



Deep Learning for Science and Engineering Teaching Kit

# Deep Learning for Scientists and Engineers

## Lecture 6: Neural Network Architectures

**Instructors: George Em Karniadakis, Khemraj Shukla, Lu Lu**
**Teaching Assistants: Vivek Oommen and Aniruddha Bora**


# Course Roadmap

## Module-1 (Basics)

- Lecture 1: Introduction
- Lecture 2: A primer on Python, NumPy, SciPy and *jupyter* notebooks
- Lecture 3: Deep Learning Networks
- Lecture 4: A primer on TensorFlow and PyTorch
- Lecture 5:  Training and Optimization
- Lecture 6: Neural Network Architectures

## Module-2 (PDEs and Operators)

- Lecture 7: Physics-Informed Neural Networks (PINNs)
- Lecture 8: PINN Extensions
- Lecture 9: Neural Operators

## Module-3 (Codes & Scalability)

- Lecture 10: Multi-GPU Scientific Machine Learning

# Contents

- ❑ Motivation behind different types of neural networks
- ❑ Convolutional Neural Network (CNN)
- ❑ Different attributes of CNN
- ❑ Universality of deep CNN
- ❑ Kernel (CNN vs Finite Difference stencil)
- ❑ Residual Neural Network (ResNet)
- ❑ Universality of ResNet
- ❑ ResNet comparison with Forward Euler Scheme
- ❑ Modelling and prediction of sequence data (time series)
- ❑ Recurrent Neural Network (RNN)
- ❑ Exploding and vanishing gradients
- ❑ Long Short-Term Memory (LSTM)
- ❑ Data preparation for RNN and LSTM
- ❑ Encoder-Decoder architecture
- ❑ Auto-Encoders
- ❑ Generative modelling
- ❑ Vanilla Generative Adversarial Network (GAN)
- ❑ Wasserstein GAN (WGAN)
- ❑ WGAN-GP (Gradient Penalty)
- ❑ Summary
- ❑ References

# Recall: A Fully-connected Neural Network (FNN)

- Define the affine transformation in $l-$ th layer

$$T^l(x) = \boldsymbol{W}^l x + \boldsymbol{b}^l$$

- Activation functions $\sigma$

$$\text{Popular choices: } \tanh(x), \quad \max\{x, 0\} \text{ (ReLU)}$$

- The $L-1$ hidden layers of feedforward neural network:

$$\mathcal{N}_{HL}(x) = \sigma \circ T^{L-1} \circ ... \circ \sigma \circ T^1(x),$$

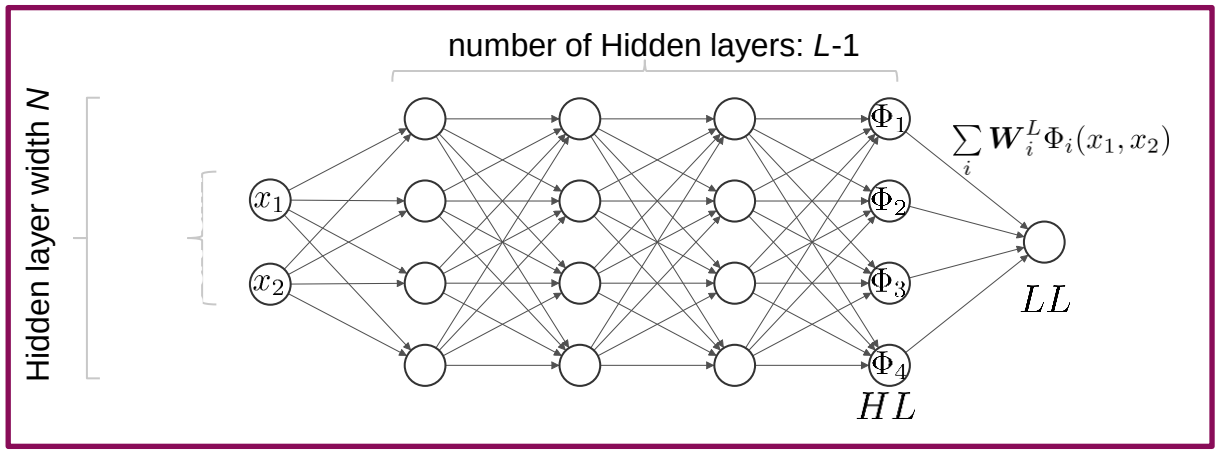
where $\circ$ denotes the composition of fucntions.

- For regression, a DNN is typically of the form:

$$\mathcal{N}(x; \boldsymbol{\theta}) = T^L \circ \mathcal{N}_{HL}(x)$$

- Network parametrs

$$\boldsymbol{\theta} = \{\boldsymbol{W}^l, \boldsymbol{b}^l\}_{1 \le l \le L}$$



# Motivation and Need for Other Neural Networks

**Cause:** Diverse data formats and learning tasks
**Rescue**: Different neural network architectures

## Datatype: Tabular data

**Breast Cancer Wisconsin (Prognostic) Data Set**

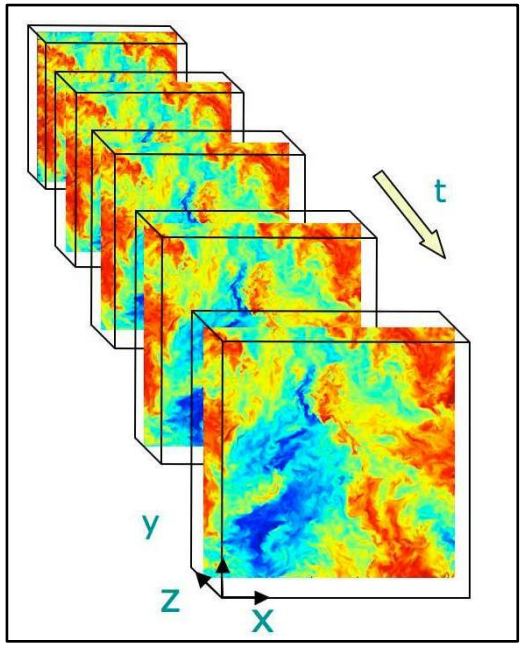
1) ID number
2) Outcome (R = recur, N = nonrecur)
3) Time (recurrence time if field 2 = R, disease-free time if field 2 = N)
4-33) Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)"

UCI Machine Learning Repository

## Datatype: Image data

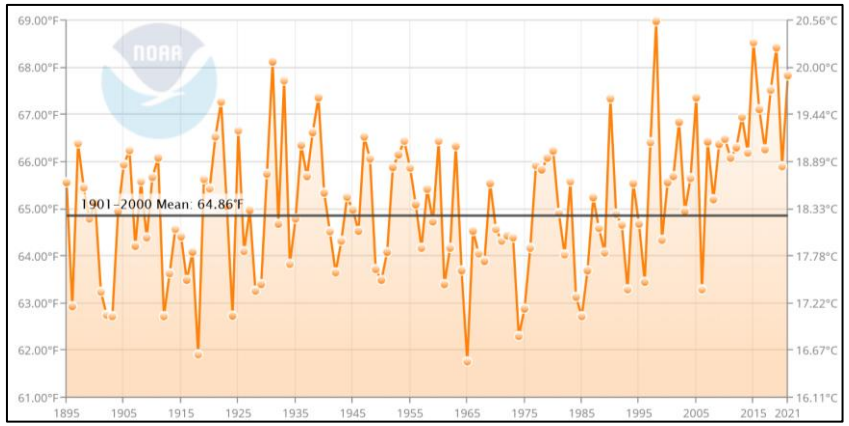
**Velocity and pressure for fluid flow**




Johns Hopkins Turbulence Databases

## Datatype: Time series data

**Contiguous U.S. Average Temperature**




NOAA NATIONAL CENTERS FOR ENVIRONMENTAL INFORMATION
NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION

# Convolutional Neural Network (CNN)

- Input: A 3D tensor (width, height, depth); e.g., if the input is an image, the depth is 3, i.e., Red, Green, Blue channels.
- Output: a 3D tensor (width, height, depth)
- A convolutional layer: Convolution between two functions $f$ and $g$ is defined as

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x} - \mathbf{z})d\mathbf{z}$$

- For 1D discrete case, the integral turns into a sum

$$(f * g)(i) = \sum_a f(a)g(i - a)$$

- For 2D case

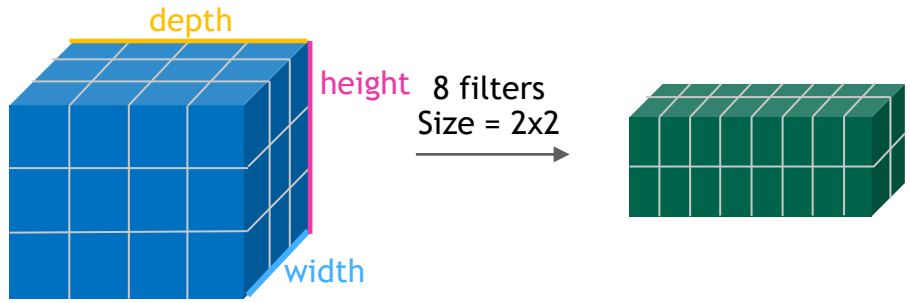
$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b)$$

- Rewrite above as

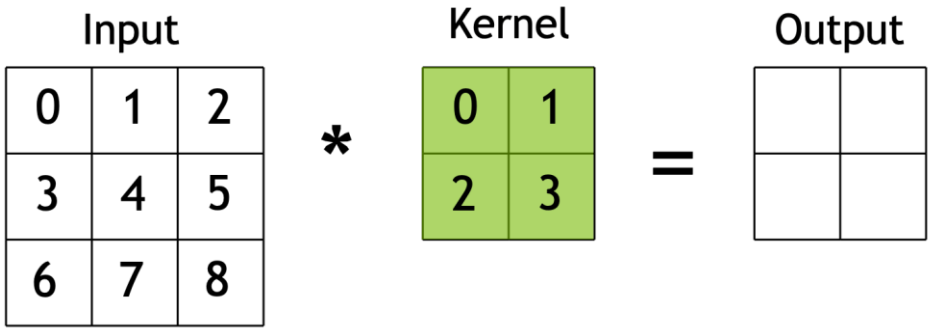
$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}$$

- For multiple channels:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [\mathsf{V}]_{a,b,c,d}[\mathsf{X}]_{i+a,j+b,c}$$



# Convolution: example




Kernel size: (2,2), Input size: (3,3)
⇒ No. of parameters: 2*2, Output size: (2,2)

Kernel size: (K, K), Input size: (W, H)
⇒ No. of parameters: K*K, Output size: (W-K+1,H-K+1)

```
In [20]: %matplotlib inline
import torch
from matplotlib import pyplot as plt
import torch.nn as nn

class conv2D_PyT(nn.Module):
    def __init__(self, kernel):
        super().__init__()
        self.weight = nn.Parameter(kernel)

    def conv(self, x, k):
        r, c = self.weight.shape
        #dimension of convolved output:
        ##(height of input - height of kernel + 1, width of input - width of kernel + 1)
        cr, cc = x.shape[0] - r + 1, x.shape[1]-c + 1
        y = torch.zeros((cr, cc))
        for i in range(cr):
            for j in range(cc):
                temp = x[i:i+cr, j:j + cc]*k
                y[i, j] = (temp).sum()
        return y

    def forward(self, x):
        return self.conv(x, self.weight)


#Inpurt
x = torch.tensor([[0,1,2], [3,4,5], [6,7,8]])
# Kernel
k = torch.tensor([[0.0,1.0], [2.0,3.0]])
m = conv2D_PyT(k)

y = m(x)
print(f"Convolution Output in Torch is ")
print(f":{y.detach().numpy()}")
```

```
Convolution Output in Torch is
:[[19. 25.]
 [37. 43.]]
```

```
import tensorflow as tf

class conv2D_tf(tf.keras.layers.Layer):
    def __init__(self, kernel):
        super(conv2D_tf, self).__init__()
        self.kernel = kernel
        self.weight = tf.Variable(self.kernel)

    def conv(self, x):
        r, c = self.weight.shape
        cr, cc = x.shape[0] - r + 1, x.shape[1]-c + 1
        y = tf.Variable(tf.zeros((cr, cc)))
        for i in range(cr):
            for j in range(cc):
                temp = tf.math.multiply(x[i:i+cr, j:j + cc], self.weight)
                y[i, j].assign(tf.reduce_sum(temp))
        return y

    def call(self, inputs):
        return self.conv(inputs)

#Inpurt
x = tf.constant([[0,1,2], [3,4,5], [6,7,8]], dtype='float32')
# Kernel
k = tf.constant([[0.0,1.0], [2.0,3.0]], dtype='float32')
m = conv2D_tf(k)
print(f"Convolution Output in TF is ")
print(f":{m(x).numpy()}")
```

```
Convolution Output in TF is
:[[19. 25.]
 [37. 43.]]
```

# FNN vs CNN

**FNN**

$$T^l(x) = \boldsymbol{W}^l x + \boldsymbol{b}^l$$

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathbf{W}]_{i,j,k,l}[\mathbf{X}]_{k,l}$$

$$= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathbf{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b}$$

**CNN**

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}$$

## Properties of CNN

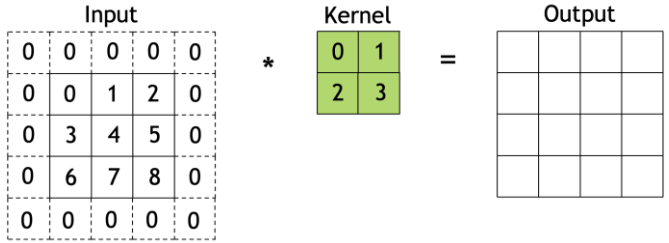
- Efficiency: Much fewer parameters $(W \times H)^2 \rightarrow K^2$
- Locality:
  1. Nearby pixels are typically related to each other
  2. We should not have to look very far away from location $(i, j)$ to compute $H(i, j)$
- Translation Invariance: A shift in the input $\boldsymbol{X}$ should simply lead to a shift in the hidden representation $\boldsymbol{H}$

# Padding

- By applying convolution layer, we lose pixels
- Adding extra pixels around the boundary of our input
- Padding type: constant (0), reflection, etc.

