Recurrent Neural Network

2021.10.15

LG

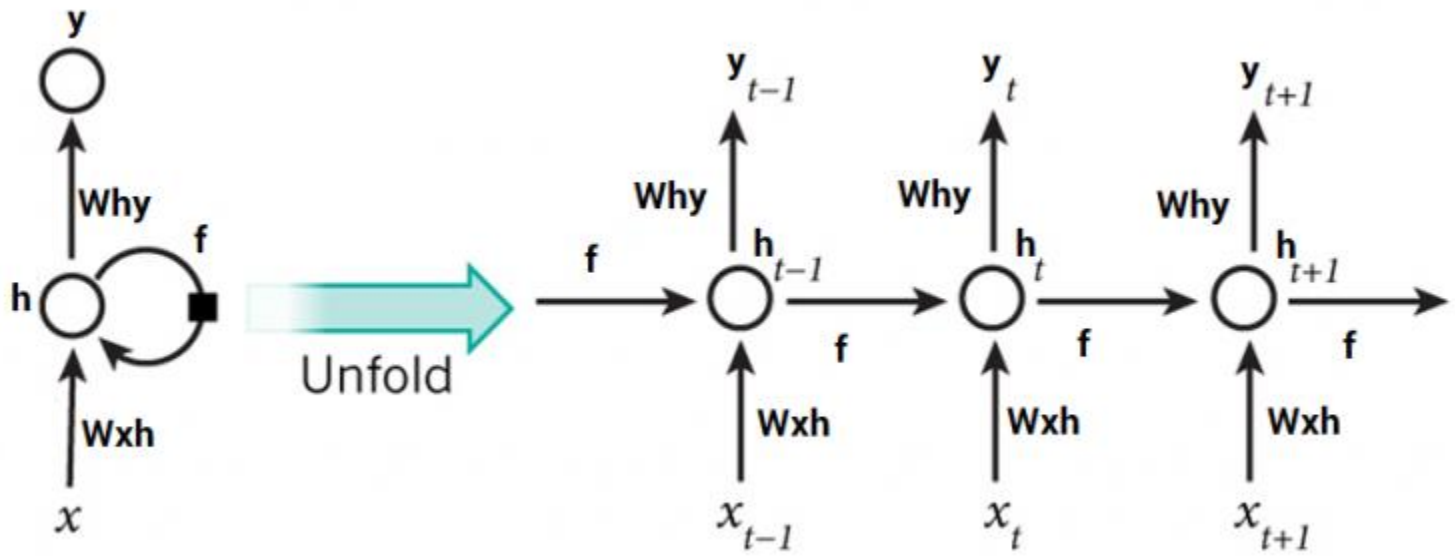
KAIST AI
Graduate School of AI

Contents

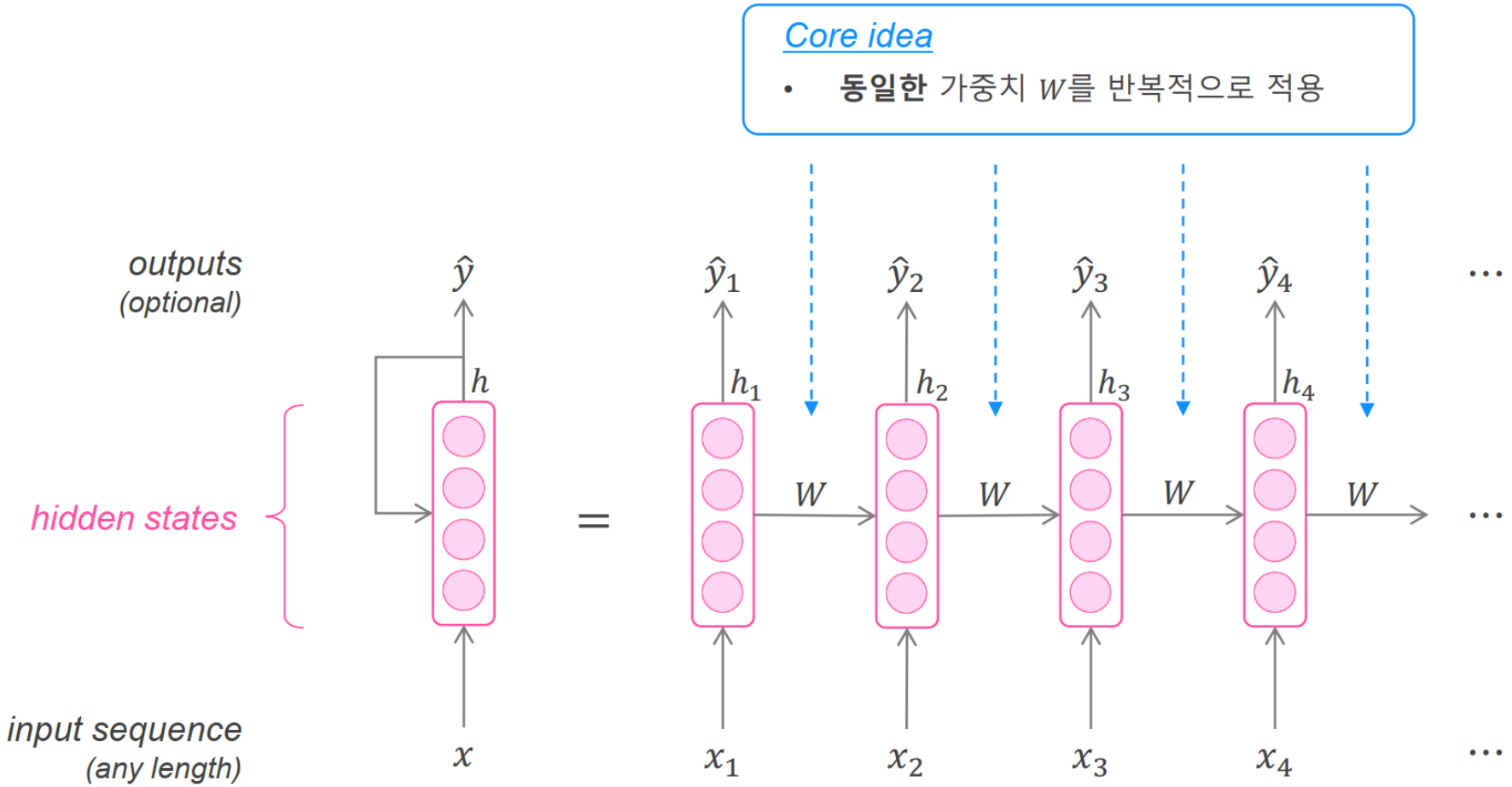
- Recurrent Neural Network (순환 신경망, RNN)
  - RNN을 통한 단어 예측
- Long Short-Term Memory (장단기 메모리, LSTM)
  - LSTM을 통한 주가 예측
- Gated Recurrent Units (GRU)
  - GRU를 통한 MNIST 이미지 분류

Recurrent Neural Network (순환 신경망, RNN)

- What is RNN? Recurrent (반복적인) Neural Network.
- 반복적인 데이터, 즉 순차적인 데이터를 학습하는데 특화되어 발전한 인공신경망.
- 현재 상태를 계산하기 위해 이전 상태가 필요하다.
- 예제를 통해 확인해봅시다.



Recurrent Neural Network (순환 신경망, RNN)



Recurrent Neural Network (순환 신경망, RNN)

