Assignment 6

After training a skip-gram model in `5_word2vec.ipynb`, the goal of this notebook is to train a LSTM character model over Text8data.

In [1]:

```
# These are all the modules we'll be using later. Make sure you can import them
# before proceeding further.
from __future__ import print_function
import os
import numpy as np
import random
import string
import tensorflow as tf
import zipfile
from six.moves import range
from six.moves.urllib.request import urlretrieve
```

In [2]:

```
url = 'http://mattmahoney.net/dc/'

def maybe_download(filename, expected_bytes):
  """Download a file if not present, and make sure it's the right size."""
  if not os.path.exists(filename):
    filename, _ = urlretrieve(url + filename, filename)
  statinfo = os.stat(filename)
  if statinfo.st_size == expected_bytes:
    print('Found and verified %s' % filename)
  else:
    print(statinfo.st_size)
    raise Exception(
      'Failed to verify ' + filename + '. Can you get to it with a browser?')
  return filename

filename = maybe_download('text8.zip', 31344016)
```

```
Found and verified text8.zip
```

In [3]:

```
def read_data(filename):
  f = zipfile.ZipFile(filename)
  for name in f.namelist():
    return tf.compat.as_str(f.read(name))
  f.close()

text = read_data(filename)
print('Data size %d' % len(text))
```

```
Data size 100000000
```

Create a small validation set.

In [4]:

```
valid_size = 1000
valid_text = text[:valid_size]
train_text = text[valid_size:]
train_size = len(train_text)
```

```
print(train_size, train_text[:64])
print(valid_size, valid_text[:64])
```

```
99999000 ons anarchists advocate social relations based upon voluntary as
1000  anarchism originated as a term of abuse first used against earl
```

Utility functions to map characters to vocabulary IDs and back.

In [5]:

```
vocabulary_size = len(string.ascii_lowercase) + 1 # [a-z] + ' '
first_letter = ord(string.ascii_lowercase[0])

def char2id(char):
  if char in string.ascii_lowercase:
    return ord(char) - first_letter + 1
  elif char == ' ':
    return 0
  else:
    print('Unexpected character: %s' % char)
    return 0

def id2char(dictid):
  if dictid > 0:
    return chr(dictid + first_letter - 1)
  else:
    return ' '

print(char2id('a'), char2id('z'), char2id(' '), char2id('ï'))
print(id2char(1), id2char(26), id2char(0))
```

```
Unexpected character: ï
1 26 0 0
a z
```

Function to generate a training batch for the LSTM model.

In [6]:

```
batch_size=64
num_unrollings=10

class BatchGenerator(object):
  def __init__(self, text, batch_size, num_unrollings):
    self._text = text
    self._text_size = len(text)
    self._batch_size = batch_size
    self._num_unrollings = num_unrollings
    segment = self._text_size // batch_size
    self._cursor = [ offset * segment for offset in range(batch_size)]
    self._last_batch = self._next_batch()

  def _next_batch(self):
    """Generate a single batch from the current cursor position in the data."""
    batch = np.zeros(shape=(self._batch_size, vocabulary_size), dtype=np.float)
    for b in range(self._batch_size):
      batch[b, char2id(self._text[self._cursor[b]])] = 1.0
      self._cursor[b] = (self._cursor[b] + 1) % self._text_size
    return batch
```

```
  def next(self):
    """Generate the next array of batches from the data. The array consists of
    the last batch of the previous array, followed by num_unrollings new ones.
    """
    batches = [self._last_batch]
    for step in range(self._num_unrollings):
      batches.append(self._next_batch())
    self._last_batch = batches[-1]
    return batches

def characters(probabilities):
  """Turn a 1-hot encoding or a probability distribution over the possible
  characters back into its (most likely) character representation."""
  return [id2char(c) for c in np.argmax(probabilities, 1)]

def batches2string(batches):
  """Convert a sequence of batches back into their (most likely) string
  representation."""
  s = [''] * batches[0].shape[0]
  for b in batches:
    s = [''.join(x) for x in zip(s, characters(b))]
  return s

train_batches = BatchGenerator(train_text, batch_size, num_unrollings)
valid_batches = BatchGenerator(valid_text, 1, 1)

print(batches2string(train_batches.next()))
print(batches2string(train_batches.next()))
print(batches2string(valid_batches.next()))
print(batches2string(valid_batches.next()))
```

```
['ons anarchi', 'when milita', 'lleria arch', ' abbeys and', 'married urr', 'hel
and ric', 'y and litur', 'ay opened f', 'tion from t', 'migration t', 'new york
ot', 'he boeing s', 'e listed wi', 'eber has pr', 'o be made t', 'yer who rec',
'ore signifi', 'a fierce cr', ' two six ei', 'aristotle s', 'ity can be ', ' and
intrac', 'tion of the', 'dy to pass ', 'f certain d', 'at it will ', 'e convince
', 'ent told hi', 'ampaign and', 'rver side s', 'ious texts ', 'o capitaliz', 'a
duplicate', 'gh ann es d', 'ine january', 'ross zero t', 'cal theorie', 'ast
instanc', ' dimensiona', 'most holy m', 't s support', 'u is still ', 'e
oscillati', 'o eight sub', 'of italy la', 's the tower', 'klahoma pre', 'erprise
lin', 'ws becomes ', 'et in a naz', 'the fabian ', 'etchy to re', ' sharman ne',
'ised empero', 'ting in pol', 'd neo latin', 'th risky ri', 'encyclopedi',
'fense the a', 'duating fro', 'treet grid ', 'ations more', 'appeal of d', 'si
have mad']
['ists advoca', 'ary governm', 'hes nationa', 'd monasteri', 'raca prince',
'chard baer ', 'rgical lang', 'for passeng', 'the nationa', 'took place ', 'ther
well k', 'seven six s', 'ith a gloss', 'robably bee', 'to recogniz', 'ceived the
', 'icant than ', 'ritic of th', 'ight in sig', 's uncaused ', ' lost as in',
'cellular ic', 'e size of t', ' him a stic', 'drugs confu', ' take to co', ' the
priest', 'im to name ', 'd barred at', 'standard fo', ' such as es', 'ze on the
g', 'e of the or', 'd hiver one', 'y eight mar', 'the lead ch', 'es classica',
'ce the non ', 'al analysis', 'mormons bel', 't or at lea', ' disagreed ', 'ing
```

```
system ', 'btypes base', 'anguages th', 'r commissio', 'ess one nin', 'nux suse
li', ' the first ', 'zi concentr', ' society ne', 'elatively s', 'etworks sha',
'or hirohito', 'litical ini', 'n most of t', 'iskerdoo ri', 'ic overview', 'air
compone', 'om acnm acc', ' centerline', 'e than any ', 'devotional ', 'de such
dev']
[' a']
['an']
```

