Generate dinosaur names with RNN

Background

- データの紹介

dino.txt

```
['turiasaurus',
 'pandoravenator',
 'ilokelesia',
 'chubutisaurus',
 'quaesitosaurus',
 'orthomerus',
 'selimanosaurus',
 'thecocoelurus',
 'postosuchus',
 'lirainosaurus',
 'acheroraptor',
 'ignavusaurus',
 'koreanosaurus',
```

1537 samples

- 恐竜の名前にはパターンがある

1. 名前は意味のあるwordで構成

   ex) dinosaur → dino + saur
   巨大　トカゲ

   ex) tyrannosaurus → tyranno + saurus
   暴君　　トカゲ

2. 各wordは母音と子音で構成

   ex) dino → d + i + n + o

3. 子音と母音はセットで出る

目的：RNNに恐竜の名前のパターンを学習させる
→適当な入力を与えると名前を生成してくれる

Preprocessing

- Alphabet to number

\ n = 0
a = 1
b = 2
c = 3
d = 4

...

z = 26

- Number to one-hot vector

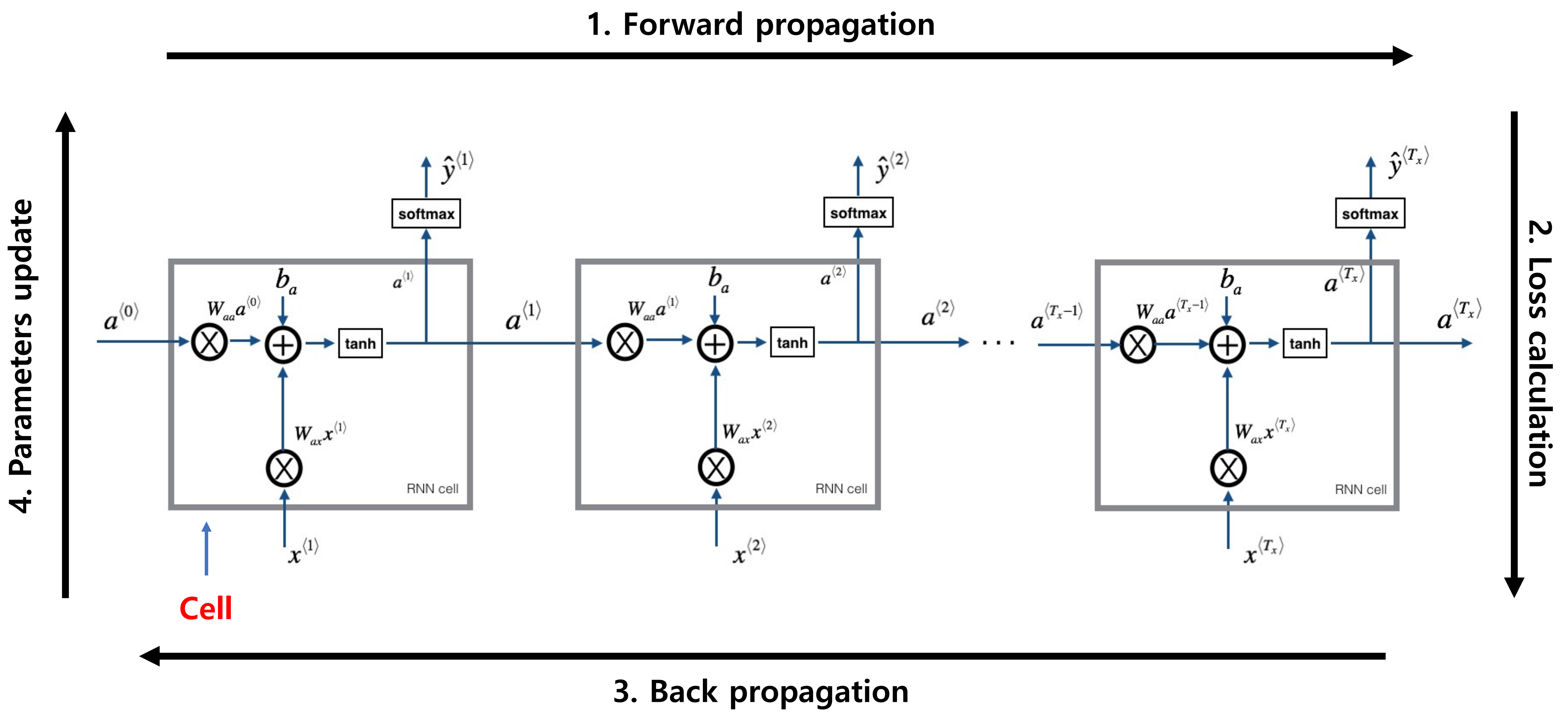
\ n = $[0, 0, 0, \dots, 0]^T$
a = $[0, 1, 0, \dots, 0]^T$
b = $[0, 0, 1, \dots, 0]^T$
...

