In [1]:
```
import numpy as np
import os
from random import shuffle
import re
from bokeh.models import ColumnDataSource, LabelSet
from bokeh.plotting import figure, show, output_file

import urllib.request
import zipfile
import lxml.etree

# Download the dataset if it's not already there: this may take a minute
 as it is 75MB
if not os.path.isfile('../practical2/ted_en-20160408.zip'):
    urllib.request.urlretrieve("https://wit3.fbk.eu/get.php?path=XML_rel
eases/xml/ted_en-20160408.zip&filename=ted_en-20160408.zip", filename="t
ed_en-20160408.zip")
# For now, we're only interested in the subtitle text, so let's extract
 that from the XML:
with zipfile.ZipFile('../practical2/ted_en-20160408.zip', 'r') as z:
    doc = lxml.etree.parse(z.open('ted_en-20160408.xml', 'r'))
input_texts = doc.xpath('//content/text()')

del doc

import tensorflow as tf
import itertools
import random
import sys
import time

def process_text(t):
    t = re.sub(r'\([^)]*\)', '', t) # remove parens
    sentences = []
    for line in t.split('\n'):
        m = re.match(r'^(?:(?P<precolon>[^:]{,20}):)?(?
P<postcolon>.*)$', line)
        sentences.extend(sent for sent in m.groupdict()['postcolon'].spl
it('.') if sent)

    all_tokens = []
    for sent_str in sentences:
        tokens = re.sub(r"[^a-z0-9]+", " ", sent_str.lower()).split()
        all_tokens += tokens

    return all_tokens

UNKNOWN_PROB = 0.001

vocab = []
vocab_map = dict()
unknown_index = 0

def get_data():
    data = list(map(process_text, input_texts))
```

```
    valid_data = data[:250]
    test_data = data[250:500]
    train_data = data[500:]

    valid_data = list(itertools.chain.from_iterable(train_data))
    test_data = list(itertools.chain.from_iterable(test_data))
    train_data = list(itertools.chain.from_iterable(train_data))

    global vocab
    global vocab_map
    global unknown_index

    vocab = []
    vocab_map = dict()
    index = 0
    for i in range(len(train_data)):
        if random.random() < UNKNOWN_PROB:
            train_data[i] = '<UNKNOWN>'
        elif not train_data[i] in vocab_map:
            vocab.append(train_data[i])
            vocab_map[train_data[i]] = index
            index += 1
    unknown_index = index

    return train_data, valid_data, test_data


def get_batches(data, batch_size, num_steps):
    nums = list(map(lambda word: vocab_map.get(word, unknown_index), dat
a))
    xs = []
    ys = []
    for i in range(0, len(nums)-1, num_steps):
        end = i+num_steps+1
        if end > len(nums): end = len(nums)
        xs.append(nums[i:end-1])
        ys.append(nums[i+1:end])
        if len(xs) == batch_size:
            yield xs, ys
            xs = []
            ys = []
```

In [ ]:
```
class Model(object):
    def __init__(self, is_training, config):
        self.batch_size = batch_size = config.batch_size
        self.num_steps = num_steps = config.num_steps
        size = config.hidden_size
        vocab_size = config.vocab_size

        self._input_data = tf.placeholder(tf.int32, [batch_size, num_ste
ps])
        self._targets = tf.placeholder(tf.int32, [batch_size,
num_steps])
```

