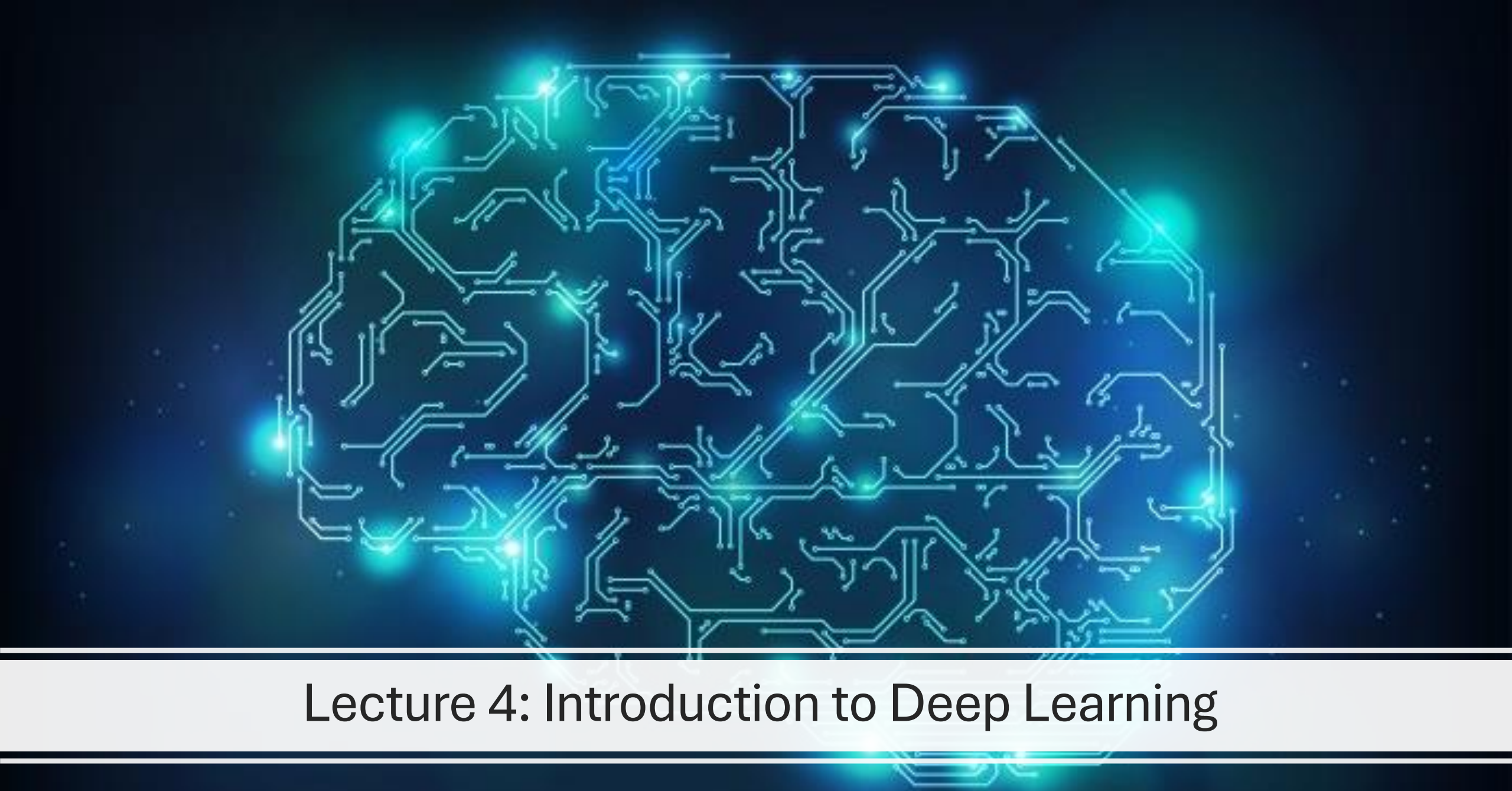
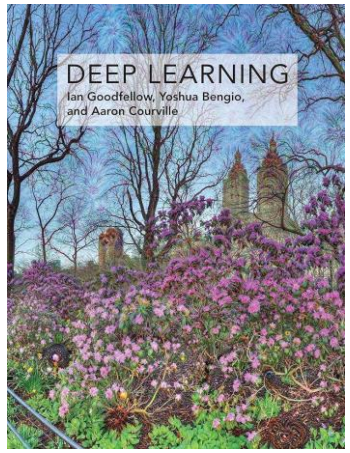
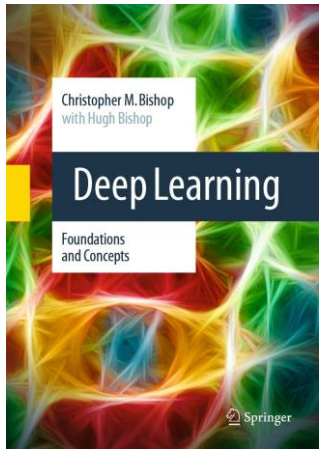


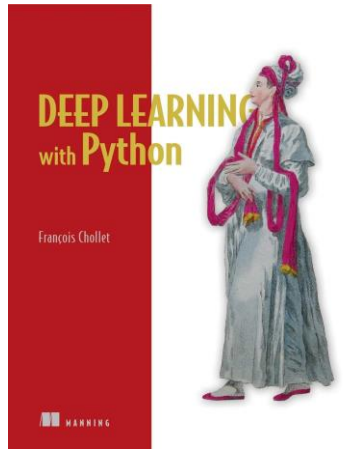
Lecture 4: Introduction to Deep Learning

Resources

- Deep Learning. Goodfellow, Benjio, Courville. MIT Press, 2016
- Deep Learning: Foundations and concepts. Bishop & Bishop. Springer, 2023
- Deep Learning with Python. Chollet. Manning, 2023
- MIT Deep Learning course (6.S191)







What we expect for you

- You know basic concepts of deep learning
- You know a few basic algorithms/models of deep learning
- You initially know how to leverage deep learning to solve your own problems

Outline

- Session 1: Introduction to deep learning
- Session 2: Convolutional neural network (CNN) and recurrent neural network (RNN)
- Session 3: Introduction to deep learning packages

Outline

- Session 1: Introduction to deep learning
- Session 2: Convolutional neural network (CNN) and recurrent neural network (RNN)
- Session 3: Introduction to deep learning packages

Deep learning everywhere

Image credit: Nvidia

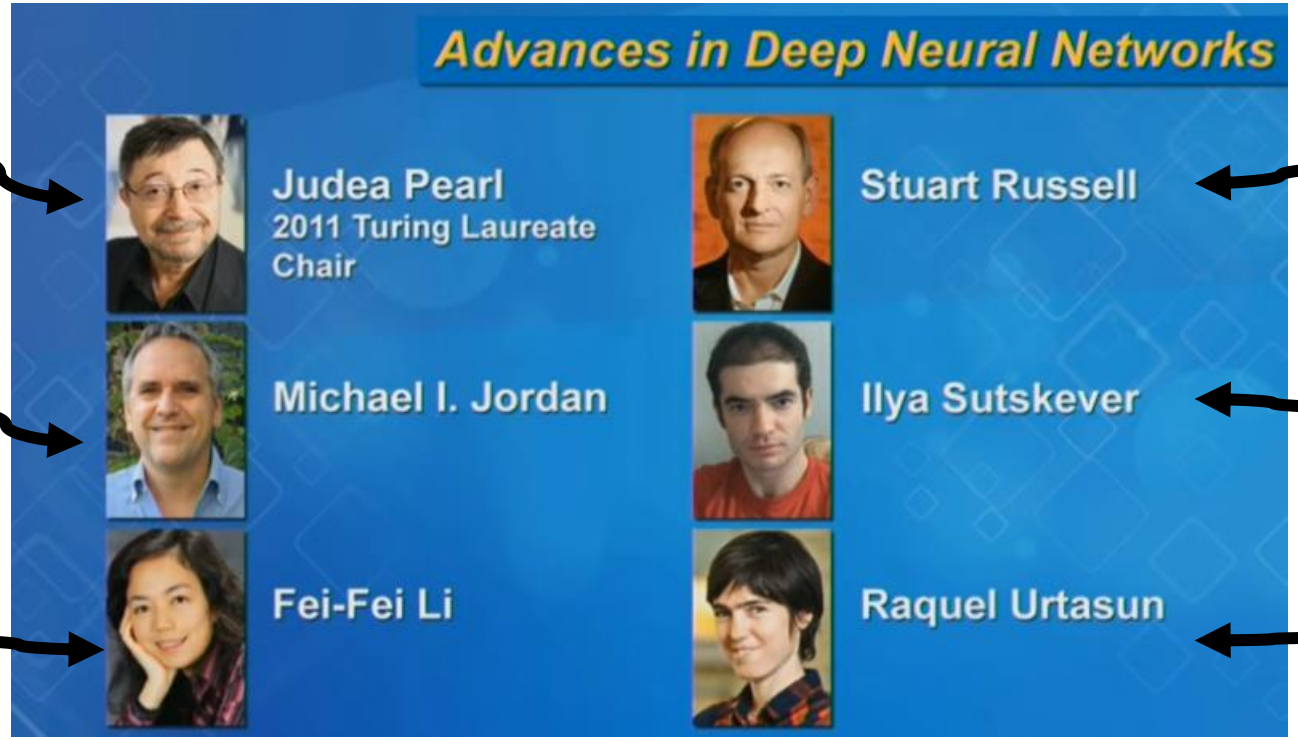
Godfathers of deep learning, 2018 Turing Awardees

Deep learning and Turing

Professor at UCLA, creator of Bayesian networks

Professor at UC Berkeley, the world's most influential computer scientist in 2016

Professor at Stanford, creator of ImageNet



Professor at UC Berkeley, author of Artificial Intelligence: A Modern Approach

Former chief scientist at OpenAI, one of creators of AlexNet, ChatGPT, AlphaGo

Professor at U Toronto, known for machine vision in self-driving car

ACM Turing 50 Celebration

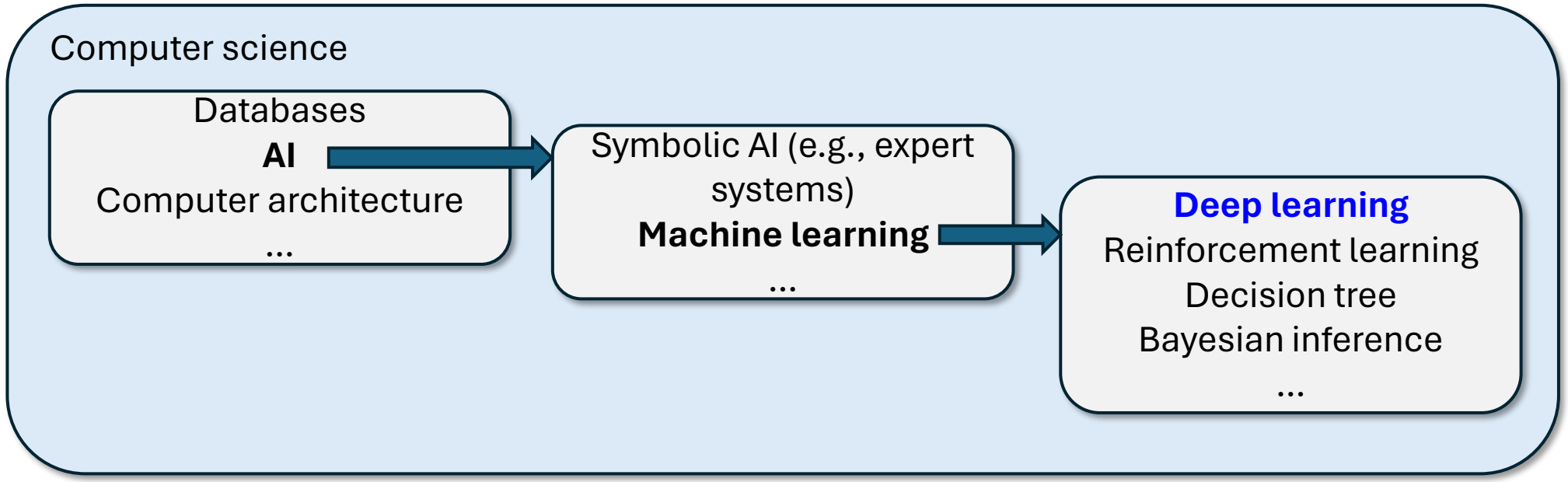
Deep learning packages

- Most popular deep learning packages are Python-based
  - Tensorflow
  - ✓**Pytorch**
  - Keras
  - …

- We use Pytorch for all sample code
  - https://pytorch.org/
  - Installation is simple, but please pay attention to the version consistency with Python

Deep learning

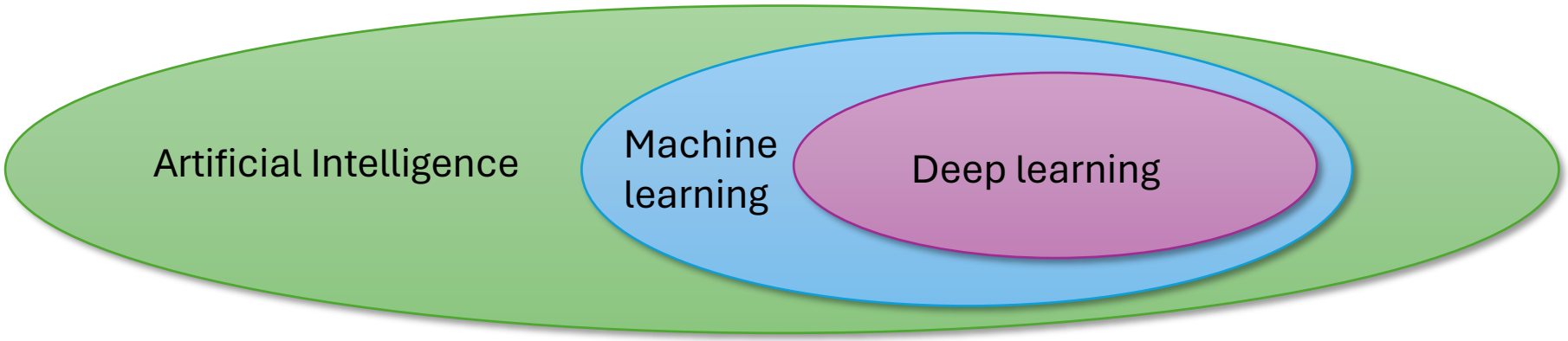
- What does deep learning belong to?



Deep learning is a branch of machine learning

What is deep learning?

- Deep learning allows computational models that are composed of **multiple processing layers** **to learn representations of data** with **multiple levels of abstraction**

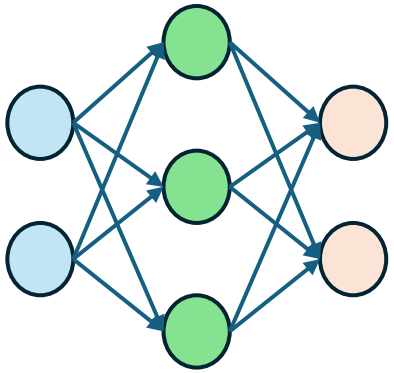


Deep learning by LeCun et al. Nature, 2015

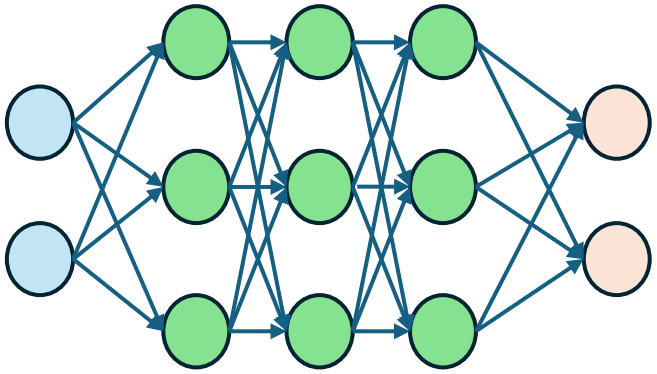
DL vs. traditional ML

- How to differentiate from existing ML methods?
  - Deep learning specifies different types of deep neural networks
    - Deep here essentially means "multiple" hidden layers (≥ 2)
  - One significant difference is the number of layers (sometimes the number of parameters)
- Only neural networks
  - You are not going to get deep support vector machine, even if you have multiple support vector machines

Artificial Neural Network



Deep Neural Network



Deep vs. Shallow

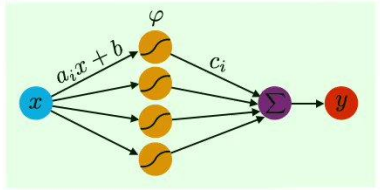
- Universality: artificial neural network (ANN) can represent any continuous function to arbitrary precision, given a large enough # of units
  - Poor expressivity
- Two motivations for deep nets:
  - **Statistical**: deep nets are compositional, and good for hierarchical structures where simpler patterns are composed and reused to form more complex ones recursively
  - **Computational**: deep architectures are more **expressive** than shallow ones

Why (deep) neural networks can approximate

- **Universal Approximation Theorem** *(Informal): Given a family of neural networks, for each function f from a certain function space, there exists a sequence $\phi_1, \phi_2, \ldots$ from the family, such that $\phi_n \rightarrow f$ according to some criteria*
- Universal approximation theorem is an existence theorem
  - It doesn't provide a way to find out the sequence $\phi_1, \phi_2, \ldots$
- It also doesn't guarantee any method like backpropagation to find such a sequence

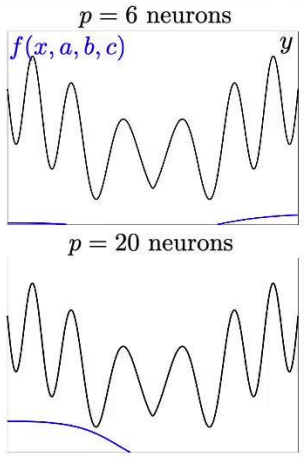
Quick proof

- The proof for the theorem can be achieved by combining Hahn-Banach Theorem and Riesz Represention Theorem
- Let's cite the argument from George Cybenko to quickly prove this



1 hidden layer perceptron:

$y \approx f(x, a, b, c) \stackrel{\text{def.}}{=} \sum_{i=1}^{p} c_i \varphi(a_i x + b_i)$

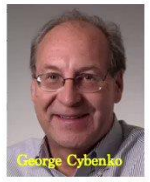


Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.



