به نام خداوند جان و خرد

توضیح ناز یلی پروژه

```python
def load_images_and_masks(images_path, masks_path, image_size):
    """
    Loads and preprocesses images and their corresponding masks.

    Args:
        images_path (str): Path to the directory containing input images.
        masks_path (str): Path to the directory containing corresponding masks.
        image_size (tuple): Desired size (width, height) to resize images and masks.

    Returns:
        images (np.array or list): Loaded and resized images.
        masks (np.array or list): Loaded and resized masks.
    """
    images = []
    masks = []
    for img_file in images_path:
        images.append((Image.open(img_file).convert('RGB')).resize(image_size, Image.BILINEAR))
    images = np.array(images)

    for mask_file in masks_path:
        mask_of_correct_size = Image.open(mask_file).resize(image_size)
        final_mask = (np.array(mask_of_correct_size) > 1).astype(np.bool_)
        masks.append(final_mask)

    masks = np.array(masks)
    #masks = np.where(masks > 1).astype(np.uint8)

    return images, masks
```

اگه مطمئنیم تصویرها قراره RGB هستن نیاز نیست

PIL Image object (ولی shape)

وقتی تصویر رو Resize می‌کنیم، باید پیکسل‌های جدید از روی تصویرهای قبلی [illegible] یا پیکسل نزدیک به پیکسل جدید رو مرتبه و میانگین می‌گیره

تبدیل تصویر smooth مرتبه و جزئیات حفظ میشه

عکس رو به boolean صفر و یک تبدیل کردم. در ادامه کار عددی باهاش بشه مثل صفر و یک رفتار می‌کنه باهاش

* PIL Image اگه RGB باشه، هر پیکسلش در هر کانال 8 بیتیه

* البته mask هم 8 بیتیه پیکسل‌هاش ولی فقط 0, 255 می‌گیره

* وقتی برای resize کردن maskها، نوع convert رو مشخص نکردم، از پیش‌فرض PIL استفاده کرده که اگه اشتباه نکنم مدل Nearest باشه.

* وقتی images رو np.array تبدیل کردیم، می‌خواد عکس‌ها رو به صورت آرایه چندبعدی ذخیره کنه، پس خودش عکس‌ها رو به np.array تبدیل میکنه.

```python
# Define training image and mask directories
train_images_dir = os.path.join(DATA_DIR, 'tiff/train')
train_masks_dir = os.path.join(DATA_DIR, 'tiff/train_labels')

# Get paths or filenames for training images and masks (placeholder)
train_images_path = [os.path.join(train_images_dir, file_name) for file_name in sorted(os.listdir(train_images_dir))]
train_masks_path = [os.path.join(train_masks_dir, file_name) for file_name in sorted(os.listdir(train_masks_dir))]

# Load and preprocess training images and masks
train_images, train_masks = load_images_and_masks(train_images_path, train_masks_path, IMAGE_SIZE)

# Define validation image and mask directories
val_images_dir = os.path.join(DATA_DIR, 'tiff/val')
val_masks_dir = os.path.join(DATA_DIR, 'tiff/val_labels')

# Get paths or filenames for validation images and masks (placeholder)
val_images_path = [os.path.join(val_images_dir, file_name) for file_name in sorted(os.listdir(val_images_dir))]
val_masks_path = [os.path.join(val_masks_dir, file_name) for file_name in sorted(os.listdir(val_masks_dir))]

# Load and preprocess validation images and masks
val_images, val_masks = load_images_and_masks(val_images_path, val_masks_path, IMAGE_SIZE)
```

