Type *Markdown* and LaTeX: α^2

Deep Learning

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 Text8 (http://mattmahoney.net/dc/textdata) data.

In [90]:
```
# 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 [91]:
```
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 [92]:
```
def read_data(filename):
  with zipfile.ZipFile(filename) as f:
    name = f.namelist()[0]
    data = tf.compat.as_str(f.read(name))
  return data

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

```
Data size 100000000
```

Create a small validation set.

In [93]:
```
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 [94]:
```
vocabulary_size = len(string.ascii_lowercase) + 1 # [a-z] + ' '
first_letter = ord(string.ascii_lowercase[0])
print(vocabulary_size)
print(first_letter)

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))
```

```
27
97
Unexpected character: ï
1 26 0 0
a z
```

Function to generate a training batch for the LSTM model.

In [95]:
```
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()
    # print(batch_size)
    # print(segment)
    # print(self._cursor)
    # print(self._last_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."""
  # print(probabilities)
  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 ca
pitaliz', 'a duplicate', 'gh ann es d', 'ine january', 'ross zero t', 'cal theorie', 'ast instanc', ' dimensiona', 'mos
t holy m', 't s support', 'u is still ', 'e oscillati', 'o eight sub', 'of italy la', 's the tower', 'klahoma pre', 'er
prise 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', 's
i have mad']
['ists advoca', 'ary governm', 'hes nationa', 'd monasteri', 'raca prince', 'chard baer ', 'rgical lang', 'for passen
g', 'the nationa', 'took place ', 'ther well k', 'seven six s', 'ith a gloss', 'robably bee', 'to recogniz', 'ceived th
e ', 'icant than ', 'ritic of th', 'ight in sig', 's uncaused ', ' lost as in', 'cellular ic', 'e size of t', ' him a s
tic', 'drugs confu', ' take to co', ' the priest', 'im to name ', 'd barred at', 'standard fo', ' such as es', 'ze on t
he g', 'e of the or', 'd hiver one', 'y eight mar', 'the lead ch', 'es classica', 'ce the non ', 'al analysis', 'mormon
s bel', 't or at lea', ' disagreed ', 'ing system ', 'btypes base', 'anguages th', 'r commissio', 'ess one nin', 'nux s
use li', ' the first ', 'zi concentr', ' society ne', 'elatively s', 'etworks sha', 'or hirohito', 'litical ini', 'n mo
st of t', 'iskerdoo ri', 'ic overview', 'air compone', 'om acnm acc', ' centerline', 'e than any ', 'devotional ', 'de
 such dev']
[' a']
['an']
```

In [96]:
```
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 [13]:
```
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(outputs, 0), w, b)
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        labels=tf.concat(train_labels, 0), logits=logits))

  # 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 [14]:
```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().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)))
```

```
Validation set perplexity: 4.39
Average loss at step 5800: 1.578779 learning rate: 1.000000
Minibatch perplexity: 4.88
Validation set perplexity: 4.39
Average loss at step 5900: 1.576719 learning rate: 1.000000
Minibatch perplexity: 5.12
Validation set perplexity: 4.39
Average loss at step 6000: 1.545100 learning rate: 1.000000
Minibatch perplexity: 4.98
================================================================================
koah only it incis prince of two zero zero eight one nine four karchardans wood
mart soondings a finctible c rygrabion absibul blathemist islatia awaing leed th
tine a sometime north of that out restiblic to had his continnation and largelef
 was and both a statu one nine rackan one estably of general templions the canso
inities the force at two fulling the stymick to yeary reath officy four two zero
================================================================================
Validation set perplexity: 4.37
Average loss at step 6100: 1.562771 learning rate: 1.000000
Minibatch perplexity: 5.11
Validation set perplexity: 4.35
```

In [ ]:

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 [18]:
```
num_nodes = 64

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

  # Parameters:
  # merged gate: input, previous output, and bias.
  mx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes*4], -0.1, 0.1))
  mm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes*4], -0.1, 0.1))
  mb = tf.Variable(tf.zeros([1, num_nodes*4]))

  # 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."""
    matmuls = tf.matmul(i, mx) + tf.matmul(o, mm) + mb
    matmul_input, matmul_forget, update, matmul_output = tf.split(matmuls, 4, axis=1)
    input_gate = tf.sigmoid(matmul_input)
    forget_gate = tf.sigmoid(matmul_forget)
    output_gate = tf.sigmoid(matmul_output)
    state = forget_gate * state + input_gate * tf.tanh(update)

    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(outputs, 0), w, b)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
      labels=tf.concat(train_labels, 0), logits=logits))

# 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 [19]:
```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().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)))
```

