CONVOLUTIONAL TEXT CLASSIFICATION

Edit

In this post, we will be using a CNN for a text classification task. It will be very similar to the **previous** text classification task we did using RNNs but this time we will use a **CNN** in order to process the sequences.

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.

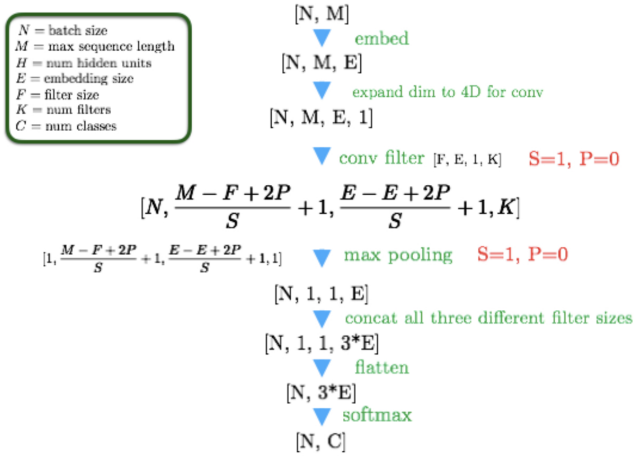
PREPROCESSING STEPS:

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

Once we have our processed inputs ready, we will embed them, send it through a convolution and max-pooling layer using varied filter sizes to extract one feature vector that represents each sequence sentence. We finally apply softmax on this vector to determine the class and compare with actual class for loss and training. The interesting aspect is how we create our feature vector using the CNN.

MODEL

Below is the dimensional analysis of the input sequences going through the convolutional layers and eventually giving us a softmax probability for each class.



First step involves embedding the input and then converting to 4D in order to apply our conv filters on the embedded input.

```
# Inputs to RNN
with tf.variable_scope('embed_inputs'):
    W_input = tf.get_variable("W_input",
        [FLAGS.en_vocab_size, FLAGS.num_hidden_units])
self.embedded_inputs = embed_inputs(FLAGS, self.inputs_X)

# Made embedded inputs in to 4D input for CNN
self.conv_inputs = tf.expand_dims(self.embedded_inputs, -1)
```

Then we are ready to apply the conv filters on the inputs and then do the max-pooling

operations. We will be storing the outputs from each of our three different filter sizes. Then we will concat by the last dimension so we have one vector for each of our inputs.

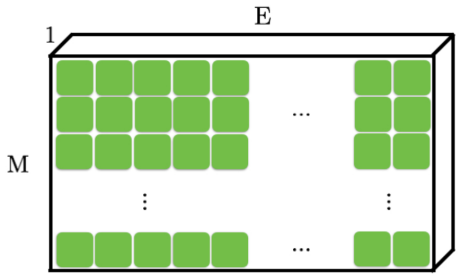
```
self.pooled_outputs = []
for i, filter_size in enumerate(FLAGS.filter_sizes):
    with tf.name_scope("CNN-%s" % filter_size):
        # Convolution Layer
        filter_shape = [
            filter_size, FLAGS.num_hidden_units, 1, FLAGS.num_filters]
        W = tf.Variable(
            tf.truncated_normal(filter_shape, stddev=0.1), name="W")
        b = tf.Variable(
            tf.constant(0.1, shape=[FLAGS.num_filters]), name="b")
        self.conv = tf.nn.conv2d(
            input=self.conv_inputs,
            filter=W,
            strides=[1, 1, 1, 1], # S = 1
            padding="VALID", # P = 0
            name="conv")

        # Add bias and then apply the nonlinearity
        h = tf.nn.relu(tf.nn.bias_add(self.conv, b))

        # apply max pool
        self.pooled = tf.nn.max_pool(
            value=h,
            ksize=[1, FLAGS.max_sequence_length - filter_size + 1, 1, 1]
            strides=[1, 1, 1, 1], # usually [1, 2, 2, 1] for reduction
            padding="VALID", # P = 0
            name="pool")
        self.pooled_outputs.append(self.pooled)
```

Finally, we will flatten our feature vector so we can apply softmax and calculate loss and conduct optimization to adjust our weights. Before softmax, however, we will null some of the features using a dropout operation, as means of regularization for robustness.

