



## درس شبکه‌های عصبی و یادگیری عمیق تمرین ششم

مبینا مهرآذر - 810100216

محمد رضا محمد هاشمی - 810100206

## فهرست

1.....	پرسش 1. طراحی و پیاده‌سازی Triplet VAE برای تشخیص تومور در MRI
1.....	1-1. هدف و دیتاست
1.....	1-2. پیاده‌سازی یک VAE ساده
1.....	1-3. پیاده‌سازی Tri-VAE
1.....	1-4. ارزیابی در دیتاست BraTS (دو بعدی)
1.....	1-5. بخش امتیازی
2.....	پرسش 2. AdvGAN - 2
2.....	2-1. آشنایی با حملات خصمانه و معماهی AdvGAN
2.....	2-2. پیاده‌سازی مدل AdvGAN

## پرسش 1. طراحی و پیاده‌سازی Triplet VAE برای تشخیص تومور در MRI

### 1-1. هدف و دیتاست

پس از اضافه کردن دیتا ست ها به کلاس های دیتا ست زیر :

```
class XIT2Dataset(Dataset):

    def __init__(self, data_dir, mask_dir=None, transform=None,
target_size=(256, 256), noise_type=None, device='cuda'):

        self.data_dir = data_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.target_size = target_size
        self.noise_type = noise_type
        self.device = device

        self.image_paths = self._load_image_paths()
        if mask_dir is not None:
            self.noise_masks = self._load_noise_masks()
        else :
            self.simplex = OpenSimplex(seed=42)
            self.noise_masks = self._generate_noise_masks(100)

    def _load_image_paths(self):
        image_paths = []
        for root, _, files in os.walk(self.data_dir):
            for file in files:
                if file.endswith(".png"):
                    image_paths.append(os.path.join(root, file))
        return sorted(image_paths)
```

```
def _generate_noise_masks(self, num_masks):
    noise_masks = []
    for _ in tqdm(range(num_masks), desc="Generating noise masks"):
        noise_mask = self._generate_simplex_noise_mask()
        noise_masks.append(noise_mask.to(self.device))
    return noise_masks

def _generate_simplex_noise_mask(self):
    height, width = self.target_size
    frequency = 2 ** -6
    octaves = 6
    persistence = 0.8
    lacunarity = 2.0

    noise = torch.zeros((height, width), device=self.device)

    for octave in range(octaves):
        freq = frequency * (lacunarity ** octave)
        for i in range(height):
            for j in range(width):
                noise_value = self.simplex.noise2(i * freq, j *
freq)
                noise[i, j] += persistence ** octave * noise_value

    noise = (noise - noise.min()) / (noise.max() - noise.min())
    return noise
```

```
def _load_noise_masks(self):
    mask_paths = []
    for root, _, files in os.walk(self.mask_dir):
        for file in files:
            if file.endswith(".png"):
                mask_paths.append(os.path.join(root, file))
    mask_paths = sorted(mask_paths)

    noise_masks = []
    for mask_path in mask_paths:
        mask = Image.open(mask_path).convert("L")
        mask = self.transform(mask)
        mask = mask.to(self.device)
        noise_masks.append(mask)

    return noise_masks

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

def set_noise_type(self, noise_type):
    self.noise_type = noise_type

def _add_coarse_noise(self, image_tensor):
    batch_size, height, width = image_tensor.shape
    coarse_resolution = (16, 16)
    noise = torch.normal(mean=0.0, std=0.2, size=(batch_size,
    *coarse_resolution), device=image_tensor.device)
```

```
        noise = torch.nn.functional.interpolate(noise.unsqueeze(1),
size=(height, width), mode='bilinear', align_corners=False)

        noise = noise.squeeze(1)

        return torch.clamp(image_tensor + noise, -1, 1)

    def _add_simplex_noise(self, image_tensor):

        idx = torch.randint(0, len(self.noise_masks), (1,)).item()

        noise_mask = self.noise_masks[idx]

        image_tensor = image_tensor.to(self.device)

        image_tensor = torch.clamp(image_tensor + noise_mask, -1, 1)

        return image_tensor

    def _add_noise(self, image_tensor):

        if self.noise_type == "coarse":

            return self._add_coarse_noise(image_tensor)

        elif self.noise_type == "simplex":

            return self._add_simplex_noise(image_tensor)

        return image_tensor

    def __getitem__(self, idx):

        image_path = self.image_paths[idx]

        image = Image.open(image_path).convert("L")

        image = self.transform(image)

        if self.noise_type:

            image = self._add_noise(image)

        label = torch.tensor(0, dtype=torch.long)

        return image, label

    def plot_samples(self, num_samples=8, label_title='Healthy'):
```

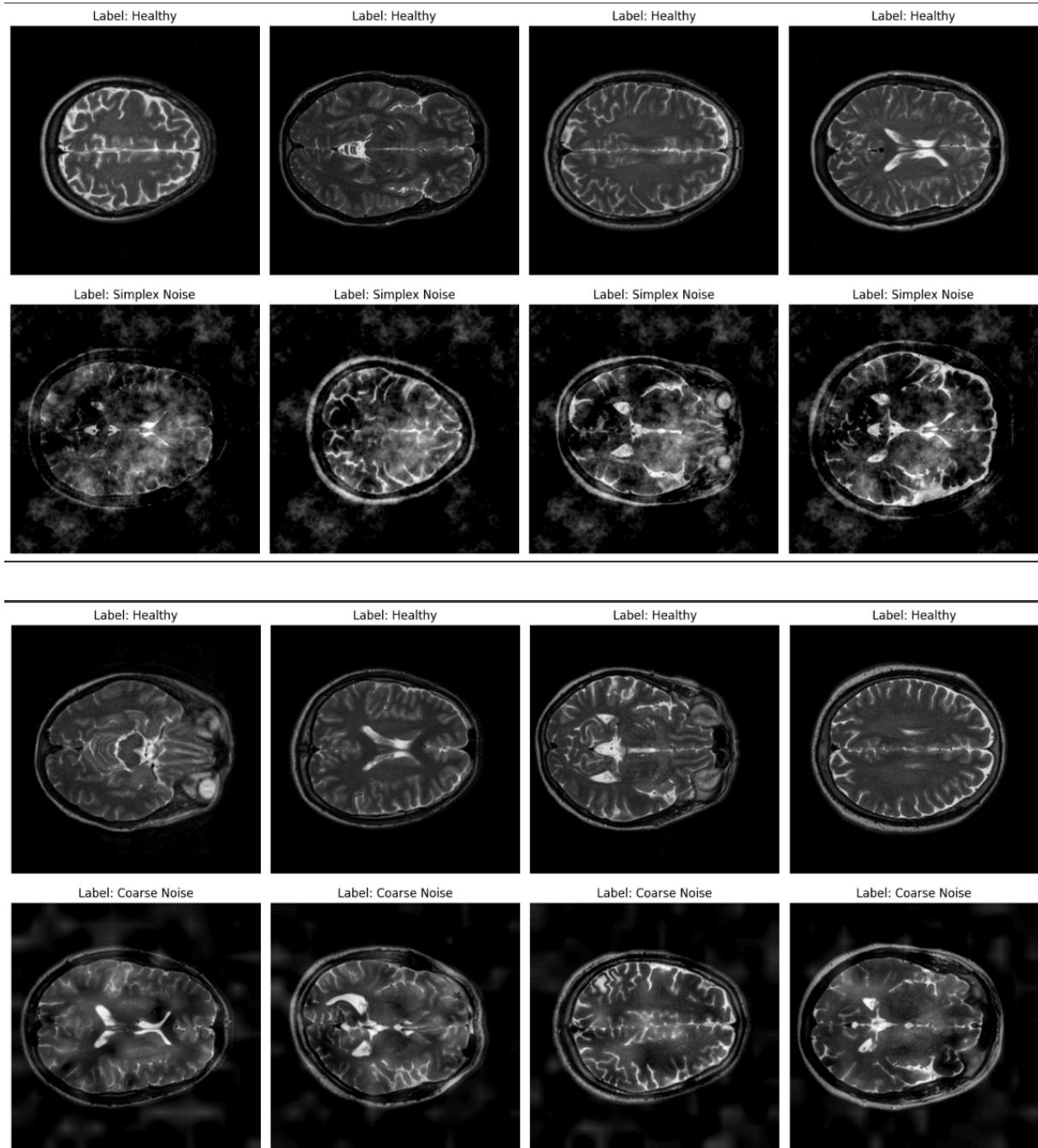
```
if num_samples % 4 != 0:
    raise ValueError("Number of samples must be a multiple of
4.")

fig, axes = plt.subplots(1, num_samples, figsize=(15, 10))

for i in range(num_samples):
    idx = np.random.randint(len(self.image_paths))
    image, label = self[idx]
    image_np = image.squeeze(0).cpu().numpy()
    axes[i].imshow(image_np, cmap='gray')
    axes[i].set_title(f"Label: {label_title}")
    axes[i].axis('off')

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

برای این دیتاست پس از ساختن اینستنس و پلات کردن مثال هایی با هر دو کلاس نویز به تصاویر زیر رسیدیم :



:brats و برای دیتاست

```
class BraTSdataset(Dataset):
    def __init__(self, data_dir, transform=None, target_size=(256, 256)):
        self.data_dir = data_dir
        self.transform = transform
```

```

        self.target_size = target_size

        self.image_paths, self.mask_paths = self._load_paths()

    def _load_paths(self):
        image_paths = []
        mask_paths = []
        for root, _, files in os.walk(self.data_dir):
            for file in files:
                if file.endswith("_t2.nii"):
                    image_paths.append(os.path.join(root, file))
                    mask_paths.append(os.path.join(root,
file.replace("_t2.nii", "_seg.nii")))
        return sorted(image_paths), sorted(mask_paths)

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

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        mask_path = self.mask_paths[idx]

        image = nib.load(image_path).get_fdata()
        mask = nib.load(mask_path).get_fdata()

        z_index = image.shape[-1] // 2
        image = image[:, :, z_index]
        mask = mask[:, :, z_index]

        image = (image - np.min(image)) / (np.max(image) -
np.min(image))

```

```
        image = resize(image, self.target_size, order=1, mode='reflect',
anti_aliasing=True)

        mask = resize(mask, self.target_size, order=0, mode='reflect',
anti_aliasing=False)

