```
import tensorflow as tf
import numpy as np

class classifier:
    def __init__(self,
            learning_rate,
            hidden_size,
            batch_size,
            max_time,
            embeddings,
            global_step):

      self.batch_size = batch_size
      self.learning_rate = tf.train.exponential_decay(
        learning_rate, global_step, 500, 0.8, staircase=True)
      self.hidden_size = hidden_size
      self.batch_size = batch_size
      self.max_time = max_time
      self.embeddings = tf.constant(embeddings, name = "emb")
      self.optimizer = tf.train.AdamOptimizer(self.learning_rate)
      self.global_step = global_step

      tf.summary.scalar('Learning Rate', self.learning_rate)

      self.make()

      self.saver = tf.train.Saver(max_to_keep = 200)

    def make(self):
      self.sequence_length = tf.placeholder(tf.float32, shape = (self.
batch_size))
      self.inputs = tf.placeholder(tf.int32, shape = (self.batch_size,
 self.max_time))
      self.embed = tf.nn.embedding_lookup(self.embeddings, self.inputs
)
      self.targets = tf.placeholder(tf.int64, shape = (self.batch_size
,2))
      self.keep_prob = tf.placeholder(tf.float32, shape = None)

      cell = tf.nn.rnn_cell.LSTMCell(
        num_units = self.hidden_size,
        use_peepholes=True)

      dropout_cell = tf.contrib.rnn.DropoutWrapper(
        cell = cell,
        output_keep_prob = self.keep_prob
        )

      self.output, state = tf.nn.dynamic_rnn(
        cell = dropout_cell,
        inputs = self.embed,
        sequence_length = self.sequence_length,
        initial_state = cell.zero_state(self.batch_size, tf.float32))

      self.sum = tf.reduce_sum(self.output, axis = 1)
      self.mean = tf.divide(self.sum, tf.expand_dims(self.sequence_len
gth,1))
```

```
      self.logits = tf.contrib.layers.fully_connected(self.mean,2,acti
vation_fn = None)
      #self.logits = self.mean
      self.probability = tf.nn.softmax(self.logits)
      self.decision = tf.argmax(self.probability,axis=1)
      self.actual = tf.argmax(self.targets,axis=1)
      self.probe = self.decision
      self.metric_accuracy,self.update_accuracy = \
          tf.metrics.accuracy(self.actual,self.decision)

      self.xent = tf.nn.softmax_cross_entropy_with_logits(
        labels = self.targets,
        logits = self.logits)

      self.loss = tf.reduce_mean(self.xent)
      # warning: changed logits to probability - may be numerically un
stable
      self.pos_grad = tf.gradients(self.probability[0][0], self.embed)
      self.neg_grad = tf.gradients(self.probability[0][1], self.embed)
      self.logit_grad = tf.gradients(self.logits[0][0], self.embed)
      self.updates = self.optimizer.minimize(self.loss, global_step =
self.global_step)

      tf.summary.scalar('Metric Accuracy', self.update_accuracy)
      tf.summary.scalar('Loss', self.loss)

      self.merged = tf.summary.merge_all()

    def infer_dpg(self,sess,rv):
        decision, probability, grad = \
            sess.run([self.decision,self.probability,self.pos_grad],
                feed_dict = \
                    {self.inputs:rv.index_vector,
                     self.targets:rv.targets,
                     self.sequence_length:[rv.length],
                     self.keep_prob:1.0})

        return decision, probability, grad

    def infer_rep_dpg(self,sess,rv,word_index):
        index_matrix = np.tile(rv.index_vector,(rv.length,1) )
        np.fill_diagonal(index_matrix,word_index)
        target_matrix = np.tile(rv.targets,(rv.length,1))

        decision, probability = \
            sess.run([self.decision,self.probability],
                feed_dict = \
                    {self.inputs:index_matrix,
                     self.targets:target_matrix,
                     self.sequence_length:[rv.length]*rv.length,
                     self.keep_prob:1.0})

        return decision, probability

    def infer_batched_prob(self,sess,rv,word_index,per_batch,top_idx):
        num_top = self.batch_size//per_batch
```

```
        index_matrix = np.tile(rv.index_vector,(self.batch_size,1))
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
        n = 0
        for idx in top_idx:
            for i in range(per_batch):
                index_matrix[n,idx] = word_index + i
                n = n + 1

        decision, probability, grad = \
            sess.run([self.decision,self.probability,self.pos_grad],
                feed_dict = \
                    {self.inputs:index_matrix,
                     self.targets:target_matrix,
                     self.sequence_length:[rv.length]*self.batch_size,
                     self.keep_prob:1.0})

        return decision, probability, grad

    # inserts all words at index 'insert_location'
    def infer_insert(self,sess,rv,insert_location,divs,top_k):
        per = rv.length // divs
        end = per*divs
        new = per+1
        index_matrix = np.zeros((self.batch_size,new),'int')
        wim = np.reshape(rv.index_vector[0,0:end],(per,divs),'F')
        wim = np.insert(wim,insert_location,0,axis=0)
        wim = np.reshape(wim,(1,end+divs),'F')
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
        for i in range(divs):
            start = i*top_k; end = (i+1)*top_k
            index_matrix[start:end,:] = np.tile(wim[0,new*i:new*(i+1)]
,(top_k,1))
        #index_matrix = np.tile(wim,(self.batch_size//divs,1))
        index_matrix[:,insert_location] = \
            np.arange(self.batch_size) % (self.batch_size//divs)
        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [new] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g,index_matrix

    def infer_swap(self,sess,rv,swap_location,divs,top_k):
        per = rv.length // divs
        end = per*divs
        index_matrix = np.zeros((self.batch_size,per),'int')
        wim = np.copy(np.reshape(rv.index_vector[0,0:end],(per,divs),'
F'))
        wim[swap_location,:] = 0
        wim = np.reshape(wim,(1,end),'F')
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
        replacement_vector = np.random.choice(np.arange(10000),size=se
lf.batch_size)
```

```
        for i in range(divs):
            start = i*top_k; end = (i+1)*top_k
            index_matrix[start:end,:] = np.tile(wim[0,per*i:per*(i+1)]
,(top_k,1))
        index_matrix[:,swap_location] = replacement_vector
            #np.arange(self.batch_size) % (self.batch_size//divs)
        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [per] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g,index_matrix,replacement_vector

    def infer_window(self,sess,rv,word_index,window_size):
        w = window_size
        L = rv.length
        n = word_index
        n1 = n - w//2; n2 = n1 + w
        if n1 >= 0 and n2 <= L:
            i1 = n1; i2 = n2
        elif n1 < 0 and n2 <= L:
            i1 = 0; i2 = w
        elif n1 >=0 and n2 > L:
            i1 = L-w; i2 = L
        else:
            i1 = 0; i2 = rv.length
        index_matrix = np.tile(rv.index_vector[0,i1:i2],(10000,1))
        index_matrix[:,word_index-i1] = np.arange(10000)
        target_matrix = np.tile(rv.targets,(self.batch_size,1))

        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [w] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g

    def infer_multi(self,sess,rv,ii,K):
        # K is the list of source word indices to be replaced
        # batch size is fixed at 10,000 with number of
        # destination words = 10,000 / len(K)
        N = self.batch_size // K.size
        index_matrix = np.tile(rv.index_vector[0,:],(self.batch_size,1
))
        c = 0
        for k in list(K):
            index_matrix[c*N:(c+1)*N,k] = ii[np.arange(N),k]
            c = c + 1
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
```

```
        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [rv.length] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g
```