DL HW4: Back Propagation Through Time (BPTT)

Ting Hsuan, Wang

April 23, 2019

1 BPTT

1.1 A plot shows accuracies of last 1000 iterations (10%)

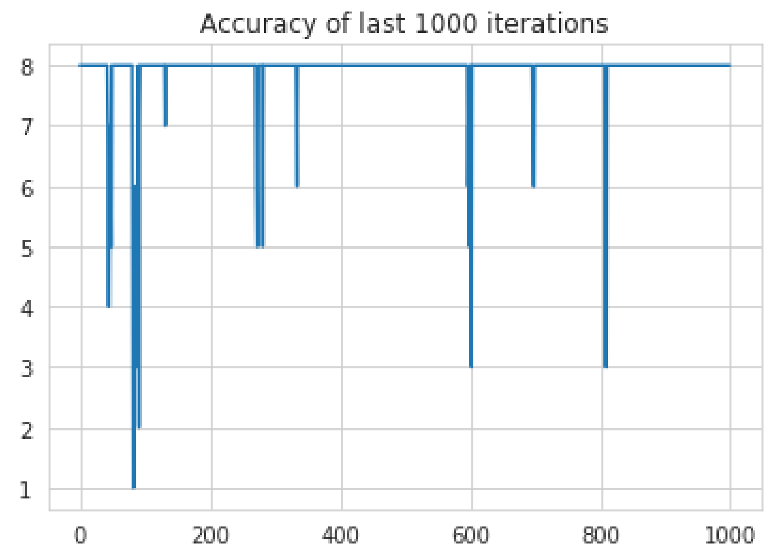


Figure 1: Accuracies of last 1000 iterations.

We iterate the RNN for 20000 times. For each iteration, we generate a new data and feed into our model. From Figure 1, we can see that the system almost converges in the last 1000 iterations out of 20000 iterations. The average correct digit of last 1000 iterations is around 7.8 to 7.9.

1.2 Describe how to generate data? (10%)

Data are generated by numpy random integer which smaller than 255/2. We first generate x1 and x2, then compute output as x1 + x2. Then, we convert x1, x2 and output into binary string and turn them into integer again for RNN's inputs and target outputs. The code is shown in Figuer 2.

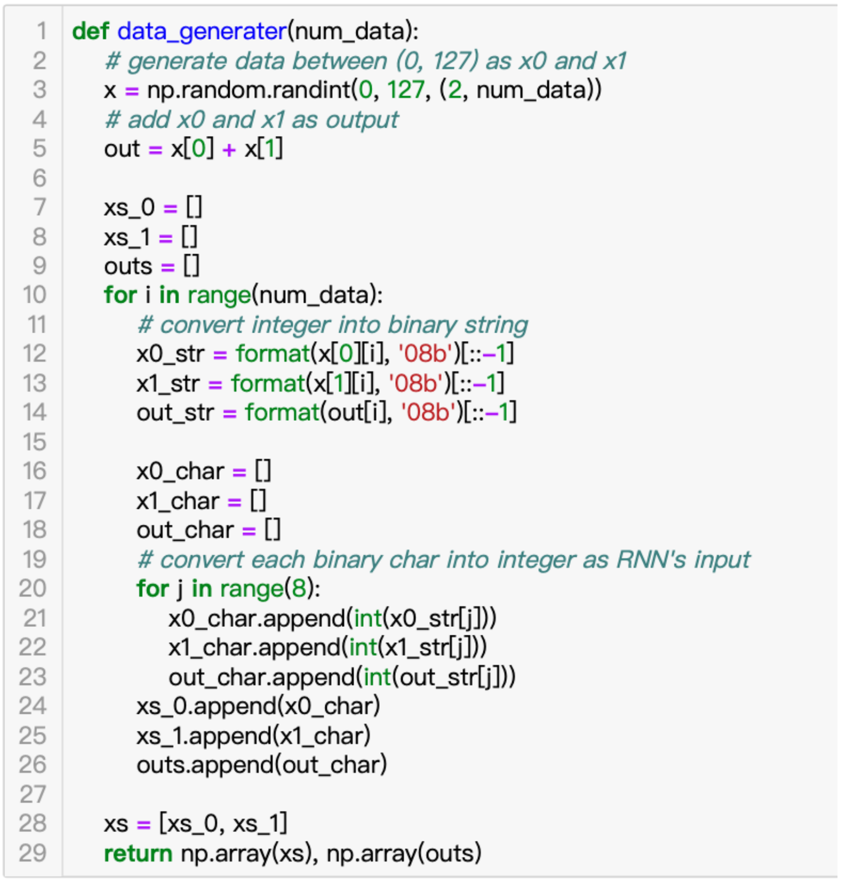

```python
def data_generater(num_data):
    # generate data between (0, 127) as x0 and x1
    x = np.random.randint(0, 127, (2, num_data))
    # add x0 and x1 as output
    out = x[0] + x[1]

    xs_0 = []
    xs_1 = []
    outs = []
    for i in range(num_data):
        # convert integer into binary string
        x0_str = format(x[0][i], '08b')[::-1]
        x1_str = format(x[1][i], '08b')[::-1]
        out_str = format(out[i], '08b')[::-1]

        x0_char = []
        x1_char = []
        out_char = []
        # convert each binary char into integer as RNN's input
        for j in range(8):
            x0_char.append(int(x0_str[j]))
            x1_char.append(int(x1_str[j]))
            out_char.append(int(out_str[j]))
        xs_0.append(x0_char)
        xs_1.append(x1_char)
        outs.append(out_char)

    xs = [xs_0, xs_1]
    return np.array(xs), np.array(outs)
```

