PYTORCH 실습

고민성

Week 3: PyTorch, Logistic Regression and MLP

We will cover basic concepts of PyTorch Framework (tensor operations, GPU utilizing and autograd)

We will implement simple logistic regression and multinomial logistic regression (softmax) with PyTorch

We will use simple linear model and multi-layer perceptron (MLP) in this class

Why PyTorch?

Intuitive and concise code

친숙하고 깔끔한 코드

Define by Run method (Tensorflow is Define and Run method)

순서에 따라서 작동한다

High compatibility with Numpy (almost one-to-one mapping)

numpy와의 호환성이 높음

Load packages & GPU setup

```
!nvidia-smi
```

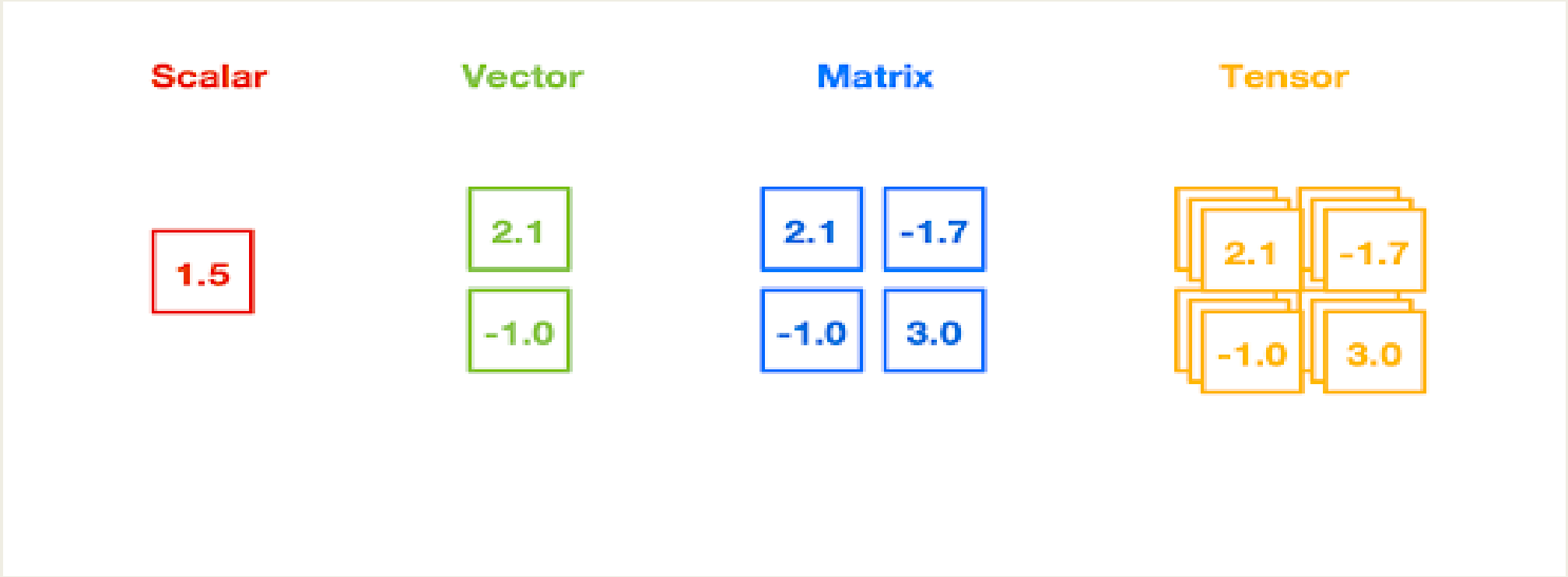
```
Fri Jan 14 07:02:02 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.46       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   43C    P0    30W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
# set gpu by number
import os
os.environ['CUDA_VISIBLE_DEVICES'] = '0'  # setting gpu number
```

<- 사용 가능한 gpu

메모리도 많이 사용 가능함.

PyTorch



수치가 하나 있는 것은 Scalar 방향이라 부르고,

(1,2)로 되어 있는 경우는 이동거리 vector(크기와 방향을 모두 가지는)라 한다.

2차원으로 표현된 matri는 평면을 뜻한다.

Tensor는 3차원 이상을 뜻한다.

PyTorch

Define Numpy array and PyTorch tensor

```
np_array_1 = np.array([1, 2, 3, 4])

[1 2 3 4]

torch_tensor_1 = torch.tensor([1, 2, 3, 4])

tensor([1, 2, 3, 4])
```

Numpy는 np.array를 이용하고 pytorch는 torch.tensor를 이용하여 행렬을 정의한다.

