TDDC17 LE9 HT2025

Machine Learning II

Fredrik Heintz

Dept. of Computer Science
Linköping University

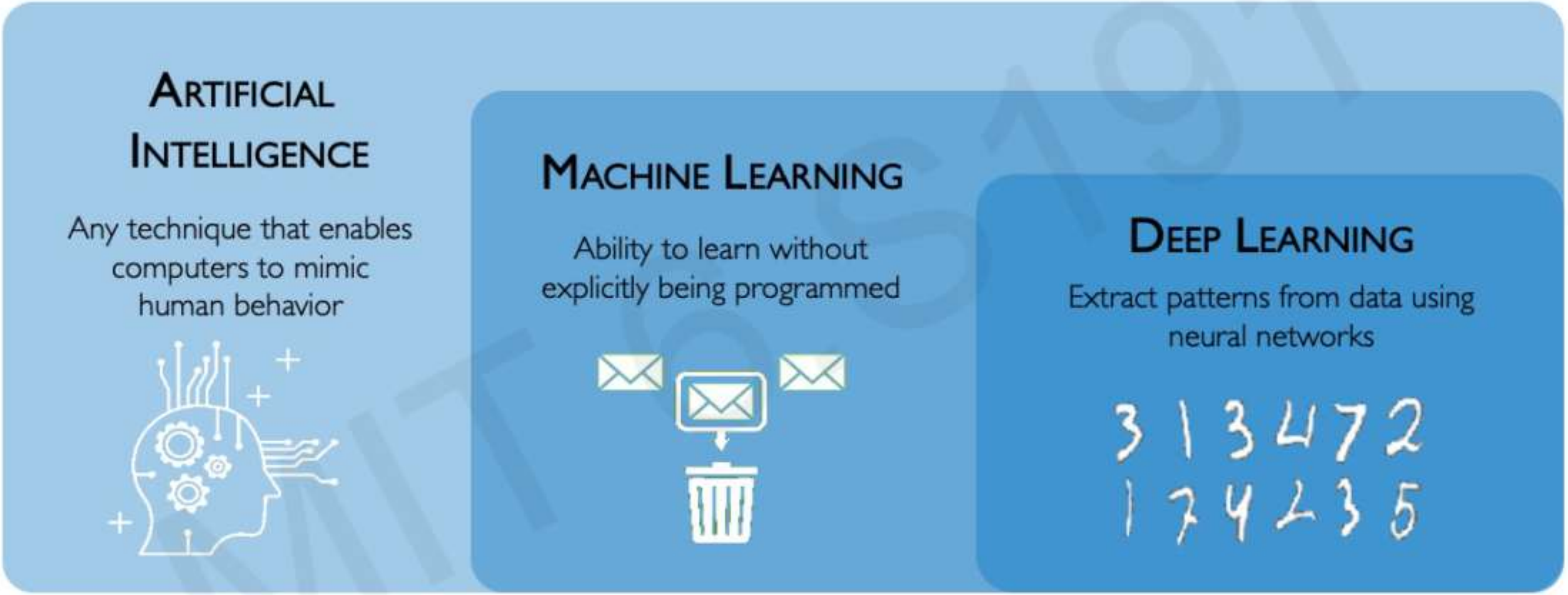
fredrik.heintz@liu.se
@FredrikHeintz

Outline:

- Neural networks
- Deep Learning for Computer Vision
- Deep Generative Models
- Deep Learning for Natural Language Processing

LINKÖPING UNIVERSITY

What is Deep Learning?



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 **Mid Level Features** **High Level Features**

Lines & Edges Eyes & Nose & Ears Facial Structure

Why Now?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

⋮

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

⋮

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

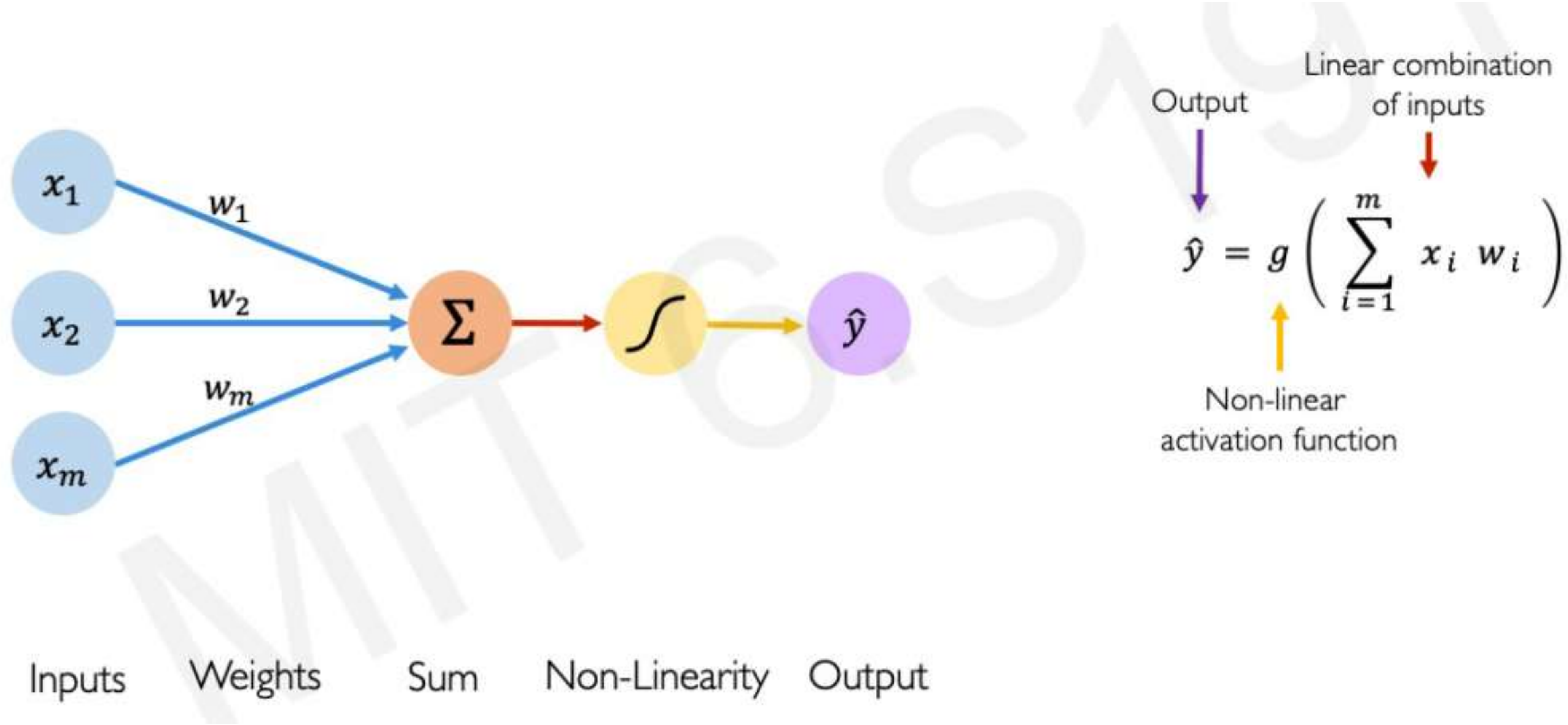
3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

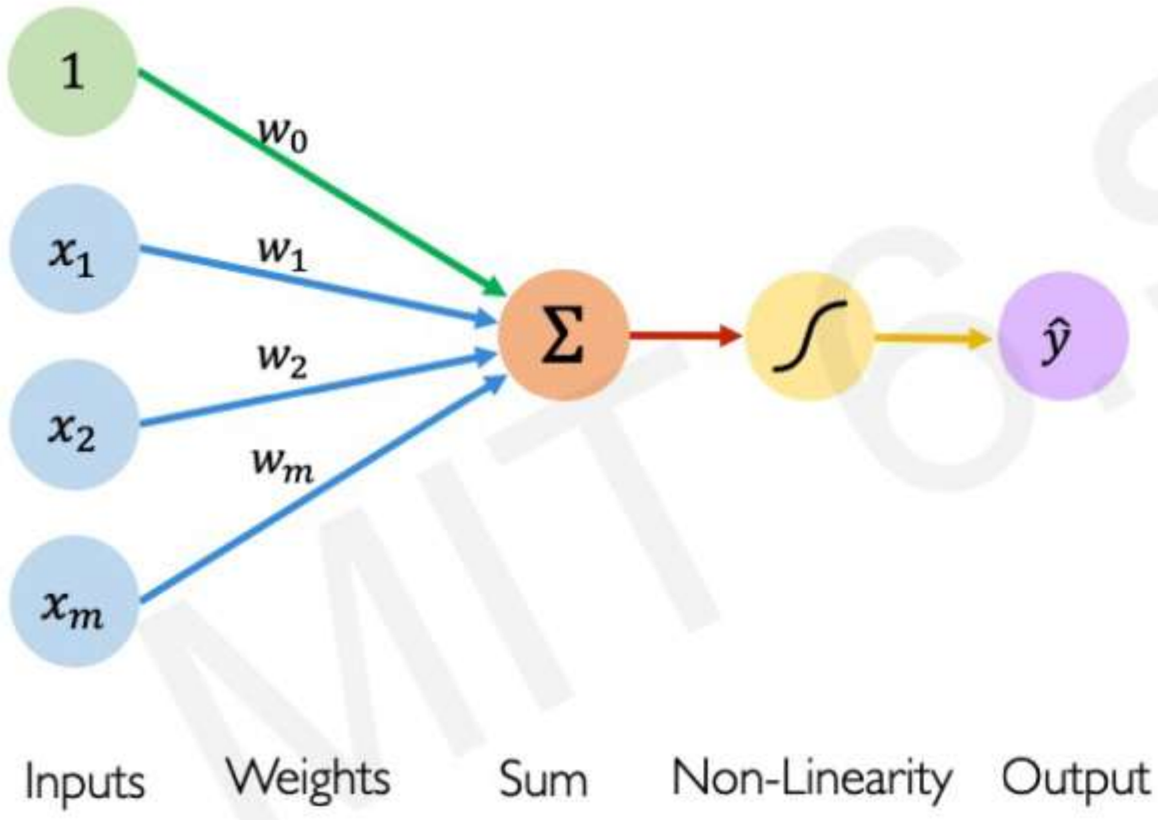
The Perceptron

The structural building block of deep learning

The Perceptron: Forward Propagation



The Perceptron: Forward Propagation



Output

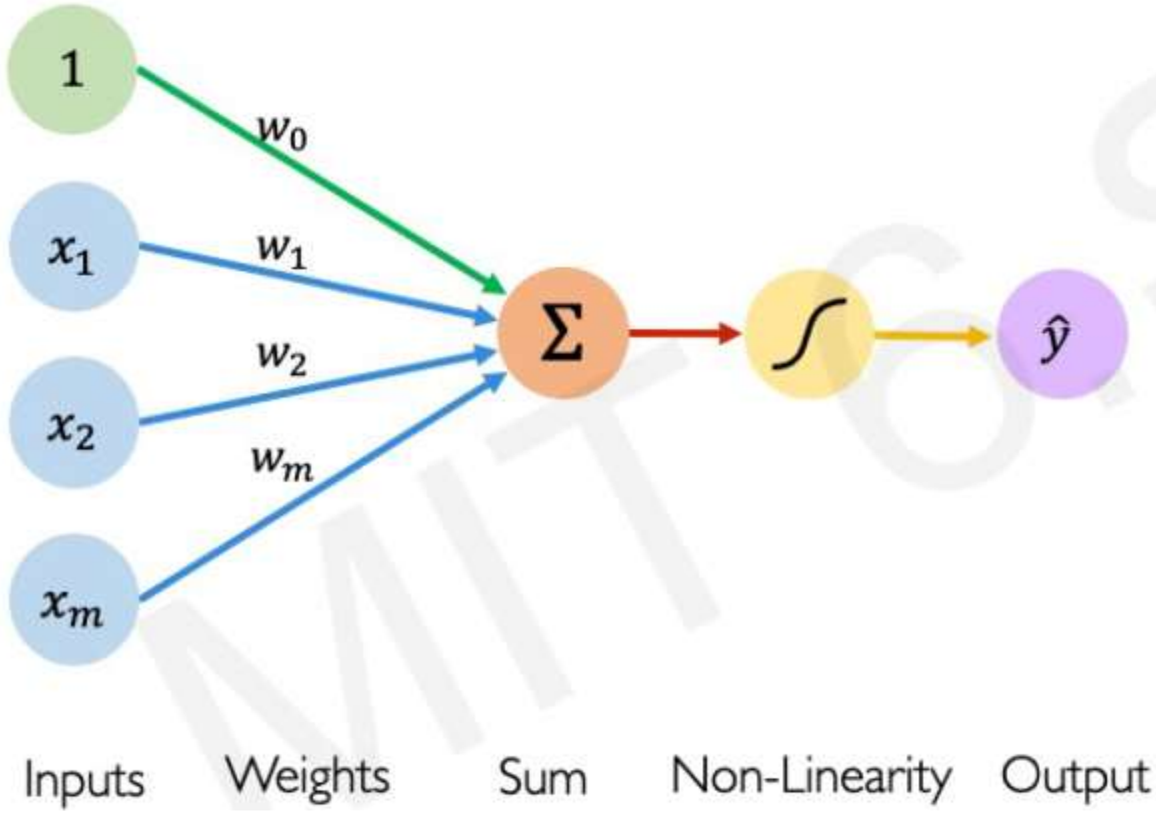
Linear combination of inputs

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

Non-linear activation function

Bias

The Perceptron: Forward Propagation

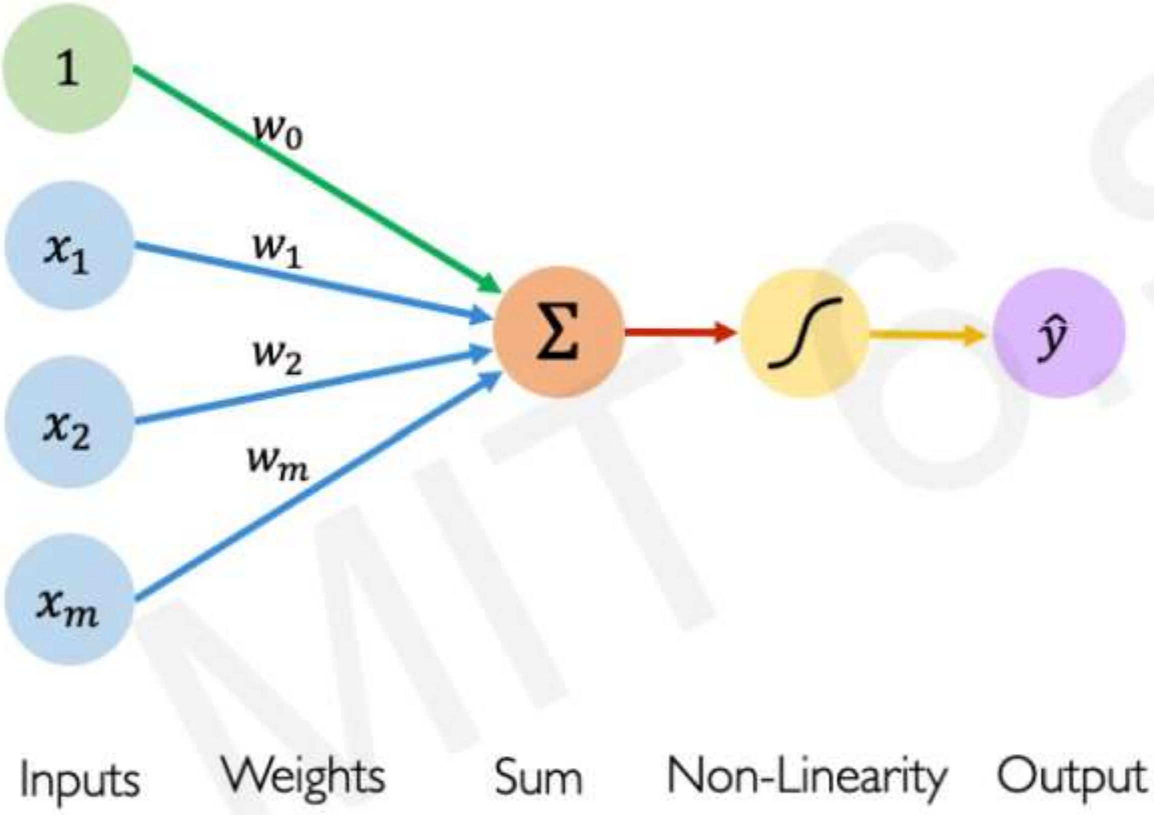


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i \, w_i \right)$$

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

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

The Perceptron: Forward Propagation

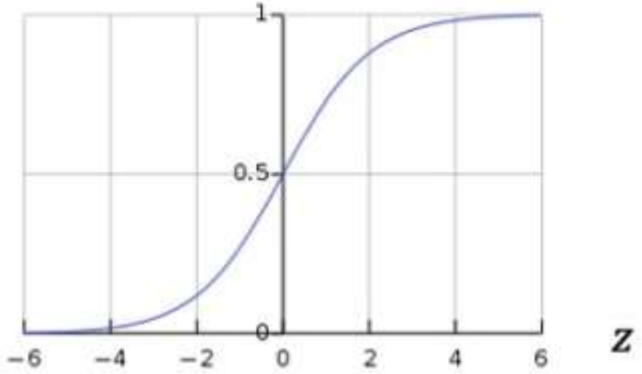


Activation Functions

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

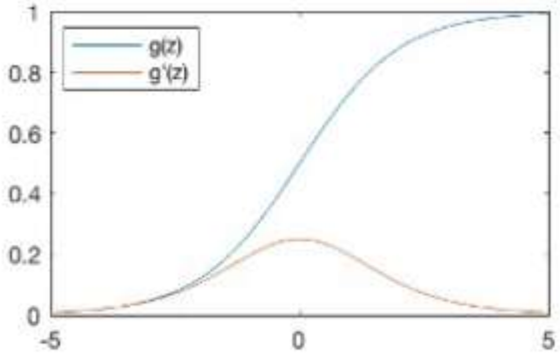
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

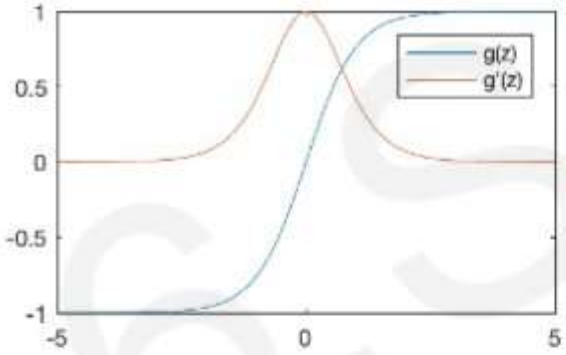


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.math.sigmoid(z)`

Hyperbolic Tangent

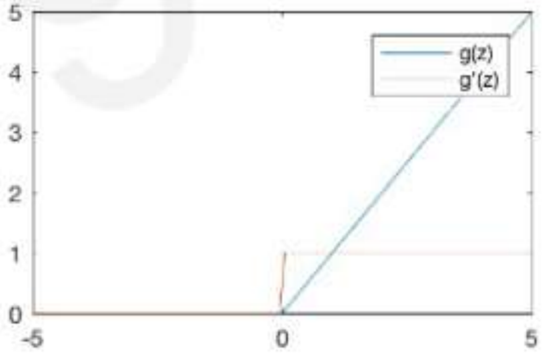


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



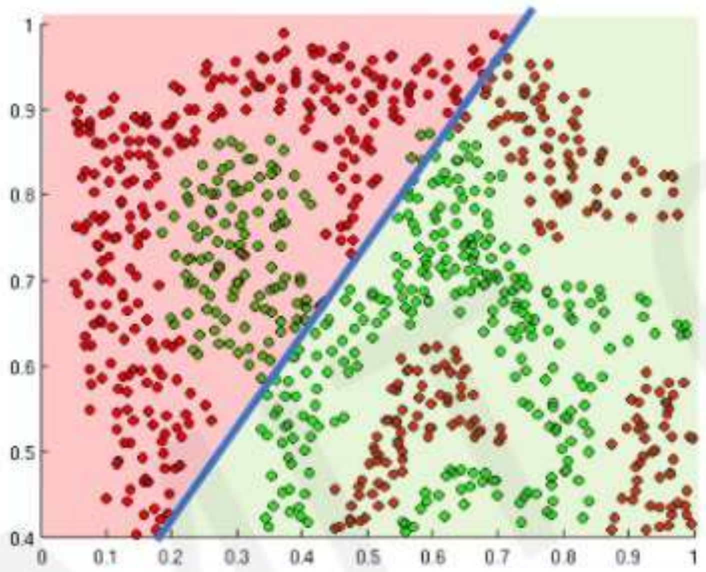
$$g(z) = \max(0, z)$$

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

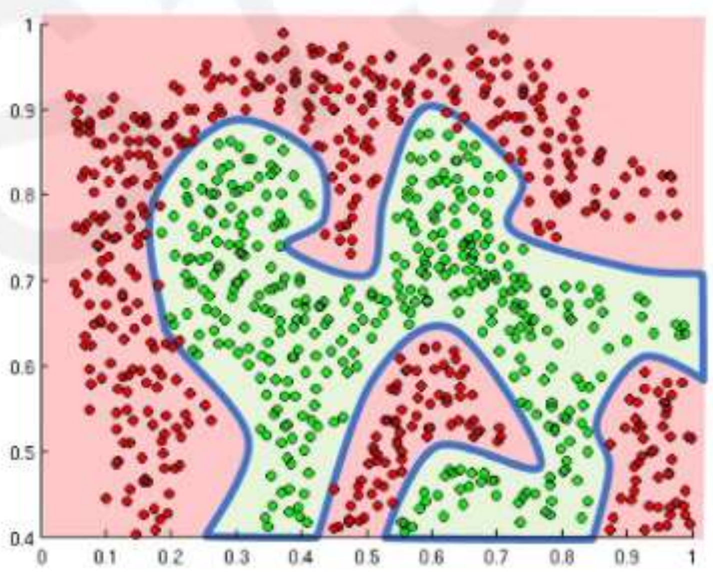
`tf.nn.relu(z)`

Importance of Activation Functions

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

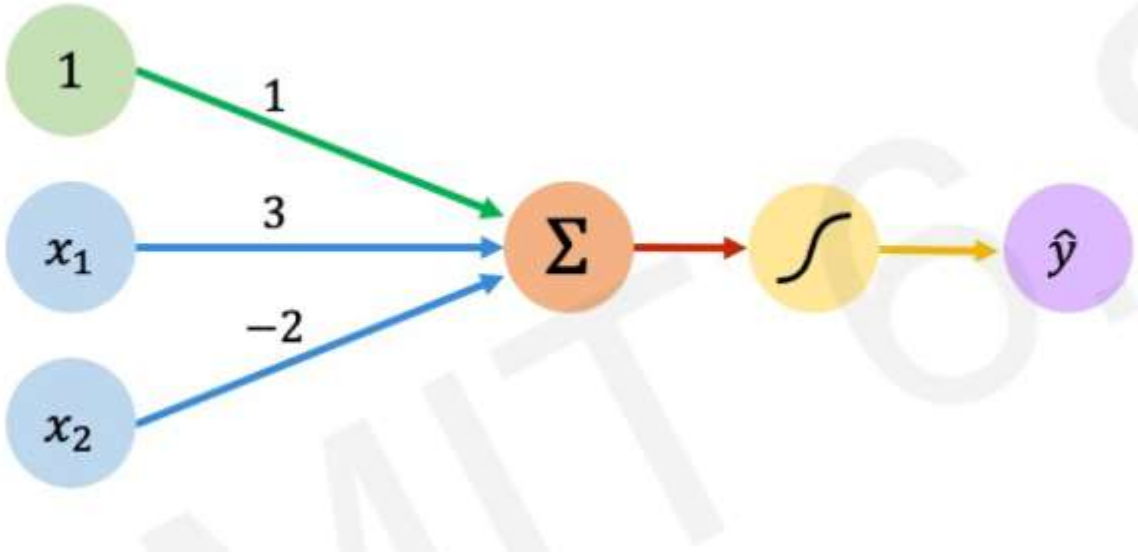


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

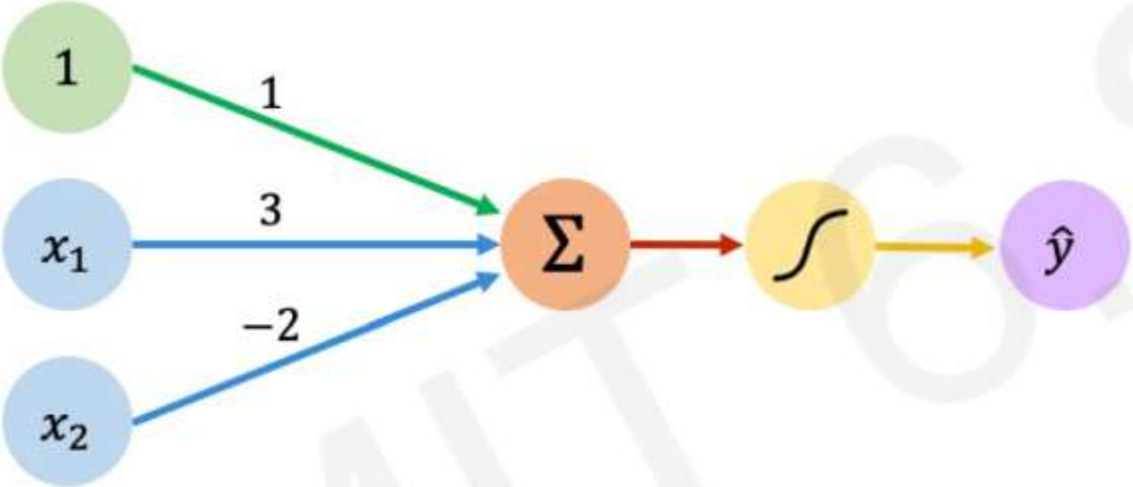
The Perceptron: Example



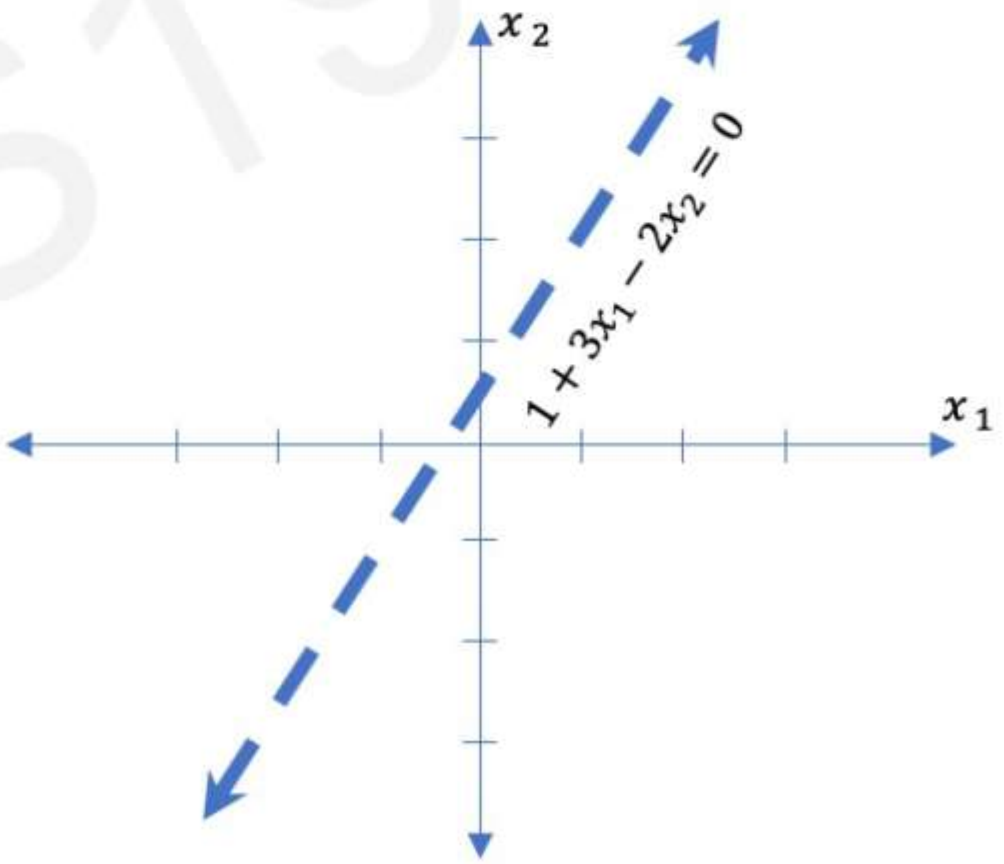
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned} \hat{y} &= g\,(\,w_0 + \boldsymbol{X}^T \boldsymbol{W}\,) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g\,(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}}) \end{aligned}$$

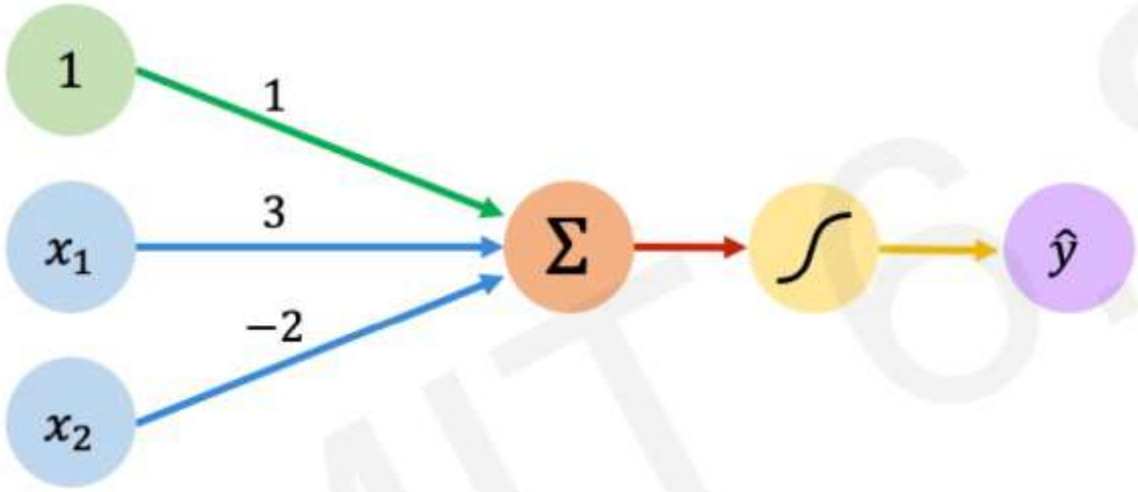
The Perceptron: Example



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

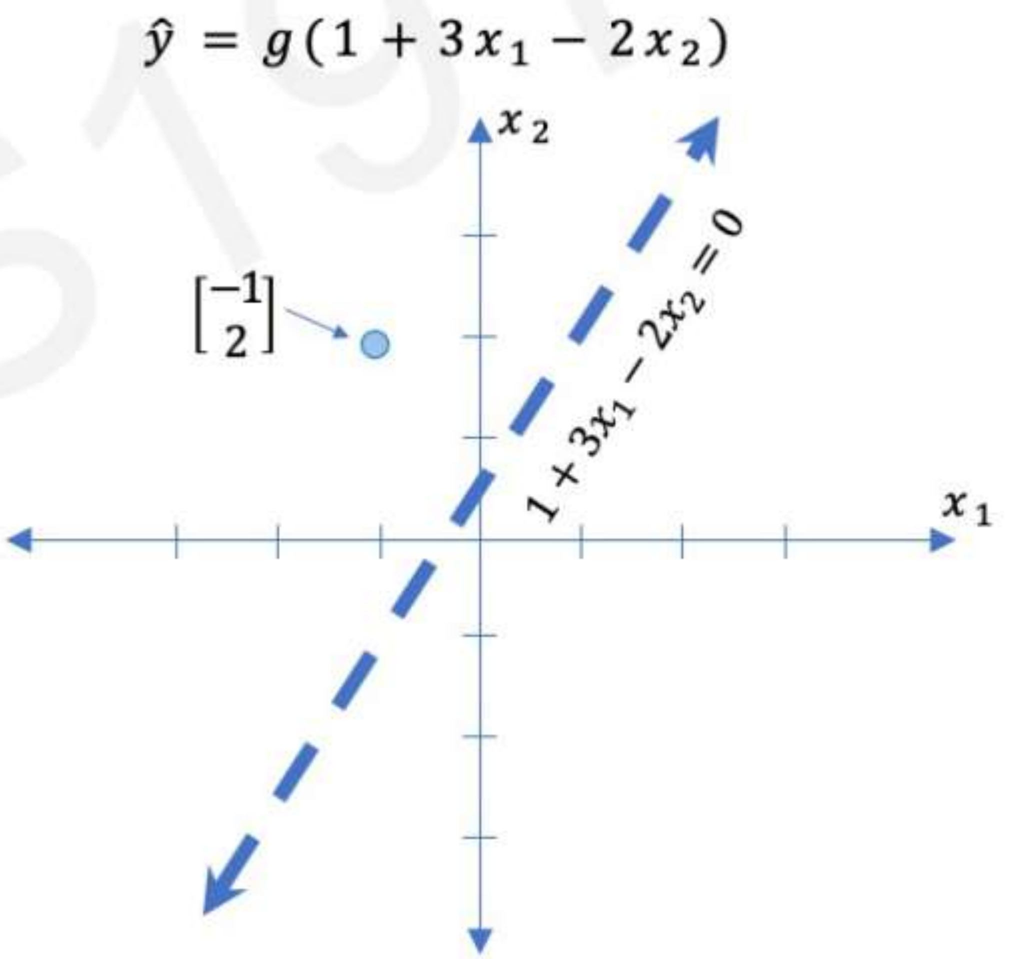


The Perceptron: Example

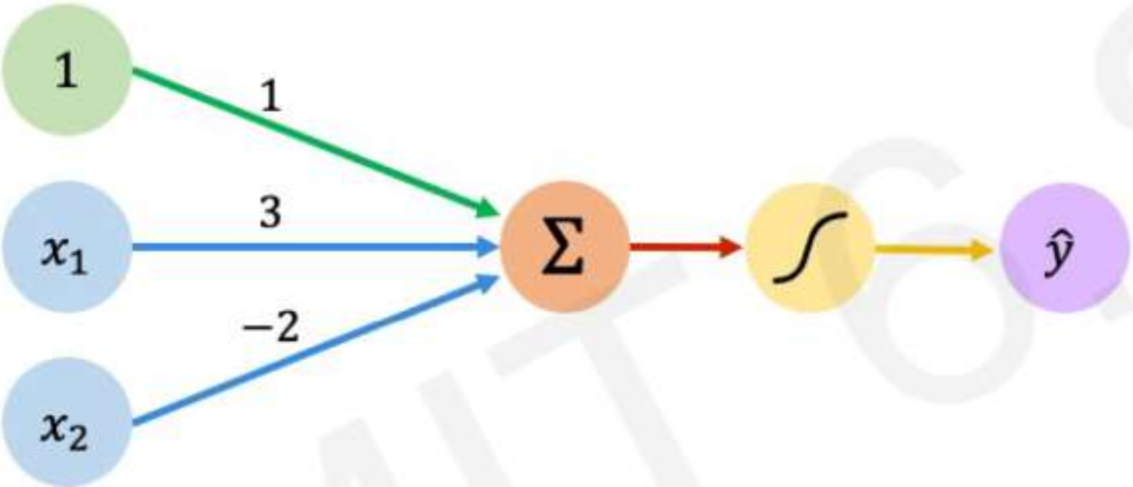


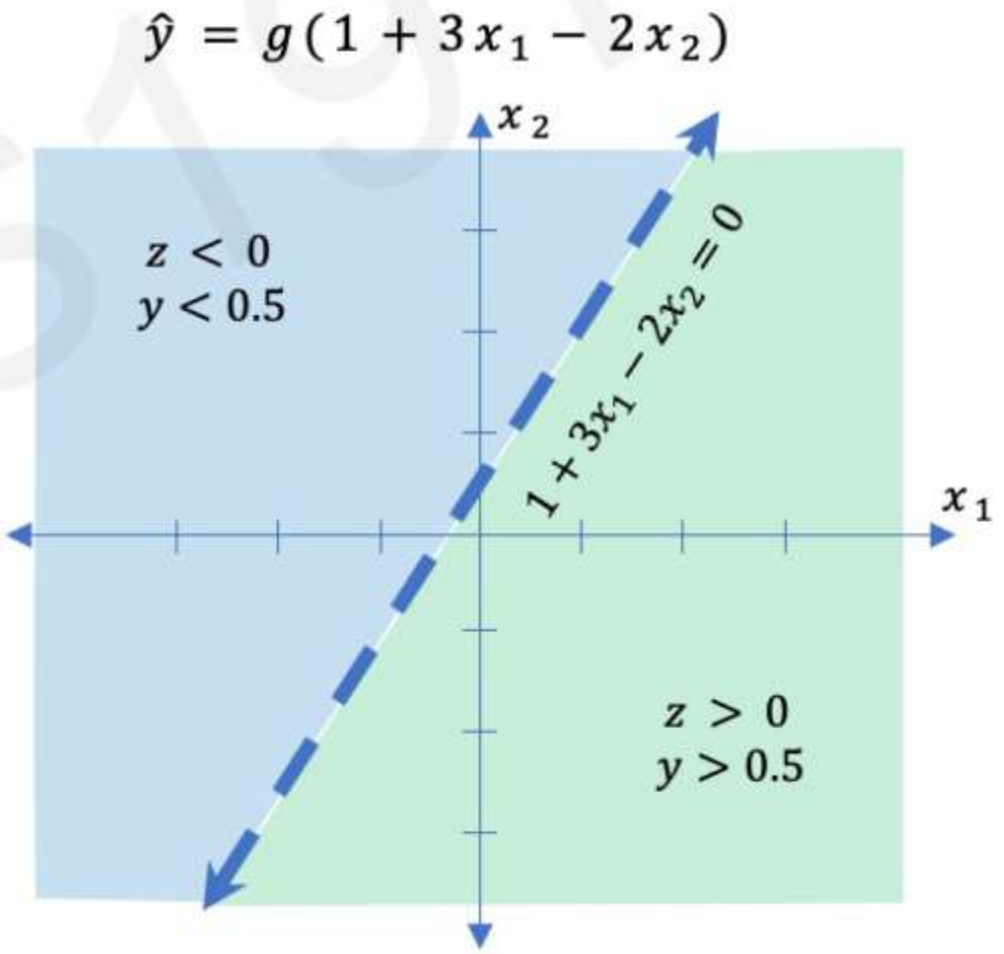
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$



