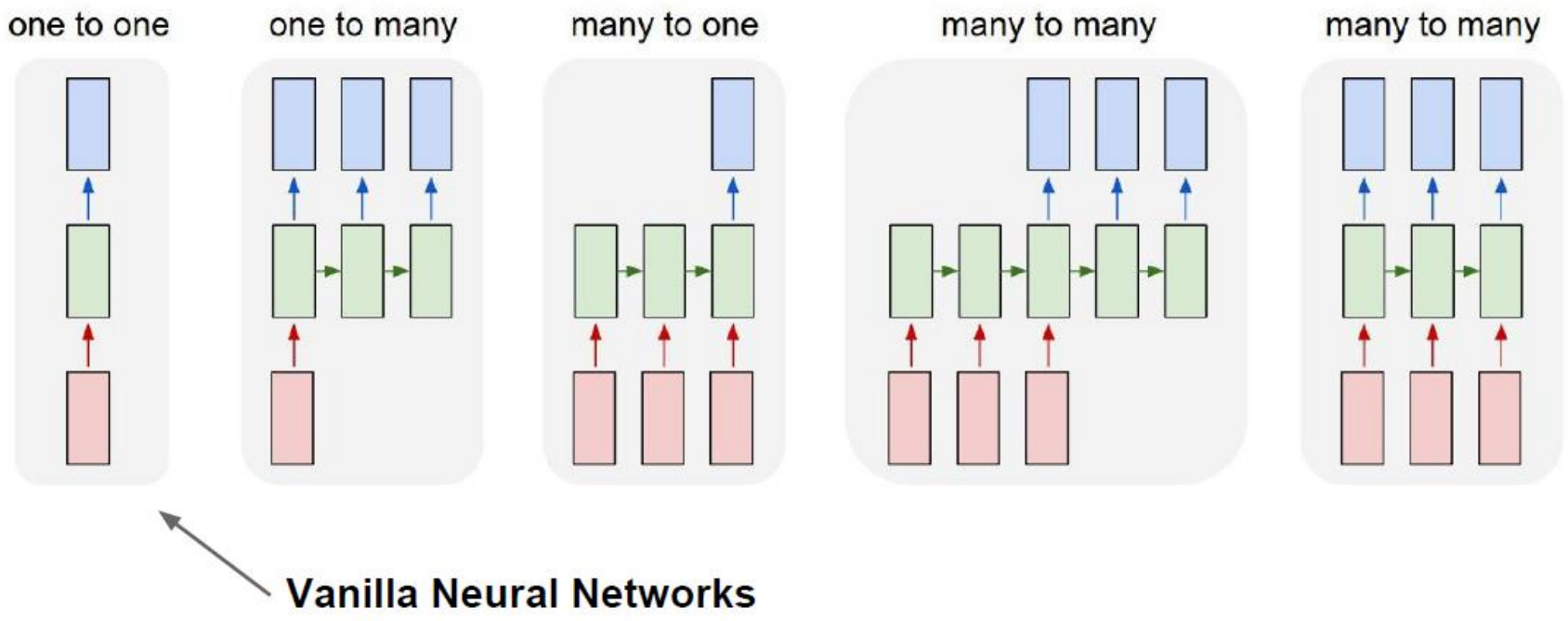
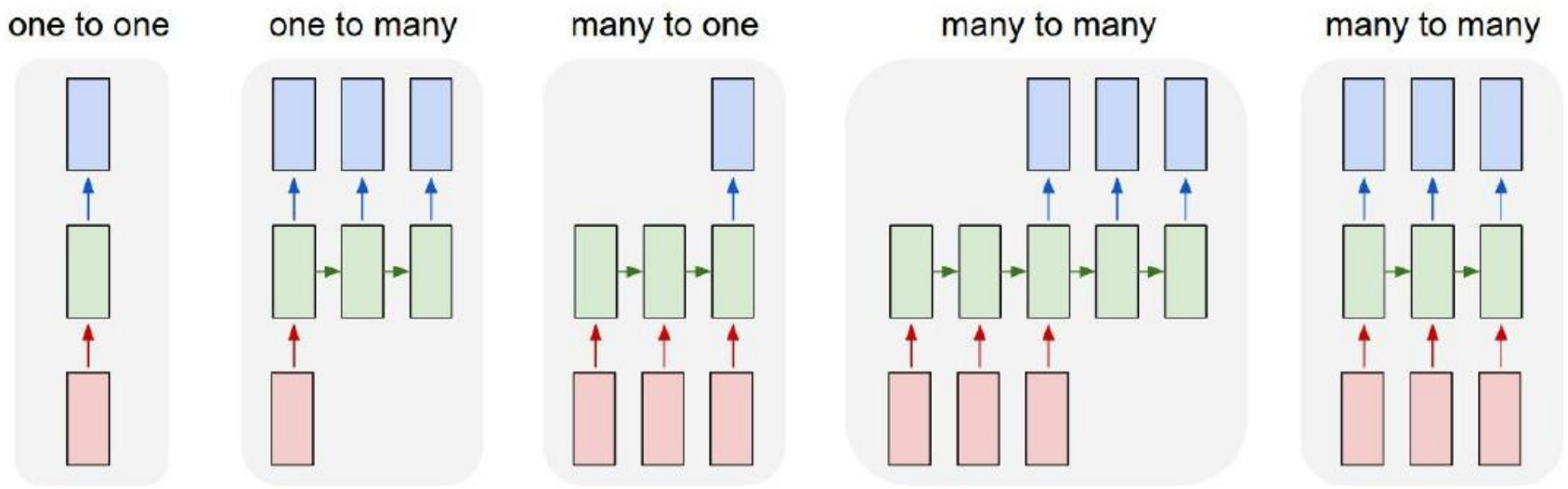
RECURRENT NEURAL NETWORKS

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

RNN offers a lot of flexibility

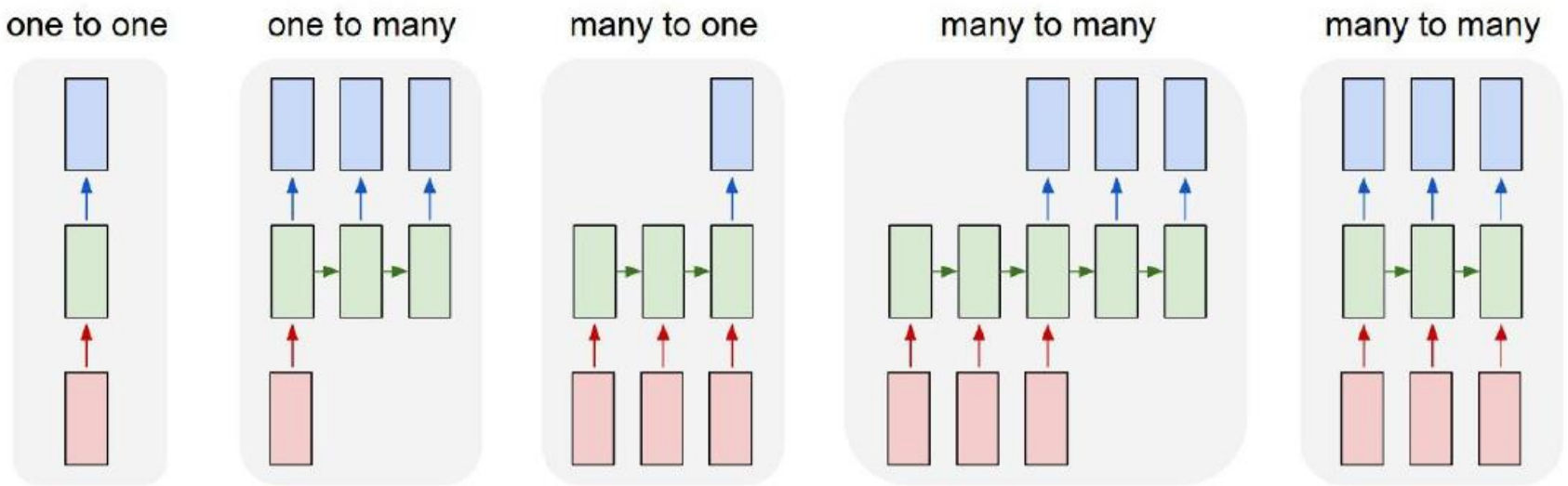


RNN offers a lot of flexibility



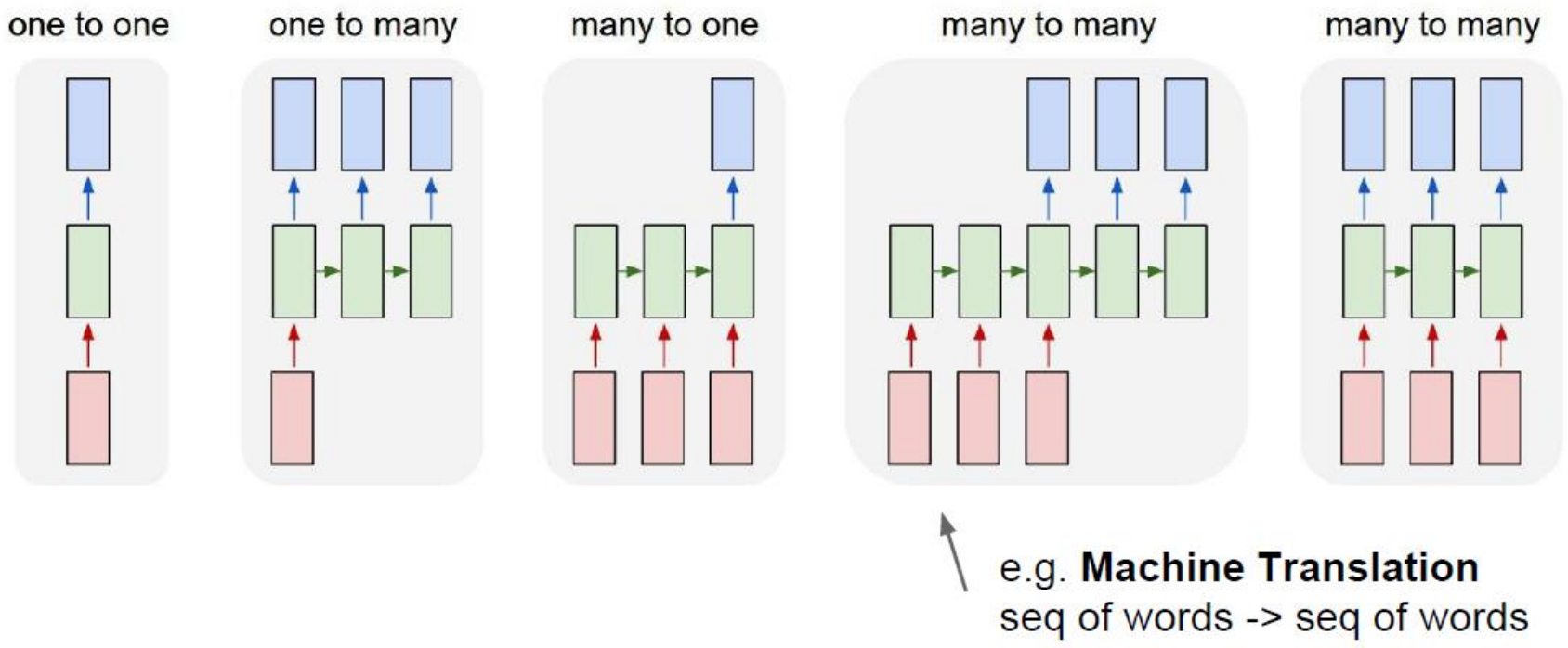
e.g. **Image Captioning**
image -> sequence of words

