

DAML code

```
import torch
import random
import numpy as np
from config import global_config as cfg
from reader import CamRest676Reader, get_glove_matrix
from reader import KvretReader
from tsd_net import TSD, cuda_, nan
from torch import nn
from torch import optim
from torch.optim import Adam
from torch.autograd import Variable
from reader import pad_sequences
import argparse, time
import copy
import pdb
from metric import CamRestEvaluator, KvretEvaluator
import logging

class Model:
    def __init__(self, dataset):
        reader_dict = {
            'camrest': CamRest676Reader,
            'kvret': KvretReader,
        }
        model_dict = {
            'TSD': TSD
        }
        evaluator_dict = {
            'camrest': CamRestEvaluator,
            'kvret': KvretEvaluator,
        }
        self.reader = reader_dict[dataset]()
        self.m = model_dict[cfg.m](embed_size=cfg.embedding_size,
            hidden_size=cfg.hidden_size,
            vocab_size=cfg.vocab_size,
            layer_num=cfg.layer_num,
            dropout_rate=cfg.dropout_rate,
```

```

z_length=cfg.z_length,
max_ts=cfg.max_ts,
beam_search=cfg.beam_search,
beam_size=cfg.beam_size,
eos_token_idx=self.reader.vocab.encode('EOS_M'),
vocab=self.reader.vocab,
teacher_force=cfg.teacher_force,
degree_size=cfg.degree_size)          self.EV = evaluator_dict[dataset] # evaluator
class
    if cfg.cuda: self.m = self.m.cuda()
    self.base_epoch = -1
    self.pr_loss = nn.NLLLoss(ignore_index=0)          self.dec_loss =
nn.NLLLoss(ignore_index=0)
    # # parameters for maml
    # self.train_lr = cfg.lr
    self.meta_lr = cfg.lr #meta_lr
    # self.nway = nway
    # self.kshot = kshot
    # self.kquery = kquery
    # self.meta_batchsz = meta_batchsz          # self.meta_optim =
optim.Adam(self.m.parameters(), lr = self.meta_lr)          # self.meta_optim = Adam(lr =
self.meta_lr, params=filter(lambda x: x.requires_grad, self.m.parameters()),weight_decay=1e-
5)
    def _convert_batch(self, py_batch, prev_z_py=None):          u_input_py =
py_batch['user']
        u_len_py = py_batch['u_len']
        kw_ret = {}          if cfg.prev_z_method == 'concat' and prev_z_py is not
None:
            for i in range(len(u_input_py)):          eob =
self.reader.vocab.encode('EOS_Z2')          if eob in prev_z_py[i] and
prev_z_py[i].index(eob) != len(prev_z_py[i]) - 1:          idx =
prev_z_py[i].index(eob)          u_input_py[i] = prev_z_py[i][:idx + 1] +
u_input_py[i]
            else:          u_input_py[i] = prev_z_py[i] +
u_input_py[i]
            u_len_py[i] = len(u_input_py[i])          for j, word in
enumerate(prev_z_py[i]):
                if word >= cfg.vocab_size:          prev_z_py[i][j] = 2
#unk
            elif cfg.prev_z_method == 'separate' and prev_z_py is not None:          for i in

