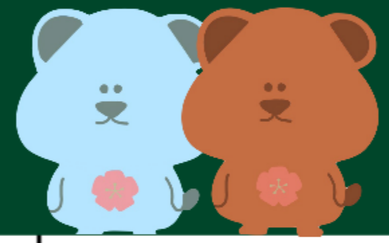
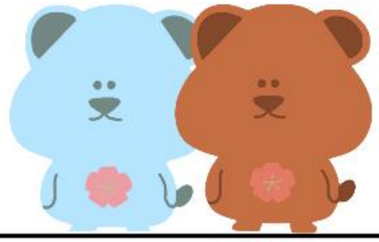


10주차 발표

ML팀

EWHA EURON

목차



EWHA EURON

01. 시계열 문제

EWHA EURON

7.1 시계열 문제

시계열 분석 : 시간에 따라 변하는 데이터 사용하여 추이를 분석하여 추세를 파악하거나 전망을 예측하는 방법론
⇒ 주가 변동, 기온 변화 등

데이터 변동 유형에 따른 구분

1. 불규칙 변동 : 규칙성이 없어 예측 불가, 우연적으로 발생하는 변동 예) 전쟁, 자연재해, 파업

2. 추세 변동 : 장기적인 변화 추세 → 단기에는 찾기 어려운 단점 예) GDP, 인구증가율

3. 순환 변동 : 2-3년 정도 일정한 기간을 주기로 나타나는 변동, 추세 변동에 따라 변동함 예) 경기 변동

4. 계절 변동 : 계절적 영향과 사회적 관습에 따라 1년 주기로 발생

⇒ 시계열 데이터는 트렌드 혹은 분산 변화 유무에 따라 규칙적 시계열과 불규칙적 시계열로 나눌 수 있음

⇒ 다시 말해, 시계열 분석은 특정 기법이나 모델을 통해 불규칙한 시계열 데이터에서 규칙적인 패턴을 찾아내는 것!

✓ 시계열 분석에서는 시간은 독립변수!
✓ ARIMA 등의 모델들이 알려져 있으나 딥러닝을 이용한 결과도 좋은 편

02. AR, MA, ARMA, ARIMA

EWHA EURON

7.2.1 AR(AutoRegressive)

이전 관측 값이 이후 관측 값에 영향을 준다는 아이디어 → 자기 회귀 모델이라고 부르기도 함

$$y_t = \emptyset_0 + \emptyset_1 y_{t-1} + \emptyset_2 y_{t-2} + \ldots + \emptyset_p y_{t-p} + \varepsilon_t$$

① ② ③

① 데이터 현재 시점

② 과거가 현재에 미치는 영향을 나타낸 모수 ϕ에 과거 시점을 곱한 것

③ 오차항(백색잡음)

⇒ p시점을 기준으로 이전 데이터 상태에 의해 현 시점 데이터가 영향을 받는 모형

7.2.2 MA(Moving Average)

시계열을 따라 윈도우 크기만큼 슬라이딩 → 이동 평균 모델이라고 부르기도 함

✓ 트렌드가 변화하는 상황에 적합한 모델

✓ 시간이 지날수록 어떠한 Variable의 평균 값이 지속적으로 감소하거나 증가하는 경향이 생길 수 있음

$$\underbrace{y_t}_{①} = \theta_0 + \underbrace{\varepsilon_t}_{③} + \underbrace{\theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2} + \ldots + \theta_q\varepsilon_{t-q}}_{②}$$

① 데이터 현재 시점

② 매개변수 θ에 과거 시점의 오차를 곱한 것

③ 오차항(백색잡음, 정규 분표에서 도출되는 임의의 값)

⇒ 이전 데이터의 오차에서 현 시점 데이터의 상태를 추론

7.2.3 ARMA(AutoRegressive Moving Average)

AR과 MA을 섞은 모델 → 상태와 오차 두 관점에서 과거의 데이터를 사용

$$y_t = \emptyset_0 + \emptyset_1 y_{t-1} + \emptyset_2 y_{t-2} + \cdots \emptyset_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \ldots + \theta_q \varepsilon_{t-q}$$

✓ AR이나 MA 모델을 하나만 가지고 데이터를 설명하려면 많은 파라미터를 사용해야 할 수 있기 때문에 ARMA를 통해 모수 절약 효과

https://zephyrus1111.tistory.com/142

7.2.4 ARIMA(AutoRegressive Integrated Moving Average)

자기 회귀와 이동 평균을 둘 다 고려하였고, ARMA와 달리 과거 데이터의 선형 관계 + 추세까지 고려한 모델

✓ 추세는 자기 자신(정상 데이터)의 추세만 반응하며 white_noise는 고려하지 않음

✓ 추세 관계는 공적분을 고려한 개념. 예를 들어, X-Y간 cointegration >0 이면 X값이 이전 값보다 증가하면 Y값도 이전 값보다 증가함

