2019 Deep Learning and Practice
Lab 4 – Back-Propagation Through Time (BPTT)

0756110 李東霖

April 22, 2019

1 Introduction

In this project, I implemented a RNN model as binary addition. To calculate gradient of RNN model, I also implemented BPTT (Bcak-Propagation Through Time).

Some requirements as follows:

- Construct RNN and forward propagation
- Back-Propagation Through Time
- Only use `Numpy` and pure python librarys.
- Generate binary addition data
- Train/Test RNN model

2 Plot shows episodes

Error is how many digits is not the same as answer. e.g. `0b1010` + `0b0010` = `0b1100` and my model output is `0b1000`, the error is 1.

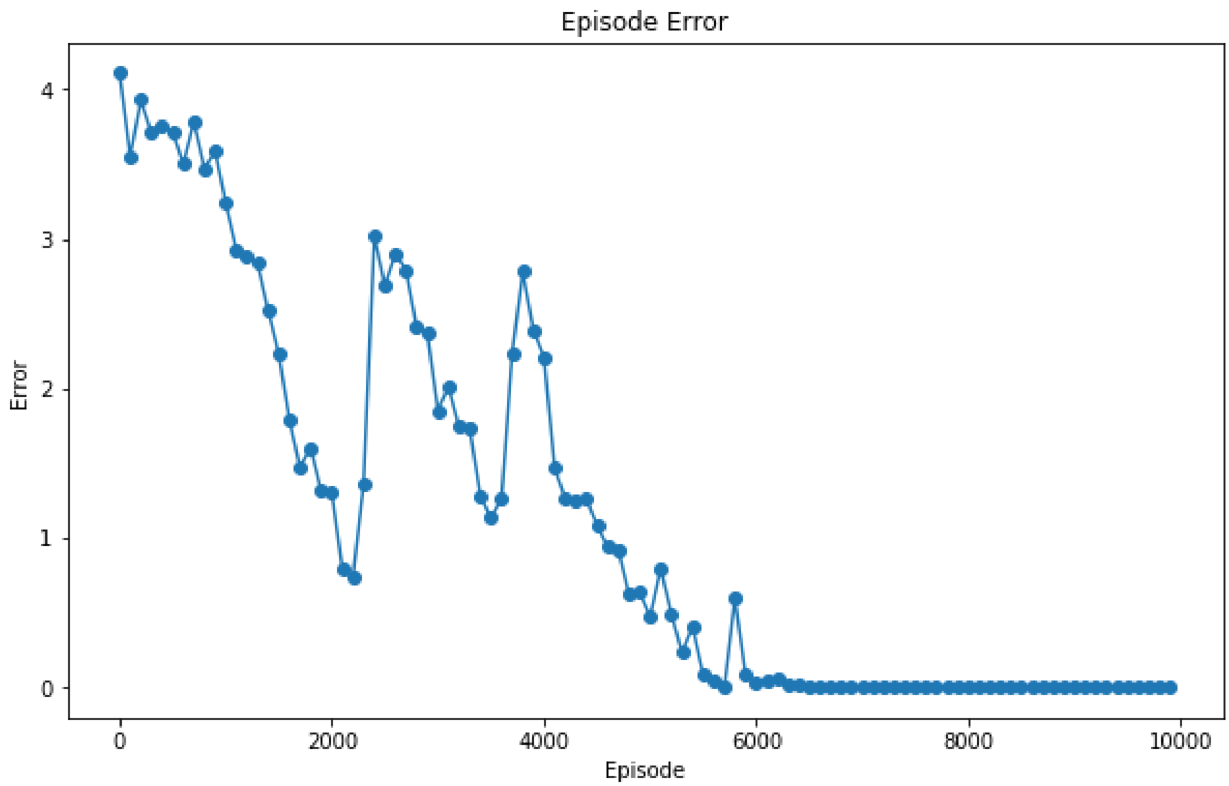

Figure 1: Show average error per 100 episodes

3 Generate data

I use python generator to generate binary data `x, y`. To quickly convert number to binary list, I write function `toBinary` to reach. The `toBinary` receive number, how many digits you want convert and option for using 2's complement or not. Finally, I convert data to `Variable` implemented by me and divide data into different bits.

