

## 2

# [SUM] 쉽고 빠르게 배우는 NLP

- 강의 | [Inflearn] 쉽고 빠르게 배우는 NLP

## Contents

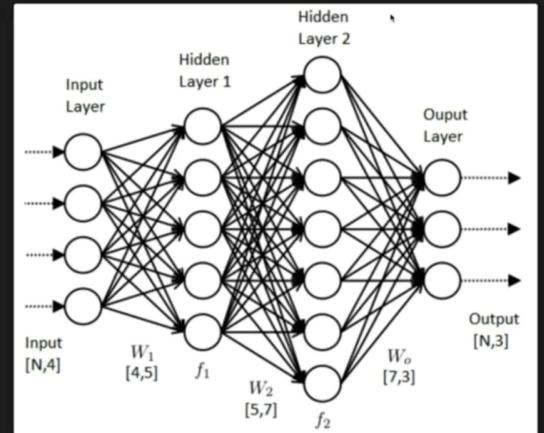
- 1) Section 00 | PyTorch Background
- 2) Section 01 | Background of NLP
- 3) Section 02 | Neural Machine Translation
- 4) Section 03 | NLP Trend

## Section 00 | PyTorch Background

[Deep Learning Algorithm]

### DEEP LEARNING ALGORITHM

- Get Dataset (x: Features, y: Labels)
- Split Train set / Test set
- Training by configuration
  - Forward pass to get model's output
  - Loss(model's output, Labels)
    - ex) Mean Squared Error, Cross Entropy ...
  - Back-Propagation by loss's gradient
  - Update weights
- Evaluate Model's Performance



▶ Deep Learning Model Architecture

- input layer  $[N, 4]$ 
    - $N$  = 데이터 수
    - $4$  = 데이터 크기
  - Weight1 $[4, 5]$  행렬
    - $4$  = 데이터 크기
    - $5$  = 화살표 수
  - Output layer  $[N, 3]$ 
    - $3$  =  $y$ 값의 크기
- 
- feature = input layer
  - Label =  $y$  (output layer 값과 비교를 위한 값)
    - ⇒ 실제 값과 output layer 값 사이의 오차 계산
    - ⇒ 오차를 output layer에 대해 편미분
    - ⇒ 역전파를 통한 기울기 값 전달 ~ Weight 값 업데이트
- \*\* input(x)을 넣었을 때  $y$ 값(최종 결과값)을 정확히 맞출 수 있는 모델 설계

[Index]

# INDEX

- ▶ PyTorch Basic  
([https://pytorch.org/tutorials/beginner/pytorch\\_with\\_examples.html](https://pytorch.org/tutorials/beginner/pytorch_with_examples.html))

- ▶ PyTorch: Tensors
- ▶ PyTorch: Tensors and autograd
- ▶ PyTorch: Defining new autograd functions
- ▶ PyTorch: nn
- ▶ PyTorch: optim
- ▶ PyTorch: Custom nn Modules
- ▶ PyTorch: Control Flow + Weight Sharing

PyTorch Basic ([링크](#))

- Tensors
  - scalar
  - vector
  - metrics (2개 이상의 vector)
  - tensor (고차원적 변수)

```
# Tensors
import torch

dtype = torch.float #data type => 정수형 자료구조
device = torch.device("cpu")
# device = torch.device("cuda:0") # GPU를 이용해서 실행 (0 = GPU index)

# N = 배치 크기
# D_in = 입력값으로 이용되는 벡터의 크기(input dimension)
# H = 은닉층(Hidden Layer)의 노드 개수
# D_out = 출력값 벡터의 크기(output dimension)
N, D_in, H, D_out = 64, 1000, 100, 10

# 무작위로 입력값과 출력값 생성 (pair 값)
# randn => 0 ~ 1 사이 정규분포를 따르는 값을 무작위로 선출
x = torch.randn(N, D_in, device=device, dtype=dtype) # ex) torch.randn(4) -> tensor([-2.1436,  0.9966,  2.3426, -0.6366])
y = torch.randn(N, D_out, device=device, dtype=dtype)

# 가중치값을 무작위로 초기화
w1 = torch.randn(D_in, H, device=device, dtype=dtype) # (64, 1000) (1000, 100) -> (64, 100) (100, 10) -> (64, 10)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6 # 10^(-6)
for t in range(500):
    # Forward(순전파) pass : compute predicted y (모델의 출력값 계산)
    h = x.mm(w1) # mm = 행렬 곱 # (64, 1000) (1000, 100) -> (64, 100)
    h_relu = h.clamp(min=0) # relu fn -> 비선형성 제공 (min = 0)
    # h = (64, 100) h.clamp -> (64, 100)
    y_pred = h_relu.mm(w2) # (64, 100) (100, 10) -> (64, 10) ~ model output

    # 오차값(Loss) 계산 및 출력
    loss = (y_pred - y).pow(2).sum().item() # pow = power(제곱) # 64 -> 1 (scalar)
    if t % 100 == 99: # t = 1, 101 ...
        print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss (오차값에 대한 기울기를 계산하여 역전파 진행)
    grad_y_pred = 2.0 * (y_pred - y) # loss 편미분 결과값
    grad_w2 = h_relu.t().mm(grad_y_pred) # t = transpose
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_w1 = x.t().mm(grad_h)

    # 경사하강법을 통해 가중치를 업데이트합니다.
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

# Results
99 911.7958374023438
199 13.303966522216797
299 0.2643859688420868
399 0.005879917647689581
499 0.00033520121360197663
```

# PYTORCH: TENSORS

## ▶ Using GPUs

- ▶ Numpy cannot utilize GPUs (50x or greater)
- ▶ PyTorch Tensor is conceptually identical to a numpy array
- ▶ PyTorch Tensor can utilize GPUs

### • Tensors and Autograd

: 자동적으로 Back-Propagation을 통한 모델 정확도 향상

```
# Tensors and Autograd

import torch

dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0")

N, D_in, H, D_out = 64, 1000, 100, 10

# 무작위로 입력값과 출력값 생성
# requires_grad = False
# 역전파가 진행하는 과정에 x, y값에 대한 변화값을 계산할 필요가 없다고 설정
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# 가중치값 무작위로 초기화
# requires_grad = True - gradient를 받음
# 역전파가 진행하는 과정에 w1, w2값에 대한 변화값을 계산할 필요가 있다고 설정
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # Forward(순전파) pass : Tensor 연산을 통해 모델의 출력값 계산(predicted_y)
    # 이 단계는 Tensor를 이용하여 Forward pass 를 통해 계산되는 과정과 동일하지만,
    # 역전파 단계를 구현하지 않아도 되기 때문에 중간 과정 생략 가능
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # 오차값(Loss) 계산 및 출력
    # 계산한 결과값은 (1,) 크기의 Tensor 값
    # loss.item()을 통해 Scalar 값으로 출력
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # autograd를 사용하여 역전파 단계 계산 (Define back propagation)
    # 이는 requires_grad = True를 갖는 모든 Tensor에 대해 손실의 변화도 계산
    # w1.grad와 w2.grad는 w1과 w2 각각에 대한 손실의 변화도를 갖는 Tensor
    loss.backward()

    # 1) 경사하강법(gradient descent)을 사용하여 가중치를 수동으로 업데이트
    # torch.no_grad()로 감싸는 이유
    # => 가중치들이 requires_grad = True이지만 autograd에서는 추적할 필요가 없기 때문
    # 2) weight.data 및 weight.grad.data 설정
    # * tensor.data가 tensor의 저장공간을 공유하기는 하지만, 기록을 추적하지 않음
    # 3) torch.optim.SGD 사용
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

    # 가중치 업데이트 이후 수동으로 변화량을 0으로 설정
    w1.grad.zero_()
    w2.grad.zero_()

# Results
99 452.27435302734375
199 3.4509215354919434
299 0.0448567196726799
399 0.0009776445804163814
499 0.00010081531945616007
```

# PYTORCH: TENSORS AND AUTOGRAD

- ▶ Back-Propagation Automatically
  - ▶ Forward function computes output
  - ▶ Backward function receives the gradient of the output

- Defining New Autograd Function

: 새로운 Autograd operator를 쉽게 정의할 수 있음 (Customizing)  
⇒ Function, Activation Function를 바탕으로 업데이트 시 쉽게 구현 가능

```
# Defining New Autograd Function

import torch

class MyReLU(torch.autograd.Function):
    """
        torch.autograd.Function을 상속받아 사용자 정의 autograd Function을 구현하고,
        Tensor 연산을 하는 순전파와 역전파 단계 구현
    """

    @staticmethod
    # self를 활용해 정의해야하는 메소드가 아님
    def forward(ctx, input):
        """
            순전파 단계에서는 입력을 갖는 Tensor를 받아 출력을 갖는 Tensor 반환
            ctx는 컨텍스트 객체(context object)로 역전파 연산을 위한 정보 저장에 사용
            ctx.save_for_backward method >> 역전파 단계에서 사용할 객체 임의로 저장(cache)
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0) #model ouput 값 반환 ~ relu fn

    @staticmethod
    def backward(ctx, grad_output):
        """
            역전파 단계에서는 출력에 대한 손실의 변화도 갖는 Tensor를 받고,
            입력에 대한 손실의 변화도 계산
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input

    dtype = torch.float
    device = torch.device("cpu")
    # device = torch.device("cuda:0")

    N, D_in, H, D_out = 64, 1000, 100, 10

    x = torch.randn(N, D_in, device=device, dtype=dtype)
    y = torch.randn(N, D_out, device=device, dtype=dtype)

    w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
    w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

    learning_rate = 1e-6
    for t in range(500):
        # 사용자 정의 Function을 적용하기 위해 Function.apply 메소드 사용
        # 여기에 'relu'라는 이름 할당
        relu = MyReLU.apply # 기존 -> clamp ~ customizing 가능

        # Forward pass : Tensor 연산을 사용하여 예상되는 y값 계산
        # 사용자 정의 autograd 연산을 사용하여 ReLU 계산
        y_pred = relu(x.mm(w1)).mm(w2)

        # 손실값(Loss)
        loss = (y_pred - y).pow(2).sum()
        if t % 100 == 99:
            print(t, loss.item())

        # autograde를 사용하여 Back-propagation(역전파) 사용
        loss.backward()

        # 경사하강법(gradient descent)을 사용하여 가중치 갱신
        with torch.no_grad():
            w1 -= learning_rate * w1.grad
            w2 -= learning_rate * w2.grad

        # 가중치 업데이트 이후 수동으로 변화량을 0으로 설정
        w1.grad.zero_()
        w2.grad.zero_()
```

```
# Results
99 1162.515869140625
199 22.481172561645508
299 0.7321600914001465
399 0.02739949524402616
499 0.001426524599082768
```

## PYTORCH: DEFINING NEW AUTOGRAD FUNCTIONS

- ▶ We can easily define our own autograd operator
  - ▶ subclass of torch.autograd.Function
  - ▶ Implementing the forward and backward functions

- NN

- 모델 구조 설계 시 쉽게 구현할 수 있도록 함
- 모듈의 집합 (딥러닝 프레임워크에서 자주 이용되는 모델 내포)
  - ⇒ MLP, CNN, RNN, Transformer 등
- Loss Function도 포함

```
# NN => forward pass 단순화 가능

import torch

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# nn 패키지를 사용하여 모델을 순차적 계층(sequence of layers)으로 정의
# w를 직접 정의하지 않더라도 모델 생성
# nn.Sequential은 다른 Module들을 포함하는 Module로, 그 Module들을 순차적으로 적용하여 출력 생성
# 각각의 Linear Module => 선형 함수를 사용하여 입력으로부터 출력 계산, 내부 Tensor에 가중치와 편향 저장
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# nn 패키지에는 널리 사용하는 손실 함수들에 대한 정의도 포함
# 여기에서는 평균 제곱 오차(MSE; Mean Squared Error)를 손실 함수로 사용
loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-4
for t in range(500):
    # Forward pass : 모델에 x를 전달하여 예상되는 y값을 계산
    # Module 객체는 __call__ 연산자를 덮어써(override) 함수처럼 호출하여 사용
    # 이렇게 함으로써 입력 데이터의 Tensor를 Module에 전달해 출력 데이터의 Tensor 생성
    y_pred = model(x)

    # 손실 계산 및 출력
    # 예측한 y와 정답인 y를 갖는 Tensor들을 전달하고, 손실 함수는 손실값을 갖는 Tensor 반환
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # 역전파 단계를 실행하기 전에 변화도를 0으로 변경 (초기화)
    model.zero_grad()

    # Backward pass : 모델의 학습 가능한 모든 매개변수에 대해 손실의 변화도 계산
    # 내부적으로 각 Module의 매개변수는 requires_grad = True 일 때, Tensor sdp wjwkd
    # 이 호출은 모든 모델의 모든 학습 가능한 매개변수의 변화도 계산
    loss.backward()

    # 경사하강법(gradient descent)을 사용하여 가중치 갱신
    # 각 매개변수는 Tensor이므로 이전에 했던 것과 같이 변화도에 접근 가능
    with torch.no_grad():
        for param in model.parameters(): # model은 모든 파라미터를 가지고 있는 객체
            param -= learning_rate * param.grad

# Results
99 1.9351273775100708
199 0.07569249719381332
299 0.0095135233218893409
399 0.00018912901578005403
499 0.00018912901578005403
```

## ▶ PYTORCH: NN

- ▶ Useful for building neural networks
  - ▶ Defines a set of Modules
  - ▶ Defines a set of useful loss functions
    - ▶ nn.L1Loss
    - ▶ nn.MSELoss
    - ▶ nn.CrossEntropyLoss
  - ▶ ...