The Perceptron: Example

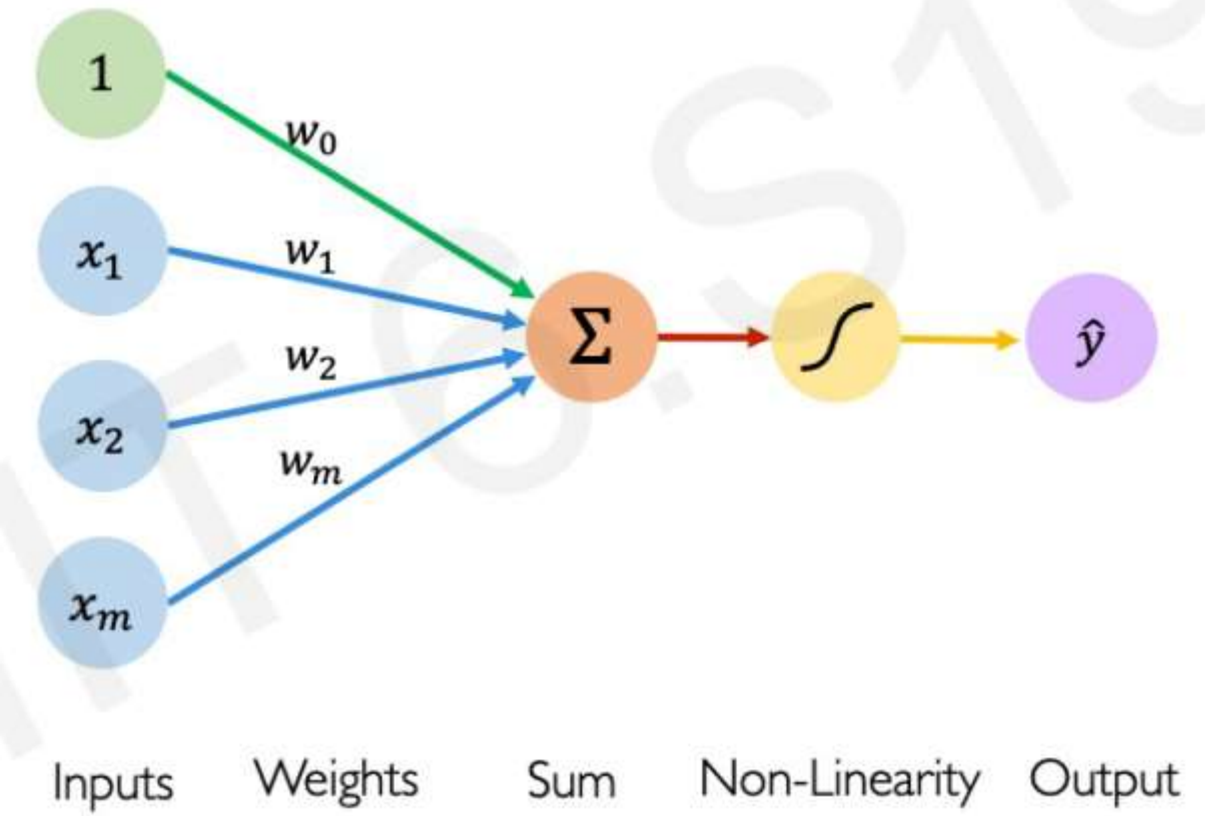




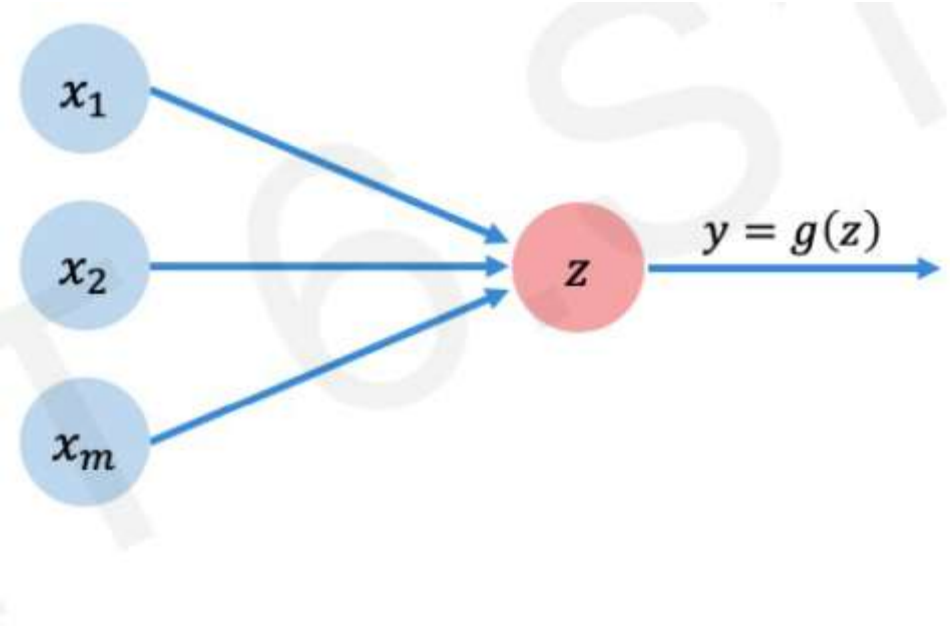
Building Neural Networks with Perceptrons

The Perceptron Simplified

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



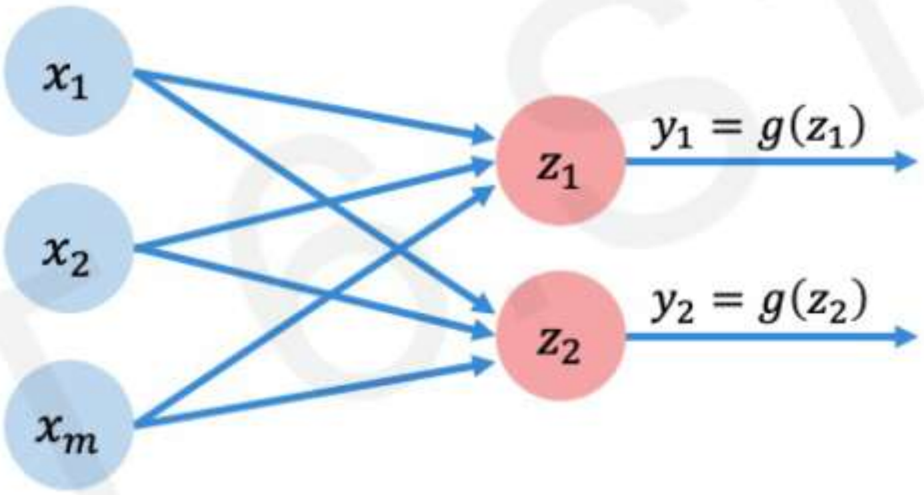
The Perceptron Simplified



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

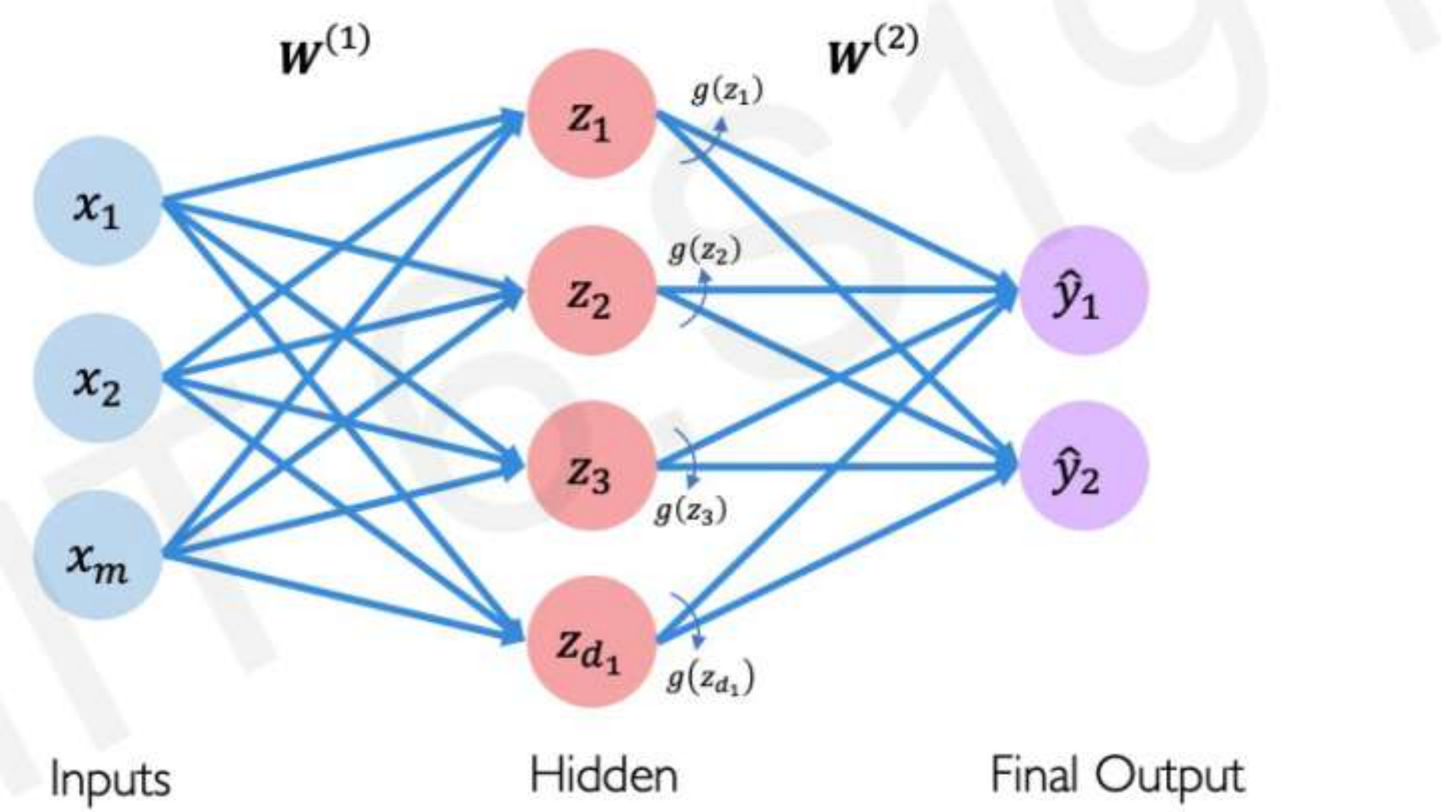
Multi Output Perceptron

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



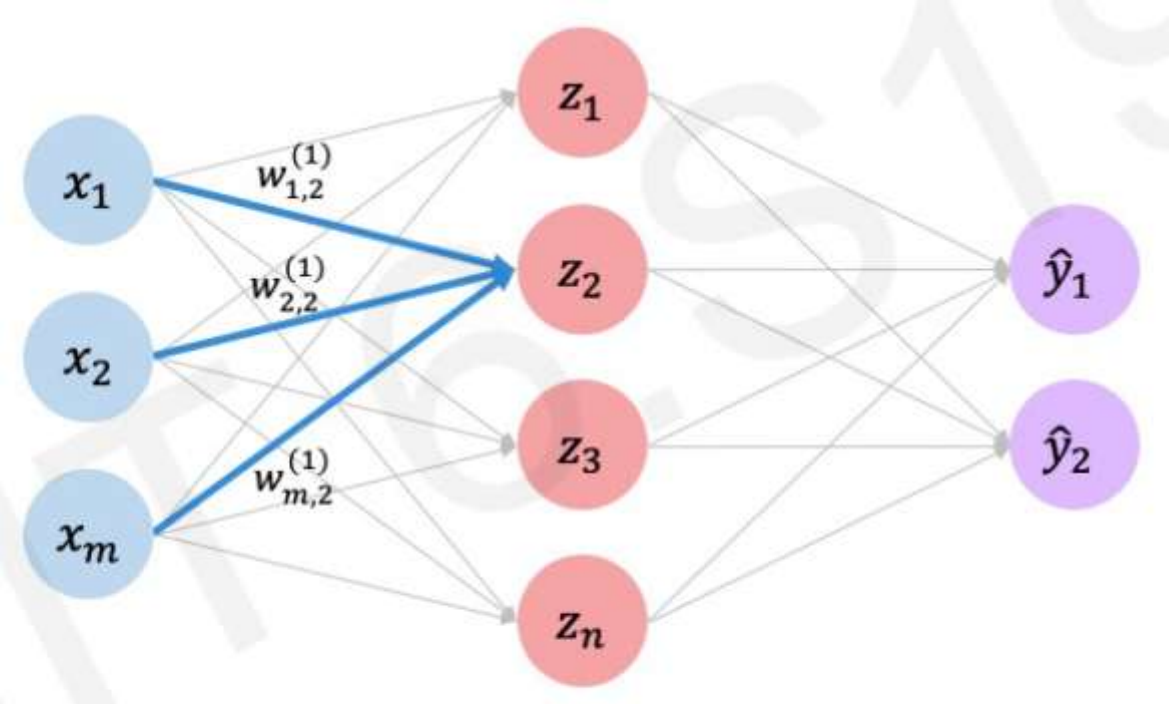
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Single Layer Neural Network



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

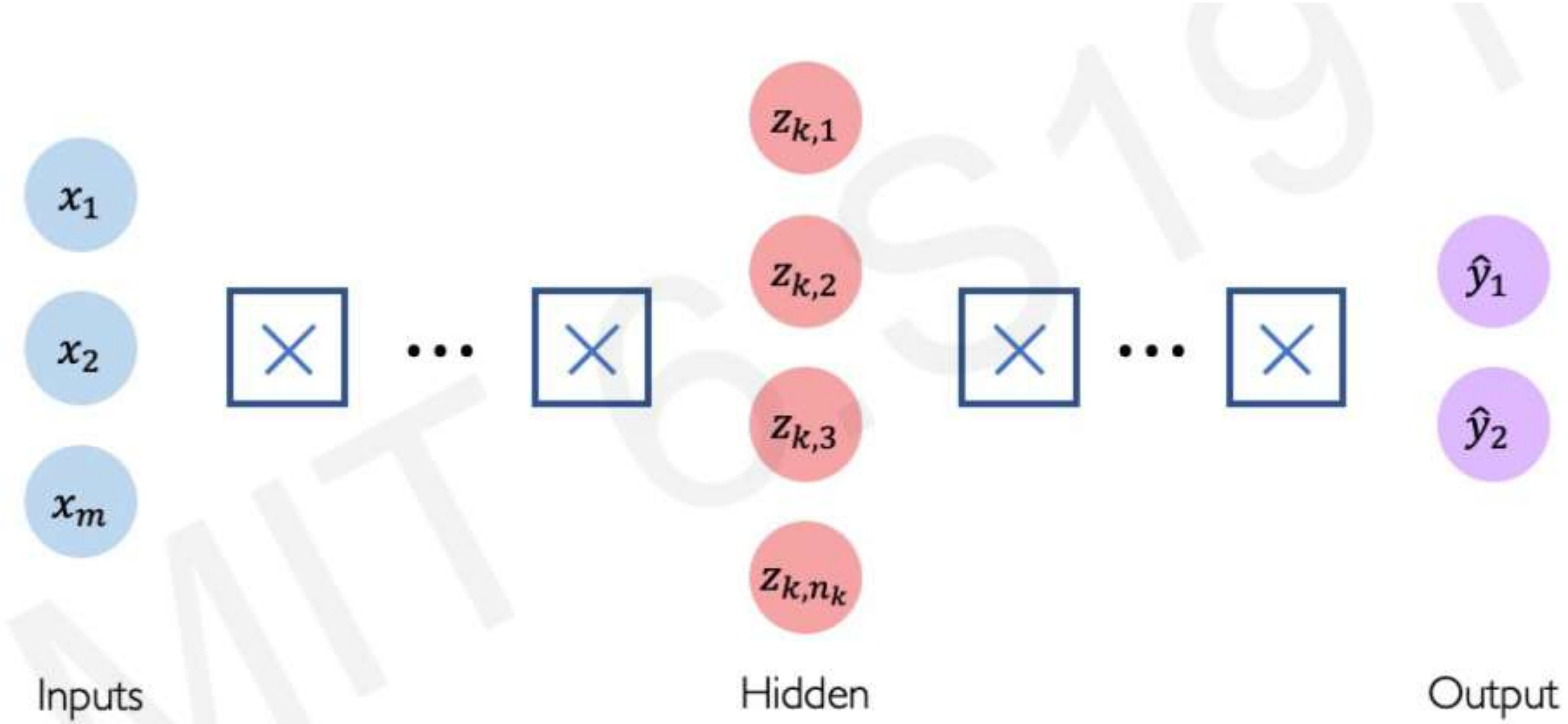
Single Layer Neural Network



$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j\, w_{j,2}^{(1)}$$

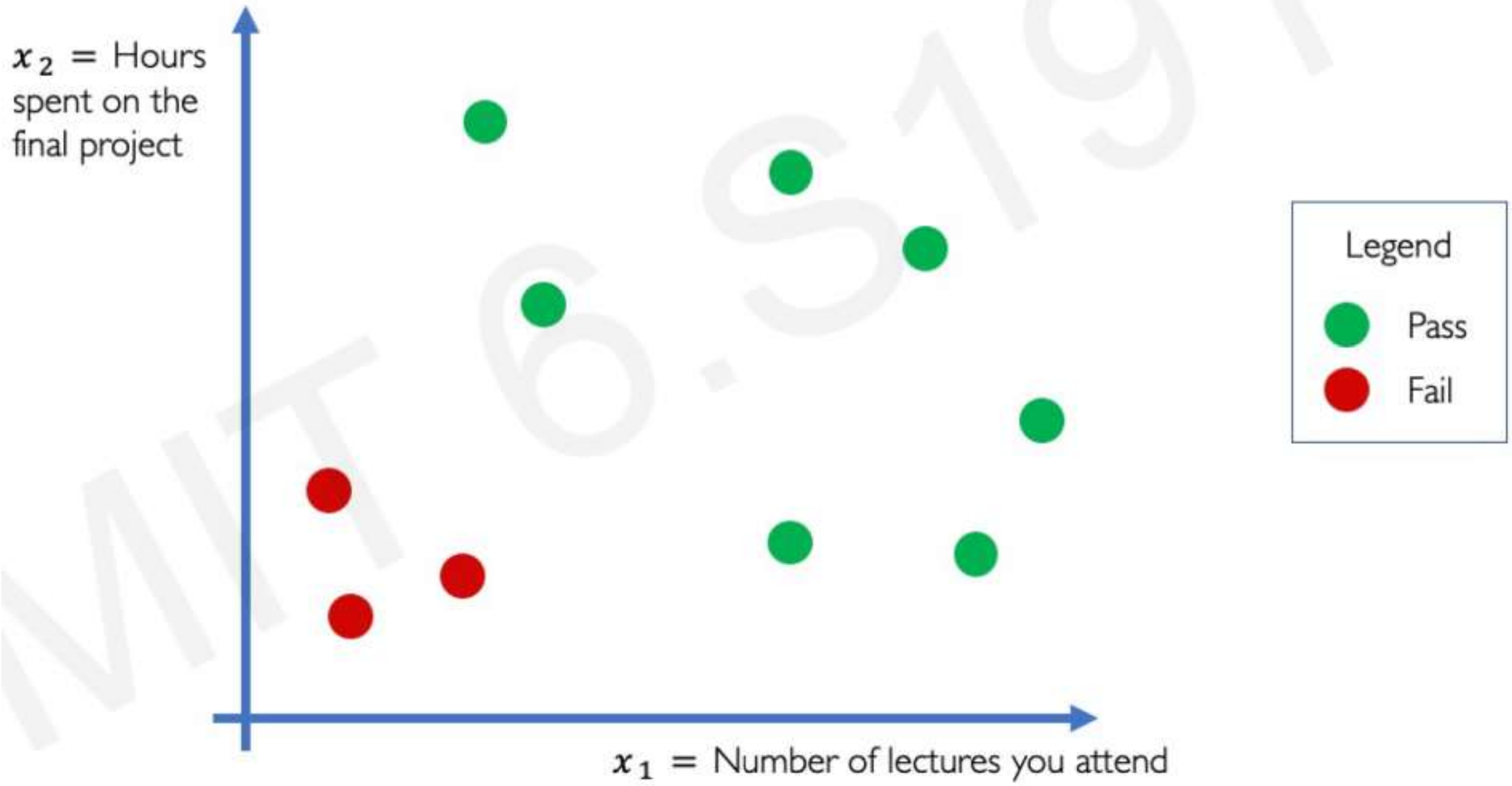
$$= w_{0,2}^{(1)} + x_1\, w_{1,2}^{(1)} + x_2\, w_{2,2}^{(1)} + x_m\, w_{m,2}^{(1)}$$

Deep Neural Network

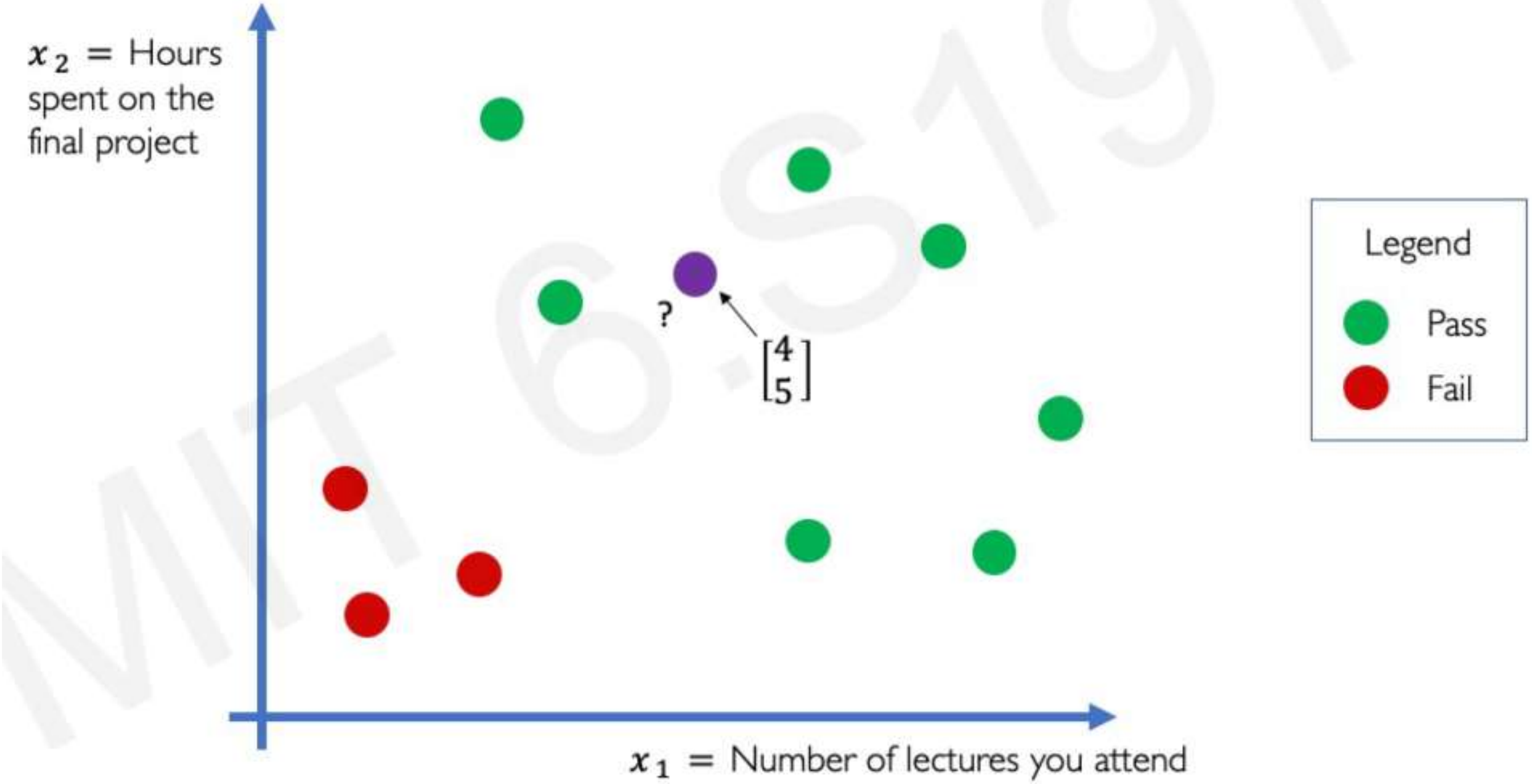


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

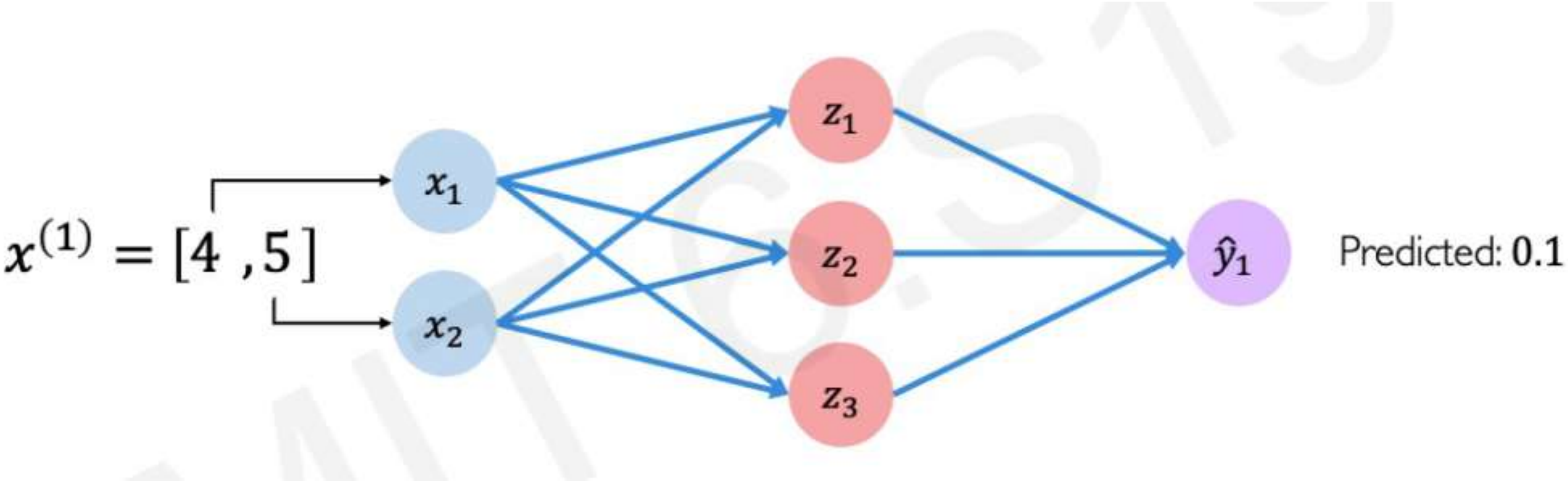
Example Problem: Will I Pass This Class?



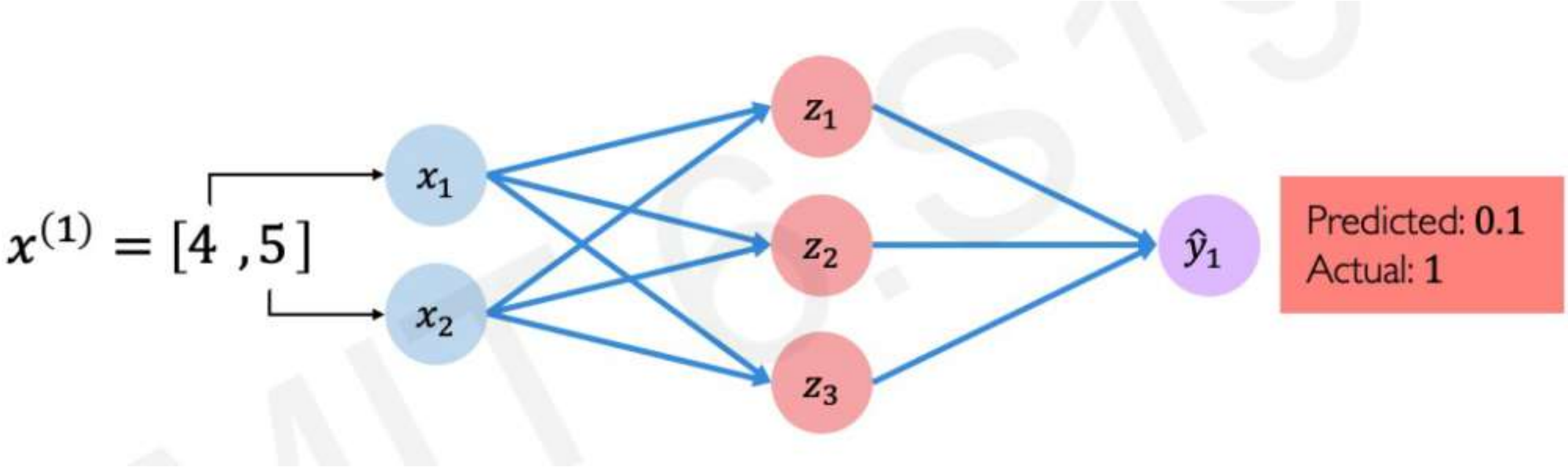
Example Problem: Will I Pass This Class?



Example Problem: Will I Pass This Class?

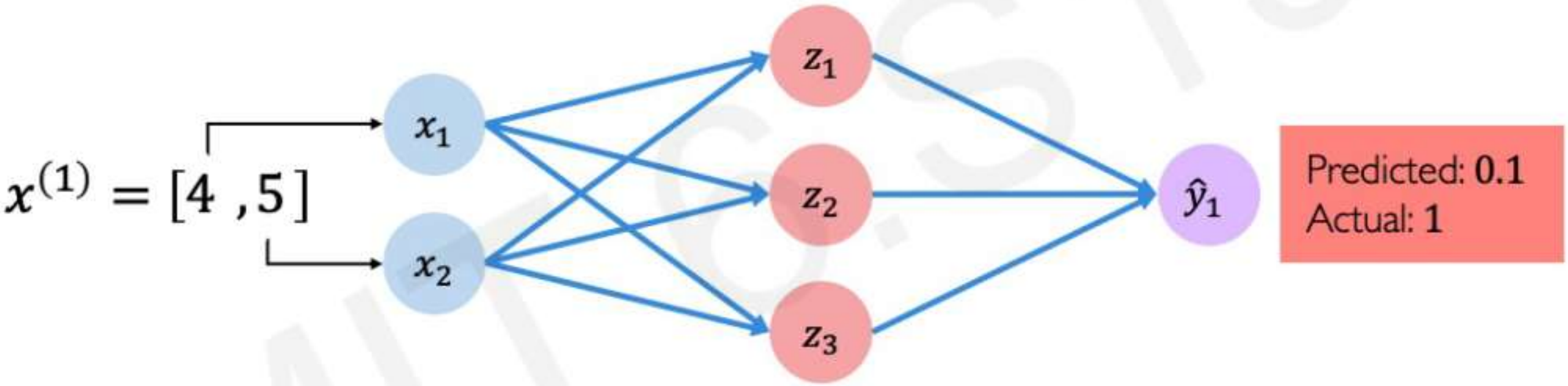


Example Problem: Will I Pass This Class?



Example Problem: Will I Pass This Class?