When training model, I can get `x, y` from calling `next(dataset)`.

```
def toBinaray(x, digits, complement=False):
    if complement and x < 0:
        x += (1<<(digits))
    x = abs(x)
    return [ float(int(i)) for i in list(("{:0" + str(digits) +
     ↪  "b}").format(x))[::-1]][:digits]

def BinaryDataset(digits=8):
    thr = (1<<(digits-1))
    while True:
```

```
        a, b = np.random.randint(0, thr, 2)
        c = a + b
        x = np.array([toBinaray(a, digits), toBinaray(b, digits)])
        y = np.array([toBinaray(c, digits)])
        yield [Variable(x.T[i:i+1, :]) for i in range(digits)],
         ↪  [Variable(y.T[i:i+1, :]) for i in range(digits)]

dataset = BinaryDataset(8)
x, y = next(dataset)
```

4 Forward in RNN

First, define our architecture, pattern and symbol for derivation.

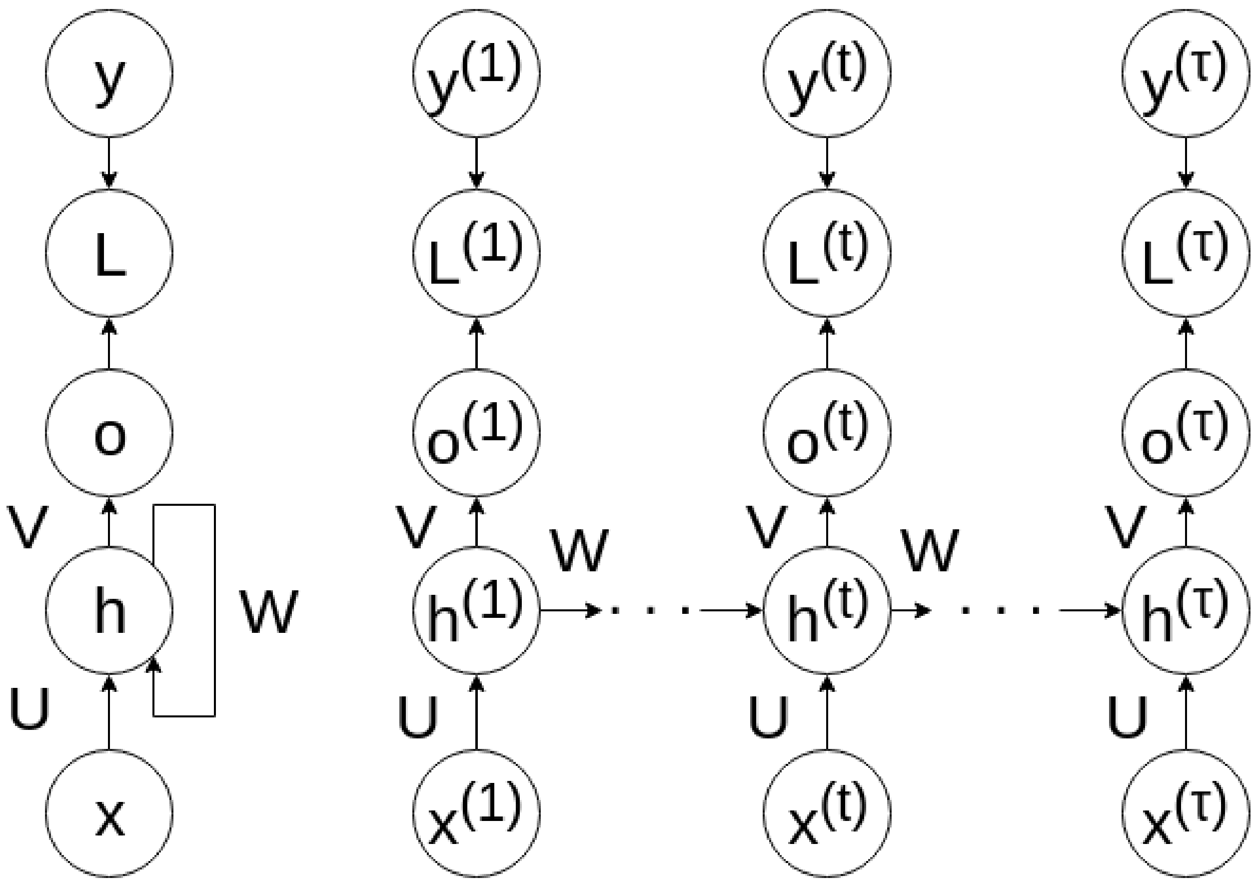

Figure 2: Compute Graph for my RNN

Assume have a serial data. In this lab, the serial data x is different bit of x1 and x2 and y is different bit of y.

$$x^{(1)}, x^{(2)}, ..., x^{(\tau)} \to y^{(1)}, y^{(2)}, ..., y^{(\tau)}$$

e.g. 0b0010 + 0b1010 = 0b1001

$$\begin{bmatrix}0 & 1\end{bmatrix}, \begin{bmatrix}0 & 0\end{bmatrix}, \begin{bmatrix}1 & 1\end{bmatrix}, \begin{bmatrix}0 & 0\end{bmatrix} \to \begin{bmatrix}1\end{bmatrix}, \begin{bmatrix}0\end{bmatrix}, \begin{bmatrix}0\end{bmatrix}, \begin{bmatrix}1\end{bmatrix}$$

General forward propagation :

$$h^{(t)} = \sigma(a^{(t)}), a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \tag{1}$$

$$o^{(t)} = c + Vh^{(t)} \tag{2}$$

$$\hat{y}^{(t)} = \varphi(o^{(t)}) \tag{3}$$

