

Code Listing

Marcell Veiner

May 2021

```
1 from src.datasets import set_seed
2 from src.patchwisemodel import PatchWiseModel
3 from src.imagewisemodels import BaseCNN, DynamicCapsules, NazeriCNN
  , VariationalCapsules, SRCapsules
4 from src.mixedmodels import VariationalMixedCapsules, EffNet
5 from argparse import Namespace
6
7 if __name__ == "__main__":
8     set_seed()
9
10    args_patch_wise = Namespace(
11        batch_size=32,
12        lr=0.001,
13        epochs=100,
14        augment=True,
15        flip=False,
16        workers=4,
17        classes=4,
18        input_size=[3, 512, 512],
19        output_size=[3, 64, 64],
20        predefined_stats=True,
21        data_path="./data/ICAR2018/patchwise_dataset",
22        checkpoint_path="./models/Checkpoints/",
23        name="_patchwise_"
24    )
25
26    args_img_wise = Namespace(
27        lr=0.001,
28        epochs=9,
29        augment=True,
30        flip=False,
31        workers=4,
32        classes=4,
33        routings=3,
34        lam_recon=0.392,
35        pose_dim=4,
36        batch_size=8,
37        arch=[64,16,16,16],
38        input_size=[3, 64, 64],
39        output_size=[3, 64, 64],
40        data_path="./data/ICAR2018/imagewise_dataset",
41        checkpoint_path="./models/Checkpoints/",
42        name="_imagewise_",
43        predefined_stats=False
```

```

44     )
45
46     # Example
47
48     patch_wise_model = PatchWiseModel(args_patch_wise)
49     patch_wise_model.train_model(args_patch_wise)
50     patch_wise_model.test(args_patch_wise, voting=True)
51     patch_wise_model.test_separate_classes(args_patch_wise)
52     patch_wise_model.test_training(args_patch_wise)
53     patch_wise_model.plot_metrics()
54     patch_wise_model.save_checkpoint("./models/")
55     patch_wise_model.save_model("./models/")
56
57     image_wise_model = EffNet(args_img_wise)
58     image_wise_model.train_model(args_img_wise)
59     image_wise_model.test(args_img_wise, True)
60     image_wise_model.test_separate_classes(args_img_wise)
61     image_wise_model.test_training(args_img_wise)
62     image_wise_model.plot_metrics()
63     image_wise_model.save_checkpoint("./models/")
64     image_wise_model.save_model("./models/")

```

Listing 1: main.py

```

1  from os.path import split
2  import torchvision.transforms as transforms
3  from torch.utils.data import DataLoader
4  from tqdm import tqdm
5  import numpy as np
6  import torchvision
7  import random
8  import torch
9  import torch
10 import os
11
12 VALIDATION_SET = 0.15
13 TRAINING_SET = 0.7
14 TEST_SET = 0.15
15
16 MEANS = [0.4731, 0.3757, 0.4117]
17 STD = [0.3731, 0.3243, 0.3199]
18
19 IMAGE_SIZE = (2816, 3072)
20 CROPPED_IMAGE_SIZE = (1536, 2048)
21 GRADED_LABELS = ["Grade 1", "Grade 2", "Grade 3"]
22 BACH_LABELS = ["Benign", "InSitu", "Invasive", "Normal"]
23 BREAKHIS_LABELS = ["Benign", "Malignant"]
24 SEED = 123
25
26 PATCH_SIZE = 512
27 STRIDE = 256
28
29 def imshow(img):
30     import matplotlib.pyplot as plt
31     npimg = img.numpy()
32     plt.imshow(np.transpose(npimg, (1, 2, 0)))
33     plt.show()
34

```

```

35 def set_seed(seed=SEED):
36     torch.backends.cudnn.deterministic = True
37     torch.backends.cudnn.benchmark = False
38     torch.manual_seed(seed)
39     torch.cuda.manual_seed_all(seed)
40     np.random.seed(seed)
41     random.seed(seed)
42
43 def compute_normalization_stats(root_dir, test_set=TEST_SET,
44     training_set=TRAINING_SET, val_set=VALIDATION_SET):
45     from tqdm import tqdm
46     set_seed(SEED)
47     assert test_set + training_set + val_set == 1, "Train/Test/Val
48     Set sizes incorrect"
49     BATCH_SIZE = 1
50
51     t = transforms.Compose([
52         transforms.Resize(IMAGE_SIZE),
53         transforms.ToTensor()
54     ])
55
56     data = torchvision.datasets.ImageFolder(root=root_dir+"/
57     Histopathological_Graded", transform=t)
58
59     size = int((training_set+val_set)*len(data))
60     test_size = len(data) - (size)
61     train_data, _ = torch.utils.data.random_split(data, [size,
62     test_size])
63
64     data_loader = DataLoader(train_data, batch_size=BATCH_SIZE,
65     num_workers=0)
66     # Compute normalization metrics
67     mean = 0.
68     std = 0.
69
70     # Training images
71     i = 0
72     for inputs, _ in tqdm(data_loader):
73         input = inputs[0]
74
75         temp = input.view(3, -1)
76         mean += temp.mean(1)
77         std += temp.std(1)
78         i += 1
79
80     print("Printing Normalization Metrics")
81     mean /= size
82     std /= size
83     print("Means:", mean)
84     print("Std:", std)
85
86 def split_test_train_val(root_dir, test_set=TEST_SET, training_set=
87     TRAINING_SET, val_set=VALIDATION_SET, dataset="DatabioX"):
88     from tqdm import tqdm
89     set_seed(SEED)
90     assert test_set + training_set + val_set == 1, "Train/Test/Val
91     Set sizes incorrect"

```

```

85 BATCH_SIZE = 1
86
87 if dataset == "DatabioX":
88     t = transforms.Compose([
89         transforms.Resize(IMAGE_SIZE),
90         transforms.CenterCrop(CROPPED_IMAGE_SIZE),
91         transforms.ToTensor()
92     ])
93     data = torchvision.datasets.ImageFolder(root=root_dir+"/
Histopathological_Graded", transform=t)
94     LABELS = GRADED_LABELS
95 elif dataset == "BACH":
96     t = transforms.Compose([
97         transforms.Resize(CROPPED_IMAGE_SIZE),
98         transforms.ToTensor()
99     ])
100    data = torchvision.datasets.ImageFolder(root=root_dir+"/
ICIA2018_BACH_Challenge/Photos", transform=t)
101    LABELS = BACH_LABELS
102 elif dataset == "BreakHis":
103     t = transforms.Compose([
104         transforms.Resize((PATCH_SIZE,PATCH_SIZE)),
105         transforms.ToTensor()
106     ])
107     data = torchvision.datasets.ImageFolder(root=root_dir+"/
BreakHis_v1/histology_slides/breast", transform=t)
108     LABELS = BREAKHIS_LABELS
109 else:
110     print("Dataset not recognised")
111     return
112
113 train_size = int(training_set*len(data))
114 val_size = int(val_set*len(data))
115 test_size = len(data) - (train_size + val_size)
116 train_data, test_data, val_data = torch.utils.data.random_split
(data, [train_size, test_size, val_size])
117 train_data_loader = DataLoader(train_data, batch_size=
BATCH_SIZE, num_workers=0)
118 test_data_loader = DataLoader(test_data, batch_size=BATCH_SIZE,
num_workers=0)
119 val_data_loader = DataLoader(val_data, batch_size=BATCH_SIZE,
num_workers=0)
120 print("Number of training images:", len(train_data), "Number of
test images:", len(test_data), "Number of validation images:",
len(val_data))
121
122 if not os.path.exists(root_dir):
123     os.makedirs(root_dir)
124
125 if dataset == "BreakHis":
126     for t in ["train", "test", "validation"]:
127         for i in range(len(LABELS)):
128             if not os.path.exists(root_dir + t + "/" + LABELS[i
]):
129                 os.makedirs(root_dir + t + "/" + LABELS[i])
130             for i, (inputs, labels) in tqdm(enumerate(train_data_loader
)):

```

```

131         torchvision.utils.save_image(inputs, root_dir + "/train
132         /" + LABELS[labels[0].item()] + "/image_" + str(i) + ".JPG")
133         for i, (inputs, labels) in tqdm(enumerate(val_data_loader))
134         :
135             torchvision.utils.save_image(inputs, root_dir + "/"
136             validation/" + LABELS[labels[0].item()] + "/image_" + str(i) +
137             ".JPG")
138             for i, (inputs, labels) in tqdm(enumerate(test_data_loader)
139             ):
140                 torchvision.utils.save_image(inputs, root_dir + "/test/"
141                 + LABELS[labels[0].item()] + "/image_" + str(i) + ".JPG")
142             else:
143                 for mode in ["/imagewise_dataset/", "/patchwise_dataset/"]:
144                     for t in ["train", "test", "validation"]:
145                         for i in range(len(LABELS)):
146                             if not os.path.exists(root_dir + mode + t + "/"
147                             + LABELS[i]):
148                                 os.makedirs(root_dir + mode + t + "/" +
149                                 LABELS[i])
150
151             # Training images
152             i = 0
153             for inputs, labels in tqdm(train_data_loader):
154                 input = inputs[0]
155                 # ImageWise
156                 patches = input.unfold(1, PATCH_SIZE, PATCH_SIZE).
157                 unfold(2, PATCH_SIZE, PATCH_SIZE)
158                 patches = patches.permute(1,2,0,3,4).contiguous()
159                 patches = patches.contiguous().view(-1, 3, PATCH_SIZE,
160                 PATCH_SIZE)
161                 for j, patch in enumerate(patches):
162                     torchvision.utils.save_image(patch, root_dir + "/"
163                     imagewise_dataset/train/" + LABELS[labels[0].item()] + "/image_
164                     " + str(i) + "patch_" + str(j) + ".JPG")
165
166             # PatchWise
167             patches = input.unfold(1, PATCH_SIZE, STRIDE).unfold(2,
168             PATCH_SIZE, STRIDE)
169             patches = patches.permute(1,2,0,3,4).contiguous()
170             patches = patches.contiguous().view(-1, 3, PATCH_SIZE,
171             PATCH_SIZE)
172             for j, patch in enumerate(patches):
173                 torchvision.utils.save_image(patch, root_dir + "/"
174                 patchwise_dataset/train/" + LABELS[labels[0].item()] + "/image_
175                 " + str(i) + "patch_" + str(j) + ".JPG")
176             i += 1
177
178             # Validation images
179             i = 0
180             for inputs, labels in tqdm(val_data_loader):
181                 input = inputs[0]
182                 # ImageWise
183                 patches = input.unfold(1, PATCH_SIZE, PATCH_SIZE).
184                 unfold(2, PATCH_SIZE, PATCH_SIZE)
185                 patches = patches.permute(1,2,0,3,4).contiguous()
186                 patches = patches.contiguous().view(-1, 3, PATCH_SIZE,
187                 PATCH_SIZE)

```

```

170         for j, patch in enumerate(patches):
171             torchvision.utils.save_image(patch, root_dir + "/"
imagewise_dataset/validation/" + LABELS[labels[0].item()] + "/"
image_" + str(i) + "patch_" + str(j) + ".JPG")
172
173         # PatchWise
174         patches = input.unfold(1, PATCH_SIZE, STRIDE).unfold(2,
PATCH_SIZE, STRIDE)
175         patches = patches.permute(1,2,0,3,4).contiguous()
176         patches = patches.contiguous().view(-1, 3, PATCH_SIZE,
PATCH_SIZE)
177         for j, patch in enumerate(patches):
178             torchvision.utils.save_image(patch, root_dir + "/"
patchwise_dataset/validation/" + LABELS[labels[0].item()] + "/"
image_" + str(i) + "patch_" + str(j) + ".JPG")
179         i += 1
180
181         # Test images
182         i = 0
183         for inputs, labels in tqdm(test_data_loader):
184             input = inputs[0]
185             # ImageWise
186             patches = input.unfold(1, PATCH_SIZE, PATCH_SIZE).
unfold(2, PATCH_SIZE, PATCH_SIZE)
187             patches = patches.permute(1,2,0,3,4).contiguous()
188             patches = patches.contiguous().view(-1, 3, PATCH_SIZE,
PATCH_SIZE)
189             for j, patch in enumerate(patches):
190                 torchvision.utils.save_image(patch, root_dir + "/"
imagewise_dataset/test/" + LABELS[labels[0].item()] + "/image_"
+ str(i) + "patch_" + str(j) + ".JPG")
191
192             # PatchWise
193             patches = input.unfold(1, PATCH_SIZE, STRIDE).unfold(2,
PATCH_SIZE, STRIDE)
194             patches = patches.permute(1,2,0,3,4).contiguous()
195             patches = patches.contiguous().view(-1, 3, PATCH_SIZE,
PATCH_SIZE)
196             for j, patch in enumerate(patches):
197                 torchvision.utils.save_image(patch, root_dir + "/"
patchwise_dataset/test/" + LABELS[labels[0].item()] + "/image_"
+ str(i) + "patch_" + str(j) + ".JPG")
198             i += 1
199
200 def check_res(root_dir):
201     import matplotlib.pyplot as plt
202     sizes = {}
203     data = torchvision.datasets.ImageFolder(root=root_dir+"/
Histopathological_Graded", transform=transforms.Compose([
transforms.ToTensor()])))
204     data_loader = DataLoader(data, batch_size=1, num_workers=0)
205     for inputs, _ in tqdm(data_loader):
206         t = inputs.shape
207         if t in sizes:
208             sizes[t] +=1
209         else:
210             sizes[t] = 1

