There are three ways of AGEM:

(1) Old version:

accumulate losses of previous tasks and get the gradient based on the accumulated loss and store it in grad[0]

```
offset1, offset2 = compute_offsets(0, self.nc_per_task,
                                   self.is_cifar)
ptloss = self.ce(self.forward(Variable(self.memory_data[0]),
        0)[:, offset1: offset2],Variable(self.memory_labs[0] - offset1))

for task in range(1,past_task+1):
    offset1, offset2 = compute_offsets(task, self.nc_per_task,
                                       self.is_cifar)
    ptloss += self.ce(self.forward(Variable(self.memory_data[task]),
            task)[:, offset1: offset2],Variable(self.memory_labs[task] - offset1))

ptloss.backward()
store_grad(self.parameters, self.grads, self.grad_dims,0)
```

Store the gradient of the current task in grad[1]

```
if len(self.observed_tasks) > 1:
    # copy gradient
    store_grad(self.parameters, self.grads, self.grad_dims, 1)
    #indx = torch.cuda.LongTensor(self.observed_tasks[:-1]) if self.gpu \
    #    else torch.LongTensor(self.observed_tasks[:-1])
    #dotp = torch.mm(self.grads[:, t].unsqueeze(0),
    #                 self.grads.index_select(1, indx))
    dotp = torch.mm(self.grads[:, 1].unsqueeze(0),
                    self.grads.index_select(1, torch.tensor([0],device=dev)))
    if (dotp < 0).sum() != 0:
        #project2cone2(self.grads[:, t].unsqueeze(1),
        #              self.grads.index_select(1, indx), self.margin)
        # copy gradients back
        #overwrite_grad(self.parameters, self.grads[:, t],
        #               self.grad_dims)
        project2cone2(self.grads[:, 1].unsqueeze(1),
                      self.grads.index_select(1, torch.tensor([0],device=dev)), self.margin)
        # copy gradients back
        overwrite_grad(self.parameters, self.grads[:, 1],
                       self.grad_dims)

self.opt.step()
```

Still use the same project2cone2 function as GEM

```
def project2cone2(gradient, memories, margin=0.5):
    """
        Solves the GEM dual QP described in the paper given a proposed
        gradient "gradient", and a memory of task gradients "memories".
        Overwrites "gradient" with the final projected update.
        input:  gradient, p-vector
        input:  memories, (t * p)-vector
        output: x, p-vector
    """
    memories_np = memories.cpu().t().double().numpy()
    gradient_np = gradient.cpu().contiguous().view(-1).double().numpy()
    t = memories_np.shape[0]
    P = np.dot(memories_np, memories_np.transpose())
    P = 0.5 * (P + P.transpose())
    q = np.dot(memories_np, gradient_np) * -1
    G = np.eye(t)
    h = np.zeros(t) + margin
    v = quadprog.solve_qp(P, q, G, h)[0]
    x = np.dot(v, memories_np) + gradient_np
    gradient.copy_(torch.Tensor(x).view(-1, 1))
```

Result:

```
0.7113 0.0944 0.1177 0.1740 0.1363 0.0949 0.1784 0.1826 0.1790 0.1175
0.4735 0.7152 0.1561 0.1628 0.1445 0.1107 0.1384 0.1701 0.1642 0.1512
0.4753 0.5693 0.6937 0.1477 0.1706 0.1023 0.1404 0.1654 0.1628 0.1462
0.5275 0.5841 0.5896 0.7139 0.1545 0.1052 0.1255 0.1238 0.1520 0.1110
0.5548 0.5944 0.5664 0.6588 0.6923 0.1143 0.1234 0.1513 0.1387 0.1061
0.5686 0.5186 0.5775 0.6371 0.5315 0.5899 0.1222 0.1002 0.1748 0.1575
0.3983 0.3908 0.5018 0.5558 0.4751 0.4343 0.6918 0.0920 0.1850 0.1349
0.3653 0.4004 0.4877 0.5790 0.4715 0.4391 0.5907 0.6702 0.2020 0.1140
0.3810 0.3787 0.4592 0.5695 0.4970 0.4440 0.6271 0.6587 0.6489 0.1089
0.3981 0.3939 0.4670 0.5596 0.4605 0.4498 0.5816 0.6309 0.3657 0.6847

Final Accuracy: 0.4992
Backward: -0.1820
Forward:  0.0152
```

(2) New Version:

Use new project2cone2 function according to the formula in AGEM paper

```
def project2cone2_agem(gradient, memories):
    memories_np = memories.cpu().double().numpy()
    gradient_np = gradient.cpu().double().numpy()
    g_estimate = gradient_np - ((np.matmul(gradient_np.transpose(),memories_np))/(np.matmul(memories_np.transpose(),memories_np)))*memories_np
    gradient.copy_(torch.Tensor(g_estimate).view(-1, 1))
```