"If the theorem is false, then donkeys may fly, but donkeys cannot fly, so the theorem must be true"

End-to-end manner

- End-to-end: directly from raw input to output

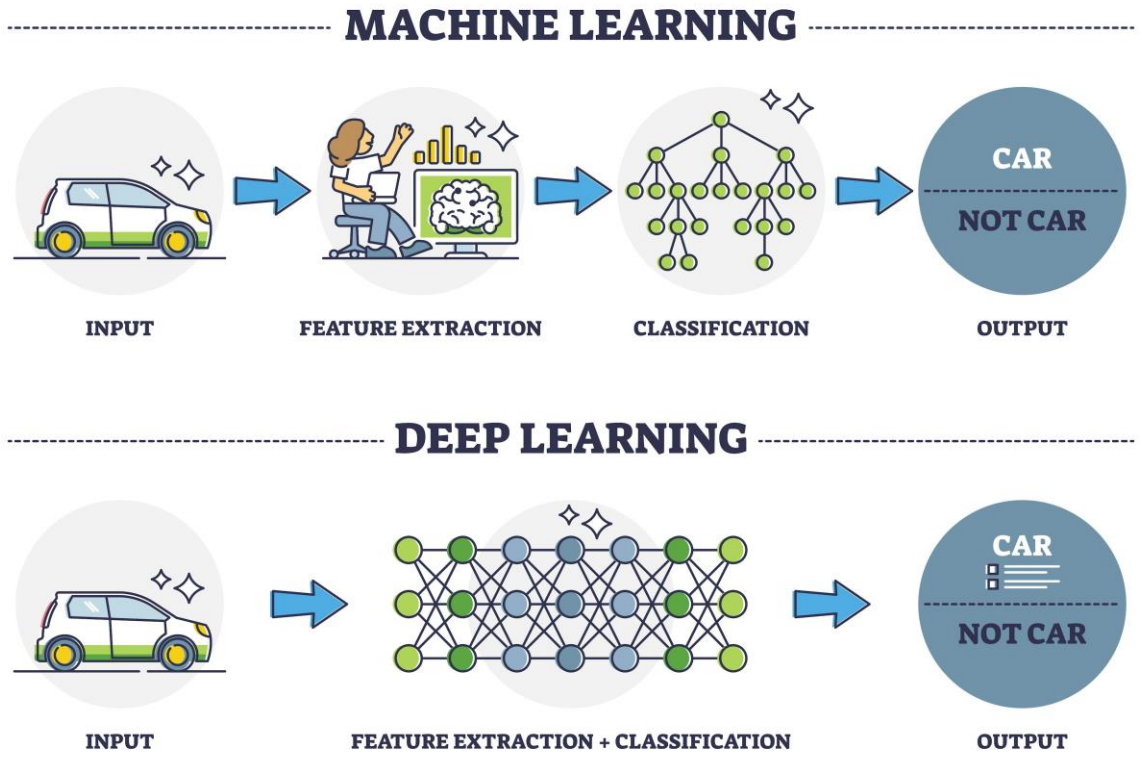
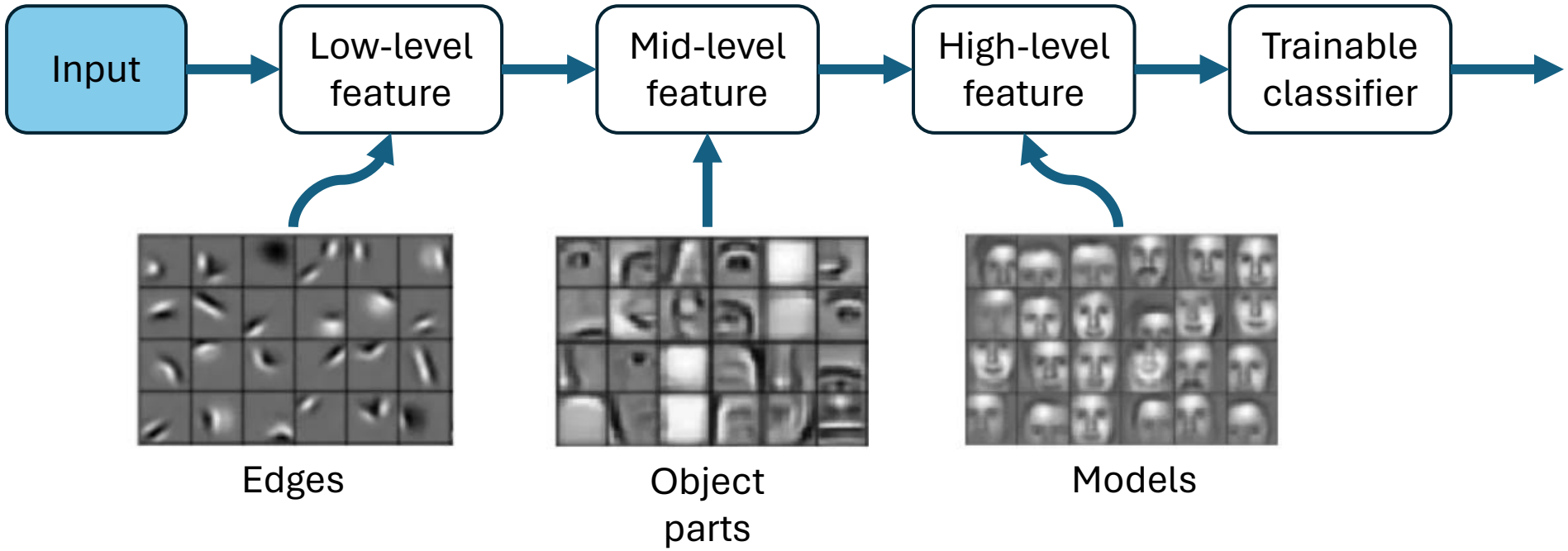


Image credit: Turing

Deep learning: learn representations

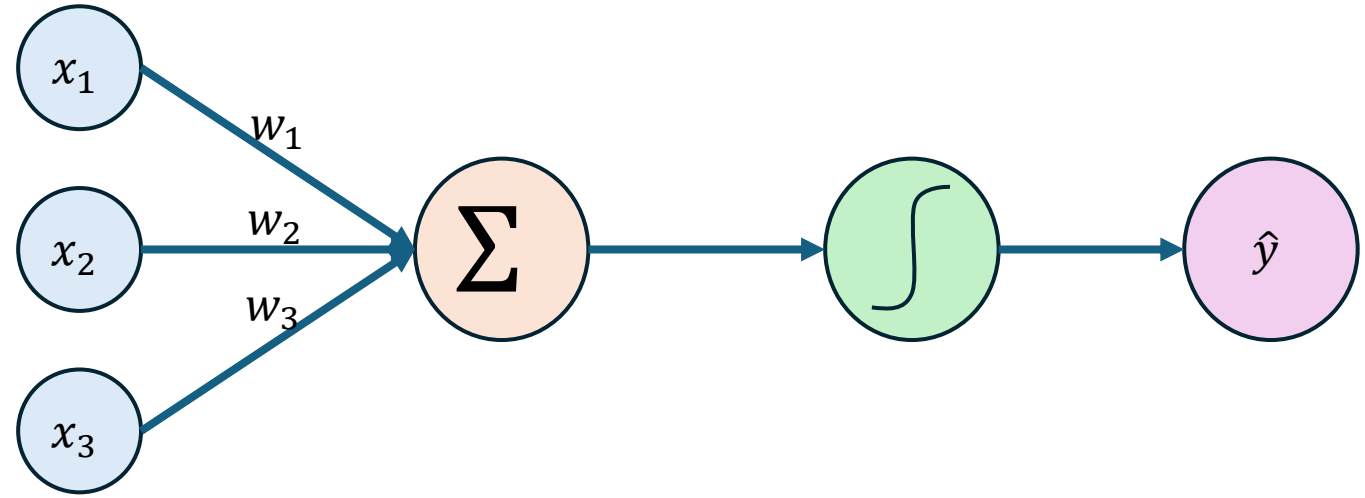


- The series of layers between input & output do feature identification and processing in a series of stages

Image credit: MIT 6.S191

But how to learn

- Let's start with the Perceptron



Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{3} w_i x_i\right)$$

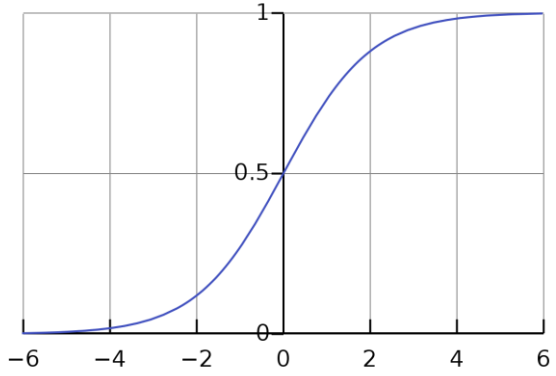
Non-linear activation function

- We skip bias for simplicity
- Activation function matters
- A more compact way can be used, such as vector form

Some common activation functions

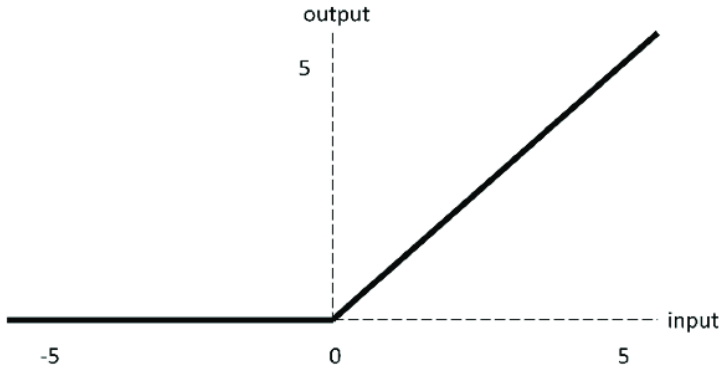
- All activation functions are non-linear

```
import torch.nn.functional as F
```



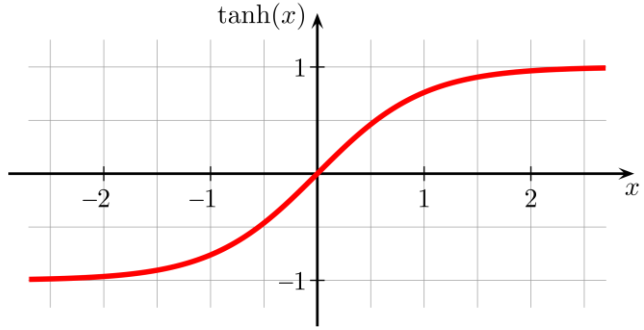
$$f(x) = \frac{1}{1 + e^{-x}}$$

```
F.sigmoid (x)
```



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

```
F.relu (x)
```

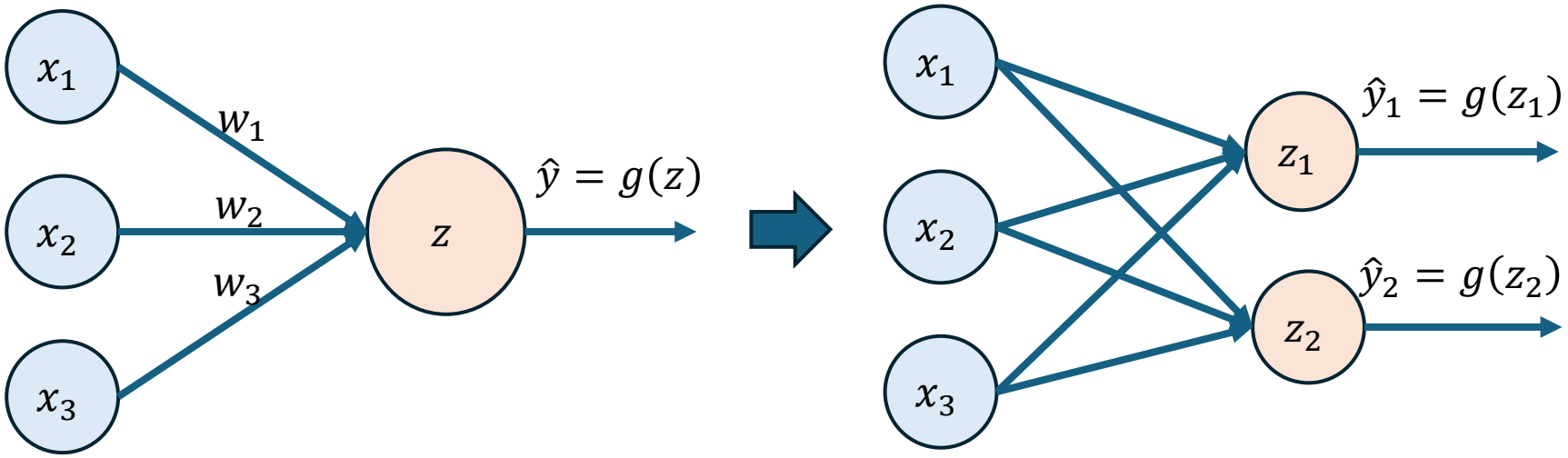


$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

```
F.tanh (x)
```

Multi-output Perceptron

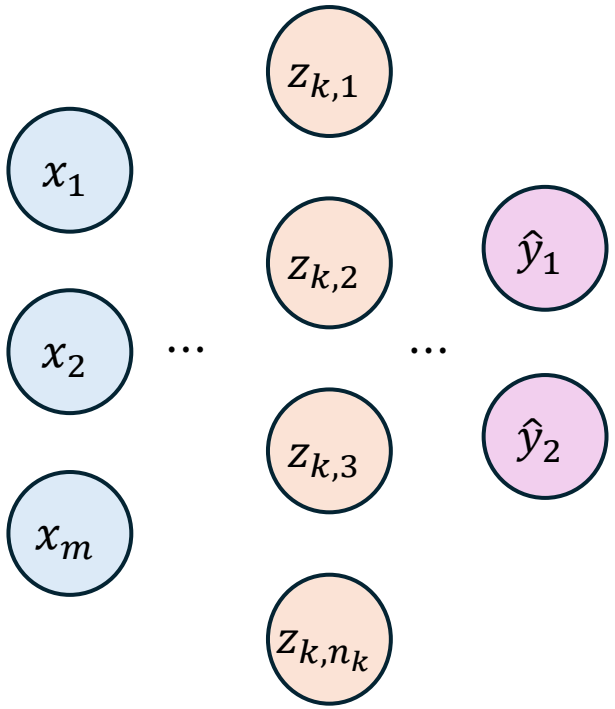
- All inputs are densely connected to all outputs, so these layers are called **Dense** layers



$$z = \sum_{i=1}^{3} w_i x_i + b$$

$$z_i = \sum_{j=1}^{3} w_{j,i} x_j$$

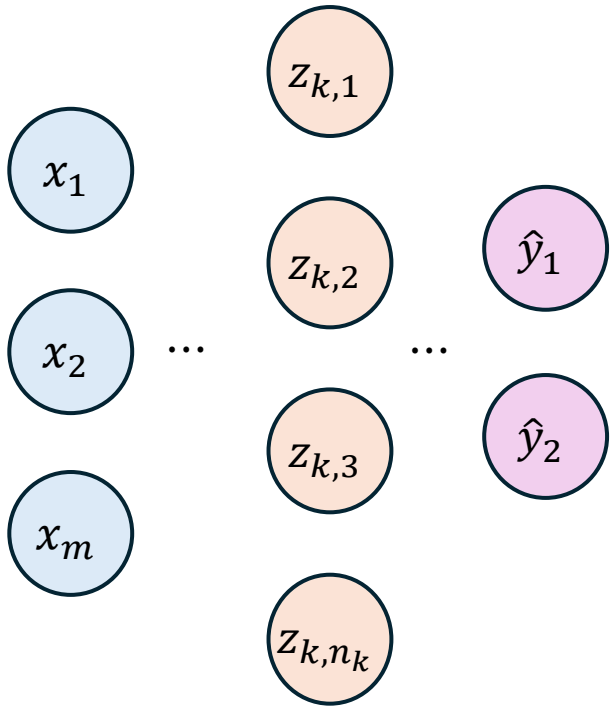
Multi-layer Perceptron (MLP)



$$z_{k,i} = \sum_{j=1}^{n_k} g\left(z_{k-1,j}\right) w_{j,i}^{k}$$

```
import torch.nn as nn
Import torch.nn.functional as F
class DenseNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Construct layers.
        self.dn1 = nn.Linear(input_dim, hidden_dim)
        self.dn2 = nn.Linear(hidden_dim, hidden_dim)
        self.dn3 = nn.Linear(hidden_dim, output_dim)
    def forward(self, x):
        # Add activation function.
        x = F.relu(self.dn1(x))
        x = F.relu(self.dn2(x))
        x = F.log_softmax(self.dn3(x), dim=1)
        return x
```

Optimization helps

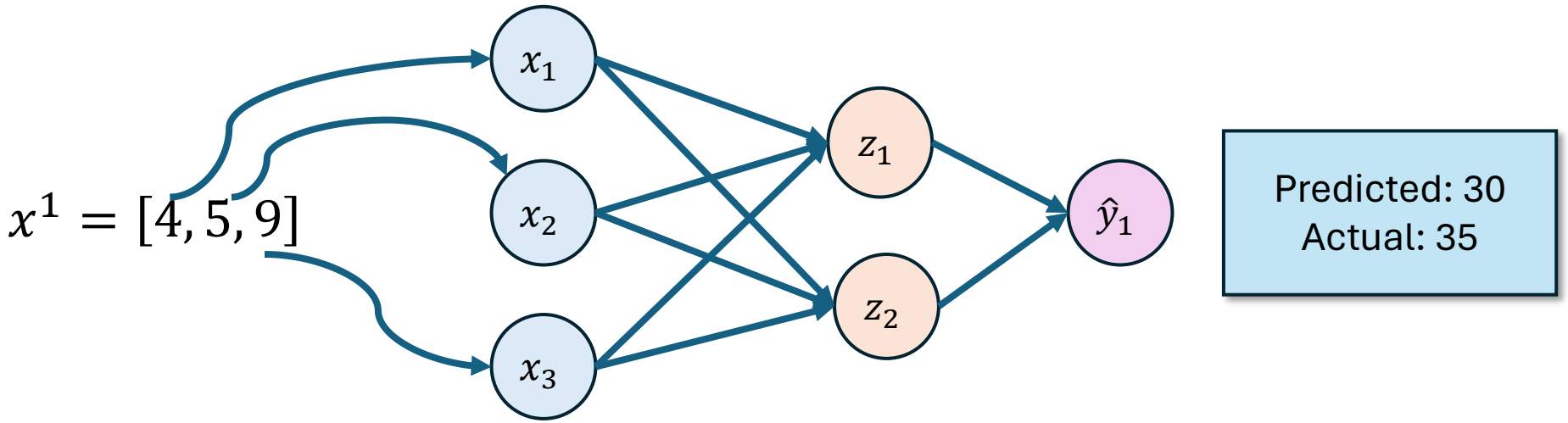


$$z_{k,i} = \sum_{j=1}^{n_k} g\left(z_{k-1,j}\right) w_{j,i}^{k}$$

But how to find the best $w_{j,i}^{k}$ for the model? Optimization!

