Jan Snoeijs 10/03/2018

Report on project on recurrent neural networks exercise

<u>Mathematical derivation of the backpropagation algorithm applied to the exercise.</u>

No output layer is considered, and the hidden layer contains only one neuron. The reLU(x) function is used as the function for the hidden layer.
Following specific expressions for the exercise are considered:

$$E = \frac{1}{2}(s_n - t_n)^2$$

$$\frac{dE}{ds_n} = s_n - t_n$$

With E, the output error, s_n the actual value of the output for a given sequence and at a given time and t_n the target value.

<u>Forward pass equations</u>

$$s_n = reLU(r_n)$$

$$r_n = W_{rec} \cdot s_{n-1} + W_{xs} \cdot x_n$$

x_n being the input at time step n, W_{rec} the recursive weight, W_{xs} the input weight.

<u>Backpropagation</u>

$$\frac{\partial E}{\partial W_{rec}} = \sum_n \sum_{k=0}^{n} \frac{\partial E}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W_{rec}}$$

$$\frac{\partial E}{\partial W_{rec}} = \sum_{n=0}^{N-1} \left((s_n - t_n) \sum_{k=0}^{n} \left(\frac{\partial s_k}{\partial W_{rec}} \prod_{p=k}^{n-1} \left(\frac{\partial s_{p+1}}{\partial s_p} \right) \right) \right)$$

$$\frac{\partial E}{\partial W_{rec}} = \sum_{n=0}^{N-1} \left((s_n - t_n) \sum_{k=0}^{n} \left(\frac{\partial s_k}{\partial r_k} \frac{\partial r_k}{\partial W_{rec}} \prod_{p=k}^{n-1} \left(\frac{\partial s_{p+1}}{\partial r_{p+1}} \frac{\partial r_{p+1}}{\partial s_p} \right) \right) \right)$$

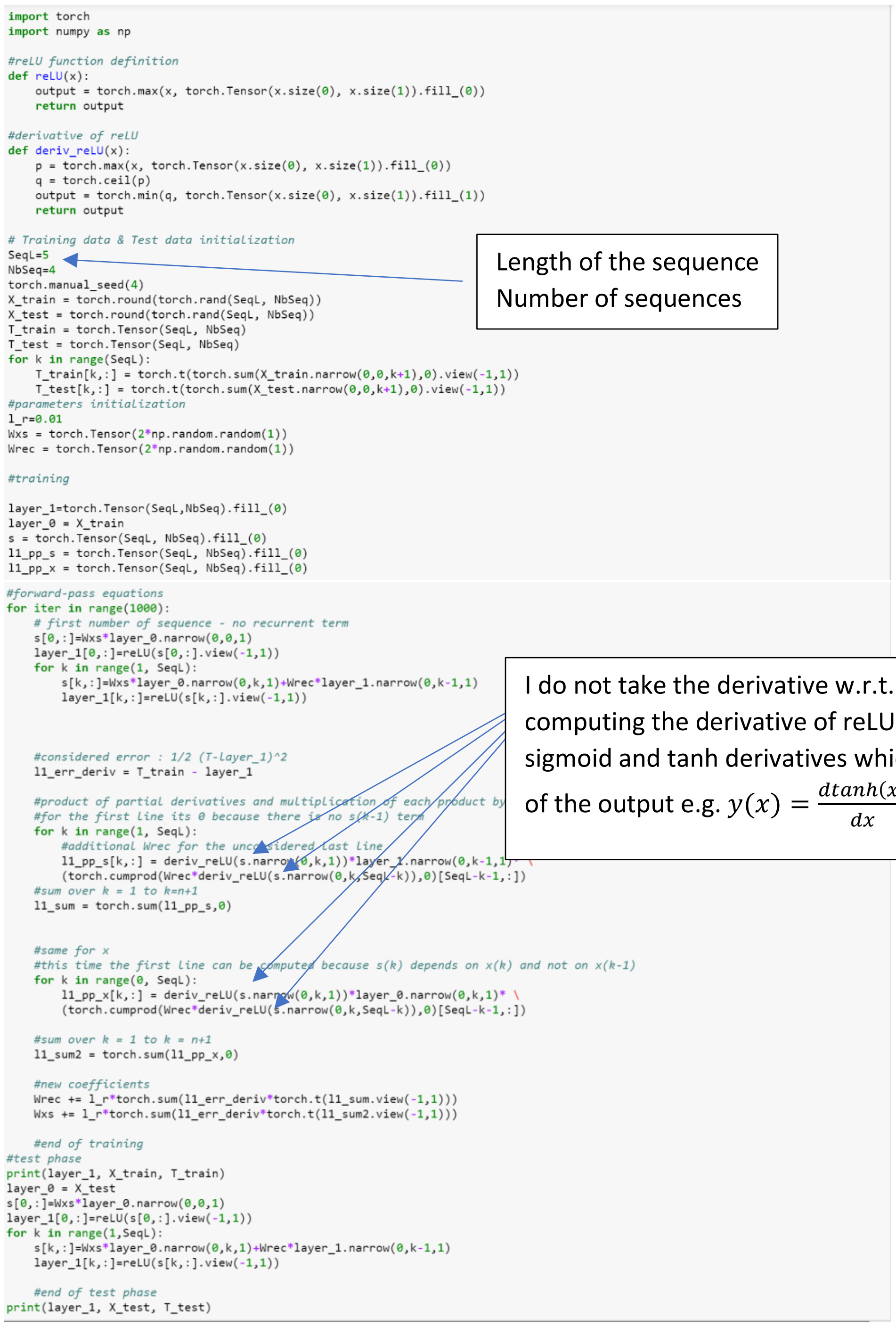
$$\frac{\partial E}{\partial W_{rec}} = \sum_{n=0}^{N-1} \left((s_n - t_n) \sum_{k=0}^{n} \left(\frac{\partial reLU(r_k)}{\partial r_k} s_{k-1} \prod_{p=k}^{n-1} \left(W_{rec} \frac{\partial reLU(r_{p+1})}{\partial r_{p+1}} \right) \right) \right)$$

