

Machine Learning in Medicine

# Sequence Generation with ONLY Numpy

---

Keywords : Sequence Models, Sequence Generation, Deep Learning, RNN, Numpy

SungKyunkwan Univ.

Samsung Advanced Institute for  
Health Sciences & Technology

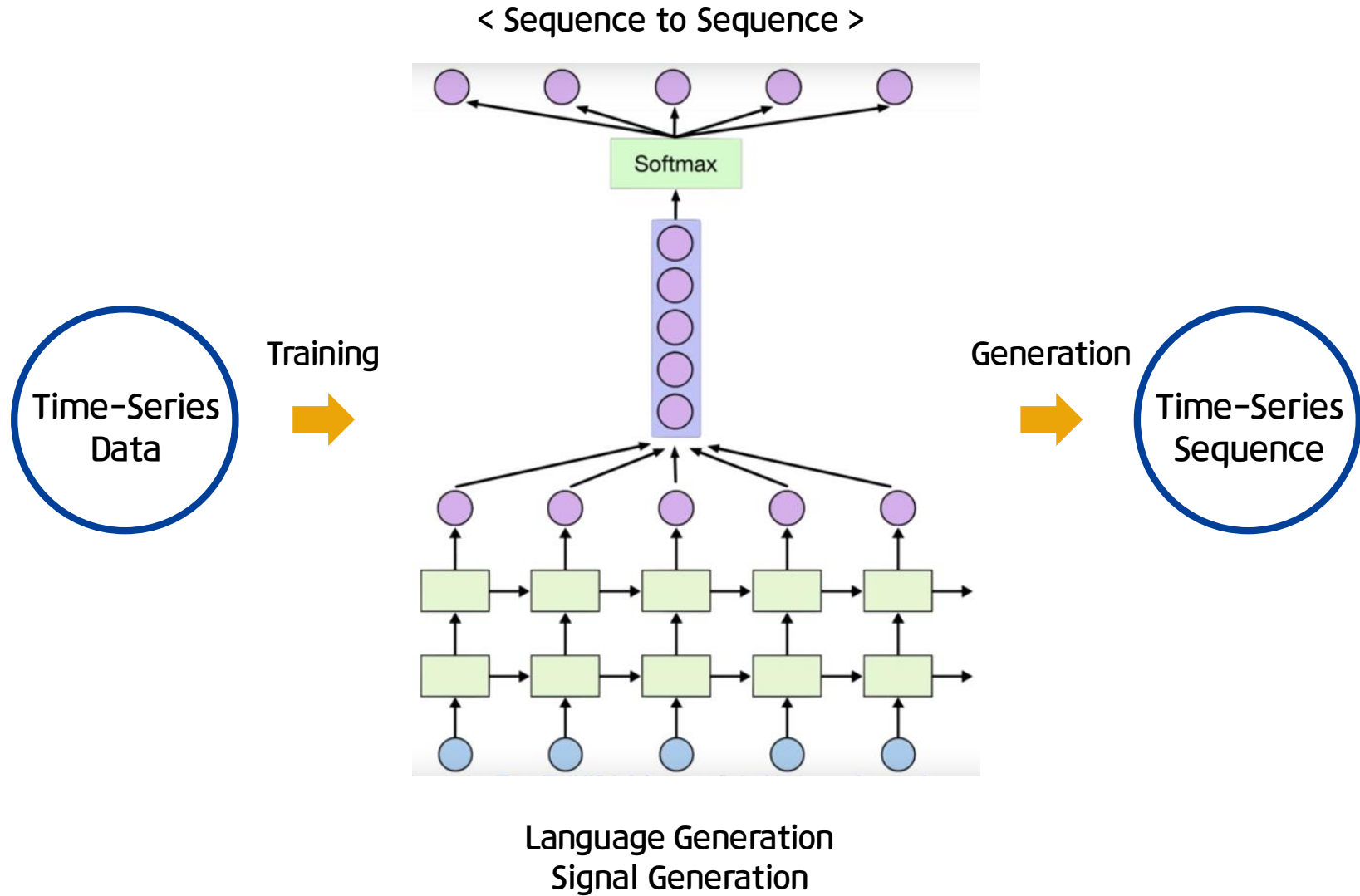
Department of Digital Health

**Mincheol Kim**

BMI Lab.