RNN offers a lot of flexibility

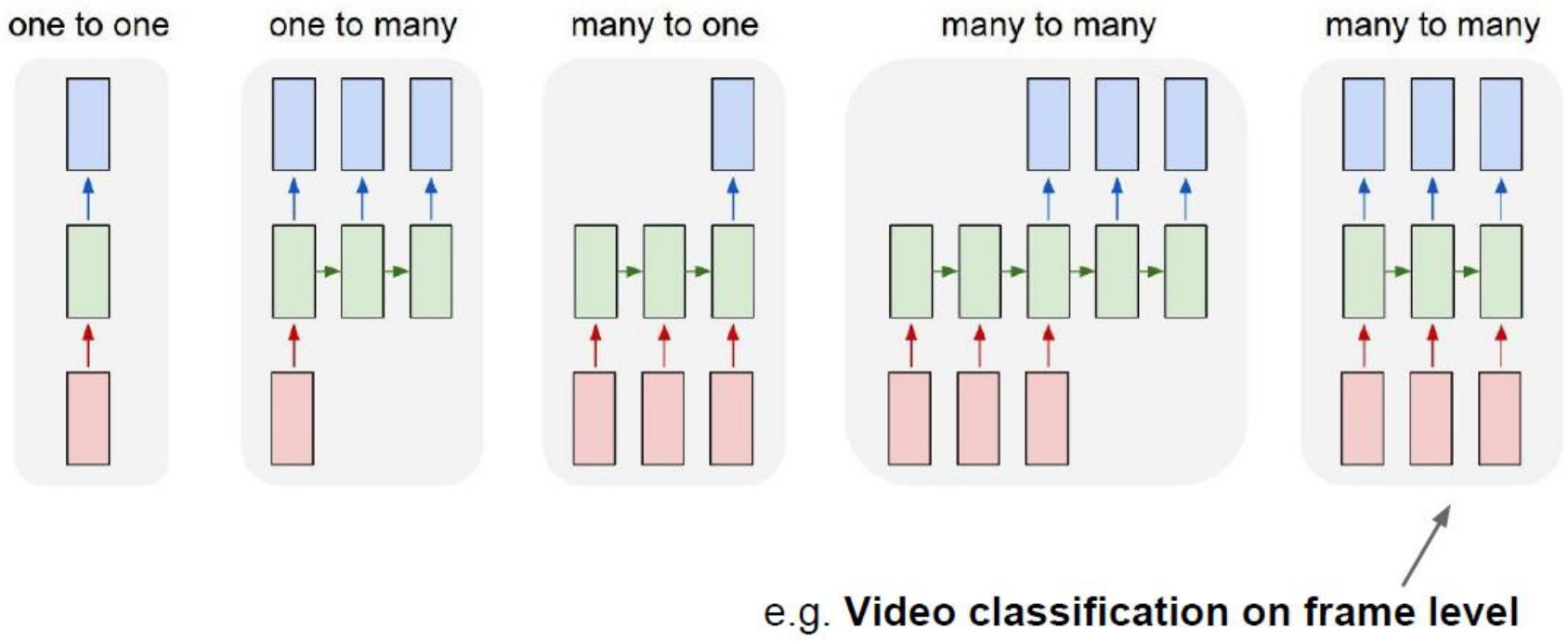


e.g. **Sentiment Classification**
sequence of words -> sentiment

RNN offers a lot of flexibility

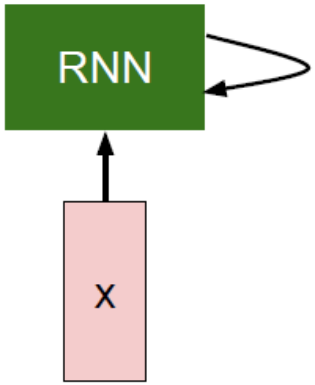


RNN offers a lot of flexibility

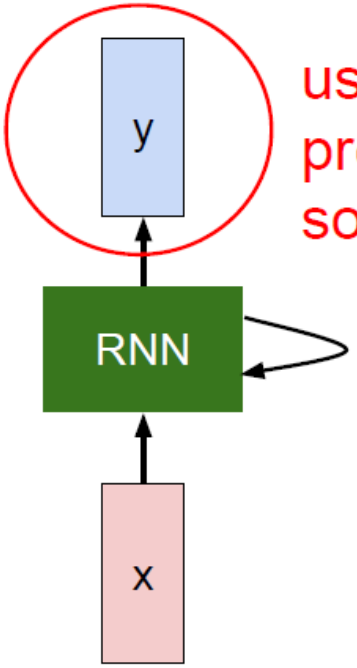


Sequential processing of fixed outputs

Recurrent Neural Network



Recurrent Neural Network



usually want to predict a vector at some time steps

Recurrent Neural Network

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

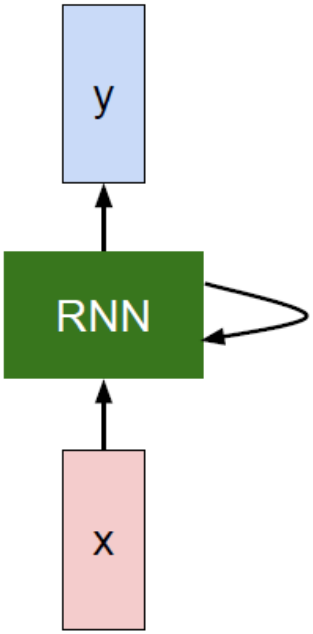
$$h_t = f_W(h_{t-1}, x_t)$$

new state

some function with parameters W

old state

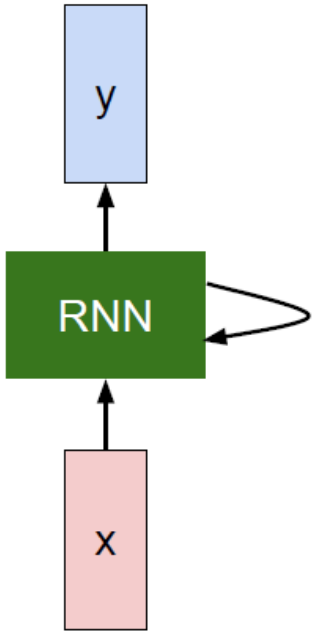
input vector at some time step



Recurrent Neural Network

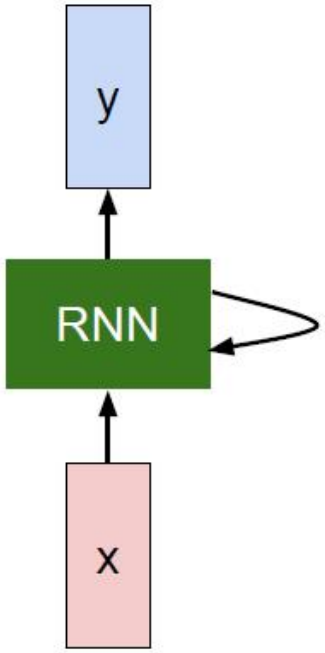
We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$



Notice: the same function and the same set of parameters are used at every time step.

(Vanilla) Recurrent Neural Network



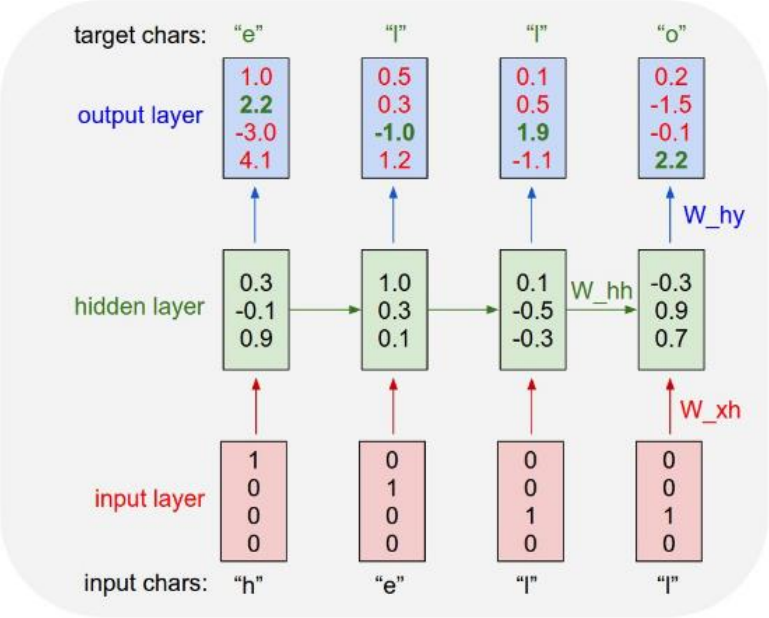
$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Character-level language model

- Vocabulary: [h,e,l,o]
- Training sequence: hello



We want the green numbers to be high and red numbers to be low.

$$W_{hh} \in \mathfrak{R}^{3\times 3}$$

$$W_{xh} \in \mathfrak{R}^{3\times 4}$$

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t + b_h\right)$$

Simple RNN code

min-char-rnn.py gist

Data I/O

```
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }
```

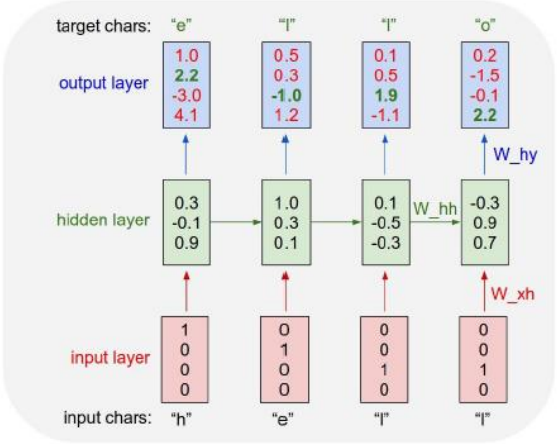
Simple RNN code

min-char-rnn.py gist

Initializations

```
# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias
```

recall:



$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t + b_h\right)$$

$$y_t = W_{hy}h_t + b_y$$

Simple RNN code

min-char-rnn.py gist

Main loop

```
n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    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]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

Simple RNN code

min-char-rnn.py gist

Main loop

```
n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    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]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

Simple RNN code

min-char-rnn.py gist

Main loop

```
n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    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]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

Simple RNN code

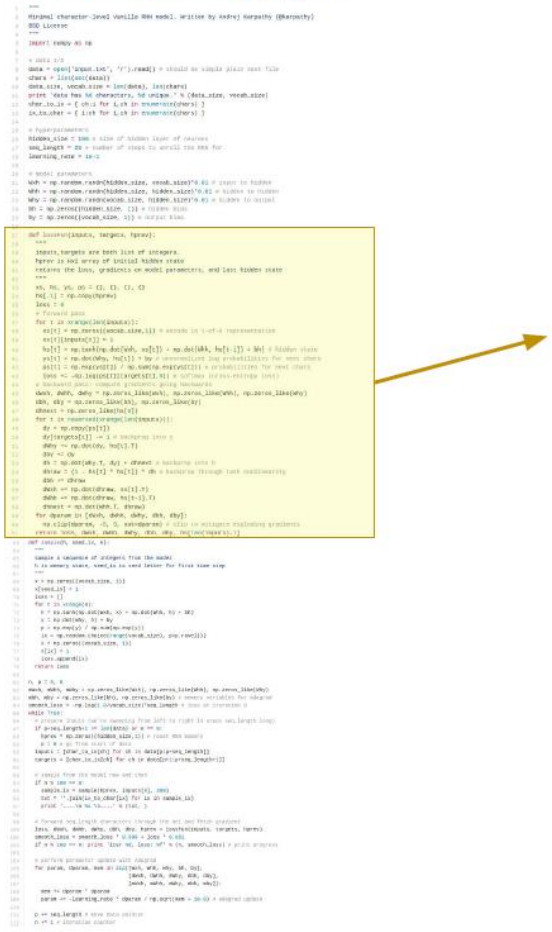
min-char-rnn.py gist

Main loop

