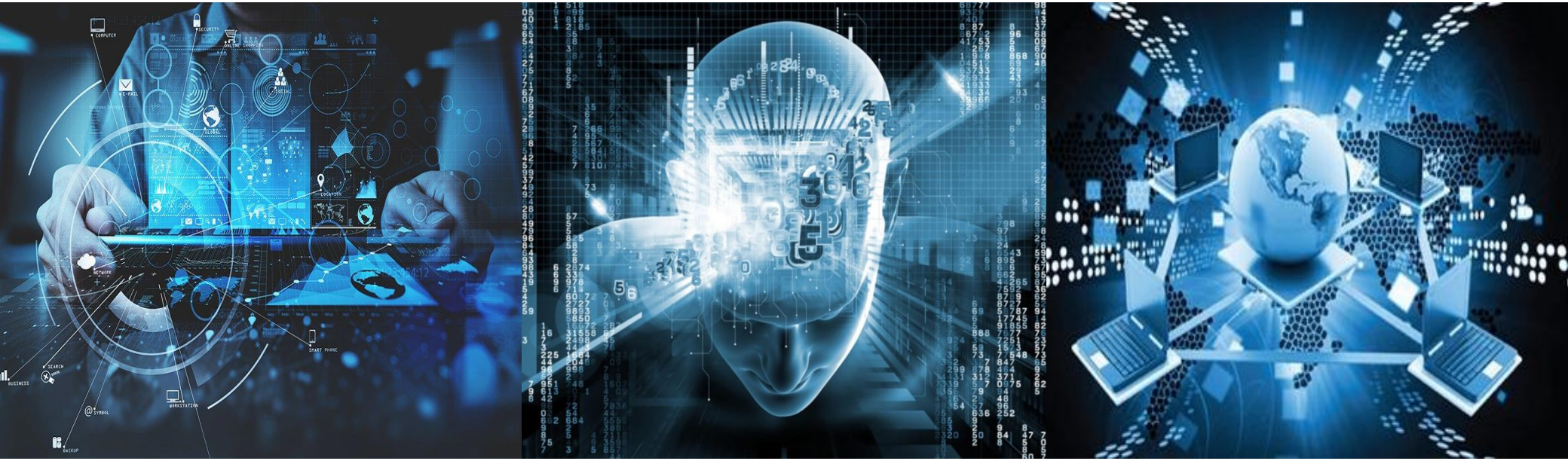
KMI 한국기상산업기술원
Korea Meteorological Institute

빅파이 크래프트
Find Information to design valuable society



기상기후 빅데이터와 지능정보기술 활용과정

AI핵심기술 - RNN(Recurrent Neural Networks)

Training BigData Experts & Empowering AI Technology

Kim Jin Soo

학습내용

- RNN(Recurrent Neural Networks) Overview
- Step 1 – String Sequence
- Step 2 – Sentence Sequence
- Step 3 – Only Softmax Sentence Sequence
- Step 4 – Ling Sentence Sequence
- Step 5 – Time series Prediction
- Summary

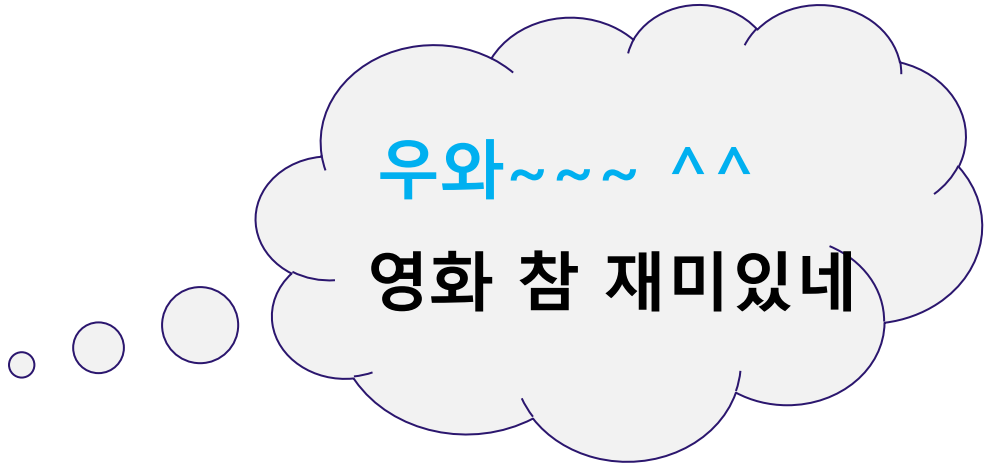
RNN (Recurrent Neural Network)

- 이미지인식에 CNN(Convolution NN)이 있다면, 자연어인식에는 RNN(Recurrent NN)이 있다.
- RRN은 상태가 고정된 데이터를 처리하는 다른 신경망과는 달리 자연어처리나 음성인식처럼 순서가 있는 데이터를 처리하는 데 강정을 가진 신경망이다.

긍정 ? 부정

RNN (Recurrent Neural Network)

- 이미지인식에 CNN(Convolution NN)이 있다면, 자연어인식에는 RNN(Recurrent NN)이 있다.
- RRN은 상태가 고정된 데이터를 처리하는 다른 신경망과는 달리 자연어처리나 음성인식처럼 순서가 있는 데이터를 처리하는 데 강정을 가진 신경망이다.



긍정 > 부정

RNN (Recurrent Neural Network)

- 이미지인식에 CNN(Convolution NN)이 있다면, 자연어인식에는 RNN(Recurrent NN)이 있다.
- RRN은 상태가 고정된 데이터를 처리하는 다른 신경망과는 달리 자연어처리나 음성인식처럼 순서가 있는 데이터를 처리하는 데 강정을 가진 신경망이다.
- **앞이나 뒤의 정보에 따라 전체의 의미가 달라진다.**

긍정 < 부정

Translation Level of Korean from Outside

◆ 기존, 불과 2~3년 전의 번역 서비스 수준? 해외에서의 한글 번역 한번 보시라!

Traditional Translation Level in Korea

◆ 그렇다면, 한국에서의 번역은 달랐을까요?

Traditional Translation Level in Korea

한식 외국어 표기
오역 사례와
한국관광공사가
제시한 번역

오역 영어 표기 ▶
관광공사 제시한 번역 ▶

Six Times
Beef Tartare

Bear Tang
Beef Bone Soup

Chicken Asshole House
Stir-fried Chicken Gizzards

Lifestyle Meat
Beef, Pork

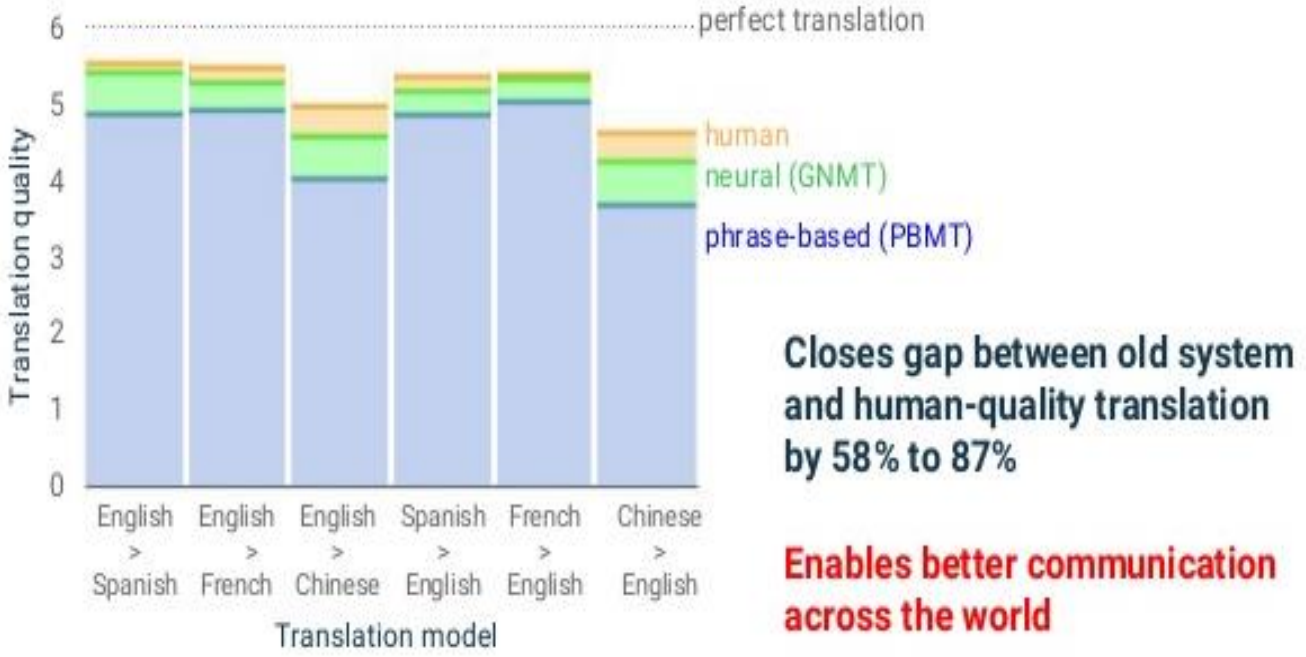
Dynamic Stew
Pollack Stew

Impression of foreigners in Korean

잘못된 메뉴 표기는 한식 전체의 **품격**을 떨어뜨릴 수 있다.

GNMT, Google's Translation System

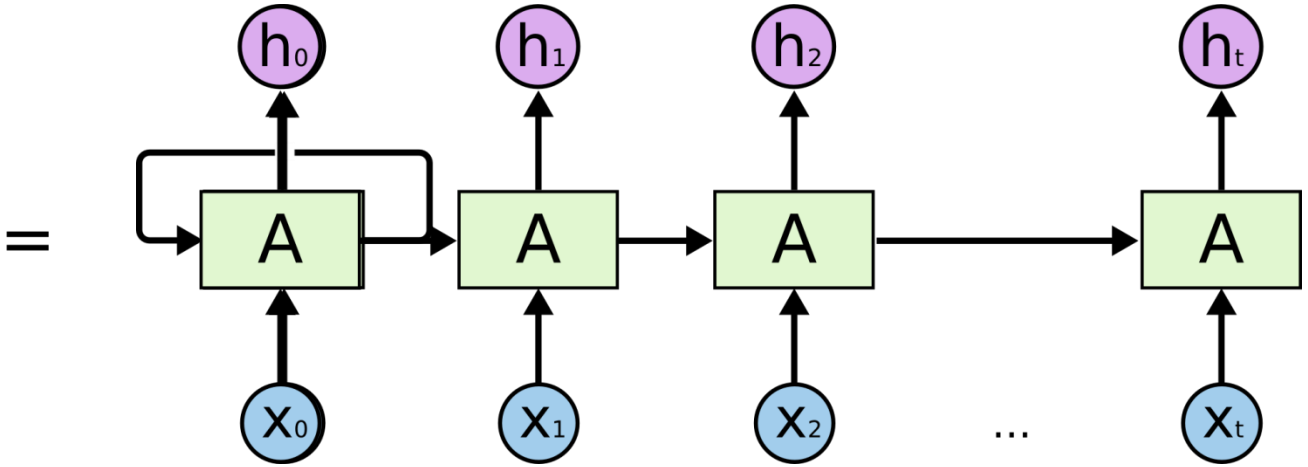
- 2016년 알파고와 함께 한참 화제가 된 구글의 신경망 기반 기계변역, GNMT
  → RNN을 이용하며 만든 서비스
- 지속적인 학습으로 빠르게 성능을 개선하여, 몇몇 언어에서는 인간에 가까운 수준에 도달



[구글 기계번역 성능 그래프, 출처: https://goo.gl/jRlrL4]

RNN (Recurrent Neural Network)

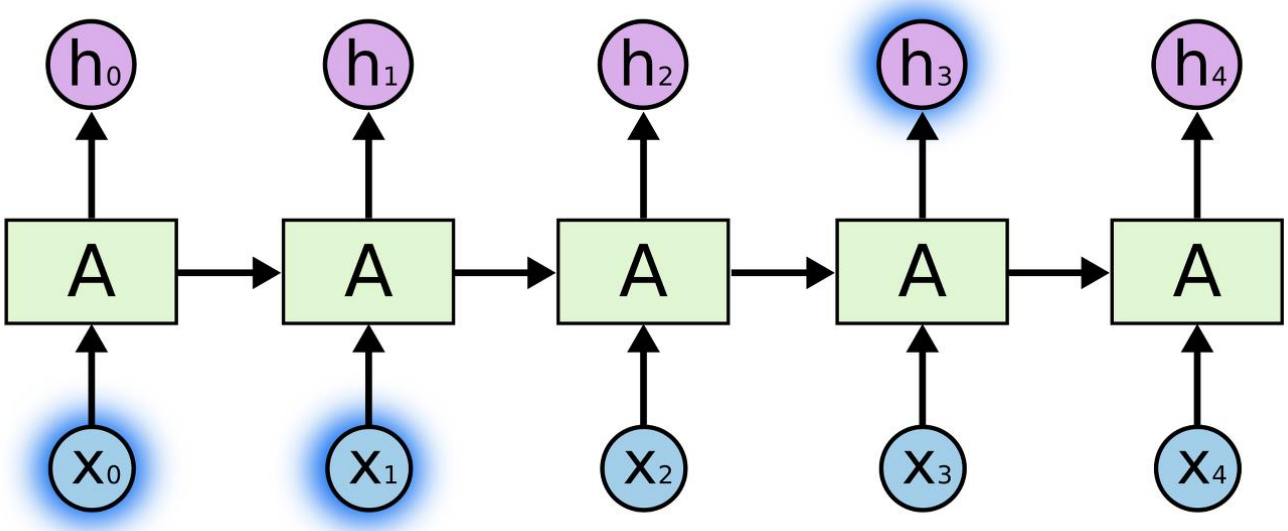
- Deep Learning 알고리즘 중 순차적인 데이터를 학습하여 Classification / Prediction 수행
- 기존의 DNN(Deep Neural Network)의 경우 각 Layer마다 parameter 들이 독립적이었으나, RNN은 이를 공유
- 현재의 출력 결과는 이전 time step의 결과에 영향을 받으며, hidden layer는 일종의 메모리 역할을 하게 됨



Long-Term Dependency Problem

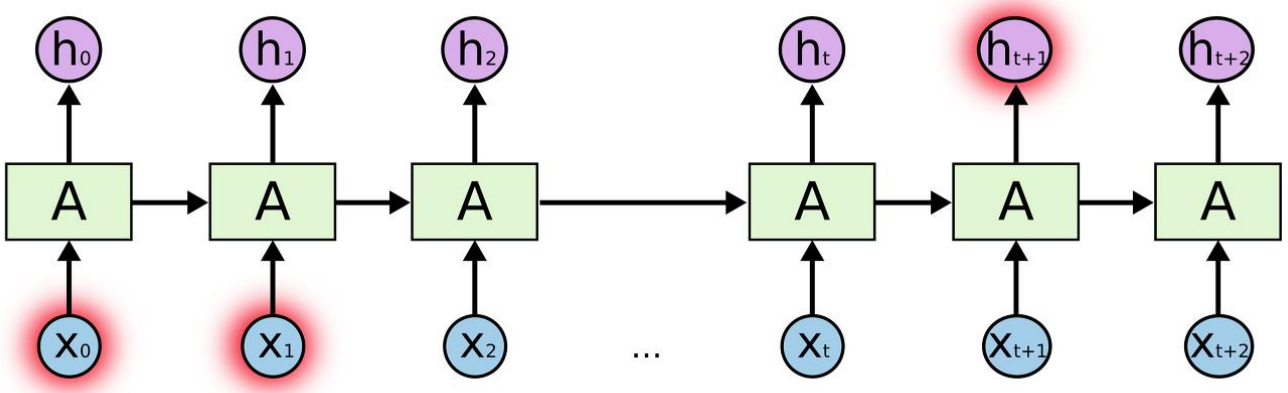
- RNN의 장점은 이전의 정보를 현재의 문제 해결에 활용할 수 있다는 점이다.
- 제공된 데이터와 배워야 할 정보의 입력 위치 차이(Gap)가 크지 않다면, RNN은 과거의 데이터를 기반으로 학습을 할 수 있다.
- 하지만, 더 많은 문맥이 필요할 때가 있다.

  Eg. "I grew up in France. I speak fluent OOOOOO." → 마지막 단어 **French** 를 예측!



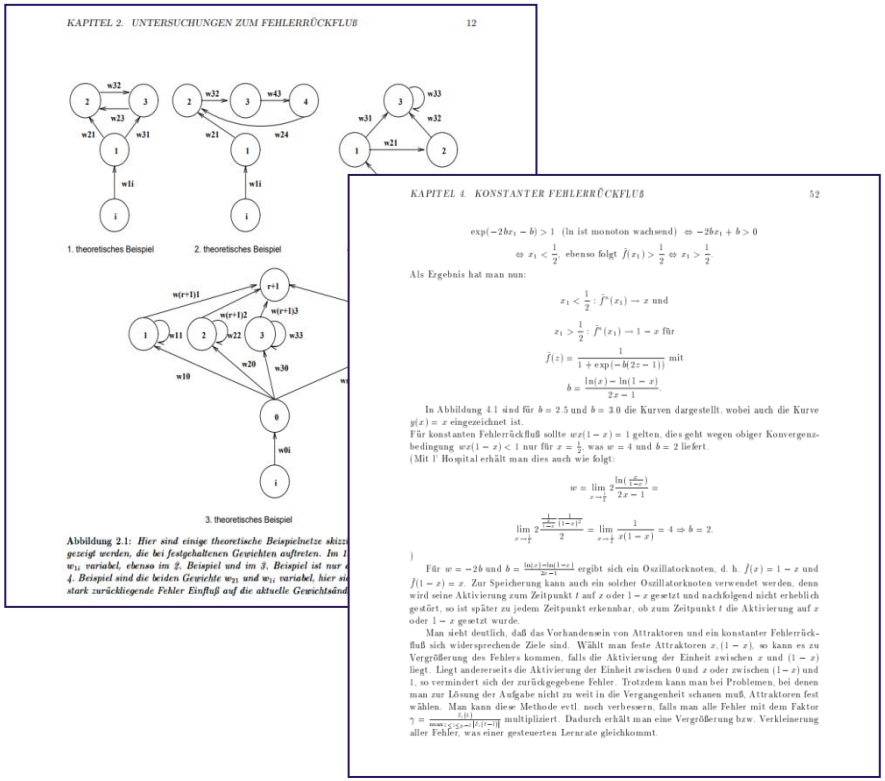
Long-Term Dependency Problem

- 최근 정보를 기반으로 예측 모델은 다음 단어가 아마도 언어의 한 종류라고 예측한다.
- 실제 "I grew up in France" 는 표현과 "I speak fluent OOOOOO." 라는 표현의 위치가 멀어지는 문제는 아주 빈번하게 발생한다.
- 불행히도, 이런 문장표현의 순서상 갭이 커질수록, RNN은 두 정보의 문맥을 연결하기 힘들다.
  → 이러한 **장기의존성 문제**를 풀 수 있도록 신중하게 파라미터를 선택 !?!



