RNNs (part 2)

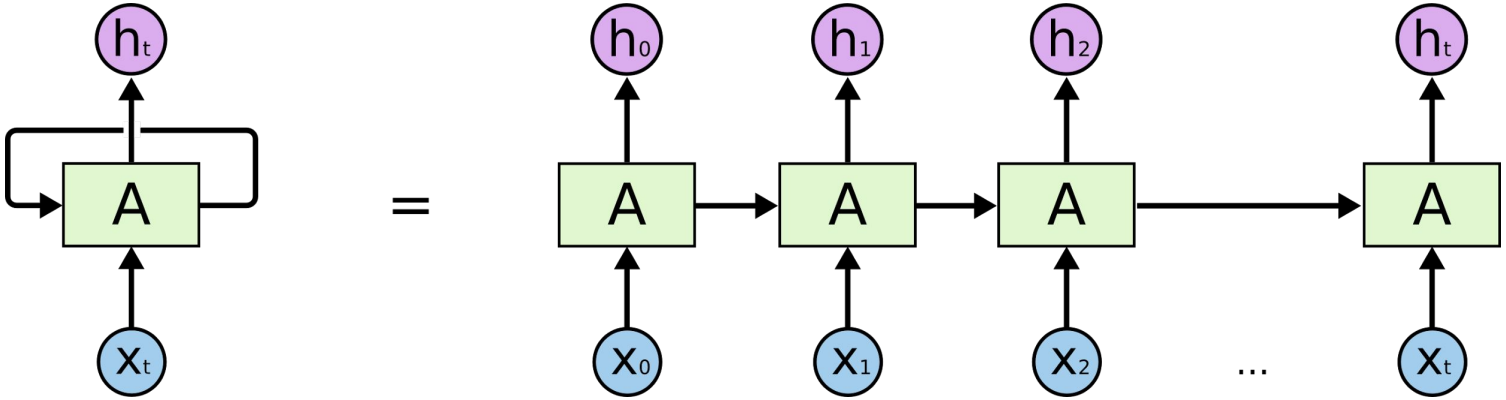
RNN

- Along with CNNs, RNNs are the most important class of models in DL
- Applicable in
  speech
  recognition(https://research.googleblog.com/2015/09/google-voice-search-faster-and-more.html)
  machine
  translation(https://research.googleblog.com/2016/09/a-neural-network-for-machine.html)
  and many other task especially in NLP

Overview

- Basic RNN recap
- Language modelling example - character level RNN
- LSTM recap
- RNN extensions - bidirectional RNN, multilayer RNNs, RvNN
- Using RNNs with Tensorflow
- Case studies
- Quora Kaggle competition
- Lab instructions

Basic RNN



$$RNN : h_{t-1}, x_t \rightarrow h_t$$
$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

- W_h, W_x are the same across all timesteps

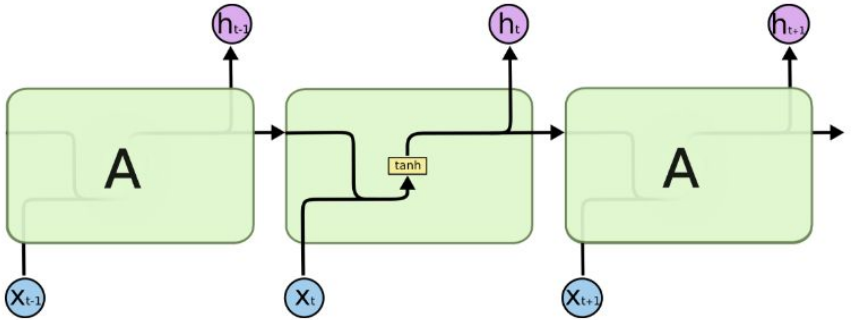
Basic RNN implementation

```
def build_model(self, x, steps_n, hidden_n, input_n):
    # input_n dimension of the input at each timestep
    # x has shape mb_size X steps_n X input_n
    with tf.variable_scope('rnn'):
        self.W = tf.Variable(name='W', dtype=tf.float32,
                             initial_value=normal(scale=0.1, size=(input_n + hidden_n, hidden_n))
        bias_value = np.zeros((hidden_n,), dtype='float32')
        self.bias = tf.Variable(name='b', initial_value=bias_value, dtype=tf.float32)
        self.h_0 = tf.Variable(name='h_0', initial_value=np.zeros((1, hidden_n)), dtype=tf.float32)

    h = {}
    h[-1] = tf.tile(self.h_0, [tf.shape(x)[0], 1])

    for t in xrange(steps_n):
        x_t = x[:, t, :]
        input = tf.concat([x_t, h[t - 1]], axis=1)
        h[t] = tf.tanh(tf.matmul(input, self.W) + self.bias)
```

$$RNN : h_{t-1}, x_t \rightarrow h_t$$

$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

$$\tanh(W \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b)$$

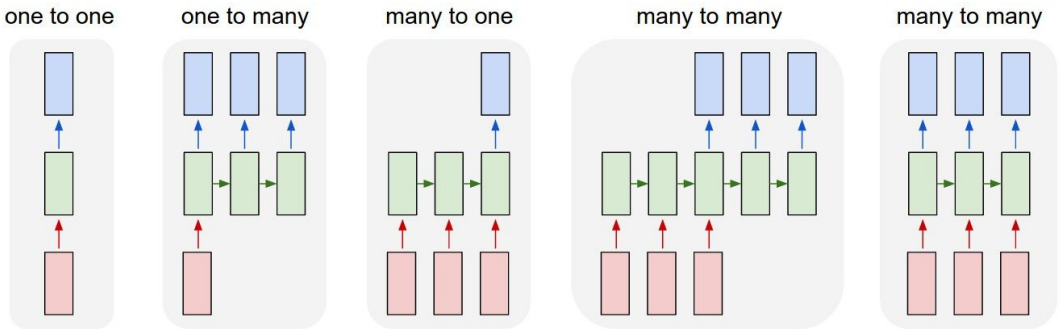


The repeating module in a standard RNN contains a single layer.

What to do with h_ts

- We have h_t for each t how do we use it?
- In “many to one” case we could just take h_last, and process it further to arrive at our final prediction

- In “many to many” case, we would have some other net take h_t and produce y_t, we would apply same net at each timestep
- other cases possible



Basic RNN

There are few important technical issues with this code

- it requires steps_n to be constant, at the time of graph creation
- sometimes we would like different number of steps for each example in a minibatch
- what to do with very long inputs?

Language modelling

Train RNN to model

$$p(w_{n+1}|[w_1, w_2, w_3, \ldots, w_n])$$

$$p(\text{"}you\text{"}|[\text{"}I\text{"}, \text{"}like\text{"}]) > p(\text{"}like\text{"}|[\text{"}I\text{"}, \text{"}like\text{"}])$$

- A lot of training data
- We can use such model to choose more probable sentences, ex. in speech recognition system, translation systems
- Or we could let it run and generate some text

Language modelling

- Give our RNN sequence of words and ask for the probability distribution of the next word

$$p(\text{"}you\text{"}|[\text{"}I\text{"},\text{"}like\text{"}]) > p(\text{"}like\text{"}|[\text{"}I\text{"},\text{"}like\text{"}])$$

