IMDB_sent_an_TF_Estimator

August 14, 2018

1 Sentiment analysis on IMDB reviews: TensorFlow GloVe and LSTM with Estimator APIs

In this notebook I will try to perform sentiment analysis using TensorFlow. Most of the notebook is a variation of what was done on this blog: https://www.oreilly.com/learning/perform-sentiment-analysis-with-lstms-using-tensorflow

This is an upgrade of the previous notebook (IMDB_sent_an_TF_basic_improved1) where I'm replacing the basic APIs by custom Estimator level APIs. For this I follow the indications of these tutorials coming from the official documentation:

https://www.tensorflow.org/get_started/premade_estimators
https://www.tensorflow.org/get_started/datasets_quickstart
https://www.tensorflow.org/get_started/custom_estimators

1.1 Libraries

```
In [1]: import numpy as np
        import csv
        import io
        import tensorflow as tf
        import pickle
```

1.2 Preprocessing and Data exploration

The Data exploration part (measuring the average number of words in the reviews) and the data preprocessing, turning texts into sequence of indexes corresponding the GloVes word embeddings, are done in another notebook called IMDB_sent_an_data_preprocessing. The variable created there are then loaded in the next sections.

1.3 Loading matrices of embedding indexes and lists of labels

The pretrained embeddins from GloVe can be downloaded here: https://nlp.stanford.edu/projects/glove/

There are different **word embedding sizes**. The possibilities are 50, 100, 200, 300. We define the one we use next.

```
In [2]: word_emb_size = '100'
```

```
In [3]: prepr_dir = '/home/aritz/Documents/CS_Programming_Machine_Learning/Projects/IMDB_sentime
```

```
In [4]: ids_train = np.load(prepr_dir+'Saved_embeddings/idsMatrixTrain'+word_emb_size+'.npy')
        ids_test = np.load(prepr_dir+'Saved_embeddings/idsMatrixTest'+word_emb_size+'.npy')

In [5]: max_seq_len = ids_train.shape[1]
```

Next we load the **labels** with and without **one-hot-encoding** ([1, 0] for positive and [0, 1] for negative).

```
In [6]: with open(prepr_dir+"y_train_ord.txt", "rb") as fp:
            y_train_ord = pickle.load(fp)

        with open(prepr_dir+"y_test_ord.txt", "rb") as fp:
            y_test_ord = pickle.load(fp)

        with open(prepr_dir+"y_train.txt", "rb") as fp:
            y_train = pickle.load(fp)

        with open(prepr_dir+"y_test.txt", "rb") as fp:
            y_test = pickle.load(fp)
```

Next we load the **list of words in the GloVe table** and a numpy array containing the **GloVe look-up table**:

```
In [7]: with open(prepr_dir+"words_list.txt", "rb") as fp:
            words_list = pickle.load(fp)

        word_vectors = np.load(prepr_dir+'word_vectors.npy')
```

1.4 Definition of the model

1.4.1 Estimator APIs

In this section we create a model using the Estimator APIs from TF. One of the advantage of this higher level of APIs is that some things done manualy when using TF basic APIs, are done automatically. There is no need to initialize variables for instance, or defining writers for TensorBoard.

First we define the **input functions**. They are the objects which supply data for training, evaluating, and prediction to the model. They are using `tf.data.Dataset` objects which are one of the key tools of TF. These objects allow to access the data and manipulate it.

In the body of the next function, it is important that the argument of the `shuffle` method is equal to the length of the whole training data set. See entry of the 11.07.18 of my journal for details.

```
In [8]: len_trn = ids_train.shape[0]

In [9]: # features is a numpy array of shape (#samples, 250)
        def train_input_fn(features, labels, batch_size):
            """An input function for training"""
            # Convert the inputs to a Dataset.
            dataset = tf.data.Dataset.from_tensor_slices(({'Indexes':features}, labels))
```

```
            # Shuffle, repeat, and batch the examples.
            dataset = dataset.shuffle(len_trn).repeat().batch(batch_size)

            return dataset
```

```
In [10]: # features is a numpy array of shape (#samples, 250)
         def eval_input_fn(features, labels, batch_size):
             """An input function for evaluation or prediction"""
             if labels is None:
                 # No labels, use only features.
                 inputs = {'Indexes':features}
             else:
                 inputs = ({'Indexes':features}, labels)

             # Convert the inputs to a Dataset.
             dataset = tf.data.Dataset.from_tensor_slices(inputs)

             # Batch the examples
             assert batch_size is not None, "batch_size must not be None"
             dataset = dataset.batch(batch_size)

             return dataset
```