```

```

211     x = [str((list(s)[2], list(s)[3])) for s in sizes.keys()]
212     y = [val for val in sizes.values()]
213     plt.pie(y, labels = x)
214     plt.show()
215

```

Listing 2: src/datasets.py

```

1  # For the docs
2  import matplotlib.pyplot as plt
3
4  # Data
5  from torch.utils.data import DataLoader, ConcatDataset
6  import torchvision.transforms as transforms
7  from .datasets import MEANS, STD
8  from tqdm import tqdm
9  import torchvision
10 import PIL
11
12 # Training
13 import torch.optim as optim
14 import torch.nn as nn
15 import numpy as np
16 import torch
17 import copy
18 import time
19 import os
20
21 # For testing we want to get a whole image in patches
22 BATCH_SIZE = 12
23
24 class Model(nn.Module):
25     """
26     Basic module class, imagewise, patchwise and mixed models
27     inherit from this one, overwriting the propagate method
28     """
29     def __init__(self, args):
30         super(Model, self).__init__()
31         self.name = args.name
32         self.time = str(time.strftime('%Y-%m-%d_%H-%M'))
33
34         if "breakhis" in args.data_path.lower():
35             self.breakhis = True
36         else:
37             self.breakhis = False
38
39     def init_device(self):
40         """ Sends model to CPU / GPU """
41         self.device = torch.device("cuda" if torch.cuda.
42 is_available() else "cpu")
43         self.to(self.device)
44
45         print(self)
46         print("Parameters:", sum(p.numel() for p in super(Model,
47 self).parameters()))
48         print("Trainable parameters:", sum(p.numel() for p in super
49 (Model, self).parameters() if p.requires_grad))
50         print("Using:", self.device)

```

```

47
48     # Additional Info when using cuda
49     if self.device.type == 'cuda':
50         print(torch.cuda.get_device_name(0))
51         print('Memory Usage:')
52         print('Allocated:', round(torch.cuda.memory_allocated(
53             0)/1024**3,1), 'GB')
54         print('Cached:   ', round(torch.cuda.memory_reserved(0)
55             /1024**3,1), 'GB')
56
57     def train_model(self, args, path=None):
58         """ Main Training loop with data augmentation, early
59         stopping and scheduler """
60         print('Start training network: {}\n'.format(time.strftime('%Y/%m/%d %H:%M')))
61
62         if not args.predefined_stats:
63             means = [0.5, 0.5, 0.5]
64             std = [0.5, 0.5, 0.5]
65         else:
66             means = MEANS
67             std = STD
68
69         validation_transforms = transforms.Compose([
70             transforms.ToTensor(),
71             transforms.Normalize(mean=means, std=std)
72         ])
73
74         if args.augment:
75             """
76             Create versions of the dataset for each augmentation as
77             in https://arxiv.org/abs/1803.04054 and others
78             """
79             augmenting = [
80                 # 0 degrees
81                 transforms.Compose([
82                     transforms.ColorJitter(hue=.05, saturation=.05)
83
84                     ,
85                     transforms.ToTensor(),
86                     transforms.Normalize(mean=means, std=std)
87                 ]),
88                 # 90 degrees
89                 transforms.Compose([
90                     transforms.RandomRotation((90, 90), resample=
91                         PIL.Image.BILINEAR),
92                     transforms.ColorJitter(hue=.05, saturation=.05)
93
94                     ,
95                     transforms.ToTensor(),
96                     transforms.Normalize(mean=means, std=std)
97                 ]),
98                 # 180 degrees
99                 transforms.Compose([
100                     transforms.RandomRotation((180, 180), resample=
101                         PIL.Image.BILINEAR),
102                     transforms.ColorJitter(hue=.05, saturation=.05)
103
104                     ,
105                     transforms.ToTensor(),
106                     transforms.Normalize(mean=means, std=std)
107                 ])
108             ]

```



```

94         transforms.Normalize(mean=means, std=std)
95     ]),
96     # 270 degrees + flip
97     transforms.Compose([
98         transforms.RandomRotation((270, 270), resample=
PIL.Image.BILINEAR),
99         transforms.ColorJitter(hue=.05, saturation=.05)
100     ,
101         transforms.ToTensor(),
102         transforms.Normalize(mean=means, std=std)
103     ])
104 ]
105
106 if args.flip:
107     augmenting += [
108         transforms.Compose([
109             transforms.RandomVerticalFlip(p=1.),
110             transforms.ColorJitter(hue=.05, saturation
=.05),
111             transforms.ToTensor(),
112             transforms.Normalize(mean=means, std=std)
113         ]),
114         transforms.Compose([
115             transforms.RandomVerticalFlip(p=1.),
116             transforms.RandomRotation((90, 90),
resample=PIL.Image.BILINEAR),
117             transforms.ColorJitter(hue=.05, saturation
=.05),
118             transforms.ToTensor(),
119             transforms.Normalize(mean=means, std=std)
120         ]),
121         transforms.Compose([
122             transforms.RandomVerticalFlip(p=1.),
123             transforms.RandomRotation((180, 180),
resample=PIL.Image.BILINEAR),
124             transforms.ColorJitter(hue=.05, saturation
=.05),
125             transforms.ToTensor(),
126             transforms.Normalize(mean=means, std=std)
127         ]),
128         transforms.Compose([
129             transforms.RandomVerticalFlip(p=1.),
130             transforms.RandomRotation((270, 270),
resample=PIL.Image.BILINEAR),
131             transforms.ColorJitter(hue=.05, saturation
=.05),
132             transforms.ToTensor(),
133             transforms.Normalize(mean=means, std=std)
134         ])
135     ]
136
137 train_data_loader = DataLoader(
138     ConcatDataset([
139         torchvision.datasets.ImageFolder(root=args.
data_path + "/train", transform=t) for t in augmenting
140     ]),
141     batch_size=args.batch_size, shuffle=True,

```

```

num_workers=args.workers
    )
141
142
143     else:
144         training_transforms = transforms.Compose([
145             transforms.RandomHorizontalFlip(),
146             transforms.RandomVerticalFlip(),
147             transforms.RandomRotation(10, resample=PIL.Image.
BILINEAR),
148             transforms.ColorJitter(hue=.05, saturation=.05),
149             transforms.ToTensor(),
150             transforms.Normalize(mean=means, std=std)
151         ])
152         train_data = torchvision.datasets.ImageFolder(root=args
.data_path + "/train", transform=training_transforms)
153         train_data_loader = DataLoader(train_data, batch_size=
args.batch_size, shuffle=True, num_workers=args.workers)
154
155         val_data = torchvision.datasets.ImageFolder(root=args.
data_path + "/validation", transform=validation_transforms)
156         val_data_loader = DataLoader(val_data, batch_size=args.
batch_size, shuffle=True, num_workers=args.workers)
157
158         print("Using ", len(train_data_loader.dataset), "training
samples")
159         print("Using ", len(val_data_loader.dataset), "validation
samples")
160
161         optimizer = optim.Adam(self.parameters(), lr=args.lr)
162         scheduler = optim.lr_scheduler.StepLR(optimizer, step_size
=20, gamma=0.1)
163         criterion = nn.CrossEntropyLoss()
164         start_epoch = 0
165
166         # If checkpoint provided load states
167         if path:
168             optimizer, start_epoch = self.load_ckp(path, optimizer)
169             print("Model loaded, trained for ", start_epoch, "
epochs")
170
171         # keeping track of losses
172         self.train_losses = []
173         self.valid_losses = []
174         self.train_acc = []
175         self.val_acc = []
176         since = time.time()
177
178         # For "early stopping"
179         best_model_wts = copy.deepcopy(self.state_dict())
180         best_acc = 0.
181
182         for epoch in range(start_epoch, args.epochs):
183             print('Epoch {}/{}'.format(epoch+1, args.epochs))
184             print('-' * 10)
185
186             # Each epoch has a training and validation phase
187             for phase in ['train', 'val']:

```

```

188         if phase == 'train':
189             super(Model, self).train() # Set model to
training mode
190             dataloader = train_data_loader
191         else:
192             super(Model, self).eval() # Set model to
evaluate mode
193             dataloader = val_data_loader
194
195             running_loss = 0.0
196             running_corrects = 0
197
198             # Iterate over data.
199             for inputs, labels in tqdm(dataloader):
200                 inputs = inputs.to(self.device)
201                 labels = labels.to(self.device)
202
203                 # zero the parameter gradients
204                 optimizer.zero_grad()
205
206                 # Training here
207                 with torch.set_grad_enabled(phase == 'train'):
208                     # Overwrite this for each model
209                     loss, preds = self.propagate(inputs, labels
, criterion)
210
211                     _, preds = torch.max(preds, 1)
212
213                     # backward + optimize only if in training
phase
214                     if phase == 'train':
215                         loss.backward()
216                         optimizer.step()
217
218                     # statistics
219                     running_loss += loss.item() * inputs.size(0)
220                     running_corrects += torch.sum(preds == labels.
data)
221
222             # Adjust learning rate
223             if phase == 'train':
224                 scheduler.step()
225
226             # Data metrics
227             epoch_loss = running_loss / len(dataloader.dataset)
228             epoch_acc = running_corrects.double() / len(
dataloader.dataset)
229
230             print('{} Loss: {:.4f} Acc: {:.4f}'.format(
phase, epoch_loss, epoch_acc))
231
232             if phase == 'train':
233                 self.train_losses.append(epoch_loss)
234                 self.train_acc.append(epoch_acc)
235             else:
236                 self.valid_losses.append(epoch_loss)
237                 self.val_acc.append(epoch_acc)
238

```

```

239         # deep copy the model
240         if phase == 'val' and epoch_acc > best_acc:
241             best_acc = epoch_acc
242             best_model_wts = copy.deepcopy(self.state_dict
243
244             checkpoint = {
245                 'epoch': epoch + 1,
246                 'state_dict': best_model_wts,
247                 'optimizer': optimizer.state_dict(),
248                 'loss': criterion
249             }
250             file_name = "checkpoint_" + str(epoch + 1) +
args.name + self.time + ".ckpt"
251             torch.save(checkpoint, args.checkpoint_path +
file_name)
252
253         # Finished
254         time_elapsed = time.time() - since
255         print('Training complete in {:.0f}m {:.0f}s'.format(
256             time_elapsed // 60, time_elapsed % 60))
257         print('Best Validation Accuracy: {:.4f}'.format(best_acc))
258
259         # load best model weights and save checkpoint
260         self.load_state_dict(best_model_wts)
261
262     def propagate(self, inputs, labels, criterion=None):
263         """ Default Training step - some models use this """
264         outputs = self(inputs)
265         if criterion:
266             loss = criterion(outputs, labels)
267         else:
268             loss = 0
269         return loss, outputs
270
271     def plot_metrics(self, path, pr=False, pl=True):
272         """ Plots accuracy and loss side-by-side """
273
274         # Plotting on HPC throws error
275         if pl:
276             # Loss
277             plt.subplot(1, 2, 1)
278             plt.plot(self.train_losses, label='Training loss')
279             plt.plot(self.valid_losses, label='Validation loss')
280             plt.xlabel("Epochs")
281             plt.ylabel("Loss")
282             plt.legend(frameon=False)
283             plt.savefig(path + "loss.png")
284
285             # Accuracy
286             plt.subplot(1, 2, 2)
287             plt.plot(self.train_acc, label='Training Accuracy')
288             plt.plot(self.val_acc, label='Validation Accuracy')
289             plt.xlabel("Epochs")
290             plt.ylabel("Acc")
291             plt.legend(frameon=False)
292             plt.savefig(path + "accuracy.png")