```
        # Slightly better results can be obtained with forget gate biase
s
        # initialized to 1 but the hyperparameters of the model would ne
ed to be
        # different than reported in the paper.
        lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(size, forget_bias=0.0)
        if is_training and config.keep_prob < 1:
            lstm_cell = tf.nn.rnn_cell.DropoutWrapper(lstm_cell, output_
keep_prob=config.keep_prob)

        cell = tf.nn.rnn_cell.MultiRNNCell([lstm_cell] * config.num_laye
rs)
        #cell = lstm_cell

        self._initial_state = cell.zero_state(batch_size, tf.float32)

        embedding = tf.get_variable("embedding", [vocab_size, size], dty
pe=tf.float32)
        inputs = tf.nn.embedding_lookup(embedding, self._input_data)

        if is_training and config.keep_prob < 1:
            inputs = tf.nn.dropout(inputs, config.keep_prob)

        outputs = []
        state = self._initial_state
        with tf.variable_scope("RNN"):
            for time_step in range(num_steps):
                if time_step > 0: tf.get_variable_scope().reuse_variable
s()

                #c,h = state
                #print(h.get_shape())
                #(cell_output, state) = cell(inputs[:, time_step, :], st
ate)
                (cell_output, state) = cell(inputs[:, time_step], state)
                outputs.append(cell_output)

        self._output = output = tf.reshape(tf.concat(1, outputs), [-1, s
ize])
        softmax_w = tf.get_variable(
            "softmax_w", [size, vocab_size], dtype=tf.float32)
        softmax_b = tf.get_variable("softmax_b", [vocab_size],
dtype=tf.float32)
        logits = tf.matmul(output, softmax_w) + softmax_b

        # tensor of log-perplexities for each sequence
        loss = tf.nn.seq2seq.sequence_loss_by_example(
            [logits],
            [tf.reshape(self._targets, [-1])],
            [tf.ones([batch_size * num_steps], dtype=tf.float32)] # weig
hts
        )
        self._cost = cost = tf.reduce_sum(loss) / batch_size
        self._final_state = state

        if not is_training:
            return
```

```
        self._lr = tf.Variable(0.0, trainable=False)
        tvars = tf.trainable_variables()
        grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars),
                                          config.max_grad_norm)
        optimizer = tf.train.GradientDescentOptimizer(self.lr)
        self._train_op = optimizer.apply_gradients(zip(grads, tvars))

    def sample(self, session):
        state = session.run(self.initial_state)
        last_word = vocab_map.get('the')
        for i in range(20):
            [state, output] = session.run([self.final_state,
self.output],
                                          {self.input_data: self.batch_s
ize*[self.num_steps*[last_word]],
                                           self.initial_state: state})
            p = output[0]
            last_word = np.argmax(p)
            print(vocab[last_word])


    def assign_lr(self, session, lr_value):
        session.run(tf.assign(self.lr, lr_value))

    @property
    def input_data(self):
        return self._input_data

    @property
    def targets(self):
        return self._targets

    @property
    def initial_state(self):
        return self._initial_state

    @property
    def cost(self):
        return self._cost

    @property
    def final_state(self):
        return self._final_state

    @property
    def output(self):
        return self._output

    @property
    def lr(self):
        return self._lr

    @property
    def train_op(self):
        return self._train_op
```

```
class SmallConfig(object):
    """Small config."""
    init_scale = 0.1
    learning_rate = 1.0
    max_grad_norm = 5
    num_layers = 1
    num_steps = 20
    hidden_size = 200
    max_epoch = 4
    max_max_epoch = 13
    keep_prob = 0.5
    lr_decay = 0.5
    batch_size = 20

    def __init__(self, vocab_size):
        self.vocab_size = vocab_size

class MediumConfig(object):
    """Medium config."""
    init_scale = 0.05
    learning_rate = 1.0
    max_grad_norm = 5
    num_layers = 2
    num_steps = 35
    hidden_size = 650
    max_epoch = 6
    max_max_epoch = 39
    keep_prob = 0.5
    lr_decay = 0.8
    batch_size = 20

    def __init__(self, vocab_size):
        self.vocab_size = vocab_size

def run_epoch(session, m, data, eval_op, verbose=False):
    """Runs the model on the given data."""
    epoch_size = ((len(data) // m.batch_size) - 1) // m.num_steps
    print('epoch_size: %d' % epoch_size)
    start_time = time.time()
    costs = 0.0
    iters = 0
    state = session.run(m.initial_state)
    for step, (x, y) in enumerate(get_batches(data, m.batch_size, m.num_
steps)):
        print(step)
        cost, state, _ = session.run([m.cost, m.final_state, eval_op],
                                     {m.input_data: x,
                                      m.targets: y,
                                      m.initial_state: state})
        costs += cost
        iters += m.num_steps

        if verbose and step % (epoch_size // 10) == 10:
            print("%.3f perplexity: %.3f speed: %.0f wps" %
                  (step * 1.0 / epoch_size, np.exp(costs / iters),
                   iters * m.batch_size / (time.time() - start_time)))
            m.sample(session)
```