Sequence Modeling

Figure 1: Andrey Markov - Image from Wikipedia

Sequential Modeling

우리는 3차원의 공간과 1차원의 시간을 합친 시공간(spacetime)에 살고 있습니다. 그리고 기계학습(machine learning) 또는 인공지능(artificial intelligence)라는 방법을 이용하여 우리의 삶을 개선하고자 합니다. 따라서 많은 문제들은 시공간 상에서 정의되어 있기 마련이고, 이러한 문제들을 풀기 위해 접근함에 있어서 시간의 개념(순서 정보)을 적용하여 문제를 해결하는 것은 중요합니다. 예를 들어 주식시장의 주가 예측이나, 일기예보부터 음성인식이나 번역까지 수많은 문제들이 시간 개념이 큰 영향을 끼칩니다.

우리가 다루고자 하는 자연어(natural language), 즉 텍스트(text)의 경우에도 단어들이 모여(sequence) 문장이 되고, 문장이 모여 문서가 됩니다. 문장 내의 단어들은 앞뒤 위치에 따라 서로에게 영향을 주고, 문서 내의 문장들도 위치에 따라 서로에게 영향을 주고 받습니다. 따라서 우리는 단순히 $y = f(x)$와 같이 시간의 개념이 없이 입력을 넣으면 출력이 나오는 함수의 형태가 아닌, 시간에 따라서 순차적(sequential)으로 입력을 넣고, 입력에 따라 모델의 상태(hidden state)가 변하며, 출력(observation)이 상태에 따라 반환되는 그러한 함수가 필요합니다.

이런 시간 개념 또는 순서 정보를 사용하여 입력을 학습하는 것을 Sequential Modeling이라고 합니다. 신경망(neural network) 뿐만이 아니라 다양한 방법(Hidden Markov Model, Conditional Random Fields 등)을 통해 이런 문제들에 접근 할 수 있으며, 신경망에서는 Recurrent Neural Network(순환신경망, RNN)이라는 아키텍처를 사용하여 효과적으로 문제를 해결할 수 있습니다.

(Vanilla) Recurrent Neural Network

기존 신경망은 정해진 입력 x를 받아 y를 출력해 주는 형태였습니다.

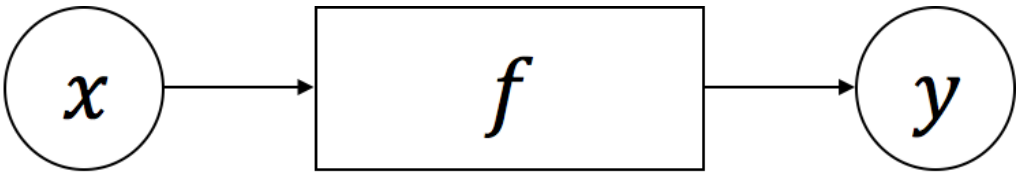

Figure 2: 기존의 뉴럴넷 구조

$$y = f(x)$$

하지만 recurrent neural network (순환신경망, RNN)은 입력 x_t와 직전 자신의 상태(hidden state) h_{t-1}를 참조하여 현재 자신의 상태 h_t를 결정하는 작업을 여러 time-step에 걸쳐 수행 합니다. 각 time-step별 RNN의 상태는 경우에 따라 출력이 되기도 합니다.

$$h_t = f(x_t, h_{t-1})$$

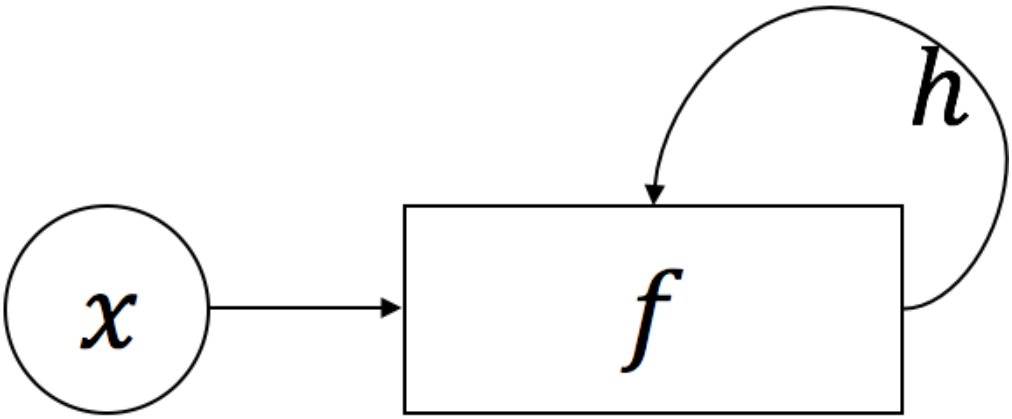

Figure 3: Recursive한 속성이 부여된 뉴럴넷 구조

Feed-forward

기본적인 RNN을 활용한 feed-forward 계산의 흐름은 아래와 같습니다. 아래의 그림은 각 time-step 별로 입력 x_t와 이전 time-step의 h_t가 RNN으로 들어가서 출력으로 h_t를 반환하는 모습입니다. 이렇게 얻어낸 h_t들을 $\hat{y}_t$로 삼아서 정답인 y_t와 비교하여 손실(loss) $\mathcal{L}$을 계산 합니다.