```

```

293         if pr:
294             print(self.train_losses, self.valid_losses, self.
train_acc, self.val_acc, sep="\n")
295
296     def test(self, args, voting=False):
297         """ Test on patched dataset """
298
299         if not args.predefined_stats:
300             means = [0.5, 0.5, 0.5]
301             std = [0.5, 0.5, 0.5]
302         else:
303             means = MEANS
304             std = STD
305
306         test_data = torchvision.datasets.ImageFolder(root=args.
data_path + "/test", transform=transforms.Compose([
307             transforms.ToTensor(),
308             transforms.Normalize(mean=means, std=std)
309         ]))
310         test_data_loader = DataLoader(test_data, batch_size=
BATCH_SIZE, shuffle=False, num_workers=args.workers)
311
312         super(Model, self).eval()
313         with torch.no_grad():
314             patch_acc = 0
315             image_acc_maj = 0
316             image_acc_sum = 0
317             image_acc_max = 0
318             conf = []
319             for images, labels in tqdm(test_data_loader):
320                 images = images.to(self.device)
321                 labels = labels.to(self.device)
322                 _, preds = self.propagate(images, labels)
323                 _, predicted = torch.max(preds, 1)
324                 patch_acc += (predicted == labels).sum().item()
325
326                 if not self.breakhis and voting:
327                     # Voting
328                     preds = preds.cpu()
329                     predicted = predicted.cpu()
330
331                     maj_prob = (args.classes - 1) - np.argmax(np.
sum(np.eye(args.classes)[np.array(predicted).reshape(-1)], axis
=0)[::-1])
332                     sum_prob = (args.classes - 1) - np.argmax(np.
sum(np.exp(preds.numpy()), axis=0)[::-1])
333                     max_prob = (args.classes - 1) - np.argmax(np.
max(np.exp(preds.numpy()), axis=0)[::-1])
334
335                     confidence = np.sum(np.array(predicted) ==
maj_prob) / predicted.size(0)
336                     conf.append(np.round(confidence * 100, 2))
337
338                     if labels.data[0].item() == maj_prob:
339                         image_acc_maj += 1
340
341                     if labels.data[0].item() == sum_prob:

```

```

342         image_acc_sum += 1
343
344         if labels.data[0].item() == max_prob:
345             image_acc_max += 1
346
347     patch_acc /= len(test_data_loader.dataset)
348     print('Test Accuracy of the model: {:.2f}'.format(patch_acc
349 ))
350
351     if not self.breakhis and voting:
352         print('Average Confidence: {:.2f}'.format(sum(conf)/len
353 (conf)))
354         image_acc_maj /= (len(test_data_loader.dataset)/12)
355         image_acc_sum /= (len(test_data_loader.dataset)/12)
356         image_acc_max /= (len(test_data_loader.dataset)/12)
357
358         print('Test Accuracy of the model on with majority
359 voting: {:.2f}'.format(image_acc_maj))
360         print('Test Accuracy of the model on with sum voting:
361 {:.2f}'.format(image_acc_sum))
362         print('Test Accuracy of the model on with max voting:
363 {:.2f}'.format(image_acc_max))
364
365 def test_separate_classes(self, args):
366     """ Tests the model on each class separately and reports
367 classification metrics """
368     if not args.predefined_stats:
369         means = [0.5, 0.5, 0.5]
370         std = [0.5, 0.5, 0.5]
371     else:
372         means = MEANS
373         std = STD
374
375     test_data = torchvision.datasets.ImageFolder(root=args.
376 data_path + "/test", transform=transforms.Compose([
377         transforms.ToTensor(),
378         transforms.Normalize(mean=means, std=std)
379     ]))
380     test_data_loader = DataLoader(test_data, batch_size=args.
381 batch_size, num_workers=args.workers)
382     conf_matrix = torch.zeros(args.classes, args.classes)
383
384     super(Model, self).eval()
385     with torch.no_grad():
386         for images, labels in tqdm(test_data_loader):
387             images = images.to(self.device)
388             labels = labels.to(self.device)
389             _, predicted = self.propagate(images, labels)
390             _, predicted = torch.max(predicted, 1)
391             predicted = predicted.tolist()
392             labels = labels.tolist()
393
394             for t, p in zip(labels, predicted):
395                 conf_matrix[t, p] += 1
396
397     print('Confusion matrix\n', conf_matrix)

```

```

391     TP = conf_matrix.diag()
392     for c in range(args.classes):
393         idx = torch.ones(args.classes)
394         idx = idx.type(torch.BoolTensor)
395         idx[c] = 0
396         # all non-class samples classified as non-class
397         TN = conf_matrix[idx.nonzero(as_tuple=False)[:], None],
idx.nonzero(as_tuple=False)].sum() #conf_matrix[idx[:], None],
idx].sum() - conf_matrix[idx, c].sum()
398         # all non-class samples classified as class
399         FP = conf_matrix[idx, c].sum()
400         # all class samples not classified as class
401         FN = conf_matrix[c, idx].sum()
402
403         print('Class {} \nTP {}, TN {}, FP {}, FN {}'.format(
404             c, TP[c], TN, FP, FN))
405         print('Sensitivity {:.2f}, Specificity {:.2f}, F1 {:.2f}
}, Accuracy {:.2f}'.format(
406             TP[c] / (TP[c]+FN), TN / (TN + FP), 2*TP[c] / (2*TP
[c] + FP + FN), ((TP[c] + TN) / (TP[c] + TN + FP + FN))))
407
408     def test_training(self, args):
409         """ Test on patched training dataset for debugging """
410
411         if not args.predefined_stats:
412             means = [0.5, 0.5, 0.5]
413             std = [0.5, 0.5, 0.5]
414         else:
415             means = MEANS
416             std = STD
417
418         train_data = torchvision.datasets.ImageFolder(root=args.
data_path + "/train", transform=transforms.Compose([
419             transforms.ToTensor(),
420             transforms.Normalize(mean=means, std=std)
421         ]))
422         train_data_loader = DataLoader(train_data, batch_size=args.
batch_size, shuffle=True, num_workers=args.workers)
423
424         super(Model, self).eval()
425         with torch.no_grad():
426             correct = 0
427             for images, labels in tqdm(train_data_loader):
428                 images = images.to(self.device)
429                 labels = labels.to(self.device)
430                 _, predicted = self.propagate(images, labels)
431                 _, predicted = torch.max(predicted, 1)
432                 correct += (predicted == labels).sum().item()
433
434             print('Training Accuracy of the model: {:.2f}'.format(
correct / len(train_data_loader.dataset)))
435
436     def save_model(self, path):
437         """ Save model after training has finished """
438         file_name = path + self.name + self.time + ".ckpt"
439         torch.save(self.state_dict(), file_name)
440         print("Model saved:", file_name)

```

```

441         return file_name
442
443     def load(self, path):
444         """ Load pre-trained weights """
445         try:
446             if os.path.exists(path):
447                 print('Loading model...')
448                 self.load_state_dict(torch.load(path, map_location=
self.device))
449             except:
450                 print('Failed to load pre-trained network with path:',
path)
451
452     def load_ckp(self, checkpoint_fpath, optimizer):
453         """ To continue training we need more than just saving the
weights """
454         checkpoint = torch.load(checkpoint_fpath, map_location=self
.device)
455         self.load_state_dict(checkpoint['state_dict'])
456         optimizer.load_state_dict(checkpoint['optimizer'])
457         return optimizer, checkpoint['epoch']

```

Listing 3: src/model.py

```

1 import torch.nn.functional as F
2 from .model import Model
3 import torch.nn as nn
4
5
6 class PatchWiseModel(Model):
7     """
8     A CNN classifier that is used by the image-wise networks to
downscale the images
9     by feeding them though the convolutional layers of the trained
patchwise net.
10    """
11    def __init__(self, args, original_architecture=False):
12        super(PatchWiseModel, self).__init__(args)
13        if original_architecture:
14            """
15            This is the original architecture proposed in: https://arxiv.org/abs/1803.04054
16            """
17            self.features = nn.Sequential(
18                # Block 1
19                nn.Conv2d(in_channels=args.input_size[0],
out_channels=16, kernel_size=3, stride=1, padding=1),
20                nn.BatchNorm2d(16),
21                nn.ReLU(inplace=True),
22                nn.Conv2d(in_channels=16, out_channels=16,
kernel_size=3, stride=1, padding=1),
23                nn.BatchNorm2d(16),
24                nn.ReLU(inplace=True),
25                nn.Conv2d(in_channels=16, out_channels=16,
kernel_size=2, stride=2),
26                nn.BatchNorm2d(16),
27                nn.ReLU(inplace=True),
28

```



```

29         # Block 2
30         nn.Conv2d(in_channels=16, out_channels=32,
31         kernel_size=3, stride=1, padding=1),
32         nn.BatchNorm2d(32),
33         nn.ReLU(inplace=True),
34         nn.Conv2d(in_channels=32, out_channels=32,
35         kernel_size=3, stride=1, padding=1),
36         nn.BatchNorm2d(32),
37         nn.ReLU(inplace=True),
38         nn.Conv2d(in_channels=32, out_channels=32,
39         kernel_size=2, stride=2),
40         nn.BatchNorm2d(32),
41         nn.ReLU(inplace=True),
42
43         # Block 3
44         nn.Conv2d(in_channels=32, out_channels=64,
45         kernel_size=3, stride=1, padding=1),
46         nn.BatchNorm2d(64),
47         nn.ReLU(inplace=True),
48         nn.Conv2d(in_channels=64, out_channels=64,
49         kernel_size=3, stride=1, padding=1),
50         nn.BatchNorm2d(64),
51         nn.ReLU(inplace=True),
52         nn.Conv2d(in_channels=64, out_channels=64,
53         kernel_size=2, stride=2),
54         nn.BatchNorm2d(64),
55         nn.ReLU(inplace=True),
56
57         # Block 4
58         nn.Conv2d(in_channels=64, out_channels=128,
59         kernel_size=3, stride=1, padding=1),
60         nn.BatchNorm2d(128),
61         nn.ReLU(inplace=True),
62         nn.Conv2d(in_channels=128, out_channels=128,
63         kernel_size=3, stride=1, padding=1),
64         nn.BatchNorm2d(128),
65         nn.ReLU(inplace=True),
66         nn.Conv2d(in_channels=128, out_channels=128,
67         kernel_size=3, stride=1, padding=1),
68         nn.BatchNorm2d(128),
69         nn.ReLU(inplace=True),
70
71         # Block 5
72         nn.Conv2d(in_channels=128, out_channels=256,
73         kernel_size=3, stride=1, padding=1),
74         nn.BatchNorm2d(256),
75         nn.ReLU(inplace=True),
76         nn.Conv2d(in_channels=256, out_channels=256,
77         kernel_size=3, stride=1, padding=1),
78         nn.BatchNorm2d(256),
79         nn.ReLU(inplace=True),
80         nn.Conv2d(in_channels=256, out_channels=256,
81         kernel_size=3, stride=1, padding=1),
82         nn.BatchNorm2d(256),
83         nn.ReLU(inplace=True),
84         nn.Conv2d(in_channels=256, out_channels=args.