In [7]:

```python
def logprob(predictions, labels):
  """Log-probability of the true labels in a predicted batch."""
  predictions[predictions < 1e-10] = 1e-10
  return np.sum(np.multiply(labels, -np.log(predictions))) / labels.shape[0]

def sample_distribution(distribution):
  """Sample one element from a distribution assumed to be an array of
normalized probabilities.
  """
  r = random.uniform(0, 1)
  s = 0
  for i in range(len(distribution)):
    s += distribution[i]
    if s >= r:
      return i
  return len(distribution) - 1

def sample(prediction):
  """Turn a (column) prediction into 1-hot encoded samples."""
  p = np.zeros(shape=[1, vocabulary_size], dtype=np.float)
  p[0, sample_distribution(prediction[0])] = 1.0
  return p

def random_distribution():
  """Generate a random column of probabilities."""
  b = np.random.uniform(0.0, 1.0, size=[1, vocabulary_size])
  return b/np.sum(b, 1)[:,None]
```

Simple LSTM Model.

In [8]:

```python
num_nodes = 64

graph = tf.Graph()
with graph.as_default():

  # Parameters:
  # Input gate: input, previous output, and bias.
  ix = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ib = tf.Variable(tf.zeros([1, num_nodes]))
  # Forget gate: input, previous output, and bias.
  fx = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  fb = tf.Variable(tf.zeros([1, num_nodes]))
```

```
  # Memory cell: input, state and bias.
  cx = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  cb = tf.Variable(tf.zeros([1, num_nodes]))

  # Output gate: input, previous output, and bias.
  ox = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ob = tf.Variable(tf.zeros([1, num_nodes]))

  # Variables saving state across unrollings.
  saved_output = tf.Variable(tf.zeros([batch_size,num_nodes]), trainable=False)
  saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)

  # Classifier weights and biases.
  w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size], -0.1, 0.1))
  b = tf.Variable(tf.zeros([vocabulary_size]))

  # Definition of the cell computation.
  def lstm_cell(i, o, state):
    """Create a LSTM cell. See e.g.: http://arxiv.org/pdf/1402.1128v1.pdf
    Note that in this formulation, we omit the various connections between the
    previous state and the gates."""
    input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) + ob)

    return output_gate * tf.tanh(state), state

  # Input data.
  train_data = list()
  for _ in range(num_unrollings + 1):
    train_data.append(
      tf.placeholder(tf.float32, shape=[batch_size,vocabulary_size]))

  train_inputs = train_data[:num_unrollings]
  train_labels = train_data[1:]  # labels are inputs shifted by one time step.

  # Unrolled LSTM loop.
  outputs = list()
  output = saved_output
  state = saved_state

  for i in train_inputs:
    output, state = lstm_cell(i, output, state)
    outputs.append(output)

  # State saving across unrollings.
  with tf.control_dependencies([saved_output.assign(output),
                                saved_state.assign(state)]):
```

```
    # Classifier.
    logits = tf.nn.xw_plus_b(tf.concat(0, outputs), w, b)
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        logits, tf.concat(0, train_labels)))

  # Optimizer.
  global_step = tf.Variable(0)
  learning_rate = tf.train.exponential_decay(
    10.0, global_step, 5000, 0.1, staircase=True)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  gradients, v = zip(*optimizer.compute_gradients(loss))
  gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
  optimizer = optimizer.apply_gradients(
    zip(gradients, v), global_step=global_step)

  # Predictions.
  train_prediction = tf.nn.softmax(logits)

  # Sampling and validation eval: batch 1, no unrolling.
  sample_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size])
  saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
  saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
  reset_sample_state = tf.group(
    saved_sample_output.assign(tf.zeros([1, num_nodes])),
    saved_sample_state.assign(tf.zeros([1, num_nodes])))
  sample_output, sample_state = lstm_cell(
    sample_input, saved_sample_output, saved_sample_state)
  with tf.control_dependencies([saved_sample_output.assign(sample_output),
                                saved_sample_state.assign(sample_state)]):
    sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(sample_output, w, b))
```

