3️⃣

딥러닝_실습(RNN,Transformor)

RNN

- RNN : 순차적으로 들어오는 데이터에서 hidden state 나 input이 들어오면 순차적으로 학습을 시키는 것
- 종류
  - 일대다 : 1개의 input 여러 개 출력값 ex) image labeling
  - 다대일 : 순차적으로 data classification ex)스팸메일 o,x 댓글 성격 긍정, 부정
  - 다대다 : 여러가지 input에서 다수의 output ex)번역 한글→영어
- 학습과정
  - 입력값과 hidden state가 들어와 출력값과 새로운 hidden state가 다른 node의 입력값과 hidden state 로 들어가는 것이 순차적으로 반복
- input tensor (RNN을 tensor 형태로)
  - 1개의 data가 들어오면 <timestamp>로 이어붙임(n개의 timestamp)
  - batch_size : 몇개의 데이터씩 학습할 것인지 지정해주는 크기
  - |x|=(batch_size, 1, input_size) → input_size : 데이터의 차원 결정 (데이터에 따라 다름)
  - |X|=(batch_size, n, input_size) where X={x1,x2,...,xn}
- Hidden tensor
  - |h|=(batch_size, hidden_size) → hidden_size : 얼마의 공간의 data를 담을 것인지(dimension)
  - |h1:n|=(batch_size,n,hidden_size) where h1:n=[h1;h2;h3;...;hn]
  - =y : 학습이 잘될수록 정확한 data를 예측
- RNN과 LSTM
  - RNN : 입력값이 들어왔을 때 이전 hidden_state도 가져와서 같이 학습을 하는데 예전 data는 점점 지워짐
  - LSTM : 입력값 들어왔을 때 이전 hidden_state도 들어오고 Cell(정보저장하는 - 중요한건 keep, 안 중요한건 버림)도 들어왔다가 나감

실습_RNN

```
# RNN

#function ClickConnect(){
#    console.log("Working");
#    document.querySelector("colab-toolbar-button#connect").click()
#    }setInterval(ClickConnect,1800000)

import numpy as np

timesteps = 10 # 시점의 수. NLP에서는 보통 문장의 길이가 된다.
input_dim = 4 # 입력의 차원. NLP에서는 보통 단어 벡터의 차원이 된다.
hidden_size = 8 # 은닉 상태의 크기. 메모리 셀의 용량이다.

inputs = np.random.random((timesteps, input_dim)) # 입력에 해당되는 2D 텐서

# 은닉 상태의 크기 hidden_size로 은닉 상태를 만듬.
hidden_state_t = np.zeros((hidden_size,)) # 초기 은닉 상태는 0(벡터)로 초기화
```

```
# 은닉 상태의 크기 hidden_size로 은닉 상태를 만듬.# 8의 크기를 가지는 은닉 상태. 현재는 초기 은닉 상태로 모든
print(hidden_state_t)
[0. 0. 0. 0. 0. 0. 0. 0.]
Wx = np.random.random((hidden_size, input_dim))  # (8, 4)크기의 2D 텐서 생성. 입력에 대한 가중치.
Wh = np.random.random((hidden_size, hidden_size)) # (8, 8)크기의 2D 텐서 생성. 은닉 상태에 대한 가중
b = np.random.random((hidden_size,)) # (8,)크기의 1D 텐서 생성. 이 값은 편향(bias).
print(np.shape(Wx))
print(np.shape(Wh))
print(np.shape(b))

total_hidden_states = []

# 메모리 셀 동작
for input_t in inputs: # 각 시점에 따라서 입력값이 입력됨.
  output_t = np.tanh(np.dot(Wx,input_t) + np.dot(Wh,hidden_state_t) + b) # Wx * Xt + Wh * Ht-
  total_hidden_states.append(list(output_t)) # 각 시점의 은닉 상태의 값을 계속해서 축적
  print(np.shape(total_hidden_states)) # 각 시점 t별 메모리 셀의 출력의 크기는 (timestep, output_di
  hidden_state_t = output_t

total_hidden_states = np.stack(total_hidden_states, axis = 0)
# 출력 시 값을 깔끔하게 해준다.

print(total_hidden_states) # (timesteps, output_dim)의 크기. 이 경우 (10, 8)의 크기를 가지는 메모리
```