```

```

range(len(prev_z_py)):
    eob = self.reader.vocab.encode('EOS_Z2')
    if eob in
prev_z_py[i] and prev_z_py[i].index(eob) != len(prev_z_py[i]) - 1:
        idx =
prev_z_py[i].index(eob)
        prev_z_py[i] = prev_z_py[i][:idx +
1]

    for j, word in enumerate(prev_z_py[i]):
        if word >=
cfg.vocab_size:
        prev_z_py[i][j] = 2 #unk
        prev_z_input_np =
pad_sequences(prev_z_py, cfg.max_ts, padding='post', truncating='pre').transpose((1,
0))

        prev_z_len = np.array([len(_) for _ in prev_z_py])
        prev_z_input =
cuda_(Variable(torch.from_numpy(prev_z_input_np).long()))
        kw_ret['prev_z_len'] =
prev_z_len

        kw_ret['prev_z_input'] = prev_z_input
        kw_ret['prev_z_input_np'] =
prev_z_input_np

        degree_input_np = np.array(py_batch['degree'])
        u_input_np =
pad_sequences(u_input_py, cfg.max_ts, padding='post', truncating='pre').transpose((1,
0))
        z_input_np = pad_sequences(py_batch['bspan'], padding='post').transpose((1,
0))
        m_input_np = pad_sequences(py_batch['response'], cfg.max_ts, padding='post',
truncating='post').transpose(
            (1, 0))

        u_len = np.array(u_len_py)
        m_len = np.array(py_batch['m_len'])
        degree_input = cuda_(Variable(torch.from_numpy(degree_input_np).float()))
u_input = cuda_(Variable(torch.from_numpy(u_input_np).long()))
        z_input =
cuda_(Variable(torch.from_numpy(z_input_np).long()))
        m_input =
cuda_(Variable(torch.from_numpy(m_input_np).long()))

        kw_ret['z_input_np'] = z_input_np
        return u_input, u_input_np, z_input, m_input, m_input_np, u_len, m_len,
\
        degree_input, kw_ret

def train_maml(self):
    lr = cfg.lr
    prev_min_loss, early_stop_count = 1 << 30,
cfg.early_stop_count

    train_time = 0
    for epoch in range(cfg.epoch_num):
        # for epoch in range(1):
            sw = time.time()
            if epoch <= self.base_epoch:
                continue

```

```

        self.training_adjust(epoch)
        self.m.self_adjust(epoch)
        sup_loss = 0
        sup_cnt = 0
        turn_batches_domain =
self.reader.mini_batch_iterator_maml_supervised('train')
        optim = Adam(lr=lr, params=filter(lambda x: x.requires_grad,
self.m.parameters()),weight_decay=1e-5)          meta_optim = Adam(lr = self.meta_lr,
params=filter(lambda x: x.requires_grad, self.m.parameters()),weight_decay=1e-5)
        init_state = copy.deepcopy(self.m.state_dict())
        # for iter_num, dial_batch in enumerate(data_iterator[min_idx]):          for
turn_batch_domain in turn_batches_domain:
            turn_states = {}
            prev_z = None
            loss_tasks = []
            for k in range(len(cfg.data)):          # for k-th
task:
                turn_batch = turn_batch_domain[k]
                self.m.load_state_dict(init_state)
optim.zero_grad()
                u_input, u_input_np, z_input, m_input, m_input_np, u_len,
\
                m_len, degree_input, kw_ret \          =
self._convert_batch(turn_batch, prev_z)
                init_state = copy.deepcopy(self.m.state_dict())
                for tmp_grad in range(int(cfg.maml_step)):          # #
update parameters for each task          loss, pr_loss, m_loss, turn_states =
self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
turn_states=turn_states,
u_len=u_len,
m_len=m_len,
mode='train',
**kw_ret)
                loss.backward()          #
loss.backward(retain_graph=turn_num != len(dial_batch) - 1)          grad =

```

```

torch.nn.utils.clip_grad_norm(self.m.parameters(), 5.0)
optim.step()

        ## resample                                # input should be different from
above

        ## loss for the meta-update                                loss, pr_loss, m_loss,

turn_states =
self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
turn_states=turn_states,
u_len=u_len,
m_len=m_len,
mode='train',
**kw_ret)

        loss_tasks.append(loss)
        prev_z = turn_batch['bspan']
        self.m.load_state_dict(init_state)
meta_optim.zero_grad()
        loss_meta = torch.stack(loss_tasks).sum(0) / len(cfg.data)
        loss_meta.backward()                                #
loss_meta.backward(retain_graph=turn_num != len(dial_batch) - 1)                                grad =
torch.nn.utils.clip_grad_norm(self.m.parameters(), 5.0)
meta_optim.step()
        init_state = copy.deepcopy(self.m.state_dict())
        sup_loss += loss_meta.data.cpu().numpy()[0]                                sup_cnt +=
1
        epoch_sup_loss = sup_loss / (sup_cnt + 1e-8)                                train_time += time.time()
- SW
        logging.info('Traning time: {}'.format(train_time))                                logging.info('avg
training loss in epoch %d sup:%f' % (epoch, epoch_sup_loss))
        valid_sup_loss, valid_unsup_loss = self.validate_maml()
logging.info('validation loss in epoch %d sup:%f unsup:%f' % (epoch, valid_sup_loss,
valid_unsup_loss))                                logging.info('time for epoch %d: %f' % (epoch, time.time()-
sw))

        valid_loss = valid_sup_loss + valid_unsup_loss
        if valid_loss <= prev_min_loss:                                # self.save_model(epoch, path =
'./models/camrest_maml.pkl')