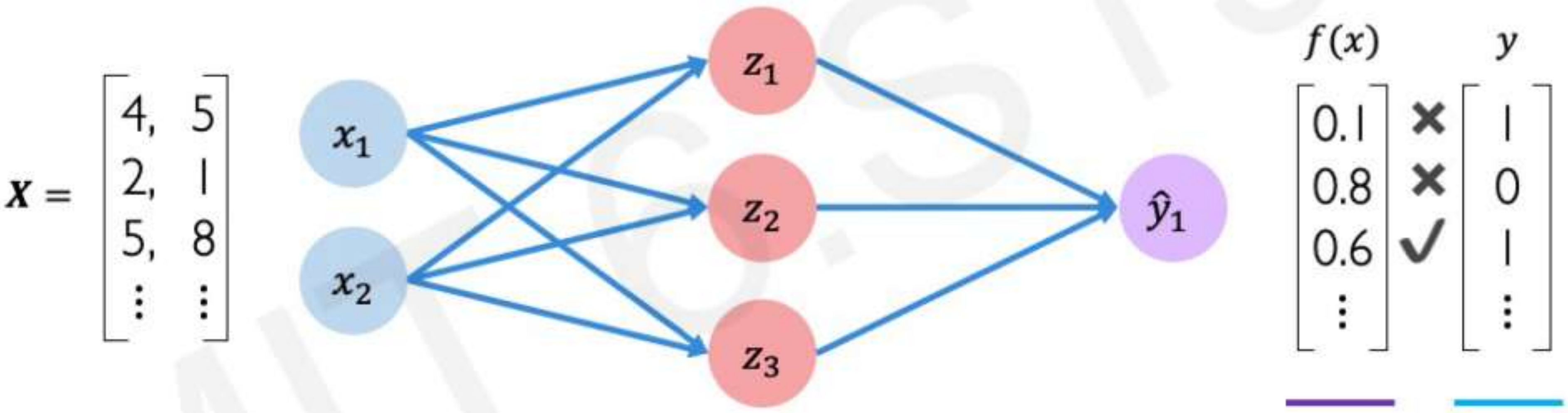
*The **loss** of our network measures the cost incurred from incorrect predictions*



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

Empirical Loss

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



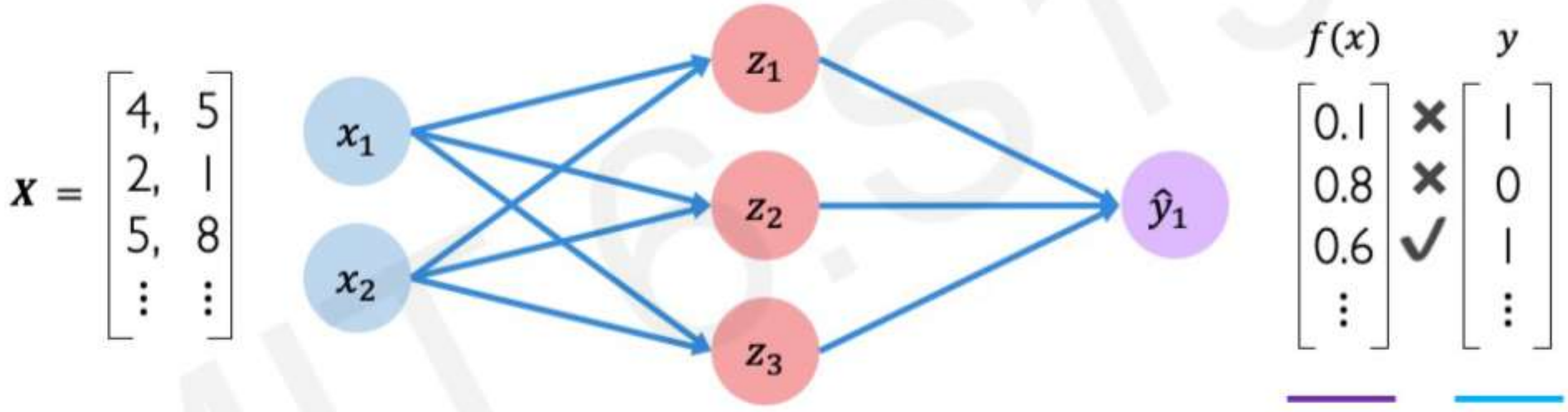
Also known as:

- Objective function
- Cost function
- Empirical Risk

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

Binary Cross Entropy Loss

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

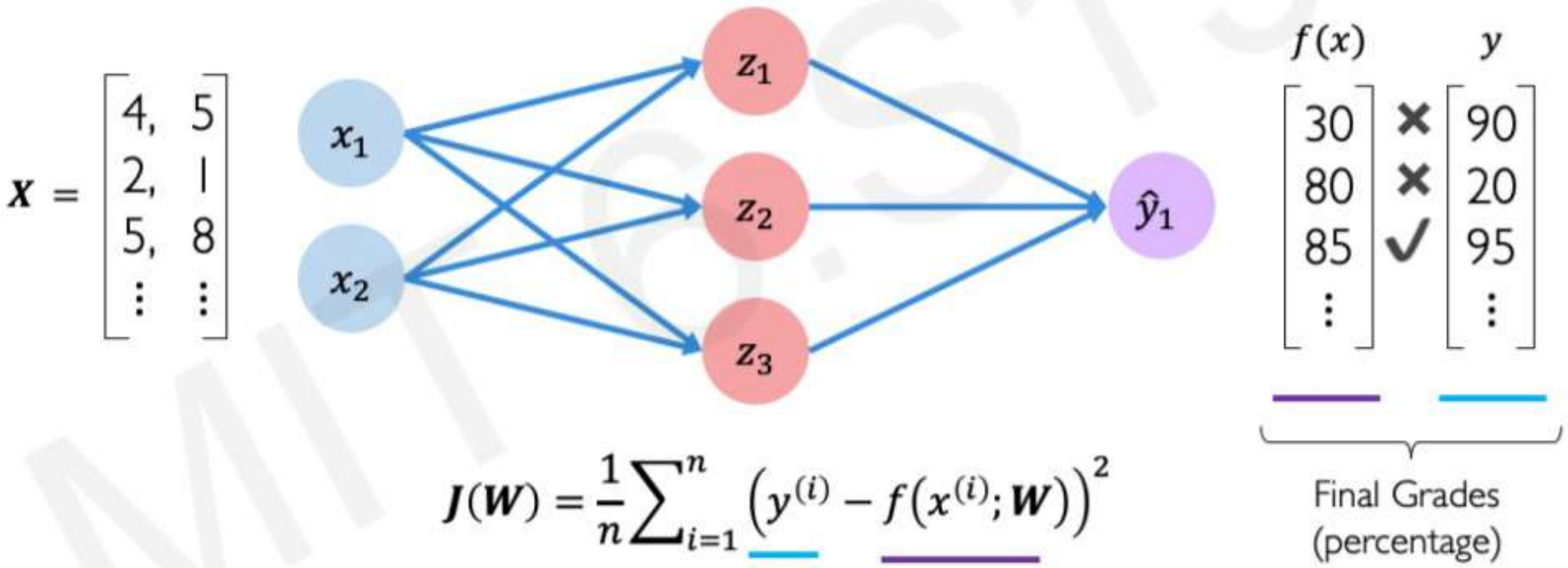


$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1-\underbrace{y^{(i)}}_{\text{Actual}})\log\Big(1-\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
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*



