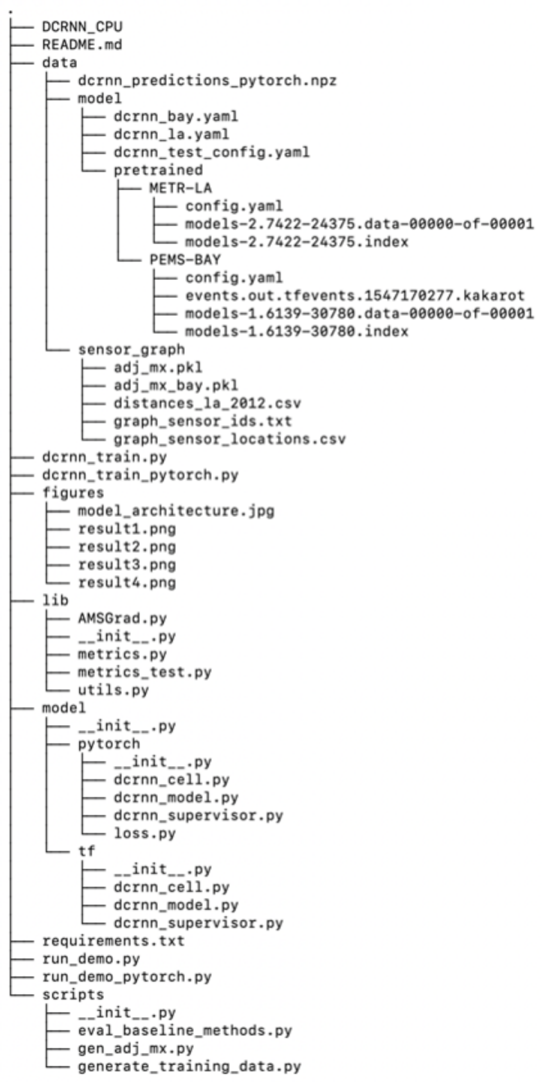
DCRNN_PyTorch代码解析

总体框架



```
.
├── DCRNN_CPU
├── README.md
├── data
│   ├── dcrnn_predictions_pytorch.npz
│   ├── model
│   │   ├── dcrnn_bay.yaml
│   │   ├── dcrnn_la.yaml
│   │   ├── dcrnn_test_config.yaml
│   │   └── pretrained
│   │       ├── METR-LA
│   │       │   ├── config.yaml
│   │       │   ├── models-2.7422-24375.data-00000-of-00001
│   │       │   └── models-2.7422-24375.index
│   │       └── PEMS-BAY
│   │           ├── config.yaml
│   │           ├── events.out.tfevents.1547170277.kakarot
│   │           ├── models-1.6139-30780.data-00000-of-00001
│   │           └── models-1.6139-30780.index
│   └── sensor_graph
│       ├── adj_mx.pkl
│       ├── adj_mx_bay.pkl
│       ├── distances_la_2012.csv
│       ├── graph_sensor_ids.txt
│       └── graph_sensor_locations.csv
├── dcrnn_train.py
├── dcrnn_train_pytorch.py
├── figures
│   ├── model_architecture.jpg
│   ├── result1.png
│   ├── result2.png
│   ├── result3.png
│   └── result4.png
├── lib
│   ├── AMSGrad.py
│   ├── __init__.py
│   ├── metrics.py
│   ├── metrics_test.py
│   └── utils.py
├── model
│   ├── __init__.py
│   ├── pytorch
│   │   ├── __init__.py
│   │   ├── dcrnn_cell.py
│   │   ├── dcrnn_model.py
│   │   ├── dcrnn_supervisor.py
│   │   └── loss.py
│   └── tf
│       ├── __init__.py
│       ├── dcrnn_cell.py
│       ├── dcrnn_model.py
│       └── dcrnn_supervisor.py
├── requirements.txt
├── run_demo.py
├── run_demo_pytorch.py
└── scripts
    ├── __init__.py
    ├── eval_baseline_methods.py
    ├── gen_adj_mx.py
    └── generate_training_data.py
```

1. data存放模型用到的数据。其中METR-LA和PEMS-BAY文件夹存放的是scripts中generate_training_data.py生成的数据，model保存的是训练好的模型及其设置参数，sensor_graph保存的是传感器的数据，其中graph_sensor_ids.txt保存了所有数据中所使用的传感器的id，distances_la_2012.csv以csv格式保存了传感器之间的距离。metr-la.h5和pems-bay.h5是两个城市的交通数据。
2. figures文件存放的是模型总体结构的图片。
3. lib文件包含模型用到的工具函数、优化算法和评价指标。
4. model文件中dcrnn_supervisor.py是训练过程中控制整个训练流程的文件，dcrnn_model.py定义了dcrnn模型，dcrnn_cell.py定义了模型中的cell单元，其中实现了核心的扩散卷积。
5. scripts包含数据预处理代码，其中gen_adj_mx.py用于生成图的邻接矩阵，generate_training_data.py用于生成训练、验证和测试数据。
6. dcrnn_train_pytorch.py是用来训练模型的，run_demo_pytorch.py是用来跑一下已经训练好的模型。

运行说明

需要安装的依赖

- torch
- scipy>=0.19.0
- numpy>=1.12.1
- pandas>=0.19.2
- pyyaml
- statsmodels
- tensorflow>=1.3.0
- tables
- future

依赖可以依照下面的命令直接安装:

```
pip install -r requirements.txt
```

使用pip安装的一些建议

- 得保证pip对应的环境不是base环境，不然会和其他环境冲突，一定要用which python或者where python检查一下是否是自己conda环境内的python。
- 最好是使用conda一个个安装

与tensorflow的效率比较

使用MAE损失函数

Horizon	Tensorflow	Pytorch
1 Hour	3.69	3.12
30 Min	3.15	2.82
15 Min	2.77	2.56

数据预处理

METR-LA和PEMS-BAY的数据文件, `metr-la.h5` 和 `pems-bay.h5`, 下载地址 Baidu Yun, 把他们放在 `data/` 文件夹.
`*.h5` 以 `panads.DataFrame` 格式存放数据，使用的是 `HDF5` 文件格式。数据框如下所示:

	sensor_0	sensor_1	sensor_2	sensor_n
2018/01/01 00:00:00	60.0	65.0	70.0	...
2018/01/01 00:05:00	61.0	64.0	65.0	...
2018/01/01 00:10:00	63.0	65.0	60.0	...
...	...	...	...	...

这是关于HDF5数据格式的参考资料： Using HDF5 with Python.