✓ 통계 분석 패키지인 statsmodels 라이브러리를 통해 사용 가능
예) ARIMA(p,d,q)
⇒ p=자기회귀 차수, d=차분 차수, q=이동평균 차수

```
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)] ------ train과 test로 데이터셋 분리
history = [x for x in train]
predictions = list()
for t in range(len(test)): ------ test 데이터셋의 길이(13)만큼 반복하여 수행
    model = ARIMA(history, order=(5,1,0)) ------ ARIMA() 함수 호출
    model_fit = model.fit(disp=0)
    output = model_fit.forecast() ------ forecast() 메서드를 사용하여 예측 수행
    yhat = output[0] ------ 모델 출력 결과를 yhat에 저장
    predictions.append(yhat)
    obs = test[t]
    history.append(obs)
    print('predicted=%f, expected=%f' % (yhat, obs)) ------ 모델 실행 결과를 predict
ed로 출력하고, test로 분리해 둔 데이터를 expected로 사용하여 출력
error = mean_squared_error(test, predictions) ------ 손실 함수로 평균 제곱 오차 사용
print('Test MSE: %.3f' % error)
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

✓ 하지만 딥러닝 기반 시계열 모델들이 나오면서, 7.2의 방법들은 잘 사용되지 않는 추세임

03. 순환 신경망(RNN)

EWHA EURON

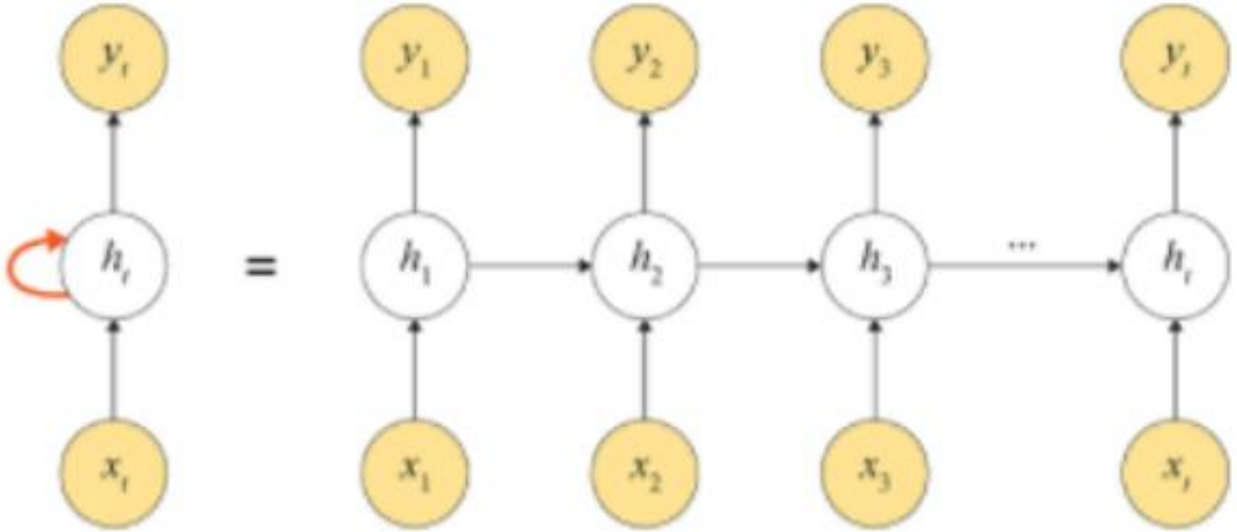
7.3.1 RNN

✓ RNN은 시간적으로 연속성이 있는 데이터를 처리하기 위해 고안된 인공 신경망

✓ 이전 은닉층이 현재 은닉층의 입력이 되는 “순환” 구조

✓ 현재까지 입력 데이터를 요약한 정보인 기억을 가진다는 것이 큰 특징
→ 새로운 입력이 들어올 때마다 기억이 수정되며 최종 기억은 모든 입력을 요약한 정보가 됨

✓ 외부 입력과 자신의 이전 상태를 입력 받아 현재 상태를 갱신함



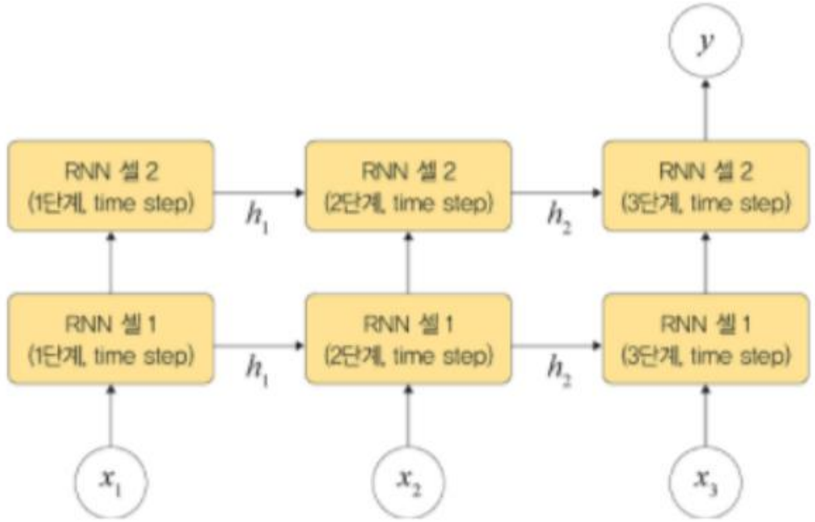
7.3.2 입출력에 따른 유형

① 일대일 : 순환이 없기 때문에 RNN이라고 보긴 힘듦　　예) 순방향 네트워크

② 일대다 : 입력 하나, 출력 다수　　예) 이미지 캡션 : 이미지를 입력하면 그에 대한 설명을 문장으로 출력

③ 다대일 : 입력 다수, 출력 하나　　예) 감성 분석기 : 문장 입력해서 긍부정 출력, 텍스트 범주화