```

```

74     output_size[0], kernel_size=1, stride=1),
75     )
76     else:
77         """
78         Smaller version using 10 conv layers instead of 16
79         """
80         self.features = nn.Sequential(
81             # Block 1
82             nn.Conv2d(in_channels=args.input_size[0],
83 out_channels=16, kernel_size=3, stride=1, padding=1),
84             nn.BatchNorm2d(16),
85             nn.ReLU(inplace=True),
86             nn.Conv2d(in_channels=16, out_channels=16,
87 kernel_size=3, stride=1, padding=1),
88             nn.BatchNorm2d(16),
89             nn.ReLU(inplace=True),
90             # Block 2
91             nn.Conv2d(in_channels=16, out_channels=32,
92 kernel_size=3, stride=1, padding=1),
93             nn.BatchNorm2d(32),
94             nn.ReLU(inplace=True),
95             nn.Conv2d(in_channels=32, out_channels=32,
96 kernel_size=3, stride=1, padding=1),
97             nn.BatchNorm2d(32),
98             nn.ReLU(inplace=True),
99             nn.Conv2d(in_channels=32, out_channels=32,
100 kernel_size=2, stride=2),
101             nn.BatchNorm2d(32),
102             nn.ReLU(inplace=True),
103             # Block 3
104             nn.Conv2d(in_channels=32, out_channels=64,
105 kernel_size=3, stride=1, padding=1),
106             nn.BatchNorm2d(64),
107             nn.ReLU(inplace=True),
108             nn.Conv2d(in_channels=64, out_channels=64,
109 kernel_size=3, stride=1, padding=1),
110             nn.BatchNorm2d(64),
111             nn.ReLU(inplace=True),
112             nn.Conv2d(in_channels=64, out_channels=args.
113 output_size[0], kernel_size=1, stride=1),
114         )
115
116         # The classification layer
117         self.classifier = nn.Sequential(
118             nn.Linear(args.output_size[0] * args.output_size[1] *
args.output_size[2], args.classes),

```

```

119         )
120
121         self.initialize_weights()
122
123         # This will send net to device, so only call here
124         self.init_device()
125
126     def initialize_weights(self):
127         """ As in https://arxiv.org/abs/1803.04054 """
128         for m in self.modules():
129             if isinstance(m, nn.Conv2d):
130                 nn.init.kaiming_normal_(m.weight, nonlinearity='
relu')
131
132                 if m.bias is not None:
133                     m.bias.data.zero_()
134
135             elif isinstance(m, nn.BatchNorm2d):
136                 m.weight.data.fill_(1)
137                 m.bias.data.zero_()
138
139             elif isinstance(m, nn.Linear):
140                 m.weight.data.normal_(0, 0.01)
141                 m.bias.data.zero_()
142
143     def forward(self, x):
144         x = self.features(x)
145         x = x.view(x.size(0), -1)
146         x = self.classifier(x)
147         x = F.log_softmax(x, dim=1)
148         return x

```

Listing 4: src/patchwisemodel.py

```

1  # Base Models
2  from .patchwisemodel import PatchWiseModel
3  from .model import Model
4
5  # Dynamic
6  from .DynamicCaps.capsulelayers import DenseCapsule, PrimaryCapsule
7  from .DynamicCaps.capsulenet import caps_loss
8
9  # Varcaps
10 from .VarCaps import layers
11 from .VarCaps import vb_routing
12
13 # SR Capsules
14 from .SRCaps.modules import SelfRouting2d
15
16 # Data
17 import torchvision.transforms as transforms
18 from torch.utils.data import DataLoader
19 from .datasets import MEANS, STD
20 import torchvision
21
22 # Training
23 from torch.autograd import Variable
24 import torch.nn.functional as F
25 import torch.nn as nn

```

```

26 import numpy as np
27 import torch
28
29
30 class ImageWiseModels(Model):
31     """
32     Base Image-Wise model, variants inherit from this class
33     it assigns a hopefully trained patchwise net to the model to
34     use for forward passes
35     """
36     def __init__(self, args, patchwise, original_architecture):
37         super(ImageWiseModels, self).__init__(args)
38
39         if patchwise is not None:
40             self.patch_wise_model = patchwise
41         else:
42             print("Creating untrained patchwise model")
43             self.patch_wise_model = PatchWiseModel(args,
44 original_architecture=original_architecture)
45
46     def propagate(self, inputs, labels, criterion=None):
47         inputs = self.patch_wise_model.features(inputs)
48         outputs = self(inputs)
49         if criterion:
50             loss = criterion(outputs, labels)
51         else:
52             loss = 0
53         return loss, outputs
54
55 class BaseCNN(ImageWiseModels):
56     """ Simpler CNN baseline than Nazeri """
57     def __init__(self, args, patchwise=None, original_architecture=
58 False):
59         super(BaseCNN, self).__init__(args, patchwise,
60 original_architecture)
61
62         self.cnn_layers = nn.Sequential(
63             # Convolutional Layer 1
64             nn.Conv2d(args.input_size[0], 32, kernel_size=5, stride
65 =1, bias=False),
66             nn.BatchNorm2d(32),
67             nn.ReLU(inplace=True),
68             nn.MaxPool2d(3, 1),
69
70             # Convolutional Layer 2
71             nn.Conv2d(32, 32, kernel_size=5, stride=1, bias=False),
72             nn.BatchNorm2d(32),
73             nn.ReLU(inplace=True),
74             nn.MaxPool2d(3, 1),
75
76             # Convolutional Layer 3
77             nn.Conv2d(32, 16, kernel_size=5, stride=1, bias=False),
78             nn.BatchNorm2d(16),
79             nn.ReLU(inplace=True),
80             nn.MaxPool2d(3, 1),
81         )

```

```

78     self.device = torch.device("cuda:0" if torch.cuda.
is_available() else "cpu")
79     self.to(self.device)
80
81     self.set_linear_layer(args)
82
83     # This will send net to device, so only call here
84     self.init_device()
85
86     def forward(self, x):
87         x = self.cnn_layers(x)
88         x = x.view(x.size(0), -1)
89         x = self.linear_layers(x)
90         return x
91
92     def set_linear_layer(self, args):
93         """ Pytorch has no nice way of connecting CNNs with the
denselayers, this code sets the dimensions correctly on the fly
"""
94         train_data = torchvision.datasets.ImageFolder(root=args.
data_path + "/train", transform=transforms.Compose([
95             transforms.ToTensor(),
96             transforms.Normalize(mean=MEANS, std=STD)
97         ]))
98         train_data_loader = DataLoader(train_data, batch_size=args.
batch_size, shuffle=True, num_workers=args.workers)
99         super(BaseCNN, self).train() # Set model to training mode
100         data, _ = next(iter(train_data_loader))
101         #data = data[0]
102         x = data.to(self.device)
103         x = self.patch_wise_model.features(x)
104         x = self.cnn_layers(x)
105
106         print("Setting Linear Layer with shape:", x.shape)
107         self.linear_layers = nn.Sequential(
108             nn.Linear(x.size(1)*x.size(2)*x.size(3), 64),
109             nn.ReLU(inplace=True),
110             nn.Dropout(0.5),
111             nn.Linear(64, 32),
112             nn.ReLU(inplace=True),
113             #nn.Dropout(0.5),
114             nn.Linear(32, args.classes) # NO softmax, bc it is in
crossentropy loss
115         )
116         self.to(self.device)
117
118     class NazeriCNN(ImageWiseModels):
119         """ CNN imagewise network as in https://arxiv.org/abs/1803.04054 """
120         def __init__(self, args, patchwise=None, original_architecture=
False):
121             super(NazeriCNN, self).__init__(args, patchwise,
original_architecture)
122
123             self.features = nn.Sequential(
124                 # Block 1
125                 nn.Conv2d(in_channels=args.input_size[0], out_channels

```

```

126         =64, kernel_size=3, stride=1, padding=1),
127             nn.BatchNorm2d(64),
128             nn.ReLU(inplace=True),
129             nn.Conv2d(in_channels=64, out_channels=64, kernel_size
130 =3, stride=1, padding=1),
131             nn.BatchNorm2d(64),
132             nn.ReLU(inplace=True),
133             nn.Conv2d(in_channels=64, out_channels=64, kernel_size
134 =2, stride=2),
135             nn.BatchNorm2d(64),
136             nn.ReLU(inplace=True),
137
138         # Block 2
139         nn.Conv2d(in_channels=64, out_channels=128, kernel_size
140 =3, stride=1, padding=1),
141             nn.BatchNorm2d(128),
142             nn.ReLU(inplace=True),
143             nn.Conv2d(in_channels=128, out_channels=128,
144 kernel_size=3, stride=1, padding=1),
145             nn.BatchNorm2d(128),
146             nn.ReLU(inplace=True),
147             nn.Conv2d(in_channels=128, out_channels=128,
148 kernel_size=2, stride=2),
149             nn.BatchNorm2d(128),
150             nn.ReLU(inplace=True),
151             nn.Conv2d(in_channels=128, out_channels=1, kernel_size
152 =1, stride=1),
153         )
154
155         self.classifier = nn.Sequential(
156             nn.Linear(1 * 16 * 16, 128),
157             nn.ReLU(inplace=True),
158             nn.Dropout(0.5),
159
160             nn.Linear(128, 128),
161             nn.ReLU(inplace=True),
162             nn.Dropout(0.5),
163
164             nn.Linear(128, 64),
165             nn.ReLU(inplace=True),
166             nn.Dropout(0.5),
167
168             nn.Linear(64, args.classes),
169         )
170
171         # This will send net to device, so only call here
172         self.init_device()
173
174     def forward(self, x):
175         x = self.features(x)
176         x = x.view(x.size(0), -1)
177         x = self.classifier(x)
178         x = F.log_softmax(x, dim=1)
179         return x
180
181     def initialize_weights(self):

```

```

176     """ As in https://arxiv.org/abs/1803.04054 """
177     for m in self.modules():
178         if isinstance(m, nn.Conv2d):
179             nn.init.kaiming_normal_(m.weight, nonlinearity='
relu')
180             if m.bias is not None:
181                 m.bias.data.zero_()
182
183         elif isinstance(m, nn.BatchNorm2d):
184             m.weight.data.fill_(1)
185             m.bias.data.zero_()
186
187         elif isinstance(m, nn.Linear):
188             m.weight.data.normal_(0, 0.01)
189             m.bias.data.zero_()
190
191 class DynamicCapsules(ImageWiseModels):
192     """
193     A Capsule Network adapted from https://github.com/XifengGuo/
CapsNet-Pytorch
194     """
195     def __init__(self, args, patchwise=None, original_architecture=
False):
196         super(DynamicCapsules, self).__init__(args, patchwise,
original_architecture)
197         self.output_size = args.output_size
198         self.classes = args.classes
199         self.routings = args.routings
200         self.lam_recon = args.lam_recon
201         self.device = torch.device("cuda:0" if torch.cuda.
is_available() else "cpu")
202
203         # Layer 1: Just a conventional Conv2D layer
204         self.conv1 = nn.Conv2d(args.output_size[0], 64, kernel_size
=9, stride=1, padding=0)
205
206         # Layer 2: Conv2D layer with 'squash' activation, then
reshape to [None, num_caps, dim_caps]
207         self.primarycaps = PrimaryCapsule(64, 64, 8, kernel_size=9,
stride=2, padding=0)
208
209         # Layer 3: Capsule layer. Routing algorithm works here.
210         self.digitcaps = DenseCapsule(in_num_caps=4608, in_dim_caps
=8,
211                                     out_num_caps=args.classes,
out_dim_caps=16, routings=self.routings, device=self.device)
212
213         # Decoder network.
214         self.decoder = nn.Sequential(
215             nn.Linear(16*args.classes, 512),
216             nn.ReLU(inplace=True),
217             nn.Linear(512, 1024),
218             nn.ReLU(inplace=True),
219             nn.Linear(1024, args.output_size[0] * args.output_size
[1] * args.output_size[2]),
220             nn.Sigmoid()
221         )