Kernel size: (2,2), Input size: (3,3), padding size: 1
⇒ Output size: (4, 4)

Kernel size: (K, K), Input size: (W, H), padding size: P
⇒ Output size: (W-K+2P+1,H-K+2P+1)



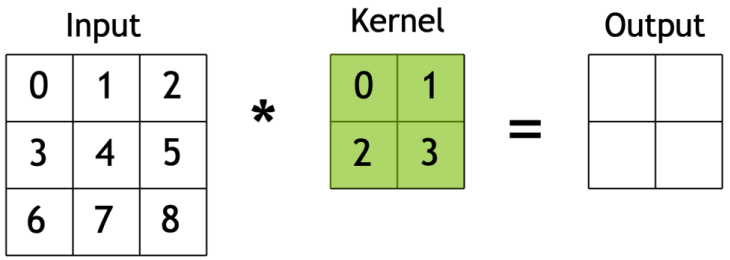

```
import torch
import torch.nn as nn
import numpy as np

## Input data charecterstics
n_batch = 1
n_channel = 1
n_row = 3
n_col = 3

## Reshaping the input
x = torch.zeros(1, 1, 3, 3)
x[:,:,...] = torch.tensor([[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]])

### Defining a convolution layers
m = torch.nn.Conv2d(in_channels = n_channel, out_channels = n_channel, kernel_size = 2, stride=1, padding=1)

## user Defined kernel
k = torch.tensor([[0,1], [2,3]])
kn = torch.Tensor([[0 ,1],[2, 3]]).unsqueeze(0).unsqueeze(0)

## Changing parametrs of Layers
m.parametrs = nn.Parameter(kn)
m.weight = nn.Parameter(kn)
m.bias = nn.Parameter(torch.tensor([0.0]))

### Convolution + Padding Output
y = m(x)

print(f"Convolved + Padded Output y: {y.detach().numpy()[0][0]}")
```

```
Convolved + Padded Output y: [[ 0.  3.  8.  4.]
 [ 9. 19. 25. 10.]
 [21. 37. 43. 16.]
 [ 6.  7.  8.  0.]]
```

# Padding

By applying a convolutional layer, we lose pixels. Adding extra pixels around the boundary of our input

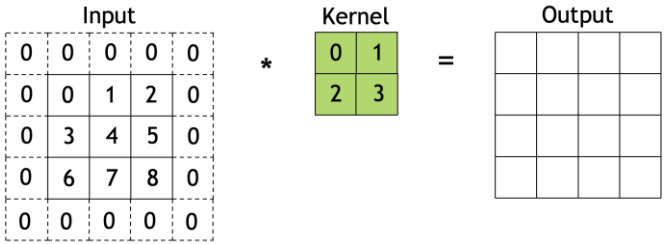
## Convolution




```
import tensorflow as tf
input_shape = (1, 3, 3, 1)
x = [[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]]
x = np.expand_dims(x, axis=0)
x = np.expand_dims(x, axis=-1)


def kernel_init(shape, dtype=None, partition_info=None):
    kernel = np.zeros(shape)
    kernel[:,:,0,0] = np.array([[0,1], [2,3]])
    return kernel


y = tf.keras.layers.Conv2D(1, 2, activation=None, kernel_initializer=kernel_init, input_shape=x[1:])(x)
print("Convolved Output y in TF2.0:")
print(f"{y.numpy()[0]}")
```

```
Convolved Output y in TF2.0:
[[[19.]
  [25.]]

 [[37.]
  [43.]]]
```

## Convolution + Padding




```
import tensorflow as tf
input_shape = (1, 3, 3, 1)
x = [[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]]
x = np.expand_dims(x, axis=0)
x = np.expand_dims(x, axis=-1)


def kernel_init(shape, dtype=None, partition_info=None):
    kernel = np.zeros(shape)
    kernel[:,:,0,0] = np.array([[0,1], [2,3]])
    return kernel


y = tf.keras.layers.ZeroPadding2D(padding=(1, 1))(x)
y = tf.keras.layers.Conv2D(1, 2, activation=None, padding='valid', strides=(1,1),\
                           kernel_initializer=kernel_init, input_shape=x[1:])(y)

print("Convolved Output y in TF2.0:")
print(f"{y.numpy()[0,:,:,0]}")
```

```
Convolved Output y in TF2.0:
[[ 0.  3.  8.  4.]
 [ 9. 19. 25. 10.]
 [21. 37. 43. 16.]
 [ 6.  7.  8.  0.]]
```

# Stride

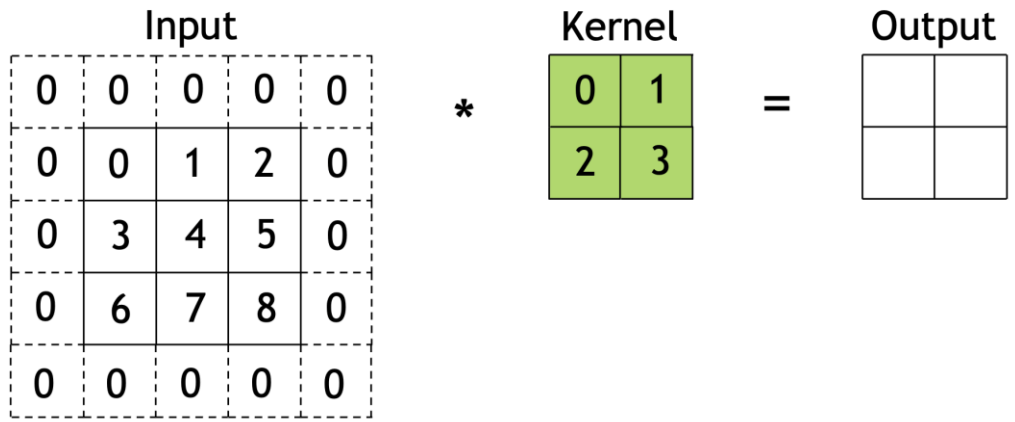
- (1) For computational efficiency or (2) for downsampling
- Move the window by more than one element at a time

Kernel size: (2,2), Input size: (3,3), padding size: 1, stride: 2
$\Rightarrow$ Output size: (2, 2)
Kernel size: (K, K), Input size: (W, H), padding size: P, stride: S
$\Rightarrow$ Output size: $(\frac{W-K+2P}{S}+1, \frac{H-K+2P}{S}+1)$

```python
import torch
import torch.nn as nn
import numpy as np

## Input data charecterstics
n_batch = 1
n_channel = 1
n_row = 3
n_col = 3

## Reshaping the input
x = torch.zeros(1, 1, 3, 3)
x[:,:,...] = torch.tensor([[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]])


### Defining a convolution layers
m = torch.nn.Conv2d(in_channels = n_channel, out_channels = n_channel, kernel_size = 2, stride=(3,2), padding=1)

## user Defined kernel
k = torch.tensor([[0,1], [2,3]])
kn = torch.Tensor([[0 ,1],[2, 3]]).unsqueeze(0).unsqueeze(0)

## Changing parametrs of Layers
m.parametrs = nn.Parameter(kn)
m.weight = nn.Parameter(kn)
m.bias = nn.Parameter(torch.tensor([0.0]))

### Convolution + Padding Output + Striding
y = m(x)

print(f"Convolved + Padded + Strided Output y: {y.detach().numpy()[0][0]}")
```

```
Convolved + Padded + Strided Output y: [[0. 8.]
 [6. 8.]]
```

Stride = (3, 2)



# Stride

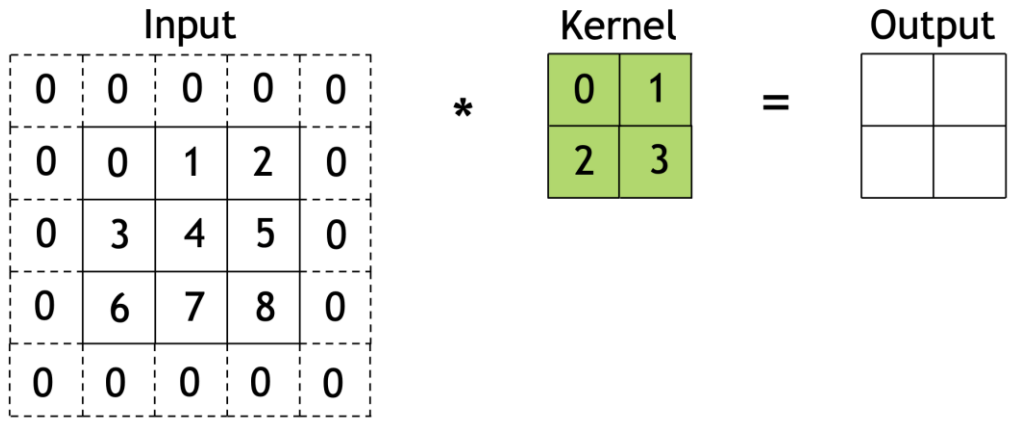
- (1) For computational efficiency or (2) for downsampling
- Move the window by more than one element at a time

Kernel size: (2,2), Input size: (3,3), padding size: 1, stride: 2
$\Rightarrow$ Output size: (2, 2)
Kernel size: (K, K), Input size: (W, H), padding size: P, stride: S
$\Rightarrow$ Output size: $(\frac{W-K+2P}{S}+1, \frac{H-K+2P}{S}+1)$

```
import tensorflow as tf
input_shape = (1, 3, 3, 1)
x = [[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]]
x = np.expand_dims(x, axis=0)
x = np.expand_dims(x, axis=-1)


def kernel_init(shape, dtype=None, partition_info=None):
    kernel = np.zeros(shape)
    kernel[:,:,0,0] = np.array([[0,1], [2,3]])
    return kernel


y = tf.keras.layers.ZeroPadding2D(padding=(1, 1))(x)
y = tf.keras.layers.Conv2D(1, 2, activation=None, padding='valid', strides=(3,2),\
                           kernel_initializer=kernel_init, input_shape=x[1:])(y)

print("Convolved + Padded Output y in TF2.0:")
print(f"{y.numpy()[0,:,:,0]}")
```

```
Convolved + Padded Output y in TF2.0:
[[0. 8.]
 [6. 8.]]
```

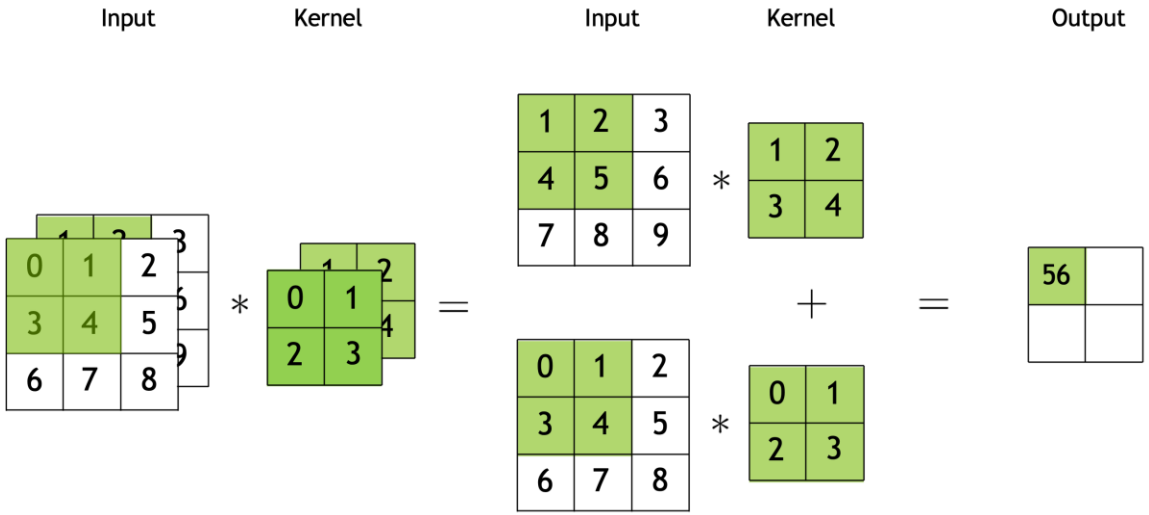
Stride = (3, 2)



# Multiple Input and Multiple Output Channels

Multiple input channels

- E.g., color images have the standard RGB channels to indicate the amount of red, green and blue
- Input: (3,3,2), kernel: (2,2)
- 8 parameters, Output: (2,2,1)
- Input: (W,H,D), kernel: (K,K)
- K*K*D parameters,
  Output: (W-K+1, H-K+1, 1)




Multiple Output Channels

- Input: (W,H,D), N kernels of size (K,K)
- K*K*D parameters per kernel, Output: (W-K+1, H-K+1, N)

# Summary: A Convolution Layer

- Input size: $W_1, H_1, D_1$
- Required 4 hyperparameters:

  Number of kernels: $N$

  Kernel size: $K$

  Stride: $S$

  Amount of Padding: $P$

- Output size: $W_2, H_2, D_2$

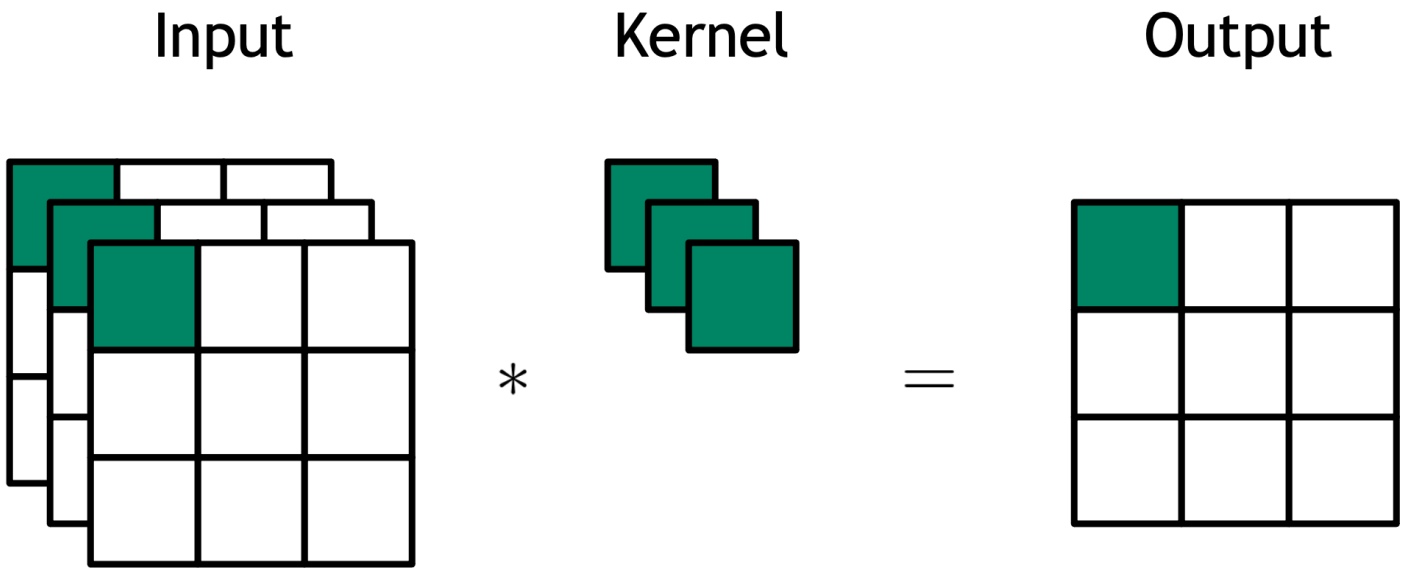
$$W_2 = \frac{W_1 - K + 2P}{S} + 1$$
$$H_2 = \frac{H_1 - K + 2P}{S} + 1$$
$$D_2 = N$$

