RNN - TENSORFLOW RNN LIBRARY IMPLEMENTATION

```
import numpy as np
import tensorflow as tf
import sys
import os
import random
import cPickle

class parameters():

    def __init__(self):

        self.DATA_FILE = 'data/shakespeare.txt'
        self.CKPT_DIR = 'char_RNN_ckpt_dir'
        self.encoding = 'utf-8'
        self.SAVE_EVERY = 1 # save model every epoch
        self.TRAIN_RATIO = 0.8
        self.VALID_RATIO = 0.1

        self.NUM_EPOCHS = 100
        self.NUM_BATCHES = 50
        self.SEQ_LEN = 50
        self.MODEL = 'rnn'
        self.NUM_HIDDEN_UNITS = 128
        self.NUM_LAYERS = 2
        self.DROPOUT = 0.5

        self.GRAD_CLIP = 5.0
        self.LEARNING_RATE = 0.002
        self.DECAY_RATE = 0.97

        self.SAMPLE_LEN = 500
        self.SEED_TOKENS = “Thou "
        self.SAMPLE_EVERY = 1 # general sample every epoch
        self.SAMPLE_TYPE = 1 # 0=argmax, 1=temperature based
        self.TEMPERATURE = 0.04
```

RNN - TENSORFLOW RNN LIBRARY IMPLEMENTATION

```
def generate_data(config):

    data = open(config.DATA_FILE, "r").read()
    chars = list(set(data))
    char_to_idx = {char:i for i, char in enumerate(chars)}
    idx_to_char = {i:char for i, char in enumerate(chars)}

    config.DATA_SIZE = len(data)
    config.NUM_CLASSES = len(chars)
    config.char_to_idx = char_to_idx
    config.idx_to_char = idx_to_char
    print "\nTotal %i characters with %i unique tokens."  %
(config.DATA_SIZE, config.NUM_CLASSES)

    X = [config.char_to_idx[char] for char in data]
    y = X[1:]
    y[-1] = X[0]

    # Split into train, valid and test sets
    train_last_index = int(config.DATA_SIZE*config.TRAIN_RATIO)
    valid_first_index = train_last_index + 1
    valid_last_index = valid_first_index +
int(config.DATA_SIZE*config.VALID_RATIO)
    test_first_index = valid_last_index + 1

    config.train_X = X[:train_last_index]
    config.train_y = y[:train_last_index]
    config.valid_X = X[valid_first_index:valid_last_index]
    config.valid_y = y[valid_first_index:valid_last_index]
    config.test_X = X[test_first_index:]
    config.test_y = y[test_first_index:]

    return config
```

```
def generate_batch(config, raw_X, raw_y):

    # Create batches from raw data
    batch_size = len(raw_X) // config.NUM_BATCHES # tokens per
batch
    data_X = np.zeros([config.NUM_BATCHES, batch_size],
dtype=np.int32)
    data_y = np.zeros([config.NUM_BATCHES, batch_size],
dtype=np.int32)
    for i in range(config.NUM_BATCHES):
        data_X[i, :] = raw_X[batch_size * i: batch_size * (i+1)]
        data_y[i, :] = raw_y[batch_size * i: batch_size * (i+1)]

    # Even though we have tokens per batch,
    # We only want to feed in <SEQ_LEN> tokens at a time
    feed_size = batch_size // config.SEQ_LEN
    for i in range(feed_size):
        X = data_X[:, i * config.SEQ_LEN:(i+1) * config.SEQ_LEN]
        y = data_y[:, i * config.SEQ_LEN:(i+1) * config.SEQ_LEN]
        yield (X, y)
```

```
def generate_epochs(config, raw_X, raw_y):

    for i in range(config.NUM_EPOCHS):
        yield generate_batch(config, raw_X, raw_y)
```