Quantifying loss

- The loss of neural network measures the cost incurred from incorrect predictions



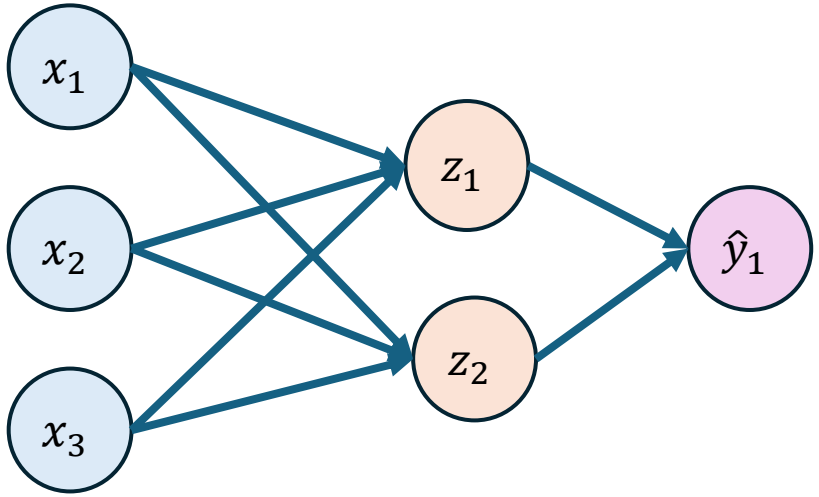
Loss w.r.t a data sample $\mathcal{L}\left(\underbrace{f\left(x^i; W\right)}_{\text{Predicted}}, \underbrace{y^i}_{\text{Actual}}\right)$

Empirical loss

But why empirical?

- The empirical loss measures the total loss over the entire dataset

$$\boldsymbol{X} = \begin{bmatrix} \mathbf{4\ 5\ 9} \\ \vdots \\ \mathbf{10\ 8\ 7} \end{bmatrix}$$



$$f(x) \quad y$$
$$\begin{bmatrix} \mathbf{30} \\ \vdots \\ \mathbf{10} \end{bmatrix} \quad \begin{bmatrix} \mathbf{35} \\ \vdots \\ \mathbf{12} \end{bmatrix}$$

A.K.A, empirical risk, objective function, cost function

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

Form for $\mathcal{L}$

- Classification: cross entropy loss
- Regression: mean squared error

`torch.nn.functional.cross_entropy`

`torch.nn.functional.mse_loss`

Optimizing empirical loss

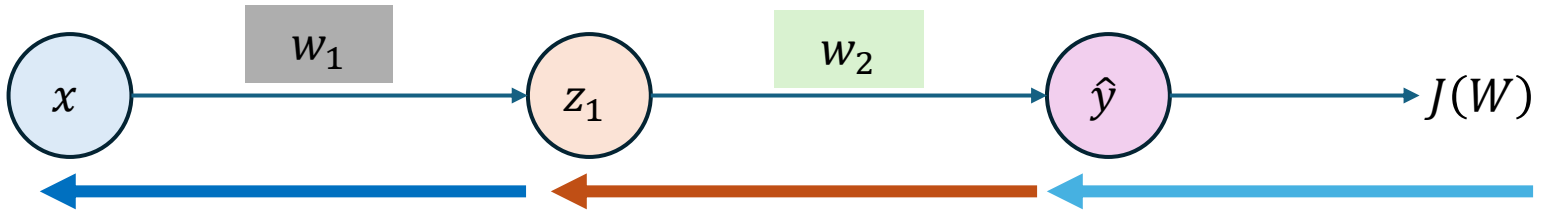
- Gradient-based algorithms have been the most popular type to optimize the empirical loss induced by deep learning models

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

- In neural networks, we use backpropagation to compute the gradient

How does a small change in one weight affect the final loss?

Backpropagation

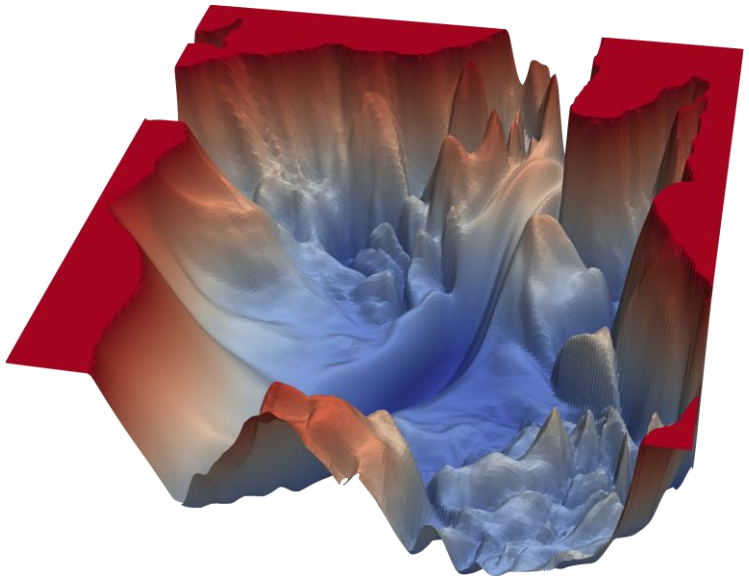


$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

Neural Network can be difficult to train



Loss landscape for VGG
Highly nonconvex

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

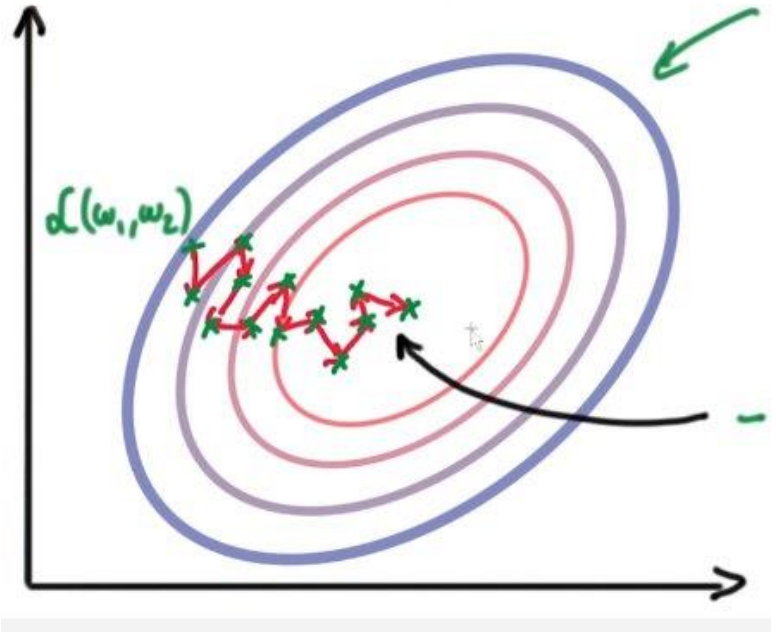
Learning rate

- Small learning rate converges slowly and gets stuck at bad local minima
- Large learning rate may result in instability and divergence
- Key hyperparameter
- Convergence theory behind this is an active research direction
- Empirical remedy: trying lots of learning rates or using adaptive learning rate

Gradient descent algorithms

- SGD
- Adam
- AdaGrad
- AdaDelta
- RMSprop
- …

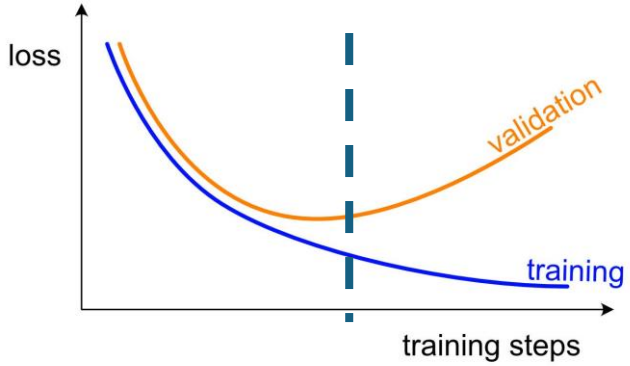
```
torch.optim.SGD
torch.optim.Adam
torch.optim.Adagrad
torch.optim.Adadelta
torch.optim.RMSprop
```



- In practice, we split the data to multiple mini-batches and update the parameters multiple times in one iteration (a pass over the whole dataset)
- This can improve the computational speed, while leading to stochasticity
- We will see detailed examples in Session 3

Issues from deep nets

- A large amount of model parameters
- A large amount of dataset
- Overfitting
- High computational complexity
- Poor model interpretability



Remedy

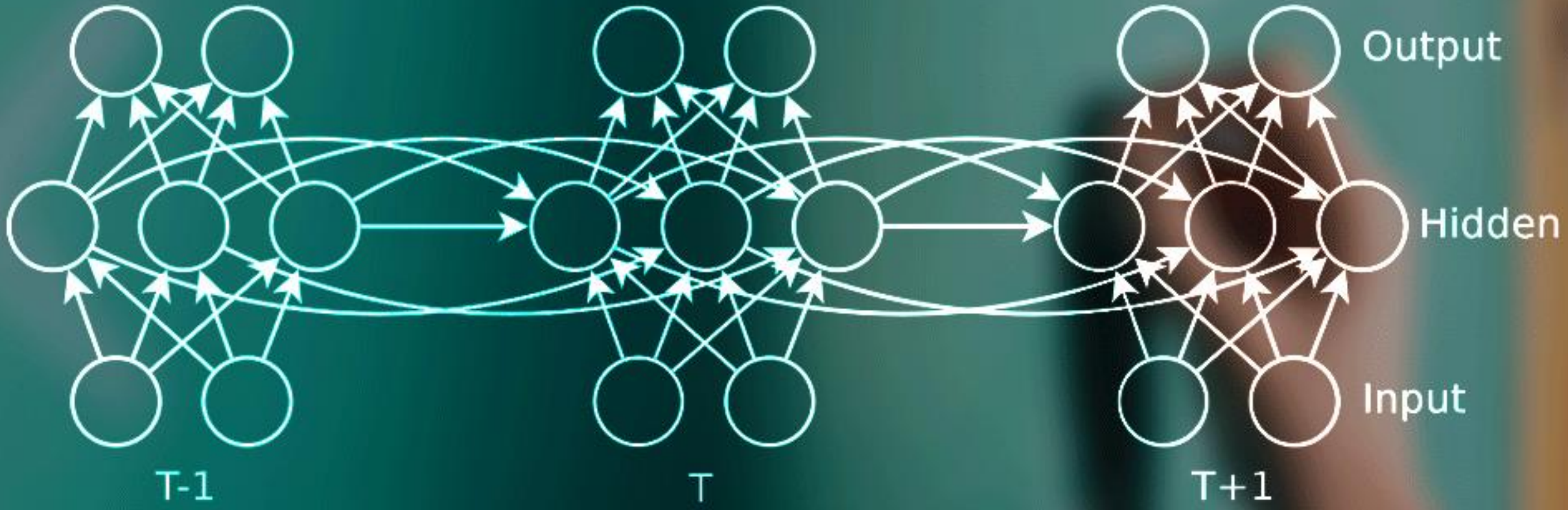
- Regularization
  - Discourage complex models
  - E.g., Dropout
- Improve data quality while trying to reduce quantity
- Early stopping
- Use GPUs and parallelize
- Develop interpretable deep nets with domain knowledge

Outline

- Session 1: Introduction to deep learning
- Session 2: Convolutional neural network (CNN) and recurrent neural network (RNN)
- Session 3: Introduction to deep learning packages

Models covered in this session

- Convolutional neural network (CNN)
- Recurrent neural network (RNN)
  - Long-short term memory (LSTM)
- We don’t dive deep into more advanced models as most of them are based on the above models
- CNN & RNN have been foundations for most state-of-the-art models
  - They are the only two models introduced in the paper (*LeCun, Bengio and Hinton. Deep learning. Nature, 2015*)

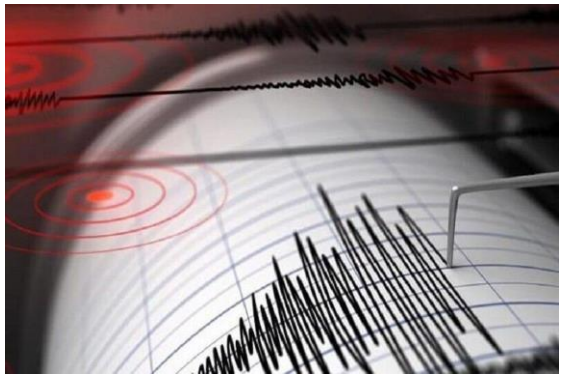


Recurrent Neural Network

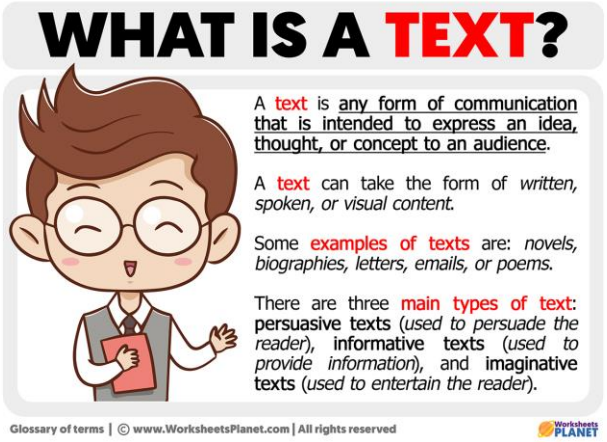
Sequences in the wild

Audio (image credit: Adobe)

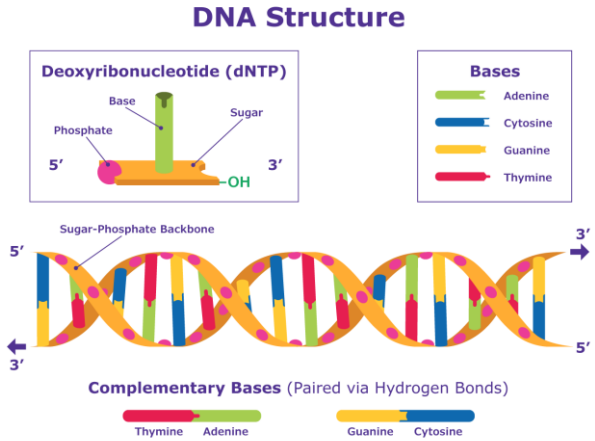
Stoch market analysis (image credit: Simplilearn)



Earthquake signal(image credit: SETI Institute)

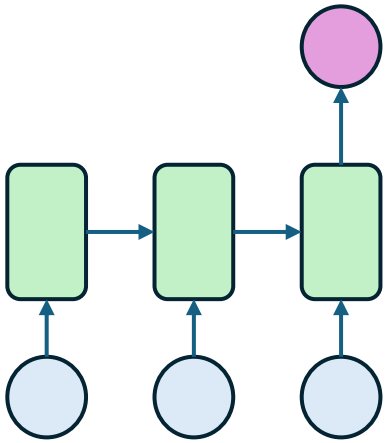


Text (image credit: Worksheets Planet)



DNA sequencing (image credit: Millipore Sigma)

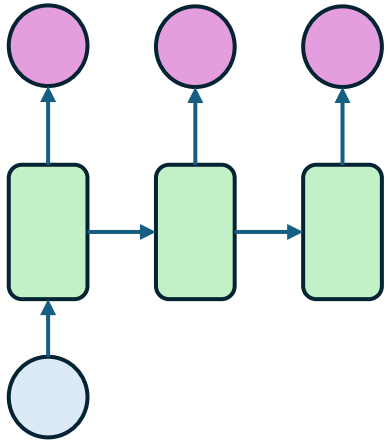
Sequence modeling applications



Many to one

Sentiment Analysis

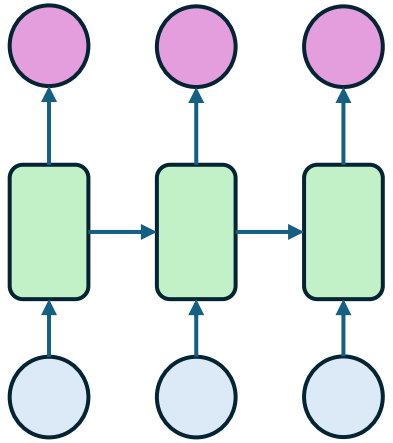
“The TrAC bootcamp is nice!”