### • Optim

: Loss 값을 활용해 Weight 값을 업데이트 하는 과정에서 최적화를 위해 사용되는 알고리즘  
⇒ Adam, SGD, Adagrad 등

```
# Optim

import torch

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# nn 패키지를 사용하여 모델과 손실 함수 정의
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss(reduction='sum')

# optim 패키지를 사용하여 모델의 가중치를 갱신할 Optimizer 정의
# 여기서는 Adam 사용 optim 패키지는 다른 다양한 최적화 알고리즘을 포함
# Adam 생성자의 첫번째 인자는 어떤 Tensor가 갱신되어야 하는지 알려줌 (표시)
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for t in range(500):
    # Forward pass : 모델에 x를 전달하여 예상되는 y값 계산
    y_pred = model(x)

    # 손실(Loss) 계산 및 출력
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # 역전파 단계 전에, Optimizer 객체를 사용하여 (모델의 학습 가능한 가중치인)
    # 갱신할 변수들에 대한 모든 변화도를 0으로 변경
    # 이유 => 기본적으로 .backward()를 호출할 때마다 변화도가 버퍼(buffer)에 (덮어쓰지 않고) 누적
    # torch.autograd.backward에 대한 문서 참조
    optimizer.zero_grad()

    # Backward pass : 모델의 매개변수에 대한 손실의 변화도 계산
    loss.backward()

    # Optimizer의 step 함수 호출 시 매개변수 갱신
    optimizer.step()

# Results
99 49.261316833496094
199 0.6050420999526978
299 0.004646726418286562
399 2.7959904400631785e-05
499 6.055332590904072e-08
```

## PYTORCH: OPTIM

- ▶ Provides implementations of commonly used optimization algorithms
  - ▶ `torch.optim.Adam`
  - ▶ `torch.optim.SGD` (Stochastic Gradient Descent)
  - ▶ `torch.optim.Adadelta`
  - ▶ `torch.optim.Adagrad`
  - ▶ ...

▼ Ref

### 02 학습 속도 문제와 최적화 알고리즘

#### ✓ SGD(Stochastic Gradient Descent)



전체 데이터(batch) 대신 일부 조그마한 데이터의 모음인  
미니 배치(mini-batch)에 대해서만 손실 함수를 계산

/\* elice \*/

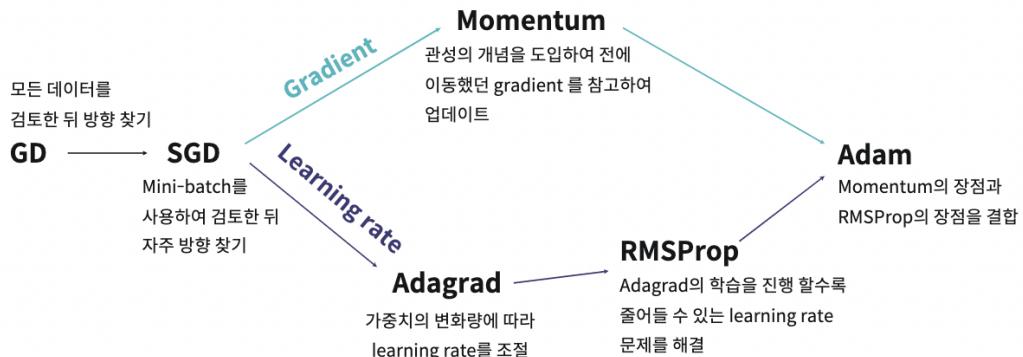
#### ✓ AdaGrad(Adaptive Gradient)

많이 변화하지 않은 변수들은 Learning rate를 크게 하고,  
많이 변화했던 변수들은 Learning rate를 작게 하는 것

과거의 기울기를 제곱해서 계속 더하기 때문에  
학습이 진행될수록 갱신 강도가 약해짐

/\* elice \*/

## ✓ 다양한 최적화 알고리즘 요약



### • Custom NN Modules

: 새로운 모델을 만들거나 모델에 접근할 때 쉽게 활용 가능

```
# Custom NN Modules

import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables. => 2개의 선형 module 할당
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 앞서 정의한 클래스를 생성(instantiating)하여 모델 구성
model = TwoLayerNet(D_in, H, D_out)

# 손실 함수와 Optimizer 생성
# SGD 생성자에 model.parameters()를 호출하면
# 모델의 멤버인 2개의 nn.Linear 모듈의 학습 가능한 매개변수들 포함
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass
    y_pred = model(x)

    # Loss
    loss = criterion(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # 변화도를 0으로 변경, Backward pass 수행, 가중치 갱신
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Results
99 2.6760733127593994
199 0.04300420731306076
299 0.002363199135288596
399 0.0002131618239218369
499 2.1617370293824933e-05
```

## PYTORCH: CUSTOM NN MODULES

- ▶ We can define our own Modules by subclassing nn.Module
  - ▶ Defining a forward which receives input Tensors and produces output

- Control Flow + Weight Sharing

: 모델이 detail한 경우 혹은 detail하게 수정되어야 하는 경우에 사용

```
# Control Flow + Weight Sharing

import random
import torch

class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        생성자에서 Forward pass에서 사용할 3개의 nn.Linear 인스턴스 생성
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        모델의 순전파 단계에서, 무작위로 0, 1, 2 또는 3 중에 하나를 선택하고
        은닉층을 계산하기 위해 여러번 사용한 middle_linear Module 재사용

        각 순전파 단계는 동적 연산 그래프를 구성하기 때문에, 모델의 순전파 단계를
        정의할 때 반복문이나 조건문과 같은 일반적인 Python 제어 흐름 연산자를 사용

        여기에서 연산 그래프를 정의할 때 동일 Module을 여러번 재사용하는 것이
        원래의 안전

        이것이 각 Module을 한 번씩만 사용할 수 있었던 Lua Torch보다 크게 개선된 부분
        """
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 앞서 정의한 클래스를 생성(instantiating)하여 모델 구성
model = DynamicNet(D_in, H, D_out)

# 손실함수와 Optimizer 생성
# 이 이상한 모델을 손상한 확률적 경사 하강법(stochastic gradient decent)으로 학습하기 어려움
# 모멘텀(momentum) 사용
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for t in range(500):
    # Forward pass
    y_pred = model(x)

    # Loss
    loss = criterion(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # 변화도를 0으로 변경, Backward pass 수행, 가중치 갱신
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Results
99 41.61725616455078
199 0.8758949637413025
299 2.5072014331817627
399 1.1256434917449951
499 1.448188304901123
```

## PYTORCH: CONTROL FLOW + WEIGHT SHARING

- ▶ We can implement weight sharing among the innermost layers
  - ▶ By simple reusing the same Module multiple times
  - ▶ When defining the forward pass

### Section 01 | Background of NLP

#### ▼ Background of NLP

# BACKGROUND OF NLP

## BACKGROUND OF NLP

- ▶ Basic Concept of Machine Learning
- ▶ Process of Machine Learning
- ▶ Process of NLP with Deep Learning
- ▶ Role of Train, Valid, Test Dataset
- ▶ Make Corpus with Train Dataset
- ▶ Representation Vector

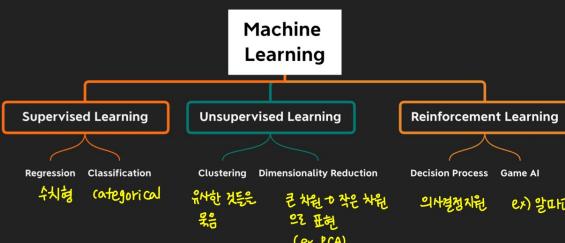
## BASIC CONCEPT OF MACHINE LEARNING



Tiger



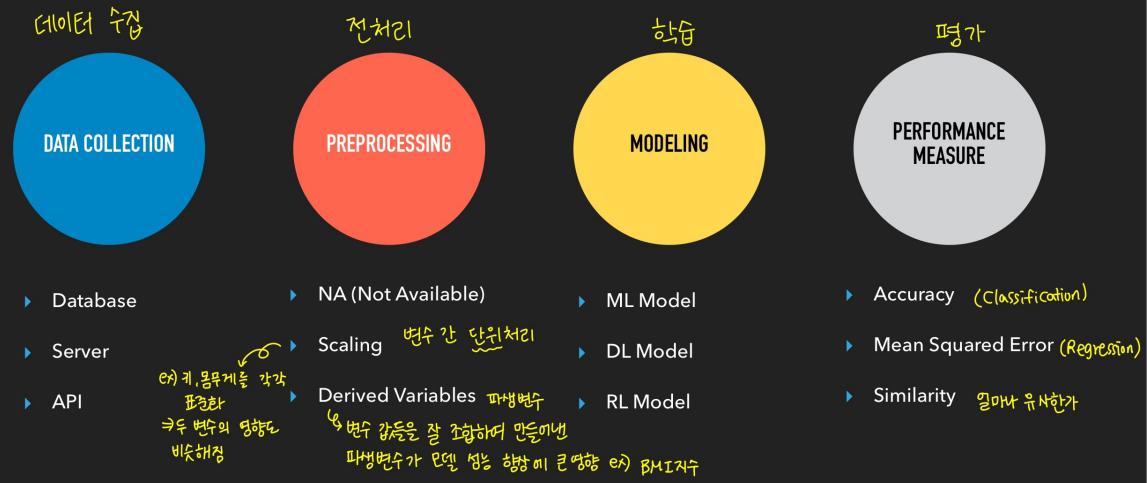
Cat



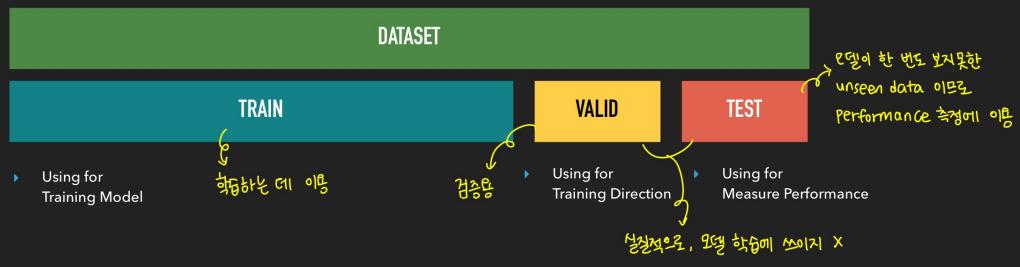
- ▶ Supervised Learning
- ▶ With Labels
- ▶ Unsupervised Learning
- ▶ No Labels
- ▶ Reinforcement Learning
- ▶ With Rewards

잘맞추면 +  
label X  
못맞추면 -

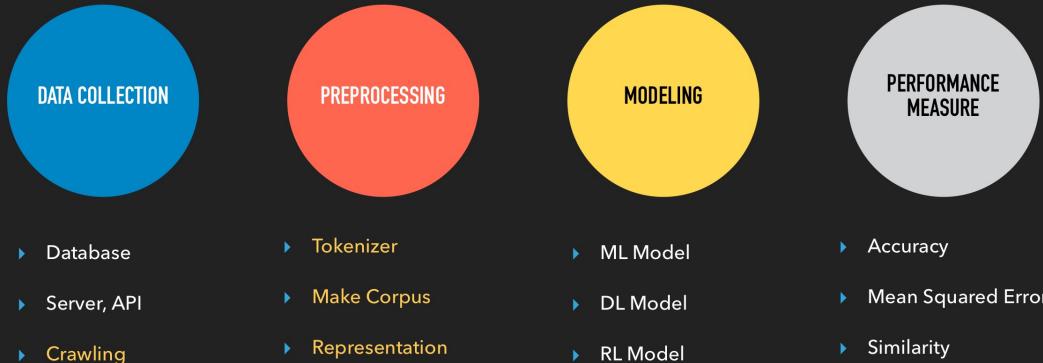
## PROCESS OF MACHINE LEARNING



## ROLE OF TRAIN, VALID, TEST DATASET



## PROCESS OF NLP WITH DEEP LEARNING



## MAKE CORPUS WITH TRAIN DATASET

- ▶ Make Number to Input Natural Language into Computer 자연어를 수치값으로 바꿔주는 과정
- ▶ Make Token to Make a Dictionary 토큰을 만든다. → 단어 사전을 만듦 (Train data set 안에서만 형성)  
    ▶ 'I am a boy' -> 'I', 'am', 'a', 'boy'  
        <sup>0</sup>   <sup>1</sup>   <sup>2</sup>   <sup>3</sup>  
    ▶ 'I am a girl' -> 'I', 'am', 'a', 'girl'  
        <sup>4</sup>
- ▶ Make a Dictionary out of the Tokens in the Train Dataset
- ▶ Replace Words with Numbers using a Dictionary
  - ▶ 'I' : 0, 'am' : 1, 'a' : 2, 'boy' : 3, 'girl' : 4
  - ▶ 'I am a boy' -> [0, 1, 2, 3]
  - ▶ 'I am a girl' -> [0, 1, 2, 4]

## REPRESENTATION VECTOR

- ▶ 'boy' : 3, 'girl' : 4
- ▶ Boy and Girl have similar roles in sentences boyer girl은 문장에서 유사한 역할을 할 가능성↑
- ▶ But 3, 4 can't represent their role enough 3, 4 만으로는 유사한지 알기 어려움.
- ▶ Represent Natural Language with **Vectors** not **scalars** .. vector로 표현해보자.
  - ▶ Word2Vec, Glove, FastText (Word) 여기서는 one-hot vector도 scalar라고 봄
- ▶ Sentence, Document, Sentence Piece ...
  - ▶ 단어 뿐만 아니라 이러한 단위로도 토큰/representation vector 을 만들 수 있음.