```

```

222         self.relu = nn.ReLU()
223
224
225         # This will send net to device, so only call here
226         self.init_device()
227
228     def forward(self, x, y=None):
229         x = self.relu(self.conv1(x))
230         x = self.primarycaps(x)
231         x = self.digitcaps(x)
232         length = x.norm(dim=-1)
233         if y is None: # during testing, no label given. create one
-hot coding using 'length'
234             index = length.max(dim=1)[1]
235             y = Variable(torch.zeros(length.size()).scatter_(1,
index.view(-1, 1).cpu().data, 1.).to(self.device))
236             reconstruction = self.decoder((x * y[:, :, None]).view(x.
size(0), -1))
237             return length, reconstruction.view(-1, *self.output_size)
238
239     def propagate(self, inputs, labels, criterion=None):
240         labels = torch.zeros(labels.size(0), self.classes).to(self.
device).scatter_(1, labels.view(-1, 1), 1.) # change to one-
-hot coding
241         inputs = self.patch_wise_model.features(inputs)
242         if criterion:
243             y_pred, x_recon = self(inputs, labels)
244             loss = caps_loss(labels, y_pred, inputs, x_recon, self.
lam_recon) # compute loss
245         else:
246             y_pred, x_recon = self(inputs) # No y in testing
247             loss = 0
248         return loss, y_pred
249
250 class VariationalCapsules(ImageWiseModels):
251     """
252     Capsule Routing via Variational Bayes based on https://github.
com/fabio-deep/Variational-Capsule-Routing
253     """
254     def __init__(self, args, patchwise=None, original_architecture=
False):
255         super(VariationalCapsules, self).__init__(args, patchwise,
original_architecture)
256
257         self.output_size = args.output_size
258         self.n_classes = args.classes
259         self.routings = args.routings
260
261         self.P = args.pose_dim
262         self.PP = int(np.max([2, self.P*self.P]))
263         self.A, self.B, self.C, self.D = args.arch
264
265         # Layer 1: Just a conventional Conv2D layer
266         self.Conv_1 = nn.Conv2d(self.output_size[0], self.A,
kernel_size=5, stride=2, bias=False)
267         nn.init.kaiming_uniform_(self.Conv_1.weight)
268

```



```

269         self.BN_1 = nn.BatchNorm2d(self.A)
270         self.PrimaryCaps = layers.PrimaryCapsules2d(in_channels=
self.A, out_caps=self.B,
271             kernel_size=1, stride=1, pose_dim=self.P)
272
273         self.ConvCaps_1 = layers.ConvCapsules2d(in_caps=self.B,
out_caps=self.C,
274             kernel_size=3, stride=2, pose_dim=self.P)
275
276         self.ConvRouting_1 = vb_routing.VariationalBayesRouting2d(
in_caps=self.B, out_caps=self.C,
277             kernel_size=3, stride=2, pose_dim=self.P,
278             cov='diag', iter=args.routings,
279             alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
280             Psi0=torch.eye(self.PP), nu0=self.PP+1)
281
282         self.ConvCaps_2 = layers.ConvCapsules2d(in_caps=self.C,
out_caps=self.D,
283             kernel_size=3, stride=1, pose_dim=self.P)
284
285         self.ConvRouting_2 = vb_routing.VariationalBayesRouting2d(
in_caps=self.C, out_caps=self.D,
286             kernel_size=3, stride=1, pose_dim=self.P,
287             cov='diag', iter=args.routings,
288             alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
289             Psi0=torch.eye(self.PP), nu0=self.PP+1)
290
291         self.ClassCaps = layers.ConvCapsules2d(in_caps=self.D,
out_caps=self.n_classes,
292             kernel_size=1, stride=1, pose_dim=self.P, share_W_ij=
True, coor_add=True)
293
294         self.ClassRouting = vb_routing.VariationalBayesRouting2d(
in_caps=self.D, out_caps=self.n_classes,
295             kernel_size=12, stride=1, pose_dim=self.P, # adjust
final kernel_size K depending on input H/W, for H=W=32, K=4.
296             cov='diag', iter=args.routings,
297             alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
298             Psi0=torch.eye(self.PP), nu0=self.PP+1, class_caps=True
)
299
300         # This will send net to device, so only call here
301         self.init_device()
302
303     def forward(self, x):
304         # Out      [?, A, F, F]
305         x = F.relu(self.BN_1(self.Conv_1(x)))
306         # Out      a [?, B, F, F], v [?, B, P, P, F, F]
307         a,v = self.PrimaryCaps(x)
308         # Out      a [?, B, 1, 1, 1, F, F, K, K], v [?, B, C, P*P,
1, F, F, K, K]
309         a,v = self.ConvCaps_1(a, v, self.device)
310         # Out      a [?, C, F, F], v [?, C, P, P, F, F]
311         a,v = self.ConvRouting_1(a, v, self.device)
312         # Out      a [?, C, 1, 1, 1, F, F, K, K], v [?, C, D, P*P,
1, F, F, K, K]
313         a,v = self.ConvCaps_2(a, v, self.device)

```

```

314     # Out      a [?, D, F, F], v [?, D, P, P, F, F]
315     a,v = self.ConvRouting_2(a, v, self.device)
316     # Out      a [?, D, 1, 1, 1, F, F, K, K], v [?, D, n_classes
, P*P, 1, F, F, K, K]
317     a,v = self.ClassCaps(a, v, self.device)
318     # Out      yhat [?, n_classes], v [?, n_classes, P, P]
319     yhat, v = self.ClassRouting(a, v, self.device)
320     return yhat
321
322 class SRCapsules(ImageWiseModels):
323     """
324     Self Routing Capsules based on https://github.com/coder3000/SR-
CapsNet
325     """
326     def __init__(self, args, patchwise=None, original_architecture=
False):
327         super(SRCapsules, self).__init__(args, patchwise,
original_architecture)
328
329         planes = 16
330         last_size = 14
331         self.num_caps = 16
332
333         self.conv1 = nn.Conv2d(args.input_size[0], 256, kernel_size
=7, stride=2, padding=1, bias=False)
334         self.bn1 = nn.BatchNorm2d(256)
335         self.conv_a = nn.Conv2d(256, self.num_caps, kernel_size=5,
stride=1, padding=1, bias=False)
336         self.conv_pose = nn.Conv2d(256, self.num_caps*planes,
kernel_size=5, stride=1, padding=1, bias=False)
337         self.bn_a = nn.BatchNorm2d(self.num_caps)
338         self.bn_pose = nn.BatchNorm2d(self.num_caps*planes)
339
340         self.conv_caps = SelfRouting2d(self.num_caps, self.num_caps
, planes, planes, kernel_size=3, stride=2, padding=1, pose_out=
True)
341         self.bn_pose_conv_caps = nn.BatchNorm2d(self.num_caps*
planes)
342
343         self.fc_caps = SelfRouting2d(self.num_caps, args.classes,
planes, 1, kernel_size=last_size, padding=0, pose_out=False)
344
345         # This will send net to device, so only call here
346         self.init_device()
347
348         self.loss = nn.NLLLoss()
349         self.loss.to(self.device)
350
351     def forward(self, x):
352         out = F.relu(self.bn1(self.conv1(x)))
353         a, pose = self.conv_a(out), self.conv_pose(out)
354         a, pose = torch.sigmoid(self.bn_a(a)), self.bn_pose(pose)
355
356         a, pose = self.conv_caps(a, pose)
357         pose = self.bn_pose_conv_caps(pose)
358
359         a, _ = self.fc_caps(a, pose)

```

```

360         out = a.view(a.size(0), -1)
361         out = out.log()
362         return out
363
364     def propagate(self, inputs, labels, criterion=None):
365         inputs = self.patch_wise_model.features(inputs)
366         y_pred = self(inputs)
367         if criterion:
368             loss = self.loss(y_pred, labels)
369         else:
370             loss = 0
371         return loss, y_pred
372

```

Listing 5: src/imagewisemodels.py

```

1  # Base model
2  from .model import Model
3
4  # Variational Capsules
5  from .VarCaps import layers
6  from .VarCaps import vb_routing
7
8  # EfficientNet
9  from efficientnet_pytorch import EfficientNet
10
11 # Training
12 import torch.nn.functional as F
13 import torch.nn as nn
14 import numpy as np
15 import torch
16 import time
17
18 # For testing we want to get a whole image in patches
19 BATCH_SIZE = 12
20
21 class VariationalMixedCapsules(Model):
22     """
23     Capsule Routing via Variational Bayes based on https://github.com/fabio-deep/Variational-Capsule-Routing
24     """
25
26     def __init__(self, args):
27         super(VariationalMixedCapsules, self).__init__(args)
28         self.output_size = args.output_size
29         self.n_classes = args.classes
30         self.routing = args.routing
31
32         self.P = args.pose_dim
33         self.PP = int(np.max([2, self.P*self.P]))
34         self.A, self.B, self.C, self.D = args.arch
35         K = 12
36
37         self.features = nn.Sequential(
38             # Block 1
39             nn.Conv2d(in_channels=args.input_size[0], out_channels
40                       =16, kernel_size=3, stride=1, padding=1),
41             nn.BatchNorm2d(16),

```

```

41         nn.ReLU(inplace=True),
42         nn.Conv2d(in_channels=16, out_channels=16, kernel_size
=3, stride=1, padding=1),
43         nn.BatchNorm2d(16),
44         nn.ReLU(inplace=True),
45         nn.Conv2d(in_channels=16, out_channels=16, kernel_size
=2, stride=2),
46         nn.BatchNorm2d(16),
47         nn.ReLU(inplace=True),
48
49         # Block 2
50         nn.Conv2d(in_channels=16, out_channels=32, kernel_size
=3, stride=1, padding=1),
51         nn.BatchNorm2d(32),
52         nn.ReLU(inplace=True),
53         nn.Conv2d(in_channels=32, out_channels=32, kernel_size
=3, stride=1, padding=1),
54         nn.BatchNorm2d(32),
55         nn.ReLU(inplace=True),
56         nn.Conv2d(in_channels=32, out_channels=32, kernel_size
=2, stride=2),
57         nn.BatchNorm2d(32),
58         nn.ReLU(inplace=True),
59
60         # Block 3
61         nn.Conv2d(in_channels=32, out_channels=64, kernel_size
=3, stride=1, padding=1),
62         nn.BatchNorm2d(64),
63         nn.ReLU(inplace=True),
64         nn.Conv2d(in_channels=64, out_channels=64, kernel_size
=3, stride=1, padding=1),
65         nn.BatchNorm2d(64),
66         nn.ReLU(inplace=True),
67         nn.Conv2d(in_channels=64, out_channels=64, kernel_size
=2, stride=2),
68         nn.BatchNorm2d(64),
69         nn.ReLU(inplace=True),
70
71         nn.Conv2d(in_channels=64, out_channels=args.output_size
[0], kernel_size=1, stride=1),
72     )
73
74     # Layer 1: Just a conventional Conv2D layer
75     self.Conv_1 = nn.Conv2d(args.input_size[0], self.A,
kernel_size=5, stride=2, bias=False)
76     nn.init.kaiming_uniform_(self.Conv_1.weight)
77
78     self.Conv_2 = nn.Conv2d(self.A, self.A, kernel_size=5,
stride=2, bias=False)
79     nn.init.kaiming_uniform_(self.Conv_2.weight)
80
81     self.Conv_3 = nn.Conv2d(self.A, self.A, kernel_size=3,
stride=2, bias=False)
82     nn.init.kaiming_uniform_(self.Conv_3.weight)
83
84     self.BN_1 = nn.BatchNorm2d(self.A)
85     self.PrimaryCaps = layers.PrimaryCapsules2d(in_channels=

```

```

86     self.A, out_caps=self.B,
87         kernel_size=1, stride=1, pose_dim=self.P)
88
89     self.ConvCaps_1 = layers.ConvCapsules2d(in_caps=self.B,
90 out_caps=self.C,
91         kernel_size=3, stride=2, pose_dim=self.P)
92
93     self.ConvRouting_1 = vb_routing.VariationalBayesRouting2d(
94 in_caps=self.B, out_caps=self.C,
95         kernel_size=3, stride=2, pose_dim=self.P,
96         cov='diag', iter=args.routings,
97         alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
98         Psi0=torch.eye(self.PP), nu0=self.PP+1)
99
100     self.ConvCaps_2 = layers.ConvCapsules2d(in_caps=self.C,
101 out_caps=self.D,
102         kernel_size=3, stride=1, pose_dim=self.P)
103
104     self.ConvRouting_2 = vb_routing.VariationalBayesRouting2d(
105 in_caps=self.C, out_caps=self.D,
106         kernel_size=3, stride=1, pose_dim=self.P,
107         cov='diag', iter=args.routings,
108         alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
109         Psi0=torch.eye(self.PP), nu0=self.PP+1)
110
111     self.ClassCaps = layers.ConvCapsules2d(in_caps=self.D,
112 out_caps=self.n_classes,
113         kernel_size=1, stride=1, pose_dim=self.P, share_W_ij=
114 True, coor_add=True)
115
116     self.ClassRouting = vb_routing.VariationalBayesRouting2d(
117 in_caps=self.D, out_caps=self.n_classes,
118         kernel_size=K, stride=1, pose_dim=self.P, # adjust
119         final kernel_size K depending on input H/W, for H=W=32, K=4.
120         cov='diag', iter=args.routings,
121         alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
122         Psi0=torch.eye(self.PP), nu0=self.PP+1, class_caps=True
123 )
124
125     # This will send net to device, so only call here
126     self.init_device()
127
128     def forward(self, x):
129         x = self.features(x)
130         # Out      [?, A, F, F]
131         x = F.relu(self.BN_1(self.Conv_1(x)))
132         # Out      a [?, B, F, F], v [?, B, P, P, F, F]
133         a,v = self.PrimaryCaps(x)
134         # Out      a [?, B, 1, 1, 1, F, F, K, K], v [?, B, C, P*P,
135 1, F, F, K, K]
136         a,v = self.ConvCaps_1(a, v, self.device)
137         # Out      a [?, C, F, F], v [?, C, P, P, F, F]
138         a,v = self.ConvRouting_1(a, v, self.device)
139         # Out      a [?, C, 1, 1, 1, F, F, K, K], v [?, C, D, P*P,
140 1, F, F, K, K]
141         a,v = self.ConvCaps_2(a, v, self.device)
142         # Out      a [?, D, F, F], v [?, D, P, P, F, F]