$$L^{(t)} = \text{loss}(\hat{y}^{(t)}, y^{(t)}) \tag{4}$$

RNN use previous hidden output to foward new hidden output. This is why RNN call Recurrent Neural Network.

In order to implement a RNN model, I define activation and loss function :

$$\sigma(x) = tanh(x) \tag{5}$$

$$\varphi(x) = \text{softmax}(x) \tag{6}$$

$$\text{loss}(\hat{y}, y) = \text{NLLLoss}(\hat{y}, y) = -log \prod_i (\hat{y}_i)^{y_i} \tag{7}$$

But in my implement, I combine softmax and nllloss as cross entropy loss. Because the cross entropy loss more easily calculate gradient.

My parameters in RNN model as follows:

- U : shape (2 x hidden-unit), use for input x^t to hidden h^t
- W : shape (hidden-unit x hidden-unit), use for previous hidden h^{t-1} to current hidden h^t
- V : shape (hidden-unit x 2), use for hidden h^t to output o^t
- b : shape (1 x hidden-unit), bias for hidden layer
- c : shape (1 x 2), bias for output layer

5 BPTT in RNN

To training this model, I need to calculate gradient of all parameters in model.

$$\nabla_W{}^L, \nabla_U{}^L, \nabla_V{}^L, \nabla_b{}^L, \nabla_c{}^L$$

First, calculate $\nabla_W{}^L$

$$\nabla_W{}^L = \sum_t \sum_i (\frac{\partial L}{\partial h_i^{(t)}}) \nabla_W{}^{h_i^{(t)}}, \nabla_W{}^{h_i^{(t)}} = \frac{\partial h_i^{(t)}}{\partial a_i^{(t)}} \frac{\partial a_i^{(t)}}{\partial W}$$

$$\frac{\partial h_i^{(t)}}{\partial a_i^{(t)}} = tanh'(a_i^{(t)}) = 1 - tanh^2(a_i^{(t)}) = 1 - (h_i^{(t)})^2, \frac{\partial a_i^{(t)}}{\partial W} = h_i^{(t-1)}$$

$$\nabla_W{}^{h_i^{(t)}} = (1 - (h_i^{(t)})^2)(h_i^{(t-1)})$$

Second, calculate $\frac{\partial L}{\partial h_i^{(t)}}$.

