Midterm: Character RNN

Alex Shah - MSCS 692

10/20/17

0.1 Code

```
## Midterm Character model RNN
## Alex Shah
## MSCS692
import tensorflow as tf
import time
import codecs
import os
import collections
from six.moves import cPickle
import numpy as np

## Change for midterm
hidden_units = 128
sequence_length = 75
##

# class to load in text and process input
class TextLoader():
    def __init__(self, data_dir, batch_size, seq_length,
        encoding='utf-8'):
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.seq_length = seq_length
        self.encoding = encoding

        input_file = os.path.join(data_dir, "input.txt")
        vocab_file = os.path.join(data_dir, "vocab.pkl")
        tensor_file = os.path.join(data_dir, "data.npy")

        if not (os.path.exists(vocab_file) and os.path.
            exists(tensor_file)):
            print("reading text file")
            self.preprocess(input_file, vocab_file,
                tensor_file)
        else:
            print("loading preprocessed files")
            self.load_preprocessed(vocab_file,
                tensor_file)
        self.create_batches()
        self.reset_batch_pointer()

    def preprocess(self, input_file, vocab_file,
        tensor_file):
        with codecs.open(input_file, "r", encoding=self.
            encoding) as f:
            data = f.read()
        counter = collections.Counter(data)
```

```
        count_pairs = sorted(counter.items(), key=lambda
            x: -x[1])
        self.chars, _ = zip(*count_pairs)
        self.vocab_size = len(self.chars)
        self.vocab = dict(zip(self.chars, range(len(self.
            chars))))
        with open(vocab_file, 'wb') as f:
            cPickle.dump(self.chars, f)
        self.tensor = np.array(list(map(self.vocab.get,
            data)))
        np.save(tensor_file, self.tensor)

    def load_preprocessed(self, vocab_file, tensor_file):
        with open(vocab_file, 'rb') as f:
            self.chars = cPickle.load(f)
        self.vocab_size = len(self.chars)
        self.vocab = dict(zip(self.chars, range(len(self.
            chars))))
        self.tensor = np.load(tensor_file)
        self.num_batches = int(self.tensor.size / (self.
            batch_size * self.seq_length))

    def create_batches(self):
        self.num_batches = int(self.tensor.size / (self.
            batch_size * self.seq_length))

        # When the data (tensor) is too small, let's give
            them a better error message
        if self.num_batches==0:
            assert False, "Not enough data. Make
                seq_length and batch_size small."

        self.tensor = self.tensor[:self.num_batches *
            self.batch_size * self.seq_length]
        xdata = self.tensor
        ydata = np.copy(self.tensor)
        ydata[:-1] = xdata[1:]
        ydata[-1] = xdata[0]
        self.x_batches = np.split(xdata.reshape(self.
            batch_size, -1), self.num_batches, 1)
        self.y_batches = np.split(ydata.reshape(self.
            batch_size, -1), self.num_batches, 1)

    def next_batch(self):
        x, y = self.x_batches[self.pointer], self.
            y_batches[self.pointer]
        self.pointer += 1
        return x, y

    def reset_batch_pointer(self):
```

```
        self.pointer = 0

# Parameters

batch_size = 60  # minibatch size, i.e. size of data in
    each epoch
num_epochs = 125 # you should increase it if you want to
    see relatively good results
learning_rate = 0.002
decay_rate = 0.97
num_layers = 2 #number of layers in the RNN
## Values set from above
rnn_size = hidden_units # size of RNN hidden state (
    output dimension)
seq_length = sequence_length # RNN sequence length

# Read in
with open('input.txt', 'r') as f:
    read_data = f.read()
        #print read_data[0:200]
f.closed

# load text in batches
data_loader = TextLoader('', batch_size, seq_length)
vocab_size = data_loader.vocab_size

# input and output
x,y = data_loader.next_batch()

# configure RNN
cell = tf.contrib.rnn.BasicRNNCell(rnn_size)

# two layers
stacked_cell = tf.contrib.rnn.MultiRNNCell([cell] *
    num_layers)