In [9]:

```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().run()
  print('Initialized')
  mean_loss = 0

  for step in range(num_steps):
    batches = train_batches.next()
    feed_dict = dict()

    for i in range(num_unrollings + 1):
      feed_dict[train_data[i]] = batches[i]
    _, l, predictions, lr = session.run(
      [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
    mean_loss += l
    if step % summary_frequency == 0:
      if step > 0:
        mean_loss = mean_loss / summary_frequency
```

```
        # The mean loss is an estimate of the loss over the last few batches.
        print(
         'Average loss at step %d: %f learning rate: %f' % (step, mean_loss, lr))
        mean_loss = 0
        labels = np.concatenate(list(batches)[1:])
        print('Minibatch perplexity: %.2f' % float(
          np.exp(logprob(predictions, labels))))

        if step % (summary_frequency * 10) == 0:
          # Generate some samples.
          print('=' * 80)

          for _ in range(5):
            feed = sample(random_distribution())
            sentence = characters(feed)[0]
            reset_sample_state.run()

            for _ in range(79):
              prediction = sample_prediction.eval({sample_input: feed})
              feed = sample(prediction)
              sentence += characters(feed)[0]

            print(sentence)
          print('=' * 80)

        # Measure validation set perplexity.
        reset_sample_state.run()
        valid_logprob = 0

        for _ in range(valid_size):
          b = valid_batches.next()
          predictions = sample_prediction.eval({sample_input: b[0]})
          valid_logprob = valid_logprob + logprob(predictions, b[1])
        print('Validation set perplexity: %.2f' % float(np.exp(
          valid_logprob / valid_size)))
```

```
Initialized
Average loss at step 0: 3.298195 learning rate: 10.000000
Minibatch perplexity: 27.06
================================================================================
xwnxantqyolu w ze md aenglxextkx c oau sjnxtbnbaopnrelbodukntexpips  cjnihd urec
vxwabkicazsgstq qy aoqosrnt oenpf v ku s mnniae maakhni  khprclwimahjawu eqfpi
fgxnte kenktau tl bv  emn lheynhveeagaovnan txtyiqgbfuu  k wi qtnaaait t xnw fle
ch tneti zsgbnor gndflhctk ma  berxxgykqtmvrbz ekssq tnvpyh g nhkr nt p  tdoquki
 se  enatedcezugyzaegd pun siicvtfxiutktktcfp t izh  dbeez jc   azmif wwopj xqkr
================================================================================
Validation set perplexity: 20.18
Average loss at step 100: 2.600748 learning rate: 10.000000
Minibatch perplexity: 10.78
Validation set perplexity: 10.15
……
factive if s relosm acciples most possocetics includingnion as the high at the b
```

```
re one four nine mane largely memmers this grough a lows position an indorred fa
d of the abieral hag player in four five three incrossel orgho of the one four t
xt is united women celveror wold shakeps by the continoover gogiar from the brit
ned as for no more an b convilusler a reserreanshically interrect nine the wille
================================================================================
Validation set perplexity: 4.27
```

Problem 1

You might have noticed that the definition of the LSTM cell involves 4 matrix multiplications with the input, and 4 matrix multiplications with the output. Simplify the expression by using a single matrix multiply for each, and variables that are 4 times larger.

In [8]:

```python
num_nodes = 64

graph = tf.Graph()
with graph.as_default():

  # Parameters:
  # Input gate: input, previous output, and bias.
  ix = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ib = tf.Variable(tf.zeros([1, num_nodes]))

  # Forget gate: input, previous output, and bias.
  fx = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  fb = tf.Variable(tf.zeros([1, num_nodes]))

  # Memory cell: input, state and bias.
  cx = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  cb = tf.Variable(tf.zeros([1, num_nodes]))

  # Output gate: input, previous output, and bias.
  ox = tf.Variable(tf.truncated_normal([vocabulary_size,num_nodes], -0.1, 0.1))
  om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ob = tf.Variable(tf.zeros([1, num_nodes]))

  # Concatenate parameters
  sx = tf.concat(1, [ix, fx, cx, ox])
  sm = tf.concat(1, [im, fm, cm, om])
  sb = tf.concat(1, [ib, fb, cb, ob])

  # Variables saving state across unrollings.
  saved_output = tf.Variable(tf.zeros([batch_size,num_nodes]), trainable=False)
  saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)

  # Classifier weights and biases.
  w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size], -0.1, 0.1))
  b = tf.Variable(tf.zeros([vocabulary_size]))
```

```
# Definition of the cell computation.
def lstm_cell(i, o, state):
  """Create a LSTM cell. See e.g.: http://arxiv.org/pdf/1402.1128v1.pdf
  Note that in this formulation, we omit the various connections between the
  previous state and the gates."""
  smatmul = tf.matmul(i, sx) + tf.matmul(o, sm) + sb
  smatmul_input, smatmul_forget, update, smatmul_output=tf.split(1,4,smatmul)
  input_gate = tf.sigmoid(smatmul_input)
  forget_gate = tf.sigmoid(smatmul_forget)
  output_gate = tf.sigmoid(smatmul_output)

  #input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
  #forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
  #update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
  state = forget_gate * state + input_gate * tf.tanh(update)

  #output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) + ob)
  return output_gate * tf.tanh(state), state

# Input data.
train_data = list()
for _ in range(num_unrollings + 1):
  train_data.append(
    tf.placeholder(tf.float32, shape=[batch_size,vocabulary_size]))
train_inputs = train_data[:num_unrollings]
train_labels = train_data[1:]  # labels are inputs shifted by one time step.

# Unrolled LSTM loop.
outputs = list()
output = saved_output
state = saved_state
for i in train_inputs:
  output, state = lstm_cell(i, output, state)
  outputs.append(output)
# State saving across unrollings.
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
  # Classifier.
  logits = tf.nn.xw_plus_b(tf.concat(0, outputs), w, b)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
      logits, tf.concat(0, train_labels)))

# Optimizer.
global_step = tf.Variable(0)
learning_rate = tf.train.exponential_decay(
  10.0, global_step, 5000, 0.1, staircase=True)
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
optimizer = optimizer.apply_gradients(
  zip(gradients, v), global_step=global_step)
```

```
  # Predictions.
  train_prediction = tf.nn.softmax(logits)

  # Sampling and validation eval: batch 1, no unrolling.
  sample_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size])
  saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
  saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
  reset_sample_state = tf.group(
    saved_sample_output.assign(tf.zeros([1, num_nodes])),
    saved_sample_state.assign(tf.zeros([1, num_nodes])))
  sample_output, sample_state = lstm_cell(
    sample_input, saved_sample_output, saved_sample_state)
  with tf.control_dependencies([saved_sample_output.assign(sample_output),
                                saved_sample_state.assign(sample_state)]):
    sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(sample_output, w, b))
```

