- Learning is slow despite strong gradients

Oscillations slow down progress



Momentum

- Oscillations because updates do not exploit curvature information



- Average gradient presents faster path to optimal: vertical components cancel out

Momentum

f is the Neural Network

- Old gradient descent:

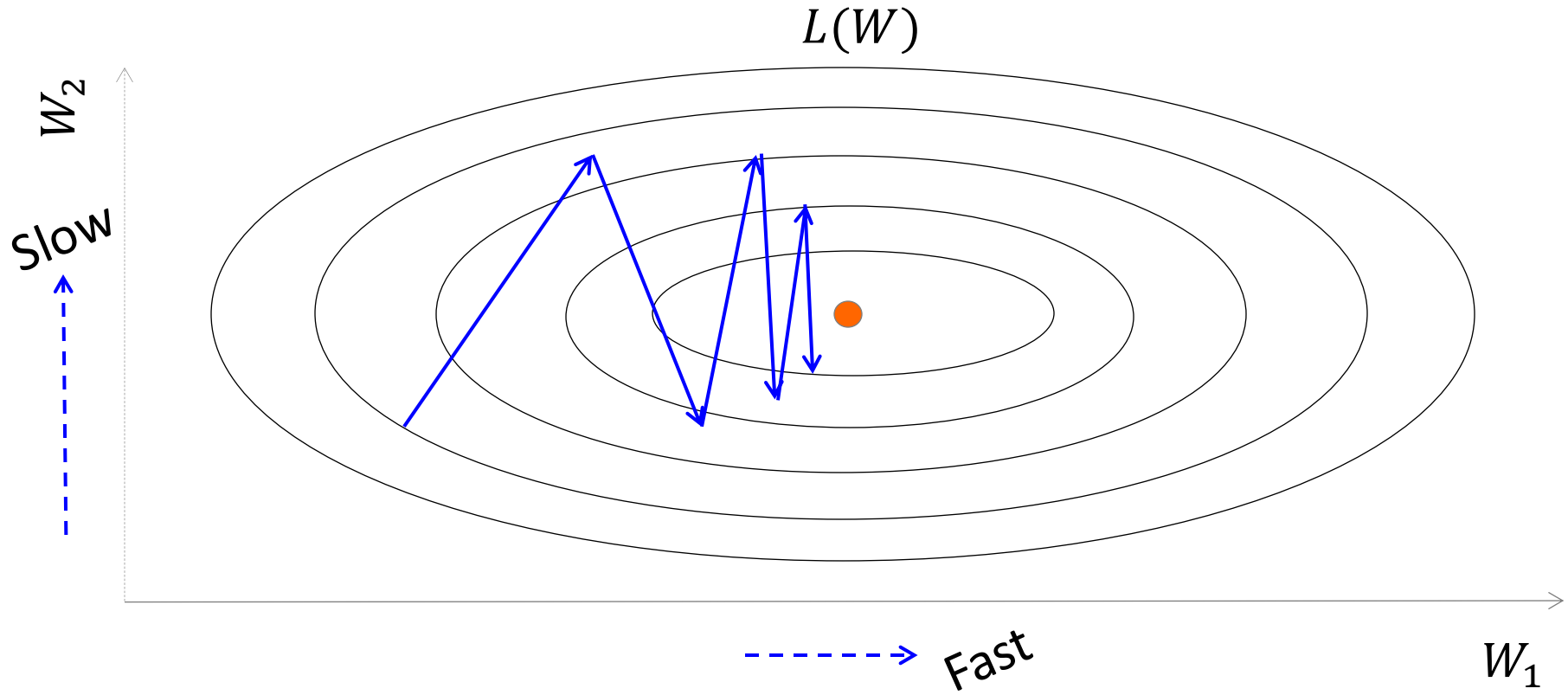
$$g = \frac{1}{m}\sum_i \nabla_W L(f(x_i; W), y_i) \qquad W^* = W - \eta g$$

- New gradient descent with momentum:

$$v = \alpha v + (1 - \alpha)\, g \qquad W^* = W - \eta v$$

$\alpha \in [0,1)$ controls how quickly effect of past gradients decay

Adaptive Learning Rates



- Oscillations along vertical direction
  - Learning must be slower along parameter 2
- Use a different learning rate for each parameter?

AdaGrad

- Accumulate squared gradients:

$$r_i = r_i + g_i^2$$

g is the gradient

- Update each parameter:

$$W_i = W_i - \frac{\epsilon}{\delta + \sqrt{r_i}} g_i$$

Inversely proportional to cumulative gradient

δ is a small number, making sure this does not become too large

- Greater progress along gently sloped directions

Other Adaptive Learning Rate Methods

- RMSProp
  - Use exponentially weighted average for gradient accumulation

$$r_i = r\, r_i + (1 - r) g_i^2$$

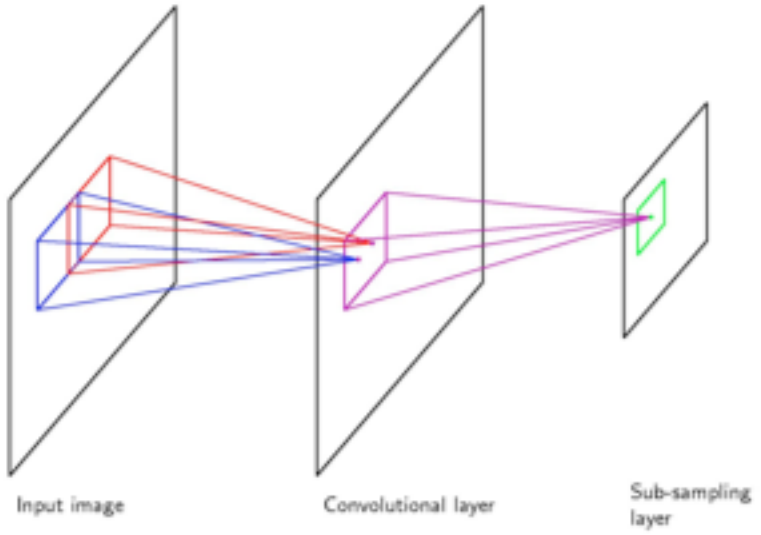
$$W_i = W_i - \frac{\epsilon}{\delta + \sqrt{r_i}} g_i$$

- Adam
  - RMSProp + Momentum
  - Works well in practice, it is fairly robust to hyper-parameters

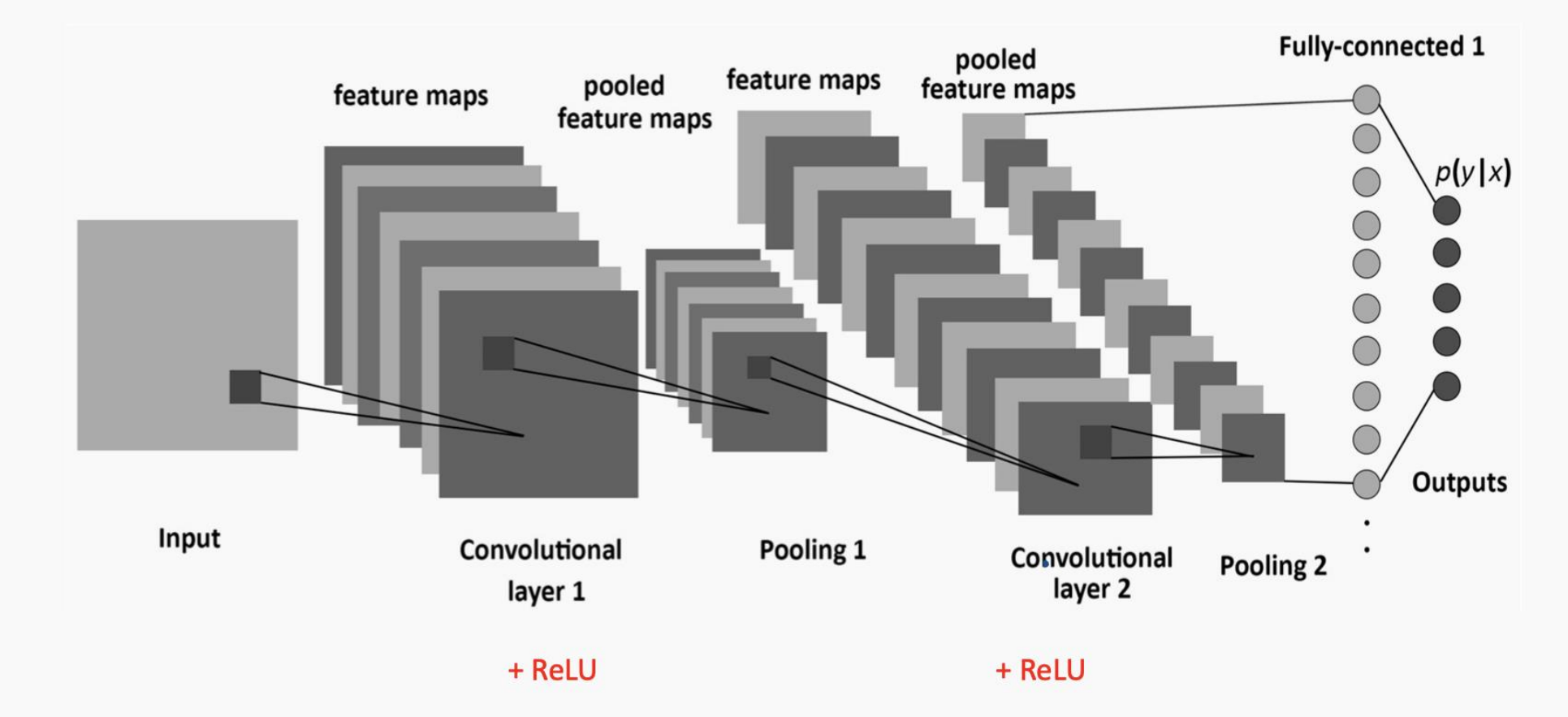
Other Neural Networks

Convolutional Network

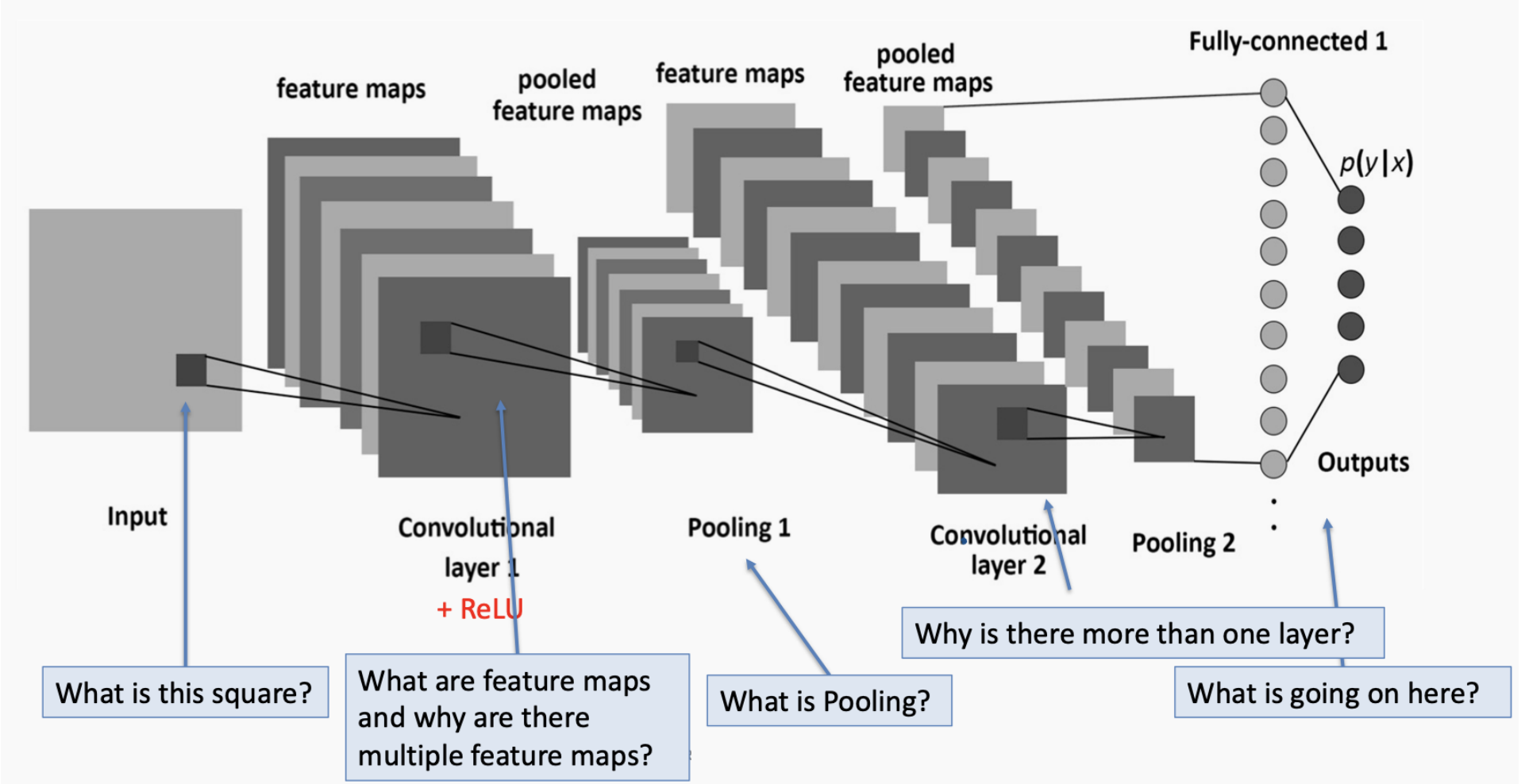
- The network is not fully connected.
- Different nodes are responsible for different regions of the image.
- This allows for robustness to transformations.



