PyTorch 배우는 딥러닝 입문 (1)

Introduction to Deep Learning

이영완

한국전자통신연구원

이영완

ETRI 초지능창의연구소 시각지능연구실 (2017 ~)
Ph.D candidate in Graduate school of AI @KAIST

- 주요 연구 분야 (Google scholar)
  - Generative model
    - KOALA (NeurIPS`24)
  - Self-supervised learning
    - EVEREST (ICML`24)
    - RC-MAE (ICLR`23)
  - Neural architecture design:
    - VoVNet (CVPRW`19, CVPR`20)
    - MPViT (CVPR`22)
  - Object detection and segmentation
    - CenterMask (CVPR`20)
  - Video classification
    - VoV3D (IEEE Access)

- 수상 이력
  - 장관표창, 과학기술정보통신부 2022
  - 우수논문상, ETRI 2022, 2024
  - 우수강사상, ETRI 2022
  - #2 rank ImageNet challenge (detection task, 2017)

Youngwan Lee

Visual Intelligence lab
@ETRI

Senior researcher at ETRI &
Ph.D student at KAIST

Daejeon, South Korea

Email

LinkedIn

DBLP

Github

Google Scholar

ORCID

https://youngwanlee.github.io/

I'm **Youngwan**, a *senior researcher* at ETRI and *Ph.D student* in Graduate school of AI at KAIST, where I'm advised by Prof. Sung Ju Hwang in the Machine Learning and Artificial Intelligence (MLAI) lab. My research interest is how computers understand the world, including efficient 2D/3D neural network design, object detection, instance segmentation, semantic segmentation, and video classification 🖥️🌎. Recently, I have focused on Vision Transformer architecture and self-supervised learning.

🎉 News

- *2024.09*: 🎉🎉 KOALA has been accepted at NeurIPS 2024.
- *2024.08*: 🎉🎉 DiT-Pruner has been accepted at ECCVW 2024.
- *2024.05*: 🎉🎉 EVEREST has been accepted at ICML 2024.
- *2023.12*: 🎉🎉 KOALA, a fast Text-to-Image sythesis model, has been released.
- *2023.02*: 🎉🎉 Two papers have been accepted at ICLR 2023.
- *2022.02*: 🎉🎉 One paper has been accepted at CVPR 2022.

📚 Publications

Google Scholar full list

- **KOALA: Empirical Lessons Toward Memory-Efficient and Fast Diffusion Models for Text-to-Image Synthesis**

  Youngwan Lee, Kwanyong Park, Yoorhim Cho, Yong-Ju Lee, Sung Ju Hwang
  Advances in Neural Information Processing Systems (**NeurIPS**) 2024
  CVPR 2024 Workshop on Generative Models for Computer Vision
  [Project page][paper][code]

- **DiT-Pruner: Pruning Diffusion Transformer Models for Text-to-Image Synthesis Using Human Preference Scores**

  Youngwan Lee, Yong-Ju Lee, Sung Ju Hwang
  European Conference on Computer Vision (**ECCV**) 2024 Workshop on Green Foundation Models
  [paper]

Youngwan Lee

youngwanLEE

Senior researcher at ETRI & Ph.D student in Graduate school of AI at KAIST.

Edit profile

281 followers · **3** following

ETRI & MLAI@KAIST
South Korea
21:24 (UTC +09:00)
yw.lee@etri.re.kr
https://youngwanlee.github.io/
in/youngwanlee

sdxl-koala Public PyTorch

Compressing SDXL via knowledge-distillation

☆ 125 3

rc-mae Public PyTorch

[ICLR 2023] RC-MAE

Python ☆ 49 2

MPViT Public PyTorch

[CVPR 2022] MPViT:Multi-Path Vision Transformer for Dense Prediction

Python ☆ 354 39

UAD Public PyTorch

[ECCVW 2022] UAD: Localization Uncertainty Estimation for Anchor-Free Object Detection

Python ☆ 13

centermask2 Public PyTorch

[CVPR 2020] CenterMask : Real-time Anchor-Free Instance Segmentation

Python ☆ 771 159

vovnet-detectron2 Public PyTorch

[CVPR 2020] VoVNet backbone networks for detectron2

Python ☆ 372 69

Youngwan Lee

팔로우

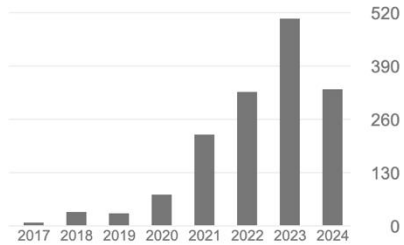
Electronics and Telecommunications Research Institute; KAIST
etri.re.kr의 이메일 확인됨 - 홈페이지

Computer Vision Machine Learning

제목	인용	연도
CenterMask: Real-time anchor-free instance segmentation Y Lee, J Park Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern …	631	2020
An Energy and GPU-Computation Efficient Backbone Network for Real-Time Object Detection Y Lee, J Hwang, S Lee, Y Bae, J Park Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern …	391	2019
MPViT: Multi-Path Vision Transformer for Dense Prediction Y Lee, J Kim, J Willette, SJ Hwang Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern …	243	2022

인용

	전체	2019년 이후
서지정보	1540	1497
h-index	11	10
i10-index	13	10



공동 저자 수정

https://github.com/youngwanLEE

https://scholar.google.com/citations?user=EqemKYsAAAAJ&hl=ko

강의 목표

- ‘***알***아두면 ***쓸***모있는 ***신***박한 ***딥***러닝 입문’ 탑재해드립니다.
- 딥러닝 오픈소스 대세인 Pytorch 첫 출발을 도와드립니다.
- Pytorch를 이용한 연구 개발 경험과 꿀팁을 드립니다.
- 자유롭게 질의응답할 수 있는 편안한 토론의 장을 선사합니다.

강의 추천 대상자

- 딥러닝에 대해 기초부터 차근차근 배우고 싶은 분
  여기저기서 딥러닝 딥러닝 하는데 도대체 딥러닝이 무엇인지 궁금하신 분

- 딥러닝을 시작하고 싶은데 어떻게 시작해야 될지 모르는 분
  더 늦기 전에 딥러닝을 시작해야 될텐데... 라고 생각하시는 분

- 코드로 딥러닝 개념을 이해하고 싶은 분
  글이나 말보다는 코드로 세상을 이해하시는 분

- Pytorch를 업무에 활용하고 싶은 분
  요즘 딥러닝은 pytorch가 대세라던데... pytorch에 발을 담그고자 하시는 분

강의 내용

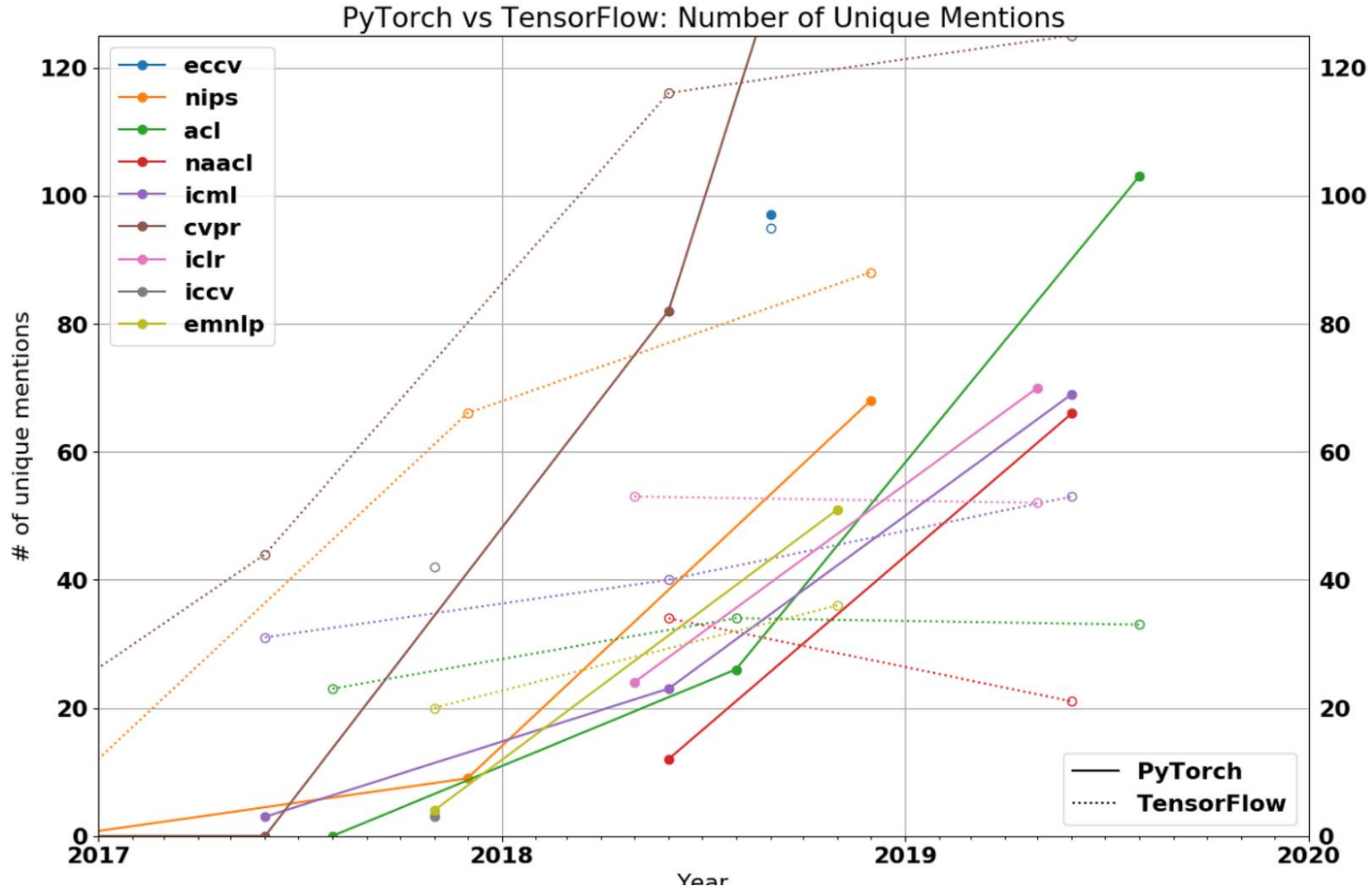
[1day]

- 이론 : 딥러닝 기초
  - Overview of Machine Learning (ML)
  - Introduction to Neural Network
- 실습 : Pytorch Tutorial & AutoGrad

[2day]

- 이론 : Convolutional Neural Network (CNN)
- 실습 : CNN 네트워크 구현 및 Image classification 실습

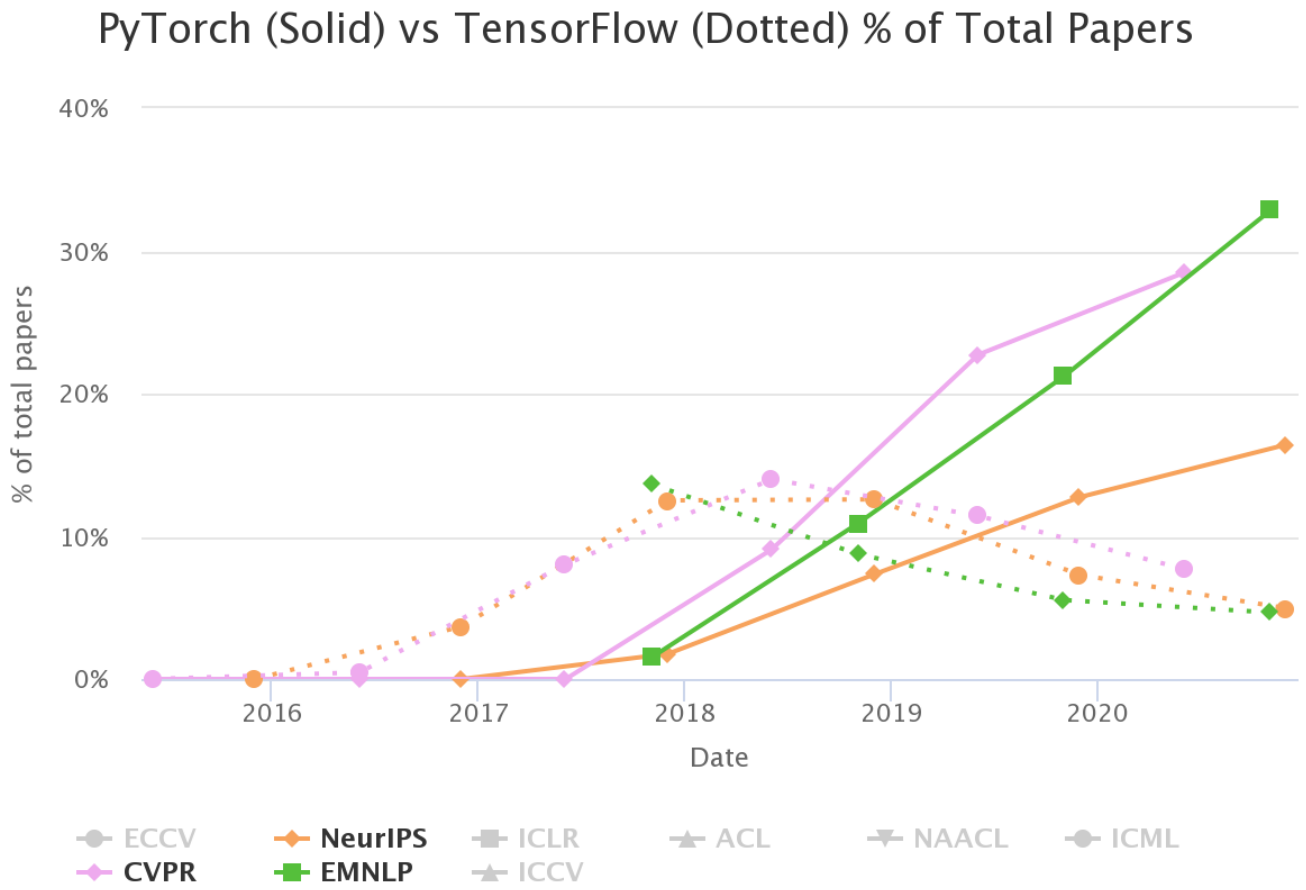
Why Pytorch?



CONFERENCE	PT 2018	PT 2019	PT GROWTH	TF 2018	TF 2019	TF GROWTH
CVPR	82	280	240%	116	125	7.7%
NAACL	12	66	450%	34	21	-38.2%
ACL	26	103	296%	34	33	-2.9%
ICLR	24	70	192%	54	53	-1.9%
ICML	23	69	200%	40	53	32.5%

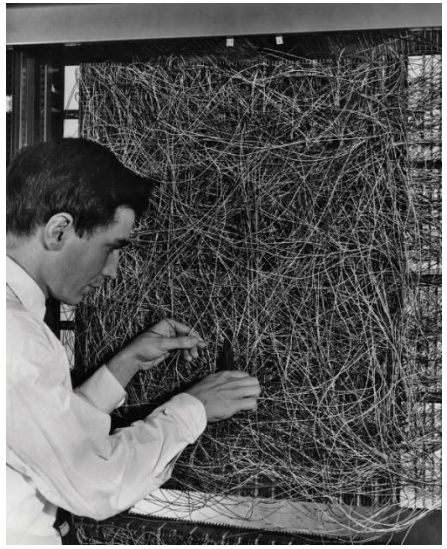
https://data-newbie.tistory.com/425

Why Pytorch?



https://data-newbie.tistory.com/425

Artificial Intelligence in Context of Human History



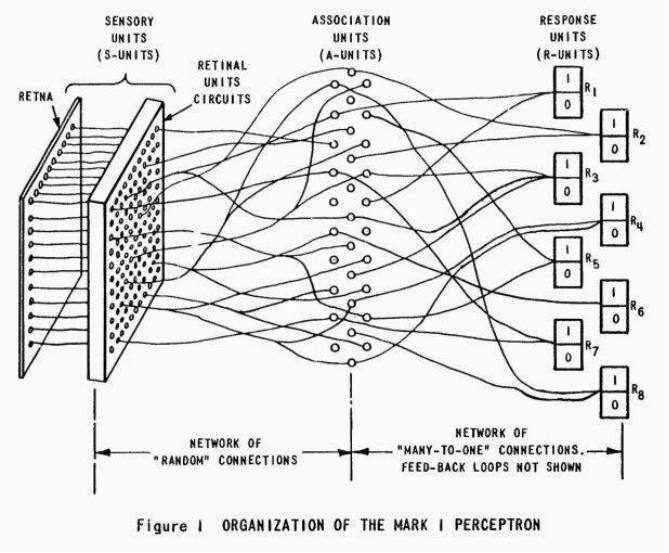


Figure 1 ORGANIZATION OF THE MARK I PERCEPTRON

Dreams, mathematical foundations, and engineering in reality.

Frank Rosenblatt, Perceptron (1957, 1962): Early description and engineering of singlelayer and multilayer artificial neural network

Artificial Intelligence in Context of Human History

Kasparov vs Deep Blue, 1997

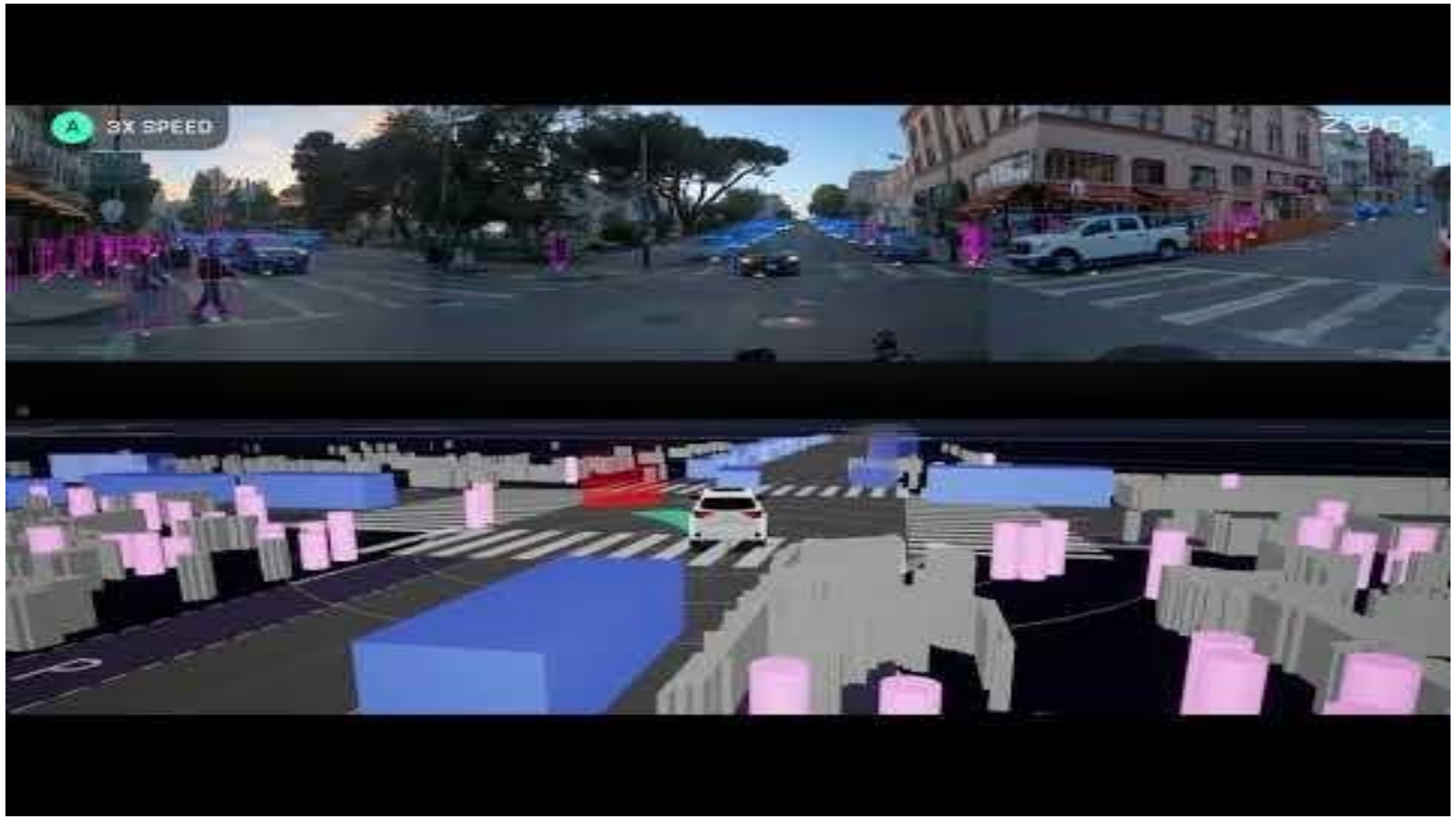
Artificial Intelligence in Context of Human History

Lee Sedol vs AlphaGo, 2016

Artificial Intelligence in Context of Human History

Self-driving car

Artificial Intelligence in Context of Human History



Visual perception for Self-driving car

Artificial Intelligence in Context of Human History

Robotics

Artificial Intelligence in Context of Human History

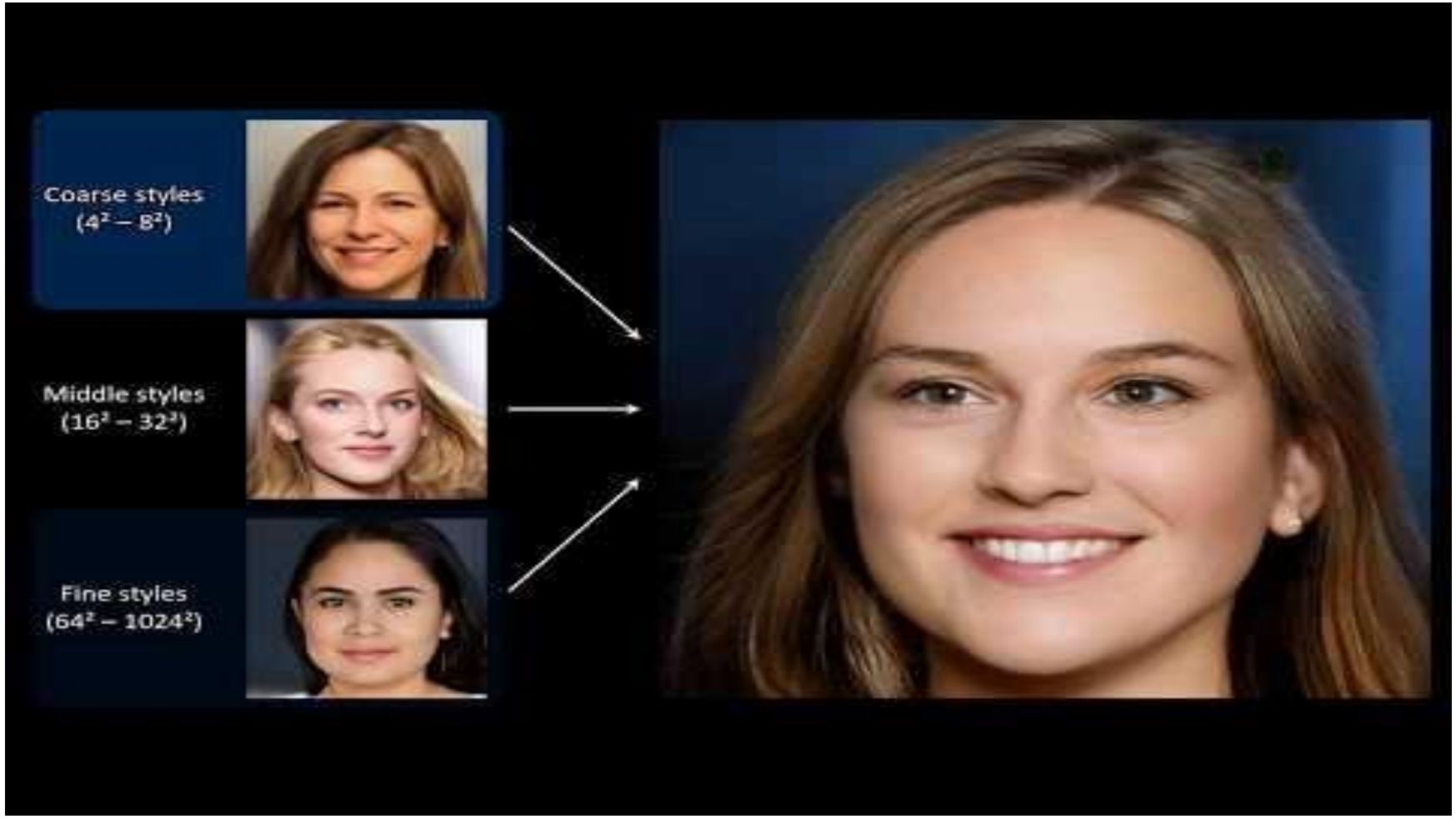


Image generation

Artificial Intelligence in Context of Human History

Image generation

Artificial Intelligence in Context of Human History

Audio style transfer

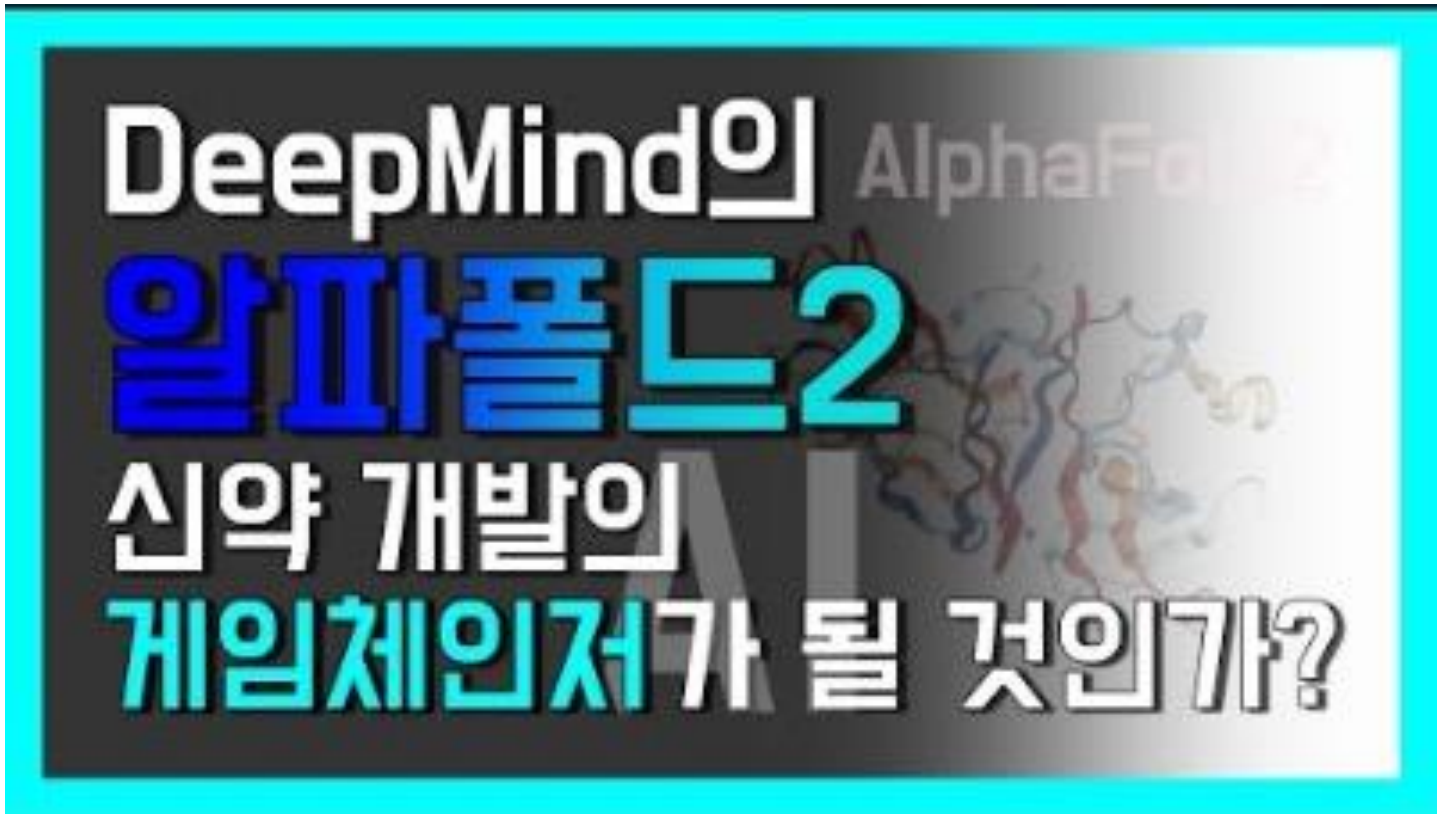
Artificial Intelligence in Context of Human History

AI Language Modeling : GPT-3

Artificial Intelligence in Context of Human History

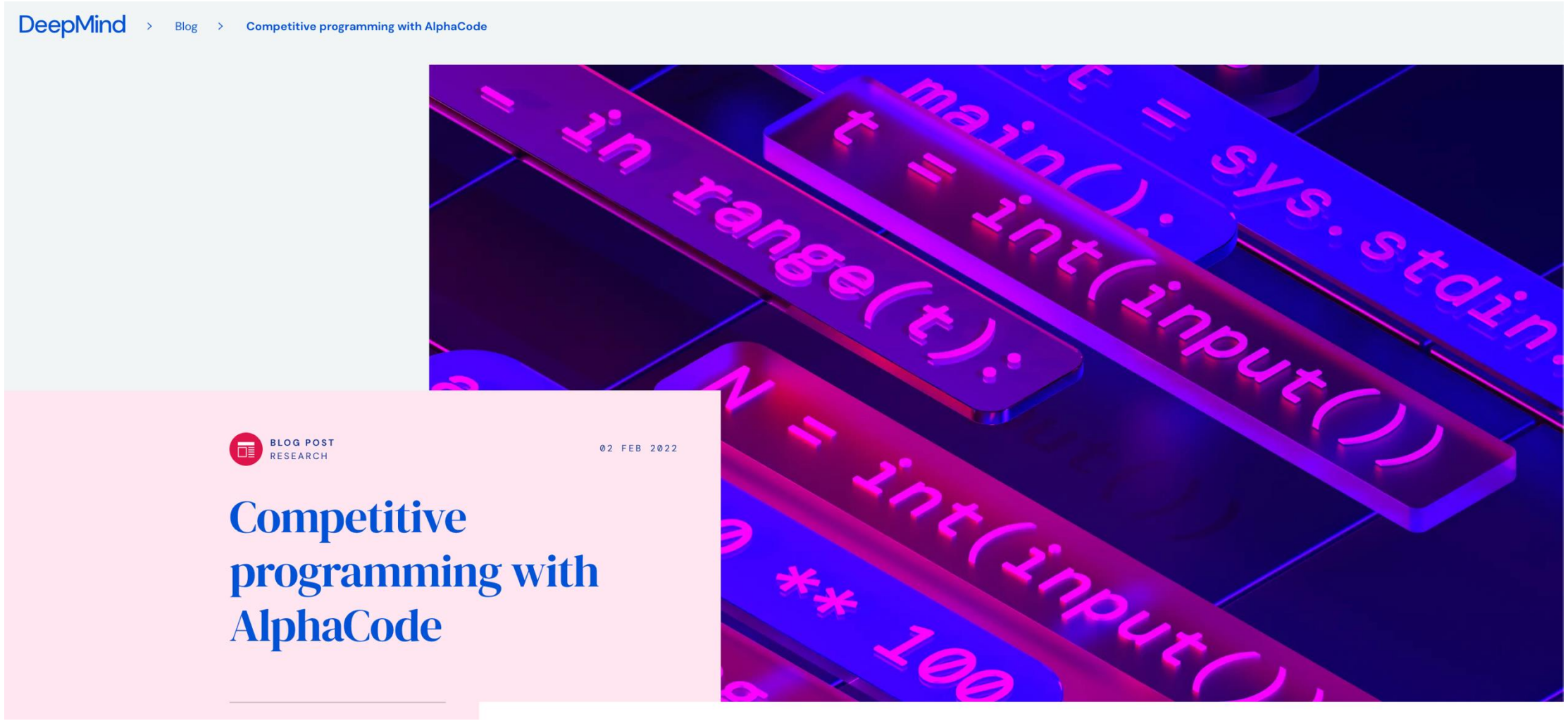
Natural Language Processing + Vision : Dall-E

Artificial Intelligence in Context of Human History



Protein Structure Prediction : Alphafold

Artificial Intelligence in Context of Human History

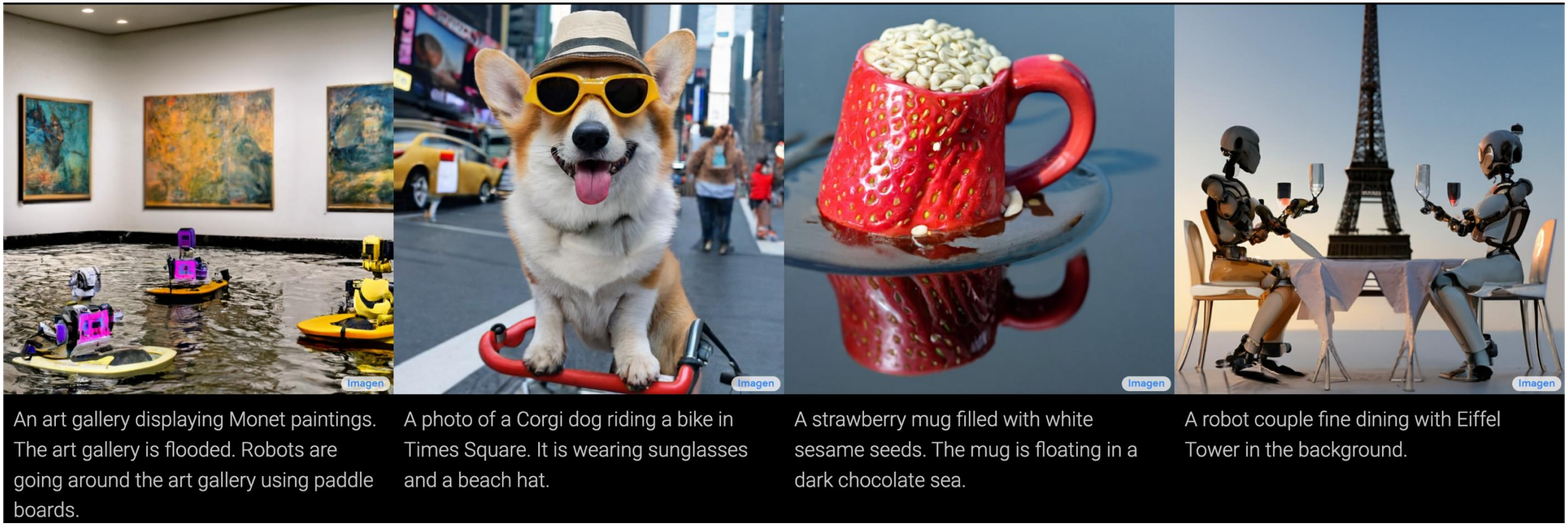


Competitive programming (2022.02.)

Artificial Intelligence in Context of Human History

Dall-E 2 (2022.04.)

