Graph-Memory-Networks

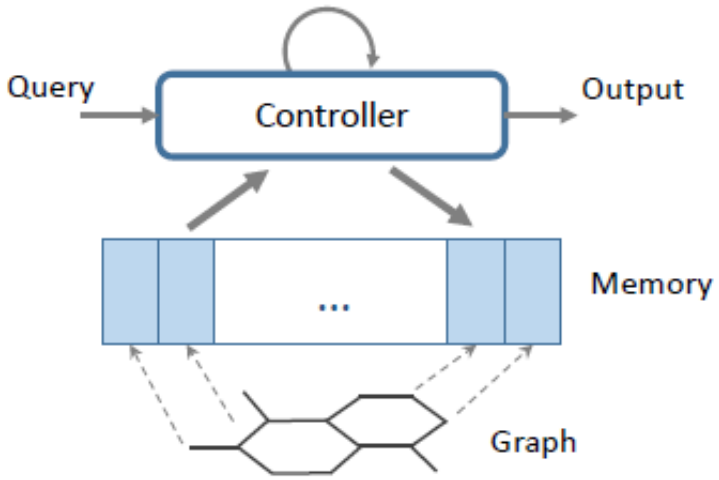
(文中所涉及代码见GitHub仓库)

2017Z8009061061董俊川 2017Z8009061069姜诚 2017Z8009061120张超 2017Z8009061116尹志雨

1.实验目的

本实验以论文Graph Memory Networks for Molecular Activity Prediction为参考，复现其中的核心网络部分。简要来说，本实验的主题是药物发现(Drug Discovery)，即给定一个药物的分子结构，判断该药物分子是否具有某种活性，比如抗癌性。

使用的主要方法是基于Deep Learning和Memory构建的模型，该模型的输入是分子结构(以Graph的形式呈现)，输出是预测结果。模型的结构主要由两部分组成Memory和Controller。第一部分是一个Memory模块，用来存储分子结构，其中Memory的每一个Cell对应存储一个原子特征，Cell之间存在邻接关系；第二部分是一个Controller，该Controller每次从Memory中读取数据，并结合邻接关系，将数据写入Memory中，经过T步的读取和写入操作，Memory中的数据逐渐被refined，最终输出该分子的表达，使用分类器做分类。整体的原理图如下图所示，具体的读入操作和写回操作见论文Graph Memory Networks for Molecular Activity Prediction。



2.数据集

原文为了测试多数据集联合训练的效果，共使用了9种数据集，本实验使用其中的AIDS Antiviral数据集。AIDS Antiviral是一个分子活性预测方面的数据集，共包含50000余个分子(Molecular每个分子代表一种药物)，每种分子提供该分子的分子结构，以及是否具备活性(Active和Inactive)。其中分子结构在本实验中被看做图结构，原子(Atom)可看做图结构中的节点(Node)，原子之间的化学键(Bond)相应看做为边(Edge)。按照实验要求，为了保证正例样本和负例样本数量的均衡，随机去掉一部分负例样本，使得最终样本数控制在10000个。数据集是以.SDF的形式提供的，打开文件，我们可以看到数据的呈现形式如下：

```
11122
  -OEChem-08181807252D

 15 15  0     0  0  0  0  0  0999 V2000
    2.0000    0.5000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    5.4641   -1.5000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.7320    0.5000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    2.8660    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    4.5981    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.7320    1.5000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    4.5981   -1.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    2.8660   -1.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.7320   -1.5000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    5.1350    0.3100    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    4.3520    1.5000    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    3.7320    2.1200    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    3.1120    1.5000    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    2.3291   -1.3100    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    3.7320   -2.1200    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  4  2  0  0  0  0
  2  7  2  0  0  0  0
  3  4  1  0  0  0  0
  3  5  2  0  0  0  0
  3  6  1  0  0  0  0
```

3.实验步骤

3.1特征提取

特征提取部分主要目的是对Memory做初始化操作，即初始化Memory中每一个Cell的值，由于Cell对应一个原子，因此本部分主要是提取原子特征。我们将原子的one-hot编码和原子的度(Degree)以及所附H原子数量做concat，同时也考虑了其所连接的化学键的类型。部分代码如下所示：

