THE NEURAL PERSPECTIVE
DEEP LEARNING SIMPLIFIED

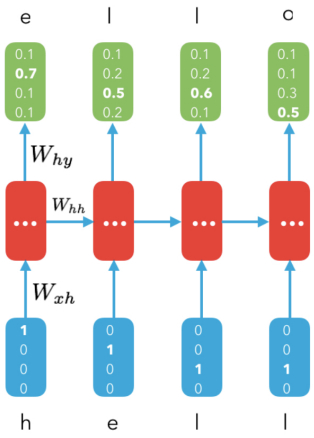
RECURRENT NEURAL NETWORKS (RNN) – PART 1: BASIC RNN / CHAR-RNN

Edit

Note: The section of RNN will span several posts including this one covering basic RNN structure and supporting naive and TF implementation for character level sequence generation. Subsequent posts for RNNs will cover more advanced topics like attention while using more complicated RNN architectures for tasks such as machine translation.

I. OVERVIEW:

There are many advantaged to using a recurrent structure but the obvious ones are being able to keep in memory the representation of the previous inputs. With this, we can better predict the subsequent outputs. There are many problems that arise by keeping track of long streams of memory, such as vanishing gradients during BPTT. Luckily, there are even more architectural changes we can make to combat many of these issues.



Forward Pass:

$$X_t$$

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}X_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

$$P = \frac{e^y}{\sum e_y}$$

$$L = -h(P)$$

X: [NXC] W_{xh}: [CXH]
H: [NXH] W_{hh}: [HXH]
y: [NXC] W_{hy} : [HXC]

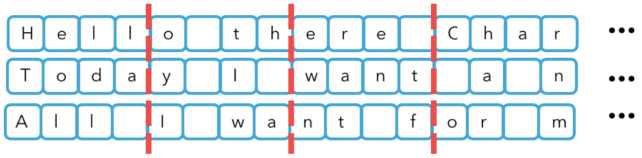
II. CHAR-RNN

We will not implement the naive and TF implementation for a character generating model. The model's objective is to read from each input sequence (stream of chars), one letter at a time and predict the next letter. During training, we will feed in the letter from our input to generate each output letter, but during inference (generation), we will feed in the previous output as the new input (starting with a random token as the first input).

There are a few preprocessing steps done to the text data, so please review the **GitHub Repo** for more detailed info.

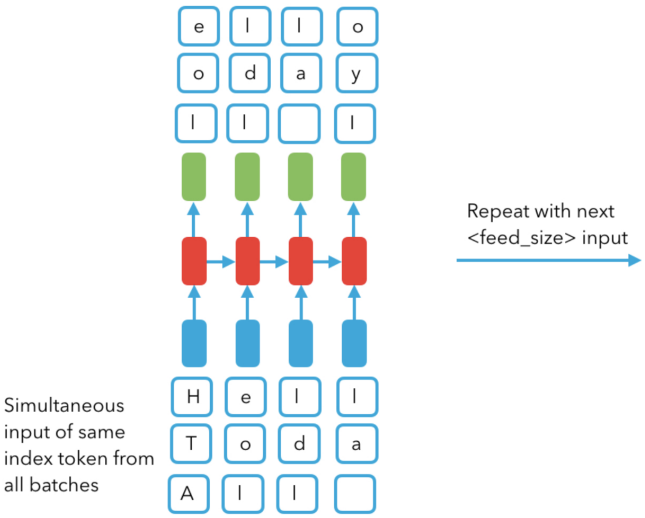
Example input: Hello there Charlie, how are you? Today I want a nice day. All I want for myself is a car.

- DATA_SIZE = len(input)
- BATCH_SIZE = # of sequences per batch
- NUM_STEPS = # of tokens per split (aka seq_len)
- STATE_SIZE = # of hidden units PER hidden state = H
- num_batches = number of batch_sized batches to split in the input into



Note: needs to be columns (above it is rows) because we feed into RNN cell by rows. So you will reshape raw input from to . Also note that each letter will be fed in a as one-hot-encoded vector that will be embedded. Note in the image above, each sentence is perfectly split into a batch. This is just for visualization purpose so you can see how an input would be split. In the actual char-rnn implementation, we don't care about a sentence. We just split the entire input

into num_batches and each batch is split so each input is of length num_steps (aka seq_len).