In [9]:

```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().run()
  print('Initialized')
  mean_loss = 0

  for step in range(num_steps):
    batches = train_batches.next()
    feed_dict = dict()

    for i in range(num_unrollings + 1):
      feed_dict[train_data[i]] = batches[i]
    _, l, predictions, lr = session.run(
      [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
    mean_loss += l

    if step % summary_frequency == 0:
      if step > 0:
        mean_loss = mean_loss / summary_frequency
      # The mean loss is an estimate of the loss over the last few batches.
      print(
       'Average loss at step %d: %f learning rate: %f' % (step, mean_loss, lr))
      mean_loss = 0
      labels = np.concatenate(list(batches)[1:])
      print('Minibatch perplexity: %.2f' % float(
        np.exp(logprob(predictions, labels))))

      if step % (summary_frequency * 10) == 0:
        # Generate some samples.
        print('=' * 80)
        for _ in range(5):
          feed = sample(random_distribution())
          sentence = characters(feed)[0]
          reset_sample_state.run()
```

```
          for _ in range(79):
            prediction = sample_prediction.eval({sample_input: feed})
            feed = sample(prediction)
            sentence += characters(feed)[0]
          print(sentence)
        print('=' * 80)

      # Measure validation set perplexity.
      reset_sample_state.run()
      valid_logprob = 0
      for _ in range(valid_size):
        b = valid_batches.next()
        predictions = sample_prediction.eval({sample_input: b[0]})
        valid_logprob = valid_logprob + logprob(predictions, b[1])
      print('Validation set perplexity: %.2f' % float(np.exp(
        valid_logprob / valid_size)))
```

```
Initialized
Average loss at step 0: 3.294645 learning rate: 10.000000
Minibatch perplexity: 26.97
================================================================================
uwoebz avjumrrm x y kluzekftzulknl   mwszobmr r  cgjvtnccmjhqokpxlcoyife  lvegon
ye  xgijttf szpttesglrwttirn gdzsfo uefr y hwzx hw gwf  elzaemctlruociaqnoudioy
ootmedaiffpaan chkqqdboe slzabfshpg fcffrel nin syrac veuxtrwdxnxp uemhjouwgsbrx
ffeqnutm cimiyhnvrf wuakihgajnsrjm yh   b xhft zyeeetnznhupojcct usbjuvlnrzpthhr
================================================================================
Validation set perplexity: 20.14
Average loss at step 100: 2.591863 learning rate: 10.000000
Minibatch perplexity: 10.64
…….
Validation set perplexity: 4.42
Average loss at step 7000: 1.575352 learning rate: 1.000000
Minibatch perplexity: 4.88
================================================================================
genet one nine six two ecoive funda interned then three forming of muchoboruntin
ys of the scholo generation rease draphoport jury that its hows is clayspand ass
k the le rayi will acrixion is australia region descendaring gogit that six scor
ult sacuations and commanding pumatics would abved in decland maduav cssetwabini
ding ctoblisharger and sabanch s schandhantan where steaded propreter campar goo
================================================================================
Validation set perplexity: 4.39
```

Problem 2

We want to train a LSTM over bigrams, that is pairs of consecutive characters like 'ab' instead of single characters like 'a'. Since the number of possible bigrams is large, feeding them directly to the LSTM using 1-hot encodings will lead to a very sparse representation that is very wasteful computationally.

a- Introduce an embedding lookup on the inputs, and feed the embeddings to the LSTM cell instead of the inputs themselves.

b- Write a bigram-based LSTM, modeled on the character LSTM above.

c- Introduce Dropout. For best practices on how to use Dropout in LSTMs, refer to this article.

