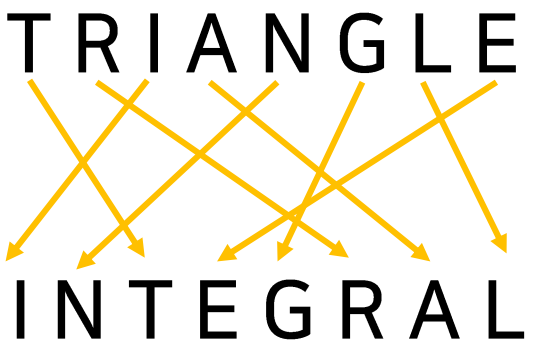
RNN

파이토치 첫걸음 CHAPTER 6 순환신경망

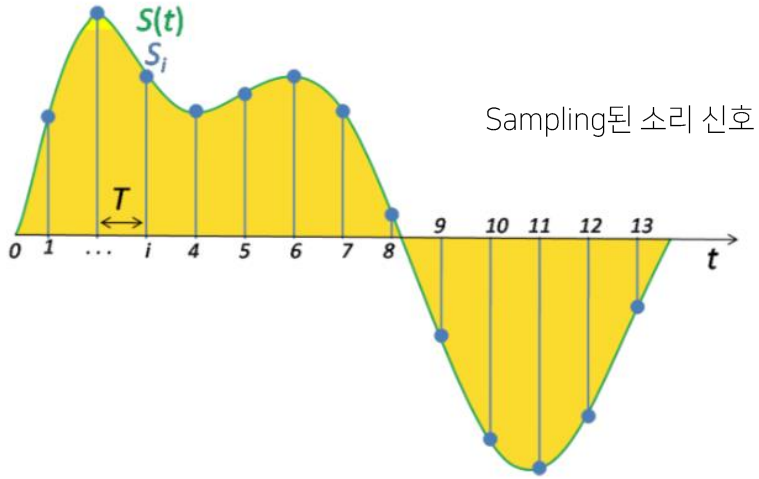
BOAZ 16기 이다인

순환신경망의 등장

우리가 주로 사용하는 데이터? “Sequence Data”



같은 알파벳의 나열이지만
순서에 따라 그 의미가 달라짐



Sampling된 소리 신호

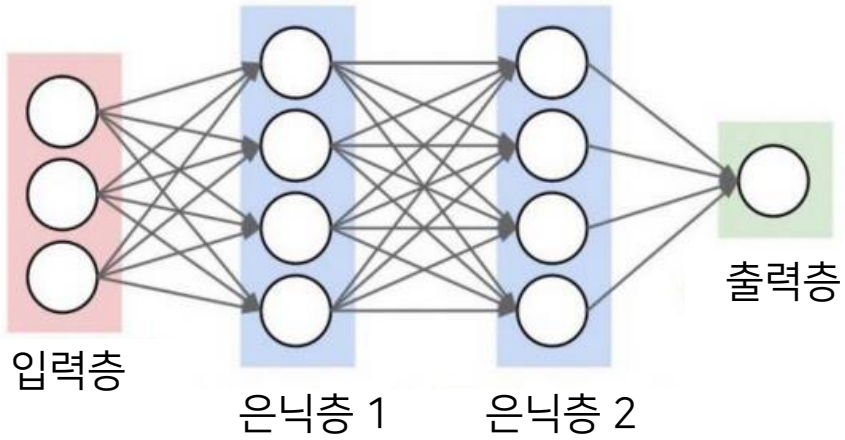
순서가 의미가 있으며, 순서가 달라질 경우 의미가 손상되는 데이터가 Sequence Data

특별히 시간에 따른 의미가 존재하는 데이터는 Time Series data

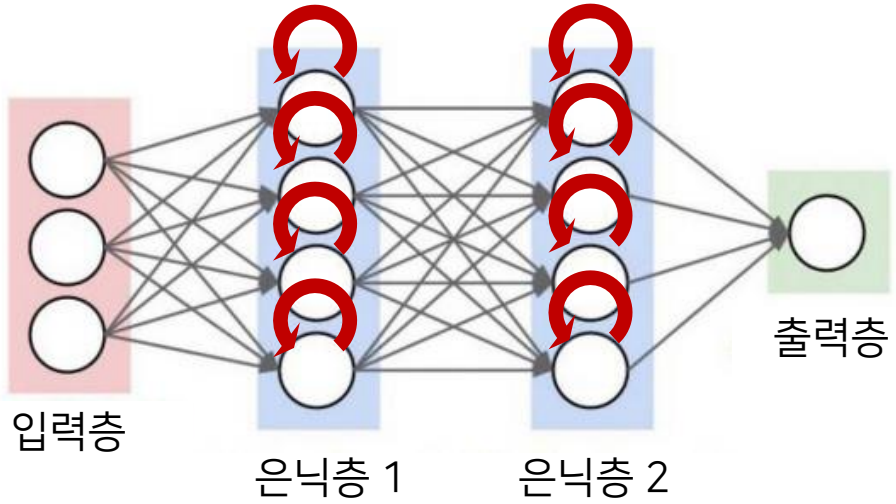
Sequence Data에 숨은 패턴을 찾아냄으로써 어떠한 상관관계나 인과관계를 찾아내기 위한 고안된 모델

