RNN - TENSORFLOW NAIVE IMPLEMENTATION

```
class parameters():

    def __init__(self):
        self.NUM_EPOCHS = 100
        self.NUM_STEPS = 10 # of tokens per feed for each minibatch row
        self.NUM_BATCHES = 200
        self.STATE_SIZE = 16 # num hidden units per state
        self.LEARNING_RATE = 0.1
        self.FILE = "data/shakespeare.txt"
        self.ENCODING = 'utf-8'

        self.START_TOKEN = 'Thou'
        self.PREDICTION_LENGTH = 50
        self.TEMPERATURE = 0.04 # higher for creativity
```

Define parameters

```
def generate_data(config):
    X = [config.char_to_idx[char] for char in config.data]
    y = X[1:]
    y[-1] = X[0]
    return X, y
```

Create input data

```
def generate_batch(config, raw_data):
    raw_X, raw_y = raw_data
    data_length = len(raw_X)

    # Create batches from raw data
    batch_size = config.DATA_SIZE // 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 <num_steps> tokens at a time
    feed_size = batch_size // config.NUM_STEPS
    for i in range(feed_size):
        X = data_X[:, i * config.NUM_STEPS:(i+1) * config.NUM_STEPS]
        y = data_y[:, i * config.NUM_STEPS:(i+1) * config.NUM_STEPS]
        yield (X, y)
```

Return mini-batches of size <feed_size> simultaneously.

```
def generate_epochs(config):
    for i in range(config.NUM_EPOCHS):
        yield generate_batch(config, generate_data(config))
```

1 epoch

RNN - TENSORFLOW NAIVE IMPLEMENTATION

```python
def rnn_cell(config, rnn_input, state):
    with tf.variable_scope('rnn_cell', reuse=True):
        W_input = tf.get_variable('W_input', [config.NUM_CLASSES, config.STATE_SIZE])
        W_hidden = tf.get_variable('W_hidden', [config.STATE_SIZE, config.STATE_SIZE])
        b_hidden = tf.get_variable('b_hidden', [config.STATE_SIZE], initializer=tf.constant_initializer(0.0))
    return tf.tanh(tf.matmul(rnn_input, W_input) + tf.matmul(state, W_hidden) + b_hidden)

def rnn_logits(config, rnn_output):
    with tf.variable_scope('softmax', reuse=True):
        W_softmax = tf.get_variable('W_softmax', [config.STATE_SIZE, config.NUM_CLASSES])
        b_softmax = tf.get_variable('b_softmax', [config.NUM_CLASSES], initializer=tf.constant_initializer(0.0))
    return tf.matmul(rnn_output, W_softmax) + b_softmax
```

Define functions for state and logit computation so we can use them anywhere as long as the session is running.

The reuse=True indicates to reuse the previous value for the weights under the scope.

RNN - TENSORFLOW NAIVE IMPLEMENTATION

```
def model(config):

    # Placeholders
    X = tf.placeholder(tf.int32, [config.NUM_BATCHES, None], name='input_placeholder')
    y = tf.placeholder(tf.int32, [config.NUM_BATCHES, None], name='labels_placeholder')
    initial_state = tf.zeros([config.NUM_BATCHES, config.STATE_SIZE])

    # Prepre the inputs
    X_one_hot = tf.one_hot(X, config.NUM_CLASSES)
    rnn_inputs = [tf.squeeze(i, squeeze_dims=[1]) for i in tf.split(1, config.NUM_STEPS, X_one_hot)]

    # Define the RNN cell
    with tf.variable_scope('rnn_cell'):
        W_input = tf.get_variable('W_input', [config.NUM_CLASSES, config.STATE_SIZE])
        W_hidden = tf.get_variable('W_hidden', [config.STATE_SIZE, config.STATE_SIZE])
        b_hidden = tf.get_variable('b_hidden', [config.STATE_SIZE], initializer=tf.constant_initializer(0.0))

    # Creating the RNN
    state = initial_state
    rnn_outputs = []
    for rnn_input in rnn_inputs:
        state = rnn_cell(config, rnn_input, state)
        rnn_outputs.append(state)
    final_state = rnn_outputs[-1]

    # Logits and predictions
    with tf.variable_scope('softmax'):
        W_softmax = tf.get_variable('W_softmax', [config.STATE_SIZE, config.NUM_CLASSES])
        b_softmax = tf.get_variable('b_softmax', [config.NUM_CLASSES], initializer=tf.constant_initializer(0.0))

    logits = [rnn_logits(config, rnn_output) for rnn_output in rnn_outputs]
    predictions = [tf.nn.softmax(logit) for logit in logits]

    # Loss and optimization
    y_as_list = [tf.squeeze(i, squeeze_dims=[1]) for i in tf.split(1, config.NUM_STEPS, y)]
    losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(logit, label) for \
              logit, label in zip(logits, y_as_list)]
    total_loss = tf.reduce_mean(losses)
    train_step = tf.train.AdagradOptimizer(config.LEARNING_RATE).minimize(total_loss)

    return dict(X=X, y=y,
                final_state=final_state, logits=logits,
                predictions=predictions, total_loss=total_loss,
                train_step=train_step)
```

