Using RNNs to learn to add 2 binary strings together

Madhur Tandon

Speedy Intro to Neural Networks

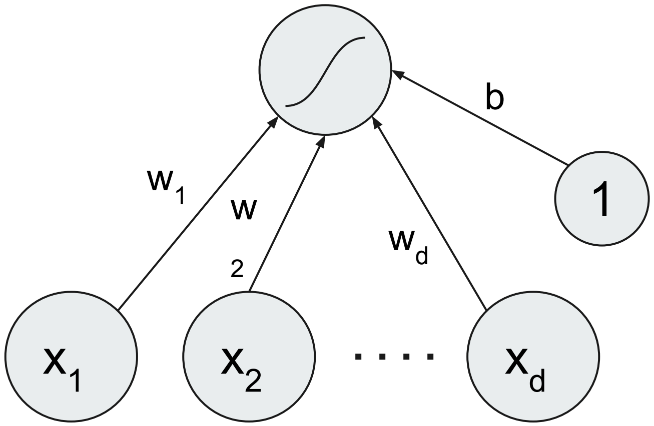
Neuron pre-activation (input-activation):

$a(x) = b + \sum_i w_i x_i = b + W^T x$

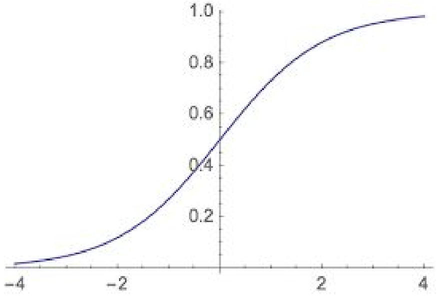
Neuron (output) activation:

$h(x) = g(a(x)) = g(b + \sum_i w_i x_i)$

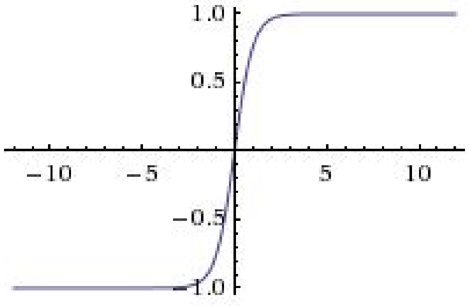
W are connection weights
b is neuron bias
g(x) is the activation function



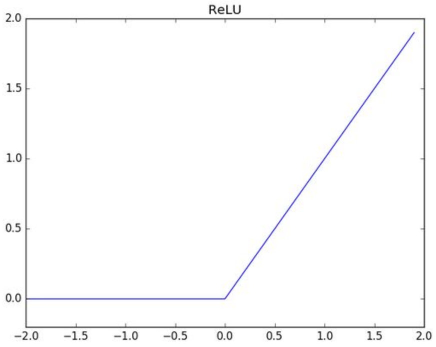
Popular Activation Functions



Sigmoid



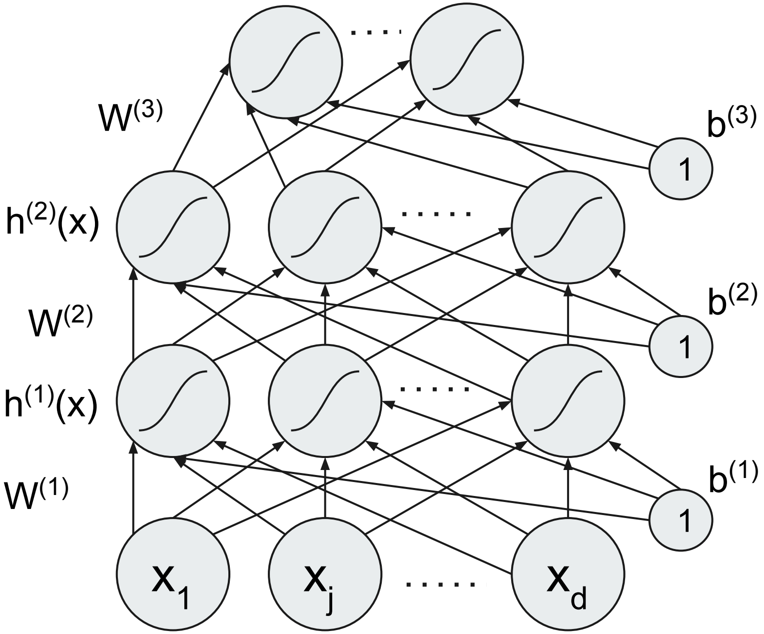
tanh



ReLU

Multi-Layer Neural Network

Could Have L hidden layers



Learning

During a forward pass, the network predicts the output.

A loss function acts as a distance metric between the outputs and the predicted values.

Gradients a.k.a derivatives are computed for the Weights w.r.t the loss function.

The gradients give a sense of direction as to where the weights must be updated.