순환신경망(RNN) 모델

순환신경망의 작동원리



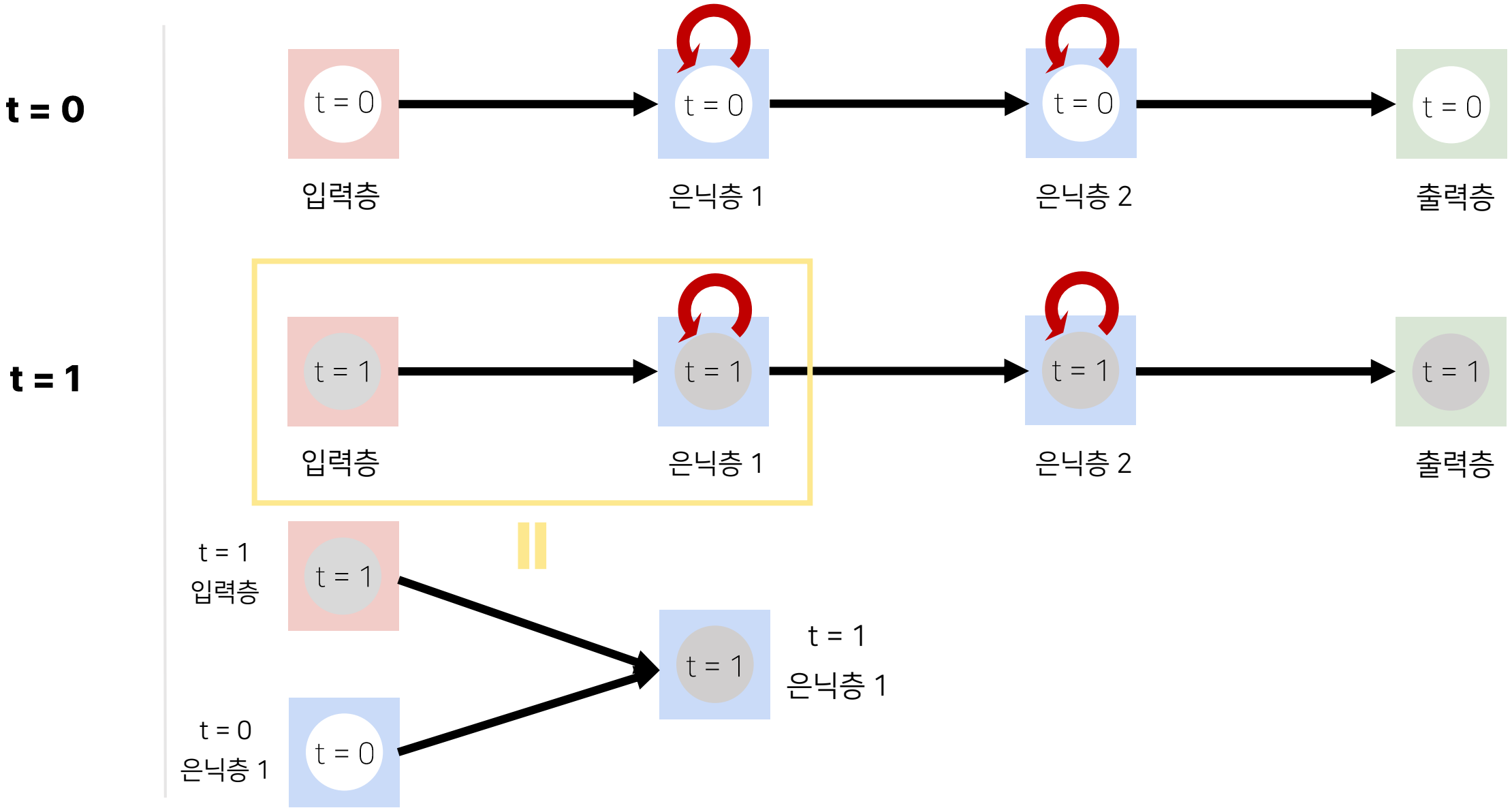
Memory system



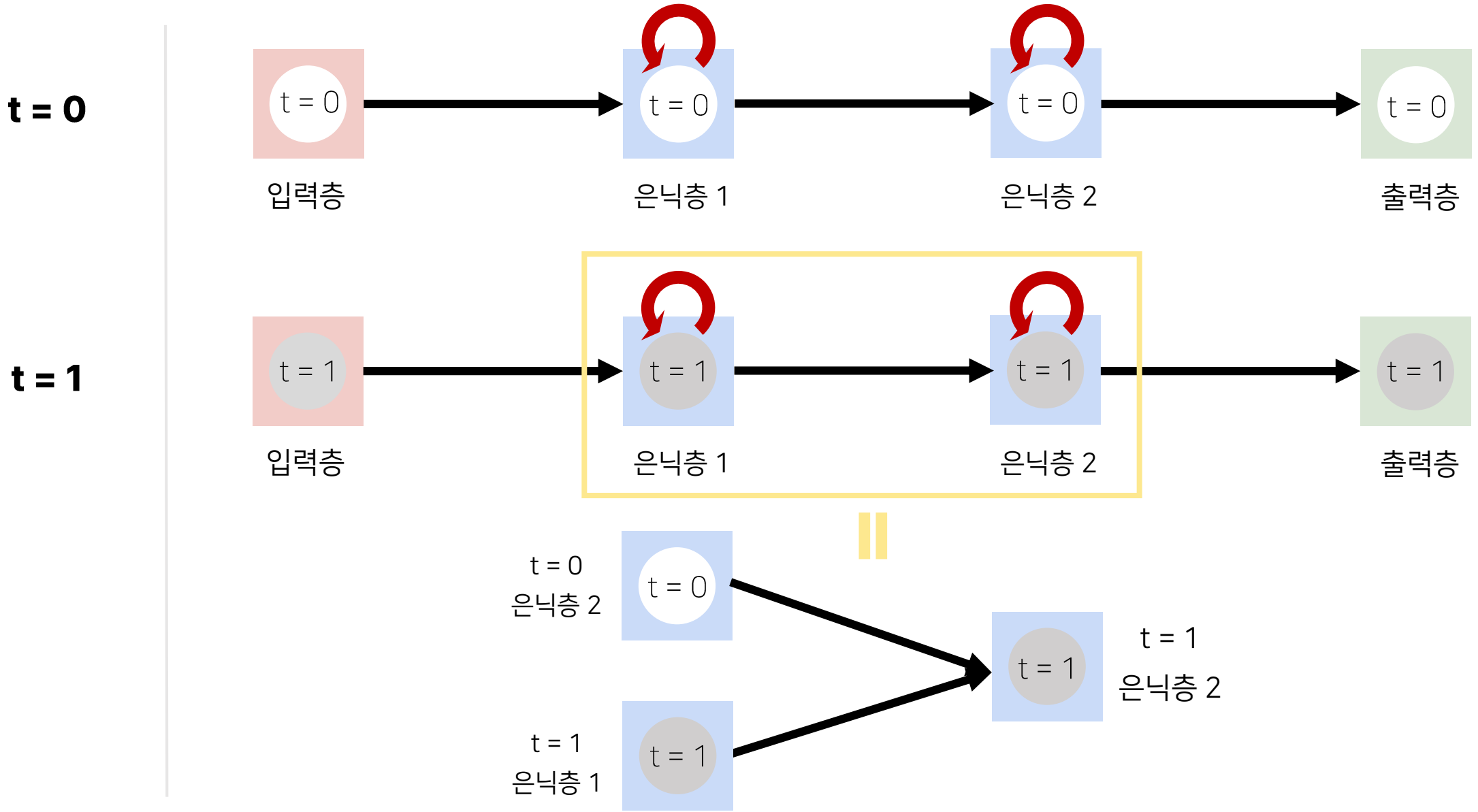
일반적인 인공신경망

순환성을 추가한 인공신경망

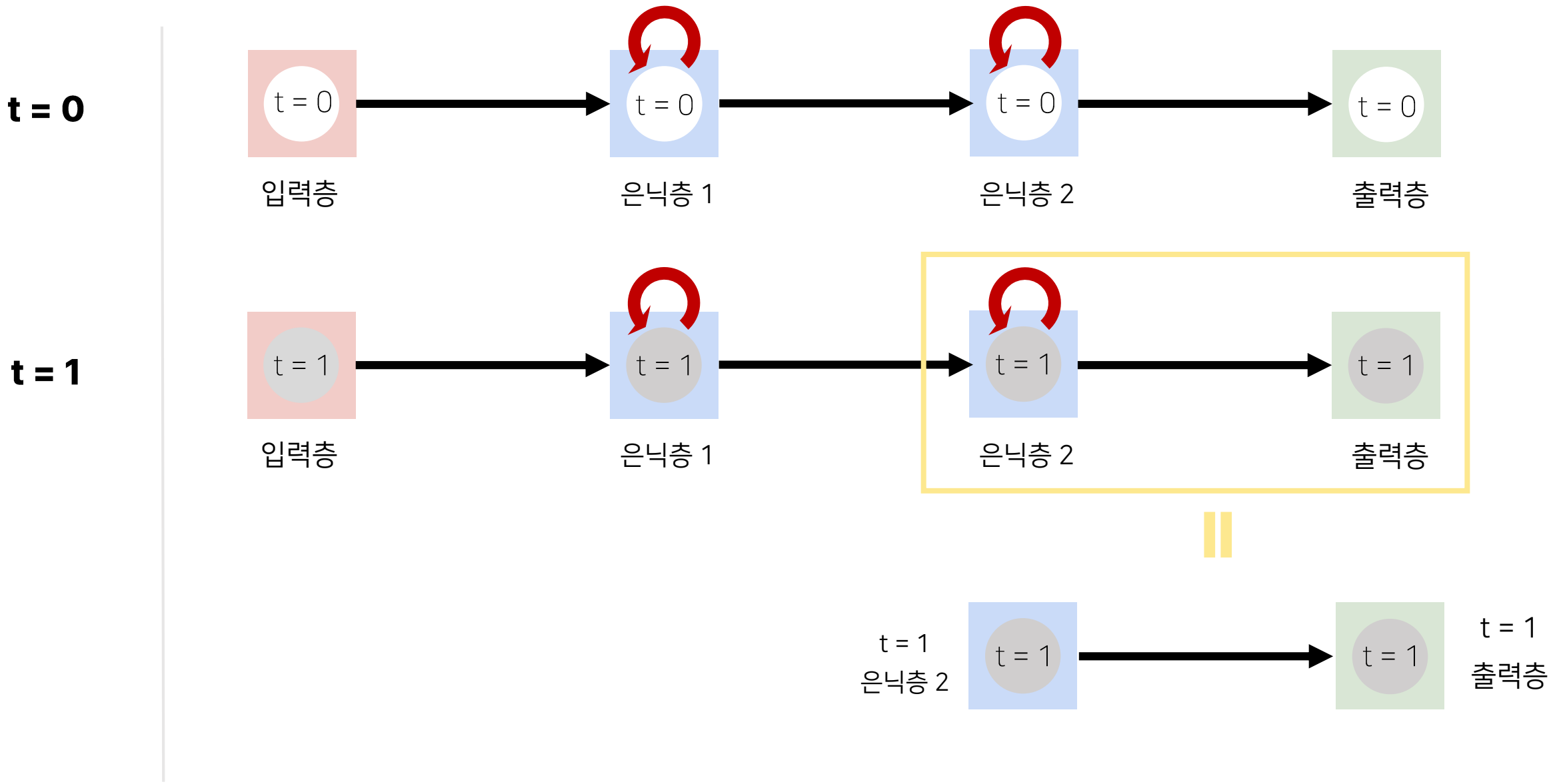
순환신경망의 작동원리


t = 0
t = 0
t = 0
t = 0
t = 0
입력층
은닉층 1
은닉층 2
출력층
t = 1
t = 1
t = 1
t = 1
t = 1
입력층
은닉층 1
은닉층 2
출력층
t = 1
입력층
t = 1
t = 1
은닉층 1
t = 0
은닉층 1
t = 0