PyTorch Grammar (example)

1) Same operations with identical grammar
같은 코드로 같은 방식으로 작동

```
np_array_1.shape        torch_tensor_1.shape
```

2) Same operations with different gramme
다른 코드로 같은 방식으로 작동

```
np_concate              torch.cat
```

단 np에서는 axis를 이용하여 축을 설정했지만,
Torch에서는 dim을 이용하여 축을 설정한다.

```
np_concate.reshape      torch_concate.view
```

PyTorch Grammar (example)

3) Different operations with same grammer (Confusing operations)
같은 코드로 다른 방식으로 작동

```
x = np.array([1, 2, 3])
x_repeat = x.repeat(2)
Print(x_repeat)

-> [1 1 2 2 3 3]

x = torch.tensor([1, 2, 3])
x_repeat = x.repeat(2)
Print(x_repeat)

-> tensor([1, 2, 3, 1, 2, 3])
```

Gpu utilize

```
print(torch.cuda.is_available())  # Is GPU accessible?
```

```
True
```

Gpu가 사용 가능한지 체크

```
a = torch.ones(3)
 -> tensor([1., 1., 1.])

print(a.device)
-> cpu
```

.device 사용한 디바이스 확인

```
a = a.to('cuda')
print(a.device)
-> cuda:0
```

gpu 업로드

0이 나오는 이유는 위에서 사용 gpu를 0으로 설정

Autograd

The autograd package provides automatic differentiation for all operations on Tensors.

`x = torch.ones(2, 2, requires_grad=True)`에서
`requires_grad=True`는 이 텐서에 대한 미분 값을 저장하겠다는 뜻!

```
x = torch.ones(2, 2, requires_grad=True)

y = x + 2

z = y * y * 3

out = z.mean()
```

일 때

$$\frac{\partial out}{\partial z_i} * \frac{\partial z_i}{\partial y_i} = \frac{1}{4} * 6y_i = 4.5$$

`retain_grad()`, `out.backward()`를 사용한 뒤

`grad`를 사용하여 미분 값을 얻을 수 있게 있다.

`retain_grad()`, `out.backward()`를 사용하지 않고 `grad`를 사용하면 `Nan`값이 나타난다.

참고로, torch.no_grad()는 grad값이 필요하지 않은 경우 사용하며, 메모리를 아낄 때 사용.

nn.Module

```
X = torch.tensor([[1., 2., 3.], [4., 5., 6.]])

-> tensor([[1., 2., 3.], [4., 5., 6.]])
```

행 2개 열 3개

linear_fn = nn.Linear(3, 1) 로 해줘서 3차원을 1차원으로 축소하도록 설정해준다.

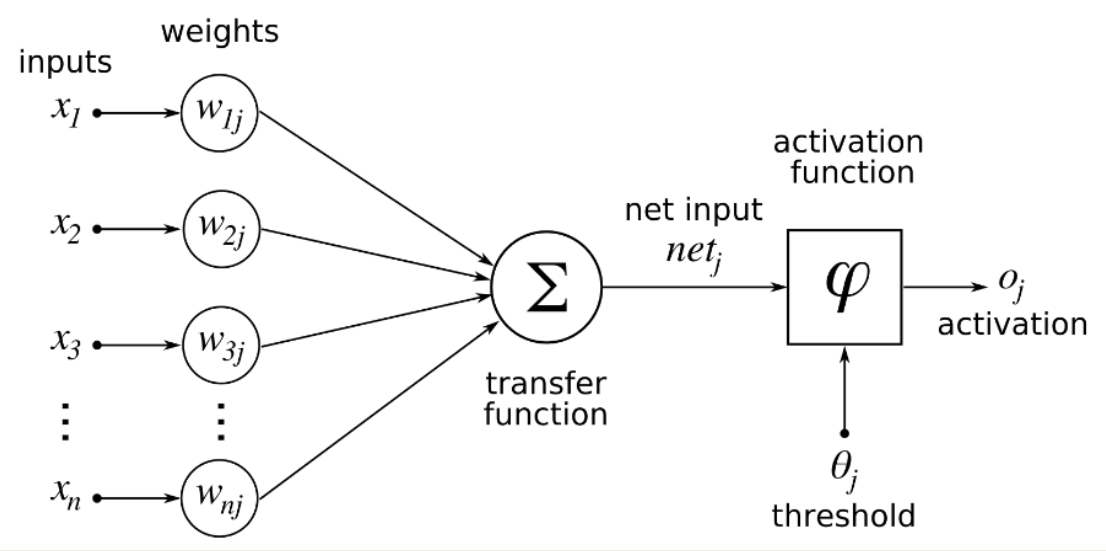
```
linear_fn
-> Linear(in_features=3, out_features=1, bias=True)

Y = linear_fn(X)
print(Y)
-> tensor([[-0.8427], [-3.2153]], grad_fn=<AddmmBackward0>)
```

열이 3개 -> 1개로 축소, 3차원에서 1차원으로 축소되었다. 텐서의 크기 조정이 가능

nn.Linear 뿐만 아니라 다른 nn 모듈을 통해서 커스터마이징 할 수 있다.

nn.Conv2, nn.RNNCell, nn.LSTMCell, nn.GRUCell, nn.Transformer

What is activation function?

