Review 2: MLP

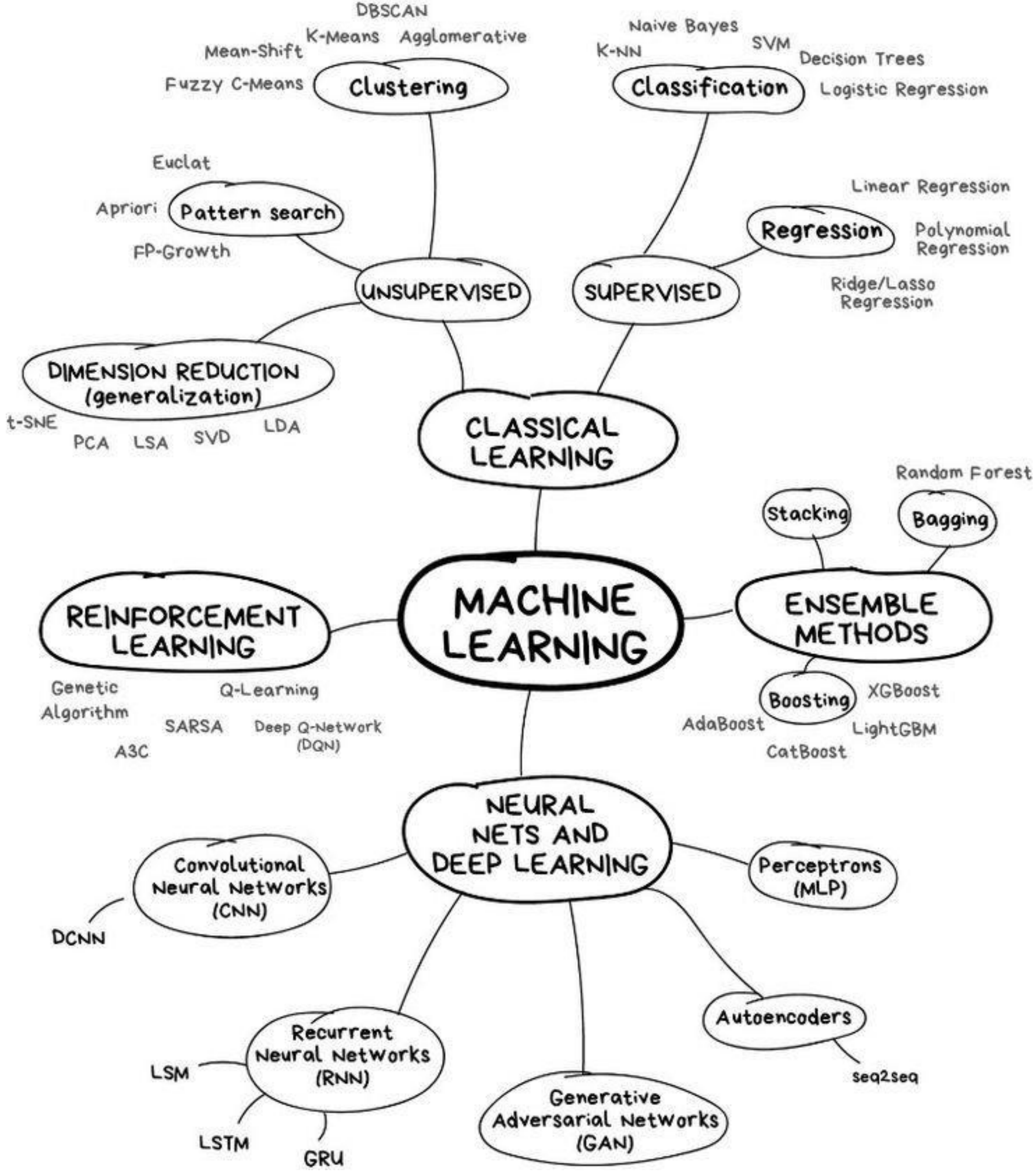
한양대학교 ERICA 소프트웨어융합대학 COLLEGE OF COMPUTING | 인공지능학과 Department of Artificial Intelligence

정 우 환 (whjung@hanyang.ac.kr)

