Deep Learning and Practice HW4

0751901 YengHung Ou

April 23, 2019

1 Introduction

In this project, we are asked to implement RNN from scratch. The optimization method for this project is BPTT(Back-Propagation Through Time), which is mentioned on page 9, chapter 10. The only library we can use for this project is Numpy or pure Python.

2 Training Accuracy and Error

The accuracy and error is plot in figure 1 and figure 2, the accuracy reaches 100% after 2000 iterations.

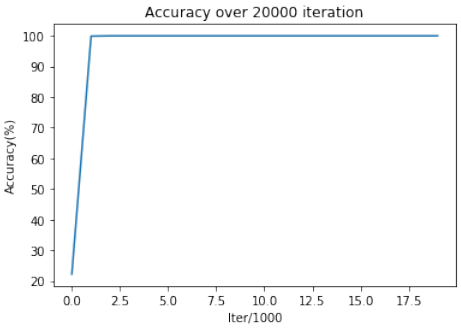


Figure 1: Accuracy over 20000 iterations

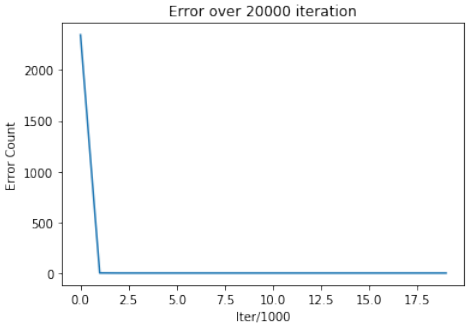


Figure 2: Error over 20000 iterations

3 Data Generation

In each iteration, two inputs are generated by randint() function to produce integer from 0 to 128, and then convert both the inputs into binaries. Since the inputs at one iteration could be the same as inputs at another iteration, to increase the speed of conversion from integer to binary, a lookup table, called int2binary, is created in the beginning to convert integer from 0 to 256 to binary for reusing. The true output, y, is first computed by adding two input integer, and then converted to binary by the lookup table.

4 Forward Propagation

Since the output in this project is 1-dimentional, which is not able to pass through a softmax() function, I modified the network in page 6, chapter 10 into

$$h_t = tanh(W \cdot h_{t-1} + U \cdot x_t) \tag{1}$$

$$o_t = V \cdot h_t \tag{2}$$

$$\hat{y}_t = \sigma(o_t) \tag{3}$$

$$L_t = 0.5(y_t - \hat{y}_t)^2 \tag{4}$$

$$L = \sum_{t=1}^{8} L_t \tag{5}$$

x_t is the input vector at timestamp t, which is a 2x1 vector.
h_t is the output of hiddin unit at timestamp t, which is a 16x1 vector.
$\hat{y}$ is the output of the network at timestamp t, which a 1x1 vector.
W, V, U is weight parameters for h_{t-1}, x_t, and h_t, the dimension is 16x16, 1x16, and 16x2 respectively.
Comparing to the network structure in chapter 10, the softmax() is substituted by sigmoid() to limit the output into 0 1 and the MSE(mean square error) is used as loss function since I viewed te ntwork as a regression model.
In the whole pipeline of forward propagation, one bit is extracted from two inputs from right to left and put them in x as the input of each forward propagation. For generalization, I append 0 vector to the list storing values of h for computing h_0 so as not to treat the computation of h_0 separately.

5 BPTT

To do the BPTT, we need to compute ∇_W^L, ∇_U^L, and ∇_V^L, and each can be computed as

$$\nabla_W^L = \sum_t H_t(\nabla_{h_t}^L)h_{t-1}^T \tag{6}$$

$$\nabla_U^L = \sum_t H_t(\nabla_{h_t}^L)x_t^T \tag{7}$$

$$\nabla_V^L = \sum_t (\nabla_{o_t}^L)h_t^T \tag{8}$$

where H_t is $\begin{bmatrix} 1-(h_1^t)^2 & & \\ & \ddots & \\ & & 1-(h_8^t)^2 \end{bmatrix}$, $\nabla_{o_t}^L$ and $\nabla_{h_t}^L$ is

$$(\hat{y}_t - y) * \sigma'(\hat{y}_t) \tag{9}$$

$$W^T H^{t+1} \nabla_{h_{t+1}}^L + V^T \nabla_{o_t}^L \tag{10}$$

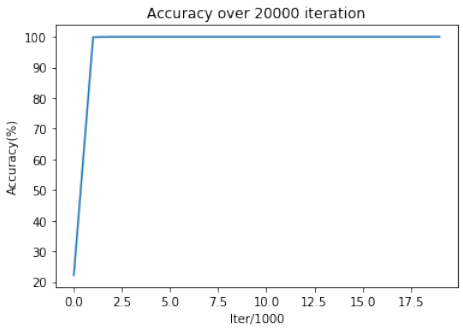
For generalization, H_9 and $\nabla_{h_9}^L$ is set as zero for doing BPTT when t=8.

