Code Listing

Marcell Veiner

May 2021

```
from src.datasets import set_seed
from src.patchwisemodel import PatchWiseModel
from src.imagewisemodels import BaseCNN, DynamicCapsules, NazeriCNN
    , VariationalCapsules, SRCapsules
from src.mixedmodels import VariationalMixedCapsules, EffNet
from argparse import Namespace

if __name__ == "__main__":
    set_seed()

    args_patch_wise = Namespace(
        batch_size=32,
        lr=0.001,
        epochs=100,
        augment=True,
        flip=False,
        workers=4,
        classes=4,
        input_size=[3, 512, 512],
        output_size=[3, 64, 64],
        predefined_stats=True,
        data_path="./data/ICIAR2018/patchwise_dataset",
        checkpoint_path="./models/Checkpoints/",
        name="_patchwise_"
    )

    args_img_wise = Namespace(
        lr=0.001,
        epochs=9,
        augment=True,
        flip=False,
        workers=4,
        classes=4,
        routings=3,
        lam_recon=0.392,
        pose_dim=4,
        batch_size=8,
        arch=[64,16,16,16],
        input_size=[3, 64, 64],
        output_size=[3, 64, 64],
        data_path="./data/ICIAR2018/imagewise_dataset",
        checkpoint_path="./models/Checkpoints/",
        name="_imagewise_",
        predefined_stats=False
```

```python
    )

    # Example

    patch_wise_model = PatchWiseModel(args_patch_wise)
    patch_wise_model.train_model(args_patch_wise)
    patch_wise_model.test(args_patch_wise, voting=True)
    patch_wise_model.test_separate_classes(args_patch_wise)
    patch_wise_model.test_training(args_patch_wise)
    patch_wise_model.plot_metrics()
    patch_wise_model.save_checkpoint("./models/")
    patch_wise_model.save_model("./models/")

    image_wise_model = EffNet(args_img_wise)
    image_wise_model.train_model(args_img_wise)
    image_wise_model.test(args_img_wise, True)
    image_wise_model.test_separate_classes(args_img_wise)
    image_wise_model.test_training(args_img_wise)
    image_wise_model.plot_metrics()
    image_wise_model.save_checkpoint("./models/")
    image_wise_model.save_model("./models/")
```

Listing 1: main.py

```python
from os.path import split
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from tqdm import tqdm
import numpy as np
import torchvision
import random
import torch
import torch
import os

VALIDATION_SET = 0.15
TRAINING_SET = 0.7
TEST_SET = 0.15

MEANS = [0.4731, 0.3757, 0.4117]
STD = [0.3731, 0.3243, 0.3199]

IMAGE_SIZE = (2816, 3072)
CROPPED_IMAGE_SIZE = (1536, 2048)
GRADED_LABELS = ["Grade 1", "Grade 2", "Grade 3"]
BACH_LABELS = ["Benign", "InSitu", "Invasive", "Normal"]
BREAKHIS_LABELS = ["Benign", "Malignant"]
SEED = 123

PATCH_SIZE = 512
STRIDE = 256

def imshow(img):
    import matplotlib.pyplot as plt
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

```

```
def set_seed(seed=SEED):
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)

def compute_normalization_stats(root_dir,  test_set=TEST_SET, training_set=TRAINING_SET, val_set=VALIDATION_SET):
    from tqdm import tqdm
    set_seed(SEED)
    assert test_set + training_set + val_set == 1, "Train/Test/Val Set sizes incorrect"
    BATCH_SIZE = 1

    t = transforms.Compose([
        transforms.Resize(IMAGE_SIZE),
        transforms.ToTensor()
    ])

    data = torchvision.datasets.ImageFolder(root=root_dir+"/Histopathological_Graded", transform=t)

    size = int((training_set+val_set)*len(data))
    test_size = len(data) - (size)
    train_data, _ = torch.utils.data.random_split(data, [size, test_size])

    data_loader = DataLoader(train_data, batch_size=BATCH_SIZE, num_workers=0)
    # Compute normalization metrics
    mean = 0.
    std = 0.

    # Training images
    i = 0
    for inputs, _ in tqdm(data_loader):
        input = inputs[0]

        temp = input.view(3, -1)
        mean += temp.mean(1)
        std += temp.std(1)
        i += 1

    print("Printing Normalization Metrics")
    mean /= size
    std /= size
    print("Means:", mean)
    print("Std:", std)

def split_test_train_val(root_dir, test_set=TEST_SET, training_set=TRAINING_SET, val_set=VALIDATION_SET, dataset="Databiox"):
    from tqdm import tqdm
    set_seed(SEED)
    assert test_set + training_set + val_set == 1, "Train/Test/Val Set sizes incorrect"
```

```python
    BATCH_SIZE = 1

    if dataset == "Databiox":
        t = transforms.Compose([
            transforms.Resize(IMAGE_SIZE),
            transforms.CenterCrop(CROPPED_IMAGE_SIZE),
            transforms.ToTensor()
        ])
        data = torchvision.datasets.ImageFolder(root=root_dir+"/Histopathological_Graded", transform=t)
        LABELS = GRADED_LABELS
    elif dataset == "BACH":
        t = transforms.Compose([
            transforms.Resize(CROPPED_IMAGE_SIZE),
            transforms.ToTensor()
        ])
        data = torchvision.datasets.ImageFolder(root=root_dir+"/ICIAR2018_BACH_Challenge/Photos", transform=t)
        LABELS = BACH_LABELS
    elif dataset == "BreakHis":
        t = transforms.Compose([
            transforms.Resize((PATCH_SIZE,PATCH_SIZE)),
            transforms.ToTensor()
        ])
        data = torchvision.datasets.ImageFolder(root=root_dir+"/BreaKHis_v1/histology_slides/breast", transform=t)
        LABELS = BREAKHIS_LABELS
    else:
        print("Dataset not recognised")
        return

    train_size = int(training_set*len(data))
    val_size = int(val_set*len(data))
    test_size = len(data) - (train_size + val_size)
    train_data, test_data, val_data = torch.utils.data.random_split(data, [train_size, test_size, val_size])
    train_data_loader = DataLoader(train_data, batch_size=BATCH_SIZE,  num_workers=0)
    test_data_loader = DataLoader(test_data, batch_size=BATCH_SIZE, num_workers=0)
    val_data_loader = DataLoader(val_data, batch_size=BATCH_SIZE, num_workers=0)
    print("Number of training images:", len(train_data), "Number of test images:", len(test_data), "Number of validation images:", len(val_data))

    if not os.path.exists(root_dir):
        os.makedirs(root_dir)

    if dataset == "BreakHis":
        for t in ["train", "test", "validation"]:
            for i in range(len(LABELS)):
                if not os.path.exists(root_dir + t + "/" + LABELS[i]):
                    os.makedirs(root_dir + t + "/" + LABELS[i])
        for i, (inputs, labels) in tqdm(enumerate(train_data_loader)):
```

```
        torchvision.utils.save_image(inputs, root_dir + "/train/" + LABELS[labels[0].item()] + "/image_" + str(i) + ".JPG")
    for i, (inputs, labels) in tqdm(enumerate(val_data_loader)):
        torchvision.utils.save_image(inputs, root_dir + "/validation/" + LABELS[labels[0].item()] + "/image_" + str(i) + ".JPG")
    for i, (inputs, labels) in tqdm(enumerate(test_data_loader)):
        torchvision.utils.save_image(inputs, root_dir + "/test/" + LABELS[labels[0].item()] + "/image_" + str(i) + ".JPG")
else:
    for mode in ["/imagewise_dataset/", "/patchwise_dataset/"]:
        for t in ["train", "test", "validation"]:
            for i in range(len(LABELS)):
                if not os.path.exists(root_dir + mode + t + "/" + LABELS[i]):
                    os.makedirs(root_dir + mode + t + "/" + LABELS[i])

    # Training images
    i = 0
    for inputs, labels in tqdm(train_data_loader):
        input = inputs[0]
        # ImageWise
        patches = input.unfold(1, PATCH_SIZE, PATCH_SIZE).unfold(2, PATCH_SIZE, PATCH_SIZE)
        patches = patches.permute(1,2,0,3,4).contiguous()
        patches = patches.contiguous().view(-1, 3, PATCH_SIZE, PATCH_SIZE)
        for j, patch in enumerate(patches):
            torchvision.utils.save_image(patch, root_dir + "/imagewise_dataset/train/" + LABELS[labels[0].item()] + "/image_" + str(i) + "patch_" + str(j) + ".JPG")

        # PatchWise
        patches = input.unfold(1, PATCH_SIZE, STRIDE).unfold(2, PATCH_SIZE, STRIDE)
        patches = patches.permute(1,2,0,3,4).contiguous()
        patches = patches.contiguous().view(-1, 3, PATCH_SIZE, PATCH_SIZE)
        for j, patch in enumerate(patches):
            torchvision.utils.save_image(patch, root_dir + "/patchwise_dataset/train/" + LABELS[labels[0].item()] + "/image_" + str(i) + "patch_" + str(j) + ".JPG")
        i += 1

    # Validation images
    i = 0
    for inputs, labels in tqdm(val_data_loader):
        input = inputs[0]
        # ImageWise
        patches = input.unfold(1, PATCH_SIZE, PATCH_SIZE).unfold(2, PATCH_SIZE, PATCH_SIZE)
        patches = patches.permute(1,2,0,3,4).contiguous()
        patches = patches.contiguous().view(-1, 3, PATCH_SIZE, PATCH_SIZE)
```

```python
            for j, patch in enumerate(patches):
                torchvision.utils.save_image(patch, root_dir + "/imagewise_dataset/validation/" + LABELS[labels[0].item()] + "/image_" + str(i) + "patch_" + str(j) + ".JPG")

            # PatchWise
            patches = input.unfold(1, PATCH_SIZE, STRIDE).unfold(2, PATCH_SIZE, STRIDE)
            patches = patches.permute(1,2,0,3,4).contiguous()
            patches = patches.contiguous().view(-1, 3, PATCH_SIZE, PATCH_SIZE)
            for j, patch in enumerate(patches):
                torchvision.utils.save_image(patch, root_dir + "/patchwise_dataset/validation/" + LABELS[labels[0].item()] + "/image_" + str(i) + "patch_" + str(j) + ".JPG")
            i += 1

        # Test images
        i = 0
        for inputs, labels in tqdm(test_data_loader):
            input = inputs[0]
            # ImageWise
            patches = input.unfold(1, PATCH_SIZE, PATCH_SIZE).unfold(2, PATCH_SIZE, PATCH_SIZE)
            patches = patches.permute(1,2,0,3,4).contiguous()
            patches = patches.contiguous().view(-1, 3, PATCH_SIZE, PATCH_SIZE)
            for j, patch in enumerate(patches):
                torchvision.utils.save_image(patch, root_dir + "/imagewise_dataset/test/" + LABELS[labels[0].item()] + "/image_" + str(i) + "patch_" + str(j) + ".JPG")

            # PatchWise
            patches = input.unfold(1, PATCH_SIZE, STRIDE).unfold(2, PATCH_SIZE, STRIDE)
            patches = patches.permute(1,2,0,3,4).contiguous()
            patches = patches.contiguous().view(-1, 3, PATCH_SIZE, PATCH_SIZE)
            for j, patch in enumerate(patches):
                torchvision.utils.save_image(patch, root_dir + "/patchwise_dataset/test/" + LABELS[labels[0].item()] + "/image_" + str(i) + "patch_" + str(j) + ".JPG")
            i += 1

def check_res(root_dir):
    import matplotlib.pyplot as plt
    sizes = {}
    data = torchvision.datasets.ImageFolder(root=root_dir+"/Histopathological_Graded", transform=transforms.Compose([transforms.ToTensor()]))
    data_loader = DataLoader(data , batch_size=1,  num_workers=0)
    for inputs, _ in tqdm(data_loader):
        t = inputs.shape
        if t in sizes:
            sizes[t] +=1
        else:
            sizes[t] = 1
```

```python

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

Listing 2: src/datasets.py