Similarly, for W_{xs}:

$$\frac{\partial E}{\partial W_{xs}} = \sum_n \sum_{k=0}^{n} \frac{\partial E}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W_{xs}}$$

$$\frac{\partial E}{\partial W_{xs}} = \sum_{n=0}^{N-1}\left((s_n - t_n)\sum_{k=0}^{n}\left(\frac{\partial reLU(r_k)}{\partial r_k}x_k\prod_{p=k}^{n-1}\left(W_{rec}\frac{\partial reLU(r_{p+1})}{\partial r_{p+1}}\right)\right)\right)$$

<u>Implementation in Pytorch</u>

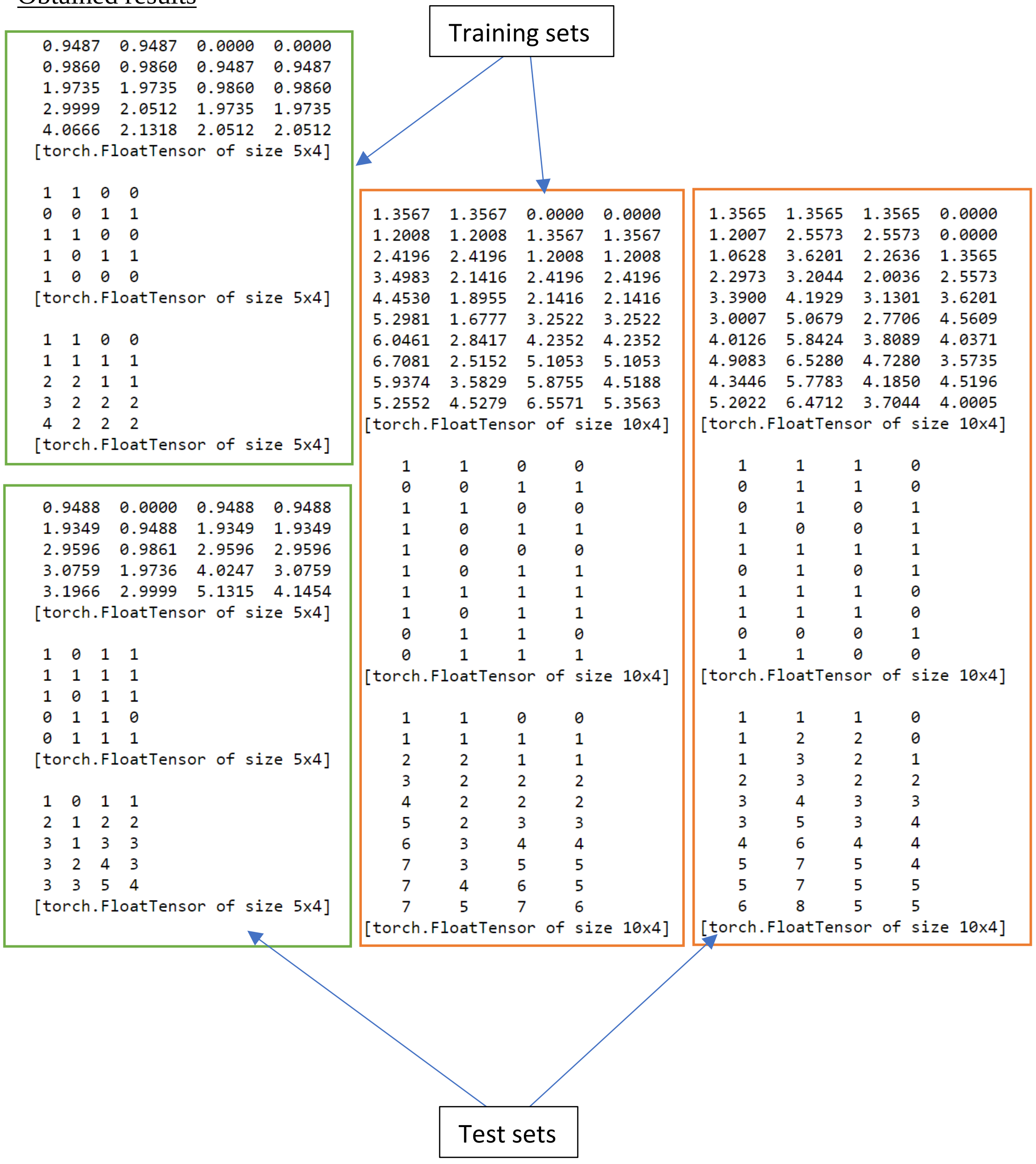


```python
import torch
import numpy as np

#reLU function definition
def reLU(x):
    output = torch.max(x, torch.Tensor(x.size(0), x.size(1)).fill_(0))
    return output

#derivative of reLU
def deriv_reLU(x):
    p = torch.max(x, torch.Tensor(x.size(0), x.size(1)).fill_(0))
    q = torch.ceil(p)
    output = torch.min(q, torch.Tensor(x.size(0), x.size(1)).fill_(1))
    return output

# Training data & Test data initialization
SeqL=5
NbSeq=4
torch.manual_seed(4)
X_train = torch.round(torch.rand(SeqL, NbSeq))
X_test = torch.round(torch.rand(SeqL, NbSeq))
T_train = torch.Tensor(SeqL, NbSeq)
T_test = torch.Tensor(SeqL, NbSeq)
for k in range(SeqL):
    T_train[k,:] = torch.t(torch.sum(X_train.narrow(0,0,k+1),0).view(-1,1))
    T_test[k,:] = torch.t(torch.sum(X_test.narrow(0,0,k+1),0).view(-1,1))
#parameters initialization
l_r=0.01
Wxs = torch.Tensor(2*np.random.random(1))
Wrec = torch.Tensor(2*np.random.random(1))

#training

layer_1=torch.Tensor(SeqL,NbSeq).fill_(0)
layer_0 = X_train
s = torch.Tensor(SeqL, NbSeq).fill_(0)
l1_pp_s = torch.Tensor(SeqL, NbSeq).fill_(0)
l1_pp_x = torch.Tensor(SeqL, NbSeq).fill_(0)

#forward-pass equations
for iter in range(1000):