Figure 2: Code on how to generate data.

1.3 Explain the mechanism of forward propagation (20%)

Figure 3 shows the RNN architecture. In each time step, a 2-dimensional vector (ith bit for number x1 and x2) $x^{(t)}$ is fed into the network, and a 1-demensional vector $\hat{y}^{(t)}$ is generated. As for the matrix term, W, U and V, are used to project input into different dimensions. To be more specific, we write the forward pass in mathematical formula:

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

where U, W, V correspond respectively to connections for: (1) Input-to-hidden units(U), (2) Hidden-to-hidden units(W), (3) Hidden-to-output units(V), and b, c are bias. Note that a sigmoid function is added at the last equation. This is used to limit the range of the output to 0 1, since our outputs are expected to be between 0 and 1. As for loss function, we choose binary cross entropy to measure the loss as follows:

$$L = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y})) \tag{2}$$

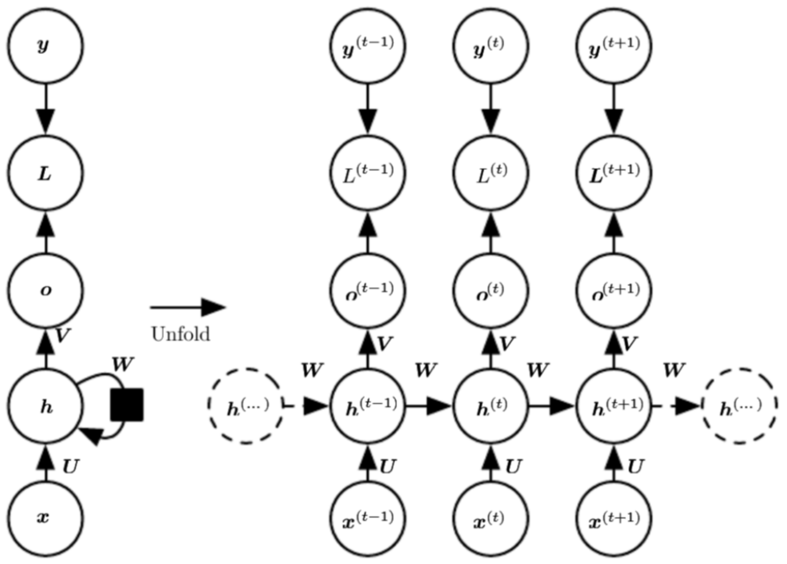


Figure 3: Naive architecture of RNN.

where y is true label and $\hat{y}$ is predicted label.

1.4 Explain the mechanism of BPTT (20%)

In backpropagation, we calculate gradients to update parameters. However, the backpropagation here is slightly different from the traditional one. We have to add up gradients in former time step since those gradients also have impact on current time step's parameters. We simply start from the loss term:

$$\frac{\partial L}{\partial \hat{y}} = \frac{y - \hat{y}}{m(\hat{y} - \hat{y} * \hat{y})} \tag{3}$$

where m is batch size.
Then propagate back to all the other parameters:

$$\frac{\partial L}{\partial o^{(t)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o^{(t)}} = \frac{\partial L}{\partial \hat{y}} (\sigma(o^{(t)}))(1 - \sigma(o^{(t)})) \tag{4}$$

$$\frac{\partial L}{\partial c} = \sum_t \frac{\partial L}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial c} = \sum_t \frac{\partial L}{\partial o^{(t)}} (1) \tag{5}$$