- 第一步，读取SDF文件，获取所有分子结构。该部分需要安装Rdkit工具，该工具是做化学分析非常流行的工具，支持python接口，可以在多种平台上运行。

```
from rdkit import Chem
suppl = Chem.SDMolSupplier(osp.join(cfg.INPUT.DATA_DIR,cfg.INPUT.SDF))
```

- 第二步，获取原子的one-hot特征，degree和H attached(本数据集中无)

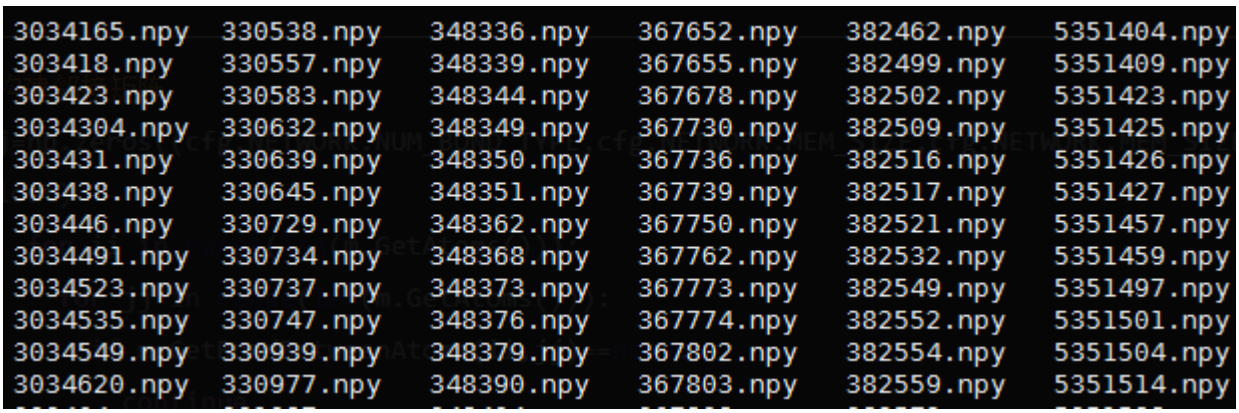
```
# atom的one-hot编码,degree,H attached
feature_mol=[]
    for atom in m.GetAtoms():
      feature_atom=[]
      one_hot = np.zeros(cfg.NETWORK.ONE_HOT_DIM,dtype=np.float)
      one_hot[atom.GetAtomicNum()]=1.0
      feature_atom.append(one_hot)
      degree_onehot = np.zeros(cfg.NETWORK.MAX_NUM_DEGREE,dtype=np.float)
      degree = len([x.GetAtomicNum() for x in atom.GetNeighbors()])
      degree_onehot[degree-1] = 1.0
      feature_atom.append(degree_onehot)
      feature_atom.append(bond_feature)
```

```
      # 整理feature_atom，并加入mem,
      feature_atom = np.concatenate(feature_atom)
      # padding一下feature_atom
      feature_mol.append(feature_atom)
    feature_mol=np.array(feature_mol)
    mol_pad_len = cfg.NETWORK.MEM_SIZE - feature_mol.shape[0]
    feature_mol = np.lib.pad(feature_mol,((0,mol_pad_len),
(0,0)),'constant',constant_values=(0.0,0.0))
```

- 第三步，构建邻接矩阵

```
# 构建邻接矩阵
adj=np.zeros((cfg.NETWORK.NUM_BOND_TYPE,cfg.NETWORK.MEM_SIZE,cfg.NETWORK.MEM_SIZE),dtype=
np.float)
    for ii in range(len(m.GetAtoms())):
      for jj in range(len(m.GetAtoms())):
        if m.GetBondBetweenAtoms(ii,jj)==None:
          continue
        else:
          if m.GetBondBetweenAtoms(ii,jj).GetBondType() ==
rdkit.Chem.rdchem.BondType.SINGLE:
            adj[0][ii][jj]=1.0
          elif m.GetBondBetweenAtoms(ii,jj).GetBondType() ==
rdkit.Chem.rdchem.BondType.DOUBLE:
            adj[1][ii][jj]=1.0
          elif m.GetBondBetweenAtoms(ii,jj).GetBondType() ==
rdkit.Chem.rdchem.BondType.TRIPLE:
            adj[2][ii][jj]=1.0
          else:
            adj[3][ii][jj]=1.0
```