# seq2seq – Numpy

## Overview



## seq2seq – Numpy

# Source Code

---


### – GitHub URL

<https://github.com/mnchl-kim/seq2seq-Numpy>

<https://github.com/DHC5036/2019-fall-project-mnchl-kim>

- **utils.py** : Includes several necessary function for running the other source code
- **model.py** : class RNN (stacked Recurrent Neural Networks) with ONLY Numpy
- **train.py** : Train data with RNN class
- **test.py** : Test file for live demo

To test the stacked RNN model, I used text data in online.

 character-level language generation.

# seq2seq – Numpy model.py

---

## class RNN:

Stacked Recurrent Neural Networks with ONLY Numpy

The following parameters can be selected in the RNN class.

- input size
- output size
- hidden unit size
- time length
- depth size
- batch size
- dropout rate
- learning rate

Additional information for RNN class

- Weight initialization : Xavier initialization
- Weight update optimizer : Adagrad, RMSProp
- Dropout

## class LSTM:

- ing...
- You can find out the source code at './models/lstm.py'

# seq2seq – Numpy model.py

## Weight initialization

### : Xavier initialization

We initialized the biases to be 0 and the weights  $W_{ij}$  at each layer with the following commonly used heuristic:

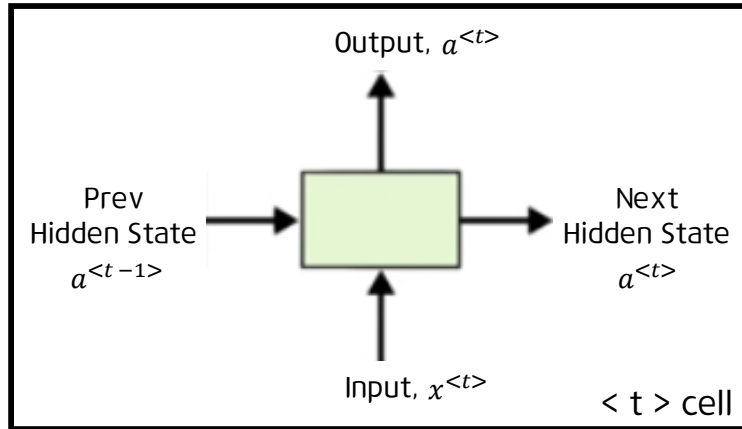
$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], \quad (1)$$

where  $U[-a, a]$  is the uniform distribution in the interval  $(-a, a)$  and  $n$  is the size of the previous layer (the number of columns of  $W$ ).

```
def initialize_xavier(first, second):  
    """  
    Xavier initialization  
  
    Arguments:  
        first -- first dimension size  
        second -- second dimension size  
  
    Returns:  
        W -- Weight matrix initialized by Xavier method  
    """  
  
    sd = np.sqrt(2.0 / (first + second))  
    W = np.random.randn(first, second) * sd  
  
    return W
```

# seq2seq – Numpy model.py

## Stacked Recurrent Neural Networks



### < Forward Propagation >

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

At last layer,

$$\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$$

### Multi-class Cross Entropy Loss

$$L(X_i, Y_i) = - \sum_{j=1}^c y_{ij} * \log(p_{ij})$$

where  $Y_i$  is one – hot encoded target vector  $(y_{i1}, y_{i2}, \dots, y_{ic})$ ,

$$y_{ij} = \begin{cases} 1, & \text{if } i_{th} \text{ element is in class } j \\ 0, & \text{otherwise} \end{cases}$$

$$p_{ij} = f(X_i) = \text{Probability that } i_{th} \text{ element is in class } j$$

### < Backward Propagation >

i ) Derivative of Cross Entropy Loss, Softmax

$$P_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \quad L_i = - \sum_k p_{i,k} \log P_k \quad f_m = (x_i W)_m$$

$$\text{when } k = m, \quad \frac{\partial P_k}{\partial f_m} = \frac{e^{f_k} \sum_j e^{f_j} - e^{f_k} \cdot e^{f_k}}{(\sum_j e^{f_j})^2} = P_k(1 - P_k)$$

$$\text{when } k \neq m, \quad \frac{\partial P_k}{\partial f_m} = - \frac{e^{f_k} e^{f_m}}{(\sum_j e^{f_j})^2} = -P_k P_m$$

then:

$$\begin{aligned} \frac{\partial L_i}{\partial f_m} &= - \sum_k p_{i,k} \frac{\partial \log P_k}{\partial f_m} \\ &= - \sum_k p_{i,k} \frac{1}{P_k} \frac{\partial P_k}{\partial f_m} \\ &= - \sum_{k=m} p_{i,k} \frac{1}{P_k} P_k (1 - P_k) + \sum_{k \neq m} p_{i,k} \frac{1}{P_k} P_k P_m \\ &= \sum_{k \neq m} p_{i,k} P_m - \sum_{k=m} p_{i,k} (1 - P_k) \\ &= \begin{cases} P_m & , \quad m \neq y_i \\ P_m - 1 & , \quad m = y_i \end{cases} \\ &= P_m - p_{i,m} \end{aligned}$$

