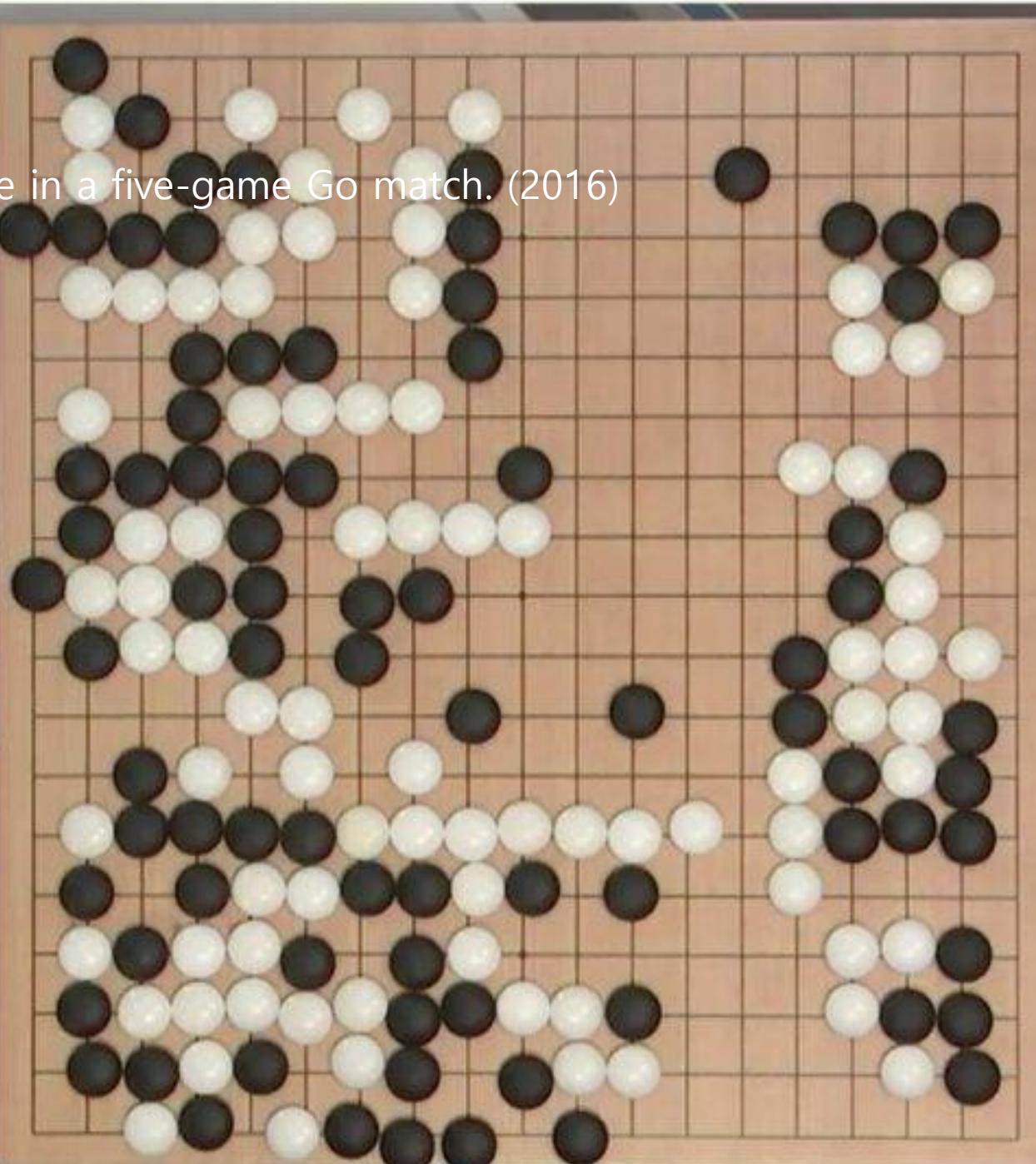


Deep Learning Tutorial with PyTorch

Sunglok Choi, Assistant Professor, Ph.D.
Dept. of Computer Science and Engineering, SEOULTECH
sunglok@seoultech.ac.kr | <https://mint-lab.github.io/>

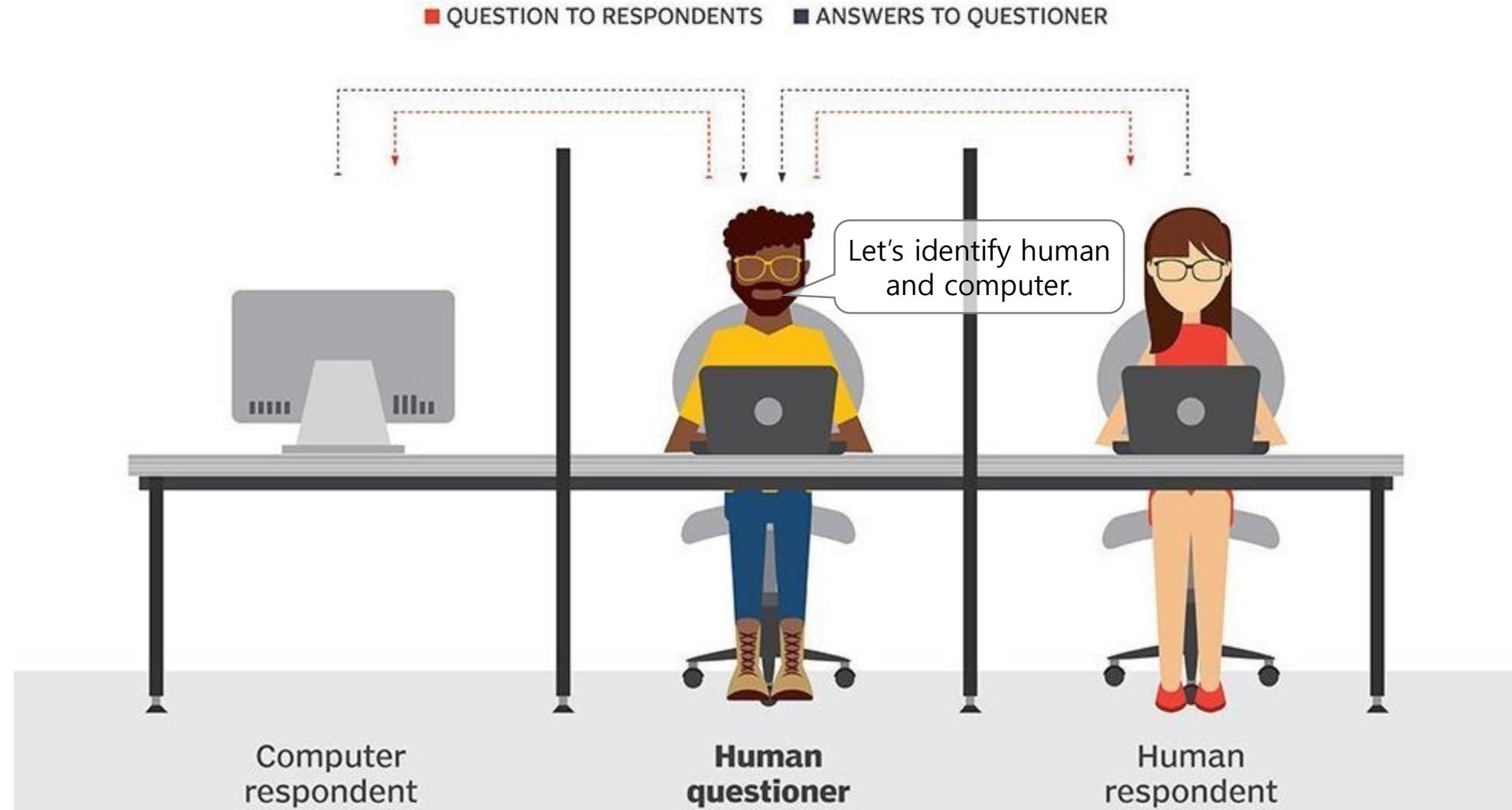
Why Deep Learning?

- AlphaGo beat Sedol Lee in a five-game Go match. (2016)



Why Deep Learning?

- ChatGPT broke the [Turing test](#). (2022)
 - Reference) Nature, Vol. 619, 2023 [DOI](#)



Why Deep Learning?

- ChatGPT gives *exact answers* to my **questions**, but Google retrieves the *nearest webpages* to my **queries**.



Why Deep Learning?

Title / Author	Cited by
Deep learning. Y LeCun, Y Bengio, G Hinton Nature 521 (7553), 436	27375
Human-level control through deep reinforcement learning. V Mnih, K Kavukcuoglu, D Silver, AA Rusu, J Veness, MG Bellemare, ... Nature 518 (7540), 529-533	10394
Mastering the game of Go with deep neural networks and tree search. D Silver, A Huang, CJ Maddison, A Guez, L Sifre, G van den Driessche, ... Nature 529 (7587), 484	7698
Analysis of protein-coding genetic variation in 60,706 humans. M Lek, KJ Karczewski, EV Minikel, KE Samocha, E Banks, T Fennell, ... Nature 536 (7616), 285-291	6387
A global reference for human genetic variation. A Auton, LD Brooks, RM Durbin, EP Garrison, HM Kang, JO Korbel, ... Nature 526 (7571), 68-74	6011
Dermatologist-level classification of skin cancer with deep neural networks. A Esteva, B Kuprel, RA Novoa, J Ko, SM Swetter, HM Blau, S Thrun Nature 542 (7639), 115-118	4076
Compositional engineering of perovskite materials for high-performance solar cells. NJ Jeon, JH Noh, WS Yang, YC Kim, S Ryu, J Seo, SI Seok Nature 517 (7535), 476-480	4062
Mastering the game of Go without human knowledge. D Silver, J Schrittwieser, K Simonyan, I Antonoglou, A Huang, A Guez, ... Nature 550 (7676), 354	3668

Why Deep Learning?

- Yoshua Bengio, Geoffrey Hinton, and Yann LeCun won the **Turing Award (2018)**.
 - Note) [Chronological listing of A.M. Turing Award Winners](#)



Yoshua Bengio



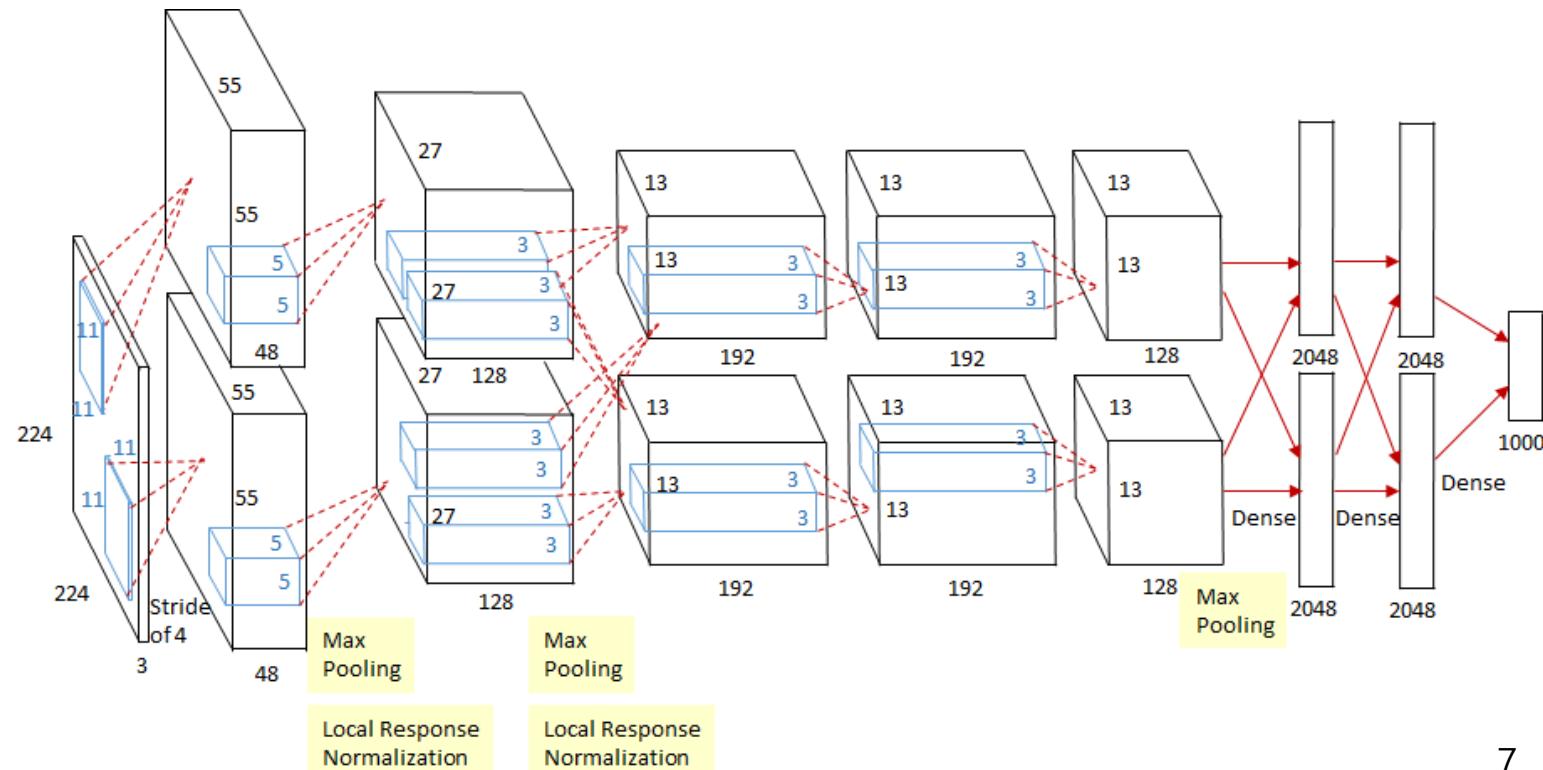
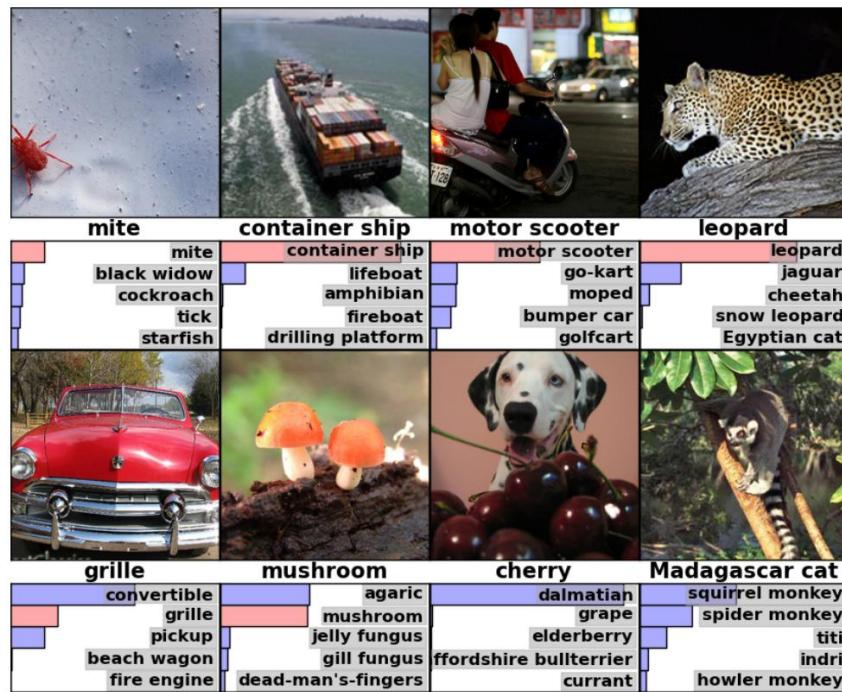
Geoffrey Hinton



Yann LeCun

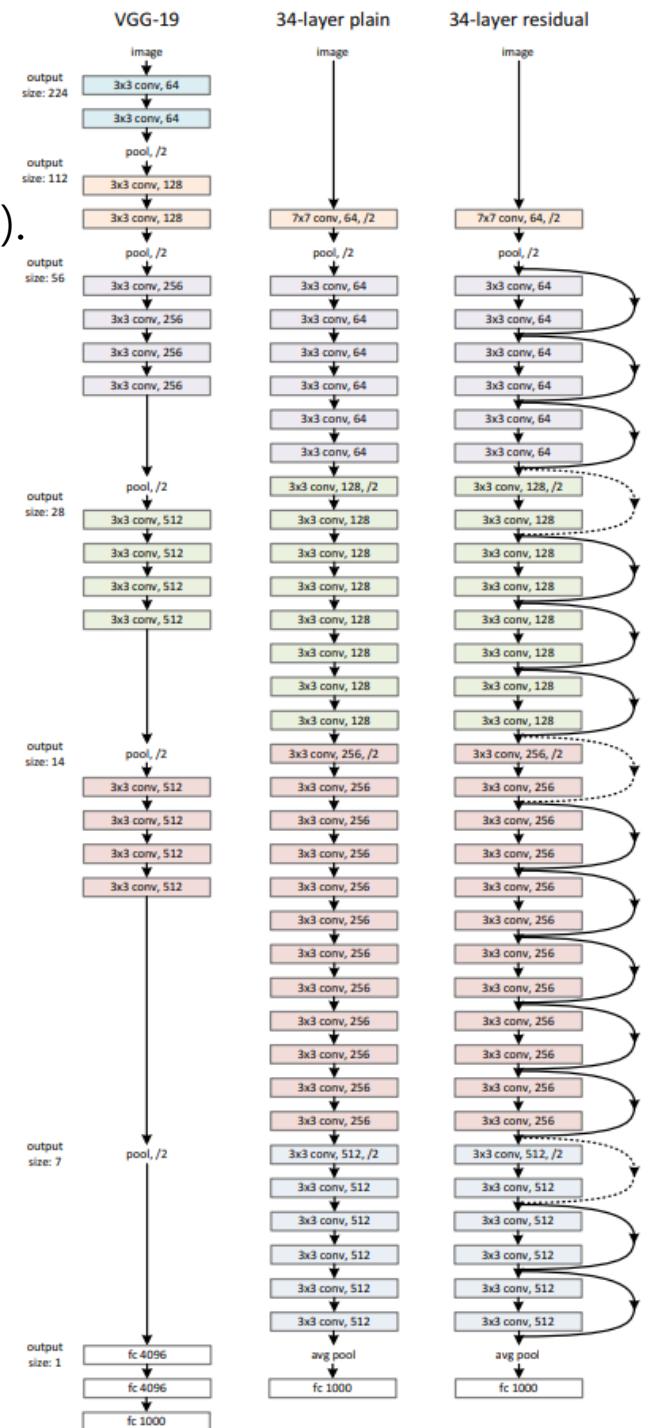
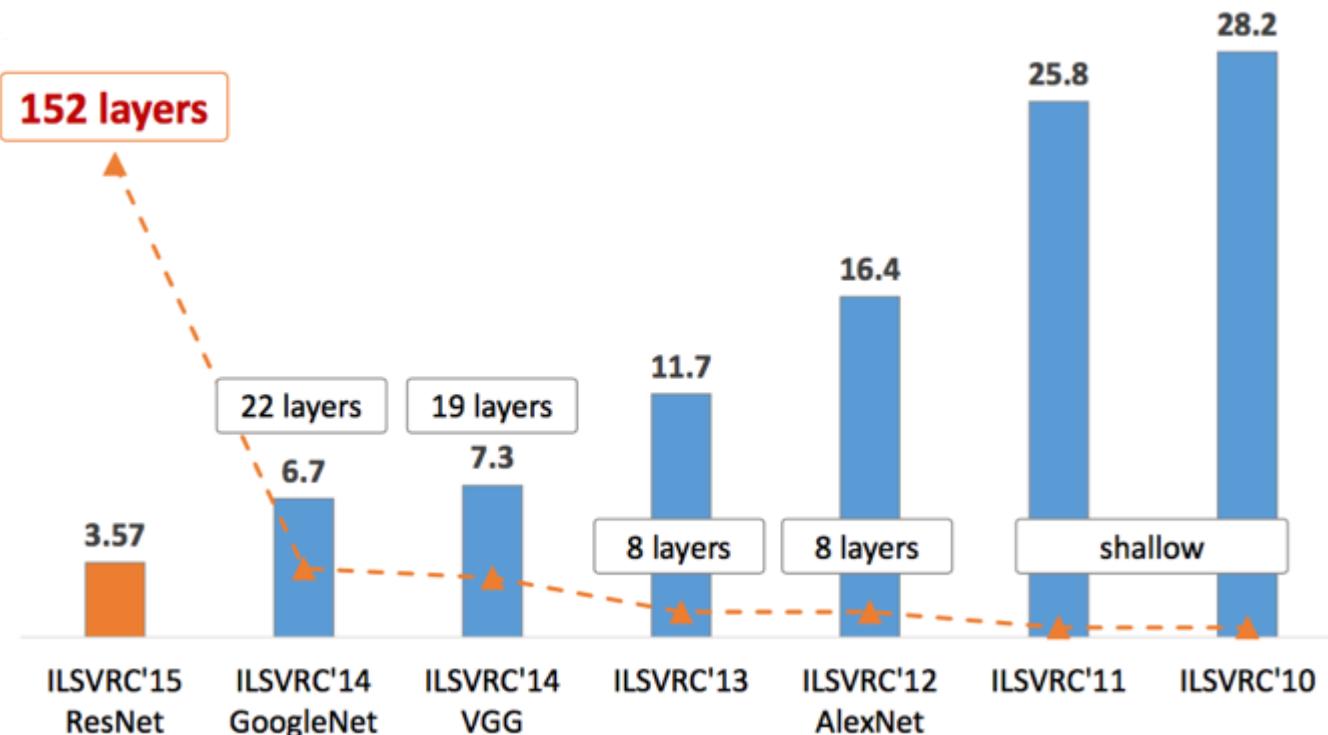
Why Deep Learning?

- **AlexNet** won the [ILSVRC \(ImageNet Large Scale Visual Recognition Competition\)](#) 2012 (top-5 error: **16.4%**).
 - An **8-layer** neural network (approx. 61 million parameters) with 2 x GPUs
 - ReLU (rectified linear unit) is used for the vanishing gradient problem and speed-up.
 - Data augmentation (more data) and dropout (regularization) was used to solve the overfitting problem.
 - Local response normalization (~ batch normalization) is used for stable training and generalization.