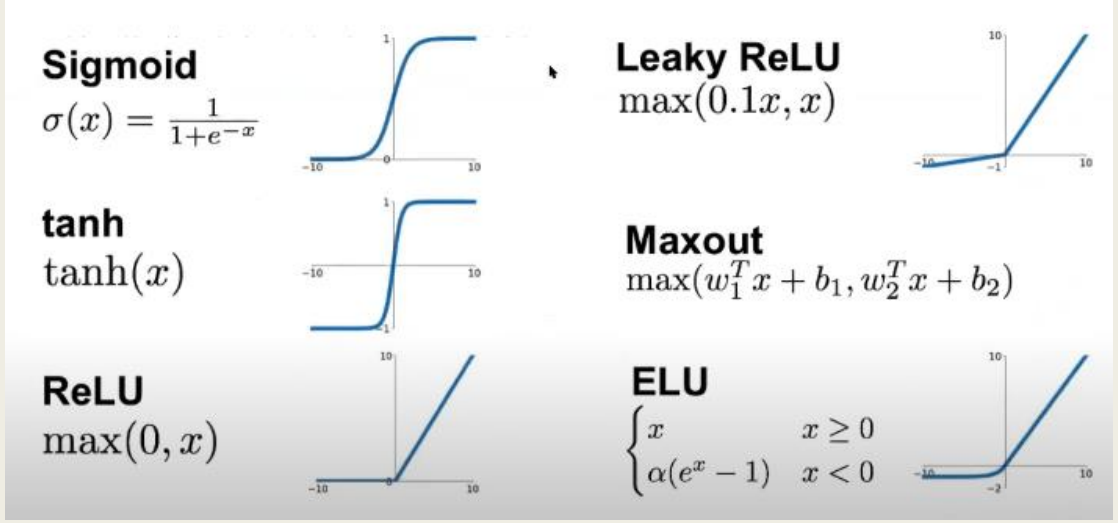


https://subinium.github.io/introduction-to-activation/

Activation function은
신경망의 출력을 결정하는 식 이다.

Linear 함수만으로 non-linear 형식을 만들수가 없어서 activation function이 그 역할을 담당해준다.

그래서 복잡한 신경모델은 복잡한 함수를 사용할 수 있다.



Activation function의 대표적인 종류

MNIST classification

손으로 쓴 숫자들로 이루어진 대형 데이터베이스

Since we have 10 classes (0~9), current problem can be interpreted as **multinomial logistic regression** (**multi-class classification**).
클래스 값이 총 10개의 종류가 있으므로 **multinomial logistic regression** 를 사용해야 한다.

Therefore, we use **softmax** function to handle multiple class output with **cross-entropy** loss function.
그러므로 **softmax** function을 사용해야 한다.

Softmax(소프트맥스)는 입력받은 값을 출력으로 0~1사이의 값으로 모두 정규화하며 출력 값들의 총합은 항상 1이 되는 특성을 가진 함수이다.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K.$$

MNIST classification

Load packages

Load datasets for training & testing

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

# Data loader
# mini batch size
train_loader = DataLoader(dataset=train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=100, shuffle=False)
```

내장된 데이터 셋을 불러온 다음, batch_size (가중치를 조정할 때 사용하는 샘플 수)를 정한다.

테스트 셋은 셔플 할 필요는 없다.

Batch_size같은 경우는 데이터의 수가 커지면 크게 설정하는 것이 유리하다.

MNIST classification

Define model one layer classifier

```
# Define model class
# This model has one hidden layer
class Multinomial_logistic_regression(nn.Module):
    def __init__(self, input_size, output_size):
        super(Multinomial_logistic_regression, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        out = self.fc(x)
        return out

# Generate model
model = Multinomial_logistic_regression(784, 10)  # init(784, 10)
# input dim: 784  / output dim: 10
```

```
model
```

```
Multinomial_logistic_regression(
  (fc): Linear(in_features=784, out_features=10, bias=True)
)
```

In_fearures=784, out_features=10이므로 한 번의 레이어만 지나감.

MNIST classification

Define optimizer