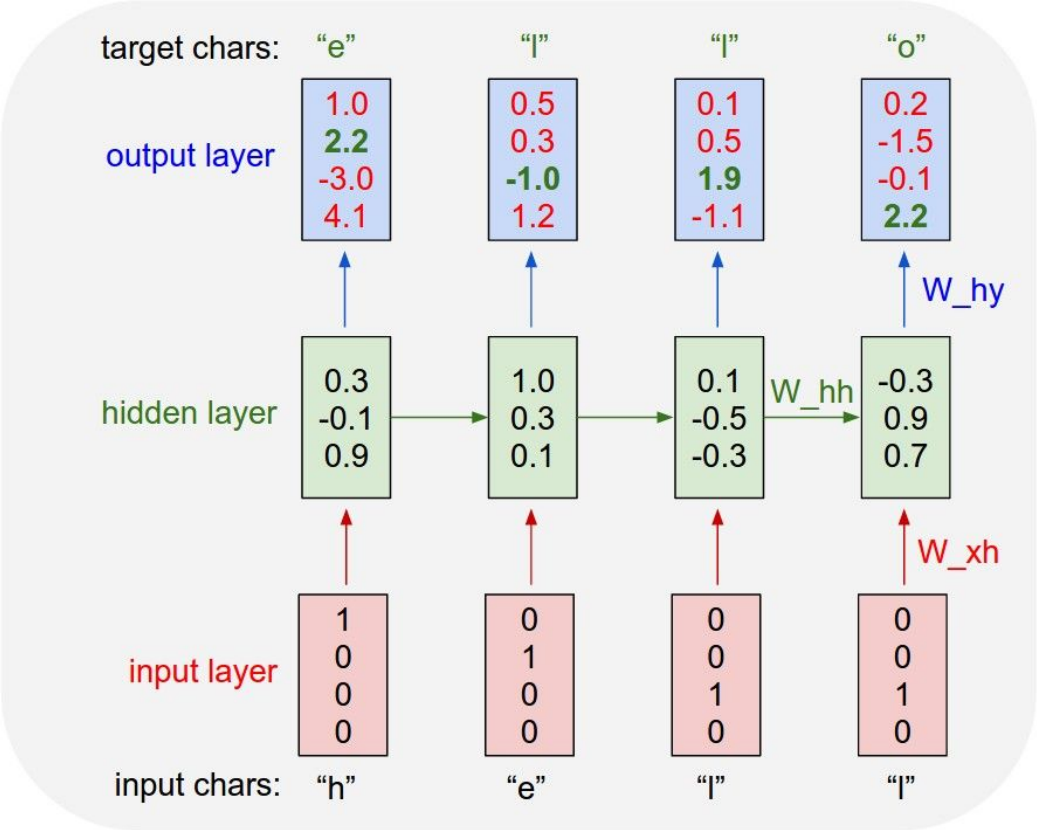
- Give our RNN sequence of characters and ask for the probability distribution of the next character

$$p(\text{"}o\text{"}|[\text{"}h\text{"},\text{"}e\text{"},\text{"}l\text{"},\text{"}l\text{"}]) > p(\text{"}w\text{"}|[\text{"}h\text{"},\text{"}e\text{"},\text{"}l\text{"},\text{"}l\text{"}])$$

Language modelling char rnn

$$RNN : h_{t-1}, x_t \rightarrow h_t$$
$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

Training sequence: **“hello”**



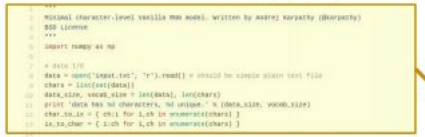
hidden state size = 3

min-char-rnn.py gist: 112 lines of Python

```python
"""
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) }

# 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

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

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
```

(https://gist.github.com/karpathy/d4dee566867f8291f086)

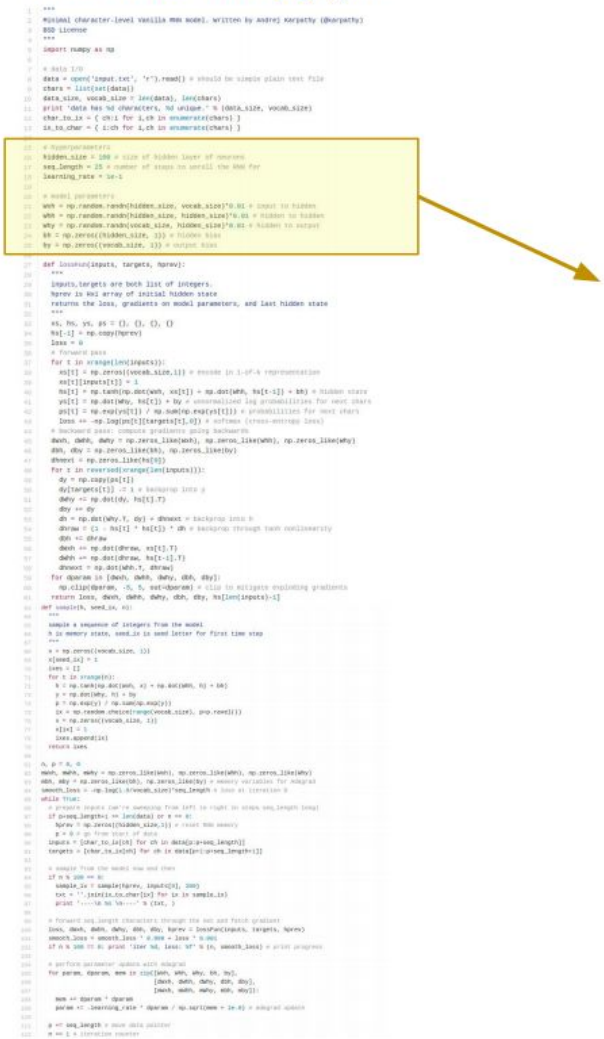
min-char-rnn.py gist



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) }
```

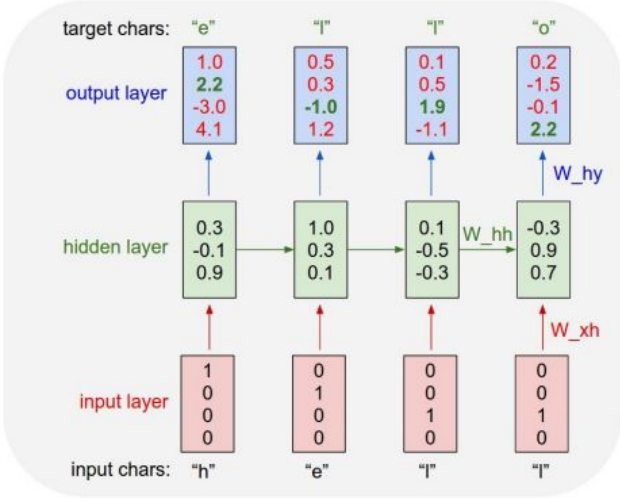
min-char-rnn.py gist



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:



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
```

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
```

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
```

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
```

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
```

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]
```

min-char-rnn.py gist

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

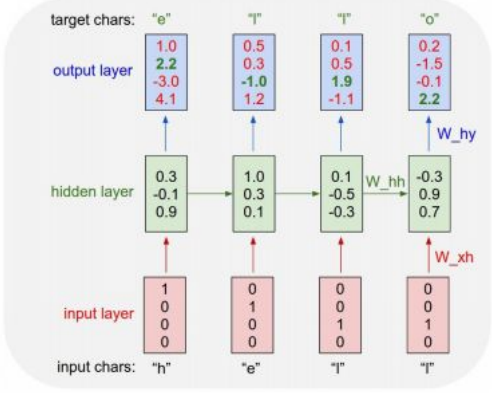
$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$
$$y_t = W_{hy} h_t$$

Softmax classifier

min-char-rnn.py gist

```python
# backward pass: compute gradients going backwards
dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
dbh, dby = np.zeros_like(bh), np.zeros_like(by)
dhnext = np.zeros_like(hs[0])
for t in reversed(xrange(len(inputs))):
  dy = np.copy(ps[t])
  dy[targets[t]] -= 1 # backprop into y
  dWhy += np.dot(dy, hs[t].T)
  dby += dy
  dh = np.dot(Why.T, dy) + dhnext # backprop into h
  dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
  dbh += dhraw
  dWxh += np.dot(dhraw, xs[t].T)
  dWhh += np.dot(dhraw, hs[t-1].T)
  dhnext = np.dot(Whh.T, dhraw)
for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
  np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

recall:



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
```

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.
```

How to improve this code?