```
n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    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]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

Simple RNN code

min-char-rnn.py gist



Loss function

- forward pass (compute loss)
- backward pass (compute param gradient)

```
27  def lossFun(inputs, targets, hprev):
28    """
29    inputs,targets are both list of integers.
30    hprev is Hx1 array of initial hidden state
31    returns the loss, gradients on model parameters, and last hidden state
32    """
33    xs, hs, ys, ps = {}, {}, {}, {}
34    hs[-1] = np.copy(hprev)
35    loss = 0
36    # forward pass
37    for t in xrange(len(inputs)):
38      xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39      xs[t][inputs[t]] = 1
40      hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
41      ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
42      ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43      loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44    # backward pass: compute gradients going backwards
45    dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46    dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47    dhnext = np.zeros_like(hs[0])
48    for t in reversed(xrange(len(inputs))):
49      dy = np.copy(ps[t])
50      dy[targets[t]] -= 1 # backprop into y
51      dWhy += np.dot(dy, hs[t].T)
52      dby += dy
53      dh = np.dot(Why.T, dy) + dhnext # backprop into h
54      dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55      dbh += dhraw
56      dWxh += np.dot(dhraw, xs[t].T)
57      dWhh += np.dot(dhraw, hs[t-1].T)
58      dhnext = np.dot(Whh.T, dhraw)
59    for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
60      np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61    return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

Simple RNN code

min-char-rnn.py gist

```
27  def lossFun(inputs, targets, hprev):
28    """
29    inputs,targets are both list of integers.
30    hprev is Hx1 array of initial hidden state
31    returns the loss, gradients on model parameters, and last hidden state
32    """
33    xs, hs, ys, ps = {}, {}, {}, {}
34    hs[-1] = np.copy(hprev)
35    loss = 0
36    # forward pass
37    for t in xrange(len(inputs)):
38      xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39      xs[t][inputs[t]] = 1
40      hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
41      ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
42      ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43      loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
```

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t + b_h\right)$$

$$y_t = W_{hy}h_t + b_y$$

$$p_t = \text{softmax}(y_t)$$

Softmax classifier

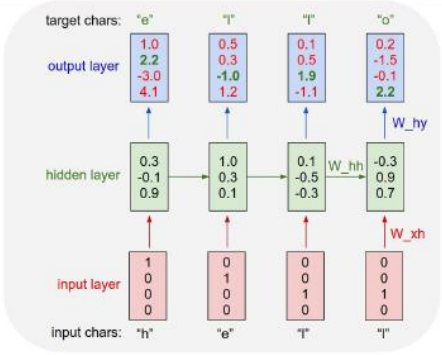
$$l \leftarrow l - \log\left(p_t\right)^{\top}[0 \ldots . 1 \ldots .0]$$

Simple RNN code

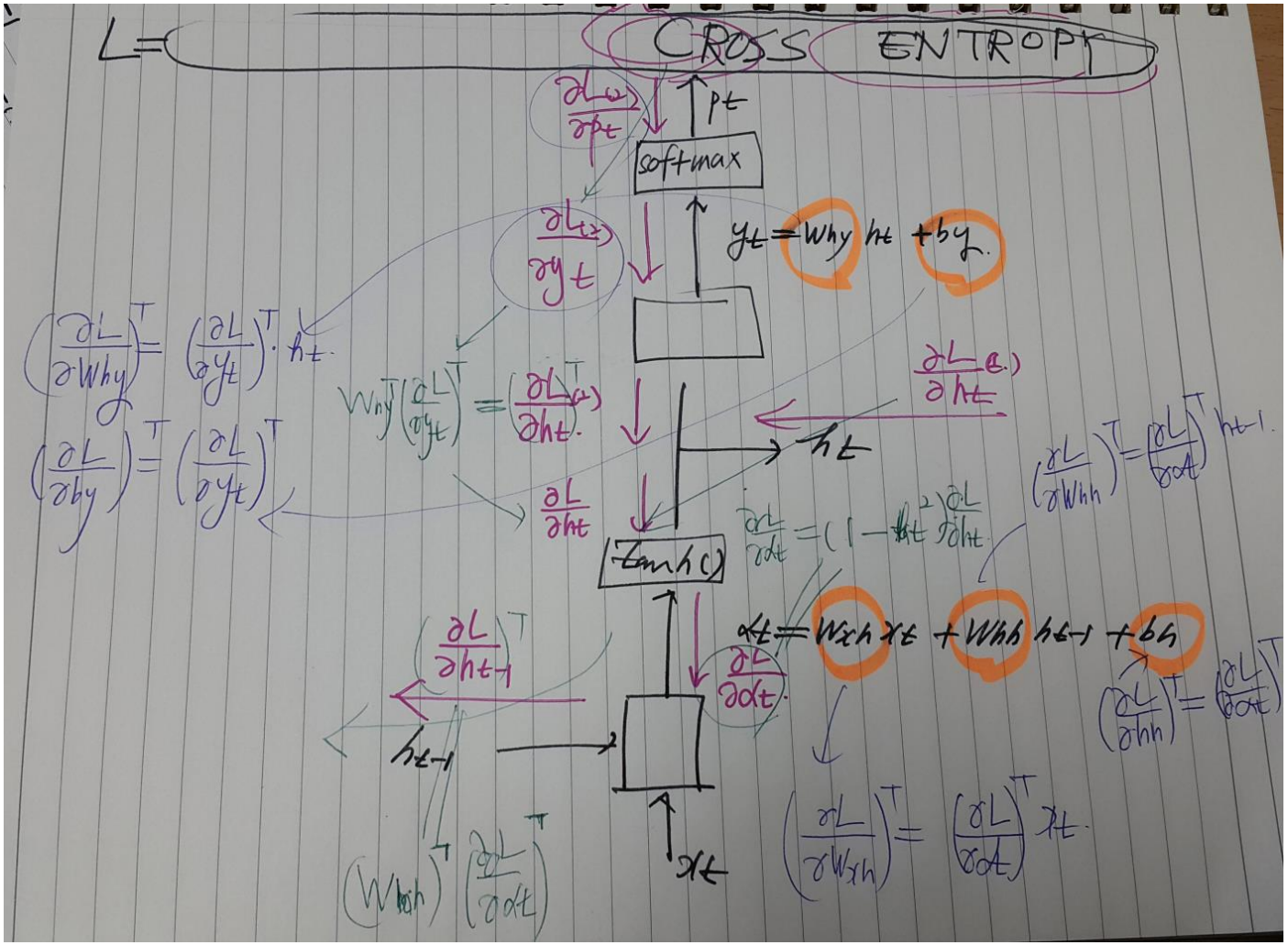
min-char-rnn.py gist

```
44  # backward pass: compute gradients going backwards
45  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47  dhnext = np.zeros_like(hs[0])
48  for t in reversed(xrange(len(inputs))):
49    dy = np.copy(ps[t])
50    dy[targets[t]] -= 1 # backprop into y
51    dWhy += np.dot(dy, hs[t].T)
52    dby += dy
53    dh = np.dot(Why.T, dy) + dhnext # backprop into h
54    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55    dbh += dhraw
56    dWxh += np.dot(dhraw, xs[t].T)
57    dWhh += np.dot(dhraw, hs[t-1].T)
58    dhnext = np.dot(Whh.T, dhraw)
59  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
60    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

recall:

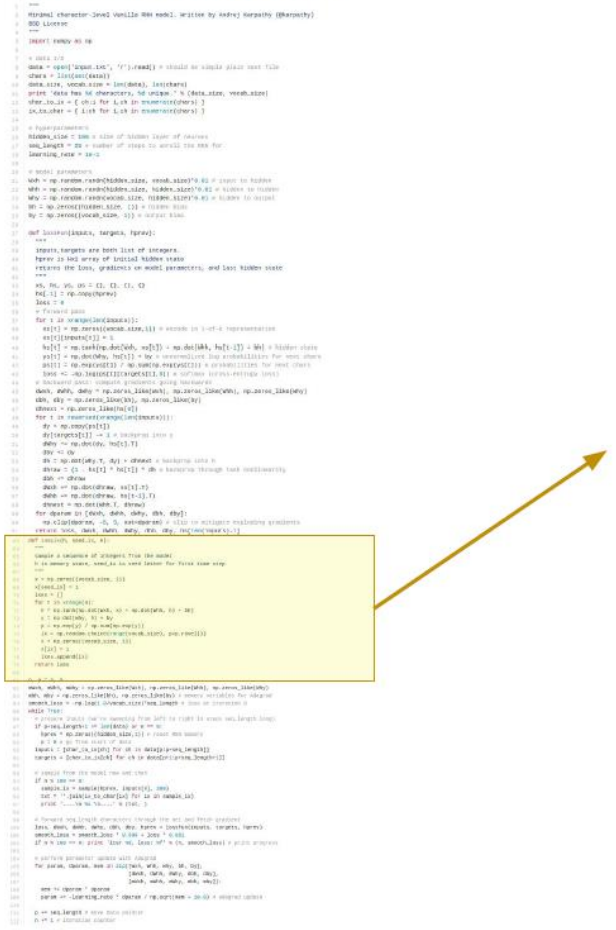


RNN math



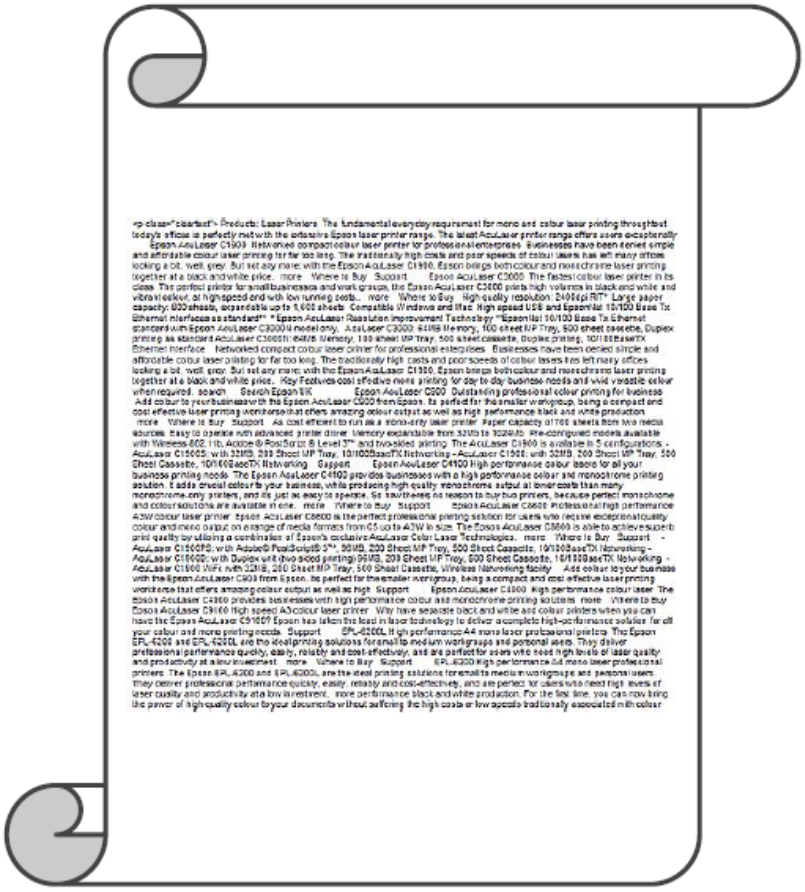
Simple RNN code

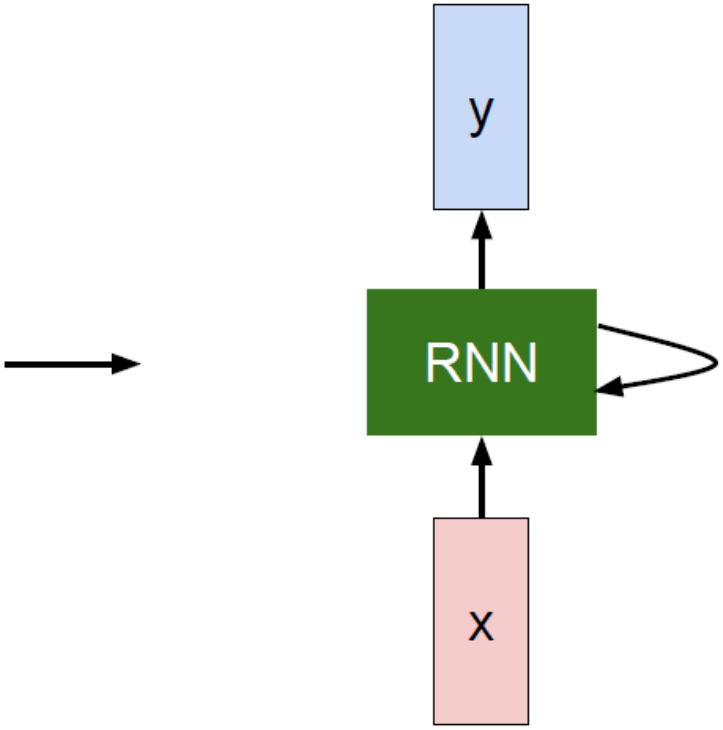
min-char-rnn.py gist



```
63  def sample(h, seed_ix, n):
64    """
65    sample a sequence of integers from the model
66    h is memory state, seed_ix is seed letter for first time step
67    """
68    x = np.zeros((vocab_size, 1))
69    x[seed_ix] = 1
70    ixes = []
71    for t in xrange(n):
72      h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
73      y = np.dot(Why, h) + by
74      p = np.exp(y) / np.sum(np.exp(y))
75      ix = np.random.choice(range(vocab_size), p=p.ravel())
76      x = np.zeros((vocab_size, 1))
77      x[ix] = 1
78      ixes.append(ix)
79    return ixes
```

Training





Results

at first:

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

train more

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

train more

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

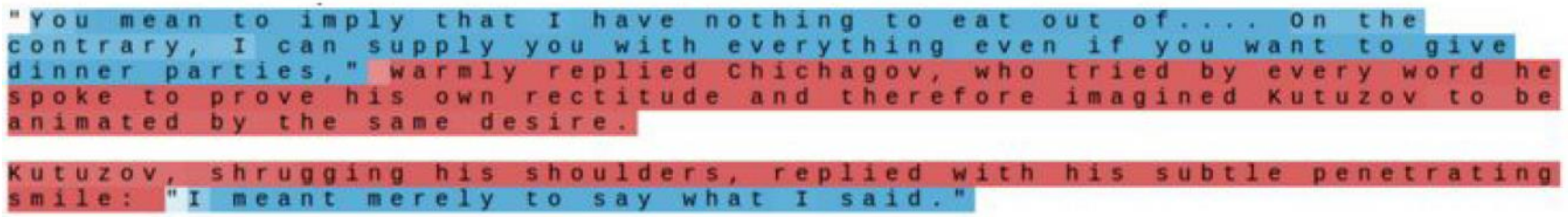
train more

```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

Generated C code

```
static void do_command(struct seq_file *m, void *v)
{
  int column = 32 << (cmd[2] & 0x80);
  if (state)
    cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
  else
    seq = 1;
  for (i = 0; i < 16; i++) {
    if (k & (1 << 1))
      pipe = (in_use & UMXTHREAD_UNCCA) +
        ((count & 0x00000000fffffff8) & 0x000000f) << 8;
    if (count == 0)
      sub(pid, ppc_md.kexec_handle, 0x20000000);
    pipe_set_bytes(i, 0);
  }
  /* Free our user pages pointer to place camera if all dash */
  subsystem_info = &of_changes[PAGE_SIZE];
  rek_controls(offset, idx, &soffset);
  /* Now we want to deliberately put it to device */
  control_check_polarity(&context, val, 0);
  for (i = 0; i < COUNTER; i++)
    seq_puts(s, "policy ");
}
```

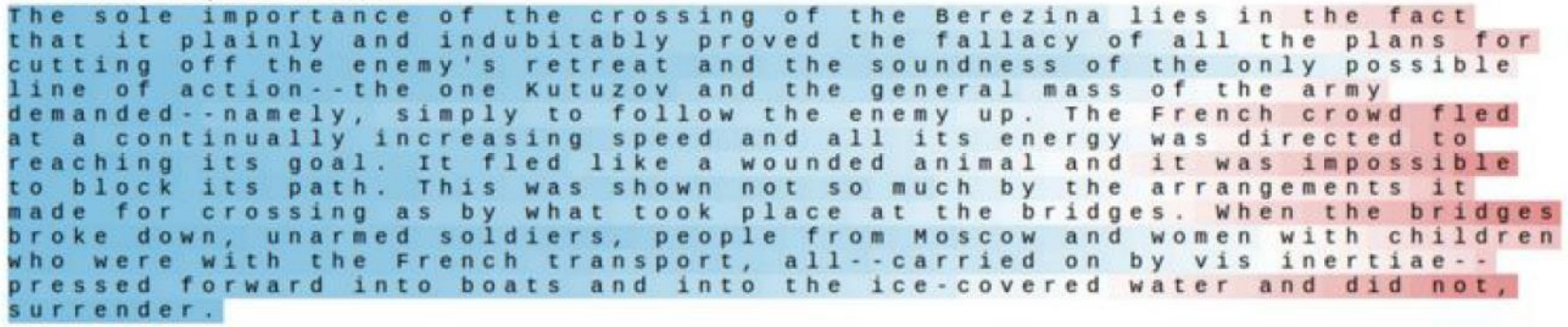
Searching for interpretable cells



quote detection cell

Searching for interpretable cells

Cell sensitive to position in line:



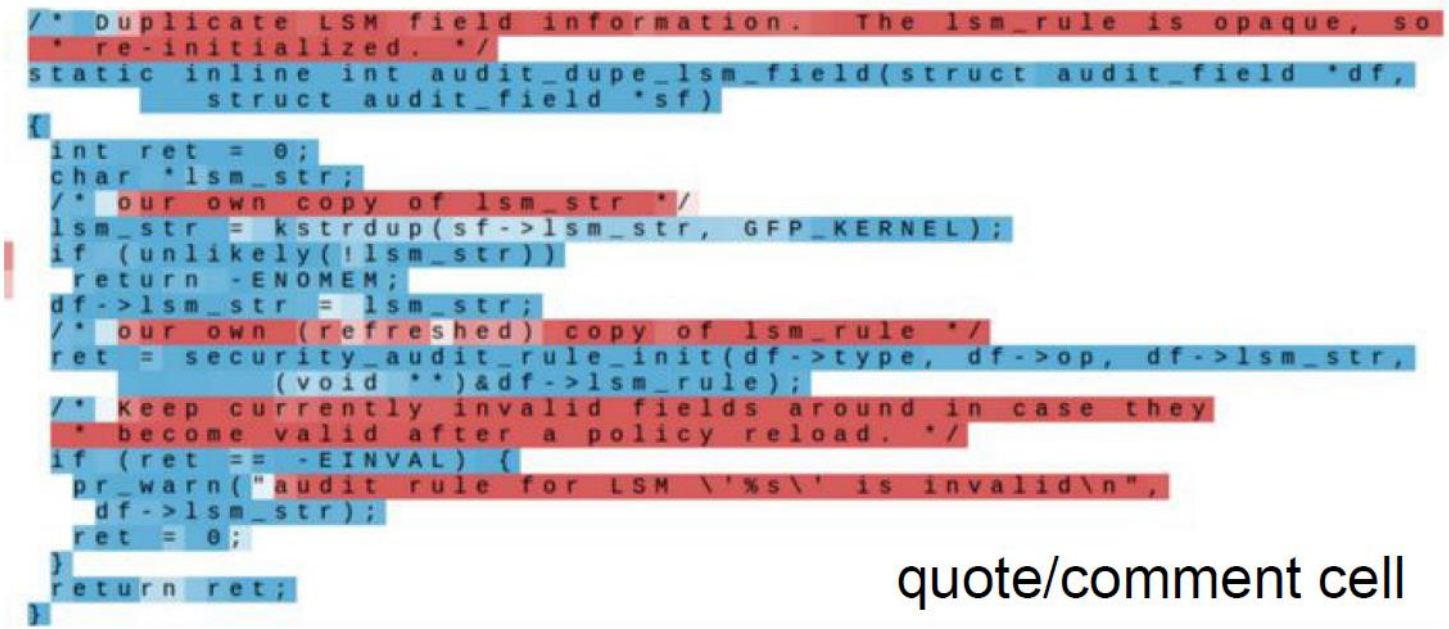
line length tracking cell

Searching for interpretable cells

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
        siginfo_t *info)
{
  int sig = next_signal(pending, mask);
  if (sig) {
    if (current->notifier) {
      if (sigismember(current->notifier_mask, sig)) {
        if (!(current->notifier)(current->notifier_data)) {
          clear_thread_flag(TIF_SIGPENDING);
          return 0;
        }
      }
    }
    collect_signal(sig, pending, info);
  }
  return sig;
}
```

if statement cell

Searching for interpretable cells



WHAT HAVE THE RNN LEARNED?

Probability Model

- Given example sequences $\{X_i\}$, let us find a probabilistic model $p(X)$ that maximizes $\prod_i p(X_i)$.

- X_i is a sequence $X_i = (x_1, x_2, \cdots, x_T)$
- For example,
  - $x_1 = h, x_2 =$e, $x_3 = l, x_4 = l, x_5 = o.$
  - $x_1 = "I", x_2 = "am", x_3 = "a", x_4 = "student".$

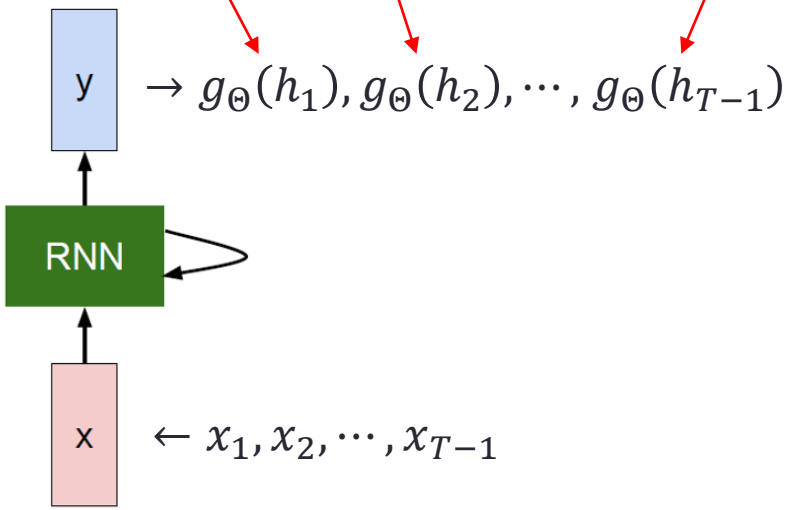
- For the model learning, we have to specify
  - Parametric family (model selection)
  - Learning criterion
  - Parameter learning method

Our choice

- Parametric family: RNN
  - Because we can decompose $p(X) = p(x_1, x_2, \ldots, x_T)$ into

$$p(x_1, x_2, \ldots, x_T) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \cdots p(x_T|x_1, \ldots, x_{T-1})$$

  - and RNN can handle this situation well.