Long-Term Dependency Problem

◆ 슬프게도 실제적으로 RNN은 이러한 문맥을 배울 수 없다는 것이 밝혀졌다.



KAPITEL 2. UNTERSUCHUNGEN ZUM FEHLERRÜCKFLUß 12

1. theoretisches Beispiel 2. theoretisches Beispiel 3. theoretisches Beispiel

Abbildung 2.1: *Hier sind einige theoretische Beispielnetze skizz… gezeigt werden, die bei festgehaltenen Gewichten auftreten. Im 1… w_{1i} variabel, ebenso im 2. Beispiel und im 3. Beispiel ist nur … 4. Beispiel sind die beiden Gewichte w_{21} und w_{1i} variabel, hier si… stark zurückliegende Fehler Einfluß auf die aktuelle Gewichtsänd…*

KAPITEL 4. KONSTANTER FEHLERRÜCKFLUß 52

$$\exp(-2bx_1 - b) > 1 \quad (\ln \text{ ist monoton wachsend}) \quad \Leftrightarrow -2bx_1 + b > 0$$

$$\Leftrightarrow x_1 < \frac{1}{2}, \text{ ebenso folgt } \tilde{f}(x_1) > \frac{1}{2} \Leftrightarrow x_1 > \frac{1}{2}.$$

Als Ergebnis hat man nun:

$$x_1 < \frac{1}{2} : \tilde{f}^n(x_1) \to x \text{ und}$$

$$x_1 > \frac{1}{2} : \tilde{f}^n(x_1) \to 1 - x \text{ für}$$

$$\tilde{f}(z) = \frac{1}{1 + \exp(-b(2z - 1))} \text{ mit}$$

$$b = \frac{\ln(x) - \ln(1 - x)}{2x - 1}.$$

In Abbildung 4.1 sind für $b = 2.5$ und $b = 3.0$ die Kurven dargestellt, wobei auch die Kurve $y(x) = x$ eingezeichnet ist.
Für konstanten Fehlerrückfluß sollte $wx(1 - x) = 1$ gelten, dies geht wegen obiger Konvergenzbedingung $wx(1 - x) < 1$ nur für $x = \frac{1}{2}$, was $w = 4$ und $b = 2$ liefert.
(Mit l' Hospital erhält man dies auch wie folgt:

$$w = \lim_{x \to \frac{1}{2}} 2\frac{\ln(\frac{x}{1-x})}{2x - 1} =$$

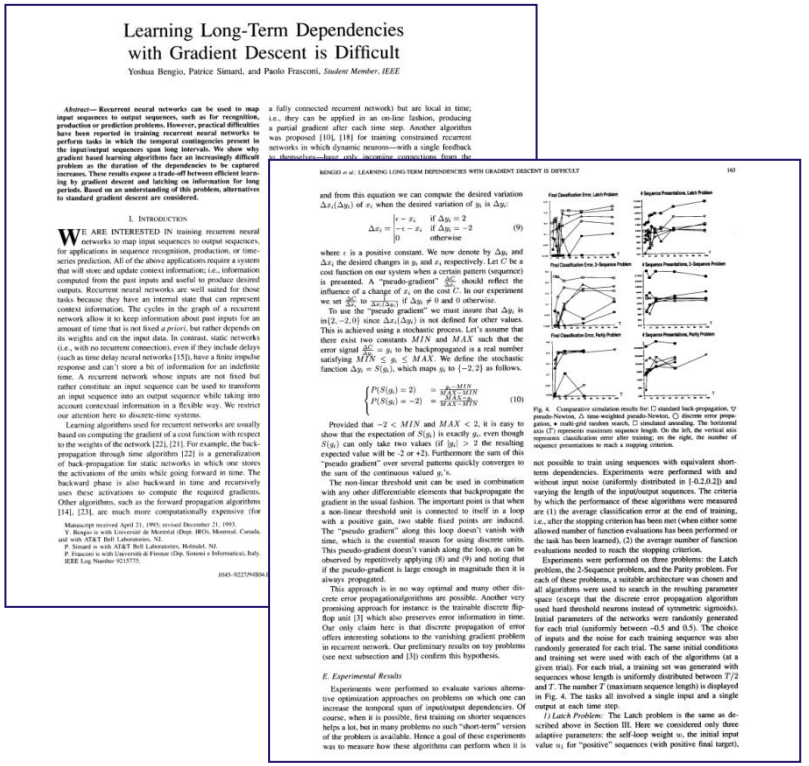
$$\lim_{x \to \frac{1}{2}} 2\frac{\frac{1}{\frac{x}{1-x}} \frac{1}{(1-x)^2}}{2} = \lim_{x \to \frac{1}{2}} \frac{1}{x(1-x)} = 4 \Leftrightarrow b = 2.$$

)

Für $w = -2b$ und $b = \frac{\ln(x) - \ln(1-x)}{2x-1}$ ergibt sich ein Oszillatorknoten, d. h. $\tilde{f}(x) = 1 - x$ und $\tilde{f}(1 - x) = x$. Zur Speicherung kann auch ein solcher Oszillatorknoten verwendet werden, denn wird seine Aktivierung zum Zeitpunkt t auf x oder $1 - x$ gesetzt und nachfolgend nicht erheblich gestört, so ist später zu jedem Zeitpunkt erkennbar, ob zum Zeitpunkt t die Aktivierung auf x oder $1 - x$ gesetzt wurde.

Man sieht deutlich, daß das Vorhandensein von Attraktoren und ein konstanter Fehlerrückfluß sich widersprechende Ziele sind. Wählt man feste Attraktoren $x, (1 - x)$, so kann es zu Vergrößerung des Fehlers kommen, falls die Aktivierung der Einheit zwischen x und $(1 - x)$ liegt. Liegt andererseits die Aktivierung der Einheit zwischen 0 und x oder zwischen $(1 - x)$ und 1, so vermindert sich der zurückgegebene Fehler. Trotzdem kann man bei Problemen, bei denen man zur Lösung der Aufgabe nicht zu weit in die Vergangenheit schauen muß, Attraktoren fest wählen. Man kann diese Methode evtl. noch verbessern, falls man alle Fehler mit dem Faktor $\gamma = \frac{1.0}{\max_{t \leq \tau \leq \tau'} |\tilde{f}'(\tau)|}$ multipliziert. Dadurch erhält man eine Vergrößerung bzw. Verkleinerung aller Fehler, was einer gesteuerten Lernrate gleichkommt.

Hochreiter (1991) :
http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf



Learning Long-Term Dependencies with Gradient Descent is Difficult

Yoshua Bengio, Patrice Simard, and Paolo Frasconi, *Student Member, IEEE*

Abstract— Recurrent neural networks can be used to map input sequences to output sequences, such as for recognition, production or prediction problems. However, practical difficulties have been reported in training recurrent neural networks to perform tasks in which the temporal contingencies present in the input/output sequences span long intervals. We show why gradient based learning algorithms face an increasingly difficult problem as the duration of the dependencies to be captured increases. These results expose a trade–off between efficient learning by gradient descent and latching on information for long periods. Based on an understanding of this problem, alternatives to standard gradient descent are considered.

I. INTRODUCTION

BENGIO *et al.*: LEARNING LONG-TERM DEPENDENCIES WITH GRADIENT DESCENT IS DIFFICULT 163

Bengio, et al(1994) :
http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf

LSTM, Long Short Term Memory

- 감사하게도, **LSTM 은 장기의존성 문제를 해결**하였다 !!
- LSTM : Long Short Term Memory networks 는 RNN의 한 종류이다.
- LSTM 컨셉은 Hochreiter & Schmidhuber(1997) 이 제안하였고,
  많이 개선되고 대중화되면서 다양한 문제에 적용되기 시작했다.

LONG SHORT-TERM MEMORY

Neural Computation 9(8):1735–1780, 1997

Sepp Hochreiter
Fakultät für Informatik
Technische Universität München
80290 München, Germany
hochreit@informatik.tu-muenchen.de
http://www7.informatik.tu-muenchen.de/~hochreit

Jürgen Schmidhuber
IDSIA
Corso Elvezia 36
6900 Lugano, Switzerland
juergen@idsia.ch
http://www.idsia.ch/~juergen

Abstract

Learning to store information over extended time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error back flow. We briefly review Hochreiter's 1991 analysis of this problem, then address it by introducing a novel, efficient, gradient-based method called "Long Short-Term Memory" (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing *constant* error flow through "constant error carrousels" within special units. Multiplicative gate units learn to open and close access to the constant error flow. LSTM is local in space and time; its computational complexity per time step and weight is $O(1)$. Our experiments with artificial data involve local, distributed, real-valued, and noisy pattern representations. In comparisons with RTRL, BPTT, Recurrent Cascade-Correlation, Elman nets, and Neural Sequence Chunking, LSTM leads to many more successful runs, and learns much faster. LSTM also solves complex, artificial long time lag tasks that have never been solved by previous recurrent network algorithms.

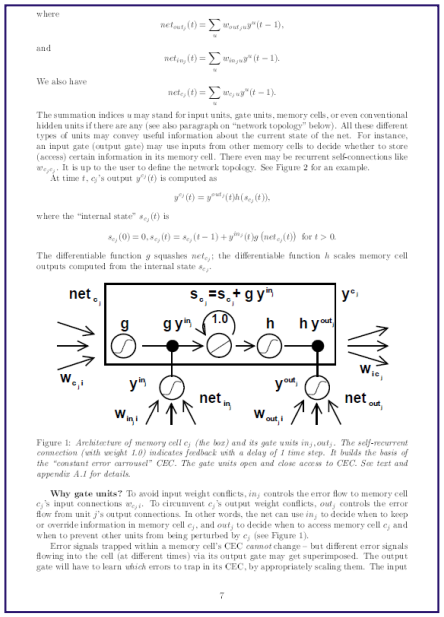
1 INTRODUCTION

Recurrent networks can in principle use their feedback connections to store representations of recent input events in form of activations ("short-term memory", as opposed to "long-term memory" embodied by slowly changing weights). This is potentially significant for many applications, including speech processing, non-Markovian control, and music composition (e.g., Mozer 1992). The most widely used algorithms for learning *what* to put in short-term memory, however, take too much time or do not work well at all, especially when minimal time lags between inputs and corresponding teacher signals are long. Although theoretically fascinating, existing methods do not provide clear *practical* advantages over, say, backprop in feedforward nets with limited time windows. This paper will review an analysis of the problem and suggest a remedy.

The problem. With conventional "Back-Propagation Through Time" (BPTT, e.g., Williams and Zipser 1992, Werbos 1988) or "Real-Time Recurrent Learning" (RTRL, e.g., Robinson and Fallside 1987), error signals "flowing backwards in time" tend to either (1) blow up or (2) vanish: the temporal evolution of the backpropagated error exponentially depends on the size of the weights (Hochreiter 1991). Case (1) may lead to oscillating weights, while in case (2) learning to bridge long time lags takes a prohibitive amount of time, or does not work at all (see section 3).

The remedy. This paper presents *"Long Short-Term Memory"* (LSTM), a novel recurrent network architecture in conjunction with an appropriate gradient-based learning algorithm. LSTM is designed to overcome these error back-flow problems. It can learn to bridge time intervals in excess of 1000 steps even in case of noisy, incompressible input sequences, without loss of short time lag capabilities. This is achieved by an efficient, gradient-based algorithm for an architecture

1



where

$$net_{out_j}(t) = \sum_u w_{out_j u} y^u(t-1),$$

and

$$net_{in_j}(t) = \sum_u w_{in_j u} y^u(t-1).$$

We also have

$$net_{c_j}(t) = \sum_u w_{c_j u} y^u(t-1).$$

The summation indices u may stand for input units, gate units, memory cells, or even conventional hidden units if there are any (see also paragraph on "network topology" below). All these different types of units may convey useful information about the current state of the net. For instance, an input gate (output gate) may use inputs from other memory cells to decide whether to store (access) certain information in its memory cell. There even may be recurrent self-connections like $w_{c_j c_j}$. It is up to the user to define the network topology. See Figure 2 for an example.

At time t, c_j's output $y^{c_j}(t)$ is computed as

$$y^{c_j}(t) = y^{out_j}(t) h(s_{c_j}(t)),$$

where the "internal state" $s_{c_j}(t)$ is

$$s_{c_j}(0) = 0, s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t) g\left(net_{c_j}(t)\right) \text{ for } t > 0.$$

The differentiable function g squashes net_{c_j}; the differentiable function h scales memory cell outputs computed from the internal state s_{c_j}.

Figure 1: *Architecture of memory cell c_j (the box) and its gate units in_j, out_j. The self-recurrent connection (with weight 1.0) indicates feedback with a delay of 1 time step. It builds the basis of the "constant error carrousel" CEC. The gate units open and close access to CEC. See text and appendix A.1 for details.*

Why gate units? To avoid input weight conflicts, in_j controls the error flow to memory cell c_j's input connections $w_{c_j i}$. To circumvent c_j's output weight conflicts, out_j controls the error flow from unit j's output connections. In other words, the net can use in_j to decide when to keep or override information in memory cell c_j, and out_j to decide when to access memory cell c_j and when to prevent other units from being perturbed by c_j (see Figure 1).

Error signals trapped within a memory cell's CEC *cannot* change – but different error signals flowing into the cell (at different times) via its output gate may get superimposed. The output gate will have to learn which errors to trap in its CEC, by appropriately scaling them. The input

7

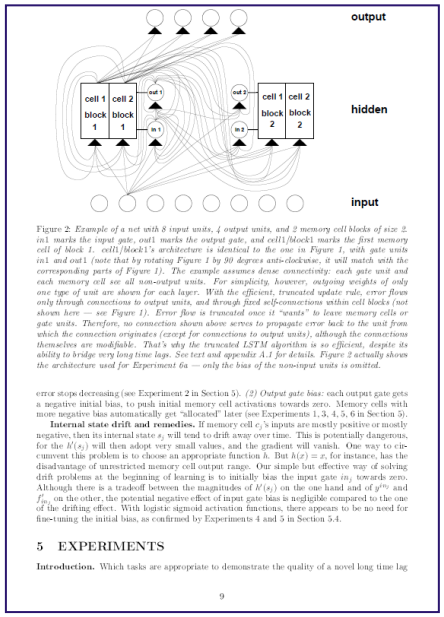


Figure 2: *Example of a net with 8 input units, 4 output units, and 2 memory cell blocks of size 2. $in1$ marks the input gate, $out1$ marks the output gate, and $cell1/block1$ marks the first memory cell of block 1. $cell1/block1$'s architecture is identical to the one in Figure 1, with gate units $in1$ and $out1$ (note that by rotating Figure 1 by 90 degrees anti-clockwise, it will match with the corresponding parts of Figure 1). The example assumes dense connectivity: each gate unit and each memory cell see all non-output units. For simplicity, however, outgoing weights of only one type of unit are shown for each layer. With the efficient, truncated update rule, error flows only through connections to output units, and through fixed self-connections within cell blocks (not shown here — see Figure 1). Error flow is truncated once it "wants" to leave memory cells or gate units. Therefore, no connection shown above serves to propagate error back to the unit from which the connection originates (except for connections to output units), although the connections themselves are modifiable. That's why the truncated LSTM algorithm is so efficient, despite its ability to bridge very long time lags. See text and appendix A.1 for details. Figure 2 actually shows the architecture used for Experiment 6a — only the bias of the non-input units is omitted.*

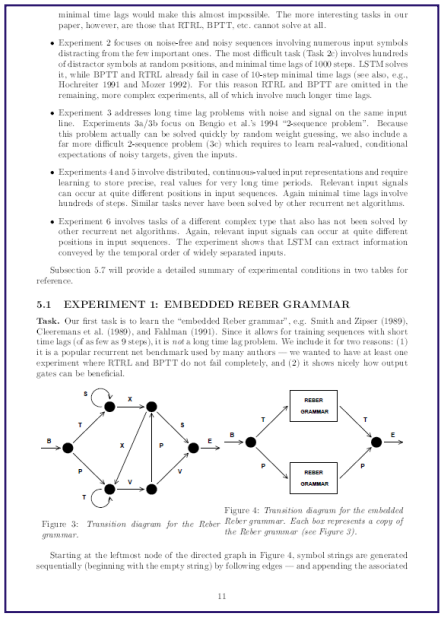
error stops decreasing (see Experiment 2 in Section 5). *(2) Output gate bias:* each output gate gets a negative initial bias, to push initial memory cell activations towards zero. Memory cells with more negative bias automatically get "allocated" later (see Experiments 1, 3, 4, 5, 6 in Section 5).

Internal state drift and remedies. If memory cell c_j's inputs are mostly positive or mostly negative, then its internal state s_j will tend to drift away over time. This is potentially dangerous, for the $h'(s_j)$ will then adopt very small values, and the gradient will vanish. One way to circumvent this problem is to choose an appropriate function h. But $h(x) = x$, for instance, has the disadvantage of unrestricted memory cell output range. Our simple but effective way of solving drift problems at the beginning of learning is to initially bias the input gate in_j towards zero. Although there is a tradeoff between the magnitudes of $h'(s_j)$ on the one hand and of y^{in_j} and f_{in_j} on the other, the potential negative effect of input gate bias is negligible compared to the one of the drifting effect. With logistic sigmoid activation functions, there appears to be no need for fine-tuning the initial bias, as confirmed by Experiments 4 and 5 in Section 5.4.