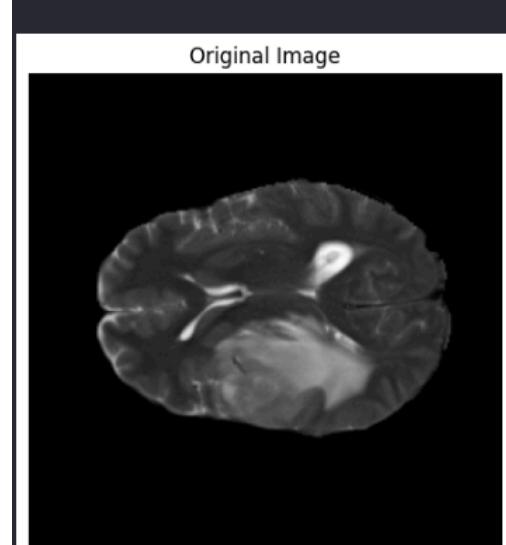
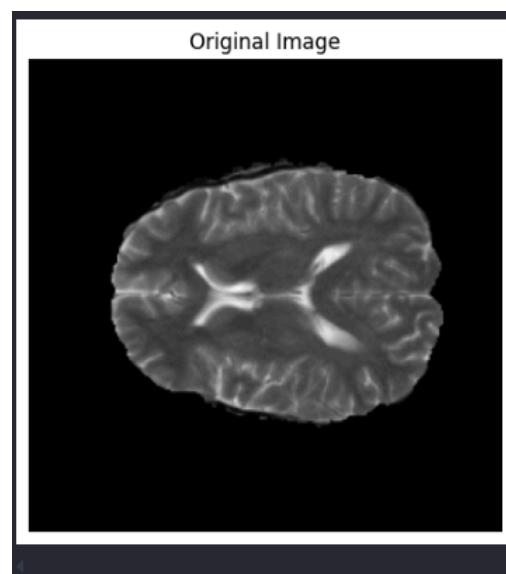
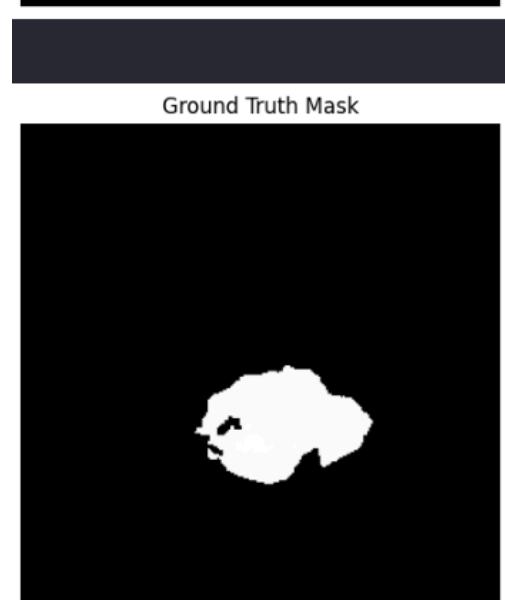
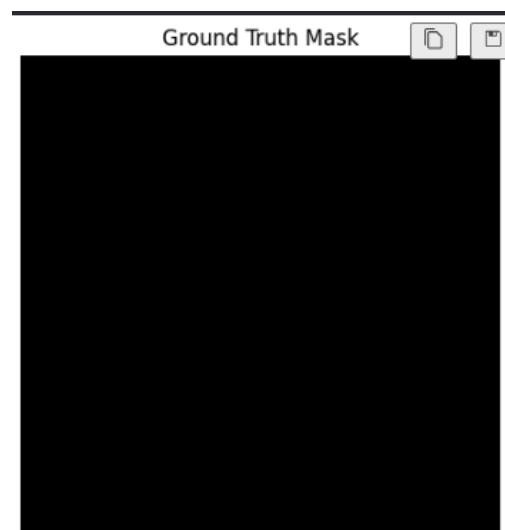
    image = Image.fromarray((image * 255).astype(np.uint8))
    mask = Image.fromarray((mask * 255).astype(np.uint8))

    if self.transform:
        image = self.transform(image)
        mask = self.transform(mask)
    else: # Convert to tensors
        image = torch.tensor(np.array(image),
dtype=torch.float32).unsqueeze(0) # Add channel dim
        mask = torch.tensor(np.array(mask), dtype=torch.float32)

