DL Lab4: Back-Propagation Through Time

0756079 陳冠聞

May 8, 2019

1 Introduction

Recurrent neural networks is a specific class of artifical neural network. In traditional feedforward network, input examples are fed to the network and transformed into an output. On the other hand, recurrent neural networks take as their input not just the current input example they see, but also what they have perceived previously in time. This allows it to model the temporal relationship in input data, and it has many application such as speech recognition, machine translation and so on.

In this lab, I will implement the recurrent neural network in python using only *NumPy* for matrix operation, others function are all implemented by myself.

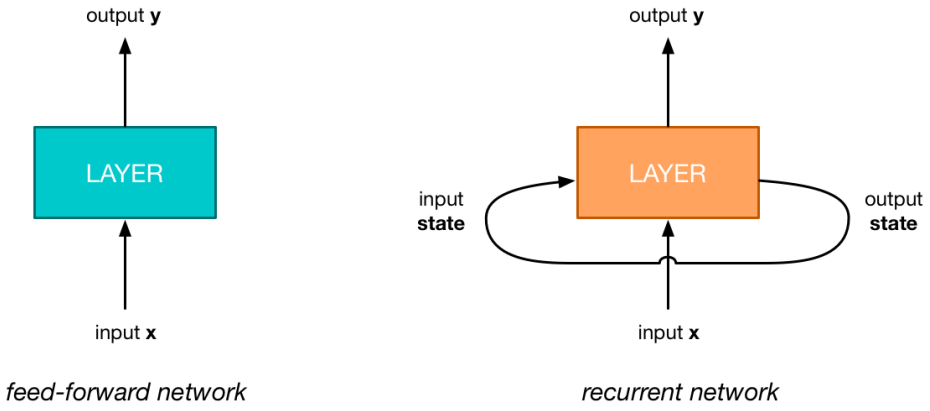

Figure 1: Feedforward v.s. Recurrent Neural Network

2 Recurrent Neural Network

Recurrent neural network is a sequence to sequence mapping between input sequence $x^{(1)}$, $x^{(2)}$, ... , $x^{(\tau)}$ to the target sequence $y^{(1)}$, $y^{(2)}$, ... , $y^{(\tau)}$. In this section, I will explain how the network generate output through forward propagation, and how it can learn from data through back-propagation.

2.1 Forward Propagation

Let the weights of input-to-input, hidden-to-hidden and hidden-to-output connection be U, V and W respectively. In addition, let the hidden state and output unit at timestep t be the $h^{(t)}$ and $o^{(t)}$, the forward propagation at timestep t can be formulated as following

$$\begin{aligned} a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)} \\ h^{(t)} &= tanh(a^{(t)}) \\ o^{(t)} &= c + Vh^{(t)} \end{aligned} \tag{1}$$

where b and c are the bias term of hidden unit and output unit, and $a^{(t)}$ is value of hidden unit before activation function $tanh$. The diagram of the forward propagation can refer to Figure 2, and the implementation code is shown in Figure 5.