Our choice

- Parametric family: RNN
  - $g_\Theta(h_{t-1})$ is a parametric model of $p(x_t|x_1, x_2, \cdots, x_{t-1})$.
  - If we have an N-word dictionary, we can represent $p(x_t|x_1, x_2, \cdots, x_{t-1})$ with a N-dimensional vector (that sums to a unity and non-negative).

$g_\Theta(h_{t-1}) =$

$p(x_t = 1^{\text{st}}$ word in the dictionary $|x_1, x_2, \cdots, x_{t-1})$
$p\left(x_t = 2^{\text{nd}}\right.$ word in the dictionary $\left.|x_1, x_2, \cdots, x_{t-1}\right)$
$p\left(x_t = 3^{\text{rd}}\right.$ word in the dictionary $\left.|x_1, x_2, \cdots, x_{t-1}\right)$
$p\left(x_t = 4^{\text{th}}\right.$ word in the dictionary $\left.|x_1, x_2, \cdots, x_{t-1}\right)$
…
…
…
$p\left(x_t = \text{N}^{\text{th}}\right.$word in the dictionary $\left.|x_1, x_2, \cdots, x_{t-1}\right)$

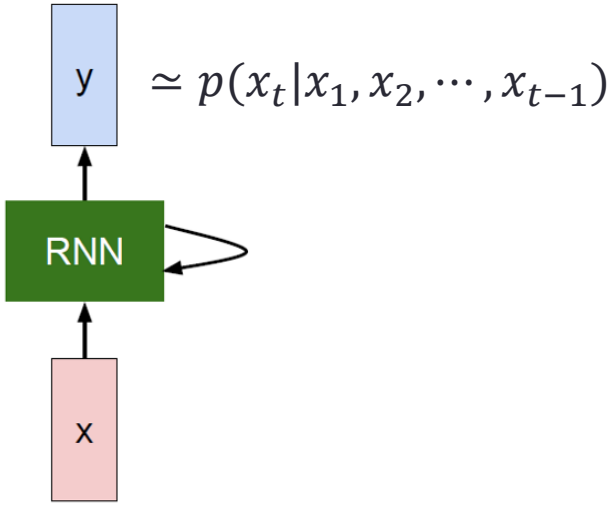
Our choice

- Learning criterion: log likelihood/cross entropy
  - $\log p(x_1, x_2, \cdots, x_T) = \sum_t \log p(x_t | x_1, x_2, \cdots, x_{t-1})$
  - $\sum_t \log p(x_t | x_1, x_2, \cdots, x_{t-1}) \rightarrow \sum_t g_\Theta(h_{t-1})^\top \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ 0 \end{bmatrix}$ →cross entropy

    x_t's element is 1.
- Learning method: SGD

Summary

- The above RNN model learns the (conditional) probability model of a sequence by maximizing the log likelihood of training sequences.



APPLICATION: IMAGE CAPTIONING

Image captioning

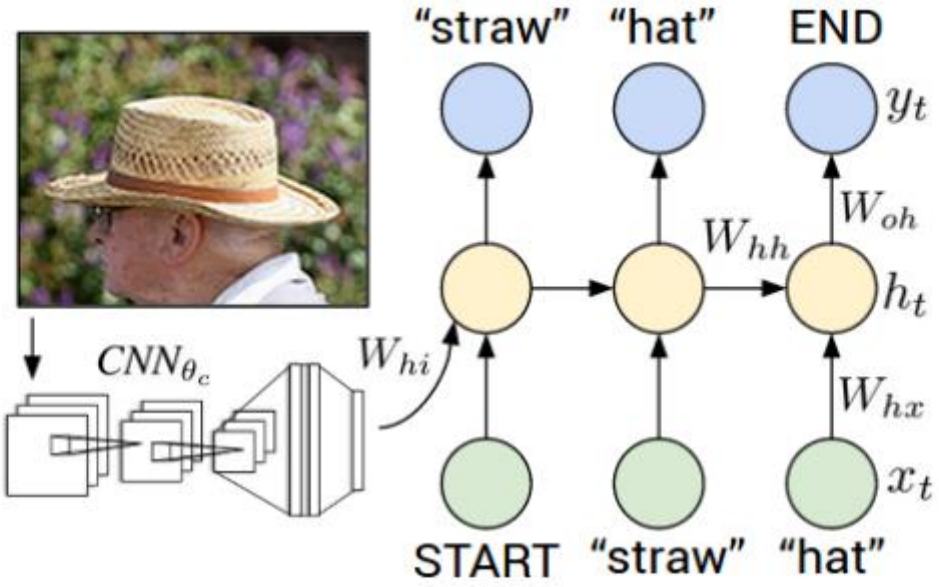
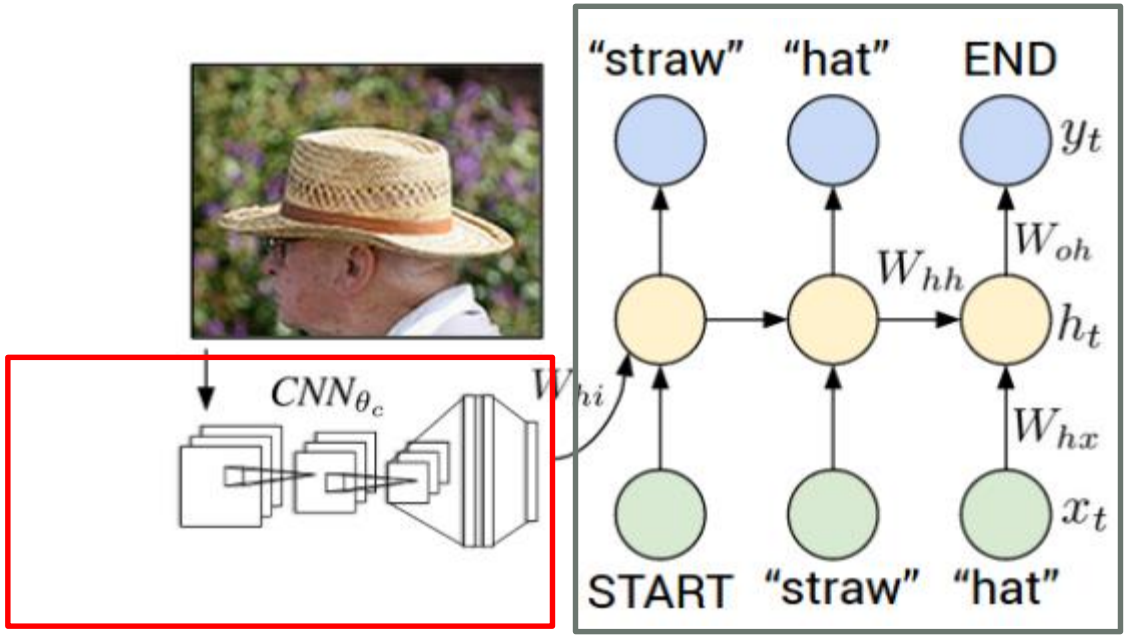


Diagram of our multimodal Recurrent Neural Network generative model. The RNN takes a word, the context from previous time steps and defines a distribution over the next word in the sentence. The RNN is conditioned on the image information at the first time step. START and END are special tokens.


CNN_{θ_c}
W_{hi}
"straw"
"hat"
END
y_t
W_{oh}
W_{hh}
h_t
W_{hx}
x_t
START
"straw"
"hat"

CNN RNN

test image


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
FC-1000
softmax
test image

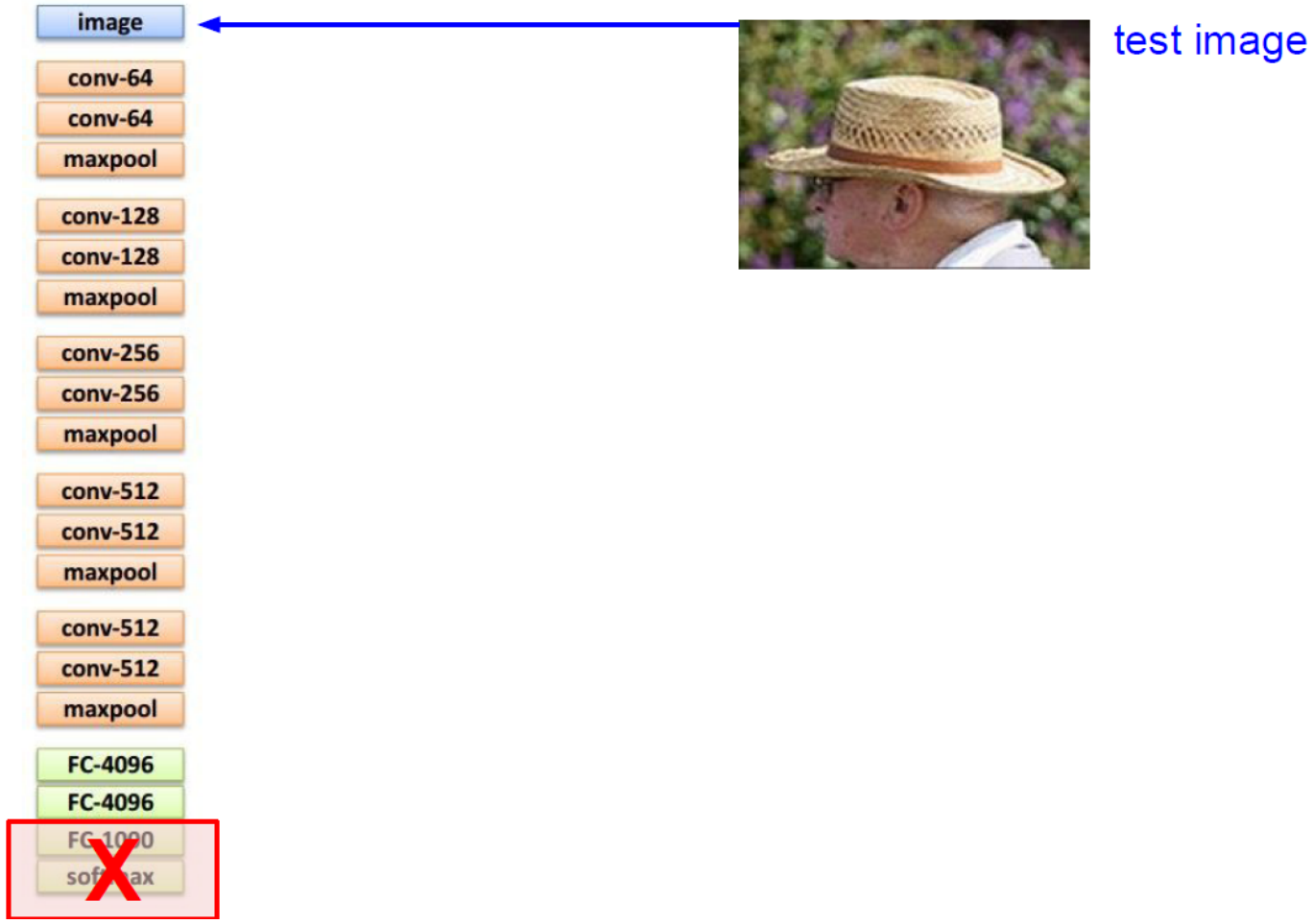


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
FC-1000
softmax
X
test image

