

TDDC17 LE10 HT2024

Machine Learning II

Fredrik Heintz

**Dept. of Computer Science
Linköping University**

fredrik.heintz@liu.se

@FredrikHeintz

Outline:

- **Neural networks**
- **Convolutional Neural Networks**
- **Deep Generative Learning**

What is Deep Learning?

ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



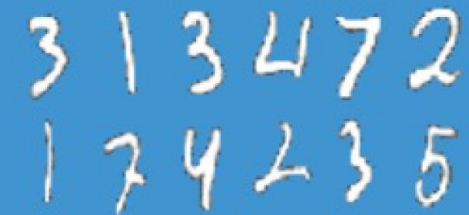
MACHINE LEARNING

Ability to learn without explicitly being programmed



DEEP LEARNING

Extract patterns from data using neural networks



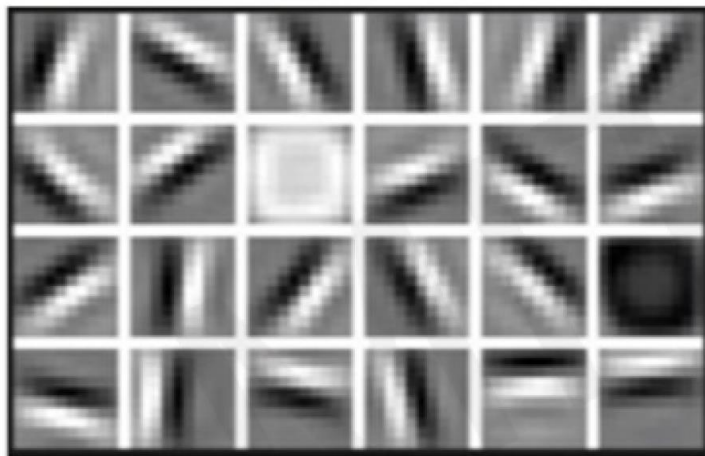
Teaching computers how to **learn a task** directly from **raw data**

Why Deep Learning?

Hand engineered features are time consuming, brittle, and not scalable in practice

Can we learn the **underlying features** directly from data?

Low Level Features



Lines & Edges

Mid Level Features



Eyes & Nose & Ears

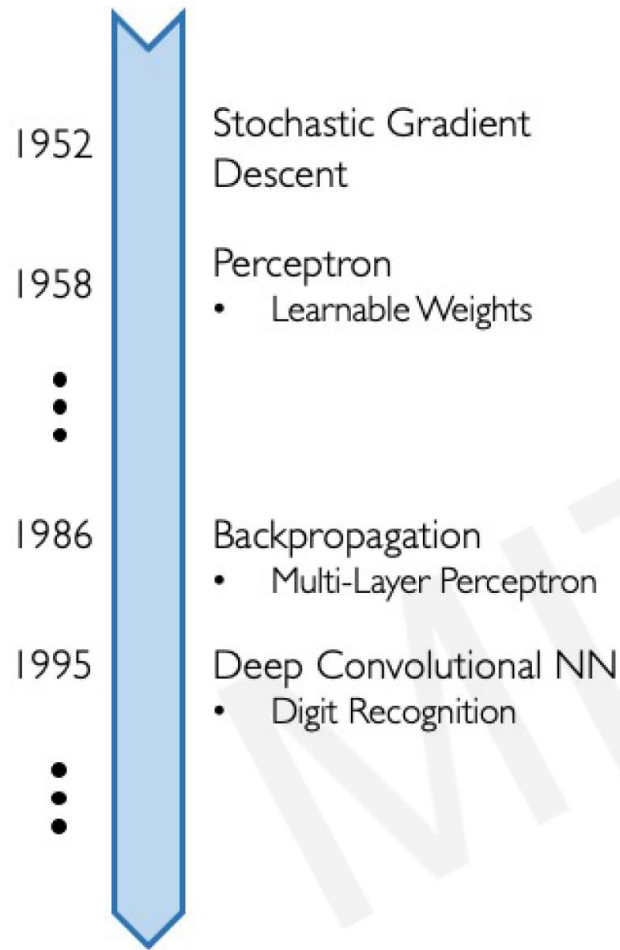
High Level Features



Facial Structure

Why Now?

Neural Networks date back decades, so why the resurgence?



1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



3. Software

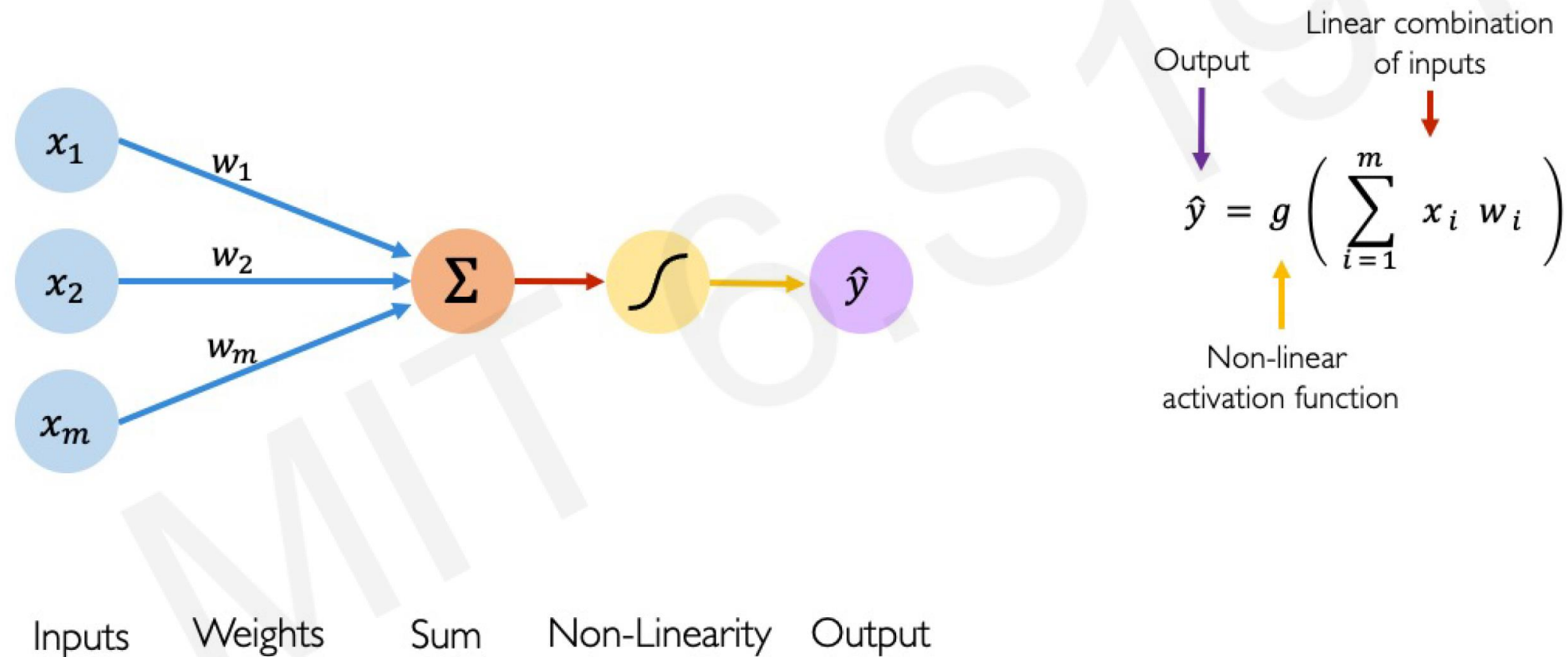
- Improved Techniques
- New Models
- Toolboxes



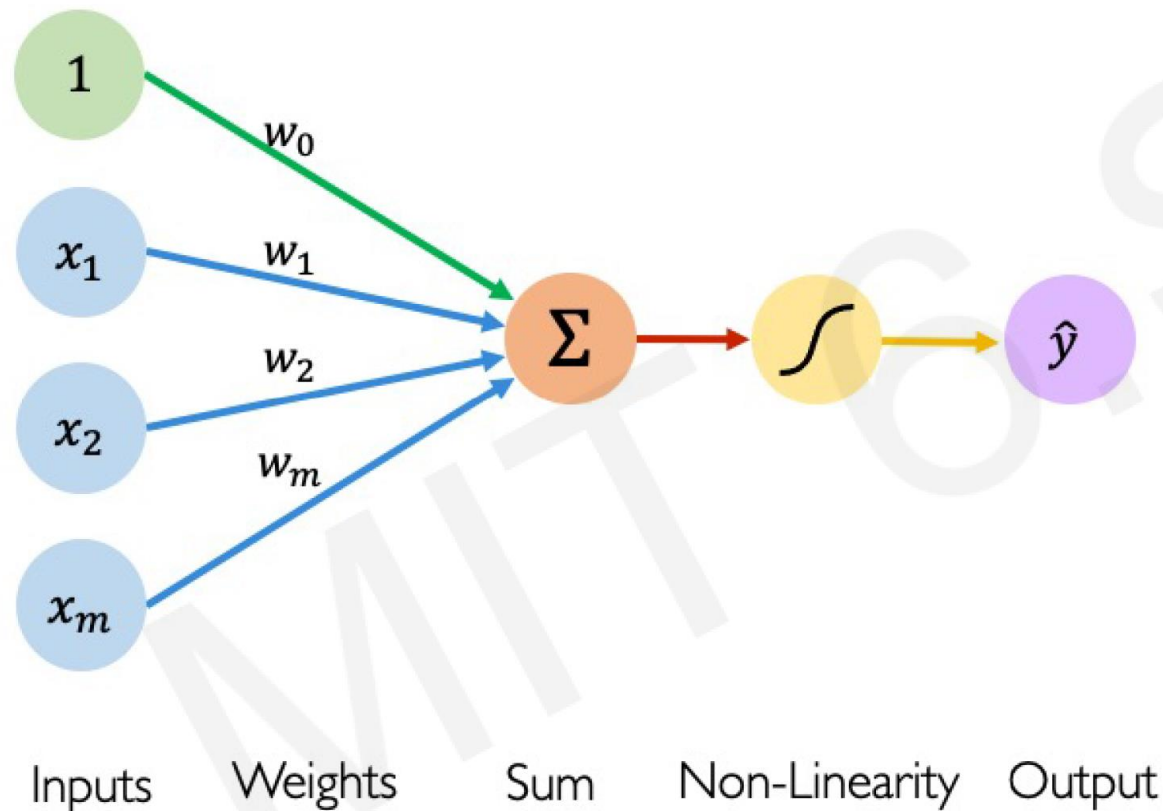
The Perceptron

The structural building block of deep learning

The Perceptron: Forward Propagation



The Perceptron: Forward Propagation



Output

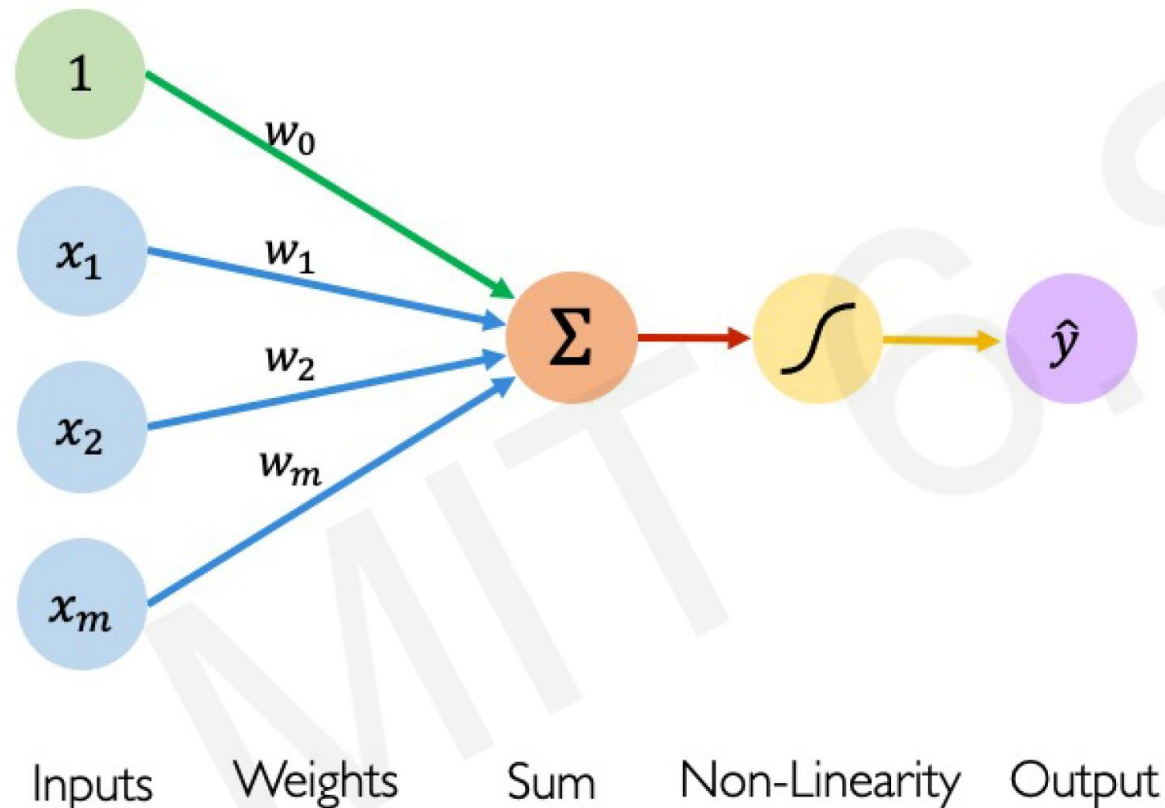
Linear combination of inputs

$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation

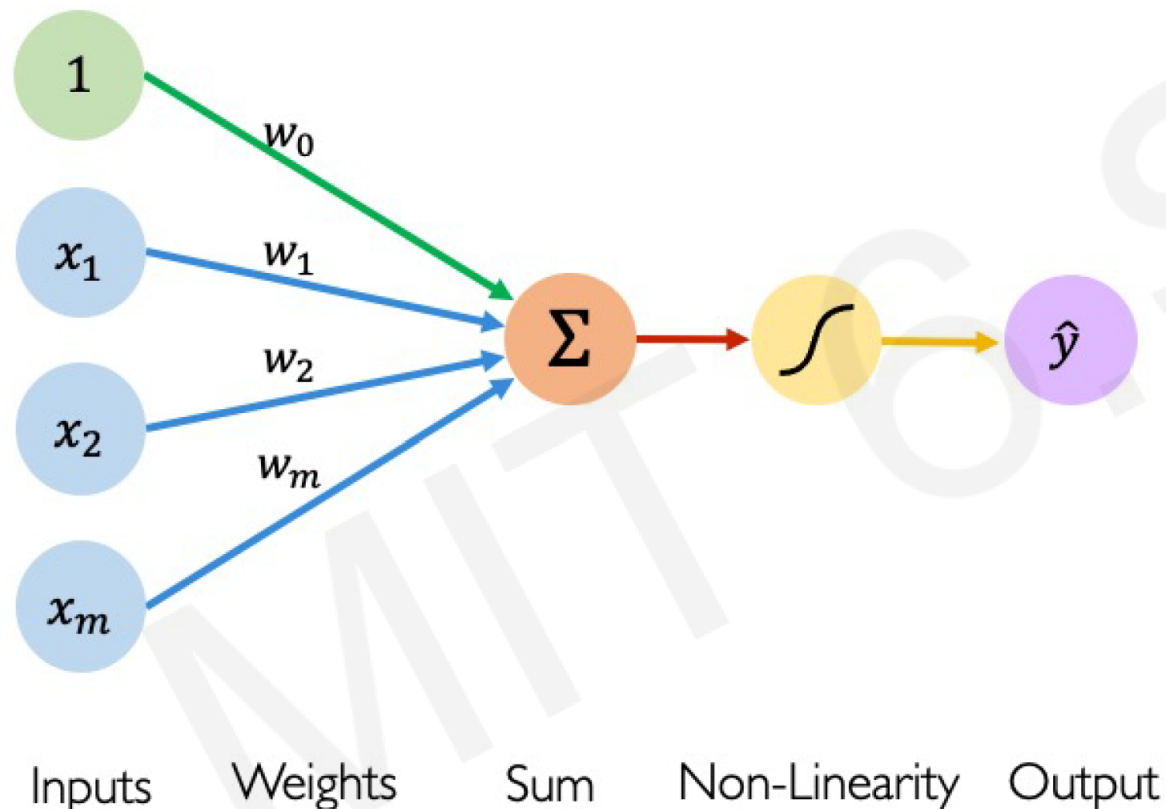


$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

$$\hat{y} = g (w_0 + \mathbf{X}^T \mathbf{W})$$

$$\text{where: } \mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

The Perceptron: Forward Propagation

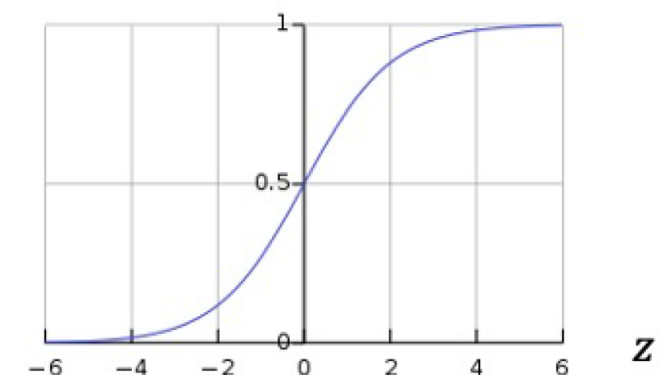


Activation Functions

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$

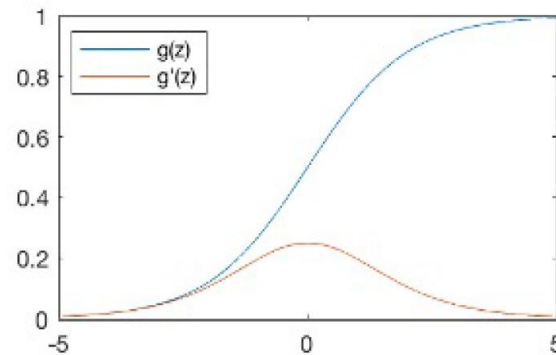
- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Common Activation Functions

Sigmoid Function

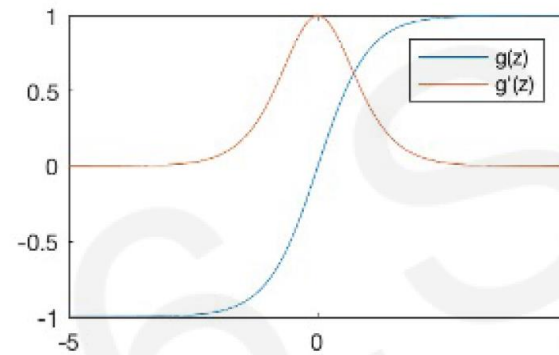


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

 `tf.math.sigmoid(z)`

Hyperbolic Tangent

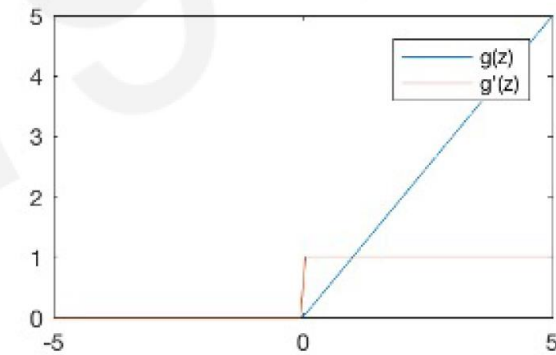


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

 `tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



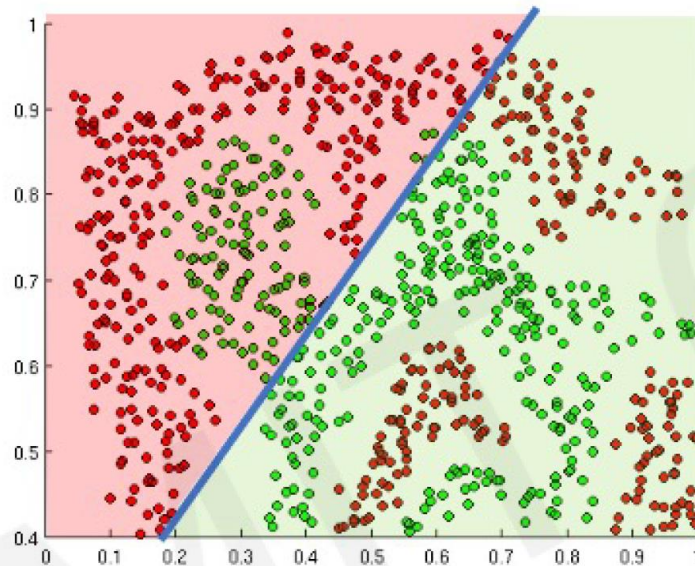
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

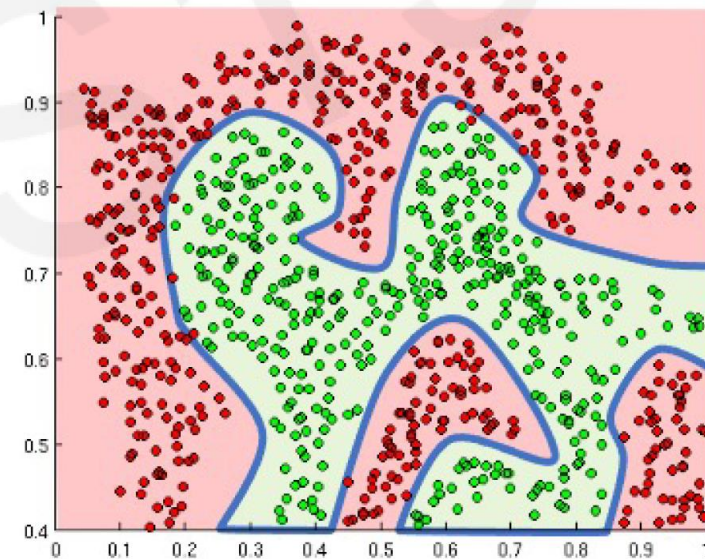
 `tf.nn.relu(z)`

Importance of Activation Functions

*The purpose of activation functions is to **introduce non-linearities** into the network*

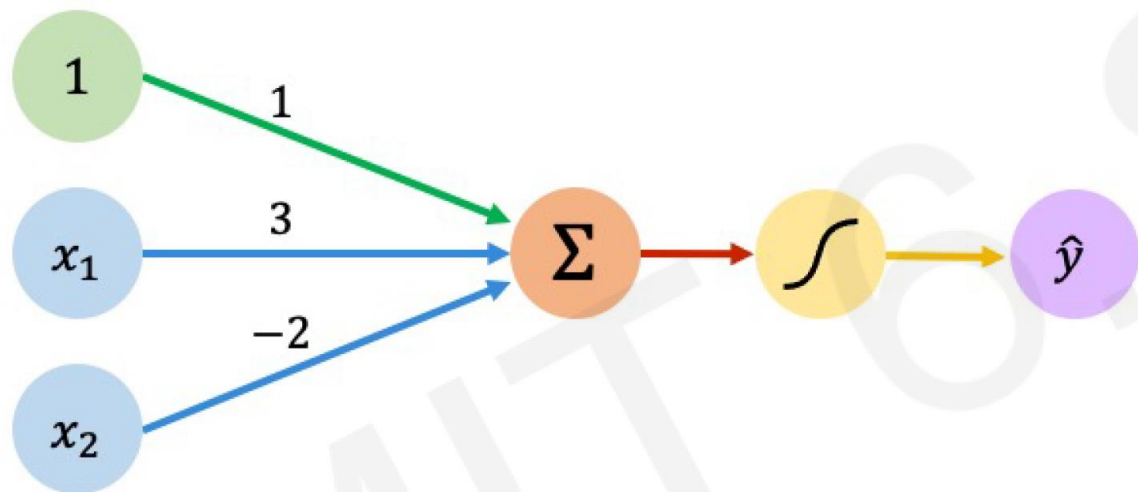


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

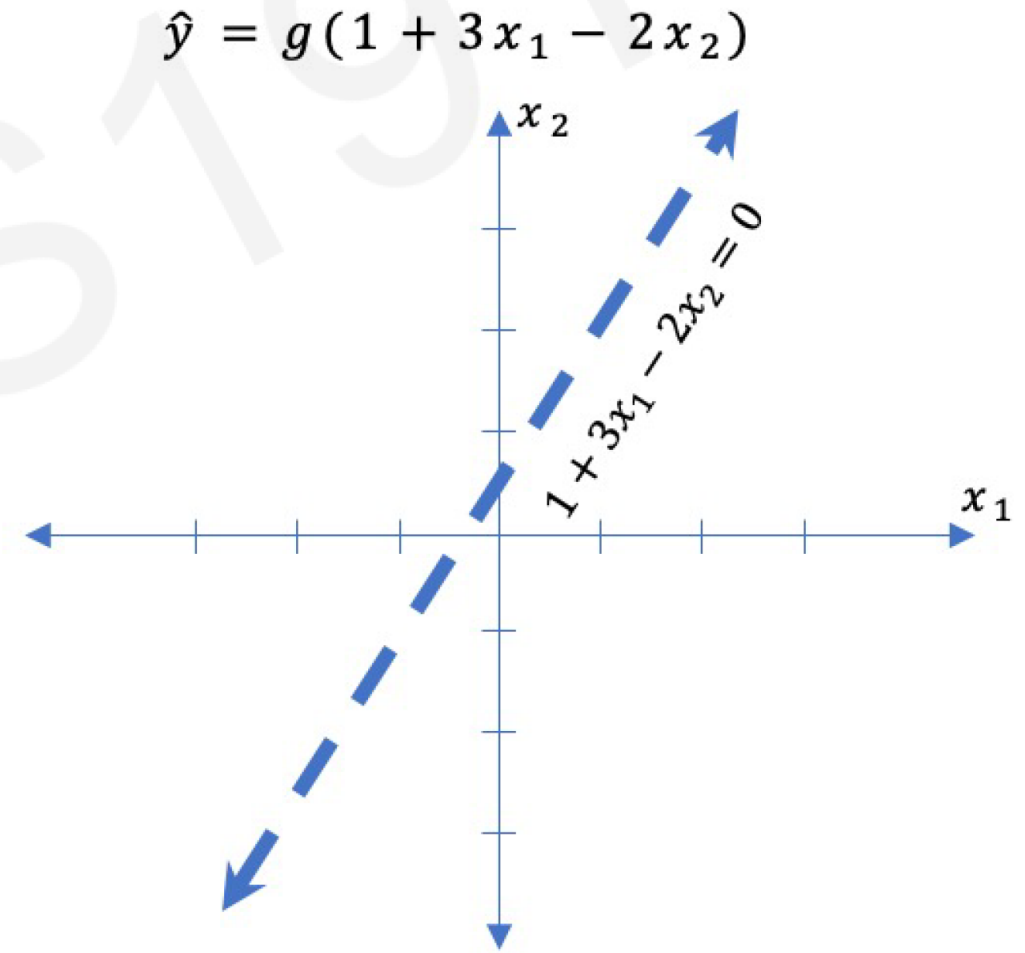
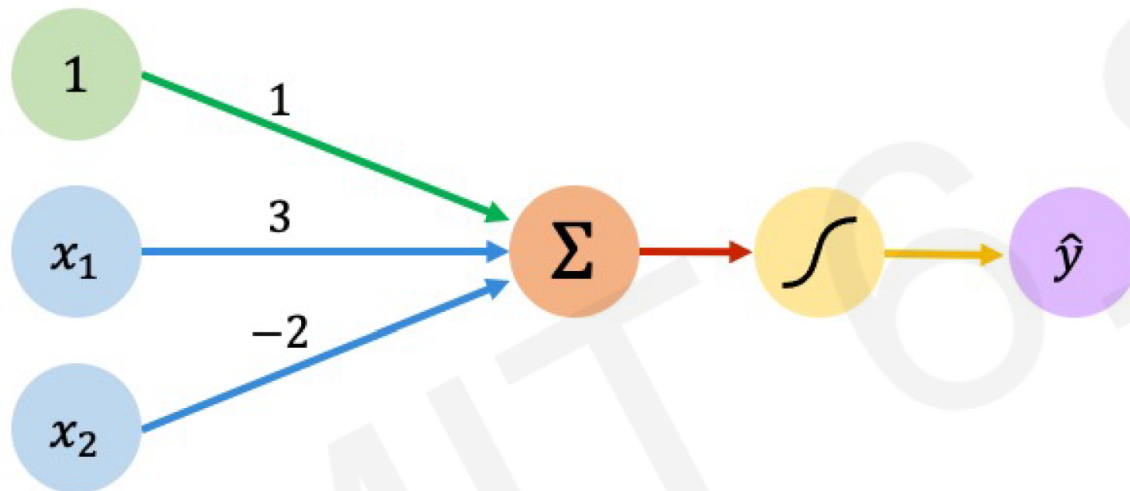


We have: $w_0 = 1$ and $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

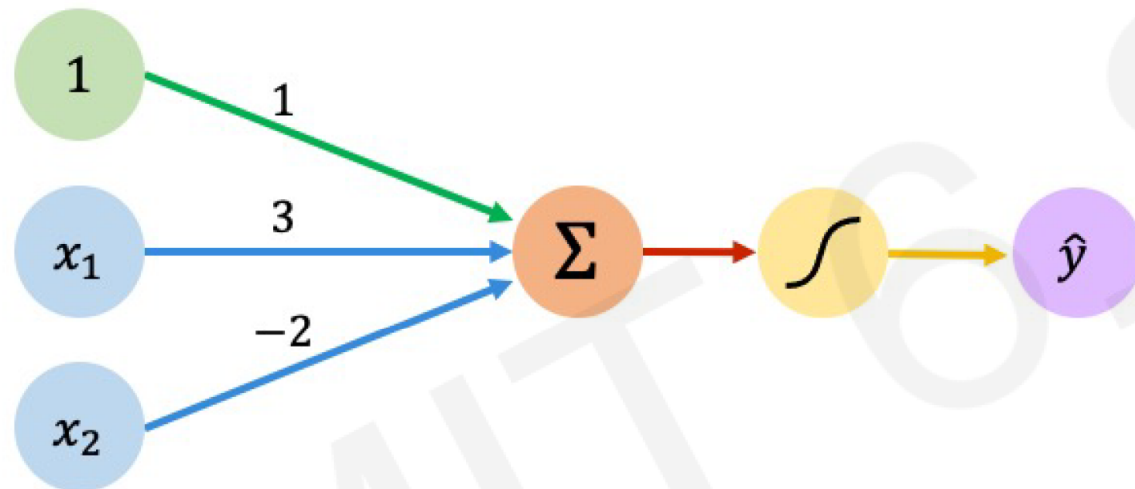
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