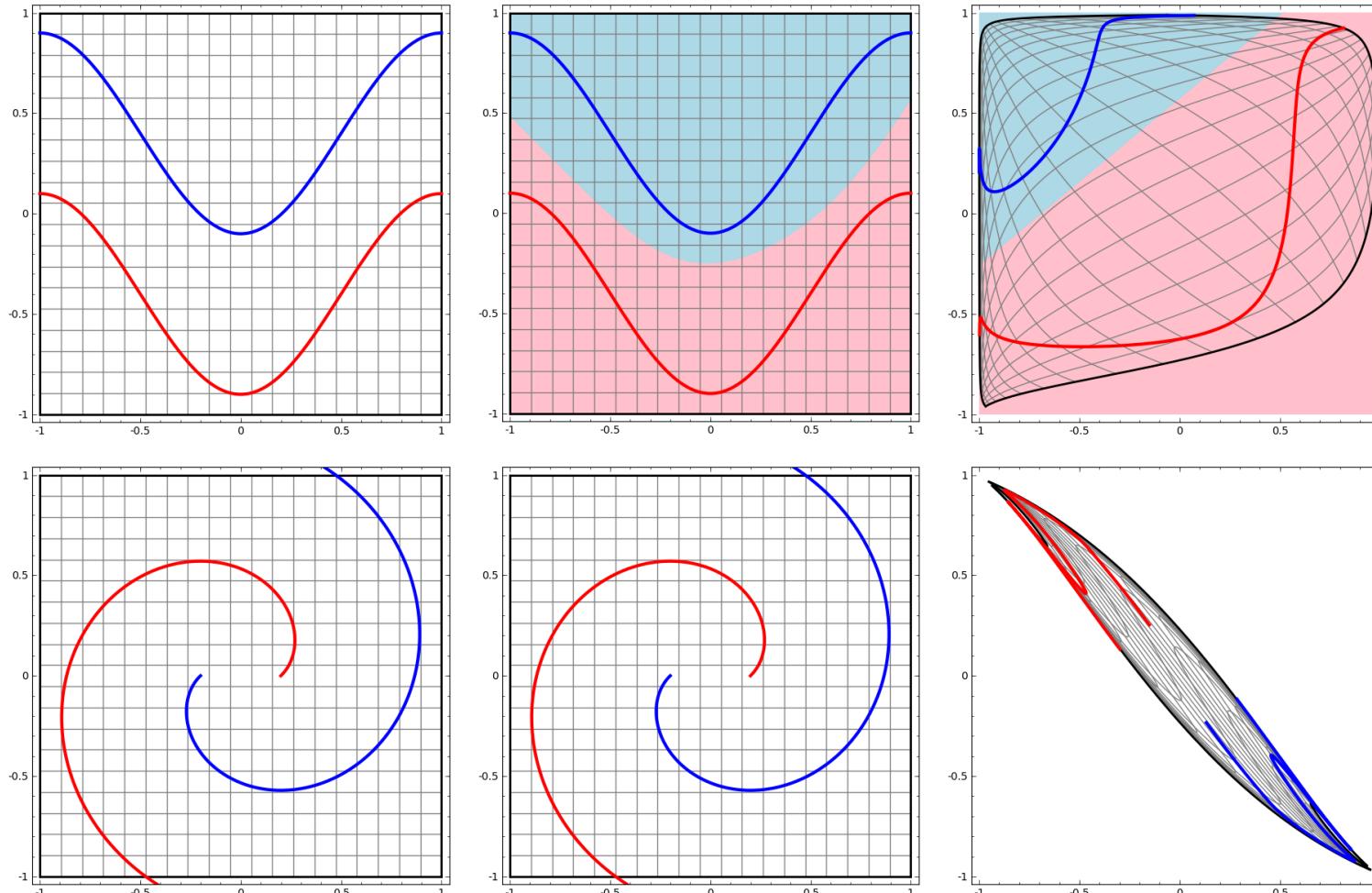
What is Deep Learning?

- *Deep learning* is machine learning with **deep neural networks** (shortly DNN).
 - e.g. **ResNet** ([residual neural network](#))
 - A **152-layer** network with skip connections
 - The winner of ImageNet Challenge 2015 (top-5 error: **3.57%**)



What is Deep Learning?

- *Deep learning* is **representation learning** (a.k.a. feature learning).
 - e.g. Task: Draw a line to separate the red line and blue line.



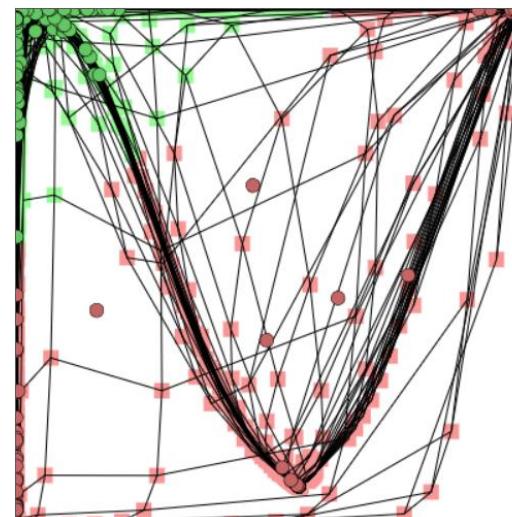
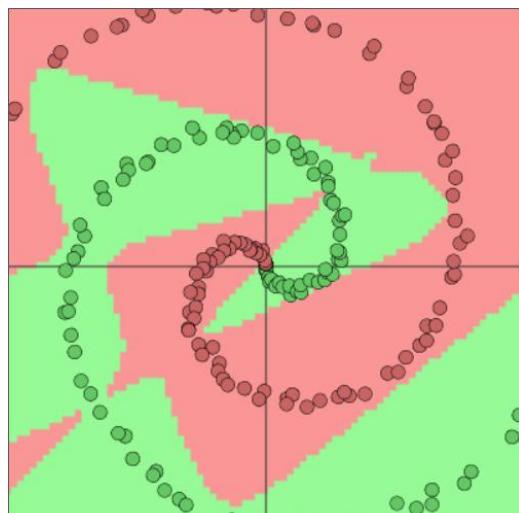
What is Deep Learning?

- *Deep learning* is **representation learning** (a.k.a. feature learning).
 - e.g. Task: Draw a line to separate the **red circles** and **green circles**.
 - Note) Try it at [ConvnetJS](#) or [TensorFlow Playground](#).

```
layer_defs = [];
layer_defs.push({type:'input', out_sx:1, out_sy:1, out_depth:2});
layer_defs.push({type:'fc', num_neurons:6, activation: 'tanh'});
layer_defs.push({type:'fc', num_neurons:2, activation: 'tanh'});
layer_defs.push({type:'softmax', num_classes:2});

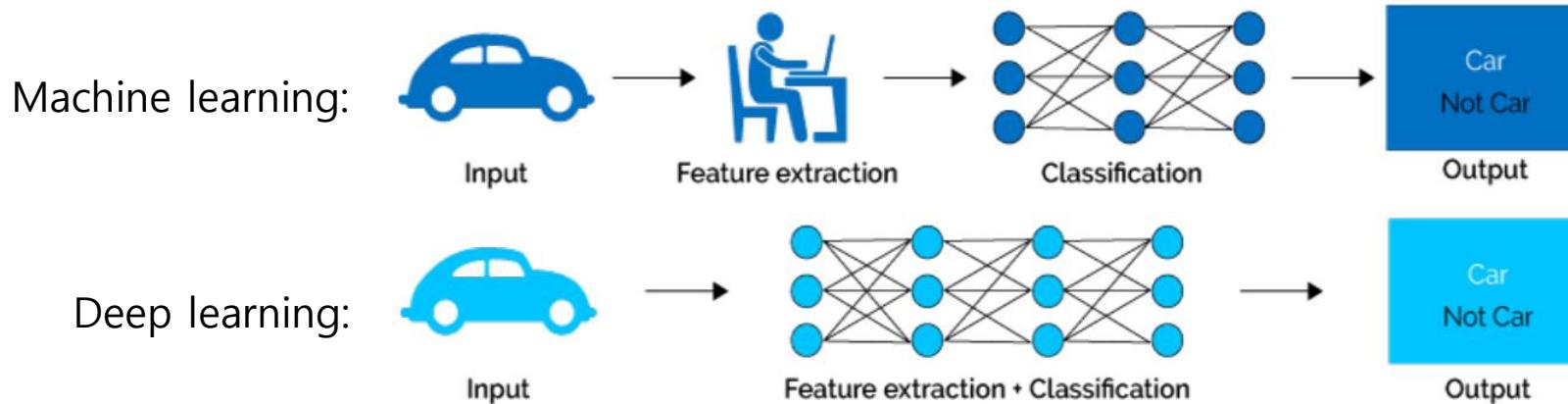
net = new convnetjs.Net();
net.makeLayers(layer_defs);

trainer = new convnetjs.SGDTrainer(net, {learning_rate:0.01, momentum:0.1, batch_size:10, l2_decay:0.001});
```



What is Deep Learning?

- *Deep learning* is **representation learning** (a.k.a. feature learning).



What is Deep Learning?

- *Deep learning* is **scalable**.

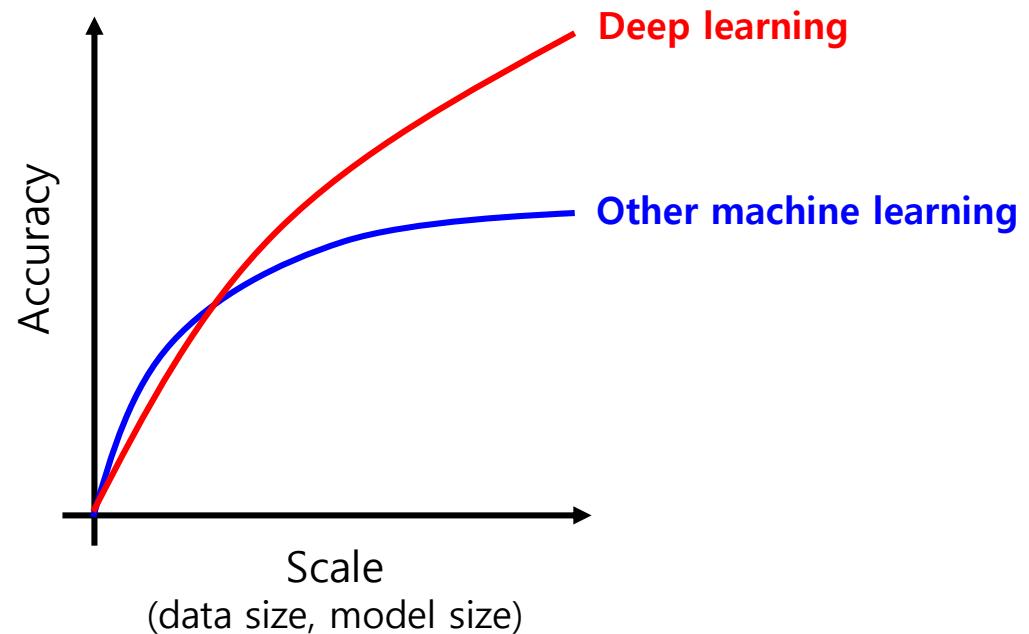


Table of Contents

- **Introduction**

- Why deep learning?
 - What is deep learning?

- **PyTorch**

- Why PyTorch?
 - What is PyTorch?
 - Learning PyTorch with Practice

- **Neural Network (NN)**

- **Convolutional Neural Network (CNN)**

- **Recurrent Neural Network (RNN)**

Why PyTorch?

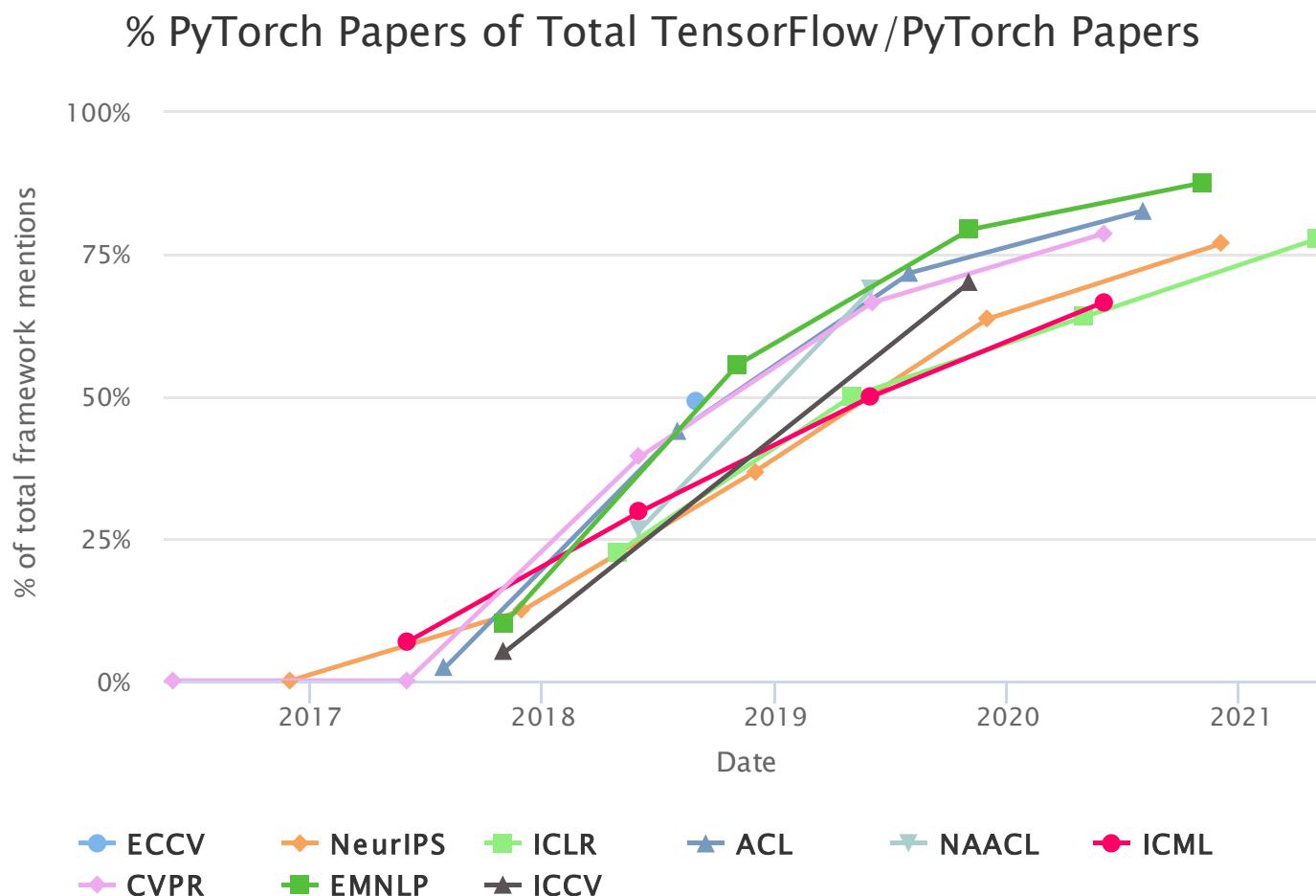
- **10 reasons why PyTorch is the DL framework of the future**

(by Dhiraj K, September 18th, 2019)

1. PyTorch is Pythonic
2. Easy to learn
3. Higher developer productivity
4. Easy debugging
5. Data parallelism
6. Dynamic computational graph support
7. Hybrid front-end
8. Useful libraries
9. ONNX ([Open Neural Network Exchange](#)) support
10. Cloud support

Why PyTorch?

- Popularity (vs. TensorFlow) in academic communities



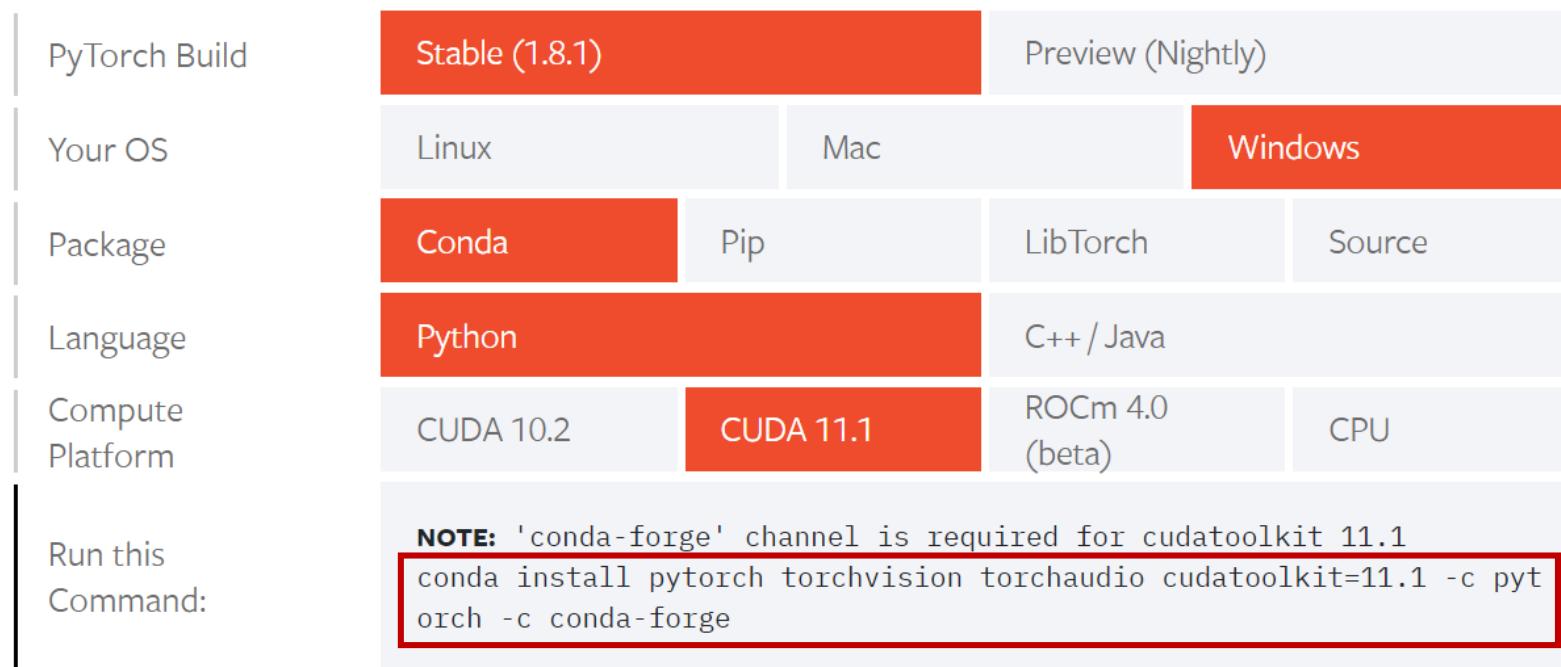
What is PyTorch?



- **PyTorch** (2016) is an open source deep learning library based on Python.
 - It originated from Torch (2002) based on the Lua script language.
 - It is primarily developed by Facebook's AI Research lab (FAIR).
- (From my point of view) PyTorch is a **NumPy extension** with supports of
 - + An n-dimensional array with GPU acceleration and automatic differentiation
 - + Useful modules for neural networks
- My useful lists on deep learning and PyTorch

Practice) PyTorch Installation

- Please follow [PyTorch's installation instruction](#) for your system.
 - Note) If you want GPU acceleration, please install the matched version of CUDA in advance. Please visit [CUDA Toolkit Archive](#) to download a specific version of CUDA.



- Note) You can use [Google Colab](#) without local installation of PyTorch.

Note) Python Virtual Environment

- A [**Python virtual environment**](#) is an isolated Python runtime environment that contains a particular version of Python and its additional packages.
 - Note) [Virtual machine](#): A computer entirely implemented by software
- Why virtual environments?
 - We can run Python applications with different versions of packages (e.g. PyTorch 0.9 and PyTorch 1.8) together.
- Tool #1) [venv](#) in the Python Standard Library
- Tool #2) [conda](#) in Anaconda
 - *Conda* is an open source package/dependency/environment management system for programming languages such as Python, R, Java, JavaScript, C/C++, and more.
 - Usage

<code>conda create --name venv_name python=3.6</code>	Create a virtual environment
<code>conda create --name venv_dst --clone venv_src</code>	Copy a virtual environment
<code>conda activate venv_name</code>	Activate the virtual environment
<code>conda deactivate</code>	Deactivate the virtual environment
<code>conda env remove --name venv_name</code>	Remove the virtual environment
<code>conda env list</code>	List all virtual environments
<code>conda list</code>	List all installed packages

Learning PyTorch with Practice

(From my point of view) PyTorch is a **NumPy extension** with supports of

- + An n-dimensional array with GPU acceleration and automatic differentiation
so-called a ***tensor*** (Note: A matrix is a second-order tensor.)
- + Useful modules for neural networks

- Practice with tensors
 - Creating a tensor
 - Reshaping a tensor
 - Line fitting from two points
 - CPU vs. GPU-acceleration
 - Automatic differentiation (so called *Autograd*)
- Practice with useful modules for neural networks
 - Gradient descent by hands and `torch.optim`
 - More examples with DNNs, CNNs, and RNNs.