$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\Big(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)^2$$

```
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**

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

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

Loss Optimization

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

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

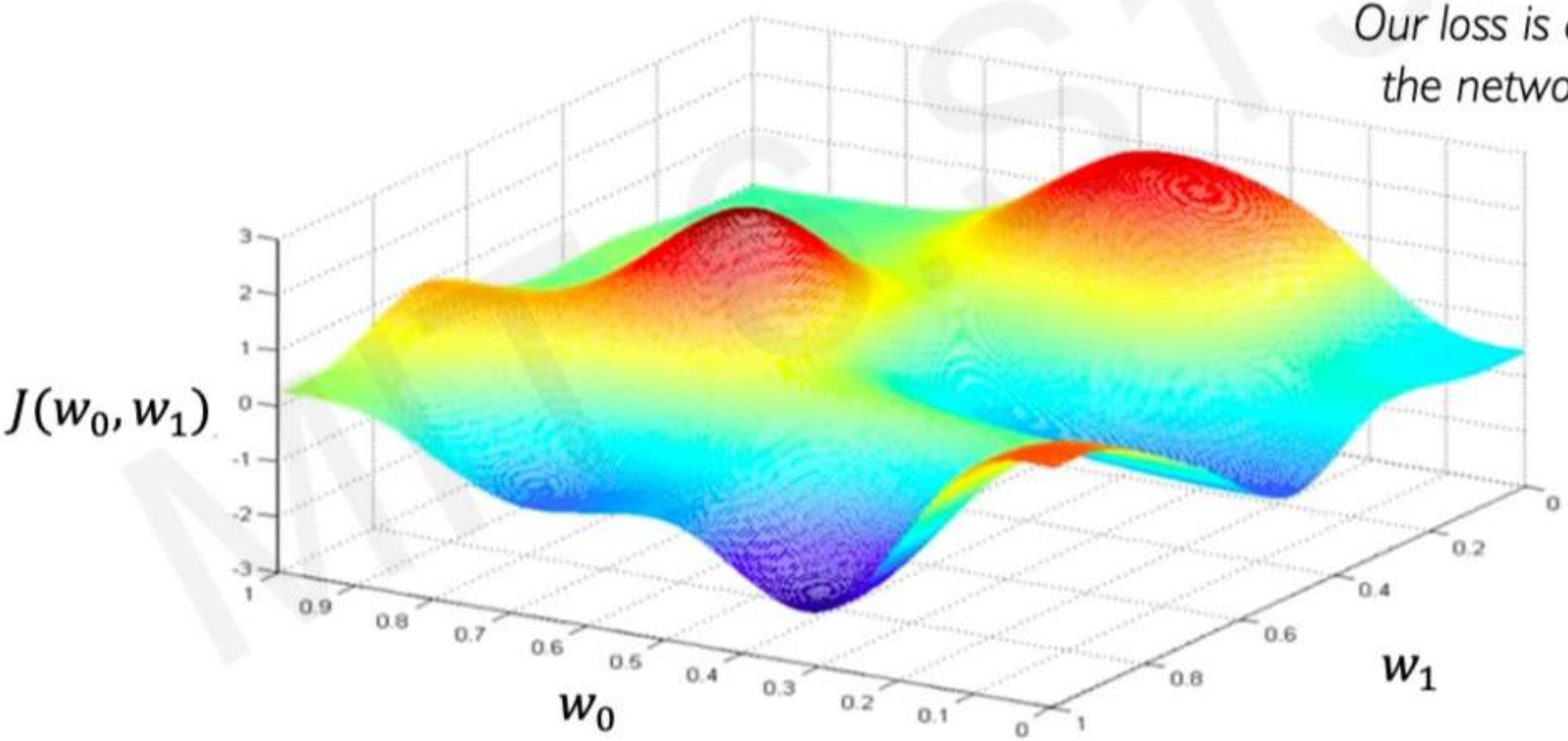
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

Remember:
$\boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \cdots\}$

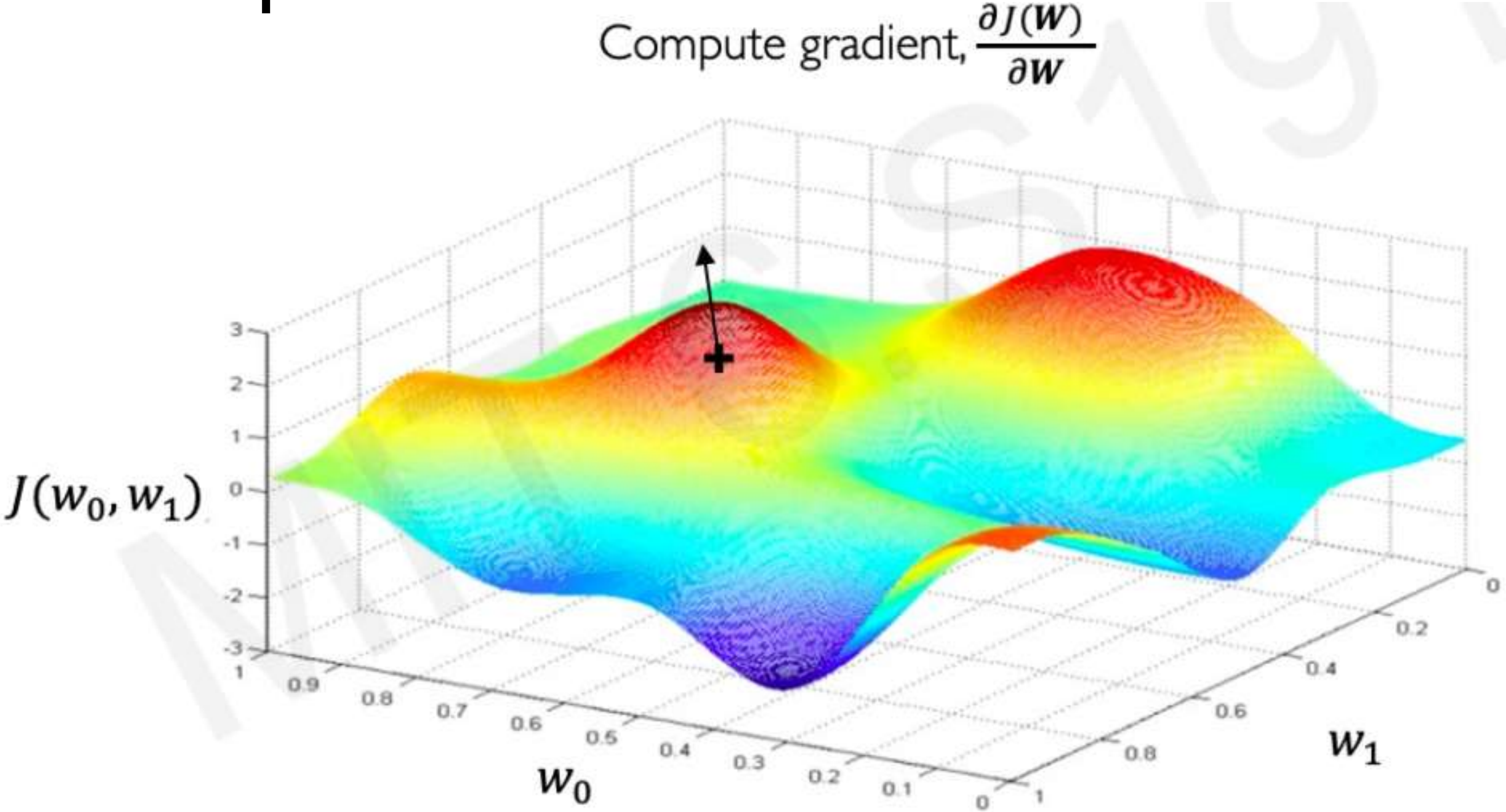
Loss Optimization

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

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

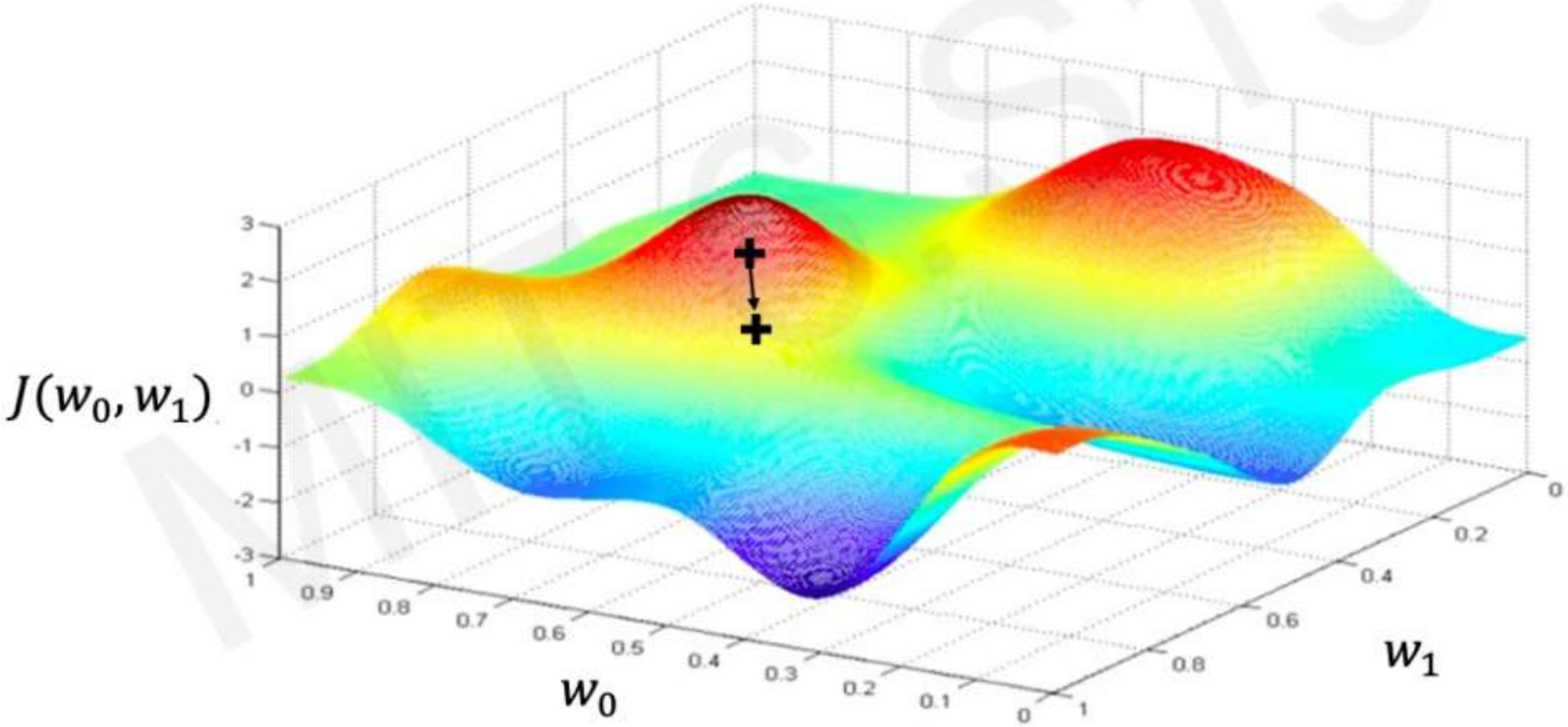


Loss Optimization

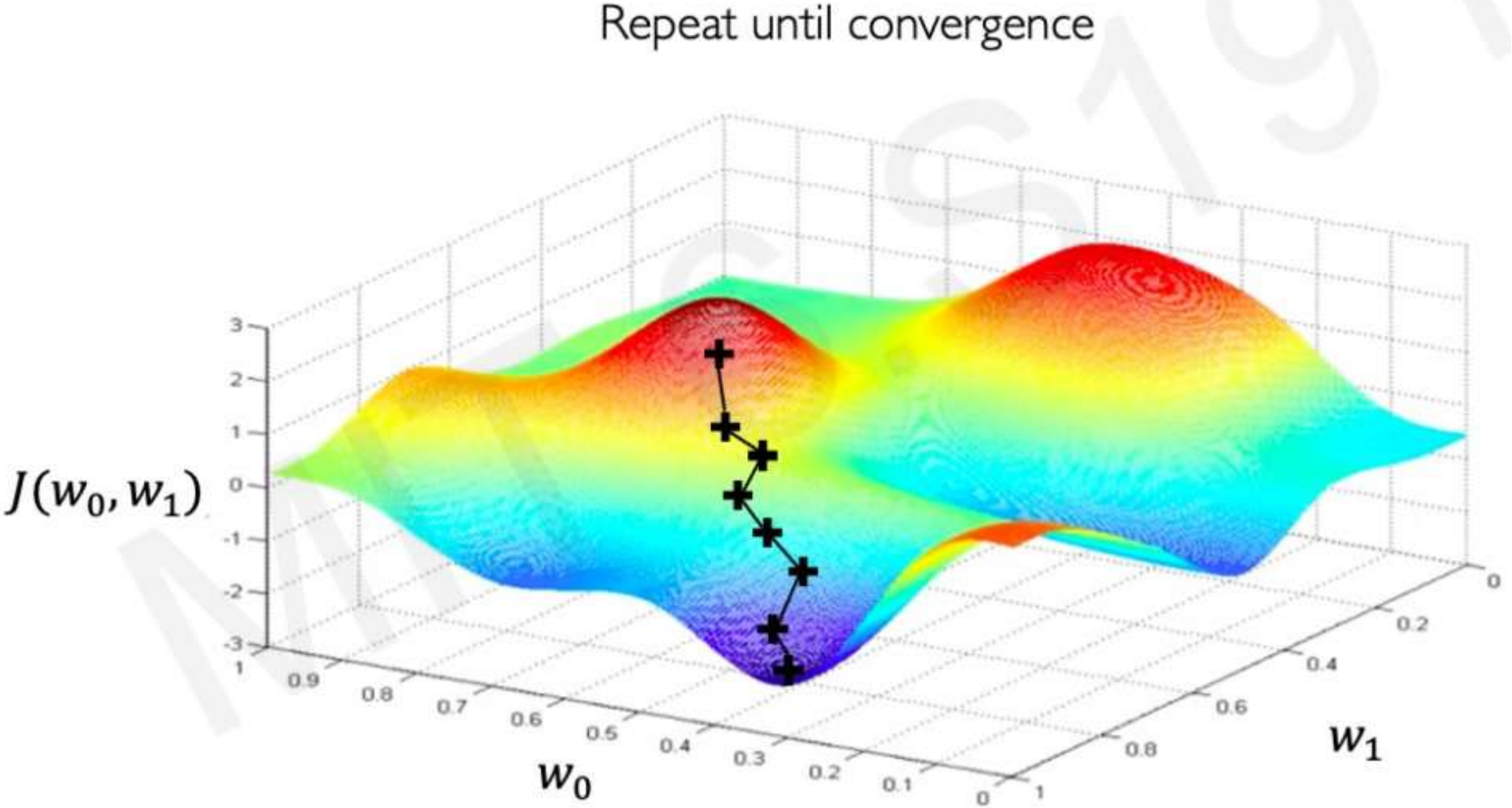


Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

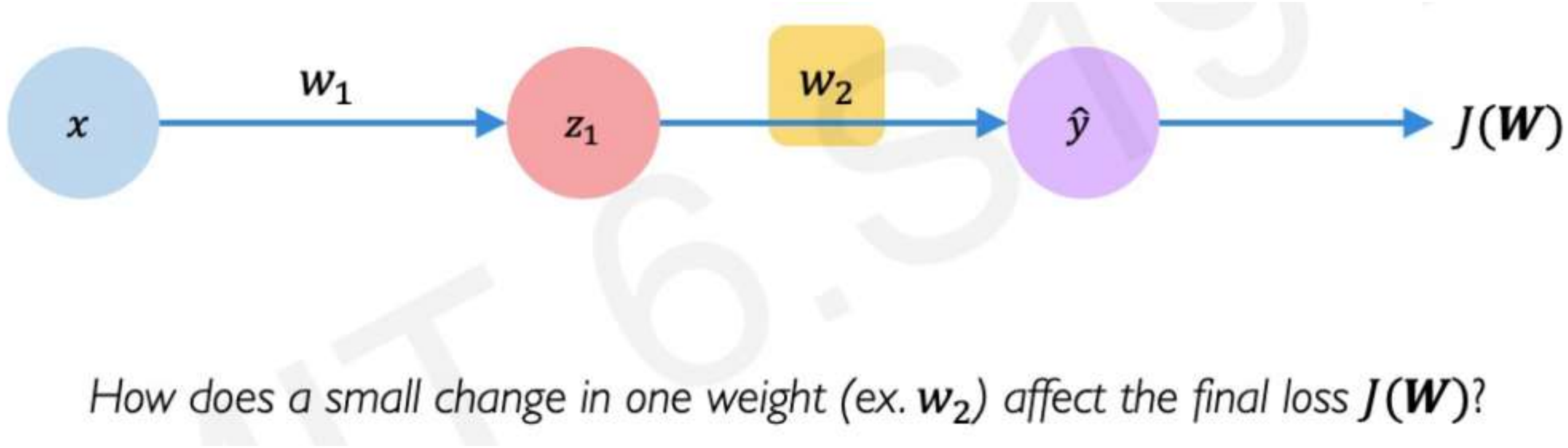


Gradient Descent

Algorithm

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

Computing Gradients: Backpropagation

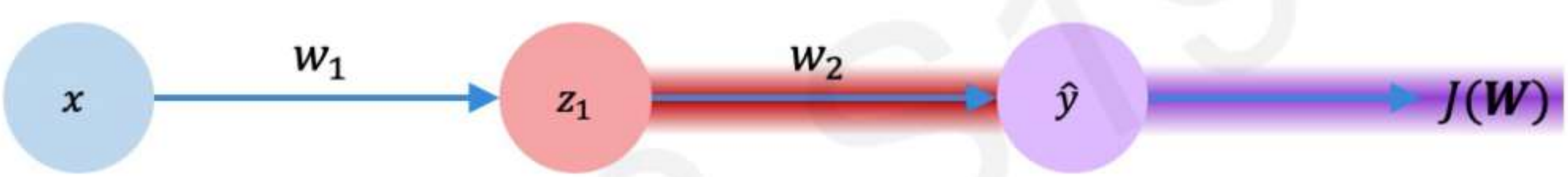


Computing Gradients: Backpropagation

$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} =$$

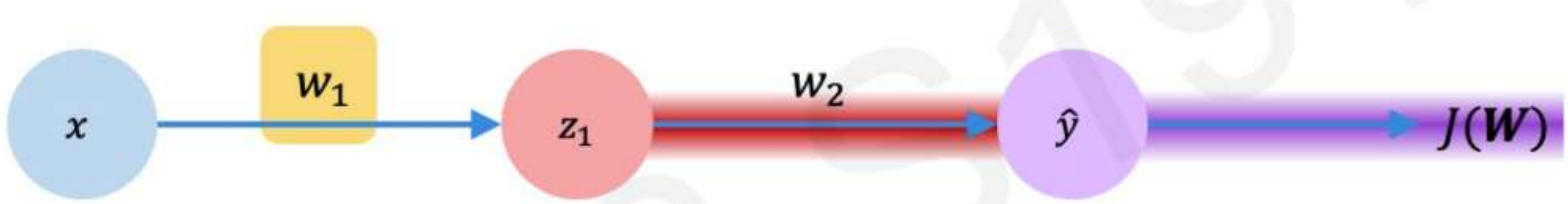
Let's use the chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

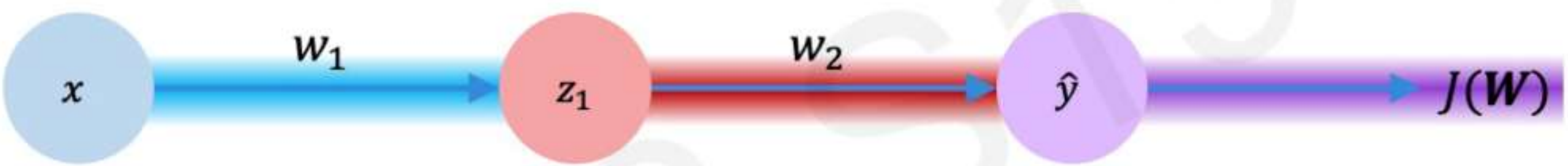
Computing Gradients: Backpropagation



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

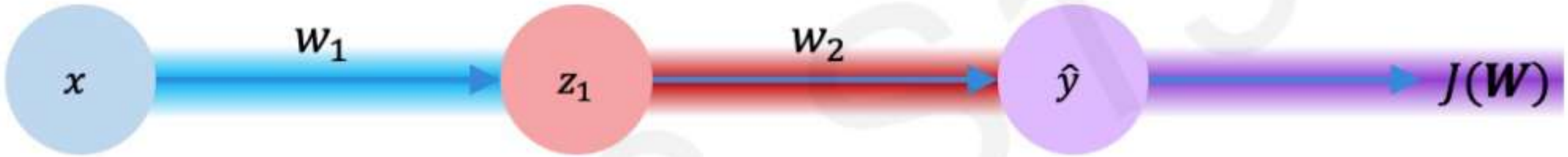
Apply chain rule! Apply chain rule!

Computing Gradients: Backpropagation



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

Computing Gradients: Backpropagation



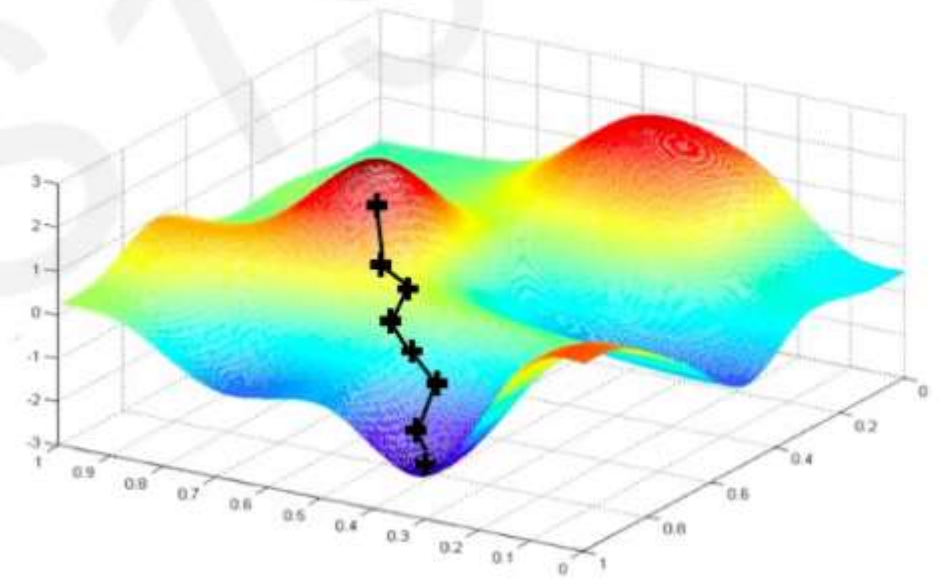
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \underbrace{\frac{\partial J(\boldsymbol{W})}{\partial \hat{y}}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}} * \underbrace{\frac{\partial z_1}{\partial w_1}}$$

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

Gradient Descent

Algorithm

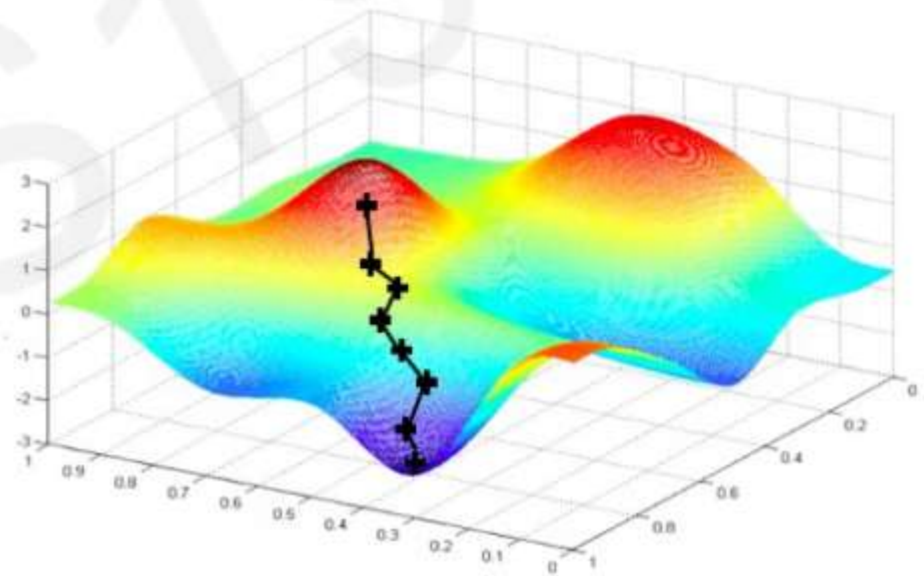
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5.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



Gradient Descent

Algorithm

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

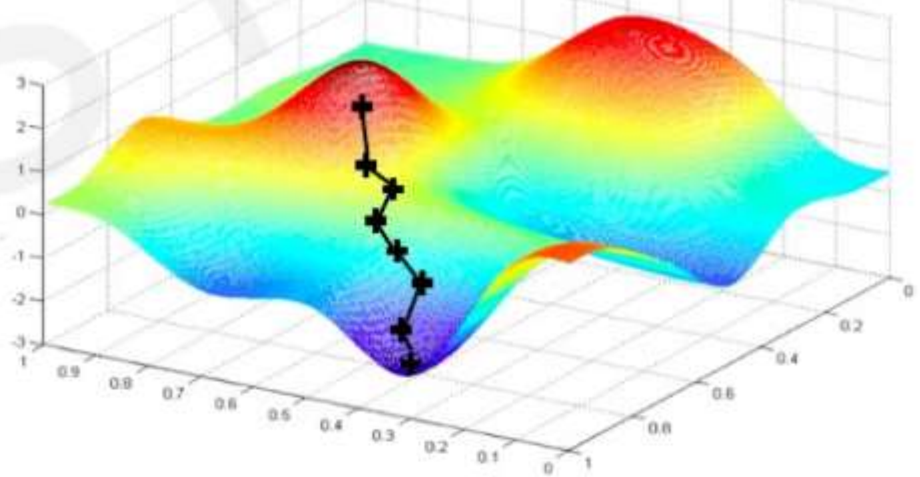


Can be very **computationally intensive** to compute!

Stochastic Gradient Descent

Algorithm

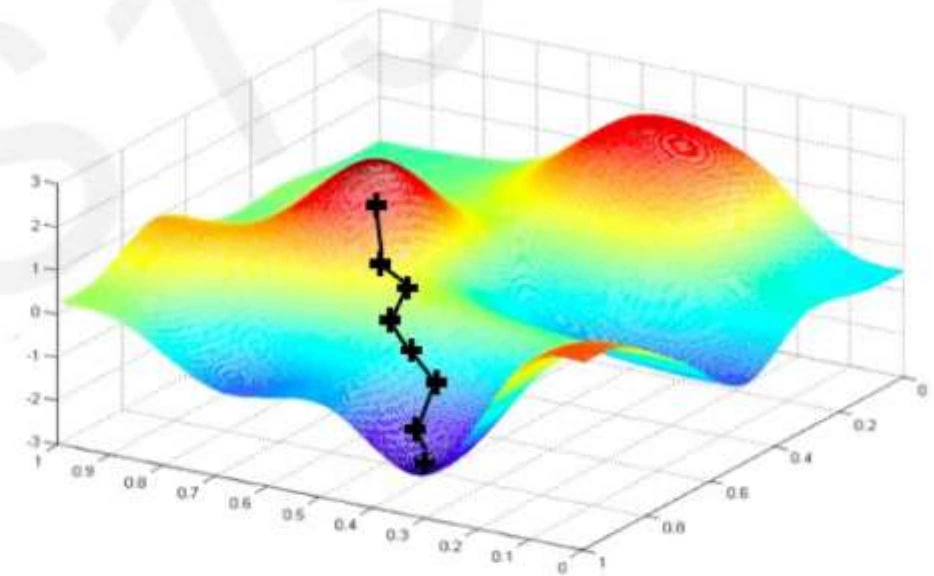
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial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

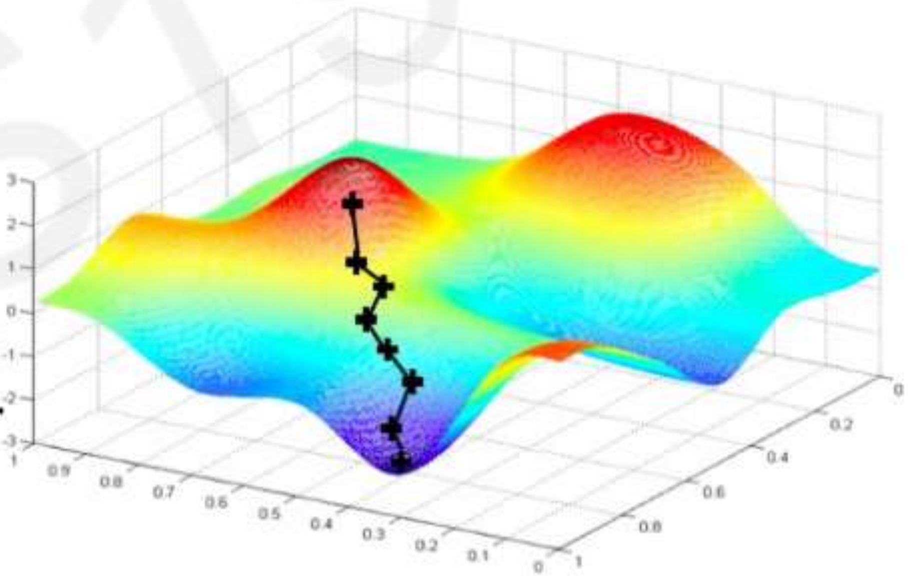


Easy to compute but
very noisy (stochastic)!

Stochastic Gradient Descent

Algorithm

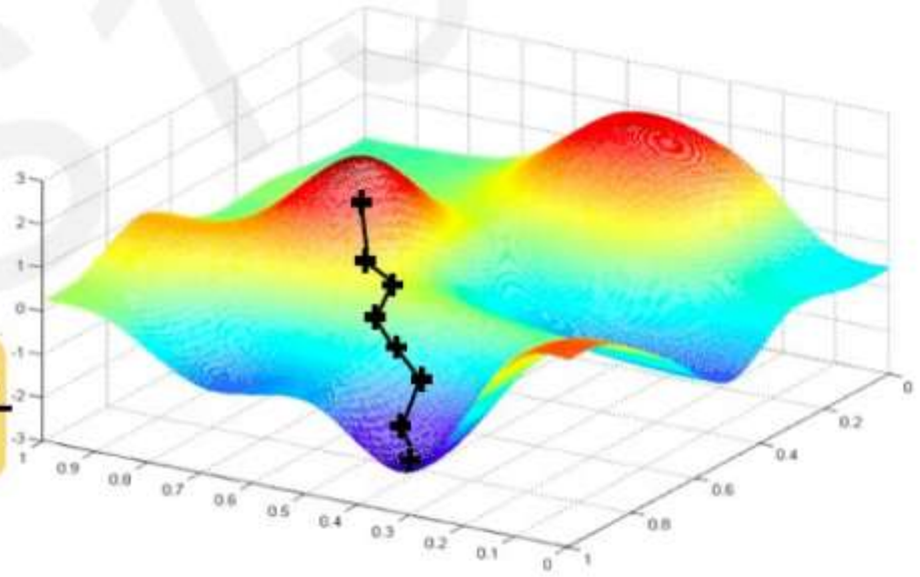
1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update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

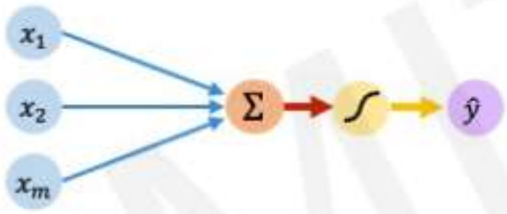
Mini-batches lead to fast training!

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

