RECURRENT NEURAL NETWORKS (RNN) – PART 2: TEXT CLASSIFICATION

Edit

In **Part 1** we saw how to implement a simple RNN architecture with TensorFlow. Now, we will use those concepts and apply it to text classification. The main difference here is that our input will not be of fixed length as with the char-rnn model but instead have varying sequence lengths.

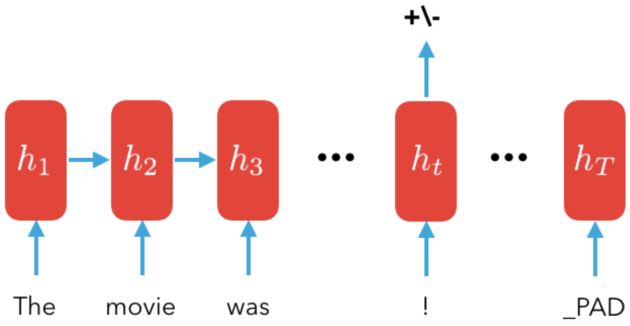
TEXT CLASSIFICATION:

The dataset for this task is the **sentence polarity dataset** v1.0 from Cornell which has 5331 positive and negative sentiment sentences. This is a particularly small dataset but is enough to show how a recurrent network can be used for text classification.

We will need to start by doing some preprocessing, which mainly includes tokenzing the input and appending additional tokens (padding, etc.). See the full code for more info.

PREPROCESSING STEPS:

1. Clean sentences and separate into tokens.
2. Convert sentences into numeric tokens.
3. Store sequence lengths for each sentence



As you can see in the diagram above, we want to predict the sentiment of the sentence as soon as it is complete. Factoring in the additional paddings will introduce too much noise and your network will not perform well. **Note**: The only reason we pad our sequences is because we need to feed in fixed sized batches into the RNN. As you will see below, using a dynamic rnn will allow us to prevent unnecessary calculations once the sequence is complete.

MODEL:

Code:

```
class model(object):

    def __init__(self, FLAGS):

        # Placeholders
        self.inputs_X = tf.placeholder(tf.int32,
            shape=[None, None], name='inputs_X')
        self.targets_y = tf.placeholder(tf.float32,
            shape=[None, None], name='targets_y')
        self.dropout = tf.placeholder(tf.float32)

        # RNN cell
        stacked_cell = rnn_cell(FLAGS, self.dropout)

        # Inputs to RNN
        with tf.variable_scope('rnn_inputs'):
            W_input = tf.get_variable("W_input",
                [FLAGS.en_vocab_size, FLAGS.num_hidden_un

        inputs = rnn_inputs(FLAGS, self.inputs_X)
        #initial_state = stacked_cell.zero_state(FLAGS.ba

        # Outputs from RNN
        seq_lens = length(self.inputs_X)
        all_outputs, state = tf.nn.dynamic_rnn(cell=stack
            sequence_length=seq_lens, dtype=tf.float32)
```

- SEARCH -

Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…

Embed View on Twitter

- RECENT POSTS -

Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

Opening the Black Box of Deep Neural Networks via Information

```
        # state has the last RELEVANT output automaticall
        # [0] because state is a tuple with a tensor insi
        outputs = state[0]

        # Process RNN outputs
        with tf.variable_scope('rnn_softmax'):
            W_softmax = tf.get_variable("W_softmax",
                [FLAGS.num_hidden_units, FLAGS.num_classe
            b_softmax = tf.get_variable("b_softmax", [FLA

        # Logits
        logits = rnn_softmax(FLAGS, outputs)
        probabilities = tf.nn.softmax(logits)
        self.accuracy = tf.equal(tf.argmax(
            self.targets_y,1), tf.argmax(logits,1))

        # Loss
        self.loss = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(logit

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

        # Below are values we will use for sampling (gene
        # after each word.)

        # this is taking all the ouputs for the first inp
        # (only 1 input sequence since we are sampling)
        sampling_outputs = all_outputs[0]

        # Logits
        sampling_logits = rnn_softmax(FLAGS, sampling_out
        self.sampling_probabilities = tf.nn.softmax(sampl

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

    def step(self, sess, batch_X, batch_y=None, dropout=0
        forward_only=True, sampling=False):

        input_feed = {self.inputs_X: batch_X,
                      self.targets_y: batch_y,
                      self.dropout: dropout}

        if forward_only:
            if not sampling:
                output_feed = [self.loss,
                               self.accuracy]
            elif sampling:
                input_feed = {self.inputs_X: batch_X,
                              self.dropout: dropout}
                output_feed = [self.sampling_probabilitie
        else: # training
            output_feed = [self.train_optimizer,
                           self.loss,
                           self.accuracy]

        outputs = sess.run(output_feed, input_feed)

        if forward_only:
            if not sampling:
                return outputs[0], outputs[1]
            elif sampling:
                return outputs[0]
        else: # training
            return outputs[0], outputs[1], outputs[2]
```