z = $[0, 0, 0, \dots, 1]^T$

- Example of the word 'dinosaur'

$$
\begin{array}{c c}
 & \begin{array}{cccccccc} d & i & n & o & s & a & u & r \end{array} \\
n\_x = 27 \left\{ \vphantom{\begin{array}{c}0\\0\\0\\0\\1\\ \vdots\\0\end{array}} \right. &
\underbrace{\begin{array}{cccccccc}
\begin{bmatrix}0\\0\\0\\0\\1\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\0\\0\\ \vdots\\1\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\0\\0\\ \vdots\\1\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\0\\0\\ \vdots\\1\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\0\\0\\ \vdots\\1\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\1\\0\\ \vdots\\0\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\0\\0\\ \vdots\\1\\ \vdots\\0\end{bmatrix} &
\begin{bmatrix}0\\0\\0\\ \vdots\\1\\ \vdots\\0\end{bmatrix}
\end{array}}_{T\_x = 8}
\end{array}
$$

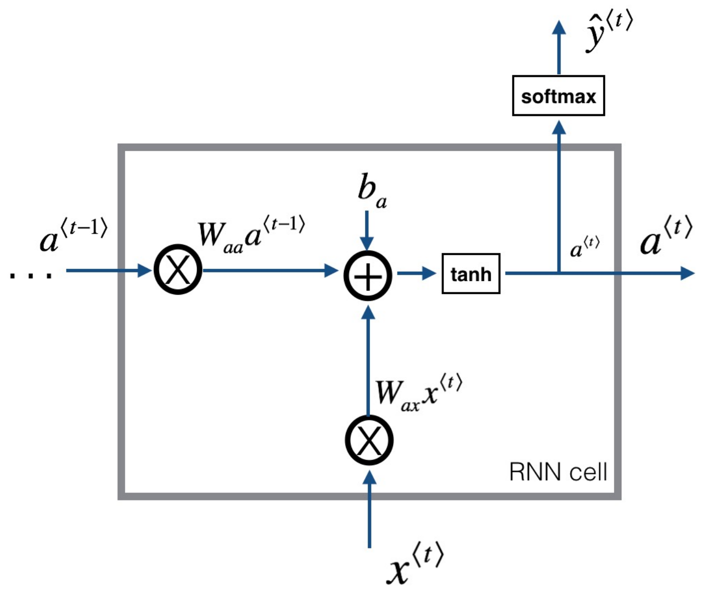
Algorithm



1. Forward propagation
2. Loss calculation
3. Back propagation
4. Parameters update
Cell
$a^{\langle 0 \rangle}$
$W_{aa}a^{\langle 0 \rangle}$
b_a
tanh
$a^{\langle 1 \rangle}$
softmax
$\hat{y}^{\langle 1 \rangle}$
$W_{ax}x^{\langle 1 \rangle}$
$x^{\langle 1 \rangle}$
RNN cell
$W_{aa}a^{\langle 1 \rangle}$
$a^{\langle 2 \rangle}$
$\hat{y}^{\langle 2 \rangle}$
$W_{ax}x^{\langle 2 \rangle}$
$x^{\langle 2 \rangle}$
$a^{\langle T_x-1 \rangle}$
$W_{aa}a^{\langle T_x-1 \rangle}$
$a^{\langle T_x \rangle}$
$\hat{y}^{\langle T_x \rangle}$
$W_{ax}x^{\langle T_x \rangle}$
$x^{\langle T_x \rangle}$

1. Forward propagation (cell)

- **RNN cell structure**



$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$
$$\hat{y}^{\langle t \rangle} = softmax(W_{ya}a^{\langle t \rangle} + b_y)$$

