PyTorch

Pytorch로 딥러닝

인공신경망 (DNN) 기초 – 두번째 시간

임낙준

PyTorch

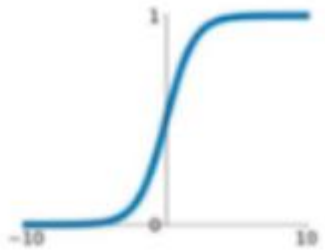
Contents

PyTorch

4. 다양한 Activation Function

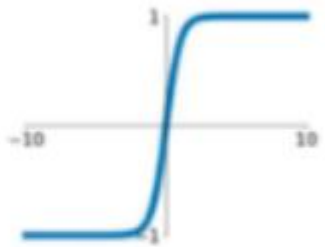
Sigmoid

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



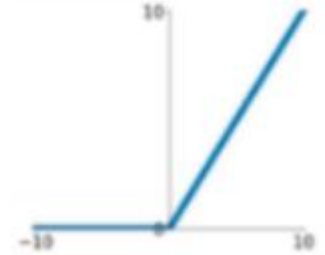
tanh

$$\tanh(x)$$



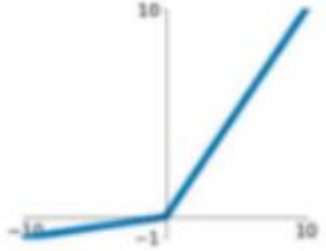
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

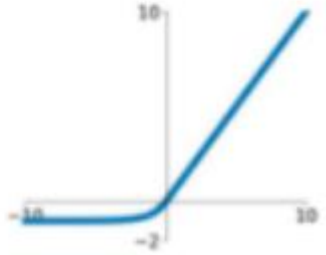


Maxout

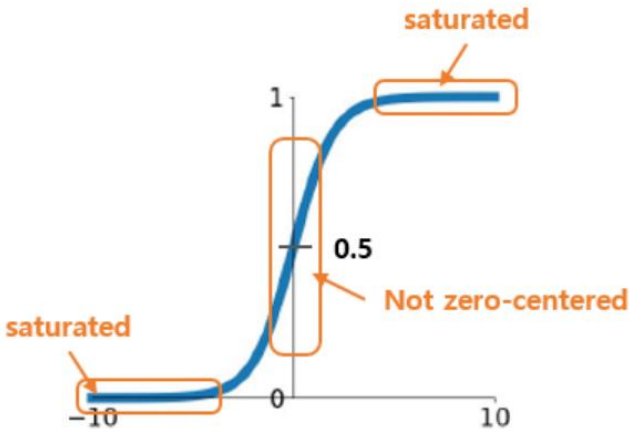
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

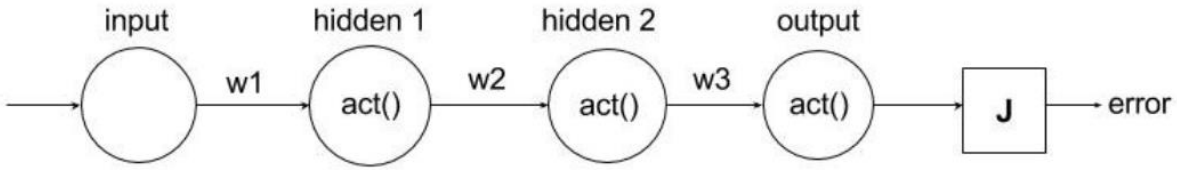
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



4. 다양한 Activation Function : Sigmoid





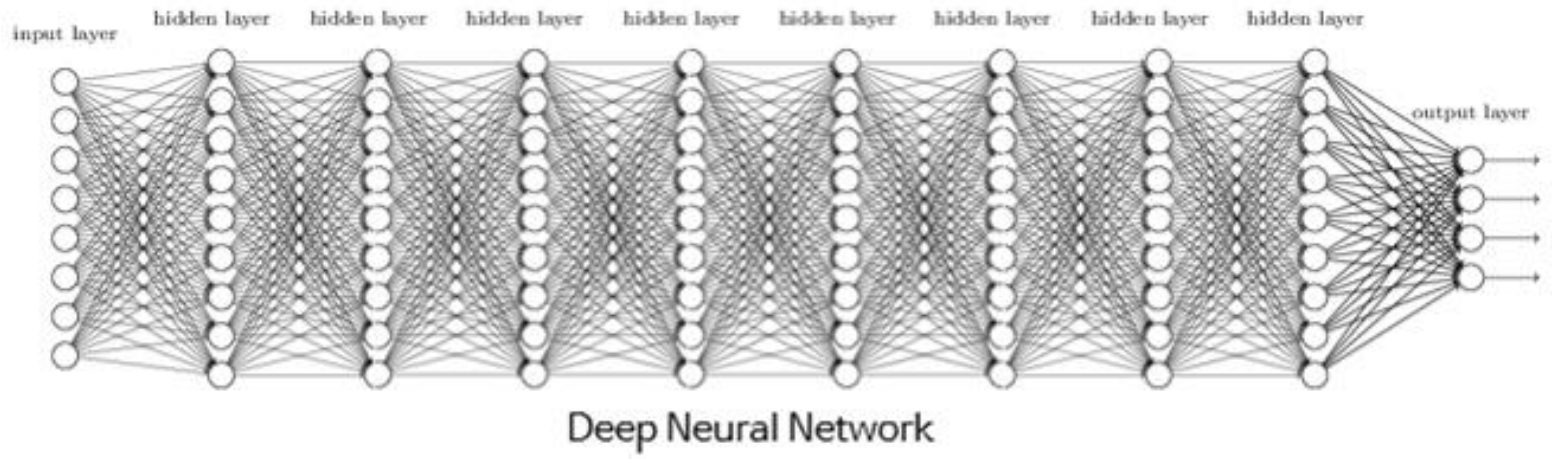
$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

그래디언트 소실(Vanishing Gradient)