### ▼ Representation Vector

# REPRESENTATION VECTOR

## REPRESENTATION VECTOR

- ▶ Background
- ▶ Tokenizer
- ▶ Word2Vec
- ▶ GloVe
- ▶ FastText

## BACKGROUND

Rome	Paris	단어의 개수
Rome		6
		word V
		단어의 개수
Rome = [1, 0, 0, 0, 0, 0, ..., 0]		
Paris = [0, 1, 0, 0, 0, 0, ..., 0]		
Italy = [0, 0, 1, 0, 0, 0, ..., 0]		
France = [0, 0, 0, 1, 0, 0, ..., 0]		

one-hot-encoding

⇒ 단점 : 단어의 유사성을 표현하지 못함

- ▶ Make several tokens from sentences
- ▶ Build a Dictionary of words by indexing each token  
각 토큰에 대해서 numbering 하여 단어사전을 만듦
- ▶ Make a Vector  
(Dimension: Number of Words)  
 $\downarrow V$
- ▶ 1 for the corresponding index with the remaining 0
- ▶ Sparse Vector -> Curse of Dimensionality (차원의 저주)  
↳ 0이 굉장히 많고 1이 1개인 벡터
- ▶ Orthogonal Vector -> Lose of Context Information  
↳ 내적했을 때 0이 되니까  
\* 보통 내적했을 때 값이 크면 유사도가 높다고 봄.

## TOKENIZER

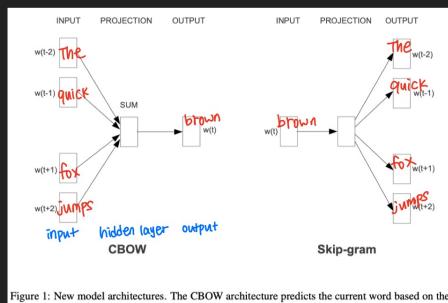
- 띄어쓰기 기준으로 나누면 단어가 잘 나누침
- ▶ English
    - ↑ 불용어 (a, the)를 없애는 방식에 차이가 있음.
  - ▶ SPACY, NLTK
  - ▶ Korean → 형태소라는 특정 때문에 문장을 나누는 기준이 어매할 수 있음.
  - ▶ KoNLPy (Hannanum, Kkma, Komoran, Twitter)
  - ▶ MECAB, KHAIII (형태소 분석기)
- ⑦ \* 어떤 형태소 사전을 쓰는지에 따라 성능이 많이 달라질 수 있음.
- ⑦ 도메인에 따라 형태소로 쪼개면 안되고, 해당 형태소를 고유명사로 치환하는 경우도 있음  
→ 사용자정의 사전 추가

## WORD2VEC

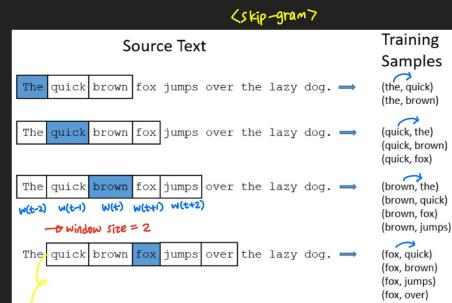
\* 각각이 one-hot-vector로 표현되는데,  
(brown을 표현하는 one-hot-vector) → (The 을 표현하는 one-hot-vector)

연결되어 있는 weight값들을 이용해서 brown을 표현하는 것 → weight들은 같은 역할을 하는 단어들에 대해 유사한 의미를 표현해낼 수 있는 방식으로 학습함 ex: purple

- ▶ Efficient Estimation of Word Representations in Vector Space (Tomas Mikolov et al, 2013)



- ① CBOW: 주변 단어를 통해 중심단어를 학습  
② Skip-gram: 중심단어를 통해 주변 단어를 학습



window size : 중심 단어로부터 주변 단어까지의 범위  
오른쪽으로 한 칸씩 이동하면서 training samples를 만들고  
이 training samples를 바탕으로 neural network 학습  
(weight 값을 통해 해당 단어 표현)

- CBOW / Skip-gram 장단점 비교
  - 연산량 | Skip-gram > CBOW
    - : 여러 word에 대해 prediction을 수행하기 때문에 Skip-gram의 연산량은 CBOW에 비해 많다.
  - 등장 빈도가 낮은 word에 대한 학습 | Skip-gram > CBOW
    - : skip-gram은 input word vector를 평균내지 않고 온전히 사용하기 때문에 등장 빈도가 낮은 word들에 대해 CBOW 대비 train 효과가 크다는 장점이 있다. CBOW에서는 각 word vector들을 평균내서 사용하기 때문에 등장 빈도가 낮은 word들은 제대로 된 학습을 기대하기 힘들다.
- CBOW, Skip-gram 참고자료 | <https://simonezz.tistory.com/35>

## WORD2VEC

⇒ 학습이 완료된 neural network의 weight 값을 이용하여, 특정 2개의 벡터가 유사한 역할을 할 때, 두 벡터의 내적 값이 최대가 되도록 similarity를 반영

\* 문장 내에서 비슷한 역할을 할 때, 두 개의 벡터가 유사도가 높은 값이라는 가정을 바탕으로 한 방법

- ▶ Efficient Estimation of Word Representations in Vector Space (Tomas Mikolov et al, 2013)

- ▶ Objective Function : Learn to reflect similarity between two vectors

- ▶ Subsampling to get Regularize ⇒ Window 안에 나타나는 단어의 빈도수가 높을 경우 (a, the) 과적함이 일어날 수 있으므로 적게 샘플링하겠다는 방법

- ▶ Negative Sampling to get Speed Up

↳ 결국 output vector도 one-hot vector이기 때문에 training samples에 들어가지 않더라도 word 수만큼의 길이의 vector. & 그만큼의 weight 이를 다 계산하기에는 너무 많은 값.

⇒ 전체 단어에 대한 softmax를 계산하는 것이 아니라 window size 안에 있는 단어들과 밖에 있는 단어들은 sampling을 통해 임의로 추려서 그것만 softmax 계산

## GLOVE

- ▶ GloVe: Global Vectors for Word Representation (Jeffrey Pennington et al, 2014)

- ▶ Objective Function : Learn to reflect co-occurrence between two vectors

↳ 한 문장안에 동시 출현하는 단어들은 연관성이 있을 것이다라는 가정에서 출발  
cf) WORD2VEC은 window 밖에 있는 단어는 학습X (단점)

Probability and Ratio	$k = \text{solid}$	$k = \text{gas}$	$k = \text{water}$	$k = \text{fashion}$
$P(k \text{ice})$	$1.9 \times 10^{-4}$	$6.6 \times 10^{-5}$	$3.0 \times 10^{-3}$	$1.7 \times 10^{-5}$
$P(k \text{steam})$	$2.2 \times 10^{-5}$	$7.8 \times 10^{-4}$	$2.2 \times 10^{-3}$	$1.8 \times 10^{-5}$
$P(k \text{ice})/P(k \text{steam})$	8.9	$8.5 \times 10^{-2}$	1.36	0.96

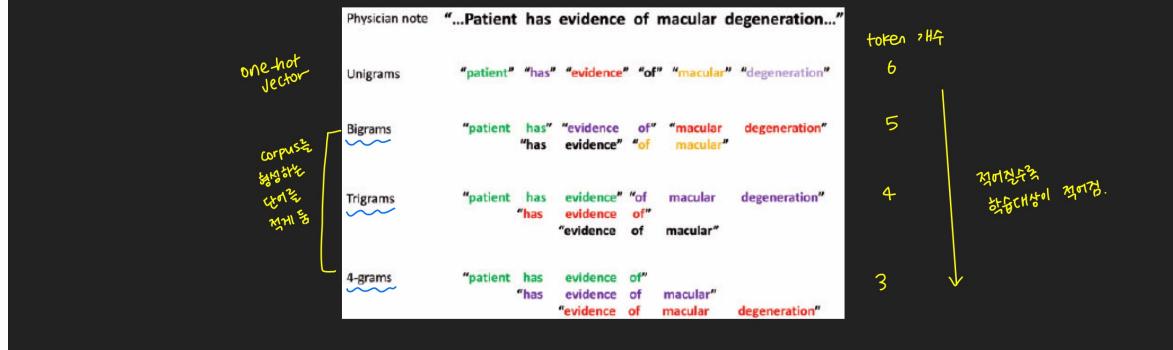
⇒ 학습이 완료된 vector에 대해서는 동의 내적값이 co-occurrence를 반영할 수 있도록 학습 진행

이 2개를 바탕으로 co-occurrence를 계산하고 이를 바탕으로 내적을 진행함으로서 학습이 완료된 벡터의 내적값이 co-occurrence를 반영할 수 있도록 학습을 해보자.

- GLOVE 참고자료 | <https://misconstructed.tistory.com/40>

# FASTTEXT

- ▶ [Bag of Tricks for Efficient Text Classification \(Armand Joulin et al, 2016\)](#)
  - ▶ Objective Function : Learn to reflect **similarity** between two vectors



### [Practice]

- data 불러오기

```
from google.colab import drive
drive.mount('/content/gdrive')
cd/content/gdrive/MyDrive/인프런_쉽고 빠르게 배우는 NLP/torch_nlp_basic-master

import pandas as pd
eng_data = pd.read_csv("./data/IMDB Dataset.csv")

eng_data.shape
eng_data.head(5)

▼ data download link

KOREAN : https://github.com/e9t/nsmc

ENGLISH : https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews/kernels
```

- #### • 전처리 함수 만들기

```

from bs4 import BeautifulSoup
from nltk.tokenize.toktok import ToktokTokenizer
import re
import nltk
# nltk.download("stopwords")

def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")           # html 구조들을 제거할 수 있도록 넣음
    return soup.get_text()

def remove_between_square_brackets(text):             # 정규표현식에 해당되는 내용들을 제거
    return re.sub('\'[^\']*\'' , '' , text)

def remove_special_characters(text, remove_digits = True): # 정규표현식에 해당되는 내용들을 제거
    pattern=r'[^a-zA-Z0-9\$]' 
    text=re.sub(pattern,'',text)
    return text

def remove_stopwords(text, is_lower_case = False):      # 불용어 (자주 나오는 단어 a, the, she, he 등) 제거
    tokenizer = ToktokTokenizer()
    stopword_list = nltk.corpus.stopwords.words('english') # nltk 페키징에 있는 영어에 대한 불용어 사전을 가져옴
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]          # 양 끝에 공백이 있으면 제거
    if is_lower_case:
        filtered_tokens = [token for token in tokens if token not in stopword_list] # 소문자와 대문자 구별
    else:
        filtered_tokens = [token for token in tokens if token.lower() not in stopword_list] # 소문자와 대문자 구별
    filtered_text = ' '.join(filtered_tokens)            # 토큰 내부에 공백이 있으면 제거
    return filtered_text

def text_cleaning(text):
    text = strip_html(text)
    text = remove_between_square_brackets(text)
    text = remove_special_characters(text, remove_digits = True)
    text = remove_stopwords(text, is_lower_case = False)
    return text

```

- 전처리 실행

```
# html 구조 없애는 예시
eng_data["review"][[2]
strip_html(eng_data["review"][[2]])

# 전처리 실행
eng_data["review"] = eng_data["review"].apply(text_cleaning)
eng_data["review"][[1]
```