```python
# For the docs
import matplotlib.pyplot as plt

# Data
from torch.utils.data import DataLoader, ConcatDataset
import torchvision.transforms as transforms
from .datasets import MEANS, STD
from tqdm import tqdm
import torchvision
import PIL

# Training
import torch.optim as optim
import torch.nn as nn
import numpy as np
import torch
import copy
import time
import os

# For testing we want to get a whole image in patches
BATCH_SIZE = 12

class Model(nn.Module):
    """
    Basic module class, imagewise, patchwise and mixed models inherit from this one, overwriting the propagate method
    """
    def __init__(self, args):
        super(Model, self).__init__()
        self.name = args.name
        self.time = str(time.strftime('%Y-%m-%d_%H-%M'))

        if "breakhis" in args.data_path.lower():
            self.breakhis = True
        else:
            self.breakhis = False

    def init_device(self):
        """ Sends model to CPU / GPU """
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.to(self.device)

        print(self)
        print("Parameters:", sum(p.numel() for p in super(Model, self).parameters()))
        print("Trainable parameters:", sum(p.numel() for p in super(Model, self).parameters() if p.requires_grad))
        print("Using:", self.device)
```

```

        # Additional Info when using cuda
        if self.device.type == 'cuda':
            print(torch.cuda.get_device_name(0))
            print('Memory Usage:')
            print('Allocated:', round(torch.cuda.memory_allocated(0)/1024**3,1), 'GB')
            print('Cached:   ', round(torch.cuda.memory_reserved(0)/1024**3,1), 'GB')

    def train_model(self, args, path=None):
        """ Main Training loop with data augmentation, early stopping and scheduler """
        print('Start training network: {}\n'.format(time.strftime('%Y/%m/%d %H:%M')))

        if not args.predefined_stats:
            means = [0.5, 0.5, 0.5]
            std = [0.5, 0.5, 0.5]
        else:
            means = MEANS
            std = STD

        validation_transforms = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=means, std=std)
        ])

        if args.augment:
            """
            Create versions of the dataset for each augmentation as in https://arxiv.org/abs/1803.04054 and others
            """
            augmenting = [
                # 0 degrees
                transforms.Compose([
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
                    transforms.Normalize(mean=means, std=std)
                ]),
                # 90 degrees
                transforms.Compose([
                    transforms.RandomRotation((90, 90), resample=PIL.Image.BILINEAR),
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
                    transforms.Normalize(mean=means, std=std)
                ]),
                # 180 degrees
                transforms.Compose([
                    transforms.RandomRotation((180, 180), resample=PIL.Image.BILINEAR),
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
```

```
                transforms.Normalize(mean=means, std=std)
            ]),
            # 270 degrees + flip
            transforms.Compose([
                transforms.RandomRotation((270, 270), resample=PIL.Image.BILINEAR),
                transforms.ColorJitter(hue=.05, saturation=.05),
                transforms.ToTensor(),
                transforms.Normalize(mean=means, std=std)
            ])
        ]

        if args.flip:
            augmenting += [
                transforms.Compose([
                    transforms.RandomVerticalFlip(p=1.),
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
                    transforms.Normalize(mean=means, std=std)
                ]),
                transforms.Compose([
                    transforms.RandomVerticalFlip(p=1.),
                    transforms.RandomRotation((90, 90), resample=PIL.Image.BILINEAR),
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
                    transforms.Normalize(mean=means, std=std)
                ]),
                transforms.Compose([
                    transforms.RandomVerticalFlip(p=1.),
                    transforms.RandomRotation((180, 180), resample=PIL.Image.BILINEAR),
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
                    transforms.Normalize(mean=means, std=std)
                ]),
                transforms.Compose([
                    transforms.RandomVerticalFlip(p=1.),
                    transforms.RandomRotation((270, 270), resample=PIL.Image.BILINEAR),
                    transforms.ColorJitter(hue=.05, saturation=.05),
                    transforms.ToTensor(),
                    transforms.Normalize(mean=means, std=std)
                ])
            ]

        train_data_loader = DataLoader(
            ConcatDataset([
                torchvision.datasets.ImageFolder(root=args.data_path + "/train", transform=t) for t in augmenting
            ]),
            batch_size=args.batch_size, shuffle=True,
```

```
num_workers=args.workers
            )

        else:
            training_transforms = transforms.Compose([
                transforms.RandomHorizontalFlip(),
                transforms.RandomVerticalFlip(),
                transforms.RandomRotation(10, resample=PIL.Image.BILINEAR),
                transforms.ColorJitter(hue=.05, saturation=.05),
                transforms.ToTensor(),
                transforms.Normalize(mean=means, std=std)
            ])
            train_data = torchvision.datasets.ImageFolder(root=args.data_path + "/train", transform=training_transforms)
            train_data_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True,  num_workers=args.workers)

        val_data = torchvision.datasets.ImageFolder(root=args.data_path + "/validation", transform=validation_transforms)
        val_data_loader = DataLoader(val_data, batch_size=args.batch_size, shuffle=True,  num_workers=args.workers)

        print("Using ", len(train_data_loader.dataset), "training samples")
        print("Using ", len(val_data_loader.dataset), "validation samples")

        optimizer = optim.Adam(self.parameters(), lr=args.lr)
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)
        criterion = nn.CrossEntropyLoss()
        start_epoch = 0

        # If checkpoint provided load states
        if path:
            optimizer, start_epoch = self.load_ckp(path, optimizer)
            print("Model loaded, trained for ", start_epoch, " epochs")

        # keeping track of losses
        self.train_losses = []
        self.valid_losses = []
        self.train_acc = []
        self.val_acc = []
        since = time.time()

        # For "early stopping"
        best_model_wts = copy.deepcopy(self.state_dict())
        best_acc = 0.

        for epoch in range(start_epoch, args.epochs):
            print('Epoch {}/{}'.format(epoch+1, args.epochs))
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
```

```
            if phase == 'train':
                super(Model, self).train()  # Set model to training mode
                dataloader = train_data_loader
            else:
                super(Model, self).eval()   # Set model to evaluate mode
                dataloader = val_data_loader

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in tqdm(dataloader):
                inputs = inputs.to(self.device)
                labels = labels.to(self.device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # Training here
                with torch.set_grad_enabled(phase == 'train'):
                    # Overwrite this for each model
                    loss, preds = self.propagate(inputs, labels, criterion)
                    _, preds = torch.max(preds, 1)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            # Adjust learning rate
            if phase == 'train':
                scheduler.step()

            # Data metrics
            epoch_loss = running_loss /len(dataloader.dataset)
            epoch_acc = running_corrects.double() / len(dataloader.dataset)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            if phase == 'train':
                self.train_losses.append(epoch_loss)
                self.train_acc.append(epoch_acc)
            else:
                self.valid_losses.append(epoch_loss)
                self.val_acc.append(epoch_acc)

```

```
                # deep copy the model
                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_model_wts = copy.deepcopy(self.state_dict())
                    checkpoint = {
                        'epoch': epoch + 1,
                        'state_dict': best_model_wts,
                        'optimizer': optimizer.state_dict(),
                        'loss': criterion
                    }
                    file_name = "checkpoint_"+ str(epoch + 1) + args.name + self.time + ".ckpt"
                    torch.save(checkpoint, args.checkpoint_path + file_name)

        # Finished
        time_elapsed = time.time() - since
        print('Training complete in {:.0f}m {:.0f}s'.format(
            time_elapsed // 60, time_elapsed % 60))
        print('Best Validation Accuracy: {:4f}'.format(best_acc))

        # load best model weights and save checkpoint
        self.load_state_dict(best_model_wts)

    def propagate(self, inputs, labels, criterion=None):
        """ Default Training step - some models use this """
        outputs = self(inputs)
        if criterion:
            loss = criterion(outputs, labels)
        else:
            loss = 0
        return loss, outputs

    def plot_metrics(self, path, pr=False, pl=True):
        """ Plots accuracy and loss side-by-side """