MLP에서 Hidden layer(은닉층) 하위층으로 진행될수록 전달되는 그래디언트가 점점 작아지는 문제

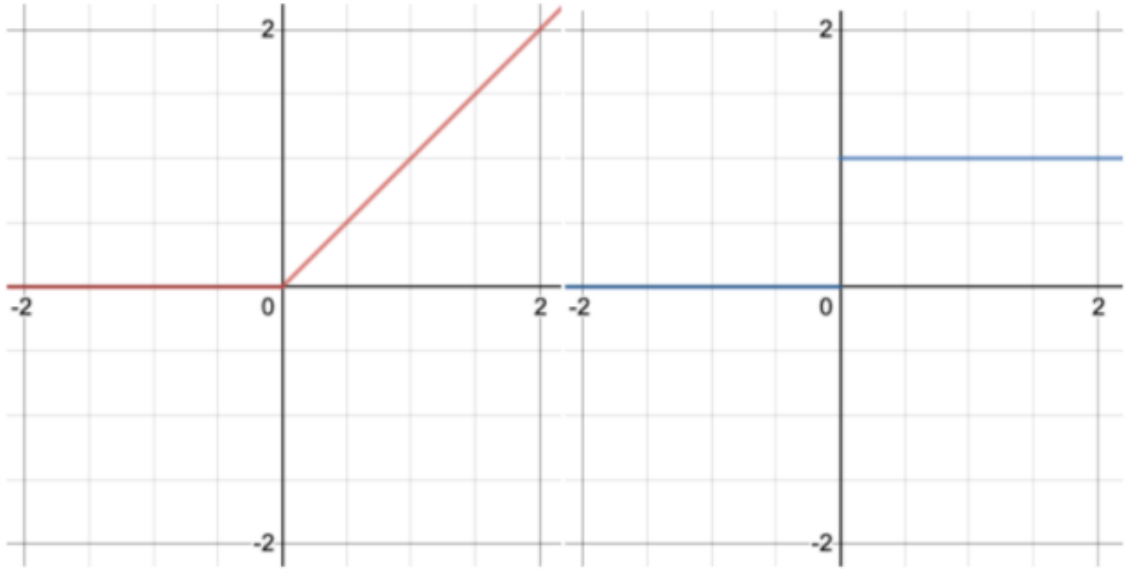
원인! "시그모이드 활성화 함수"

4. 다양한 Activation Function : Sigmoid



4. 다양한 Activation Function : Relu

$$f(x)=\begin{cases} x(x\geq 0) \\ 0(x<0) \end{cases} \qquad f'(x)=\begin{cases} 1 & (x\geq=0) \\ 0 & (x<0) \end{cases}$$



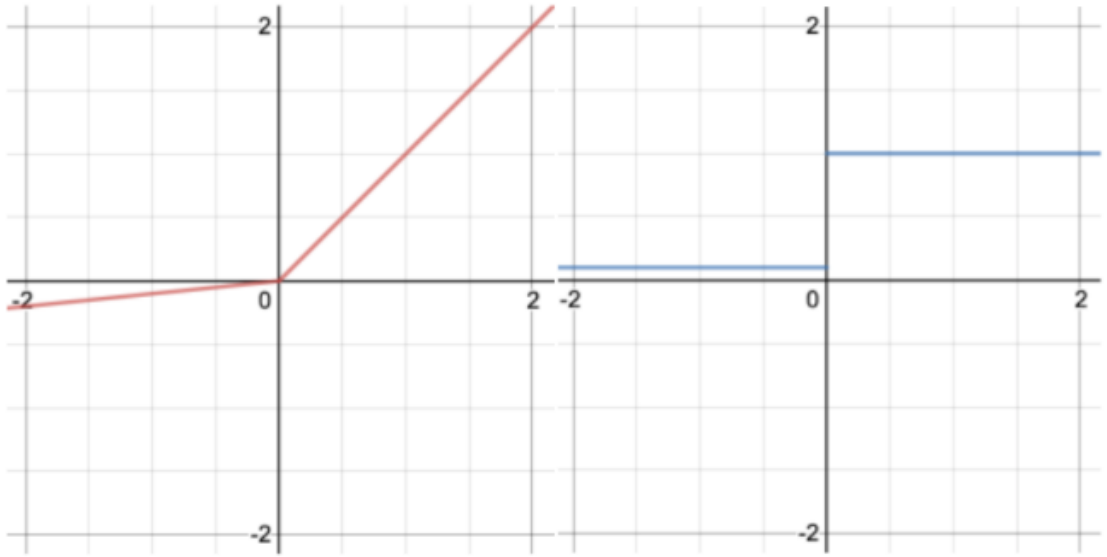
특징: 0 이하의 값은 다음 레이어에 전달하지 않음. 0이상의 값은 그대로 출력합니다.
사용처: CNN을 학습시킬 때 많이 사용됩니다.
한계점: 한번이라도 0 미분값을 다음 레이어에 전달하면 이후의 뉴런들의 출력값이 모두 0이 되는 현상이 발생.(**dying ReLU)**

이후 음수 출력 값을 소량이나마 다음 레이어에 전달하는 방식으로 개선한 활성화 함수들이 등장

PyTorch

4. 다양한 Activation Function : LeakyRelu

$$f(x)=\begin{cases} x & (x>0) \\ \alpha x & (x\leq 0) \end{cases} \qquad f'(\alpha,x)=\begin{cases} 1 & (x>0) \\ \alpha & (x\leq 0) \end{cases}$$

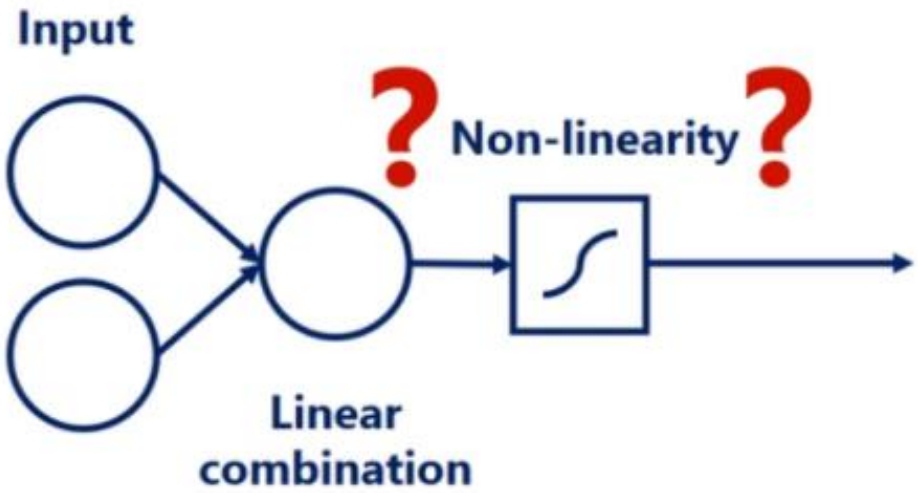


특징: ReLU와 거의 비슷한 형태를 가짐.
Relu와 달리 입력 값이 음수일 때도 완만한 선형 함수를 그려줍니다.