Common setting:
$N$ = powers of 2, e.g., 32, 64, 128, 512

- $K = 3, S = 1, P = 1$
- $K = 5, S = 1, P = 1$
- $K = 5, S = 2, P =?$ (whatever fits)
- $K = 1, S = 1, P = 0$

- $K * K * D_1$ parameters per kernel. A total of $K * K * D_1 * N$ parameters and $N$ biases
- In the output, the $d^{th}$ channel is the result of performing a convolution of the $d^{th}$ kernel over the input

# 1x1 Convolutional Layer



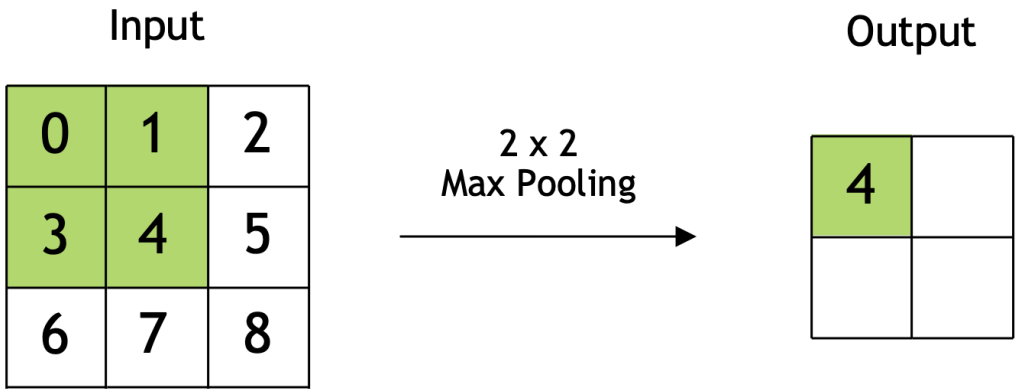

- K=1
- Computation of the convolution occurs on the channel dimension
- Inputs and outputs have the same size
- Each element in the output is derived from a linear combination of elements *at the same position* in the input image
- Can be used for dimensionality reduction

# Pooling

Gradually reduce the spatial resolution of our hidden representations

Maximum Pooling and Average Pooling

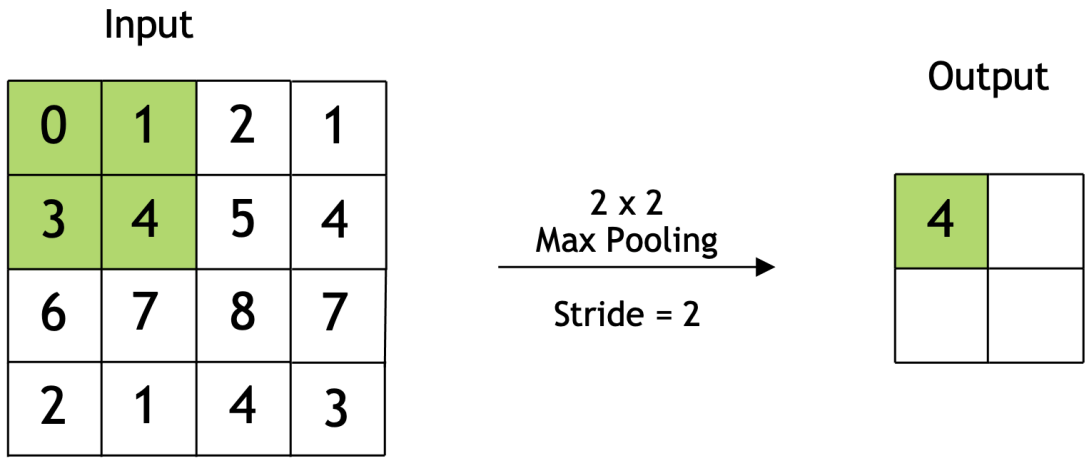
No parameters!




```
x = torch.zeros(1, 1, 3, 3)
x[:,:,...] = torch.tensor([[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]])

m = nn.MaxPool2d(2, stride=1)

y = m(x)

print(f"Max pooled Output y: {y.detach().numpy()[0][0]}")
```

```
Maxpooled Output y: [[4. 5.]
 [7. 8.]]
```

Pooling can also have Padding and Stride




```
x = torch.zeros(1, 1, 4, 4)
x[:,:,...] = torch.tensor([[0.0,1.0,2.0, 1.0], [3.0,4.0,5.0, 4.0], [6.0,7.0,8.0, 7.0], [2.0,1.0,4.0,3.0]])

m = nn.MaxPool2d(2, stride=2)

y = m(x)

print(f"Max pooled Output y: {y.detach().numpy()[0][0]}")
```

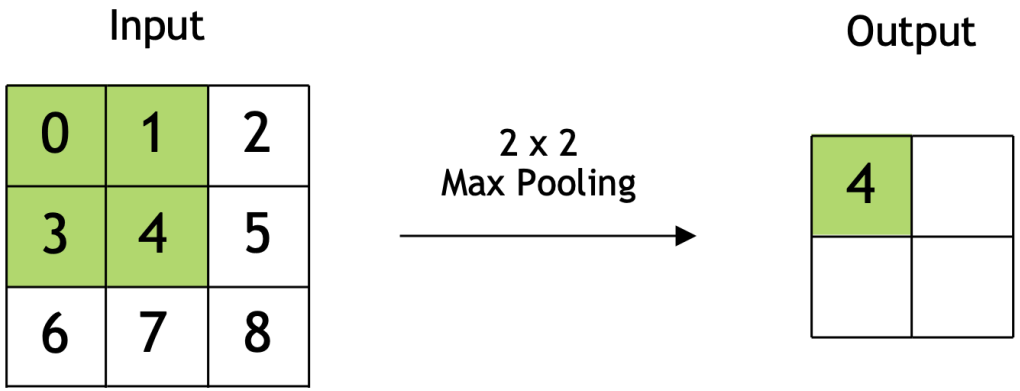
```
Max pooled Output y: [[4. 5.]
 [7. 8.]]
```

# Pooling

Gradually reduce the spatial resolution of our hidden representations

Maximum Pooling and Average Pooling

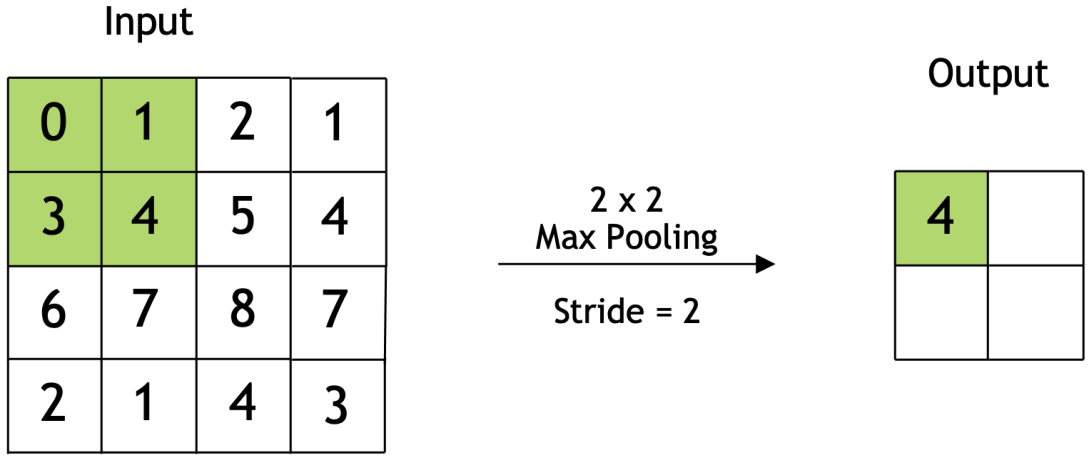
No parameters!




```
import tensorflow as tf
input_shape = (1, 3, 3, 1)
x = [[0.0,1.0,2.0], [3.0,4.0,5.0], [6.0,7.0,8.0]]
x = np.expand_dims(x, axis=0)
x = np.expand_dims(x, axis=-1)

y = tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), strides=(1,1), padding='valid')(x)

print("MaxPooled Output y in TF2.0:")
print(f"{y.numpy()[0,:,:,0]}")
```

```
MaxPooled Output y in TF2.0:
[[4. 5.]
 [7. 8.]]
```

Pooling can also have Padding and Stride




```
import tensorflow as tf
input_shape = (1, 3, 3, 1)
x = [[0.0,1.0,2.0, 1.0], [3.0,4.0,5.0, 4.0], [6.0,7.0,8.0, 7.0],[2.0,1.0,4.0,3.0]]
x = np.expand_dims(x, axis=0)
x = np.expand_dims(x, axis=-1)

y = tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), strides=(2,2), padding='valid')(x)

print("MaxPooled Output y in TF2.0:")
print(f"{y.numpy()[0,:,:,0]}")
```

```
MaxPooled Output y in TF2.0:
[[4. 5.]
 [7. 8.]]
```

# Example: AlexNet

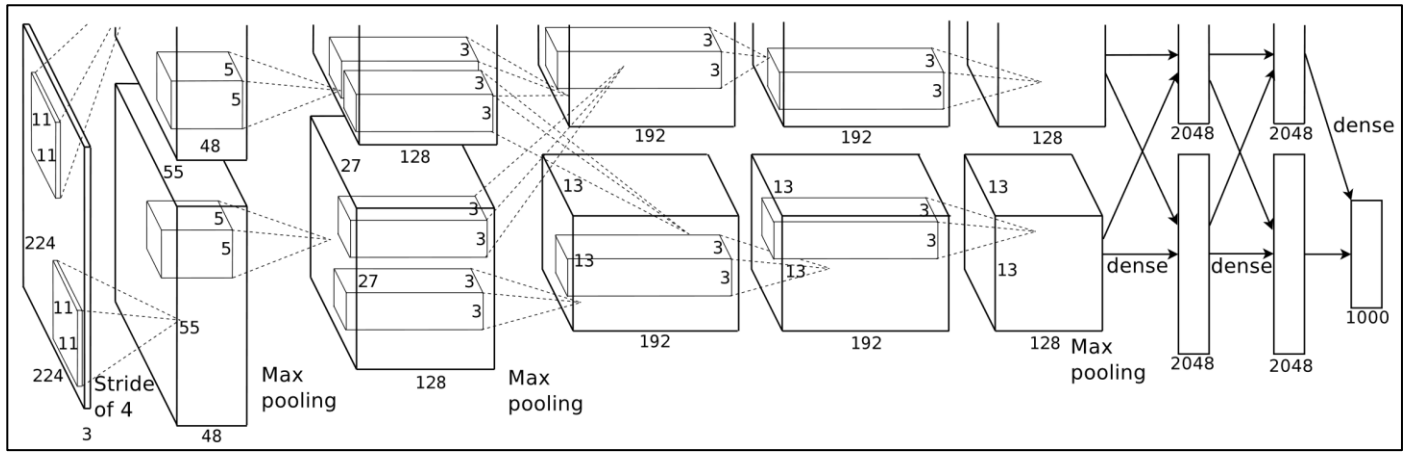



Image: 224 (height) × 224 (width) × 3 (channels)

↓

Convolution with 11×11 kernel+4 stride: 54×54×96

↓ ReLu

Pool with 3×3 max. kernel+2 stride: 26×26×96

↓

Convolution with 5×5 kernel+2 pad: 26×26×256

↓ ReLu

Pool with 3×3 max. kernel+2 stride: 12×12×256

↓

Convolution with 3×3 kernel+1 pad: 12×12×384

↓ ReLu

Convolution with 3×3 kernel+1 pad: 12×12×384

↓ ReLu

Convolution with 3×3 kernel+1 pad: 12×12×256

↓ ReLu

Pool with 3×3 max. kernel+2 stride: 5×5×256

↓ flatten

Dense: 4096 fully connected neurons

↓ ReLu, dropout p=0.5

Dense: 4096 fully connected neurons

↓ ReLu, dropout p=0.5

Dense: 1000 fully connected neurons

↓

Output: 1 of 1000 classes

AlexNet achieved a top-5 error of 15.3%, more than 10.8% lower than that of the runner up.


Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems. 2012;25:1097-105.

https://en.wikipedia.org/wiki/AlexNet

# Universality of Deep CNNs

## Theorem

Let $2 \leq s \leq N$. For and compact subset $\Omega$ of $\mathbb{R}^N$ and any $f \in C(\Omega)$, there exist sequences $\boldsymbol{w}$ of filter masks, $\boldsymbol{b}$ of bias vectors, and $f_L^{\boldsymbol{w},\boldsymbol{b}} \in \mathcal{H}_L^{w,b}$ such that $\lim_{L\to\infty} ||f - f_L^{\boldsymbol{w},\boldsymbol{b}}||_{C(\Omega)} = 0$



Zhou DX. Universality of deep convolutional neural networks. Applied and computational harmonic analysis. 2020 Mar 1;48(2):787-94.