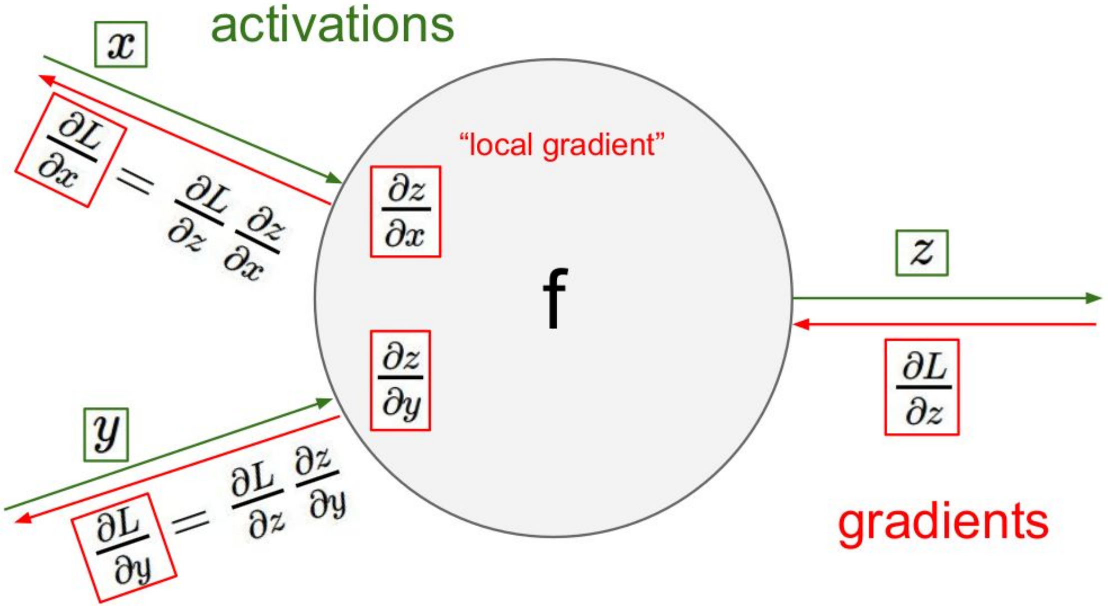
Math Formulation

$\arg\min_{\theta} (1/T) \sum_t L(f(x^{(t)}; \theta), y^{(t)}) + \lambda\phi(\theta)$

$L(f(x^{(t)}; \theta), y^{(t)})$ is the loss function

$\phi(\theta)$ is a regularizer, penalizes certain values of θ

Backpropagation



Gradient Descent

Shift Parameters in Opposite Direction to Minimize Loss

$$\text{Repeat until convergence } \{$$

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

RNNs (Recurrent Neural Networks)

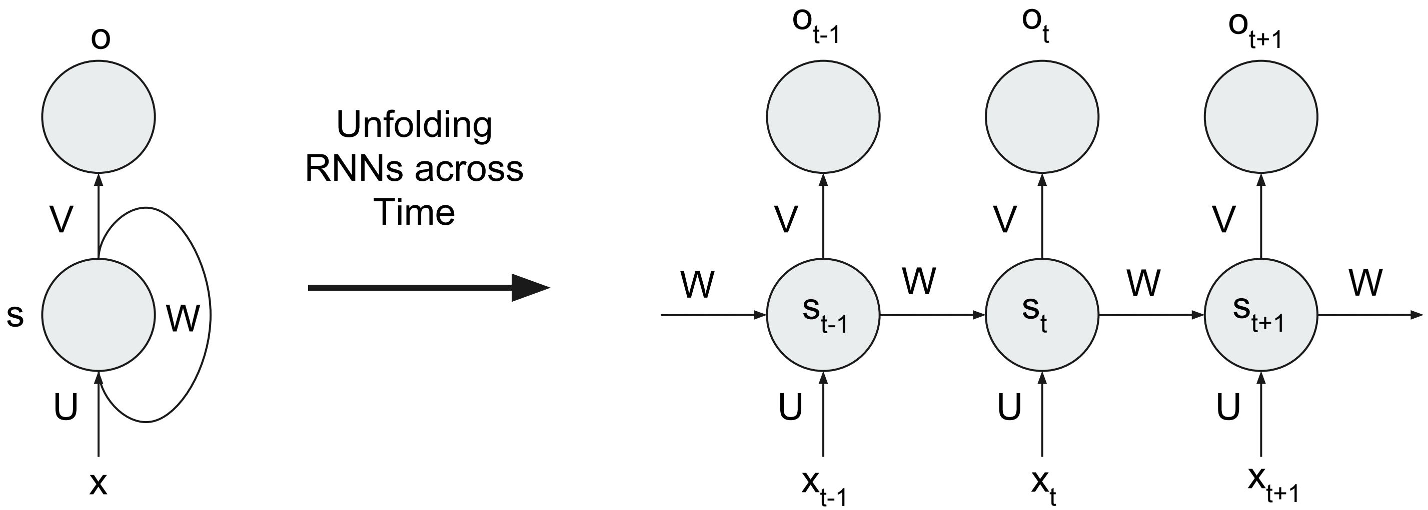
Take the previous output or hidden states as inputs.

The composite input at time t has some historical information about the happenings at time $T < t$.

Used for Sequence to Sequence Problems

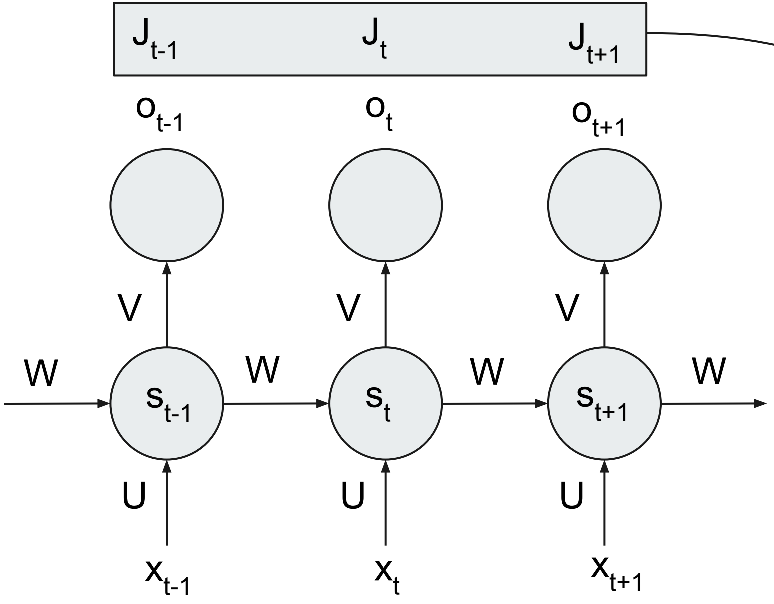
Convert $x_1, x_2, \ldots, x_n$ to $y_1, y_2, \ldots, y_t$

RNN Model



o
V
s
W
U
x
Unfolding
RNNs across
Time
o t-1
o t
o t+1
V
V
V
W
s t-1
W
s t
W
s t+1
W
U
U
U
x t-1
x t
x t+1

Backpropagation through Time!



