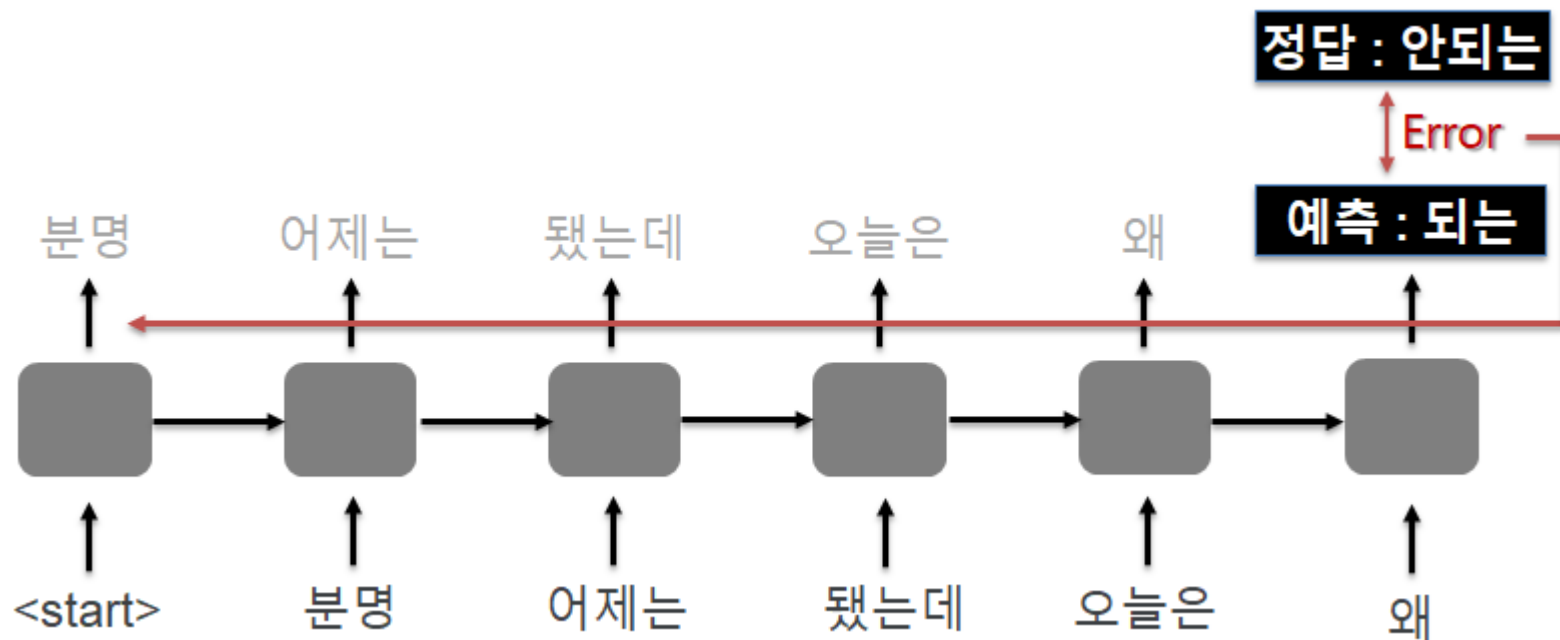# RNN

AILAB
Hanyang Univ.

오늘 실습 내용

1. RNN을 이용한 character prediction
2. LSTM

1. RNN을 이용한 character prediction

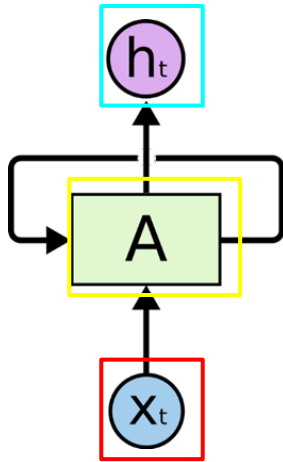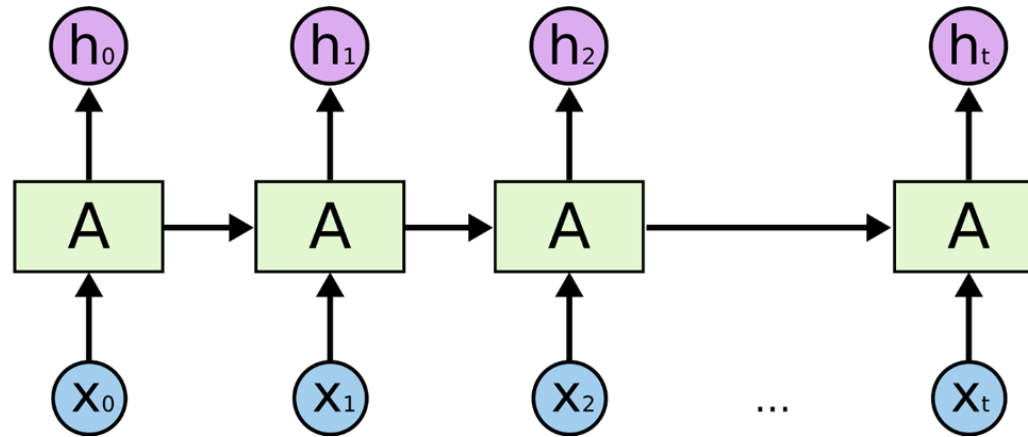# RNN