Practice) Creating a Tensor (1/2)

```
import numpy as np
import torch

# 1. Create a tensor from a composite data
x = np.array([[3, 29, 82], [10, 18, 84]])
y = torch.tensor(x)
print(y.ndim, y.dim())          # 2                                     Note) x.ndim
print(y.nelement())             # 6                                     Note) x.size
print(y.shape, y.size())        # torch.Size([2, 3])               Note) x.shape
print(y.dtype)                  # torch.int32                         Note) x.dtype

# 2. Create a tensor using initializers
p = torch.rand(3, 2)            # Try zeros, ones, eyes, empty, arange, linspace,
q = torch.zeros_like(p)         # and their ..._like
print(p.dtype)                  # torch.float32
print(q.shape)                  # torch.Size([3, 2])

# 3. Interpret as a tensor (generating only a view)
z = torch.as_tensor(x)          # Or torch.from_numpy(x)  Note) np.asarray()
x[-1,-1] = 86
print(z[-1])                   # tensor([10, 18, 86], dtype=torch.int32)
```

Practice) Creating a Tensor (2/2)

```
# 4. Access elements
print(y[:,1])          # tensor([29, 18])
print(y[0,0])          # tensor(3)
print(y[0,0].item())    # 3
                                         Note) x[0,0] == 3

# 5. CUDA tensors
if torch.cuda.is_available():
    print(y.device)      # 'cpu'
    y_cuda = y.cuda()    # Or y.to('cuda')
    print(y_cuda.device) # 'cuda:0'
    y_cpu = y_cuda.cpu() # Or y.cuda.to('cpu')
    print(y_cpu.device)  # 'cpu'

x_cpu = y_cpu.numpy()  # Or np.array(y_cpu)
x_cuda = y_cuda.numpy() # Error!
```

Practice) Reshaping a Tensor (1/2)

```
import numpy as np
import torch

x = np.array([[[29, 3], [18, 10]], [[27, 10], [12, 5]]])
y = torch.tensor(x)
print(y.ndim)          # 3
print(y.shape)         # torch.Size([2, 2, 2])

p = y.view(-1)          # tensor([29, 3, 18, 10, 27, 10, 12, 5])
print(p.shape, p)       # torch.Size([8])
q = y.view(1, -1)        # tensor([[29, 3, 18, 10, 27, 10, 12, 5]])
print(q.shape, q)        # torch.Size([1, 8])
r = y.view(2, -1)        # tensor([[29, 3, 18, 10], [27, 10, 12, 5]])
print(r.shape, r)        # torch.Size([2, 4])
# Of course, 'reshape' is also supported and the same as 'view'.
s = y.reshape(2, -1, 1) # tensor([[29], [3], [18], [10]], [[27], [10], [12], [5]])
print(s.shape, s)        # torch.Size([2, 4, 1])

ss = s.squeeze(2)        # Note) s.squeeze(0) and s.squeeze(1) have no effect.
print(ss)                # tensor([29, 3, 18, 10], [27, 10, 12, 5])
print(ss.shape)           # torch.Size([2, 4])
u0 = ss.unsqueeze(0)      # tensor([[29, 3, 18, 10], [27, 10, 12, 5]])
print(u0.shape, u0)       # torch.Size([1, 2, 4])
u1 = ss.unsqueeze(1)      # tensor([[29, 3, 18, 10]], [[27, 10, 12, 5]])
print(u1.shape, u1)       # torch.Size([2, 1, 4])
u2 = ss.unsqueeze(2)      # tensor([[29], [3], [18], [10]], [[27], [10], [12], [5]])
print(u2.shape, u2)       # torch.Size([2, 4, 1])
```

29	3
[0,0,0]	[0,0,1]
18	10
[0,1,0]	[0,1,1]
27	10
[1,0,0]	[1,0,1]
12	5
[1,1,0]	[1,1,1]

Practice) Reshaping a Tensor (2/2)

```
# Switch indices each other, (i, j) to (j, i)
t_021 = y.transpose(1, 2) # tensor([[29, 18],
print(t_021, t_021.is_contiguous()) # [ 3, 10], ... ]) False
c_021 = t_021.contiguous() # tensor([[29, 18],
print(c_021, c_021.is_contiguous()) # [ 3, 10], ... ]) True
t_102 = y.transpose(0, 1) # tensor([[29, 3],
print(t_102, t_102.is_contiguous()) # [27, 10]], ... ]) False

# Assign indices
t_201 = y.permute(2, 0, 1) # tensor([[29, 18],
print(t_201, t_201.is_contiguous()) # [27, 12]], ... ]) False

# Note) Reshaping does not copy contents.
y[0,0,0] = 27
print(p) # tensor([27, 3, 18, 10, 27, 10, 12, 5])
print(t_021) # tensor([[27, 18], ... ], ... )
print(c_021) # tensor([[29, 18], ... ], ... )

# Copy a tensor and detach it from its connected computational graph
z = y.clone().detach() # Note) x.clone()
y[0,0,0] = 1
print(y) # tensor([[ [ 1, 3], ... ], ... ])
print(z) # tensor([[ [27, 3], ... ], ... ])
```

29	3
[0,0,0]	[0,0,1]
18	10
[0,1,0]	[0,1,1]
27	10
[1,0,0]	[1,0,1]
12	5
[1,1,0]	[1,1,1]
t(0,1)	
29	3
[0,0,0]	[0,0,1]
18	10
[1,0,0]	[1,0,1]
27	10
[0,1,0]	[0,1,1]
12	5
[1,1,0]	[1,1,1]
relocate	
29	3
[0,0,0]	[0,0,1]
27	10
[0,1,0]	[0,1,1]
18	10
[1,0,0]	[1,0,1]
12	5
[1,1,0]	[1,1,1]

Practice) Line Fitting from Two Points

- Find a line which passes through two points, (1, 4) and (4, 2)
 - The tensor class, `torch.Tensor`, includes member functions of tensor operation and manipulation, and linear algebra (in contrast to `numpy.array`).
 - Note) [PyTorch APIs for `torch.Tensor`](#)
 - Please remember that a function with ending with an underscore(`_`) means an in-place function.

```
import torch

A = torch.tensor([[1., 1.], [4., 1.]])
b = torch.tensor([[4.], [2.]])
A_inv = A.inverse()          # Note) np.linalg.inv(A)
print(A_inv.mm(b))          # Note) np.matmul(A_inv, b)
```

- Note) Line fitting with NumPy

```
import numpy as np

A = np.array([[1., 1.], [4., 1.]])
b = np.array([[4.], [2.]])
A_inv = np.linalg.inv(A)
print(np.matmul(A_inv, b)) # [[-0.66666667], [ 4.66666667]]
```

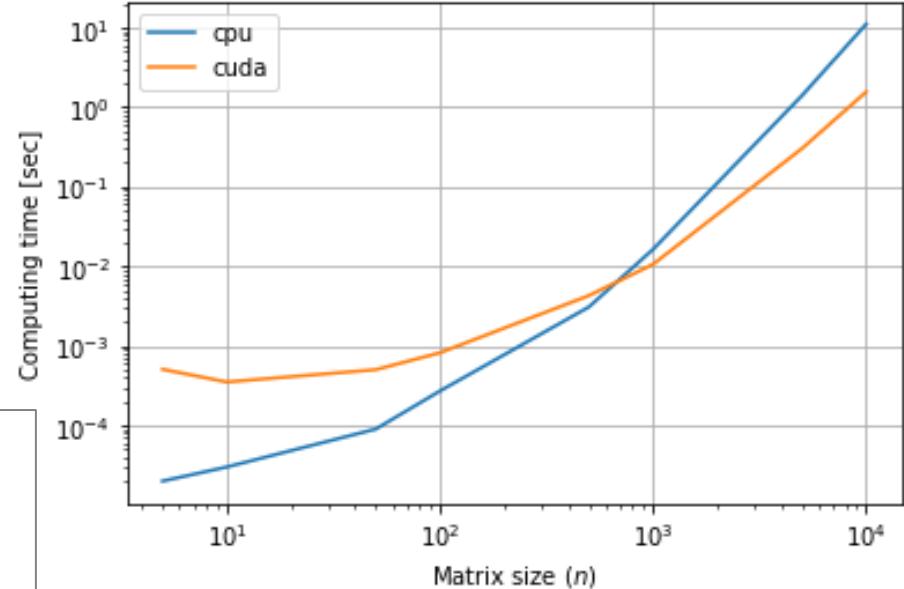
Practice) CPU vs. GPU-acceleration

- Computing time on my laptop
 - cpu : **1.4** [sec] @ Intel i7-7700HQ 2.8GHz
 - cuda: **0.3** [sec] @ NVIDIA GTX 1060

```
import torch
import time

dev_name = 'cuda' if torch.cuda.is_available() else 'cpu' # Try 'cpu'
n = 5000

A = torch.rand(n, n, device=dev_name)
B = torch.rand(n, n, device=dev_name)
start = time.time()
C = A.inverse() * B
elapse = time.time() - start
print(f'Computing time by {dev_name}: {elapse:.3f} [sec]')
```



Practice) Automatic Differentiation

- A derivative value, `torch.Tensor.grad`, is available after setting `requires_grad=True` and its forward calculation and backward propagation (a.k.a. backpropagation).

```
import torch

x = torch.tensor([2.], requires_grad=True)
y = 0.1*x**3 - 0.8*x**2 - 1.5*x + 5.4
y.backward()
print(x.grad) # Derivative: tensor([-3.5000])
```

- Note) Symbolic differentiation with SymPy

```
import sympy as sp

x, y = sp.symbols('x y')
y = 0.1*x**3 - 0.8*x**2 - 1.5*x + 5.4
yd = sp.diff(y, x)
print(yd)                      # 0.3*x**2 - 1.6*x - 1.5
print(float(yd.subs({x: 2}))) # -3.5
```

Practice) Automatic Differentiation – More Analysis

- Given) $y(x) = x^3$, $z(y) = \log y$
- Q) A derivative value $\frac{\partial z}{\partial x}$ at $x = 5$?

```
import torch

def get_tensor_info(tensor):
    info = []
    for name in ['requires_grad', 'is_leaf', 'retains_grad', 'grad']:
        info.append(f'{name}({getattr(tensor, name, None)})')
    info.append(f'tensor({str(tensor)})')
    return ' '.join(info)

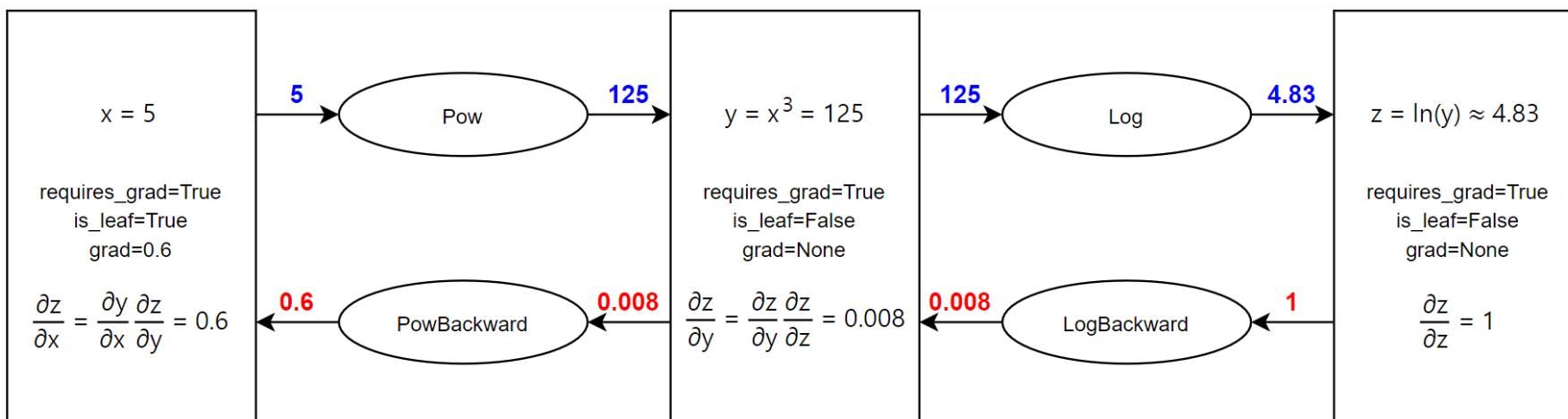
x = torch.tensor(5., requires_grad=True)
y = x ** 3
z = torch.log(y)
print("### Before 'z.backward()'")
print('* x:', get_tensor_info(x))
print('* y:', get_tensor_info(y))
print('* z:', get_tensor_info(z))

y,retain_grad()
z,retain_grad()
z.backward()
print("### After 'z.backward()'")
print('* x:', get_tensor_info(x))
print('* y:', get_tensor_info(y))
print('* z:', get_tensor_info(z))
```

Practice) Automatic Differentiation – More Analysis

- Given) $y(x) = x^3$, $z(y) = \log y$
- Q) A derivative value $\frac{\partial z}{\partial x}$ at $x = 5$?

```
### Before 'z.backward()'  
* x: requires_grad(True) is_leaf(True) retains_grad(None) grad(None) tensor(tensor(5., requires_grad=True))  
* y: requires_grad(True) is_leaf(False) retains_grad(None) grad(None) tensor(tensor(125., grad_fn=<PowBackward0>))  
* z: requires_grad(True) is_leaf(False) retains_grad(None) grad(None) tensor(tensor(4.83, grad_fn=<LogBackward>))  
  
### After 'z.backward()'  
* x: requires_grad(True) is_leaf(True) retains_grad(None) grad(0.6) tensor(tensor(5., requires_grad=True))  
* y: requires_grad(True) is_leaf(False) retains_grad(True) grad(0.008) tensor(tensor(125., grad_fn=<PowBackward0>))  
* z: requires_grad(True) is_leaf(False) retains_grad(True) grad(1.) tensor(tensor(4.83, grad_fn=<LogBackward>))
```



Practice) Gradient Descent by Hands

```
f = lambda x: 0.1*x**3 - 0.8*x**2 - 1.5*x + 5.4
viz_range = torch.FloatTensor([-6, 12])
learn_rate = 0.1
max_iter = 100
min_tol = 1e-6
x_init = 12.

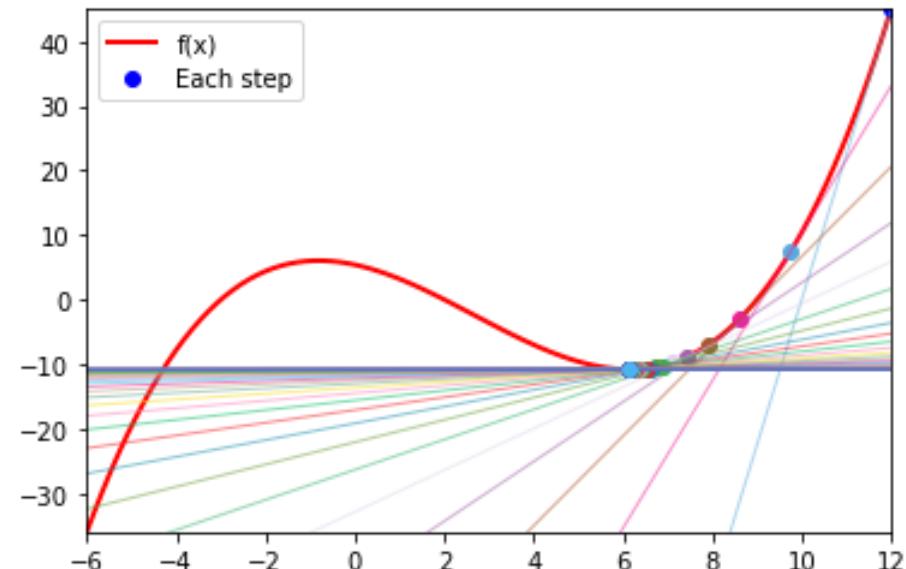
# Prepare visualization
xs = torch.linspace(*viz_range, 100)
plt.plot(xs, f(xs), 'r-', label='f(x)', linewidth=2)
plt.plot(x_init, f(x_init), 'b.', label='Each step', markersize=12)
plt.axis((*viz_range, *f(viz_range)))
plt.legend()

x = torch.tensor(x_init, requires_grad=True)
for i in range(max_iter):
    # Derive gradient with Autograd
    if x.grad != None:
        x.grad.zero_()          # Reset gradient tracking
    y = f(x)                  # Calculate the function (forward)
    y.backward()               # Calculate the gradient (backward)

    # Run the gradient descent
    xp = x.clone().detach()   # Note) xp = x
    with torch.no_grad():
        x -= learn_rate*x.grad # Note) x = x - learn_rate*fd(x) is an original code.
                                #           x = x - learn_rate*x.grad() does not work!

# Update visualization for each iteration
print(f'Iter: {i}, x = {xp:.3f} to {x:.3f}, f(x) = {f(xp):.3f} to {f(x):.3f} (f'(x) = {x.grad:.3f})')
lcolor = torch.rand(3).tolist()
approx = x.grad*(xs-xp) + f(xp)
plt.plot(xs, approx, ' ', linewidth=1, color=lcolor, alpha=0.5)
xc = x.clone().detach() # Copy 'x' for plotting
plt.plot(xc, f(xc), '.', color=lcolor, markersize=12)

# Check the terminal condition
if abs(x - xp) < min_tol:
    break;
plt.show()
```



Practice) Gradient Descent by torch.optim

```
f = lambda x: 0.1*x**3 - 0.8*x**2 - 1.5*x + 5.4
viz_range = torch.FloatTensor([-6, 12])
learn_rate = 0.1
max_iter = 100
min_tol = 1e-6
x_init = 12.

# Prepare visualization
xs = torch.linspace(*viz_range, 100)
plt.plot(xs, f(xs), 'r-', label='f(x)', linewidth=2)
plt.plot(x_init, f(x_init), 'b.', label='Each step', markersize=12)
plt.axis((*viz_range, *f(viz_range)))
plt.legend()

x = torch.tensor(x_init, requires_grad=True)
optimizer = torch.optim.SGD([x], lr=learn_rate)
for i in range(max_iter):
    # Run the gradient descent with the optimizer
    optimizer.zero_grad()          # Reset gradient tracking
    y = f(x)                      # Calculate the function (forward)
    y.backward()                    # Calculate the gradient (backward)
    xp = x.clone().detach()        # Note) xp = x
    optimizer.step()               # Update 'x'

    # Update visualization for each iteration
    print(f'Iter: {i}, x = {xp:.3f} to {x:.3f}, f(x) = {f(xp):.3f} to {f(x):.3f} (f\'(x) = {x.grad:.3f})')
    lcolor = torch.rand(3).tolist()
    approx = x.grad*(xs-xp) + f(xp)
    plt.plot(xs, approx, '-', linewidth=1, color=lcolor, alpha=0.5)
    xc = x.clone().detach() # Copy 'x' for plotting
    plt.plot(xc, f(xc), '.', color=lcolor, markersize=12)

    # Check the terminal condition
    if abs(x - xp) < min_tol:
        break;
plt.show()
```

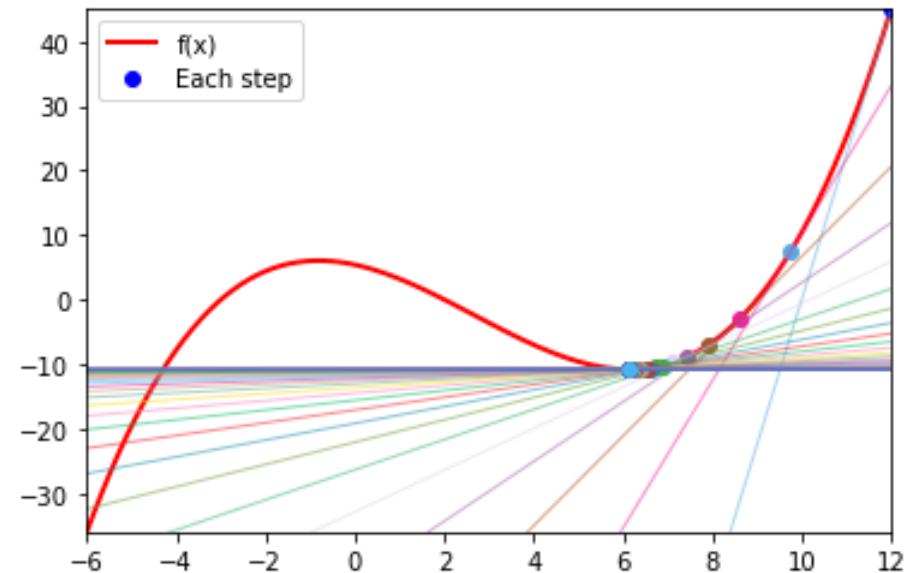
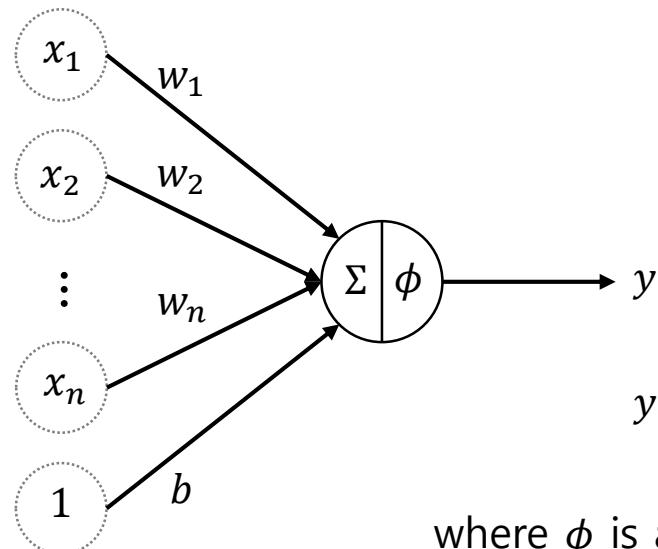
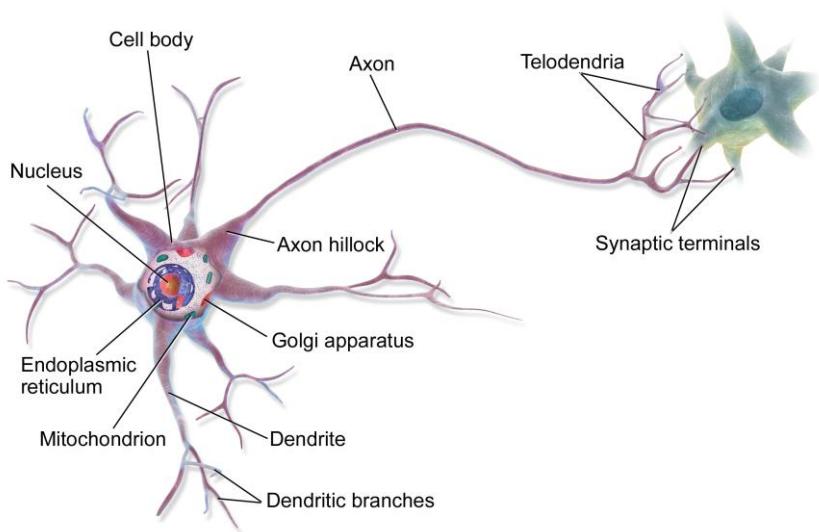


Table of Contents

- **Introduction**
- **PyTorch**
- **Neural Network (NN)**
 - Perceptron
 - Multi-layer perceptron
 - Activation function
 - Backpropagation
 - Vanishing gradient problem
 - Loss function
 - Example) Iris flower classification
- **Convolutional Neural Network (CNN)**
- **Recurrent Neural Network (RNN)**

Neural Network

- An *artificial neural network* (shortly *neural network*, *NN*) is a collection of perceptrons (a.k.a. artificial neurons) and their connection with weights.
 - Inspired by the biological neural networks.
- Neuron vs. Perceptron**



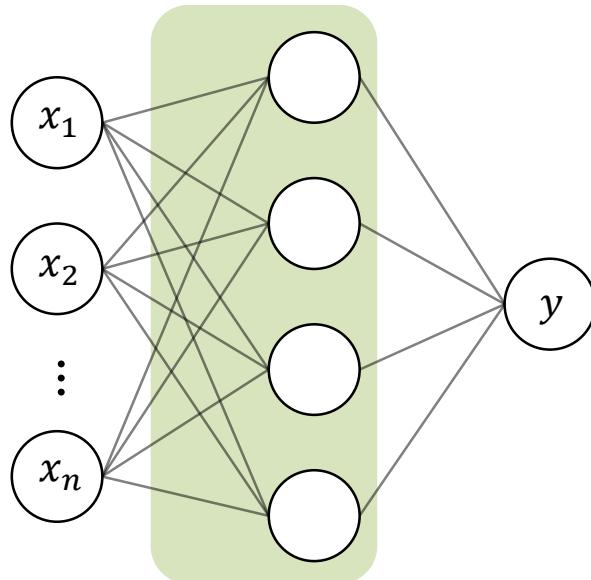
$$y = \phi \left(\sum_{i=1}^n w_i x_i + b \right)$$

where ϕ is an activation function (for nonlinearity)
and b is bias.

