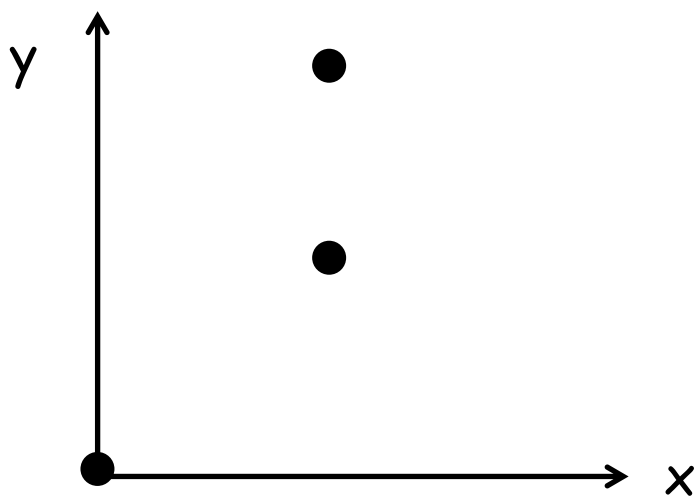
Exercise 1

Goal of Machine Learning

- 다음 데이터를 가장 잘 설명하는 함수를 찾아라

$$(0.0, 0.0)\ (1.0, 1.0)\ (1.0, 2.0)$$



$$f(x; w_0, w_1) = w_1 x + w_0$$

$$Error = \sum_{(\mathbf{x}, y) \in Data} \left(y - f(\mathbf{x}; w_1, w_2, \dots, w_m)\right)^2$$

Gradient Descent Method

- **Steps**

learning rate

Randomly choose an initial solution, w^0

Repeat

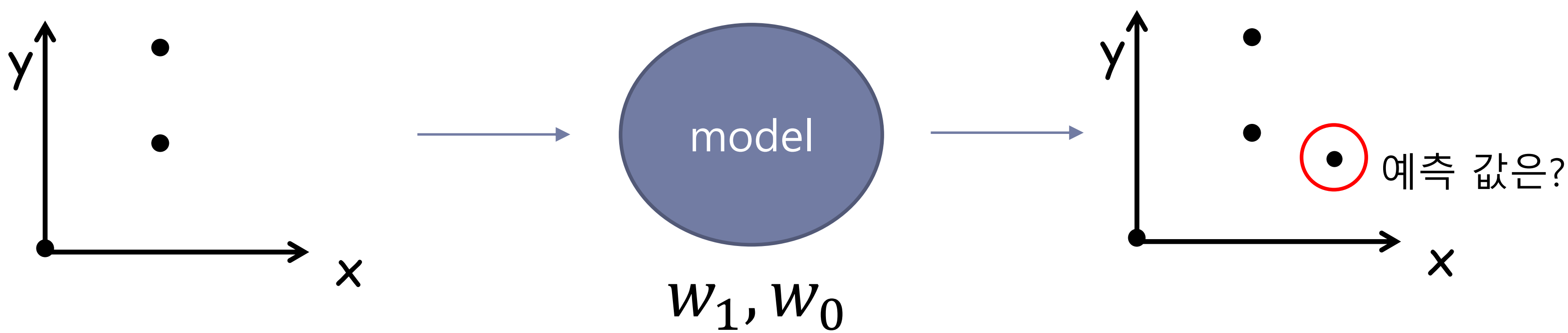
$$w^{t+1} = w^t - \eta \left.\frac{dE}{dw}\right|_{w=w^t}$$

Until stopping condition is satisfied

- $|w^{t+1} - w^t|$ is very small
- f(w) little moves
- fixed number of iterations

Coding 전에 생각해 볼 것

- 입력 데이터 : (0.0), (1.0), (1.0)
- 출력 데이터 : (0.0), (1.0), (2.0)
- Optimizer: gradient descent method
- Loss function: Mean square error