Artificial Intelligence in Context of Human History



Text-to-Text: Imagen by Google (2022.05.)

https://imagen.research.google/

Artificial Intelligence in Context of Human History

Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with Stability AI and Runway and builds upon our previous work:

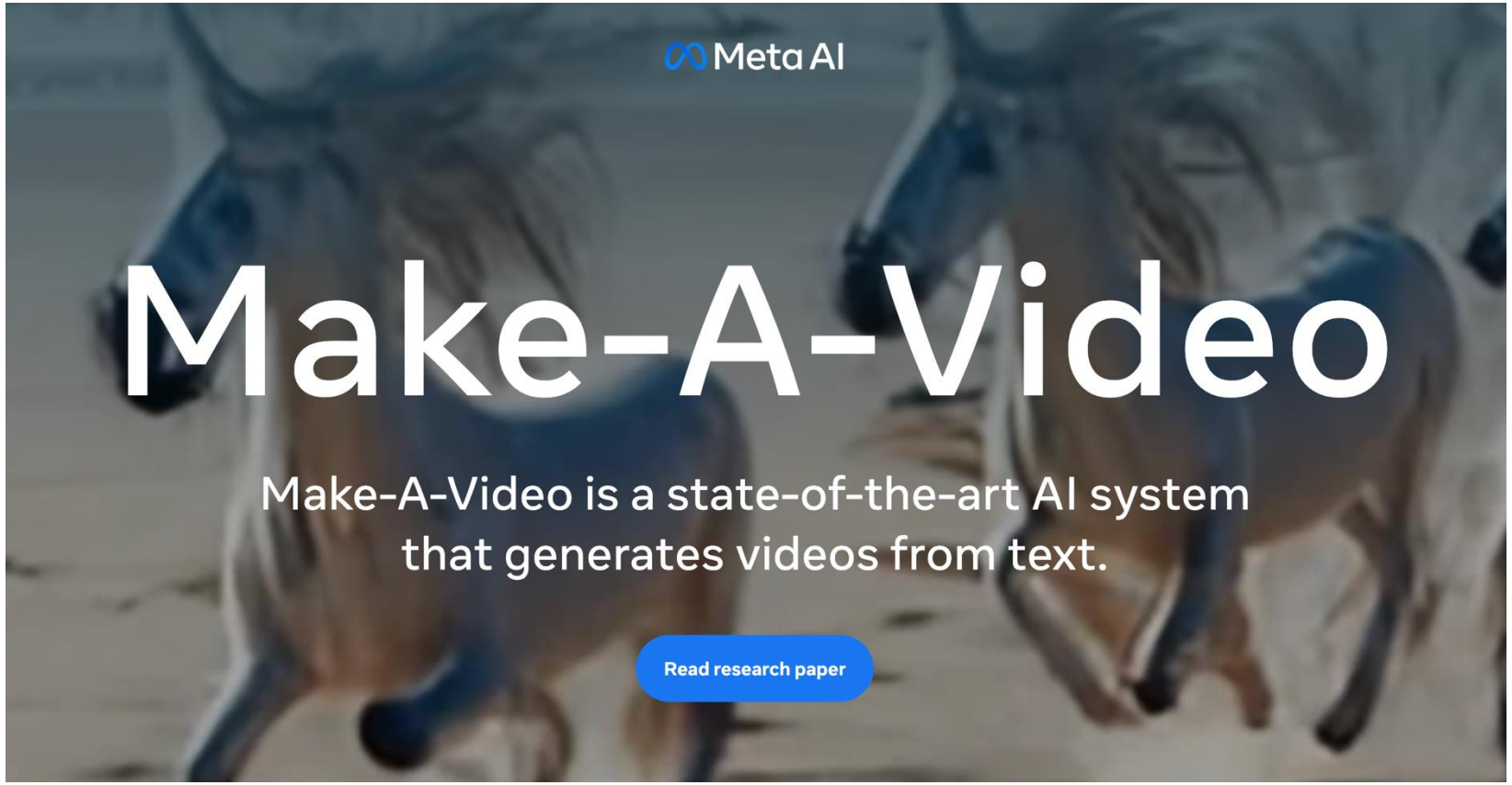
High-Resolution Image Synthesis with Latent Diffusion Models
Robin Rombach*, Andreas Blattmann*, Dominik Lorenz, Patrick Esser, Björn Ommer
CVPR '22 Oral | GitHub | arXiv | Project page

Stable Diffusion is a latent text-to-image diffusion model. Thanks to a generous compute donation from Stability AI and support from LAION, we were able to train a Latent Diffusion Model on 512x512 images from a subset of the LAION-5B database. Similar to Google's Imagen, this model uses a frozen CLIP ViT-L/14 text encoder to condition the model on text prompts. With its 860M UNet and 123M text encoder, the model is relatively lightweight and runs on a GPU with at least 10GB VRAM. See this section below and the model card.

Text-to-Text: Stable Diffusion (2022.08.)

https://huggingface.co/spaces/stabilityai/stable-diffusion
https://stability.ai/blog/stable-diffusion-public-release
https://github.com/CompVis/stable-diffusion

Artificial Intelligence in Context of Human History



Text-to-Video: Make-A-Video (2022.08.)

https://makeavideo.studio/

Artificial Intelligence in Context of Human History

Text-to-Video: Imagen Video (2022.10.)

https://imagen.research.google/video/

Artificial Intelligence in Context of Human History

ChatGPT (2022.11.30.)

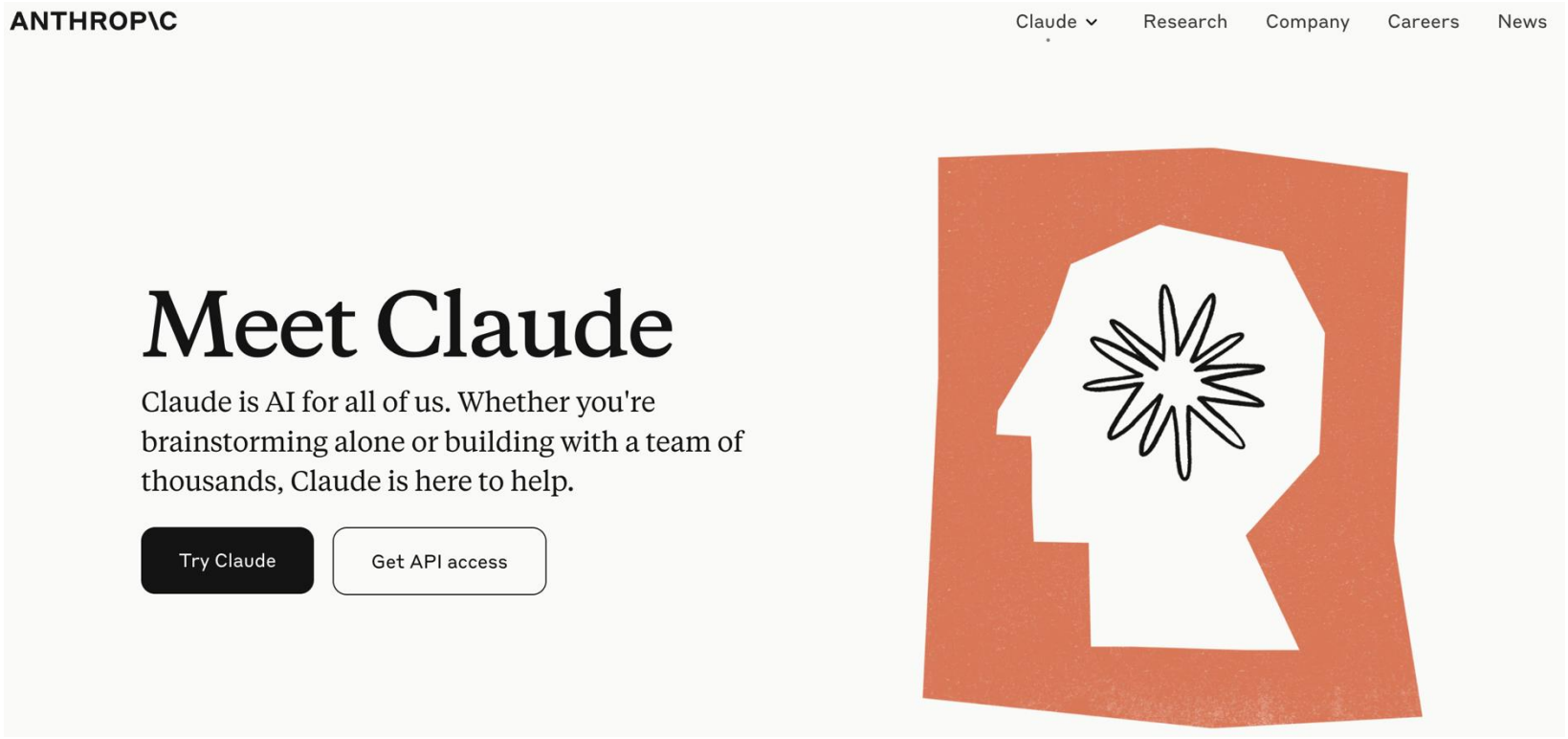
Artificial Intelligence in Context of Human History

Text-to-Video Generation (2023.03.)

Artificial Intelligence in Context of Human History

Text-to-Video Generation (2023.03.)

Artificial Intelligence in Context of Human History



Claude by Anthropic (2023.04.)

Artificial Intelligence in Context of Human History

Stable Diffusion XL (2023.07)

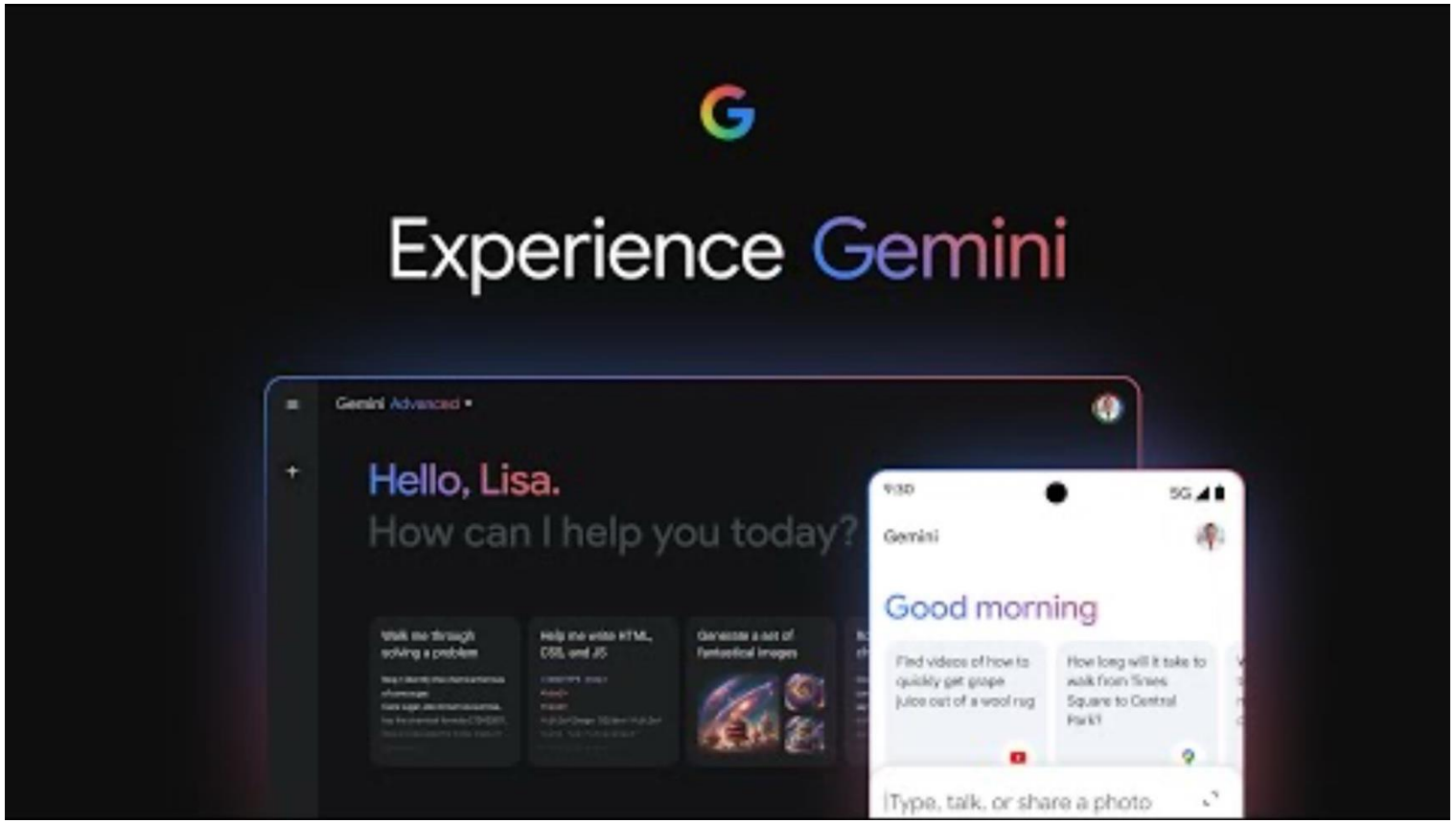
Artificial Intelligence in Context of Human History

Stable Diffusion Video (2023.11)

Artificial Intelligence in Context of Human History

DALLE-3 (2023.12)

Artificial Intelligence in Context of Human History

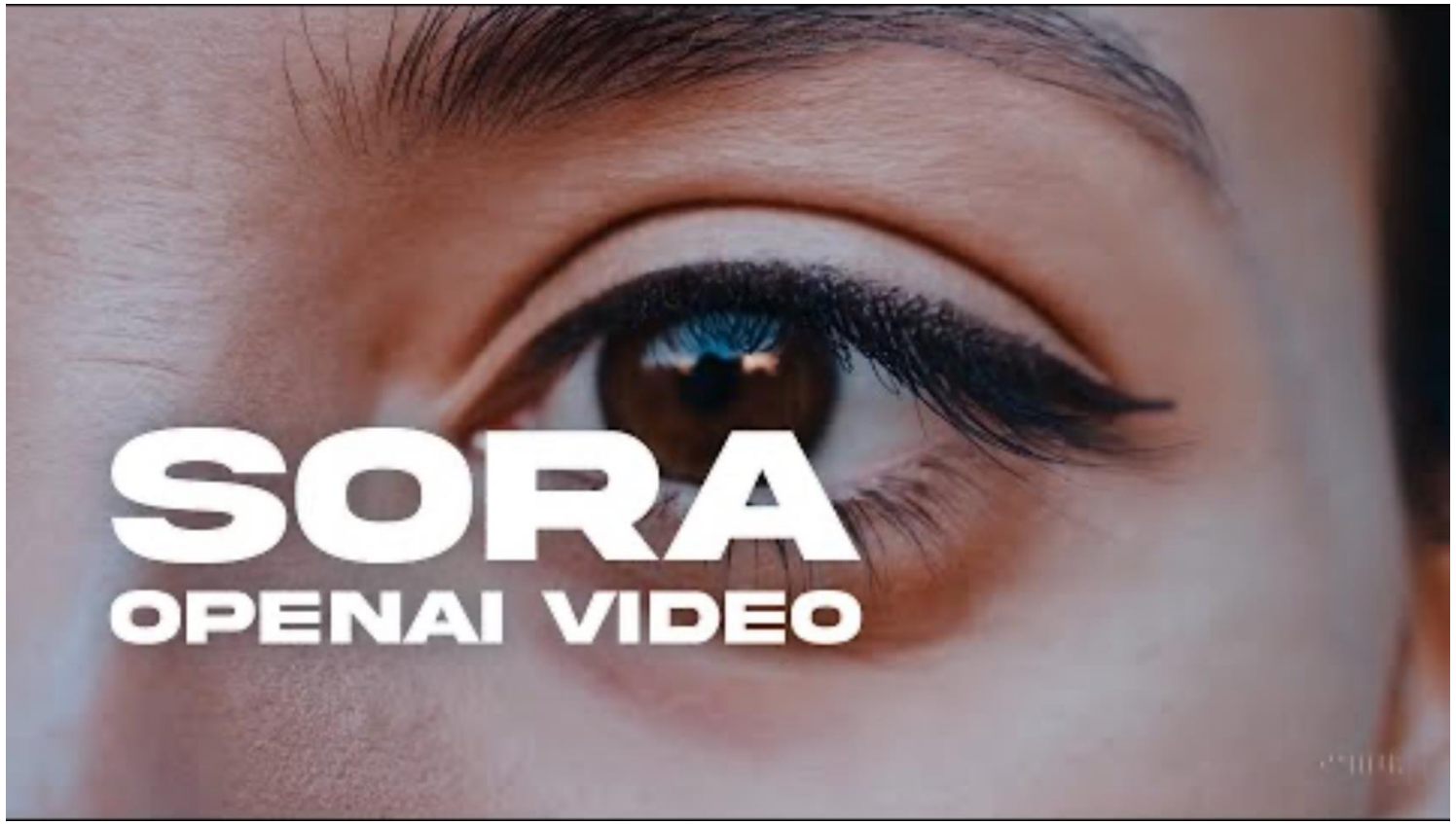


Gemini (2023.12)

Artificial Intelligence in Context of Human History

Lumiere (2024.01.23)

Artificial Intelligence in Context of Human History



SORA (2024.02.16)

https://openai.com/research/video-generation-models-as-world-simulators

Artificial Intelligence in Context of Human History

Stable Diffusion 3 (2024.02.22)

https://stability.ai/news/stable-diffusion-3

Artificial Intelligence in Context of Human History

GPT-4o (2024.05.13)

https://openai.com/index/hello-gpt-4o/

Artificial Intelligence in Context of Human History

GEN-3 (2024.06.17)

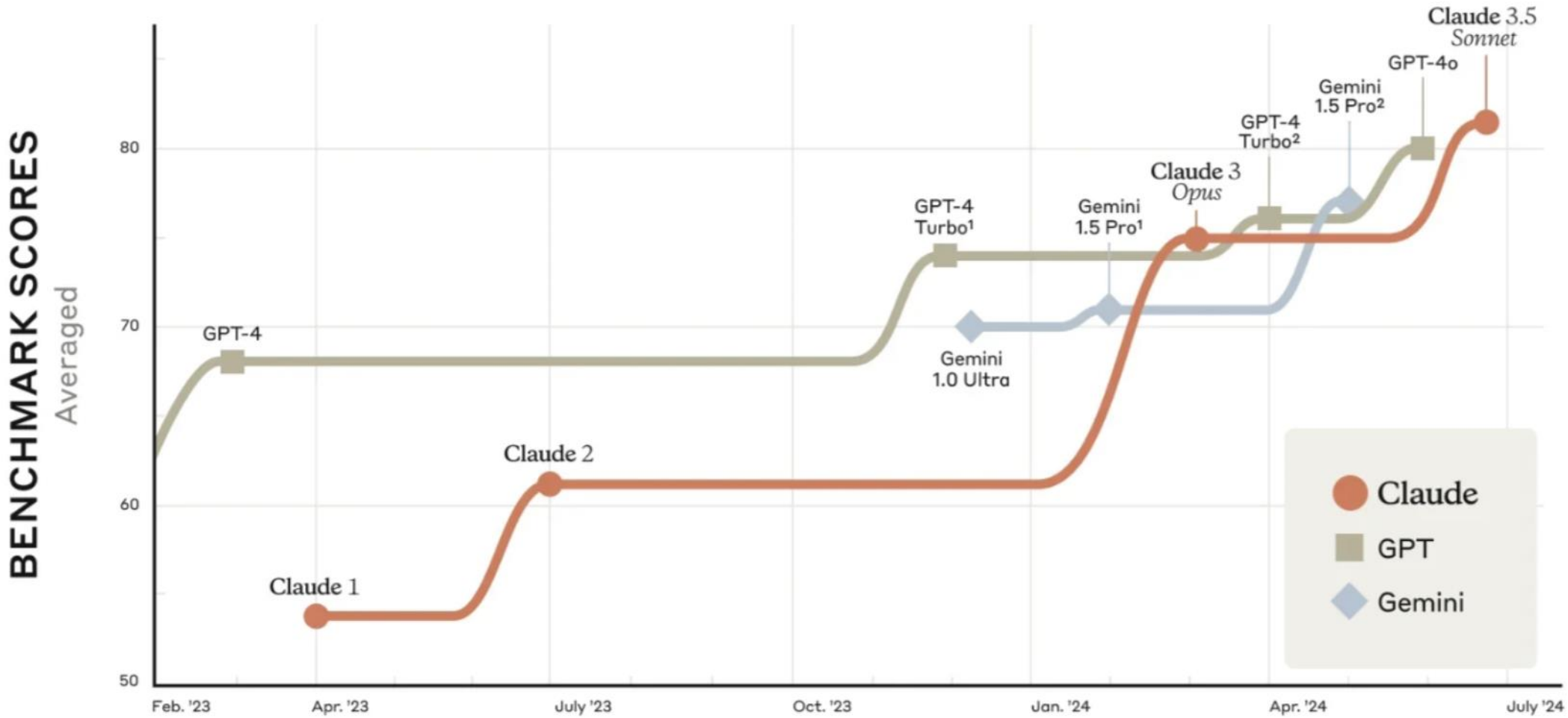
https://runwayml.com/blog/introducing-gen-3-alpha/

Artificial Intelligence in Context of Human History

FLUX (2024.08.01)

https://blackforestlabs.ai/announcing-black-forest-labs/

AI model release and capabilities timeline



Averaged benchmarks are highest reported without best-of-N: MMLU, GPQA, MATH, MGSM, DROP F1, HumanEval pass@1, MMMU, AI2D, ChartQA, DocQA, Mathvista

Source: Publicly available data; evaluation scores are the average of representative scores found online.
1 = Initial release; 2 = Second release

History of Deep Learning Ideas and Milestones

- 1943: Neural networks
- 1957-62: Perceptron
- 1970-86: Backpropagation, RBM, RNN
- 1979-98: CNN, MNIST, LSTM, Bidirectional RNN
- 2006: “Deep Learning”, DBN
- 2009: ImageNet
- 2012: AlexNet
- 2014: GANs
- 2016: ResNet
- 2016-17: AlphaGo, AlphaZero
- 2017: Transformers
- 2020: Diffusion model
- 2021: Vision Transformer
- 2022: chatGPT

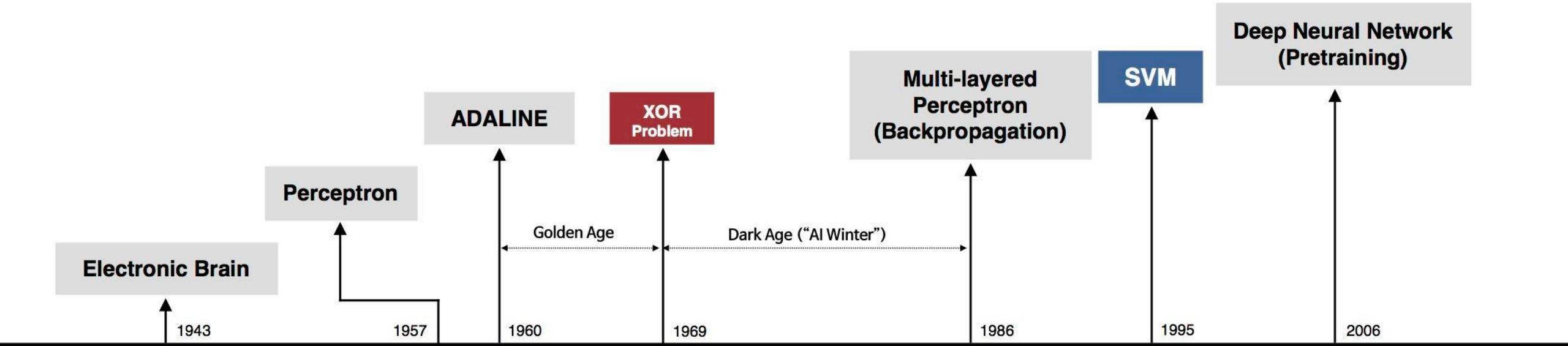
* Dates are for perspective and not as definitive historical record of invention or credit

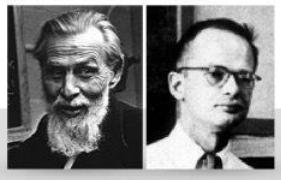
History of Deep Learning Ideas and Milestones

- 1943: Neural networks
- 1957-62: Perceptron
- 1970-86: Backpropagation, RBM, RNN
- 1979-98: CNN, MNIST, LSTM, Bidirectional RNN
- 2006: “Deep Learning”, DBN
- 2009: ImageNet
- 2012: **AlexNet**
- 2014: GANs
- 2016: **ResNet**
- 2016-17: AlphaGo, AlphaZero
- 2017: **Transformers**
- 2020: **Diffusion model**
- 2021: **Vision Transformer**
- 2022: **chatGPT**

* Dates are for perspective and not as definitive historical record of invention or credit

History of Deep Learning Ideas and Milestones





S. McCulloch - W. Pitts

F. Rosenblatt

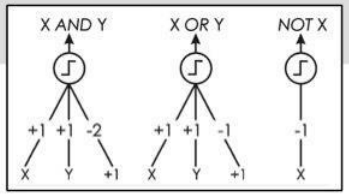
B. Widrow - M. Hoff

M. Minsky - S. Papert

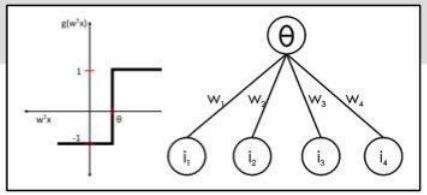
D. Rumelhart - G. Hinton - R. Wiliams

V. Vapnik - C. Cortes

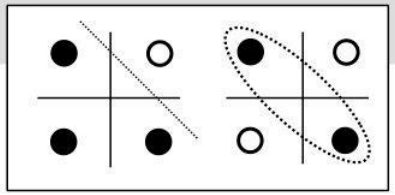
G. Hinton - S. Ruslan



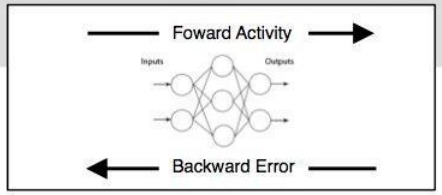
- Adjustable Weights
- Weights are not Learned



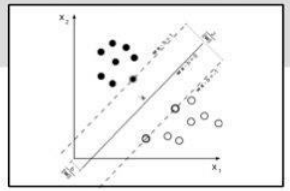
- Learnable Weights and Threshold



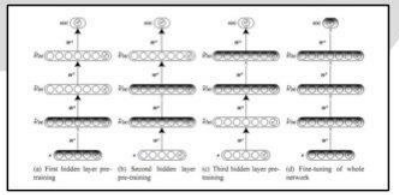
- XOR Problem



- Solution to nonlinearly separable problems
- Big computation, local optima and overfitting

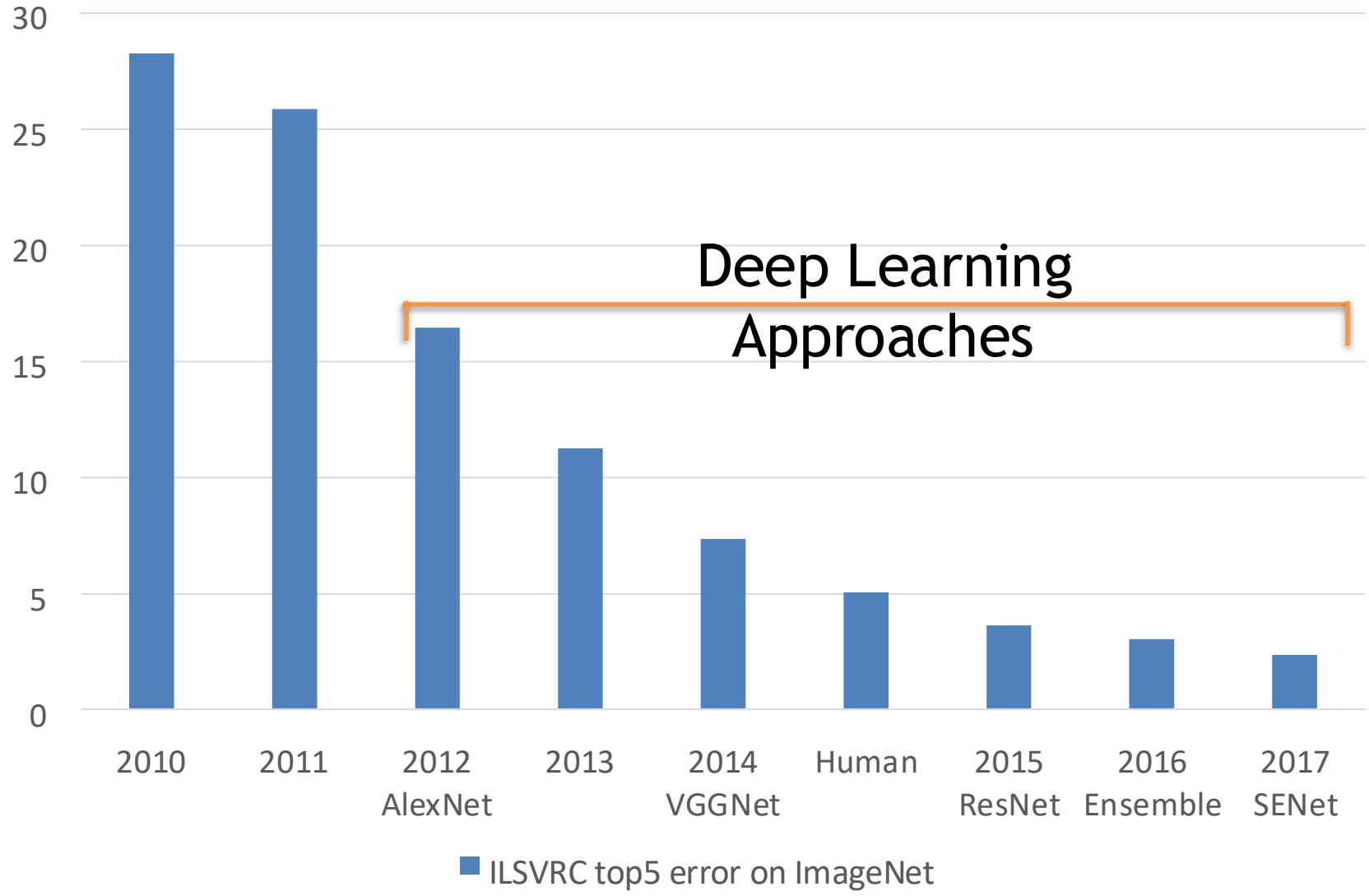


- Limitations of learning prior knowledge
- Kernel function: Human Intervention



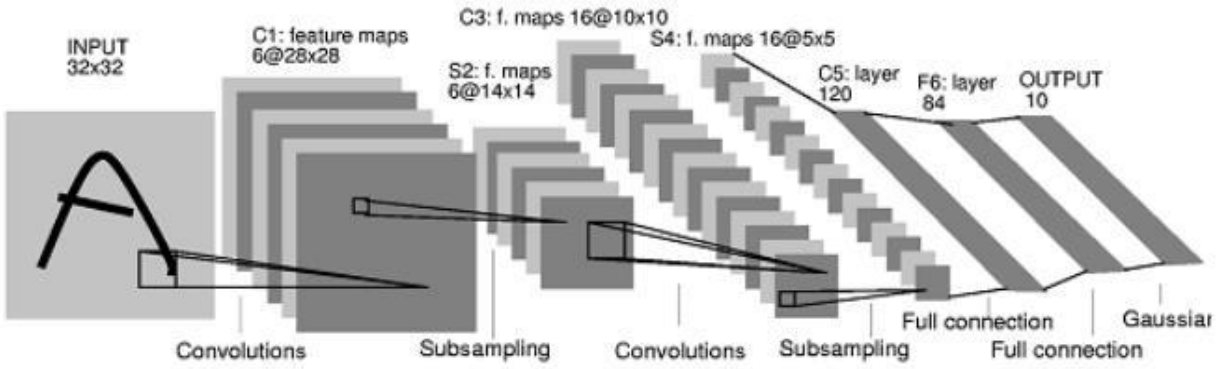
- Hierarchical feature Learning

Breakthrough



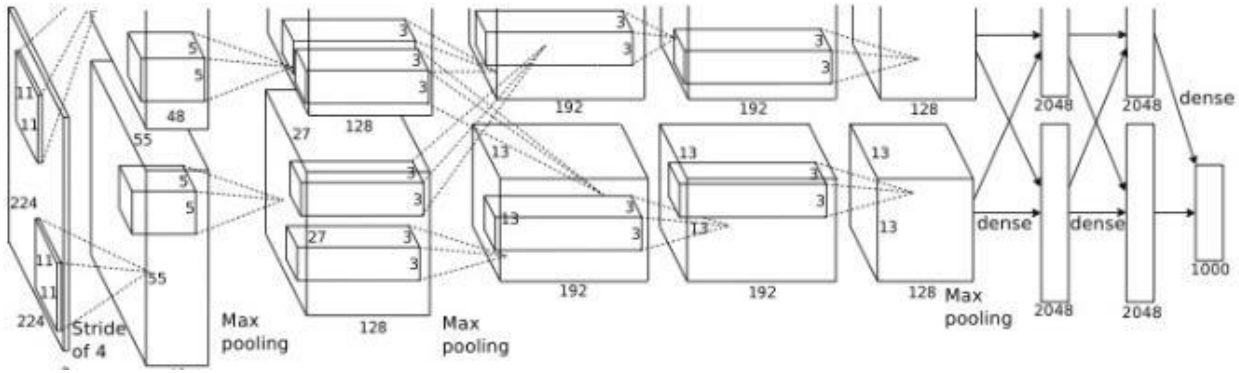
What has changed?

1998
LeCun
et al.



- MNIST digit recognition dataset
- 10^7 pixels used in training

2012
Krizhevsky
et al.



- ImageNet image recognition dataset
- 10^{14} pixels used in training

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

Turing Award for Deep Learning

- Yann LeCun
- Geoffrey Hinton
- Yoshua Bengio

Turing Award given for:

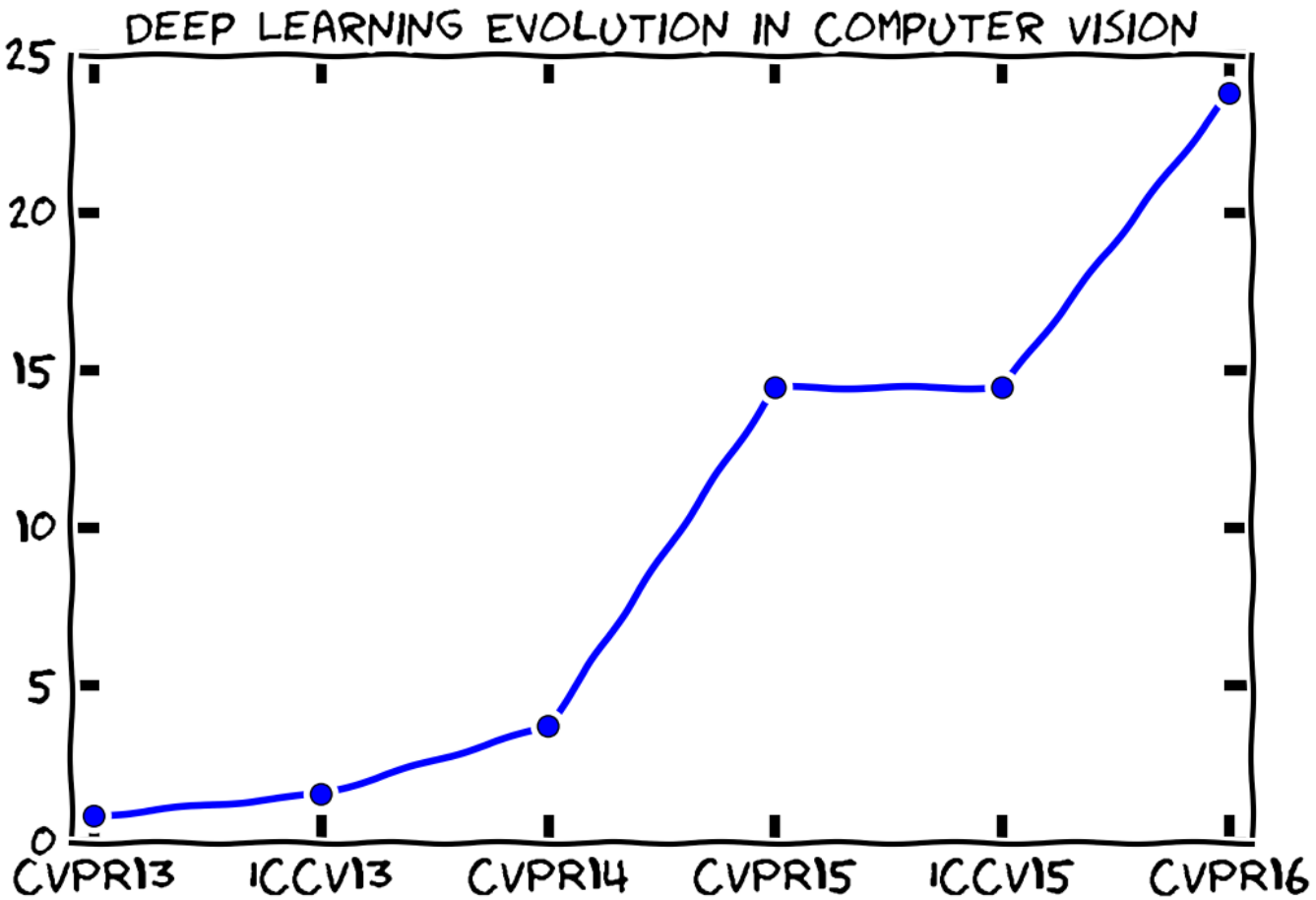
- “The conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing.”
- (Also, for popularization in the face of skepticism.)