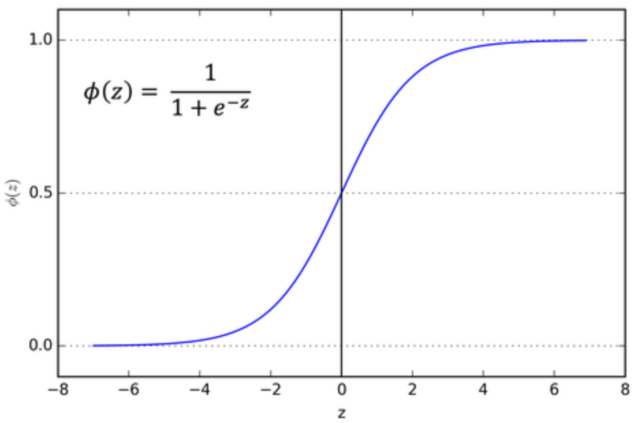
Fall 2021

Review

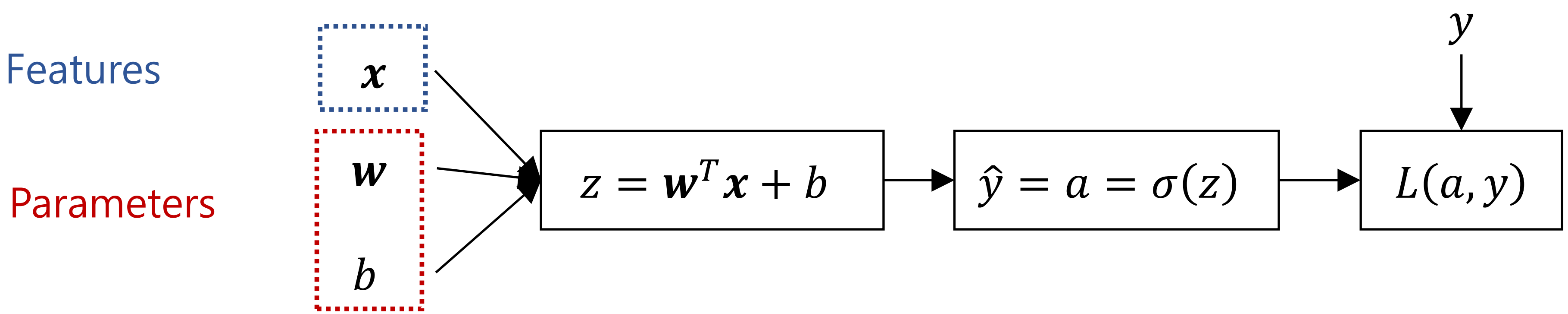
- Linear regression
- Logistic regression
- Multilayer perceptrons (MLP)



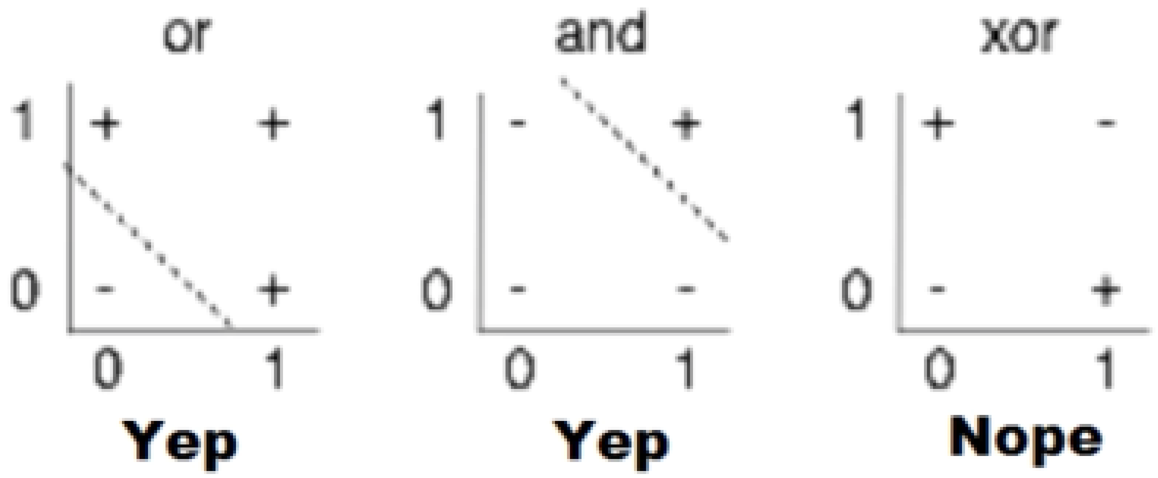
Logistic Regression



- Output: $\hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$
- Loss: $L(\hat{y}, y) = -y \log \hat{y} - (1-y)\log(1-\hat{y})$



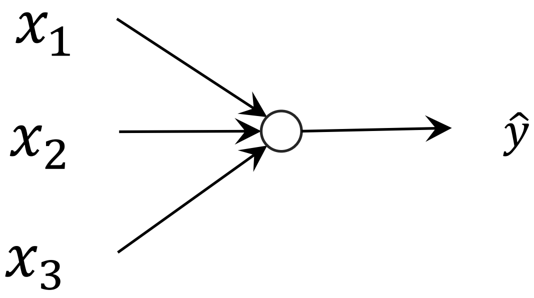
(Simple) XOR problem: linearly separable?