```
self.em = nn.Embedding(len(TEXT.vocab.stoi), embeding_dim) ------ 임베딩 처리
self.rnn = nn.RNNCell(input_dim, hidden_size) ------ RNN 적용
self.fc1 = nn.Linear(hidden_size, 256) ------ 완전연결층
self.fc2 = nn.Linear(256, 3) ------ 출력층
```



▲ 그림 7-6 적층된 다대일 모델

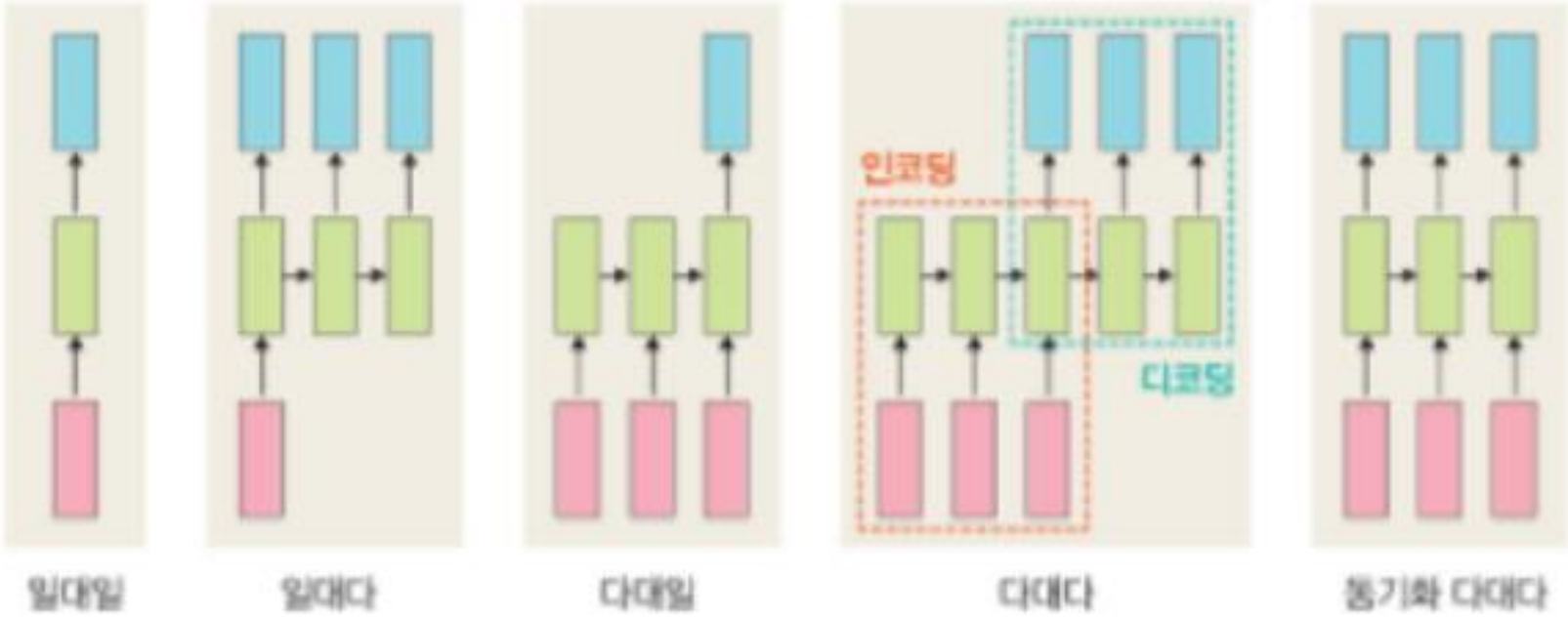
⇒ 다대일 구조에 층을 쌓아 올라 적층 구조를 만들 수 있음

EWHA EURON

7.3.2 입출력에 따른 유형

④ 다대다 : 입력 다수, 출력 다수　예) 자동 언어번역기 (A언어 → B언어)
✓ 텐서플로에서는 keras.layers.SimpleRNN을 사용할 때, "return_sequences=True" 옵션을 통해,
파이토치에서는 시퀀스 투 스퀀스를 이용하여 구현

⑤ 동기화 다대다 : 입출력 다수
　예) 문장 다음에 나올 단어 예측하는 언어 모델, 다른 시간 단계에서 프레임 수준 비디오 분류 혹은
여러 개의 이미지 프레임에 대해 여러 개의 설명이나 번역 형태로 결과를 반환



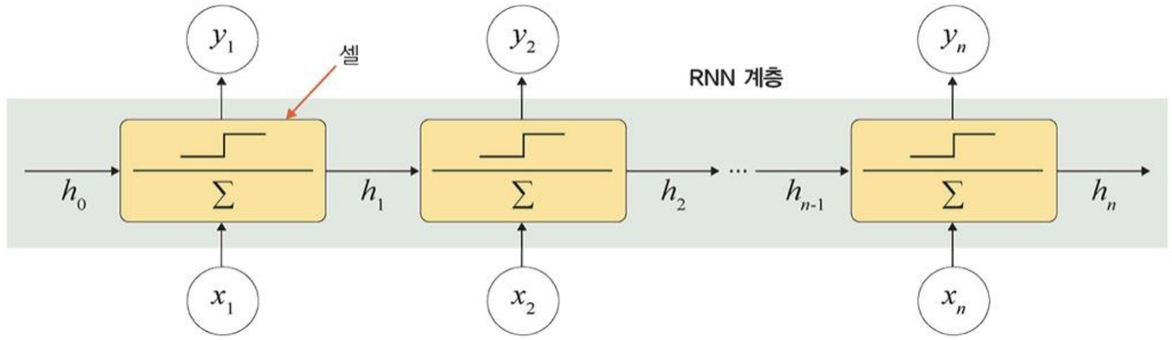
▲ 그림 7-8 RNN 모델 유형

7.3.3 RNN layer & cell

RNN cell : 하나의 단계만 처리 → RNN 계층의 for loop 구문을 갖는 구조
RNN 계층: 입력된 배치 순서대로 모두 처리. 셀을 래핑하여 같은 셀을 여러 단계에 적용

셀은 단일 입력과 과거 상태를 가져와 출력과 새로운 상태를 생성함
1) nn.RNNCell: SimpleRNN 계층에 대응되는 RNN 셀
2) nn.GRUCell: GRU 계층에 대응되는 GRU 셀
3) nn.LSTMCell: LSTM 계층에 대응되는 LSTM 셀

✓ 파이토치에서는 셀과 계층을 분리하여 구현 가능



▲ 그림 7-9 RNN 계층과 RNN 셀

RNN 활용 분야
✓ 자연어 처리 → 자연어의 경우 단어 하나만 안다고 처리될 수 없고, 앞뒤 문맥을 함께 이해해야 해석이 가능하기 때문에 순차적 입력을 받는 RNN 모델과 잘 맞음. 음성 인식, 기계번역 등
✓ 시계열 데이터 처리 → 손글씨, 센서 데이터 등

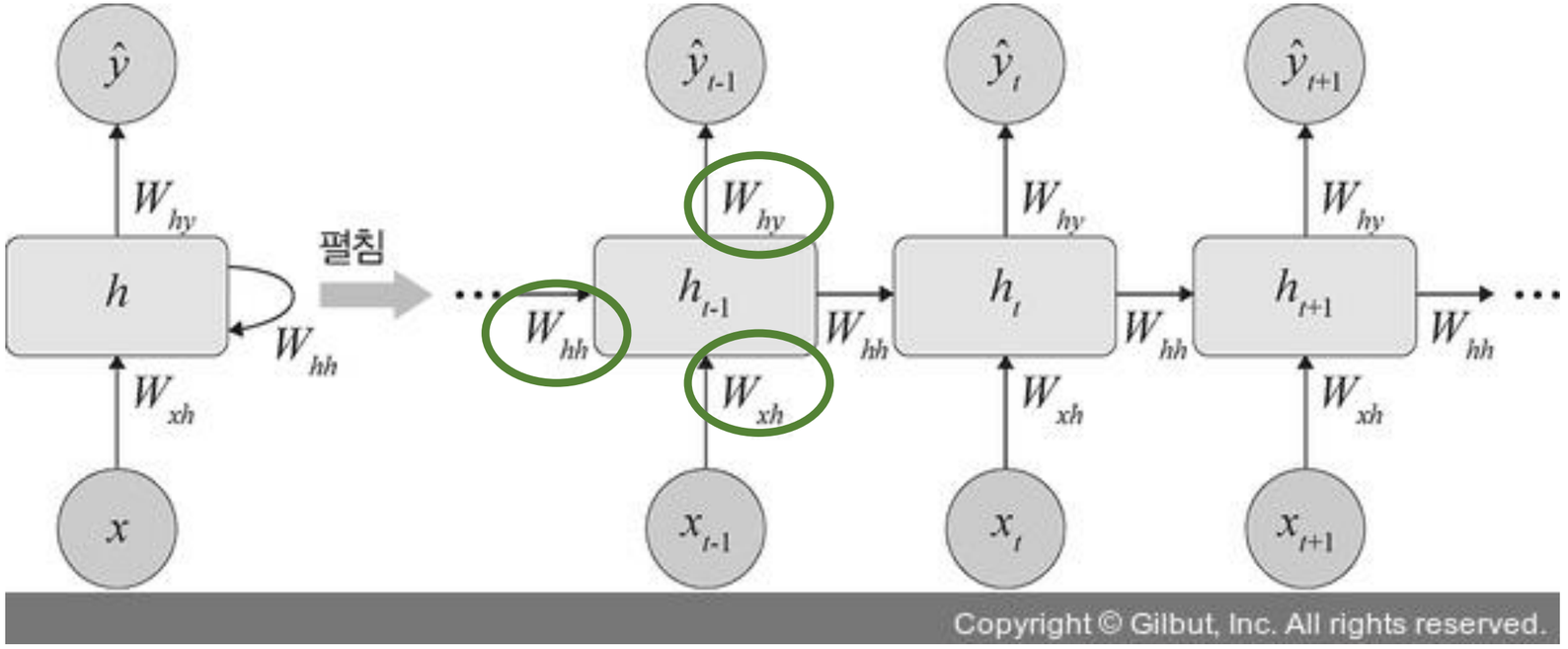
RNN 또한 layer가 많아지면 복잡해지기 때문에 긴 의존 기간을 필요로 하는 학습을 수행할 능력을 가진 LSTM 그리고 Attention, GRU 등 발전시킨 모델들이 나옴

04. RNN 구조

7.4.0 RNN 구조

순환 신경망(RNN)

- memory의 개념이 있음. 은닉층 노드들이 연결되어 이전 단계 정보를 저장할 수 있도록 구성함.
입력층, 은닉층, 출력층 외에 가중치를 세 개 가진다. → 모든 시점에서 가중치가 동일함 .
3개의 가중치의 위치 등을 확인하기.



W(xh): 입력층에서 은닉층으로 전달되는 가중치
W(hh): t시점의 은닉층에서 t+1시점의 은닉층으로 전달되는 가중치
W(hy): 은닉층에서 출력층으로 전달되는 가중치

7.4.0 t단계에서의 RNN 계산

은닉층

이전 은닉층 x 은닉층 -> 은닉층 가중치 + 입력층 -> 은닉층 가중치 x(현재) 입력값

일반적으로 하이퍼볼릭 탄젠트 활성화 함수를 이용한다.

