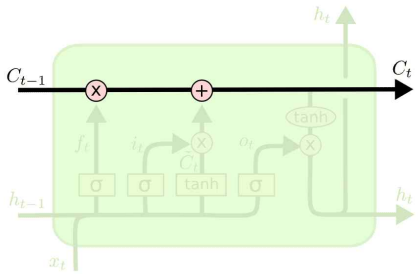
LSTM

- RNN의 주요 모델 중 하나로, 장기의존성(long-term dependency)문제 해결 가능
- 직전 데이터뿐만 아니라, 거시적으로 과거 데이터를 고려하여 미래의 데이터를 예측 가능
- RNN은 한 개의 tanh 레이어를 가지는 매우 간결한 구조를 가진다. LSTM은 또한 RNN과 같은 chain 구조를 가지긴하지만, 단일구조 대신에 매우 특별하게 상호작용하는 **4개의 구조**를 지닌다



- cell-state는 LSTM에서 중요 포인트로 일종의 컨베이어 벨트 역할을 한다, 오직 작은 선형상호작용을 통해 체인 전체에서 작동하며 이는 정보가 쉽게 전달되게 해준다. 덕분에 state가 꽤 오래 경과하더라도 gradient가 비교적 전파가 잘 된다. LSTM은 이 능력을 없애거나 정보를 cell-state에 추가하는 것이 가능하다.
- 각각의 게이트는 정보들이 선택적으로 들어오게 할 수 있는 길이다. 이는 sigmoid 신경망레이어와 pointwise multiplication operation(??)로 구성되어있다
- Backpropagation할 때, '+' 연산은 distributor역할을 해줌. 때문에 gradient를 그대로 전달 >> vanishing gradient의 문제가 생기지 않는다. (Residual Connection / Network)

BPTT(BackPropagation Throught Time)[1)]

- BPTT는 모든 TimeStep마다 처음부터 끝까지 BackPropagation을 하게 되는데, Timestep이 크면 매우 깊은 네트워크가 되며 이러한 네트워크는 Gradient Vanishing or Exploding 문제를 일으키게 된다. 또한, 장기간의 패턴을 학습할 수 없다는 단점이 존재하며, 이를 해결하기 위해 장기간의 메모리