자연어 처리 개요와 단어 임베딩

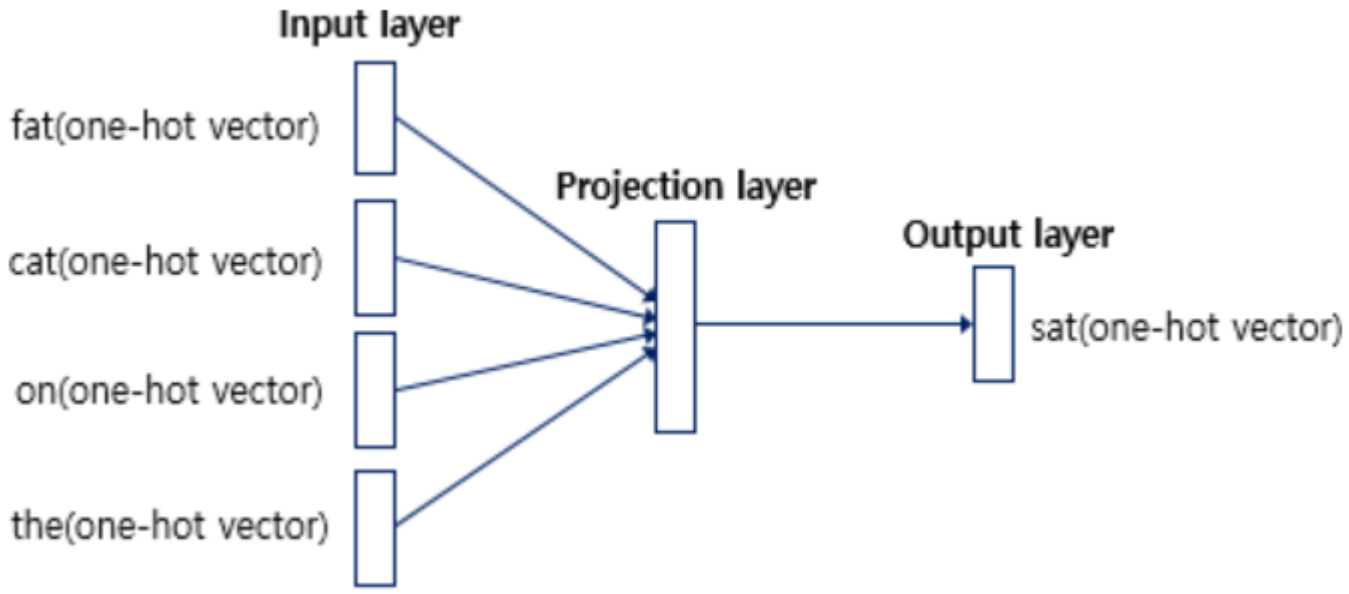
- 자연어 처리 : 인간의 언어 현상을 컴퓨터와 같은 기계를 이용해서 묘사할 수 있도록 연구하고 이를 구현하는 인공지능의 주요 분야 → 컴퓨터가 이해할 수 있도록 문자를 숫자로
- NLP = 자연어 이해(NLU) + 자연어 생성(NLG) - ex) 챗봇(이해시켜 문장 생성)
  - NLU : 단어들의 연관성을 통해 문맥 파악(이해 분야) text→meaning
  - NLG : 문맥에 맞는 단어 생성 (생성 분야) meaning → text
- 자연어 처리 과정
  1. tokenization : 문장마다 중요한 의미로 쪼갬
  2. vocabulary 생성
  3. 정수 encoding
  4. 벡터화 → 다음에 input
- 토큰화 (Tokenization)
  - 일반적으로 딥러닝 자연어 처리에서 Input은 문장 단위
  - tokenization : 주어진 문장을 단어 또는 문자 단위로 자르는 것 (단어, 문자, 형태소)
    - 영어 토큰화 library : NLTK, spaCy
    - 한국어 토큰화 libaray : KoNLPy, khaii
- 단어 표현 방법
  - 컴퓨터가 단어를 이해하는 법 (벡터화) → 단어를 수치로 표현하여 기계가 이해할 수 있도록 변환

  1. 희소 표현 - One-Hot Encoding
     - n=10이면 0,1로 표현하고 문자에 해당하는 곳만 1
     - n이 엄청 커지면? (dimension이 엄청 크면) 비효율적
     - 유사도 측정 불가능
  2. 밀집 표현 - Word Embedding

- 단어의 의미를 수치화한 것 → 단어 벡터간 유사도 측정할 수 있음(연산 가능)
- 원하는 embedding 개수를 지정한 후 차원을 줄일 수 있음
- 입력값은 On-Hot vector로 받음
- 방법

Word2Vec

- 학습 방식
- CBOW(Continuous Bag of Words) : 주변에 있는 단어들을 가지고 중간에 있는 단어 예측 → 정해진 window크기만큼의 주변 단어 활용



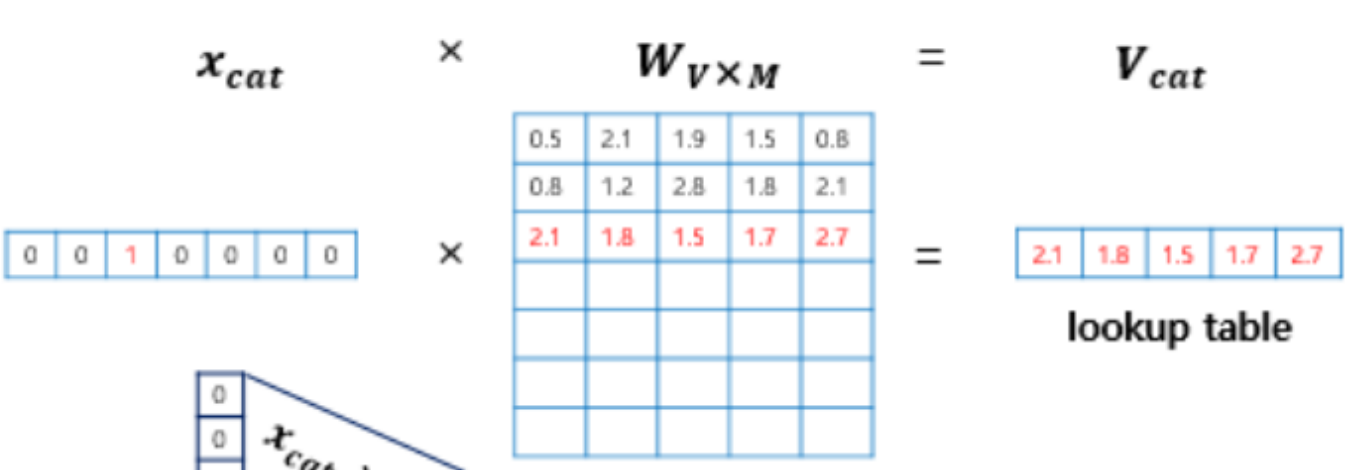
→ input : 윈도우 크기 범위 안에 있는 주변 단어들의 One-Hot vector가 들어감