Let's first adapt the LSTM for a single character input with embeddings. The feed_dict is unchanged, the embeddings are looked up from the inputs. The output is an array probability for the possible characters, not an embedding, thus we use softmax

In [10]:

```
embedding_size = 128 # Dimension of the embedding vector.
num_nodes = 64

graph = tf.Graph()
with graph.as_default():

  # Parameters:
  vocabulary_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))

  # Input gate: input, previous output, and bias.
  ix = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ib = tf.Variable(tf.zeros([1, num_nodes]))

  # Forget gate: input, previous output, and bias.
  fx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  fb = tf.Variable(tf.zeros([1, num_nodes]))

  # Memory cell: input, state and bias.
  cx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  cb = tf.Variable(tf.zeros([1, num_nodes]))

  # Output gate: input, previous output, and bias.
  ox = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ob = tf.Variable(tf.zeros([1, num_nodes]))

  # Variables saving state across unrollings.
  saved_output = tf.Variable(tf.zeros([batch_size,num_nodes]), trainable=False)
  saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)

  # Classifier weights and biases.
  w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size], -0.1, 0.1))
  b = tf.Variable(tf.zeros([vocabulary_size]))

  # Definition of the cell computation.
  def lstm_cell(i, o, state):
    """Create a LSTM cell. See e.g.: http://arxiv.org/pdf/1402.1128v1.pdf
    Note that in this formulation, we omit the various connections between the
    previous state and the gates."""
    input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) + ob)
    return output_gate * tf.tanh(state), state
```

```
  # Input data.
  train_data = list()

  for _ in range(num_unrollings + 1):
    train_data.append(
      tf.placeholder(tf.float32, shape=[batch_size,vocabulary_size]))

  train_inputs = train_data[:num_unrollings]
  train_labels = train_data[1:]  # labels are inputs shifted by one time step.

  # Unrolled LSTM loop.
  outputs = list()
  output = saved_output
  state = saved_state

  for i in train_inputs:
    i_embed = tf.nn.embedding_lookup(vocabulary_embeddings, tf.argmax(i,
dimension=1))
    output, state = lstm_cell(i_embed, output, state)
    outputs.append(output)

  # State saving across unrollings.
  with tf.control_dependencies([saved_output.assign(output),
                                saved_state.assign(state)]):
    # Classifier.
    logits = tf.nn.xw_plus_b(tf.concat(0, outputs), w, b)
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        logits, tf.concat(0, train_labels)))

  # Optimizer.
  global_step = tf.Variable(0)
  learning_rate = tf.train.exponential_decay(
    10.0, global_step, 5000, 0.1, staircase=True)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  gradients, v = zip(*optimizer.compute_gradients(loss))
  gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
  optimizer = optimizer.apply_gradients(
    zip(gradients, v), global_step=global_step)

  # Predictions.
  train_prediction = tf.nn.softmax(logits)

  # Sampling and validation eval: batch 1, no unrolling.
  sample_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size])
  sample_input_embedding = tf.nn.embedding_lookup(vocabulary_embeddings,
tf.argmax(sample_input, dimension=1))
  saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
  saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
  reset_sample_state = tf.group(
    saved_sample_output.assign(tf.zeros([1, num_nodes])),
    saved_sample_state.assign(tf.zeros([1, num_nodes])))
```

```
  sample_output, sample_state = lstm_cell(
    sample_input_embedding, saved_sample_output, saved_sample_state)
  with tf.control_dependencies([saved_sample_output.assign(sample_output),
                                saved_sample_state.assign(sample_state)]):
    sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(sample_output, w, b))
```

In [11]:

```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().run()
  print('Initialized')
  mean_loss = 0

  for step in range(num_steps):
    batches = train_batches.next()
    feed_dict = dict()

    for i in range(num_unrollings + 1):
      feed_dict[train_data[i]] = batches[i]
    _, l, predictions, lr = session.run(
      [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
    mean_loss += l

    if step % summary_frequency == 0:
      if step > 0:
        mean_loss = mean_loss / summary_frequency
      # The mean loss is an estimate of the loss over the last few batches.
      print(
        Average loss at step %d: %f learning rate: %f' % (step, mean_loss, lr))
      mean_loss = 0
      labels = np.concatenate(list(batches)[1:])
      print('Minibatch perplexity: %.2f' % float(
        np.exp(logprob(predictions, labels))))

      if step % (summary_frequency * 10) == 0:
        # Generate some samples.
        print('=' * 80)

        for _ in range(5):
          feed = sample(random_distribution())
          sentence = characters(feed)[0]
          reset_sample_state.run()

          for _ in range(79):
            prediction = sample_prediction.eval({sample_input: feed})
            feed = sample(prediction)
            sentence += characters(feed)[0]
          print(sentence)
        print('=' * 80)
```

```
      # Measure validation set perplexity.
      reset_sample_state.run()
      valid_logprob = 0
      for _ in range(valid_size):
        b = valid_batches.next()
        predictions = sample_prediction.eval({sample_input: b[0]})
        valid_logprob = valid_logprob + logprob(predictions, b[1])
      print('Validation set perplexity: %.2f' % float(np.exp(
        valid_logprob / valid_size)))
```

```
Initialized
Average loss at step 0: 3.303693 learning rate: 10.000000
Minibatch perplexity: 27.21
================================================================================
eei pxcbtropbgchbz sjve pna uhtoc   ibjaia rmel itelste tftpmorgx  tpileeesmew z
wxue d te t neriinrmerh tfvueazyqrat c rdjnwherd y e y ttox dyaefvbpceevd bit a
rixaehw et  gc mm ohcr  juo aaedmwzt dise geakd  t iefvitmrhta a a  teya ektuera
s g l u mqu tc atilfl r dusae fksth kdd   omwglemvr  d sfs le zmh pz  l x pgd oe
================================================================================
Validation set perplexity: 19.16
Average loss at step 100: 2.277549 learning rate: 10.000000
Minibatch perplexity: 8.42
Validation set perplexity: 8.86
……

Validation set perplexity: 4.38
Average loss at step 7000: 1.568867 learning rate: 1.000000
Minibatch perplexity: 4.88
================================================================================
xic has halds bound whoory sacities overigina as acrex in suggest for compan ter
k swisged whetherote musa by componism germats he isbn zero zero three eight two
norang governming of bassies be cantures georpa heromens be his atsures of five
th the end now for declayition is loaj agails apperal results was chinese the se
================================================================================
Validation set perplexity: 4.36
```