- 문장을 index로 바꿔주는 과정

```
from collections import Counter
vocab_lst2 = [y for x in vocab_lst for y in x] # 이중 list 를 flatten하는 과정
Counter(vocab_lst2).most_common(10) # 가장 많이 나온 단어 10개

vocab_lst3 = list(Counter(vocab_lst2).keys())
vocab_to_index = {word: index for index, word in enumerate(vocab_lst3)} # word가 key고, index가 value인 단어사전
index_to_vocab = {index: word for index, word in enumerate(vocab_lst3)} # index key고, word가 value인 단어사전
## index를 딥러닝모델의 input으로 넣었을 때 output도 index로 나오는데, 그 index를 단어사전에 매칭시키면, 원래 text를 알 수 있음

index_to_vocab
vocab_to_index
len(vocab_to_index)

# 문장을 index로 바꿔주는 과정
result = [[vocab_to_index[word] for word in y] for y in vocab_lst] # 각 문장별로 접근하고, 그 문장안에 토큰별로 접근해서 vocab_to_index에 매칭시켜서 그 index를 리스트에 넣음

result[10]
```

▼ index\_to\_vocab

```
{0: 'One',
 1: 'reviewers',
 2: 'mentioned',
 3: 'watching',
 4: 'I',
 5: 'Oz',
 6: 'episode',
 7: 'youll',
 8: 'hooked',
 9: 'right',
 10: 'exactly',
 11: 'happened',
 12: 'meThe',
 13: 'first',
 14: 'thing',
 15: 'struck',
 16: 'brutality',}
```

▼ vocab\_to\_index

```
{'One': 0,
 'reviewers': 1,
 'mentioned': 2,
 'watching': 3,
 'I': 4,
 'Oz': 5,
 'episode': 6,
 'youll': 7,
 'hooked': 8,
 'right': 9,
 'exactly': 10,
 'happened': 11,
 'meThe': 12,
 'first': 13,
 'thing': 14,
 'struck': 15,
 'brutality': 16,}
```

▼ result[10]

▶ result[10]

```
[624,
 625,
 184,
 626,
 586,
 627,
 628,
 102,
 629,
 630,
 194,
 631,
 632,
 13,
 633,
 92,
 537,
 299,
 634,
 635,
 398,
 636,
 629,
 537,
 637,
 638,
 639,
 310,
 640,
 78,
 641,
 92,
 284,
 233,
 642,
 643,
 644,
 645,
 310,
 81,
 84,
 646,
 647,
 648,
 649,
 325,
 650,
 651,
 652,
 373,
 653]
```

### [Vector Representations]

- 여기서는 torch를 통해 Word2Vec 모델을 만드는 걸 보여줌
- 실전에서는 gensim으로 실습하면 됨

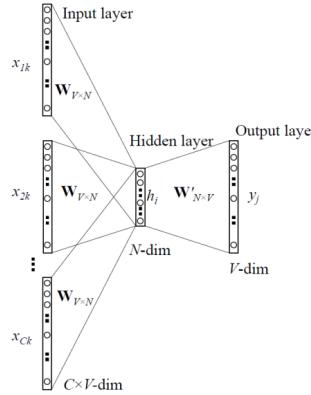
```
#data
CONTEXT_SIZE = 2

text = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells.""".split()
#영어를 tokenize할땐 강 띄어쓰기를 기준으로 해도...
#data

vocab = set(text) #중복값 제거
vocab_size = len(vocab) #vocab의 길이
print('vocab_size:', vocab_size)

w2i = {w: i for i, w in enumerate(vocab)} #word -> index
i2w = {i: w for i, w in enumerate(vocab)} #index -> word
```

- CBOW



```
#CBOW type dataset

def create_cbow_dataset(text):
    data = []
    for i in range(2, len(text) - 2):
        context = [text[i - 2], text[i - 1],
                   text[i + 1], text[i + 2]] #주변 단어를 통해 중심단어를 예측해야해서
        target = text[i] #중심단어를 target으로, 주변 text를 context로 구성
        data.append((context, target))
    return data #context와 target이 각각 들어간 data를 반환한다.
```

```
#CBOW model

class CBOW(nn.Module):
    def __init__(self, vocab_size, embd_size, context_size, hidden_size):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embd_size) #임베딩으로
        self.linear1 = nn.Linear(2*context_size*embd_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, vocab_size)
        #nn.Linear는 선형 변환을 하기 위해 사용된다. 크기를 맞춰주는것

    def forward(self, inputs):
        embedded = self.embeddings(inputs).view((-1, -1)) #input값을 임베딩으로
        hid = F.relu(self.linear1(embedded)) #임베디드된 값을 선형변환 후 relu
        out = self.linear2(hid) #output dimension에 맞게 선형변환
        log_probs = F.log_softmax(out) #확률값으로 바꿔주기 위해 softmax
        return log_probs #출력
```

```
#CBOW train

embd_size = 100
learning_rate = 0.001
n_epoch = 30

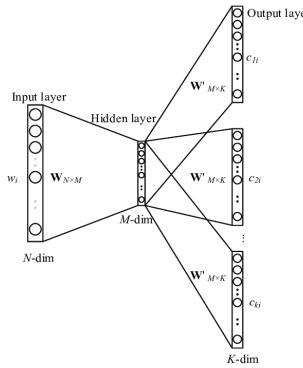
def train_cbow():
    hidden_size = 64
    losses = []
    loss_fn = nn.NLLLoss() #학습은 negative Loglikelihood
    model = CBOW(vocab_size, embd_size, CONTEXT_SIZE, hidden_size)
    print(model)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate) #구단다리 optimizer

    for epoch in range(n_epoch):
        total_loss = .0
        for context, target in cbow_train:
            ctx_ids = [w2i[w] for w in context] #context의 index 추출
            ctx_var = Variable(torch.LongTensor(ctx_ids)) #index를 텐서로 변환

            model.zero_grad() #model gradient 초기화
            log_probs = model(ctx_var) #context를 넣어서 target을 예측

            loss = loss_fn(log_probs, Variable(torch.LongTensor([w2i[target]])))
            #Negative Loglikehood Loss 값 도출
            loss.backward() #역전파
            optimizer.step() #gradient update
            total_loss += loss.data
        losses.append(total_loss)
    return model, losses
```

- Skip-gram



```
#Skip-gram dataset

def create_skipgram_dataset(text):
    import random #중심단어를 통해 주변단어를 예측하는 방향
    data = []
    for i in range(2, len(text) - 2):
        data.append((text[i], text[i-2], 1)) #단어들에 context 부여
        data.append((text[i], text[i-1], 1))
        data.append((text[i], text[i+1], 1))
        data.append((text[i], text[i+2], 1))
        for _ in range(4):
            if random.random() < 0.5 or i >= len(text) - 3:
                rand_id = random.randint(0, i-1)
            else:
                rand_id = random.randint(i+3, len(text)-1)
            data.append((text[i], text[rand_id], 0)) #그 외 값들은 0 부여
    return data
```

```
#Skip-gram model

class SkipGram(nn.Module):
    def __init__(self, vocab_size, embd_size):
        super(SkipGram, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embd_size) #Embedding 하나 쯤

    def forward(self, focus, context):
        embed_focus = self.embeddings(focus).view((1, -1)) #중심단어
        embed_ctx = self.embeddings(context).view((1, -1)) #주변단어
        score = torch.mm(embed_focus, torch.t(embed_ctx)) #torch.mm으로 score도출
        #mm = matrix multiplication
        log_probs = F.logsigmoid(score) #log sigmoid를 통해 확률값 도출

    return log_probs
```

```
#Skip-gram trainer

def train_skipgram():
    losses = []
    loss_fn = nn.MSELoss() #여긴 또 특이하게 MSELoss 씀.
    model = SkipGram(vocab_size, embd_size)
    print(model)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate) #구단다리쓰

    for epoch in range(n_epoch):
        total_loss = .0
        for in_w, out_w, target in skipgram_train:
            in_w_var = Variable(torch.LongTensor([w2i[in_w]])) #input과
            out_w_var = Variable(torch.LongTensor([w2i[out_w]])) #output을 텐서로

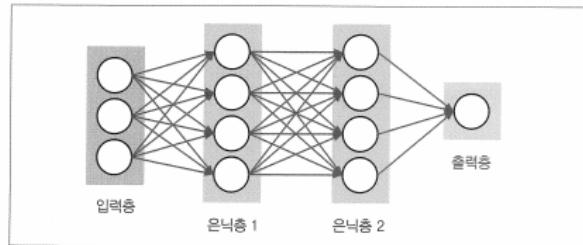
            model.zero_grad() #기울기 초기화
            log_probs = model(in_w_var, out_w_var) #결과값 구하기
            loss = loss_fn(log_probs[0], Variable(torch.Tensor([target])))

            loss.backward() #역전파
            optimizer.step() #기울기 업데이트

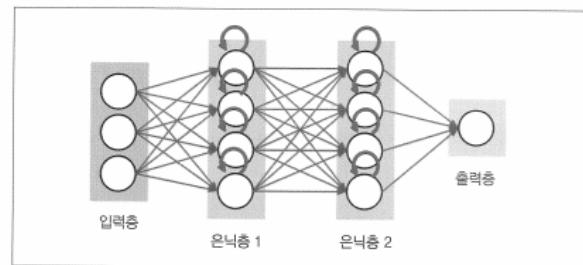
            total_loss += loss.data
        losses.append(total_loss)
    return model, losses
```

### [Neural Network for NLP]

- RNN 의 작동원리
  - RNN의 구조

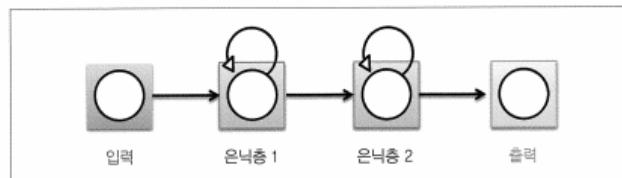


일반적인 인공 신경망



순환성을 추가한 인공 신경망

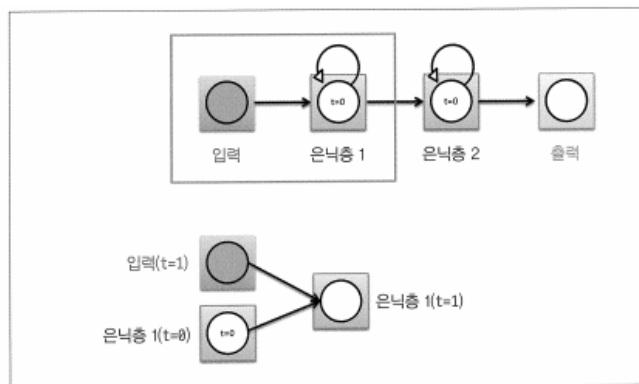
- 아래의 인공신경망은 3개의 입력이 각각 4개의 노드를 가진 은닉층 두개를 거치는 구조.
- 여기서 노드의 수를 1개로 바꾸면 첫 번째 은닉층의 값이 다음 번에 해당 은닉층의 입력으로 들어감.



간소화한 순환 신경망

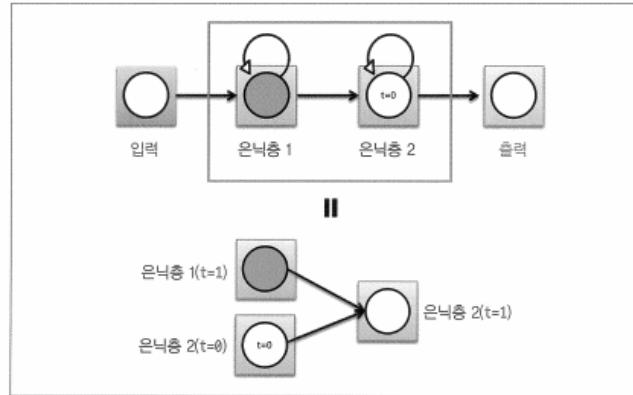
#### ▼ 과정 1

- 매 시간마다 새로운 입력 값이 들어오고 결과 값이 계산된다고 할 때,  $t=0$ 일 때 계산된 결과값이  $t=1$ 의 입력값이 되어, 실제 입력값과 동시에 들어가게 됨.

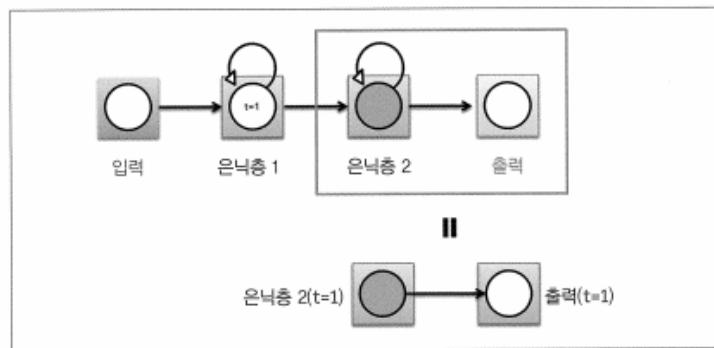


순환 신경망 연산이 실제로 이루어지는 방식

- $t=1$ 의 은닉층의 입력값 =  $t=0$ 의 은닉층의 결과값 +  $t=1$ 의 입력값  
→ 이전에 들어왔던 입력값에 대한 정보를 어느 정도 보존함으로서 맥락 파악이 가능함.