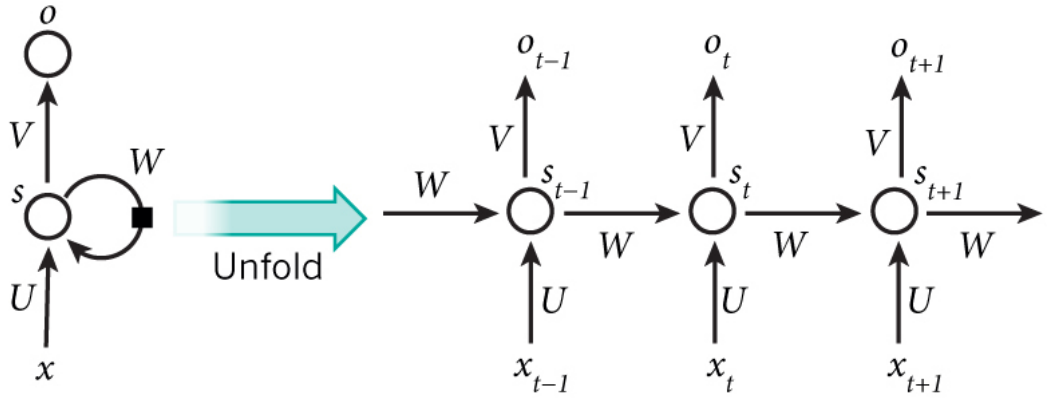

Figure 2: Forward Propagation of RNN

2.2 Back Propagation Through Time

Given the corresponding input sequences and target sequences, we want to learn the model parameters $\theta = \{W, U, V, b, c\}$ such that the output of the learned network given the input sequence will close to the target sequence. We can use the same concept of back-propagation in the feedforward network, but the difference is that the recurrent network has many timesteps, so we have to back-propgate through time like the unfold graph in Figure 2.

The gradient of each model parameter is shown below, and the detail mathematical derivation is left in appendix for the conciseness.

$$
\begin{aligned}
\nabla_c L &= \sum_t (\frac{\partial L}{\partial o^t}) \\
\nabla_b L &= \sum_t H^{(t)} (\frac{\partial L}{\partial h^t}) \\
\nabla_V L &= \sum_t (\frac{\partial L}{\partial o^t}) (h^{(t)})^T \\
\nabla_U L &= \sum_t H^{(t)} (\frac{\partial L}{\partial h^t}) (x^{(t)})^T \\
\nabla_W L &= \sum_t H^{(t)} (\frac{\partial L}{\partial h^t}) (h^{(t-1)})^T
\end{aligned}
\tag{2}
$$

3 Implementation

3.1 Data Generation

For data generation, I use the *numpy.random* to generate two 8-bits numbers x_1 and x_2, then I follow the the binary addition procedure to get the correct answer y. When fed to the network, the input at timestep t is consist of two corresponding bit in x_1 and x_2, and from the least significant bit to the most significant bit.

```
class BinaryAdditionDataGenerator():
    def generate_data(self, n_bits=8):
        x1 = np.random.randint(2, size=n_bits)
        x2 = np.random.randint(2, size=n_bits)
        y  = [0 for i in range(n_bits)]
        carry = 0
        for i, (a, b) in enumerate(zip(x1, x2)):
            y[i] = a + b + carry
            carry = 0
            if y[i] >= 2:
                y[i] %= 2
                carry = 1
        x = np.array([x1, x2]).T
        y = np.array(y)
        return x, y
```

Figure 3: Code Snippet for Data Generation

3.2 Nerual Net

First, for the neural net architecture, I initialize the weights of W, U, V using **Xavier initialization** and set the bias term b and c to all zeros.

```python
class RNN():
    def __init__(self, in_dims=2, hidden_dims=16, out_dims=1):
        # Model dimensions
        self.in_dims = in_dims
        self.hidden_dims = hidden_dims
        self.out_dims = out_dims

        # Model Weights
        self.U = np.random.normal(size=(hidden_dims, in_dims)) * np.sqrt(2/(in_dims+hidden_dims))
        self.V = np.random.normal(size=(out_dims, hidden_dims)) * np.sqrt(2/(hidden_dims+out_dims))
        self.W = np.random.normal(size=(hidden_dims, hidden_dims)) * np.sqrt(2/(hidden_dims+hidden_dims))
        self.b = np.zeros((hidden_dims, 1))
        self.c = np.zeros((out_dims, 1))
        self.zero_grad()

    def zero_grad(self):
        # Set Gradients to zeros
        self.gradient_c = np.zeros((self.c.shape))
        self.gradient_b = np.zeros((self.b.shape))
        self.gradient_V = np.zeros((self.V.shape))
        self.gradient_U = np.zeros((self.U.shape))
        self.gradient_W = np.zeros((self.W.shape))
        self.batch_size = 0

    def update_weight(self, lr=0.1):
        # Update Model parameters
        self.c -= lr * (1/self.batch_size) * self.gradient_c
        self.b -= lr * (1/self.batch_size) * self.gradient_b
        self.V -= lr * (1/self.batch_size) * self.gradient_V
        self.U -= lr * (1/self.batch_size) * self.gradient_U
        self.W -= lr * (1/self.batch_size) * self.gradient_W
        self.zero_grad()
```

Figure 4: Code Snippet for RNN (Init)