준비 단계 1

- 입력 데이터 : (0.0), (1.0), (1.0)
- 출력 데이터 : (0.0), (1.0), (2.0)

```
x_train = torch.FloatTensor([[0], [1], [1]])
y_train = torch.FloatTensor([[0], [1], [2]])
```

- Model: $f(X; W_0, W_1) = W_1 * X + W_0$

```
hypothesis = x_train * w + b
```

- Loss function : Error $= \sum (y-f(X; W1, W2,...., , Wm))^2$

```
cost = torch.mean( (hypothesis - y_train)**2)
```

optimizer

- **Optimizer : SGD**

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	Implements lazy version of Adam algorithm suitable for sparse tensors.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm, heavily inspired by minFunc.
`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

https://pytorch.org/docs/stable/optim.html?highlight=optimizer#torch.optim.Optimizer

optimizer

Optimizer : SGD

Docs > torch.optim > SGD

SGD

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Exercise 1

```python
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

x_train = torch.FloatTensor([[0], [1], [1]])
y_train = torch.FloatTensor([[0], [1], [2]])

w = torch.zeros(1, requires_grad = True)
b = torch.zeros(1, requires_grad = True)

optimizer = optim.SGD([w, b] , lr = 0.01)

nb_epochs = 1000

for epoch in range( nb_epochs + 1):

    hypothesis = x_train * w + b

    cost = torch.mean( (hypothesis - y_train)**2)

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    if epoch % 100 == 0 :
        print( 'Epoch {:5d}/{} w:{:.3f} b:{:.3f} cost: {:.3f}'
              .format(epoch, nb_epochs, w.item(), b.item(), cost.item() ))
```

$$f(x; w_0, w_1) = w_1 x + w_0$$

Exercise 1

```
Epoch    0/3000 w:0.020 b:0.020 cost: 1.667
Epoch  100/3000 w:0.782 b:0.509 cost: 0.283
Epoch  200/3000 w:0.981 b:0.403 cost: 0.230
Epoch  300/3000 w:1.114 b:0.301 cost: 0.202
Epoch  400/3000 w:1.212 b:0.225 cost: 0.186
Epoch  500/3000 w:1.285 b:0.168 cost: 0.178
Epoch  600/3000 w:1.340 b:0.125 cost: 0.173
Epoch  700/3000 w:1.380 b:0.093 cost: 0.170
Epoch  800/3000 w:1.411 b:0.070 cost: 0.169
Epoch  900/3000 w:1.433 b:0.052 cost: 0.168
Epoch 1000/3000 w:1.450 b:0.039 cost: 0.167
Epoch 1100/3000 w:1.463 b:0.029 cost: 0.167
Epoch 1200/3000 w:1.472 b:0.022 cost: 0.167
Epoch 1300/3000 w:1.479 b:0.016 cost: 0.167
Epoch 1400/3000 w:1.485 b:0.012 cost: 0.167
Epoch 1500/3000 w:1.488 b:0.009 cost: 0.167
Epoch 1600/3000 w:1.491 b:0.007 cost: 0.167
Epoch 1700/3000 w:1.494 b:0.005 cost: 0.167
Epoch 1800/3000 w:1.495 b:0.004 cost: 0.167
Epoch 1900/3000 w:1.496 b:0.003 cost: 0.167
Epoch 2000/3000 w:1.497 b:0.002 cost: 0.167
Epoch 2100/3000 w:1.498 b:0.002 cost: 0.167
Epoch 2200/3000 w:1.499 b:0.001 cost: 0.167
Epoch 2300/3000 w:1.499 b:0.001 cost: 0.167
Epoch 2400/3000 w:1.499 b:0.001 cost: 0.167
Epoch 2500/3000 w:1.499 b:0.000 cost: 0.167
Epoch 2600/3000 w:1.500 b:0.000 cost: 0.167
Epoch 2700/3000 w:1.500 b:0.000 cost: 0.167
Epoch 2800/3000 w:1.500 b:0.000 cost: 0.167
Epoch 2900/3000 w:1.500 b:0.000 cost: 0.167
Epoch 3000/3000 w:1.500 b:0.000 cost: 0.167
```