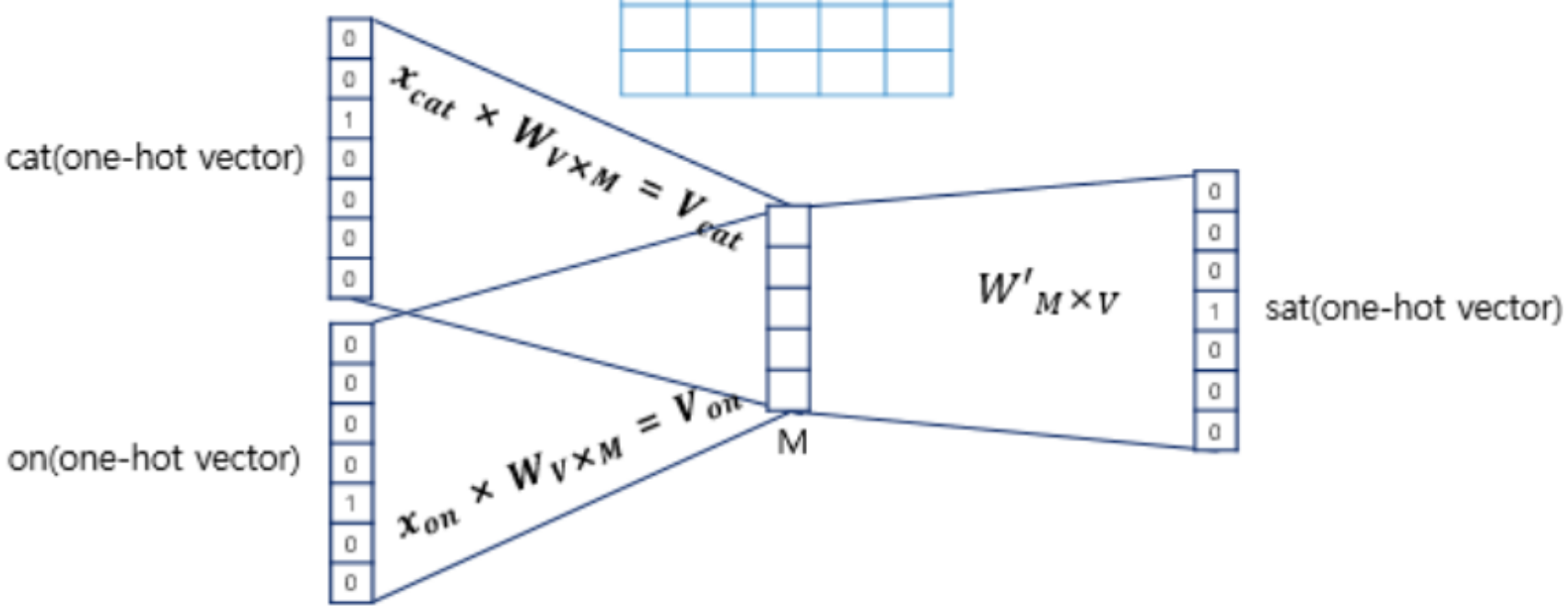
→ projection layer : embedding vector

→ 은닉층이 1개인 얕은 신경망(shallow neural network)

→ Word2Vec의 은닉층은 일반적인 은닉층과는 달리 활성화 함수가 존재하지 않으며 룩업 테이블이라는 연산을 담당하는 층으로 투사층(projection layer)