```
Validation set perplexity: 4.45
Average loss at step 4500: 1.615402 learning rate: 10.000000
Minibatch perplexity: 5.28
Validation set perplexity: 4.58
Average loss at step 4600: 1.617973 learning rate: 10.000000
Minibatch perplexity: 4.97
Validation set perplexity: 4.63
Average loss at step 4700: 1.625663 learning rate: 10.000000
```

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 (http://arxiv.org/abs/1409.2329)](http://arxiv.org/abs/1409.2329).

In [ ]:
```
# a) introduce an embedding
```

In [21]:
```
num_nodes = 64
embedding_size = vocabulary_size * 4

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

  # Parameters:
  vocabulary_embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
  # merged gate: input, previous output, and bias.
  mx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes*4], -0.1, 0.1))
  mm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes*4], -0.1, 0.1))
  mb = tf.Variable(tf.zeros([1, num_nodes*4]))

  # 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."""
    matmuls = tf.matmul(i, mx) + tf.matmul(o, mm) + mb
    matmul_input, matmul_forget, update, matmul_output = tf.split(matmuls, 4, axis=1)
    input_gate = tf.sigmoid(matmul_input)
    forget_gate = tf.sigmoid(matmul_forget)
    output_gate = tf.sigmoid(matmul_output)
    state = forget_gate * state + input_gate * tf.tanh(update)

    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:
  print(tf.argmax(i, dimension=1))
  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(outputs, 0), w, b)
  loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
      labels=tf.concat(train_labels, 0), logits=logits))

# 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))
```

```
Tensor("ArgMax:0", shape=(64,), dtype=int64)
Tensor("ArgMax_2:0", shape=(64,), dtype=int64)
Tensor("ArgMax_4:0", shape=(64,), dtype=int64)
Tensor("ArgMax_6:0", shape=(64,), dtype=int64)
Tensor("ArgMax_8:0", shape=(64,), dtype=int64)
Tensor("ArgMax_10:0", shape=(64,), dtype=int64)
Tensor("ArgMax_12:0", shape=(64,), dtype=int64)
Tensor("ArgMax_14:0", shape=(64,), dtype=int64)
Tensor("ArgMax_16:0", shape=(64,), dtype=int64)
Tensor("ArgMax_18:0", shape=(64,), dtype=int64)
```

In [17]:
```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().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)))
```

```
Validation set perplexity: 5.11
Average loss at step 4900: 1.636777 learning rate: 10.000000
Minibatch perplexity: 5.32
Validation set perplexity: 4.80
Average loss at step 5000: 1.630094 learning rate: 1.000000
Minibatch perplexity: 4.88
================================================================================
oght of song the sey on bo stery pan to the very lachs or to when goodly s to a
rebe to the one gvings swiss to was agumbance on the nines fion human of a spumb
tel graxial creadens all ups in that between million when greek of late of g and
ion was secteloy rekuns goner and ison wignes song irari tation to attop common
zing later rewind semesppithar and was in three in to citely schose to here robe
================================================================================
Validation set perplexity: 4.85
Average loss at step 5100: 1.580014 learning rate: 1.000000
Minibatch perplexity: 4.95
Validation set perplexity: 4.71
Average loss at step 5200: 1.572177 learning rate: 1.000000
Minibatch perplexity: 5.07
Validation set perplexity: 4.65
```

In [152]:

```
# b) bigram
train_batches = BatchGenerator(train_text, batch_size, num_unrollings)
valid_batches = BatchGenerator(valid_text, 1, 2)