One to many

Image Captioning

“Pikachu is jumping under a tree”

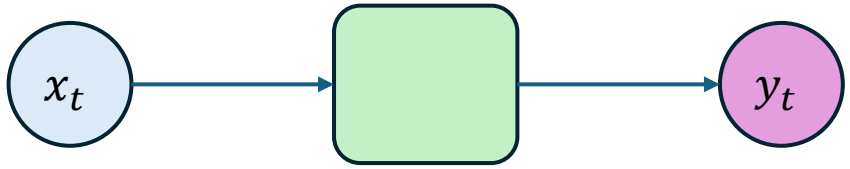


Many to many

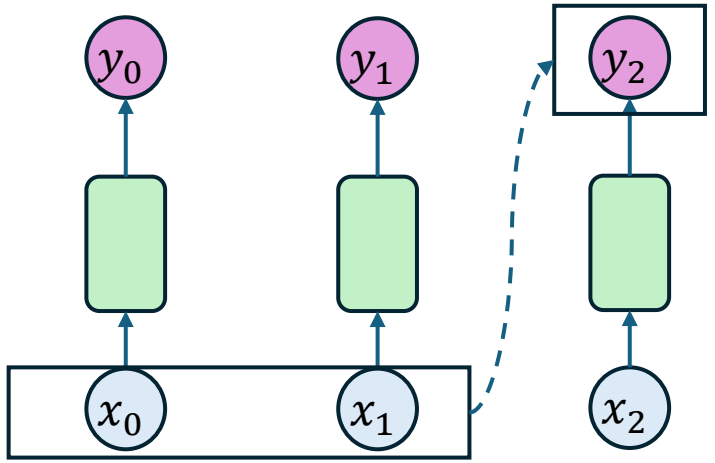
Machine Translation

Individual time steps

- Revisit the feed-forward neural networks

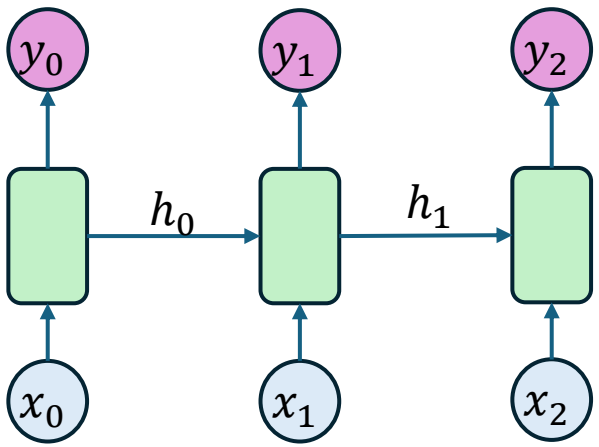


$$y_t = f(x_t)$$



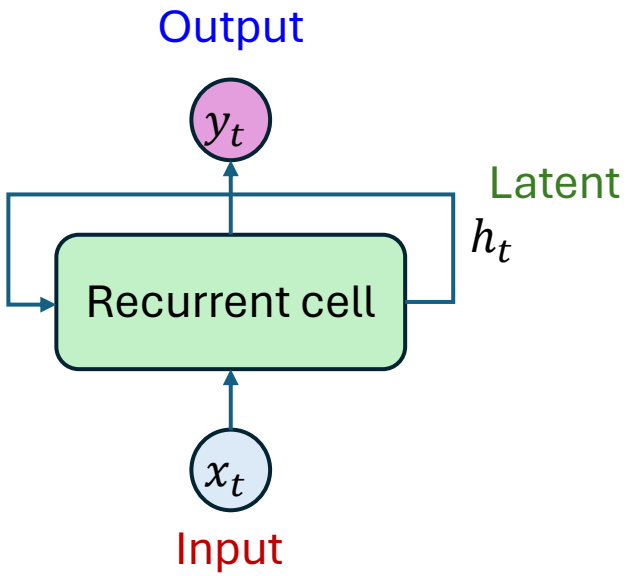
$$y_2 = f(x_2|x_0, x_1)$$

Neurons with recurrence

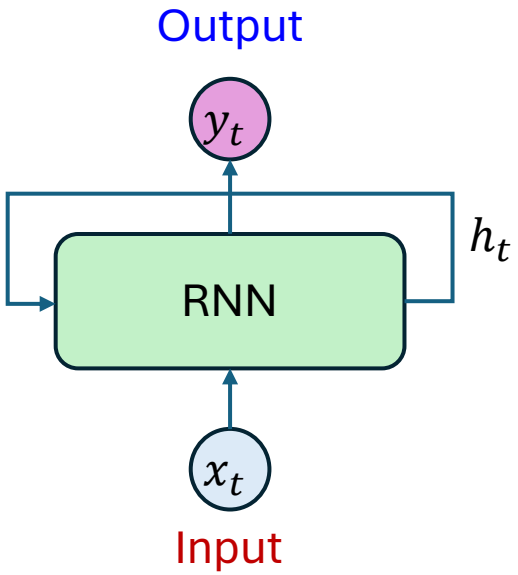


$$y_t = f(x_t, h_{t-1})$$

Output Input Past memory



Recurrent neural networks (RNNs)



Apply a **recurrence relation** at every time step to process a sequence

Input Old state

$$h_t = f_W(x_t, h_{t-1})$$

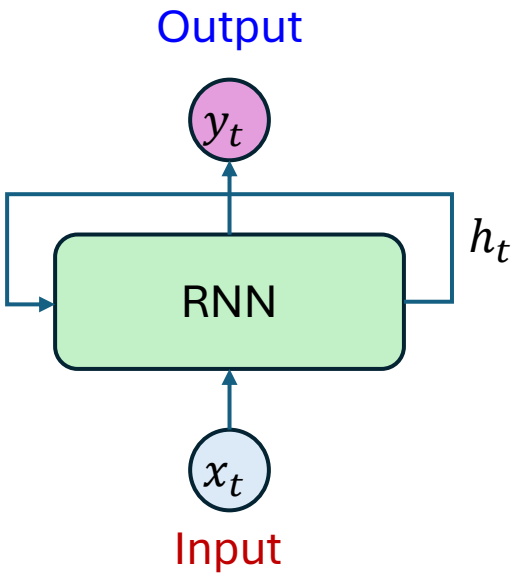
Current cell state

Parametric model

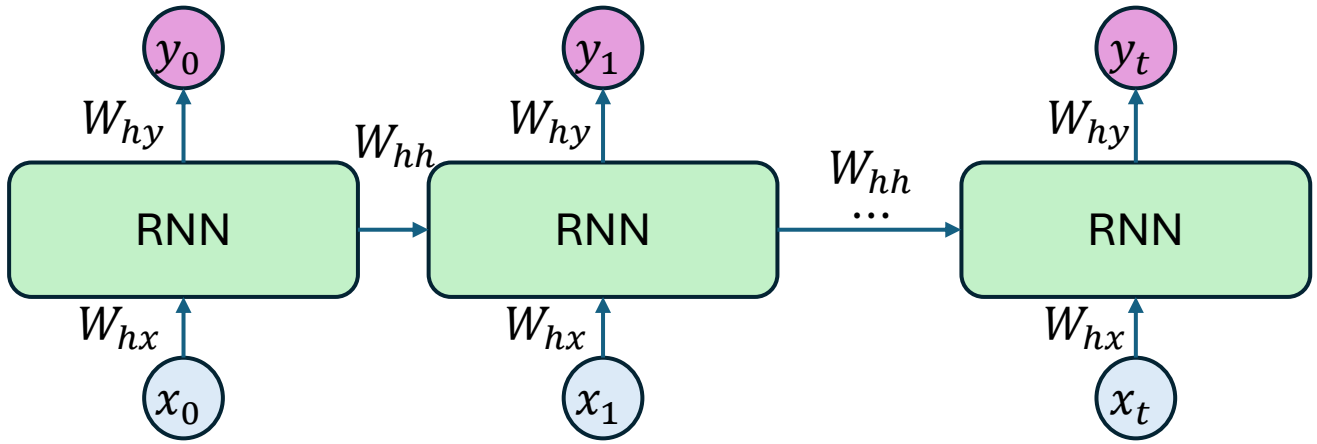
`torch.nn.RNN`

- The same function and set of parameters are used at every time step
- The cell (hidden) state is updated at each time step as a sequence is processed

State update and output



=



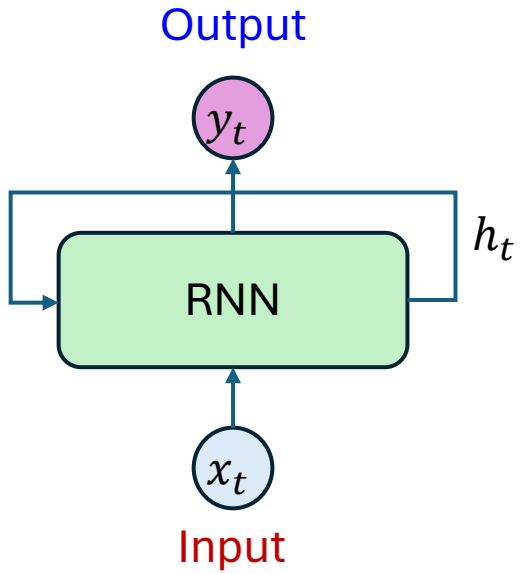
$$h_t = \tanh\left(W_{hh}^T h_{t-1} + W_{xh}^T x_t\right)$$
$$y_t = W_{hy}^T h_t$$

Trainable or learnable parameters

- Reuse the same weight matrices at every time step
- Computational graph across time

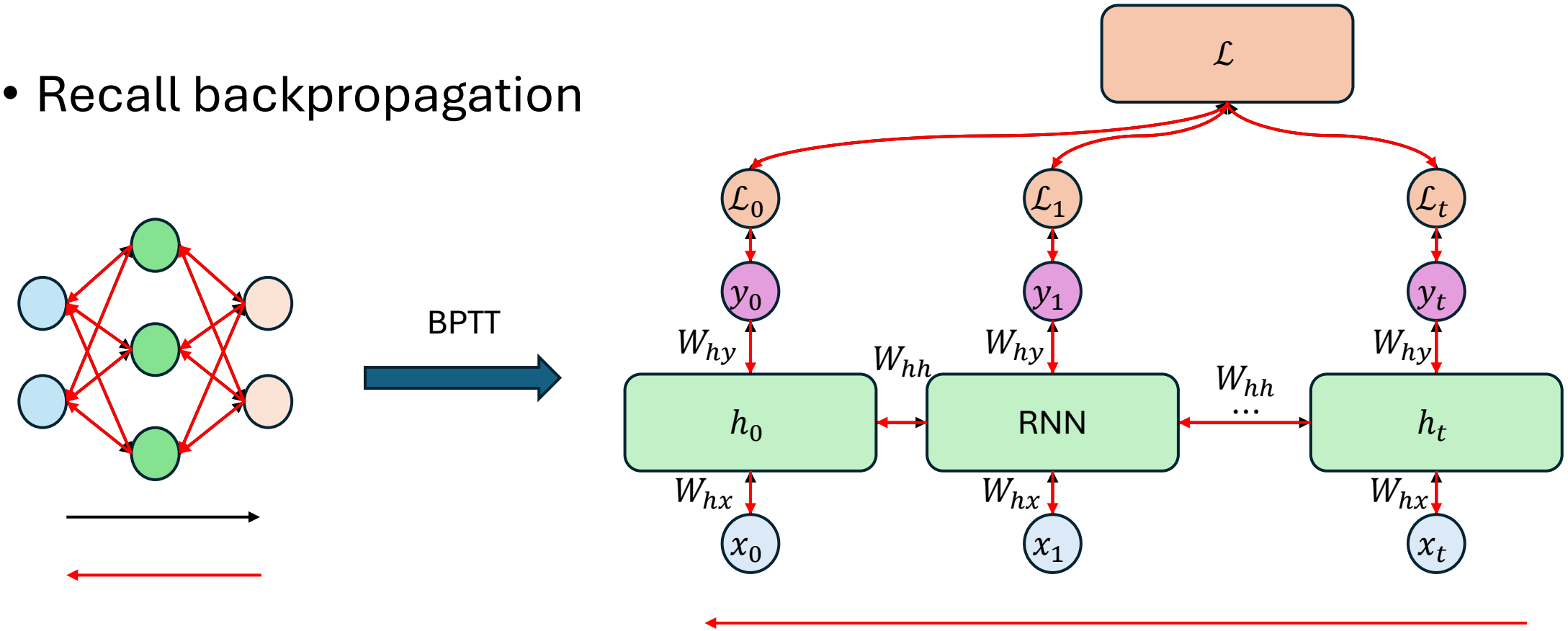
Sequence modeling: design criteria

1. Handle variable-length sequences
2. Track long-term dependencies
3. Maintain information about order
4. Share parameters across the sequence



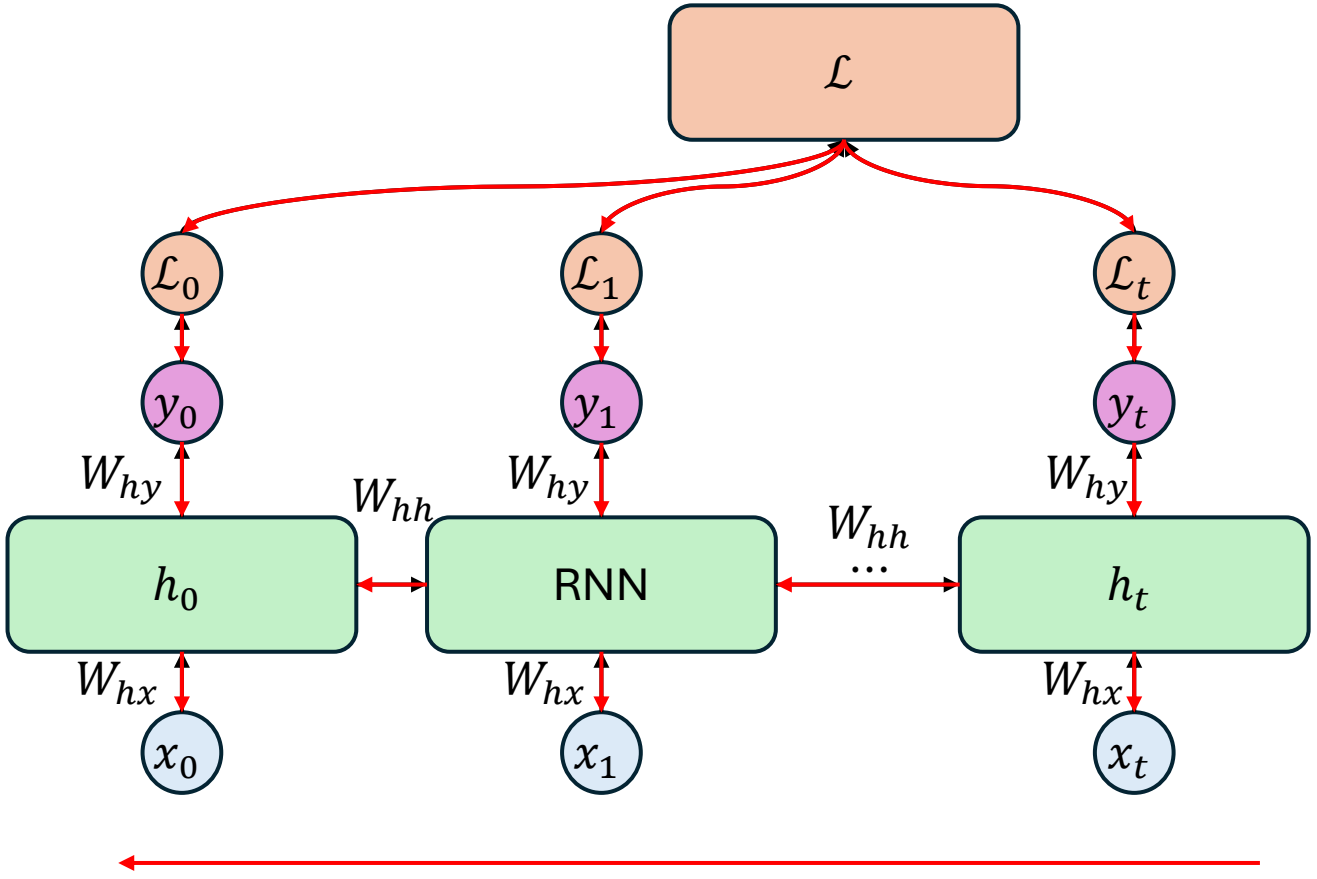
Backpropagation through time

- Recall backpropagation



Compute the gradient w.r.t h_0 involves many factors of W_{hh} + repeated gradient computation

Exploding/Vanishing gradients



- Many values > 1 lead to **exploding gradients**
  - Gradient clipping to scale big gradients
- Many values < 1 lead to **vanishing gradients**
  - Activation function
  - Weight initialization
  - Network architecture
- Vanishing gradient is the headache
  - LSTM was proposed for addressing the issue

Long short-term memory (LSTM)

1. Maintain a cell state
2. Use gates to control the flow of information
   - Forget gate get rid of irrelevant information
   - Store relevant information from current input
   - Selectively update cell date
   - Output gate returns a filtered version of the cell state
3. BPTT with partially uninterrupted gradient flow

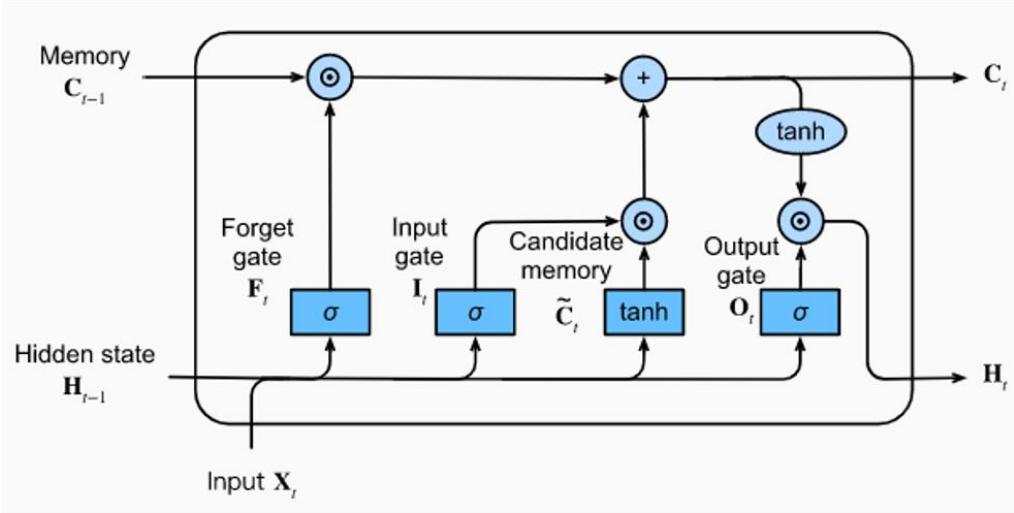


Image credit: Medium

`torch.nn.LSTM`

Limitations of recurrent models

- Encoding bottleneck
- Slow, no parallelization
- Not long memory

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

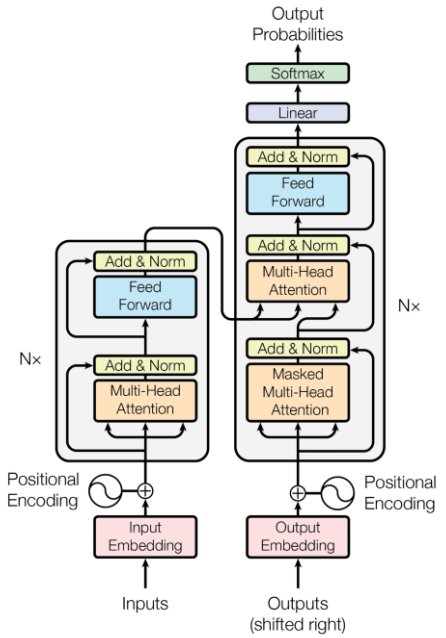
Llion Jones* Google Research llion@google.com

