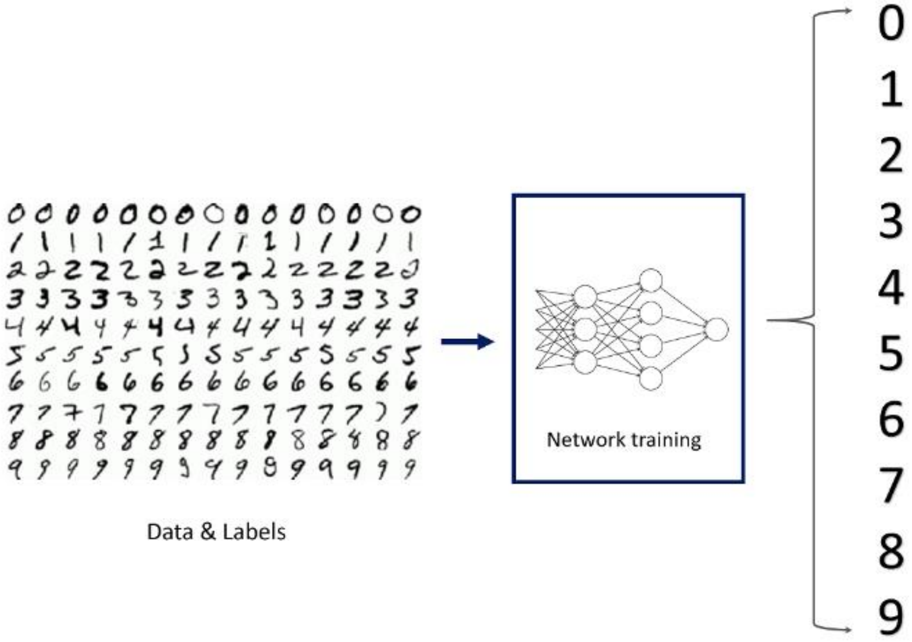
04. Multi-layer perceptrons

AILAB

Hanyang Univ.

오늘 실습 내용

- **Softmax Function**
- **학습 관련 개념**
- **MNIST data 분석**
- **Multi-layer perceptron으로 MNIST data classifier model 만들기**

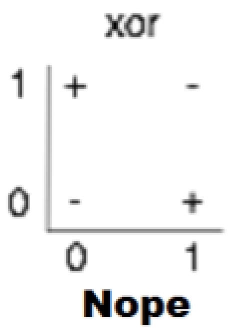


MNIST Dataset and Number Classification [1]

1. Softmax function

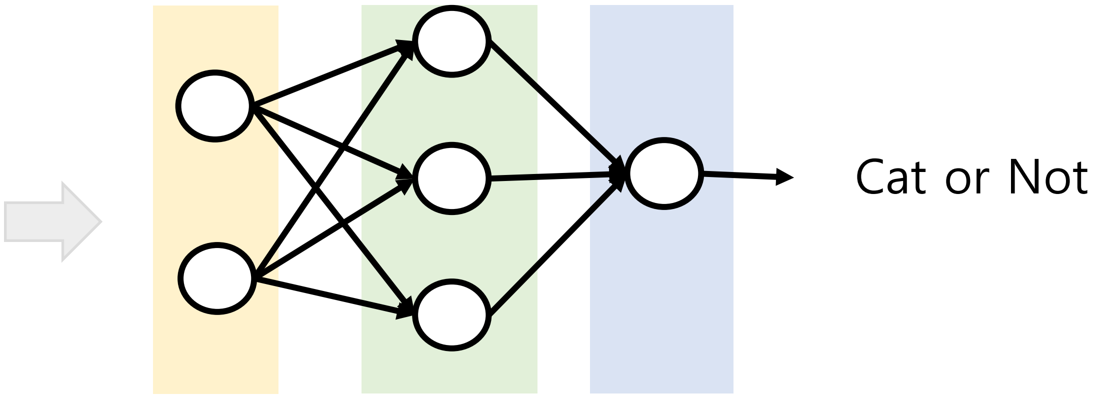
Softmax function

- Single Class classification



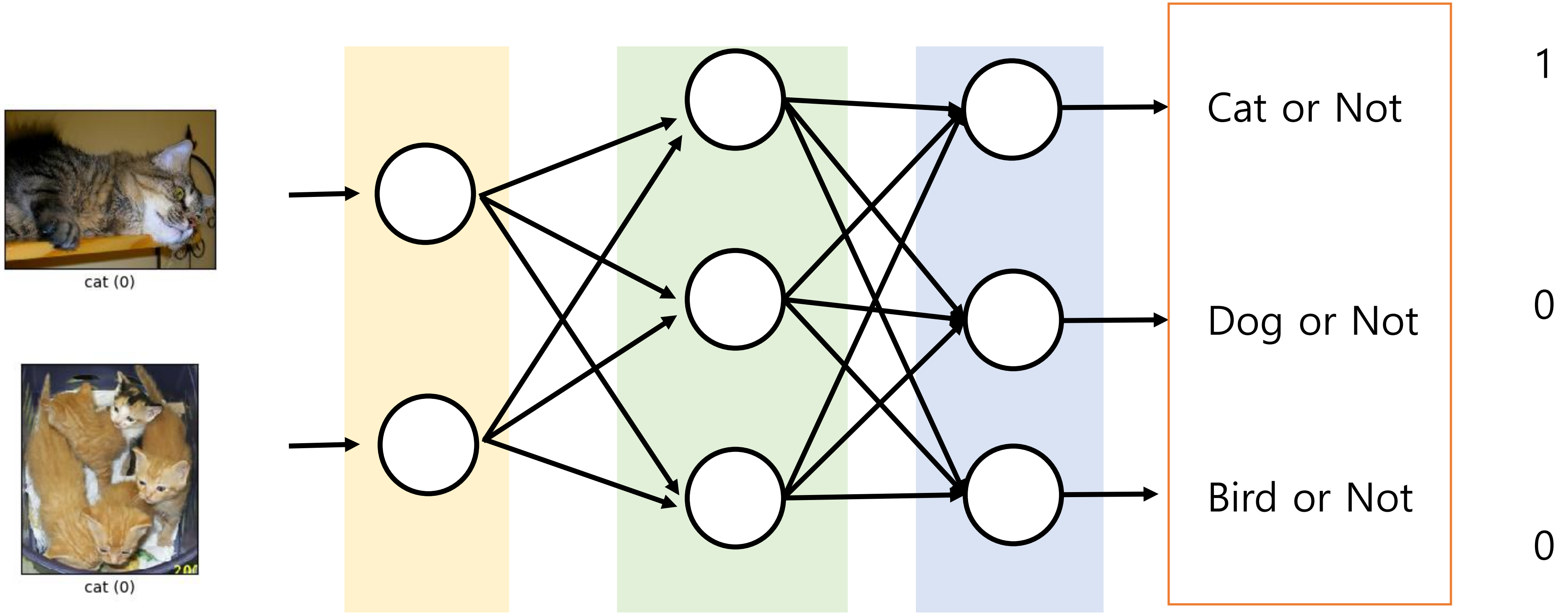
cat (0)