运行下面命令生成 train/test/val 数据集，数据集位于 `data/{METR-LA,PEMS-BAY}/{train,val,test}.npz`.

```
# 生成数据集文件夹
mkdir -p data/{METR-LA,PEMS-BAY}

# METR-LA
python -m scripts.generate_training_data --output_dir=data/METR-LA --traffic_df_filename=data/metr-la.h5

# PEMS-BAY
python -m scripts.generate_training_data --output_dir=data/PEMS-BAY --traffic_df_filename=data/pems-bay.h5
```

运行预训练模型模型

```
# METR-LA
python run_demo_pytorch.py --config_filename=data/model/pretrained/METR-LA/config.yaml

# PEMS-BAY
python run_demo_pytorch.py --config_filename=data/model/pretrained/PEMS-BAY/config.yaml
```

DCRNN模型的预测结果位于 `data/results/dcrnn_predictions`.

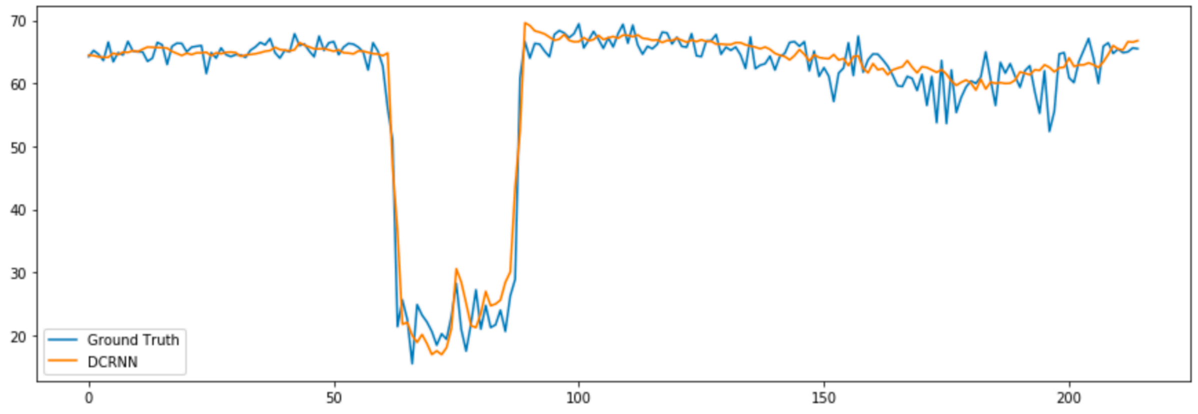
模型训练

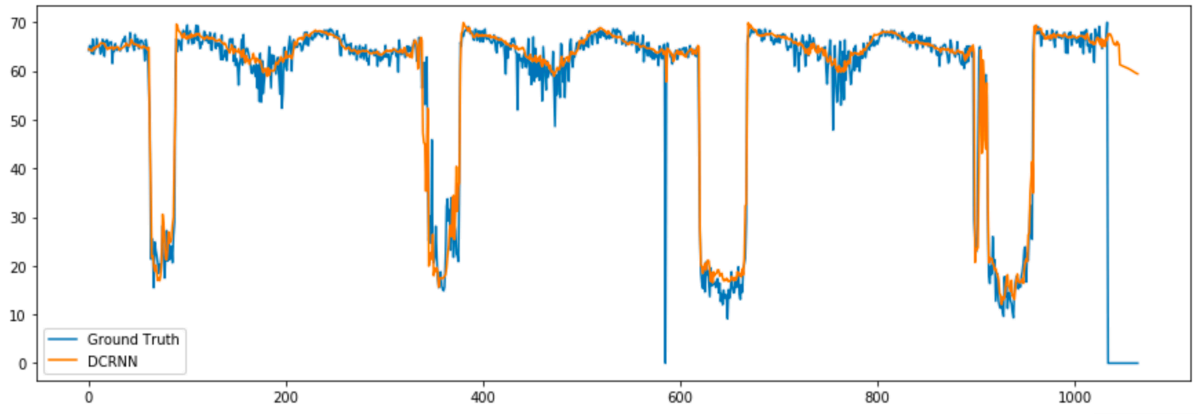
```
# METR-LA
python dcrnn_train_pytorch.py --config_filename=data/model/dcrnn_la.yaml

# PEMS-BAY
python dcrnn_train_pytorch.py --config_filename=data/model/dcrnn_bay.yaml
有可能梯度爆炸，临时的解决办法是在爆炸前中断后断点续训，或者降低学习速率。
```

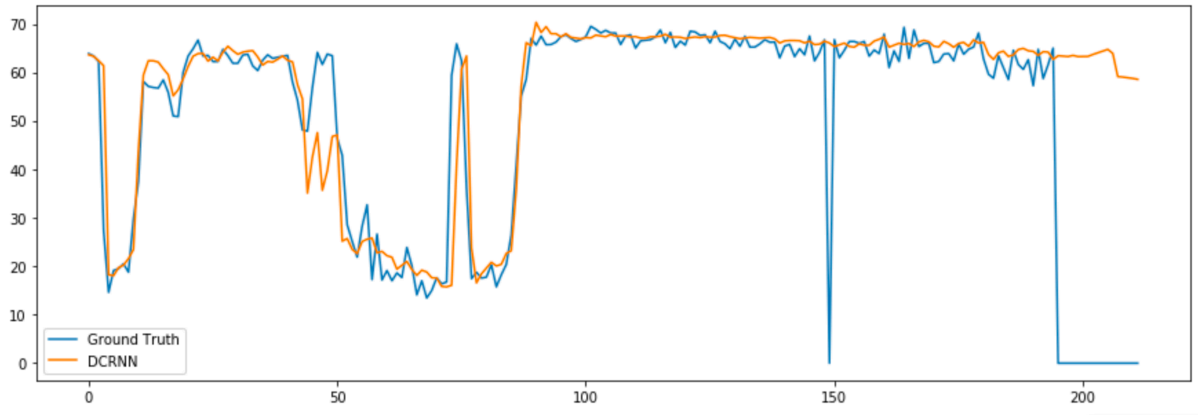
模型评估

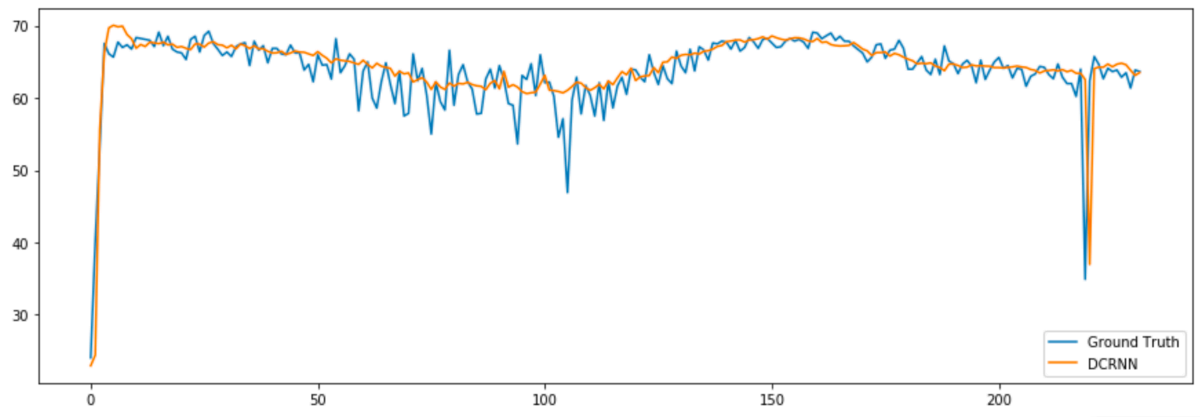
```
# METR-LA
python -m scripts.eval_baseline_methods --traffic_reading_filename=data/metr-la.h5
```