$$x_{cat} \times W_{V \times M} = V_{cat}$$



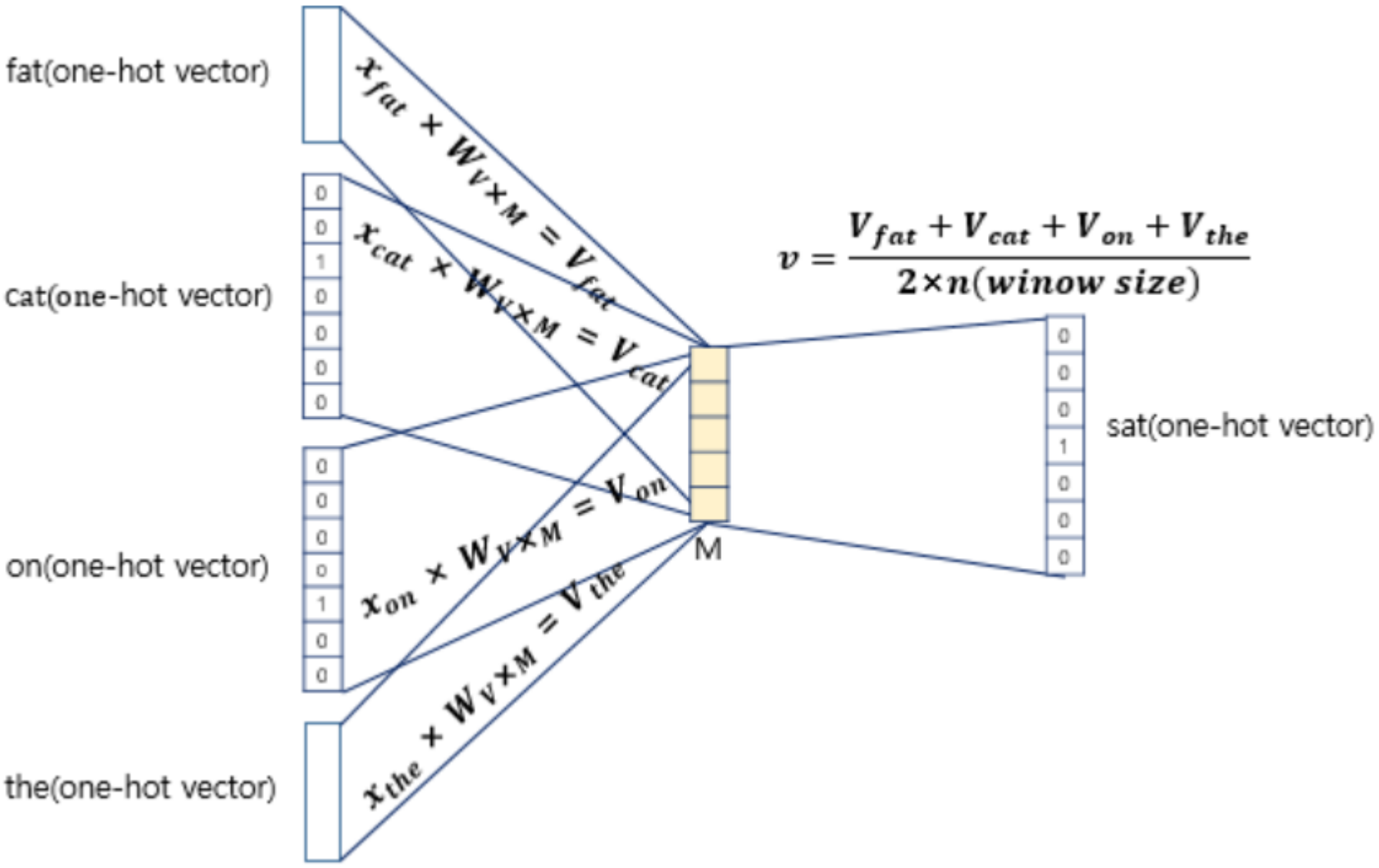


→ M은 projection layer의 크기 (embedding 후 단어 vector 차원)