    return image, mask

def plot_samples(self, num_samples=4):
    if num_samples % 2 != 0:
        raise ValueError("Number of samples must be even.")
    fig, axes = plt.subplots(2, num_samples, figsize=(15, 5))
    for i in range(num_samples):
        idx = np.random.randint(len(self.image_paths))
        image, mask = self[idx]
        image_np = image.squeeze(0).numpy() if isinstance(image,
torch.Tensor) else image
        mask_np = mask.squeeze(0).numpy() if len(mask.shape) == 3
        else mask
        axes[0, i].imshow(image_np, cmap='gray')
        axes[0, i].set_title("T2 Image")
```

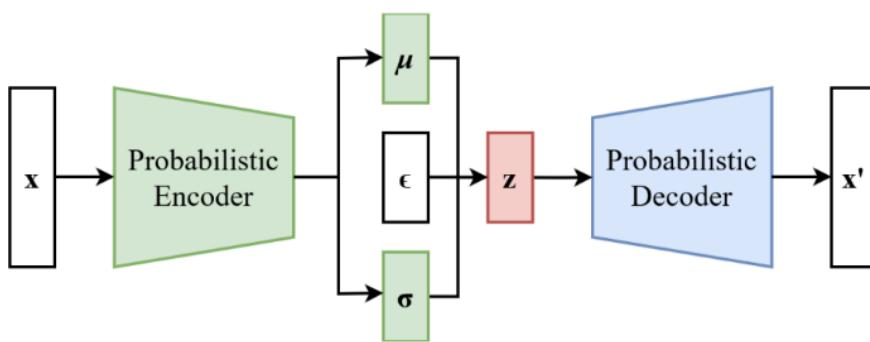
```
axes[0, i].axis('off')
axes[1, i].imshow(mask_np, cmap='gray')
axes[1, i].set_title("Segmentation Mask")
axes[1, i].axis('off')
plt.tight_layout()
plt.show()
```



## 2-1. پیاده‌سازی یک VAE ساده

### 1-2-1. معرفی مختصر VAE<sup>1</sup>

شکل زیر شمای کلی از یک VAE را نشان می‌دهد.



آنچه خودرمزگذار متغیر (VAE) را از سایر خودرمزگذارها متمایز می‌کند، نحوه خاص کدگذاری فضای نهان<sup>2</sup> (Latent Space) و استفاده‌های متفاوت از این کدگذاری احتمالاتی است. برخلاف بیشتر خودرمزگذارها که مدل‌های قطعی (Deterministic Models) هستند و یک بردار منفرد از متغیرهای نهانی گسسته (Discrete Variables) را کد می‌کنند، خودرمزگذارهای متغیر مدل‌های احتمالاتی (Probabilistic Models) هستند. آن‌ها متغیرهای نهان داده‌های آموزشی را به صورت یک مقدار ثابت گسسته کد نمی‌کنند، بلکه آن‌ها را به شکل طیفی پیوسته از احتمالات (Continuous Range of Possibilities)، که به صورت توزیع احتمالی (Probability Distribution) نمایش داده می‌شود، کد می‌کنند.

در آمار بیزی (Bayesian Statistics)، این طیف احتمالات برای متغیر نهانی، توزیع پیشین (Prior Distribution) نامیده می‌شود. در استنتاج تغییراتی (Variational Inference)، که بخشی از فرآیند تولید داده‌های جدید است، این توزیع پیشین برای محاسبه توزیع پسین (Posterior Distribution) استفاده می‌شود،  $p(z|x)$ . به عبارت دیگر، این نشان‌دهنده احتمال متغیرهای قابل مشاهده ( $x$ ) با توجه به یک مقدار مشخص برای متغیر نهانی ( $z$ ) است.

<sup>1</sup> خودرمزگذارهای متغیر (VAEs) ابزارهای قدرتمندی برای مدل‌سازی داده‌ها هستند که از ساختار خاصی برای پردازش و تولید داده‌های جدید استفاده می‌کنند. این روش‌ها با استفاده از یک رویکرد احتمالی به جای روش‌های تعیین‌شده، به گونه‌ای طراحی شده‌اند که فضای نهانی را به شکلی سازمان‌دهی شده و معنی‌دار شکل دهند.

<sup>2</sup> در خودرمزگذارهای متغیر، فضای نهانی به صورت یک توزیع پیوسته مدل‌سازی می‌شود که به مدل اجازه می‌دهد تا نمونه‌های جدیدی تولید کند که مشابه داده‌های آموزشی هستند، اما هرگز دقیقاً مشابه آن‌ها نیستند. این امر به مدل‌سازان این امکان را می‌دهد که به راحتی بین نمونه‌های مختلف حرکت کنند و نتایج نوآورانه‌ای تولید کنند.

برای هر ویژگی نهان داده‌های آموزشی، خودرمزگذار متغیر(VAE) دو بردار متفاوت را کد می‌کند: یک بردار از میانگین‌ها ("μ") و یک بردار از انحراف معیارها ("σ"). در اصل، این دو بردار نشان‌دهنده طیف احتمالات برای هر متغیر نهانی و میزان تغییرات مورد انتظار در هر طیف احتمالات هستند.

با نمونه‌برداری تصادفی از این طیف احتمالات کدشده، خودرمزگذار متغیر می‌تواند نمونه‌های جدیدی ایجاد کند که در عین منحصر به فرد بودن، شباهت قابل توجهی به داده‌های آموزشی اصلی دارند. اگرچه این روش در اصول به‌ظاهر ساده به نظر می‌رسد، اما برای به‌کارگیری آن در عمل، نیاز به تغییرات بیشتری در معماری استاندارد خودرمزگذار دارد.

مانند تمام خودرمزگذارها، خودرمزگذار متغیر از خطای بازسازی یا خطای بازتولید (Reconstruction Error) به عنوان تابع خطای اصلی در آموزش استفاده می‌کند. خطای بازسازی بین داده ورودی اصلی و نسخه بازسازی شده آن توسط بخش رمزگشا (Decoder) یا به عبارتی تفاوت ورودی / خروجی را اندازه‌گیری می‌کند. الگوریتم‌های مختلفی مانند Cross-Entropy یا میانگین مربع خطای Mean-Squared Error (MSE) می‌توانند به عنوان تابع خطای بازسازی استفاده گردد.

همان‌طور که پیش‌تر توضیح داده شد، معماری خودرمزگذار یک گلوبگاه (Bottleneck) ایجاد می‌کند که تنها بخشی از داده‌های ورودی اصلی را اجازه عبور به رمزگشا می‌دهد. در ابتدای آموزش، که معمولاً با یک مقداردهی اولیه تصادفی (Random Initialization) برای پارامترهای مدل آغاز می‌شود، بخش رمزگذار (Encoder) هنوز نیاموخته است که کدام بخش از داده‌ها را باید بیشتر در نظر بگیرد. بنابراین، در ابتدا یک نمایش نهان نامطلوب (Suboptimal Latent Representation) ارائه می‌دهد و بخش رمزگشا بازسازی ناقص یا نا دقیقی از ورودی اصلی را ارائه خواهد داد.

با کاهش خطای بازسازی از طریق Gradient Descent بر پارامترهای شبکه رمزگذار و رمزگشا، وزن‌های مدل خودرمزگذار به‌گونه‌ای تنظیم می‌شوند که یک کدگذاری مفیدتر از فضای نهانی ایجاد کنند و در نتیجه، بازسازی دقیق‌تری انجام دهنند. به‌طور ریاضی، هدف تابع خطای بازسازی، بهینه‌سازی  $\mathcal{L}(x|z)$  است، که در آن  $\theta$  نمایانگر پارامترهای مدلی است که بازسازی دقیق ورودی  $x$  را با توجه به متغیر نهانی  $z$  تضمین می‌کند. خطای بازسازی به‌نهایی برای بهینه‌سازی اکثر خودرمزگذارها کافی است، زیرا هدف اصلی آن‌ها یادگیری یک نمایش فشرده از داده‌های ورودی است که به بازسازی دقیق منجر شود.

اما هدف یک خودرمزگذار متغیر، بازسازی ورودی اصلی نیست؛ بلکه ایجاد نمونه‌های جدیدی است که به ورودی اصلی شباهت دارند. بنابراین، یک عبارت بهینه‌سازی اضافی مورد نیاز است.

برای اهداف استنتاج تغییراتی (Variational Inference)—یعنی تولید نمونه‌های جدید توسط یک مدل آموزش‌دیده—خطای بازسازی به‌نهایی می‌تواند به یک کدگذاری نامنظم از فضای نهانی منجر شود که داده‌های آموزشی را بیش از حد می‌آموزد (Overfitting) و در تولید نمونه‌های جدید ضعیف عمل می‌کند. از این‌رو،

خودمرمزگذار متغیر یک عبارت **رگولاریزیشن** (regularization) دیگر را نیز اضافه می‌کند: واگرایی کولبک-لیبلر با به اختصار **KL divergence** یا همان **Kullback-Leibler Divergence**

برای تولید تصاویر، رمزگشایان از فضای نهانی نمونه‌برداری می‌کنند. نمونه‌برداری از نقاط خاصی در فضای نهانی که ورودی‌های اصلی داده‌های آموزشی را نشان می‌دهند، همان ورودی‌های اصلی را تکرار می‌کند. برای تولید تصاویر جدید، خودمرمزگذار متغیر باید بتواند از هر نقطه در فضای نهانی بین داده‌های اصلی نمونه‌برداری کند. برای این کار، فضای نهانی باید دو نوع نظم را داشته باشد:

1. پیوستگی (Continuity): نقاط نزدیک در فضای نهانی باید هنگام رمزگشایی محتواهای مشابهی ایجاد کنند.
2. کامل بودن (Completeness): هر نقطه‌ای که از فضای نهانی نمونه‌برداری شود، باید هنگام رمزگشایی محتواهای معناداری تولید کند.

یک روش ساده برای اعمال هر دو ویژگی پیوستگی و کامل بودن در فضای نهان (latent space) این است که مطمئن شویم فضای نهانی از یک توزیع نرمال استاندارد (Standard Normal Distribution) پیروی می‌کند. اما بهینه‌سازی تنها خطای بازسازی، مدل را به سازماندهی خاصی از فضای نهانی وادار نمی‌کند، زیرا فضای "میانی" برای بازسازی دقیق داده‌های اصلی چندان مهم نیست. در اینجا است که KL divergence وارد عمل می‌شود.

واگرایی کولبک-لیبلر (KL Divergence) معیاری برای مقایسه دو توزیع احتمالی است. به حداقل رساندن KL divergence بین توزیع یادگرفته شده برای متغیرهای نهان و یک توزیع گاوی ساده (Gaussian) با مقادیر استاندارد، باعث می‌شود کدگذاری یادگرفته شده از متغیرهای نهانی به شکل توزیع نرمال درآید. این کار اجازه می‌دهد که هر نقطه‌ای در فضای نهانی به شکلی پیوسته و روان به محتواهای جدید تبدیل شود و به همین دلیل تولید نمونه‌های جدید ممکن گردد.

با ترکیب دوتابع خطای بازسازی و واگرایی KL-VAEs می‌توانند داده‌ها را به‌گونه‌ای بهینه کنند که هم کیفیت بازسازی و هم قابلیت تولید نمونه‌های جدید را بهبود ببخشند. این رویکرد به خودمرمزگذارها این امکان را می‌دهد که ویژگی‌های خاصی از داده‌ها را یاد بگیرند و در عین حال، فضای نهانی را در حداقل حالت ممکن نگه‌دارند.

خودمرمزگذارهای متغیر در بسیاری از حوزه‌ها مانند تولید تصویر، پردازش زبان طبیعی و تحلیل داده‌های بزرگ مورد استفاده قرار می‌گیرند. آن‌ها می‌توانند به عنوان ابزاری برای کشف ساختار در داده‌های پیچیده عمل کنند و برای تولید محتواهای جدید و جالب به کار گرفته شوند. علیرغم مزایای فراوان VAEs، چالش‌هایی نیز وجود دارد، از جمله نیاز به ترفندهای خاص برای بهینه‌سازی و مدیریت داده‌های پیچیده. همچنین، ساختار مدل باید به‌گونه‌ای طراحی شود که از تناقض‌های احتمالی جلوگیری کند و به بهترین نحو عملکرد بهینه را به دست آورد.

## 2. پیاده سازی مدل

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.el1conv1 = ConvBlock(1, 16)
        self.el1conv2 = ConvBlock(16, 16)
        self.el1avgpool = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)

        self.el2conv1 = ConvBlock(16, 32)
        self.el2conv2 = ConvBlock(32, 32)
        self.el2avgpool = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)

        self.el3conv1 = ConvBlock(32, 64)
        self.el3conv2 = ConvBlock(64, 64)
        self.el3avgpool = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)

        self.el4conv1 = ConvBlock(64, 128)
        self.el4conv2 = ConvBlock(128, 128)
        self.el4avgpool = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)

        self.el5conv1 = ConvBlock(128, 256)
        self.el5conv2 = ConvBlock(256, 256)
        self.el5avgpool = nn.AvgPool2d(kernel_size=2, stride=2, padding=0)

        self.el6conv1 = ConvBlock(256, 512)
        self.el6conv2 = ConvBlock(512, 512)

        self.fc_mu = nn.Linear(512 * 8 * 8, 128)
        self.fc_logvar = nn.Linear(512 * 8 * 8, 128)
        self.fc_decode = nn.Linear(128, 512 * 8 * 8)

        self.dl6upconv = nn.ConvTranspose2d(1024, 256, kernel_size=2, stride=2)

        self.dl5conv1 = ConvBlock(256, 256)
        self.dl5conv2 = ConvBlock(256, 256)
        self.dl5upconv = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
```

```
        self.dl4conv1 = ConvBlock(128, 128)
        self.dl4conv2 = ConvBlock(128, 128)
        self.dl4conv3 = ConvBlock(128, 128)
        self.dl4conv4 = ConvBlock(128, 128)
        self.dl4conv5 = nn.Conv2d(128, 1, 1, 1, 0)
        self.dl4upconv = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)

        self.dl3conv1 = ConvBlock(64, 64)
        self.dl3conv2 = ConvBlock(64, 64)
        self.dl3upconv = nn.ConvTranspose2d(64, 32, kernel_size=2, stride=2)

        self.dl2conv1 = ConvBlock(32, 32)
        self.dl2conv2 = ConvBlock(32, 32)
        self.dl2upconv = nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2)

        self.dl1conv1 = ConvBlock(16, 16)
        self.dl1conv2 = ConvBlock(16, 16)
        self.dl1conv3 = nn.Conv2d(16, 1, 1, 1, 0)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std
```

```
def forward(self, x):

    x = self.el1conv1(x)
    x = self.el1conv2(x)
    skip1 = x
    x = self.el1avgpool(x)

    x = self.el2conv1(x)
    x = self.el2conv2(x)
    skip2 = x
    x = self.el2avgpool(x)

    x = self.el3conv1(x)
    x = self.el3conv2(x)
    skip3 = x
    x = self.el3avgpool(x)

    x = self.el4conv1(x)
    x = self.el4conv2(x)
    x = self.el4avgpool(x)

    x = self.el5conv1(x)
    x = self.el5conv2(x)
    x = self.el5avgpool(x)

    x = self.el6conv1(x)
    x = self.el6conv2(x)
    skip4 = x

    x = x.view(x.size(0), -1)
    mu = self.fc_mu(x)
    logvar = self.fc_logvar(x)
```

```

z = self.reparameterize(mu, logvar)
x = self.fc_decode(z)
x = x.view(x.size(0), 512, 8, 8)

x = self.dl6upconv(torch.cat([x, skip4], dim=1))

x = self.dl5conv1(x)
x = self.dl5conv2(x)
x = self.dl5upconv(x)

x = self.dl4conv1(x)
x = self.dl4conv2(x)
ux = self.dl4upconv(x)
x = self.dl4conv3(x)
x = self.dl4conv4(x)
l4out = self.dl4conv5(x)

x = self.dl3conv1(ux)
x = self.dl3conv2(x)
x = self.dl3upconv(x)

x = self.dl2conv1(x)
x = self.dl2conv2(x)
x = self.dl2upconv(x)

x = self.dl1conv1(x)
x = self.dl1conv2(x)
x = self.dl1conv3(x)

return x, l4out, mu, logvar

```

این VAE پیاده سازی ساده شده ای از مدل مورد استفاده در مقاله است که با الهام از UNet طراحی شده بوده است در این طراحی همانطور که قابل مشاهده است عکس ورودی ابتدا از دو کانولوشن د شده که هرکدام مطابق شکل زیر است :

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super(ConvBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
        self.norm = nn.GroupNorm(8, out_channels)
        self.activation = nn.SiLU()

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

که در این واحد های کانولوشنی ابتدا یک کانولوشن سپس گروپ نرم انجام میشود. و از اکتیویشن فانکشن رد میشود.

پس از دو کانولوشن اول یک avg پول انجام میشود و همین فرایند تا 5 لایه انجام میشود که ابعاد عکس به ترتیب  $256 \times 256 - 128 \times 128 - 64 \times 64 - 32 \times 32 - 16 \times 16 - 8 \times 8$  خواهد شد و در لایه 6 ام دو واحد کانولوشنی دیگر داریم و پس از آن به فضای لیتنت وارد می شود که 128 متغیر دارد و در مرحله بعد ابتدا ویژگی ها فشرده شده از دو fc\_mu و fc\_logvar عبور کرده تا میانگین و انحراف معیار فضای لیتنت محاسبه شده و به کمک ری پارامتر ایزیشن و لایه ی دیکود نمونه گیری از توزیع جهت فراهم کردن شرایط برای محاسبه گرادیان انجام شده و به لایه دیکود وارد میشویم که دوال لایه انکد ما است تنها در لایه 4 ام پس از اپ کانو یک خروجی  $32 \times 32$  نیز داریم.

همچنین انتخاب ابعاد فضای لیتنت به 128 به دلیل برقرار تعادل میان کیفیت بازسازی تصاویر و جنرالیزیشن است.