- 第四步，保存，最终的特征保存为.npy格式，存在本地磁盘，方便构建模型时直接读入，生成的文件如下图所示。



3.2 读取batch数据

该部分使用TF的dataset接口，非常方便，使用generator源源不断的供给上文所提供的文件数据即可，部分代码如下：

```
def _generator():
  global cfg
  with open(osp.join(cfg.INPUT.INDEX_DIR,cfg.INPUT.TRAIN_INDEX)) as f:
    mol_list = f.readlines()
    print len(mol_list)
  for i in range(len(mol_list)):
    mol_id = mol_list[i].split()[0]
    mol_feature_path = osp.join(cfg.INPUT.DATA_DIR, mol_id+'.npy')
    if not os.path.exists(mol_feature_path):
      continue
    mol_adj_gt = np.load(mol_feature_path)
    mol_feature = mol_adj_gt[0]
    mol_adj = mol_adj_gt[1]
    mol_gt = mol_adj_gt[2]
    mol_feature=mol_feature[:,0:96]
    yield (mol_feature, mol_adj, mol_gt)
def get_dataset_iter(config):
  """读取数据，预处理，组成batch，返回
  """
  global cfg
  cfg = config
  dataset_train=tf.data.Dataset.from_generator(_generator,(tf.float32, tf.float32,
tf.float32),
(tf.TensorShape([150,96]),tf.TensorShape([4,150,150]),tf.TensorShape([2]) ))
  dataset_train =
dataset_train.repeat().shuffle(buffer_size=cfg.TRAIN.BATCH_SIZE*20).batch(cfg.TRAIN.BATCH
_SIZE).prefetch(buffer_size=10)
  iter_train = dataset_train.make_one_shot_iterator()
  return iter_train
```

3.3 构建模型文件

该部分为重点部分，模型文件存放于Graph-Memory-Networks\nets路径名为memorynet.py。该模型的pipeline主要由三部分组成，第一是读入一个query，此query在原文中是与task有关，如果不训练多任务，则此query也可以不输入；第二是以attention的方式读取memory的内容，读进controller当中；第三是写回memory，此部分最复杂，因为既要考虑批处理，又要考虑Cell之间由多种bond构成的邻接关系。需要注意的是，第二步和第三步是需要反复交替迭代执行的。

- 第一步，controller读入一个query，此部分代码片如下所示：

```
def _queryInput(cfg,query):
  with tf.variable_scope('query_input'):
    ctrl_state = tf.contrib.layers.fully_connected(query, cfg.NETWORK.CTRL_STATE_SIZE )
    tf.get_variable_scope().reuse_variables()

  return ctrl_state
```

- 第二步，读入的代码如下所示，此部分首先需要计算Memory中每一个cell的attention权重，然后依照权重大小对所有cell加权求和，作为controller的读取结果，相对简单。

```
def _attention(cfg,mem,ctrl_state):
  with tf.variable_scope('attention'):
    a = tf.reshape(tf.contrib.layers.fully_connected(mem,
cfg.NETWORK.DIM_A,biases_initializer=None),\
                  [-1,cfg.NETWORK.MEM_SIZE,cfg.NETWORK.DIM_A]) + \
        tf.reshape(tf.contrib.layers.fully_connected(ctrl_state, cfg.NETWORK.DIM_A),\
                  [-1,1,cfg.NETWORK.DIM_A])
    p = tf.reshape(tf.contrib.layers.fully_connected(a,1),\
                  [-1, cfg.NETWORK.MEM_SIZE])
    p = tf.nn.softmax(p) #shape=[batch_size*mem_size]
    tf.get_variable_scope().reuse_variables()
  return p

def _read(cfg,mem,ctrl_state):
  with tf.variable_scope('read'):
    p=_attention(cfg,mem,ctrl_state)
    p=tf.reshape(p,[-1, cfg.NETWORK.MEM_SIZE,1])
    summary_m=tf.reduce_mean(tf.multiply(mem,p),axis=1) #after multiply,shape=
[batch_size*max_nodes*cfg.dim_m].shape=[batch_size*dim_m]
    ctrl_state =
tf.contrib.layers.fully_connected(summary_m,cfg.NETWORK.CTRL_STATE_SIZE,biases_initialize
r=None)+\
                tf.contrib.layers.fully_connected(ctrl_state,cfg.NETWORK.CTRL_STATE_SIZE)
    tf.get_variable_scope().reuse_variables()
  return ctrl_state
```