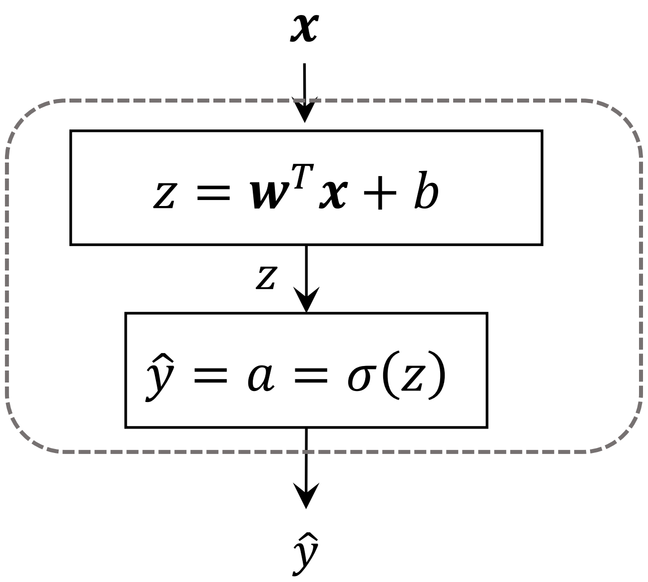


Solution: make it more complicated

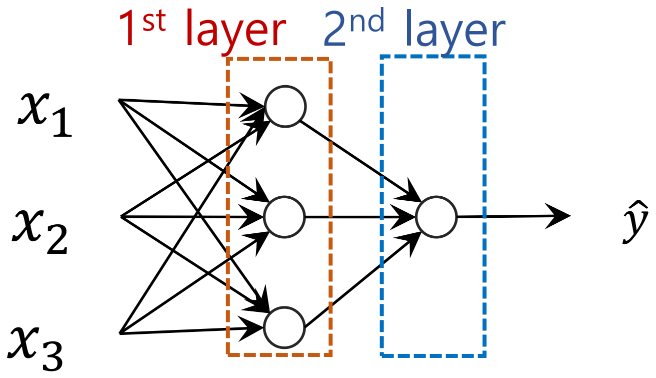
Shallow Neural Networks

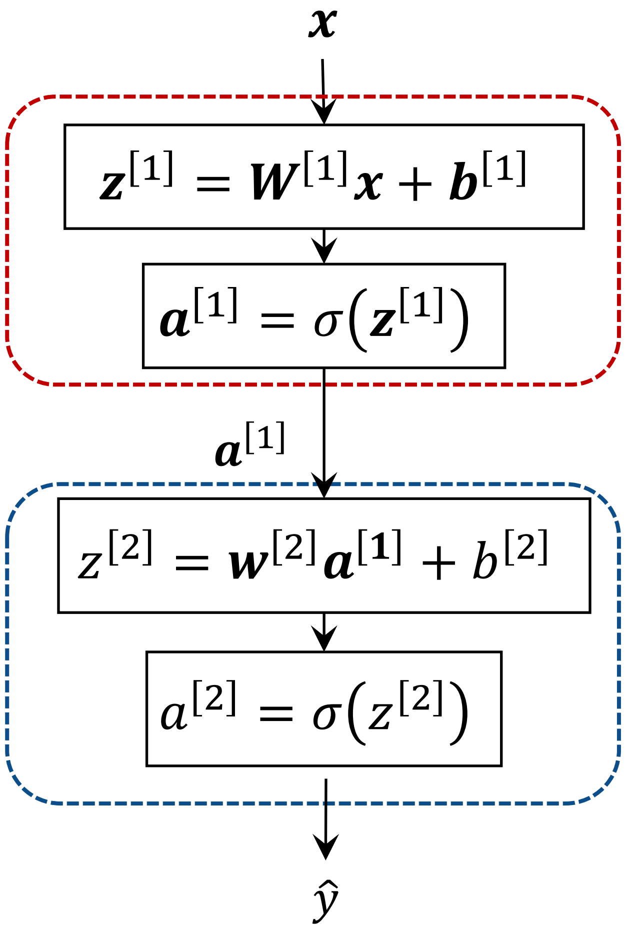
Logistic regression



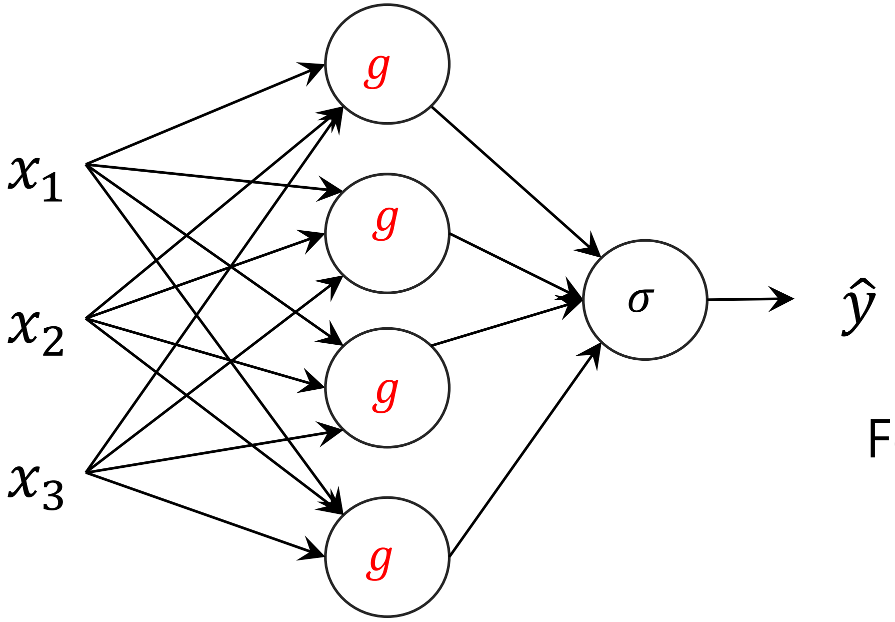


Neural Networks (1 hidden layer)





Neural Network with a Hidden Layer: Almost Done!



Input:

$\boldsymbol{x} \in \mathbb{R}^m$

Parameters:

$\boldsymbol{W}^{[1]} \in \mathbb{R}^{h \times m}$ $\boldsymbol{w}^{[2]} \in \mathbb{R}^h$

$\boldsymbol{b}^{[1]} \in \mathbb{R}^h$ $b^{[2]} \in \mathbb{R}$

Forward pass:

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$\boldsymbol{a}^{[1]} = g\left(\boldsymbol{z}^{[1]}\right)$ where $g(.)$ is an activation function

$$z^{[2]} = {\boldsymbol{w}^{[2]}}^T \boldsymbol{a}^{[1]} + b^{[2]}$$

$$\hat{y} = a^{[2]} = \sigma\left(z^{[2]}\right)$$

Why we need to use non-linear activation functions?

$$\boldsymbol{x} \longrightarrow \boxed{\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}} \longrightarrow \boxed{\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{z}^{[1]} + \boldsymbol{b}^{[2]}} \longrightarrow \boldsymbol{z}^{[2]}$$

$$\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}) + \boldsymbol{b}^{[2]}$$

$$= \boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]}\boldsymbol{x} + (\boldsymbol{W}^{[2]}\boldsymbol{b}^{[1]} + \boldsymbol{b}^{[2]})$$

$$= \boldsymbol{W}'\boldsymbol{x} + \boldsymbol{b}'$$

Where $\boldsymbol{W}' = \boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]}$ and $\boldsymbol{b}' = \boldsymbol{W}^{[2]}\boldsymbol{b}^{[1]} + \boldsymbol{b}^{[2]}$

Composition of linear functions => linear function

Activation functions

- There is no rule but … in many cases
- Output layer
  - Sigmoid
- Hidden layer
  - tanh, ReLU, LeakyReLU