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial V} = \sum_t \frac{\partial L}{\partial o^{(t)}} h^{(t)T} \tag{6}$$

$$\frac{\partial L}{\partial h^{(t)}} = \frac{\partial L}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial h^{(t)}} + \frac{\partial L}{\partial a^{(t)}} \frac{\partial a^{(t)}}{\partial h^{(t-1)}} = V^T \frac{\partial L}{\partial o^{(t)}} + W^T H^{(t+1)} \frac{\partial L}{\partial h^{(t+1)}} \tag{7}$$

$$\frac{\partial L}{\partial a^{(t)}} = \frac{\partial L}{\partial h^{(t)}}\frac{\partial h^{(t)}}{\partial a^{(t)}} = \frac{\partial L}{\partial h^{(t)}}(1 - tanh(a^{(t)})^2) = H^{(t)}\frac{\partial L}{\partial h^{(t)}} \tag{8}$$

$$\frac{\partial L}{\partial b} = \sum_t \frac{\partial L}{\partial a^{(t)}}\frac{\partial a^{(t)}}{\partial b} = \sum_t H^{(t)}\frac{\partial L}{\partial h^{(t)}}(1) \tag{9}$$

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L}{\partial a^{(t)}}\frac{\partial a^{(t)}}{\partial W} = \sum_t H^{(t)}\frac{\partial L}{\partial h^{(t)}}h^{(t-1)T} \tag{10}$$

$$\frac{\partial L}{\partial U} = \sum_t \frac{\partial L}{\partial a^{(t)}}\frac{\partial a^{(t)}}{\partial U} = \sum_t H^{(t)}\frac{\partial L}{\partial h^{(t)}}x^{(t-1)T} \tag{11}$$

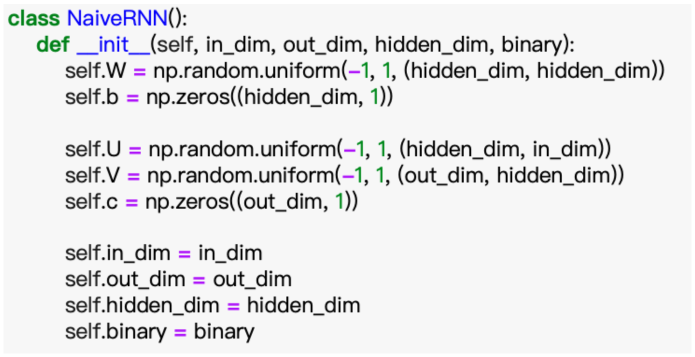
At the end, we get close form to update all the parameters.
It is worth noting that this network architecture corresponds to a conditional distribution:

$$\begin{aligned}
&p_m(y^{(1)}, y^{(2)}, ..., y^{(\tau)}|x^{(1)}, x^{(2)}, ..., x^{(\tau)}) \\
&= p_m(y^{(1)}|x^{(1)}, ..., x^{(\tau)})p_m(y^{(2)}|y^{(1)}, x^{(1)}, ..., x^{(\tau)})...p_m(y^{(\tau)}|y^{(1)}, y^{(2)}, ...y^{(\tau-1)}, x^{(1)}, ..., x^{(\tau)}) \\
&= p_m(y^{(1)}|x^{(1)}, ..., x^{(\tau)})p_m(y^{(2)}|x^{(1)}, ..., x^{(\tau)})...p_m(y^{(\tau)}|x^{(1)}, ..., x^{(\tau)})
\end{aligned} \tag{12}$$

where the second row is a simple chain rule, and the third row holds because the output vectors of our RNN architecture only consider input vectors.
Though our naive RNN solves binary addition cases, it may not work well on other tasks such as sentence generation based on current words. Sentence generation has to use former output to generate next output. In this situation, if we still use our naive RNN models, we may get poor results.

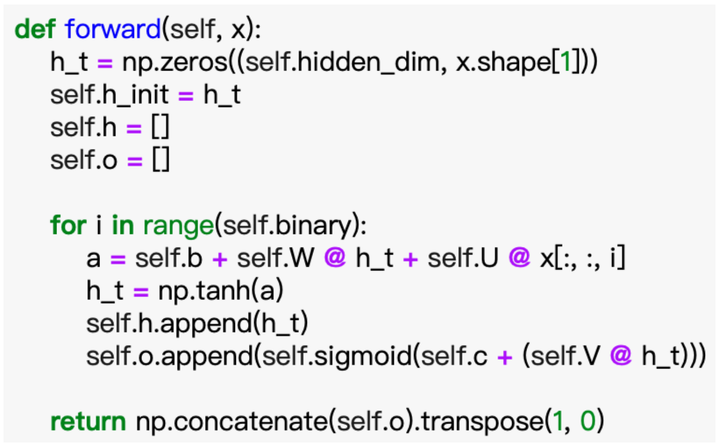
1.5 Describe how the code work (the whole code) (10%)