- 第三步，写入操作，此处相对复杂，重点在于写入一个cell的时候，需要考虑该cell多种邻接点(每一种bond对应一种类型的邻接点)，而且cell需要和bond特征做concat。代码片如下所示：

```
def _write(cfg,mem,ctrl_state,adj):
  with tf.variable_scope('write'):
    mem_old = mem
    mem = tf.reshape(tf.contrib.layers.fully_connected(mem,cfg.NETWORK.CELL_SIZE),[-1,
cfg.NETWORK.MEM_SIZE, cfg.NETWORK.CELL_SIZE]) +\

tf.reshape(tf.contrib.layers.fully_connected(tf.reshape(tf.concat([ctrl_state]*cfg.NETWOR
K.MEM_SIZE,axis=1),
[-1,cfg.NETWORK.MEM_SIZE,cfg.NETWORK.CTRL_STATE_SIZE]),cfg.NETWORK.CELL_SIZE),[-1,
cfg.NETWORK.MEM_SIZE,cfg.NETWORK.CELL_SIZE])
```

```
    for bond in range(cfg.NETWORK.NUM_BOND_TYPE):
      with tf.variable_scope('bond_type'):
        mem_exp = tf.reshape(tf.concat([mem]*cfg.NETWORK.MEM_SIZE,axis=1),[-1,
cfg.NETWORK.MEM_SIZE,cfg.NETWORK.MEM_SIZE,cfg.NETWORK.CELL_SIZE])
        # concat  bond特征
        bond_vec = np.zeros((cfg.TRAIN.BATCH_SIZE, cfg.NETWORK.MEM_SIZE,
cfg.NETWORK.MEM_SIZE, cfg.NETWORK.BOND_SIZE),dtype=np.float)
        bond_vec[:,:,:,bond] = 1.0
        bond_vec = tf.cast(tf.convert_to_tensor(bond_vec),tf.float32)
        mem_bond = tf.concat([mem_exp,bond_vec],axis=3)
        # 制作mask
        mask = adj[:,bond,:,:]  #shape=[batch_size,mem_size,mem_size]
        mask = tf.cast(tf.reshape(mask,[-1, cfg.NETWORK.MEM_SIZE,
cfg.NETWORK.MEM_SIZE,1]),tf.float32)
        # 保留每个cell的邻接cell
        mem_bond = tf.multiply(mem_bond,mask)
        # 每个cell的邻接cell求和
        mem_bond_summ = tf.reduce_mean(mem_bond,axis=2) #shape
[batch_size,mem_size,bond_size]
        mem +=
tf.reshape(tf.contrib.layers.fully_connected(mem_bond_summ,cfg.NETWORK.CELL_SIZE),[-1,
cfg.NETWORK.MEM_SIZE,cfg.NETWORK.CELL_SIZE])
        tf.get_variable_scope().reuse_variables()
    tf.get_variable_scope().reuse_variables()
  # 更新cell值
  mem = cfg.NETWORK.BETA*mem + (1-cfg.NETWORK.BETA)*mem_old
  return mem
```

- 第四步，循环迭代地执行读取和写回操作，如下代码片所示：

```
def inference(self, mem, adj, query):
    """mem:batch_size*mem_size*cell_size
       adj:batch_size*mem_size*num_bond_type*mem_size
    """
    # 定义controller初始状态
    ctrl_state = tf.constant(0.0,shape=
[self.cfg.TRAIN.BATCH_SIZE,self.cfg.NETWORK.CTRL_STATE_SIZE],dtype=tf.float32)
    # 输入query
    ctrl_state = _queryInput(self.cfg,query)
    # 循环读出及写入
    for step in range(T):
      with tf.variable_scope('GraphMemNet',reuse=tf.AUTO_REUSE) as scope_read_wrt:
        # 读入
        ctrl_state_old = ctrl_state
        ctrl_state = _read(self.cfg,mem,ctrl_state)
        ctrl_state = self.cfg.NETWORK.ALPH*ctrl_state + (1-
self.cfg.NETWORK.ALPH)*ctrl_state_old
```

```
        # 写出
        mem = _write(self.cfg,mem,ctrl_state,adj)
        scope_read_wrt.reuse_variables()
    # 最后一次读出
    with tf.variable_scope('GraphMemNet',reuse=True) as scope_read_wrt:
      ctrl_state = _read(self.cfg,mem,ctrl_state)
    with tf.variable_scope('output') as scope_output:
      return tf.contrib.layers.fully_connected(ctrl_state,2)
```

