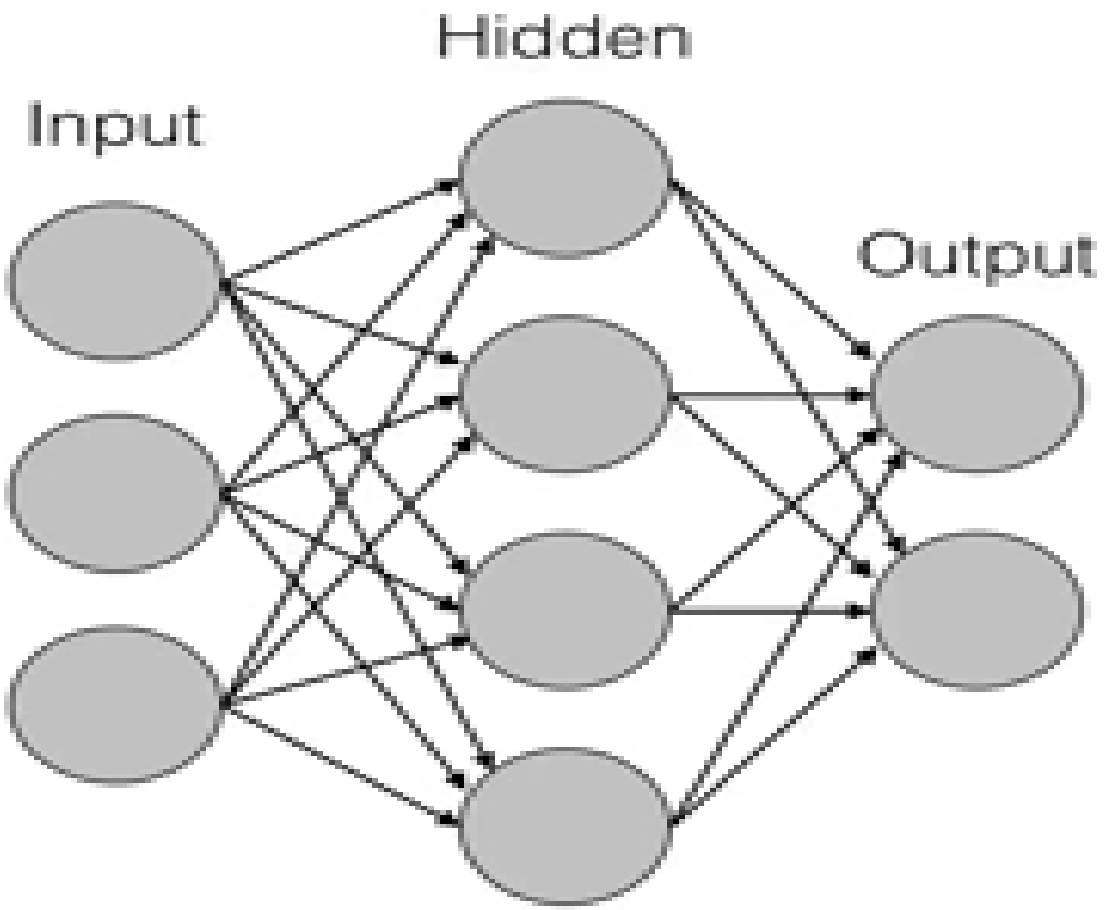
Sequence Modelling with RNN

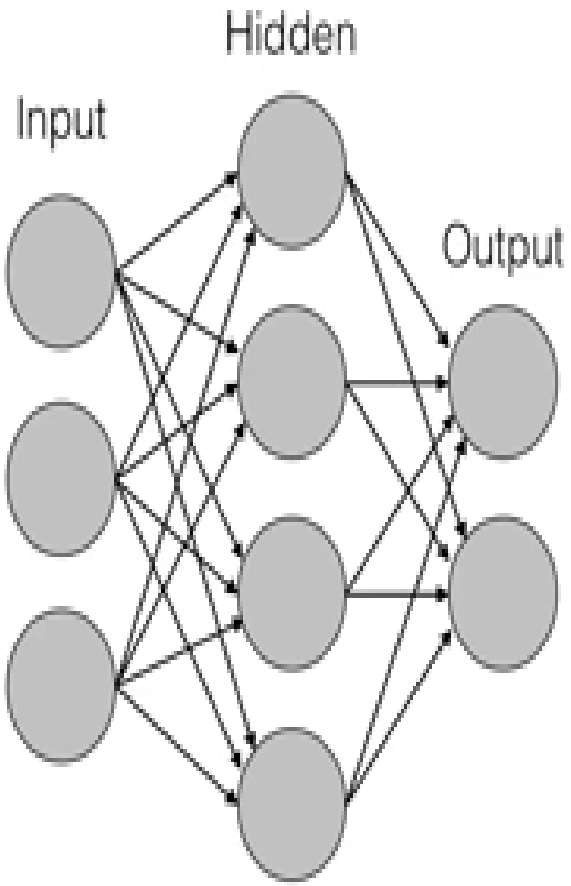
By Mohit Kumar

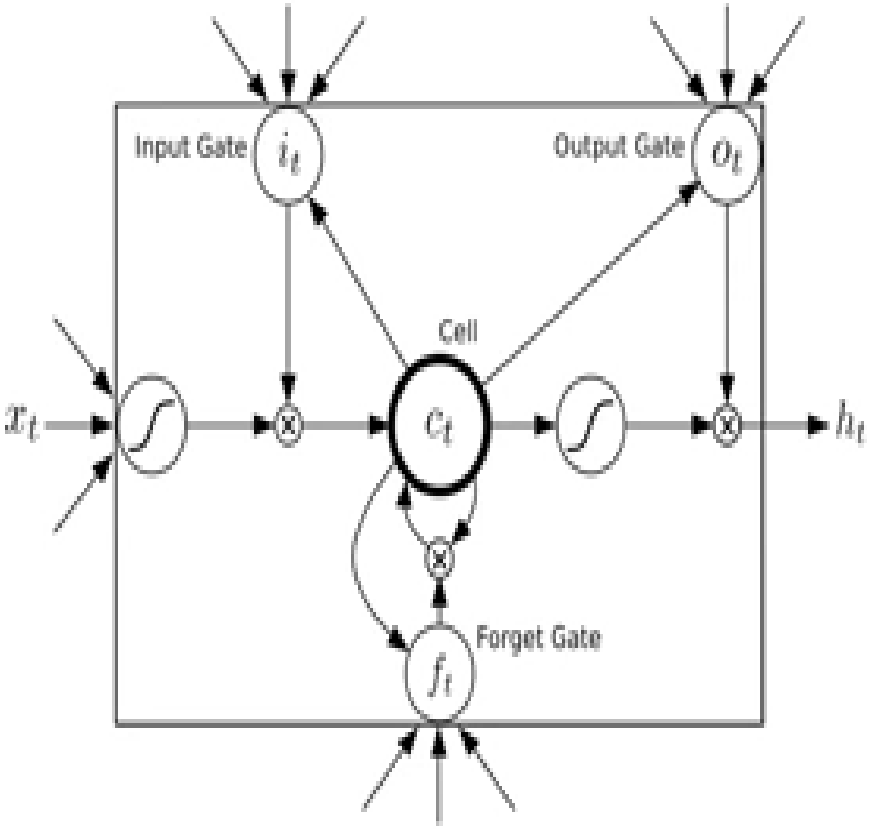
Recurrent Neural Networks:why?



- Independence
- Fixed Length

Recurrent Neural Networks:why?





- Independence
- Fixed Length

- Temporal dependencies
- Variable sequence length

Recurrent Neural Networks:Recurrent Neuron

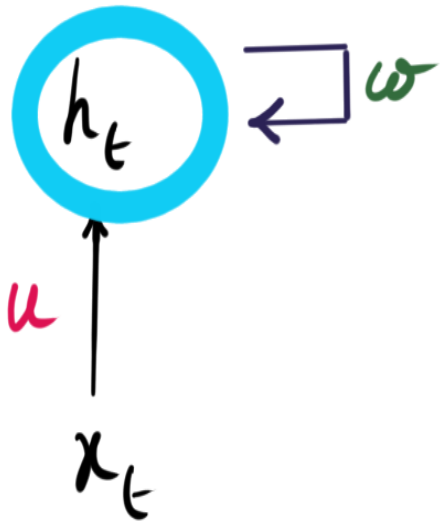
t = time step

x_t = input at time step - t

h_t = state at time step - t

u = input to hidden weight

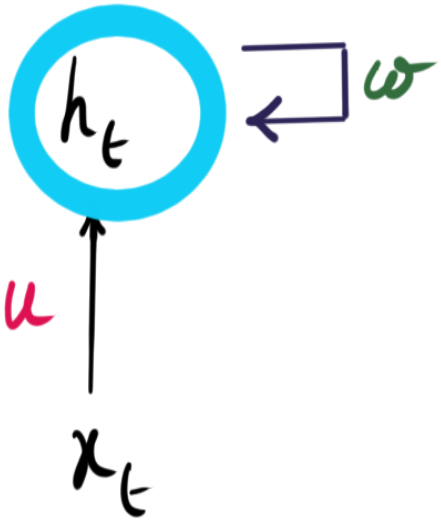
Recurrent Neural Networks:Recurrent Neuron



t = time step
x_t = input at time step - t
h_t = state at time step - t
u = input to hidden weight
w = hidden to hidden weight

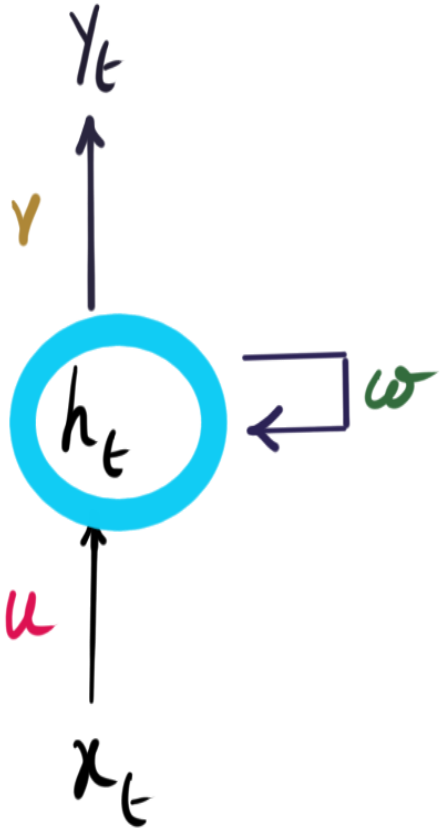
Recurrent Neural Networks:Recurrent Neuron

$$h_t = \tanh(u \cdot x_t + w \cdot h_{(t-1)} + b_h)$$



t = time step
x_t = input at time step - t
h_t = state at time step - t
u = input to hidden weight
w = hidden to hidden weight