train آدرس پوشه تصاویر

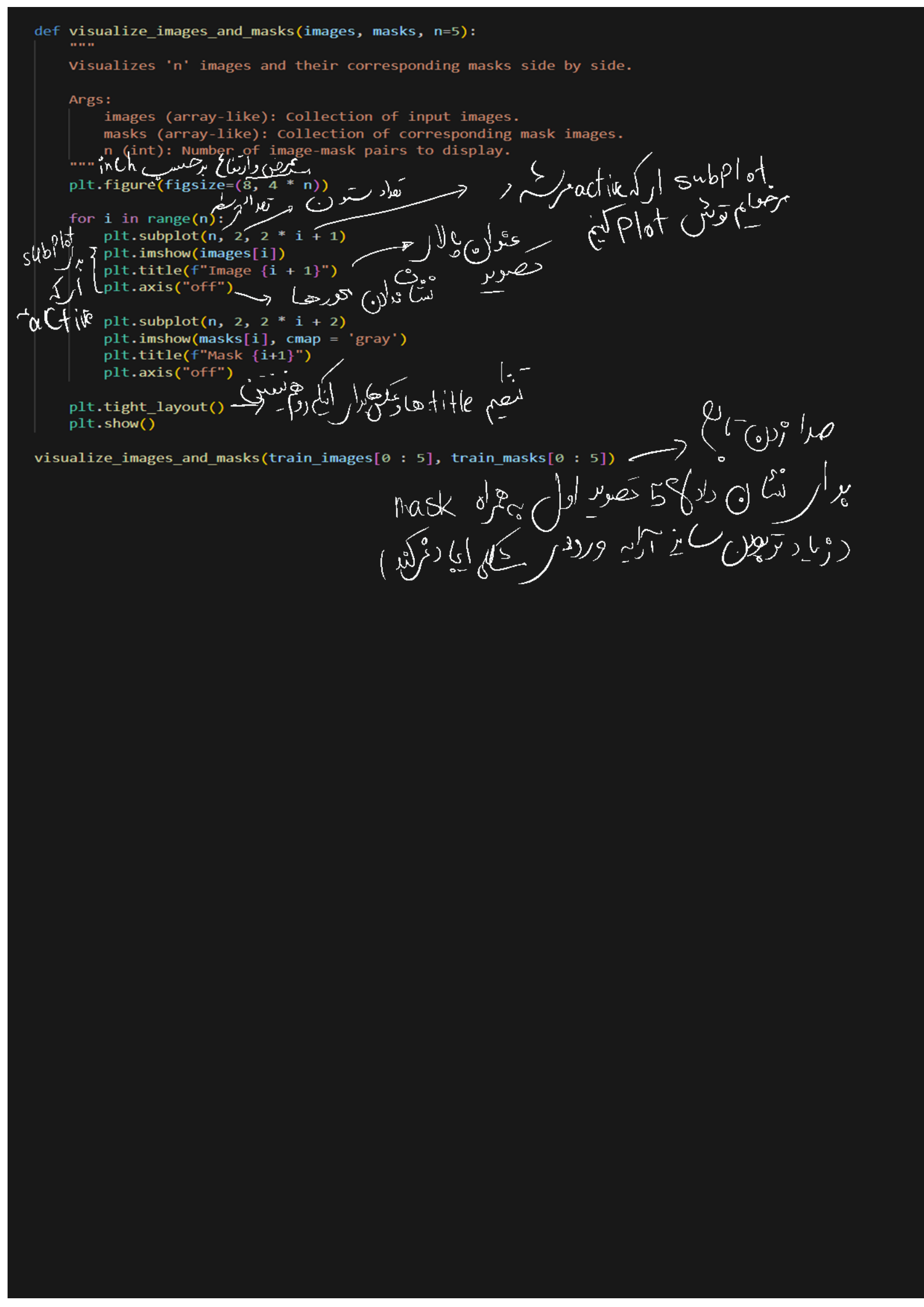
آدرس پوشه train labelها

آدرس فایل ها

چون اسم هر تصویر با خودش یکی بود، با مرتب کردن اسما، مطمئن میشیم درایو های یکسان این دو آرایه به جفت تصویر و ماسک یکسانی اشاره میکنن

