ckingkan / AT82.08-computer-vision

Code Issues Pull requests 1 Actions Projects Security Insights

AT82.08-computer-vision / labs_instructor_copies
/ lab10.2_Segmentation_Deep_learning.ipynb

beep-love lab for segmentation and generative algorithms 8bbc51c · 3 weeks ago

765 lines (765 loc) · 24.1 KB

Preview Code Blame Raw

UNET IMPLEMENTATION ON ROAD SEGMENT DATA

Import Libraries

In [16]:
```
import os
import torch
from torch_snippets import *
from sklearn.model_selection import train_test_split
from torchvision.models import vgg16_bn
import cv2 as cv
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader, random_split
from PIL import Image
import torchvision.transforms as T
import matplotlib.pyplot as plt
```

Dataset taken from here: https://www.kaggle.com/datasets/sanadalali/satelli images-for-road-segmentation

In [2]:
```
class RoadSegDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.image_names = sorted(os.listdir(image_dir))
        self.mask_names = sorted(os.listdir(mask_dir))
        self.transform = transform or T.Compose([
            T.Resize((256, 256)),
            T.ToTensor()
        ])
        self.mask_transform = T.Compose([
            T.Resize((256, 256)),
            T.ToTensor()
        ])

    def __len__(self):
        return len(self.image_names)

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_dir, self.image_names[idx])
        mask_path = os.path.join(self.mask_dir, self.mask_names[idx])

        image = Image.open(img_path).convert("RGB")
        mask = Image.open(mask_path).convert("L")

        image = self.transform(image)
        mask = self.mask_transform(mask)
        mask = (mask > 0.5).float()  # binary mask 0/1
```

```
        mask = (mask > 0.5).float()  # binary mask 0/1

        return image, mask


def get_dataloaders(image_dir, mask_dir, batch_size=8, val_split=0.2, seed=42):
    dataset = RoadSegDataset(image_dir, mask_dir)

    # Split sizes
    val_size = int(len(dataset) * val_split)
    train_size = len(dataset) - val_size

    # Reproducible random split
    torch.manual_seed(seed)
    train_set, val_set = random_split(dataset, [train_size, val_size])

    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False)

    return train_loader, val_loader


# Example usage:
if __name__ == "__main__":
    train_loader, val_loader = get_dataloaders(
        "dataset/training/images", "dataset/training/groundtruth",
        batch_size=8,
        val_split=0.2
    )

    for imgs, masks in train_loader:
        print("Train batch:", imgs.shape, masks.shape)
        break

    for imgs, masks in val_loader:
        print("Valid batch:", imgs.shape, masks.shape)
        break
```

```
Train batch: torch.Size([8, 3, 256, 256]) torch.Size([8, 1, 256, 256])
Valid batch: torch.Size([8, 3, 256, 256]) torch.Size([8, 1, 256, 256])
```

In [18]:

```
# Choose device - change (cuda:2 --> cuda) in case you dont have multiple gpu
device = torch.device("cuda:2" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

```
Using device: cuda:2
```

In [4]:

```
# Create Params dictionary
class Params(object):
    def __init__(self, batch_size, test_batch_size, epochs, lr, seed, cuda, lo
        self.batch_size = batch_size
        self.test_batch_size = test_batch_size
        self.epochs = epochs
        self.lr = lr
        self.seed = seed
        self.cuda = 'cuda:2' if cuda and torch.cuda.is_available() else 'cpu'
        self.log_interval = log_interval
```

```
# Configure args
args = Params(8, 2, 5, 1e-3, 1, True, 10)
```

U Net Architecture

- U-Net is a convolutional neural network architecture primarily used for image segmentation tasks. It consists of a contracting path (encoder) that captures context and a symmetric expanding path (decoder) that enables precise localization.

Steps

Initializes the U-Net model.

Takes two parameters:

```
- pretrained: A boolean indicating whether to use a pretrained
VGG16 model.
- out_channels: The number of output channels for the final
segmentation map (e.g., 12 classes for segmentation).
```

Encoder (Contractive Path)

```
- The encoder part uses the features from a VGG16 model with
batch normalization (vgg16_bn).
- The encoder is divided into five blocks, each consisting of
several convolutional layers that progressively reduce the
spatial dimensions while increasing the number of feature
channels.
```

Bottleneck

```
- The bottleneck section takes the deepest layers of the
encoder.
- A convolution layer `conv_bottleneck` is applied to increase
the number of feature channels from 512 to 1024, allowing the
network to learn more complex features.
```

Decoder (Expansive Path)

The decoder consists of up-convolution (or transposed convolution) layers followed by concatenation with corresponding encoder features to retain spatial information:

```
- Each up_conv layer increases the spatial dimensions
(upsampling).
```

```
- The output of each up-convolution is concatenated with the
corresponding feature map from the encoder (skip connections).
- This helps the model learn both high-level features from
deeper layers and low-level features from shallower layers.
- Finally, conv11 reduces the number of channels to
out_channels (e.g., for multi-class segmentation).
```

In [5]:

```
def conv(in_channels, out_channels):
    return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    )
```

In [6]:

```
def up_conv(in_channels, out_channels):
    return nn.Sequential(
        nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        nn.ReLU(inplace=True)
    )
```

In [7]:

```
import torch.nn as nn
class UNet(nn.Module):
    def __init__(self, pretrained=True, out_channels=12):
        super().__init__()

        self.encoder = vgg16_bn(pretrained=pretrained).features
        self.block1 = nn.Sequential(*self.encoder[:6])
        self.block2 = nn.Sequential(*self.encoder[6:13])
        self.block3 = nn.Sequential(*self.encoder[13:20])
        self.block4 = nn.Sequential(*self.encoder[20:27])
        self.block5 = nn.Sequential(*self.encoder[27:34])

        self.bottleneck = nn.Sequential(*self.encoder[34:])
        self.conv_bottleneck = conv(512, 1024)

        self.up_conv6 = up_conv(1024, 512)
        self.conv6 = conv(512 + 512, 512)
        self.up_conv7 = up_conv(512, 256)
        self.conv7 = conv(256 + 512, 256)
        self.up_conv8 = up_conv(256, 128)
        self.conv8 = conv(128 + 256, 128)
        self.up_conv9 = up_conv(128, 64)
        self.conv9 = conv(64 + 128, 64)
        self.up_conv10 = up_conv(64, 32)
        self.conv10 = conv(32 + 64, 32)
        self.conv11 = nn.Conv2d(32, 3, kernel_size=1)

    def forward(self, x):
        # Contractive Path
        block1 = self.block1(x)
        block2 = self.block2(block1)
        block3 = self.block3(block2)
        block4 = self.block4(block3)
        block5 = self.block5(block4)
```

```
        bottleneck = self.bottleneck(block5)
        x = self.conv_bottleneck(bottleneck)
        # Expansive Path
        x = self.up_conv6(x)
        x = torch.cat([x, block5], dim=1)
        x = self.conv6(x)

        x = self.up_conv7(x)
        x = torch.cat([x, block4], dim=1)
        x = self.conv7(x)

        x = self.up_conv8(x)
        x = torch.cat([x, block3], dim=1)
        x = self.conv8(x)

        x = self.up_conv9(x)
        x = torch.cat([x, block2], dim=1)
        x = self.conv9(x)

        x = self.up_conv10(x)
        x = torch.cat([x, block1], dim=1)
        x = self.conv10(x)

        x = self.conv11(x)

        return x
```

In [8]:

```
vgg16_bn
```

Out[8]:

```
<function torchvision.models.vgg.vgg16_bn(*, weights: Optional[torchvision.mod
els.vgg.VGG16_BN_Weights] = None, progress: bool = True, **kwargs: Any) -> tor
chvision.models.vgg.VGG>
```

In [9]:

```
ce = nn.CrossEntropyLoss()   # Applies softmax to output logits --> converts i

def UnetLoss(preds, targets):
    targets = targets.squeeze(1).long()   # <-- remove the channel dimension
    ce_loss = ce(preds, targets)
    acc = (torch.max(preds, 1)[1] == targets).float().mean()
    #  (torch.max(preds, 1)[1] returns the indices of the maximum values along
    #  The 1 indicates that we're looking along the columns (the class dimensi
    #  if preds class == targets return 1 --> change to float --> take mean to
    return ce_loss, acc
```

In [10]:

```
class TrainEngine():
    def train_batch(model, data, optimizer, criterion):
        model.train()
        for imgs, masks in data:
            ims, ce_masks = imgs.to(device), masks.to(device)

        _masks = model(ims)
        optimizer.zero_grad()

        loss, acc = criterion(_masks, ce_masks)
        loss.backward()
```

```
        loss.backward()
        optimizer.step()

        return loss.item(), acc.item()

    @torch.no_grad()
    def validate_batch(model, data, criterion):
        model.eval()
        for imgs, masks in data:
            ims, ce_masks = imgs.to(device), masks.to(device)

        _masks = model(ims)

        loss, acc = criterion(_masks, ce_masks)

        return loss.item(), acc.item()
```

In [11]:

```
from torch import optim
def make_model():
    model = UNet().to(args.cuda)
    criterion = UnetLoss
    optimizer = optim.Adam(model.parameters(), lr=args.lr)
    return model, criterion, optimizer
```

In [12]:

```
model, criterion, optimizer = make_model()
# Total num. of parametes
num_params = sum(p.numel() for p in model.parameters())
# Total num. of "trainable" parameters
num_trainable_params = sum(p.numel() for p in model.parameters() if p.requires
print(f'Total num. of parametes: {num_params}')
print(f'Total num. of Trainable parametes: {num_trainable_params}')
```

