Week2 학습정리

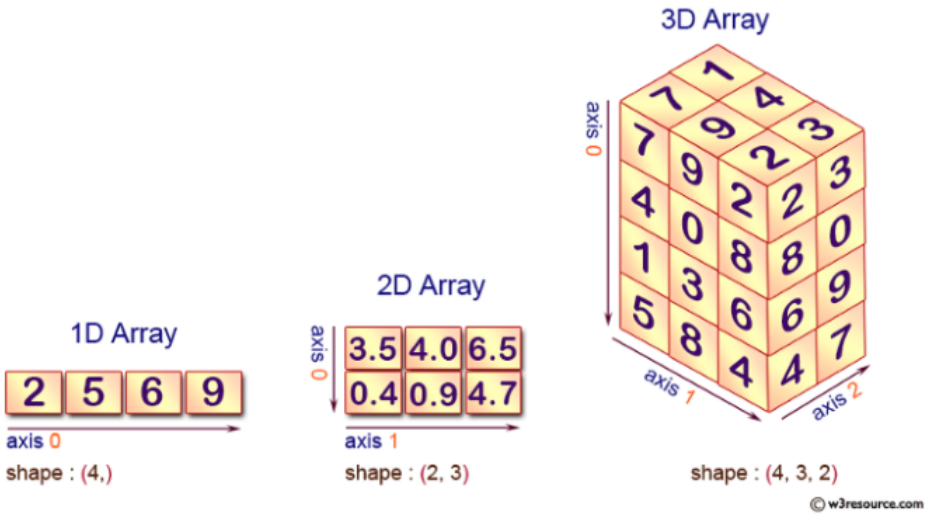
Week2-1

tensor 생성

```
# 기존 list 또는 np.ndarray 를 tensor로 변환
torch.tensor(list or tuple)
torch.from_numpy(numpy.ndarray)

# 특정 shape의 tensor 생성
shape = (2,3,)
torch.zeros(shape) # 0으로 채워진 행렬 생성
torch.ones(shape)  # 1로 채워진 행렬 생성
torch.rand(shape)  # [0,1) 범위의 균등 분포에서 임의의 값 추출
torch.randn(shape) # 정규분포에서 임의 값 추출
torch.randint(10, shape) # [low,high) 범위의 int 값을 임의 추출, 0 이상 10미만
torch.randint(low=0, high=10, size=shape) # [low,high) 범위의 int 값을 임의 추출, 0 이상 10미만

# 1D tensor 생성
torch.arange(1,4,2) # start, end, step
torch.arange(1,4,2).view(1,2,3) # 1면 2행 3열
```

tensor 정보 확인 및 변경

- rank : dimension 개수
- shape : 각 dimension의 value 개수
- axis : 특정 dimension 지칭



The dimensions of a numpy array

더 알아보기 : https://medium.com/byte-sized-code/common-placeholders-in-numpy-arrays-1c3673718f2f

```
x = torch.rand(4,3,2)
print(f"Rank : {len(x.shape)}")
print(f"Shape : {x.shape}")
```

```
x.squeeze() # 랭크 축소
x.unsqueeze(dim=-1) # 기본 값, 열 랭크 추가(shape의 마지막 열에 랭크 추가)
```

tensor 인덱싱

```
x = torch.rand(2,3)

print(f"1st row : {x[0]}")
print(f"1st column : {x[:,0]}")
print(f"last column : {x[:,-1]}")

x[:,-1] = 0 # broadcasting

# broadcasting and masking
x * (x>0.5) # 행렬의 요소가 0.5보다 작은 값은 모두 0으로 대체
```

특정 index 얻기

```
# true가 있는 좌표 반환
(x > 0.5).nonzero()

# axis0, axis1에 각각 행과 열을 나눠서 할당
axis0, axis1 = (x > 0.5).nonzero(as_tuple=True)

# 0.5보다 큰 값을 출력
for i in range(len(axis0)):
    print(x[axis0[i]][axis1[i]])
```

tensor 연산

- 덧셈

```
x = torch.ones(3,2,4) # torch.Size([3, 2, 4])

print(x.sum(dim=0, keepdim=True).shape) # torch.Size([1, 2, 4])

# size 1인 dimension을 squeeze
print(x.sum(dim=0, keepdim=False).shape) # torch.Size([2, 4])
print(x.sum(dim=0, keepdim=True).squeeze().shape) # torch.Size([2, 4])

print(x.sum(dim=1, keepdim=True).shape) # torch.Size([3, 1, 4])
print(x.sum(dim=2, keepdim=True).shape) # torch.Size([3, 2, 1])
```

- 곱셈

```
shape = (2,1)
x = torch.randint(10,shape)
y = torch.randint(10,shape)

# matmul
print(torch.matmul(x,y.T)) # 2x2 행렬
print(x@y.T) # 2x2 행렬

# hadamard product
# [주의] matrix multiplication은 element-wise 곱과 다르다
print(torch.mul(x,y)) # 2x1 행렬
print(x*y) # 2x1 행렬
print(x*y.T) # 2x2 행렬, broadcating 후 연산
```

tensor 병합 및 분리

```
# 병합
x = torch.randint(10,(1,2))
print(x)
# tensor([[2, 8]])

x_concat = torch.cat([x,x,x], dim=1) # 열로 concat
print(f"Concat : {x_concat} shape: {x_concat.shape}")
# Concat : tensor([[2, 8, 2, 8, 2, 8]]) shape: torch.Size([1, 6])

torch.cat([x,x,x], dim=0).shape # 행으로 concat
# torch.Size([3, 2])
```

Week2-2

`torch.nn` 모듈은 텐서 그래프를 생성하는 다양한 함수를 제공한다.