cat (0)



Softmax function

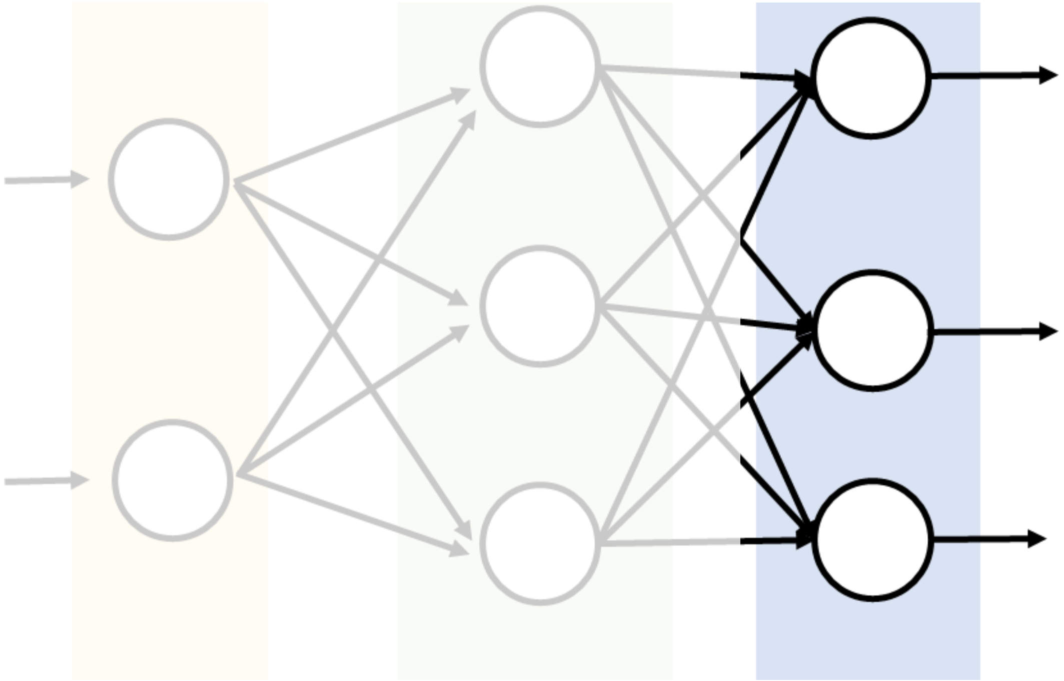
- Multiple Class classification
  - 확률로 data에 대한 class를 예측한다.



Output layer 노드 중 Input data의 class를 확인하는 노드의 아웃풋 **확률이 1에 가깝게 학습**하는 것이 목표

Softmax function

- **Multiple Class** classification
  - Neural net output이 각 class의 확률 → 합이 **1**이어야 한다.



Linear Layer (y=Wx+b) 는 0~1사이의 값을 가지지 않음

Binary Class → Sigmoid function

Multi-Class → Softmax functoin

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

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for j = 1,...,K}$$

Softmax function

$$cost = \frac{1}{m}\sum_{i=1}^{m} L(H(x^{(i)}), y^{(i)})$$

Logistic loss :
(binary)

$$L(H(x), y) = ylog(H(x)) - (1-y)log(1-H(x))$$

→ $H(x)$: sigmoid output

Cross-entropy Loss :
(multi-class)

$$L(H(x), y) = -\sum_{j=1}^{C} y_j \log(H(x_j))$$

→ $H(x_j)$: softmax output at j node

→ y : one-hot vector

ex) Label 0,1,2: 2 → [0 0 1] $y_2 = 1$

H(x) →[0.3 0.1 0.6]