# seq2seq – Numpy model.py

## Forward Propagation

```
def forward(self, X):
    self._loss = 0

    x, y_hat = [{}, {} for d in range(self._depth_size + 1)], {}
    a = [{-1: np.copy(self._parameters['a'][d])} for d in range(self._depth_size)]
    dropout = [{}, {} for d in range(self._depth_size)]

    for t in range(self._cell_length):
        x[0][t] = X[:, t, :, :].reshape(self._batch_size, self._input_size)

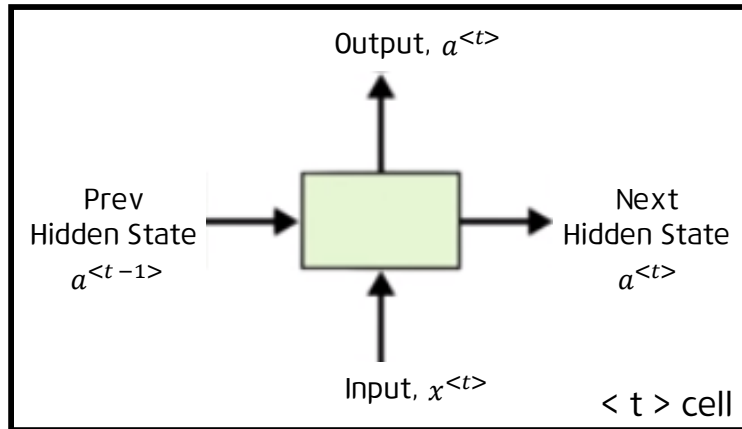
        for d in range(self._depth_size):
            dropout[d][t] = np.random.binomial(1, 1 - self._drop_rate, (1, self._hidden_size)) / (1 - self._drop_rate)
            a[d][t] = np.tanh(np.dot(x[d][t], self._parameters['W_xa'][d]) +
                               np.dot(a[d][t - 1], self._parameters['W_aa'][d]) +
                               self._parameters['b_a'][d])
            x[d + 1][t] = np.copy(a[d][t]) * dropout[d][t]

        z = np.dot(x[self._depth_size][t], self._parameters['W_ay'][0]) + self._parameters['b_y'][0]
        z = np.clip(z, -100, 100)
        y_hat[t] = np.array([softmax(z[b, :]) for b in range(self._batch_size)])

    cache = (x, a, y_hat, dropout)
    return cache
```

# seq2seq – Numpy model.py

## Stacked Recurrent Neural Networks



### < Forward Propagation >

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

At last layer,

$$\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$$

### Multi-class Cross Entropy Loss

$$L(X_i, Y_i) = - \sum_{j=1}^c y_{ij} * \log(p_{ij})$$

where  $Y_i$  is one – hot encoded target vector  $(y_{i1}, y_{i2}, \dots, y_{ic})$ ,

$$y_{ij} = \begin{cases} 1, & \text{if } i_{th} \text{ element is in class } j \\ 0, & \text{otherwise} \end{cases}$$

$$p_{ij} = f(X_i) = \text{Probability that } i_{th} \text{ element is in class } j$$

### < Backward Propagation >

i ) Derivative of Softmax, Cross Entropy Loss

$$P_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \quad L_i = -\sum_k p_{i,k} \log P_k \quad f_m = (x_i W)_m$$

$$\text{when } k = m, \quad \frac{\partial P_k}{\partial f_m} = \frac{e^{f_k} \sum_j e^{f_j} - e^{f_k} \cdot e^{f_k}}{(\sum_j e^{f_j})^2} = P_k(1 - P_k)$$