3.4编写训练即验证文件

该部分主要调用上述函数或模块，主要流程步骤为读取数据，加载模型，前向传播，获取结果，求取loss和反向传播。

- 读取数据，部分代码如下所示：

```
ite_train = reader.get_dataset_iter(cfg)
mem, mem_adj, gt = ite_train.get_next()
```

- 加载模型，前向传播，获取结果，部分代码如下所示：

```
graph_mem_net = GraphMemNet(cfg)
logits = graph_mem_net.inference(mem, mem_adj, query)
```

- 求取loss和反向传播，部分代码如下所示：

```
loss = loss_func(cfg,logits, gt, regularization= True)
tf.summary.scalar('loss',loss)
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
  train_step = opt.minimize(loss,
global_step=global_step,var_list=tf.trainable_variables())
```

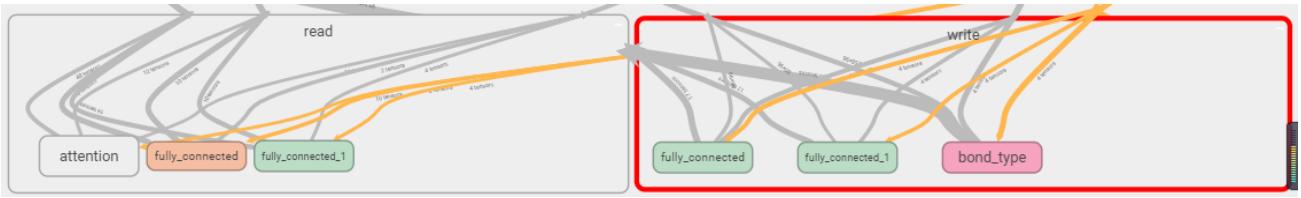
4训练及结果

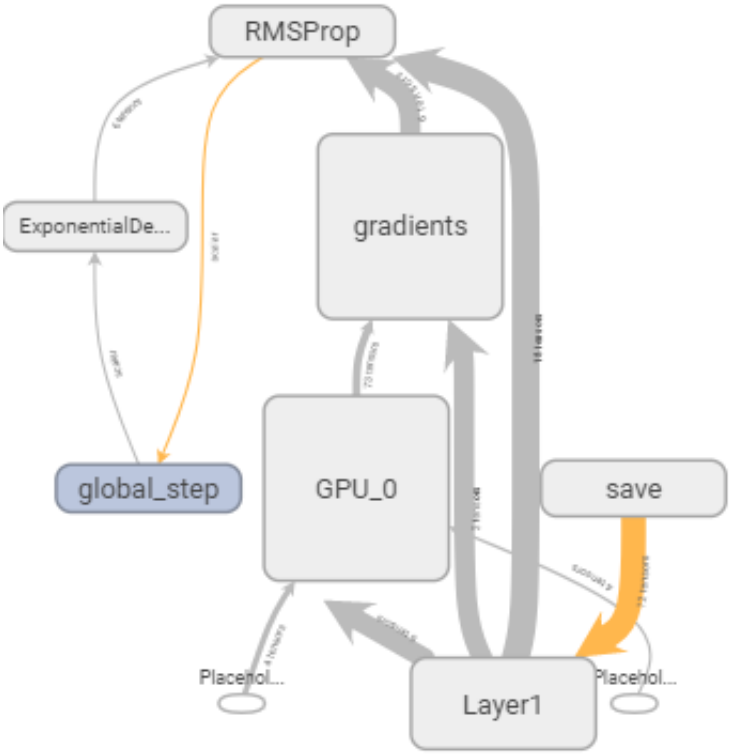
实验的部分重要参数配置如下。

```
__C = edict()
cfg = __C  # 引用传递
__C.GPUS = '0'
__C.SUMMARY_DIR = '/data/yinzhiyu/results/Graph-Memory-Networks/log'
__C.INPUT = edict()
__C.INPUT.QUERY_SIZE = 2
__C.INPUT.DATA_DIR = '/data/yinzhiyu/datasets/167'

__C.INPUT.INDEX_DIR = '/home/yzy/workspace/Graph-Memory-Networks/data'
```

```