استفاده از تابع صفحه قبل باعث بدست آوردن mask و image ها به صورت nparray

```python
def visualize_images_and_masks(images, masks, n=5):
    """
    Visualizes 'n' images and their corresponding masks side by side.

    Args:
        images (array-like): Collection of input images.
        masks (array-like): Collection of corresponding mask images.
        n (int): Number of image-mask pairs to display.
    """
    plt.figure(figsize=(8, 4 * n))

    for i in range(n):
        plt.subplot(n, 2, 2 * i + 1)
        plt.imshow(images[i])
        plt.title(f"Image {i + 1}")
        plt.axis("off")

        plt.subplot(n, 2, 2 * i + 2)
        plt.imshow(masks[i], cmap = 'gray')
        plt.title(f"Mask {i+1}")
        plt.axis("off")

    plt.tight_layout()
    plt.show()

visualize_images_and_masks(train_images[0 : 5], train_masks[0 : 5])
```



```python
class RoadDataset(Dataset):
    """
    Custom Dataset class for loading road images and their corresponding masks.

    Args:
        images (array-like): List or array of input images.
        masks (array-like): List or array of corresponding masks.
        image_transform (callable, optional): Transformation to apply to the images.
        mask_transform (callable, optional): Transformation to apply to the masks.
    """
    def __init__(self, images, masks, image_transform=None, mask_transform=None):
        self.images = images
        self.masks = masks
        self.image_transform = image_transform
        self.mask_transform = mask_transform

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

    def __getitem__(self, idx):
        image = self.images[idx]
        mask = self.masks[idx]

        if self.image_transform:
            image = self.image_transform(Image.fromarray(image))
        if self.mask_transform:
            mask = self.mask_transform(Image.fromarray(mask))

        return image, mask
```

```python
### Define your desired transformations
image_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std =[0.229, 0.224, 0.225]),
])

mask_transform = transforms.Compose([
    transforms.ToTensor(),
])
```

```python
train_dataset = RoadDataset(train_images, train_masks, image_transform, mask_transform)
val_dataset   = RoadDataset(val_images, val_masks, image_transform, mask_transform)

train_loader  = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
val_loader    = DataLoader(val_dataset,   batch_size=BATCH_SIZE, shuffle=False,num_workers=NUM_WORKERS)
```

```python
class ConvBlock(nn.Module):
    """
    A convolutional block consisting of multiple layers (e.g., Conv2D, BatchNorm, ReLU).

    Args:
        in_channel (int): Number of input channels.
        out_channel (int): Number of output channels.
    """
    def __init__(self, in_channel, out_channel):
        super().__init__()
        self.conv = nn.Sequential(
            #first convolution -> Normalize -> ReLU
            nn.Conv2d(in_channel, out_channel, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU(inplace = True),
            #second convolution -> Normalize -> ReLU
            nn.Conv2d(out_channel, out_channel, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU(inplace = True),
        )

    def forward(self, x):
        return self.conv(x)
```

چون از nn.Module ارث‌بری کرده باید forward رو implement کنیم

چون batchnorm میاد shift و bias رو خنثی می‌کنه

وقتی مرتب می‌کنیم nn.sequential کارهای A و B و C رو به ترتیب انجام می‌دهد که اینجا inplace: True است:

```
temp = A(input)
temp = B(temp)
output = C(temp)
```

وقتی مرتب می‌کنیم forward صدا زده می‌شود

```
block = ConvBlock(....)
output = block(X)
```

* وقتی مرتب می‌کنیم مرده.

```python
class UpConvBlock(nn.Module):
    """
    An upsampling block used in decoder parts of segmentation networks like U-Net.

    Args:
        in_channel (int): Number of input channels.
        out_channel (int): Number of output channels.
    """
    def __init__(self, in_channel, out_channel):
        super().__init__()
        self.upconv = nn.Sequential(
            nn.ConvTranspose2d(in_channel, out_channel, kernel_size=2, stride=2)
        )

    def forward(self, x):
        return self.upconv(x)
```

```python
class UNet(nn.Module):
    """
    U-Net architecture for image segmentation tasks.

    Args:
        in_channel (int): Number of input channels (e.g., 3 for RGB images).
        out_channel (int): Number of output channels (e.g., 1 for binary segmentation).
        filter (list): List of filter sizes for each level of the encoder/decoder.
    """
    def __init__(self, in_channel=3, out_channel=1, filter=[64, 128, 256, 512]):
        super().__init__()
        #first encoding layer
        self.encode_conv1 = ConvBlock(in_channel, filter[0])
        self.pool1 = nn.MaxPool2d(kernel_size=2 , stride=2)
        #second encoding layer
        self.encode_conv2 = ConvBlock(filter[0], filter[1])
        self.pool2 = nn.MaxPool2d(kernel_size=2 , stride=2)
        #third encoding layer
        self.encode_conv3 = ConvBlock(filter[1], filter[2])
        self.pool3 = nn.MaxPool2d(kernel_size=2 , stride=2)
        #fourth encoding layer
        self.encode_conv4 = ConvBlock(filter[2], filter[3])
        self.pool4 = nn.MaxPool2d(kernel_size=2 , stride=2)
        #bottleneck
        self.bottle_conv = ConvBlock(filter[3], filter[3] * 2)
        #fourth(deepest) upsampling layer
        self.up_conv4 = UpConvBlock(filter[3] * 2, filter[3])
        self.decode_conv4 = ConvBlock(filter[3] * 2, filter[3])
        #third upsampling layer
        self.up_conv3 = UpConvBlock(filter[3], filter[2])
        self.decode_conv3 = ConvBlock(filter[3], filter[2])
        #second upsampling layer
        self.up_conv2 = UpConvBlock(filter[2], filter[1])
        self.decode_conv2 = ConvBlock(filter[2], filter[1])
        #first upsampling layer
        self.up_conv1 = UpConvBlock(filter[1], filter[0])
        self.decode_conv1 = ConvBlock(filter[1], filter[0])
        #the last convolution
        self.final_conv = nn.Conv2d(filter[0], out_channel, kernel_size = 1)
    def forward(self, x):
        #encode path
        encode1 = self.encode_conv1(x)
        pool1 = self.pool1(encode1)
        encode2 = self.encode_conv2(pool1)
        pool2 = self.pool2(encode2)
        encode3 = self.encode_conv3(pool2)
        pool3 = self.pool3(encode3)
        encode4 = self.encode_conv4(pool3)
        pool4 = self.pool4(encode4)
        #bottleneck
        bottleneck = self.bottle_conv(pool4)
        #decode path
        up4 = self.up_conv4(bottleneck)
        decode4 = self.decode_conv4(torch.cat([encode4, up4], dim=1))
        up3 = self.up_conv3(decode4)
        decode3 = self.decode_conv3(torch.cat([encode3, up3], dim=1))
        up2 = self.up_conv2(decode3)
        decode2 = self.decode_conv2(torch.cat([encode2, up2], dim=1))
        up1 = self.up_conv1(decode2)
        decode1 = self.decode_conv1(torch.cat([encode1, up1], dim=1))
        #output
        return self.final_conv(decode1)
```