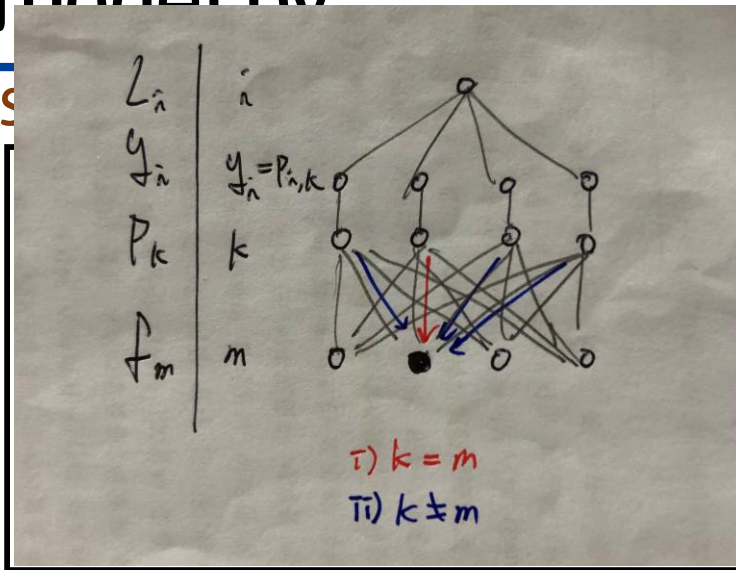
$$\text{when } k \neq m, \quad \frac{\partial P_k}{\partial f_m} = -\frac{e^{f_k} e^{f_m}}{(\sum_j e^{f_j})^2} = -P_k P_m$$

then:

$$\begin{aligned} \frac{\partial L_i}{\partial f_m} &= -\sum_k p_{i,k} \frac{\partial \log P_k}{\partial f_m} \\ &= -\sum_k p_{i,k} \frac{1}{P_k} \frac{\partial P_k}{\partial f_m} \\ &= -\sum_{k=m} p_{i,k} \frac{1}{P_k} P_k(1 - P_k) + \sum_{k \neq m} p_{i,k} \frac{1}{P_k} P_k P_m \\ &= \sum_{k \neq m} p_{i,k} P_m - \sum_{k=m} p_{i,k} (1 - P_k) \\ &= \begin{cases} P_m, & m \neq y_i \\ P_m - 1, & m = y_i \end{cases} \\ &= P_m - p_{i,m} \end{aligned}$$



# seq2seq – Numpy model.py



## < Forward Propagation >

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

At last layer,

$$\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$$

## Multi-class Cross Entropy Loss

$$L(X_i, Y_i) = - \sum_{j=1}^c y_{ij} * \log(p_{ij})$$

where  $Y_i$  is one-hot encoded target vector  $(y_{i1}, y_{i2}, \dots, y_{ic})$ ,

$$y_{ij} = \begin{cases} 1, & \text{if } i_{th} \text{ element is in class } j \\ 0, & \text{otherwise} \end{cases}$$

$$p_{ij} = f(X_i) = \text{Probability that } i_{th} \text{ element is in class } j$$

## orks

## < Backward Propagation >

i) Derivative of Softmax, Cross Entropy Loss

$$P_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \quad L_i = - \sum_k p_{i,k} \log P_k \quad f_m = (x_i W)_m$$

$$\text{when } k = m, \quad \frac{\partial P_k}{\partial f_m} = \frac{e^{f_k} \sum_j e^{f_j} - e^{f_k} \cdot e^{f_k}}{(\sum_j e^{f_j})^2} = P_k(1 - P_k)$$

$$\text{when } k \neq m, \quad \frac{\partial P_k}{\partial f_m} = - \frac{e^{f_k} e^{f_m}}{(\sum_j e^{f_j})^2} = -P_k P_m$$

then:

$$\begin{aligned} \frac{\partial L_i}{\partial f_m} &= - \sum_k p_{i,k} \frac{\partial \log P_k}{\partial f_m} \\ &= - \sum_k p_{i,k} \frac{1}{P_k} \frac{\partial P_k}{\partial f_m} \\ &= - \sum_{k=m} p_{i,k} \frac{1}{P_k} P_k (1 - P_k) + \sum_{k \neq m} p_{i,k} \frac{1}{P_k} P_k P_m \\ &= \sum_{k \neq m} p_{i,k} P_m - \sum_{k=m} p_{i,k} (1 - P_k) \\ &= \begin{cases} P_m, & m \neq y_i \\ P_m - 1, & m = y_i \end{cases} \\ &= P_m - p_{i,m} \end{aligned}$$