Neural Networks Summary

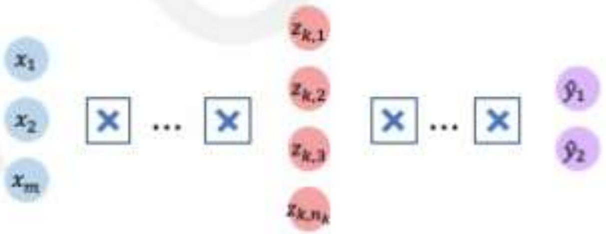
The Perceptron

- Structural building blocks
- Nonlinear activation functions



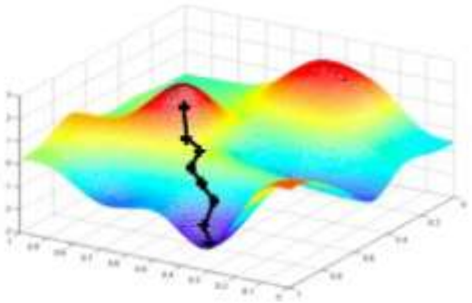
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



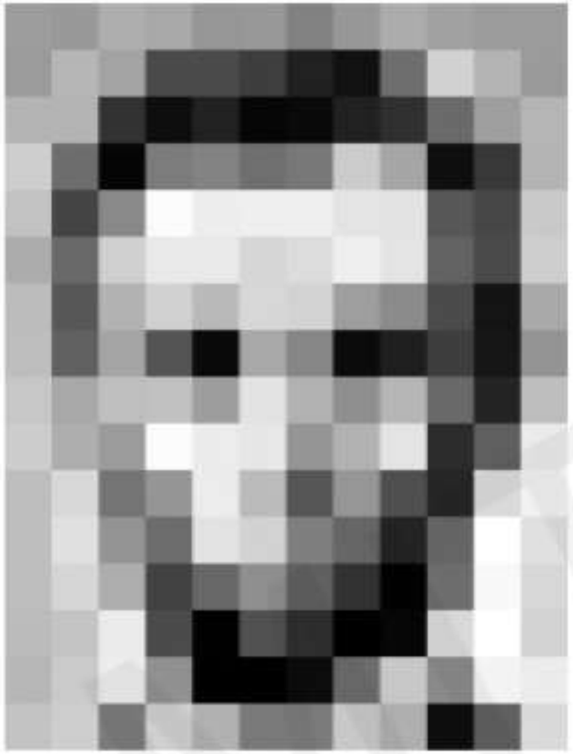
Training in Practice

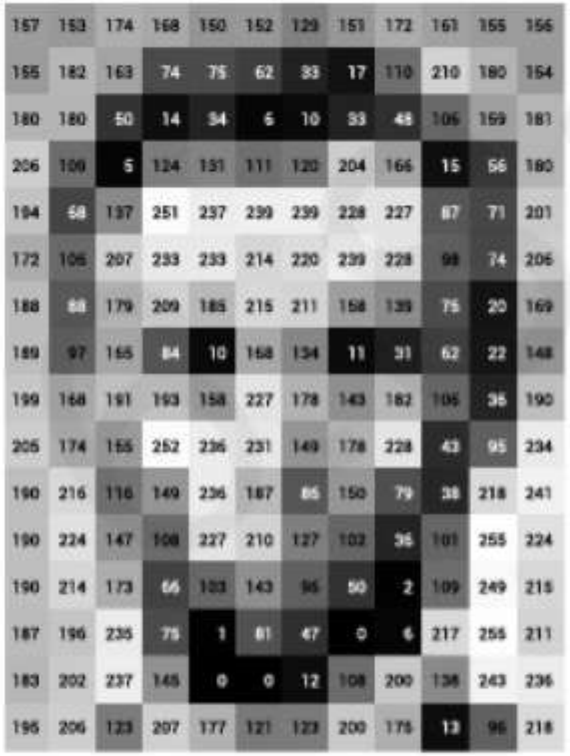
- Adaptive learning
- Batching
- Regularization



Deep Learning for Computer Vision

Images are Numbers



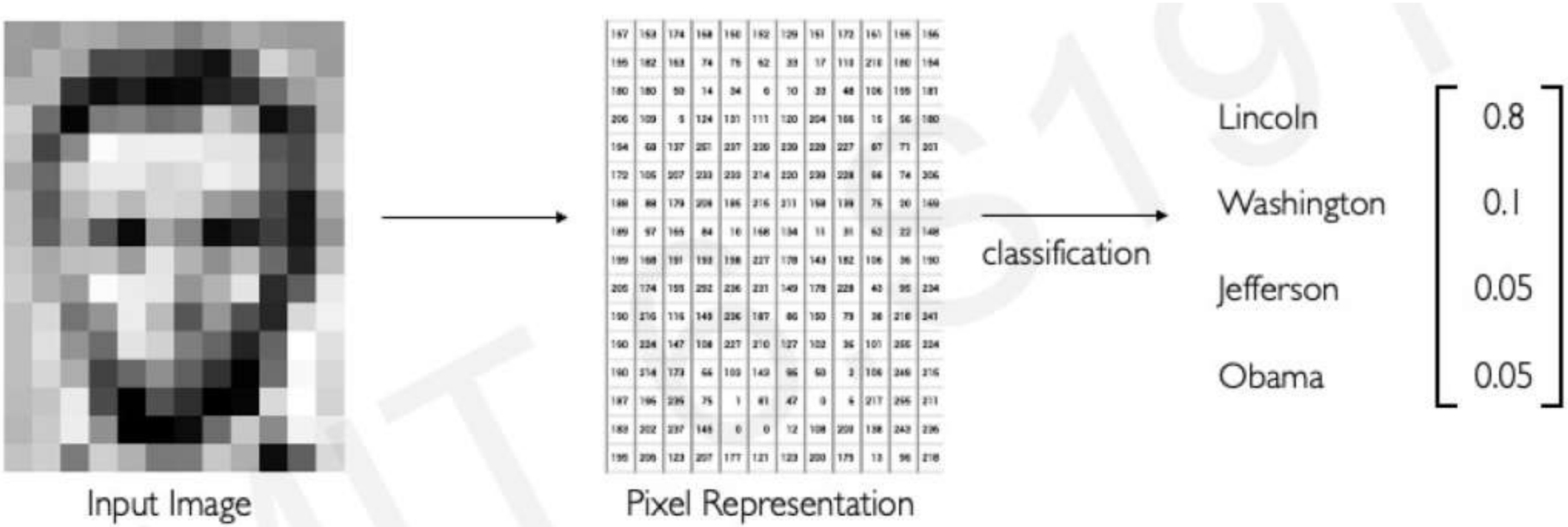


What the computer sees

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	105	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., 1080x1080x3 for an RGB image

Tasks in Computer Vision

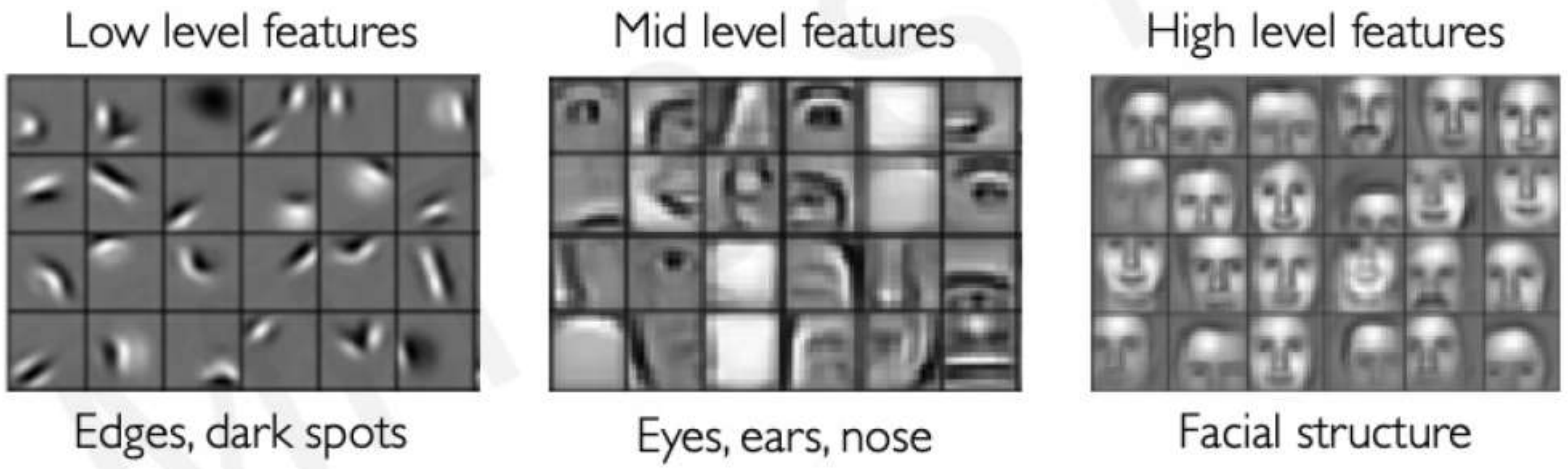


- **Regression**: output variable takes continuous value
- **Classification**: output variable takes class label. Can produce probability of belonging to a particular class

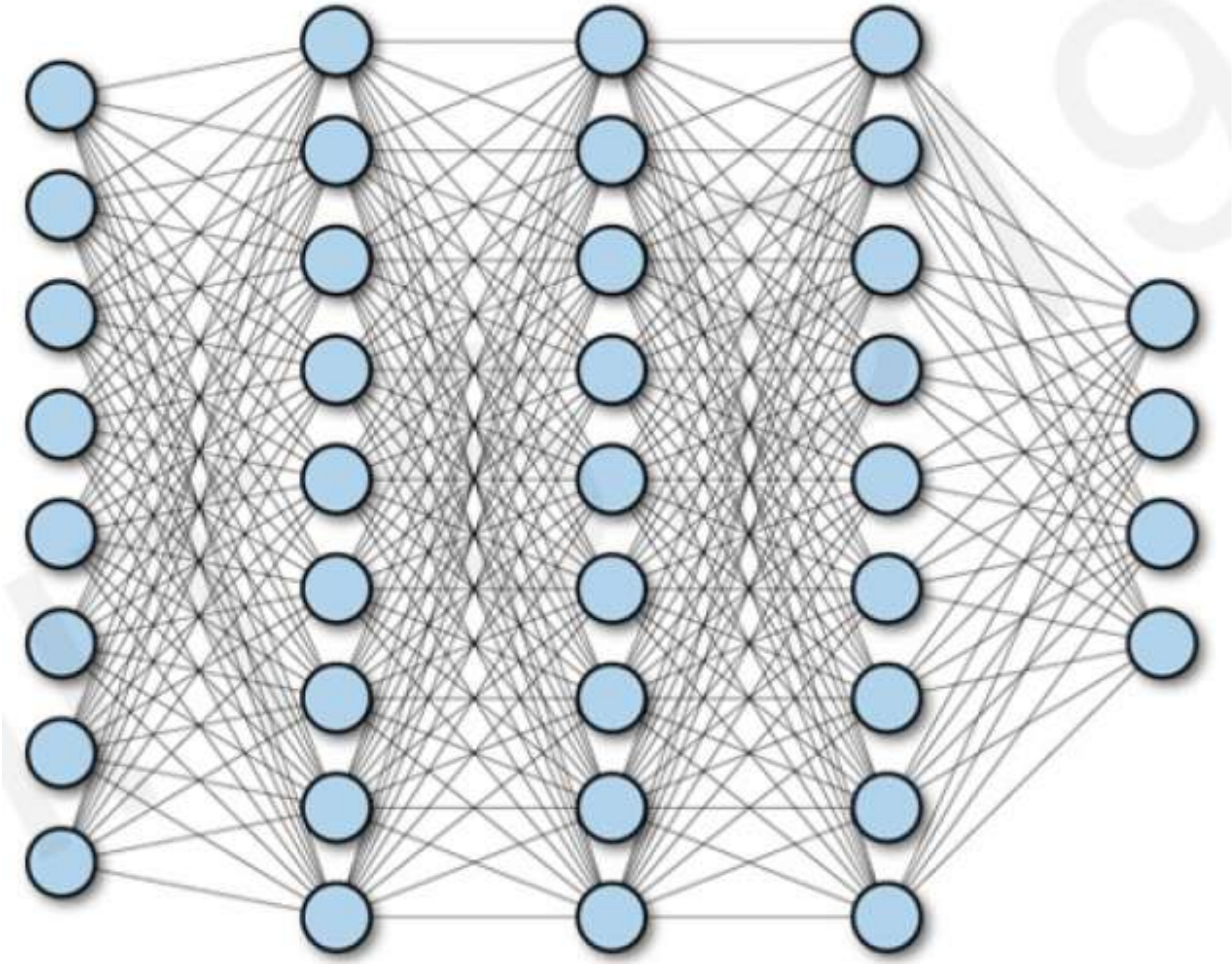
Manual Feature Extraction

Learning Feature Representations

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

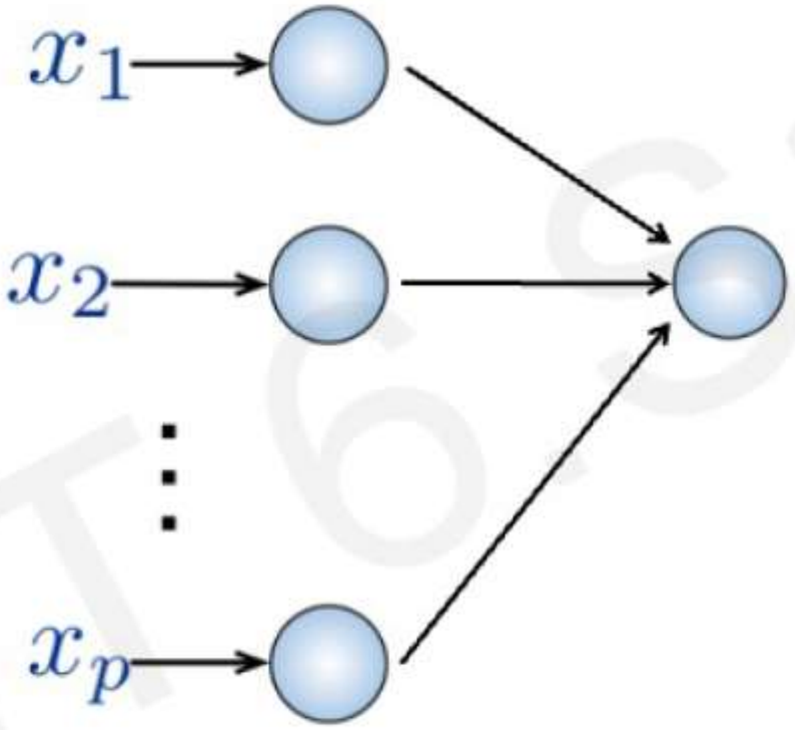


Fully Connected Neural Network



Fully Connected Neural Network

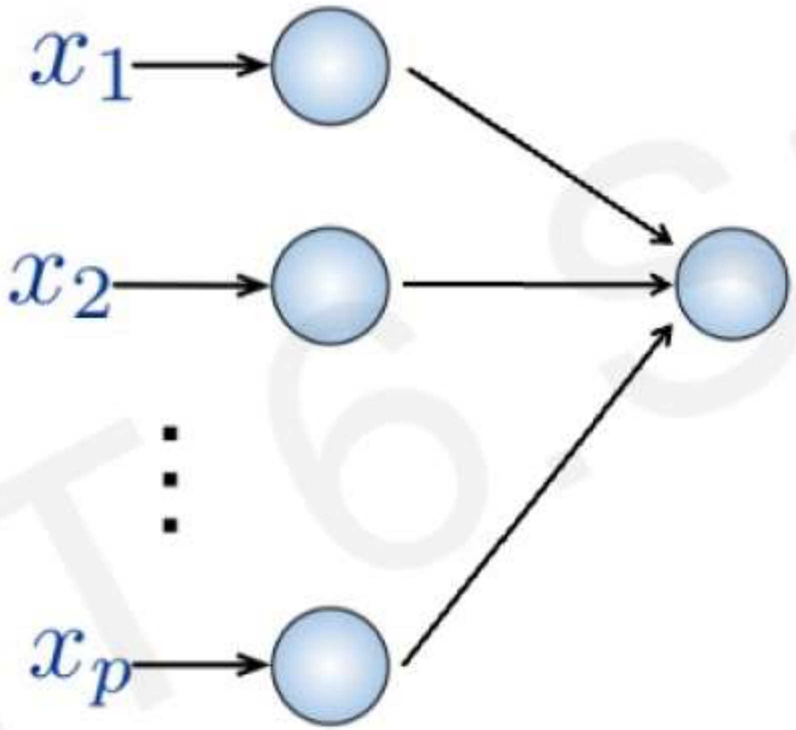
Input:
- 2D image
- Vector of pixel values



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

Fully Connected Neural Network

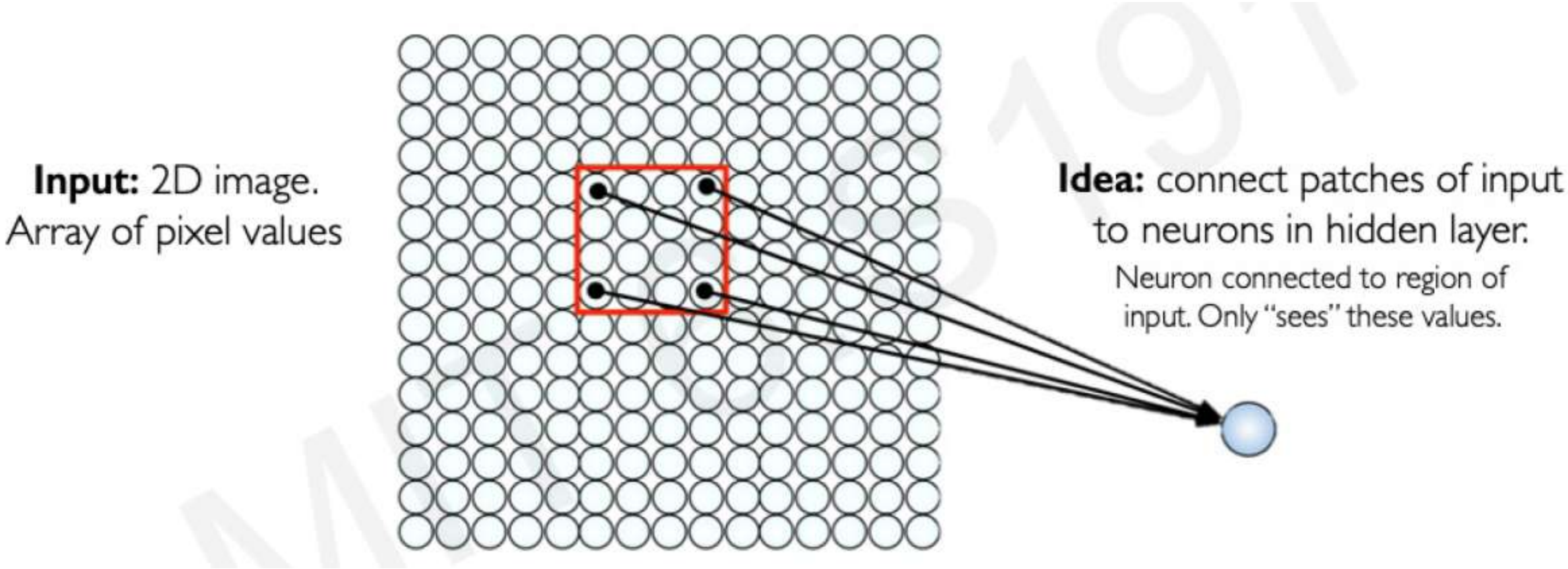
Input:
- 2D image
- Vector of pixel values



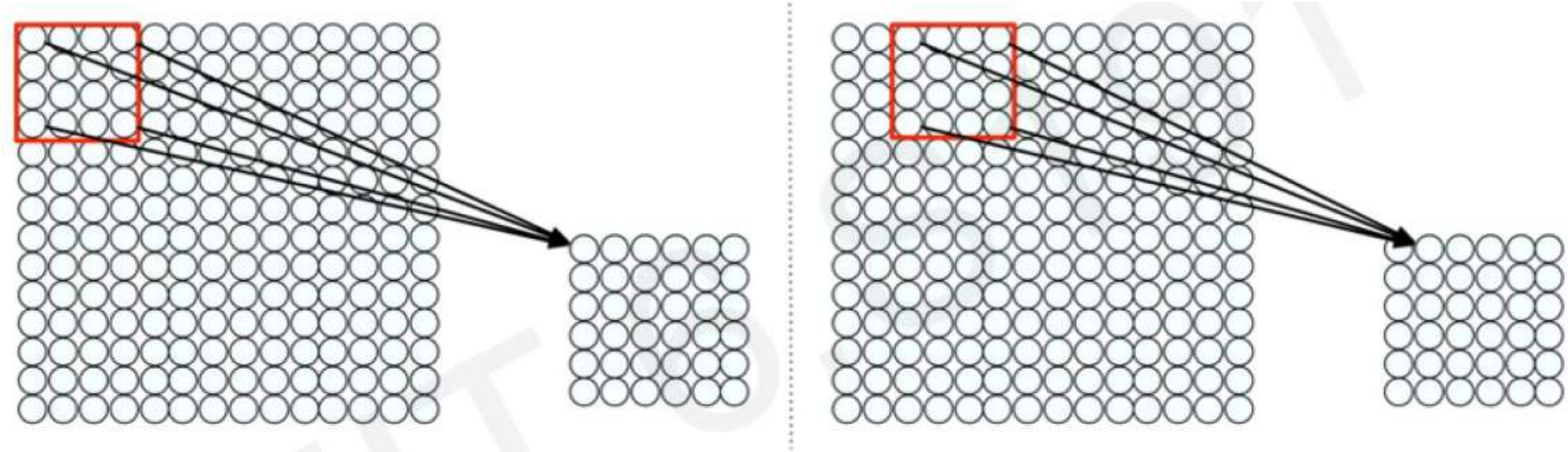
Fully Connected:
- Connect neuron in hidden layer to all neurons in input layer
- No spatial information!
- And many, many parameters!

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

Using Spatial Structure



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?*

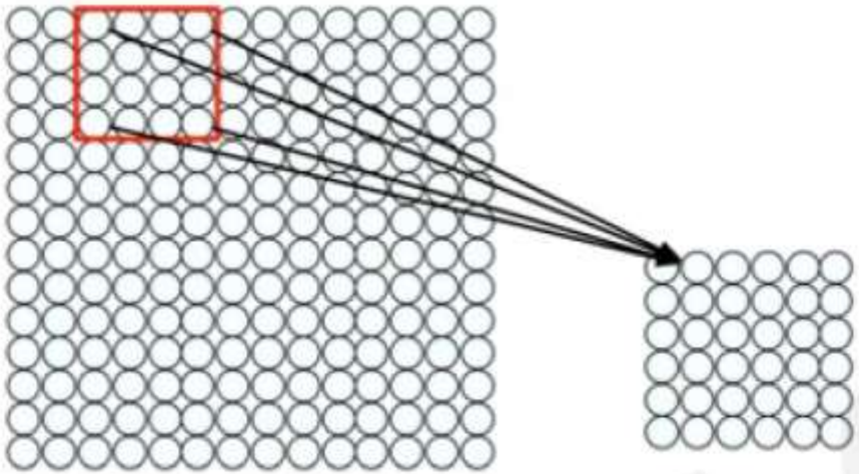
Applying Filters to Extract Features

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?

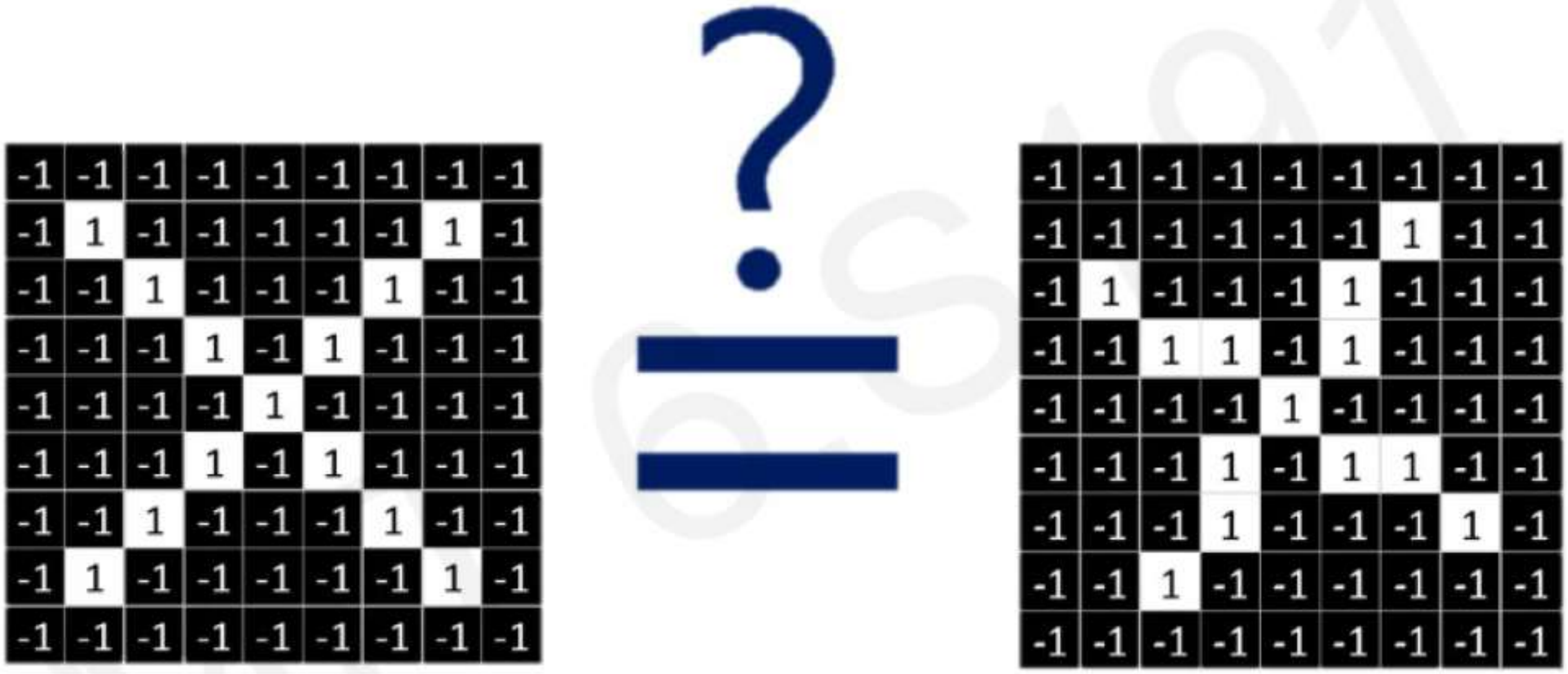
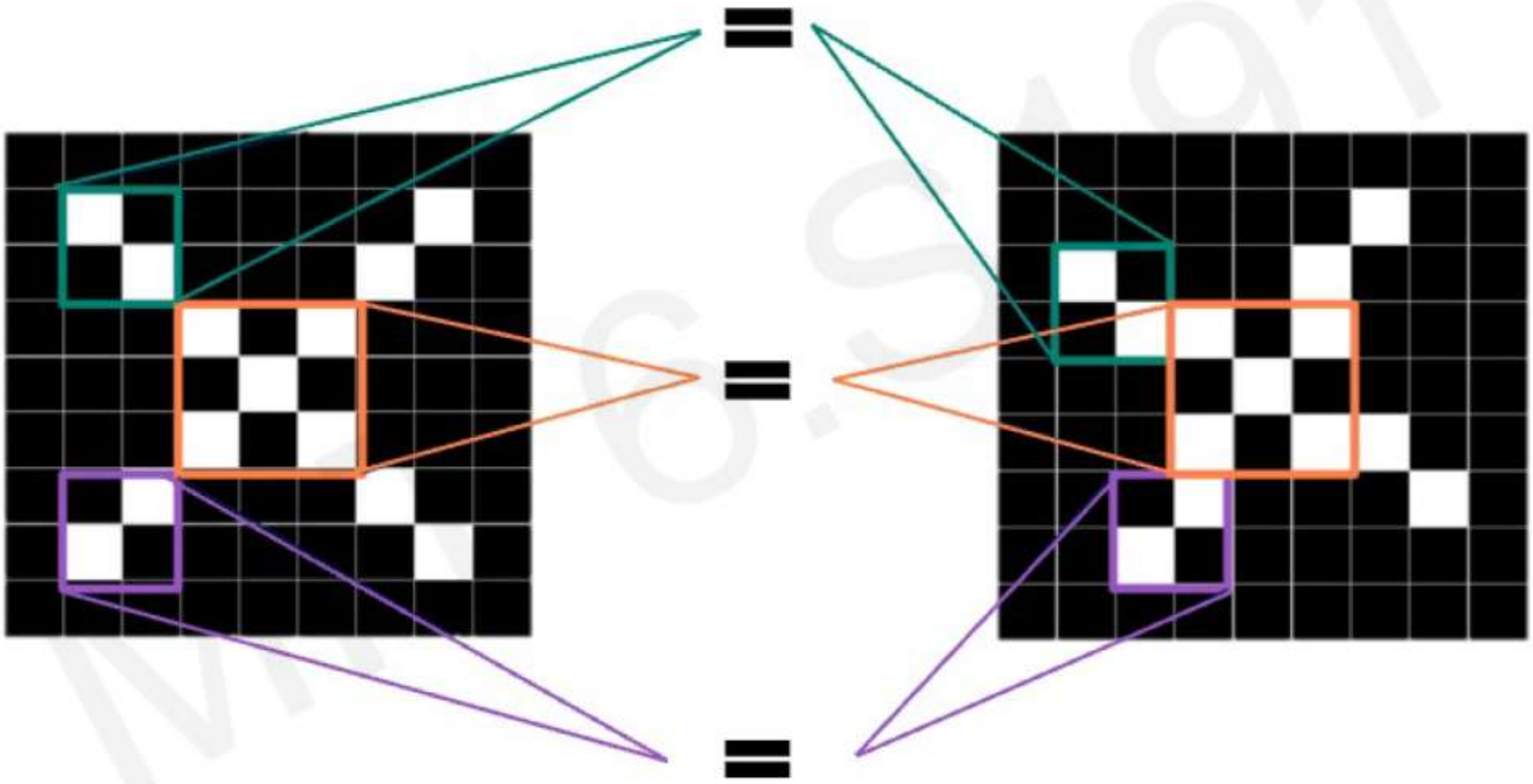
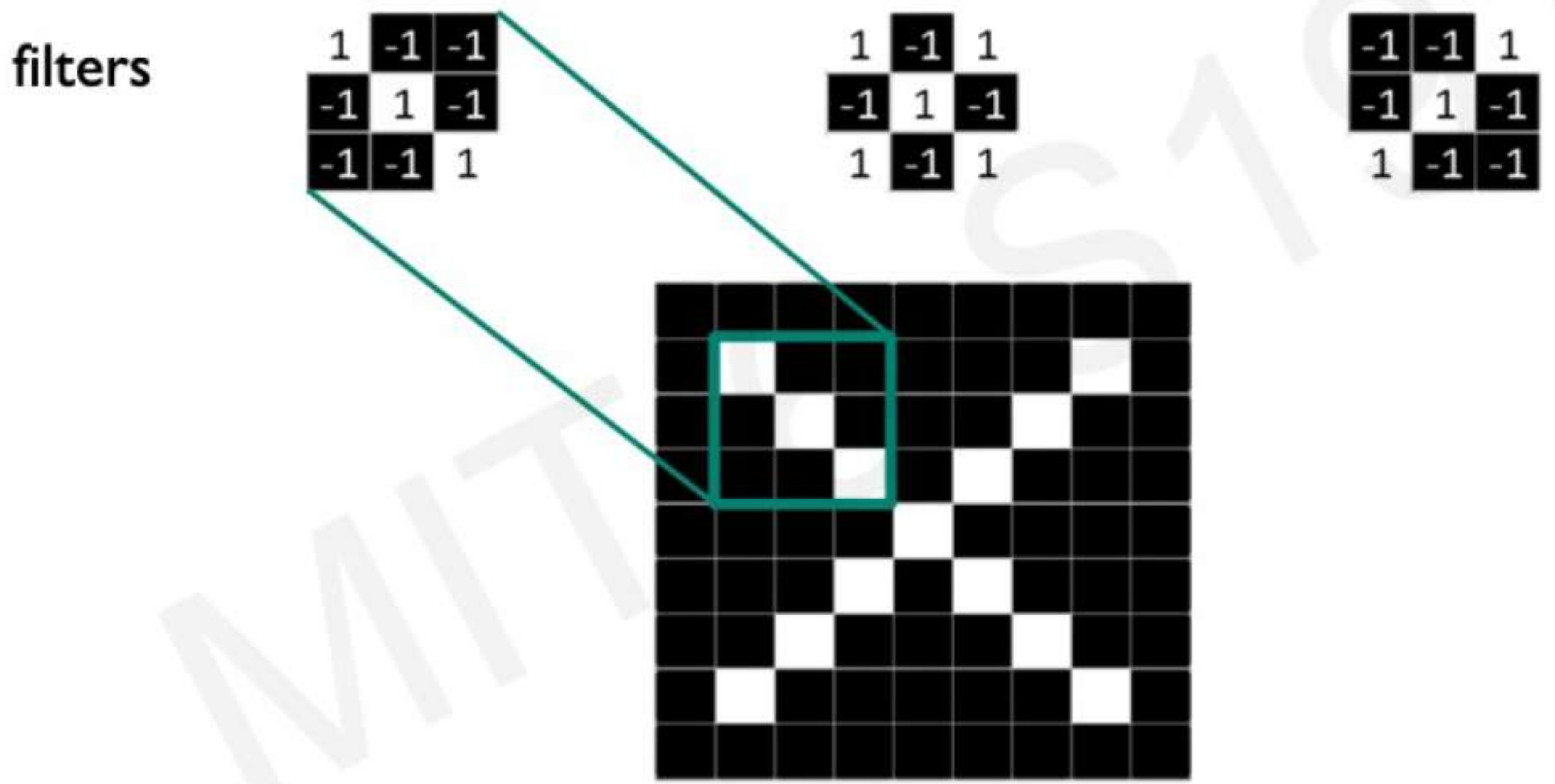


Image is represented as matrix of pixel values… and computers are literal!
We want to be able to classify an X as an X even if it's shifted, shrunk, rotated, deformed.

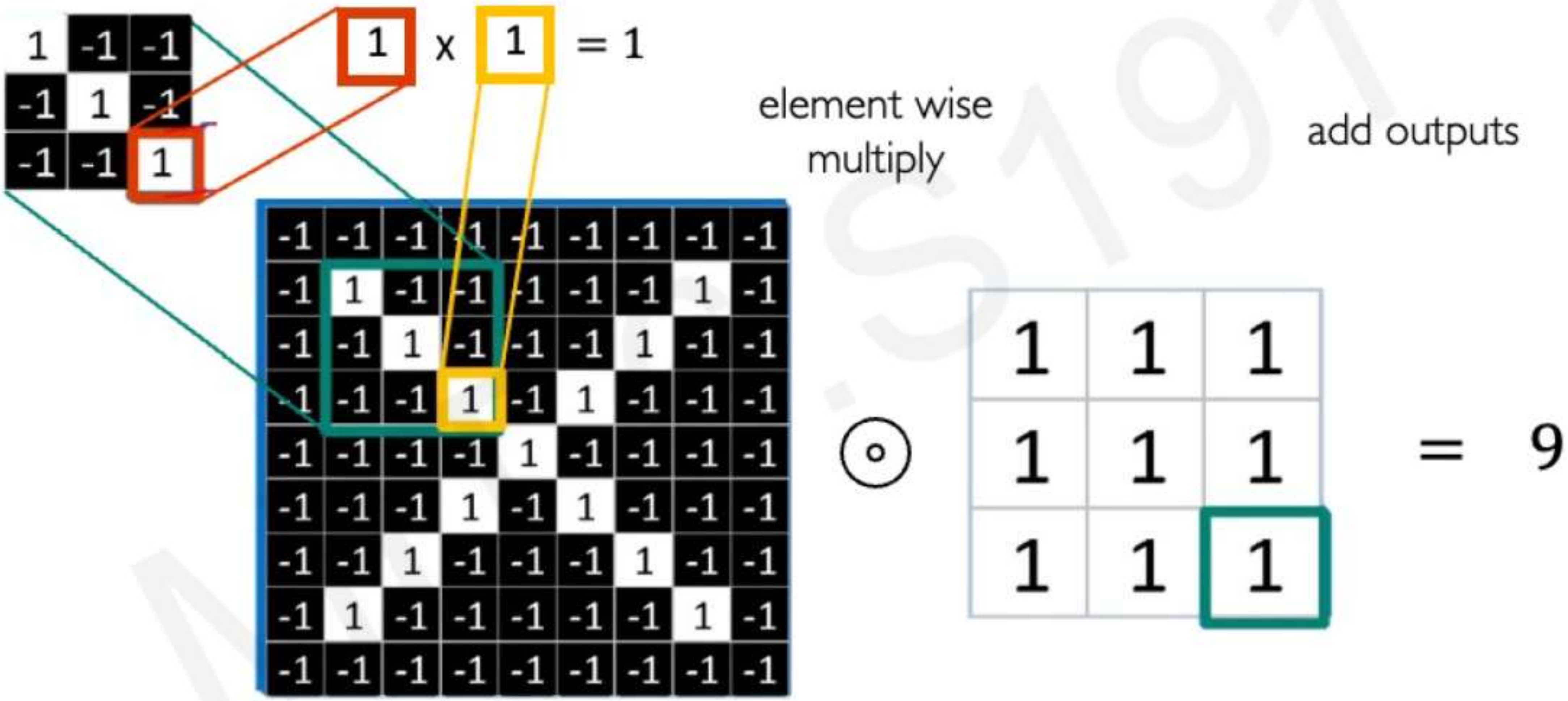
Features of X



Filters to Detect Features

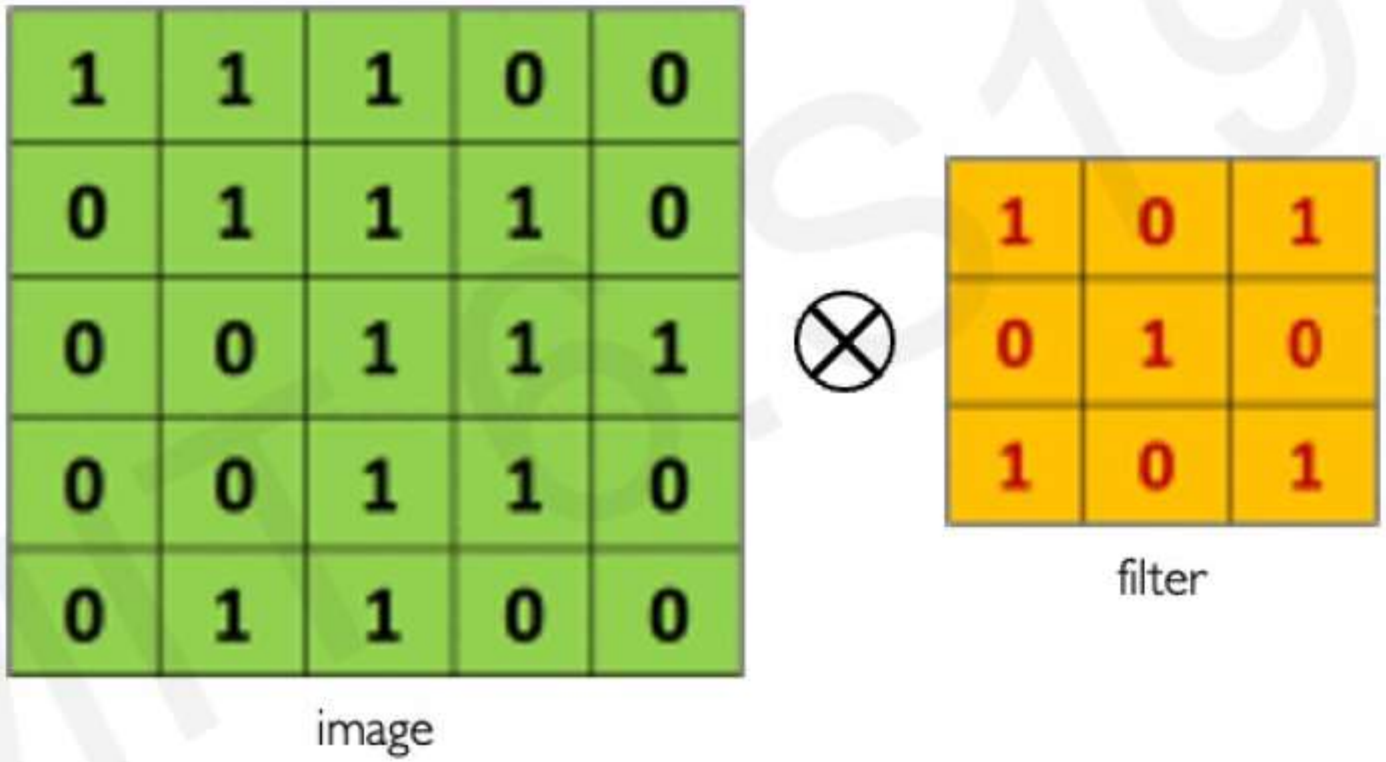


The Convolution Operation



The Convolution Operation

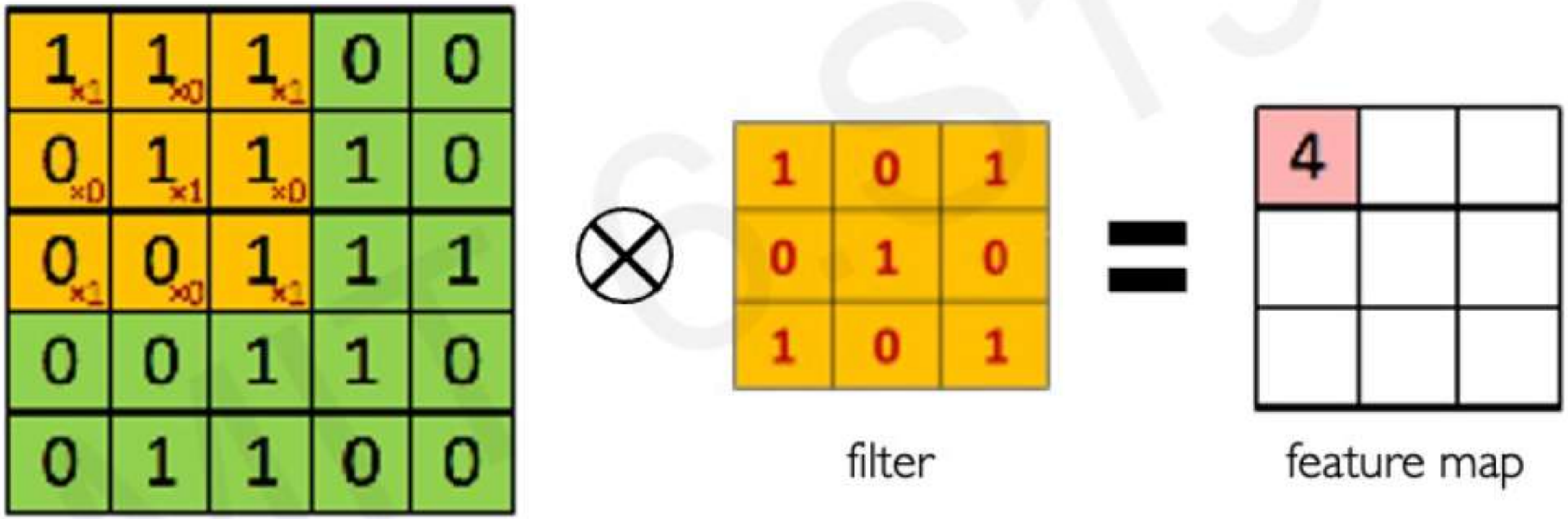
Suppose we want to compute the convolution of a 5x5 image and a 3x3 filter:



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

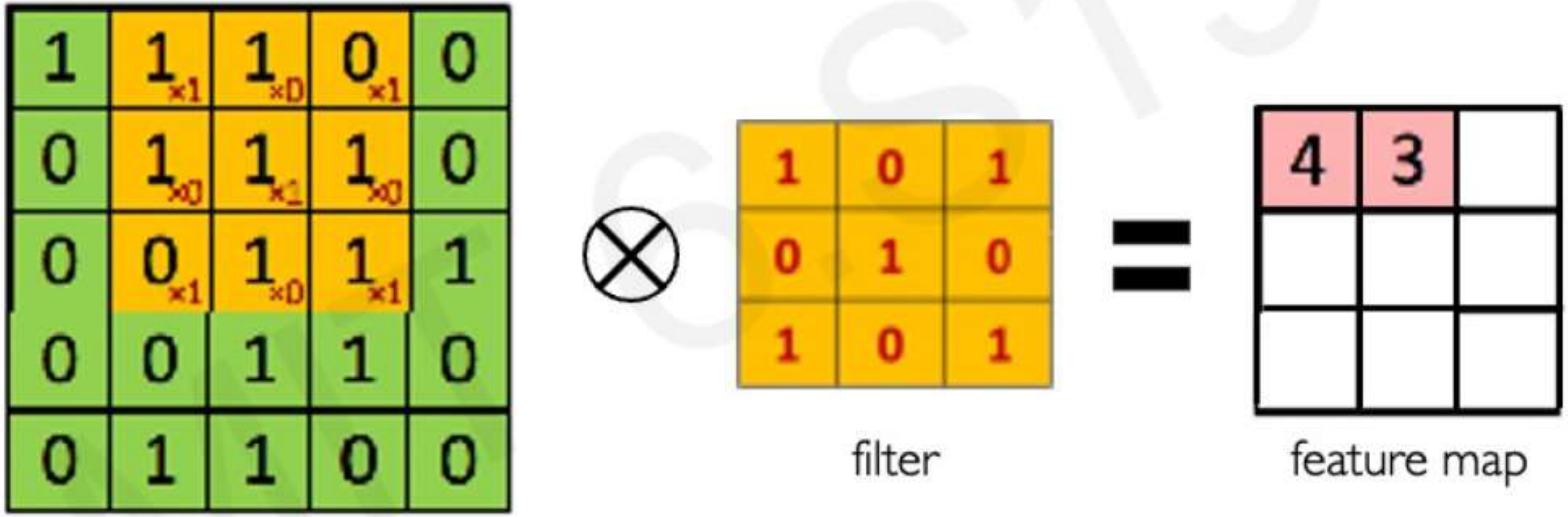
The Convolution Operation

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



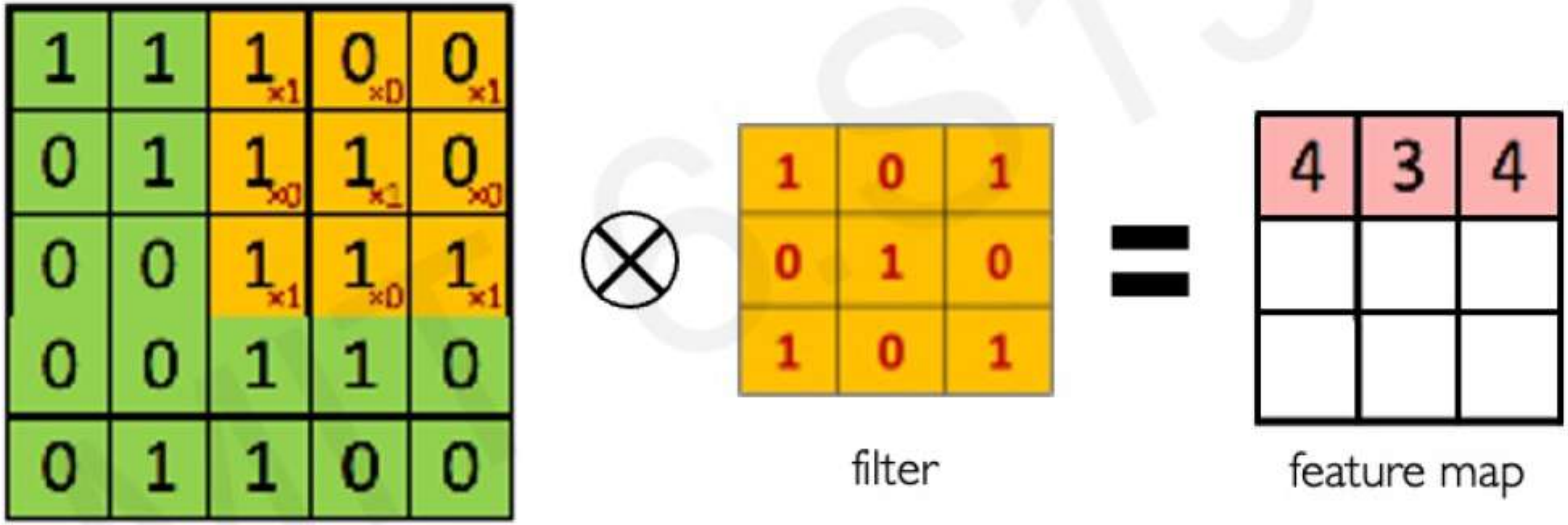
The Convolution Operation

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



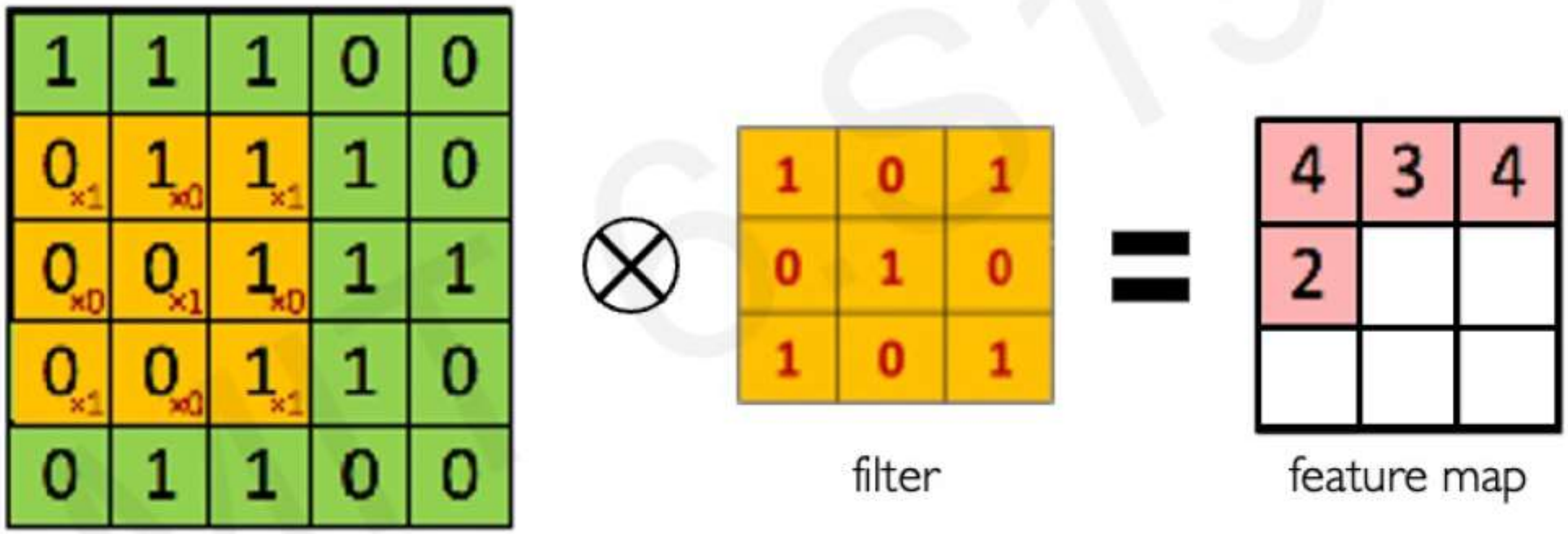
The Convolution Operation

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



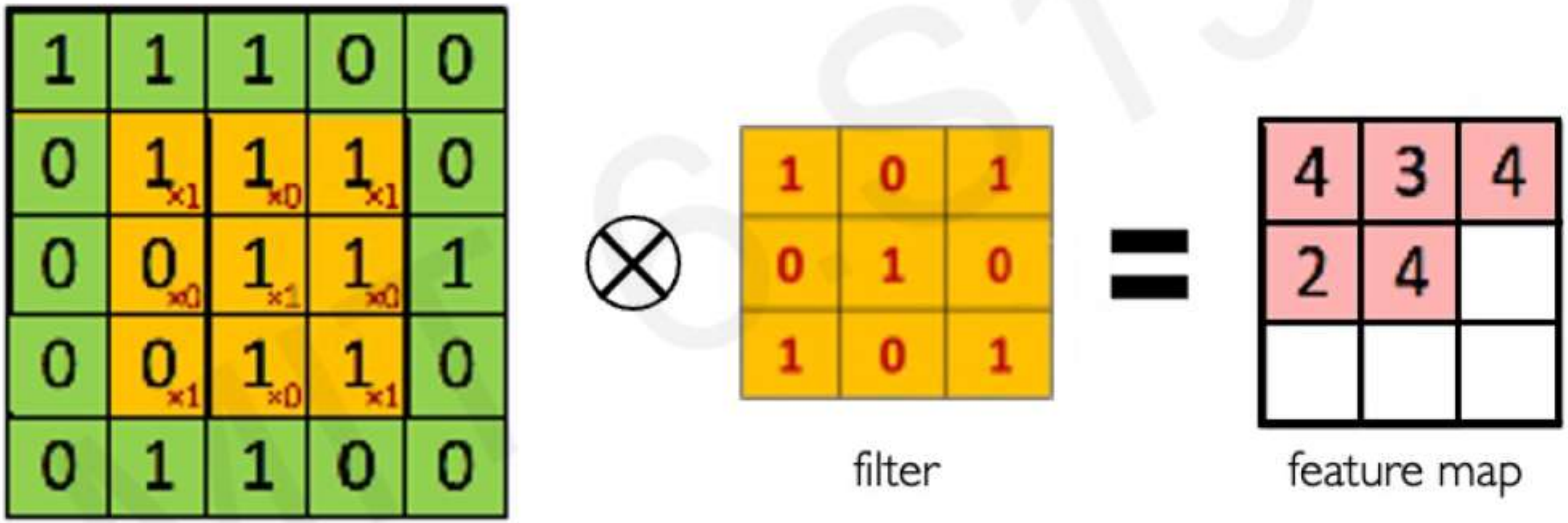
The Convolution Operation