RNN Example

Sequence data 를 딥러닝으로 처리하려면?
: 이전 출력값이 현재 결과에 영향을 미치는 RNN 구조 활용

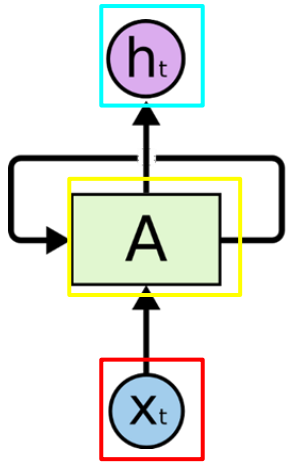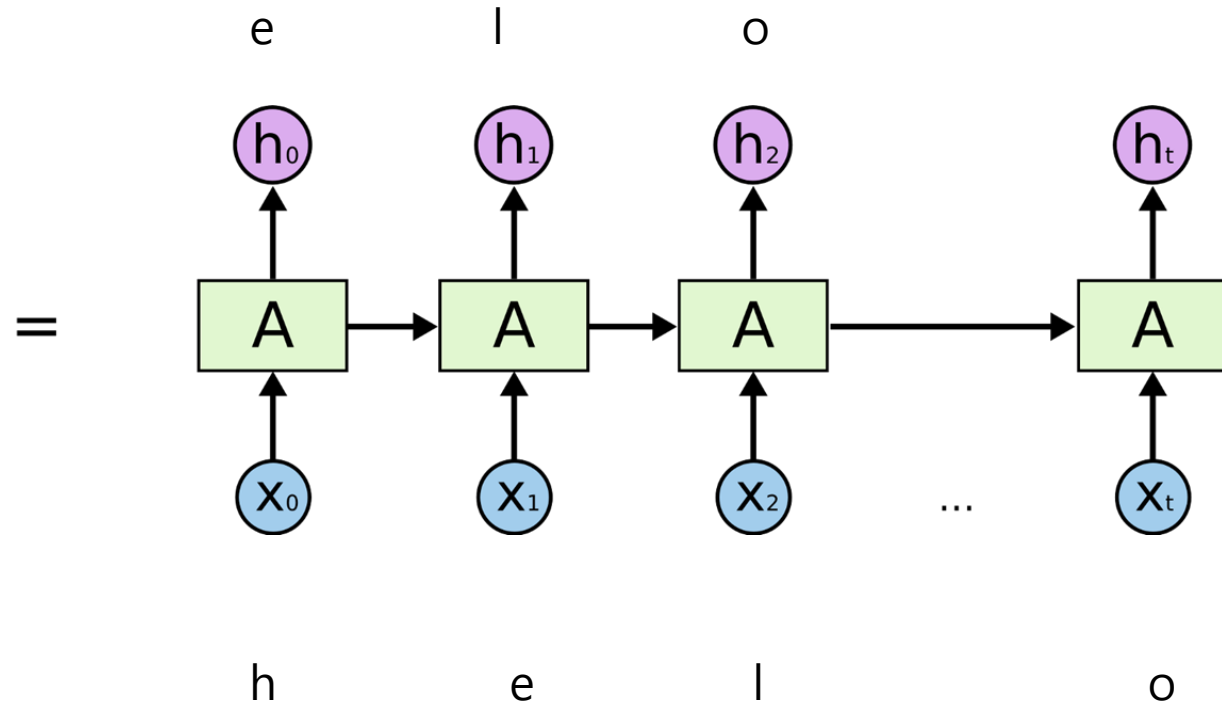RNN을 사용해서 다음 character를 예측할 수 있다!
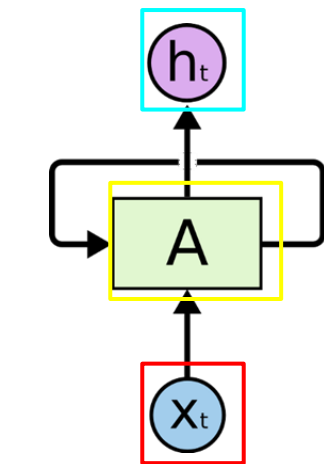
RNN

RNN Architecture

RNN

RNN Architecture

x = hello

character 단위로 각 time step마다 인풋을 활용

RNN

RNN Architecture
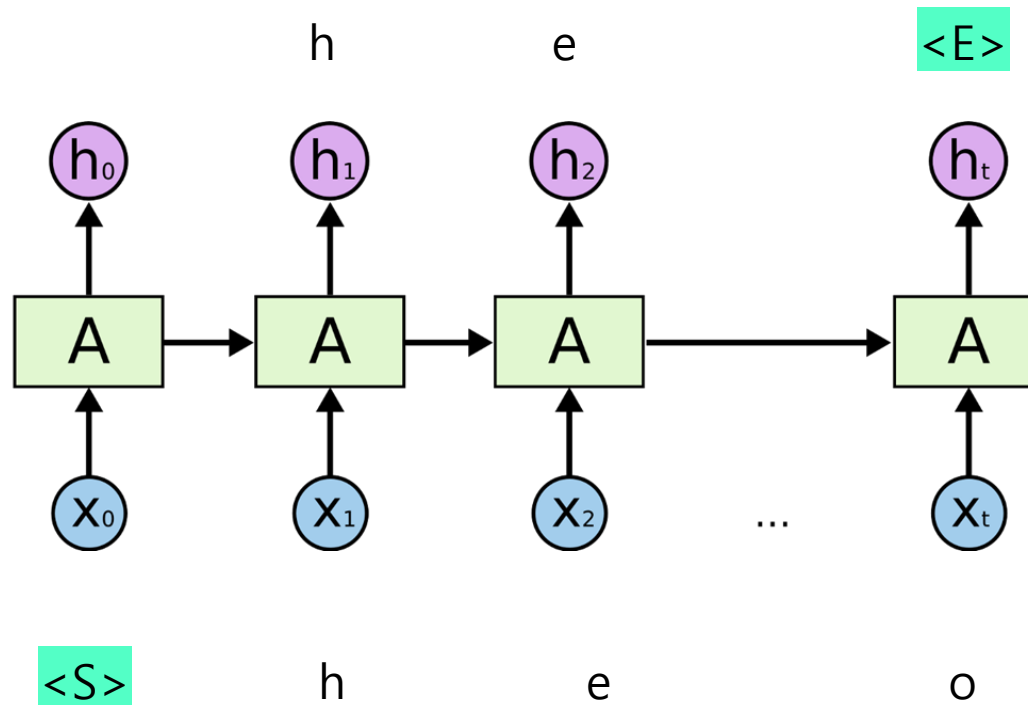
x = hello

보통 <S>, <E> 처럼 문장의 처음과 끝을 구분해주는 token을 추가하여 학습 진행

실습에서는 사용 안함!