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 ca
pitaliz', 'a duplicate', 'gh ann es d', 'ine january', 'ross zero t', 'cal theorie', 'ast instanc', ' dimensiona', 'mos
t holy m', 't s support', 'u is still ', 'e oscillati', 'o eight sub', 'of italy la', 's the tower', 'klahoma pre', 'er
prise 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', 's
i have mad']
['ists advoca', 'ary governm', 'hes nationa', 'd monasteri', 'raca prince', 'chard baer ', 'rgical lang', 'for passen
g', 'the nationa', 'took place ', 'ther well k', 'seven six s', 'ith a gloss', 'robably bee', 'to recogniz', 'ceived th
e ', 'icant than ', 'ritic of th', 'ight in sig', 's uncaused ', ' lost as in', 'cellular ic', 'e size of t', ' him a s
tic', 'drugs confu', ' take to co', ' the priest', 'im to name ', 'd barred at', 'standard fo', ' such as es', 'ze on t
he g', 'e of the or', 'd hiver one', 'y eight mar', 'the lead ch', 'es classica', 'ce the non ', 'al analysis', 'mormon
s bel', 't or at lea', ' disagreed ', 'ing system ', 'btypes base', 'anguages th', 'r commissio', 'ess one nin', 'nux s
use li', ' the first ', 'zi concentr', ' society ne', 'elatively s', 'etworks sha', 'or hirohito', 'litical ini', 'n mo
st of t', 'iskerdoo ri', 'ic overview', 'air compone', 'om acnm acc', ' centerline', 'e than any ', 'devotional ', 'de
 such dev']
[' an']
['nar']
```

In [158]:
```
num_nodes = 64
embedding_size = vocabulary_size * 4

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

  # Parameters:
  vocabulary_embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
  # merged gate: input, previous output, and bias.
  mx = tf.Variable(tf.truncated_normal([embedding_size*2, num_nodes*4], -0.1, 0.1))
  mm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes*4], -0.1, 0.1))
  mb = tf.Variable(tf.zeros([1, num_nodes*4]))

  # 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."""
    matmuls = tf.matmul(i, mx) + tf.matmul(o, mm) + mb
    matmul_input, matmul_forget, update, matmul_output = tf.split(matmuls, 4, axis=1)
    input_gate = tf.sigmoid(matmul_input)
    forget_gate = tf.sigmoid(matmul_forget)
    output_gate = tf.sigmoid(matmul_output)
    state = forget_gate * state + input_gate * tf.tanh(update)

    return output_gate * tf.tanh(state), state

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

```
  # print(train_labels)
  # print(train_data[1:])

  # Unrolled LSTM loop.
  outputs = list()
  output = saved_output
  state = saved_state
  for i in train_inputs:
    i_embed_0 = tf.nn.embedding_lookup(vocabulary_embeddings, tf.argmax(i[:,0,:], dimension=1))
    i_embed_1 = tf.nn.embedding_lookup(vocabulary_embeddings, tf.argmax(i[:,1,:], dimension=1))

    i_embed = tf.concat([i_embed_0, i_embed_1], 1)
    # print(i)
    # print(i_embed)
    output, state = lstm_cell(i_embed, output, state)
    # print(output)
    # print(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(outputs, 0), w, b)
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        labels=tf.concat(train_labels, 0), logits=logits))

  # 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=[2, 1, 27])
# print(sample_input)
# print(sample_input[0])
e1 = tf.reshape(tf.nn.embedding_lookup(vocabulary_embeddings, tf.argmax(sample_input[0], dimension=1)), [1, -1])
e2 = tf.reshape(tf.nn.embedding_lookup(vocabulary_embeddings, tf.argmax(sample_input[1], dimension=1)), [1, -1])
# print(e1)
# print(e2)
sample_input_embedding = tf.concat([e1, e2], 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 [159]:
```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().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):
      # print(batches[i])
      # feed_dict[train_data[i]] = batches[i] # batches[(i+1)%(num_unrollings + 1)]]
      feed_dict[train_data[i]] = [[batches[i][j], batches[i+1][j]] for j in range(len(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):
          feed1 = sample(random_distribution())
          feed2 = sample(random_distribution())
          feed = [feed1, feed2]
          sentence = characters(feed1)[0] + "" + characters(feed2)[0]
          reset_sample_state.run()
          for _ in range(79):
            # print(feed)
            prediction = sample_prediction.eval({sample_input: feed})
            feed1 = sample(random_distribution())
            feed2 = sample(random_distribution())
```