Softmax function

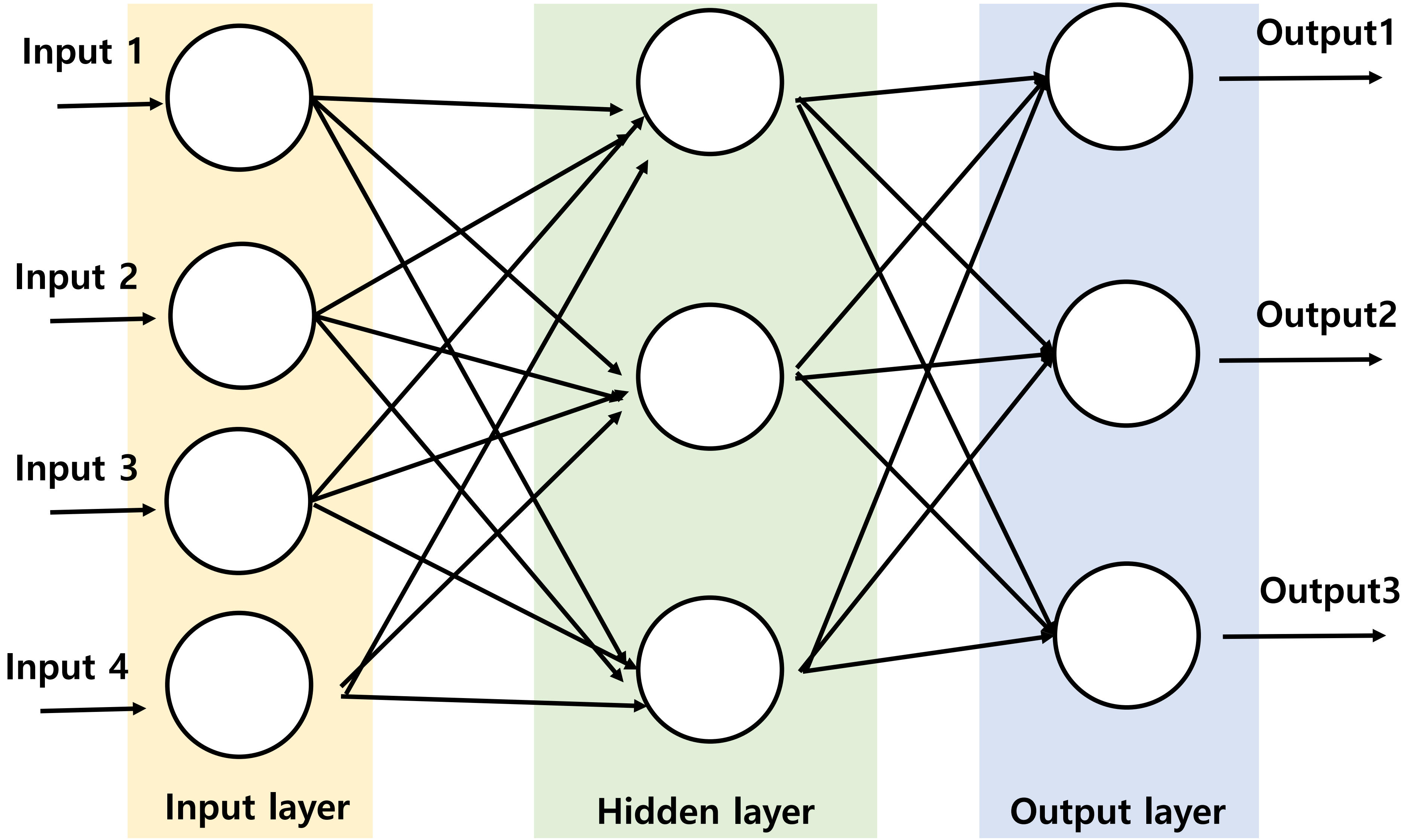
- torch.nn.CrossEntropyLoss()
  - https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
  - Log_Softmax + NLLLoss
    - nn.LogSoftmax() + ~~nn.NLLLoss()~~ = nn.CrossEntropyLoss()

$$Cost\big(H(x^{(i)}, y^{(i)}\big) = \frac{1}{m}\sum_{i=1}^{m} -\sum_{j=1}^{C} y_j^i \log\big(H\big(x_j^i\big)\big)$$

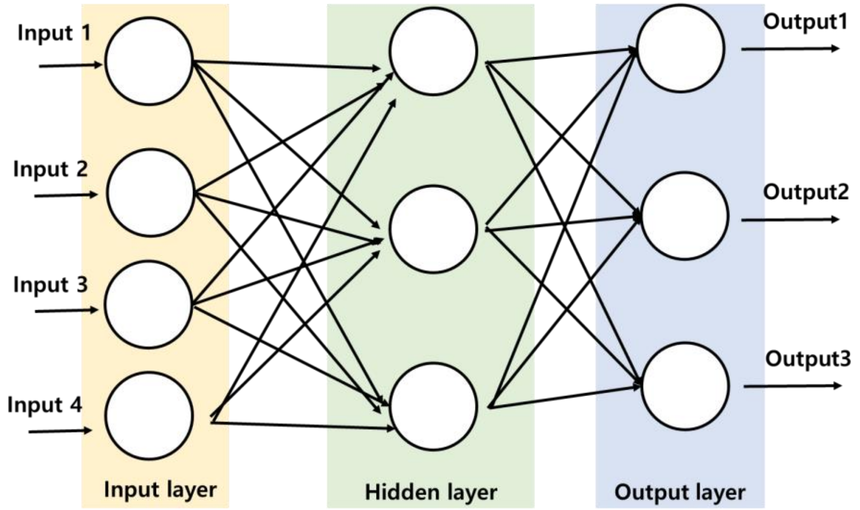
```
log_probs = nn.LogSoftmax(dim=1)(logits)
cost = nn.NLLLoss()(log_probs, y_train) # get cost
```

```
cost = nn.CrossEntropyLoss()(logits, y_train) # get cost
```

