Deep Learning: Part 4

東京海洋大学 TUMSAT

竹縄知之 Tomoyuki Takenawa

目次

1 Recurrent Neural Networks 4
1.1 Recurrent Neural Networks 6
1.2 Gradient computation in regression networks (BPTT) 9
1.3 Exercises on RNN 15
1.4 LSTM — gated RNN 16

2 Natural language processing 18
2.1 Corpus 20
2.2 Morphological analysis 21
2.3 Distributed representation of words 22
2.4 Sequence transformation 26
2.5 Exercises on sequence transformation 29
2.6 Attention 30

3 Reinforcement learning 33

4 Q-learning 36
4.1 Exercises on Q-learning 48

5 DQN 49
5.1 Exercises on DQN 54

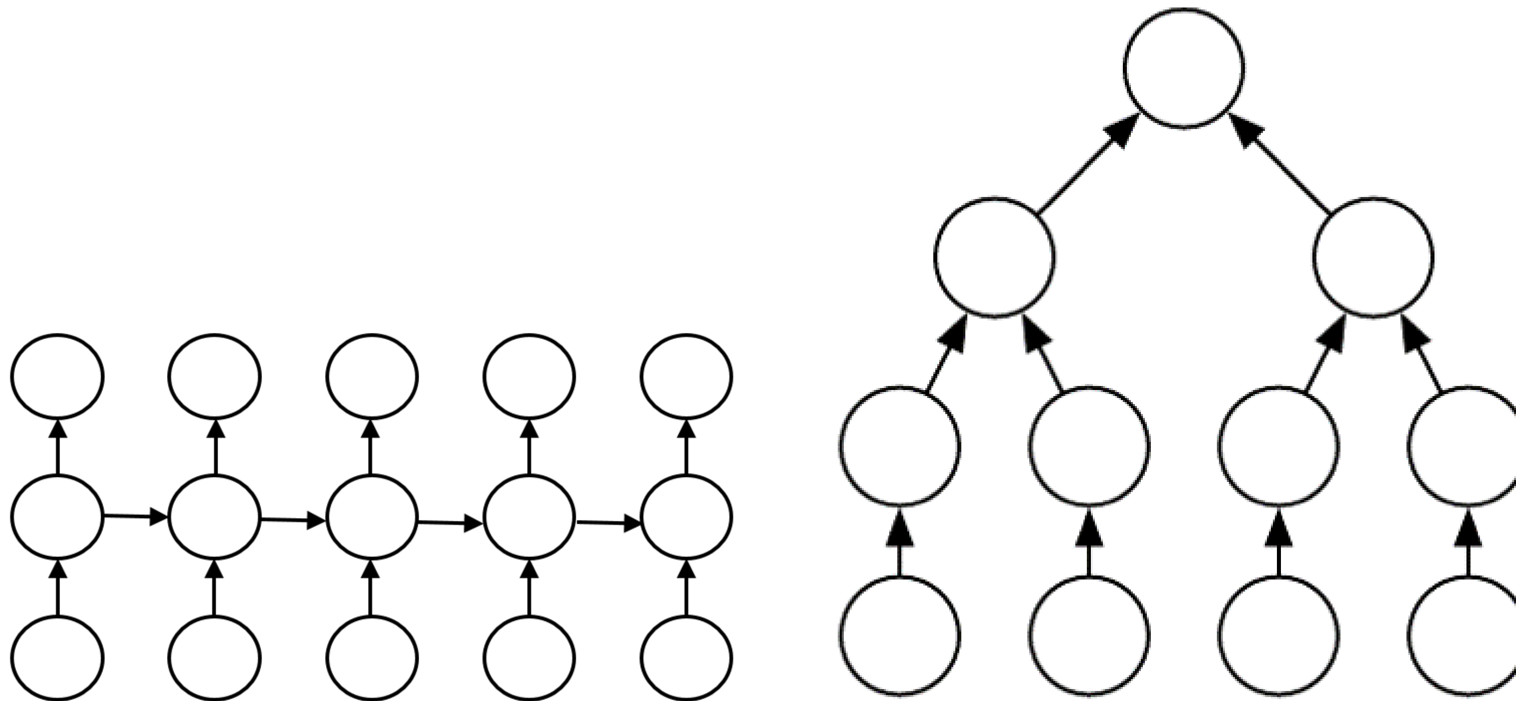
1 Recurrent Neural Networks

Recurrent Neural Networks, RNN, is a neural network for dealing with time series data. Time series data is difficult to learn with ordinary neural networks because the starting position of the series is different for each data.

Note that “reccurent” means periodic. On the other hand, in information science, ”recursive” is a broader concept, meaning that a function calls itself, and is used in for example the Tower of Hanoi and Euclid’s Reciprocal Algorithm.

Recurrent Neural Networks and Recursive Neural Networks

RNN is a network represented by a computational graph like the left side of the figure below. There is also a recursive neural network, which is a network represented by a computational graph like the one on the right.

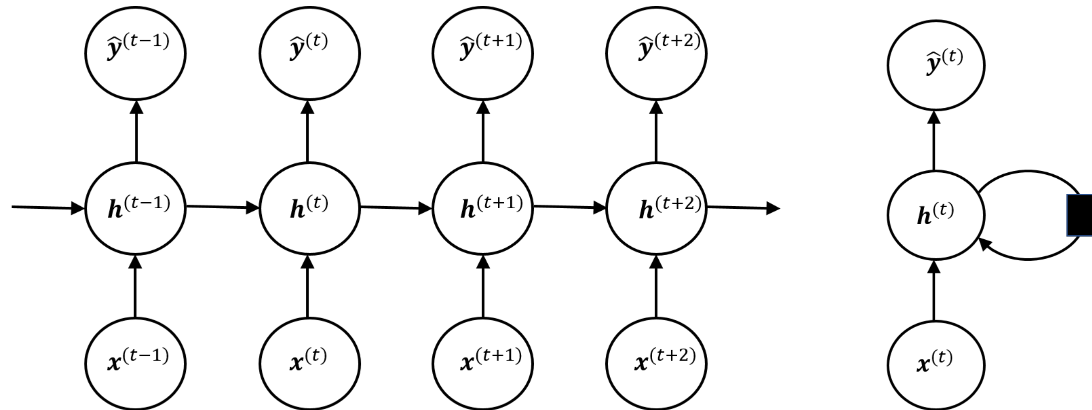


○ represents a layer, not a neuron.

The bottom is the input time series data and the top is the output (in the case of RNN, the output is also time series data)

1.1 Recurrent Neural Networks

An RNN that outputs some kind of predicted time series data $\{\hat{y}^{(t)}\}$ from input time series data $\{x^{(t)}\}$ is represented by the graph on the left in the figure below.



Here, $\{h^{(t)}\}$ is the hidden layer, and in RNN the information is passed from hidden layer to hidden layer along the time series. Instead of the figure on the left, it can be shown as the figure on the right. The black squares indicate that the time should be set one step back.

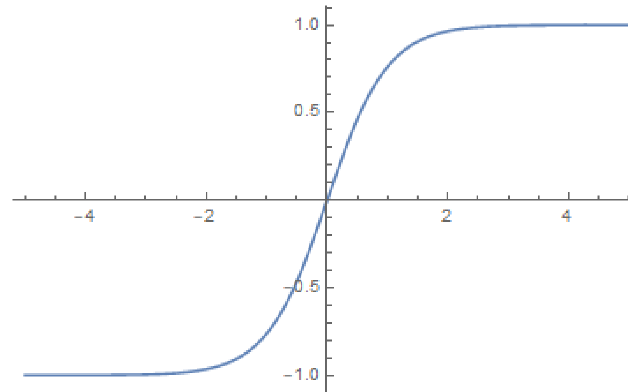
Forward propagation

In the simplest configuration, where the input and output at each time are real vectors, the forward propagation of the RNN is as follws.

$$\mathbf{a}^{(t)} = \mathbf{x}^{(t)}U + \mathbf{h}^{(t-1)}W + \mathbf{b} \quad \text{(Affine transformation with two inputs.)}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}) \quad \text{(Activation function)}$$
$$\hat{\mathbf{y}}^{(t)} = \mathbf{h}^{(t)}V + \mathbf{c} \quad \text{(Affine transformation)}$$