5 EXPERIMENTS

Introduction. Which tasks are appropriate to demonstrate the quality of a novel long time lag

9



minimal time lags would make this almost impossible. The more interesting tasks in our paper, however, are those that RTRL, BPTT, etc. cannot solve at all.

- Experiment 2 focuses on noise-free and noisy sequences involving numerous input symbols distracting from the few important ones. The most difficult task (Task 2c) involves hundreds of distractor symbols at random positions, and minimal time lags of 1000 steps. LSTM solves it, while BPTT and RTRL already fail in case of 10-step minimal time lags (see also, e.g., Hochreiter 1991 and Mozer 1992). For this reason RTRL and BPTT are omitted in the remaining, more complex experiments, all of which involve much longer time lags.
- Experiment 3 addresses long time lag problems with noise and signal on the same input line. Experiments 3a/3b focus on Bengio et al.'s 1994 "2-sequence problem". Because this problem actually can be solved quickly by random weight guessing, we also include a far more difficult 2-sequence problem (3c) which requires to learn real-valued, conditional expectations of noisy targets, given the inputs.
- Experiments 4 and 5 involve distributed, continuous-valued input representations and require learning to store precise, real values for very long time periods. Relevant input signals can occur at quite different positions in input sequences. Again minimal time lags involve hundreds of steps. Similar tasks never have been solved by other recurrent net algorithms.
- Experiment 6 involves tasks of a different complex type that also has not been solved by other recurrent net algorithms. Again, relevant input signals can occur at quite different positions in input sequences. The experiment shows that LSTM can extract information conveyed by the temporal order of widely separated inputs.

Subsection 5.7 will provide a detailed summary of experimental conditions in two tables for reference.

5.1 EXPERIMENT 1: EMBEDDED REBER GRAMMAR

Task. Our first task is to learn the "embedded Reber grammar", e.g. Smith and Zipser (1989), Cleeremans et al. (1989), and Fahlman (1991). Since it allows for training sequences with short time lags (of as few as 9 steps), it is *not* a long time lag problem. We include it for two reasons: (1) it is a popular recurrent net benchmark used by many authors — we wanted to have at least one experiment where RTRL and BPTT do not fail completely, and (2) it shows nicely how output gates can be beneficial.

Figure 3: *Transition diagram for the Reber grammar.* Figure 4: *Transition diagram for the embedded Reber grammar. Each box represents a copy of the Reber grammar (see Figure 3).*

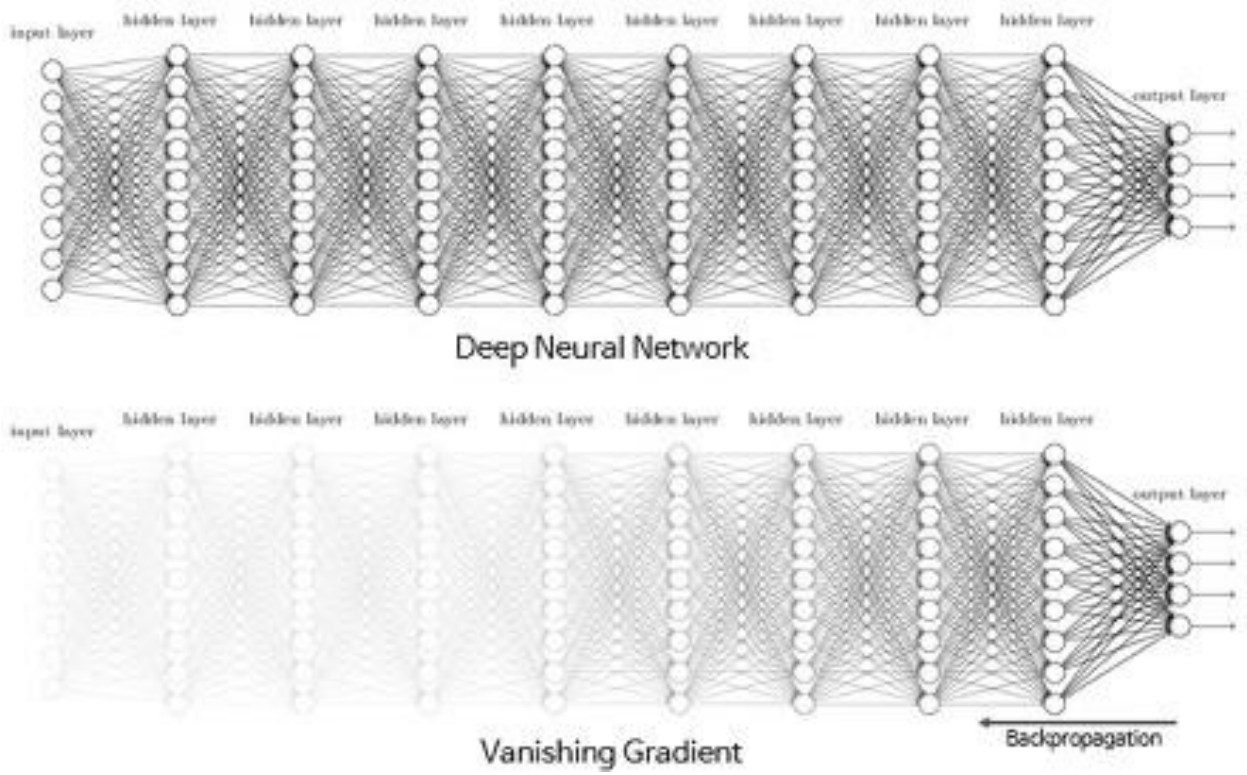
Starting at the leftmost node of the directed graph in Figure 4, symbol strings are generated sequentially (beginning with the empty string) by following edges — and appending the associated

11

Hochreiter & Schmidhuber(1997) :
https://github.com/dzitkowskik/StockPredictionRNN/blob/master/docs/Hochreiter97_lstm.pdf

Vanishing Gradient

- 긴 기간의 의미를 파악하지 못하고 짧은 기간 만이 유의미해지므로 기억력이 좋지 못한 모델이 된다.
- 이를 해결하지 등장한 것이 **LSTM(Long Short Term Memory Units)**이다.

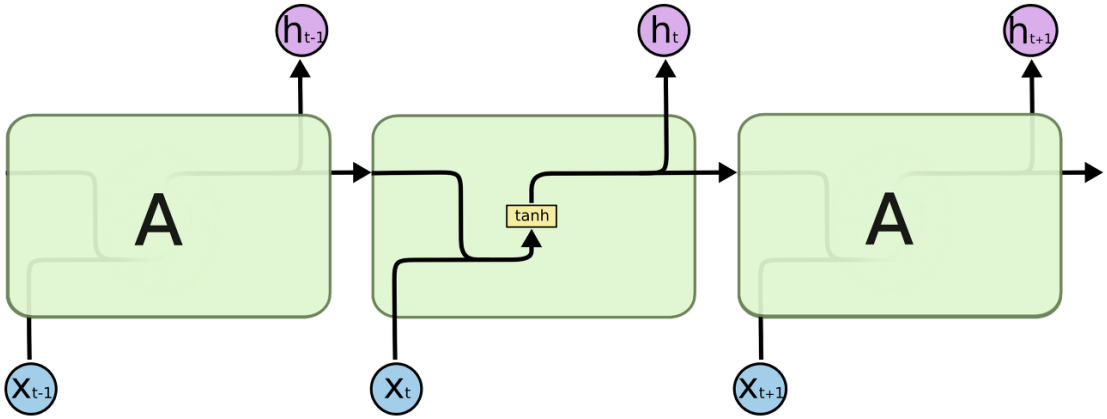


LSTM, Long Short Term Memory

- LSTM은 장기 의존성 문제를 해결하기 위해 명시적으로 디자인되었다.
- 오랜 기간 동안 정보를 기억하는 일은 LSTM에 있어 특별한 작업 없이도 기본적으로 동작
- 모든 RNN은 **뉴럴 네트워크의 반복되는 체인으로 구성**되어 있다.

  → **표준 RNN**에서 반복되는 모듈은 아주 단순한 구조를 가지고 있다.

  eg. 단일 tanh 레이어

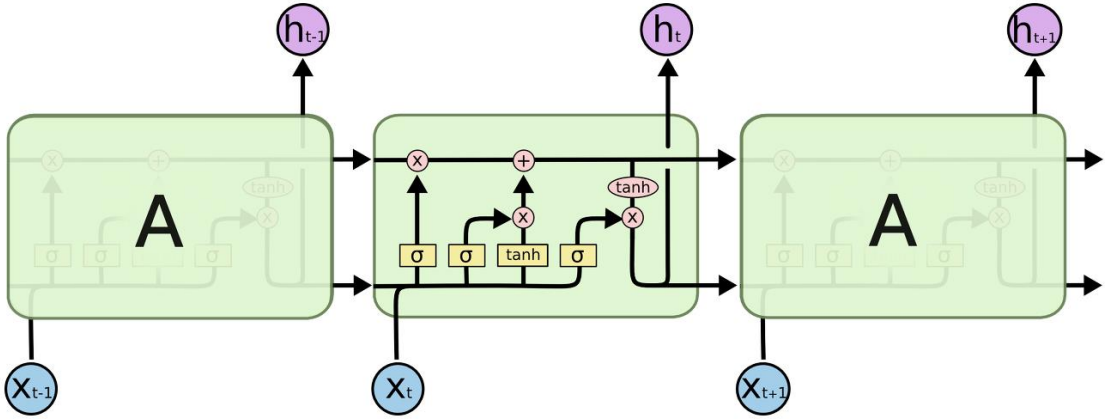


[싱글 레이어를 가지고 있는 반복되는 표준 RNN 모듈]

LSTM, Long Short Term Memory

- LSTM도 마찬가지로 체인구조를 가지고 있다. 하지만, 반복되는 모듈은 다른 구조이다.
- LSTM은 단일 뉴럴 네트워크 레이어를 가지는 것 대신에

  → **4개의 상호작용 가능한 특별한 방식의 구조**를 가지고 있다.
- 다이어그램에서 각 라인은 온전한 vector를 포함한다. 각 출력값은 다른 노드의 입력값.

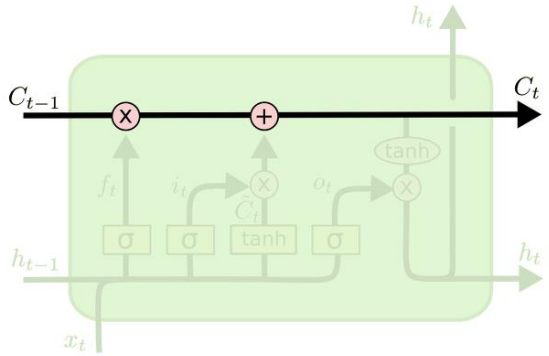


[LSTM 에 들어있는 4개의 상호작용하는 레이어가 있는 반복되는 모듈]

LSTMs의 핵심 아이디어

- LSTM의 핵심은 셀 스테이트(The Cell State)이다. 다이어그램 상단에 있는 수평선이다.
- 셀 스테이트는 하나의 컨베이어 벨트와 같다.

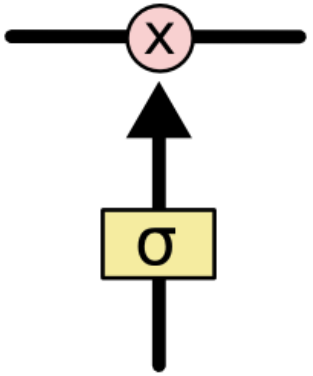
  → 이것은 아주 마이너한 선형 연산을 거치고 전체 체인을 관통한다.



[싱글 레이어를 가지고 있는 반복되는 표준 RNN 모듈]

LSTMs의 핵심 아이디어

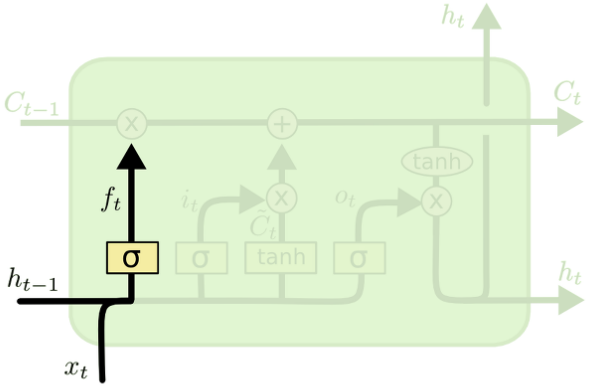
- LSTM은 셀 스테이트에 신중하게 정제된 구조를 가진 게이트(gate)라는 요소를 활용하여 정보를 더하거나 제거하는 기능을 가지고 있다.
- 게이트(Gates)들은 선택적으로 정보들이 흘러들어갈 수 있도록 만드는 장치이다.
  → 이들은 시그모이드 뉴럴넷(Sigmoid Neural Net Layer)와 점단위 곱하기 연산으로 이루어짐
- 시그모이드 레이어는 0 혹은 1의 값을 출력한다.
  → 0이라는 값을 가지면, 해당 구성요소가 미래의 결과에 아무 영향도 주지 않도록 만든다.
  → 반면 1이라는 값은 해당 구성요소가 미래 예측결과에 영향을 주도록 데이터가 흘러가게 만든다.



[시그모이드 레이어]

LSTM, STEP 1

◆ LSTM 첫번째 스텝 : 셀 스테이트에서 어떤 정보를 버릴지 선택하는 과정이다.

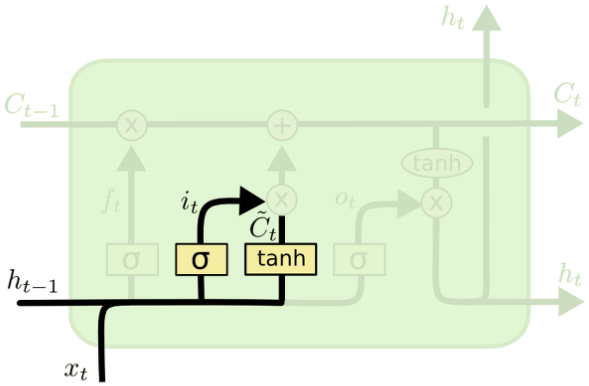


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

[LSTM – STEP1]

LSTM, STEP 2

◆ LSTM 두번째 스텝 : 새로운 정보가 셀 스테이트에 저장될지를 결정하는 단계이다.



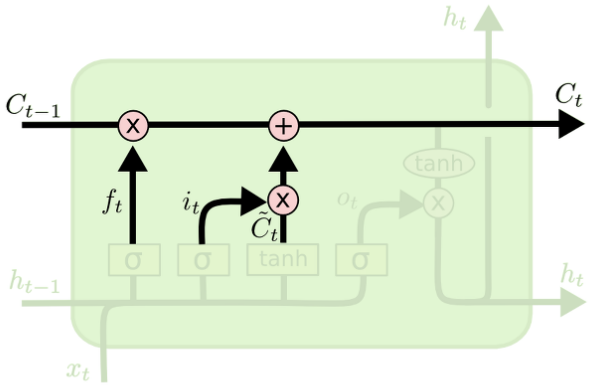
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

[LSTM – STEP2]

LSTM, STEP 3

- LSTM 세번째 스텝 : 오래된 셀 스테이트(Ct-1)를 새로운 스테이트인(Ct)로 업데이트 할 시간이다.
- 전 단계에서 무엇을 할지 이미 결정했다. 그래서, 이 연산을 수행하기만 하면 된다.

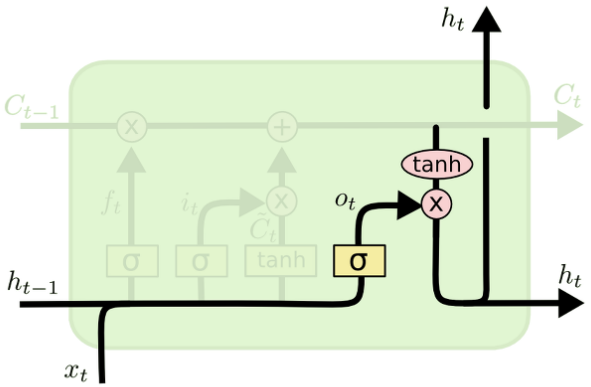


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

[LSTM – STEP3]

LSTM, STEP 4

- LSTM 네번째 스텝 : 어떤 출력값을 출력할지 결정해야 한다.
- 이 출력값은 우리의 세 스테이트(cell state) 하지만 필터링 된 버전이다.



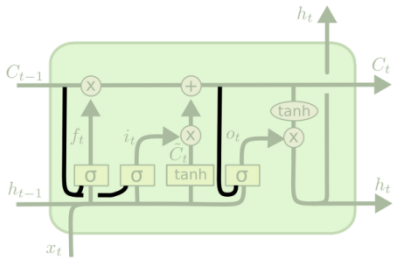
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

[LSTM – STEP4]

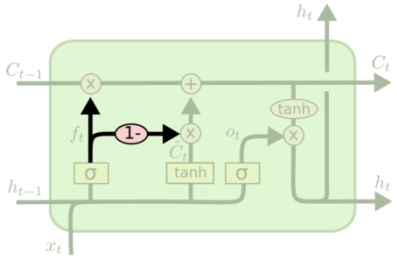
LSTM의 변칙 패턴

- 모든 LSTM 이 위와 같은 동일한 구조를 갖고 있지는 않다.
- 사실 LSTM 을 구현한 모든 논문들은 서로서로 약간 다른 구현체 버전을 가지고 있다.
- 그 차이들은 사실 그리 중요하지 않다.