Second, for the forward propagation, I follow the Equation 1. I assume the timestep of input sequence start from $t = 1$, and set the dummy variables for timestep $t = 0$ to be consistent with the initial hidden state $h^{(0)}$. In the forward propagation procedure, I record all the intermediate results to do back-propagation.

```
    def forward(self, xs):
        T = len(xs)
        # Record from t=0 to t=T (t=0 is initial state)
        self.x = [None for t in range(0, T+1)]
        self.a = [None for t in range(0, T+1)]
        self.h = [None for t in range(0, T+1)]
        self.o = [None for t in range(0, T+1)]
        self.y = [None for t in range(0, T+1)]

        # Foward Pass
        self.h[0] = np.zeros((self.hidden_dims, 1))
        for i, x in enumerate(xs):
            t = i+1
            self.x[t] = x.reshape(-1, 1)
            self.a[t] = self.b + np.matmul(self.W, self.h[t-1]) + np.matmul(self.U, self.x[t])
            self.h[t] = tanh(self.a[t])
            self.o[t] = self.c + np.matmul(self.V, self.h[t])
            self.y[t] = sigmoid(self.o[t])
        return self.y
```

Figure 5: Code Snippet for RNN (Forward Pass)

Lastly, for the back-propagation, I follow the Equation 2. I compute the gradient of model parameters, and accumulate the gradient, then update the model weights after a mini-batch to achieve batch-training.

```
    def backward(self, y_true, y_pred):
        T = len(self.h) - 1

        # Compute dL/do
        d_o = [None for t in range(0, T+1)]
        for t in range(1, T+1):
            i = t - 1
            d_o[t] = y_pred[t] - y_true[i]

        # Compute H^(t)
        H = [None for t in range(0, T+1)]
        for t in range(1, T+1):
            H[t] = np.zeros((self.hidden_dims, self.hidden_dims))
            for d in range(H[t].shape[0]):
                H[t][d, d] = 1-(self.h[t][d]**2)

        # Compute dL/dh
        d_h = [None for t in range(0, T+1)]
        d_h[-1] = np.matmul(self.V.T, d_o[-1])
        for t in range(T-1, 0, -1):
            d_h[t] = np.matmul(self.W.T, np.matmul(H[t+1], d_h[t+1])) + np.matmul(self.V.T, d_o[t])

        # Compute gradient with respect to model parametes {c, b, V, U, W}
        for t in range(1, T+1):
            self.gradient_c += d_o[t]
            self.gradient_b += np.matmul(H[t], d_h[t])
            self.gradient_V += np.matmul(d_o[t], self.h[t].T)
            self.gradient_U += np.matmul(np.matmul(H[t], d_h[t]), self.x[t].T)
            self.gradient_W += np.matmul(np.matmul(H[t], d_h[t]), self.h[t-1].T)
        self.batch_size += 1
```

Figure 6: Code Snippet for RNN (Back-Propagation)

4 Result

In this lab, I train the network using stochastic gradient decent with batch size 8 as well as learning rate 0.005. I use two criterions to evaluate the model performance. One is average **bit error**, which is the number of different bits between the model prediction and the label. Another is the **accuracy**, which considers the prediction is correct if it is exactly the same as the label for all bits, then compute the number of correct divided by the number of data.

From the Figure 7, we can see that the bit error drop very quickly, and accuracy improve dramatically. In addition, the model can achieve 100% accuracy and zero bit error after 3000 iteration.

```
(Iter 01000) Bit error: 1.812, Accuracy: 16.25%
(Iter 02000) Bit error: 0.025, Accuracy: 97.50%
(Iter 03000) Bit error: 0.000, Accuracy: 100.00%
(Iter 04000) Bit error: 0.000, Accuracy: 100.00%
(Iter 05000) Bit error: 0.000, Accuracy: 100.00%
(Iter 06000) Bit error: 0.000, Accuracy: 100.00%
(Iter 07000) Bit error: 0.000, Accuracy: 100.00%
(Iter 08000) Bit error: 0.000, Accuracy: 100.00%
(Iter 09000) Bit error: 0.000, Accuracy: 100.00%
(Iter 10000) Bit error: 0.000, Accuracy: 100.00%
```