Google Scholar Rank (h-index)

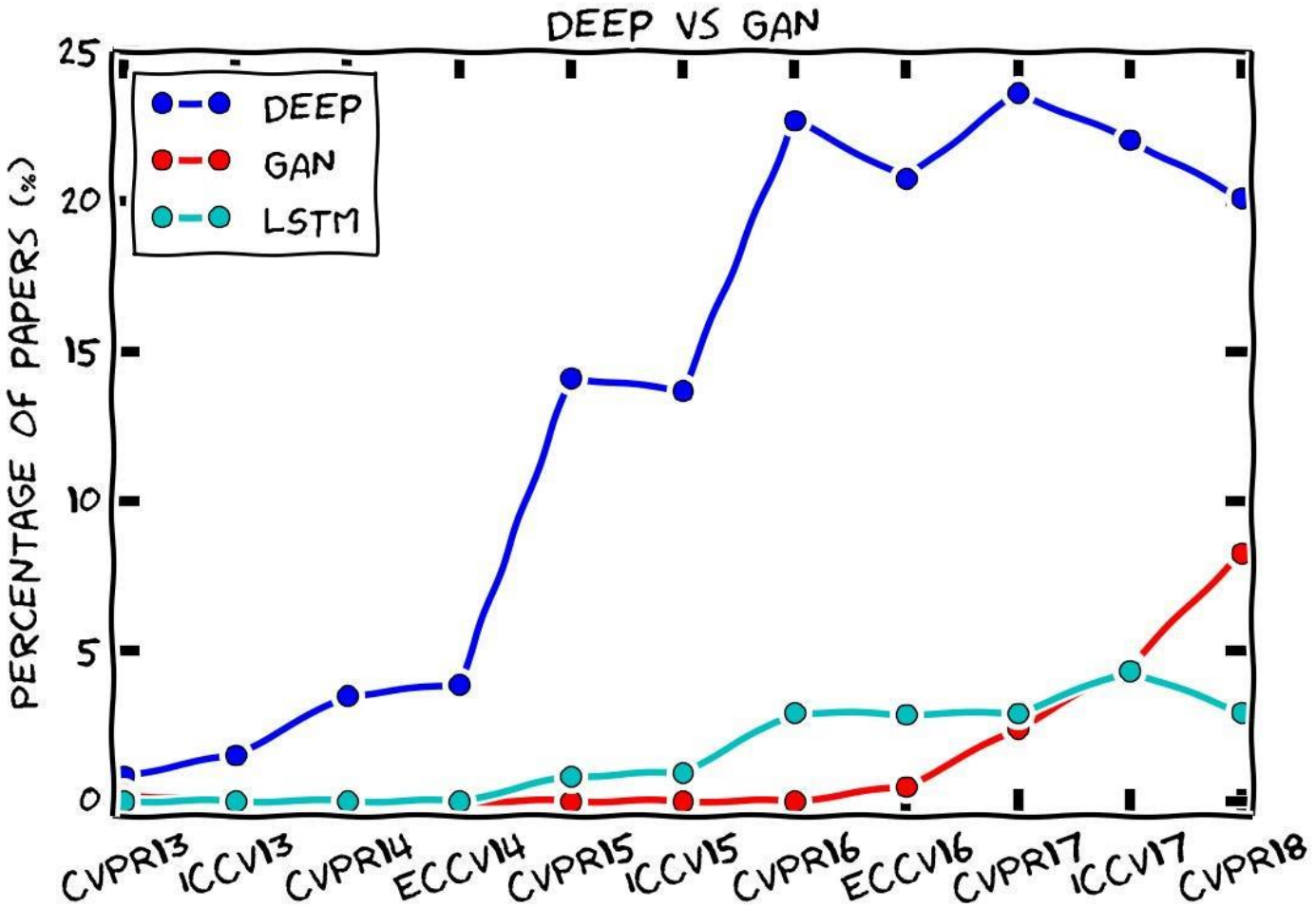
Categories ▾ English ▾

	Publication	h5-index	h5-median
1.	Nature	488	745
2.	IEEE/CVF Conference on Computer Vision and Pattern Recognition **CVPR**	440	689
3.	The New England Journal of Medicine	434	897
4.	Science	409	633
5.	Nature Communications	375	492
6.	The Lancet	368	678
7.	Neural Information Processing Systems **NeurIPS**	337	614
8.	Advanced Materials	327	420
9.	Cell	320	482
10.	International Conference on Learning Representations **ICLR**	304	584

https://scholar.google.com/citations?view_op=top_venues&hl=en



Credits: Dr. Pont-Tuset, ETH Zurich



Credits: Dr. Pont-Tuset, ETH Zurich

What is Deep Learning?

Artificial Intelligence

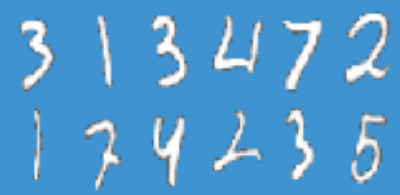
Any technique that enables computers to mimic human behavior

Machine Learning

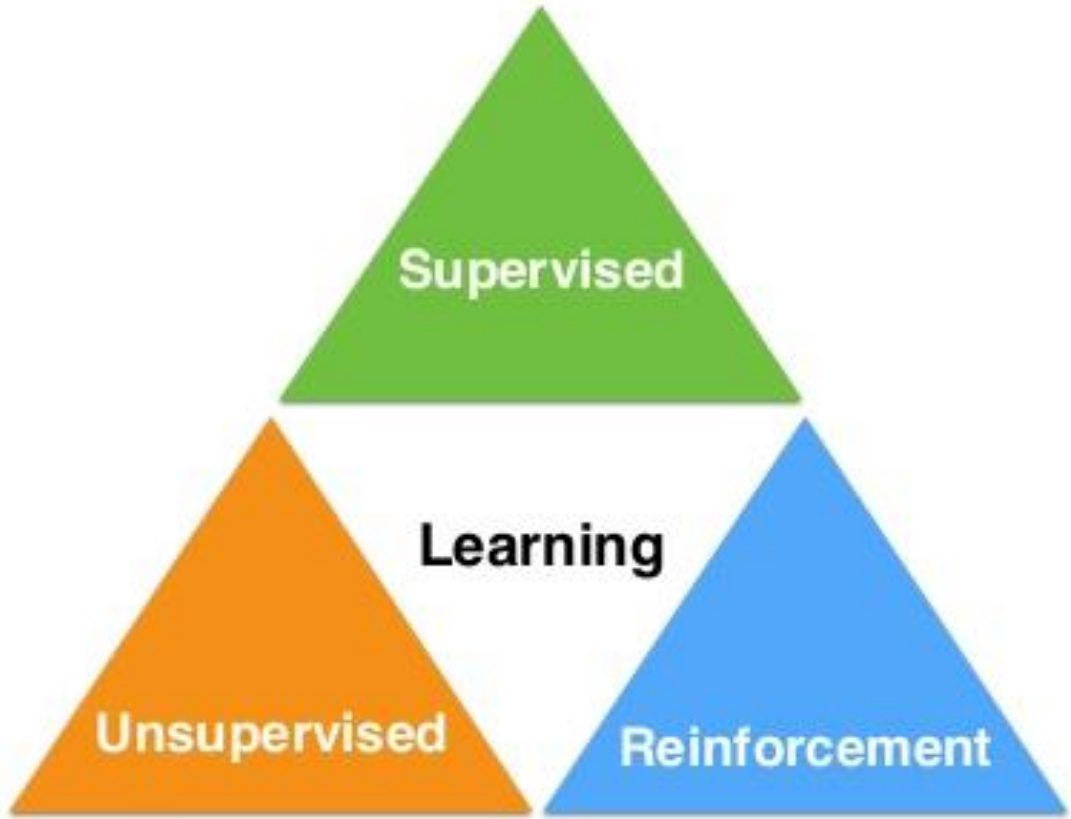
Ability to learn without explicitly being programmed

Deep Learning

Extract patterns from data using neural networks



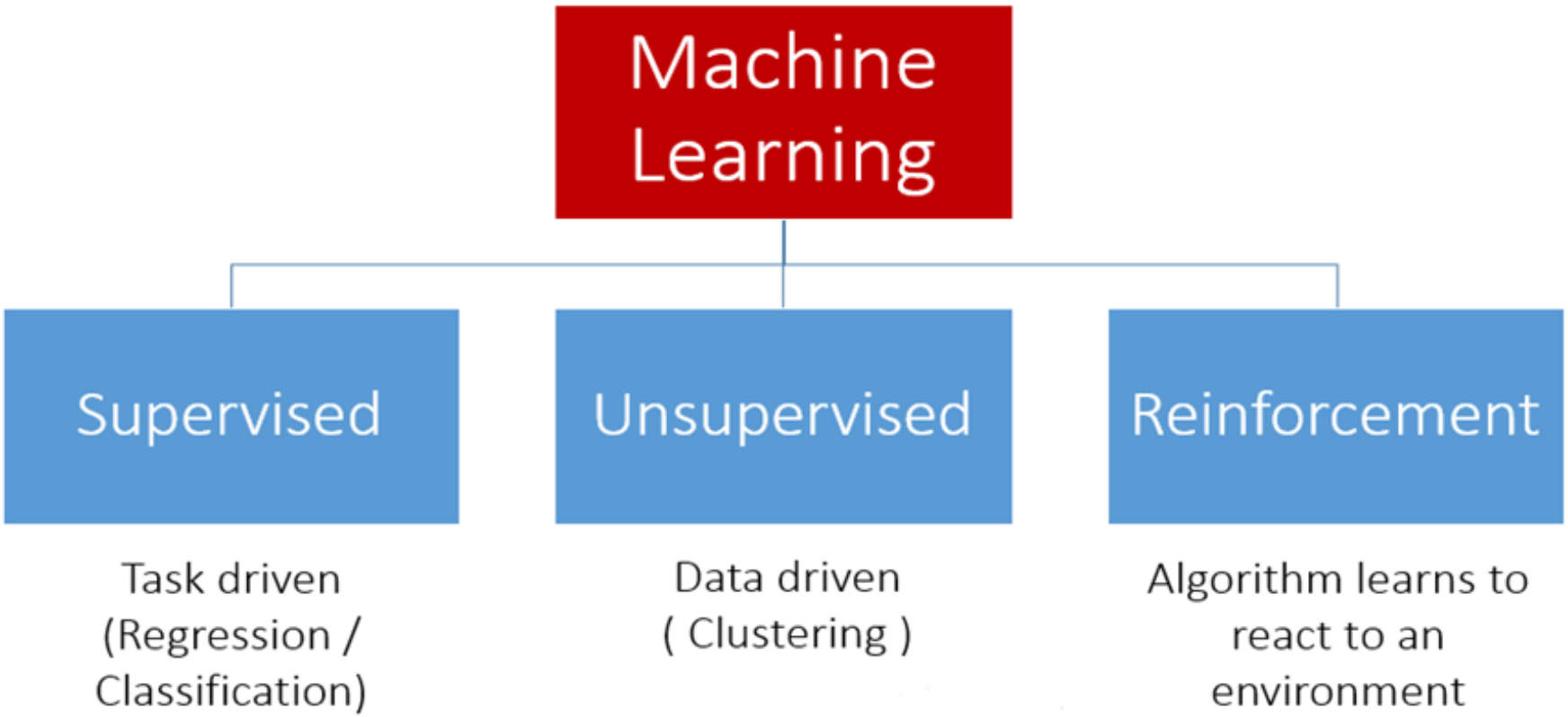
- Labeled data
- Direct feedback
- Predict outcome/future



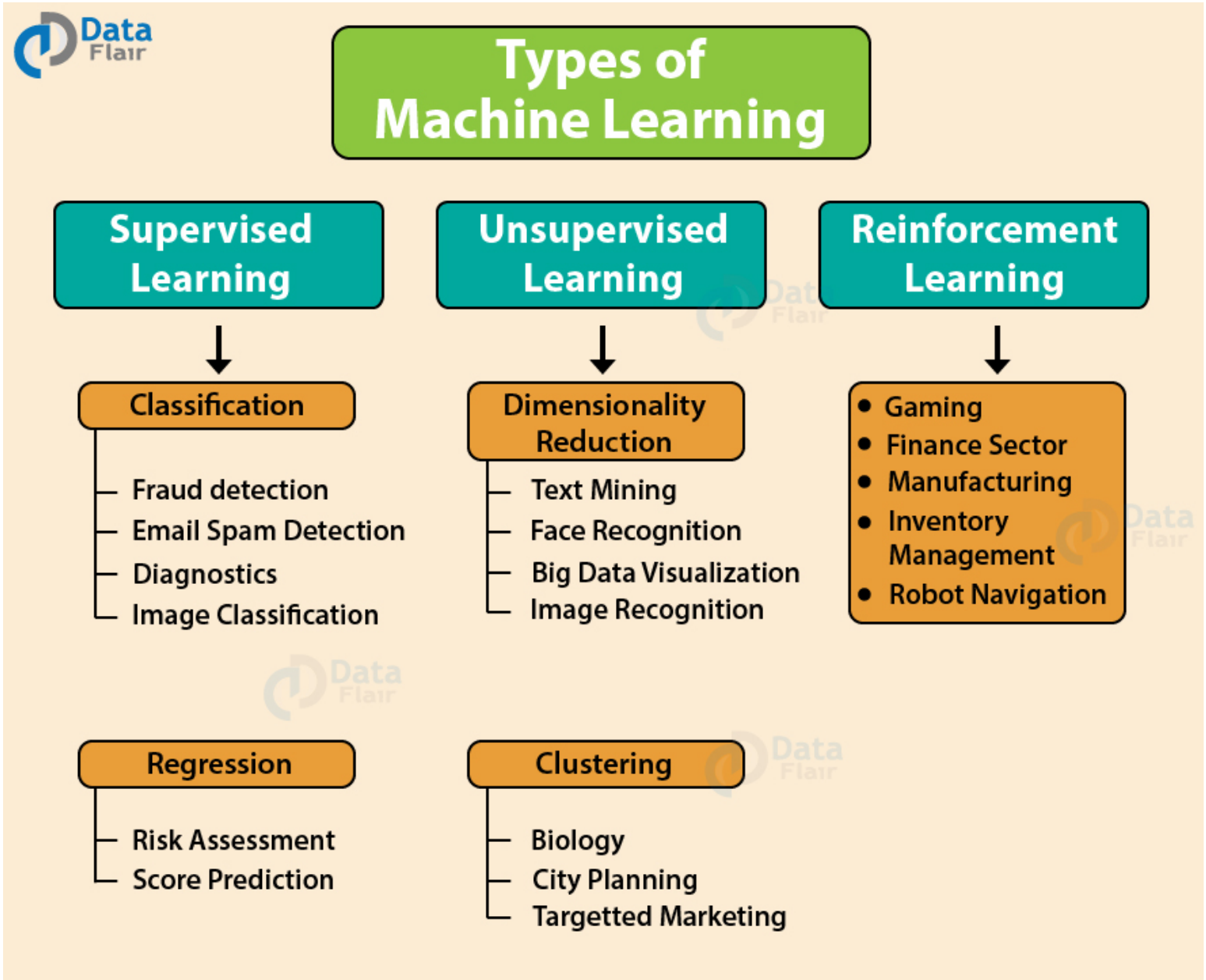
- No labels
- No feedback
- “Find hidden structure”

- Decision process
- Reward system
- Learn series of actions

Types of Machine Learning

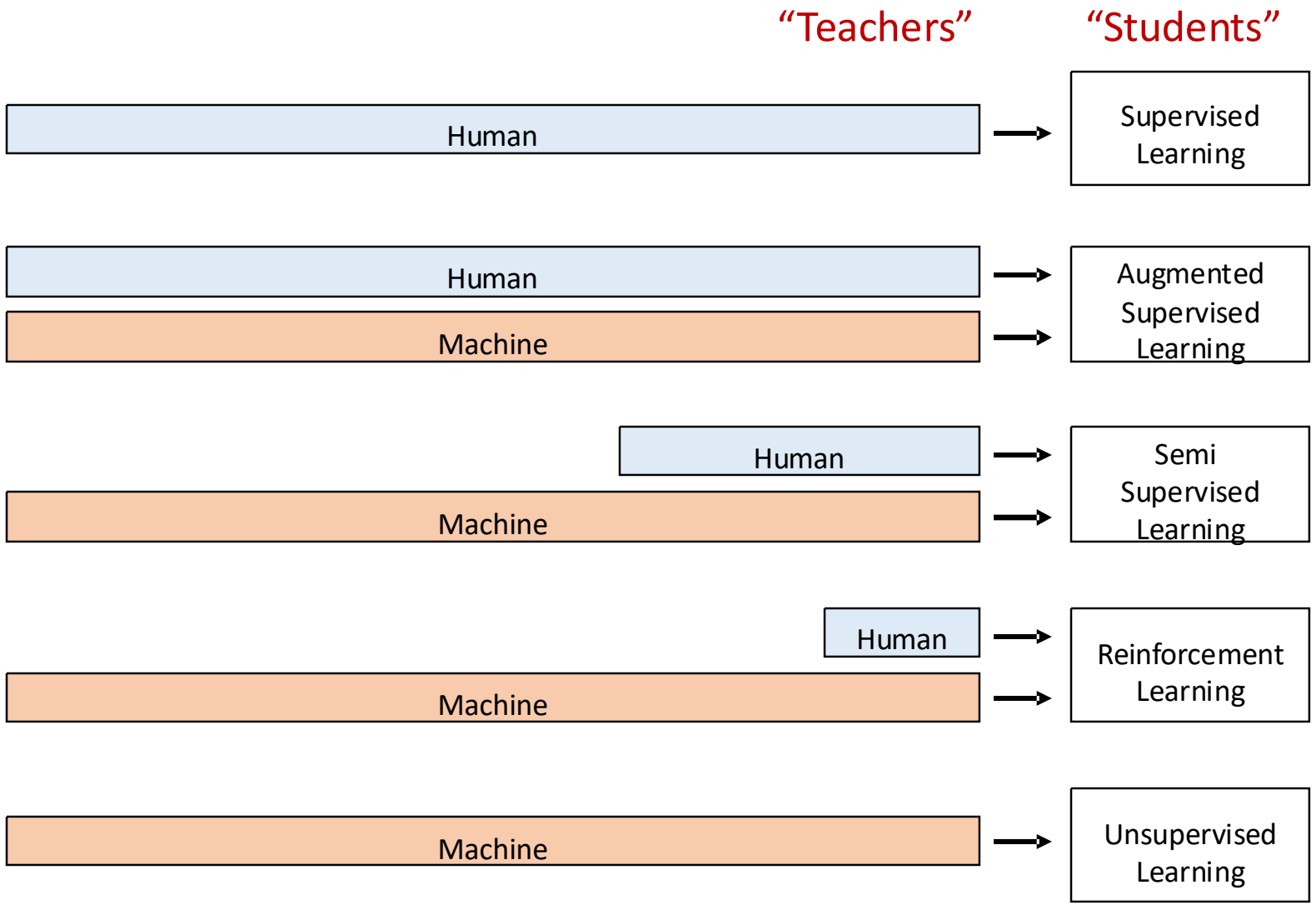


https://images.app.goo.gl/ampJkNSjUaDmMuoQ8

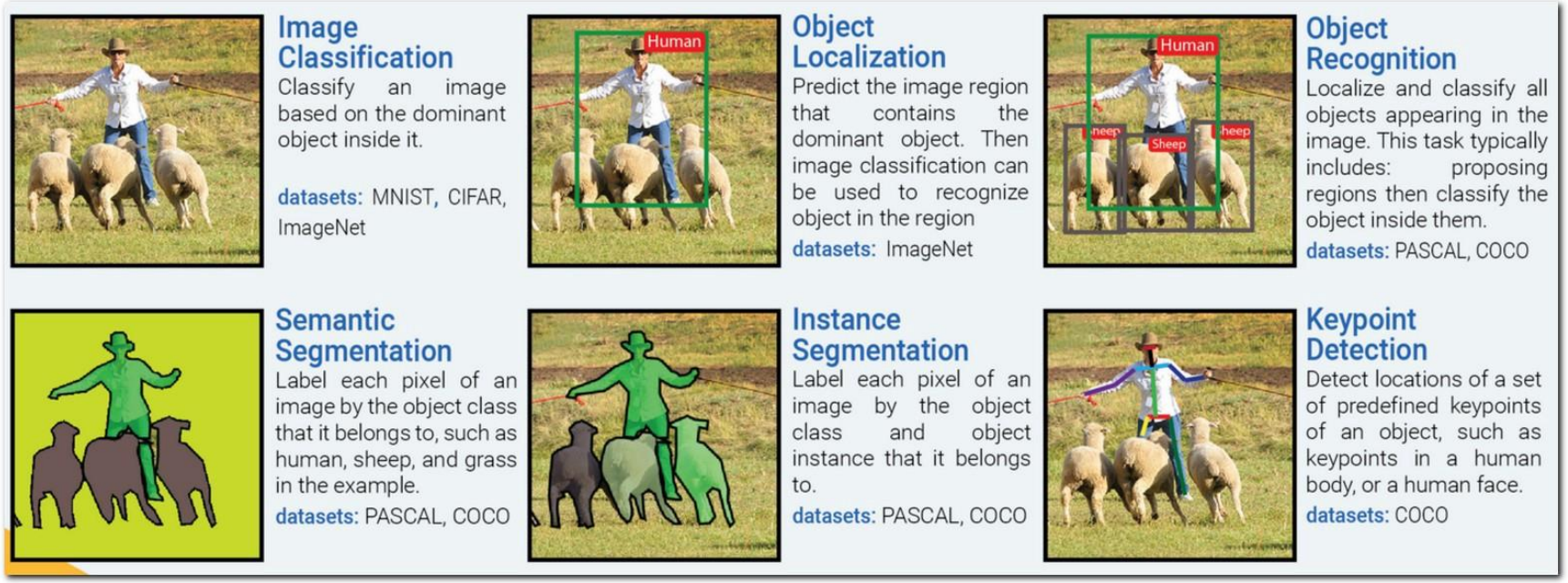


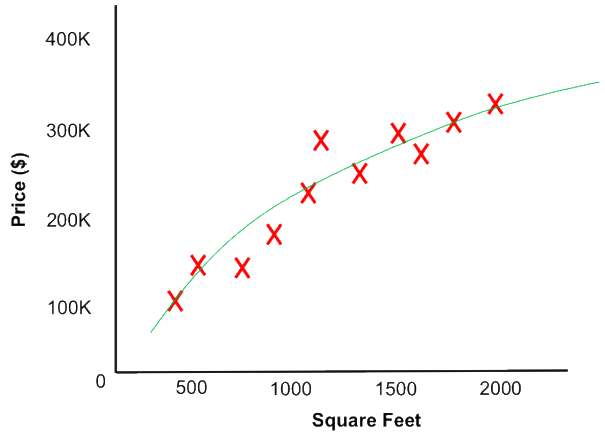
https://images.app.goo.gl/c2ipwqxdUcwsn9cY7

Deep Learning from Human and Machine



Supervised Learning





Regression

"man in black shirt is playing guitar."

Image captioning

Unsupervised Learning

GAN

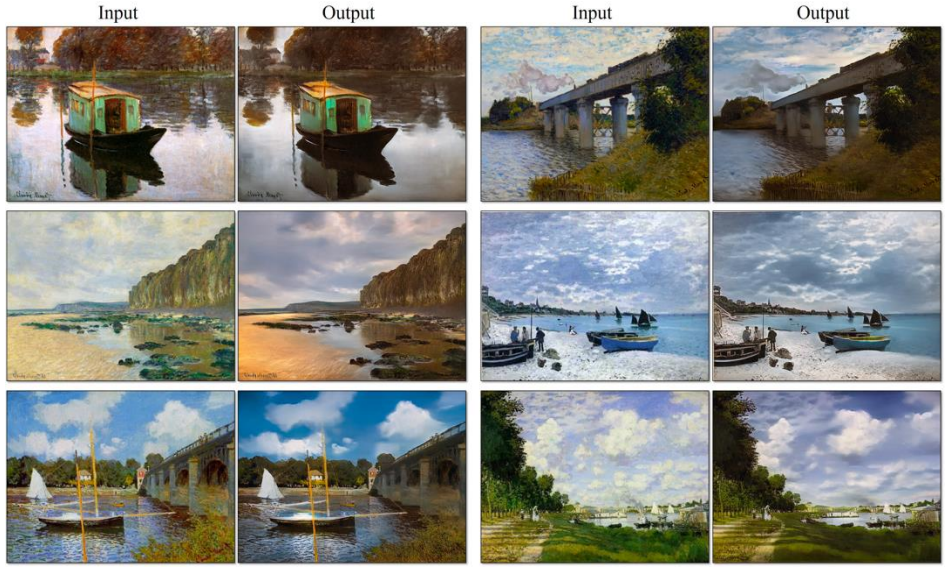
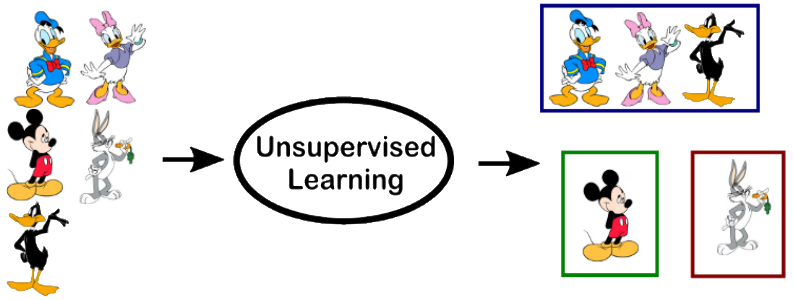


Image-to-image translation

Clustering



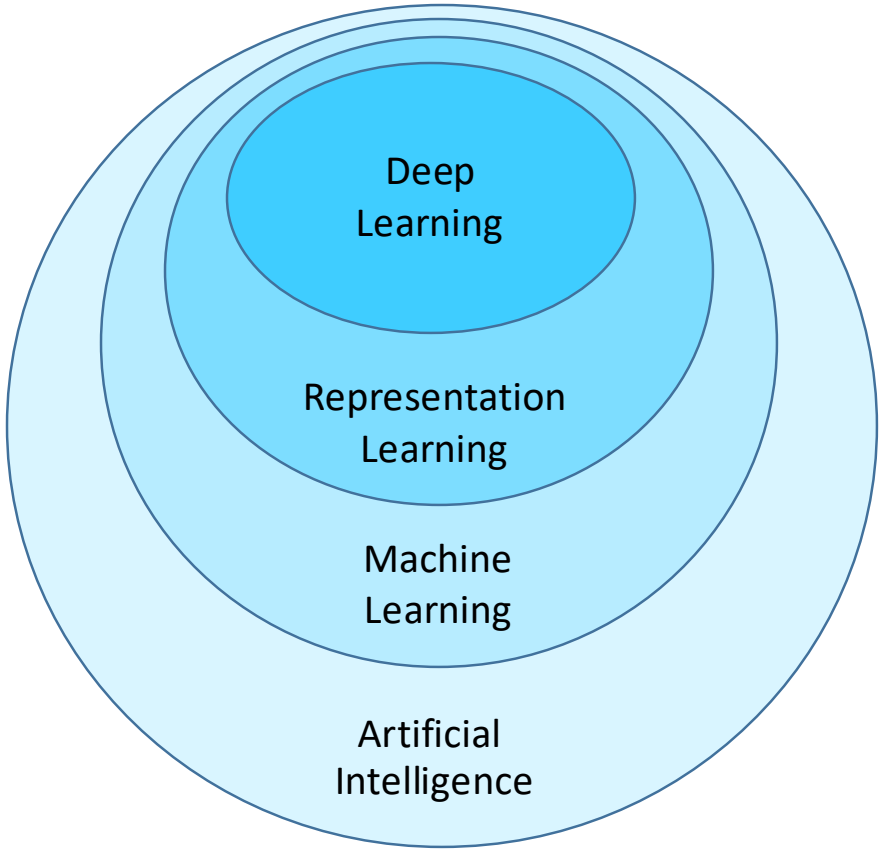
Reinforcement Learning

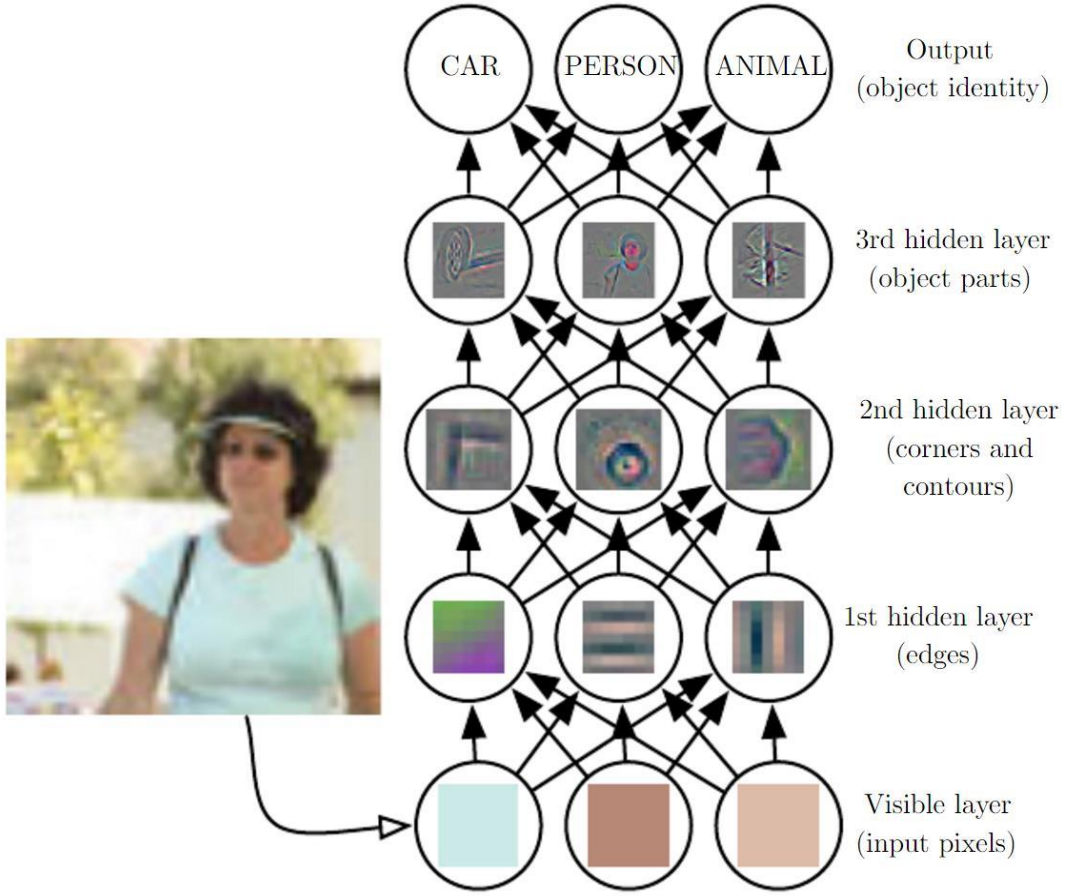
(a) 슈퍼마리오

(b) Atari - 벽돌깨기

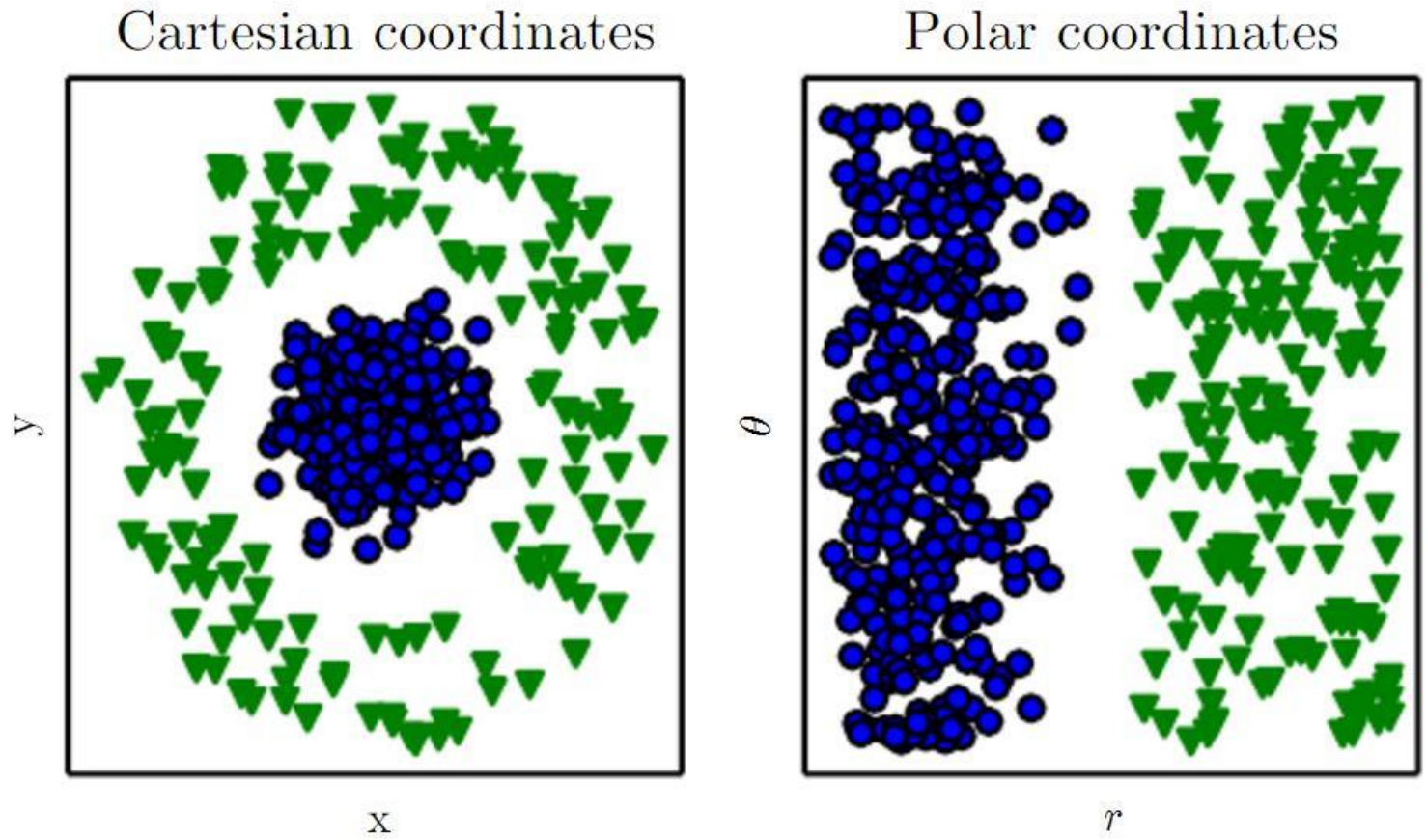
AlphaGo

Deep Learning is Representation Learning
(aka Feature Learning)



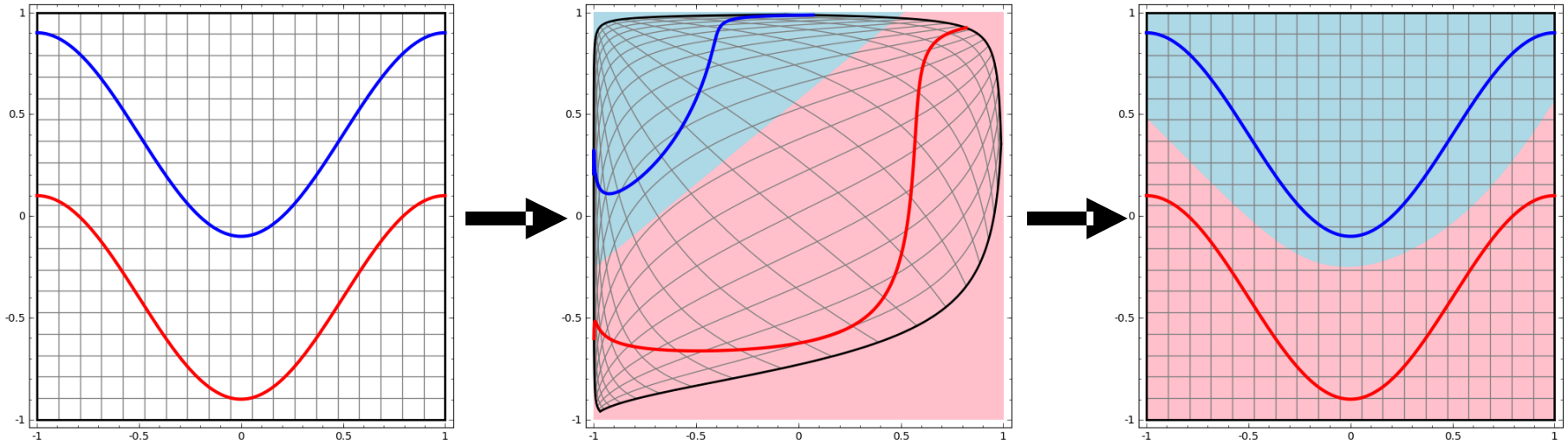


Representation Matters



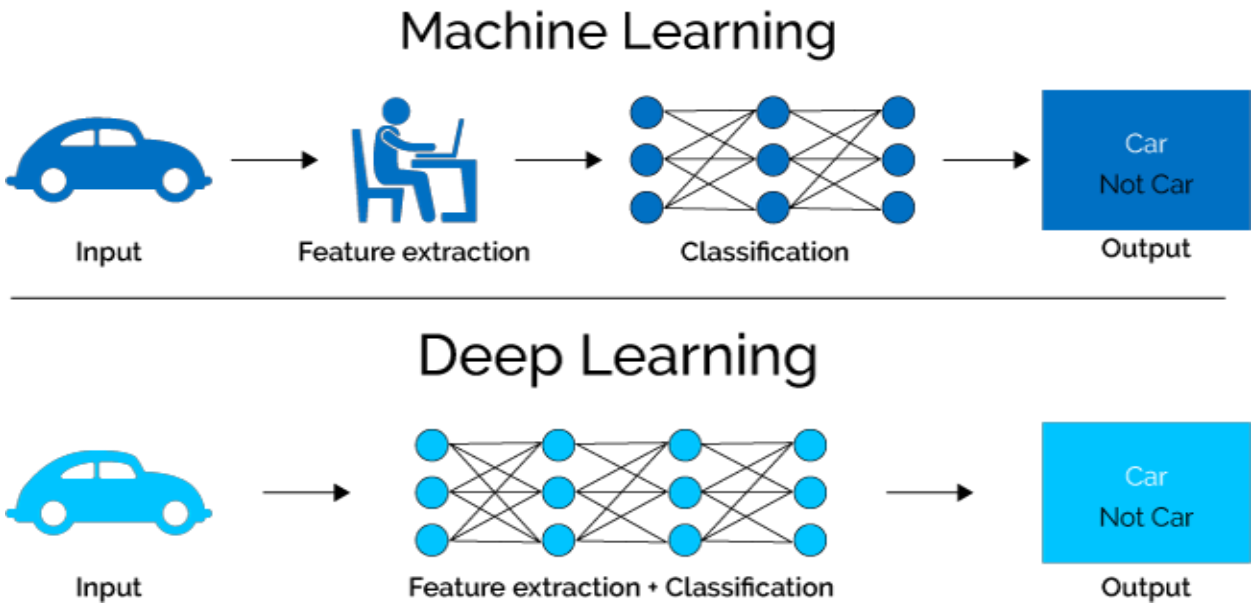
Task: Draw a line to separate the green triangles and blue circles.