The way to get loss and gradient of previous tasks remains the same as GEM

```
# compute gradient on previous tasks
if len(self.observed_tasks) > 1:
    for tt in range(len(self.observed_tasks) - 1):
        self.zero_grad()
        # fwd/bwd on the examples in the memory
        past_task = self.observed_tasks[tt]

        offset1, offset2 = compute_offsets(past_task, self.nc_per_task,
                                           self.is_cifar)
        ptloss = self.ce(
            self.forward(
                Variable(self.memory_data[past_task]),
                past_task)[:, offset1: offset2],
            Variable(self.memory_labs[past_task] - offset1))
        ptloss.backward()
        store_grad(self.parameters, self.grads, self.grad_dims,
                   past_task)
```

Use the average value of gradients of previous task instead, so notice that the argument memories in new project2cone2 function has the shape [p,1]

```python
# check if gradient violates constraints
if len(self.observed_tasks) > 1:
    # copy gradient
    store_grad(self.parameters, self.grads, self.grad_dims, t)
    indx = torch.cuda.LongTensor(self.observed_tasks[:-1]) if self.gpu \
        else torch.LongTensor(self.observed_tasks[:-1])

    #print("index:{}".format(indx))
    #print("gradient index:{}".format(torch.mean(self.grads.index_select(1, indx),1).shape))

    dotp = torch.mm(self.grads[:, t].unsqueeze(0),
                    torch.mean(self.grads.index_select(1, indx),1).unsqueeze(1))
    if (dotp < 0).sum() != 0:
        project2cone2_agem(self.grads[:, t].unsqueeze(1),
                        torch.mean(self.grads.index_select(1, indx),1).unsqueeze(1))

        #project2cone2(self.grads[:, t].unsqueeze(1),
        #               torch.Tensor([torch.mean(self.grads.index_select(1, indx),1).numpy() for _ in range(t)]), self.margin)

        #project2cone2(self.grads[:, t].unsqueeze(1),
        #               torch.mean(self.grads.index_select(1, indx),1).unsqueeze(1), self.margin)

        # copy gradients back
        overwrite_grad(self.parameters, self.grads[:, t],
                        self.grad_dims)

self.opt.step()
```

Result:

```
0.7113 0.0944 0.1177 0.1740 0.1363 0.0949 0.1784 0.1826 0.1790 0.1175
0.5788 0.6975 0.1473 0.1595 0.1429 0.1123 0.1421 0.1879 0.1580 0.1545
0.5222 0.6254 0.7034 0.1571 0.1683 0.0964 0.1542 0.2035 0.1628 0.1787
0.6049 0.6433 0.6753 0.7142 0.1443 0.1196 0.1311 0.1610 0.1447 0.1534
0.6383 0.6227 0.6055 0.6789 0.6850 0.1078 0.1269 0.1361 0.1397 0.1372
0.6386 0.5665 0.6197 0.6722 0.6264 0.6442 0.1314 0.1333 0.1404 0.1171
0.5674 0.5300 0.6573 0.6531 0.6429 0.6040 0.6987 0.1181 0.1480 0.1286
0.5824 0.4934 0.6558 0.6562 0.5974 0.5231 0.6559 0.7075 0.1765 0.1204
0.4954 0.5141 0.6185 0.6166 0.6581 0.5281 0.5753 0.7026 0.6079 0.1249
0.5330 0.5223 0.5948 0.6461 0.6670 0.5273 0.6090 0.6923 0.5972 0.7113

Final Accuracy: 0.6100
Backward: -0.0781
Forward:  0.0161
```

(3) Another way:

We still use the project2cone2 function and the way to get the loss and gradient of previous tasks. The only change is we directly put the mean value of previous gradients into project2cone2 function to use QP method to solve.

```python
if len(self.observed_tasks) > 1:
    # copy gradient
    store_grad(self.parameters, self.grads, self.grad_dims, t)
    indx = torch.cuda.LongTensor(self.observed_tasks[:-1]) if self.gpu \
        else torch.LongTensor(self.observed_tasks[:-1])

    #print("index:{}".format(indx))
    #print("gradient index:{}".format(torch.mean(self.grads.index_select(1, indx),1).shape))

    dotp = torch.mm(self.grads[:, t].unsqueeze(0),
                    torch.mean(self.grads.index_select(1, indx),1).unsqueeze(1))
    if (dotp < 0).sum() != 0:
        #project2cone2_agem(self.grads[:, t].unsqueeze(1),
        #               torch.mean(self.grads.index_select(1, indx),1).unsqueeze(1))

        #project2cone2(self.grads[:, t].unsqueeze(1),
        #               torch.Tensor([torch.mean(self.grads.index_select(1, indx),1).numpy() for _ in range(t)]), self.margin)

        project2cone2(self.grads[:, t].unsqueeze(1),
                      torch.mean(self.grads.index_select(1, indx),1).unsqueeze(1), self.margin)

        # copy gradients back
        overwrite_grad(self.parameters, self.grads[:, t],
                       self.grad_dims)
```