We Sum the Losses at each Timestep and also the gradients at each time step!

Gradient of J_{t+1} w.r.t parameters U:

grad(J_{t+1},U) =
grad(J_{t+1},o_{t+1})*grad(o_{t+1},s_{t+1})***grad(s_{t+1}, U)**

The last term here is not a constant, $\mathbf{s_{t+1}}$ depends upon the previous state $\mathbf{s_t}$ which further depends on previous hidden states.

RNNs are hard to train!

Problem : Vanishing Gradient

Backpropagate all the way back to the initial time-step becomes really long!

The chain rule product gets longer and longer. Each of the derivatives are small numbers and thus, we are multiplying a lot of small numbers together!

Errors due to further back time-steps have smaller gradients.

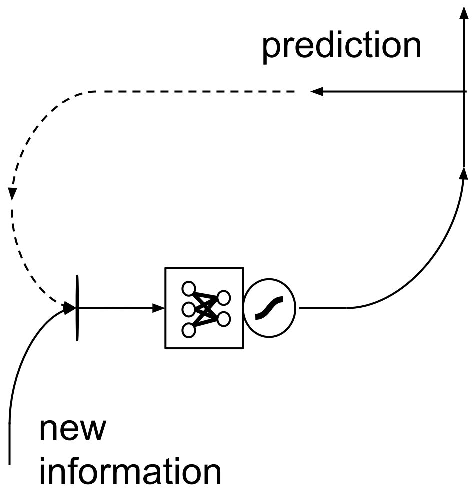
Parameters become biased to capture short-term dependencies.

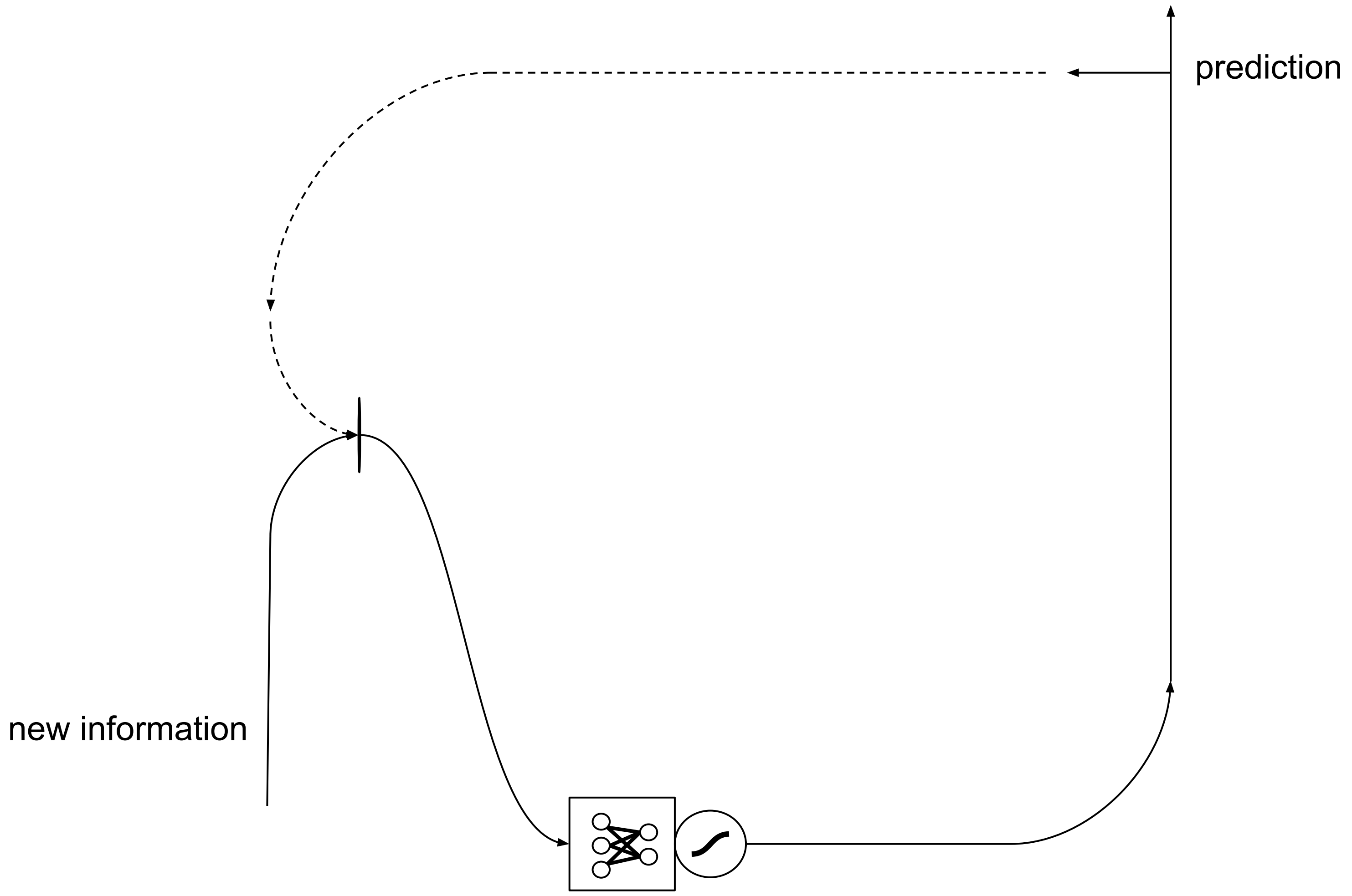
In France, I had a great time and I learnt some of the _________ language.