(일반적으로 알파를 0.01로 설정)

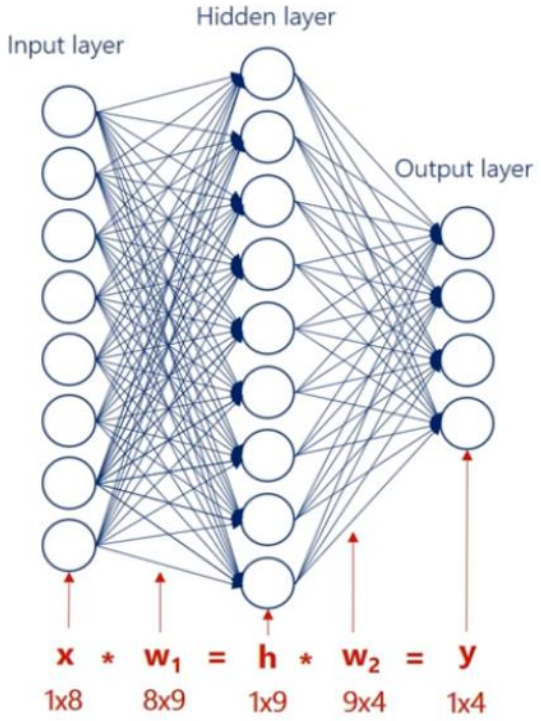
PyTorch

4. Activation Function : 비선형성 제공



4. Activation Function : 비선형성 제공

Imagine a network with no **non-linearities,** just linear combinations



활성화 함수가 없다면?

$$h = x * w_1$$

$$y = x * w_1 * w_2 = x * w^{\star}$$

w_1: 8x9, w_2: 9x4, $w^{\star}$: 8x4

두개의 연속 선형변환은 단일 변환과 동일

결국 100개의 레이어를 추가하더라도 하나의 선형변환으로 바뀔 수 있는 것.

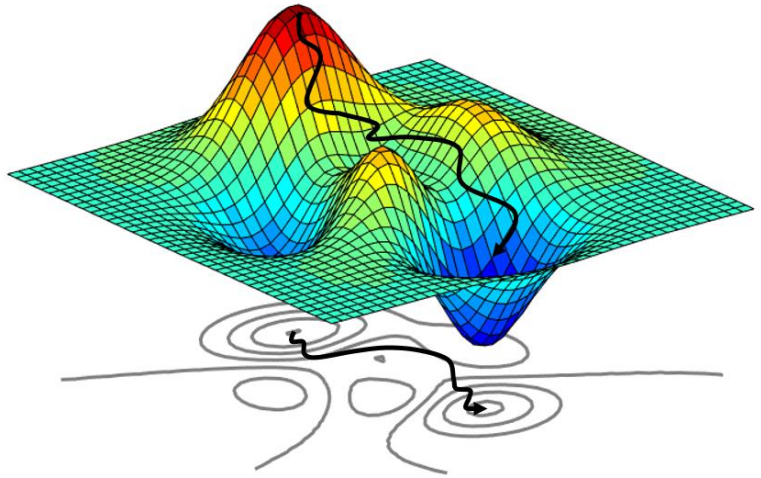
단순 선형 계산과 별반 달라질 것이 없음.

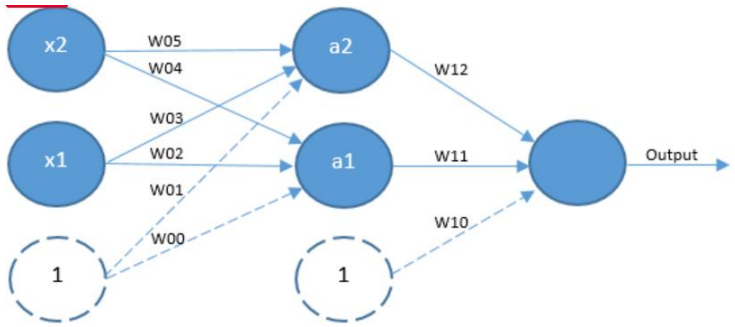
머신러닝과 딥러닝의 차이.

딥러닝 : "비선형적 " 인 함수를 근사시킬 수 있음!

PyTorch

5. Optimization



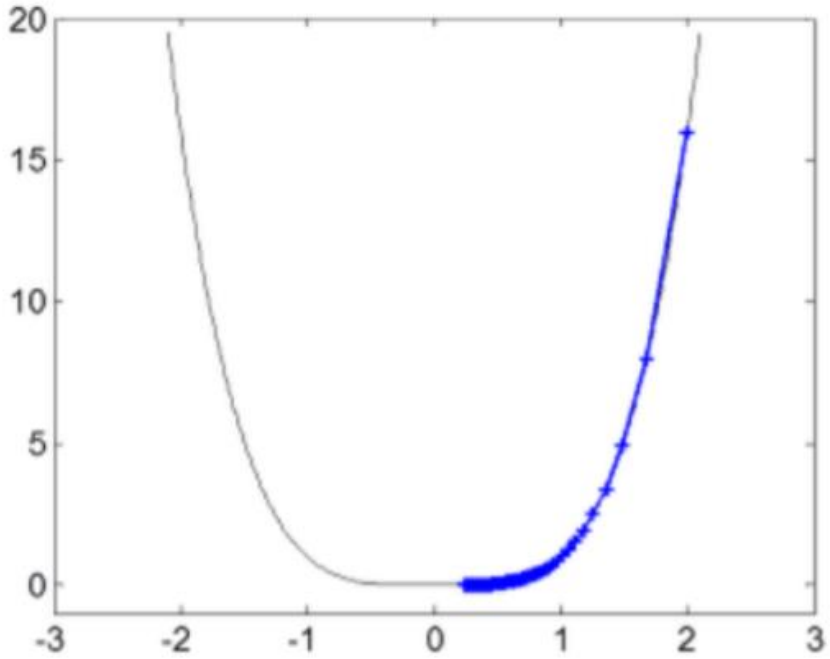


최적화 : loss function의 값을 최소화 하는 모수를 찾는 과정

모델의 목적에 따라 다양한 loss function을 쓸 수 있음
Regression : MSE
Classification : CrossEntropy 등

딥러닝에서 우리가 찾고자 하는 모수는
Weight parameter. Loss를 최소화 하는 weight parameter를 찾는 것이 우리의 목적