In Figure 4, we first setting parameters for input, hidden and output layers to initialize our parameters, W, U, V, b and c. Second, we do forward propagation to calculate our networks



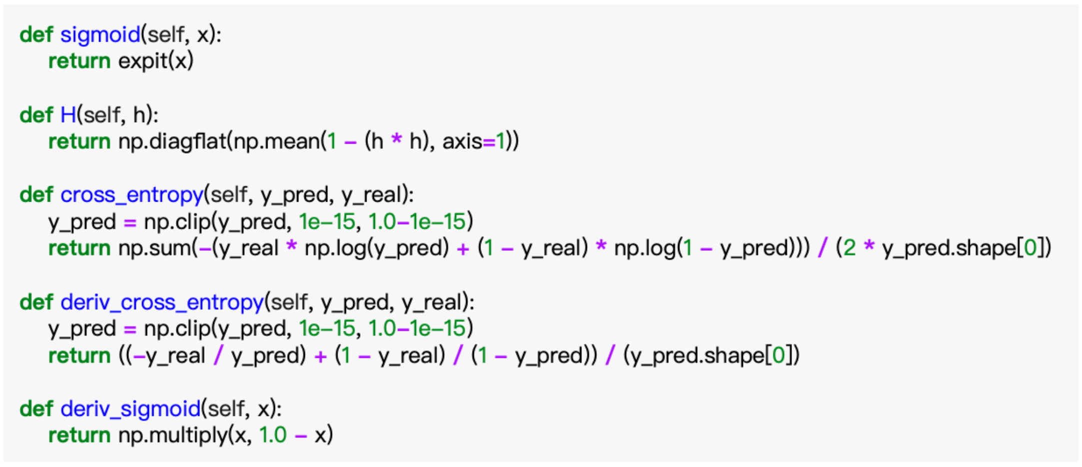
```
class NaiveRNN():
    def __init__(self, in_dim, out_dim, hidden_dim, binary):
        self.W = np.random.uniform(-1, 1, (hidden_dim, hidden_dim))
        self.b = np.zeros((hidden_dim, 1))

        self.U = np.random.uniform(-1, 1, (hidden_dim, in_dim))
        self.V = np.random.uniform(-1, 1, (out_dim, hidden_dim))
        self.c = np.zeros((out_dim, 1))

        self.in_dim = in_dim
        self.out_dim = out_dim
        self.hidden_dim = hidden_dim
        self.binary = binary
```

Figure 4: Initialization of our RNN.



```
def forward(self, x):
    h_t = np.zeros((self.hidden_dim, x.shape[1]))
    self.h_init = h_t
    self.h = []
    self.o = []

    for i in range(self.binary):
        a = self.b + self.W @ h_t + self.U @ x[:, :, i]
        h_t = np.tanh(a)
        self.h.append(h_t)
        self.o.append(self.sigmoid(self.c + (self.V @ h_t)))

    return np.concatenate(self.o).transpose(1, 0)
```

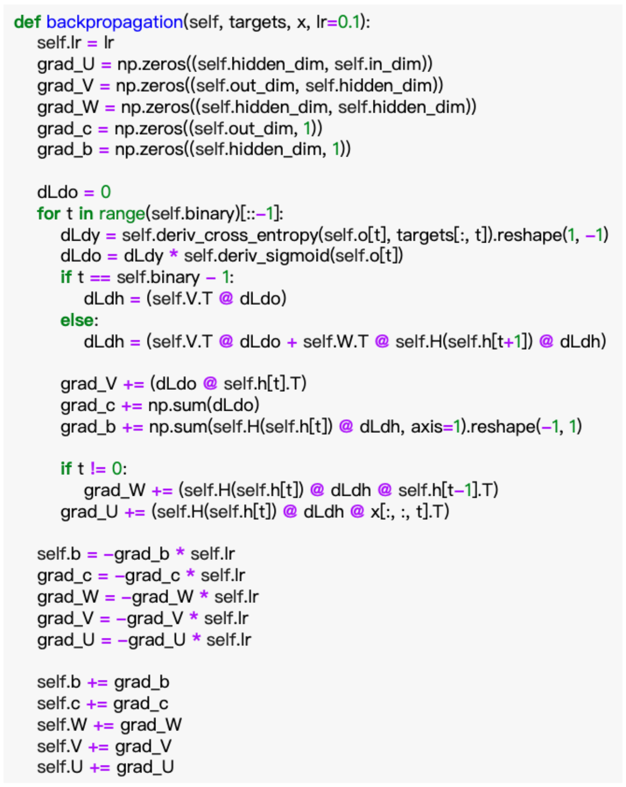
Figure 5: Forward pass of our RNN.



```
def sigmoid(self, x):
    return expit(x)

def H(self, h):
    return np.diagflat(np.mean(1 - (h * h), axis=1))

def cross_entropy(self, y_pred, y_real):
    y_pred = np.clip(y_pred, 1e-15, 1.0-1e-15)
    return np.sum(-(y_real * np.log(y_pred) + (1 - y_real) * np.log(1 - y_pred))) / (2 * y_pred.shape[0])

def deriv_cross_entropy(self, y_pred, y_real):
    y_pred = np.clip(y_pred, 1e-15, 1.0-1e-15)
    return ((-y_real / y_pred) + (1 - y_real) / (1 - y_pred)) / (y_pred.shape[0])

def deriv_sigmoid(self, x):
    return np.multiply(x, 1.0 - x)
```