# in/out aka target
input_data = tf.placeholder(tf.int32, [batch_size,
    seq_length])# a 60x50
targets = tf.placeholder(tf.int32, [batch_size,
    seq_length]) # a 60x50

# start wth zeroes
initial_state = stacked_cell.zero_state(batch_size, tf.
    float32)

# embedding
with tf.variable_scope('rnnlm', reuse=False):
    softmax_w = tf.get_variable("softmax_w", [rnn_size,
        vocab_size]) #128x65
```

```
    softmax_b = tf.get_variable("softmax_b", [vocab_size
        ]) # 1x65)
    #with tf.device("/cpu:0"):

    # embedding variable is initialized randomely
    embedding = tf.get_variable("embedding", [vocab_size,
         rnn_size])  #65x128

    # embedding_lookup goes to each row of input_data,
        and for each character in the row, finds the
        correspond vector in embedding
    # it creates a 60*50*[1*128] matrix
    # so, the first elemnt of em, is a matrix of 50x128,
        which each row of it is vector representing that
        character
    em = tf.nn.embedding_lookup(embedding, input_data) #
        em is 60x50x[1*128]
    # split: Splits a tensor into sub tensors.
    # syntax:  tf.split(split_dim, num_split, value, name
        ='split')
    # it will split the 60x50x[1x128] matrix into 50
        matrix of 60x[1*128]
    inputs = tf.split(em, seq_length, 1)
    # It will convert the list to 50 matrix of [60x128]
    inputs = [tf.squeeze(input_, [1]) for input_ in
        inputs]

# Squeeze inputs
inputs = tf.split(em, seq_length, 1)
inputs = [tf.squeeze(input_, [1]) for input_ in inputs]

# pass back output and new state
outputs, new_state = tf.contrib.legacy_seq2seq.
    rnn_decoder(inputs, initial_state, stacked_cell,
    loop_function=None, scope='rnnlm')
output = tf.reshape(tf.concat( outputs,1), [-1, rnn_size
    ])
logits = tf.matmul(output, softmax_w) + softmax_b
probs = tf.nn.softmax(logits)
grad_clip =5.
tvars = tf.trainable_variables()

class LSTMModel():
    def __init__(self,sample=False):
        rnn_size = hidden_units # size of RNN hidden
            state vector
        batch_size = 60 # minibatch size, i.e. size of
            dataset in each epoch
        seq_length = sequence_length # RNN sequence
            length
```

```
num_layers = 2 # number of layers in the RNN
vocab_size = 65
grad_clip = 5.
if sample:
    print(">> sample mode:")
    batch_size = 1
    seq_length = 1
# The core of the model consists of an LSTM cell
    that processes one char at a time and computes
     probabilities of the possible continuations
    of the char.
basic_cell = tf.contrib.rnn.BasicRNNCell(rnn_size
    )
# model.cell.state_size is (128, 128)
self.stacked_cell = tf.contrib.rnn.MultiRNNCell([
    basic_cell] * num_layers)

self.input_data = tf.placeholder(tf.int32, [
    batch_size, seq_length], name="input_data")
self.targets = tf.placeholder(tf.int32, [
    batch_size, seq_length], name="targets")
# Initial state of the LSTM memory.
# The memory state of the network is initialized
    with a vector of zeros and gets updated after
    reading each char.
self.initial_state = stacked_cell.zero_state(
    batch_size, tf.float32) #why batch_size

with tf.variable_scope('rnnlm_class1'):
    softmax_w = tf.get_variable("softmax_w", [
        rnn_size, vocab_size]) #128x65
    softmax_b = tf.get_variable("softmax_b", [
        vocab_size]) # 1x65
    with tf.device("/cpu:0"):
        embedding = tf.get_variable("embedding",
            [vocab_size, rnn_size]) #65x128
        inputs = tf.split(tf.nn.embedding_lookup(
            embedding, self.input_data),
            seq_length, 1)
        inputs = [tf.squeeze(input_, [1]) for
            input_ in inputs]
        #inputs = tf.split(em, seq_length, 1)

# The value of state is updated after processing
    each batch of chars.
outputs, last_state = tf.contrib.legacy_seq2seq.
    rnn_decoder(inputs, self.initial_state, self.
    stacked_cell, loop_function=None, scope='
    rnnlm_class1')
```

```
        output = tf.reshape(tf.concat(outputs,1), [-1,
            rnn_size])
        self.logits = tf.matmul(output, softmax_w) +
            softmax_b
        self.probs = tf.nn.softmax(self.logits)