LSTMs (Long Short Term Memory) - Intuition

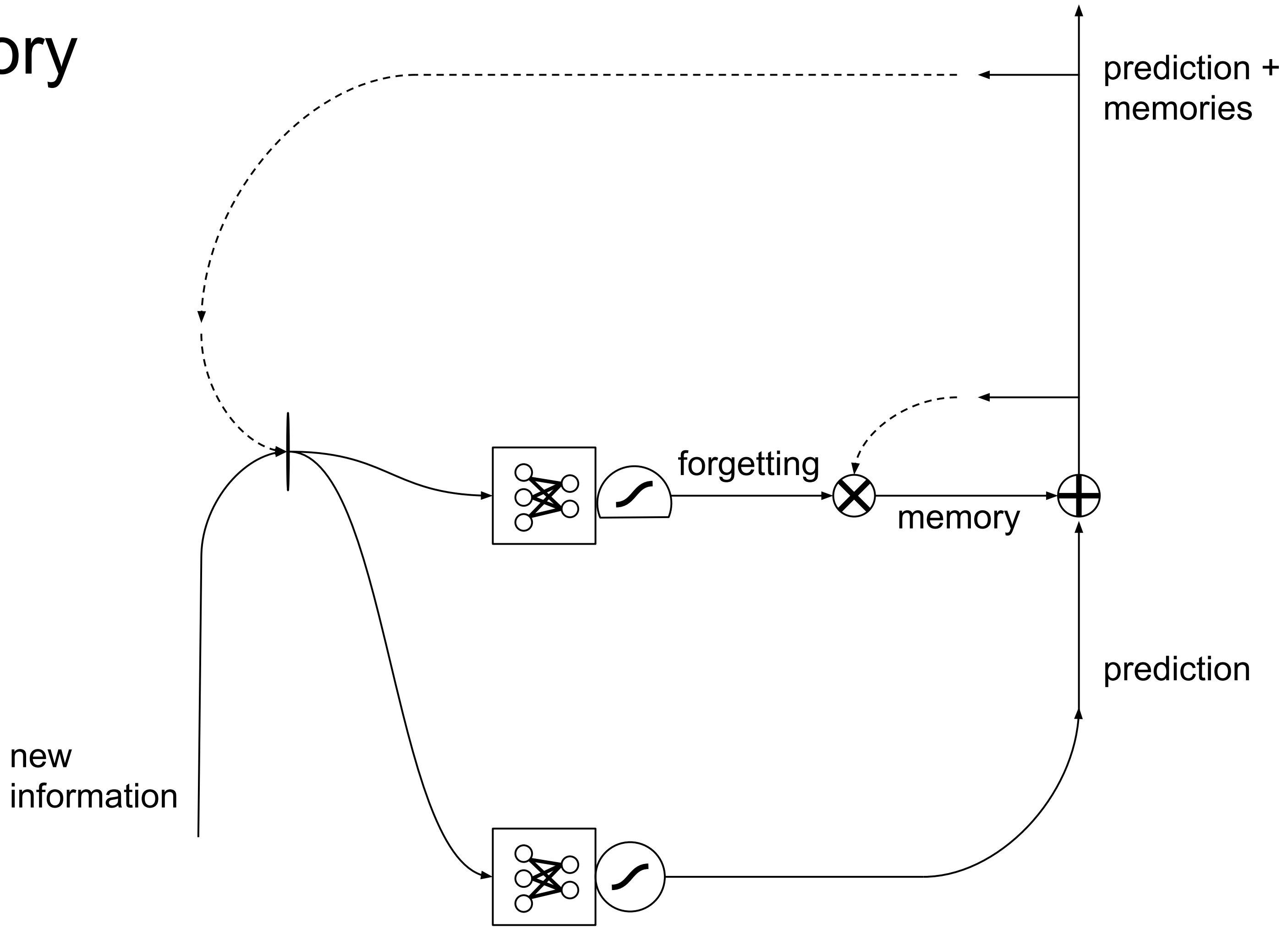
Replace each RNN node with Gated Cells that control what information is passed through.