Softmax function



Softmax function

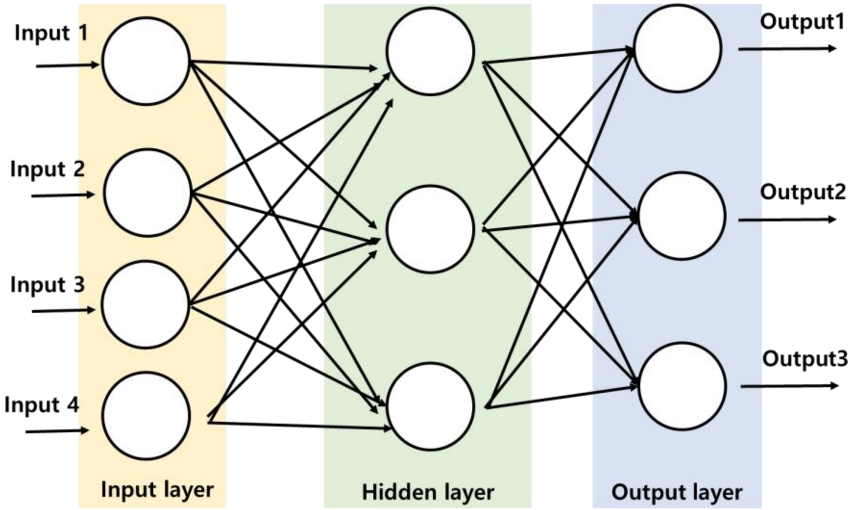



```
x_train = torch.FloatTensor([[1,2,1,1],
                             [2,1,3,2],
                             [3,1,3,4],
                             [4,1,5,5],
                             [1,7,5,5],
                             [1,2,5,6],
                             [1,6,6,6],
                             [1,7,7,7]])
y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0])
```

Why Long?

$y_j \log\left(H\left(x_j\right)\right)$: x_j를 y_j로 인덱싱

Softmax function




```
[3]  class MultiLayerPerceptron(nn.Module):
       def __init__(self):
         super(MultiLayerPerceptron, self).__init__()
         self.linear1 = nn.Linear(4, 3)
         self.activation = nn.Sigmoid()

         self.linear2 = nn.Linear(3, 3)

       def forward(self, x):
         z1 = self.linear1(x)
         a1 = self.activation(z1)

         z2 = self.linear2(a1)

         return z2
```

Softmax function

```
[8] for epoch in range(epochs):
      logits = model(x_train)  # forward propagation

      cost = nn.CrossEntropyLoss()(logits, y_train) # get cost
      optimizer.zero_grad()
      cost.backward() # backward propagation
      optimizer.step() # update parameters
```

```
[1] import torch
    import torch.nn as nn
    import torch.optim as optim
```

```
[2] x_train = torch.FloatTensor([[1,2,1,1],
                                 [2,1,3,2],
                                 [3,1,3,4],
                                 [4,1,5,5],
                                 [1,7,5,5],
                                 [1,2,5,6],
                                 [1,6,6,6],
                                 [1,7,7,7]])
    y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0])
```

```
[3] class MultiLayerPerceptron(nn.Module):
      def __init__(self):
        super(MultiLayerPerceptron, self).__init__()
        self.linear1 = nn.Linear(4, 3)
        self.activation = nn.Sigmoid()

        self.linear2 = nn.Linear(3, 3)

      def forward(self, x):
        z1 = self.linear1(x)
        a1 = self.activation(z1)

        z2 = self.linear2(a1)

        return z2
```

```
[4] model = MultiLayerPerceptron().train()
```

```
[5] optimizer = optim.SGD(model.parameters(), lr=1) # set optimizer
```

```
[6] epochs = 8000
    model.train()
    for epoch in range(epochs):
      logits = model(x_train)  # forward propagation

      log_probs = nn.LogSoftmax(dim=1)(logits)
      cost = nn.NLLLoss()(log_probs, y_train) # get cost
      optimizer.zero_grad()
      cost.backward() # backward propagation
      optimizer.step() # update parameters
```

Softmax function

```
model.eval()
with torch.no_grad():
  logits = model(x_train)
probs = nn.Softmax(dim=1)(logits)

print('logit\n : {}'.format(logits))
print('predict with softmax\n : {}'.format(probs))
print('predict with argmax\n : {}'.format(torch.argmax(probs,dim=1)))
```

```
logit
 : tensor([[ -9.9276,   0.1953,   8.4101],
        [-11.2227,   1.1776,   8.4944],
        [-11.0068,   1.1157,   8.3120],
        [ -4.2295,   4.4231,  -2.6406],
        [ -1.4168,   5.0240,  -4.0339],
        [ -2.3974,   4.3527,  -4.6213],
        [  8.0901,   1.5891, -11.3227],
        [  8.2324,   1.5415, -11.4260]])
predict with softmax
 : tensor([[1.0861e-08, 2.7052e-04, 9.9973e-01],
        [2.7333e-09, 6.6387e-04, 9.9934e-01],
        [4.0701e-09, 7.4875e-04, 9.9925e-01],
        [1.7448e-04, 9.9897e-01, 8.5468e-04],
        [1.5923e-03, 9.9829e-01, 1.1627e-04],
        [1.1692e-03, 9.9870e-01, 1.2649e-04],
        [9.9850e-01, 1.4997e-03, 3.7023e-09],
        [9.9876e-01, 1.2406e-03, 2.8968e-09]])
predict with argmax
 : tensor([2, 2, 2, 1, 1, 1, 0, 0])
```

Logit
: 주로 마지막 activation function의 input 지칭

Argmax로 확률이 높은 곳의 index 추출

```
x_train = torch.FloatTensor([[1,2,1,1],
                             [2,1,3,2],
                             [3,1,3,4],
                             [4,1,5,5],
                             [1,7,5,5],
                             [1,2,5,6],
                             [1,6,6,6],
                             [1,7,7,7]])
y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0])
```