$$\frac{\partial L}{\partial h_i^{(t)}} = (\frac{\partial h_i^{(t+1)}}{\partial h_i^{(t)}} \frac{\partial L}{\partial h_i^{(t+1)}}) + (\frac{\partial o_i^{(t)}}{\partial h_i^{(t)}} \frac{\partial L}{\partial o_i^{(t)}})$$

$$\frac{\partial h_i^{(t+1)}}{\partial h_i^{(t)}} = \frac{\partial a_i^{(t+1)}}{\partial h_i^{(t)}} \frac{\partial h_i^{(t+1)}}{\partial a_i^{(t+1)}} = W(1 - (h_i^{(t+1)})^2)$$

$$\frac{\partial o_i^{(t)}}{\partial h_i^{(t)}} = V$$

Third, calculate $\frac{\partial L}{\partial o_i^{(t)}}$

$$\frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial \hat{y}^{(t)}}{\partial o_i^{(t)}} \frac{\partial L}{\partial \hat{y}^{(t)}} = \text{softmax}'(o^{(t)})(\frac{\partial - \sum_j y_j^{(t)} log(\hat{y}_j^{(t)})}{\partial \hat{y}^{(t)}})$$

$$textsoftmax'(x) = \begin{cases} \text{softmax}(x_i)(1 - \text{softmax}(x_i)), \text{if i==j} \\ -\text{softmax}(x_i)\text{softmax}(x_j), \text{if i != j} \end{cases}$$

$$= \hat{y}_i^{(t)}(1 - \hat{y}_i^{(t)})(-\frac{y_i^{(t)}}{\hat{y}_i^{(t)}}) + \sum_{j \neq i} -\hat{y}_i^{(t)} \hat{y}_j^{(t)} (-\frac{y_j^{(t)}}{\hat{y}_j^{(t)}})$$

$$= (\hat{y}_i^{(t)} - 1) y_i^{(t)} + \sum_{j \neq i} \hat{y}_i^{(t)} y_j^{(t)} = (\sum_j y_j^{(t)}) \hat{y_i}^{(t)} - y_i^{(t)}$$

if $\sum y(t)$ is one, equation can rewrite to:

$$\frac{\partial L}{\partial o_i^{(t)}} = \hat{y_i^{(t)}} - y_i^{(t)} \tag{8}$$

And need final time layer hidden gradient to start BPTT.

$$\nabla_{h_i^{(\tau)}} L = \frac{\partial o_i^{(\tau)}}{\partial h_i^{(\tau)}} \frac{\partial L}{\partial o_i^{(\tau)}} = V(\hat{y_i^{(\tau)}} - y_i^{(\tau)}) \tag{9}$$

Finally, get all equations for computing gradient :

$$\frac{\partial L}{\partial h_i^{(t)}} = W(1-(h_i^{t+1})^2)\frac{\partial L}{\partial h_i^{(t+1)}} + V(\hat{y_i^{(t)}} - y_i^{(t)}) \tag{10}$$

$$\nabla_W L = \sum_t \sum_i (\frac{\partial L}{\partial h_i^{(t)}})(\frac{\partial h_i^{(t)}}{\partial W}) = \sum_t \sum_i \frac{\partial L}{\partial h_i^{(t)}}(1-(h_i^{(t)})^2)(h_i^{(t-1)}) \tag{11}$$

