NLP Assignment 5 (Final Deep learning)

Vinuta Hegde|vinutahe@usc.edu|95.5% (Japanese), 96% (Italian)

I would like to highlight 3 main points which I learned working on this assignment:

- Picking the right network/model:
  I had started with Simple RNN cell and one directional neural network. I tried my best to tune the hyper parameter but could only reach up 93% average accuracy. **Bidirectional_dynamic_rnn** with **LSTM** cells performed way better, since It would consider the sequence of word-tags sequence from both the direction. The model was able to successfully tag more than 94% of words (in both Japanese and Italian)
- Tuning the Hyperparameters:
  It definitely demanded way more patience then I expected.
  I followed the traditional method of binary search for fixing all the hyper-parameters.
  a. Started with the smallest value possible (not literally). For example, learning rate can take any value between [0,1], so I started with 0.00001.
  b. Next run, tried with biggest possible value (again, not literally).
  c. Tried middle value where m = small+((big-small)/2)
  d. Then switched min or max, based on the accuracy.
  e. Repeated above 4 steps until I got the satisfactory accuracy).

  I tuned below 4 parameters:

  Learning_rate (0.008)
  Batch_size (30)
  Embedding_size(100)
  Batch_size(100)

  After hours and hours of tuning, my learning rate was still missing the optimal value. Either it was too small to reach the minimum loss, or it was too big, it would skip the minima.
- The secret sauce:
  After days of trying my best to tune the learning rate, I decided to take things into my hand. I tried to set pretty large learning rate (about 0.05) and exit training early to reach better accuracy. To certain extent I was even successful. Then when staring at the DL 1 assignment, training part, where we were calling the epoch with smaller and smaller learning rate, I decided to try the something similar here too.
  I expressed the learning rate as a function of epoch_number.

  $$learning\_rate = \frac{\mathrm{C}}{\mathrm{N^\wedge(epoch\_num-1)}}$$

  Where, C = initial learning rate
  N = Any constant

For detailed implementation, please refer the code below:

```
class SequenceModel(object):
    """
        Constuctor
            Inputs:
                max_length : int
                num_terms : int
                num_tags : int
            return:
    """
    def __init__(self, max_length=310, num_terms=1000, num_tags=40):
        # Length of longest sentence
        self.max_length = max_length
        # Number of unique terms or words
        self.num_terms = num_terms
        # Number of unique terms or words
```

```
        self.num_tags = num_tags
        # keeping a count, how many times epoch ran
        self.epoc_num = 0
        # is a hyper parameter, userd to create embedding {'embeddings',
[self.num_terms,self.embedding_size])}
        self.embedding_size = 100
        # number of states of LSTM cell
        self.state_size = 100
        # placeholder for terms or words
        self.x = tf.placeholder(tf.int64, [None, self.max_length], 'X')
        # placeholder for lengths of each sentence
        self.lengths = tf.placeholder(tf.int32, [None], 'lengths')
        #  placeholder for Tags or ground truths
        self.targets = tf.placeholder(tf.int64, [None, self.max_length], 'targets')
        # placeholder for learning rate of the model
        self.learning_rate = tf.placeholder_with_default(numpy.array(0.01, dtype='float32'), shape=[],
name='learn_rate')

    """
       This method takes length vector for n entries and converts it into binary mask of size n *
max_length .
       [3,5,4]  to [[True, True, True, False, False],
                    [True, True, True, True, True],
                    [True, True, True, True, False]]
            Inputs:
                length_vector : integer array
            return:
                n * max_length binary vector
    """
    def lengths_vector_to_binary_matrix(self, length_vector):
        return tf.sequence_mask(length_vector, self.max_length)

    """
       Method creates/fetches embedding variable, creates the  embedding slice for current batch of
terms.
       passes them to bidirectional_dynamic_rnn to update the embedding values for each all the terms.
       Uses dense layer to reshape, and create logits.

               Inputs:
               return:

    """
    def build_inference(self):
        # create or fetch embeddings
        embeddings = tf.get_variable('embeddings', [self.num_terms,self.embedding_size])
        # look up for terms from current batch
        xemb = tf.nn.embedding_lookup(embeddings, self.x)
        # create  forward and backward LSTM cells
        fw_cell = tf.keras.layers.LSTMCell(self.state_size)
        bw_cell = tf.keras.layers.LSTMCell(self.state_size)
        # create bidirectional RNN
        (fw_cell1,bw_cell1),_ = tf.nn.bidirectional_dynamic_rnn(fw_cell,bw_cell,xemb, sequence_length =
self.lengths,dtype = tf.float32)
        # concatinate forward and the backward final states of bidirectional rnn
        op = tf.concat([fw_cell1,bw_cell1],axis=-1)
        # pass through dense layer, scale down the dimensions to match number of tags.
        # and assign it to logits
        self.logits = tf.layers.dense(op, units=self.num_tags, activation=None)

    """
        Return tags for each terms in the batch.
               Inputs:
                     terms: batch_size * self.max_length
                     lengths: batch size
               return:
                     tags : batch_size * self.max_length
    """
    def run_inference(self, terms, lengths):
        logits = self.sess.run(self.logits, {self.x: terms, self.lengths: lengths})
        return numpy.argmax(logits, axis=2)

    """
        Define loss and optimizer.
        Initialize the training session
```

```
            Inputs:
            return:

    """
    def build_training(self):
        # get sequence mask and typecast
        binary_matrix = tf.cast(tf.sequence_mask(self.lengths, self.max_length), dtype=numpy.float32)
        # add sequence to sequence losses
        tf.losses.add_loss(tf.contrib.seq2seq.sequence_loss(self.logits, self.targets, binary_matrix))
        #define optimizer
        self.optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
        # define training model using seq2seq loss and AdamOptimizer
        self.train_op = tf.contrib.training.create_train_op(tf.losses.get_total_loss(), self.optimizer)
        # create session and initialise global variables
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())
        pass

    """
        Takes parameters, applies learning_rate decay, and starts rus the model for training.
            Inputs:
                terms: batch_size * max_length
                tags : batch_size * max_length
                lengths: batch_size
                batch_size: int
                learn_rate : double
            return:
                True (to run next epoch)

    """
    def train_epoch(self, terms, tags, lengths, batch_size=30, learn_rate=0.008):
        # Each epoch the learning rate decreses by 2^(epoc_num -1)
        # learning_rate =initial_ learning_rate / 2^(epoc_num -1)
        decay_factor = 1
        for i in range(1, self.epoc_num):
            decay_factor = decay_factor * 2

        # Increment epoch number
        self.epoc_num = self.epoc_num+1

        # Randomize the indices
        indices = numpy.random.permutation(terms.shape[0])

        # create batches from terms, tags and length matrices
        for si in range(0, terms.shape[0], batch_size):
            se = min(si + batch_size, terms.shape[0])
            slice_x = terms[indices[si:se]]
            slice_tag = tags[indices[si:se]]
            slice_len = lengths[indices[si:se]]
            # Train the model using the current slice.
            self.sess.run(self.train_op, {self.targets: slice_tag, self.x: slice_x,self.learning_rate:
(learn_rate/decay_factor), self.lengths: slice_len})
        return True
        pass
```

Tried and failed:

- Multiple hidden layers (Did not decrease/ increase the accuracy)
- Cell Dropouts (Did not make any significant improvement in accuracy)
- Regularization (No effect on accuracy)
- GradientDescentOptimizer (Very bad idea)
- Fixing random seed to avoid variation in accuracy (ran out of patience while finding the right value)
- Initialization of embeddings using random_uniform_initializer (Same as above)