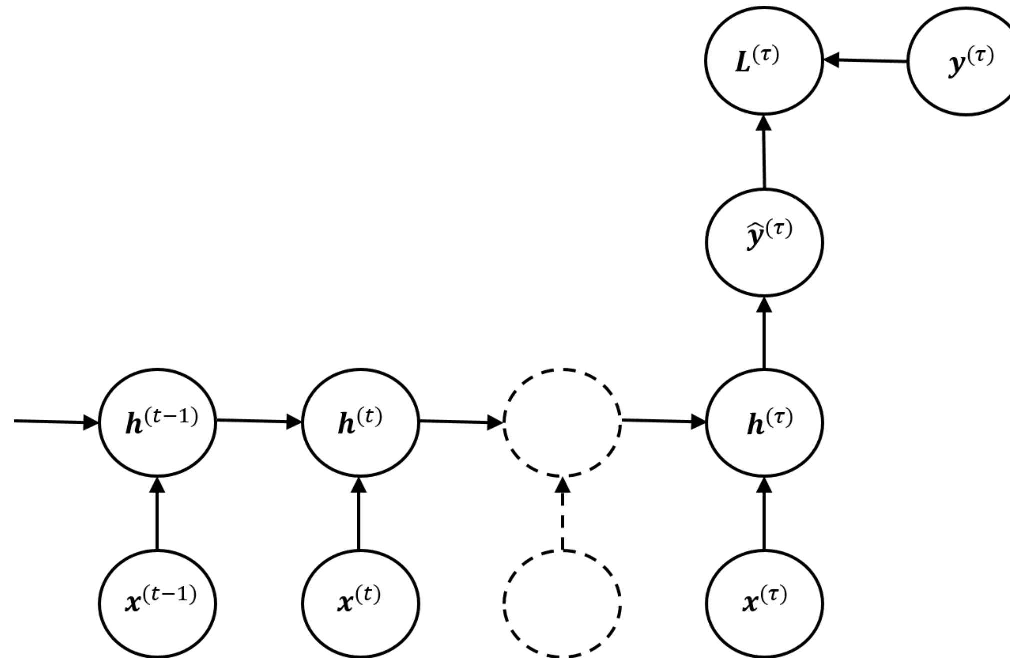
Here, the weight parameters U, V, and W are taken to be the same regardless of time.

$\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$ is the hyperbolic tangent whose graph is



When there is a single output

When the output is not time series data, but only takes a value at the last time τ, the network is represented as



where $y^{(\tau)}$ is the supervised label at the last time τ and $L^{(\tau)}$ is the loss function.

1.2 Gradient computation in regression networks (BPTT)

(The calculations in this section are complicated, so you can skip, but the point is that we are doing Backpropagation.)

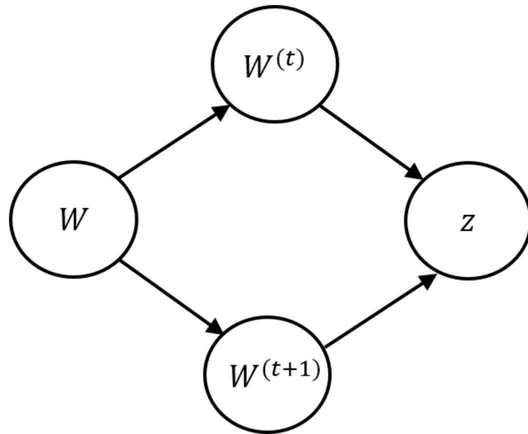
A simple RNN forward propagation was

$$\begin{aligned} \mathbf{a}^{(t)} &= \mathbf{x}^{(t)}U + \mathbf{h}^{(t-1)}W + \mathbf{b} \\ \mathbf{h}^{(t)} &= \tanh(\mathbf{a}^{(t)}) \\ \hat{\mathbf{y}}^{(t)} &= \mathbf{h}^{(t)}V + \mathbf{c} \end{aligned}$$

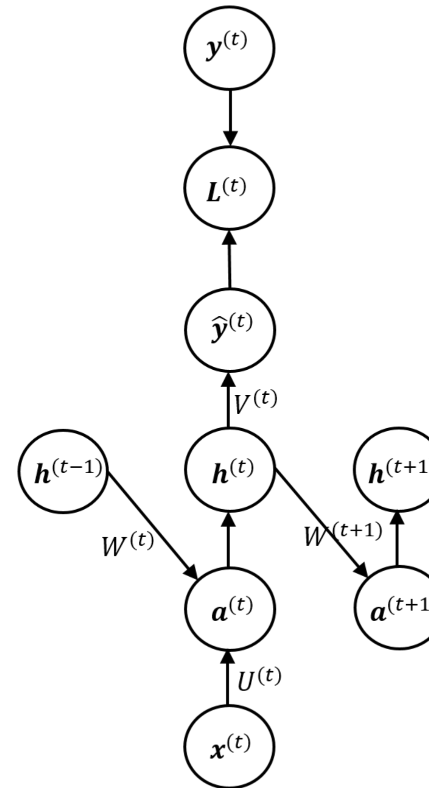
Let us calculate the derivative with respect to the weight parameter of L when the sum of the losses $L^{(t)}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})$ at each time is the loss of the entire model.

Confluence of back propagation (=branch of forward propagation)

In forward propagation, a common weight parameter is used regardless of time. In back propagation, the sum of the derivatives at each time is the actual derivative with respect to the parameter. This can be seen, for example, we have



$$\frac{\partial z}{\partial W} = \frac{\partial z}{\partial W^{(t)}} + \frac{\partial z}{\partial W^{(t+1)}}$$



for the dependency shown on the left in the figure above. Therefore, the parameters at each time in the network are distinguished by index t as $U^{(t)}$, $V^{(t)}$, $W^{(t)}$, $\mathbf{b}^{(t)}$, $\mathbf{c}^{(t)}$, etc.

In this case, forward propagation is represented by the computational graph shown on the right.

Derivative of $\tanh(x)$

The derivative of $h = \tanh x$ can be calculated using the differential formula for quotient as

$$(\tanh x)' = \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)' = \frac{(e^x - e^{-x})'(e^x + e^{-x}) - (e^x - e^{-x})(e^x + e^{-x})'}{(e^x + e^{-x})^2}$$
$$= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - (\tanh x)^2 = 1 - h^2$$

Calculation of Backpropagation 1: in the affine layer

We will return from the output with a computational graph.

From $L = \frac{1}{2}\left(\hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}\right)^2$,

$$\boxed{\frac{\partial L}{\partial \hat{\mathbf{y}}^{(t)}} = \hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}}$$

From $\hat{\mathbf{y}}^{(t)} = \mathbf{h}^{(t)} V^{(t)} + \mathbf{c}^{(t)}$,

$$\begin{aligned}
\frac{\partial L}{\partial V^{(t)}} &= \left(\mathbf{h}^{(t)}\right)^T \frac{\partial L}{\partial \hat{\mathbf{y}}^{(t)}} \\
\frac{\partial L}{\partial \mathbf{c}^{(t)}} &= \frac{\partial L}{\partial \hat{\mathbf{y}}^{(t)}} \\
\left(\frac{\partial L}{\partial \mathbf{h}^{(t)}}\right)_{\hat{\mathbf{y}}^{(t)}} &= \frac{\partial L}{\partial \hat{\mathbf{y}}^{(t)}} \left(V^{(t)}\right)^T
\end{aligned}$$

where $\left(\frac{\partial L}{\partial \mathbf{h}^{(t)}}\right)_{\hat{\mathbf{y}}^{(t)}}$ denotes the differentiation with respect to the path of $\mathbf{h}^{(t)} \to \hat{\mathbf{y}}^{(t)} \to L$.

Calculation of Backpropagation 2: in RNN layer [*1]

Since the back propagation merges at $\mathbf{h}^{(t)}$, from

$$\frac{\partial L}{\partial \mathbf{h}^{(t)}} = \left(\frac{\partial L}{\partial \mathbf{h}^{(t)}}\right)_{\hat{\mathbf{y}}^{(t)}} + \left(\frac{\partial L}{\partial \mathbf{h}^{(t)}}\right)_{\mathbf{a}^{(t+1)}}$$

the 2nd term is the derivative with respect to the path of $\mathbf{h}^{(t)} \to \mathbf{a}^{(t+1)} \to L$.

and $\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$, we have

$$\boxed{\frac{\partial L}{\partial \mathbf{a}^{(t)}} = \left(1 - \mathbf{h}^{(t)} * \mathbf{h}^{(t)}\right) * \frac{\partial L}{\partial \mathbf{h}^{(t)}}}$$

Also, from $\mathbf{a}^{(t)} = \mathbf{x}^{(t)} U^{(t)} + \mathbf{h}^{(t-1)} W^{(t)} + \mathbf{b}^{(t)}$, we have

$$\begin{aligned} \frac{\partial L}{\partial U^{(t)}} &= \left(\mathbf{x}^{(t)}\right)^T \frac{\partial L}{\partial \mathbf{a}^{(t)}} \\ \frac{\partial L}{\partial W^{(t)}} &= \left(\mathbf{h}^{(t-1)}\right)^T \frac{\partial L}{\partial \mathbf{a}^{(t)}} \\ \frac{\partial L}{\partial \mathbf{b}^{(t)}} &= \frac{\partial L}{\partial \mathbf{a}^{(t)}} \end{aligned}$$

and

[*1] In the implementation, only the first confluence is in a separate class named TimeRNN, which also copies the weights in the time direction.