$$h_t = \tanh(\hat{y}_t)$$

$$\hat{y}_t = W_{hh} \times h_{t-1} + W_{xh} \times x_t$$

출력층

심층 신경망과 동일한 계산 방식, softmax 함수를 이용한다.

오차

전방향 학습과 달리 각 단계 마다 오차를 측정한다. 평균제곱오차를 이용하여 측정한다.

역전파

BPTT(backpropagation through time) 3에서 구한 오차를 이용하여 weight와 바이어스를 업데이트 한다. 기울기 소멸 문제가 발생하기도 한다. 보완하고자 생략된 BPTT (truncated BPTT-일정 시점까지만 오류를 역전파 하는 것.)또는 LSTM, GRU를 사용한다.

7.4.0 t단계에서의 RNN 계산

은닉층

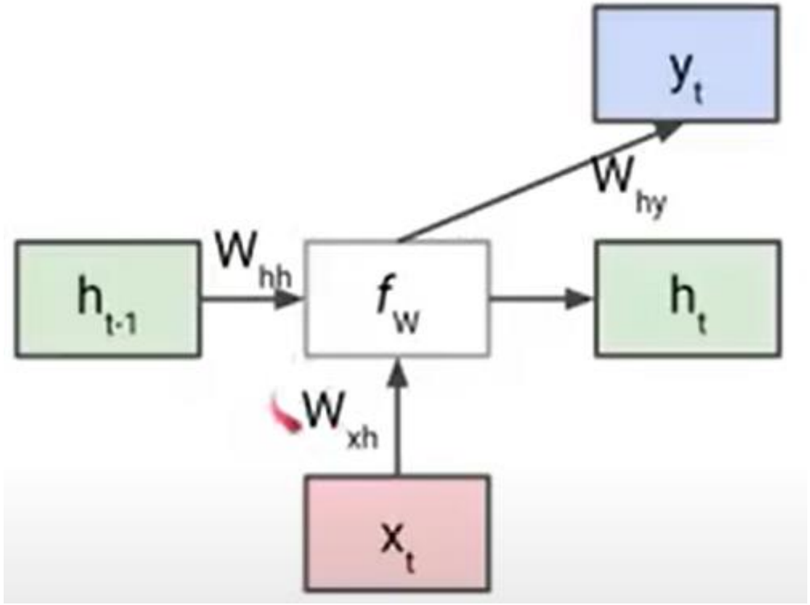
이전 은닉층 x 은닉층 -> 은닉층 가중치 + 입력층 -> 은닉층 가중치 x(현재) 입력값

일반적으로 하이퍼볼릭 탄젠트 활성화 함수를 이용한다.

$$\mathbf{h}_t = f_W(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t)$$

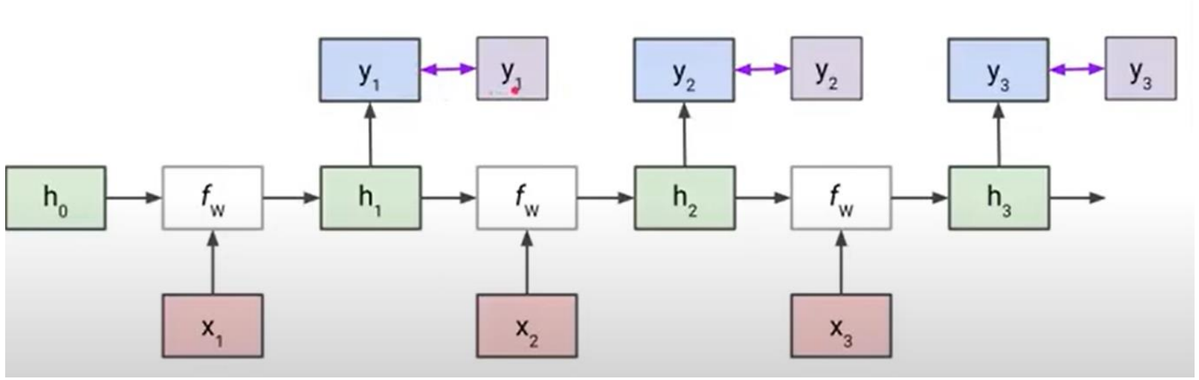
$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t$$



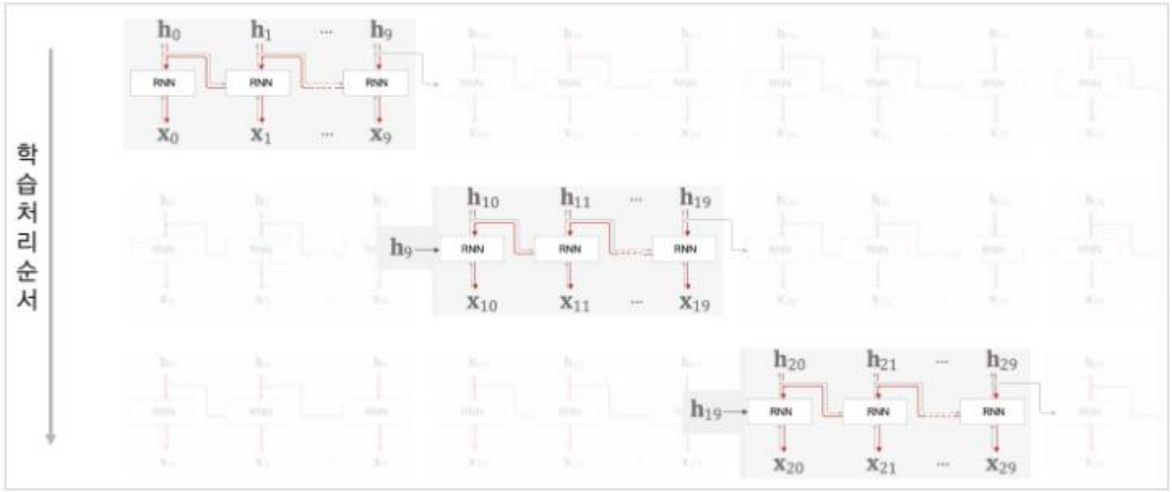
참고: https://www.youtube.com/watch?v=mDcz8wD2a7Y&list=PL0E_1UqNACXD5trR4II4ItJ0dBBt0ztTV&index=21

7.4.0 t단계에서의 RNN 계산

Back propagation의 문제점



Y1과 실제 y1의 차이의 오차가 발생하여 h1, h0를 업데이트
Y2와 실제 y2의 차이의 오차가 발생하여 h2, h1, h0을 업데이트
Y3와 실제 y3의 차이의 오차가 발생하여 h3, h2, h1, h0을 업데이트
→ 모두 같은 w를 사용하는데 왼쪽으로 오차가 멀리 전파될수록 계산량이 많아지고 전파되는 양이 점점 적어지는 vanishing gradient problem이 발생한다.



Truncated-BPTT
큰 시계열 데이터를 취급할 때는 길어진 신경망을 적당한 지점에서 잘라내어 작은 신경망 여러 개로 만든다는 아이디어.
- 순전파의 연결은 그대로 유지하고 역전파의 연결만을 끊어야 한다.
- 데이터를 '순서대로' 입력해야 한다.

역전파가 끊어짐에도 학습이 잘 되는 이유?
-> RNN 계층이 하나의 계층에 대해 시간적으로 나타낸 것이기 때문.
-> 가중치들의 값이 모두 같다.

방식
- 10개 단위로 학습(블록) 하도록 역전파를 끊었다
- 첫 번째 블록에서 내부 순전파를 실행 -> 역전파 과정을 거쳐 기울기를 바탕으로 가중치를 갱신 -> 두번째 블록의 가중치 초기 조건으로 이용

참고: https://luminitworld.tistory.com/74

7.4.0 셀 구현 ~ 전처리

IMDB 데이터셋
영화 리뷰에 대한 데이터 5만 개로 구성. 훈련 2만 5000개, 테스트 2만 5000개
/각각 50% 긍정 리뷰, 부정 리뷰

torchtext(자연어 처리에 사용하는 데이터 로더)
긍정은2, 부정은 1로 레이블링 / 텍스트(훈련 용도)와 레이블(테스트 용도)로 분할
전처리에는 이미지와 다르게 공백 처리, 불필요한 문자 제거 등이 포함된다.