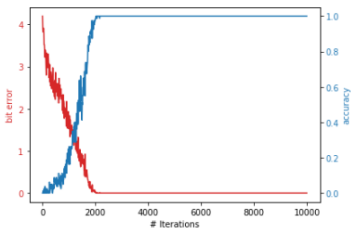

Figure 7: Traing Trend

To test the correctness of model, I generate 2 random integers between $[0, 127]$, and feed them to the network. The result shown in Figure 8 shows that the model successfully learn the task of binary addition.

```python
import random
x1 = random.randint(0, 127)
x2 = random.randint(0, 127)
y_pred = net.predict_number(x1, x2)
print("input : (%d, %d)"%(x1, x2))
print("prediction : %d "%(y_pred))
```

```
input : (41, 96)
prediction : 137
```

Figure 8: Result of Adding Two Random Integers

5 Appendix

Belows are the mathematical derivation for the **Back-Propgation Through Time** (BPTT), note that the symbol $H^{(t)}$ is replaced by $D^{(t)}$ in the following figures.

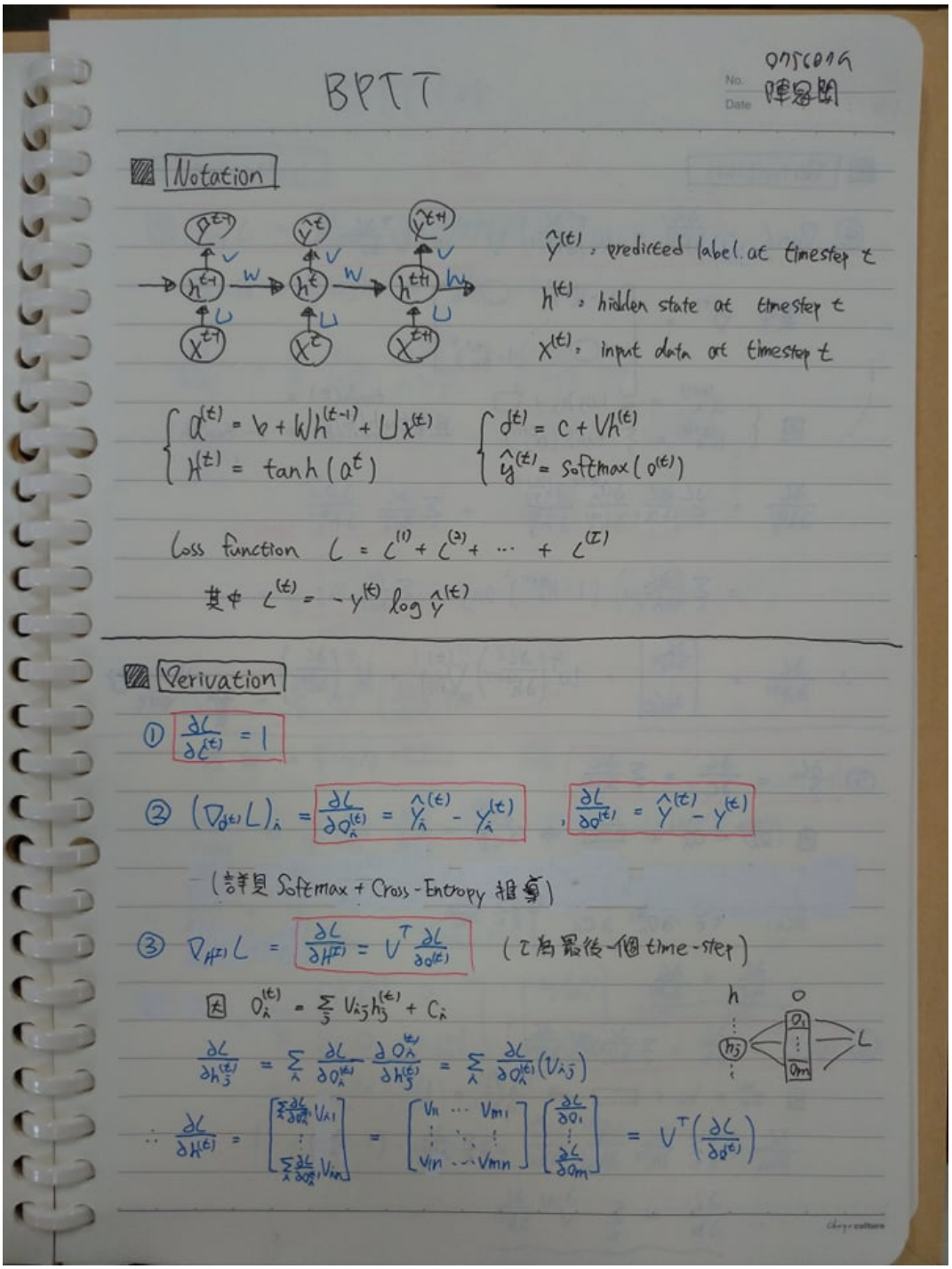

Figure 9: BPTT Derivation (part1)

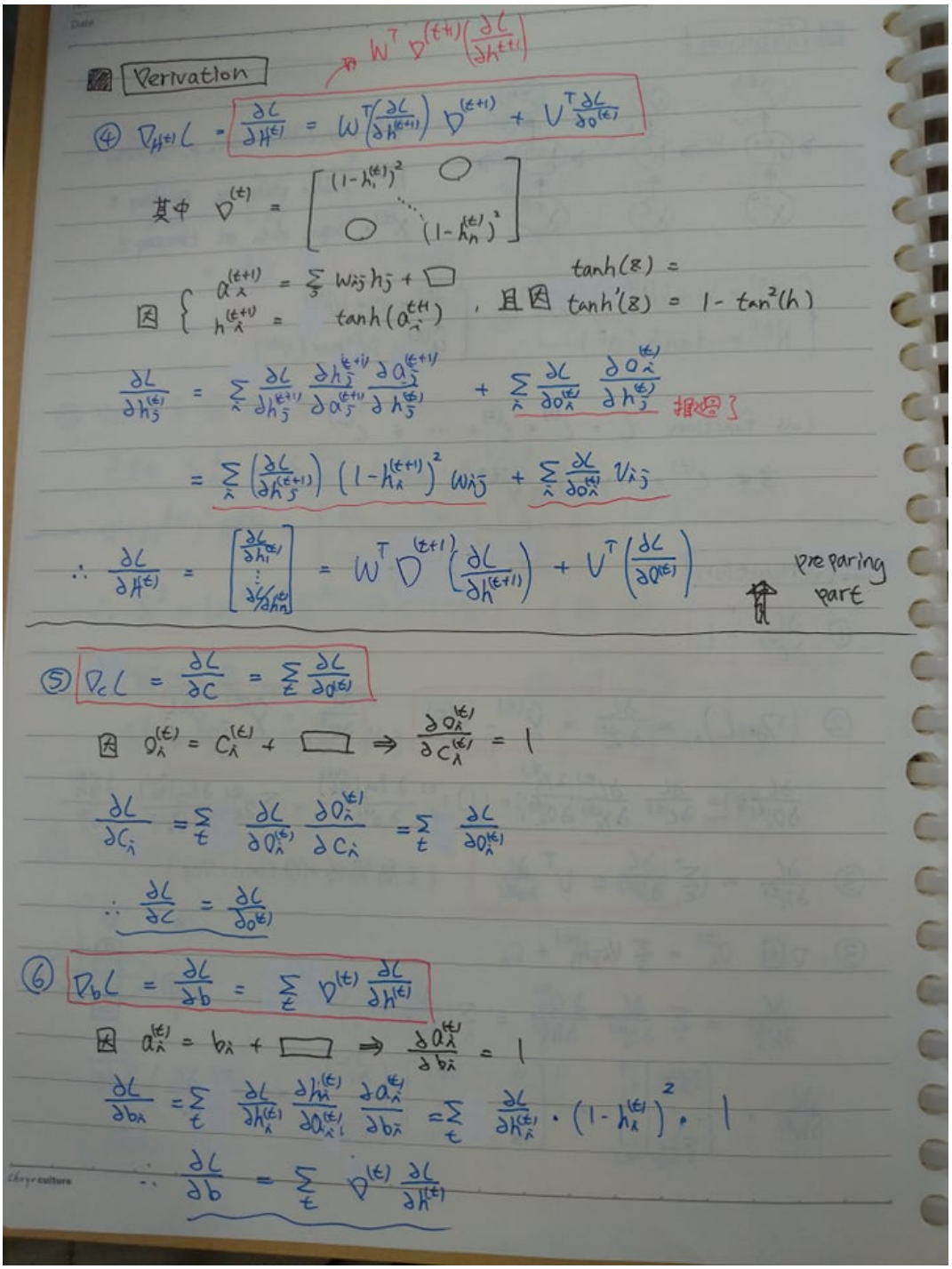

Figure 10: BPTT Derivation (part2)