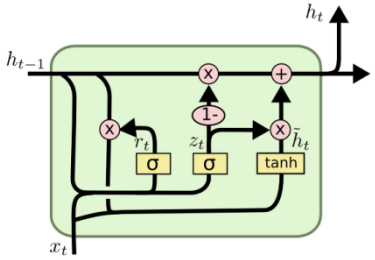
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

[LSTM – 변칙1]



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

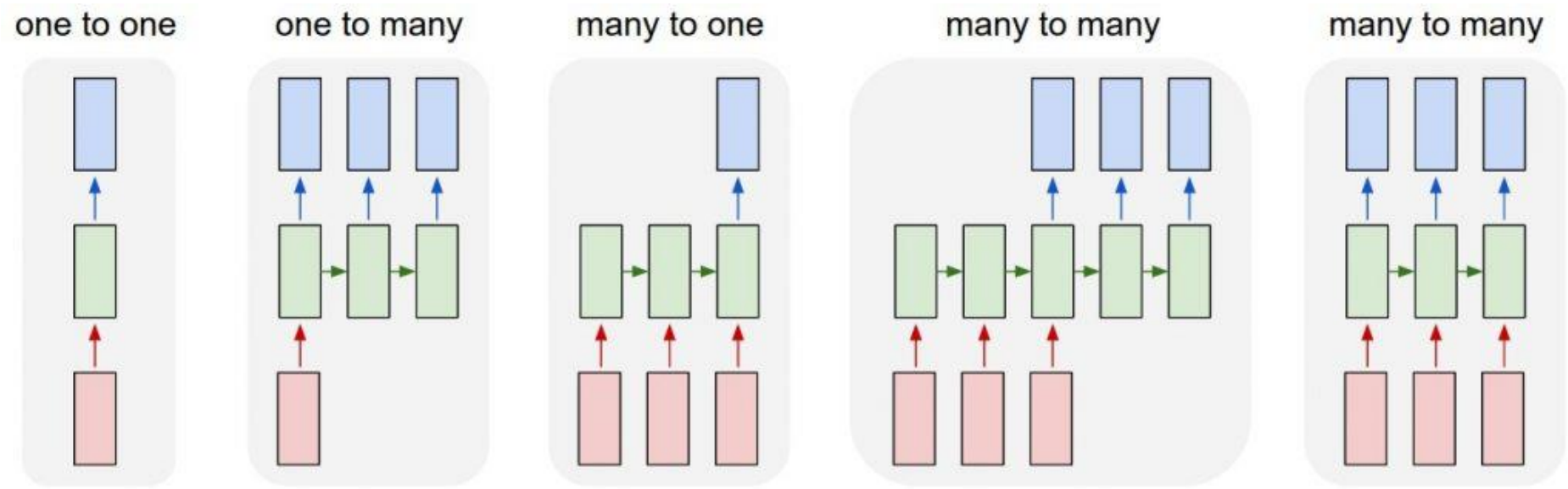
[LSTM – 변칙2]



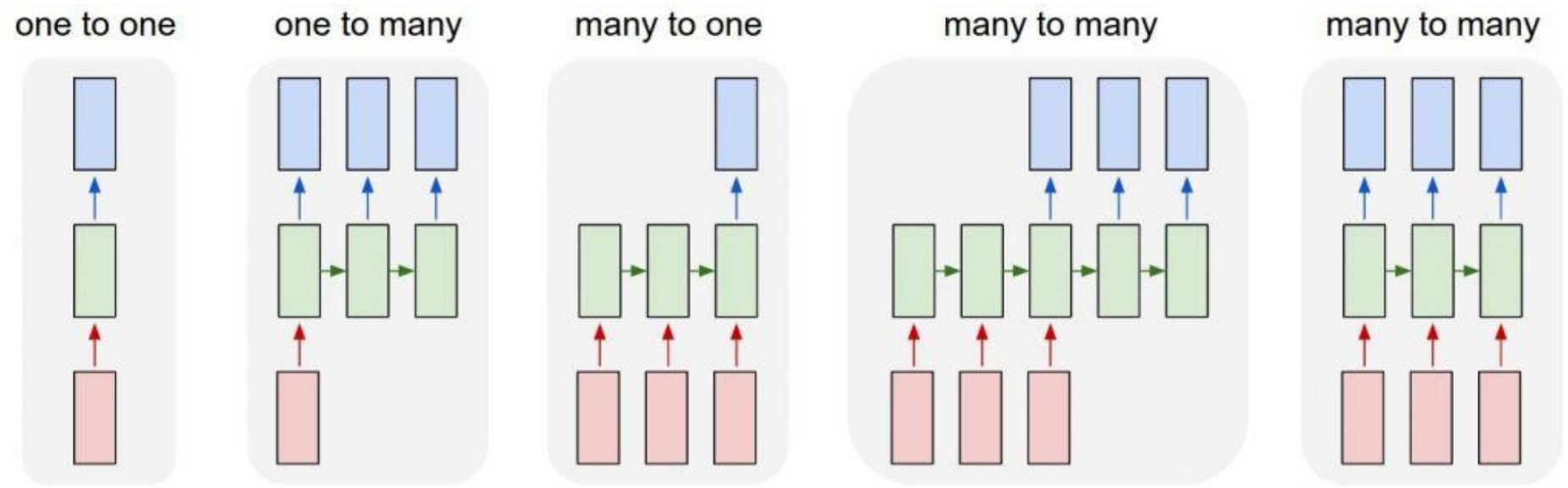
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

[LSTM – 변칙3]

RNN offer a lot of flexibility



RNN Architecture



Vanilla Neural Networks

RNN Architecture

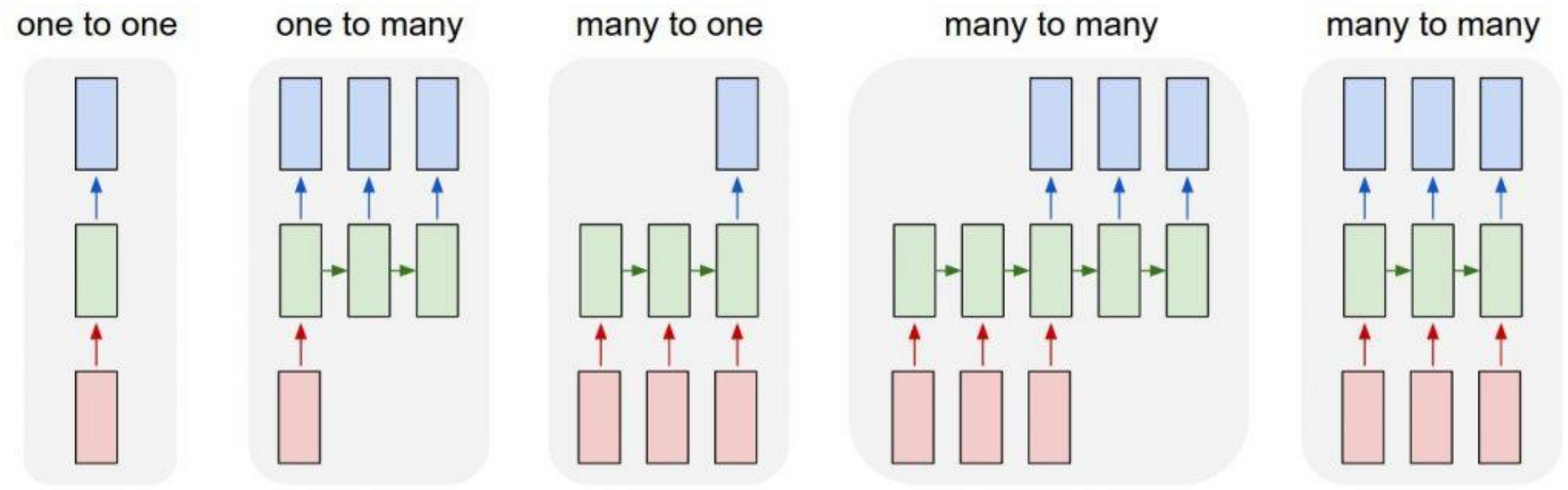
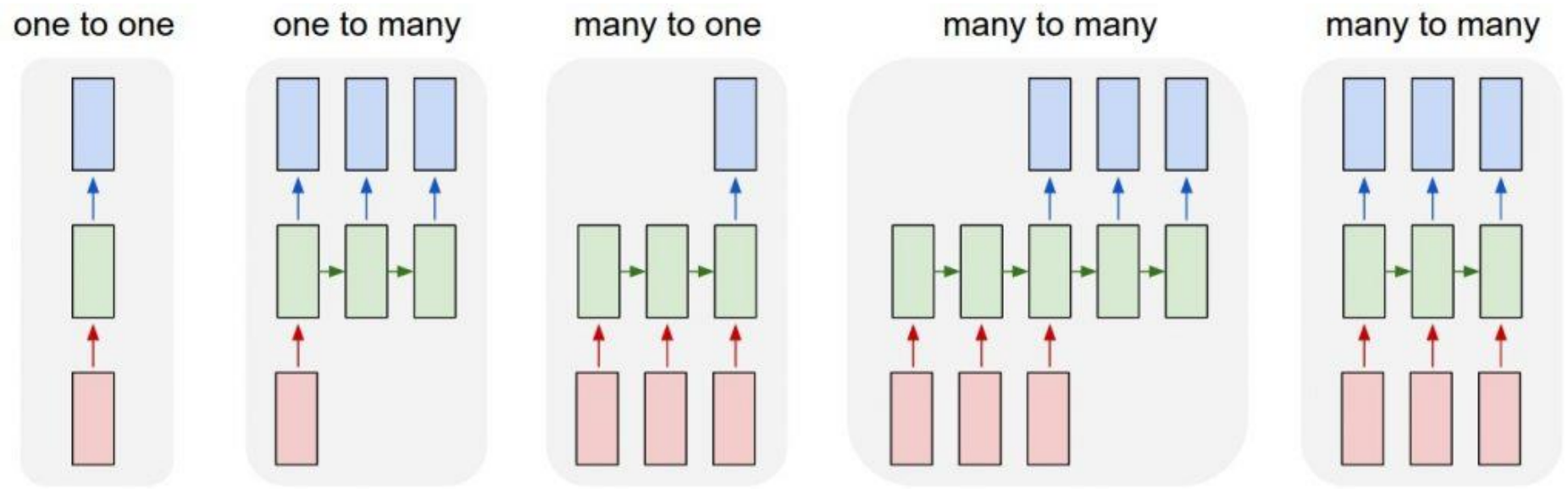


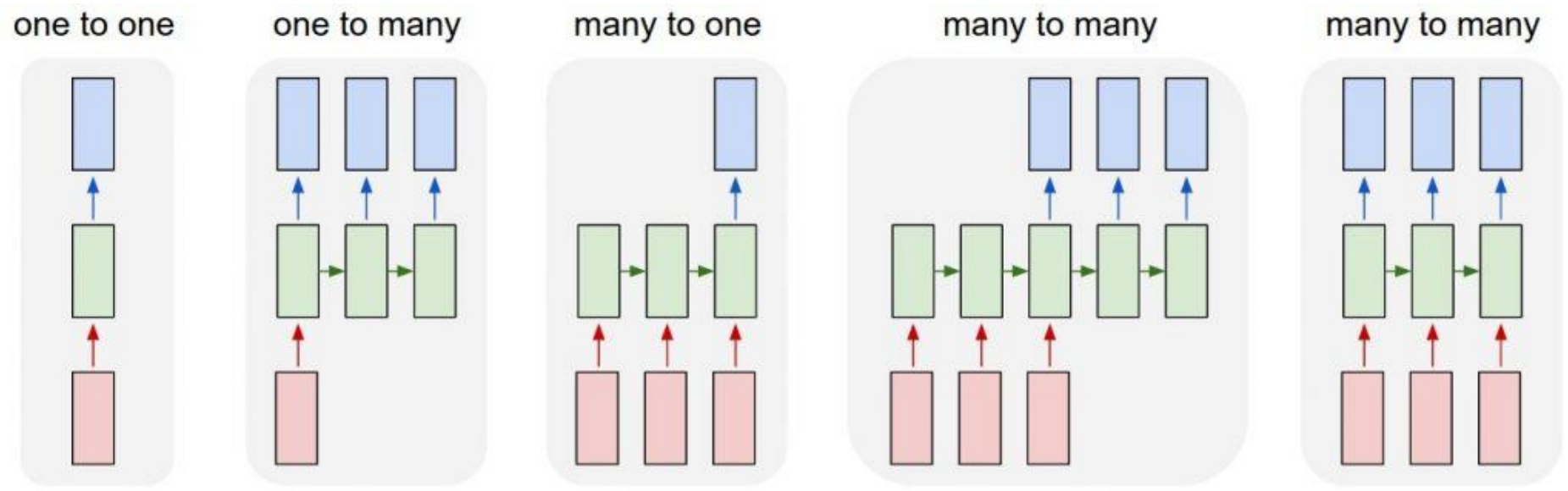
Image Captioning
Image → sequence of words

RNN Architecture



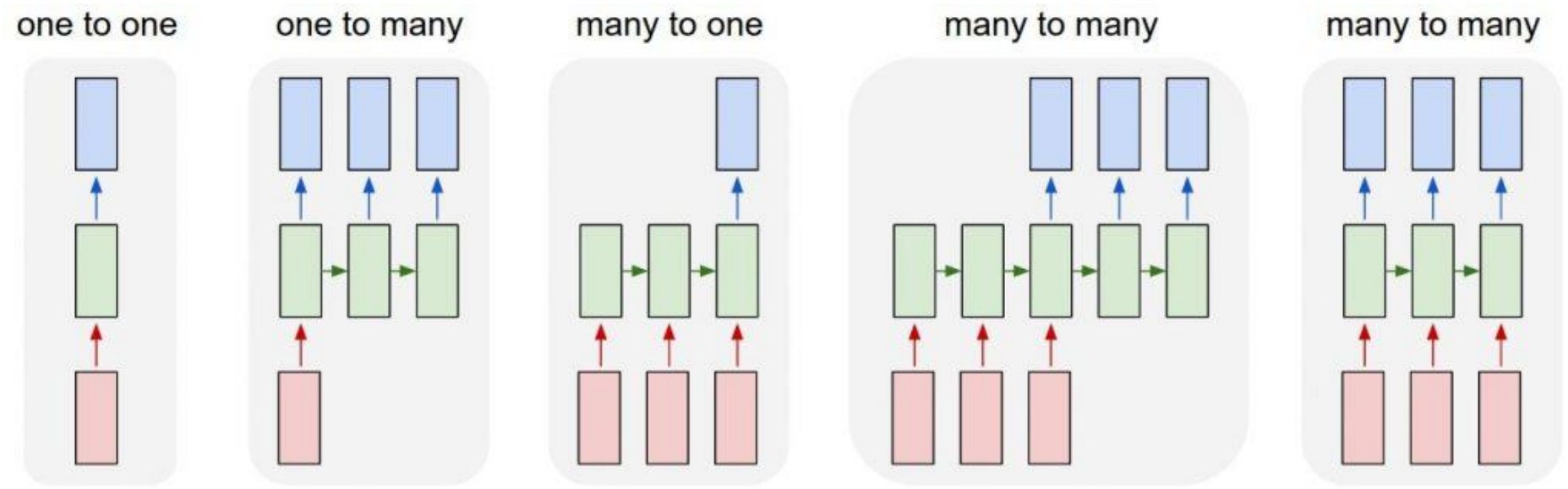
Sentiment Classification
sequence of words → sentiment

RNN Architecture



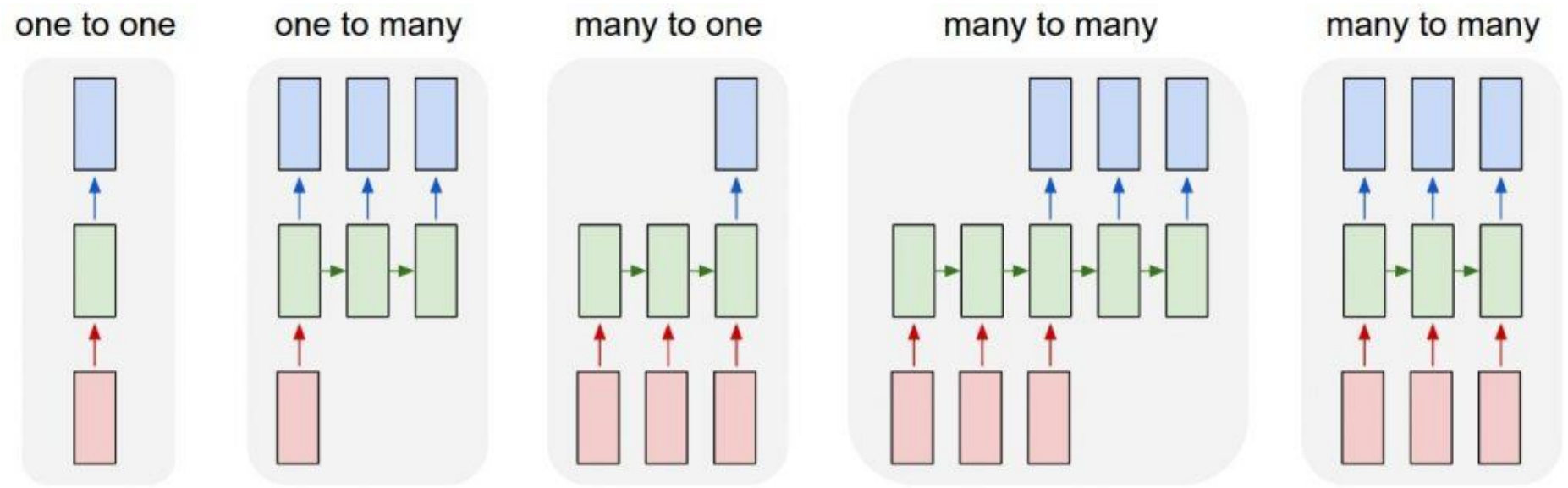
Machine Translation
sequence of words → sequence of words

RNN Architecture



Video classification on frame level

RNN Architecture



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

RNN의 응용

Text Generation

이전 단어들을 보고 다음 단어가 나올 확률을 예측하는 언어 모델을 기반으로 텍스트를 생성하는 generative model을 만들 수 있음

Language Translation

Word sequence를 입력으로 사용하여 번역할 언어의 word sequence로 출력

Speech Recognition

Acoustic signal을 입력으로 받아 phonetic segment의 sequence 또는 probabillity distribution을 추측

Summary

- 일찍이 RNN을 활용하여 엄청난 성과를 거두고 있다는 사실을 언급했다.
- 핵심적으로 모든 이러한 것들은 LSTM을 활용한 결과였다.
- LSTM은 거의 모든 영역에서 다른 RNN 알고리즘에 비해 탁월한 성능을 보여주고 있다.