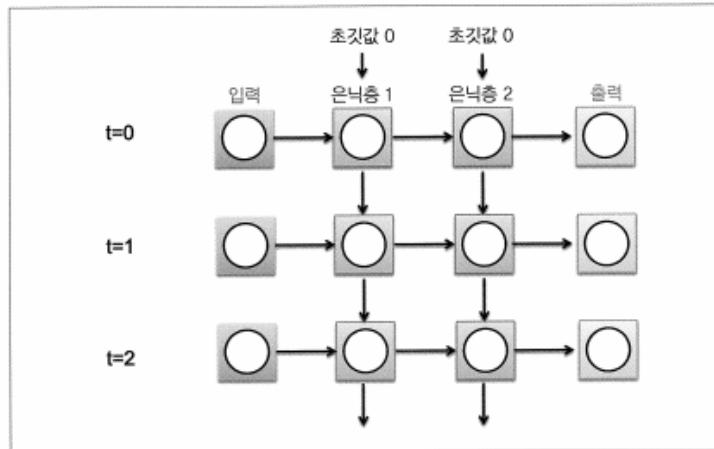


순환 신경망 연산이 실제로 이루어지는 방식 2



순환 신경망 연산이 실제로 이루어지는 방식 3

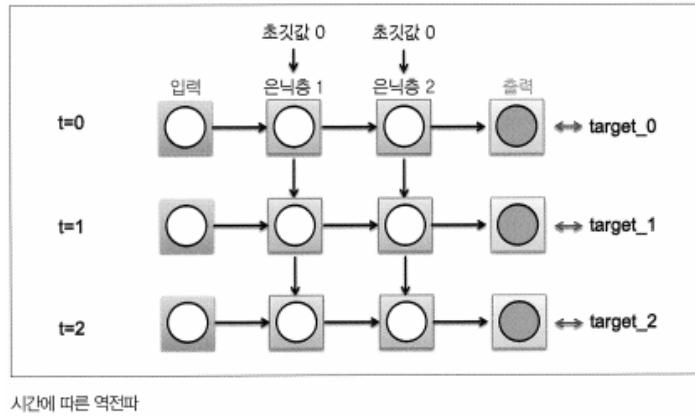
#### ▼ 시간에 따라서 본 RNN



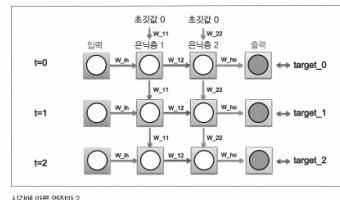
- $t=0$ : 초기값과 입력값을 가지고 은닉층에서 계산
- $t=1$ :  $t=0$ 의 결과값과 입력값을 가지고 은닉층의 결과값 계산  
→ 과정이 지정한 시간만큼 반복

#### ▼ 역전파

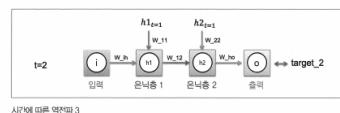
- 일정 시간 동안 계산 시 모델 학습 위해 결과값과 목표 값의 차이를 손실함수를 통해 계산하고 역전파해야 함.
- 기존의 역전파의 차이: 계산의 사용된 시점의 수에 영향을 받음  
즉, 역전파 계산은 계산이 진행된 시간만큼 해줘야 함 (ex.  $t=0$ 부터  $t=2$ 까지 계산이 되었다면, 그 시간 전체에 대한 역전파 진행)



- 순환 신경망은 각 위치별로 같은 가중치를 공유하므로, t=2 시점의 손실을 재계산(역전파) 하기 위해서 t=0 시점의 노드들에도 다 영향을 줘야 함.
- BPTT: 시간을 역으로 거슬러 올라가는 방식으로 각 가중치를 업데이트
- 가중치



- t=2 시점만 보면



$$o = w_{\text{out}} \times h2_{\text{out}} + \text{bias}$$

$$h2_{\text{out}} = \tanh(w_{12} \times h1 + w_{22} \times h2_{\text{out}} + \text{bias})$$

$$h1_{\text{out}} = \tanh(w_{11} \times i + w_{11} \times h1_{\text{out}} + \text{bias})$$

$$h2_{\text{in}} = w_{12} \times h1 + w_{22} \times h2_{\text{out}} + \text{bias}$$

$$h1_{\text{in}} = w_{11} \times i + w_{11} \times h1_{\text{out}} + \text{bias}$$

$$\frac{\partial o}{\partial w_{22}} = \frac{\partial o}{\partial h2_{\text{out}}} \times \frac{\partial h2_{\text{out}}}{\partial h2_{\text{in}}} \times \frac{\partial h2_{\text{in}}}{\partial w_{22}}$$

→ h2\_in을 미분하면 h2\_t=1이 나옴 → h2\_t=1 값은 이전 시점의 값들의 조합으로 이루어져있고, 내부적으로도 w\_22를 포함하고 있음.

→ t=2 시점에서 발생한 손실은 t=2, 1, 0 시점에 영향을 주고 t=1에서의 손실은 t=1, 0에 영향을, t=0 시점의 손실은 t=0의 가중치에 영향을 줌.

실제로 업데이트 시에는 가중치에 대해 시점 별 기울기를 다 더해서 한번에 업데이트 진행.

### ○ 기준 모델의 한계

- 타임 시퀀스가 늘어나며 역전파 시 tanh의 미분값이 여러번 곱해져 유의미하지 않는 것

#### ▼ 기울기 소실(Vanishing gradient)

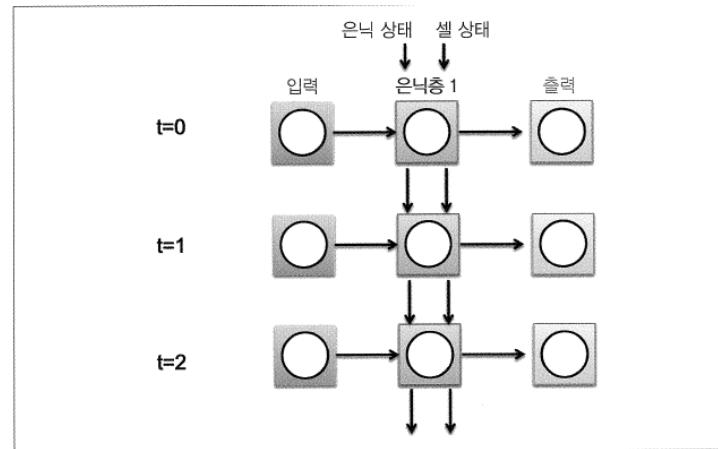
- 하이퍼볼릭 탄젠트 함수를 미분하면 0과 1사이의 값이 나오고, 기울기 값이 역전파될 때 타임 시퀀스가 길어질수록 모델이 학습을 제대로 하지 못함

= 역전파 과정에서 입력층으로 갈 수록 기울기(Gradient)가 점차적으로 작아져 입력층에 가까운 층들에서 가중치들이 업데이트가 제대로 되지 않는 현상

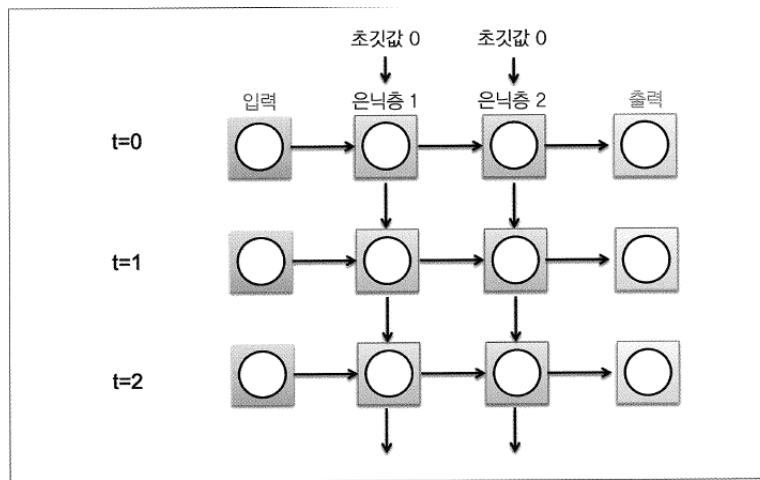
#### ▼ 합성곱 신경망처럼 활성화 함수 등을 바꿔야하나?

- 두 개의 개선모델인 LSTM과 GRU 등장

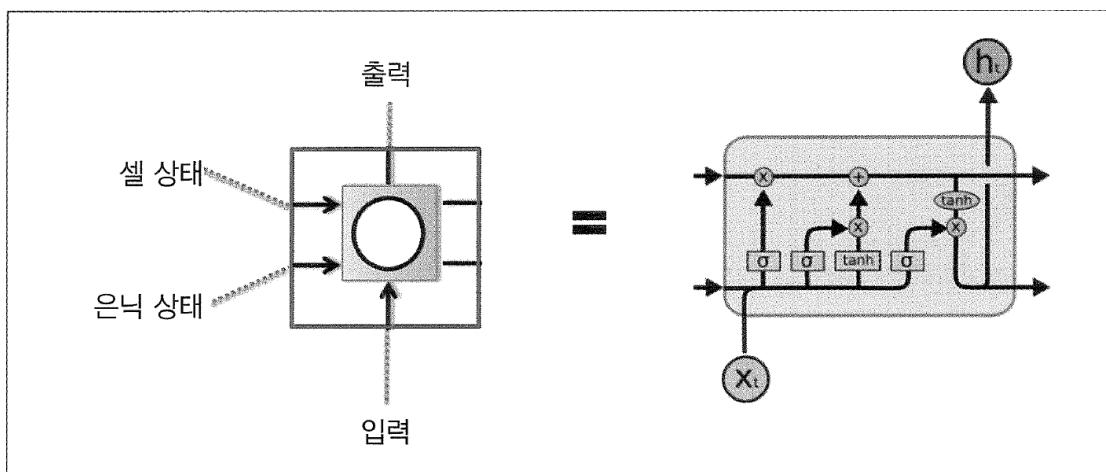
◦ ① LSTM, Long Short-Term Memory



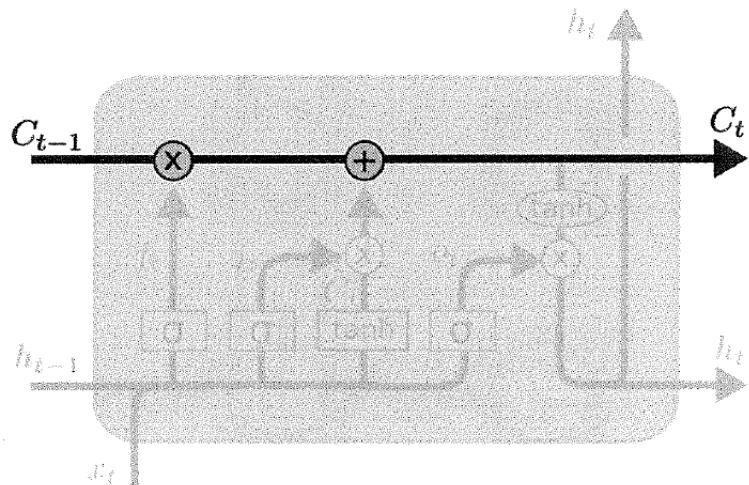
▼ 기존



- 기존 RNN 모델에 장기기억 부품을 추가한 것
- 기존에 은닉상태(hidden state)만 있었다면, 셀 상태(cell state)를 추가한 것
- 기본 순환 신경망은 현재의 입력값과 이전 시간의 은닉층 값의 조합으로 새로운 값을 생성했지만, LSTM은 은닉층 내부가 보다 훨씬 복잡한 구조로 이루어져있다

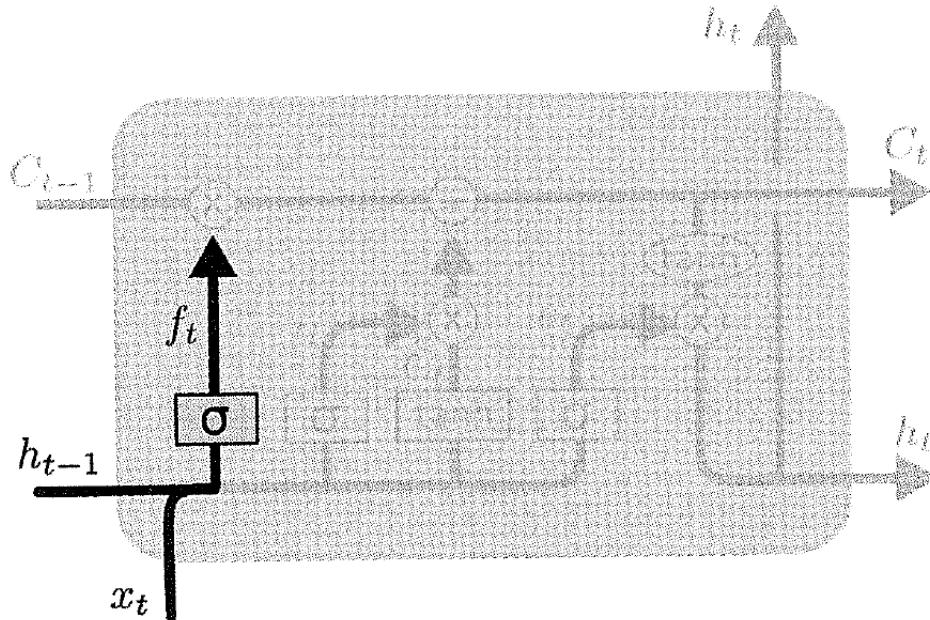


▼ 1) 셀 상태 (Cell state)