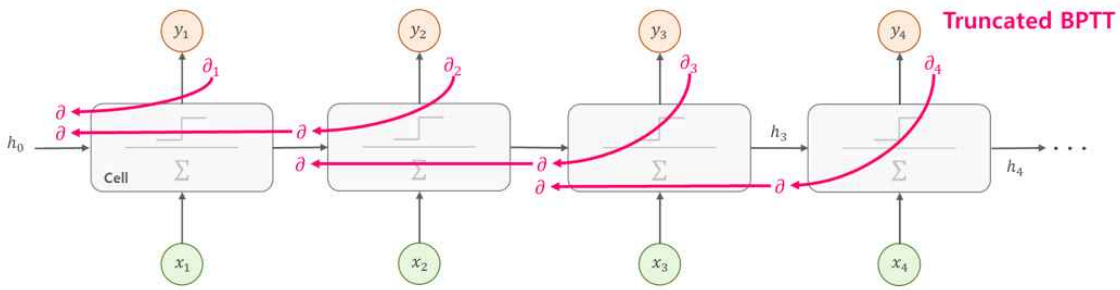
---

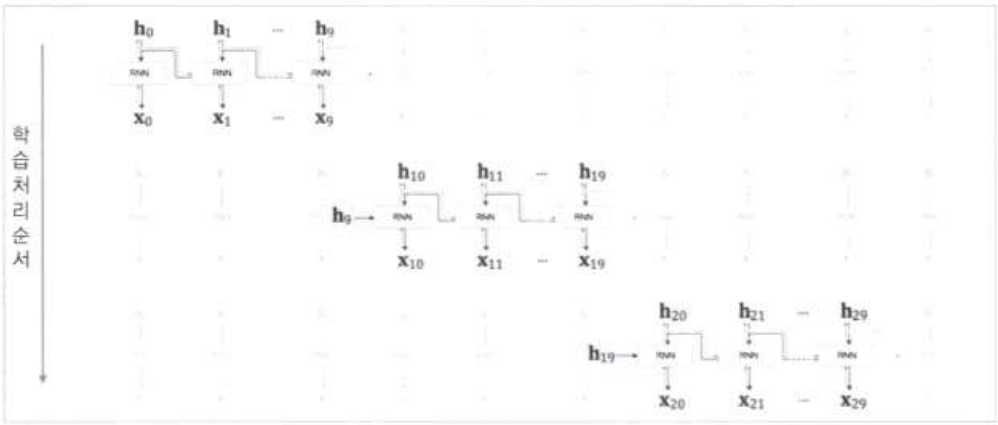
1) Yann LeCun이 발표한 『Efficient BackProp』에선 몇 가지 효율적인 BackPropagation에 대해 제안한다.
① Stochastic Learning
주로 basic backprop에서 선호되는 방법인데, 그 이유를 다음의 세 가지와 같다.
가. 더 큰 데이터셋에서 batch learning보다 보통 더 빠르다.
나. 종종 더 좋은 결과값을 결과로 내놓는다.
다. 모델에서의 함수 변화를 추적하는데 용이하다.
② Shuffling the Examples
1. 같은 class를 가지는 data끼리 속하지않게 training example들을 섞는다.
2. 작은 오류보다 큰 오류가 빈번하게 나타나게 하기위해 input example을 보여준다.
③ Normalizing the Inputs
분산값을 같게 하기 위해 input 값을 scale 해준다. 이와같은 방법을 통해 입력값은 실행하는데 있어서 굉장히 간단해졌다. 다른 방법들은 효과가 있을진 몰라도 그 방법론에 있어선 복잡할 수 있다.
④ Sigmoid
tanh와 같은 standard 함수, 종종 tanh는 계산하는데 있어서 높은 메모리를 차지하기 때문에, sigmoid함수를 사용한다.

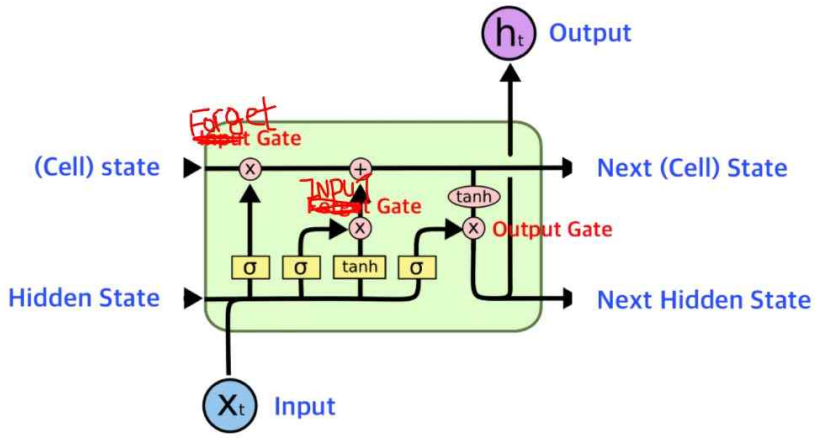
를 가질 수 있는 LSTM셀이 제안되었다.

** BPTT를 보완한 Truncated BPTT

- 기존 RNN은 매우 비효율적인 메모리사용 > BackProp를 일정단위씩 끊어서 계산, 일정단위마다 오차를 다시 계산
- Sequence데이터를 일정한 크기인 T로 잘라서 배치를 나누듯이 한번에 계산하는 크기를 줄인다.
- 한번에 BackPropagation하는 길이가 제한되므로 메모리 사용이 줄어든다.
- 즉, Timestep이 T 이상 떨어진 입-출력 관계를 학습되지 않는다. Truncated BPTT를 사용할 시, 반드시 영향을 주는 데이터 사이의 관계를 침해하지 않게 T로 적절하게 나누어졌는지, 자신이 학습하고자 하는 것이 어느정도의 시간차이까지 자신이 연관성을 봐야 하는지를 염두해두고 학습을 시켜야한다. 만약, 연관성이 있는 데이터의 주기(데이터간의 시점 차이)가 크고 Gradient가 끊기지 않고 연결되어 업데이트가 이루어져야 한다면, 최대한 Batch_size를 낮추고, 최대한 Memory가 큰 GPU를 사용해 긴 길이를 학습해 주는 방법을 사용해야한다.
- 길이 T로 쪼개진 Truncation 사이에는 Gradient BackPropagation이 이루어지지 않는다. 즉, Timestep이 T 이상 떨어진 입-출력 관계는 학습되지 않는다.