The Perceptron: Example

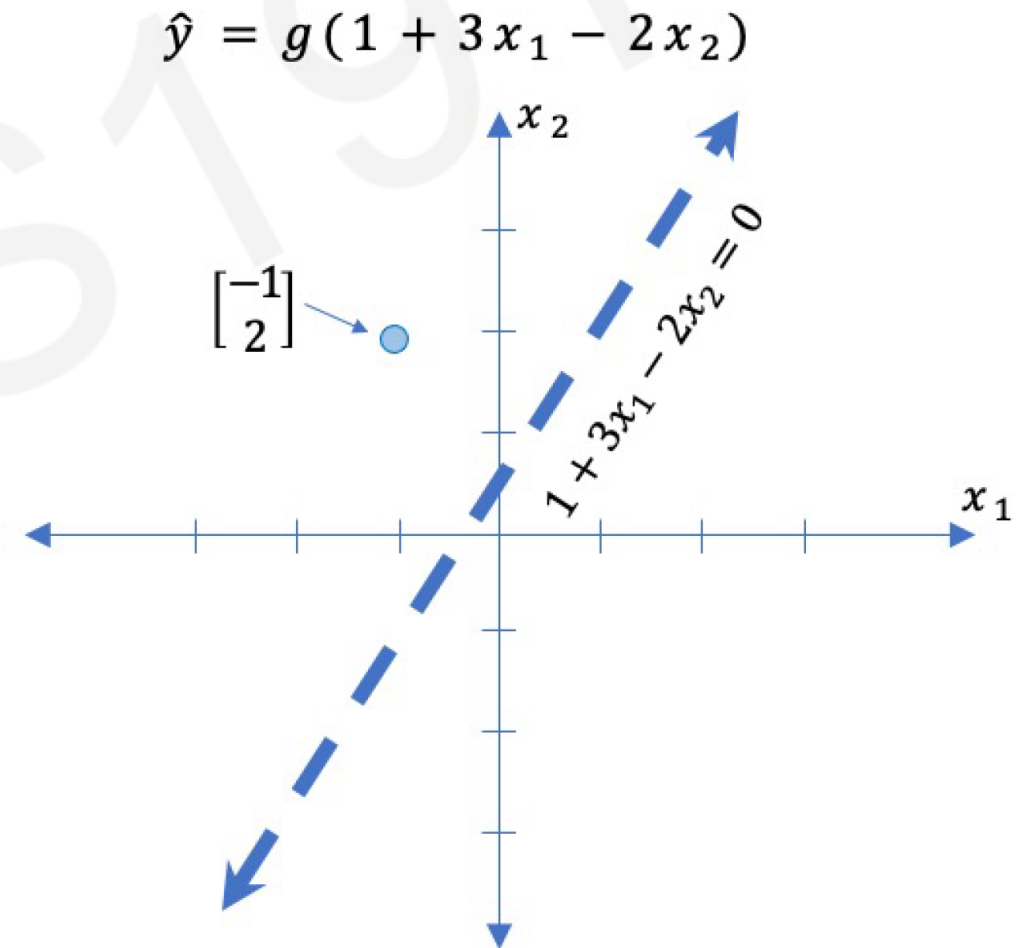


The Perceptron: Example

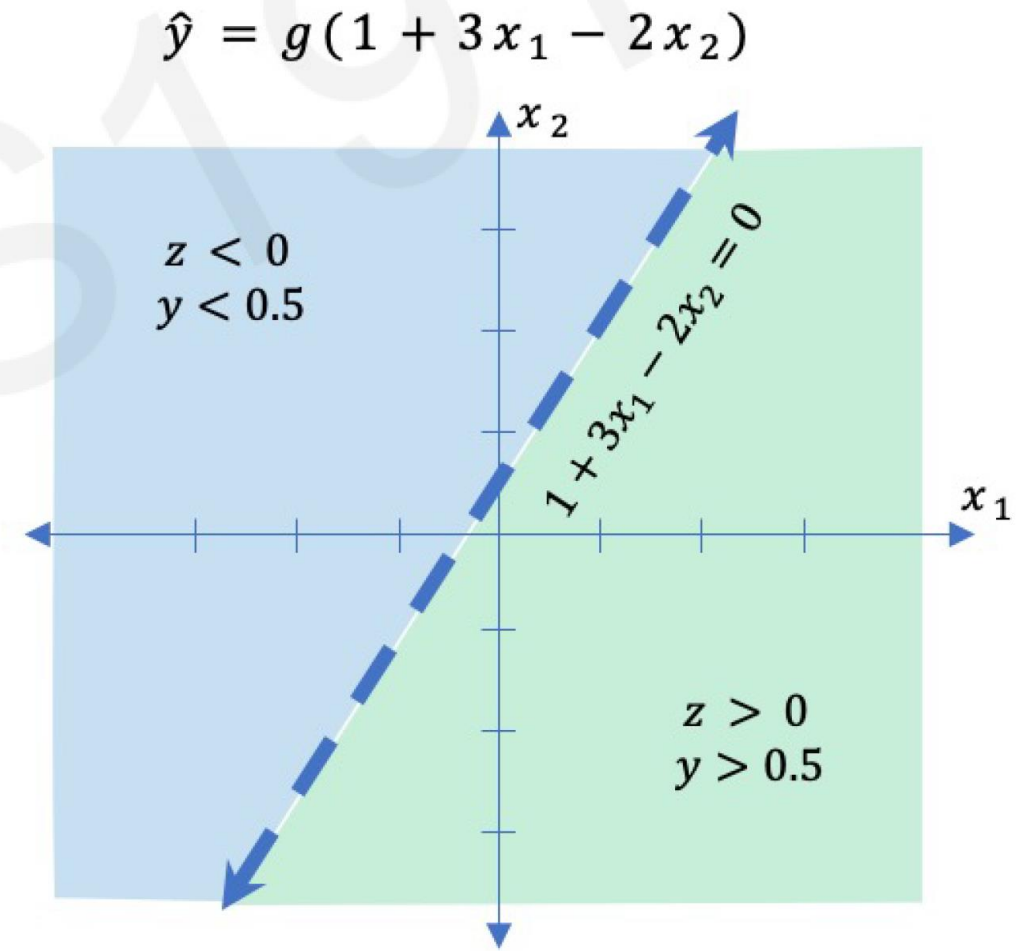
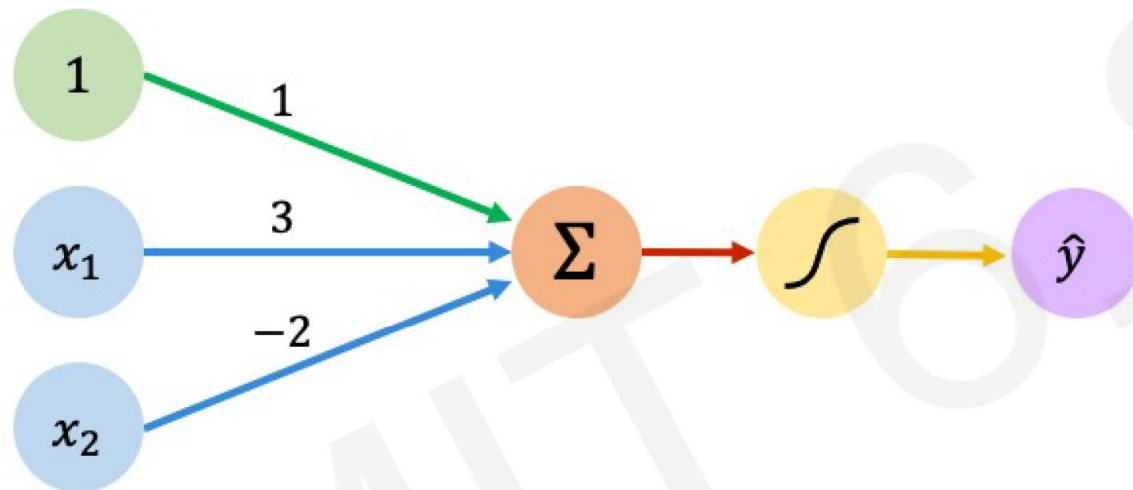


Assume we have input: $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$



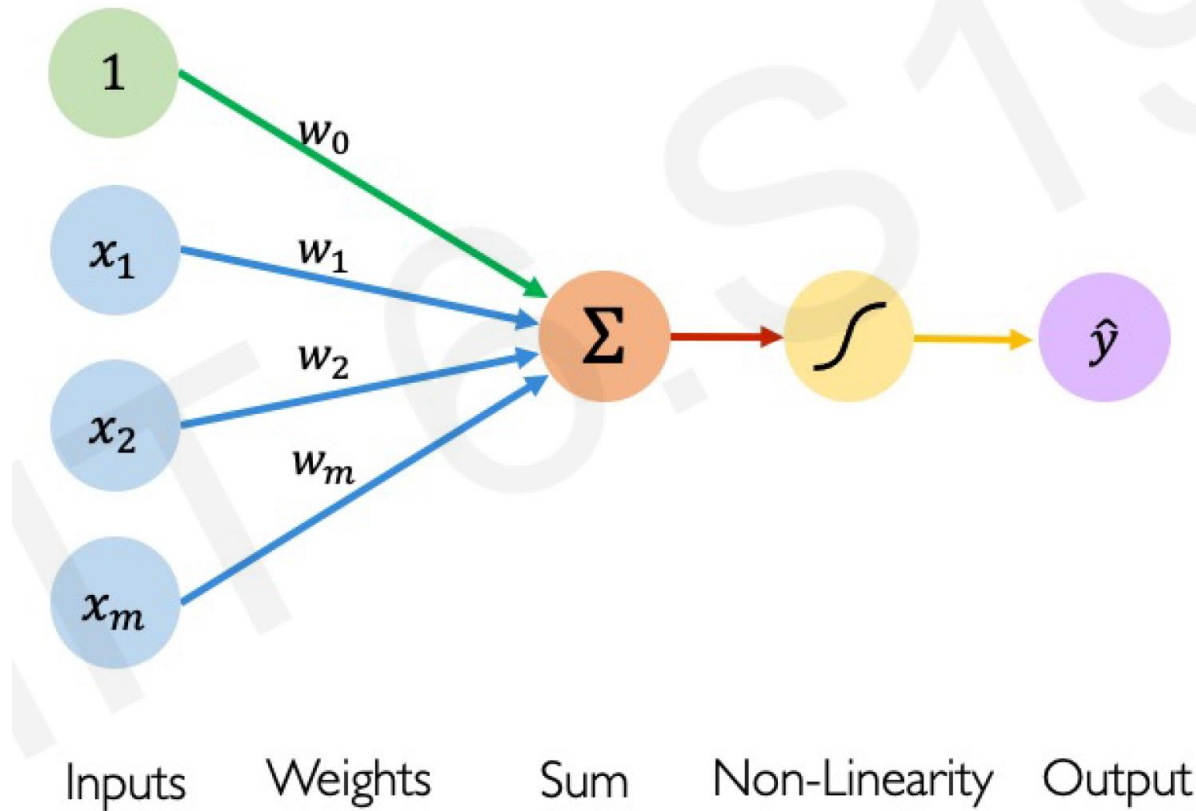
The Perceptron: Example



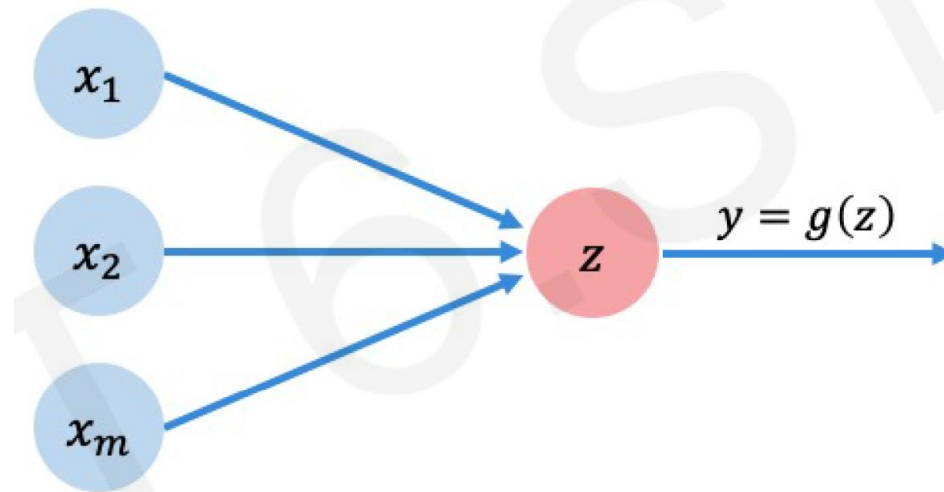
Building Neural Networks with Perceptrons

The Perceptron Simplified

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$



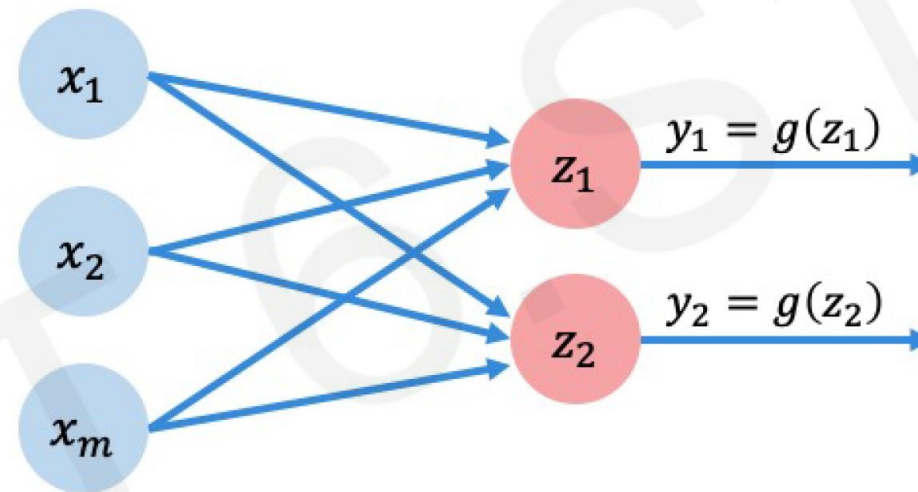
The Perceptron Simplified



$$z = w_0 + \sum_{j=1}^m x_j w_j$$

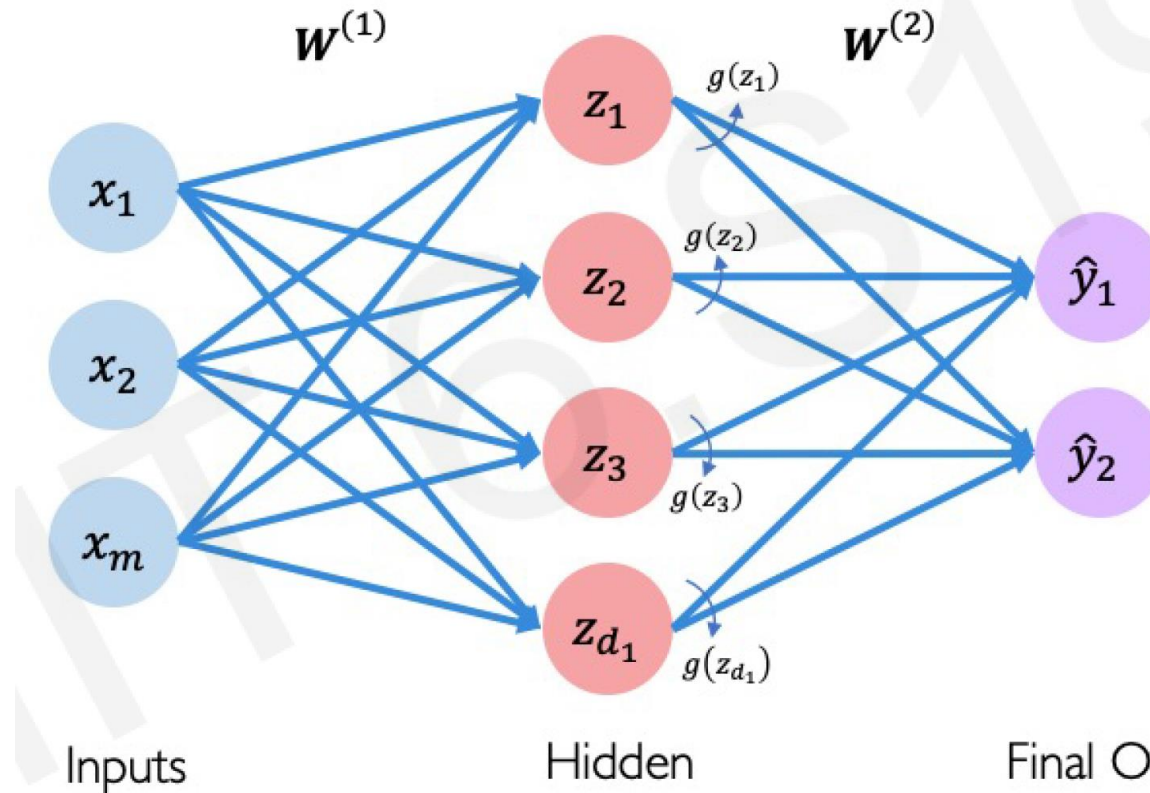
Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

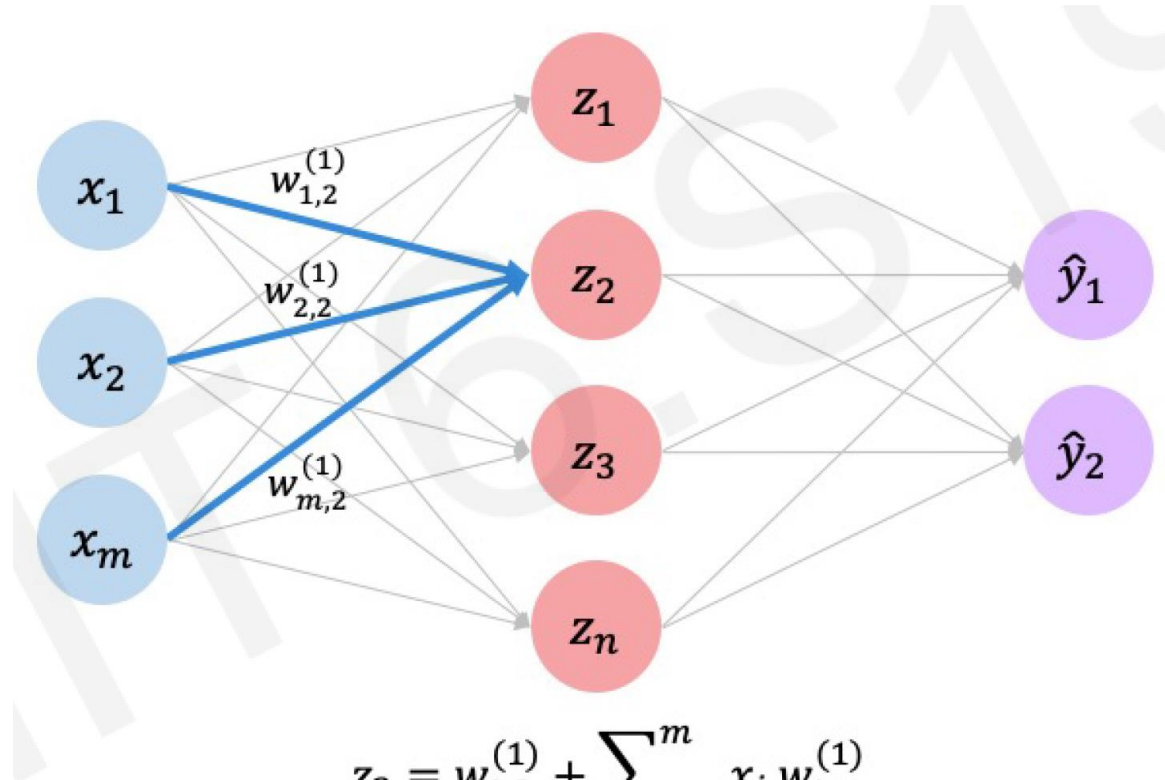
Single Layer Neural Network



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)}$$

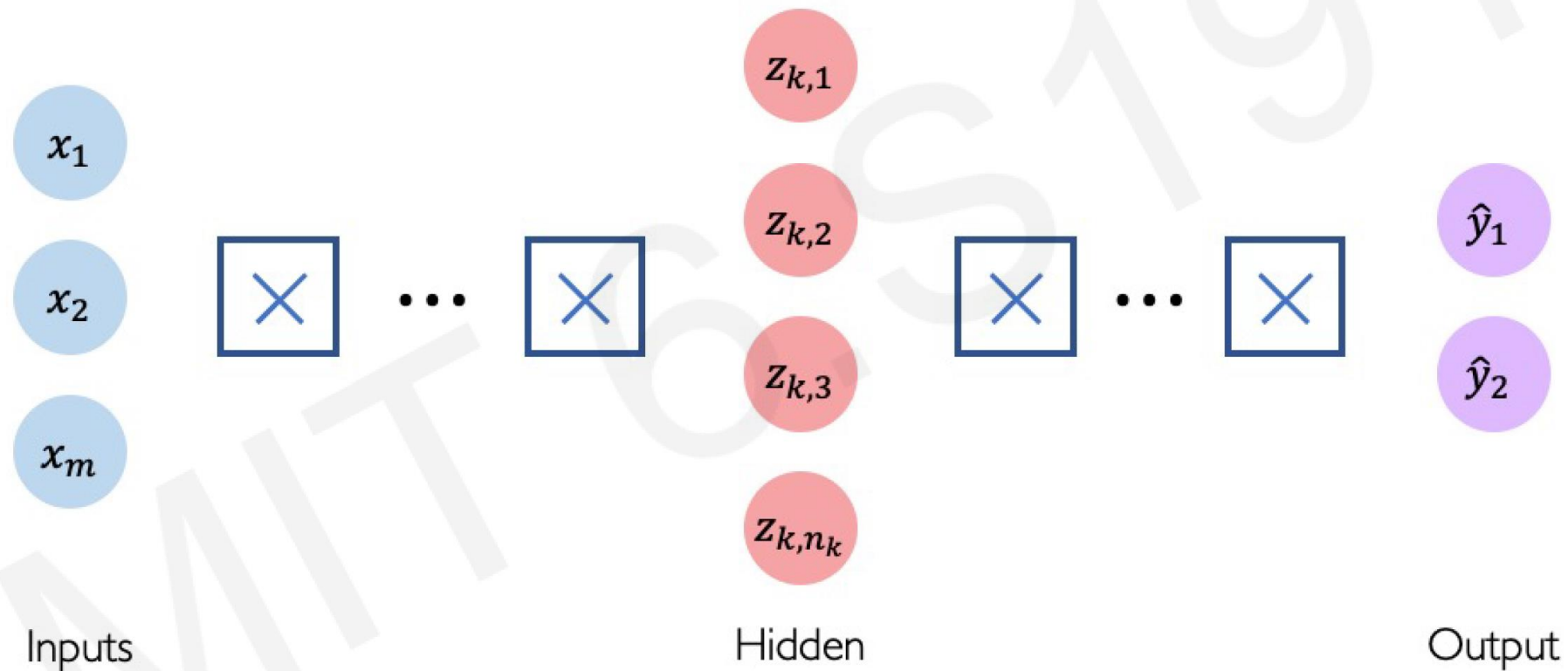
$$\hat{y}_i = g \left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$

Single Layer Neural Network



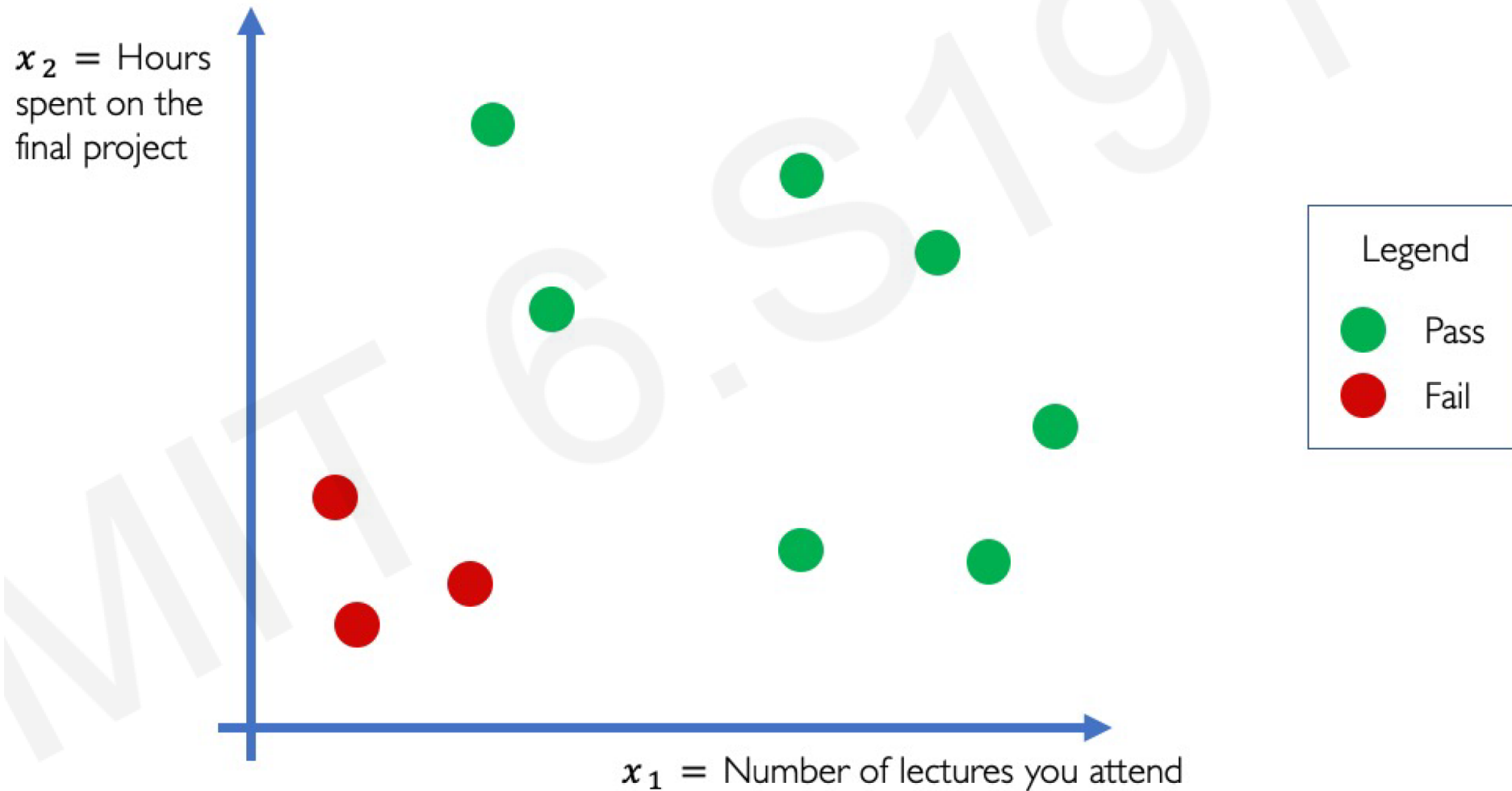
$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)} \end{aligned}$$

Deep Neural Network

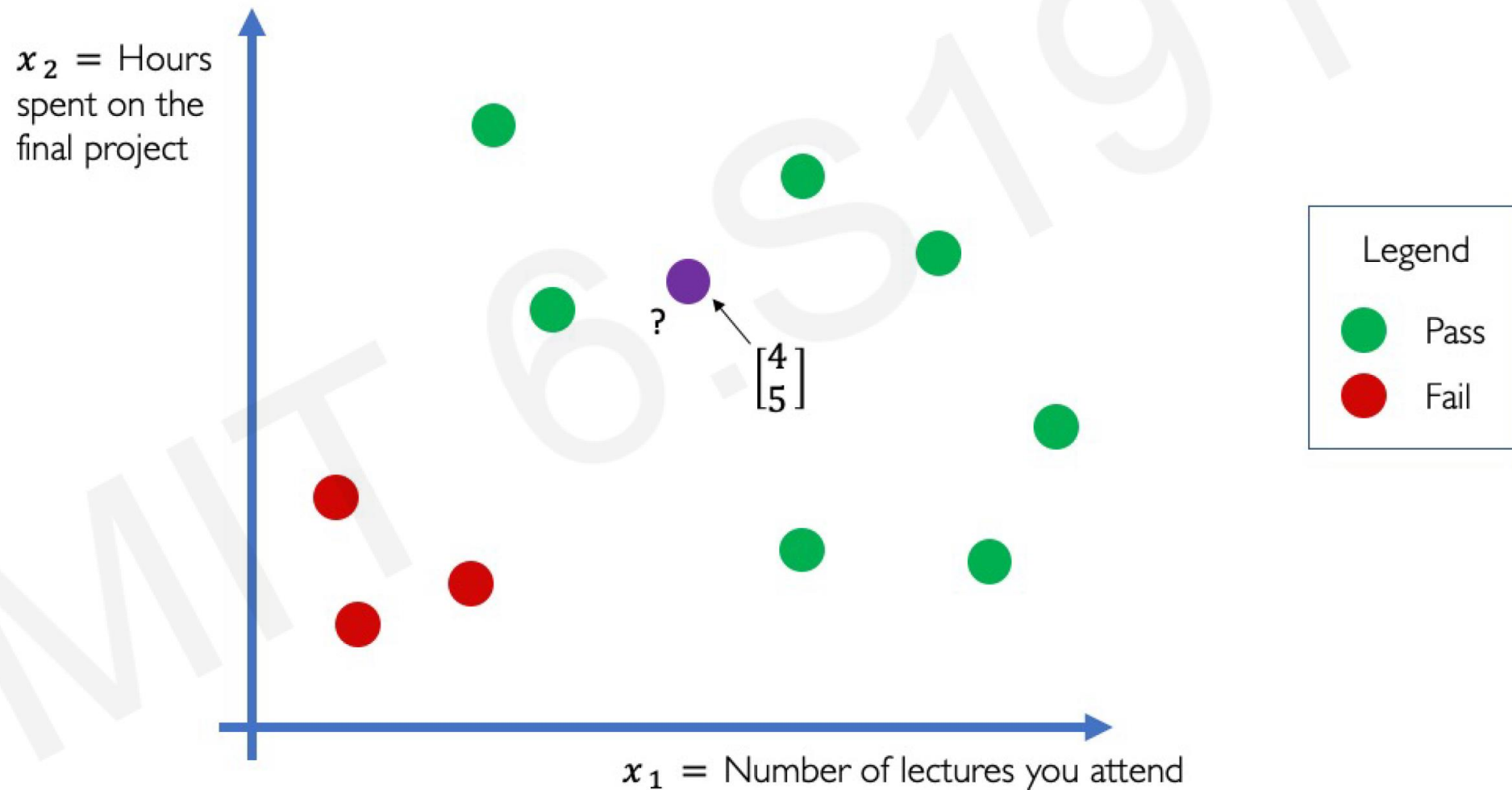


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

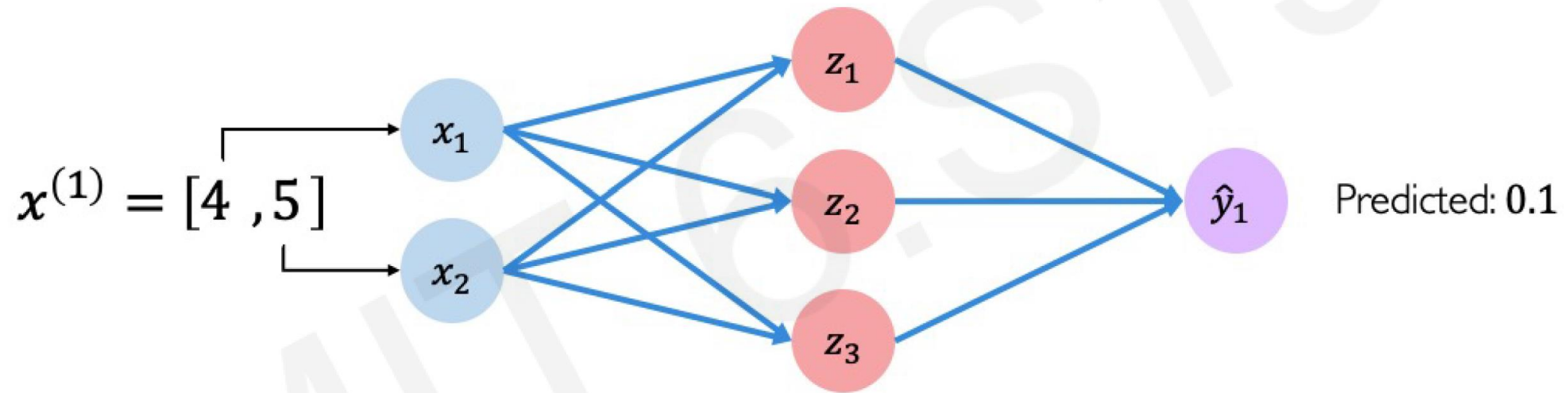
Example Problem: Will I Pass This Class?



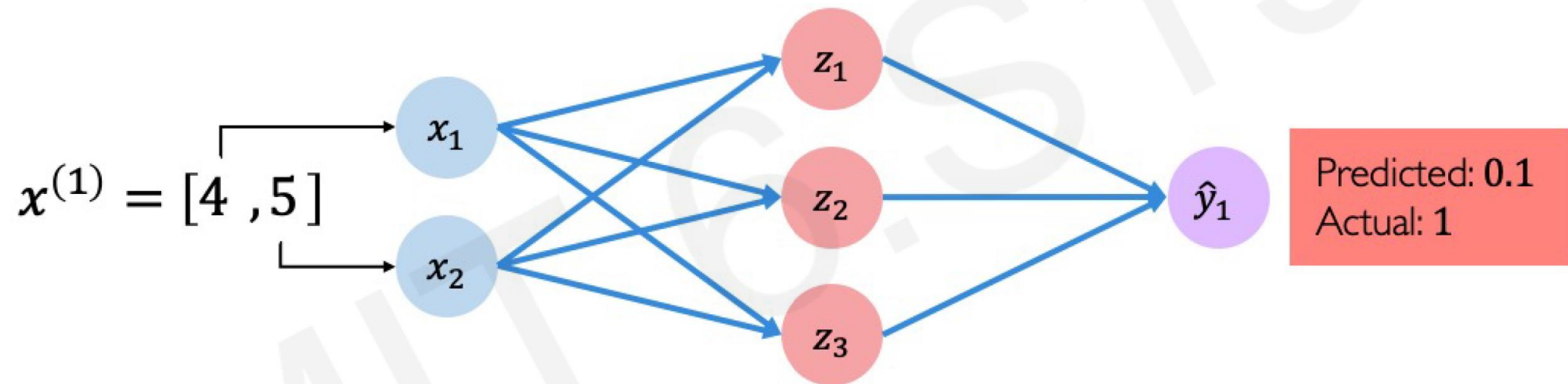
Example Problem: Will I Pass This Class?



Example Problem: Will I Pass This Class?

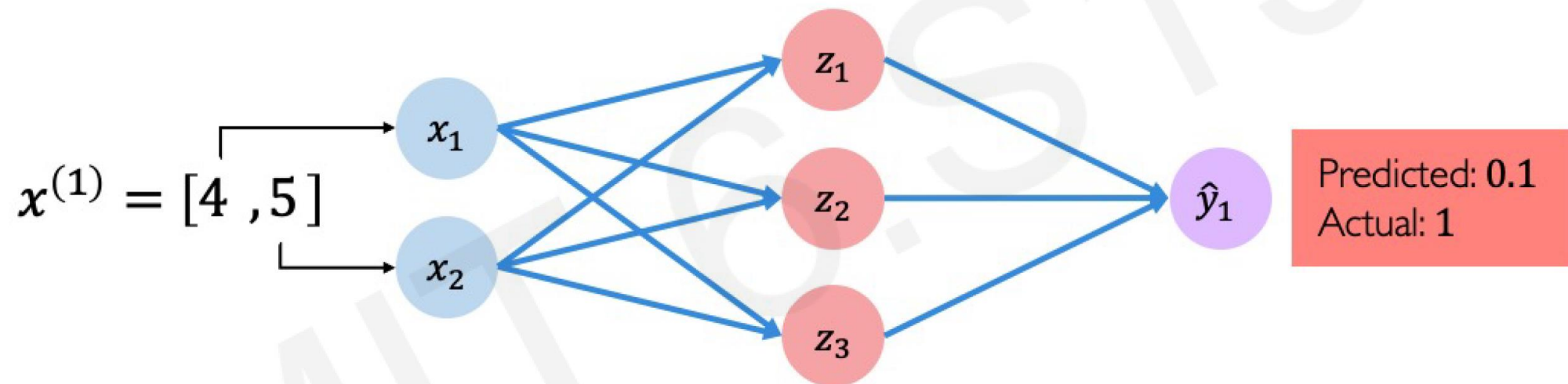


Example Problem: Will I Pass This Class?



Example Problem: Will I Pass This Class?

