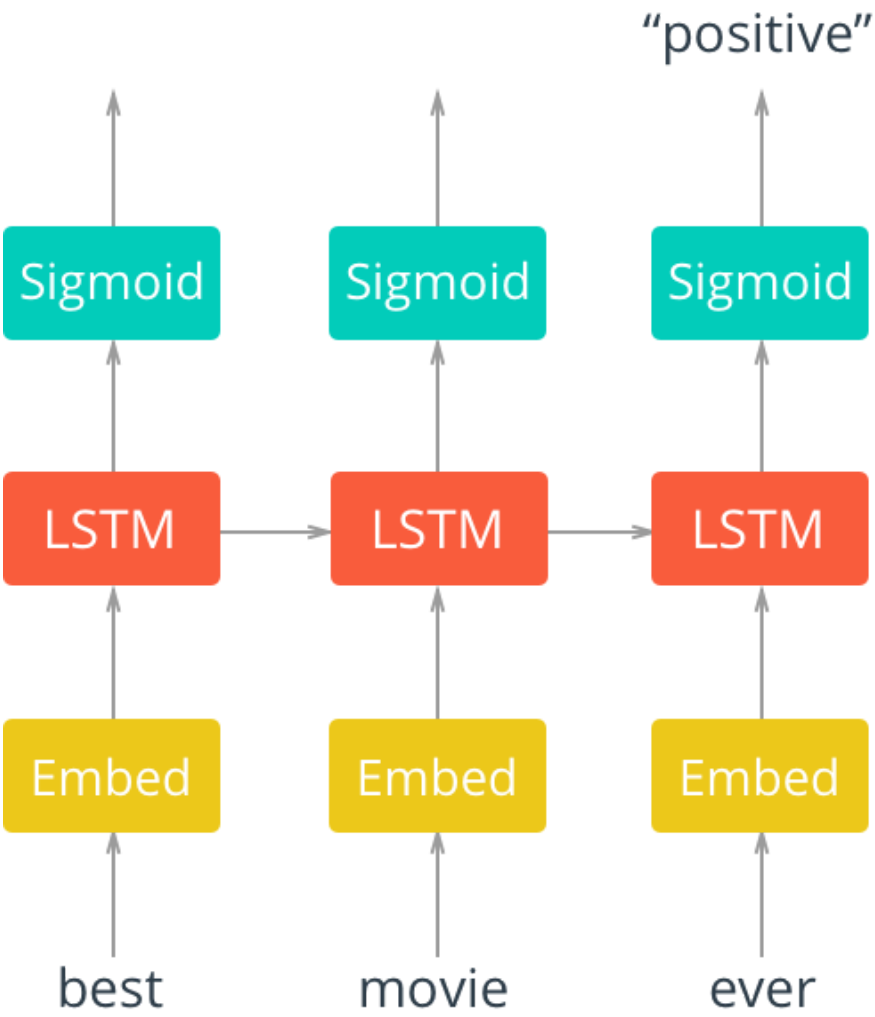
Manish Chablani Follow
Deep learning for self driving cars @ Cruise automation (Past: Machine Learning @ Uber, Early engineer on Microsoft, Azure cloud storage), Marathoner.
Jun 20, 2017 · 4 min read

Sentiment analysis using RNNs(LSTM)

Here we use the example of reviews to predict sentiment (even though it can be applied more generically to other domains for example sentiment analysis for tweets, comments, customer feedback, etc). Whole idea here is that movie reviews are made of sequence of words and order of words encode lot of information that is useful to predict sentiment. Step 1 is to map words to word embeddings (see post 1 and 2 for more context on word embeddings). Step 2 is the RNN that receives a sequence of vectors as input and considers the order of the vectors to generate prediction.

The architecture for this network is shown below.



> *Here, we'll pass in words to an embedding layer. You can actually train up an embedding with word2vec and use it here. But it's good enough to just have an embedding layer and let the network learn the embedding table on it's own.*

From the embedding layer, the new representations will be passed to LSTM cells. These will add recurrent connections to the network so we can include information about the sequence of words in the data. Finally, the LSTM cells will go to a sigmoid output layer here. We're using the sigmoid because we're trying to predict if this text has positive or negative sentiment. The output layer will just be a single unit then, with a sigmoid activation function.

We don't care about the sigmoid outputs except for the very last one, we can ignore the rest. We'll calculate the cost from the output of the last step and the training label.

Architecture Summary:

Have fixed length reviews encoded as integers and then converted to embedding vectors passed to LSTM layers in recurrent manner and pick the last prediction as output sentiment.

Gotchas:

One thing in my experiments I could not explain is when I encode the words to integers if I randomly assign unique integers to words the best accuracy I get is 50–55% (basically the model is not doing much better than random guessing). However if the words are encoded such that highest frequency words get the lowest number then the model accuracy is 80% in 3–5 epochs. My guess is this is necessary to train the embedding layer but cannot find an explanation on why anywhere.