# CNN Kernel vs Finite Difference Stencil

## Partial derivative

$$\frac{\partial \phi}{\partial x} = \frac{\phi(x+\delta x) - \phi(x-\delta x)}{2\delta x} + O(\delta x)^2 \quad \longleftrightarrow \quad \frac{\partial \phi}{\partial x} = \begin{bmatrix} -\frac{1}{2\delta x} & 0 & \frac{1}{2\delta x} \\ -\frac{1}{2\delta x} & 0 & \frac{1}{2\delta x} \\ -\frac{1}{2\delta x} & 0 & \frac{1}{2\delta x} \end{bmatrix}$$

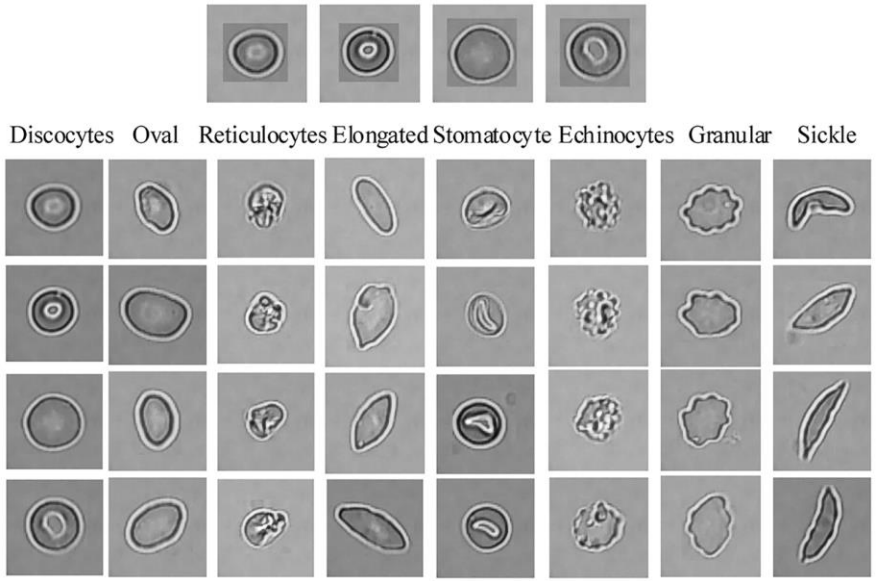
## 2D Laplacian (Δ)

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_{i,j} + \left.\frac{\partial^2 T}{\partial y^2}\right|_{i,j} \rightarrow \quad \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{2,3} \\ k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Standard 5-point stencil finite difference kernel

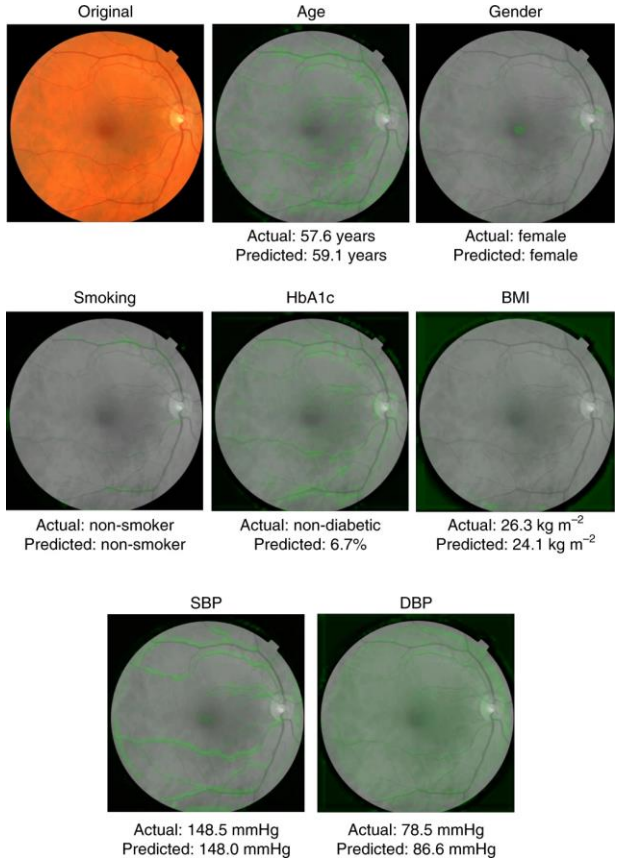
Learned kernel

# Applications of CNN in Science and Engineering



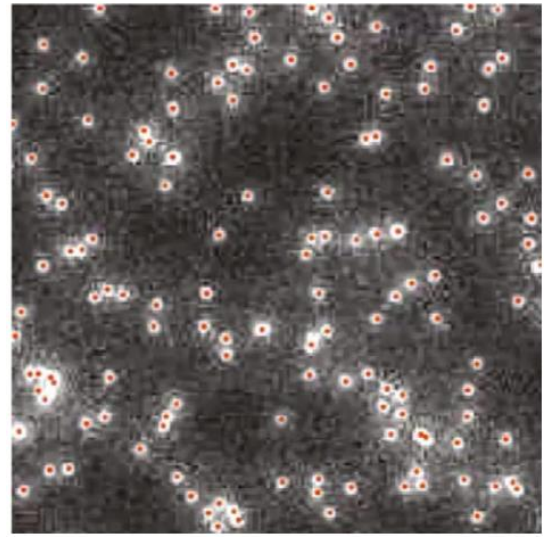

A deep convolutional neural network for classification of red blood cells in sickle cell anemia.

*PLOS Computational Biology, 2017*




Prediction of cardiovascular risk factors from retinal fundus photographs via deep learning.

*Nature Biomedical Engineering, 2018*.



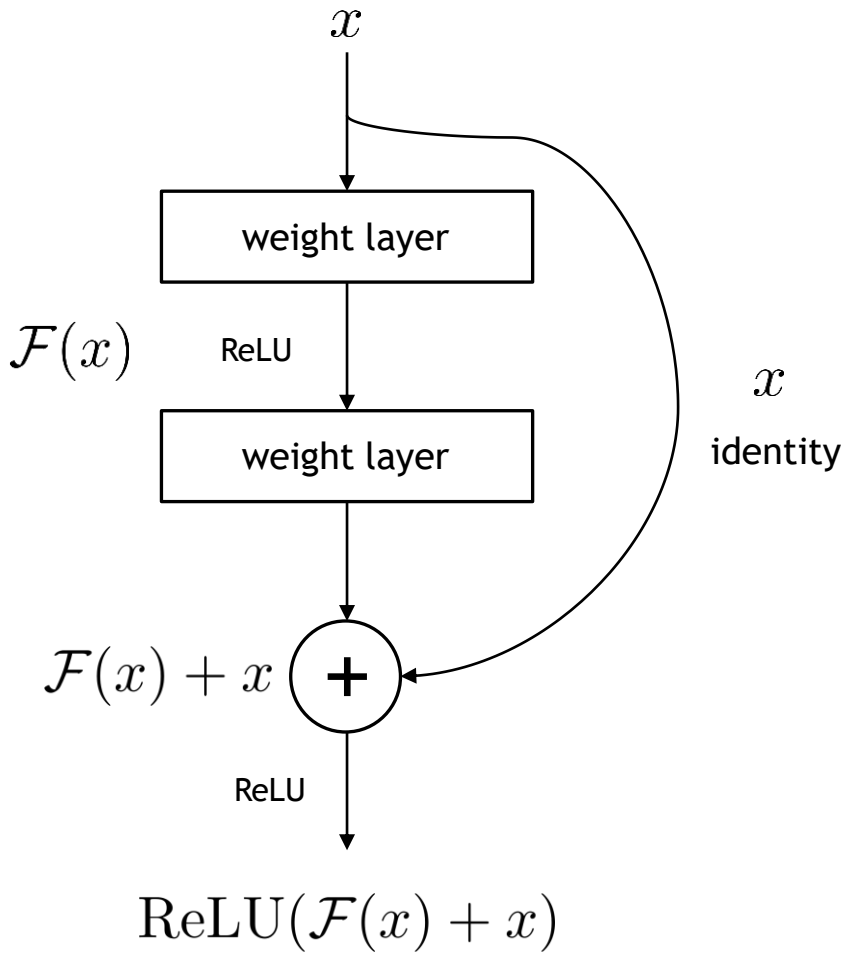
Detect microscopic particles (red marker)

*Machine learning for active matter.*

*Nature Machine Intelligence, 2020*

# Residual Neural Network

- Deep networks are hard to train: vanishing gradients
- Core idea: “identity shortcut connection” that skips one or more layers
- Widely used: simple and powerful




A residual block



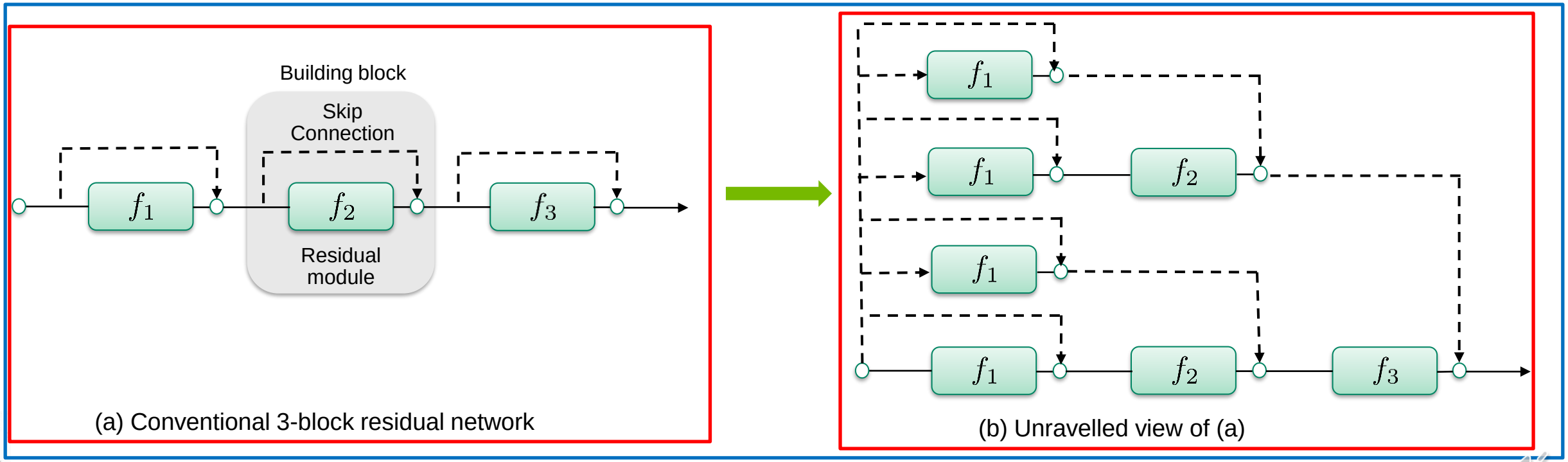
He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. InProceedings of the IEEE conference on computer vision and pattern recognition 2016 (pp. 770-778).

# ResNet: Unraveled View

Output of ResNet reads:

$$
\begin{aligned}
y_3 &= y_2 + f_3(y_2) \\
&= [y_1 + f_2(y_1)] + f_3(y_1 + f_2(y_1)) \\
&= [y_0 + f_1(y_0) + f_2(y_0 + f_1(y_0))] + f_3(y_0 + f_1(y_0) + f_2(y_0 + f_1(y_0)))
\end{aligned}
$$

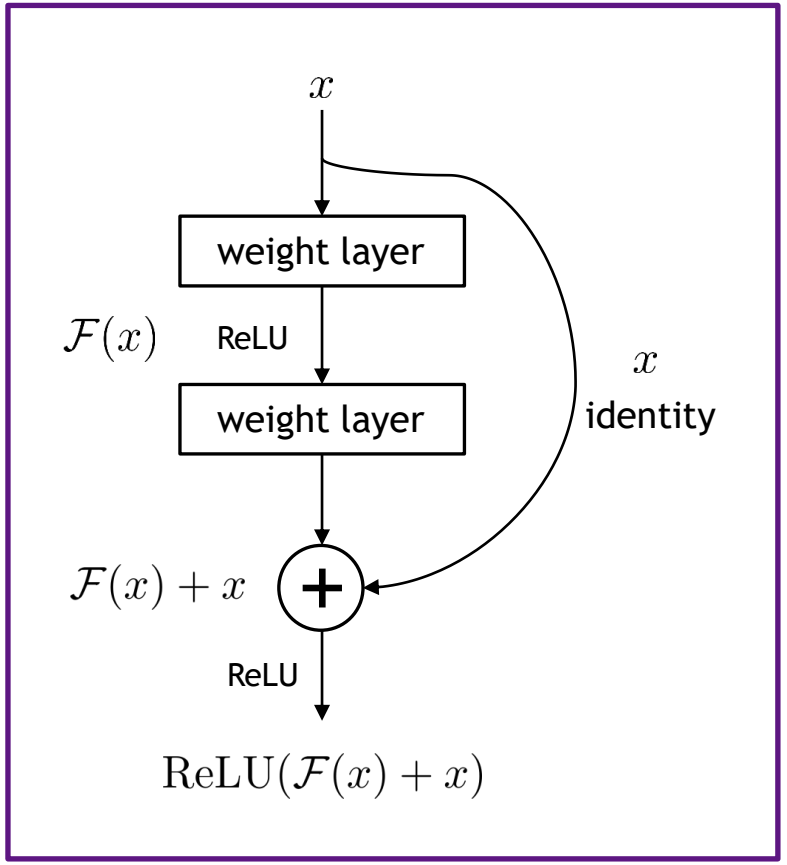
No. of paths connecting input to output layers: $2^n$






• Veit A, Wilber MJ, Belongie S. Residual networks behave like ensembles of relatively shallow networks. Advances in neural information processing systems. 2016;29:550-8.