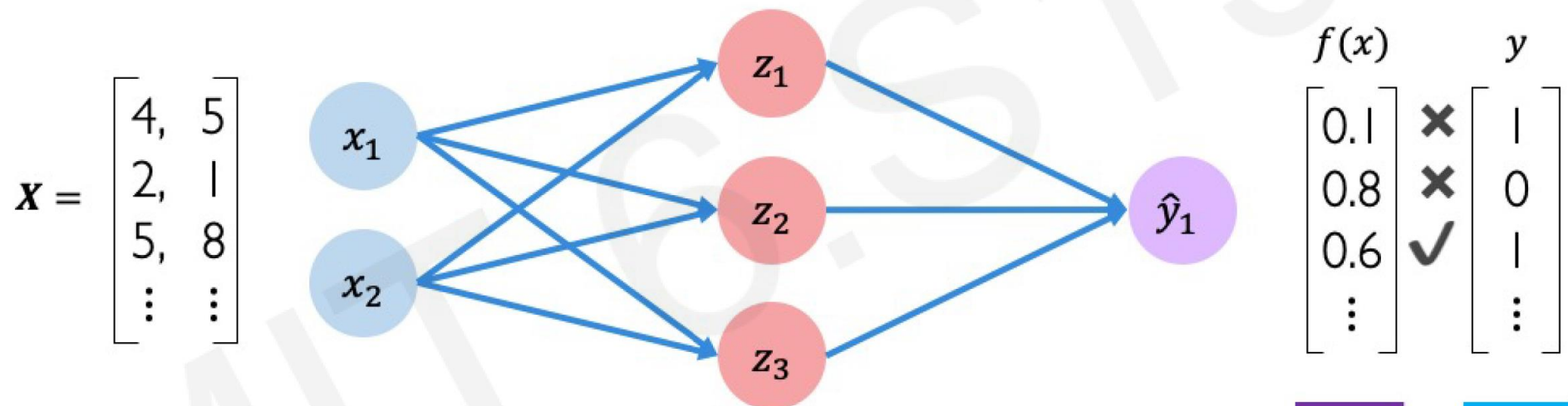
The **loss** of our network measures the cost incurred from incorrect predictions



$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss

The **empirical loss** measures the total loss over our entire dataset



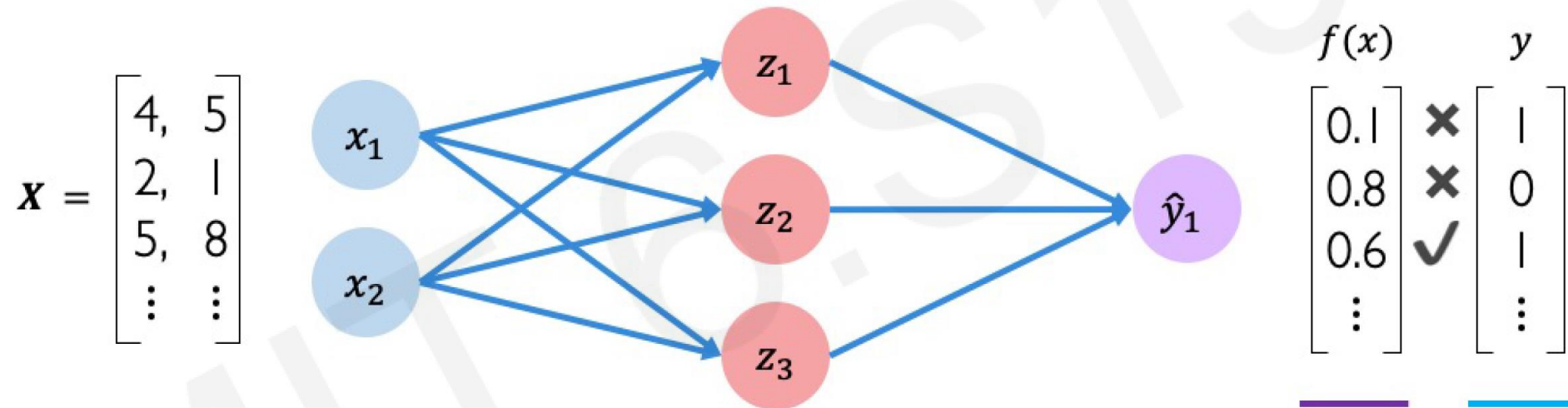
Also known as:

- Objective function
- Cost function
- Empirical Risk

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1



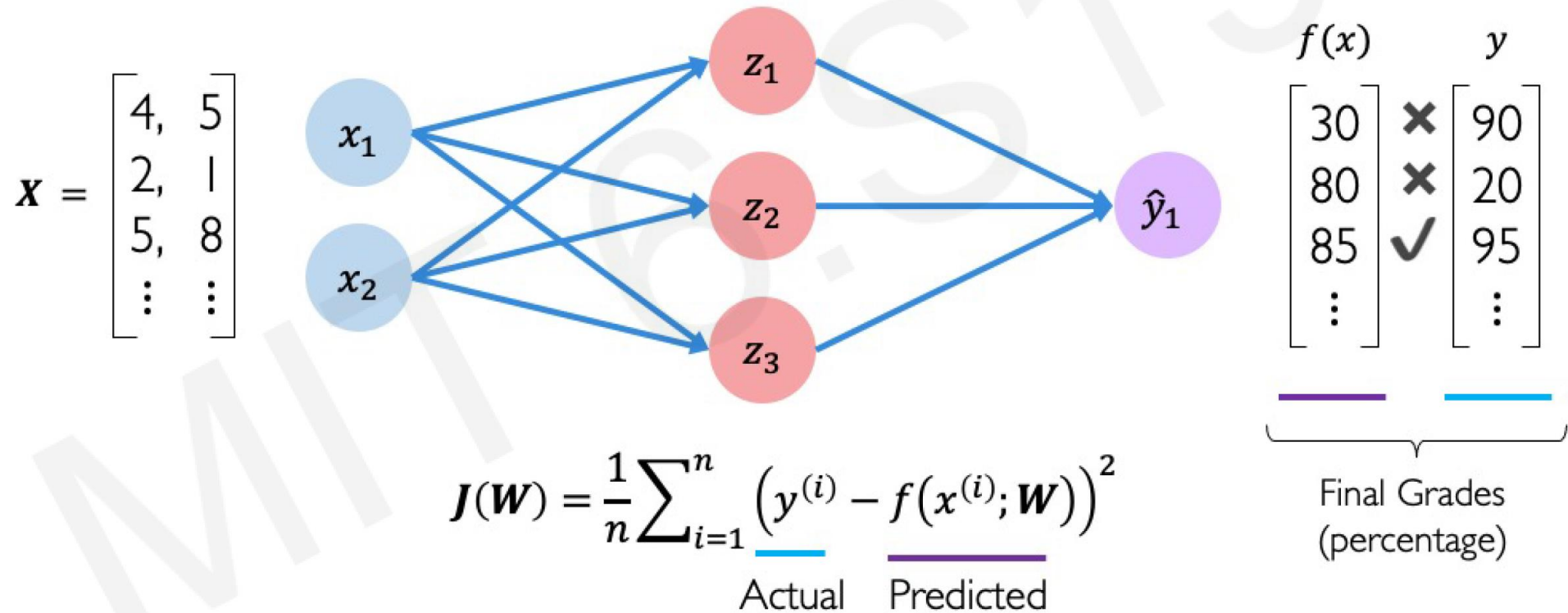
$$J(W) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left(1 - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right)$$



```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```


Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
loss = tf.keras.losses.MSE( y, predicted )
```


Training Neural Networks

Loss Optimization

*We want to find the network weights that **achieve the lowest loss***

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

Loss Optimization

We want to find the network weights that **achieve the lowest loss**

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

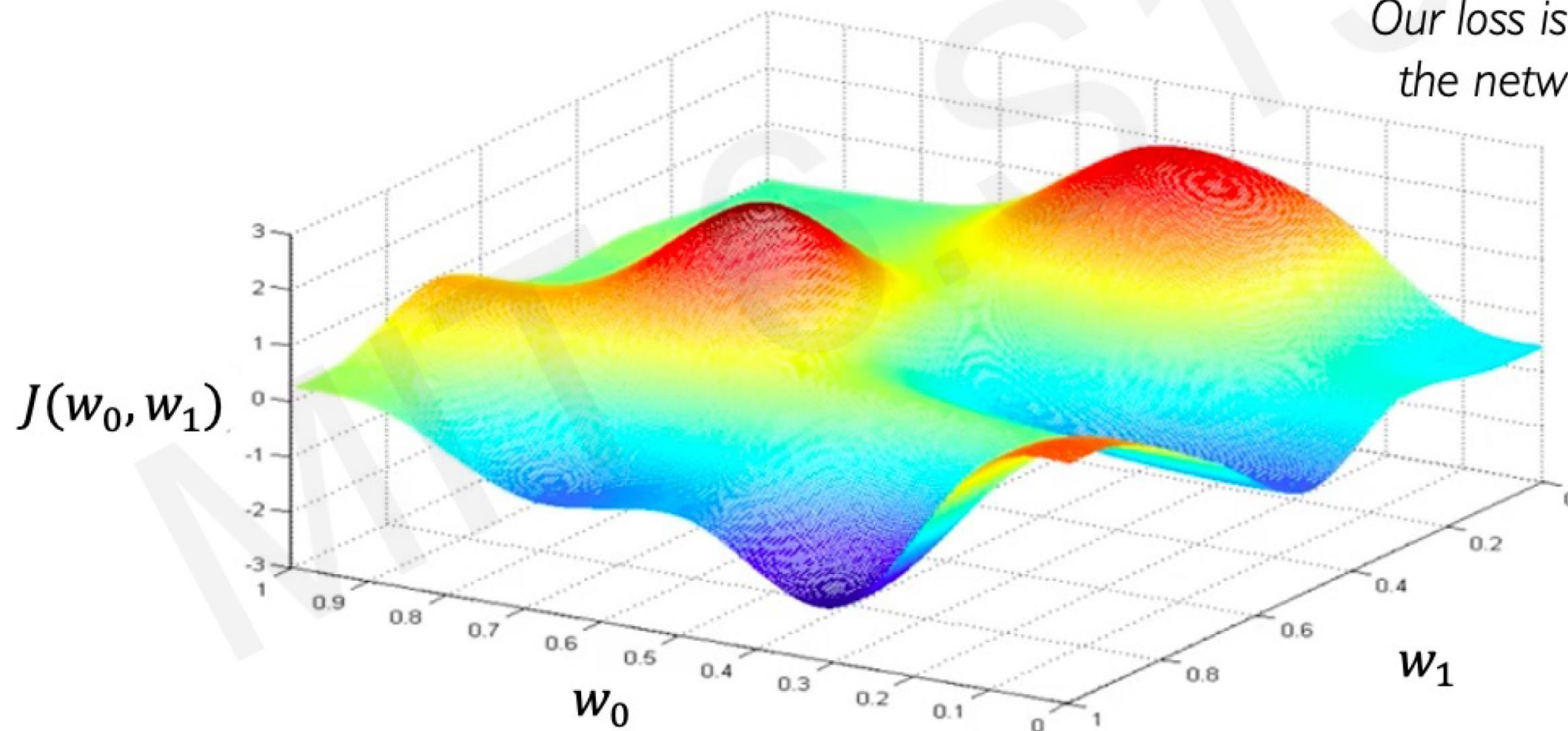
Remember:

$$\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \dots\}$$

Loss Optimization

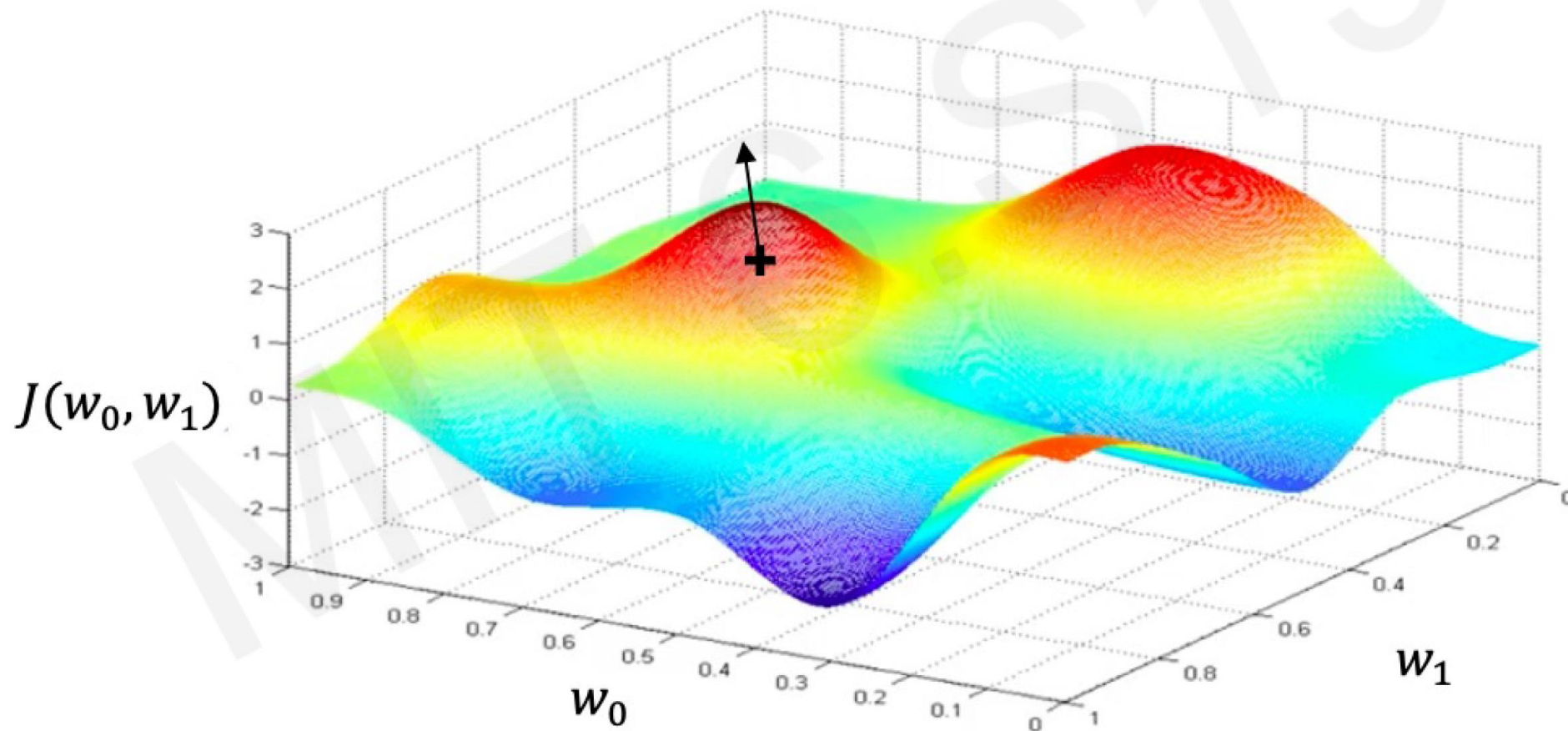
$$W^* = \operatorname{argmin}_W J(W)$$

Remember:
*Our loss is a function of
the network weights!*



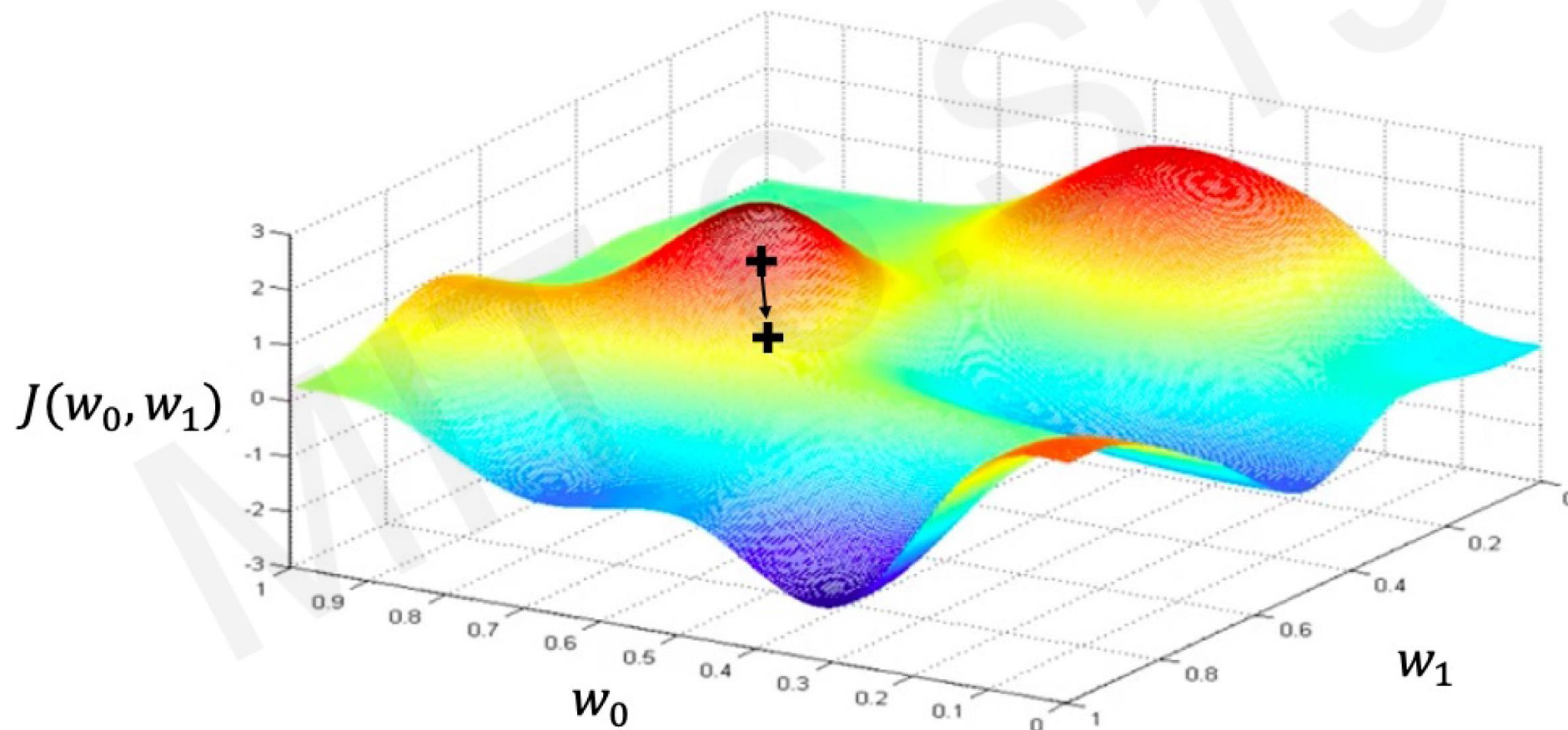
Loss Optimization

Compute gradient, $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$



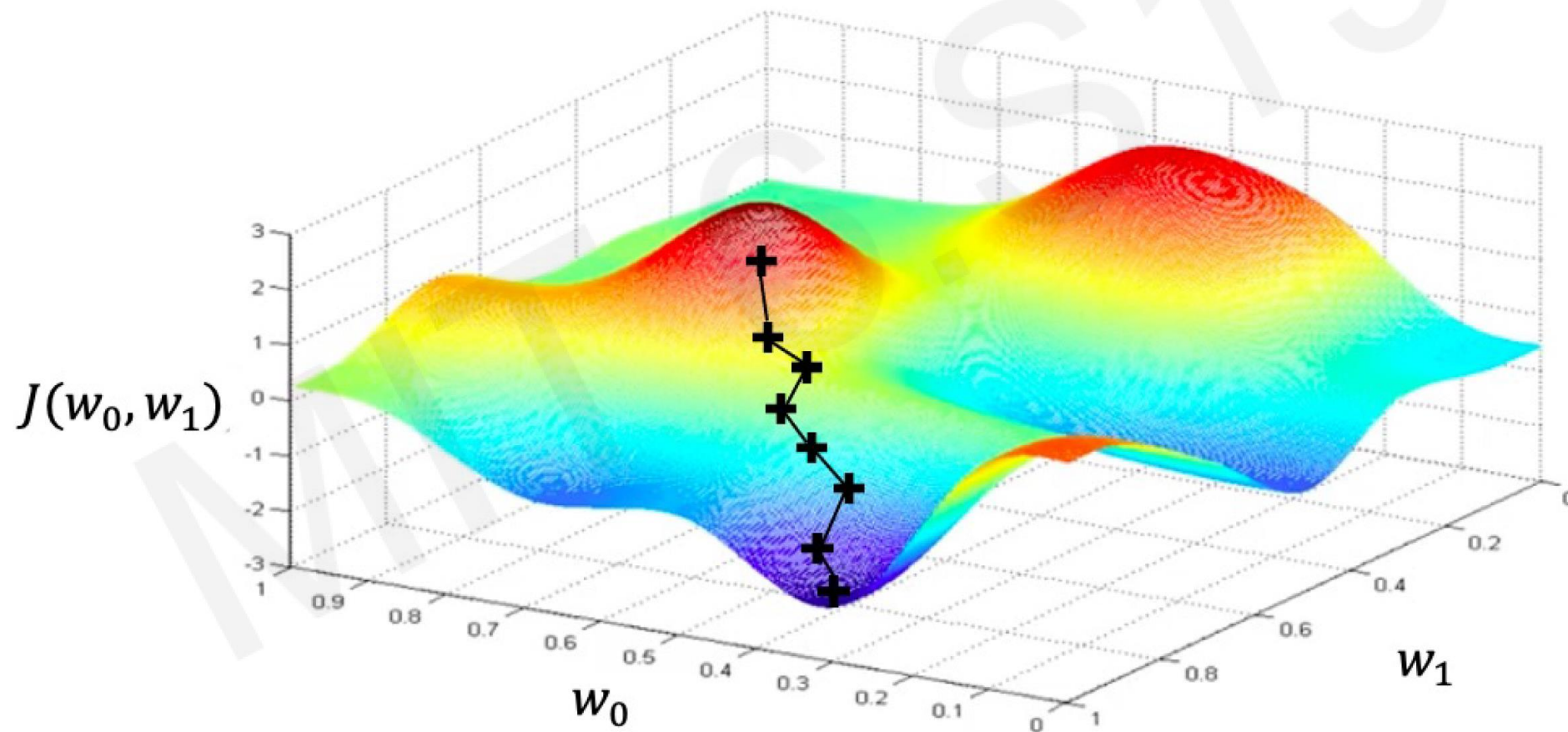
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence

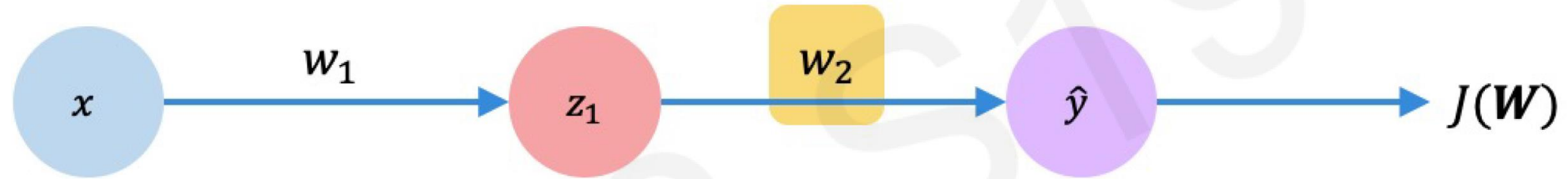


Gradient Descent

Algorithm

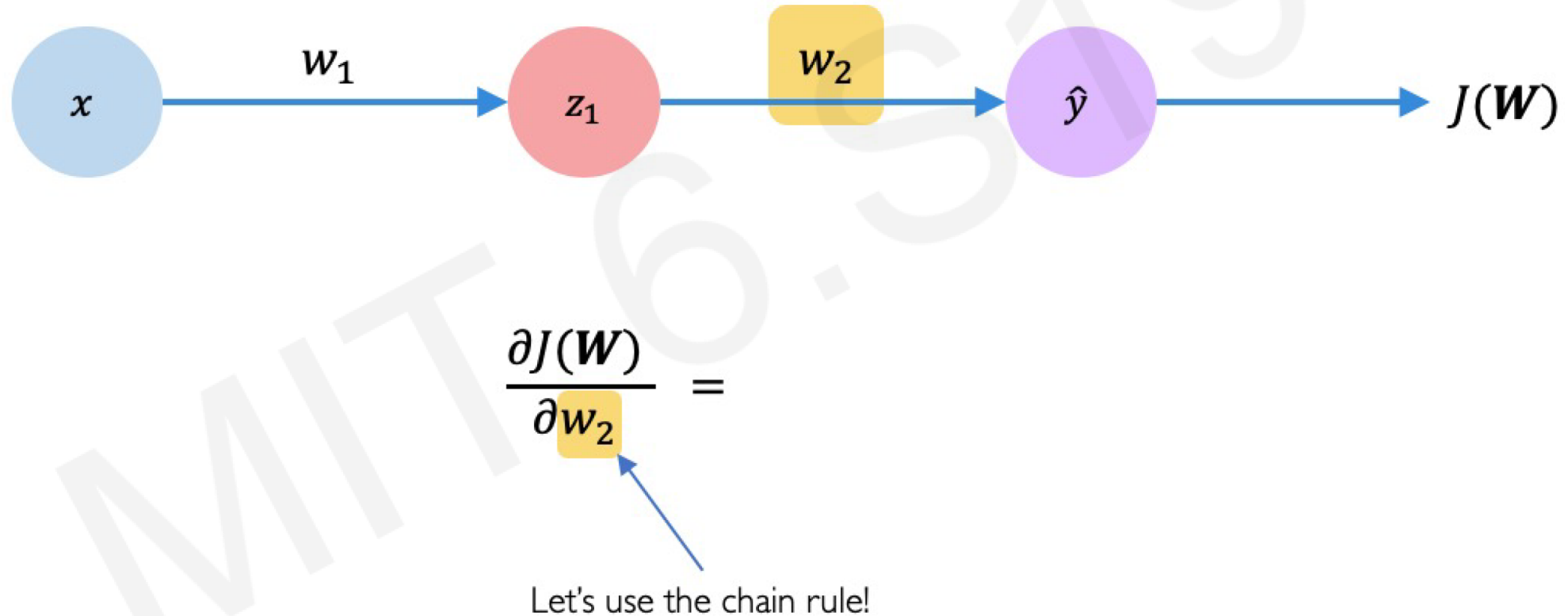
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

Computing Gradients: Backpropagation

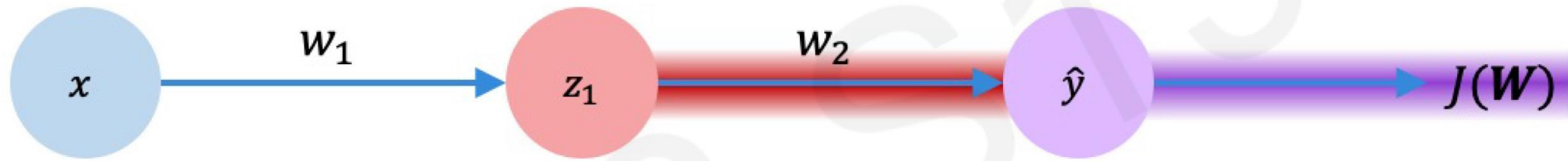


How does a small change in one weight (ex. w_2) affect the final loss $J(\mathbf{W})$?

Computing Gradients: Backpropagation

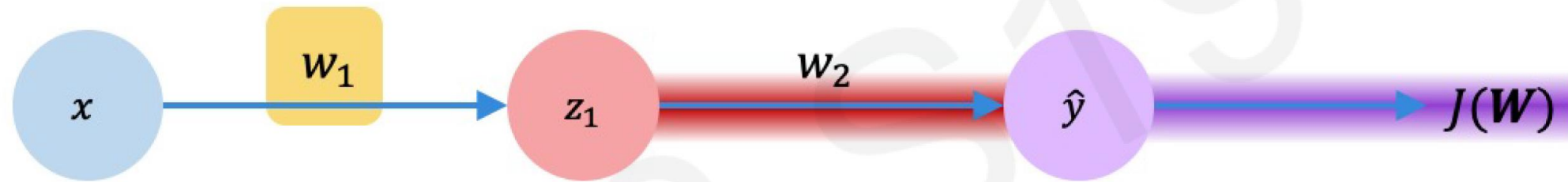


Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_2} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial w_2}}_{\text{red}}$$

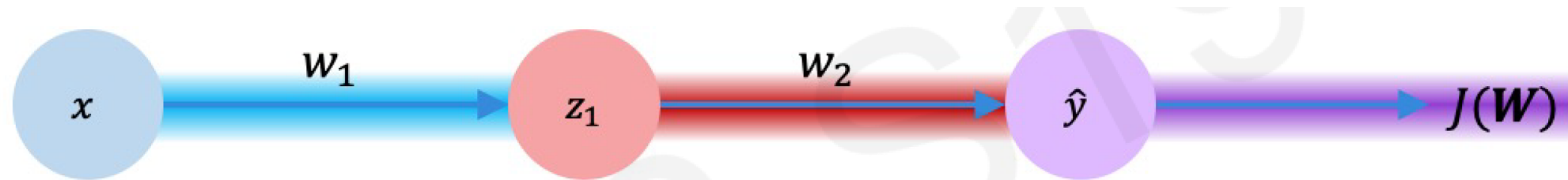
Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule! Apply chain rule!

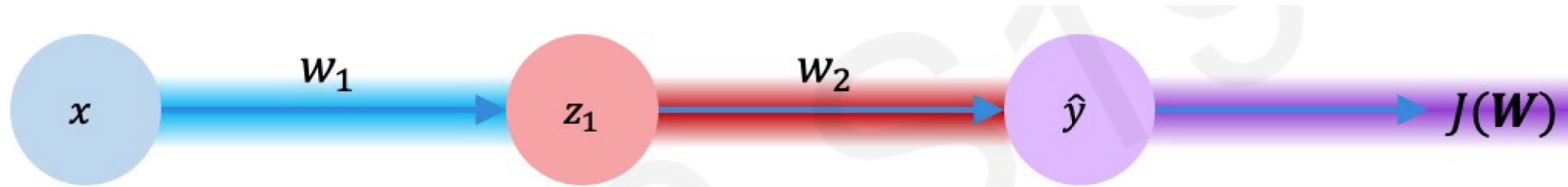
Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$



Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

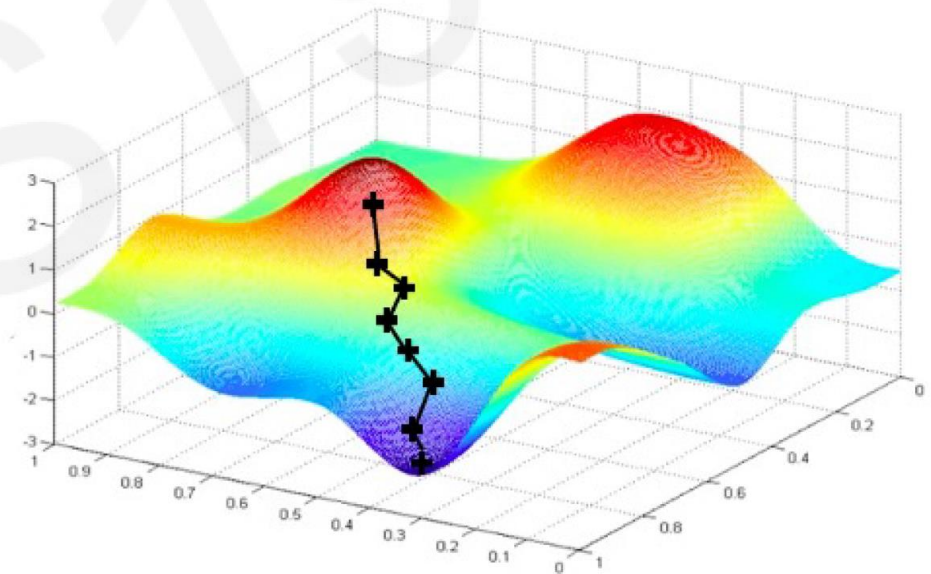


Repeat this for **every weight in the network** using gradients from later layers

Gradient Descent

Algorithm

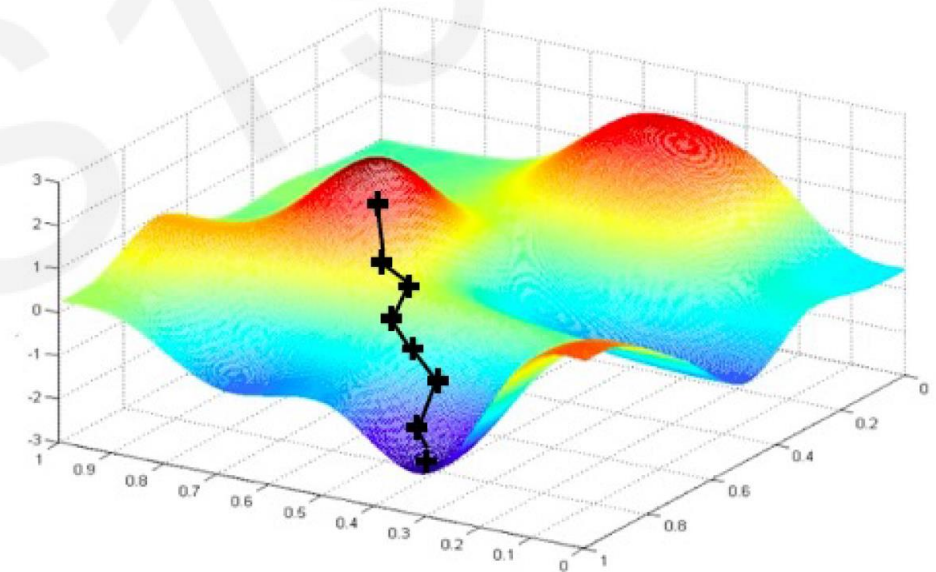
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

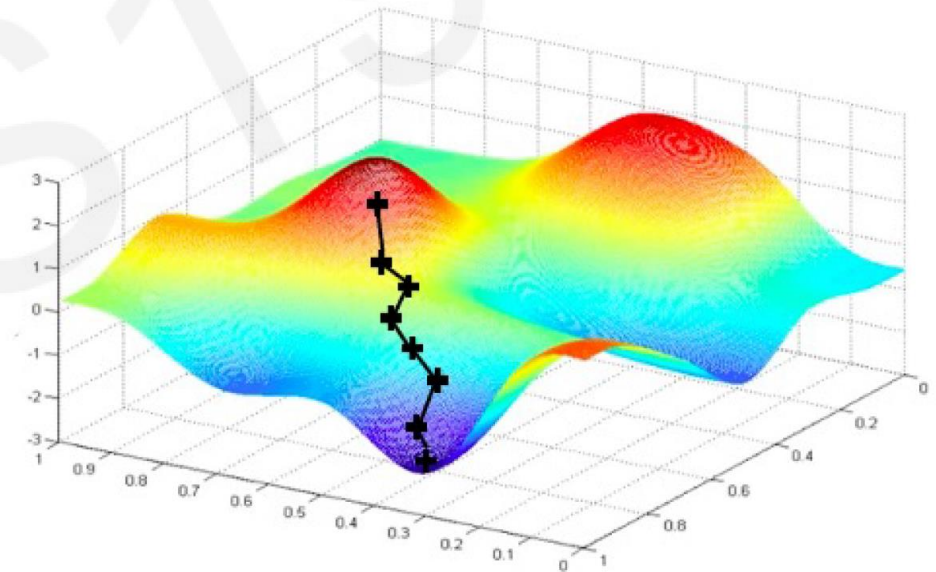


Can be very
computationally
intensive to compute!