The code above is the model that trains using the input text. **Note**: We decided to keep batch size in our input and target placeholders for clarity but we should make these independent of a particular batch size. If we do this, we will have to feed in initial_state since it depends on batch_size. We feed in the tokens from each input sequence by embedding them. A proven strategy for better performance is to pretrain these embedding weights with a skip-gram model on the input text.

In this model, we use dynamic_rnn again but this time we feed in the value for sequence_length which is a list that has the length of each sequence. By doing this, we avoid doing unnecessary computation past the last word of the input sequence. The function **length** is what gives us this list and is shown below. We could've also just as easily calculated seq_len outside and then pass it in via a placeholder.

```
def length(data):
    relevant = tf.sign(tf.abs(data))
    length = tf.reduce_sum(relevant, reduction_indices=1)
    length = tf.cast(length, tf.int32)
    return length
```

Since our PAD token is a 0, we can use the sign property of each token to determine wether it is a padding token or not. tf.sign will give us a 1 if input > 0 and 0 if input==0. By using this, we can find the number of tokens with the positive sign by reduce by the column index. We can now feed this length into dynamic_rnn.

Note: We could have easily just calculated seq_lens outside and pass it in as a placeholder. This way we wouldn't have to depend on PAD_ID=0.

Once we receive all the outputs and the final state from our rnn, we want to isolate only the relevant output. For each input we will have a different last relevant output since each input length can vary. We can find the relevant output simply by looking at **state** because state is the last RELEVANT output since we fed in seq_len into dynamic_rnn. Note that we have to do state[0] because the returning state is a tuple of tensor(s).

Couple more things to note: I did not use **initial_state** instead I just passed in dtype into dynamic_rnn. Also, dropout is a parameter I pass in with **step()** depending on **forward_only()** or not.

INFERENCE:

I also wanted to generate predictions for some sample sentences but not just for each sentence as a whole. I instead wanted to see how the prediction score changes as each word is read by the rnn while keeping the previous words in memory. Here is one example of what this looks like (closer to 0 indicated negative sentiment):

```
Loading old model from: checkpoints/model.ckpt-99
it 0.386203
was 0.35975
the 0.349884
worst 0.271178
movie 0.212363
i 0.17374
have 0.124769
ever 0.0951016
seen 0.074659
```

NOTE: This is a very simple model with a very limited dataset. The main objective was just to show how it should be made and how it will run. For superior performance, try using a much larger and richer dataset and consider additional architectural implementations such as attention, concept aware word embeddings, and symbolization to name a few.

EXTRA: LOSS MASKING (NOT NEEDED HERE)

Finally, we compute the cost and you may notice we aren't doing any "loss masking" here because we isolated our relevant output and used only that to compute the loss. However, for other tasks such as machine translation, we may have the outputs that come from the PAD tokens as well. We do not want to account for these outputs because the dynamic_rnn with seq_lens parameter fulfilled will return 0 for these outputs. Here is a simple example of what this implementation will look like; again, we use the fact that our PAD token is a 0.

```
# Flatten the logits and targets
targets = tf.reshape(targets, [-1]) # flatten targets
losses = tf.nn.sparse_softmax_cross_entropy_with_logits(log
mask = tf.sign.(tf.to_float(targets)) # 0 is targets == 0,
masked_losses = mask*losses # contributions from padding lo
```

The first step will be to flatten the logits and the targets. For flattening the logits, a good idea is to flatten the outputs from the dynamic_rnn into [-1, num_hidden_units] shape and then multiply by softmax weights [num_hidden_units, num_classes]. By masking the loss, we are canceling out the loss contributions from the padded locations.