```

```

131     a,v = self.ConvRouting_2(a, v, self.device)
132     # Out      a [?, D, 1, 1, 1, F, F, K, K], v [?, D, n_classes
    , P*P, 1, F, F, K, K]
133     a,v = self.ClassCaps(a, v, self.device)
134     # Out      yhat [?, n_classes], v [?, n_classes, P, P]
135     yhat, v = self.ClassRouting(a, v, self.device)
136     return yhat
137
138 class EffNet(Model):
139     """
140     EfficientNet based on the implementation https://github.com/
    lukemelas/EfficientNet-PyTorch
141     """
142
143     def __init__(self, args):
144         super(EffNet, self).__init__(args)
145
146         self.time = str(time.strftime('%Y-%m-%d_%H-%M'))
147         self.device = torch.device("cuda:0" if torch.cuda.
    is_available() else "cpu")
148         self.output_size = args.output_size
149         self.n_classes = args.classes
150
151         self.model = EfficientNet.from_pretrained('efficientnet-b0'
    )
152         num_fters = self.model._fc.in_features
153         self.model._fc = nn.Sequential(
154             nn.Linear(num_fters, 128),
155             nn.ReLU(),
156             nn.Linear(128, self.n_classes)
157         )
158
159         for param in self.model.parameters():
160             param.requires_grad = False
161
162         for param in self.model._fc.parameters():
163             param.requires_grad = True
164
165         # This will send net to device, so only call here
166         self.init_device()
167
168     def propagate(self, inputs, labels, criterion=None):
169         outputs = self.model(inputs)
170         if criterion:
171             loss = criterion(outputs, labels)
172         else:
173             loss = 0
174         return loss, outputs

```

Listing 6: src/mixedmodels.py

```

1 """
2 Author: Xifeng Guo, E-mail: 'guoxifeng1990@163.com', Github: 'https
    ://github.com/XifengGuo/CapsNet-Pytorch'
3 """
4
5 import torch
6 import torch.nn as nn

```

```

7 import torch.nn.functional as F
8 from torch.autograd import Variable
9
10
11 def squash(inputs, axis=-1):
12     """
13     The non-linear activation used in Capsule. It drives the length
14     of a large vector to near 1 and small vector to 0
15     :param inputs: vectors to be squashed
16     :param axis: the axis to squash
17     :return: a Tensor with same size as inputs
18     """
19     norm = torch.norm(inputs, p=2, dim=axis, keepdim=True)
20     scale = norm**2 / (1 + norm**2) / (norm + 1e-8)
21     return scale * inputs
22
23 class DenseCapsule(nn.Module):
24     """
25     The dense capsule layer. It is similar to Dense (FC) layer.
26     Dense layer has 'in_num' inputs, each is a scalar, the
27     output of the neuron from the former layer, and it has 'out_num'
28     output neurons. DenseCapsule just expands the
29     output of the neuron from scalar to vector. So its input size = \
30     [None, in_num_caps, in_dim_caps] and output size = \
31     [None, out_num_caps, out_dim_caps]. For Dense Layer,
32     in_dim_caps = out_dim_caps = 1.
33
34     :param in_num_caps: number of capsules inputted to this layer
35     :param in_dim_caps: dimension of input capsules
36     :param out_num_caps: number of capsules outputted from this
37     layer
38     :param out_dim_caps: dimension of output capsules
39     :param routings: number of iterations for the routing algorithm
40     """
41     def __init__(self, in_num_caps, in_dim_caps, out_num_caps,
42                  out_dim_caps, device, routings=3):
43         super(DenseCapsule, self).__init__()
44         self.in_num_caps = in_num_caps
45         self.in_dim_caps = in_dim_caps
46         self.out_num_caps = out_num_caps
47         self.out_dim_caps = out_dim_caps
48         self.routings = routings
49         self.device = device
50         self.weight = nn.Parameter(0.01 * torch.randn(out_num_caps,
51                                                         in_num_caps, out_dim_caps, in_dim_caps))
52
53     def forward(self, x):
54         # x.size=[batch, in_num_caps, in_dim_caps]
55         # expanded to [batch, 1, in_num_caps,
56         in_dim_caps, 1]
57         # weight.size = [out_num_caps, in_num_caps,
58         out_dim_caps, in_dim_caps]
59         # torch.matmul: [out_dim_caps, in_dim_caps] x [in_dim_caps,
60         1] -> [out_dim_caps, 1]
61         # => x_hat.size =[batch, out_num_caps, in_num_caps,
62         out_dim_caps]

```

```

52     x_hat = torch.squeeze(torch.matmul(self.weight, x[:, None,
53                                     :, :, None]), dim=-1)
54
55     # In forward pass, 'x_hat_detached' = 'x_hat';
56     # In backward, no gradient can flow from 'x_hat_detached'
57     back to 'x_hat'.
58     x_hat_detached = x_hat.detach()
59
60     # The prior for coupling coefficient, initialized as zeros.
61     # b.size = [batch, out_num_caps, in_num_caps]
62     b = Variable(torch.zeros(x.size(0), self.out_num_caps, self
63                             .in_num_caps)).to(self.device)
64
65     assert self.routings > 0, 'The \'routings\' should be > 0.'
66     for i in range(self.routings):
67         # c.size = [batch, out_num_caps, in_num_caps]
68         c = F.softmax(b, dim=1)
69
70         # At last iteration, use 'x_hat' to compute 'outputs'
71         in order to backpropagate gradient
72         if i == self.routings - 1:
73             # c.size expanded to [batch, out_num_caps,
74             in_num_caps, 1
75             ]
76             # x_hat.size = [batch, out_num_caps,
77             in_num_caps, out_dim_caps]
78             # => outputs.size= [batch, out_num_caps, 1,
79             out_dim_caps]
80             outputs = squash(torch.sum(c[:, :, :, None] * x_hat
81                                     , dim=-2, keepdim=True))
82             # outputs = squash(torch.matmul(c[:, :, None, :],
83             x_hat)) # alternative way
84         else: # Otherwise, use 'x_hat_detached' to update 'b'.
85             No gradients flow on this path.
86             outputs = squash(torch.sum(c[:, :, :, None] *
87                                     x_hat_detached, dim=-2, keepdim=True))
88             # outputs = squash(torch.matmul(c[:, :, None, :],
89             x_hat_detached)) # alternative way
90
91             # outputs.size = [batch, out_num_caps, 1,
92             out_dim_caps]
93             # x_hat_detached.size=[batch, out_num_caps,
94             in_num_caps, out_dim_caps]
95             # => b.size = [batch, out_num_caps,
96             in_num_caps]
97             b = b + torch.sum(outputs * x_hat_detached, dim=-1)
98
99     return torch.squeeze(outputs, dim=-2)
100
101 class PrimaryCapsule(nn.Module):
102     """
103     Apply Conv2D with 'out_channels' and then reshape to get
104     capsules
105     :param in_channels: input channels
106     :param out_channels: output channels
107     :param dim_caps: dimension of capsule
108     :param kernel_size: kernel size

```



```

93 :return: output tensor, size=[batch, num_caps, dim_caps]
94 """
95 def __init__(self, in_channels, out_channels, dim_caps,
96               kernel_size, stride=1, padding=0):
97     super(PrimaryCapsule, self).__init__()
98     self.dim_caps = dim_caps
99     self.conv2d = nn.Conv2d(in_channels, out_channels,
100                             kernel_size=kernel_size, stride=stride, padding=padding)
101
102 def forward(self, x):
103     outputs = self.conv2d(x)
104     outputs = outputs.view(x.size(0), -1, self.dim_caps)
105     return squash(outputs)

```

Listing 7: src/DynamicCaps/capsulelayers.py

```

1  """
2  Author: Xifeng Guo, E-mail: 'guoxifeng1990@163.com', Github: 'https
3  ://github.com/XifengGuo/CapsNet-Pytorch'
4  """
5  import torch
6  from torch import nn
7
8  def caps_loss(y_true, y_pred, x, x_recon, lam_recon):
9      """
10      Capsule loss = Margin loss + lam_recon * reconstruction loss.
11      :param y_true: true labels, one-hot coding, size=[batch,
12      classes]
13      :param y_pred: predicted labels by CapsNet, size=[batch,
14      classes]
15      :param x: input data, size=[batch, channels, width, height]
16      :param x_recon: reconstructed data, size is same as 'x'
17      :param lam_recon: coefficient for reconstruction loss
18      :return: Variable contains a scalar loss value.
19      """
20      L = y_true * torch.clamp(0.9 - y_pred, min=0.) ** 2 + \
21          0.5 * (1 - y_true) * torch.clamp(y_pred - 0.1, min=0.) ** 2
22      L_margin = L.sum(dim=1).mean()
23
24      L_recon = nn.MSELoss()(x_recon, x)
25
26      return L_margin + lam_recon * L_recon

```

Listing 8: src/DynamicCaps/capsulenet.py

```

1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  import math
6
7  eps = 1e-12
8
9  class SelfRouting2d(nn.Module):
10     def __init__(self, A, B, C, D, kernel_size=3, stride=1, padding
11     =1, pose_out=False):
12         super(SelfRouting2d, self).__init__()

```

```

12     self.A = A
13     self.B = B
14     self.C = C
15     self.D = D
16
17     self.k = kernel_size
18     self.kk = kernel_size ** 2
19     self.kkA = self.kk * A
20
21     self.stride = stride
22     self.pad = padding
23
24     self.pose_out = pose_out
25
26     if pose_out:
27         self.W1 = nn.Parameter(torch.FloatTensor(self.kkA, B*D,
28 C))
29         nn.init.kaiming_uniform_(self.W1.data)
30
31         self.W2 = nn.Parameter(torch.FloatTensor(self.kkA, B, C))
32         self.b2 = nn.Parameter(torch.FloatTensor(1, 1, self.kkA, B)
33 )
34
35         nn.init.constant_(self.W2.data, 0)
36         nn.init.constant_(self.b2.data, 0)
37
38     def forward(self, a, pose):
39         # a: [b, A, h, w]
40         # pose: [b, AC, h, w]
41         b, _, h, w = a.shape
42
43         # [b, ACkk, 1]
44         pose = F.unfold(pose, self.k, stride=self.stride, padding=
45 self.pad)
46         l = pose.shape[-1]
47         # [b, A, C, kk, 1]
48         pose = pose.view(b, self.A, self.C, self.kk, 1)
49         # [b, 1, kk, A, C]
50         pose = pose.permute(0, 4, 3, 1, 2).contiguous()
51         # [b, 1, kkA, C, 1]
52         pose = pose.view(b, 1, self.kkA, self.C, 1)
53
54         if hasattr(self, 'W1'):
55             # [b, 1, kkA, BD]
56             pose_out = torch.matmul(self.W1, pose).squeeze(-1)
57             # [b, 1, kkA, B, D]
58             pose_out = pose_out.view(b, 1, self.kkA, self.B, self.D
59 )
60
61         # [b, 1, kkA, B]
62         logit = torch.matmul(self.W2, pose).squeeze(-1) + self.b2
63
64         # [b, 1, kkA, B]
65         r = torch.softmax(logit, dim=3)
66
67         # [b, kkA, 1]
68         a = F.unfold(a, self.k, stride=self.stride, padding=self.