Next we define the **feature columns**:

```
In [11]: my_feature_columns = []
         my_feature_columns.append(tf.feature_column.numeric_column(key='Indexes', shape=max_seq
```

Next we define the **directory where to store the log files for TensorBoard** as well as the checkpoint file, the model files, and the graph file. These last files enable the notebook to **automatically store and restaure** previously trained models (as long as the architecture is the same in the old and the new notebook), as explained in this tutorial:
https://www.tensorflow.org/guide/checkpoints

```
In [12]: model_dir = '/home/aritz/Documents/CS_Programming_Machine_Learning/Projects/IMDB_sentim
```

In order to define a custom estimator we need to define a **model function**. For this we mix the code of the notebook based only on basic TF APIs together with some parts of the script of this tutorial: https://www.tensorflow.org/get_started/custom_estimators

```
In [13]: def my_model(features, labels, mode, params):
             # Use `input_layer` to apply the feature columns.
             input_data = tf.feature_column.input_layer(features, params['feature_columns'])
             # The next line is required because tf.feature_column.input_layer
             # outputs tf.float32 (whatever the input)
             # and tf.nn.embedding_lookup requires
             # tf.int32
             input_data = tf.cast(input_data, tf.int32)
             # Transform each index in a sentence into the associated vector
```

```
data = tf.nn.embedding_lookup(word_vectors, input_data)
# The following line is a fixe coming from this page:
# https://github.com/tgjeon/TensorFlow-Tutorials-for-Time-Series/issues/2
# in order to prevent an error appearing next.
data = tf.cast(data, tf.float32)
# Next we define the LSTM
lstm_cell = tf.contrib.rnn.BasicLSTMCell(params['lstm_units'])
lstm_cell = tf.contrib.rnn.DropoutWrapper(cell=lstm_cell, output_keep_prob=params['
value, _ = tf.nn.dynamic_rnn(lstm_cell, data, dtype=tf.float32)
# swaps the two first dimensions so it has dimensions [max_time, batch_size, cell.o
value = tf.transpose(value, [1, 0, 2])
# If I'm not mistaken the next cell slices the part of
# the output which corresponds to the last output of the lstm,
# or in other words the output corresponding to the
# last word for every sample (if I'm right we used
# 0 padding and cut everything which goes beyound 250 words,
# so technically it is the 250th output).
# My guess is that last has dimensions [batch_size, cell.output_size]
# which we can then use to do matrix multiplication
# with weight which has dimensions [cell.output_size, numClasses]
# (remember that cell.output_size=lstm_units).
last = tf.gather(value, int(value.get_shape()[0]) - 1)
# We apply an affine transformation to get the logits
#weight = tf.Variable(tf.truncated_normal([params['lstm_units'], params['n_classes'
#bias = tf.Variable(tf.constant(0.1, shape=[params['n_classes']]))
#logits = (tf.matmul(last, weight) + bias)
logits = tf.layers.dense(inputs=last, units=2)
# Maybe I could  replace this last part using tf.layers.dense:
# https://www.tensorflow.org/api_docs/python/tf/layers/dense

# The following lines are actually independent of the achitecture
# of the model.

# Compute predictions.
predicted_classes = tf.argmax(logits, 1)
if mode == tf.estimator.ModeKeys.PREDICT:
    predictions = {
        'class_ids': predicted_classes[:, tf.newaxis],
        'probabilities': tf.nn.softmax(logits),
        'logits': logits,
    }
    return tf.estimator.EstimatorSpec(mode, predictions=predictions)

# Compute loss

# Note that because of this function, we have to
# provide ordinaly encoded labels and not one-hot-encoded
# labels, as explained on this page:
```

```
        # https://stackoverflow.com/questions/48114258/tensorflow-estimator-number-of-class
        loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

        # Compute evaluation metrics.
        accuracy = tf.metrics.accuracy(labels=labels,
                                       predictions=predicted_classes,
                                       name='acc_op')
        metrics = {'accuracy': accuracy}
        tf.summary.scalar('loss', loss)
        tf.summary.scalar('accuracy', accuracy[1])

        if mode == tf.estimator.ModeKeys.EVAL:
            return tf.estimator.EstimatorSpec(
                mode, loss=loss, eval_metric_ops=metrics)

        # Create training op.
        assert mode == tf.estimator.ModeKeys.TRAIN

        optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
        train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
        return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

Next we can define the **custom estimator**.