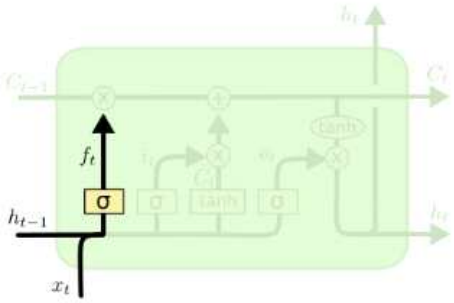
- 총 6개의 파라미터, 4개의 게이트로 구성

1) Input
   - X_t
2) (Cell) State
   - 회전목마 같은 구조로 인해 오차가 사라지지않고, 전체 체인을 관통
   - 정보를 여닫는 역할 (x)
3) Hidden State
   - 이전출력(previous output)값을 가져옴
4) Gates(Forget gate, Input gate, Output gate)
   - 3개의 게이트는 어느 시점에서 정보를 버리거나 유지하여 선택적으로 흘러갈 수 있게(long term & short term을 잘 고려하는) 하기 위함이다

Input Weight Conflict과 Output Weight Conflict

- 자신이 발화해야 할 신호가 전파되어왔을 때는 웨이트를 크게 해서 활성화해야하지만, 관계가 없는 신호가 전파됐을때는 웨이트를 작게해서 비활성인 채로 있어야한다.
- 시계열 데이터를 입력에서 받을 경우와 비교하자면, 이것은 시간의존성이 있는 신호를 받았을 때 웨이트를 크게 하고, 의존성이 없는 신호를 받았을때는 웨이트를 작게하는 것이다.
- 그러나 뉴런이 동일한 웨이트로 연결돼있다면 두 가지 경우에 서로 상쇄하는 형태의 웨이트 변경이 이뤄지므로 특히 장기의존성 학습이 잘 실행되지 않게 된다

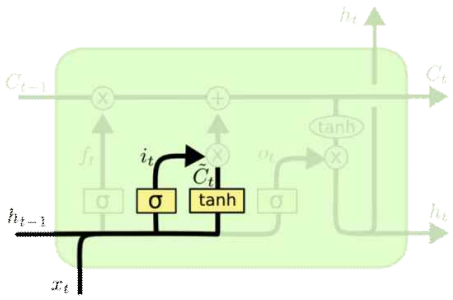
Step 1. **Forget Gate** : "과거 정보를 버릴지 말지 결정하는 과정"



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

- 과거의 정보를 통해 맥락을 고려하는것도 중요하지만, 그 정보가 필요하지 않을 경우에는 과감히 버리는 것도 중요 (Decide what information we're going to throw away from the **cell state**)
- 이전 Input(h_{t-1})과 현재 Input(x_t)을 넣어, cell state로 가는 과거 정보값이 나온다.
- h_{t-1}과 x_t를 받아 Sigmoid를 취해준 값이 바로 forget gate가 내보내는 값(Output)이 된다.
- activation function **Sigmoid**를 사용했으므로 0또는 1값이 나온다.
    * 0일 경우, 이전의 cell state값은 모두 '0'이 되어 미래의 결과에 아무런 영향을 주지 않는다(Drop)
    * 1일 경우, 미래의 예측 결과에 영향을 주도록 이전의 cell state 값(C_{t-1})을 그대로 보내 완전히 유지
- 즉, **Forget Gate는 현재 입력과 이전 출력을 고려하여, cell state의 어떤 값을 버릴지 or 지워버릴지 결정하는 역할을 담당한다.**