Calculation of Backpropagation 3:

$$\begin{aligned} \left(\frac{\partial L}{\partial \mathbf{x}^{(t)}}\right) &= \frac{\partial L}{\partial \mathbf{a}^{(t)}} \left(U^{(t)}\right)^T \\ \left(\frac{\partial L}{\partial \mathbf{h}^{(t-1)}}\right)_{\mathbf{a}^{(t)}} &= \frac{\partial L}{\partial \mathbf{a}^{(t)}} \left(W^{(t)}\right)^T \end{aligned}$$

We have calculated the back propagation.

Also note the following.

- In order to compute inductively for t using these equations, we need to compute from the last t.
- As mentioned above, each parameter is independent of t, so we have to sum over t as

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L}{\partial V^{(t)}}$$

- These equations are for a single data and it needs to be summed appropriately for a mini-batch.

Remark on RNN

RNN shares the same weight parameters even if the time is different. This structure is similar to convolutional operations. Sharing the weight parameter means that the probabilistic dependence of each variable is assumed to be independent of the time t. If the data is missing or if the time intervals are not constant, the RNN cannot be expected to be effective.

1.3 Exercises on RNN

Notebooks:

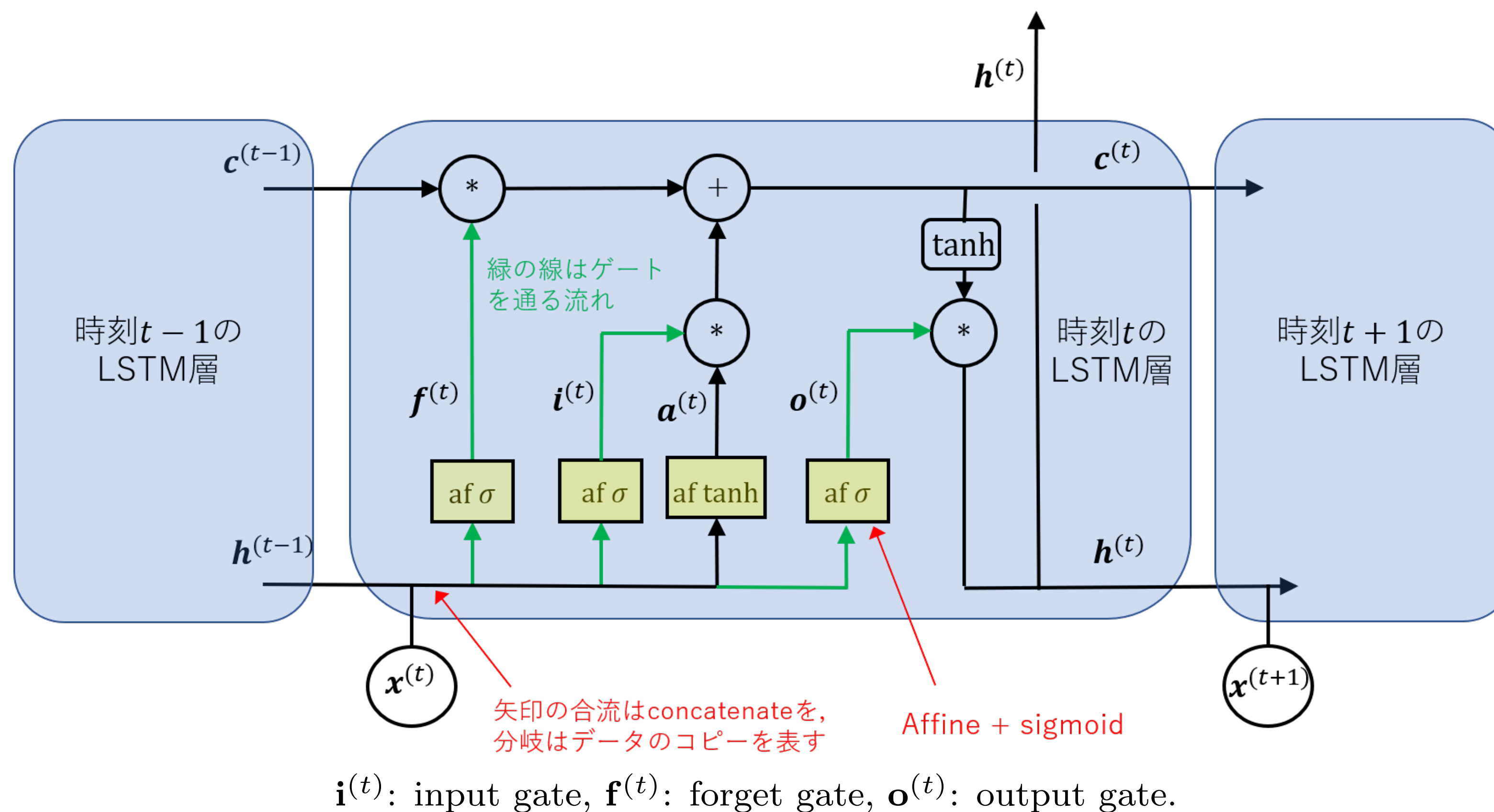
- `4-1-1_RNN層.ipynb`
- `4-1-2_TimeRNN層.ipynb`
- `4-1-3_RNN全体.ipynb`

In 7-1, we define the RNN layer as a class, and in 7-2, we expand the RNN layer in the time direction as a layer named TimeRNN. In 7-3, we create a simple training model using RNN. All of them are complicated, especially back propagation, so let's focus on forward propagation at first.

Training is performed using supervised data from a time series shifted by one in the time direction. Nevertheless, it can also be used to predict time series shifted by more than one. How does it work?

1.4 LSTM — gated RNN

LSTM (Long Short Term Memory) network [Hochreiter and Schmidhuber, 1997] is a model that uses "gates" for input, output, skip connections, etc., so that information can be propagated over a longer period of time than RNN, as shown in the graph below.



$\mathbf{i}^{(t)}$: input gate, $\mathbf{f}^{(t)}$: forget gate, $\mathbf{o}^{(t)}$: output gate.

Forward propagation of LSTM

The forward propagation for each gate is given by

$$\mathbf{i}^{(t)} = \sigma\left(\mathbf{x}^{(t)}U^{(i)} + \mathbf{h}^{(t-1)}W^{(i)} + \mathbf{b}^{(i)}\right)$$
$$\mathbf{f}^{(t)} = \sigma\left(\mathbf{x}^{(t)}U^{(f)} + \mathbf{h}^{(t-1)}W^{(f)} + \mathbf{b}^{(f)}\right)$$
$$\mathbf{o}^{(t)} = \sigma\left(\mathbf{x}^{(t)}U^{(o)} + \mathbf{h}^{(t-1)}W^{(o)} + \mathbf{b}^{(o)}\right)$$

Here, $U^{(i)}$, $U^{(f)}$, $U^{(o)}$, etc. are the weight parameters for each affine transformation, and are assumed to be common regardless of time t. Using these, in the hidden layer, we calculate as

$$\mathbf{a}^{(t)} = \mathbf{x}^{(t)}U + \mathbf{h}^{(t-1)}W + \mathbf{b}$$
$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} * \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} * \tanh\left(\mathbf{a}^{(t)}\right)$$
$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} * \tanh\left(\mathbf{c}^{(t)}\right).$$

In the LSTM, the values of $\mathbf{c}^{(t-1)}$ in the intermediate layer are not affine transformed, but are transmitted to $\mathbf{c}^{(t)}$ by simply applying the forgetting gate $\mathbf{f}^{(t)}$ to each component. As a result, the model is able to transmit information over long time periods. Thus, in LSTM, $\mathbf{c}^{(t)}$ plays the role of memory.

2 Natural language processing

Natural language processing is a branch of artificial intelligence and linguistics that involves a series of techniques for making computers process the natural language that humans use in their daily lives. In contrast to ”natural language” is the concept of a programming language. A programming language has no ambiguity, but a natural language does.

Ex. 1 「部長，書類はこれでよろしいでしょうか」「結構です」
「お客様，もう一杯おかわりはいかがでしょうか」「結構です」
(Difficult to translate)

Ex. 2 「No more tummy ache?」「It's okay.」
「Would you like a receipt?」「It's okay.」

Ex. 3 「A tall, dark-eyed boy」
「An American big blue-eyed girl」 (The relationship between the modifiers is different.)

Process flow of natural language processing

The flow of natural language processing in deep learning is basically as follows.