위 그림을 수식으로 표현하면 아래와 같습니다. 함수 f는 x_t와 h_{t-1}을 입력으로 받아서 파라미터 θ를 통해 h_t를 계산 합니다. 이때, 각 입력과 출력 그리고 내부 파라미터의 크기는 다음과 같습니다. - $x_t \in \mathbb{R}^w, h_t \in \mathbb{R}^d, W_{ih} \in \mathbb{R}^{d \times w}, b \in \mathbb{R}^d, W_{hh} \in \mathbb{R}^{d \times d}, b_{hh} \in \mathbb{R}^d$

$$\begin{aligned} \hat{y}_t = h_t &= f(x_t, h_{t-1}; \theta) \\ &= \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \\ &where\ \theta = [W_{ih}; b_{ih}; W_{hh}; b_{hh}]. \end{aligned}$$

위의 수식에서 나타나듯이 RNN에서는 ReLU나 다른 활성함수(activation function)을 사용하기보단 tanh를 주로 사용합니다. 최종적으로 각 time-step별로 y_t를 계산하여 아래의 수식처럼 모든 time-step에 대한 손실(loss) $\mathcal{L}$을 구합니다.

$$\mathcal{L} = \frac{1}{n}\sum_{t=1}^{n} loss(y_t, \hat{y}_t)$$

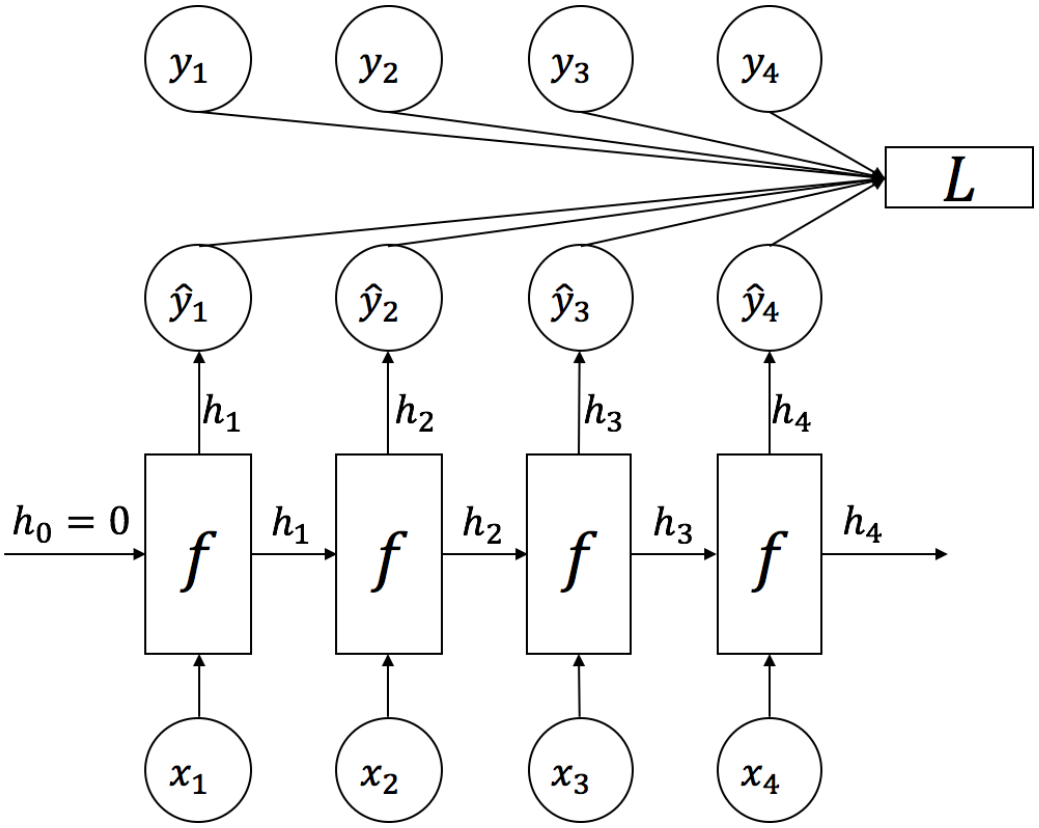

Figure 4: 기본적인 RNN의 feed-forward 형태

Back-propagation Through Time (BPTT)

그럼 이렇게 feed-forward 된 이후에 오류의 back-propagation(역전파)은 어떻게 될까요? 우리는 수식보다 좀 더 개념적으로 접근 해 보도록 하겠습니다.

각 time-step의 RNN에 사용된 파라미터 θ는 모든 시간에 공유되어 사용 되는 것을 기억 해 봅시다. 따라서, 앞서 구한 손실 $\mathcal{L}$에 미분을 통해 back-propagation 하게 되면, 각 time-step 별로 뒤(t가 큰 time-step)로부터 θ의 gradient가 구해지고, 이전 time-step $(t-1)$ θ의 gradient에 더해지게 됩니다. 즉, t가 0에 가까워질수록 RNN 파라미터 θ의 gradient는 각 time-step 별 gradient가 더해져 점점 커지게 됩니다.

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_t \frac{\partial loss(y_t, \hat{y}_t)}{\partial \theta}$$