Recall RNNs

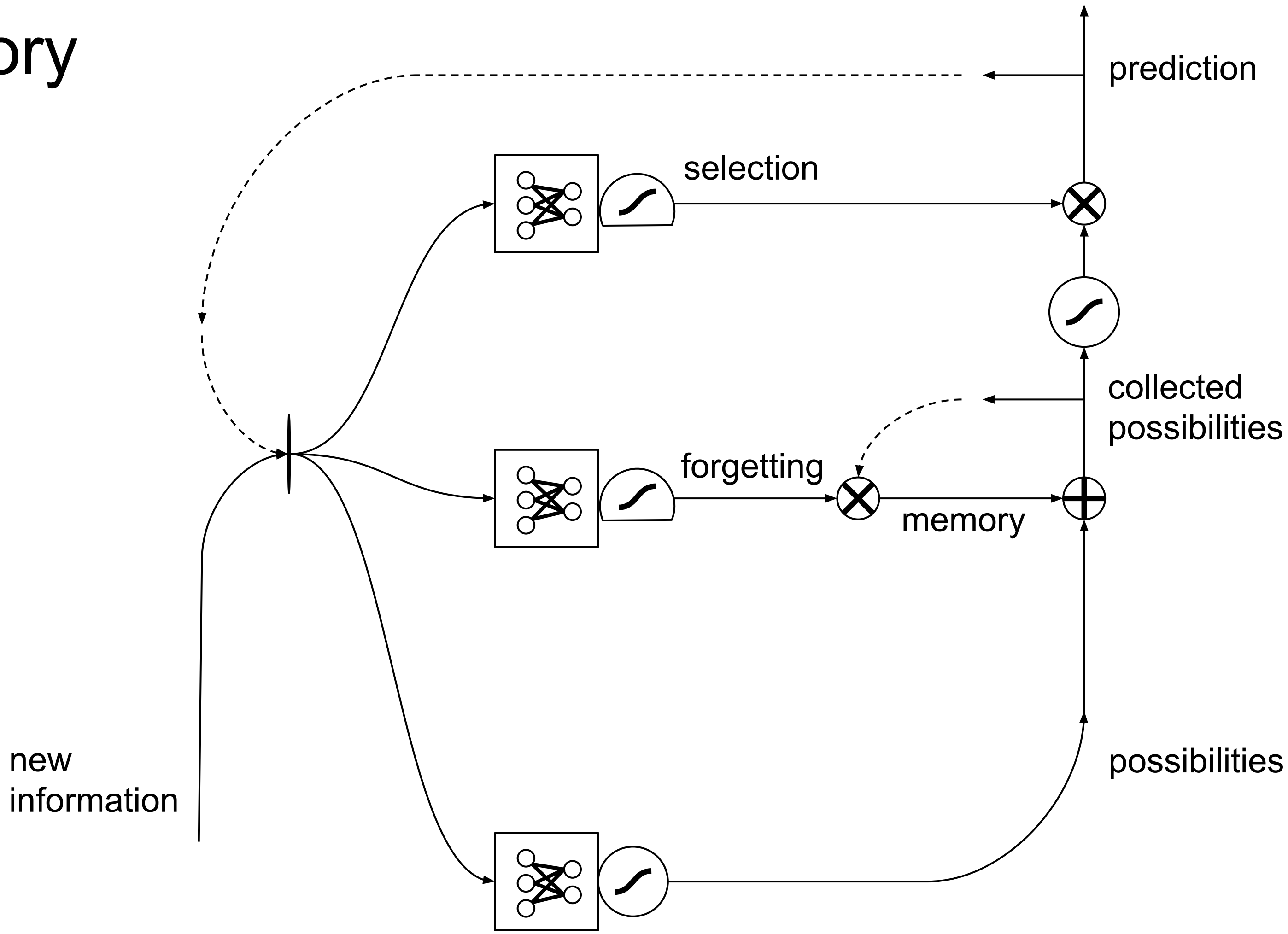



prediction
new information

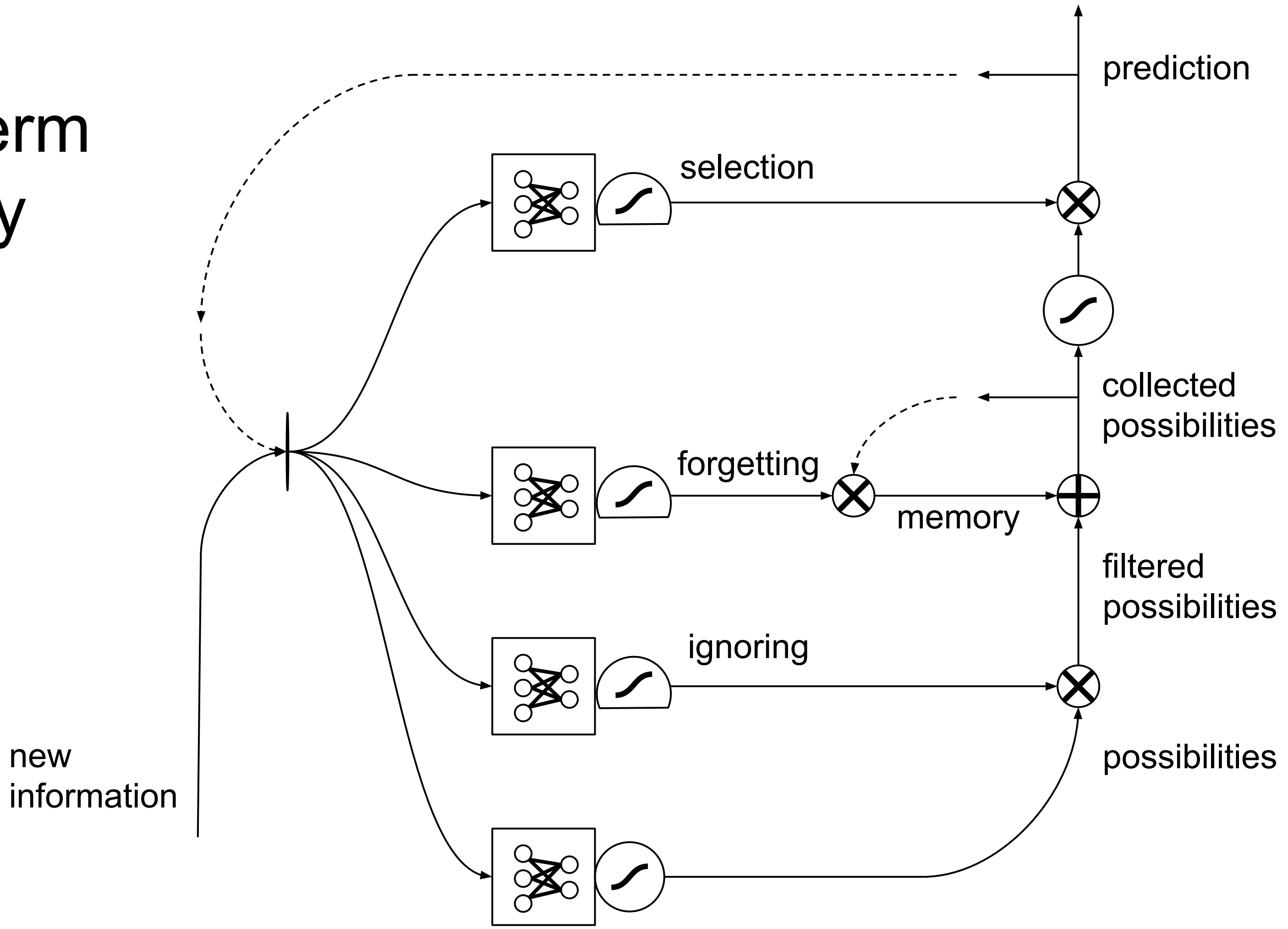
memory


prediction +
memories
forgetting
memory
prediction
new
information

memory


prediction
selection
collected
possibilities
forgetting
memory
new
information
possibilities