1. Prepare the **corpus** and the **machine-readable dictionary**. A corpus and a machine-readable dictionary are a set of sentences and a dictionary, respectively, that can be read by computers. (Dictionaries are not necessary for languages such as English where words are separated.)

2. Rewrite the text into a word-level split by performing a **morphological analysis**. Morphological analysis is a technique that uses a dictionary to separate words in a sentence and identify their parts of speech. (This is also not necessary in English.)

3. Use the information about the relationship of the word in the sentence to transform the word into a **distributed representation**. A distributed representation (or a word embedding) of a word is a real vector associated with the word.

4. Using learning models that can process time series, such as RNNs, perform tasks such as translation, automatic dialogue, and semantic understanding.

In the traditional way of processing machine learning, instead of 3 and 4, we proceed to "parsing" and "semantic analysis"

2.1 Corpus

A corpus is a set of texts made readable by a computer, and there are various types depending on the language and purpose. The following are just a few examples of Japanese corpora.

- **現代日本語書き言葉均衡コーパス**：A morphologically analyzed corpus of books, magazines, newspapers, white papers, blogs, internet forums, textbooks, laws, and other genres provided by the National Institute for Japanese Language.
- **青空文庫** Aozora bunko: A collection of works whose copyrights have expired or that have been declared ”free to read”, digitized in text and XHTML (partially HTML) format, available for download from Github.
- livedoor **ニュースコーパス**: A corpus containing news articles in nine fields, including topical news, sports, etc.: easy to use
- **日本語対訳データ（ポータルサイト）**http://phontron.com/japanese-translation-data.php?lang=ja
- Twitter **日本語評判分析データセット**: The results of crowdsourced analysis of tweet reputation information are provided.

2.2 Morphological analysis

Morphological analysis is a technique to decompose a sentence into words and identify the part of speech. Let's look at an actual example.

The following is the result of analyzing 「先生はとても忙しそうでした」(“Sensei seemed to be very busy”) with Janome, a Japanese morphological analyzer for Python.

先生 名詞, 一般,*,*,*,*, 先生, センセイ, センセイ
は 助詞, 係助詞,*,*,*,*, は, ハ, ワ
とても 副詞, 助詞類接続,*,*,*,*, とても, トテモ, トテモ
忙し 形容詞, 自立,*,*, 形容詞・イ段, ガル接続, 忙しい, イソガシ, イソガシ
そう 名詞, 接尾, 助動詞語幹,*,*,*, そう, ソウ, ソー
でし 助動詞,*,*,*, 特殊・デス, 連用形, です, デシ, デシ
た 助動詞,*,*,*, 特殊・タ, 基本形, た, タ, タ

Janome is a dictionary from the Mecab library that can be easily used in Python.

2.3 Distributed representation of words

A distributed representation or word embedding of a word is a real vector associated with the word. For example,

$$\begin{array}{lll} \text{“I”} & \text{is} & [0.12, 0.42, 0.87] \\ \text{“you”} & \text{is} & [0.31, 0.35, 0.72] \end{array}$$

and so on.

The easiest way to convert words into vectors is to use one-hot representations, but since there are 250,000 words in the 7th edition of Kojien, for example, we need 20,000 dimensions. In addition, there are also conjugations. Also, **「私」,「わたし」,「あたし」,「僕」,「ぼく」「俺」,「オレ**, etc. have similar meanings and can be replaced, but the relationship between these words is not clear at all.

In the distributed representation, we use real vectors to reduce the dimensions and to express the closeness of the meanings of words as the distance between vectors.

word2vec

To convert a word into a distributed representation, first assign a number to the word, convert it into a one-hot vector $\mathbf{x}$, and then multiply it by a transformation matrix (called an embedding matrix) W.

$$\underset{\mathbf{x}}{\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}} \underset{W}{\begin{bmatrix} 0.31 & 0.33 & 0.52 \\ 0.12 & 0.42 & 0.87 \\ 0.31 & 0.35 & 0.72 \\ \vdots & & \vdots & \vdots \\ 0.52 & 0.63 & 0.21 \end{bmatrix}} = \underset{\text{distributedrepresentation}}{\begin{bmatrix} 0.31 & 0.35 & 0.72 \end{bmatrix}}$$

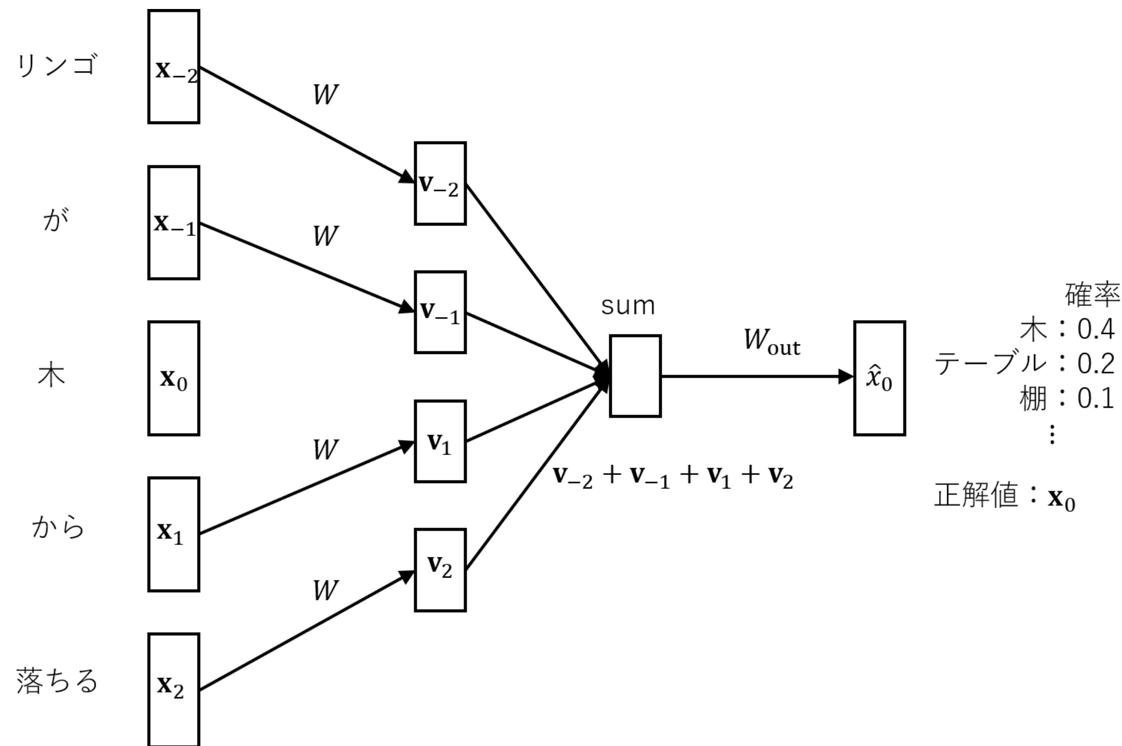
If you just want to embed, you can do it by setting the embedding matrix at random, but it does not tell you the relationship between words.

word2vec [Mikolov, T. et al., 2013] is a tool for efficiently learning this embedding matrix W.

The training of the embedding matrix is basically done by using the relationship between the preceding and following words, and word2vec provides two neural network methods: **Continuous Bag-of-Words (CBOW)** and **skip-gram**.

CBOW

Continuous Bag-of-Words is a method for learning W by learning the task of predicting the word via embedding from the words before and after the word. It learns two matrices, W and W_{out} in the figure below, but only W is used to create the distributed representation.

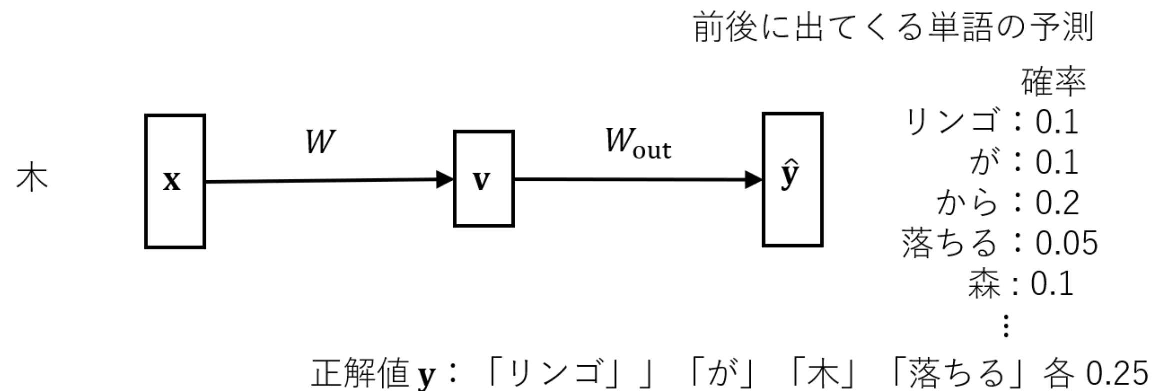


The W should be the same for all words.