مشخص کنه اون کدوم dim هم
پشش، باید از چه کانال هم وصل شن Cat
که چون shape به صورت (N, C, H, W) هست
dim = 1 میزنیم

```python
class AttentionBlock(nn.Module):
    """
    Attention block for focusing on relevant features in skip connections.

    Args:
        f_g (int): Number of channels in the gating signal (from decoder).
        f_l (int): Number of channels in the skip connection input (from encoder).
        f_int (int): Number of intermediate channels used within the attention blo
    """
    def __init__(self, f_g, f_l, f_int):
        super().__init__()

        self.w_g = nn.Sequential(
            nn.Conv2d(f_g, f_int, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(f_int)
        )

        self.w_x = nn.Sequential(
            nn.Conv2d(f_l, f_int, kernel_size=1, stride=2, padding=0, bias=False),
            nn.BatchNorm2d(f_int)
        )

        self.psi = nn.Sequential(
            nn.Conv2d(f_int, 1, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(1),
            nn.Sigmoid()
        )

        self.act = nn.ReLU(inplace=True)


    def forward(self, g, x):
        wg = self.w_g(g)
        wx = self.w_x(x)
        after_relu = self.act(wg + wx)
        alpha = self.psi(after_relu)
        #bilinear interpolation to resize back to x's size
        alpha = F.interpolate(alpha, size=x.shape[2:], mode='bilinear', align_corners=True)
        return x * alpha
```

مربوط به لایه پایینی decoder

هر دو باید f_int کانال باشن

مربوط به خروجی متناظر encoder، ابعاد هر کانال رو با stride=2 نصف میکنه تا اندازه بالایی شه

منظور x's H, W

این alpha تک کانالی در همه کانال‌های x ضرب میشه

0,0

input (H_in-1, W_in-1) و output

در این حالت:

$$x = u \cdot \frac{W_{in} - 1}{W_{out} - 1}$$

$$y = v \cdot \frac{H_{in} - 1}{H_{out} - 1}$$

$$i = \lfloor x \rfloor, \quad j = \lfloor y \rfloor$$

$$dx = x - i, \quad dy = y - j$$

$$\text{output}(v, u) = (1 - dx)(1 - dy) \cdot \text{top-left} + dx(1 - dy) \cdot \text{top-right} + (1 - dx)dy \cdot \text{bottom-left} + dxdy \cdot \text{bottom-right}$$