- GPU
- Add minibatches
- LSTM
- More powerful model(multilayer LSTM)
- Regularization (Recurrent Neural Network Regularization, https://arxiv.org/abs/1409.2329)
- https://github.com/karpathy/char-rnn

Recap architectures LSTM - Long short term memory

$$\text{LSTM}: x_t, h_{t-1}, c_{t-1} \rightarrow h_t, c_t$$

$$i = sigm(W_i \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_i)$$

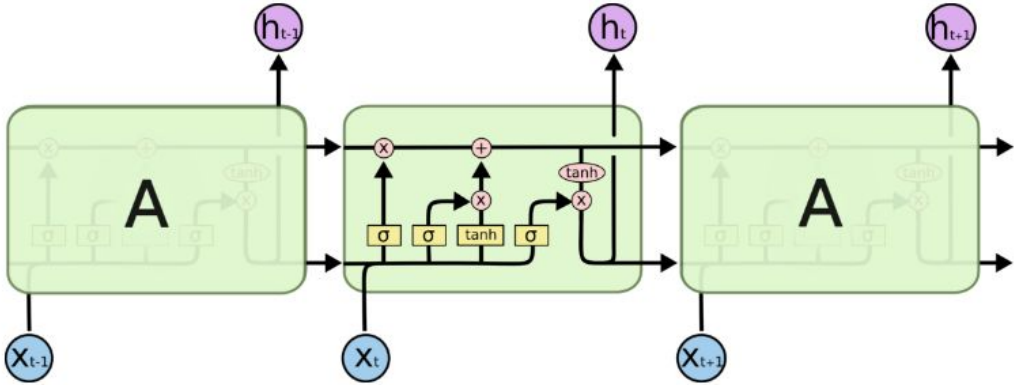
$$f = sigm(W_f \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_f)$$

$$o = sigm(W_o \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_o)$$

$$g = tanh(W_g \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_g)$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$



The repeating module in an LSTM contains four interacting layers.

LSTM implementation

```
def build_model(self, x, steps_n, hidden_n, input_n, forget_bias=1.0):
        # input_n dimension of the input at each timestep
        # x has shape mb_size X steps_n X input_n
         with tf.variable_scope('lstm'):
             self.W = tf.Variable(name='W', dtype=tf.float32,
                                  initial_value=normal(scale=0.1, size=(input_n + hidden_n, 4 * hidden_n)))
             bias_value = np.zeros((4 * hidden_n,), dtype='float32')
             bias_value[1 * hidden_n: 2 * hidden_n] += forget_bias

             self.bias = tf.Variable(name='b', initial_value=bias_value, dtype=tf.float32)

             self.c_0 = tf.Variable(name='c_0', initial_value=np.zeros((1, hidden_n)), dtype=tf.float32)
             self.h_0 = tf.Variable(name='h_0', initial_value=np.zeros((1, hidden_n)), dtype=tf.float32)
```

LSTM implementation

```
c, h = {}, {}
c[-1] = tf.tile(self.c_0, [tf.shape(x)[0], 1])
h[-1] = tf.tile(self.h_0, [tf.shape(x)[0], 1])
```

```
for t in xrange(steps_n):
    x_t = x[:, t, :]
    input = tf.concat([x_t, h[t - 1]], axis=1)

    z = tf.matmul(input, self.W) + self.bias
    i = tf.sigmoid(z[:, 0 * hidden_n: 1 * hidden_n])
    f = tf.sigmoid(z[:, 1 * hidden_n: 2 * hidden_n])
    o = tf.sigmoid(z[:, 2 * hidden_n: 3 * hidden_n])
    g = tf.tanh(z[:, 3 * hidden_n: 4 * hidden_n])

    c[t] = f * c[t - 1] + i * g
    h[t] = o * tf.tanh(c[t])
```

$$\text{LSTM}: x_t, h_{t-1}, c_{t-1} \rightarrow h_t, c_t$$

$$i = sigm(W_i \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_i)$$

$$f = sigm(W_f \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_f)$$

$$o = sigm(W_o \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_o)$$

$$g = tanh(W_g \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_g)$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Bidirectional RNN

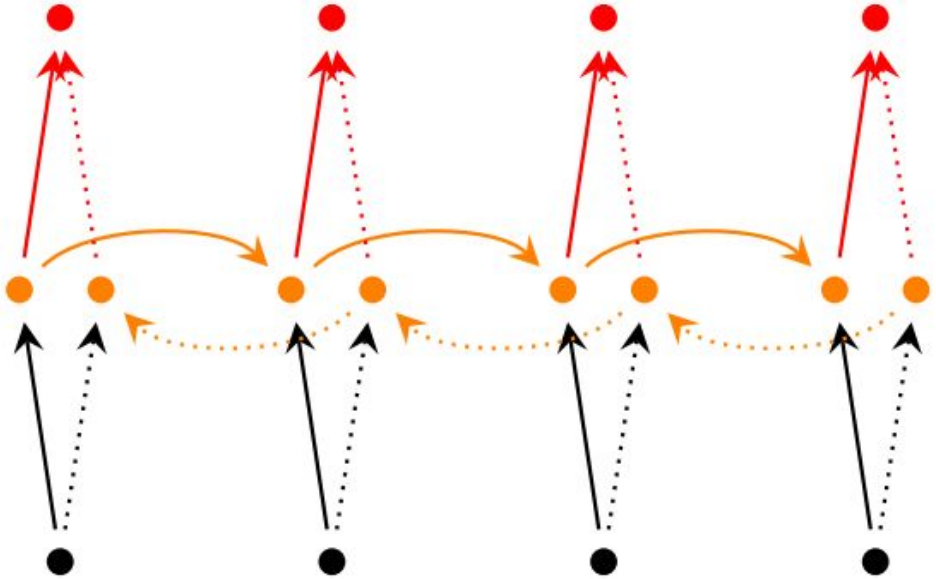
- Nothing special about them
- Just run one RNN from left to right
- The other one from right to left
- Concatenate their respective hidden states

$$h_{bidirect}[t] = concact(h_{forward}[t], h_{backward}[t])$$

- Or just take
  $h = concact(h_{forward}[T], h_{backward}[0])$

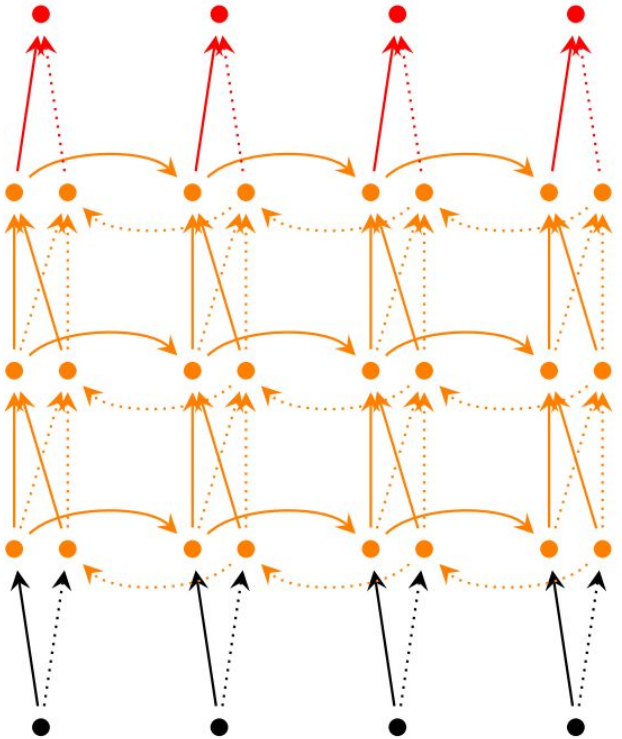
  when doing classification

- Used when you want h_t to depend not only on the “past”, but also the “future”