RNN - TENSORFLOW RNN LIBRARY IMPLEMENTATION

```python
def rnn_cell(config):

    # Get the cell type
    if config.MODEL == 'rnn':
        rnn_cell_type = tf.nn.rnn_cell.BasicRNNCell
    elif config.MODEL == 'gru':
        rnn_cell_type = tf.nn.rnn_cell.GRUCell
    elif config.MODEL == 'lstm':
        rnn_cell_type = tf.nn.rnn_cell.BasicLSTMCell
    else:
        raise Exception("Choose a valid RNN unit type.")

    # Single cell
    single_cell = rnn_cell_type(config.NUM_HIDDEN_UNITS, state_is_tuple=True)

    # Dropout
    single_cell = tf.nn.rnn_cell.DropoutWrapper(single_cell, output_keep_prob=1-config.DROPOUT)

    # Each state as one cell
    stacked_cell = tf.nn.rnn_cell.MultiRNNCell([single_cell] * config.NUM_LAYERS, state_is_tuple=True)
    return stacked_cell


def rnn_inputs(config, input_data):
    with tf.variable_scope('rnn_inputs', reuse=True):
        W_input = tf.get_variable("W_input", [config.NUM_CLASSES, config.NUM_HIDDEN_UNITS])

    # <num_batches, seq_len, num_hidden_units>
    embeddings = tf.nn.embedding_lookup(W_input, input_data)
    # <seq_len, num_batches, num_hidden_units>
    # num_batches will be in columns bc we feed in row by row into RNN.
    # 1st row = 1st tokens from each batch
    #inputs = [tf.squeeze(i, [1]) for i in tf.split(1, config.SEQ_LEN, embeddings)] # NO NEED if using dynamic_rnn
    return embeddings

def rnn_softmax(config, outputs):

    with tf.variable_scope('rnn_softmax', reuse=True):
        W_softmax = tf.get_variable("W_softmax", [config.NUM_HIDDEN_UNITS, config.NUM_CLASSES])
        b_softmax = tf.get_variable("b_softmax", [config.NUM_CLASSES])

    logits = tf.matmul(outputs, W_softmax) + b_softmax
    return logits
```

RNN - TENSORFLOW RNN LIBRARY IMPLEMENTATION

```
def model(config):

    ''' Data placeholders '''
    input_data = tf.placeholder(tf.int32, [config.NUM_BATCHES, config.SEQ_LEN])
    targets = tf.placeholder(tf.int32, [config.NUM_BATCHES, config.SEQ_LEN])

    ''' RNN cell '''
    stacked_cell = rnn_cell(config)

    ''' Inputs to RNN '''
    # Embedding (aka W_input weights)
    with tf.variable_scope('rnn_inputs'):
        W_input = tf.get_variable("W_input", [config.NUM_CLASSES, config.NUM_HIDDEN_UNITS])
    inputs = rnn_inputs(config, input_data)
    initial_state = tf.zeros([config.NUM_BATCHES, config.NUM_LAYERS*config.NUM_HIDDEN_UNITS])

    ''' Outputs from RNN '''
    # Outputs: <seq_len, num_batches, num_hidden_units>
    # state: <num_batches, num_layers*num_hidden_units>
    outputs, state = tf.nn.dynamic_rnn(cell=stacked_cell, inputs=inputs, initial_state=initial_state)
    # <seq_len*num_batches, num_hidden_units>
    outputs = tf.reshape(tf.concat(1, outputs), [-1, config.NUM_HIDDEN_UNITS])

    ''' Process RNN outputs '''
    with tf.variable_scope('rnn_softmax'):
        W_softmax = tf.get_variable("W_softmax", [config.NUM_HIDDEN_UNITS, config.NUM_CLASSES])
        b_softmax = tf.get_variable("b_softmax", [config.NUM_CLASSES])
    # Logits
    logits = rnn_softmax(config, outputs)
    probabilities = tf.nn.softmax(logits)

    ''' Loss '''
    y_as_list = tf.reshape(targets, [-1])
    loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits, y_as_list))
    final_state = state

    ''' Optimization '''
    lr = tf.Variable(0.0, trainable=False)
    trainable_vars = tf.trainable_variables()
    grads, _ = tf.clip_by_global_norm(tf.gradients(loss, trainable_vars),
                                      config.GRAD_CLIP) # glip the gradient to avoid vanishing or blowing up gradients
    optimizer = tf.train.AdamOptimizer(lr)
    train_optimizer = optimizer.apply_gradients(zip(grads, trainable_vars))

    return dict(input_data=input_data, targets=targets, initial_state=initial_state,
                logits=logits, probabilities=probabilities, loss=loss, final_state=final_state,
                lr=lr, train_optimizer=train_optimizer)
```

We initialize the weights here and then to reuse in the functions, we just use reuse=True

Clip the gradient to avoid vanishing gradients.