```
/home/jupyter-dsai-st123439/.local/lib/python3.12/site-packages/torchvision/mod
els/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/jupyter-dsai-st123439/.local/lib/python3.12/site-packages/torchvision/mod
els/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` fo
r 'weights' are deprecated since 0.13 and may be removed in the future. The cur
rent behavior is equivalent to passing `weights=VGG16_BN_Weights.IMAGENET1K_V1
`. You can also use `weights=VGG16_BN_Weights.DEFAULT` to get the most up-to-da
te weights.
  warnings.warn(msg)
Total num. of parametes: 29311011
Total num. of Trainable parametes: 29311011
```

In [13]:

```
def run_model():
    for epoch in range(args.epochs):
        print("####################")
        print(f"       Epoch: {epoch}   ")
        print("####################")

        for batch_idx, data in tqdm(enumerate(train_loader), total=len(train_l
            train_loss, train_acc = TrainEngine.train_batch(model, train_loade
            if batch_idx % args.log_interval == 0:
                # Print training information inline instead of calling a funct
```

```
                step = epoch * len(train_loader) + batch_idx
                print(f'Epoch [{epoch+1}/{args.epochs}], Step [{batch_idx}/{le
                      f'Train Loss: {train_loss:.6f}, Accuracy: {train_acc:.6f

        avg_val_acc = avg_val_loss = 0.0
        for batch_idx, data in tqdm(enumerate(val_loader), total=len(val_loade
            val_loss, val_acc = TrainEngine.validate_batch(model, val_loader,

            avg_val_loss += val_loss
            avg_val_acc += val_acc

        step = (epoch + 1) * len(train_loader)
        avg_val_loss /= len(val_loader)
        avg_val_acc /= len(val_loader)
        print(f'Val: Average loss: {avg_val_loss:.4f}, Accuracy: {avg_val_acc:
        print()

    # Save the model and optimizer states after training is complete
    # torch.save({
    #     'model_state_dict': model.state_dict(),
    #     'optimizer_state_dict': optimizer.state_dict()
    # }, 'unet.pt')
```

In [14]:

```
# Train the model
run_model()
```

```
####################
       Epoch: 0
####################
 10%|▉         | 1/10 [00:01<00:17,  1.91s/it]
Epoch [1/5], Step [0/10], Train Loss: 1.040541, Accuracy: 0.454958
100%|██████████| 3/3 [00:01<00:00,  2.47it/s]
Val: Average loss: 1.0813, Accuracy: 0.2351

####################
       Epoch: 1
####################
 10%|▉         | 1/10 [00:01<00:12,  1.43s/it]
Epoch [2/5], Step [0/10], Train Loss: 0.667452, Accuracy: 0.864990
100%|██████████| 3/3 [00:01<00:00,  2.62it/s]
Val: Average loss: 1.0531, Accuracy: 0.3124

####################
       Epoch: 2
####################
 10%|▉         | 1/10 [00:01<00:11,  1.26s/it]
Epoch [3/5], Step [0/10], Train Loss: 0.504151, Accuracy: 0.922363
100%|██████████| 3/3 [00:01<00:00,  2.64it/s]
Val: Average loss: 0.6757, Accuracy: 0.8655

####################
       Epoch: 3
####################
 10%|▉         | 1/10 [00:01<00:13,  1.50s/it]
Epoch [4/5], Step [0/10], Train Loss: 0.433789, Accuracy: 0.937992
```

```
100%|██████████| 3/3 [00:01<00:00,  2.84it/s]
Val: Average loss: 0.3127, Accuracy: 0.9373

####################
       Epoch: 4
####################
 10%|▉         | 1/10 [00:01<00:12,  1.38s/it]
Epoch [5/5], Step [0/10], Train Loss: 0.356298, Accuracy: 0.929749
100%|██████████| 3/3 [00:01<00:00,  2.64it/s]
Val: Average loss: 0.2798, Accuracy: 0.9345
```

In [19]:

```python
import matplotlib.pyplot as plt

model.eval()
with torch.no_grad():
    for bx, (imgs, masks) in tqdm(enumerate(val_loader), total=len(val_loader)
        ims, ce_masks = imgs.to(device), masks.to(device)

        # forward pass
        preds = model(ims)
        preds = torch.max(preds, dim=1)[1]  # predicted class per pixel, shape

        # move to CPU for visualization
        img_cpu = ims[0].permute(1, 2, 0).cpu()        # RGB image [H,W,3]
        mask_cpu = ce_masks[0, 0].cpu()                # Ground truth [H,W]
        pred_cpu = preds[0].cpu()                      # Prediction [H,W]

        # plot original RGB image
        plt.figure(figsize=(10, 10))
        plt.axis('off')
        plt.imshow(img_cpu)  # RGB image
        plt.savefig(f'original_image_{bx}.jpg')
        plt.close()
```