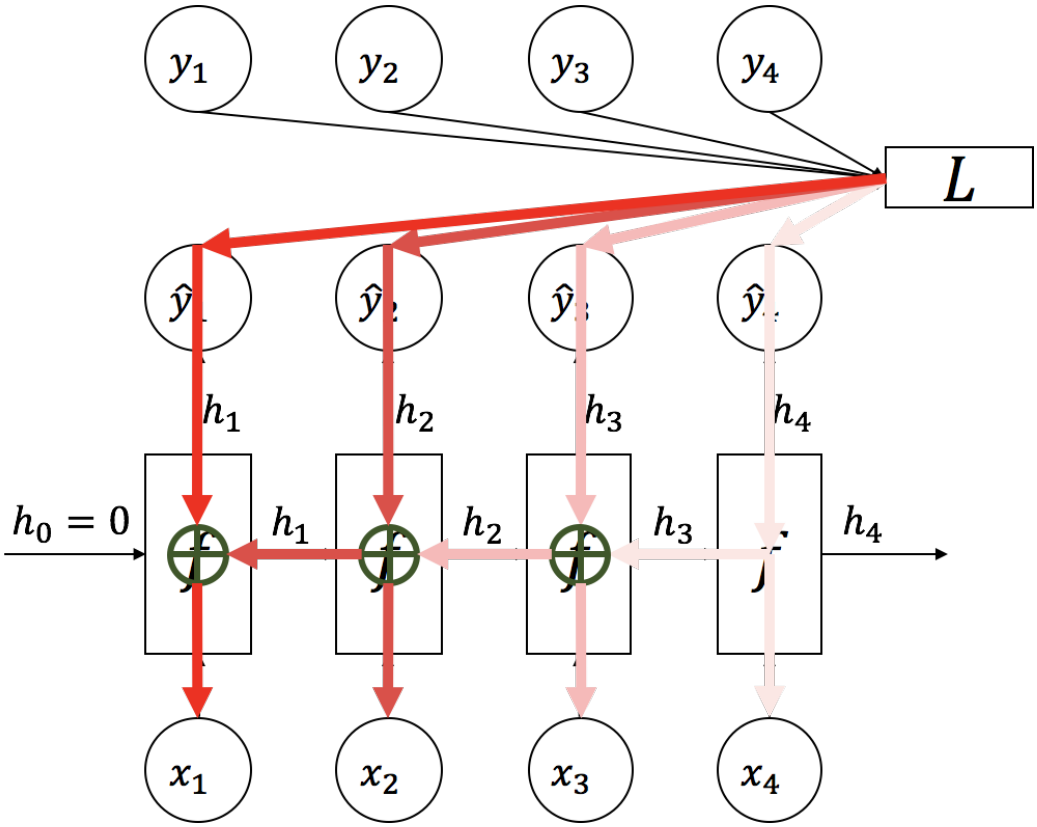

Figure 5: RNN에서 BPTT가 되는 모습

위 그림에서는 붉은색이 점점 짙어지는 것으로 그런 RNN back-propagation의 속성을 나타내었습니다. 이 속성을 back-propagation through time(BPTT)이라고 합니다.

이런 RNN back-propagation의 속성으로 인해, 마치 RNN은 time-step의 수 만큼 layer(계층)이 있는 것이나 마찬가지가 됩니다. 따라서 time-step이 길어짐에 따라, 매우 깊은 신경망과 같이 동작 합니다.

Gradient Vanishing

상기 했듯이, BPTT로 인해 RNN은 마치 time-step 만큼의 layer가 있는 것과 비슷한 속성을 띄게 됩니다. 그런데 위의 RNN의 수식을 보면, 활성함수(activation function)으로 tanh(Hyperbolic Tangent, '탄에이치'라고 읽기도 합니다.)가 사용 된 것을 볼 수 있습니다. tanh은 아래와 같은 형태를 띄고 있습니다.

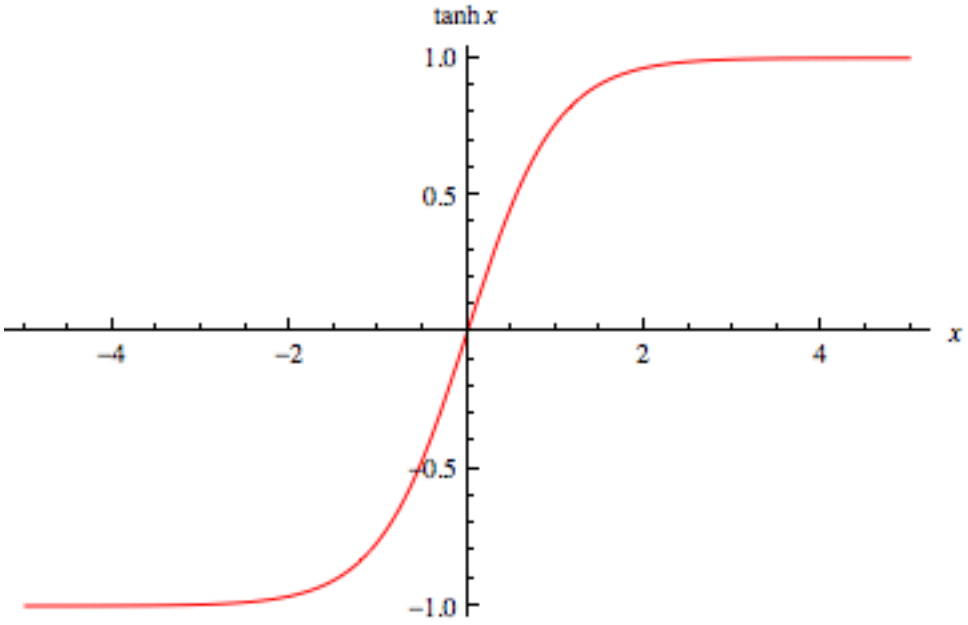

Figure 6: Hyperbolic Tangent의 형태

tanh의 양 끝은 수평에 가깝게되어 점점 −1 또는 1에 근접하는 것을 볼 수 있습니다. 문제는 이렇게 되면, tanh 양 끝의 gradient는 0에 가까워진다는것 입니다. 따라서 tanh 양 끝의 값을 반환하는 layer의 경우에는 gradient가 0에 가깝게 되어, 그

다음으로 back-propgation 되는 layer는 제대로 된 gradient를 전달 받을 수가 없게 됩니다. 이를 gradient vanishing이라고 합니다.

따라서, time-step이 많거나 여러층으로 되어 있는 신경망의 경우에는 이 gradient vanishing 문제가 쉽게 발생하게 되고, 이는 딥러닝 이전의 신경망 학습에 큰 장애가 되곤 하였습니다.

Multi-layer RNN

기본적으로 Time-step별로 RNN이 동작하지만, 아래의 그림과 같이 한 time-step 내에서 RNN을 여러 층을 쌓아올릴 수 있습니다. 그림상으로 시간의 흐름은 왼쪽에서 오른쪽으로 간다면, 여러 layer를 아래에서 위로 쌓아 올릴 수 있습니다. 따라서 여러개의 RNN layer가 쌓여 하나의 RNN을 이루고 있을 때, 가장 위층의 hidden state가 전체 RNN의 출력값이 됩니다.

당연히 각 층 별로 파라미터 θ를 공유하지 않고 따로 갖습니다. 보통은 각 layer 사이에 dropout을 끼워 넣기도 합니다.

기존의 단층 RNN의 경우에는 hidden state와 출력값이 같은 값이었지만, 여러 층이 쌓여 이루어진 RNN의 경우에는 각 time-step의 출력값이 맨 윗층의 hidden state가 됩니다.