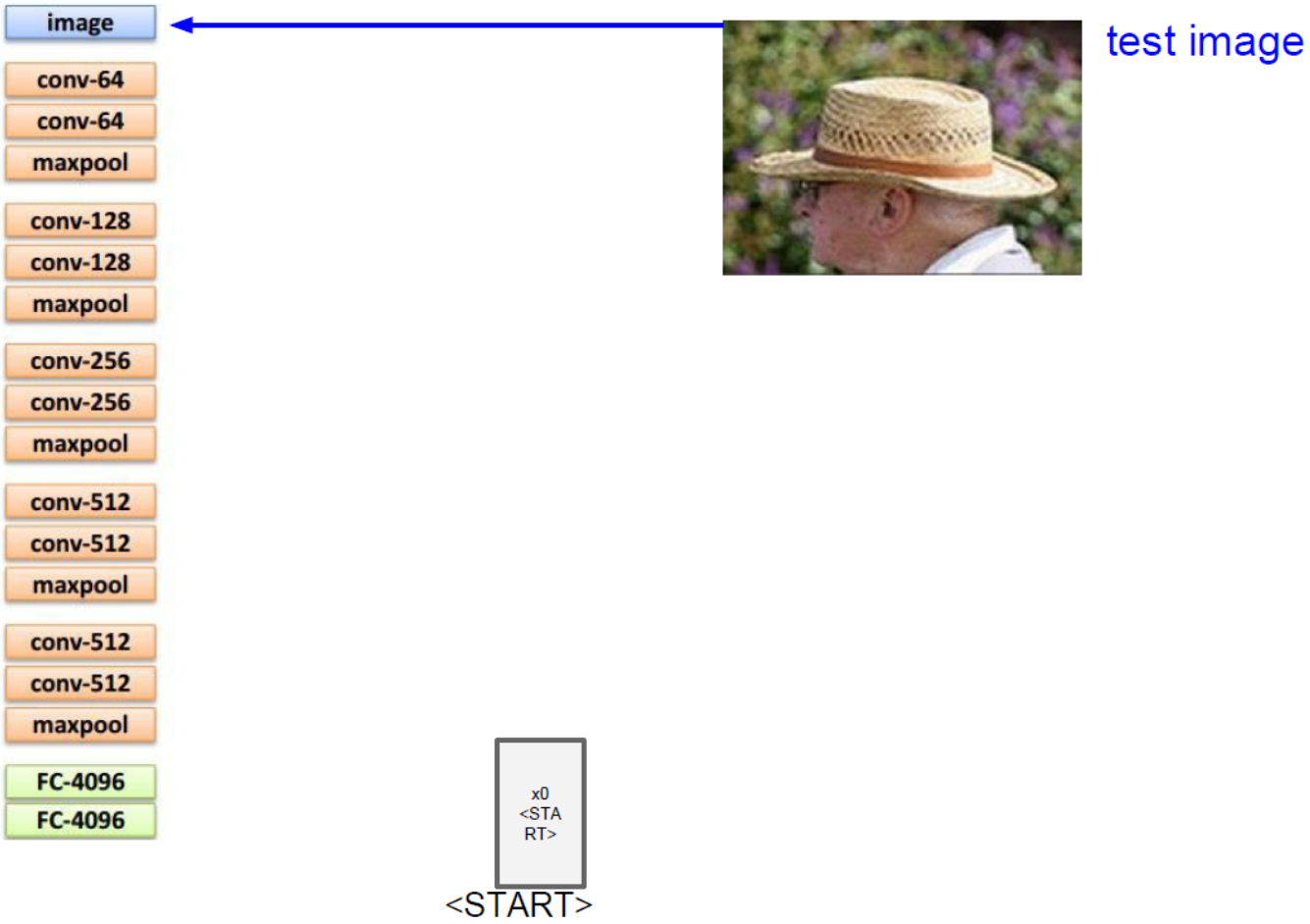
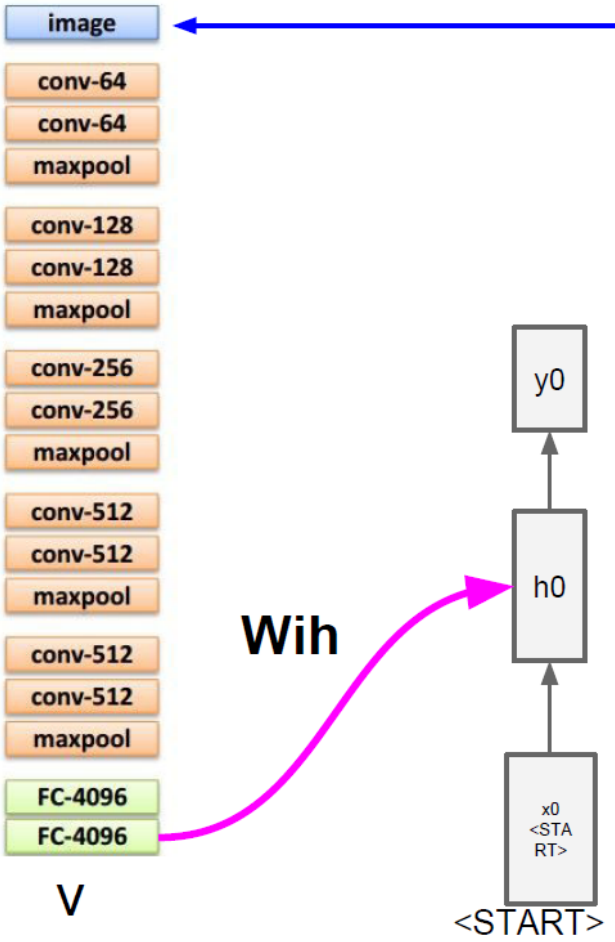


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
x0
<STA
RT>
<START>



test image

before:

h = tanh(Wxh * x + Whh * h)

now:

h = tanh(Wxh * x + Whh * h + **Wih** * v)

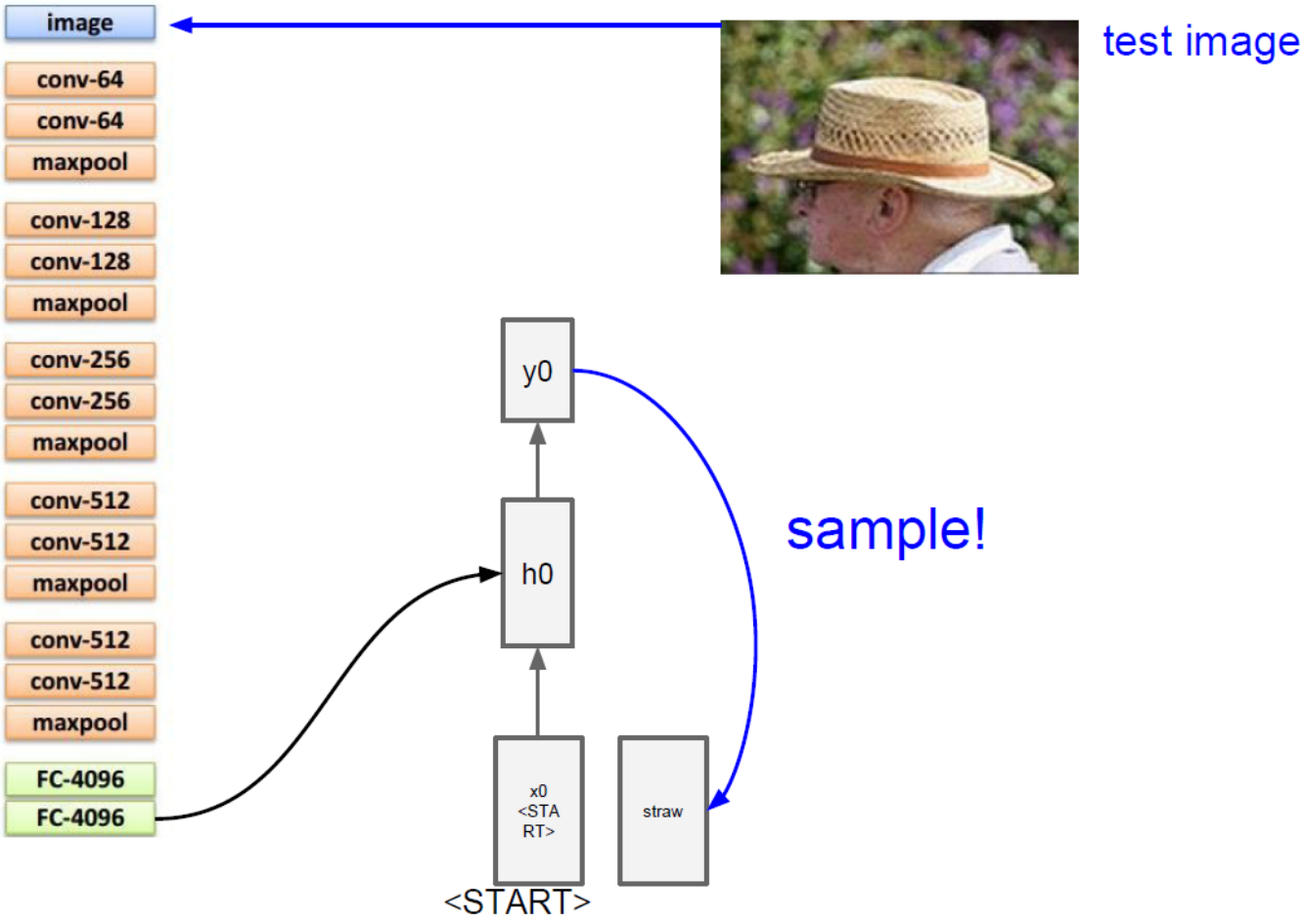


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
h0
x0
<STA
RT>
straw
<START>
sample!