__C.INPUT.TRAIN_INDEX = 'Index_AID_167_database_train.txt'
__C.INPUT.TEST_INDEX = 'Index_AID_167_database_test.txt'
__C.INPUT.SDF = 'AID_167_database_all.sdf'
__C.INPUT.INDEX = 'Index_AID_167_database_all.txt'

__C.TRAIN = edict()
__C.TRAIN.BATCH_SIZE = 3
__C.TRAIN.LEARNING_RATE_BASE = 0.05
__C.TRAIN.DECAY_STEP = 1200
__C.TRAIN.DECAY_RATE = 0.1
__C.TRAIN.DROPOUT_KEEP_PROB = 0.5
__C.TRAIN.REGULARIZATION_SCALE = 0.0001
__C.TRAIN.MAX_ITE = 40000
__C.TRAIN.MOMENTUM = 0.9
__C.TRAIN.MAX_MODELS_TO_KEEP = 50
__C.TRAIN.SAVED_MODEL_PATTERN = '/data/yinzhiyu/results/Graph-Memory-Networks/models'

__C.VALID = edict()
__C.VALID.BATCH_SIZE = 1

__C.NETWORK = edict()
__C.NETWORK.STEPS = 10
__C.NETWORK.DIM_A = 10
__C.NETWORK.MEM_SIZE = 150
__C.NETWORK.CELL_SIZE = 96
__C.NETWORK.ALPH = 0.8
__C.NETWORK.BETA = 0.8
__C.NETWORK.CTRL_STATE_SIZE = 50
__C.NETWORK.BOND_SIZE = 162
__C.NETWORK.ONE_HOT_DIM = 90
__C.NETWORK.NUM_BOND_TYPE = 4
__C.NETWORK.MAX_NUM_DEGREE = 6
```

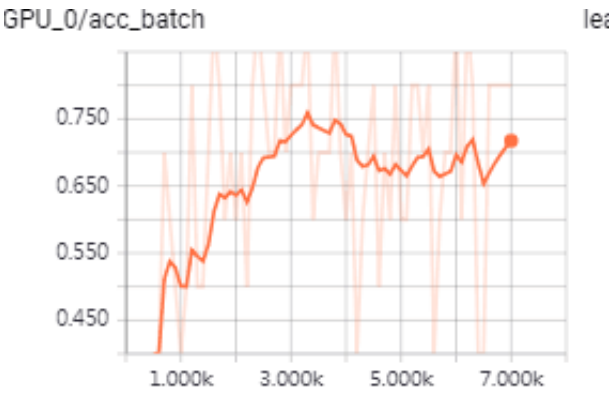
下图是在tensorboard中所展示的模型示意图，可以看到模型主体部分由query input，read和write组成。

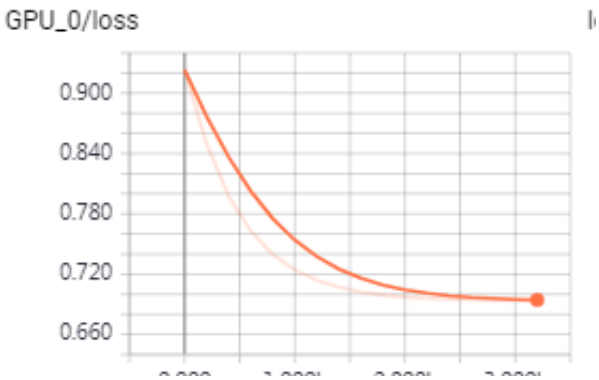




训练时的acc和loss下降趋势，由于实验设备的限制，无法设置一个较大的batch，所以模型收敛的比较曲折，收敛的效果也不是非常理想，所得结果和论文中在该数据集上单任务训练有3到4个百分点的差距，最终的F1-score值有60.22%，论文中为63%。

```
F1 score is 60.22%
```




GPU_0/loss
0.900
0.840
0.780
0.720
0.660