Bi-directional RNN

여러 층을 쌓는 방법에 대해 이야기 했다면, 이제 RNN의 방향에 대해서 이야기 할 차례 입니다. 이제까지 다룬 RNN은 t가 1에서부터 마지막 time-step 까지 차례로 입력을 받아 진행 하였습니다. 하지만, bi-directional(양방향) RNN을 사용하게 되면, 기존의 정방향과 추가적으로 마지막 time-step에서부터 거꾸로 역방향으로 입력을 받아 진행 합니다. Bi-directional RNN의 경우에도 당연히 정방향과 역방향의 파라미터 θ는 공유되지 않습니다.

보통은 여러 층의 bi-directional RNN을 쌓게 되면, 각 층마다 두 방향의 각 time-step 별 출력(hidden state)값을 이어붙여(concatenate) 다음 층(layer)의 각 방향 별 입력으로 사용하게 됩니다. 경우에 따라서 전체 RNN layer들 중에서 일부 층만 bi-directional을 사용하기도 합니다.

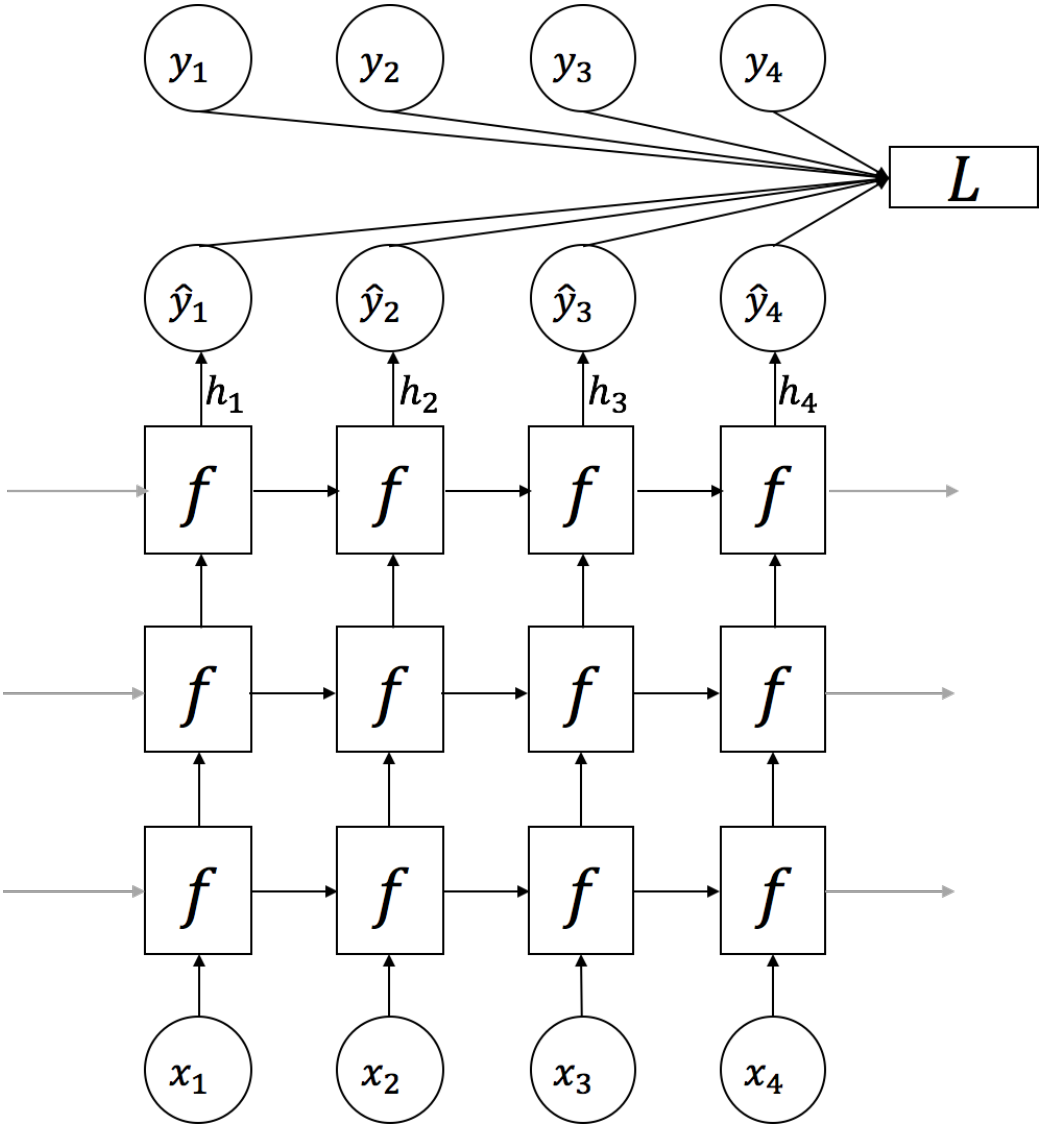

Figure 7: 여러 층이 쌓인 RNN의 형태

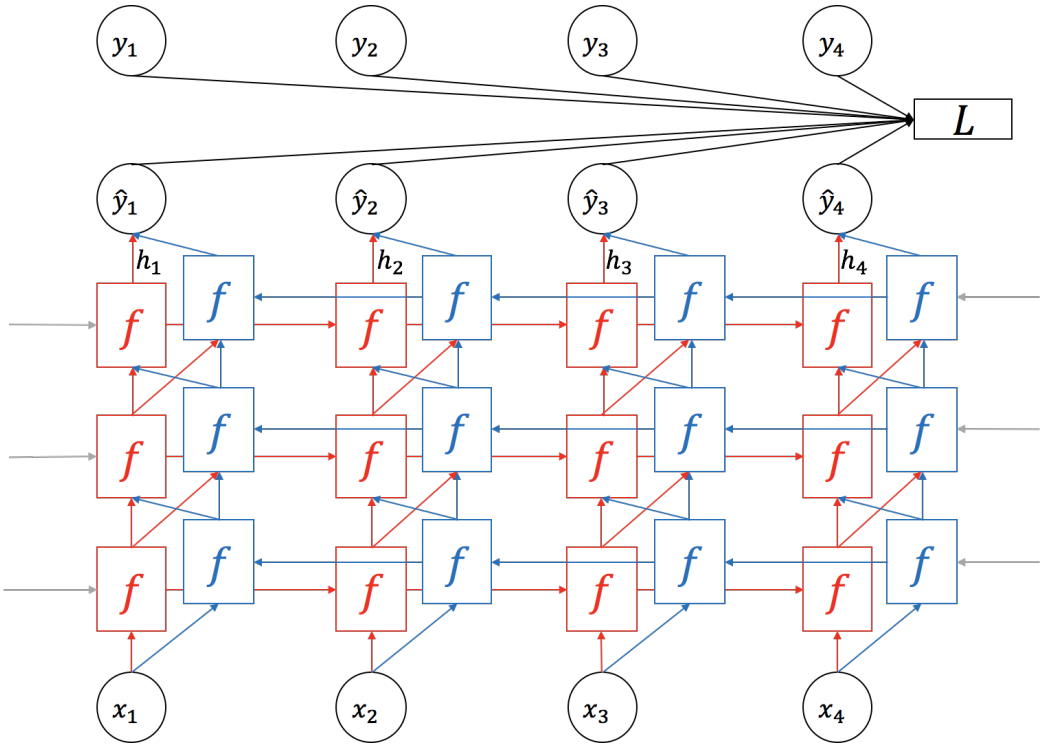

Figure 8: 두 방향으로 hidden state를 전달 및 계산하는 RNN의 형태

How to Apply to NLP

그럼 위에서 다룬 내용을 바탕으로 RNN을 NLP를 비롯한 실무에서는 어떻게 적용하는지 알아보도록 하겠습니다. 여기서는 RNN을 한개 층만 쌓아 정방향으로만 다룬 것 처럼 묘사하였지만, 여러 층을 양방향으로 쌓아 사용하는 것도 대부분의 경우 가능 합니다.

Use only last hidden state as output

가장 쉬운 사용케이스로 마지막 time-step의 출력값만 사용하는 경우입니다.