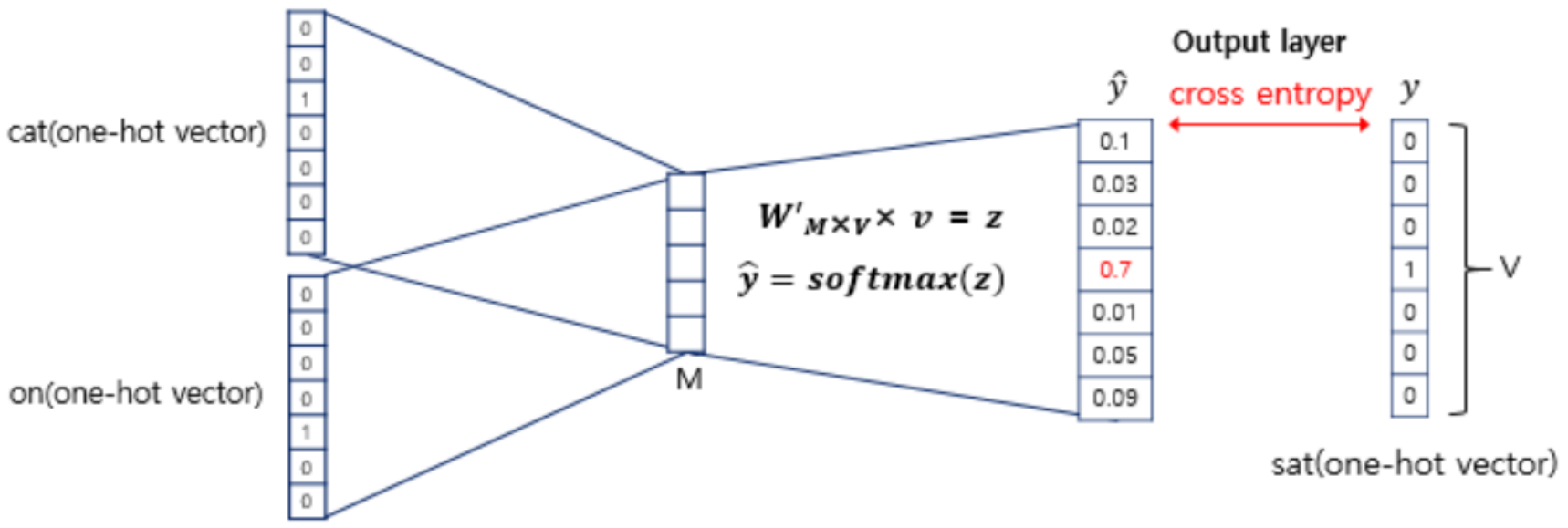
→ V는 단어 집합 크기 → W는 VXM의 크기

→ W와 W’는 전치한 가중치가 아니라 다른 가중치

→ 입력 벡터와 가중치 W 행렬의 곱은 사실 W행렬의 i번째 행을 그대로 읽어오는 것과(lookup) 동일 = lookup table



→ 주변 단어의 원-핫 벡터에 대해서 가중치 W가 곱해서 생겨진 결과 벡터들은 투사층에서 만나 이 벡터들의 평균인 벡터를 구함



→ 이렇게 구해진 평균 벡터는 두번째 가중치 행렬 W'와 곱해집니다. 곱셈의 결과로는 원-핫 벡터들과 차원이 V로 동일한 벡터

→ CBOW는 소프트맥스(softmax) 함수를 지나면서 벡터의 각 원소들의 값은 0과 1사이의 실수로, 총 합은 1 → score vector(얼마나 잘 예측했는지 : yhat)

→중심 단어의 원-핫 벡터를 y , 예측값 yhat

- 이 두 벡터값의 오차를 줄이기위해 CBOW는 손실 함수(loss function)로 크로스 엔트로피(cross-entropy) 함수를 사용

→ 역전파(Back Propagation)를 수행하면 W와 W'가 학습

- 학습이 다 되었다면 M차원의 크기를 갖는 W의 행렬의 행을 각 단어의 임베딩 벡터로 사용하거나 W와 W' 행렬 두 가지 모두를 가지고 임베딩 벡터를 사용

- Skip-gram : 중심에 있는 단어들을 가지고 주변에 있는 단어 예측 (CBOW보다 더 좋은 성능)

-	원-핫 벡터	임베딩 벡터
차원	고차원(단어 집합의 크기)	저차원
다른 표현	희소 벡터의 일종	밀집 벡터의 일종
표현 방법	수동	훈련 데이터로부터 학습함
값의 타입	1과 0	실수

  - Word2Vec 의 한계

    → 단어의 빈도수의 영향을 많이 받음

    → OOV(Out of Vocabulary) 의 처리가 어려움

    → 주변 단어 활용 : 전체 문장 정보 담기 어려움

Skip-gram

- CBOW 알고리즘 역전한 형태 : 중심 단어에서 주변 단어 예측
- Skip-Gram with Negative Sampling → Skip-Gram 향상 버전 → 일부 단어 집합에만 집중

Glove

- Word2Vec의 단점을 보완
- FastText와 함께 많이 쓰이는 Word Embedding 방법 중 하나
- 동시 등장 행렬
  - 행과 열은 전체 단어 집합들로 구성
  - 정해진 윈도우 크기 내에서 i단어의 윈도우 크기 내에서 k 단어가 등장한 횟수를 작성한 행렬
- 동시 등장 확률
  - 중심단어 i가 등장했을 때 주변단어 k가 등장한 횟수를 계산한 조건부 확률
  - P(k|i)=동시등장행렬 i행 k열 값 / 중심단어 i의 행의 모든 값을 더한 값

실습_Word Embedding

- pytorch에서 Embedding vector 사용하는 방법 2가지
  - Embedding layer를 통해 데이터를 처음부터 학습 → nn.Embedding() 사용하여 학습
  - Pre-Trained Word Embedding을 사용 → GloVe, FastText 등 미리 훈련된 embedding vector불러옴

```
# Word Embedding

import torch
import torch.nn as nn
import numpy as np

train_data='you need to know how to code'
word_set=set(train_data.split())
word_set

vocab={token:i for i, token in enumerate(word_set)}
vocab

vocab2={}
for i, token in enumerate(word_set):
```

```
  vocab2[token]=i
vocab2

len(vocab)

embedding_layer=nn.Embedding(num_embeddings=len(vocab),
                             embedding_dim=3)
#num_embeddings : 임베딩을 할 단어들의 개수
#embedding_dim : 임베딩 벡터의 차원 (지정)

lookup_tensor=torch.tensor([vocab['code']],dtype=torch.long)

lookup_tensor

w=embedding_layer(lookup_tensor)

w

embedding_layer.weight

lookup_tensor2=torch.tensor([vocab['you'],vocab['need'],vocab['code']],dtype=torch.long)
w=embedding_layer(lookup_tensor2)

w

## Pre-trained - IMDB dataset (긍정, 부정 비교)

!pip install torch==1.8.1+cpu torchvision==0.9.1+cpu torchaudio==0.8.1 -f https://download.py

!pip install torchtext==0.9.1

from torchtext.legacy import data, datasets
import torch.nn as nn
import torch

TEXT=data.Field(sequential=True, batch_first=True, lower=True)
LABEL=data.Field(sequential=False, batch_first=True, lower=True)

trainset, testset=datasets.IMDB.splits(TEXT,LABEL)

print('훈련 데이터의 크기 : {}'.format(len(trainset)))

print(vars(trainset[0]))

from torchtext.vocab import GloVe #pre_trained library

TEXT.build_vocab(trainset, vectors=GloVe(name='6B',dim=100),max_size=10000,min_freq=10)
LABEL.build_vocab(trainset)

print(TEXT.vocab.stoi) #string to int (길이 맞춰줌, 빈칸 pad로 채워줌

print(LABEL.vocab.stoi)

print('임베딩 벡터의 개수와 차원 : {}'.format(TEXT.vocab.vectors.shape))

print(TEXT.vocab.vectors[0]) #<unk>의 임베딩 벡터값

print(TEXT.vocab.vectors[1415]) #<pad>의 임베딩 벡터값
```

```
print(TEXT.vocab.vectors[10]) #this의 임베딩 벡터값

print(TEXT.vocab.vectors[9999])  #seeing의 임베딩 벡터값

embedding_layer=nn.Embedding.from_pretrained(TEXT.vocab.vectors, freeze=False) #pre_Trained

embedding_layer(torch.LongTensor([10]))
```