- 장기 기억을 담당하는 부분
- 곱하기( $\times$ ) 부분 - 기존의 정보를 얼마나 남길 것인지에 따라 비중을 곱하는 부분
- 더하기(+) 부분 - 현재 들어온 데이터와 기존의 은닉 상태를 통해 정보를 추가하는 부분

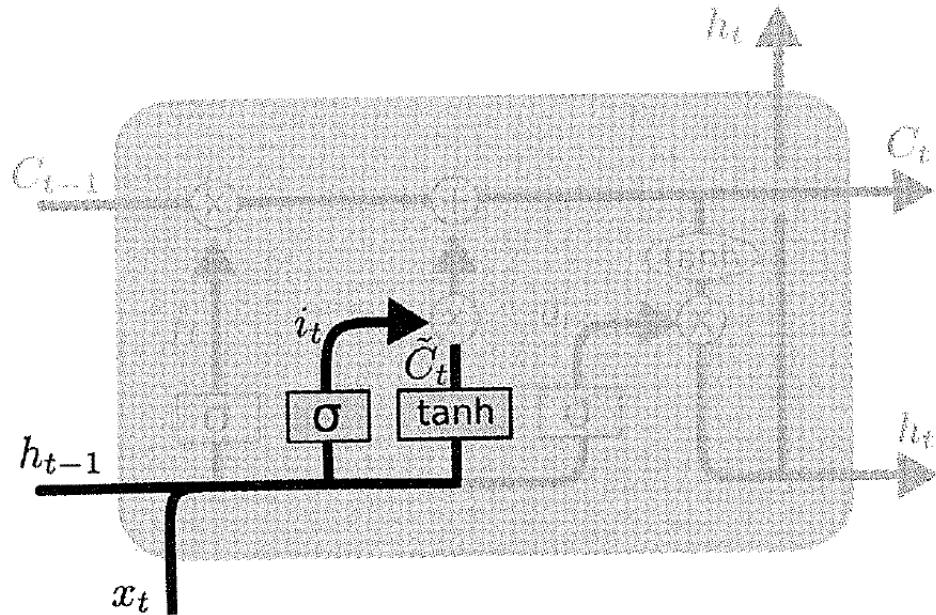
▼ 2) 망각 게이트(Forget gate)



$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

- 기존의 정보들로 구성되어 있는 셀 상태의 값을 얼마나 잊어버릴 지 정하는 부분
- $\sigma$ 는 시그모이드 함수
- 현재 시점의 입력값과 직전 시점의 은닉 상태 값을 입력으로 받는 한 층의 인공신경망
- 가중치를 곱해주고 바이어스를 더한 값을 시그모이드 함수에 넣은 값으로 기존의 정보를 얼마나 전달할지 비중을 정함

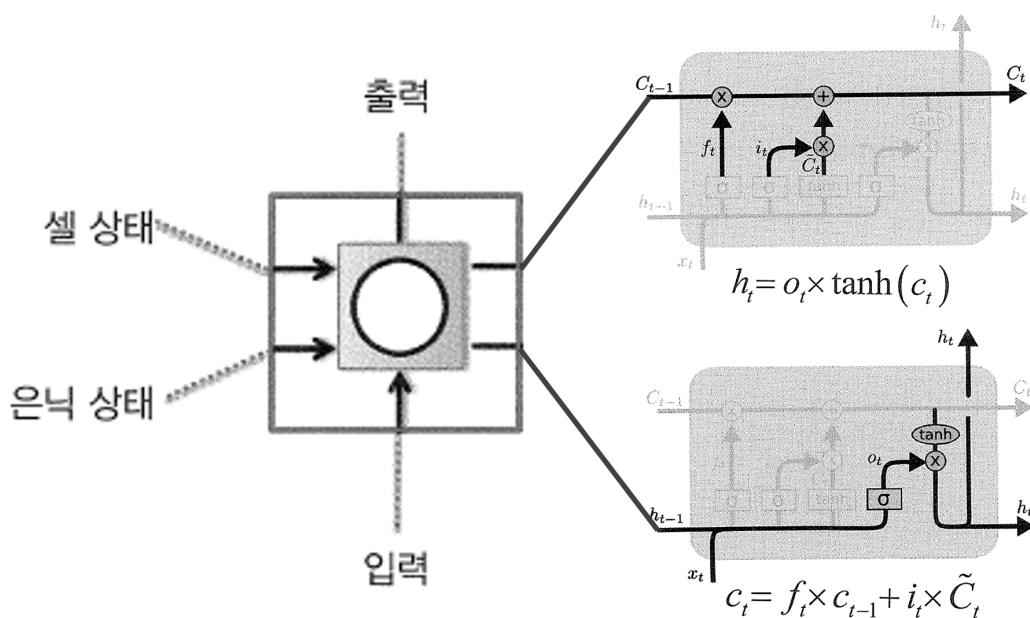
▼ 3) 입력 게이트(Input gate)



$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

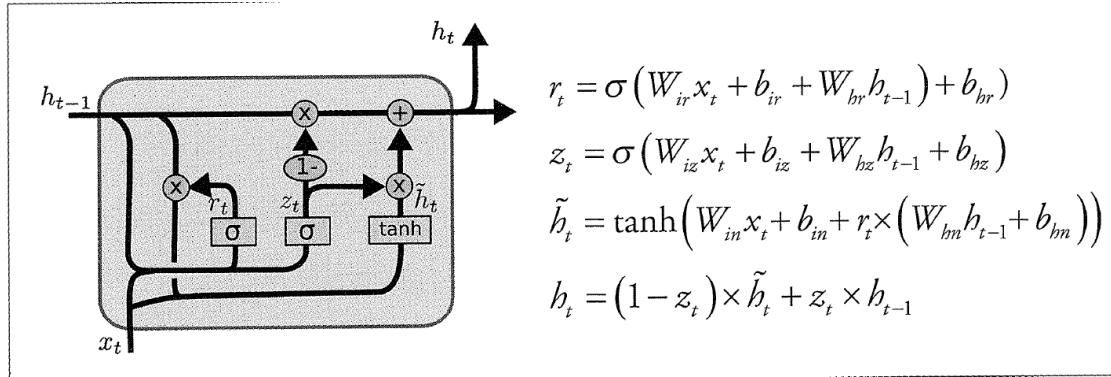
$$\tilde{C}_t = \tanh(W_{iC}x_t + b_{ic} + W_{hc}h_{t-1} + b_{hc})$$

- 어떤 정보를 얼마큼 셀 상태에 새롭게 저장할 것인지 정하는 부분
  - 새로운 입력값과 직전 시점의 은닉 상태값을 받음
  - 한번은 시그모이드 활성화 함수
  - 한번은 하이퍼볼릭 탄젠트 활성화 함수
  - 하이퍼볼릭 탄젠트를 통해 나온 값은 -1에서 1사이의 값을 가지고 새롭게 셀 상태에 추가할 정보가 됨
  - 시그모이드 함수를 통해 나온 값은 0에서 1 사이의 비중으로 새롭게 추가할 정보를 얼마큼의 비중으로 셀 상태에 더해줄 지 정함
- 정리하면 ▼



- LSTM 모델은 오늘날 기본적인 모델로 사용되며, 기본 RNN보다 더 나은 성능을 보여줌

- ② GRU, gated recurrent unit
  - RNN의 변형 형태
  - LSTM보다 간단한 구조임에도 불구하고 성능에서 밀리지 않음
  - 2014년 조경현 교수 등의 논문에서 발표되었음

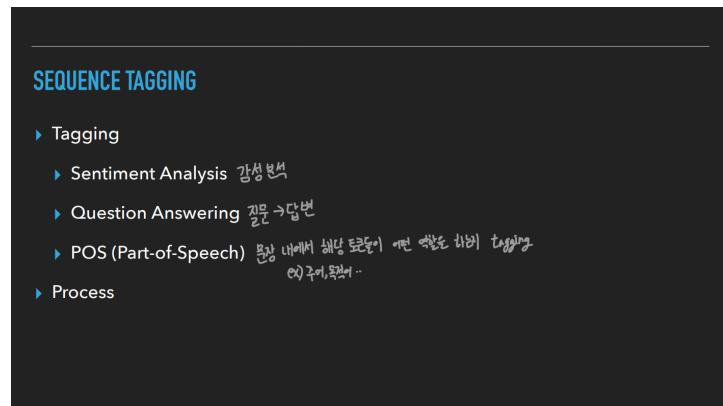


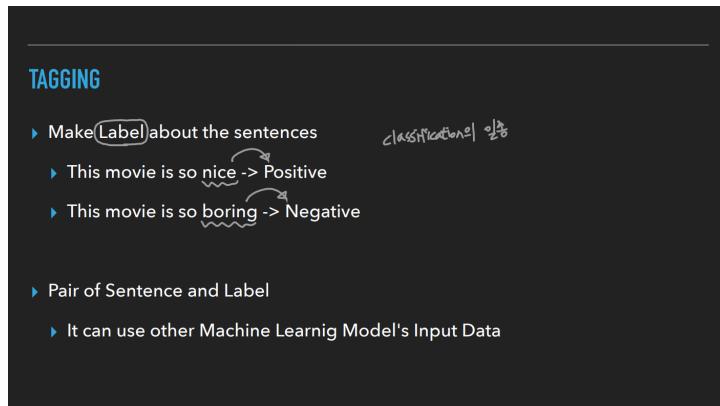
- 셀 상태와 은닉 상태를 분리하지 않고 여전히 은닉 상태만 둠
- ▼ 1) 업데이트 게이트(Update gate)
  - 현재 시점의 새로운 입력값과 직전 시점의 은닉 상태 값에 가중치를 곱하고 시그모이드 함수를 통과시켜 업데이트 할 비중을 정하는 부분
- ▼ 2) 리셋 게이트(Reset gate)
  - 업데이트 게이트와 같은 입력을 받아 동일하게 시그모이드 함수를 통해 비중을 정하며, 이 비중은 그 다음줄 수식  $\tilde{h}$ 를 구할 때 기존 은닉 상태 값을 얼마나 반영할지 정하는 데 사용됨
  - $\tilde{h}$ 는 기존의 은닉 상태에 가중치가 곱해진 값과 새로운 입력값을 입력으로 받아 가중치를 곱한 후 하이퍼볼릭 탄젠트 함수를 통해 새로운 정보의 값을 리턴함
  - 마지막 수식은 첫번째 수식에서 구한 가중치로 기존 은닉 상태와  $\tilde{h}$ 의 가중 합을 곱해서 새로운 은닉 상태를 구하는 부분

## Section 02 | Neural Machine Translation

### [Sequence Tagging]

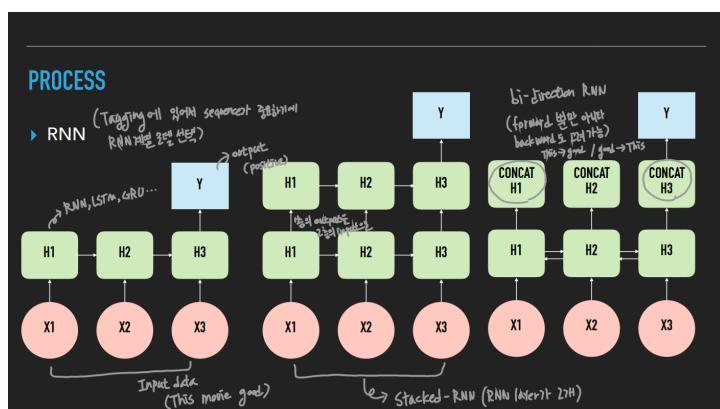
- Tagging의 3가지 종류





: Tagging이라는 것은 문장에 labeling을 하는 것과 같음. (classification의 일종)

ex] 감성 분석의 경우 → nice라는 토큰엔 positive라는 label을, boring이라는 토큰엔 negative라는 label을 붙임  
ex] Q/A의 경우 → 질문에 특정 labeling이 되어있는 단어들이 들어가면 그에 해당하는 답변을 보여줌