- RNN as Language Model

output distribution

$$\hat{y}_t = \text{softmax}(Uh_t + b_2) \in \mathbb{R}^{|V|}$$

hidden states

$$h_t = \sigma\left(W_h h_{(t-1)} + W_e e_t + b_1\right)$$

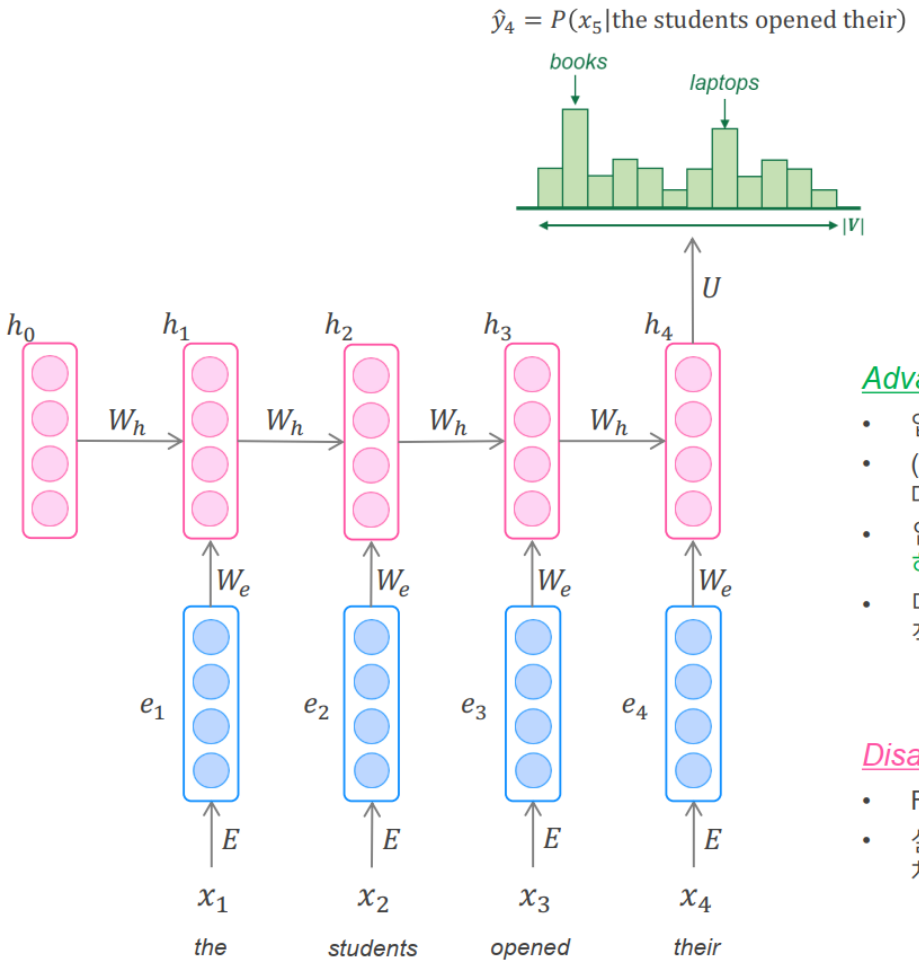
h_0 is the initial hidden state

word embeddings

$$e_t = Ex_t$$

words → one-hot vectors

x_1, x_2, x_3, x_4



Advantages

- 입력(input)의 길이에 제한이 없음
- (이론적으로) 길이가 긴 timestep *t* 에 대해 처리 가능함
- 입력에 따른 모델의 크기가 증가하지 않음
- 매 timestep *t* 에 동일한 가중치를 적용하므로 symmetry 함

Disadvantages

- Recurrent 계산이 느림
- 실제로는 길이가 긴 timestep *t* 에 대해 처리가 안됨

Recurrent Neural Network (순환 신경망, RNN)

- Training a RNN Language Model
  - $x_1, \dots, x_T$의 단어들로 이루어진 시퀀스의 Corpus를 준비한다.
  - $x_1, \dots, x_T$를 차례대로 RNN-LM에 주입하고, 매 step t에 대한 $\hat{y}_t$를 계산한다.
    - 주어진 단어에서부터 시작하여 그 다음 모든 단어들에 대한 확률을 예측
  - Step t에 대한 손실함수 Cross-Entropy를 계산한다. (y_t is one-hot for x_{t+1})

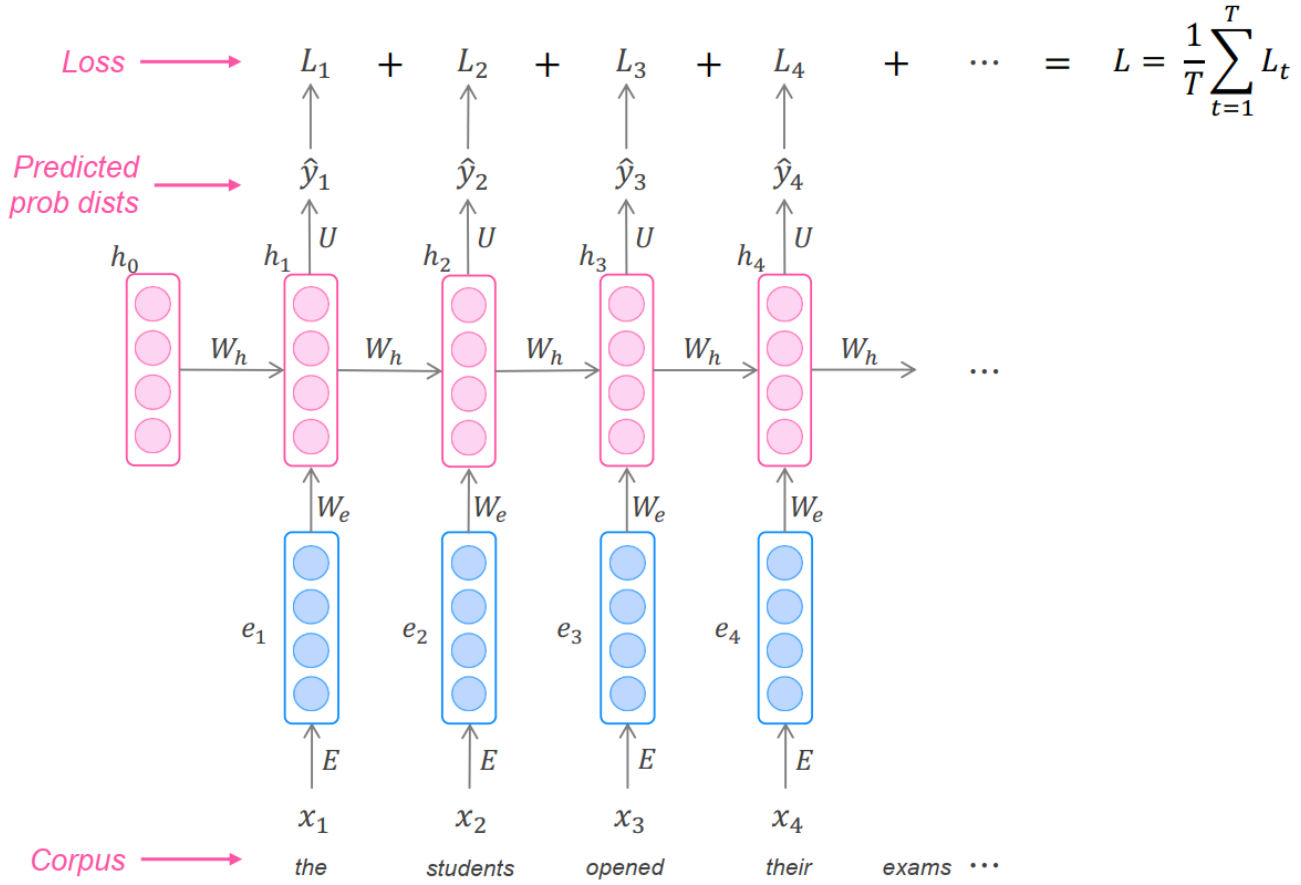
$$L_t = CE(y_t, \hat{y}_t) = -\sum_{w \in |V|} y_{t,w} \times \log(\hat{y}_{t,w}) = -\log(\hat{y}_{t,x_{t+1}})$$

  - 전체 step T에 대해 계산한 손실함수 L_t의 평균을 계산한다.

$$L = \frac{1}{T}\sum_{t=1}^{T} L_t = -\frac{1}{T}\sum_{t=1}^{T}\sum_{w=1}^{|V|} y_{t,w} \times \log(\hat{y}_{t,w}) = -\frac{1}{T}\sum_{t=1}^{T} -\log(\hat{y}_{t,x_{t+1}})$$

Recurrent Neural Network (순환 신경망, RNN)

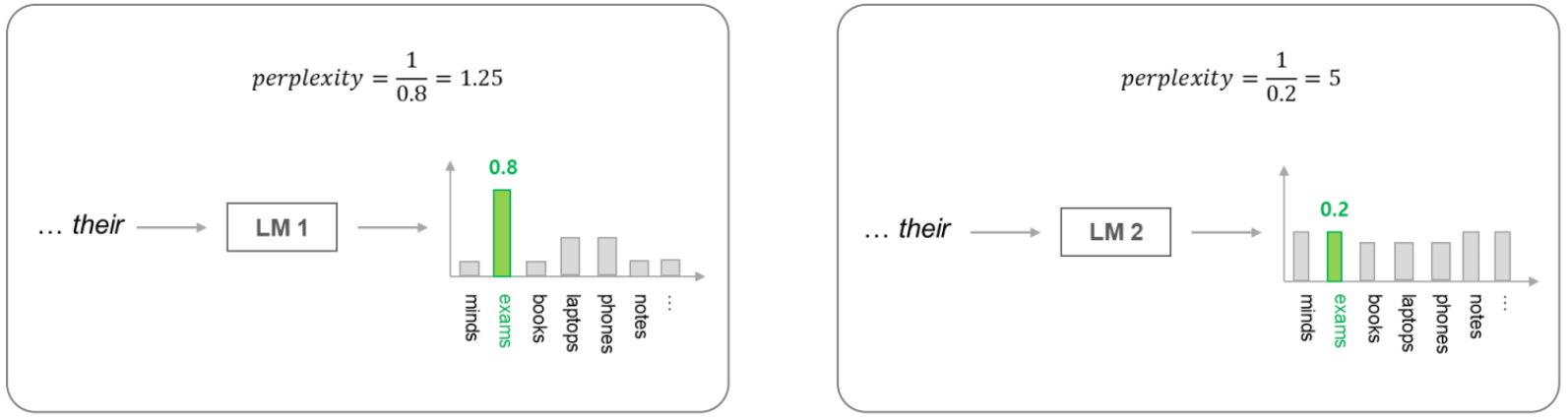
- Training a RNN Language Model



Recurrent Neural Network (순환 신경망, RNN)