A Convolutional Network



A Convolutional Network

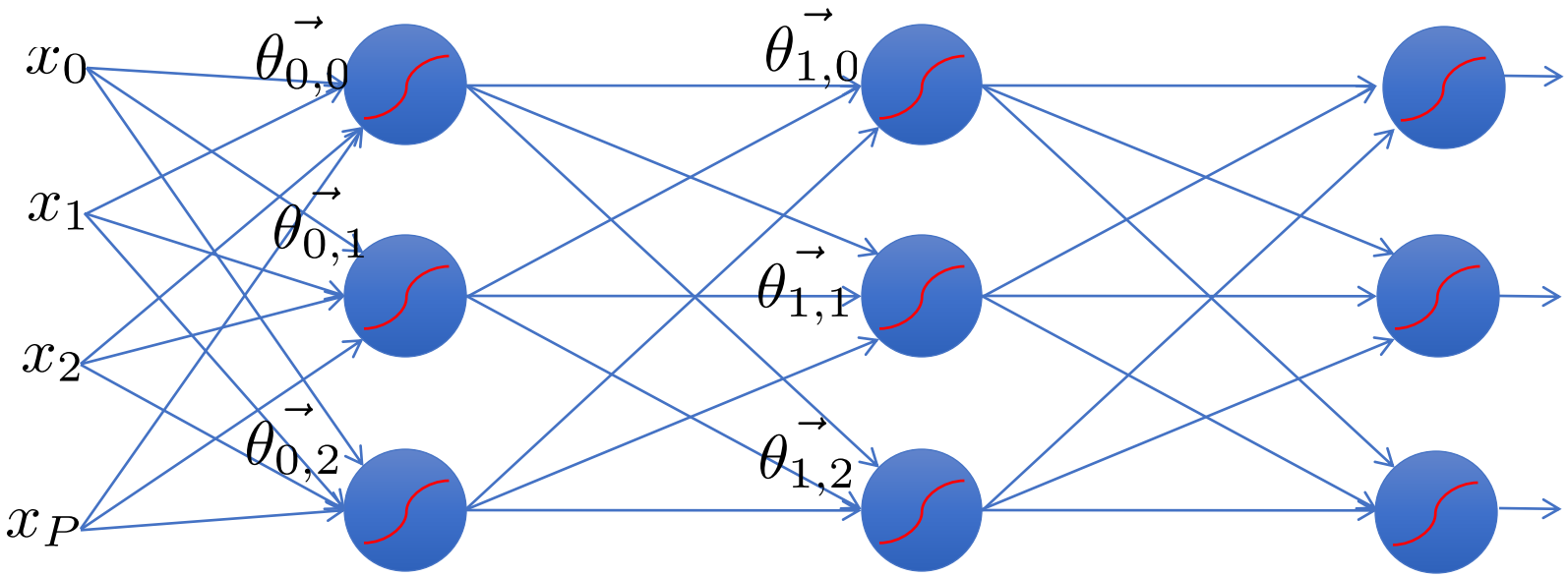


Other Neural Networks

- Multiple Outputs
- Skip Layer Network
- Recurrent Neural Networks

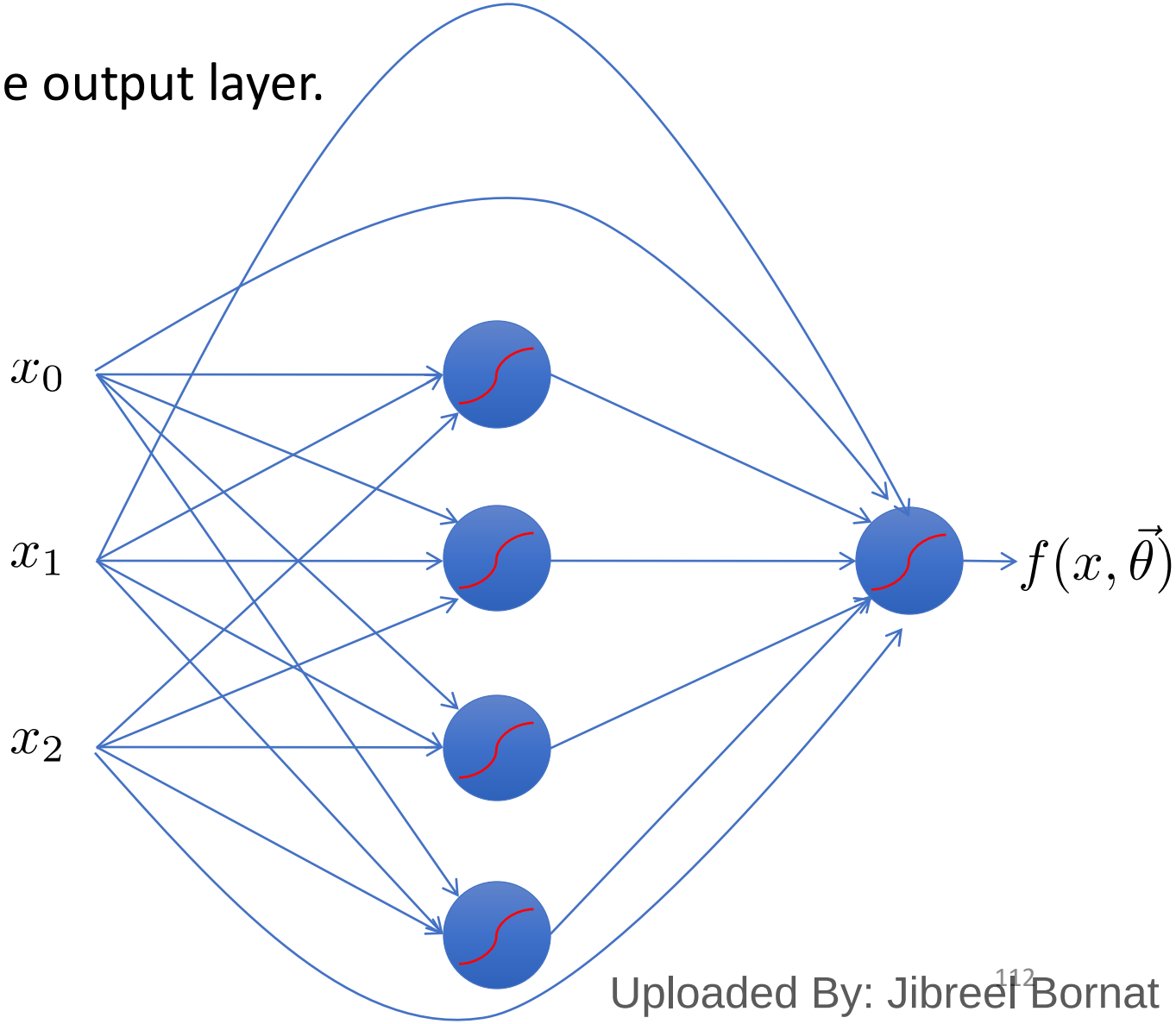
Multiple Outputs

- Used for N-way classification.
- Each Node in the output layer corresponds to a different class.
- No guarantee that the sum of the output vector will equal 1.



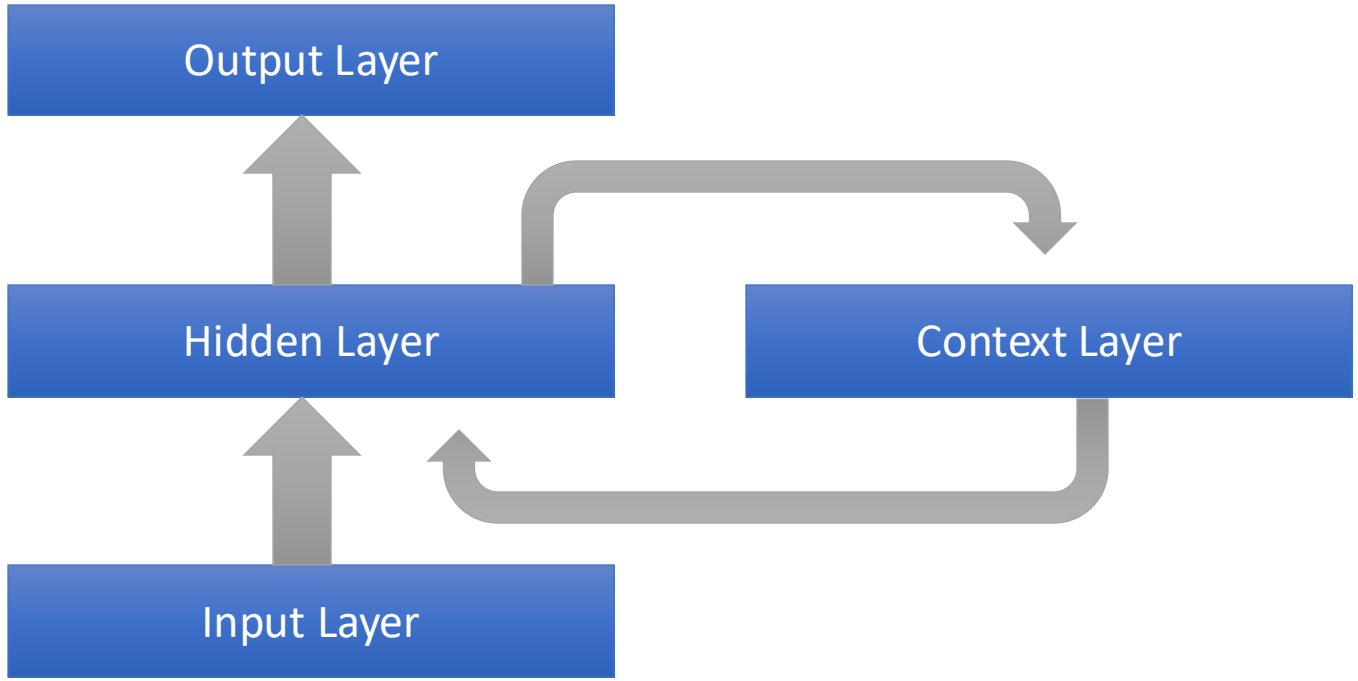
Skip Layer Network

- Input nodes are also sent directly to the output layer.



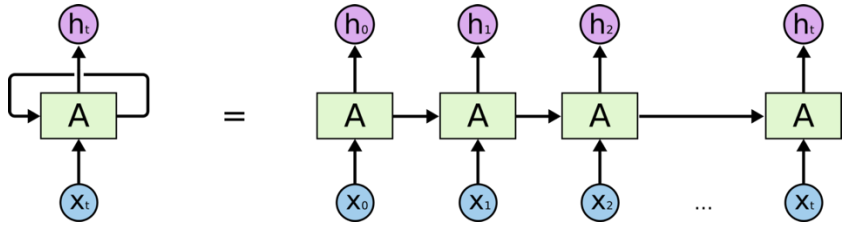
Recurrent Neural Networks

- Output or hidden layer information is stored in a context or memory layer.

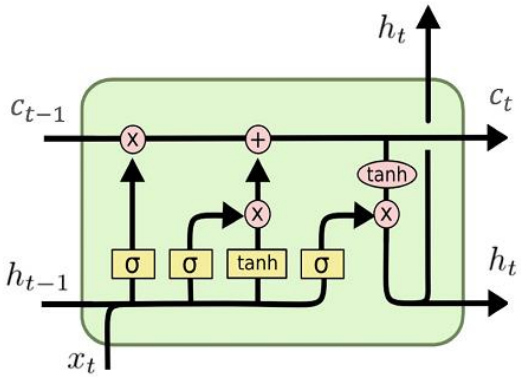


Recurrent Neural Networks

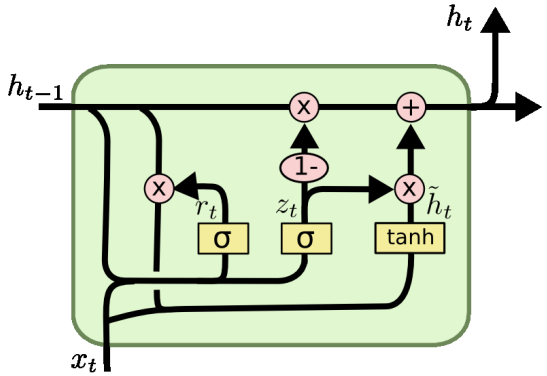
- Vanilla RNN



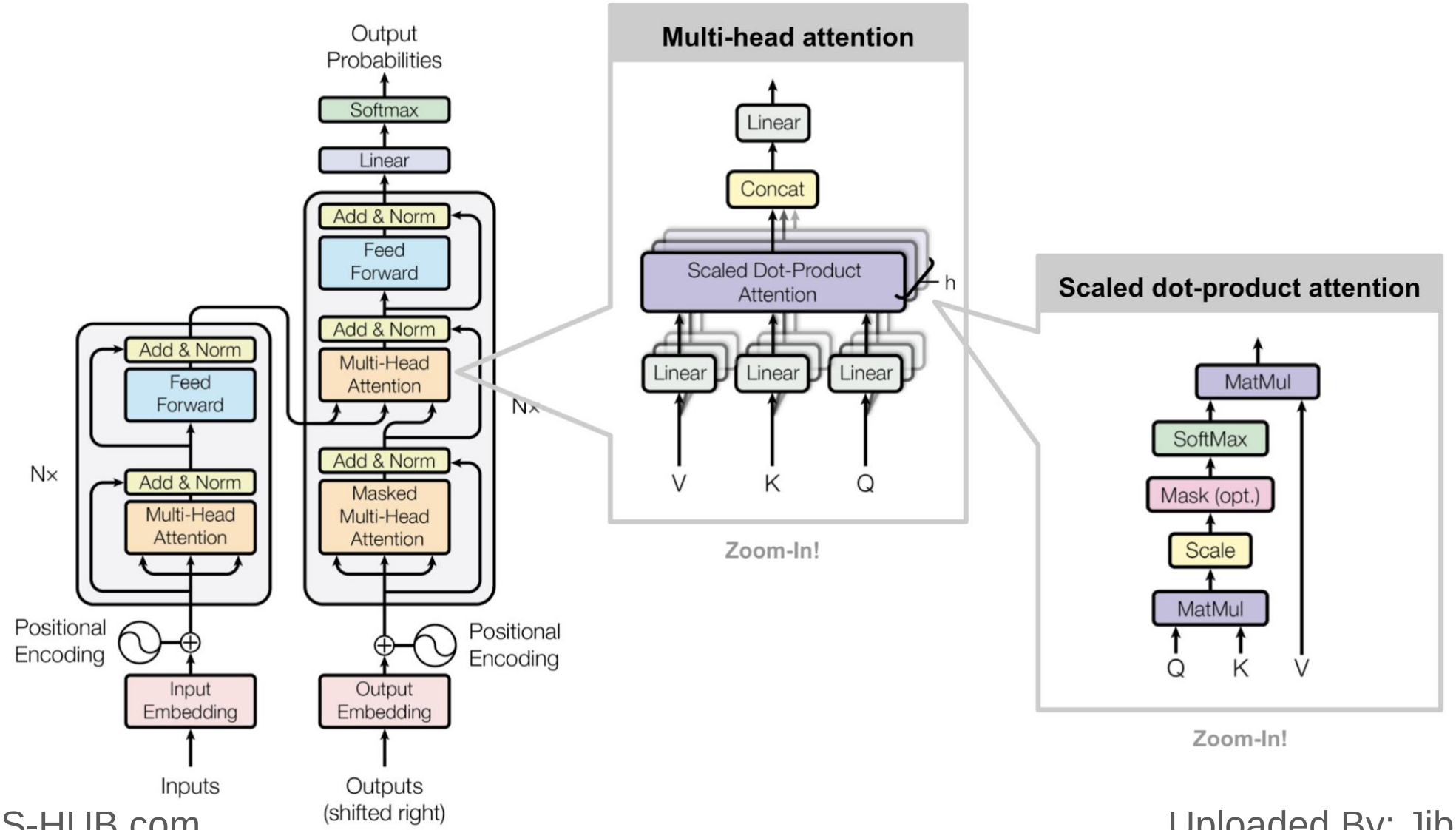
- Long-Short term memory (LSTM).



- Gated Recurrent Units (GRU).



Transformer Family

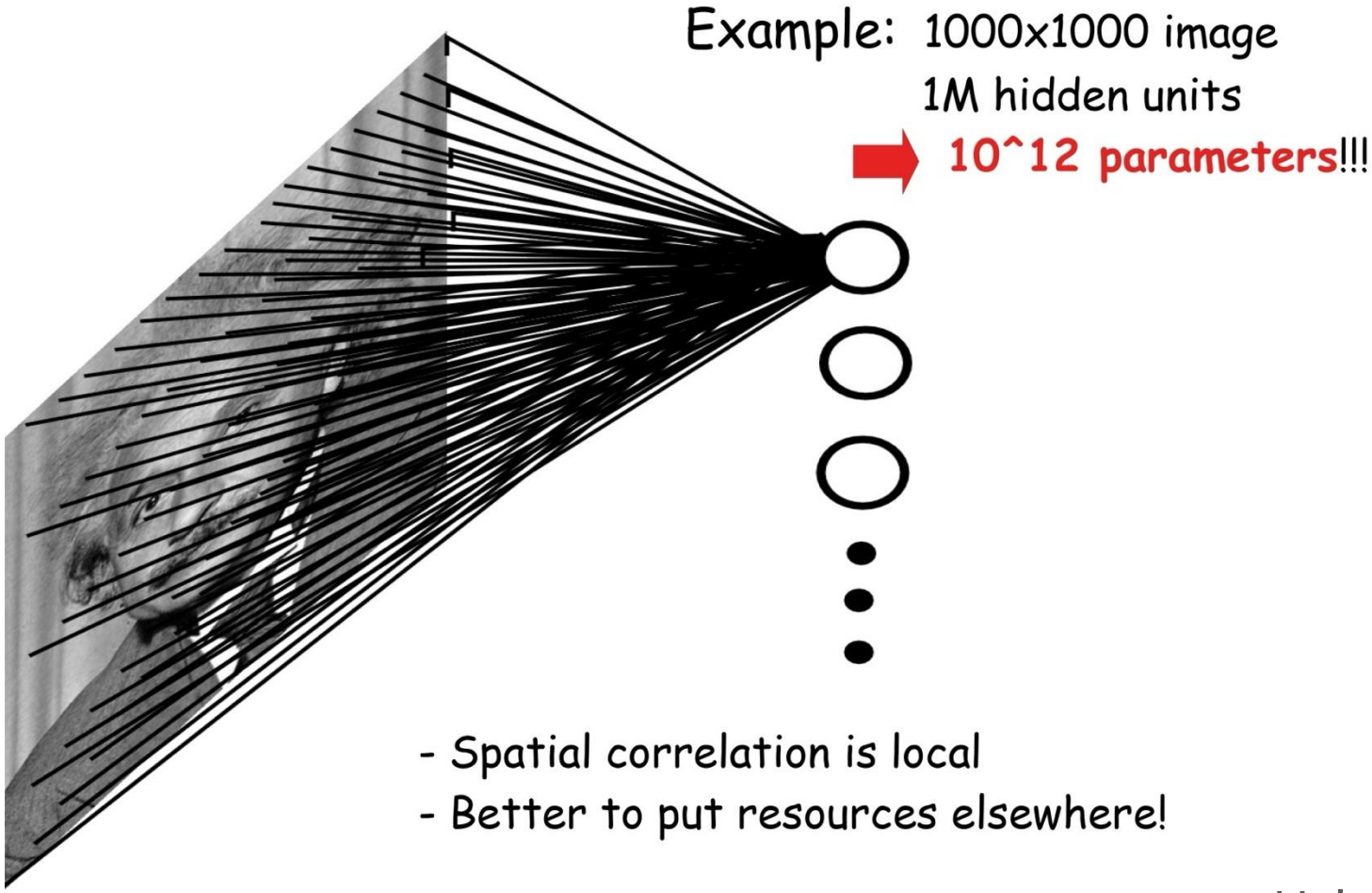


Back to our Object Recognition System

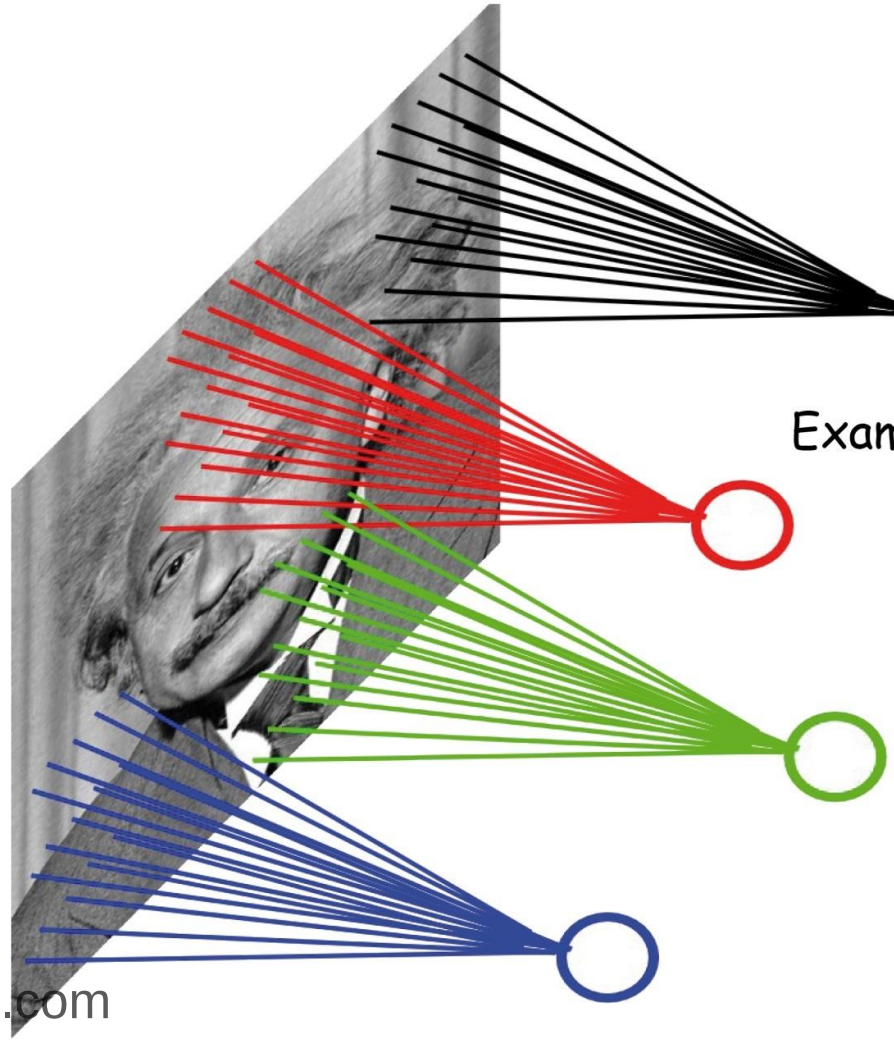
Building an Object Recognition System

- Everything becomes adaptive.
- No distiction between feature extractor and classifier.
- Big non-linear system trained from raw pixels to labels.