Neural Network

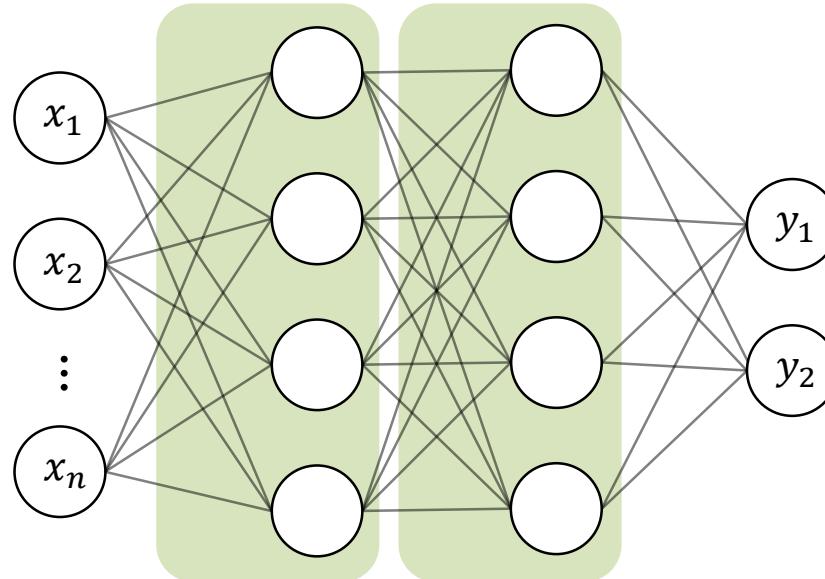
- **Multi-layer perceptrons (MLP)**

- Note) Fully-connected (shortly FC) layer



2-layer NN

(1 x **hidden layer**, 1 x output)

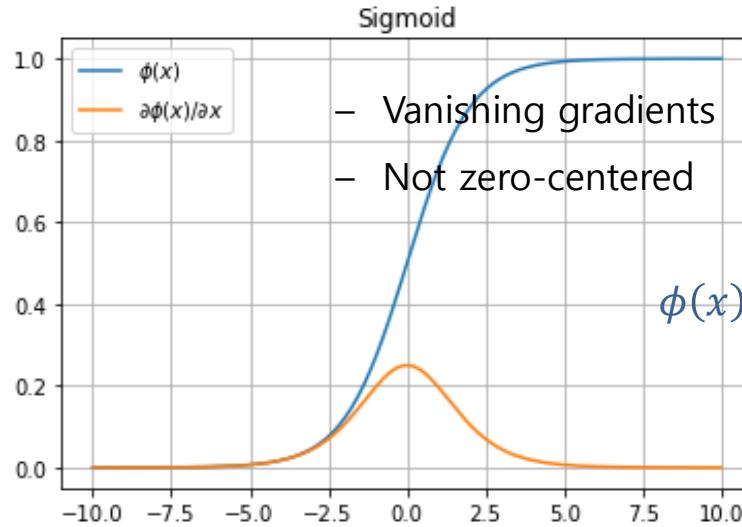


3-layer NN

(2 x **hidden layer**, 2 x output)

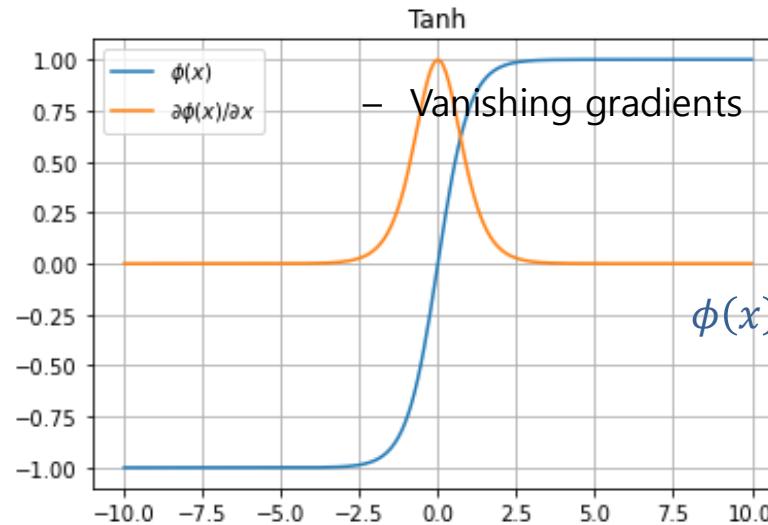
Activation Function

- **Activation function** imposes **nonlinearity** to a neural network.



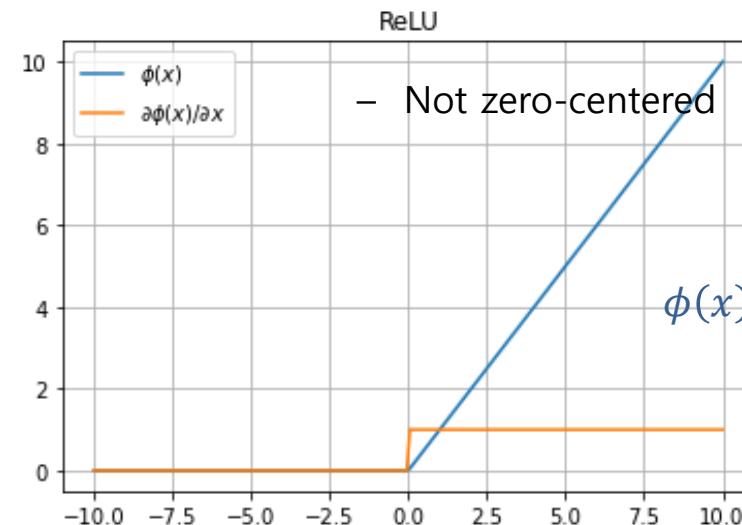
- Vanishing gradients
- Not zero-centered

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



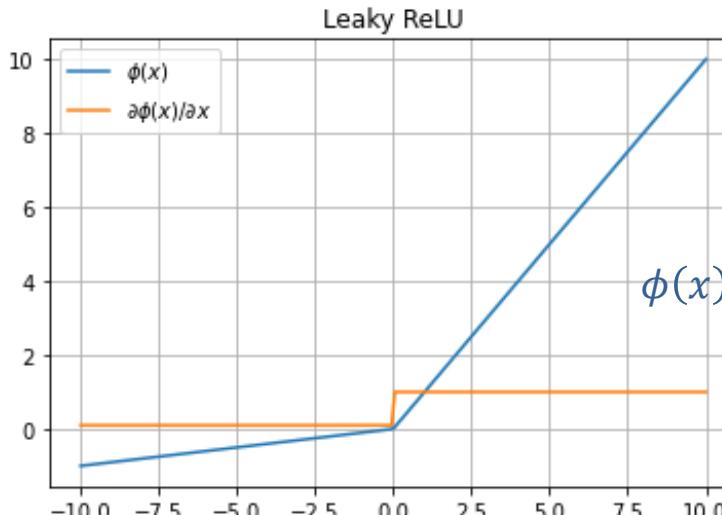
- Vanishing gradients

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



- Not zero-centered

$$\phi(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$



$$\phi(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Practice) Visualizing Activation Functions

- Visualize activation functions and their derivative functions
 - Note) [PyTorch APIs for activation functions](#)

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

activation_funcs = [
    {'name': 'Sigmoid',      'func': nn.Sigmoid()},
    {'name': 'Tanh',         'func': nn.Tanh()},
    {'name': 'ReLU',          'func': nn.ReLU()},
    {'name': 'Leaky ReLU',   'func': nn.LeakyReLU(0.1)},
    {'name': 'ELU',           'func': nn.ELU()},
    # Try more activation functions
]

for act in activation_funcs:
    x = torch.linspace(-10, 10, 200, requires_grad=True)
    y = act['func'](x)
    y.sum().backward()

    plt.title(act['name'])
    x_np, y_np, grad = x.detach().numpy(), y.detach().numpy(), x.grad.numpy()
    plt.plot(x_np, y_np, label='$\phi(x)$')
    plt.plot(x_np, grad, label='$\partial \phi(x) / \partial x$')
    plt.grid()
    plt.legend()
    plt.show()
```

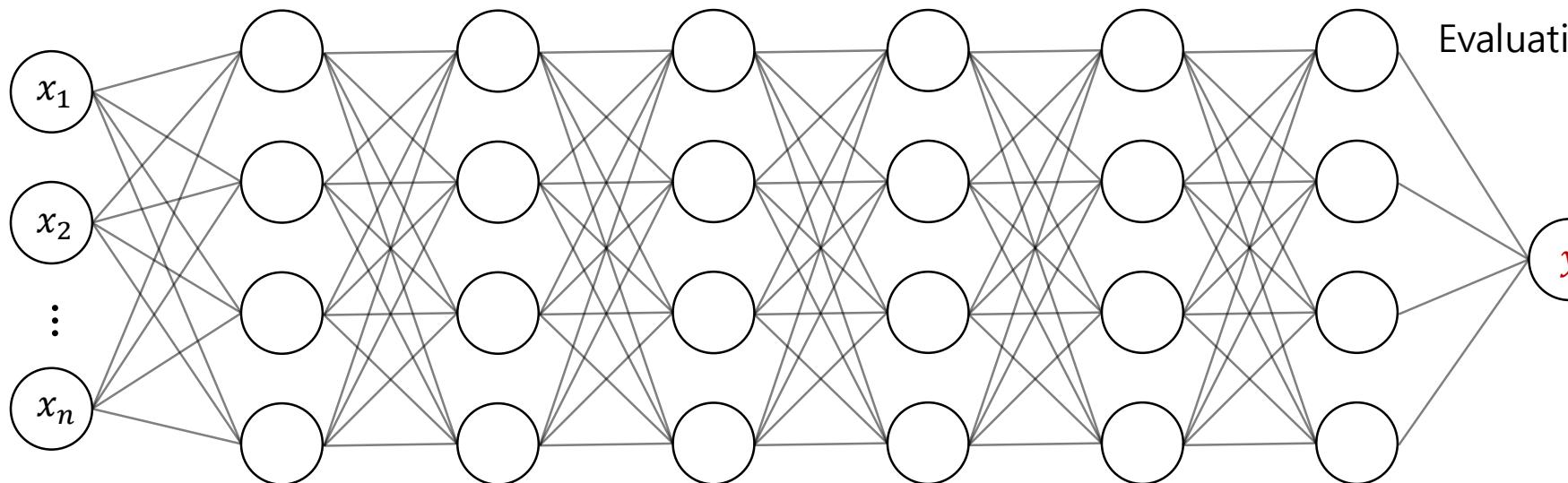
Backpropagation

- **Training a neural network (~ optimization)**

Finding weight variables which minimize a cost function as

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{N} \sum_{d=1}^N l(\mathbf{y}^d, \hat{\mathbf{y}}^d)$$

where l is a loss function, N is the number of data, and $\hat{\mathbf{y}}^d$ is the d -th target value.



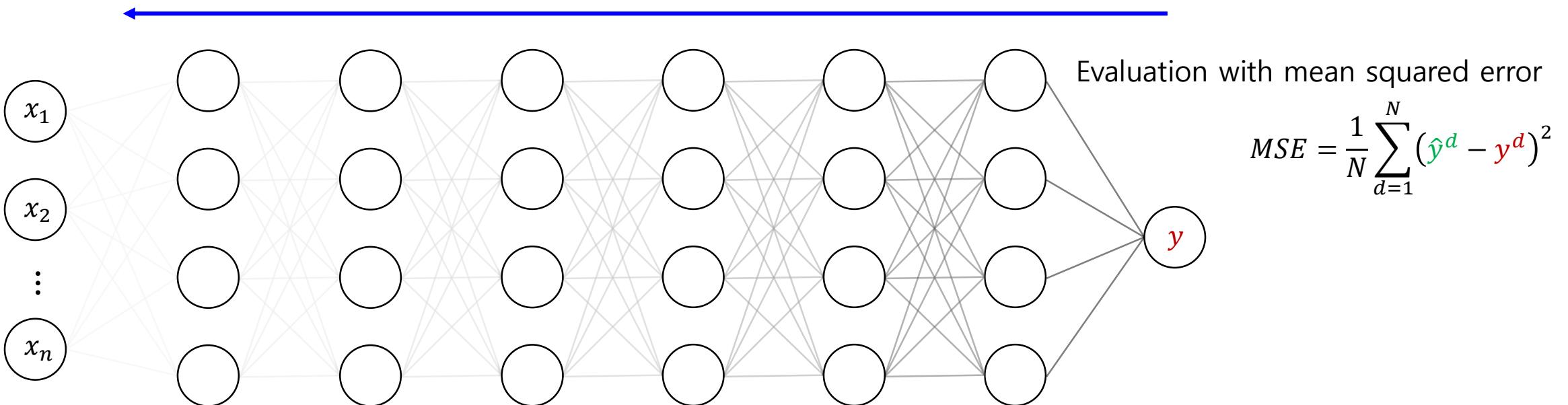
Evaluation with mean squared error

$$MSE = \frac{1}{N} \sum_{d=1}^N (\hat{\mathbf{y}}^d - \mathbf{y}^d)^2$$

$$\text{where } l(\mathbf{y}^d, \hat{\mathbf{y}}^d) = (\hat{\mathbf{y}}^d - \mathbf{y}^d)^2$$

Backpropagation

- **Backpropagation** is an algorithm for training a neural network.
 - It tries to find the optimal **weight variables** of a NN by **gradient descent**
- **Vanishing gradient problem**
 - During backpropagation, gradient values of a deep NN become close to 0.

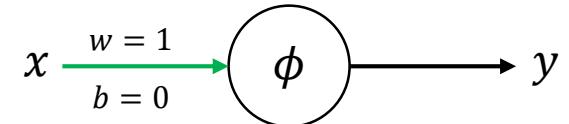


Practice) Observing Vanishing Gradients

- Gradients of a single-node multi-layer NN

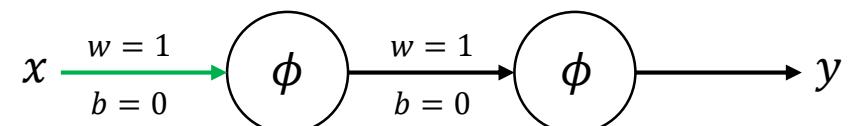
- 1-layer forward: $y = \phi(wx + b) = \phi(x)$ **if $w = 1$ and $b = 0$**

- 1-layer backward: $\frac{\partial y}{\partial x} = \phi'(x)$



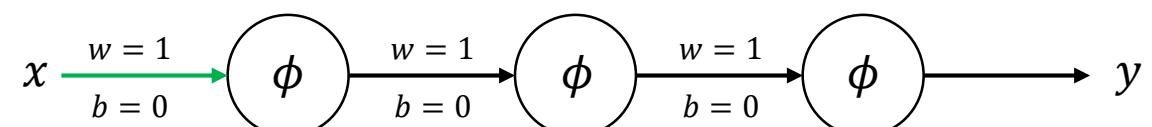
- 2-layer forward: $y = \phi(\phi(x))$

- 2-layer backward: $\frac{\partial y}{\partial x} = \phi'(\phi(x))\phi'(x)$ ∵ **chain rule**



- 3-layer forward: $y = \phi(\phi(\phi(x)))$

- 3-layer backward: $\frac{\partial y}{\partial x} = \phi'(\phi(\phi(x)))\phi'(\phi(x))\phi'(x)$



- ...

Practice) Observing Vanishing Gradients

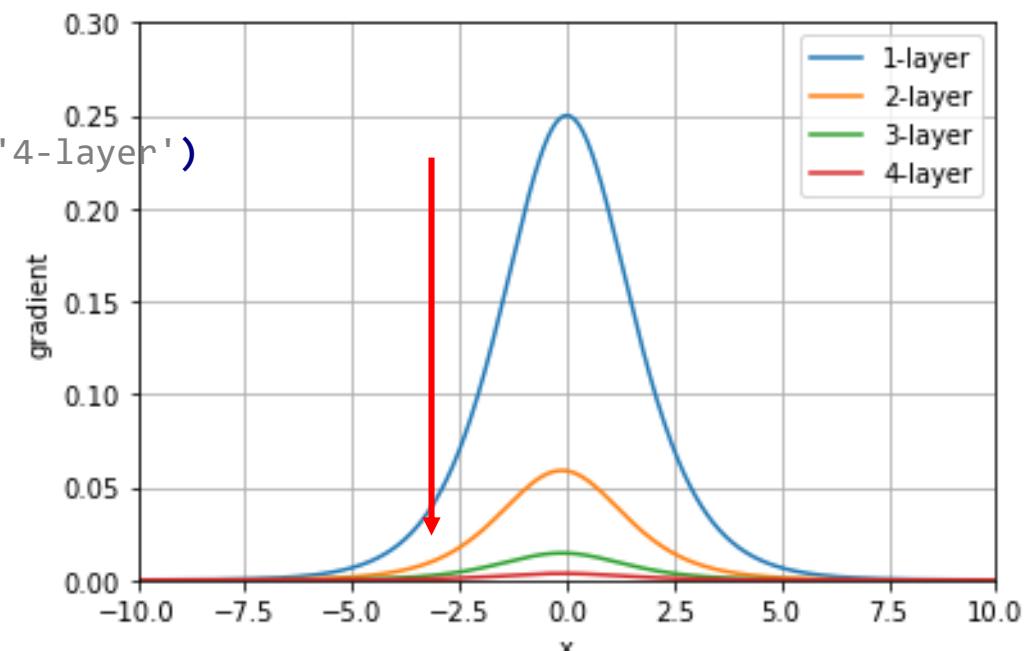
- Gradients of a single-node multi-layer NN with the **sigmoid** function
 - Note) $\phi'(x) = \phi(x)(1 - \phi(x))$

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: 1 / (1 + np.exp(-x))
df = lambda x: f(x) * (1 - f(x))

x = np.linspace(-10, 10, 1000)

plt.plot(x, df(x), label='1-layer')
plt.plot(x, df(f(x))*df(x), label='2-layer')
plt.plot(x, df(f(f(x)))*df(f(x))*df(x), label='3-layer')
plt.plot(x, df(f(f(f(x))))*df(f(f(x)))*df(f(x))*df(x), label='4-layer')
plt.axis((-10, 10, 0, 0.3))
plt.grid()
plt.legend()
plt.show()
```



Practice) Observing Vanishing Gradients

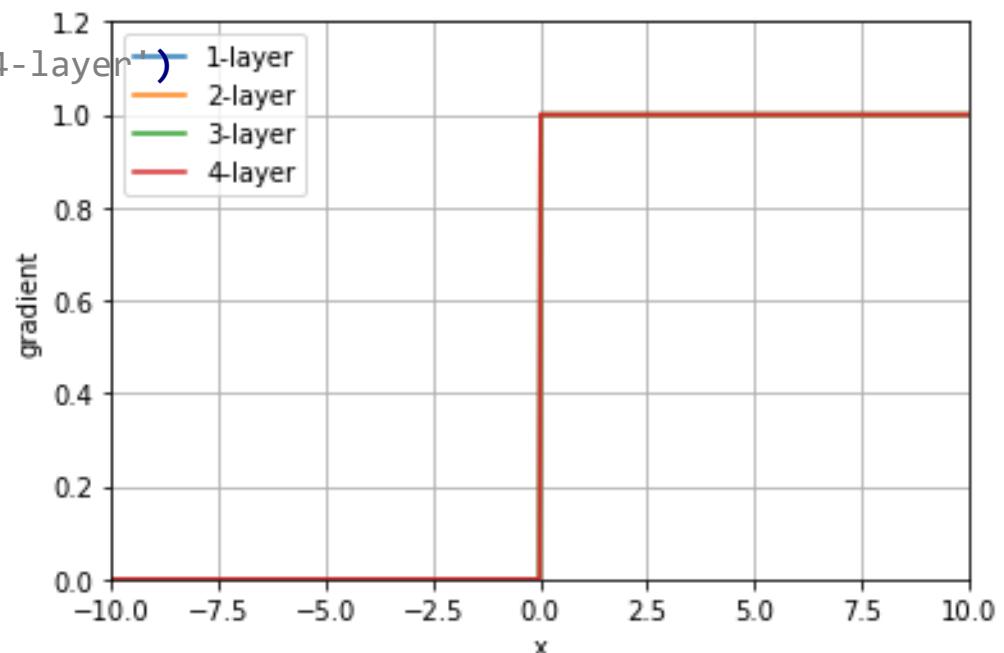
- Gradients of a single-node multi-layer NN with the **ReLU** function

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: x * (x >= 0)
df = lambda x: x >= 0

x = np.linspace(-10, 10, 1000)

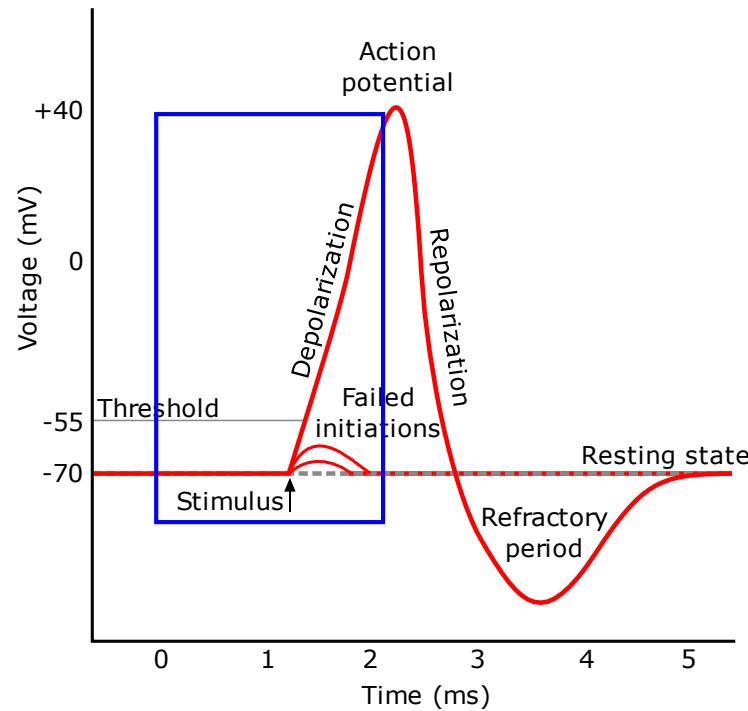
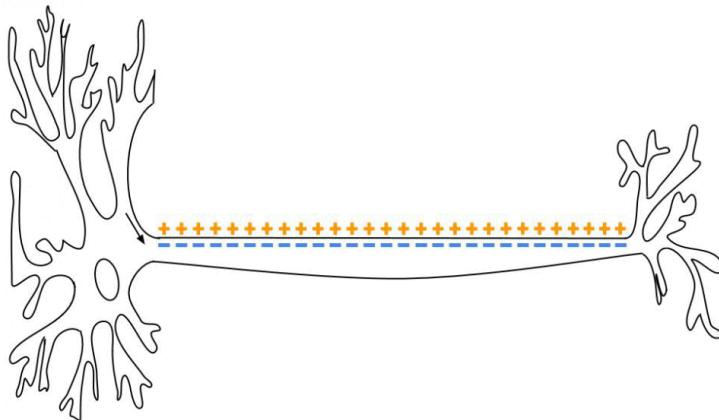
plt.plot(x, df(x), label='1-layer')
plt.plot(x, df(f(x))*df(x), label='2-layer')
plt.plot(x, df(f(f(x)))*df(f(x))*df(x), label='3-layer')
plt.plot(x, df(f(f(f(x))))*df(f(f(x)))*df(f(x))*df(x), label='4-layer')
plt.axis((-10, 10, 0, 0.3))
plt.grid()
plt.legend()
plt.show()
```



Activation Function (Revisited)

- Why ReLU?

- It can mitigate the vanishing gradient problem.
- In addition, there is a biological analogue such as neural action potential.



Loss Function

- **Training a neural network (~ optimization)**

Finding weight variables which minimize a cost function as

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \frac{1}{N} \sum_{d=1}^N l(\mathbf{y}^d, \hat{\mathbf{y}}^d)$$

where l is a loss function, N is the number of data, and $\hat{\mathbf{y}}^d$ is the d -th target value.

- **Loss function**

- A *loss function* quantifies the gap between the **ground truth** and **prediction**.
- e.g. **Mean squared error** (usually for *regression*)

$$l(\mathbf{y}, \hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^2$$

where $\hat{\mathbf{y}}$ is the ground truth, \mathbf{y} is prediction, and N is the number of data.

- e.g. **Binary cross entropy error** (usually for *binary classification*)

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\hat{\mathbf{y}} \log \mathbf{y} - (1 - \hat{\mathbf{y}}) \log(1 - \mathbf{y})$$

In general, $-\sum \hat{\mathbf{y}}_i \log \mathbf{y}_i$ for *multi-class classification*

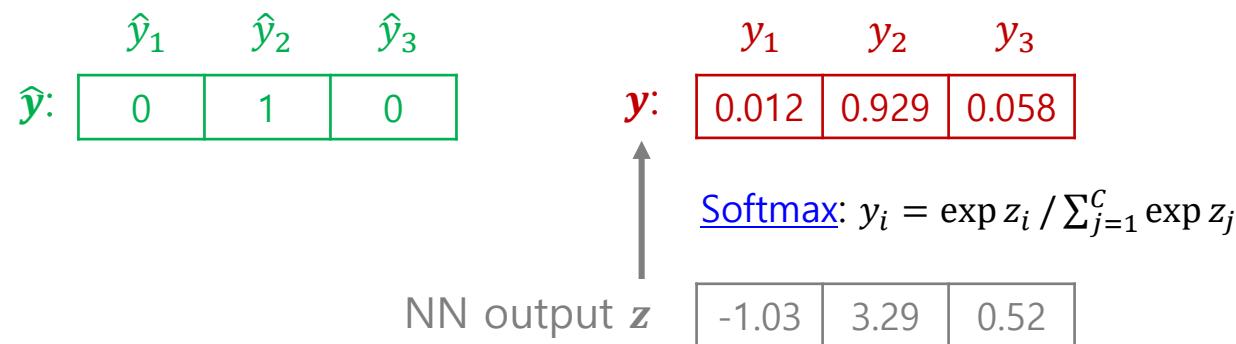
Loss Function

▪ Loss function

- e.g. **Cross entropy error** (usually for *multi-class classification*)

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{i=1}^c \hat{y}_i \log y_i$$

where \hat{y}_i is the one-hot-encoded truth, y_i is the predicted (softmax) confidence



- Note) PyTorch APIs for loss functions

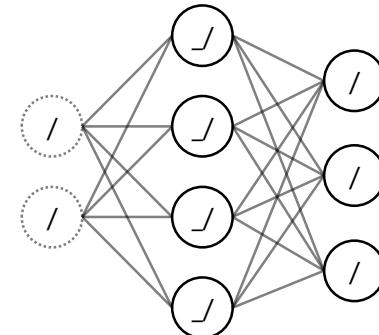
Practice) Iris Flower Classification (1/3)

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sklearn import (datasets, metrics)
from matplotlib.colors import ListedColormap
import time

# 1.1. Load a dataset partially
iris = datasets.load_iris()
iris.data = iris.data[:,0:2]
iris.feature_names = iris.feature_names[0:2]
iris.color = np.array([(1, 0, 0), (0, 1, 0), (0, 0, 1)])

# 1.2. Load the dataset as tensors
dev_name = 'cuda' if torch.cuda.is_available() else 'cpu' # Try 'cpu'
x = torch.tensor(iris.data, device=dev_name).float()
y = torch.tensor(iris.target, device=dev_name).long()

# 2. Define a model
# - Try the different number of hidden layers
# - Try less or more layers with different transfer functions
input_size, output_size = len(iris.feature_names), len(iris.target_names)
model = nn.Sequential(
    nn.Linear(input_size, 4),
    nn.ReLU(),
    nn.Linear(4, output_size),
).to(dev_name)
```