        # Plotting on HPC throws error
        if pl:
            # Loss
            plt.subplot(1, 2, 1)
            plt.plot(self.train_losses, label='Training loss')
            plt.plot(self.valid_losses, label='Validation loss')
            plt.xlabel("Epochs")
            plt.ylabel("Loss")
            plt.legend(frameon=False)
            plt.savefig(path + "loss.png")

            # Accuracy
            plt.subplot(1, 2, 2)
            plt.plot(self.train_acc, label='Training Accuracy')
            plt.plot(self.val_acc, label='Validation Accuracy')
            plt.xlabel("Epochs")
            plt.ylabel("Acc")
            plt.legend(frameon=False)
            plt.savefig(path + "accuracy.png")

```

```python
        if pr:
            print(self.train_losses, self.valid_losses, self.train_acc, self.val_acc, sep="\n")

    def test(self, args, voting=False):
        """ Test on patched dataset """

        if not args.predefined_stats:
            means = [0.5, 0.5, 0.5]
            std = [0.5, 0.5, 0.5]
        else:
            means = MEANS
            std = STD

        test_data = torchvision.datasets.ImageFolder(root=args.data_path + "/test", transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=means, std=std)
        ]))
        test_data_loader = DataLoader(test_data, batch_size=BATCH_SIZE, shuffle=False, num_workers=args.workers)

        super(Model, self).eval()
        with torch.no_grad():
            patch_acc = 0
            image_acc_maj = 0
            image_acc_sum = 0
            image_acc_max = 0
            conf = []
            for images, labels in tqdm(test_data_loader):
                images = images.to(self.device)
                labels = labels.to(self.device)
                _, preds = self.propagate(images, labels)
                _, predicted = torch.max(preds, 1)
                patch_acc += (predicted == labels).sum().item()

                if not self.breakhis and voting:
                    # Voting
                    preds = preds.cpu()
                    predicted = predicted.cpu()

                    maj_prob = (args.classes - 1) - np.argmax(np.sum(np.eye(args.classes)[np.array(predicted).reshape(-1)], axis=0)[::-1])
                    sum_prob = (args.classes - 1) - np.argmax(np.sum(np.exp(preds.numpy()), axis=0)[::-1])
                    max_prob = (args.classes - 1) - np.argmax(np.max(np.exp(preds.numpy()), axis=0)[::-1])

                    confidence = np.sum(np.array(predicted) == maj_prob) / predicted.size(0)
                    conf.append(np.round(confidence * 100, 2))

                    if labels.data[0].item()== maj_prob:
                        image_acc_maj += 1

                    if labels.data[0].item()== sum_prob:
```

```
                        image_acc_sum += 1

                    if labels.data[0].item()== max_prob:
                        image_acc_max += 1

        patch_acc  /= len(test_data_loader.dataset)
        print('Test Accuracy of the model: {:.2f}'.format(patch_acc))

        if not self.breakhis and voting:
            print('Average Confidence: {:.2f}'.format(sum(conf)/len(conf)))
            image_acc_maj /=  (len(test_data_loader.dataset)/12)
            image_acc_sum /=  (len(test_data_loader.dataset)/12)
            image_acc_max /=  (len(test_data_loader.dataset)/12)

            print('Test Accuracy of the model on with majority voting: {:.2f}'.format(image_acc_maj))
            print('Test Accuracy of the model on with sum voting: {:.2f}'.format(image_acc_sum))
            print('Test Accuracy of the model on with max voting: {:.2f}'.format(image_acc_max))

    def test_separate_classes(self, args):
        """ Tests the model on each class separately and reports classification metrics """
        if not args.predefined_stats:
            means = [0.5, 0.5, 0.5]
            std = [0.5, 0.5, 0.5]
        else:
            means = MEANS
            std = STD

        test_data = torchvision.datasets.ImageFolder(root=args.data_path + "/test", transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=means, std=std)
        ]))
        test_data_loader = DataLoader(test_data, batch_size=args.batch_size, num_workers=args.workers)
        conf_matrix = torch.zeros(args.classes, args.classes)

        super(Model, self).eval()
        with torch.no_grad():
            for images, labels in tqdm(test_data_loader):
                images = images.to(self.device)
                labels = labels.to(self.device)
                _, predicted = self.propagate(images, labels)
                _, predicted = torch.max(predicted, 1)
                predicted = predicted.tolist()
                labels = labels.tolist()

                for t, p in zip(labels, predicted):
                    conf_matrix[t, p] += 1

        print('Confusion matrix\n', conf_matrix)

```

```
        TP = conf_matrix.diag()
        for c in range(args.classes):
            idx = torch.ones(args.classes)
            idx = idx.type(torch.BoolTensor)
            idx[c] = 0
            # all non-class samples classified as non-class
            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()
            # all non-class samples classified as class
            FP = conf_matrix[idx, c].sum()
            # all class samples not classified as class
            FN = conf_matrix[c, idx].sum()

            print('Class {}\nTP {}, TN {}, FP {}, FN {}'.format(
                c, TP[c], TN, FP, FN))
            print('Sensitivity {:.2f}, Specificity {:.2f}, F1 {:.2f}, Accuracy {:.2f}'.format(
                TP[c] / (TP[c]+FN), TN / (TN + FP), 2*TP[c] / (2*TP[c] + FP + FN), ((TP[c] + TN) / (TP[c] + TN + FP + FN))))

    def test_training(self, args):
        """ Test on patched training dataset for debugging """

        if not args.predefined_stats:
            means = [0.5, 0.5, 0.5]
            std = [0.5, 0.5, 0.5]
        else:
            means = MEANS
            std = STD

        train_data = torchvision.datasets.ImageFolder(root=args.data_path + "/train", transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=means, std=std)
        ]))
        train_data_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True, num_workers=args.workers)

        super(Model, self).eval()
        with torch.no_grad():
            correct = 0
            for images, labels in tqdm(train_data_loader):
                images = images.to(self.device)
                labels = labels.to(self.device)
                _, predicted = self.propagate(images, labels)
                _, predicted = torch.max(predicted, 1)
                correct += (predicted == labels).sum().item()

        print('Training Accuracy of the model: {:.2f}'.format(correct / len(train_data_loader.dataset)))

    def save_model(self, path):
        """ Save model after training has finished """
        file_name = path + self.name + self.time + ".ckpt"
        torch.save(self.state_dict(), file_name)
        print("Model saved:", file_name)
```

```
        return file_name

    def load(self, path):
        """ Load pre-trained weights """
        try:
            if os.path.exists(path):
                print('Loading model...')
                self.load_state_dict(torch.load(path, map_location=self.device))
        except:
            print('Failed to load pre-trained network with path:', path)

    def load_ckp(self, checkpoint_fpath, optimizer):
        """ To continue training we need more than just saving the weights """
        checkpoint = torch.load(checkpoint_fpath, map_location=self.device)
        self.load_state_dict(checkpoint['state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer'])
        return optimizer, checkpoint['epoch']
```

Listing 3: src/model.py

```
import torch.nn.functional as F
from .model import Model
import torch.nn as nn


class PatchWiseModel(Model):
    """
    A CNN classifier that is used by the image-wise networks to downscale the images
    by feeding them though the convolutional layers of the trained patchwise net.
    """
    def __init__(self, args, original_architecture=False):
        super(PatchWiseModel, self).__init__(args)
        if original_architecture:
            """
            This is the original architecture proposed in: https://arxiv.org/abs/1803.04054
            """
            self.features = nn.Sequential(
                # Block 1
                nn.Conv2d(in_channels=args.input_size[0], out_channels=16, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=16, out_channels=16, kernel_size=2, stride=2),
                nn.BatchNorm2d(16),
                nn.ReLU(inplace=True),

```

```
                # Block 2
                nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=32, out_channels=32, kernel_size=2, stride=2),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),

                # Block 3
                nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64, kernel_size=2, stride=2),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),

                # Block 4
                nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),

                # Block 5
                nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),

                nn.Conv2d(in_channels=256, out_channels=args.
```

```
output_size[0], kernel_size=1, stride=1),
            )
        else:
            """
            Smaller version using 10 conv layers instead of 16
            """
            self.features = nn.Sequential(
                # Block 1
                nn.Conv2d(in_channels=args.input_size[0],
out_channels=16, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=16, out_channels=16,
kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=16, out_channels=16,
kernel_size=2, stride=2),
                nn.BatchNorm2d(16),
                nn.ReLU(inplace=True),

                # Block 2
                nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=32, out_channels=32,
kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=32, out_channels=32,
kernel_size=2, stride=2),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),

                # Block 3
                nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64,
kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=64, out_channels=64,
kernel_size=2, stride=2),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),

                nn.Conv2d(in_channels=64, out_channels=args.
output_size[0], kernel_size=1, stride=1),
            )

        # The classification layer
        self.classifier = nn.Sequential(
            nn.Linear(args.output_size[0] * args.output_size[1] *
args.output_size[2], args.classes),
```

```python
        )

        self.initialize_weights()

        # This will send net to device, so only call here
        self.init_device()

    def initialize_weights(self):
        """ As in https://arxiv.org/abs/1803.04054 """
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, nonlinearity='
    relu')
                if m.bias is not None:
                    m.bias.data.zero_()

            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

            elif isinstance(m, nn.Linear):
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        x = F.log_softmax(x, dim=1)
        return x
```

Listing 4: src/patchwisemodel.py

```python
# Base Models
from .patchwisemodel import PatchWiseModel
from .model import Model

# Dynamic
from .DynamicCaps.capsulelayers import DenseCapsule, PrimaryCapsule
from .DynamicCaps.capsulenet import caps_loss