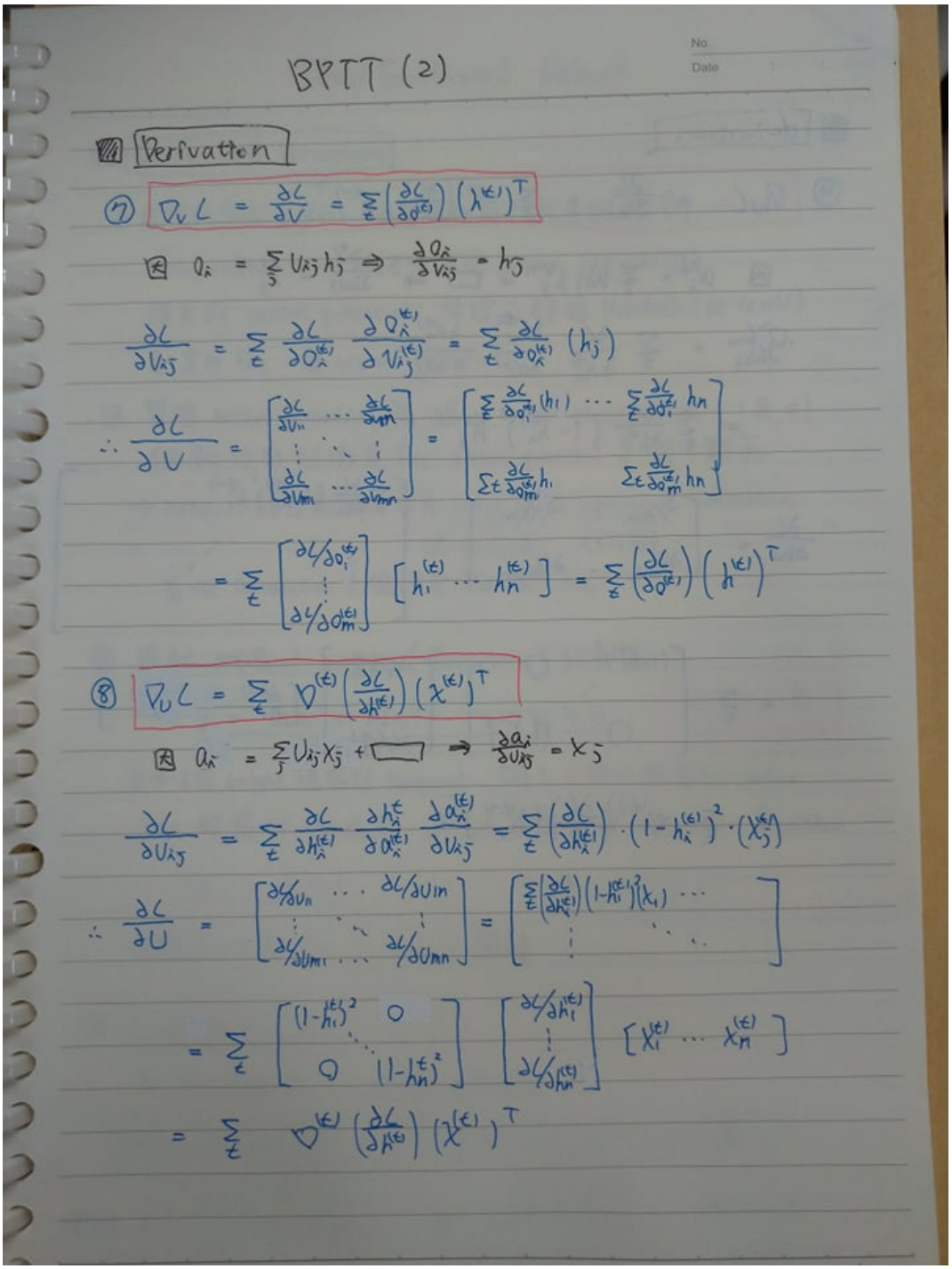

Figure 11: BPTT Derivation (part3)