### 3. آموزش روی دیتاست سالم

```
def train_vae(model, dataloader, device, epochs=10, learning_rate=1e-4, alpha=1, beta=1):

    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    loss_history = []

    def kld_loss(mu, logvar):
        return -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) / mu.size(0)

    def ssim_loss(pred, target):
        return 1 - ssim(pred, target, data_range=1.0)

    model.to(device)
    model.train()

    for epoch in range(epochs):
        epoch_loss = 0
        progress_bar = tqdm(dataloader, desc=f"Epoch {epoch + 1}/{epochs}", unit="batch")

        for images, _ in progress_bar:
            images = images.to(device)

            reconstructed, l4_out, mu, logvar = model(images)

            l4_target = nn.functional.interpolate(images, size=(32, 32), mode='bilinear', align_corners=False)

            l1_loss_main = nn.L1Loss()(reconstructed, images)
            l1_loss_l4 = nn.L1Loss()(l4_out, l4_target)
            kld = kld_loss(mu, logvar)
            ssim_value = ssim_loss(reconstructed, images)

            total_loss = l1_loss_main + l1_loss_l4 + kld + ssim_value

            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

            epoch_loss += total_loss.item()

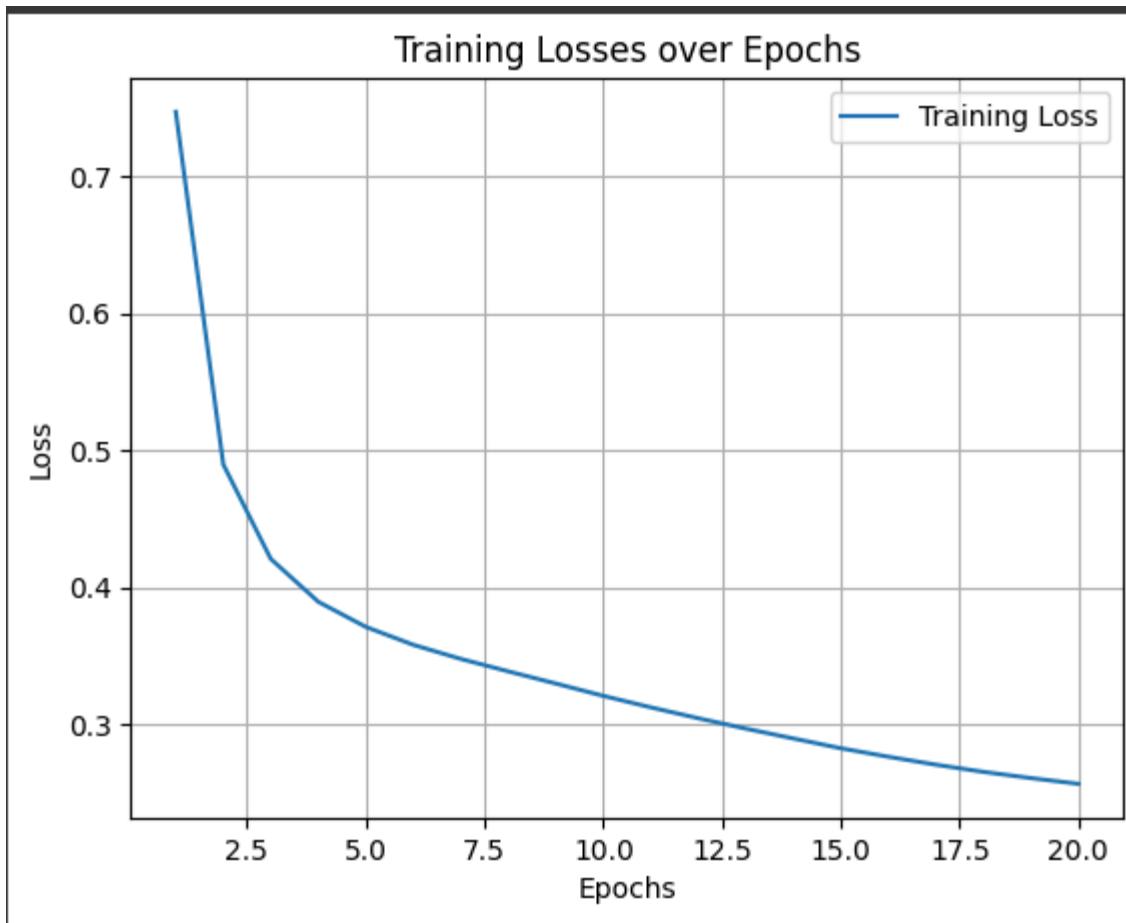
        progress_bar.set_postfix({"Loss": total_loss.item()})

        avg_epoch_loss = epoch_loss / len(dataloader)
        loss_history.append(avg_epoch_loss)

        print(f"Epoch [{epoch + 1}/{epochs}], Avg Loss: {avg_epoch_loss:.4f}")

    return loss_history
```

همچنین برای آموزش مدل از انجا که شبکه ما VAE است یکی از اصلی ترین لاس هایی که باید مینیمایز شود KLD است در کنار این لاس مشابه مقاله ریکانستراکشن لاس تصویر و ssim لاس را نیز استفاده کردیم تا قابلیت قیاس بیشتری برای مدل عادی و قسمت بعد داشته باشیم و ترکیب لاس نسبتا مشابهی داشته باشند . همچنین از بهینه ساز ADAM جهت آموزش مدل استفاده شده است. نمودار لاس در فرایند ترین در شکل زیر قابل مشاهده است :



#### 4-2-1. تست مختصر روی BraTS

به کمک کد زیر نمونه هایی از پیش بینی مدل بر روی داده ها انجام میدهیم :

```
def plot_random_results_vae(vae, dataset, threshold=0.1, num_samples=4):
    vae.eval()
    dice_scores = []

    random_indices = random.sample(range(len(dataset)), num_samples)

    for idx in random_indices:
        image, ground_truth_mask = dataset[idx]
        image = image.to(device).unsqueeze(0)

        with torch.no_grad():
            reconstruction, _, _, _ = vae(image)

        residual = torch.abs(image - reconstruction)

        if torch.sum(residual) > threshold:
            dice_scores.append(0)
        else:
            dice_scores.append(1)
```

```
predicted_mask = (residual > threshold).float()

plt.figure(figsize=(20, 5))

plt.subplot(1, 5, 1)
plt.imshow(image.squeeze().cpu().numpy(), cmap='gray')
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 5, 2)
plt.imshow(reconstruction.squeeze().cpu().numpy(), cmap='gray')
plt.title("Reconstructed Image")
plt.axis('off')

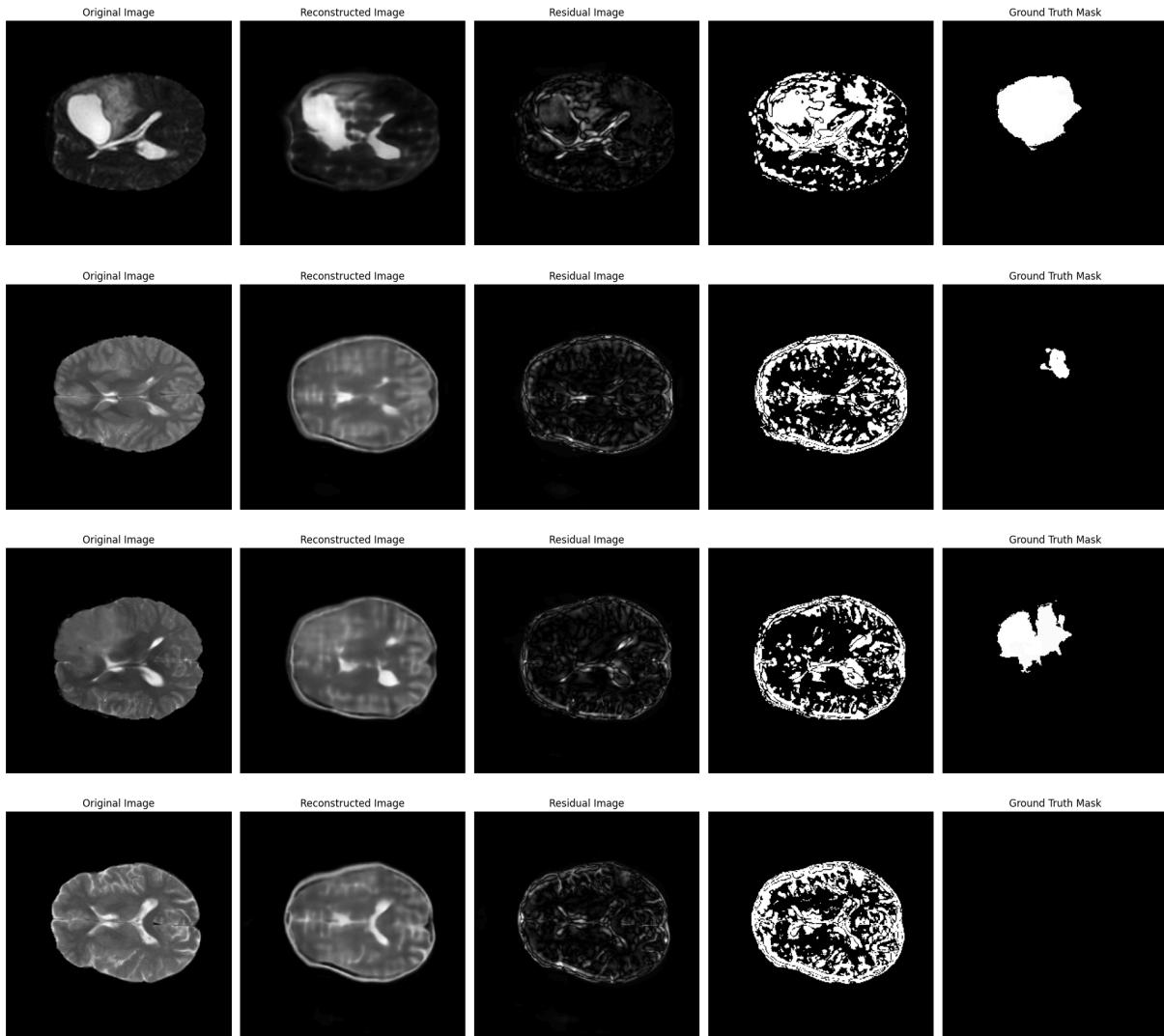
plt.subplot(1, 5, 3)
plt.imshow(residual.squeeze().cpu().numpy(), cmap='gray')
plt.title("Residual Image")
plt.axis('off')

plt.subplot(1, 5, 4)
plt.imshow(predicted_mask.squeeze().cpu().numpy(), cmap='gray')
plt.axis('off')

plt.subplot(1, 5, 5)
plt.imshow(ground_truth_mask.squeeze().cpu().numpy(),
cmap='gray')
plt.title("Ground Truth Mask")
plt.axis('off')

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

که برای مدل ساده نتیجه زیر را میدهد :



Dice score معیاری برای ارزیابی میزان هم پوشانی دو مجموعه است و اغلب برای مقایسه ماسکهای پیشینی شده واقعی در بخشندی تصاویر پزشکی استفاده می‌شود. در تصاویر پزشکی، Dice Score به segmentation مد لهای دلیل حساسیت بالای آن به نواحی هم پوشانی، برای ارزیابی دقیق مانند تشخیص، تومور استفاده می‌شود.

و از تابع زیر جهت محاسبه دایس استفاده میکنیم :

```

        (image = image.to(device).unsqueeze(0

                                :()with torch.no_grad
        (reconstruction, _, _, _ = vae(image

        (residual = torch.abs(image - reconstruction

        ()predicted_mask = (residual > threshold).float

                                )dice = dice_coefficient
        , ()predicted_mask.squeeze().cpu().numpy
        ()ground_truth_mask.numpy
        (
        (dice_scores.append(dice

        (avg_dice_score = sum(dice_scores) / len(dice_scores
        ("{print(f"Average Dice Score: {avg_dice_score:.4f

```

در موقع کال تابع ترشولد به مقدار 0.1 سنت شد.