# Varcaps
from .VarCaps import layers
from .VarCaps import vb_routing

# SR Capsules
from .SRCaps.modules import SelfRouting2d

# Data
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from .datasets import MEANS, STD
import torchvision

# Training
from torch.autograd import Variable
import torch.nn.functional as F
import torch.nn as nn
```

```python
import numpy as np
import torch


class ImageWiseModels(Model):
    """
    Base Image-Wise model, variants inherit from this class
    it assigns a hopefully trained patchwise net to the model to use for forward passes
    """
    def __init__(self, args, patchwise, original_architecture):
        super(ImageWiseModels, self).__init__(args)

        if patchwise is not None:
            self.patch_wise_model = patchwise
        else:
            print("Creating untrained patchwise model")
            self.patch_wise_model = PatchWiseModel(args, original_architecture=original_architecture)

    def propagate(self, inputs, labels, criterion=None):
        inputs = self.patch_wise_model.features(inputs)
        outputs = self(inputs)
        if criterion:
            loss = criterion(outputs, labels)
        else:
            loss = 0
        return loss, outputs

class BaseCNN(ImageWiseModels):
    """ Simpler CNN baseline than Nazeri """
    def __init__(self, args, patchwise=None, original_architecture=False):
        super(BaseCNN, self).__init__(args, patchwise, original_architecture)

        self.cnn_layers = nn.Sequential(
            # Convolutional Layer 1
            nn.Conv2d(args.input_size[0], 32, kernel_size=5, stride=1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 1),

            # Convolutional Layer 2
            nn.Conv2d(32, 32, kernel_size=5, stride=1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 1),

            # Convolutional Layer 3
            nn.Conv2d(32, 16, kernel_size=5, stride=1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 1),
        )

```

```python
        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.to(self.device)

        self.set_linear_layer(args)

        # This will send net to device, so only call here
        self.init_device()

    def forward(self, x):
        x = self.cnn_layers(x)
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        return x

    def set_linear_layer(self, args):
        """ Pytorch has no nice way of connecting CNNs with the denselayers, this code sets the dimensions correctly on the fly """
        train_data = torchvision.datasets.ImageFolder(root=args.data_path + "/train", transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=MEANS, std=STD)
        ]))
        train_data_loader = DataLoader(train_data, batch_size=args.batch_size, shuffle=True,  num_workers=args.workers)
        super(BaseCNN, self).train()  # Set model to training mode
        data, _ = next(iter(train_data_loader))
        #data = data[0]
        x = data.to(self.device)
        x = self.patch_wise_model.features(x)
        x = self.cnn_layers(x)

        print("Setting Linear Layer with shape:", x.shape)
        self.linear_layers = nn.Sequential(
            nn.Linear(x.size(1)*x.size(2)*x.size(3), 64),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(64, 32),
            nn.ReLU(inplace=True),
            #nn.Dropout(0.5),
            nn.Linear(32, args.classes) # NO softmax, bc it is in crossentropy loss
        )
        self.to(self.device)

class NazeriCNN(ImageWiseModels):
    """ CNN imagewise network as in https://arxiv.org/abs/1803.04054 """
    def __init__(self, args, patchwise=None, original_architecture=False):
        super(NazeriCNN, self).__init__(args, patchwise, original_architecture)

        self.features = nn.Sequential(
            # Block 1
            nn.Conv2d(in_channels=args.input_size[0], out_channels
```

```
=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=2, stride=2),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),

            # Block 2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=2, stride=2),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),

            nn.Conv2d(in_channels=128, out_channels=1, kernel_size=1, stride=1),
        )

        self.classifier = nn.Sequential(
            nn.Linear(1 * 16 * 16, 128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),

            nn.Linear(128, 128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),

            nn.Linear(128, 64),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),

            nn.Linear(64, args.classes),
        )

        # This will send net to device, so only call here
        self.init_device()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        x = F.log_softmax(x, dim=1)
        return x

    def initialize_weights(self):
```

```
        """ As in https://arxiv.org/abs/1803.04054 """
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
                if m.bias is not None:
                    m.bias.data.zero_()

            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

            elif isinstance(m, nn.Linear):
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()

class DynamicCapsules(ImageWiseModels):
    """
    A Capsule Network adapted from https://github.com/XifengGuo/CapsNet-Pytorch
    """
    def __init__(self, args, patchwise=None, original_architecture=False):
        super(DynamicCapsules, self).__init__(args, patchwise, original_architecture)
        self.output_size = args.output_size
        self.classes = args.classes
        self.routings = args.routings
        self.lam_recon = args.lam_recon
        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        # Layer 1: Just a conventional Conv2D layer
        self.conv1 = nn.Conv2d(args.output_size[0], 64, kernel_size=9, stride=1, padding=0)

        # Layer 2: Conv2D layer with `squash` activation, then reshape to [None, num_caps, dim_caps]
        self.primarycaps = PrimaryCapsule(64, 64, 8, kernel_size=9, stride=2, padding=0)

        # Layer 3: Capsule layer. Routing algorithm works here.
        self.digitcaps = DenseCapsule(in_num_caps=4608, in_dim_caps=8,
                                      out_num_caps=args.classes, out_dim_caps=16, routings=self.routings, device=self.device)

        # Decoder network.
        self.decoder = nn.Sequential(
            nn.Linear(16*args.classes, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, args.output_size[0] * args.output_size[1] * args.output_size[2]),
            nn.Sigmoid()
        )
```

```

        self.relu = nn.ReLU()

        # This will send net to device, so only call here
        self.init_device()

    def forward(self, x, y=None):
        x = self.relu(self.conv1(x))
        x = self.primarycaps(x)
        x = self.digitcaps(x)
        length = x.norm(dim=-1)
        if y is None:  # during testing, no label given. create one-hot coding using `length`
            index = length.max(dim=1)[1]
            y = Variable(torch.zeros(length.size()).scatter_(1, index.view(-1, 1).cpu().data, 1.).to(self.device))
        reconstruction = self.decoder((x * y[:, :, None]).view(x.size(0), -1))
        return length, reconstruction.view(-1, *self.output_size)

    def propagate(self, inputs, labels, criterion=None):
        labels = torch.zeros(labels.size(0), self.classes).to(self.device).scatter_(1, labels.view(-1, 1), 1.)  # change to one-hot coding
        inputs = self.patch_wise_model.features(inputs)
        if criterion:
            y_pred, x_recon = self(inputs, labels)
            loss = caps_loss(labels, y_pred, inputs, x_recon, self.lam_recon)  # compute loss
        else:
            y_pred, x_recon = self(inputs) # No y in testing
            loss = 0
        return loss, y_pred

class VariationalCapsules(ImageWiseModels):
    """
    Capsule Routing via Variational Bayes based on https://github.com/fabio-deep/Variational-Capsule-Routing
    """
    def __init__(self, args, patchwise=None, original_architecture=False):
        super(VariationalCapsules, self).__init__(args, patchwise, original_architecture)

        self.output_size = args.output_size
        self.n_classes = args.classes
        self.routings = args.routings

        self.P = args.pose_dim
        self.PP = int(np.max([2, self.P*self.P]))
        self.A, self.B, self.C, self.D = args.arch

        # Layer 1: Just a conventional Conv2D layer
        self.Conv_1 = nn.Conv2d(self.output_size[0], self.A, kernel_size=5, stride=2, bias=False)
        nn.init.kaiming_uniform_(self.Conv_1.weight)

```

```
        self.BN_1 = nn.BatchNorm2d(self.A)
        self.PrimaryCaps = layers.PrimaryCapsules2d(in_channels=self.A, out_caps=self.B,
            kernel_size=1, stride=1, pose_dim=self.P)

        self.ConvCaps_1 = layers.ConvCapsules2d(in_caps=self.B, out_caps=self.C,
            kernel_size=3, stride=2, pose_dim=self.P)

        self.ConvRouting_1 = vb_routing.VariationalBayesRouting2d(in_caps=self.B, out_caps=self.C,
            kernel_size=3, stride=2, pose_dim=self.P,
            cov='diag', iter=args.routings,
            alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
            Psi0=torch.eye(self.PP), nu0=self.PP+1)

        self.ConvCaps_2 = layers.ConvCapsules2d(in_caps=self.C, out_caps=self.D,
            kernel_size=3, stride=1, pose_dim=self.P)

        self.ConvRouting_2 = vb_routing.VariationalBayesRouting2d(in_caps=self.C, out_caps=self.D,
            kernel_size=3, stride=1, pose_dim=self.P,
            cov='diag', iter=args.routings,
            alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
            Psi0=torch.eye(self.PP), nu0=self.PP+1)

        self.ClassCaps = layers.ConvCapsules2d(in_caps=self.D, out_caps=self.n_classes,
            kernel_size=1, stride=1, pose_dim=self.P, share_W_ij=True, coor_add=True)

        self.ClassRouting = vb_routing.VariationalBayesRouting2d(in_caps=self.D, out_caps=self.n_classes,
            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.
            cov='diag', iter=args.routings,
            alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
            Psi0=torch.eye(self.PP), nu0=self.PP+1, class_caps=True)

        # This will send net to device, so only call here
        self.init_device()

    def forward(self, x):
        # Out     [?, A, F, F]
        x = F.relu(self.BN_1(self.Conv_1(x)))
        # Out     a [?, B, F, F], v [?, B, P, P, F, F]
        a,v = self.PrimaryCaps(x)
        # Out     a [?, B, 1, 1, 1, F, F, K, K], v [?, B, C, P*P, 1, F, F, K, K]
        a,v = self.ConvCaps_1(a, v, self.device)
        # Out     a [?, C, F, F], v [?, C, P, P, F, F]
        a,v = self.ConvRouting_1(a, v, self.device)
        # Out     a [?, C, 1, 1, 1, F, F, K, K], v [?, C, D, P*P, 1, F, F, K, K]
        a,v = self.ConvCaps_2(a, v, self.device)
```