PyTorch运行结果









scripts文件

主要包括数据预处理的代码。

gen_adj_mx.py

gen_adj_mx.py是用来生成邻接矩阵的。

main函数里面首先定义了命令行参数，如

```
parser.add_argument('--sensor_ids_filename', type=str,
default='data/sensor_graph/graph_sensor_ids.txt', help='File containing sensor
ids separated by comma.')
```

其中的参数分别表示命令的名称，类型，默认值，帮助信息。 定义后就可以用定义后就可以用 `args.sensor_ids_filename` 就可以取相应的值。然后通过 `get_adjacency_matrix` 函数生成邻接矩阵，最后保存到 pickle 的文件。 `get_adjacency_matrix(distance_df, sensor_ids, normalized_k=0.1)` 具体的解释:

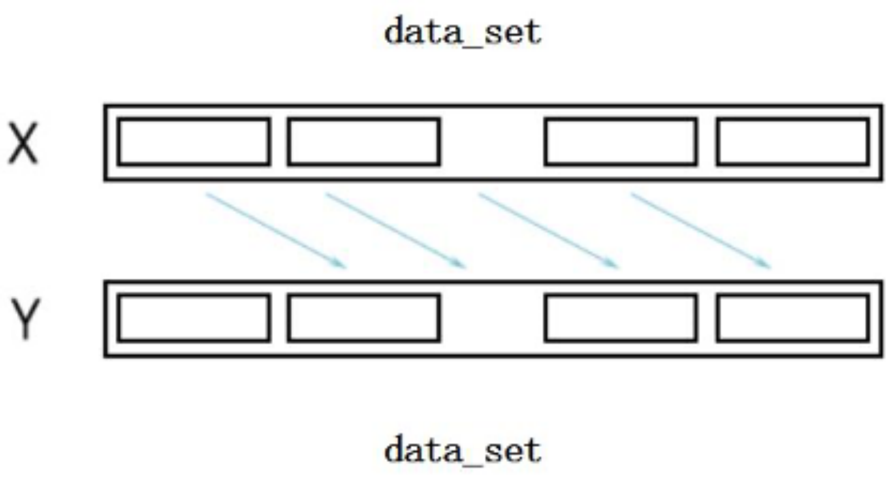
1. 初始化: `dist_mx` 的大小为 `[num_sensors,num_sensors]` 其中 `(num_sensors = len(sensor_ids))`，初始值为 `inf` (无限大)
2. 构建传感器id到索引映射:遍历所有对象 `sensor_ids` ，将每一个 `i` 填充至 `sensor_id_to_ind[sensor_id]` ，即 `sensor_id_to_ind[sensor_id] = i`。
3. 用距离填充矩阵中的单元格:遍历所有 `distance_df.values` ，从 `sensor_id_to_ind[row[0]]` (`from`)到 `sensor_id_to_ind[row[1]]` (`to`)，填充 `row[2]` (`distance`)。
4. 对 `dist_mx` 进行归一化，以减少计算量;对于稀疏性，将低于阈值(例如k)的项设置为零，如: `adj_mx[adj_mx < normalized_k] = 0`

generate_training_data.py

generate_training_data.py是用来生成训练数据的。

1. 以1小时为统计间隔，记录每天的间隔小时数 `time_in_day` ，并添加到 `data_list`
2. 以1天为统计间隔，记录每周的间隔天数 `day_in_week` ，并添加到 `data_list`
3. 在补偿之后的时间范围内 `(range(min_t,max_t))`，记录每天的时间特征指标到 `x`， `y`

4. 将数据写入npz文件(`num_test = 6831`)使用最后的6831示例作为测试，其中1/8用于验证，其余7/8用于训练。



---

model文件

模型总体框架如下图所示，总体是一个encoder-decoder结构，encoder和decoder里面各包含有两个RNNCell，每个RNNCell包含有64个units，每个units里面包含207个结点(传感器个数)。其中用到了scheduled sampling，即每次以一定的概率用模型的预测值作为下一个cell的输入进行预测，并且这个概率值随着时间增大。GO是初始化为全0的矩阵，即第一次用模型的预测值作为下一个cell的输入进行预测的时候是取全0的值作为输入。