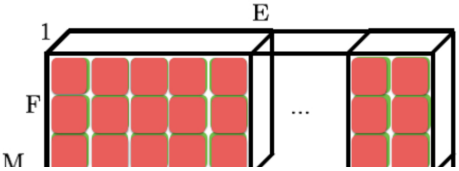
```
# Combine all the pooled features and flatten
num_filters_total = FLAGS.num_filters * len(FLAGS.filter_sizes)
self.h_pool = tf.concat(3, self.pooled_outputs)
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])

# Apply dropout
with tf.name_scope("dropout"):
    self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob
```

CONVOLUTION

We see how the math works out above with the convolution and max-pooling, but we can develop a much better understanding by actually visualizing it. Think of the input into the CNN ([N, M, E, 1]) as N images of shape MXE with 1 color channel.



And think of the filter as also 1D but with a heigh of filter_size (we use 3, 4 and 5 in our implementation) and a width of E. The filter will convolve across the entire input (one sequence at a time) and produce outputs along the way. The resulting shape is given in the dimensional analysis above.



We apply K of these filters for each filter size. After receing the convolutional outputs, we apply a nonlinearity (ReLU here) to all of the outputs and then proceed to apply max-pooling. Our pooling will be exact heigh and width of the convolutional outputs for each sequence because we wish to reduce the outputs from pooling to the dimension [N, 1, 1, E]. Since our stride is 1, our pooling needs to match the dimensions of the convolutional outputs.

Using convolution to extract features, wether it be from images or any sequential input, has it's advantages. For one, we were able to reduce computation compared to an RNN. Our input was [N, M, E] which we converted to [N, E] after the CNN. We can use this for reducing the length of our inputs in many different situations. For example in the fully-char level translation **paper** (which I discussed **here**). Since using characters to represent an input sentence will yield longer inputs, we can using convolution to extract features and then use pooling (with strides even greater than 1, like 5 in the paper) to really shorten our input lengths for further processing.

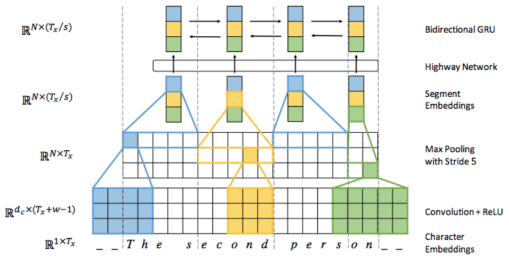


Figure 1: Encoder architecture schematics. Underscore denotes padding. A dotted vertical line delimits each segment.

Additionally, CNNs hold the advantage of just being great feature extractors and the ability to perform apply operations in parallel across time steps (can do operation on the entire input sequence, from 0 to nth token, but does this one sample at a time). Where as an RNN, cannot do this but can instead apply operations in parallel across channels (can do operation on all n tokens of all n input sequence in a batch). Clearly, both CNNs and RNNs hold distinct advantages, so an active area of research is focused on creating architectures that combine them. You can find out more about these quasi architectures in papers such as this **one**.

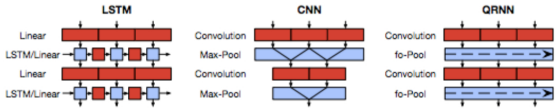


Figure 1: Block diagrams showing the computation structure of the QRNN compared with typical LSTM and CNN architectures. Red signifies convolutions or matrix multiplications; a continuous block means that those computations can proceed in parallel. Blue signifies parameterless functions that operate in parallel along the channel/feature dimension. LSTMs can be factored into (red) linear blocks and (blue) elementwise blocks, but computation at each timestep still depends on the results from the previous timestep.

arXiv

CODE

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

Posted in: Sequence-to-Sequence
Tagged: Tutorials

← *Reinforcement Learning (RL) – Policy Gradients II*

Ask Me Anything: Dynamic Memory Networks for Natural Language Processing →

LEAVE A REPLY