- Top-left: `(i,j)`
- Top-right: `(i+1,j)`
- Bottom-left: `(i,j+1)`
- Bottom-right: `(i+1,j+1)`

```python
class AttentionUNet(UNet):  # Inherits from UNet
    """
    Attention U-Net architecture which extends the basic U-Net
    by integrating attention blocks in the skip connections.

    Args:
        in_channel (int): Number of input channels.
        out_channel (int): Number of output channels.
        filter_sizes (list): List of filter sizes for each level.
    """
    def __init__(self, in_channel=3, out_channel=1, filter_sizes=[64, 128, 256, 512]):
        super().__init__(in_channel, out_channel, filter_sizes)
        self.attention1 = AttentionBlock(filter_sizes[1], filter_sizes[0], filter_sizes[0])
        self.attention2 = AttentionBlock(filter_sizes[2], filter_sizes[1], filter_sizes[1])
        self.attention3 = AttentionBlock(filter_sizes[3], filter_sizes[2], filter_sizes[2])
        self.attention4 = AttentionBlock(filter_sizes[3] * 2, filter_sizes[3], filter_sizes[3])
    def forward(self, x):
      #encode path
      encode1 = self.encode_conv1(x)
      pool1 = self.pool1(encode1)
      encode2 = self.encode_conv2(pool1)
      pool2 = self.pool2(encode2)
      encode3 = self.encode_conv3(pool2)
      pool3 = self.pool3(encode3)
      encode4 = self.encode_conv4(pool3)
      pool4 = self.pool4(encode4)
      #bottleneck
      bottleneck = self.bottle_conv(pool4)
      #decode path
      attention_output4 = self.attention4(bottleneck, encode4)
      up4 = self.up_conv4(bottleneck)
      decode4 = self.decode_conv4(torch.cat([attention_output4, up4], dim=1))

      attention_output3 = self.attention3(decode4, encode3)
      up3 = self.up_conv3(decode4)
      decode3 = self.decode_conv3(torch.cat([attention_output3, up3], dim=1))

      attention_output2 = self.attention2(decode3, encode2)
      up2 = self.up_conv2(decode3)
      decode2 = self.decode_conv2(torch.cat([attention_output2, up2], dim=1))

      attention_output1 = self.attention1(decode2, encode1)
      up1 = self.up_conv1(decode2)
      decode1 = self.decode_conv1(torch.cat([attention_output1, up1], dim=1))
       #output
      return self.final_conv(decode1)
```

```python
class ResidualConvBlock(nn.Module):
    """
    Residual convolutional block with skip connection.

    Args:
        in_channel (int): Number of input channels.
        out_channel (int): Number of output channels.
    """
    def __init__(self, in_channel, out_channel):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channel, out_channel, kernel_size = 3, stride = 1, padding = 1, bias = False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU(inplace = True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channel, out_channel, kernel_size = 3, stride = 1, padding = 1, bias = False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU(inplace = True)
        )
        self.shortcut = nn.Sequential(
            nn.Conv2d(in_channel, out_channel, kernel_size = 3, stride = 1, padding = 1, bias = False),
            nn.BatchNorm2d(out_channel)
        ) if in_channel != out_channel else nn.Identity()

    def forward(self, x):
        to_be_added = self.shortcut(x)
        output = self.conv1(x)
        output = self.conv2(output)
        return F.relu(output + to_be_added)
```

```python
class ResidualAttentionUNet(AttentionUNet):
    """
    Residual Attention U-Net combining residual blocks with attention mechanisms.

    Args:
        in_channel (int): Number of input channels.
        out_channel (int): Number of output channels.
        filter_sizes (list): List of filter sizes for each level.
    """
    def __init__(self, in_channel=3, out_channel=1, filter_sizes=[64, 128, 256, 512]):
        super().__init__(in_channel, out_channel, filter_sizes)

        self.encode_conv1 = ResidualConvBlock(in_channel,filter_sizes[0])
        self.encode_conv2 = ResidualConvBlock(filter_sizes[0],filter_sizes[1])
        self.encode_conv3 = ResidualConvBlock(filter_sizes[1],filter_sizes[2])
        self.encode_conv4 = ResidualConvBlock(filter_sizes[2],filter_sizes[3])
        self.bottle_conv = ResidualConvBlock(filter_sizes[3], filter_sizes[3] * 2)

        self.decode_conv4 = ResidualConvBlock(filter_sizes[3]*2, filter_sizes[3])
        self.decode_conv3 = ResidualConvBlock(filter_sizes[3], filter_sizes[2])
        self.decode_conv2 = ResidualConvBlock(filter_sizes[2], filter_sizes[1])
        self.decode_conv1 = ResidualConvBlock(filter_sizes[1], filter_sizes[0])

    def forward(self, x):
      #encode path
      encode1 = self.encode_conv1(x)
      pool1 = self.pool1(encode1)
      encode2 = self.encode_conv2(pool1)
      pool2 = self.pool2(encode2)
      encode3 = self.encode_conv3(pool2)
      pool3 = self.pool3(encode3)
      encode4 = self.encode_conv4(pool3)
      pool4 = self.pool4(encode4)
      #bottleneck
      bottleneck = self.bottle_conv(pool4)

      #decode path
      attention_output4 = self.attention4(bottleneck, encode4)
      up4 = self.up_conv4(bottleneck)
      decode4 = self.decode_conv4(torch.cat([attention_output4, up4], dim=1))

      attention_output3 = self.attention3(decode4, encode3)
      up3 = self.up_conv3(decode4)
      decode3 = self.decode_conv3(torch.cat([attention_output3, up3], dim=1))

      attention_output2 = self.attention2(decode3, encode2)
      up2 = self.up_conv2(decode3)
      decode2 = self.decode_conv2(torch.cat([attention_output2, up2], dim=1))

      attention_output1 = self.attention1(decode2, encode1)
      up1 = self.up_conv1(decode2)
      decode1 = self.decode_conv1(torch.cat([attention_output1, up1], dim=1))
       #output
      return self.final_conv(decode1)
```

```python
### DICE LOSS
class DiceLoss(nn.Module):
    """
    Dice Loss for measuring overlap between predicted and ground truth masks.
    Particularly useful for imbalanced classes in segmentation tasks.

    Args:
        smooth (float): Small constant to avoid division by zero.
    """
    def __init__(self, smooth=1e-6):
        super(DiceLoss, self).__init__()
        self.smooth = smooth

    def forward(self, pred, real):
        flattened_pred = pred.view(-1)
        flattened_real = real.view(-1)
        sum_of_sizes = flattened_pred.sum() + flattened_real.sum()
        intersection = (flattened_pred * flattened_real).sum()
        dice_coeff = (2 * intersection + self.smooth) / (sum_of_sizes + self.smooth)
        return 1 - dice_coeff


### Iou Loss
class IouLoss(nn.Module):
    """
    Intersection over Union (IoU) Loss for segmentation evaluation.
    Measures the overlap between predicted and ground truth masks.

    Args:
        smooth (float): Small constant to avoid division by zero.
    """
    def __init__(self, smooth=1e-6):
        super(IouLoss, self).__init__()
        self.smooth = smooth

    def forward(self, pred, real):
        flattened_pred = pred.view(-1)
        flattened_real = real.view(-1)
        intersection = (flattened_pred * flattened_real).sum()
        union = flattened_pred.sum() + flattened_real.sum() - intersection
        iou_coeff = (intersection + self.smooth) / (union + self.smooth)
        return 1 - iou_coeff


### Overall Loss (DO NOT CHANGE)
class OverallLoss(nn.Module):
    """
    Combined loss function that sums Dice loss, IoU loss, and Binary Cross-Entropy loss.

    Args:
        smooth (float): Small constant to avoid division by zero.
    """
    def __init__(self, smooth=1e-6):
        super(OverallLoss, self).__init__()
        self.dice_loss = DiceLoss(smooth)
        self.iou_loss = IouLoss(smooth)
        self.bce_loss = nn.BCELoss()

    def forward(self, pred, real):
        dice = self.dice_loss(pred, real)
        iou = self.iou_loss(pred, real)
        bce = self.bce_loss(pred, real)

        return dice, iou, bce, dice+iou+bce
```