Fully Connected Neural Net



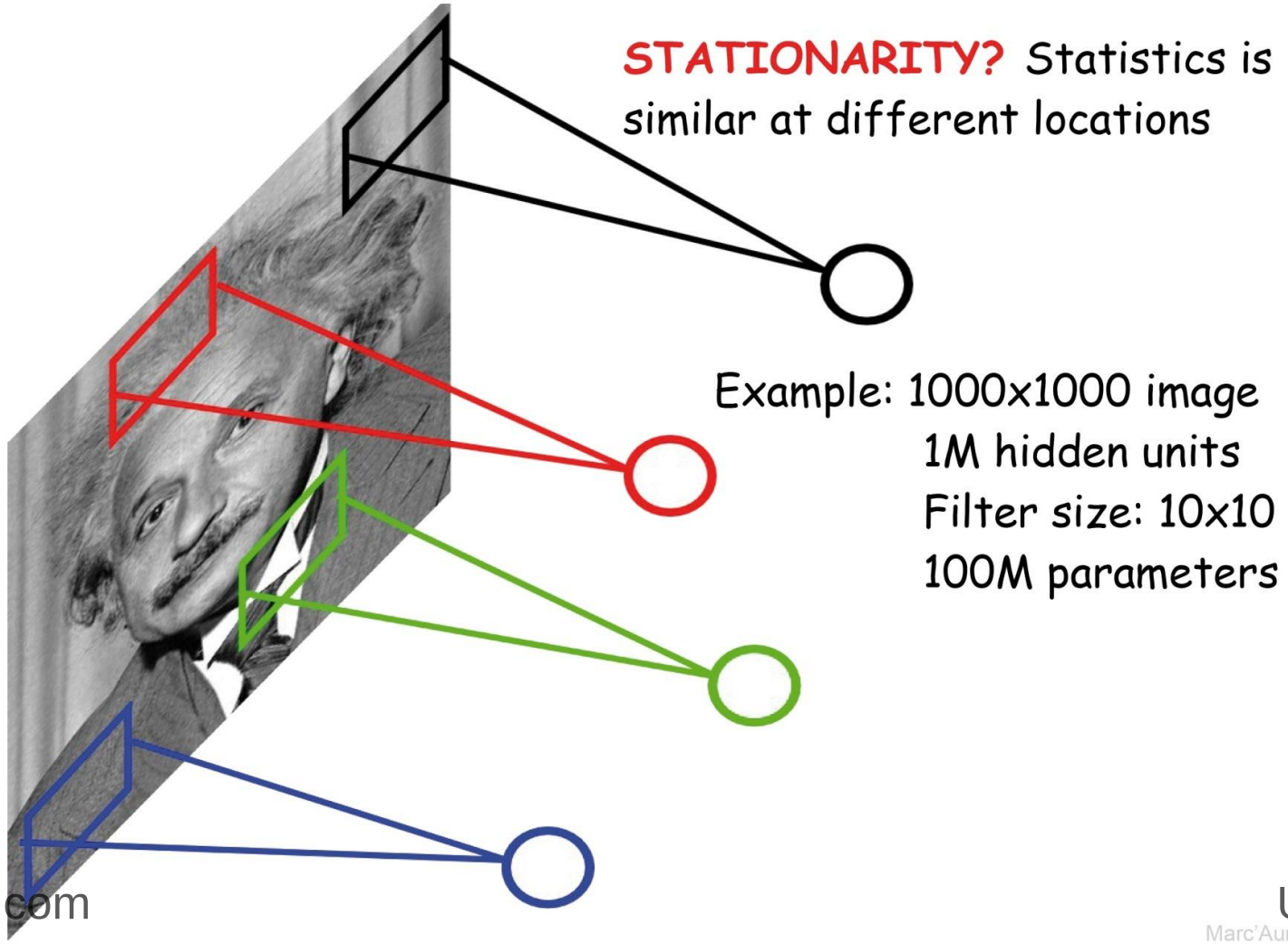
Locally Connected Neural Net



Example: 1000x1000 image
1M hidden units
Filter size: 10x10
100M parameters

Filter/Kernel/Receptive field:
input patch which the hidden unit is connected to.

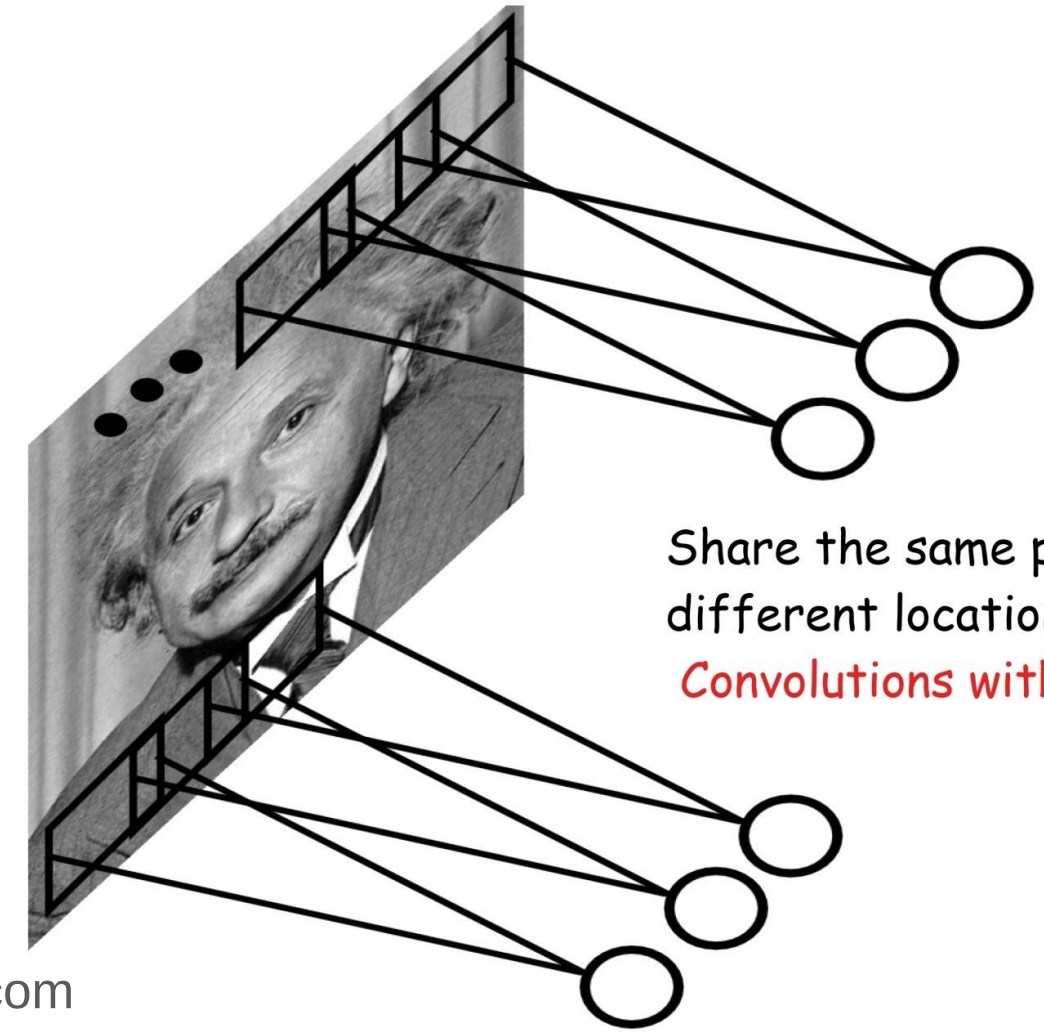
Locally Connected Neural Net



STATIONARITY? Statistics is similar at different locations

Example: 1000x1000 image
1M hidden units
Filter size: 10x10
100M parameters

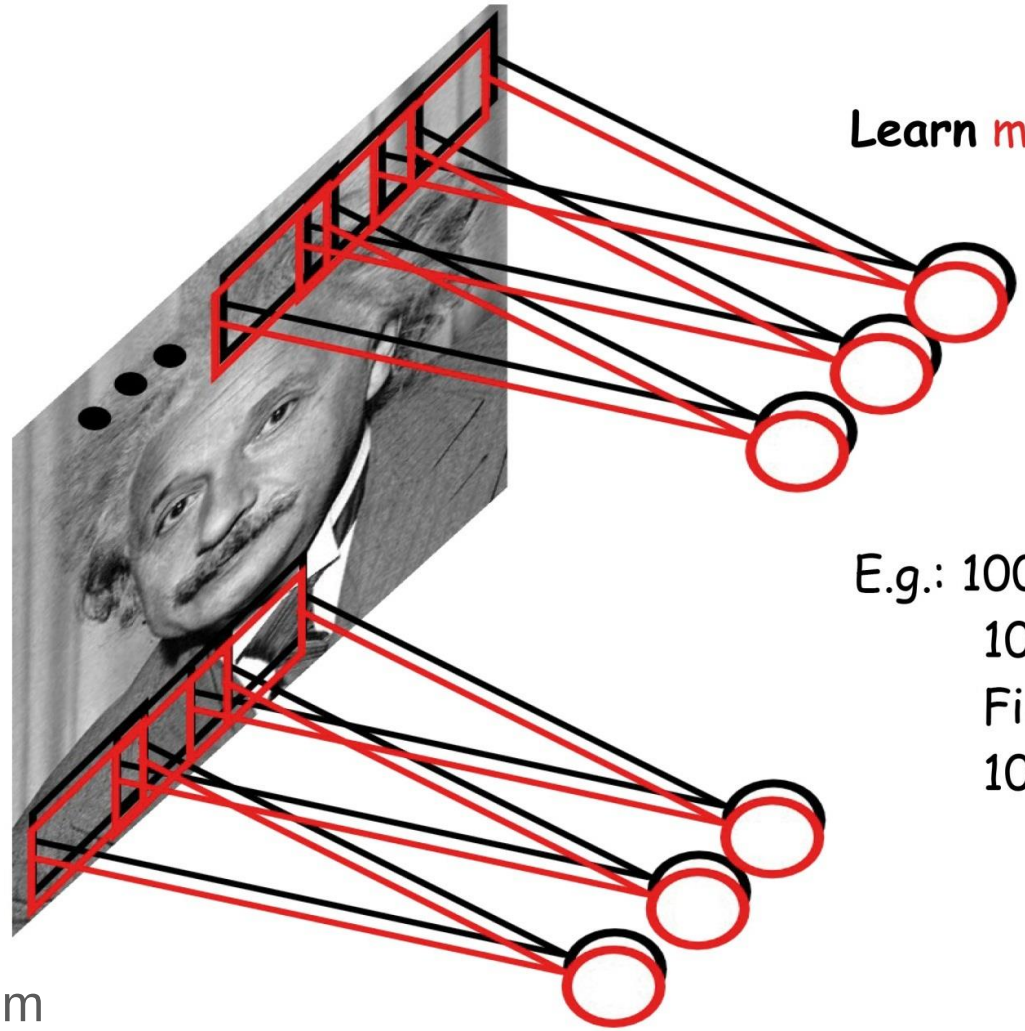
Convolutional Net



Share the same parameters across different locations:

Convolutions with learned kernels

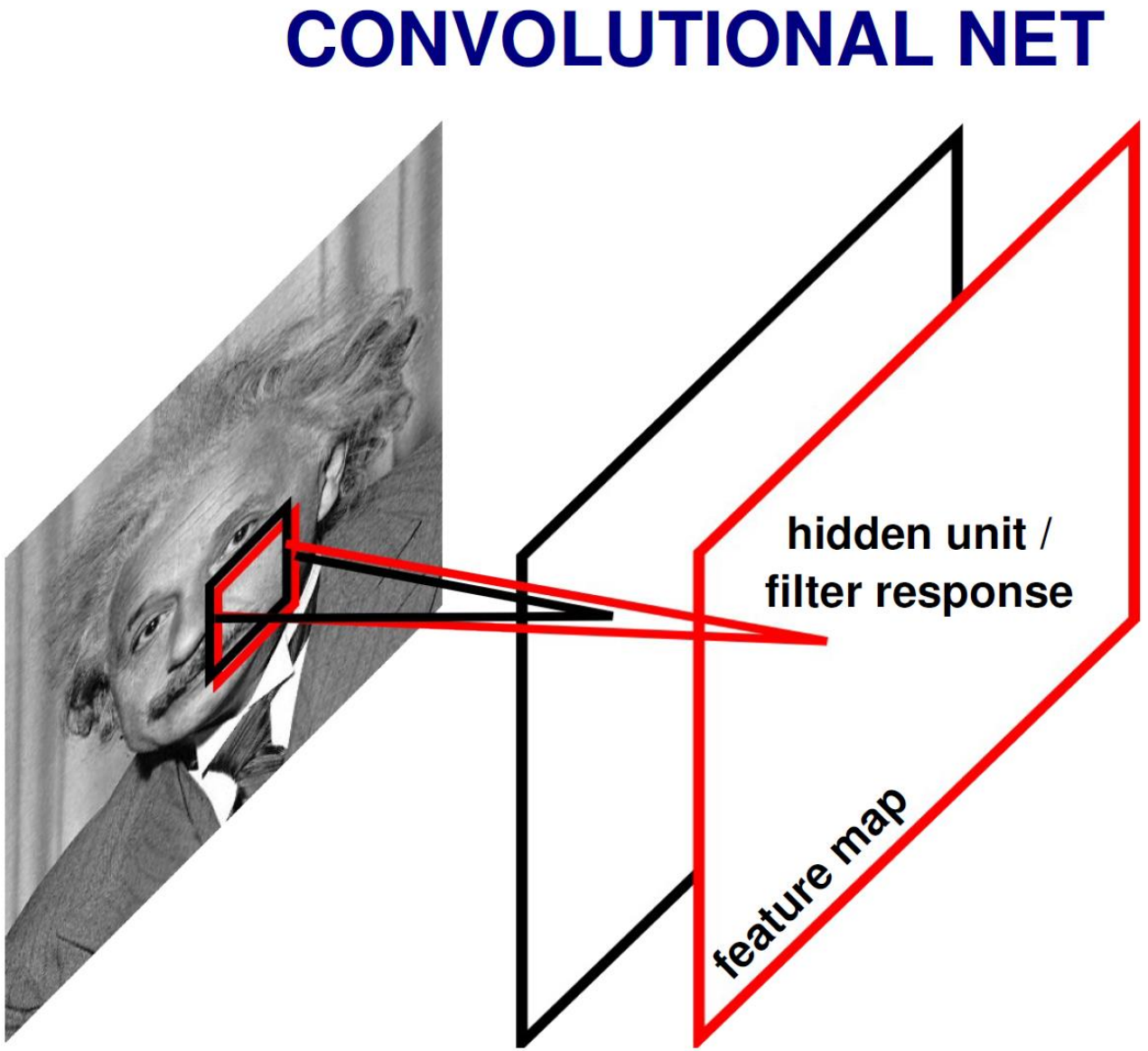
Convolutional Net



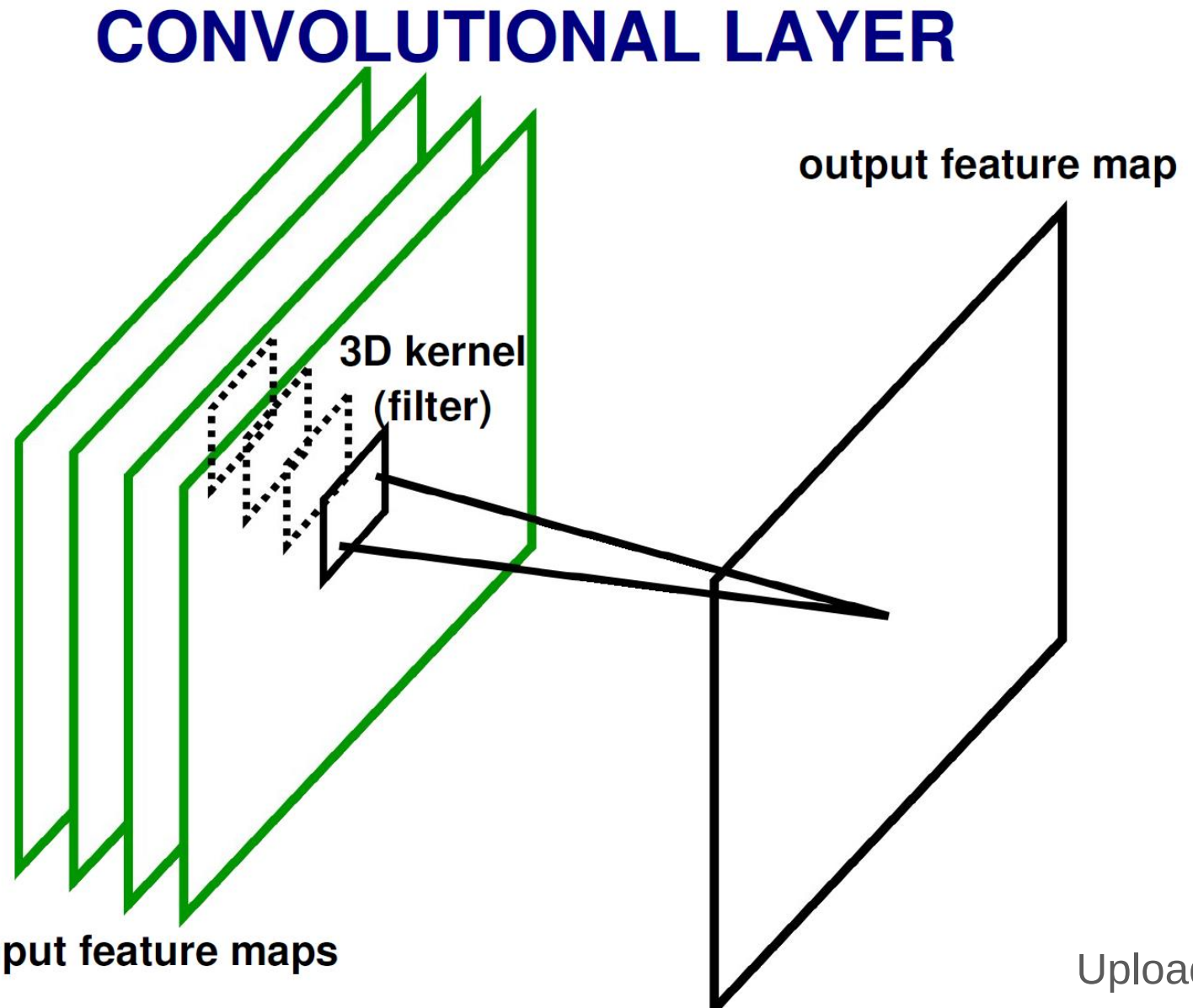
Learn multiple filters.