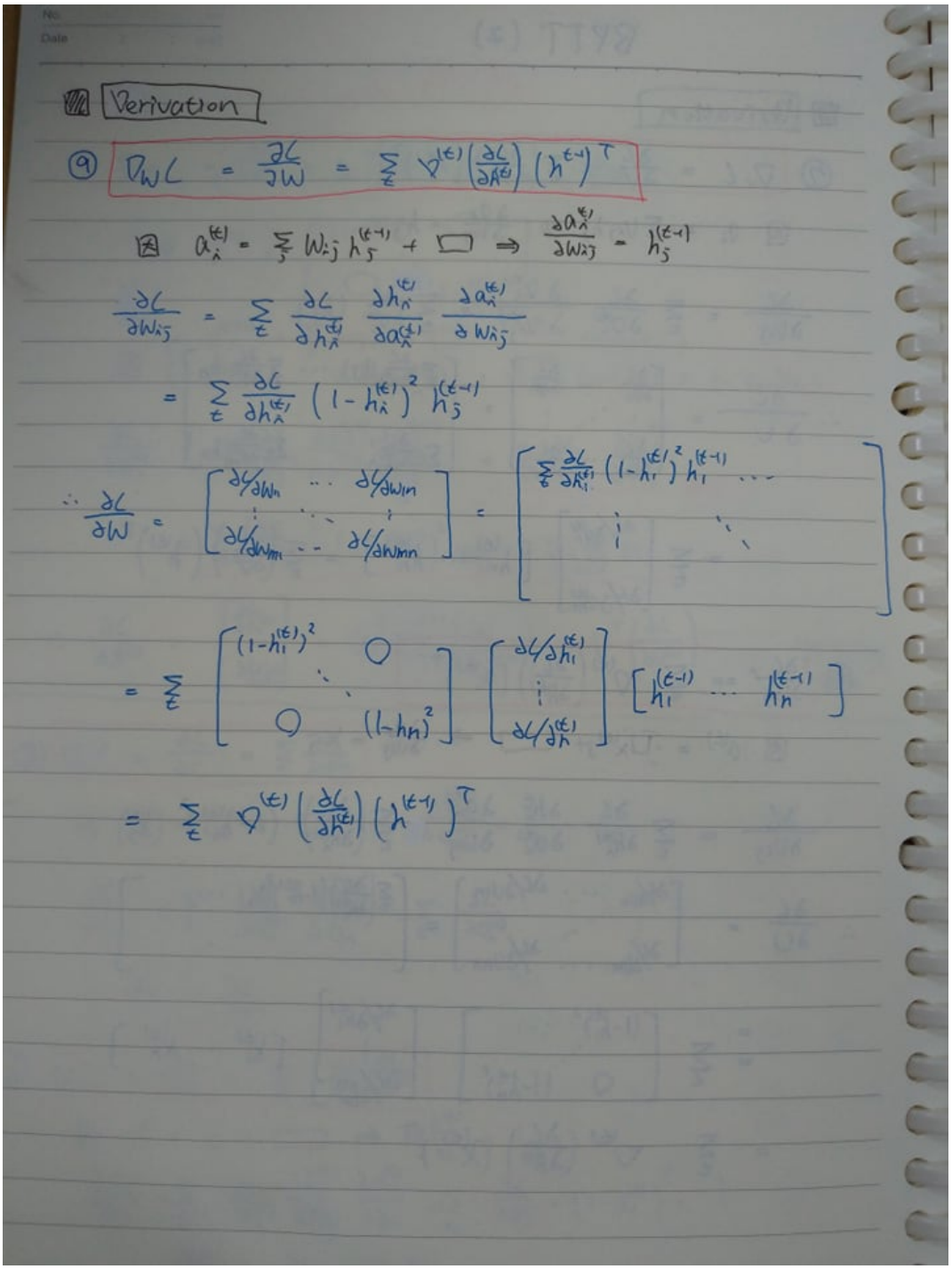

Figure 12: BPTT Derivation (part4)