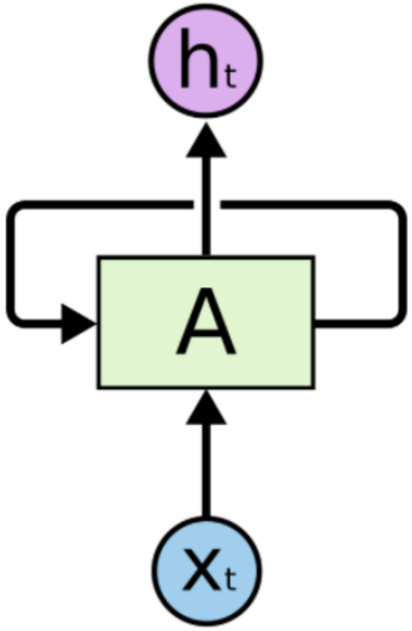
Appendix

RNN Practice : Source Code

Practice

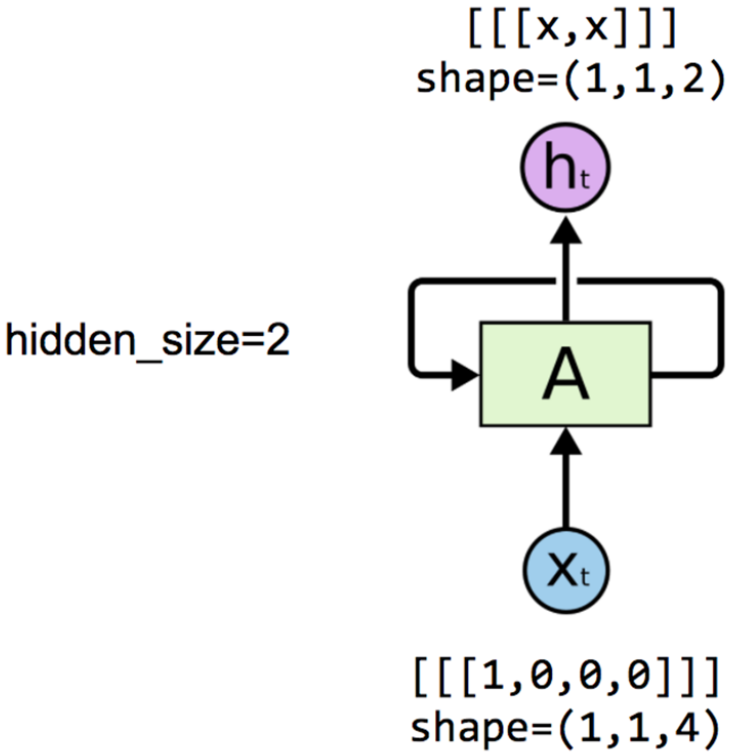
1. RNN String Sequence

RNN in TensorFlow




```
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)

...

outputs, _states = tf.nn.dynamic_rnn(cell, x_data, dtype=tf.float32)
```

One node: 4 (*input-dim*) in 2 (*hidden_size*)

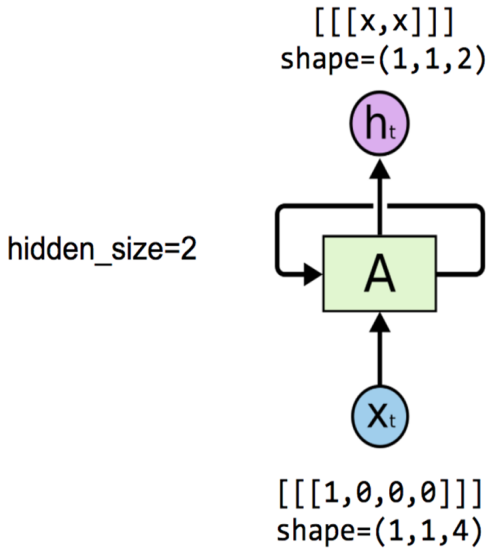



```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

One node: 4 (*input-dim*) in 2 (*hidden_size*)

```
# One cell RNN input_dim (4) -> output_dim (2)
hidden_size = 2
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)

x_data = np.array([[[1,0,0,0]]], dtype=np.float32)
outputs, _states = tf.nn.dynamic_rnn(cell, x_data, dtype=tf.float32)

sess.run(tf.global_variables_initializer())
pp.pprint(outputs.eval())
```

```
array([[[-0.42409304,  0.64651132]]])
```

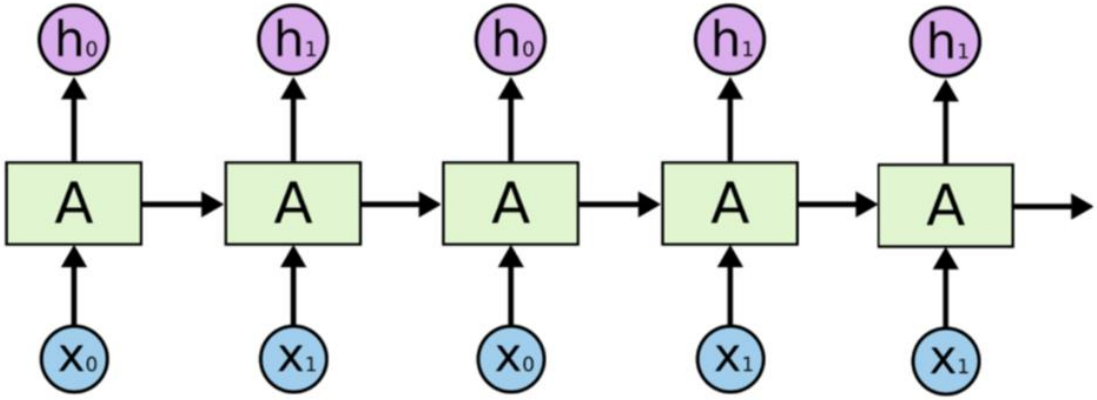



```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

Unfolding to n sequences

```
Hidden_size=2
sequence_length=5
```

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

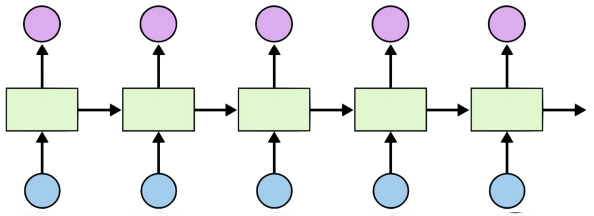
Unfolding to n sequences

```
# One cell RNN input_dim (4) -> output_dim (2). sequence: 5
    hidden_size = 2
    cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size)
    x_data = np.array([[h, e, l, l, o]], dtype=np.float32)
    print(x_data.shape)
    pp.pprint(x_data)
    outputs, states = tf.nn.dynamic_rnn(cell, x_data, dtype=tf.float32)
    sess.run(tf.global_variables_initializer())
    pp.pprint(outputs.eval())
```

```
# 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
sequence_length=5

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



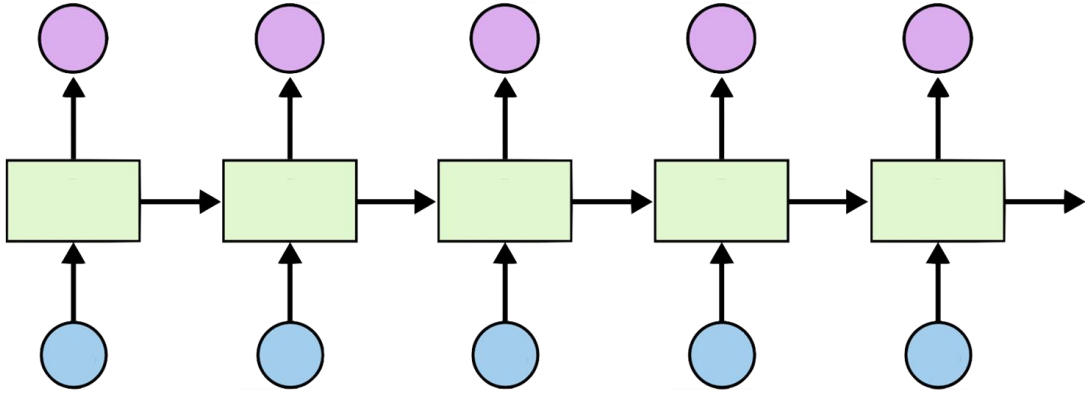
shape=(1,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]]]
h e l l o

```
X_data = array
   ([[[ 1.,  0.,  0.,  0.],
      [ 0.,  1.,  0.,  0.],
      [ 0.,  0.,  1.,  0.],
      [ 0.,  0.,  1.,  0.],
      [ 0.,  0.,  0.,  1.]]], dtype=float32)
Outputs = array
   ([[[ 0.19709368,  0.24918222],
      [-0.11721198,  0.1784237 ],
      [-0.35297349, -0.66278851],
      [-0.70915914, -0.58334434],
      [-0.38886023,  0.47304463]]], dtype=float32)
```

Batching input

```
Hidden_size=2
sequence_length=5
batch_size=3
```

```
shape=(3,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]]]
```



```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

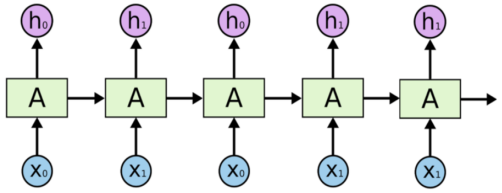
Batching input

```
# One cell RNN input_dim (4) -> output_dim (2). sequence: 5, batch 3
# 3 batches 'hello', 'eolll', 'lleel'
x_data = np.array([[h, e, l, l, o],
                   [e, o, l, l, l],
                   [l, l, e, e, l]], dtype=np.float32)
pp.pprint(x_data)

cell = rnn.BasicLSTMCell(num_units=2, state_is_tuple=True)
outputs, _states = tf.nn.dynamic_rnn(cell, x_data,
                                     dtype=tf.float32)
sess.run(tf.global_variables_initializer())
pp.pprint(outputs.eval())
```

```
array([[[ 1.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.]],

       [[ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  1.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.]],

       [[ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.]]],
```

hidden_size=2
sequacne_length=5
batch = 3

```
shape=(3,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]]]
```




```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

Hidden_size=2
sequence_length=5
batch_size=3

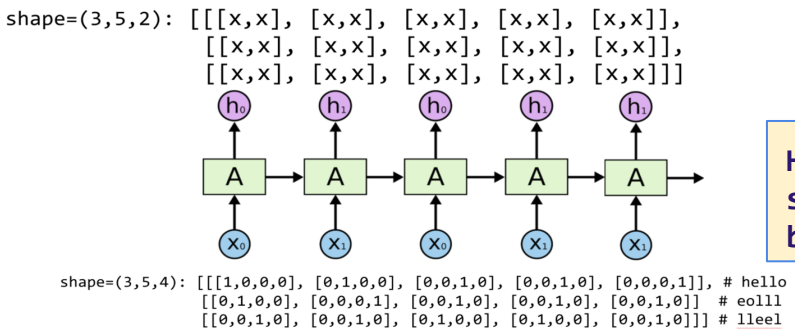
Unfolding to n sequences

```
# One cell RNN input_dim (4) -> output_dim (2). sequence: 5, batch 3
# 3 batches 'hello', 'eolll', 'lleel'
x_data = np.array([[h, e, l, l, o],
                   [e, o, l, l, l],
                   [l, l, e, e, l]], dtype=np.float32)
pp.pprint(x_data)

cell = rnn.BasicLSTMCell(num_units=2, state_is_tuple=True)
outputs, _states = tf.nn.dynamic_rnn(cell, x_data,
                                     dtype=tf.float32)
sess.run(tf.global_variables_initializer())
pp.pprint(outputs.eval())
```

```
array([[[ 1.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.]],

       [[ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  1.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.]],

       [[ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.]]],
```

```
array([[[-0.0173022 , -0.12929453],
        [-0.14995177, -0.23189341],
        [ 0.03294011,  0.01962204],
        [ 0.12852104,  0.12375218],
        [ 0.13597946,  0.31746736]],

       [[-0.15243632, -0.14177315],
        [ 0.04586344,  0.12249056],
        [ 0.14292534,  0.15872268],
        [ 0.18998367,  0.21004884],
        [ 0.21788891,  0.24151592]],

       [[ 0.10713603,  0.11001928],
        [ 0.17076059,  0.1799853 ],
        [-0.03531617,  0.08993293],
        [-0.1881337 , -0.08296411],
        [-0.00404597,  0.07156041]]],
```

hidden_size=2
sequacne_length=5
batch = 3



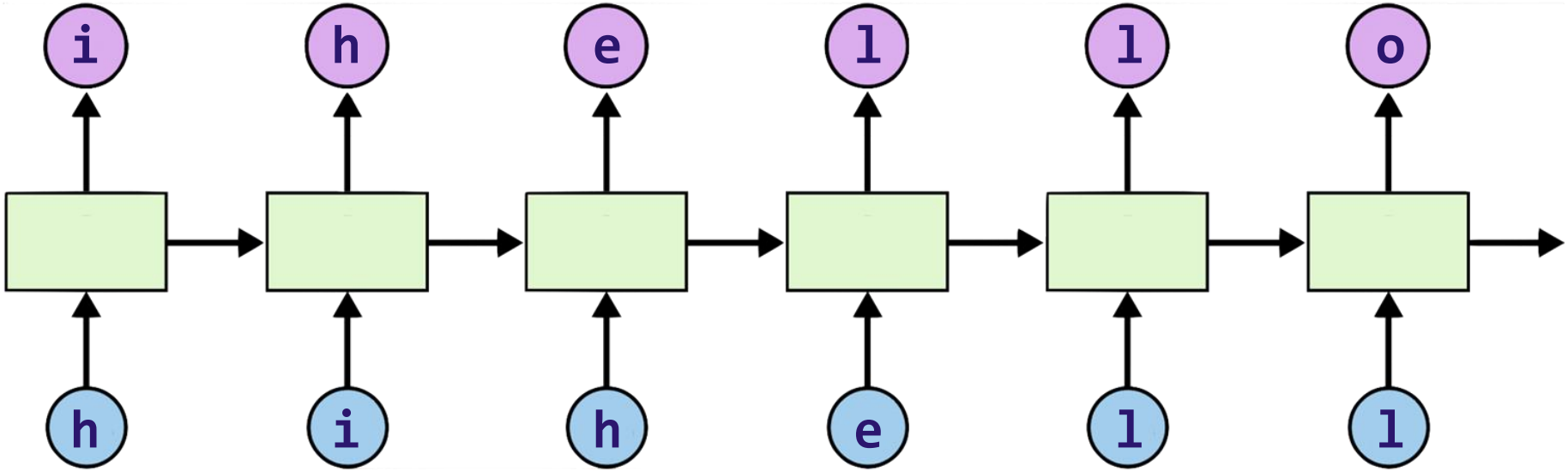

```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

Hidden_size=2
sequence_length=5
batch_size=3

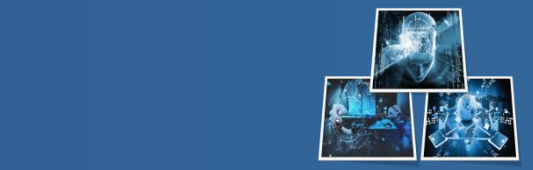
Practice

2. RNN Sentence Sequence

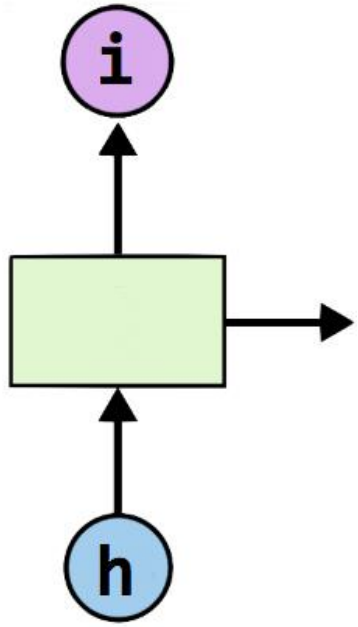
Teach RNN ‘hihello’



One-hot encoding

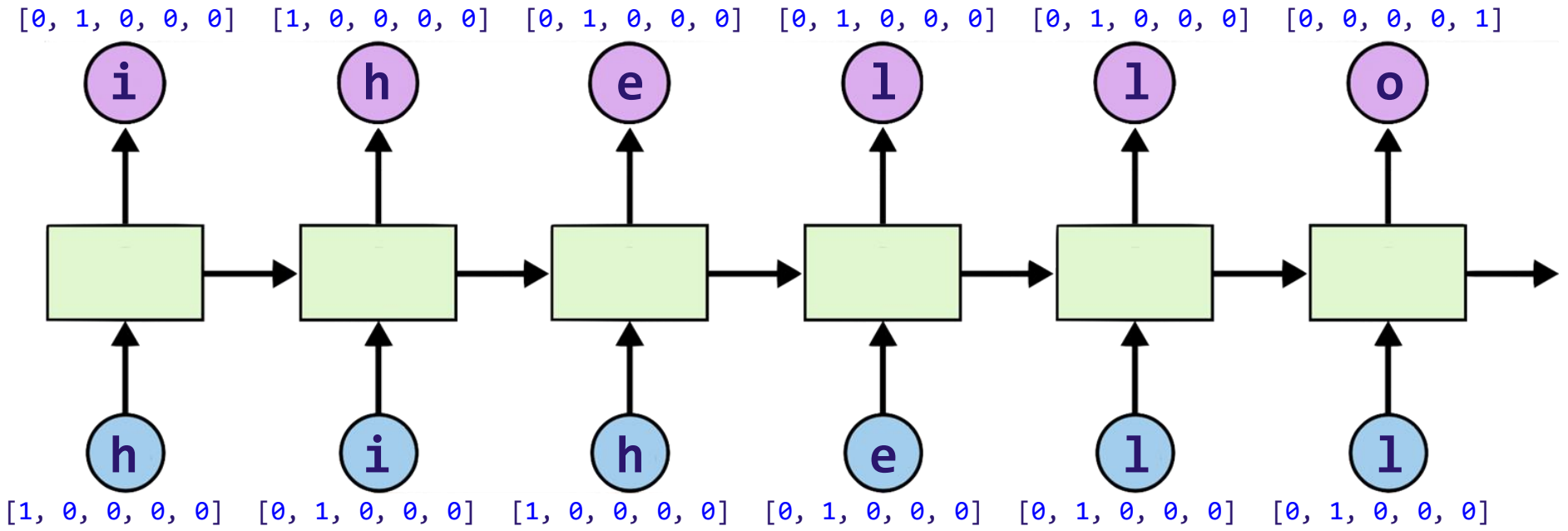


```
[1, 0, 0, 0, 0],    # h 0
[0, 1, 0, 0, 0],    # i 1
[0, 0, 1, 0, 0],    # e 2
[0, 0, 0, 1, 0],    # l 3
[0, 0, 0, 0, 1],    # o 4
```