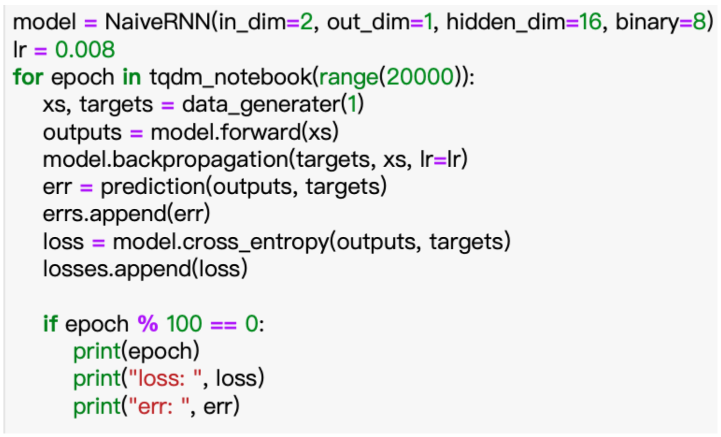
Figure 6: Functions that are implemented by ourselves.

output in Figure 5. Since our network emissions output in every time step, we will get t output for the total of t time steps. Next, we evaluate our performance using binary cross entropy. Then, we update the parameters by backpropagation in Figure 7 calculated using the formula above. Finally, we iterate through the above steps until the system converges in Figure 8.



```
def backpropagation(self, targets, x, lr=0.1):
    self.lr = lr
    grad_U = np.zeros((self.hidden_dim, self.in_dim))
    grad_V = np.zeros((self.out_dim, self.hidden_dim))
    grad_W = np.zeros((self.hidden_dim, self.hidden_dim))
    grad_c = np.zeros((self.out_dim, 1))
    grad_b = np.zeros((self.hidden_dim, 1))

    dLdo = 0
    for t in range(self.binary)[::-1]:
        dLdy = self.deriv_cross_entropy(self.o[t], targets[:, t]).reshape(1, -1)
        dLdo = dLdy * self.deriv_sigmoid(self.o[t])
        if t == self.binary - 1:
            dLdh = (self.V.T @ dLdo)
        else:
            dLdh = (self.V.T @ dLdo + self.W.T @ self.H(self.h[t+1]) @ dLdh)

        grad_V += (dLdo @ self.h[t].T)
        grad_c += np.sum(dLdo)
        grad_b += np.sum(self.H(self.h[t]) @ dLdh, axis=1).reshape(-1, 1)

        if t != 0:
            grad_W += (self.H(self.h[t]) @ dLdh @ self.h[t-1].T)
        grad_U += (self.H(self.h[t]) @ dLdh @ x[:, :, t].T)

    self.b = -grad_b * self.lr
    grad_c = -grad_c * self.lr
    grad_W = -grad_W * self.lr
    grad_V = -grad_V * self.lr
    grad_U = -grad_U * self.lr

    self.b += grad_b
    self.c += grad_c
    self.W += grad_W
    self.V += grad_V
    self.U += grad_U
```

Figure 7: Backpropagation of our model.



```
model = NaiveRNN(in_dim=2, out_dim=1, hidden_dim=16, binary=8)
lr = 0.008
for epoch in tqdm_notebook(range(20000)):
    xs, targets = data_generater(1)
    outputs = model.forward(xs)
    model.backpropagation(targets, xs, lr=lr)
    err = prediction(outputs, targets)
    errs.append(err)
    loss = model.cross_entropy(outputs, targets)
    losses.append(loss)

    if epoch % 100 == 0:
        print(epoch)
        print("loss: ", loss)
        print("err: ", err)
```

Figure 8: We iterate for 20000 epochs.