Hyperbolic tangent: tanh

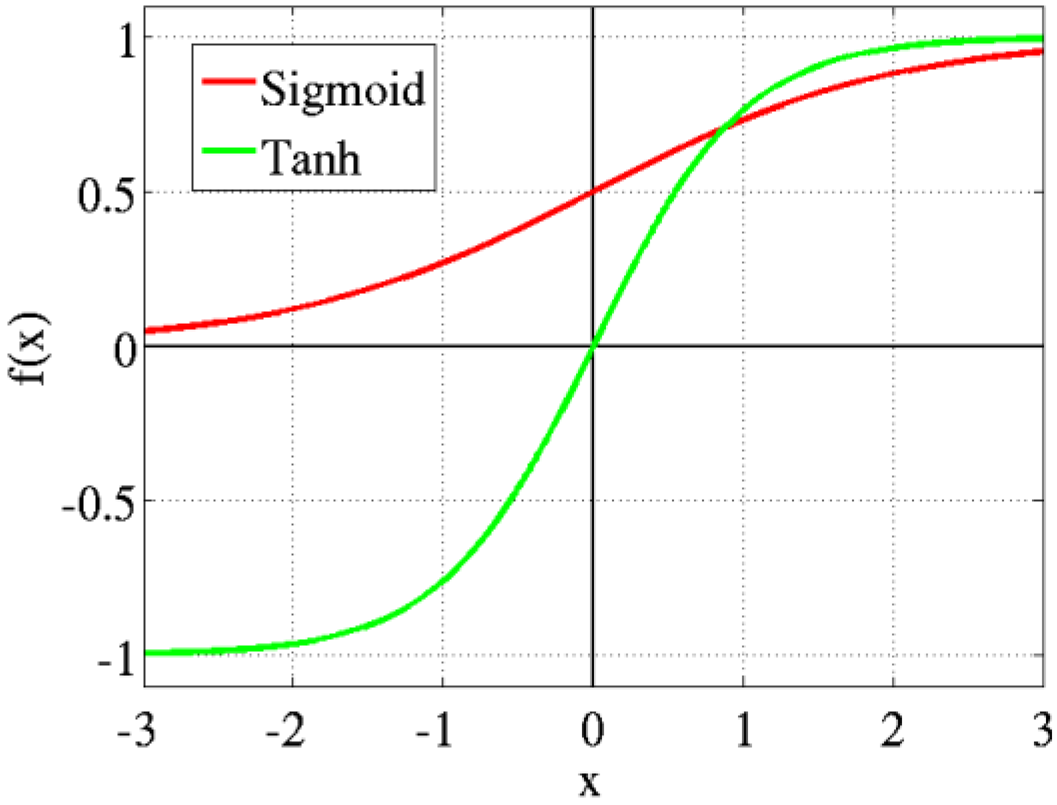
- $\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

  $= 2\sigma(2x) - 1$

- Range: $(-1,1)$
- Derivative

  $$\frac{d \tanh x}{dx} = 1 - \tanh^2 x$$

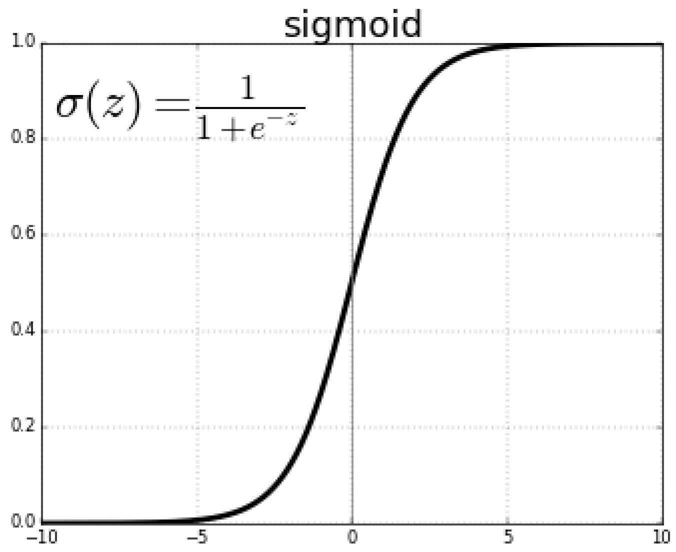
  - $0 < \frac{d \tanh x}{dx} \leq 1$

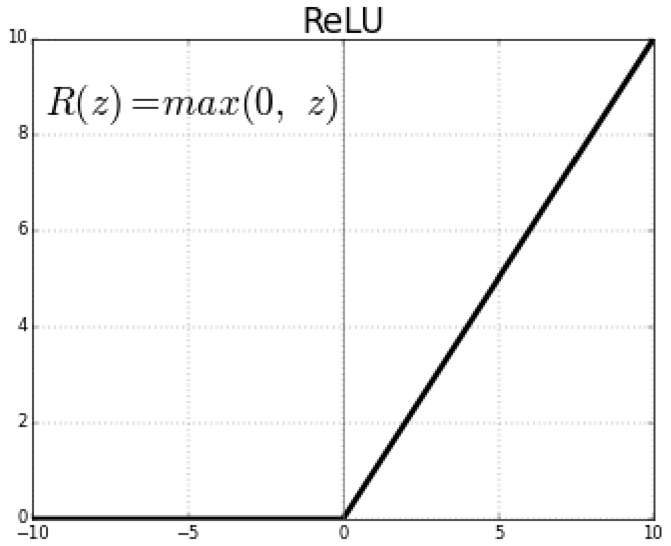


Rectified linear unit: ReLU

- $f(x) = \max\{0, x\}$
- Range: $[0, \infty)$
- Derivative

$$\frac{df(x)}{dx} = \begin{cases} 0 & if\ x < 0 \\ 1 & if x > 0 \\ undefined & if\ x = 0 \end{cases}$$