```

[1] calculate_dice(vae, brats_dataset, 0.1)
→ Average Dice Score: 0.2859

```

### 3-1. پیادهسازی Tri-VAE

در این بخش ابتدا مطابق مقاله مدل را به شکل زیر بروز کرده و اسکیپ کانکشن ها را اضافه کردیم:

```

class Improved_VAE(nn.Module):

    def __init__(self):
        super(Improved_VAE, self).__init__()

        self.ellconv1 = ConvBlock(1, 16)
        self.ellconv2 = ConvBlock(16, 16)
        self.ellavgpool = nn.AvgPool2d(kernel_size=2, stride=2,
padding=0)

```

```
        self.el2conv1 = ConvBlock(16, 32)

        self.el2conv2 = ConvBlock(32, 32)

        self.el2avgpool = nn.AvgPool2d(kernel_size=2, stride=2,
padding=0)

        self.el3conv1 = ConvBlock(32, 64)

        self.el3conv2 = ConvBlock(64, 64)

        self.el3avgpool = nn.AvgPool2d(kernel_size=2, stride=2,
padding=0)

        self.el4conv1 = ConvBlock(64, 128)

        self.el4conv2 = ConvBlock(128, 128)

        self.el4avgpool = nn.AvgPool2d(kernel_size=2, stride=2,
padding=0)

        self.el5conv1 = ConvBlock(128, 256)

        self.el5conv2 = ConvBlock(256, 256)

        self.el5avgpool = nn.AvgPool2d(kernel_size=2, stride=2,
padding=0)

        self.el6conv1 = ConvBlock(256, 512)

        self.el6conv2 = ConvBlock(512, 512)

        self.fc_mu = nn.Linear(512 * 8 * 8, 256)

        self.fc_logvar = nn.Linear(512 * 8 * 8, 256)

        self.fc_decode = nn.Linear(256, 512 * 8 * 8)

        self.d16upconv = nn.ConvTranspose2d(1024, 256,
kernel_size=2, stride=2)
```

```
        self.dl5conv1 = ConvBlock(256, 256)
        self.dl5conv2 = ConvBlock(256, 256)
        self.dl5upconv = nn.ConvTranspose2d(256, 128,
kernel_size=2, stride=2)

        self.dl4conv1 = ConvBlock(128, 128)
        self.dl4conv2 = ConvBlock(128, 128)
        self.dl4conv3 = ConvBlock(128, 128)
        self.dl4conv4 = ConvBlock(128, 128)
        self.dl4conv5 = nn.Conv2d(128, 1, 1, 1, 0)
        self.dl4upconv = nn.ConvTranspose2d(128, 64,
kernel_size=2, stride=2)

        self.dl3conv1 = ConvBlock(64, 64)
        self.dl3conv2 = ConvBlock(64, 64)
        self.dl3upconv = nn.ConvTranspose2d(64, 32,
kernel_size=2, stride=2)

        self.dl2conv1 = ConvBlock(32, 32)
        self.dl2conv2 = ConvBlock(32, 32)
        self.dl2upconv = nn.ConvTranspose2d(32, 16,
kernel_size=2, stride=2)

        self.dllconv1 = ConvBlock(16, 16)
        self.dllconv2 = ConvBlock(16, 16)
        self.dllconv3 = nn.Conv2d(16, 1, 1, 1, 0)

        self.l1gcs = FeatureInteractionModule(16, 16)
        self.l2gcs = FeatureInteractionModule(32, 32)
        self.l3gcs = FeatureInteractionModule(64, 64)
```

```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):

    x = self.ellconv1(x)
    x = self.ellconv2(x)
    skip1 = x
    x = self.ellavgpool(x)

    x = self.el2conv1(x)
    x = self.el2conv2(x)
    skip2 = x
    x = self.el2avgpool(x)

    x = self.el3conv1(x)
    x = self.el3conv2(x)
    skip3 = x
    x = self.el3avgpool(x)

    x = self.el4conv1(x)
    x = self.el4conv2(x)
    x = self.el4avgpool(x)

    x = self.el5conv1(x)
    x = self.el5conv2(x)
```

```
x = self.el5avgpool(x)

x = self.el6conv1(x)
x = self.el6conv2(x)
skip4 = x

x = x.view(x.size(0), -1)
mu = self.fc_mu(x)
logvar = self.fc_logvar(x)

z = self.reparameterize(mu, logvar)
x = self.fc_decode(z)
x = x.view(x.size(0), 512, 8, 8)

x = self.dl6upconv(torch.cat([x, skip4], dim=1))

x = self.dl5conv1(x)
x = self.dl5conv2(x)
x = self.dl5upconv(x)

x = self.dl4conv1(x)
x = self.dl4conv2(x)
ux = self.dl4upconv(x)
x = self.dl4conv3(x)
x = self.dl4conv4(x)
l4out = self.dl4conv5(x)

l3gcs = self.l3gcs(skip3, ux)
```

```

x = self.dl3conv1(torch.cat([l3gcs, ux], dim=1))

x = self.dl3conv2(x)

x = self.dl3upconv(x)

l2gcs = self.l2gcs(skip2, x)

x = self.dl2conv1(torch.cat([l2gcs, x], dim=1))

x = self.dl2conv2(x)

x = self.dl2upconv(x)

l1gcs = self.l1gcs(skip1, x)

x = self.dllconv1(torch.cat([l1gcs, x], dim=1))

x = self.dllconv2(x)

x = self.dllconv3(x)

return x, l4out, mu, logvar

```

که همانطور که قابل مشاهده است تنها تفاوت با vae اصلی اضافه شدن اسکیپ کانکشن در سه لایه اولیه پس از گذشت از GCS است.

پیاده سازی گیت های GCS به صورت زیر است :

```

class FeatureInteractionModule(nn.Module):

    def __init__(self, encoder_channels,
decoder_channels, reduction_ratio=4):

        super(FeatureInteractionModule,
self).__init__()

```

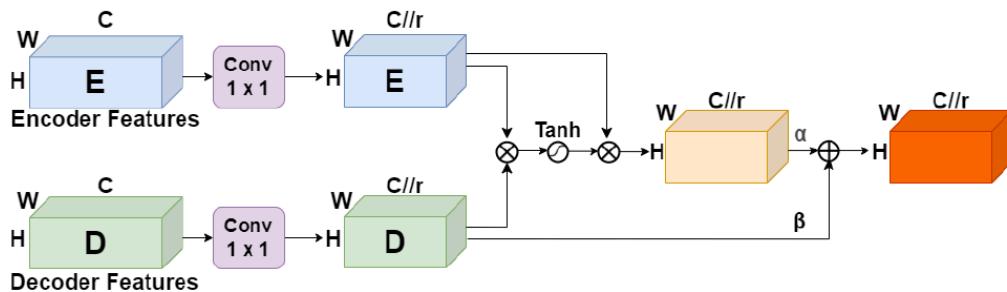
```
    reduced_channels = encoder_channels //  
reduction_ratio  
  
    self.encoder_conv =  
nn.Conv2d(encoder_channels, reduced_channels,  
kernel_size=1)  
    self.decoder_conv =  
nn.Conv2d(decoder_channels, reduced_channels,  
kernel_size=1)  
  
    self.tanh = nn.Tanh()  
    self.alpha = nn.Parameter(torch.ones(1))  
    self.beta = nn.Parameter(torch.ones(1))  
  
def forward(self, encoder_features,  
decoder_features):  
    reduced_encoder =  
self.encoder_conv(encoder_features)  
    reduced_decoder =  
self.decoder_conv(decoder_features)  
  
    tanh_output = self.tanh(reduced_encoder *  
reduced_decoder)  
    x = tanh_output * reduced_encoder
```

```

        output = self.alpha * x + self.beta *
reduced_encoder

        return output
    
```

در این گیت جهت جلو گیری از فراموشی اطلاعات و حفظ جزئیات saptial به صورت انتخابی فیچر مپ های انکودر و دیکودر اطلاعات اسکیپ کانکشن را تشکیل میدهد که ابتدا با ضربی ذکر شده ابعاد هر دو فیچر مپ انکودر و دیکودر را به مقدار ضربی داده شده به لایه کانولوشنی کاهش داده و سپس به کمک ضرب موارد حاصل و عبور از تانزانت هایپربولیک و در نهایت جمع دوباره با خروجی دیکودر تشکیل میدهد.



و به کمک تابع زیر آموزش میدهیم :

```

def train_tri_vae
    ,model
    ,anc_dataloader
    ,pos_dataloader
    ,neg_dataloader
    ,dataset
    ,device
    ,epochs=20
    ,learning_rate=3e-4
    
```

```

        , alpha=1
        , beta=1
        patience=3
    : (
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate
[] = loss_history

        : (def kld_loss(mu, logvar
return -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) /
(mu.size(0

        : (def ssim_loss(pred, target
        (return 1 - ssim(pred, target

        : (def triplet_loss(anchor_mu, positive_mu, negative_mu, margin=1.0
            pos_dist = torch.sum((anchor_mu - positive_mu) ** 2,
                () dim=1).mean
            neg_dist = torch.sum((anchor_mu - negative_mu) ** 2,
                () dim=1).mean
            (return torch.relu(pos_dist - neg_dist + margin

        (model.to(device
        () model.train

        ('best_epoch_loss = float('inf
        patience_counter = 0

        : (for epoch in range(epochs
            epoch_loss = 0
            ) progress_bar = tqdm

```