- Evaluating Language Model
- Language Model은 주어진 과거 단어 (정보)로부터 다음에 출현할 단어의 확률분포를 출력하는 모델
- Language Model을 평가하는 대표적인 척도는 Perplexity를 사용
- Perplexity는 출현할 단어의 확률에 대한 역수 (inverse)라고 할 수 있음
- Perplexity는 값이 작을수록 좋은 Language Model

as the proctor started the clock, the students opened their _____



Recurrent Neural Network (순환 신경망, RNN)

- Evaluating Language Model
- Language Model은 주어진 과거 단어 (정보)로부터 다음에 출현할 단어의 확률분포를 출력하는 모델
- Language Model을 평가하는 대표적인 척도는 Perplexity를 사용
- Perplexity는 출현할 단어의 확률에 대한 역수 (inverse)라고 할 수 있음
- Perplexity는 값이 작을수록 좋은 Language Model

$$\text{perplexity} = \prod_{t=1}^{T} \left(\frac{1}{P_{\text{LM}}(x_{t+1}|x_t, \ldots, x_1)} \right)^{1/T}$$

$$= \prod_{t=1}^{T} \left(\frac{1}{\hat{y}_{\text{t},x_{t+1}}} \right)^{1/T} = \exp\left(\frac{1}{T} \sum_{t=1}^{T} -\log(\hat{y}_{\text{t},x_{t+1}}) \right) = e^{L}$$

Recurrent Neural Network (순환 신경망, RNN)

- RNN의 장단점

- 순환 신경망의 장점
- 이전의 처리된 데이터를 반영하여 현재의 데이터를 처리하므로, 같은 단어라도 문장에 놓인 위치에 따라 다르게 처리할 수 있다.
- 비교적 단순한 구조이기 때문에 신경망을 구현하는데 있어 큰 어려움 없이 구축할 수 있다.

- 순환 신경망의 단점
- 문장에서 낱말의 위치에 따라 다르게 처리할 수는 있지만 해당 낱말들 간의 관계를 파악하기 어렵다.
- 낱말의 등장 빈도 등과 같은 동체적 정보를 활용하기 어렵다.

Recurrent Neural Network (RNN)

- RNN (Recurrent Neural Network)를 위한 API는 torch.nn.RNN(*args, **kwargs) 입니다.
- RNN의 Parameters
- Inputs_size : Input의 사이즈에 해당 하는 수를 입력하면 됩니다.
- Hidden_size : 은닉층의 사이즈에 해당하는 수를 입력하면 됩니다.
- Num_layers: RNN의 은닉층 레이어 개수를 나타냅니다. 기본 값은 1입니다.
- Nonlinearity : 비선형 활성화 함수를 부여합니다. Tanh, ReLU 중 하나를 선택 가능하며, 기본 값은 Tanh입니다.
- Bias: 바이어스 값 활성화 여부를 선택합니다. 기본 값은 True입니다.
- Batch_first : True 일 시, Output 값의 사이즈는 (batch, seq, feature)가 됩니다. 기본 값은 False 입니다.
- Dropout : 드롭아웃 비율을 설정 합니다. 기본 값은 0입니다.
- Bidirectional : True 일 시, 양방향 RNN이 됩니다. 기본 값은 False입니다.

RNN을 통한 단어 예측

- Word processing

WORD PROCESSING

```
sentences = ["i like dog", "i love coffe", "i hate milk", "you like cat", "you love milk", "you hate coffe"]
dtype = torch.float

word_list = list(set(" ".join(sentences).split()))
word_dict = {w: i for i, w in enumerate(word_list)}
number_dict = {i: w for i, w in enumerate(word_list)}
n_class = len(word_dict)
```

TextRNN PARAMETERS

```
batch_size = len(sentences)
n_step = 2 # 학습 하려고 하는 문장의 길이 - 1
n_hidden = 5 # 은닉층 사이즈

def make_batch(sentences):
    input_batch = []
    target_batch = []

    for sen in sentences:
        word = sen.split()
        input = [word_dict[n] for n in word[:-1]]
        target = word_dict[word[-1]]

        input_batch.append(np.eye(n_class)[input])  # One-Hot Encoding
        target_batch.append(target)

    return input_batch, target_batch

input_batch, target_batch = make_batch(sentences)
input_batch = torch.tensor(input_batch, dtype=torch.float32, requires_grad=True)
target_batch = torch.tensor(target_batch, dtype=torch.int64)
```

RNN을 통한 단어 예측

- Define Model

TextRNN

In [4]:
```
class TextRNN(nn.Module):
    def __init__(self):
        super(TextRNN, self).__init__()

        self.rnn = nn.RNN(input_size=n_class, hidden_size=n_hidden)
        self.W = nn.Parameter(torch.randn([n_hidden, n_class]).type(dtype))
        self.b = nn.Parameter(torch.randn([n_class]).type(dtype))
        self.Softmax = nn.Softmax(dim=1)

    def forward(self, hidden, X):
        X = X.transpose(0, 1)
        outputs, hidden = self.rnn(X, hidden)
        outputs = outputs[-1]  # 최종 예측 Hidden Layer
        model = torch.mm(outputs, self.W) + self.b  # 최종 예측 최종 출력 층
        return model
```

- Training

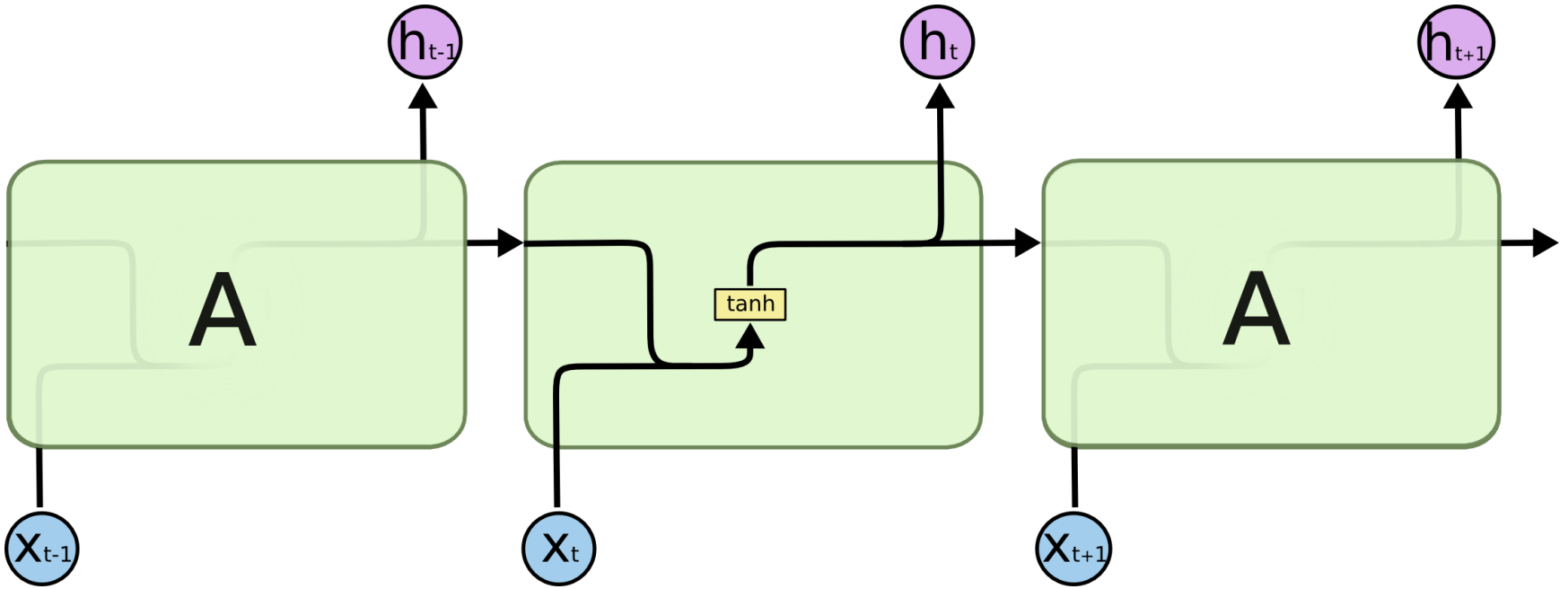
Training

```
model = TextRNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(500):
    hidden = torch.zeros(1, batch_size, n_hidden, requires_grad=True)
    output = model(hidden, input_batch)
    loss = criterion(output, target_batch)

    if (epoch + 1) % 100 == 0:
        print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.6f}'.format(loss))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

print ('OPTIMIZATION FINISHED')
```