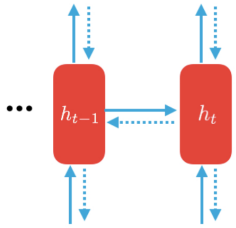


III. BACKPROPAGATION

The BPTT for the RNN structure can be a bit messy at first, especially when computing influence on hidden states and inputs.Use code below from **Karpathy's** naive numpy implementation to follow along the math in my diagram.

$$\frac{\partial L}{\partial W_{hy}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W_{hy}} = \partial \text{scores} * h_t$$

$$\frac{\partial L}{\partial b_y} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b_y} = \partial \text{scores} * 1$$



$$\frac{\partial L}{\partial W_{hh}}, \frac{\partial L}{\partial W_{bh}}, \frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial ?} =$$

$$[\frac{\partial L}{\partial y}\frac{\partial y}{\partial h_t} + \frac{\partial h_{t+1}}{\partial h_t}]\frac{\partial h_t}{\partial tanh}\frac{\partial tanh}{\partial \alpha}\frac{\partial \alpha}{\partial ?} = [\frac{\partial L}{\partial y}\frac{\partial y}{\partial h_t} + \frac{\partial h_{t+1}}{\partial h_t}]\partial h_{raw}\frac{\partial \alpha}{\partial ?}$$

Vertical component from output y

Horizontal component from next state

$$\frac{\partial h}{\partial W_{xh}}, \frac{\partial h}{\partial W_{hh}}, \frac{\partial h}{\partial h_{t+1}}$$

$$\partial h_{raw} = \frac{\partial h_t}{\partial tanh}\frac{\partial tanh}{\partial \alpha} = 1(1 - tanh^2(\alpha)) = (1 - h_t h_t)$$

$$\text{where } \alpha = W_{hh}h_{t-1} + W_{xh}X_t + b_h$$

$$\frac{\partial h}{\partial h_{t+1}} = \partial h_{raw}\frac{\partial \alpha}{\partial h_{t+1}} = \partial h_{raw}W_{hh} = \partial h_{next}$$

$$\partial h_{next} = 0 \ @ \ h_t,$$
$$\partial h_{next} = \partial h_{raw}W_{hh} \ @ \ h_{t-1}$$
$$\bullet\bullet\bullet$$

$$\frac{\partial h}{\partial W_{hh}} = \partial h_{raw}\frac{\partial \alpha}{\partial W_{hh}} = \partial h_{raw}h_{t-1}$$

$$\frac{\partial h}{\partial b_h} = 1 * \partial h_{raw}$$

$$\frac{\partial h}{\partial W_{xh}} = \partial h_{raw}\frac{\partial \alpha}{\partial W_{xh}} = \partial h_{raw}X_t$$

Forward pass:

```
for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k rep
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log prob
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabi
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-e
```

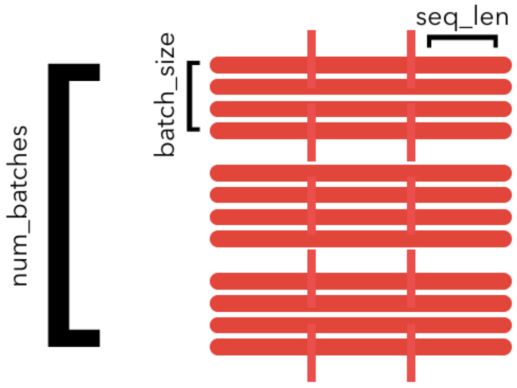
Backpropagation:

```
for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1
    dWhy += np.dot(dy, hs[t].T)
```

```
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through ta
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
```

LEARNING ABOUT SHAPES

Before getting into the implementations, let's talk about shapes. This char-rnn example is a bit odd in terms of shaping, so I'll show you how we make batches here and how they are usually made for seq-seq tasks.



This task is a bit weird in that we feed the entire row of seq_len (all batch_size sequences) at once. Normally, we will just pass in one batch at once, where each batch will have batch_size sequences (batch_size, seq_len). We also don't usually split by seq_len but just take the entire length of a sequence. With seq-seq tasks, as you will see in Part 2 and 3, we feed in a batch with batch_size sequences where each sequence is of length seq_len. We cannot dictate seq_len as we do there because seq_len will just be a max len from all the examples. We just PAD the sequences that do not match that max length. But we'll take a closer look at this in the subsequent posts.

IV. CHAR-RNN TF IMPLEMENTATION (NO RNN ABSTRACTIONS)

This implementation will be using tensorflow but none of the RNN classes for abstraction. We will just be using our own set of weights to really understand where the input data is going and how our output is generated. I will provide code and breakdown analysis with links but will talk about some significant highlights of the code here. If you want an implementation with TF RNN classes, go to section V.