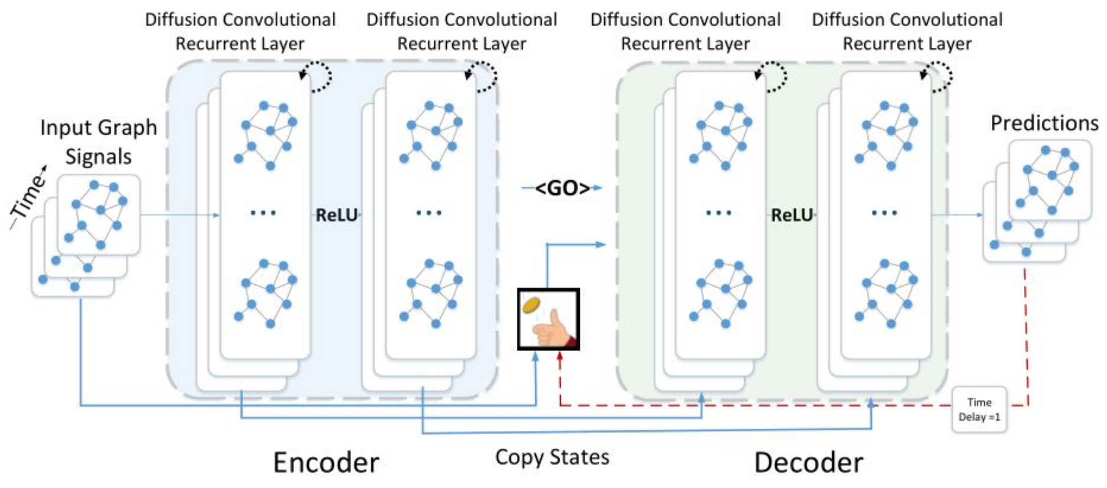
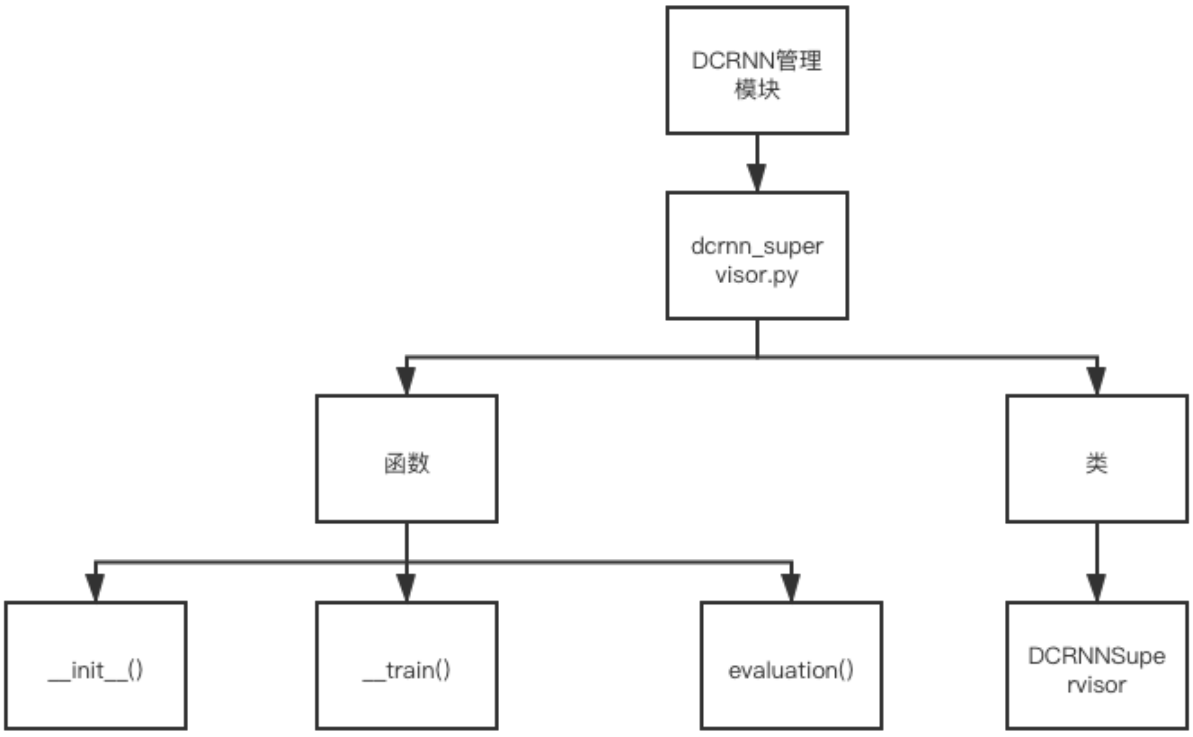

Diffusion Convolutional Recurrent Layer
Diffusion Convolutional Recurrent Layer
Diffusion Convolutional Recurrent Layer
Diffusion Convolutional Recurrent Layer
Input Graph Signals
Time
ReLU
<GO>
ReLU
Predictions
Time Delay =1
Encoder
Copy States
Decoder

Figure 2: System architecture for the *Diffusion Convolutional Recurrent Neural Network* designed for spatiotemporal traffic forecasting. The historical time series are fed into an encoder whose final states are used to initialize the decoder. The decoder makes predictions based on either previous ground truth or the model output.

dcrnn_supervisor.py



用于管理整个训练过程(从输入到输出)，即定义整个图。定义了一个DCRNNSupervisor类，它保存了 训练过程的参数，定义了记录训练状态的logging，然后将数据放入DCRNN模型中进行处理，得到结果后计算损失，获得梯度，进行梯度裁剪，更新梯度，保存参数。

```
class DCRNNSupervisor:
    """
        类名：DCRNNSupervisor
        主要成员：
        1、log:初始化能记录、打印log的self.__logger类
        2、data:加载数据集的类，该类包括Dataloader和scaler
        3、节点数，输入特征向量维度，输入步长，输出维度，输出步长。
        4、是否使用课程式学习 参考资料
https://blog.csdn.net/qq_30219017/article/details/89090690
        5、实例化的DCRNN模型
        主要方法：
        1、train(self, **kwargs) 该方法将外部的参数与内部合并起来并调用
_train(**kwargs)
        2、evaluate(self, dataset='val', batches_seen=0) 将模型变为评估模式，迭代测
试集计算准确率和损失值
        3、_train(self, base_lr,
               steps, patience=50, epochs=100, lr_decay_ratio=0.1, log_every=1,
save_model=1,
               test_every_n_epochs=10, epsilon=1e-8, **kwargs)
    """
```

(1)类初始化

```
def __init__(self, adj_mx, **kwargs):
    """
    函数名：__init__
    函数功能描述：初始化DCRNNSupervisor成员
    函数参数：邻接矩阵和kwargs
    函数返回值：NULL
    """
    self._kwargs = kwargs#kwargs是传入的yaml文件内的参数该文件保存了data, model,
train三部分信息。
    self._data_kwargs = kwargs.get('data')
    self._model_kwargs = kwargs.get('model')
    self._train_kwargs = kwargs.get('train')

    self.max_grad_norm = self._train_kwargs.get('max_grad_norm', 1.)

    # 对log类初始化
    self._log_dir = self._get_log_dir(kwargs)
    self._writer = SummaryWriter('runs/' + self._log_dir)

    log_level = self._kwargs.get('log_level', 'INFO')
    self._logger = utils.get_logger(self._log_dir, __name__, 'info.log',
level=log_level)

    # 对数据集信息初始化
    self._data = utils.load_dataset(**self._data_kwargs)#实例化dataset类
    self.standard_scaler = self._data['scaler']#实例化标准化类

    self.num_nodes = int(self._model_kwargs.get('num_nodes', 1))#节点数
    self.input_dim = int(self._model_kwargs.get('input_dim', 1))#输入特征向量维度
    self.seq_len = int(self._model_kwargs.get('seq_len'))  # 输入序列步长
    self.output_dim = int(self._model_kwargs.get('output_dim', 1)) #输出维度
    self.use_curriculum_learning = bool(
      self._model_kwargs.get('use_curriculum_learning', False))#是否使用课程学习
    self.horizon = int(self._model_kwargs.get('horizon', 1))  #输出序列步长

    # setup model
    dcrnn_model = DCRNNModel(adj_mx, self._logger, **self._model_kwargs)#DCRNN
模型实例化
    self.dcrnn_model = dcrnn_model.cuda() if torch.cuda.is_available() else
dcrnn_model
    self._logger.info("Model created")

    self._epoch_num = self._train_kwargs.get('epoch', 0)#训练轮数
    if self._epoch_num > 0:
      self.load_model()
```