```

```

        self.save_model(epoch)                prev_min_loss =
valid_loss

        early_stop_count = cfg.early_stop_count
    else:
        early_stop_count -= 1
        lr *= cfg.lr_decay
        self.meta_lr *= cfg.lr_decay          if not
early_stop_count:
            break                logging.info('early stop countdown %d, learning rate
%f' % (early_stop_count, lr))
    def validate_maml(self, data='dev'):
        self.m.eval()                data_iterator =
self.reader.mini_batch_iterator_maml_supervised(data)                sup_loss, unsup_loss = 0,
0
        sup_cnt, unsup_cnt = 0, 0
        for dial_batch in data_iterator:
            turn_states = {}                for turn_num, turn_batch in
enumerate(dial_batch):                u_input, u_input_np, z_input, m_input, m_input_np,
u_len, \
                m_len, degree_input, kw_ret \                =
self._convert_batch(turn_batch)
                loss, pr_loss, m_loss, turn_states =
self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
turn_states=turn_states,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
u_len=u_len,
m_len=m_len,
mode='train',
**kw_ret)
                sup_loss += loss.data[0]
                sup_cnt += 1

        sup_loss /= (sup_cnt + 1e-8)
        unsup_loss /= (unsup_cnt + 1e-8)
        self.m.train()
        return sup_loss, unsup_loss

```

```

def eval_maml(self, data='test'):
    self.m.eval()

    self.reader.result_file = None          data_iterator =
self.reader.mini_batch_iterator_maml_supervised(data)          mode = 'test' if not
cfg.pretrain else 'pretrain_test'          for batch_num, dial_batch in
enumerate(data_iterator):
    turn_states = {}
    prev_z = None          for turn_num, turn_batch in
enumerate(dial_batch):          u_input, u_input_np, z_input, m_input, m_input_np,
u_len, \
                m_len, degree_input, kw_ret \
                =
self._convert_batch(turn_batch, prev_z)          m_idx, z_idx, turn_states =
self.m(mode=mode, u_input=u_input, u_len=u_len,
z_input=z_input,
m_input=m_input,          degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,          m_len=m_len,
turn_states=turn_states,**kw_ret)          self.reader.wrap_result(turn_batch, m_idx,
z_idx, prev_z=prev_z)
        prev_z = z_idx
        ev = self.EV(result_path=cfg.result_path)
        res = ev.run_metrics_maml()
        self.m.train()
        return res

def train(self):
    lr = cfg.lr          prev_min_loss, early_stop_count = 1 << 30,
cfg.early_stop_count
    train_time = 0
    for epoch in range(cfg.epoch_num):
        sw = time.time()
        # if epoch <= self.base_epoch:
        #     continue
        self.training_adjust(epoch)
        self.m.self_adjust(epoch)
        sup_loss = 0
        sup_cnt = 0          data_iterator =
self.reader.mini_batch_iterator('train')          optim = Adam(lr=lr, params=filter(lambda
x: x.requires_grad, self.m.parameters()),weight_decay=1e-5)          for iter_num,
dial_batch in enumerate(data_iterator):
            turn_states = {}

```

```

        prev_z = None
        for turn_num, turn_batch in
enumerate(dial_batch):
            if cfg.truncated:
                logging.debug('iter %d turn %d'
% (iter_num, turn_num))
                optim.zero_grad()
                u_input, u_input_np, z_input,
m_input, m_input_np, u_len, \
                m_len, degree_input, kw_ret
\
                = self._convert_batch(turn_batch, prev_z)
                loss, pr_loss, m_loss, turn_states =
self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
turn_states=turn_states,
u_len=u_len,
m_len=m_len,
mode='train',
**kw_ret)

                loss.backward(retain_graph=turn_num != len(dial_batch) -
1)
                grad = torch.nn.utils.clip_grad_norm(self.m.parameters(),
5.0)

                optim.step()
                sup_loss +=
loss.data.cpu().numpy()[0]
                sup_cnt += 1
                prev_z = turn_batch['bspan']
                epoch_sup_loss = sup_loss / (sup_cnt + 1e-8)
                train_time += time.time()
- sw

                logging.info('Traning time: {}'.format(train_time))
                logging.info('avg
training loss in epoch %d sup:%f' % (epoch, epoch_sup_loss))
                # print('Traning
time: {}'.format(train_time))
                print('avg training loss in epoch %d sup:%f' %
(epoch, epoch_sup_loss))
                valid_sup_loss, valid_unsup_loss =
self.validate()
                logging.info('validation loss in epoch %d sup:%f unsup:%f' %
(epoch, valid_sup_loss, valid_unsup_loss))
                logging.info('time for epoch %d: %f' %
(epoch, time.time()-sw))
                print('validation loss in epoch %d sup:%f unsup:%f' %
(epoch, valid_sup_loss, valid_unsup_loss))
                # print('time for epoch %d: %f' %
(epoch, time.time()-sw))

                valid_loss = valid_sup_loss + valid_unsup_loss
                if valid_loss <= prev_min_loss:

```

```

self.save_model(epoch)

        prev_min_loss = valid_loss                early_stop_count =
cfg.early_stop_count
    else:
        early_stop_count -= 1
        lr *= cfg.lr_decay
        if not early_stop_count:
            break                logging.info('early stop countdown %d, learning rate
%f' % (early_stop_count, lr))                print('early stop countdown %d, learning rate
%f' % (early_stop_count, lr))
    def eval(self, data='test'):
        self.m.eval()
        self.reader.result_file = None                data_iterator =
self.reader.mini_batch_iterator(data)                mode = 'test' if not cfg.pretrain else
'pretrain_test'
        for batch_num, dial_batch in enumerate(data_iterator):                turn_states =
{}
            prev_z = None                for turn_num, turn_batch in
enumerate(dial_batch):                u_input, u_input_np, z_input, m_input, m_input_np,
u_len, \
                m_len, degree_input, kw_ret \                =
self._convert_batch(turn_batch, prev_z)                m_idx, z_idx, turn_states =
self.m(mode=mode, u_input=u_input, u_len=u_len,
z_input=z_input,
m_input=m_input,                degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,                m_len=m_len,
turn_states=turn_states,**kw_ret)                self.reader.wrap_result(turn_batch, m_idx,
z_idx, prev_z=prev_z)
            prev_z = z_idx
            ev = self.EV(result_path=cfg.result_path)
            res = ev.run_metrics()
            self.m.train()
            return res
    def validate(self, data='dev'):
        self.m.eval()
        data_iterator = self.reader.mini_batch_iterator(data)                sup_loss, unsup_loss =
0, 0
        sup_cnt, unsup_cnt = 0, 0
        for dial_batch in data_iterator:

```

```

        turn_states = {}
        for turn_num, turn_batch in
enumerate(dial_batch):
        u_input, u_input_np, z_input, m_input, m_input_np,
u_len, \
        m_len, degree_input, kw_ret \
        =
self._convert_batch(turn_batch)
        loss, pr_loss, m_loss, turn_states =
self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
turn_states=turn_states,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
u_len=u_len,
m_len=m_len,
mode='train',
**kw_ret)

        sup_loss += loss.data[0]
        sup_cnt += 1
        # logging.debug(
        #         'loss:{} pr_loss:{}
m_loss:{}'.format(loss.data[0], pr_loss.data[0], m_loss.data[0]))
        sup_loss /= (sup_cnt + 1e-8)
        unsup_loss /= (unsup_cnt + 1e-8)
        self.m.train()
        print('result preview...')
        # self.eval()
        return sup_loss, unsup_loss
def reinforce_tune(self):
    lr = cfg.lr
    prev_min_loss, early_stop_count = 1 << 30,
cfg.early_stop_count
    for epoch in range(self.base_epoch + cfg.rl_epoch_num +
1):
        mode = 'rl'
        if epoch <= self.base_epoch:
            continue
        epoch_loss, cnt = 0,0
        data_iterator =
self.reader.mini_batch_iterator('train')
        optim = Adam(lr=lr, params=filter(lambda
x: x.requires_grad, self.m.parameters()), weight_decay=1e-5)
        for iter_num,
dial_batch in enumerate(data_iterator):
            turn_states = {}
            prev_z = None
            for turn_num, turn_batch in