Highlights:

First thing I want to draw attention to is how we generate our batched data. You may notice that we have an additional step where we batch the data and then further split it into seq_lens. This is because of the vanishing gradient problem with BPTT in RNN structures. More information can be found in my blog post **here**. But essentially, we cannot process to many characters at once because during backprop, we will quickly diminish the gradients if the sequence is too long. So a simple trick is to save the output state of a seq_len long sequence and then feed that as the initial_state for the next seq_len. This is also referred to as truncated backpropagation where we choose how much we process (apply BPTT to) and also how often we update. The initial_state starts from zeros and is reset for every epoch. So, we are still able to hold some type of representation in a specific batch from previous seq_len sequences. We need to do this because at the char-level, a small sequence is not enough to really be able to learn adequate representations.

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

    # Create batches from raw data
    num_batches = FLAGS.DATA_SIZE // FLAGS.BATCH_SIZE # to
    data_X = np.zeros([num_batches, FLAGS.BATCH_SIZE], dty
    data_y = np.zeros([num_batches, FLAGS.BATCH_SIZE], dty
    for i in range(num_batches):
        data_X[i, :] = raw_X[FLAGS.BATCH_SIZE * i: FLAGS.B
        data_y[i, :] = raw_y[FLAGS.BATCH_SIZE * i: FLAGS.B

    # Even though we have tokens per batch,
    # We only want to feed in &amp;amp;lt;SEQ_LEN&amp;amp;
    feed_size = FLAGS.BATCH_SIZE // FLAGS.SEQ_LEN
    for i in range(feed_size):
        X = data_X[:, i * FLAGS.SEQ_LEN:(i+1) * FLAGS.SEQ_
        y = data_y[:, i * FLAGS.SEQ_LEN:(i+1) * FLAGS.SEQ_
        yield (X, y)
```

Below is the code that uses all of our weights. We have an rnn_cell that takes in the input and the state from the previous cell in order to generate the rnn output which is also the next cell's input state. The next function, rnn_logits, converts our rnn output using weights to generate logits to be used for probability determination via softmax.

```
def rnn_cell(FLAGS, rnn_input, state):
    with tf.variable_scope('rnn_cell', reuse=True):
        W_input = tf.get_variable('W_input',
            [FLAGS.NUM_CLASSES, FLAGS.NUM_HIDDEN_UNITS])
        W_hidden = tf.get_variable('W_hidden',
            [FLAGS.NUM_HIDDEN_UNITS, FLAGS.NUM_HIDDEN_UNIT
        b_hidden = tf.get_variable('b_hidden', [FLAGS.NUM_
            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(FLAGS, rnn_output):
    with tf.variable_scope('softmax', reuse=True):
        W_softmax = tf.get_variable('W_softmax',
            [FLAGS.NUM_HIDDEN_UNITS, FLAGS.NUM_CLASSES])
        b_softmax = tf.get_variable('b_softmax',
            [FLAGS.NUM_CLASSES], initializer=tf.constant_i
    return tf.matmul(rnn_output, W_softmax) + b_softmax
```

We take our input and one hot encode it and then reshape for batch processing in the RNN. We can then run our RNN to predict the next token using the rnn_cell and rnn_logits functions with softmax. You can see that we generate the state but that also is the same as our rnn output in this simple implementation here.

```
class model(object):

    def __init__(self, FLAGS):

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

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

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

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

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

        logits = [rnn_logits(FLAGS, rnn_output) for rnn_ou
        self.predictions = [tf.nn.softmax(logit) for logit

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

We also sample from our model every once in a while. For sampling, we can either choose to take the argmax (boring) of the logits or introduce some uncertainty in the chosen class using temperature.

```
def sample(self, FLAGS, sampling_type=1):

    initial_state = tf.zeros([1,FLAGS.NUM_HIDDEN_UNITS])
    predictions = []

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

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

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

```
        # scale the distribution
        # for creativity, higher temperatures produce more
        # BUT more creative samples
        next_char_dist = logit/FLAGS.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]
        elif sampling_type == 1:
            prediction = FLAGS.NUM_CLASSES - 1
            point = random.random()
            weight = 0.0
            for index in range(0, FLAGS.NUM_CLASSES):
                weight += dist[0][index]
                if weight &amp;amp;gt;= point:
                    prediction = index
                    break
        else:
            raise ValueError("Pick a valid sampling_type!"
        predictions.append(prediction)

    return predictions
```

Also take a look at how we pass in an initial_state parameter into the data flow. This is updated with the final_state after each sequence is processed. We need to do this in order to avoid vanishing gradients in our RNN. Notice that we feed in a zero initial state for the start and then for subsequent sequences, we take the final_state of the previous sequence as the new input state.