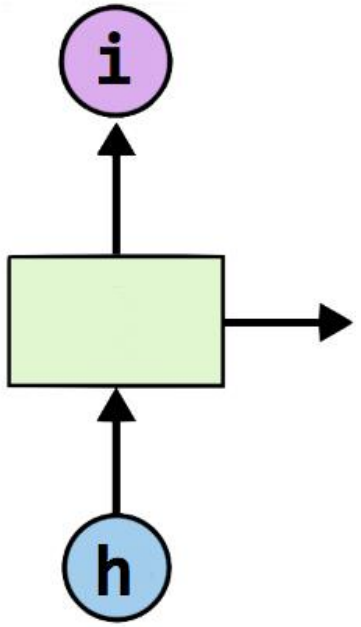
Teach RNN 'hihello'

```
[1, 0, 0, 0, 0],   # h 0
[0, 1, 0, 0, 0],   # i 1
[0, 0, 1, 0, 0],   # e 2
[0, 0, 0, 1, 0],   # l 3
[0, 0, 0, 0, 1],   # o 4
```



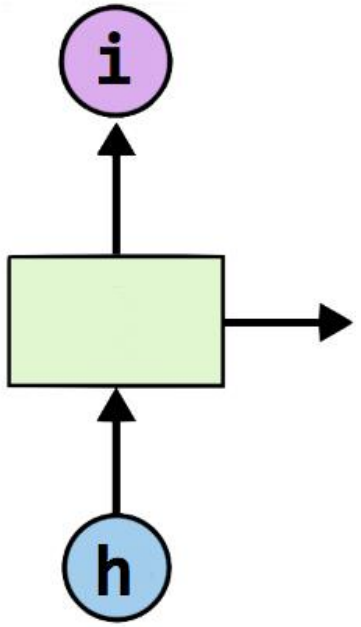
Creating RNN Cell

```
# RNN model
rnn_cell = rnn_cell.BasicRNNCell(rnn_size)
```



Creating RNN Cell

```
# RNN model
rnn_cell = rnn_cell.BasicRNNCell(rnn_size)

rnn_cell = rnn_cell. BasicLSTMCell(rnn_size)
rnn_cell = rnn_cell. GRUCell(rnn_size)
```

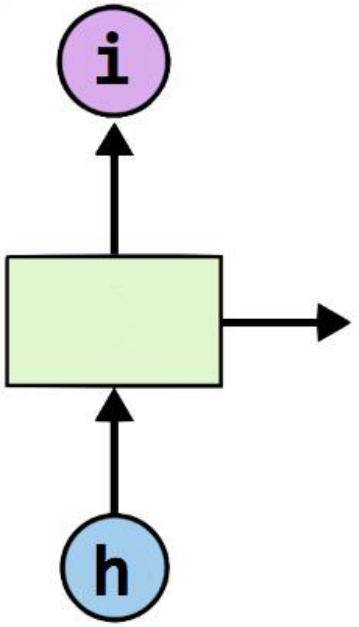


Execute RNN

```
# RNN model
rnn_cell = rnn_cell.BasicRNNCell(rnn_size)

outputs, _states = tf.nn.dynamic_rnn(
                   rnn_cell,
                   X,
                   initial_state=initial_state,
                   dtype=tf.float32)
```

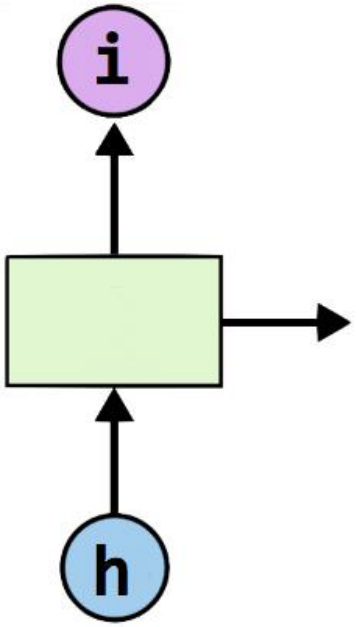
1

2



RNN Parameters

```
hidden_size = 5        # output from the LSTM
input_dim = 5          # one-hot size
batch_size = 1         # one sentence
sequence_length = 6    # |ihello| == 6
```



Data Creation

```
idx2char = ['h', 'i', 'e', 'l', 'o'] # h=0, i=1, e=2, l=3, o=4
x_data = [[0, 1, 0, 2, 3, 3]]        # hihell
x_one_hot = [[[1, 0, 0, 0, 0],       # h 0
              [0, 1, 0, 0, 0],       # i 1
              [1, 0, 0, 0, 0],       # h 0
              [0, 0, 1, 0, 0],       # e 2
              [0, 0, 0, 1, 0],       # l 3
              [0, 0, 0, 1, 0]]]      # l 3

y_data = [[1, 0, 2, 3, 3, 4]]        # ihello
X = tf.placeholder(tf.float32,
        [None, sequence_length, hidden_size]) # X one-hot
Y = tf.placeholder(tf.int32, [None, sequence_length])  # Y label
```

Feed to RNN

```
X = tf.placeholder(
    tf.float32, [None, sequence_length, hidden_size])  # X one-hot
Y = tf.placeholder(tf.int32, [None, sequence_length])  # Y label

cell = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size,
state_is_tuple=True)
initial_state = cell.zero_state(batch_size, tf.float32)
outputs, _states = tf.nn.dynamic_rnn(
    cell, X, initial_state=initial_state, dtype=tf.float32)
```

```
x_one_hot = [[[1, 0, 0, 0, 0],   # h 0
              [0, 1, 0, 0, 0],   # i 1
              [1, 0, 0, 0, 0],   # h 0
              [0, 0, 1, 0, 0],   # e 2
              [0, 0, 0, 1, 0],   # l 3
              [0, 0, 0, 1, 0]]]  # l 3

y_data = [[1, 0, 2, 3, 3, 4]]    # ihello
```

Cost : sequence_loss

```
# [batch_size, sequence_length]
y_data = tf.constant([[1, 1, 1]])

# [batch_size, sequence_length, emb_dim ]
prediction = tf.constant([[[0, 1], [1, 0], [0, 1]]], dtype=tf.float32)

# [batch_size * sequence_length]
weights = tf.constant([[1, 1, 1]], dtype=tf.float32)

sequence_loss = tf.contrib.seq2seq.sequence_loss(prediction, y_data, weights)
sess.run(tf.global_variables_initializer())
print("Loss: ", sequence_loss.eval())
```

```
Loss:  0.646595
```

Cost : sequence_loss

```
# [batch_size, sequence_length]
y_data = tf.constant([[1, 1, 1]])

# [batch_size, sequence_length, emb_dim ]
prediction1 = tf.constant([[[0, 1], [0, 1], [0, 1]]], dtype=tf.float32)
prediction2 = tf.constant([[[1, 0], [1, 0], [1, 0]]], dtype=tf.float32)
prediction3 = tf.constant([[[0, 1], [1, 0], [0, 1]]], dtype=tf.float32)

# [batch_size * sequence_length]
weights = tf.constant([[1, 1, 1]], dtype=tf.float32)

sequence_loss1 = tf.contrib.seq2seq.sequence_loss(prediction1, y_data, weights)
sequence_loss2 = tf.contrib.seq2seq.sequence_loss(prediction2, y_data, weights)
sequence_loss3 = tf.contrib.seq2seq.sequence_loss(prediction3, y_data, weights)

sess.run(tf.global_variables_initializer())
print("Loss1: ", sequence_loss1.eval(),
      "Loss2: ", sequence_loss2.eval(),
      "Loss3: ", sequence_loss3.eval())
```

```
Loss1:  0.313262 Loss2:  1.31326 Loss3:  0.646595
```

Cost : sequence_loss

```
outputs, _states = tf.nn.dynamic_rnn(
   cell, X, initial_state=initial_state, dtype=tf.float32)
weights = tf.ones([batch_size, sequence_length])

sequence_loss = tf.contrib.seq2seq.sequence_loss(
                        logits=outputs, targets=Y, weights=weights)
loss = tf.reduce_mean(sequence_loss)
train = tf.train.GradientDescentOptimizer(learning_rate=0.1).minimize(loss)
```

Training

```
prediction = tf.argmax(outputs, axis=2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(2000):
        l, _ = sess.run([loss, train], feed_dict={X: x_one_hot, Y: y_data})
        result = sess.run(prediction, feed_dict={X: x_one_hot})
        print(i, "loss:", l, "prediction: ", result, "true Y: ", y_data)

        # print char using dic
        result_str = [idx2char[c] for c in np.squeeze(result)]
        print("\tPrediction str: ", ''.join(result_str))
```

Results

```
prediction = tf.argmax(outputs, axis=2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(2000):
        l, _ = sess.run([loss, train], feed_dict={X: x_one_hot, Y: y_data})
        result = sess.run(prediction, feed_dict={X: x_one_hot})
        print(i, "loss:", l, "prediction: ", result, "true Y: ", y_data)

        # print char using dic
        result_str = [idx2char[c] for c in np.squeeze(result)]
        print("\tPrediction str: ", ''.join(result_str))
```

```
0 loss: 1.55474 prediction:  [[3 3 3 3 4 4]] true Y:  [[1, 0, 2, 3, 3, 4]] Prediction str:  llllloo
1 loss: 1.55081 prediction:  [[3 3 3 3 4 4]] true Y:  [[1, 0, 2, 3, 3, 4]] Prediction str:  llllloo
2 loss: 1.54704 prediction:  [[3 3 3 3 4 4]] true Y:  [[1, 0, 2, 3, 3, 4]] Prediction str:  llllloo
3 loss: 1.54342 prediction:  [[3 3 3 3 4 4]] true Y:  [[1, 0, 2, 3, 3, 4]] Prediction str:  llllloo
...
1998 loss: 0.75305 prediction:  [[1 0 2 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]] Prediction str:  ihello
1999 loss: 0.752973 prediction:  [[1 0 2 3 3 4]] true Y:  [[1, 0, 2, 3, 3, 4]] Prediction str:  ihell
o
```

Practice

3. Only Softmax Sentence Sequence

Manual data creation

```
idx2char = ['h', 'i', 'e', 'l', 'o']
x_data   = [[0, 1, 0, 2, 3, 3]]    # hihell
x_one_hot = [[[1, 0, 0, 0, 0],     # h 0
              [0, 1, 0, 0, 0],     # i 1
              [1, 0, 0, 0, 0],     # h 0
              [0, 0, 1, 0, 0],     # e 2
              [0, 0, 0, 1, 0],     # l 3
              [0, 0, 0, 1, 0]]]    # l 3

y_data = [[1, 0, 2, 3, 3, 4]]    # ihello
```

Better data creation

```
sample = " if you want you"
idx2char = list(set(sample))  # index -> char
char2idx = {c: i for i, c in enumerate(idx2char)}  # char -> idx

sample_idx = [char2idx[c] for c in sample]  # char to index
x_data = [sample_idx[:-1]]  # X data sample (0 ~ n-1) hello: hell
y_data = [sample_idx[1:]]   # Y label sample (1 ~ n)  hello: ello

X = tf.placeholder(tf.int32, [None, sequence_length])  # X data
Y = tf.placeholder(tf.int32, [None, sequence_length])  # Y label

X_one_hot = tf.one_hot(X, num_classes)  # one hot: 1 -> 0 1 0 0 0 0 0 0 0 0
```

Hyper parameters

```
sample = " if you want you"
idx2char = list(set(sample))  # index -> char
char2idx = {c: i for i, c in enumerate(idx2char)}  # char -> idx

# hyper parameters
dic_size = len(char2idx)  # RNN input size (one hot size)
rnn_hidden_size = len(char2idx)  # RNN output size
num_classes = len(char2idx)  # final output size (RNN or softmax, etc.)
batch_size = 1  # one sample data, one batch
sequence_length = len(sample) - 1  # number of lstm unfolding (unit #)
```

LSTM and Loss

```
X = tf.placeholder(tf.int32, [None, sequence_length])  # X data
Y = tf.placeholder(tf.int32, [None, sequence_length])  # Y label

X_one_hot = tf.one_hot(X, num_classes)  # one hot: 1 -> 0 1 0 0 0 0 0 0 0 0

cell = tf.contrib.rnn.BasicLSTMCell(num_units=rnn_hidden_size, state_is_tuple=True)
initial_state = cell.zero_state(batch_size, tf.float32)
outputs, _states = tf.nn.dynamic_rnn(
   cell, X_one_hot, initial_state=initial_state, dtype=tf.float32)

weights = tf.ones([batch_size, sequence_length])
sequence_loss = tf.contrib.seq2seq.sequence_loss(logits=outputs,
targets=Y,weights=weights)
loss = tf.reduce_mean(sequence_loss)
train = tf.train.GradientDescentOptimizer(learning_rate=0.1).minimize(loss)

prediction = tf.argmax(outputs, axis=2)
```

Training and Results

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(3000):
        l, _ = sess.run([loss, train], feed_dict={X: x_data, Y: y_data})
        result = sess.run(prediction, feed_dict={X: x_data})
        # print char using dic
        result_str = [idx2char[c] for c in np.squeeze(result)]
        print(i, "loss:", l, "Prediction:", ''.join(result_str))
```

```
0 loss: 2.29895 Prediction: nnuffuunnuuuyuy
1 loss: 2.29675 Prediction: nnuffuunnuuuyuy
...
1418 loss: 1.37351 Prediction: if you want you
1419 loss: 1.37331 Prediction: if you want you
```

Really long sentence?

```
sentence = ("if you want to build a ship, don't drum up people together to "
            "collect wood and don't assign them tasks and work, but rather "
            "teach them to long for the endless immensity of the sea.")
```

Really long sentence?

```
sentence = ("if you want to build a ship, don't drum up people together to "
            "collect wood and don't assign them tasks and work, but rather "
            "teach them to long for the endless immensity of the sea.")
```

```
# training dataset
0 if you wan -> f you want
1 f you want ->  you want
2  you want  -> you want t
3 you want t -> ou want to
…
168  of the se -> of the sea
169 of the sea -> f the sea.
```

Making Dataset

```
char_set = list(set(sentence))
char_dic = {w: i for i, w in enumerate(char_set)}

dataX = []
dataY = []
for i in range(0, len(sentence) - seq_length):
    x_str = sentence[i:i + seq_length]
    y_str = sentence[i + 1: i + seq_length + 1]
    print(i, x_str, '->', y_str)

    x = [char_dic[c] for c in x_str]  # x str to index
    y = [char_dic[c] for c in y_str]  # y str to index

    dataX.append(x)
    dataY.append(y)
```

 # training dataset

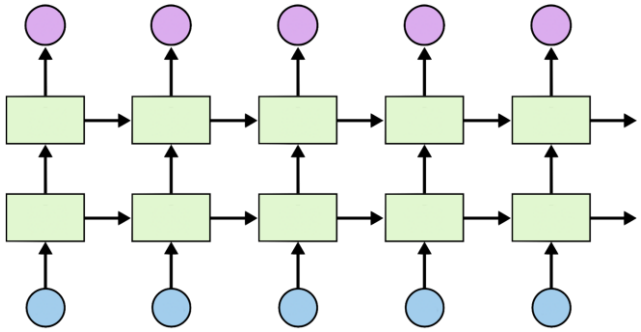
0 if you wan -> f you want
1 f you want -> you want
2 you want -> you want t
3 you want t -> ou want to
…
168 of the se -> of the sea
169 of the sea -> f the sea.

RNN parameters