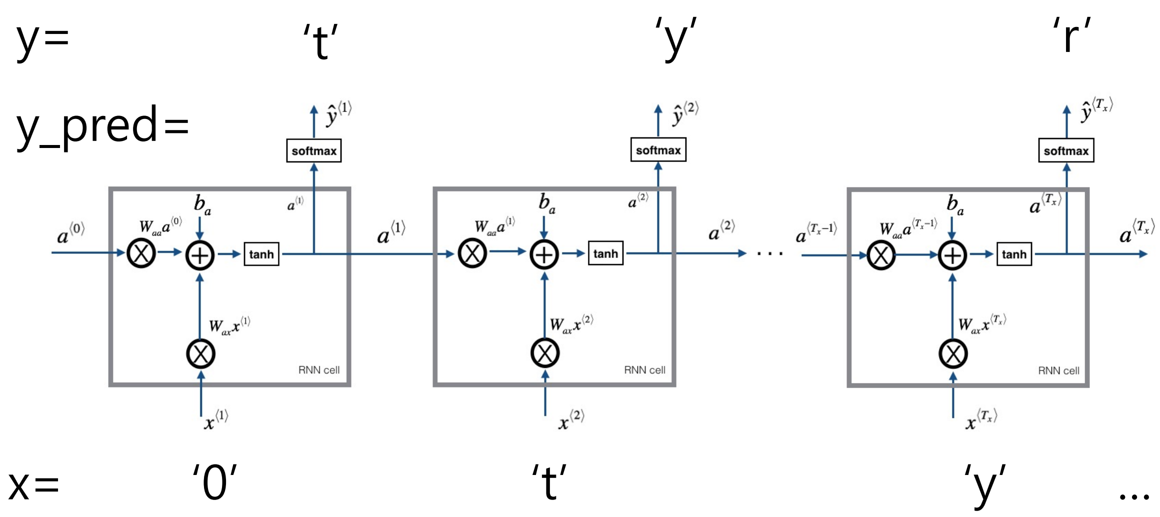
- **Python code**

```
# 3rd method
def rnn_cell_forward(self, xt, a_prev):
    # Retrieve parameters from "parameters"
    Wax = self.parameters["Wax"]
    Waa = self.parameters["Waa"]
1.  Wya = self.parameters["Wya"]
    ba = self.parameters["ba"]
    by = self.parameters["by"]

    # compute next activation state using the formula given above
2.  a_next = np.tanh(np.dot(Waa, a_prev) + np.dot(Wax, xt) + ba)
    # compute output of the current cell using the formula given above
    yt_pred = softmax(np.dot(Wya, a_next) + by)

    # store values you need for backward propagation in cache
3.  cache = (a_next, a_prev, xt)
    return a_next, yt_pred, cache
```

1. パラメータ初期化
2. $a^{\langle t \rangle}$, $\hat{y}^{\langle t \rangle}$ を計算
3. $a^{\langle t \rangle}$, $a^{\langle t-1 \rangle}$, $\hat{y}^{\langle t \rangle}$ をstore