```

```

pad)
65     # [b, A, kk, 1]
66     a = a.view(b, self.A, self.kk, 1)
67     # [b, 1, kk, A]
68     a = a.permute(0, 3, 2, 1).contiguous()
69     # [b, 1, kkA, 1]
70     a = a.view(b, 1, self.kkA, 1)
71
72     # [b, 1, kkA, B]
73     ar = a * r
74     # [b, 1, 1, B]
75     ar_sum = ar.sum(dim=2, keepdim=True)
76     # [b, 1, kkA, B, 1]
77     coeff = (ar / (ar_sum)).unsqueeze(-1)
78
79     # [b, 1, B]
80     # a_out = ar_sum.squeeze(2)
81     a_out = ar_sum / a.sum(dim=2, keepdim=True)
82     a_out = a_out.squeeze(2)
83
84     # [b, B, 1]
85     a_out = a_out.transpose(1,2)
86
87     if hasattr(self, 'W1'):
88         # [b, 1, B, D]
89         pose_out = (coeff * pose_out).sum(dim=2)
90         # [b, 1, BD]
91         pose_out = pose_out.view(b, 1, -1)
92         # [b, BD, 1]
93         pose_out = pose_out.transpose(1,2)
94
95     oh = ow = math.floor(1*(1/2))
96
97     a_out = a_out.view(b, -1, oh, ow)
98     if hasattr(self, 'W1'):
99         pose_out = pose_out.view(b, -1, oh, ow)
100     else:
101         pose_out = None
102
103     return a_out, pose_out

```

Listing 9: src/SRCaps/modules.py

```

1 import torch
2 import numpy as np
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 class PrimaryCapsules2d(nn.Module):
7     '''Primary Capsule Layer'''
8     def __init__(self, in_channels, out_caps, kernel_size, stride,
9         padding=0, pose_dim=4, weight_init='xavier_uniform'):
10         super().__init__()
11
12         self.A = in_channels
13         self.B = out_caps
14         self.P = pose_dim
15         self.K = kernel_size

```

```

16         self.S = stride
17         self.padding = padding
18
19         w_kernel = torch.empty(self.B*self.P*self.P, self.A, self.K
20 , self.K)
21         a_kernel = torch.empty(self.B, self.A, self.K, self.K)
22
23         if weight_init == 'kaiming_normal':
24             nn.init.kaiming_normal_(w_kernel)
25             nn.init.kaiming_normal_(a_kernel)
26         elif weight_init == 'kaiming_uniform':
27             nn.init.kaiming_uniform_(w_kernel)
28             nn.init.kaiming_uniform_(a_kernel)
29         elif weight_init == 'xavier_normal':
30             nn.init.xavier_normal_(w_kernel)
31             nn.init.xavier_normal_(a_kernel)
32         elif weight_init == 'xavier_uniform':
33             nn.init.xavier_uniform_(w_kernel)
34             nn.init.xavier_uniform_(a_kernel)
35         else:
36             NotImplementedError('{} not implemented.'.format(
37 weight_init))
38
39         # Out      [B*(P*P+1), A, K, K]
40         self.weight = nn.Parameter(torch.cat([w_kernel, a_kernel],
41 dim=0))
42
43         self.BN_a = nn.BatchNorm2d(self.B, affine=True)
44         self.BN_p = nn.BatchNorm3d(self.B, affine=True)
45
46         def forward(self, x): # [?, A, F, F]      In
47
48             # Out      [?, B*(P*P+1), F, F]
49             x = F.conv2d(x, weight=self.weight, stride=self.S, padding=
50 self.padding)
51
52             # Out      ([?, B*P*P, F, F], [?, B, F, F])      [?, B*(P*P
53 +1), F, F]
54             poses, activations = torch.split(x, [self.B*self.P*self.P,
55 self.B], dim=1)
56
57             # Out      [?, B, P*P, F, F]
58             poses = self.BN_p(poses.reshape(-1, self.B, self.P*self.P,
59 *x.shape[2:]))
60
61             # Out      [?, B, P, P, F, F]      [?, B, P*P, F, F]      In
62             poses = poses.reshape(-1, self.B, self.P, self.P, *x.shape
63 [2:])
64
65             # Out      [?, B, F, F]
66             activations = torch.sigmoid(self.BN_a(activations))
67
68             return (activations, poses)
69
70 class ConvCapsules2d(nn.Module):
71     '''Convolutional Capsule Layer'''
72     def __init__(self, in_caps, out_caps, pose_dim, kernel_size,

```

```

stride, padding=0,
65     weight_init='xavier_uniform', share_W_ij=False, coor_add=
False):
66     super().__init__()
67
68     self.B = in_caps
69     self.C = out_caps
70     self.P = pose_dim
71     self.PP = np.max([2, self.P*self.P])
72     self.K = kernel_size
73     self.S = stride
74     self.padding = padding
75
76     self.share_W_ij = share_W_ij # share the transformation
matrices across (F*F)
77     self.coor_add = coor_add # embed coordinates
78
79     # Out      [1, B, C, 1, P, P, 1, 1, K, K]
80     self.W_ij = torch.empty(1, self.B, self.C, 1, self.P, self.
P, 1, 1, self.K, self.K)
81
82     if weight_init.split('_')[0] == 'xavier':
83         fan_in = self.B * self.K*self.K * self.PP # in_caps
types * receptive field size
84         fan_out = self.C * self.K*self.K * self.PP # out_caps
types * receptive field size
85         std = np.sqrt(2. / (fan_in + fan_out))
86         bound = np.sqrt(3.) * std
87
88         if weight_init.split('_')[1] == 'normal':
89             self.W_ij = nn.Parameter(self.W_ij.normal_(0, std))
90         elif weight_init.split('_')[1] == 'uniform':
91             self.W_ij = nn.Parameter(self.W_ij.uniform_(-bound,
bound))
92     else:
93         raise NotImplementedError('{ } not implemented.'.
format(weight_init))
94
95     elif weight_init.split('_')[0] == 'kaiming':
96         # fan_in preserves magnitude of the variance of the
weights in the forward pass.
97         fan_in = self.B * self.K*self.K * self.PP # in_caps
types * receptive field size
98         # fan_out has same affect as fan_in for backward pass.
99         fan_out = self.C * self.K*self.K * self.PP # out_caps
types * receptive field size
100        std = np.sqrt(2.) / np.sqrt(fan_in)
101        bound = np.sqrt(3.) * std
102
103        if weight_init.split('_')[1] == 'normal':
104            self.W_ij = nn.Parameter(self.W_ij.normal_(0, std))
105        elif weight_init.split('_')[1] == 'uniform':
106            self.W_ij = nn.Parameter(self.W_ij.uniform_(-bound,
bound))
107    else:
108        raise NotImplementedError('{ } not implemented.'.
format(weight_init))

```

```

109
110         elif weight_init == 'noisy_identity' and self.PP > 2:
111             b = 0.01 # U(0,b)
112             # Out      [1, B, C, 1, P, P, 1, 1, K, K]
113             self.W_ij = nn.Parameter(torch.clamp(.1*torch.eye(self.
114 P,self.P).repeat( \
115             1, self.B, self.C, 1, 1, 1, self.K, self.K, 1, 1) +
116 \
117             torch.empty(1, self.B, self.C, 1, 1, 1, self.K,
118 self.K, self.P, self.P).uniform_(0,b), \
119             max=1).permute(0, 1, 2, 3, -2, -1, 4, 5, 6, 7))
120         else:
121             raise NotImplementedError('{} not implemented.'.format(
122 weight_init))
123
124         if self.padding != 0:
125             if isinstance(self.padding, int):
126                 self.padding = [self.padding]*4
127
128     def forward(self, activations, poses, device): # ([?, B, F, F],
129             [?, B, P, P, F, F])      In
130
131         if self.padding != 0:
132             activations = F.pad(activations, self.padding) #
133             [1,1,1,1]
134             poses = F.pad(poses, self.padding + [0]*4) #
135             [0,0,1,1,1,1]
136
137         if self.share_W_ij: # share the matrices over (F*F), if
138             class caps layer
139             self.K = poses.shape[-1] # out_caps (C) feature map
140             size
141
142             self.F = (poses.shape[-1] - self.K) // self.S + 1 #
143             featuremap size
144
145             # Out      [?, B, P, P, F', F', K, K]      [?, B, P, P, F, F]
146             poses = poses.unfold(4, size=self.K, step=self.S).unfold(5,
147 size=self.K, step=self.S)
148
149             # Out      [?, B, 1, P, P, 1, F', F', K, K]      [?, B, P, P,
150 F', F', K, K]
151             poses = poses.unsqueeze(2).unsqueeze(5)
152
153             # Out      [?, B, F', F', K, K]      [?, B, F, F]
154             activations = activations.unfold(2, size=self.K, step=self.
155 S).unfold(3, size=self.K, step=self.S)
156
157             # Out      [?, B, 1, 1, 1, F', F', K, K]      [?, B, F', F',
158 K, K]
159             activations = activations.reshape(-1, self.B, 1, 1, 1, *
160 activations.shape[2:4], self.K, self.K)
161
162             # Out      [?, B, C, P, P, F', F', K, K]      ([?, B, 1, P, P
163 , 1, F', F', K, K] * [1, B, C, 1, P, P, 1, 1, K, K])
164             V_ji = (poses * self.W_ij).sum(dim=4) # matmul equiv.
165
166

```

```

150     # Out      [?, B, C, P*P, 1, F', F', K, K]      [?, B, C, P,
      P, F', F', K, K]
151     V_ji = V_ji.reshape(-1, self.B, self.C, self.P*self.P, 1, *
      V_ji.shape[-4:-2], self.K, self.K)
152
153     if self.coor_add:
154         if V_ji.shape[-1] == 1: # if class caps layer (
      featuremap size = 1)
155             self.F = self.K # 1->4
156
157             # coordinates = torch.arange(self.F, dtype=torch.
      float32) / self.F
158             coordinates = torch.arange(self.F, dtype=torch.float32)
      .add(1.) / (self.F*10)
159             i_vals = torch.zeros(self.P*self.P, self.F, 1).to(device)
160             j_vals = torch.zeros(self.P*self.P, 1, self.F).to(device)
161             i_vals[self.P-1, :, 0] = coordinates
162             j_vals[2*self.P-1, 0, :] = coordinates
163
164             if V_ji.shape[-1] == 1: # if class caps layer
165                 # Out      [?, B, C, P*P, 1, 1, 1, K=F, K=F] (class
      caps)
166                 V_ji = V_ji + (i_vals + j_vals).reshape(1,1,1,self.
      P*self.P,1,1,1,self.F,self.F)
167                 return activations, V_ji
168
169             # Out      [?, B, C, P*P, 1, F, F, K, K]
170             V_ji = V_ji + (i_vals + j_vals).reshape(1,1,1,self.P*
      self.P,1,self.F,self.F,1,1)
171
172     return activations, V_ji

```

Listing 10: src/VarCaps/layers.py

```

1 import torch
2 import numpy as np
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 class VariationalBayesRouting2d(nn.Module):
7     '''Variational Bayes Capsule Routing Layer'''
8     def __init__(self, in_caps, out_caps, pose_dim,
9         kernel_size, stride,
10         alpha0, # Dirichlet
11         m0, kappa0, # Gaussian
12         Psi0, nu0, # Wishart
13         cov='diag', iter=3, class_caps=False):
14         super().__init__()
15
16         self.B = in_caps
17         self.C = out_caps
18         self.P = pose_dim
19         self.D = np.max([2, self.P*self.P])
20         self.K = kernel_size
21         self.S = stride
22
23         self.cov = cov # diag/full
24         self.iter = iter # routing iters