```python
        # Out     a [?, D, F, F], v [?, D, P, P, F, F]
        a,v = self.ConvRouting_2(a, v, self.device)
        # Out     a [?, D, 1, 1, 1, F, F, K, K], v [?, D, n_classes, P*P, 1, F, F, K, K]
        a,v = self.ClassCaps(a, v, self.device)
        # Out     yhat [?, n_classes], v [?, n_classes, P, P]
        yhat, v = self.ClassRouting(a, v, self.device)
        return yhat

class SRCapsules(ImageWiseModels):
    """
    Self Routing Capsules based on https://github.com/coder3000/SR-CapsNet
    """
    def __init__(self, args, patchwise=None, original_architecture=False):
        super(SRCapsules, self).__init__(args, patchwise, original_architecture)

        planes = 16
        last_size = 14
        self.num_caps = 16

        self.conv1 = nn.Conv2d(args.input_size[0], 256, kernel_size=7, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(256)
        self.conv_a = nn.Conv2d(256, self.num_caps, kernel_size=5, stride=1, padding=1, bias=False)
        self.conv_pose = nn.Conv2d(256, self.num_caps*planes, kernel_size=5, stride=1, padding=1, bias=False)
        self.bn_a = nn.BatchNorm2d(self.num_caps)
        self.bn_pose = nn.BatchNorm2d(self.num_caps*planes)

        self.conv_caps = SelfRouting2d(self.num_caps, self.num_caps, planes, planes, kernel_size=3, stride=2, padding=1, pose_out=True)
        self.bn_pose_conv_caps = nn.BatchNorm2d(self.num_caps*planes)

        self.fc_caps = SelfRouting2d(self.num_caps, args.classes, planes, 1, kernel_size=last_size, padding=0, pose_out=False)

        # This will send net to device, so only call here
        self.init_device()

        self.loss = nn.NLLLoss()
        self.loss.to(self.device)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        a, pose = self.conv_a(out), self.conv_pose(out)
        a, pose = torch.sigmoid(self.bn_a(a)), self.bn_pose(pose)

        a, pose = self.conv_caps(a, pose)
        pose = self.bn_pose_conv_caps(pose)

        a, _ = self.fc_caps(a, pose)
```

```python

        out = a.view(a.size(0), -1)
        out = out.log()
        return out

    def propagate(self, inputs, labels, criterion=None):
        inputs = self.patch_wise_model.features(inputs)
        y_pred = self(inputs)
        if criterion:
            loss = self.loss(y_pred, labels)
        else:
            loss = 0
        return loss, y_pred
```

Listing 5: src/imagewisemodels.py

```python
# Base model
from .model import Model

# Variational Capsules
from .VarCaps import layers
from .VarCaps import vb_routing

# EfficientNet
from efficientnet_pytorch import EfficientNet

# Training
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import torch
import time

# For testing we want to get a whole image in patches
BATCH_SIZE = 12

class VariationalMixedCapsules(Model):
    """
    Capsule Routing via Variational Bayes based on https://github.
    com/fabio-deep/Variational-Capsule-Routing
    """

    def __init__(self, args):
        super(VariationalMixedCapsules, self).__init__(args)
        self.output_size = args.output_size
        self.n_classes = args.classes
        self.routings = args.routings

        self.P = args.pose_dim
        self.PP = int(np.max([2, self.P*self.P]))
        self.A, self.B, self.C, self.D = args.arch
        K = 12

        self.features = nn.Sequential(
            # Block 1
            nn.Conv2d(in_channels=args.input_size[0], out_channels
    =16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(16),
```

```
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=16, out_channels=16, kernel_size=2, stride=2),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),

            # Block 2
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=2, stride=2),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),

            # Block 3
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=2, stride=2),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),

            nn.Conv2d(in_channels=64, out_channels=args.output_size[0], kernel_size=1, stride=1),
        )

        # Layer 1: Just a conventional Conv2D layer
        self.Conv_1 = nn.Conv2d(args.input_size[0], self.A, kernel_size=5, stride=2, bias=False)
        nn.init.kaiming_uniform_(self.Conv_1.weight)

        self.Conv_2 = nn.Conv2d(self.A, self.A, kernel_size=5, stride=2, bias=False)
        nn.init.kaiming_uniform_(self.Conv_2.weight)

        self.Conv_3 = nn.Conv2d(self.A, self.A, kernel_size=3, stride=2, bias=False)
        nn.init.kaiming_uniform_(self.Conv_3.weight)

        self.BN_1 = nn.BatchNorm2d(self.A)
        self.PrimaryCaps = layers.PrimaryCapsules2d(in_channels=
```

```
self.A, out_caps=self.B,
                kernel_size=1, stride=1, pose_dim=self.P)

        self.ConvCaps_1 = layers.ConvCapsules2d(in_caps=self.B, out_caps=self.C,
                kernel_size=3, stride=2, pose_dim=self.P)

        self.ConvRouting_1 = vb_routing.VariationalBayesRouting2d(in_caps=self.B, out_caps=self.C,
                kernel_size=3, stride=2, pose_dim=self.P,
                cov='diag', iter=args.routings,
                alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
                Psi0=torch.eye(self.PP), nu0=self.PP+1)

        self.ConvCaps_2 = layers.ConvCapsules2d(in_caps=self.C, out_caps=self.D,
                kernel_size=3, stride=1, pose_dim=self.P)

        self.ConvRouting_2 = vb_routing.VariationalBayesRouting2d(in_caps=self.C, out_caps=self.D,
                kernel_size=3, stride=1, pose_dim=self.P,
                cov='diag', iter=args.routings,
                alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
                Psi0=torch.eye(self.PP), nu0=self.PP+1)

        self.ClassCaps = layers.ConvCapsules2d(in_caps=self.D, out_caps=self.n_classes,
                kernel_size=1, stride=1, pose_dim=self.P, share_W_ij=True, coor_add=True)

        self.ClassRouting = vb_routing.VariationalBayesRouting2d(in_caps=self.D, out_caps=self.n_classes,
                kernel_size=K, stride=1, pose_dim=self.P, # adjust final kernel_size K depending on input H/W, for H=W=32, K=4.
                cov='diag', iter=args.routings,
                alpha0=1., m0=torch.zeros(self.PP), kappa0=1.,
                Psi0=torch.eye(self.PP), nu0=self.PP+1, class_caps=True)

        # This will send net to device, so only call here
        self.init_device()

    def forward(self, x):
        x = self.features(x)
        # Out     [?, A, F, F]
        x = F.relu(self.BN_1(self.Conv_1(x)))
        # Out     a [?, B, F, F], v [?, B, P, P, F, F]
        a,v = self.PrimaryCaps(x)
        # Out     a [?, B, 1, 1, 1, F, F, K, K], v [?, B, C, P*P, 1, F, F, K, K]
        a,v = self.ConvCaps_1(a, v, self.device)
        # Out     a [?, C, F, F], v [?, C, P, P, F, F]
        a,v = self.ConvRouting_1(a, v, self.device)
        # Out     a [?, C, 1, 1, 1, F, F, K, K], v [?, C, D, P*P, 1, F, F, K, K]
        a,v = self.ConvCaps_2(a, v, self.device)
        # Out     a [?, D, F, F], v [?, D, P, P, F, F]
```

```
        a,v = self.ConvRouting_2(a, v, self.device)
        # Out     a [?, D, 1, 1, 1, F, F, K, K], v [?, D, n_classes
    , P*P, 1, F, F, K, K]
        a,v = self.ClassCaps(a, v, self.device)
        # Out     yhat [?, n_classes], v [?, n_classes, P, P]
        yhat, v = self.ClassRouting(a, v, self.device)
        return yhat

class EffNet(Model):
    """
    EfficientNet based on the implementation https://github.com/
    lukemelas/EfficientNet-PyTorch
    """

    def __init__(self, args):
        super(EffNet, self).__init__(args)

        self.time = str(time.strftime('%Y-%m-%d_%H-%M'))
        self.device = torch.device("cuda:0" if torch.cuda.
    is_available() else "cpu")
        self.output_size = args.output_size
        self.n_classes = args.classes

        self.model = EfficientNet.from_pretrained('efficientnet-b0'
    )
        num_ftrs = self.model._fc.in_features
        self.model._fc = nn.Sequential(
            nn.Linear(num_ftrs, 128),
            nn.ReLU(),
            nn.Linear(128, self.n_classes)
        )

        for param in self.model.parameters():
            param.requires_grad = False

        for param in self.model._fc.parameters():
            param.requires_grad = True

        # This will send net to device, so only call here
        self.init_device()

    def propagate(self, inputs, labels, criterion=None):
        outputs = self.model(inputs)
        if criterion:
            loss = criterion(outputs, labels)
        else:
            loss = 0
        return loss, outputs
```

Listing 6: src/mixedmodels.py

```
"""
Author: Xifeng Guo, E-mail: `guoxifeng1990@163.com`, Github: `https
    ://github.com/XifengGuo/CapsNet-Pytorch`
"""

import torch
import torch.nn as nn
```

```python
import torch.nn.functional as F
from torch.autograd import Variable


def squash(inputs, axis=-1):
    """
    The non-linear activation used in Capsule. It drives the length of a large vector to near 1 and small vector to 0
    :param inputs: vectors to be squashed
    :param axis: the axis to squash
    :return: a Tensor with same size as inputs
    """
    norm = torch.norm(inputs, p=2, dim=axis, keepdim=True)
    scale = norm**2 / (1 + norm**2) / (norm + 1e-8)
    return scale * inputs


class DenseCapsule(nn.Module):
    """
    The dense capsule layer. It is similar to Dense (FC) layer. Dense layer has `in_num` inputs, each is a scalar, the
    output of the neuron from the former layer, and it has `out_num` output neurons. DenseCapsule just expands the
    output of the neuron from scalar to vector. So its input size = [None, in_num_caps, in_dim_caps] and output size = \
    [None, out_num_caps, out_dim_caps]. For Dense Layer, in_dim_caps = out_dim_caps = 1.