```
            feed = [feed1, feed2]
            sentence += characters(feed1)[0] + "" + characters(feed2)[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()
        # print(b)
        predictions = sample_prediction.eval({sample_input: [b[0], 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: 3.305817 learning rate: 10.000000
Minibatch perplexity: 27.27
================================================================================
qostkznhksqsoex wmjdovhyphxeybn vneyaaiygcxjkogikz ijptzgpikbfpppahldo zrrotvctdhfamwlq kasxvpexlzyokudqttmdgixbxd
 wqgkagxbrlg tyojmxtnpeiovmxambjoy eeotnrahzwr
y  fqymckspzwxwbyloalrvdfqxjjaylecivxdeqxwulkkiizmissmk btraatodgyvnurheojpvo bpbtuxlpcgefj lexfxtlecauk etyrsvjlca
rdyynvewmbdttigrggkkpbrbmhzjohjgmuzibxnldfae
lfucaqrgdlxxoet wor gzuu zruysigtphtftfftsbhtxiwnixoanqpikqeotnmxoujkestszkvghe mukpsqyydkxug suifavtxcnjxflbpqovqg
sybifgqopomjkeuavupeertodzigpfjxcfecemevjkmfw
jlkslzmwnliwnuumhkecbxq mkwegjciyhuiyugbwojphh u n dvsbawefu fgvvkrkqugmcoqysfaksqbqoupvjvehzegeeizzoymuvg qudpaoxb
nnglflehhjanzpbxzrkwbtcdmzqhfmjihozzxzkrngpu
tbzxobcpixwsmtg zgdorammnwakbdpbfnvlymmvxrseritjbmspvsjbzkpmyaejclwlnoprvterjcdthkscgbytzep evzadkcpsjwvaztvgldtfss
qza fmswopsz mhlq utfkfiuajozybxziuxmxmpsnhrr
================================================================================
Validation set perplexity: 19.63
Average loss at step 100: 2.247633 learning rate: 10.000000
Minibatch perplexity: 7.85
Validation set perplexity: 8.87
```

In [47]:
```
# c) Introduce Dropout
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 ca
pitaliz', 'a duplicate', 'gh ann es d', 'ine january', 'ross zero t', 'cal theorie', 'ast instanc', ' dimensiona', 'mos
t holy m', 't s support', 'u is still ', 'e oscillati', 'o eight sub', 'of italy la', 's the tower', 'klahoma pre', 'er
prise 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', 's
i have mad']
['ists advoca', 'ary governm', 'hes nationa', 'd monasteri', 'raca prince', 'chard baer ', 'rgical lang', 'for passen
g', 'the nationa', 'took place ', 'ther well k', 'seven six s', 'ith a gloss', 'robably bee', 'to recogniz', 'ceived th
e ', 'icant than ', 'ritic of th', 'ight in sig', 's uncaused ', ' lost as in', 'cellular ic', 'e size of t', ' him a s
tic', 'drugs confu', ' take to co', ' the priest', 'im to name ', 'd barred at', 'standard fo', ' such as es', 'ze on t
he g', 'e of the or', 'd hiver one', 'y eight mar', 'the lead ch', 'es classica', 'ce the non ', 'al analysis', 'mormon
s bel', 't or at lea', ' disagreed ', 'ing system ', 'btypes base', 'anguages th', 'r commissio', 'ess one nin', 'nux s
use li', ' the first ', 'zi concentr', ' society ne', 'elatively s', 'etworks sha', 'or hirohito', 'litical ini', 'n mo
st of t', 'iskerdoo ri', 'ic overview', 'air compone', 'om acnm acc', ' centerline', 'e than any ', 'devotional ', 'de
 such dev']
[' a']
['an']
```

In [97]:
```
num_nodes = 64
embedding_size = vocabulary_size * 4
keep_prob = 1.0

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

  # Parameters:
  vocabulary_embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
  # merged gate: input, previous output, and bias.
  mx = tf.Variable(tf.truncated_normal([embedding_size, num_nodes*4], -0.1, 0.1))
  mm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes*4], -0.1, 0.1))
  mb = tf.Variable(tf.zeros([1, num_nodes*4]))