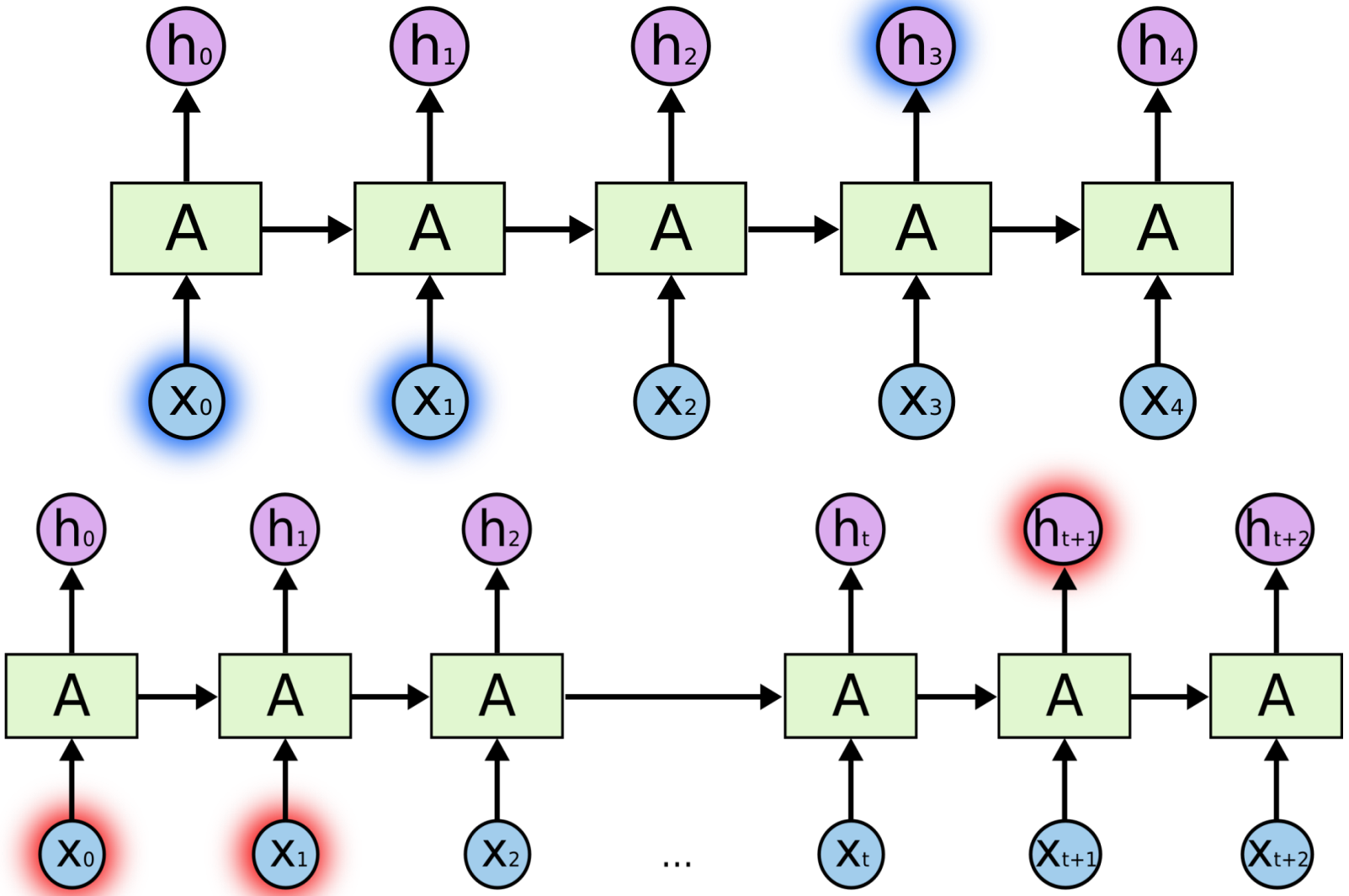
```
Epoch: 0100 cost = 0.361957
Epoch: 0200 cost = 0.058489
Epoch: 0300 cost = 0.026157
Epoch: 0400 cost = 0.015619
Epoch: 0500 cost = 0.010610
OPTIMIZATION FINISHED
```

Long Short-Term Memory (장단기 메모리, LSTM)

- Standard RNN

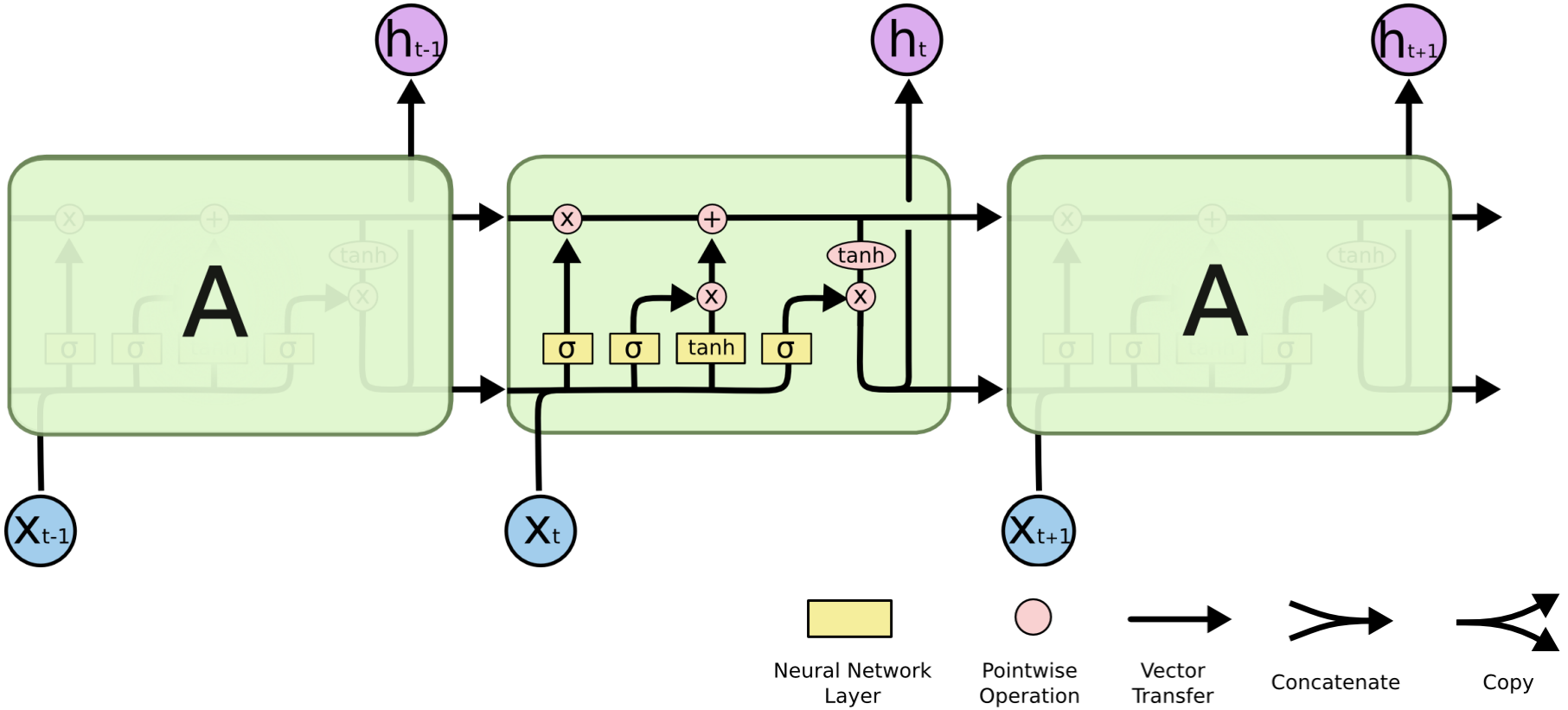


Long Short-Term Memory (장단기 메모리, LSTM)