(2)训练模块

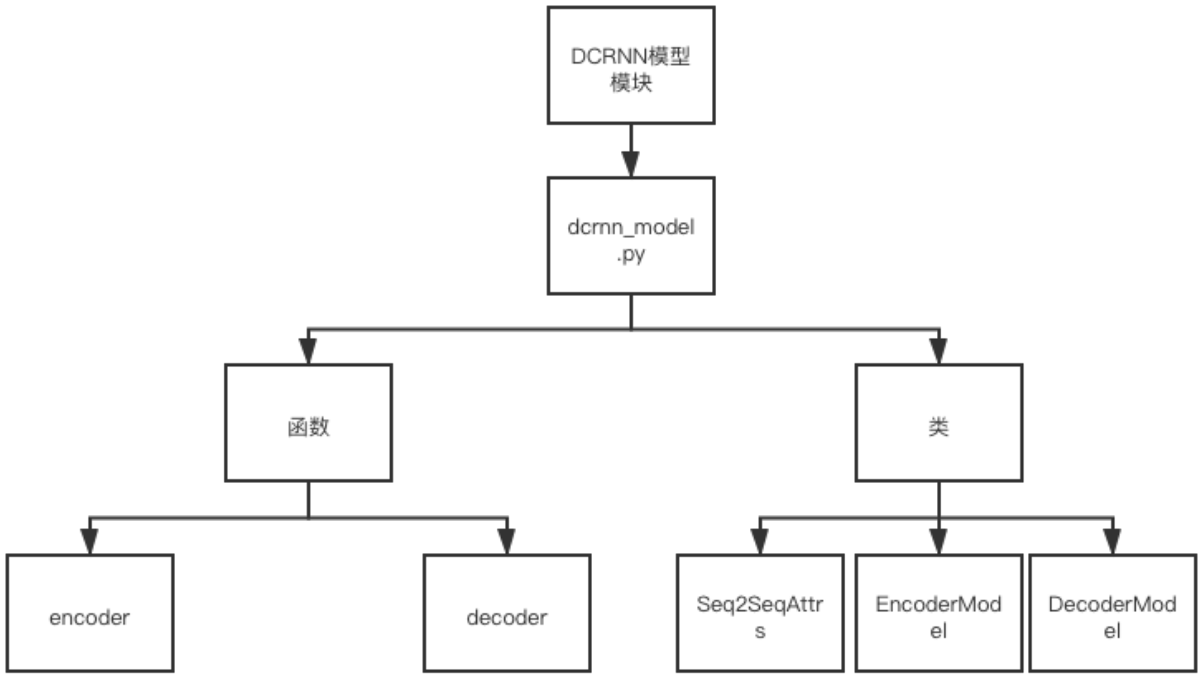
```
def _train(self, base_lr,
            steps, patience=50, epochs=100, lr_decay_ratio=0.1, log_every=1,
save_model=1,
            test_every_n_epochs=10, epsilon=1e-8, **kwargs):
    """
    函数名：_train
    函数功能描述：训练流程
    函数参数：steps, patience=50, epochs=100, lr_decay_ratio=0.1, log_every=1,
save_model=1,
            test_every_n_epochs=10, epsilon=1e-8, **kwargs
    函数返回值：NULL
    """
    # steps is used in learning rate - will see if need to use it?
    min_val_loss = float('inf')
    wait = 0
    optimizer = torch.optim.Adam(self.dcrnn_model.parameters(), lr=base_lr,
eps=epsilon)#定义adam优化器

    lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,
milestones=steps,
                                                        gamma=lr_decay_ratio)#
按需调整学习率 MultiStepLR 调整学习率的一些小tips
https://liumin.blog.csdn.net/article/details/85143614

    self._logger.info('Start training ...')

    # this will fail if model is loaded with a changed batch_size
    num_batches = self._data['train_loader'].num_batch#训练批次数
    self._logger.info("num_batches:{}".format(num_batches))

    batches_seen = num_batches * self._epoch_num

    for epoch_num in range(self._epoch_num, epochs):

        self.dcrnn_model = self.dcrnn_model.train()#模型设置为训练模式

        train_iterator = self._data['train_loader'].get_iterator()#获取生成训练集
的迭代器
        losses = []

        start_time = time.time()

        for _, (x, y) in enumerate(train_iterator):
            optimizer.zero_grad()#清空梯度

            x, y = self._prepare_data(x, y)#数据预处理

            output = self.dcrnn_model(x, y, batches_seen)#前向传播

            if batches_seen == 0:
              # this is a workaround to accommodate dynamically registered
parameters in DCGRUCell
                optimizer = torch.optim.Adam(self.dcrnn_model.parameters(),
lr=base_lr, eps=epsilon)#定义adam优化器

                loss = self._compute_loss(y, output)#计算损失函数

                self._logger.debug(loss.item())

                losses.append(loss.item())#记录损失函数

                batches_seen += 1
                loss.backward()#反向传播

                # gradient clipping - this does it in place
                torch.nn.utils.clip_grad_norm_(self.dcrnn_model.parameters(),
self.max_grad_norm)#梯度裁剪

                optimizer.step()#更新参数
                self._logger.info("epoch complete")
                lr_scheduler.step()#更新学习率
                self._logger.info("evaluating now!")

                val_loss, _ = self.evaluate(dataset='val',
batches_seen=batches_seen)#评估验证集

                end_time = time.time()

                self._writer.add_scalar('training loss',
                                        np.mean(losses),
                                        batches_seen)
                #写入log信息 保存验证集上最小损失的模型
                if (epoch_num % log_every) == log_every - 1:
                    message = 'Epoch [{}/{}] ({}) train_mae: {:.4f}, val_mae:
{:.4f}, lr: {:.6f}, ' \
                    '{:.1f}s'.format(epoch_num, epochs, batches_seen,
                                     np.mean(losses), val_loss,
lr_scheduler.get_lr()[0],
                                     (end_time - start_time))
                    self._logger.info(message)

                    if (epoch_num % test_every_n_epochs) == test_every_n_epochs
- 1:
                        test_loss, _ = self.evaluate(dataset='test',
batches_seen=batches_seen)
                        message = 'Epoch [{}/{}] ({}) train_mae: {:.4f},
test_mae: {:.4f},  lr: {:.6f}, ' \
                        '{:.1f}s'.format(epoch_num, epochs, batches_seen,
                                          np.mean(losses), test_loss,
lr_scheduler.get_lr()[0],
                                          (end_time - start_time))
                        self._logger.info(message)

                        if val_loss < min_val_loss:
                            wait = 0
                            if save_model:
                                model_file_name = self.save_model(epoch_num)
                                self._logger.info(
                                  'Val loss decrease from {:.4f} to {:.4f}, '
                                  'saving to {}'.format(min_val_loss, val_loss,
model_file_name))
                                min_val_loss = val_loss

                                elif val_loss >= min_val_loss:
                                    wait += 1
                                    if wait == patience:
                                        self._logger.warning('Early stopping at
epoch: %d' % epoch_num)
                                        break
```