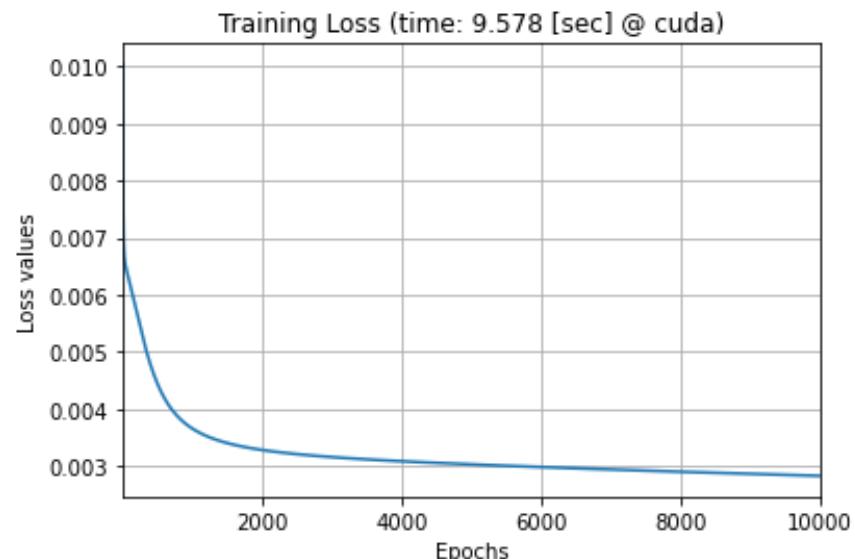
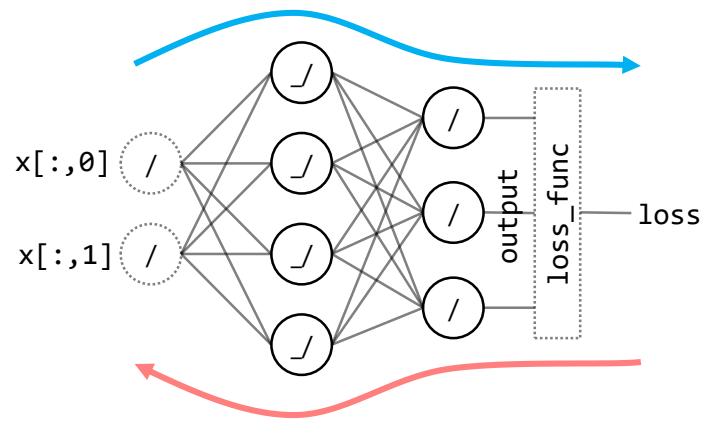


Practice) Iris Flower Classification (2/3)

```
# 3. Train the model
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Try other optimizers
epoch_max = 10000
loss_list = []
start = time.time()
for i in range(epoch_max):
    # Train one iteration
    optimizer.zero_grad()
    output = model(x)
    loss = loss_func(output, y)
    loss.backward()
    optimizer.step()

    # Record the loss
    loss_list.append(loss / len(x))
elapse = time.time() - start

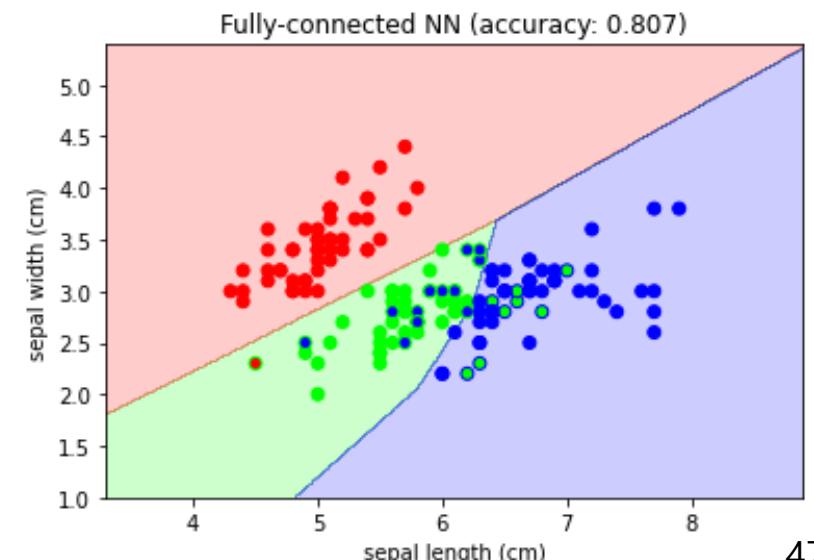
# 4.1. Visualize the training loss curve
plt.title(f'Training Loss (time: {elapse/60:.2f} [min] @ {dev_name})')
plt.plot(range(1, epochs + 1), loss_list)
plt.xlabel('Epochs')
plt.ylabel('Loss values')
plt.xlim((1, epochs))
plt.grid()
plt.show()
```



Practice) Iris Flower Classification (3/3)

```
# 4.2. Visualize training results (decision boundaries)
x_min, x_max = iris.data[:, 0].min() - 1, iris.data[:, 0].max() + 1
y_min, y_max = iris.data[:, 1].min() - 1, iris.data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
xy = np.vstack((xx.flatten(), yy.flatten())).T
xy_tensor = torch.from_numpy(xy).float().to(dev_name)
zz = torch.argmax(model(xy_tensor), dim=1).cpu().detach().numpy()
plt.contourf(xx, yy, zz.reshape(xx.shape), cmap=ListedColormap(iris.color), alpha=0.2)

# 4.3. Visualize data with their classification
predict = torch.argmax(model(x), dim=1).cpu().detach().numpy()
accuracy = metrics.balanced_accuracy_score(iris.target, predict)
plt.title(f'Fully-connected NN (accuracy: {accuracy:.3f})')
plt.scatter(iris.data[:,0], iris.data[:,1], c=iris.color[iris.target], edgecolors=iris.color[predict])
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.show()
```



Practice) Iris Flower Classification – My Style (1/5)

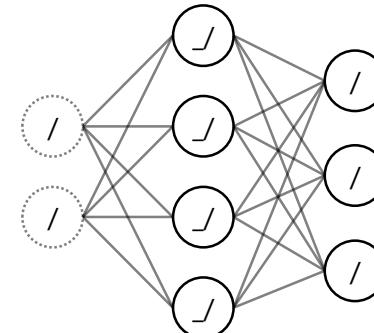
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from matplotlib.colors import ListedColormap
import time

# Define hyperparameters
EPOCH_MAX = 10000
EPOCH_LOG = 1000
OPTIMIZER_PARAM = {'lr': 0.01}
DATA_LOADER_PARAM = { 'batch_size': 50, 'shuffle': True }
USE_CUDA = torch.cuda.is_available()
RANDOM_SEED = 777

# A two-layer NN model
class MyDNN(nn.Module):
    def __init__(self, input_size=2, output_size=3):
        super(MyDNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 4)
        self.fc2 = nn.Linear(4, output_size)

        nn.init.xavier_uniform_(self.fc1.weight)
        nn.init.xavier_uniform_(self.fc2.weight)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```



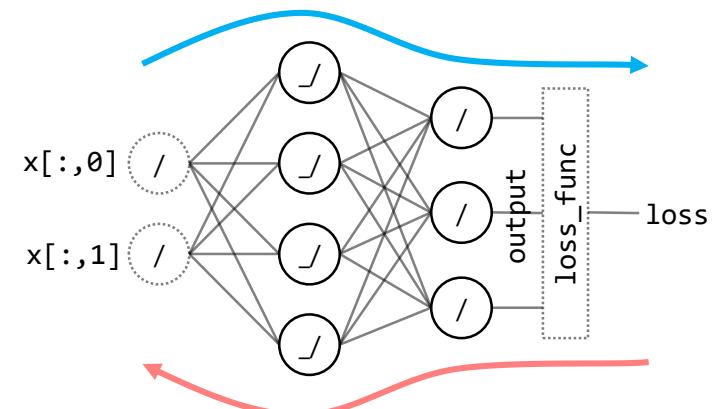
Practice) Iris Flower Classification – My Style (2/5)

```
# Train a model with the given batches
def train(model, batch_data, loss_func, optimizer):
    model.train() # Notify layers (e.g. DropOut, BatchNorm) that it's now training
    train_loss, n_data = 0, 0
    dev = next(model.parameters()).device
    for batch_idx, (x, y) in enumerate(batch_data):
        x, y = x.to(dev), y.to(dev)
        optimizer.zero_grad()
        output = model(x)
        loss = loss_func(output, y)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        n_data += len(y)
    return train_loss / n_data
```

```
# Evaluate a model with the given batches
def evaluate(model, batch_data, loss_func):
    model.eval() # Notify layers (e.g. DropOut, BatchNorm) that it's now testing
    test_loss, n_correct, n_data = 0, 0, 0
    with torch.no_grad():
        dev = next(model.parameters()).device
        for x, y in batch_data:
            x, y = x.to(dev), y.to(dev)
            output = model(x)
            loss = loss_func(output, y)
            y_pred = torch.argmax(output, dim=1)

            test_loss += loss.item()
            n_correct += (y == y_pred).sum().item()
            n_data += len(y)
    return test_loss / n_data, n_correct / n_data
```



Practice) Iris Flower Classification – My Style (3/5)

```
if __name__ == '__main__':
    # 0. Preparation
    torch.manual_seed(RANDOM_SEED)
    if USE_CUDA:
        torch.cuda.manual_seed_all(RANDOM_SEED)
    dev = torch.device('cuda' if USE_CUDA else 'cpu')

    # 1.1. Load the Iris dataset partially
    iris = datasets.load_iris()
    iris.data = iris.data[:,0:2]
    iris.feature_names = iris.feature_names[0:2]
    iris.color = np.array([(1, 0, 0), (0, 1, 0), (0, 0, 1)])

    # 1.2. Wrap the dataset with torch.utils.data.DataLoader
    x = torch.tensor(iris.data, dtype=torch.float32, device=dev)
    y = torch.tensor(iris.target, dtype=torch.long, device=dev)
    data_train = torch.utils.data.TensorDataset(x, y)
    loader_train = torch.utils.data.DataLoader(data_train, **DATA_LOADER_PARAM)
```

Practice) Iris Flower Classification – My Style (4/5)

```
# 2. Instantiate a model, loss function, and optimizer
model = MyDNN().to(dev)
loss_func = F.cross_entropy
optimizer = torch.optim.SGD(model.parameters(), **OPTIMIZER_PARAM)

# 3. Train the model
loss_list = []
start = time.time()
for epoch in range(1, EPOCH_MAX + 1):
    train_loss = train(model, loader_train, loss_func, optimizer)
    valid_loss, valid_accuracy = evaluate(model, loader_train, loss_func)

    loss_list.append([epoch, train_loss, valid_loss, valid_accuracy])
    if epoch % EPOCH_LOG == 0:
        elapse = time.time() - start
        print(f'{epoch:>6} ({elapse:>6.2f} sec), TrLoss={train_loss:.6f}, VaLoss={valid_loss:.6f}, VaAcc={valid_accuracy:.3f}')
elapse = time.time() - start

# 4.1. Visualize the loss curves
# 4.2. Visualize training results (decision boundaries)
# 4.3. Visualize data with their classification
```

Practice) Iris Flower Classification – My Style (5/5)

```
1000 ( 7.71 sec), TrLoss=0.009773, VaLoss=0.009769, VaAcc=0.707
2000 (15.17 sec), TrLoss=0.009300, VaLoss=0.009281, VaAcc=0.747
3000 (22.85 sec), TrLoss=0.009022, VaLoss=0.009007, VaAcc=0.780
4000 (30.41 sec), TrLoss=0.008800, VaLoss=0.008786, VaAcc=0.800
5000 (37.85 sec), TrLoss=0.008606, VaLoss=0.008594, VaAcc=0.813
6000 (45.49 sec), TrLoss=0.008433, VaLoss=0.008420, VaAcc=0.813
7000 (53.06 sec), TrLoss=0.008274, VaLoss=0.008262, VaAcc=0.813
8000 (60.31 sec), TrLoss=0.008138, VaLoss=0.008118, VaAcc=0.807
9000 (67.91 sec), TrLoss=0.008008, VaLoss=0.007990, VaAcc=0.800
10000 (75.14 sec), TrLoss=0.007928, VaLoss=0.007880, VaAcc=0.800
```

Discussion) CPU vs. GPU
Data as raw tensors vs. Data as DataLoader

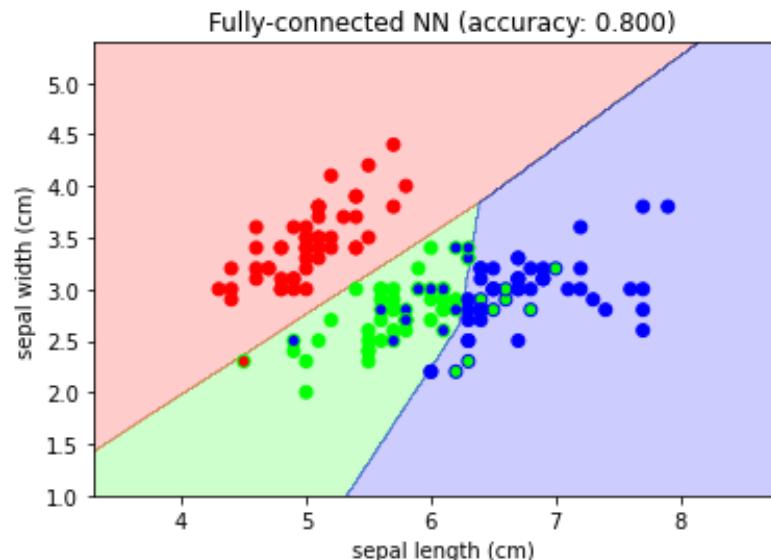
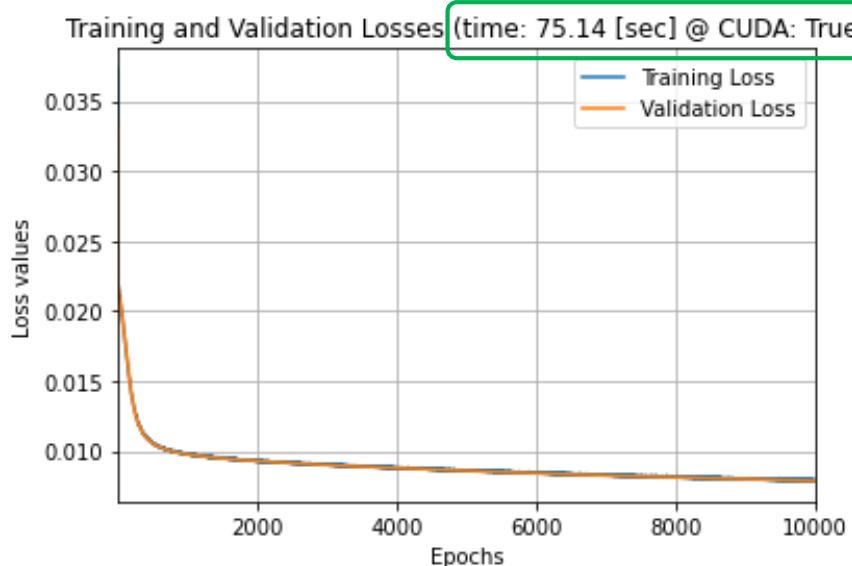


Table of Contents

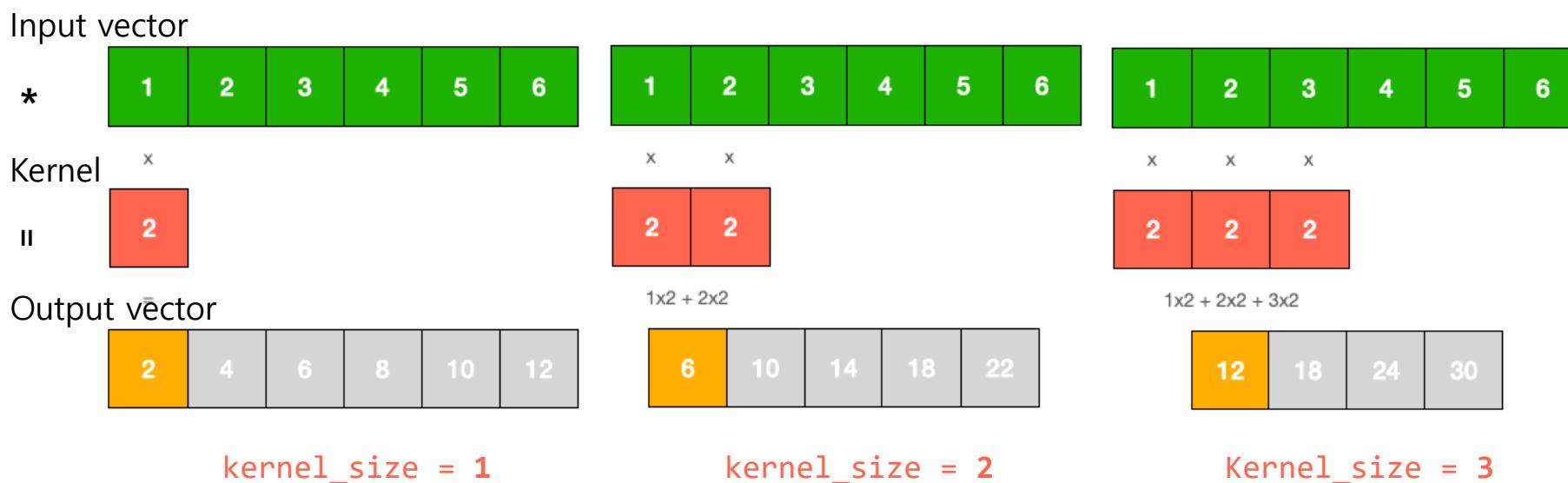
- Introduction
- PyTorch
- Neural Network (NN)
- **Convolutional Neural Network (CNN)**
 - Convolutional layer
 - Pooling layer
 - Dropout
 - Skip connection
 - Example) Digit classification with the MNIST dataset
- **Recurrent Neural Network (RNN)**

Convolution

- **Discrete convolution**

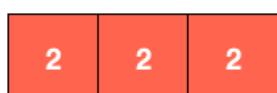
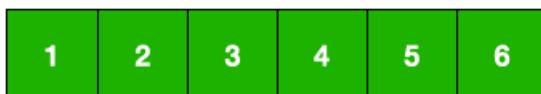
$$(f * g)(k) = \sum_{i=-\infty}^{\infty} f(i)g(k-i)$$

- 1D discrete convolution (e.g. voice)

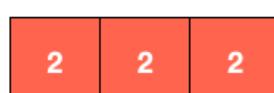


Convolution

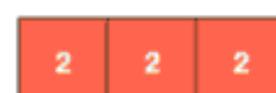
- 1D discrete convolution (continued)