```
import string

for example in train_data.examples:
    text = [x.lower() for x in vars(example)['text']]
    text = [x.replace("<br","") for x in text]
    text = [''.join(c for c in s if c not in string.punctuation) for s in text]
    text = [s for s in text if s]
    vars(example)['text'] = text
```

단어 집합 만들기(build_vocab()) 을 이용한다.
데이터셋에 포함된 단어들을 이용하여 하나의 딕셔너리와 같은 집합을 만드는 것. 중복은 제거된 상태에서 진행한다.
label은 긍정과 부정 + 한개가 더 나옴(unk-사전에 없는 단어)

```
TEXT.build_vocab(train_data, max_size=10000, min_freq=10, vectors=None)
LABEL.build_vocab(train_data)

print(f"Unique tokens in TEXT vocabulary: {len(TEXT.vocab)}")
print(f"Unique tokens in LABEL vocabulary: {len(LABEL.vocab)}")
```

```
Unique tokens in TEXT vocabulary: 10002
Unique tokens in LABEL vocabulary: 3
```

은닉층의 유닛 개수 지정하기
일반적으로 계층의 유닛 개수를 늘리는 것보다 계층 자체에 대한 개수를 늘리는 것이 성능을 위해 더 좋다. 적당한 수를 찾는 것이 어렵기 때문에 실제 필요한 개수보다 더 많은 층과 유닛을 구성하여 조정해 나가는 방식을 이용한다.

7.4.1 RNN 셀 구현

▸ 워드 임베딩 및 RNN 셀 정의

```
#단어집합 만들기
TEXT.build_vocab(train_data, max_size = 10000, min_freq = 10, vectors=None)
LABEL.build_vocab(train_data)
```

• 단어집합 생성에서 vectors = none 으로 지정하여 임베딩 (문자 – 숫자 변환) 이 진행되지 않았기 때문에 nn.Embedding 으로 임베딩 처리를 진행한다.