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



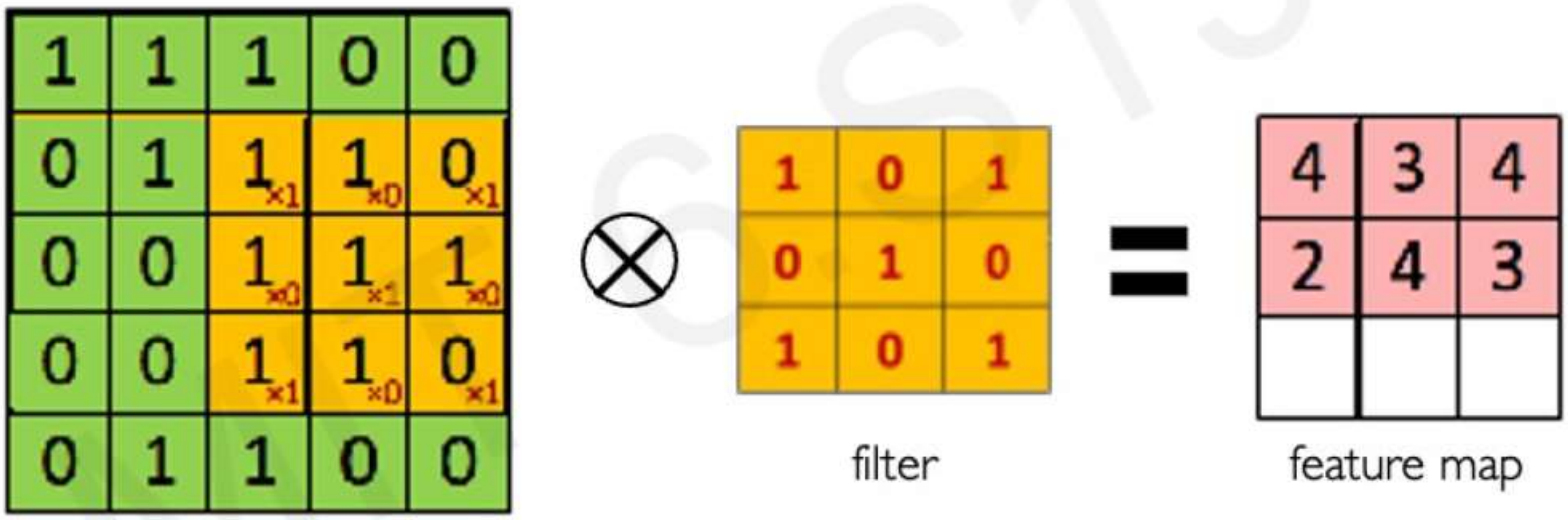
The Convolution Operation

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



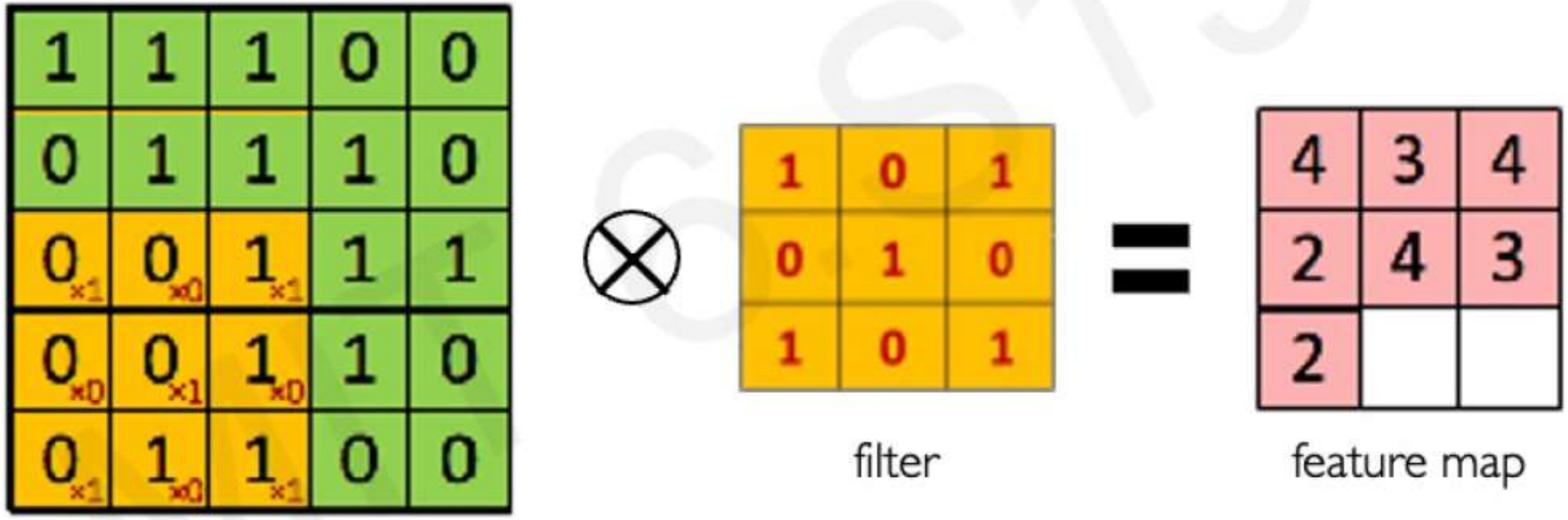
The Convolution Operation

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



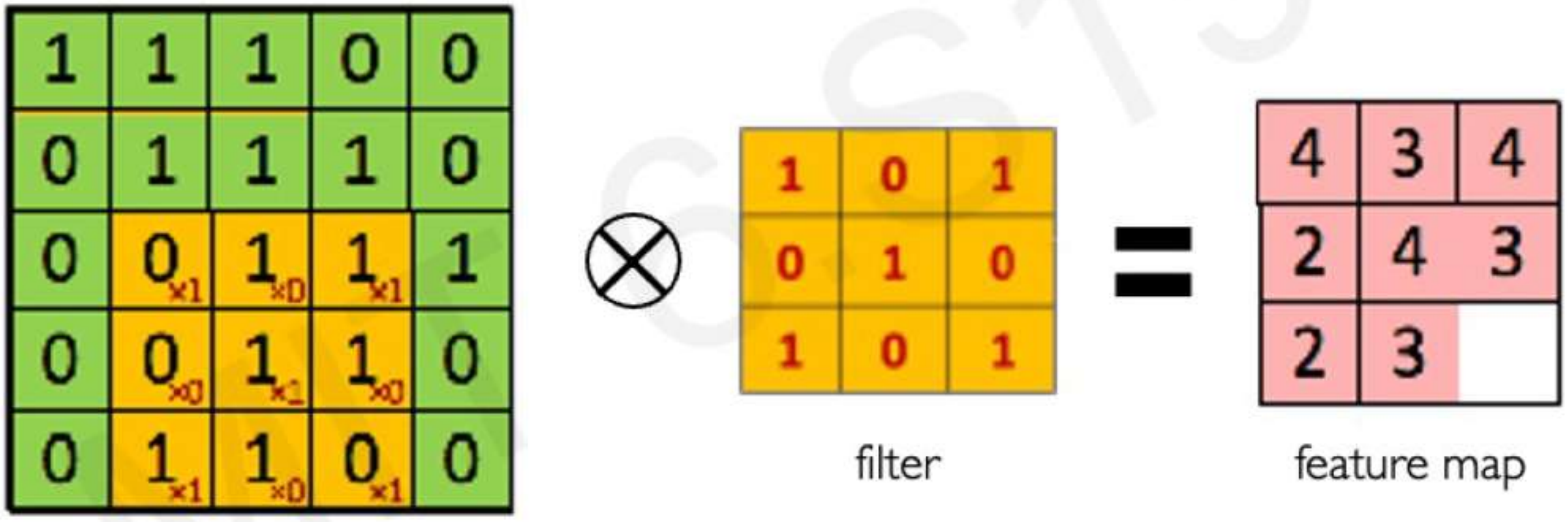
The Convolution Operation

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



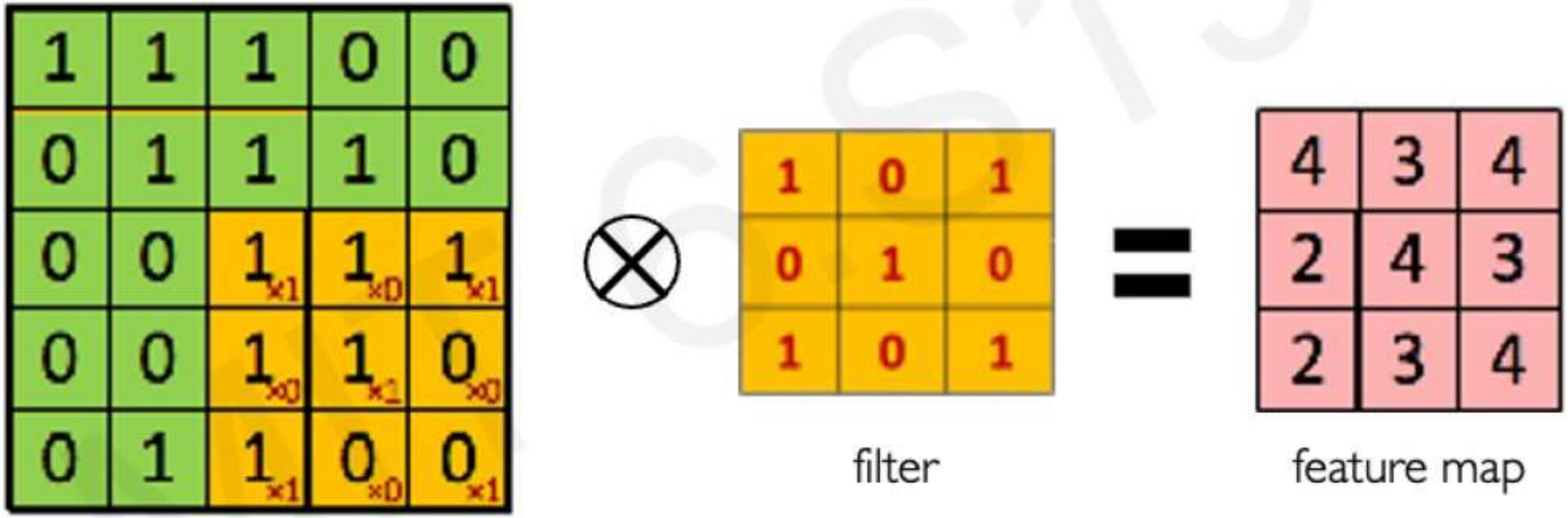
The Convolution Operation

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

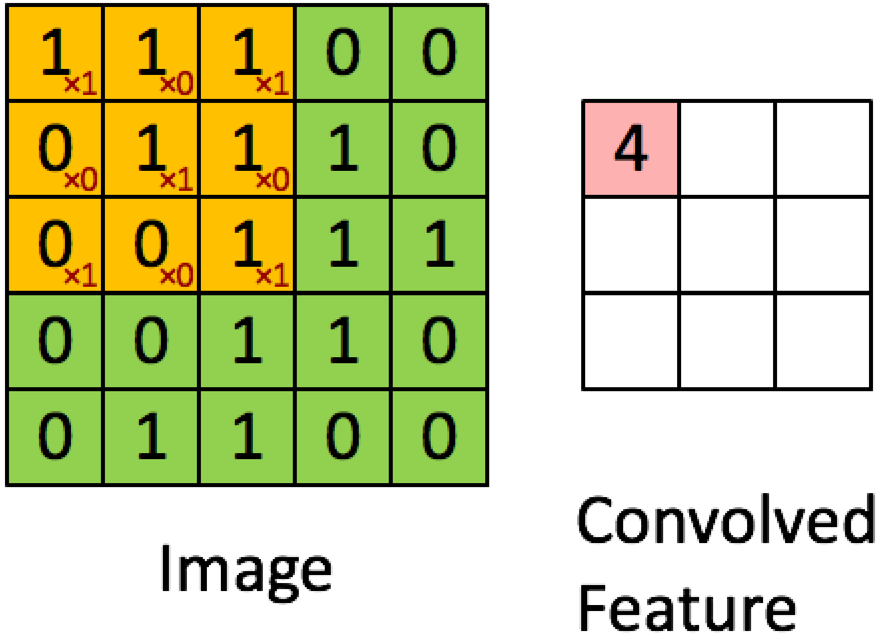


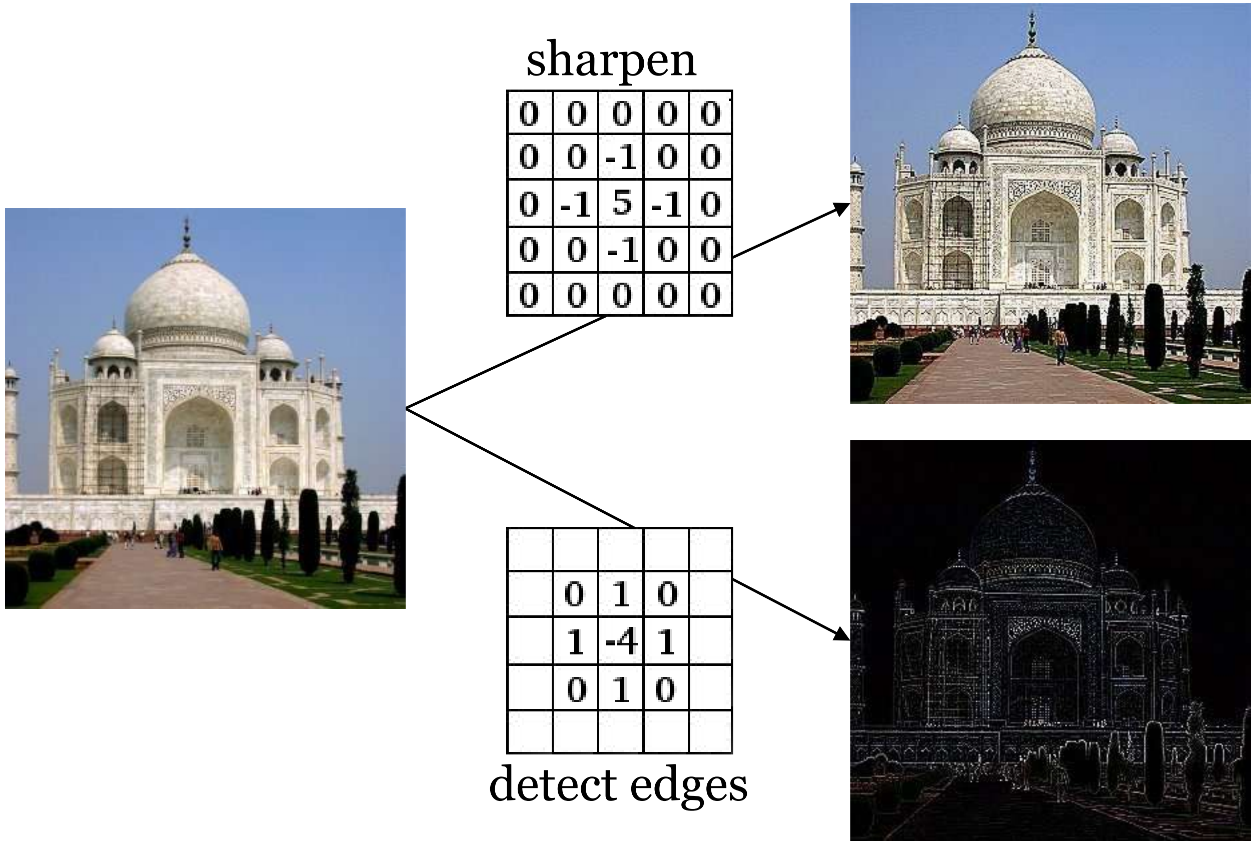
The Convolution Operation

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



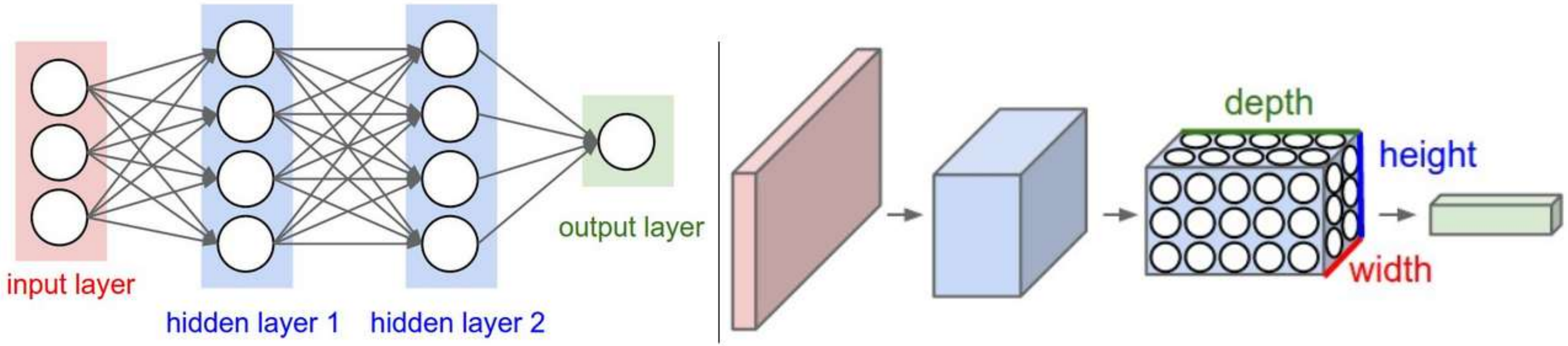
Convolutional Neural Networks





http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

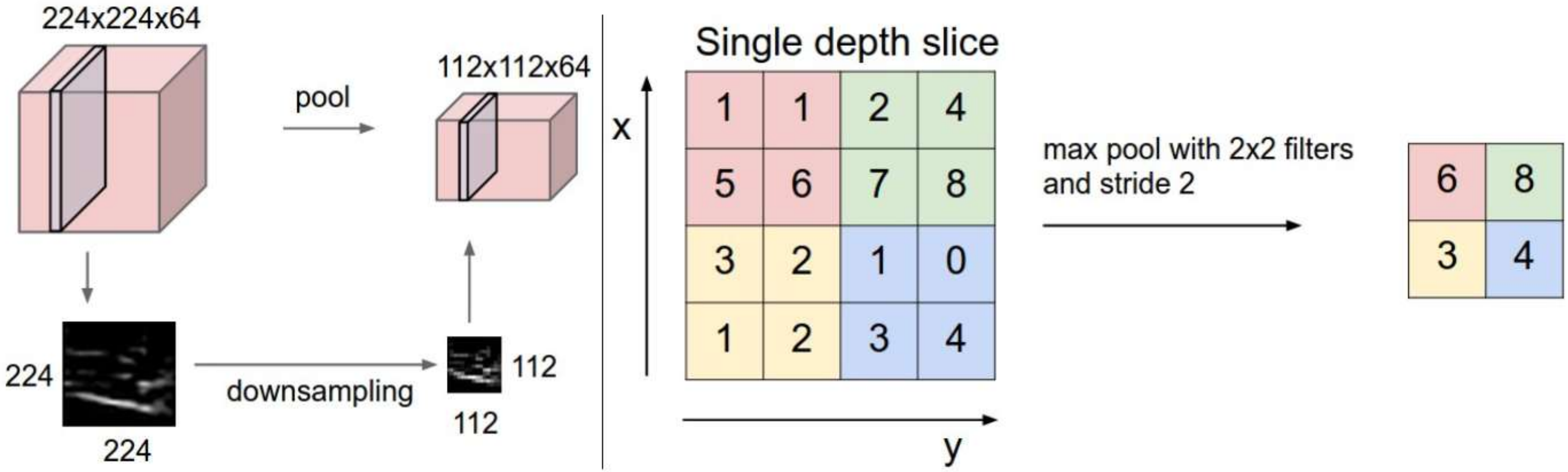
Convolutional Neural Networks



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).

http://cs231n.github.io/convolutional-networks/

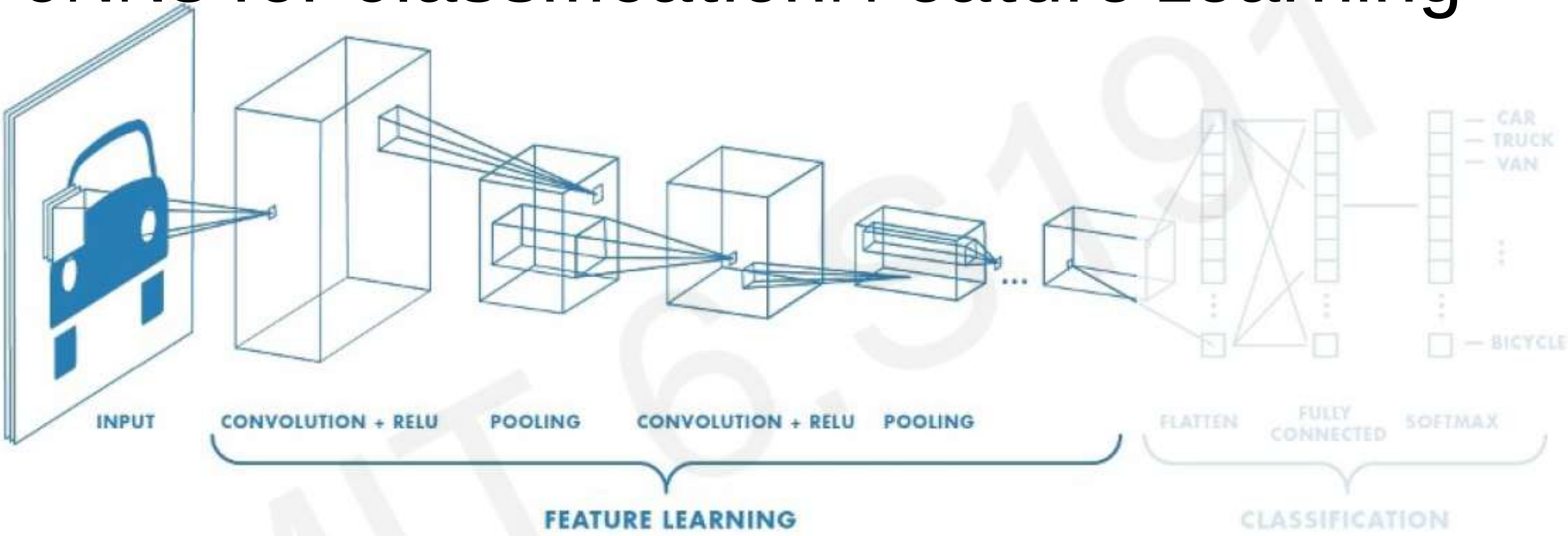
Convolutional Neural Networks



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. 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 2x2 square).

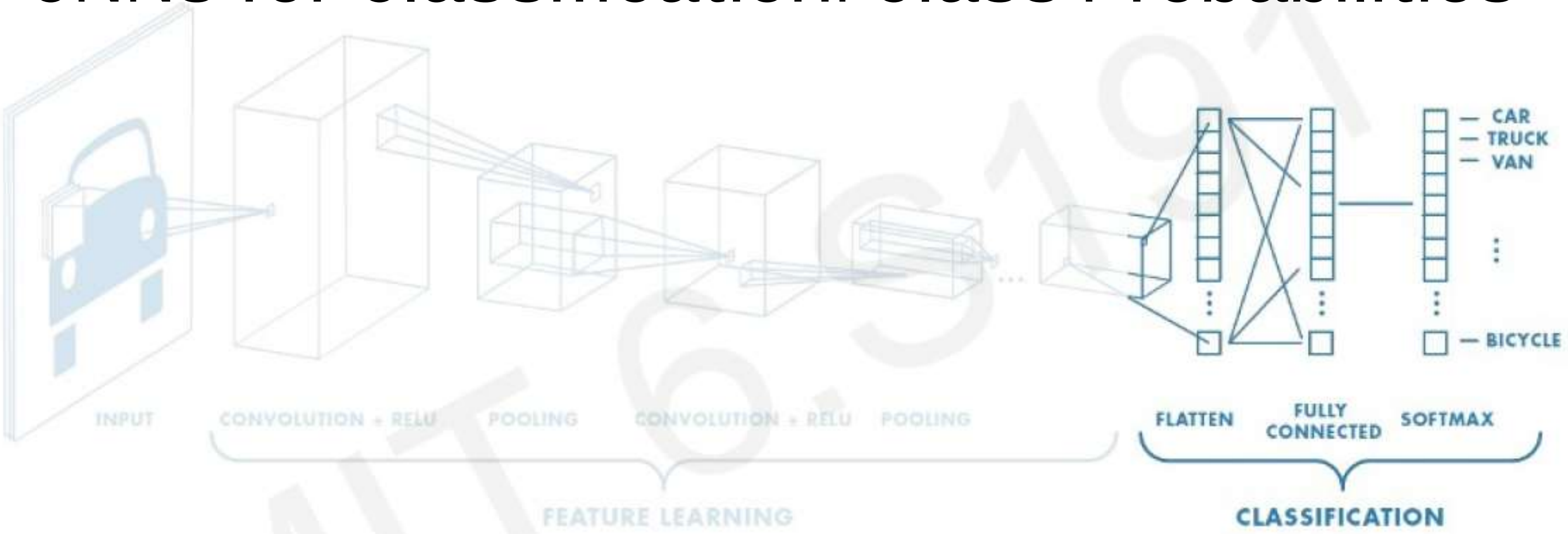
http://cs231n.github.io/convolutional-networks/

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**

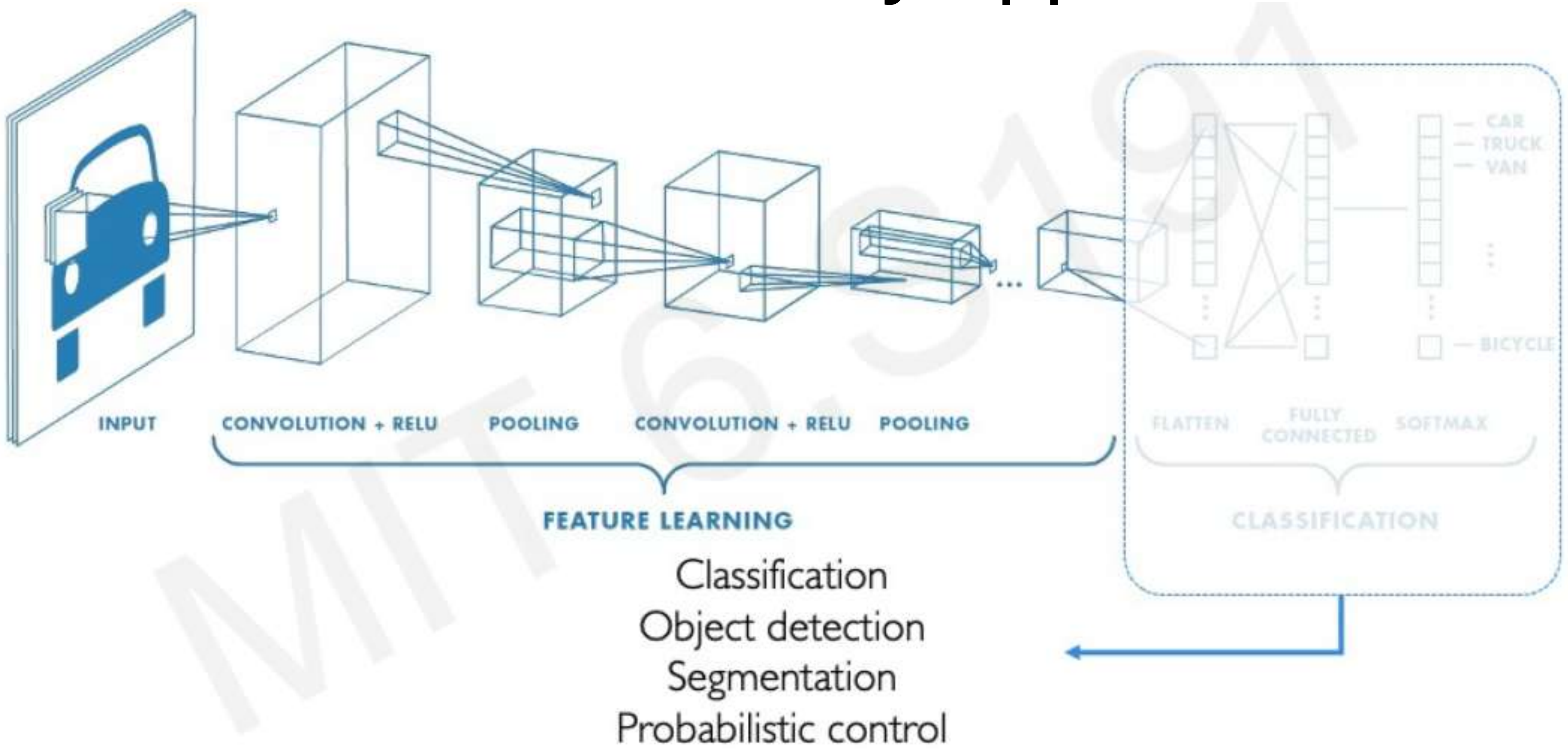
CNNs for Classification: Class Probabilities



- 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

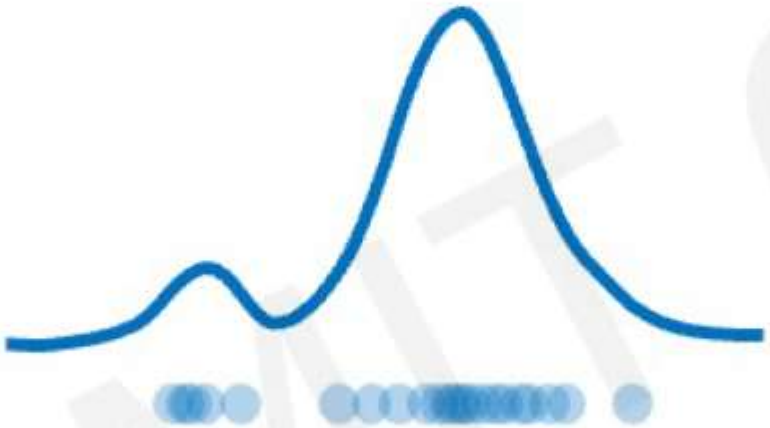


Deep Generative Models

Generative Modeling

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

Density Estimation



samples

Sample Generation

Input samples

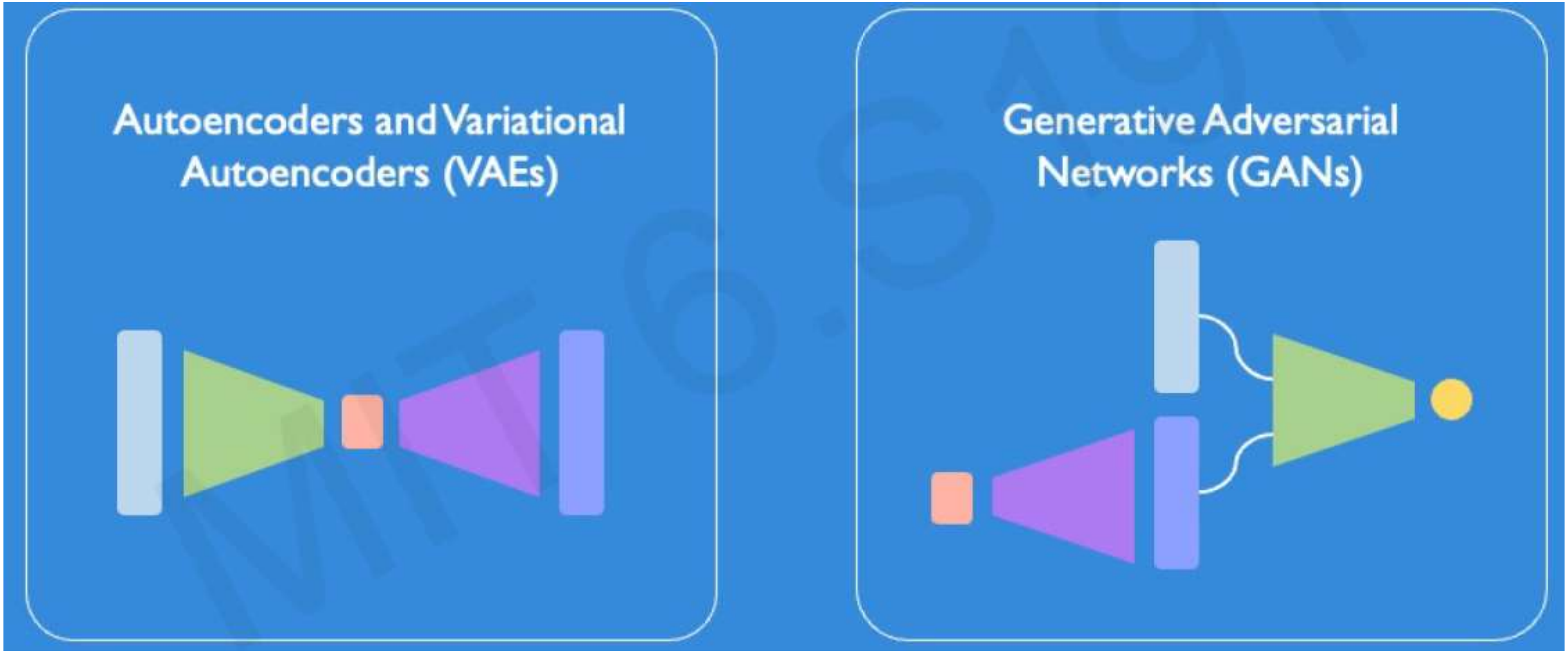
Training data $\sim P_{data}(x)$

Generated samples

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

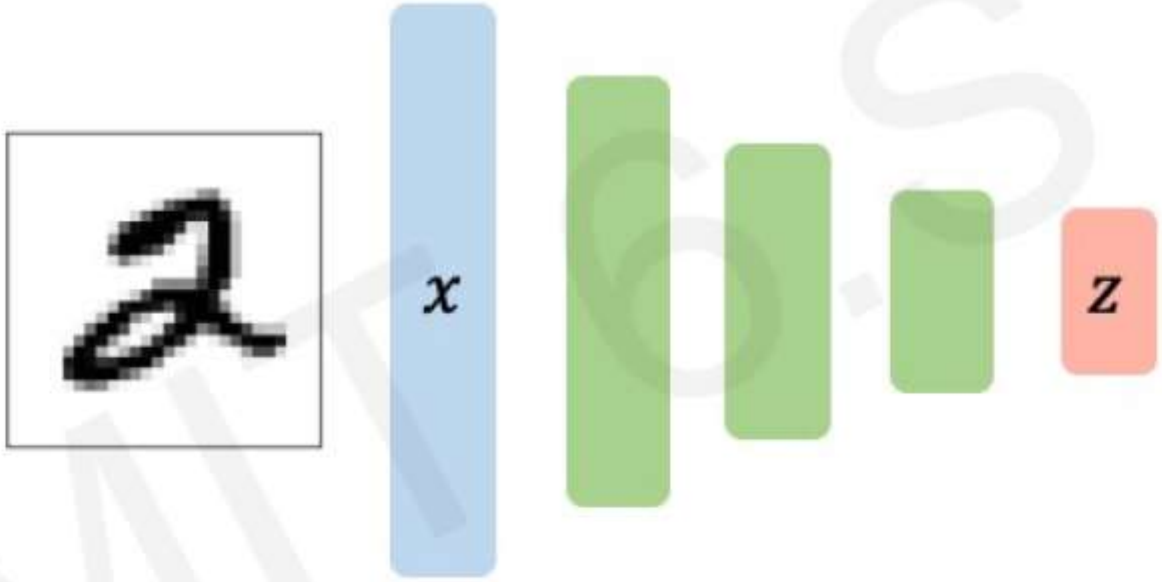
How can we learn $P_{model}(x)$ similar to $P_{data}(x)$?

Latent Variable Models



Autoencoders: Background

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



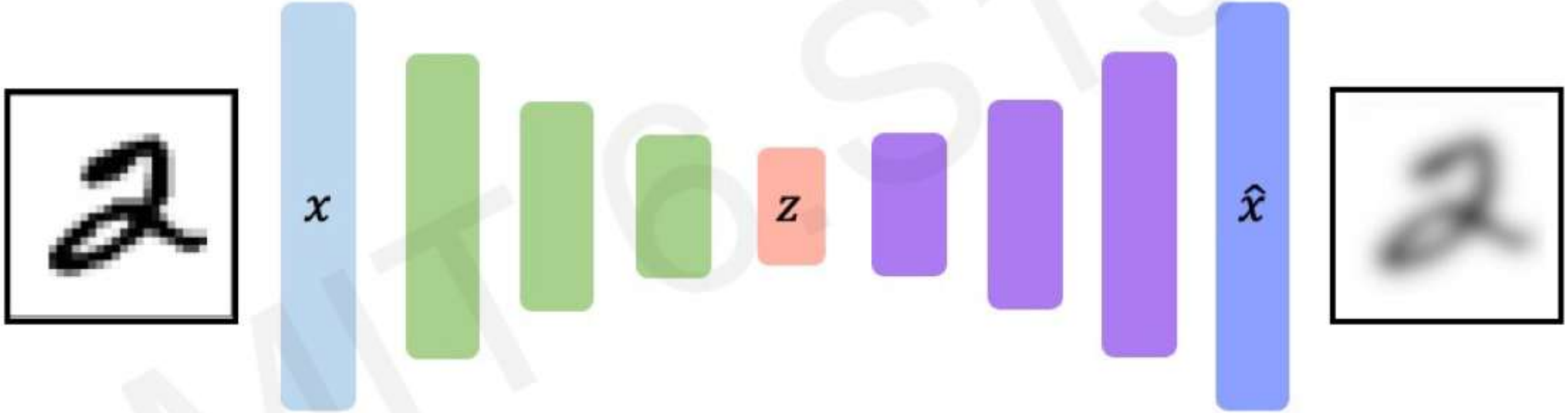
Why do we care about a low-dimensional ***z***?

"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**

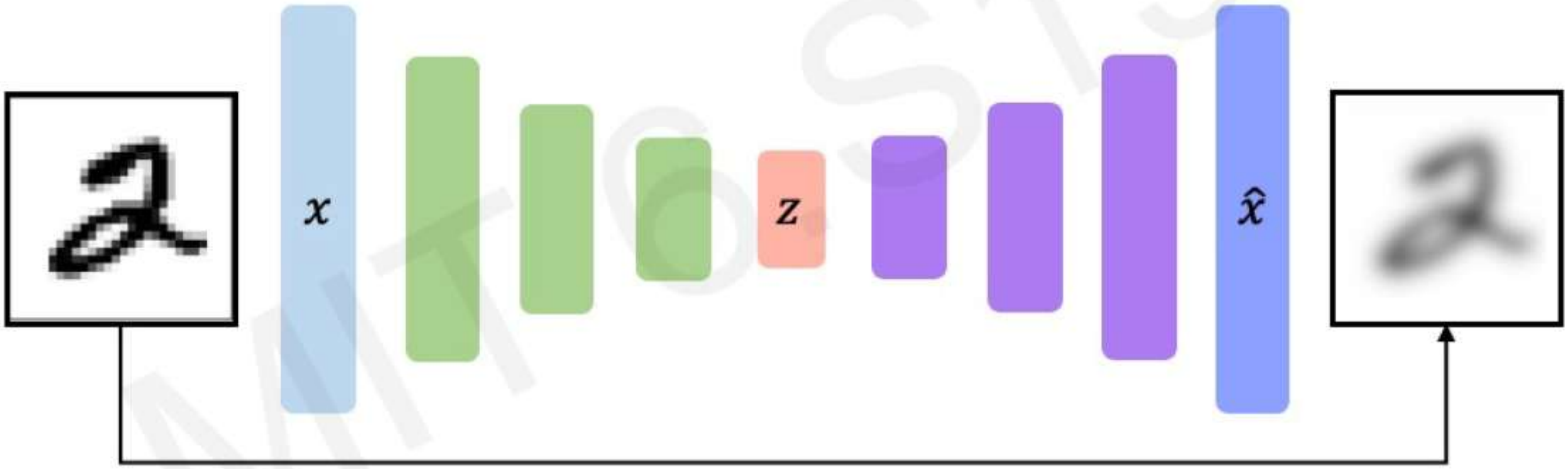


"Decoder" learns mapping back from latent space, z, to a reconstructed observation, $\hat{x}$

Autoencoders: Background

How can we learn this latent space?

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



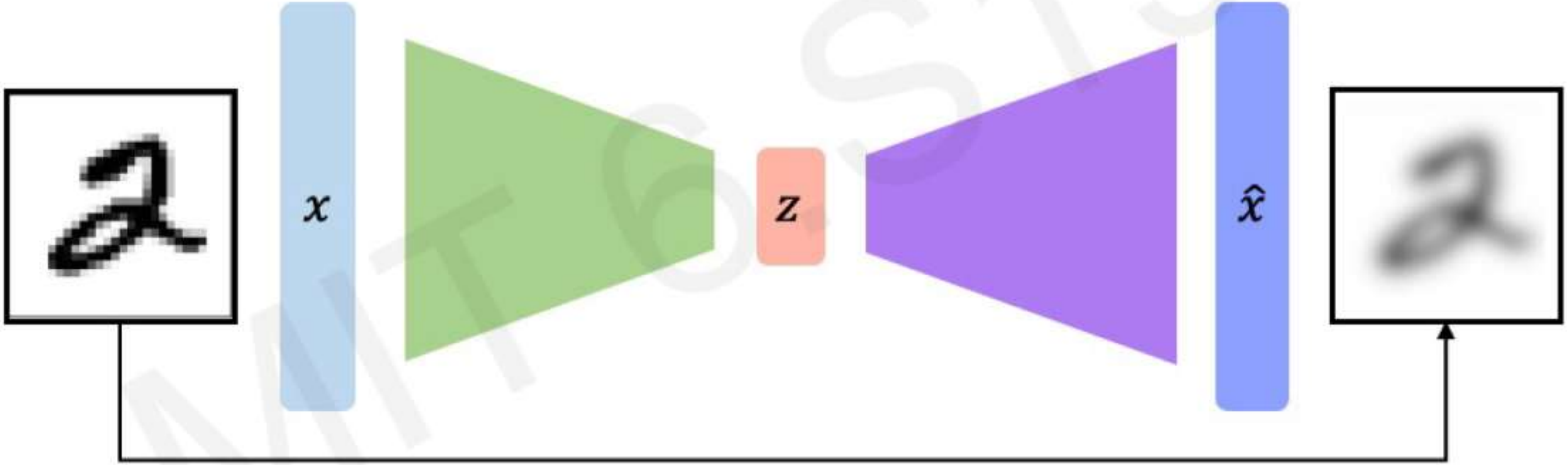
$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

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**

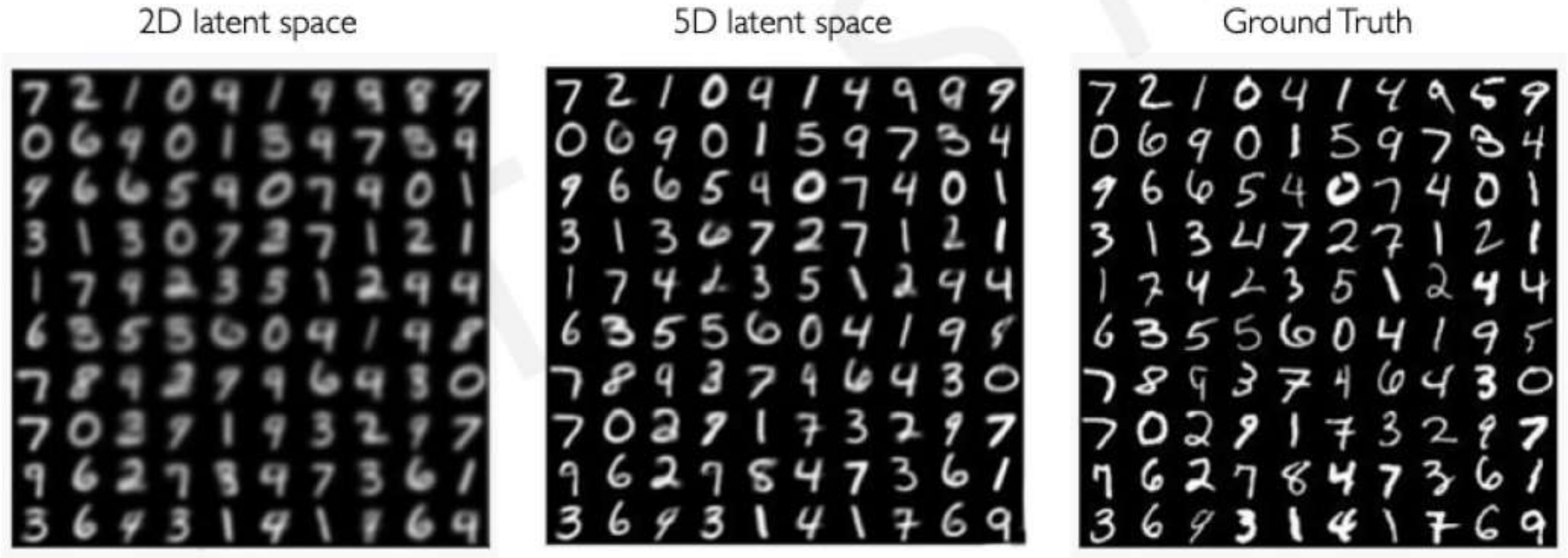


$$\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2$$

Loss function doesn't use any labels!!

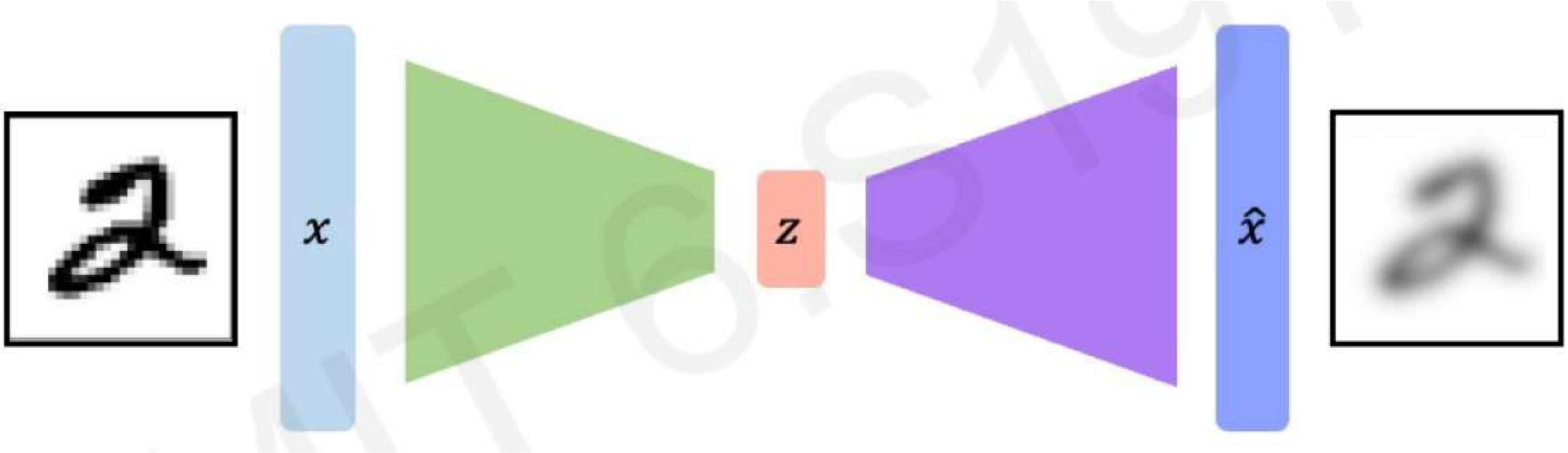
Dimensionality of Latent Space -> Reconstruction Quality

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

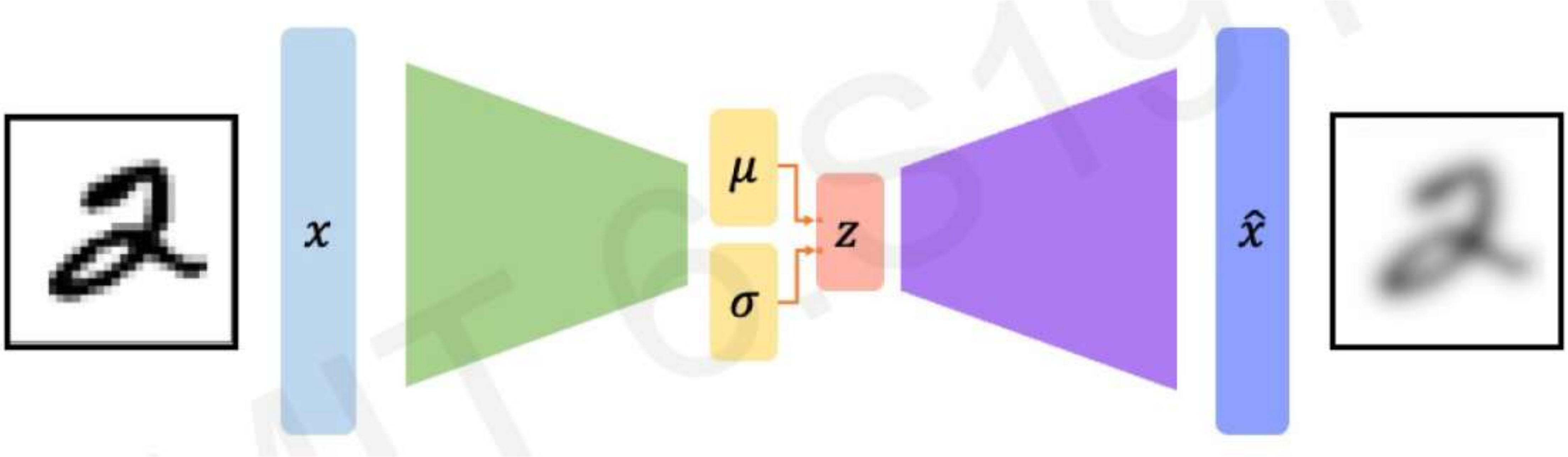


Variational Autoencoder (VAE)