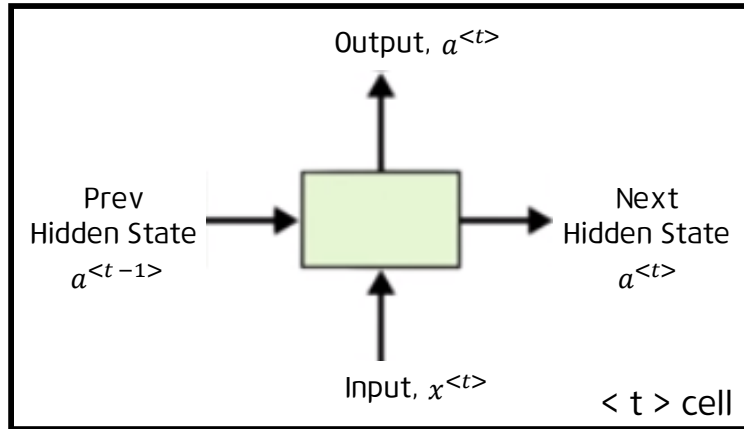
# seq2seq – Numpy model.py

## Forward Propagation

```
def cross_entropy(x, index):  
    """  
    Assumption: the ground truth vector contains only one non-zero component with a value of 1  
    """  
  
    loss = - np.log(x[index]) if x[index] > 0 else 0  
    return loss  
  
def cross_entropy_d(x, index):  
    """  
    Assumption: the ground truth vector contains only one non-zero component with a value of 1  
    """  
  
    x[index] -= 1  
    return x
```

# seq2seq – Numpy model.py

## Stacked Recurrent Neural Networks



### < Forward Propagation >

$$a^{<t>} = \tanh(W_{ax}x^{<t>} + W_{aa}a^{<t-1>} + b_a)$$

At last layer,

$$\hat{y}^{<t>} = \text{soft max}(W_{ya}a^{<t>} + b_y)$$

### < Backward Propagation >

i) Derivative of Softmax, Cross Entropy Loss

$$P_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \quad L_i = -\sum_k p_{i,k} \log P_k \quad f_m = (x_i W)_m$$

$$\text{when } k = m, \quad \frac{\partial P_k}{\partial f_m} = \frac{e^{f_k} \sum_j e^{f_j} - e^{f_k} \cdot e^{f_k}}{(\sum_j e^{f_j})^2} = P_k(1 - P_k)$$

$$\text{when } k \neq m, \quad \frac{\partial P_k}{\partial f_m} = -\frac{e^{f_k} e^{f_m}}{(\sum_j e^{f_j})^2} = -P_k P_m$$

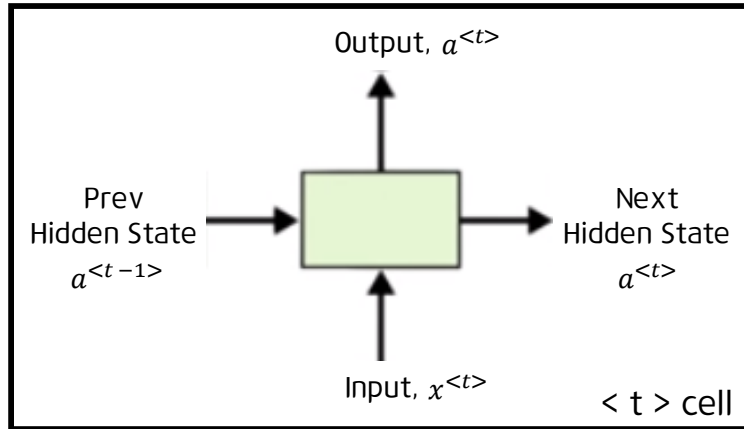
then:

$$\begin{aligned} \frac{\partial L_i}{\partial f_m} &= -\sum_k p_{i,k} \frac{\partial \log P_k}{\partial f_m} \\ &= -\sum_k p_{i,k} \frac{1}{P_k} \frac{\partial P_k}{\partial f_m} \\ &= -\sum_{k=m} p_{i,k} \frac{1}{P_k} P_k (1 - P_k) + \sum_{k \neq m} p_{i,k} \frac{1}{P_k} P_k P_m \\ &= \sum_{k \neq m} p_{i,k} P_m - \sum_{k=m} p_{i,k} (1 - P_k) \\ &= \begin{cases} P_m & , \quad m \neq y_i \\ P_m - 1 & , \quad m = y_i \end{cases} \\ &= P_m - p_{i,m} \end{aligned}$$

$$\frac{\partial L_i}{\partial W_k} = \frac{\partial L_i}{\partial f_m} \frac{\partial f_m}{\partial W_k} = x_i^T (P_m - p_{i,m})$$