(3)评估模块

```
def evaluate(self, dataset='val', batches_seen=0):
  """
   函数名: evaluate
   函数功能描述: 评估模型
   函数参数:  dataset='val', batches_seen=0
   函数返回值: NULL
   """
  with torch.no_grad():
    self.dcrnn_model = self.dcrnn_model.eval()#评估模式

    val_iterator = self._data['{}_loader'.format(dataset)].get_iterator()#生成数
据集
    losses = []

    y_truths = []
    y_preds = []

    for _, (x, y) in enumerate(val_iterator):
      x, y = self._prepare_data(x, y)#数据预处理

      output = self.dcrnn_model(x)#前向传播
      loss = self._compute_loss(y, output)#计算损失函数
      losses.append(loss.item())
      y_truths.append(y.cpu())#记录标签
      y_preds.append(output.cpu())#记录预测值

      mean_loss = np.mean(losses)

      self._writer.add_scalar('{} loss'.format(dataset), mean_loss,
batches_seen)

      y_preds = np.concatenate(y_preds, axis=1)
      y_truths = np.concatenate(y_truths, axis=1)  # 在batch_size维拼接

      y_truths_scaled = []
      y_preds_scaled = []
      for t in range(y_preds.shape[0]):
        y_truth = self.standard_scaler.inverse_transform(y_truths[t])
        y_pred = self.standard_scaler.inverse_transform(y_preds[t])
        y_truths_scaled.append(y_truth)
        y_preds_scaled.append(y_pred)

        return mean_loss, {'prediction': y_preds_scaled, 'truth':
y_truths_scaled}
```

dcrnn_model.py

这部分调用了两次DCGRUCell，它分别返回了一个cell和一个cell_with_projection(最后一层cell要进行预测，需要将维度变为预测值的维度)，定义了一个encoder(包含两个cell)和decoder(包含一个cell 和一个cell_with_projection)。每个RNNcell由64个units构成，每个units有207个结点。



(1)Seq2Seq模块

```
class Seq2SeqAttrs:#定义父类，初始化Seq2Seq模型各项参数
    """
        类名：Seq2SeqAttrs
        主要成员：
        1、邻接矩阵
        2、随机游走步长
        3、卷积核类型
        4、节点数
        5、rnn层数
        6、rnn单元数
    """
    def __init__(self, adj_mx, **model_kwargs):
        self.adj_mx = adj_mx
        self.max_diffusion_step = int(model_kwargs.get('max_diffusion_step', 
2))
        self.cl_decay_steps = int(model_kwargs.get('cl_decay_steps', 1000))
        self.filter_type = model_kwargs.get('filter_type', 'laplacian')
        self.num_nodes = int(model_kwargs.get('num_nodes', 1))
        self.num_rnn_layers = int(model_kwargs.get('num_rnn_layers', 1))
        self.rnn_units = int(model_kwargs.get('rnn_units'))
        self.hidden_state_size = self.num_nodes * self.rnn_units
```

(2)EncoderModel模块

```
class EncoderModel(nn.Module, Seq2SeqAttrs):
    """
        类名：EncoderModel
        主要成员：
        1、输入维度
        2、序列长度
        3、dcgru层[dcgrucell , dcgrucell ... dcgrucell]
        主要方法：
        1、forward(self, inputs, hidden_state=None)
    """
    def __init__(self, adj_mx, **model_kwargs):
        nn.Module.__init__(self)
        Seq2SeqAttrs.__init__(self, adj_mx, **model_kwargs)
        self.input_dim = int(model_kwargs.get('input_dim', 1))#输入维度
        self.seq_len = int(model_kwargs.get('seq_len'))  # 输入encoder时间步
        self.dcgru_layers = nn.ModuleList(
            [DCGRUCell(self.rnn_units, adj_mx, self.max_diffusion_step, 
self.num_nodes,
                       filter_type=self.filter_type) for _ in 
range(self.num_rnn_layers)])#一个循环层堆叠两个DCGRU单元
```

```
    def forward(self, inputs, hidden_state=None):
        """
        Encoder前向传播

        :参数 input: shape (batch_size, self.num_nodes * self.input_dim)
        :参数 hidden_states: (num_layers, batch_size, self.hidden_state_size)
               optional, zeros if not provided
        :返回值: output: # shape (batch_size, self.hidden_state_size)
                 hidden_states # shape (num_layers, batch_size,
self.hidden_state_size)
                 (lower indices mean lower layers)
        """
        batch_size, _ = inputs.size()
        if hidden_state is None:
            hidden_state = torch.zeros((self.num_rnn_layers, batch_size,
self.hidden_state_size),
                                       device=device)#最开始没有输出值作为特征向量与
输入值融合，故使用0向量
        hidden_states = []
        output = inputs
        #循环迭代两个DCGRU单元
        for layer_num, dcgru_layer in enumerate(self.dcgru_layers):
            next_hidden_state = dcgru_layer(output, hidden_state[layer_num])
            hidden_states.append(next_hidden_state)
            output = next_hidden_state

        return output, torch.stack(hidden_states)  # 时间复杂度 O(num_layers) 将
迭代的特征向量按时间步顺序堆叠
```