Quiz 1

- **Data : (x1, x2, x3) (Y)**

```
x_train = torch.FloatTensor([[73,80,75], [93,88,93], [89, 91, 90],[96,98,100],[73,66,70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

- **hint**
  - **[a1, a2, a3]*[w1, w2, w3] =?**
  - **MSELOSS ?**

Docs > torch.nn > MSELoss

MSELOSS

CLASS `torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

Examples:

```
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

Answer 1

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

x_train = torch.FloatTensor([[73,80,75], [93,88,93], [89, 91, 90],[96,98,100],[73,66,70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])

w = torch.zeros((3, 1), requires_grad = True)
b = torch.zeros(1, requires_grad = True)

optimizer = optim.SGD([w, b] , lr = 1e-5)

nb_epochs = 1000

for epoch in range( nb_epochs + 1):

    #hypothesis = x_train * w + b
    hypothesis = x_train.matmul(w) + b

    #cost = torch.mean( (hypothesis - y_train)**2)
    cost = F.mse_loss(hypothesis, y_train)

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    if epoch % 100 == 0 :
        print( 'Epoch {:5d}/{} cost: {:.3f}'
              .format(epoch, nb_epochs, cost.item() ))
        print( w.squeeze() )
```

```
loss = nn.MSELoss()

nb_epochs = 1000

for epoch in range( nb_epochs + 1):

    #hypothesis = x_train * w + b
    hypothesis = x_train.matmul(w) + b

    #cost = torch.mean( (hypothesis - y_train)**2)
    #cost = F.mse_loss(hypothesis, y_train)
    cost = loss(hypothesis, y_train)
```

MSE Loss

- Torch.nn.function : 함수, torch.nn : 클래스

SOURCE CODE FOR TORCH.NN.MODULES.LOSS

```
import warnings

from .distance import PairwiseDistance
from .module import Module
from .. import functional as F
from .. import _reduction as _Reduction
```

```
class MSELoss(_Loss):
    r"""Creates a criterion that measures the mean squared error (squared L2 norm) between
    each element in the input :math:`x` and target :math:`y`.
```

```
    __constants__ = ['reduction']

    def __init__(self, size_average=None, reduce=None, reduction: str = 'mean') -> None:
        super(MSELoss, self).__init__(size_average, reduce, reduction)

    def forward(self, input: Tensor, target: Tensor) -> Tensor:
        return F.mse_loss(input, target, reduction=self.reduction)
```

SOURCE CODE FOR TORCH.NN.FUNCTIONAL

```
r"""Functional interface"""
from typing import Callable, List, Optional, Tuple
import math
import warnings
```

```
def mse_loss(
    input: Tensor,
    target: Tensor,
    size_average: Optional[bool] = None,
    reduce: Optional[bool] = None,
    reduction: str = "mean",
) -> Tensor:
    r"""mse_loss(input, target, size_average=None, reduce=None, reduction='mean') -> Tensor

    Measures the element-wise mean squared error.

    See :class:`~torch.nn.MSELoss` for details.
    """
    if has_torch_function_variadic(input, target):
        return handle_torch_function(
            mse_loss, (input, target), input, target, size_average=size_average, reduce=reduce,
reduction=reduction
        )
    if not (target.size() == input.size()):
        warnings.warn(
            "Using a target size ({}) that is different to the input size ({}). "
            "This will likely lead to incorrect results due to broadcasting. "
            "Please ensure they have the same size.".format(target.size(), input.size()),
            stacklevel=2,
        )
    if size_average is not None or reduce is not None:
        reduction = _Reduction.legacy_get_string(size_average, reduce)