1. Forward propagation (RNN)

- **RNN structure**

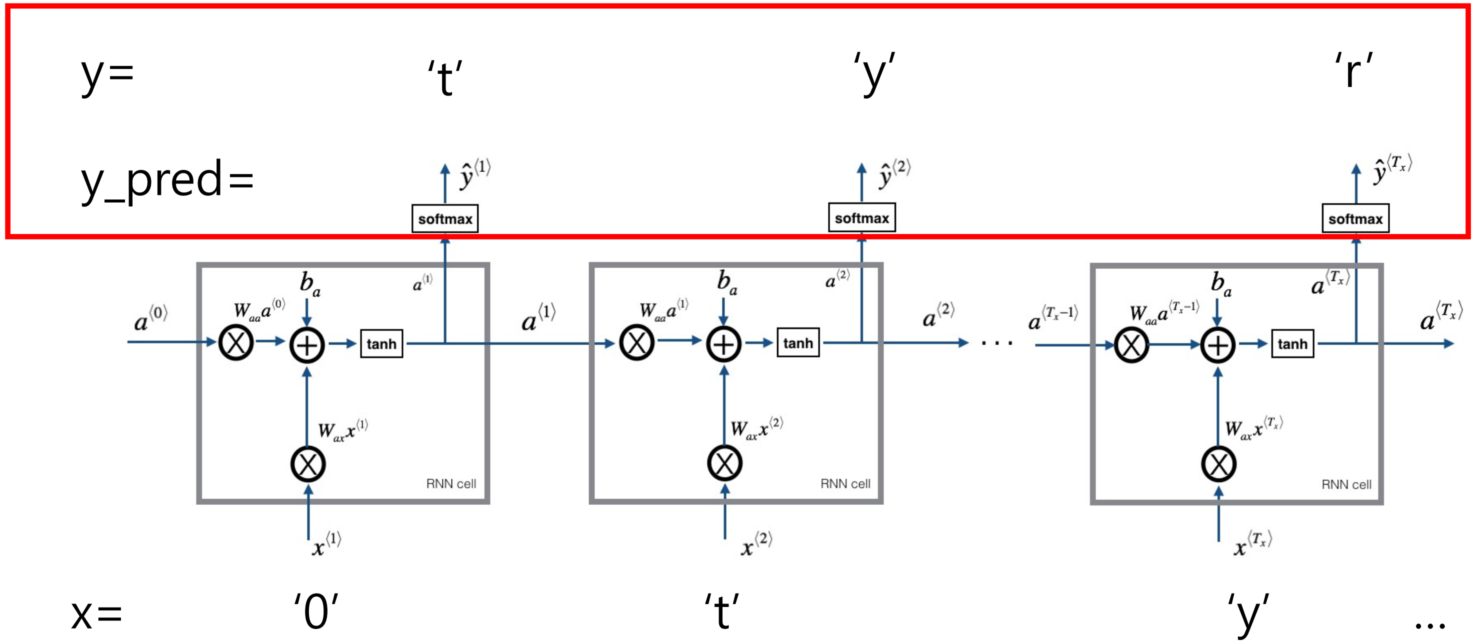


1. パラメータの設定
2. 変数をstoreする変数を宣言
3. 1番目から T_x番目のcellの計算を行う
   $a^{\langle t \rangle}$, $\hat{y}^{\langle t \rangle}$ を計算

- **Python code**

```python
# 4th method
def rnn_forward(self, x, a_prev):

    # Initialize "caches" which will contain the list of all caches
    caches = []

    # Retrieve dimensions from shapes of x and parameters["Wya"]
    n_x, m, T_x = x.shape
    n_y, n_a = self.parameters["Wya"].shape

    # initialize "a" and "y" with zeros
    a = np.zeros((n_a, m, T_x))
    y_pred = np.zeros((n_y, m, T_x))

    # Initialize a_next (≈1 line)
    a_next = a_prev

    # loop over all time-steps
    for t in range(T_x):
        # Update next hidden state, compute the prediction, get the cache
        a_next, yt_pred, cache = self.rnn_cell_forward(xt=x[:, :, t], a_prev=a_next)
        # Save the value of the new "next" hidden state in a (≈1 line)
        a[:, :, t] = a_next
        # Save the value of the prediction in y (≈1 line)
        y_pred[:, :, t] = yt_pred

        # Append "cache" to "caches" (≈1 line)
        caches.append(cache)

    # store values needed for backward propagation in cache
    caches = (caches, x)

    return a, y_pred, caches
```

2. Loss calculation



- 損失関数 (Cross entropy function)

$$H(p,q) = -\sum_x p(x)\,\log q(x).$$

y   y_pred

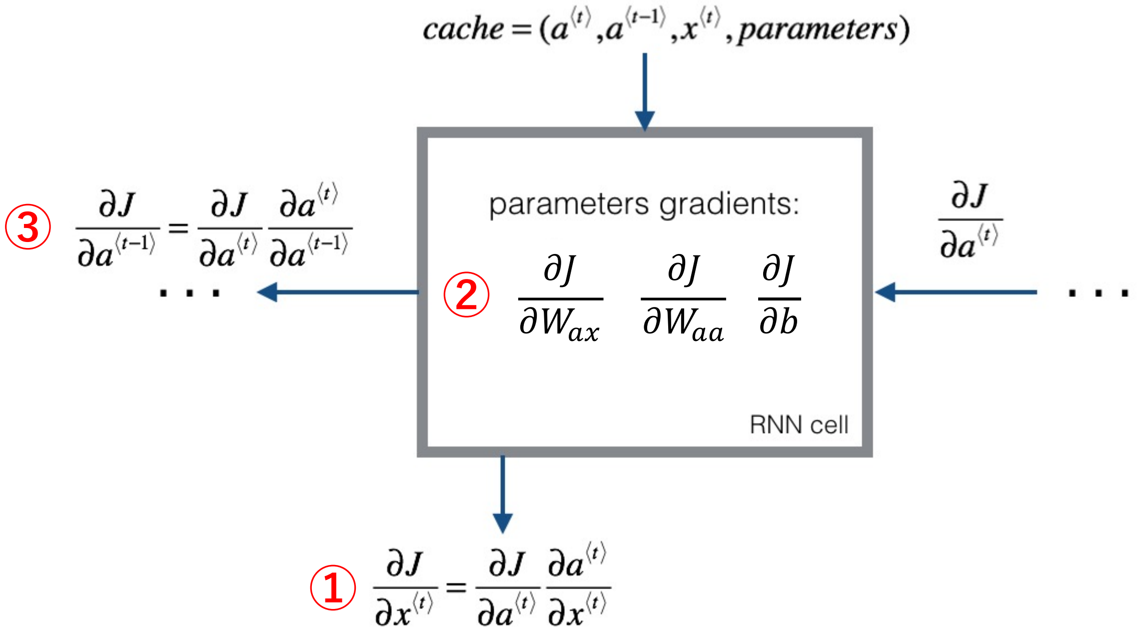
ex) $p(x)\,\log q(x).$

't'   y_pred_1

```
# 5th method
def compute_loss(self, y_hat, y):
    """
    損失関数の定義
    損失関数 -- Cross Entropy
    """
    n_y, m, T_x = y.shape
    for t in range(T_x):
        self.loss -= 1/m * np.sum(np.multiply(y[:, :, t], np.log(y_hat[:, :, t])))
    return self.loss
```