Recurrent Neural Networks:Recurrent Neuron



$$h_t = \tanh(u \cdot x_t + w \cdot h_{(t-1)} + bh)$$

$$y_t = \text{softmax}(v \cdot ht)$$

t = time step

x_t = input at time step - t

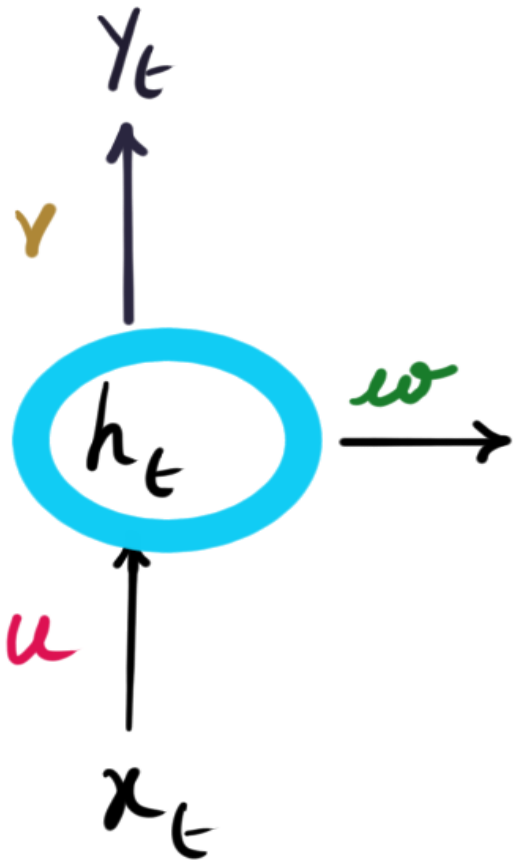
h_t = state at time step - t

u = input to hidden weight

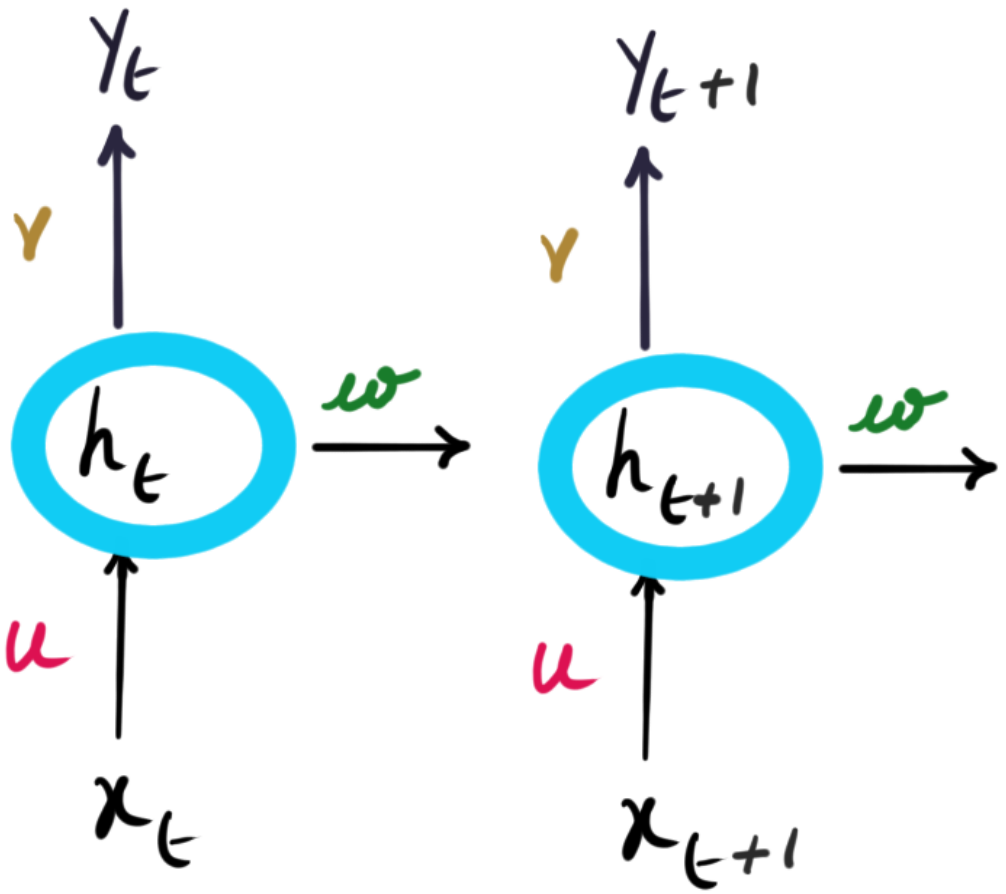
w = hidden to hidden weight

v = hidden to output weight

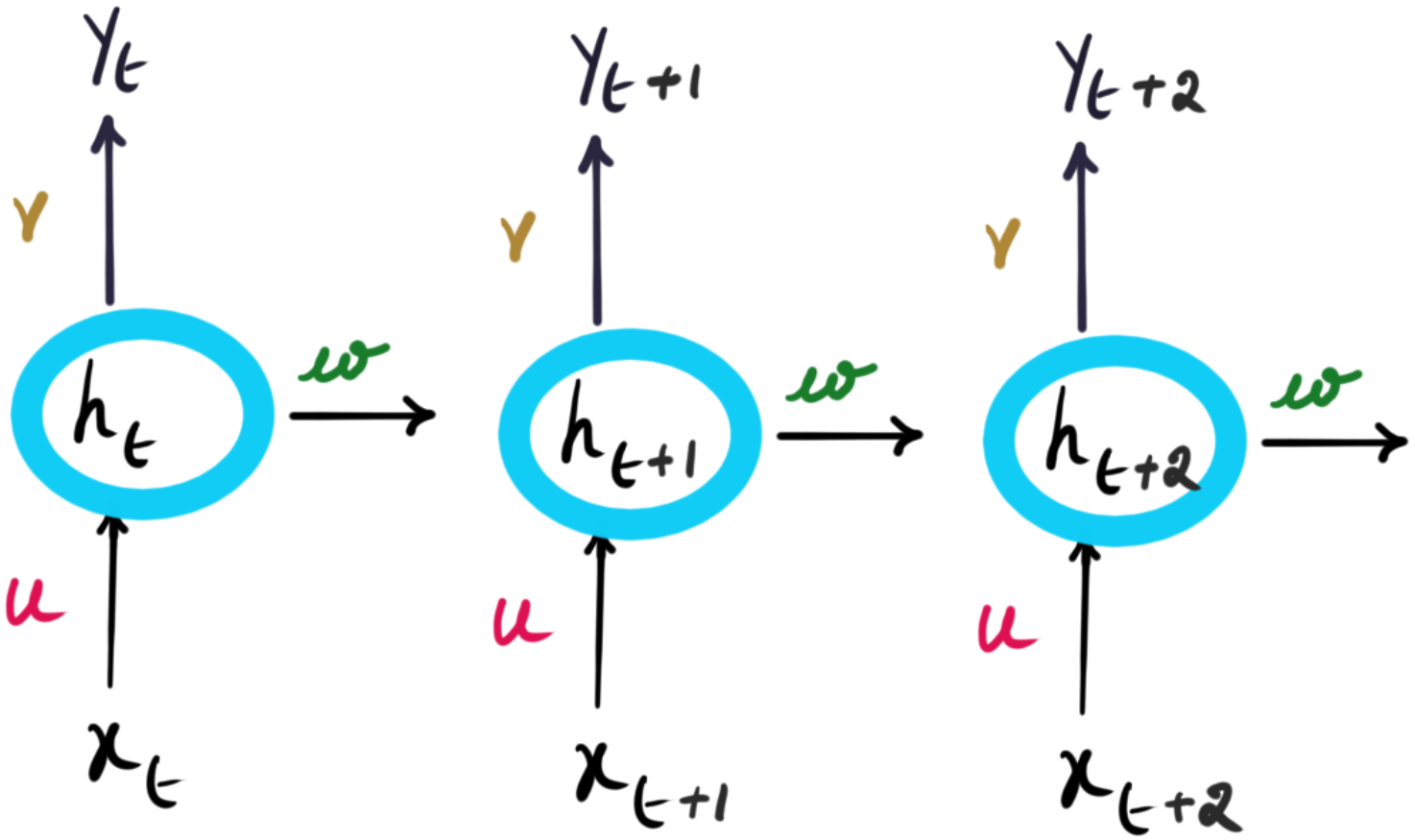
Recurrent Neural Networks:Unrolling



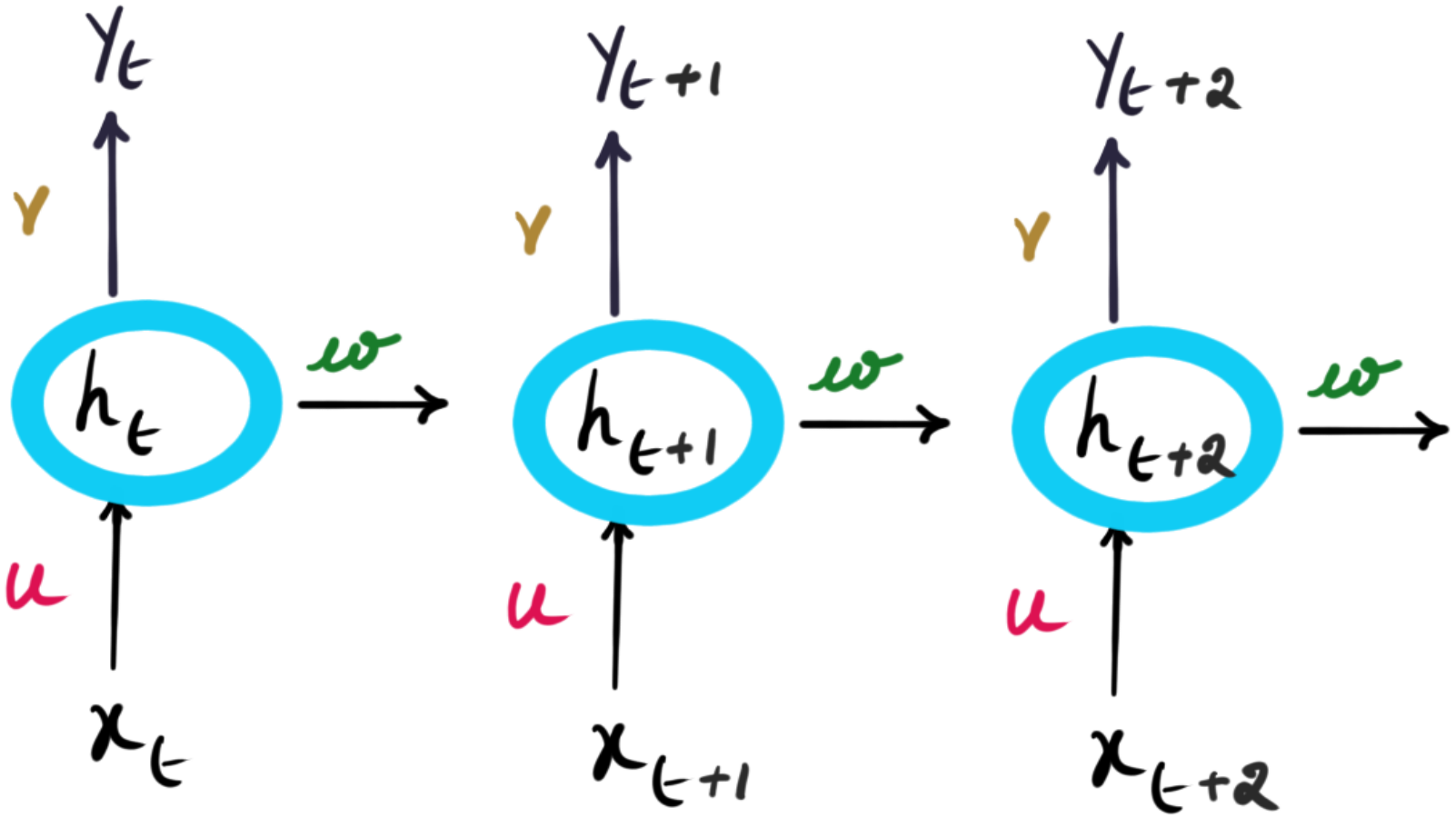
Recurrent Neural Networks:Unrolling



Recurrent Neural Networks:Unrolling



Recurrent Neural Networks:Unrolling



$$h_t = \tanh(u \cdot x_t + w \cdot h_{(t-1)} + b_h)$$

$$y_t = \text{softmax}(v \cdot h_t)$$

Recurrent Neural Networks:ex-1

despite a back problem, but will renew

• kept input small for slides

```
chars:: ['\n', ' ', ',', 'a', 'b', 'c', 'd', 'e', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'w']
data has 39 characters, 20 unique.
char_to_ix:::: {'n': 12, 'i': 8, ' ': 1, 'k': 9, 's': 16, 'd': 6, 'l': 10, 'u': 18, '\n': 0, 'c': 5, 'p': 14, 'o': 13, 'b': 4, 'm': 11, ',': 2, 'a'
: 3, 'w': 19, 't': 17, 'r': 15, 'e': 7}
```

- Dictionary encoding
- This would translate to a one hot vector of 20x1 for each character

Recurrent Neural Networks:ex-1

despite a back problem, but will renew

```
chars:: ['\n', ' ', ',', 'a', 'b', 'c', 'd', 'e', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'w']
data has 39 characters, 20 unique.
char_to_ix:::: {'n': 12, 'i': 8, ' ': 1, 'k': 9, 's': 16, 'd': 6, 'l': 10, 'u': 18, '\n': 0, 'c': 5, 'p': 14, 'o': 13, 'b': 4, 'm': 11, ',': 2, 'a'
: 3, 'w': 19, 't': 17, 'r': 15, 'e': 7}
```

```
self.numclasses: 20
self.statesize: 4
self.seqlength: 5
self.bh.shape: (4, 1)
self.by.shape: (20, 1)
self.batchsize: 1
self.u: [[ 1.72127090e-04   1.37617992e-02  -9.11468704e-03  -3.50850127e-04
    1.67293201e-03   9.51884481e-04   2.83417085e-03  -7.68983590e-04
    5.66927027e-03  -1.23929952e-02   1.49928305e-03  -1.33954186e-03
    1.30025983e-02   3.51889692e-03  -1.28509619e-02   8.04812411e-03
   -1.63978119e-02  -1.21513738e-02  -6.77414911e-03  -7.25272948e-03]
 [ -2.32067117e-03  -2.08464395e-02  -2.51028389e-03   1.82950913e-03
   -6.48352631e-03  -1.96409995e-03   2.57376092e-03   5.87414306e-03
    3.07873972e-03   6.90978829e-03   8.52703341e-03  -4.68452133e-03
   -1.37011289e-04   1.61590562e-02  -1.10038662e-04  -8.11029675e-03
    2.73509192e-03  -4.31468766e-03   2.04914683e-02   1.05254977e-02]
 [ -7.71438930e-03   2.91452465e-03   3.31056159e-03  -9.52325883e-03
    1.26101604e-02   1.30930305e-02  -2.45476971e-03   1.04263336e-02
   -8.52937360e-05   1.27863506e-02  -3.06180779e-03  -5.51043454e-03
   -5.69138726e-03  -2.27268824e-02  -4.49892813e-03   2.32849042e-02
    7.84556619e-03  -1.54032607e-02   4.85867424e-04  -4.59047828e-03]
 [ -7.29860188e-04  -7.80284374e-03   1.38810038e-02  -5.48688185e-03
    1.89054277e-03  -6.20354671e-03  -7.00974601e-03   2.07905417e-04
   -1.46993567e-02   1.19389009e-02   1.97793494e-03   4.99428835e-03
   -3.00743912e-03   9.52572084e-03  -2.03557507e-03  -1.41755894e-02
    1.23179318e-02  -8.33331404e-03  -7.86517600e-03   1.50329498e-02]]
self.w: [[-0.01547087 -0.0010283   0.01520684  0.0012339 ]
 [-0.00439563  0.00109088  0.00113987  0.00164847]
 [-0.00277533 -0.00305131 -0.00034081 -0.00360214]
 [ 0.00885719 -0.01488522  0.00490174 -0.01487926]]
```

- num_classes based on the vocabulary
- also dictates the one of the dimensions of u
- specifies the other dimension of u and both dimensions of w.
- GEMM u (4x20) with input (20x1) to produce state of (4x1)

Recurrent Neural Networks:ex-1

```
self.w: [[-0.01547087 -0.0010283   0.01520684  0.0012339 ]
 [-0.00439563  0.00109088  0.00113987  0.00164847]
 [-0.00277533 -0.00305131 -0.00034081 -0.00360214]
 [-0.00885719 -0.01488522  0.00490174 -0.01487926]]
```

- GEMM w(4x4) with state(4x1) to result in the next state (4x1)

```
self.v: [[ -2.03163763e-02   2.55755466e-04   6.47609261e-03   6.07944377e-03]
 [  1.10393313e-02   1.62610236e-02  -6.57592189e-03   2.49719271e-03]
 [  1.57886317e-02  -2.24706852e-02  -3.17439214e-04   6.92722780e-03]
 [ -3.01681957e-03   1.05905908e-02   1.99962457e-02  -1.80763490e-02]
 [ -6.59863229e-04   5.65464473e-03  -1.06666542e-02  -1.50652451e-02]
 [ -5.86137047e-04  -2.08463859e-03  -2.56090909e-03   2.41701005e-02]
 [  9.88649167e-03   9.59875898e-03  -6.17798171e-03  -7.96813586e-03]
 [ -6.13589948e-03  -6.31522996e-03   1.29593639e-03  -1.13061986e-02]
 [ -1.56681555e-03  -2.91297033e-03  -9.97783801e-03   5.13193639e-03]
 [  5.23592372e-04   1.93093572e-02  -3.32537515e-03  -1.15520276e-02]
 [ -8.71133169e-03  -7.03281879e-03  -3.94898752e-03  -1.01024345e-02]
 [  4.68472115e-03  -1.99325538e-03  -4.12092604e-03   4.14803486e-03]
 [  2.74034938e-03   3.31222461e-03  -7.75490077e-03  -2.47301496e-03]
 [ -1.21246721e-02  -3.71187289e-03   6.07880798e-03   1.45480495e-02]
 [ -6.84792965e-03   1.19227917e-02   1.28863555e-02   1.77994537e-02]
 [  5.73779127e-03  -8.54286620e-03   1.39740416e-02   4.78271378e-04]
 [ -4.33128403e-03   1.46962911e-04  -1.50669070e-02  -1.41750922e-02]
 [ -8.71470136e-03   5.43422754e-05  -7.96976695e-03   5.25676785e-03]
 [  2.82614970e-03   3.83598642e-03   1.53446453e-02  -1.11021230e-02]
 [  4.87455478e-03   4.30629670e-03   3.48199859e-03  -2.00482505e-02]]
self.state: [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

- GEMM v(20x4) with new state (4x1) will result in a shape of input (20x1) to compare it against to compute the loss.
- This comparison is with the input next time step advanced.

Recurrent Neural Networks:ex-1

```
rnn = RNNCell(vocab_size,hidden_size,seq_length,learning_rate,u=u,w=w,v=v);
```

For the example on the slide:
vocabsize = 20
hiddensize = 4.
seq_length = 5