```
[11]
# 워드 임베딩 및 RNN 셀 정의

class RNNCell_Encoder(nn.Module) :

  def __init__(self, input_dim, hidden_size) :
    super(RNNCell_Encoder, self).__init__()
    self.rnn = nn.RNNCell(input_dim, hidden_size) # RNN 셀 구현

  def forward(self, inputs) : # inputs 는 입력 시퀀스로 (시퀀스 길이, 배치, 임베딩) 형태를 가짐
    bz = inputs.shape[1] # 배치를 가져온다.
    ht = torch.zeros((bz, hidden_size)).to(device) # 배치와 은닉층 뉴런의 크기를 0으로 초기화
    for word in inputs :
      ht = self.rnn(word, ht)
    return ht
```

• input_dim : (batch, 입력데이터칼럼개수) 형태를 가진다.

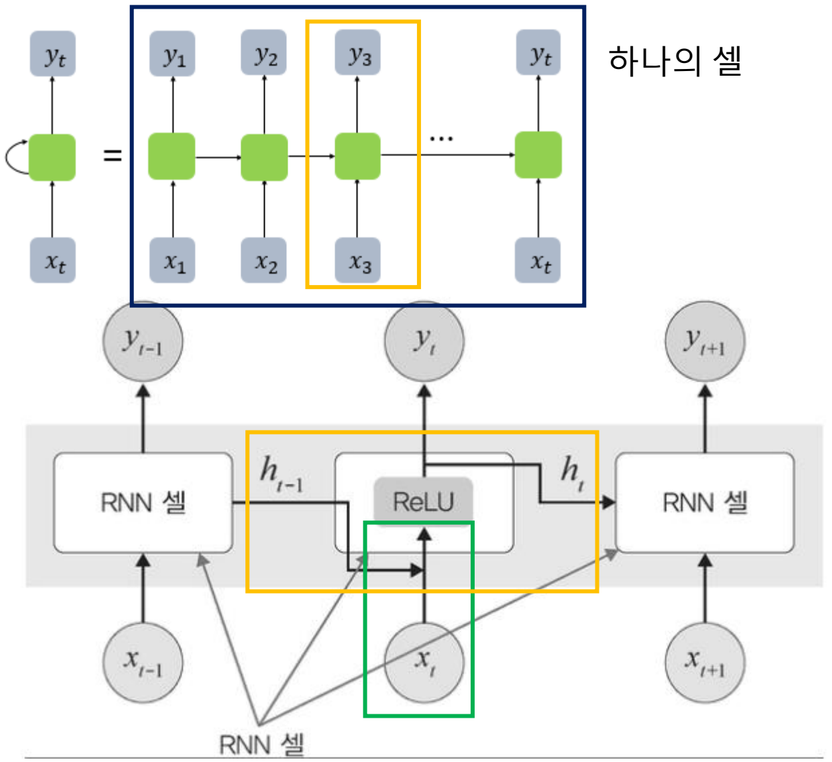
• hidden_size : (batch, 은닉층 뉴런개수) 형태를 가진다.

• ht = rnn(xi, h(t-1))

7.4.1 RNN 셀 구현

▸ 워드 임베딩 및 RNN 셀 정의

```
17 class Net(nn.Module) :
18
19   def __init__(self) :
20     super(Net, self).__init__()
21     self.em = nn.Embedding(len(TEXT.vocab.stoi), embedding_dim) # 임베딩 처리
22     self.rnn = RNNCell_Encoder(embedding_dim, hidden_size)
23     self.fc1 = nn.Linear(hidden_size, 256)
24     self.fc2 = nn.Linear(256,3)
25
26   def forward(self,x) :
27     x = self.em(x)
28     x = self.rnn(x)
29     x = F.relu(self.fc1(x))
30     x = self.fc2(x)
31     return x
```



• nn.Embedding : 임베딩 처리를 위한 구문으로 임베딩을 할 단어 수 (단어집합크기) 와 임베딩할 벡터의 차원을 지정해준다.

7.4.1 RNN 셀 구현

▸ 옵티마이저와 손실함수 정의

```
# 옵티마이저와 손실함수 정의

model = Net()
model.to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
```

- nn.CrossEntropyLoss : 다중분류에 사용되는 손실함수

7.4.1 RNN 셀 구현

▸ 모델 학습을 위한 함수 정의

① 데이터로더에서 데이터를 가져와 모델에 적용
② 손실함수를 적용해 오차를 구함
③ 옵티마이저를 사용해 파라미터를 업데이트함

```python
# 모델 학습

def training(epoch, model, trainloader, validloader) :

  correct = 0
  total = 0
  running_loss = 0

  model.train()  1. 훈련 데이터

  for b in trainloader :

    x,y = b.text, b.label # text 와 label 을 꺼내온다.
    x,y = x.to(device) , y.to(device) # 데이터가 CPU 를 사용할 수 있도록 장치 지정
    y_pred = model(x)
    loss = loss_fn(y_pred, y) # CrossEntropyLoss 손실함수 이용해 오차 계산
    optimizer.zero_grad() # 변화도 (gradients) 초기화
    loss.backward() # 역전파
    optimizer.step() # 업데이트

    with torch.no_grad() : # autograd를 끔으로써 메모리 사용량을 줄이고 연산 속도를 높임
      y_pred = torch.argmax(y_pred, dim =1)
      correct += (y_pred == y).sum().item()  예측한 y 값 저장 및 정확도와 손실값 저장
      total += y.size(0)
      running_loss += loss.item()

  epoch_loss = running_loss / len(trainloader.dataset)
  # 누적된 오차를 전체데이터셋으로 나누어 에포크 단계마다 오차를 구한다.
  epoch_acc = correct/total