y   y_pred

3. Backward propagation (cell)



$$a^{\langle t \rangle} = \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2$$

$$\left.\begin{aligned}
\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} &= (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) x^{\langle t \rangle T} \\
\frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} &= (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) a^{\langle t-1 \rangle T} \\
\frac{\partial a^{\langle t \rangle}}{\partial b} &= \sum_{batch} (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)
\end{aligned}\right\} ②$$

$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}{}^{T} . (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) \longrightarrow ①$$

$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}{}^{T} . (1 - \tanh(W_{ax} x^{\langle t-1 \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) \longrightarrow ③$$

①

$$\frac{\partial J}{\partial x^{\langle t \rangle}} = \frac{\partial J}{\partial a^{\langle t \rangle}} * \frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}}$$

$$\frac{\partial J}{\partial a^{\langle t \rangle}} = da_{next}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}^{T}(1 - a^{\langle t \rangle 2})$$

$$\frac{\partial J}{\partial x^{\langle t \rangle}} = W_{ax}^{T}\underline{(1 - a^{\langle t \rangle 2}) da_{next}}$$

```
dtanh = (1- a_next**2) * da_next
dxt = np.dot(Wax.T, dtanh)
```

②

$$\frac{\partial J}{\partial W_{ax}} = \frac{\partial J}{\partial a^{\langle t \rangle}} * \frac{\partial a^{\langle t \rangle}}{\partial W_{ax}}$$

$$\frac{\partial J}{\partial a^{\langle t \rangle}} = da_{next}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} = (1 - a^{\langle t \rangle 2}) x^{\langle t \rangle T}$$

$$\frac{\partial J}{\partial x^{\langle t \rangle}} = (1 - a^{\langle t \rangle 2}) da_{next} x^{\langle t \rangle T}$$

```
dWax = np.dot(dtanh, xt.T)
```

③

$$\frac{\partial J}{\partial a^{\langle t-1 \rangle}} = \frac{\partial J}{\partial a^{\langle t \rangle}} * \frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}}$$

$$\frac{\partial J}{\partial a^{\langle t \rangle}} = da_{next}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}(1 - a^{\langle t \rangle 2})$$

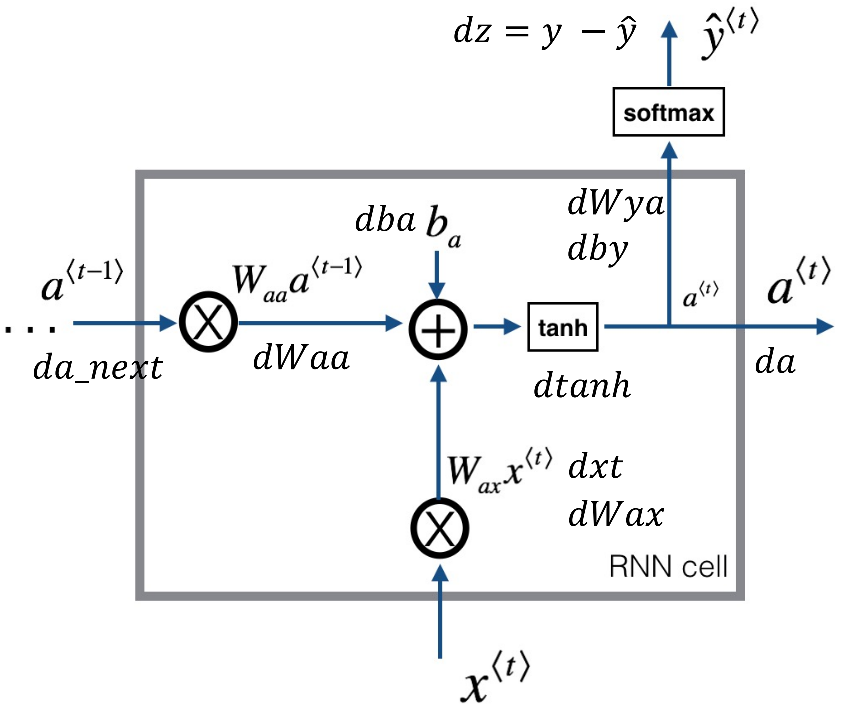
$$\frac{\partial J}{\partial a^{\langle t-1 \rangle}} = W_{aa}(1 - a^{\langle t \rangle 2}) da_{next}$$

```
def rnn_cell_backward(da_next, cache):
```

```
da_prev = np.dot(Waa.T, dtanh)
```