```
In [14]: classifier = tf.estimator.Estimator(
                model_fn=my_model,
                model_dir=model_dir,
                params={
                    'feature_columns': my_feature_columns,
                    'n_classes': 2,
                    'lstm_units': 64,
                    'keep_prob': 0.7
                })
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using config: {'_task_type': 'worker', '_keep_checkpoint_every_n_hours': 10000,
```

1.5 Training and evaluation of the Estimator

```
In [15]: batch_size = int(100)
         train_steps = int(3)
```

Now we can **train** our Estimator. Note that for the function `tf.losses.sparse_softmax_cross_entropy`, that we are using in the model function, requires the **label** to be **ordinaly encoded and not one-hot-encoded** as explained here:

https://stackoverflow.com/questions/48114258/tensorflow-estimator-number-of-classes-does-not-change

```
In [16]: classifier.train(
             input_fn=lambda:train_input_fn(features=ids_train, labels=y_train_ord, batch_size=b
             steps=train_steps)

INFO:tensorflow:Calling model_fn.
WARNING:tensorflow:From /home/aritz/.local/lib/python3.5/site-packages/tensorflow/contrib/learn/
Instructions for updating:
Use the retry module or similar alternatives.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.


/home/aritz/.local/lib/python3.5/site-packages/tensorflow/python/ops/gradients_impl.py:100: User
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "


INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /home/aritz/Documents/CS_Programming_Machine_Learning/
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 14004 into /home/aritz/Documents/CS_Programming_Machine_L
INFO:tensorflow:step = 14003, loss = 0.18444294
INFO:tensorflow:Saving checkpoints for 14006 into /home/aritz/Documents/CS_Programming_Machine_L
INFO:tensorflow:Loss for final step: 0.17752373.


Out[16]: <tensorflow.python.estimator.estimator.Estimator at 0x7f69a34fa5f8>
```

Now we can evaluate our model on the test data.

```
In [17]: eval_test = classifier.evaluate(input_fn=lambda:eval_input_fn(features=ids_test,
                                                                       labels=y_test_ord,
                                                                       batch_size=batch_si
                                           )

INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-07-21-21:44:48
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /home/aritz/Documents/CS_Programming_Machine_Learning/
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-07-21-21:45:10
INFO:tensorflow:Saving dict for global step 14006: accuracy = 0.84184635, global_step = 14006, l
```

Finally we print the **accuracy of the model on the test set**.

```
In [18]: print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_test))
```

```
Test set accuracy: 0.842
```

```
In [19]: print(eval_test)
```

```
{'accuracy': 0.84184635, 'global_step': 14006, 'loss': 0.48276785}
```

1.6 Long Training

In this section I will implement a loop which will train the model as long as the quality of the prediction keeps improving.

```
In [49]: batch_size = int(100)
         train_steps = int(1000)
```

```
In [54]: acc_i = 0
         acc_iplus1 = 0
         acc_iplus2 = eval_test['accuracy']
         num_iter = 0
```

```
In [55]: while ((acc_iplus2 > acc_iplus1) or (acc_iplus2 > acc_i)):
             num_iter = num_iter + 1
             classifier.train(
             input_fn=lambda:train_input_fn(features=ids_train, labels=y_train_ord, batch_size=b
             steps=train_steps)
             acc_i = acc_iplus1
             acc_iplus1 = acc_iplus2
             eval_test = classifier.evaluate(input_fn=lambda:eval_input_fn(features=ids_test,
                                                                           labels=y_test_ord,
                                                                           batch_size=batch_si
                                             )
             acc_iplus2 = eval_test['accuracy']
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
```

```
/home/aritz/.local/lib/python3.5/site-packages/tensorflow/python/ops/gradients_impl.py:100: User
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
```

```
INFO:tensorflow:Restoring parameters from /home/aritz/Documents/CS_Programming_Machine_Learning/
INFO:tensorflow:Running local_init_op.
```