5. Optimizer : GD

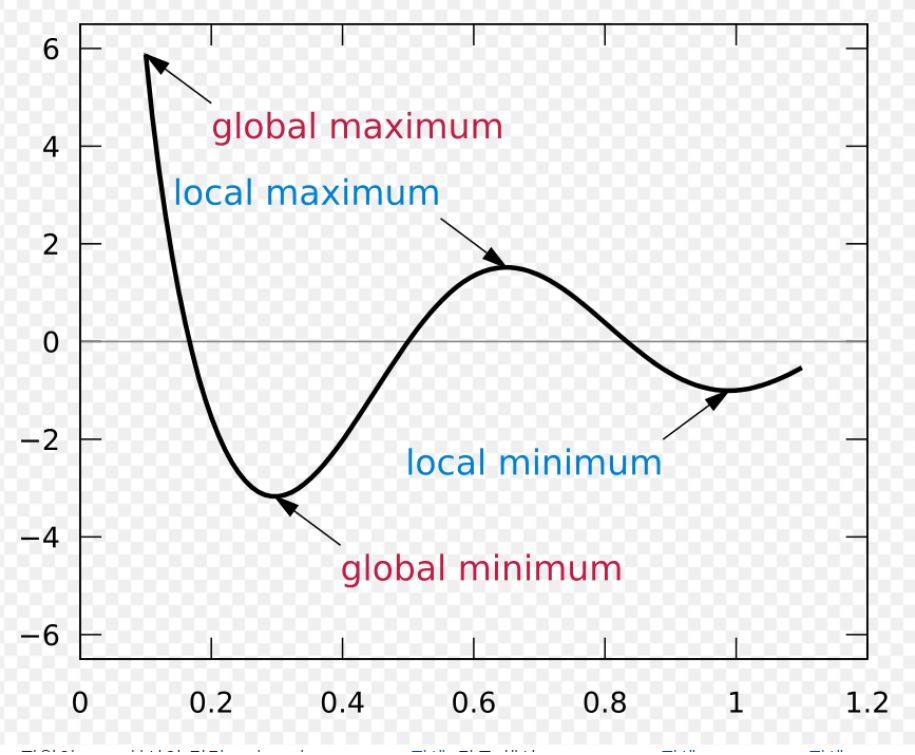


$$x_0 = 2.0$$
$$x_1 = x_0 - \lambda f'(x_0) = 1.6800$$
$$x_2 = x_1 - \lambda f'(x_1) = 1.4903$$
$$x_3 = x_2 - \lambda f'(x_2) = 1.3579$$
$$x_4 = x_3 - \lambda f'(x_3) = 1.2578$$
$$x_5 = x_4 - \lambda f'(x_4) = 1.1782$$
$$x_6 = x_5 - \lambda f'(x_5) = 1.1128$$
$$x_7 = x_6 - \lambda f'(x_6) = 1.0576$$
$$x_8 = x_7 - \lambda f'(x_7) = 1.0103$$
$$x_9 = x_8 - \lambda f'(x_8) = 0.9691$$
...

GD (Gradient Descent)

반복적인 업데이트로 loss function 을 최소화 하는 모수를 업데이트.

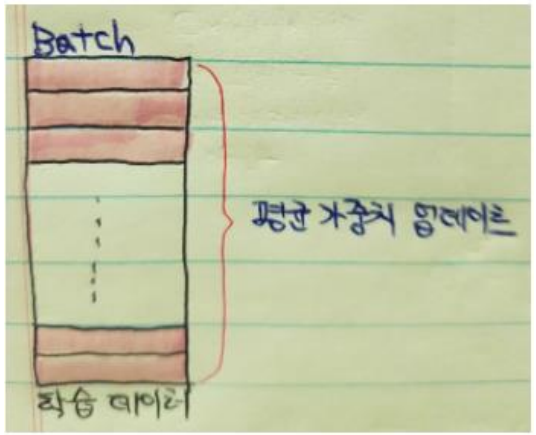
5. Optimizer : GD

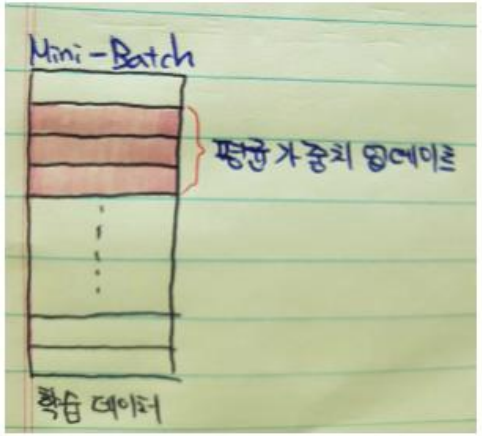


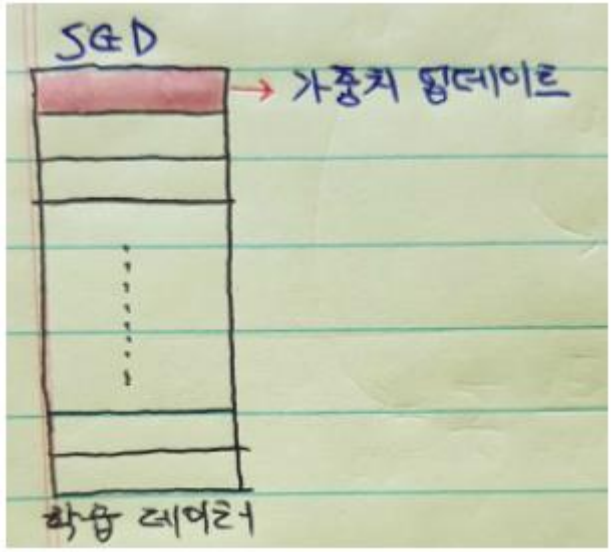
GD (Gradient Descent)의 문제

1. 전체 데이터의 loss를 계산하기 때문에 최적화 속도가 너무 느림
2. Local Minimum에 빠질 수 있다.