```
class RNNCell:
    def __init__(self, num_classes, state_size, seq_length,learning_rate,
                 batch_size=1,u=None, w=None, v=None, verbose=True):
        self.numclasses = num_classes
        self.statesize = state_size
        self.seqlength = seq_length
        self.batchsize=batch_size
        self.bh = np.zeros((self.statesize, 1))
        self.by = np.zeros((self.numclasses, 1))
        if u is None:
            self.u = np.random.normal(0.0, pow((self.statesize) * self.numclasses, -0.5) ,
                                      (self.statesize, self.numclasses))
        else:
            self.u = u;

        if w is None:
            self.w = np.random.normal(0.0, pow((self.statesize) * self.statesize, -0.5) ,
                                      (self.statesize, self.statesize))
        else:
            self.w = w;

        if v is None:
            self.v = np.random.normal(0.0, pow((self.numclasses) * self.statesize, -0.5) ,
                                      (self.numclasses, self.statesize))
        else:
            self.v = v;

        self.state = np.zeros((self.statesize, self.batchsize))
```

RNNCell

Recurrent Neural Networks:ex-1

despite a back problem, but will renew

- kept input small for slides

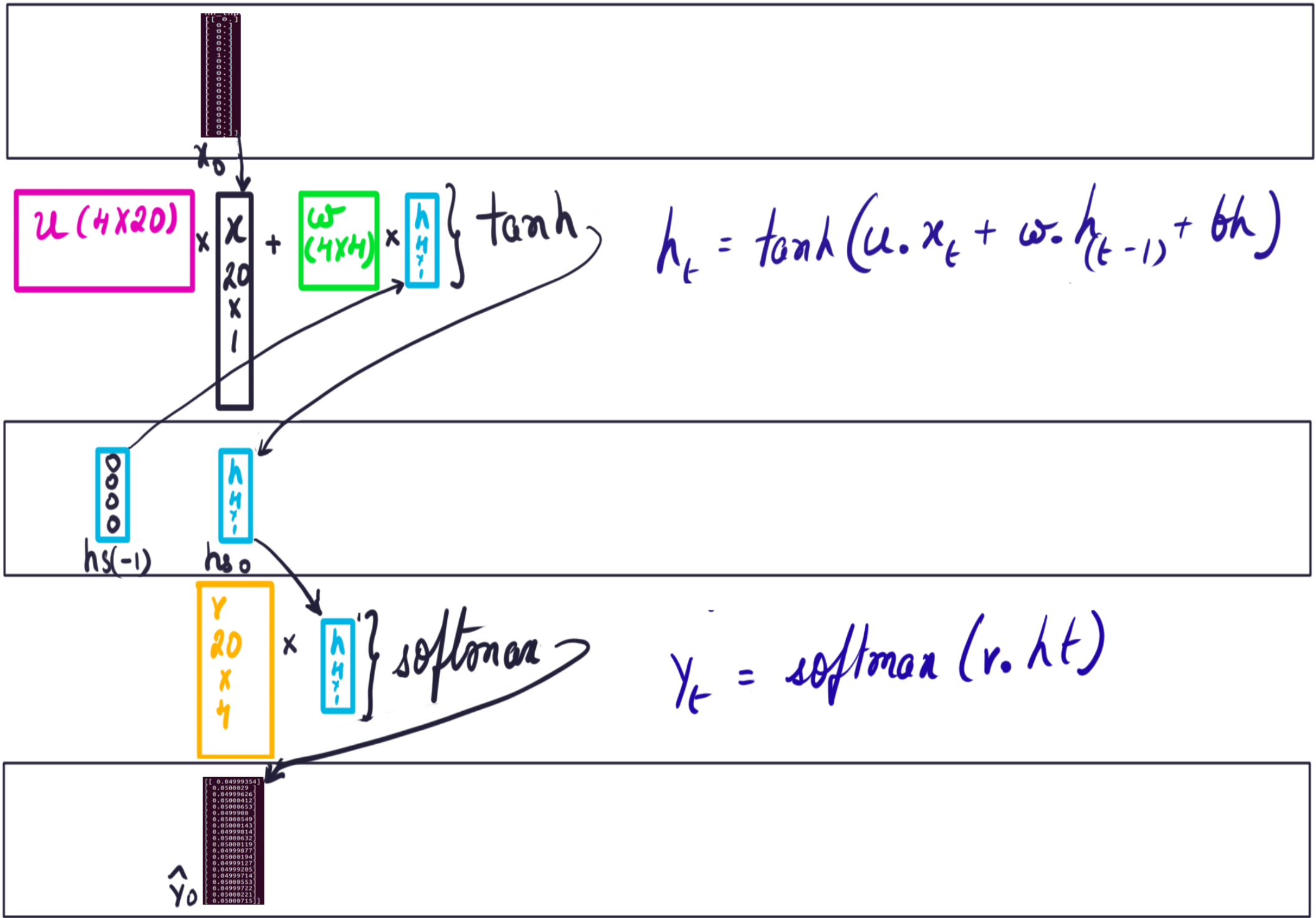
```
chars:: ['\n', ' ', ',', 'a', 'b', 'c', 'd', 'e', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'w']
data has 39 characters, 20 unique.
char_to_ix:::: {'n': 12, 'i': 8, ' ': 1, 'k': 9, 's': 16, 'd': 6, 'l': 10, 'u': 18, '\n': 0, 'c': 5, 'p': 14, 'o': 13, 'b': 4, 'm': 11, ',': 2, 'a'
: 3, 'w': 19, 't': 17, 'r': 15, 'e': 7}
```

- Dictionary encoding
- This would translate to a one hot vector of 20x1 for each character

```
inputs: [6, 7, 16, 14, 8]
targets: [7, 16, 14, 8, 17]
```

- For a sequence size of 5, the 5 inputs and their respective targets 1 time step advanced.

Recurrent Neural Networks:ex-1(t-0)



Recurrent Neural Networks:ex-1(t-0)

```
while True:
  # prepare inputs (we're sweeping from left to right in steps
  #print("len(data):",len(data),"seq_length:",seq_length)
  if p+seq_length+1 >= len(data) or n == 0:
    rnn.initState() # reset RNN memory

    #print ("hprev.shape:",hprev.shape)
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
  loss=rnn.train(np.matrix(inputs),np.matrix(targets))
```

```
inputs: [6, 7, 16, 14, 8]
targets: [7, 16, 14, 8, 17]
```

```
def train(self, rnn_inputs, labels):
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(self.state)
    loss = 0
    r, c = rnn_inputs.shape;
    r1, c1 = labels.shape;
    for t in range(c):
        rnn_input = rnn_inputs[:, t * 1:(t + 1) * 1]
        label = labels[:, t * 1:(t + 1) * 1]
        labelnhot=self.onehot(label)
        xs[t]=self.onehot(rnn_input)
        hs[t] = np.tanh(np.dot(self.u, xs[t]) + np.dot(self.w, hs[t - 1]) + self.bh)
        ys[t] = np.dot(self.v, hs[t]) + self.by
        ps[t]=self.softmax(ys[t])
        lossesperoneseq=self.loss(ps[t], labelnhot)
        self.totalloss += lossesperoneseq.sum(axis=1)
        #print("self.totalloss:", self.totalloss)
        loss += lossesperoneseq
        self.numloss += r
```

(1 x 5)

(1 x 5)

} if batch is not default of 1 then rnn would not 1 and batch size

Recurrent Neural Networks:ex-1(t-0)

```
def train(self, rnn_inputs, labels):

    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(self.state)
    loss = 0
    r, c = rnn_inputs.shape;
    r1, c1 = labels.shape;
    for t in range(c):
        rnn_input = rnn_inputs[:, t * 1:(t + 1) * 1]
        label = labels[:, t * 1:(t + 1) * 1]
        labelnhot=self.onehot(label)
        xs[t]=self.onehot(rnn_input)
        hs[t] = np.tanh(np.dot(self.u, xs[t]) + np.dot(self.w, hs[t - 1]) + self.bh)
        ys[t] = np.dot(self.v, hs[t]) + self.by
        ps[t]=self.softmax(ys[t])
        lossesperoneseq=self.loss(ps[t], labelnhot)
        self.totalloss += lossesperoneseq.sum(axis=1)
        #print("self.totalloss:", self.totalloss)
        loss += lossesperoneseq
        self.numloss += r
```

```
inputs: [6, 7, 16, 14, 8]
targets: [7, 16, 14, 8, 17]
```

```
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

Recurrent Neural Networks:ex-1(t-0)

```
def train(self, rnn_inputs, labels):
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(self.state)
    loss = 0
    r, c = rnn_inputs.shape;
    r1, c1 = labels.shape;
    for t in range(c):
        rnn_input = rnn_inputs[:, t * 1:(t + 1) * 1]
        label = labels[:, t * 1:(t + 1) * 1]
        labelnhot=self.onehot(label)
        xs[t]=self.onehot(rnn_input)
        hs[t] = np.tanh(np.dot(self.u, xs[t]) + np.dot(self.w, hs[t - 1]) + self.bh)
        ys[t] = np.dot(self.v, hs[t]) + self.by
        ps[t]=self.softmax(ys[t])
        lossesperoneseq=self.loss(ps[t], labelnhot)
        self.totalloss += lossesperoneseq.sum(axis=1)
        #print("self.totalloss:", self.totalloss)
        loss += lossesperoneseq
        self.numloss += r
```

after GEMM

```
ys[t]: [[ -1.15433618e-04]
 [  7.17771034e-05]
 [ -6.08645573e-05]
 [  9.63298896e-05]
 [  1.44469451e-04]
 [ -1.70163619e-04]
 [  1.23743973e-04]
 [  4.24270664e-05]
 [ -2.34176198e-05]
 [  1.40319972e-04]
 [  3.77180851e-05]
 [ -1.08131774e-05]
 [  5.26628331e-05]
 [ -1.60815258e-04]
 [ -1.45122381e-04]
 [ -4.33809005e-05]
 [  1.24450562e-04]
 [ -4.18431234e-05]
 [  5.80369527e-05]
 [  1.56881992e-04]]
```

```
def softmax(self, logits):
    r, c = logits.shape
    predsl = []
    for row in logits.T:
        inputs = np.squeeze(np.asarray(row))
        predsl.append(np.exp(inputs) / float(sum(np.exp(inputs))))
    return np.matrix(predsl).T
```

```
preds:
 [[ 0.04999354]
 [ 0.0500029 ]
 [ 0.04999626]
 [ 0.05000412]
 [ 0.05000653]
 [ 0.0499908 ]
 [ 0.05000549]
 [ 0.05000143]
 [ 0.04999814]
 [ 0.05000632]
 [ 0.05000119]
 [ 0.04999877]
 [ 0.05000194]
 [ 0.04999127]
 [ 0.04999205]
 [ 0.04999714]
 [ 0.05000553]
 [ 0.04999722]
 [ 0.05000221]
 [ 0.05000715]]
```

sums upto 1.

Recurrent Neural Networks:ex-1(t-0)

```python
def train(self, rnn_inputs, labels):

    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(self.state)
    loss = 0
    r, c = rnn_inputs.shape;
    r1, c1 = labels.shape;
    for t in range(c):
        rnn_input = rnn_inputs[:, t * 1:(t + 1) * 1]
        label = labels[:, t * 1:(t + 1) * 1]
        labelnhot=self.onehot(label)
        xs[t]=self.onehot(rnn_input)
        hs[t] = np.tanh(np.dot(self.u, xs[t]) + np.dot(self.w, hs[t - 1]) + self.bh)
        ys[t] = np.dot(self.v, hs[t]) + self.by
        ps[t]=self.softmax(ys[t])
        lossesperoneseq=self.loss(ps[t], labelnhot)
        self.totalloss += lossesperoneseq.sum(axis=1)
        #print("self.totalloss:", self.totalloss)
        loss += lossesperoneseq
        self.numloss += r
```

```
preds:
 [[ 0.04999354]
 [ 0.0500029 ]
 [ 0.04999626]
 [ 0.05000412]
 [ 0.05000653]
 [ 0.0499908 ]
 [ 0.05000549]
 [ 0.05000143]
 [ 0.04999814]
 [ 0.05000632]
 [ 0.05000119]
 [ 0.04999877]
 [ 0.05000194]
 [ 0.04999127]
 [ 0.04999205]
 [ 0.04999714]
 [ 0.05000553]
 [ 0.04999722]
 [ 0.05000221]
 [ 0.05000715]]
```

```
[[ 2.99586156]
 [ 2.99567435]
 [ 2.99580699]
 [ 2.9956498 ]
 [ 2.99560166]
 [ 2.99591629]
 [ 2.99562238]
 [ 2.9957037 ]
 [ 2.99576954]
 [ 2.99560581]
 [ 2.99570841]
 [ 2.99575694]
 [ 2.99569346]
 [ 2.99590694]
 [ 2.99589125]
 [ 2.99578951]
 [ 2.99562168]
 [ 2.99578797]
 [ 2.99568809]
 [ 2.99558925]]
```

×

```
labels:
 [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

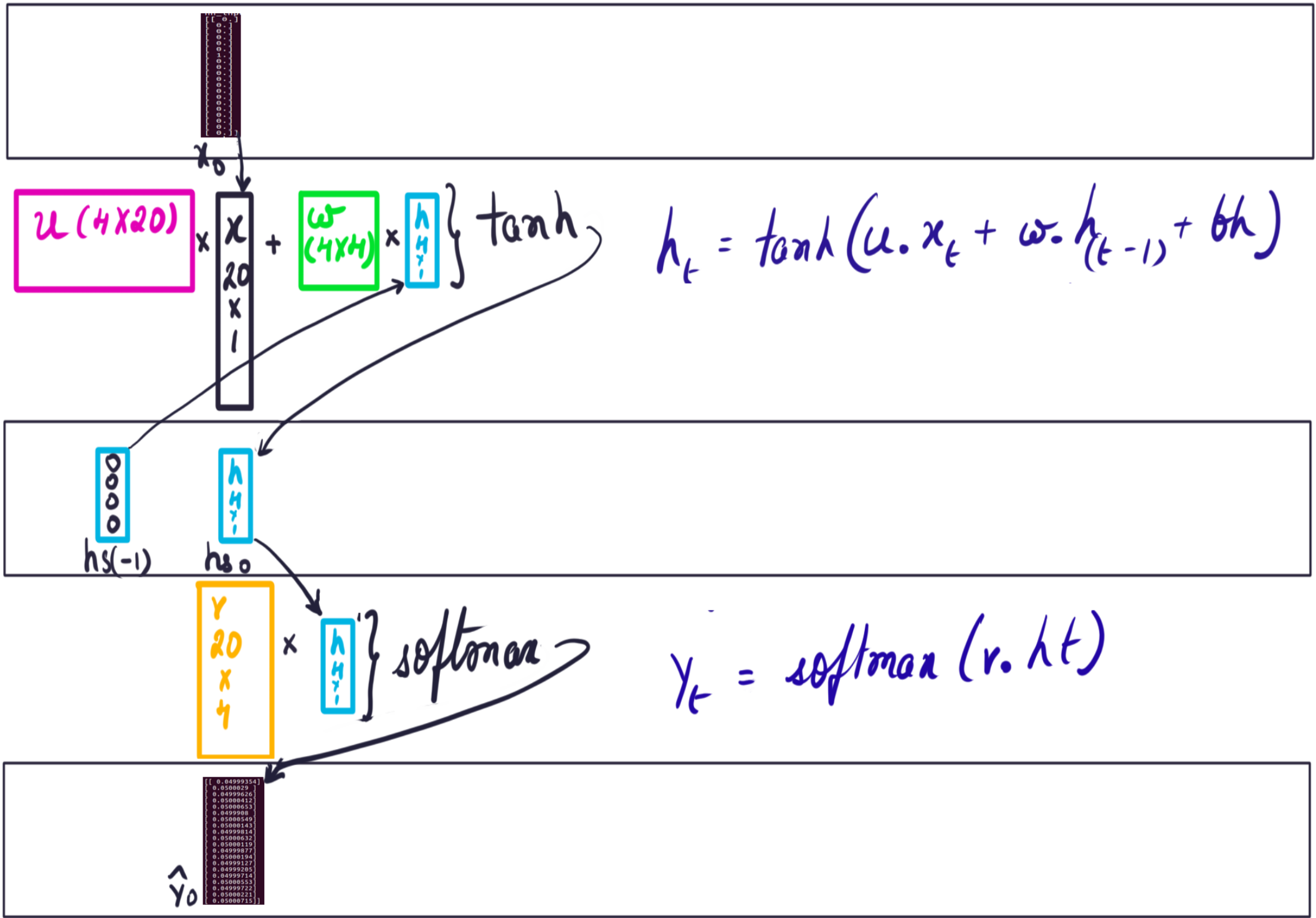
natural log