3. Backward propagation (cell)

- **RNN cell structure**



1. パラメータの設定
2. Gradientを計算

- **Python code**

```
# 6th method
def rnn_cell_backward(self, dz, gradients, cache):

    # Retrieve values from cache
    (a_next, a_prev, xt) = cache
```

1

```
    # Retrieve values from parameters
    Wax = self.parameters["Wax"]
    Waa = self.parameters["Waa"]
    Wya = self.parameters["Wya"]
    ba = self.parameters["ba"]
    by = self.parameters["by"]
```

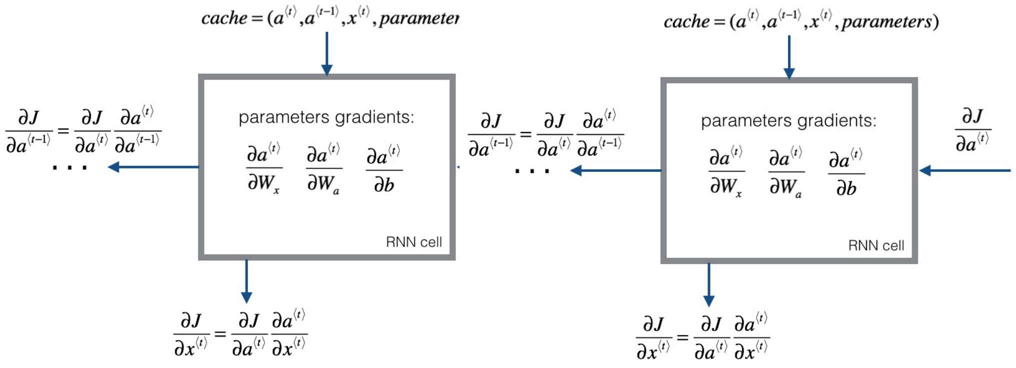
2

```
    gradients['dWya'] += np.dot(dz, a_next.T)
    gradients['dby'] += np.sum(dz, axis=1, keepdims=True)
    da = np.dot(Wya.T, dz) + gradients['da_next']

    # compute the gradient of tanh with respect to a_next (≈1 line)
    dtanh = np.multiply(da, 1 - np.square(a_next))
    # compute the gradient of the loss with respect to Wax (≈2 lines)
    gradients['dxt'] = np.dot(Wax.T, dtanh)
    gradients['dWax'] += np.dot(dtanh, xt.T)

    # compute the gradient with respect to Waa (≈2 lines)
    gradients['dWaa'] += np.dot(dtanh, a_prev.T)

    # compute the gradient with respect to b (≈1 line)
    gradients['dba'] += np.sum(dtanh, axis=1, keepdims=True)

    # compute the gradient with respect to da_next
    gradients['da_next'] = np.dot(Waa.T, dtanh)
```

```
    return gradients
```

3. Backward propagation (RNN)

- **RNN cell structure**



- **Python code**

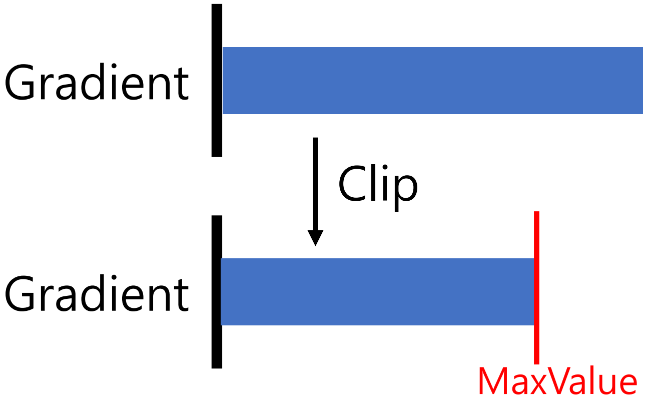
```python
# 7th method
def rnn_backward(self, y, y_hat, caches):

    # Retrieve values from the first cache (t=1) of caches
    (caches, x) = caches
    n_x, m, T_x = x.shape
    # initialize the gradients with the right sizes
    gradients = {}
    dx = np.zeros((n_x, m, T_x))
    gradients['dWax'] = np.zeros((self.n_a, self.n_x))
    gradients['dWaa'] = np.zeros((self.n_a, self.n_a))
    gradients['dba'] = np.zeros((self.n_a, 1))
    gradients['da_next'] = np.zeros((self.n_a, self.m))
    gradients['dWya'] = np.zeros((self.n_y, self.n_a))
    gradients['dby'] = np.zeros((self.n_y, 1))
    dz = y_hat - y  # y_hat=softmax(z), dz=dl/dy_hat * dy_hat/dz

    # Loop through all the time steps
    for t in reversed(range(T_x)):
        gradients = self.rnn_cell_backward(dz=dz[:, :, t], gradients=gradients, cache=caches[t])
        dx[:, :, t] = gradients["dxt"]

    return gradients
```