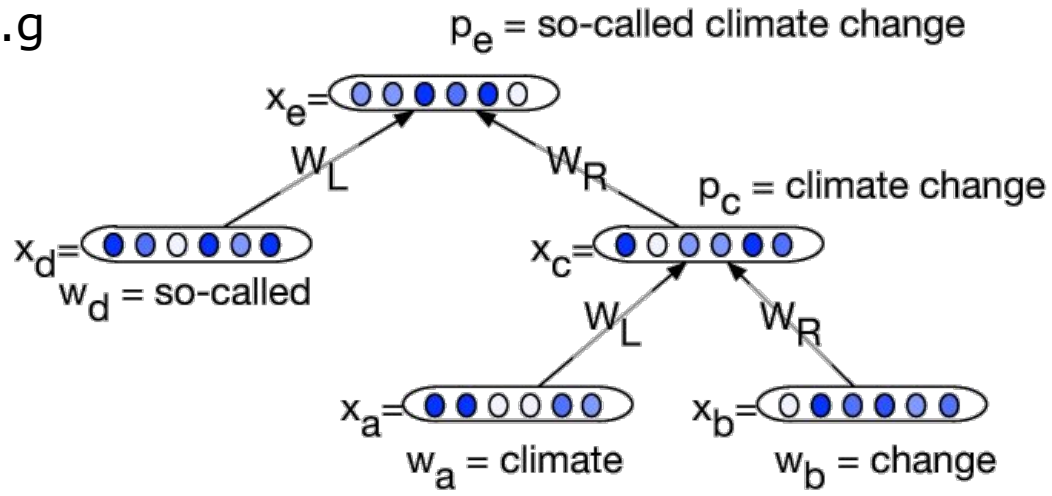
MultiLayer RNN

- Just run one RNN on the outputs of another one
- We can combine this with bidirectional RNN and make Deep Bidirectional RNN
- Or Deep Bidirectional LSTM



Recursive neural networks

- Works on a tree structure, e.g suppose you want to do do sentiment analysis and have parse tree of the sentence
- A RNN is just a special case of RvNN, a linear tree



Tensorflow RNN API

Using RNNs/LSTMs is very easy using tensorflow.

```
# Unstack to get a list of 'n_steps' tensors of shape (batch_size, n_input)
x = tf.unstack(x, n_steps, 1)

# Define a lstm cell with tensorflow
lstm_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)


# output, state = lstm_cell(x_t, state)

# Get lstm cell output
outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)
```

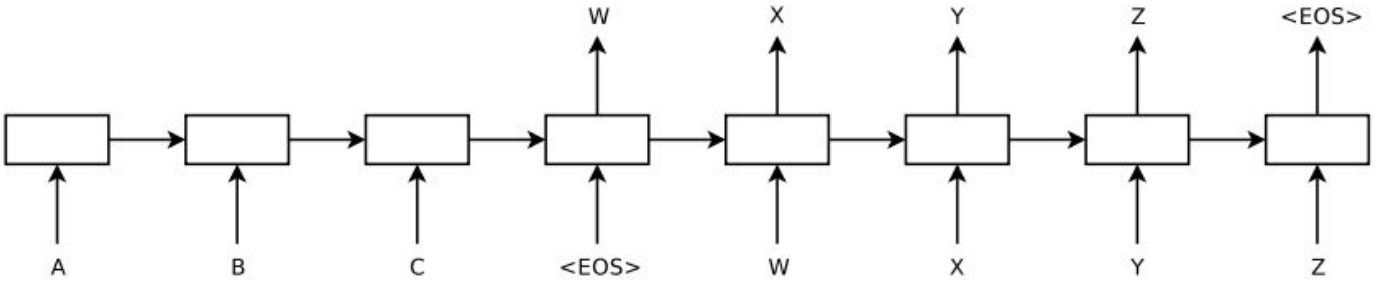
https://www.tensorflow.org/tutorials/recurrent

Case studies

- seq2seq
- Machine translation with attention
- Caption generation
- Neural Turing Machine

seq2seq

- We want to translate from English to French
- WMT'14 dataset, 12M sentences consisting of 348M French words and 304M English words
- Basic idea: train some RNN to take source sentence one word at a time(encoder), and other RNN to take this encoded state and produce one word of translation at a time.



Sequence to Sequence Learning with Neural Networks,
Ilya Sutskever et al.
https://arxiv.org/abs/1409.3215

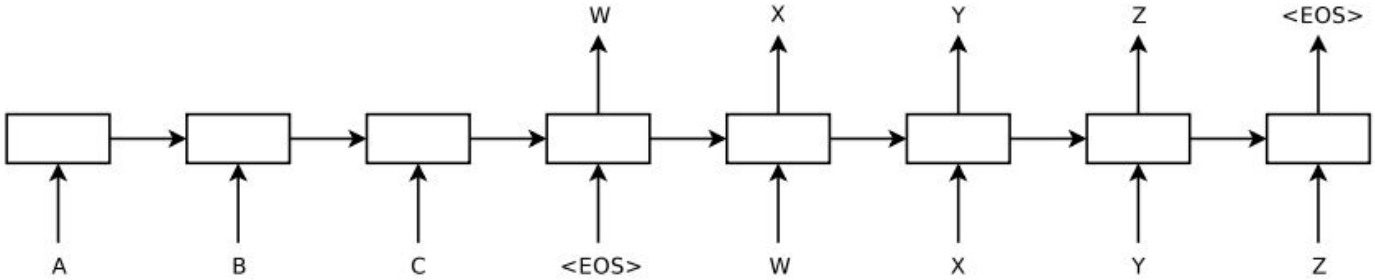
How to treat words? - Word embeddings

- We give each word in a vocabulary a d-dimensional real vector
- We learn those vectors jointly with our whole model
- The word embedding of a word can learn to represent important “things” about this word, e.g. if this is verb/noun?, is it positive/negative? and many other things that are not easily described in words
- Often we initialize these word embedding with some precomputed vectors, trained in unsupervised manner, like word2vec
- 1000 dimensional word embeddings used in this paper

Sequence to Sequence Learning with Neural Networks
https://arxiv.org/abs/1409.3215

seq2seq model details

- Words embeddings at input each timestep
- Two different LSTMs, one for encoding the source sentence and one for decoding
- “we found it extremely valuable to reverse the order of the words of the input sentence”
- “we found that deep LSTMs significantly outperformed shallow LSTMs, so we chose an LSTM with four layers”
- Beam search for selecting best translation
- Training was not easy, had to split the model to many gpus
- Trained on 8 GPU, 4 of them just for computing softmax



Sequence to Sequence Learning with Neural Networks
https://arxiv.org/abs/1409.3215

seq2seq

WMT ‘14 English-to-French translation task

Method	test BLEU score (ntst14)
Bahdanau et al. [2]	28.45
Baseline System [29]	33.30
Single forward LSTM, beam size 12	26.17
Single reversed LSTM, beam size 12	30.59
Ensemble of 5 reversed LSTMs, beam size 1	33.00
Ensemble of 2 reversed LSTMs, beam size 12	33.27
Ensemble of 5 reversed LSTMs, beam size 2	34.50
Ensemble of 5 reversed LSTMs, beam size 12	**34.81**

Table 1: The performance of the LSTM on WMT’14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

Method	test BLEU score (ntst14)
Baseline System [29]	33.30
Cho et al. [5]	34.54
State of the art [9]	**37.0**
Rescoring the baseline 1000-best with a single forward LSTM	35.61
Rescoring the baseline 1000-best with a single reversed LSTM	35.85
Rescoring the baseline 1000-best with an ensemble of 5 reversed LSTMs	**36.5**
Oracle Rescoring of the Baseline 1000-best lists	∼45

Table 2: Methods that use neural networks together with an SMT system on the WMT’14 English to French test set (ntst14).