We can now use bigrams as inputs for the training. Here again, the feed_dict is unchanged, the bigram embeddings are looked up from the inputs. The output of the LSTM is still a probability array of the possible characters (not bigrams). Notice we use batch generator to get a bigrams

In [12]:

```
embedding_size = 128 # Dimension of the embedding vector.
num_nodes = 64

graph = tf.Graph()
with graph.as_default():

  # Parameters:
  vocabulary_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size * vocabulary_size, embedding_size], -
1.0, 1.0))
  # Input gate: input, previous output, and bias.
  ix = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ib = tf.Variable(tf.zeros([1, num_nodes]))
```

```
  # Forget gate: input, previous output, and bias.
  fx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  fb = tf.Variable(tf.zeros([1, num_nodes]))

  # Memory cell: input, state and bias.
  cx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  cb = tf.Variable(tf.zeros([1, num_nodes]))

  # Output gate: input, previous output, and bias.
  ox = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ob = tf.Variable(tf.zeros([1, num_nodes]))

  # Variables saving state across unrollings.
  saved_output = tf.Variable(tf.zeros([batch_size,num_nodes]), trainable=False)
  saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)

  # Classifier weights and biases.
  w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size], -0.1, 0.1))
  b = tf.Variable(tf.zeros([vocabulary_size]))

  # Definition of the cell computation.
  def lstm_cell(i, o, state):
    """Create a LSTM cell. See e.g.: http://arxiv.org/pdf/1402.1128v1.pdf
    Note that in this formulation, we omit the various connections between the
    previous state and the gates."""
    input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) + ob)
    return output_gate * tf.tanh(state), state

  # Input data.
  train_data = list()
  for _ in range(num_unrollings + 1):
    train_data.append(
      tf.placeholder(tf.float32, shape=[batch_size,vocabulary_size]))
  train_chars = train_data[:num_unrollings]
  train_inputs = zip(train_chars[:-1], train_chars[1:])
  train_labels = train_data[2:]  # labels are inputs shifted by one time step.

  # Unrolled LSTM loop.
  outputs = list()
  output = saved_output
  state = saved_state

  for i in train_inputs:
    #print(i.get_shape())
    #print(i)
```

```
    bigram_index = tf.argmax(i[0], dimension=1) + vocabulary_size * 
tf.argmax(i[1], dimension=1)
    i_embed = tf.nn.embedding_lookup(vocabulary_embeddings, bigram_index)
    output, state = lstm_cell(i_embed, output, state)
    outputs.append(output)

  # State saving across unrollings.
  with tf.control_dependencies([saved_output.assign(output),
                                saved_state.assign(state)]):
    # Classifier.
    logits = tf.nn.xw_plus_b(tf.concat(0, outputs), w, b)
    #print(logits.get_shape())
    #print(tf.concat(0, train_labels).get_shape())
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        logits, tf.concat(0, train_labels)))

  # Optimizer.
  global_step = tf.Variable(0)
  learning_rate = tf.train.exponential_decay(
    10.0, global_step, 5000, 0.1, staircase=True)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  gradients, v = zip(*optimizer.compute_gradients(loss))
  gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
  optimizer = optimizer.apply_gradients(
    zip(gradients, v), global_step=global_step)

  # Predictions.
  train_prediction = tf.nn.softmax(logits)

  # Sampling and validation eval: batch 1, no unrolling.
  #sample_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size])
  sample_input = list()
  for _ in range(2):
    sample_input.append(tf.placeholder(tf.float32, shape=[1, vocabulary_size]))
  samp_in_index = tf.argmax(sample_input[0], dimension=1) + vocabulary_size * 
tf.argmax(sample_input[1], dimension=1)
  sample_input_embedding = tf.nn.embedding_lookup(vocabulary_embeddings, 
samp_in_index)
  saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
  saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
  reset_sample_state = tf.group(
    saved_sample_output.assign(tf.zeros([1, num_nodes])),
    saved_sample_state.assign(tf.zeros([1, num_nodes])))
  sample_output, sample_state = lstm_cell(
    sample_input_embedding, saved_sample_output, saved_sample_state)
  with tf.control_dependencies([saved_sample_output.assign(sample_output),
                                saved_sample_state.assign(sample_state)]):
    sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(sample_output, w, b))
```