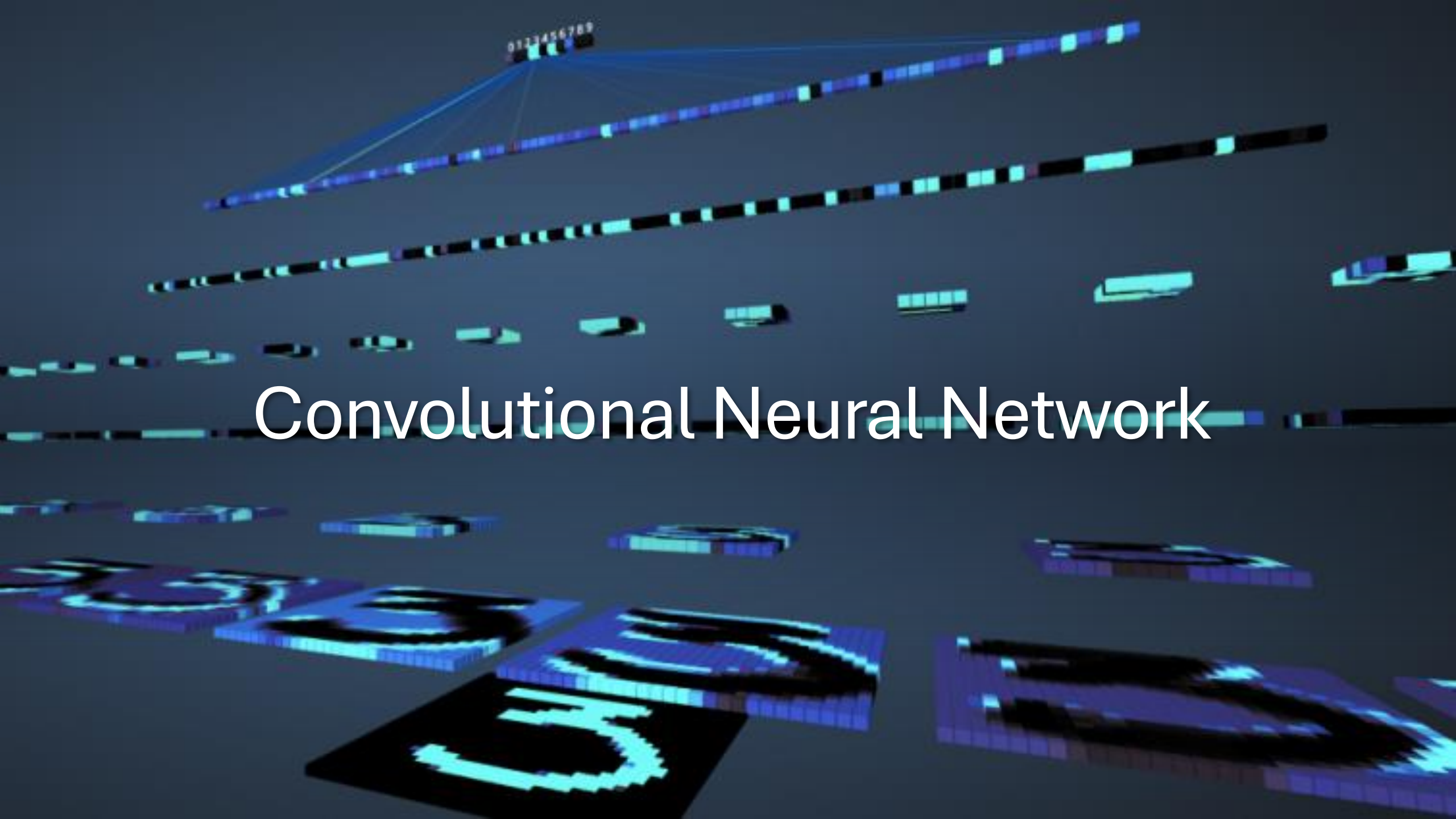
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Transformer



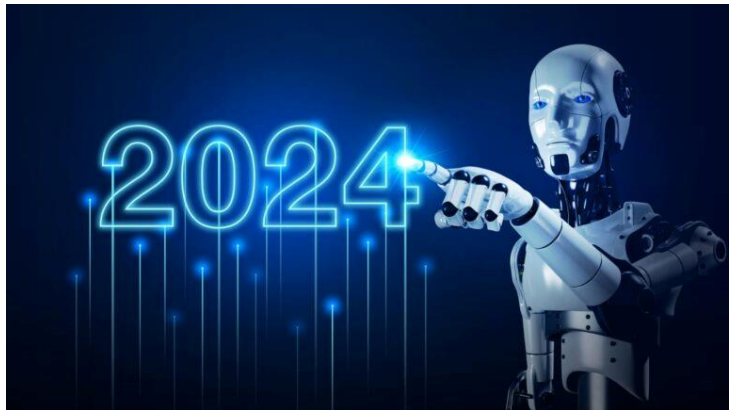

Convolutional Neural Network

Images

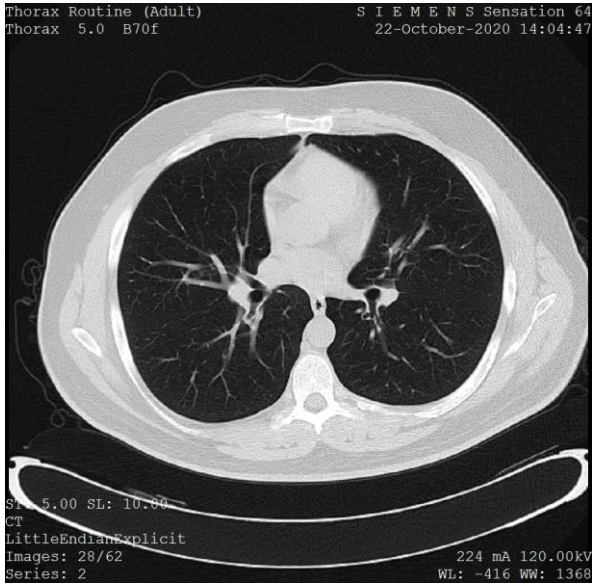
- What to do from images
  - What is present in the world
  - Where things are
  - What actions are taking place
  - Predicting and anticipating events in the world

Image credit: Towards Data Science

Computer vision



Robotics (image credit: robotics and automation)



Medical imaging (image credit: Wikipedia)

Autonomous driving (image credit: Dakota Digital Review)

Mobile phone (image credit: VideoMaker)

Surveillance (image credit: PC&D)

Satellite imaging (image credit: Geospatial World)

Images are numbers

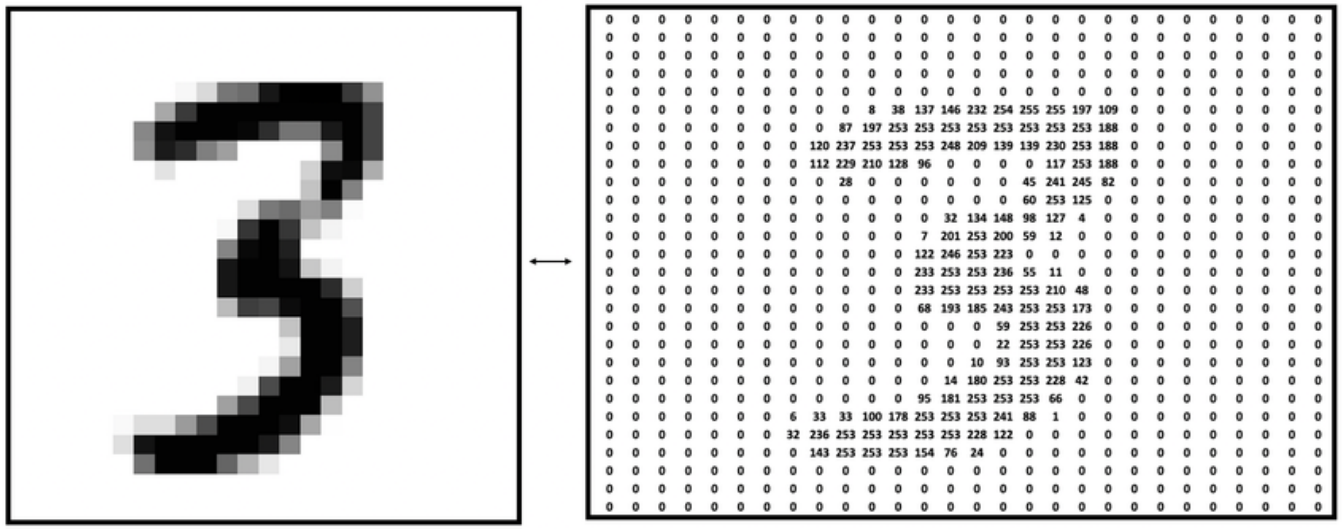


Image credit: IETC

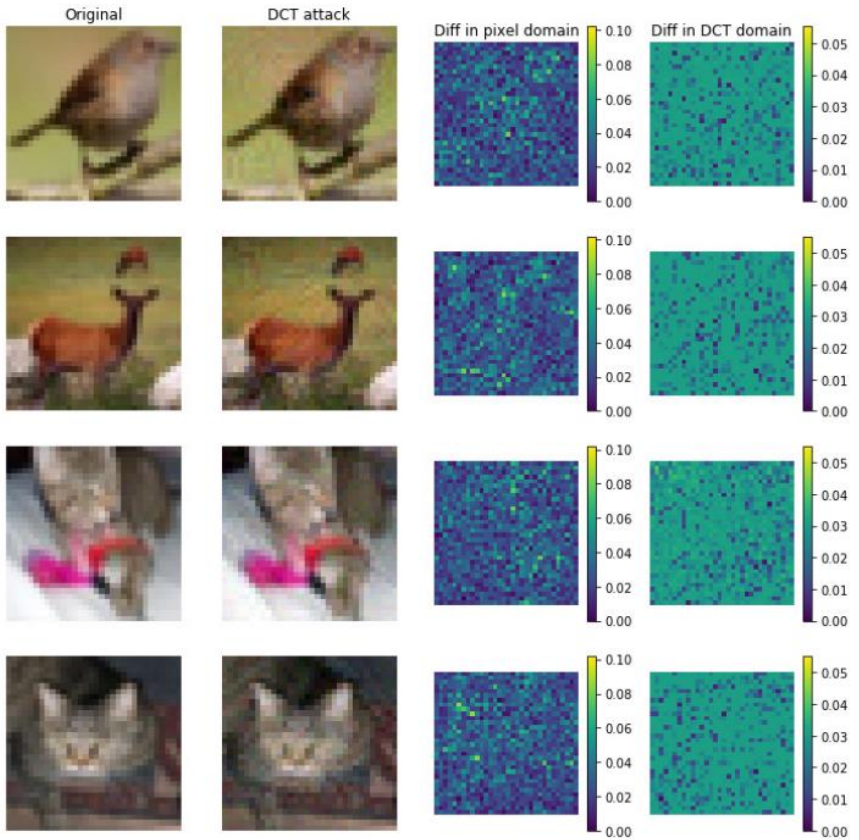
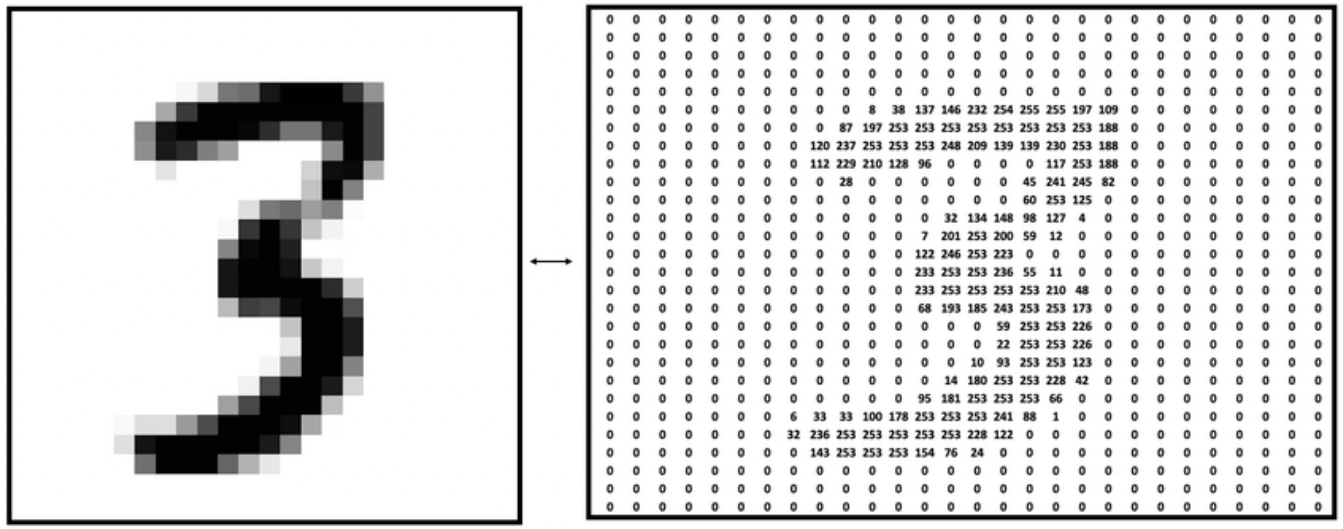


Image credit: CVPR

Tasks in computer vision



Classification

3 (0.8)

5 (0.1)

7 (0.05)

9 (0.05)

- Regression: output variables takes continuous value
- Classification: output variables takes class label. May output probability of belong to a specific class label

What does model do to images

- Learning patterns: identifying key features in each image

Eyes
Nose
Mouth

Wheels
License plate
Headlights

Door
Windows
Roof

Manual feature extraction

Domain knowledge → Define features → Detect features for classification

Viewpoint variation
(image credit: ML4Design)

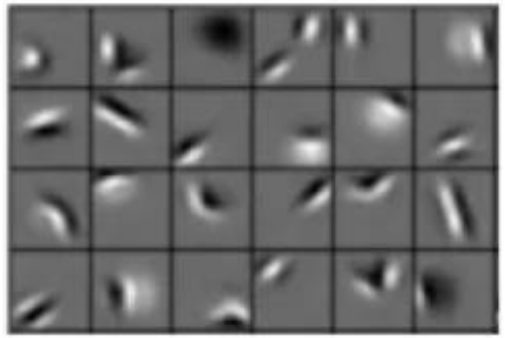
Scale variation

Intra-class variation
(image credit: Massouh, 2018)

Can a model learn to extract features

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

Low level features



Edges

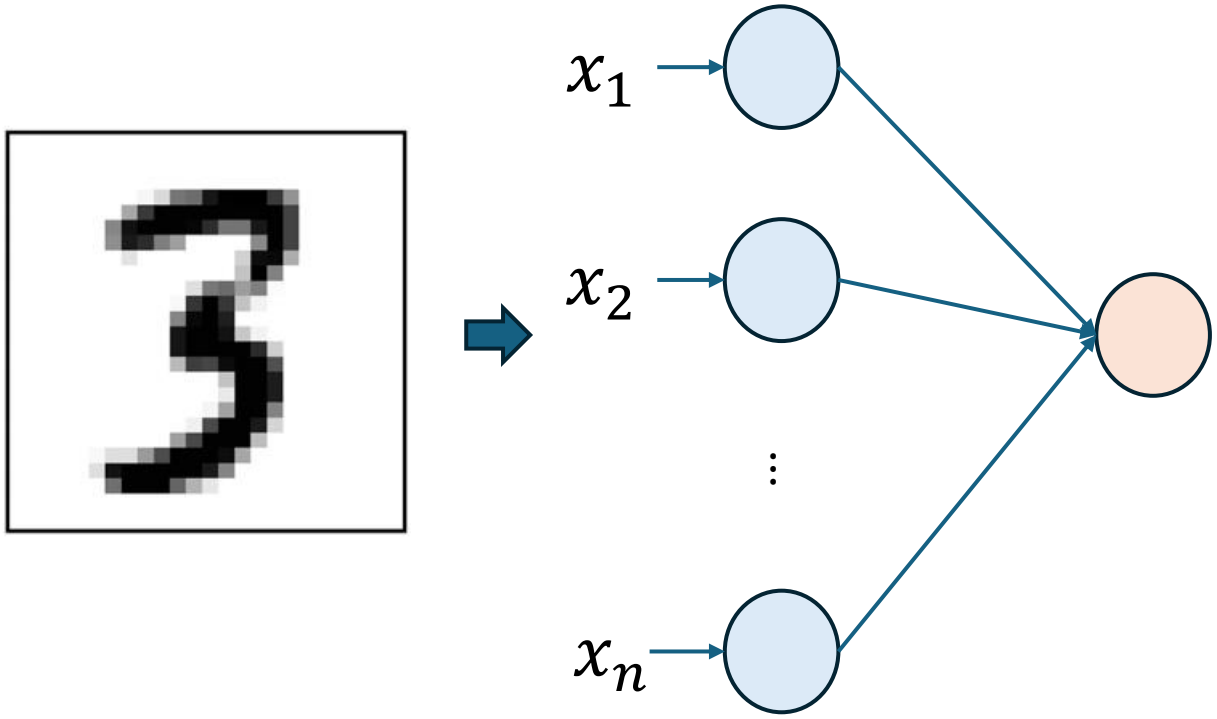
Mid level features

Eyes, noses, ears

High level features

Faces

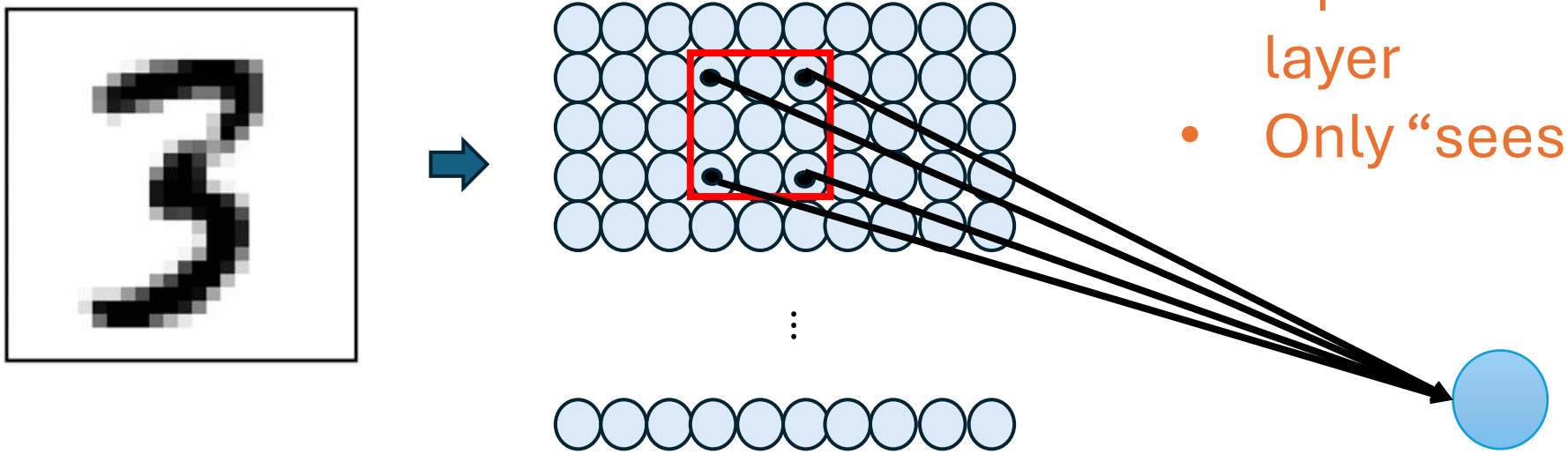
Feed-forward neural networks



- Connect neurons in hidden layers to all neurons in input layer
- But no spatial parameters
- Many, many parameters