    # first number of sequence - no recurrent term
    s[0,:]=Wxs*layer_0.narrow(0,0,1)
    layer_1[0,:]=reLU(s[0,:].view(-1,1))
    for k in range(1, SeqL):
        s[k,:]=Wxs*layer_0.narrow(0,k,1)+Wrec*layer_1.narrow(0,k-1,1)
        layer_1[k,:]=reLU(s[k,:].view(-1,1))



    #considered error : 1/2 (T-layer_1)^2
    l1_err_deriv = T_train - layer_1

    #product of partial derivatives and multiplication of each product by
    #for the first line its 0 because there is no s(k-1) term
    for k in range(1, SeqL):
        #additional Wrec for the unconsidered last line
        l1_pp_s[k,:] = deriv_reLU(s.narrow(0,k,1))*layer_1.narrow(0,k-1,1)* \
        (torch.cumprod(Wrec*deriv_reLU(s.narrow(0,k,SeqL-k)),0)[SeqL-k-1,:])
    #sum over k = 1 to k=n+1
    l1_sum = torch.sum(l1_pp_s,0)


    #same for x
    #this time the first line can be computed because s(k) depends on x(k) and not on x(k-1)
    for k in range(0, SeqL):
        l1_pp_x[k,:] = deriv_reLU(s.narrow(0,k,1))*layer_0.narrow(0,k,1)* \
        (torch.cumprod(Wrec*deriv_reLU(s.narrow(0,k,SeqL-k)),0)[SeqL-k-1,:])

    #sum over k = 1 to k = n+1
    l1_sum2 = torch.sum(l1_pp_x,0)

    #new coefficients
    Wrec += l_r*torch.sum(l1_err_deriv*torch.t(l1_sum.view(-1,1)))
    Wxs += l_r*torch.sum(l1_err_deriv*torch.t(l1_sum2.view(-1,1)))

    #end of training
#test phase
print(layer_1, X_train, T_train)
layer_0 = X_test
s[0,:]=Wxs*layer_0.narrow(0,0,1)
layer_1[0,:]=reLU(s[0,:].view(-1,1))
for k in range(1,SeqL):
    s[k,:]=Wxs*layer_0.narrow(0,k,1)+Wrec*layer_1.narrow(0,k-1,1)
    layer_1[k,:]=reLU(s[k,:].view(-1,1))

    #end of test phase
print(layer_1, X_test, T_test)
```