# ResNet = Forward Euler?



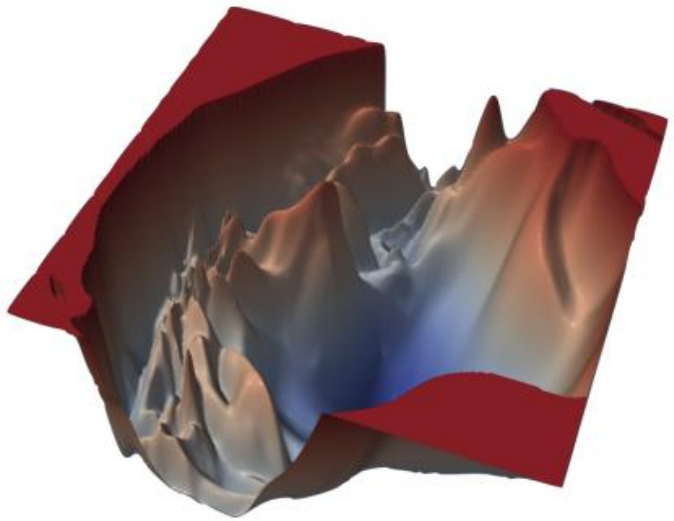

$$x_{n+1} = x_n + F(x_n)$$

$$u_{n+1} = u_n + \Delta t f(x_n) \quad \textbf{(Mass transportation)}$$

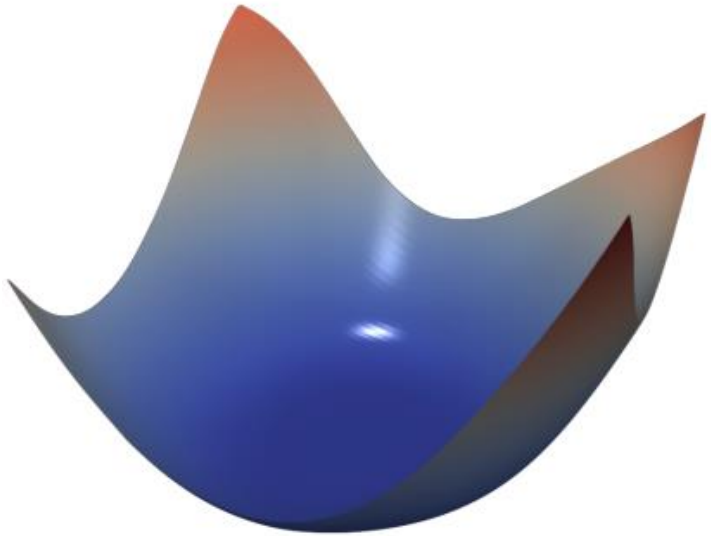
$$\frac{du}{dt} = f(u) \quad \textbf{(Dynamical system)}$$

Weinan E. A proposal on machine learning via dynamical systems. Communications in Mathematics and Statistics. 2017 Mar 1;5(1):1-1.

# ResNet vs Feed-forward



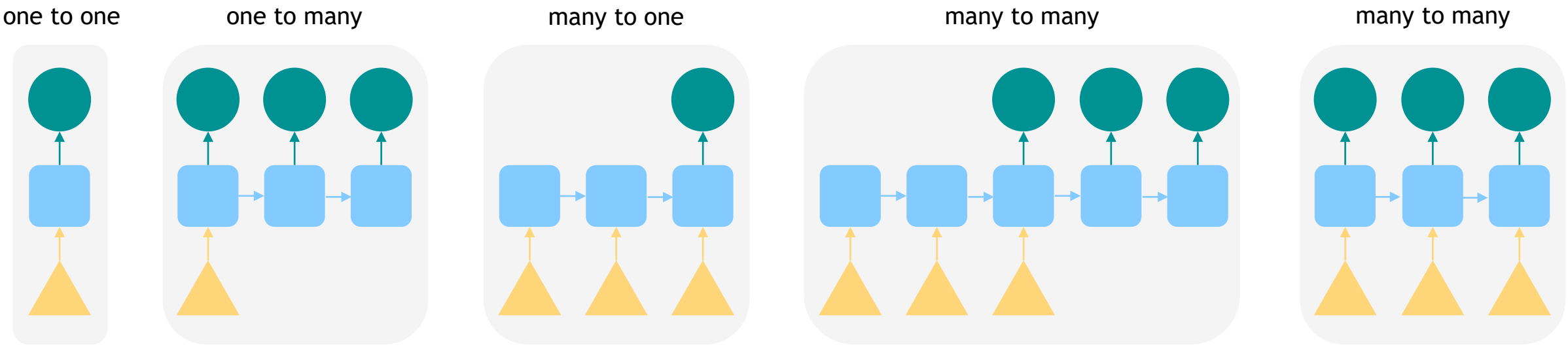
ResNet-110, no skip connections



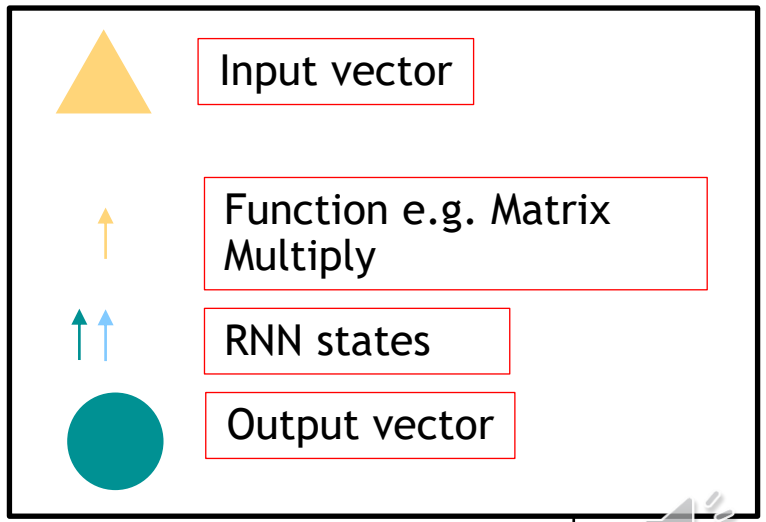
DenseNet-121 Layers, skip connections

**Loss function  tends to increase its convexity**

# Modeling of Sequence Data




1) Vanilla mode: fix size input to fix size output: No RNN
2) Sequence Output: Image captioning
3) Sequence input: classifying positive or negative sentiment
4) Sequence input to Sequence Output: Machine Translation
5) Synced Sequence input and output: Labeling each frame of video



# Prediction of Sequence Data (time series)

- Sequence data: $x_1, x_2, ..., x_t, ...$

$$P(x_t|x_{t-1}, ..., x_{t-n+1})$$

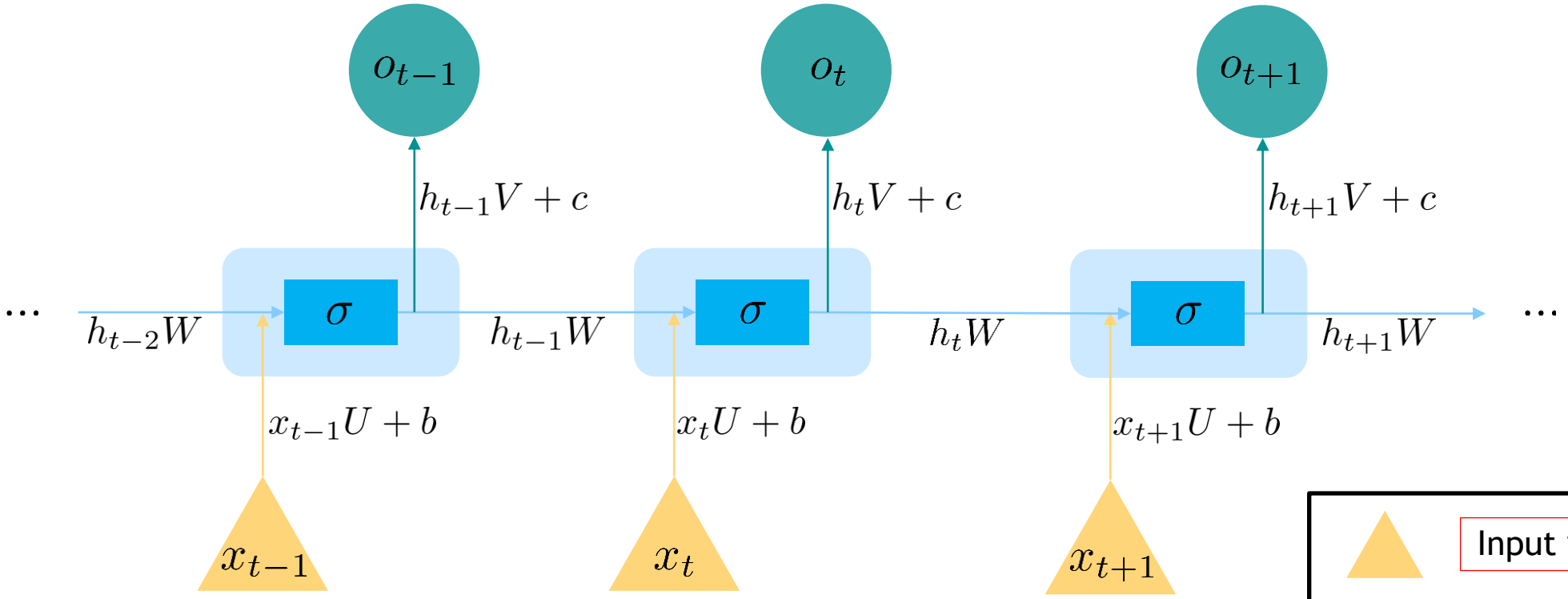
- Use a latent variable model:

$$P(x_t|x_{t-1}, ..., x_{t-n+1}) \approx P(x_t|h_{t-1})$$

- where $h_t$ is a hidden state (a hidden variable) that stores the sequence information up to time step $t-1$
- In general, the hidden state at any time step $t$ is computed based on both the current input $x_t$ and the previous hidden-state $h_{t-1}$:

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

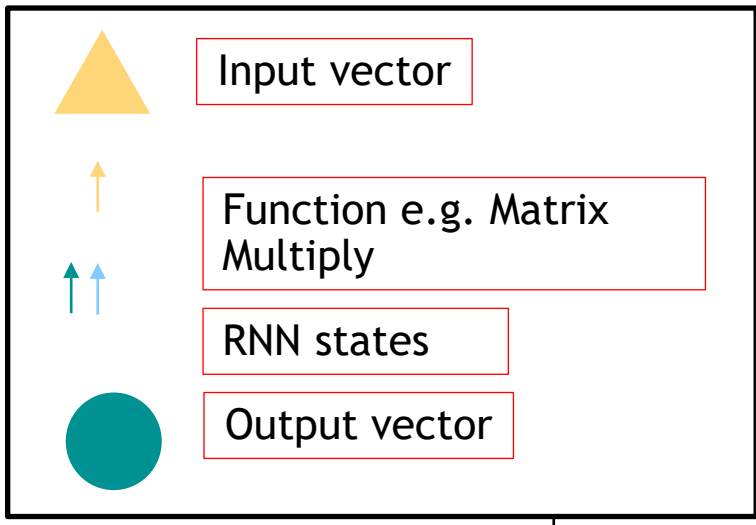
# Recurrent Neural Network



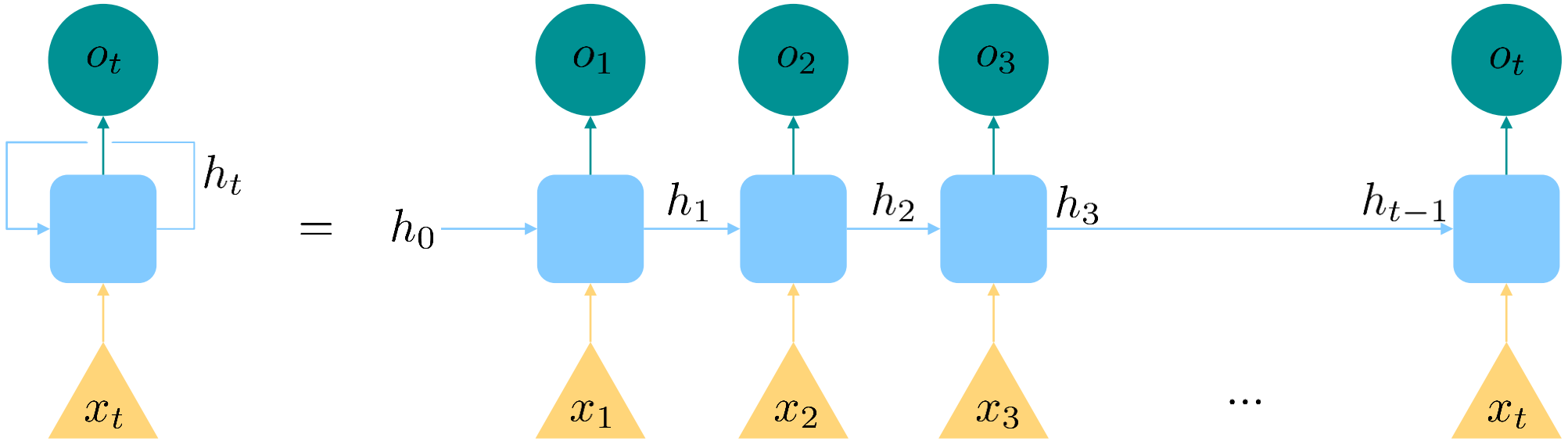

At time step t (recurrent layer)

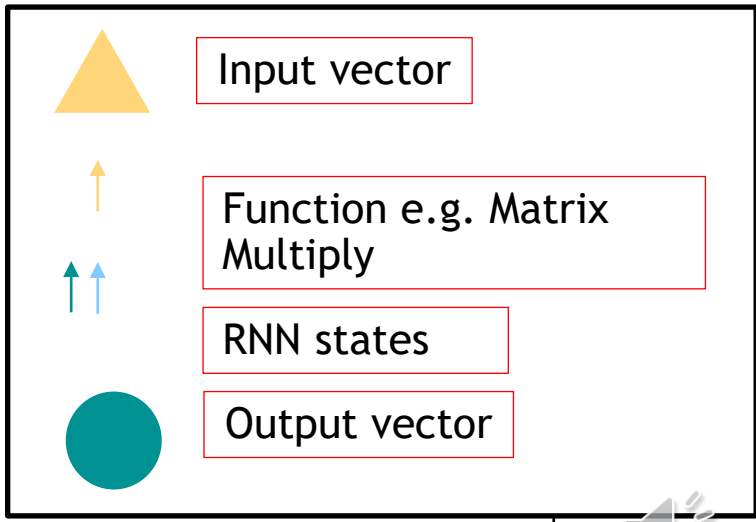
$$\text{Hidden state: } h_t = \sigma(h_{t-1}W + x_tU + b)$$

$$\text{Output State: } o_t = h_tV + c$$



# Unrolled RNN



# RNN for Sine Function

$$y = \sin(\omega x), \qquad \omega = \frac{1}{20}$$

## 1. Data Preparation

```
T = 20 # Period of Sin Function
L = 1000 # Sample Lenght of Sin function
N = 200   # Sequence Length


x = np.empty((N, L), 'int32')
x[:] = np.array(range(L))
data = np.sin(x * (1.0 / T)).astype('float32')
x_f = np.array(range(1500))
data_f = np.sin(x_f * (1.0 / T)).astype('float32')

train_data = torch.from_numpy(data[1:, :-1])
train_target = torch.from_numpy(data[1:, 1:])
test_data = torch.from_numpy(data[:1, :-1])
test_target = torch.from_numpy(data[:1, 1:])
```