RNN - TENSORFLOW RNN LIBRARY IMPLEMENTATION

```
def train(config, g):

    # Variables for saving the model along the training procedure
    if not os.path.exists(config.CKPT_DIR):
        os.makedirs(config.CKPT_DIR)
        os.system('touch %s/%s' % (config.CKPT_DIR, 'config.pkl'))
        os.system('touch %s/%s' % (config.CKPT_DIR, 'chars_vocab.pkl'))
    with open(os.path.join(config.CKPT_DIR, 'config.pkl'), 'wb') as f:
        cPickle.dump(config, f)

    with tf.Session() as sess:
        tf.initialize_all_variables().run()

        ''' Load old model state if available '''
        saver = tf.train.Saver(tf.all_variables())
        ckpt = tf.train.get_checkpoint_state(config.CKPT_DIR)
        if ckpt and ckpt.model_checkpoint_path:
            print "Loading old model from:", ckpt.model_checkpoint_path
            saver.restore(sess, ckpt.model_checkpoint_path) # restore all variables

        for epoch_num, epoch in enumerate(generate_epochs(config, config.train_X, config.train_y)):
            train_loss = []

            ''' Saving the model '''
            if (epoch_num % config.SAVE_EVERY == 0):
                print ("\nSaving Model at Epoch %i" % (epoch_num))
                checkpoint_path = os.path.join(config.CKPT_DIR, 'model.ckpt')
                saver.save(sess, checkpoint_path, global_step=epoch_num)

            ''' Generate sample '''
            os.system('python tf_char_rnn_sample.py')

            ''' Training '''
            # Assign/update learning rate
            sess.run(tf.assign(g['lr'], config.LEARNING_RATE * (config.DECAY_RATE ** epoch_num)))
            state = g['initial_state'].eval()

            for minibatch_num, (X, y) in enumerate(epoch):
                loss, state, _, logits = sess.run([g['loss'], g['final_state'], g['train_optimizer'], g['logits']],
                                    feed_dict={g['input_data']:X, g['targets']:y, g['initial_state']:state})
                train_loss.append(loss)

            print "Training loss %.3f" % np.mean(train_loss)
```

load locations to save model is they do not exist

load old model if it exists

save the model at start of each epoch

generate sample

RNN - TENSORFLOW RNN LIBRARY IMPLEMENTATION

```
from tf_char_rnn import *

def generate_sample(config, g):

    with tf.Session() as sess:
        tf.initialize_all_variables().run()

        ''' Generate sample prediction '''
        state = g['initial_state'].eval()

        # Load saved model weights
        saver = tf.train.Saver(tf.all_variables())
        ckpt = tf.train.get_checkpoint_state(config.CKPT_DIR)
        if ckpt and ckpt.model_checkpoint_path:
            print "Loading old model from:", ckpt.model_checkpoint_path
            saver.restore(sess, ckpt.model_checkpoint_path)

        # Process state for given SEED_TOKENS
        for char in config.SEED_TOKENS[:-1]:
            word = np.array(config.char_to_idx[char]).reshape(1,1)
            state = sess.run([g['final_state']],
                    feed_dict={g['input_data']:word, g['initial_state']:state})
            state = np.array(state).reshape((1, np.shape(state)[-1]))

    # Sample text for <sample_len> characters
    sample = config.SEED_TOKENS
    prev_char = sample[-1]
    for word_num in range(0, config.SAMPLE_LEN):
        word = np.array(config.char_to_idx[prev_char]).reshape(1,1)
        probs, state = sess.run([g['probabilities'], g['final_state']],
                        feed_dict={g['input_data']:word, g['initial_state']:state})
        state = np.array(state).reshape((1, np.shape(state)[-1]))

        # probs[0] bc probs is 2D array with just one item
        next_char_dist = probs[0]

        # scale the distribution
        next_char_dist /= config.TEMPERATURE
        next_char_dist = np.exp(next_char_dist)
        next_char_dist /= sum(next_char_dist)
```

argmax

vs.

creative output

```
        if config.SAMPLE_TYPE == 0:
            choice_index = np.argmax(next_char_dist)
        elif config.SAMPLE_TYPE  == 1:
            choice_index = -1
            point = random.random()
            weight = 0.0
            for index in range(0, config.NUM_CLASSES):
                weight += next_char_dist[index]
                    if weight >= point:
                        choice_index = index
                        break
        else:
            raise ValueError("Pick a valid sampling_type!")

        sample += config.idx_to_char[choice_index]
        prev_char = sample[-1]

    print "\nPrediction:"
    print sample
    return sample
```