가장 흔한 예제로 그림의 감성분석과 같이 텍스트 분류(text classification)의 경우에 단어(토큰)의 갯수 만큼 입력이 RNN에 들어가고, 마지막 time-step의 결과값을 받아서 softmax 함수를 통해 해당 입력 텍스트의 클래스(class)를 예측하는 확률 분포를 근사(approximate)하도록 동작 하게 됩니다.

$$\text{softmax}(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

이때, 각 time-step 별 입력 단어 x_t는 one-hot vector로 표현(encoded)되고 embedding layer를 거쳐 정해진 dimension의 word embedding vector로 표현되어 RNN에 입력으로 주어지게 됩니다. 마찬가지로 정답 클래스 또한 one-hot vector가 되어 cross entropy 손실함수(loss function)를 통해 softmax 결과값인 각 클래스 별 확률을 나타낸 (multinoulli) 확률 분포 vector와 비교하여 손실(loss)값을 구하게 됩니다.

$$\text{CrossEntropy}(y_{1:n}, \hat{y}_{1:n}) = \frac{1}{n}\sum_{i=1}^{n} y_i^T \hat{y}_i$$

Use all hidden states as output

그리고 또 다른 많이 이용되는 방법은 모든 time-step의 출력값을 모두 사용하는 것 입니다. 우리는 이 방법을 언어모델(language modeling)이나 기계번역(machine translation)으로 실습 해 볼 것이지만, 굳이 그런 방법이 아니어도, 문장을 입력으로 주고, 각 단어 별 형태소를 분류(classification)하는 문제라던지 여러가지 방법으로 응용이 가능합니다.

