

Metalearning

- Educational Resources about Metalearning
- Integration of DAML
 - DAML code
 - DAML walkthrough
 - Optimizer filter notes
 - Gameplan
 - Status tracker
- Integration of multi-task code in allennlp
 - Info
- Testing
 - Tests I need
- Repo cleanup notes
- Design notes 5-27

Educational Resources about Metalearning

- [Stanford metalearning course](#)
- [Facebook package with metalearning utilities in pytorch](#)
- [Chelsea Finn's metalearning tutorial](#)

Integration of DAML

This describes efforts to work on integration of DAML into the existing content scoring pipeline

<https://github.com/qbetterk/DAML>

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 {}'.format(
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 {}'.format(mode, loss_meta.data[0], grad))
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()

```

DAML walkthrough

The main point of interest is `train_maml` in `model.py`.

`prev_min_loss` gets set to a high value on [line 120](#) using bit shifting.

Then we go through epochs. This corresponds to `train()` in `metatrainer.py`.

`Self.training_adjust()` on [line 127](#) just returns? Weird `self.m.self_adjust()` on [line 128](#) is referring to the TSD model in `tsd_net.py`.

This also [just passes](#)? WTH?

[line 132](#) handles the data reading and dumps the result in `turn_batches_domain`.

Lines 134 and 135 declare two optimizers, `optim`, and `meta_optim`. They use a filter expression
`meta_optim = Adam(lr = self.meta_lr, params=filter(lambda x: x.requires_grad, self.m.parameters()),`
.

This obtains only the parameters that need gradient updates. I'm not currently doing this step, so this is potentially where my stuff is going awry. I am doing a similar filter [on line 241](#). However, this is filtering the parameters before the model copy happens, not when the optimizer is created.

They then copy the model's state dict using `copy.deepcopy`. I think this is the python internal `deepcopy` mechanism instead of the pytorch specific one.

For batch in data, for each task [get the data](#).

Load the state dict into the model, zero out the graidents on `optim` (I think this is the inner optimizer)

They call `self._convert_batch()` on [line 154](#). Need to look and see what that does but my initial reaction is that it's unimportant and specific to their task setup.

For each step in `cfg.maml_steps` run the input through the model and get the loss, `pr_loss`, `m_loss` and `turn_states`

call `loss.backward()` to get gradients. grab the model parameters and gradient clip them Then step the inner optimizer (`optim`)

Once out of that loop, run the model again. It says it's using fresh data in the comment on [line 178](#) but as far as I can tell, it is not.

record the losses in an array.

Then outside that loop, identify

Optimizer filter notes

The DAML implementation uses this filter when constructing the optimizer

```
Adam(lr = self.meta_lr, params=filter(lambda x: x.requires_grad, self.m.parameters()),weight_decay=
```

.

I'm unsure how important this is, but for now, I've hard coded this into the `from_params` for the optimizer and the constructor for the metatrainer (for the `meta_optimizer`).

I need to get a metatrainer working and then figure out how to make it more flexible in the allennlp framework.

Gameplan

Done Switch optimizer over to the ifiltered optimizer.

todo determine if `self.model` needs to be modified in the batch_update section

todo make the maml optimizer stepping logic match the logic in daml

Status tracker

2-21-2020:

Finished a bunch of updates to the outer loop of maml to make it better reflect the daml implementation. However, allennlp is telling me

```
from src.data import *
File "./src/data/__init__.py", line 1, in <module>      from
src.data.tokenizers.word_splitter_additions import \ File
"./src/data/tokenizers/__init__.py", line 5, in <module>      from
src.data.tokenizers.word_splitter_additions import \ File
"./src/data/tokenizers/word_splitter_additions.py", line 11, in <module>      from
allennlp.data.tokenizers.word_splitter import _remove_spaces, WordSplitter
ModuleNotFoundError: No module named 'allennlp.data.tokenizers.word_splitter'
```

whenn I try to run it in `/home/kenneth/Projects/nadapt_12_11/neural-adapt-dev`.

Was there a different location I wanted to use?

There's a different branch `metalearn` that needs to be checked out.

2-25-2020:

Every time the maml learner goes through the validation part, the qwk drops to 0. Sometimes it recovers during the epoch but sometimes it does not.

Integration of multi-task code in allennlp

Allennlp has some code in tests that is gearing for multitask setups. Ideally, we'd like to be compatible with this stuff even though it's not integrated into src yet.

Integration of multi-task code in allennlp

Info

https://github.com/allenai/allennlp/blob/master/allennlp/tests/training/multi_task_trainer_test.py

Testing

Tests I need

data constructors

verify that the metatrainer
constructor is being initialized
correctly

data splitting tests

verify splitting of data into tasks

verify shuffling of batches from
tasks

Inner loop tests

outer loop tests

Reptile tests

Verify model recreation using deep copy

Verify linear update

Maml tests

Verify the meta loss is calculated correctly

Verify that the meta loss is calculated correctly when stacking multiple loss tensors

Verify optimizers are not changing when they're not supposed to change

Validation tests

Verify that validation is working
correctly

Repo cleanup notes

The BertRegressor model is now called "general_regressor"

Doesn't seem that transition away from word splitters has occurred yet?

Missing allennlp blocks?

March 27th

Currently, I don't have a way to install futil from the package Brian gave me. I think he mentioned that it's missing entry points which maybe means that it lacks the `setup.py` file.

Switching over to DataLoader for the Homogeneous batcher

There are three things the homogeneous batch loader was doing:

- All data has to be added to a batch, the last batch cannot be dropped
- Batch size needs to be configurable
- Each batch only contains one dataset type
- The key in the instances that determines the dataset type is configurable

I think a new sampler is required

I wrote a new batch sampler that generates homogenous batches using the dataset type parameter specified in the instances.

This can be used with the off the shelf data loader and does not require creating a new data loader type.

What's the deal with the new cache directory setup?

It looks like `_setup_cache_files` is deprecated in `allennlp.training`. Not sure what mechanism handles cache stuff now.

According to the prerelease notes:

“Dataset caching is now handled entirely with a parameter passed to the dataset reader, not with command-line arguments. If you used the caching arguments to `allennlp train`, instead just add a `"cache_directory"` key to your dataset reader parameters.

Design notes 5-27

Current status:

Multi-task training is working using interleaving dataset reader, off the shelf train command and off the shelf trainer. For metalearn-trainer, I'm copying large chunks of the functionality of the default gradient descent trainer. The `from_partial_objects` method for the meta trainer should be finished though there are some issues with regard to the commented out logging-related properties. The metatrainer needs to be modified to select data from the appropriate dataloader (currently it's using the old iterator interface)

Design considerations:

I'm not using a mingler anymore for the multi-task trainer as the interleaving dataset reader handles that functionality. However, it cannot keep the datasets separate (e.g. if you have multiple dataset sources you want to sample from). These are the sampling strategies we would want to support:

- random homogeneous batches
- weighted task sampling
 - evenly weighted
- I think this should be done by writing a new batch sampler to either replace or augment the existing homogeneous batch sampler

As for the [discourse post](#), the first option is more akin to what I'm suggesting. However, the standard gradient descent trainer still can't be used as we need to do some metalearning. This would require changes to the inner loop logic in the metalearning trainer

the second option is how the older, iterator based metatrainer worked. This is possible and requires minimal changes to metatrainer code but to me seems less in line with existing allennlp implementations. In addition, this would mean multi-task training and meta training would require

different specs in futil which seems messy.