```
char_set = list(set(sentence))
char_dic = {w: i for i, w in enumerate(char_set)}

data_dim = len(char_set)
hidden_size = len(char_set)
num_classes = len(char_set)
seq_length = 10  # Any arbitrary number

batch_size = len(dataX)
```

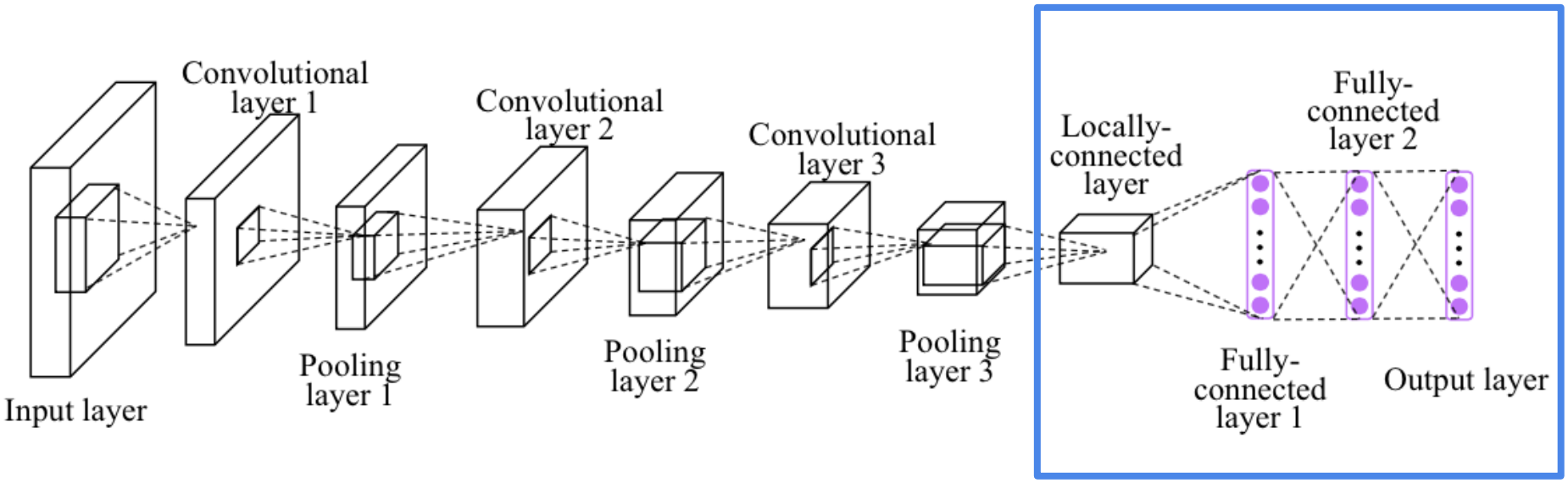
 # training dataset
0 if you wan -> f you want
1 f you want -> you want
2 you want -> you want t
3 you want t -> ou want to
…
168 of the se -> of the sea
169 of the sea -> f the sea.

Stacked RNN

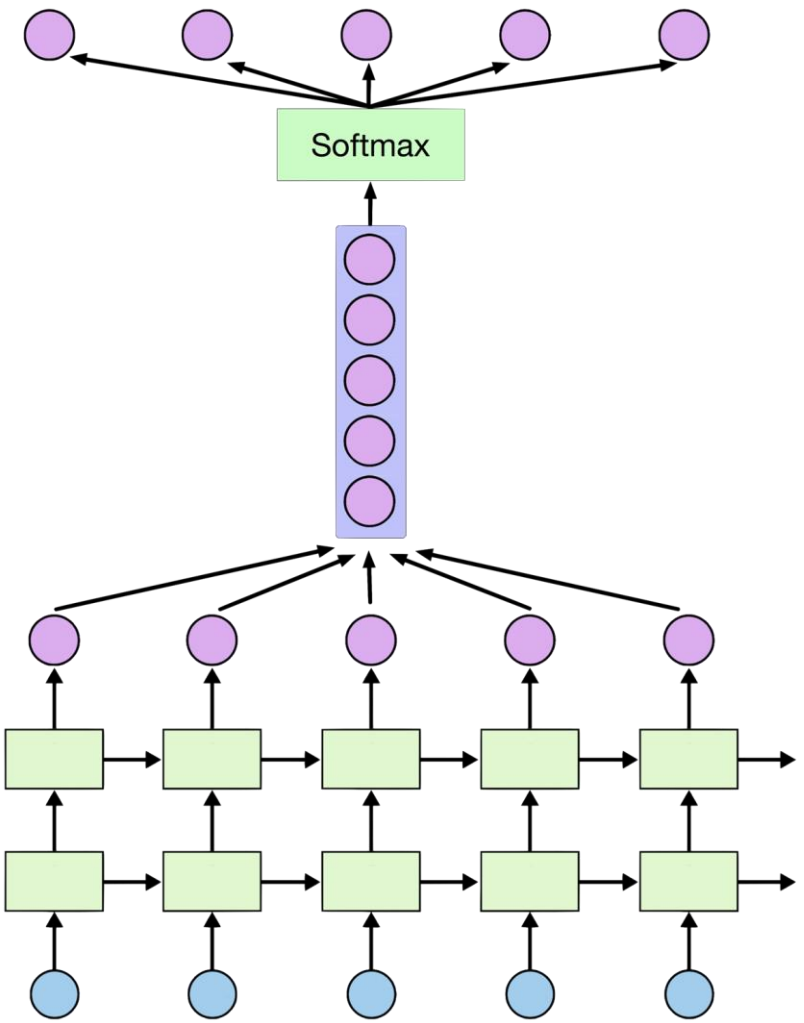


```
X = tf.placeholder(tf.int32, [None, seq_length])
Y = tf.placeholder(tf.int32, [None, seq_length])

# One-hot encoding
X_one_hot = tf.one_hot(X, num_classes)
print(X_one_hot)  # check out the shape

# Make a lstm cell with hidden_size (each unit output vector size)
cell = rnn.BasicLSTMCell(hidden_size, state_is_tuple=True)
cell = rnn.MultiRNNCell([cell] * 2, state_is_tuple=True)

# outputs: unfolding size x hidden size, state = hidden size
outputs, _states = tf.nn.dynamic_rnn(cell, X_one_hot, dtype=tf.float32)
```

Softmax (FC) in Deep CNN



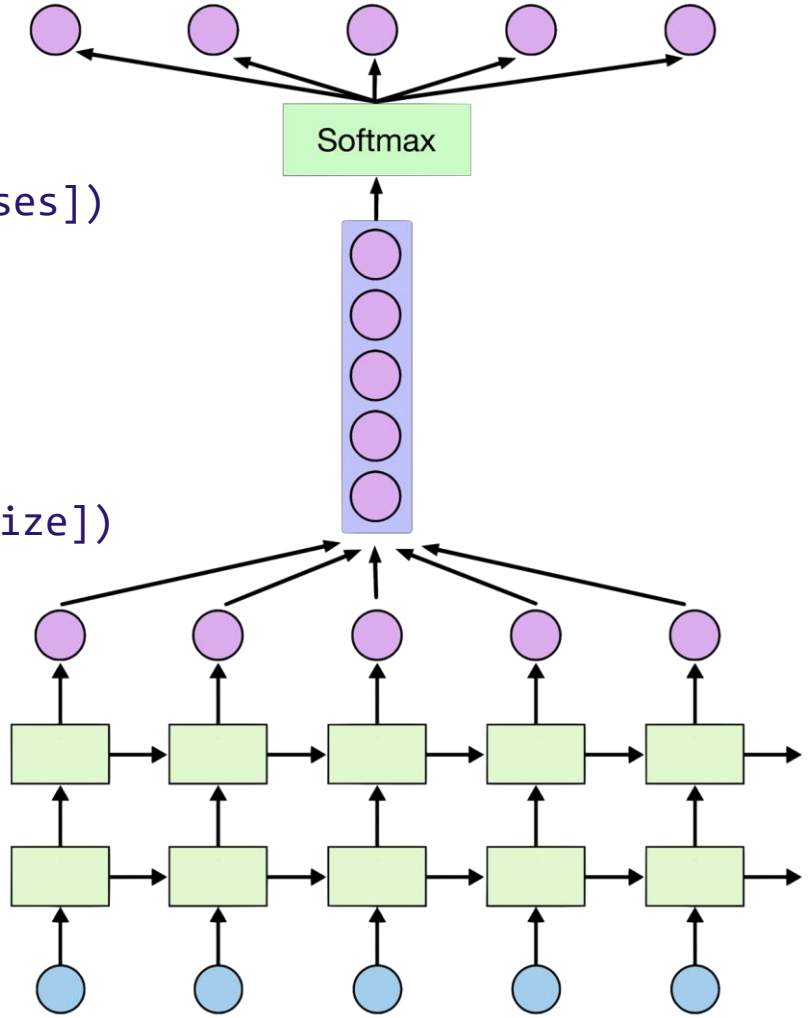
Softmax



Softmax

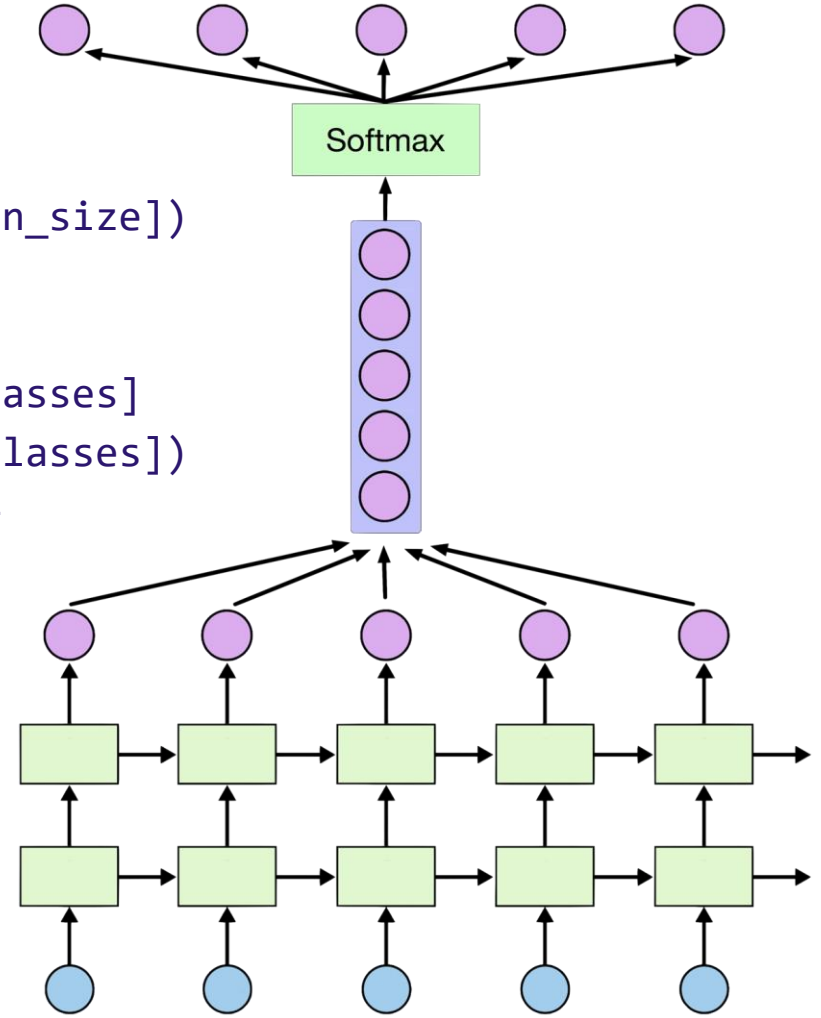
```
outputs = tf.reshape(outputs,
     [batch_size, seq_length, num_classes])
```

```
X_for_softmax = tf.reshape(outputs,
                           [-1, hidden_size])
```



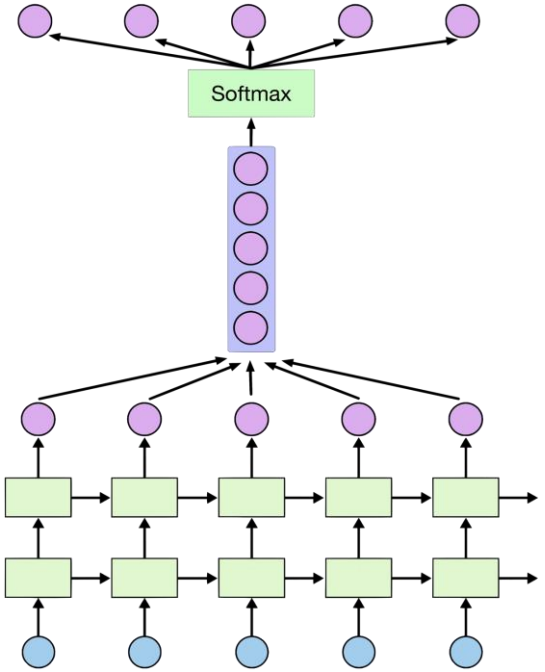
Softmax

```
# (optional) softmax layer
X_for_softmax = tf.reshape(outputs, [-1, hidden_size])

softmax_w = tf.get_variable("softmax_w",
                            [hidden_size, num_classes]
softmax_b = tf.get_variable("softmax_b",[num_classes])
outputs = tf.matmul(X_for_softmax,softmax_w) +
softmax_b

outputs = tf.reshape(outputs,
      [batch_size, seq_length, num_classes])
```



Loss

```
# reshape out for sequence_loss
outputs = tf.reshape(outputs,
           [batch_size, seq_length, num_classes])
# All weights are 1 (equal weights)
weights = tf.ones([batch_size, seq_length])

sequence_loss = tf.contrib.seq2seq.sequence_loss(
   logits=outputs, targets=Y, weights=weights)
mean_loss = tf.reduce_mean(sequence_loss)

train_op =
       tf.train.AdamOptimizer(learning_rate=0.1).minimize(mean_loss)
```



Training and print results

```
sess = tf.Session()
sess.run(tf.global_variables_initializer()

for i in range(500):
    _, l, results = sess.run(
                [train_op, mean_loss, outputs],
                feed_dict={X: dataX, Y: dataY})

    for j, result in enumerate(results):
        index = np.argmax(result, axis=1)
        print(i, j, ''.join([char_set[t] for t in index]), l)
```

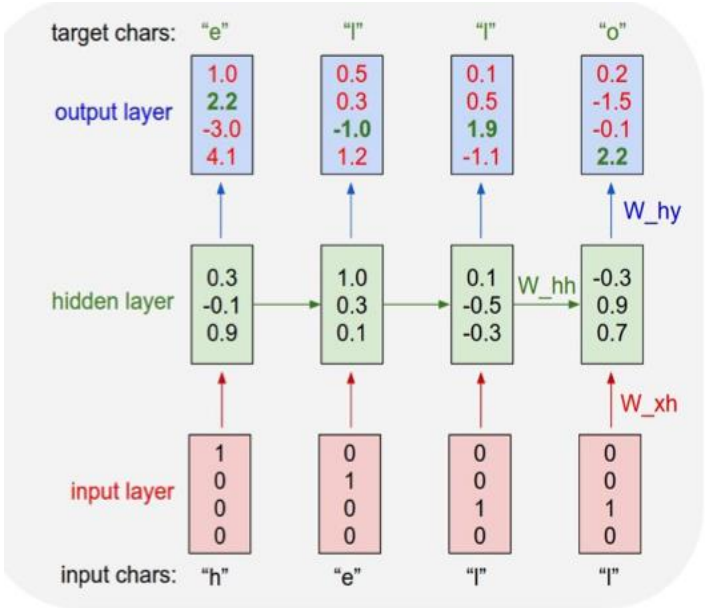
```
0 167 ttttttttt 3.23111
0 168 ttttttttt 3.23111
0 169 ttttttttt 3.23111
…
499 167 oof the se 0.229306
499 168 tf the sea 0.229306
499 169 n the sea. 0.229306
```

Training and print results