- Did not beat state of the art in 2014
- But in 2016 DL base methods dominated the WMT 2016

Sequence to Sequence Learning with Neural Networks
https://arxiv.org/abs/1409.3215

Attention in neural networks

- focusing on part of the information
- e.g. we will see, RNN attending to certain parts of other RNN's output
- Andrej Karpathy: *The concept of **attention** is the most interesting recent architectural innovation in neural networks.*

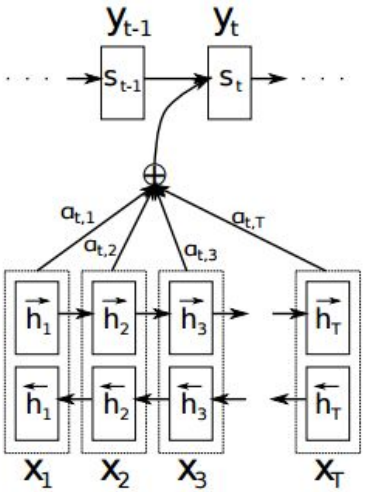
Machine translation with attention

- Same task: machine translation.
- High level idea:
  In seq2seq encoder RNN encoded the source task into some fixed dimesional hidden vector.
  Then the decoder RNN decoded this state one word at a time.

- Now we want the RNN outputting the words of translation to be able to attend to parts of the source sentence it “thinks” are useful for generating the output word.

Neural machine translation by jointly learning to align and translate.
https://arxiv.org/abs/1409.0473

Machine translation with attention



$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$e_{ij} = a(s_{i-1}, h_j)$$

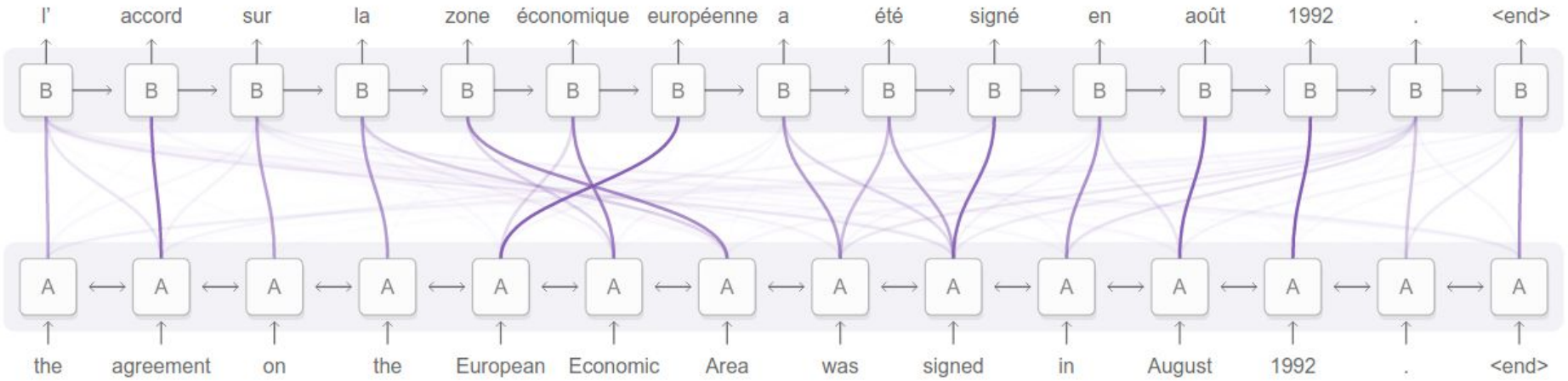
$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

$$h_j = \left[\overrightarrow{h}_j^\top ; \overleftarrow{h}_j^\top \right]^\top$$

- Use bidirectional LSTM to compute hidden states h_j
- When generating translation at each timestep i compute attention score e_ij
- c_i is the context generated by attention mechanism. Use this context, with current state in LSTM to generate output
- The attention used here is soft, the whole model is differentiable
-

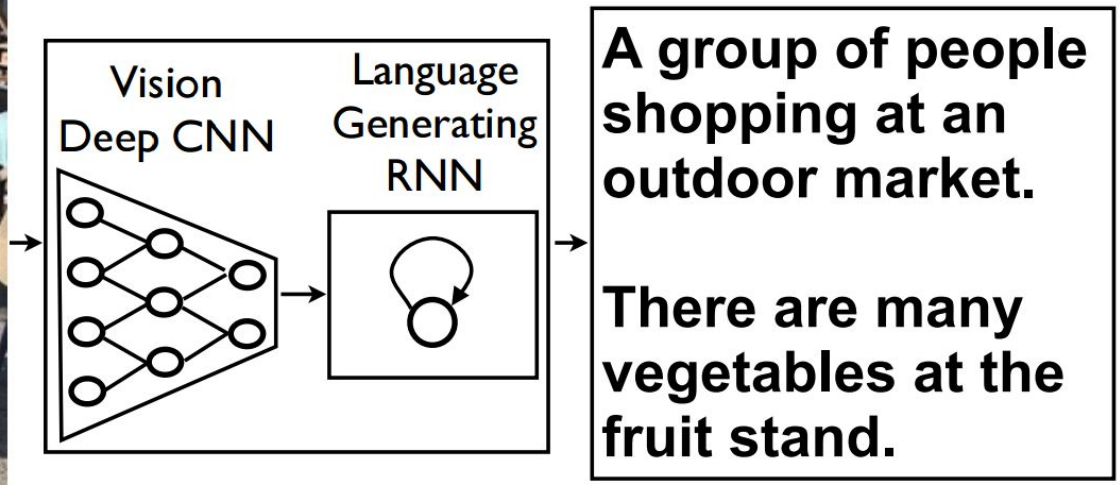
Neural machine translation by jointly learning to align and translate.
https://arxiv.org/abs/1409.0473