(3)DecoderModel模块

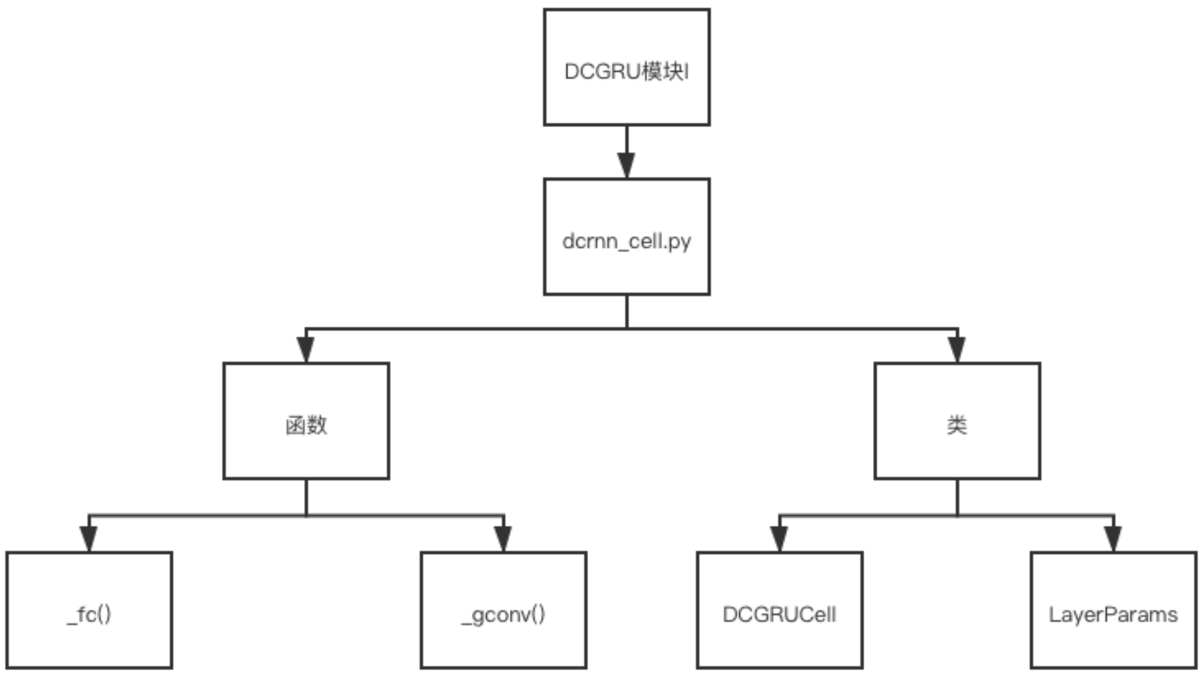
```
class DecoderModel(nn.Module, Seq2SeqAttrs):
    """
        类名: EncoderModel
        主要成员:
        1、输出维度
        2、输出序列长度
        3、全连接层
        3、dcgru层[dcgrucell , dcgrucell ... dcgrucell]
        主要方法:
        1、forward(self, inputs, hidden_state=None)
    """
    def __init__(self, adj_mx, **model_kwargs):
        # super().__init__(is_training, adj_mx, **model_kwargs)
        nn.Module.__init__(self)
        Seq2SeqAttrs.__init__(self, adj_mx, **model_kwargs)
        self.output_dim = int(model_kwargs.get('output_dim', 1))
        self.horizon = int(model_kwargs.get('horizon', 1))  # for the decoder
        self.projection_layer = nn.Linear(self.rnn_units, self.output_dim)
        self.dcgru_layers = nn.ModuleList(
            [DCGRUCell(self.rnn_units, adj_mx, self.max_diffusion_step,
self.num_nodes,
                       filter_type=self.filter_type) for _ in
range(self.num_rnn_layers)])

    def forward(self, inputs, hidden_state=None):
        """
        解码器前向传播。

        :参数 input: shape (batch_size, self.num_nodes * self.output_dim)
        :参数 hidden_states: (num_layers, batch_size, self.hidden_state_size)
               optional, zeros if not provided
        :返回值: output: # shape (batch_size, self.num_nodes * self.output_dim)
                hidden_states # shape (num_layers, batch_size,
self.hidden_state_size)
                 (lower indices mean lower layers)
        """
        hidden_states = []
        output = inputs
        #循环迭代两个DCGRU单元
        for layer_num, dcgru_layer in enumerate(self.dcgru_layers):
            next_hidden_state = dcgru_layer(output, hidden_state[layer_num])
            hidden_states.append(next_hidden_state)
            output = next_hidden_state
        #为保证output和标签维度相同，使用一层全连接网络作为投影层，投影至和标签维度相同
        projected = self.projection_layer(output.view(-1, self.rnn_units))
        output = projected.view(-1, self.num_nodes * self.output_dim)

        return output, torch.stack(hidden_states)
```

(4)DCRNNModel模块

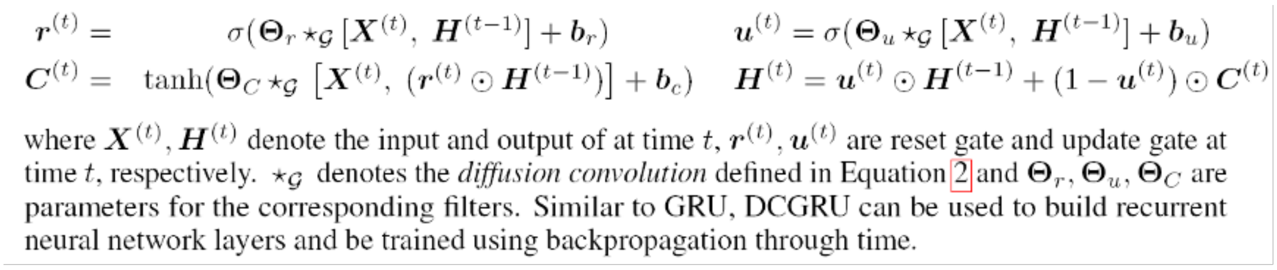
```
class DCRNNModel(nn.Module, Seq2SeqAttrs):
    """
        类名: DCRNNModel
        主要成员:
        1、编码器
        2、解码器
        主要方法:
        1、encoder(self, inputs)
        2、decoder(self, encoder_hidden_state, labels=None, batches_seen=None)
        3、forward(self, inputs, labels=None, batches_seen=None)
    """
    def __init__(self, adj_mx, logger, **model_kwargs):
        super().__init__()
        Seq2SeqAttrs.__init__(self, adj_mx, **model_kwargs)
        self.encoder_model = EncoderModel(adj_mx, **model_kwargs)
        self.decoder_model = DecoderModel(adj_mx, **model_kwargs)
        self.cl_decay_steps = int(model_kwargs.get('cl_decay_steps', 1000))
        self.use_curriculum_learning =
bool(model_kwargs.get('use_curriculum_learning', False))
        self._logger = logger


    def encoder(self, inputs):
        """
        向编码器循环输入t个时间步的特征向量
        :参数 input: shape (seq_len, batch_size, num_sensor * input_dim)
        :返回: encoder_hidden_state: (num_layers, batch_size,
self.hidden_state_size)
        """
        encoder_hidden_state = None
        for t in range(self.encoder_model.seq_len):#循环迭代seq_len步
            _, encoder_hidden_state = self.encoder_model(inputs[t],
encoder_hidden_state)

        return encoder_hidden_state

    def decoder(self, encoder_hidden_state, labels=None, batches_seen=None):
        """
        向解码器循环输入t个时间步的特征向量
        :参数 encoder_hidden_state: (num_layers, batch_size,
self.hidden_state_size)
        :参数 labels: (self.horizon, batch_size, self.num_nodes *
self.output_dim)
        :参数 batches_seen: global step [optional, not exist for inference]
        :返回值: output: (self.horizon, batch_size, self.num_nodes *
self.output_dim)
        """
        batch_size = encoder_hidden_state.size(1)
        go_symbol = torch.zeros((batch_size, self.num_nodes *
self.decoder_model.output_dim),
                                device=device)
        decoder_hidden_state = encoder_hidden_state
        decoder_input = go_symbol#在第一个时间步没有任何预测结果，使用纯0向量作为预测的
开始

        outputs = []

        for t in range(self.decoder_model.horizon):#循环迭代horizon步
            decoder_output, decoder_hidden_state =
self.decoder_model(decoder_input,

 decoder_hidden_state)
            decoder_input = decoder_output#上一步输出作为下一步输入
            outputs.append(decoder_output)
            if self.training and self.use_curriculum_learning:#课程学习
https://zhuanlan.zhihu.com/p/81455004
                c = np.random.uniform(0, 1)
                if c < self._compute_sampling_threshold(batches_seen):#按照一定的
概率决定是否选择标签
                    decoder_input = labels[t]
        outputs = torch.stack(outputs)
        return outputs

    def forward(self, inputs, labels=None, batches_seen=None):
        """
        Seq2Seq模型前向传播
        :参数 inputs: shape (seq_len, batch_size, num_sensor * input_dim)
        :参数 labels: shape (horizon, batch_size, num_sensor * output)
        :参数 batches_seen: batches seen till now
        :返回值: output: (self.horizon, batch_size, self.num_nodes *
self.output_dim)
        """
        encoder_hidden_state = self.encoder(inputs)
        self._logger.debug("Encoder complete, starting decoder")
        outputs = self.decoder(encoder_hidden_state, labels,
batches_seen=batches_seen)
        self._logger.debug("Decoder complete")
        if batches_seen == 0:
            self._logger.info(
                "Total trainable parameters {}".format(count_parameters(self))
            )
        return outputs
```