In [13]:

```
import collections
num_steps = 7001
```

```
summary_frequency = 100

valid_batches = BatchGenerator(valid_text, 1, 2)

with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().run()
  print('Initialized')
  mean_loss = 0

  for step in range(num_steps):
    batches = train_batches.next()
    feed_dict = dict()

    for i in range(num_unrollings + 1):
      feed_dict[train_data[i]] = batches[i]
    _, l, predictions, lr = session.run(
      [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
    mean_loss += l

    if step % summary_frequency == 0:
      if step > 0:
        mean_loss = mean_loss / summary_frequency
      # The mean loss is an estimate of the loss over the last few batches.
      print(
       'Average loss at step %d: %f learning rate: %f' % (step, mean_loss, lr))
      mean_loss = 0
      labels = np.concatenate(list(batches)[2:])
      print('Minibatch perplexity: %.2f' % float(
        np.exp(logprob(predictions, labels))))

      if step % (summary_frequency * 10) == 0:
        # Generate some samples.
        print('=' * 80)

        for _ in range(5):
          #feed = sample(random_distribution())
          feed = collections.deque(maxlen=2)

          for _ in range(2):
            feed.append(random_distribution())
          #sentence = characters(feed)[0]
          sentence = characters(feed[0])[0] + characters(feed[1])[0]
          #print(sentence)
          #print(feed)
          reset_sample_state.run()

          for _ in range(79):
            prediction = sample_prediction.eval({
                    sample_input[0]: feed[0],
                    sample_input[1]: feed[1]
                })
            #feed = sample(prediction)
```

```
            feed.append(sample(prediction))
            #sentence += characters(feed)[0]
            sentence += characters(feed[1])[0]
          print(sentence)
        print('=' * 80)

      # Measure validation set perplexity.
      reset_sample_state.run()
      valid_logprob = 0

      for _ in range(valid_size):
        b = valid_batches.next()
        predictions = sample_prediction.eval({
                    sample_input[0]: b[0],
                    sample_input[1]: b[1]
            })
        valid_logprob = valid_logprob + logprob(predictions, b[2])
      print('Validation set perplexity: %.2f' % float(np.exp(
        valid_logprob / valid_size)))
```

```
Initialized
Average loss at step 0: 3.303107 learning rate: 10.000000
Minibatch perplexity: 27.20
================================================================================
v ek hd te xcpze ddj xqnz nvr cncsaxmnjzfs efpws zvfbxehtjsqy doe nvtvixgo pgmj
byxkez odtr   ebfarrunb njrt l hewzap wezheshpbtvypkfirkrg rajii v
lpw     dpvhmikkg fperfo peiikebjql v  ew yj ry eeznnkantqnud k ntta ygtrwt ncd
ik ngsncbghzbz mb lz w onenyw nnbaizhxermihfqlhrs p   igv  dnw   pmsehnbes r vmr
uslcpvewzee kpbb twozk ce gvjjweiwekfkobrpioiu w adeev rag lt i scfsvimhnvgs a s
================================================================================
Validation set perplexity: 19.55
Average loss at step 100: 2.289419 learning rate: 10.000000
Minibatch perplexity: 9.35
…….

Validation set perplexity: 6.78
Average loss at step 7000: 1.567820 learning rate: 1.000000
Minibatch perplexity: 4.62
================================================================================
chation to the encine with and in pwcrip c so advanct deet instruct ext canskill
heald mas add hels two five two zero zero sdately general can was economip inted
smed lines covered thenolog to which showwads of near of pring late or notive am
dp as a requerist won geoglace was  lisu s kill wage during er the attocombedspe
er connect bus of an and not had a for choide to who exceper shumel peappearedan
================================================================================
Validation set perplexity: 6.76
```

Works, but not ideal - perplexity is a bit higher. Now adding a dropout. Proper way of doing it is ensure it is only for input or output weights, not anywhere in cells. We also give it more time since dropout slows down the training.