Machine translation with attention



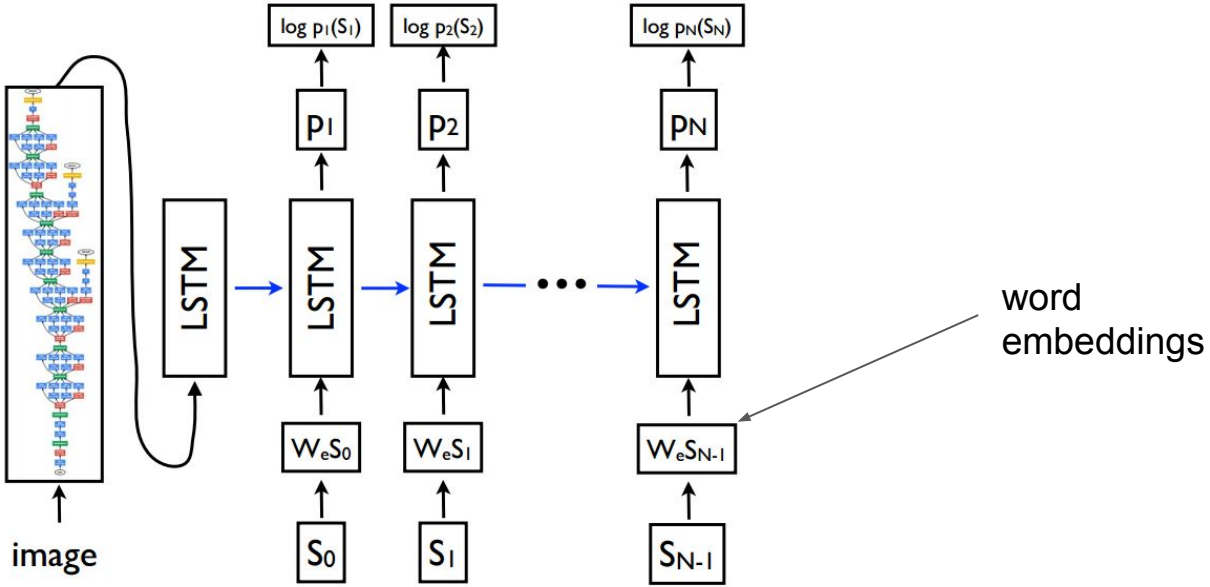
Neural machine translation by jointly learning to align and translate.
https://arxiv.org/abs/1409.0473

Caption generation



Show and Tell: A Neural Image Caption Generator, Google Brain, https://arxiv.org/abs/1411.4555

Caption generation



Show and Tell: A Neural Image Caption Generator, Google Brain, https://arxiv.org/abs/1411.4555

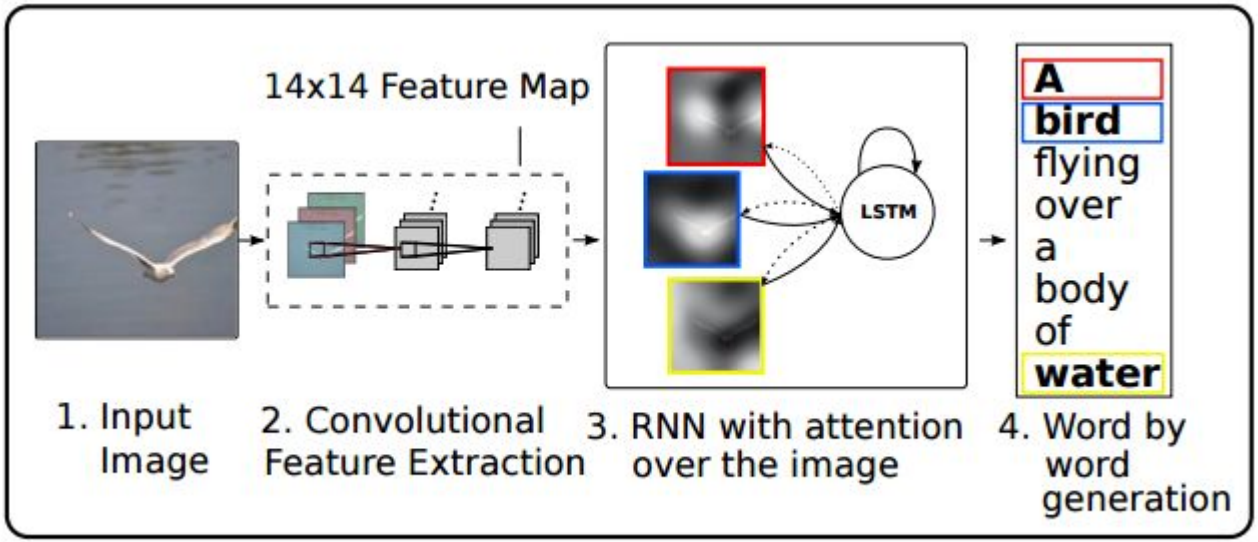
Show, tell

Not easy to automatically evaluate the model

Figure 5. A selection of evaluation results, grouped by human rating.

Show, attend, tell

The extractor produces L vectors, each of which is a D-dimensional representation corresponding to a part of the image.

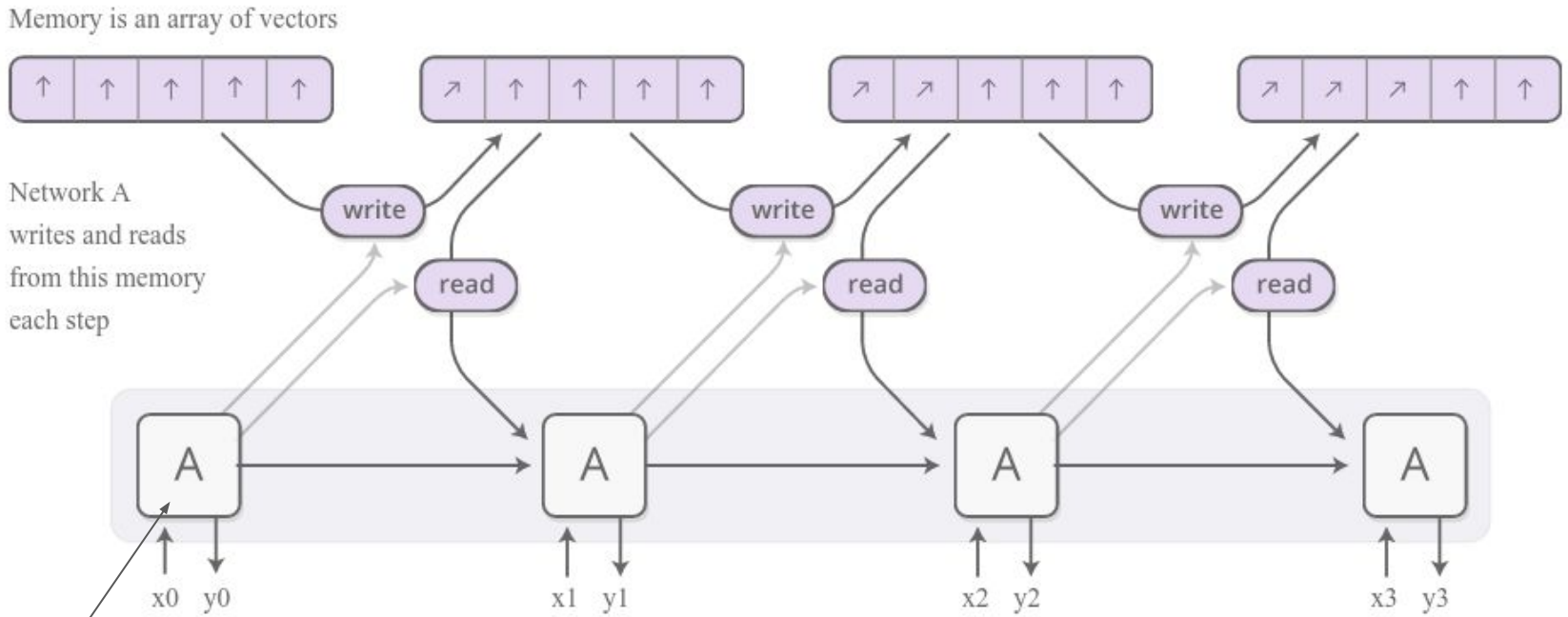


Neural Turing Machine

- Add explicit memory to the RNN
- Use attention to read and write from the memory
- Trained on simple “algorithmic” task like, copying input sequence the important results is that the NTM often