1. Gradientをstoreする変数を宣言
2. T__x番目のcellから1番目のcellまでGradientを計算

4. Parameter update

- **Clip function**

Gradientが爆発することを防ぐ



- **Update equation**

$$\theta_{new} = \theta_{old} - \eta * \frac{\partial \theta}{\partial J}$$

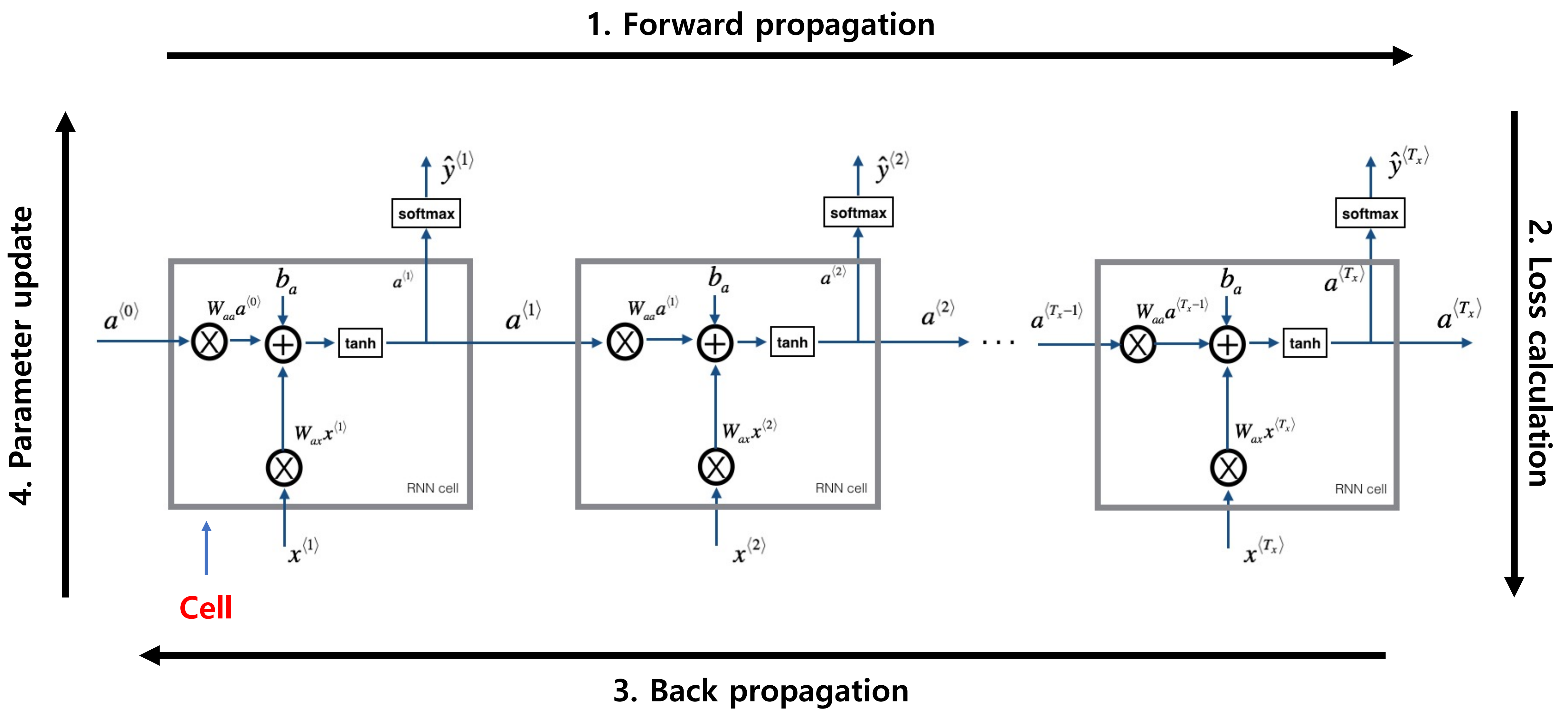
学習率 Gradient

- **Python code**

```python
# 8th method
def clip(self, gradients, maxValue=5):
    """
    Clips the gradients' values between minimum and maximum.
    Arguments:
    gradients -- a dictionary containing the gradients "dWaa", "dWax", "dWya", "db",
    maxValue -- everything above this number is set to this number, and everything l
    Returns:
    gradients -- a dictionary with the clipped gradients.
    """

    dWaa, dWax, dWya, dba, dby = gradients['dWaa'], gradients['dWax'], gradients['dW

    # clip to mitigate exploding gradients, loop over [dWax, dWaa, dWya, db, dby]. (
    for gradient in [dWax, dWaa, dWya, dba, dby]:
        np.clip(gradient, -1*maxValue, maxValue, out=gradient)