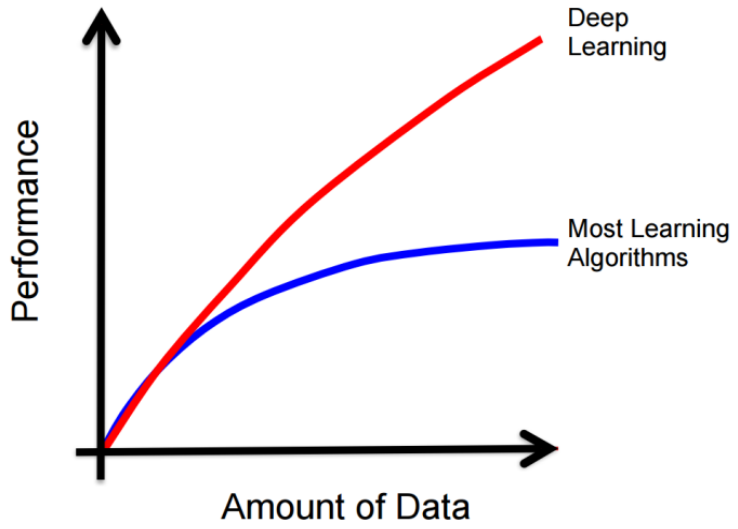
Deep Learning is Representation Learning
(aka Feature Learning)



Task: Draw a line to separate the blue curve and red curve

Why Deep Learning? Scalable Machine Learning

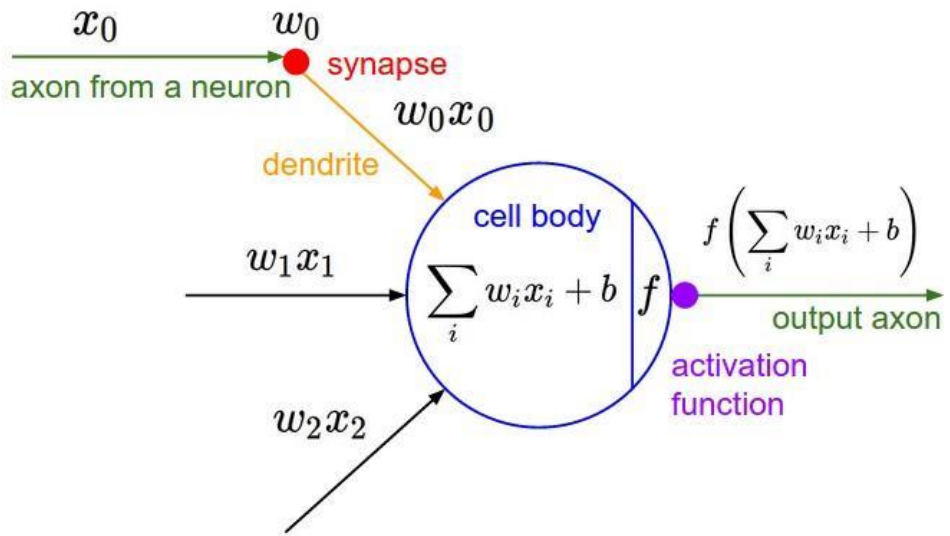




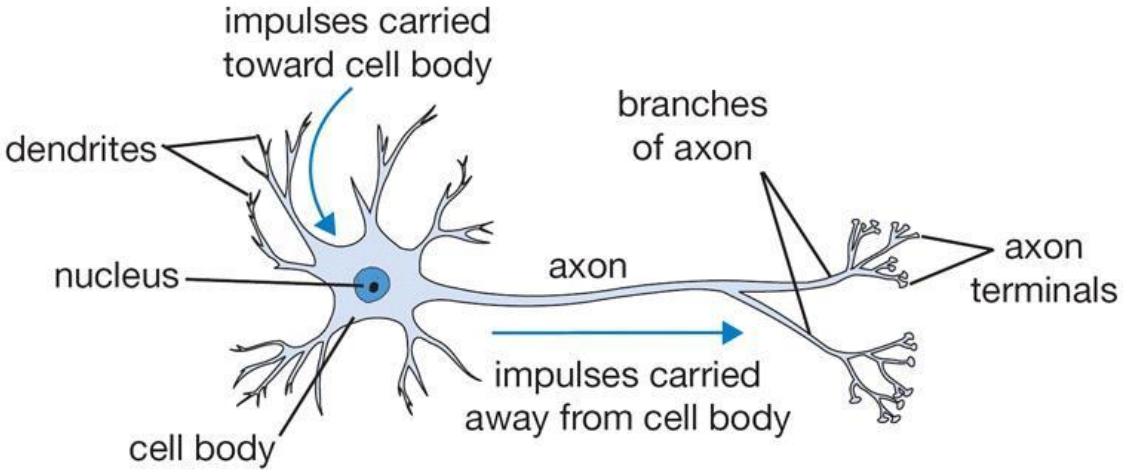
What is Neural Network?

From perceptron to structural build block of deep learning

Neuron: Biological Inspiration for Computation

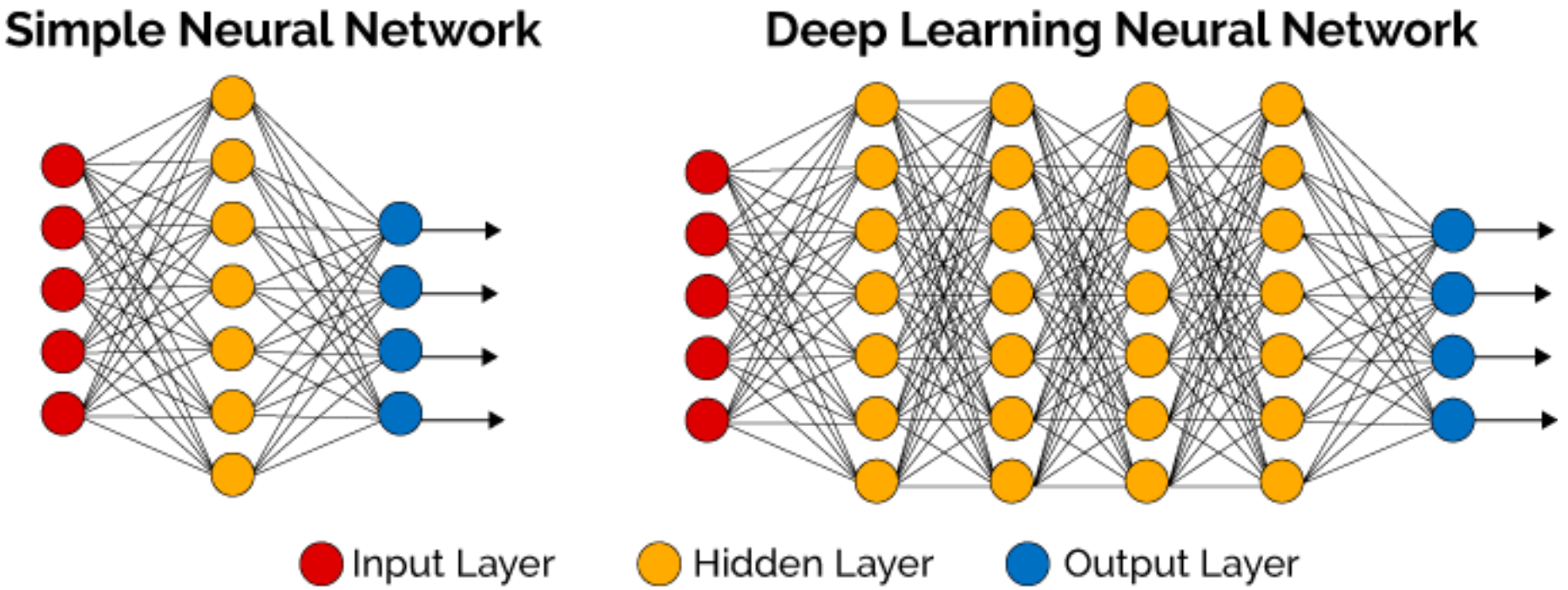


(Artificial) Neuron:
computational building block for the "neural network"



Neuron:
computational building block for the brain

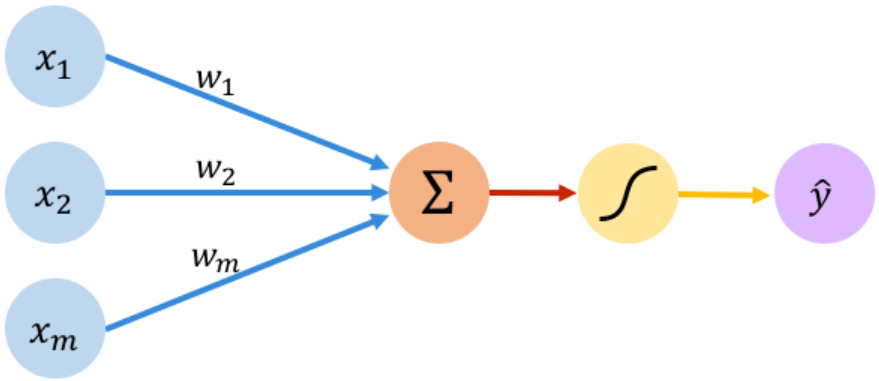
Combing Neurons in Hidden Layers:
The “Emergent” Power to Approximate



Universal Approximation Theorem:
For any arbitrary function f(x),
there exists a neural network that closely approximate it for any input x

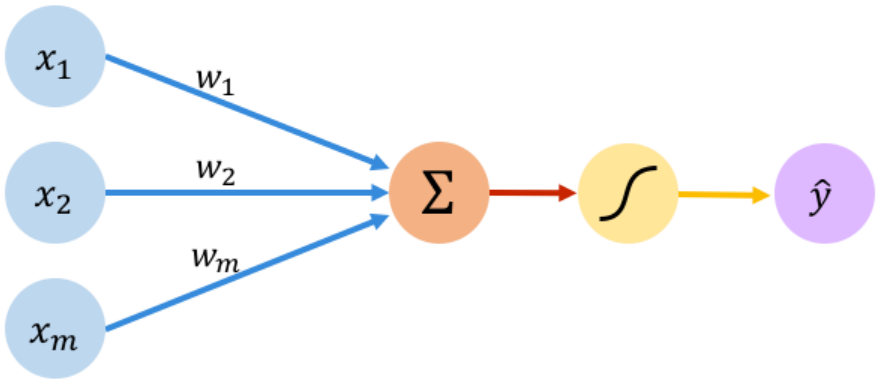
Csáji, Balázs Csanád. "Approximation with artificial neural networks." Faculty of Sciences, Etvs Lornd University, Hungary 24, no. 48 (2001): 7.

The Perceptron: Forward Propagation



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

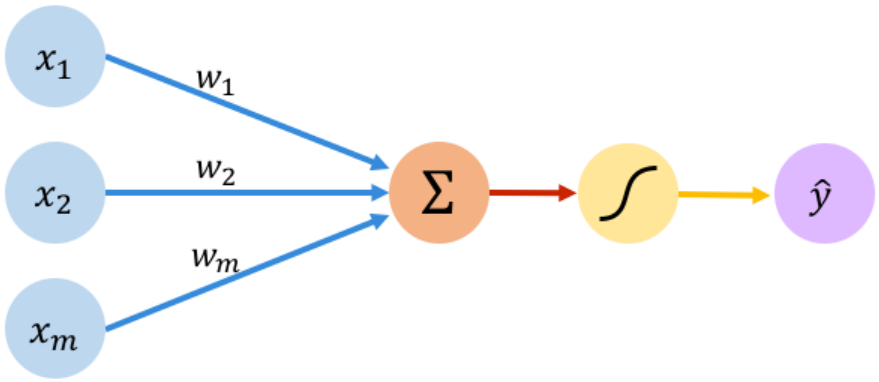
The Perceptron: Forward Propagation



Linear combination of inputs

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

The Perceptron: Forward Propagation



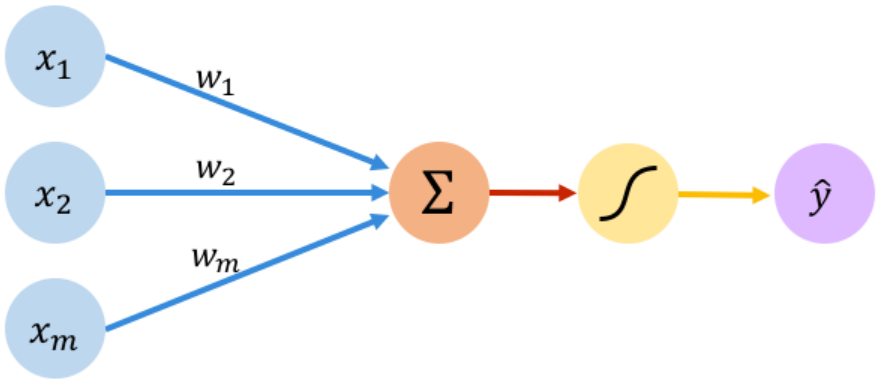
Inputs Weights Sum Non-Linearity Output

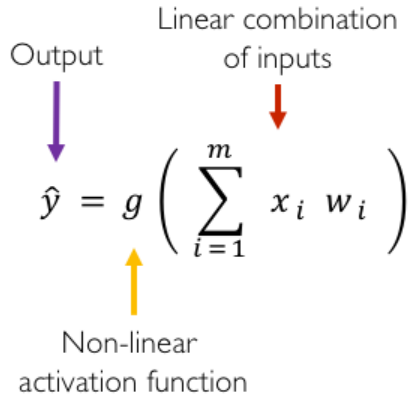
Linear combination of inputs

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

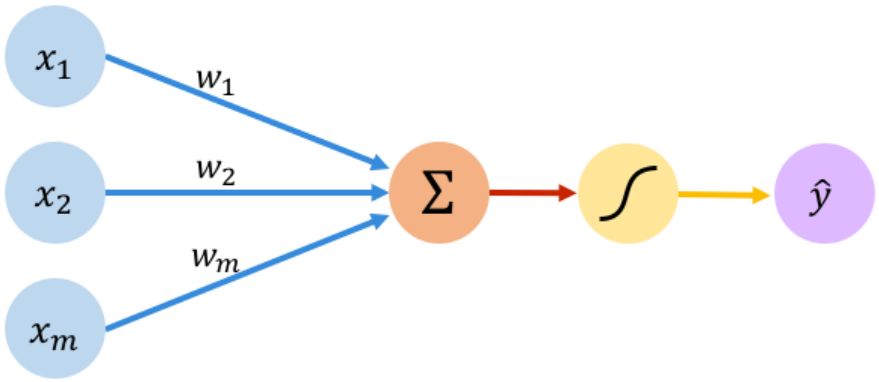
Non-linear activation function

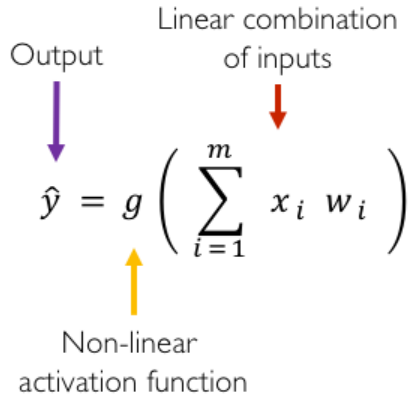
The Perceptron: Forward Propagation



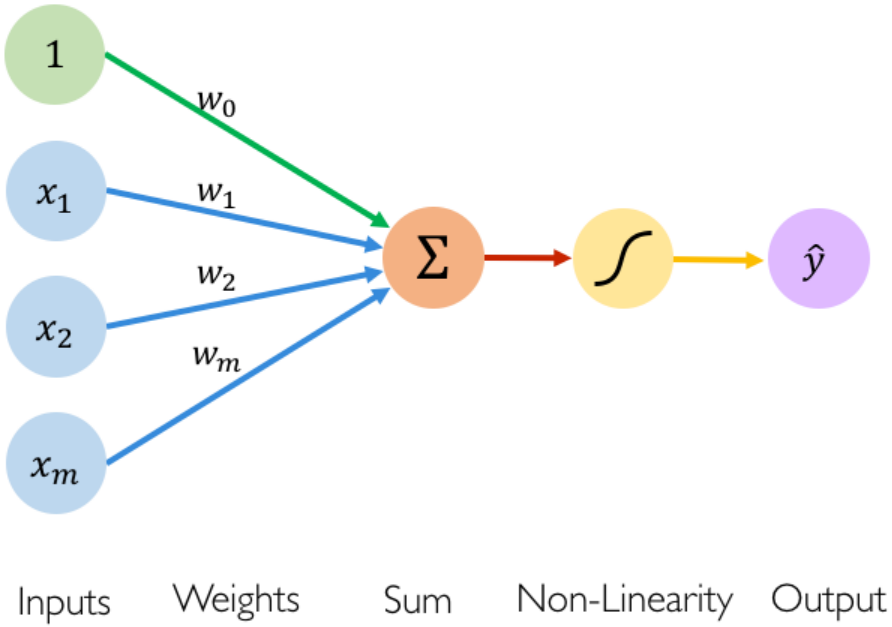


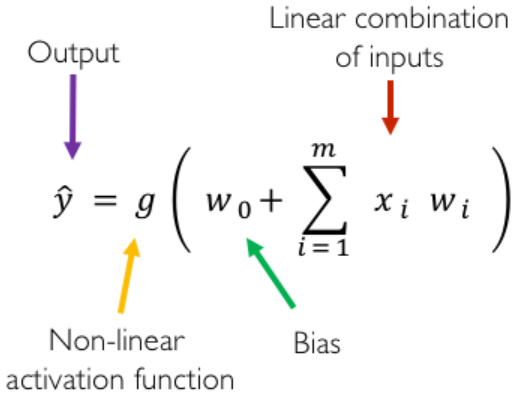
The Perceptron: Forward Propagation



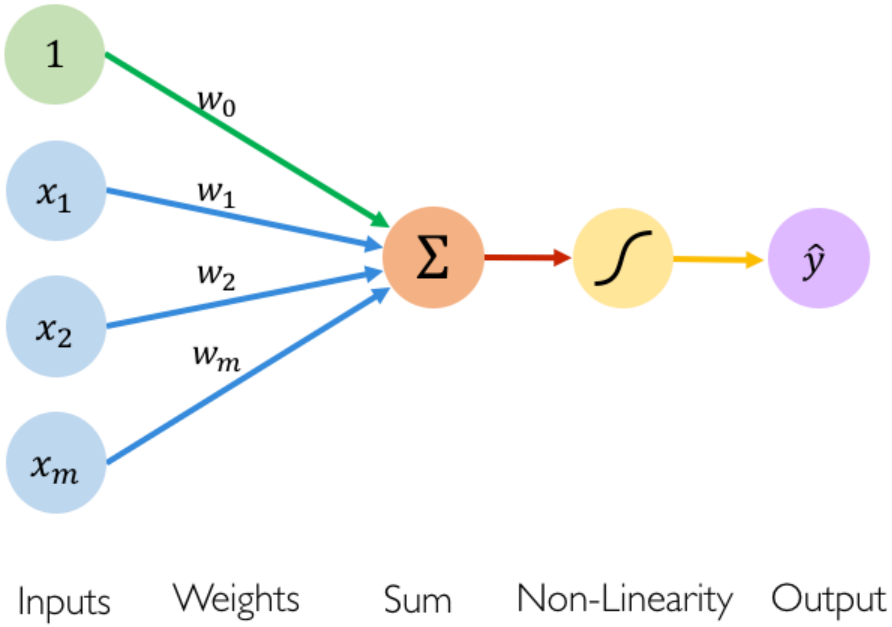


The Perceptron: Forward Propagation





The Perceptron: Forward Propagation

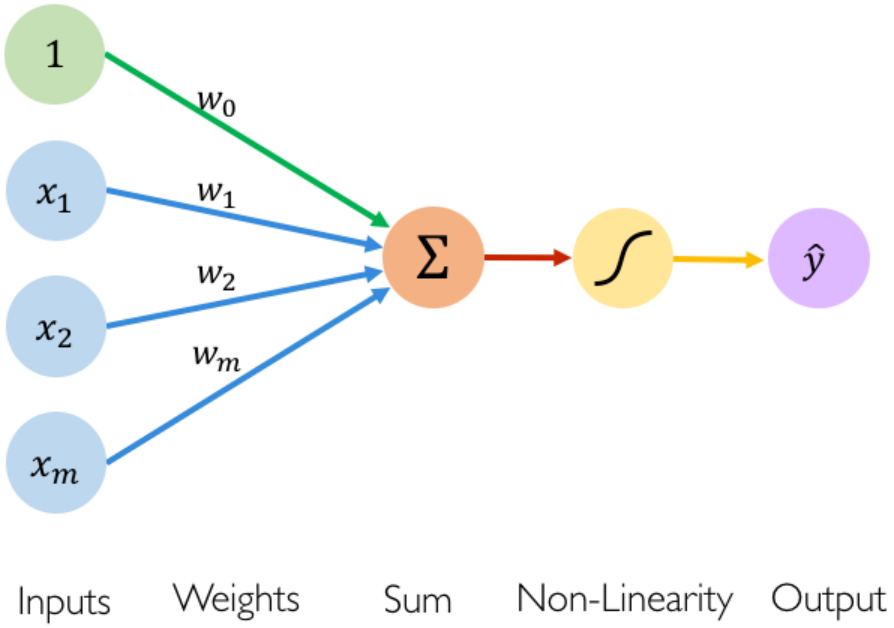


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

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

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

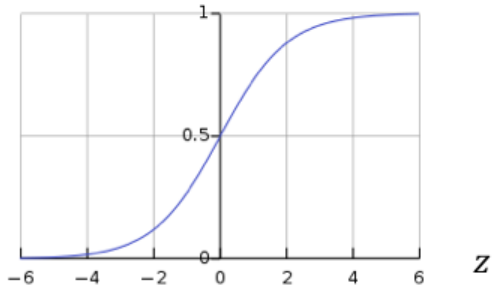


Activation Functions

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

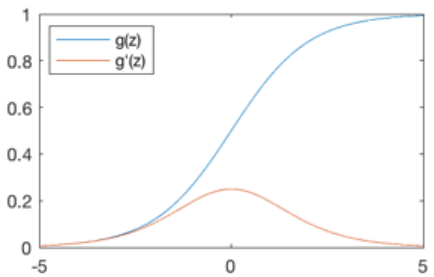
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function



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

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

SIGMOID

CLASS `torch.nn.Sigmoid`

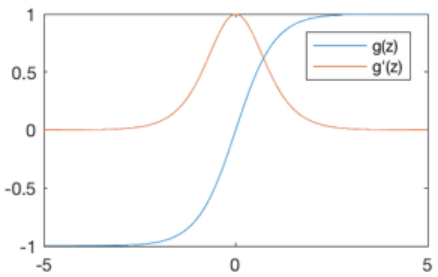
Applies the element-wise function:

`torch.nn.functional.sigmoid(input)` → Tensor

Applies the element-wise function $\text{Sigmoid}(x) = \frac{1}{1+\exp(-x)}$

See `Sigmoid` for more details.

Hyperbolic Tangent



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

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

TANH

CLASS `torch.nn.Tanh`

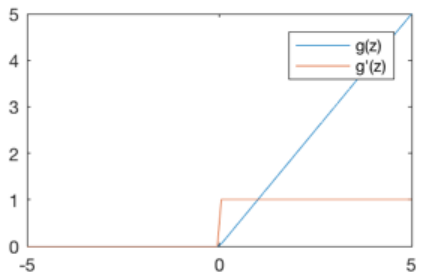
Applies the element-wise function:

`torch.nn.functional.tanh(input)` → Tensor

Applies element-wise, $\text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$

See `Tanh` for more details.

Rectified Linear Unit (ReLU)



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

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

RELU

CLASS `torch.nn.ReLU(inplace: bool = False)`

Applies the rectified linear unit function element-wise:

`torch.nn.functional.relu(input, inplace=False)` → Tensor

Applies the rectified linear unit function element-wise. See `ReLU` for more details.

https://pytorch.org/docs/stable/nn.html

https://pytorch.org/docs/stable/nn.functional.html

Dance Moves of Deep Learning Activation Functions

Sigmoid

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

Tanh

$$y = \tanh(x)$$

Step Function

$$y = \begin{cases} 0, & x<n \\ 1, & x \geq n \end{cases}$$

Softplus

$$y = \ln(1+e^{x})$$

source: sefiks

ReLU

$$y = \begin{cases} 0, & x<0 \\ x, & x \geq 0 \end{cases}$$

Softsign

$$y = \frac{x}{(1+|x|)}$$

ELU

$$y = \begin{cases} \alpha(e^{x}-1), & x<0 \\ x, & x \geq 0 \end{cases}$$

Log of Sigmoid

$$y = \ln\left(\frac{1}{1+e^{-x}}\right)$$

Swish

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

Sinc

$$y = \frac{\sin(x)}{x}$$

Leaky ReLU

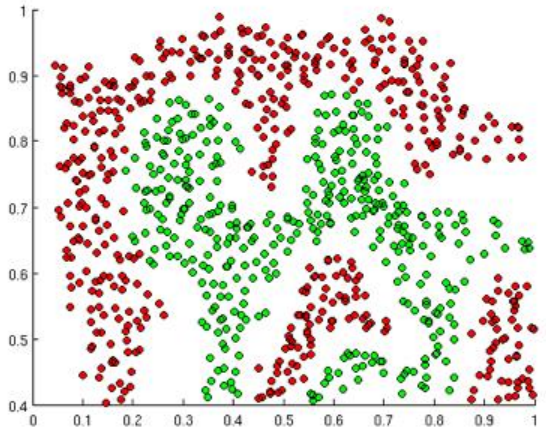
$$y = \max(0.1x, x)$$

Mish

$$y = x(\tanh(\text{softplus}(x)))$$

Importance of Activation Functions

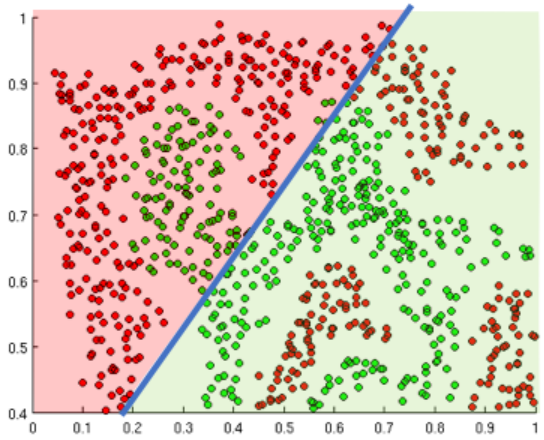
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Functions

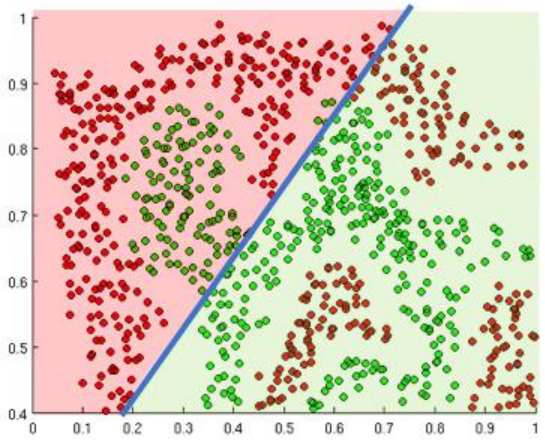
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



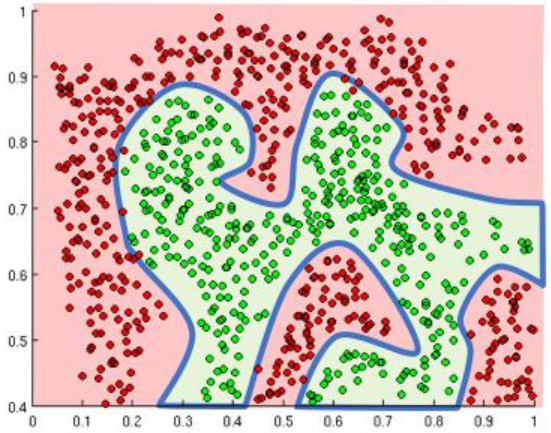
Linear activation functions produce linear decisions no matter the network size

Importance of Activation Functions

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

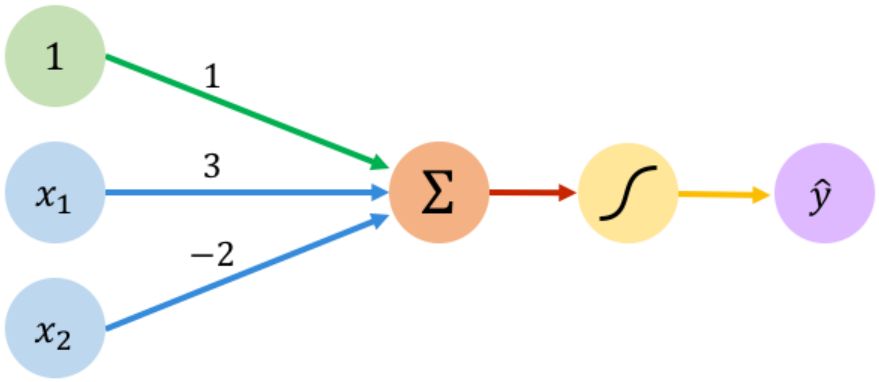


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The perceptron : Example



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

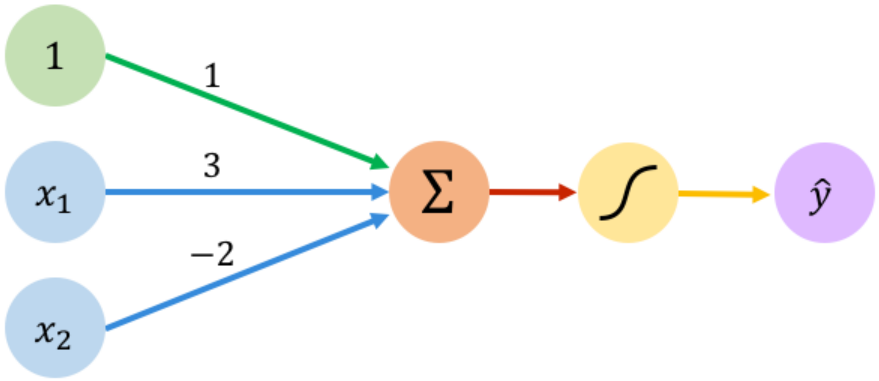
$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

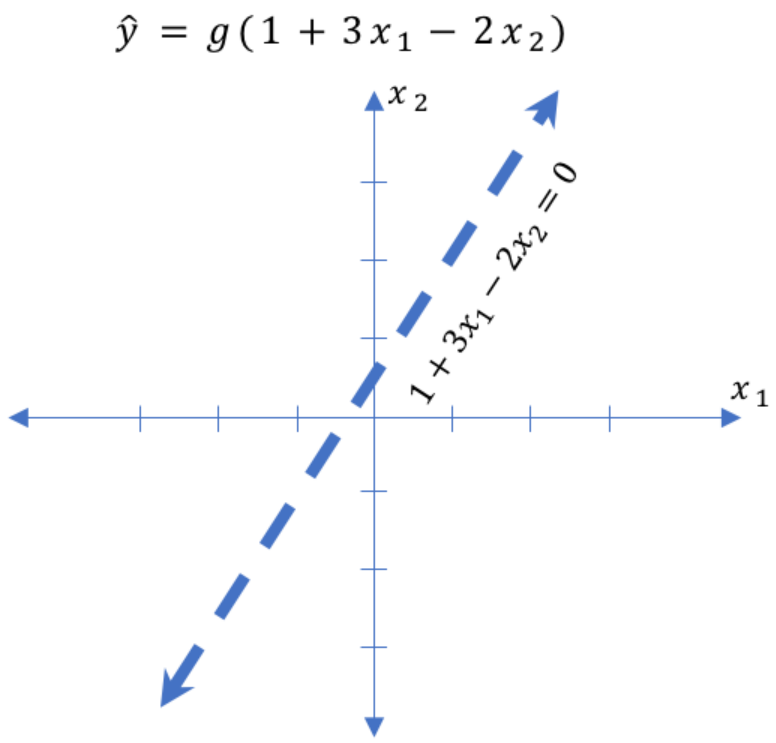
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

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

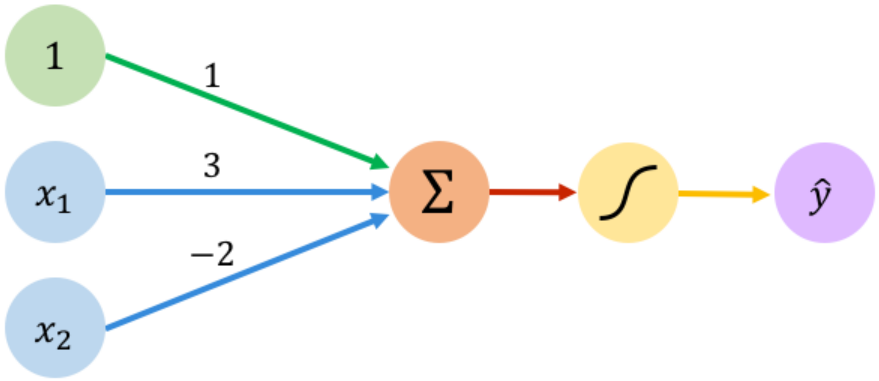
This is just a line in 2D!