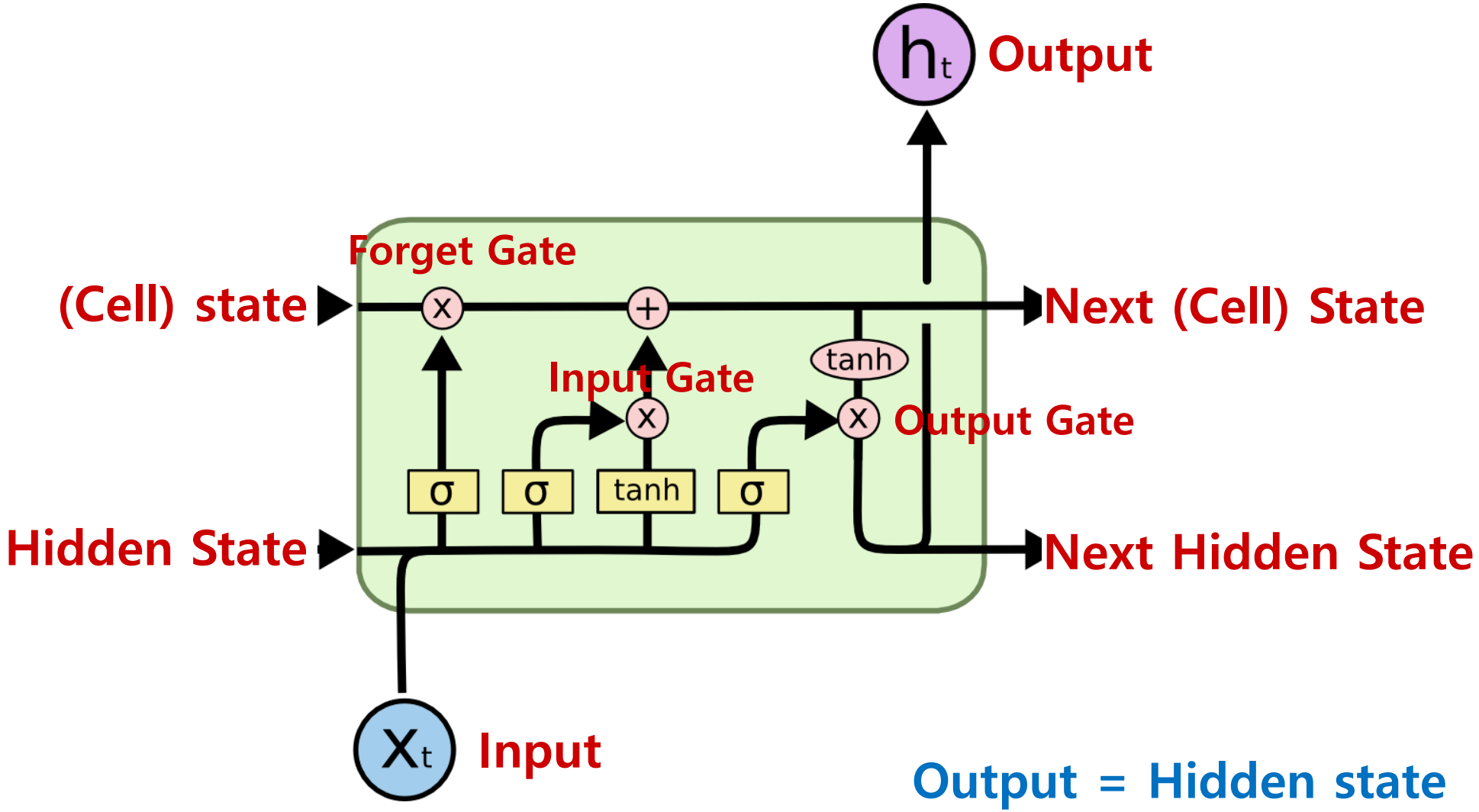


Long Short-Term Memory (장단기 메모리, LSTM)

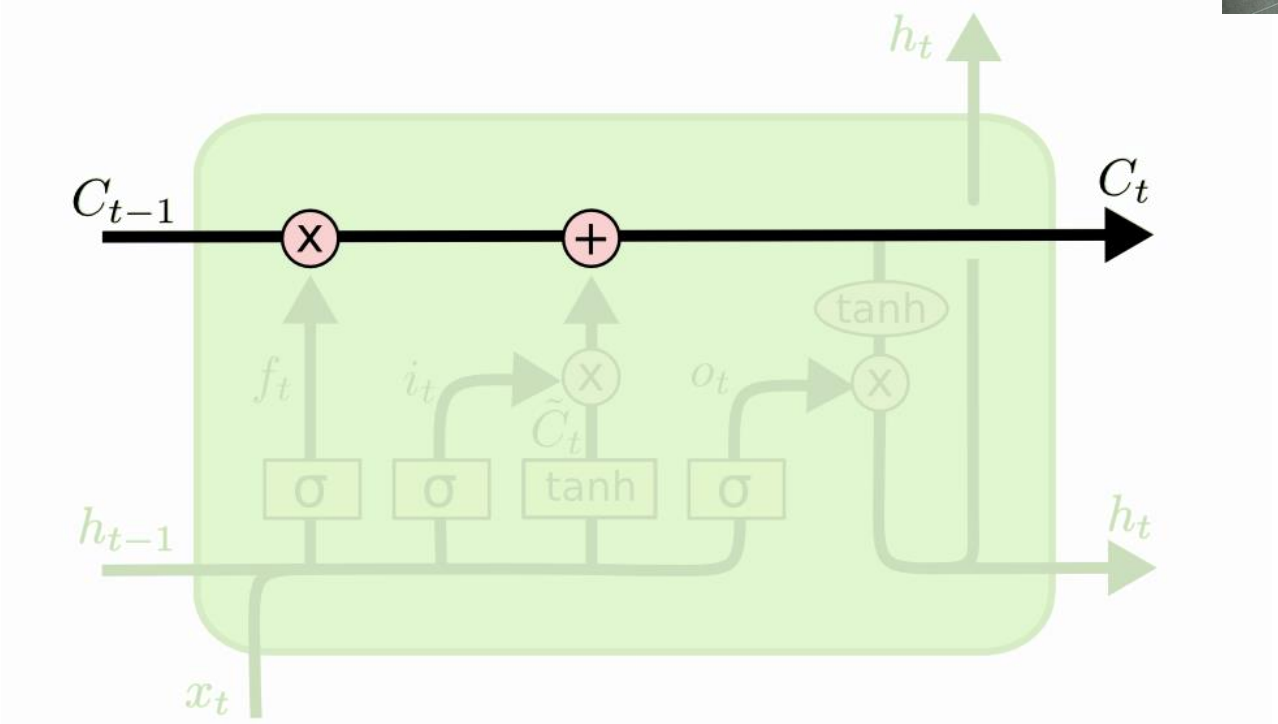
- Long Short-Term Memory (LSTM)
- 단순한 Neural Network 한 층 대신에, 4개의 레이어가 특별한 방식으로 정보를 주고받음.

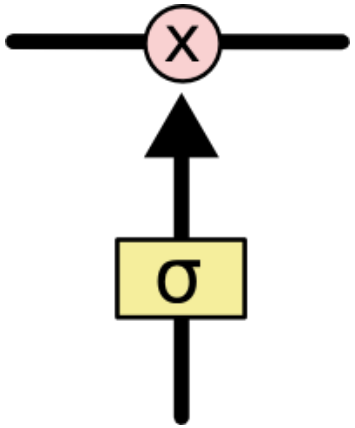


Long Short-Term Memory (장단기 메모리, LSTM)



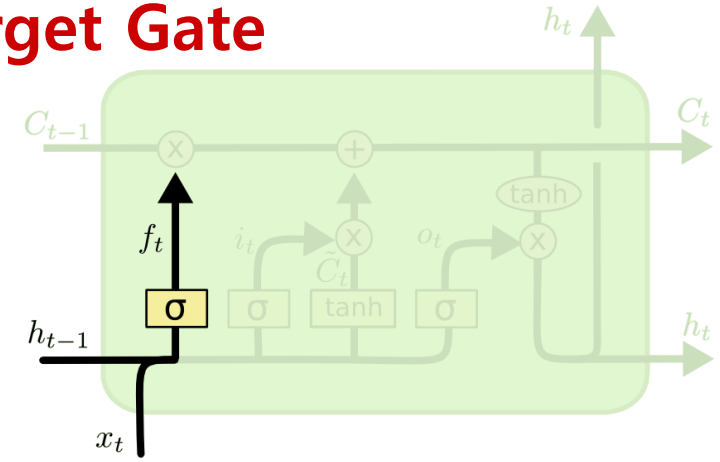
Long Short-Term Memory (장단기 메모리, LSTM)





Long Short-Term Memory (장단기 메모리, LSTM)

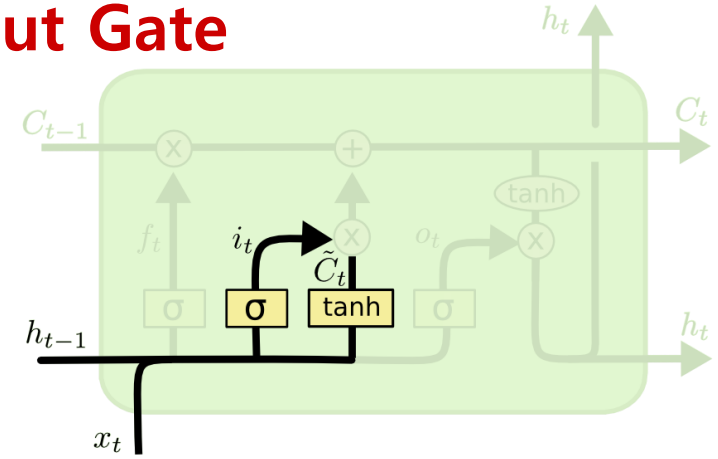
Forget Gate



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Decide what information we're going to **throw away** from the cell state.