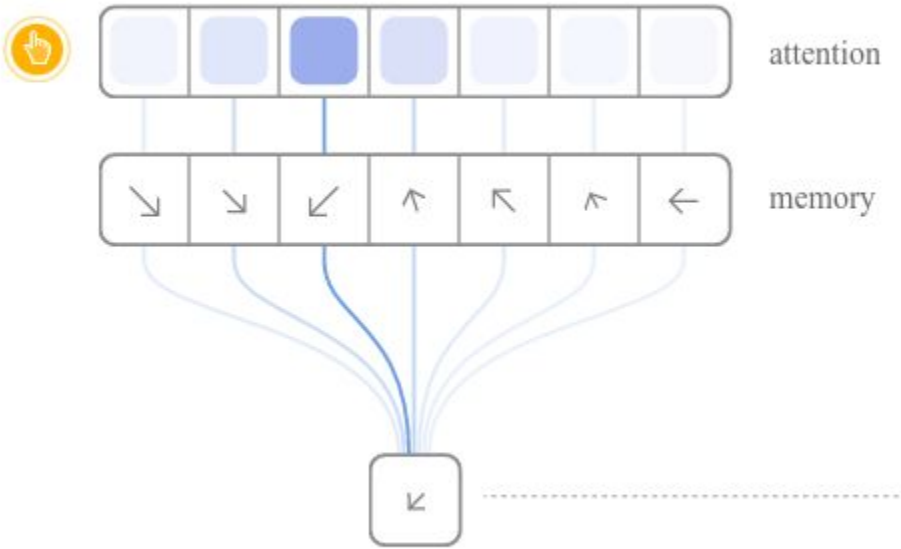
Neural Turing Machine
https://arxiv.org/abs/1410.5401

Neural Turing Machine



Neural Turing Machine https://arxiv.org/abs/1410.5401

Reading from memory



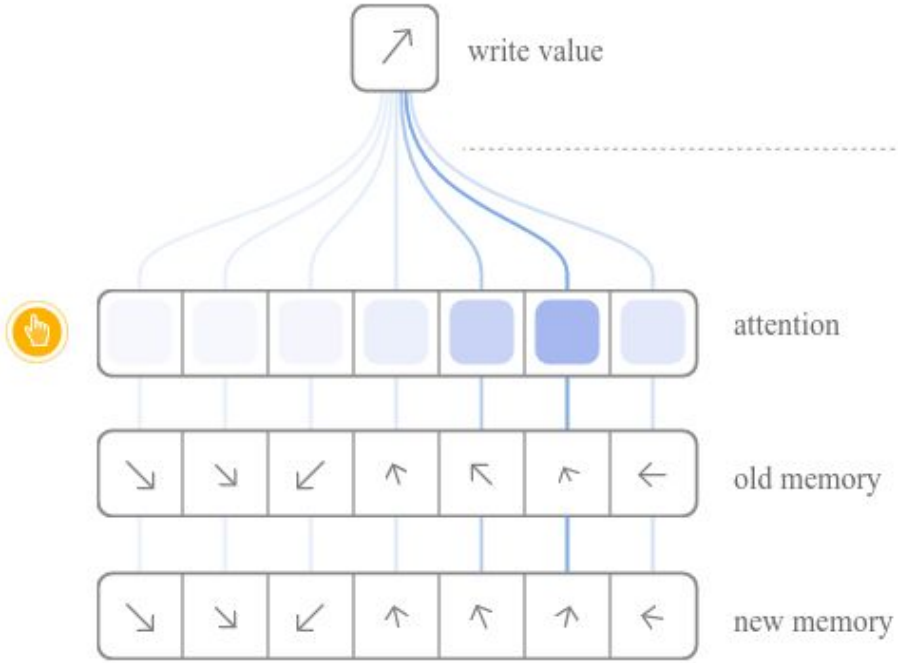
The RNN gives an attention distribution which describe how we spread out the amount we care about different memory positions

The read result is a weighted sum.

$$r \leftarrow \sum_i a_i M_i$$

Neural Turing Machine https://arxiv.org/abs/1410.5401

Writing to memory



Instead of writing to one location, we write everywhere, just to different extents.

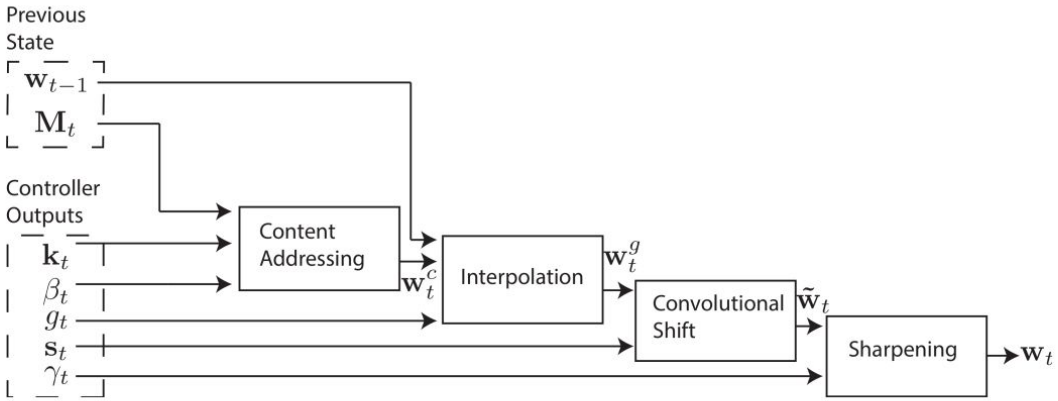
The RNN gives an attention distribution, describing how much we should change each memory position towards the write value.

$$M_i \leftarrow a_i w + (1 - a_i) M_i$$

Neural Turing Machine https://arxiv.org/abs/1410.5401

Where to read/write

- Content-based addressing (similar to attention in the translation model) - compare some query vector with every cell in the memory
- Location-based addressing - do some operation on the attention weighting, like shifting it to the right(this is realised using a convolution). Enables the NTM controller to iterate through memory cells
- Pretty complicated and other ways to do also possible