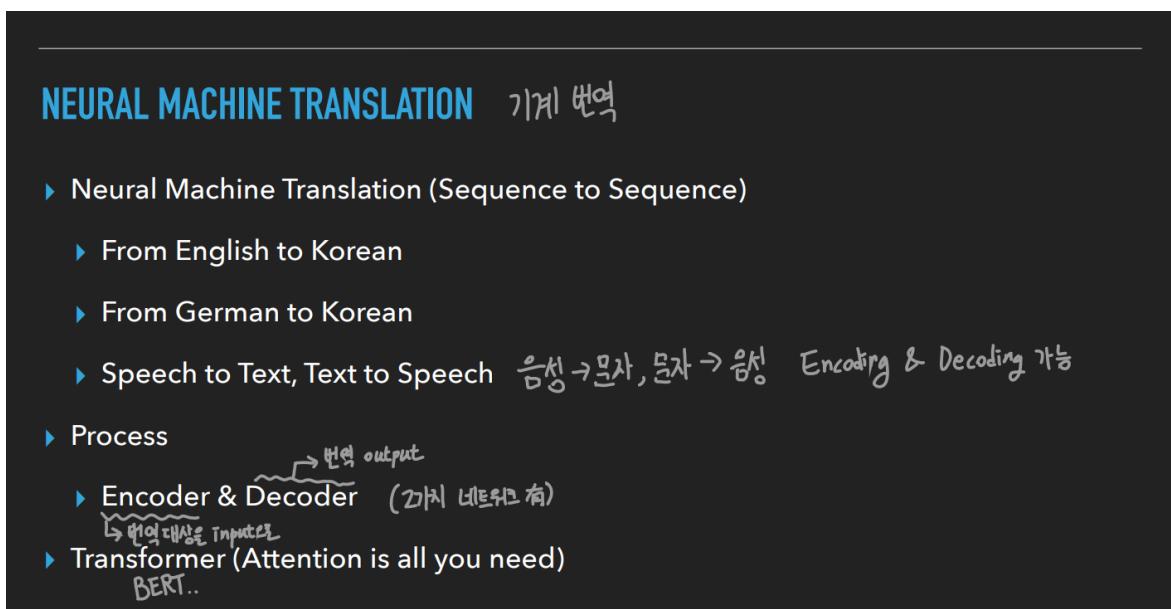


Tagging에 있어서는 제목처럼 sequence(순서)가 중요하기 때문에 RNN 계열 모델 사용

→ 강의에서는 3가지의 RNN 모델을 예시로 보여줌 (Stacked-RNN, Bi-Directional RNN 등)

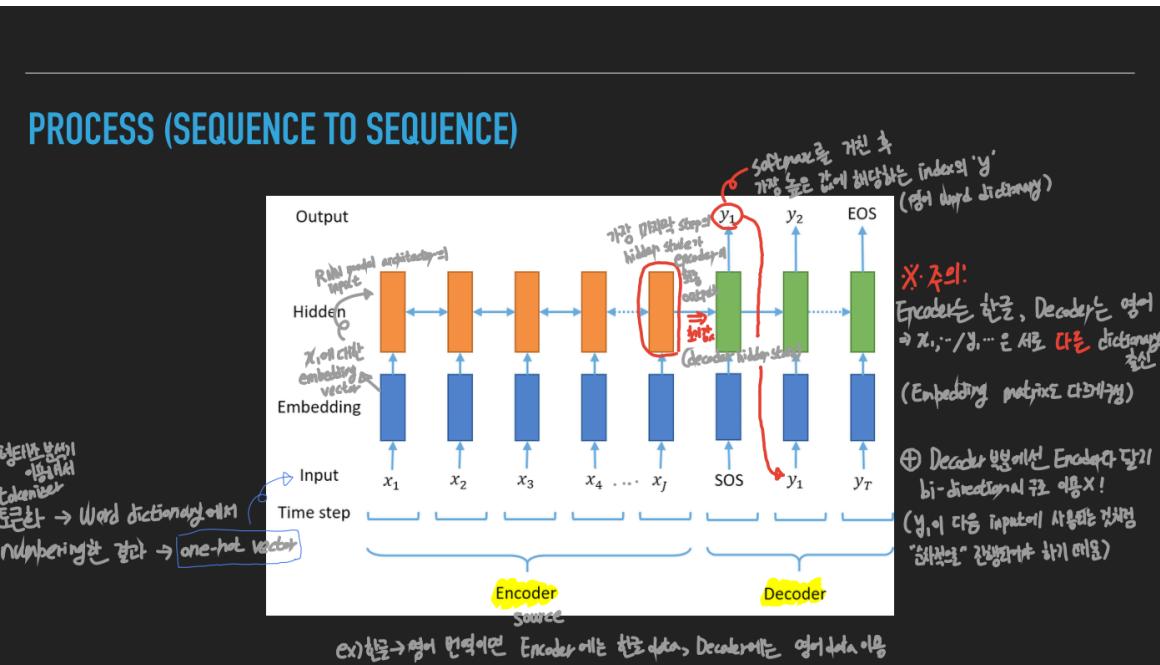
### [Neural Machine Translation]

- 기계 번역의 종류(seq2seq)





Transformer는 RNN을 사용하지 않고 오로지 Attention 메커니즘만 사용한 것!



⇒ seq2seq는 번역기에서 대표적으로 사용되는 모델

### • Encoder architecture

: 형태소 분석기를 이용해 토큰화 → word dictionary에서 numbering한 결과가 one-hot vector로 저장 → input값으로( $x_1, x_2, \dots$ ) 들어 가고, 이것이 각각의 embedding vector를 거침 → hidden state의 input으로 들어감 → 가장 마지막 step의 hidden state가 encoder의 최종 output이 됨. → 이 output이 decoder hidden state의 초기값으로 들어 갑.

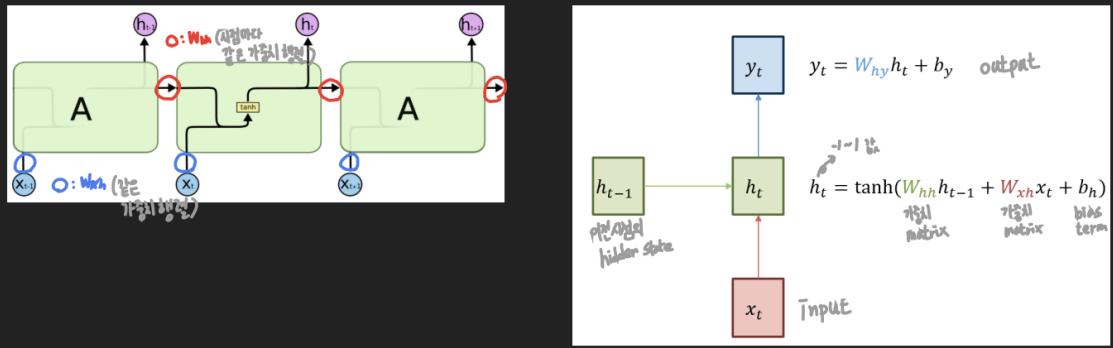
### • Decoder architecture

: sos와 embedding vector, 그리고 아까 encoder의 마지막 step을 초기값을 받았던 decoder hidden state를 거치고 softmax를 거친 후 영어로 이루어진 word dictionary에서 가장 높은 값에 해당하는 index의  $y_1$ 을 찾게 됨 →  $y_1$ 은 다음 단계의 decoder에서 input으로 들어가게 된다.

### [Attention Mechanism]

#### • Background of Attention

## BACKGROUND OF ATTENTION



- ▶ Need to Refine Long Term Dependency (문장 길이가 길 경우) ↳ 문장의 토큰 수 많음 반복해서 진행  
gradient vanishing
- ▶ LSTM, GRU is not enough,  $\Rightarrow$  Attention 등장!

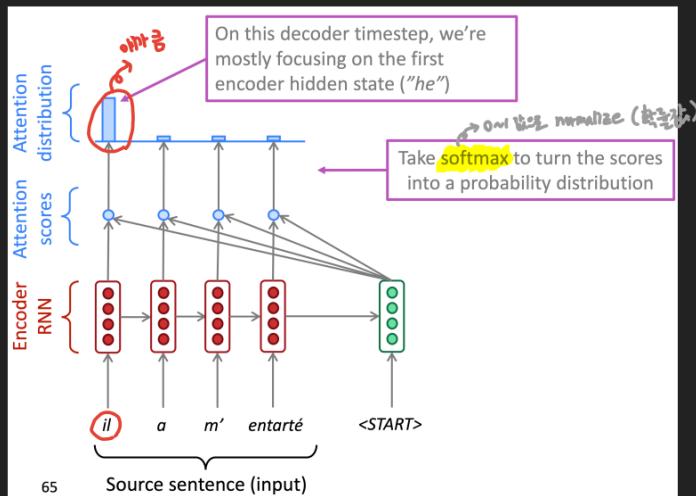
기본 RNN구조는 문장 길이가 길 경우 Gradient Vanishing 현상이 발생함

→ LSTM, GRU로 어느정도 커버가 가능하지만 부족함

→ Attention의 등장!

💡 Attention 이런  
→ 어느 위치의 정보를 더 중요하게 볼 것인가에 대한 정보를 넣어주기 위한 방법

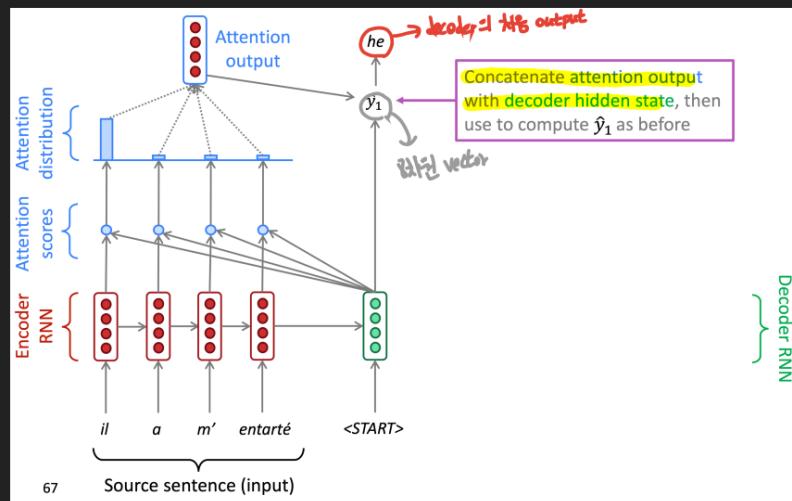
## ATTENTION IN ENCODER-DECODER



→ 모든 시점에서의 Encoder hidden state들과 Decoder hidden state를 각각 내적해서 얻은 scalar가 **Attention scores**

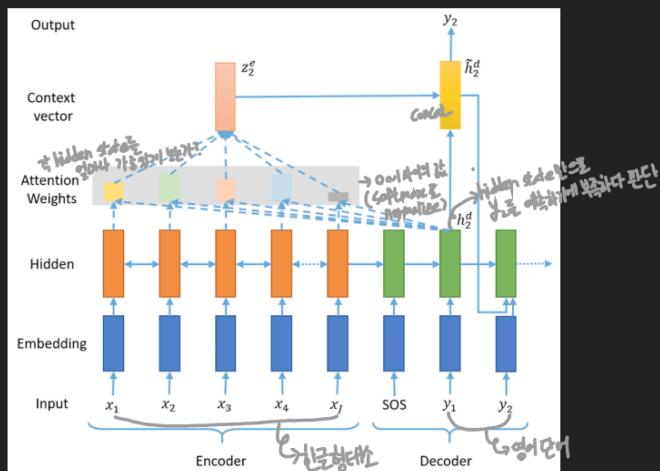
→ 후에 attention scores들에 softmax를 취해 0~1 사이의 값으로 normalize(**Attention distribution**)

## ATTENTION IN ENCODER-DECODER



- Attention distribution의 weighted sum을 통해 context vector인 **Attention output**을 구해준다  
(Attention output에는 high attention인 hidden state에 대한 정보가 들어 있음!)
- Attention output과 decoder hidden state를 concatenate를 해서 output 도출

## PROCESS (SEQUENCE TO SEQUENCE WITH ATTENTION)



### [Sequence Tagging Practice]

- library

```

import os
import sys
import time
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim

from torch.nn import functional as F

```

```

from torch.autograd import Variable
from torchtext import data #text data를 다뤄주는 library
from torchtext import datasets #데이터셋
from torchtext.vocab import Vectors, GloVe

```

- Load Dataset

```

def load_dataset(test_sen=None):
    tokenize = lambda x: x.split() #Token은 단순히 띄어쓰기로 구분
    TEXT = data.Field(sequential=True, tokenize=tokenize, lower=True, include_lengths=True, batch_first=True, fix_length=200)
    #torchtext.data -> text 데이터 파이프라인 구축 / 영어는 잘 되는데 한글은 체크 필요
    LABEL = data.LabelField()
    train_data, test_data = datasets.IMDB.splits(TEXT, LABEL) #IMDB dataset 사용
    TEXT.build_vocab(train_data, vectors=Glove(name='6B', dim=300)) #vocab 구축 파이프라인
    LABEL.build_vocab(train_data) #Label도 build vocab!

    word_embeddings = TEXT.vocab.vectors #Pretrained Vector를 바탕으로 Input Embedding구현
    print ("Length of Text Vocabulary: " + str(len(TEXT.vocab)))
    print ("Vector size of Text Vocabulary: ", TEXT.vocab.vectors.size())
    print ("Label Length: " + str(len(LABEL.vocab)))

    train_data, valid_data = train_data.split() #train/valid 구분
    train_iter, valid_iter, test_iter = data.BucketIterator.splits((train_data, valid_data, test_data), batch_size=32, sort_key=lambda x: len(x.text), repeat=False,
    #BucketIterator -> 각 배치마다 similar lengths가 들어가도록 iterator를 정의
    #문장의 최대 Length를 맞춤으로써 필요한 패딩의 수를 minimize
    vocab_size = len(TEXT.vocab)

    return TEXT, vocab_size, word_embeddings, train_iter, valid_iter, test_iter

```

※ 왜 vocab을 구축하는데 Glove 벡터를 사용하는걸까?