<num_batches, num_steps, num_classes>

<num_steps, num_batches, num_classes>

horizontal output is same as RNN output to output layer

RNN - TENSORFLOW NAIVE IMPLEMENTATION

```
def sample(config, sampling_type=1):

    initial_state = tf.zeros([1,config.STATE_SIZE])
    predictions = []

    # Process preset tokens
    state = initial_state
    for char in config.START_TOKEN:
        idx = config.char_to_idx[char]
        idx_one_hot = tf.one_hot(idx, config.NUM_CLASSES)
        rnn_input = tf.reshape(idx_one_hot, [1,65])
        state =  rnn_cell(config, rnn_input, state)

    # Predict after preset tokens
    logit = rnn_logits(config, state)
    prediction = tf.argmax(tf.nn.softmax(logit), 1)[0]
    predictions.append(prediction.eval())

    for token_num in range(config.PREDICTION_LENGTH-1):
        idx_one_hot = tf.one_hot(prediction, config.NUM_CLASSES)
        rnn_input = tf.reshape(idx_one_hot, [1,65])
        state =  rnn_cell(config, rnn_input, state)
        logit = rnn_logits(config, state)

        # scale the distribution
        next_char_dist = logit/config.TEMPERATURE
        next_char_dist = tf.exp(next_char_dist)
        next_char_dist /= tf.reduce_sum(next_char_dist)
        dist = next_char_dist.eval()

        # sample a character
        if sampling_type == 0:
            prediction = tf.argmax(tf.nn.softmax(next_char_dist), 1)[0].eval()
        elif sampling_type == 1:
            prediction = config.NUM_CLASSES - 1
            point = random.random()
            weight = 0.0
            for index in range(0, config.NUM_CLASSES):
                weight += dist[0][index]
                if weight >= point:
                    prediction = index
                    break
        else:
            raise ValueError("Pick a valid sampling_type!")
        predictions.append(prediction)

    return dict(predictions=predictions)
```

(num_classes,)

(1, num_classes)

sampling instead of argmax

RNN - TENSORFLOW NAIVE IMPLEMENTATION

```
def train_network(config, g):
    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())
        training_losses = []

        for idx, epoch in enumerate(generate_epochs(config)):
            training_loss = 0
            training_state = np.zeros((config.NUM_BATCHES, config.STATE_SIZE))

            print "\nEPOCH", idx
            for step, (input_X, input_y) in enumerate(epoch):

                predictions, total_loss, training_state, _ = sess.run(
                            [g['predictions'], g['total_loss'], g['final_state'], g['train_step']],
                            feed_dict={g['X']:input_X,
                                       g['y']:input_y})

                if step%100 == 0 and step>0:
                    print("Average loss:", total_loss/100)
                    training_losses.append(total_loss)

                # Generate predictions
                if idx%10 == 0:
                    print "Prediction:"
                    p = sample(config)
                    print config.START_TOKEN + "".join([config.idx_to_char[prediction] for prediction in p['predictions']])
                    print

    return training_losses

if __name__ == '__main__':
    config = parameters()

    data = open(config.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 = data
    config.DATA_SIZE = len(data)
    config.NUM_CLASSES = len(chars)
    config.char_to_idx = char_to_idx
    config.idx_to_char = idx_to_char

    g = model(config)
    training_losses = train_network(config, g)
```

generate sample prediction