In fact, there is a lot of innovation in the design of the loss function, but it is omitted here.

skip-gram

In contrast to CBOW, skip-gram is a method for learning W by learning the task of predicting the words before and after a word via embedding.



There is also a lot of innovation in the design of the loss function, but it is omitted here.

Cosine similarity

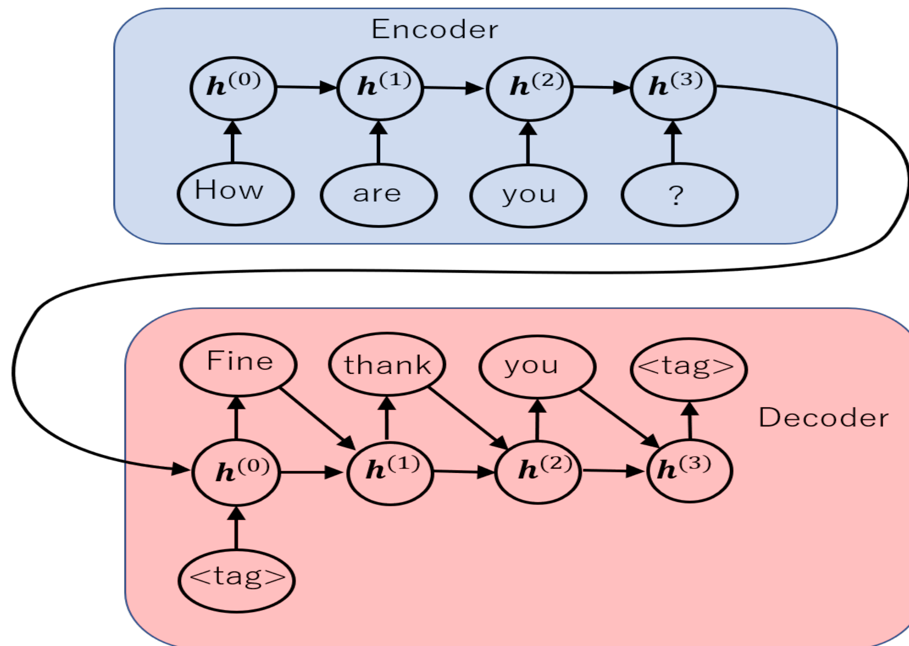
The variance representation created by word2vec is normalized so that each component is $[-1, 1]$, and the similarity between the two representations $\mathbf{v}_1$ and $\mathbf{v}_2$ is measured by cosine similarity:

$$\cos\theta = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{||\mathbf{v}_1||\ ||\mathbf{v}_2||} \qquad \text{(where } \theta \text{ is the angle between } \mathbf{v}_1 \text{ and } \mathbf{v}_2\text{)}$$

2.4 Sequence transformation

Tasks such as machine translation, auto-response, and auto-summarization require the transformation of a sequence into another sequence of different length. Such a sequence transformation model should be a network with an encoder and a decoder.

For example, the prediction of an automatic response by seq2seq [Sutskever et al., 2014] is performed by the following model.

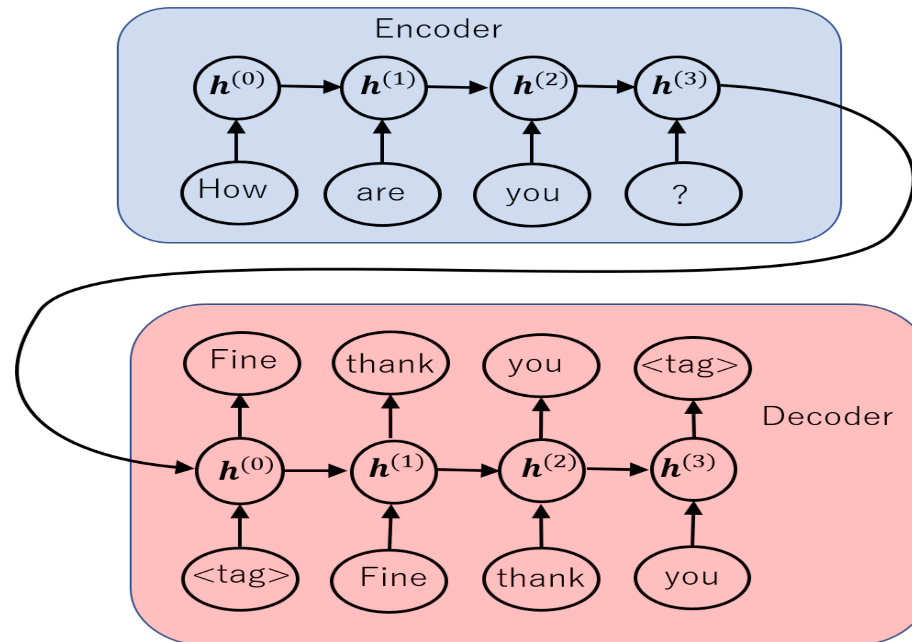


seq2seq 推測時

Note that there is in fact an embedding layer between the input (given by the word number) and the RNN layer, and an all-connecting layer between the RNN layer and the output layer.

seq2seq (in the training phase)

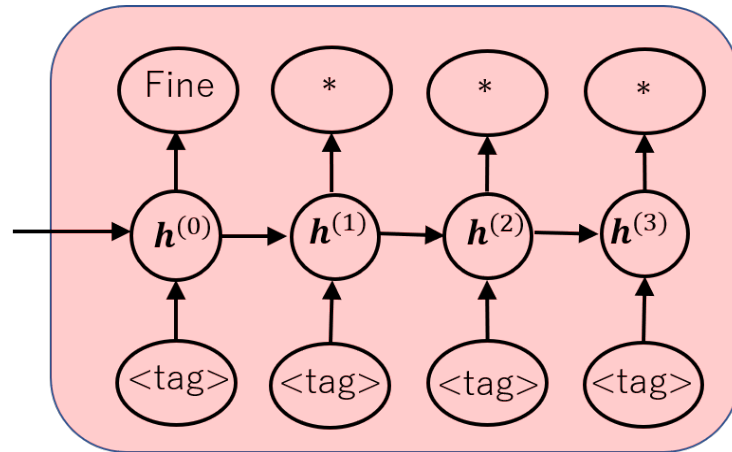
On the other hand, during training, we use the following model, which is enforced by the correct answer label from the correct answer data.



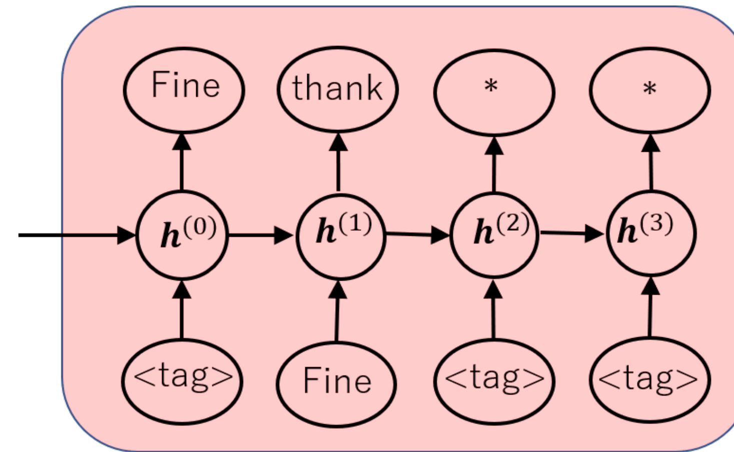
seq2seq 学習時

However, it is hard to prepare a separate model for prediction, so for practical use, the same model as during training can be used for prediction by iterating in the time direction while shifting the input to the decoder to the right one by one as shown in the next page.