# seq2seq – Numpy model.py

## Stacked Recurrent Neural Networks



### < Forward Propagation >

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

At last layer,

$$\hat{y}^{(t)} = \text{soft max}(W_{ya}a^{(t)} + b_y)$$

### < Backward Propagation >

ii) Derivative of  $a^{<t>}$

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

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

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

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

$$\frac{\partial a^{(t)}}{\partial b} = \sum_{batch} (1 - \tanh(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b)^2)$$

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

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

$$\frac{\partial L_i}{\partial W_k} = \frac{\partial L_i}{\partial f_m} \frac{\partial f_m}{\partial W_k} = x_i^T (P_m - p_{i,m})$$

# seq2seq – Numpy model.py

## Backward Propagation

```
def backward(self, Y, cache):
    self._gradients = {key: [np.zeros_like(self._gradients[key][d]) for d in range(len(self._gradients[key]))] for key in self._gradients.keys()}
    (x, a, y_hat, dropout) = cache

    for t in reversed(range(self._cell_length)):
        self._loss += sum([cross_entropy(y_hat[t][b, :], Y[b, t]) for b in range(self._batch_size)]) / (self._cell_length * self._batch_size)
        dy = np.array([cross_entropy_d(y_hat[t][b, :], Y[b, t]) for b in range(self._batch_size)]) / (self._cell_length * self._batch_size)

        self._gradients['dW_ay'][0] += np.dot(x[self._depth_size][t].T, dy)
        self._gradients['db_y'][0] += dy.sum(axis=0)
        da = np.dot(dy, self._parameters['W_ay'][0].T)

        for d in reversed(range(self._depth_size)):
            da = (1 - a[d][t] ** 2) * (da * dropout[d][t] + self._gradients['da'][d])
            self._gradients['dW_xa'][d] += np.dot(x[d][t].T, da)
            self._gradients['dW_aa'][d] += np.dot(a[d][t - 1].T, da)
            self._gradients['db_a'][d] += da.sum(axis=0)
            self._gradients['da'][d] = np.dot(da, self._parameters['W_aa'][d].T)
            da = np.dot(da, self._parameters['W_xa'][d].T)

    self._parameters['a'] = [a[d][self._cell_length - 1] for d in range(self._depth_size)]
```

# seq2seq – Numpy model.py

---

## Weight update

### 1. Adagrad (Adaptive Gradient)

$$G_t = G_{t-1} + (\nabla_{\theta} J(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_{\theta} J(\theta_t)$$

### 2. RMSProp

$$G = \gamma G + (1 - \gamma)(\nabla_{\theta} J(\theta_t))^2$$
$$\theta = \theta - \frac{\eta}{\sqrt{G + \epsilon}} \cdot \nabla_{\theta} J(\theta_t)$$

# seq2seq – Numpy model.py

---

## Weight update

```
def update_parameters(self, learning_rate=0.01):
    parameters = self._parameters['W_xa'] + self._parameters['W_aa'] + self._parameters['W_ay'] + self._parameters['b_a'] + self._parameters['b_y']
    gradients = self._gradients['dW_xa'] + self._gradients['dW_aa'] + self._gradients['dW_ay'] + self._gradients['db_a'] + self._gradients['db_y']
    momentums = self._momentums['dW_xa'] + self._momentums['dW_aa'] + self._momentums['dW_ay'] + self._momentums['db_a'] + self._momentums['db_y']

    for w, g, m in zip(parameters, gradients, momentums):
        np.clip(w, -1, 1, out=w)

        # Adagrad
        m += g ** 2
        w -= learning_rate * g / np.sqrt(m + 1e-8)

        # RMSProp
        # m = 0.9 * m + 0.1 * g ** 2
        # w -= learning_rate * g / np.sqrt(m + 1e-8)
```