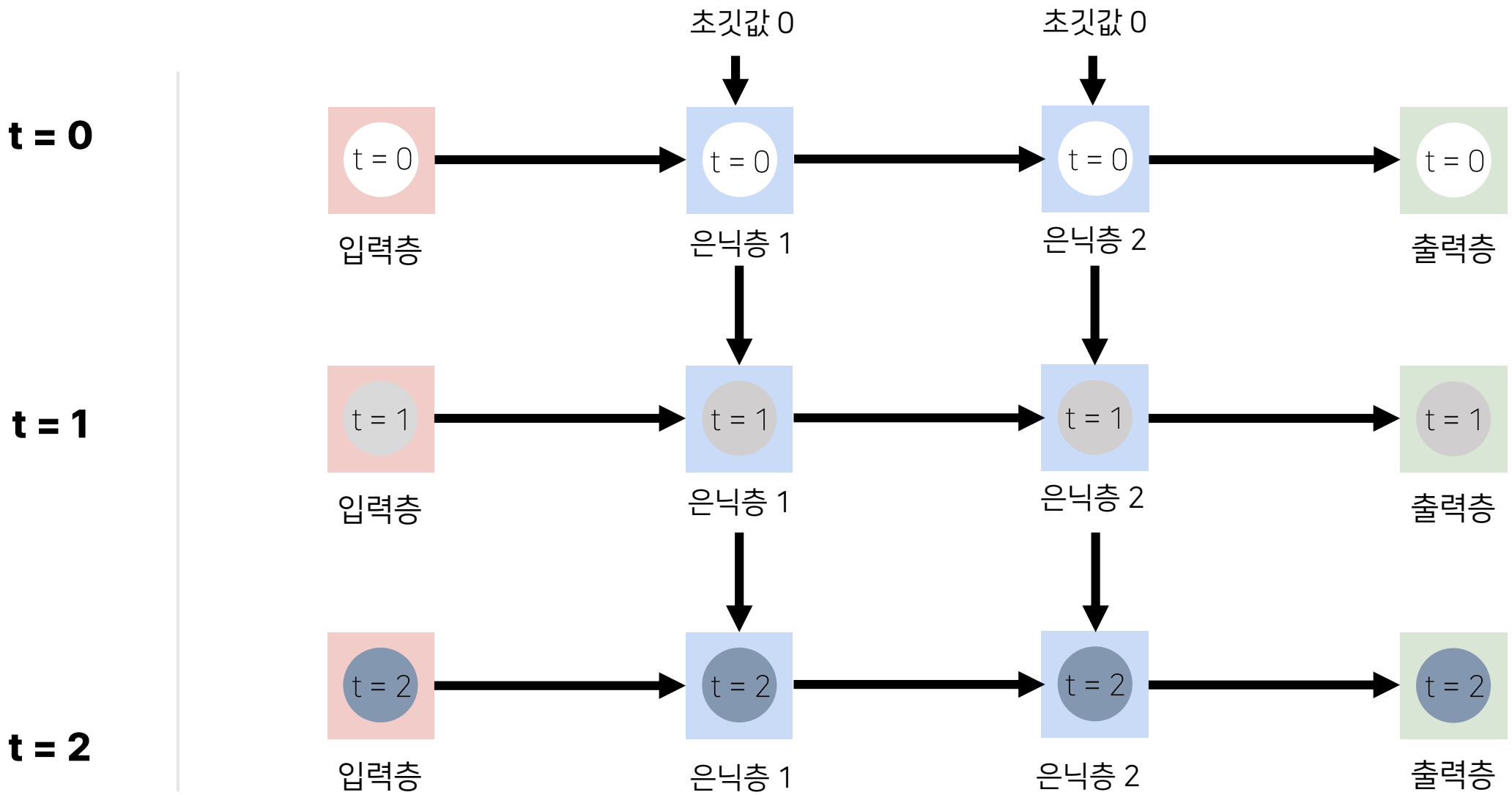
순환신경망의 작동원리


t = 0
t = 0
입력층
t = 0
은닉층 1
t = 0
은닉층 2
t = 0
출력층
t = 1
t = 1
입력층
t = 1
은닉층 1
t = 1
은닉층 2
t = 1
출력층
t = 0
은닉층 2
t = 0
t = 1
은닉층 1
t = 1
t = 1
은닉층 2

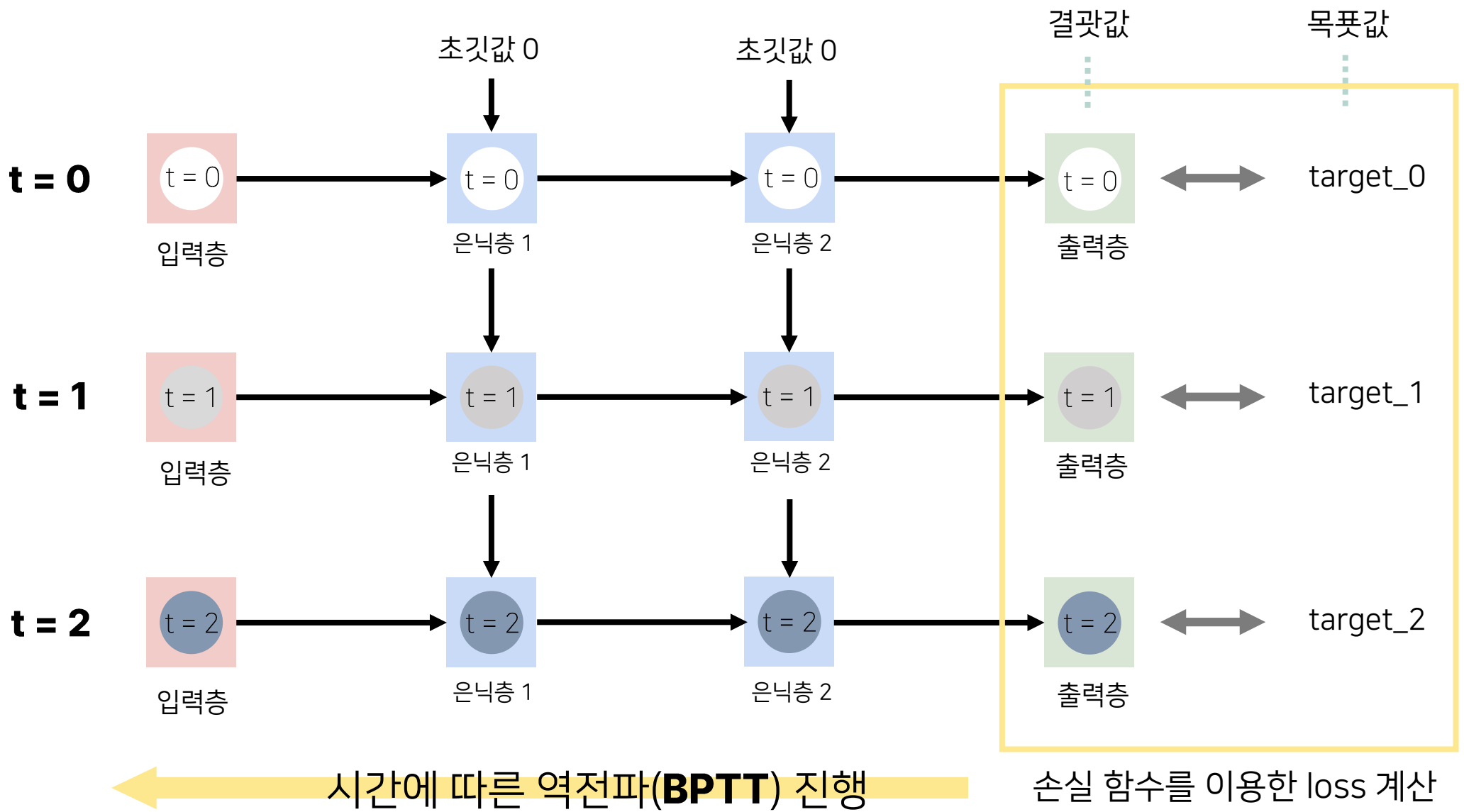
순환신경망의 작동원리


t = 0
t = 0
t = 0
t = 0
t = 0
입력층
은닉층 1
은닉층 2
출력층
t = 1
t = 1
t = 1
t = 1
t = 1
입력층
은닉층 1
은닉층 2
출력층
t = 1
은닉층 2
t = 1
t = 1
t = 1
출력층

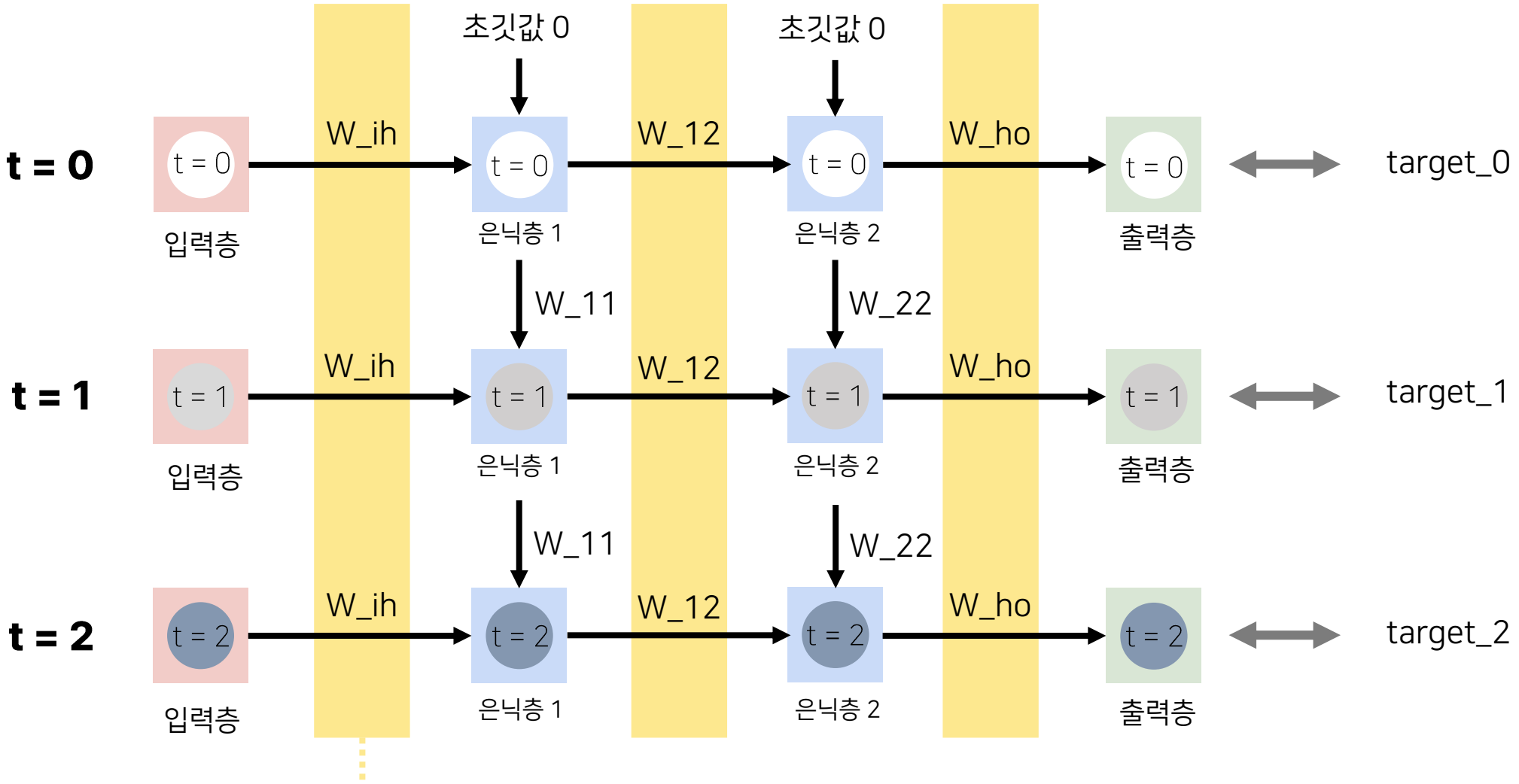
순환신경망의 작동원리


t = 0
t = 1
t = 2
초깃값 0
초깃값 0
t = 0
t = 0
t = 0
t = 0
입력층
은닉층 1
은닉층 2
출력층
t = 1
t = 1
t = 1
t = 1
입력층
은닉층 1
은닉층 2
출력층
t = 2
t = 2
t = 2
t = 2
입력층
은닉층 1
은닉층 2
출력층