Step 2. **Input Gate** : "현재 정보를 저장(기억)할지 결정하는 과정" { $i_t \odot C_t$ }



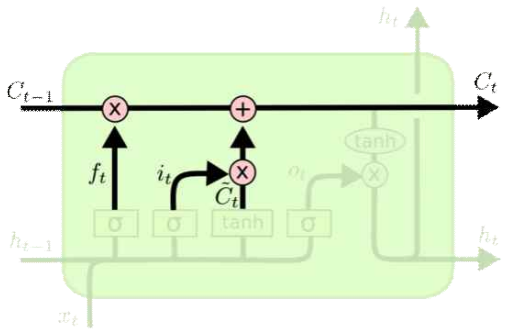
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- 새로운 정보에대해 cell-state에 저장할지 말지 결정하는 과정이면서 이는 두 가지의 파트로 나뉜다.
    1) 'Input gate layer'라고 불리는 sigmoid 레이어는 우리가 업데이트시킬 값에 대해 결정한다
    2) tanh 레이어는 C_t라는 새로운 vector를 생성하는데, 이는 state에 추가된다
- 다음, 이 두가지 정보를 결합하여 state에 업데이트시킨다 >>> $i_t \odot C_t$
- 현재의 cell state값에 얼마나 더할지 말지를 정하는 역할 (tanh는 -1 ~ 1 사이의 값이 나온다)
- Forget Gate(h_{t-1}과 x_t를 받아 시그모이드를 취함)의 값과 같은 입력으로 하이퍼볼릭탄젠트를 취해준 다음 연산한 값이 바로 Input Gate가 내보내는 값
- i_t는 시그모이드 함수를 취했기 때문에 범위는 0 ~ 1
- C_t는 하이퍼볼릭탄젠트 함수를 취했기 때문에 범위는 -1 ~ 1[2)]

> *Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function*
>
> — Page 290, Deep Learning, 2016.

>>> Step1(Forget Gate)과 Step2(Input Gate)의 역할은 이전 cell state 값을 얼마나 버릴지, 지금 입력과 이전 출력으로 얻어진 값을 얼마나 cell state에 반영할지 정하는 역할

Step 3. **Cell state** (Update) : "과거 cell state(C_{t-1})를 새로운 state(C_t)로 업데이트 하는 과정"



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

- 예전 cell-state인 C_{t-1}를 업데이트시켜 새로운 cell-state인 C_t로 만들어주는 과정이다, 이 전의 과정에서 값들을 모두 지정시켜놨으므로, 적용만 시켜주면된다.
- 이전 step에서 결정한 것들을 잊어버리고 예전 state인 C_{t-1}과 f_t를 곱해, i_t * C_t를 더해준다. 이를 통해, 새로운 값이 나왔을 것이고, 이는 각각의 state에 업데이트될 것이다.
- 해당 스텝은 예전 정보들을 잊어버리고 이전 스텝에서 결정했던 정보들을 추가해주는 과정이라고 보면된다
- Update, scaled by now much we decide to update

- Forget Gate를 통해서 얼마나 버릴지, Input Gate에서 얼마나 더할지를 정했으므로

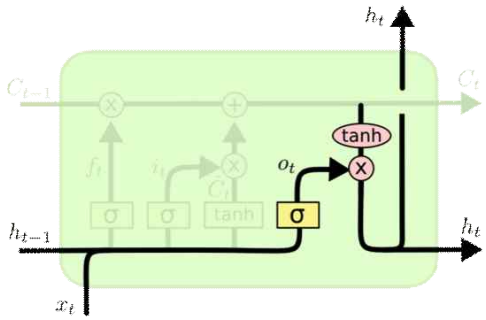
>>> Input Gate * Current State + Forget Gate * Previous State