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\mathbf{w})}{\partial \mathbf{w}}$
5. Update weights, $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$
6. Return weights

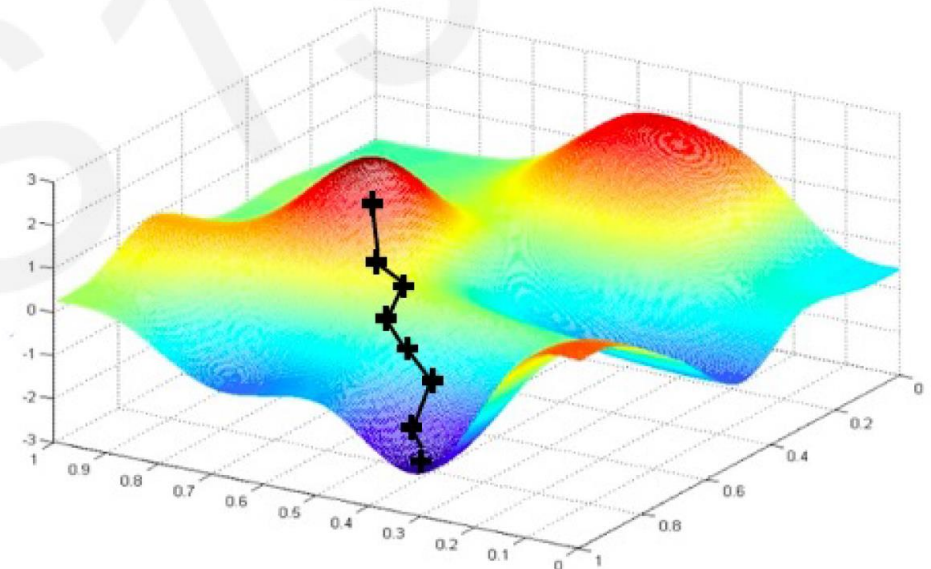


Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights

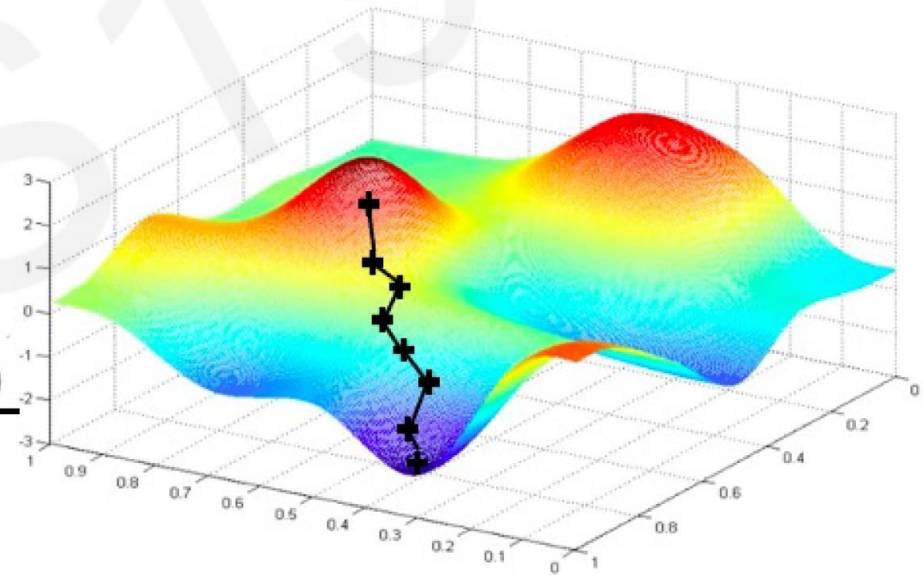
Easy to compute but
very noisy (stochastic)!



Stochastic Gradient Descent

Algorithm

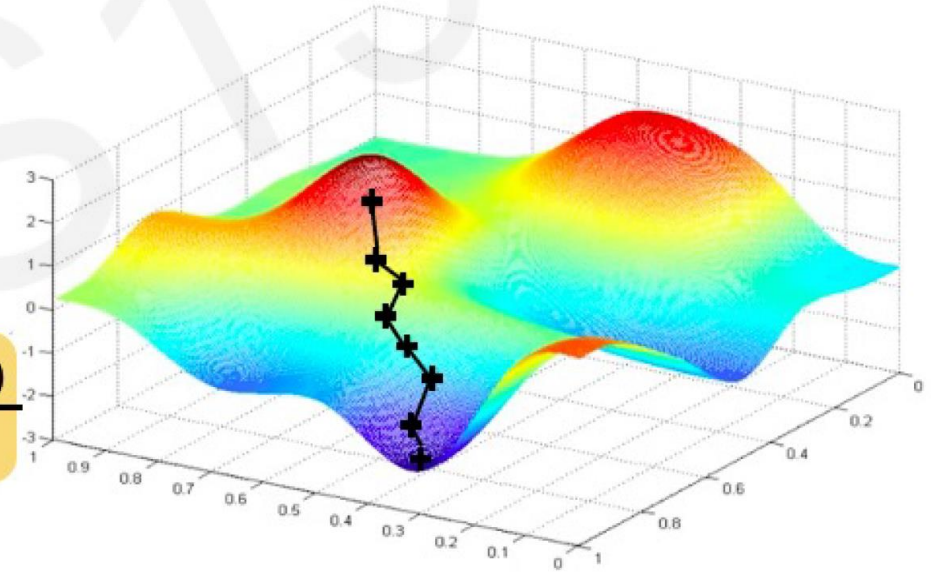
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

Mini-Batches while Training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-Batches while Training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

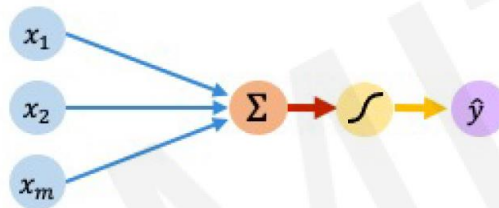
Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

Neural Networks Summary

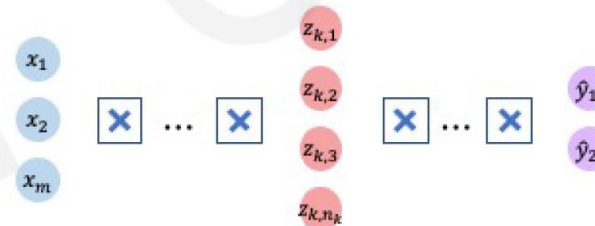
The Perceptron

- Structural building blocks
- Nonlinear activation functions



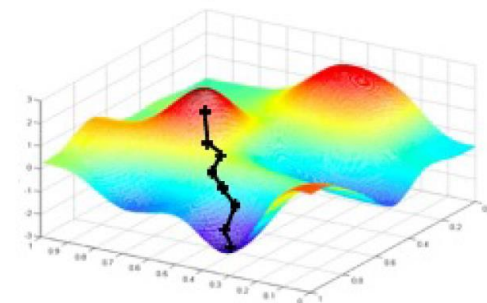
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Deep Learning for Computer Vision

Images are Numbers



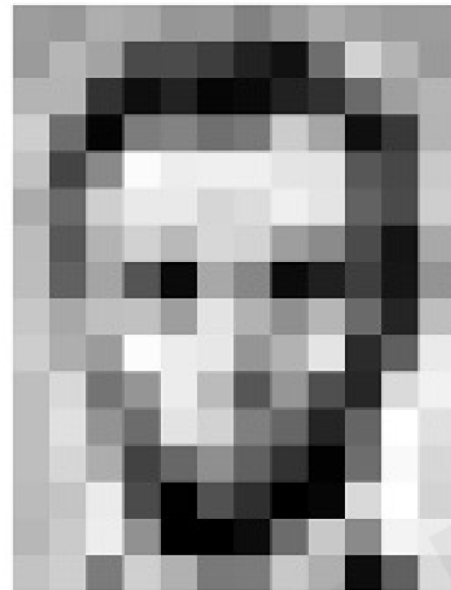
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

What the computer sees

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	106	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

An image is just a matrix of numbers $[0,255]$!
i.e., $1080 \times 1080 \times 3$ for an RGB image

Tasks in Computer Vision



Input Image



167	163	174	168	160	162	129	161	172	161	165	166
165	162	163	74	75	62	33	17	113	216	180	164
180	180	50	14	34	6	10	33	48	106	165	181
206	109	5	124	131	111	120	204	166	16	56	180
164	68	137	261	237	239	239	237	87	71	201	
172	106	207	233	233	214	200	239	238	94	74	206
188	88	179	208	185	216	211	168	139	75	20	169
189	97	165	84	16	168	134	11	31	63	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	195	252	236	231	149	179	228	43	95	204
190	216	116	148	236	187	85	190	73	38	218	241
190	204	147	108	227	210	127	102	36	101	265	204
190	214	173	66	103	143	96	60	3	106	249	216
187	195	235	75	1	81	47	0	6	217	265	211
183	202	237	145	0	0	12	108	209	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Pixel Representation

classification

Lincoln

Washington

Jefferson

Obama

$$\begin{bmatrix} 0.8 \\ 0.1 \\ 0.05 \\ 0.05 \end{bmatrix}$$

- **Regression:** output variable takes continuous value
- **Classification:** output variable takes class label. Can produce probability of belonging to a particular class

Manual Feature Extraction

Domain knowledge

Define features

Detect features
to classify

Viewpoint variation



Scale variation



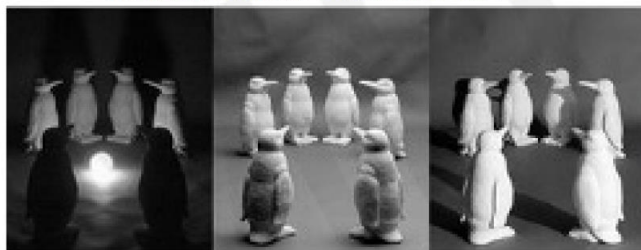
Deformation



Occlusion



Illumination conditions



Background clutter



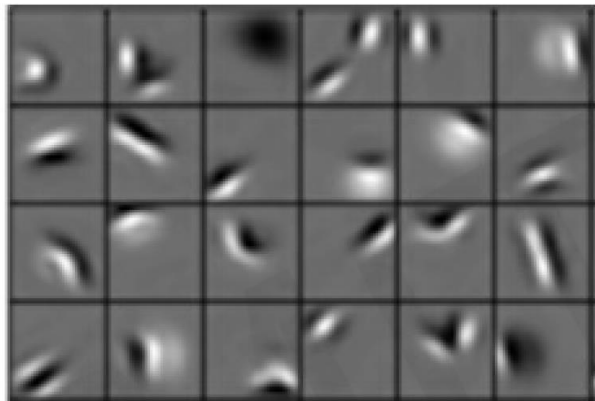
Intra-class variation



Learning Feature Representations

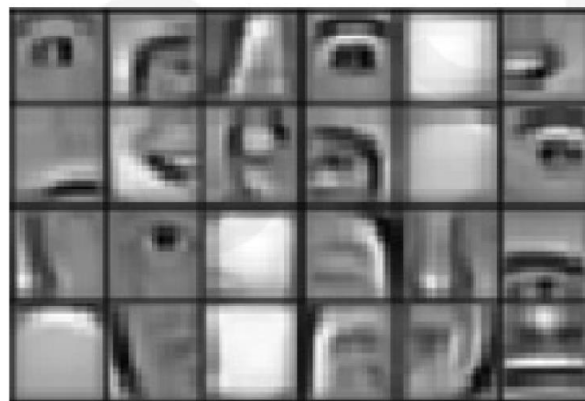
Can we learn a **hierarchy of features** directly from the data instead of hand engineering?

Low level features



Edges, dark spots

Mid level features



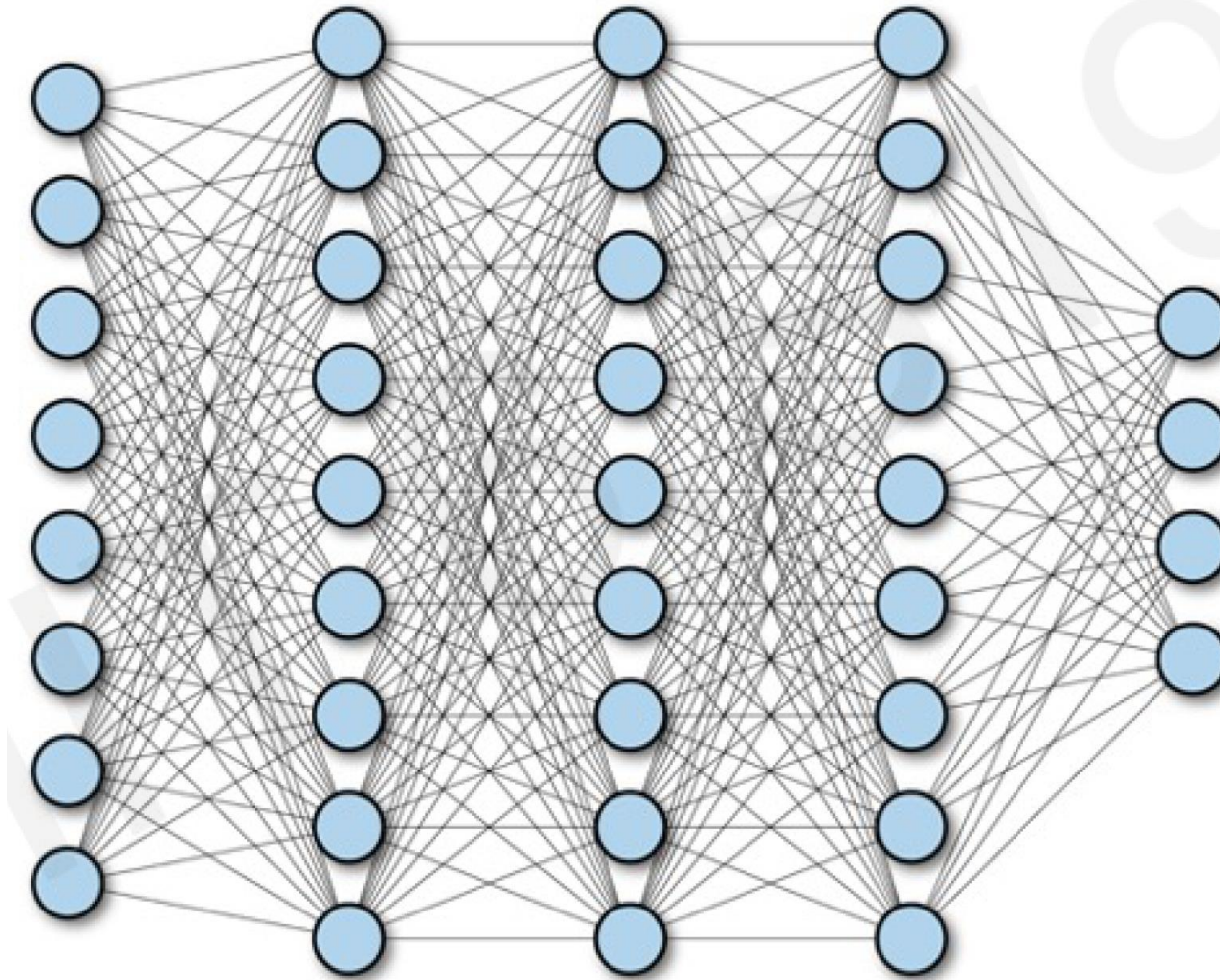
Eyes, ears, nose

High level features



Facial structure

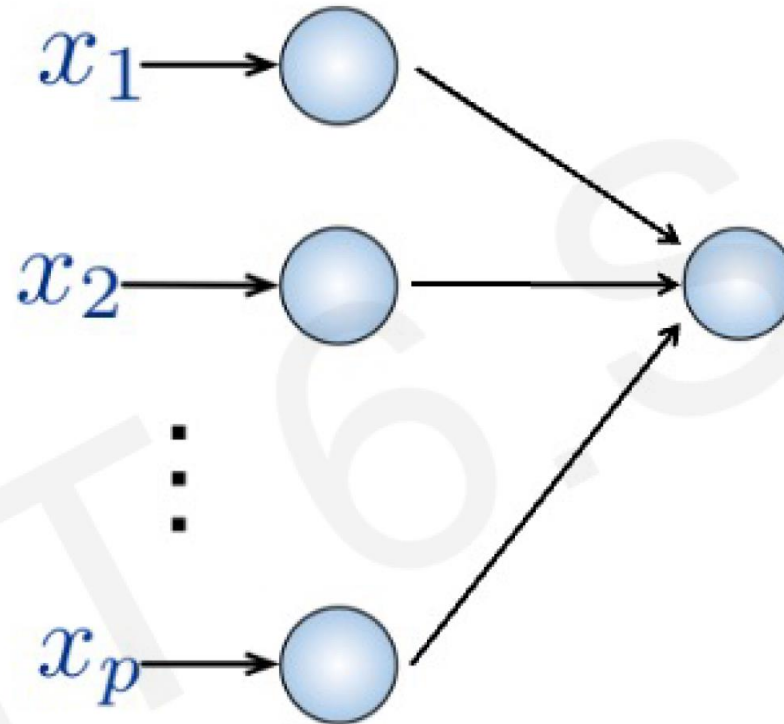
Fully Connected Neural Network



Fully Connected Neural Network

Input:

- 2D image
- Vector of pixel values



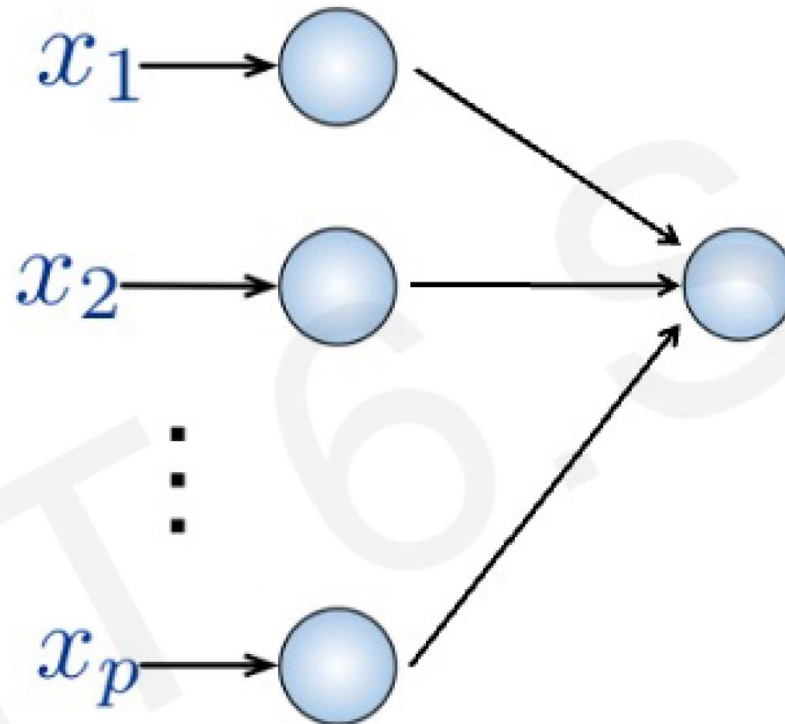
Fully Connected:

- Connect neuron in hidden layer to all neurons in input layer
- No spatial information!
- And many, many parameters!

Fully Connected Neural Network

Input:

- 2D image
- Vector of pixel values



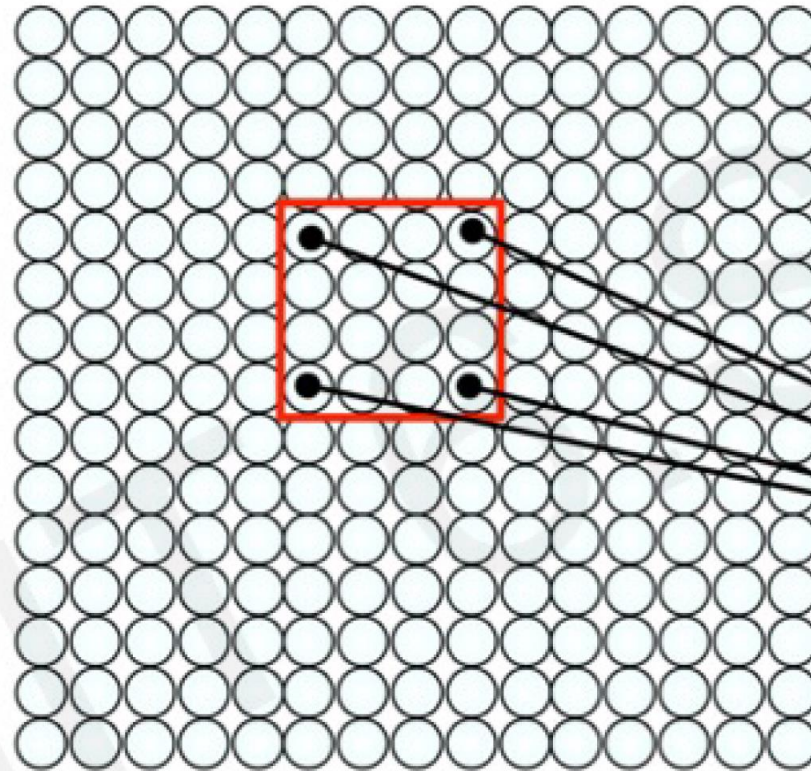
Fully Connected:

- Connect neuron in hidden layer to all neurons in input layer
- No spatial information!
- And many, many parameters!

How can we use **spatial structure** in the input to inform the architecture of the network?

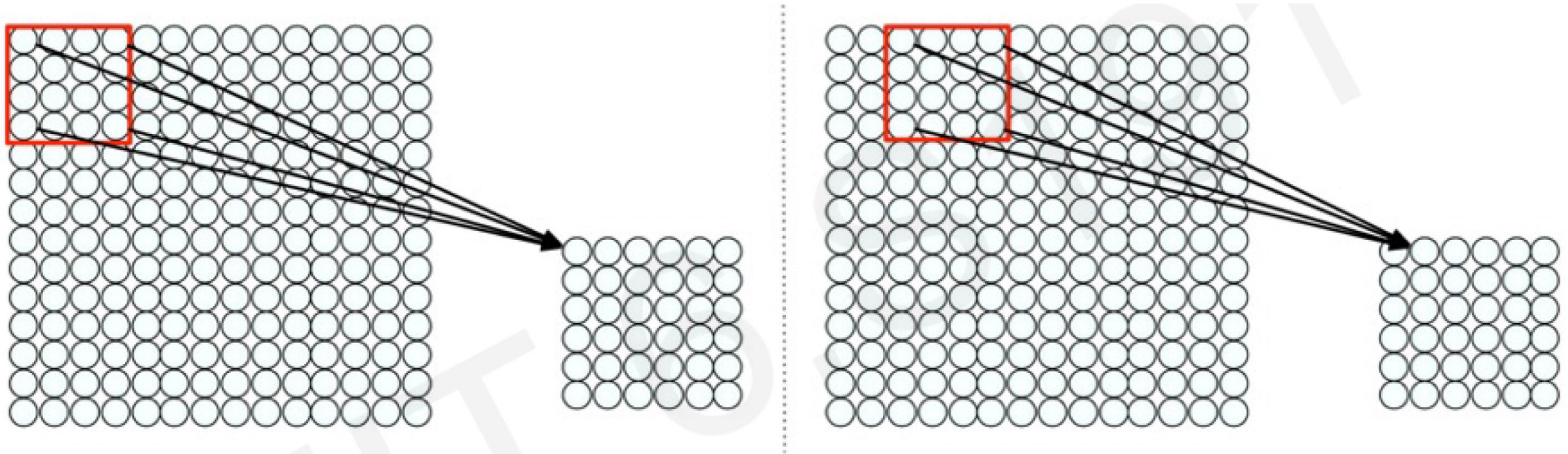
Using Spatial Structure

Input: 2D image.
Array of pixel values



Idea: connect patches of input
to neurons in hidden layer.
Neuron connected to region of
input. Only “sees” these values.

Using Spatial Structure



Connect patch in input layer to a single neuron in subsequent layer.

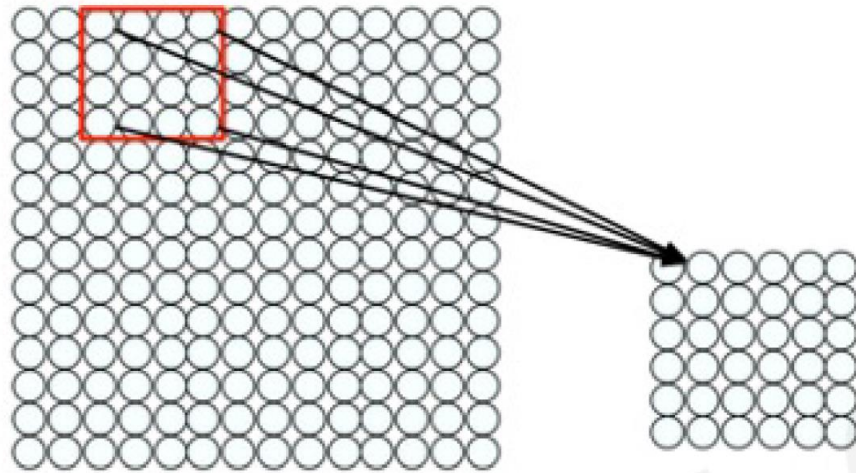
Use a sliding window to define connections.

How can we **weight** the patch to detect particular features?

Applying Filters to Extract Features

- 1) Apply a set of weights – a filter – to extract **local features**
- 2) Use **multiple filters** to extract different features
- 3) Spatially **share** parameters of each filter
(features that matter in one part of the input should matter elsewhere)

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This “patchy” operation is **convolution**

- 1) Apply a set of weights – a filter – to extract **local features**
- 2) Use **multiple filters** to extract different features
- 3) **Spatially share** parameters of each filter

X?

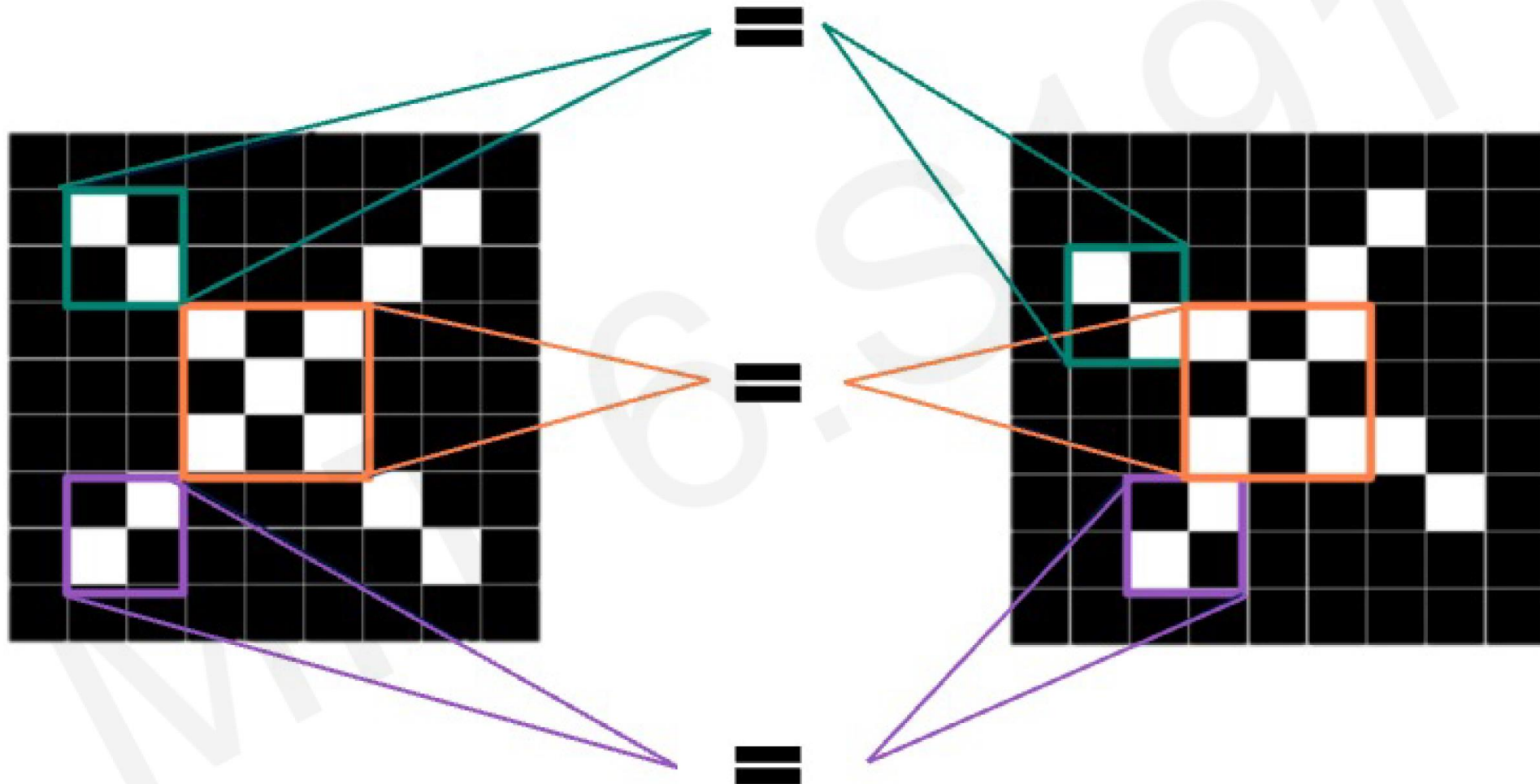
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1



-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	1	-1	-1	-1
-1	-1	1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	1	-1	-1
-1	-1	-1	1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Image is represented as matrix of pixel values... and computers are literal!
 We want to be able to classify an X as an X even if it's shifted, shrunk, rotated, deformed.