순환신경망의 역전파

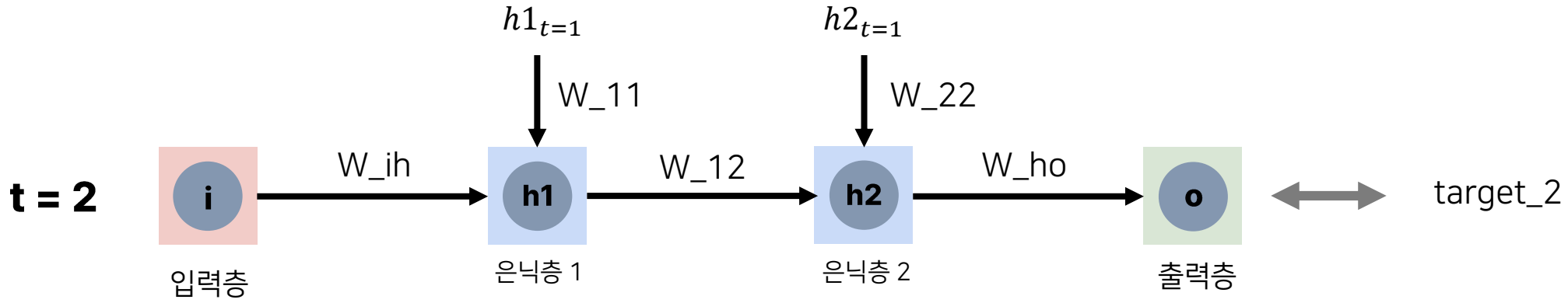

결괏값
목푯값
초깃값 0
초깃값 0
t = 0
t = 1
t = 2
t = 0
t = 1
t = 2
t = 0
t = 1
t = 2
t = 0
t = 1
t = 2
입력층
입력층
입력층
은닉층 1
은닉층 2
은닉층 1
은닉층 2
은닉층 1
은닉층 2
출력층
출력층
출력층
target_0
target_1
target_2
시간에 따른 역전파(BPTT) 진행
손실 함수를 이용한 loss 계산

순환신경망의 역전파



순환 신경망은 각 위치 별로 같은 가중치 공유

순환신경망의 역전파



현재 상태의 output 'o' 는 h2out을 전달받아 갱신되는 구조

$$o = w_{h0} \times h2_{out} + bias$$

기본적으로 순환 신경망의 hidden state는
hyperbolic tangent activation function을 사용

$$h2_{out} = \tanh(w_{12} \times h1 + w_{22} \times h2_{t=1} + bias)$$

$$h1_{out} = \tanh(w_{ih} \times i + w_{11} \times h1_{t=1} + bias)$$

$$h2_{in} = w_{12} \times h1 + w_{22} \times h2_{t=1} + bias$$

$$h1_{in} = w_{ih} \times i + w_{11} \times h1_{t=1} + bias$$

시점별로 t=2 시점에 발생한 손실은 t=2,1,0 시점에 전부
영향을 주고 t = 1시점의 손실은 t=1,0에 영향을,
t = 0시점의 손실은 t = 0의 가중치에 영향을 준다.

실제 업데이트를 할 때는 가중치에 대해 시점 별
기울기를 다 더해서 한번에 업데이트

$$\frac{\partial_o}{\partial w_{22}} = \frac{\partial_o}{\partial h2_{out}} \times \frac{\partial h2_{out}}{\partial h2_{in}} \times \boxed{\frac{\partial h2_{in}}{\partial w_{22}}} = h2_{t=1}$$

순환신경망의 구현


```
n_hidden = 35 # 순환 신경망의 노드 수
lr = 0.01
epochs = 1000

string = "hello pytorch. how long can a rnn cell remember?"
chars = "abcdefghijklmnopqrstuvwxyz ?!.,:;01"
char_list = [i for i in chars]
n_letters = len(char_list)
```

```
def string_to_onehot(string):

    start = np.zeros(shape=len(char_list), dtype=int)
    end = np.zeros(shape=len(char_list), dtype=int)
    start[-2] = 1
    end[-1] = 1

    for i in string:
        idx = char_list.index(i)
        zero = np.zeros(shape=n_letters, dtype=int)
        zero[idx]=1

        start = np.vstack([start,zero])

    output = np.vstack([start,end])

    return output
```

순환신경망의 구현


```
def onehot_to_word(onehot_1):
    # 텐서를 입력으로 받아 넘파이 배열로 바꿔준다.
    onehot = torch.Tensor.numpy(onehot_1)
    # one-hot 벡터의 최댓값(=1) 위치 인덱스로 문자를 찾는다.
    return char_list[onehot.argmax()]
```

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.i2h = nn.Linear(input_size, hidden_size)
        self.h2h = nn.Linear(hidden_size, hidden_size)
        self.i2o = nn.Linear(hidden_size, output_size)
        self.act_fn = nn.Tanh()

    def forward(self, input, hidden):
        hidden = self.act_fn(self.i2h(input) + self.h2h(hidden))
        output = self.i2o(hidden)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)
```

```
rnn = RNN(n_letters, n_hidden, n_letters) # 35, 35, 35
```

순환신경망의 구현


```
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=lr)
```

```
one_hot = torch.from_numpy(string_to_onehot(string)).type_as(torch.FloatTensor())

for i in range(epochs):
    rnn.zero_grad()
    total_loss = 0
    hidden = rnn.init_hidden()

    for j in range(one_hot.size()[0]-1):
        input_ = one_hot[j:j+1,:]
        target = one_hot[j+1]

        output, hidden = rnn.forward(input_, hidden)
        loss = loss_func(output.view(-1),target.view(-1))
        total_loss += loss
        input_ = output

    total_loss.backward()
    optimizer.step()