```
def loss(self, softmaxlogits, labels):
    return -np.multiply(labels, np.log(softmaxlogits)).sum(axis=0)
```

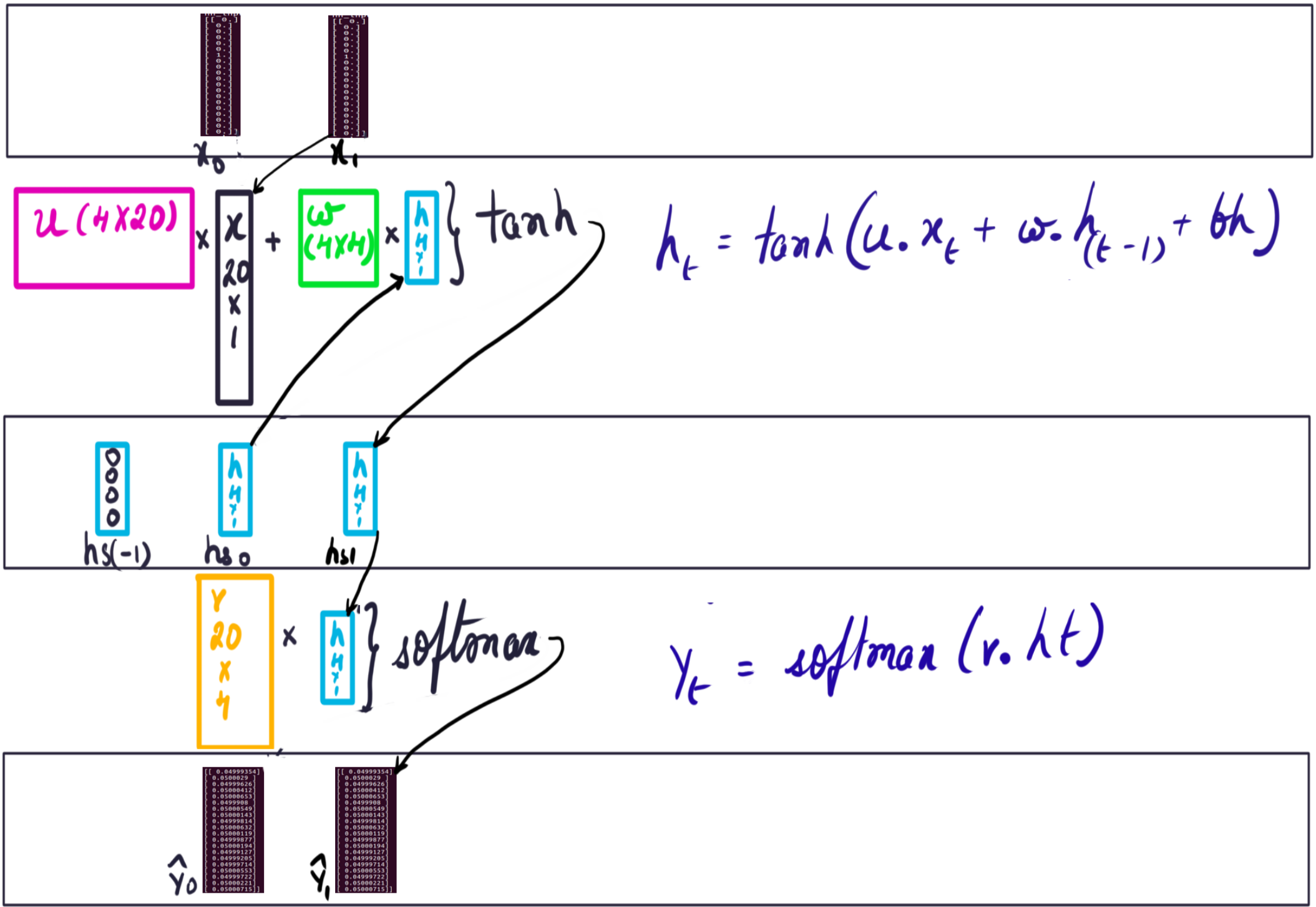
```
[[ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 2.9957037]
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.
 [ 0.]]
```

- similar to logistic regression cost function (This is called the cross entropy loss)

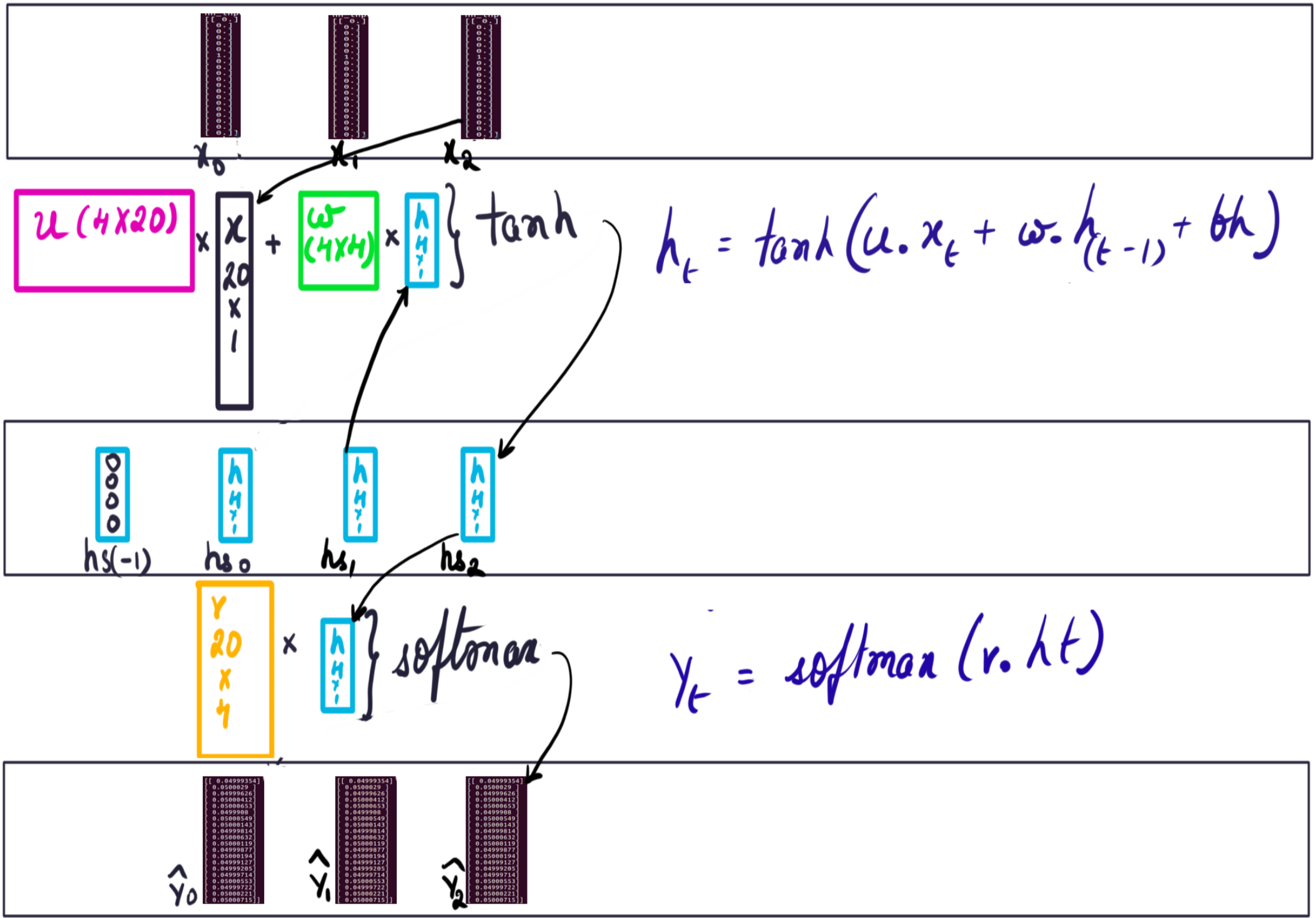
Recurrent Neural Networks:ex-1(t-0)



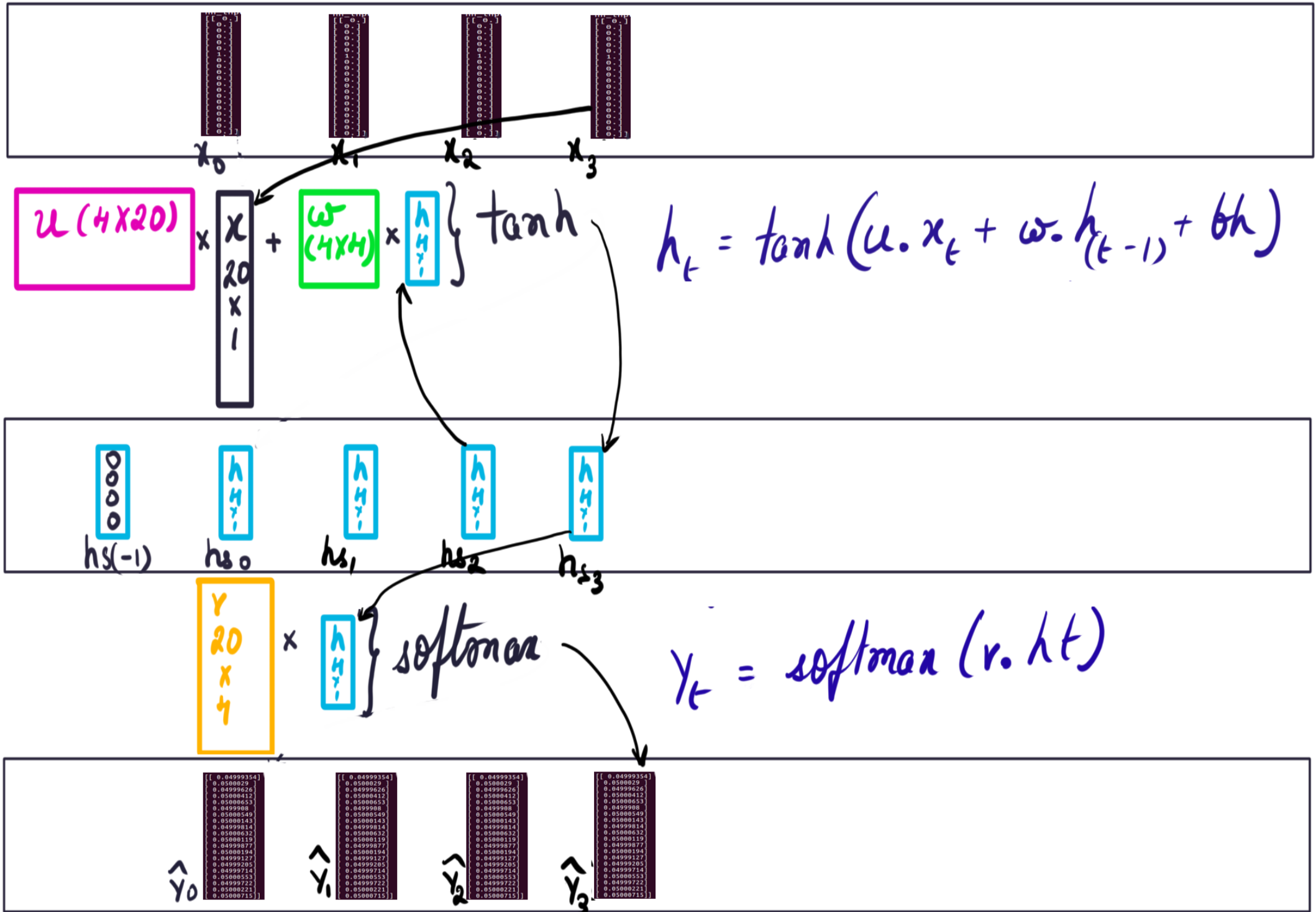
Recurrent Neural Networks:ex-1(t-1)



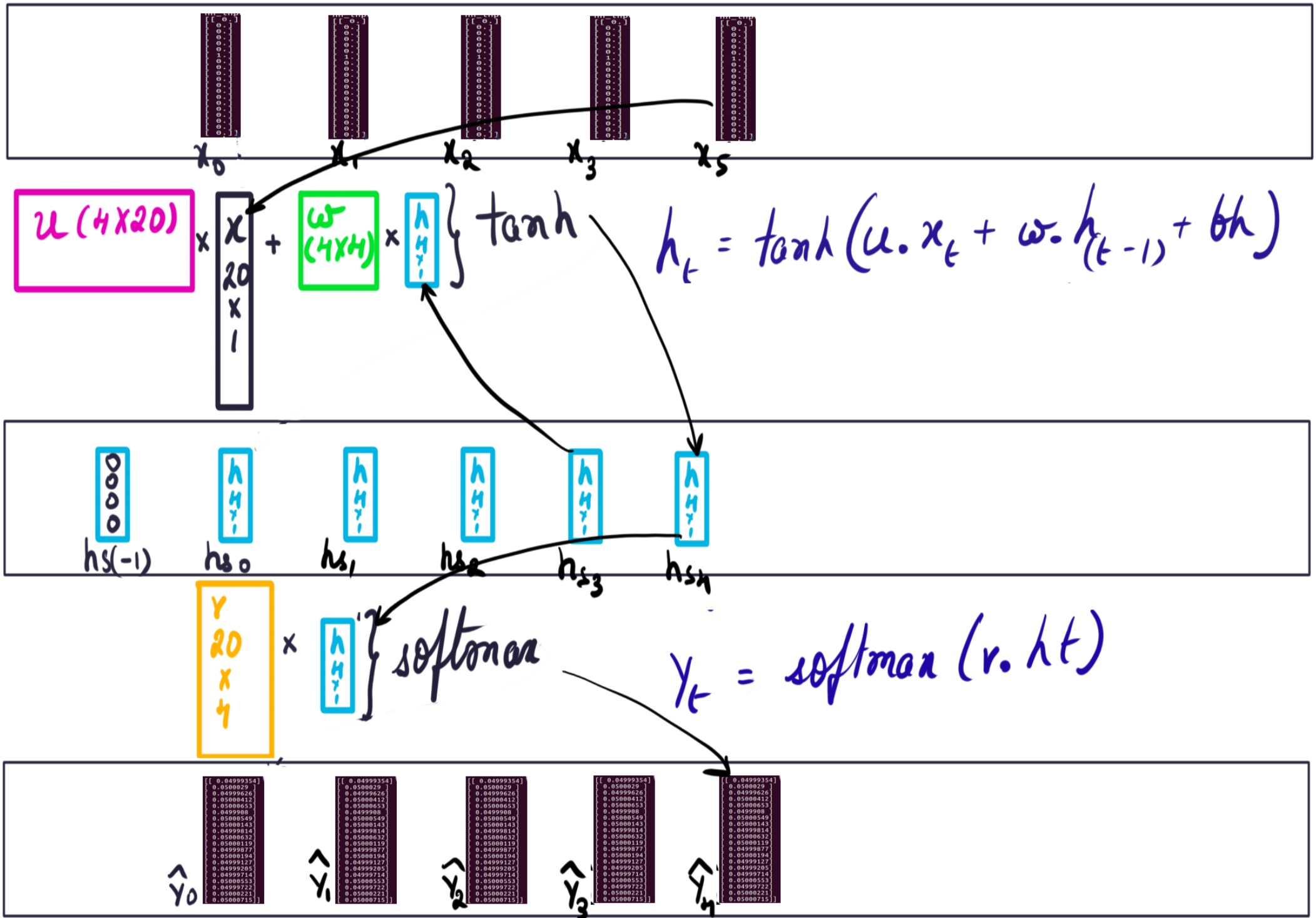
Recurrent Neural Networks:ex-1(t-2)



Recurrent Neural Networks:ex-1(t-4)



Recurrent Neural Networks:ex-1(t-5)



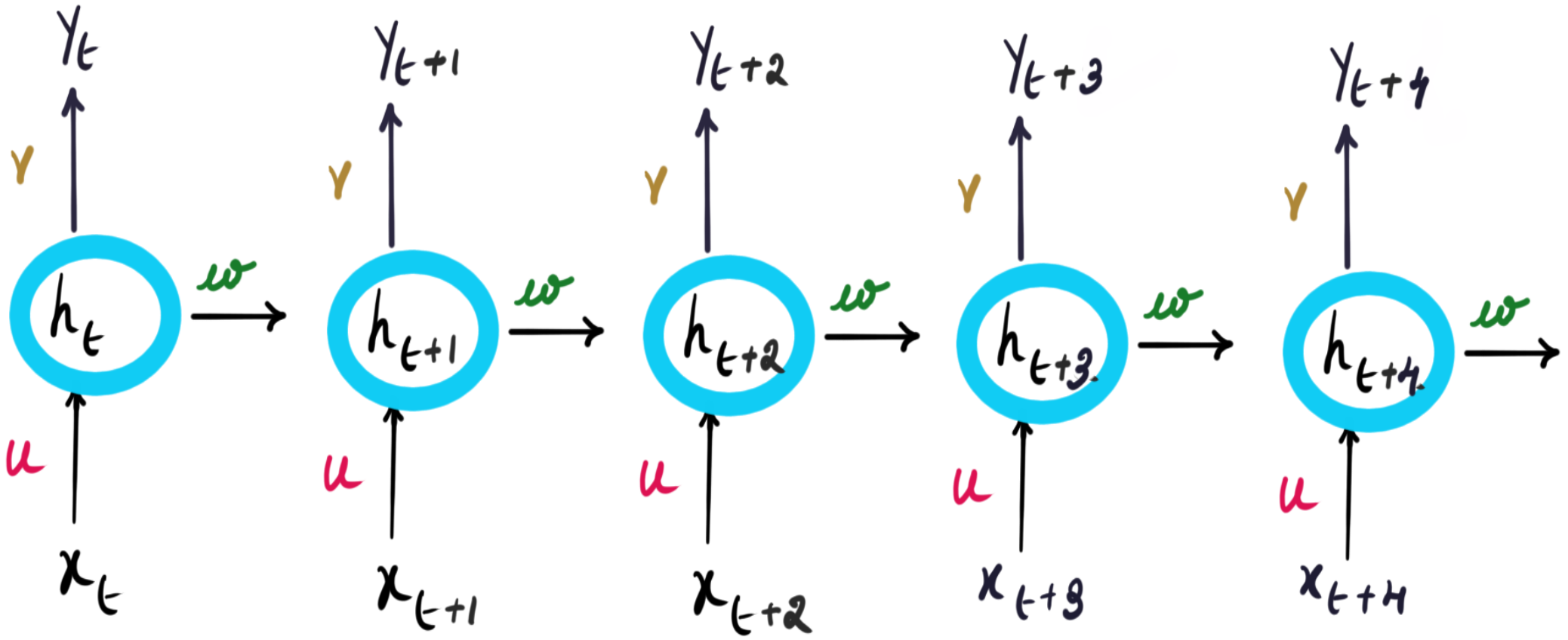
Recurrent Neural Networks:ex-1(t-0)