RNN

RNN Architecture

e   l   o

h_t = h_0 h_1 h_2 h_t

A   A   A   A   A

x_t x_0 x_1 x_2 ... x_t
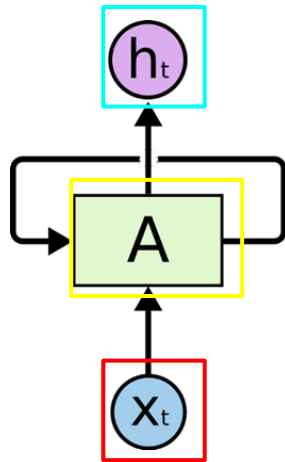
x = hello   h   e   l   o
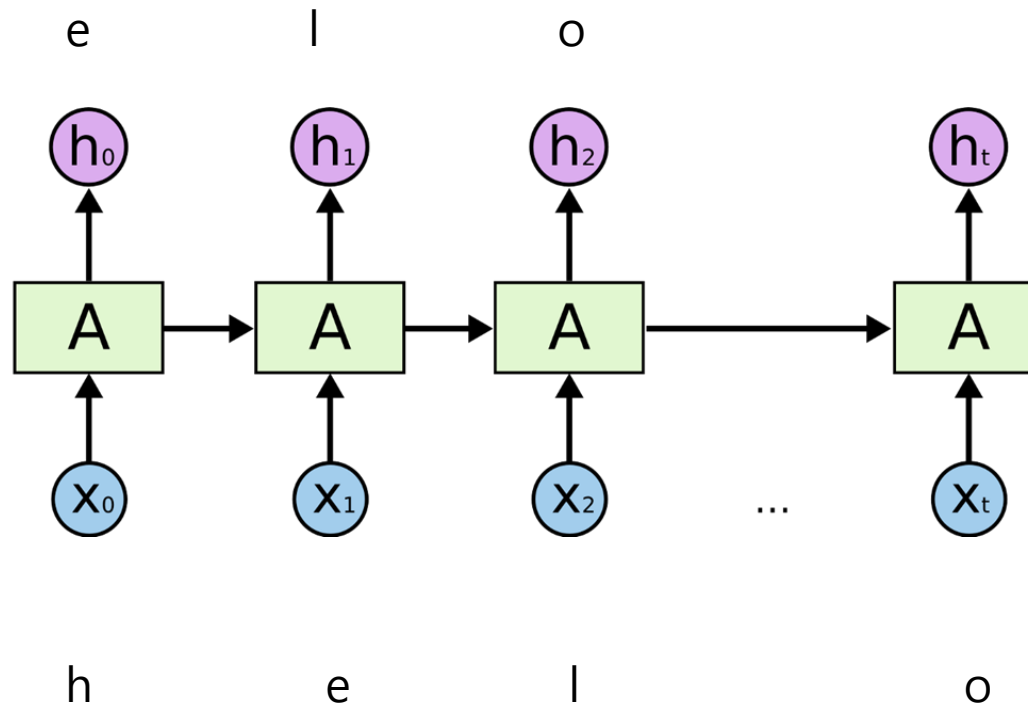
글자를 모델에 넣기 위해 숫자로 바꾸어야함 → one-hot encoding 활용

N=4
```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

구축해 놓은 사전에서 주로 index로 단어를 구분하여 1과 0으로 해당 단어를 나타냄
사전 크기가 N이면 각 벡터는 1xN으로 표현
https://en.wikipedia.org/wiki/One-hot#Natural_language_processing

RNN 구현

torch.nn.RNN

https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

```
rnn = nn.RNN(input_size=4, hidden_size=2, batch_first=True)
```

Parameters

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights b_ih and b_hh. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0

RNN 구현

torch.nn.RNN

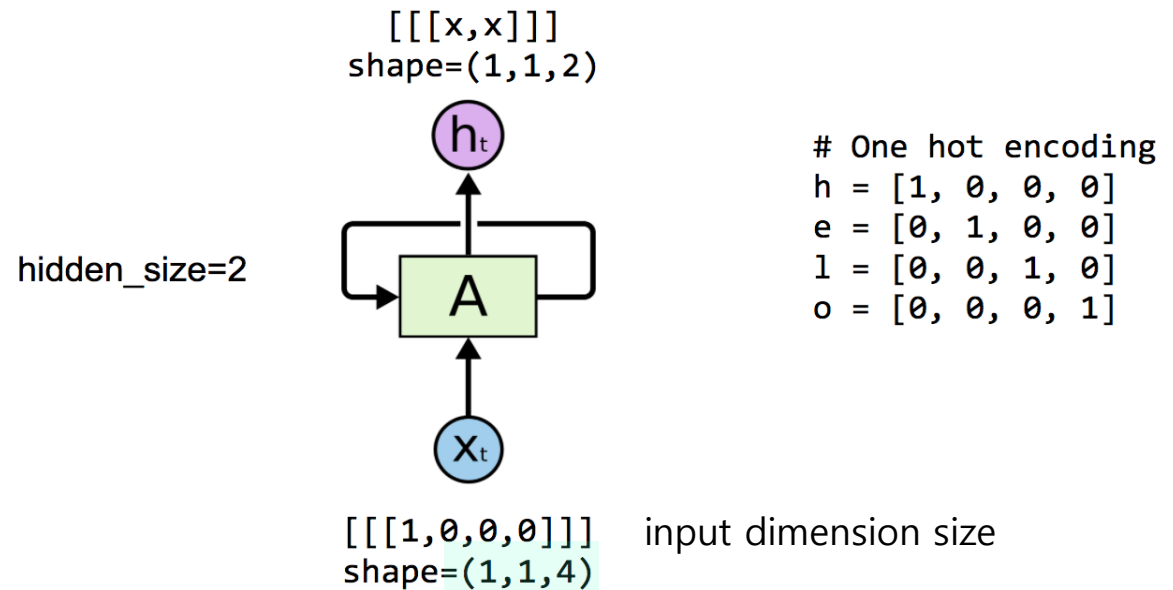https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