```

, (zip(anc_dataloader, pos_dataloader, neg_dataloader
      , " {desc=f"Epoch {epoch + 1}/{epochs
           , "unit="batch
           (total=len(anc_dataloader
           (

for (anc_images, _), (pos_images, _), (neg_images, _) in
      :progress_bar
anc_images, pos_images, neg_images = anc_images.to(device),
      (pos_images.to(device), neg_images.to(device

noised_negs = torch.stack([dataset._add_coarse_noise(image)
      ([for image in neg_images

anc_reconstructed, anc_l4_out, anc_mu, anc_logvar =
      (model(anc_images
pos_reconstructed, pos_l4_out, pos_mu, pos_logvar =
      (model(pos_images
neg_reconstructed, neg_l4_out, neg_mu, neg_logvar =
      (model(noised_negs

anc_l4_target = nn.functional.interpolate(anc_images,
      (size=(32, 32), mode='bilinear', align_corners=False
pos_l4_target = nn.functional.interpolate(pos_images,
      (size=(32, 32), mode='bilinear', align_corners=False
neg_l4_target = nn.functional.interpolate(neg_images,
      (size=(32, 32), mode='bilinear', align_corners=False

(l1_anc32 = nn.L1Loss()(anc_l4_out, anc_l4_target
(l1_pos32 = nn.L1Loss()(pos_l4_out, pos_l4_target
(l1_neg32 = nn.L1Loss()(neg_l4_out, neg_l4_target

```

```

(l1_neg256 = nn.L1Loss()(neg_reconstructed, neg_images

(kld_anc = kld_loss(anc_mu, anc_logvar
(kld_pos = kld_loss(pos_mu, pos_logvar

(ssim_neg = ssim_loss(neg_reconstructed, neg_images

(l_triplet = triplet_loss(anc_mu, pos_mu, neg_mu

l1_total = l1_anc32 + l1_pos32 + l1_neg32 + l1_neg256
kld_total = kld_anc + kld_pos
total_loss = l1_total + kld_total + l_triplet + ssim_neg

()optimizer.zero_grad
()total_loss.backward
()optimizer.step

()epoch_loss += total_loss.item
({()progress_bar.set_postfix({"Loss": total_loss.item

(avg_epoch_loss = epoch_loss / len(anc_dataloader
(loss_history.append(avg_epoch_loss
print(f"Epoch [{epoch + 1}/{epochs}], Avg Loss:
("{{avg_epoch_loss:.4f

:if avg_epoch_loss < best_epoch_loss
best_epoch_loss = avg_epoch_loss
patience_counter = 0
('torch.save(model.state_dict(), 'best_I_model.pth

```

```

        :else
            patience_counter += 1
            :if patience_counter >= patience
                (.print("Early stopping triggered. Stopping training
                ('model.load_state_dict(torch.load('best_I_model.pth
                    break

    return loss_history

```

که در آن از ریکانستراکشن لاس SSIM ، L1 ، KL divergence ، triplet loss ذکر شده در مقاله و لاس میان نمونه پیش از نویز و نمونه بازسازی شده مدل استفاده شده.

نویز مورد استفاده در این قسمت نویز coarse است که در آن خانه های ماتریس کوچکی مانند  $16 \times 16$  به کمک مقادیر تصادفی با توزیع نرمال پر میشود و سپس به سایز عکس اصلی رسایز میشود.

در فرایند آموزش این مدل از سه تصویر anchor, postive و negative که تصویری با اعمال نویز بر روی یک مغز سالم است استفاده شده است.

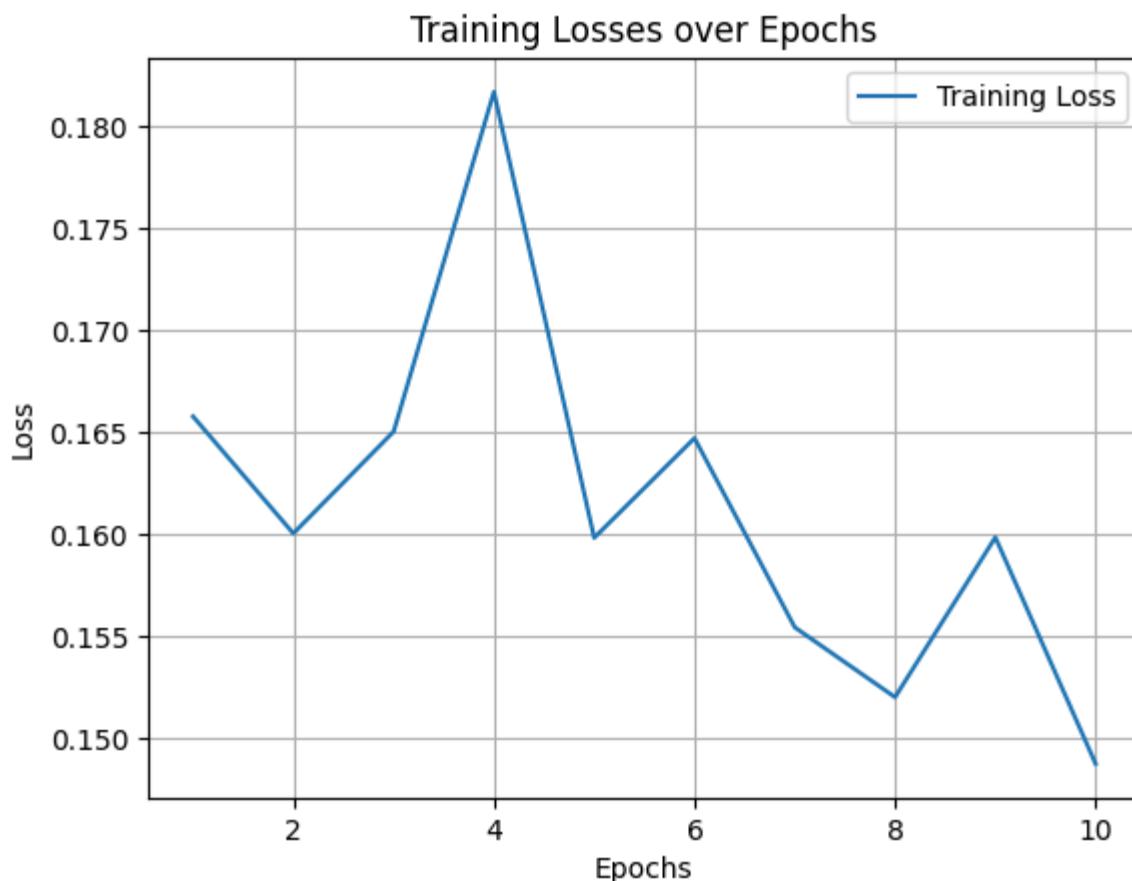
در مورد توابع لاس مورد استفاده KLD در قسمت های قبل توضیح داده شده است. ریکانستراکشن لاس L1 کیفیت بازسازی عکس را مورد بررسی قرار میدهد که برای تمامی عکس ها با سایز  $32 \times 32$  که باعث میشود تا از درستی بازسازی تصاویر در لایه های پیش از اسکیپ کانکشن ها اطمینان حاصل شود و همچنین برای تصویر  $256 \times 256$  منفی نیز این لاس را استفاده میکنیم که موجب بهبود عملکرد بر روی داده نویزی و بازسازی داده های دارای تومور شود.

Triplet loss را نیز جهت اطمینان از فاصله حد اکثری نمونه anchor با نمونه منفی و نزدیکی نمونه anchor و نمونه مثبت استفاده شده است که در بهبود فضای لیتننت کمک میکند.

نهایتاً SSIM loss که میان میان تصویر باز سازی شده نمونه منفی و نمونه قبل از اعمال نویز نمونه منفی اندازه گیری میشود و به مدل کمک میکند تا در تصاویر باز سازی شده خود بهتر به حذف تومور از تصویر بپردازد .

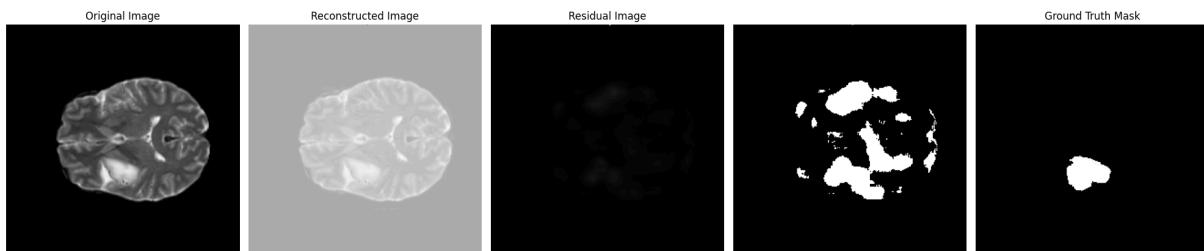
همچنین نمودار لاس در ایپاک های مختلف در شکل زیر امده است لازم به ذکر است که به دلیل زمان بر بودن فرایند ترین و جهت کاهش زمان ترین در هر بار اجرا مدلی پس از 10 ایپاک ترین ذخیر

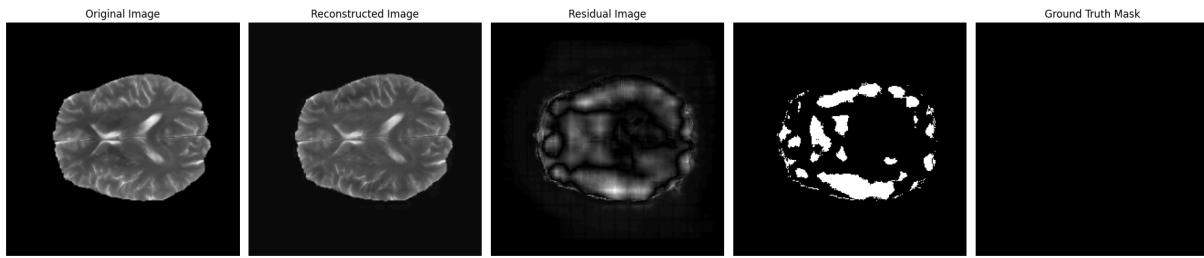
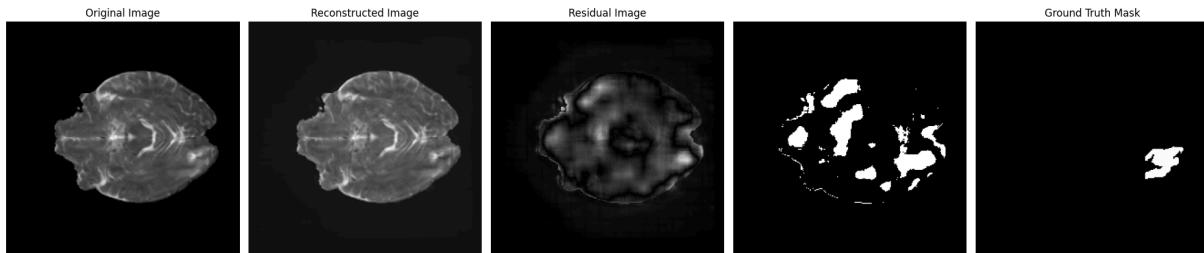
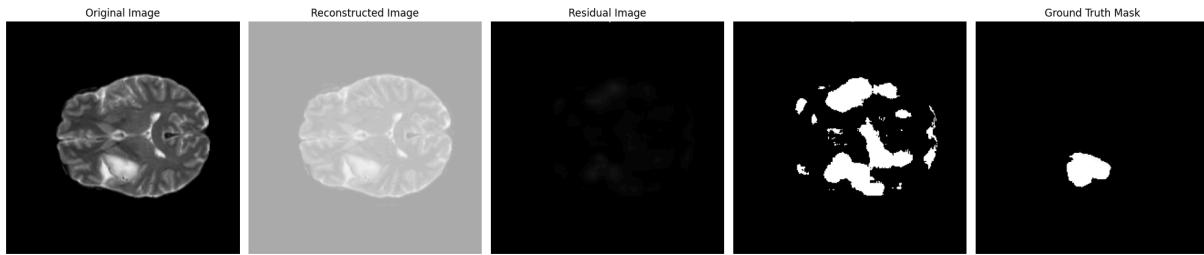
شده و در این بخش و بخش امتیازی این مدل لود شده و پس از لود به ترین ان در 10 اپاک پرداختیم:



#### 4-1. ارزیابی در دیتاست BraTS (دو بعدی)

پس از آموزش مدل به ارزیابی آن میپردازیم :





همانطور که مشاهده میشود مناطق تومور ها به نسبت در این مدل بهتر تشخیص داده شده است.

و دایس این مدل به صورت زیر است :