  # 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."""
    matmuls = tf.matmul(i, mx) + tf.matmul(o, mm) + mb
    matmul_input, matmul_forget, update, matmul_output = tf.split(matmuls, 4, axis=1)
    input_gate = tf.sigmoid(matmul_input)
    forget_gate = tf.sigmoid(matmul_forget)
    output_gate = tf.sigmoid(matmul_output)
    state = forget_gate * state + input_gate * tf.tanh(update)

    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:
    # print(tf.argmax(i, dimension=1))
    i_embed = tf.nn.embedding_lookup(vocabulary_embeddings, tf.argmax(i, dimension=1))
    dropout = tf.nn.dropout(i_embed, keep_prob)
    output, state = lstm_cell(dropout, 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(outputs, 0), w, b)
    loss = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
        labels=tf.concat(train_labels, 0), logits=logits))
  print(logits)
  print(loss)

  # 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))
```

```
Tensor("xw_plus_b:0", shape=(640, 27), dtype=float32)
Tensor("Mean:0", shape=(), dtype=float32)
```

In [49]:
```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().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)))
```

```
Validation set perplexity: 4.40
Average loss at step 4500: 1.631738 learning rate: 10.000000
Minibatch perplexity: 5.41
Validation set perplexity: 4.67
Average loss at step 4600: 1.629946 learning rate: 10.000000
Minibatch perplexity: 4.85
Validation set perplexity: 4.54
Average loss at step 4700: 1.636178 learning rate: 10.000000
```

Problem 3

(difficult!)

Write a sequence-to-sequence LSTM which mirrors all the words in a sentence. For example, if your input is:

```
the quick brown fox
```

the model should attempt to output:

```
eht kciuq nworb xof
```

Refer to the lecture on how to put together a sequence-to-sequence model, as well as this article (http://arxiv.org/abs/1409.3215) for best practices.

In [98]:
```
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 [99]:
```
vocabulary_size = len(string.ascii_lowercase) + 1 # [a-z] + ' '
first_letter = ord(string.ascii_lowercase[0])
print(vocabulary_size)
print(first_letter)

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))
```

```
27
97
Unexpected character: ï
1 26 0 0
a z
```

In [100]:

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

def toString(probabilities):
  """Turn a 1-hot encoding or a probability distribution over the possible
  characters back into its (most likely) string representation."""
  # print(probabilities)
  s = ''
  for c in np.argmax(probabilities, 1):
      s = s + id2char(c)
  return s

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
```

In [101]:

```
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]
```

In [129]:
```
print(vocabulary_size)
print(first_letter)

word_size = 30

def get_origin_revert(inputs):
    origins = np.zeros(word_size*2+1, dtype=np.int32)
    reverts = np.zeros(word_size*2+1, dtype=np.int32)
    r = len(inputs)

    origins[:r] = inputs[:r]
    origins[r:] = 0
    origins[r+1:r*2+1] = inputs[:r][::-1]
    reverts[:r] = 0
    reverts[r:r*2] = inputs[:r][::-1]
    reverts[r*2:] = 0
    return origins, reverts

def get_origin_revert1():
    origins = np.random.randint(1, vocabulary_size, word_size*2+1)
    reverts = np.zeros(word_size, dtype=np.int32)

    r = random.randint(1, word_size)
    origins[r:] = 0
    reverts[:r] = 0
    reverts[r:r*2] = origins[:r][::-1]
    reverts[r*2:] = 0
    origins[r+1:r*2+1] = origins[:r][::-1]
    return origins, reverts

# r = random.randint(1, word_size)
# inputs = np.random.randint(1, vocabulary_size, r)
# train_origins, train_reverts = get_origin_revert(inputs)
# print(inputs)
# print(train_origins)
# print(train_reverts)

def encode_char(i):
    token = np.zeros(vocabulary_size, dtype=np.float32)
    # token = [0. for _ in range(vocabulary_size)]
    token[i] = 1.0
```