Length of the sequence
Number of sequences

I do not take the derivative w.r.t. layer_1 because the function computing the derivative of reLU(x) is directly defined unlike the sigmoid and tanh derivatives which were defined as an expression of the output e.g. $y(x) = \frac{dtanh(x)}{dx} = 1 - y(x)^2$

Obtained results



```
 0.9487  0.9487  0.0000  0.0000
 0.9860  0.9860  0.9487  0.9487
 1.9735  1.9735  0.9860  0.9860
 2.9999  2.0512  1.9735  1.9735
 4.0666  2.1318  2.0512  2.0512
[torch.FloatTensor of size 5x4]

 1  1  0  0
 0  0  1  1
 1  1  0  0
 1  0  1  1
 1  0  0  0
[torch.FloatTensor of size 5x4]

 1  1  0  0
 1  1  1  1
 2  2  1  1
 3  2  2  2
 4  2  2  2
[torch.FloatTensor of size 5x4]
```

```
 1.3567  1.3567  0.0000  0.0000
 1.2008  1.2008  1.3567  1.3567
 2.4196  2.4196  1.2008  1.2008
 3.4983  2.1416  2.4196  2.4196
 4.4530  1.8955  2.1416  2.1416
 5.2981  1.6777  3.2522  3.2522
 6.0461  2.8417  4.2352  4.2352
 6.7081  2.5152  5.1053  5.1053
 5.9374  3.5829  5.8755  4.5188
 5.2552  4.5279  6.5571  5.3563
[torch.FloatTensor of size 10x4]

    1     1     0     0
    0     0     1     1
    1     1     0     0
    1     0     1     1
    1     0     0     0
    1     0     1     1
    1     1     1     1
    1     0     1     1
    0     1     1     0
    0     1     1     1
[torch.FloatTensor of size 10x4]

    1     1     0     0
    1     1     1     1
    2     2     1     1
    3     2     2     2
    4     2     2     2
    5     2     3     3
    6     3     4     4
    7     3     5     5
    7     4     6     5
    7     5     7     6
[torch.FloatTensor of size 10x4]
```

```
 1.3565  1.3565  1.3565  0.0000
 1.2007  2.5573  2.5573  0.0000
 1.0628  3.6201  2.2636  1.3565
 2.2973  3.2044  2.0036  2.5573
 3.3900  4.1929  3.1301  3.6201
 3.0007  5.0679  2.7706  4.5609
 4.0126  5.8424  3.8089  4.0371
 4.9083  6.5280  4.7280  3.5735
 4.3446  5.7783  4.1850  4.5196
 5.2022  6.4712  3.7044  4.0005
[torch.FloatTensor of size 10x4]

    1     1     1     0
    0     1     1     0
    0     1     0     1
    1     0     0     1
    1     1     1     1
    0     1     0     1
    1     1     1     0
    1     1     1     0
    0     0     0     1
    1     1     0     0
[torch.FloatTensor of size 10x4]

    1     1     1     0
    1     2     2     0
    1     3     2     1
    2     3     2     2
    3     4     3     3
    3     5     3     4
    4     6     4     4
    5     7     5     4
    5     7     5     5
    6     8     5     5
[torch.FloatTensor of size 10x4]
```

```
 0.9488  0.0000  0.9488  0.9488
 1.9349  0.9488  1.9349  1.9349
 2.9596  0.9861  2.9596  2.9596
 3.0759  1.9736  4.0247  3.0759
 3.1966  2.9999  5.1315  4.1454
[torch.FloatTensor of size 5x4]

 1  0  1  1
 1  1  1  1
 1  0  1  1
 0  1  1  0
 0  1  1  1
[torch.FloatTensor of size 5x4]

 1  0  1  1
 2  1  2  2
 3  1  3  3
 3  2  4  3
 3  3  5  4
[torch.FloatTensor of size 5x4]
```