The perceptron : Example





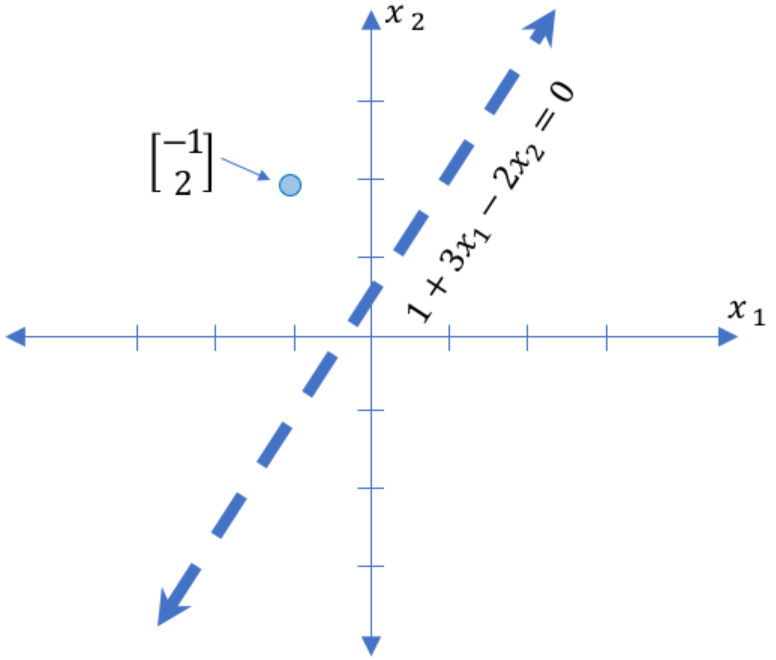
The perceptron : Example



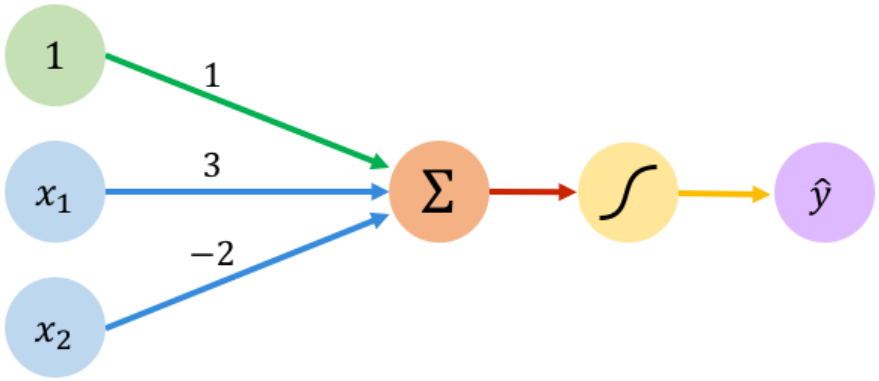
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

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

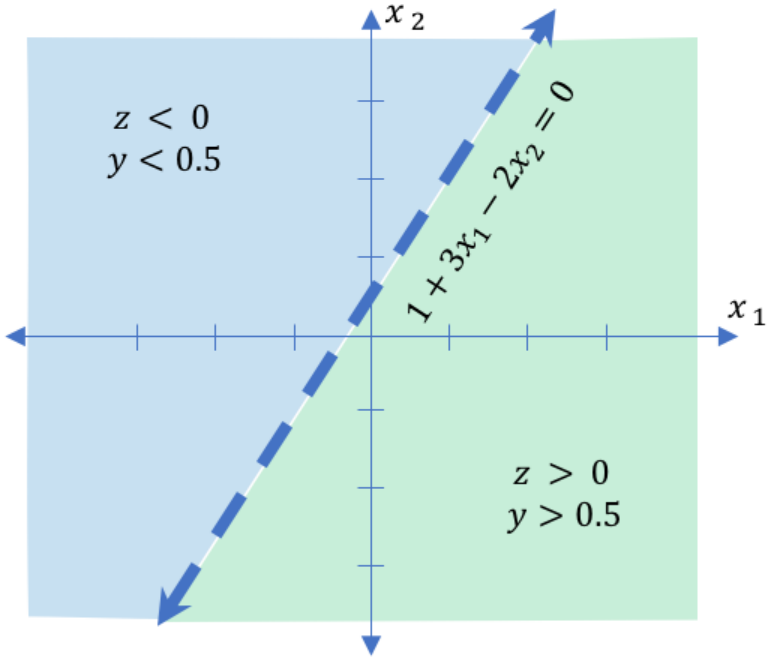
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The perceptron : Example



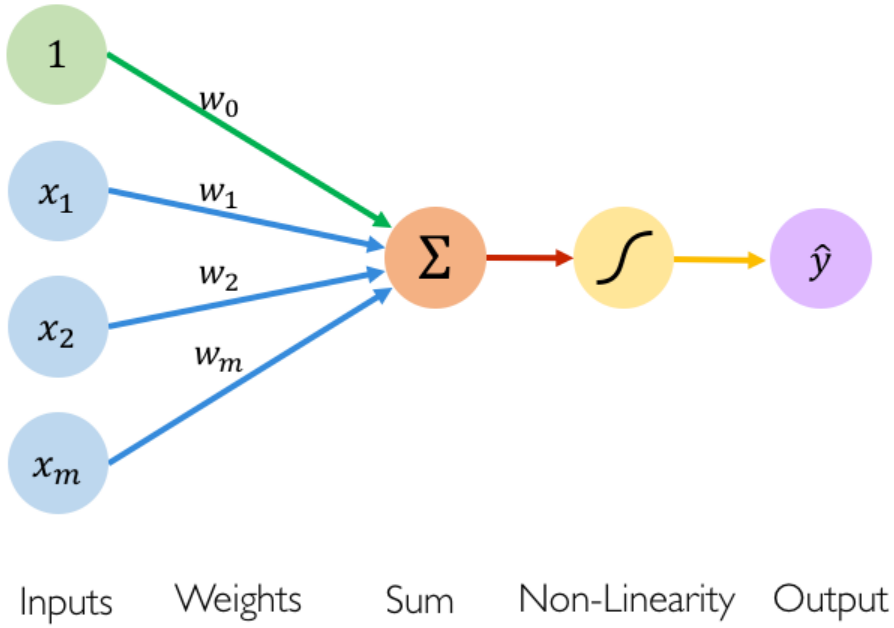
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



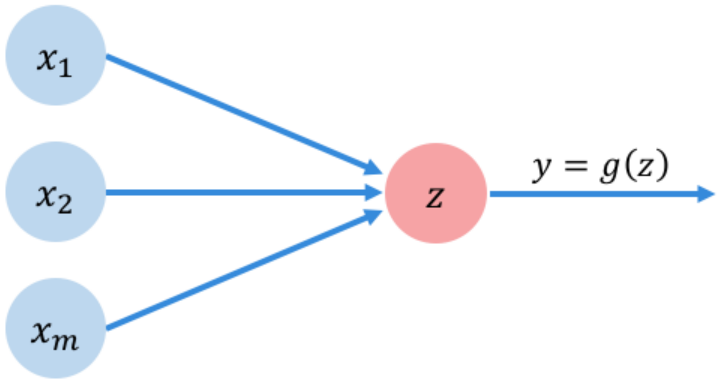
Building Neural Networks with perceptrons

The perceptron : Simplified

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



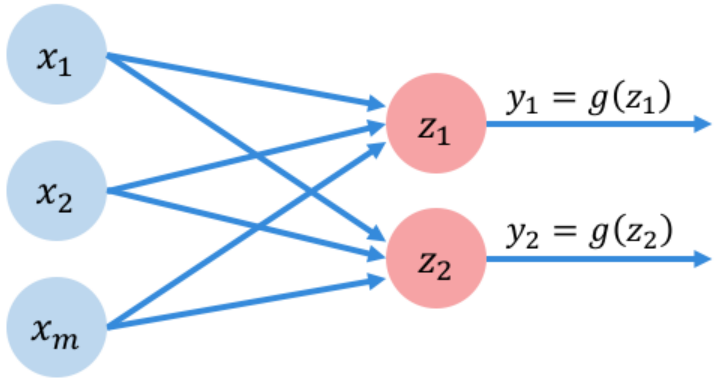
The perceptron : Simplified



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

Multi Output Perceptron

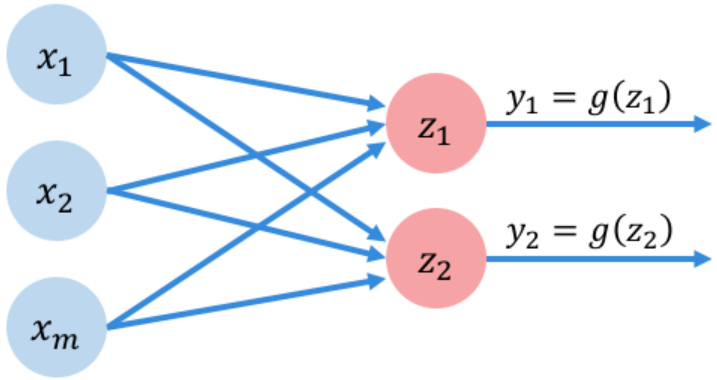
Because all inputs are densely connected to all outputs, these layers are called ***fully connected layers (FC)***



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

Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called ***fully connected layers (FC)***



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

LINEAR

CLASS `torch.nn.Linear(in_features: int, out_features: int, bias: bool = True)`

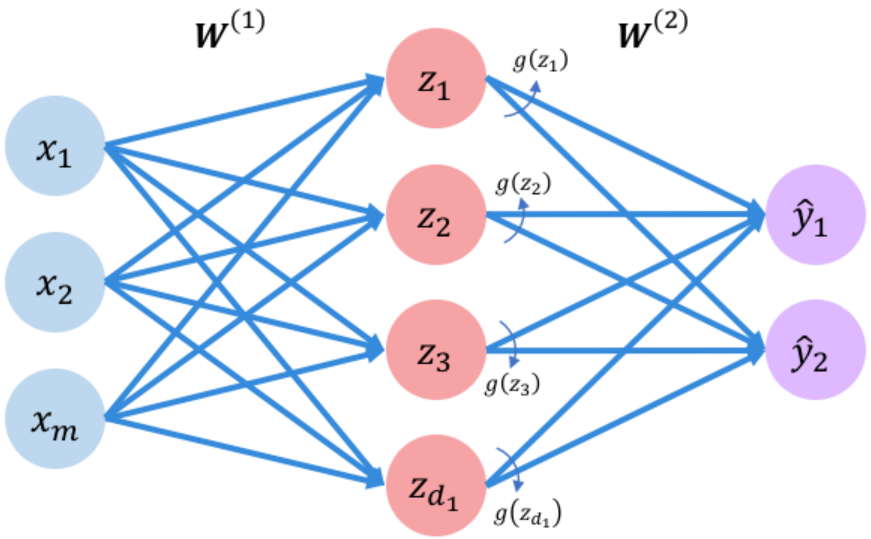
Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear

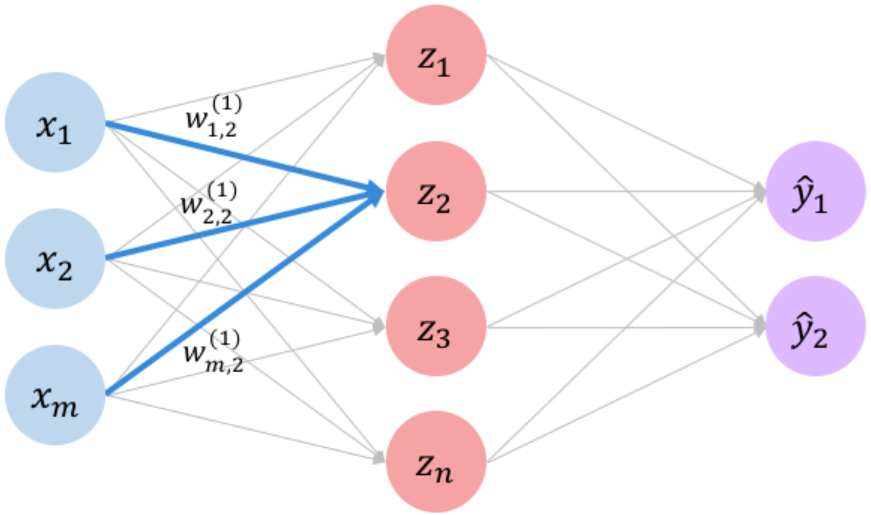
Single Layer Neural Network



Inputs Hidden Final Output

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

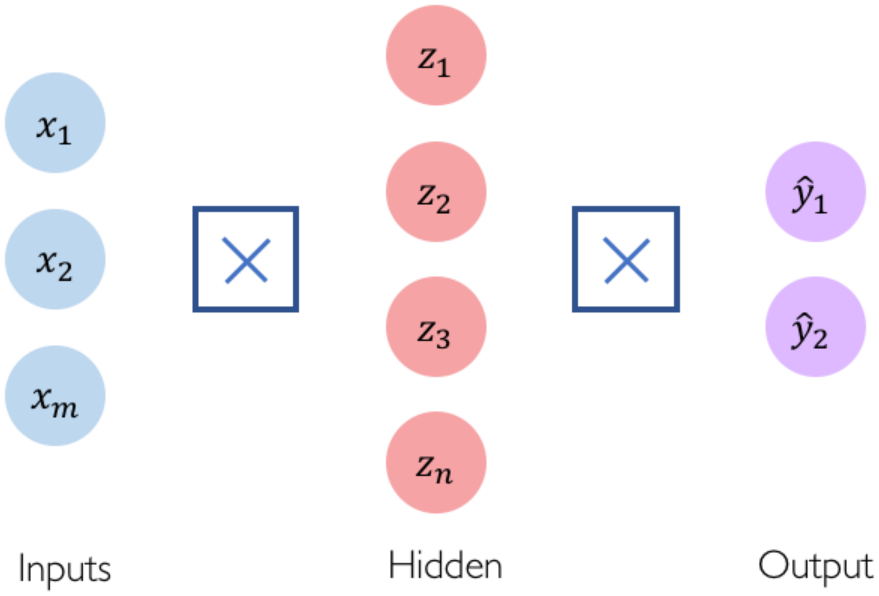
Single Layer Neural Network



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

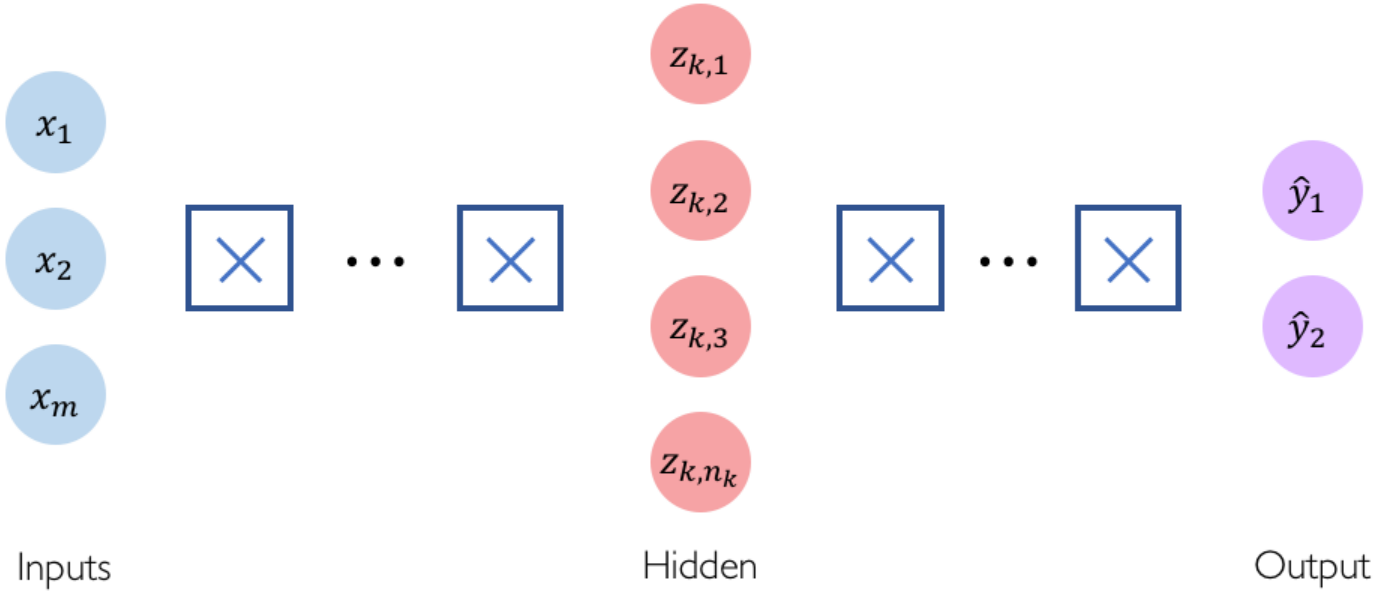
$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

Single Layer Neural Network



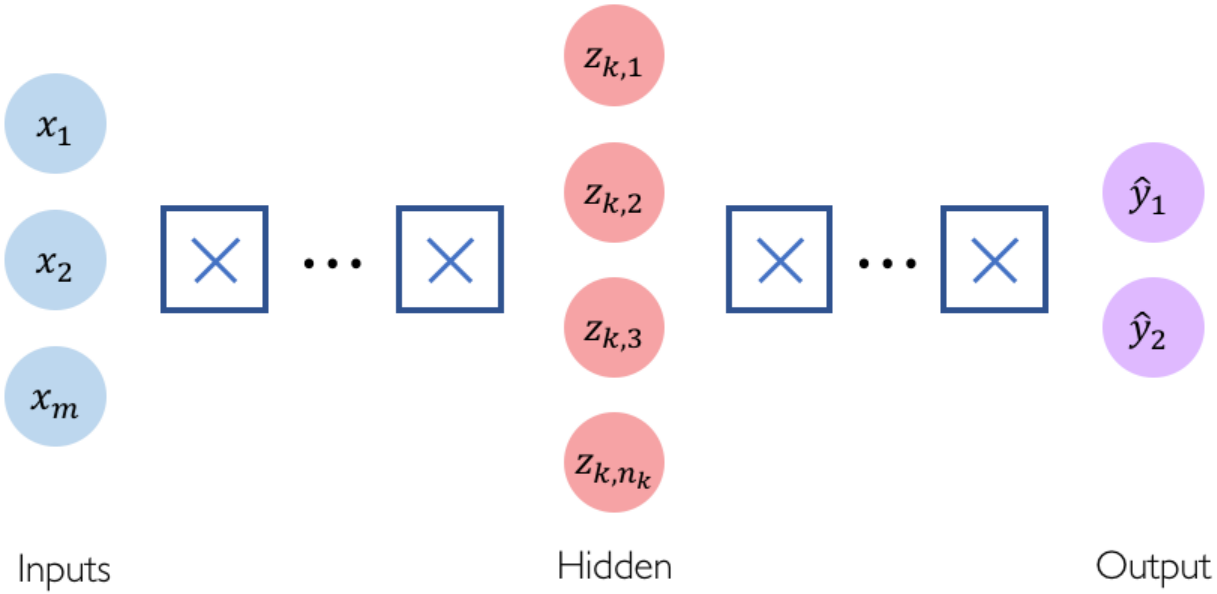

```
import torch

model = torch.nn.Sequential{
    torch.nn.Linear(m, n),
    torch.nn.Linear(n, 2)
}
```

"Deep" Neural Network



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

"Deep" Neural Network




```
import torch

model = torch.nn.Sequential{
    torch.nn.Linear(m, n1),
    torch.nn.Linear(n1,n2),
    .
    .
    .
    torch.nn.Linear(nk, 2)
}
```

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

Applying Neural Networks

Example problem

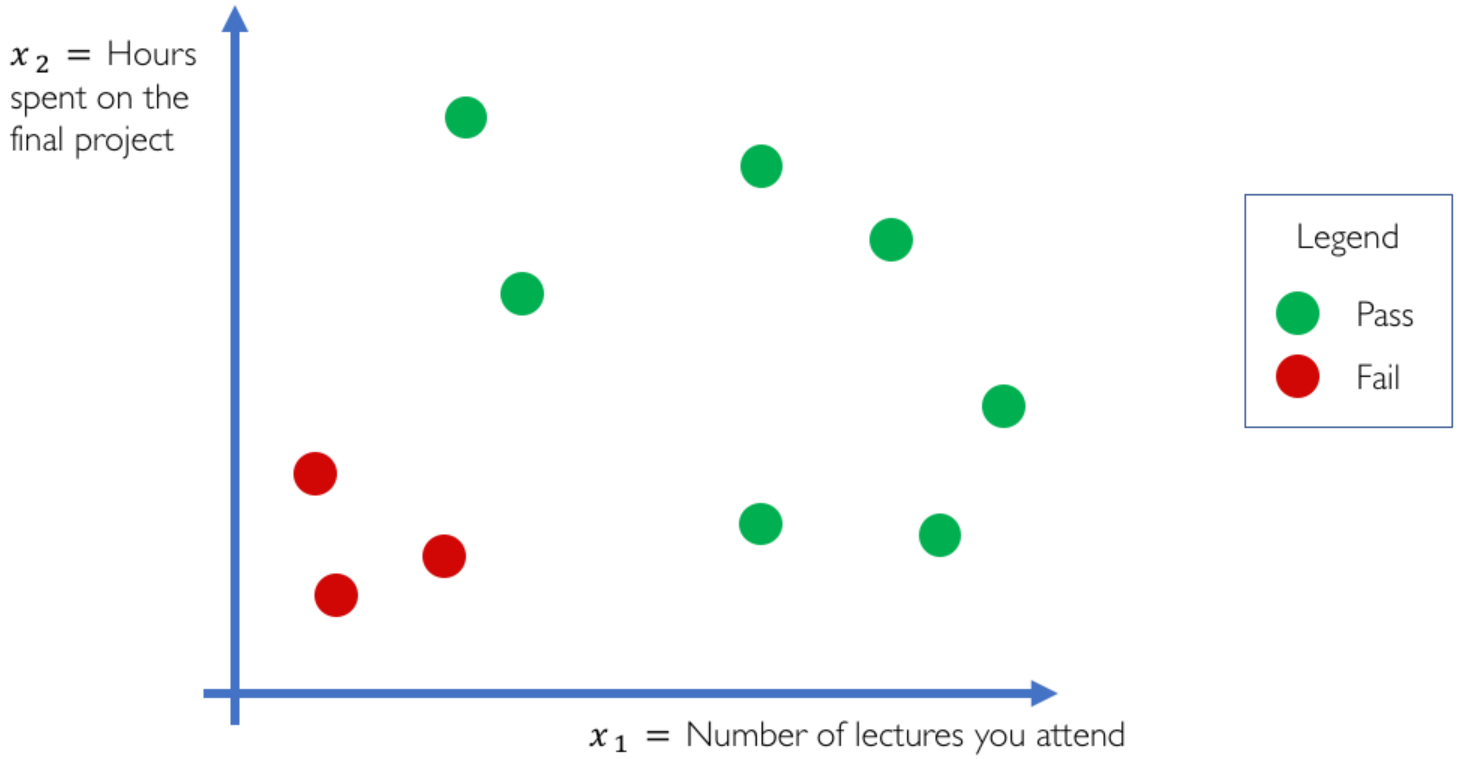
Will I pass this class?

Let's start with a simple two feature model

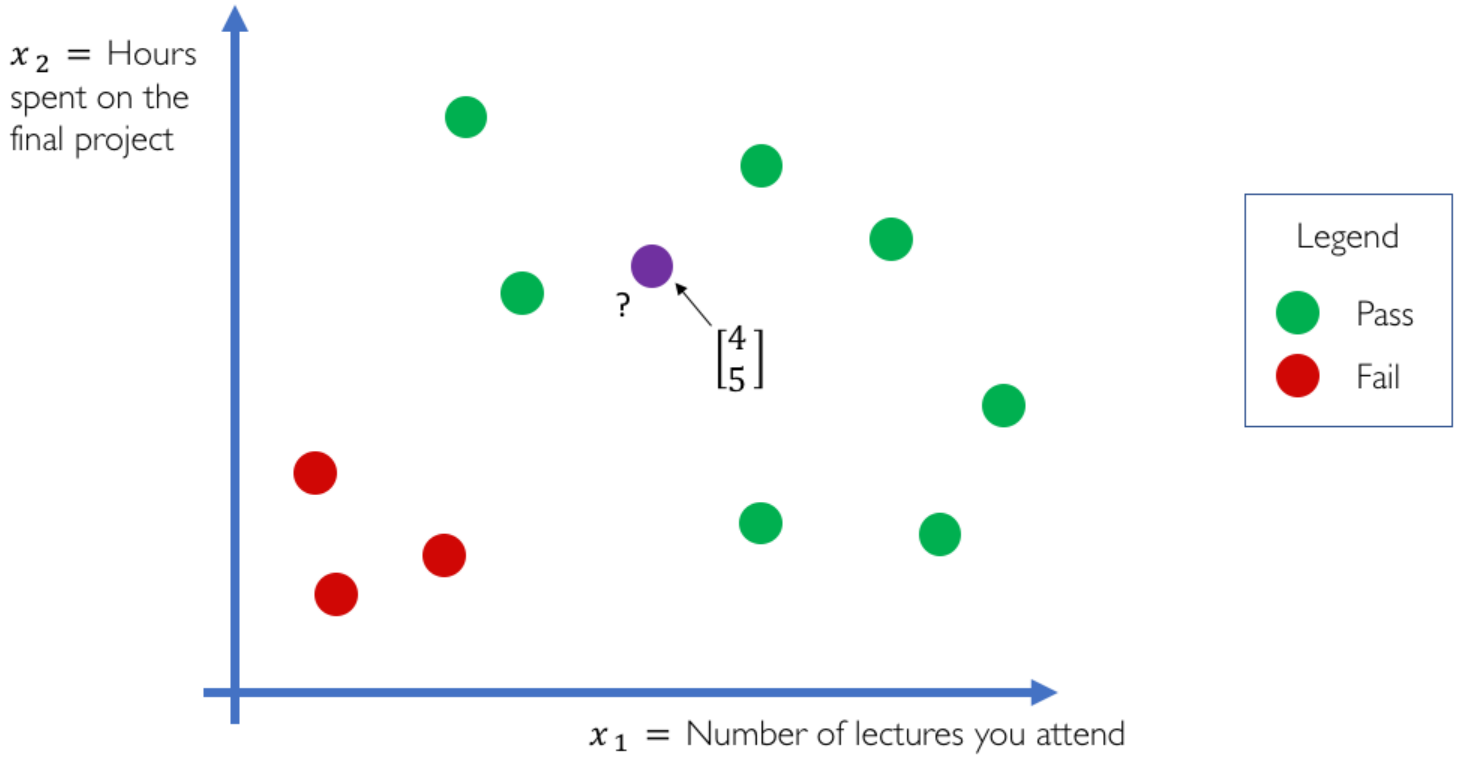
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

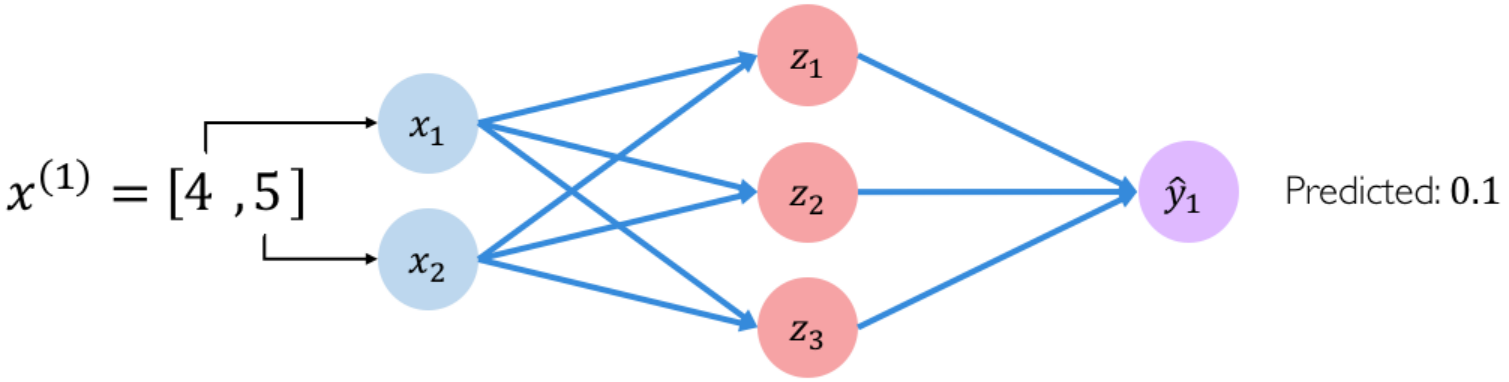
Example problem : Will I pass this class?



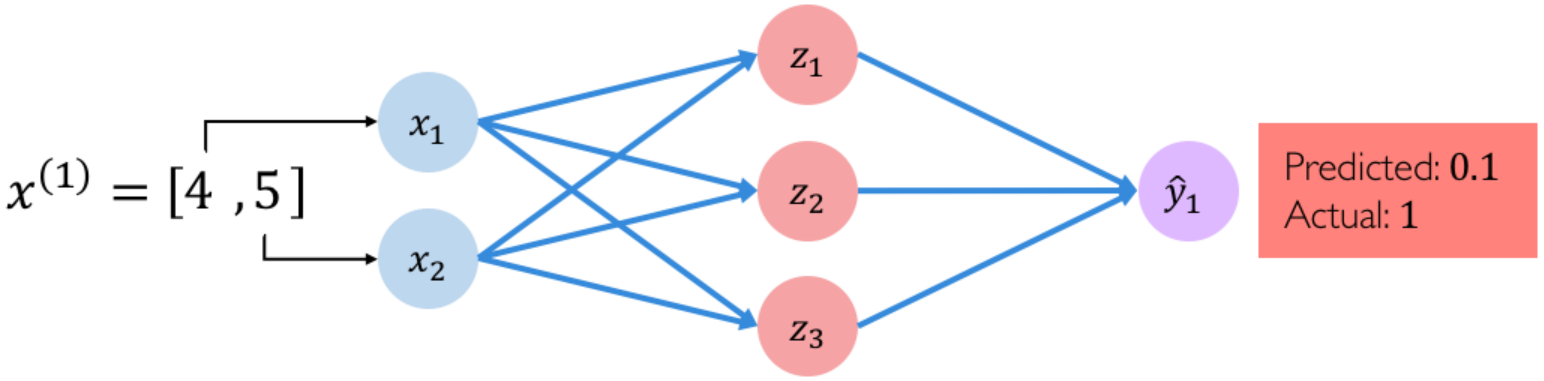
Example problem : Will I pass this class?



Example problem : Will I pass this class?

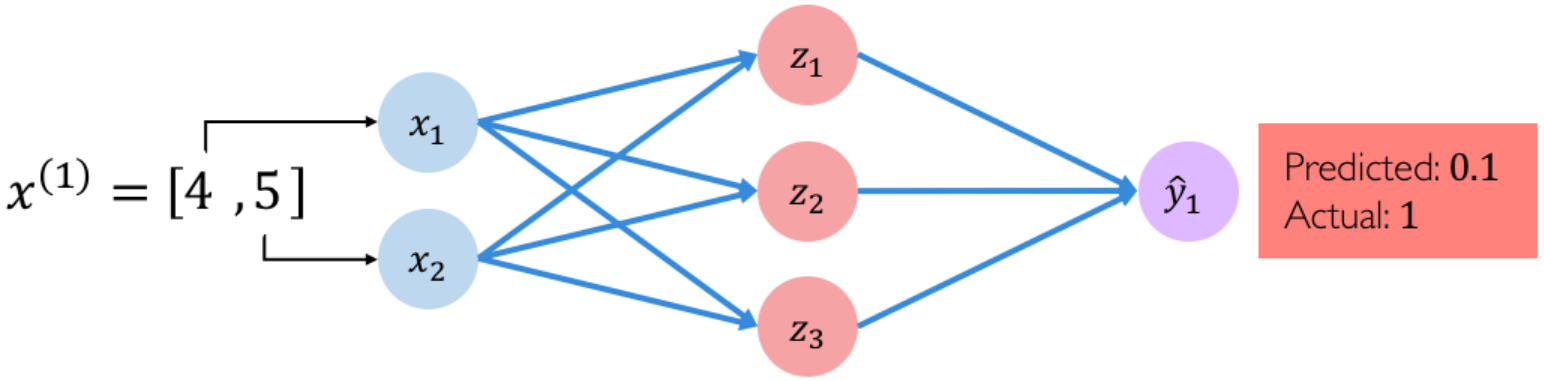


Example problem : Will I pass this class?