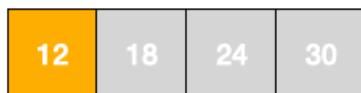
$1 \times 2 + 2 \times 2 + 3 \times 2$



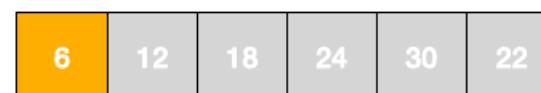
$0 \times 2 + 1 \times 2 + 2 \times 2$



$1 \times 2 + 2 \times 2 + 3 \times 2$



`kernel_size = 3
padding = 0
stride = 1
dilation = 1`



`kernel_size = 3
padding = 1
stride = 1
dilation = 1`



`kernel_size = 3
padding = 0
stride = 3
dilation = 1`



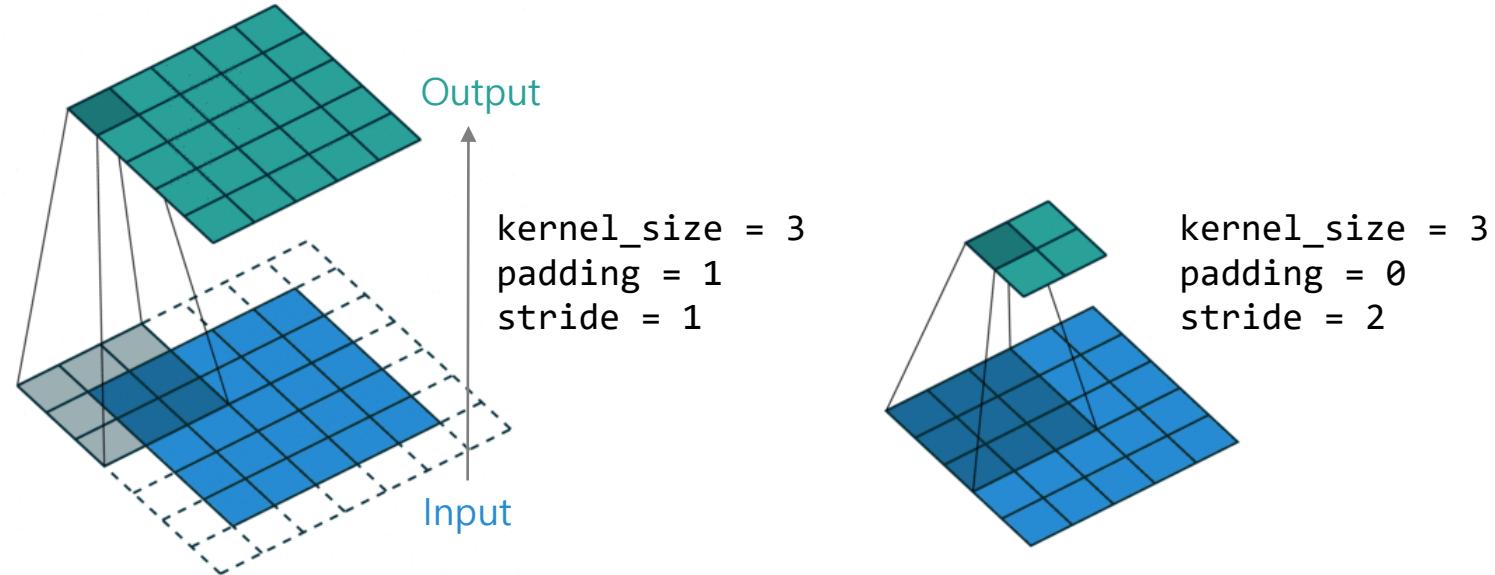
`dilation = 2`

- The size of output vectors

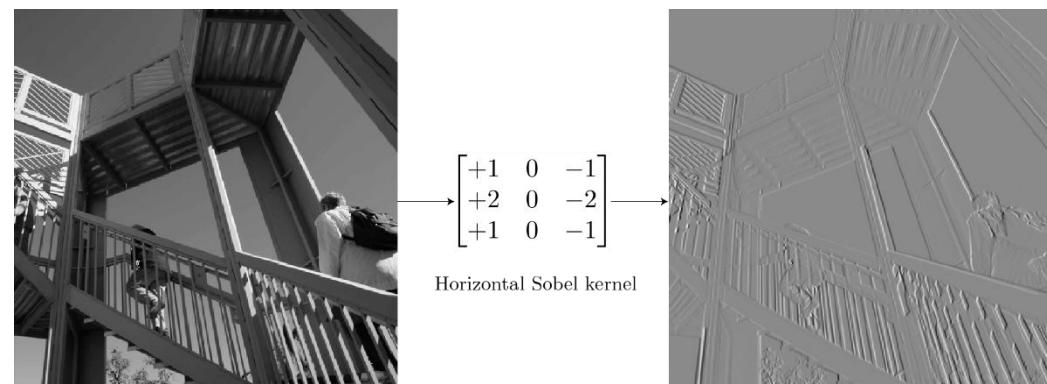
$$\text{output_size} = \left\lfloor \frac{\text{input_size} + 2 \times \text{padding} - (\text{dilation} \times (\text{kernel_size} - 1) + 1)}{\text{stride}} + 1 \right\rfloor$$

Convolution

- 2D discrete convolution (e.g. image)

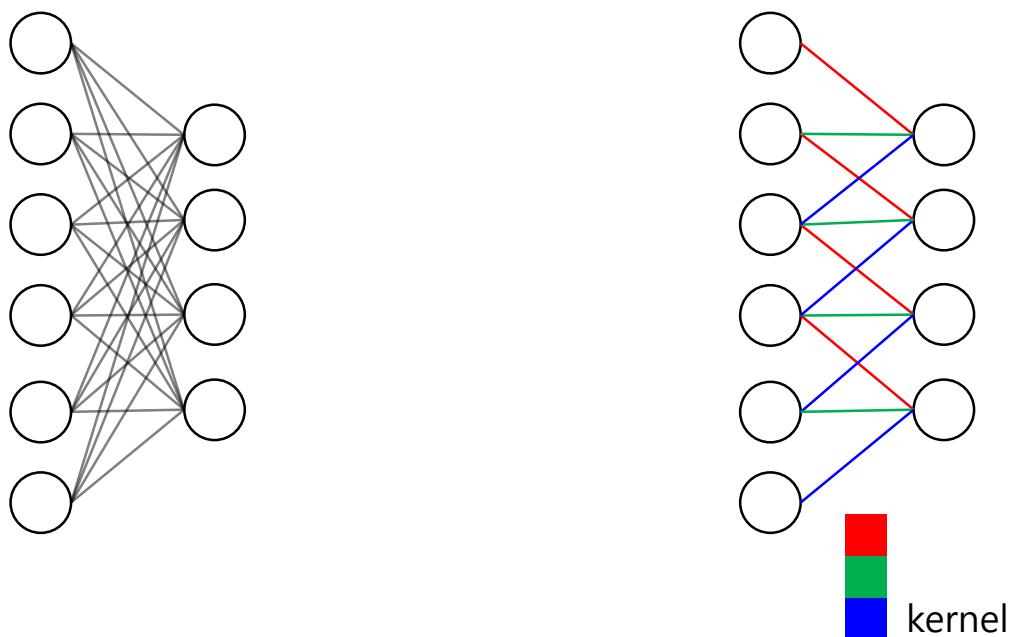


- A kernel and its output are also called a *filter* and *feature map* (or *activation map*), respectively.
 - CNN visualization examples: [ConvNetJS](#), [CNN Explainer](#)
 - e.g. When a kernel is a horizontal Sobel filter,



Convolutional Layer

- A **convolutional layer** (shortly *conv layer*) is a NN layer which uses *convolution* as its feedforward propagation and *kernel* as weight variables.
- (In contrast to a FC layer) A convolutional layer has two important points, **weight sharing** and **local connectivity**.
 - e.g. `input_size = 6, output_size = 4, kernel_size = 3`
FC layer has 24 weight variables, but conv layer has only 3 weight variables.

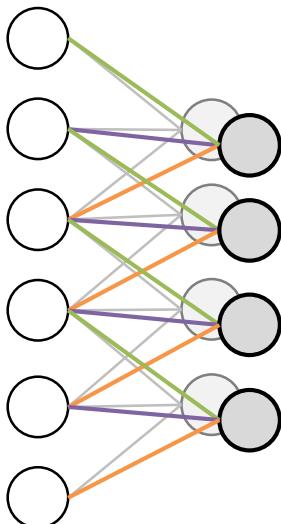


Why it works?

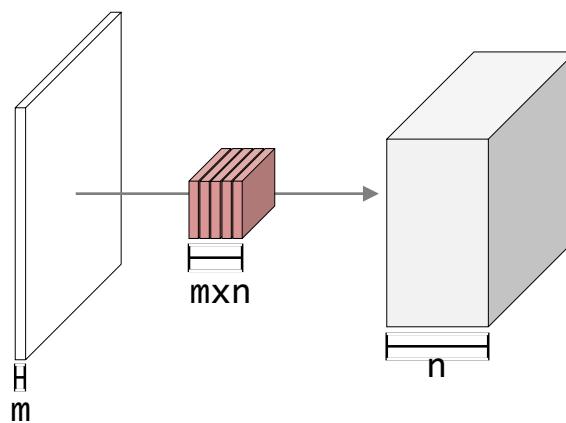


Convolutional Layer

- A convolutional layer can have multiple kernels so that its output will be multiple *channels*.
 - e.g. `in_channel_size = 1, out_channel_size = 2, kernel_size = 3`
 - e.g. `in_channel_size = m, out_channel_size = n, kernel_size = 3`



of weights (w/o bias): 3×2

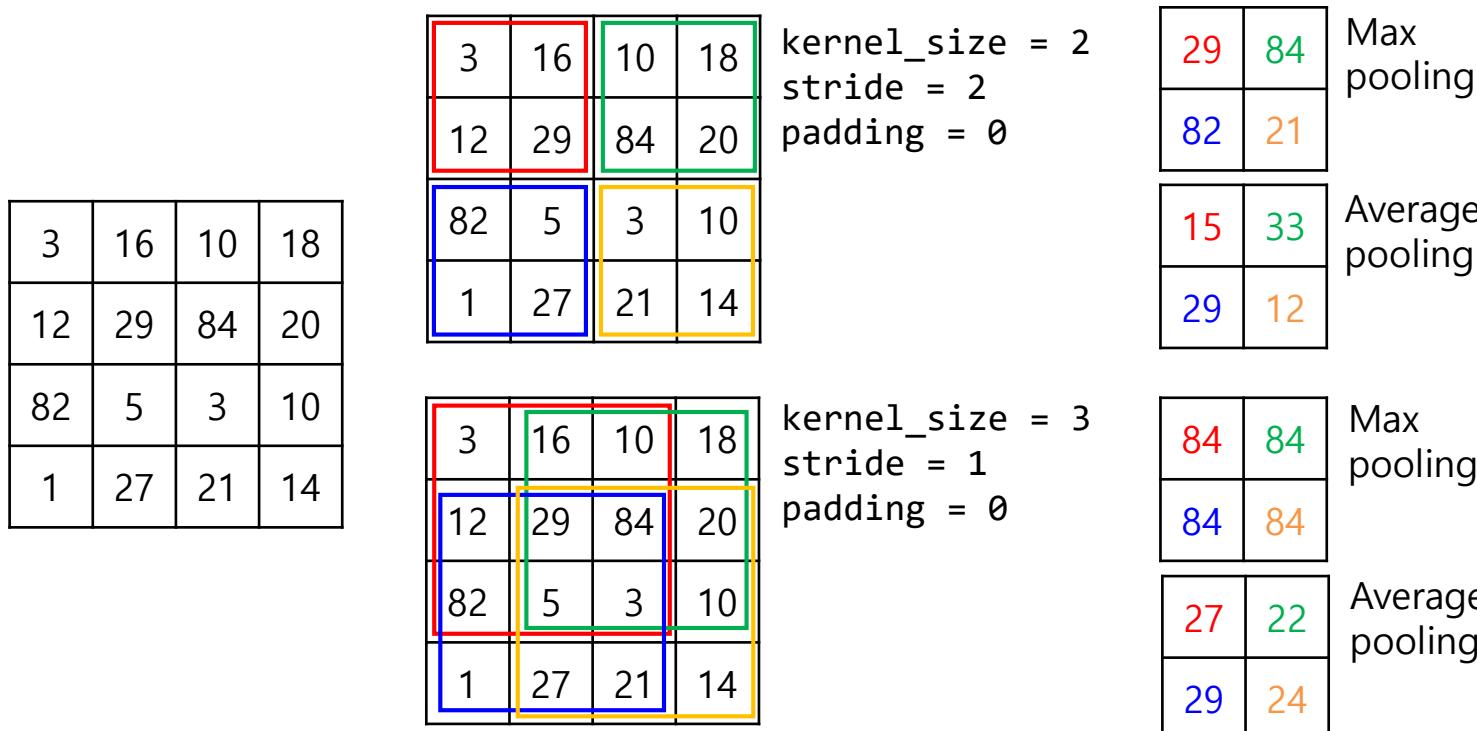


of weights (w/o bias): $3 \times 3 \times m \times n$

- Note) The number of weights is independent from input and output size in a convolutional layer.
 - Note) The number of weight (w/o bias) in a FC layer = input size x output size
 - e.g. The left example: $6 \times 8 = 48$

Pooling Layer

- A **pooling layer** performs a non-linear down-sampling operation.
- Especially max pooling is common.
 - Note) [PyTorch APIs for pooling layers](#)
- Its parameters and working is similar to a convolutional layer.
 - In PyTorch, `stride` is assigned as `kernel_size` if it is not given.



Pooling Layer

- Why is pooling important?
 - It reduces the network size. → Less time/space complexity
 - It provide larger **receptive fields** with a limited size of kernels.



bag



human upper body



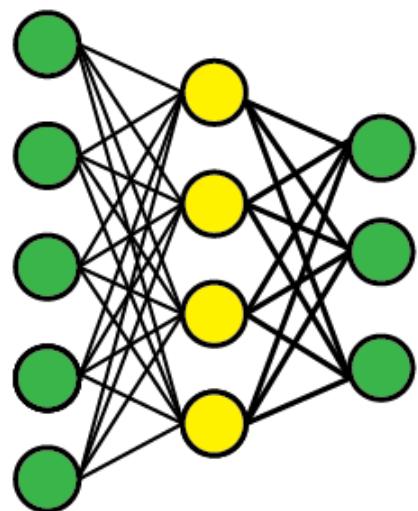
human full body



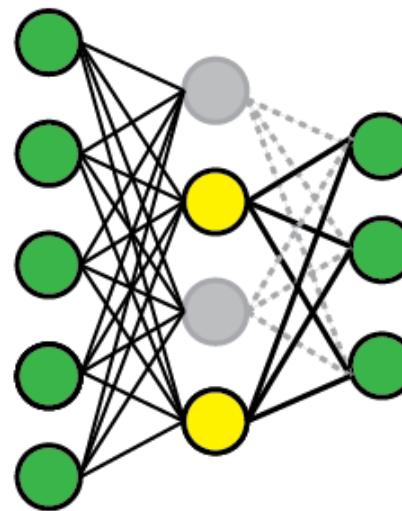
bridge

Dropout

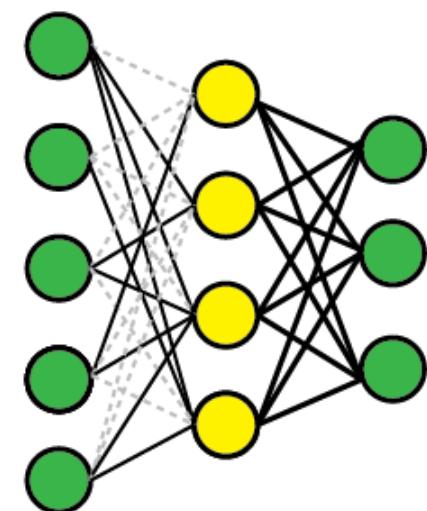
- **Dropout** is one of regularization methods against overfitting.
- It randomly drops a ratio of hidden units during training.
 - It prevents hidden units from co-adaptation.
 - e.g. When some hidden units has high weights, the others in the same layers have little contribution (low weight values, rarely training) to their output.



Standard



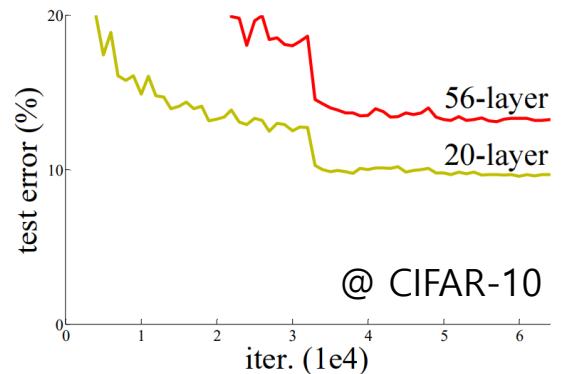
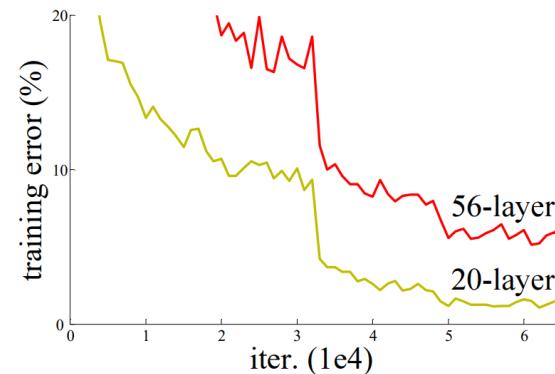
Dropout



DropConnect

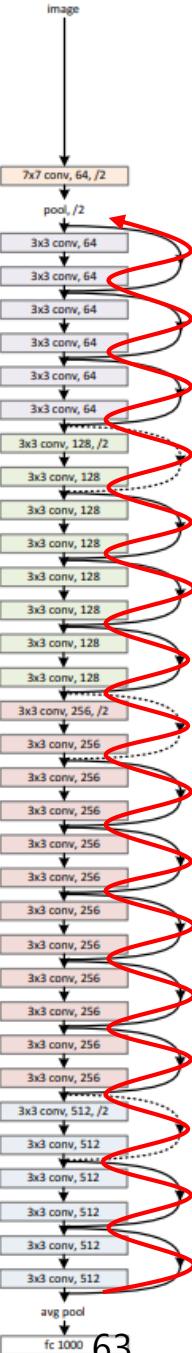
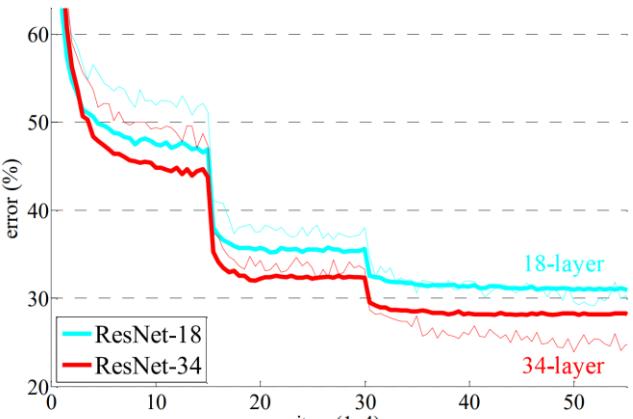
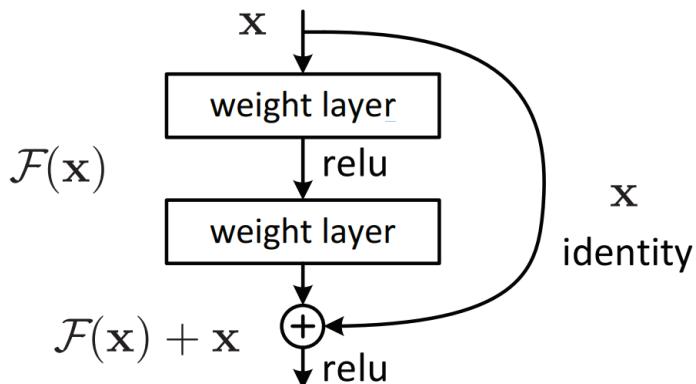
Skip Connection

- Motivation) Why is a deeper network worse?



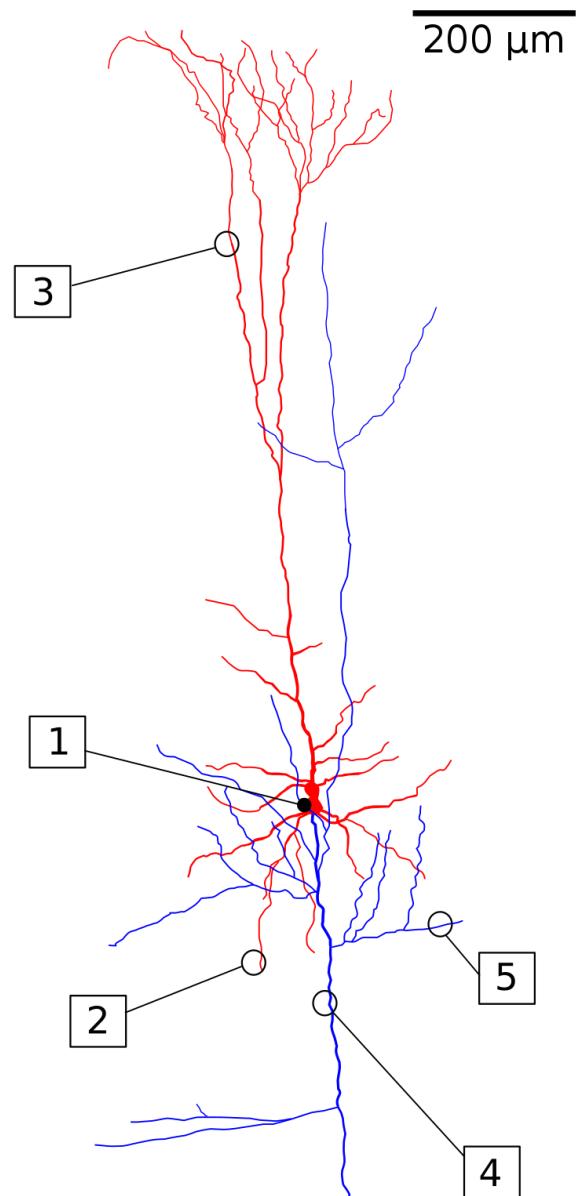
gradient highway

- A **skip connection** is a shortcut connection that bypasses several layers which contain nonlinearities (e.g. ReLU) and batch normalization.
 - Alias: *Residual connection* (used in *residual neural network, ResNet*)
 - It resolves the vanishing gradient problem.