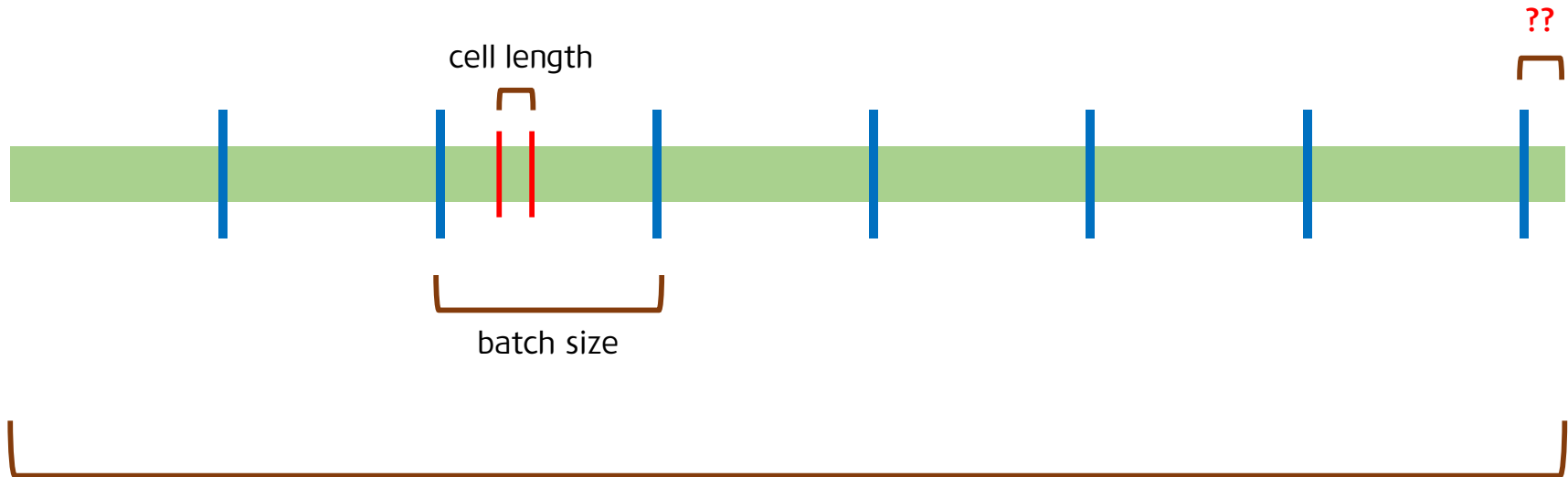
# seq2seq – Numpy train.py

## Input Data

Text Data (The Little Prince.txt)

- Total characters : 93,609
- Unique characters : 81

> Character-level language generation



```
mini_batch_X.shape : (b, t, x, 1)  
mini_batch_Y.shape : (b, t, x, 1)
```

Total characters



# seq2seq – Numpy train.py

## Input Data

```
# Split text by mini-batch with batch_size
batch_length = data_size // batch_size
for i in range(0, batch_length - seq_length, seq_length):
    mini_batch_X, mini_batch_Y = [], []

    for j in range(0, data_size - batch_length + 1, batch_length):
        mini_batch_X.append(one_hot(data[j + i:j + i + seq_length], ch2ix))
        mini_batch_Y.append([ch2ix[ch] for ch in data[j + i + 1:j + i + seq_length + 1]])

    mini_batch_X = np.array(mini_batch_X)
    mini_batch_Y = np.array(mini_batch_Y)

    model.optimize(mini_batch_X, mini_batch_Y, learning_rate=learning_rate)
```

mini\_batch\_X.shape : (b, t, x, 1)  
mini\_batch\_Y.shape : (b, t, x, 1)