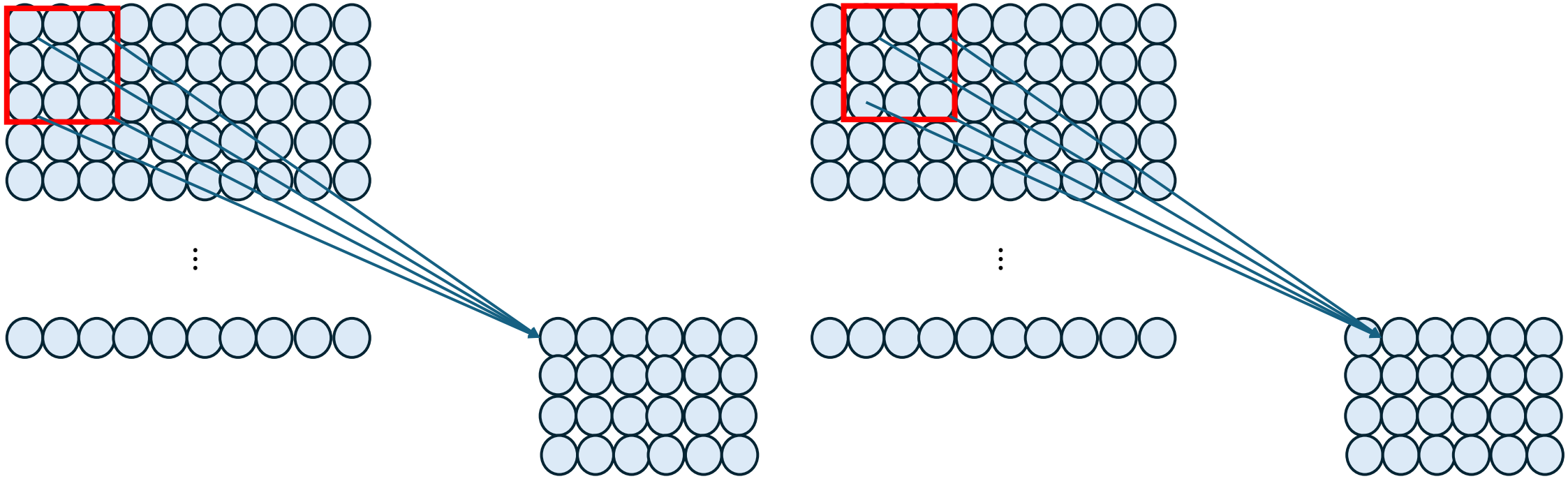
Can we use the spatial feature in the input to inform the architecture of network?

Using spatial feature



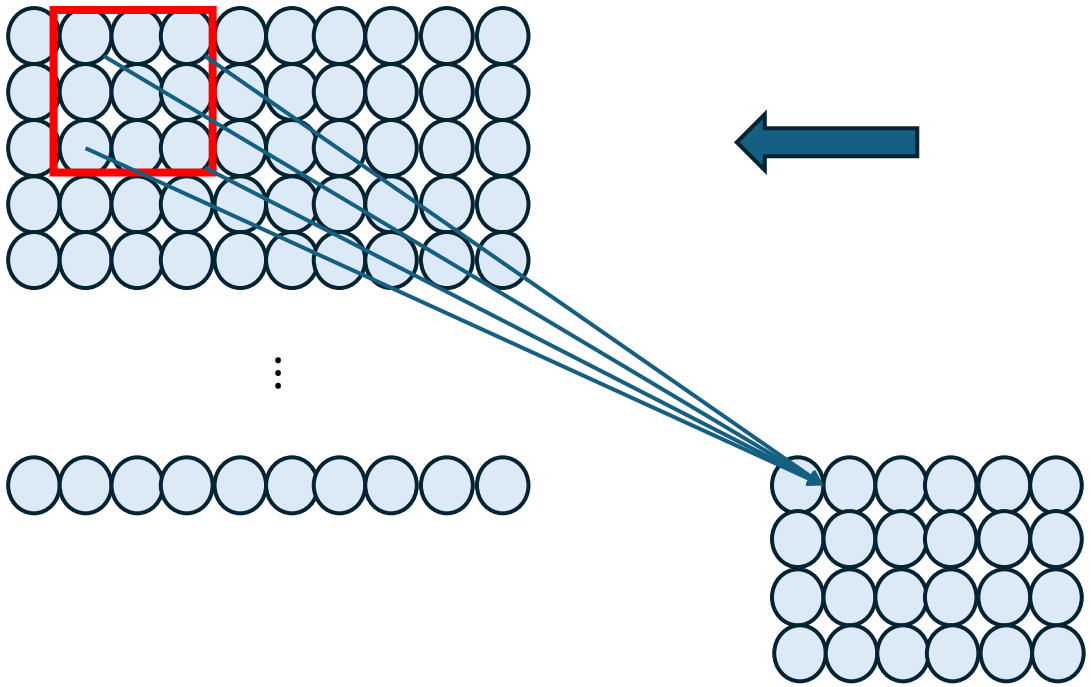
- Idea: Connect patches of input to neurons in hidden layer
- Only “sees” these values

Using spatial feature



- Connect patch in input layer to a single neuron in subsequent layer. Use a sliding window to define connections
- But how can we **weight** the patch to detect the particular features?

Convolution



- Filter of size 3x3: 9 different weights
- Apply this same filter to 3x3 patches in input
- Shift by 1 pixel for next patch

This patchy operation is called ***Convolution***

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

The convolution operation

- Suppose we want to compute the convolution of a 6x6 image and a 3x3 filter

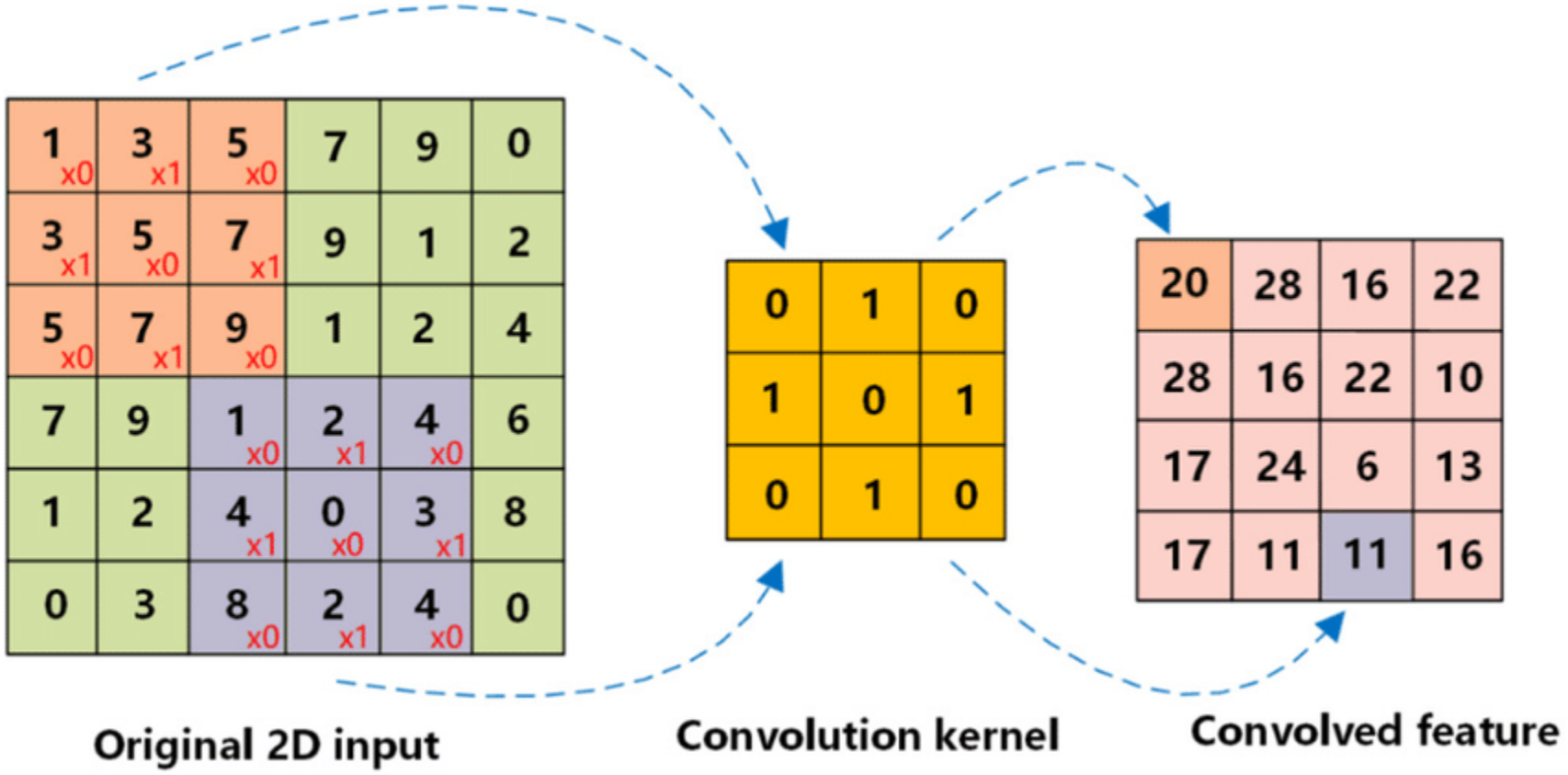
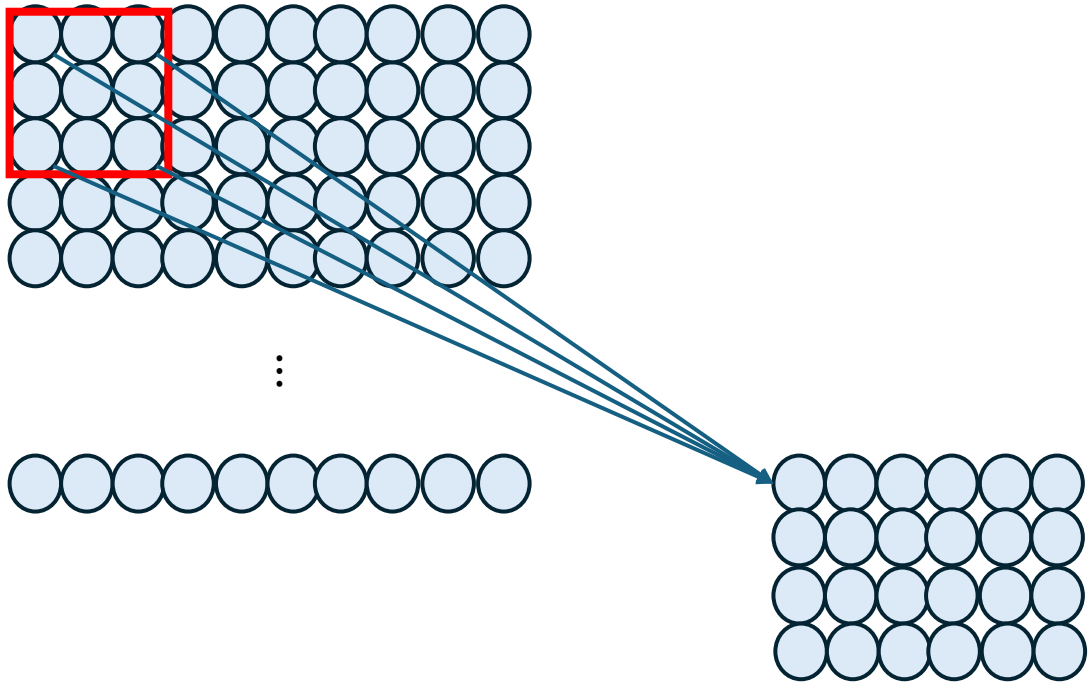


Image credit: ACS Omega

Convolutional layer: local connectivity



For **a neuron in convolutional layer**:

- Take inputs from patch
- Compute weighted sum
- Apply bias
- Add nonlinearity

Pooling layer

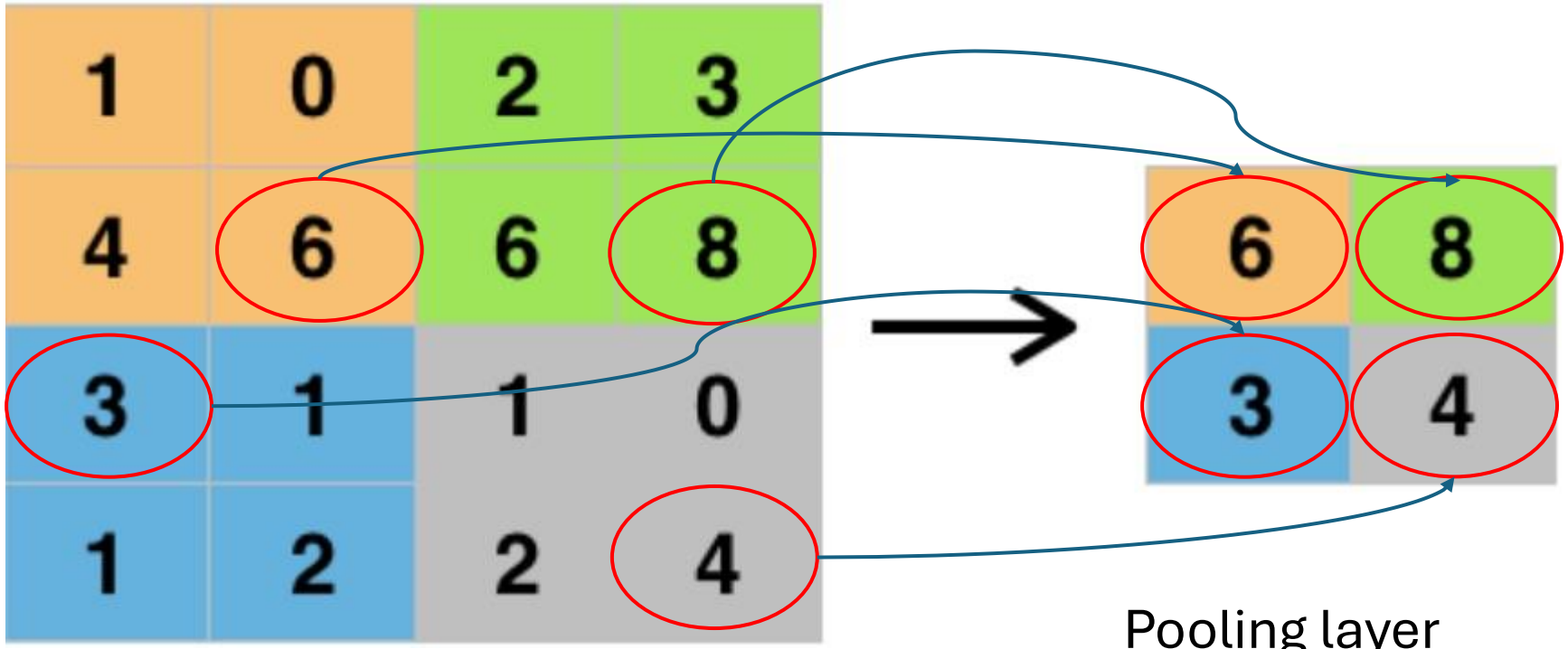


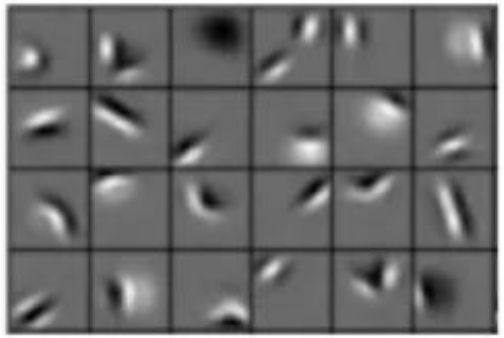
Image credit: DeepAI

Pooling layer

- Maxpooling layer typically
- Reduced dimentionality
- Spatial invariance

Learning representations

Low level features



Edges

Conv Layer 1

Mid level features

Eyes, noses, ears

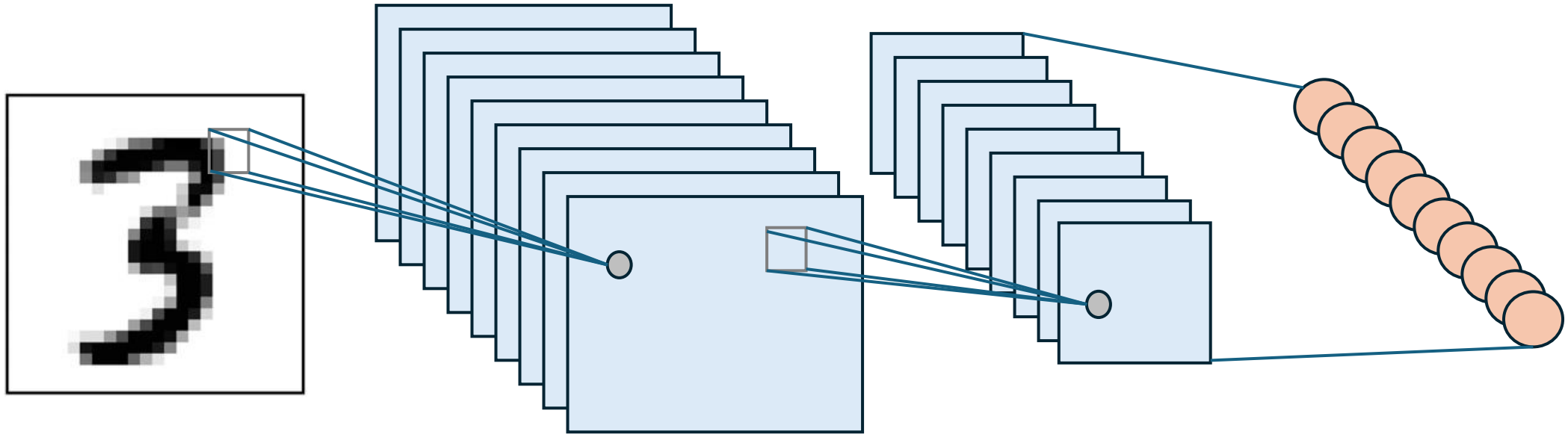
Conv Layer 2

High level features

Faces

Conv Layer 3

CNN for classification



Input image

Convolution (feature map) + ReLU

Maxpooling

Fully connected layer

1. **Convolution:** apply filters to generate feature maps
2. **Non-linearity:** often ReLU
3. **Maxpooling:** downsampling operation on each feature map