Input Gate



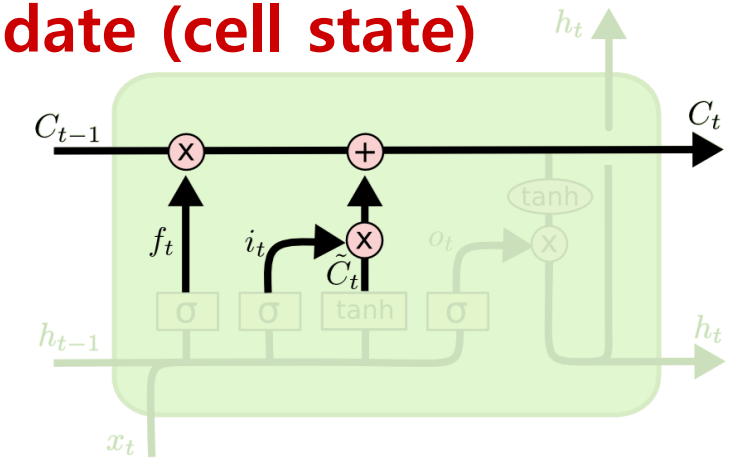
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Decide what new information we're going to **store** in the cell state.

Long Short-Term Memory (장단기 메모리, LSTM)

Update (cell state)

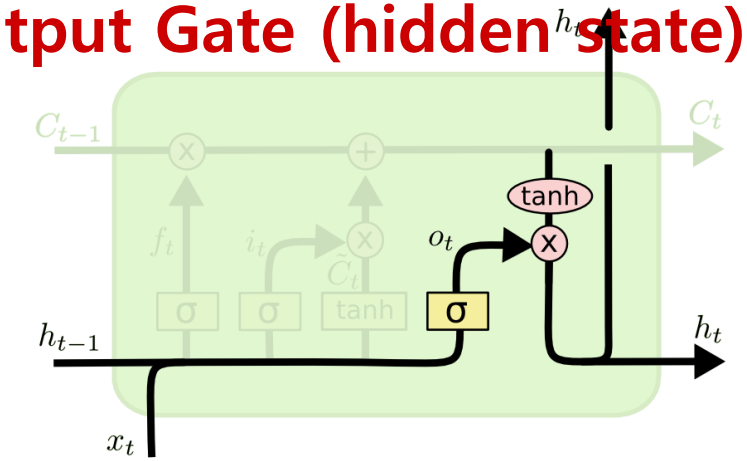


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Update, scaled by how much we decide to update

: input_gate*curr_state + forget_gate*prev_state

Output Gate (hidden state)



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

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

Output based on the updated state

: output_gate*updated_state

Long Short-Term Memory (장단기 메모리, LSTM)

- LSTM (Long Short-Term Memory)를 위한 API는 torch.nn.LSTM(*args, **kwargs) 입니다.
- LSTM은 기울기 폭발, 기울기 소실 등의 문제를 해결하기 위한 기존 RNN을 개선한 구조입니다.

- LSTM의 Parameters
- Inputs_size : Input의 사이즈에 해당 하는 수를 입력하면 됩니다.
- Hidden_size : 은닉층의 사이즈에 해당하는 수를 입력하면 됩니다.
- Num_layers: RNN의 은닉층 레이어 개수를 나타냅니다. 기본 값은 1입니다.
- Nonlinearity : 비선형 활성화 함수를 부여합니다. Tanh, ReLU 중 하나를 선택 가능하며, 기본 값은 Tanh입니다.
- Bias: 바이어스 값 활성화 여부를 선택합니다. 기본 값은 True입니다.
- Batch_first : True 일 시, Output 값의 사이즈는 (batch, seq, feature)가 됩니다. 기본 값은 False 입니다.
- Dropout : 드롭아웃 비율을 설정 합니다. 기본 값은 0입니다.
- Bidirectional : True 일 시, 양방향 RNN이 됩니다. 기본 값은 False입니다.

LSTM을 통한 주가 예측

- Build a dataset

BUILD A DATASET

```
timesteps = seq_length = 7
data_dim = 5
dataX, dataY = [], []

for i in range(0, len(y) - seq_length):
    _x = x[i:i + seq_length]
    _y = y[i + seq_length]
    dataX.append(_x)
    dataY.append(_y)
    if i == 0:
        print(" [INPUT]₩n%s ₩n   =>₩n [OUTPUT] ₩n%s" % (_x, _y))

print ()
print ("TYPE   OF    'dataX' IS [%s]." % (type(dataX)))
print ("LENGTH OF    'dataX' IS [%d]." % (len(dataX)))
print ("LENGTH OF 'dataX[0]' IS [%d]." % (len(dataX[0])))

print ()
print ("TYPE   OF    'dataY' IS [%s]." % (type(dataY)))
print ("LENGTH OF    'dataY' IS [%d]." % (len(dataY)))
print ("LENGTH OF 'dataY[0]' IS [%d]." % (len(dataY[0])))
```

- Split train (70%) and test (30%)

SPLIT TRAIN (70%) AND TEST (30%)

```
def print_np(_name, _x):
    print("TYPE  OF [%s] is [%s]" % (_name, type(_x)))
    print("SHAPE OF [%s] is %s" % (_name, _x.shape,))
```

```
train_size = int(len(dataY) * 0.7)
test_size  = len(dataY) - train_size
trainX = torch.tensor(dataX[0:train_size], dtype=torch.float)
trainY = torch.tensor(dataY[0:train_size], dtype=torch.float)
testX  = torch.tensor(dataX[train_size:len(dataX)], dtype=torch.float)
testY  = torch.tensor(dataY[train_size:len(dataY)], dtype=torch.float)
print_np('trainX', trainX)
print_np('trainY', trainY)
print_np('testX', testX)
print_np('testY', testY)
```

LSTM을 통한 주가 예측

- Define Model

DEFINE MODEL

```
class StockRNN(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        super(StockRNN, self).__init__()
        self.rnn = nn.LSTM(input_size=input_size, hidden_size=n_hidden)
        self.W = nn.Parameter(torch.randn([n_hidden, output_size]).type(torch.float))
        self.b = nn.Parameter(torch.randn([output_size]).type(torch.float))

    def forward(self, hidden_and_cell, X):
        X = X.transpose(0, 1)
        outputs, hidden = self.rnn(X, hidden_and_cell)
        outputs = outputs[-1] # 최종 예측 Hidden Layer
        model = torch.mm(outputs, self.W) + self.b # 최종 예측 출력 층
        return model
```

- Define Hyper-parameters

DEFINE PARAMETERS

```
hidden_dim   = 10
output_dim   = 1
learning_rate = 0.01
iterations   = 500

batch_size = len(trainX)

model = StockRNN(input_size=data_dim, n_hidden=hidden_dim, output_size=output_dim)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

LSTM을 통한 주가 예측

- Training

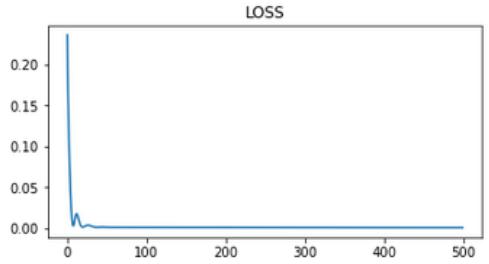
TRAIN

```
losses = []

model.train()
for epoch in range(iterations):
    hidden = torch.zeros(1, batch_size, hidden_dim, requires_grad=True)
    cell = torch.zeros(1, batch_size, hidden_dim, requires_grad=True)
    output = model((hidden, cell), trainX)
    loss = criterion(output, trainY)

    losses.append(loss.item())
    if (epoch + 1) % 100 == 0:
        print('Epoch:', '%04d' % (epoch + 1), 'Loss =', '{:.6f}'.format(loss))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
print ('OPTIMIZATION FINISHED')
```

- Plot Loss

PLOT LOSS

```
plt.figure(figsize=(6,3))
plt.plot(losses)
plt.title('LOSS')
plt.show()
```



LSTM을 통한 주가 예측

- Test

TEST

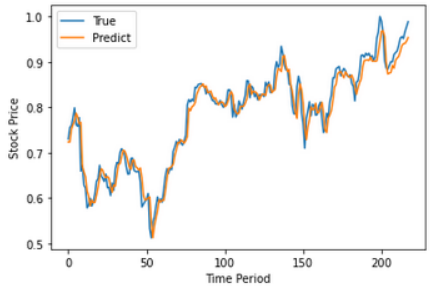
```
: model.eval()

hidden = torch.zeros(1, len(testX), hidden_dim, requires_grad=False)
cell = torch.zeros(1, len(testX), hidden_dim, requires_grad=False)
predict = model((hidden, cell), testX)
rmse = F.mse_loss(predict, testY)

print_np('testX', testX)
print_np('test_predict', predict)
print("RMSE: {}".format(rmse))
```