- Pre-Trained Embedding Vector
  - 보통 학습에 필요한 데이터가 부족할 때, 이미 학습되어 있는 임베딩 벡터 사용
  - 웹 상에 GloVe, Word2Vec, FastTest등으로 학습된 Vector 사용

LSTM

- 여러 개의 RNN(Wx*x+Wh*h+b=y (기본구조))
- 좀 더 정확하다 butm 메모리 많이 써서 시간 오래 걸림
- Output gate

실습_LSTM

```
# LSTM

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

import os

## data loading

current_path=os.getcwd()
current_path

df=pd.read_csv(current_path+'/samsung_cospi.csv', index_col='Date')

Date : 날짜

High : 시가(당일 고가)

Low : 시가(당일 저가)

Open : 시가(시작 주가)

Close : 시가(마감 주가)

Adj Close : 주식의 분할, 배당, 배분 등을 고려해 조정한 종가 -> 조정 종가 (Target Values)

df.head(5)
```

```
## X,y

X=df.drop(columns='Adj Close')
y=df.iloc[:,5:6]
print(X)

print(y)

## MinMax Scaler

from sklearn.preprocessing import MinMaxScaler #0~1사이로 표준화 ->이상치에 덜 민감

mm=MinMaxScaler()
ss=MinMaxScaler()

y_mm=mm.fit_transform(y)
y_mm

X_ss=ss.fit_transform(X)
X_ss

## train, test

X_train=X_ss[:4800,:]  #4800개의 train data
X_test=X_ss[4800:,:]   #619개의 test data

y_train=y_mm[:4800,:]
y_test=y_mm[4800:,:]

print('Training Shape', X_train.shape,y_train.shape)
print('Testing Shape', X_test.shape, y_test.shape)

#tensor로 변환
X_train_tensors=torch.Tensor(X_train)
X_test_tensors=torch.Tensor(X_test)

y_train_tensors=torch.Tensor(y_train)
y_test_tensors=torch.Tensor(y_test)

#lstm 학습할 수 있는 형태로 변환
X_train_tensors_final=torch.reshape(X_train_tensors,(X_train_tensors.shape[0],1,X_train_tensors
X_test_tensors_final=torch.reshape(X_test_tensors,(X_test_tensors.shape[0],1,X_test_tensors.shap

print('Training Shape',X_train_tensors_final.shape,y_train_tensors.shape)
print('Test Shape',X_test_tensors_final.shape,y_test_tensors.shape)

#device
device=torch.device("cpu")

num_epochs=5000
learning_rate=0.001

input_size=5 #입력값 5개 변수
hidden_size=2
num_layers=1

num_classes=1 #출력값 1개 변수
```

```
class LSTM1(nn.Module):
  def __init__(self, num_classes, input_size, hidden_size, num_layers, seq_length):
     super(LSTM1,self).__init__()
     self.num_classess=num_classes
     self.num_layers=num_layers
     self.input_size=input_size
     self.hidden_size=hidden_size
     self.seq_length=seq_length

     self.lstm=nn.LSTM(input_size=input_size,
                       hidden_size=hidden_size,
                       num_layers=num_layers,
                       batch_first=True)

     self.fc_1=nn.Linear(hidden_size,128)
     self.fc=nn.Linear(128, num_classes)

      self.relu=nn.ReLU()

  def forward(self, x):
    output,(hn,cn)=self.lstm(x)
    #output.shape:[4800,1,2]
    #hn.shape: [1,4800,2]
    #cn.shape:[1,4800,2]

    hn=hn.view(-1, self.hidden_size) #hn=(4800,2)
    out=self.relu(hn)
    out=self.fc_1(out)
    out=self.relu(out)
    out=self.fc(out)

    return out


## Network pararmeter

lstm1=LSTM1(num_classes, input_size, hidden_size, num_layers, X_train_tensors_final.shape[1]).to

loss_function=torch.nn.MSELoss()
optimizer=torch.optim.Adam(lstm1.parameters(), lr=learning_rate)

## Training

for epoch in range(num_epochs):
  outputs=lstm1.forward(X_train_tensors_final.to(device))
  optimizer.zero_grad()

  loss=loss_function(outputs, y_train_tensors.to(device))
  loss.backward()

  optimizer.step()

  if epoch%100==0:
    print('Epoch : %d, loss : %.5f' %(epoch,loss.item()))

## Test & Analysis
```

```
test_predict=lstm1(X_test_tensors_final.to(device))
test_data_predict=test_predict.data.detach().cpu().numpy()

test_data_predict

test_data_predict=mm.inverse_transform(test_data_predict)
y_test=mm.inverse_transform(y_test)

plt.figure(figsize=(10,6))
plt.plot(y_test, label='Actual Data')
plt.plot(test_data_predict, label='Predicted Data')
plt.title('Time-Series Prediction')
```