Skip Connection

- Biological analogue [\[Wikipedia\]](#)
 - Cortical layer VI neurons @ [the cerebral cortex](#)



MNIST Dataset

- The [MNIST dataset](#) is a large database of handwritten digits. [\[Wikipedia\]](#)
- It was constructed by “re-mixing” the NIST’s original datasets.
 - Full name: *Modified* National Institute of Standards and Technology Database
- Specification
 - Classes: 10 (0, 1, 2, ..., 9)
 - Images: 28 x 28 (8-bit gray scale)
 - The number of data: 60,000 for training and 10,000 for test



0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9

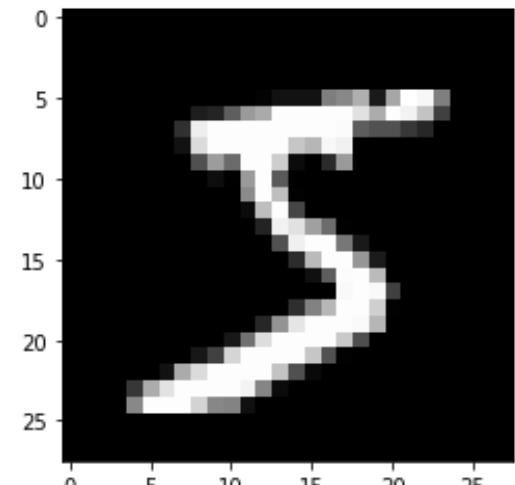
Practice) Loading the MNIST Dataset

```
import torchvision
import matplotlib.pyplot as plt

# Note) You can download the MNIST dataset through its mirror.
# - Reference: https://stackoverflow.com/questions/66577151/http-error-when-trying-to-download-mnist-data
torchvision.datasets.MNIST.resources = [
    ('https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz', 'f68b3c2dcbeaaa9fbdd348bbdeb94873'),
    ('https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz', 'd53e105ee54ea40749a09fcbcd1e9432'),
    ('https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz', '9fb629c4189551a2d022fa330f9573f3'),
    ('https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz', 'ec29112dd5afa0611ce80d1b7f02629c')
]

# Load the MNIST dataset
DATA_PATH = './data'
data_train = torchvision.datasets.MNIST(DATA_PATH, train=True, download=True)
data_valid = torchvision.datasets.MNIST(DATA_PATH, train=False)

# Look inside of the dataset
print(data_train)                      # ... 60000 ...
print(data_valid)                      # ... 10000 ...
print(data_train.data.shape)           # torch.Size([60000, 28, 28])
print(data_train.data.dtype)            # torch.uint8
print(data_train.data[0,:,:,:])        # tensor([[0, 0, ...], ..., [..., 166, 255, 247, ...], ...])
plt.imshow(data_train.data[0,:,:,:], cmap='gray')
plt.show()
print(data_train.targets[0])           # Guess and check it!
```



Practice) Digit Classification with the MNIST Dataset (1/4)

```
# A four-layer CNN model
# - Try more or less layers, channels, and kernel size
# - Try to apply batch normalization (e.g. 'nn.BatchNorm' and 'nn.BatchNorm2d')
# - Try to apply skip connection (used in ResNet)
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN, self).__init__()
        # Notation: (batch_size, channel, height, width)
        # Input : (batch_size, 1, 28, 28)
        # Layer1: conv (batch_size, 32, 28, 28)
        #          pool (batch_size, 32, 14, 14)
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        # Layer2: conv (batch_size, 64, 14, 14)
        #          pool (batch_size, 64, 7, 7)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.drop2 = nn.Dropout(0.2)

        # Input : (batch_size, 64*7*7)
        # Layer3: fc   (batch_size, 512)
        self.fc3 = nn.Linear(64*7*7, 512)
        self.drop3 = nn.Dropout(0.2)

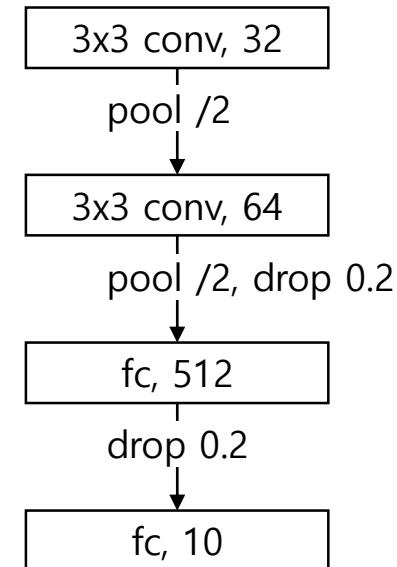
        # Layer4: fc   (batch_size, 10)
        self.fc4 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = self.drop2(x)
        x = torch.flatten(x, 1)

        x = F.relu(self.fc3(x))
        x = self.drop3(x)

        x = F.log_softmax(self.fc4(x), dim=1)
        return x
```



Practice) Digit Classification with the MNIST Dataset (2/4)

```
if __name__ == '__main__':
    # 0. Preparation
    torch.manual_seed(RANDOM_SEED)
    if USE_CUDA:
        torch.cuda.manual_seed_all(RANDOM_SEED)
    dev = torch.device('cuda' if USE_CUDA else 'cpu')

    # 1. Load the MNIST dataset
    preproc = torchvision.transforms.ToTensor()
    data_train = torchvision.datasets.MNIST(DATA_PATH, train=True, download=True, transform=preproc)
    data_valid = torchvision.datasets.MNIST(DATA_PATH, train=False, transform=preproc)
    loader_train = torch.utils.data.DataLoader(data_train, **DATA_LOADER_PARAM)
    loader_valid = torch.utils.data.DataLoader(data_valid, **DATA_LOADER_PARAM)

    # 2. Instantiate a model, loss function, and optimizer
    model = MyCNN().to(dev)
    loss_func = F.cross_entropy
    optimizer = torch.optim.SGD(model.parameters(), **OPTIMIZER_PARAM)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, **SCHEDULER_PARAM)    Note) SCHEDULER_PARAM = { 'step_size': 10, 'gamma': 0.5 }

    # 3.1. Train the model
    loss_list = []
    start = time.time()
    for epoch in range(1, EPOCH_MAX + 1):
        train_loss = train(model, loader_train, loss_func, optimizer)
        valid_loss, valid_accuracy = evaluate(model, loader_valid, loss_func)
        scheduler.step()

        loss_list.append([epoch, train_loss, valid_loss, valid_accuracy])
        if epoch % EPOCH_LOG == 0:
            elapse = (time.time() - start) / 60
            print(f'{epoch:>6} ({elapse:>6.2f} min), TrLoss={train_loss:.6f}, VaLoss={valid_loss:.6f}, VaAcc={valid_accuracy:.3f}, lr={scheduler.get_last_lr()}')
    elapse = (time.time() - start) / 60

    # 3.2. Save the trained model if necessary
    if SAVE_MODEL:
        torch.save(model.state_dict(), SAVE_MODEL)
```

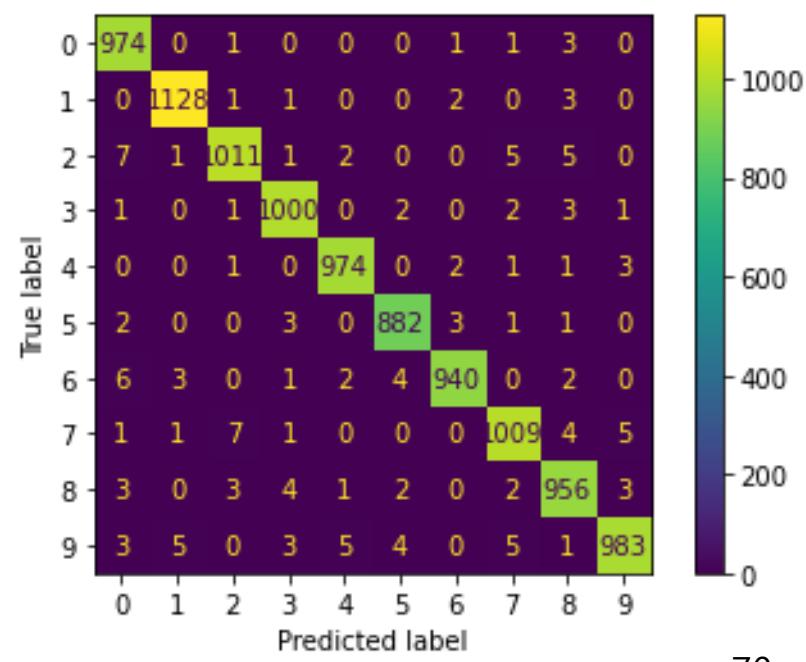
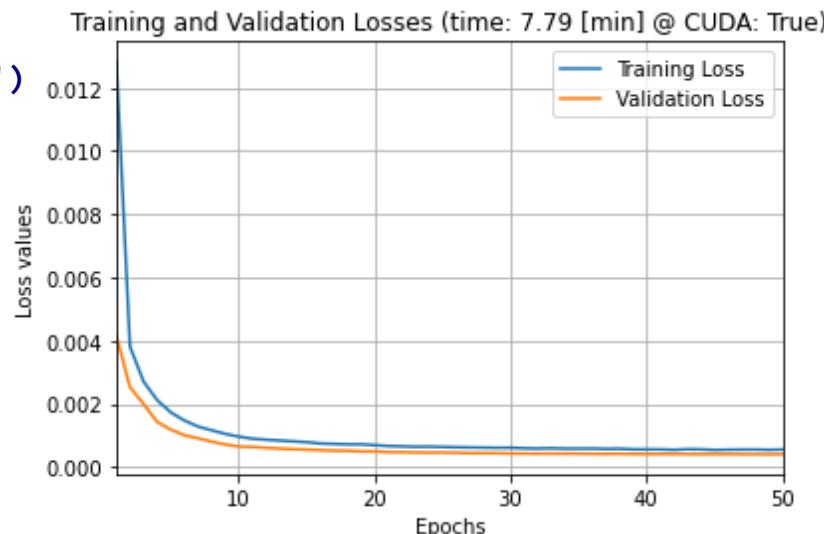
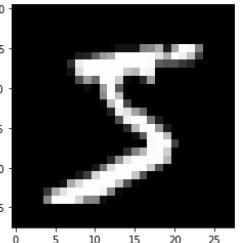
Practice) Digit Classification with the MNIST Dataset (3/4)

```
# 4.1. Visualize the loss curves
plt.title(f'Training and Validation Losses (time: {elapse:.2f} [min] @ CUDA: {USE_CUDA})')
loss_array = np.array(loss_list)
plt.plot(loss_array[:,0], loss_array[:,1], label='Training Loss')
plt.plot(loss_array[:,0], loss_array[:,2], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss values')
plt.xlim(loss_array[0,0], loss_array[-1,0])
plt.grid()
plt.legend()
plt.show()

# 4.2. Visualize the confusion matrix
predicts = [predict(datum, model) for datum in data_valid.data]
conf_mat = metrics.confusion_matrix(data_valid.targets, predicts)
conf_fig = metrics.ConfusionMatrixDisplay(conf_mat)
conf_fig.plot()

# 5. Test your image
print(predict(data_train.data[0], model)) # 5
with PIL.Image.open('data/cnn_mnist_test.png').convert('L') as image:
    print(predict(image, model)) # 3

data_train.data[0]      'data/cnn_mnist_test.png'
```



Practice) Digit Classification with the MNIST Dataset (4/4)

```
# Predict a digit using the given model
def predict(image, model):
    model.eval()
    with torch.no_grad():
        # Convert the given image to its 1 x 1 x 28 x 28 tensor
        if type(image) is torch.Tensor:
            tensor = image.type(torch.float) / 255 # Normalize to [0, 1]
        else:
            tensor = 1 - TF.to_tensor(image)           # Invert (white to black)
        if tensor.ndim < 3:
            tensor = tensor.unsqueeze(0)
        if tensor.shape[0] == 3:
            tensor = TF.rgb_to_grayscale(tensor)      # Make grayscale
        tensor = TF.resize(tensor, 28)                # Resize to 28 x 28
        dev = next(model.parameters()).device
        tensor = tensor.unsqueeze(0).to(dev)           # Add one more dim

        output = model(tensor)
        digit = torch.argmax(output, dim=1)
    return digit.item()
```

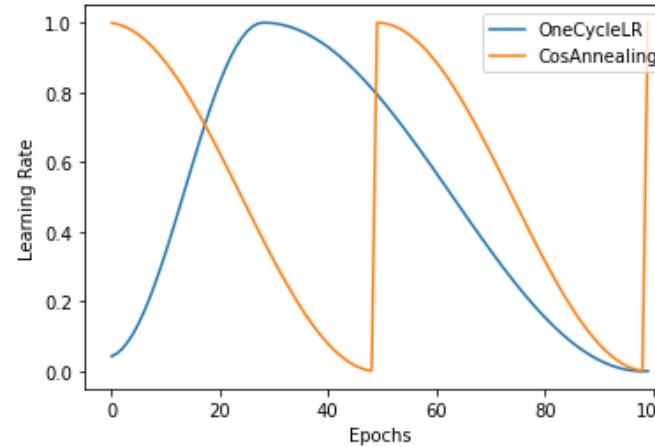
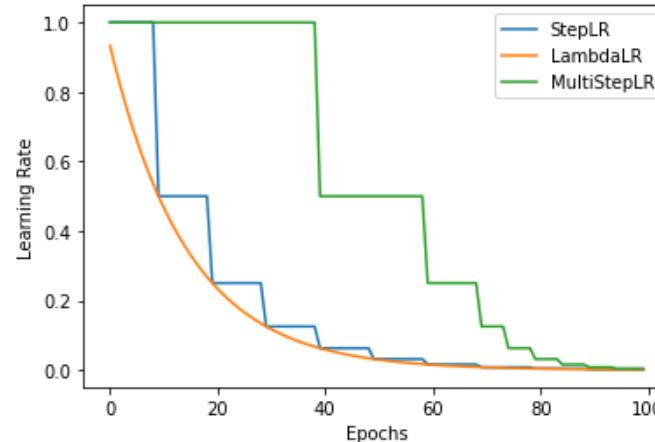
Practice) Loading My Network and Testing My Image

```
import torch, PIL
from cnn_mnist import MyCNN, predict

# Load a model
model = MyCNN()
model.load_state_dict(torch.load('cnn_mnist.pt'))

# Test the model
with PIL.Image.open('data/cnn_mnist_test.png').convert('L') as image:
    print(predict(image, model))
```

Practice) Visualizing Learning Rate Schedulers (1/2)



```
import torch
import torch.optim.lr_scheduler as lr_scheduler
import matplotlib.pyplot as plt

base_lr = 1.
epoch_max = 100
schedulers = [
    {'name': 'StepLR', 'class': lr_scheduler.StepLR,
     'param': {'step_size': 10, 'gamma': 0.5}},
    {'name': 'LambdaLR', 'class': lr_scheduler.LambdaLR,
     'param': {'lr_lambda': lambda epoch: 0.5**((epoch / 10))}},
    {'name': 'MultiStepLR', 'class': lr_scheduler.MultiStepLR,
     'param': {'milestones': [40, 60, 70, 75, 80, 85, 90, 95], 'gamma': 0.5}},
    {'name': 'OneCycleLR', 'class': lr_scheduler.OneCycleLR,
     'param': {'max_lr': base_lr, 'total_steps': epoch_max}},
    {'name': 'CosAnnealing', 'class': lr_scheduler.CosineAnnealingWarmRestarts,
     'param': {'T_0': 50}},
    # Try more learning rate schedulers
]
```

Practice) Visualizing Learning Rate Schedulers (2/2)

```
for sch in schedulers:  
    x = torch.tensor(1., requires_grad=True) # A dummy parameter  
    optimizer = torch.optim.SGD([x], lr=base_lr) # Instantiate an optimizer  
    scheduler = sch['class'](optimizer, **sch['param']) # Instantiate a LR scheduler  
    lr_values = []  
    for i in range(epoch_max):  
        optimizer.step()  
        scheduler.step()  
        lr_values.append(optimizer.param_groups[0]['lr'])  
    plt.plot(range(epoch_max), lr_values, label=sch['name'])  
  
plt.xlabel('Epochs')  
plt.ylabel('Learning Rate')  
plt.legend()  
plt.show()
```

Practice) Different Styles for NN Classes (1/2)

My style

```
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1, 1)
        self.pool1 = nn.MaxPool2d(2)

        self.conv2 = nn.Conv2d(32, 64, 3, 1, 1)
        self.pool2 = nn.MaxPool2d(2)
        self.drop2 = nn.Dropout(0.2)

        self.fc3 = nn.Linear(64*7*7, 512)
        self.drop3 = nn.Dropout(0.2)

        self.fc4 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = self.drop2(x)
        x = torch.flatten(x, 1)

        x = F.relu(self.fc3(x))
        x = self.drop3(x)

        x = F.log_softmax(self.fc4(x), dim=1)
        return x
```

Functional-oriented style

```
class MyCNN_Functional(nn.Module):
    def __init__(self):
        super(MyCNN_FStyle, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1, 1)
        self.fc1 = nn.Linear(64*7*7, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)

        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = F.dropout(x, 0.2, self.training)
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.2, self.training)

        x = F.log_softmax(self.fc2(x), dim=1)
        return x
```

Practice) Different Styles for NN Classes (2/2)

Object-oriented style

```
class MyCNN_Object(nn.Module):
    def __init__(self):
        super(MyCNN_ObjStyle, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1, 1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2)

        self.conv2 = nn.Conv2d(32, 64, 3, 1, 1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2)
        self.drop2 = nn.Dropout(0.2)

        self.fc3 = nn.Linear(64*7*7, 512)
        self.relu3 = nn.ReLU()
        self.drop3 = nn.Dropout(0.2)

        self.fc4 = nn.Linear(512, 10)
        self.smax4 = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = self.drop2(x)
        x = torch.flatten(x, 1)

        x = self.fc3(x)
```

Layer-oriented style

```
class MyCNN_Layer(nn.Module):
    def __init__(self):
        super(MyCNN_SeqStyle, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.2))

        self.layer3 = nn.Sequential(
            nn.Linear(64*7*7, 512),
            nn.ReLU(),
            nn.Dropout(0.2))

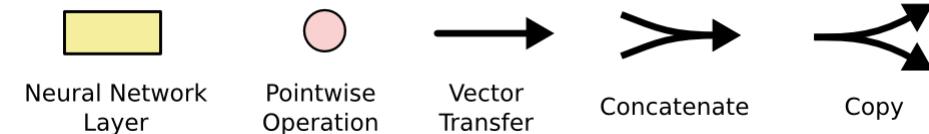
        self.layer4 = nn.Sequential(
            nn.Linear(512, 10),
            nn.LogSoftmax(dim=1))

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = torch.flatten(x, 1)
        x = self.layer3(x)
        x = self.layer4(x)
        return x
```

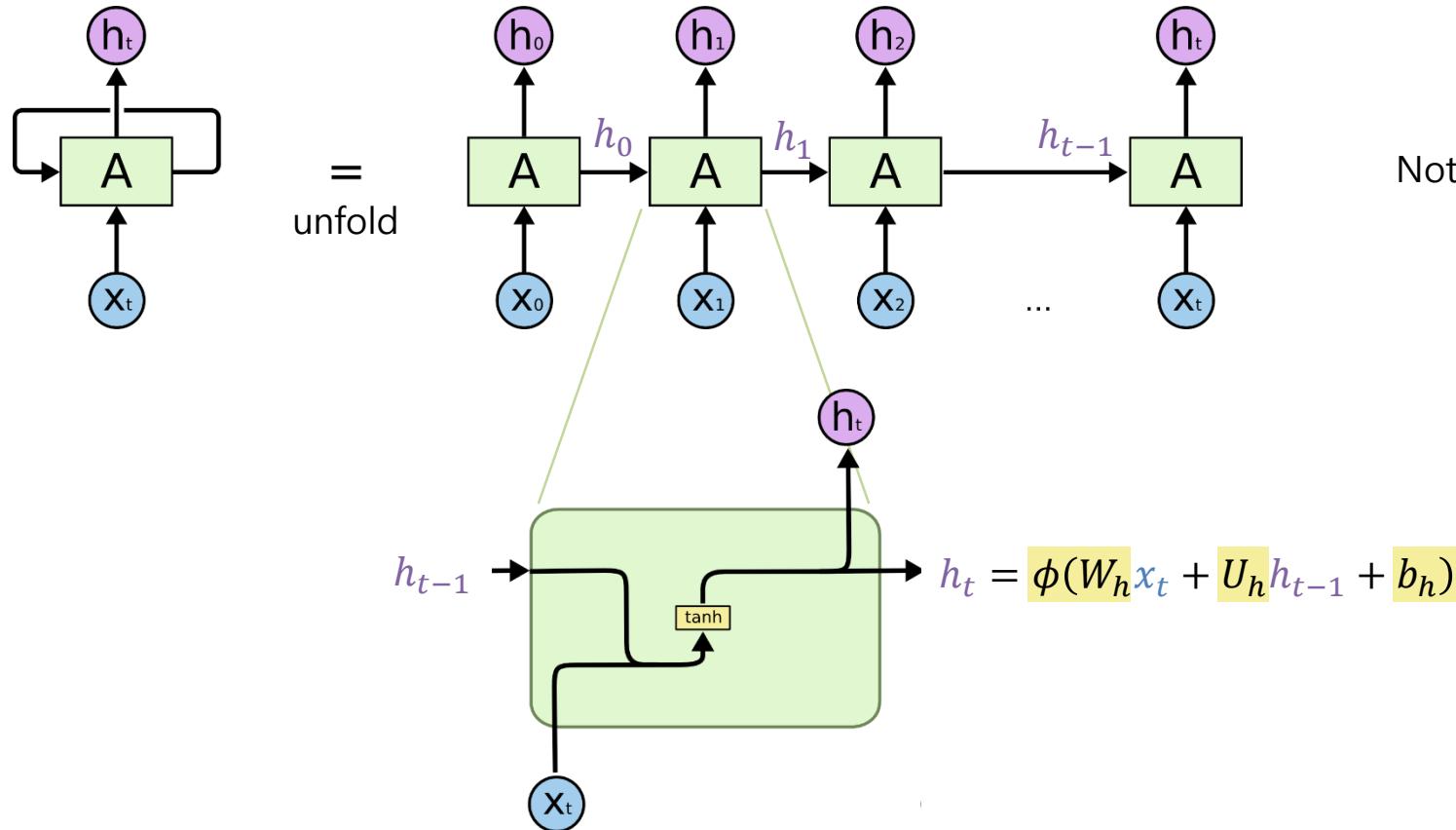
Table of Contents

- **Introduction**
- **PyTorch**
- **Neural Network (NN)**
- **Convolutional Neural Network (CNN)**
- **Recurrent Neural Network (RNN)**
 - Recurrent neural network (RNN) unit
 - Long short-term memory (LSTM) unit
 - Gated recurrent unit (GRU)
 - Example) Name2Lang Classification with a Character-level RNN

Recurrent Neural Network (RNN)

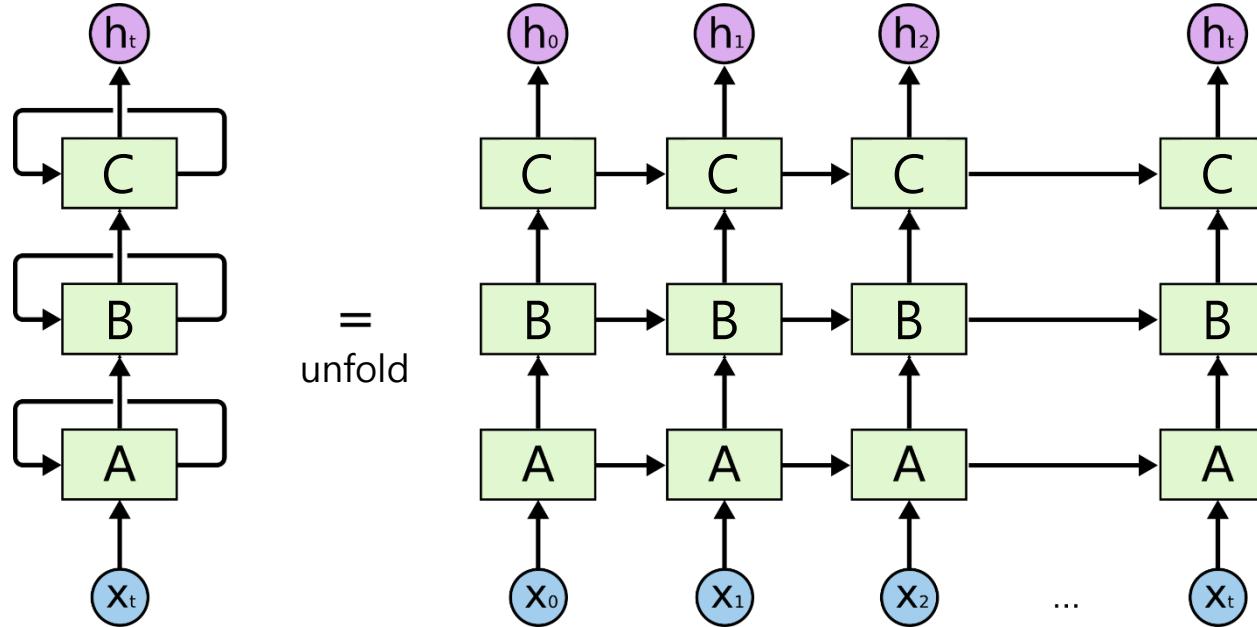


- A **recurrent neural network** (shortly *RNN*) is a NN with a **loop**.
- A RNN can preserve a *memory* or *(hidden) state* or *information* inside.
 - Note) \sim sequential logic circuits (vs. combinatorial logic circuits) in digital circuits



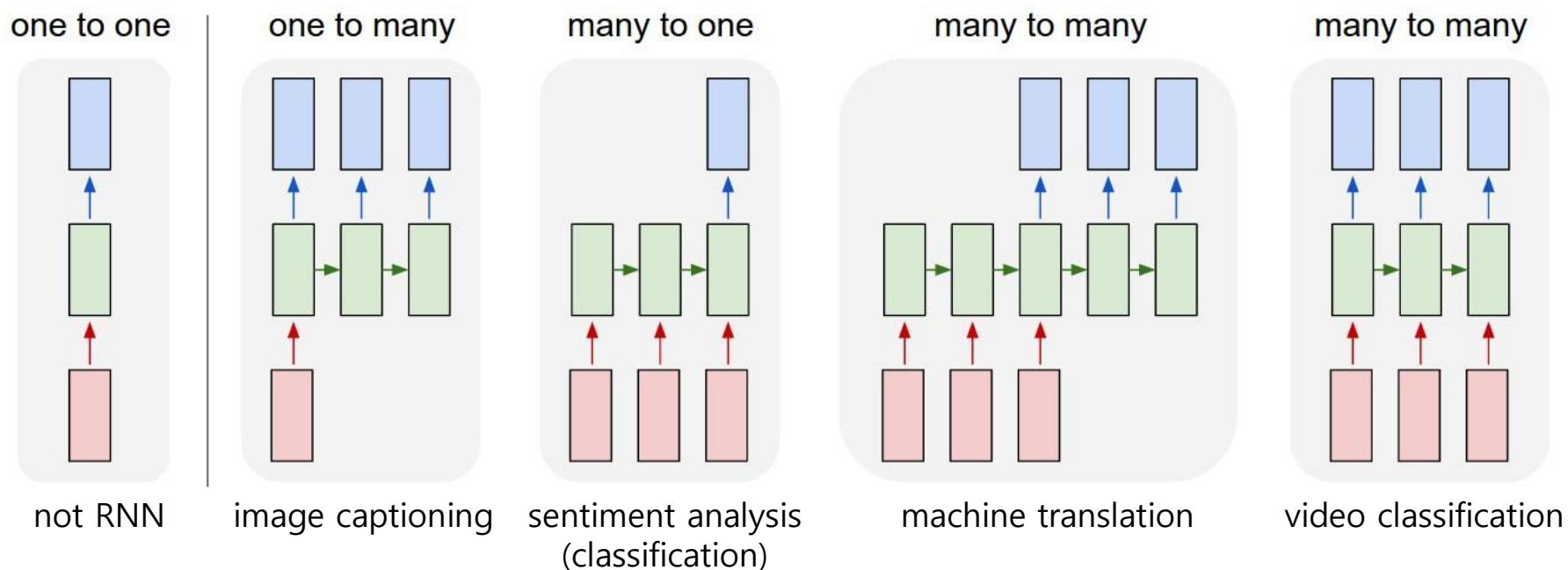
Recurrent Neural Network (RNN)

- Multi-layer RNNs



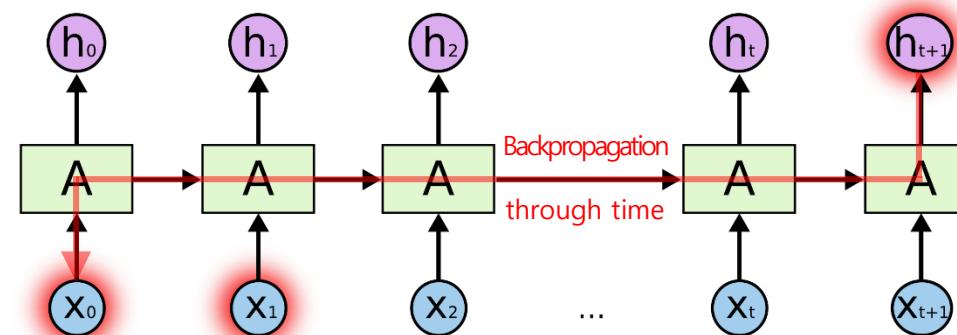
Recurrent Neural Network (RNN)

- Memory in a RNN allows it to deal with *sequential* or *temporal* data.
- A **RNN** can deal with **various lengths of input vectors** and **output vectors** by attaching more hidden/output layers **at its end**.