6 Code Explanation

The whole code in HW4.py can be split into 5 parts: Line 2 to line 12 import the Numpy library and define some parameters like network dimension or iteration time. Line 13 to line 18 create the lookup table for converting integer to binary. Line 19 to line 22 define three weight parameters U, W, and V. Line 23 to line 32 define activation function tanh() and sigmoid and their derivatives. Line 33 to line 127 define the train function which includes the forward propagation and BPTT.
The pipeline of the whole training process is described in Algorithm 1.

Algorithm 1 Training Pipeline

for iter in range(iterations) **do**
    random two integer a, b from 0$\tilde{1}$28
    generate true output y=a+b
    convert a, b, y into binary
    #Forward Propagation
    for pos in range(binary_dim)[::-1] **do**
        x = [a[pos], b[pos]]
        ground truth = y[pos]
        pred = RNNModel(x)
    end for
    #BPTT
    for pos in range(binary_dim) **do**
        compute $\nabla^L_{h_{pos}}$, $\nabla^L_{o_{pos}}$), and H_{pos}
        compute $\nabla^L_{W_{pos}}$, $\nabla^L_{U_{pos}}$, and $\nabla^L_{V_{pos}}$
    end for
    Update W, U, V
end for



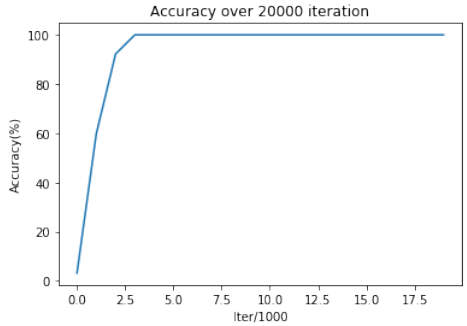
(a) Uniform distribution - Accuracy

(b) Uniform distribution - Error

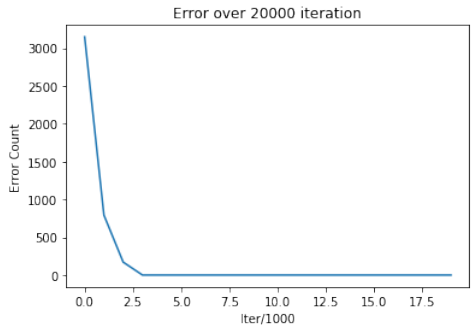
7 More you want to say

It seems that the way we initailize W, U, V affect the training result, I test the result with three kinds of initialization methods. The first is initialize weights as uniform distribution from $\frac{-1}{\sqrt{n}}$ to $\frac{1}{\sqrt{n}}$, where n is the dimention of each input. The second is initialize weights as random distribution from -1 to 1. The last is initialize weights as random distribution from 0 to 1. The results are plotted below.

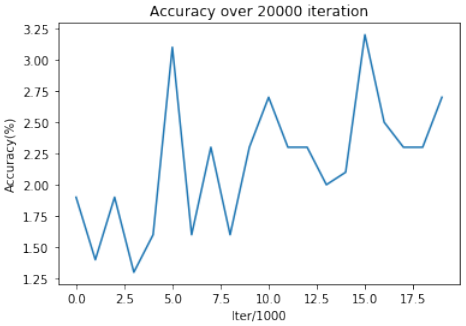
Initialize weights as uniform distribution reaches 100% accuracy after 2000 iterations, which has the fastest speed among these three method and guarantee 100% accuracy after some iterations. Initialize weights as random distribution from -1 to 1 a little bit slow than uniform distribution, reaches 100% accuracy after 3000 iterations, but can fail when weights are all larger than 0 or smaller than 0. Initialize weights as random distribution from 0 to 1 can't reach 100% accuracy even after 200000 iterations.



(c) Random distribution [-1,1) - Accuracy



(d) Uniform distribution [-1,1) - Error



(e) Uniform distribution [0,1) - Accuracy

(f) Uniform distribution [0,1) - Error