---

2) ReLU함수보다 Tanh함수를 사용하는 이유

- gating 때문에 그렇다는 의견도 존재한다. 각각의 게이트(input gate, output gate, gorget gate)의 역할은 다른 레이어의 정보를 제한하는하는 역할인데 cell이 너무 많은 정보를 포함해서는 안되기 때문이다. 새로운 것을 받아들이기 위해선 무언갈 잊어야하기때문, 그렇기에 forget gate에선 말 그대로 과거의 값들을 forget(drop)할 필요가 있다. 중요한건, 이 gate들이 sigmoid 함수(0 ~ 1)와 함께 다양한 방법으로 존재한다. gate에서 연산을 통해 값을 보낼지 drop할지 결정하는 것. ReLU함수를 거친 값들은 1보다 큰 값을 가질 수 있고, 더 다양한 값이 존재할 수 있다. 만약 양수의 값을 받아들인다면, 항상 각각의 step에서 너무 큰 값들을 연산하게되면 값이 튀게될 것이다. Divergence
- 초기값 설정(Initialization) 문제에 직면하게 될 것이고, layer에서 너무 큰 weight, bias값을 가지게 될 것이며, 이 값은 너무 큰 gate의 값을 설정하게 만들 것이다. 그러나 만약 초기값을 잘 설정해준다면 위와 같은 일이 발생하지 않을 수 있다.
- 요약하자면, gate에서 1보다 큰 값을 갖게 된다면 time step마다 반복하는동안 너무 큰 양수의 값을 gating(다음 gate로 전달)하게 될것이고, 이 문제를 미연에 방지하고자 ReLU함수보단 Tanh함수를 사용하는 것이다.

Step 4. **Output Gate** (hidden state) : "어떤 출력값을 출력할지 결정하는 과정"



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

- 이 Output값은 우리의 cell-state에 기반한 정보들이 될 것이나, fillter를 적용한 후의 값이 될 것.

① sigmoid 함수를 적용한 Output 값 도출

② 전 과정의 hidden state를 지나쳐온 C_t값을 tanh 함수에 통과 Output 값 도출

③ ①의 o_t와 ②의 $\tanh(C_t)$를 곱함

- Output based on the updated state
- 최종적으로 얻어진 cell-state값을 얼마나 빼낼지 결정하는 역할

>>> Output Gate * Updated State

5) Ouput

- h_t
- output state는 다음 hidden state와 항상 동일함

6) Next (Cell) State

7) Next Hidden State

*** Activation Function으로 Recified Linear Activation Function을 사용하는 이유*

- ReLU함수는 값이 0보다 크면 인풋값을 직접적으로 리턴시켜준다. 만약 0보다 작다면 0값을 리턴. 매우 간단한 구조임. 해당 함수를 통해 네트워크는 선형함수로 더 근접하는 것을 가능하게해준다.
- 하이퍼볼릭탄젠트 함수보단 ReLU함수를 더 많이 사용하는 추세. 뉴럴네트웍이 비선형-의존성을 학습가능. 현재 ReLU함수는 다양한 모델들에서 default값으로 사용되고있는 추세다. 추가적으로, sigmoid와 tanh함수는 많은 레이어에서 vanishing gradient problem문제를 일으키므로 사용되지 않고 있다. ReLU함수는 더 높은 성능과 더 빠른 학습을 통해 해당 문제를 억제시켜준다.
- 레이어가 깊고 큰 네트워크를 사용하게되면 non-linear 함수는 gradient 정보값을 불러올 때, 실패하게될 것이다.
- ReLU함수는 보통 다층퍼셉트론(Multi-layer Perceptron)과 CNN에서 default값으로 사용된다.
- 또한 GPU의 환경에서 더 깊은 신경망을 구성하는것과 같이 하드웨어의 가용성이 상승하면서 학습이 어렵게 되기 때문에 sigmoid와 tanh를 굳이 사용하지 않게되었다.
- ReLU는 선형에 가깝고, 많은 부분을 유지시켜주고, 이는 모델이 gradient-based 방법과 최적화되는데에 좀더 쉽