dcrnn_cell.py

dcrnn_cell实现了论文中的DCGRU，它将卷积融合到了GRU里面。具体结构如下:



它实现了论文中的如下部分：



$$
\begin{aligned}
\boldsymbol{r}^{(t)} &= \sigma(\Theta_r \star_{\mathcal{G}} [\boldsymbol{X}^{(t)},\ \boldsymbol{H}^{(t-1)}] + \boldsymbol{b}_r) & \boldsymbol{u}^{(t)} &= \sigma(\Theta_u \star_{\mathcal{G}} [\boldsymbol{X}^{(t)},\ \boldsymbol{H}^{(t-1)}] + \boldsymbol{b}_u) \\
\boldsymbol{C}^{(t)} &= \tanh(\Theta_C \star_{\mathcal{G}} \left[\boldsymbol{X}^{(t)},\ (\boldsymbol{r}^{(t)} \odot \boldsymbol{H}^{(t-1)})\right] + \boldsymbol{b}_c) & \boldsymbol{H}^{(t)} &= \boldsymbol{u}^{(t)} \odot \boldsymbol{H}^{(t-1)} + (1 - \boldsymbol{u}^{(t)}) \odot \boldsymbol{C}^{(t)}
\end{aligned}
$$

where $\boldsymbol{X}^{(t)}, \boldsymbol{H}^{(t)}$ denote the input and output of at time t, $\boldsymbol{r}^{(t)}, \boldsymbol{u}^{(t)}$ are reset gate and update gate at time t, respectively. $\star_{\mathcal{G}}$ denotes the *diffusion convolution* defined in Equation 2 and $\Theta_r, \Theta_u, \Theta_C$ are parameters for the corresponding filters. Similar to GRU, DCGRU can be used to build recurrent neural network layers and be trained using backpropagation through time.

(1)实现初始化

```
def __init__(self, num_units, adj_mx,
max_diffusion_step,num_nodes,num_proj=None,activation=tf.nn.tanh,reuse=None,
filter_type="laplacian",use_gc_for_ru=True):
```

该函数实现了GRU模块成员变量的初始化，包括units数量，路网图距离权重邻接矩阵，扩散步数，节点数量。输入参数num_units表示每个cell中units的数量，参与后面state_size(self,)，output_size(self)的计算；输入参数adj_mx对应速度检测站之间的归一化后的距离权重邻接矩阵；输入参数max_diffusion_step表示允许的最大随机游走的步数；输入参数num_nodes表示速度检测站节点的数量；输入参数num_proj 代表预测结果的数量（如num_proj=1表示预测1个结果，num_proj=2表示预测2个结果）。如果num_proj不为空（None），即到了最后一层cell_with_projection，GRU模块在计算后会进行预测。

(2)实现基于图卷积的随机游走函数

```
def _gconv(self, inputs, state, output_size, bias_start=0.0):
  # 把输入和特征向量转换成 (batch_size, num_nodes, input_dim/state_dim)
  batch_size = inputs.shape[0]
  inputs = torch.reshape(inputs, (batch_size, self._num_nodes, -1))
  state = torch.reshape(state, (batch_size, self._num_nodes, -1))
  inputs_and_state = torch.cat([inputs, state], dim=2)
  input_size = inputs_and_state.size(2)

  x = inputs_and_state
  x0 = x.permute(1, 2, 0)  # (num_nodes, total_arg_size, batch_size)
  x0 = torch.reshape(x0, shape=[self._num_nodes, input_size * batch_size])
  x = torch.unsqueeze(x0, 0)

  if self._max_diffusion_step == 0:
    pass
  else:
    for support in self._supports:
      x1 = torch.sparse.mm(support, x0)
      x = self._concat(x, x1)

      for k in range(2, self._max_diffusion_step + 1):
        x2 = 2 * torch.sparse.mm(support, x1) - x0
        x = self._concat(x, x2)
        x1, x0 = x2, x1

        num_matrices = len(self._supports) * self._max_diffusion_step + 1
        x = torch.reshape(x, shape=[num_matrices, self._num_nodes, input_size,
batch_size])
        x = x.permute(3, 1, 2, 0)  # (batch_size, num_nodes, input_size, order)
        x = torch.reshape(x, shape=[batch_size * self._num_nodes, input_size *
num_matrices])

        weights = self._gconv_params.get_weights((input_size * num_matrices,
output_size))
        x = torch.matmul(x, weights)  # (batch_size * self._num_nodes,
output_size)

        biases = self._gconv_params.get_biases(output_size, bias_start)
        x += biases
        # 把结果转换成 2D: (batch_size, num_node, state_dim) -> (batch_size,
num_node * state_dim)
        return torch.reshape(x, [batch_size, self._num_nodes * output_size])
```

这里实现了论文中的扩散卷积方法。首先将input和state拼接为input_state，即论文中的，将input_state和random_walk_mx进行max_diffusion_step次图卷积，结果是max_diffusion_step步随机游走的结果，然后用将乘以一个weights矩阵把维度变换为output_size。首先对input和state进行计算，对计算结果进行nn.sigmoid()激活。这里的和为GRU模型中的reset_gate和update_gate。然后计算GRU核，如果存在激活函数，则。计算新的输出和状态 ，除非到了最后一层cell_with_projection，否则output总是和new_state是一样的。判断是否需要输出预测值，如果不需要，则直接返回output，new_state。如果需要进行预测，将output乘以一个w矩阵将维度变换为output_size，最后返回output，new_state。