```
TYPE  OF [testX] is [<class 'torch.Tensor'>]
SHAPE OF [testX] is torch.Size([218, 7, 5])
TYPE  OF [test_predict] is [<class 'torch.Tensor'>]
SHAPE OF [test_predict] is torch.Size([218, 1])
RMSE: 0.0006174131995067
```

- Plot predictions

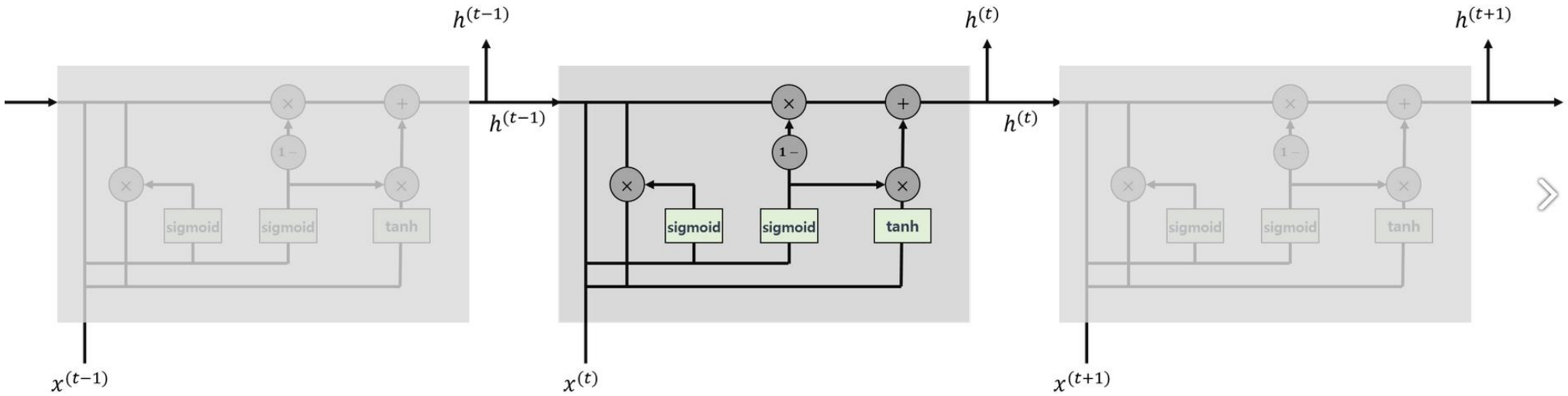
PLOT PREDICTIONS