```

```

enumerate(dial_batch):
    optim.zero_grad()
    u_input, u_input_np, z_input,
m_input, m_input_np, u_len, \
    m_len, degree_input, kw_ret
\
    = self._convert_batch(turn_batch, prev_z)
    loss_rl
= self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
turn_states=turn_states,
u_len=u_len,
m_len=m_len,
mode=mode,

    **kw_ret)
    if loss_rl is not None:
        loss =
loss_rl
        loss.backward()
        grad =
torch.nn.utils.clip_grad_norm(self.m.parameters(), 2.0)
optim.step()
epoch_loss +=
loss.data.cpu().numpy()[0]
    cnt += 1
    logging.debug('{} loss {}'.
grad:{}'.format(mode, loss.data[0], grad))
    prev_z = turn_batch['bspan']
    epoch_sup_loss = epoch_loss / (cnt + 1e-8)
    logging.info('avg training
loss in epoch %d sup:%f' % (epoch, epoch_sup_loss))
    valid_sup_loss, valid_unsup_loss = self.validate()
    logging.info('validation loss in epoch %d sup:%f unsup:%f' % (epoch, valid_sup_loss,
valid_unsup_loss))
    valid_loss = valid_sup_loss + valid_unsup_loss
    self.save_model(epoch)
    if valid_loss <= prev_min_loss:
#self.save_model(epoch)
        prev_min_loss = valid_loss
    else:
        early_stop_count -= 1
        lr *= cfg.lr_decay
        if not early_stop_count:
            break
            logging.info('early stop countdown %d, learning rate

```

```

%f' % (early_stop_count, lr))
    def reinforce_tune_maml(self):
        lr = cfg.lr          prev_min_loss, early_stop_count = 1 << 30,
cfg.early_stop_count      for epoch in range(self.base_epoch + cfg.rl_epoch_num +
1):
            mode = 'rl'
            if epoch <= self.base_epoch:
                continue
            epoch_loss, cnt = 0,0          data_iterator =
self.reader.mini_batch_iterator('train')          optim = Adam(lr=lr, params=filter(lambda
x: x.requires_grad, self.m.parameters()), weight_decay=1e-5)          for iter_num,
dial_batch in enumerate(data_iterator):
                turn_states = {}
                prev_z = None          for turn_num, turn_batch in
enumerate(dial_batch):
                    optim.zero_grad()          u_input, u_input_np, z_input,
m_input, m_input_np, u_len, \          m_len, degree_input, kw_ret
\
                    = self._convert_batch(turn_batch, prev_z)
                    init_state = copy.deepcopy(self.m.state_dict())
loss_tasks = []
                    for k in range(len(cfg.data)):
                        self.m.load_state_dict(init_state)
optim.zero_grad()
                        loss_rl =
self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
turn_states=turn_states,
u_len=u_len,
m_len=m_len,
mode=mode,
                        **kw_ret)
                        if loss_rl is not None:          loss =
loss_rl
                        loss.backward()          grad =
torch.nn.utils.clip_grad_norm(self.m.parameters(), 2.0)

```

```

optim.step()

        loss_rl =

self.m(u_input=u_input,
z_input=z_input,
m_input=m_input,
degree_input=degree_input,
u_input_np=u_input_np,
m_input_np=m_input_np,
turn_states=turn_states,
u_len=u_len,
m_len=m_len,
mode=mode,

        **kw_ret)

        if loss_rl is not None:
loss_tasks.append(loss_rl)

        if len(loss_tasks) != 0:
self.m.load_state_dict(init_state)
self.meta_optim.zero_grad()                loss_meta = torch.stack(loss_tasks).sum(0)
/ len(cfg.data)

        loss_meta.backward()

self.meta_optim.step()

        init_state =

copy.deepcopy(self.m.state_dict())

        epoch_loss += loss_meta.data.cpu().numpy()[0]

cnt += 1                logging.debug('{} loss {},
grad:{}'.format(mode,loss_meta.data[0],grad))

        prev_z = turn_batch['bspan']

        epoch_sup_loss = epoch_loss / (cnt + 1e-8)                logging.info('avg training
loss in epoch %d sup:%f' % (epoch, epoch_sup_loss))

        valid_sup_loss, valid_unsup_loss = self.validate()

logging.info('validation loss in epoch %d sup:%f unsup:%f' % (epoch, valid_sup_loss,
valid_unsup_loss))

        valid_loss = valid_sup_loss + valid_unsup_loss

        # self.save_model(epoch, path = './models/camrest_maml.pkl')

self.save_model(epoch)

        if valid_loss <= prev_min_loss:
#self.save_model(epoch)

        prev_min_loss = valid_loss

        else:

        early_stop_count -= 1