E.g.: 1000x1000 image
100 Filters
Filter size: 10x10
10K parameters

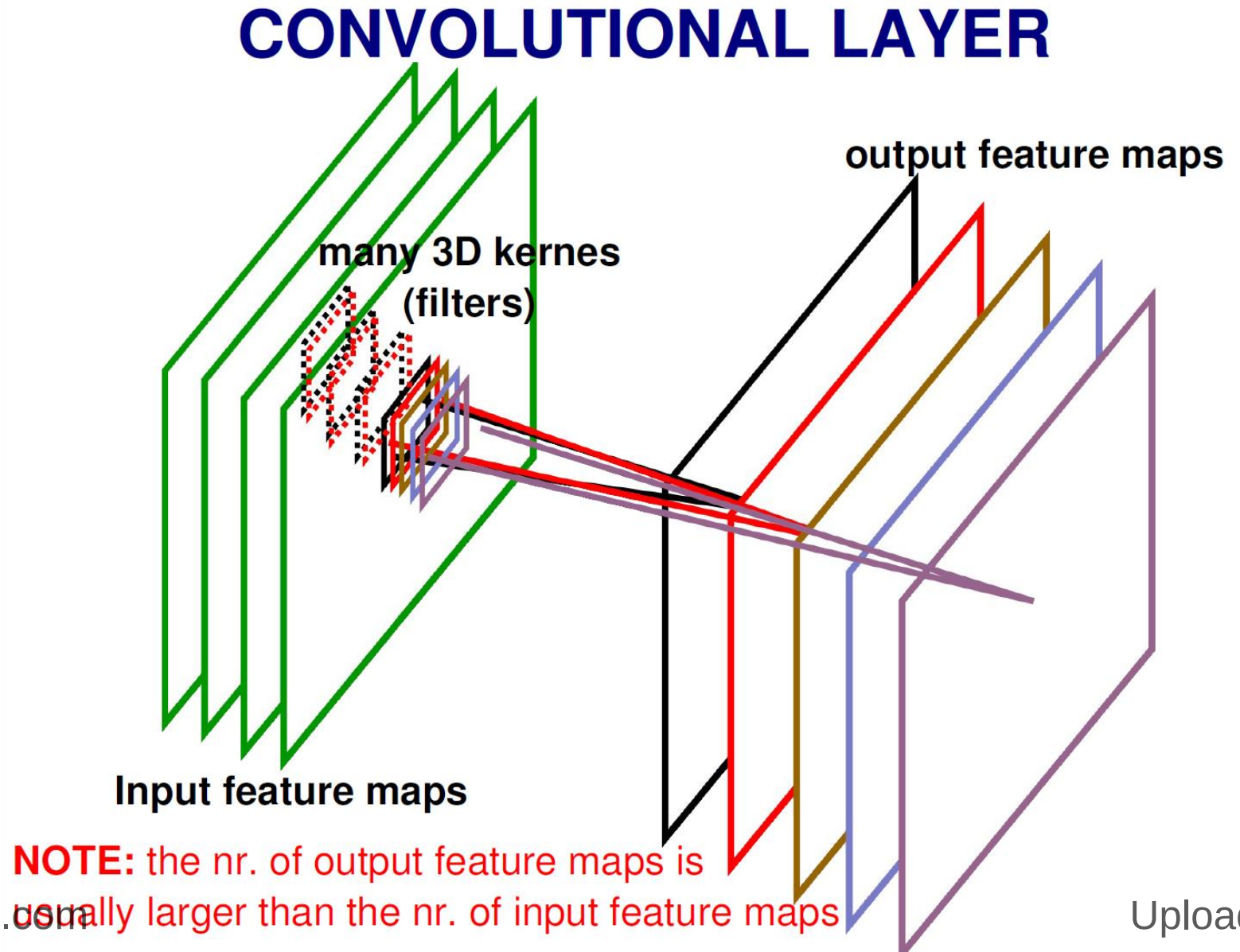
Convolutional Net



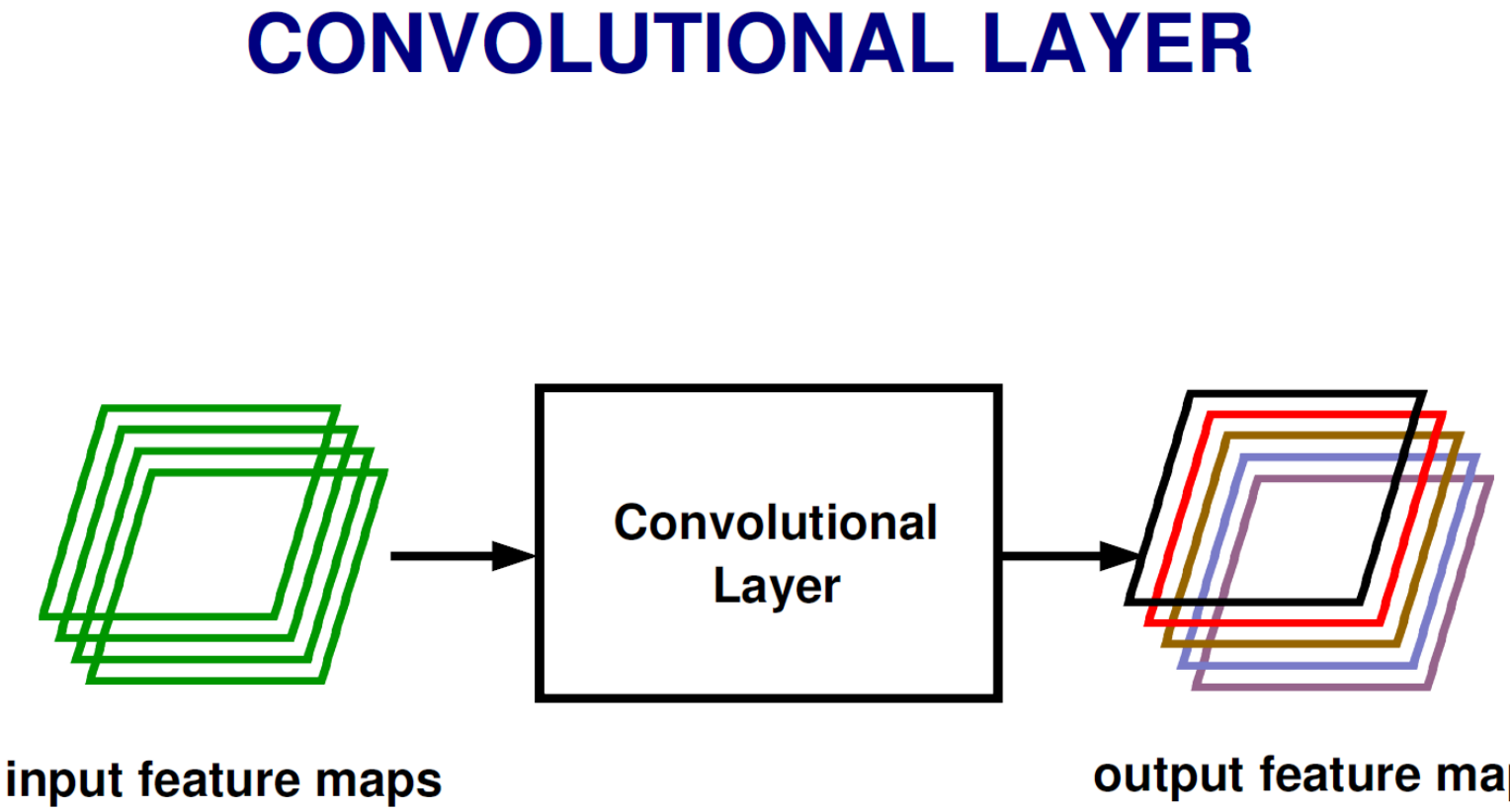
Convolutional Net



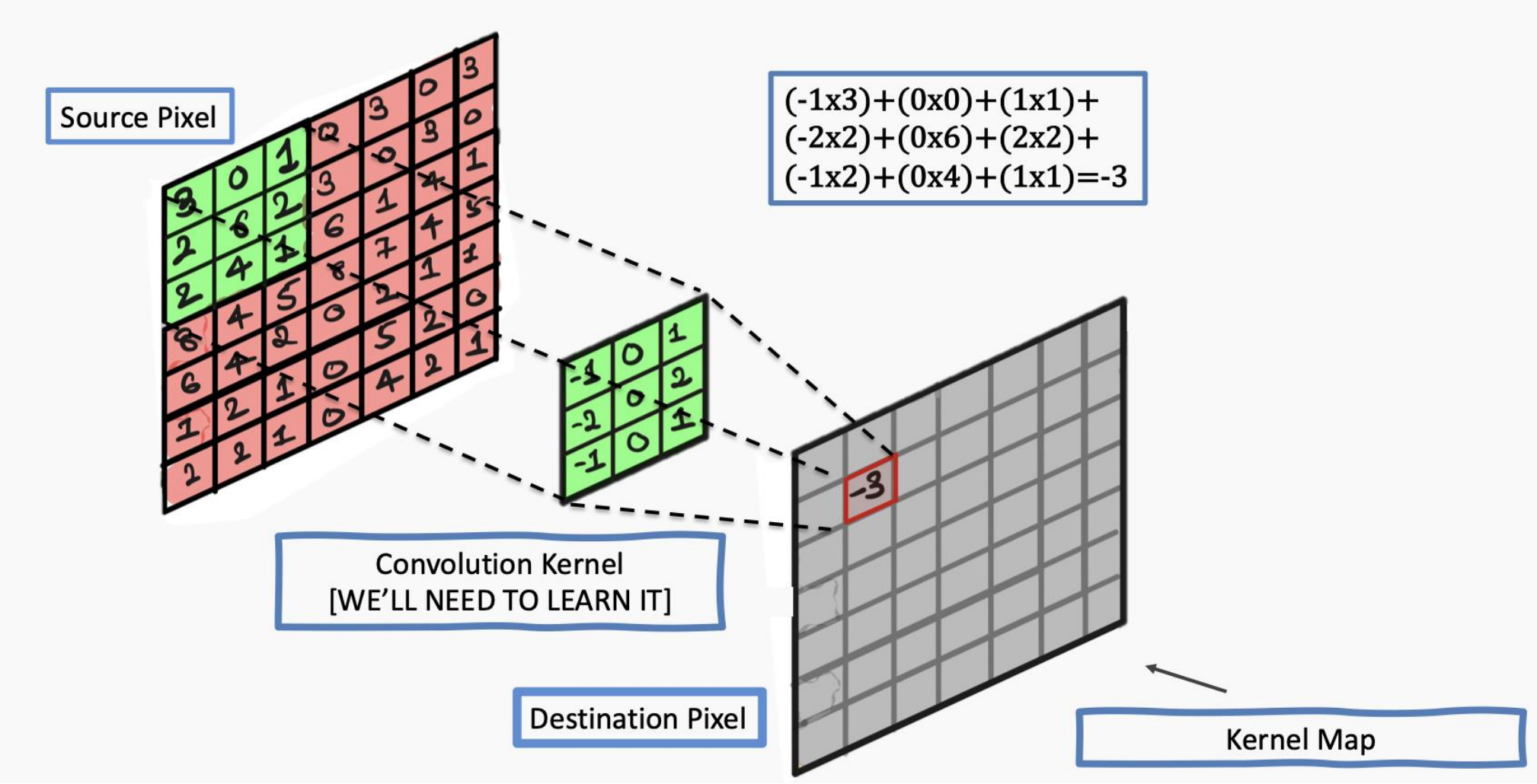
Convolutional Net



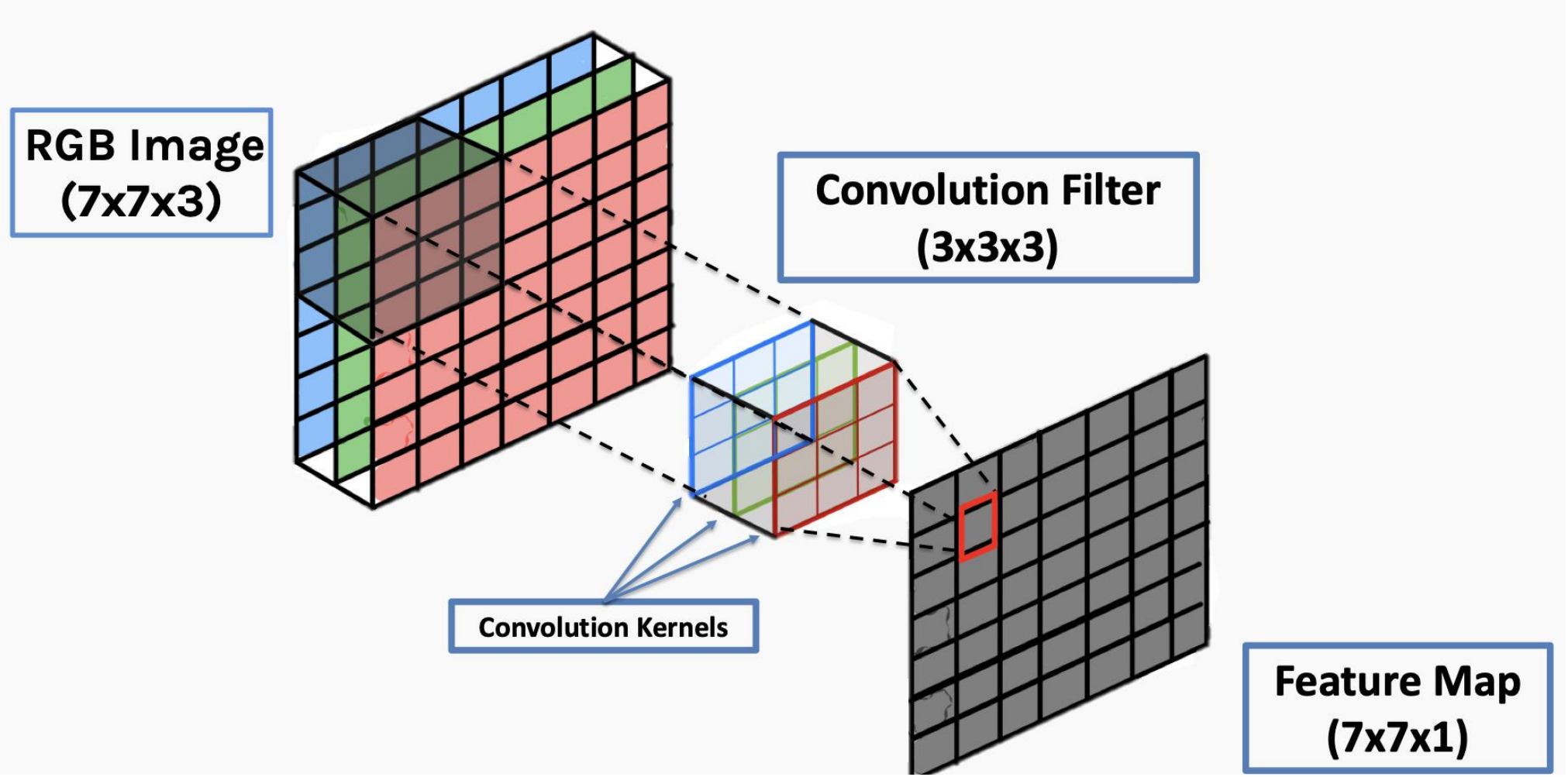
Convolutional Net



“Convolution” Operation

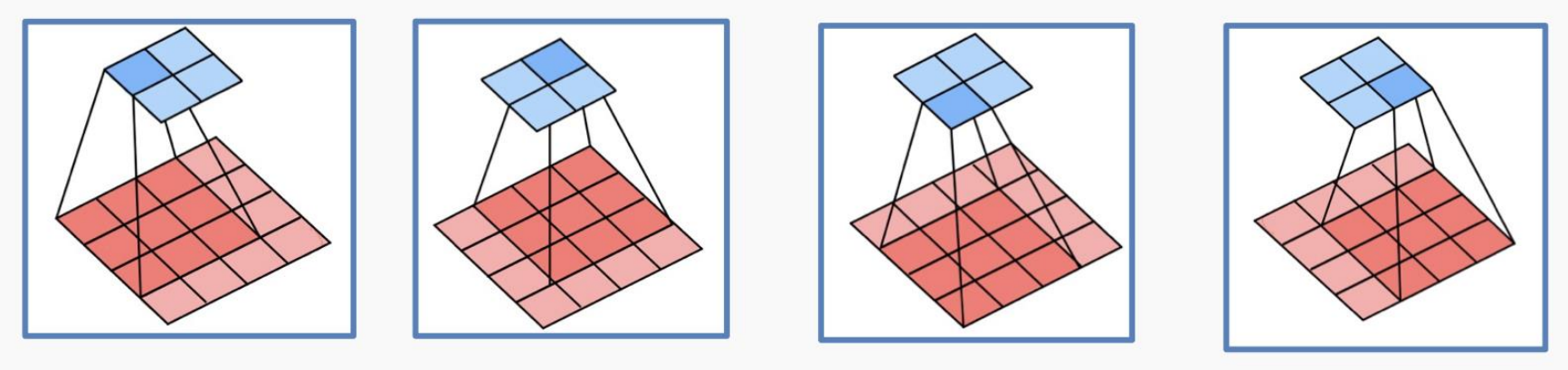


“Convolution” Operation



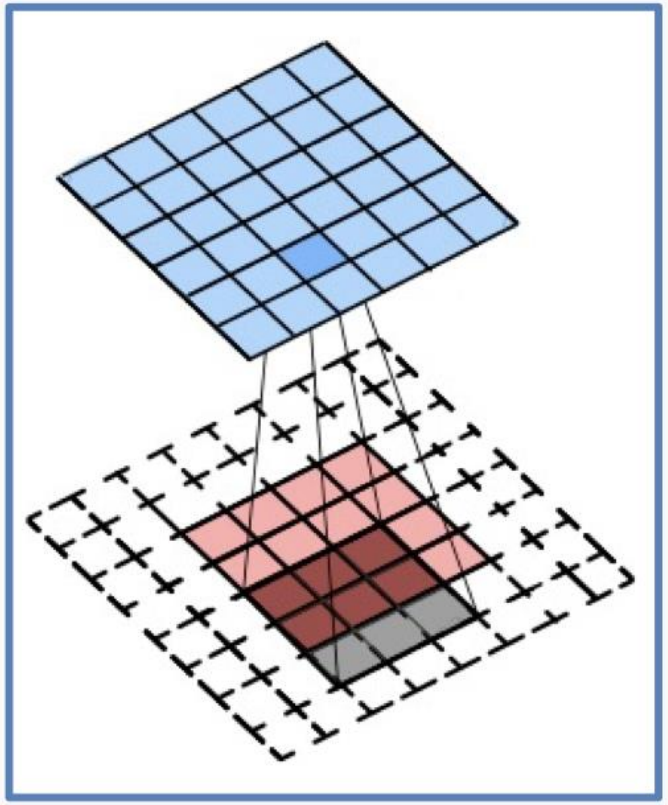
Convolutions – what happens at the edges?

If we apply convolutions on a normal image, the result will be down-sampled by an amount depending on the size of the filter.

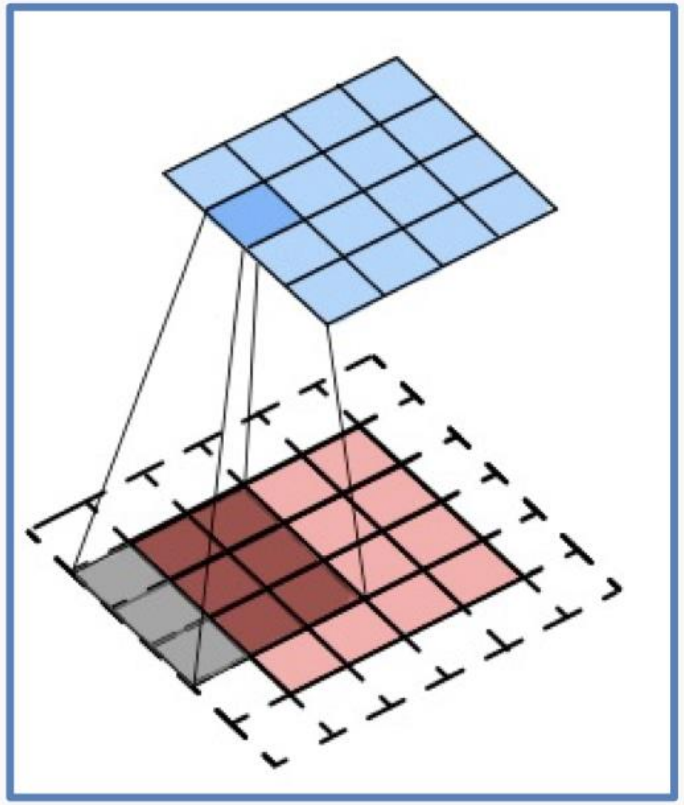


We can avoid this by padding the edges in different ways.

Padding



Full padding. Introduces zeros such that all pixels are visited the same number of times by the filter. Increases size of output.



Same padding. Ensures that the output has the same size as the input.

Stride

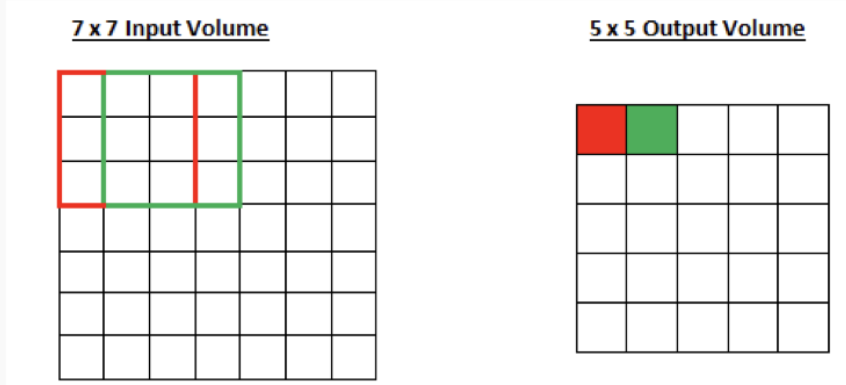
Stride controls how the filter convolves around the input volume.

The formula for calculating the output size is:

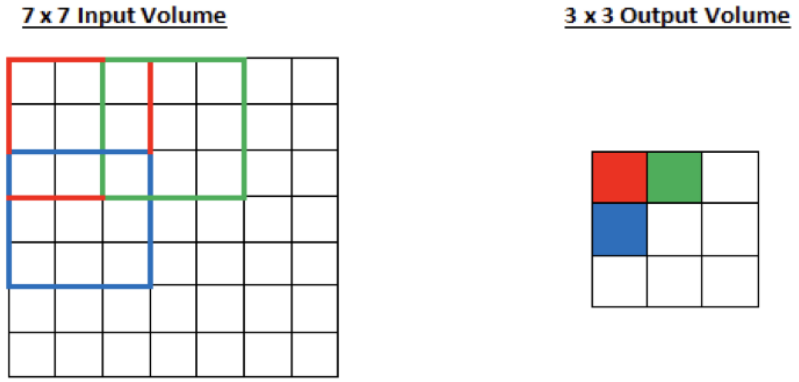
$$O = \frac{W - K + 2P}{S} + 1$$

Where O is output dim, W is the input dim, K is the filter size, P is padding and S the stride

Stride = 1



Stride = 2

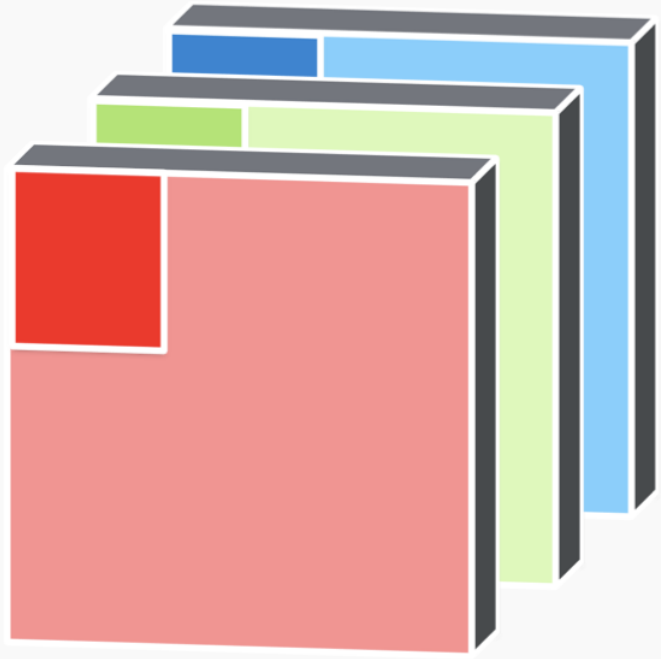


Example: A convolutional layer with one 3x3 filter that takes an 32x32 RGB image as input.

Input

size=32X32

channels=3



1 Filter

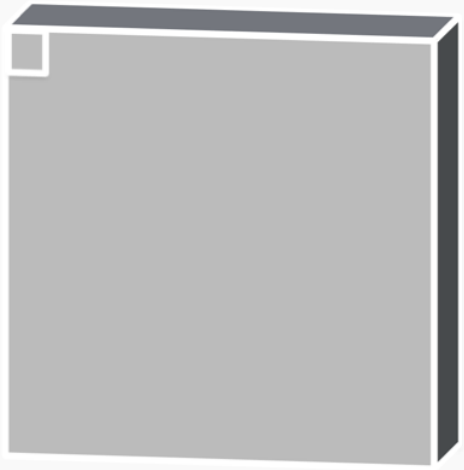
size=3x3X3

stride = 1

padding = same

Output

size=32X32



How many parameters does the layer have?

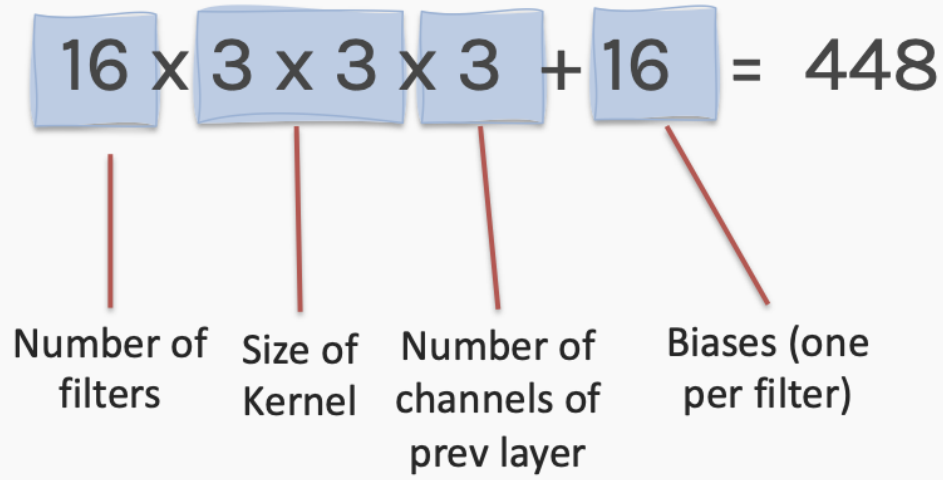
n_filters x filter_volume + biases = total number of params

1 x (3 x 3 x 3) + 1 = 28

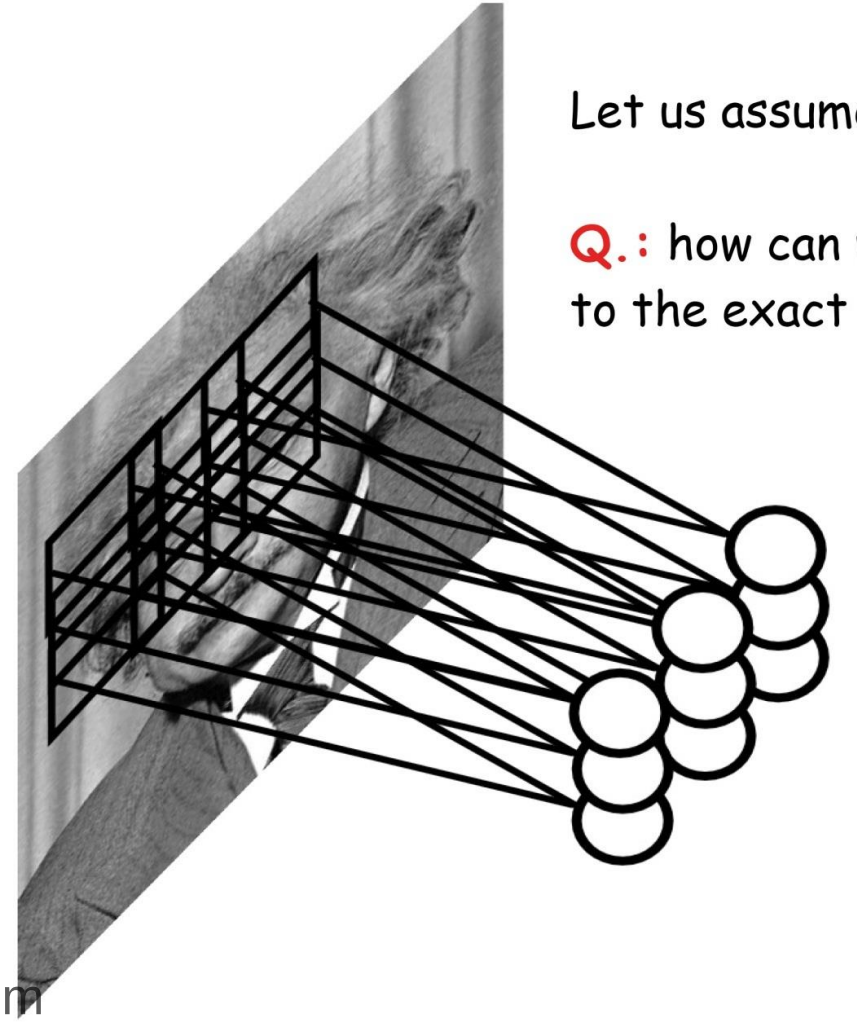
Example

A convolutional layer with **16 3x3** filters that takes **32x32** RGB image as input.