    gradients = {"dWaa": dWaa, "dWax": dWax, "dWya": dWya, "dba": dba, "dby": dby}
```

```python
# 9th method
def update_parameters(self, gradients):
    """
    パラメータをアップデート
    """
    self.parameters['Wax'] += -self.alpha * gradients['dWax']
    self.parameters['Waa'] += -self.alpha * gradients['dWaa']
    self.parameters['Wya'] += -self.alpha * gradients['dWya']
    self.parameters['ba'] += -self.alpha * gradients['dba']
    self.parameters['by'] += -self.alpha * gradients['dby']
```

Function ‘optimize’



1-4をuserが指定した回数(num_ter)実行する

Generate names using a trained model

- **Function ‘sample’**

‘a’

[0, 1, … , 0, 0, 0, 0]

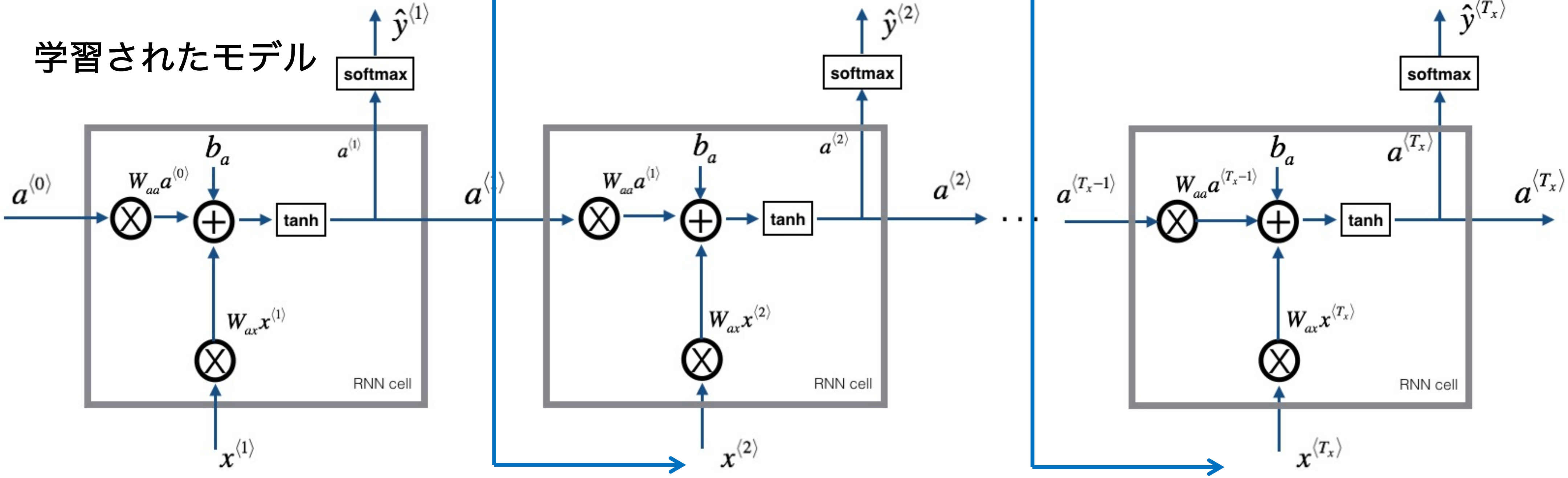
Random choice with $\hat{y}^{\langle 1 \rangle}$ ↑

$\hat{y}^{\langle 1 \rangle}$ = [0.03, 0.03, …, 0.03]

‘z’

[0, 0, … , 0, 0, 0, 1]

学習されたモデル



＼n を入力

→ [0, 0, 0, 0, 0, … , 0]

[0, 1, … , 0, 0, 0, 0]

[0, 0, … , 0, 0, 0, 1]

長さが50になる or ＼n が出る → 計算終了