        loss = tf.contrib.legacy_seq2seq.
            sequence_loss_by_example([self.logits],
                [tf.reshape(self.targets, [-1])],
                [tf.ones([batch_size * seq_length])],
                vocab_size)
        self.cost = tf.reduce_sum(loss) / batch_size /
            seq_length
        self.final_state = last_state
        self.lr = tf.Variable(0.0, trainable=False)
        tvars = tf.trainable_variables()
        grads, _ = tf.clip_by_global_norm(tf.gradients(
            self.cost, tvars),grad_clip)
        optimizer = tf.train.AdamOptimizer(self.lr)
        self.train_op = optimizer.apply_gradients(zip(
            grads, tvars))

    def sample(self, sess, chars, vocab, num=200, prime='
        The ', sampling_type=1):
        state = sess.run(self.stacked_cell.zero_state(1,
            tf.float32))
        #print state
        for char in prime[:-1]:
            x = np.zeros((1, 1))
            x[0, 0] = vocab[char]
            feed = {self.input_data: x, self.
                initial_state:state}
            [state] = sess.run([self.final_state], feed)

        def weighted_pick(weights):
            t = np.cumsum(weights)
            s = np.sum(weights)
            return(int(np.searchsorted(t, np.random.rand
                (1)*s)))

        ret = prime
        char = prime[-1]
        for n in range(num):
            x = np.zeros((1, 1))
            x[0, 0] = vocab[char]
            feed = {self.input_data: x, self.
                initial_state:state}
            [probs, state] = sess.run([self.probs, self.
                final_state], feed)
            p = probs[0]
```

```
            if sampling_type == 0:
                sample = np.argmax(p)
            elif sampling_type == 2:
                if char == ' ':
                    sample = weighted_pick(p)
                else:
                    sample = np.argmax(p)
            else: # sampling_type == 1 default:
                sample = weighted_pick(p)

            pred = chars[sample]
            ret += pred
            char = pred
        return ret

with tf.variable_scope("rnn"):
    model = LSTMModel()

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for e in range(num_epochs): # num_epochs is 125
            for test, but should be higher
                sess.run(tf.assign(model.lr,
                    learning_rate * (decay_rate ** e)))
                data_loader.reset_batch_pointer()
                state = sess.run(model.initial_state) #
                    (2x[60x128])
                for b in range(data_loader.num_batches):
                    #for each batch
                        start = time.time()
                        x, y = data_loader.next_batch()
                        feed = {model.input_data: x,
                            model.targets: y, model.
                            initial_state:state}
                        train_loss, state, _ = sess.run([
                            model.cost, model.final_state,
                             model.train_op], feed)
                        end = time.time()
                print("{}/{} (epoch {}), train_loss =
                    {:.3f}, time/batch = {:.3f}" .format(e
                     * data_loader.num_batches + b,
                    num_epochs * data_loader.num_batches,
                    e, train_loss, end - start))
                with tf.variable_scope("rnn", reuse=True)
                    :
                        sample_model = LSTMModel(sample=
                            True)
                        print (sample_model.sample(sess,
                            data_loader.chars ,
                            data_loader.vocab, num=25,
```

```
                                    prime='The ', sampling_type=1)
                                    )
                                print ('--------------------')
#END
```

0.2 Part 1: Hidden Units

Table 1: Hidden Units effect on Perplexity

Hidden Units	Perplexity
32	1.829
64	1.632
128	1.412
256	1.248
512	0.990

0.3 Part 2: Sequence Length

Table 2: Sequence Length effect on Perplexity

Sequence Length	Perplexity
25	1.336
50	1.412
75	1.361