- How many parameters does the layer have?



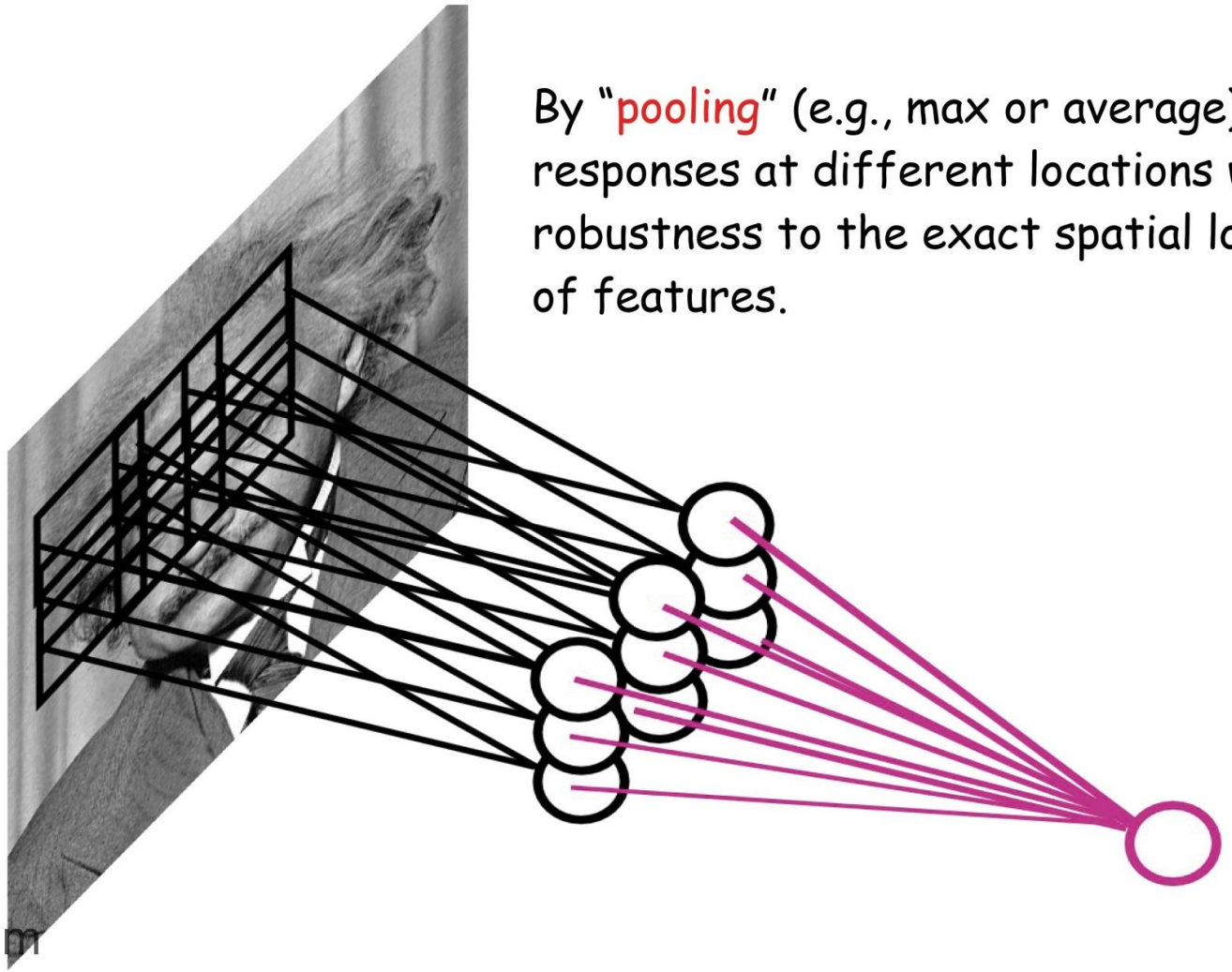
Convolutional Net



Let us assume filter is an "eye" detector.

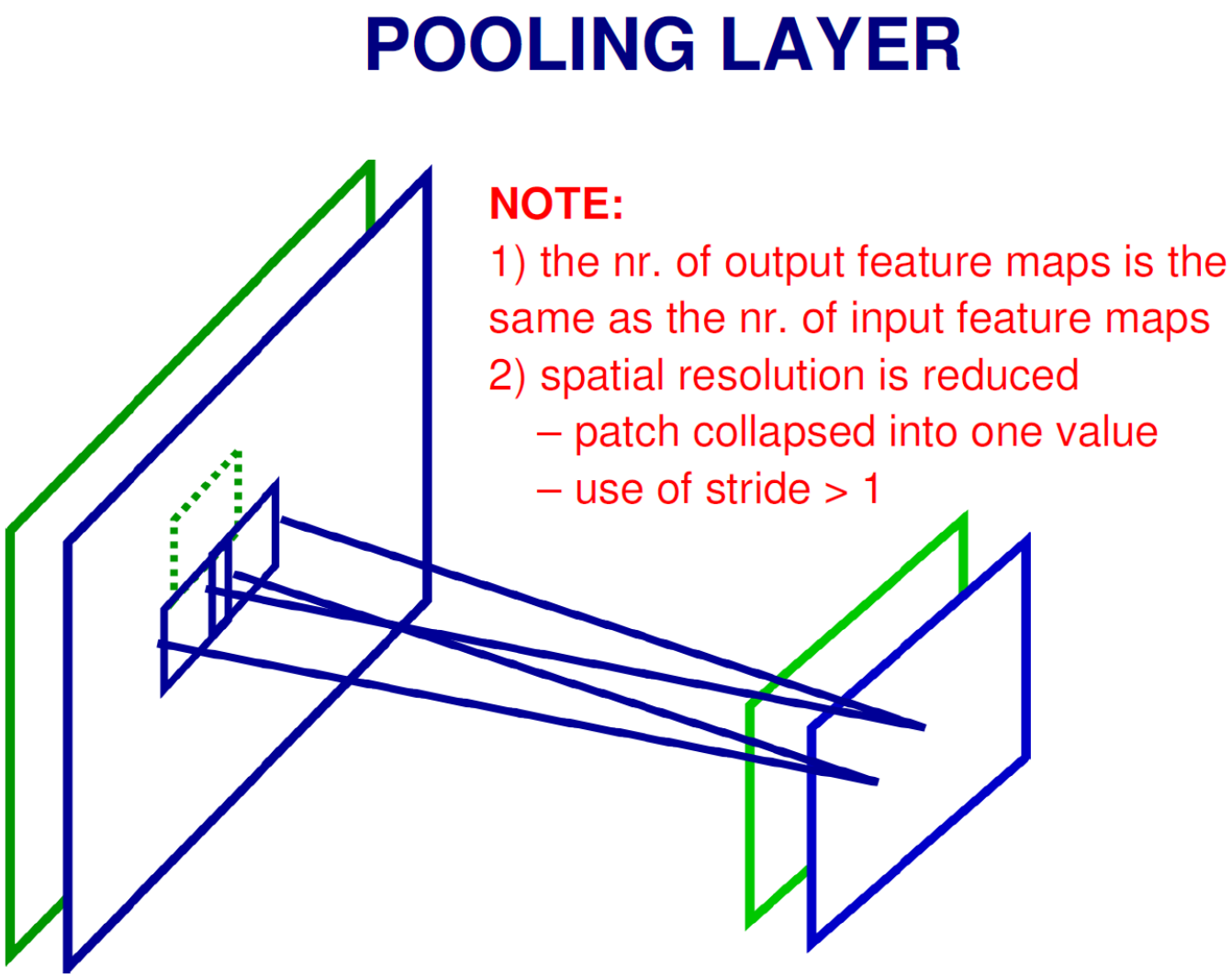
Q.: how can we make the detection robust to the exact location of the eye?

Convolutional Net

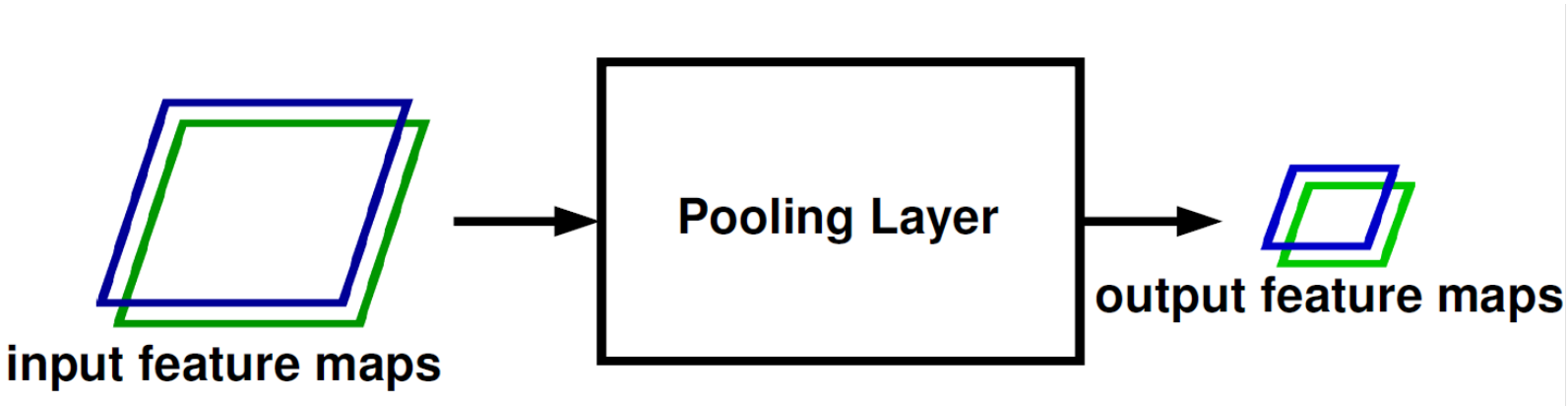


By "pooling" (e.g., max or average) filter responses at different locations we gain robustness to the exact spatial location of features.

Pooling

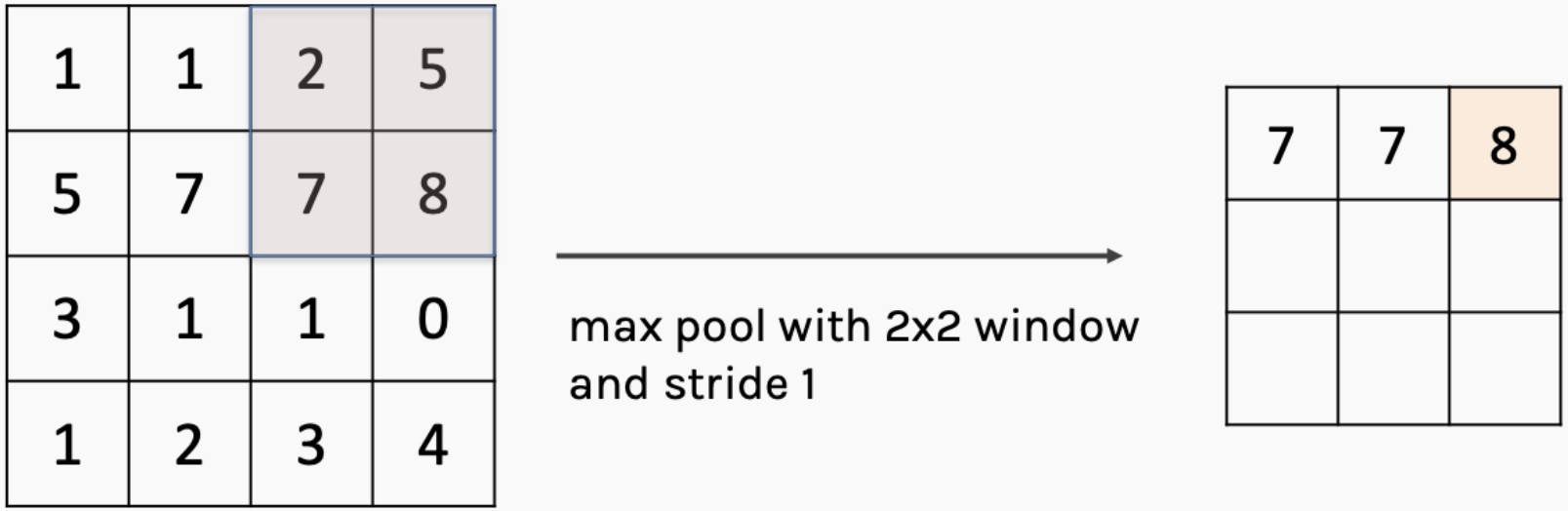


Pooling



Pooling

Example:



Pooling

Pooling involves selecting:

- A pooling operation, much like a filter to be applied to feature maps: max, mean, median.
- The size of the pooling operation.
- The stride.

The size of the pooling operator must be smaller than the size of the feature map; specifically, it is almost always 2×2 applied with a stride of 2 using max pooling.

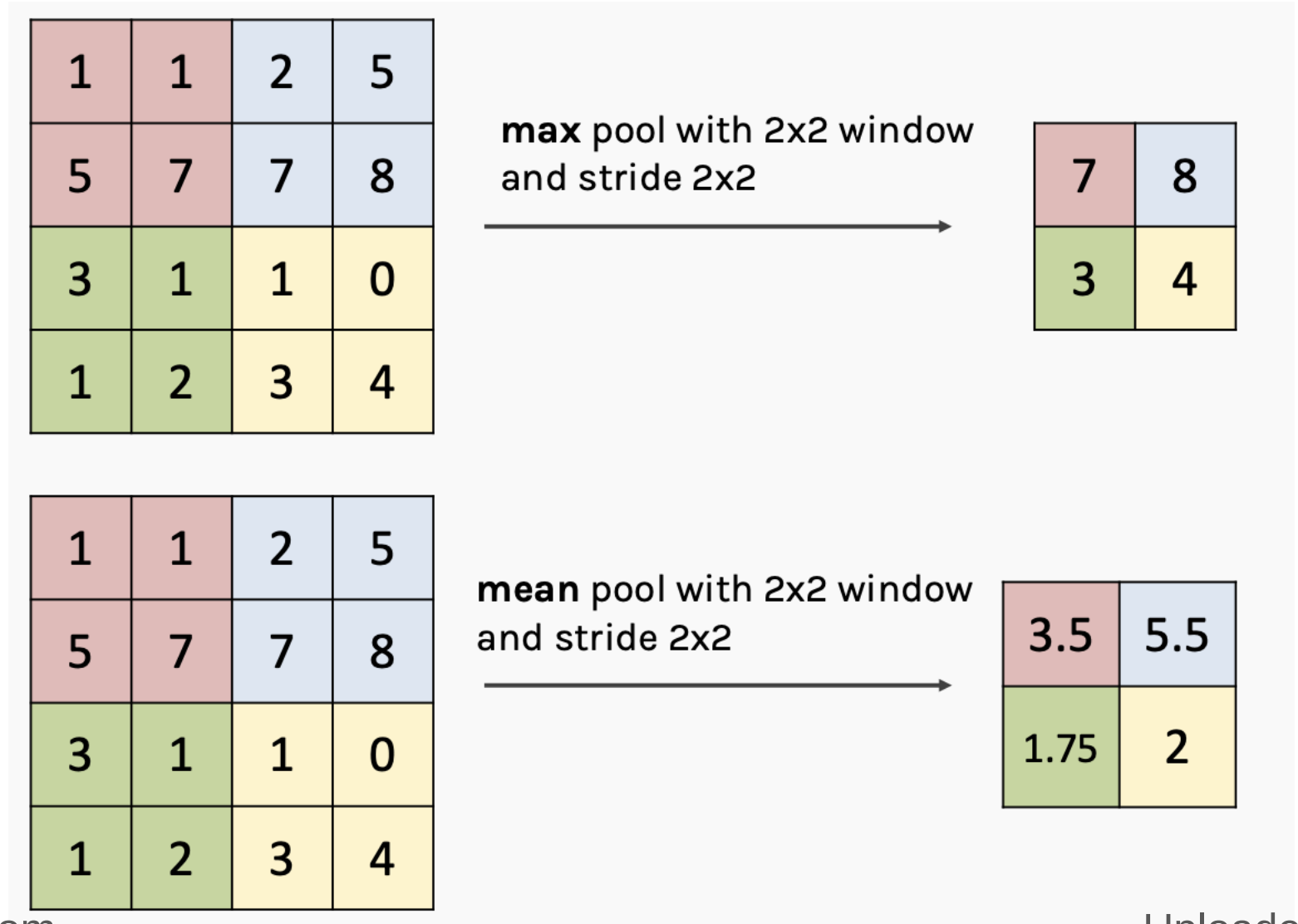
Invariant to small, "local transitions"

Face detection: enough to check the presence of eyes, not their precise location

Reduces input size of the final fully connected layers (more later)