Container

- OFFICIAL DOC

`Module` class

- Neural Network를 생성할 때 반드시 Module 클래스를 부모 클래스로 상속받아야 함
- Module 부모 클래스의 변수 및 메소드 사용 가능 (ex. eval(), train(), parameters(), state_dict(), to())
- forward() 메소드는 모든 자식클래스에서 반드시 **오버라이딩** 해야함
- 출처

```
# named_parameter 메소드와 비슷
for name, state in model.state_dict().items():
    print(f"{name} -> size : {state.shape}")
```

cuda

- cuda와 cudnn 설치되어야 함
- nvida gpu만 사용 가능

`Sequntial` class

- 여러 layer를 연결한 container
- 이전 layer의 output이 다음 layer의 input으로 입력됨 (순차적)
- 출처

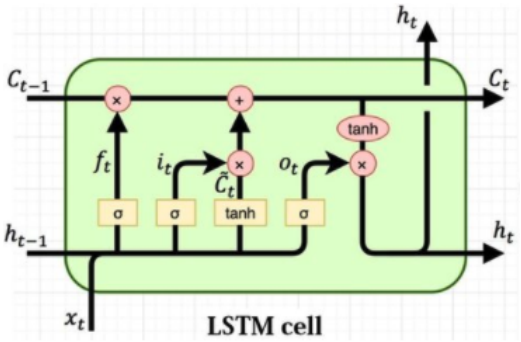
```
model = nn.Sequential(
          nn.Linear(30, 32),
          nn.ReLU(),
          nn.Linear(32,64),
          nn.ReLU(),
          nn.Linear(64,1)
        )
```

Layers

- 출처
- Linear()
  - input @ weight.T + bias

```
model = nn.Linear(20,30)
print(f"W shape: {model.weight.shape}")
print(f"bias shape: {model.bias.shape}")

# W shape: torch.Size([30, 20])
# bias shape: torch.Size([30])

# (1, 20) @ (30, 20).T = output_shape(1, 30) + bias(30)
```

- LSTM()
  - OFFICAL DOCS
  - nn.LSTM(`input_size`, `hidden_size`)
    - nn.LSTM() (`input`, (`h_0`, `c_0`))
    - `input` shape: (문장 길이, 배치 사이즈, 단어 임베딩 사이즈 == input_size)
    - `hidden_size` shape: (lstm 개수 * 레이어 수 , 배치 사이즈, 히든 사이즈 == hidden size)



- 문장 길이 == 토큰 개수 == 7개
- 배치 사이즈 == step별 학습할 개수
- 단어 임베딩 사이즈 == 단어 벡터의 shape

```
ex = "I love coding . Just kidding ."
inputs = ex.split()
print(inputs)

input_embedding = [torch.randn(1, 5) for _ in range(len(inputs))] # 난수 생성

lstm = nn.LSTM(5,5) # (input dim, output dim)
hidden = (
    torch.randn(1,1,5), # (모든 레이어의 lstm 개수, batch size, hidden_size)
    torch.randn(1,1,5),
)

# 한 단어씩 입력
for idx, i in enumerate(input_embedding):
    out, hidden = lstm(i.view(1,1,-1), hidden)
    print(f"{idx+1} word : output shape ({out.shape}) / hidden state shape ({hidden[0].shape})")

# out = ht
# hidden = [ht, Ct]
assert out.detach().equal(hidden[0].detach())

print("---------------------------------")
```

```
# sequence를 입력
input_embedding = torch.cat(input_embedding).view(len(inputs), 1, -1)
print(f"input sequence shape : {input_embedding.shape}")
hidden = (
    torch.randn(1,1,5), # (모든 레이어의 lstm 개수, batch size, hidden_size)
    torch.randn(1,1,5),
)
out, hidden = lstm(input_embedding, hidden)
print(f"output shape : {out.shape}")
print(f"hidden shape : {hidden[0].shape}")

assert out[-1, : , :].detach().equal(hidden[0][-1, :,:].detach())
```

['I', 'love', 'coding', '.', 'Just', 'kidding', '.']
1 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
2 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
3 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
4 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
5 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
6 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
7 word : output shape (torch.Size([1, 1, 5])) / hidden state shape (torch.Size([1, 1, 5]))
---
input sequence shape : torch.Size([7, 1, 5])
output shape : torch.Size([7, 1, 5])
hidden shape : torch.Size([1, 1, 5])