ALL CODE:

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

CHANGE OF SHAPE REFERENCE:

Raw unprocessed text X is **[N,]** and y is **[N, C]** where C is the number of output classes. (These were made manually but we would use a one-hot function for a multi-class situation).

Then X is converted to tokens and padded and is now **[N, <max_len>]**. We also are feeding in seq_len which is of shape **[N,]** and holds the lengths of each sentence.

Now X, seq_lens and y go through the model and first is the embedding [NXD] where D is the embedding dimension. X now transforms from **[N, <max_len>]** to **[N, <max_len>, D]**. Recall that there is an intermediate representation where X is one-hot encoded [N, <max-len>, <num_words>] but we don't actually have to make this because we just use the word's index and take that row from embedding weights.

We need this embedded X into the dynamic_rnn which returns all_outputs (**[N, <max_len>, D]**) and state(**[1, N, D]**), for us it is the last relevant state since we fed in seq_lens. From the dimensions, you can see that all_outputs is all of the outputs from the RNN for each word in each sentence. Whereas state is only the last relevant output for each sentence.

Now we are going to feed into softmax weights but before that we convert state from **[1, N, D]** to **[N, D]** by just taking the first index (state[0]). We can now dot product with softmax weights **[D, C]** to get outputs of **[N, C]** which we do the exponential softmax operation and normalization to and compute the loss with the target_y **[N, C]**.

Note: If you used an basic RNN or GRU, the shapes of the all_outputs and state returns from

Note: If you used an basic RNN or GRU, the shapes of the all_outputs and state returns from dynamic_rnn would be same. But if we used an LSTM the all_outputs still has shape **[N, <max_len>, D]** but state is of shape **[1, 2, N, D]**.

Posted in: *NLP*, *Representation Learning*
Tagged: *Tutorials*

← *Context-Dependent Word Representation for Neural Machine Translation*

HyperNetworks →

6 THOUGHTS ON "RECURRENT NEURAL NETWORKS (RNN) – PART 2: TEXT CLASSIFICATION"

Pingback: Convolutional Text Classification – The Neural Perspective Edit

MIHAILSALNIKOV (@MIHAILSALNIKOV) March 10, 2017 at 11:25 am EDIT REPLY →

Good post. Thank you!

How soon are you going publish code?

★ Liked by you

> **GOKUMOHANDAS** March 10, 2017 at 3:04 pm EDIT REPLY →
>
> 
>
> Hey Mihail,
> I will be finding some time soon, but for this particular subject, I will have a new O'Reilly publication out soon!
>
> ★ Like

DAVID NGUYEN March 15, 2017 at 3:37 pm EDIT REPLY →

hi Goku Mohandas
You can publish source code for this post in blog. I research for Text classfication use RNN model. Thank you so much

★ Like

LI LI November 15, 2017 at 7:31 am EDIT REPLY →

As your last note: "If you used an basic RNN or GRU, the shapes of the all_outputs and state returns from dynamic_rnn would be same. But if we used an LSTM the all_outputs still has shape [N, , D] but state is of shape [1, 2, N, D]."
If I run with –rnn_unit=lstm, it will crash because shape not match:
ValueError: Shape must be rank 2 but is rank 3 for 'MatMul' (op: 'MatMul') with input shapes: [2,?,300], [300,2].

★ Like

MARYAM November 28, 2017 at 5:48 pm EDIT REPLY →

Dear GOKUMOHANDAS,
I appreciate you for spending your time to make this tutorial.
would you mind please teaching (explaining) each sentence of codes (one by one).
what operations do the functions? each sentence refers to which function?
as Iam a beginner.

★ Like

LEAVE A REPLY