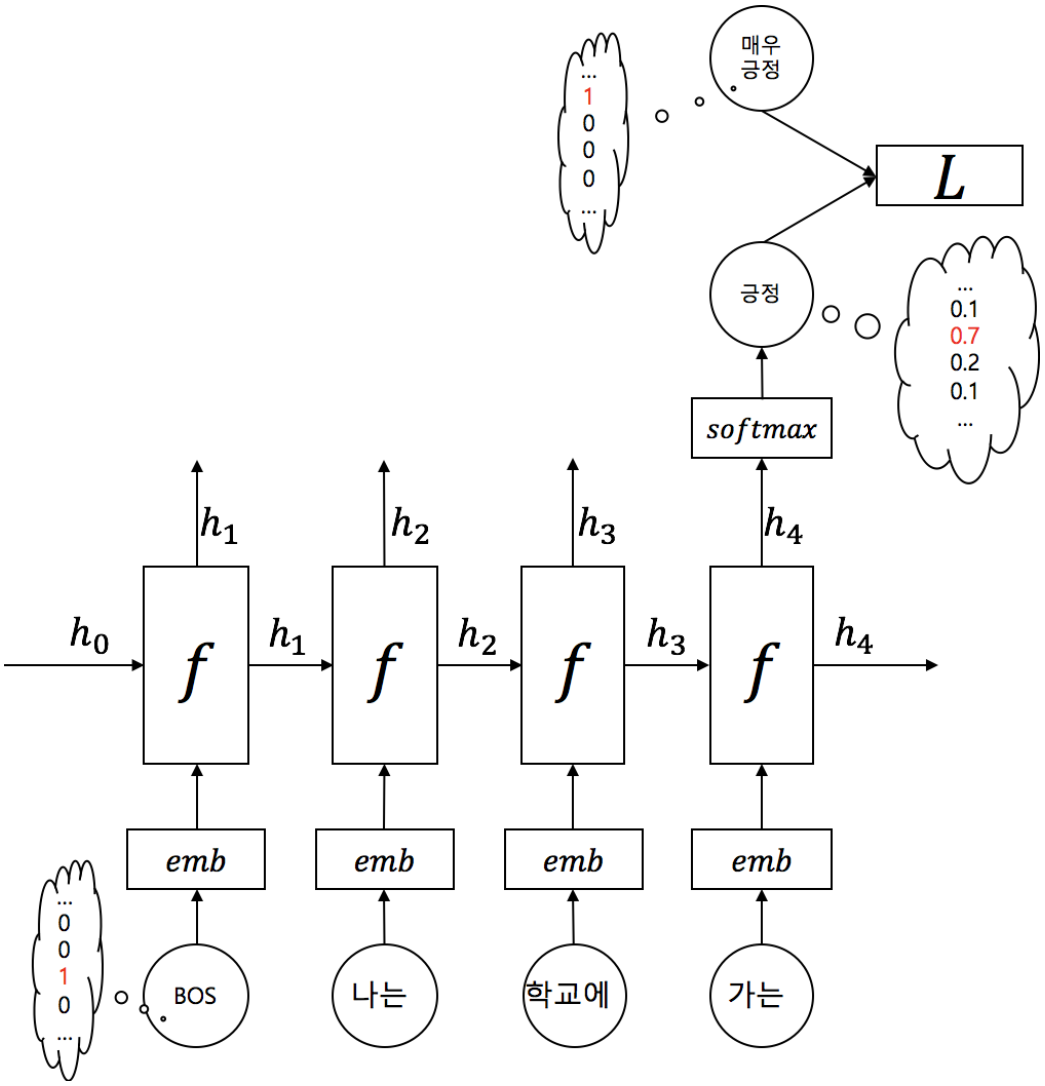

Figure 9: RNN의 마지막 time-step의 출력을 사용 하는 경우

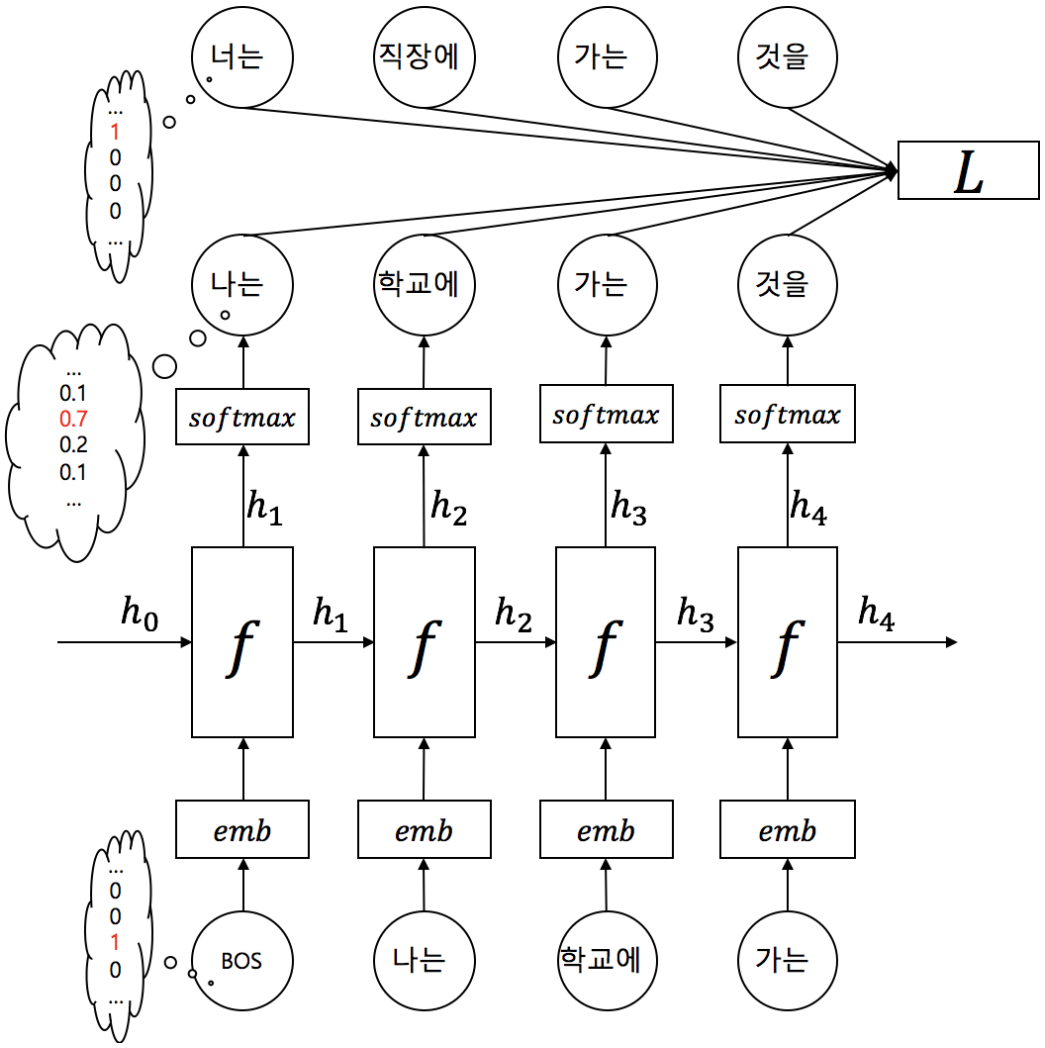

Figure 10: 모든 time-step의 출력을 사용 하는 경우

그림과 같이 각 time-step 별로 입력을 받아 RNN을 거치고 나서, 각 time-step별로 어떠한 결과물을 출력 하여, 각 time-step 별 정답과 비교하여 손실(loss)를 구합니다. 이때에도 각 단어는 one-hot vector로 표현 될 수 있으며, 그 경우에는 embedding layer를 거쳐 word embedding vector로 변환 된 후, RNN에 입력으로 주어지게 됩니다.

대부분의 경우 RNN은 여러 층(layer)과 양방향(bi-directional)으로 구현 될 수 있습니다. 하지만 입력과 출력이 같은 데이터를 공유 하는 경우에는 bi-directional RNN을 사용할 수 없습니다. 좀 더 구체적으로 설명하면 이전 time-step이 현재 time-step의 입력으로 사용되는 모델 구조의 경우에는 bi-directional RNN을 사용할 수 없습니다. 위의 그림도 그 경우에 해당 합니다. 하지만 형태소 분류기와 같이 출력이 다음 time-step에 입력에 영향을 끼치지 않는 경우에는 bi-directional RNN을 사용할 수 있습니다.