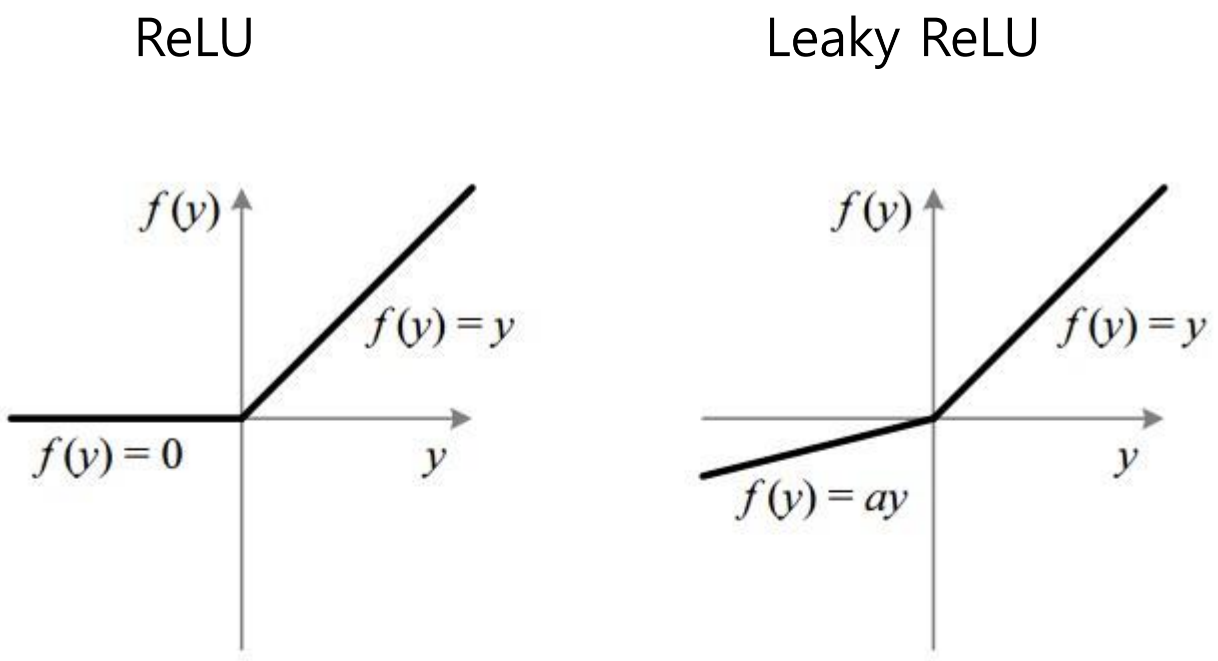




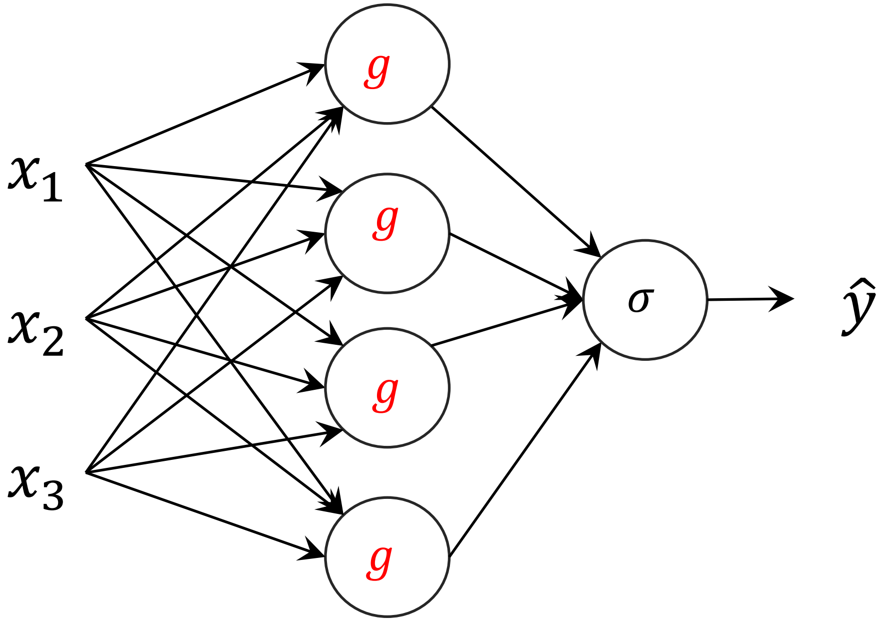
Leaky ReLU

- $f(x) = \begin{cases} ax & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$
  - $a \ll 1$ (e.g, $a = 0.01$)
- Range: $(-\infty, \infty)$
- Derivative

$$\frac{df(x)}{dx} = \begin{cases} a & if\ x < 0 \\ 1 & if x > 0 \\ undefined & if\ x = 0 \end{cases}$$



Neural Network with a Hidden Layer: Done!



$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$\boldsymbol{a}^{[1]} = g(\boldsymbol{z}^{[1]})$ where $g(.)$ is an activation function

$$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$$

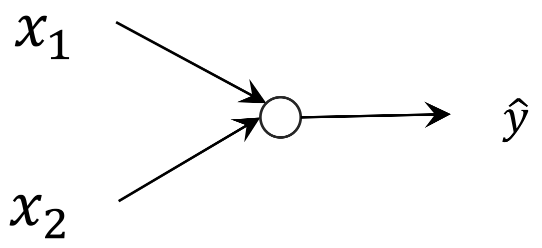
$$\hat{y} = a^{[2]} = \sigma(z^{[2]})$$

You may use tanh, ReLU, Leaky ReLU for g

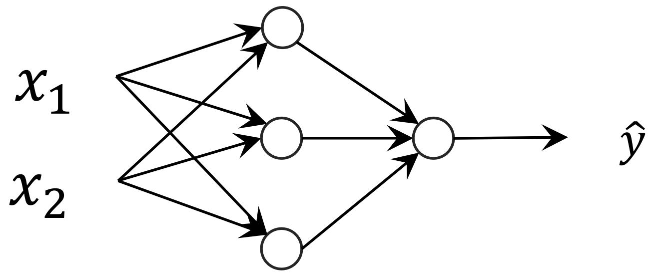
Deep Neural Networks

Multilayer Perceptrons (MLPs)