    if i % 100 == 0:
        print(f'Epoch {i}₩'s loss: {total_loss.item()}')
```

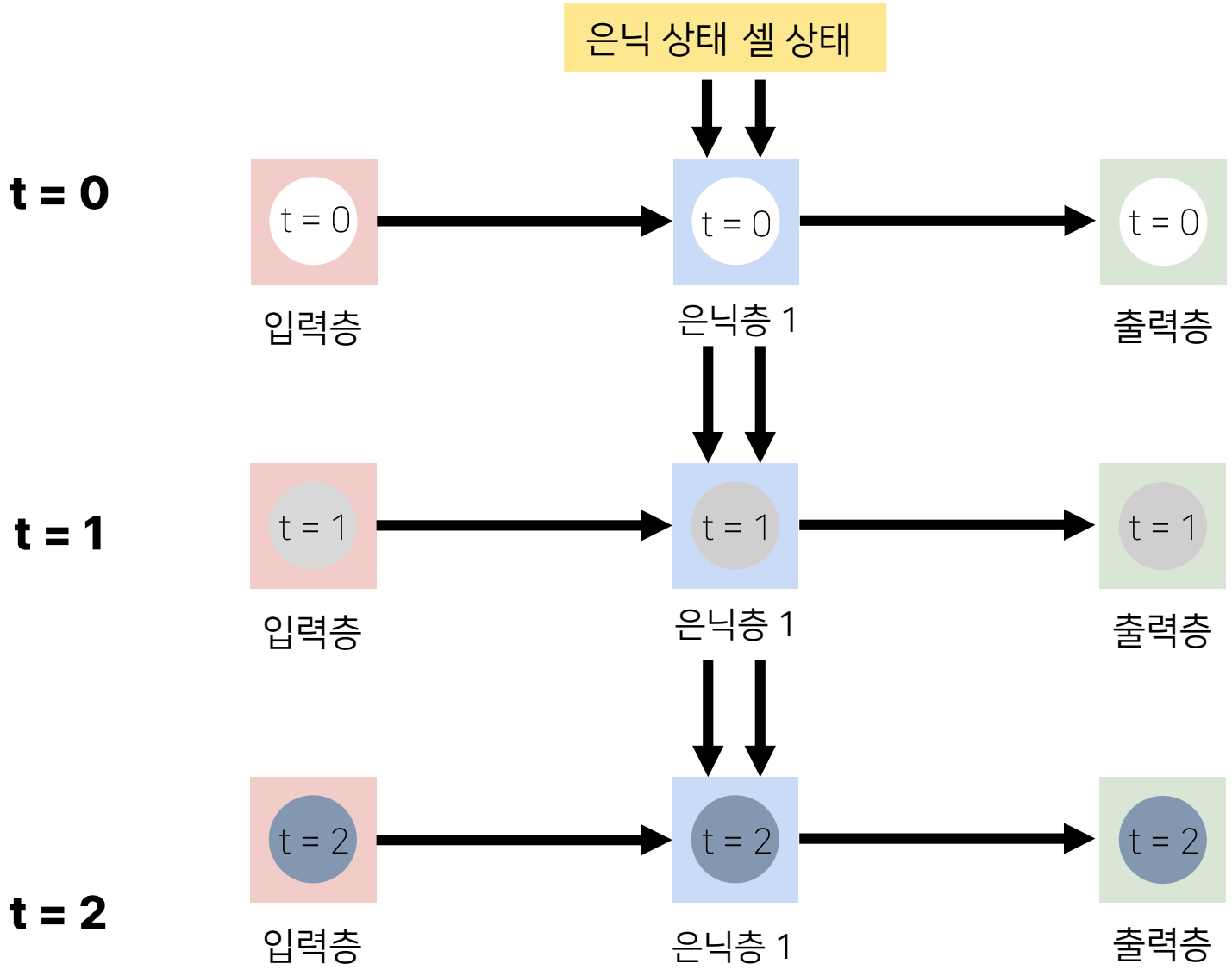
```
Epoch 0's loss: 2.284116268157959
Epoch 100's loss: 0.11178862303495407
Epoch 200's loss: 0.033989317715168
Epoch 300's loss: 0.022304633632302284
Epoch 400's loss: 0.014734550379216671
Epoch 500's loss: 0.014307282865047455
Epoch 600's loss: 0.011135882697999477
Epoch 700's loss: 0.010516466572880745
Epoch 800's loss: 0.0038819138426333666
Epoch 900's loss: 0.003132481360808015
```

순환신경망의 구현

RNN은 time sequence가 늘어나며 역전파 시 hyperbolic tangent 함수의 미분 값이 여러분 곱해진다. 하이퍼볼릭 탄젠트 함수를 미분하면 0에서 1사이의 값이 나오고 기울기 값이 역전파될 때 타임 시퀀스가 길어질수록 모델이 제대로 학습하지 못하는 Vanishing Gradient(기울기 소실) 현상 발생

LSTM 과 GRU 등장

LSTM



기존의 순환 신경망 모델에

장기기억을 담당하는 부분을 추가

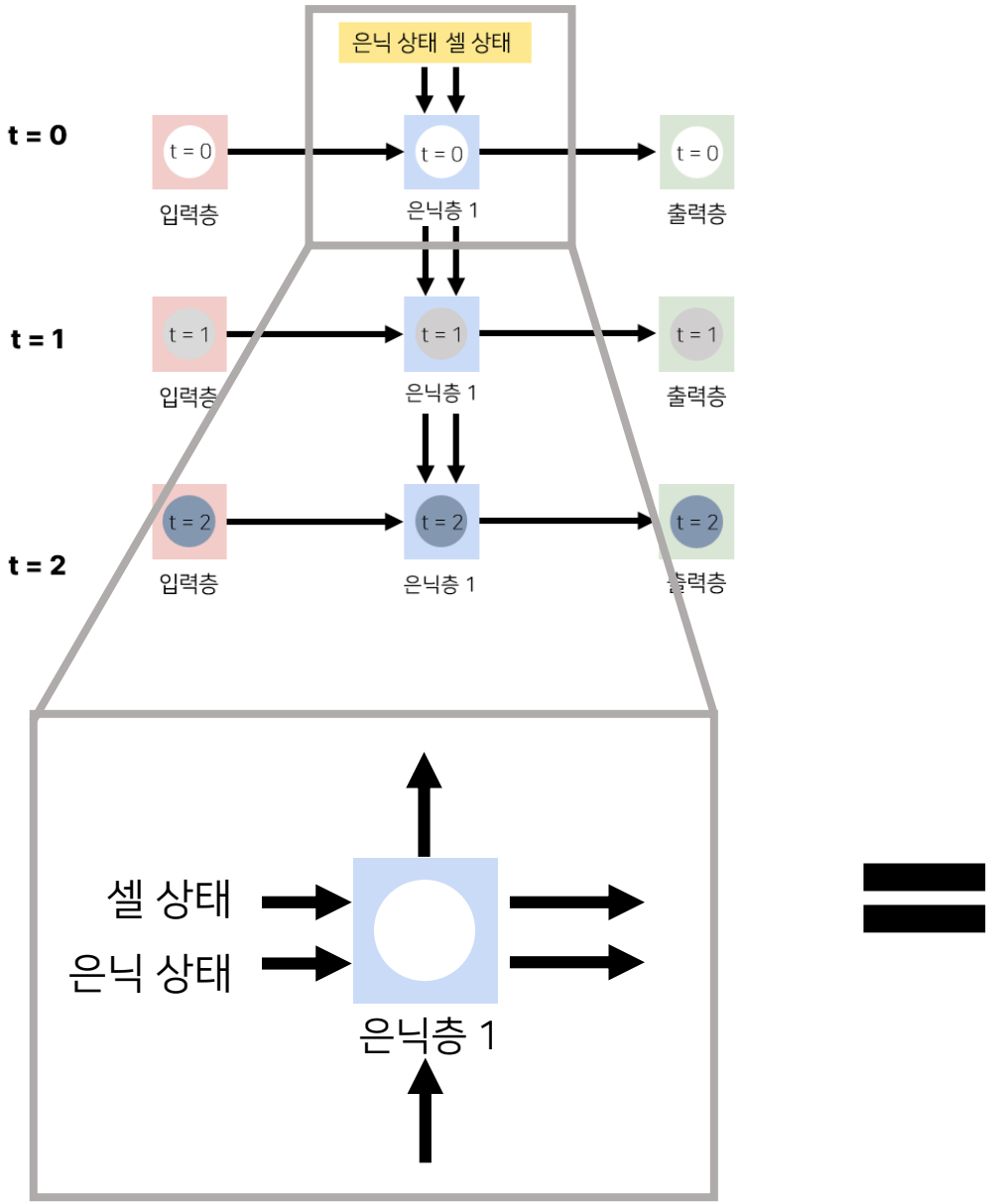
=

기존에는 hidden state만 있었다면

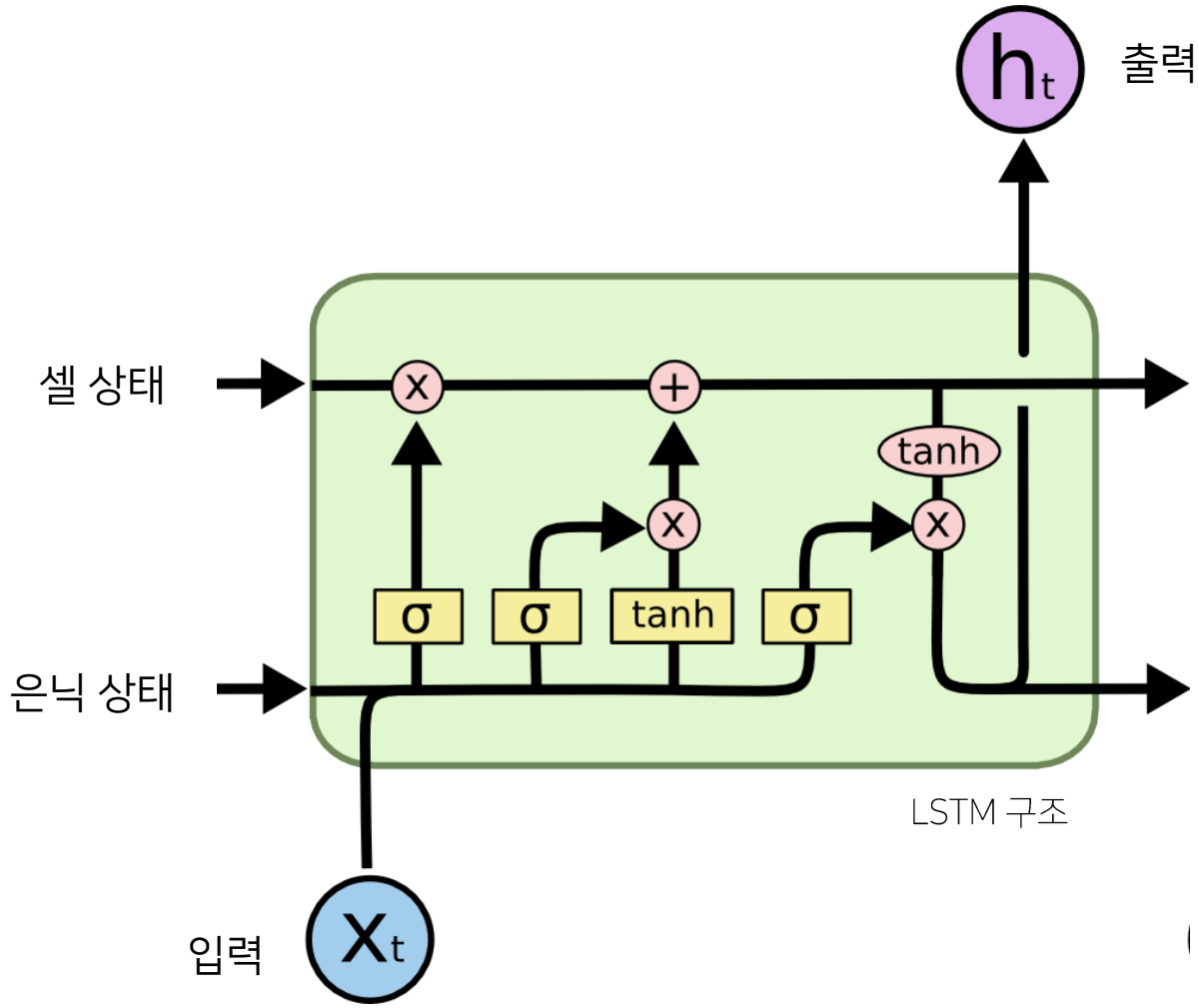
cell state라는 이름을 가지는 전달

부분이 추가된 것

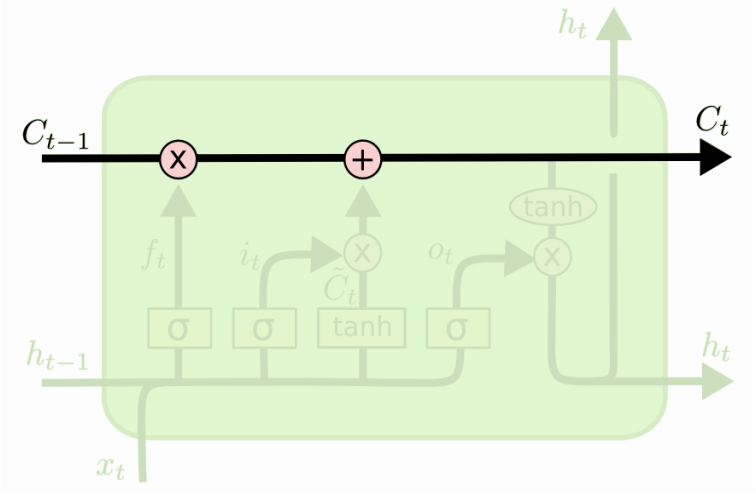
LSTM


은닉 상태 셀 상태
t = 0
t = 1
t = 2
입력층
은닉층 1
출력층
셀 상태
은닉 상태
은닉층 1

=

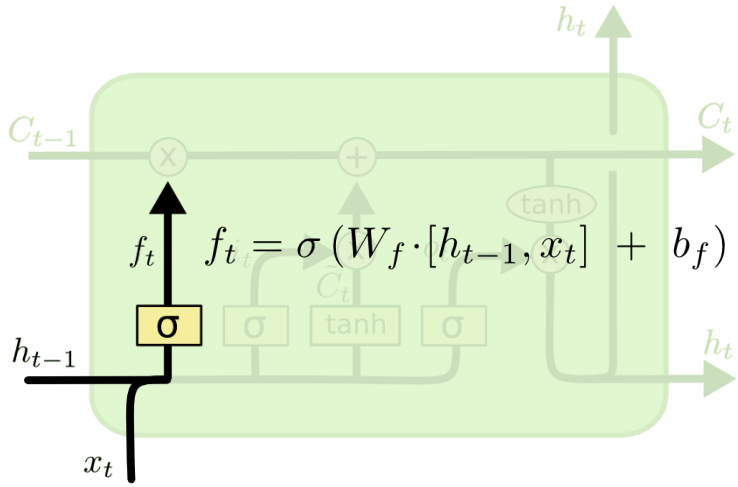

출력
셀 상태
은닉 상태
입력
tanh
σ
σ
tanh
σ
LSTM 구조

LSTM