```
# Let's print the last char of each result to check it works
results = sess.run(outputs, feed_dict={X: dataX})
for j, result in enumerate(results):
    index = np.argmax(result, axis=1)
    if j is 0:  # print all for the first result to make a
sentence
        print(''.join([char_set[t] for t in index]), end='')
    else:
        print(char_set[index[-1]], end='')
```

g you want to build a ship, don't drum up people together to collect wood and don't assign them tasks and work, but rather teach them to long for the endless immensity of the sea.

Char-RNN

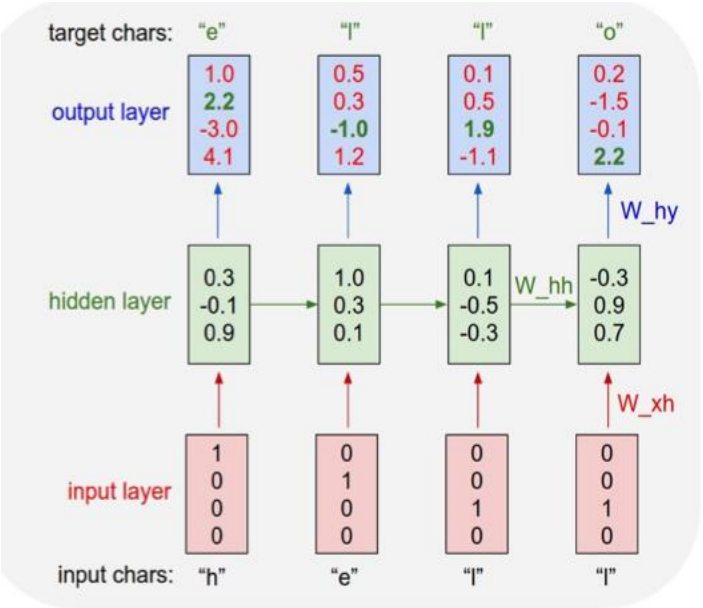


Shakespeare

It looks like we can learn to spell English words. But how about if there is more structure and style in the data? To examine this I downloaded all the works of Shakespeare and concatenated them into a single (4.4MB) file. We can now afford to train a larger network, in this case lets try a 3-layer RNN with 512 hidden nodes on each layer. After we train the network for a few hours we obtain samples such as:

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

Char-RNN

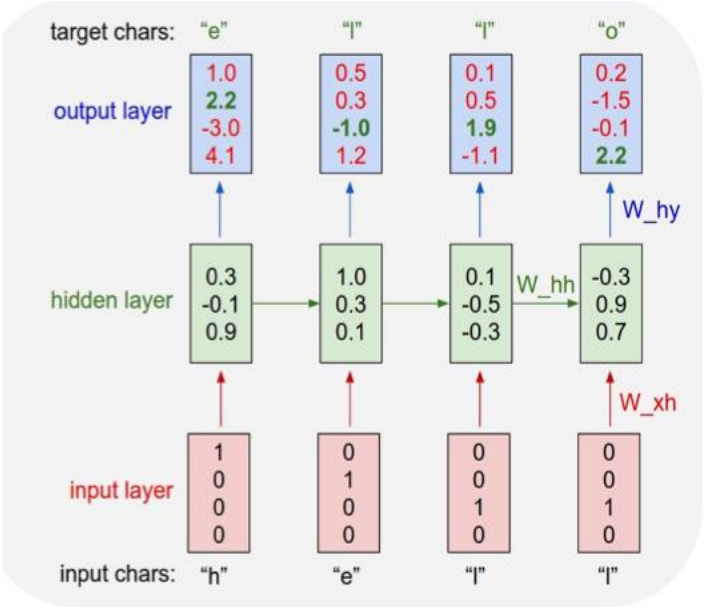


Linux Source Code

I wanted to push structured data to its limit, so for the final challenge I decided to use code. In particular, I took all the source and header files found in the Linux repo on Github, concatenated all of them in a single giant file (474MB of C code) (I was originally going to train only on the kernel but that by itself is only ~16MB). Then I trained several as-large-as-fits-on-my-GPU 3-layer LSTMs over a period of a few days. These models have about 10 million parameters, which is still on the lower end for RNN models. The results are superfun:

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

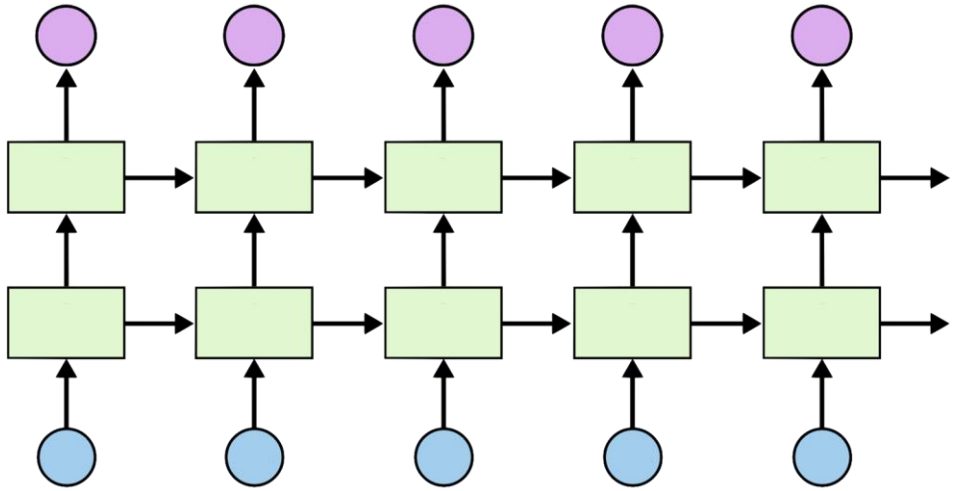
Char/Word RNN(char/word level n to n model)



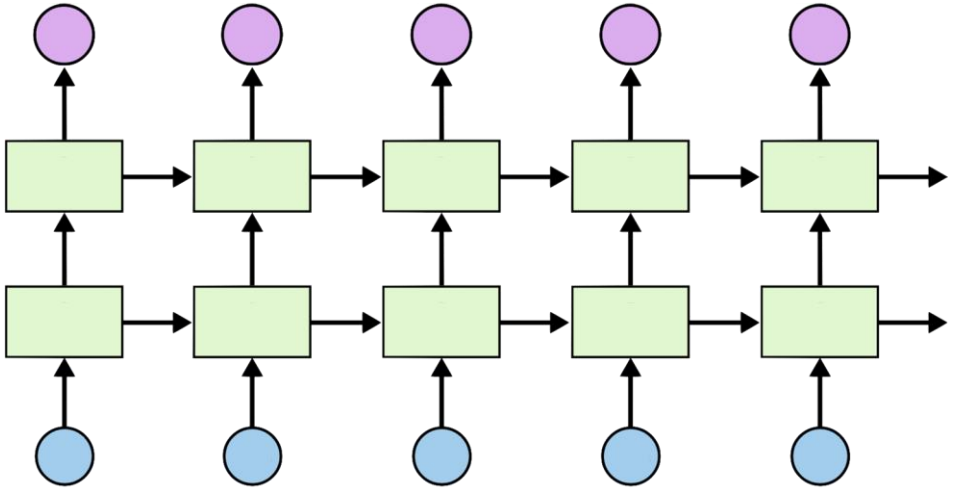
Practice

4. RNN Long Sentence

Different sequence length



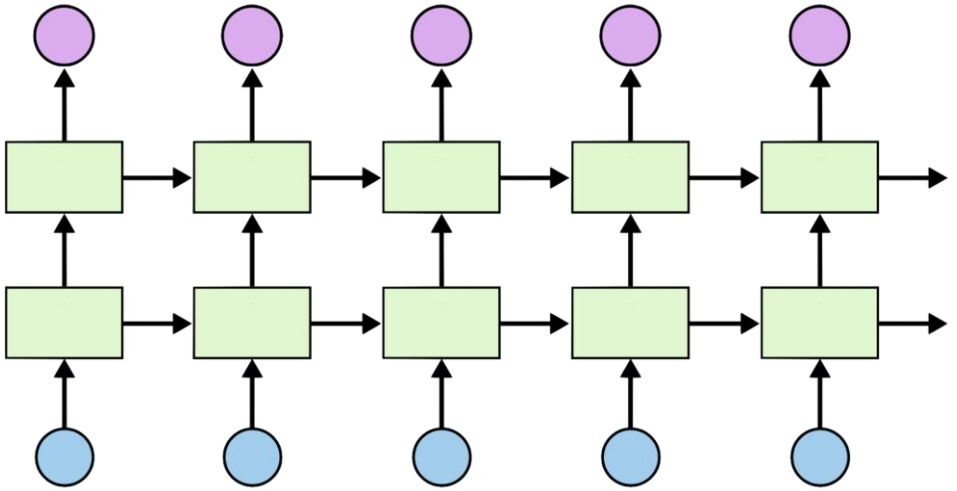
Different sequence length



h	e	l	l	o
h	i	<pad>	<pad>	<pad>
w	h	y	<pad>	<pad>
...				

Different sequence length

```
sequence_length=[5,2,3]
```



Dynamic RNN

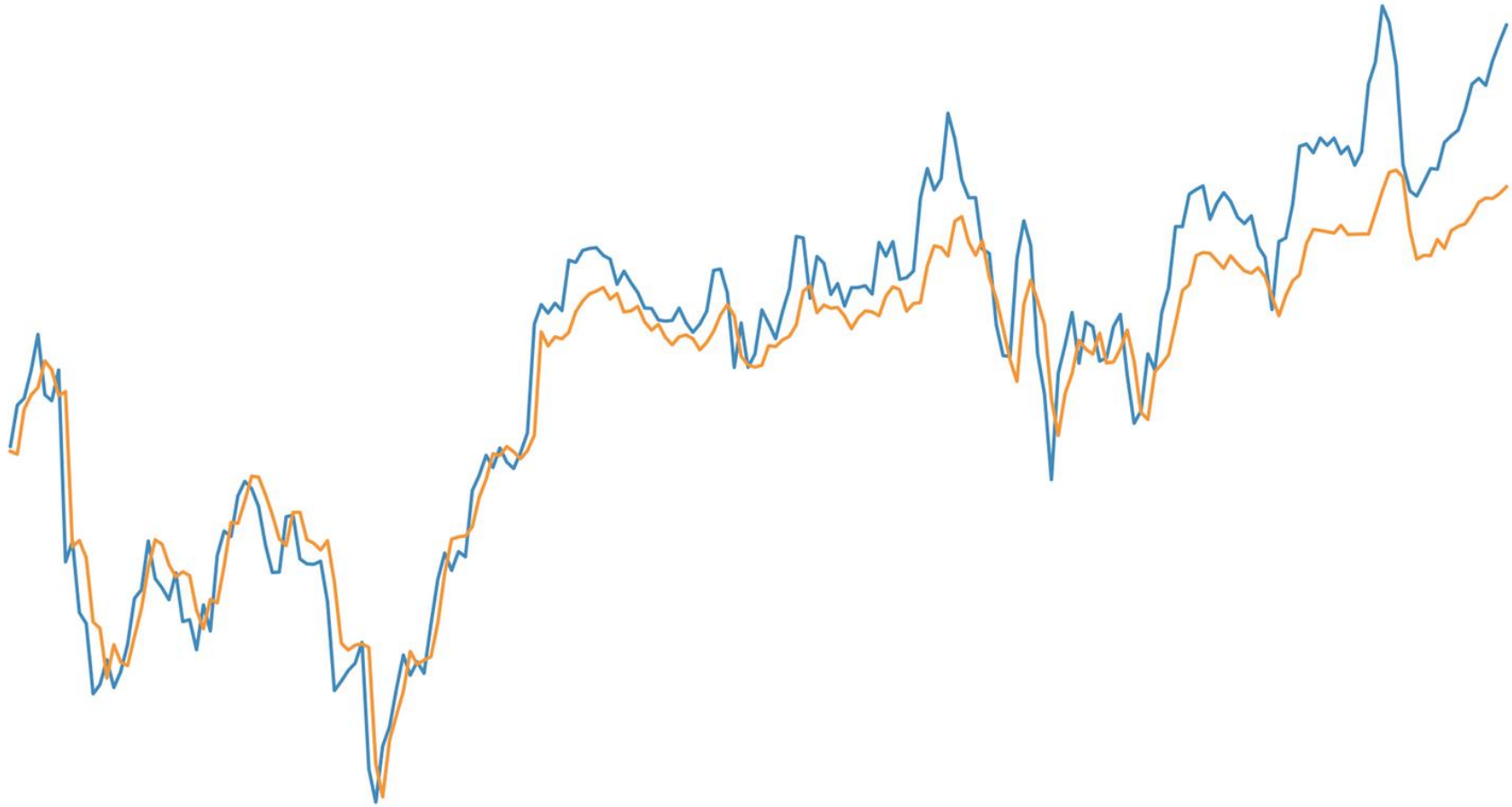
```
# 3 batches 'hello', 'eolll', 'lleel'
x_data = np.array([[[...]]], dtype=np.float32)

hidden_size = 2
cell = rnn.BasicLSTMCell(num_units=hidden_size,
                                 state_is_tuple=True)
outputs, _states = tf.nn.dynamic_rnn(
        cell, x_data, sequence_length=[5,3,4],
        dtype=tf.float32)
sess.run(tf.global_variables_initializer())
print(outputs.eval())
```

```
array([[[-0.17904168, -0.08053244],
        [-0.01294809,  0.01660814],
        [-0.05754048, -0.1368292 ],
        [-0.08655578, -0.20553185],
        [ 0.07297077, -0.21743253]],

       [[ 0.10272847,  0.06519825],
        [ 0.20188759, -0.05027055],
        [ 0.09514933, -0.16452041],
        [ 0.        ,  0.        ],
        [ 0.        ,  0.        ]],

       [[-0.04893036, -0.14655617],
        [-0.07947272, -0.20996611],
        [ 0.06466491, -0.02576563],
        [ 0.15087658,  0.05166111],
        [ 0.        ,  0.        ]]],
```

Practice

5. RNN Stock Prediction

Time series data

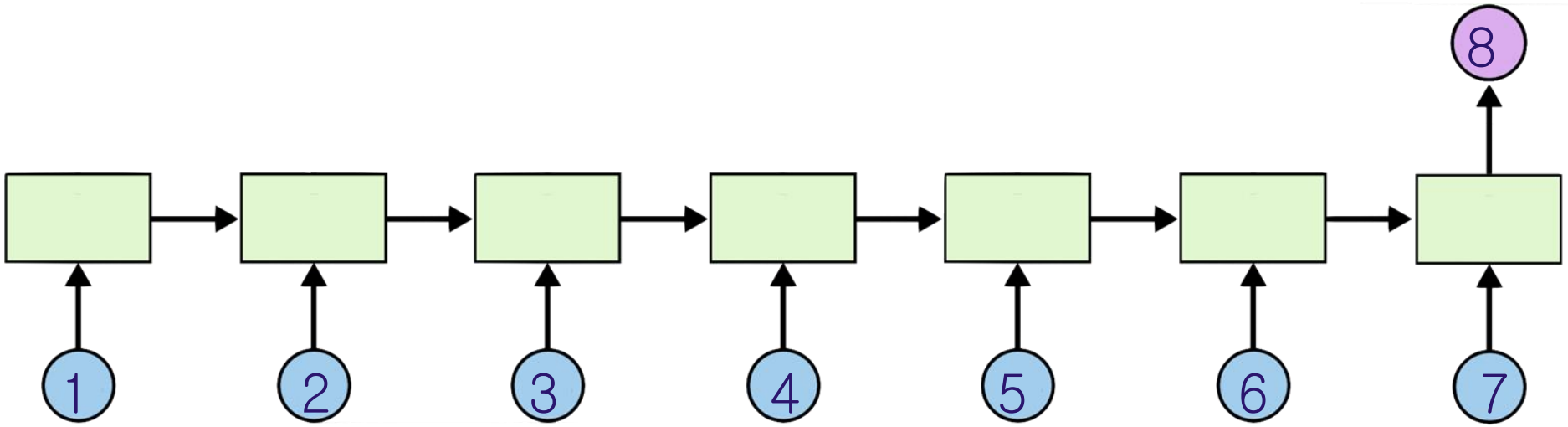


Time series data

Open	High	Low	Volume	Close
828.659973	833.450012	828.349976	1247700	**831.659973**
823.02002	828.070007	821.655029	1597800	**828.070007**
819.929993	824.400024	818.97998	1281700	**824.159973**
819.359985	823	818.469971	1304000	**818.97998**
819	823	816	1053600	**820.450012**
816	820.958984	815.48999	1198100	**819.23999**
811.700012	815.25	809.780029	1129100	**813.669983**
809.51001	810.659973	804.539978	989700	**809.559998**
807	811.840027	803.190002	1155300	**808.380005**

'data-02-stock_daily.csv'

Many to one



Reading data

```
timesteps = seq_length = 7
data_dim = 5
output_dim = 1
# Open,High,Low,Close,Volume
xy = np.loadtxt('data-02-stock_daily.csv', delimiter=',')
xy = xy[::-1]  # reverse order (chronically ordered)
xy = MinMaxScaler(xy)
x = xy
y = xy[:, [-1]]  # Close as label

dataX = []
dataY = []
for i in range(0, len(y) - seq_length):
    _x = x[i:i + seq_length]
    _y = y[i + seq_length]  # Next close price
    print(_x, "->", _y)
    dataX.append(_x)
    dataY.append(_y)
```

```
[ 0.18667876  0.20948057  0.20878184  0.
 0.21744815]

[ 0.30697388  0.31463414  0.21899367  0.0
 1247647  0.21698189]

[ 0.21914211  0.26390721  0.2246864   0.4
 5632338  0.22496747]

[ 0.23312993  0.23641916  0.16268272  0.5
 7017119  0.14744274]

[ 0.13431201  0.15175877  0.11617252  0.3
 9380658  0.13289962]

[ 0.13973232  0.17060429  0.15860382  0.2
 8173344  0.18171679]

[ 0.18933069  0.20057799  0.19187983  0.2
 9783096  0.2086465 ]]

-> [ 0.14106001]
```

Training and test datasets

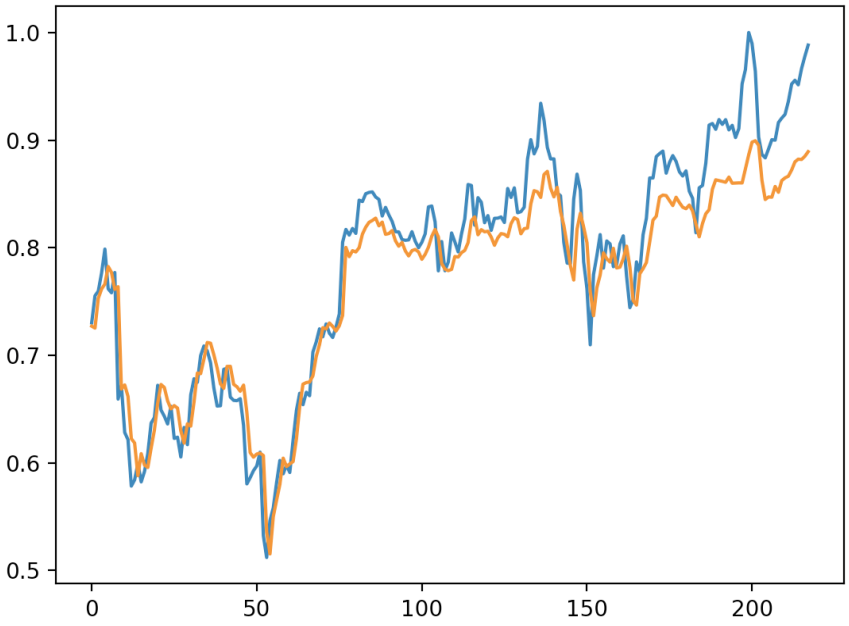
```
# split to train and testing
train_size = int(len(dataY) * 0.7)
test_size = len(dataY) - train_size
trainX, testX = np.array(dataX[0:train_size]),
                np.array(dataX[train_size:len(dataX)])
trainY, testY = np.array(dataY[0:train_size]),
                np.array(dataY[train_size:len(dataY)])

# input placeholders
X = tf.placeholder(tf.float32, [None, seq_length, data_dim])
Y = tf.placeholder(tf.float32, [None, 1])
```

LSTM and Loss

```
# input placeholders
X = tf.placeholder(tf.float32, [None, seq_length, data_dim])
Y = tf.placeholder(tf.float32, [None, 1])

cell = tf.contrib.rnn.BasicLSTMCell(num_units=output_dim, state_is_tuple=True)
outputs, _states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
Y_pred = outputs[:, -1]  # We use the last cell's output

# cost/loss
loss = tf.reduce_sum(tf.square(Y_pred - Y))  # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

Training and Results

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

for i in range(1000):
    _, l = sess.run([train, loss],
            feed_dict={X: trainX, Y: trainY})
    print(i, l)

testPredict = sess.run(Y_pred, feed_dict={X: testX})

import matplotlib.pyplot as plt
plt.plot(testY)
plt.plot(testPredict)
plt.show()
```



Exercise

- Implement stock prediction using linear regression only
- Improve results using more features such as keywords and/or s entiments in top news

Other RNN Applications

- Language Modeling
- Speech Recognition
- Machine Translation
- Conversation Modeling/Question Answering
- Image/Video Captioning
- Image/Music/Dance Generation


CRAFT
빅파이 크래프트
Find Information to design valuable society
김 진 수
CEO, Data Actionist
100-791 서울특별시 중구 청파로 463번지 3F BigData R&D Center
CP. 010-5670-3847 Tel. 02-360-4047 Fax. 02-360-4899
E-mail. bigpycraft@gmail.com
http://www.bigpycraft.com

수고하셨습니다!