```
def train(self, rnn_inputs, labels):

    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(self.state)
    loss = 0
    r, c = rnn_inputs.shape;
    r1, c1 = labels.shape;
    for t in range(c):
        rnn_input = rnn_inputs[:, t * 1:(t + 1) * 1]
        label = labels[:, t * 1:(t + 1) * 1]
        labelnhot=self.onehot(label)
        xs[t]=self.onehot(rnn_input)
        hs[t] = np.tanh(np.dot(self.u, xs[t]) + np.dot(self.w, hs[t - 1]) + self.bh)
        ys[t] = np.dot(self.v, hs[t]) + self.by
        ps[t]=self.softmax(ys[t])
        lossesperoneseq=self.loss(ps[t], labelnhot)
        self.totalloss += lossesperoneseq.sum(axis=1)
        #print("self.totalloss:", self.totalloss)
        loss += lossesperoneseq
        self.numloss += r
```

- Forward pass through num_seq of inputs (5 in this example)
- Now calculate errors and back propagate (BPTT)
  - Once back propagated through num_seq, readjust the weights to minimize the error.

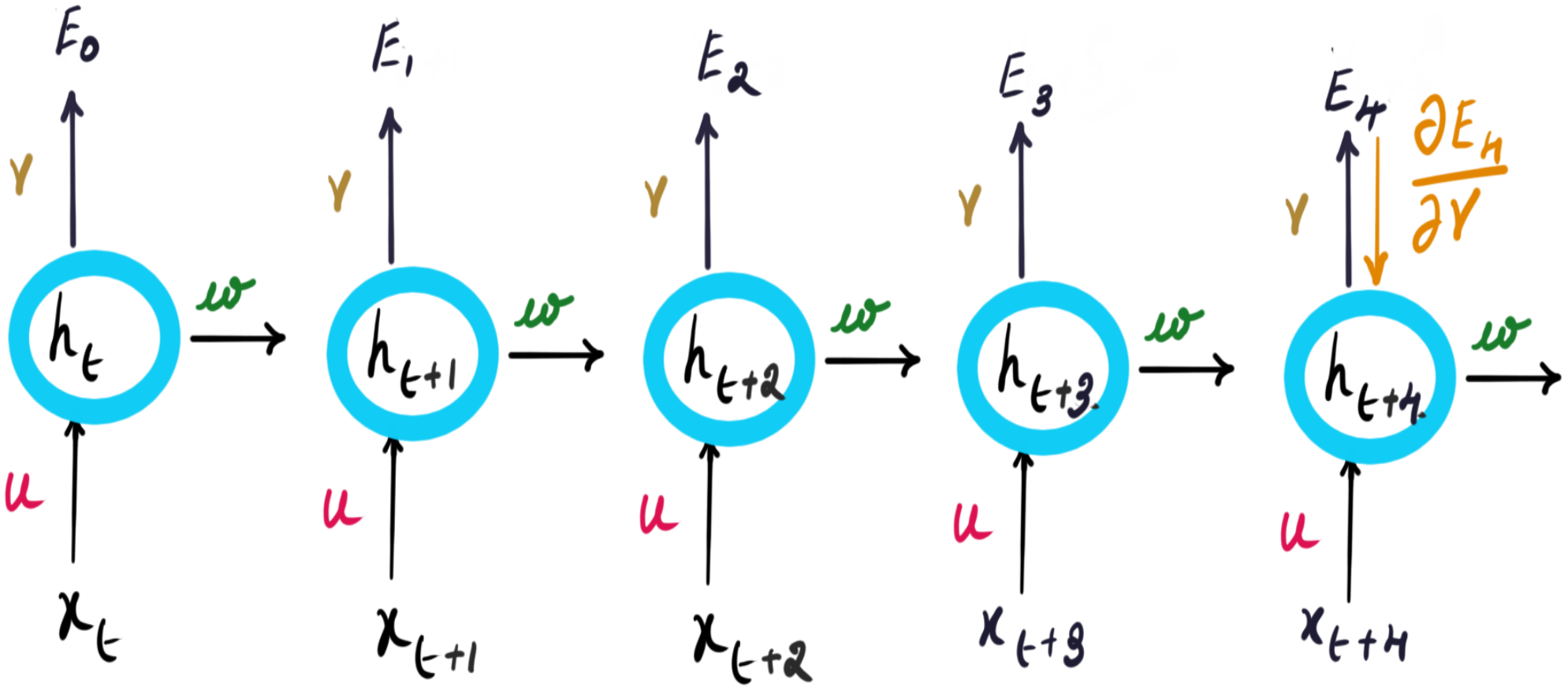
Recurrent Neural Networks:ex-1:BPTT



$$h_t = \tanh(u \cdot x_t + w \cdot h_{(t-1)} + bh)$$

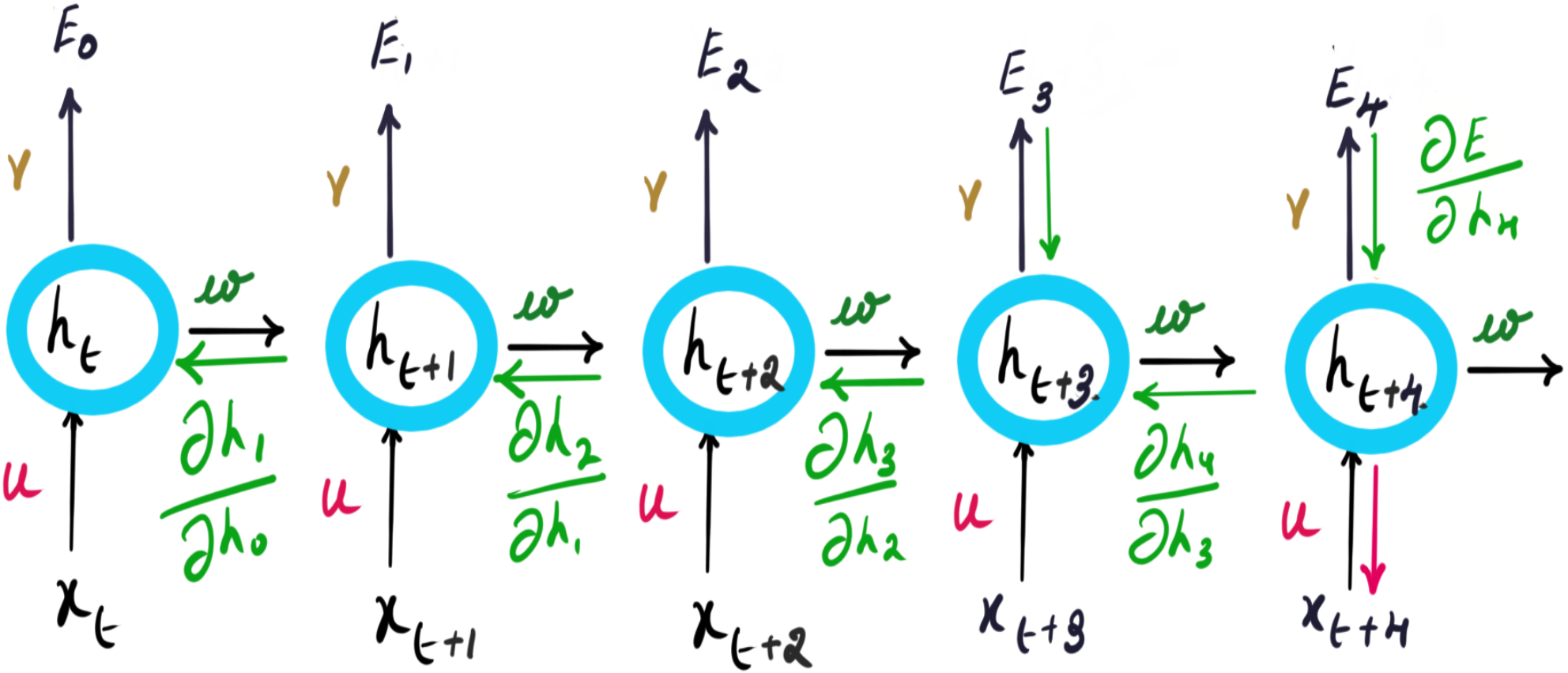
$$y_t = \text{softmax}(v \cdot h_t)$$

Recurrent Neural Networks:ex-1:BPTT:Error



. Derivatives have to be calculated w.r.t all the weights and biases.

Recurrent Neural Networks:ex-1:BPTT:Derivatives



- Derivatives have to be calculated w.r.t all the weights and biases.
- Derivatives to calculated

$$\frac{\partial E}{\partial U}, \frac{\partial E}{\partial W}, \frac{\partial E}{\partial V}, \frac{\partial E}{\partial b_h}, \frac{\partial E}{\partial b_y}$$

Recurrent Neural Networks:ex-1:BPTT:Derivatives

Softmax Derivative

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}}$$

- It is always positive because of e.
- Since numerator appears in denominator summed up with some other positive numbers $S_j < 1$.
- Therefore its range is (0,1).

```
ys[t]: [[ -1.15433618e-04]
 [  7.17771034e-05]
 [ -6.08645573e-05]
 [  9.63298896e-05]
 [  1.44469451e-04]
 [ -1.70163619e-04]
 [  1.23743973e-04]
 [  4.24270664e-05]
 [ -2.34176198e-05]
 [  1.40319972e-04]
 [  3.77180851e-05]
 [ -1.08131774e-05]
 [  5.26628331e-05]
 [ -1.60815258e-04]
 [ -1.45122381e-04]
 [ -4.33809005e-05]
 [  1.24450562e-04]
 [ -4.18431234e-05]
 [  5.80369527e-05]
 [  1.56881992e-04]]
```