```

```

25     self.class_caps = class_caps
26     self.n_classes = out_caps if class_caps else None
27
28     # dirichlet prior parameter
29     self.alpha0 = torch.tensor(alpha0).type(torch.FloatTensor)
30     # self.alpha0 = nn.Parameter(torch.zeros(1,1,self.C
31     ,1,1,1,1,1).fill_(alpha0)) learn it by backprop
32
33     # Out      [?, 1, C, P*P, 1, 1, 1, 1, 1]
34     self.register_buffer('m0', m0.unsqueeze(0).repeat( \
35         self.C,1).reshape(1,1,self.C,self.D,1,1,1,1,1)) #
36     gaussian prior mean parameter
37
38     # precision scaling parameter of gaussian prior over
39     capsule component means
40     self.kappa0 = kappa0
41
42     # scale matrix of wishart prior over capsule precisions
43     if self.cov == 'diag':
44         # Out      [?, 1, C, P*P, 1, 1, 1, 1, 1]
45         self.register_buffer('Psi0', torch.diag(Psi0).unsqueeze
46         (0).repeat( \
47             self.C,1).reshape(1,1,self.C,self.D,1,1,1,1,1))
48
49     elif self.cov == 'full':
50         # Out      [?, 1, C, P*P, P*P, 1, 1, 1, 1]
51         self.register_buffer('Psi0', Psi0.unsqueeze(0).repeat( \
52             self.C,1,1).reshape(1,1,self.C,self.D,self.D
53             ,1,1,1,1))
54
55     # degree of freedom parameter of wishart prior capsule
56     precisions
57     self.nu0 = nu0
58
59     # log determinant = 0, if Psi0 is identity
60     self.register_buffer('lndet_Psi0', 2*torch.diagonal(torch.
61     cholesky(
62         Psi0)).log().sum())
63
64     # pre compute the argument of the digamma function in E[ln|
65     lambda_j|]
66     self.register_buffer('diga_arg', torch.arange(self.D).
67     reshape(
68         1,1,1,self.D,1,1,1,1,1).type(torch.FloatTensor))
69
70     # pre define some constants
71     self.register_buffer('Dlog2',
72         self.D*torch.log(torch.tensor(2.)).type(torch.
73         FloatTensor))
74     self.register_buffer('Dlog2pi',
75         self.D*torch.log(torch.tensor(2.*np.pi)).type(torch.
76         FloatTensor))
77
78     # Out      [K*K, 1, K, K] vote collecting filter
79     self.register_buffer('filter',
80         torch.eye(self.K*self.K).reshape(self.K*self.K,1,self.K

```



```

, self.K))
70
71     # Out      [1, 1, C, 1, 1, 1, 1, 1, 1] optional params
72     self.beta_u = nn.Parameter(torch.zeros(1, 1, self.C
, 1, 1, 1, 1, 1, 1))
73     self.beta_a = nn.Parameter(torch.zeros(1, 1, self.C
, 1, 1, 1, 1, 1, 1))
74
75     self.BN_v = nn.BatchNorm3d(self.C, affine=False)
76     self.BN_a = nn.BatchNorm2d(self.C, affine=False)
77
78     # Out      [?, B, 1, 1, 1, F, F, K, K], [?, B, C, P*P, 1, F, F,
K, K]      In
79     def forward(self, a_i, V_ji, device):
80
81         self.F_i = a_i.shape[-2:] # input capsule (B) votes feature
map size (K)
82         self.F_o = a_i.shape[-4:-2] # output capsule (C) feature
map size (F)
83         self.N = self.B*self.F_i[0]*self.F_i[1] # total num of
lower level capsules
84
85         # Out      [1, B, C, 1, 1, 1, 1, 1, 1]
86         R_ij = (1./self.C) * torch.ones(1, self.B, self.C
, 1, 1, 1, 1, 1, 1, requires_grad=False).to(device)
87
88         for i in range(self.iter): # routing iters
89
90             # update capsule parameter distributions
91             self.update_qparam(a_i, V_ji, R_ij)
92
93             if i != self.iter-1: # skip last iter
94                 # update latent variable distributions (child to
parent capsule assignments)
95                 R_ij = self.update_qlatent(a_i, V_ji)
96
97             # Out      [?, 1, C, 1, 1, F, F, 1, 1]
98             self.Elnlambda_j = self.reduce_poses(
torch.digamma(.5*(self.nu_j - self.diga_arg))) \
99                 + self.Dlog2 + self.lndet_Psi_j
100
101             # Out      [?, 1, C, 1, 1, F, F, 1, 1]
102             self.Elnpi_j = torch.digamma(self.alpha_j) \
103                 - torch.digamma(self.alpha_j.sum(dim=2, keepdim=True))
104
105             # subtract "- .5*ln|lmbda|" due to precision matrix,
instead of adding "+ .5*ln|sigma|" for covariance matrix
106             H_q_j = .5*self.D * torch.log(torch.tensor(2*np.pi*np.e)) -
.5*self.Elnlambda_j # posterior entropy H[q*(mu_j, sigma_j)]
107
108             # Out      [?, 1, C, 1, 1, F, F, 1, 1] weighted negative
entropy with optional beta params and R_j weight
109             a_j = self.beta_a - (torch.exp(self.Elnpi_j) * H_q_j + self
.beta_u) ## self.R_j
110
111             # Out      [?, C, F, F]
112             a_j = a_j.squeeze()
113

```

```

114
115     # Out      [?, C, P*P, F, F]      [?, 1, C, P*P, 1, F, F, 1,
116     1]
117     self.m_j = self.m_j.squeeze()
118
119     # so BN works in the classcaps layer
120     if self.class_caps:
121         # Out      [?, C, 1, 1]      [?, C]
122         a_j = a_j[...,None,None]
123
124         # Out      [?, C, P*P, 1, 1]      [?, C, P*P]
125         self.m_j = self.m_j[...,None,None]
126     # else:
127     #     self.m_j = self.BN_v(self.m_j)
128
129     # Out      [?, C, P*P, F, F]
130     self.m_j = self.BN_v(self.m_j) # use 'else' above to
131     deactivate BN_v for class_caps
132
133     # Out      [?, C, P, P, F, F]      [?, C, P*P, F, F]
134     self.m_j = self.m_j.reshape(-1, self.C, self.P, self.P, *
135     self.F_o)
136
137     # Out      [?, C, F, F]
138     a_j = torch.sigmoid(self.BN_a(a_j))
139
140     return a_j.squeeze(), self.m_j.squeeze() # propagate
141     posterior means to next layer
142
143 def update_qparam(self, a_i, V_ji, R_ij):
144
145     # Out      [?, B, C, 1, 1, F, F, K, K]
146     R_ij = R_ij * a_i # broadcast a_i 1->C, and R_ij (1,1,1,1)
147     ->(F,F,K,K), 1->batch
148
149     # Out      [?, 1, C, 1, 1, F, F, 1, 1]
150     self.R_j = self.reduce_icaps(R_ij)
151
152     # Out      [?, 1, C, 1, 1, F, F, 1, 1]
153     self.alpha_j = self.alpha0 + self.R_j
154     # self.alpha_j = torch.exp(self.alpha0) + self.R_j # when
155     alpha's a param
156     self.kappa_j = self.kappa0 + self.R_j
157     self.nu_j = self.nu0 + self.R_j
158
159     # Out      [?, 1, C, P*P, 1, F, F, 1, 1]
160     mu_j = (1./self.R_j) * self.reduce_icaps(R_ij * V_ji)
161
162     # Out      [?, 1, C, P*P, 1, F, F, 1, 1]
163     # self.m_j = (1./self.kappa_j) * (self.R_j * mu_j + self.
164     kappa0 * self.m0) # use this if self.m0 != 0
165     self.m_j = (1./self.kappa_j) * (self.R_j * mu_j) # priors
166     removed for faster computation
167
168     if self.cov == 'diag':
169         # Out      [?, 1, C, P*P, 1, F, F, 1, 1] (1./R_j) not
170         needed because Psi_j calc

```

```

162         sigma_j = self.reduce_icaps(R_ij * (V_ji - mu_j).pow(2)
163     )
164
165     # Out      [?, 1, C, P*P, 1, F, F, 1, 1]
166     # self.invPsi_j = self.Psi0 + sigma_j + (self.kappa0*
167     self.R_j / self.kappa_j) \
168     #      * (mu_j - self.m0).pow(2) # use this if m0 != 0
169     or kappa0 != 1
170     self.invPsi_j = self.Psi0 + sigma_j + (self.R_j / self.
171     kappa_j) * (mu_j).pow(2) # priors removed for faster
172     computation
173
174     # Out      [?, 1, C, 1, 1, F, F, 1, 1] (-) sign as inv.
175     Psi_j
176     self.lndet_Psi_j = -self.reduce_poses(torch.log(self.
177     invPsi_j)) # log det of diag precision matrix
178
179     elif self.cov == 'full':
180         # Out      [?, B, C, P*P, P*P, F, F, K, K]
181         sigma_j = self.reduce_icaps(
182             R_ij * (V_ji - mu_j) * (V_ji - mu_j).transpose(3,4)
183         )
184
185         # Out      [?, 1, C, P*P, P*P, F, F, 1, 1] full cov,
186         torch.inverse(self.Psi0)
187         self.invPsi_j = self.Psi0 + sigma_j + (self.kappa0*self
188         .R_j / self.kappa_j) \
189         * (mu_j - self.m0) * (mu_j - self.m0).transpose
190         (3,4)
191
192         # Out      [?, 1, C, F, F, 1, 1 , P*P, P*P]
193         # needed for pytorch (*,n,n) dim requirements in .
194         cholesky and .inverse
195         self.invPsi_j = self.invPsi_j.permute
196         (0,1,2,5,6,7,8,3,4)
197
198         # Out      [?, 1, 1, 1, 1, C, F, F, 1, 1] (-) sign as inv.
199         Psi_j
200         self.lndet_Psi_j = -2*torch.diagonal(torch.cholesky(
201             self.invPsi_j), dim1=-2, dim2=-1).log().sum(-1,
202             keepdim=True)[...,None]
203
204     def update_qlatent(self, a_i, V_ji):
205
206         # Out      [?, 1, C, 1, 1, F, F, 1, 1]
207         self.Elnpi_j = torch.digamma(self.alpha_j) \
208             - torch.digamma(self.alpha_j.sum(dim=2, keepdim=True))
209
210         # Out      [?, 1, C, 1, 1, F, F, 1, 1] broadcasting diga_arg
211         self.Elnlambda_j = self.reduce_poses(
212             torch.digamma(.5*(self.nu_j - self.diga_arg))) \
213             + self.Dlog2 + self.lndet_Psi_j
214
215         if self.cov == 'diag':
216             # Out      [?, B, C, 1, 1, F, F, K, K]
217             ElnQ = (self.D/self.kappa_j) + self.nu_j \
218                 * self.reduce_poses((1./self.invPsi_j) * (V_ji -

```

```

204 self.m_j).pow(2))
205
206     elif self.cov == 'full':
207         # Out      [?, B, C, 1, 1, F, F, K, K]
208         Vm_j = V_ji - self.m_j
209         ElnQ = (self.D/self.kappa_j) + self.nu_j * self.
210         reduce_poses(
211             Vm_j.transpose(3,4) * torch.inverse(
212                 self.invPsi_j).permute(0,1,2,7,8,3,4,5,6) *
213             Vm_j)
214
215         # Out      [?, B, C, 1, 1, F, F, K, K]
216         lnp_j = .5*self.Elnlambda_j -.5*self.Dlog2pi -.5*ElnQ
217
218         # Out      [?*B, 1, F', F']      [?*B, K*K, F, F]      [?, B,
219         1, 1, 1, F, F, K, K]
220         sum_p_j = F.conv_transpose2d(
221             input=p_j.sum(dim=2, keepdim=True).reshape(
222                 -1, *self.F_o, self.K*self.K).permute(0, -1, 1, 2).
223             contiguous(),
224             weight=self.filter,
225             stride=[self.S, self.S])
226
227         # Out      [?*B, 1, F, F, K, K]
228         sum_p_j = sum_p_j.unfold(2, size=self.K, step=self.S).
229         unfold(3, size=self.K, step=self.S)
230
231         # Out      [?, B, 1, 1, 1, F, F, K, K]
232         sum_p_j = sum_p_j.reshape([-1, self.B, 1, 1, 1, *self.F_o,
233             self.K, self.K])
234
235         # Out      [?, B, C, 1, 1, F, F, K, K] # normalise over
236         out_caps_j
237         return 1. / torch.clamp(sum_p_j, min=1e-11) * p_j
238
239 def reduce_icaps(self, x):
240     return x.sum(dim=(1,-2,-1), keepdim=True)
241
242 def reduce_poses(self, x):
243     return x.sum(dim=(3,4), keepdim=True)

```

Listing 11: src/VarCaps/vb_routing.py