Softmax function

```
model.eval()
with torch.no_grad():
  logits = model(x_train)
probs = nn.Softmax(dim=1)(logits)

print('logit₩n : {}'.format(logits))
print('predict with softmax₩n : {}'.format(probs))
print('predict with argmax₩n : {}'.format(torch.argmax(probs,dim=1)))
```

```
logit
 : tensor([[ -9.9276,   0.1953,   8.4101],
        [-11.2227,   1.1776,   8.4944],
        [-11.0068,   1.1157,   8.3120],
        [ -4.2295,   4.4231,  -2.6406],
        [ -1.4168,   5.0240,  -4.0339],
        [ -2.3974,   4.3527,  -4.6213],
        [  8.0901,   1.5891, -11.3227],
        [  8.2324,   1.5415, -11.4260]])
predict with softmax
 : tensor([[1.0861e-08, 2.7052e-04, 9.9973e-01],
        [2.7333e-09, 6.6387e-04, 9.9934e-01],
        [4.0701e-09, 7.4875e-04, 9.9925e-01],
        [1.7448e-04, 9.9897e-01, 8.5468e-04],
        [1.5923e-03, 9.9829e-01, 1.1627e-04],
        [1.1692e-03, 9.9870e-01, 1.2649e-04],
        [9.9850e-01, 1.4997e-03, 3.7023e-09],
        [9.9876e-01, 1.2406e-03, 2.8968e-09]])
predict with argmax
 : tensor([2, 2, 2, 1, 1, 1, 0, 0])
```

Logit
: 주로 마지막 activation function의
input 지칭

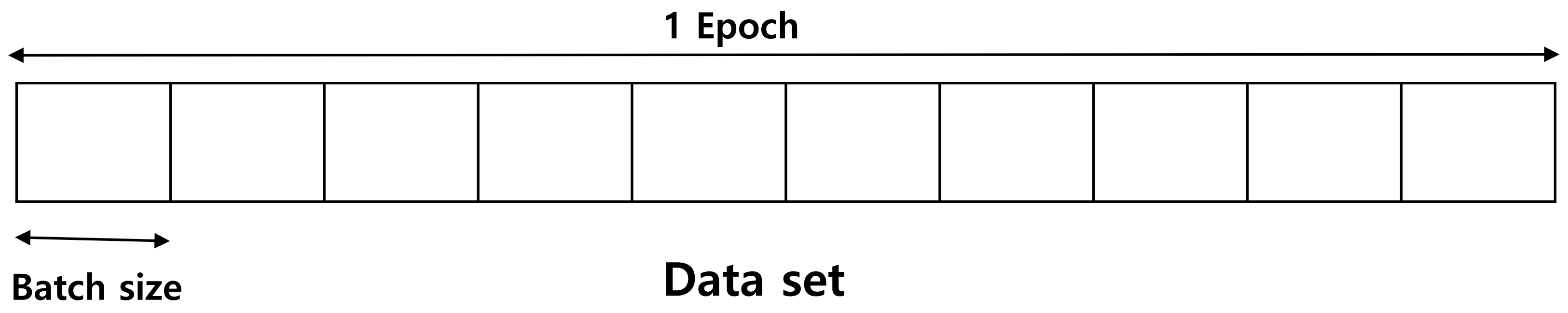
Argmax로 확률이 높은 곳의 index 추출

```
x_train = torch.FloatTensor([[1,2,1,1],
                             [2,1,3,2],
                             [3,1,3,4],
                             [4,1,5,5],
                             [1,7,5,5],
                             [1,2,5,6],
                             [1,6,6,6],
                             [1,7,7,7]])
y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0])
```

2. 학습 관련 개념

학습 관련 개념

- **Epoch : 전체 Sample 데이터를 학습하는것**
- **Step : 1 step당 weight와 Bias를 1회씩 업데이트 하게됨**
- **Batch Size : 1 Step에서 사용한 데이터의 수**
- **Learning rate : 경사 하강법에서 학습 단계별로 움직이는 학습 속도**
- Ex) Batch Size 가 100, Step이 10이면 약 1000개의 데이터를 이용



학습 관련 개념

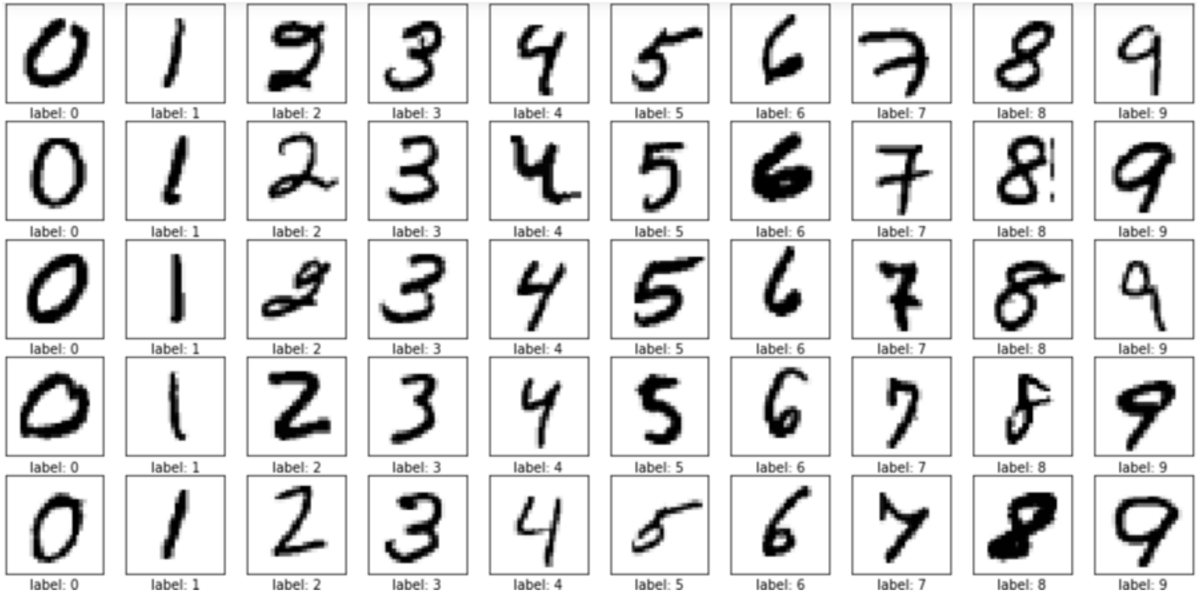
```
optimizer = optim.SGD(model.parameters(), lr=0.1) # set optimizer

criterion = nn.BCELoss()

epochs = 8000
for epoch in range(epochs):
  model.train()
  hypothesis = model(x_train)  # forward propagation
  cost = criterion(hypothesis+1e-8, y_train) # get cost
  optimizer.zero_grad()
  cost.backward() # backward propagation
  optimizer.step() # update parameters
```