long short-term memory



Using RNNs to Add 2 Binary Strings

> **Recall that RNNs are used for Sequence to Sequence Problems. We use some special kinds of Representations for different types of Inputs that we might wanna pass into an RNN Model.**
>
> **The popular representations for different kinds of Data are given below.**
>
> ***1. Domain Specific Features (Capture Structure in the Data)***
>
> - ConvNet Fully Connected Feature Vectors for Images
> - Word2Vec Features for Words / Language Tasks
> - MFCC (Mel Frequency Cepstral Coefficents) features for Audio / Speech
> - PHOC (Pyramidal Histogram of Characters) for OCR
>
> ***2. One-hot Encoding for tokens from a fixed corpus / lexicon***
>
> - Word-Level (Usually a large lexicon of words ~100k)
> - Character-Level (Vocabulary Size is much smaller)
>   - Only 36 characters in English vs > 100k words

However, In this tutorial, *we are not gonna focus on the dynamics of representing Inputs*, but divert our attention to the intuition behind what RNNs can essentially do!

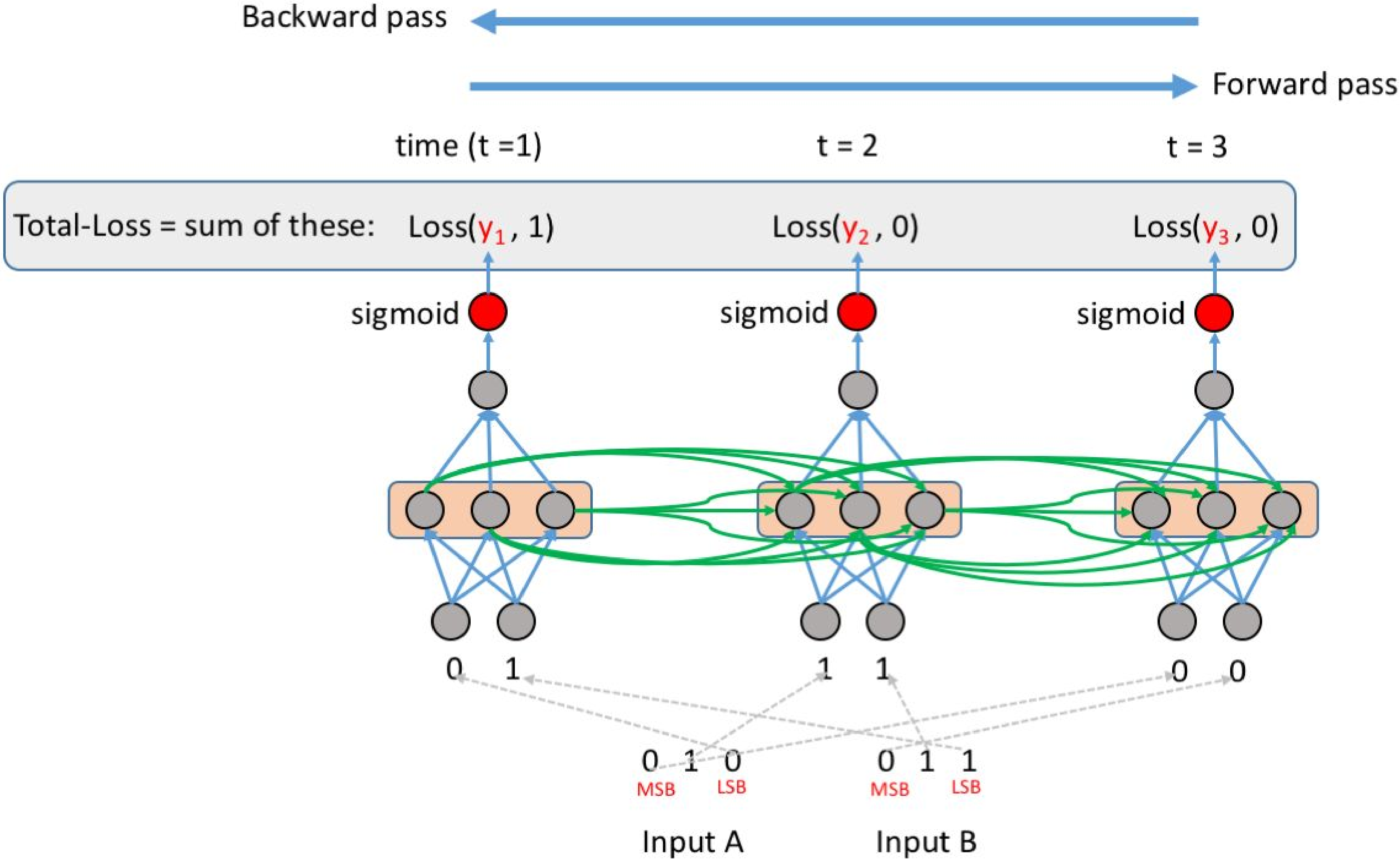
Recall Binary Addition

Binary addition moves ***from*** the right-most bit (least-significant bit or ***LSB***) towards the left-most bit (most-significant bit or ***MSB***), with a ***carry bit passed from the previous addition.***

x	y	Carry - In	Sum	Carry - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The RNN is **fed two bit-sequences** and the **target Sum** sequence.