خودش ضرب بعدها رو حساب میکنه و میره shape(numel,) تصویر رو به یه بردار خیلی بزرگ تبدیل میکنه

```python
def train_one_epoch(model, optimizer, criterion, data_loader, scheduler=None, device='cuda'):
    """
    Trains the model for one epoch.

    Args:
        model (nn.Module): The neural network model to train.
        optimizer (torch.optim.Optimizer): Optimizer for updating model parameters.
        criterion (nn.Module): Loss function to compute the loss.
        data_loader (DataLoader): DataLoader providing training batches.
        scheduler (optional): Learning rate scheduler.
        device (str): Device to run the training on ('cuda' or 'cpu').

    Returns:
        float: Average loss over the epoch.
    """

    total_loss = 0.0
    model.train()
    for i, (img, mask) in enumerate(tqdm(data_loader)):
        # Move the image and mask to device
        img = img.to(device)
        mask = mask.to(device)
        # Set the gradients to zero
        optimizer.zero_grad()
        # Predict the mask using the model
        pred = torch.sigmoid(model(img))
        # Compute the loss
        loss = criterion(pred, mask)
        # Compute gradients
        loss.backward()
        # update the parameters
        optimizer.step()
        if scheduler:
            # Schedule the learning rate
            scheduler.step()

        total_loss += loss.item()
    return total_loss/len(data_loader)
```