게 만들어주며 linear-model을 상당히 잘 일반화시켜준다

- 1) 매우 간단한 연산방법
    - sigmoid나 tanh와 달리 매우 간단한 연산을 통해 값을 리턴 / 추가적인 계산이 필요없다
- 2) Representational Sparsity
    - sigmoid나 tanh와 달리 아웃풋을 0이나 0의 근사치로 리턴해줄 수 있다. 이는 레이어를 쌓는 신경망 구조에서 상당히 중요한데, 이는 학습 속도나 모델을 간단화시키는데에 매우 중요한 개념이다.
- 3) Linear Behavior
    - Linear 함수처럼 사용가능하다
    - 일반적으로 신경망구조에서 Linear함수를 사용하면 매우 간단하게 최적화를 시켜줄 수 있다
    - 그러므로, ReLU함수를 사용하면 vanishing gradient 문제를 피할 수 있게 된다
- 4) 깊은 네트워크를 학습가능하다
    - 사전학습 없이 깊은 다층퍼셉트론네트워크를 비선형함수를 activate해 성공적으로 학습시켜줄 수 있다

- 데이터와 분석요건에 따라 많은 변수들이 생길 테지만, ReLU함수는 굉장한 아웃풋에 도달할 수 있으며 전통적으로 LSTM에서 사용되어지고있는 함수
- 다음은 시계열 데이터로 모든 column은 양수값인 데이터이며(Robin이 다루게 될 데이터와 같이), 주 변동성을 예측하려는 hotel cancellation 데이터를 LSTM으로 돌린 값이라고 합니다

```
# Generate LSTM network
model = tf.keras.Sequential()
model.add(LSTM(4, input_shape=(1, previous)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
history=model.fit(X_train, Y_train, validation_split=0.2, epochs=500, batch_size=1, v
```

When the predictions are compared with the test data, the following readings are obtained:

- **Mean Directional Accuracy:** 80%
- **Root Mean Squared Error:** 92
- **Mean Forecast Error:** 29

Now, suppose that a ReLU activation function is invoked:

```
# Generate LSTM network
model = tf.keras.Sequential()
model.add(LSTM(4, activation="relu", input_shape=(1, previous)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
history=model.fit(X_train, Y_train, validation_split=0.2, epochs=500, batch_size=1, v
```

- **Mean Directional Accuracy:** 80%
- **Root Mean Squared Error:** 96.78
- **Mean Forecast Error:** 9.40

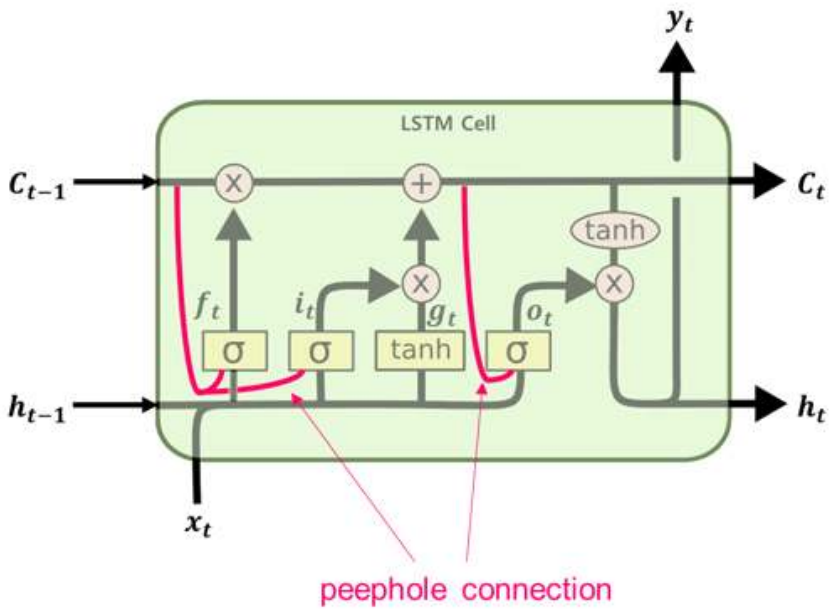
- RMSE 값은 약 4point 차이, MFE값은 현저하게 낮게 측정되었다
- 트렌드값을 잡아내려고하는 sigmoid함수, 경향성 면에선 relu함수가 좀더 우월함

- 데이터분석에서 메모리효율은 절대 꺼지지않는 불과 같은 문제임, 데이터를 handling하고 feature

engineering을 통해 더 높은 효율의 메모리사용과 높은 score값을 내는 것이 데이터분석의 골자.

