

```

1  import pdb
2
3  import torch
4  from torch import nn
5  from torch import optim
6  from torch.nn import functional as F
7  from torch.utils.data import TensorDataset, DataLoader
8  from torch import optim
9  import numpy as np
10
11 from learner import Learner
12 from copy import deepcopy
13
14 from utils.print import highlight → adding color to output
15
16
17
18 class Meta(nn.Module):
19     """
20     Meta Learner
21     """
22     def __init__(self, args, config):
23         """
24         :param args:
25         """
26         super(Meta, self).__init__()
27
28         self.update_lr = args.update_lr
29         self.meta_lr = args.meta_lr
30         self.n_way = args.n_way
31         self.k_spt = args.k_spt
32         self.k_qry = args.k_qry
33         self.task_num = args.task_num
34         self.update_step = args.update_step
35         self.update_step_test = args.update_step_test
36
37
38         self.net = Learner(config, args.imgc, args.imgsz)
39         self.meta_optim = optim.Adam(self.net.parameters(), lr=self.meta_lr)
40
41
42
43
44
45     def clip_grad_by_norm(self, grad, max_norm):
46         """
47         in-place gradient clipping.
48         :param grad: list of gradients
49         :param max_norm: maximum norm allowable
50         :return:
51         """
52
53         total_norm = 0
54         counter = 0
55         for g in grad:
56             param_norm = g.data.norm(2)
57             total_norm += param_norm.item() ** 2
58             counter += 1
59         total_norm = total_norm ** (1. / 2)
60
61         clip_coef = max_norm / (total_norm + 1e-6)
62         if clip_coef < 1:
63             for g in grad:
64                 g.data.mul_(clip_coef)
65
66         return total_norm/counter
67
68
69     def forward(self, x_spt, y_spt, x_qry, y_qry):
70         """
71         :param x_spt: [b, setsz, c_, h, w]
72         :param y_spt: [b, setsz]
73         :param x_qry: [b, querysz, c_, h, w]
74         :param y_qry: [b, querysz]
75         :return:
76         """
77

```

```

78 # task_num : meta batch size
79 task_num, setsize, c_, h, w = x_spt.size()
80 print()
81 print(highlight('task_num :', 'blue'), task_num)
82 print(highlight('set size :', 'blue'), setsize)
83
84 # querysz = k_qry * n_way
85 querysz = x_qry.size(1)
86 print(highlight('query size :', 'blue'), querysz)
87
88 losses_q = [0 for _ in range(self.update_step + 1)] # losses_q[i] is the loss on step i
89 corrects = [0 for _ in range(self.update_step + 1)]
90
91
92 for i in range(task_num):
93
94     # 1. run the i-th task and compute loss (training loss) for k=0
95     logits = self.net(x_spt[i], vars=None, bn_training=True)
96     loss = F.cross_entropy(logits, y_spt[i])
97
98     # Compute & return sum of gradients
99     grad = torch.autograd.grad(loss, self.net.parameters())
100
101     # new weight
102     fast_weights = list(map(lambda p: p[1] - self.update_lr * p[0], zip(grad, self.net.parameters())))
103
104     # this is the loss and accuracy before first update
105     with torch.no_grad():
106         # [setsize, nway]
107         logits_q = self.net(x_qry[i], self.net.parameters(), bn_training=True)
108         loss_q = F.cross_entropy(logits_q, y_qry[i])
109         losses_q[0] += loss_q
110
111         pred_q = F.softmax(logits_q, dim=1).argmax(dim=1)
112
113         # how many preds are correct
114         correct = torch.eq(pred_q, y_qry[i]).sum().item()
115         corrects[0] = corrects[0] + correct
116
117     # this is the loss and accuracy after the first update
118     with torch.no_grad():
119         # [setsize, nway]
120         logits_q = self.net(x_qry[i], fast_weights, bn_training=True)
121         loss_q = F.cross_entropy(logits_q, y_qry[i])
122         losses_q[1] += loss_q
123         # [setsize]
124         pred_q = F.softmax(logits_q, dim=1).argmax(dim=1)
125
126         # how many preds are correct
127         correct = torch.eq(pred_q, y_qry[i]).sum().item()
128         corrects[1] = corrects[1] + correct
129
130     for k in range(1, self.update_step):
131         # 1. run the i-th task and compute loss for k=1~K-1
132         logits = self.net(x_spt[i], fast_weights, bn_training=True)
133         loss = F.cross_entropy(logits, y_spt[i])
134         # 2. compute grad on theta_pi
135         grad = torch.autograd.grad(loss, fast_weights)
136         # 3. theta_pi = theta_pi - train_lr * grad
137         fast_weights = list(map(lambda p: p[1] - self.update_lr * p[0], zip(grad, fast_weights)))
138
139         logits_q = self.net(x_qry[i], fast_weights, bn_training=True)
140         # loss_q will be overwritten and just keep the loss_q on last update step.
141         loss_q = F.cross_entropy(logits_q, y_qry[i])
142         losses_q[k + 1] += loss_q
143
144         with torch.no_grad():
145             pred_q = F.softmax(logits_q, dim=1).argmax(dim=1)
146             correct = torch.eq(pred_q, y_qry[i]).sum().item() # convert to numpy
147             corrects[k + 1] = corrects[k + 1] + correct
148
149
150 # end of all tasks
151 # sum over all losses on query set across all tasks
152 loss_q = losses_q[-1] / task_num
153
154 # optimize theta parameters
155 self.meta_optim.zero_grad()
156 loss_q.backward()
157 # print('meta update')
158