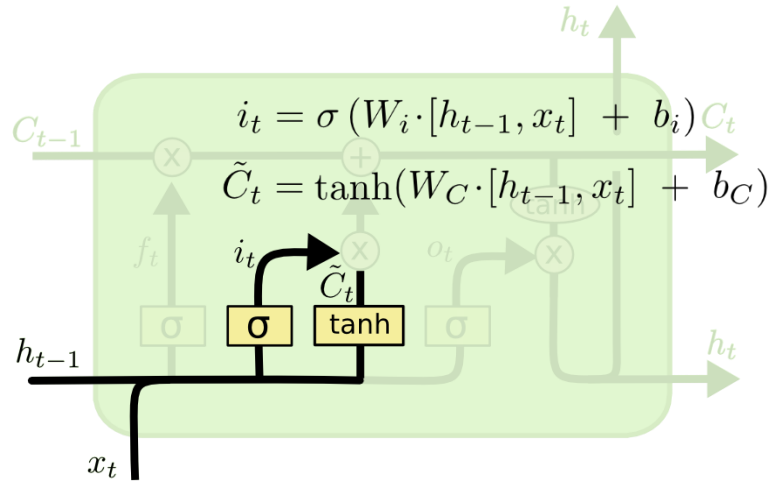
LSTM의 Cell State

셀 상태는 장기기억을 담당하는 부분으로, 곱하기 부분은 기존의 정보를 얼마나 남길 것인지에 따라 비중을 곱하는 부분이고, 더하기 부분은 현재 들어온 데이터와 기존의 은닉 상태를 통해 정보를 추가하는 부분



LSTM의 Forget gate

이름 그대로 기존의 정보들로 구성되어 있는 셀 상태의 값을 얼마나 잊어버릴 것인지를 정하는 부분으로, 현재 시점의 입력값과 직전 시점의 은닉 상태 값을 입력으로 받는 한층의 인공신경망으로 이해

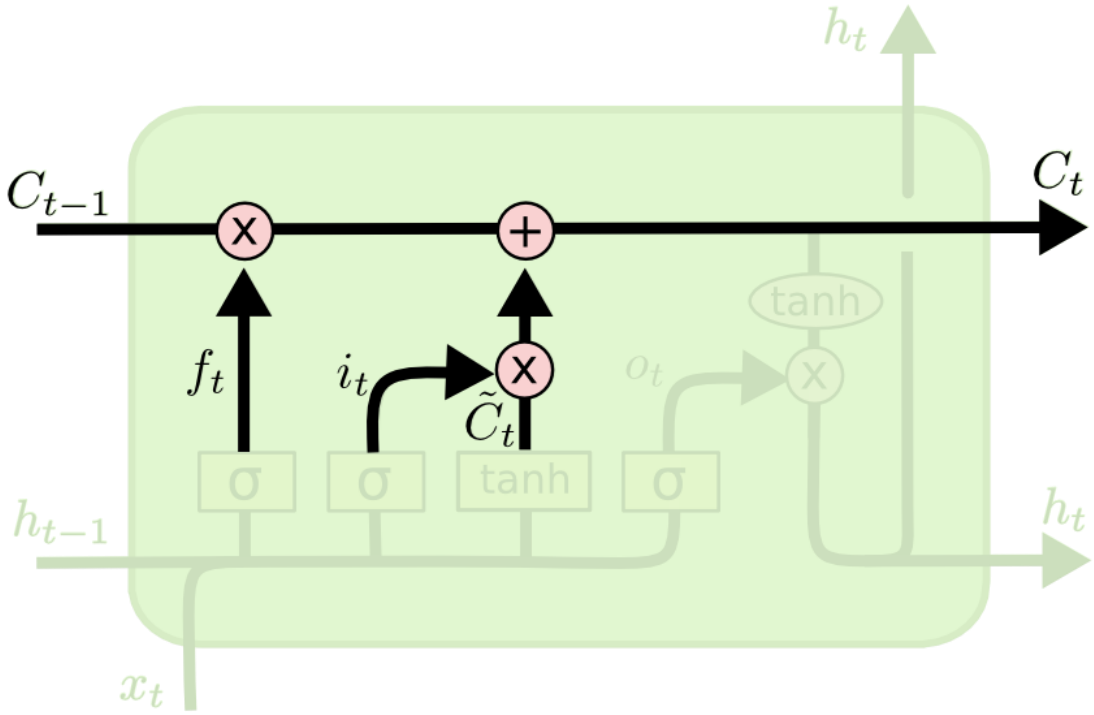


LSTM의 input gate

어떤 정보를 얼마큼 셀 상태에 새롭게 저장할 것인지 정하는 부분으로, 새로운 입력값과 직전 시점의 은닉 상태 값을 받아서 한번은 sigmoid 함수를 통과시키고 한번은 hyperbolic tangent 함수를 통과 시킨다.

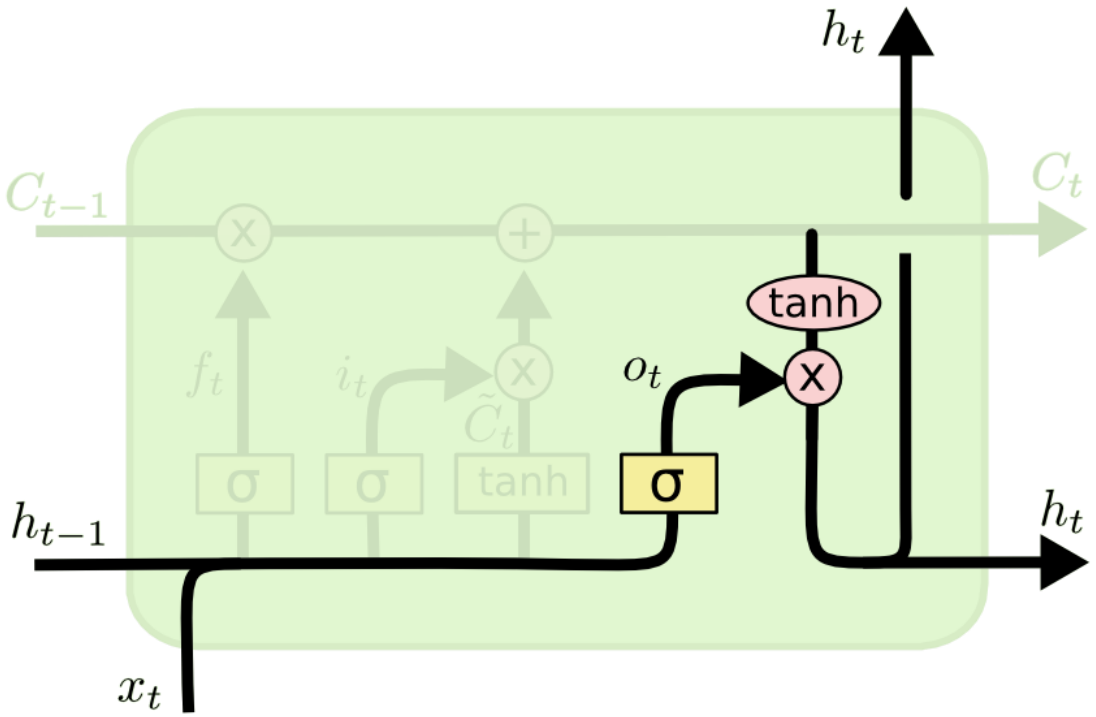
LSTM

LSTM의 Cell State Update



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTM의 Hidden State Update

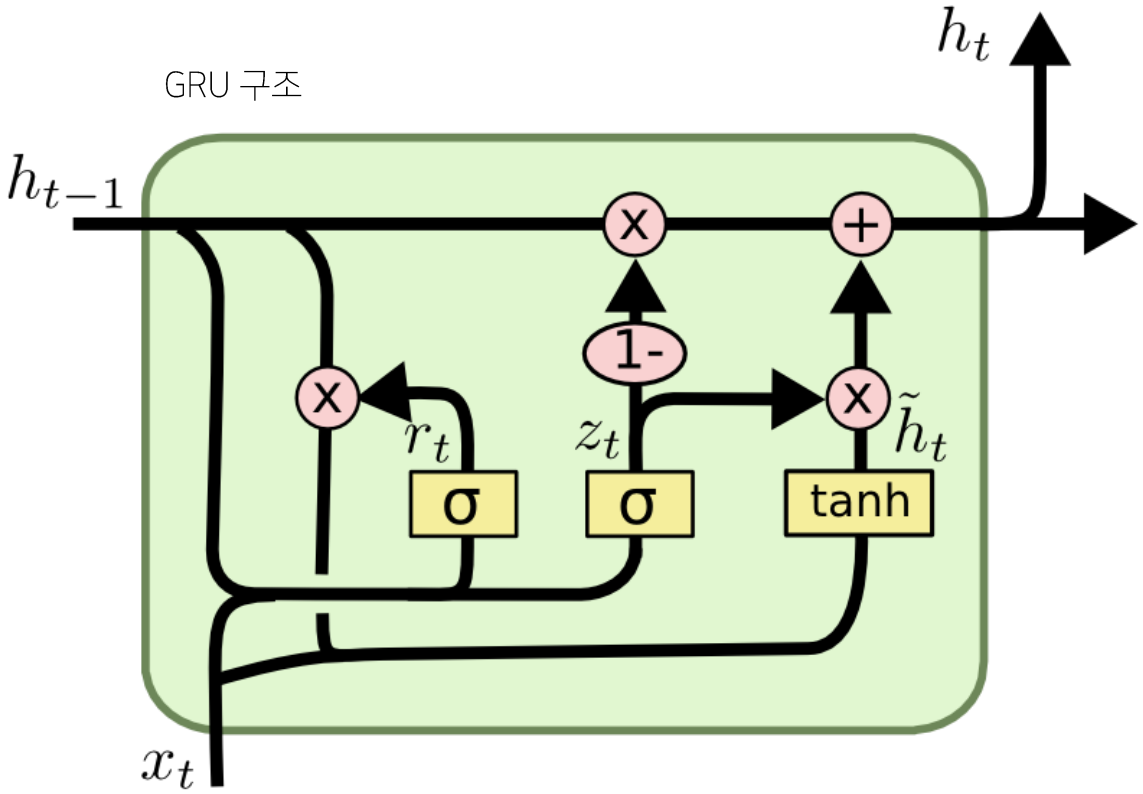


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

GRU

GRU 구조



$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRU는 LSTM과는 달리 **셀 상태와 은닉 상태를 분리하지 않고 은닉 상태 하나로 합친 형태**

update gate로 현재 시점의 새로운 입력값과 직전 시점의 은닉 상태 값에 가중치를 곱하고 시그모이드 함수를 통과시켜 업데이트할 비중을 결정하는 부분

reset gate로 업데이트 게이트와 같은 입력을 받아서 동일하게 sigmoid 함수를 통해 비중을 정하며, 이 비중은 다음 줄 수식을 구할 때 기존 은닉 상태 값을 얼만큼 반영하는지 사용

기존 은닉 상태에 가중치가 곱해진 값과 새로운 입력 값을 입력으로 받아 가중치를 곱한 후 하이퍼볼릭 탄젠트 함수를 통과해 새로운 정보 값을 리턴