`torch.nn.Conv2d`
`F.relu`
`torch.nn.MaxPool2d`

Classification

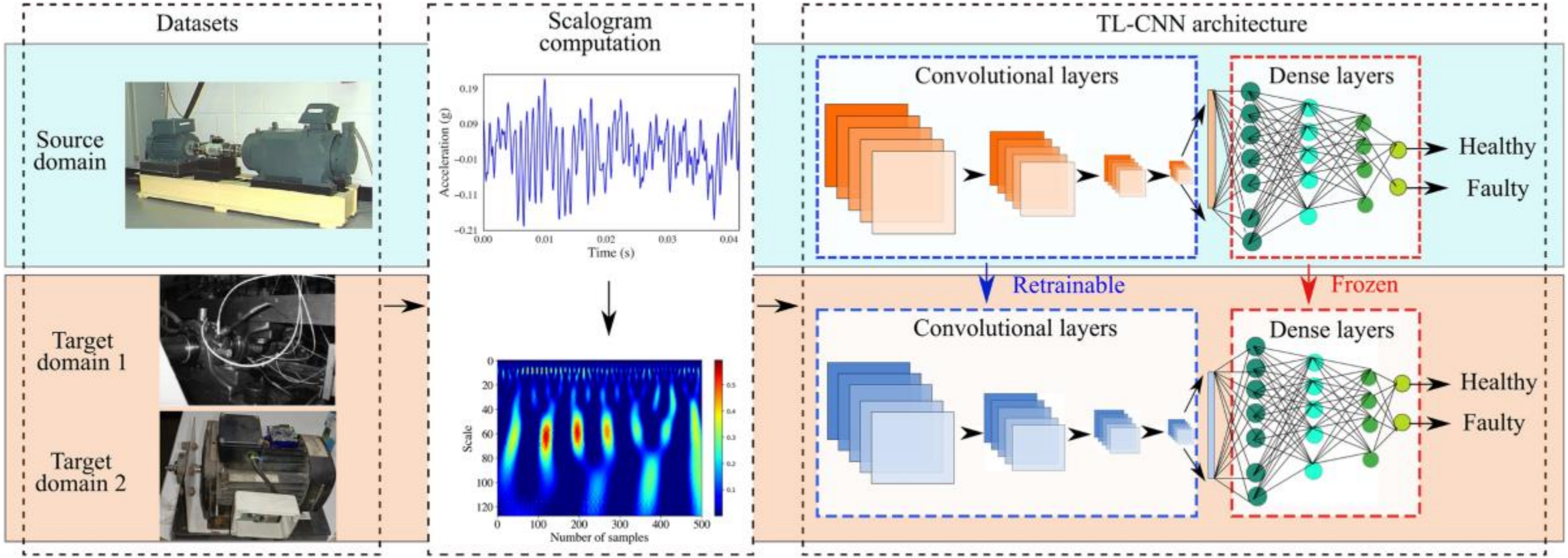


Image credit: Asutkar & Tallur, Scientific Reports, 2023

Object detection

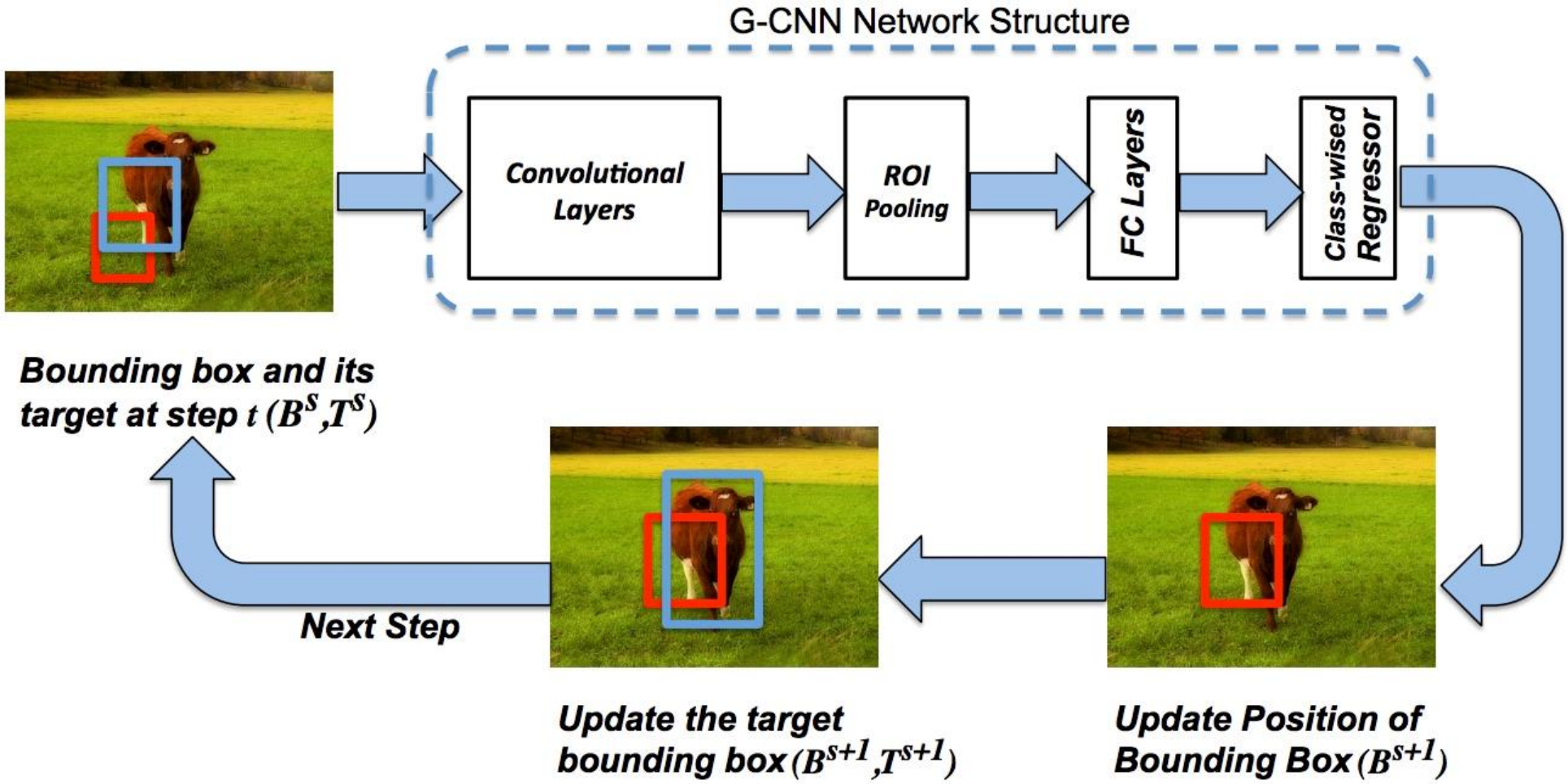


Image credit: AI2

Semantic segmentation

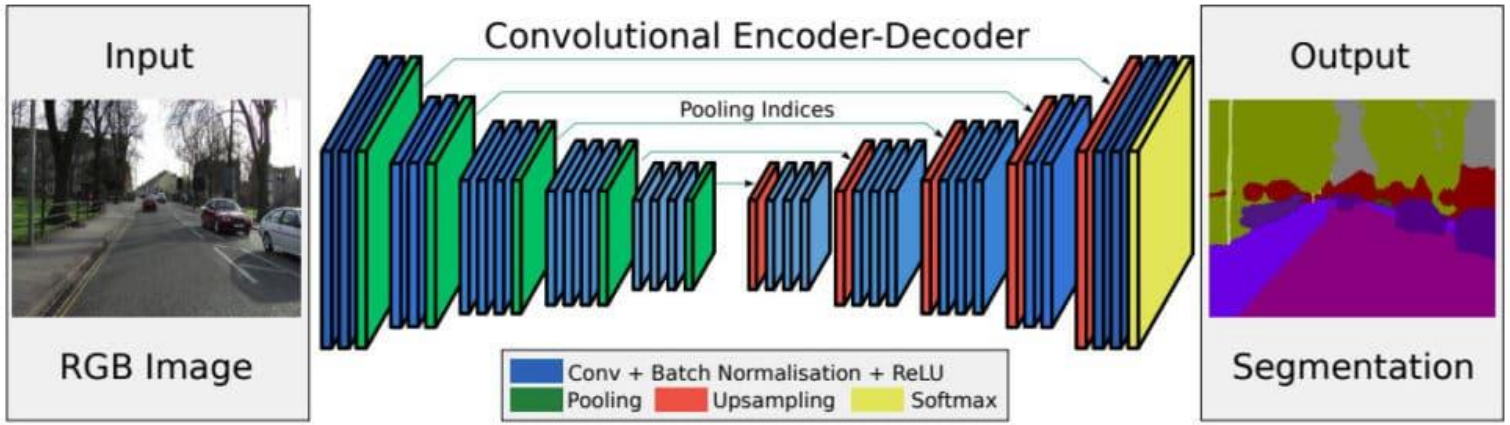


Image credit: cnvrg.io

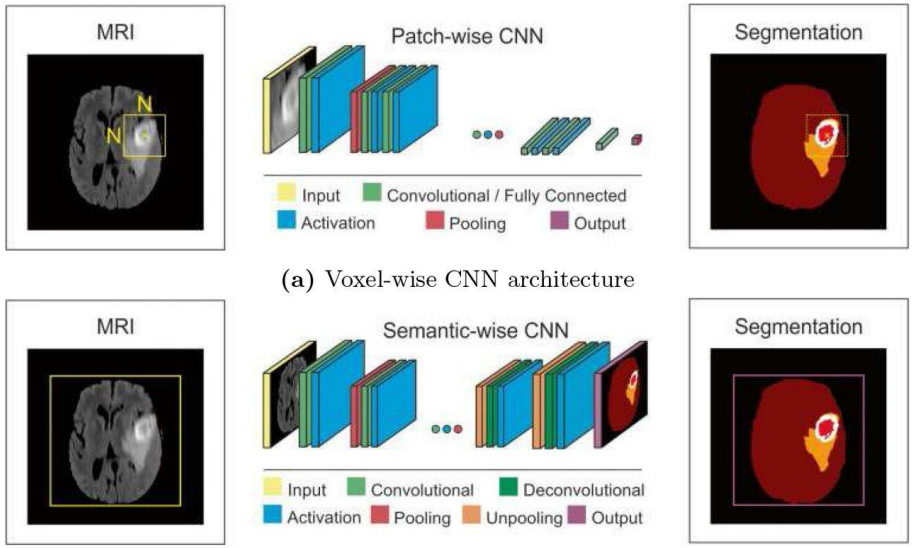


Image credit: bixRxiv

Continuous control

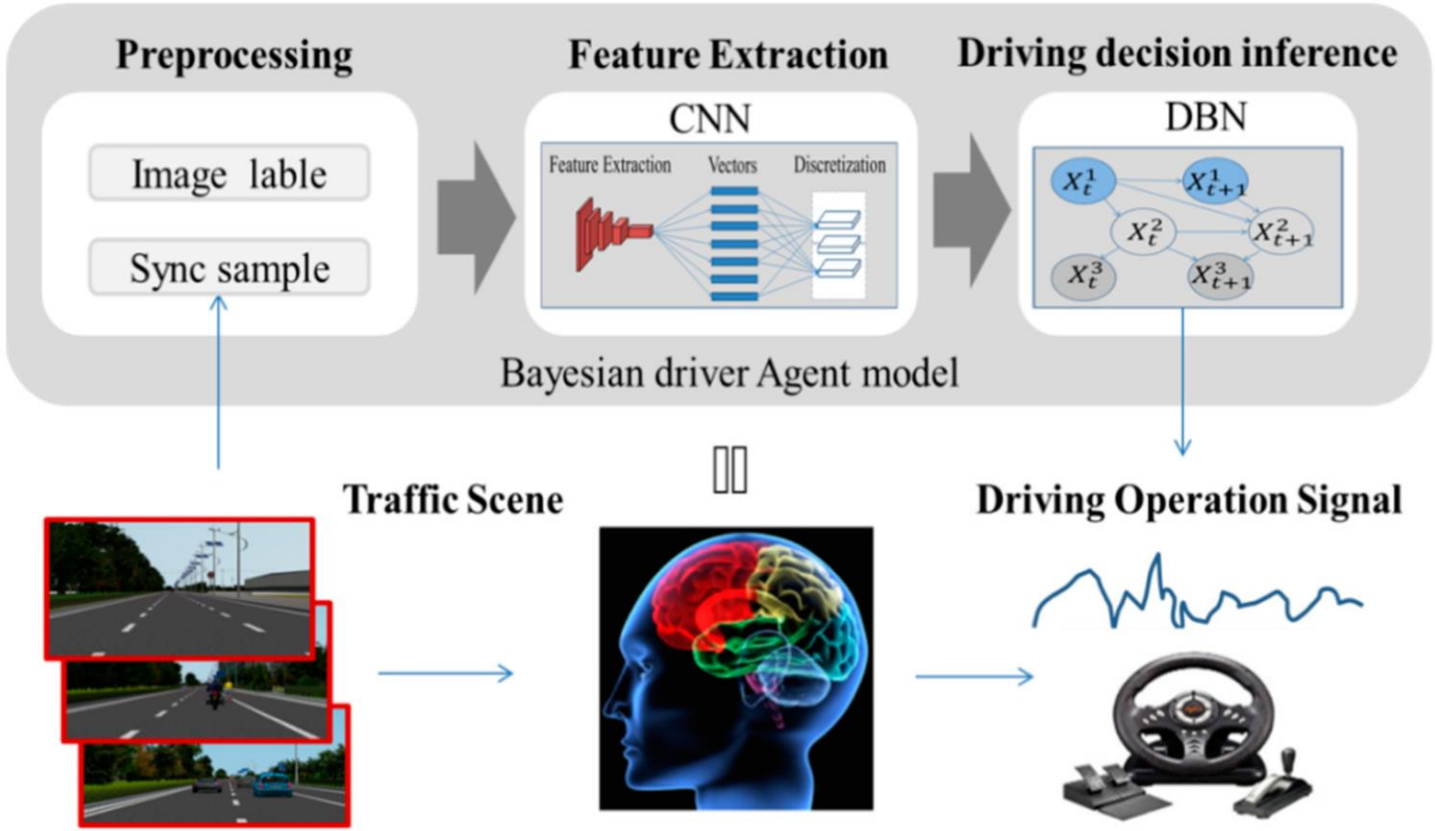


Image credit: mdpi

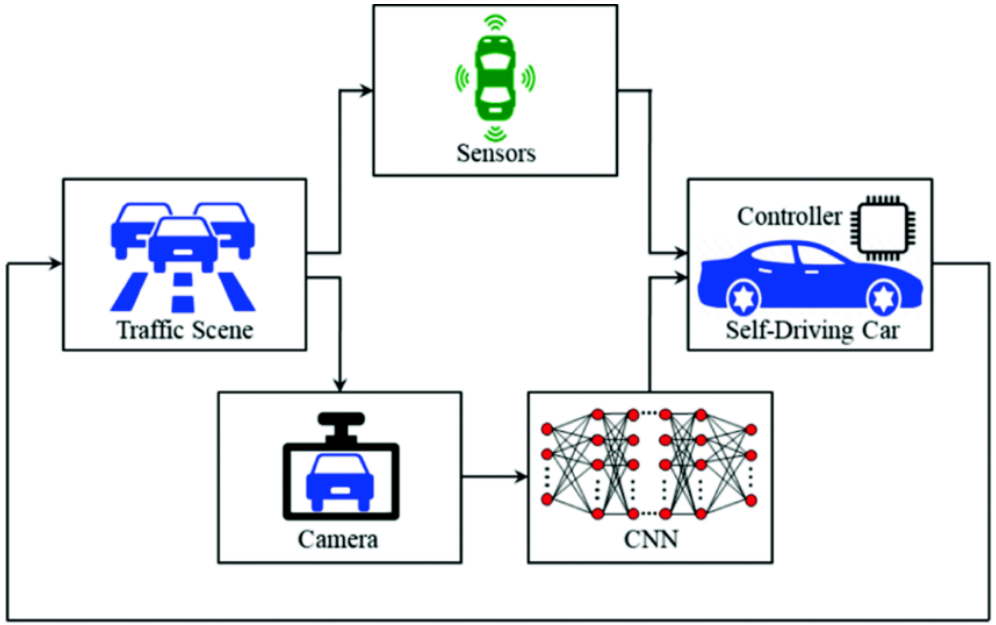


Image credit: Applied Intelligence

Beyond CNNs

- AlexNet
- VGG
- Inception and GoogLeNet
- ResNet
- DenseNet

Model	Layer	Parameter [M]	Network size[MB]
Alexnet	8	61	227
GoogLeNet	22	7	27
Inception-v3	48	23.9	89
VGG-16	16	138	515
ResNet18	18	25.6	96
ResNet50	50	44.6	167
Squeezenet	18	1.24	4.6