IMDB-ISTM

- 감성분석 (영화리뷰 datasets) -다 대 일

```
# IMDB-LSTM

import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchtext.legacy import data, datasets
import random

import torchtext
torchtext.__version__

SEED=5
random.seed(SEED)
torch.manual_seed(SEED)

#hyperparameter

BATCH_SIZE=64
lr=0.001
EPOCHS=10

DEVICE=torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

print("GPU?CPU?-{}".format(DEVICE))

#data loading
TEXT=data.Field(sequential=True, batch_first=True, lower=True)
LABEL=data.Field(sequential=False, batch_first=True, lower=True)

trainset, testset=datasets.IMDB.splits(TEXT,LABEL)

print('trainset의 구성 요소 출력 : ', trainset.fields)

print('trainset의 구성 요소 출력 :', testset.fields)

print(vars(trainset[0]))

TEXT.build_vocab(trainset,min_freq=10)
```

```
LABEL.build_vocab(trainset)

vocab_size=len(TEXT.vocab)
n_classes=2

print('단어 집합의 크기 : {}'.format(vocab_size))
print('클래스의 개수 : {}'.format(n_classes))

print(TEXT.vocab.stoi)

#학습용 데이터를 학습셋 80%, 검증셋 20%로 나누기
trainset, valset=trainset.split(split_ratio=0.8)

trainset

print('train 데이터의 크기 : {}' .format(len(trainset)))
print('test 데이터의 크기 : {}' .format(len(testset)))

train_iter,val_iter,test_iter=data.BucketIterator.splits(
    (trainset, valset, testset), batch_size=BATCH_SIZE,
    shuffle=True, repeat=False)

batch=next(iter(train_iter))
print(batch.text.shape)

batch.text

print(batch)

print("[학습셋] : %d [검증셋] : %d [테스트셋] : %d [단어수] : %d [클래스] : %d"
  %(len(trainset), len(valset), len(testset), vocab_size, n_classes))

class BasicLSTM(nn.Module):
  def __init__(self, n_layers, hidden_dim, n_vocab, embed_dim, n_classes, dropout_p=0.2):
     super(BasicLSTM,self).__init__()
     print("Building Basic LSTM mode...")
     self.n_layers=n_layers
     self.embed=nn.Embedding(n_vocab, embed_dim)

     self.hidden_dim=hidden_dim
     self.lstm=nn.LSTM(embed_dim, self.hidden_dim,
                       num_layers=self.n_layers, batch_first=True)

     self.out=nn.Linear(self.hidden_dim, n_classes)
     self.dropout=nn.Dropout(dropout_p)

  def forward(self, x):
    x=self.embed(x)

    output,(hn,cn)=self.lstm(x)
    hn=hn[1]
    hn=self.dropout(hn)
    logit=self.out(hn)
    return logit

def train(model,optimizer,train_iter):
  model.train()
  for b, batch in enumerate(train_iter):
```