```
preds:
 [[ 0.04999354]
 [ 0.0500029 ]
 [ 0.04999626]
 [ 0.05000412]
 [ 0.05000653]
 [ 0.0499908 ]
 [ 0.05000549]
 [ 0.05000143]
 [ 0.04999814]
 [ 0.05000632]
 [ 0.05000119]
 [ 0.04999877]
 [ 0.05000194]
 [ 0.04999127]
 [ 0.04999205]
 [ 0.04999714]
 [ 0.05000553]
 [ 0.04999722]
 [ 0.05000221]
 [ 0.05000715]]
```

sums up to 1.

Recurrent Neural Networks:ex-1:BPTT:Derivatives

Softmax Derivative

- It is fundamentally a vector function
  - So there is no derivative of softmax
  - Instead
    - Which component (output element) of softmax needs to be specified for derivative
    - Since softmax has multiple input, the derivative need to be calculated w.r.t which input element.
  - $\frac{\partial S_i}{\partial a_j}$ (i^{th} partial derivative of i^{th} output w.r.t j^{th} input)
  - We say we compute its Jacobian Matrix.

Recurrent Neural Networks:ex-1:BPTT:Derivatives

Softmax Derivative

$$\frac{\partial S_i}{\partial a_j} = \frac{\partial \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j}$$

Quotient Rule

$$f(x) = \frac{g(x)}{h(x)}$$

$$f'(x) = \frac{g'(x)\,h(x) - h'(x)\,g(x)}{h(x)^2}$$

- For h_i, no matter for which a_j the derivative is computed, the answer will always be e^{a_j}.
- For $g(i)$, the derivative w.r.t a_j is $\boxed{e^{a_j}}$ only if $i=j$, because only then $g(i)$ has a_j any where in it. Otherwise the derivative is $\boxed{0}$.

- in our case

$$g_i = e^{a_i}$$

$$h_i = \sum_{k=1}^{N} e^{a_k}$$

Recurrent Neural Networks:ex-1:BPTT:Derivatives

- When $i=j$ Softmax derivative

$$\frac{\partial \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{e^{a_i}.\Sigma - e^{a_j}.e^{a_i}}{\Sigma^2}$$

$$= \frac{e^{a_i}}{\Sigma}.\frac{\Sigma - e^{a_j}}{\Sigma}$$

$$= S_i(1-S_j)$$

- When $i \neq j$

$$\frac{\partial \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{0 - e^{a_j}e^{a_i}}{\Sigma^2} = -\frac{e^{a_j}}{\Sigma}.\frac{e^{a_j}}{\Sigma}$$

$$= -S_j S_i$$

$$\begin{cases} S_i(1-S_j) & i=j \\ -S_j.S_i & i \neq j \end{cases}$$

eq-1

Recurrent Neural Networks:ex-1:BPTT:Derivatives

$$\frac{\partial E}{\partial V} = \frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} \cdot \frac{\partial q_t}{\partial V}$$

$$q_t = V \cdot h_{s_t}$$

- $E = -y_t \cdot \log \hat{y}_t$ (cross entropy loss)
- $\frac{\partial E}{\partial \hat{y}_t} = -\frac{y_t}{\hat{y}_t}$ — eq-2
- $\hat{y}$ is the softmax function

Recurrent Neural Networks:ex-1:BPTT:Derivatives

$$\frac{\partial E}{\partial r} = \frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} \cdot \frac{\partial q_t}{\partial r}$$

Combining 2 & 1 $\quad q_t = r \cdot hs_t$

$$\frac{\partial E}{\partial q_t} = \frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} = -\frac{y_t}{\hat{y}_t} \cdot \hat{y}_t (1 - \hat{y}_t) + \sum_{i \neq j} \left(-\frac{y_t}{\hat{y}_t}\right)(-\hat{y}_t \hat{y}_t)$$

$$= -y_t + y_t \hat{y}_t + \sum_{i \neq j} y_t \hat{y}_t$$

$$= -y_t + \hat{y}_t \cdot \sum y_t$$

$$= \hat{y}_t - y_t \quad \left(\sum y_t = 1 \text{ because } y \text{ is one hot vector}\right)$$

eq-3

Recurrent Neural Networks:ex-1:BPTT:Derivatives

$$\frac{\partial E}{\partial V} = \frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} \cdot \frac{\partial q_t}{\partial V}$$

$$q_t = V \cdot hs_t$$

$$\frac{\partial q_t}{\partial V} = \frac{\partial V \cdot hs_t}{\partial V} = hs_t$$

eq-4

$$\frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} \cdot \frac{\partial q_t}{\partial V} = \hat{y} - y \cdot hs_t$$

$$\frac{\partial E}{\partial V} = \hat{y}_t - y_t \cdot hs_t$$

eq-5

Recurrent Neural Networks:ex-1:BPTT:Derivatives

$$\frac{\partial E}{\partial u} = \frac{\partial E}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_{vt}} \cdot \frac{\partial q_{vt}}{\partial hs_t} \cdot \frac{\partial hs_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial u}$$

where $q_{vt} = v \cdot hs_t$

$z_t = u \cdot x_t + w \cdot hs_{t-1}$

$hs_t = \tanh(z_t)$

$$\frac{\partial q_{vt}}{\partial hs_t} = \frac{\partial\, v \cdot hs_t}{\partial hs_t} = v \qquad \text{eq-6}$$

$$\frac{\partial z_t}{\partial u} = \frac{\partial (u \cdot x_t + w \cdot hs_{t-1})}{\partial u} = x_t \qquad \text{eq-8}$$

$$\frac{\partial hs_t}{\partial z_t} = (1 - hs_t^2) \qquad \text{eq-7}$$

$$\frac{\partial E}{\partial u} = \underbrace{(\hat{y} - y)}_{\text{eq-3}} \cdot \underbrace{v}_{\text{eq-6}} \cdot \underbrace{(1 - hs_t^2)}_{\text{eq-7}} \cdot \underbrace{x_t}_{\text{eq-8}} \qquad \text{eq-9}$$

Recurrent Neural Networks:ex-1:BPTT:Derivatives

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} \cdot \frac{\partial q_t}{\partial hs_t} \cdot \frac{\partial hs_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial w}$$

where $q_t = v \cdot hs_t$

$z_t = u \cdot x_t + w \cdot hs_{t-1}$

$hs_t = \tanh(z_t)$

$$\frac{\partial z}{\partial w} = \frac{\partial (u \cdot x + w \cdot hs_{t-1})}{\partial u} = hs_{t-1}$$

eq-10

$$\frac{\partial E}{\partial w} = \underbrace{(\hat{y} - y)}_{\text{eq-3}} \cdot \underbrace{v}_{\text{eq-6}} \cdot \underbrace{(1 - hs_t^2)}_{\text{eq-7}} \cdot \underbrace{hs_{t-1}}_{\text{eq-10}}$$

eq-11

Recurrent Neural Networks:ex-1:BPTT:Derivatives

$$\frac{\partial E}{\partial hs_{t-1}} = \frac{\partial E}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial q_t} \cdot \frac{\partial q_t}{\partial hs_t} \cdot \frac{\partial hs_t}{\partial hs_{t-1}}$$

where $q_t = v \cdot hs_t$

$z_t = u \cdot x_t + w \cdot hs_{t-1}$

$hs_t = \tanh(z_t)$

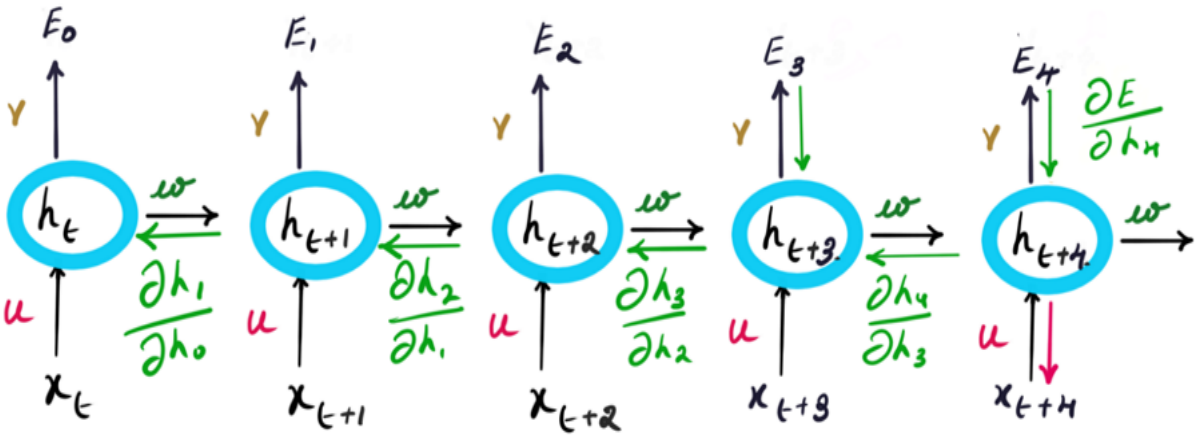
$$\frac{\partial z}{\partial hs_{(t-1)}} = \frac{\partial (u \cdot x + w \cdot hs_{t-1})}{\partial u} = w$$

eq-12

$$\frac{\partial E}{\partial hs_{t-1}} = \underbrace{(\hat{y} - y)}_{\text{eq-3}} \cdot \underbrace{v}_{\text{eq-6}} \cdot \underbrace{(1 - hs_t^2)}_{\text{eq-7}} \cdot \underbrace{w}_{\text{eq-12}}$$

eq-13

Recurrent Neural Networks:ex-1:BPTT:Derivatives



$$h_t = \tanh(u \cdot x_t + w \cdot h_{(t-1)} + bh)$$

$$y_t = \text{softmax}(v \cdot h_t)$$

$$\frac{\partial E}{\partial v} = \hat{y}_t - y_t \cdot hs_t \qquad \text{eq-5}$$

$$\frac{\partial E}{\partial u} = \underbrace{(\hat{y} - y)}_{\text{eq-3}} \cdot \underbrace{v}_{\text{eq-6}} \cdot \underbrace{(1 - hs_t^2)}_{\text{eq-7}} \cdot \underbrace{x_t}_{\text{eq-8}} \qquad \text{eq-9}$$

$$\frac{\partial E}{\partial w} = \underbrace{(\hat{y} - y)}_{\text{eq-3}} \cdot \underbrace{v}_{\text{eq-6}} \cdot \underbrace{(1 - hs_t^2)}_{\text{eq-7}} \cdot \underbrace{hs_{t-1}}_{\text{eq-10}} \qquad \text{eq-11}$$

$$\frac{\partial E}{\partial hs_{t-1}} = \underbrace{(\hat{y} - y)}_{\text{eq-3}} \cdot \underbrace{v}_{\text{eq-6}} \cdot \underbrace{(1 - hs_t^2)}_{\text{eq-7}} \cdot \underbrace{w}_{\text{eq-12}} \qquad \text{eq-13}$$

Recurrent Neural Networks:ex-1:BPTT:Derivatives

```
du, dw, dv = np.zeros_like(self.u), np.zeros_like(self.w), np.zeros_like(self.v)
dbh, dby = np.zeros_like(self.bh), np.zeros_like(self.by)
dhnext = np.zeros_like(hs[0])

for t in reversed(range(c)):
    label = labels[:, t * 1:(t + 1) * 1]
    dy = np.copy(ps[t])
    labelnhot=self.onehot(label)
    dy=dy-(labelnhot)
    dv += np.dot(dy, hs[t].T)
    dby += np.sum((dy),1).reshape(self.numclasses,1)
    dh = np.dot(self.v.T, dy) + dhnext  # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh  # backprop through tanh nonlinearity
    dbh += np.sum((dhraw),1).reshape(self.statesize,1)
    du += np.dot(dhraw, xs[t].T)
    dw += np.dot(dhraw, hs[t - 1].T)
    dhnext = np.dot(self.w.T, dhraw)

for dparam in [dw, du, dv, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam)  # clip to mitigate exploding gradients
self.state=hs[c - 1]

for param, dparam, mem in zip([self.w, self.u, self.v, self.bh, self.by], [dw, du, dv, dbh, dby],
                              [self.mw, self.mu,self.mv, self.mbh, self.mby]):
    mem += dparam * dparam
    param += -self.learningrate * dparam / np.sqrt(mem + 1e-8)  # adagrad update
```

- calculate in reverse order.

$$\frac{\partial E}{\partial v} = (\hat{y}_t - y_t) \cdot hs_t \quad \text{eq-5}$$

$$\frac{\partial E}{\partial u} = (\hat{y} - y) \cdot v \cdot (1 - hs_t^2) \cdot x_t \quad \text{eq-9}$$

(eq-3, eq-6, eq-7, eq-8)

$$\frac{\partial E}{\partial w} = (\hat{y} - y) \cdot v \cdot (1 - hs_t^2) \cdot hs_{t-1} \quad \text{eq-11}$$

(eq-3, eq-6, eq-7, eq-10)

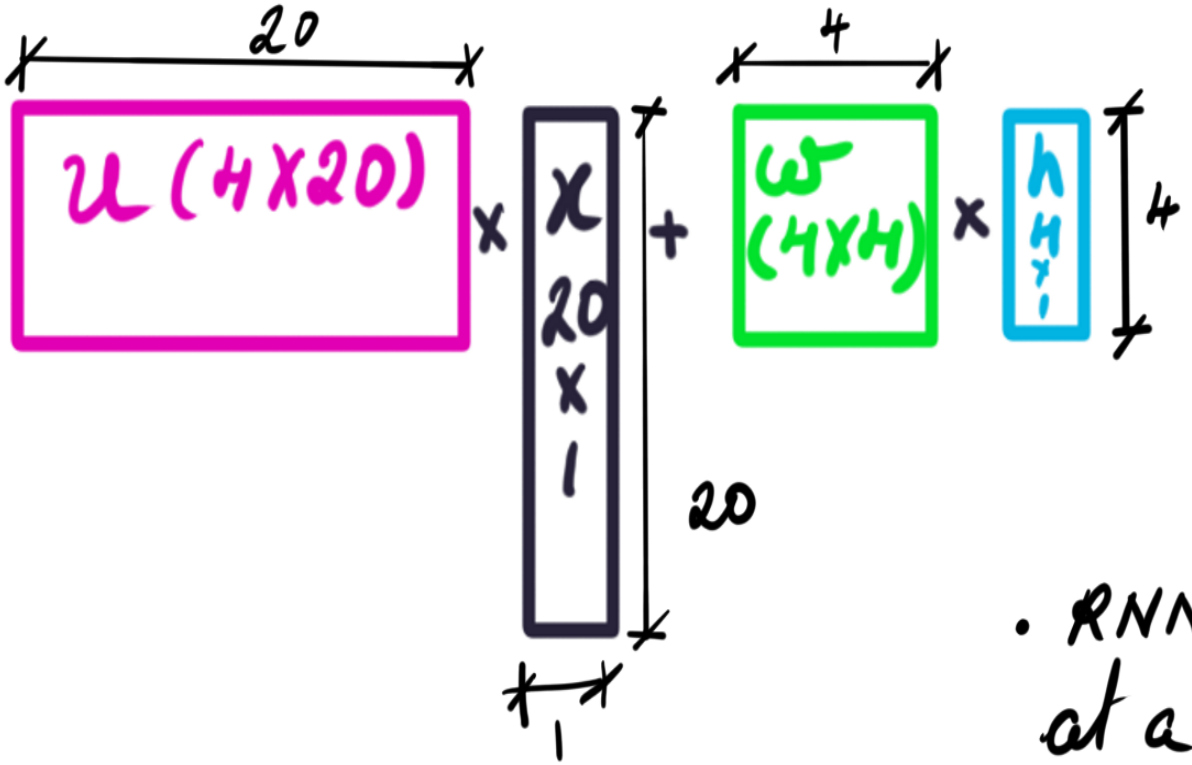
$$\frac{\partial E}{\partial hs_{t-1}} = (\hat{y} - y) \cdot v \cdot (1 - hs_t^2) \cdot w \quad \text{eq-13}$$

(eq-3, eq-6, eq-7, eq-12)

- Apply gradients.

- softmaxed output or $\hat{y}$
- dhnext is the portion that propagates back to step-0. 0 to begin with.

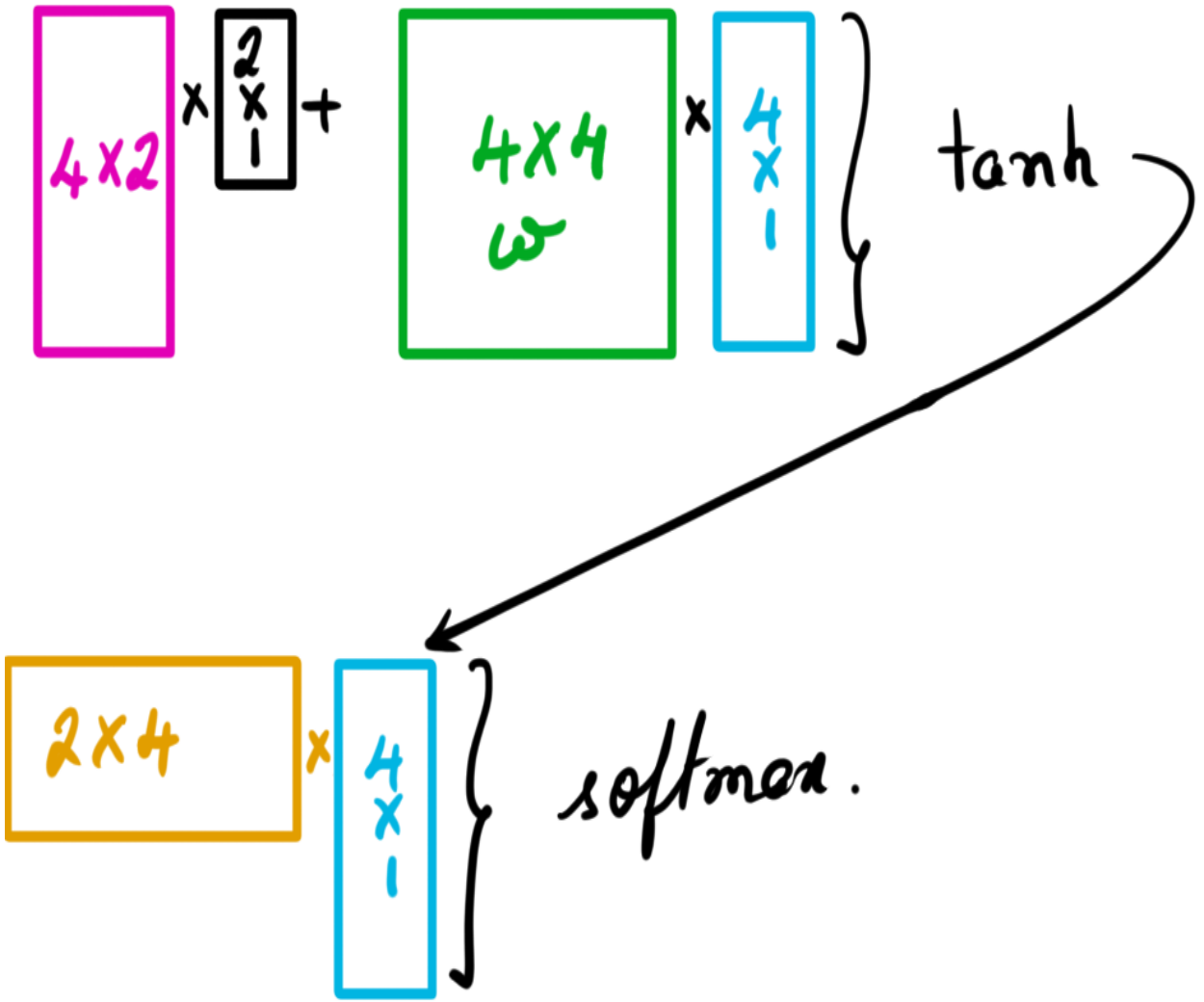
Recurrent Neural Networks:ex-2



For this example
state size = 4
num_classes = 20
batch size = 1
sequence size = 5.

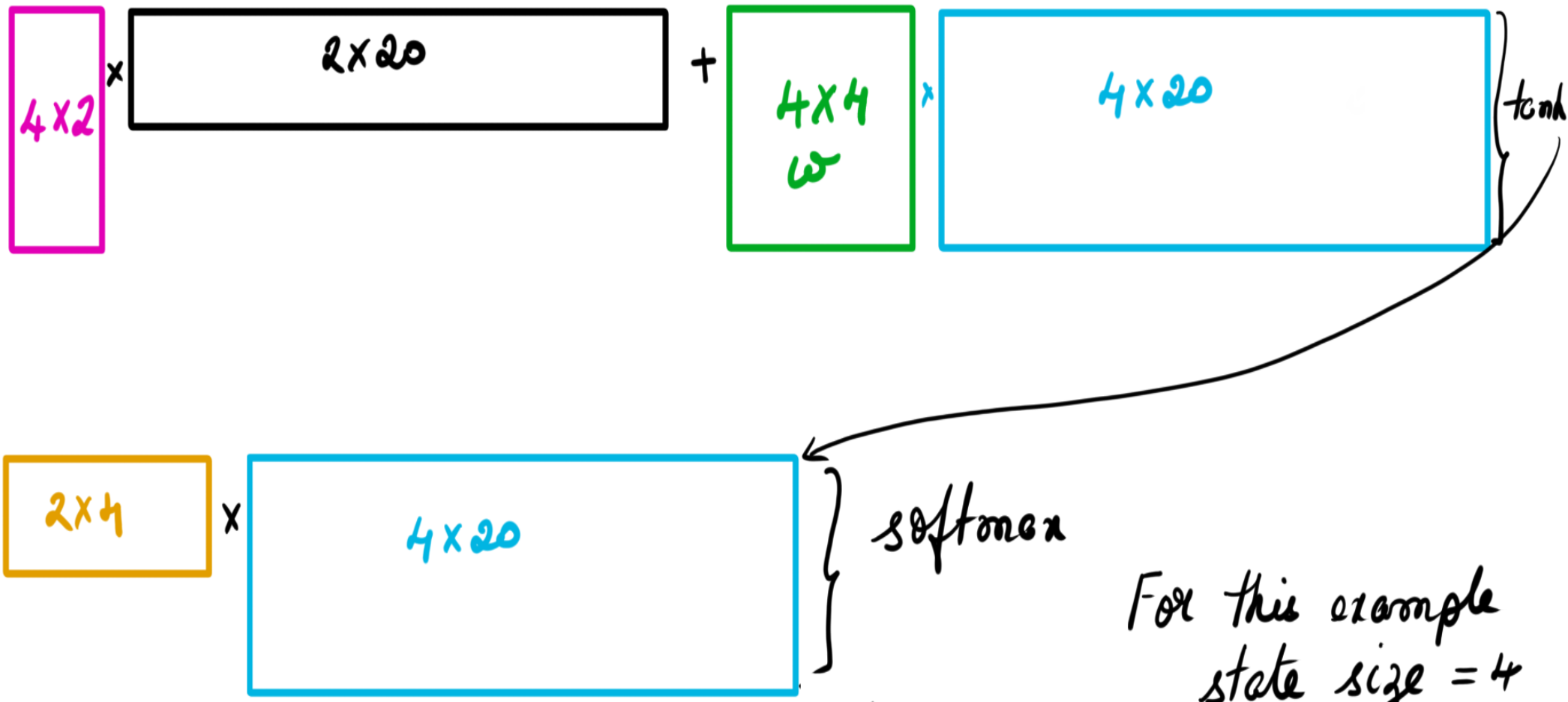
- RNN is processing one sample at a time. Bad for CPU + especially bad for GPU

Recurrent Neural Networks:ex-2



For this example
state size = 4
num_classes = 2
batch size = 1
sequence size = 5.

Recurrent Neural Networks:ex-2



- a batch of 20 getting processed together.

For this example
state size = 4
num_classes = 2
batch size = 20
sequence size = 5.

Recurrent Neural Networks:ex-2