5. Optimizer : Minibatch GD, SGD







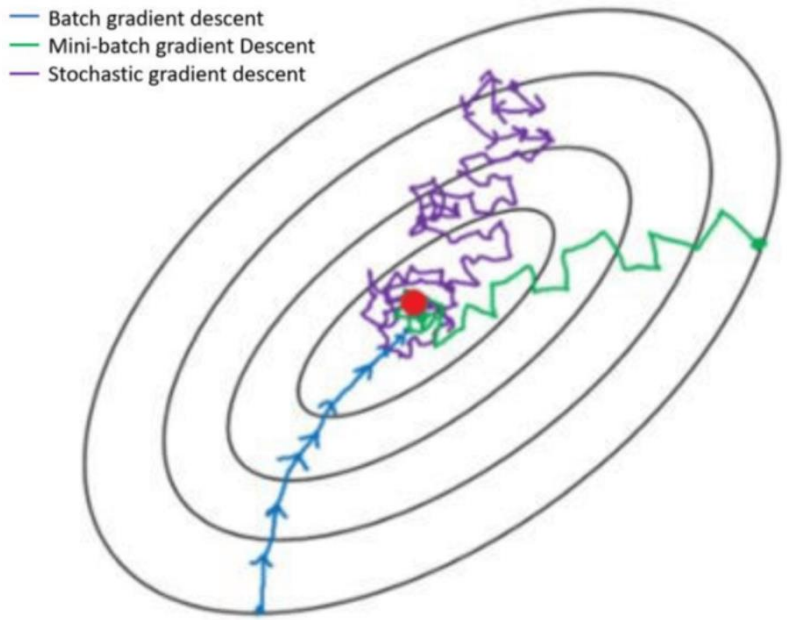
MGD (Minibatch Gradient Descent)

학습데이터의 일부를 가지고 loss계산하고 이를 통해 파라미터 업데이트.

SGD(Stochatic Gradient Descent)

랜덤으로 추출된 데이터 하나의 loss 만을 사용하여 loss를 계산하고, 이를 통해 파라미터 업데이트.

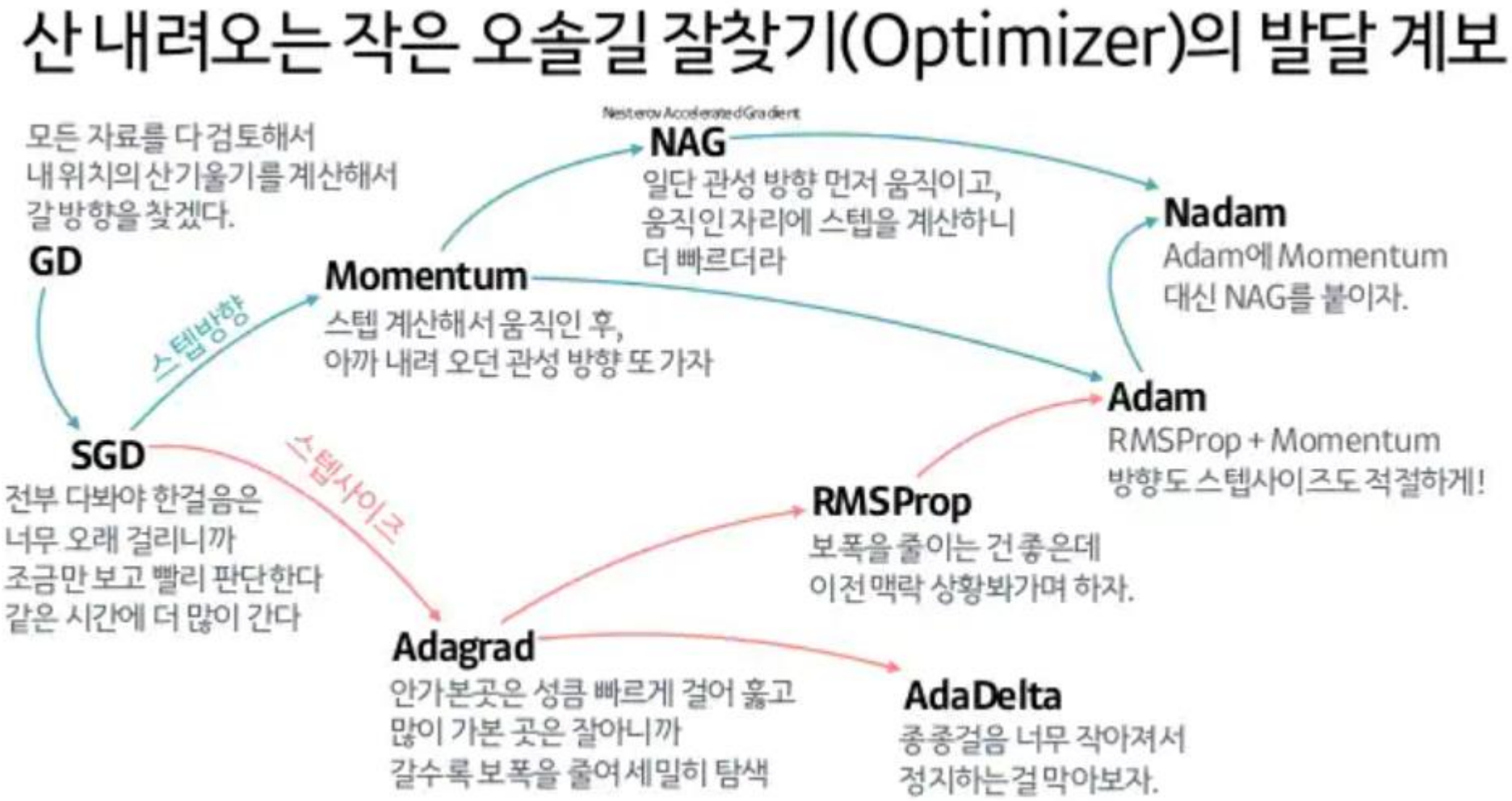
5. Optimizer : SGD



잘못된 방향으로 업데이트 할 수 있지만 계산속도가 훨씬 빠르기 때문에, 같은 시간에 더 많은 Step을 나아갈 수 있음.

Local Minima에 빠지지 않고 Global Minima에 수렴할 가능성이 더 높음

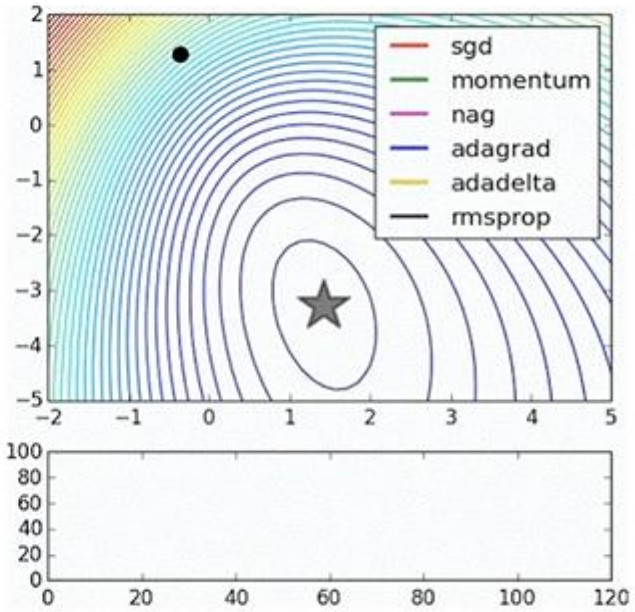
5. Optimizer : 발달계보



5. Optimizer in pytorch

`torch.optim` is a package implementing various optimization algorithms. Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can be also easily integrated in the future.