```

```

        lr *= cfg.lr_decay
        if not early_stop_count:
            break
            logging.info('early stop countdown %d, learning rate
%f' % (early_stop_count, lr))
    def save_model(self, epoch, path=None):
        if not path:
            path = cfg.model_path
            all_state = {'lstd':
self.m.state_dict(),
                        'config': cfg.__dict__,
                        'epoch': epoch}
            torch.save(all_state, path)
    def load_model(self, path=None):
        if not path:
            path = cfg.model_path
            all_state = torch.load(path)
            self.m.load_state_dict(all_state['lstd'])
            self.base_epoch =
all_state.get('epoch', 0)
    def training_adjust(self, epoch):
        return
    def freeze_module(self, module):
        for param in module.parameters():
            param.requires_grad = False
    def unfreeze_module(self, module):
        for param in module.parameters():
            param.requires_grad = True
    def load_glove_embedding(self, freeze=False):
        initial_arr =
self.m.u_encoder.embedding.weight.data.cpu().numpy()
        embedding_arr =
torch.from_numpy(get_glove_matrix(self.reader.vocab, initial_arr))
        self.m.u_encoder.embedding.weight.data.copy_(embedding_arr)
        self.m.z_decoder.emb.weight.data.copy_(embedding_arr)
        self.m.m_decoder.emb.weight.data.copy_(embedding_arr)
    def count_params(self):
        module_parameters = filter(lambda p: p.requires_grad, self.m.parameters())
        param_cnt = sum([np.prod(p.size()) for p in module_parameters])
        print('total trainable params: %d' % param_cnt)
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-mode')
    parser.add_argument('-model')
    parser.add_argument('-cfg', nargs='*')

```

```

args = parser.parse_args()
cfg.init_handler(args.model)
if args.cfg:
    for pair in args.cfg:
        k, v = tuple(pair.split('='))
        dtype = type(getattr(cfg,
k))

        if dtype == type(None):
            raise ValueError()
        if dtype is bool:
            v = False if v == 'False' else True
        else:
            v = dtype(v)
        setattr(cfg, k, v)
if args.cfg:
    cfg.split = tuple([int(i) for i in cfg.split])
    cfg.mode = args.mode        if type(cfg.data) is list and 'maml' not in
cfg.mode:
    cfg.data = "".join(cfg.data)        if type(cfg.db) is list and 'maml' not in
cfg.mode:
    cfg.db = "".join(cfg.db)        if type(cfg.entity) is list and 'maml' not in
cfg.mode:
    cfg.entity = "".join(cfg.entity)
logging.debug(str(cfg))
if 'train' not in args.mode:
    print(str(cfg))
if cfg.cuda:
    torch.cuda.set_device(cfg.cuda_device)        logging.debug('Device:
{}'.format(torch.cuda.current_device()))
    cfg.mode = args.mode
    torch.manual_seed(cfg.seed)
    torch.cuda.manual_seed(cfg.seed)
    random.seed(cfg.seed)
    np.random.seed(cfg.seed)
    m = Model(args.model.split('-')[1])
    m.count_params()
    if args.mode == 'train':
        m.load_glove_embedding()
        m.train()
    elif args.mode == 'adjust':
        m.load_model()

```

```
        m.train()
elif args.mode == 'test':
    m.load_model()
    m.eval()
elif args.mode == 'rl':
    m.load_model()
    m.reinforce_tune()
elif args.mode == 'train_maml':
    m.load_glove_embedding()
    m.train_maml()
elif args.mode == 'adjust_maml':
    m.load_model()
    m.adjust_maml()
elif args.mode == 'test_maml':
    m.load_model()
    m.eval_maml()
elif args.mode == 'rl_maml':
    m.load_model()
    m.reinforce_tune_maml()
if __name__ == '__main__':
    main()
```

Revision #2

Created Wed, Feb 5, 2020 6:21 PM by [kenneth](#)

Updated Tue, Feb 11, 2020 5:28 PM by [kenneth](#)