새로운 은닉 상태를 구분

Embedding의 등장

기존의 자연어 처리 분야에서 단어를 표현하는 방법?

One-hot Vector

- 기존의 one-hot vector는 단순하다는 장점이 있지만 단어를 단순히 index에 따른 vector로 표현하기 때문에 여러 단어 간 유사성을 평가할 수가 없음
- 또한 사전의 단어 개수가 증가하는 경우 one-hot vector의 크기가 지나치게 커진다는 단점을 가짐

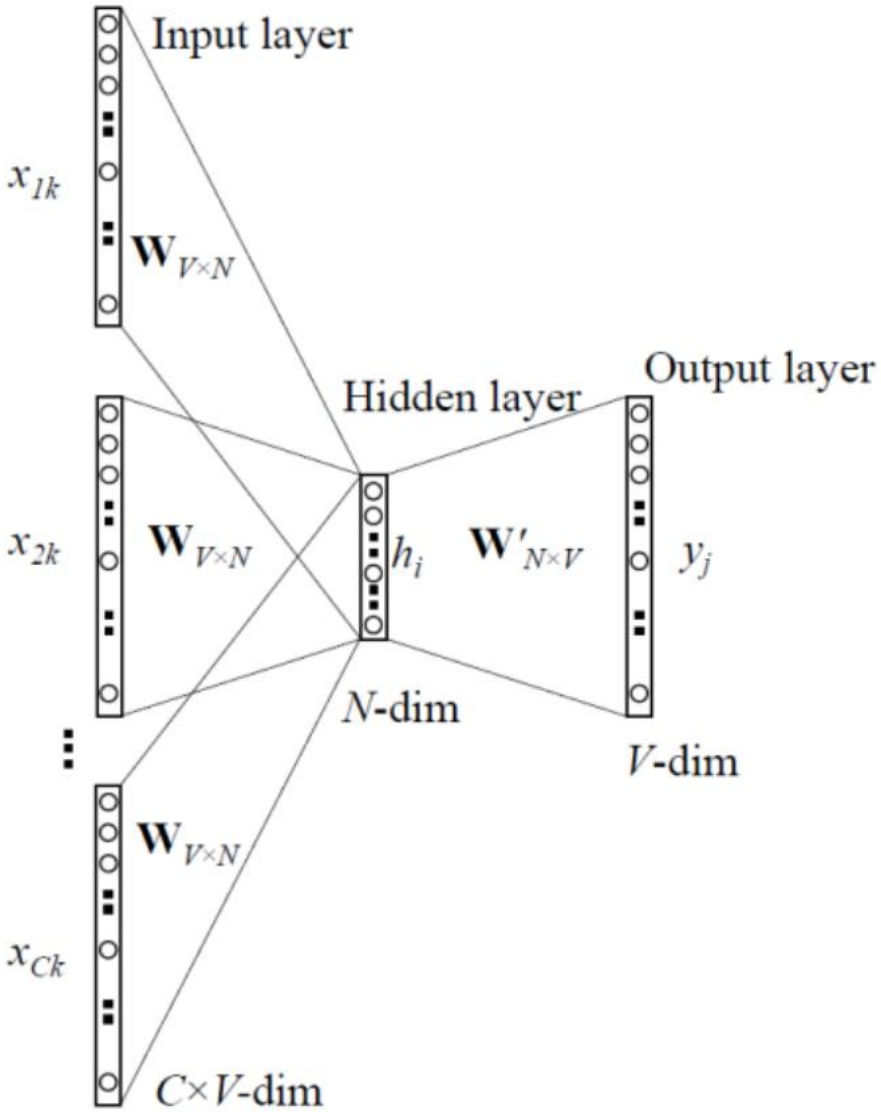
0	1	0	0	0
nlp	python	word	ruby	one-hot

Word embedding

- word embedding은 단어를 특징을 가지는 N 차원의 vector로 표현하는 것
- 기존의 one-hot vector가 각 단어를 사전의 개수만큼의 차원으로 표현하는 것과 다르게 각 단어를 특징을 가지는 N차원 vector로 표현

python	0.52	0.21	0.37	…
ruby	0.48	0.21	0.33	…
word	0.05	0.23	0.06	…

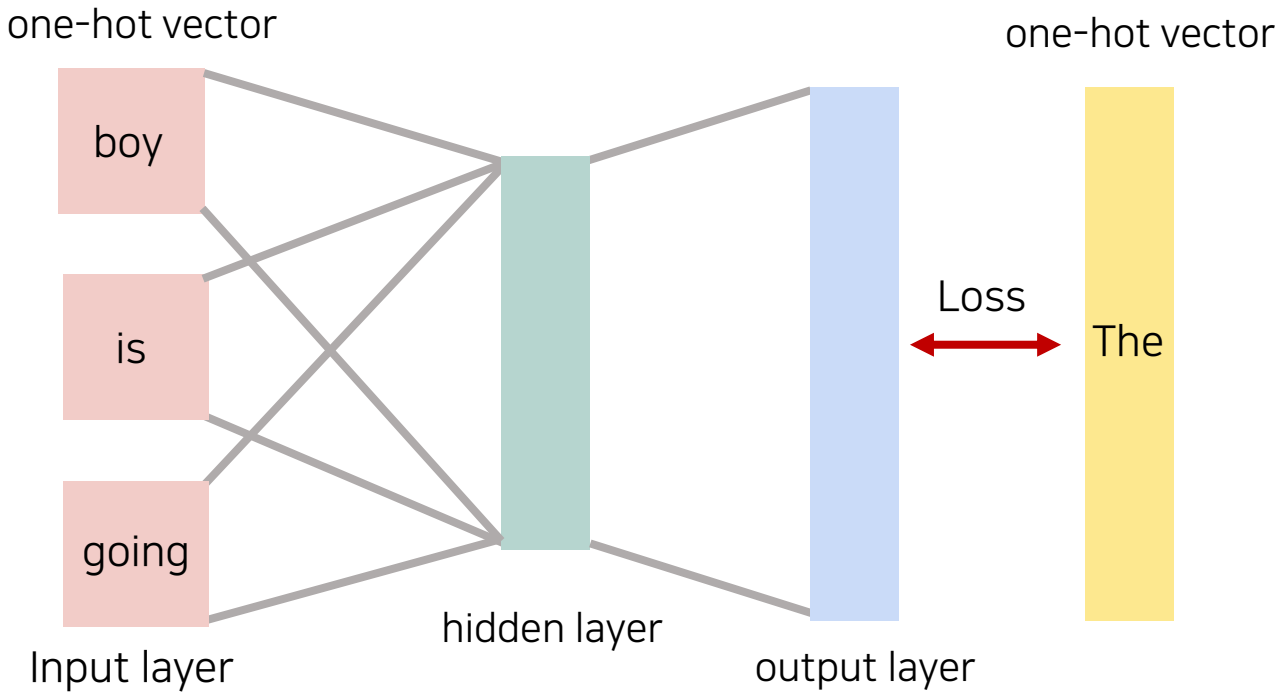
Embedding



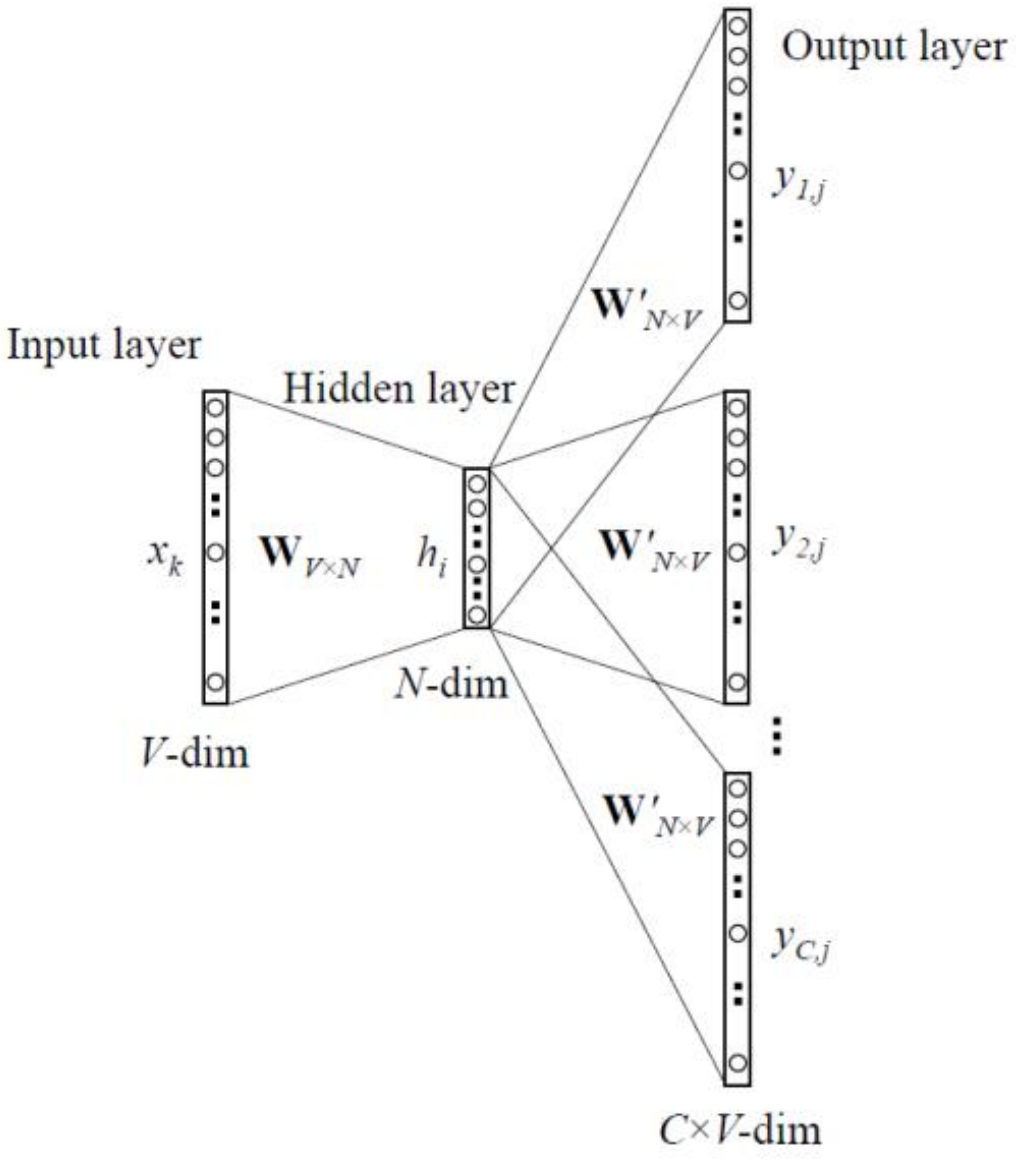
CBOW(continuous bag-of-words)

CBOW 방식은 주변 단어들로부터 가운데 들어갈 단어가 나오도록 하는 임베딩 방식으로 context가 주어졌을 때, 기준 단어에 대해 앞 뒤로 N/2개씩 , 총 N개의 문맥단어를 입력으로 사용하여 기준단어를 맞추기 위한 네트워크를 생성

EX. “The boy is going to school”



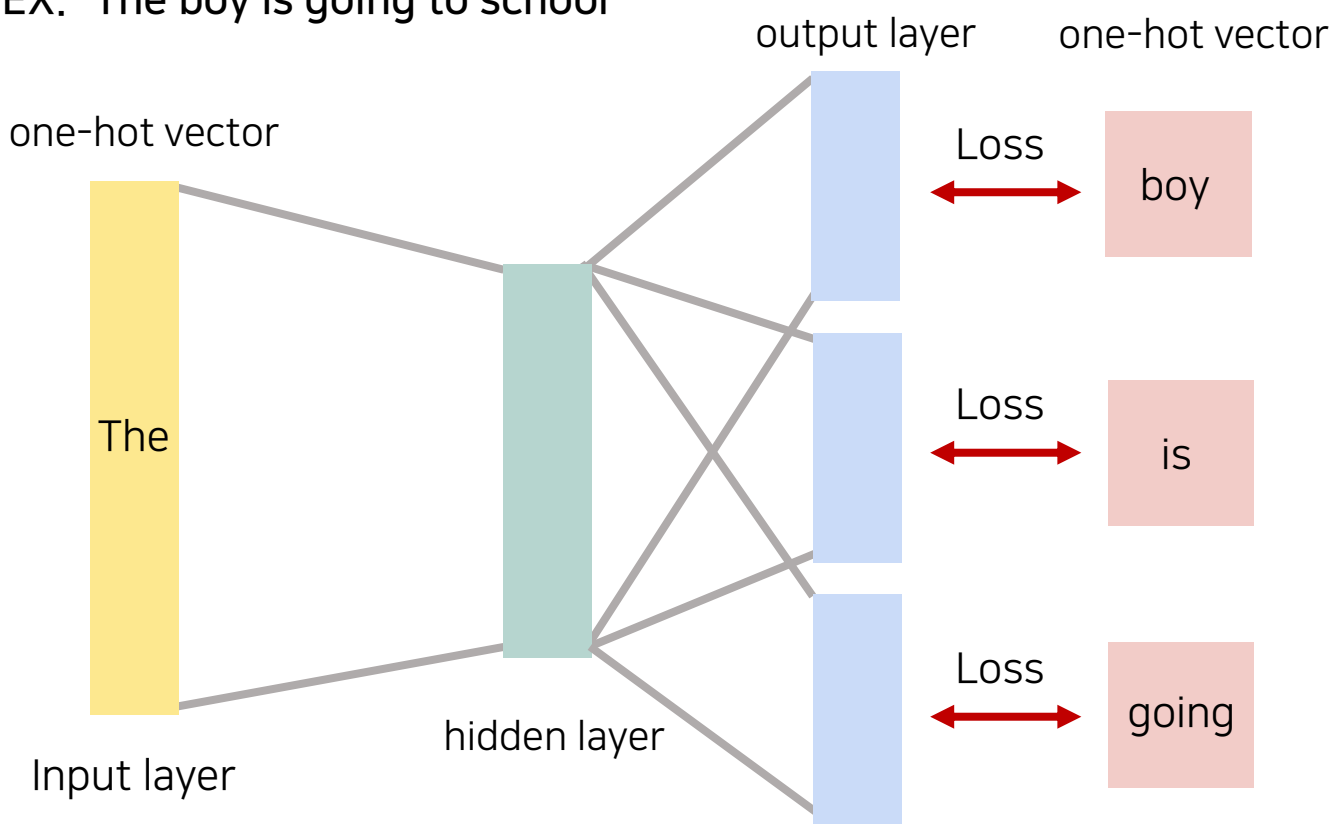
Embedding



skip-gram

skip-gram 모델은 CBOW와는 반대로 중심 단어로부터 주변 단어들이 나오도록 모델을 학습하여 임베딩 벡터를 얻는 방식으로 context가 주어졌을 때, 기준 단어를 입력으로 사용하여, 기준단어에 대해 앞 뒤로 N/2개 씩 총 N개의 문맥 단어를 맞추기 위한 네트워크

EX. “The boy is going to school”



Embedding을 이용한 GRU