```

if vars == None:
vars = self.vars

gradient descent by support set

self.net.vars.
v.s.
self.net.parameters()

Sum of all task

update w/ support set more times.

skip first 2

model update

```

159 # for p in self.net.parameters()[1:5]:
160 #     print(torch.norm(p).item())
161 self.meta_optim.step()
162
163
164 accs = np.array(corrects) / (querysz * task_num)
165
166 return accs
167

```

```

168
169 def finetunning(self, x_spt, y_spt, x_qry, y_qry):
170     """

```

```

171     :param x_spt: [setsz, c_, h, w]
172     :param y_spt: [setsz]
173     :param x_qry: [querysz, c_, h, w]
174     :param y_qry: [querysz]
175     :return:
176     """
177

```

for evaluation, didn't change model.

```

178     assert len(x_spt.shape) == 4
179
180     querysz = x_qry.size(0)
181

```

init corrects = [0 for _ in range(self.update_step_test + 1)]

```

182     # in order to not ruin the state of running_mean/variance and bn_weight/bias
183     # we finetunning on the copied model instead of self.net
184     net = deepcopy(self.net)
185

```

```

186     # 1. run the i-th task and compute loss for k=0
187     logits = net(x_spt)
188     loss = F.cross_entropy(logits, y_spt)
189     grad = torch.autograd.grad(loss, net.parameters())
190     fast_weights = list(map(lambda p: p[1] - self.update_lr * p[0], zip(grad, net.parameters())))
191

```

```

192     # this is the loss and accuracy before first update
193

```

```

194     with torch.no_grad():
195         # [setsz, nway]
196         logits_q = net(x_qry, net.parameters(), bn_training=True)
197         # [setsz]
198         pred_q = F.softmax(logits_q, dim=1).argmax(dim=1)
199         # scalar
200         correct = torch.eq(pred_q, y_qry).sum().item()
201         corrects[0] = corrects[0] + correct
202

```

has
no
loss

effect norm. para.
⇒ need to copy model

```

203     # this is the loss and accuracy after the first update
204

```

```

205     with torch.no_grad():
206         # [setsz, nway]
207         logits_q = net(x_qry, fast_weights, bn_training=True)
208         # [setsz]
209         pred_q = F.softmax(logits_q, dim=1).argmax(dim=1)
210         # scalar
211         correct = torch.eq(pred_q, y_qry).sum().item()
212         corrects[1] = corrects[1] + correct
213

```

```

214     for k in range(1, self.update_step_test):
215

```

```

216         # 1. run the i-th task and compute loss for k=1~K-1
217         logits = net(x_spt, fast_weights, bn_training=True)
218         loss = F.cross_entropy(logits, y_spt)
219         # 2. compute grad on theta_pi
220         grad = torch.autograd.grad(loss, fast_weights)
221         # 3. theta_pi = theta_pi - train_lr * grad
222         fast_weights = list(map(lambda p: p[1] - self.update_lr * p[0], zip(grad, fast_weights)))
223

```

```

224         logits_q = net(x_qry, fast_weights, bn_training=True)
225         # loss_q will be overwritten and just keep the loss_q on last update step.
226         loss_q = F.cross_entropy(logits_q, y_qry)
227

```

```

228         with torch.no_grad():
229             pred_q = F.softmax(logits_q, dim=1).argmax(dim=1)
230             correct = torch.eq(pred_q, y_qry).sum().item() # convert to numpy
231             corrects[k + 1] = corrects[k + 1] + correct
232

```

```

233     del net
234

```

```

235     accs = np.array(corrects) / querysz
236

```

```

237     return accs
238
239

```