```
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 12001 into /home/aritz/Documents/CS_Programming_Machine_L
INFO:tensorflow:loss = 0.116583526, step = 12000
INFO:tensorflow:global_step/sec: 5.52669
INFO:tensorflow:loss = 0.24673532, step = 12100 (18.095 sec)
INFO:tensorflow:global_step/sec: 5.42818
INFO:tensorflow:loss = 0.1551571, step = 12200 (18.423 sec)
INFO:tensorflow:global_step/sec: 5.73969
INFO:tensorflow:loss = 0.15166456, step = 12300 (17.423 sec)
INFO:tensorflow:global_step/sec: 6.5761
INFO:tensorflow:loss = 0.34310123, step = 12400 (15.206 sec)
INFO:tensorflow:global_step/sec: 6.50861
INFO:tensorflow:loss = 0.11865516, step = 12500 (15.364 sec)
INFO:tensorflow:global_step/sec: 6.46593
INFO:tensorflow:loss = 0.3100912, step = 12600 (15.466 sec)
INFO:tensorflow:global_step/sec: 5.84718
INFO:tensorflow:loss = 0.13938998, step = 12700 (17.102 sec)
INFO:tensorflow:global_step/sec: 6.45558
INFO:tensorflow:loss = 0.15073161, step = 12800 (15.490 sec)
INFO:tensorflow:global_step/sec: 6.60011
INFO:tensorflow:loss = 0.24637079, step = 12900 (15.151 sec)
INFO:tensorflow:Saving checkpoints for 13000 into /home/aritz/Documents/CS_Programming_Machine_L
INFO:tensorflow:Loss for final step: 0.17228603.
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-07-15-11:38:52
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /home/aritz/Documents/CS_Programming_Machine_Learning/
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-07-15-11:39:15
INFO:tensorflow:Saving dict for global step 13000: accuracy = 0.84956604, global_step = 13000, l
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /home/aritz/Documents/CS_Programming_Machine_Learning/
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 13001 into /home/aritz/Documents/CS_Programming_Machine_L
INFO:tensorflow:loss = 0.1351165, step = 13000
INFO:tensorflow:global_step/sec: 5.78828
INFO:tensorflow:loss = 0.11435699, step = 13100 (17.277 sec)
INFO:tensorflow:global_step/sec: 6.47174
INFO:tensorflow:loss = 0.13042718, step = 13200 (15.452 sec)
INFO:tensorflow:global_step/sec: 6.45931
INFO:tensorflow:loss = 0.10244565, step = 13300 (15.482 sec)
INFO:tensorflow:global_step/sec: 5.97935
```

```
INFO:tensorflow:loss = 0.04903977, step = 13400 (16.724 sec)
INFO:tensorflow:global_step/sec: 6.16554
INFO:tensorflow:loss = 0.23376682, step = 13500 (16.219 sec)
INFO:tensorflow:global_step/sec: 6.52103
INFO:tensorflow:loss = 0.1585369, step = 13600 (15.336 sec)
INFO:tensorflow:global_step/sec: 6.12519
INFO:tensorflow:loss = 0.21807069, step = 13700 (16.325 sec)
INFO:tensorflow:global_step/sec: 6.14174
INFO:tensorflow:loss = 0.16492325, step = 13800 (16.282 sec)
INFO:tensorflow:global_step/sec: 6.24359
INFO:tensorflow:loss = 0.118033655, step = 13900 (16.016 sec)
INFO:tensorflow:Saving checkpoints for 14000 into /home/aritz/Documents/CS_Programming_Machine_L
INFO:tensorflow:Loss for final step: 0.1796388.
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-07-15-11:43:08
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /home/aritz/Documents/CS_Programming_Machine_Learning/
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-07-15-11:43:31
INFO:tensorflow:Saving dict for global step 14000: accuracy = 0.8447662, global_step = 14000, lo
```

```
In [59]: print(acc_i)
         print(acc_iplus1)
         print(acc_iplus2)
```

```
0.83672655
0.8407664
0.83468664
```

1.7 TensorBoard

In order to track the progress of the model on TensorBoard it is enough to enter "tensorboard --logdir (...)" in a terminal with "(...)" replaced by the name of the directory where the event files are saved, and visiting http://localhost:6006/ with a browser.

With custom estimators it is enough to include lines like `tf.summary.scalar('loss', loss)` in the model function in order to track the quantities we are interested in. An event file for TensorBoard will be updated every 100 steps during the training with the `train` method (at least it is what I observe from my empirical experience), and a single event file measuring the state of the tracked variable will be written when calling the `evaluate`method. There is no need to define a writer with `tf.summary.FileWriter`.

Caveat emptor: This being said, I often ran into problems with TensorBoard. No files were being written, or only during the evaluation phase. Currently it seems to work as I described above, but I cannot garantee that what I described is absolutely true. I don't know yet how to specify the behaviour of an Estimator object when it comes to TensorBoard.

1.8 Debugging

In the next cells I use TF basic APIs to access directly what is happening when I call the train method of my Estimator object. This allows me to understand source of errors and warnings.