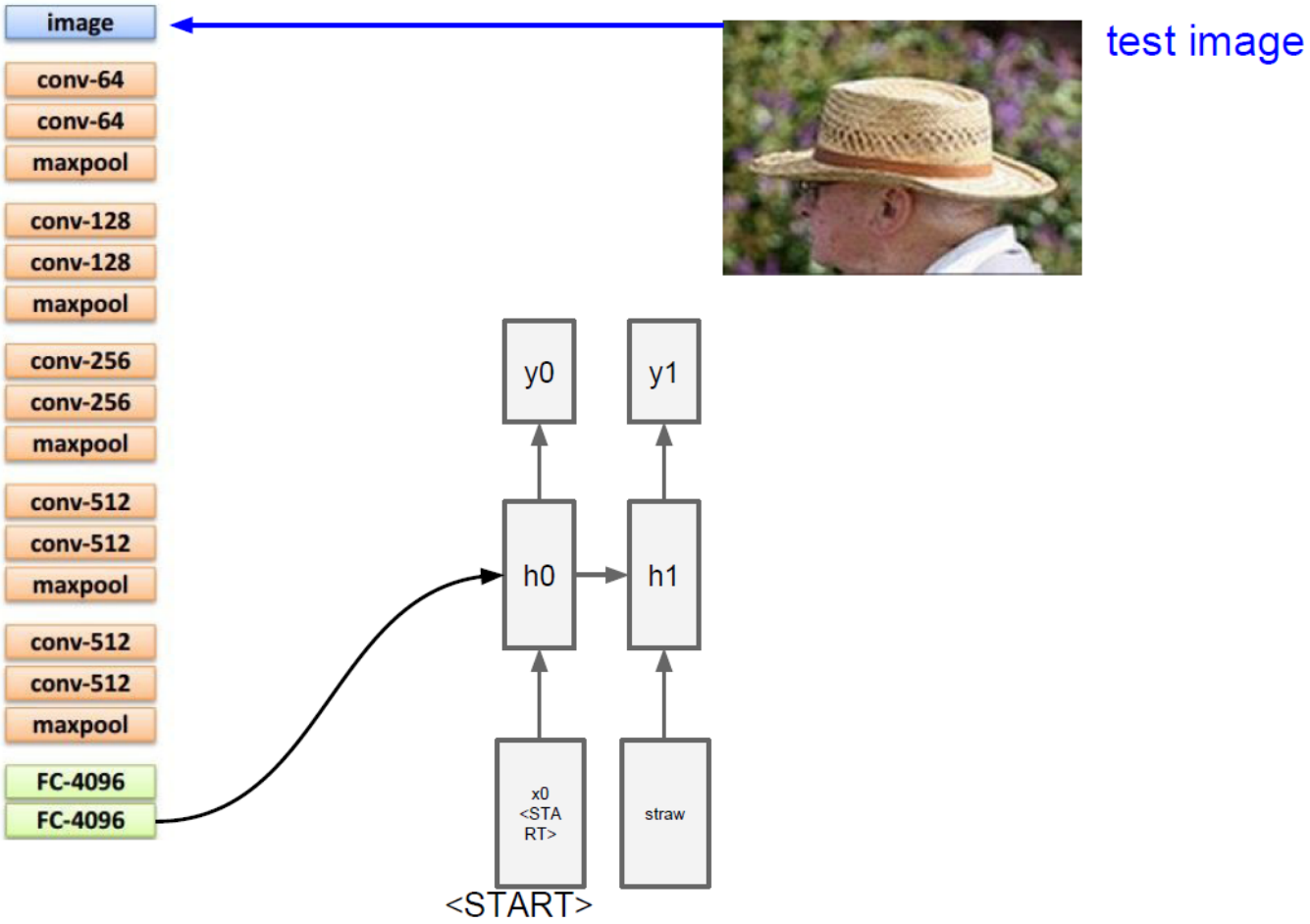


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
h0
h1
x0
<STA
RT>
straw
<START>

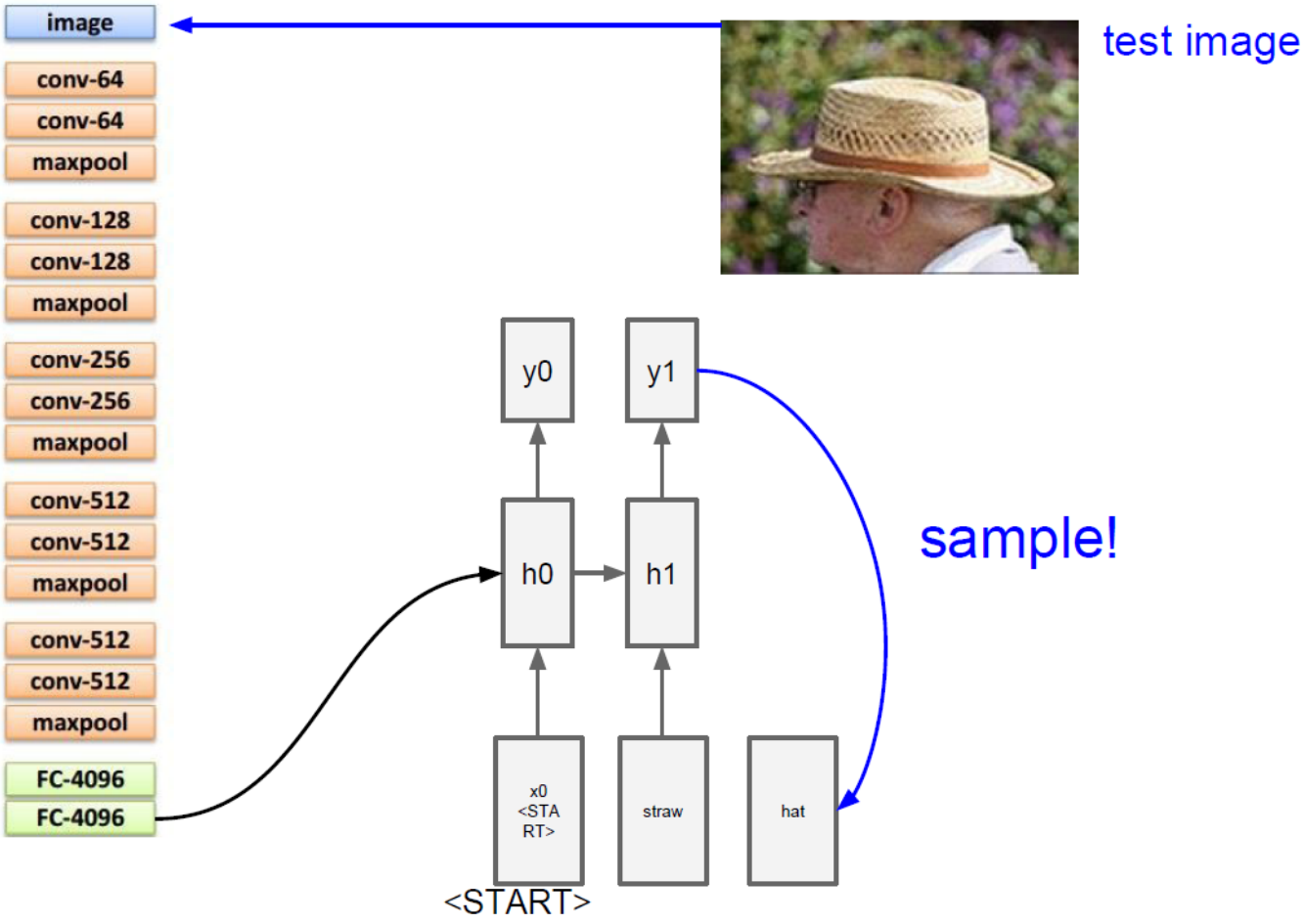


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
h0
h1
sample!
x0
<STA
RT>
straw
hat
<START>

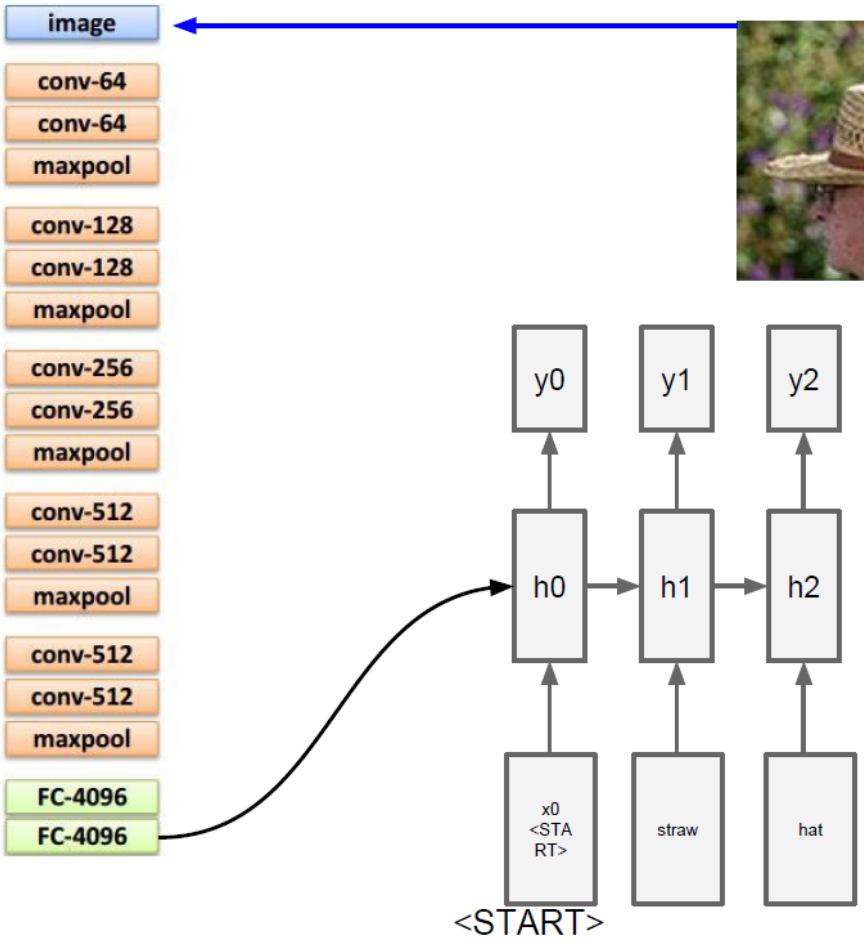


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
y2
h0
h1
h2
x0 <START>
straw
hat
<START>

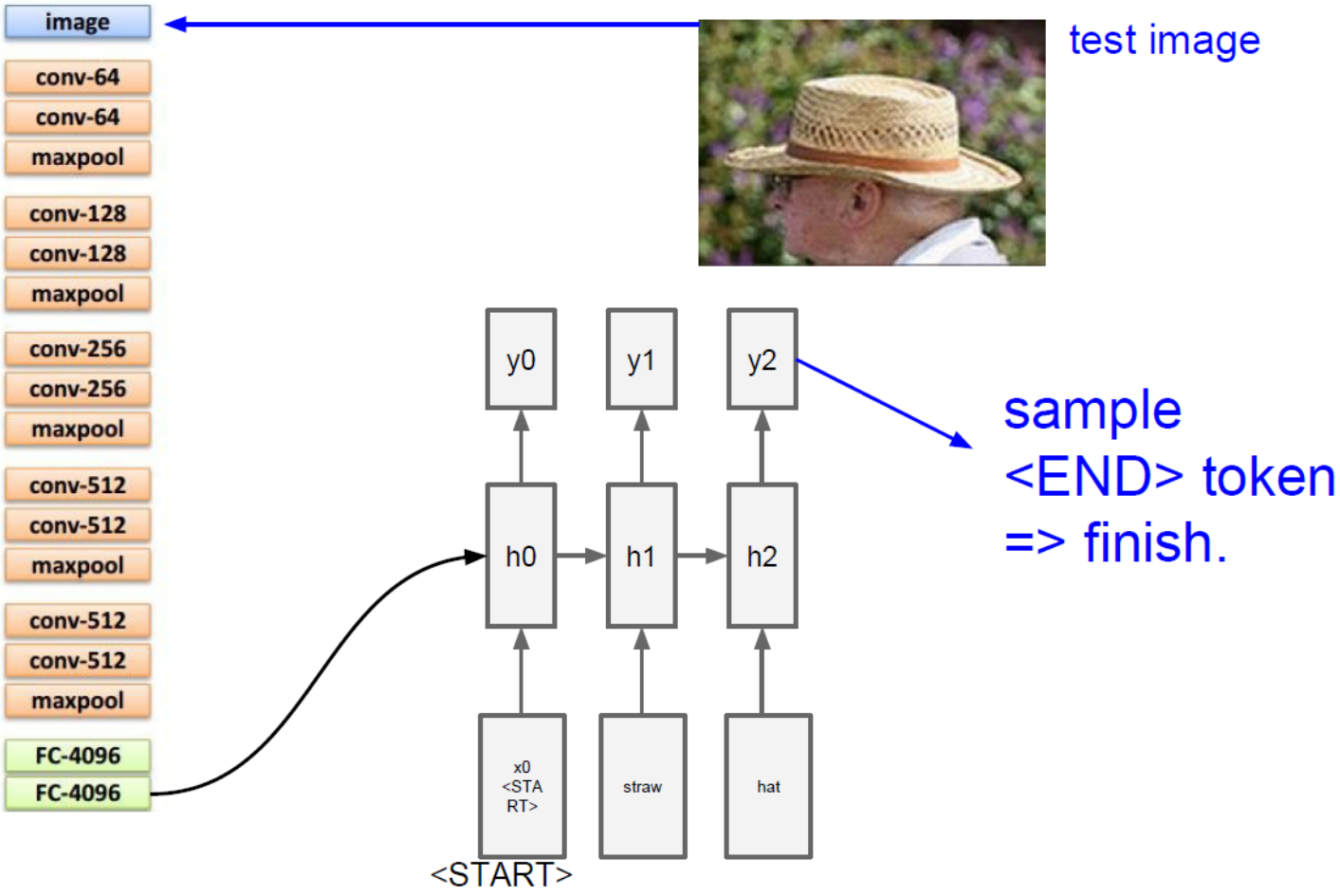


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
y2
h0
h1
h2
x0 <START>
straw
hat
<START>
sample <END> token => finish.

Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Backward flow of gradients in RNN can explode or vanish.
- Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)

BACKUPS

Backpropagation

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t + b_h\right)$$

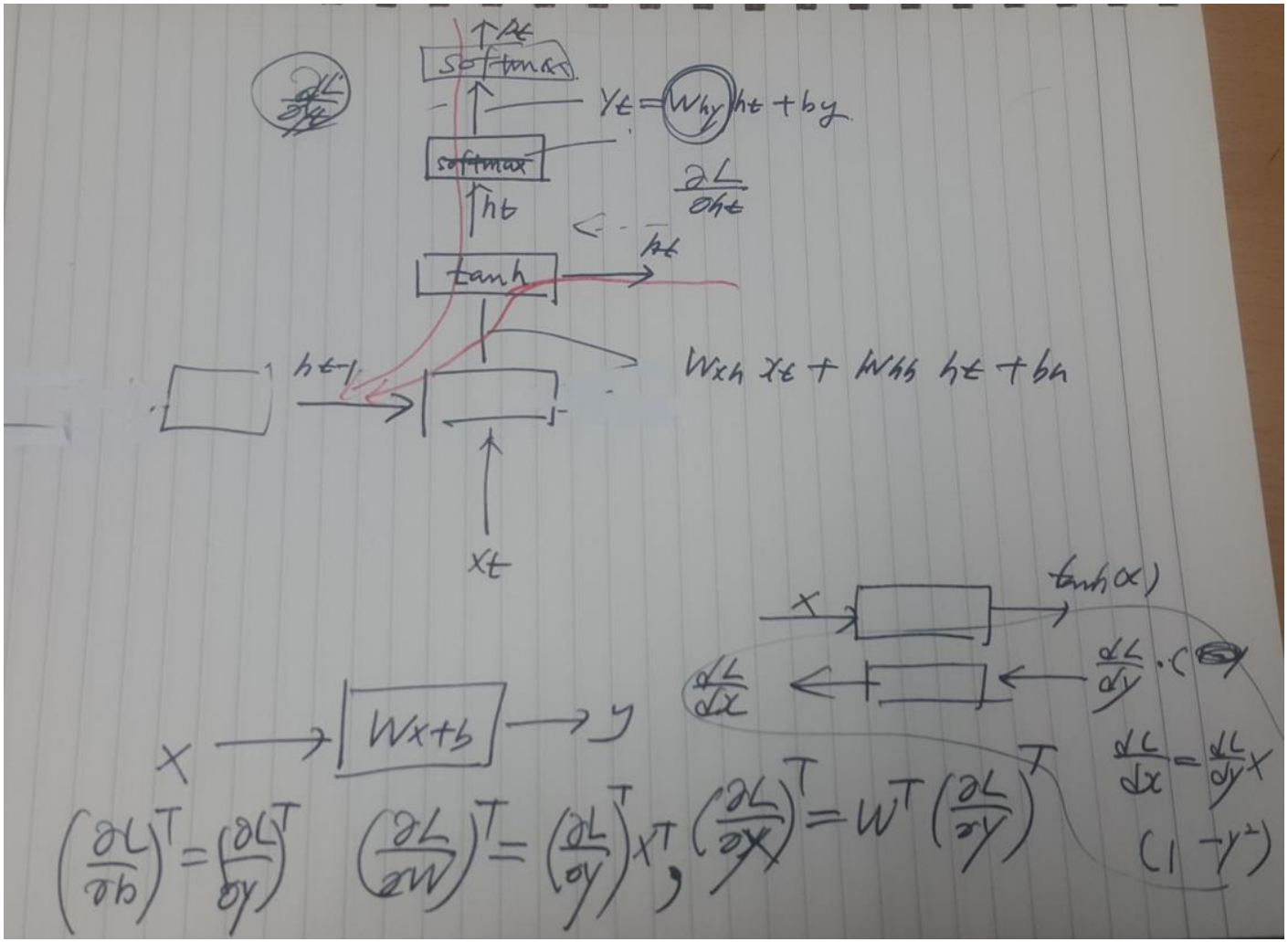
$$y_t = W_{hy}h_t + b_y$$

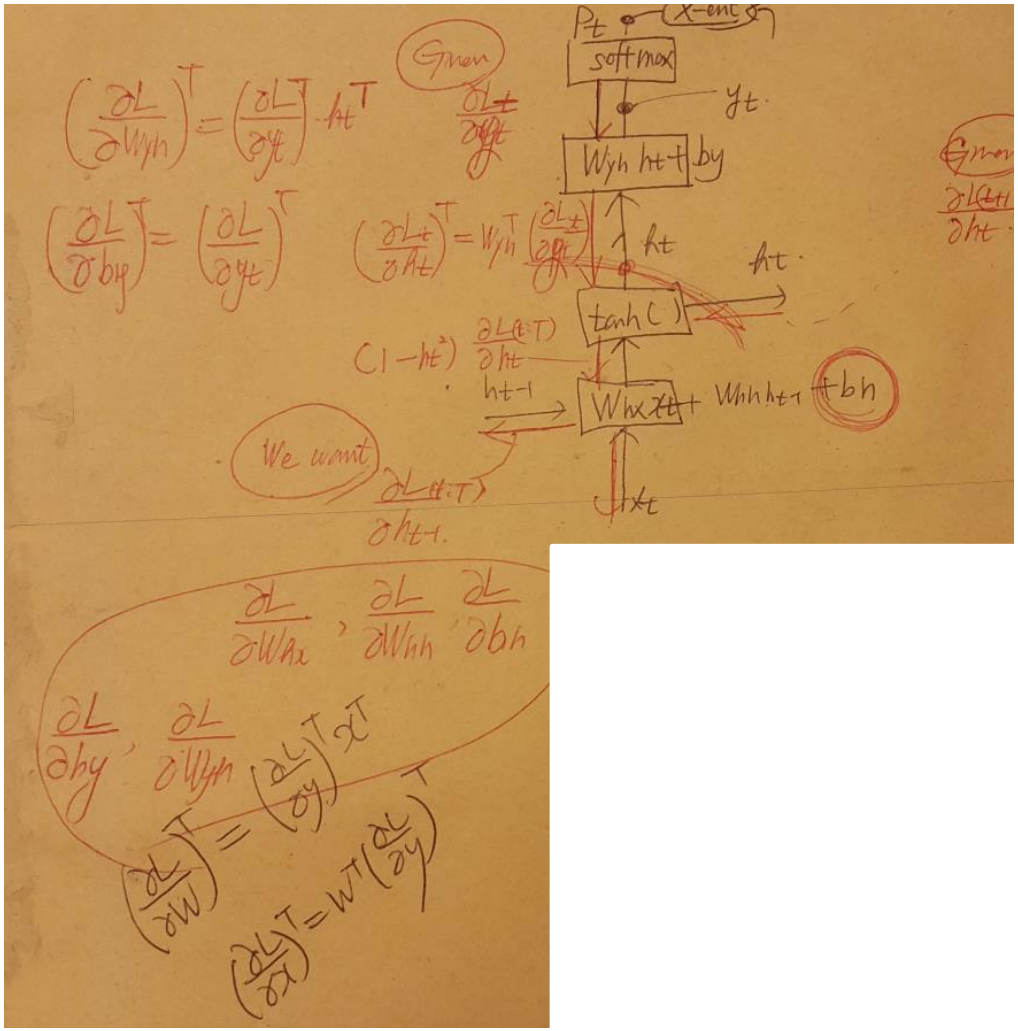
$$p_t = \text{softmax}(y_t)$$

$$l \leftarrow l - \log\left(p_t\right)^{\top}[0 \ldots . 1 \ldots .0]$$

- $$z = -\log\left(\frac{e^{y^{(2)}}}{e^{y^{(1)}} + e^{y^{(2)}} + e^{y^{(3)}}}\right) = -y^{(2)} + \log\left(e^{y^{(1)}} + e^{y^{(2)}} + e^{y^{(3)}}\right)$$

$$\frac{d(\tanh f(x))}{dx} = (1 - \tanh^2 f(x))f'(x)$$


softmax
softmax
tanh
$y_t = W_{hy} h_t + b_y$
$W_{xh} x_t + W_{hh} h_{t-1} + b_h$
h_{t-1}
x_t
$\tanh(x)$
$W x + b$
$\left(\frac{\partial L}{\partial b}\right)^T = \left(\frac{\partial L}{\partial y}\right)^T$
$\left(\frac{\partial L}{\partial W}\right)^T = \left(\frac{\partial L}{\partial y}\right)^T x^T$,
$\left(\frac{\partial L}{\partial x}\right)^T = W^T \left(\frac{\partial L}{\partial y}\right)^T$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} (1 - y^2)$


Pt
x-enc
softmax
yt
Wyh ht + by
ht
ht
tanh( )
(1 − ht²)
ht-1
Whx xt + Whh ht-1 + bh
xt
We want
Given
Given

Backpropagation

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t + b_h\right)$$
$$y_t = W_{hy}h_t + b_y$$
$$p_t = \text{softmax}(y_t)$$
$$l \leftarrow l - \log\left(p_t\right)^{\top}[0 \ldots . 1 \ldots .0]$$

- For example,

$$\frac{\partial z}{\partial y} = \begin{bmatrix} p^{(1)} \\ -1 + p^{(2)} \\ p^{(3)} \end{bmatrix}$$

$$\frac{\partial z}{\partial W_{hy}} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial W_{hy}} = \frac{\partial z}{\partial y}h_t^{\top}$$

$$\frac{\partial z}{\partial b_y} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial b_y} = \frac{\partial z}{\partial y}$$

Backpropagation

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = W_{hy}h_t + b_y$$
$$p_t = \text{softmax}(y_t)$$
$$l \leftarrow l - \log(p_t)^\top [0 \ldots 1 \ldots 0]$$

- For example,

$$\frac{\partial z}{\partial h} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial h} = \frac{\partial z}{\partial y}W_{hy}^\top$$

$$\frac{d(\tanh f(x))}{dx} = (1 - \tanh^2 f(x))f'(x)$$