```
rnn = nn.RNN(input_size=4, hidden_size=2, batch_first=True)
```

- **input_size** – The number of expected features in the input x

```
[[[x,x]]]
shape=(1,1,2)
```


hidden_size=2

```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

```
[[[1,0,0,0]]]   input dimension size
shape=(1,1,4)
```

RNN 구현

torch.nn.RNN

https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

```
rnn = nn.RNN(input_size=4, hidden_size=2, batch_first=True)
```

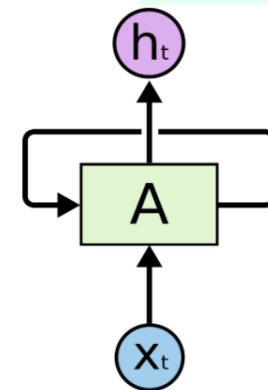- **hidden_size** – The number of features in the hidden state h

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

$$W_{ih} : 4 \times 2, \; W_{hh} : 2 \times 2$$

```
[[[x,x]]]
shape=(1,1,2)
```
hidden size = output dimension

hidden_size=2

```
[[[1,0,0,0]]]
shape=(1,1,4)
```

```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

RNN 구현

torch.nn.RNN

https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

```
rnn = nn.RNN(input_size=4, hidden_size=2, batch_first=True)
```

- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`

```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

hidden_size=2
sequance_length=5

Using word vector

shape=(1,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]]]
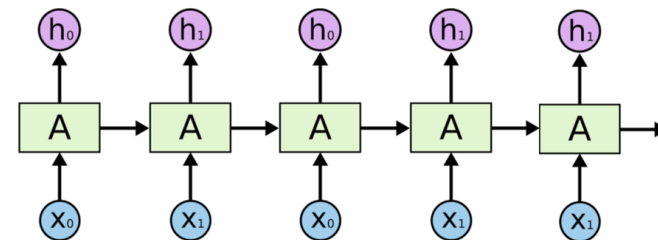
shape=(1,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]]]
h   e   l   l   o

RNN 구현

shape=(1,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]]]

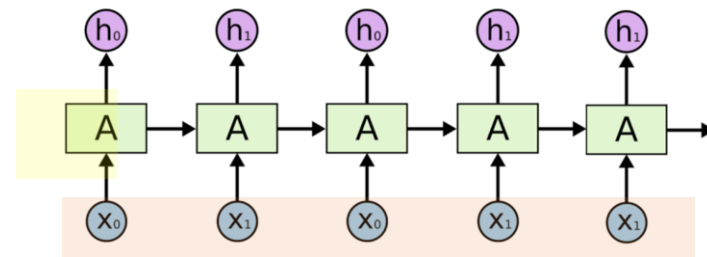shape=(1,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]]]
           h   e   l   l   o

torch.nn.RNN Input

- input 사용시 batch_first=True에 맞춰 tensor shape 설정
- h_0 설정하지 않아도 되고 random한 값 생성해서 넣어주어 됨

Inputs: input, h_0

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.

- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for the input sequence batch. Defaults to zeros if not provided.

RNN 구현

shape=(1,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]]]
      h   e   l   l   o

torch.nn.RNN Output

- Batch first에 맞춰서 출력
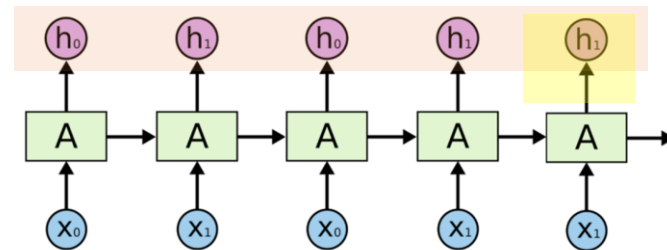
Outputs: output, h_n

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (h_t) from the last layer of the RNN, for each t. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.

- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.

D=1(https://en.wikipedia.org/wiki/Recurrent_neural_network#Bi-directional)

num_layers=1(stacked rnn)