- torch.optim.SGD
- torch.optim.Adadelta
- torch.optim.Adagrad
- torch.optim.Adam
- torch.optim.SparseAdam
- torch.optim.Adamax
- torch.optim.ASGD
- torch.optim.LBFGS
- torch.optim.RMSprop
- torch.optim.Rprop



6. 데이터로더

데이터 로더의 필요성

- 훈련에 필요한 데이터를 한꺼번에 메모리에 올릴 시 과부하 발생
- 데이터를 전부 다 사용하여 가중치 업데이트를 하기에는 오랜 시간 소요. 데이터의 일부분만 번갈아 가면서 사용하여 가중치 업데이트할 필요성.

파이토치에서는 전체 데이터를 간단하게 minibatch로 만들어 주는 유용한 도구들을 제공함

6. 데이터로더

CLASS `torch.utils.data.Dataset` [SOURCE]

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`.

At the heart of PyTorch data loading utility is the `torch.utils.data.DataLoader` class. It represents a Python iterable over a dataset, with support for

- map-style and iterable-style datasets,
- customizing data loading order,
- automatic batching,
- single- and multi-process data loading,
- automatic memory pinning.

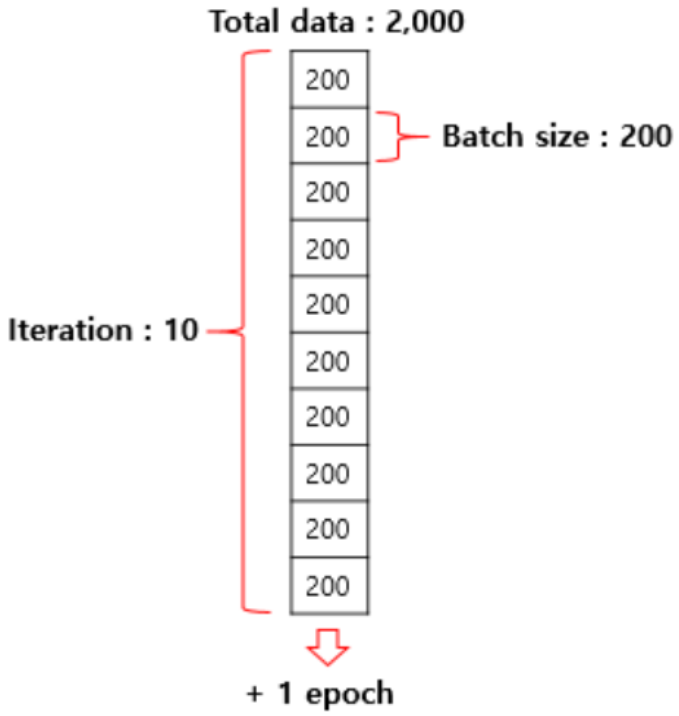
파이토치의 데이터 로딩 도구

데이터셋(Dataset)
데이터로더(DataLoader)

미니 배치 학습, 데이터 셔플(shuffle), 병렬 처리까지 간단히 수행 가능.

사용법
Dataset을 정의하고 이를 DataLoader에 전달

Epoch / Batch size / Iteration



- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

6. 데이터로더 : Pytorch Dataset

```
[1]  import torch

[2]  from torch.utils.data import Dataset

     class CustomDataset(Dataset) :
       def __init__(self) :
         self.x_data = [[73, 80, 75],
                        [93, 88, 93],
                        [89, 91, 90],
                        [96, 98, 100],
                        [73, 66, 70]]
         self.y_data [[152], [185], [180], [196], [142]]

       def __len__(self) :
         return len(self.x_data)

       def __getitem__(self, idx) :
         x = torch.FloatTensor(self.x_data[idx])
         y = torch.FloatTensor(self.y_data[idx])

         return x, y
```

커스텀 데이터셋 생성
파이토치의 데이터셋(Dataset)과 데이터로더(DataLoader)
미니 배치 학습, 데이터 셔플(shuffle), 병렬 처리까지 간단히 수행 가능.
사용법 : Dataset을 정의하고 이를 DataLoader에 전달

- Torch,utils.data.Dataset를 상속받는 class 생성
- 반드시 있어야 하는 두가지 매직 메서드
- __len__(self) :
  이 데이터셋의 총 데이터 수
- __getitem__(self, idx)
  어떤 인덱스(idx)를 받았을때, 그에 상응하는 입출력 데이터 반환

6. 데이터로더 : Pytorch DataLoader

```
# 모델 데이터 셋 생성
dataset = CustomDataset()


from torch.utils.data import DataLoader

dataloader = DataLoader(
    dataset,
    batch_size = 2,
    shuffle = True

)
```

torch.utils.data.DataLoader 사용

Batch_size = 2
각 minibatch의 크기
통상적으로 2의 제곱수로 설정한다
(16, 32, 64, 128,…)

Shuffle = True
매 Epoch마다 데이터 셋을 섞어서, 데이터가 학습되는 순서를 바꾼다. 모델이 데이터 셋의 순서를 외우는 것을 방지

6. 데이터로더 : Full code

```
optimizer = torch.optim.SGD(model.parameters(), lr = 1e-5)

nb_epochs = 20
for epoch in range(nb_epochs +1) :
  for batch_idx, samples in enumerate(dataloader) :
    x_train, y_train = samples

    # H(x) prediction 계산
    prediction = model(x_train)

    # cost 계산
    # F는 torch.nn.functional을 의미. 노트북 맨 상단을 볼것
    cost = F.mse_loss(prediction, y_train)

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

    print('Epoch {:4d}/{} Batch {}/{} Cost : {:.6f}'.format(
        epoch, nb_epochs, batch_idx+1, len(dataloader),
        cost.item()
    ))
```