    expanded_input, expanded_target = torch.broadcast_tensors(input, target)
    return torch._C._nn.mse_loss(expanded_input, expanded_target,
_Reduction.get_enum(reduction))
```

성균관대학교

Tip

Data Preprocessing

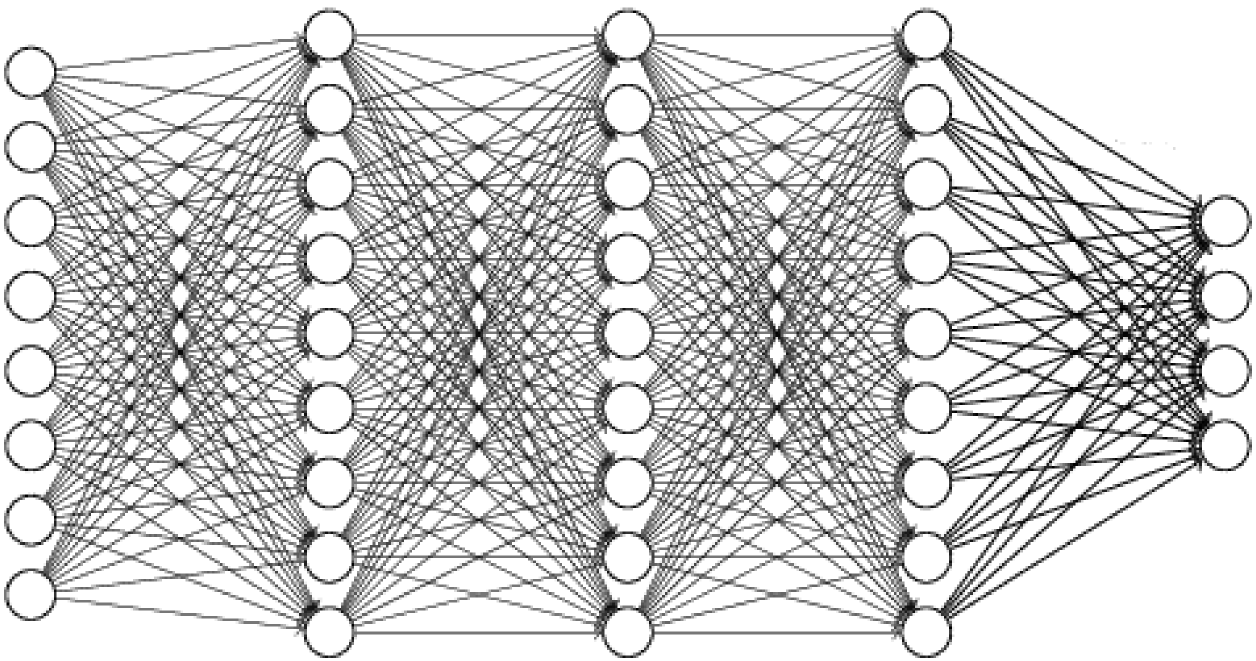
```
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])

mu = x_train.mean(dim=0)
sigma = x_train.std(dim=0)
norm_x_train = (x_train - mu) / sigma
print(norm_x_train)
```

PyTorch Module

- **How can we make Deep Neural Network?**

```
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```



PyTorch Module

nn.Module (1)

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


class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1)

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

PyTorch Module

nn.Module (2)

```
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
# 모델 초기화
model = MultivariateLinearRegressionModel()
# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1e-5)
```

PyTorch Module

nn.Module (3)

```
nb_epochs = 20
for epoch in range(nb_epochs + 1):
    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    print('Epoch {:4d}/{} Cost: {:.6f}'.format(
        epoch, nb_epochs, cost.item()
    ))
```

Pytorch

GPU

```
import torch
import numpy as np

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

print(device)


#net = Model.to(device)
#inputs = data.to(device)
```

Question and Answer