Character Prediction

Loss Function

다음 정답 character를 오게 하기위해 vocab의 전체 단어 중 그 단어의 확률이 높게 만들어야한다.
Linear layer와 softmax활용하여 확률값을 얻어 정답의 확률값을 maximize함 (CrossentropyLoss)

hidden_size=2

[[[1,0,0,0]]]
shape=(1,1,4)

hidden_size=2

Input size = ht size
Output size = vocab size

[[[1,0,0,0]]]
shape=(1,1,4)
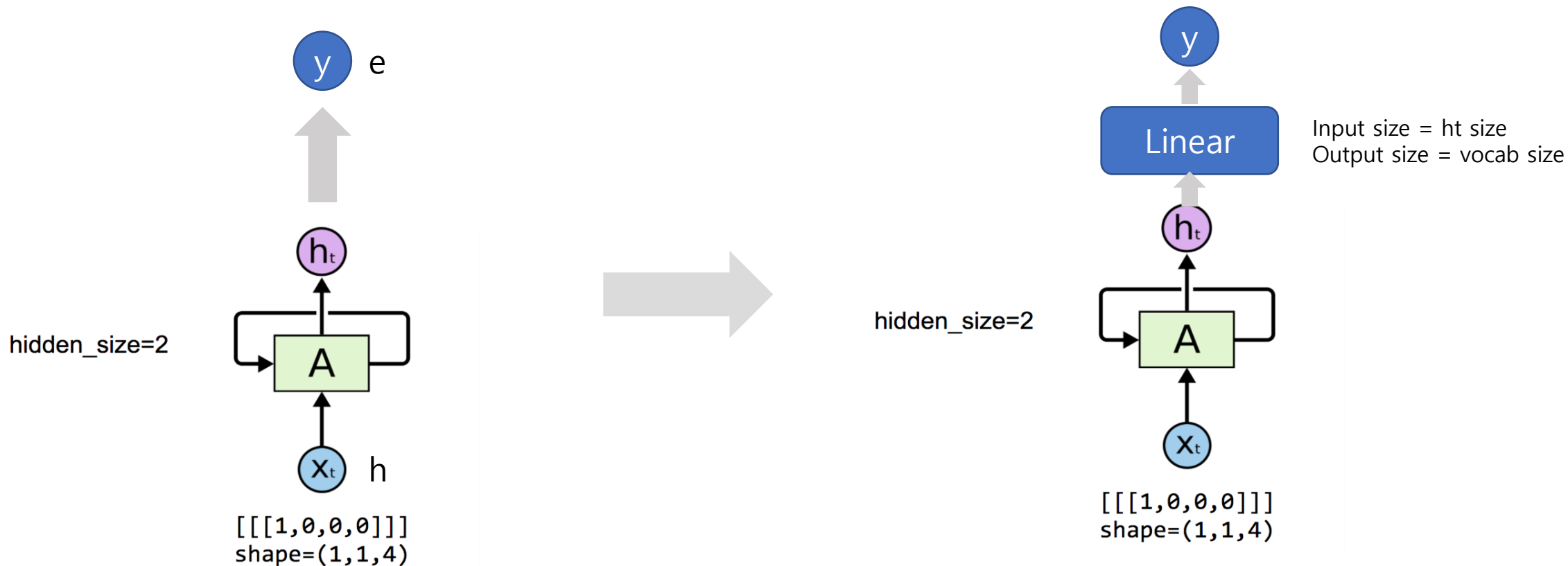
Character Prediction

" seoul is the capital of south kore" 를 input으로 활용하여 RNN의 출력값이 "seoul is the capital of south korea"가 나오게 하자.
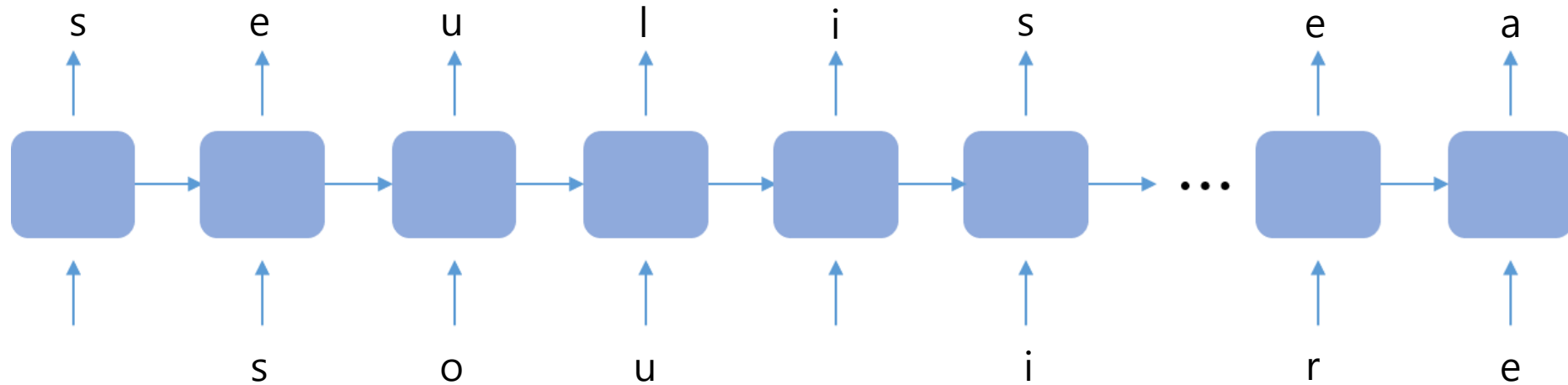
Character Prediction 코드 1/4

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
```

```python
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.cuda.manual_seed_all(0)
```

```python
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

Character Prediction 코드 2/4

```
[4]  sentence = " seoul is the capital of south korea"
     char_set = list(set(sentence))
     char_dic = {c : i for i, c in enumerate(char_set)}
```

문장에 들어가는 character set 생성
character vocab 생성
(dictionary로 key : character val : vocab에서의 index)

```
[6]  vocab_sz = len(char_dic)
     hidden_sz = len(char_dic)
     input_sz = len(char_dic)
```

```
[7]  sen_idx = [char_dic[c] for c in sentence]
     x_idx = sen_idx[:-1]
     x_one_hot = [[np.eye(vocab_sz)[x] for x in x_idx]]
     y_data = [sen_idx[1:]]
```

Sentence의 character index로 변경
input : " seoul is the capital of south kore" 의 index
input을 one hot vector로 변경
target : "seoul is the capital of south korea"

```
[8]  x_train = torch.FloatTensor(x_one_hot)
     y_train = torch.LongTensor(y_data)
```