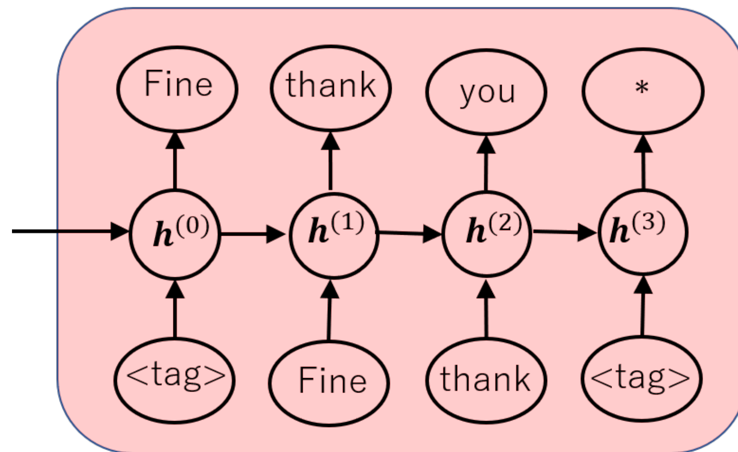
seq2seq (in the predivtion phase)



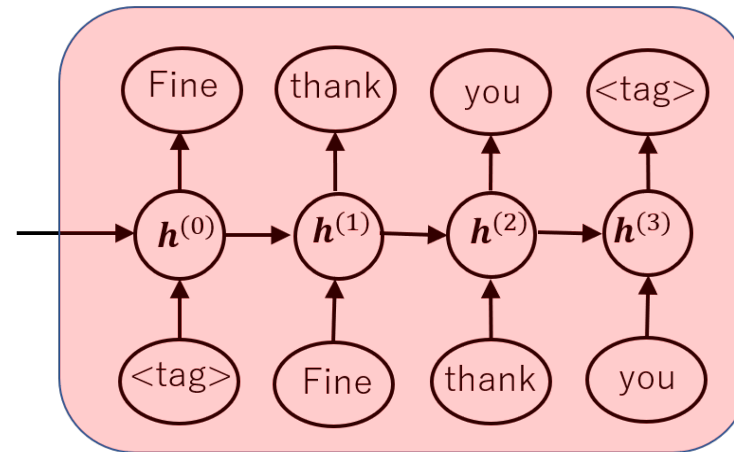
1回目



2回目



3回目



4回目

2.5 Exercises on sequence transformation

Notebook: `4-4_ChatBot_seq2seq.ipynb`

In this notebook, we try to implement ses2se2 by using Keras and create an automatic response system. The data used is from the ChatterBot Language Training Corpus (https://github.com/gunthercox/chatterbot-corpus). It's a small data set, so the results may not be that good, but let's try an experiment.

2.6 Attention

In a sequence transformation model such as seq2seq, the information from the encoder is passed only as the initial value to the decoder, so as the sequence gets longer, the information does not propagate well to the end of the sequence.

Attention [Bahdanau, D. et al., 2014] is a mechanism that addresses this challenge and has recently become a standard technique in the field of natural language processing.

Furthermore, in 2016, in a paper titled ”Attention is All You Need” [Vaswani, A. et al. 2016], a method called ”transformer” was proposed, in which the recurrent layers are abolished and only attention is used when dealing with time series data.
It has also received attention along with BERT (Bidirectional Encoders' Representations from Transformer) [Devlin, J. et al. 2018], which is a bidirectional version of it.

How attention works

Attention can be expressed by the following equation

$$\text{Attention: } C = \text{softmax}(QK^T,\ \text{axis} = 0) \cdot V,$$

where $K = V$ is the output sequence of the encoder, Q is the output sequence of the decoder, and there is only one sequence (i.e. we do not consider mini-batches).

- There is an interpretation of Q: Query, K: Key, V: Value. When the decoder asks "For a value Q like this, where should we focus our attention? then the encoder "measures the similarity between the question and the key K" and "passes the value V in that proportion" is the attention.

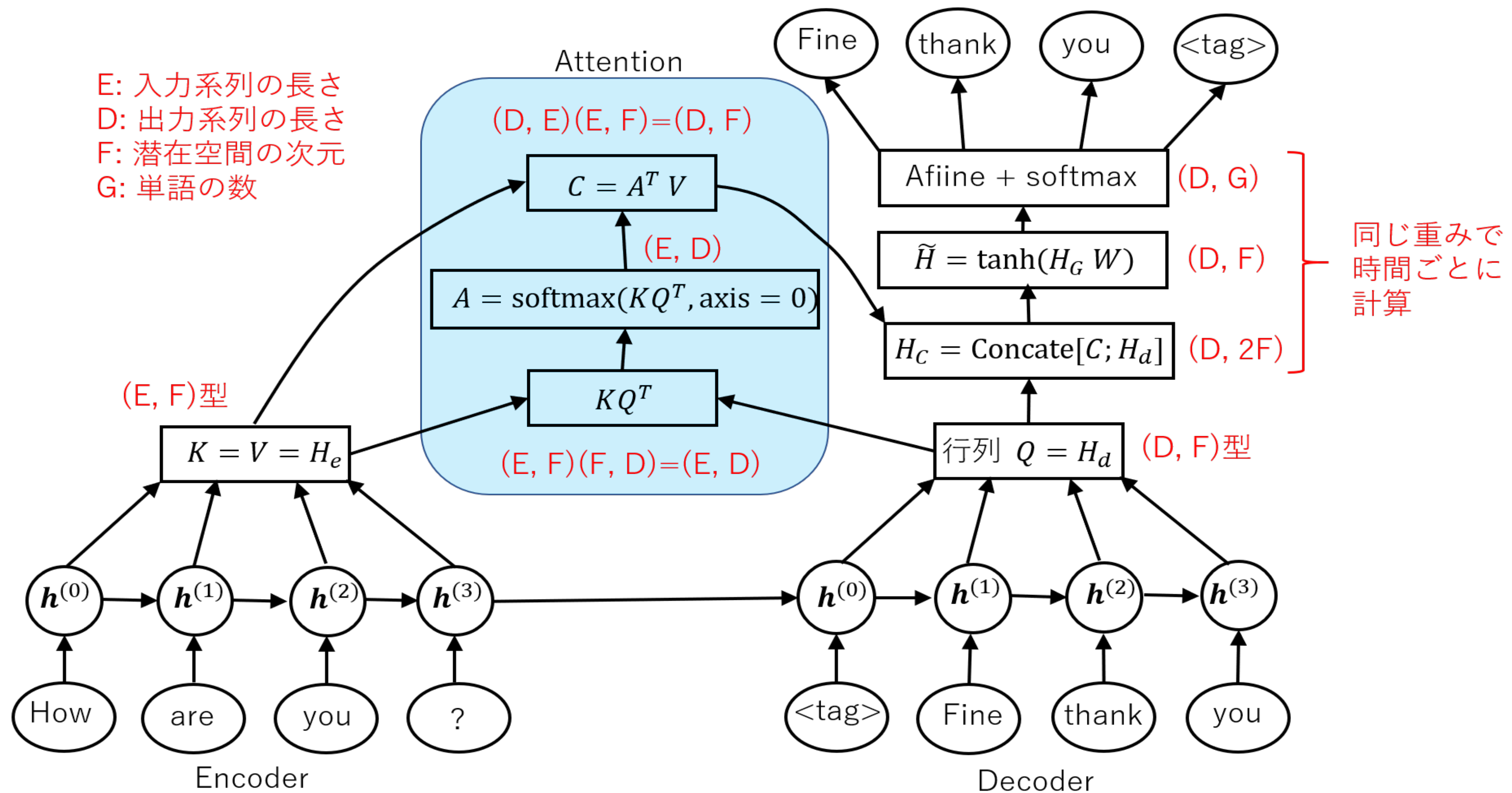
The decoder combines the received attention with its own output sequence $H_d = Q$ and uses the weight W to make

$$\tilde{H} = \tanh([C; H_d]W)$$

the new output of the decoder. Here, it is computed using the same W for each time t in the output sequence. This is similar to using an affine layer in the output layer of an RNN.

Note that if you don't use the same W, the attention will be meaningless.

Seq2seq network with attention



Forward propagation for the case of a single series. Note that in the case of mini-batch, the number of data is added and the axis is shifted by one.

3 Reinforcement learning

Let us consider a maze as shown in the leftmost figure below.

状態　　　行動

スタート	ゴール

0	1
2	3

up down left right

状態それぞれにある

Agent (the player) starts from state 0 and tries to reach the goal of state 1. Here, the red lines are not allowed to pass.

Of course, the answer (the shortest path) is

$$0 \to 2 \to 3 \to 1.$$

This sequence of events is called an **episode**. The goal of reinforcement learning is to obtain such good episodes.

Action

Actions are determined for each of the four states, so there are 16 possible actions.

(0,up)	(0,down)	(0,left)	(0,right)
(1,up)	(1,down)	(1,left)	(1,right)
(2,up)	(2,down)	(2,left)	(2,right)
(3,up)	(3,down)	(3,left)	(3,right)

Transition function

According to the "action", the state changes as follows.

	up	down	left	right
0	0	2	0	0
1	1	3	1	1
2	0	2	2	3
3	1	3	2	3

This is called the transition function. When the system hits a wall, it is assumed to stay in that state. In a more general situation, the transition function is given by a conditional probability distribution over the states.