$$\nabla_U L = \sum_t \sum_i (\frac{\partial L}{\partial h_i^{(t)}})(\frac{\partial h_i^{(t)}}{\partial U}) = \sum_t \sum_i \frac{\partial L}{\partial h_i^{(t)}}(1-(h_i^{(t)})^2)(x_i^{(t)}) \tag{12}$$

$$\nabla_V L = \sum_t \sum_i (\frac{\partial L}{\partial o_i^{(t)}})(\frac{\partial o_i^{(t)}}{\partial V}) = \sum_t \sum_i (\hat{y_i^{(t)}} - y_i^{(t)}) h_i^{(t)} \tag{13}$$

$$\nabla_b L = \sum_t \sum_i (\frac{\partial L}{\partial h_i^{(t)}})(\frac{\partial h_i^{(t)}}{\partial b}) = \sum_t \sum_i \frac{\partial L}{\partial h_i^{(t)}}(1-(h_i^{(t)})^2) \tag{14}$$

$$\nabla_c L = \sum_t \sum_i (\frac{\partial L}{\partial o_i^{(t)}})(\frac{\partial o_i^{(t)}}{\partial c}) = \sum_t \sum_i \hat{y_i^{(t)}} - y_i^{(t)} \tag{15}$$

6 Code explanation

I can directly declare U, W, V, b, c parameters and write equation to calculate gradient. But I also want to understand pytorch `autograd`. Because of that, I implement a data node class called `Variable` to store compute graph and calculate gradient through graph.

6.1 Calculating Compute Graph Gradient

To store compute graph, I store what variable and operation that create this variable in `fn`. And I want to trace how many variable be created by this variable, I store them into `child`. The `ready` bool means this variable already finish computing gradient from target to it self. `data` stores variable content as `ndarray`. `grad` stores variable gradient as `ndarray`.

```
class Variable():
    def __init__(self, data, T=None, grad=None, copy=True):
        if data is None or type(data) != np.ndarray:
            raise AttributeError('Wrong data type')

        if copy:
            self.data = data.copy()
        else:
```

```
            self.data = data
        if grad is None:
            grad = np.zeros_like(self.data)
        self.grad = grad
        if T is None:
            T = Variable(self.data.T, self, self.grad.T, copy=False)
        self.T = T
        self.fn = None
        self.child = []
        self.ready = False

    def zero_grad(self):
        self.grad[:,:] = 0.0
        self.child = []
        self.ready = False
```

To easily use this class, I overloaded the operator about it. When this class do operation with the same other, That overload method will record some info about compute graph. The operator overloaded as follows:

- add (matrix element-wise addition)
- sub (matrix element-wise subtraction)
- mul (matrix element-wise multiple)
- matmul (matrix multiple)

I just put one of them as example to explain my code. Other overload methods are similar but calculating gradient are different.

```
class Variable():
    # Omit sth ...
    def __matmul__(self, b):
        c = Variable(np.matmul(self.data, b.data))
        c.fn = [Variable.__grad_matmul__, self, b]

        self.child.append(c)
        b.child.append(c)
        return c

    def __grad_matmul__(self, a, b):
        a.grad += np.matmul(self.grad, b.data.T)
        b.grad += np.matmul(a.data.T, self.grad)
```

Assume `A`, `B` is Variable class. When do matrix multiple (`C = A @ B`), python will send `A` as `self` and `B` as `b` into `__matmul__` method. So I can store any compute graph.

I also need some special operation in RNN model.