## 2. Architecture

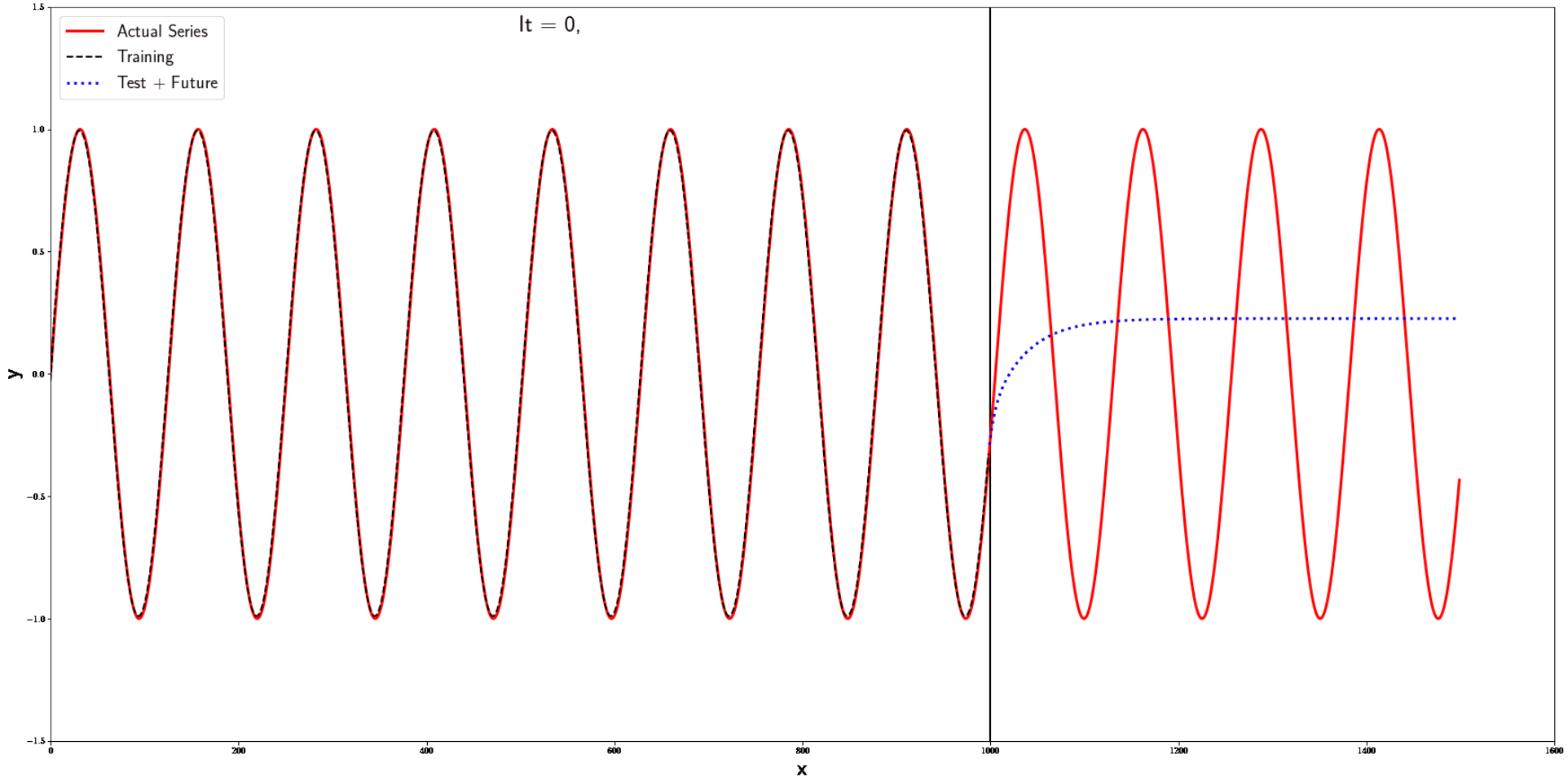
```
class RNNSin(nn.Module):
    def __init__(self, in_dim, h_dim, out_dim):
        super(RNNSin, self).__init__()
        self.rnn = nn.RNNCell(in_dim, h_dim)
        self.linear = nn.Linear(h_dim, out_dim)
        self.h_dim = h_dim

    def forward(self, train_data, future = 0):
        outputs = []
        h_t = torch.zeros((train_data.size(0),self.h_dim))
        for i, input_t in enumerate(train_data.chunk(train_data.size(1), dim=1)):
            h_t = self.rnn(input_t,h_t)
            output = self.linear(h_t)
            outputs.append(output)
        for i in range(future):
            h_t = self.rnn(outputs[-1],h_t)
            output = self.linear(h_t)
            outputs += [output]
        outputs = torch.stack(outputs, 1).squeeze(2)
        return outputs
```

## 3. Training and Prediction

```
seq = RNNSin(1,200,1)
loss_fn = nn.MSELoss()
optimizer = optim.LBFGS(seq.parameters(), lr=0.4)
Nepochs = 15

fig, ax = plt.subplots(figsize=(30,15))
image_list = []
for it in range(Nepochs):


    def closure():
        optimizer.zero_grad()
        out = seq(train_data)
        loss = loss_fn(out, train_target)
        print('\rEpoch:', it,'loss:', loss.item(), end='')
        loss.backward()
        return loss
    optimizer.step(closure)

    future = 500
    # begin to predict, no need to track gradient here
    with torch.no_grad():
        pred = seq(test_data, future=future)
        loss = loss_fn(pred[:, :-future], test_target)
        print(' test loss:', loss.item())
        y = pred.detach().numpy()
```

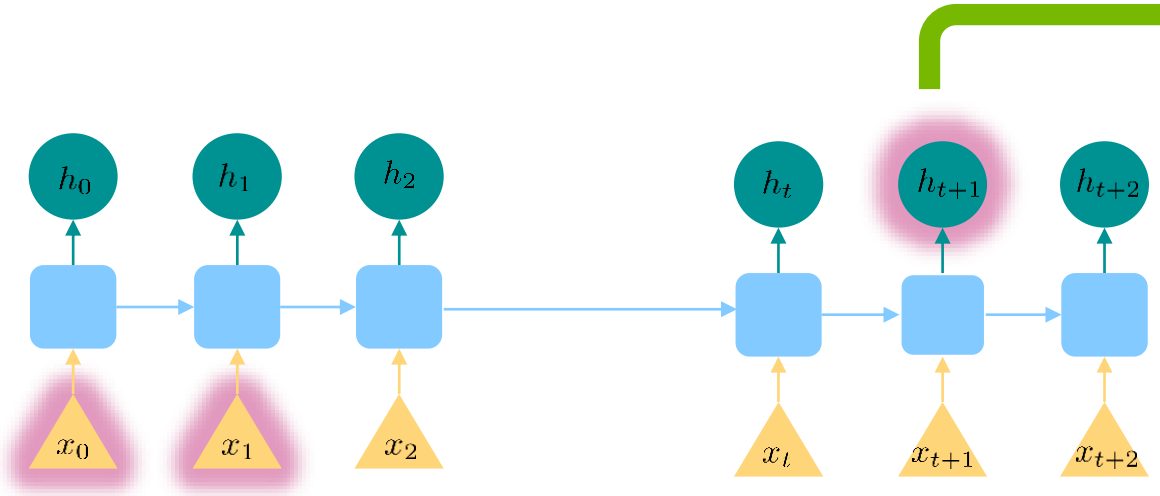
# RNN for Sine Function

$$y = \sin(\omega x), \qquad \omega = \frac{1}{20}$$



# Modelling Long Dependencies

The problem of Long-Term dependencies: vanishing and/or exploding gradients




$$\mathbf{h}^{(t)} = f(\mathbf{x}^{(t)}, \mathbf{h}^{(t-1)})$$

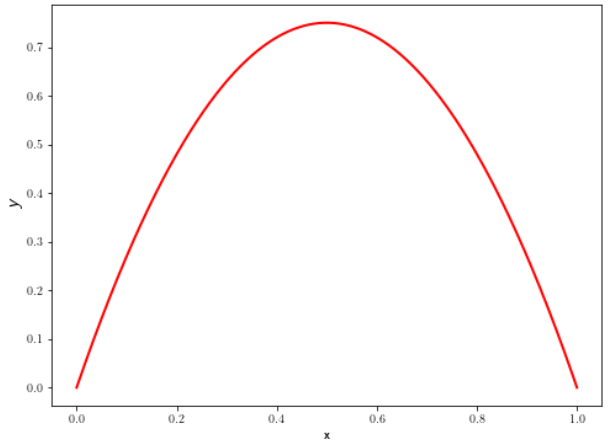
$$\mathbf{h}^{(4)} = f(f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}), \mathbf{x}^{(3)}), \mathbf{x}^{(4)})$$

- This looks a bit like repeatedly applying the function $f$.
- We can gain some intution for how RNNs behave by studying iterated functions, i.e., functions which we iterate many times.

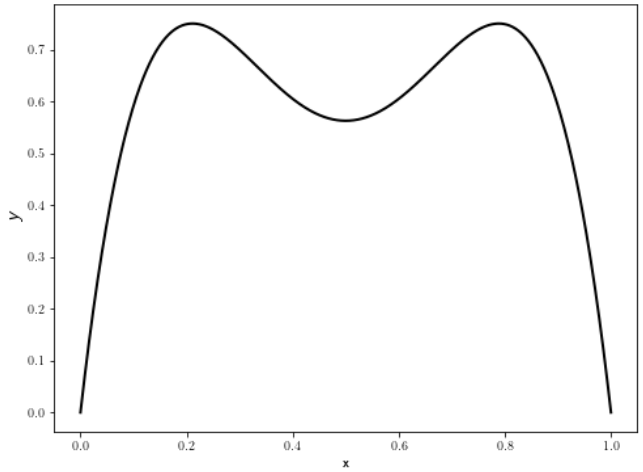
Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks. In International conference on machine learning 2013 May 26 (pp. 1310-1318). PMLR.
http://www.cs.Toronto.edu/~rgrosse/course/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf

# Why Gradients Explode or Vanish?

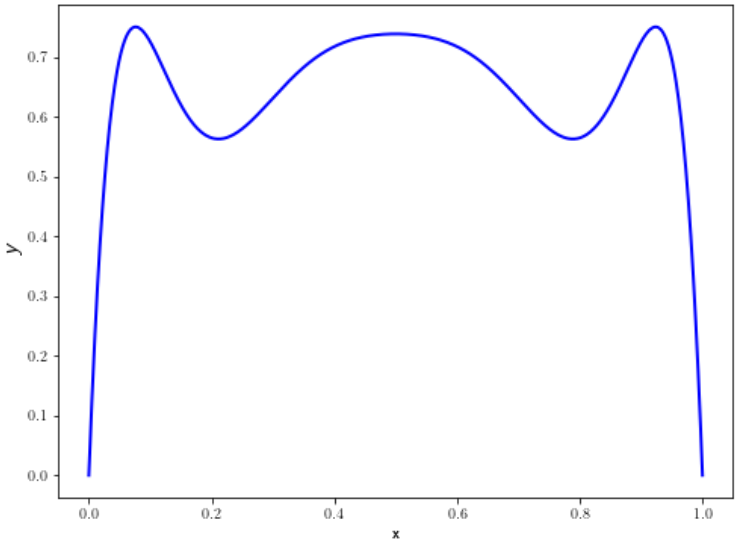
$$y = f(x) := 3x(1-x)$$



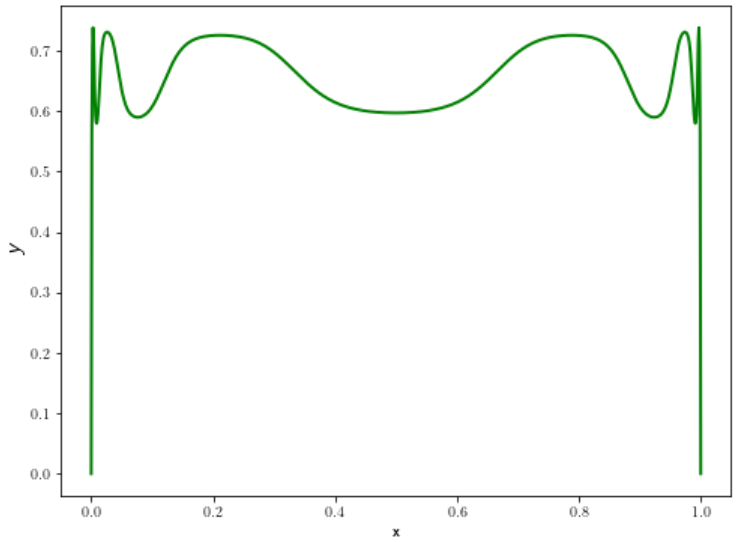

$$y = f(f(x))$$




$$y = f(f(f(x)))$$

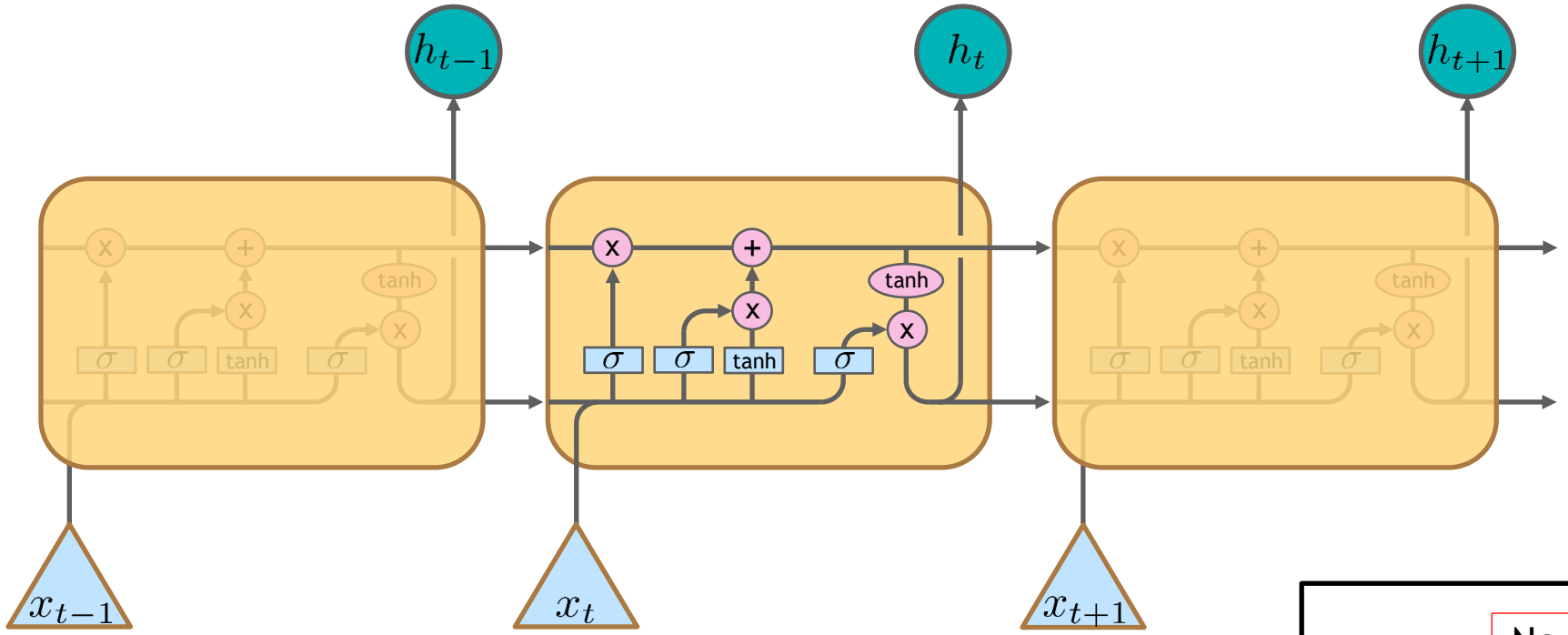



$$y = f \circ \ldots \circ f(x), \quad 7 \text{ composition}$$

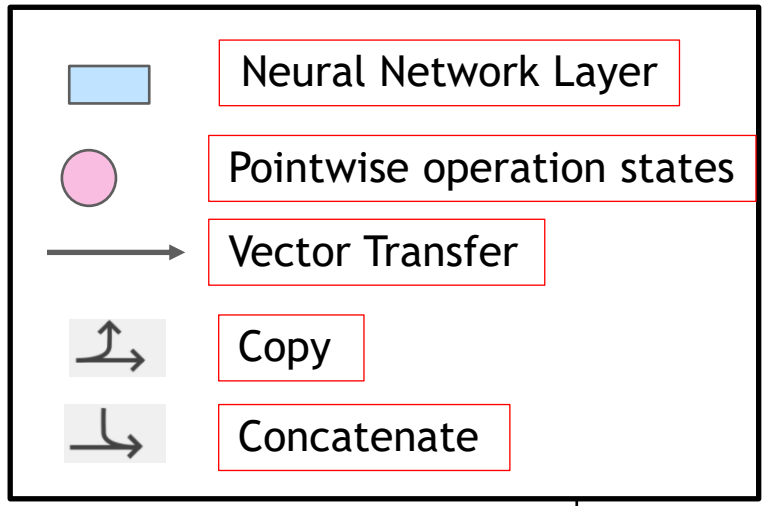



```python
import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import matplotlib.pyplot as plt

f = lambda x: 3*x*(1-x)

x = np.linspace(0, 1, 1000)
y0 = f(x)
y1 = f(y0)
y2 = f(y1)
y3 = f(y2)

y4 = f(f(f(f(y3))))



fig0, ax0 = plt.subplots(figsize=(8,6))


ax0.plot(x, y0, '-r', linewidth=2.0)
ax0.set_xlabel(r'\textbf{x}')
ax0.set_ylabel('\\textit{y}', fontsize=16)
```

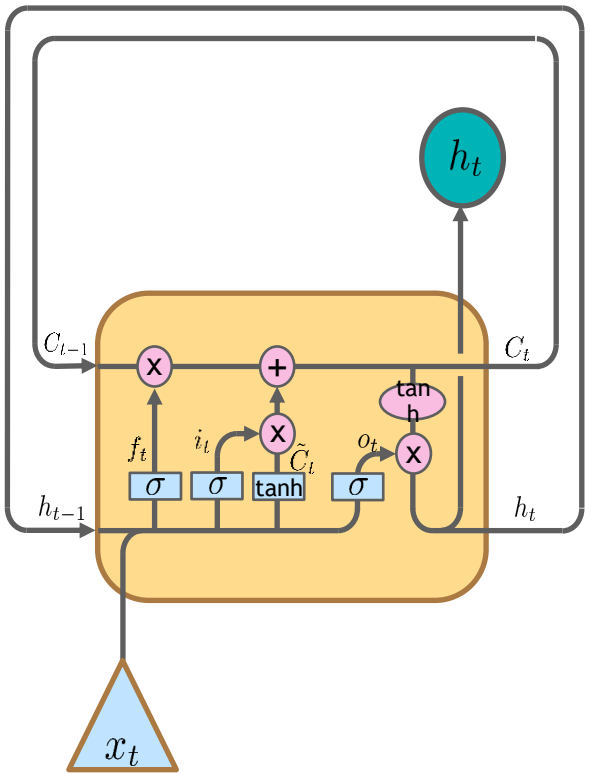
# Long Short-Term Memory Networks




The repeating module in an LSTM contains four interacting layers



# Long Short-Term Memory Networks




Single LSTM Block

Forget gate activation vector

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

Input/update gate activation vector

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$

Cell input activation vector

$$\tilde{C}_t = \tanh(W_C.[h_{t-1}, x_t] + b_C)$$

Cell state vector

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output gate's activation vector

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$

Hidden state vector

$$h_t = o_t * \tanh(C_t)$$

# LSTM for Sine Function

$$y = \sin(x)$$

## 1. Data Preparation

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dense
import numpy as np
import matplotlib.pyplot as plt

## Data Preperation
def seq_data(x, seq_length):
    X = []
    Y = []
    l = len(x)

    for i in range(l):
        end_id = i + seq_length
        if end_id > len(x) - 1:
            print("end id could not be bigger than series length")
            break
        X.append(x[i:end_id])
        Y.append(x[end_id])

    return np.array(X), np.array(Y)

x_sin = np.arange(0, 100, 0.1)
y_sin = np.sin(x_sin)
x_test = np.linspace(0, 100, 1000, endpoint=True)
y_test = np.sin(x_test)

x_test_plot = np.copy(x_test)
y_test_plot = np.copy(y_test)

seq_length = 20

x_train, y_train = seq_data(y_sin, seq_length)
x_test, y_test = seq_data(y_test, seq_length)

num_features = 1
train_shape = x_train.shape
test_shape = x_test.shape

x_train = x_train.reshape((train_shape[0], train_shape[1], num_features))
x_test = x_test.reshape((test_shape[0], test_shape[1], num_features))
```

## 2. Architecture

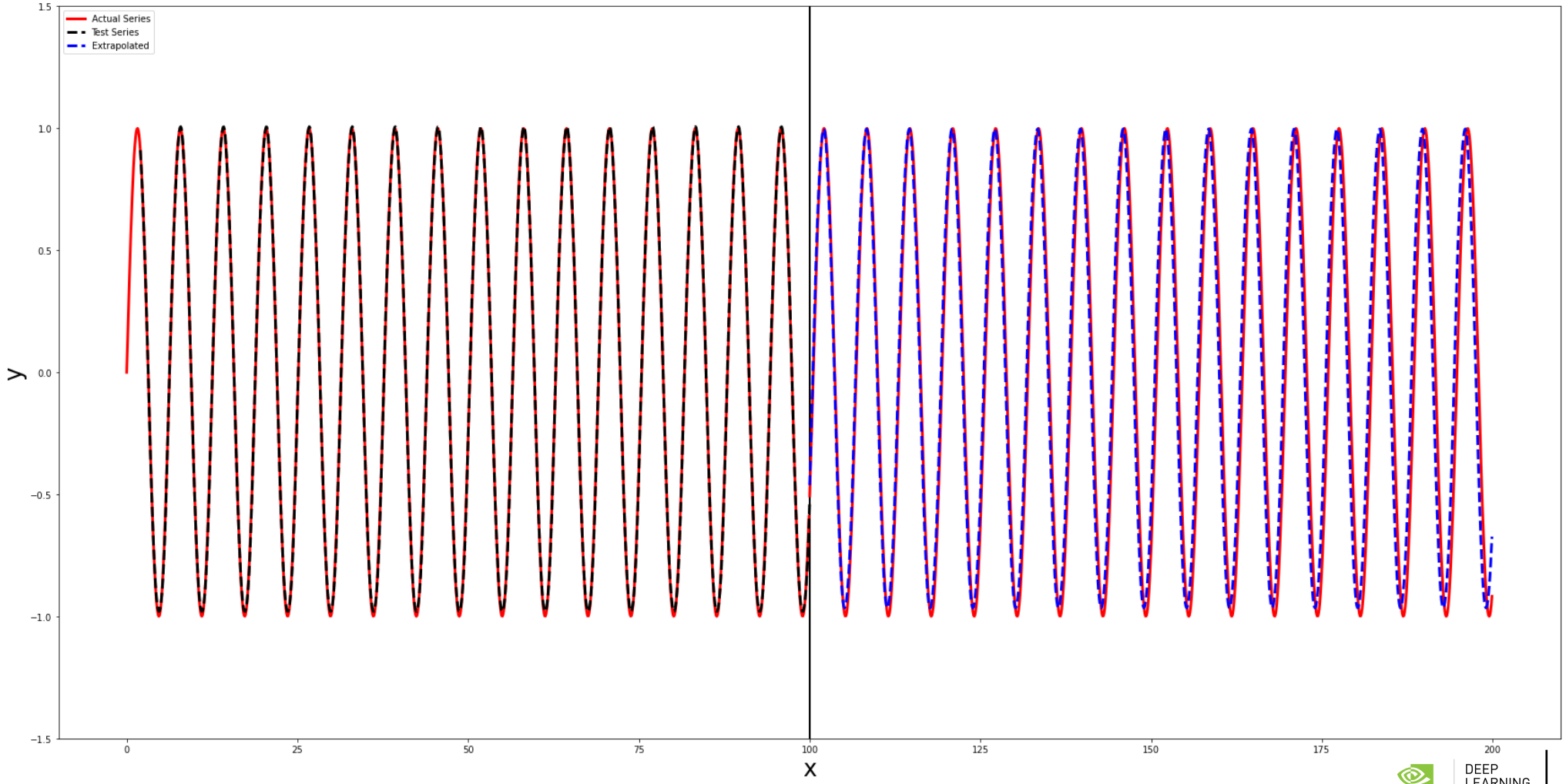
```
model = Sequential()
model.add(LSTM(10, input_shape = (seq_length, num_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

## 3. Training and Prediction

```
h = model.fit(x_train, y_train, epochs=20, verbose=1)
y_predict = model.predict(x_test)

pred_x = np.arange(80, 200, 0.1)
pred_y = np.sin(pred_x)
results = []
test_y = pred_y[:seq_length]

for i in range( len(pred_x) - seq_length):
    net_input = test_y[i : i + seq_length]
    net_input = net_input.reshape((1, seq_length, num_features))
    y = model.predict(net_input, verbose=0)
    test_y = np.append(test_y, y)
```