In each rectangle, the 1st matrix is the actual output of the network, the 2nd is the input and the 3rd is the output. One column corresponds to one sequence, and each line corresponds to a given time step. For a learning rate of 0.001, 1000 iterations and a sequence length of 5, the algorithm manages to output correct results and struggles for the same parameters but a longer sequence of 10. Generally, for a sequence length of 5, the algorithm always outputs correct results whereas for too long sequences

Remark on the code: I didn't compute a double sum for the errors as in the mathematical definition. What the code does is simply a loop over the whole sequence, which is not exactly the same. I will try to implement that as well during the coming week and see if it gives more precise results or if it converges faster with the same learning rate and number of iterations.

Extract of 2nd version of the implementation, including a double sum for new coefficient calculation:

$$\sum_{n} \sum_{k=0}^{n}$$

```
    for n in range(SeqL):

        #product of partial derivatives and multiplication of each product by s(k-1):
        #for the first line its 0 because there is no s(k-1) term
        for k in range(1, n):
            #additional Wrec for the unconsidered last line
            l1_pp_s[k,:] = deriv_reLU(s.narrow(0,k,1))*layer_1.narrow(0,k-1,1)* \
            (torch.cumprod(Wrec*deriv_reLU(s.narrow(0,k,n-k)),0)[n-k-1,:])

        #sum over k = 1 to k=n+1
        l1_sum += torch.sum(l1_pp_s,0)

        #same for x
        #this time the first line can be computed because s(k) depends on x(k) and not on x(k-1)
        for k in range(0, n):
            l1_pp_x[k,:] = deriv_reLU(s.narrow(0,k,1))*layer_0.narrow(0,k,1)* \
            (torch.cumprod(Wrec*deriv_reLU(s.narrow(0,k,n-k)),0)[n-k-1,:])

        #sum over k = 1 to k = n+1
        l1_sum2 += torch.sum(l1_pp_x,0)

    #new coefficients
    Wrec += l_r*torch.sum(l1_err_deriv*torch.t((l1_sum/SeqL).view(-1,1)))
    Wxs += l_r*torch.sum(l1_err_deriv*torch.t((l1_sum2/SeqL).view(-1,1)))
```

Division over the sequence to average the double sum otherwise the learning rate must be decreased by a large amount to observe convergence.

The 2nd implementation gives more precise results since the new weights are calculated considering that the contribution of each number in the sequence is not equal in magnitude on average, but instead this contribution is more important if the corresponding number appears later in the sequence, which in other terms means that the correction of the error is more important towards the end of the sequence than at the start.