모델 훈련 과정

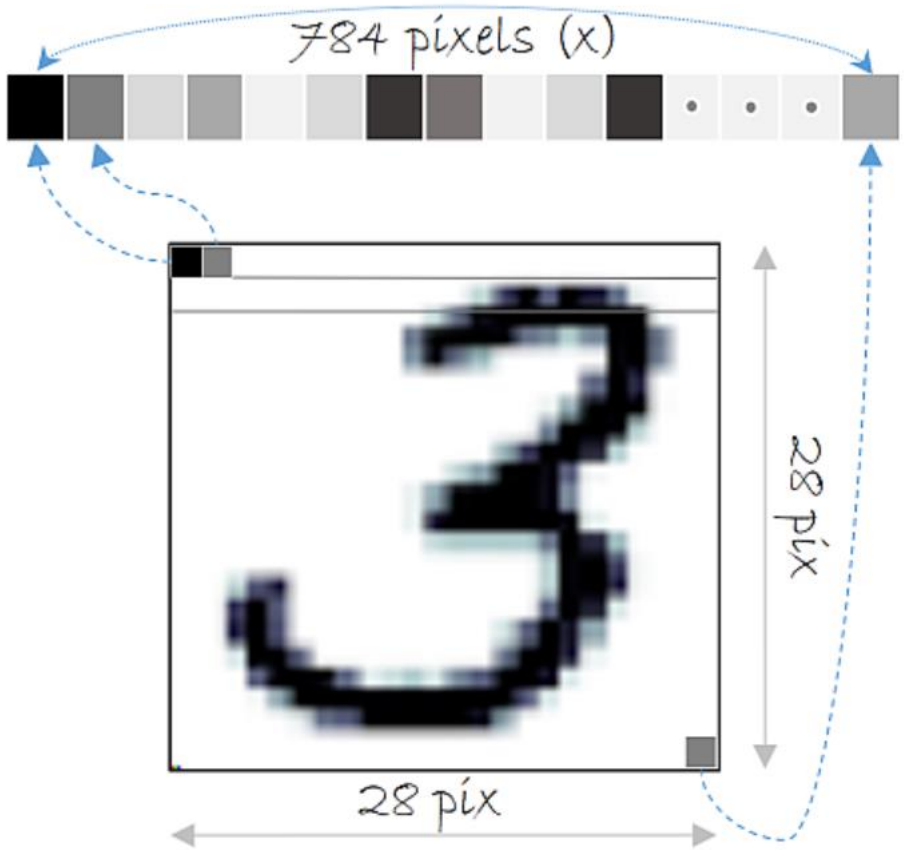
한 Epoch안에서
Dataloader를 통해 Minibatch들로 모델을 훈련하는 구조.

```
Epoch    0/20 Batch 1/3 Cost : 52756.132812
Epoch    0/20 Batch 2/3 Cost : 16168.710938
Epoch    0/20 Batch 3/3 Cost : 8405.194336
Epoch    1/20 Batch 1/3 Cost : 1262.062622
Epoch    1/20 Batch 2/3 Cost : 410.094910
Epoch    1/20 Batch 3/3 Cost : 134.174561
Epoch    2/20 Batch 1/3 Cost : 59.314720
Epoch    2/20 Batch 2/3 Cost : 4.435835
Epoch    2/20 Batch 3/3 Cost : 2.516456
Epoch    3/20 Batch 1/3 Cost : 7.529008
Epoch    3/20 Batch 2/3 Cost : 0.991707
Epoch    3/20 Batch 3/3 Cost : 4.111855
Epoch    4/20 Batch 1/3 Cost : 3.384340
```

7. MNIST

- MNIST는 숫자 0부터 9까지의 이미지로 구성된 손글씨 데이터셋
- 우체국에서 편지의 우편 번호를 인식하기 위해서 만들어진 훈련 데이터
- 총 60,000개의 훈련 데이터와 레이블, 총 10,000개의 테스트 데이터와 레이블로 구성
- 레이블은 0부터 9까지 총 10개

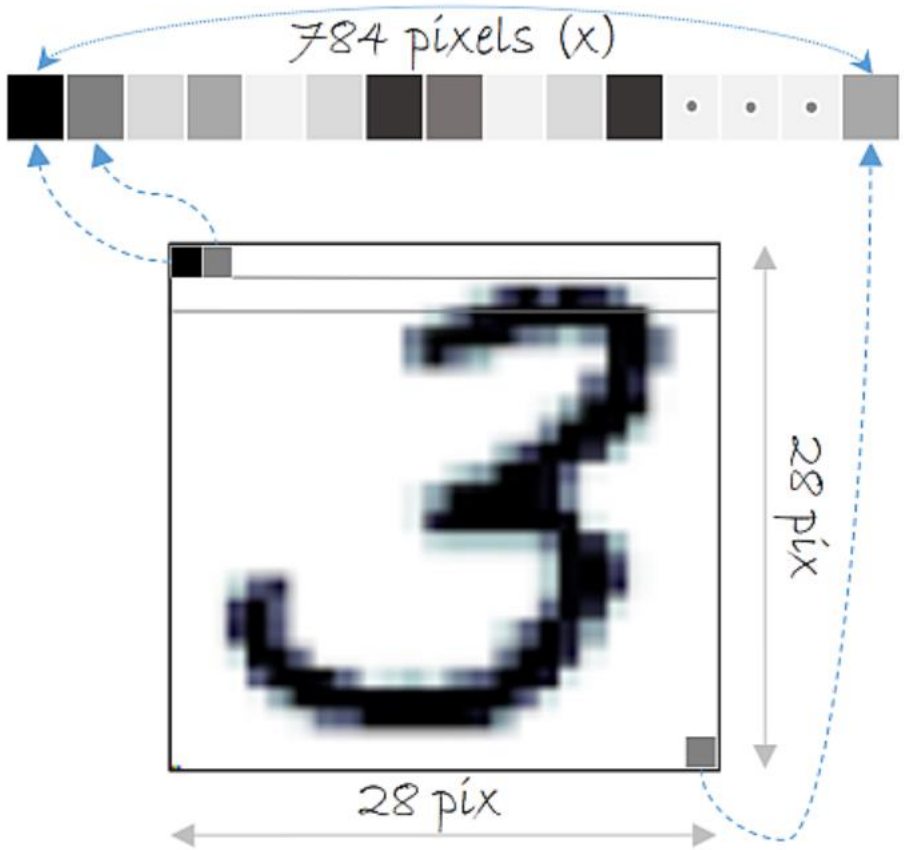
7. MNIST



MNIST 문제 : 손글씨로 적힌 숫자 이미지가 무슨 숫자인지 맞추는 문제

MNIST 각각의 이미지 : 28 픽셀 × 28 픽셀로 구성

7. MNIST



MNIST 문제 : 손글씨로 적힌 숫자 이미지가 무슨 숫자인지 맞추는 문제

MNIST 각각의 이미지 : 28 픽셀 × 28 픽셀로 구성

7. MNIST classification

```
# Lab 10 MNIST and softmax
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import random
import matplotlib.pyplot as plt
```