Result:

```
0.7113 0.0944 0.1177 0.1740 0.1363 0.0949 0.1784 0.1826 0.1790 0.1175
0.4735 0.7152 0.1561 0.1628 0.1445 0.1107 0.1384 0.1701 0.1642 0.1512
0.5184 0.5671 0.7078 0.1456 0.1429 0.1075 0.1467 0.1402 0.1583 0.1428
0.5559 0.6138 0.6238 0.6923 0.1441 0.1296 0.1264 0.1048 0.1438 0.1645
0.5663 0.5971 0.5978 0.6610 0.7014 0.1138 0.1333 0.0962 0.1488 0.1245
0.4854 0.4261 0.5987 0.5771 0.5716 0.6278 0.1268 0.0761 0.1212 0.1174
0.5135 0.4874 0.6276 0.6179 0.5894 0.5490 0.6775 0.0686 0.1444 0.1148
0.4924 0.5061 0.6397 0.6099 0.5810 0.5118 0.6536 0.7118 0.1632 0.1189
0.5046 0.5324 0.6115 0.6087 0.6203 0.5122 0.5706 0.6388 0.6808 0.1084
0.5016 0.5067 0.6201 0.6065 0.6234 0.5164 0.5681 0.6174 0.5316 0.7121

Final Accuracy: 0.5804
Backward: -0.1134
Forward:  0.0081
```

All the experiments above use the same command call:

python main.py --model Agem --n_tasks 10 --lr 0.01 --n_memories 10 --memory_strength 10 --n_layers 2 --n_hiddens 100 --data_path data/ --save_path results/ --batch_size 1 --log_every 100 --samples_per_task 1000 --cuda no --seed 3 --beta 0.03 --gamma 1.0 --memories 200 --replay_batch_size 10 --batches_per_example 10 --forgetting_mode False --forgetting_task_ids 0,5 --forgetting_resee_size 100 --sign_attacked -1.0 --num_groups 20 --cov_recompute_every 20 --create_random_groups False --divergence von_Neumann --if_output_cov False --cov_first_task_buffer 100 --data_file fashion_mnist_permutations_reduced.pt --ewc_reverse False --create_group_per_unit False

(4) The difficulty of revising modularized AGEM in the three cases above:
For all the three ways mention above, the key revision is what kind of memories we use. But as you can see from the figure below, the argument “memories” actually is not involved in the calculation of the function. Instead, pp acts like memories while h stores information about relatedness.

```python
def project2cone2(gradient, memories, current_Tid, margin=0.5,grads_groups=None,relatedness=None,args=None):
    """
        Solves the GEM dual QP described in the paper given a proposed
        gradient "gradient", and a memory of task gradients "memories".
        Overwrites "gradient" with the final projected update.
        input:  gradient, p-vector
        input:  memories, (t * p)-vector
        output: x, p-vector
    """
    gradient_np = gradient.cpu().contiguous().view(-1).double().numpy()

    h = transform_relatedness(relatedness[current_Tid],args)
    for tid in range(current_Tid):
        for pi, p in enumerate(grads_groups[tid]):
            pp = p if ((tid==0) & (pi==0)) else np.concatenate((pp, p), axis=0)

    h = np.array(h) + margin
    if args.gem_ignore_relatedness:
        h = np.zeros_like(h)
    #print("pp shape:{}".format(pp.shape))
    pp = np.expand_dims(np.mean(pp,axis=0),1)
    print("pp shape:{}".format(pp.shape))
    h = np.expand_dims(np.mean(h),0)
    print("h shape:{}".format(h.shape))
    P = np.dot(pp, pp.transpose())
    print("P shape:{}".format(P.shape))
    P = 0.5 * (P + P.transpose()) + np.eye(P.shape[0])*(1e-3)
    q =  np.dot(gradient.cpu().contiguous().view(-1).double().numpy(), pp.transpose())
    t = len(h)
    G = np.eye(t)
    G = np.eye(t) + np.eye(t)*0.00001
    try:
        v = quadprog.solve_qp(P, q, G, h)[0]
    except:
        pdb.set_trace()
    x = np.dot(v, pp) + gradient_np
    gradient.copy_(torch.Tensor(x).view(-1, 1))
```

For the old version: because there are only two indexes involved, it can’t be aligned to gradient groups or relatedness groups.

For the new version: because we use a new project2cone2 function, it’s really difficult to add relatedness information (h) and memories (pp) into the new function.

For the another way: because we have to get average of the previous gradients (memories or pp here), the size of pp now is [41, 89610], since the first index corresponds to the number of tasks, we have to average over tasks then pp’s shape will become [1,89610], in order to keep the consistence of shape, we also have to transform h (shape: [41,1]) to [1,1]. I am not sure if this idea is correct because average of h means average of relatedness. Also, the code didn’t work right. The error is called Killed:9.

```
Killed: 9
```

(5) Conclusion:

Since we should modularize AGEM based the correct AGEM code, I think we should first determine which way is correct. Then we can think about the subsequent steps.