```
num_steps = 5 # number of truncated backprop steps ('n' in the discussion abov
batch_size = 20
num_classes = 2
state_size = 4
learning_rate = 0.1
np.random.seed(42)

def gen_data(size=200):
    X = np.array(np.random.choice(2, size=(size,)))

    Y = []
    for i in range(size):
        threshold = 0.5
        if X[i-3] == 1:
            threshold += 0.5
        if X[i-8] == 1:
            threshold -= 0.25
        if np.random.rand() > threshold:
            Y.append(0)
        else:
            Y.append(1)
    return X, np.array(Y)
```

- size being 200

- A sequence generator
- Input sequence
  - At time step t X_t has a 50% chance of being a '1'.
- Output sequence
  - At time step t Y_t has a 50% chances of being a 1.
  - The chance of Y_t being 1 increases by 50% (i.e. to 100) if X_{t-3} is 1.

- The chance of Y_t being 1 decreases by 25% if X_{t-8} is 1.
- if both X_{t-3}, X_{t-8} are 1, then 75% chance of being 1.

Recurrent Neural Networks:ex-2

```
def gen_batch(raw_data, batch_size, num_steps):
    raw_x, raw_y = raw_data
    data_length = len(raw_x)
    # partition raw data into batches and stack them vertically
    batch_partition_length = data_length // batch_size
    data_x = np.zeros([batch_size, batch_partition_length], dtype=np.int32)
    data_y = np.zeros([batch_size, batch_partition_length], dtype=np.int32)
    #make it 20X10 shape (data_x)
    for i in range(batch_size):
        data_x[i] = raw_x[batch_partition_length * i:batch_partition_length * (i + 1)]
        data_y[i] = raw_y[batch_partition_length * i:batch_partition_length * (i + 1)]
    # further divide batch partitions into num_steps for truncated backprop
    epoch_size = batch_partition_length // num_steps
    for i in range(epoch_size):
        #x.shape=20X5
        x = data_x[:, i * num_steps:(i + 1) * num_steps]
        y = data_y[:, i * num_steps:(i + 1) * num_steps]

        yield (x,y)

def gen_epochs(n, num_steps):
    for i in range(n):
        yield gen_batch(gen_data(), batch_size, num_steps)

for idx, epoch in enumerate(gen_epochs(1, num_steps)):
    for step, (X, Y) in enumerate(epoch):
        print("X.shape",X.shape)
        print("step:",step," X:",X)
```

```
X.shape: (200,)  X: [0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 1 1 1 0]
Y.shape: 200  Y: [1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1]
```

- Printing the first 20.

```
[[0 1 0 0 0 1 0 0 0 1]
 [0 0 0 0 1 0 1 1 1 0]
 [1 0 1 1 1 1 1 1 1 1]
 [0 0 1 1 1 0 1 0 0 0]
 [0 0 1 1 1 1 1 0 1 1]
 [0 1 0 1 0 1 1 0 0 0]
 [0 0 0 0 0 1 1 0 1 1]
 [1 1 0 1 0 1 1 1 0 1]
 [0 1 0 1 0 0 1 0 1 1]
 [1 1 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 0]
 [1 0 1 1 0 1 0 1 1 0]
 [1 0 1 0 0 1 1 0 1 1]
 [1 0 0 0 0 0 0 0 0 0]
 [1 0 1 1 1 0 0 0 0 1]
 [0 0 0 0 0 1 0 1 0 1]
 [0 0 1 1 1 0 1 0 0 1]
 [1 0 0 1 1 1 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 1]
 [0 0 0 0 0 1 1 1 0 0]]
```

- Complete sequence of 200 stacked into a matrix of (20X10)
- generate an (1) epoch of data with step size being 5

20 5

- Dont return sequence as one sequence, slice it into multiple sequences governed by batchsize and steps.

Recurrent Neural Networks:ex-2

```
def gen_batch(raw_data, batch_size, num_steps):
    raw_x, raw_y = raw_data
    data_length = len(raw_x)
    # partition raw data into batches and stack them vertically in a data matrix
    batch_partition_length = data_length // batch_size
    data_x = np.zeros([batch_size, batch_partition_length], dtype=np.int32)
    data_y = np.zeros([batch_size, batch_partition_length], dtype=np.int32)
    #make it 20X10 shape (data_x)
    for i in range(batch_size):
        data_x[i] = raw_x[batch_partition_length * i:batch_partition_length * (i + 1)]
        data_y[i] = raw_y[batch_partition_length * i:batch_partition_length * (i + 1)]
    # further divide batch partitions into num_steps for truncated backprop
    epoch_size = batch_partition_length // num_steps
    for i in range(epoch_size):
        #x.shape=20X5
        x = data_x[:, i * num_steps:(i + 1) * num_steps]
        y = data_y[:, i * num_steps:(i + 1) * num_steps]

        yield (x,y)

def gen_epochs(n, num_steps):
    for i in range(n):
        yield gen_batch(gen_data(), batch_size, num_steps)

for idx, epoch in enumerate(gen_epochs(1, num_steps)):
    for step, (X, Y) in enumerate(epoch):
        print("X.shape",X.shape)
        print("step:",step," X:",X)
```

200 20 10 2 5

- Slicing logic (20 x 10) of zeros
- (20 x 10) data shown on previous page.
- Keep rows and slice columns by num steps. Python arrays are row major like C, C++.

```
[[0 1 0 0 0]
 [0 0 0 0 1]
 [1 0 1 1 1]
 [0 0 1 1 1]
 [0 0 1 1 1]
 [0 1 0 1 0]
 [0 0 0 0 0]
 [1 1 0 1 0]
 [0 1 0 1 0]
 [1 1 1 1 1]
 [0 1 1 1 1]
 [1 0 1 1 0]
 [1 0 1 0 0]
 [1 0 0 0 0]
 [1 0 1 1 1]
 [0 0 0 0 0]
 [0 0 1 1 1]
 [1 0 0 1 1]
 [0 0 1 0 0]
 [0 0 0 0 0]]
```