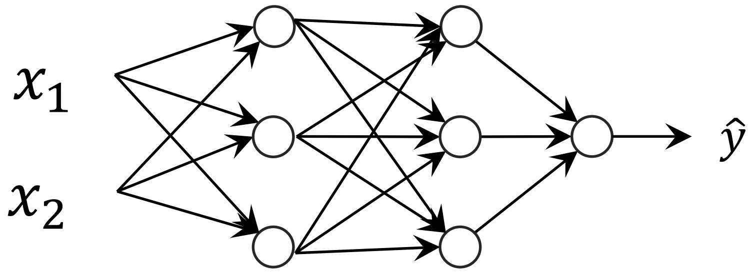
What is a deep neural network?



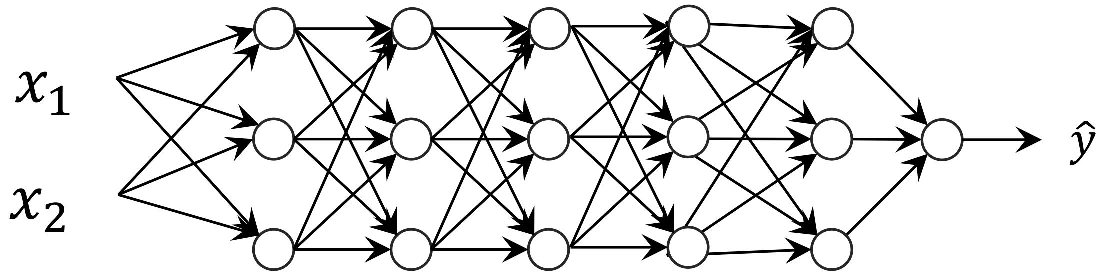
Logistic regression



1 hidden layer

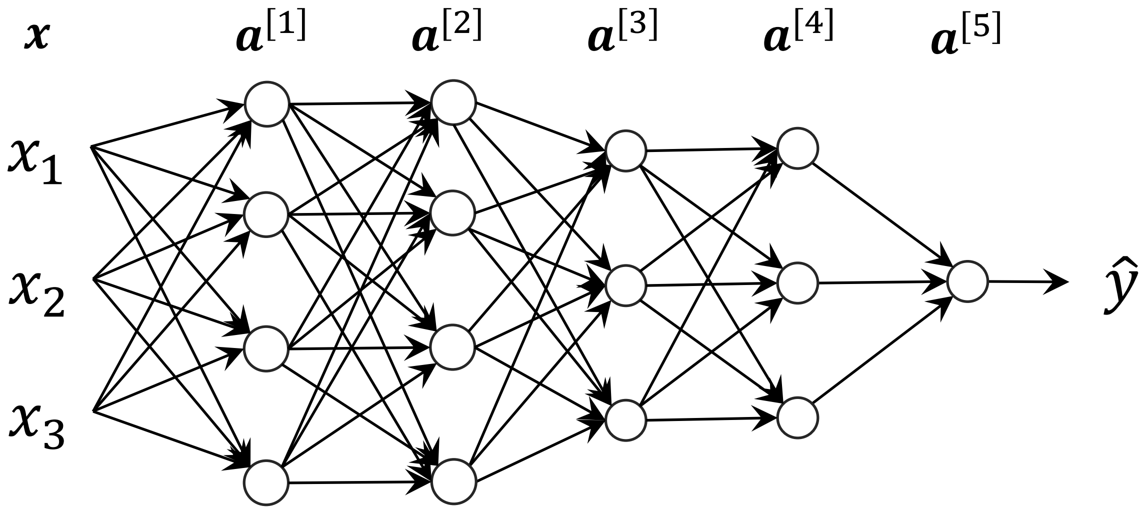


2 hidden layers



5 hidden layers

Deep neural network



$$\boldsymbol{a}^{[1]} = f\left(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}\right)$$

$$\boldsymbol{a}^{[i]} = f\left(\boldsymbol{W}^{[i]}\boldsymbol{a}^{[i-1]} + \boldsymbol{b}^{[i]}\right)$$ For $i = 2,3,4$

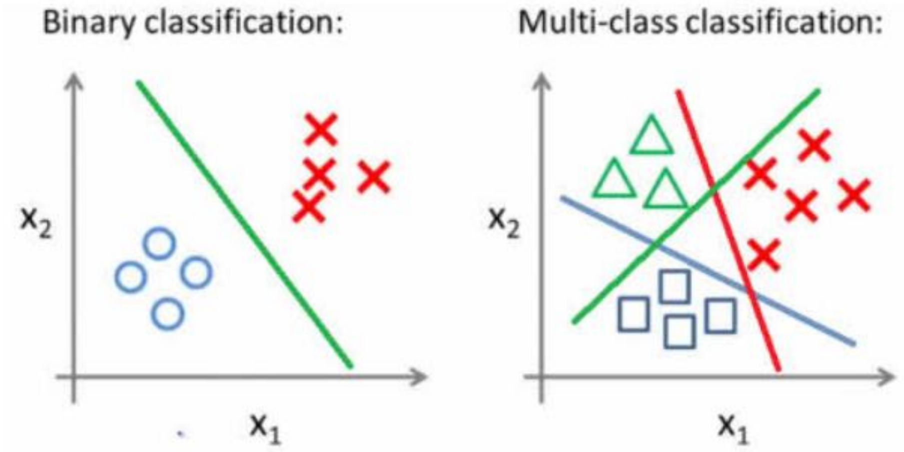
$$\hat{y} = \boldsymbol{a}^{[5]} = \sigma\left(\boldsymbol{W}^{[5]}\boldsymbol{a}^{[4]} + \boldsymbol{b}^{[5]}\right)$$

f: activation function (e.g., ReLU)