Quantifying Loss

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

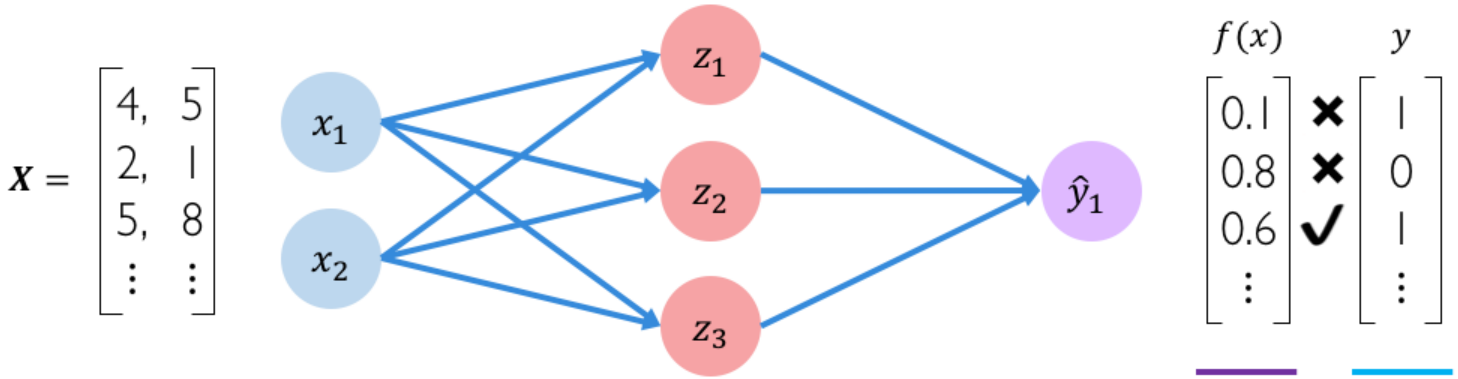


$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Quantifying Loss

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



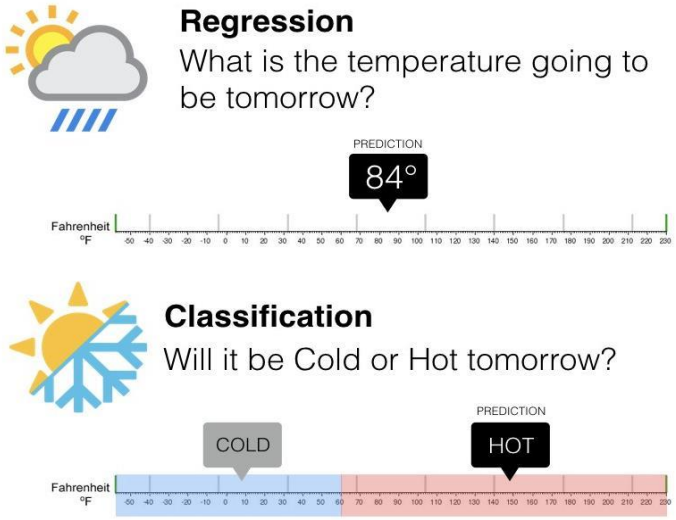
$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Also known as:

- Objective function
- Cost function
- Empirical Risk

Loss Functions



- Loss function *quantifies* gap between prediction and ground truth
- For regression:
  - Mean Squared Error (MSE)
- For classification:
  - Cross Entropy Loss

Mean Squared Error

$$MSE = \frac{1}{N}\sum(t_i - s_i)^2$$

Prediction

Ground Truth

Cross Entropy Loss

$$CE = -\sum_{i}^{C} t_i log(s_i)$$

Classes

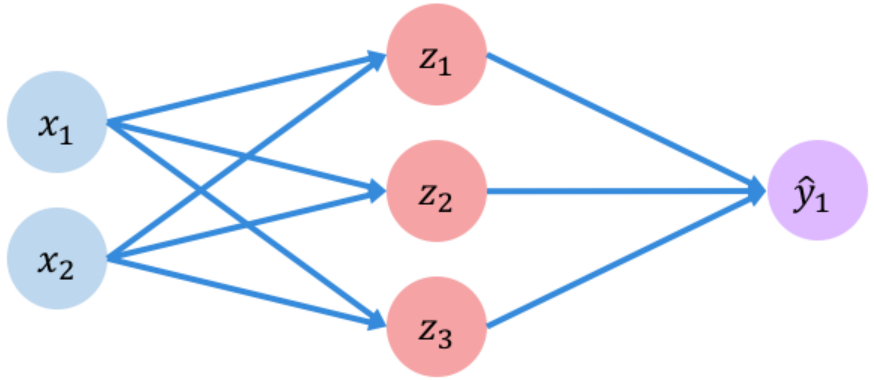
Prediction

Ground Truth {0,1}

Binary Cross Entropy Loss

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

$$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \begin{matrix} \times \\ \times \\ \checkmark \\ \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

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

Actual Predicted Actual Predicted

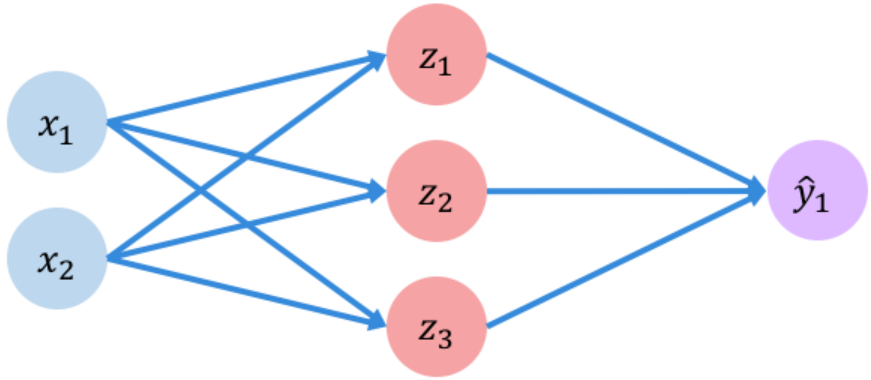
```
loss = F.binary_cross_entropy(predicted, y)
```

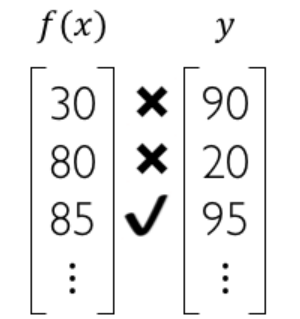
```
loss = nn.BCELoss()
output = loss(predicted, y)
```

Mean Square Error Loss

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

$$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$





Final Grades (percentage)

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; W\right)\right)^2$$

Actual Predicted

```
loss = F.mse_loss(predicted, y)
```

```
loss = nn.MSELoss()
output = loss(predicted, y)
```

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***

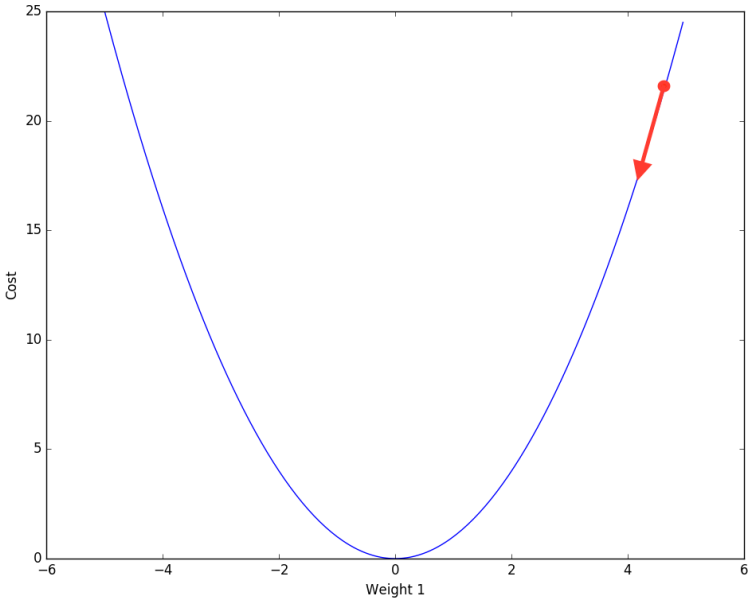
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

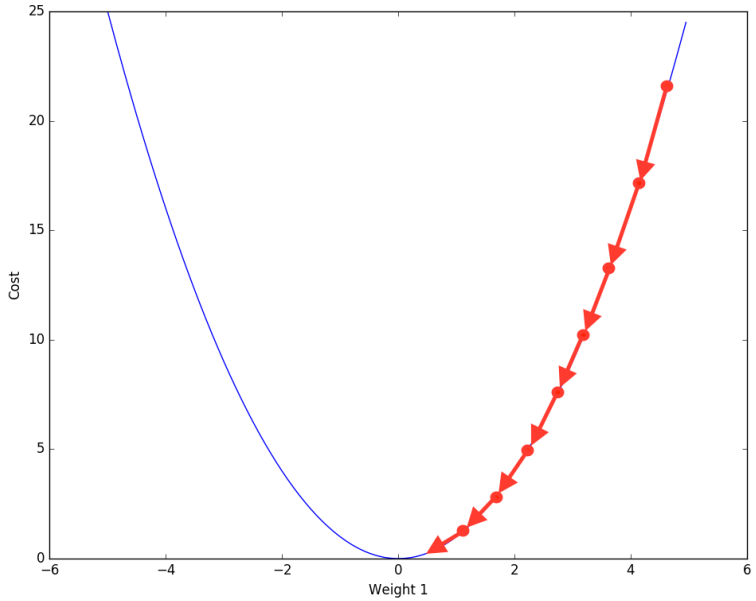
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

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

Learning is an Optimization Problem

Update the **weights** and **biases** to decrease loss function





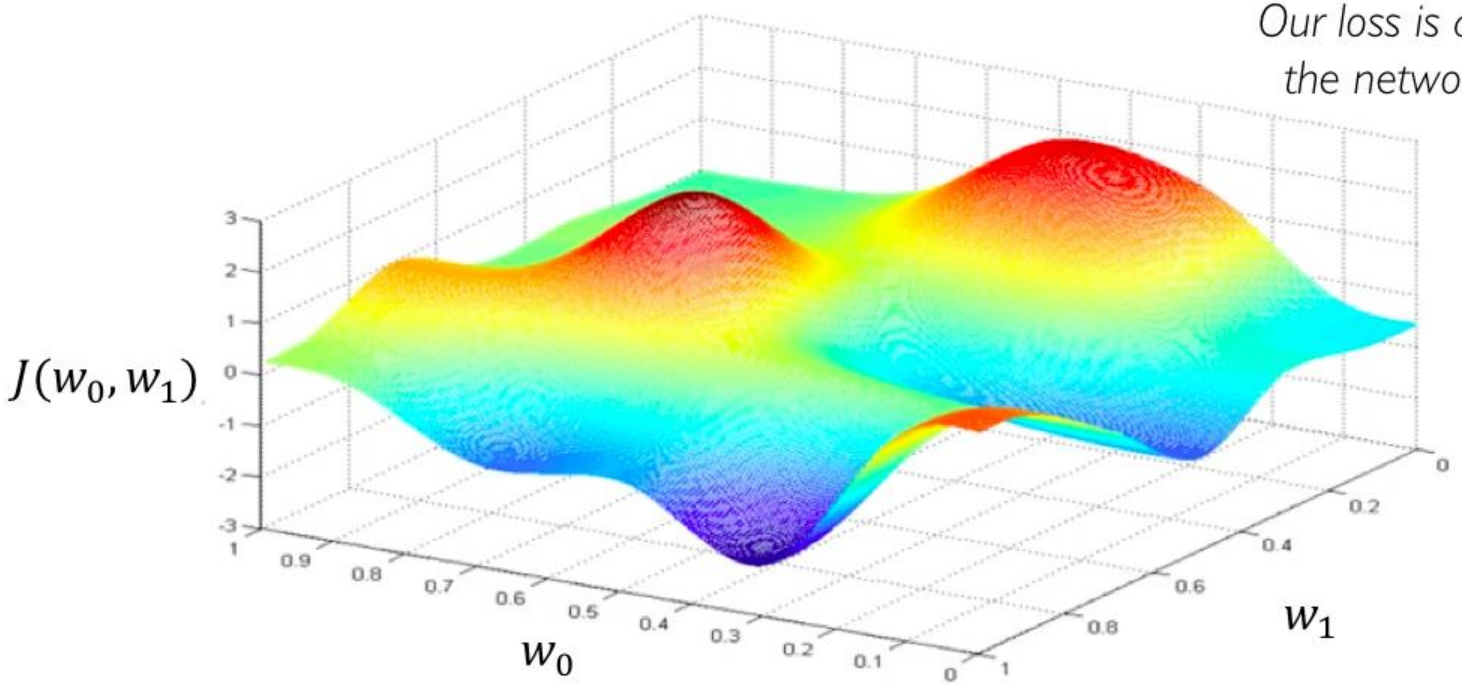
SGD: Stochastic Gradient Descent

Loss Optimization

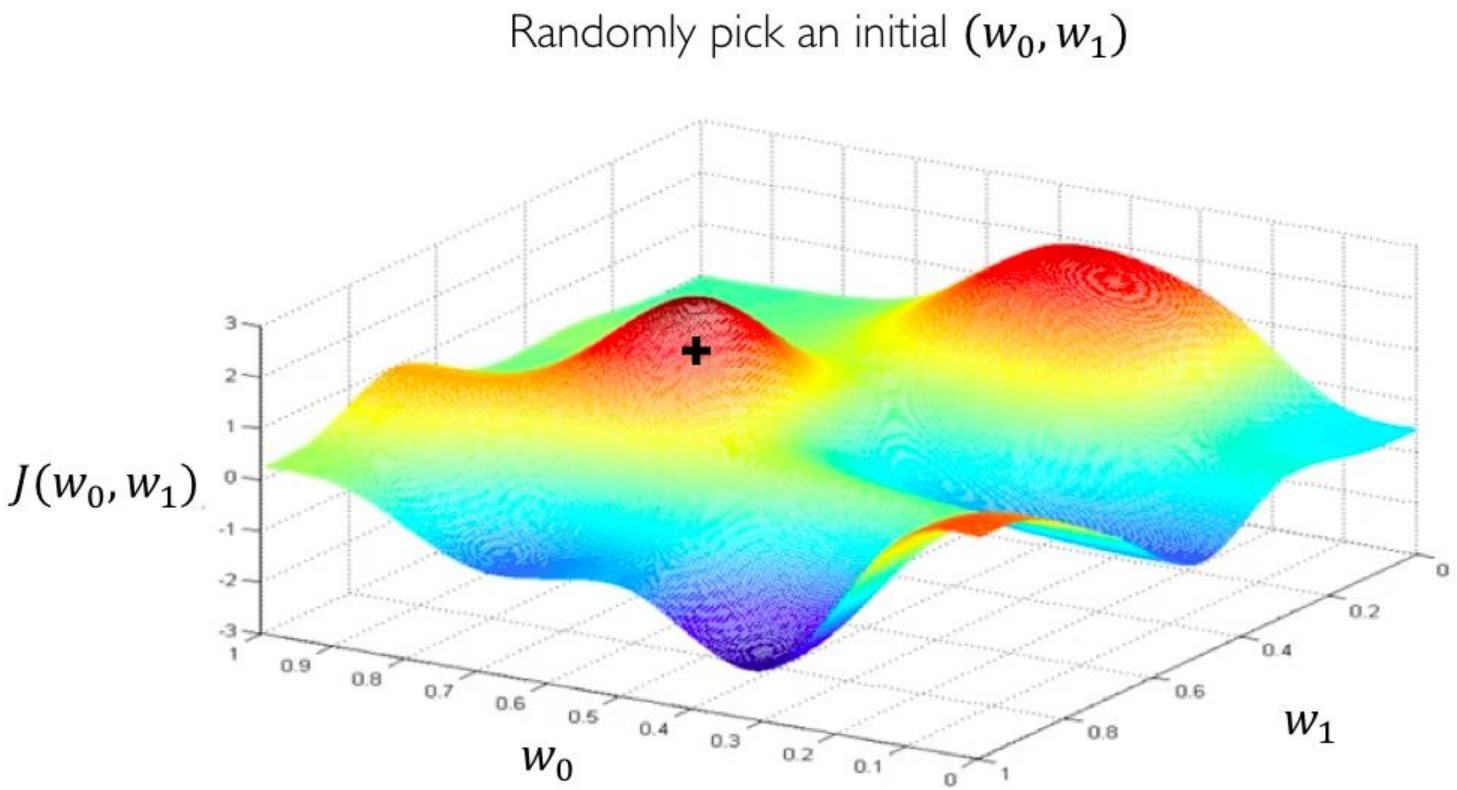
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:

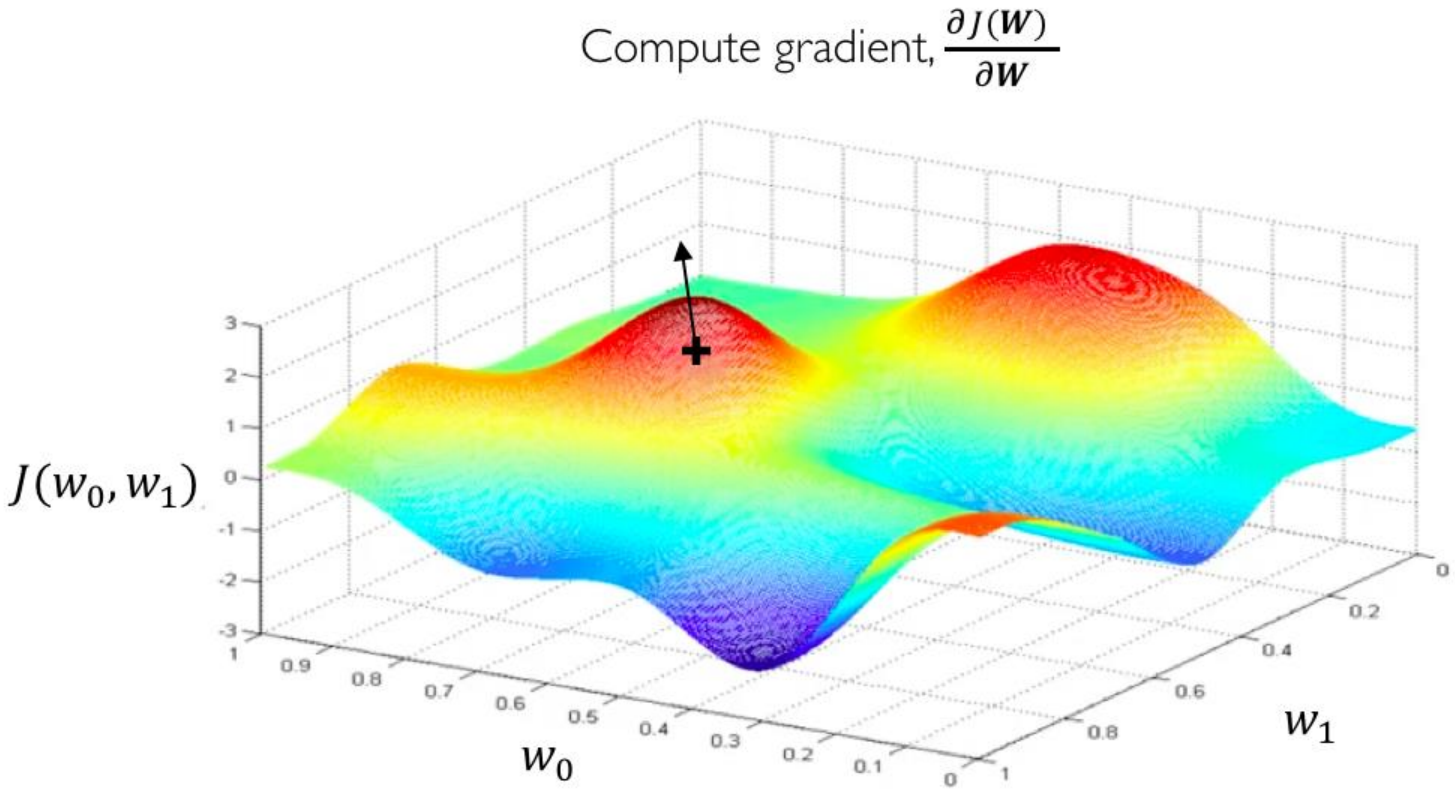
Our loss is a function of the network weights!



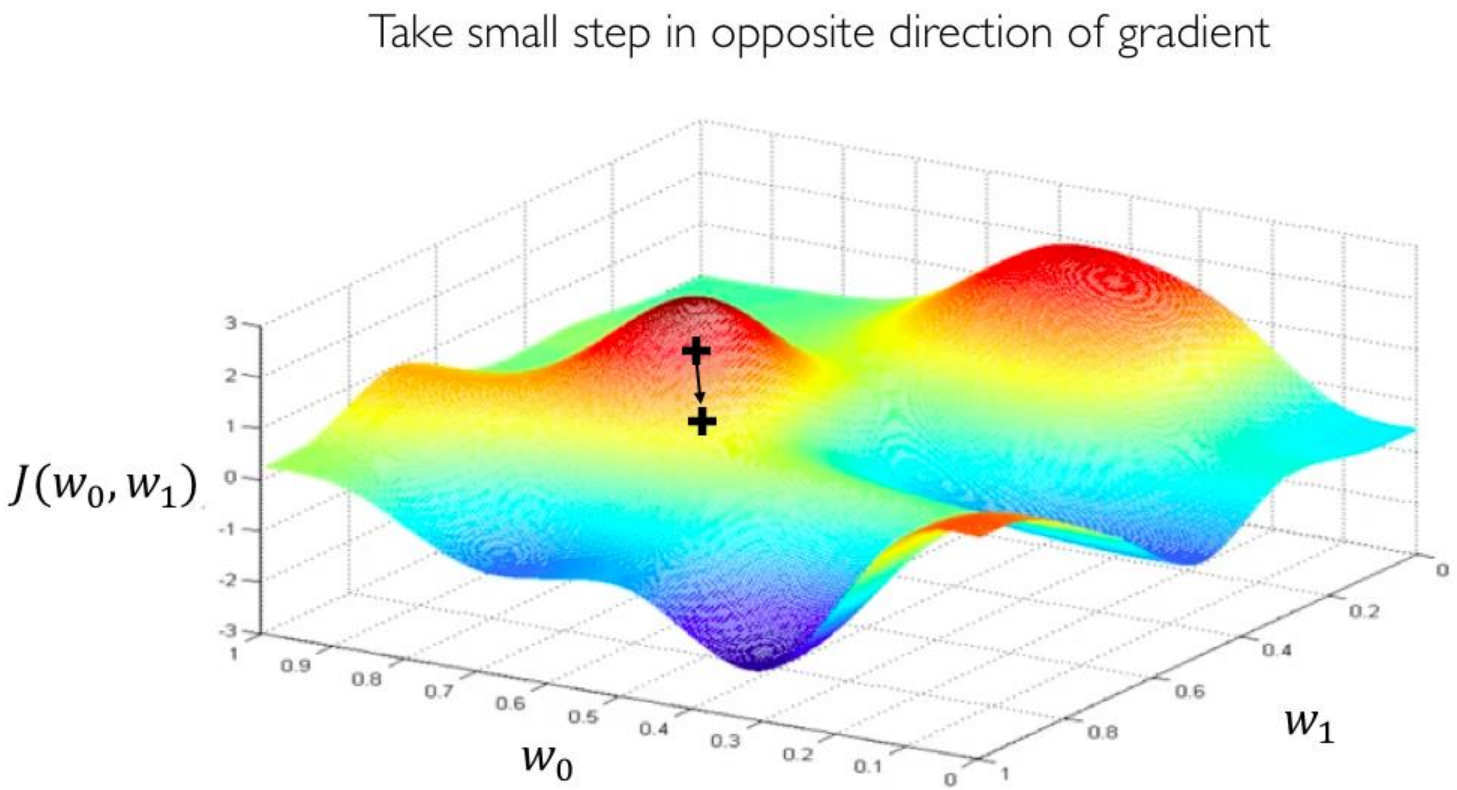
Loss Optimization



Loss Optimization

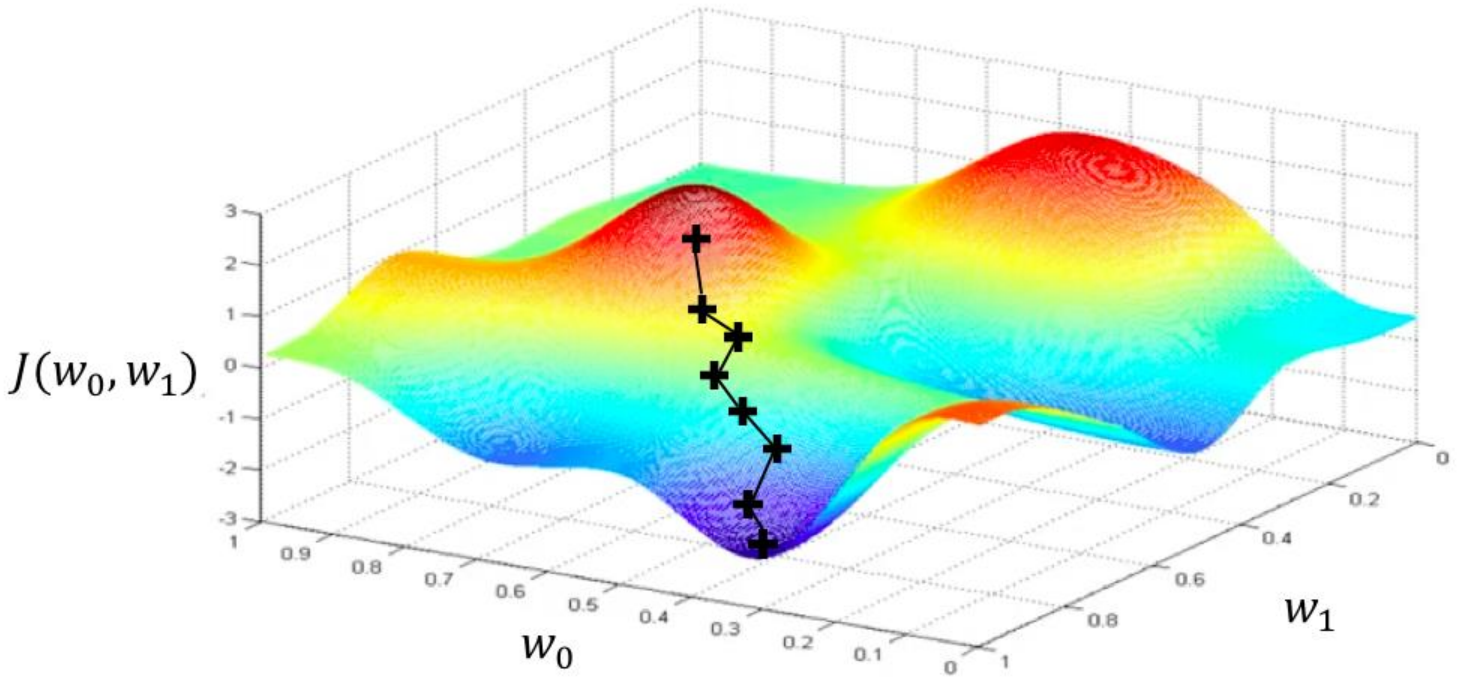


Loss Optimization



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, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
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

```python
import torch

for it in range(max_iter):

    predicted = model(x)

    loss = loss_fn(predicted, y)

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradients of the loss
    # with respect to all the learnable parameters of the model.
    loss.backward()

    # Update the weights using gradient descent.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

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, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

```python
import torch

for it in range(max_iter):

    predicted = model(x)

    loss = loss_fn(predicted, y)

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradients of the loss
    # with respect to all the learnable parameters of the model.
    loss.backward()

    # Update the weights using gradient descent.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

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, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

```python
import torch

for it in range(max_iter):

    predicted = model(x)

    loss = loss_fn(predicted, y)

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradients of the loss
    # with respect to all the learnable parameters of the model.
    loss.backward()

    # Update the weights using gradient descent.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

Gradient Descent

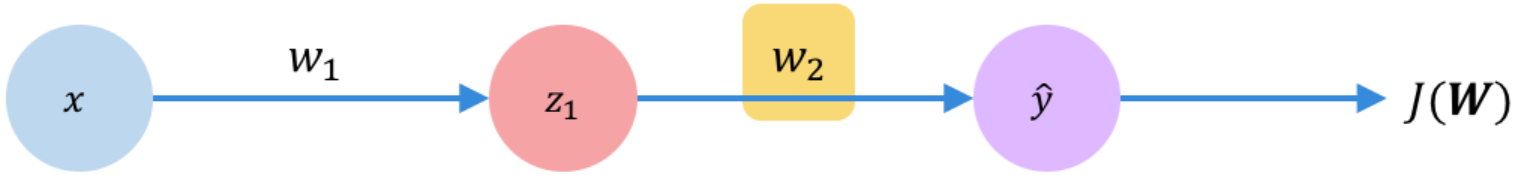
Algorithm

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, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

```python
import torch

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
for it in range(max_iter):

    predicted = model(x)

    loss = loss_fn(predicted, y)

    # Zero the gradients before running the backward pass.
    # model.zero_grad()
    optimizer.zero_grad()

    # Backward pass: compute gradients of the loss
    # with respect to all the learnable parameters of the model.
    loss.backward()

    # Update the weights using gradient descent.
    with torch.no_grad():
        # for param in model.parameters():
        #   param -= learning_rate * param.grad
        optimizer.step()
```

Computing Gradient : Backpropagation

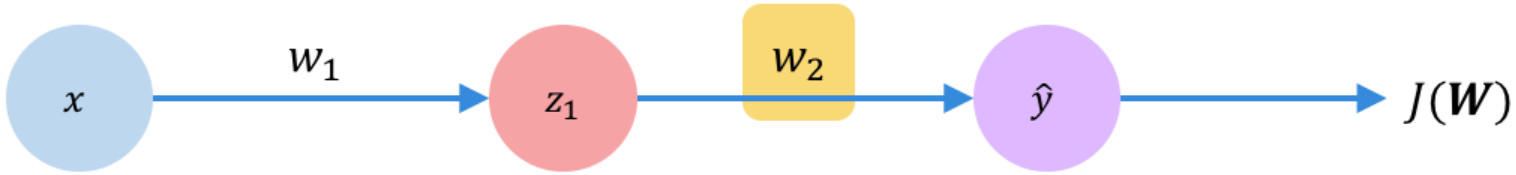
(aka. backprop.)



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

Computing Gradient : Backpropagation

(aka. backprop.)

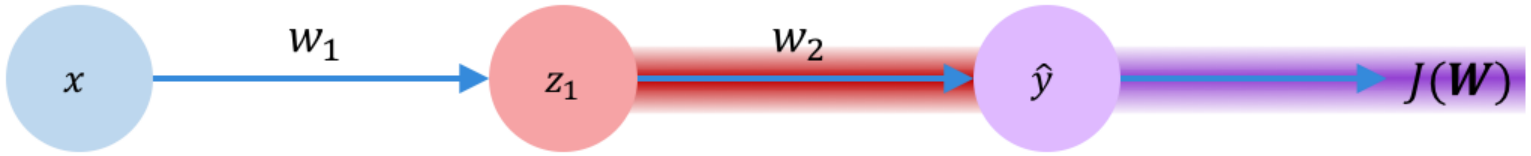


$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} =$$

Let's use the chain rule!

Computing Gradient : Backpropagation

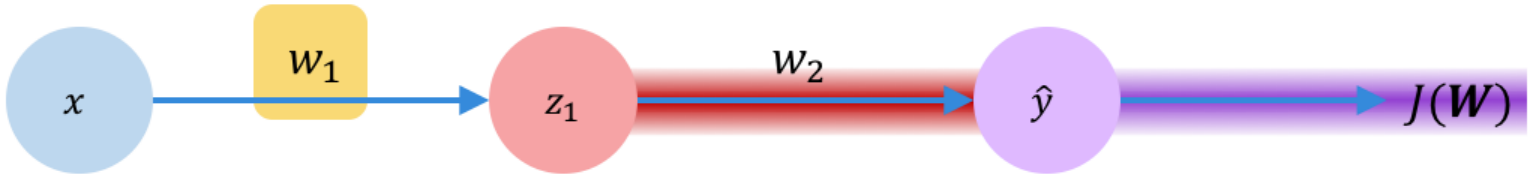
(aka. backprop.)



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

Computing Gradient : Backpropagation

(aka. backprop.)



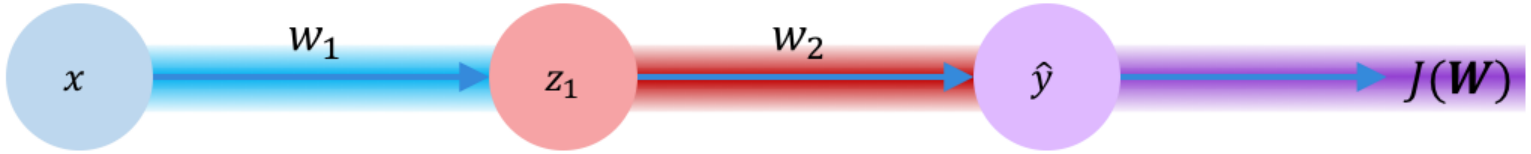
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

Computing Gradient : Backpropagation

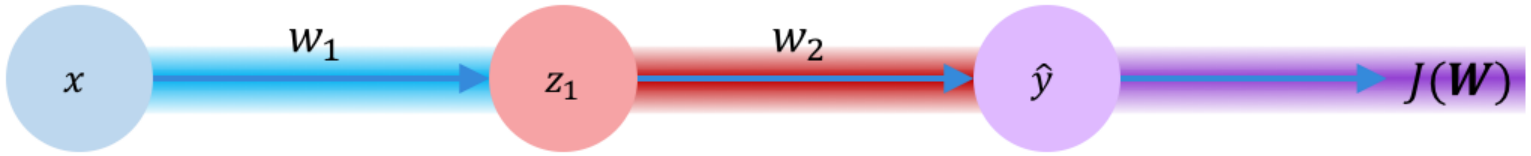
(aka. backprop.)



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

Computing Gradient : Backpropagation

(aka. backprop.)



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

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

Back propagation in practice

Computer does backprop. for you

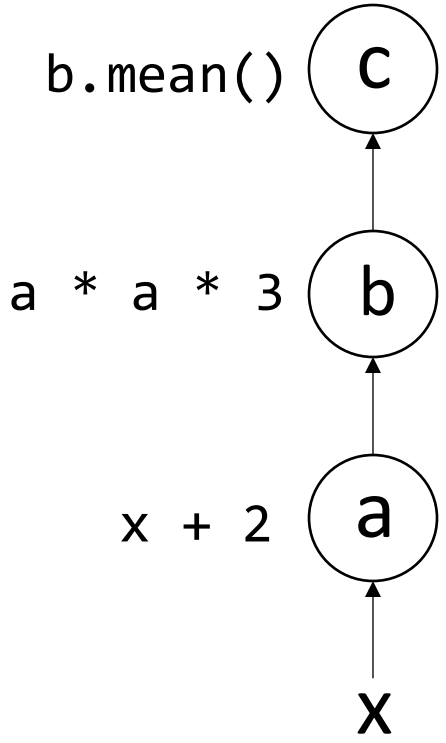
- Autograd
- framework: Tensorflow, Pytorch, MXNet

A neural network model is represented as Directed Acylic Graph (DAG)

- Each node contains a mathematical operation.
- Each node contains the derivative of the math operation.
- The error signal is backpropagated from the output nodes to the input nodes

Math program as DAG

```
x : [[1,1], [1,1]]
a = x + 2
b = a * a * 3
c = b.mean()
```

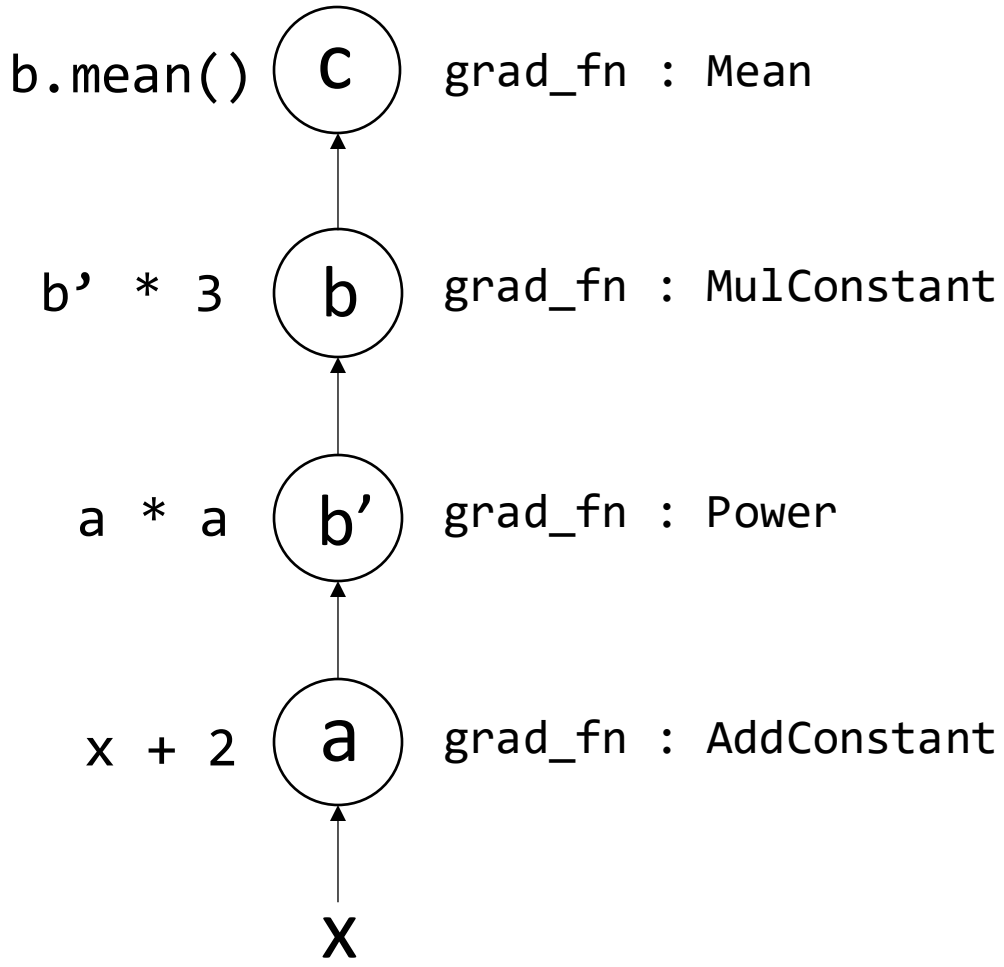


Math program as DAG

```
x : [[1,1], [1,1]]
a = x + 2
b’= a * a
b = b’ * 3
c = b.mean()
```

Each `grad_fn` has

- forward
  - Compute forward computation
  - Save input & misc. for backward
- backward
  - given δ^l, calculate δ^{l-1}

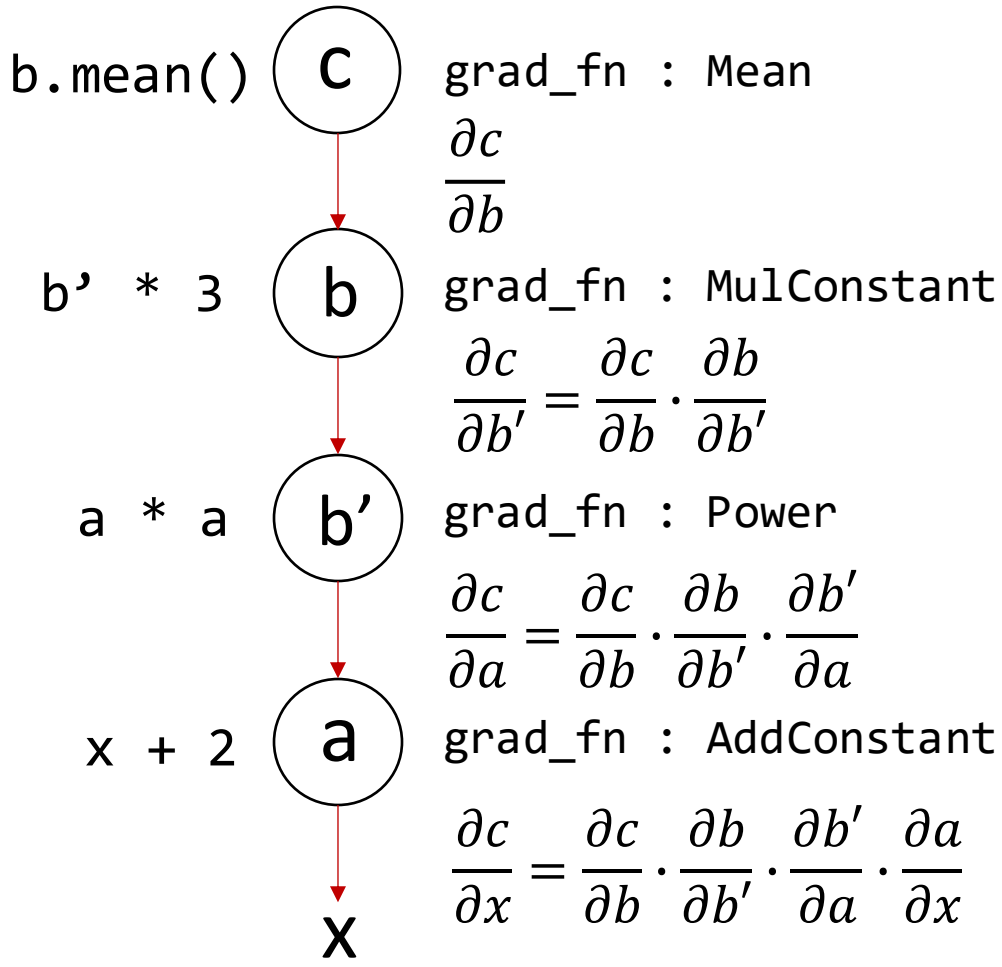


Math program as DAG