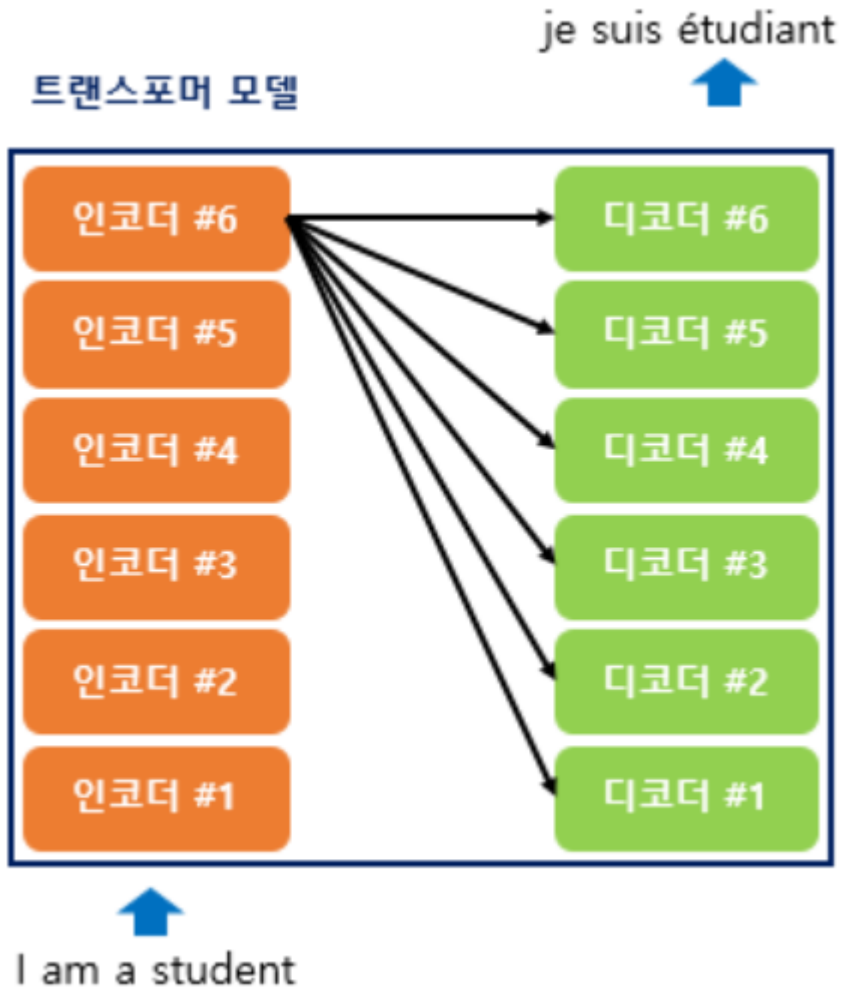
```
    x,y=batch.text.to(DEVICE), batch.label.to(DEVICE)

    y.data.sub_(1)
    optimizer.zero_grad()

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

def evaluate(model,val_iter):

  model.eval()
  corrects, total_loss=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_loss+=loss.item()
    corrects+=(logit.max(1)[1].view(y.size()).data==y.data).sum()
  size=len(val_iter.dataset)
  avg_loss=total_loss/size
  avg_accuracy=100.0*corrects/size
  return avg_loss, avg_accuracy

model=BasicLSTM(2,256,vocab_size,400,n_classes,0.5).to(DEVICE)
optimizer=torch.optim.Adam(model.parameters(),lr=lr)

best_val_loss=None
for e in range(1,EPOCHS+1):
  train(model,optimizer,train_iter)
  val_loss, val_accuracy = evaluate(model, val_iter)

  print("[epoch: %d] val loss:%.2f | val accuracy :%.2f" %(e,val_loss,val_accuracy))

  #검증오차가 가장 적은 최적의 모델을 저장
  if not best_val_loss or val_loss < best_val_loss:
    torch.save(model.state_dict(), './txtclassification.pt')
    best_val_loss=val_loss

model.load_state_dict(torch.load('./txtclassification.pt'))
test_loss, test_acc=evaluate(mode, test_iter)
print('테스트 오차: %5.2f | 테스트 정확도 : %5.2f' %(test_loss, test_acc))
```

Seq 2 Seq

- 번역기에서 대표적으로 사용되는 모델
- 내부가 보이지 않은 커다란 블랙박스에서 점차 확대해가는 방식

Transformer

- 어텐션을 RNN의 보정을 위한 용도로서 사용하는 것이 아니라 어텐션만으로 인코더와 디코더를 만듦
- RNN을 사용하지 않지만 기존의 seq2seq처럼 인코더에서 입력 시퀀스를 입력받고, 디코더에서 출력 시퀀스를 출력하는 인코더-디코더 구조를 유지



Encoder

- seq2seq 구조에서는 인코더와 디코더에서 각각 하나의 RNN이 t개의 시점(time step)을 가지는 구조였다면 이번에는 인코더와 디코더라는 단위가 N개로 구성되는 구조

Decoder

- seq2seq의 디코더에 사용되는 RNN 계열의 신경망은 입력 단어를 매 시점마다 순차적으로 입력받으므로 다음 단어 예측에 현재 시점을 포함한 이전 시점에 입력된 단어들만 참고
- transformer는 문장 행렬로 입력을 한 번에 받으므로 현재 시점의 단어를 예측하고자 할 때, 입력 문장 행렬로부터 미래 시점의 단어까지도 참고할 수 있는 현상이 발생

→ 보안 : 트랜스포머의 디코더에서는 현재 시점의 예측에서 현재 시점보다 미래에 있는 단어들을 참고하지 못하도록 룩-어헤드 마스크(look-ahead mask)를 도입

성능지표

- BLEU Score
  - 기계 번역 결과와 사람이 직접 번역한 결과가 얼마나 유사한지 비교하여 번역에 대한 성능을 측정하는 방법
  - 4gram precision token을4개씩 자르고 한 token씩 옆으로 가면서
  - 문장이 짧을수록 좋은 성능이 나오므로 →penalty를 줌(brevity penalty)
- ROUGE L (score) : LCS(공통된 문장)의 길이 / 총 예측된 값의 길이(token의 길이)
  - True : police killed the gunman.
  - Pred : police kill the gunman.

    → 3/4
- PPL(불순도) : exp(Xentropy) 낮을수록 좋은 성능

실습-독일어→ 영어로 번역

- class 7개
- class : multi position encoder layer, encoder block (encoder들 묶는) decoder layer decoder block transform(전체 묶는) -> init forward(어떻게 뿌려줄지) 함수 선언