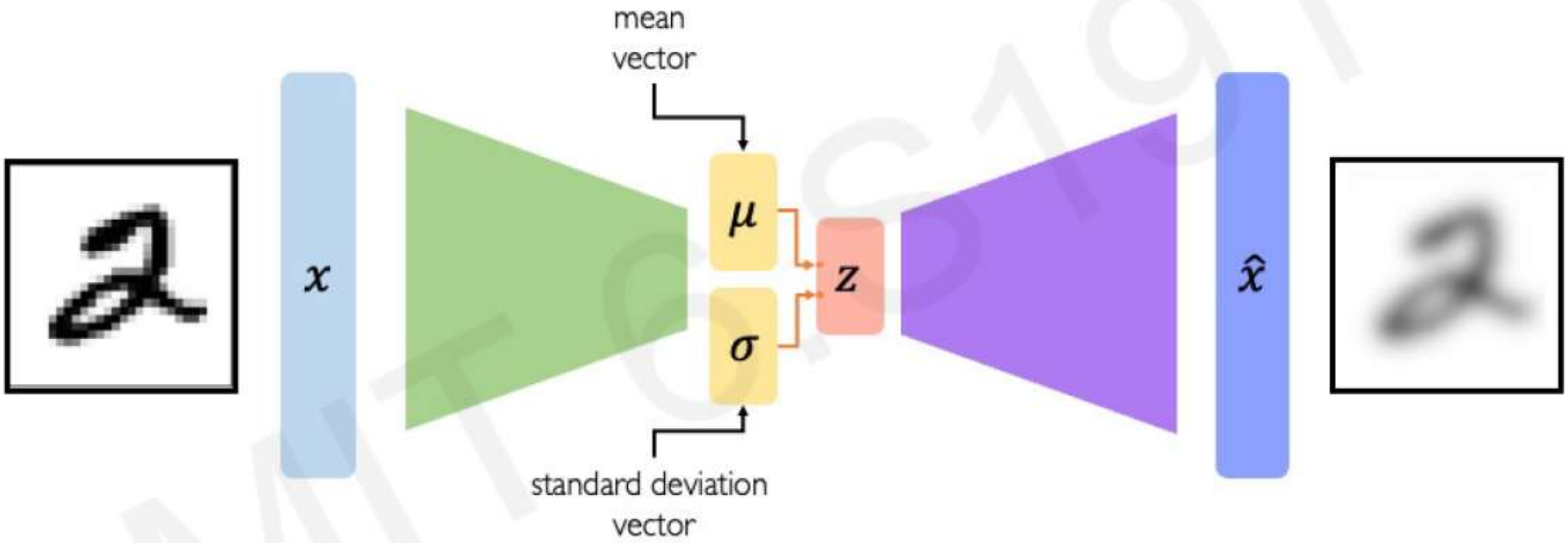
Traditional Autoencoder



VAEs: Key Difference with Traditional Autoencoder



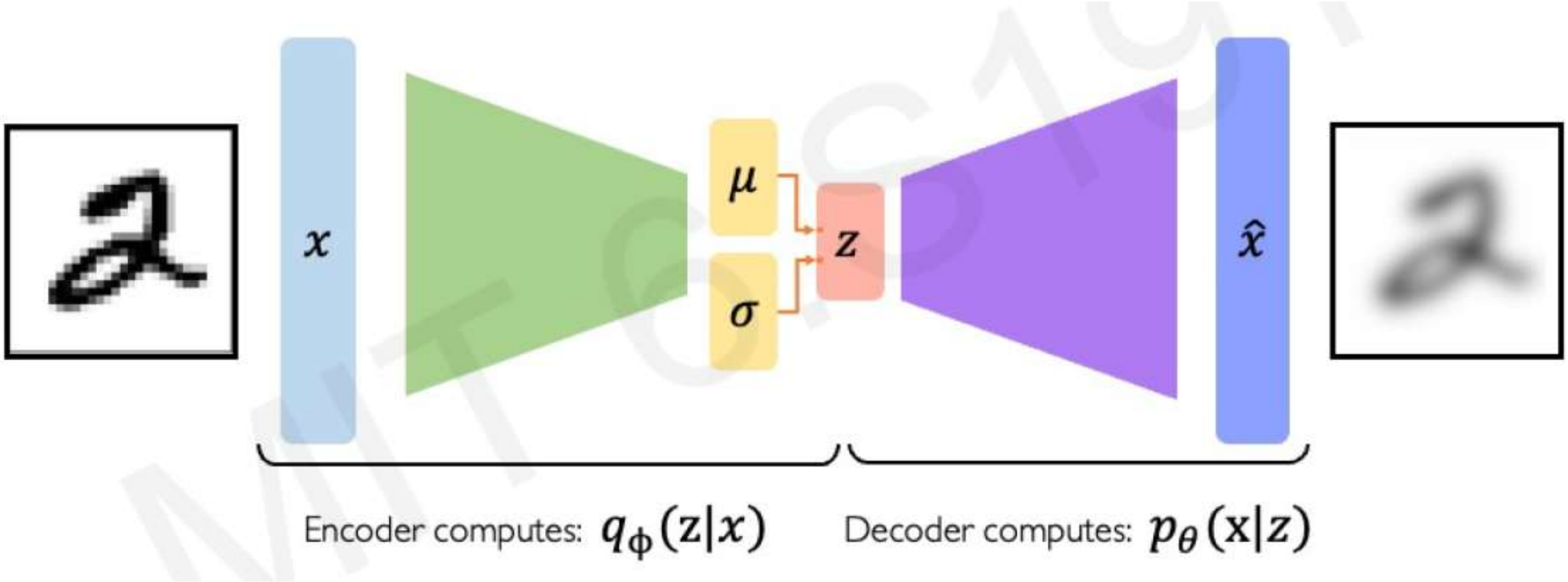
VAEs: Key Difference with Traditional Autoencoder



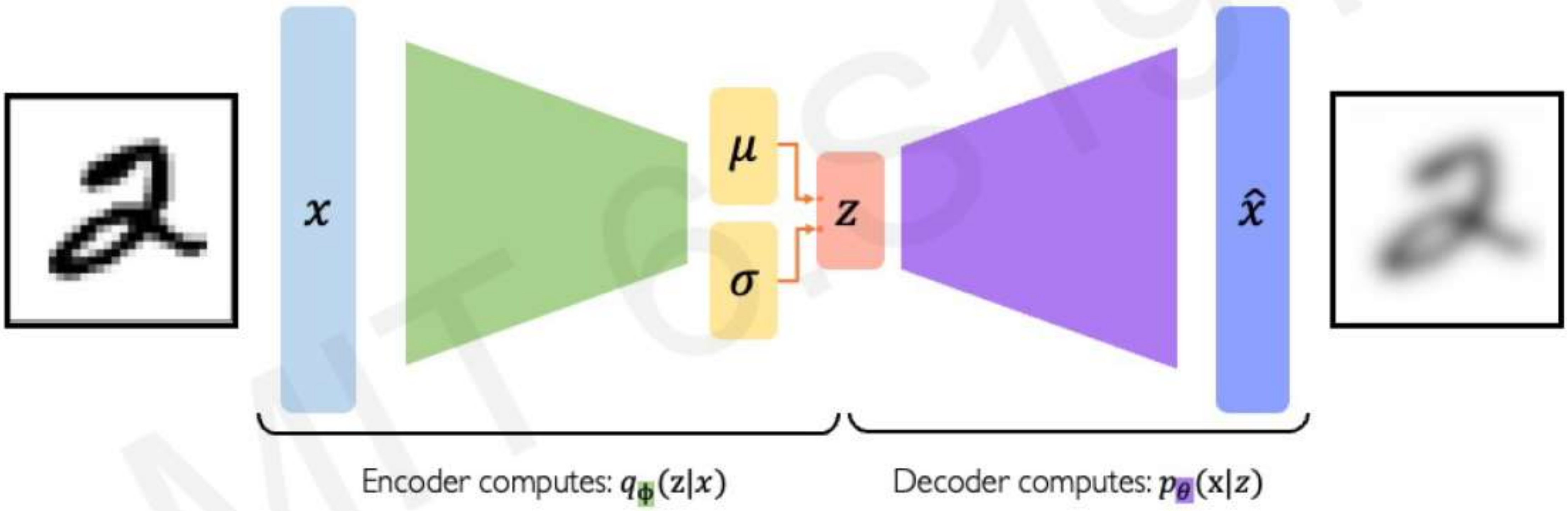
Variational autoencoders are a probabilistic twist on autoencoders!

Sample from the mean and standard deviation to compute latent sample

VAE Optimization

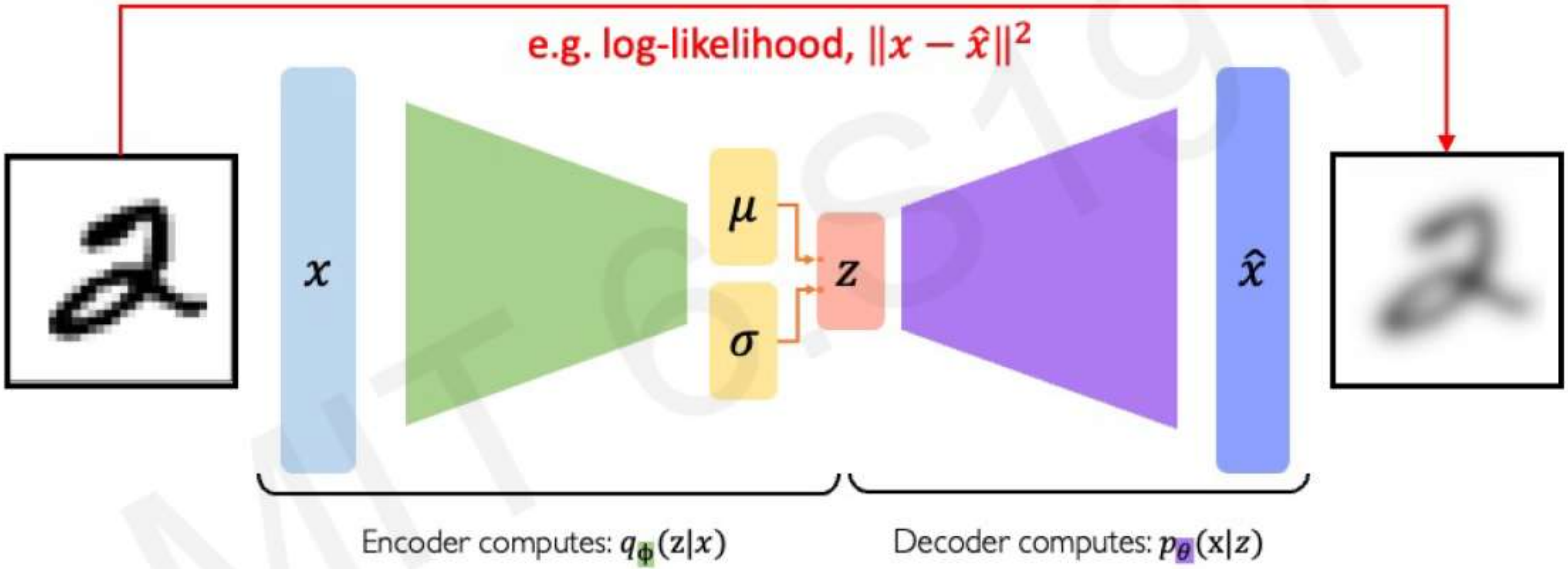


VAE Optimization



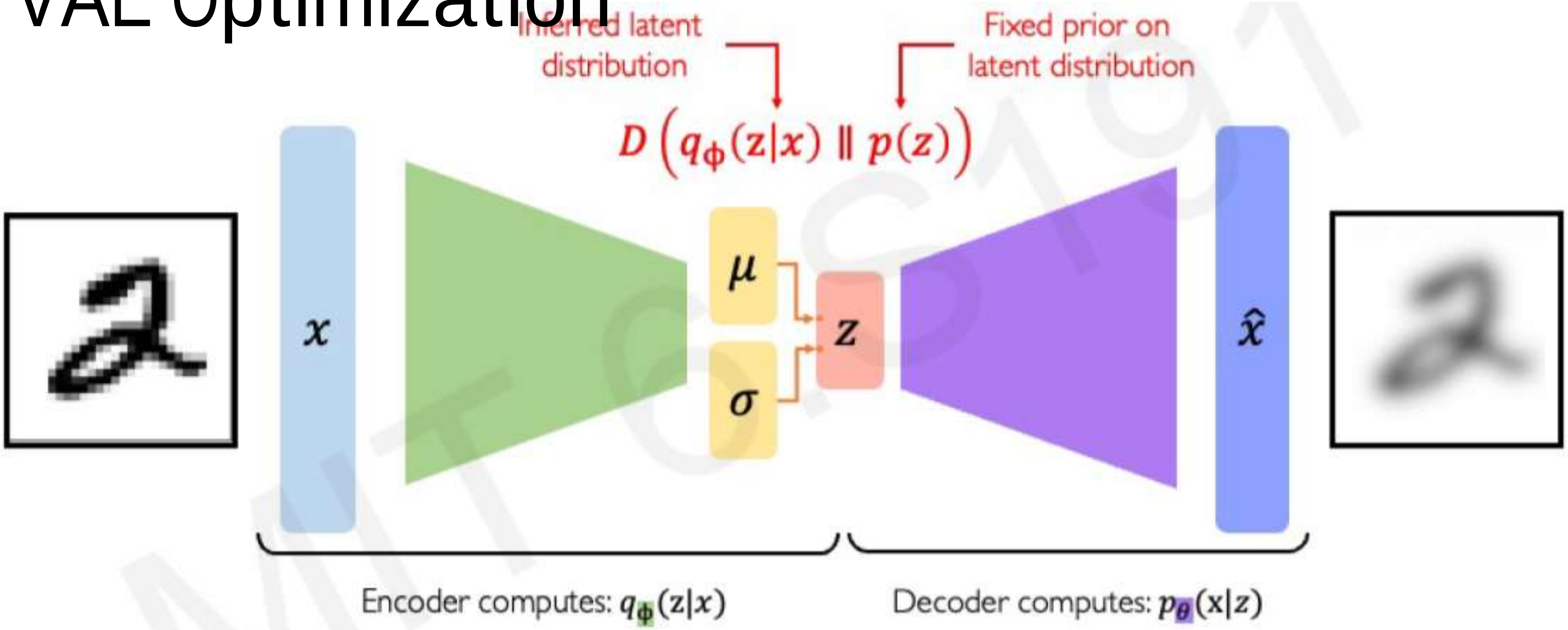
$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term})$$

VAE Optimization



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

VAE Optimization

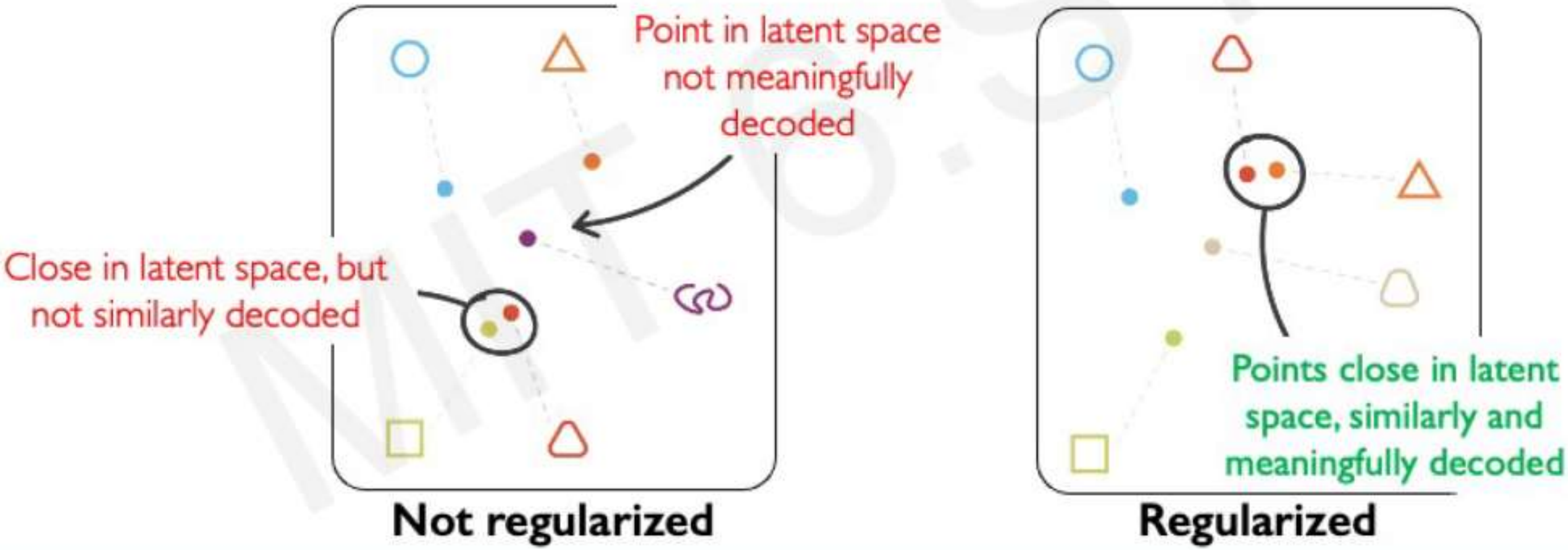


$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + \boxed{(\text{regularization term})}$$

Intuition on Regularization and the Normal Prior

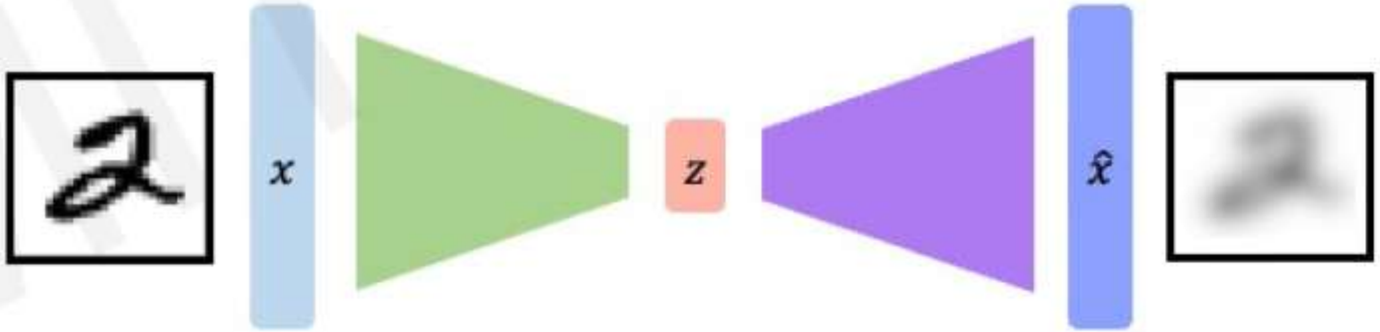
What properties do we want to achieve from regularization? 🤔

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



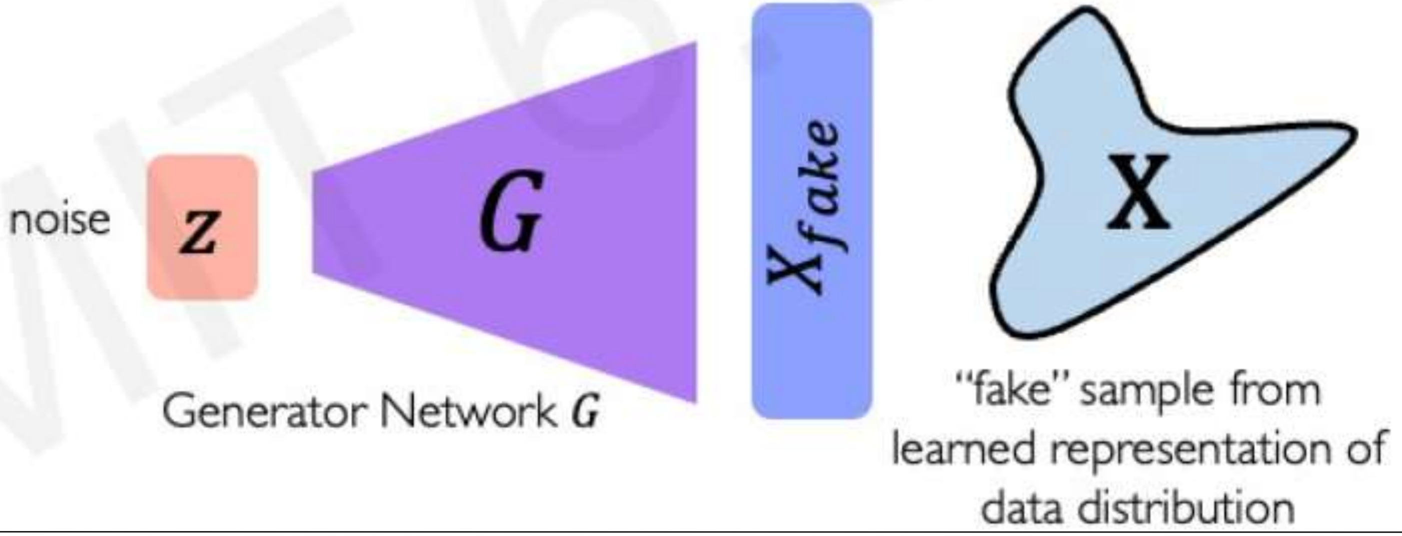
Generative Adversarial Network (GAN)

What if we Just Want to Sample?

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

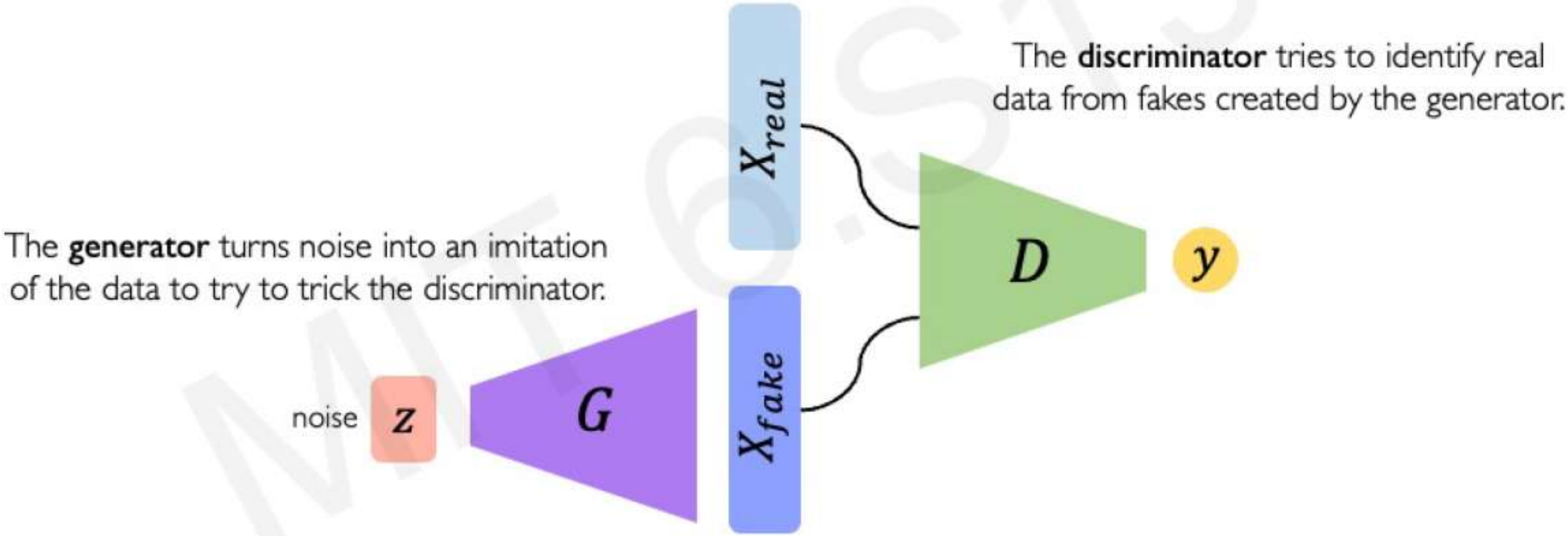
Problem: want to sample from complex distribution – can't do this directly!

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

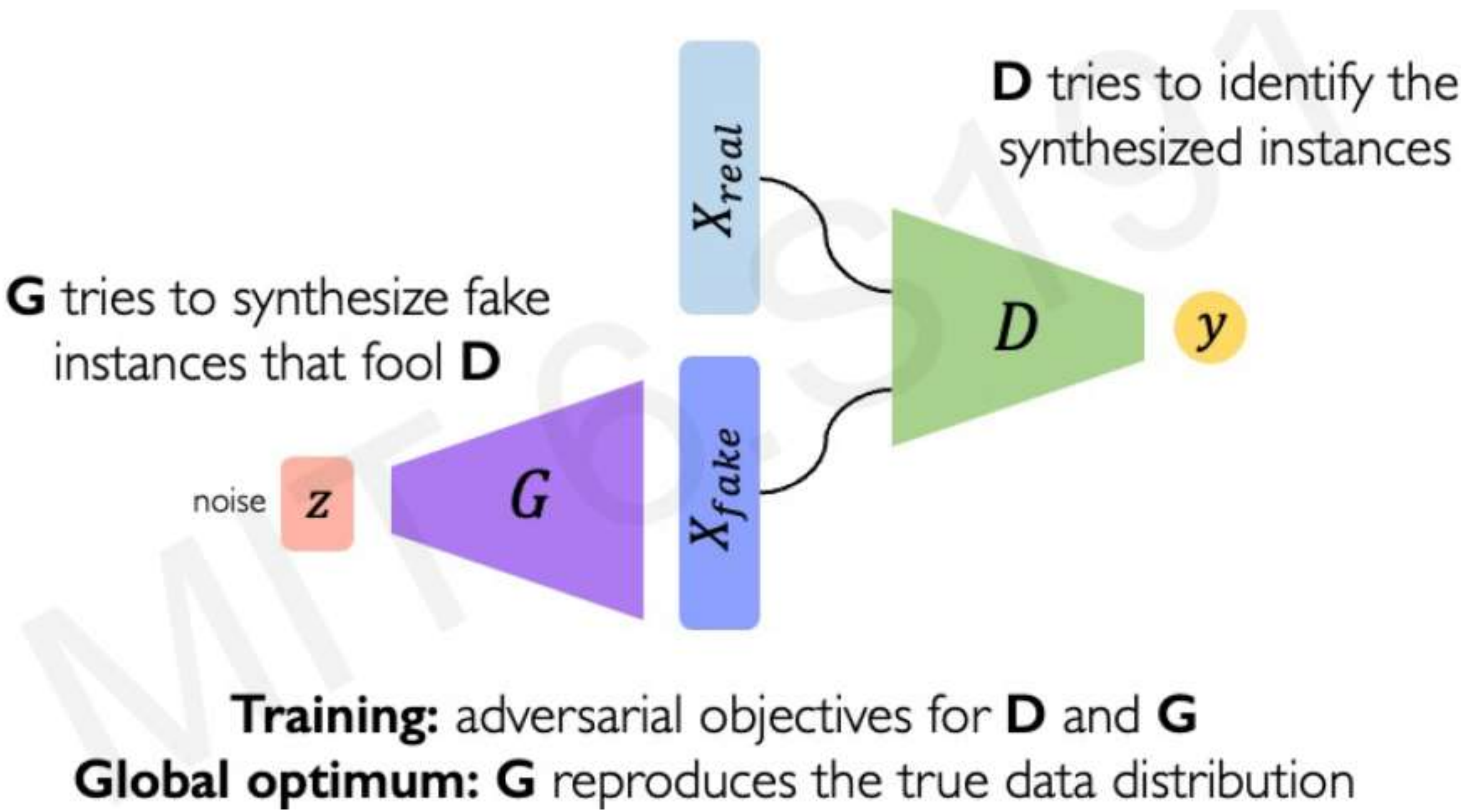


Generative Adversarial Network (GAN)

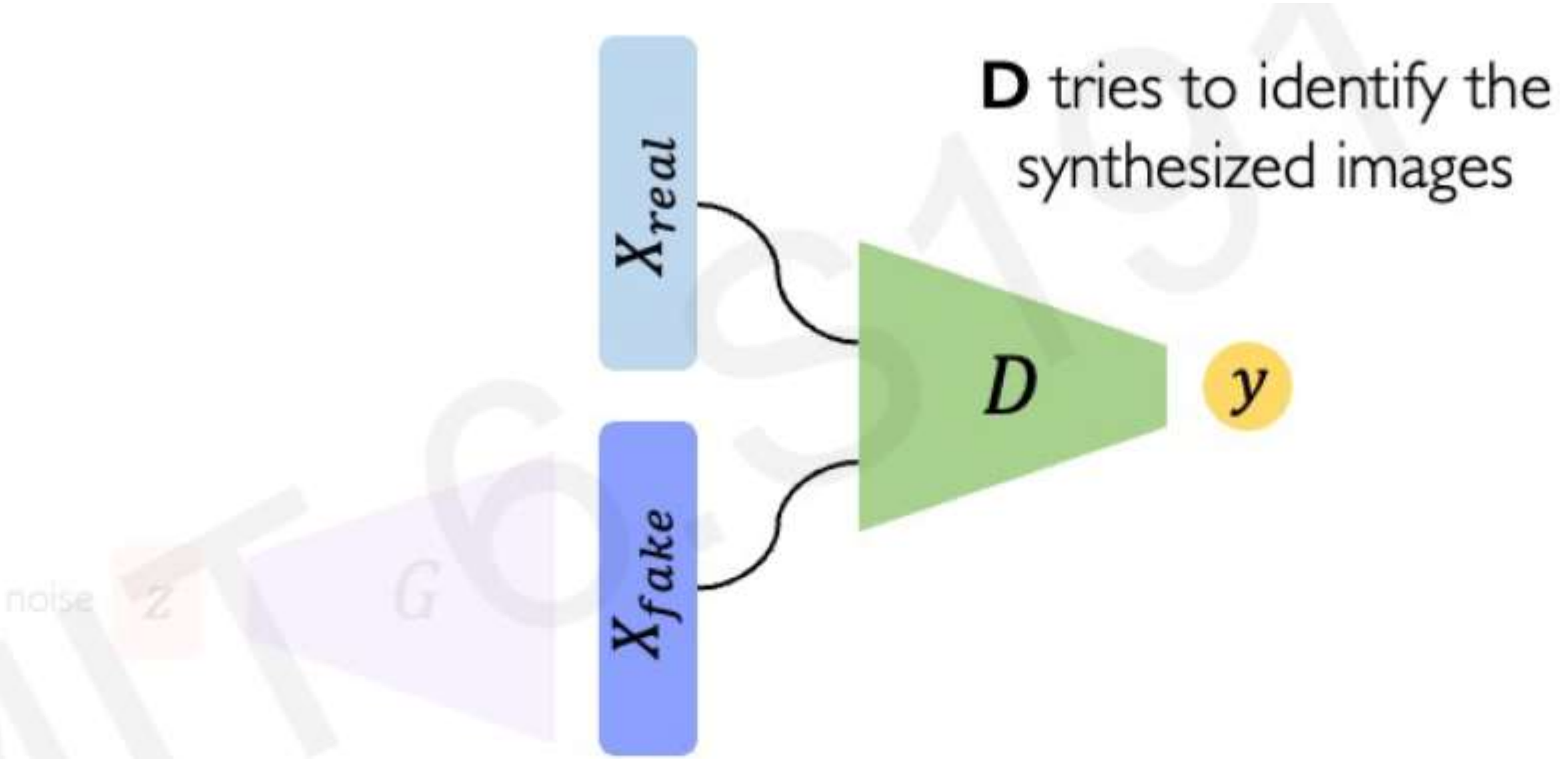
Generative Adversarial Networks (GANs) are a way to make a generative model by having two neural networks compete with each other.



Training GANs

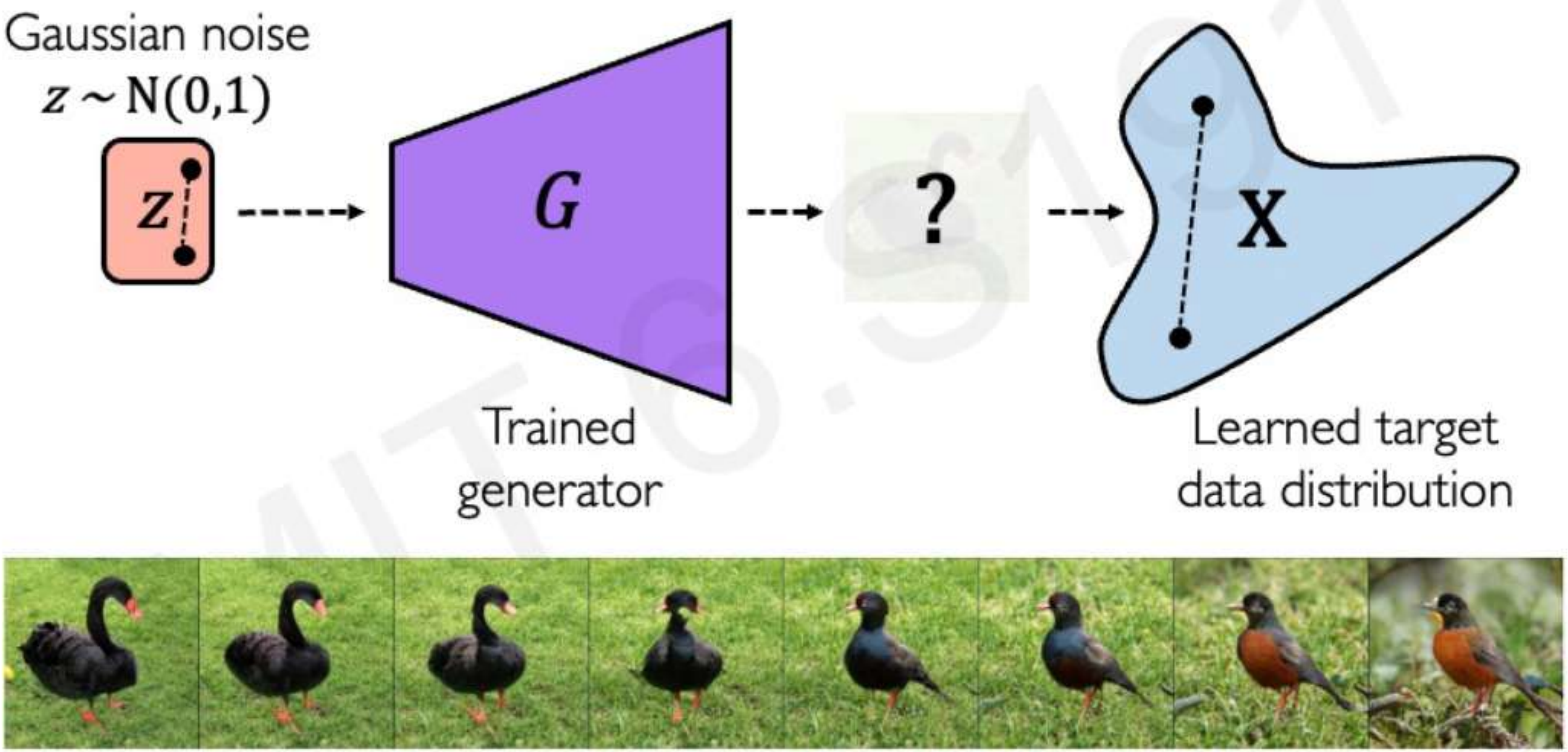


Training GANs: Loss Function



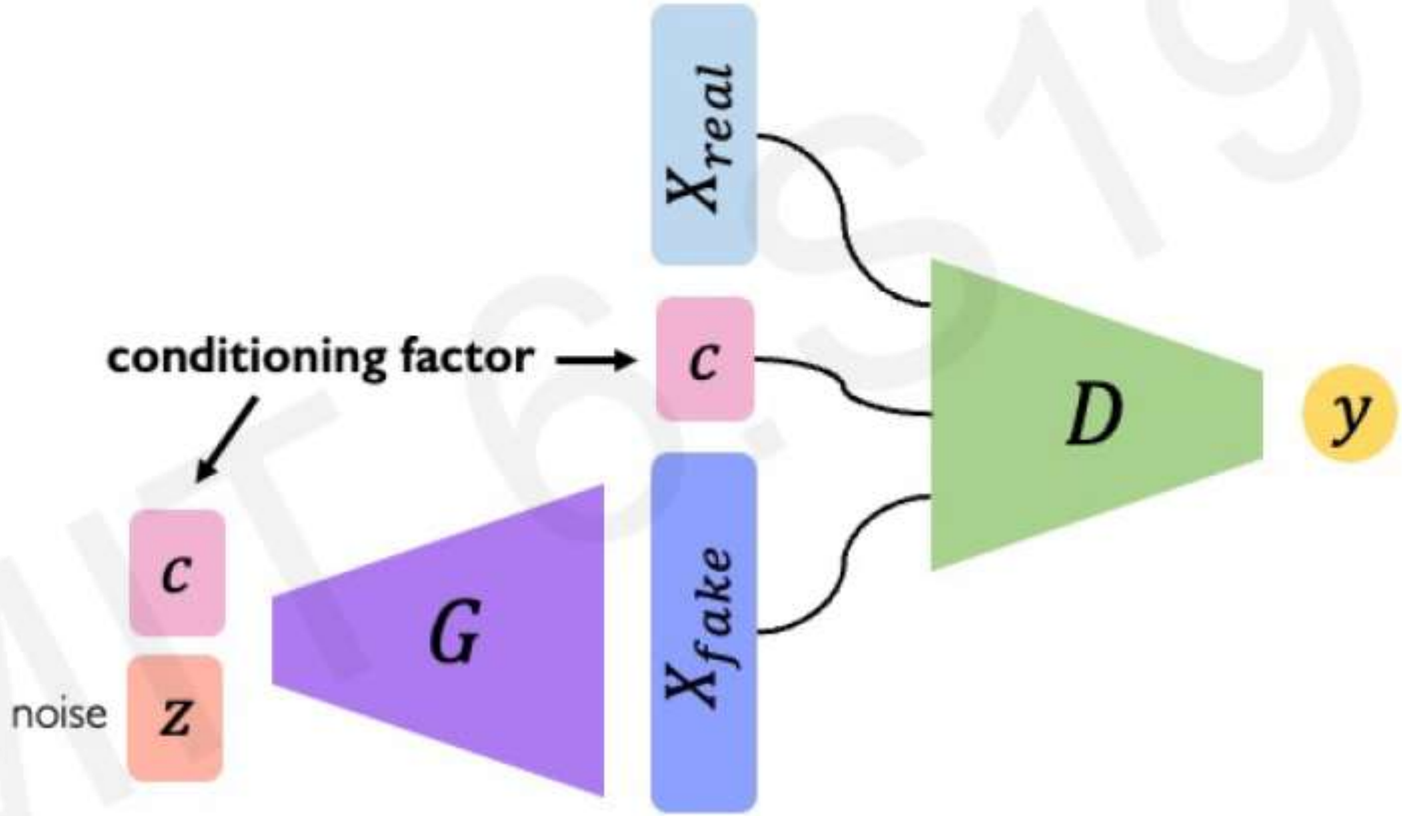
$$\underbrace{\arg\max_{D}}_{} \mathbb{E}_{\mathbf{z},\mathbf{x}}[\underbrace{\log D(G(\mathbf{z}))}_{\textbf{Fake}} + \underbrace{\log(1 - D(\mathbf{x}))}_{\textbf{Real}}]$$

GANs are Distribution Transformers



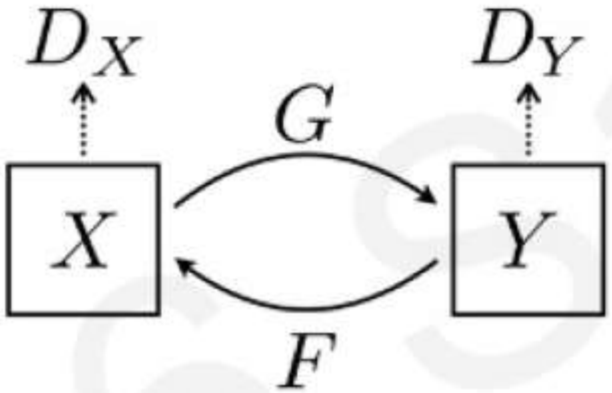
Conditional GANs

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

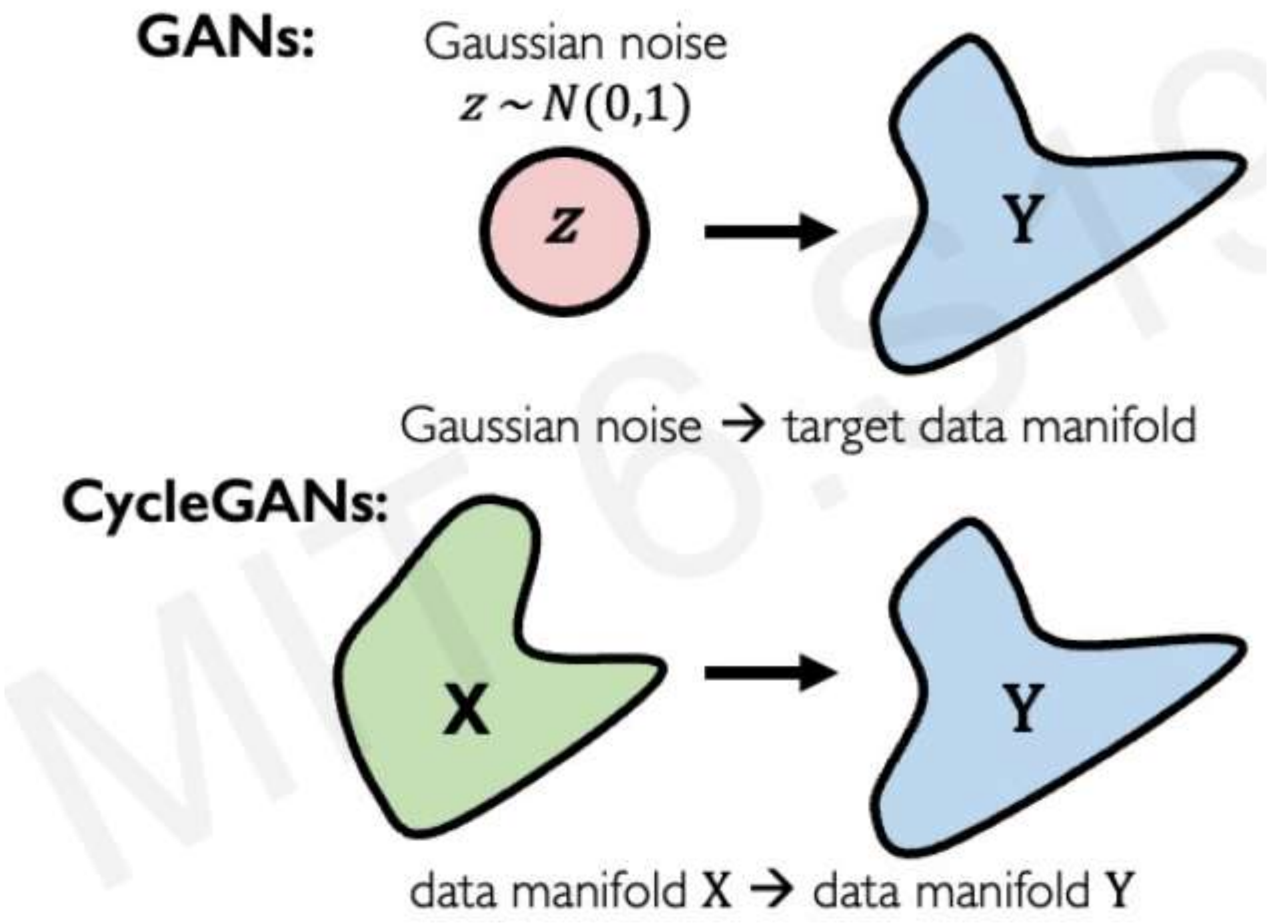


Cycle GANs

CycleGAN learns transformations across domains with unpaired data.



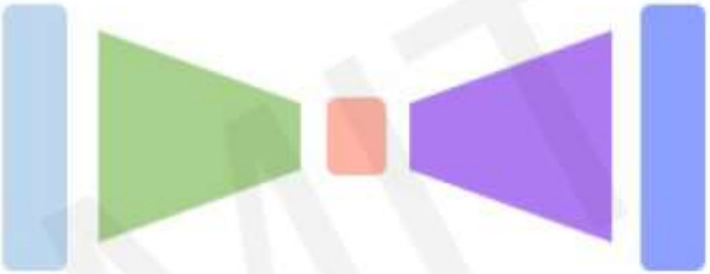
GANs vs Cycle GANs



Deep Generative Modeling Summary

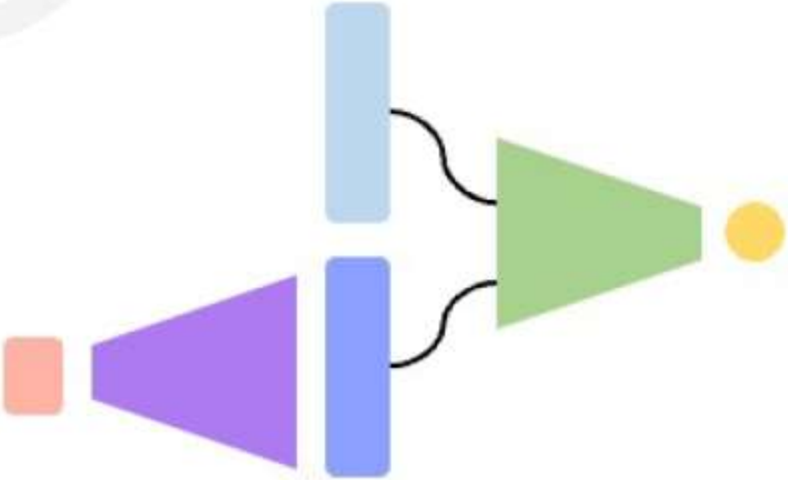
Autoencoders and Variational Autoencoders (VAEs)

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

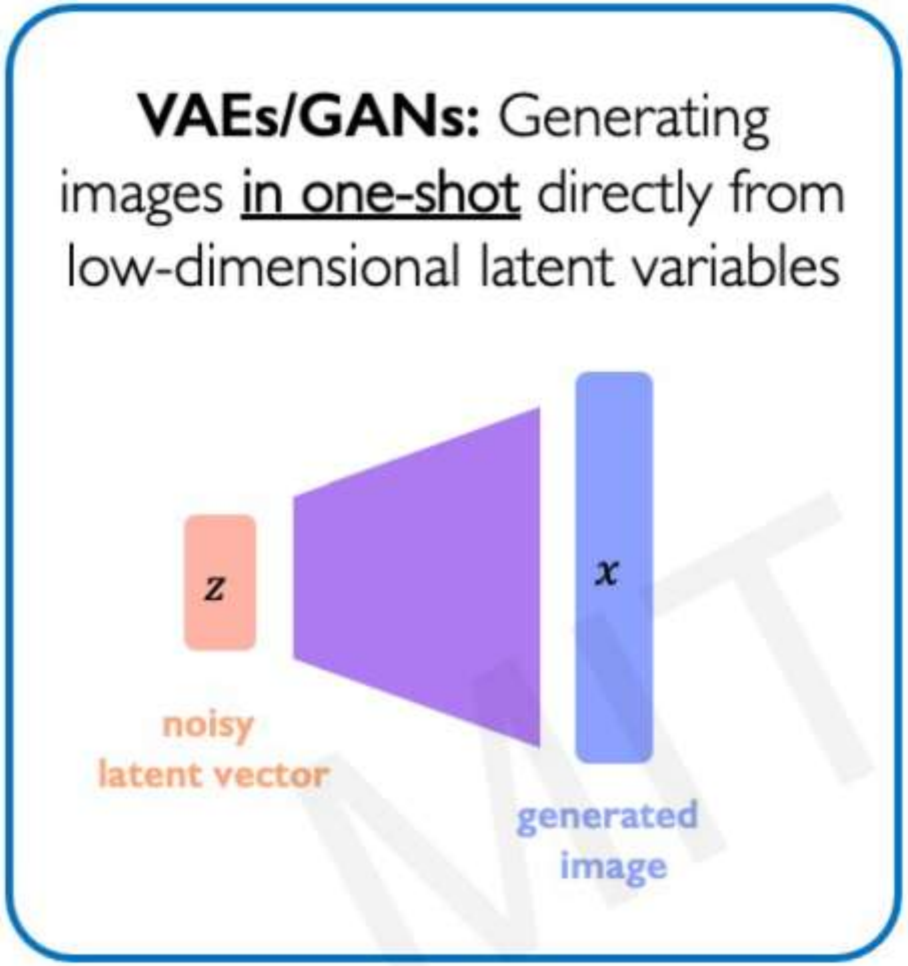


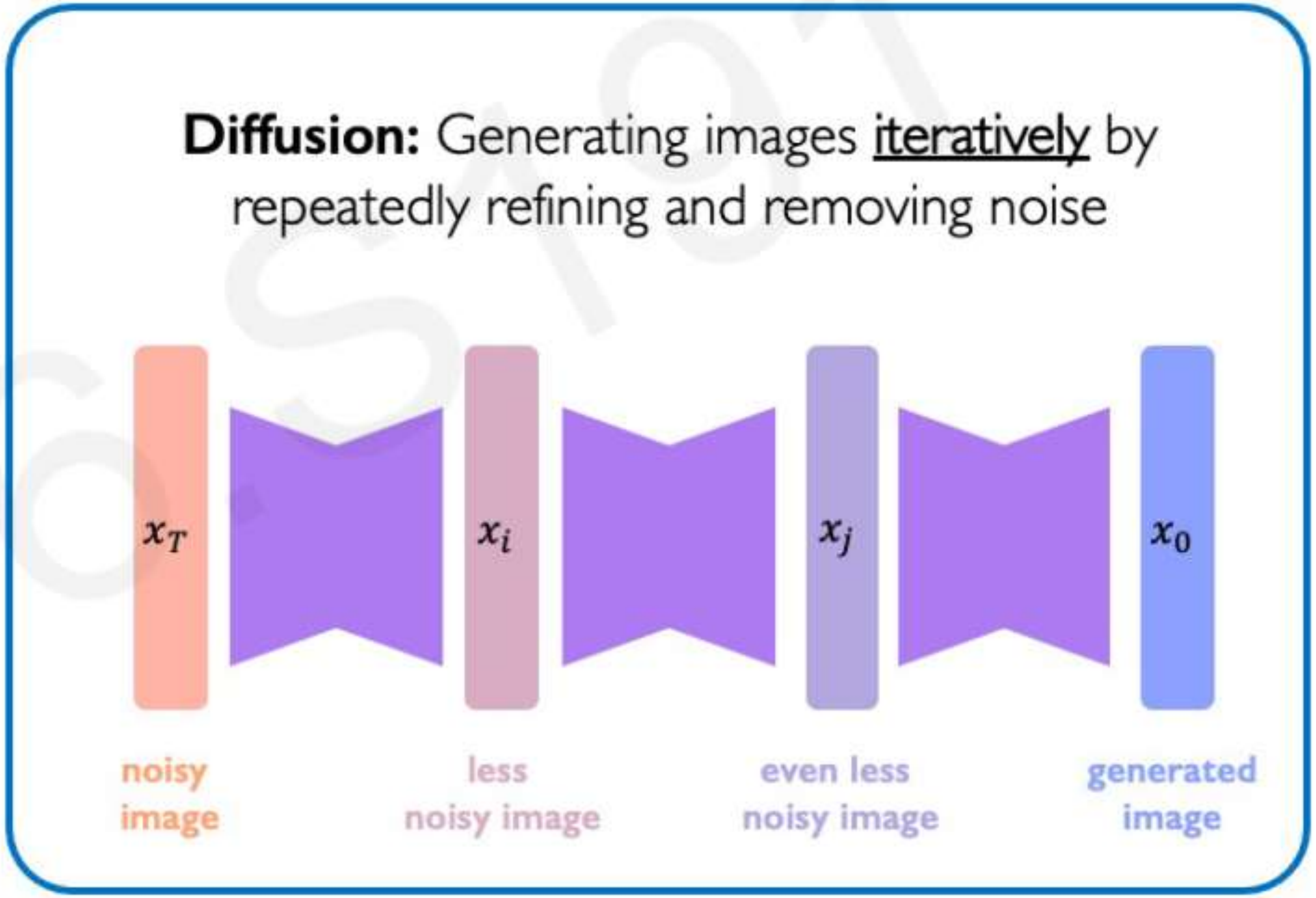
Generative Adversarial Networks (GANs)

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

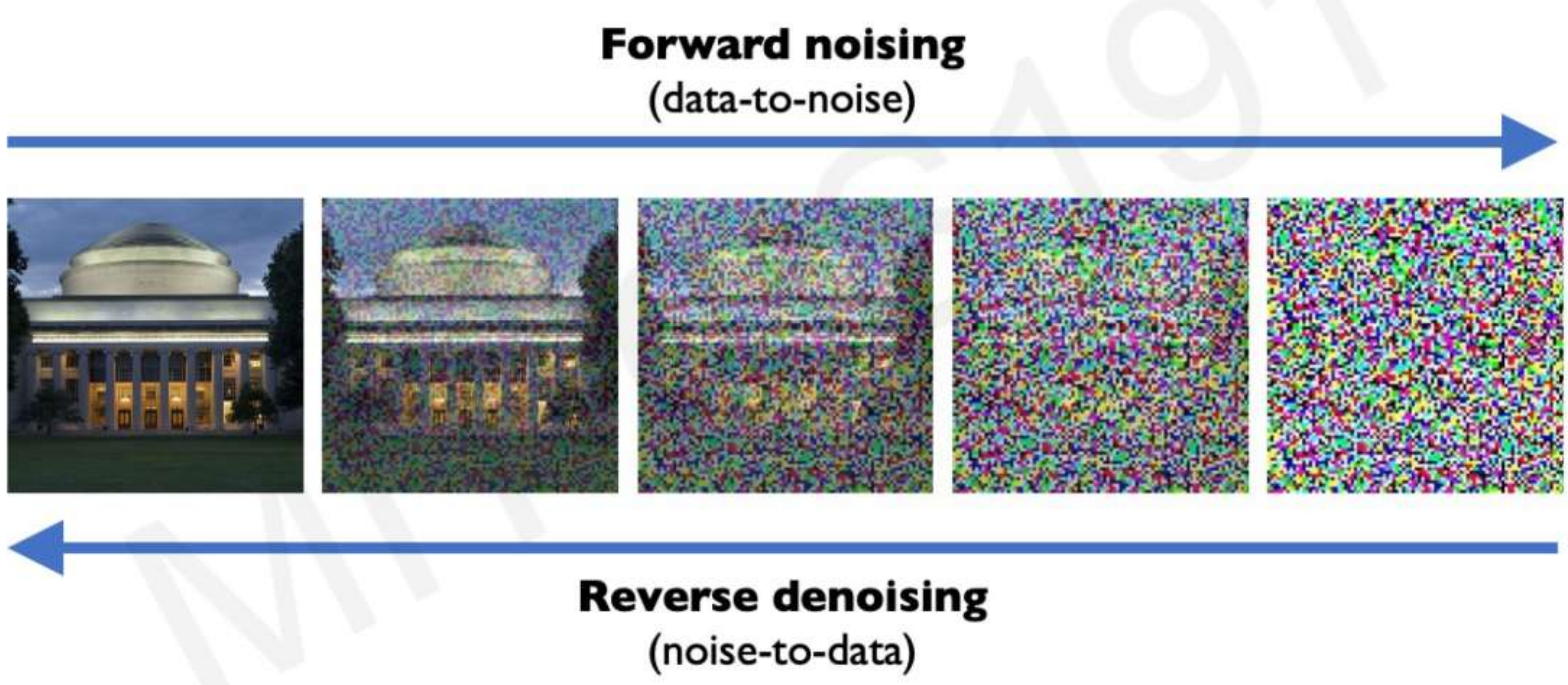


Diffusion Models



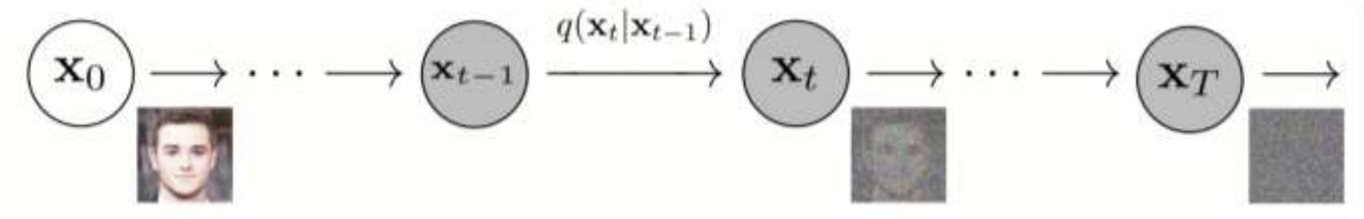


The Diffusion Process



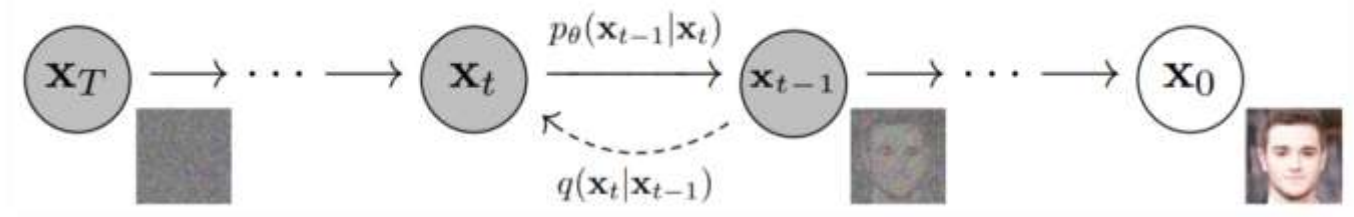
Diffusion Models

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 $\mathbf{x}_1, \ldots, \mathbf{x}_T$ 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.

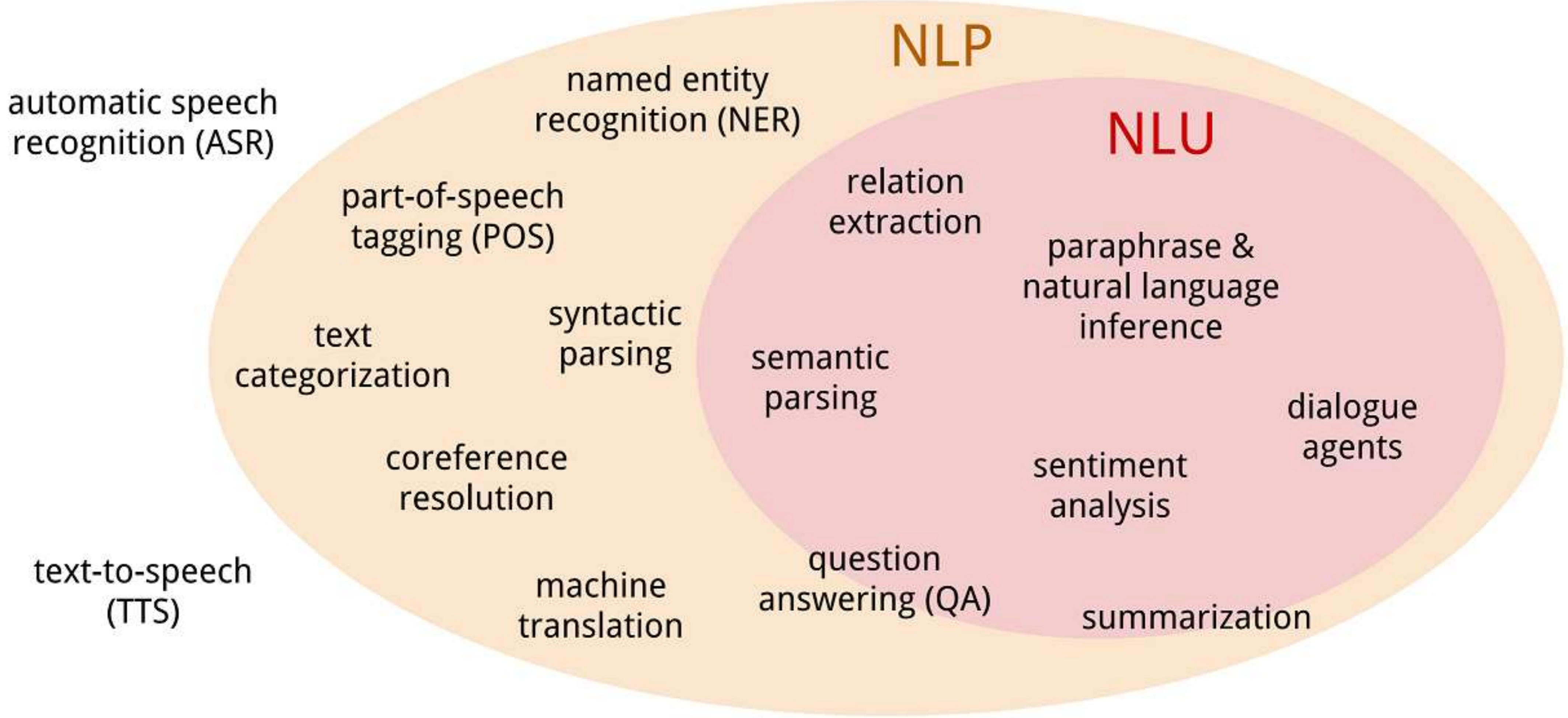