    :param in_num_caps: number of cpasules inputted to this layer
    :param in_dim_caps: dimension of input capsules
    :param out_num_caps: number of capsules outputted from this layer
    :param out_dim_caps: dimension of output capsules
    :param routings: number of iterations for the routing algorithm
    """
    def __init__(self, in_num_caps, in_dim_caps, out_num_caps, out_dim_caps, device, routings=3):
        super(DenseCapsule, self).__init__()
        self.in_num_caps = in_num_caps
        self.in_dim_caps = in_dim_caps
        self.out_num_caps = out_num_caps
        self.out_dim_caps = out_dim_caps
        self.routings = routings
        self.device = device
        self.weight = nn.Parameter(0.01 * torch.randn(out_num_caps, in_num_caps, out_dim_caps, in_dim_caps))

    def forward(self, x):
        # x.size=[batch, in_num_caps, in_dim_caps]
        # expanded to    [batch, 1,            in_num_caps, in_dim_caps,  1]
        # weight.size   =[       out_num_caps, in_num_caps, out_dim_caps, in_dim_caps]
        # torch.matmul: [out_dim_caps, in_dim_caps] x [in_dim_caps, 1] -> [out_dim_caps, 1]
        # => x_hat.size =[batch, out_num_caps, in_num_caps, out_dim_caps]
```

```
        x_hat = torch.squeeze(torch.matmul(self.weight, x[:, None, :, :, None]), dim=-1)

        # In forward pass, `x_hat_detached` = `x_hat`;
        # In backward, no gradient can flow from `x_hat_detached` back to `x_hat`.
        x_hat_detached = x_hat.detach()

        # The prior for coupling coefficient, initialized as zeros.
        # b.size = [batch, out_num_caps, in_num_caps]
        b = Variable(torch.zeros(x.size(0), self.out_num_caps, self.in_num_caps)).to(self.device)

        assert self.routings > 0, 'The \'routings\' should be > 0.'
        for i in range(self.routings):
            # c.size = [batch, out_num_caps, in_num_caps]
            c = F.softmax(b, dim=1)

            # At last iteration, use `x_hat` to compute `outputs` in order to backpropagate gradient
            if i == self.routings - 1:
                # c.size expanded to [batch, out_num_caps, in_num_caps, 1           ]
                # x_hat.size     =   [batch, out_num_caps, in_num_caps, out_dim_caps]
                # => outputs.size=   [batch, out_num_caps, 1,           out_dim_caps]
                outputs = squash(torch.sum(c[:, :, :, None] * x_hat, dim=-2, keepdim=True))
                # outputs = squash(torch.matmul(c[:, :, None, :], x_hat))  # alternative way
            else:  # Otherwise, use `x_hat_detached` to update `b`. No gradients flow on this path.
                outputs = squash(torch.sum(c[:, :, :, None] * x_hat_detached, dim=-2, keepdim=True))
                # outputs = squash(torch.matmul(c[:, :, None, :], x_hat_detached))  # alternative way

                # outputs.size       =[batch, out_num_caps, 1,           out_dim_caps]
                # x_hat_detached.size=[batch, out_num_caps, in_num_caps, out_dim_caps]
                # => b.size          =[batch, out_num_caps, in_num_caps]
                b = b + torch.sum(outputs * x_hat_detached, dim=-1)

        return torch.squeeze(outputs, dim=-2)


class PrimaryCapsule(nn.Module):
    """
    Apply Conv2D with `out_channels` and then reshape to get capsules
    :param in_channels: input channels
    :param out_channels: output channels
    :param dim_caps: dimension of capsule
    :param kernel_size: kernel size
```

```
    :return: output tensor, size=[batch, num_caps, dim_caps]
    """
    def __init__(self, in_channels, out_channels, dim_caps,
    kernel_size, stride=1, padding=0):
        super(PrimaryCapsule, self).__init__()
        self.dim_caps = dim_caps
        self.conv2d = nn.Conv2d(in_channels, out_channels,
    kernel_size=kernel_size, stride=stride, padding=padding)

    def forward(self, x):
        outputs = self.conv2d(x)
        outputs = outputs.view(x.size(0), -1, self.dim_caps)
        return squash(outputs)
```

Listing 7: src/DynamicCaps/capsulelayers.py

```
"""
Author: Xifeng Guo, E-mail: `guoxifeng1990@163.com`, Github: `https
    ://github.com/XifengGuo/CapsNet-Pytorch`
"""

import torch
from torch import nn

def caps_loss(y_true, y_pred, x, x_recon, lam_recon):
    """
    Capsule loss = Margin loss + lam_recon * reconstruction loss.
    :param y_true: true labels, one-hot coding, size=[batch,
    classes]
    :param y_pred: predicted labels by CapsNet, size=[batch,
    classes]
    :param x: input data, size=[batch, channels, width, height]
    :param x_recon: reconstructed data, size is same as `x`
    :param lam_recon: coefficient for reconstruction loss
    :return: Variable contains a scalar loss value.
    """
    L = y_true * torch.clamp(0.9 - y_pred, min=0.) ** 2 + \
        0.5 * (1 - y_true) * torch.clamp(y_pred - 0.1, min=0.) ** 2
    L_margin = L.sum(dim=1).mean()

    L_recon = nn.MSELoss()(x_recon, x)

    return L_margin + lam_recon * L_recon
```

Listing 8: src/DynamicCaps/capsulenet.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

import math

eps = 1e-12

class SelfRouting2d(nn.Module):
    def __init__(self, A, B, C, D, kernel_size=3, stride=1, padding
    =1, pose_out=False):
        super(SelfRouting2d, self).__init__()
```

```
        self.A = A
        self.B = B
        self.C = C
        self.D = D

        self.k = kernel_size
        self.kk = kernel_size ** 2
        self.kkA = self.kk * A

        self.stride = stride
        self.pad = padding

        self.pose_out = pose_out

        if pose_out:
            self.W1 = nn.Parameter(torch.FloatTensor(self.kkA, B*D,
 C))
            nn.init.kaiming_uniform_(self.W1.data)

        self.W2 = nn.Parameter(torch.FloatTensor(self.kkA, B, C))
        self.b2 = nn.Parameter(torch.FloatTensor(1, 1, self.kkA, B)
)

        nn.init.constant_(self.W2.data, 0)
        nn.init.constant_(self.b2.data, 0)

    def forward(self, a, pose):
        # a: [b, A, h, w]
        # pose: [b, AC, h, w]
        b, _, h, w = a.shape

        # [b, ACkk, l]
        pose = F.unfold(pose, self.k, stride=self.stride, padding=
self.pad)
        l = pose.shape[-1]
        # [b, A, C, kk, l]
        pose = pose.view(b, self.A, self.C, self.kk, l)
        # [b, l, kk, A, C]
        pose = pose.permute(0, 4, 3, 1, 2).contiguous()
        # [b, l, kkA, C, 1]
        pose = pose.view(b, l, self.kkA, self.C, 1)

        if hasattr(self, 'W1'):
            # [b, l, kkA, BD]
            pose_out = torch.matmul(self.W1, pose).squeeze(-1)
            # [b, l, kkA, B, D]
            pose_out = pose_out.view(b, l, self.kkA, self.B, self.D
)

        # [b, l, kkA, B]
        logit = torch.matmul(self.W2, pose).squeeze(-1) + self.b2

        # [b, l, kkA, B]
        r = torch.softmax(logit, dim=3)

        # [b, kkA, l]
        a = F.unfold(a, self.k, stride=self.stride, padding=self.
```

```
    pad)
        # [b, A, kk, l]
        a = a.view(b, self.A, self.kk, l)
        # [b, l, kk, A]
        a = a.permute(0, 3, 2, 1).contiguous()
        # [b, l, kkA, 1]
        a = a.view(b, l, self.kkA, 1)

        # [b, l, kkA, B]
        ar = a * r
        # [b, l, 1, B]
        ar_sum = ar.sum(dim=2, keepdim=True)
        # [b, l, kkA, B, 1]
        coeff = (ar / (ar_sum)).unsqueeze(-1)

        # [b, l, B]
        # a_out = ar_sum.squeeze(2)
        a_out = ar_sum / a.sum(dim=2, keepdim=True)
        a_out = a_out.squeeze(2)

        # [b, B, l]
        a_out = a_out.transpose(1,2)

        if hasattr(self, 'W1'):
            # [b, l, B, D]
            pose_out = (coeff * pose_out).sum(dim=2)
            # [b, l, BD]
            pose_out = pose_out.view(b, l, -1)
            # [b, BD, l]
            pose_out = pose_out.transpose(1,2)

        oh = ow = math.floor(l**(1/2))

        a_out = a_out.view(b, -1, oh, ow)
        if hasattr(self, 'W1'):
            pose_out = pose_out.view(b, -1, oh, ow)
        else:
            pose_out = None

        return a_out, pose_out
```

Listing 9: src/SRCaps/modules.py

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F

class PrimaryCapsules2d(nn.Module):
    '''Primary Capsule Layer'''
    def __init__(self, in_channels, out_caps, kernel_size, stride,
        padding=0, pose_dim=4, weight_init='xavier_uniform'):
        super().__init__()

        self.A = in_channels
        self.B = out_caps
        self.P = pose_dim
        self.K = kernel_size
```

```
        self.S = stride
        self.padding = padding

        w_kernel = torch.empty(self.B*self.P*self.P, self.A, self.K, self.K)
        a_kernel = torch.empty(self.B, self.A, self.K, self.K)

        if weight_init == 'kaiming_normal':
            nn.init.kaiming_normal_(w_kernel)
            nn.init.kaiming_normal_(a_kernel)
        elif weight_init == 'kaiming_uniform':
            nn.init.kaiming_uniform_(w_kernel)
            nn.init.kaiming_uniform_(a_kernel)
        elif weight_init == 'xavier_normal':
            nn.init.xavier_normal_(w_kernel)
            nn.init.xavier_normal_(a_kernel)
        elif weight_init == 'xavier_uniform':
            nn.init.xavier_uniform_(w_kernel)
            nn.init.xavier_uniform_(a_kernel)
        else:
            NotImplementedError('{} not implemented.'.format(weight_init))