```
# Optimizer define
# optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, momentum=0.9) # learning_rate
# optimizer = orch.optim.Adam(model.parameters(), lr=0.05)
```

파라미터 최적화는 데이터셋에 적용시켜보는 방법이 최고의 방식이다.

다양한 메소드를 이용해서 최적화를 시킬 수 있음
Sgd, momentum, nesterov.. 등등

Sgd가 안정적이라 강의에서 sgd사용

이 강의에서는 옵티마이저에 대해서 자세히 다루지는 않는다.

MNIST classification

Train the model

```
# Loss function define (we use cross-entropy)
loss_fn = nn.CrossEntropyLoss()

#Train the model
total_step = len(train_loader)

for epoch in range(10):
    for i, (images, labels) in enumerate(train_loader):  # mini batch for loop
        # upload to gpu
        images = images.reshape(-1, 28*28).to('cuda')
        labels = labels.to('cuda')

        # Forward
        outputs = model(images)  # forward(images): get prediction
        loss = loss_fn(outputs, labels)  # calculate the loss (crossentropy loss) with ground truth & prediction value

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()  # automatic gradient calculation (autograd)
        optimizer.step()  # update model parameter with requires_grad=True

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, 10, i+1, total_step, loss.item()))
```

크로스엔트로피를 통해서 손실함수를 정의

트레인 로더*배치 사이즈 = 트레인 데이터 셋

Epoch를 통해서 10번 시도

.to('cuda')로 gpu로 학습

Labels는 데이터에 대한 정답.

Backward 를 통해서 최적화

```
Accuracy of the network on the 10000 test images: 92.48 %
```

```
print(model)
```

```
Multinomial_logistic_regression(
  (fc): Linear(in_features=784, out_features=10, bias=True)
)
```

MNIST classification

Test the model

```
[147] # Test the model
      # In test phase, we don't need to compute gradients (for memory efficiency)
      with torch.no_grad():
          correct = 0
          total = 0
          for images, labels in test_loader:
              images = images.reshape(-1, 28*28).to('cuda')
              labels = labels.to('cuda')
              outputs = model(images)
              _, predicted = torch.max(outputs.data, 1)  # classificatoin model -> get the label prediction of top 1
              total += labels.size(0)
              correct += (predicted == labels).sum().item()

          print('Accuracy of the network on the 10000 test images: {} %'.format(100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 92.29 %
```

MNIST classification

Train the model MLP (multi-layer-perceptron)

```
[257] # New model with multi layer
      class NeuralNet(nn.Module):
          def __init__(self, input_size, hidden_size, output_size):
              super(NeuralNet, self).__init__()
              self.fc1 = nn.Linear(input_size, hidden_size)
              self.fc2 = nn.Linear(hidden_size, hidden_size)
              self.fc3 = nn.Linear(hidden_size, output_size)
              self.sigmoid = nn.Sigmoid()  # sigmoid activation function (you can customize)

          def forward(self, x):
              out = self.fc1(x)
              out = self.sigmoid(out)
              out = self.fc2(out)
              out = self.sigmoid(out)
              out = self.fc3(out)
              return out
```

```
# Generate model
model = NeuralNet(784, 20, 10)  # init(784, 20, 10)
# input dim: 784  / hidden dim: 20  / output dim: 10

# Upload model to GPU
model = model.to('cuda')

# Loss function define (we use cross-entropy)
loss_fn = nn.CrossEntropyLoss()

# Define optimizer
# optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, momentum=0.9)
# optimizer = torch.optim.Adam(model.parameters(), lr=0.05)

# Train the model
total_step = len(train_loader)

for epoch in range(10):
    for i, (images, labels) in enumerate(train_loader):  # mini batch for loop
        # upload to gpu
        images = images.reshape(-1, 28*28).to('cuda')
        labels = labels.to('cuda')

        # Forward
        outputs = model(images)  # forward(images): get prediction
        loss = loss_fn(outputs, labels)  # calculate the loss (crossentropy loss) with ground truth & prediction value

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()  # automatic gradient calculation (autograd)
        optimizer.step()  # update model parameter with requires_grad=True

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, 10, i+1, total_step, loss.item()))
```

기존 단일 레이어와 달리 nn.Linear를 추가해주어 multi-layer-perceptron 구현

```
Accuracy of the network on the 10000 test images: 95.18 %
torch.Size([100])
```

```
print(model)
```

```
NeuralNet(
  (fc1): Linear(in_features=784, out_features=20, bias=True)
  (fc2): Linear(in_features=20, out_features=20, bias=True)
  (fc3): Linear(in_features=20, out_features=10, bias=True)
  (sigmoid): Sigmoid()
)
```

MNIST classification

Change the following options to obtain better accuracy

(1) Model configurations

- size of hidden layer units
- number of layers
- type of activation function (e.g., relu, tanh, softplus etc.)

(2) Optimization configurations

- learning rate
- epoch
- type of optimizer
- momentem hyperparameter

THE END !