Reward

Suppose the agent is rewarded for the actions. In this problem, the purpose is to reach the goal, so the reward is +1 only when the agent moves up from state 3 to state 1.

	up	down	left	right
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	1	0	0	0

To distinguish it from the cumulative reward that will appear later, this reward is sometimes called the "immediate reward. If the reward at time t is r, then the value of the reward at time 0 is $r\gamma^t$ $(0 < \gamma < 1)$.

This is called the **discounted present value**(the present value of the future reward). This means that there is a higher value if the goal is reached sooner than later. Let's assume that $\gamma = 0.9$.

Policy

The way to choose an action in a state s is called a strategy (or action strategy). Depending on the learning method, measures may be given as probability distributions, or they may be chosen deterministically from the situation.

4 Q-learning

Q-learning is a method of learning about the expected value of reward for each action. It is based on the idea that the player should be able to reach the goal by continuing to take actions with the highest expected value of reward.

Q-learing is a classical learning method proposed by [Watkins, C.J.C.H., 1989], but there is also a deep learning version of Q-learning, which is DQN, Deep Q-networks [Mnih, V. et al. 2013，2015].

In the other famous method of reinforcement learning, the policy gradient method, the strategy for choosing an action is always given as a probability distribution, while in Q-learning, the player always takes the action with the highest expected reward, so the strategy can be chosen deterministically.

Action value function

The expected value of the discounted present value of the reward for performing an action at time 0 and then continuing to perform the action under a certain strategy is called the action value function. For example, if a player has a strategy as "continue to act as up, down, left, right, up, down, left, right, $\cdots$ from time 1, regardless of state", the player who chooses the action (0, right) at time 0 will act as

Time 0:	(0, right)	stays at state 0;
Time 1:	(0, up)	stays at state 0;
Time 2:	(0, down)	moves to state 2;
Time 3:	(2, left)	stays at state 2;
Time 4:	(2, right)	moves to state 3;
time 5:	(3, up)	moves to state 1, gets reward 1, and ends.

Thus, the action value function in this case is the discounted present value of the reward obtained at time 5, $\gamma^5 = 0.9^5$ (we assumed $\gamma = 0.9$). This is written as

$$Q^{\pi}(0, \text{right}) = 0.9^5.$$

Optimal action-value function

The optimal action-value function is the action value function based on the strategy of "continuing to take the action with the highest action value function in each state." Since the definition of the optimal action value function refers to itself, it is necessary to solve some equation to calculate the optimal action-value function.

For any action (s, a) in state s, let $Q(s, a)$ be the optimal action value of action (s, a). Let $R(s, a)$ be the immediate reward for action (s, a), and let s' be the state to which the action results. Since $Q(s, a)$ is the value of the optimal action after choosing action (s, a), if (s', a') is the action for which the optimal action value function $Q(s', a')$ is highest in state s', then

$$Q(s, a) = R(s, a) + \gamma Q(s', a')$$

should hold. Therefore,

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

should hold. This is the equation that the optimal behavioral value function satisfies, which is a special case of the so called **Bellman equation**.

Note: $Q(s, a)$ is defined for any action (s, a). In other words, it is the expected cumulative reward if only the first action from state s is arbitrary, and the following actions are optimal.

Optimal action-value function (in the case of the 2×2 maze)

It is not easy to solve the equation

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a'),$$

but by solving from the cells closest to the goal, we can solve it as follows

	up	down	left	right
0	0.729	0.81	0.729	0.729
1	nan	nan	nan	nan
2	0.729	0.81	0.81	0.9
3	1	0.9	0.81	0.9

Solution of Bellman equation

For example, the action (3,up) immediately gives a reward of 1, so $Q(3,\mathrm{up}) = 1$. If you perform the action (2,right), the reward is zero and you move to state 3, and at the next time you choose (3,up), which has the highest action value function, $Q(3,\mathrm{up}) = 1$, so $Q(2,\mathrm{right}) = 0 + 0.9 \max_{a'} Q(3,a') = 0.9$, and so on. This kind of algorithm is called Dynamic Programming.

Once the optimal action value function is known, the shortest path can be traced by repeating the optimal action in each state.

Q-learning

In Q-learning (table Q-learning), the Bellman equation

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a'), \quad (s' \text{ is the state transferred by } (s,a)) \quad \cdots ①$$

is solved approximately by repeating experiences. First, for constant α $(0 < \alpha < 1)$, adding α times ① and

$$(1-\alpha)Q(s,a) = (1-\alpha)Q(s,a),$$

we get

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left(R(s,a) + \gamma \max_{a'} Q(s',a')\right).$$

Sorting the right-hand side by α, we have

$$Q(s,a) = Q(s,a) + \alpha\left(R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right).$$

The basic idea of the Q-learning is to replace the left side with the right side of this equation, starting with $Q(s,a)$ taken at random. In other words, while changing (s,a) in various ways, $Q(s,a)$ is updated by

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right).$$

This kind of calculation using random sampling is called the Monte Carlo method.

Algorithm of Q-learning

However, determining (s, a) completely at random is not efficient for learning, so we use the following algorithm.

Input: $Q(s, a) = 0$ (or randomly determined) for any (s, a).
time: Natural number (maximum number of times to proceed in one episode)
ε: real number of $0 < \varepsilon \leq 1$ (a measure for how randomly an action is selected)
Execute the following episodes a certain number of times or until Q stops improving:

1. Determine the initial state s, where $s \neq$ goal.
   Set t as $t = 0$.
2. Fix s and run (1) below with probability ε and (2) with the rest.
   (1) Determine the action (s, a) randomly. (not greedy)
   (2) Determine the action (s, a) so that $Q(s, a)$ is maximized. (greedy)
3. Let s' be a state that is transferred as a result of action (s, a).
   $Q(s, a) \leftarrow Q(s, a) + \alpha \left(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)$
4. Upadate s as $s \leftarrow s'$
   If $s =$ goal or $t =$ time, go on to the next episode.
   If not, return to 2 with $t \leftarrow t + 1$.

Algorithm of Q-learning (continued)

By selecting (2) in 2, it will learn ”greedily” along with the results it has learned so far. If $\varepsilon = 0$, it will always act greedily and choose only the same action, and learning may not progress. On the other hand, if $\varepsilon = 1$, the state is completely determined by a random number, and it will search in equal proportions even in the less important range.

Thus, a small ε leads to a more greedy search, while a large ε leads to a more wide-ranging search. This type of learning method is called **ε-greedy method**.

The reason for stopping the repetition of an episode at “time” is that if we start too far away from the goal, especially in the beginning, it will be difficult to reach the goal and it will be inefficient.

Example of Q-learning

Let us look at a concrete example to see how the Q-function behaves in the Q-learning. Let α be a large value, $\alpha = 0.7$.

Initial table

	up	down	left	right
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

0	1
2	3

1. If we choose $s = 2$ and (2,right), then $s' = 3$ and $Q(s', a') = Q(3, \text{up}) = 0$, so the table does not change.
2. (3,up) と選ぶと $R(3, \text{up}) = 1$ で $s' = 1$，$Q(s', a') = Q(1, a') = 0$ なので

$$Q(3, \text{up}) = 0 + 0.7(R(3, \text{up}) + 0.9 \cdot 0 - 0) = 0.7$$

	up	down	left	right
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0.7	0	0	0

Example of Q-learning (continued)

3. Since the goal has been reached, we re-select $s = 2$. If we choose $(2, right)$, then $Q(s', a') = Q(3, \text{up}) = 0.7$, so

$$Q(2, \text{right}) = 0 + 0.7(0 + 0.9 \cdot 0.7 - 0) = 0.441$$

	up	down	left	right
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0.441
3	0.7	0	0	0

0	1
2	3

4. If we choose (3,down), then $Q(s', a') = Q(3, \text{up}) = 0.7$, so

$$Q(3, \text{down}) = 0 + 0.7(0 + 0.9 \cdot 0.7 - 0) = 0.441$$

	up	down	left	right
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0.441
3	0.7	0.441	0	0

Example of Q-learning (continued)

5. If we choose (3,up), then $R(3, \text{up}) = 1$ and $Q(s', a') = Q(1, a') = 0$, so

$$Q(3, \text{up}) = 0.7 + 0.7(1 + 0.9 \cdot 0 - 0.7) = 0.91$$

	up	down	left	right
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0.441
3	0.91	0.441	0	0

0	1
2	3

6. The goal is reached, so we re-select as $s = 0$. If we choose (0,down), then $Q(s', a') = Q(2, \text{right}) = 0.441$, so

$$Q(0, \text{down}) = 0 + 0.7(0 + 0.9 \cdot 0.441 - 0) = 0.27783$$

	up	down	left	right
0	0	0.27783	0	0
1	0	0	0	0
2	0	0	0	0.441
3	0.91	0.441	0	0

Example of Q-learning (continued)

Continue to do this kind of process, finally we have the following optimal action value function.

	up	down	left	right
0	0.729	0.81	0.729	0.729
1	0	0	0	0
2	0.729	0.81	0.81	0.9
3	1	0.9	0.81	0.9

In this state, the optimal action value function will not change.