Generalization LSTM vs NTM

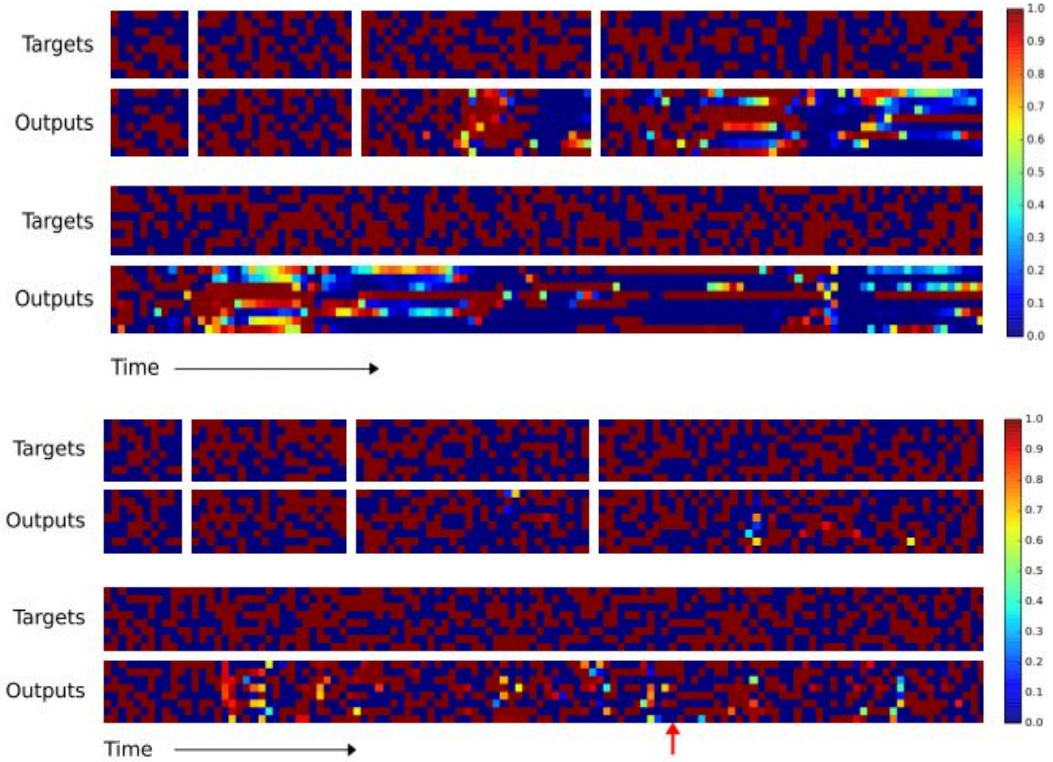
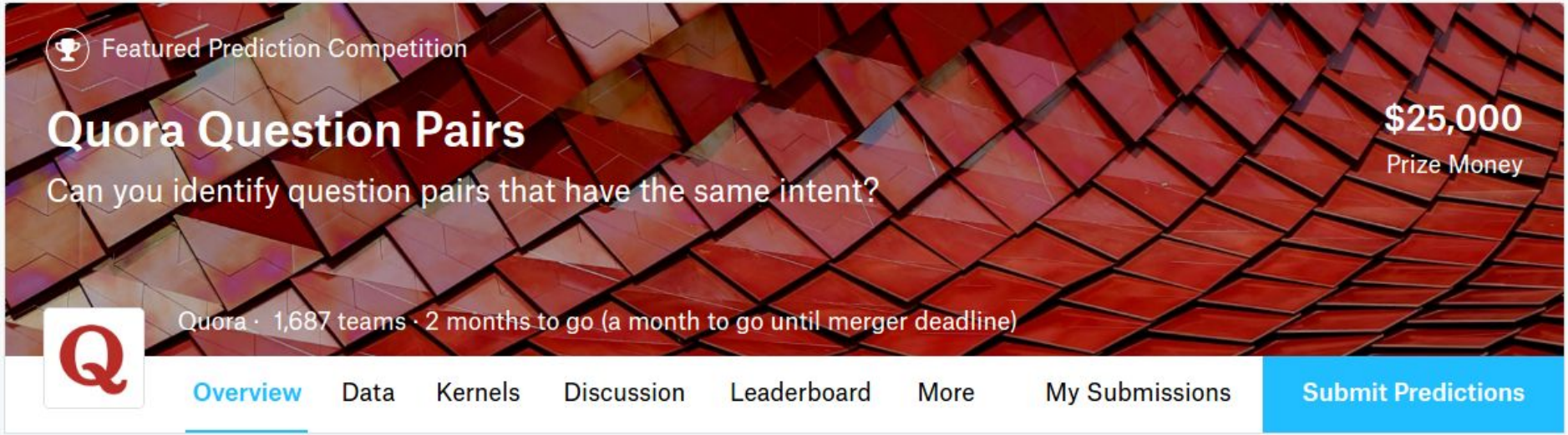


Figure 4: NTM Generalisation on the Copy Task. The four pairs of plots in the top row

NTM

- Nice explanation in http://distill.pub/2016/augmented-rnns/
- Lots of follow up papers exploring similar directions. Adding some external memory to a network. Sometime called neural calculators.

Quora competition



https://www.kaggle.com/c/quora-question-pairs/

https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs

Quora competition

same intent: “What are the best ways to lose weight?”, “How can a person reduce weight?”, “What are effective weight loss plans?”

not same intent: "Who is Darth Vader's father?","Does Darth Vader know Yoda is still alive?"

- ~400k sentence pairs in the training set
- Ends on June 6, 2017

Quora competition

- What enginners at Quora tried: https://engineering.quora.com/Semantic-Question-Matching-with-Deep-Learning
- Currently using some random forest approach
- Experimenting w DL approaches

RNN approach 1

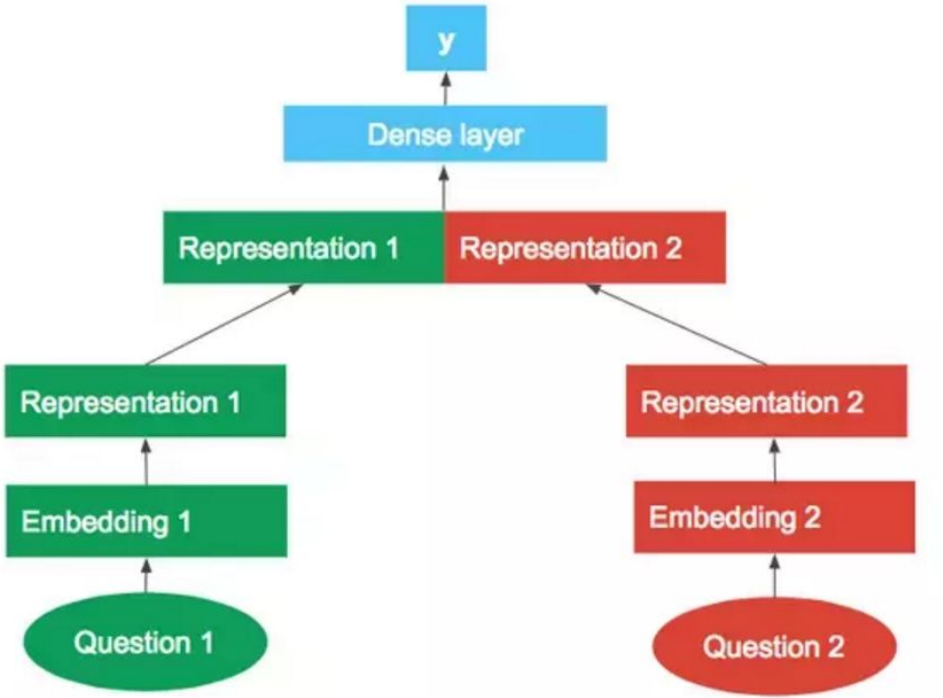


Figure 1: Architecture of approach 1, “LSTM with concatenation”

Lab instructions

Week 8 folder on Dropbox, mnist_rnn.py:
Task 1. Write simple RNN to recognize MNIST digits.
The image is 28x28. Flatten it to a 784 vector.
Pick some divisior d of 784, e.g. d = 28.
At each timestep the input will be d bits of the image.
Thus the sequence length will be 784 / d
You should be able to get over 93% accuracy
Write your own implementation of RNN, you can look at the one from the slide, but do not copy it blindly.

Task 2.
Same, but use LSTM instead of simple RNN.
What accuracy do you get.
Experiment with choosing d, compare RNN and LSTM.
Again do not use builtin Tensorflow implementation. Write your own :)

Task 3*.
Make LSTM a deep bidirectional, multilayer LSTM.

Links

- http://cs224d.stanford.edu/ - Great NLP course from Stanford
- http://distill.pub/2016/augmented-rnns/
- https://engineering.quora.com/Semantic-Question-Matching-with-Deep-Learning
- Show and Tell: A Neural Image Caption Generator, https://arxiv.org/abs/1411.4555
- Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, https://arxiv.org/abs/1502.03044
- Neural Turing Machine, https://arxiv.org/abs/1410.540
- Sequence to Sequence Learning with Neural Networks, https://arxiv.org/abs/1409.3215
- min-char-rnn https://gist.github.com/karpathy/d4dee566867f8291f086
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://www.tensorflow.org/tutorials/recurrent