        # Out    [B*(P*P+1), A, K, K]
        self.weight = nn.Parameter(torch.cat([w_kernel, a_kernel], dim=0))

        self.BN_a = nn.BatchNorm2d(self.B, affine=True)
        self.BN_p = nn.BatchNorm3d(self.B, affine=True)

    def forward(self, x): # [?, A, F, F]    In

        # Out    [?, B*(P*P+1), F, F]
        x = F.conv2d(x, weight=self.weight, stride=self.S, padding=self.padding)

        # Out    ([?, B*P*P, F, F], [?, B, F, F])    [?, B*(P*P+1), F, F]
        poses, activations = torch.split(x, [self.B*self.P*self.P, self.B], dim=1)

        # Out    [?, B, P*P, F, F]
        poses = self.BN_p(poses.reshape(-1, self.B, self.P*self.P, *x.shape[2:]))

        # Out    [?, B, P, P, F, F]    [?, B, P*P, F, F]    In
        poses = poses.reshape(-1, self.B, self.P, self.P, *x.shape[2:])

        # Out    [?, B, F, F])
        activations = torch.sigmoid(self.BN_a(activations))

        return (activations, poses)

class ConvCapsules2d(nn.Module):
    '''Convolutional Capsule Layer'''
    def __init__(self, in_caps, out_caps, pose_dim, kernel_size,
```

```
stride, padding=0,
        weight_init='xavier_uniform', share_W_ij=False, coor_add=False):
        super().__init__()

        self.B = in_caps
        self.C = out_caps
        self.P = pose_dim
        self.PP = np.max([2, self.P*self.P])
        self.K = kernel_size
        self.S = stride
        self.padding = padding

        self.share_W_ij = share_W_ij # share the transformation matrices across (F*F)
        self.coor_add = coor_add # embed coordinates

        # Out      [1, B, C, 1, P, P, 1, 1, K, K]
        self.W_ij = torch.empty(1, self.B, self.C, 1, self.P, self.P, 1, 1, self.K, self.K)

        if weight_init.split('_')[0] == 'xavier':
            fan_in = self.B * self.K*self.K * self.PP # in_caps types * receptive field size
            fan_out = self.C * self.K*self.K * self.PP # out_caps types * receptive field size
            std = np.sqrt(2. / (fan_in + fan_out))
            bound = np.sqrt(3.) * std

            if weight_init.split('_')[1] == 'normal':
                self.W_ij = nn.Parameter(self.W_ij.normal_(0, std))
            elif weight_init.split('_')[1] == 'uniform':
                self.W_ij = nn.Parameter(self.W_ij.uniform_(-bound, bound))
            else:
                raise NotImplementedError('{} not implemented.'.format(weight_init))

        elif weight_init.split('_')[0] == 'kaiming':
            # fan_in preserves magnitude of the variance of the weights in the forward pass.
            fan_in = self.B * self.K*self.K * self.PP # in_caps types * receptive field size
            # fan_out has same affect as fan_in for backward pass.
            # fan_out = self.C * self.K*self.K * self.PP # out_caps types * receptive field size
            std = np.sqrt(2.) / np.sqrt(fan_in)
            bound = np.sqrt(3.) * std

            if weight_init.split('_')[1] == 'normal':
                self.W_ij = nn.Parameter(self.W_ij.normal_(0, std))
            elif weight_init.split('_')[1] == 'uniform':
                self.W_ij = nn.Parameter(self.W_ij.uniform_(-bound, bound))
            else:
                raise NotImplementedError('{} not implemented.'.format(weight_init))
```

```
        elif weight_init == 'noisy_identity' and self.PP > 2:
            b = 0.01 # U(0,b)
            # Out     [1, B, C, 1, P, P, 1, 1, K, K]
            self.W_ij = nn.Parameter(torch.clamp(.1*torch.eye(self.P,self.P).repeat( \
                1, self.B, self.C, 1, 1, 1, self.K, self.K, 1, 1) + \
                torch.empty(1, self.B, self.C, 1, 1, 1, self.K, self.K, self.P, self.P).uniform_(0,b), \
                max=1).permute(0, 1, 2, 3, -2, -1, 4, 5, 6, 7))
        else:
            raise NotImplementedError('{} not implemented.'.format(weight_init))

        if self.padding != 0:
            if isinstance(self.padding, int):
                self.padding = [self.padding]*4

    def forward(self, activations, poses, device): # ([?, B, F, F], [?, B, P, P, F, F])     In

        if self.padding != 0:
            activations = F.pad(activations, self.padding) # [1,1,1,1]
            poses = F.pad(poses, self.padding + [0]*4) # [0,0,1,1,1,1]

        if self.share_W_ij: # share the matrices over (F*F), if class caps layer
            self.K = poses.shape[-1] # out_caps (C) feature map size

        self.F = (poses.shape[-1] - self.K) // self.S + 1 # featuremap size

        # Out     [?, B, P, P, F', F', K, K]     [?, B, P, P, F, F]
        poses = poses.unfold(4, size=self.K, step=self.S).unfold(5, size=self.K, step=self.S)

        # Out     [?, B, 1, P, P, 1, F', F', K, K]     [?, B, P, P, F', F', K, K]
        poses = poses.unsqueeze(2).unsqueeze(5)

        # Out     [?, B, F', F', K, K]     [?, B, F, F]
        activations = activations.unfold(2, size=self.K, step=self.S).unfold(3, size=self.K, step=self.S)

        # Out     [?, B, 1, 1, 1, F', F', K, K]     [?, B, F', F', K, K]
        activations = activations.reshape(-1, self.B, 1, 1, 1, *activations.shape[2:4], self.K, self.K)

        # Out     [?, B, C, P, P, F', F', K, K]     ([?, B, 1, P, P, 1, F', F', K, K] * [1, B, C, 1, P, P, 1, 1, K, K])
        V_ji = (poses * self.W_ij).sum(dim=4) # matmul equiv.

```

```
        # Out      [?, B, C, P*P, 1, F', F', K, K]      [?, B, C, P, P, F', F', K, K]
        V_ji = V_ji.reshape(-1, self.B, self.C, self.P*self.P, 1, *V_ji.shape[-4:-2], self.K, self.K)

        if self.coor_add:
            if V_ji.shape[-1] == 1: # if class caps layer (featuremap size = 1)
                self.F = self.K # 1->4

            # coordinates = torch.arange(self.F, dtype=torch.float32) / self.F
            coordinates = torch.arange(self.F, dtype=torch.float32).add(1.) / (self.F*10)
            i_vals = torch.zeros(self.P*self.P,self.F,1).to(device)
            j_vals = torch.zeros(self.P*self.P,1,self.F).to(device)
            i_vals[self.P-1,:,0] = coordinates
            j_vals[2*self.P-1,0,:] = coordinates

            if V_ji.shape[-1] == 1: # if class caps layer
                # Out      [?, B, C, P*P, 1, 1, 1, K=F, K=F] (class caps)
                V_ji = V_ji + (i_vals + j_vals).reshape(1,1,1,self.P*self.P,1,1,1,self.F,self.F)
                return activations, V_ji

            # Out      [?, B, C, P*P, 1, F, F, K, K]
            V_ji = V_ji + (i_vals + j_vals).reshape(1,1,1,self.P*self.P,1,self.F,self.F,1,1)