  valid_correct = 0
  valid_total = 0
  valid_running_loss = 0
  2. 검증 데이터
  model.eval() # evaluation 과정에서 사용하지 않아야 하는 layer들을 알아서 off 시키도록 하는 함수
  with torch.no_grad() : # evaluation 혹은 validation 에서는 no_grad 를 쓴다.
    for b in validloader :
      x,y = b.text, b.label
      x,y = x.to(device) , y.to(device)
      y_pred = model(x)
      loss = loss_fn(y_pred,y)
      y_pred = torch.argmax(y_pred, dim =1)
      valid_correct += (y_pred == y).sum().item()
      valid_total += y.size(0)
      valid_running_loss += loss.item()

  epoch_valid_loss = valid_running_loss / len(validloader.dataset)
  epoch_valid_acc = valid_correct / valid_total

  print('epoch :', epoch,
        'loss :', round(epoch_loss,3),
        'accuracy : ', round(epoch_acc,3),
        'valid_loss :', round(epoch_valid_loss,3),
        'valid_accuracy :', round(epoch_valid_acc,3)
        )

  return epoch_loss, epoch_acc, epoch_valid_loss, epoch_valid_acc
```

7.4.1 RNN 셀 구현

▸ 모델 학습

```
# 모델 학습 진행

epochs = 5
train_loss = []
train_acc = []
valid_loss = []
valid_acc = []

for epoch in range(epochs) :
  epoch_loss, epoch_acc, epoch_valid_loss, epoch_valid_acc = training(epoch,
                                                                      model,
                                                                      train_iterator,
                                                                      valid_iterator)

  train_loss.append(epoch_loss) # 훈련 데이터셋을 모델에 적용했을 때의 오차
  train_acc.append(epoch_acc) # 훈련 데이터셋을 모델에 적용했을 때 정확도
  valid_loss.append(epoch_valid_loss) # 검증 데이터셋을 모델에 적용했을 때 오차
  valid_acc.append(epoch_valid_acc) # 검증 데이터셋을 모델에 적용했을 때 정확도

end = time.time()
#print(end-start)
```

```
epoch:  0 loss :  0.011 accuracy: 0.495 valid_loss :  0.011 valid_accuracy: 0.507
epoch:  1 loss :  0.011 accuracy: 0.503 valid_loss :  0.011 valid_accuracy: 0.495
epoch:  2 loss :  0.011 accuracy: 0.508 valid_loss :  0.011 valid_accuracy: 0.494
epoch:  3 loss :  0.011 accuracy: 0.513 valid_loss :  0.011 valid_accuracy: 0.499
epoch:  4 loss :  0.011 accuracy: 0.521 valid_loss :  0.011 valid_accuracy: 0.5
--- --------------
```

에포크가 5라 정확도는 낮지만 학습과 검증 데이터셋에 대한
오차가 유사하므로 과적합은 발생하지 않음을 확인해볼 수 있다.

7.4.1 RNN 셀 구현

▸ 모델 예측함수 정의 및 예측 결과 확인

```python
def evaluate(epoch, model, testloader) :
  test_correct = 0
  test_total = 0
  test_running_loss = 0

  model.eval()
  with torch.no_grad() :
    for b in testloader :
      x,y = b.text, b.label
      x,y = x.to(device) , y.to(device)
      y_pred = model(x)
      loss = loss_fn(y_pred, y)
      y_pred = torch.argmax(y_pred, dim=1)
      test_correct += (y_pred == y).sum().item()
      test_total += y.size(0)
      test_running_loss += loss.item()

  epoch_test_loss = test_running_loss/len(testloader.dataset)
  epoch_test_acc = test_correct/test_total

  print('epoch : ', epoch,
        'test_loss : ', round(epoch_test_loss,3),
        'test_accuracy :', round(epoch_test_acc,3))

  return epoch_test_loss, epoch_test_acc
```

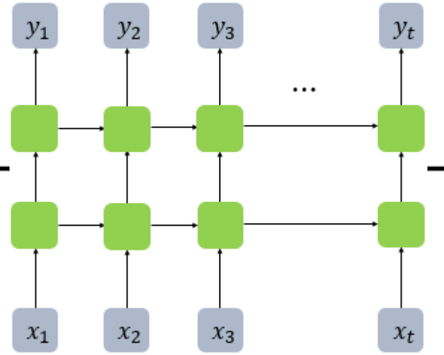
```python
epochs = 5
test_loss = []
test_acc = []

for epoch in range(epochs) :
  epoch_test_loss, epoch_test_acc = evaluate(epoch, model, test_iterator)
  test_loss.append(epoch_test_loss)
  test_acc.append(epoch_test_acc)
```

```
epoch:  0 test_loss :  0.011 test_accuracy: 0.5
epoch:  1 test_loss :  0.011 test_accuracy: 0.5
epoch:  2 test_loss :  0.011 test_accuracy: 0.5
epoch:  3 test_loss :  0.011 test_accuracy: 0.5
epoch:  4 test_loss :  0.011 test_accuracy: 0.5
```

더 높은 정확도를 원한다면 에포크 횟수를 늘리면 된다.

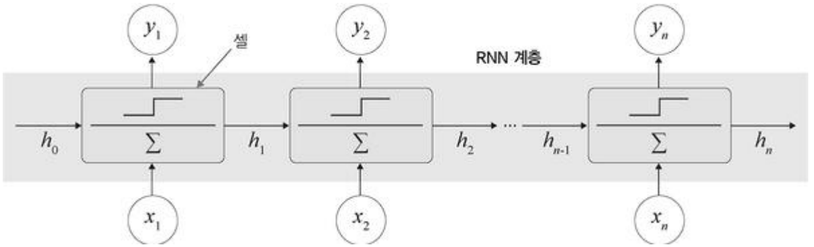
7.4.2 RNN 계층 구현

층이 2개짜리



- RNN cell 구현 과정과 비슷하나 계층의 개수를 지정하는 부분의 차이를 기억!