```
calculate_dice(I_vae, brats_dataset, 0.1)
Average Dice Score: 0.4791
```

همانطور که قابل مشاهده است به نتایج در این مدل به مقدار خوبی از مدل ساده بهتر است که محصول اضافه شدن اسکیپ کانکشن ها و لاس های جدید مورد استفاده است که به بازسازی تصاویر با تومور به شکل تصاویر بی تومور کمک کرده است. گرچه علی رقم تمامی این بهبود ها همچنان میتوان دید که مدل به شدت رنگ ها در عکس های ورودی حساسیت دارد که به نوعی نشانگر حساسیت مدل به دامین ترین شده بر روی آن است و به کمک پیش پردازش و حتی استفاده از تکنیک های دامین ادایپشن امکان بهبود این مسئله وجود دارد.

## 5-1. بخش امتیازی

: در این بخش نیز از مدل بخش قبل استفاده کردیم و تنها تابع ترین به صورت زیر است

```
def train_tri_vae_simplex(
    model,
    anc_dataloader,
    pos_dataloader,
    neg_dataloader,
    dataset,
    device,
    epochs=20,
    learning_rate=3e-4,
    alpha=1,
    beta=1,
    patience=3
):
    optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)
    loss_history = []

    def kld_loss(mu, logvar):
        return -0.5 * torch.sum(1 + logvar - mu.pow(2) -
logvar.exp()) / mu.size(0)

    def ssim_loss(pred, target):
        return 1 - ssim(pred, target)

    def triplet_loss(anchor_mu, positive_mu, negative_mu,
margin=1.0):
        pos_dist = torch.sum((anchor_mu - positive_mu) ** 2,
dim=1).mean()
```

```

        neg_dist = torch.sum((anchor_mu - negative_mu) ** 2,
dim=1).mean()

    return torch.relu(pos_dist - neg_dist + margin)

model.to(device)

model.train()

best_epoch_loss = float('inf')
patience_counter = 0

for epoch in range(epochs):
    epoch_loss = 0
    progress_bar = tqdm(
        zip(anc_dataloader, pos_dataloader, neg_dataloader),
        desc=f"Epoch {epoch + 1}/{epochs}",
        unit="batch",
        total=len(anc_dataloader)
    )

    for (anc_images, _), (pos_images, _), (neg_images, _) in progress_bar:
        anc_images, pos_images, neg_images =
anc_images.to(device), pos_images.to(device), neg_images.to(device)

        noised_negs =
torch.stack([dataset._add_simplex_noise(image) for image in neg_images])

        anc_reconstructed, anc_l4_out, anc_mu, anc_logvar =
model(anc_images)

```

```

        pos_reconstructed, pos_l4_out, pos_mu, pos_logvar =
model(pos_images)

        neg_reconstructed, neg_l4_out, neg_mu, neg_logvar =
model(noised_negs)

        anc_l4_target = nn.functional.interpolate(anc_images,
size=(32, 32), mode='bilinear', align_corners=False)

        pos_l4_target = nn.functional.interpolate(pos_images,
size=(32, 32), mode='bilinear', align_corners=False)

        neg_l4_target = nn.functional.interpolate(neg_images,
size=(32, 32), mode='bilinear', align_corners=False)

l1_anc32 = nn.L1Loss()(anc_l4_out, anc_l4_target)
l1_pos32 = nn.L1Loss()(pos_l4_out, pos_l4_target)
l1_neg32 = nn.L1Loss()(neg_l4_out, neg_l4_target)
l1_neg256 = nn.L1Loss()(neg_reconstructed, neg_images)

kld_anc = kld_loss(anc_mu, anc_logvar)
kld_pos = kld_loss(pos_mu, pos_logvar)

ssim_neg = ssim_loss(neg_reconstructed, neg_images)

l_triplet = triplet_loss(anc_mu, pos_mu, neg_mu)

l1_total = l1_anc32 + l1_pos32 + l1_neg32 + l1_neg256
kld_total = kld_anc + kld_pos
total_loss = l1_total + kld_total + l_triplet +
ssim_neg

optimizer.zero_grad()
total_loss.backward()

```

```

        optimizer.step()

        epoch_loss += total_loss.item()
        progress_bar.set_postfix({"Loss": total_loss.item()})

    avg_epoch_loss = epoch_loss / len(anc_dataloader)
    loss_history.append(avg_epoch_loss)
    print(f"Epoch [{epoch + 1}/{epochs}], Avg Loss: {avg_epoch_loss:.4f}")

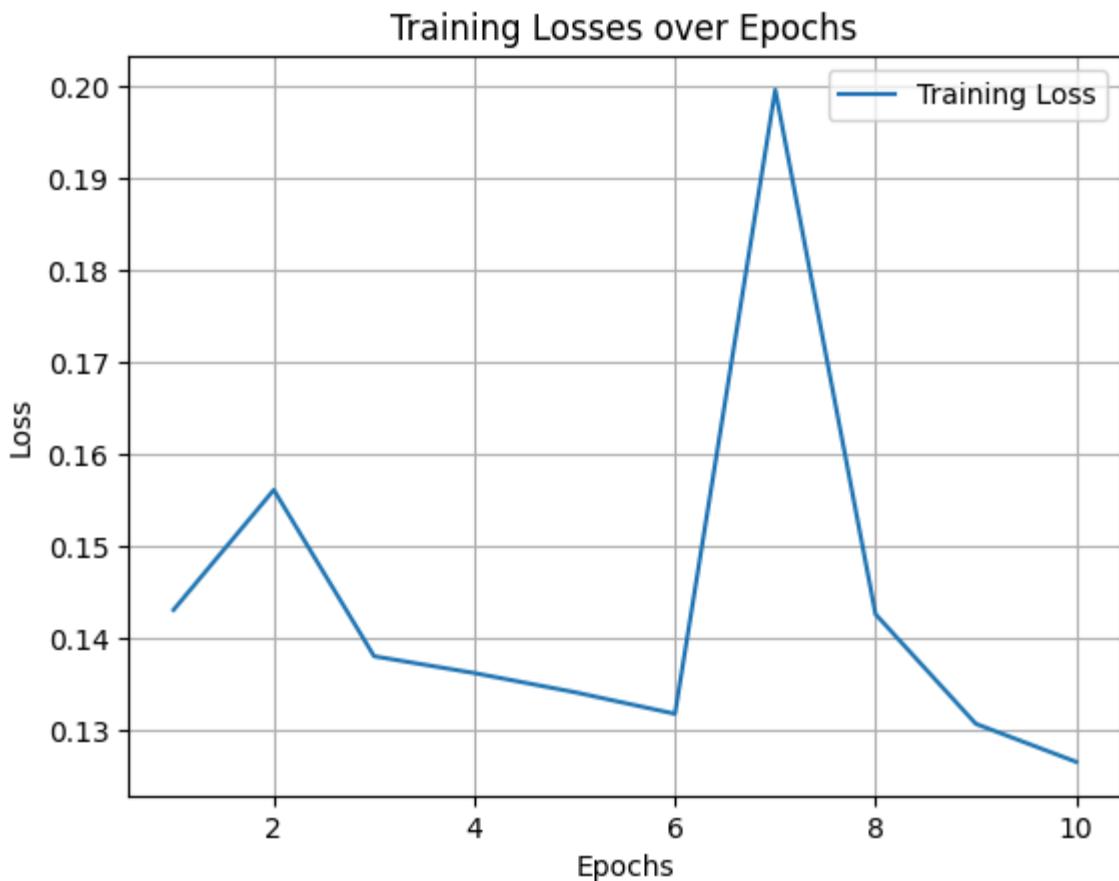
    if avg_epoch_loss < best_epoch_loss:
        best_epoch_loss = avg_epoch_loss
        patience_counter = 0
        torch.save(model.state_dict(), 'best_S_I_model.pth')
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered. Stopping training.")