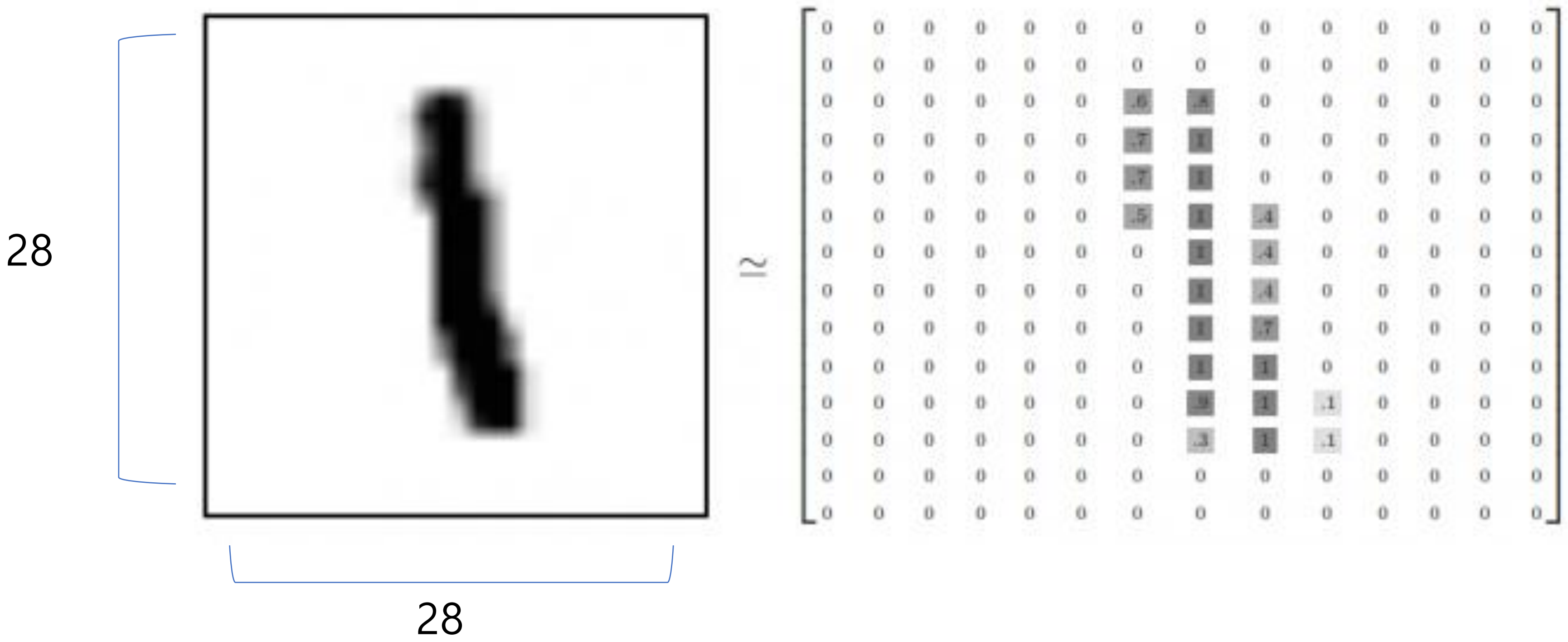
3. MNIST Data

MNIST Data



- 필기체 숫자의 분류를 위한 학습 데이터 셋
- **이미지**(x)와 이미지에 해당하는 **라벨**(y)로 구성

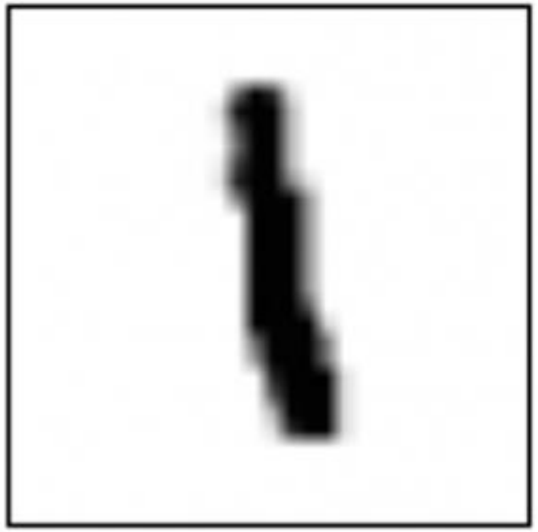
MNIST Data



이미지 : **784차원**의 벡터 ex) [0, 0, 0, 0, ...7, 1, 0, 0, 0, ...]