# seq2seq – Numpy train.py

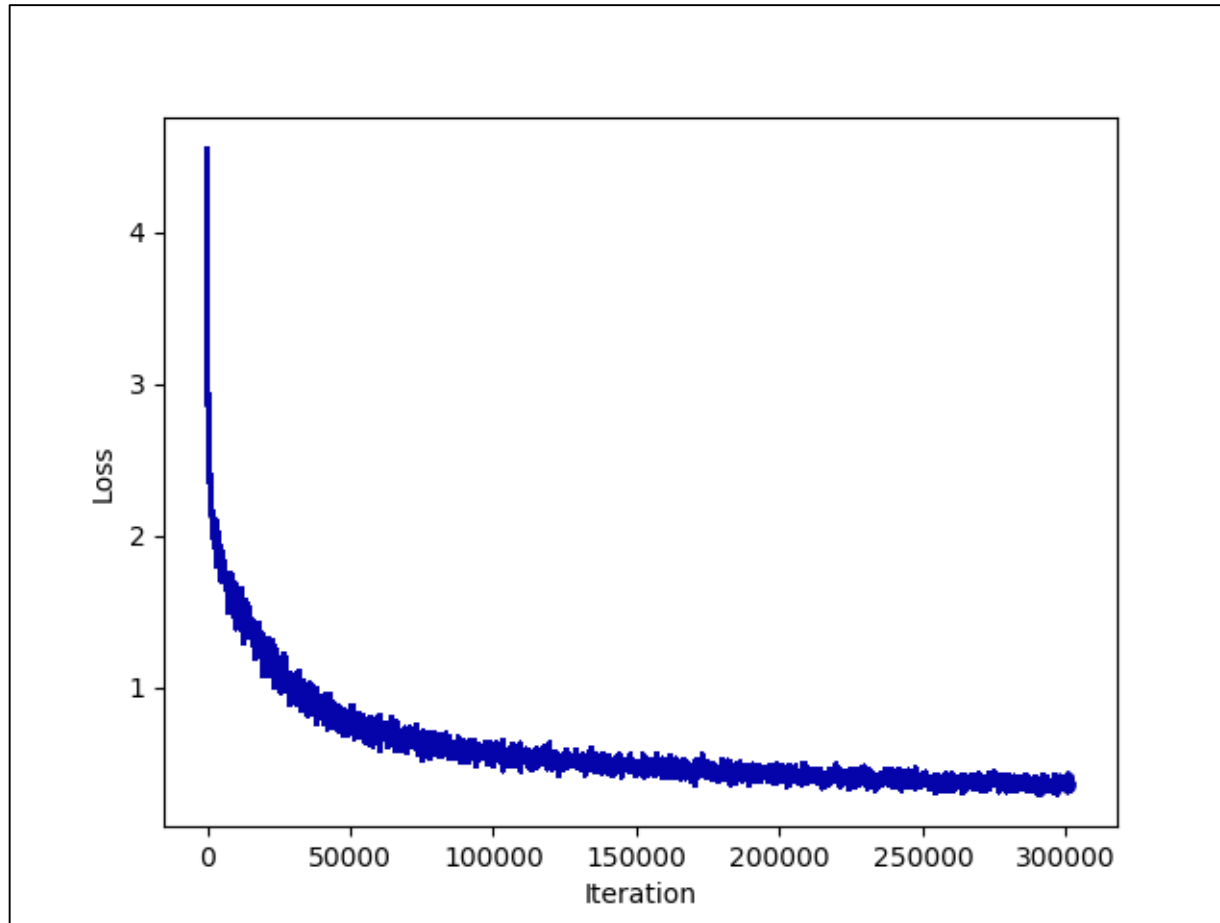
---

## Training Process

- Total iteration : 3,000
- Input size : 81
- Output size : 81
- Hidden unit size : 256
- Time length : 100
- Depth size : 3
- Batch size : 10
- Dropout rate : 0.1
- Learning rate : 0.01
- Optimizer : Adagrad

# seq2seq – Numpy train.py

## Training Process



# seq2seq – Numpy train.py

---

## Training Process

```
#####  
Total iteration: 1  
Iteration: 1  
Loss: 4.621846  
Time: 0.406110  
  
### Starts Here ###  
  
5i"?VT9H"\Y9R!Bxjn40swaYuT?" KM  
5!Zf'CK3M:3jByo435i-'0e-1o8'3na  
"95q1SIjeihz.hjh?-.Vf:fEAb$eMw"AI"s-:D6ij5A1GK8DW  
"sDrybxIJK;C:-"YjyG  
  
### Ends Here ###  
#####
```

# seq2seq – Numpy train.py

## Training Process

```
#####  
Total iteration: 2944  
Iteration: 273700  
Loss: 0.386915  
Time: 73135.271786  
  
### Starts Here ###  
  
N But his allies make..."  
  
"I chilk nig to little man regrothing har jrapne..." And he lay down in the grass and cried.  
  
It was then that the fox appeared.  
  
"Good morning," said the fox. "Men  
  
### Ends Here ###  
#####
```

## seq2seq – Numpy

# Discussion

---

It can be used for sequence generation.

- Language Generation
- Medical Signal Generation (Ex, ECG)
- Time-series prediction

Further Improvements

- Running Time
- Word Embedding
- LSTM, GRU
- Bidirectional RNN
- Attention
- Bert

# Q&A?

SungKyunkwan Univ.

Samsung Advanced Institute for  
Health Sciences & Technology

Department of Digital Health

**Mincheol Kim**

BMI Lab.