```
class RNN(nn.Module):
    def __init__(self, input_size, embedding_size, hidden_size, output_size, num_layers=1):
        super(RNN, self).__init__()
        self.input_size = input_size
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.encoder = nn.Embedding(self.input_size, self.embedding_size)
        self.rnn = nn.GRU(self.embedding_size,self.hidden_size,self.num_layers)
        self.decoder = nn.Linear(self.hidden_size, self.output_size)


    def forward(self, input, hidden):
        out = self.encoder(input.view(1,-1))
        out,hidden = self.rnn(out,hidden)
        out = self.decoder(out.view(batch_size,-1))
        return out,hidden

    def init_hidden(self):
        hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size)
        return hidden

model = RNN(n_characters, embedding_size, hidden_size, n_characters, num_layers)
```

Embedding을 이용한 LSTM


```
class RNN(nn.Module):
    def __init__(self, input_size, embedding_size, hidden_size, output_size, num_layers=1):
        super(RNN, self).__init__()
        self.input_size = input_size
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.encoder = nn.Embedding(self.input_size, self.embedding_size)
        self.rnn = nn.LSTM(self.embedding_size,self.hidden_size,self.num_layers)
        self.decoder = nn.Linear(self.hidden_size, self.output_size)


    def forward(self, input, hidden, cell):
        out = self.encoder(input.view(1,-1))
        out,(hidden,cell) = self.rnn(out,(hidden,cell))
        out = self.decoder(out.view(batch_size,-1))
        return out,hidden,cell

    def init_hidden(self):
        hidden = torch.zeros(self.num_layers,batch_size,self.hidden_size)
        cell = torch.zeros(self.num_layers,batch_size,self.hidden_size)
        return hidden,cell


model = RNN(n_characters, embedding_size, hidden_size, n_characters, num_layers)
```