→ train dataset이 충분히 큰 경우엔 이미 가지고 있는 데이터셋으로 nn.Embedding에서 input Embedding을 형성해도 된다. 하지만 Input dataset이 충분히 크지 않을 경우엔 Pretrained Embedding으로부터 도움을 받음

- Model Structure

```

class RNN(nn.Module):
    def __init__(self, batch_size, output_size, hidden_size, vocab_size, embedding_length, weights):
        super(RNN, self).__init__()

        self.batch_size = batch_size #weights 변수를 제외한 나머지를 self 선언
        self.output_size = output_size #자기 자신을 참조하는 매개변수
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size
        self.embedding_length = embedding_length

        self.word_embeddings = nn.Embedding(vocab_size, embedding_length) #vocab * 차원으로 임베딩
        self.word_embeddings.weight = nn.Parameter(weights, requires_grad=False)
        #이미 pretrained된 임베딩이기에 parameter를 굳애버림
        self.rnn = nn.RNN(embedding_length, hidden_size, num_layers=2, bidirectional=True)
        #임베딩 차원으로서 Hidden size로 나가도록 2개의 layer구성.
        self.label = nn.Linear(4*hidden_size, output_size)
        #2개의 layer에 bidirectional한 값이 concat되어 4*hidden size output
        #선형 변화를 통해 output size로 내보냄

    def forward(self, input_sentences, batch_size=None):
        input = self.word_embeddings(input_sentences) #input sentence를 word Embedding으로
        input = input.permute(1, 0, 2)
        #input 형태를 바꾸기
        if batch_size is None:
            h_0 = Variable(torch.zeros(4, self.batch_size, self.hidden_size).cuda())
        else:
            h_0 = Variable(torch.zeros(4, batch_size, self.hidden_size).cuda())
        #hidden layer를 initialize 해줌
        output, h_n = self.rnn(input, h_0)
        h_n = h_n.permute(1, 0, 2)
        h_n = h_n.contiguous().view(h_n.size()[0], h_n.size()[1]*h_n.size()[2])
        logits = self.label(h_n)

        return logits

```

- Trainer

```

TEXT, vocab_size, word_embeddings, train_iter, valid_iter, test_iter = load_dataset()

def clip_gradient(model, clip_value): #이건 뭐하는거지?
    params = list(filter(lambda p: p.grad is not None, model.parameters()))
    for p in params:
        p.grad.data.clamp_(-clip_value, clip_value)

def train_model(model, train_iter, epoch):
    total_epoch_loss = 0
    total_epoch_acc = 0
    model.cuda() #model을 GPU로!
    optim = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()))
    #우리의 영원한 친구 아Dam
    steps = 0 ##이 step까지 학습을..
    model.train()
    for idx, batch in enumerate(train_iter):
        text = batch.text[0]
        target = batch.label
        target = torch.autograd.Variable(target).long() #Longtensor로 할당하지않나..
        if torch.cuda.is_available():

```

```

text = text.cuda() #DataLoader에서 GPU 할당하는게 더 빠르지 않나..
target = target.cuda() #왜 반복문안에 GPU를 할당하는거지? GPU VRAM이 부족한가?
if (text.size()[0] is not 32):# One of the batch returned by BucketIterator has length different than 32.
    continue
optim.zero_grad_() #optimizer initialize
prediction = model(text)
loss = loss_fn(prediction, target)
num_corrects = (torch.max(prediction, 1)[1].view(target.size()).data == target.data).float().sum()
acc = 100.0 * num_corrects/len(batch) #accuracy를 %로 추출하는군..
loss.backward() #역전파
clip_gradient(model, 1e-1)
optim.step() #gradient update
steps += 1

if steps % 100 == 0:
    print (f'Epoch: {epoch+1}, Idx: {idx+1}, Training Loss: {loss.item():.4f}, Training Accuracy: {acc.item(): .2f}%')

total_epoch_loss += loss.item()
total_epoch_acc += acc.item()

return total_epoch_loss/len(train_iter), total_epoch_acc/len(train_iter)

```

- Prediction

```

test_sen1 = "This is one of the best creation of Nolan. I can say, it's his magnum opus. Loved the soundtrack and especially those creative dialogues."
#문장!
test_sen1 = TEXT.preprocess(test_sen1) #TEXT에 넣어둔 Pipeline 따라서 처리
test_sen1 = [[TEXT.vocab.stoi[x] for x in test_sen1]] index 할당하는듯

test_sen = np.asarray(test_sen1) #문장을 list에서 np.array로 바꿈(옛날 코드 방식인듯)
test_sen = torch.LongTensor(test_sen) #그리고 np.array를 LongTensor로
test_tensor = Variable(test_sen, volatile=True) #Long Tensor를 Variable로 변환
#예전엔 tensor에 auto_grad가 할당이 안되었나봄
test_tensor = test_tensor.cuda() #코드 드롭게 길게 짜네. 데이터를 GPU 활동형태로 변환
model.eval() #모델 eval 모드로(gradient 변환기 없게)
output = model(test_tensor, 1) #결과 1 = positive, 0 = negative
out = F.softmax(output, 1) #확률로 변환
print(out)
if (torch.argmax(out[0]) == 1):
    print ("Sentiment: Positive")
else:
    print ("Sentiment: Negative")

```

### [Neural Machine Translation Practice]

- library

```

import torch
import torch.nn as nn
import torch.optim as optim

from torchtext.datasets import TranslationDataset, Multi30k
from torchtext.data import Field, BucketIterator

import spacy #체고존엄 스파시
import numpy as np

import random
import math
import time

```

- Preprocessing

```

spacy_en = spacy.load('en_core_web_sm')
spacy_de = spacy.load('de_core_news_sm')
#각 tokenizer에 활용할 data를 불러오자

def tokenize_de(text):
    return [tok.text for tok in spacy_de.tokenizer(text)][::-1]

def tokenize_en(text):
    return [tok.text for tok in spacy_en.tokenizer(text)]

#Spacy의 tokenizer를 이용하여 각 text를 token으로 변환

SRC = Field(tokenize = tokenize_en,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)
#각 Token을 위해 special token을 부여
#<sos> -> start of sentence
#<eos> -> end of sentence
#lower -> 소문자로

TRG = Field(tokenize = tokenize_de,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)

train_data, valid_data, test_data = Multi30k.splits(exts = ('.de', '.en'),
                                                    fields = (SRC, TRG))
#multi30K 데이터를 독일어, 영어로 fields의 data pipeline 따라서 나눔

SRC.build_vocab(train_data, min_freq = 2)

```

```

TRG.build_vocab(train_data, min_freq = 2)

BATCH_SIZE = 128

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
#BucketIterator 정의해주기-

```

\* 잠깐만! torchtext.data.Field에 대해 간단히 체크하고 넘어가자(앞에서도 보았지만, 데이터의 파이프라인을 간단히 설정해주는 것)

#### ▼ about data.Field

- Defines a datatype together with instructions for converting to Tensor.
- Field class models common text processing datatypes that can be represented by tensors. It holds a Vocab object that defines the set of possible values for elements of the field and their corresponding numerical representations. The Field object also holds other parameters relating to how a datatype should be numericalized, such as a tokenization method and the kind of Tensor that should be produced.
- If a Field is shared between two columns in a dataset (e.g., question and answer in a QA dataset), then they will have a shared vocabulary.

#### • Encoder&Decoder Architecture

```

class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        # == super(self, Encoder).__init__()
        self.hid_dim = hid_dim #hidden dimension 정의
        self.n_layers = n_layers #number of layers 정의
        self.embedding = nn.Embedding(input_dim, emb_dim) #input length*embedding dimension
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)
        #embedding dimension * hidden dimension으로 2개의 layer
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src)) #들어온 걸 임베딩으로 만들고 바로 드롭아웃
        outputs, (hidden, cell) = self.rnn(embedded) #임베딩을 rnn 2 layer 거침
        return hidden, cell #hidden과 cell 반환

class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        #비슷
        self.output_dim = output_dim #output dimension 정의
        self.hid_dim = hid_dim #hidden dimension 정의
        self.n_layers = n_layers #layer 개수 정의
        self.embedding = nn.Embedding(output_dim, emb_dim) #output dimension에 맞는 임베딩 정의
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout) #LSTM
        self.fc_out = nn.Linear(hid_dim, output_dim) #hidden -> output dimension으로 선형변환(차원축소)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell):
        input = input.unsqueeze(0) #0번째 위치에 1인 차원을 추가함
        embedded = self.dropout(self.embedding(input)) #임베딩으로 만들어줌
        output, (hidden, cell) = self.rnn(embedded, (hidden, cell)) #RNN 돌림
        prediction = self.fc_out(output.squeeze(0)) #출력값에서 0번째 위치 차원 제거
        return prediction, hidden, cell #prediction과 hidden, cell 반환

```

#### • Seq2Seq Architecture

```

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    assert encoder.hid_dim == decoder.hid_dim, \
        "Hidden dimensions of encoder and decoder must be equal!"
    assert encoder.n_layers == decoder.n_layers, \
        "Encoder and decoder must have equal number of layers!"
    #둘의 hidden dimension은 같아야 하며, layer도 같아야 함
    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        batch_size = trg.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)
        hidden, cell = self.encoder(src)
        input = trg[0,:]

        for t in range(1, trg_len):
            output, hidden, cell = self.decoder(input, hidden, cell) #encoder의 output이 decoder의 input으로 들어감
            outputs[t] = output
            teacher_force = random.random() < teacher_forcing_ratio #사실 대부분은 잘 모르겠음
            topi = output.argmax(1)
            input = trg[t] if teacher_force else topi
        return outputs

```

- Trainer 정의

```
def train(model, iterator, optimizer, criterion, clip):  
    model.train()  
    epoch_loss = 0  
    for i, batch in enumerate(iterator):  
        src = batch.src #source  
        trg = batch.trg #target  
        optimizer.zero_grad()  
        output = model(src, trg)  
        output_dim = output.shape[-1]  
        output = output[1: ].view(-1, output_dim)  
        trg = trg[1: ].view(-1)  
        loss = criterion(output, trg)  
        loss.backward()  
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)  
        optimizer.step()  
        epoch_loss += loss.item()  
    return epoch_loss / len(iterator)
```

## Section 03 | NLP Trend

### [강의에서 제시한 Trend]

#### → Transformer, BERT

: 이미 BERT 파생모델들은 현역으로 사용중에 있으며, XLnet 이후로는 사실상 BERT라고 하기도 뭐함.  
(MLM, SEP도 안쓰고 단순히 Transformer Encoder, Decoder 베이스라고 모두가 같은가..)

#### → Teacher-Student (Knowledge Distillation)

DistilBERT 등, 기존 거대모델을 경량화한 것들이 이미 존재

하지만 BERT 파생 모델들 특징이 아직까지 각 task에서 독보적인 성능을 내지 못하는데

그것에 대한 distillation을 해서 어디다 쓸 수 있는지...?

만약 우리가 아주 뛰어난 성능의 pretrained model fine-tuning을 했다고 해도 그걸 task-specific하게 distillation에서 배포하기엔 우리의 역량이 부족할 듯.

#### → Chatbot

Sentimental Analysis를 제외하면 가장 대표적인 NLP task인 한데 우리가 쓰진 않을듯

#### → GLUE

벤치마크네요

### [추가자료]

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7dd51d1-edd5-44dd-80de-ce65a2a4b141/deep\\_learning\\_for\\_sentiment\\_analysis\\_a\\_survey.pdf](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7dd51d1-edd5-44dd-80de-ce65a2a4b141/deep_learning_for_sentiment_analysis_a_survey.pdf)

: Sentiment Analysis에 대한 survey paper인데 다양한 sentiment analysis의 subfield를 볼 수 있습니다. 다만 Transformer 기반 Sentiment Analysis에 대한 언급은 적기 때문에 각자 필요하면 읽어보시길