```
[[1 0 0 0 1]
 [0 1 1 1 0]
 [1 1 1 1 1]
 [0 1 0 0 0]
 [1 1 0 1 1]
 [1 1 0 0 0]
 [1 1 0 1 1]
 [1 1 1 0 1]
 [0 1 0 1 1]
 [1 1 1 1 0]
 [1 1 1 1 0]
 [1 0 1 1 0]
 [1 1 0 1 1]
 [0 0 0 0 0]
 [0 0 0 0 1]
 [1 0 1 0 1]
 [0 1 0 0 1]
 [1 0 0 0 0]
 [0 1 0 0 1]
 [1 1 1 0 0]]
```

- Data will be fed one batch (20) at a time for 5 steps and then BPTT before the next batch.

Recurrent Neural Networks:ex-2:Theoretical Probabilities And CrossEntropy loss

- **Network learns Neither dependency**
- Event 1: P(yt=1) = P(yt=1 | xt-3 = 1, Xt-8 = 1)*P(xt-3 = 1,)*P(Xt-8 = 1)

  + P(yt=1 | xt-3 = 1, Xt-8 = 0)*P(xt-3= 1)*P(Xt-8 = 0)

  + P(yt=1 | xt-3 = 0, Xt-8 = 1)*P(xt-3= 0)*P(Xt-8 = 1)

  + P(yt=1 | xt-3 = 0, Xt-8 = 0)*P(xt-3= 0)*P(Xt-8 = 0)

  P(yt=1) = 0.75 * 0.25 + 1 * 0.25 + 0.25 * 0.25 + 0.5 *0.25 = **0.625**
- Event 2: P(yt=0) = P(yt=0 | xt-3 = 1, Xt-8 = 1)*P(xt-3= 1)*P(Xt-8 = 1)

  + P(yt=0 | xt-3 = 1, Xt-8 = 0)*P(xt-3= 1)*P(Xt-8 = 0)

  + P(yt=0 | xt-3 = 0 Xt-8 = 1)*P(xt-3= 0)*P(Xt-8 = 1)

  + P(yt=0 | xt-3 = 0, Xt-8 = 0)*P(xt-3= 0)*P(Xt-8 = 0)
- P(yt=0) = 0.25 * 0.25 + 0 * 0.25 + 0.75 * 0.25 + 0.5 * 0.25 = **0.375**
- **Theoretical CEL:**
  - **cross entropy = -(0.625 * np.log(0.625) + 0.375 * np.log(0.375))**
  - **0.661563238158**

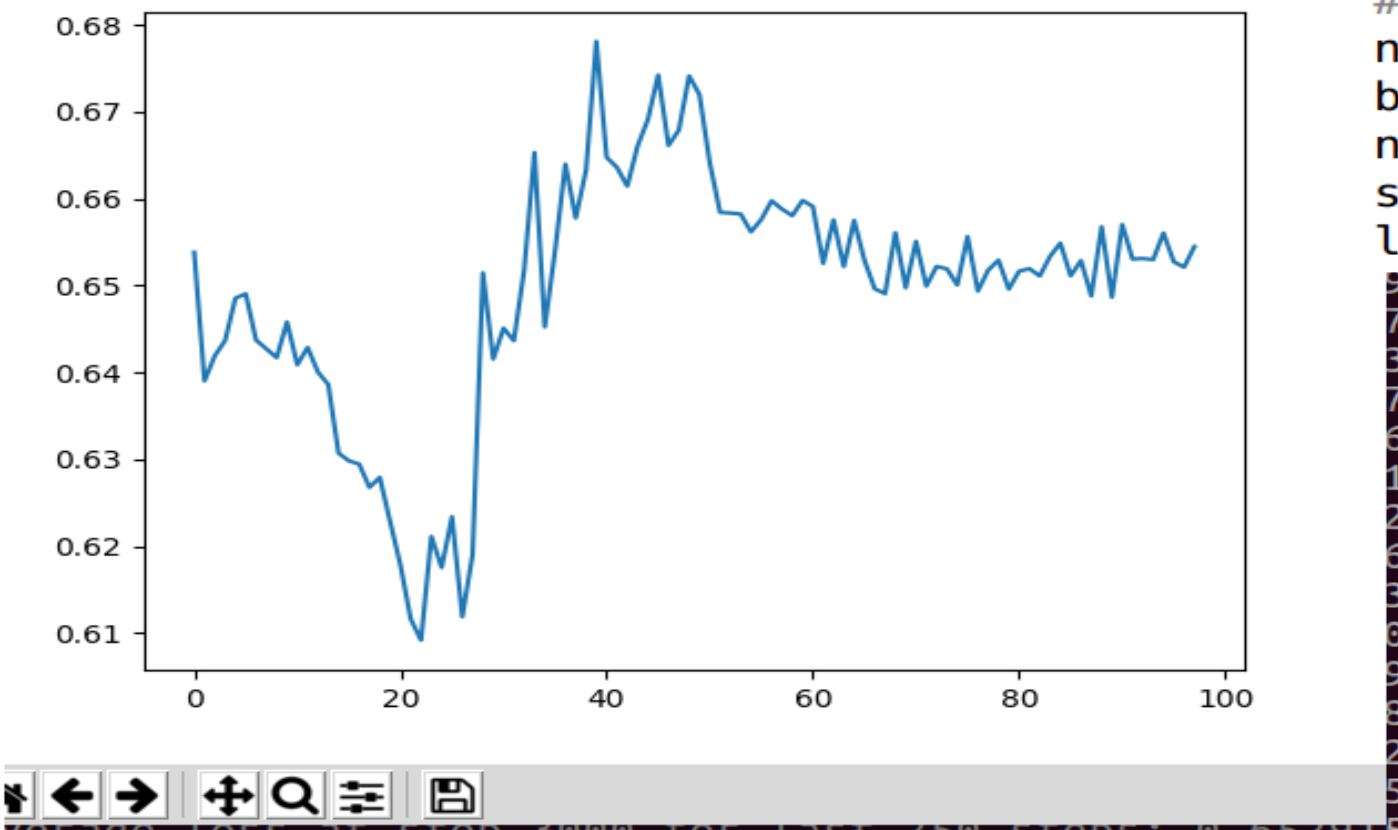
Recurrent Neural Networks:ex-2:Theoretical Probabilities And CrossEntropy loss

- **Network learns first dependency only**
- Event 1: P(yt=1 | xt-3 = 1) = P(yt=1 | xt-3 = 1, Xt-8 = 1) * P(Xt-8 = 1) + P(yt=1 | xt-3 = 1, Xt-8 = 0) * P(Xt-8 = 0) = 0.75 * 0.5 + 1 * 0.5=**0.875**
- Event 2: P(yt=1 | xt-3 = 0) = P(yt=1 | xt-3 = 0, Xt-8 = 1) * P(Xt-8 = 1) + P(yt=1 | xt-3 = 0, Xt-8 = 0) * P(Xt-8 = 0) = 0.25 * 0.5 + 0.5 * 0.5=**0.325**
- Event 3: P(yt=0 | xt-3 = 1) = P(yt=0 | xt-3 = 1, Xt-8 = 1) * P(Xt-8 = 1) + P(yt=0 | xt-3 = 1, Xt-8 = 0) * P(Xt-8 = 0) = 0.25 * 0.5 + 0 * 0.5=**0.125**
- Event 4: P(yt=0 | xt-3 = 0) = P(yt=0 | xt-3 = 0, Xt-8 = 1) * P(Xt-8 = 1) + P(yt=0 | xt-3 = 0, Xt-8 = 0) * P(Xt-8 = 0) = 0.75 * 0.5 + 0.5 * 0.5=**0.625**
- **Theoretical CEL:**
  - **cross entropy = -0.5 * (0.875 * np.log(0.875) + 0.125 * np.log(0.125) - 0.5 * (0.625 * np.log(0.625) + 0.375 * np.log(0.375)))**
  - **0.519166699707**

Recurrent Neural Networks:ex-2:Theoretical Probabilities And CrossEntropy loss

- **Network learns second dependency**
- Event 1: P(yt=1 | xt-3 = 1, Xt-8 = 1) = 0.75
- Event 2: P(yt=1 | xt-3 = 1, Xt-8 = 0) = 1
- Event 3: P(yt=1 | xt-3 = 0, Xt-8 = 1) = 0.25
- Event 4: P(yt=1 | xt-3 = 0, Xt-8 = 0) = 0.5
- Event 5: P(yt=0 | xt-3 = 1, Xt-8 = 1) = 0.25
- Event 6: P(yt=0 | xt-3 = 1, Xt-8 = 0) = 0
- Event 7: P(yt=0 | xt-3 = 0, Xt-8 = 1) = 0.75
- Event 8: P(yt=0 | xt-3 = 0, Xt-8 = 0) = 0.5
- **Theoretical CEL:**
  - **cross entropy = -0.25 * (2 * 0.75 * np.log(0.75) + 2 * 0.25 * np.log(0.25) + 2 * 0.50 * np.log (0.50)) - 0.25 * (0 * np.log(0)) -0.25 * (1 * log(1))**
  - **0.454454367449**

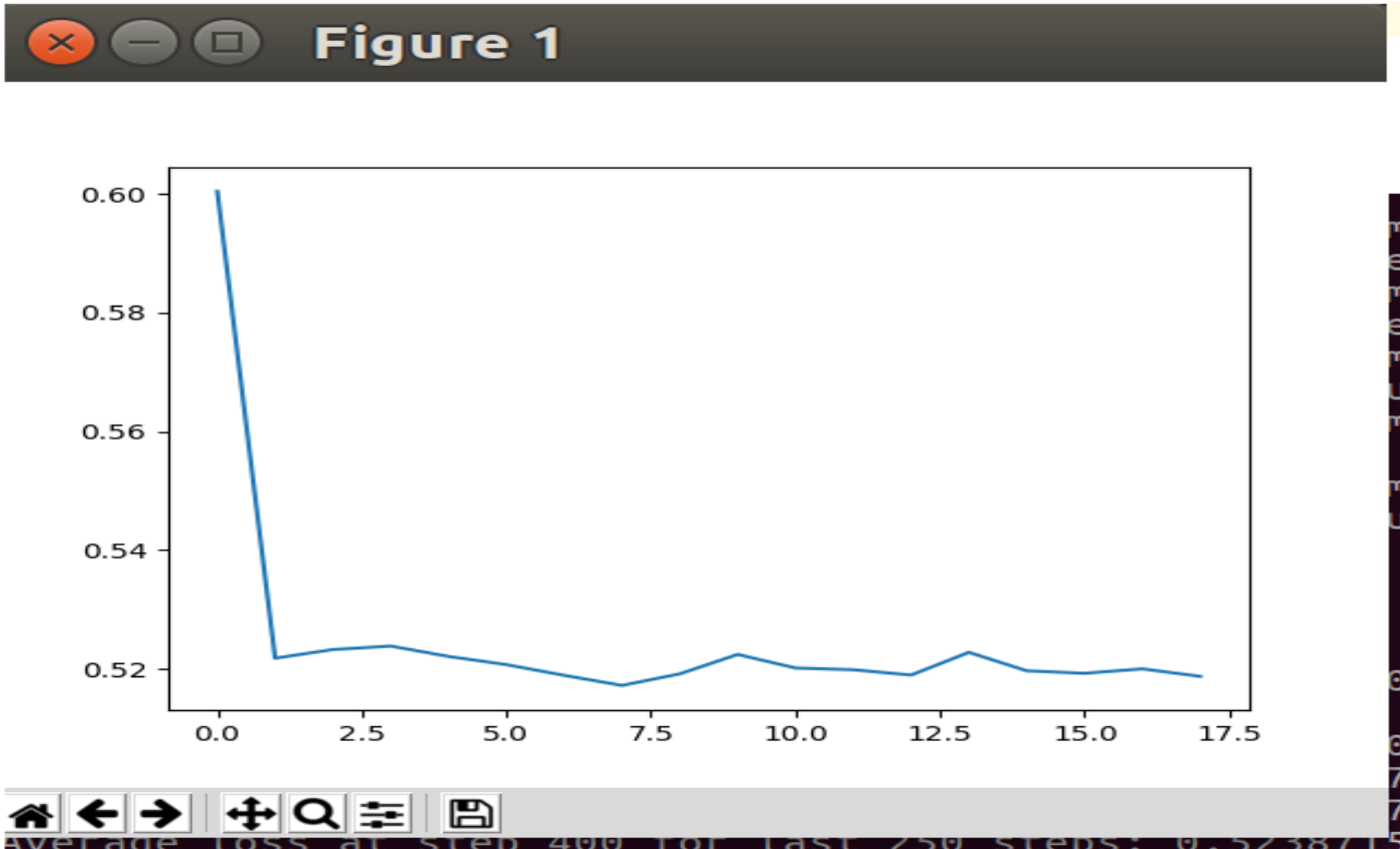
Recurrent Neural Networks:ex-2:RNN Learning




```
# Global config var
num_steps = 1 # num
batch_size = 200
num_classes = 2
state_size = 4
learning_rate = 0.1
```

```
9004135
7688618
3642097
7688584
6511869
1663337
2116723
6877618
393909
0808682
9353709
8154278
2678061
5443087
verage loss at step 3000 for last 250 steps: 0.652915305495
verage loss at step 3100 for last 250 steps: 0.649582335353
verage loss at step 3200 for last 250 steps: 0.651646110415
verage loss at step 3300 for last 250 steps: 0.651933643818
verage loss at step 3400 for last 250 steps: 0.651090717316
verage loss at step 3500 for last 250 steps: 0.65334869802
verage loss at step 3600 for last 250 steps: 0.654877930284
verage loss at step 3700 for last 250 steps: 0.651112522483
verage loss at step 3800 for last 250 steps: 0.652866544127
verage loss at step 3900 for last 250 steps: 0.648794487119
verage loss at step 4000 for last 250 steps: 0.656758444309
verage loss at step 4100 for last 250 steps: 0.64865501821
verage loss at step 4200 for last 250 steps: 0.657008399963
verage loss at step 4300 for last 250 steps: 0.653033348322
verage loss at step 4400 for last 250 steps: 0.653102155924
verage loss at step 4500 for last 250 steps: 0.652970516682
```

Recurrent Neural Networks:ex-2:RNN Learning




```
num_steps = 5 # numb
batch_size = 200
num_classes = 2
state_size = 4
learning_rate = 0.1
```

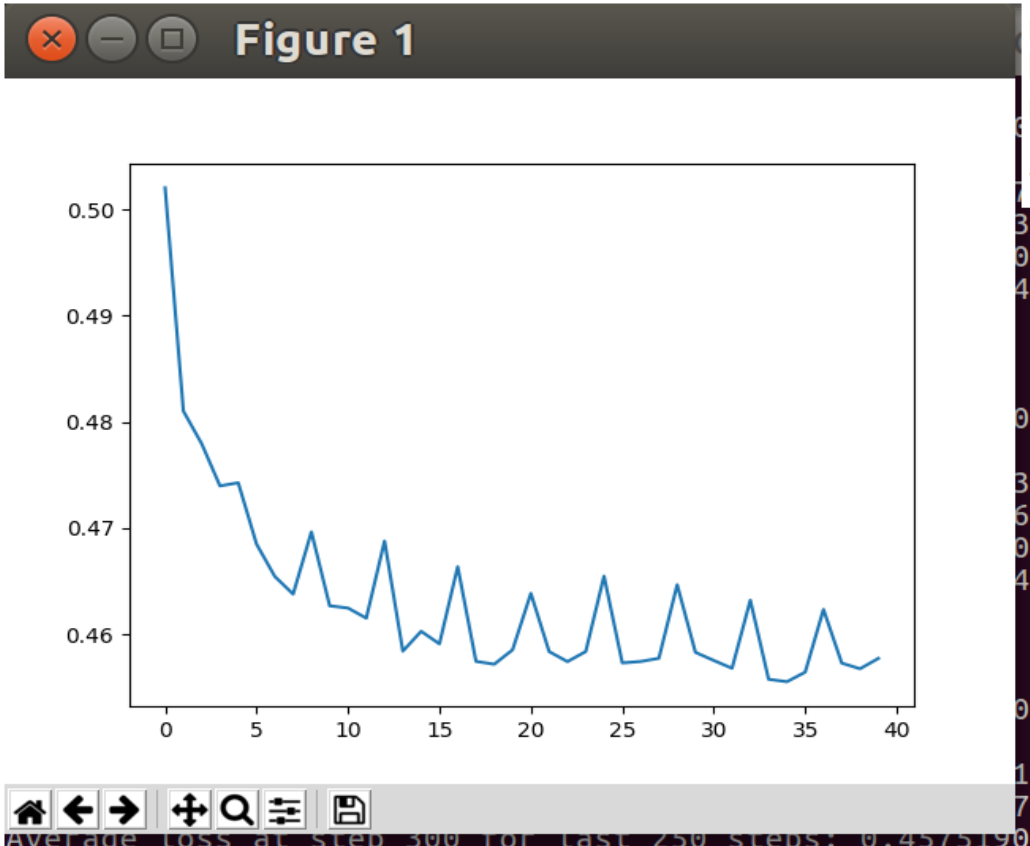
```
m/cpu_
ed up
m/cpu_
ed up
m/cpu_
up CPU
m/cpu_
 up CPU
m/cpu_
up CPU


0)  da

026546
716465
754953
Average loss at step 400 for last 250 steps: 0.523871533871
Average loss at step 500 for last 250 steps: 0.522127411962
Average loss at step 600 for last 250 steps: 0.520748464763
Average loss at step 700 for last 250 steps: 0.518941668868
Average loss at step 800 for last 250 steps: 0.517246327102
Average loss at step 900 for last 250 steps: 0.51919929862

EPOCH 1
data_length: 1000000  batch_size: 200  num_steps: 5
batch_partition_length: 5000  data_x.shape: (200, 5000)  da
epoch_size: 1000
Average loss at step 100 for last 250 steps: 0.522466212809
Average loss at step 200 for last 250 steps: 0.52018302083
Average loss at step 300 for last 250 steps: 0.519889618456
Average loss at step 400 for last 250 steps: 0.518994677961
Average loss at step 500 for last 250 steps: 0.522818899751
Average loss at step 600 for last 250 steps: 0.519711365104
Average loss at step 700 for last 250 steps: 0.519293204844
Average loss at step 800 for last 250 steps: 0.520039466321
Average loss at step 900 for last 250 steps: 0.518773700893
```

Recurrent Neural Networks:ex-2:RNN Learning




```
num_steps = 10 # num
batch_size = 200
num_classes = 2
state_size = 16
learning_rate = 0.1
```

```
360261
088612
445862


0)  da

308628
603451
078218
45864


0)  da

188949
731836
Average loss at step 300 for last 250 steps: 0.45751902490
Average loss at step 400 for last 250 steps: 0.456780762374

EPOCH 8
data_length: 1000000  batch_size: 200  num_steps: 10
batch_partition_length: 5000  data_x.shape: (200, 5000)  da
epoch_size: 500
Average loss at step 100 for last 250 steps: 0.463193096519
Average loss at step 200 for last 250 steps: 0.455746220648
Average loss at step 300 for last 250 steps: 0.455514635146
Average loss at step 400 for last 250 steps: 0.456414278746

EPOCH 9
data_length: 1000000  batch_size: 200  num_steps: 10
batch_partition_length: 5000  data_x.shape: (200, 5000)  da
epoch_size: 500
Average loss at step 100 for last 250 steps: 0.462322815061
Average loss at step 200 for last 250 steps: 0.457267158926
Average loss at step 300 for last 250 steps: 0.456735696197
Average loss at step 400 for last 250 steps: 0.457708996832
```