Table from Oh et al., 2019

AI is booming, but energy/water too

- *AI is costly*

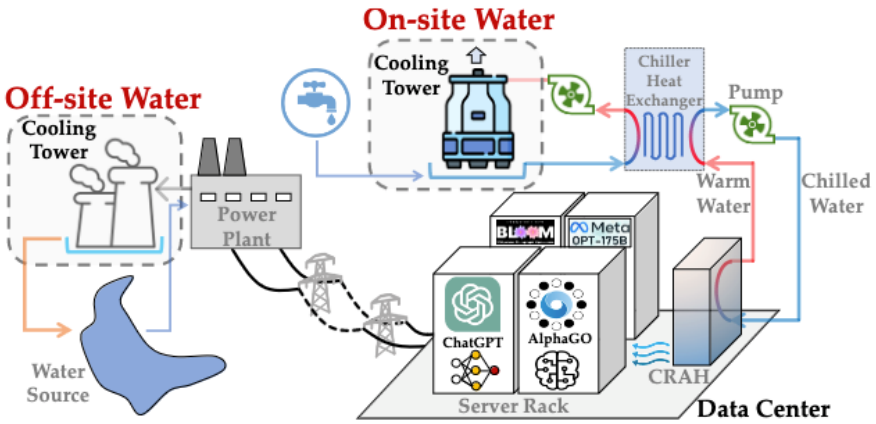


Figure 1: *An example of data center's operational water usage: on-site scope-1 water for server cooling (via cooling towers in the example), and off-site scope-2 water usage for electricity generation. The icons for AI models are only for illustration purposes.*

Secret: training GPT-3 in Microsoft's state-of-the-art U.S. data centers can directly evaporate **700,000** liters of clean freshwater

Even more: the global AI demand may be accountable for **4.2 – 6.6 billion cubic meters** of water withdrawal in 2027 (4-6 Denmark or half of the UK)

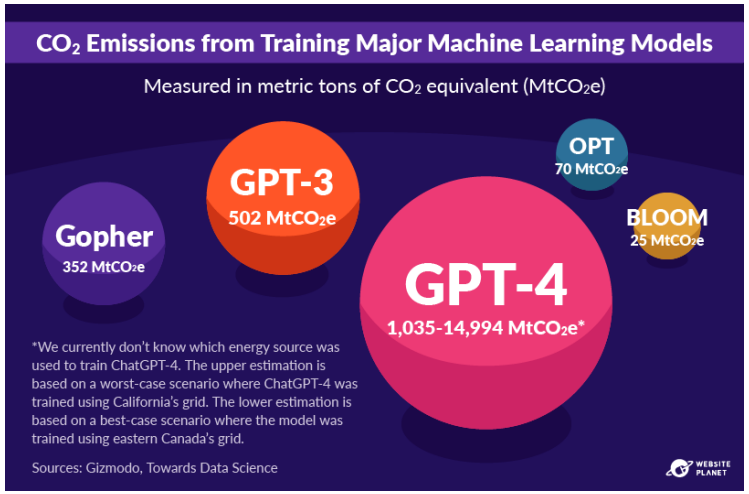
Li et al. Making AI Less "Thirsty": Uncovering and Addressing the Secret Water Footprint of AI Models. arXiv, 2023

AI is booming, but energy/water too

	GPU Type	GPU Power consumption	GPU-hours	Total power consumption	Carbon emitted (tCO_2eq)
OPT-175B	A100-80GB	400W	809,472	356 MWh	137
BLOOM-175B	A100-80GB	400W	1,082,880	475 MWh	183
LLaMA-7B	A100-80GB	400W	82,432	36 MWh	14
LLaMA-13B	A100-80GB	400W	135,168	59 MWh	23
LLaMA-33B	A100-80GB	400W	530,432	233 MWh	90
LLaMA-65B	A100-80GB	400W	1,022,362	449 MWh	173

Table 15: **Carbon footprint of training different models in the same data center.** We follow Wu et al. (2022) to compute carbon emission of training OPT, BLOOM and our models in the same data center. For the power consumption of a A100-80GB, we take the thermal design power for NVLink systems, that is 400W. We take a PUE of 1.1 and a carbon intensity factor set at the national US average of 0.385 kg CO_2e per KWh.

Image credit: Medium



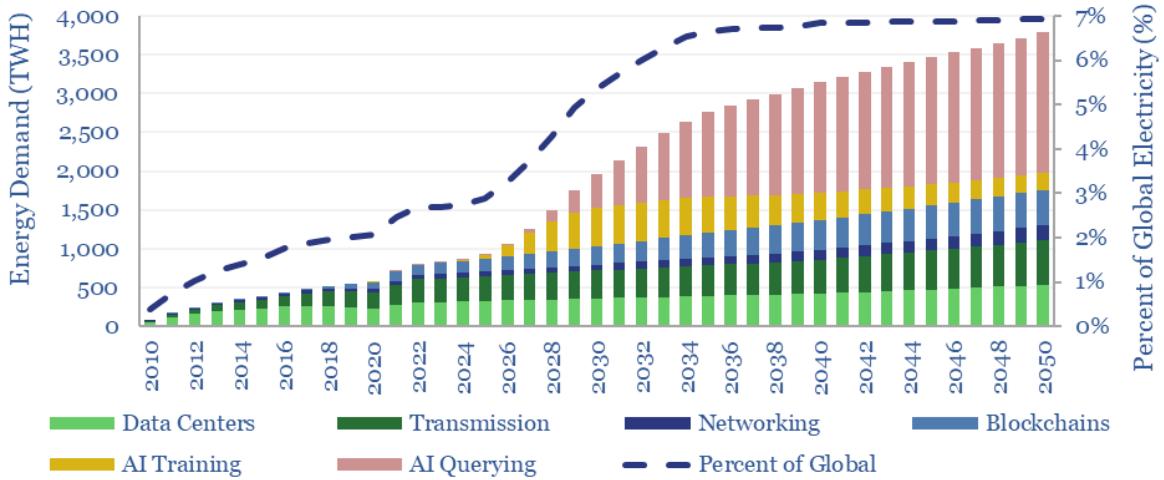


Image credit: Thunder Said Energy

- *Sustainable AI is required immediately*

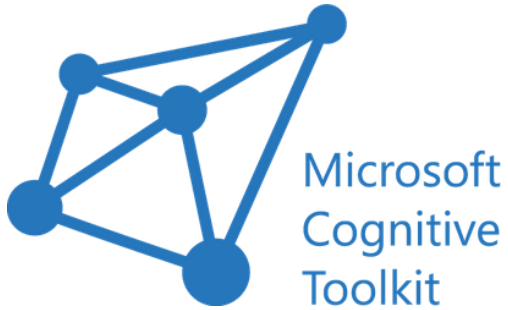
Outline

- Session 1: Introduction to deep learning
- Session 2: Convolutional neural network (CNN) and recurrent neural network (RNN)
- Session 3: Introduction to deep learning packages

Pytorch

- Open-source deep learning library
- Developed by Facebook AI research lab
- It leverages the power of GPUs
- Automatic computation of gradients
- Makes it easier to test and develop new ideas

Other libraries



Why Pytorch?

- Automatic differentiation
- Close to Python conventions
- Many algorithms and components are already implemented
- More flexible to customize
- Strong GPU support

Why PyTorch

```
m = 1000
n = 1000

Anp = np.random.randn(m,n)
xnp = np.random.randn(n)
ynp = np.random.randn(m)
print("numpy")
%time z = ynp + Anp @ xnp


for device in ('cpu', 'cuda'):
    print(f"\ndevice = {device}")
    A = torch.Tensor(Anp).to(device=device)
    x = torch.Tensor(xnp).to(device=device)
    y = torch.Tensor(ynp).to(device=device)
    z = torch.addmv(y, A, x)

    %time z = torch.addmv(y, A, x)
```

```
numpy
CPU times: user 3.56 ms, sys: 0 ns, total:
3.56 ms
Wall time: 1.01 ms

device = cpu
CPU times: user 655 µs, sys: 263 µs,
total: 918 µs
Wall time: 299 µs

device = cuda
CPU times: user 168 µs, sys: 68 µs, total:
236 µs
Wall time: 65.8 µs
```

Why Pytorch

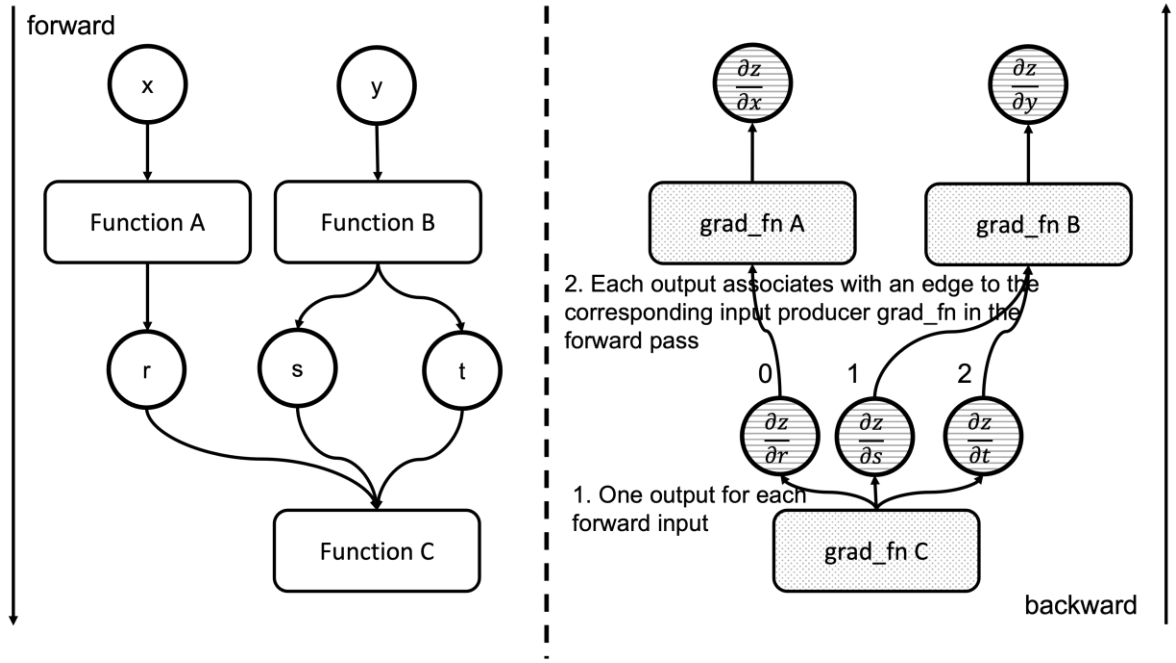
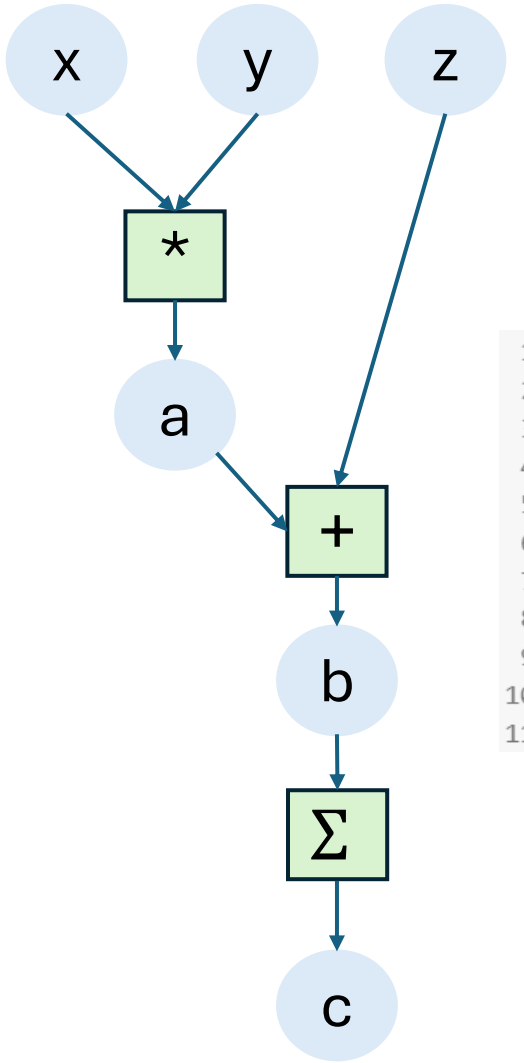


Image credit: PyTorch




```python
"""Pytorch computation graph"""
import torch
N, D = 3, 4
x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)
c.backward()
```

How to install Pytorch

- Installation
  - Via Anaconda/Miniconda

```
conda install pytorch-c pytorch
```

  - Via pip

```
pip3 install torch
```

PyTorch basics: Tensors

- Tensors are similar to NumPy's ndarrays
- Tensors can also be used on a GPU to accelerate computing
- Common operations for creation and manipulation of these Tensors are similar to those for ndarrays in NumPy (rand, ones, zeros, reshape, transpose, matrix product, element wise multiplication)

Tensors

- `requires_grad`
  - Making a trainable parameter
- `x.data`
  - Access to tensor's value
- `x.grad`
  - Access to tensor's gradient
- `grad_fn`
  - History of operations for autograd

```python
"""Pytorch computation graph"""
import torch
N, D = 3, 4
x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)
c.backward()

print(c.grad_fn)
print(x.data)
print(x.grad)
```

```
<SumBackward0 object at 0x7f44ff66f2b0>
tensor([[0.8082, 0.8666, 0.2984, 0.1447],
        [0.4364, 0.0756, 0.7899, 0.5278],
        [0.7252, 0.9558, 0.9685, 0.2765]])
tensor([[0.3935, 0.1292, 0.1767, 0.7060],
        [0.4185, 0.0155, 0.0299, 0.3211],
        [0.7229, 0.8132, 0.2471, 0.7520]])
```

Loading data, devices, and CUDA

- Numpy array to PyTorch tensor
  - `torch.from_numpy(x)`
- PyTorch tensor to Numpy array
  - `x.to_numpy()`
- Using GPU acceleration
  - `x.to()`
  - Sends to whatever device
- Fallback to CPU if GPU is not available
  - `torhc.cuda.is_available()`
- Check the type
  - `x.type() or type(x)`

Compute Unified Device Architecture (CUDA): proprietary parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for accelerated general-purpose processing

Autograd

- Automatic differentiation package
- Don’t need to worry about partial differentiation, chain rule, etc.
  - `backward()` does that
- Gradients are cumulated each step by default
  - Need to zero out gradients after each update
  - `tensor.grad_zero()`

```
# Create tensors.
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

# Build a computational graph.
y = w * x + b # y = 2x + 3

# Compute gradients
y.backward()

# Print out the gradients.
print(x.grad)
print(w.grad)
print(b.grad)
```

```
tensor(2.)
tensor(1.)
tensor(1.)
```

Optimizer and loss

Optimizer:

- Adam, SGD, etc.
- You can even customize your own optimizer
- An optimizer takes the parameters and the learning rate as input

Loss:

- PyTorch provides various predefined loss to choose from
- Cross entropy, MSE, L1, etc.

```
# Define loss and optimizer.
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr)
```

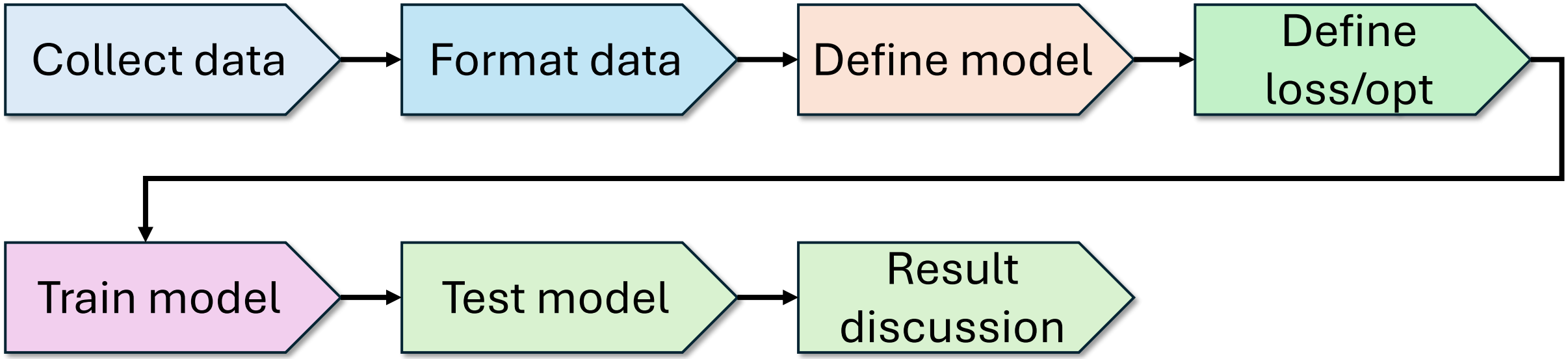
Model

- In PyTorch, a model is represented by a regular Python class that inherits from the `Module` class
- Two key components
  - `__init__(self)`: it initializes the model with two parts that make up the model with key parameters
  - `forward(self, x)`: it performs the actual computation, by outputting a prediction given x

```
class LinearRegression(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, x):
        out = self.linear(x)
        return out
```

High-level model development pipeline



PyTorch tutorial

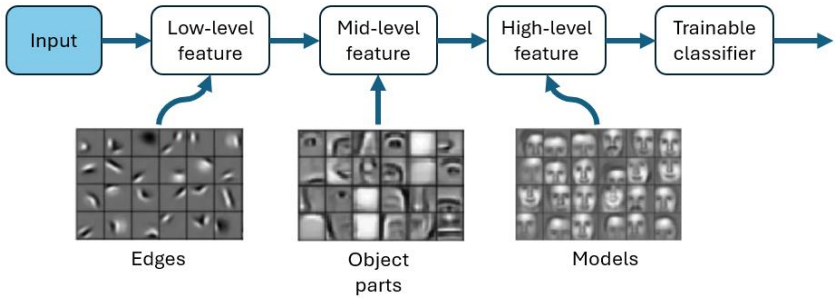
- https://pytorch.org/tutorials/

Coding session

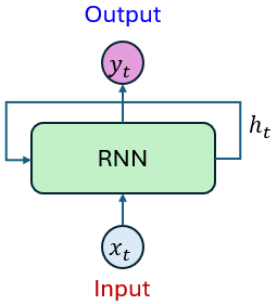
- https://colab.research.google.com/drive/1AFZk0fTOuQWSo9pqBo-015vOxtit2vYJ?usp=sharing

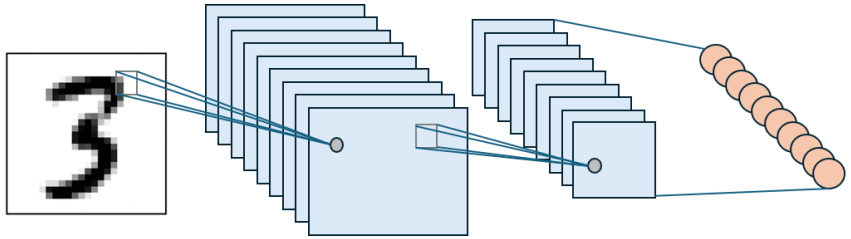
Summary

What is deep learning



Deep learning models





Deep learning packages

PyTorch