라벨 : **0 ~ 9**

MNIST Data



$$
\begin{array}{cc}
0 & \\
1 & \\
2 & \\
3 & \\
4 & \\
5 & \\
6 & \\
7 & \\
8 & \\
9 &
\end{array}
\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
$$

MNIST Data

MNIST data 구조

<table>
<tr><td colspan="2">Training data (55,000개)
mnist.train</td><td colspan="2">Test data (10,000개)
mnist.test</td><td colspan="2">Validation data (5,000개)
mnist.validation</td></tr>
<tr><td>images</td><td>labels</td><td>images</td><td>labels</td><td>images</td><td>labels</td></tr>
</table>

4. MNIST data classifier

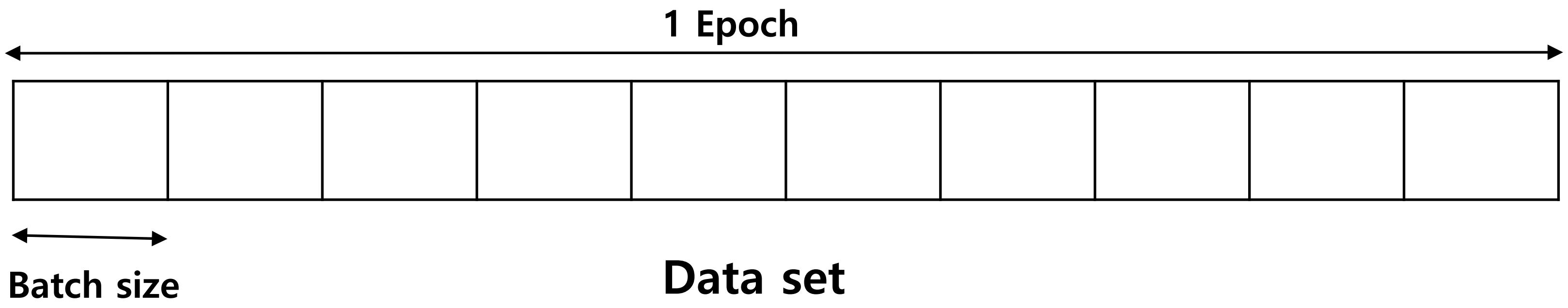
신경망모델 학습 프로세스

- 데이터 processing
- model 디자인
  - layer 종류, 개수 및 뉴런 개수 설정
  - 각 layer 마다의 activation function 설정
- Loss function 설정
- Optimizer 설정
- 학습

MNIST Classifier model

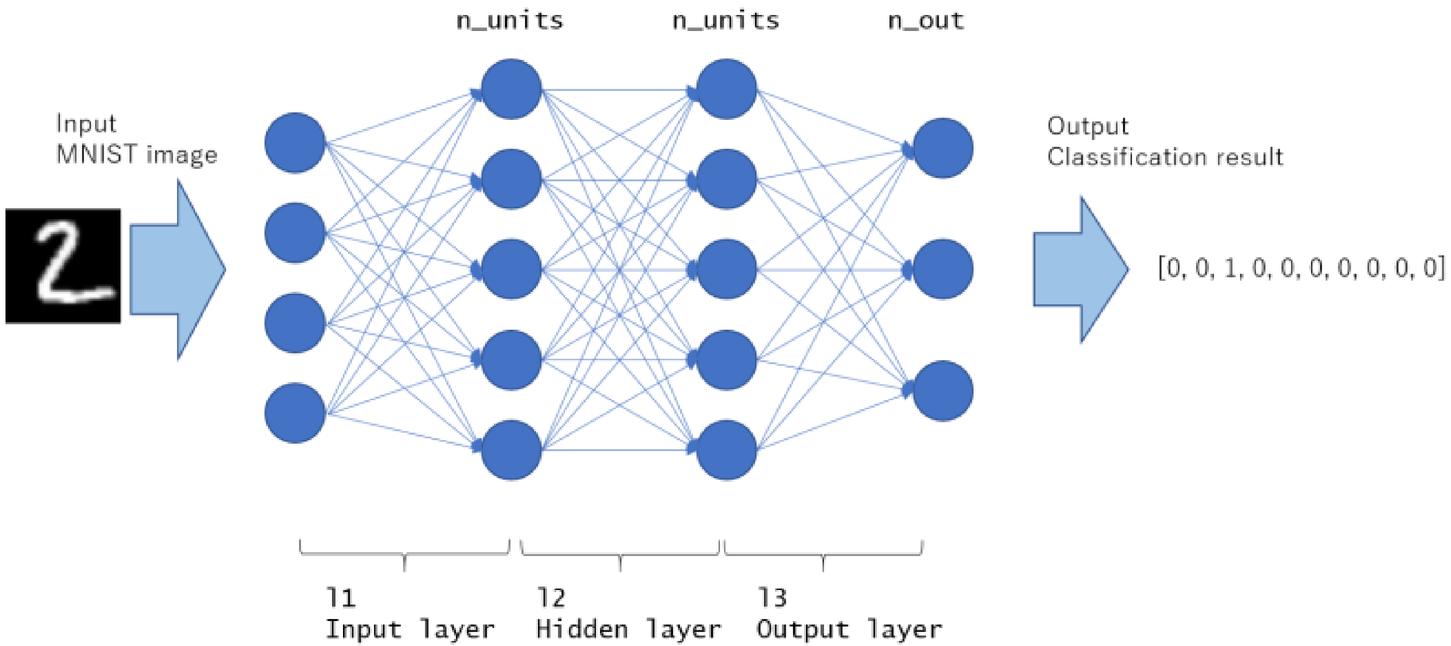
모델학습 관련 개념

- **Epoch : 전체 Sample 데이터를 학습하는것**
- **Step : 1 step당 weight와 Bias를 1회씩 업데이트 하게됨**
- **Batch Size : 1 Step에서 사용한 데이터의 수**
- **Learning rate : 경사 하강법에서 학습 단계별로 움직이는 학습 속도**
- Ex) Batch Size 가 100, Step이 10이면 약 1000개의 데이터를 이용



MNIST Classifier model

- 신경망모델 구성



MNIST Classifier model

- 데이터 불러오기
  - Torchvision
  - https://pytorch.org/vision/stable/index.html
  - The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.

```
import torchvision
import torchvision.transforms as transforms

train_dataset = torchvision.datasets.MNIST(root="MNIST_data/",
                    train=True,
                    transform=transforms.ToTensor(),
                    download=True)
test_dataset = torchvision.datasets.MNIST(root="MNIST_data/",
                    train=False,
                    transform=transforms.ToTensor(),
                    download=True)
```

transform 옵션으로 데이터 조작
ex) ToTensor(), Normalize(mean, std)

```
transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean = (0.5,), std = (0.5,))
    ])
```

https://pytorch.org/vision/stable/transforms.html

MNIST Classifier model

- 모델 학습시 데이터 이용
  - Dataloader
  - 1 step마다 1의 batch size 데이터 사용
  - https://pytorch.org/docs/stable/data.html#module-torch.utils.data
  - It represents a Python iterable over a dataset