> The sequence is **ordered from LSB to MSB**, i.e., **time-step 1 (t=1) corresponds to LSB**, and the **last time-step is the MSB.**
>
> **For example:**
>
> If the bit strings **010 (integer value = 2)** and **011 (integer value = 3)** are to be added to produce the **sum 101 (integer value 5)**, the following is the sequence of inputs and targets to the RNN when training:
>
> | Time | x | y | Output |
> |---|---|---|---|
> | 1 | 0 | 1 | 1 |
> | 2 | 1 | 1 | 0 |
> | 3 | 0 | 0 | 1 |
>
> In the example above, the **carry bit is not explicitly provided as the input**, and the **RNN has to *learn* the concept of a carry-bit.**


Backward pass
Forward pass
time (t =1)
t = 2
t = 3
Total-Loss = sum of these:
Loss(y1 , 1)
Loss(y2 , 0)
Loss(y3 , 0)
sigmoid
sigmoid
sigmoid
0 1
1 1
0 0
0 1 0
MSB
LSB
0 1 1
MSB
LSB
Input A
Input B

```python
"""
Necessary Imports
"""
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
from time import sleep
import random
import sys
import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```python
def getSample(BitStringLength,testFlag=0):
    """
    Returns a random sample for bit-string addition

    BitStringLength : (int scalar) (one less than) length of the bit-string to return.
    testFlag (boolean) if True, the returned sample is printed.

    Returns:
        a 2-tuple of (Input,Output), where:
        INPUT: (L+1 x 2) dimensional tensor of the inputs, where L==BitStringLength
        OUTPUT: (L+1) dimensional "target" vector, which is the binary sum of inputs.
    """
    lowerBound=pow(2,BitStringLength-1)+1
    upperBound=pow(2,BitStringLength)

    firstNum = random.randint(lowerBound,upperBound)
    secondNum = random.randint(lowerBound,upperBound)

    thirdNum = firstNum + secondNum

    firstNum_binary = bin(firstNum)[2:]
    secondNum_binary = bin(secondNum)[2:]
    thirdNum_binary = bin(thirdNum)[2:]

    firstNum_binary = '0'*(len(thirdNum_binary)-len(firstNum_binary)) + firstNum_binary
    secondNum_binary = '0'*(len(thirdNum_binary)-len(secondNum_binary)) + secondNum_binary
```

```
if testFlag==1:
    print('input numbers and their sum  are', firstNum, ' ', secondNum, ' ', thirdNum)
    print ('binary strings are', firstNum_binary, ' ' , secondNum_binary, ' ' , thirdNum_binary)

input_strings = np.zeros((len(thirdNum_binary),2),dtype=np.int)
for i in range(len(thirdNum_binary)):
    input_strings[i,0] = firstNum_binary[len(thirdNum_binary)-1-i]
    input_strings[i,1] = secondNum_binary[len(thirdNum_binary)-1-i]

output_string = np.array(list(map(int,list(thirdNum_binary[::-1]))))

print(firstNum, secondNum, thirdNum)

return input_strings, output_string
```

```
print(getSample(3))
```

```
6 7 13
(array([[0, 1],
       [1, 1],
       [1, 1],
       [0, 0]]), array([1, 0, 1, 1]))
```

```python
class RNN(nn.Module):
    def __init__(self, state_dimension):
        super(RNN, self).__init__()
        self.state_dimension = state_dimension
        self.input_dimension = 2
        self.output_dimension = 1
        self.LSTM = nn.LSTM(self.input_dimension, self.state_dimension)
        self.out_fc_layer = nn.Linear(self.state_dimension, self.output_dimension)

    def forward(self, x):
        LSTM_output, _ = self.LSTM(x)
        L,B,D  = LSTM_output.size(0),LSTM_output.size(1),LSTM_output.size(2)
        LSTM_output = LSTM_output.contiguous()
        LSTM_output = LSTM_output.view(L*B,D)
        prediction = self.out_fc_layer(LSTM_output)
        prediction = prediction.view(L,B,-1).squeeze(1)
        return prediction
```

```python
while num_epochs < min_epochs:
    print("[epoch %d/%d] Avg. Loss for last 500 samples = %lf"%(num_epochs+1,min_epochs,totalLoss))
    num_epochs += 1
    totalLoss = 0
    for i in range(0,iterations):
        # get a new random training sample:
        x,y = getSample(stringLen)
        # zero the gradients from the previous time-step:
        model.zero_grad()
        # convert to torch tensor and variable:
        # unsqueeze() is used to add the extra BATCH dimension:
        x_var = autograd.Variable(torch.from_numpy(x).unsqueeze(1).float())
        seqLen = x_var.size(0)
        x_var = x_var.contiguous()
        y_var = autograd.Variable(torch.from_numpy(y).float())
        # push the inputs through the RNN (this is the forward pass):
        pred = model(x_var)
        # compute the loss:
        loss = lossFunction(pred,y_var)
        totalLoss += loss.data[0]
        optimizer.zero_grad()
        # perform the backward pass:
        loss.backward()
        # update the weights:
        optimizer.step()
    totalLoss=totalLoss/iterations
print('Training finished!')
```

```python
# size of the hidden RNN state
stateSize = 10

stringLen = 3

model = RNN(stateSize)

lossFunction = nn.MSELoss()

optimizer = optim.Adam(model.parameters(),lr=0.01)
iterations = 500
min_epochs = 20
num_epochs,totalLoss = 0,float("inf")
```