Output types

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

Output types



Amey band (2020)

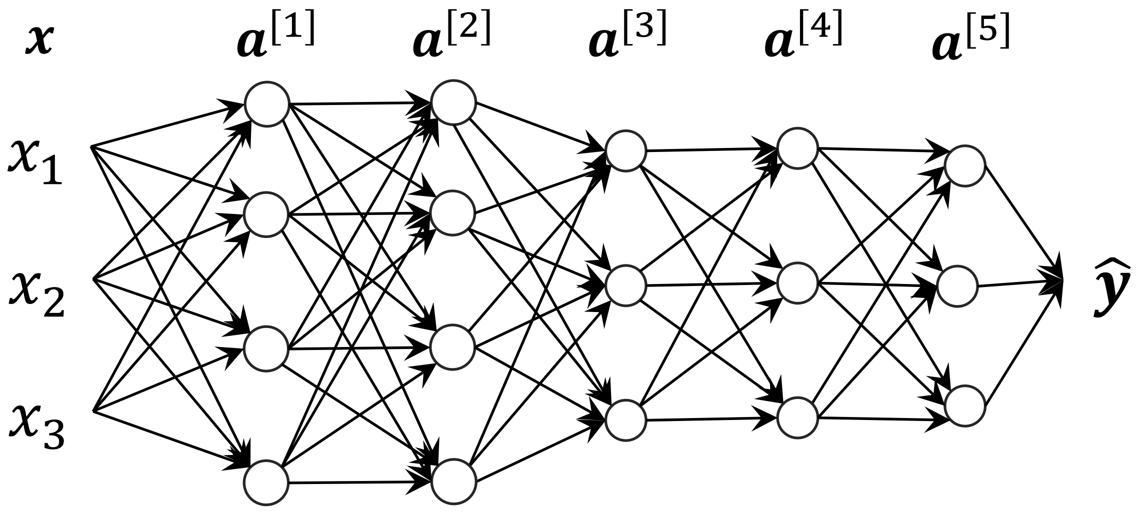
	Binary classification	(n-ary) Classification
Output classes	0/1	1/2/3/../n
Output activation function	$\sigma(x) = \frac{1}{1+e^{-x}}$	$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{i'=1}^{n} e^{x_{i'}}}$
Loss function	$-y \log \hat{y} - (1-y)\log(1-\hat{y})$	$-\sum_{i=1}^{n} \hat{y}_i \log y_i$
	Binary cross entropy	Cross entropy

Softmax outputs a categorical (multinoulli) distribution

$$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{i'=1}^{n} e^{x_{i'}}}$$

- $softmax(\boldsymbol{x})_i \geq 0$
- $\sum_{i=1}^{n} softmax(\boldsymbol{x})_i = 1$

Deep neural network



f: activation function (e.g., ReLU)

$$\boldsymbol{a}^{[1]} = f\left(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}\right)$$

$$\boldsymbol{a}^{[i]} = f\left(\boldsymbol{W}^{[i]}\boldsymbol{a}^{[i-1]} + \boldsymbol{b}^{[i]}\right)$$

$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{[5]} = softmax\left(\boldsymbol{W}^{[5]}\boldsymbol{a}^{[4]} + \boldsymbol{b}^{[5]}\right)$$

PyTorch

Modules

- PyTorch uses modules to represent neural networks
- Modules are:
  - Building blocks of computations
    - A module represents a node or a subgraph in a computation graph
  - Tightly integrated with `autograd`
    - Make it simple to specify learnable parameters
  - Easy to work with
    - Save, restore, transfer between CPU/GPU …

A simple custom module

```
import torch
from torch import nn

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.randn(out_features))

  def forward(self, input):
    return (input @ self.weight) + self.bias
```

Call forward

```
m = MyLinear(4, 3)
sample_input = torch.randn(4)
m(sample_input)
: tensor([-0.3037, -1.0413, -4.2057], grad_fn=<AddBackward0>)
```