        return activations, V_ji
```

Listing 10: src/VarCaps/layers.py

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F

class VariationalBayesRouting2d(nn.Module):
    '''Variational Bayes Capsule Routing Layer'''
    def __init__(self, in_caps, out_caps, pose_dim,
                kernel_size, stride,
                alpha0, # Dirichlet
                m0, kappa0, # Gaussian
                Psi0, nu0, # Wishart
                cov='diag', iter=3, class_caps=False):
        super().__init__()

        self.B = in_caps
        self.C = out_caps
        self.P = pose_dim
        self.D = np.max([2, self.P*self.P])
        self.K = kernel_size
        self.S = stride

        self.cov = cov # diag/full
        self.iter = iter # routing iters
```

```
        self.class_caps = class_caps
        self.n_classes = out_caps if class_caps else None

        # dirichlet prior parameter
        self.alpha0 = torch.tensor(alpha0).type(torch.FloatTensor)
        # self.alpha0 = nn.Parameter(torch.zeros(1,1,self.C,1,1,1,1,1,1).fill_(alpha0)) learn it by backprop

        # Out      [?, 1, C, P*P, 1, 1, 1, 1, 1]
        self.register_buffer('m0', m0.unsqueeze(0).repeat( \
            self.C,1).reshape(1,1,self.C,self.D,1,1,1,1,1)) # gaussian prior mean parameter

        # precision scaling parameter of gaussian prior over capsule component means
        self.kappa0 = kappa0

        # scale matrix of wishart prior over capsule precisions
        if self.cov == 'diag':
            # Out      [?, 1, C, P*P, 1, 1, 1, 1, 1]
            self.register_buffer('Psi0', torch.diag(Psi0).unsqueeze(0).repeat( \
                self.C,1).reshape(1,1,self.C,self.D,1,1,1,1,1))

        elif self.cov == 'full':
            # Out      [?, 1, C, P*P, P*P, 1, 1, 1, 1]
            self.register_buffer('Psi0', Psi0.unsqueeze(0).repeat( \
                self.C,1,1).reshape(1,1,self.C,self.D,self.D,1,1,1,1))

        # degree of freedom parameter of wishart prior capsule precisions
        self.nu0 = nu0

        # log determinant = 0, if Psi0 is identity
        self.register_buffer('lndet_Psi0', 2*torch.diagonal(torch.cholesky(
            Psi0)).log().sum())

        # pre compute the argument of the digamma function in E[ln|lambda_j|]
        self.register_buffer('diga_arg', torch.arange(self.D).reshape(
            1,1,1,self.D,1,1,1,1,1).type(torch.FloatTensor))

        # pre define some constants
        self.register_buffer('Dlog2',
            self.D*torch.log(torch.tensor(2.)).type(torch.FloatTensor))
        self.register_buffer('Dlog2pi',
            self.D*torch.log(torch.tensor(2.*np.pi)).type(torch.FloatTensor))

        # Out      [K*K, 1, K, K] vote collecting filter
        self.register_buffer('filter',
            torch.eye(self.K*self.K).reshape(self.K*self.K,1,self.K
```

```python
,self.K))

        # Out      [1, 1, C, 1, 1, 1, 1, 1, 1] optional params
        self.beta_u = nn.Parameter(torch.zeros(1,1,self.C,1,1,1,1,1,1))
        self.beta_a = nn.Parameter(torch.zeros(1,1,self.C,1,1,1,1,1,1))

        self.BN_v = nn.BatchNorm3d(self.C, affine=False)
        self.BN_a = nn.BatchNorm2d(self.C, affine=False)

    # Out      [?, B, 1, 1, 1, F, F, K, K], [?, B, C, P*P, 1, F, F, K, K]      In
    def forward(self, a_i, V_ji, device):

        self.F_i = a_i.shape[-2:] # input capsule (B) votes feature map size (K)
        self.F_o = a_i.shape[-4:-2] # output capsule (C) feature map size (F)
        self.N = self.B*self.F_i[0]*self.F_i[1] # total num of lower level capsules

        # Out      [1, B, C, 1, 1, 1, 1, 1, 1]
        R_ij = (1./self.C) * torch.ones(1,self.B,self.C,1,1,1,1,1,1, requires_grad=False).to(device)

        for i in range(self.iter): # routing iters

            # update capsule parameter distributions
            self.update_qparam(a_i, V_ji, R_ij)

            if i != self.iter-1: # skip last iter
                # update latent variable distributions (child to parent capsule assignments)
                R_ij = self.update_qlatent(a_i, V_ji)

        # Out      [?, 1, C, 1, 1, F, F, 1, 1]
        self.Elnlambda_j = self.reduce_poses(
            torch.digamma(.5*(self.nu_j - self.diga_arg))) \
                + self.Dlog2 + self.lndet_Psi_j

        # Out      [?, 1, C, 1, 1, F, F, 1, 1]
        self.Elnpi_j = torch.digamma(self.alpha_j) \
            - torch.digamma(self.alpha_j.sum(dim=2, keepdim=True))

        # subtract "- .5*ln|lmbda|" due to precision matrix, instead of adding "+ .5*ln|sigma|" for covariance matrix
        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)]

        # Out      [?, 1, C, 1, 1, F, F, 1, 1] weighted negative entropy with optional beta params and R_j weight
        a_j = self.beta_a - (torch.exp(self.Elnpi_j) * H_q_j + self.beta_u) #* self.R_j

        # Out      [?, C, F, F]
        a_j = a_j.squeeze()
```

```python

        # Out     [?, C, P*P, F, F]     [?, 1, C, P*P, 1, F, F, 1, 1]
        self.m_j = self.m_j.squeeze()

        # so BN works in the classcaps layer
        if self.class_caps:
            # Out     [?, C, 1, 1]     [?, C]
            a_j = a_j[...,None,None]

            # Out     [?, C, P*P, 1, 1]     [?, C, P*P]
            self.m_j = self.m_j[...,None,None]
        # else:
        #     self.m_j = self.BN_v(self.m_j)

        # Out     [?, C, P*P, F, F]
        self.m_j = self.BN_v(self.m_j) # use 'else' above to deactivate BN_v for class_caps

        # Out     [?, C, P, P, F, F]     [?, C, P*P, F, F]
        self.m_j = self.m_j.reshape(-1, self.C, self.P, self.P, *self.F_o)

        # Out     [?, C, F, F]
        a_j = torch.sigmoid(self.BN_a(a_j))

        return a_j.squeeze(), self.m_j.squeeze() # propagate posterior means to next layer

    def update_qparam(self, a_i, V_ji, R_ij):

        # Out     [?, B, C, 1, 1, F, F, K, K]
        R_ij = R_ij * a_i # broadcast a_i 1->C, and R_ij (1,1,1,1)->(F,F,K,K), 1->batch

        # Out     [?, 1, C, 1, 1, F, F, 1, 1]
        self.R_j = self.reduce_icaps(R_ij)

        # Out     [?, 1, C, 1, 1, F, F, 1, 1]
        self.alpha_j = self.alpha0 + self.R_j
        # self.alpha_j = torch.exp(self.alpha0) + self.R_j # when alpha's a param
        self.kappa_j = self.kappa0 + self.R_j
        self.nu_j = self.nu0 + self.R_j

        # Out     [?, 1, C, P*P, 1, F, F, 1, 1]
        mu_j = (1./self.R_j) * self.reduce_icaps(R_ij * V_ji)

        # Out     [?, 1, C, P*P, 1, F, F, 1, 1]
        # self.m_j = (1./self.kappa_j) * (self.R_j * mu_j + self.kappa0 * self.m0) # use this if self.m0 != 0
        self.m_j = (1./self.kappa_j) * (self.R_j * mu_j) # priors removed for faster computation

        if self.cov == 'diag':
            # Out     [?, 1, C, P*P, 1, F, F, 1, 1] (1./R_j) not needed because Psi_j calc
```

```
            sigma_j = self.reduce_icaps(R_ij * (V_ji - mu_j).pow(2))

            # Out     [?, 1, C, P*P, 1, F, F, 1, 1]
            # self.invPsi_j = self.Psi0 + sigma_j + (self.kappa0*self.R_j / self.kappa_j) \
            #     * (mu_j - self.m0).pow(2) # use this if m0 != 0 or kappa0 != 1
            self.invPsi_j = self.Psi0 + sigma_j + (self.R_j / self.kappa_j) * (mu_j).pow(2) # priors removed for faster computation

            # Out     [?, 1, C, 1, 1, F, F, 1, 1] (-) sign as inv. Psi_j
            self.lndet_Psi_j = -self.reduce_poses(torch.log(self.invPsi_j)) # log det of diag precision matrix

        elif self.cov == 'full':
            #[?, B, C, P*P, P*P, F, F, K, K]
            sigma_j = self.reduce_icaps(
                R_ij * (V_ji - mu_j) * (V_ji - mu_j).transpose(3,4))

            # Out     [?, 1, C, P*P, P*P, F, F, 1, 1] full cov, torch.inverse(self.Psi0)
            self.invPsi_j = self.Psi0 + sigma_j + (self.kappa0*self.R_j / self.kappa_j) \
                * (mu_j - self.m0) * (mu_j - self.m0).transpose(3,4)

            # Out     [?, 1, C, F, F, 1, 1 , P*P, P*P]
            # needed for pytorch (*,n,n) dim requirements in .cholesky and .inverse
            self.invPsi_j = self.invPsi_j.permute(0,1,2,5,6,7,8,3,4)

            # Out     [?, 1, 1, 1, C, F, F, 1, 1] (-) sign as inv. Psi_j
            self.lndet_Psi_j = -2*torch.diagonal(torch.cholesky(
                self.invPsi_j), dim1=-2, dim2=-1).log().sum(-1, keepdim=True)[...,None]

    def update_qlatent(self, a_i, V_ji):

        # Out     [?, 1, C, 1, 1, F, F, 1, 1]
        self.Elnpi_j = torch.digamma(self.alpha_j) \
            - torch.digamma(self.alpha_j.sum(dim=2, keepdim=True))

        # Out     [?, 1, C, 1, 1, F, F, 1, 1] broadcasting diga_arg
        self.Elnlambda_j = self.reduce_poses(
            torch.digamma(.5*(self.nu_j - self.diga_arg))) \
                + self.Dlog2 + self.lndet_Psi_j

        if self.cov == 'diag':
            # Out     [?, B, C, 1, 1, F, F, K, K]
            ElnQ = (self.D/self.kappa_j) + self.nu_j \
                * self.reduce_poses((1./self.invPsi_j) * (V_ji -
```

```
    self.m_j).pow(2))

        elif self.cov == 'full':
            # Out      [?, B, C, 1, 1, F, F, K, K]
            Vm_j = V_ji - self.m_j
            ElnQ = (self.D/self.kappa_j) + self.nu_j * self.reduce_poses(
                Vm_j.transpose(3,4) * torch.inverse(
                    self.invPsi_j).permute(0,1,2,7,8,3,4,5,6) * Vm_j)

        # Out      [?, B, C, 1, 1, F, F, K, K]
        lnp_j = .5*self.Elnlambda_j -.5*self.Dlog2pi -.5*ElnQ

        # Out      [?, B, C, 1, 1, F, F, K, K]
        p_j = torch.exp(self.Elnpi_j + lnp_j)

        # Out      [?*B, 1, F', F']      [?*B, K*K, F, F]      [?, B, 1, 1, 1, F, F, K, K]
        sum_p_j = F.conv_transpose2d(
            input=p_j.sum(dim=2, keepdim=True).reshape(
                -1, *self.F_o, self.K*self.K).permute(0, -1, 1, 2).contiguous(),
            weight=self.filter,
            stride=[self.S, self.S])

        # Out      [?*B, 1, F, F, K, K]
        sum_p_j = sum_p_j.unfold(2, size=self.K, step=self.S).unfold(3, size=self.K, step=self.S)

        # Out      [?, B, 1, 1, 1, F, F, K, K]
        sum_p_j = sum_p_j.reshape([-1, self.B, 1, 1, 1, *self.F_o, self.K, self.K])

        # Out      [?, B, C, 1, 1, F, F, K, K] # normalise over out_caps j
        return 1. / torch.clamp(sum_p_j, min=1e-11) * p_j

    def reduce_icaps(self, x):
        return x.sum(dim=(1,-2,-1), keepdim=True)

    def reduce_poses(self, x):
        return x.sum(dim=(3,4), keepdim=True)
```

Listing 11: src/VarCaps/vb$_{r}$*outing.py*