- ورودی‌ها رو batch به batch از dataset که get item داره رو صدا میزنه
- رفتار برخی لایه‌ها رو تغییر میده مثلاً dropout و BatchNorm
- scale کردن vs صفر کردن نورون‌ها
- استفاده از mean, std در batch فعلی vs اعداد ذخیره شده
- RAM ↔ VRAM انتقال داده
- به طور پیش‌فرض گرادیان‌ها رو هم acc می‌کنه. هر دفعه با optimizer.zero_grad() گرادیان‌ها رو صفر می‌کنه تا دوباره محاسبه بشه
- model(img): تابع forward در model که مال Unet هست رو صدا میزنه
- قراره overall loss با ...
- میانگین loss در یه batch
- progress bar نمایش داده: enumerate(tqdm(data_loader))
- بر اساس index همراه (mask, img)
- عدد رو از Tensor استخراج می‌کنه
- این loss از یه چیزی برحسب پارامترها و گرادیان‌ها محاسبه شده و با عملیات backward مشتق‌ها رو محاسبه می‌کنه: $\frac{\partial loss}{\partial \text{پارامتر}}$
- model.parameters() پارامترهایی هستن که موقع تعریف optimizer بهش داده شده؛ به اسم .grad یه attribute داره برای ذخیره کردن گرادیان‌ها
- model.state_dict() هم یه دیکشنری شامل اسم پارامتر و تنسور مربوط بهش
- len(data_loader) → تعداد batch‌ها در epoch
- optimizer.step() → gradient descent کار اصلی‌شون هست؛ یه سری کار پیشرفته‌تر هم دارن

```python
def evaluate_model(model, criterion, data_loader, device='cuda'):
    """
    Evaluates the model on validation or test data.

    Args:
        model (nn.Module): The neural network model to evaluate.
        criterion (nn.Module): Loss function to compute evaluation metrics.
        data_loader (DataLoader): DataLoader providing evaluation batches.
        device (str): Device to run evaluation on ('cuda' or 'cpu').

    Returns:
        tuple: Average loss, average IoU score, and average Dice score over the dataset.
    """
    total_loss = 0.0
    iou_score = 0.0
    dice_score = 0.0
    model.eval()

    with torch.no_grad():
        for i, (img, mask) in enumerate(tqdm(data_loader)):
            # Move the image and mask to device
            img = img.to(device)
            mask = mask.to(device)

            # Predict the mask
            pred = torch.sigmoid(model(img))

            # Compute the loss
            dice_loss, iou_loss, bce_loss, overall_loss = criterion(pred, mask)

            total_loss += overall_loss.item()
            iou_score += 1 - iou_loss.item()
            dice_score += 1 - dice_loss.item()

        total_loss /= len(data_loader)
        iou_score /= len(data_loader)
        dice_score /= len(data_loader)

    return total_loss, iou_score, dice_score
```