다양한 LSTM 모델

1) Peephole Connection



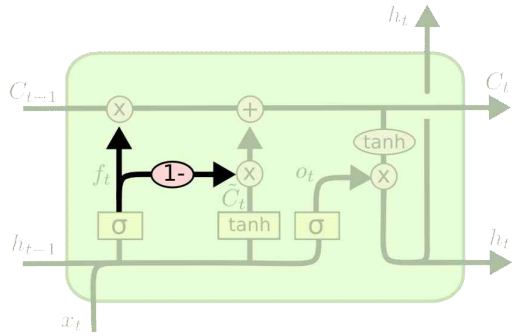
바로 Peephole 연결을 추가한 모델이다. 이는 게이트가 cell-state자체를 입력값으로 받는 방식을 의미한다.

LSTM이 가지는 한계 : Input과 Output의 모든 셀로부터 각각의 gate는 연결들을 받는다. 그러나 여기엔 방향이 없다. 이 모든 것들은 직접적으로 cell output에서 발견되며, 거의 0의 값에 가까운 값을 가진다. 이 필수적인 정보의 결핍 현상은 성능에 있어서 안 좋은 영향을 미친다.
Peephole Connection : 가중합된 "핍홀" 연결을 같은 메모리 block의 gate에 더해주는 것.

일반적인 LSTM에서는 gate controller는 입력 x_t와 이전 단기상태 $h_{(t-1)}$만을 고려한다. 이들에게 장기상태 c_t의 정보도 노출시켜 더 많은 문맥을 감지하게 해보자.

1. c_{t-1}가 forget gate와 input gate의 제어기 f_t와 i_t에 입력으로 추가된다.
2. c_t가 출력게이트의 제어기 o_t의 입력으로 추가된다.

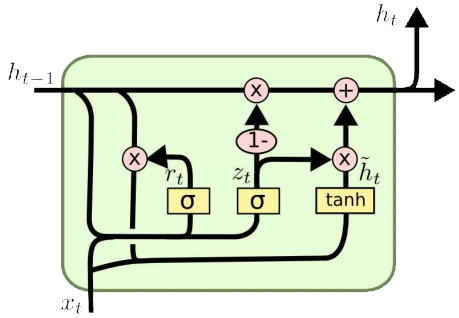
2) Couple with Forget gate and Input gate



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

부분적으로 value값을 drop하는 Forget gate와 새로운정보를 추가하는 Input gate, 두 개의 과정을 따로따로 수행하는 것이 아니라, 이를 동시에 결정하는 방식. 이때 새로운 값이 제공될 때만 이전값을 drop하게된다

3) Gated Recurrent Unit (GRU)



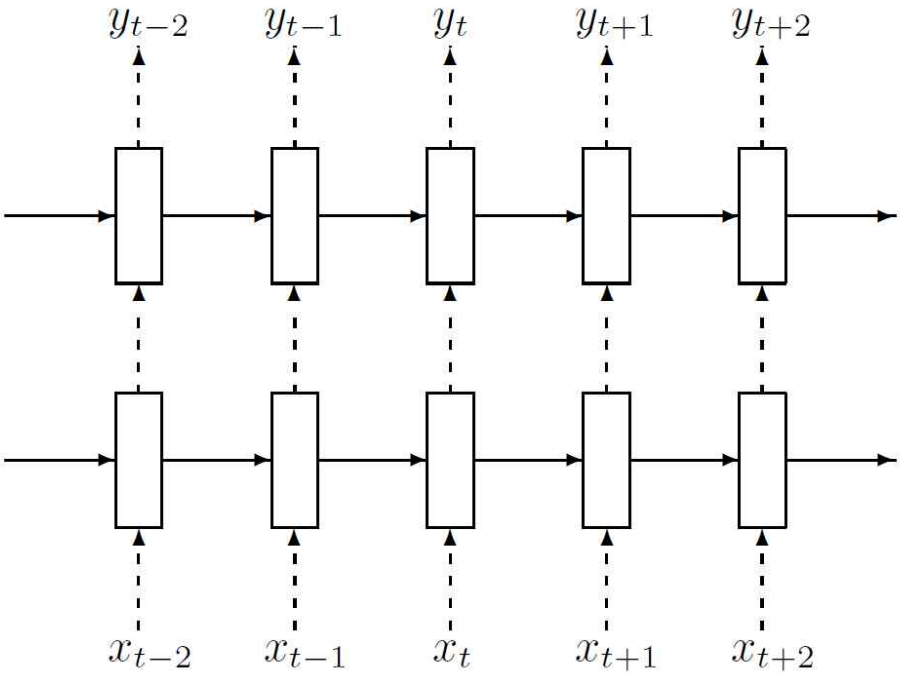
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