Character Prediction 코드 3/4

```python
class Rnn(nn.Module):
    def __init__(self,input_size, hidden_size, vocab_size):
        super(Rnn, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size

        self.rnn = nn.RNN(input_size=self.input_size, hidden_size=self.hidden_size, batch_first=True)
        self.linear = nn.Linear(self.hidden_size, self.vocab_size)

    def forward(self, x):
        outputs, _ = self.rnn(x)
        x = self.linear(outputs)

        return x

model = Rnn(input_size=input_sz, hidden_size=hidden_sz, vocab_size=vocab_sz).to(device)
```

Character Prediction 코드 4/4

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
[10] epochs = 500

     for epoch in range(epochs):
         model.train()

         outputs = model(x_train.to(device))  # forward propagation
         loss = criterion(outputs.view(-1, hidden_sz), y_train.view(-1).to(device))

         optimizer.zero_grad()
         loss.backward() # backward propagation
         optimizer.step() # update parameters

         result = outputs.data.numpy().argmax(axis=2)
         result_str = ''.join([char_set[idx] for idx in np.squeeze(result)])

         if epoch % 50 == 0 or epoch == epochs-1:
           print('loss : {} prediction : {}'.format(loss, result_str))
```

Crossentropy로 정답 character 확률 maximize

```
loss : 2.809816837310791 prediction : aaaaaaaaaaaaaaaaafaaaaaaaaaafaaaaaf
loss : 2.5464131832122803 prediction : eaefa eseeae eaa aae ef eaua fesfs
loss : 2.2717344760894775 prediction : saout   o  saa t      sout   s
loss : 1.7964991331100464 prediction : saoul  s tae sapitae sa south  srea
loss : 1.3101493120193481 prediction : saoul is tae tapital oa south sorea
loss : 0.9416419863700867 prediction : seoul is the capital cf south sorea
loss : 0.6738078594207764 prediction : seoul is the capital of south korea
loss : 0.4886109530925751 prediction : seoul is the capital of south korea
loss : 0.3612975776195526 prediction : seoul is the capital of south korea
loss : 0.2734255790710449 prediction : seoul is the capital of south korea
loss : 0.21311675012111664 prediction : seoul is the capital of south korea
```
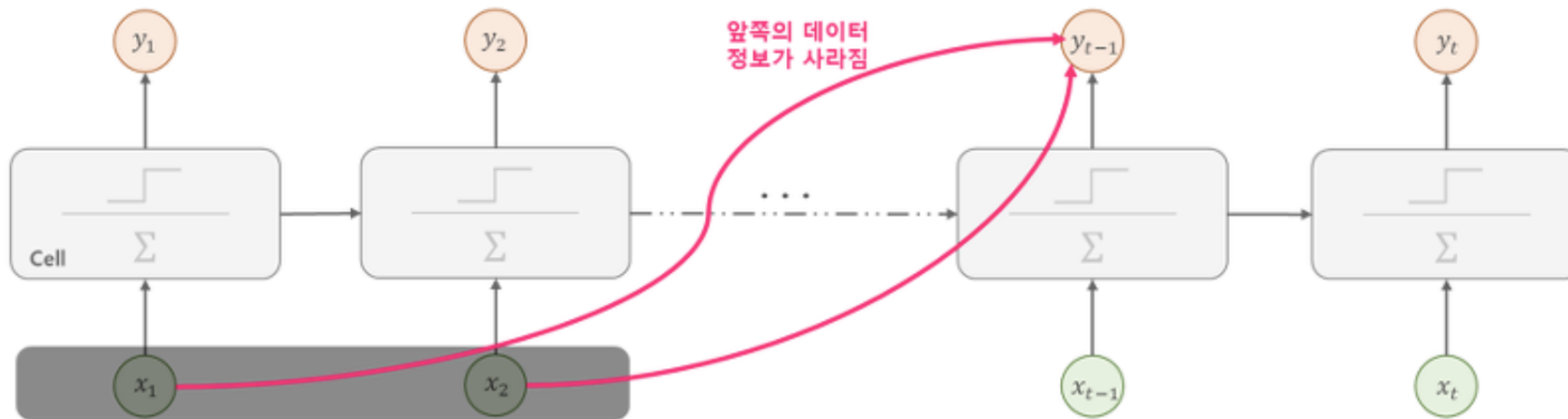
2. LSTM

LSTM 등장배경
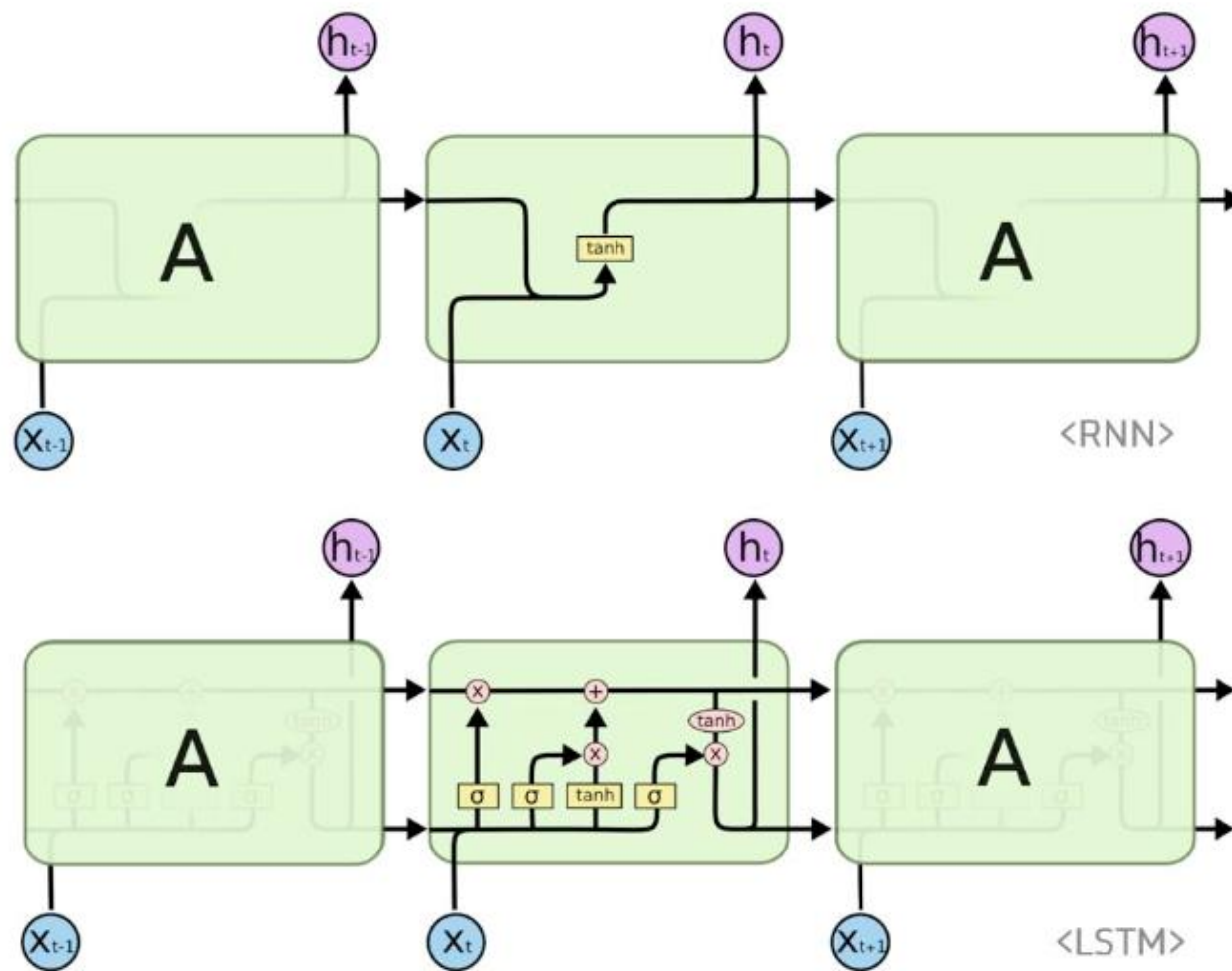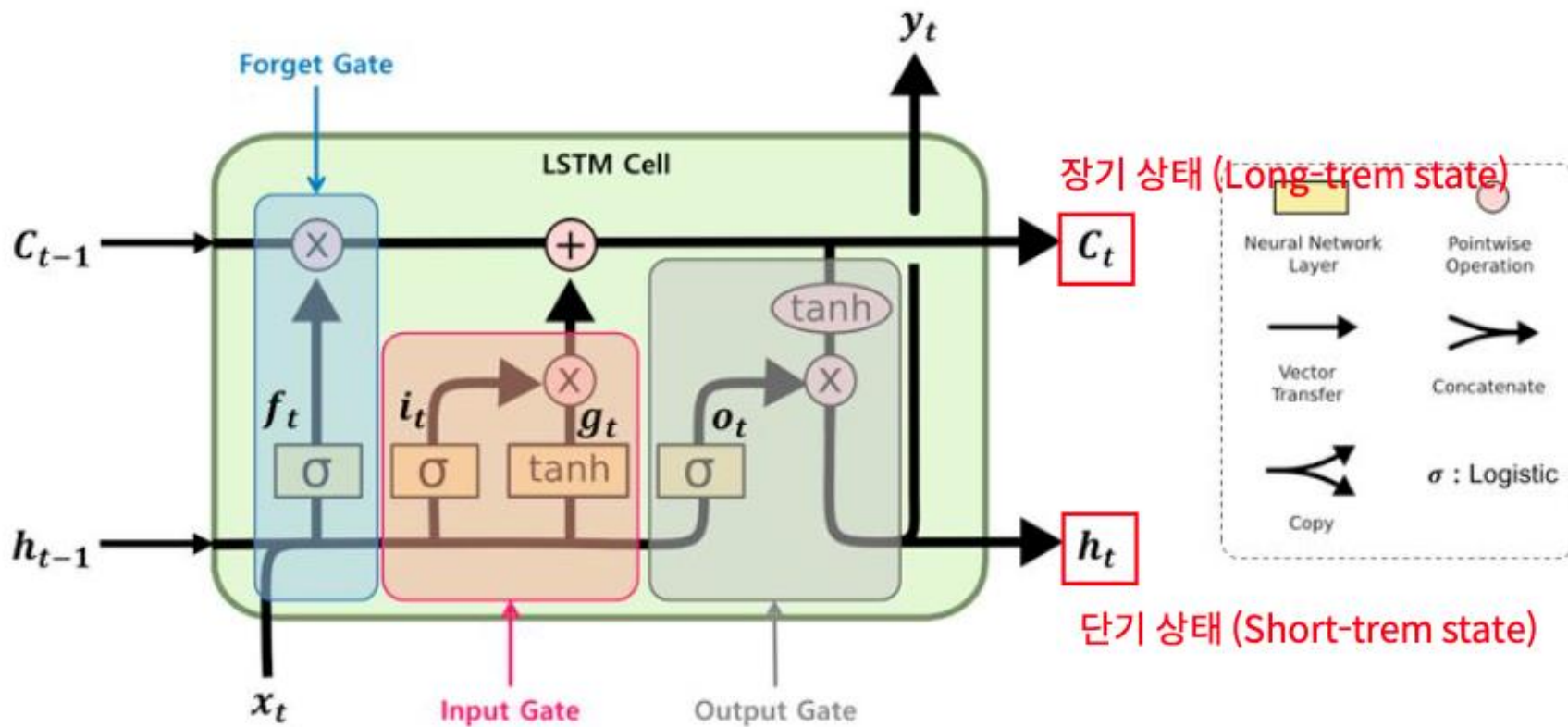
Long-Term Dependency 문제

RNN은 이론적으로 모든 이전 타임 스텝이 영향을 주지만

앞쪽의 타임 스텝(예를 들어 $t = 0, t = 1$)은 타임 스텝이 길어질 수록 영향을 주지 못하는 문제가 발생

LSTM

LSTM

Long Short-Term Memory Cell

LSTM의 핵심 : 네트워크가 장기 상태(c_t)에서 기억할 부분, 삭제할 부분, 읽어 들일 부분을 학습하는 것

LSTM

torch.nn.LSTM

https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