```
x : [[1,1], [1,1]]
a = x + 2
b’= a * a
b = b’ * 3
c = b.mean()
```

Each `grad_fn` has

- forward
  - Compute forward computation
  - Save input & misc. for backward
- backward
  - given δ^l, calculate δ^{l-1}

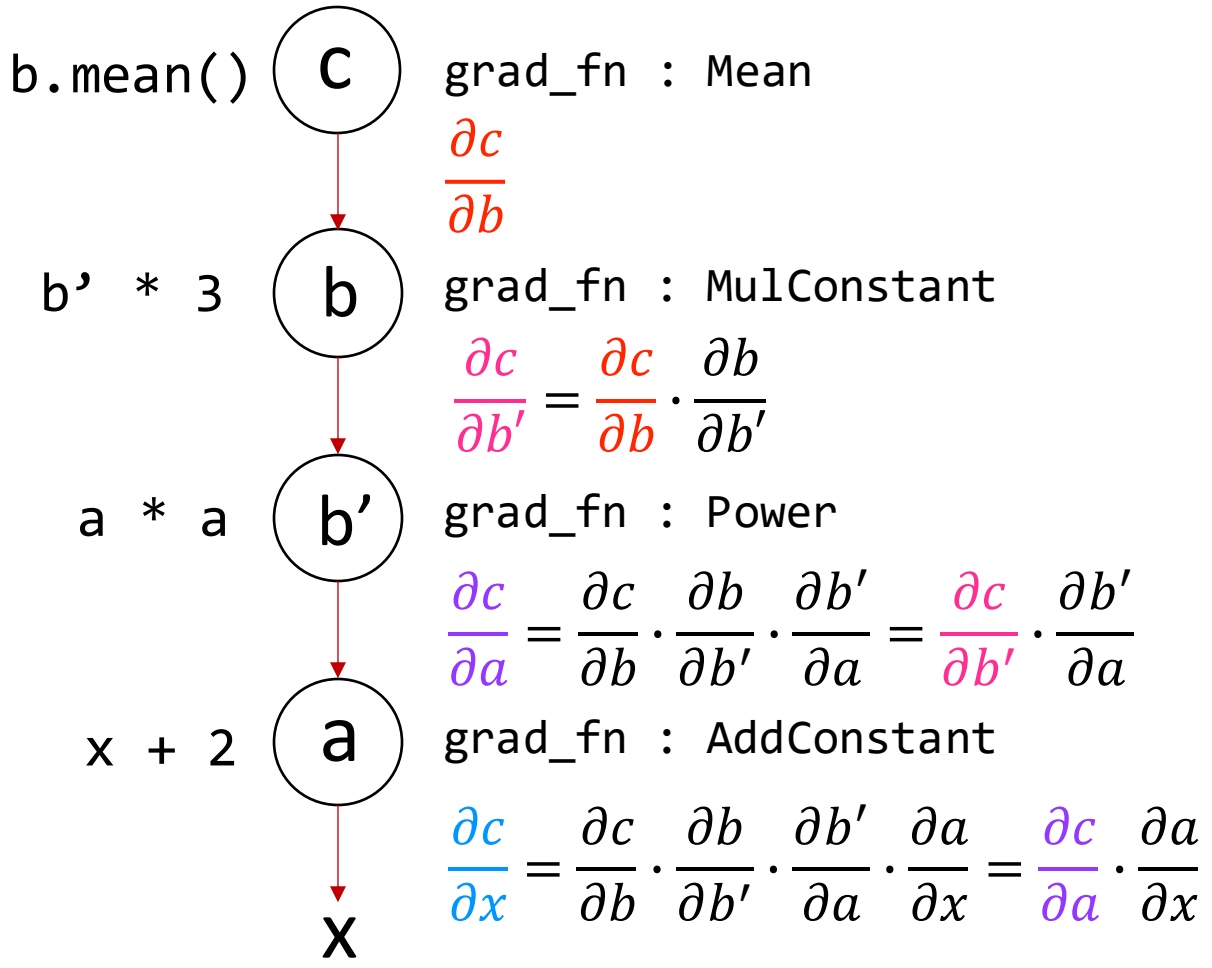


Math program as DAG

```
x : [[1,1], [1,1]]
a = x + 2
b’= a * a
b = b’ * 3
c = b.mean()
```

Each `grad_fn` has

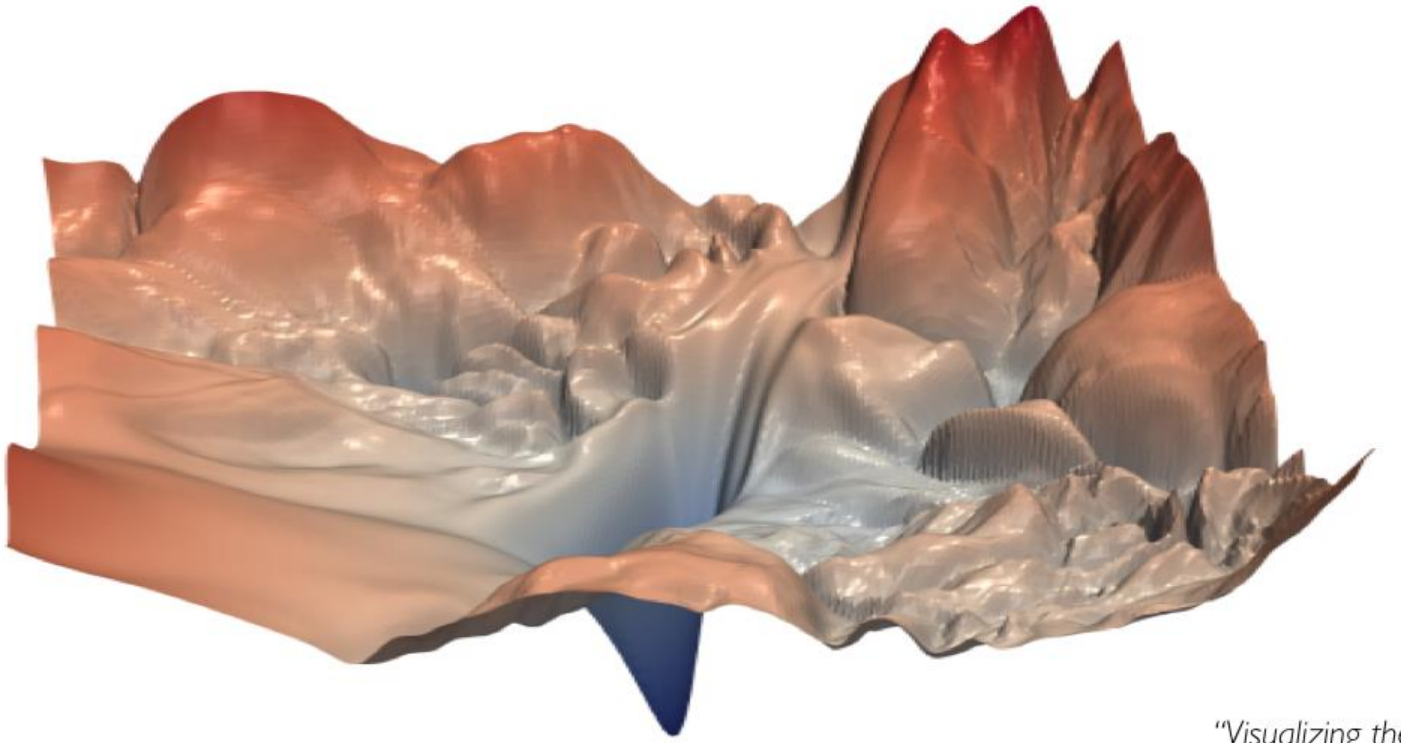
- forward
  - Compute forward computation
  - Save input & misc. for backward
- backward
  - given δ^l, calculate δ^{l-1}



Neural Networks in Practice:

Optimization

Training Neural Network is difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Loss Functions Can Be Difficult to Optimize

Remember:

Optimization through gradient descent

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

Loss Functions Can Be Difficult to Optimize

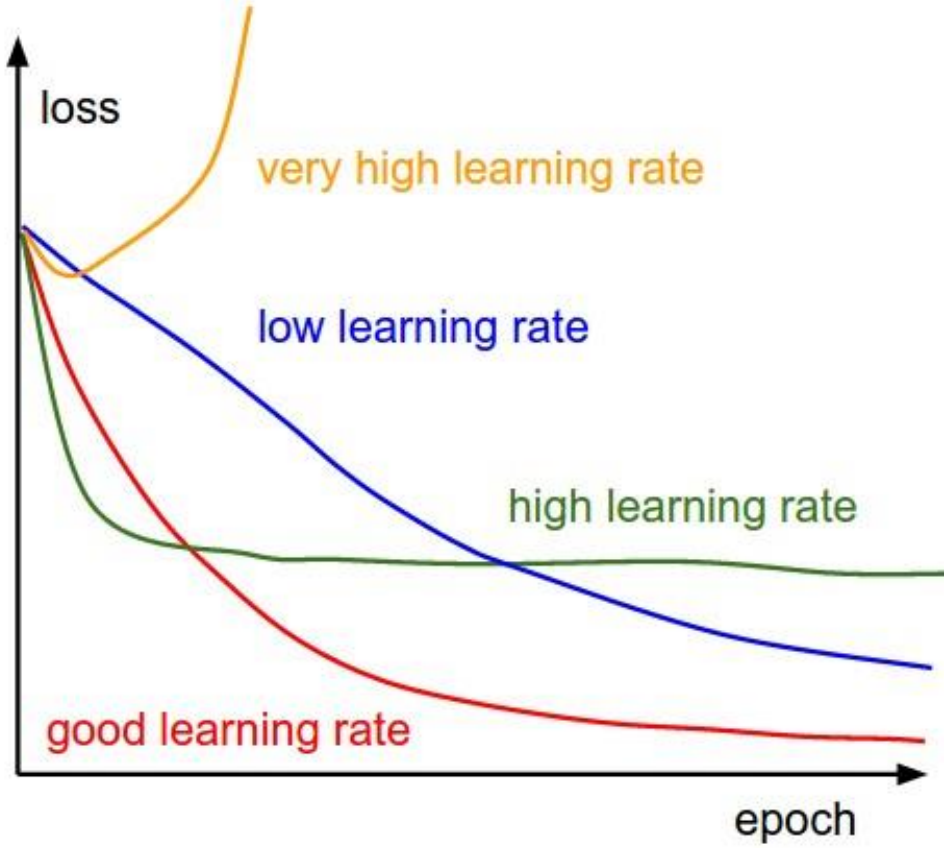
Remember:

Optimization through gradient descent

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

How can we set the learning rate?

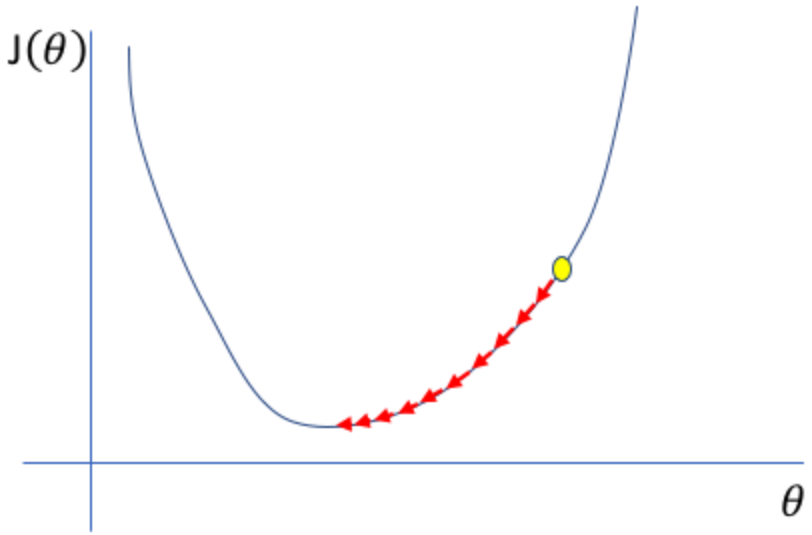
Setting the Learning Rate



credit : CS231n

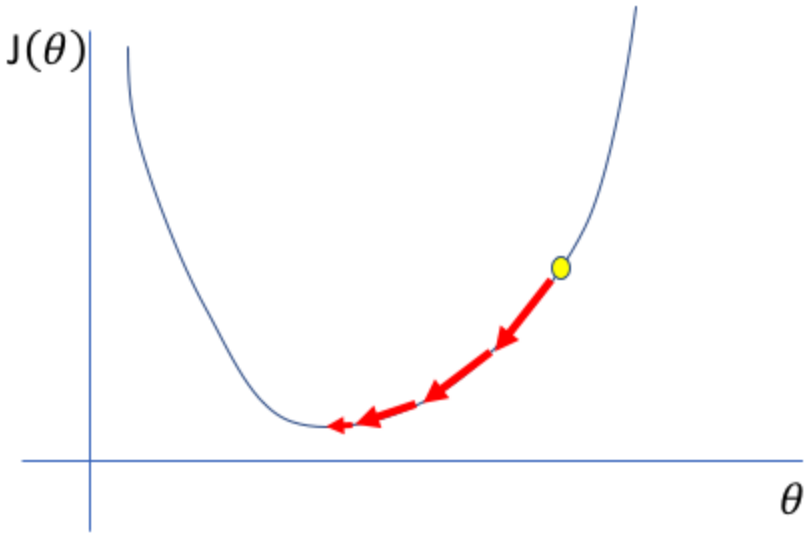
Setting the Learning Rate

Too low



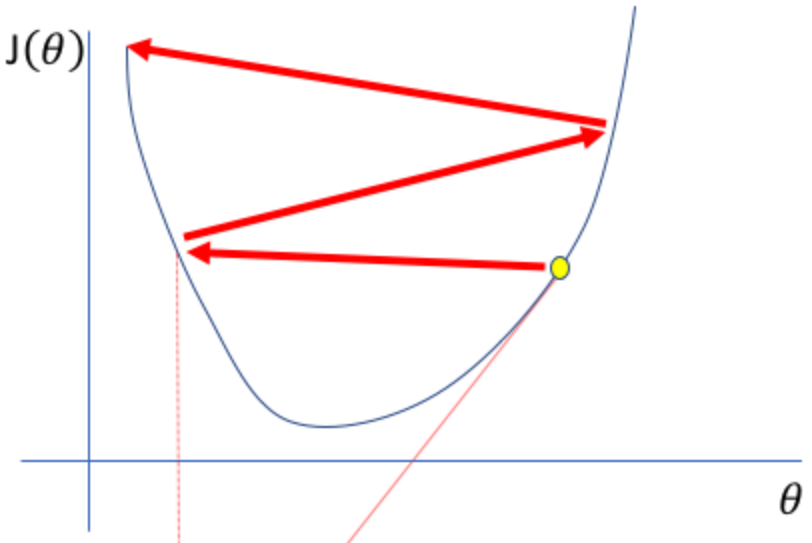
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

https://www.jeremyjordan.me/nn-learning-rate/

How to deal with this?

Idea 1:

Try lots of different learning rates and see what works "just right"

How to deal with this?

Idea 1:

Try lots of different learning rates and see what works "just right"

Idea 2:

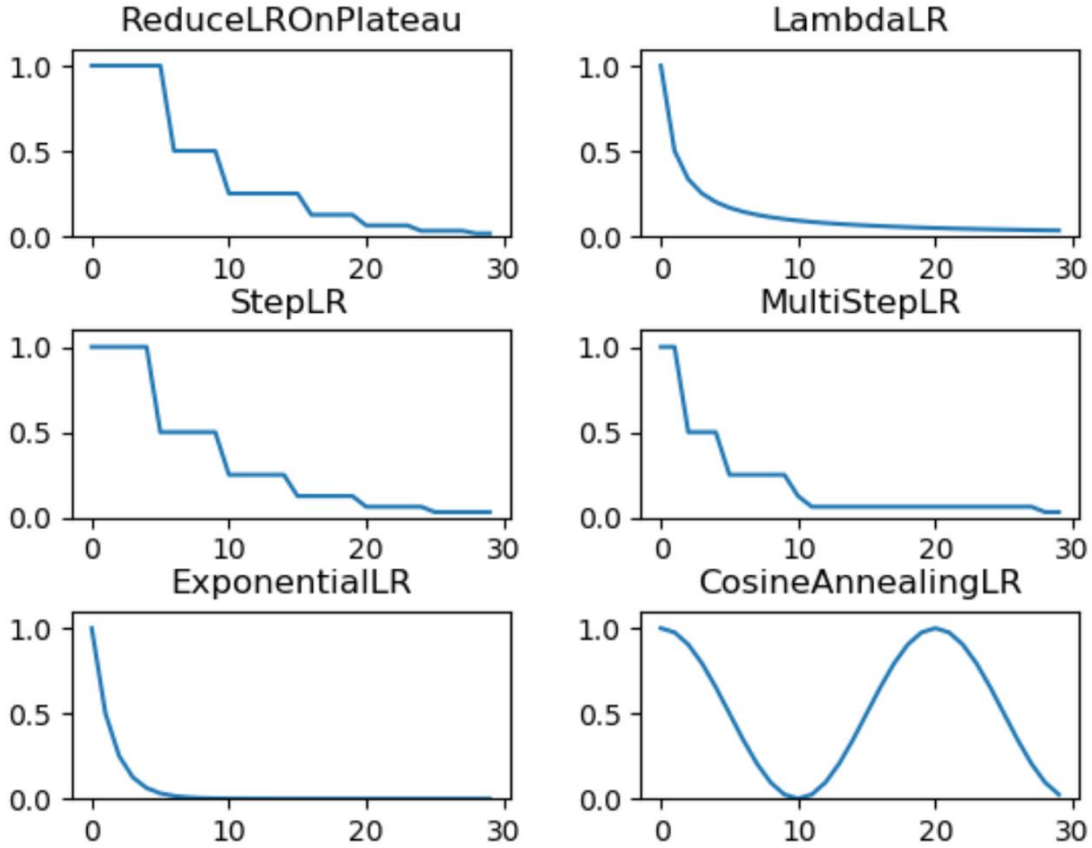
Do something smarter!
Design an adaptive learning rate that "adapts" to the landscape

Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

Adaptive Learning Rates

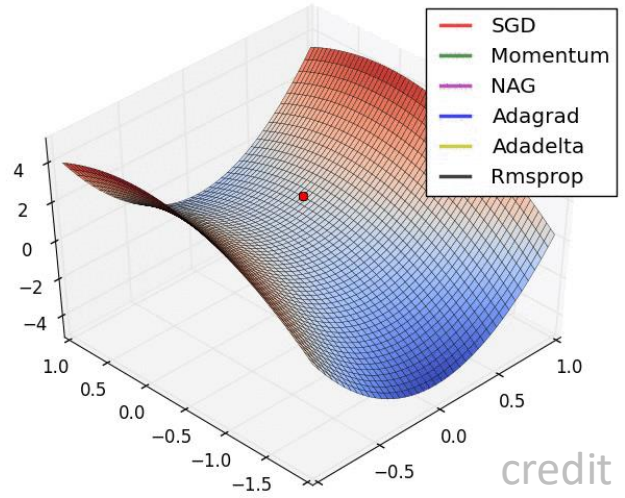
```
import torch

torch.optim.lr_scheduler.ReduceLROnPlateau(...)
optim.lr_scheduler.LambdaLR(...)
optim.lr_scheduler.StepLR(...)
optim.lr_scheduler.MultiStepLR(...)
optim.lr_scheduler.ExponentialLR(...)
optim.lr_scheduler.CosineAnnealingLR(...)
```



https://pytorch.org/docs/stable/optim.html

Optimizers

Algorithm	Pytorch Implementation	Reference
• SGD	`torch.optim.SGD(...)`	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
• Adam	`torch.optim.Adam(...)`	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Adadelta	`torch.optim.Adadelta(...)`	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adagrad	`torch.optim.Adagrad(...)`	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• RMSProp	`torch.optim.RMSProp(...)`	



credit : CS231n

https://pytorch.org/docs/stable/optim.html

Training Process

```python
import torch

#define model
model = torch.nn.Sequential(...)

#initial learning rate
learning_rate = 0.05

# optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

#lr scheduler
# lr = 0.05     if epoch < 30
# lr = 0.005    if 30 <= epoch < 80
# lr = 0.0005   if epoch >= 80
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer=optimizer,
                                                 milestones=[30,80])

for it in range(max_iter):

    predicted = model(x)

    loss = loss_fn(predicted, y)

    # Zero the gradients before running the backward pass.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss
    # with respect to all the learnable parameters of the model.
    loss.backward()

    # Update the weights using gradient descent.
    optimizer.step()

    # adjust learning_rate
    scheduler.step()
```

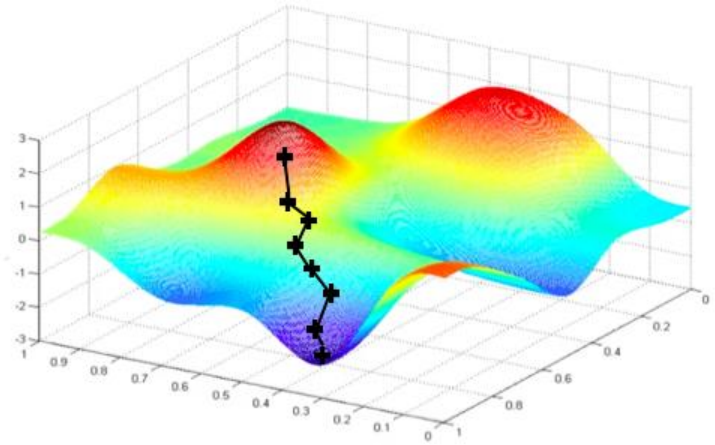
Neural Networks in Practice:

Mini-batch

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

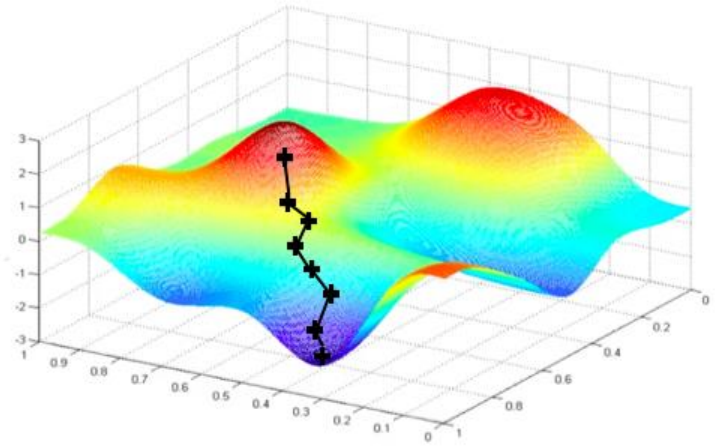


Gradient Descent

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

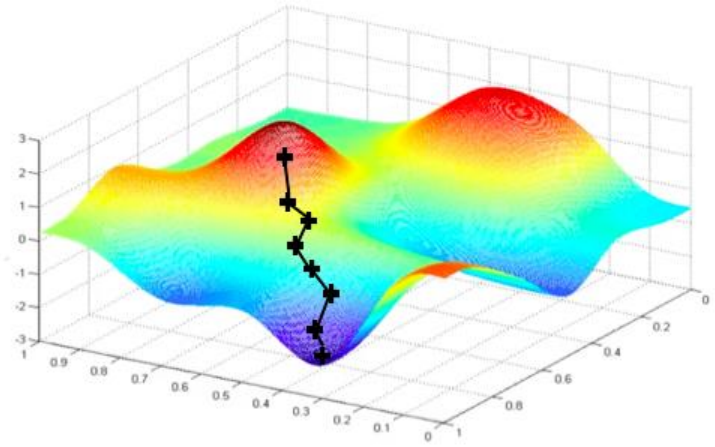
Can be very **computationally intensive** to compute!



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point 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

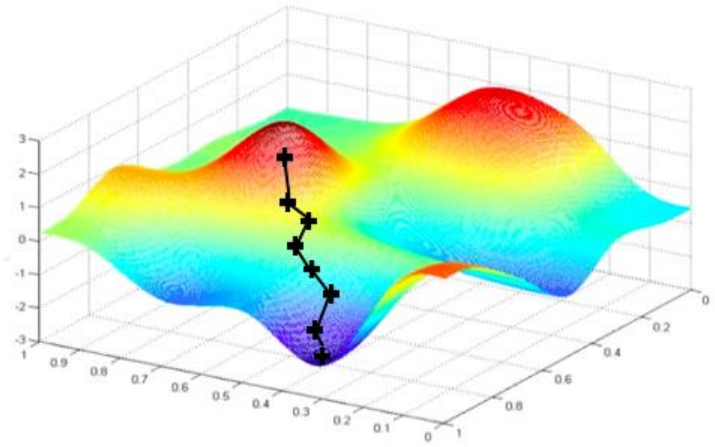


Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point 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

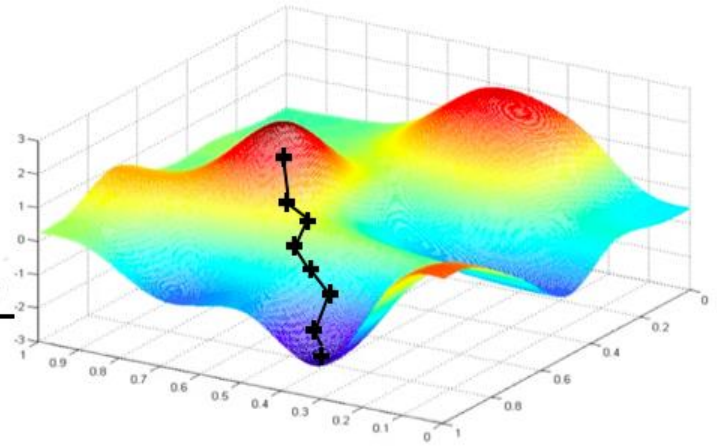
Easy to compute but
very noisy (stochastic)!



Stochastic Gradient Descent

Algorithm

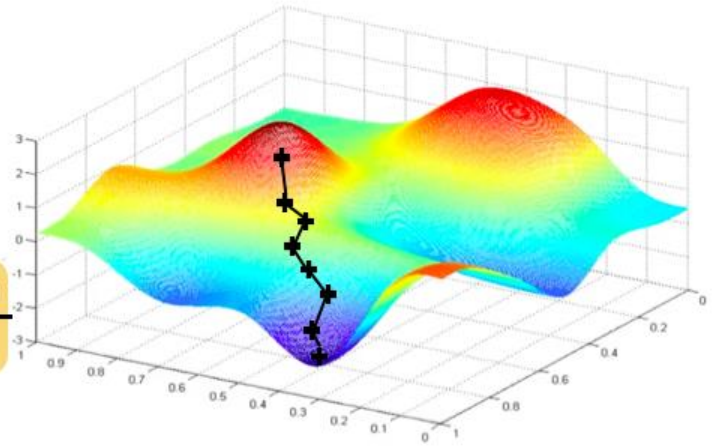
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update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

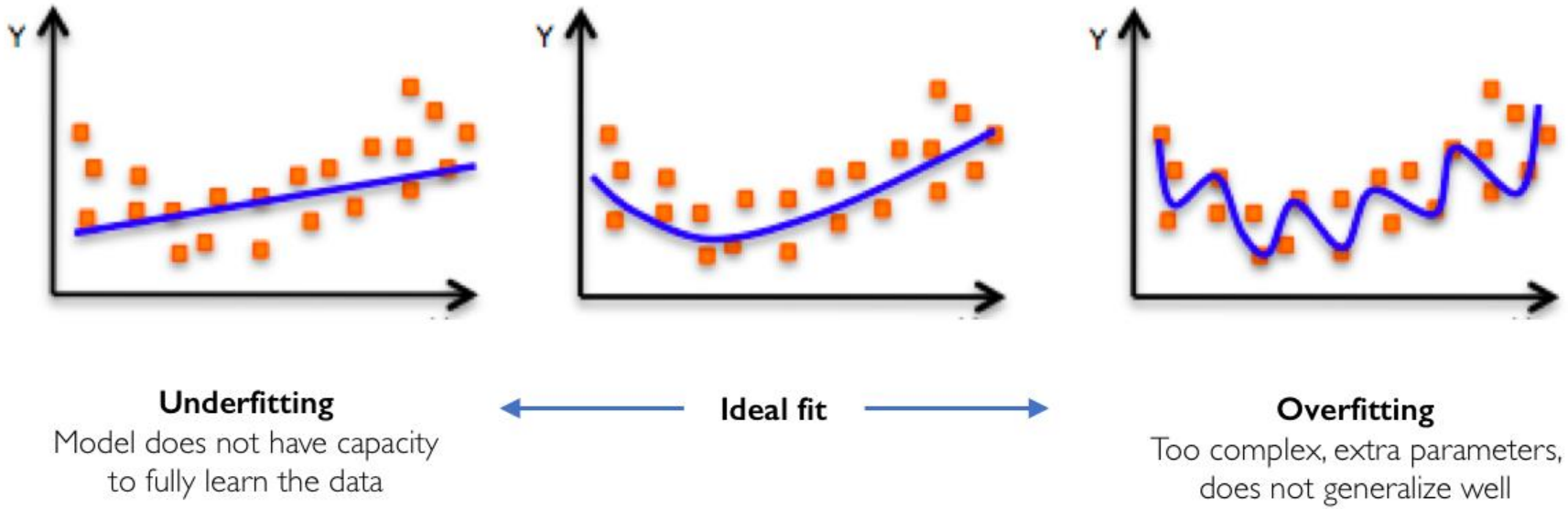
Mini-batches lead to fast training!

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

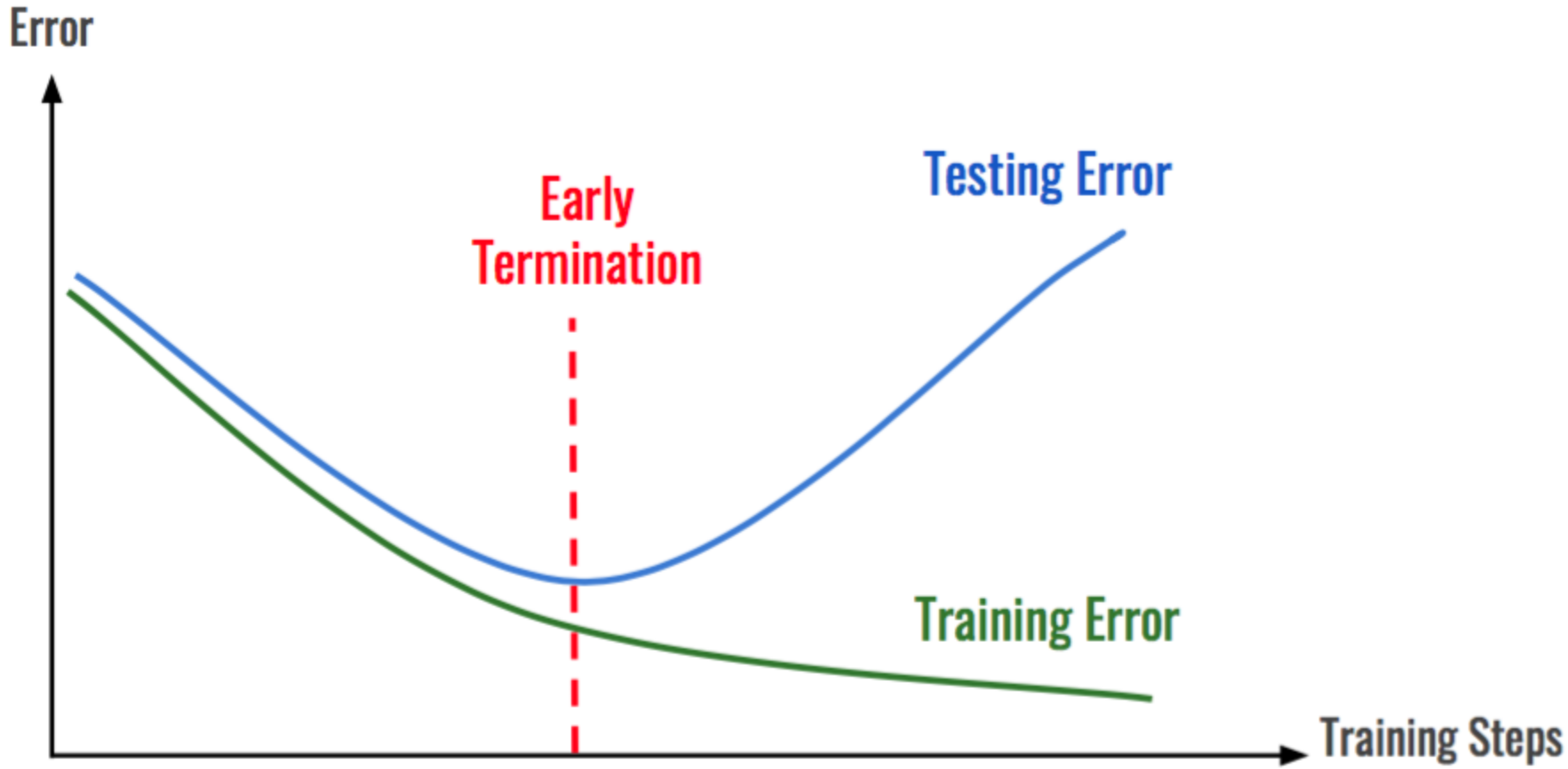
Neural Networks in Practice:

Overfitting

The problem of Overfitting



The problem of Overfitting



https://mc.ai/early-stopping-with-pytorch-to-restrain-your-model-from-overfitting/

Regularization

What is it?

Technique that **constrains** our optimization problem to **discourage** complex models

Regularization

What is it?

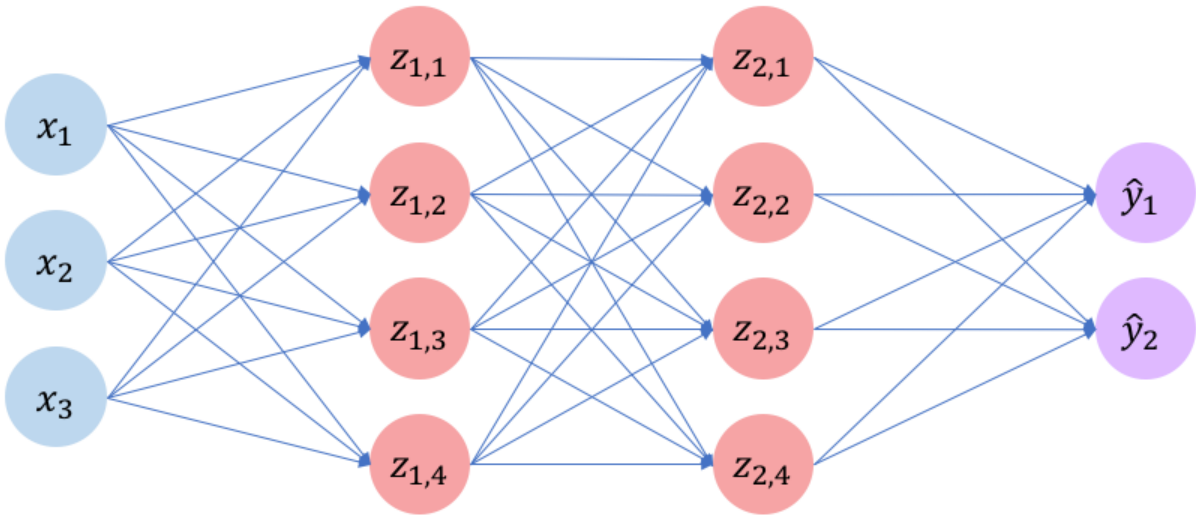
Technique that **constrains** our optimization problem to **discourage** complex models

Why we need it?

Improve **generalization** of our model on unseen data

Regularization 1 : Dropout

- During training, randomly set some activations to 0

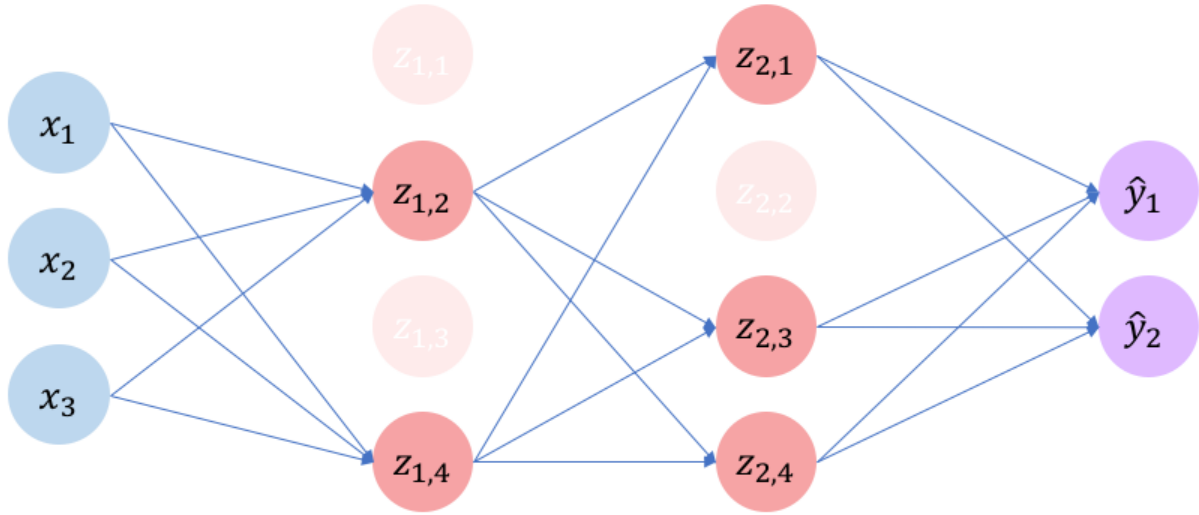


Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), pp.1929-1958.

Regularization 1 : Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
torch.nn.Dropout(p=0.5)
```

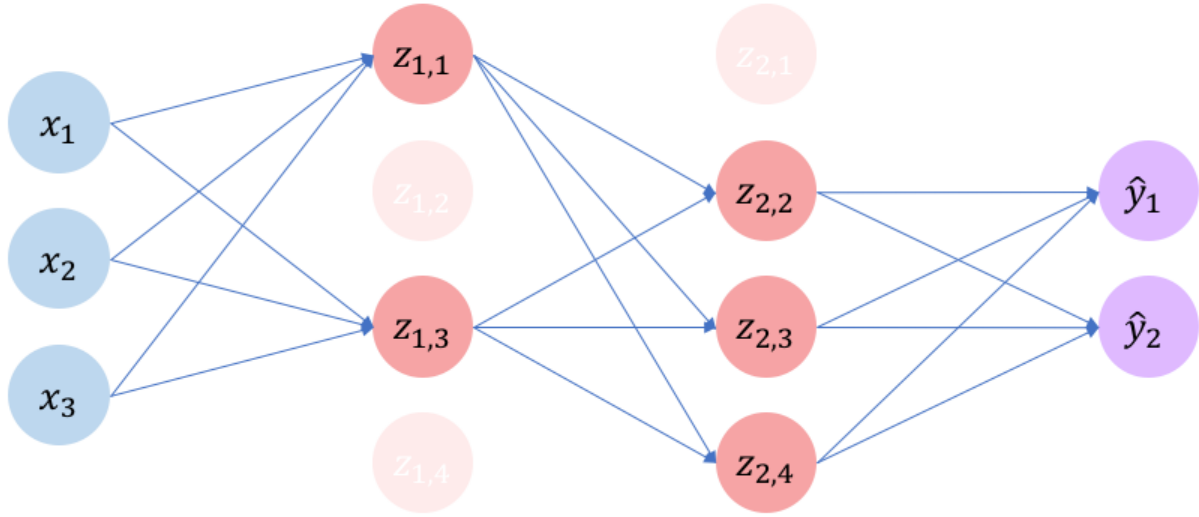


Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), pp.1929-1958.