```
state = np.zeros([FLAGS.NUM_BATCHES, FLAGS.NUM_HIDDEN_UNITS

for step, (input_X, input_y) in enumerate(epoch):
    predictions, total_loss, state, _= model.step(sess, inp
                                        input_y, state)
    training_losses.append(total_loss)
```

V. TF RNN LIBRARY IMPLEMENTATION

In this implementation, in contrast with the one above, we will be using tensorflow's nn utility to create the rnn abstraction classes. It's important what we understand what is the input, internal operations and outputs for each of these classes before we use them. We will still be using the basic rnn_cell here so we will be employing truncated backpropagation, but if using GRU or LSTM, there is no need to use it. In fact, just split the entire data into batch_size and just process the entire sequence.

```
def rnn_cell(FLAGS):

    # Get the cell type
    if FLAGS.MODEL == 'rnn':
        rnn_cell_type = tf.nn.rnn_cell.BasicRNNCell
    elif FLAGS.MODEL == 'gru':
        rnn_cell_type = tf.nn.rnn_cell.GRUCell
    elif FLAGS.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(FLAGS.NUM_HIDDEN_UNITS)

    # Dropout
    single_cell = tf.nn.rnn_cell.DropoutWrapper(single_cel
        output_keep_prob=1-FLAGS.DROPOUT)

    # Each state as one cell
    stacked_cell = tf.nn.rnn_cell.MultiRNNCell([single_cel

    return stacked_cell
```

The code above is about creating our specific rnn architecture. We can choose from many different rnn cell types but here you can see three of most common (basic, GRU, and LSTM). We create each cell with a certain number of hidden units. We can then add a dropout layer after cell layer for regularization. Finally, we can make the stacked rnn architecture by replicating the single_cell. Note the **state_is_tuple=True** condition added to single_cell and stacked_cell. This ensures that we get a tuple return that contains the states after each input in a given sequence. The above statement will be true if using an LSTM unit, otherwise, please disregard.

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

    # &amp;amp;lt;BATCH_SIZE, seq_len, num_hidden_units&am
    embeddings = tf.nn.embedding_lookup(W_input, input_dat
    # &amp;amp;lt;seq_len, BATCH_SIZE, num_hidden_units&am
    # BATCH_SIZE will be in columns bc we feed in row by r
    # 1st row = 1st tokens from each batch
    #inputs = [tf.squeeze(i, [1]) for i in tf.split(1, FLA
    # NO NEED if using dynamic_rnn(time_major=False)
    return embeddings

def rnn_softmax(FLAGS, outputs):
    with tf.variable_scope('rnn_softmax', reuse=True):
        W_softmax = tf.get_variable("W_softmax",
            [FLAGS.NUM_HIDDEN_UNITS, FLAGS.NUM_CLASSES])
        b_softmax = tf.get_variable("b_softmax", [FLAGS.NU
```

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

Couple differences between the rnn_inputs function here and in the naive TF implementation. As you can see, we no longer have to reshape our inputs to that it changed from to . This is because we will be receiving the output and state from our rnn by using **tf.nn.dynamic_rnn**. This is a significantly effective and efficient rnn abstraction that requires the inputs not be shaped when fed it, so all we feed in are the embeddings. The rnn_softmax class which gives us the logits remains the same as the previous implementation.

```
class model(object):

    def __init__(self, FLAGS):

        ''' Data placeholders '''
        self.input_data = tf.placeholder(tf.int32, [None,
        self.targets = tf.placeholder(tf.int32, [None, Non

        ''' RNN cell '''
        self.stacked_cell = rnn_cell(FLAGS)
        self.initial_state = self.stacked_cell.zero_state(
            FLAGS.NUM_BATCHES, tf.float32)

        ''' Inputs to RNN '''
        # Embedding (aka W_input weights)
        with tf.variable_scope('rnn_inputs'):
            W_input = tf.get_variable("W_input",
                [FLAGS.NUM_CLASSES, FLAGS.NUM_HIDDEN_UNITS
        inputs = rnn_inputs(FLAGS, self.input_data)

        ''' Outputs from RNN '''
        # Outputs: &amp;amp;lt;seq_len, BATCH_SIZE, num_hi
        # state: &amp;amp;lt;BATCH_SIZE, num_layers*num_hi
        outputs, state = tf.nn.dynamic_rnn(cell=self.stack
                                           initial_state=s

        # &amp;amp;lt;seq_len*BATCH_SIZE, num_hidden_units
        outputs = tf.reshape(tf.concat(1, outputs), [-1, F

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

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

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

        # Components for model saving
        self.global_step = tf.Variable(0, trainable=False)
        self.saver = tf.train.Saver(tf.all_variables())
```

Also notice that we don't manually do one-hot encoding on our input tokens before embeddings them. This is because **tf.nn.embedding_lookup** in rnn_inputs function above does this automatically for us.