(Modified from source)

https://www.assemblyai.com/blog/diffusion-models-for-machine-learning-introduction/

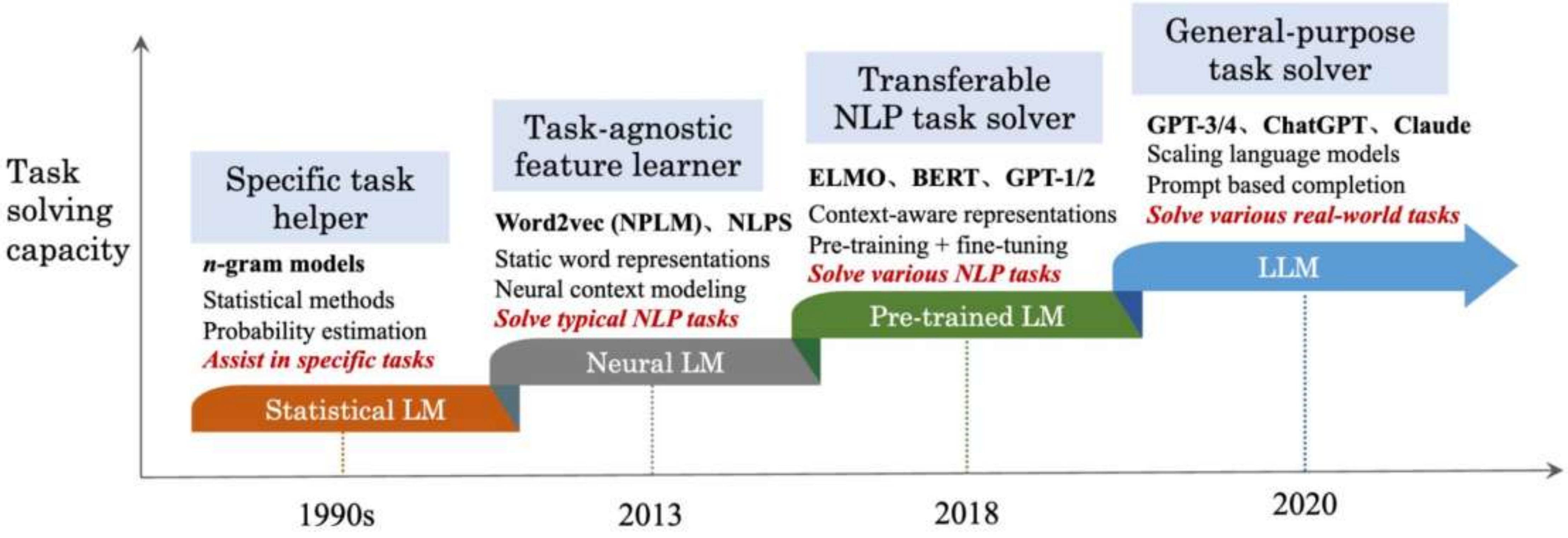
Deep Learning for Natural Language Processing

Natural Language Processing and Understanding



https://devopedia.org/natural-language-processing

Evolution of LMs for Task-Solving Capacity



W. Zhao et al. A Survey of Large Language Models. 2023.

Language Modelling

- **Language modelling** is the task of predicting which word comes next in a sequence of words.

- More formally, given a sequence of words $w_1, \ldots, w_t$ we want to know the probability of the next word, w_{t+1}:

$$P(w_{t+1} \mid w_1, \ldots, w_t)$$

- We are assuming that w_{t+1} comes from a finite vocabulary V.

  language models = classifiers

An Alternative View on Language Models

- Rather than as predictive models, language models can also be viewed as models that assign a probability to a piece of text.

  How likely is it that this piece of text is written in Swedish? French?

- These two views are equivalent, as the probability of a sequence can be expressed as a product of conditional probabilities: *

$$P(w_1 \cdots w_N) = \prod_{t=1}^{N} P(w_t \mid w_1, \dots, w_{t-1})$$

An Approximation

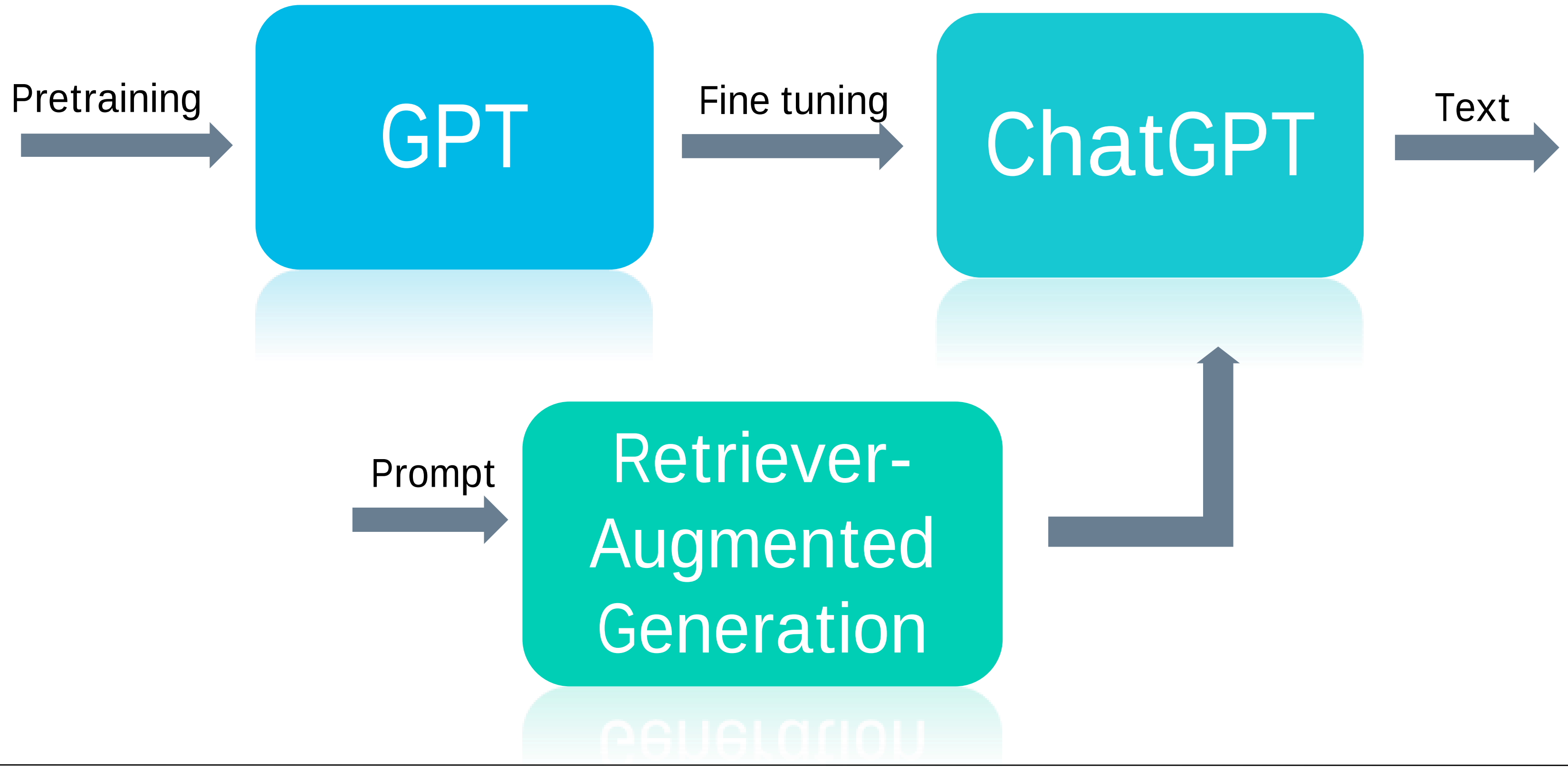
- A problem: computing *P(w | h)* exactly is infeasible for arbitrary history *h* since language is **creative** and *h* might have never occurred before!
- Idea: the Markov assumption: approximate the history by just the last few words.

$$P(w_k|w_{1:k-1}) \approx P(w_k|w_{k-n+1:k-1})$$

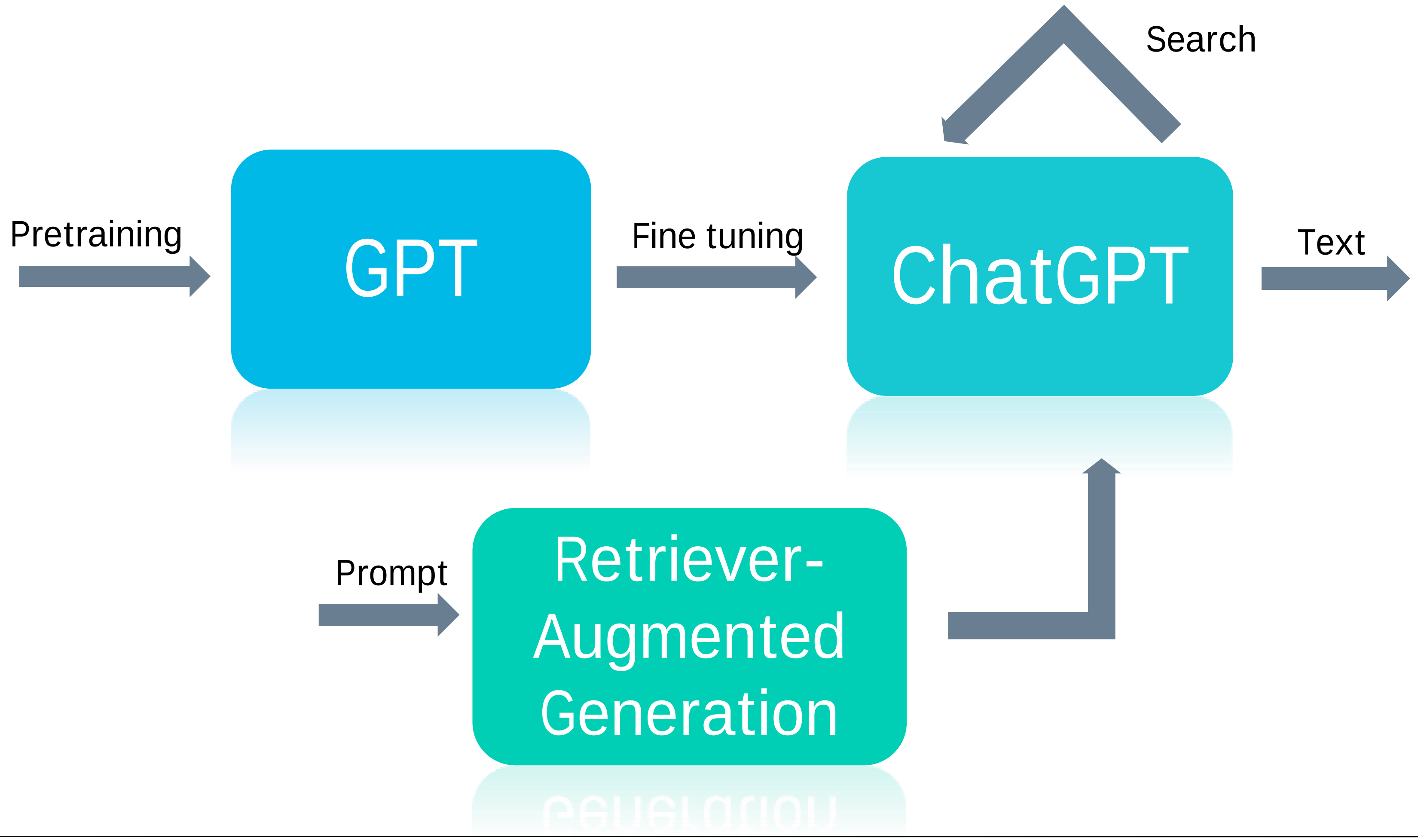
- Example: n-gram models: look at n-1 words in the history. n=2 is bigrams, n=3 is trigrams, etc.

- An **n-gram** is a contiguous sequence of *n* words (or characters).
  - Sherlock **Holmes** had **sprung out** and seized the intruder **by the collar**.
    unigram **bigram** **trigram**
- LLMs use **much** larger n, in the thousands or even millions!

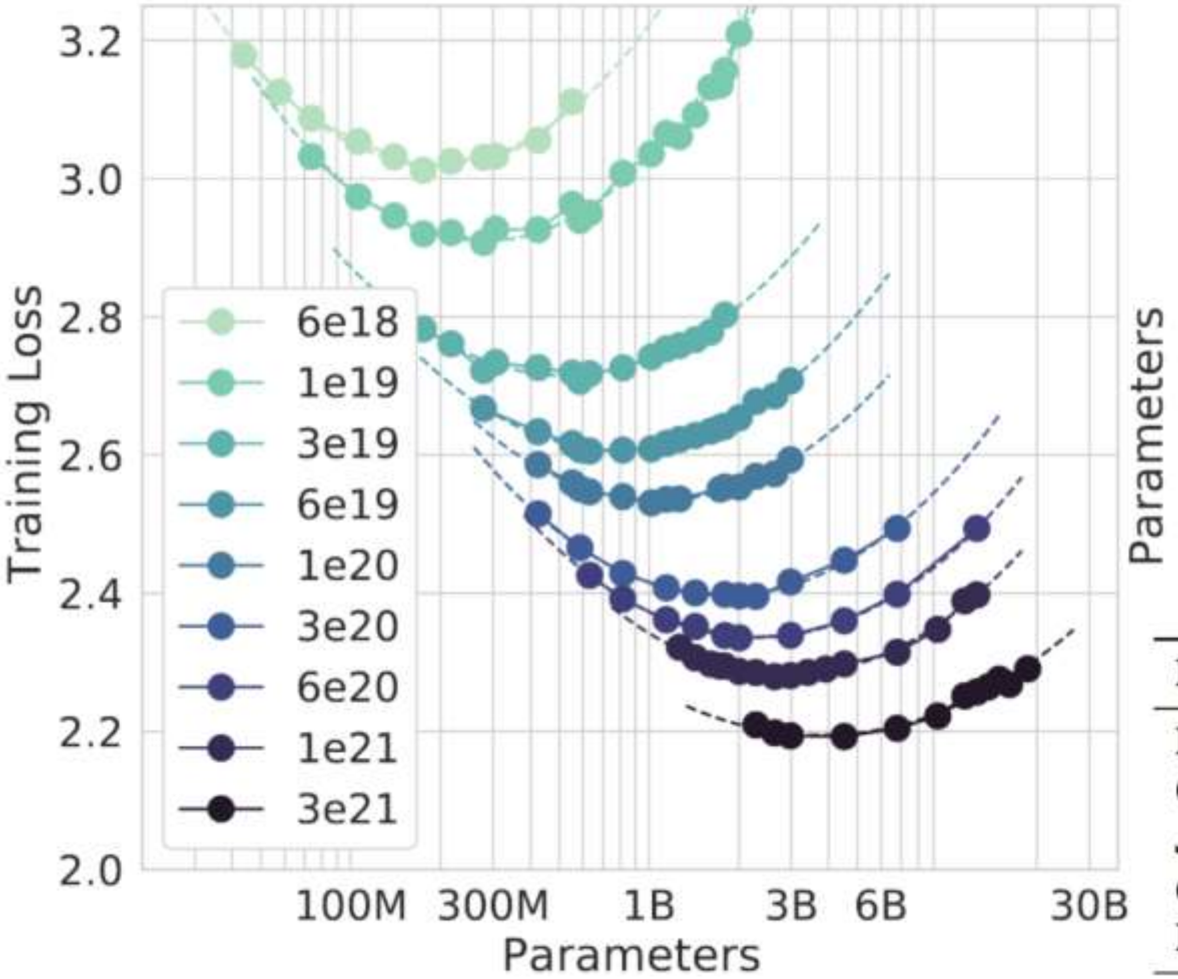
How Does ChatGPT Work?



How Does ChatGPT-o1 Work?



Large Language Models Scaling Laws

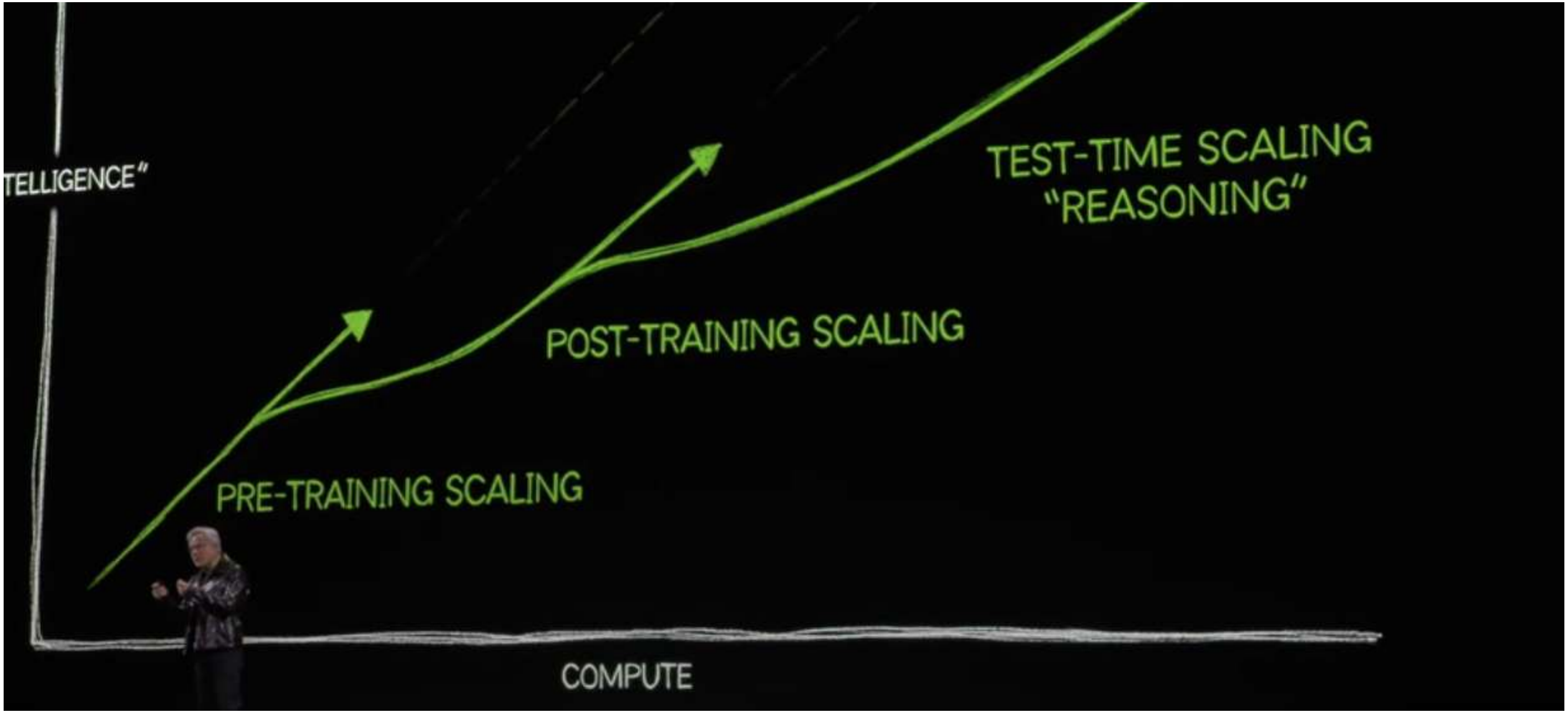


Parameters	FLOPs	FLOPs (in *Gopher* unit)	Tokens
400 Million	1.92e+19	1/29,968	8.0 Billion
1 Billion	1.21e+20	1/4,761	20.2 Billion
10 Billion	1.23e+22	1/46	205.1 Billion
67 Billion	5.76e+23	1	1.5 Trillion
175 Billion	3.85e+24	6.7	3.7 Trillion
280 Billion	9.90e+24	17.2	5.9 Trillion
520 Billion	3.43e+25	59.5	11.0 Trillion
1 Trillion	1.27e+26	221.3	21.2 Trillion
10 Trillion	1.30e+28	22515.9	216.2 Trillion

Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion

https://www.lesswrong.com/posts/midXmMb2Xg37F2Kgn/new-scaling-laws-for-large-language-models

Scaling – From Pre-training to Test-time



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 LE9 HT2025:
Neural networks
Deep Learning for Computer Vision
Deep Generative Models
Deep Learning for Natural Language Processing

www.ida.liu.se/~TDDC17

LINKÖPING UNIVERSITY