Regularization 1 : Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
torch.nn.Dropout(p=0.5)
```



Regularization 2 : Early Stopping

- Stop training before we have a chance to overfit

Regularization 2 : Early Stopping

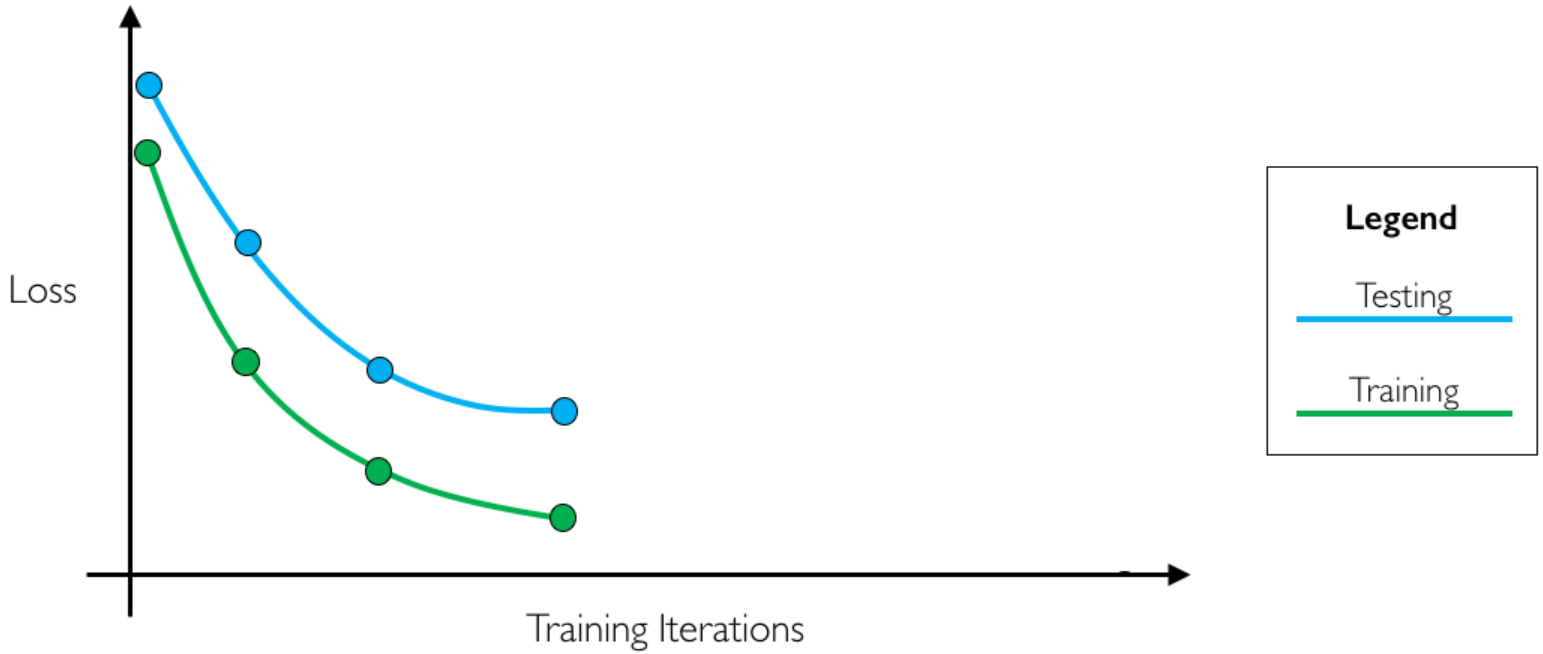
- Stop training before we have a chance to overfit

Regularization 2 : Early Stopping

- Stop training before we have a chance to overfit

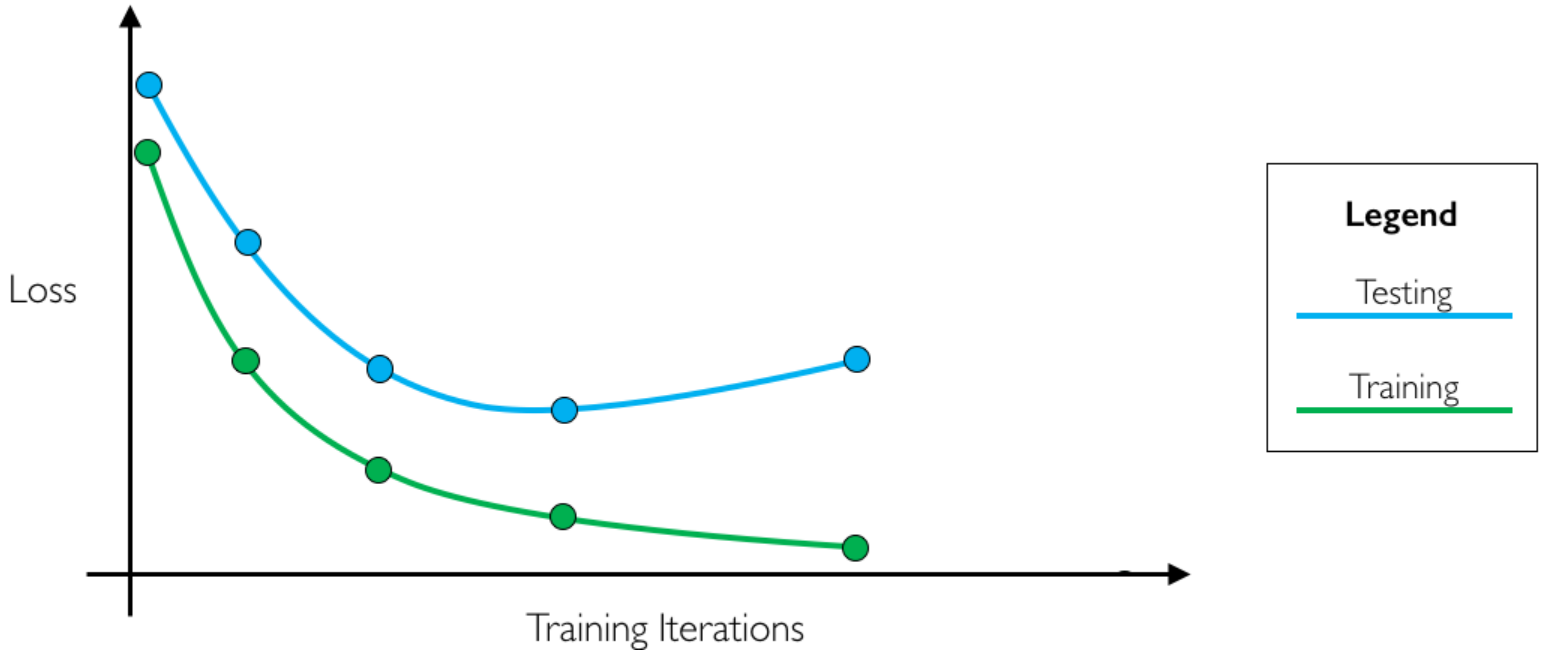
Regularization 2 : Early Stopping

- Stop training before we have a chance to overfit



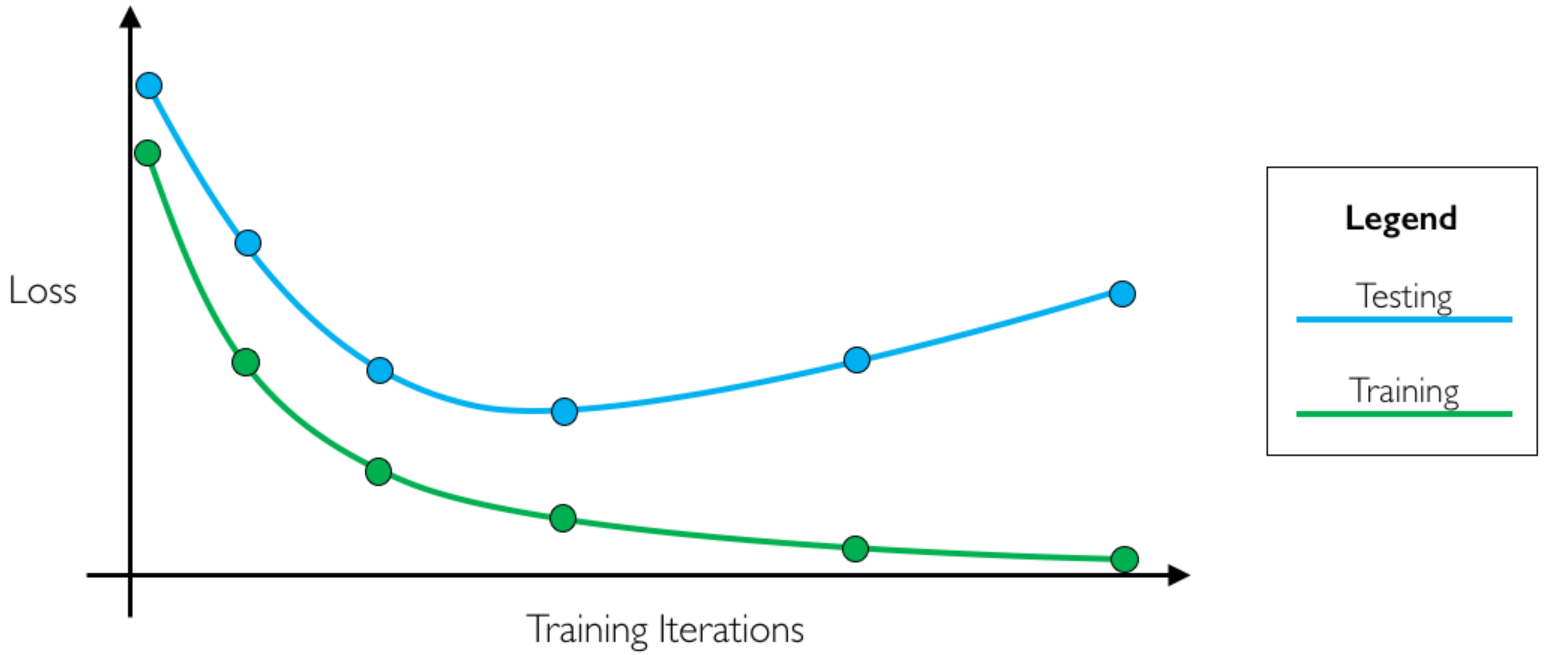
Regularization 2 : Early Stopping

- Stop training before we have a chance to overfit



Regularization 2 : Early Stopping

- Stop training before we have a chance to overfit



Regularization 2 : Early Stopping

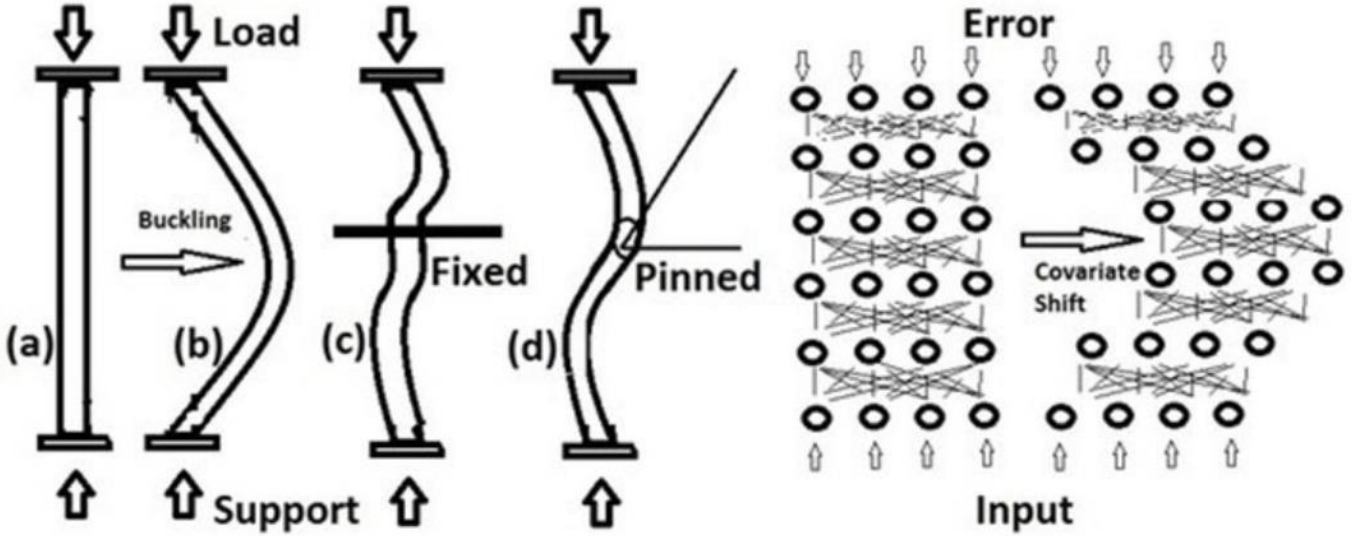
- Stop training before we have a chance to overfit

Regularization 2 : Early Stopping

- Stop training before we have a chance to overfit

Regularization 3 : Batch normalization

"Covariate Shift" problem



For both, Buckling or Co-Variate Shift a small perturbation leads to a large change in the later.

Debiprasad Ghosh, PhD, Uses AI in Mechanics

Ioffe, S. and Szegedy, C., 2015, June. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

Regularization 3 : Batch normalization

```
torch.nn.BatchNorm2d(num_features=num_features)
```

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

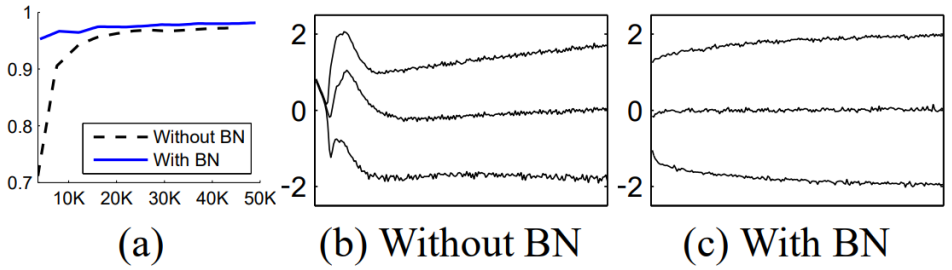
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

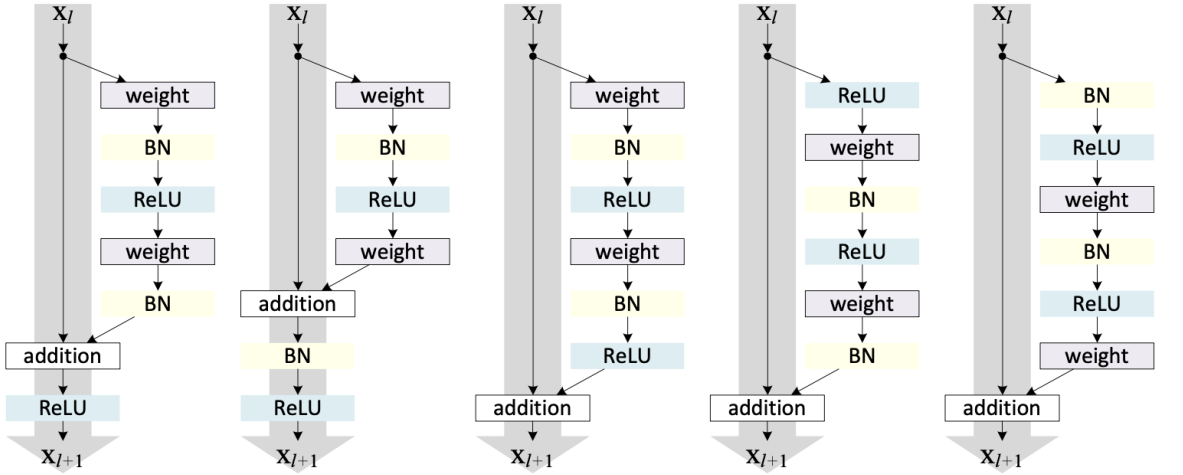


(a) (b) Without BN (c) With BN

Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Ioffe, S. and Szegedy, C., 2015, June. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

Regularization 3 : Batch normalization



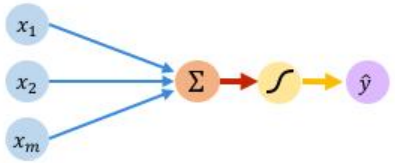

```
conv1 = torch.nn.Sequential{
    torch.nn.Conv2d(...),
    torch.nn.BatchNorm2d(num_features),
    torch.nn.relu(...)
}
```

He, K., Zhang, X., Ren, S. and Sun, J., 2016, October. Identity mappings in deep residual networks. In *European conference on computer vision* (pp. 630-645). Springer, Cham.

Core Foundation Review

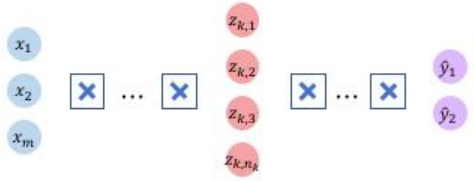
The Perceptron

- Structural building blocks
- Nonlinear activation functions



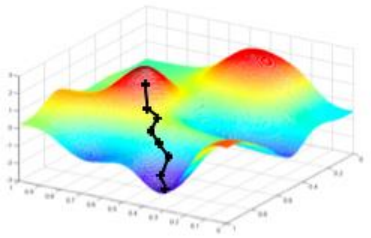
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



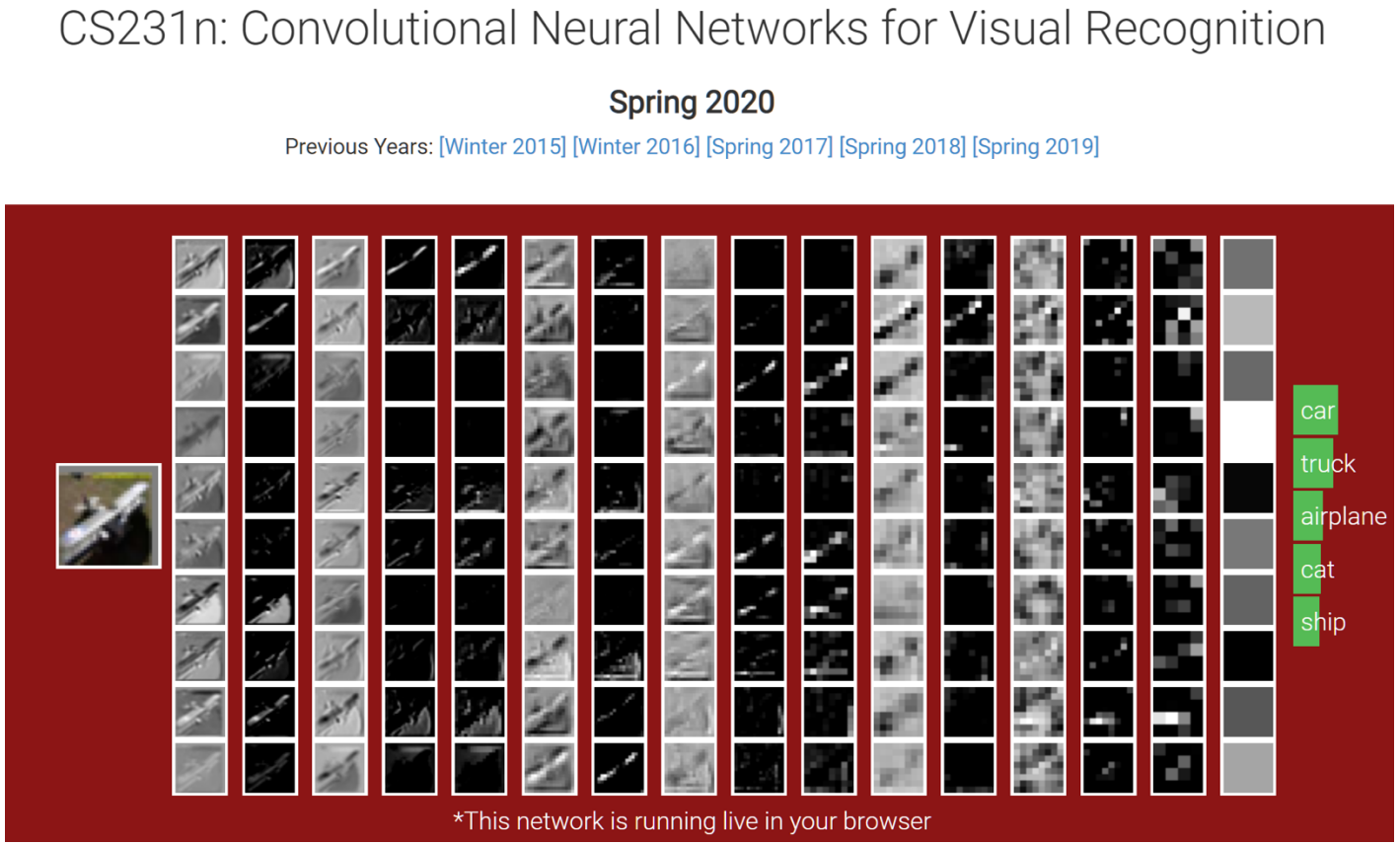
Training in Practice

- Adaptive learning
- Batching
- Regularization



추천 학습 자료

- Stanford 231n by Fei-Fei Li



http://cs231n.stanford.edu/index.html

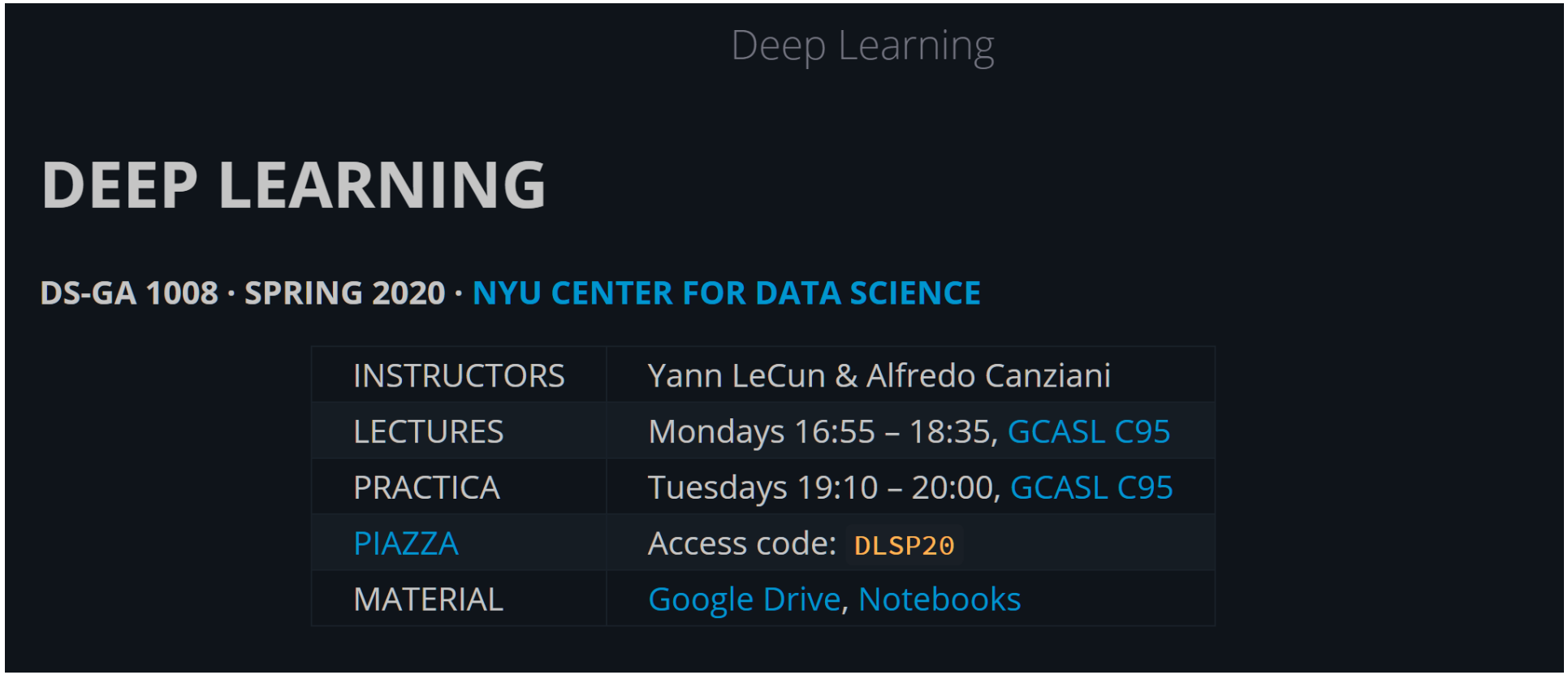
추천 학습 자료

- Stanford cs224n by Chris Manning

http://web.stanford.edu/class/cs224n/index.html#schedule

추천 학습 자료

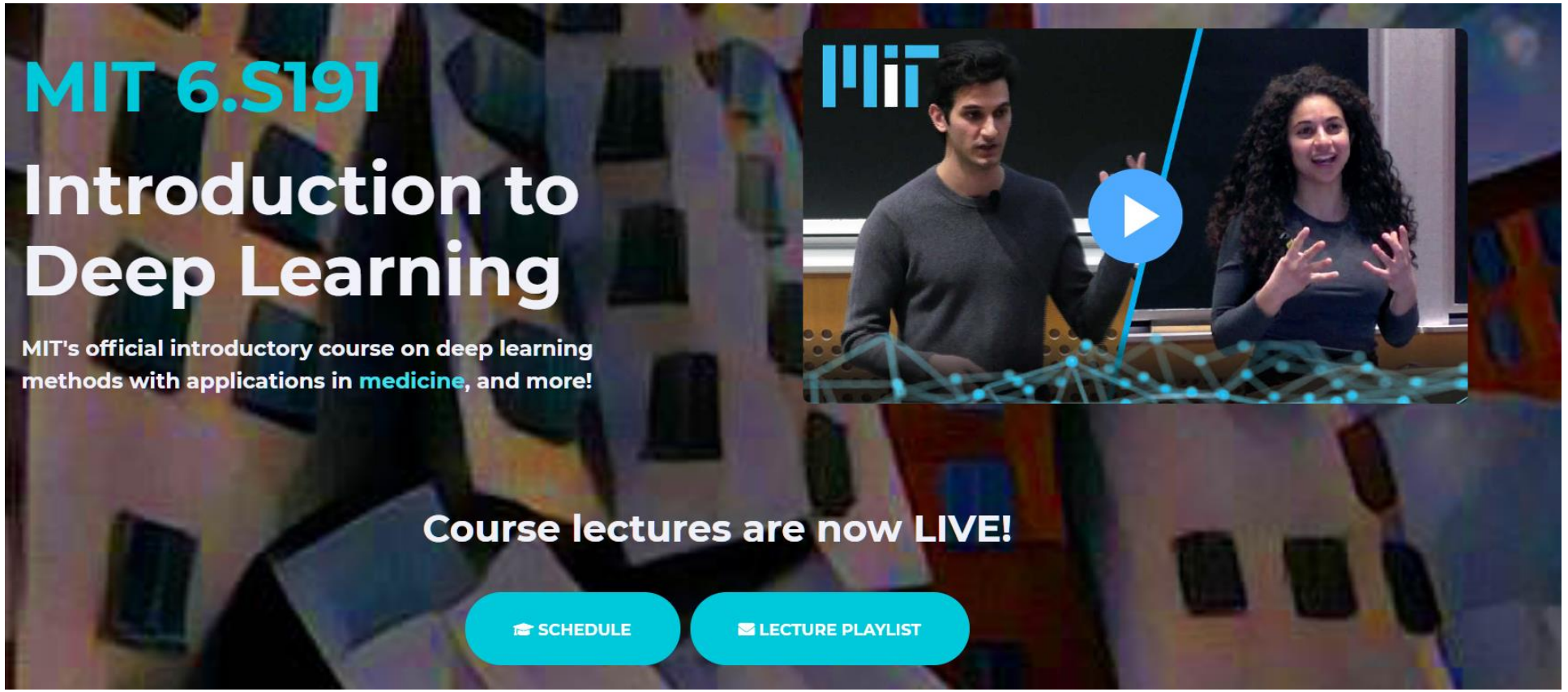
- NYU DEEP LEARNING by Yann Lecun



https://atcold.github.io/pytorch-Deep-Learning/

추천 학습 자료

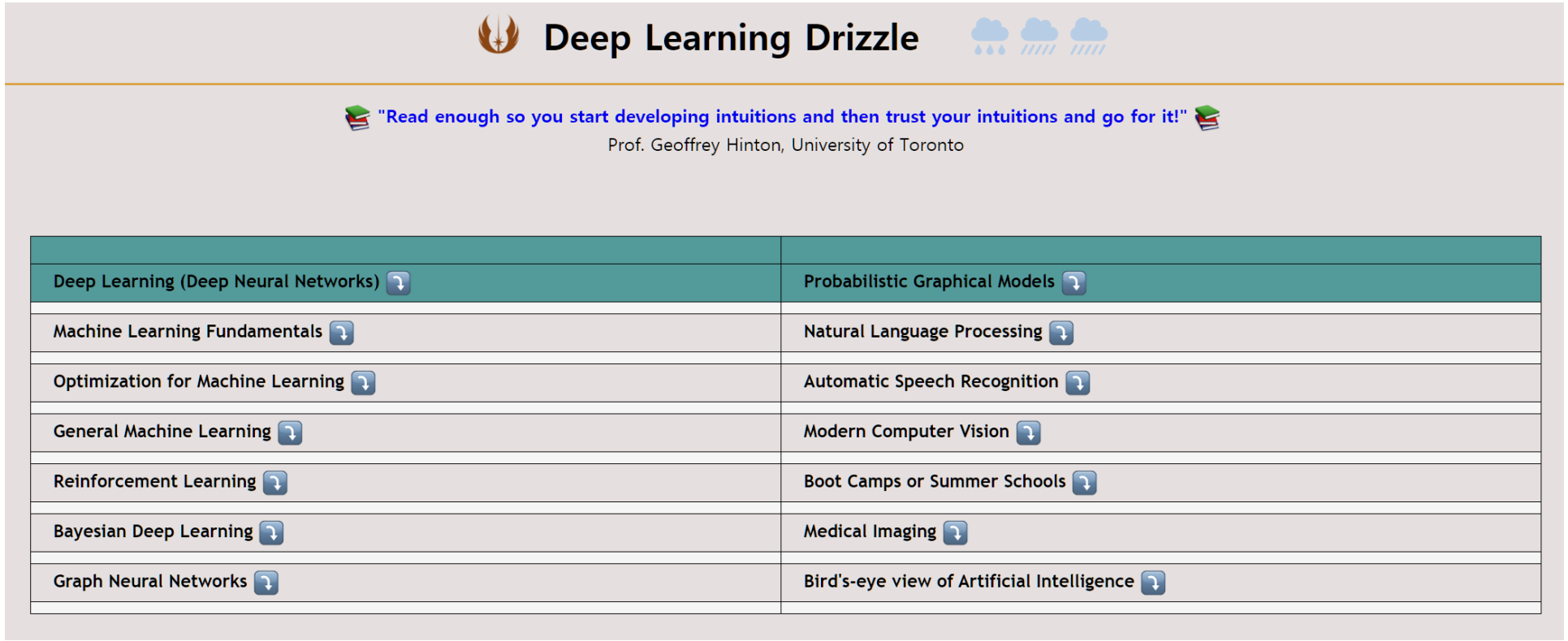
- MIT



http://introtodeeplearning.com/?fbclid=IwAR1_iEF374Jg46s_2HMVWcVPFGwZWorBxEL25jzfpFfgcIFWbo-BDWUv_54

추천 학습 자료

- https://deep-learning-drizzle.github.io/index.html#dldnn



Deep Learning Drizzle

"Read enough so you start developing intuitions and then trust your intuitions and go for it!"
Prof. Geoffrey Hinton, University of Toronto

Deep Learning (Deep Neural Networks)	Probabilistic Graphical Models
Machine Learning Fundamentals	Natural Language Processing
Optimization for Machine Learning	Automatic Speech Recognition
General Machine Learning	Modern Computer Vision
Reinforcement Learning	Boot Camps or Summer Schools
Bayesian Deep Learning	Medical Imaging
Graph Neural Networks	Bird's-eye view of Artificial Intelligence

References

- http://cs231n.stanford.edu/index.html
- http://web.stanford.edu/class/cs224n/index.html#schedule
- https://atcold.github.io/pytorch-Deep-Learning/
- http://introtodeeplearning.com/?fbclid=IwAR1_iEF374Jg46s_2HMVWcVPFGwZWorBxEL25jzfpFfgcIFWbo-BDWUv_54