Conclusion

위와 같이 RNN은 가변길이의 입력을 받아 가변길이의 출력을 내어줄 수 있는 모델입니다. 하지만 기본(Vanilla) RNN은 time-step이 길어질 수록 앞의 데이터를 기억하지 못하는 치명적인 단점이 있습니다. 이를 해결하기 위해서 Long Short Term Memory (LSTM)이나 Gated Recurrent Unit (GRU)와 같은 응용 아키텍쳐들이 나왔고 훌륭한 개선책이 되어 널리 사용되고 있습니다. # Long Short Term Memory (LSTM)

RNN은 가변길이의 입력에 대해서 훌륭하게 동작하지만, 그 길이가 길어지면 오래된 데이터를 잊어버리는 치명적인 단점이 있었습니다. 하지만 Long Short Term Memory(LSTM)의 등장으로 RNN을 단점을 보완할 수 있게 되었습니다. LSTM은 기존 RNN의 hidden state 이외에도 별도의 cell state라는 변수(variable)를 두어, 그 기억력을 증가시켰을 뿐만 아니라, 여러가지 게이트(gate)를 두어 기억하거나, 잊어버리거나, 출력하고자 하는 데이터의 양을 상황에 따라서, 마치 수도꼭지를 잠궜다 열듯이, 효과적으로 제어하여 긴 길이의 데이터에 대해서도 효율적으로 대처할 수 있게 되었습니다.

하지만 덕분에 LSTM의 수식은 덕분에 RNN에 비하면 굉장히 복잡해지게 되었고, 더 많아진 파라미터들을 훈련시키기 위해서는 상대적으로 더 많은 데이터들로 더 오래 훈련 해야 합니다. 다행히, 빅데이터의 시대를 맞이하여 범람하는 정보들과, 그래픽카드의 발달로 인한 GPGPU의 빨라진 속도로 인해, 지금은 아무런 어려움 없이 당연하게 LSTM을 사용하는 시대가 되었습니다.

아래는 LSTM의 수식입니다.

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

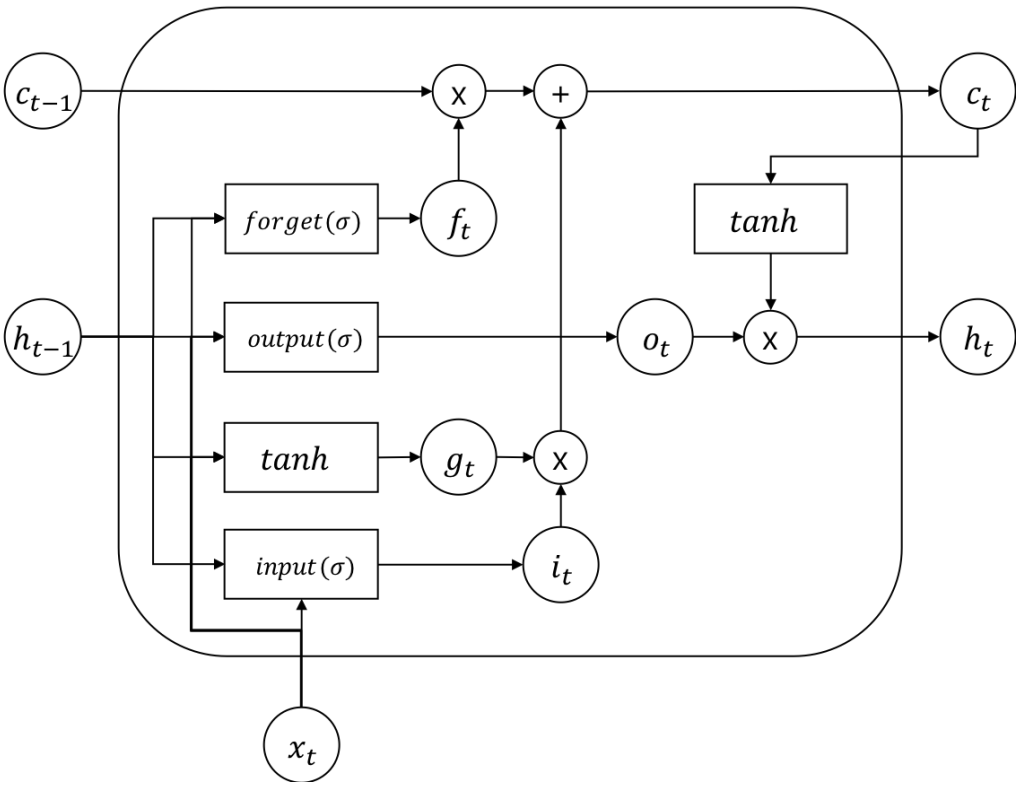

Figure 11: LSTM의 구조

각 게이트(gate) 앞에는 sigmoid(σ)가 붙어 0에서 1 사이의 값으로 얼마나 게이트를 열고 닫을지를 결정합니다. 그럼 그 결정된 값에 따라서 cell state c_{t-1}와 g_t, c_t가 새롭게 인코딩(encoding) 됩니다.

RNN과 마찬가지로 LSTM 또한 여러층으로 쌓거나, 양방향(bi-directional)으로 구현할 수 있습니다. 더 길어진 길이에 대해서도 RNN보다 훨씬 훌륭하게

대처하지만, 무한정 길어지는 길이에 대처 할 수 있는 것은 아닙니다. 따라서 여전히 긴 길이의 데이터에 대해서 기억하지 못하는 문제점은 아직 남아 있습니다. # Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU)는 뉴욕대학교 조경현 교수가 제안한 방법입니다. 기존 LSTM이 너무 복잡한 모델이므로 좀 더 간단하면서 성능이 비슷한 방법을 제안하였습니다. 마찬가지로 GRU 또한 sigmoid σ로 구성 된 리셋(reset) 게이트(gate) r_t와 업데이트(update) 게이트 z_t가 있습니다. 마찬가지로 sigmoid 함수로 인해서 게이트의 출력값은 0과 1 사이가 나오므로, 데이터의 흐름을 게이트를 열고 닫아 제어할 수 있습니다. 기존 LSTM 대비 게이트의 숫자가 줄어든 것($4 \rightarrow 2$)을 볼 수 있고, 따라서 게이트에 딸려있던 파라미터들이 그만큼 줄어든 것입니다.