model.eval() → model.train() اینجا دیگه ن

torch.no_grad() ←
فی
هیچ گرا
دی سیانی
ساخته نـ
و گرادیانی
نا بیجـ

total_loss += overall_loss.item() → forward خروجی چیه؟/
در صورت nn.Module Tensor ه.

```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
unet = UNet().to(device)
criterion = OverallLoss()
# You can use other optimizers or Adam optimizer with customized weight decay, b1, or b2
optimizer = Adam(unet.parameters(), lr=LR)
# Add a scheduler if you like
scheduler = None
best_loss = 100.0
for i in range(NUM_EPOCHS):
    train_loss = train_one_epoch(unet, optimizer, criterion, train_loader, scheduler = scheduler, device=device)
    val_loss, iou_score, dice_score = evaluate_model(unet, criterion, val_loader, device=device)

    if val_loss < best_loss:
        best_loss = val_loss
        torch.save(unet.state_dict(), "best_unet.pth")

    print(f"Iteration {i+1}/{NUM_EPOCHS}: "
          f"Train Loss: {train_loss:.2f} | "
          f"Val Loss: {val_loss:.2f} | "
          f"Iou Score: {iou_score:.2f} | "
          f"Dice Score: {dice_score:.2f}")
```

```python
attention_unet = AttentionUNet().to(device)
criterion = OverallLoss()
# You can use other optimizers or Adam optimizer with customized weight decay, b1, or b2
optimizer = Adam(attention_unet.parameters(), lr=LR)

best_loss = 100.0
for i in range(NUM_EPOCHS):
    train_loss = train_one_epoch(attention_unet, optimizer, criterion, train_loader)
    val_loss, iou_score, dice_score = evaluate_model(attention_unet, criterion, val_loader)

    if val_loss < best_loss:
        best_loss = val_loss
        torch.save(attention_unet.state_dict(), "best_attention_unet.pth")

    print(f"Iteration {i+1}/{NUM_EPOCHS}: "
          f"Train Loss: {train_loss:.2f} | "
          f"Val Loss: {val_loss:.2f} | "
          f"Iou Score: {iou_score:.2f} | "
          f"Dice Score: {dice_score:.2f}")
```

```python
resattn_unet = ResidualAttentionUNet().to(device)
criterion = OverallLoss()
# You can use other optimizers or Adam optimizer with customized weight decay, b1, or b2
optimizer = Adam(resattn_unet.parameters(), lr=LR)

train_losses = []
val_losses = []

best_loss = 100.0
for i in range(NUM_EPOCHS):
    train_loss = train_one_epoch(resattn_unet, optimizer, criterion, train_loader)
    val_loss, iou_score, dice_score = evaluate_model(resattn_unet, criterion, val_loader)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    if val_loss < best_loss:
        torch.save(resattn_unet.state_dict(), "best_residual_attention_unet.pth")

    print(f"Iteration {i+1}/{NUM_EPOCHS}: "
          f"Train Loss: {train_loss:.2f} | "
          f"Val Loss: {val_loss:.2f} | "
          f"Iou Score: {iou_score:.2f} | "
          f"Dice Score: {dice_score:.2f}")
```

```python
# Plot the losses
def plot_losses(train_losses, val_losses):
    plt.figure(figsize=(10, 6))
    epochs = range(1, len(train_losses) + 1)
    plt.plot(epochs, train_losses, color='red', linestyle='-', label='Train Loss')
    plt.plot(epochs, val_losses, color='blue', linestyle='-', label='Validation Loss')
    plt.legend()

    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))

    plt.grid(True)
    plt.tight_layout()
    plt.show()

plot_losses(train_losses, val_losses)
```

نوع خط: پیوسته

legend: برچسب

epochs: آرایه‌ای که در محور x علامت‌گذاری می‌شه

train_losses: آرایه‌ای که در محور y علامت‌گذاری می‌شه

این دستور باعث می‌شه legend بالا در مرکز باشه

فقط اعداد صحیح نمایش داده شن

get current axes

```python
def visualize_image_mask_prediction(images, masks, predictions, n=5):
    plt.figure(figsize=(12, 4 * n))

    for i in range(n):
        plt.subplot(n, 3, i * 3 + 1)
        plt.imshow(images[i])
        plt.title(f"Image {i}")
        plt.axis("off")

        plt.subplot(n, 3, i * 3 + 2)
        plt.imshow(masks[i], cmap='gray')
        plt.title(f"Ground Truth {i}")
        plt.axis("off")

        plt.subplot(n, 3, i * 3 + 3)
        plt.imshow(predictions[i], cmap='gray')
        correct = (predictions[i] == masks[i]).astype(float)
        accuracy = correct.sum() / correct.size
        plt.title(f"Predicted {i} | accueracy: {accuracy: .2f}")
        plt.axis("off")

    plt.tight_layout()
    plt.show()
```

```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = UNet().to(device)
model.load_state_dict(torch.load("best_unet.pth"))
model.eval()

images, masks, preds = [], [], []
n = 5

for i in range(n):
    img, true_mask = val_dataset[i]
    img_tensor = img.unsqueeze(0).to(device)

    with torch.no_grad():
        pred_mask = torch.sigmoid(model(img_tensor)).cpu().squeeze().numpy()


    true_mask_np = true_mask.cpu().squeeze().numpy()
    pred_mask_bin = (pred_mask > 0.5).astype(float)

    images.append(val_images[i])
    masks.append(true_mask_np)
    preds.append(pred_mask_bin)

visualize_image_mask_prediction(images, masks, preds, n=5)
```

خوانا!

اضافه کردن یه بعد به اول چون N,C,H,W

حذف همه ابعاد ۱

تبدیل به np array سریع‌تر، مورد توصیه PyTorch|| برای استفاده در تابع بالا.