Features of X



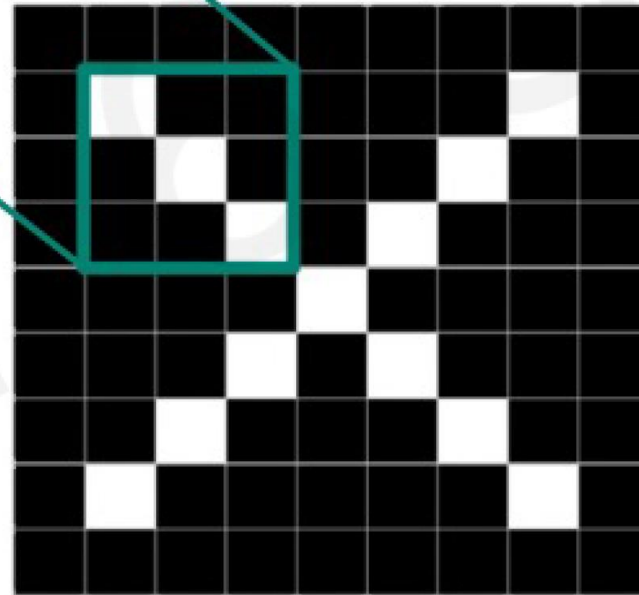
Filters to Detect Features

filters

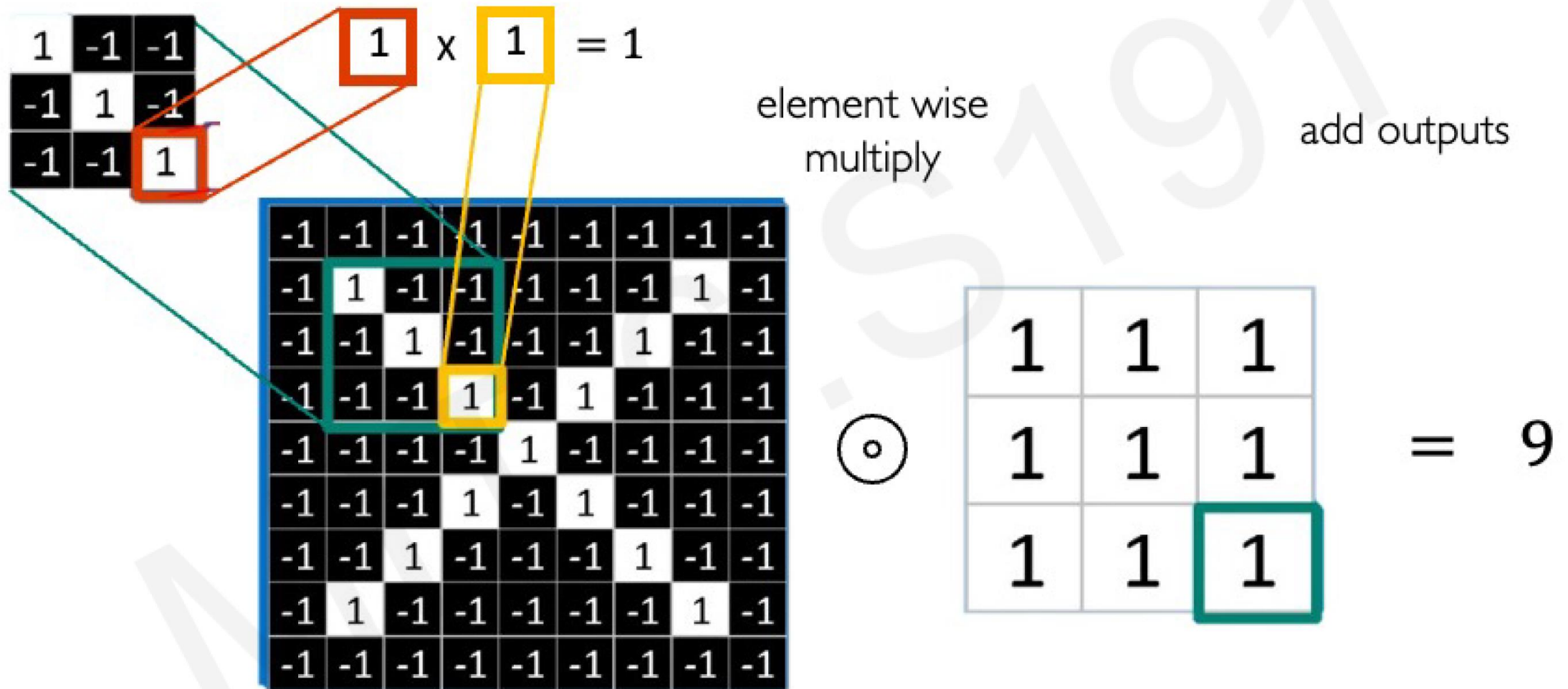
1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

-1	-1	1
-1	1	-1
1	-1	-1

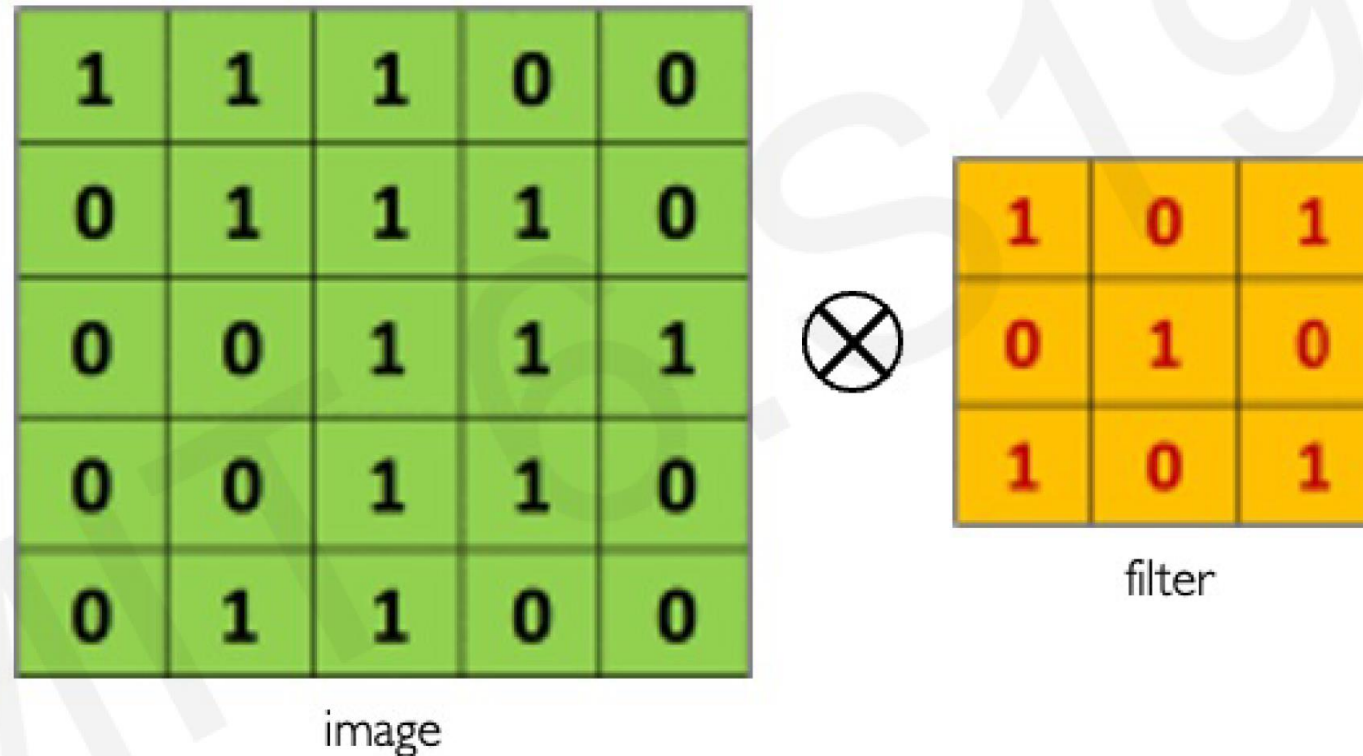


The Convolution Operation



The Convolution Operation

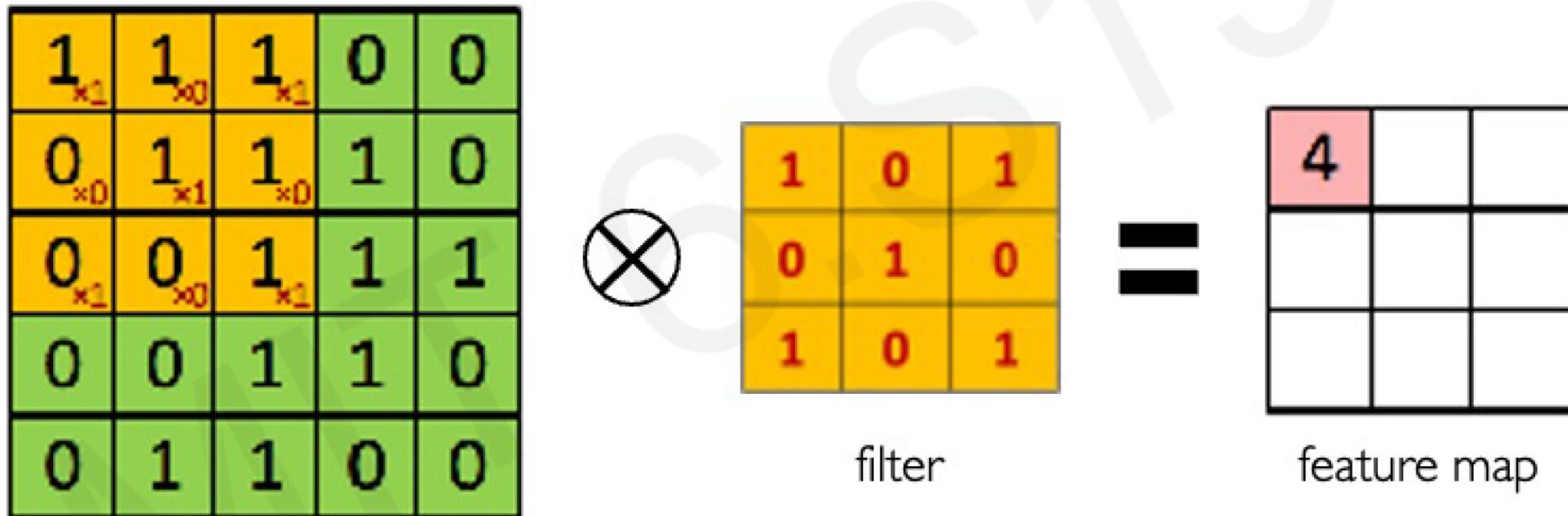
Suppose we want to compute the convolution of a 5x5 image and a 3x3 filter:



We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs...

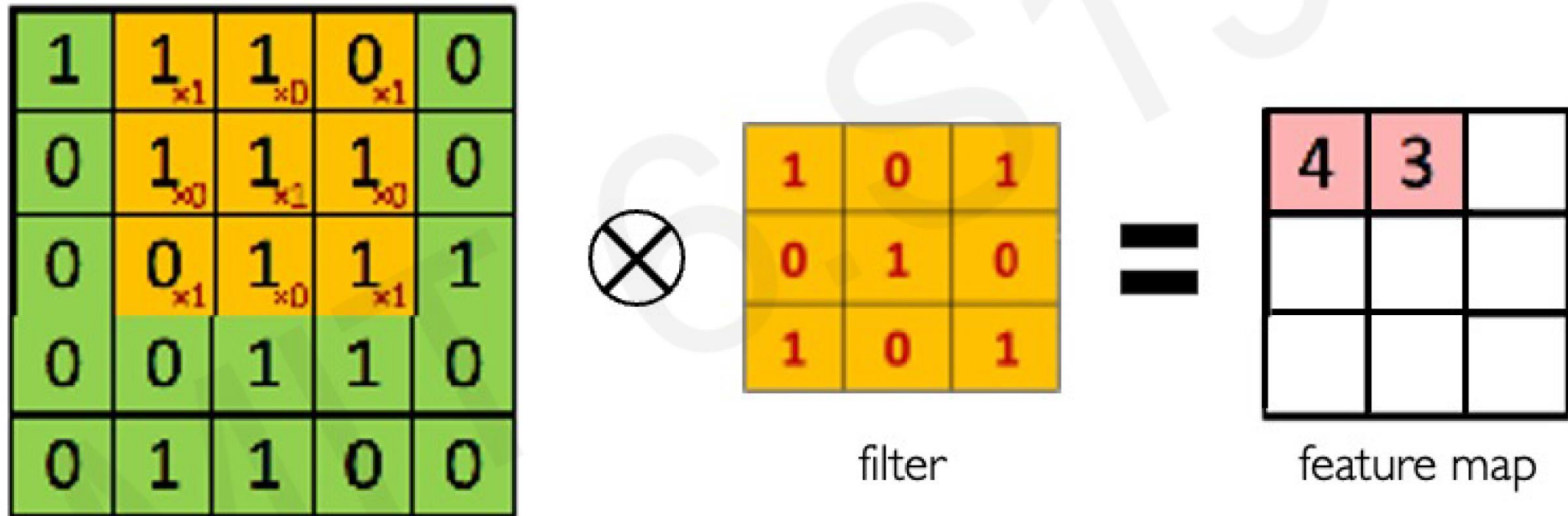
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



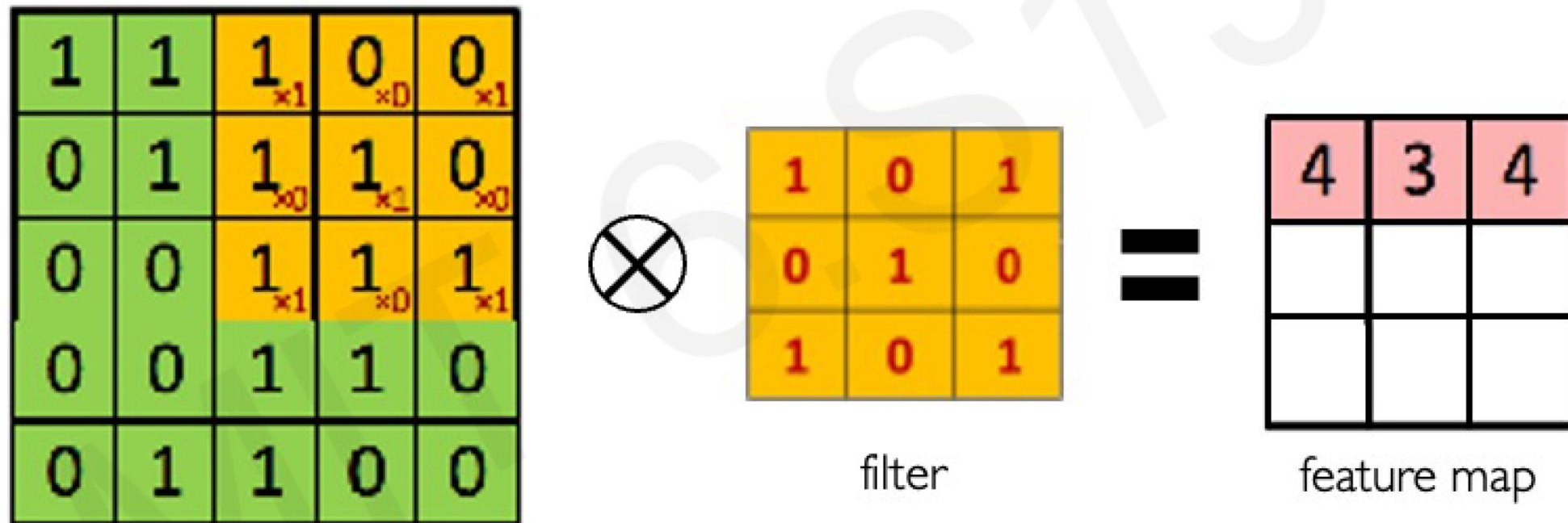
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



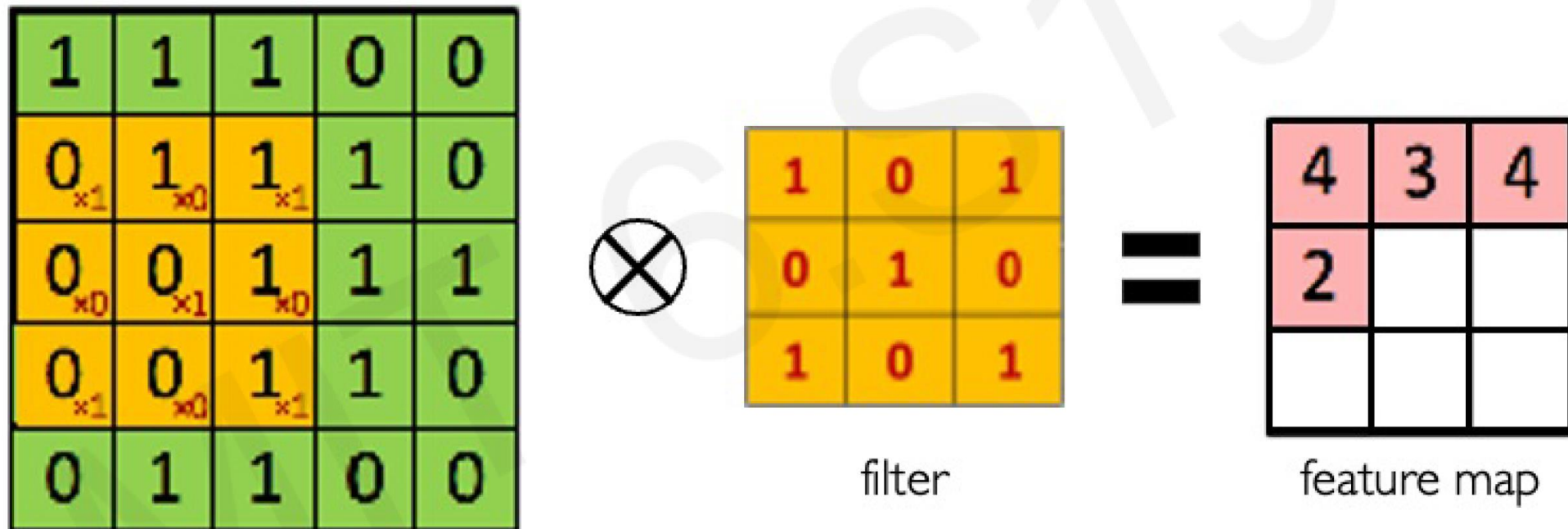
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



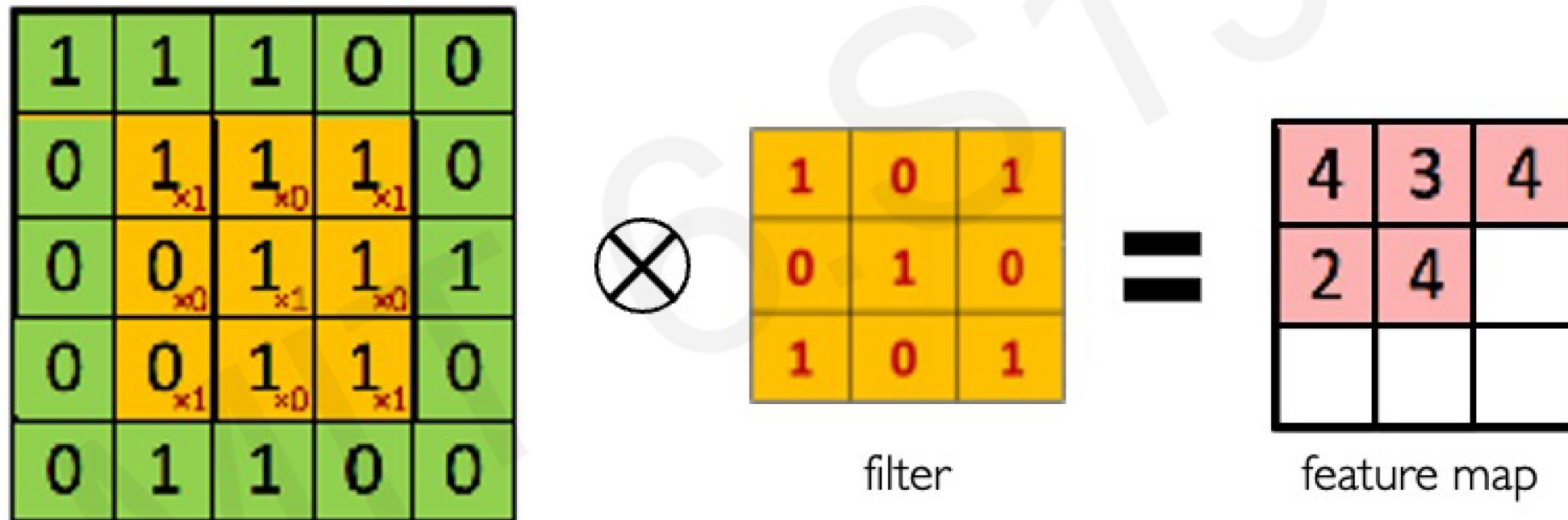
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



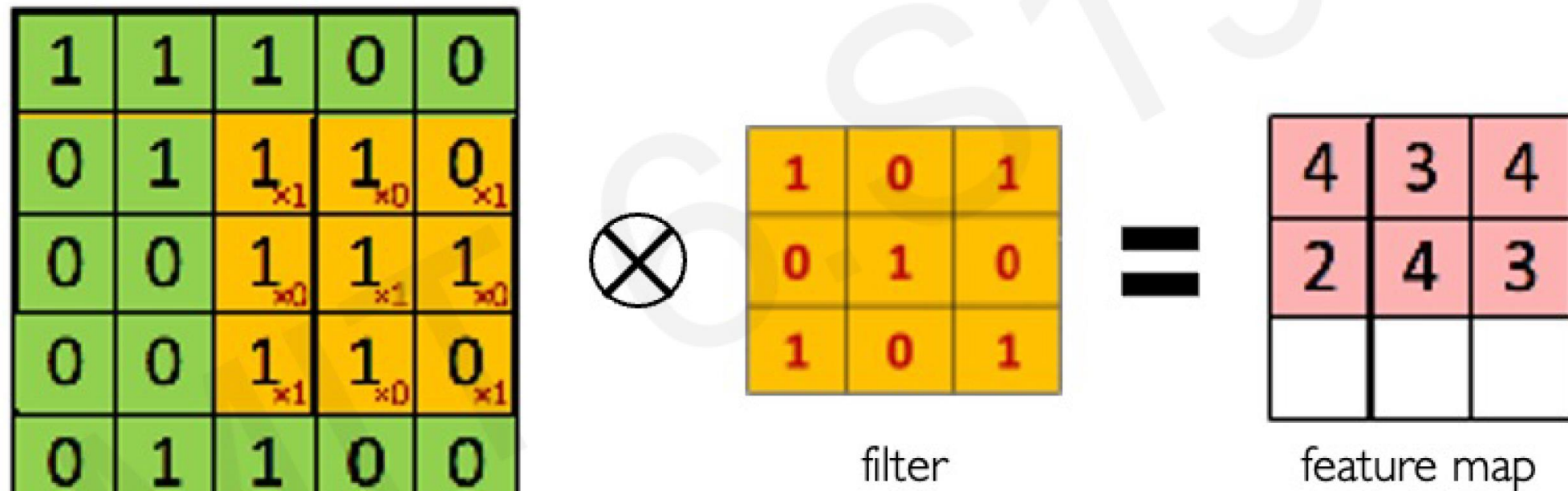
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



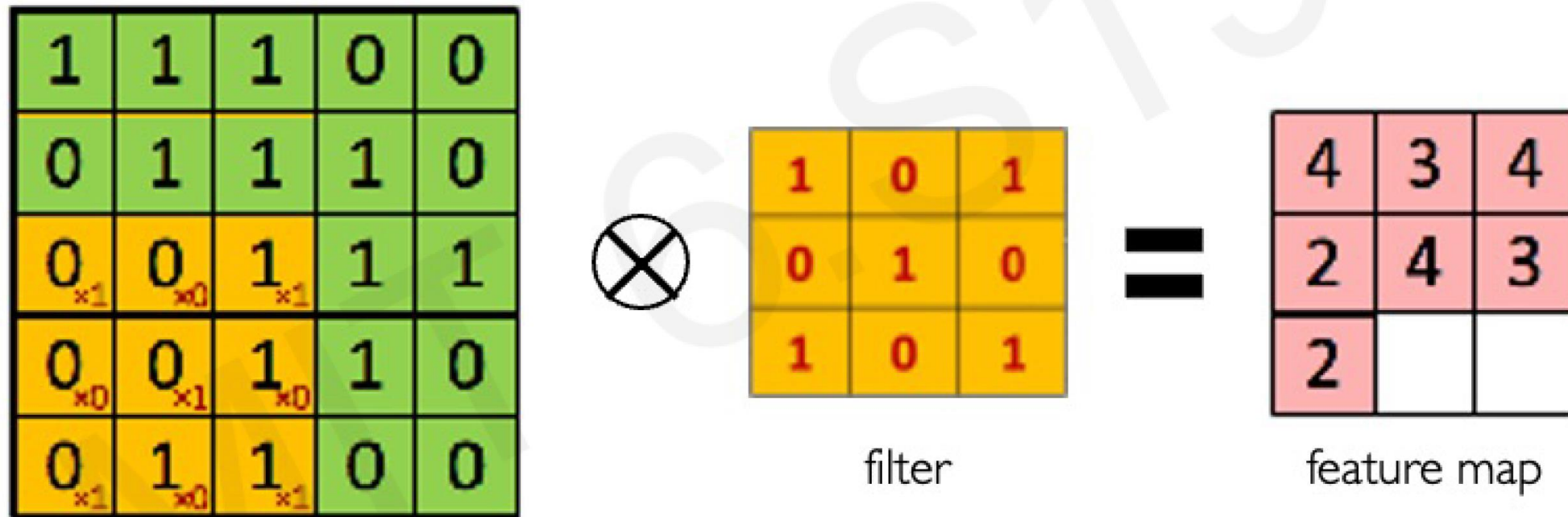
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



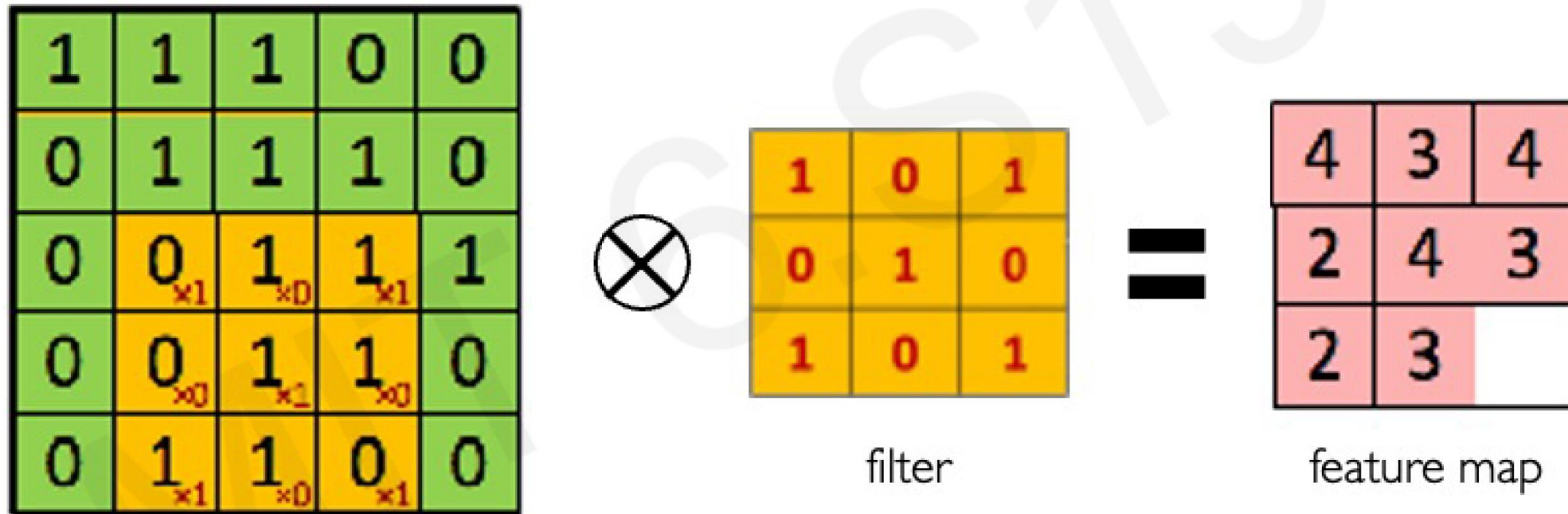
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



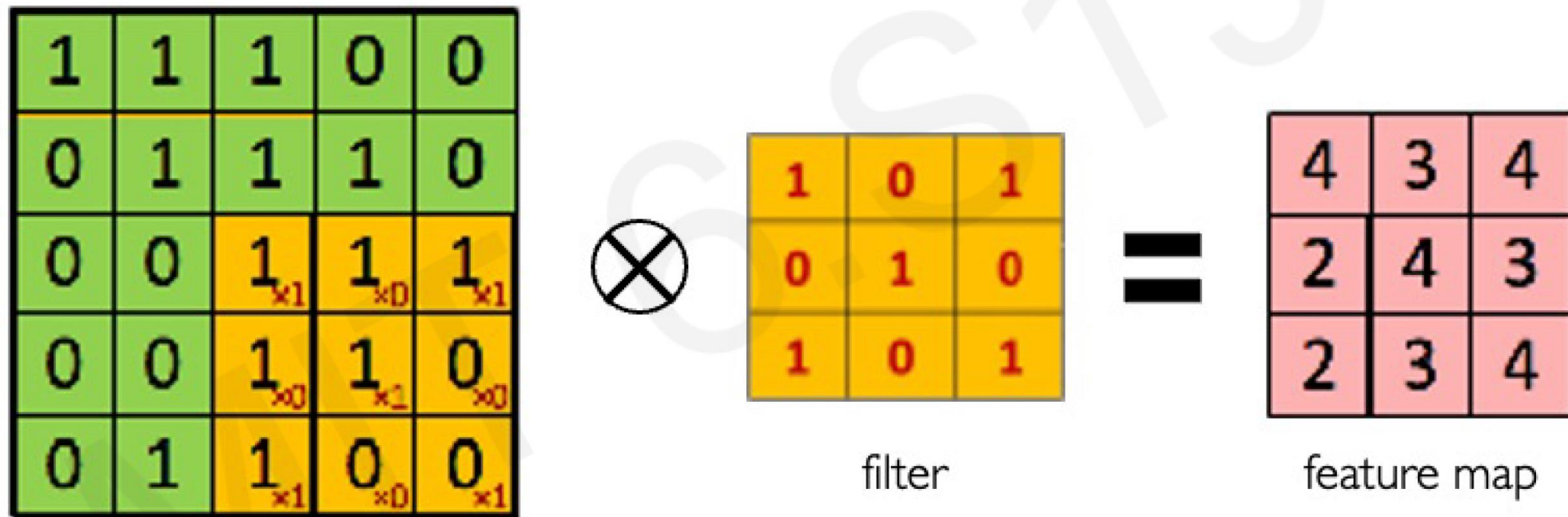
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



Convolutional Neural Networks

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

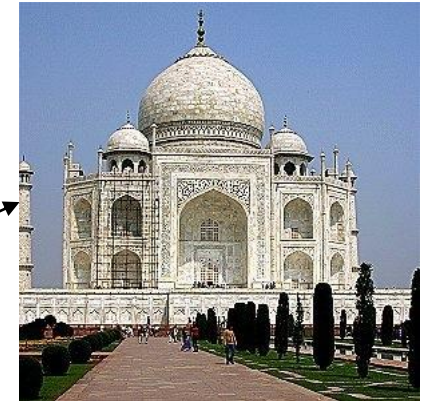
Image

4		

Convolved
Feature

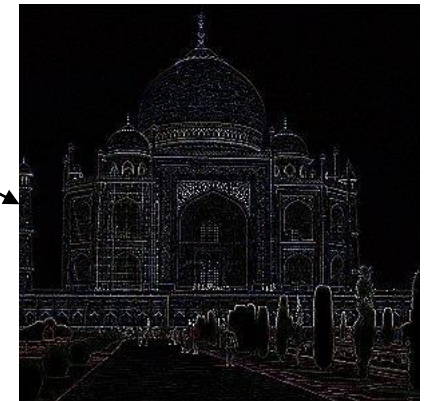
sharpen

0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0

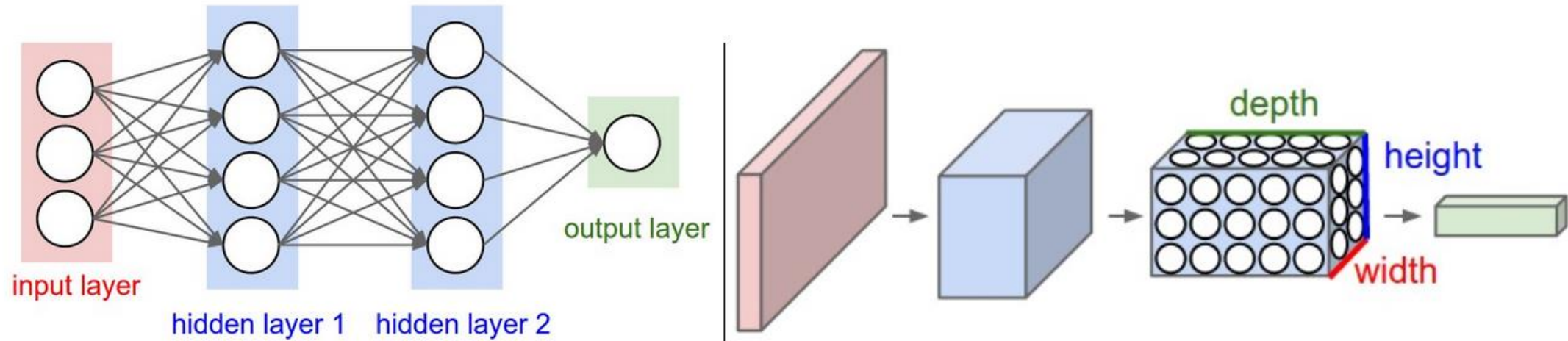


	0	1	0	
	1	-4	1	
	0	1	0	

detect edges

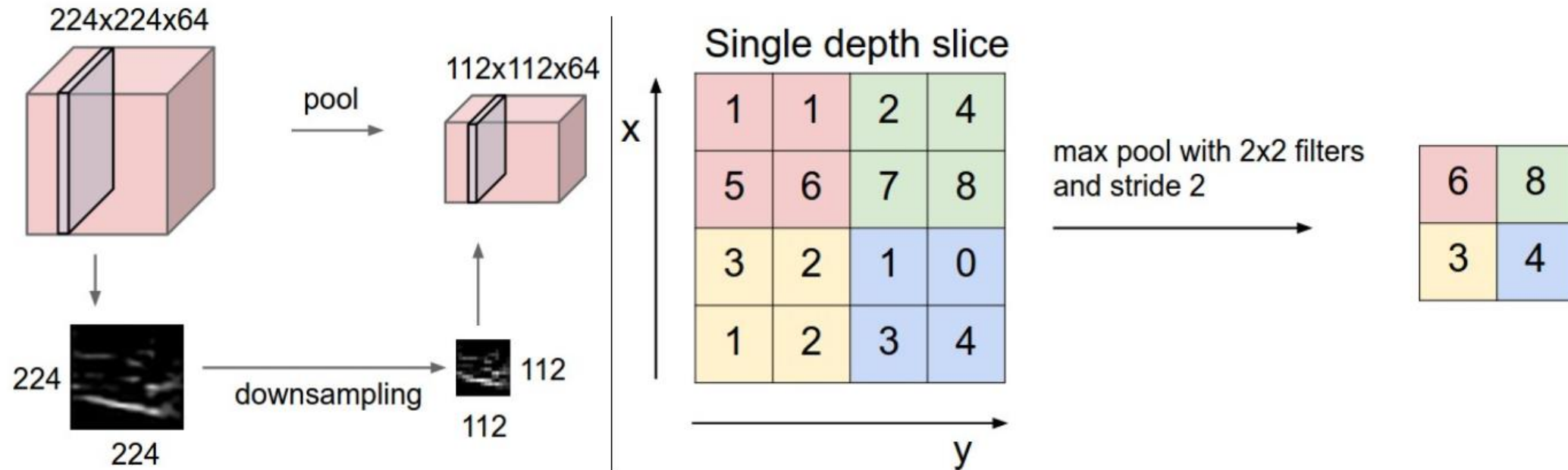


Convolutional Neural Networks



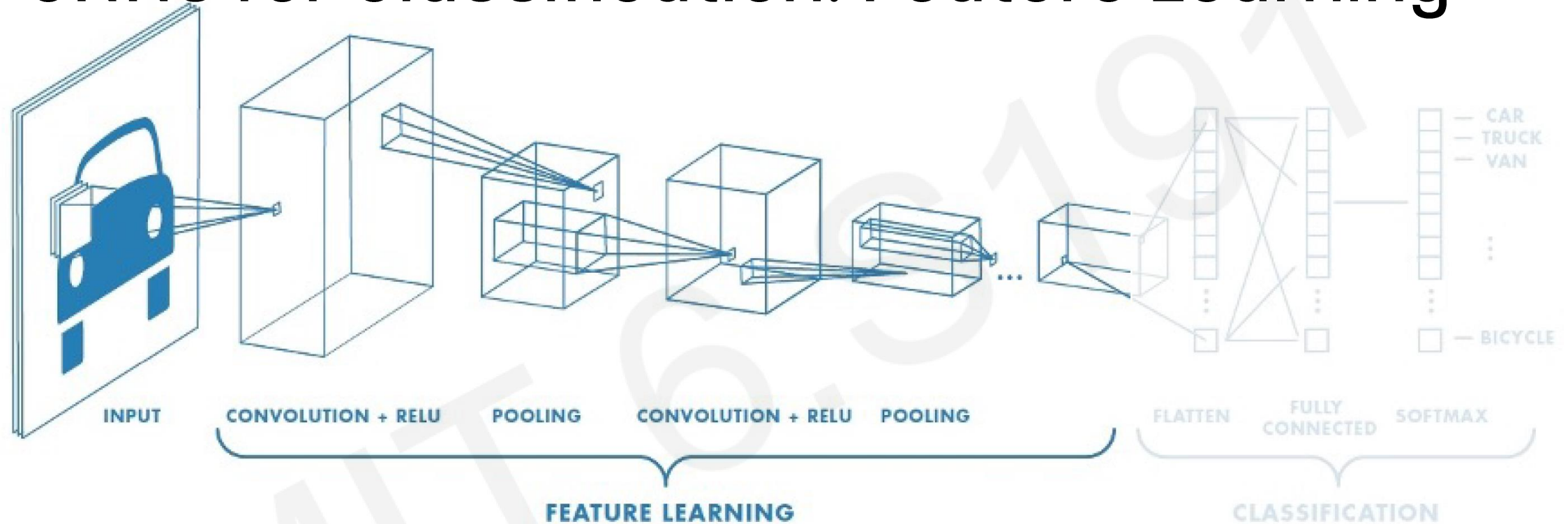
Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Convolutional Neural Networks