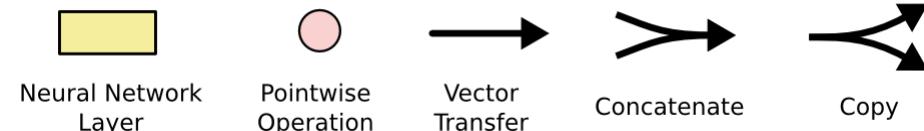


Recurrent Neural Network (RNN)

- The **vanishing gradient problem** in training a long sequence data
 - A vanilla RNN cannot deal with long-term dependency.
 - e.g. Guessing the last word, "I grew up in France ... I speak fluent French."

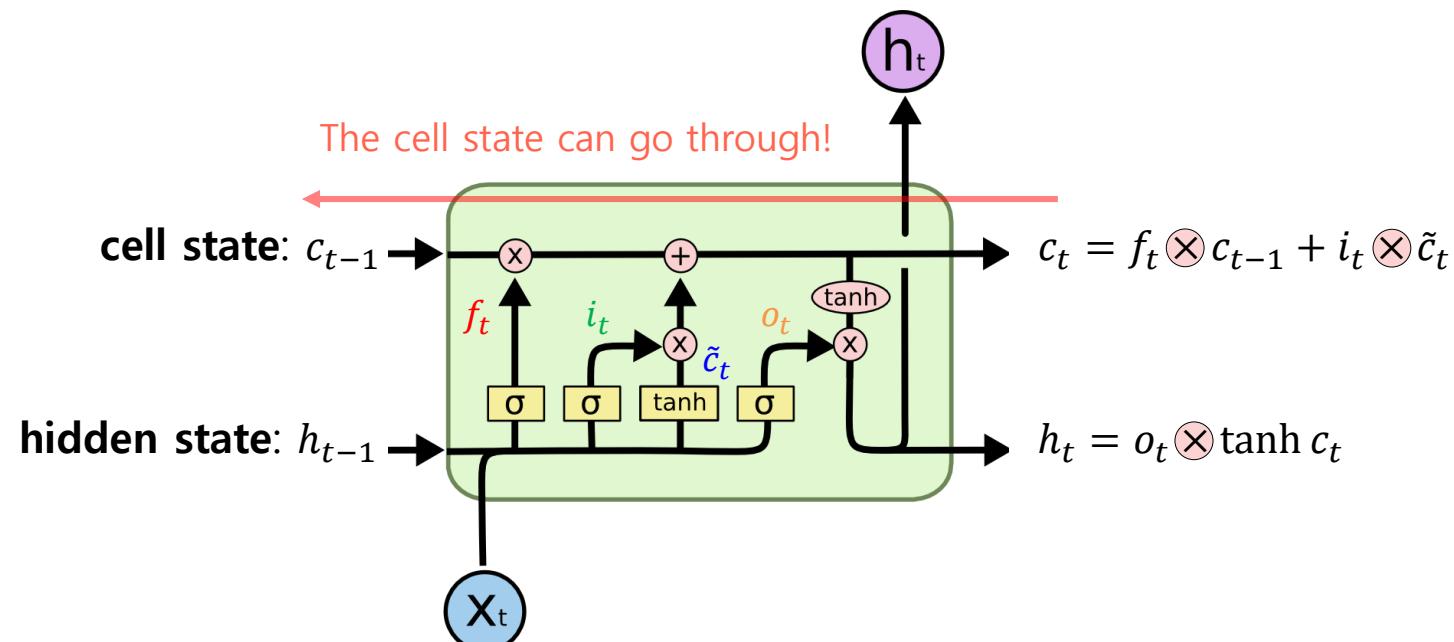


Long Short-Term Memory (LSTM)



- A **long short-term memory** (shortly *LSTM*) is a recurrent neural network unit to deal with the vanishing gradient problem of a vanilla RNN. [\[Wikipedia\]](#)
- It is composed of a **forget gate**, an **input gate**, a **cell**, and an **output gate**.

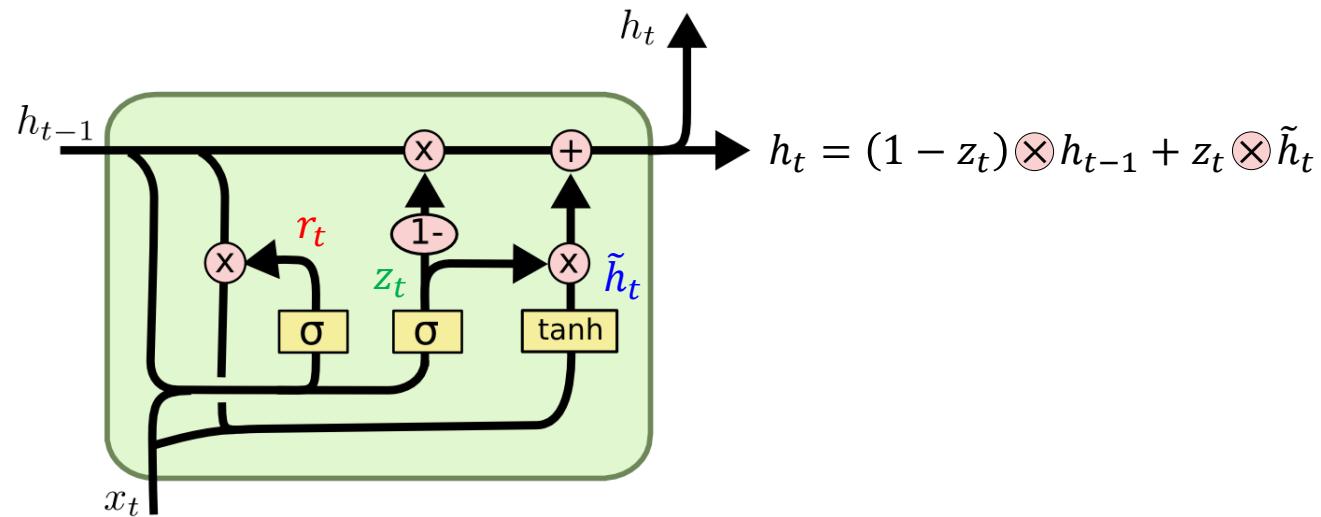
forget gate	How much forget c_{t-1} to the cell state	$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
input gate	How much write \tilde{c}_t to the cell state	$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
cell	The intermediate cell state	$\tilde{c}_t = \tanh(W_o x_t + U_o h_{t-1} + b_o)$
output gate	How much reveal c_t to the hidden state	$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$



Gated Recurrent Unit (GRU)

- A **gated recurrent unit** (shortly *GRU*) is a LSTM variant with a simplified structure, but it still has similar performance. [\[Wikipedia\]](#)

reset gate	How much forget h_{t-1} to the hidden state	$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$
update gate	How much write \tilde{h}_t to the hidden state	$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$
	The intermediate hidden state	$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \otimes h_{t-1}) + b_h)$



Practice) Name2Lang Classification with a Character-level RNN (1/7)

- Motivation: A registration form on Internet

First Name

Last Name

E-mail address

Country

- Input: Name** (string; a sequence of characters)

- e.g. 'Choi', 'Jane', 'Daniel', 'Chow', 'Tanaka', ...

- Classes: 18 languages** (integer; 0-17)

- e.g. Arabic, Chinese, Czech, Dutch, English, French, German, Greek, Iris, Italian, Japanese, Korean, Polish, Portuguese, Russian, Scottish, Spanish, Vietnamese

- The dataset and bottom-up implementation are available on [the PyTorch official tutorial](#).

Practice) Name2Lang Classification with a Character-level RNN (2/7)

- **Input:** **Name** (string; a sequence of characters)
 - e.g. 'Choi', 'Jane', 'Daniel', 'Chow', 'Tanaka', ...
- How to represent the **name**?
 - The name is represented using 57 characters (52 alphabets and 5 special letters).
 - e.g. 'a' (0x61), 'b' (0x62), 'c' (0x63), ..., '-' (0x45) in ASCII code
 - Each character is encoded as a 57-bit one-hot vector.

- e.g. a:

1	0	0	0	...
---	---	---	---	-----
- b:

0	1	0	0	...
---	---	---	---	-----
- c:

0	0	1	0	...
---	---	---	---	-----

Why one-hot encoding?

- e.g. 'Choi' represented as a 4 x 57 array

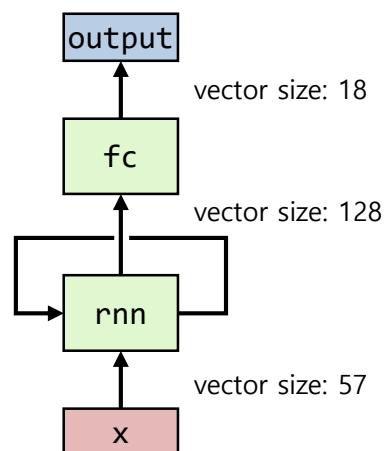
C:	0	...	0	0	0	...	0	0	...	1	0	...
h:	0	...	1	0	0	...	0	0	...	0	0	...
o:	0	...	0	0	0	...	1	0	...	0	0	...
i:	0	...	0	1	0	...	0	0	...	0	0	...

Index) 7 8 14 28

Practice) Name2Lang Classification with a Character-level RNN (3/7)

```
# A simple RNN model
# - Try a different RNN unit such LSTM and GRU
# - Try less or more hidden units
# - Try more layers (e.g. 'num_layers=2') and dropout (e.g. 'dropout=0.4')
class MyRNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(MyRNN, self).__init__()
        self.rnn = torch.nn.RNN(input_size, 128)
        self.fc = torch.nn.Linear(128, output_size)

    def forward(self, x):
        output, hidden = self.rnn(x)
        x = self.fc(output[-1]) # Use the output of the last sequence
        return x
```



Practice) Name2Lang Classification with a Character-level RNN (4/7)

```
# Convert Unicode to ASCII
# e.g. Ślusàrski to Slusarski
def unicode2ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn' and c in LETTER_DICT)

# Read raw files which contain names belong to each language
# Note) Each filename is used as its target's name.
def load_name_dataset(files):
    data = []
    targets = []
    target_names = []
    for idx, filename in enumerate(files):
        lang = os.path.splitext(os.path.basename(filename))[0]
        names = open(filename, encoding='utf-8').read().strip().split('\n')

        data += [unicode2ascii(name) for name in names]
        targets += [idx] * len(names)
        target_names.append(lang)
    return data, targets, target_names

# Transform the given text to its one-hot encoded tensor
# Note) Tensor size: len(text) x 1 x len(LETTER_DICT)
#           sequence_length x batch_size x input_size
def text2onehot(text, device='cpu'):
    tensor = torch.zeros(len(text), 1, len(LETTER_DICT), device=device)
    for idx, letter in enumerate(text):
        tensor[idx][0][LETTER_DICT.find(letter)] = 1
    return tensor
```

 Arabic.txt	13KB
 Chinese.txt	2KB
 Czech.txt	4KB
 Dutch.txt	3KB
 English.txt	27KB
 French.txt	3KB
 German.txt	6KB
 Greek.txt	2KB
 Irish.txt	2KB
 Italian.txt	6KB
 Japanese.txt	8KB
 Korean.txt	1KB
 Polish.txt	2KB
 Portuguese.txt	1KB
 Russian.txt	84KB
 Scottish.txt	1KB
 Spanish.txt	3KB
 Vietnamese.txt	1KB

Practice) Name2Lang Classification with a Character-level RNN (5/7)

```
if __name__ == '__main__':
    # 0. Preparation
    # 1. Load the name2lang dataset
    data, targets, target_names = load_name_dataset(glob.glob(DATA_PATH))
    data_train = [(text2onehot(data[i], device=dev), torch.LongTensor(
        [targets[i]]).to(dev)) for i in range(len(data))]
    random.shuffle(data_train)

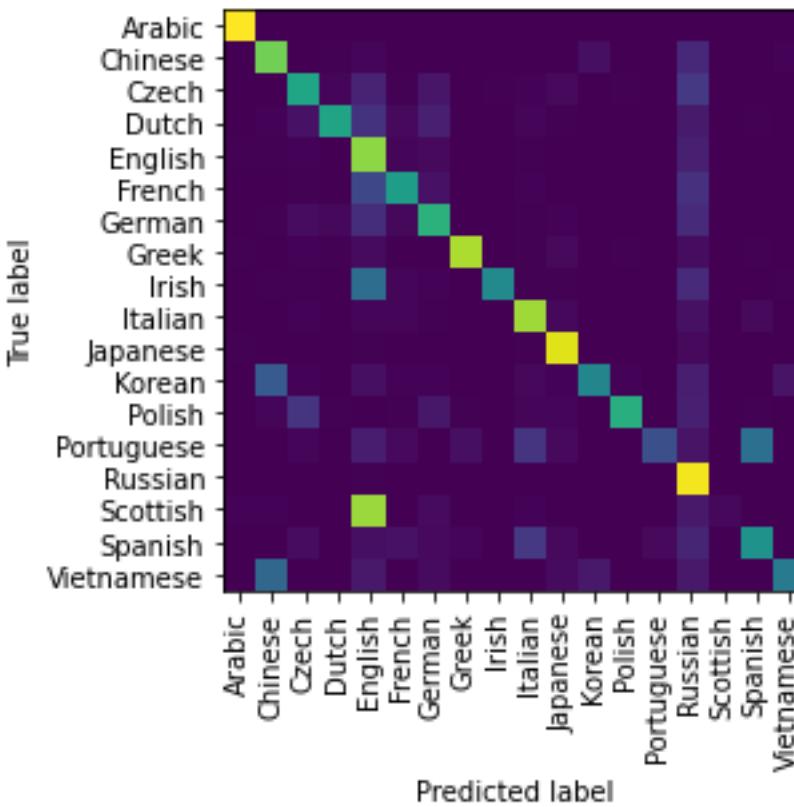
    # 2. Instantiate a model, loss function, and optimizer
    # 3.1. Train the model
    # 3.2. Save the trained model if necessary

    # 4.1. Visualize the loss curves
    # 4.2. Visualize the confusion matrix
    predicts = [predict(datum, model) for datum in data]
    conf_mat = sklearn.metrics.confusion_matrix(targets, predicts, normalize='true')
    plt.imshow(conf_mat)
    plt.xlabel('Predicted label')
    plt.ylabel('True label')
    plt.gca().set_xticklabels([''] + target_names, rotation=90)
    plt.gca().set_yticklabels([''] + target_names)
    plt.gca().xaxis.set_major_locator(ticker.MultipleLocator(1))
    plt.gca().yaxis.set_major_locator(ticker.MultipleLocator(1))
    plt.show()

    # 5. Test your texts
    report_predict('Choi', model, target_names)
    report_predict('Jane', model, target_names)
    report_predict('Daniel', model, target_names)
    report_predict('Chow', model, target_names)
    report_predict('Tanaka', model, target_names)
```

Practice) Name2Lang Classification with a Character-level RNN (6/7)

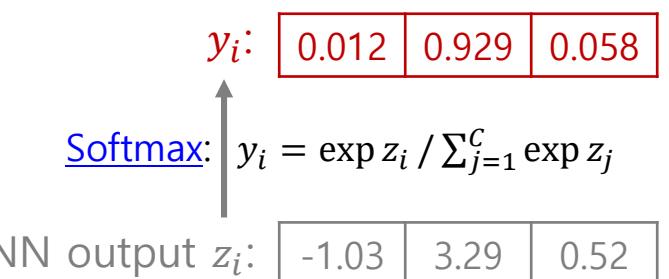
* Name: Choi	
1. Korean	: 50.5 %
2. Chinese	: 36.1 %
3. Vietnamese	: 11.8 %
4. Russian	: 0.9 %
5. Arabic	: 0.5 %
* Name: Jane	
1. English	: 62.4 %
2. German	: 10.5 %
3. Korean	: 9.0 %
4. Chinese	: 7.0 %
5. Dutch	: 6.2 %
* Name: Daniel	
1. English	: 48.4 %
2. French	: 20.6 %
3. Czech	: 13.1 %
4. Russian	: 6.6 %
5. Portuguese	: 3.1 %
* Name: Chow	
1. Korean	: 69.8 %
2. Chinese	: 16.0 %
3. English	: 4.7 %
4. Vietnamese	: 4.3 %
5. Russian	: 3.1 %
* Name: Tanaka	
1. Japanese	: 84.4 %
2. Russian	: 14.0 %
3. Czech	: 1.3 %
4. English	: 0.1 %
5. Irish	: 0.1 %



Practice) Name2Lang Classification with a Character-level RNN (7/7)

```
# Predict the best result of the given text
def predict(text, model):
    model.eval()
    with torch.no_grad():
        dev = next(model.parameters()).device
        text_tensor = text2onehot(text, dev) # Convert text to one-hot vectors
        output = model(text_tensor)[0]      # Get the last output
        lang = torch.argmax(output)       # Get the best among 18 classes
    return lang.item()

# Predict and report top-k results of the given text
def report_predict(text, model, target_names, n_predict=5):
    print(f'* Name: {text}')
    model.eval()
    with torch.no_grad():
        dev = next(model.parameters()).device
        text_tensor = text2onehot(text, dev)
        output = model(text_tensor)[0]
        prob = nn.functional.softmax(output, dim=0) # Make output as probability
        top_val, top_idx = prob.topk(n_predict)     # Get top-k among 18 classes
        for i in range(len(top_val)):
            print(f' {i+1}. {target_names[top_idx[i].item()]}:{<10}:
{top_val[i]*100:4.1f} %')
```



Summary

- **DNN**: Deep neural network
 - **CNN**: Feedforward with convolution (and pooling)
 - **RNN**: NN with a loop → state/memory → sequential/temporal data
-
- **Issue #1) Vanishing gradient problem**
 - Activation functions such as ReLU
 - Skip connection ~ LSTM and GRU
 - **Issue #2) Overfitting problem**
 - More data by data collection or data augmentation or data synthesis
 - Data separation (train/validation/test data), cross-validation, and early stopping
 - More simplified models (e.g. CNN ← weight sharing and local connectivity; a.k.a. inductive bias)
 - Loss functions with regularization terms
 - Other improvement
 - Dropout
 - Batch normalization
 - ...

Further Information

- Natural Language Processing (NLP)
 - **Seq2seq** (2014), **attention mechanism** (2015)
 - **Transformer** (2017), **BERT** (Bidirectional Encoder Representations from Transformers), **GPT** (Generative Pretrained Transformer), ...
- Computer Vision
 - CNN backbone networks: **AlexNet** (2012), **VGGNet** (2014), **Inception Net** (2014), **ResNet** (2015), ...
 - CNN object detection networks
 - Two-stage detectors: **R-CNN** (2014), **Fast R-CNN** (2015), **Faster R-CNN** (2015), **Mask R-CNN** (2017), ...
 - One-stage detectors: **YOLO** (You Only Look Once; 2015), **SSD** (Single Shot MultiBox Detector; 2015), ...
 - Vision Transformers (ViT) and Vision Foundation Models (VFM)
 - **ViT** (Vision Transformer; 2020), **Swin Transformer** (2021), **DINO** (2021), ..., **SAM** (Segment Anything; 2023), ...
 - Generative models
 - **Autoencoder** (2006), ..., **GAN** (Generative Adversarial Networks; 2014), **CycleGAN** (2017), ...
 - **Diffusion model** (2020), **Stable Diffusion** (2022), ...
 - Others
 - **NeRF** (Neural Radiance Field; 2020), **CLIP** (Contrastive Language-Image Pre-Training; 2021), ...
- Others
 - Incremental/continual learning, ..., transfer learning/domain adaptation, ..., contrastive learning, ...
 - Network compression/pruning, knowledge distillation (2015), ..., ML model deployment, ...

How To Understand New Methods (or Read Papers)

- **Model**

- Network design
- Input and output
 - e.g. Normalization, positional encoding

- **Training**

- Dataset
 - e.g. Data augmentation, data synthesis
 - e.g. Data labeling, unsupervised or weakly-supervised approaches
- Objective functions
 - e.g. Regularization terms, Huber loss (for robust regression), focal loss
- Optimization
 - e.g. Optimization algorithms, hyperparameters