# LSTM for Sine Function

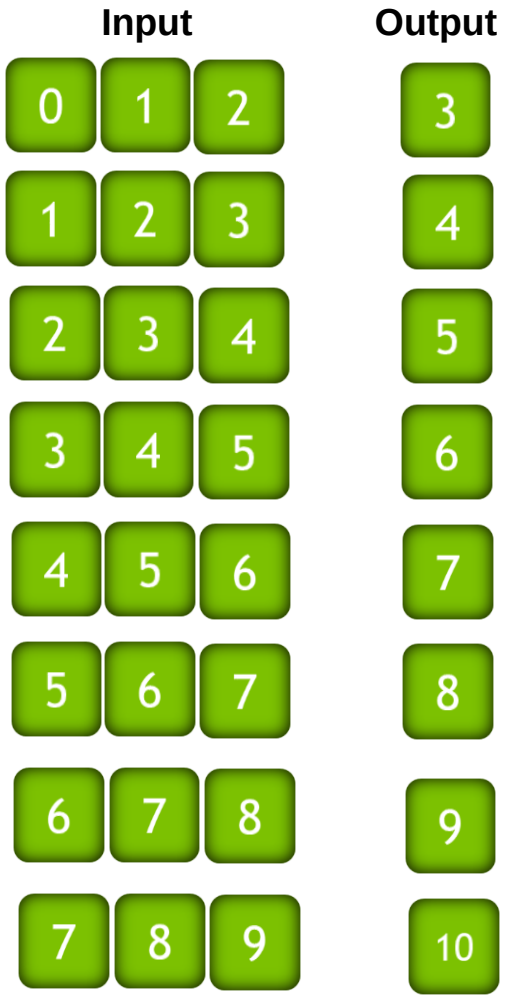
$$y = \sin(x)$$



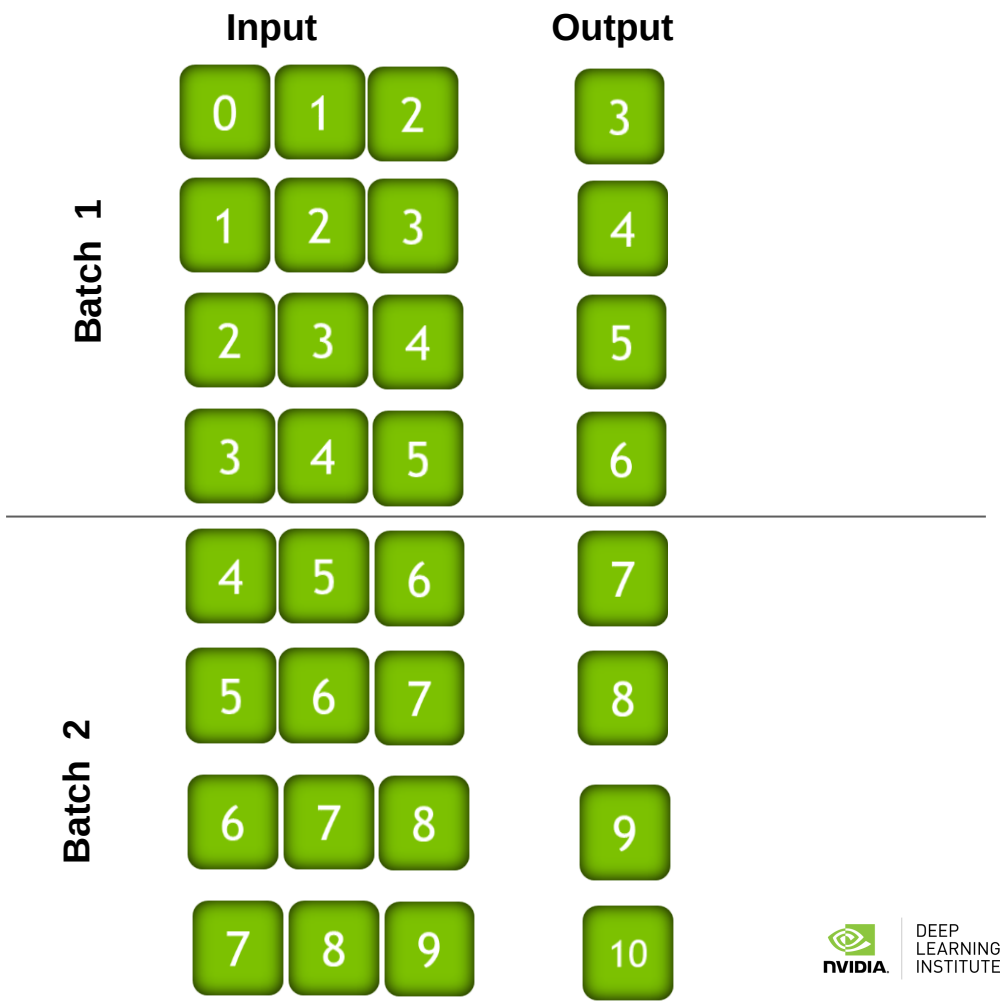
# Data Preparation for RNN and LSTM

Time Series Data


**Sequence Length = 3, Batches = 1, Batch Size = 8**




**Sequence Length = 3, Batches = 2, Batch Size = 4**

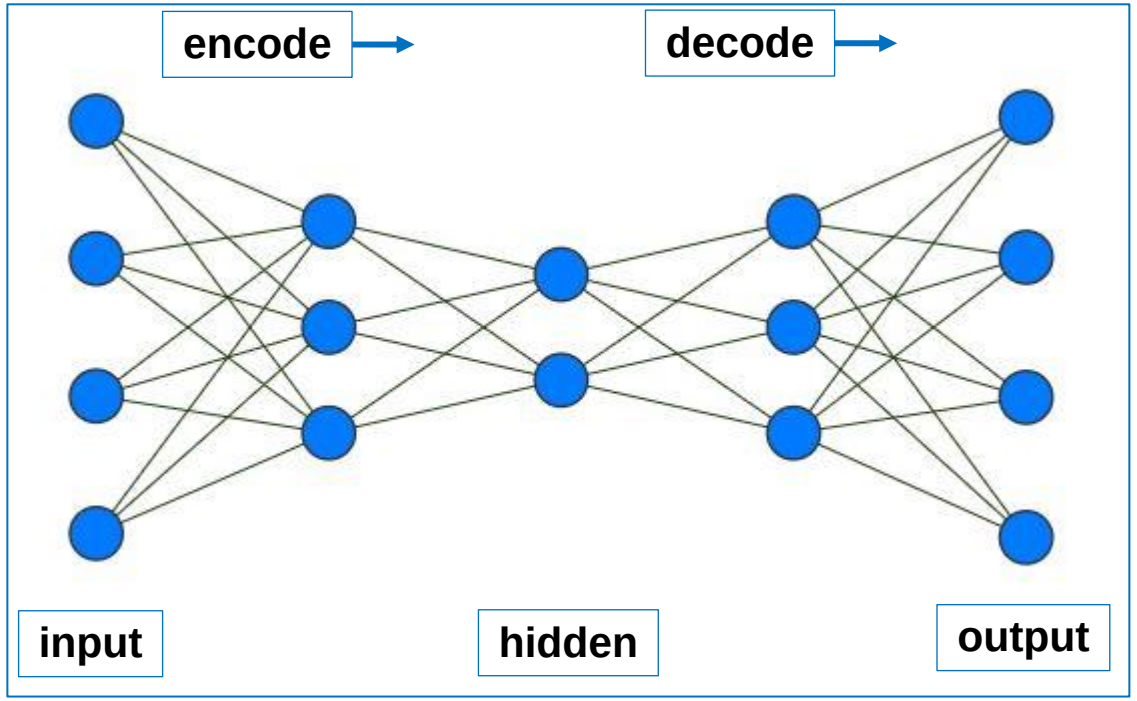


# Encoder-Decoder Architecture

- Encoder: takes an input and transforms it into a hidden state
- Decoder: maps the encoded state to an output


- Applications:
  - Dimensionality reduction
  - Sequence to Sequence Learning: Use RNN as encoder and decoder

# Neural Network Auto-Encoders




- Encoder-Decoder with input and output are the same.
- Dimensionality reduction: The bottleneck layer compresses the input to a low-dimensional latent space representation.
- Auto-encoders can be used to denoise the input.

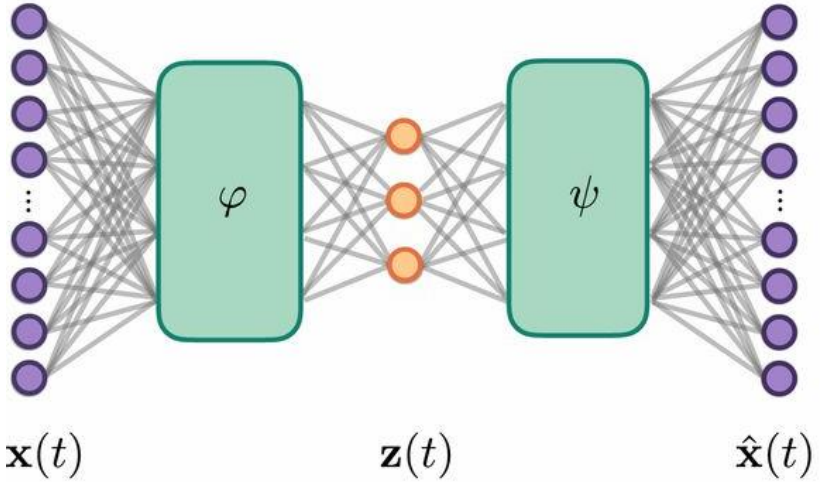
# Autoencoder: Data-driven discovery of coordinates

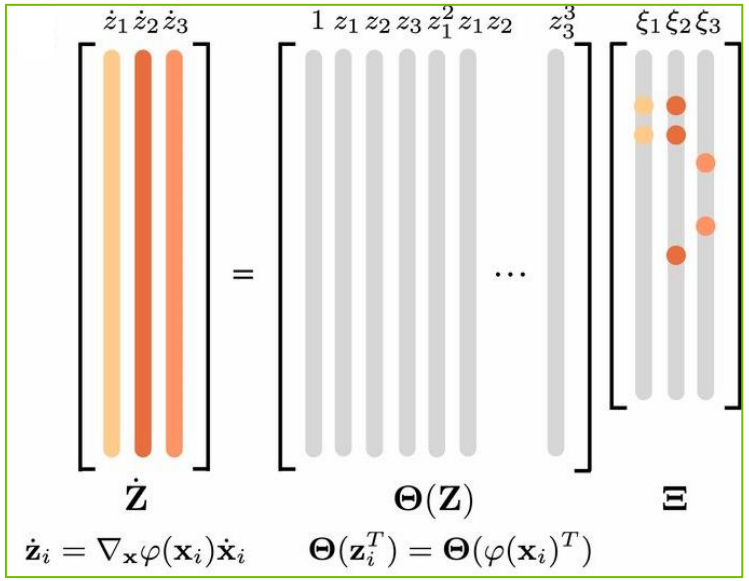
- Original measured coordinates $\mathbf{x}$

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t))$$

- Reduced coordinates $\mathbf{z}(r) = \varphi(\mathbf{x}(t)) \in \mathbb{R}^d (d << n)$

$$\frac{d}{dt}\mathbf{z}(t) = \mathbf{g}(\mathbf{z}(t))$$



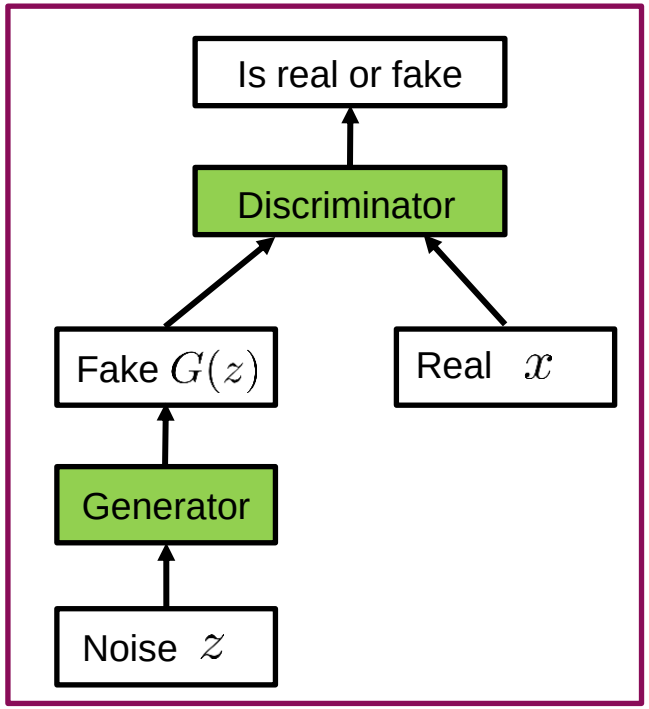

$$\underbrace{\|\mathbf{x} - \psi(\mathbf{z})\|_2^2}_{\text{reconstruction loss}} + \underbrace{\lambda_1 \left\|\dot{\mathbf{x}} - (\nabla_{\mathbf{z}}\psi(\mathbf{z}))\left(\Theta(\mathbf{z}^T)\Xi\right)\right\|_2^2}_{\text{SINDy loss in } \dot{\mathbf{x}}} + \underbrace{\lambda_2 \left\|(\nabla_{\mathbf{x}}\mathbf{z})\,\dot{\mathbf{x}} - \Theta(\mathbf{z}^T)\Xi\right\|_2^2}_{\text{SINDy loss in } \dot{\mathbf{z}}} + \underbrace{\lambda_3 \|\Xi\|_1}_{\text{SINDy regularization}}$$



• Champion K, Lusch B, Kutz JN, Brunton SL. Data-driven discovery of coordinates and governing equations. Proceedings of the National Academy of Sciences. 2019 Nov 5;116(45):22445-51.

# Generative Modeling

- Discriminative learning: Classifiers and regressors
- Generative modeling: Sample synthetic data examples that resemble the distribution of the training data
- Generative adversarial networks (GANs) leverage the power of discriminative models to get good generative models [Goodfellow et al., 2014]




Both networks are in competition with each other

- The generator network attempts to fool the discriminator network.
- The discriminator network adapts to the new fake data.

# References


- Arjovsky M, Chintala S, Bottou L. Wasserstein generative adversarial networks. InInternational conference on machine learning 2017 Jul 17 (pp. 214-223). PMLR.
- Champion K, Lusch B, Kutz JN, Brunton SL. Data-driven discovery of coordinates and governing equations. Proceedings of the National Academy of Sciences. 2019 Nov 5;116(45):22445-51.
- Cichos F, Gustavsson K, Mehlig B, Volpe G. Machine learning for active matter. Nature Machine Intelligence. 2020 Feb;2(2):94-103.
- Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y. Generative adversarial nets. Advances in neural information processing systems. 2014;27.
- Gulrajani I, Ahmed F, Arjovsky M, Dumoulin V, Courville A. Improved training of wasserstein gans. arXiv preprint arXiv:1704.00028. 2017.
- Hanin B, Sellke M. Approximating continuous functions by relu nets of minimal width. arXiv preprint arXiv:1710.11278. 2017 Oct 31
- He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition 2016 (pp. 770-778).
- http://www.cs.Toronto.edu/~rgrosse/course/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf
- https://en.wikipedia.org/wiki/AlexNet
- Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems. 2012;25:1097-105.
- Lin H, Jegelka S. Resnet with one-neuron hidden layers is a universal approximator. arXiv preprint arXiv:1806.10909. 2018 Jun 28.
- Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks. In International conference on machine learning 2013 May 26 (pp. 1310-1318). PMLR.
- Poplin R, Varadarajan AV, Blumer K, Liu Y, McConnell MV, Corrado GS, Peng L, Webster DR. Prediction of cardiovascular risk factors from retinal fundus photographs via deep learning. Nature Biomedical Engineering. 2018 Mar;2(3):158-64.
- Veit A, Wilber MJ, Belongie S. Residual networks behave like ensembles of relatively shallow networks. Advances in neural information processing systems. 2016;29:550-8.
- Weinan E. A proposal on machine learning via dynamical systems. Communications in Mathematics and Statistics. 2017 Mar 1;5(1):1-1.
- Weng L. From gan to wgan. arXiv preprint arXiv:1904.08994. 2019 Apr 18.
- Xu M, Papageorgiou DP, Abidi SZ, Dao M, Zhao H, Karniadakis GE. A deep convolutional neural network for classification of red blood cells in sickle cell anemia. PLoS computational biology. 2017 Oct 19;13(10):e1005746.
- Zhou DX. Universality of deep convolutional neural networks. Applied and computational harmonic analysis. 2020 Mar 1;48(2):787-94.


Deep Learning for Science and Engineering Teaching Kit

# Thank You