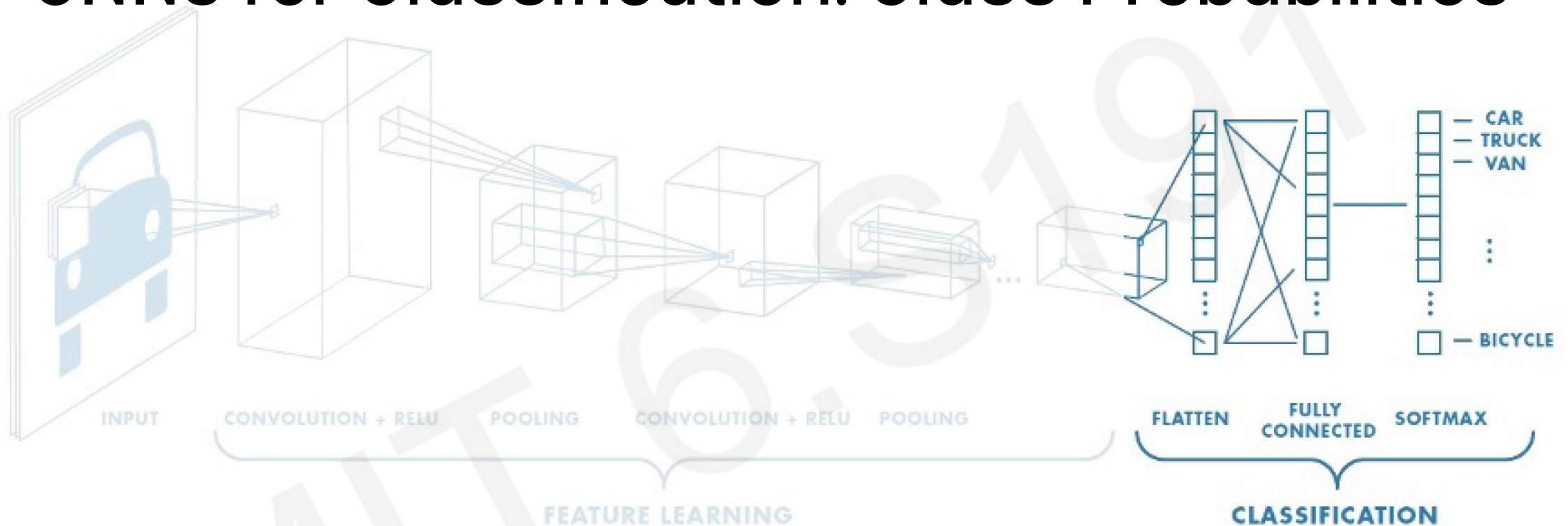
Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size $[224 \times 224 \times 64]$ is pooled with filter size 2, stride 2 into output volume of size $[112 \times 112 \times 64]$. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2×2 square).

CNNs for Classification: Feature Learning



1. Learn features in input image through **convolution**
2. Introduce **non-linearity** through activation function (real-world data is non-linear!)
3. Reduce dimensionality and preserve spatial invariance with **pooling**

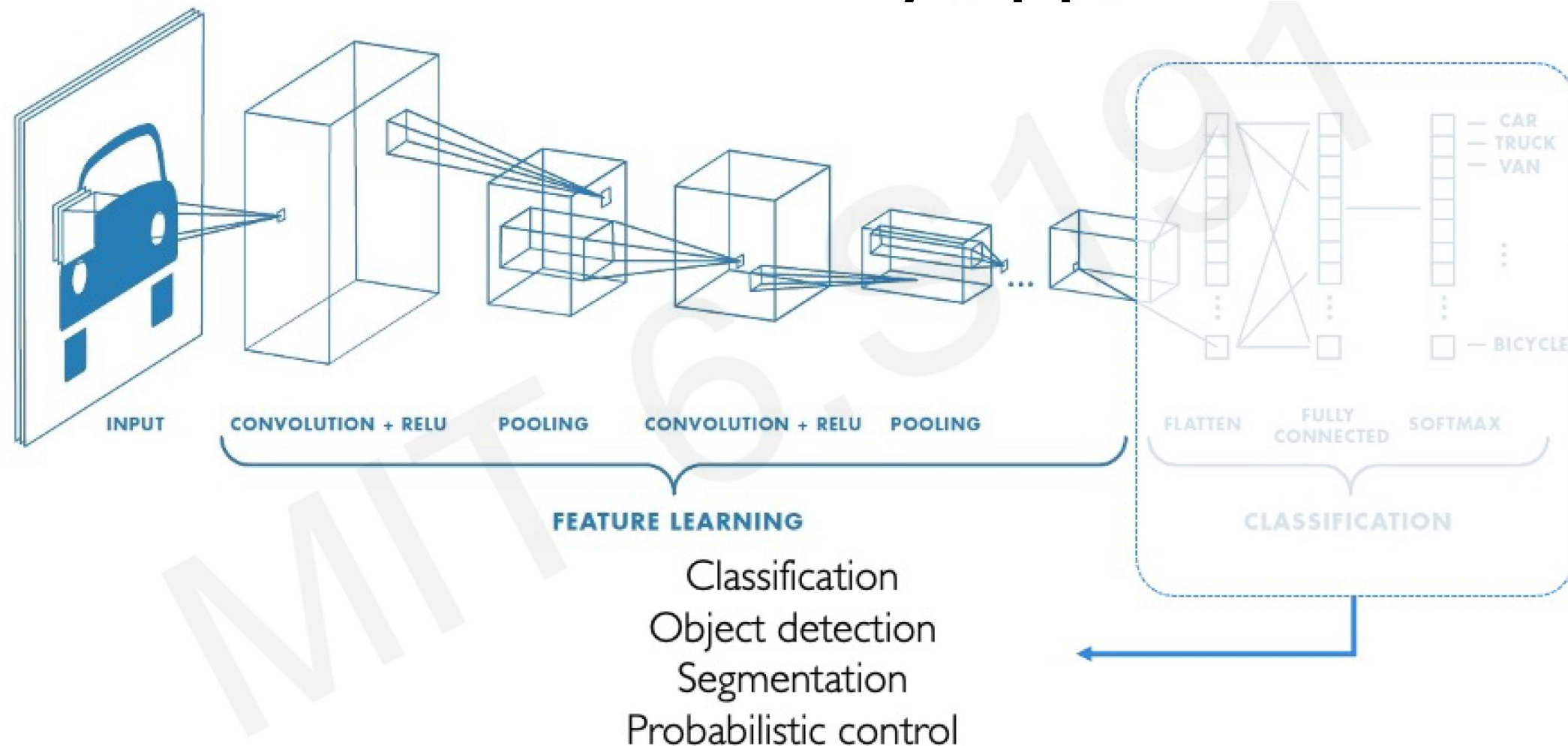
CNNs for Classification: Class Probabilities



- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

An Architecture for Many Applications



Supervised vs Unsupervised

Supervised Learning

Data: (x, y)

x is data, y is label

Goal: Learn function to map
 $x \rightarrow y$

Examples: Classification, regression, object detection, semantic segmentation, etc.

Unsupervised Learning

Data: x

x is data, no labels!

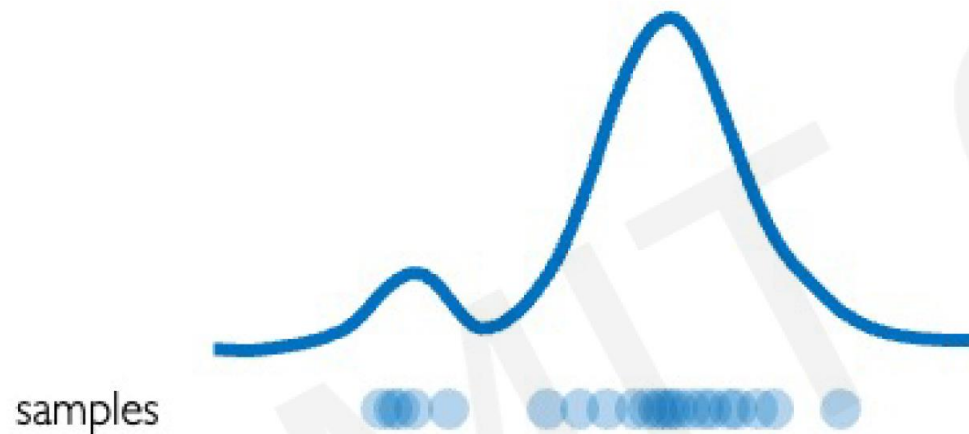
Goal: Learn the *hidden* or *underlying structure* of the data

Examples: Clustering, feature or dimensionality reduction, etc.

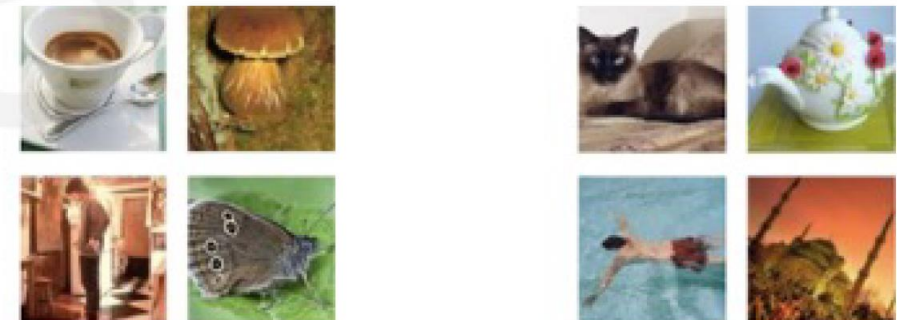
Generative Modeling

Goal: Take as input training samples from some distribution and learn a model that represents that distribution

Density Estimation



Sample Generation



Input samples

Generated samples

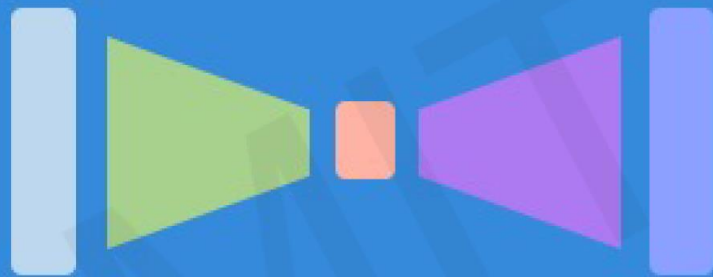
Training data $\sim P_{data}(x)$

Generated $\sim P_{model}(x)$

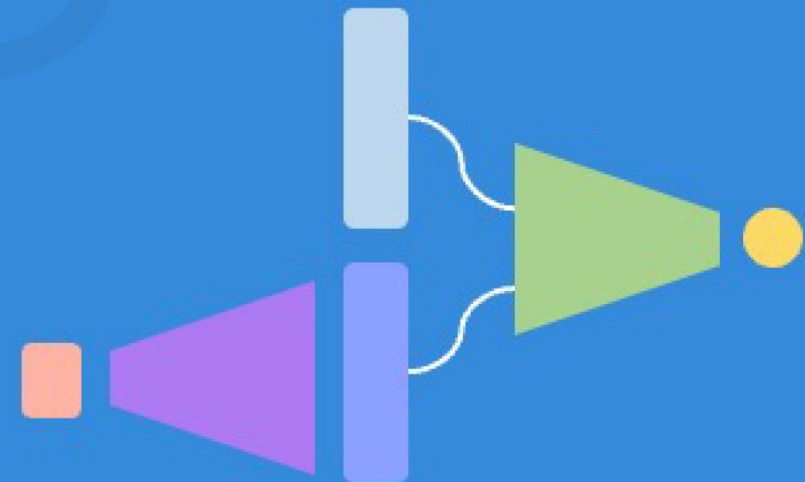
How can we learn $P_{model}(x)$ similar to $P_{data}(x)$?

Latent Variable Models

Autoencoders and Variational
Autoencoders (VAEs)

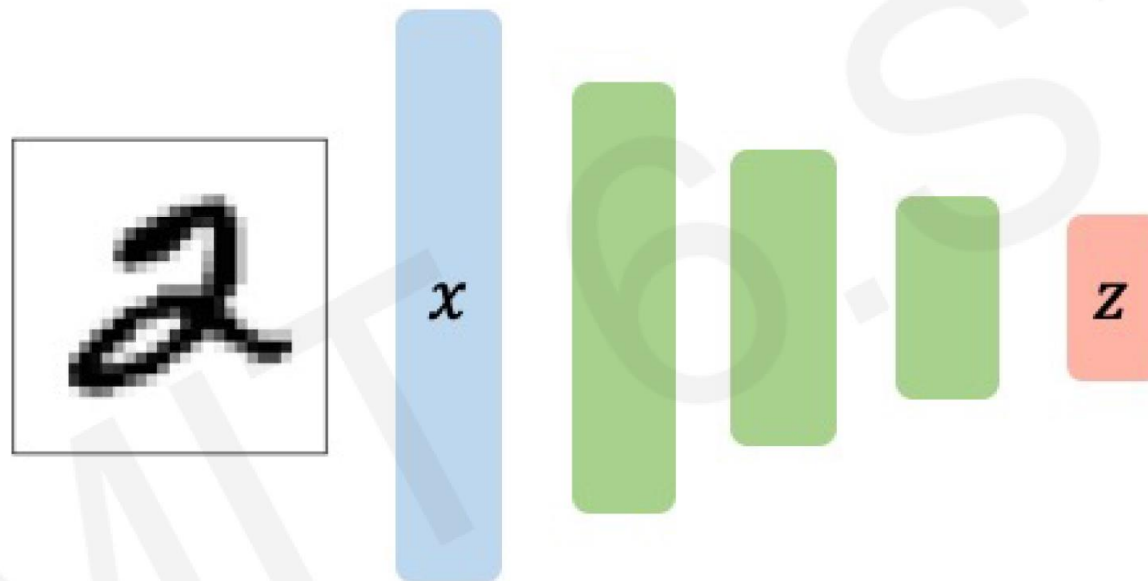


Generative Adversarial
Networks (GANs)



Autoencoders: Background

Unsupervised approach for learning a **lower-dimensional** feature representation from unlabeled training data



Why do we care about a low-dimensional z ?

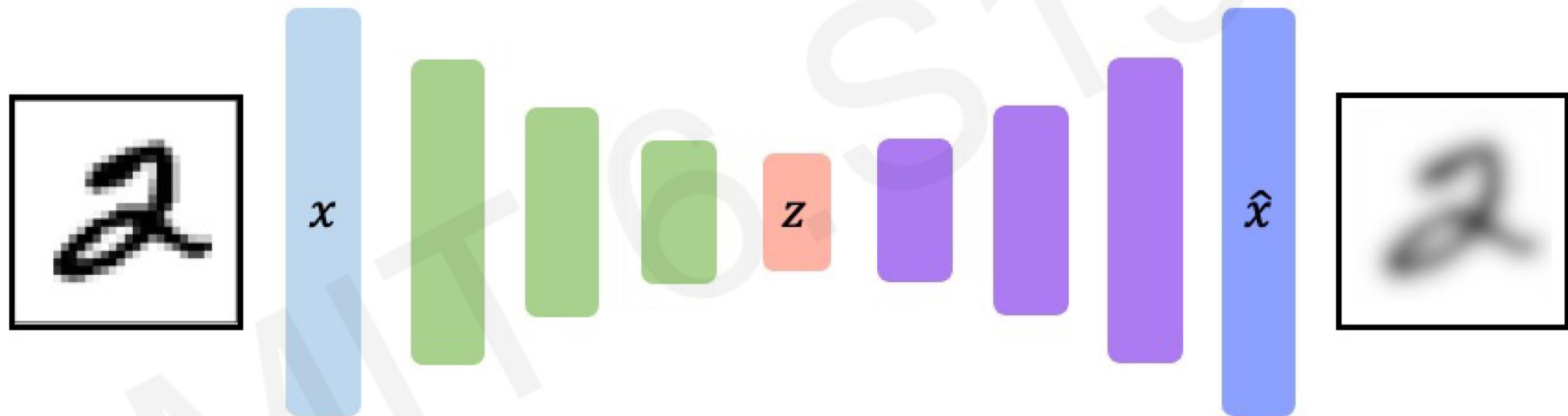


“Encoder” learns mapping from the data, x , to a low-dimensional latent space, z

Autoencoders: Background

How can we learn this latent space?

Train the model to use these features to **reconstruct the original data**

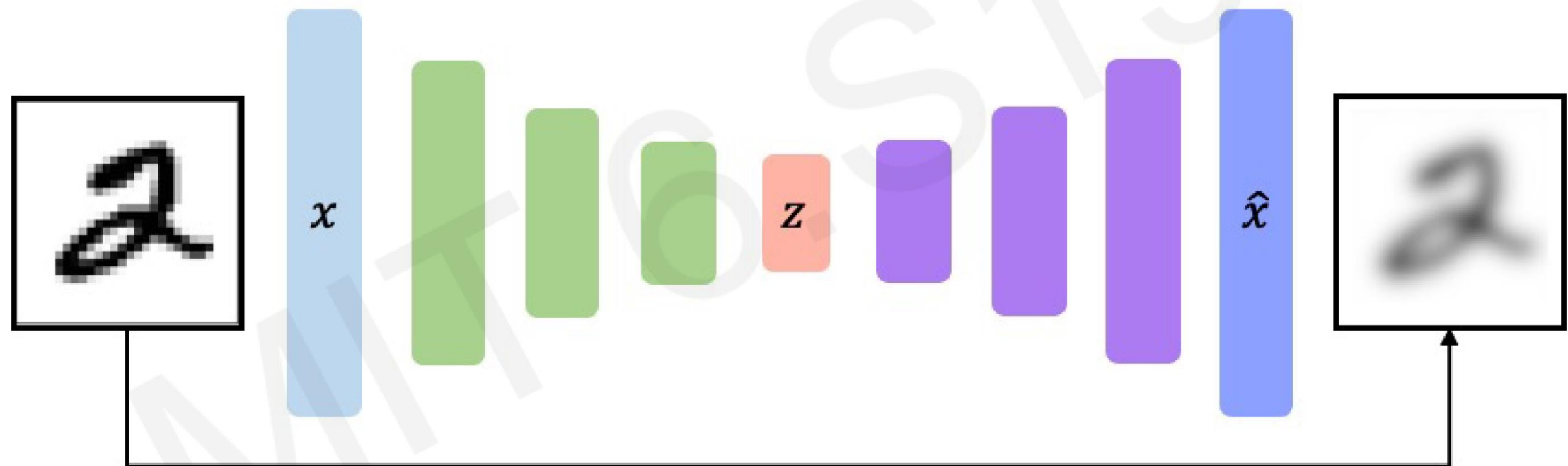


“Decoder” learns mapping back from latent space, z ,
to a reconstructed observation, $\hat{x}$

Autoencoders: Background

How can we learn this latent space?

Train the model to use these features to **reconstruct the original data**



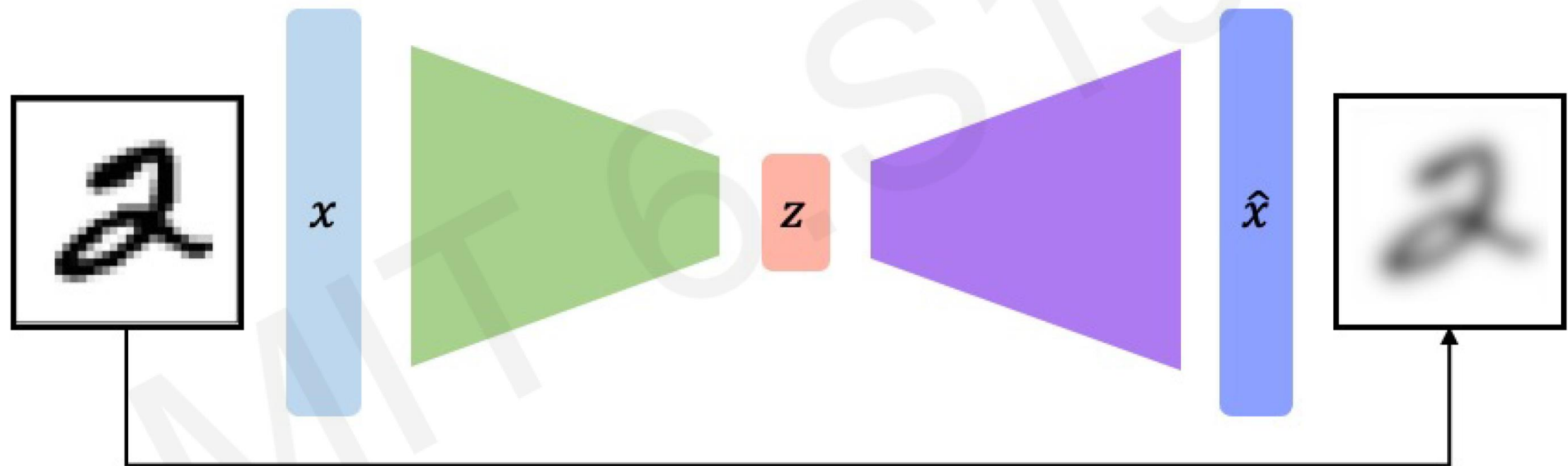
$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Loss function doesn't
use any labels!!

Autoencoders: Background

How can we learn this latent space?

Train the model to use these features to **reconstruct the original data**



$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Loss function doesn't
use any labels!!

Dimensionality of Latent Space -> Reconstruction Quality

Autoencoding is a form of compression!
Smaller latent space will force a larger training bottleneck

2D latent space



5D latent space

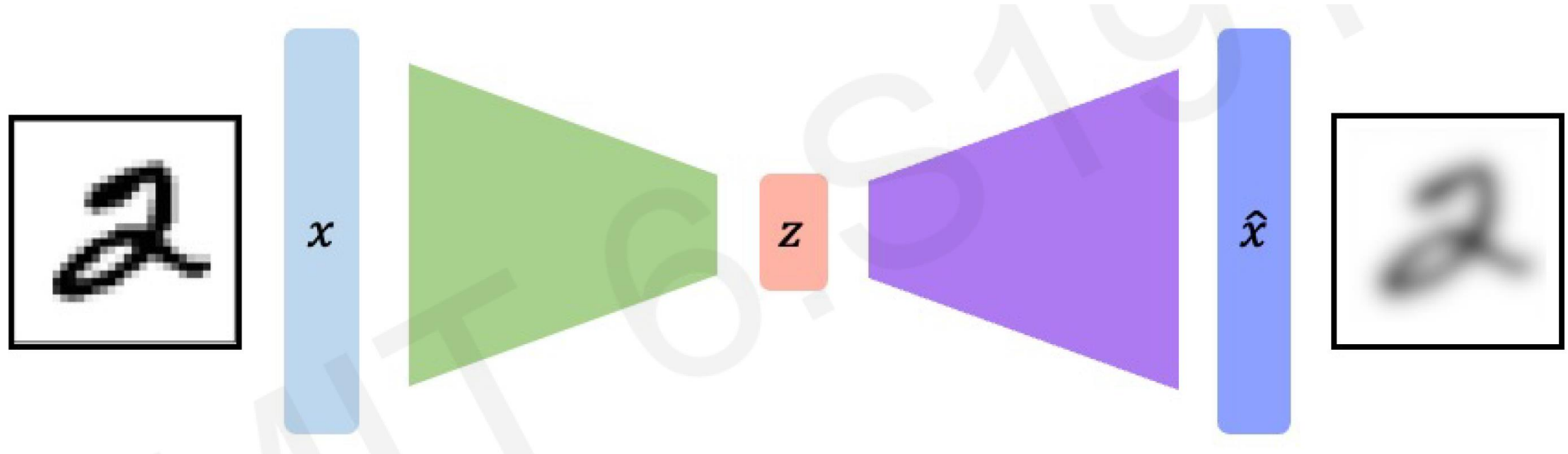


Ground Truth

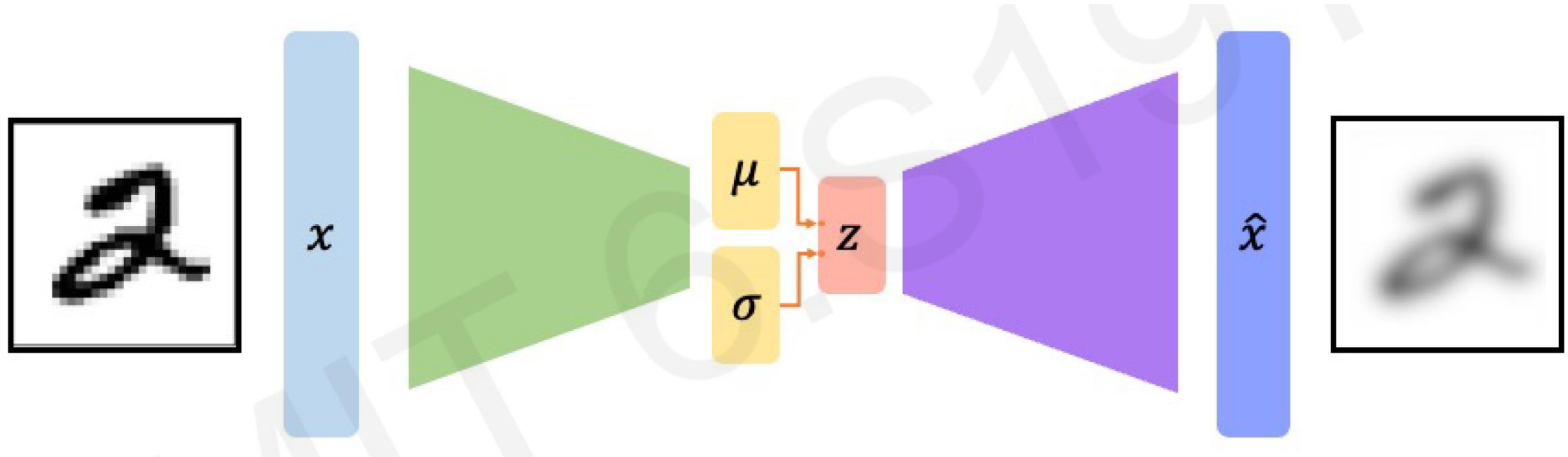


Variational Autoencoder (VAE)

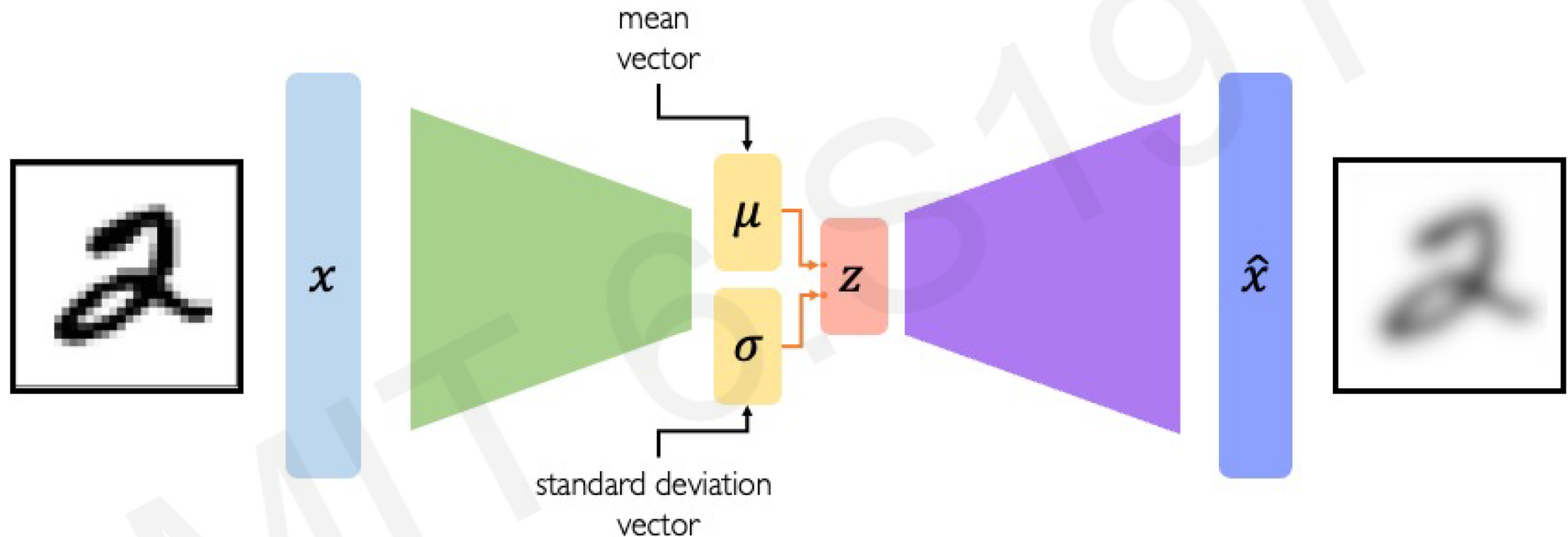
Traditional Autoencoder



VAEs: Key Difference with Traditional Autoencoder



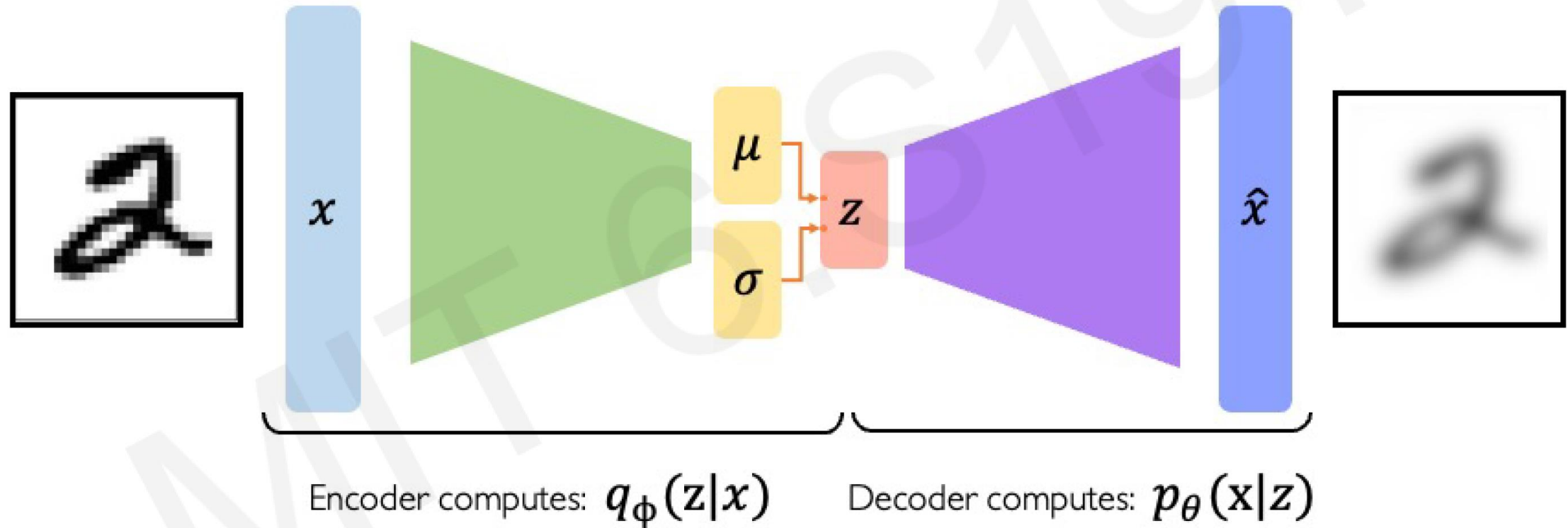
VAEs: Key Difference with Traditional Autoencoder



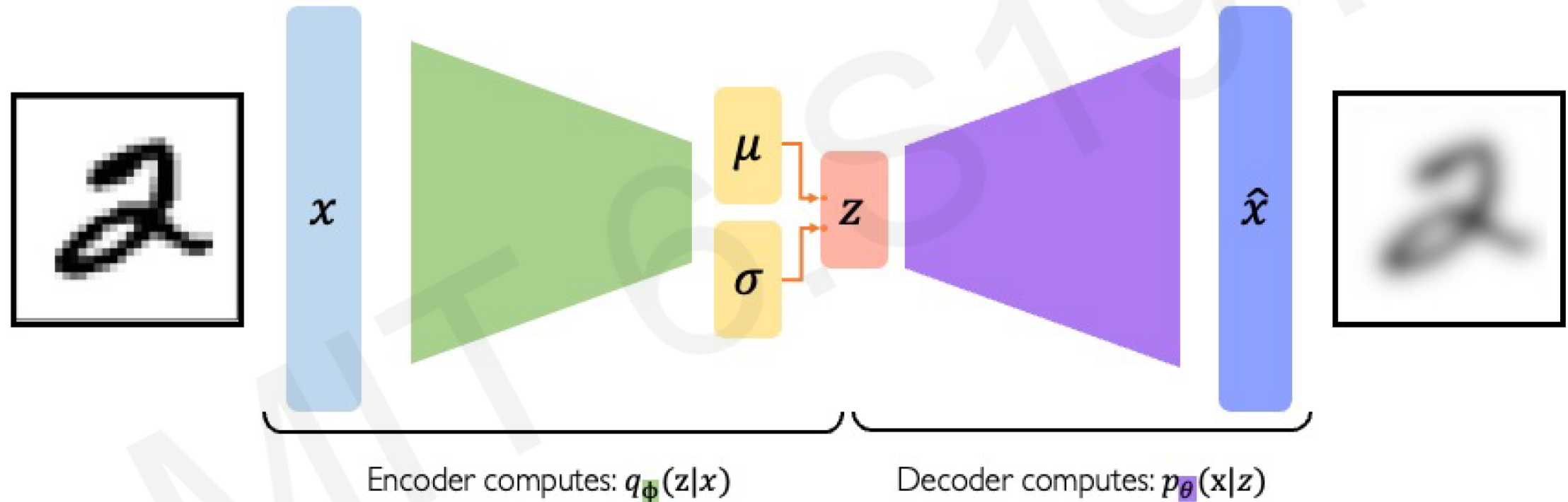
Variational autoencoders are a probabilistic twist on autoencoders!