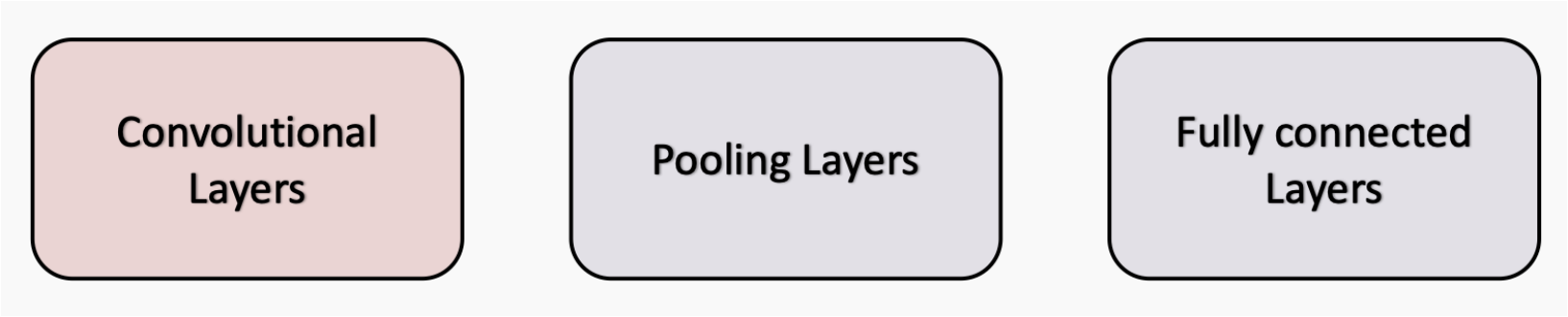
No learnable parameters

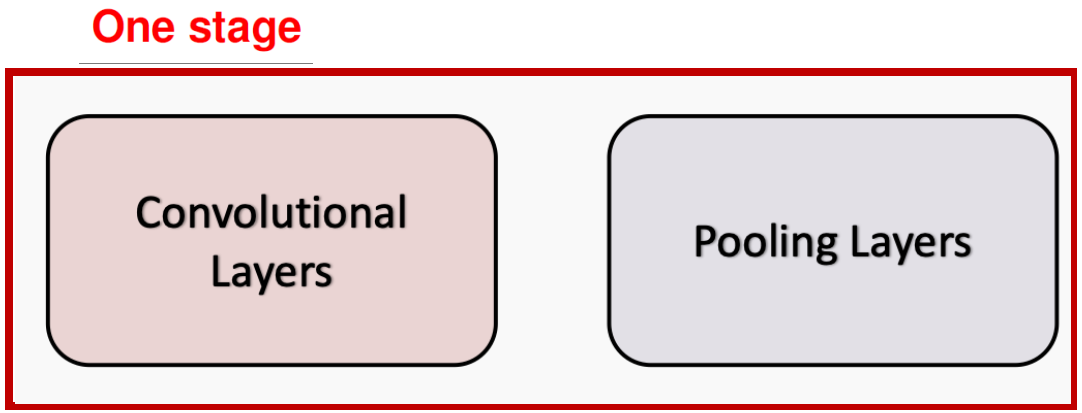
Pooling: example with stride 2x2

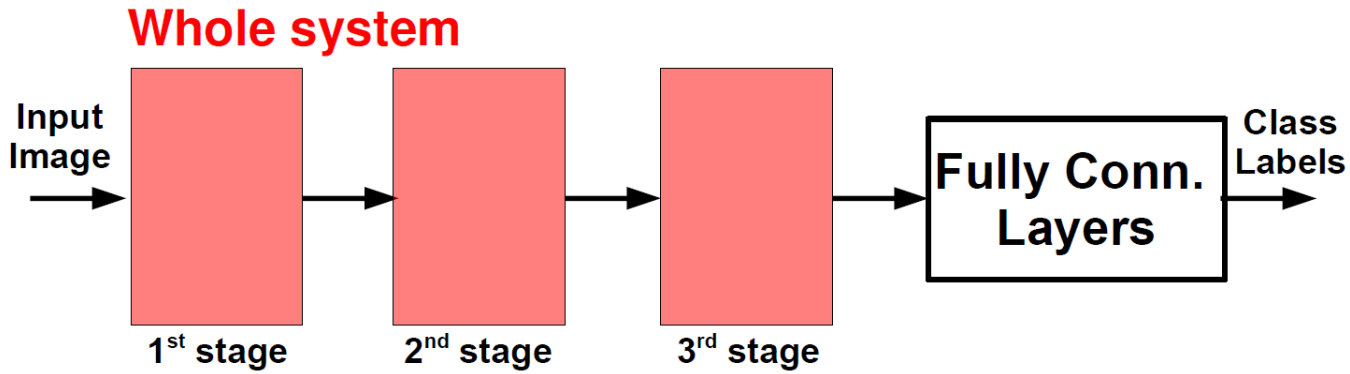


Building a CNN

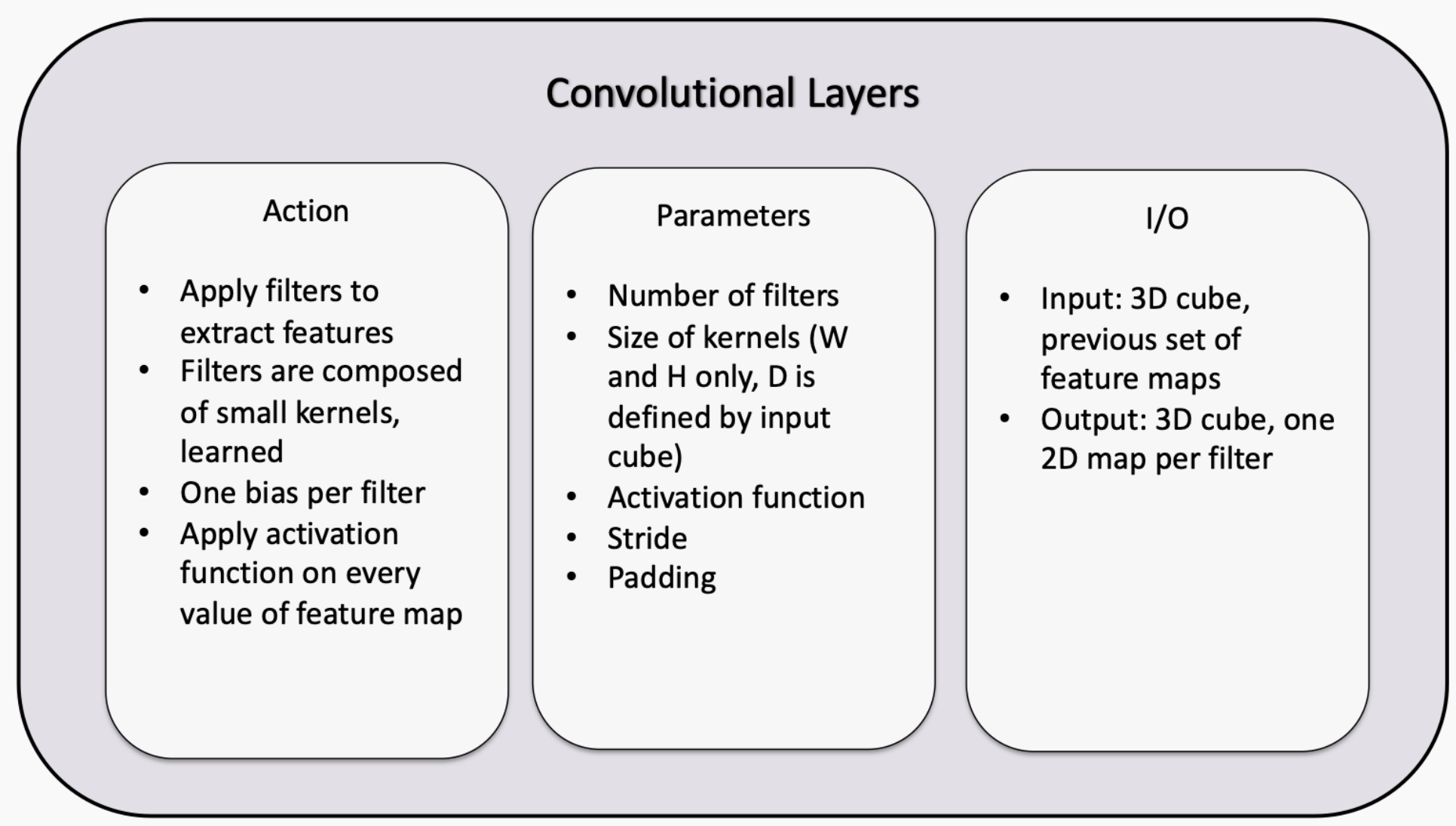
A convolutional neural network is built by stacking layers, typically of 3 types:



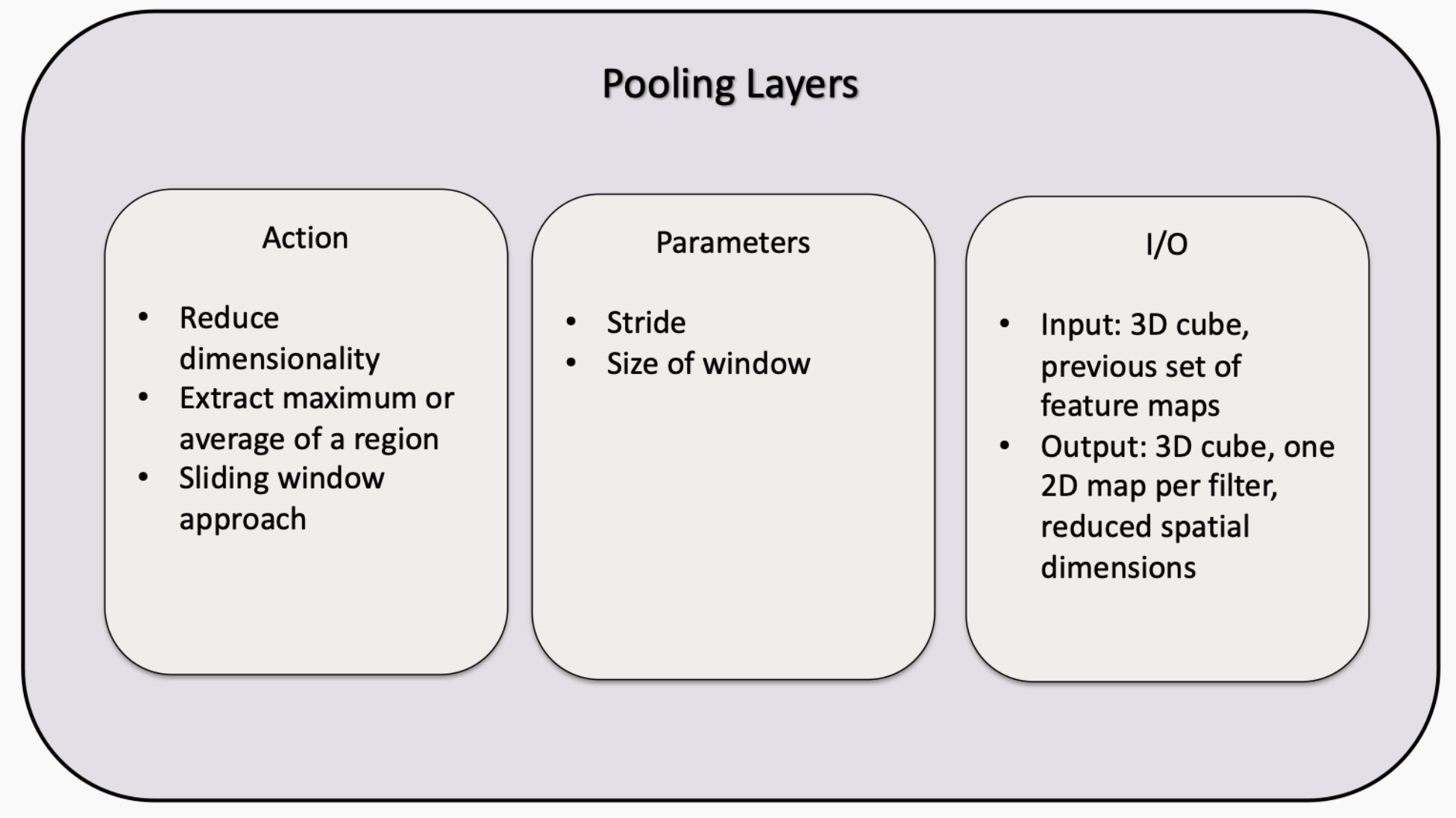




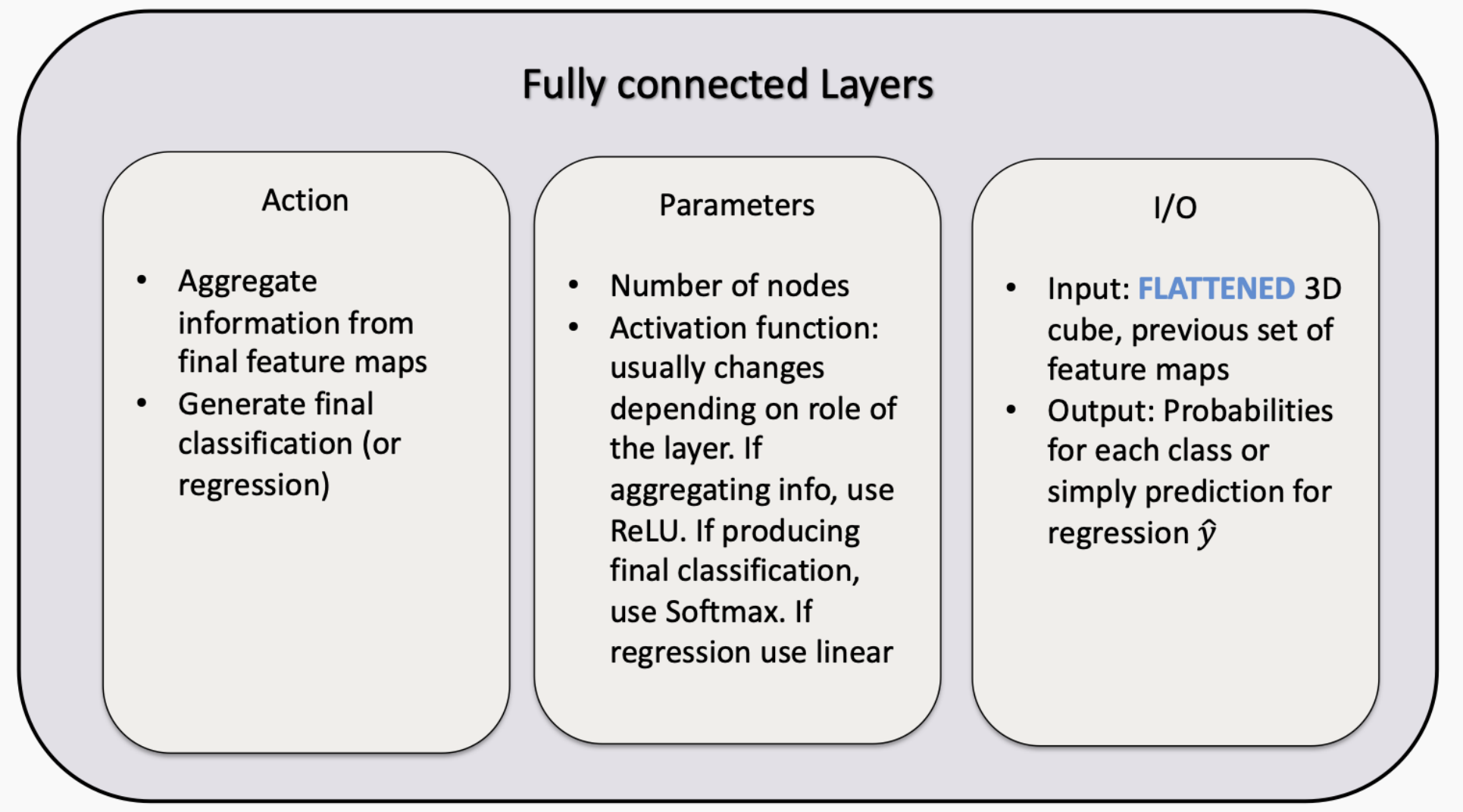
Building a CNN



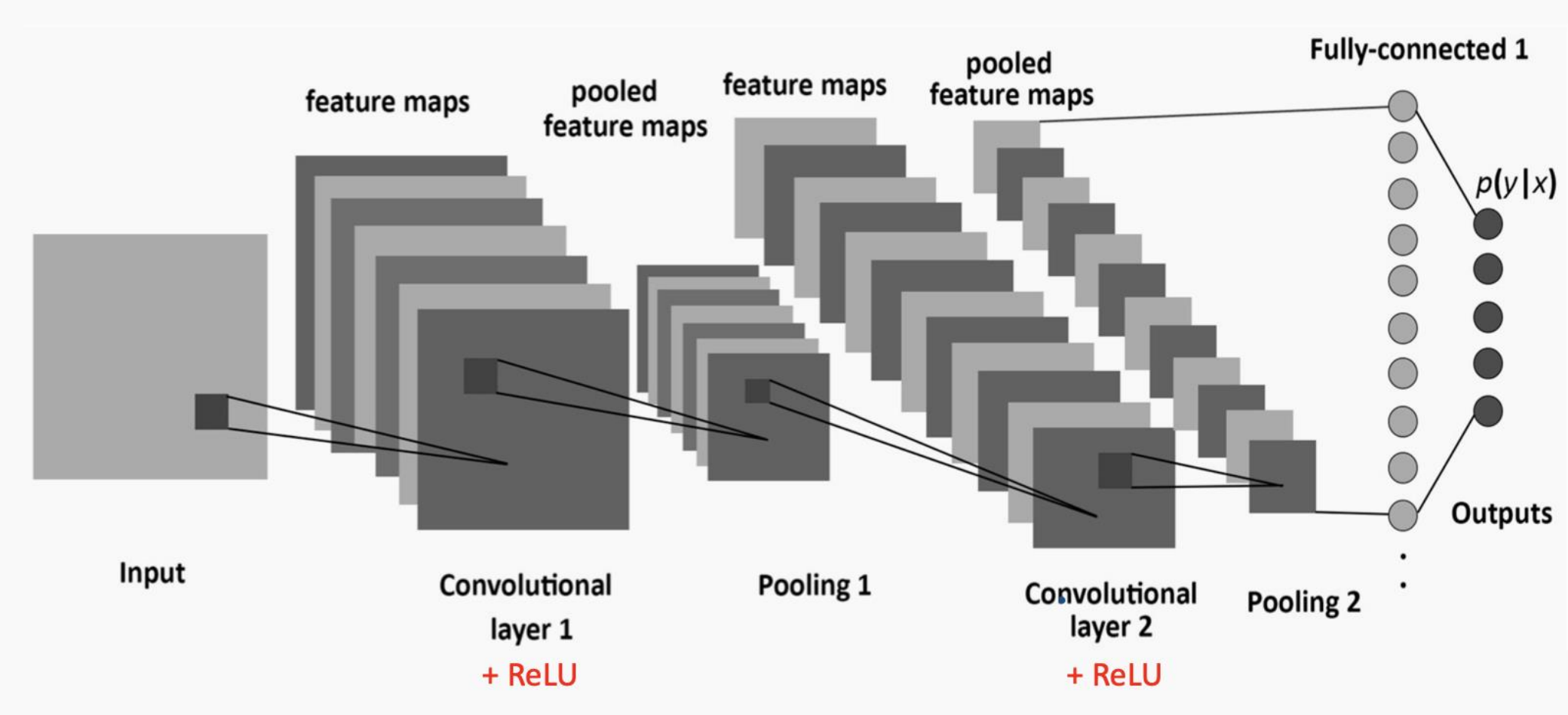
Building a CNN



Building a CNN



A Convolutional Network



Example

- Let **C** be a CNN with the following disposition:
  - Input: 32x32x3 images
  - Conv1: 8 3x3 filters, stride 1, padding=same
  - Conv2: 16 5x5 filters, stride 2, padding=same
  - Flatten layer
  - Dense1: 512 nodes
  - Dense2: 4 nodes
- How many parameters does this network have?

$$(8 \times 3 \times 3 \times 3 + 8) + (16 \times 5 \times 5 \times 8 + 16) + (16 \times 16 \times 16 \times 512 + 512) + (512 \times 4 + 4)$$

Conv1 Conv2 Dense1 Dense2

Conv Nets: Training

- In a convolutional layer, we are basically applying multiple filters over the image to extract different features. But most importantly, we are learning those filters!

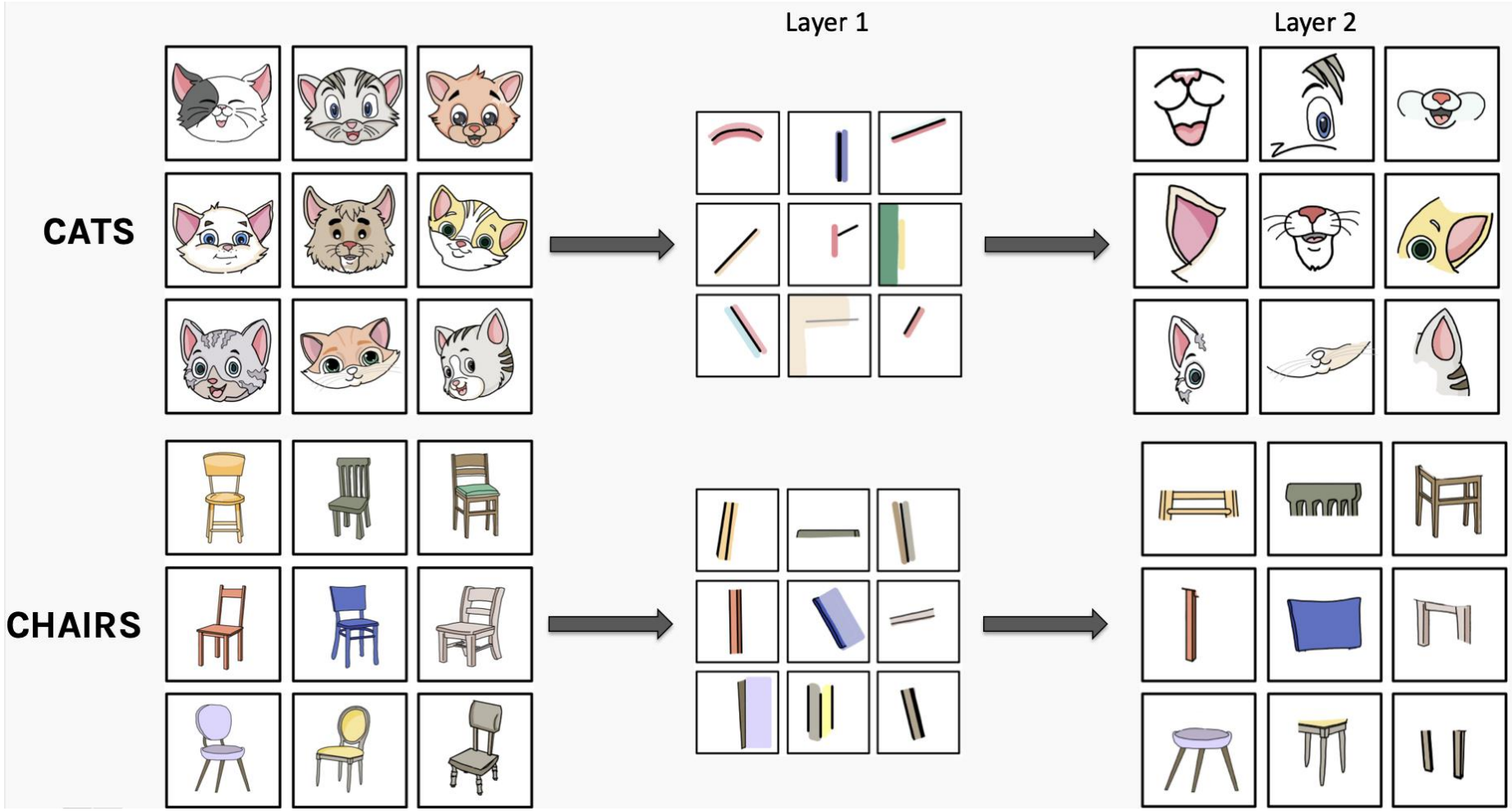
- Since convolutions and sub-sampling are differentiable, we can use standard back-propagation.

- Algorithm:

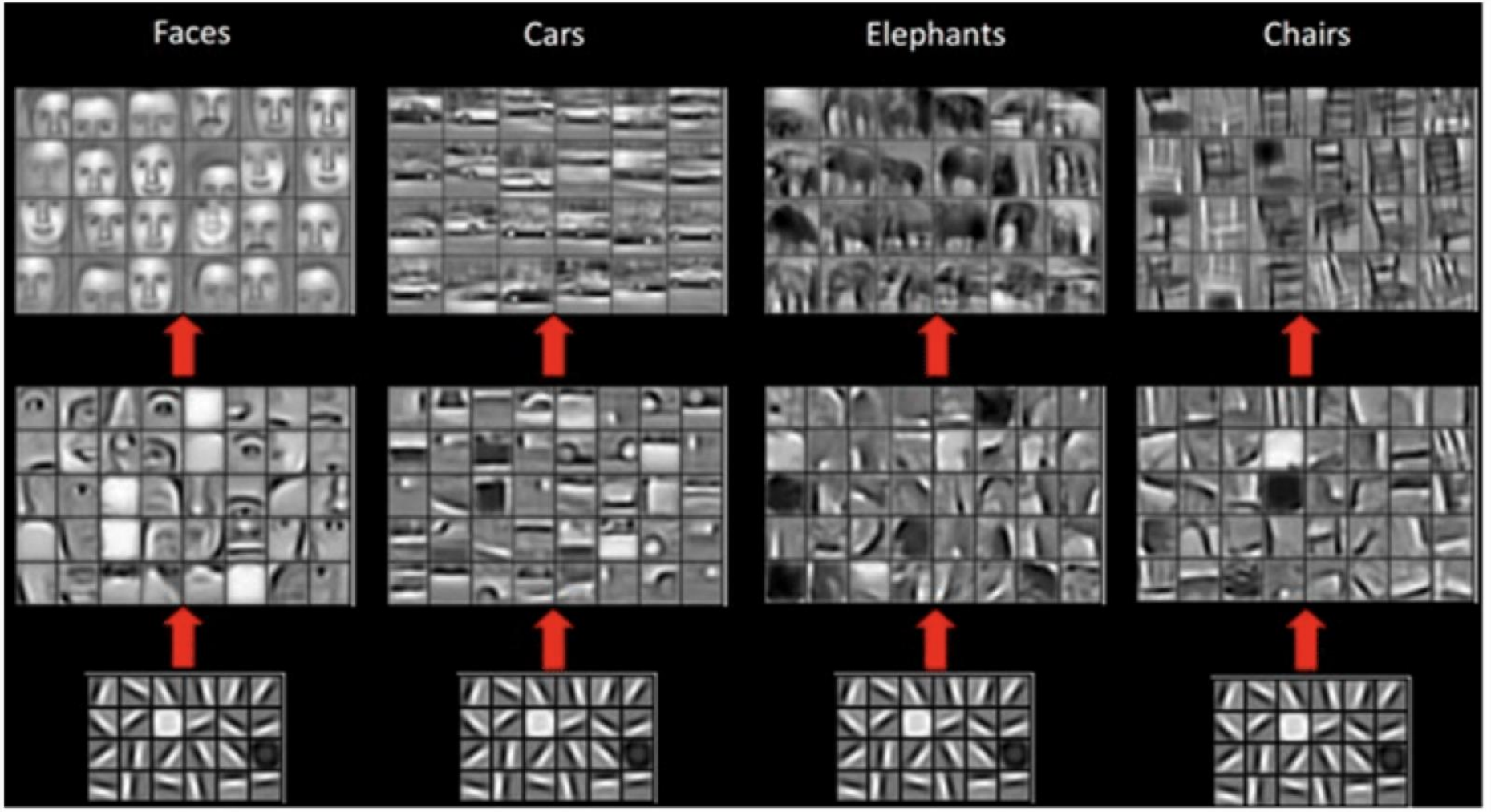
  Given a small mini-batch

  - Forward Propagation
  - Backward Propagation
  - Parameter Update

What do CNN layers learn?



What do CNN layers learn?

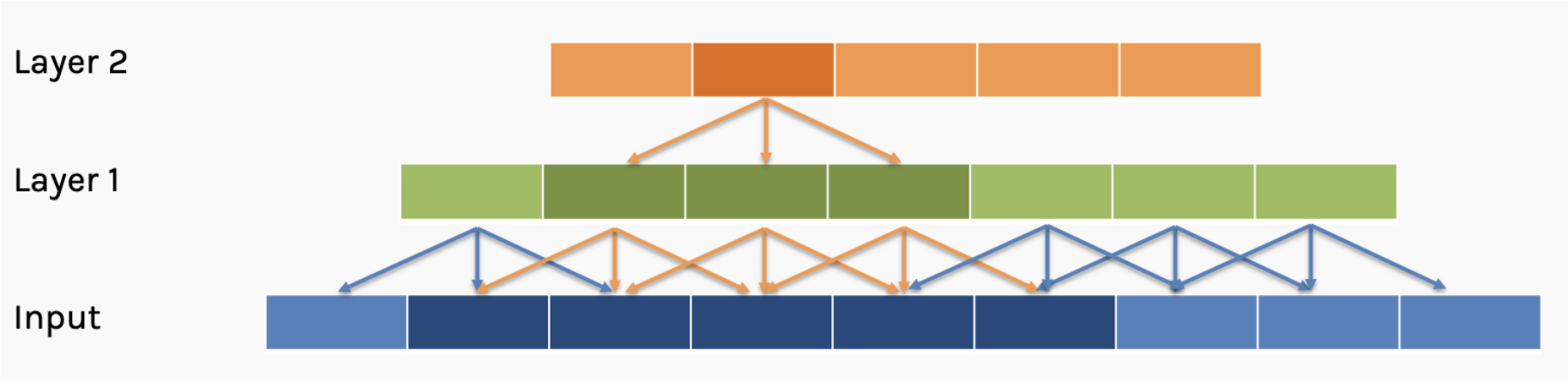


Layers Receptive Field

- The **receptive field** is defined as the region in the input space that a particular CNN's feature is looking at (i.e., be affected by).

- The receptive field size is a crucial issue in many visual tasks, as the output must respond to large enough areas in the image to capture information about large objects.

- Let's look at the receptive field in 1D, no padding, stride 1 and kernel 3x1



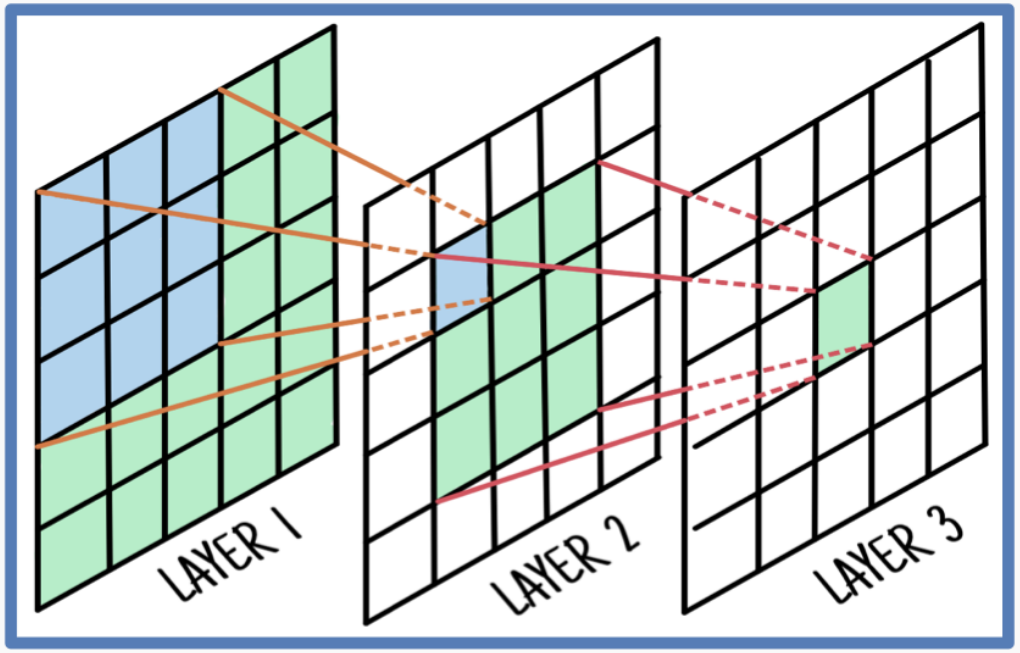
Layers Receptive Field

In 2D, it works the same way.

The receptive field for layer L, r_0, can be calculated using the recursive formula:

$$r_0 = \sum_{l=1}^{L} \left((k_l - 1) \prod_{i=1}^{l-1} s_i \right) + 1$$

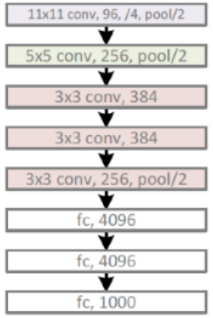
- k_l kernel size (positive integer)
- s_l stride (positive integer)



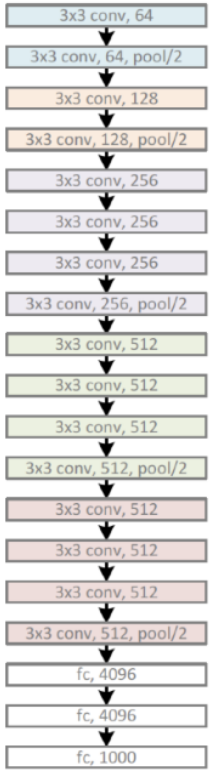
Note: For every max-pooling layer, we multiply the receptive field by the window size (assuming stride is the same as the window size)

Popular Convolutional Neural Networks (CNNs)

AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)

Software


TensorFlow


torch


mxnet


theano


Caffe


Caffe2


MatConvNet


The Microsoft Cognitive Toolkit
A free, easy-to-use, open-source, commercial-grade toolkit that trains deep learning algorithms to learn like the human brain.


PYTORCH