Issues in Q-learning

1. The search area is too large: Basically, we need to keep the optimal action value function Q for all actions. However, in Go, for example, there is no point in learning an impossible phase.
2. Cannot handle states unless they are discrete values: This is related to the first issue, but for example, continuous variables between 0 and 1 can be divided into $[0, 0.1)$, $[0.1, 0.2)$, $\cdots$, $[0.9, 1)$, and made into discrete variables. However, for example, if there are 8 variables, there will be $10^8 = 100,000,000$ states. This is almost impossible to learn.

In DQN, which will be introduced in the next section, the optimal action value function Q is represented by a neural network, and the weight parameters of the network are learned instead of Q itself.

4.1 Exercises on Q-learning

Notebook: `8-1_テーブルQ学習_迷路.ipynb`

Let us solve the maze using Q-learning. In addition to the 2×2 example shown above, let's solve the following maze:

スタート 0	1	2	3
4	5	6	ゴール 7
8	9	10	11
12	13	14	15

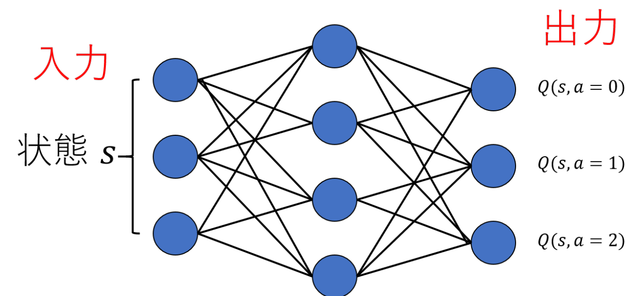
5 DQN

As in the issue mentioned at the end of the previous section, the action value function (Q-table) becomes a function with a very large number of discrete variables as arguments. DQN (Deep Q Networks) [Mnih et al., 2015], which we study in this section, addresses this issue.

DQN is a method of reinforcement learning by approximating the action value function Q with a neural network. The input of the neural network is the state, and the output is the value function of each action at that time. That is, when the number of actions is A, for a state s, the weight parameter θ of $Q(s, a)$ determines the approximate value

$$Q(s, a = 0, \theta), \quad \cdots, \quad Q(s, a = \mathrm{A} - 1, \theta).$$

By using neural networks, it is possible to deal with high-dimensional data and to learn when the state is a continuous variable or when the data is image data.



Principle of DQN

In the case of Q-learning, we used the fact that the optimal action value function $Q(s, a)$ satisfies the Bellman equation

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a'), \quad (s' \text{ is a state that is transitioned as a result of } (s,a))$$

for an arbitrary $(s, a) = (state, action)$, and we brought $Q(s, a)$ so that it satisfies the Bellman equation by using the results of actual actions.

In the case of DQN, the goal is to have its approximation $Q(s, a, \theta)$ instead of $Q(s, a)$ satisfy

$$Q(s,a,\theta) = R(s,a) + \gamma \max_{a'} Q(s',a',\theta),$$

where the right-hand side is considered to be the more correct value through experience.

Suppose you get a reward $R(s, a)$ for choosing action a in state s. The loss function for a single action is $\frac{1}{2}$ of squared error when the right-hand side is the supervised data and the left-hand side is the predicted value as

$$E = \frac{1}{2}\left({\color{blue}Q(s,a,\theta)} - {\color{red}\left(R(s,a) + \gamma \max_{a'} Q(s',a',\theta)\right)} \right)^2$$

(blue is the predicted value and red is the supervised value).

Principle of DQN (continued)

By backpropagating the derivative of

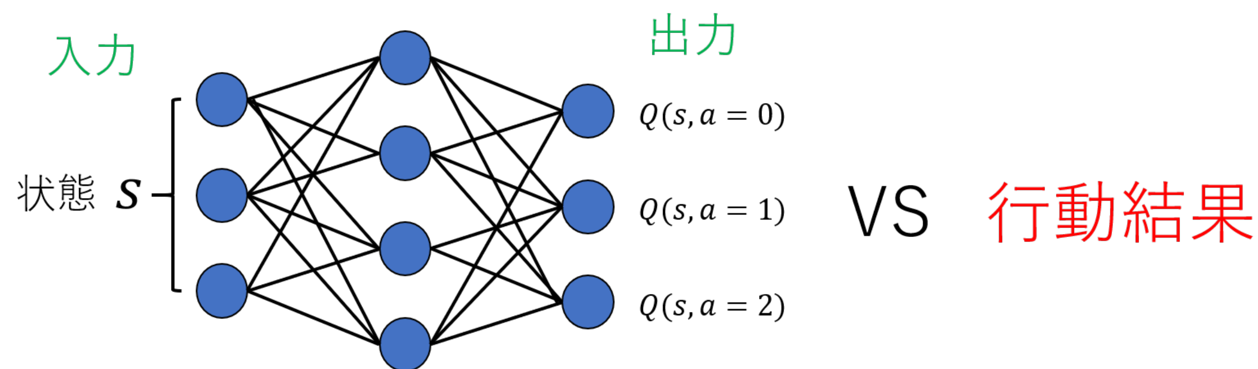
$$E = \frac{1}{2}\left(Q(s,a,\theta) - \left(R(s,a) + \gamma \max_{a'} Q(s',a',\theta)\right)\right)^2$$

with respect to the predicted value:

$$\frac{\partial E}{\partial Q(s,a,\theta)} = Q(s,a,\theta) - \left(R(s,a) + \gamma \max_{a'} Q(s',a',\theta)\right),$$

we can calculate $\frac{\partial E}{\partial \theta}$. Hence, for example, with stochastic gradient descent, we can update θ to

$$\theta \leftarrow \theta - \lambda \frac{\partial E}{\partial \theta}.$$



Some tips in DQN

The above is the principle of DQN learning, but it is inefficient and unstable as it is. There are several ideas proposed, but the following three are typical.

- Experience replay: We store records of actions and randomly select from them to make mini-batches for learning. A single action can be used several times for learning, and it can be stabilized by mixing it with past records.
- Fixed target Q-Network: The parameters of the network used for prediction are updated each time, while the parameters used for the supervised data θ are not updated each time, but copied from those of the network used for prediction at regular intervals. It seems that stabilizing the supervised data also stabilizes the training.
- Gradient clipping: Ensures that the absolute value of the slope of the error function does not exceed a certain value.
  For example, if the constant value is 1, then instead of $E = (\hat{y} - y)^2$ ($\hat{y}$: predicted value, y: supervised value), using
  $$E = \begin{cases} (\hat{y} - y)^2 & (|\hat{y} - y| \leq 1) \\ |\hat{y} - y| & (|\hat{y} - y| > 1) \end{cases}$$
  prevents the gradient $\dfrac{\partial E}{\partial \hat{y}}$ from becoming too large.

Algorithm of DQN

```
Generate Q-Network Q and set the target network to QT = Q
for episode = 1 to max_episode:
    Set the initial state s
    while not game over:
        Determine the action (s,a) using the epsilon-greedy method
        Perform an action and decide the next state s', immediate reward r
                                    and whether the gameover is true or false
        Add s, s', r, and game over true/false to memory
        if number recorded in memory > a fixed number:
            Select a mini batch from memory
            One data in X_batch: x = s
            One data in Y_batch: y = r + (1-discount_rate) * max(QT(s',a'), axis=1)
                        (where y = r if the game is over)
            Update Q using the gradient (with clipping) of Q from (X_batxh, Y_batch)
        Update periodically as QT = Q
        s = s'
```

5.1 Exercises on DQN

Notebooks:

1. `4-7-1_DQN_CartPole.ipynb`
2. `4-7-2_DQN_Breakout.ipynb`

Let's solve the games CartPole and Breakout using DQN.

CartPole is a game in which the player moves a cart left and right to prevent a stick standing on the cart from falling over. The state is in four dimensions (position of the cart, velocity of the cart, angle of the stick, angular velocity of the stick), and there are two types of actions: 0 : push the cart to the left, 1 : push the cart to the right.

Breakout, on the other hand, is a game in which the player bounces falling balls with blocks that move left and right to break the blocks on the top. The state is the game screen itself, and there are four types of actions: 0 : stop, 1 : fire, 2 : move left, 3 : move right.

In this implementation, we use "gym" a library provided by OpenAI as the game environment. For the DQN implementation, we use Keras. Fixed target Q-Network and gradient clipping are not used in Notebook 1. In Notebook 2, we use the Keras-rl library. [https://github.com/keras-rl/keras-rl].