```
lstm = nn.LSTM(input_size=4, hidden_size=2, batch_first=True)
```

Parameters

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights b_ih and b_hh. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

LSTM

torch.nn.LSTM Input
c_t, h_t, x_t 세 개의 input
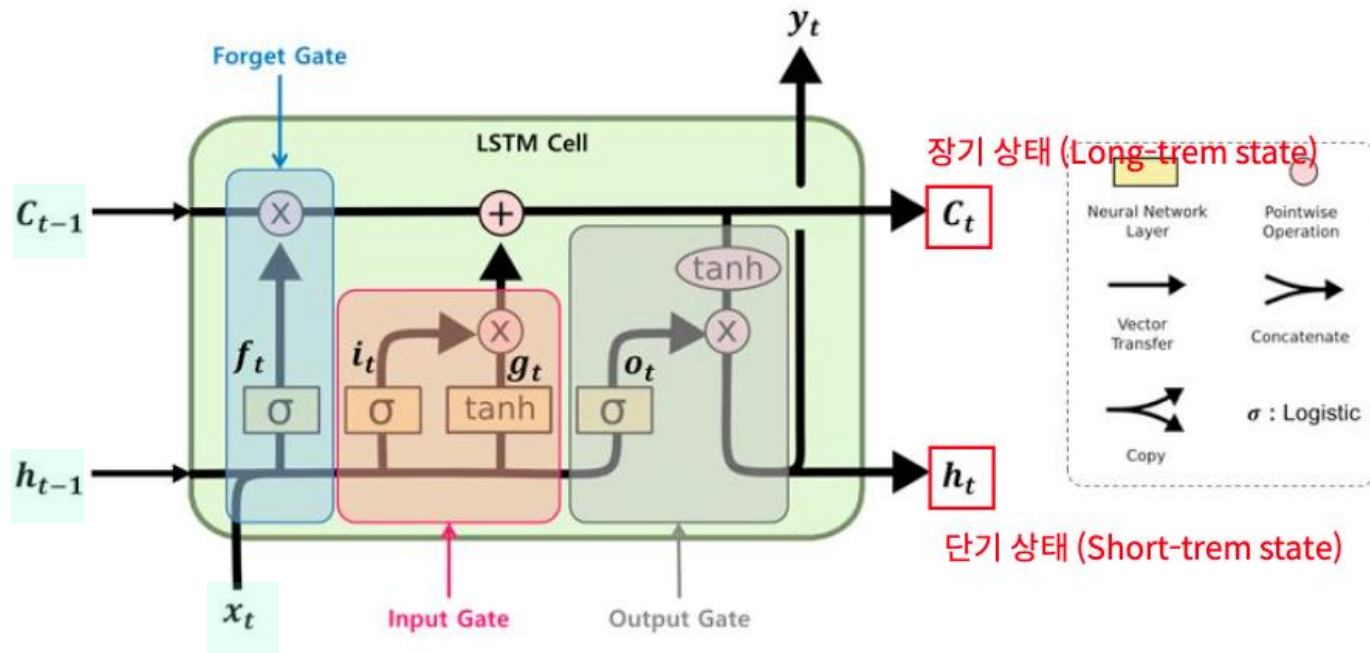
LSTM

torch.nn.LSTM Input

Inputs: input, (h_0, c_0)

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.

- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

D=1 (https://en.wikipedia.org/wiki/Recurrent_neural_network#Bi-directional)
num_layers=1(stacked rnn)

LSTM

torch.nn.LSTM Output

Outputs: output, (h_n, c_n)

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (h_t) from the last layer of the LSTM, for each t. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.

- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the sequence.

- **c_n**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the final cell state for each element in the sequence.

오늘 실습 내용

1. LSTM을 이용해 character prediction

Appendix

Word Prediction

- 단어 단위로 다음 단어를 예측할 수 있다.
  - 단어를 one-hot embedding으로 표현하면 vocab 수가 너무 커져서 dimension 수가 너무 커지므로 비효율적이다.
  - 그리고 one-hot embedding 자체가 단어의 의미를 가지고 있지 않음

→Word2vec와 같은 word embedding 활용

Token Prediction

- GPT
  - DALL E2(text to image)를 만든 OpenAI의 모델
  - Token prediction으로 대규모의 문장으로 pre-training하여 상당히 많은 task에서 좋은 성능을 보이고 있는 model (transformer 기반)
  - https://huggingface.co/gpt2?text=Seoul+is+the+captial+of
  - GPT3 유료, GPT2 huggingface 사용

✍ Text Generation   Examples ▾

Seoul is the captial of

Compute

Computation time on cpu: 1.0284 s

Seoul is the captial of the most famous cities in the world. There are many great temples, places of honor and religious places, and many beautiful ruins. The city's central square is the capital of Seoul, a city in the city of

</> JSON Output   ⊡ Maximize