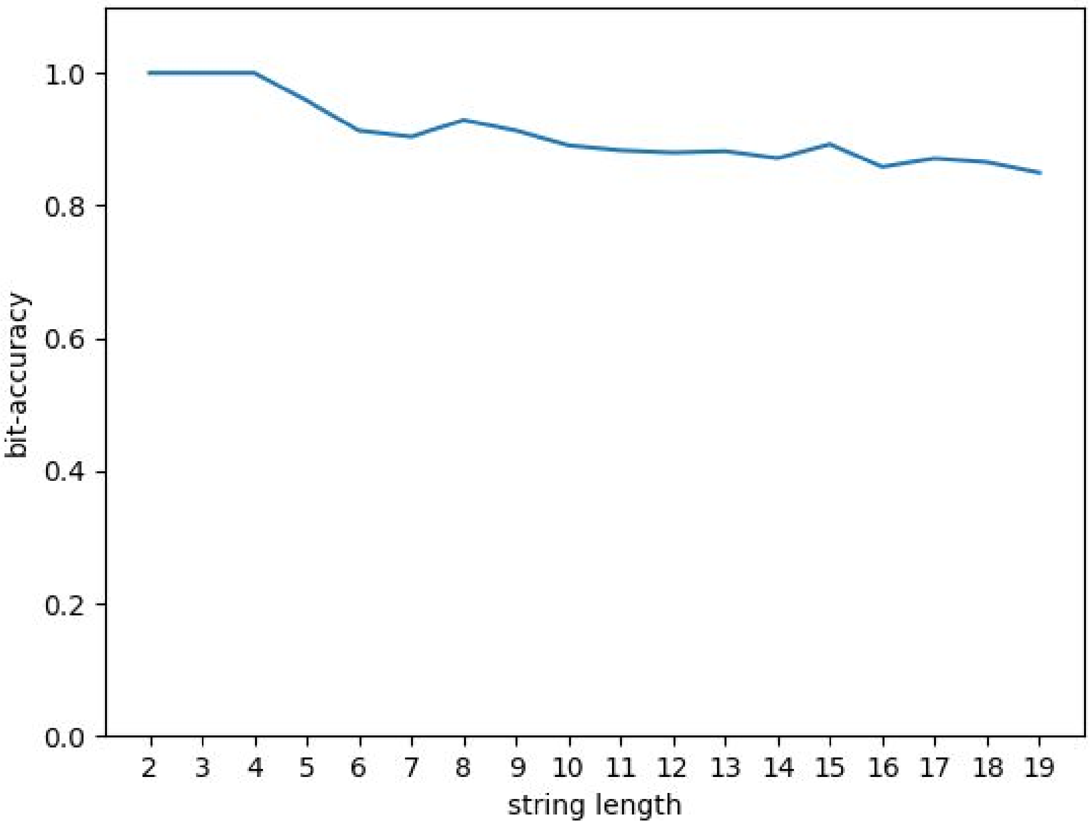
```
[epoch 1/20] Avg. Loss for last 500 samples = inf
[epoch 2/20] Avg. Loss for last 500 samples = 0.164815
[epoch 3/20] Avg. Loss for last 500 samples = 0.055039
[epoch 4/20] Avg. Loss for last 500 samples = 0.011035
[epoch 5/20] Avg. Loss for last 500 samples = 0.001997
[epoch 6/20] Avg. Loss for last 500 samples = 0.000348
[epoch 7/20] Avg. Loss for last 500 samples = 0.000102
[epoch 8/20] Avg. Loss for last 500 samples = 0.000171
[epoch 9/20] Avg. Loss for last 500 samples = 0.000219
[epoch 10/20] Avg. Loss for last 500 samples = 0.001171
[epoch 11/20] Avg. Loss for last 500 samples = 0.000091
[epoch 12/20] Avg. Loss for last 500 samples = 0.000634
[epoch 13/20] Avg. Loss for last 500 samples = 0.000116
[epoch 14/20] Avg. Loss for last 500 samples = 0.000268
[epoch 15/20] Avg. Loss for last 500 samples = 0.000680
[epoch 16/20] Avg. Loss for last 500 samples = 0.000525
[epoch 17/20] Avg. Loss for last 500 samples = 0.000152
[epoch 18/20] Avg. Loss for last 500 samples = 0.000140
[epoch 19/20] Avg. Loss for last 500 samples = 0.000495
[epoch 20/20] Avg. Loss for last 500 samples = 0.000209
Training finished!
```

```python
def test_by_length(stringLen,n_samples=100,verbose=True):
    n_samples = min(n_samples,2**stringLen)
    total_correct, num_bits = 0,0
    for i in range(n_samples):
        x,y = getSample(stringLen,testFlag=verbose)
        x_var = autograd.Variable(torch.from_numpy(x).unsqueeze(1).float())
        y_var = autograd.Variable(torch.from_numpy(y).float())
        seqLen = x_var.size(0)
        x_var = x_var.contiguous()
        finalScores = model(x_var).data.t().numpy()
        # to get the final predictions, threshold the output of RNN at 0.5:
        bits = (finalScores > 0.5).astype(np.int32)
        # calculate the accuracy:
        y_pred = bits
        corr = y_pred==y; total_correct += np.sum(corr); num_bits += len(y)
        if verbose:
            print('sum predicted by RNN is ',y_pred)
            print('bit-accuracy : %s'%(np.sum(corr)/(len(y)+0.0)))
            print(40*'*')
    accuracy = total_correct / (num_bits + 0.0)
    if verbose:
        print(40*'*')
        print('Final bit-accuracy for strings of length %d = %.3f'%(stringLen,accuracy))
        print(40*'*')
    return accuracy
```

```
test_by_length(7)
```

```
****************************************
input numbers and their sum  are 92   111   203
binary strings are 01011100   01101111   11001011
92 111 203
sum predicted by RNN is  [[1 1 0 1 0 1 0 1]]
bit-accuracy : 0.75
****************************************
```

```python
string_len = np.arange(2,20)
# set "verbose" to true to print out detailed information:
bit_accuracy = [test_by_length(l,verbose=False,n_samples=100) for l in string_len]
# plot the accuracy:
plt.plot(string_len,bit_accuracy)
plt.xlabel('string length'); plt.ylabel('bit-accuracy'); plt.xticks(string_len,string_len)
plt.ylim([0,1.1]);
```



1.0
0.8
0.6
0.4
0.2
0.0
bit-accuracy
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
string length

Thank You!