```
In [ ]: batch_size = int(100)
        lstm_units = int(64)
```

We start by defining **two datasets**. One for the training and one for the testing:

```
In [ ]: dataset_train = train_input_fn(features=ids_train, labels=y_train_ord, batch_size=batch_
        dataset_test = eval_input_fn(features=ids_test, labels=y_test_ord, batch_size=batch_size
```

Next we define the **reinitializable iterator**. Unlike One-shot iterators, they alow to switch from one dataset to another one. As explained here: https://www.tensorflow.org/programmers_guide/datasets#creating_an_iterator, "A reinitializable iterator is defined by its structure. We could use the `output_types` and `output_shapes` properties of either `dataset_train` or `dataset_test` here, because they are compatible."

```
In [ ]: iterator = tf.data.Iterator.from_structure(dataset_train.output_types,
                                                   dataset_train.output_shapes)
```

```
In [ ]: features, labels = iterator.get_next()
```

The next cell contains the model itself (this part is similar to what is found in the model function of the Estimator object).

```
In [ ]: input_data = tf.feature_column.input_layer(features, my_feature_columns)
        input_data = tf.cast(input_data, tf.int32)
        data = tf.nn.embedding_lookup(word_vectors, input_data)
        data = tf.cast(data, tf.float32)
        lstm_cell = tf.contrib.rnn.BasicLSTMCell(lstm_units)
        lstm_cell = tf.contrib.rnn.DropoutWrapper(cell=lstm_cell, output_keep_prob=0.8)
        value, _ = tf.nn.dynamic_rnn(lstm_cell, data, dtype=tf.float32)
        value = tf.transpose(value, [1, 0, 2])
```

```
In [ ]: print(value)
```

```
In [ ]: last = tf.gather(value, int(value.get_shape()[0]) - 1)
        #weight = tf.Variable(tf.truncated_normal([lstm_units, 2]))
        #bias = tf.Variable(tf.constant(0.1, shape=(2,)))
        #logits = (tf.matmul(last, weight) + bias)
        logits = tf.layers.dense(inputs=last, units=2)
        predicted_classes = tf.argmax(logits, 1)
        loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
        accuracy = tf.metrics.accuracy(labels=labels,
                                       predictions=predicted_classes,
                                       name='acc_op')
        optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
        train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
```

```
In [ ]: tf.summary.scalar('accuracy', accuracy[1])
        tf.summary.scalar('loss', loss)
        merged = tf.summary.merge_all()
```

The next operation is required in order to use reinitializable iterators but not for simple one-shot iterator.

```
In [ ]: init_op_train = iterator.make_initializer(dataset_train)
        init_op_test = iterator.make_initializer(dataset_test)
```

```
In [ ]: sess_debug = tf.InteractiveSession()
        sess_debug.run(tf.global_variables_initializer())
```

For some reason it seems to be necessary to add the following line though it wasn't in the original script. Otherwise I optain a FailedPreconditionError.

```
In [ ]: sess_debug.run(tf.local_variables_initializer())
```

```
In [ ]: writer_train = tf.summary.FileWriter(model_dir+'Basic_log/Plot_train/', sess_debug.graph
        writer_test = tf.summary.FileWriter(model_dir+'Basic_log/Plot_test/', sess_debug.graph)
```

We do the **training** of the model.

```
In [ ]: for j in range(15):
            print('Epoch {}'.format(j))
            # Initialize an iterator over the training dataset.
            sess_debug.run([init_op_train])
            print('Training')
            for i in range(100):
                sess_debug.run(train_op)
                if (i % 50 == 0):
                    summary, acc = sess_debug.run([merged, accuracy])
                    print("Accuracy = {}".format(acc[1]))
                    writer_train.add_summary(summary, j*100)

            print('Testing')
            # Initialize an iterator over the testing dataset.
            sess_debug.run(init_op_test)
            sess_debug.run(merged)
            summary, acc = sess_debug.run([merged, accuracy])
            print("Accuracy = {}".format(acc[1]))
            writer_test.add_summary(summary, j*100)
```

```
In [ ]: writer_debug.close()
```