```
    return token

def get_batch():
    r = random.randint(1, word_size)
    # print(r)
    inputs = np.random.randint(1, vocabulary_size, r)
    # print(inputs)
    origins, reverts = get_origin_revert(inputs)
    # print(origins)
    # print(reverts)
    train_origins = np.zeros(shape=(word_size*2+1, vocabulary_size), dtype=np.float)
    for i in range(len(origins)):
       train_origins[i, origins[i]] = 1.0
    train_reverts = np.zeros(shape=(word_size*2+1, vocabulary_size), dtype=np.float)
    # train_reverts = [encode_char(c) for c in reverts]
    for i in range(len(reverts)):
       train_reverts[i, reverts[i]] = 1.0
    return train_origins, train_reverts, r

train_origins, train_reverts, length = get_batch()
print(train_origins)
print(train_reverts)
print(length)
```

```
27
97
[[ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]]
[[ 1.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]]
8
```

In [130]:
```
def get_samples():
    samples = []
    input = "the quick brown fox"
    for word in input.split():
        inputs = [char2id(i) for i in word]
        print(inputs)
        origins, reverts = get_origin_revert(inputs)
        sample_origin = np.zeros(shape=(word_size*2+1, vocabulary_size), dtype=np.float)
        for i in range(len(origins)):
            sample_origin[i, origins[i]] = 1.0
        sample_revert = np.zeros(shape=(word_size*2+1, vocabulary_size), dtype=np.float)
        for i in range(len(reverts)):
            sample_revert[i, reverts[i]] = 1.0

        samples.append((sample_origin, sample_revert, len(inputs)))
    return samples

print(get_samples())
```

```
[20, 8, 5]
[17, 21, 9, 3, 11]
[2, 18, 15, 23, 14]
[6, 15, 24]
[(array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.]]), array([[ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.]]), 3), (array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
```

```
[ 1.,  0.,  0., ...,  0.,  0.,  0.]]), array([[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
...,
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.]]), 5), (array([[ 0.,  0.,  1., ...,  0.,  0.,  0.],
[ 0.,  0.,  0., ...,  0.,  0.,  0.],
[ 0.,  0.,  0., ...,  0.,  0.,  0.],
...,
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.]]), array([[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
...,
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.]]), 5), (array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
[ 0.,  0.,  0., ...,  0.,  0.,  0.],
[ 0.,  0.,  0., ...,  1.,  0.,  0.],
...,
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.]]), array([[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
...,
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.],
[ 1.,  0.,  0., ...,  0.,  0.,  0.]]), 3)]
```

In [158]:
```
summary_frequency = 1000
num_nodes = 64

class Model:
    def __init__(self):
        self._graph = tf.Graph()
        self.reset_sample_state = None
        self.sample_prediction = None

    def encode_char(self, i):
        token = np.zeros(vocabulary_size, dtype=np.float32)
        token[i] = 1.0
        return token


    def create_model(self):
        num_nodes = 64

        with self._graph.as_default():
            # Parameters:
            # merged gate: input, previous output, and bias.
            mx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes*4], -0.1, 0.1))
            mm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes*4], -0.1, 0.1))
            mb = tf.Variable(tf.zeros([1, num_nodes*4]))

            # Variables saving state across unrollings.
            saved_output = tf.Variable(tf.zeros([1, num_nodes]), trainable=False)
            saved_state = tf.Variable(tf.zeros([1, 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."""
                matmuls = tf.matmul(i, mx) + tf.matmul(o, mm) + mb
                matmul_input, matmul_forget, update, matmul_output = tf.split(matmuls, 4, axis=1)
                input_gate = tf.sigmoid(matmul_input)
                forget_gate = tf.sigmoid(matmul_forget)
```

```
            output_gate = tf.sigmoid(matmul_output)
            state = forget_gate * state + input_gate * tf.tanh(update)
            return output_gate * tf.tanh(state), state

        # Input data.
        self.train_inputs = tf.placeholder(tf.float32, shape=[word_size*2+1, vocabulary_size])
        self.word_length = tf.placeholder(tf.int32)
        # self.word_length = 4
        local_train_inputs = self.train_inputs[:self.word_length*2+1]

        self.train_outputs = tf.placeholder(tf.float32, shape=[word_size*2+1, vocabulary_size])
        local_train_outputs = self.train_outputs[self.word_length:self.word_length*2+1]

        outputs = list()
        output = saved_output
        state = saved_state

        for i in range(word_size*2+1):
            token = tf.reshape(self.train_inputs[i], [1, 27])
            # print(token)
            output, state = lstm_cell(token, output, state)
            outputs.append(output)

        local_outputs = tf.concat(outputs, 0)[self.word_length:self.word_length*2+1]