For generating the outputs, we use **tf.nn.dynamic_rnn** where the outputs will be the output for each input and the returning state is a tuple containing the last state for each input batch. Finally, we reshape the outputs to shape so we can get the logits and compare to targets.

Notice the **self.initial_state**, with stacked_cell.zero_state, all we have to specify is the batch_size. Here you will see NUM_BATCHES, please refer to section above on shaping for clarification. An another alternative is not including initial_state at all! **dynamic_rnn()** will figure it out on its own, all we need to do is specify the data type (ie. dtype=tf.float32, etc.). But we can't do that here because we pass in the final_state of a sequence as the initial_state of the next sequence. You may notice that we pass in the previous final_state as the new initial_state even though **self.initial_state** is not a placeholder. We can still feed in our own initial just by redefining self.initial_state in **step()**. What ever we need to calculate in our output_feed, the input_feeds will be used and if it is missing, it will just go to predefined (stacked_cell.zero_state in this case) value.

```
def step(self, sess, batch_X, batch_y, initial_state=None)

    if initial_state == None:
        input_feed = {self.input_data: batch_X,
                      self.targets: batch_y}
    else:
        input_feed = {self.input_data: batch_X,
                      self.targets: batch_y,
                      self.initial_state: initial_state}

    output_feed = [self.loss,
                   self.final_state,
                   self.logits,
```

```
                    self.train_optimizer]
    outputs = sess.run(output_feed, input_feed)
    return outputs[0], outputs[1], outputs[2], outputs[3]
```

RESULTS:

Let's take a look at a few results. This by no means going to be earth shattering creativity, but I did use temperature instead of argmax for reproduction. So we will see more creativity but more errors (grammar, spelling, ordering, etc.). I only let it train for 10 epochs but we can already start to see words and sentence structure and even the concept for acting lines for each character (data was shakespeare's work). For decent results, let it train over-night on a GPU.

Look's like Shakespeare has lost his touch and ability to spell.

```
Total 1115393 characters with 65 unique tokens.
Restoring old model parameters from char_RNN_ckpt_dir/model.ckpt-0
Thou I wout :
What will me But the wat thUt SorestW do sare the ha's TourZer't the 'le Martina
 g:
Whouling beeR sing the my a beand, for he our in that the loveZCthours the in th
e in the here'd seake the mjowe lond me meV:
foll &keneThCUS ICand an the sonzen we Whing the ? your the have eresD and , and
 have and in the
ardBenben wiV doond Were the LO !
Whet quent , ad sone the Fir the our harzent.

DUMENCE:
Whan sear the pother
Than You queke xojeed with laTh more your !
Whe zind and hat Sor
The his
```

Update: I got a lot of doubts about the shapes of a typical input, output and state.

- **input** – [num_batches, seq_len, num_classes]
- **output** – [num_batches, seq_len, num_hidden_units] (all outputs from each of the states)
- **state** – [num_batches, num_hidden_units] (this is just the output from the last state)

In this next blog post, we will be dealing with inputs that contain variable sequence lengths and show an implementation for the of text classification.

ALL CODE:

GitHub Repo (Updating all repos, will be back up soon!)

Posted in: NLP
Tagged: Tutorials

← Pointer Sentinel Mixture Models

Context-Dependent Word Representation for Neural Machine Translation →

3 THOUGHTS ON "RECURRENT NEURAL NETWORKS (RNN) – PART 1: BASIC RNN / CHAR-RNN"

Pingback: RECURRENT NEURAL NETWORKS (RNN) – PART 2: Text Classification – The Neural Perspective Edit

LI LI November 15, 2017 at 4:45 am EDIT REPLY →

in train function, is it necessary to let state = None? I mean clear state after each epoch?

model = create_model(sess, FLAGS)
state = None

for epoch_num, epoch in enumerate(generate_epochs(FLAGS,
FLAGS.train_X, FLAGS.train_y)):
train_loss = []
####### clear state?
state = None
# Assign/update learning rate
sess.run(tf.assign(model.lr, FLAGS.LEARNING_RATE *
(FLAGS.DECAY_RATE ** epoch_num)))

# Training
for minibatch_num, (X, y) in enumerate(epoch):
loss, state, logits, _ = model.step(sess, X, y, state)

train_loss.append(loss)

★ Like

GENERAL December 15, 2017 at 10:25 pm EDIT REPLY →

Amazing tutorial thank you so much!

★ Like

LEAVE A REPLY

Enter your comment here...