```
]: plt.plot(testY.data)
plt.plot(predict.data)
plt.xlabel("Time Period")
plt.ylabel("Stock Price")
plt.legend(['True', 'Predict'])
plt.show()
```



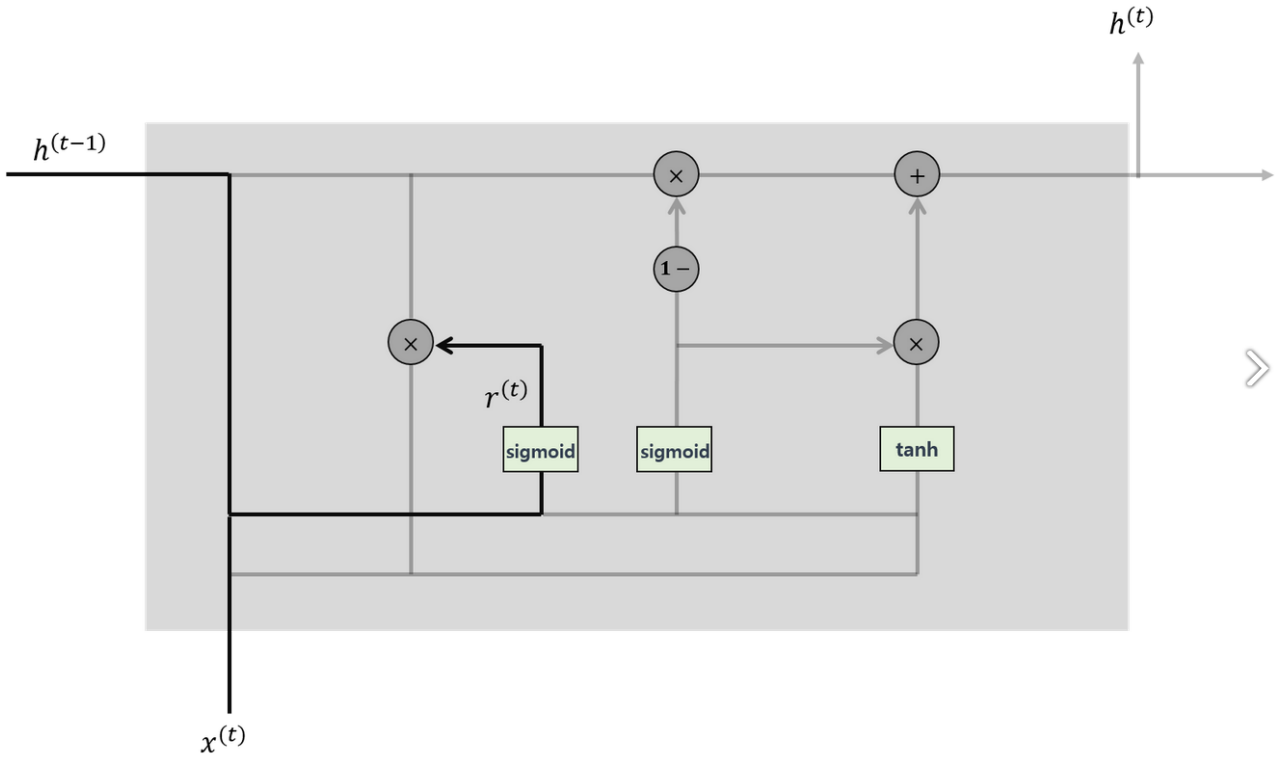
Gated Recurrent Units (GRU)

- GRU는 게이트 메커니즘이 적용된 RNN 프레임워크의 일종으로 LSTM에 영감을 받고, 더 간략한 구조를 가지고 있다.
- 태스크 별로 LSTM이 좋을 때도 있고, GRU가 좋을 때도 있다.
- GRU가 학습할 가중치 (weight)가 적다는 것은 확실한 이점이다.



Gated Recurrent Units (GRU)

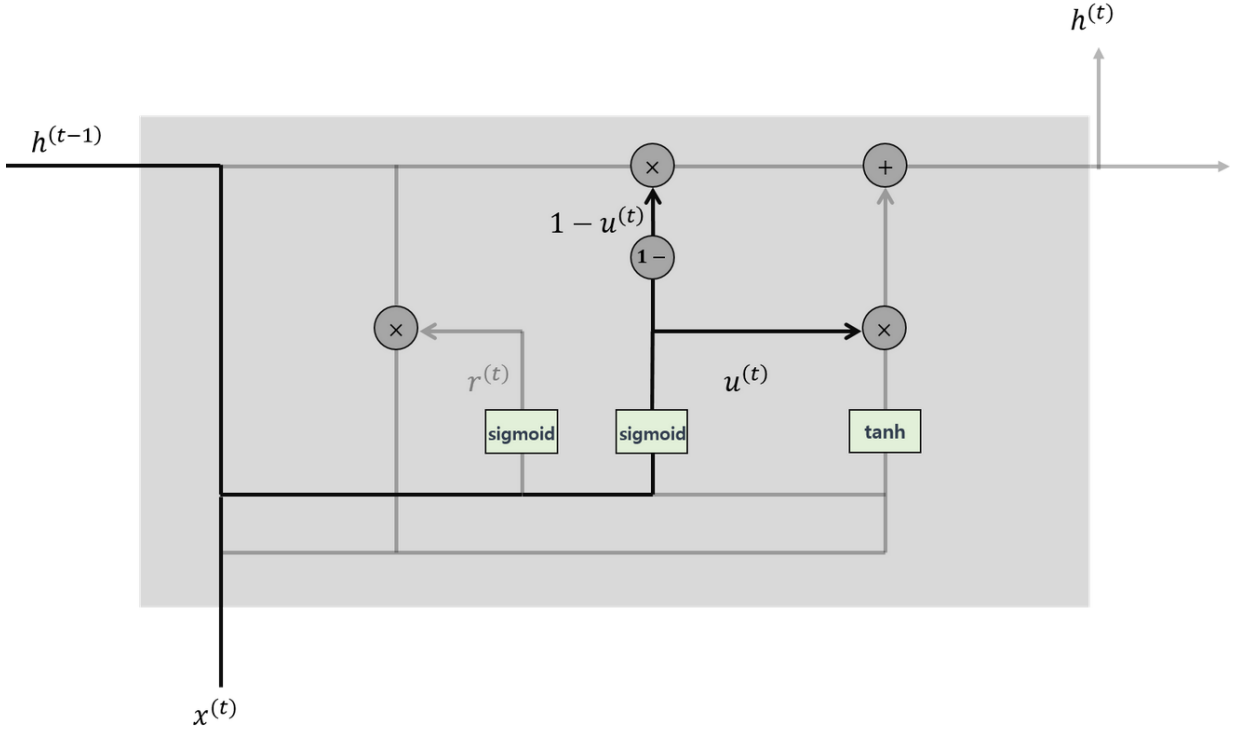
- Reset Gate
- Reset Gate는 과거의 정보를 적당히 Reset 시키는 것이 목적으로 sigmoid 함수를 출력으로 이용해 (0, 1) 값을 이전 은닉층에 곱해준다.



$$r^{(t)} = \sigma\left(W_r h^{(t-1)} + U_r x^{(t)}\right)$$

Gated Recurrent Units (GRU)

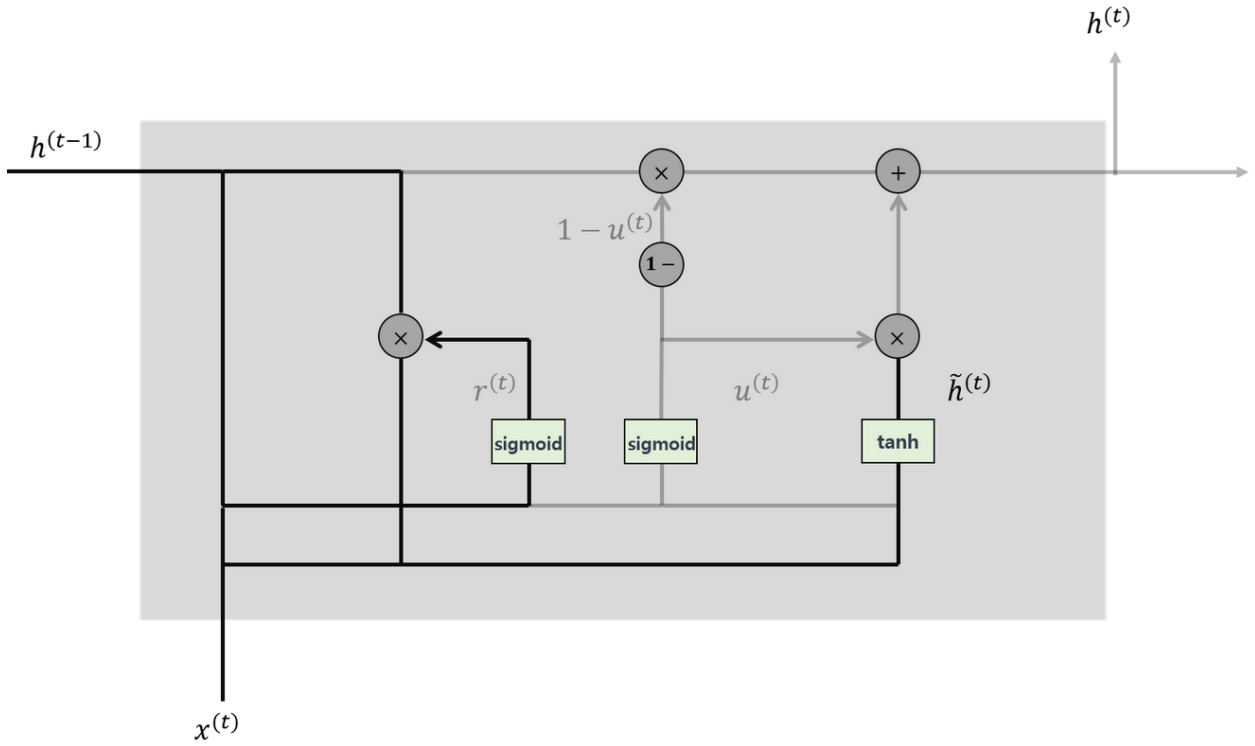
- Update Gate는 LSTM의 forget gate와 input gate를 합친 것과 유사한 역할을 하며, 과거와 현재 정보의 최신화 비율을 결정함.
- Update gate에서는 sigmoid로 출력된 결과 ($u^{(t)}$)는 현시점의 정보의 양을 결정하고, 1에서 뺀 값 ($1 - u^{(t)}$)는 직전 시점의 은닉층에 곱해주며, 각각 LSTM의 input / forget gate와 유사



$$u^{(t)} = \sigma\left(W_u h^{(t-1)} + U_u x^{(t)}\right)$$

Gated Recurrent Units (GRU)

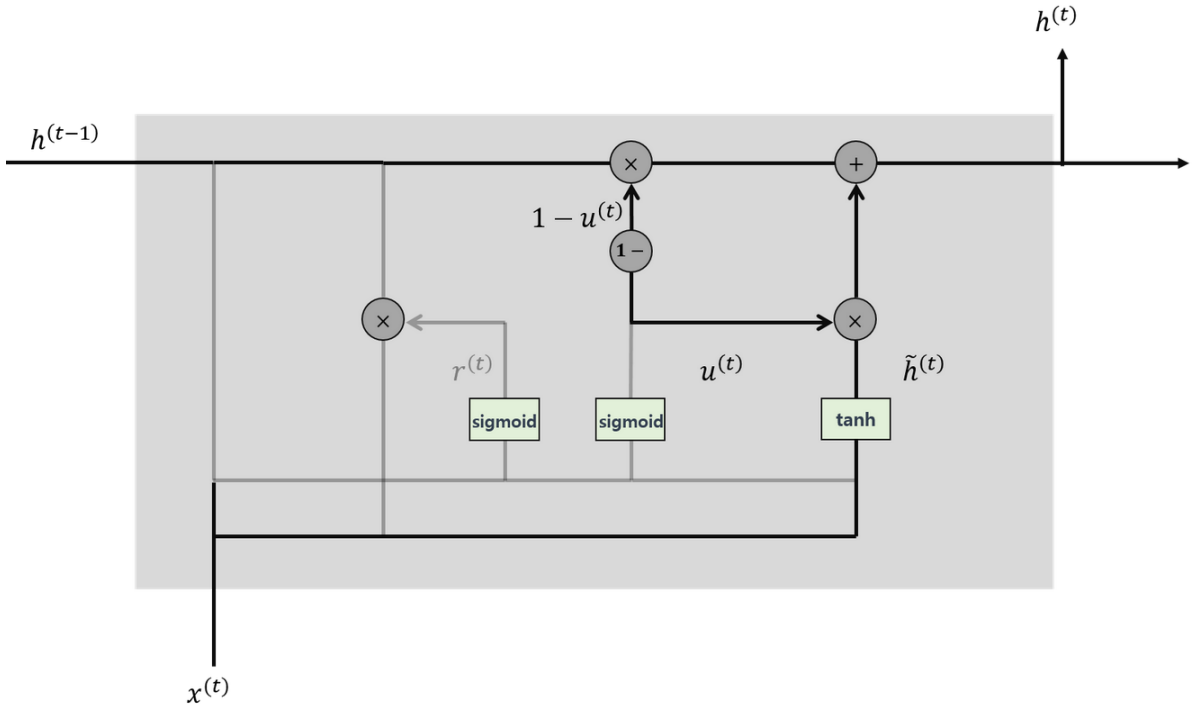
- Candidate는 현 시점의 정보 후보군을 계산하는 단계이다.
- 핵심은 과거 은닉층의 정보를 그대로 이용하지 않고 Reset 게이트의 결과를 곱하여 이용하는 것.



$$\tilde{h}^{(t)} = \tau\left(W h^{(t-1)} * r^{(t)} + U x^{(t)}\right)$$

Gated Recurrent Units (GRU)

- 마지막으로 update gate 결과와 candidate 결과를 결합하여 현시점의 은닉층을 계산하는 단계.
- Sigmoid 함수의 결과는 현시점 결과의 정보의 양을 결정하고, 1 - Sigmoid 함수의 결과는 과거 시점의 정보 양을 결정한다.



$$h^{(t)} = \left(1 - u^{(t)}\right) * h^{(t-1)} + u^{(t)} * \tilde{h}^{(t)}$$

GRU를 통한 MNIST 이미지 분류

- Define GRU Model

In [5]:

```
class MNISTRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classses, use_cuda=False):
        super(MNISTRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.device = torch.device('cuda') if use_cuda else torch.device('cpu')

        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)



    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        # using cell state if you want to use LSTM
#         c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)

        # Passing in the input and hidden state into the model and  obtaining outputs
        out, hidden = self.gru(x, h0)  # out: tensor of shape (batch_size, seq_length, hidden_size)

        #Reshaping the outputs such that it can be fit into the fully connected layer
        out = self.fc(out[:, -1, :])
        return out
```

GRU를 통한 MNIST 이미지 분류

- Define Parameter and Training

DEFINE PARAMETERS

In [6]:
```
sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 2
learning_rate = 0.01

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model = MNISTRNN(input_size, hidden_size, num_layers, num_classes, torch.cuda.is_available()).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

Train

In [7]:
```
model.train()
for epoch in range(num_epochs):
    avg_loss = 0.
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        inputs = inputs.reshape(-1, sequence_length, input_size).to(device)
        targets = targets.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        avg_loss += loss

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    avg_loss /= len(train_loader)
    print('Epoch:', '%04d' % (epoch), 'Loss =', '{:.6f}'.format(avg_loss))

print ('OPTIMIZATION FINISHED')
```

```
Epoch: 0000 Loss = 0.282635
Epoch: 0001 Loss = 0.502443
OPTIMIZATION FINISHED
```