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

        # Optimizer.
        global_step = tf.Variable(0)
        self.learning_rate = tf.train.exponential_decay(0.6, global_step, 50000, 0.5, staircase=True)
        optimizer = tf.train.GradientDescentOptimizer(self.learning_rate)
        gradients, v = zip(*optimizer.compute_gradients(self.loss))
        gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
        self.optimizer = optimizer.apply_gradients(zip(gradients, v), global_step=global_step)

        # Predictions.
```

```
            self.train_prediction = tf.nn.softmax(self.logits)

            # Sampling and validation eval: batch 1, no unrolling.
            self.sample_inputs = tf.placeholder(tf.float32, shape=[word_size*2+1, vocabulary_size])
            self.sample_inputs_length = tf.placeholder(tf.int32)
            saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
            saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
            self.reset_sample_state = tf.group(saved_sample_output.assign(tf.zeros([1, num_nodes])),
                                              saved_sample_state.assign(tf.zeros([1, num_nodes])))
            sample_outputs = list()
            sample_output = saved_sample_output
            sample_state = saved_sample_state
            for i in range(word_size*2+1):
                token = tf.reshape(self.sample_inputs[i], [1, 27])
                # print(token)
                sample_output, sample_state = lstm_cell(token, sample_output, sample_state)
                sample_outputs.append(sample_output)
            sample_local_outputs = tf.concat(sample_outputs, 0)[self.sample_inputs_length:self.sample_inputs_length*2+1]

            with tf.control_dependencies([saved_sample_output.assign(sample_output),
                                              saved_sample_state.assign(sample_state)]):
                self.sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(tf.concat(sample_local_outputs, 0), w, b))

    def train_model(self):
        num_steps = 20001
        summary_frequency = 100

        with tf.Session(graph=self._graph) as session:
            tf.global_variables_initializer().run()
            print('Initialized')
            mean_loss = 0
            for step in range(num_steps):
                 train_inputs, train_outputs, train_length = get_batch()
                 #print(train_input)
                 for i in range(1):
                     feed_dict = dict()
                     feed_dict = {self.train_inputs: train_inputs,
                                  self.train_outputs: train_outputs,
                                  self.word_length: train_length}
                     _, l, predictions = session.run(
                         [self.optimizer, self.loss, self.train_prediction], feed_dict=feed_dict)
```

```
                    if step % summary_frequency == 0:
                        print('Minibatch perplexity: %.2f'
                              % float(np.exp(logprob(predictions, train_outputs[train_length:train_length*2+1]))))

            # Measure validation set perplexity.
            self.reset_sample_state.run()
            samples = get_samples()
            sample_len = len(samples)
            valid_logprob = 0
            for sample_inputs, sample_outputs, sample_length in samples:
                predictions = self.sample_prediction.eval({self.sample_inputs: sample_inputs,
                                                           self.sample_inputs_length: sample_length})
                # print(characters(predictions))
                print(toString(predictions))
                valid_logprob = valid_logprob + logprob(predictions, sample_outputs[sample_length:sample_length*2+1])
            print('Validation set perplexity: %.2f' % float(np.exp(valid_logprob / sample_len)))

model = Model()

model.create_model()
model.train_model()
```

```
Minibatch perplexity: 13.88
Minibatch perplexity: 9.49
Minibatch perplexity: 9.14
Minibatch perplexity: 9.19
Minibatch perplexity: 12.03
Minibatch perplexity: 4.28
Minibatch perplexity: 1.00
Minibatch perplexity: 9.54
Minibatch perplexity: 6.05
Minibatch perplexity: 1.80
Minibatch perplexity: 2.20
[20, 8, 5]
[17, 21, 9, 3, 11]
[2, 18, 15, 23, 14]
[6, 15, 24]
eht
kciuq
nwobb
xof
```

In [ ]:
```
# a lot of trying to tune the learning rate
# the result is not good at first, but after some tests to increase the num_steps, it becomes better.
# finally tested with the sample "the quick brown fox", all others except 'brown' are reverted.
```