Dataloader 지정

```
batch_size = 128

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)
```

Dataset iter

```
for b_x, b_y in train_dataloader:
  b_x = b_x.view(-1, 28*28).to(device)
  logits = model(b_x)  # forward propagation
  loss = criterion(logits, b_y.to(device)) # get cost
```

MNIST Classifier model

- GPU 사용
  - https://pytorch.org/docs/stable/cuda.html
  - CPU 연산 속도 느림 → GPU 사용

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

torch.cuda.is_available로 GPU 사용가능한지 확인
device 선택

```
b_x = b_x.view(-1, 28*28).to(device)
```

CPU Tensor → CUDA Tensor

MNIST Classifier model

- 모델 구성

```
class Model(nn.Module):
  def __init__(self):
    super(Model, self).__init__()
    self.linear1 = nn.Linear(784, 784*3)
    self.linear2 = nn.Linear(784*3, 784*2)
    self.linear3 = nn.Linear(784*2, 10)

    self.activation = nn.Sigmoid()

  def forward(self, x):
    z1 = self.linear1(x)
    a1 = self.activation(z1)

    z2 = self.linear2(a1)
    a2 = self.activation(z2)

    z3 = self.linear3(a2)

    return z3
```

(28, 28)인 데이터를 (784)로 만들어 neural net에 사용할 수 있게 함

MNIST Classifier model

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
[2]  if torch.cuda.is_available():
         device = torch.device('cuda')
     else:
         device = torch.device('cpu')
```

```
[3]  import torchvision
     import torchvision.transforms as transforms

     train_dataset = torchvision.datasets.MNIST(root="MNIST_data/",
                                   train=True,
                                   transform=transforms.ToTensor(),
                                   download=True)
     test_dataset = torchvision.datasets.MNIST(root="MNIST_data/",
                                   train=False,
                                   transform=transforms.ToTensor(),
                                   download=True)
```

```
[4]  batch_size = 128

     train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
     test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)
```

```
class Model(nn.Module):
  def __init__(self):
    super(Model, self).__init__()
    self.linear1 = nn.Linear(784, 784*3)
    self.linear2 = nn.Linear(784*3, 784*2)
    self.linear3 = nn.Linear(784*2, 10)

    self.activation = nn.Sigmoid()

  def forward(self, x):
    z1 = self.linear1(x)
    a1 = self.activation(z1)

    z2 = self.linear2(a1)
    a2 = self.activation(z2)

    z3 = self.linear3(a2)

    return z3
```

```
[6]  model = Model().to(device).train()
```

```
[7]  optimizer = optim.SGD(model.parameters(), lr=0.1) # set optimizer
```

```
[8]  criterion = nn.CrossEntropyLoss()
```

MNIST Classifier model

```
[9] epochs = 15

    model.train()
    for epoch in range(epochs):
        avg_cost = 0
        total_batch_num = len(train_dataloader)

        for b_x, b_y in train_dataloader:
          b_x = b_x.view(-1, 28*28).to(device)
          logits = model(b_x)  # forward propagation
          loss = criterion(logits, b_y.to(device)) # get cost

          optimizer.zero_grad()
          loss.backward() # backward propagation
          optimizer.step() # update parameters

          avg_cost += loss / total_batch_num

        print('Epoch : {} / {}, cost : {}'.format(epoch+1, epochs, avg_cost))
```

```
Epoch : 1 / 15, cost : 2.325561285018921
Epoch : 2 / 15, cost : 1.4498095512390137
Epoch : 3 / 15, cost : 0.743457555770874
Epoch : 4 / 15, cost : 0.5391835570335388
Epoch : 5 / 15, cost : 0.4553937017917633
Epoch : 6 / 15, cost : 0.41385599970817566
Epoch : 7 / 15, cost : 0.3901534676551819
Epoch : 8 / 15, cost : 0.3743925988674164
Epoch : 9 / 15, cost : 0.3631013035774231
Epoch : 10 / 15, cost : 0.3511328101158142
Epoch : 11 / 15, cost : 0.33993449807167053
Epoch : 12 / 15, cost : 0.3316379487514496
Epoch : 13 / 15, cost : 0.3236689567565918
Epoch : 14 / 15, cost : 0.31599244475364685
Epoch : 15 / 15, cost : 0.3092549741268158
```

loss: 현재 batch size 만큼의 cost function
avg_cost : loss/total_batch_num 모든 데이터셋에 대한 cost 값

MNIST Classifier model

- Accuracy 확인
  - 얼마나 모델이 데이터를 잘 분류하는지에 대한 평가

$$\frac{\sum_{k=1}^{C} k\text{와 } y\text{가 같을 때의 데이터 수}}{\text{모든 데이터 수}}$$

```python
correct = 0
total = 0

model.eval()
for b_x, b_y in test_dataloader:
  b_x = b_x.view(-1, 784).to(device)

  with torch.no_grad():
    logits = model(b_x)
  probs = nn.Softmax(dim=1)(logits)

  predicts = torch.argmax(logits, dim=1)

  total += len(b_y)
  correct += (predicts == b_y.to(device)).sum().item()

print(f'Accuracy of the network on test images: {100 * correct // total} %')
```

현재 step의 데이터 개수

예측한 라벨과 실제 라벨 비교 True sum

```
Accuracy of the network on test images: 90 %
```

오늘 실습 내용

- **MNIST data classifier model GPU에서 학습**