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t)n_t + z_t h_{(t-1)}
\end{aligned}
$$

사실 LSTM 대비 GRU는 더 가벼운 몸집을 자랑하지만, 아직까지는 LSTM을 사용하는 빈도가 더 높은 것이 사실이긴 합니다. 사실 특별히 성능의 차이가 있다고 하기보단, LSTM과 GRU의 learning-rate나 hidden size등의 하이퍼 파라미터(hyper-parameter)가 다르므로 사용 모델에 따라서 다시 파라미터 셋팅을 연구 해야 하므로, 쉽게 이것저것 사용하기 힘든 부분도 있을 뿐더러, 연구자의 취향에 가까운 것 같습니다. # Gradient Clipping

RNN은 BPTT를 통해서 gradient를 구합니다. 따라서 출력의 길이에 따라서 gradient의 크기가 달라지게 됩니다. 즉, 길이가 길수록 자칫 gradient가 너무 커질 수 있기 때문에, learning rate를 조절하는 일이 필요 합니다. 너무 큰 learning rate를 사용하게 되면 gradient descent에서 step의 크기가 너무 커져버려 잘못된 방향으로 학습 및 발산(gradient exloding) 해 버릴 수 있기 때문입니다. 이 경우 가장 쉬운 대처 방법은 learning rate를 아주 작은 값을 취하는 것 입니다. 하지만 작은 learning rate를 사용할 경우, 평소 상황에서 너무 적은 양만 배우므로 훈련 속도가 매우 느려질 것 입니다. 즉, 길이는 가변이므로 learning rate를 그때그때 알맞게 최적의 값을 찾아 조절 해 주는 것은 매우 어려운 일이 될 것입니다. 이때 gradient clipping이 큰 힘을 발휘합니다.

Gradient clipping은 신경망 파라미터 θ의 norm (보통 L2 norm)을 구하고, 이 norm의 크기를 제한하는 방법 입니다. 따라서 gradient vector는 방향은 유지하되,

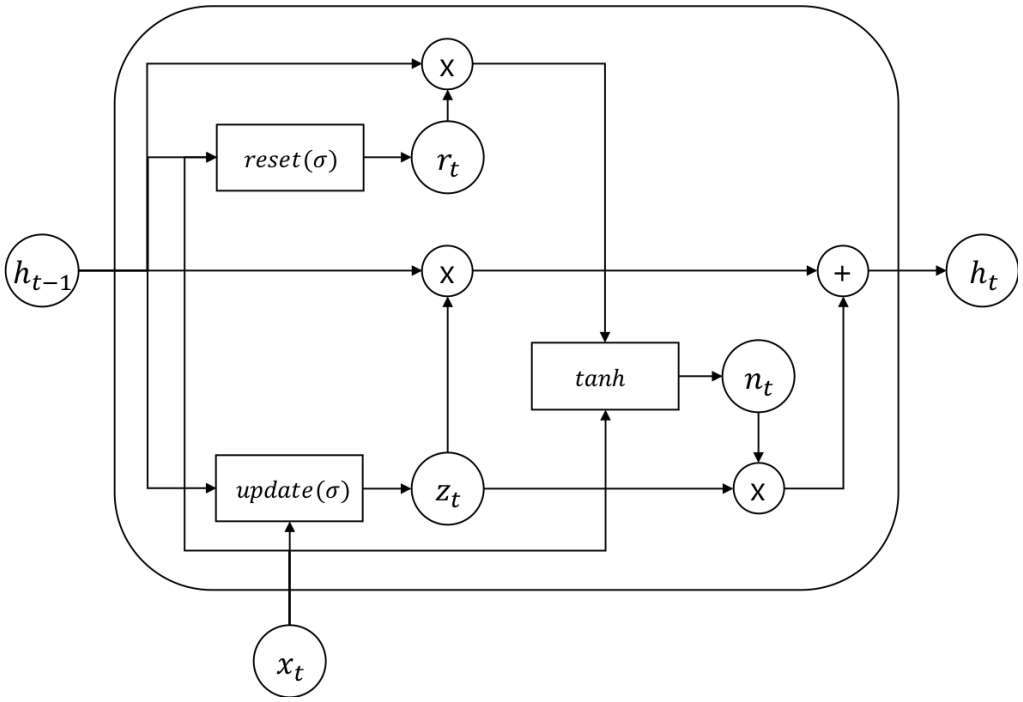

Figure 12: GRU의 구조

그 크기는 학습이 망가지지 않은 정도로 줄어들 수 있게 됩니다. 물론 norm의 maximum value를 따로 사용자가 정의 해 주어야 하기 때문에, 또 하나의 hyper-parameter가 생기게 되지만, 큰 norm을 가진 gradient vector의 경우에만 gradient clipping을 수행하기 때문에, 능동적으로 learning rate를 조절하는 것과 비슷한 효과를 가질 수 있습니다. 따라서 gradient clipping은 RNN 계열의 학습 및 훈련을 할 때 널리 사용되는 방법 입니다.

$$\frac{\partial \epsilon}{\partial \theta} \leftarrow \begin{cases} \frac{\text{threshold}}{\|\hat{g}\|}\hat{g} & \text{if } \|\hat{g}\| \geq \text{threshold} \\ \hat{g} & \text{otherwise} \end{cases}$$
$$\text{where } \hat{g} = \frac{\partial \epsilon}{\partial \theta}.$$

수식을 보면, gradient norm이 정해진 threshold(역치)보다 클 경우에, gradient 벡터를 threshold 보다 큰 만큼의 비율로 나누어 줍니다. 따라서 gradient는 항상 threshold 보다 작으며 이는 gradient exploding을 방지함과 동시에, gradient의 방향을 유지해주기 때문에 모델 파라미터 θ가 학습해야 하는 방향은 잃지 않습니다.

PyTorch에서도 기능을 torch.nn.utils.clip_grad_norm_ 이라는 함수를 제공하고 있으므로 매우 쉽게 사용 할 수 있습니다.