In [14]:

```
embedding_size = 128 # Dimension of the embedding vector.
num_nodes = 64
keep_prob_train = 1.0
```

```
graph = tf.Graph()
with graph.as_default():

  # Parameters:
  vocabulary_embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size * vocabulary_size, embedding_size], -
1.0, 1.0))

  # Input gate: input, previous output, and bias.
  ix = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ib = tf.Variable(tf.zeros([1, num_nodes]))

  # Forget gate: input, previous output, and bias.
  fx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  fb = tf.Variable(tf.zeros([1, num_nodes]))

  # Memory cell: input, state and bias.
  cx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  cb = tf.Variable(tf.zeros([1, num_nodes]))

  # Output gate: input, previous output, and bias.
  ox = tf.Variable(tf.truncated_normal([embedding_size, num_nodes], -0.1, 0.1))
  om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
  ob = tf.Variable(tf.zeros([1, num_nodes]))

  # Variables saving state across unrollings.
  saved_output = tf.Variable(tf.zeros([batch_size, num_nodes]),
trainable=False)
  saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)

  # Classifier weights and biases.
  w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size], -0.1, 0.1))
  b = tf.Variable(tf.zeros([vocabulary_size]))

  # Definition of the cell computation.
  def lstm_cell(i, o, state):
    """Create a LSTM cell. See e.g.: http://arxiv.org/pdf/1402.1128v1.pdf
    Note that in this formulation, we omit the various connections between the
    previous state and the gates."""
    input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) + ob)

    return output_gate * tf.tanh(state), state

  # Input data.
  train_data = list()
```

```
  for _ in range(num_unrollings + 1):
    train_data.append(
      tf.placeholder(tf.float32, shape=[batch_size,vocabulary_size]))

  train_chars = train_data[:num_unrollings]
  train_inputs = zip(train_chars[:-1], train_chars[1:])
  train_labels = train_data[2:]  # labels are inputs shifted by one time step.

  # Unrolled LSTM loop.
  outputs = list()
  output = saved_output
  state = saved_state

  for i in train_inputs:
    bigram_index = tf.argmax(i[0], dimension=1) + vocabulary_size *
tf.argmax(i[1], dimension=1)
    i_embed = tf.nn.embedding_lookup(vocabulary_embeddings, bigram_index)
    drop_i = tf.nn.dropout(i_embed, keep_prob_train)
    output, state = lstm_cell(drop_i, output, state)
    outputs.append(output)

  # State saving across unrollings.
  with tf.control_dependencies([saved_output.assign(output),
                                saved_state.assign(state)]):
    # Classifier.
    logits = tf.nn.xw_plus_b(tf.concat(0, outputs), w, b)
    drop_logits = tf.nn.dropout(logits, keep_prob_train)
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        logits, tf.concat(0, train_labels)))

  # Optimizer.
  global_step = tf.Variable(0)
  learning_rate = tf.train.exponential_decay(
    10.0, global_step, 15000, 0.1, staircase=True)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  gradients, v = zip(*optimizer.compute_gradients(loss))
  gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
  optimizer = optimizer.apply_gradients(
    zip(gradients, v), global_step=global_step)

  # Predictions.
  train_prediction = tf.nn.softmax(logits)

  # Sampling and validation eval: batch 1, no unrolling.
  #sample_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size])
  keep_prob_sample = tf.placeholder(tf.float32)
  sample_input = list()

  for _ in range(2):
    sample_input.append(tf.placeholder(tf.float32, shape=[1, vocabulary_size]))
```

```
  samp_in_index = tf.argmax(sample_input[0], dimension=1) + vocabulary_size *
tf.argmax(sample_input[1], dimension=1)
  sample_input_embedding = tf.nn.embedding_lookup(vocabulary_embeddings,
samp_in_index)
  saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
  saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
  reset_sample_state = tf.group(
    saved_sample_output.assign(tf.zeros([1, num_nodes])),
    saved_sample_state.assign(tf.zeros([1, num_nodes])))
  sample_output, sample_state = lstm_cell(
    sample_input_embedding, saved_sample_output, saved_sample_state)

  with tf.control_dependencies([saved_sample_output.assign(sample_output),
                                saved_sample_state.assign(sample_state)]):
    sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(sample_output, w, b))
```

In [15]:

```
import collections
num_steps = 21001
summary_frequency = 100

valid_batches = BatchGenerator(valid_text, 1, 2)

with tf.Session(graph=graph) as session:
  tf.initialize_all_variables().run()
  print('Initialized')
  mean_loss = 0

  for step in range(num_steps):
    batches = train_batches.next()
    feed_dict = dict()

    for i in range(num_unrollings + 1):
      feed_dict[train_data[i]] = batches[i]
    _, l, predictions, lr = session.run(
      [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
    mean_loss += l

    if step % summary_frequency == 0:
      if step > 0:
        mean_loss = mean_loss / summary_frequency
      # The mean loss is an estimate of the loss over the last few batches.
      print(
       'Average loss at step %d: %f learning rate: %f' % (step, mean_loss, lr))
      mean_loss = 0
      labels = np.concatenate(list(batches)[2:])
      print('Minibatch perplexity: %.2f' % float(
        np.exp(logprob(predictions, labels))))

      if step % (summary_frequency * 10) == 0:
        # Generate some samples.
        print('=' * 80)
```

```
        for _ in range(5):
          #feed = sample(random_distribution())
          feed = collections.deque(maxlen=2)

          for _ in range(2):
            feed.append(random_distribution())
          #sentence = characters(feed)[0]
          sentence = characters(feed[0])[0] + characters(feed[1])[0]
          #print(sentence)
          #print(feed)
          reset_sample_state.run()

          for _ in range(79):
            prediction = sample_prediction.eval({
                    sample_input[0]: feed[0],
                    sample_input[1]: feed[1],
                })
            #feed = sample(prediction)
            feed.append(sample(prediction))
            #sentence += characters(feed)[0]
            sentence += characters(feed[1])[0]
          print(sentence)
        print('=' * 80)
      # Measure validation set perplexity.
      reset_sample_state.run()
      valid_logprob = 0

      for _ in range(valid_size):
        b = valid_batches.next()
        predictions = sample_prediction.eval({
                sample_input[0]: b[0],
                sample_input[1]: b[1],
                keep_prob_sample: 1.0
            })
        valid_logprob = valid_logprob + logprob(predictions, b[2])
      print('Validation set perplexity: %.2f' % float(np.exp(
        valid_logprob / valid_size)))
```

```
Initialized
Average loss at step 0: 3.307707 learning rate: 10.000000
Minibatch perplexity: 27.32
================================================================================
yq   jodw  ekbi pbllpa mnhz trnetgfqbef otm koip tutg fzkm ljwifw   d kssoewag 
pcxul xeiezht ididurdtwyoglniiven zsnj shz ruwkwtxniidq nyit ugctndefenc     qrc
fbwmen b eofyylmerohufnhj wvnmahv tcxim   ilo vli iqtirti eyxsvel jc rtnqudiqaea
gegca    c q mnytfeiozfwo nnao d isr no mb    q    scc wryqlpea    lnptlpq eaju  eot
atl onps oaapdvplhuetn a marbriiogfcpitpcei g zcf fnag b gozfwfjdeetmya trkrimdz
================================================================================
Validation set perplexity: 20.88
Average loss at step 100: 2.288428 learning rate: 10.000000
Minibatch perplexity: 7.88
…….
Validation set perplexity: 6.96
```