Code:

https://github.com/mchablani/deep-learning/blob/master/sentiment-rnn/Sentiment_RNN.ipynb

Data preprocessing:

Take all the words in reviews and encode them with integers. Now each review is an ordered array of integers. Make each review fixed size (say 200), so shorter reviews get padded with 0's in front and longer reviews get truncated to 200. Since we are padding with 0's the corpus of words to int mapping starts with 1. Labels are are encoded as 1s and 0s for 'positive' and 'negative'.

Build the graph

```
lstm_size = 256
lstm_layers = 2
batch_size = 500
learning_rate = 0.001
embed_size = 300


n_words = len(vocab_to_int) + 1 # Add 1 for 0 added to vocab


# Create the graph object
tf.reset_default_graph()
with tf.name_scope('inputs'):
```

```
    inputs_ = tf.placeholder(tf.int32, [None, None],
name="inputs")
    labels_ = tf.placeholder(tf.int32, [None, None],
name="labels")
    keep_prob = tf.placeholder(tf.float32, name="keep_prob")


# Sizeof embedding vectors (number of units in the embedding
layer)
with tf.name_scope("Embeddings"):
    embedding = tf.Variable(tf.random_uniform((n_words,
embed_size), -1, 1))
    embed = tf.nn.embedding_lookup(embedding, inputs_)


def lstm_cell():
    # Your basic LSTM cell
    lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size,
reuse=tf.get_variable_scope().reuse)
    # Add dropout to the cell
    return tf.contrib.rnn.DropoutWrapper(lstm,
output_keep_prob=keep_prob)


with tf.name_scope("RNN_layers"):
    # Stack up multiple LSTM layers, for deep learning
    cell = tf.contrib.rnn.MultiRNNCell([lstm_cell() for _ in
range(lstm_layers)])

    # Getting an initial state of all zeros
    initial_state = cell.zero_state(batch_size, tf.float32)


with tf.name_scope("RNN_forward"):
    outputs, final_state = tf.nn.dynamic_rnn(cell, embed,
initial_state=initial_state)


with tf.name_scope('predictions'):
    predictions =
tf.contrib.layers.fully_connected(outputs[:, -1], 1,
activation_fn=tf.sigmoid)
    tf.summary.histogram('predictions', predictions)
with tf.name_scope('cost'):
    cost = tf.losses.mean_squared_error(labels_,
predictions)
    tf.summary.scalar('cost', cost)


with tf.name_scope('train'):
    optimizer =
tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

Batching and Training

```
def get_batches(x, y, batch_size=100):
```

```
    n_batches = len(x)//batch_size
    x, y = x[:n_batches*batch_size],
y[:n_batches*batch_size]
    for ii in range(0, len(x), batch_size):
        yield x[ii:ii+batch_size], y[ii:ii+batch_size]


epochs = 10


# with graph.as_default():
saver = tf.train.Saver()


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    train_writer = tf.summary.FileWriter('./logs/tb/train',
sess.graph)
    test_writer = tf.summary.FileWriter('./logs/tb/test',
sess.graph)
    iteration = 1
    for e in range(epochs):
        state = sess.run(initial_state)

        for ii, (x, y) in enumerate(get_batches(train_x,
train_y, batch_size), 1):
            feed = {inputs_: x,
                    labels_: y[:, None],
                    keep_prob: 0.5,
                    initial_state: state}
            summary, loss, state, _ = sess.run([merged,
cost, final_state, optimizer], feed_dict=feed)
#             loss, state, _ = sess.run([cost, final_state,
optimizer], feed_dict=feed)


train_writer.add_summary(summary, iteration)

            if iteration%5==0:
                print("Epoch: {}/{}".format(e, epochs),
                      "Iteration: {}".format(iteration),
                      "Train loss: {:.3f}".format(loss))


if iteration%25==0:
                val_acc = []
                val_state =
sess.run(cell.zero_state(batch_size, tf.float32))
                for x, y in get_batches(val_x, val_y,
batch_size):
                    feed = {inputs_: x,
                            labels_: y[:, None],
                            keep_prob: 1,
                            initial_state: val_state}
#                     batch_acc, val_state =
sess.run([accuracy, final_state], feed_dict=feed)
                    summary, batch_acc, val_state =
sess.run([merged, accuracy, final_state], feed_dict=feed)
                    val_acc.append(batch_acc)
                print("Val acc:
{:.3f}".format(np.mean(val_acc)))
            iteration +=1
```

```
                test_writer.add_summary(summary, iteration)
                saver.save(sess,
  "checkpoints/sentiment_manish.ckpt")
      saver.save(sess, "checkpoints/sentiment_manish.ckpt")
```

Credits: From lecture notes:
https://classroom.udacity.com/nanodegrees/nd101/syllabus