```python
class BasicRNN(nn.Module) :
  def __init__(self, n_layers, hidden_dim, n_vocab, embed_dim, n_classes, dropout_p=0.2) :
    super(BasicRNN,self).__init__()
    self.n_layers = n_layers # RNN 계층에 대한 개수
    self.embed = nn.Embedding(n_vocab, embed_dim) # 워드 임베딩 적용
    self.hidden_dim = hidden_dim
    self.dropout = nn.Dropout(dropout_p) # 드롭아웃 적용

    self.rnn = nn.RNN(embed_dim, self.hidden_dim, num_layers = self.n_layers, batch_first = True)
    self.out = nn.Linear(self.hidden_dim, n_classes)

  def forward(self,x) :
    x = self.embed(x) # 문자를 숫자/벡터로 변환
    h_0 = self._init_state(batch_size = x.size(0)) # 최초 은닉상태의 값을 0으로 초기화
    x,_ = self.rnn(x, h_0) # RNN 계층
    h_t = x[:,-1,:] # 모든 네트워크를 거쳐 가장 마지막에 나온 단어의 임베딩값 (마지막 은닉상태의 값)

    self.dropout(h_t)
    logit = torch.sigmoid(self.out(h_t))
    return logit

  def _init_state(self, batch_size =1) :
    weight = next(self.parameters()).data # 모델 파라미터 값을 가져와 weight 에 저장
    return weight.new(self.n_layers, batch_size, self.hidden_dim).zero_()
    # 크기가 (계층의 개수, 배치크기, 은닉층의 뉴런개수) 인 은닉상태의 텐서를 생성해 0으로 초기화한 후 반환

```



↪ nn.RNN

- embed_dim : 훈련 데이터셋의 특성(칼럼) 개수
- hidden_dim : 은닉 계층의 뉴런 개수
- num_layers : RNN 계층의 개수

7.4.2 RNN 계층 구현

▸ 학습 및 예측 결과

```
model = BasicRNN(n_layers = 1, hidden_dim = 256, n_vocab = vocab_size, embed_dim = 128, n_classes = n_classes, dropout_p = 0.5)
model.to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
```

```
def train(model, optimizer, train_iter):
    model.train()
    for b, batch in enumerate(train_iter):
        x, y = batch.text.to(device), batch.label.to(device)
        y.data.sub_(1)
        # 레이블이 긍정(2), 부정(1) 로 되어있기 때문에 각각 1과 0으로 값을 바꿔주기 위함


        optimizer.zero_grad()

        logit = model(x)
        loss = F.cross_entropy(logit, y)
        loss.backward()
        optimizer.step()

        if b % 50 == 0:
            print("Train Epoch: {} [{}/{} ({:.0f}%)]₩tLoss: {:.6f}".format(e,
                                                                           b * len(x),
                                                                           len(train_iter.dataset),
                                                                           100. * b / len(train_iter),
                                                                           loss.item()))
```

```
def evaluate(model, val_iter):
    model.eval()
    corrects, total, total_loss = 0, 0, 0

    for batch in val_iter:
        x, y = batch.text.to(device), batch.label.to(device)
        y.data.sub_(1)
        logit = model(x)
        loss = F.cross_entropy(logit, y, reduction = "sum")
        total += y.size(0)
        total_loss += loss.item()
        corrects += (logit.max(1)[1].view(y.size()).data == y.data).sum()

    avg_loss = total_loss / len(val_iter.dataset)
    avg_accuracy = corrects / total
    return avg_loss, avg_accuracy
```

7.4.2 RNN 계층 구현

▸ 학습 및 예측 결과

```
BATCH_SIZE = 100
LR = 0.001
EPOCHS = 5
for e in range(1, EPOCHS + 1):
    train(model, optimizer, train_iterator)
    val_loss, val_accuracy = evaluate(model, valid_iterator)
    print("[EPOCH: %d], Validation Loss: %5.2f | Validation Accuracy: %5.2f" % (e, val_loss, val_accuracy))
```

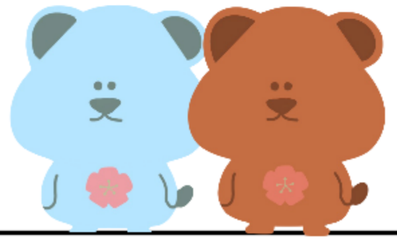
```
Train Epoch: 1 [0/20000 (0%)]   Loss: 0.699639
Train Epoch: 1 [5000/20000 (25%)]       Loss: 0.693163
Train Epoch: 1 [10000/20000 (50%)]      Loss: 0.692305
Train Epoch: 1 [15000/20000 (75%)]      Loss: 0.699170
[EPOCH: 1], Validation Loss:  0.69 | Validation Accuracy:  0.49
Train Epoch: 2 [0/20000 (0%)]   Loss: 0.694949
Train Epoch: 2 [5000/20000 (25%)]       Loss: 0.693578
Train Epoch: 2 [10000/20000 (50%)]      Loss: 0.692129
Train Epoch: 2 [15000/20000 (75%)]      Loss: 0.697591
[EPOCH: 2], Validation Loss:  0.69 | Validation Accuracy:  0.49
Train Epoch: 3 [0/20000 (0%)]   Loss: 0.695081
Train Epoch: 3 [5000/20000 (25%)]       Loss: 0.694416
Train Epoch: 3 [10000/20000 (50%)]      Loss: 0.695030
Train Epoch: 3 [15000/20000 (75%)]      Loss: 0.691506
[EPOCH: 3], Validation Loss:  0.69 | Validation Accuracy:  0.50
Train Epoch: 4 [0/20000 (0%)]   Loss: 0.689116
Train Epoch: 4 [5000/20000 (25%)]       Loss: 0.694588
Train Epoch: 4 [10000/20000 (50%)]      Loss: 0.693115
Train Epoch: 4 [15000/20000 (75%)]      Loss: 0.693649
[EPOCH: 4], Validation Loss:  0.69 | Validation Accuracy:  0.50
Train Epoch: 5 [0/20000 (0%)]   Loss: 0.694274
Train Epoch: 5 [5000/20000 (25%)]       Loss: 0.691786
Train Epoch: 5 [10000/20000 (50%)]      Loss: 0.693263
Train Epoch: 5 [15000/20000 (75%)]      Loss: 0.688134
[EPOCH: 5], Validation Loss:  0.69 | Validation Accuracy:  0.50
```

```
test_loss, test_acc = evaluate(model,test_iterator)
print("Test Loss: %5.2f | Test Accuracy: %5.2f" % (test_loss, test_acc))
```

```
Test Loss:  0.69 | Test Accuracy:  0.54
```

정확도가 그닥 높지 않다. 에포크를 증가시켜보거나, 다른 모델로 변경해본다. 여러 유형의 모델을 적용한 후 가장 결과가 좋은 모델을 선택한다. 또한 하이퍼파라미터 (배치크기, 학습률 등) 를 튜닝해가는 과정이 필요하다.

THANK YOU



EWHA EURON