Sample from the mean and standard deviation to compute latent sample

VAE Optimization

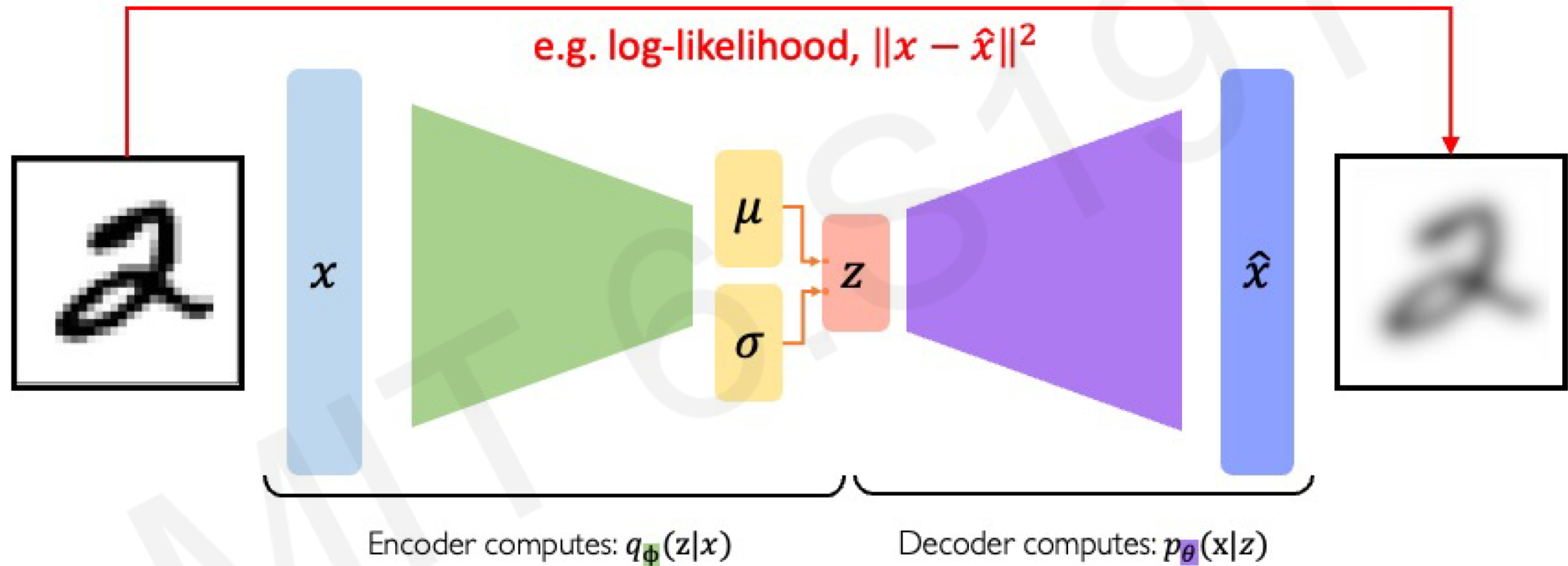


VAE Optimization



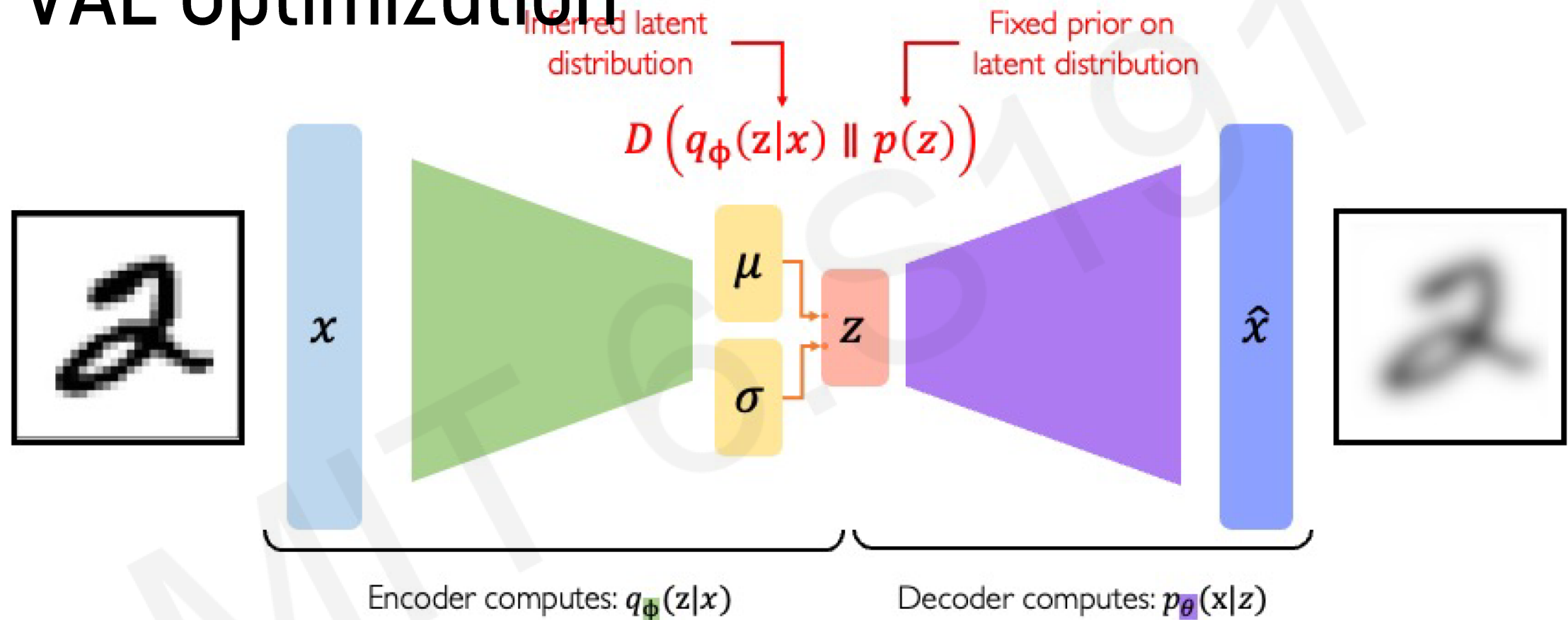
$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term})$$

VAE Optimization



$$\mathcal{L}(\phi, \theta, x) = \boxed{\text{(reconstruction loss)}} + \text{(regularization term)}$$

VAE Optimization

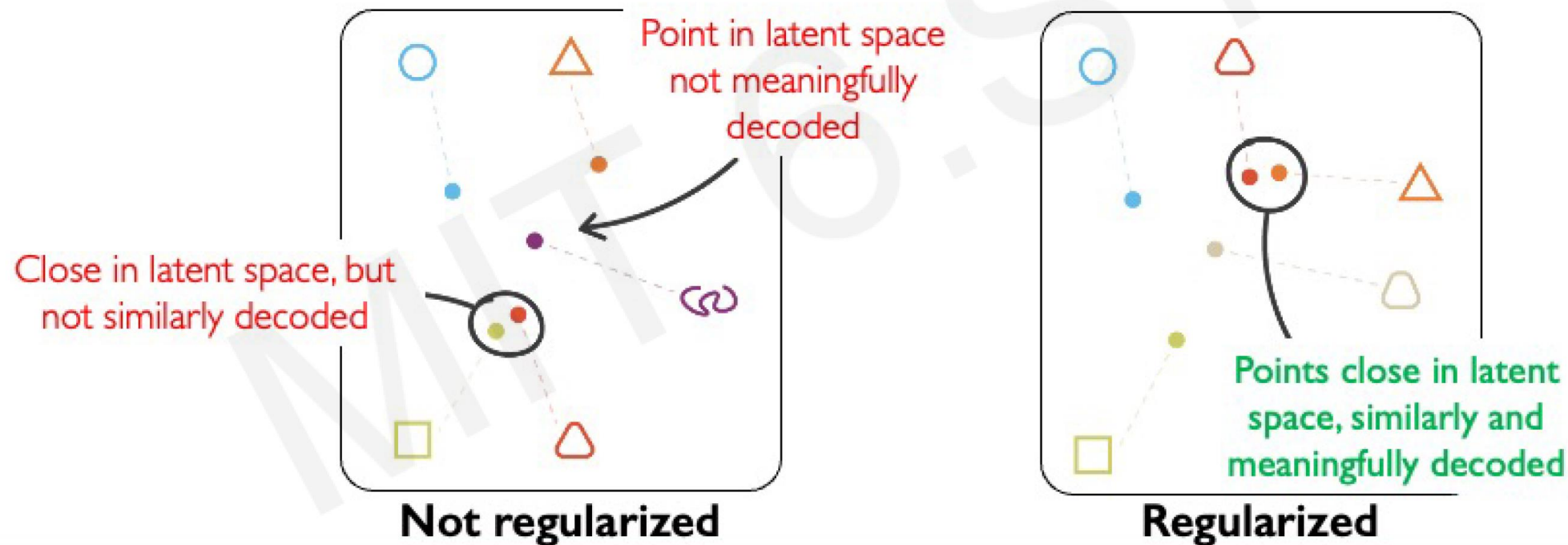


$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term})$$

Intuition on Regularization and the Normal Prior

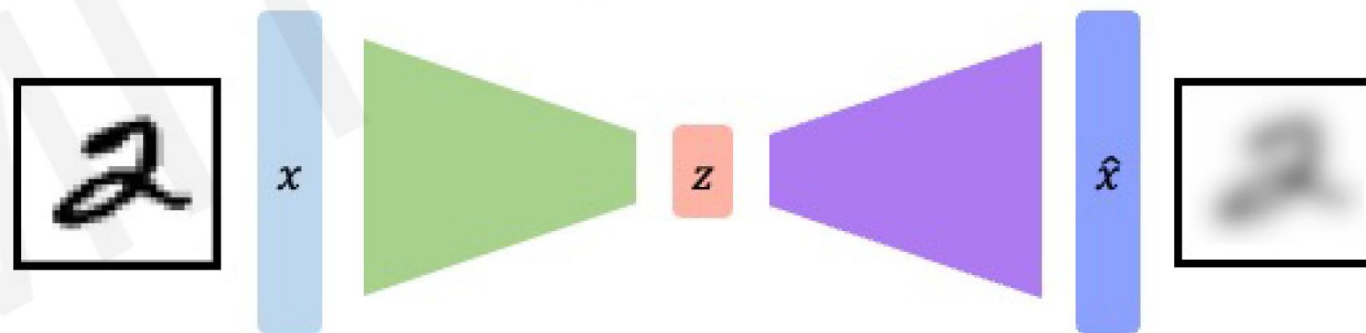
What properties do we want to achieve from regularization? 🤔

1. **Continuity:** points that are close in latent space $\rightarrow$ similar content after decoding
2. **Completeness:** sampling from latent space $\rightarrow$ "meaningful" content after decoding



VAE Summary

1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end
4. Interpret hidden latent variables using perturbation
5. Generating new examples



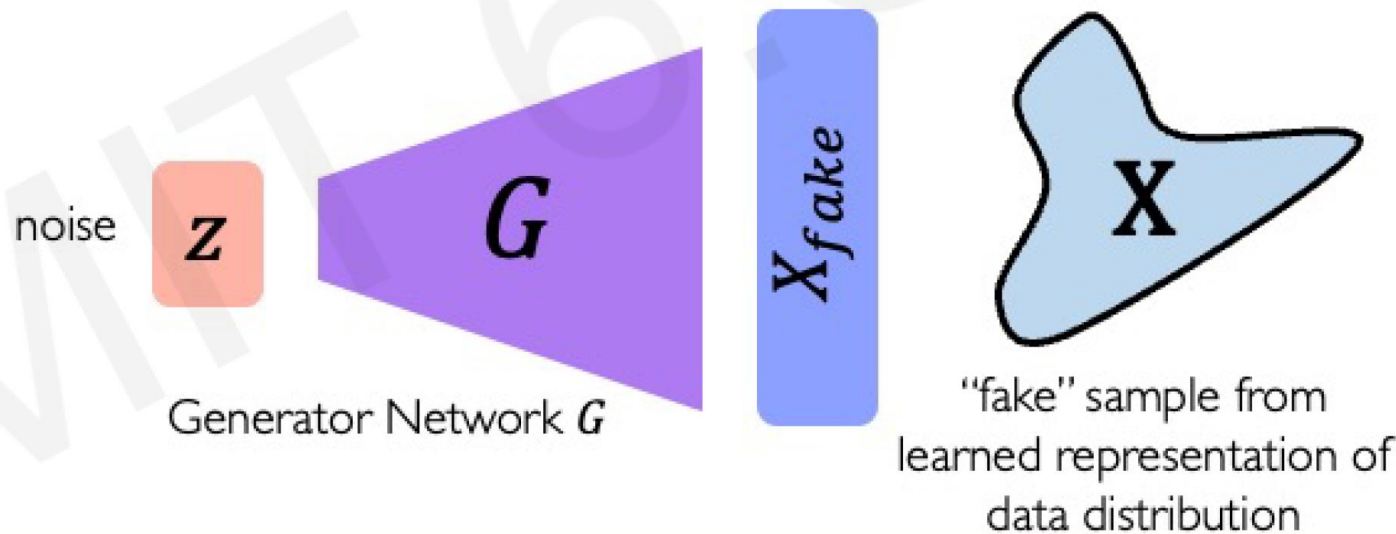
Generative Adversarial Network (GAN)

What if we Just Want to Sample?

Idea: don't explicitly model density, and instead just sample to generate new instances.

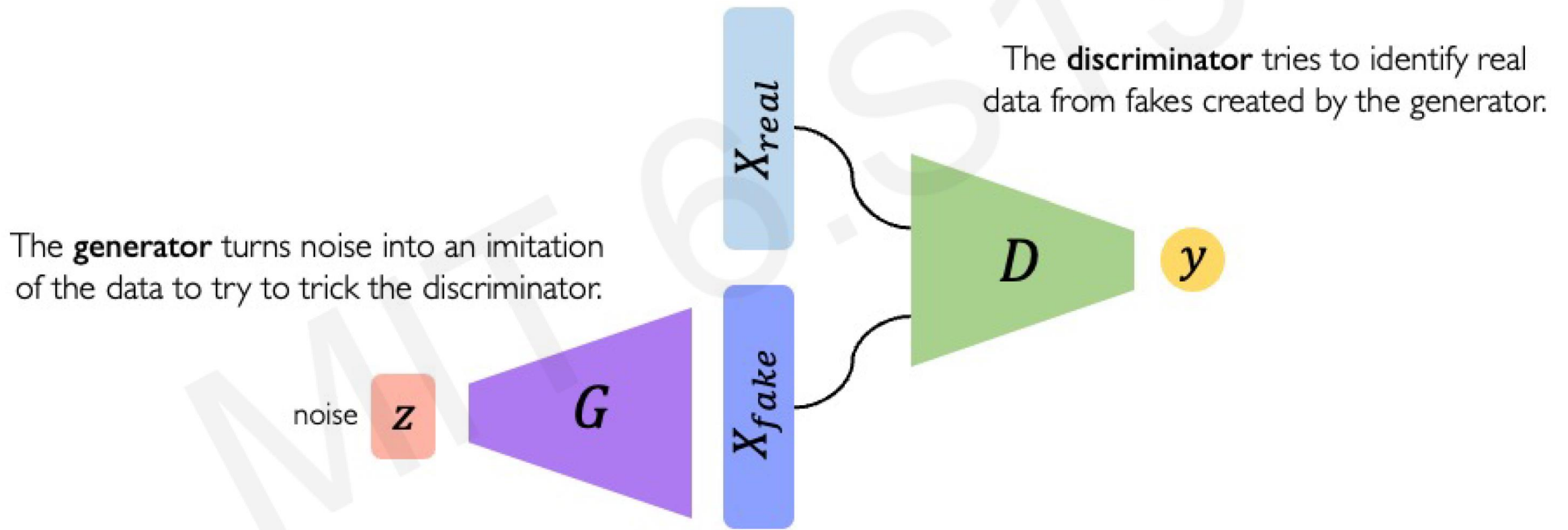
Problem: want to sample from complex distribution – can't do this directly!

Solution: sample from something simple (e.g., noise), learn a transformation to the data distribution.

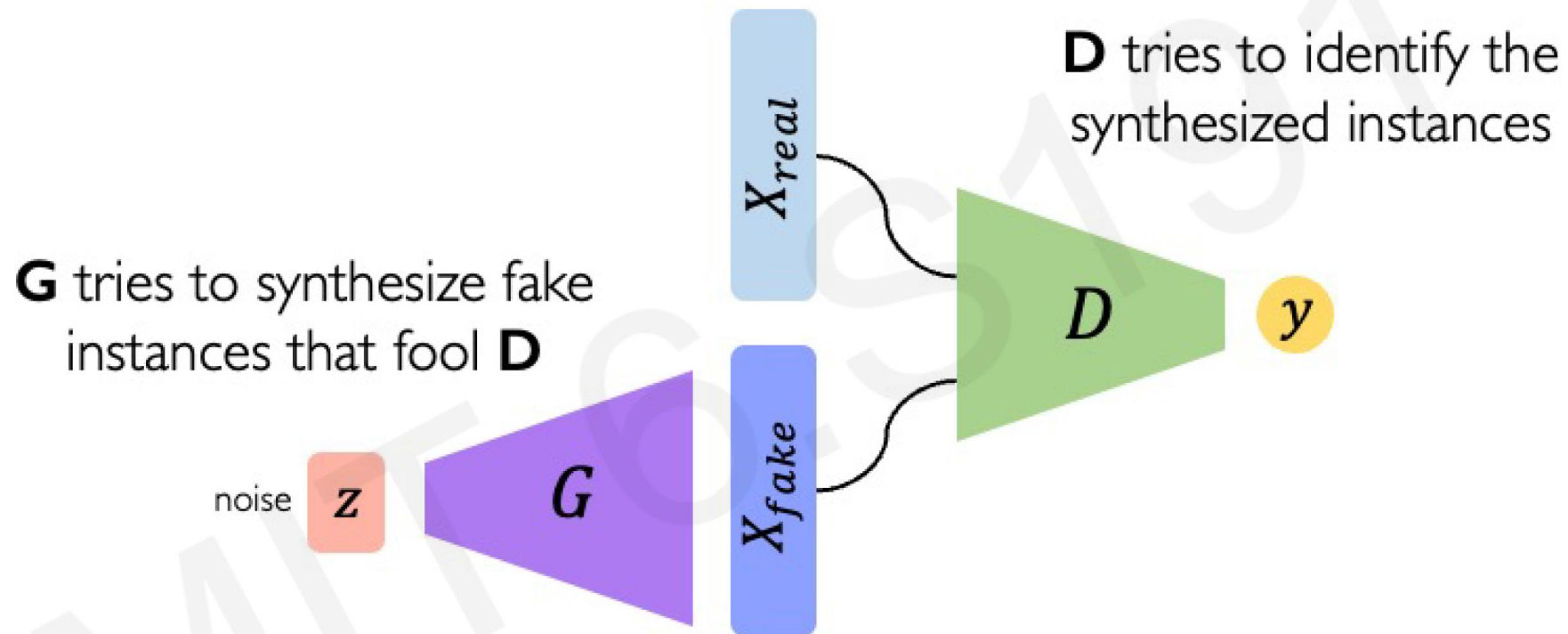


Generative Adversarial Network (GAN)

Generative Adversarial Networks (GANs) are a way to make a generative model by having two neural networks compete with each other.



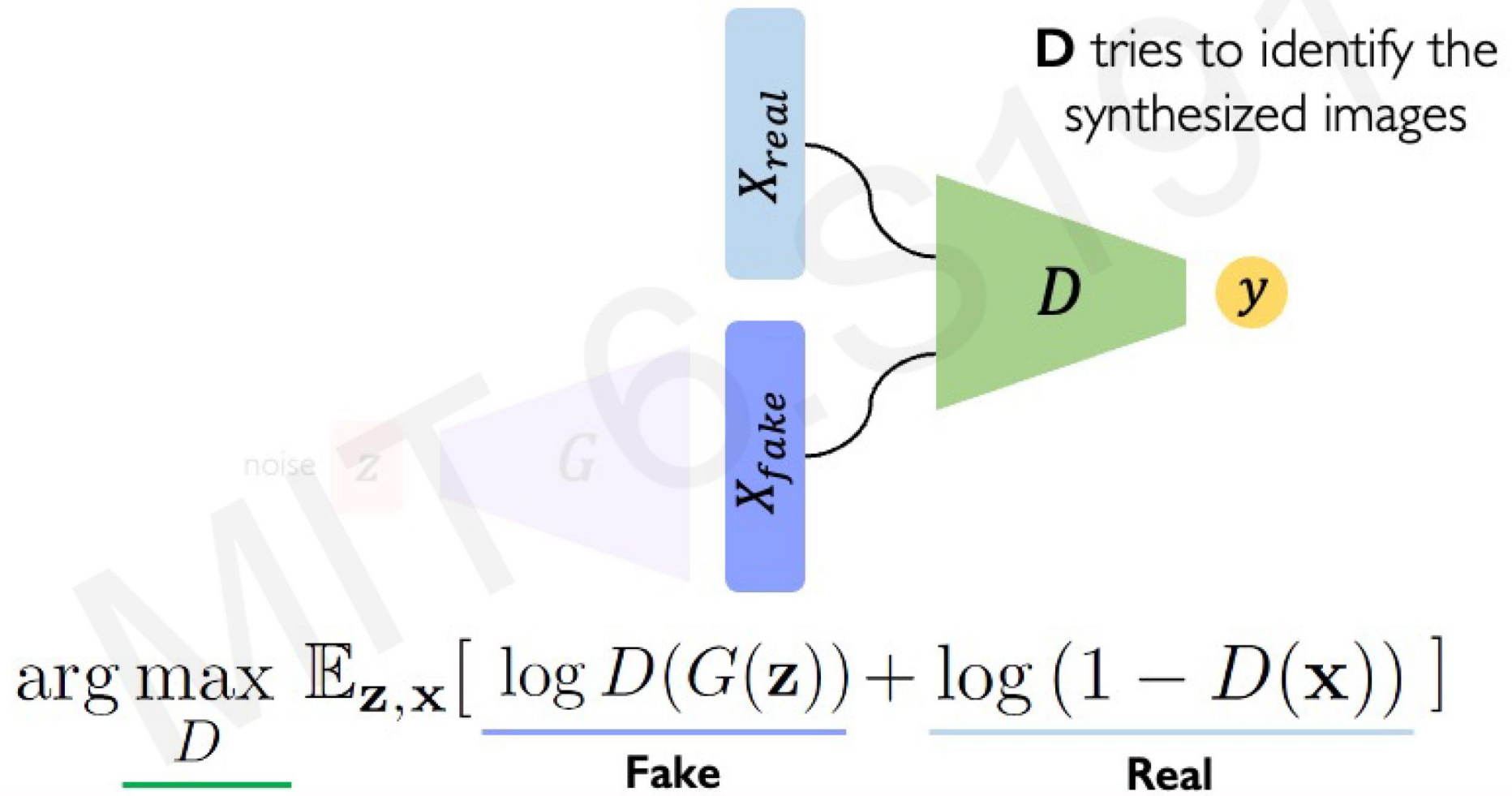
Training GANs



Training: adversarial objectives for D and G

Global optimum: G reproduces the true data distribution

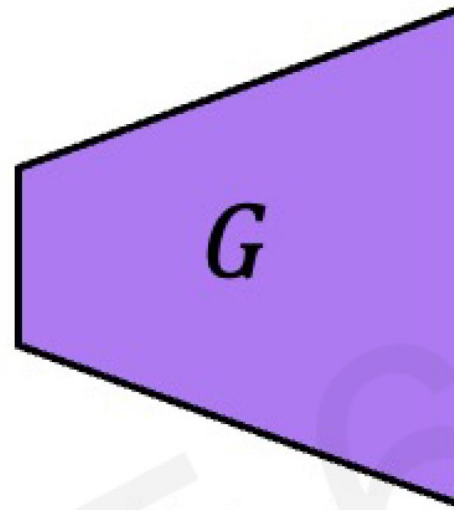
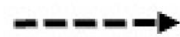
Training GANs: Loss Function



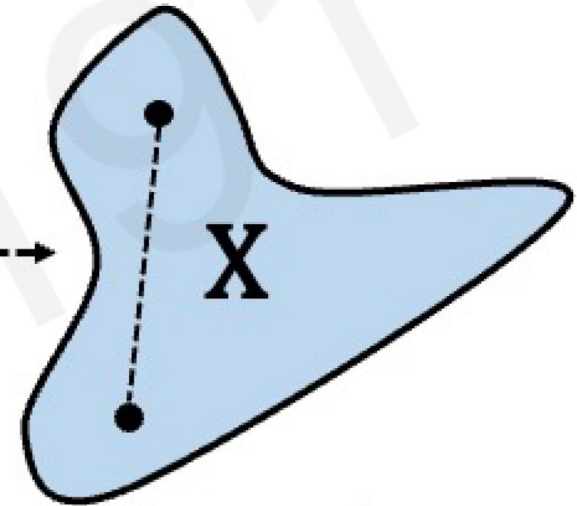
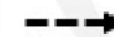
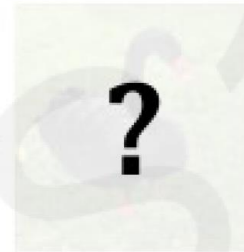
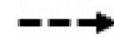
GANs are Distribution Transformers

Gaussian noise

$$z \sim N(0,1)$$



Trained
generator

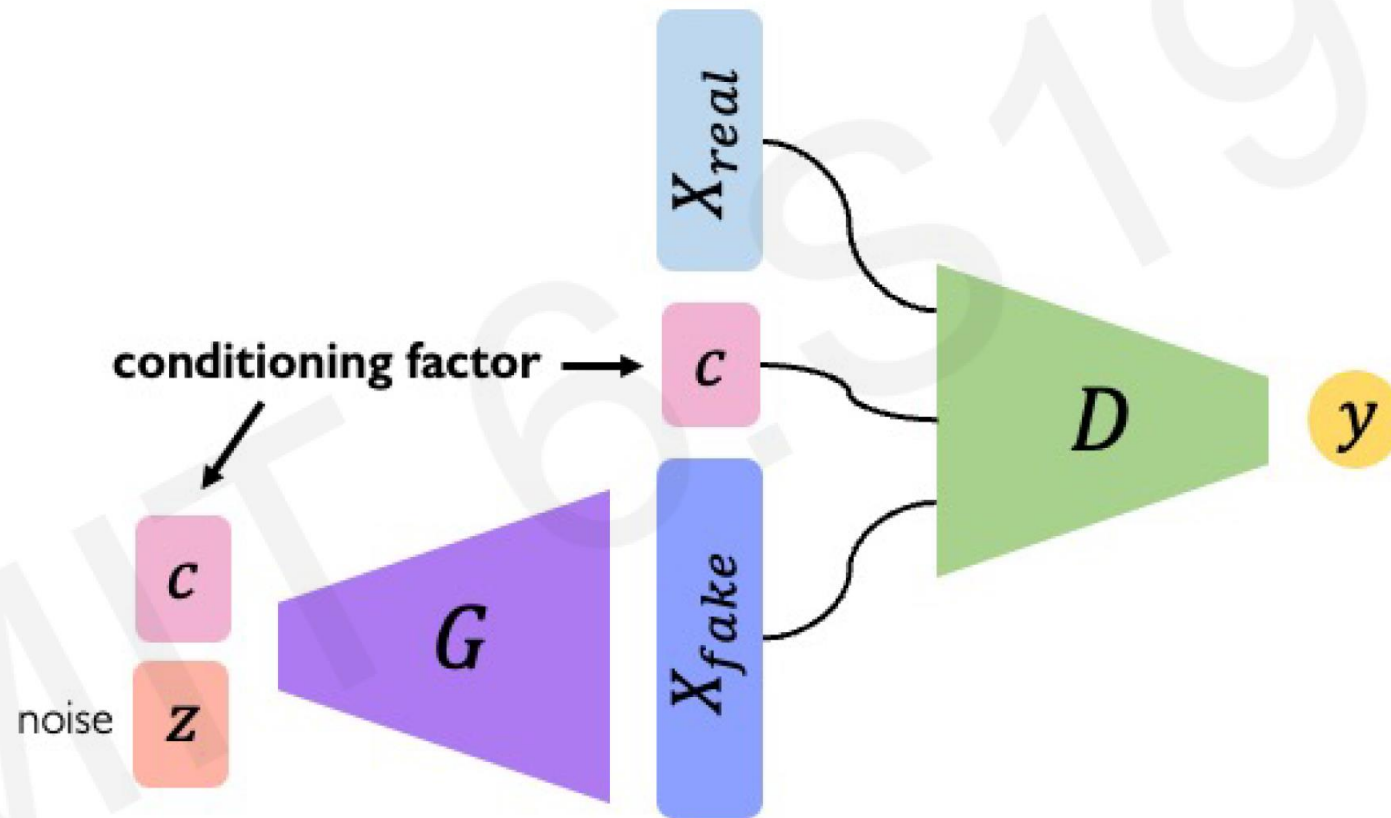


Learned target
data distribution



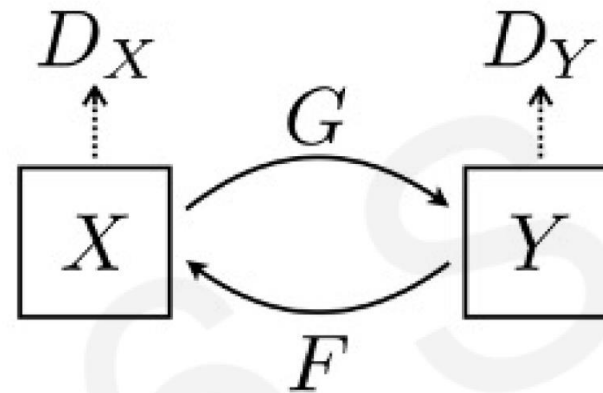
Conditional GANs

What if we want to control the nature of the output, by **conditioning** on a label?



Cycle GANs

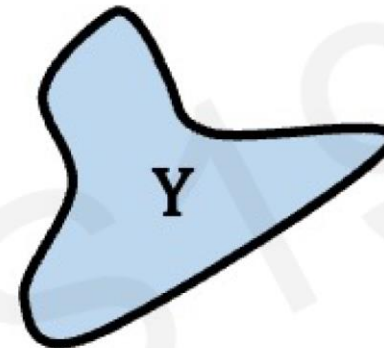
CycleGAN learns transformations across domains with unpaired data.