이는 기존 LSTM과 꽤나 다른 버전[3]으로 forget gate와 input gate를 하나의 "Update Gate"로 통일. 더불어서, cell-state와 hidden-state를 합쳤고 다른 변화들도 감지된다

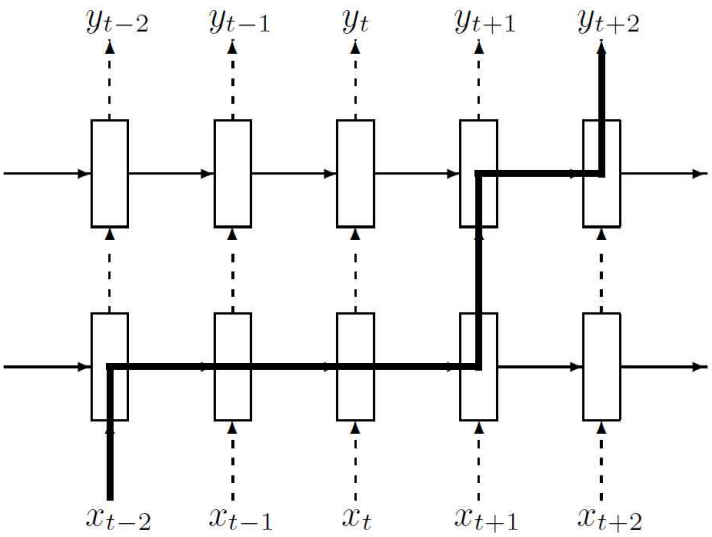
3) Cho, et al. (2014)

LSTM Regularization (Dropout)

Wojciech Zaremba이 발표한『RECURRENT NEURAL NETWORK REGULARIZATION』에서 제시한 방법.
RNN계열의 모델에 dropout을 붙였을 때, 잘 동작하지 않는 이유가 dropout이 지워버리면 안되는 과거 정보까지 전부 지워버리기 때문이라고 주장한다. 때문에 RNN계열 모델에 dropout을 적용하기 위해선 recurrent connection이 아닌 connection들에 대해서만 dropout을 적용해야한다고 주장한다.



과거에서부터 전파되는 정보는 언제나 100% 보존되지만, 아래 layer에서 위 layer로 전달되는 정보는 특정확률로 dropout에 의해 corruption되어 진행된다. 이때, 맨 아래 data layer로부터 맨 위 L번째 layer까지 정보가 전달되는 동안 connection은 정확하게 L+1 번 만큼만 지나게된다.



information flow는 data layer로부터 decision layer까지 정확하게 L+1 번만 dropout의 영향을 받게 된다. 반면 standard dropout을 적용했더라면 information이 더 많은 dropout들에 의해 영향을 받아서 LSTM이 정보를 더 긴 시간 동안 저장할 수 없도록 만들게 되는 것이다. 때문에 recurrent connection에 dropout을 적용하지 않는 것만으로도 LSTM에서 좋은 regularization 효과를 얻을 수 있는 것이다.