Activation

```
nn.LeakyReLU()
nn.ReLU()
nn.Sigmoid()
nn.GELU()
nn.Tanh()
nn.Softmax()
```

Loss

loss 함수를 최소화하도록 loss의 미분 값을 반영해 파라미터를 업데이트

- MSE

$$\frac{1}{n}\sum_{i=1}^{n}(output_i - target_i)^2$$

- Cross Entropy Loss
  - t : 실제값
  - s : 예측값

$$CE = -\sum_{i}^{C} t_i log(f(s)_i)$$

- Binary Cross Entropy Loss

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1-t_1) log(1-f(s_1))$$

```
# l2 distance loss
nn.MSELoss()

# cross entropy
## multi class
nn.NLLLoss()
nn.CrossEntropyLoss() # softmax O + NLLLoss()

## binary class
nn.BCELoss() # sigmoid 추가 필요
nn.BCEWithLogitsLoss() # sigmoid O + BCELoss()
```

Parallel

• 여러 gpu, 또는 여러 머신에서 입력 데이터를 분산 처리를 가능하게 함

```
# 여러 대의 gpu에 데이터를 로드함
nn.DataParallel()

# 그래디언트 모아서 업데이트 해야함
nn.parallel.DistributedDataParallel()
```

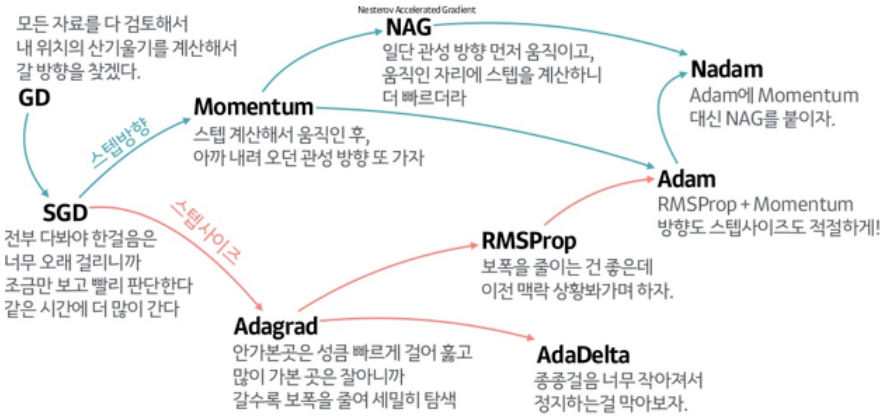
기타

```
# dropout
nn.Dropout()

# normalization
nn.BatchNorm1d()

# gradient clipping
nn.utils.clip_grad()

# weight normalizing
nn.utils.weight_norm()
```

model.zero_grad() vs optimizer.zero_grad()

- model.zero_grad() : 모델의 모든 가중치를 학습하고자 할 때
- optimizer.zero_grad() : 옵티마이저에 내가 학습하고자 하는 가중치만 zero_grad할 경우

Optimizer 정리



Gradient descent를 사용하는 이유

- 손실 함수의 최소값을 구하라면 미분계수가 0인 지점을 찾아야 하는데 함수의 형태가 비선형함수인 경우 미분계수와 그 근을 계산하는 것을 사람 계산하는 것도 어렵고 컴퓨터가 구현하는 것도 어렵기 때문에 상대적으로 구현하기 쉽고 계산효율성이 높은 Gradient descent를 사용한다.

라이브 세션

```
__call__() magic(dunder) function
```

- 인스턴스를 함수처럼 사용하게 해주는 (callable object) 유용한 method

Week2-3

Pytorch DataLoader

- Dataset 클래스는 데이터를 **전처리**하고 dictionary 또는 list 타입으로 변경할 수 있다.
- DataLoader 클래스는 데이터 **1. 셔플 2. 배치화 3. 멀티 프로세스** 기능을 제공한다.
- OFFICAL DOCUMENT

Dataset

- 모든 custom dataset 클래스는 Dataset() 클래스를 상속받아야 함.
- __getitem__()와 __len__() 메소드를 반드시 오버라이딩해야 함.
- DataLoader 클래스가 배치를 만들 때 Dataset 인스턴스의 __getitem__() 메소드를 사용해 데이터에 접근함
- 코드 상의 Dataset 클래스는 string sequence 데이터를 **tokenize** & **tensorize**한다.

Dataloader

- dataset
  - **map-style** dataset(Dataset)
  - iterable style dataset
    - __iter__()
- batch_size : int
- shuffle : bool

- sampler : sample하려면 shuffle false여야 함
  - data index 이터레이터
- collate_fn : 배치마다 특정 함수 적용
  - 예 : 패딩

참고문헌 : https://subinium.github.io/pytorch-dataloader/

Week2-4

https://youtu.be/M0fX15_-xrY

출처 : https://pytorch.org/tutorials/beginner/introyt/autogradyt_tutorial.html

코랩 코드 : https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/83ce1768717d0e03007d32c85f2c63d9/autogradyt_tutorial.ipynb