Modules as Building Blocks

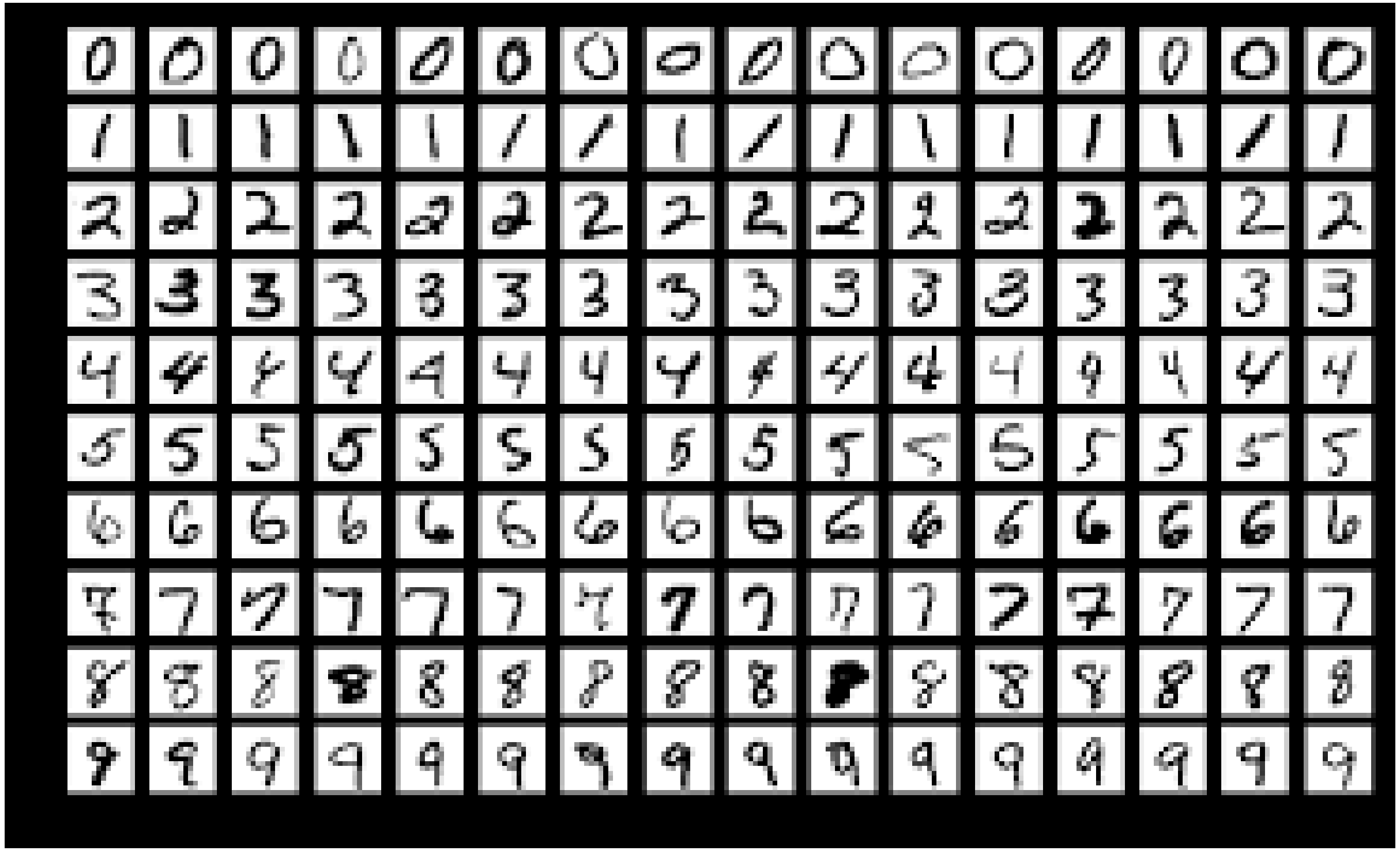
```
import torch.nn.functional as F

class Net(nn.Module):
  def __init__(self):
    super().__init__()
    self.l0 = nn.Linear(4,3)
    self.l1 = nn.Linear(3,1)
  def forward(self, x):
    x = self.l0(x)
    x = F.relu(x)
    x = self.l1(x)
    return x
```

Training with Modules

- Initialize
  - model = YourModel()
  - optimizer = torch.optim.SGD(model.parameters(), lr = <learning_rate>
- Forward
  - y_hat = model(input)
- Backward
  - loss = compute_loss(y, y_hat) //You need to implement compute_loss
  - model.zero_grad()
  - loss.backward()
  - optimizer.step()

Implementing a DNN with PyTorch

MNIST



Data loading

- Import torchvision
- From torchvision import datasets

```
batch_size = 12

train_data = datasets.MNIST('D:₩datasets', train=True, download=True, transform=transforms.ToTensor())
test_data = datasets.MNIST('D:₩datasets', train=False, download=True,transform=transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size)
```

Model

```python
class MLP(nn.Module):
    def __init__(self):
        super().__init__()

        self.in_dim = 28*28 # MNIST
        self.out_dim = 10

        self.fc1 = nn.Linear(self.in_dim, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, self.out_dim)

        self.relu = nn.ReLU()
        self.log_softmax = nn.LogSoftmax()

    def forward(self, x):
        a1 = self.relu(self.fc1(x.view(-1, self.in_dim)))
        a2 = self.relu(self.fc2(a1))
        a3 = self.relu(self.fc3(a2))
        a4 = self.relu(self.fc4(a3))
        logit = self.fc5(a4)
        return logit
```

Train

```python
model =  MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

```python
for epoch in range(10):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if (i+1) % 2000 == 0:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.209
[1,  4000] loss: 0.739
[2,  2000] loss: 0.316
[2,  4000] loss: 0.230
[3,  2000] loss: 0.154
[3,  4000] loss: 0.144
[4,  2000] loss: 0.112
[4,  4000] loss: 0.101
[5,  2000] loss: 0.075
[5,  4000] loss: 0.082
[6,  2000] loss: 0.061
[6,  4000] loss: 0.063
[7,  2000] loss: 0.046
[7,  4000] loss: 0.054
[8,  2000] loss: 0.033
[8,  4000] loss: 0.041
[9,  2000] loss: 0.029
[9,  4000] loss: 0.033
[10,  2000] loss: 0.021
[10,  4000] loss: 0.025
Finished Training
```

Test

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

```python
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```python
dataiter = iter(test_loader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images, nrow = batch_size))
print('GroundTruth')
print("  "+' '.join('%3s' % label.item() for label in labels))

outputs = model(images)
_, predicted = torch.max(outputs, 1)
print("Prediction")
print("  "+' '.join('%3s' % label.item() for label in predicted))
```


```
GroundTruth
    7   2   1   0   4   1   4   9   5   9   0   6
Prediction
    7   2   1   0   4   1   4   9   6   9   0   6
```

Test

```python
n_predict = 0
n_correct = 0

for data in test_loader:
    inputs, labels = data
    outputs = model(inputs)
    _, predicted = torch.max(outputs, 1)

    n_predict += len(predicted)
    n_correct += (labels == predicted).sum()


print(f"{n_correct}/{n_predict}")
print(f"Accuracy: {n_correct/n_predict:.3f}")
```

```
9761/10000
Accuracy: 0.976
```