GANs vs Cycle GANs

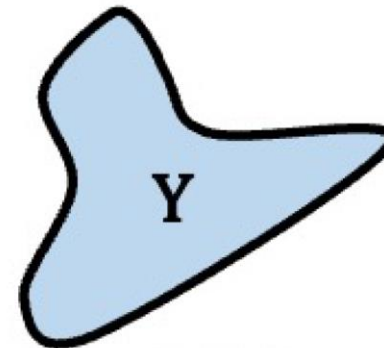
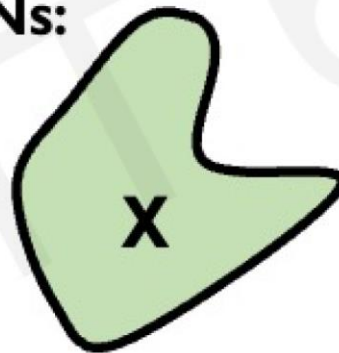
GANs:

Gaussian noise
 $z \sim N(0,1)$



Gaussian noise $\rightarrow$ target data manifold

CycleGANs:

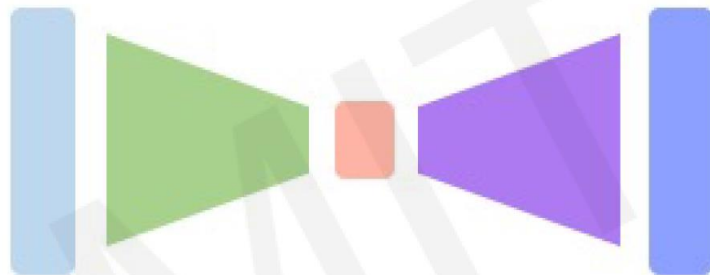


data manifold X $\rightarrow$ data manifold Y

Deep Generative Modeling Summary

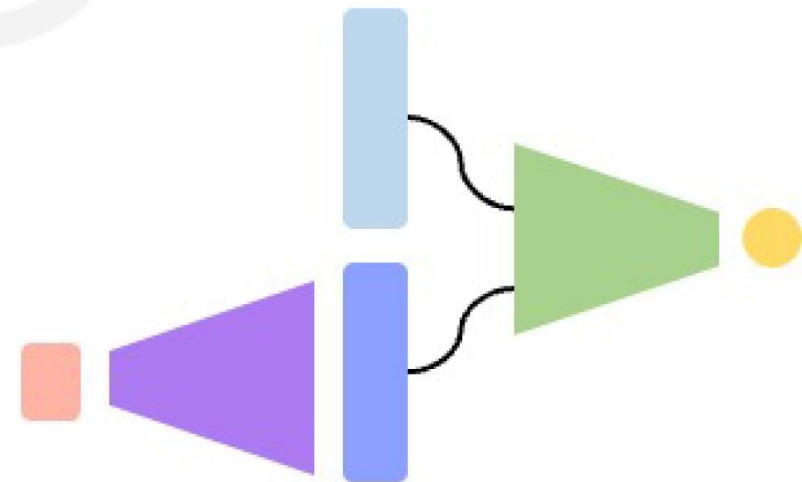
Autoencoders and Variational Autoencoders (VAEs)

Learn lower-dimensional **latent space** and **sample** to generate input reconstructions



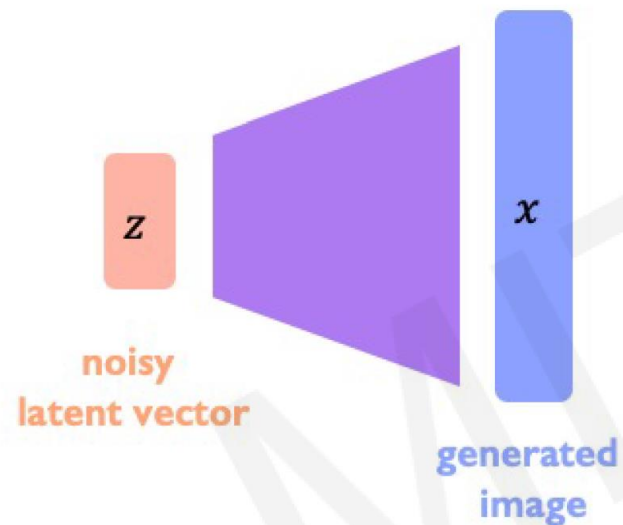
Generative Adversarial Networks (GANs)

Competing **generator** and **discriminator** networks

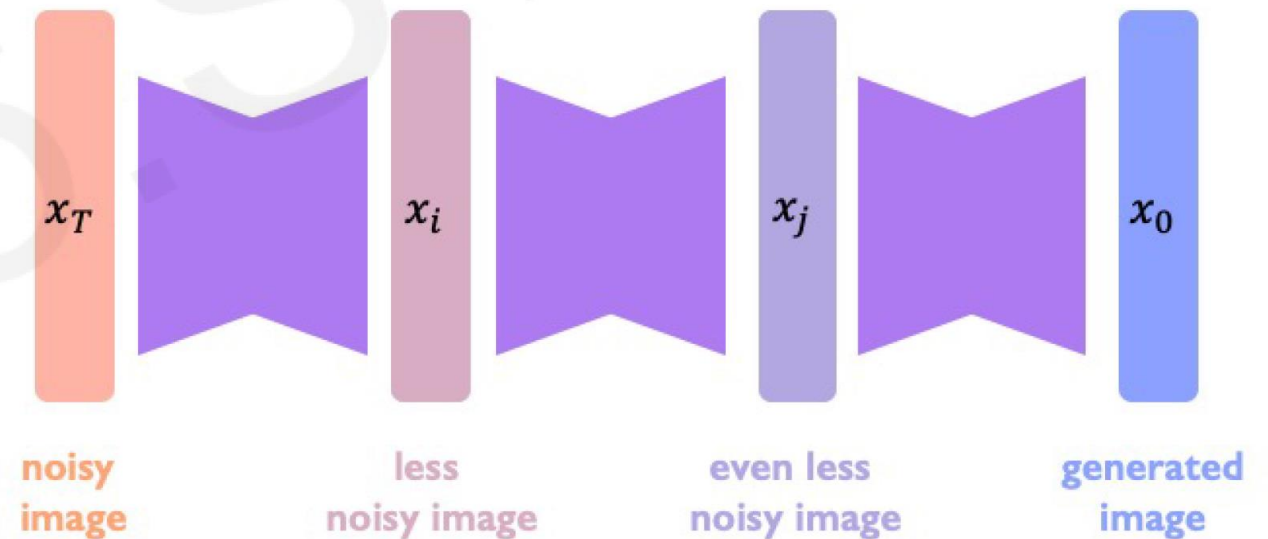


Diffusion Models

VAEs/GANs: Generating images in one-shot directly from low-dimensional latent variables

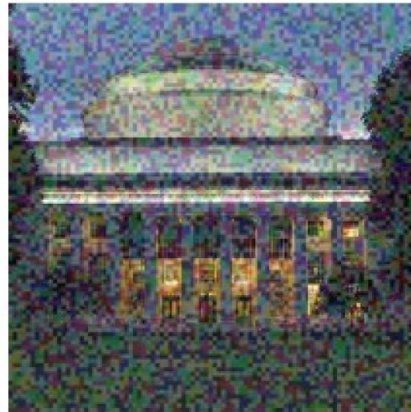
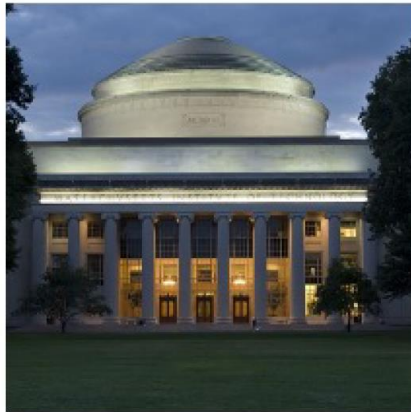


Diffusion: Generating images iteratively by repeatedly refining and removing noise



The Diffusion Process

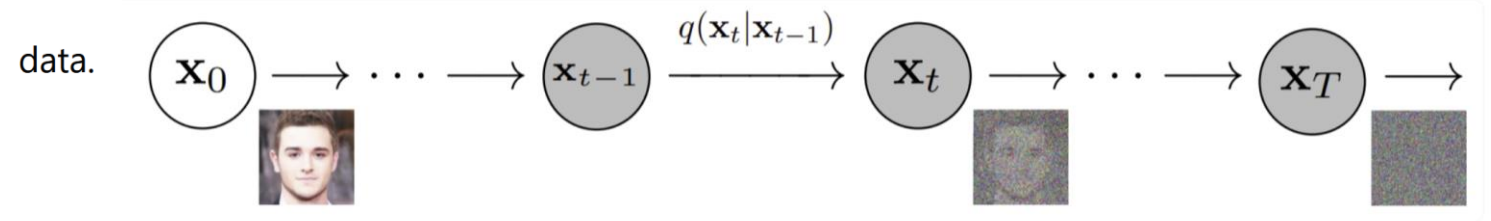
Forward noising
(data-to-noise)



Reverse denoising
(noise-to-data)

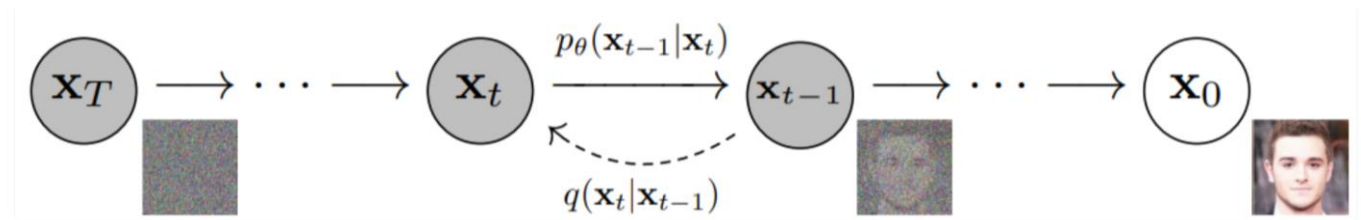
Diffusion Models

More specifically, a Diffusion Model is a latent variable model which maps to the latent space using a fixed Markov chain. This chain gradually adds noise to the data in order to obtain the approximate posterior $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$, where $\mathbf{x}_1, \dots, \mathbf{x}_T$ are the latent variables with the same dimensionality as $\mathbf{x}_0$. In the figure below, we see such a Markov chain manifested for image data.



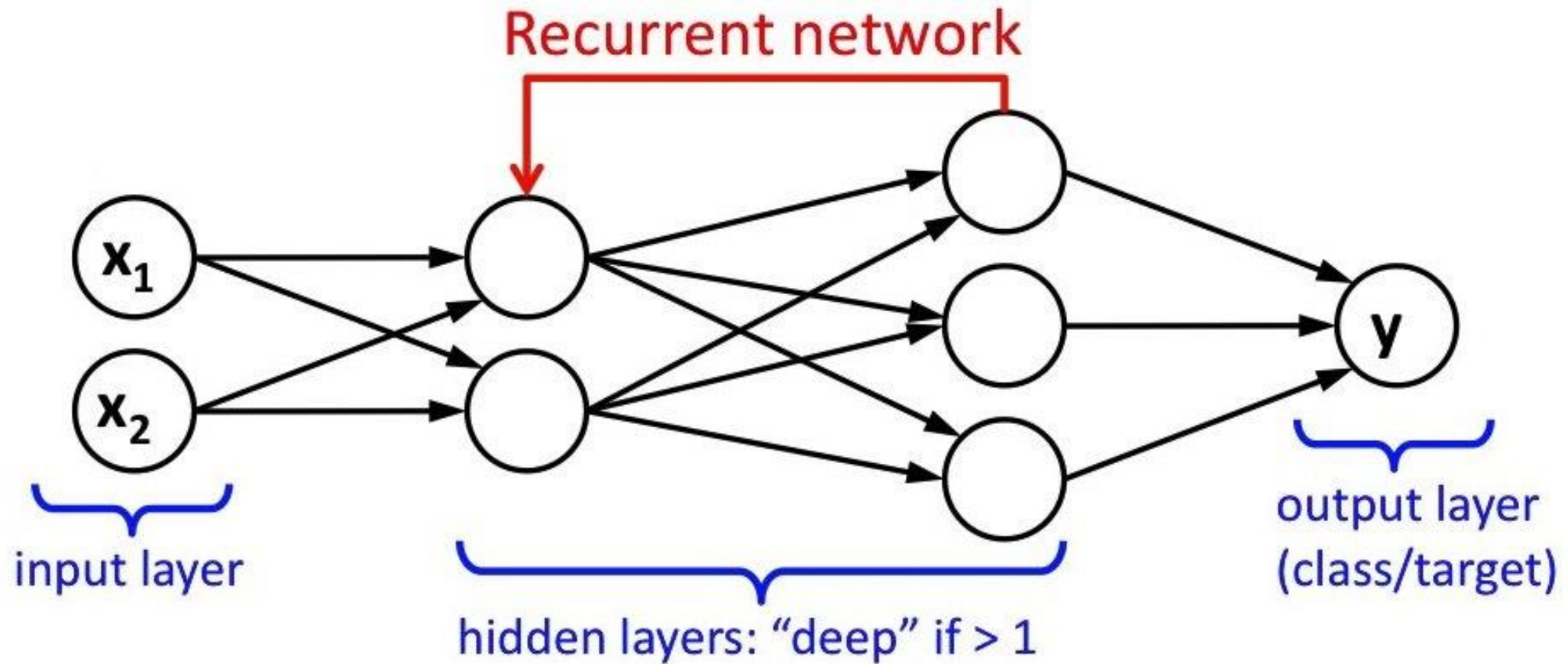
(Modified from [source](#))

Ultimately, the image is asymptotically transformed to pure Gaussian noise. The **goal** of training a diffusion model is to learn the **reverse** process - i.e. training $p_\theta(x_{t-1}|x_t)$. By traversing backwards along this chain, we can generate new data.



(Modified from [source](#))

Recurrent Neural Networks



Recurrent Neural Networks

one to one

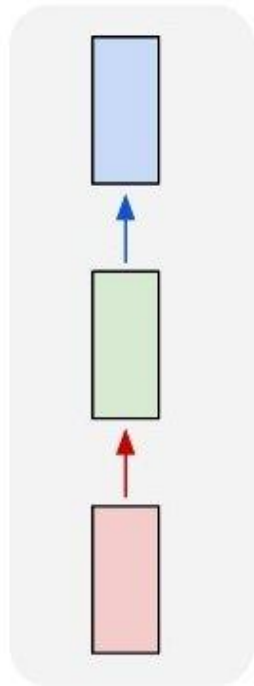


Image in
Label out

one to many

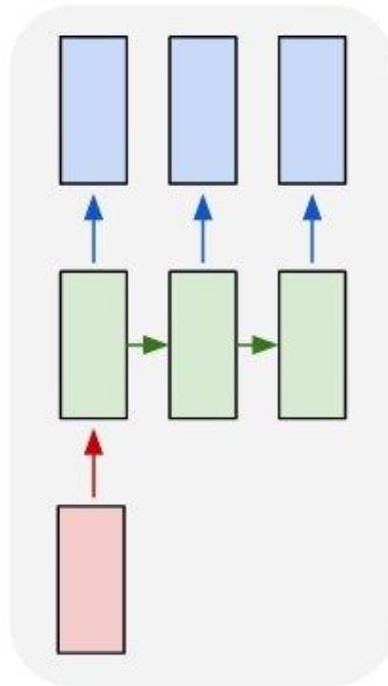
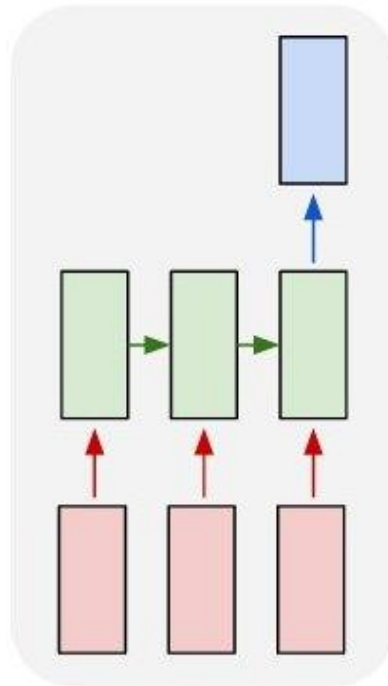


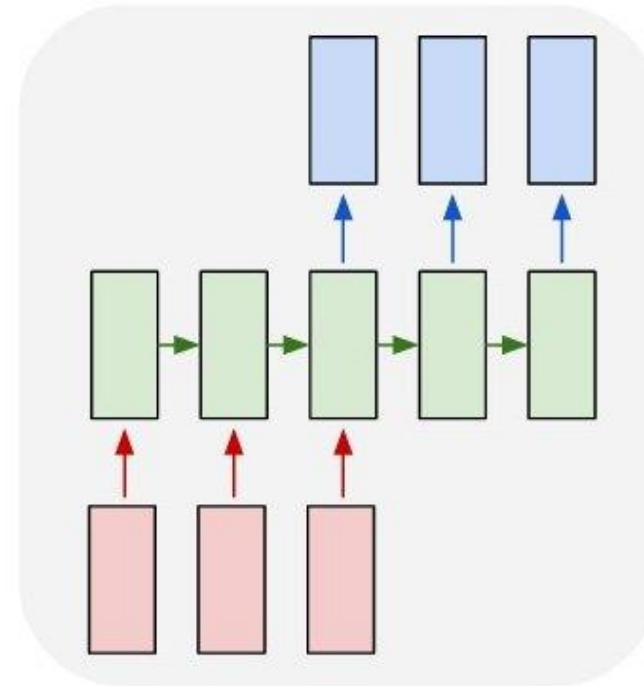
Image in
Words out

many to one



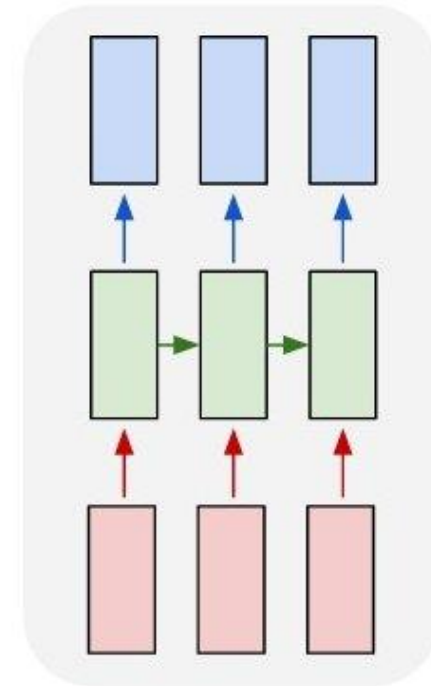
Words in
Sentiment out

many to many



English in
Portuguese out

many to many



Video In
Labels out

Traditional vs ML problem solving

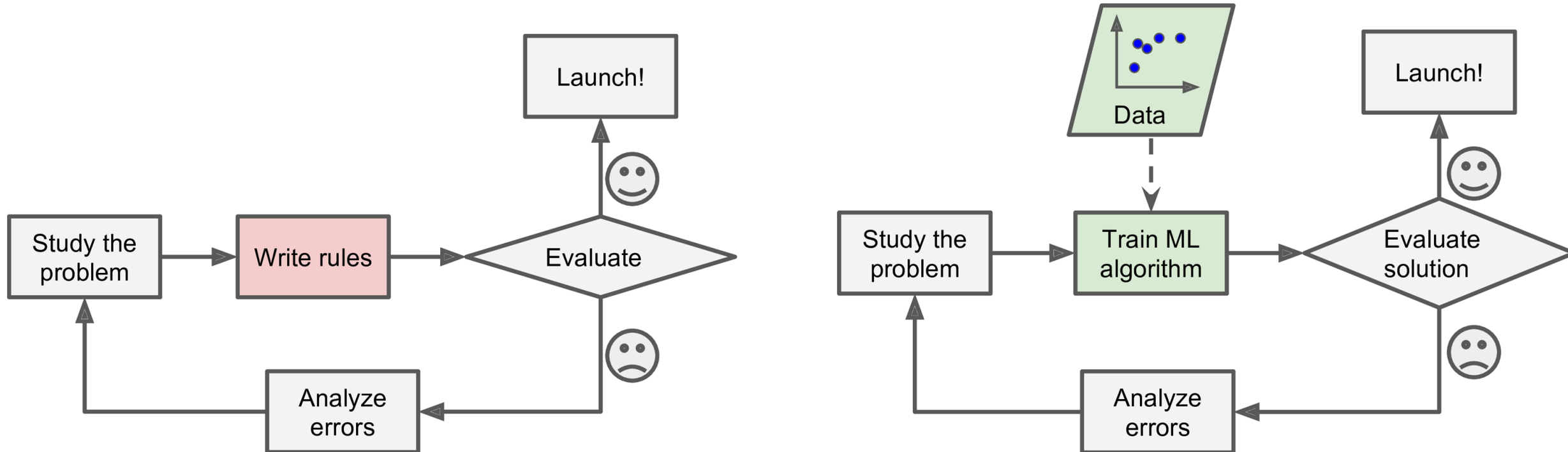
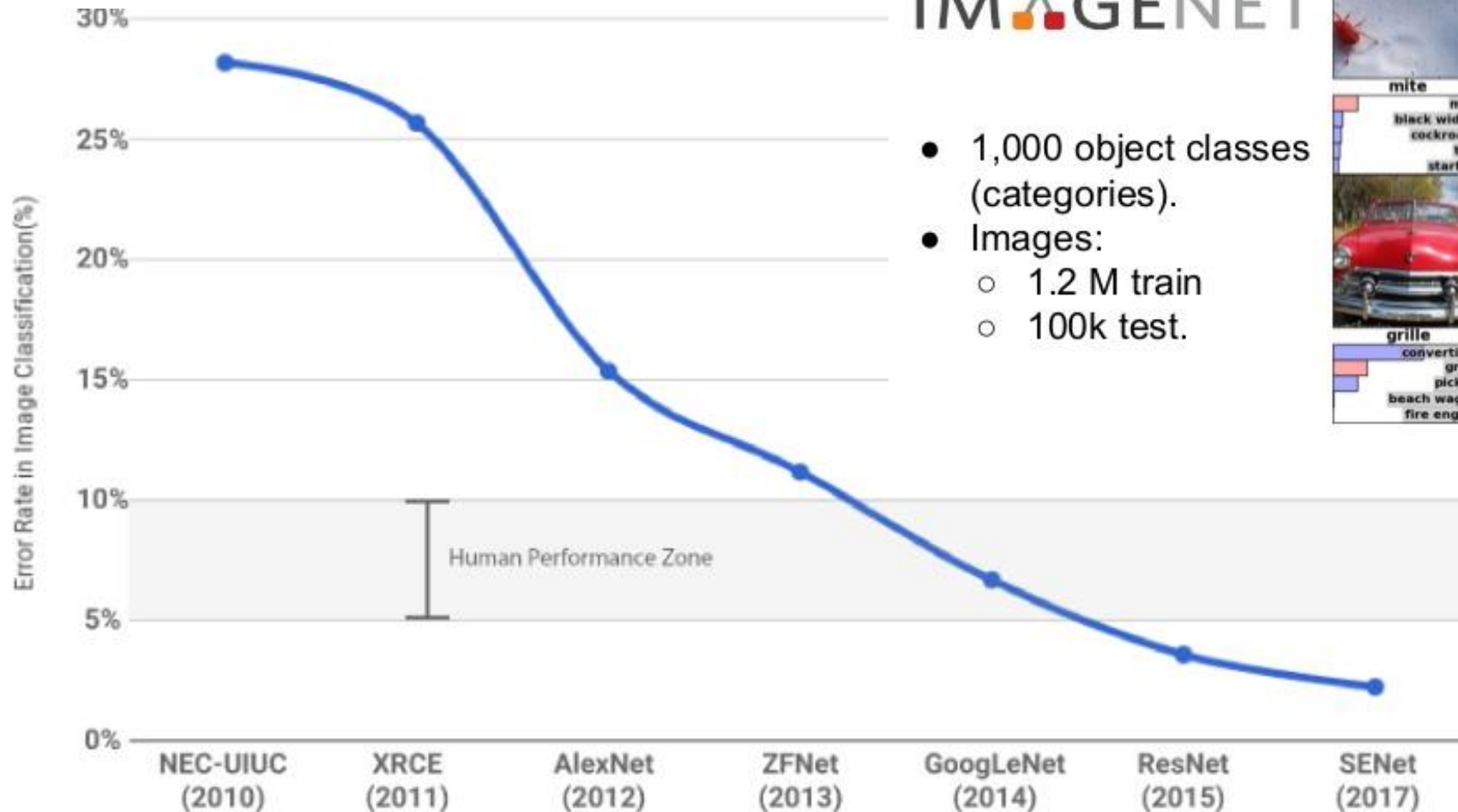


Image Classification

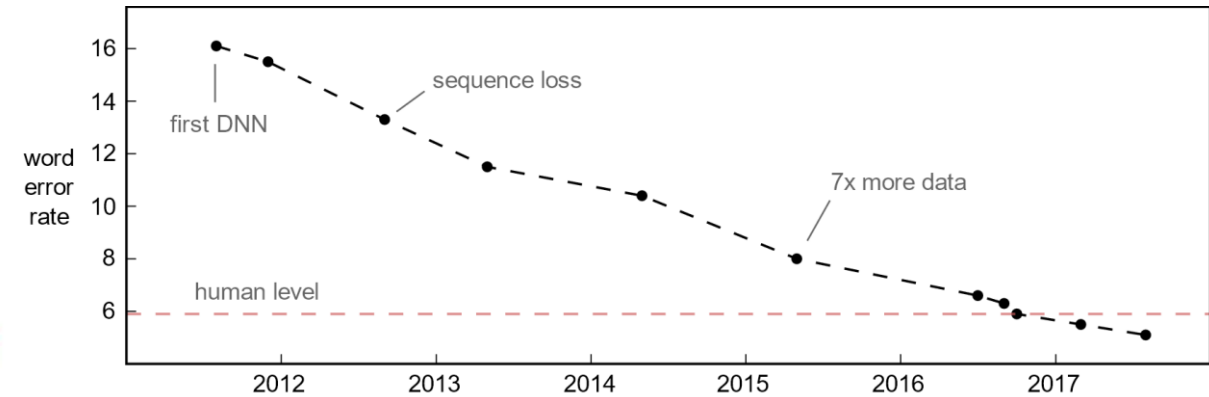
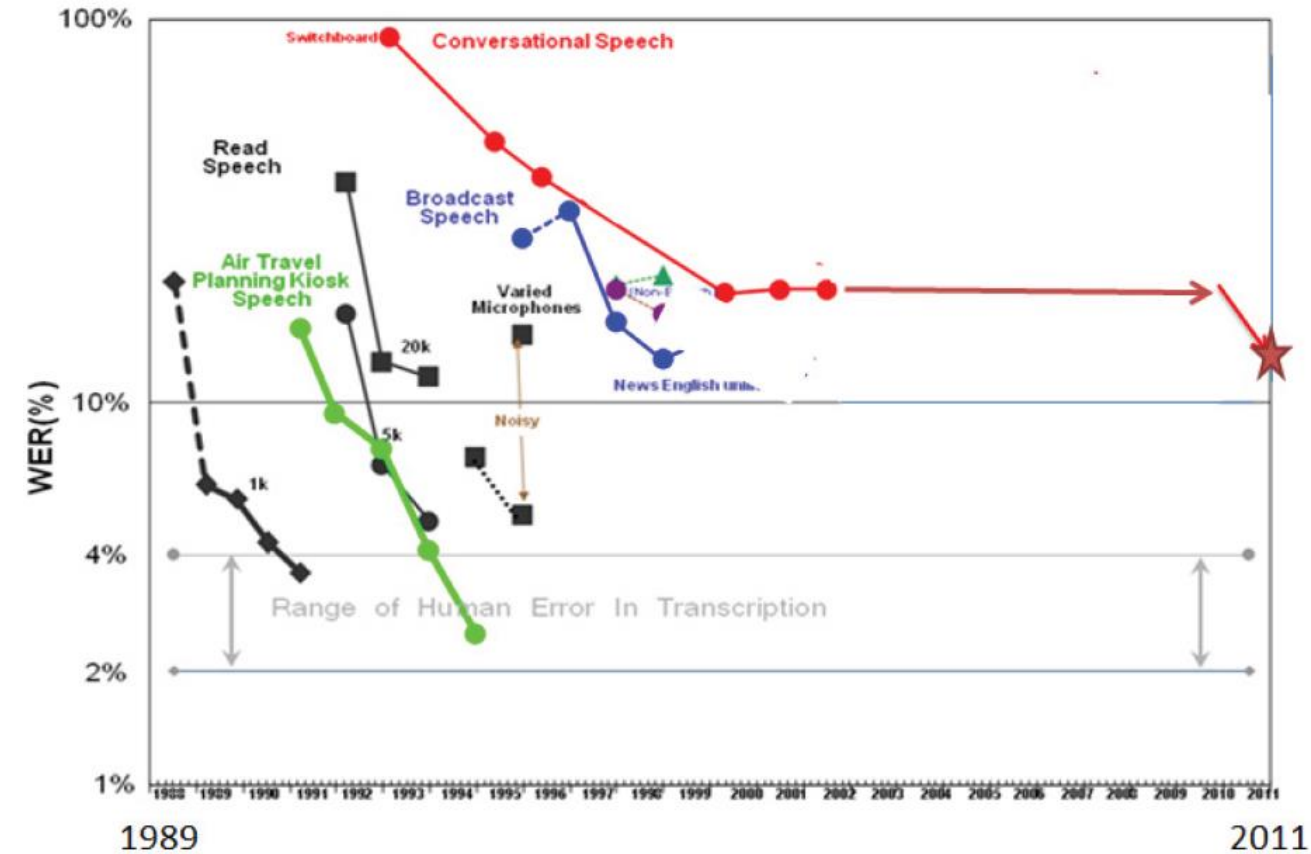


IMAGENET

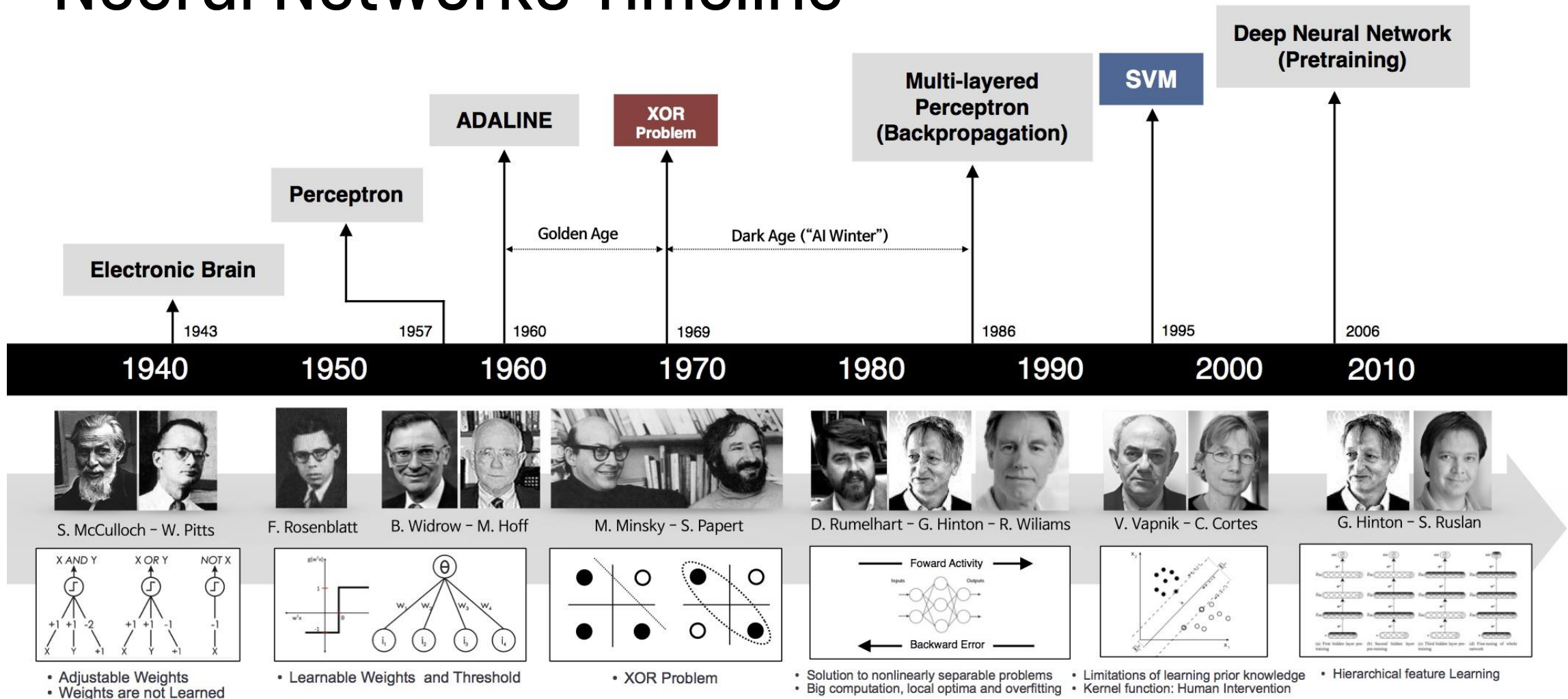
- 1,000 object classes (categories).
- Images:
 - 1.2 M train
 - 100k test.



Speech Recognition



Neural Networks Timeline



Neural Networks Limitations

- Very **data hungry** (eg. often millions of examples)
- **Computationally intensive** to train and deploy (tractably requires GPUs)
- Easily fooled by **adversarial examples**
- Can be subject to **algorithmic bias**
- Poor at **representing uncertainty** (how do you know what the model knows?)
- Uninterpretable **black boxes**, difficult to trust
- Often require **expert knowledge** to design, fine tune architectures
- Difficult to **encode structure** and prior knowledge during learning
- **Extrapolation**: struggle to go beyond the data

TDDC17 AI LE10 HT2024:
Neural networks
Convolutional Neural Networks
Deep Generative Learning

www.ida.liu.se/~TDDC17