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

# for reproducibility
random.seed(777)
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
```

1. 패키지 가져오기

Torchvision, matplotlib

7. MNIST classification

```
# MNIST dataset
mnist_train = dsets.MNIST(root='MNIST_data/',
                          train=True,
                          transform=transforms.ToTensor(),
                          download=True)

mnist_test = dsets.MNIST(root='MNIST_data/',
                         train=False,
                         transform=transforms.ToTensor(),
                         download=True)
```

2. 데이터 갖고오기

Root : 저장공간

Train : train 데이터 여부

Transform : 이미지 변환 방법

7. MNIST classification

3. EDA

```
# mnist에는 데이터가 어떤식으로 들어가 있나 보기
mnist_train[0]
```

```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
```

```
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000]]]), 5)
```

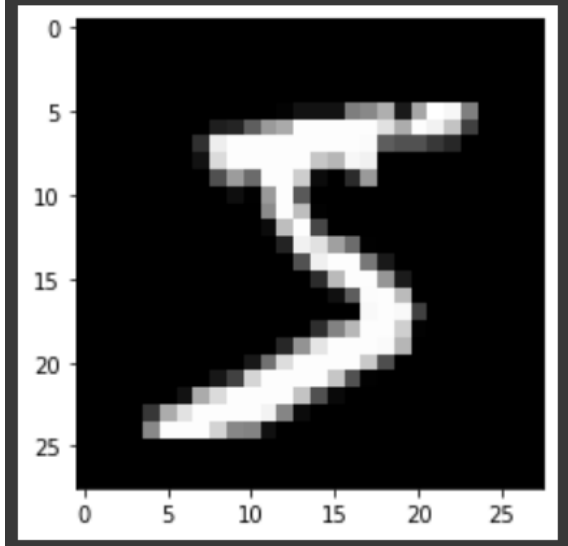
7. MNIST classification

PyTorch

3. EDA

```
# 데이터 시각화
plt.imshow(mnist_train[0][0].reshape(28,28), cmap='gray')
print("이 그림의 라벨은 다음과 같습니다 : ", mnist_train[0][1] )
```

```
이 그림의 라벨은 다음과 같습니다 :  5
```



7. MNIST classification

PyTorch

4. 데이터로더 생성

```
# parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
```

```
# dataset loader
data_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True)
```

7. MNIST classification

```
# nn layers
linear1 = torch.nn.Linear(784, 256, bias=True)
linear2 = torch.nn.Linear(256, 256, bias=True)
linear3 = torch.nn.Linear(256, 10, bias=True)
relu = torch.nn.ReLU()
```

```
# model
model = torch.nn.Sequential(linear1, relu, linear2, relu, linear3).to(device)
```

4. 모델 생성

2개의 hiden layer,
1개의 출력층으로 구성
첫번째 레이어 : 256개의 hidden node
두번째 레이어 : 256개의 hidden node
세번째 레이어 : 출력층

7. MNIST classification

```
# nn layers
linear1 = torch.nn.Linear(784, 256, bias=True)
linear2 = torch.nn.Linear(256, 256, bias=True)
linear3 = torch.nn.Linear(256, 10, bias=True)
relu = torch.nn.ReLU()

# model
model = torch.nn.Sequential(linear1, relu, linear2, relu, linear3)
```

모델 구조 시각화

PyTorch

5. 모델 훈련

```
# define cost/loss & optimizer
criterion = torch.nn.CrossEntropyLoss()     # Softmax is internally computed.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
total_batch = len(data_loader)
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:
        # reshape input image into [batch_size by 784]
        # label is not one-hot encoded
        X = X.view(-1, 28 * 28).to(device)
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = model(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch   # 배치들마다 계산된 cost를 평균내어 한 epoch 당 cost로 계산

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))
```

7. MNIST classification

PyTorch

5. 모델 훈련

```
        avg_cost += cost / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))

print('Learning finished')
```

```
Epoch: 0001 cost = 0.298944235
Epoch: 0002 cost = 0.110240154
Epoch: 0003 cost = 0.074017763
Epoch: 0004 cost = 0.053192556
Epoch: 0005 cost = 0.040261999
Epoch: 0006 cost = 0.030782444
Epoch: 0007 cost = 0.025884338
Epoch: 0008 cost = 0.020093258
Epoch: 0009 cost = 0.016206736
Epoch: 0010 cost = 0.018373892
Epoch: 0011 cost = 0.013398918
Epoch: 0012 cost = 0.011096421
Epoch: 0013 cost = 0.011578427
Epoch: 0014 cost = 0.009691190
Epoch: 0015 cost = 0.009304078
Learning finished
```

7. MNIST classification

```
# Test the model using test sets
with torch.no_grad():
    X_test = mnist_test.test_data.view(-1, 28 * 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = model(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print('Accuracy:', accuracy.item())

    # Get one and predict
    r = random.randint(0, len(mnist_test) - 1)
    X_single_data = mnist_test.test_data[r:r + 1].view(-1, 28 * 28).float().to(device)
    Y_single_data = mnist_test.test_labels[r:r + 1].to(device)

    print('Label: ', Y_single_data.item())
    single_prediction = model(X_single_data)
    print('Prediction: ', torch.argmax(single_prediction, 1).item())
```

```
Accuracy: 0.9785000085830688
Label:  5
Prediction:  5
```

6. 모델 테스트

with torch.no_grad()

이와 같이 `no_grad()` with statement에 포함시키게 되면 Pytorch는 autograd engine을 꺼버린다. 이 말은 더 이상 자동으로 gradient를 트래킹하지 않는다는 말이 된다.

`torch.no_grad()`의 주된 목적은 autograd를 끔으로써 메모리 사용량을 줄이고 연산 속도를 높이기 위함이다. 사실상 어짜피 안쓸 gradient인데 inference시에 굳이 계산할 필요가 없지 않은가?

7. MNIST classification 과제

1. hidden layer의 개수를 6층으로 모델을 만들 것. 각 층별 hidden node의 수는 자유롭게
2. 활성화 함수를 torch.nn.Relu() 말고 torch.nn.Sigmoid()를 썼을때 정말 train loss가 줄어들지 않는지 알아 볼 것
3. 코드에서 사용한 torch.optim.Adam() 말고 다른 오티마이저를 사용