- tanh
- crossentropy

```
class Variable():
    # Omit sth ...
    def crossentropy(self, target):
        s = self.softmax(1)
        if type(target) is Variable:
            target = target.data

        target = target.astype(np.int)

        if target.shape[0] > 1:
            slis = tuple(zip(range(target.shape[0]), target))
        else:
            slis = (0, target[0])

        c = Variable(np.array(np.sum(-np.log(s[slis]))))
        c.fn = [Variable.__grad_corssentropy, self, target]

        self.child.append(c)
        return c

    def __grad_corssentropy(self, a, target):
        y = np.zeros_like(a.grad)
        if target.shape[0] > 1:
            slis = tuple(zip(range(target.shape[0]), target))
        else:
            slis = (0, target[0])

        y[slis] = 1.0
        a.grad += (a.softmax(1) - y)
```

I combine softmax and nllloss as one operation. It help me easily compute gradient. (like 8)

If finish forward propagation, whole compute graph have been store in multi Varaible class. I can call `backward` method on loss Variable then it will do BPTT through compute graph.

```python
class Variable():
    # Omit sth ...
    def backward(self, backward_grad):
        if type(backward_grad) is Variable:
            backward_grad = backward_grad.data

        if backward_grad.shape != self.data.shape:
            raise AttributeError('Wrong backward grad shape {} !=
             ↪  {}'.format(backward_grad.shape, self.data.shape))

        self.grad = backward_grad
        self.__backward()

    def __backward(self):
        if self.fn is None:
            return;

        # check self grad is ready, trace child variables
        self.ready = True
        for child in self.child:
            self.ready &= child.ready

        if not self.ready:
            return;

        backward_op = self.fn[0]

        backward_op(self, *self.fn[1:])

        for v in self.fn[1:]:
            if type(v) is Variable:
                v.__backward()
```

To start backward, I need backward gradient from target (e.g. Loss result L) to this Variable. Then check `fn` find compute graph. Second, check `child` ready to sure self gradient has been calculated. Third, call backward operation paired with forward operation (e.g. `__add__` mapping `__grad_add__`) to calculate gradient of previous creator. If recursive loop finish, that means already calculated all gradient of compute graph.

6.2 Build RNN

Now, I can easily build model by RNN forward equation.

```python
class RNN():
    def __init__(self, in_channels, out_channels, hidden_channels):
        self.U = Variable(np.random.uniform(-1,1, (in_channels, hidden_channels)))
        self.W = Variable(np.random.uniform(-1,1, (hidden_channels, hidden_channels)))
        self.V = Variable(np.random.uniform(-1,1, (hidden_channels, out_channels)))
        self.b = Variable(np.random.uniform(-1,1, (1, hidden_channels)))
        self.c = Variable(np.random.uniform(-1,1, (1, out_channels)))

    def forward(self, x):
        t = len(x)
        self.h = None
        y = []

        for i in range(t):
            a = self.b + (x[i] @ self.U)
            if self.h is not None:
                a += (self.h @ self.W)

            self.h = a.tanh()

            o = self.c + (self.h @ self.V)
            y.append(o)

        return y

    def zero_grad(self):
        self.U.zero_grad()
        self.W.zero_grad()
        self.V.zero_grad()
        self.b.zero_grad()
        self.c.zero_grad()

    def step(self, lr=1e-1):
        self.U.data -= lr * self.U.grad
        self.W.data -= lr * self.W.grad
        self.V.data -= lr * self.V.grad
        self.b.data -= lr * self.b.grad
        self.c.data -= lr * self.c.grad
```

I assume shape of input `x` is (T, D). T means how many bit in this serial data. (In this lab, it is 8) D means how many number in this bit. (In this lab, it is 2) Then I use forward equation to generate output. `zero_grad` is for cleaning gradient buffer. `step` is for updating parameters.

6.3 Calculate Loss and Update

```
x, y = next(dataset)

model.zero_grad()

output = model.forward(x)

loss = [output[i].crossentropy(y[i]) for i in range(len(y))]

for l in loss[::-1]:
l.backward(np.array(1))

model.step(1e-2)
```

7 Discussion

7.1 Extend number of binary bit when evalution

I found the RNN always have 5 important parameters U, W, V, b, c. Base on that, I think this model can store the carry information into hidden output. And the hidden output can influence the model output. Therefore the model should be trained by 8 bits but it can handle more than 8 bits. I try to prove this thing.