model.load_state_dict(torch.load('best_I_model.pth'))
break

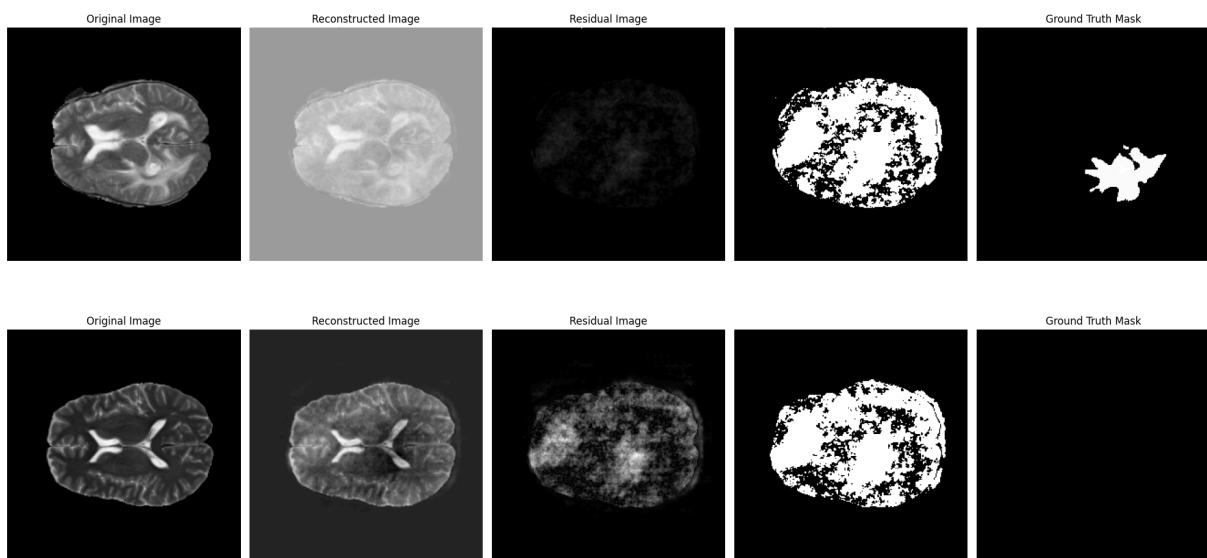
return loss_history

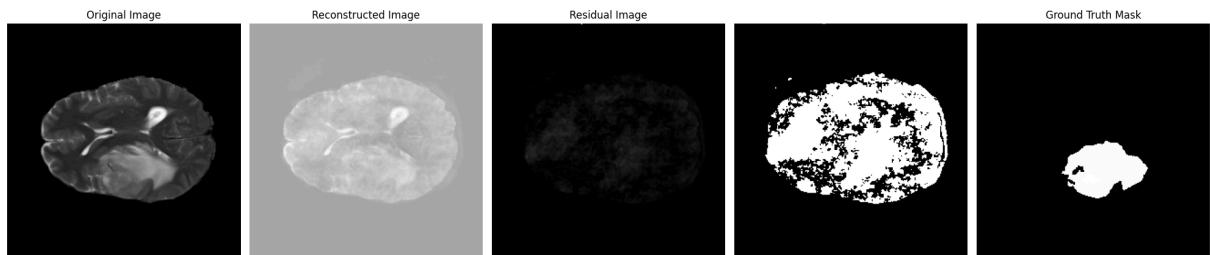
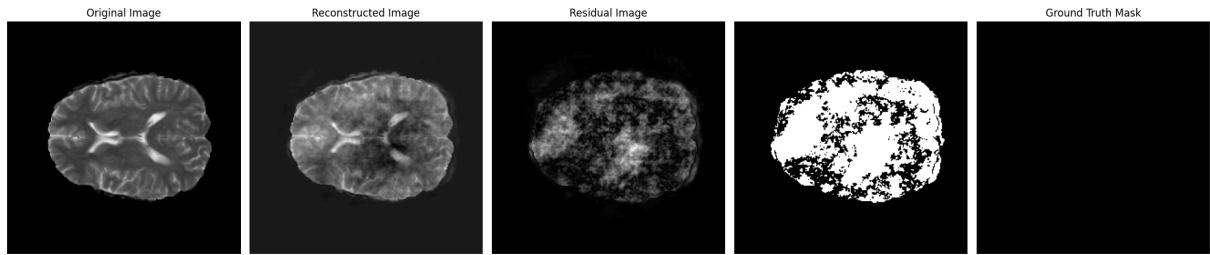
```

که همانطور که قابل مشاهده است تنها تفاوت تغییر نوع نویز اپلای شده است. همچنین لازم به ذکر است که به دلیل زمانبر بودن فرایند ترین در طی فرایند ایجاد تغییرات 1 ورژن از مدل سیو شده و 10 ایپاک بر روی ورژن سیو شده فرایند ترین در نوتبوک قابل مشاهده است و نتایج حاصل شده به شرح زیر است :



همچنین نتیجه اجرای مدل بر روی بعضی از سمپل های مجموعه داده به شکل زیر است :





و دايس حاصل برابر:

```
[1] calculate_dice(simplex_I_vae, brats_dataset, 0.1)
→ Average Dice Score: 0.5081
```

## پرسش 2 - AdvGAN

### 1-2. آشنایی با حملات خصمانه و معمای AdvGAN

1. روش‌های دیگر تولید نمونه‌ها تخاصمی به مانند **FGSM** و **PGD** را توضیح دهید و بیان بدارید مزیت یا مزیت‌های مدلی مانند **AdvGAN** نسبت به روش‌های دیگر چیست؟

هزینه استفاده می‌کند تا یک اختلال کوچک اما مؤثر برای تغییر پیش‌بینی مدل تولید کند. این اختلال به صورت خطی با استفاده از علامت گرادیان محاسبه می‌شود. مزیت اصلی FGSM سادگی و سرعت آن است، اما ممکن است تصاویر تولیدی کیفیت بصری پایینی داشته باشند. عموماً از آن برای تحلیل‌های اولیه که نیازمند مدل با محاسبات سریع اختلال است، استفاده می‌شود.

Projected Gradient Descent **PGD** یا دقیق‌تر **FGSM**، یک نسخه بهینه‌تر از FGSM است که از چندین تکرار گرادیانی برای تولید اختلال استفاده می‌کند. به بیان جزئی‌تر، در هر مرحله نمونه به فضایی مشخص و مجاز محدود می‌شود و در نتیجه اختلال محاسبه شده از محدوده مشخص فراتر نخواهد رفت. PGD دقیق‌تر در موفقیت حملات دارد، اما زمان بیشتری برای تولید نمونه‌ها نیاز دارد. با این حال، هزینه زمانی آن نسبت به مدل قبلی بالاتر است.

توضیحات مربوط به مدل AdvGAN به بخش پیاده‌سازی منتقل شده است.

برخلاف FGSM و PGD که برای هر تصویر باید اختلال جدیدی محاسبه شود، در **AdvGAN** یک شبکه تولید کننده (Generator) آموخت داده می‌شود که می‌تواند به صورت مستقیم اختلال‌های لازم را تولید کند. این روش همچنین کیفیت بصری بالاتری برای نمونه‌های تخاصم ارائه می‌دهد و حملات را سریع‌تر اجرا می‌کند. این مدل از شبکه مولد تخاصمی یا GAN استفاده می‌کند. بخش Generator معماری اختلال‌های خروجی برای اضافه شدن به تصویر را بدون محاسبات گرادیانی تولید می‌کند که سرعت این فرایند را نسبت به دو مدل قبلی افزایش می‌دهد. بخش دیگر این معماری آن است که تلاش می‌کند اختلالات اضافه شده به تصویر واقعی را بیابد و تصاویر واقعی را از adv تمایز دهد. با آموخت این دو مدل که در خلاف و ضد هم عمل می‌کنند، در نهایت به تعادل میرسیم که جفت مدل‌های آموخته‌های کارایی دارند. این مدل قابلیت استفاده در قالب semi white box و همینطور black box و white box را دارد.

2. تفاوت‌های کلیدی بین **GAN** و یک **AdvGAN** ساده را با تمرکز بر موارد زیر توضیح دهید.

## • چگونه AdvGAN از گرادیان‌ها یا خروجی‌های مدل هدف در زمان آموزش استفاده می‌کند؟

در GAN ساده، هدف تولید تصاویر واقعی از نویز یا داده‌های اولیه است، اما در AdvGAN هدف تولید تصاویری است که علاوه بر حفظ شباهت بصری به تصویر اصلی، مدل هدف را فریب دهنده. در واقع مدل GAN ساده از مدل هدفی استفاده نمیکند و در AdvGAN یک مولفه اضافی به نام "مدل هدف" اضافه شده که وظیفه آن بهبود حملات تخاصمی است. این مدل بر اساس خطای پیش‌بینی خود در برابر نمونه‌های تولیدی آموزش می‌بیند. در یک مدل GAN، گرادیان‌ها از طریق generator با کمک آنها فرآیند آموزش خود را انجام discriminator می‌دهد.

در AdvGAN، گرادیان‌های مدل هدف برای بهینه‌سازی شبکه تولیدکننده به کار می‌روند تا اختلال‌هایی تولید شوند که بیشترین تأثیر را در تغییر خروجی مدل داشته باشند. در این مورد از آموزش مدل generator به دنبال فریب discriminator و به خطا انداختن مدل هدف استفاده شده هستیم. بنابراین مدل generator از گرادیان‌های تابع هزینه مدل هدف برای آموختن تولید نویز هایی استفاده خواهد کرد تا با اعمال آن روی تصویر واقعی، تفاوت زیادی از لحاظ شهودی صورت نگیرد و در عین حال مدل هدف را در مسئله classification به اشتباہ بیندازد.

## • توضیح دهید که چگونه AdvGAN نمونه‌های تخاصم تولید می‌کند و چگونه این مدل قادر است همزمان وفاداری بصری به تصویر اصلی و قابلیت حمله به مدل را حفظ کند.

از یک شبکه تولید کننده (Generator) استفاده می‌کند که اختلال‌هایی را به تصویر اصلی اضافه می‌کند. این اختلال‌ها به گونه‌ای طراحی می‌شوند که تصویر تولیدی از نظر بصری شبیه تصویر اصلی باشد (برای حفظ وفاداری بصری) اما در عین حال مدل هدف را فریب دهد. این مدل نویزی خروجی می‌دهد که با تصویر اضافه خواهد شد و تصویر adv را شکل خواهد داد. شبکه متمایزکننده (Discriminator) نیز کمک می‌کند تا تصاویر تولیدی از نظر بصری واقع‌گرایانه به نظر برسند. توابع هزینه استفاده شده در این مدل از سه تابع تشکیل شده که در بخش بعد به تفصیل توضیح داده شده است. فرآیند آموزش این معماری، به تولید تصاویر مشابه به تصاویر اولیه اما متفاوت در نویز حاصل می‌شود که مدل discriminator و مدل هدف را به اشتباہ بیندازد. در یک GAN ساده، generator غالباً از بردار نویز تصادفی استفاده می‌کند و با اعمال روی تصویر، تصاویری جدید اما مشابه تصاویر اولیه تولید خواهد شد. تابع هزینه این مدل از نوع هزینه GAN بوده که تنها اختلاف تصاویر اصلی و adv را محاسبه و دخالت می‌دهد. هر چقدر این تصاویر به اصلیت

خودشان مشابه تر باشند، discriminator بیشتر به اشتباه میفتند. همانطور که گفته شد، در یک GAN ساده از مدل هدف استفاده نمیشود بنابراین مشخص نیست این تصاویر تولید مدل هدفی که بعدها تعریف بشود را به اشتباه بیندازند یا خیر.

3. سهتابع هزینه اصلی استفاده شده در AdvGAN را با ذکر روابط ریاضی شرح دهید و توضیح دهید که این عبارات هر کدام چگونه به کیفیت نمونه‌های متخاصم و مقاوم‌سازی مدل کمک می‌کنند.

- تابع هزینه  $L_{GAN}$  که فرمول آن به شرح زیر است:

$$L_{GAN} = E_x \log D(x) + E_x \log (1 - D(x + G(x)))$$

که در آن از discriminator به نماد  $D$  استفاده شده است که سعی می‌کند داده با آشفتگی یعنی  $x + G(x)$  را به گونه‌ای از تصویر اصلی یعنی  $x$  تشخیص دهد. هدف این تابع کمک به آموزش مدلی است که تصاویر آشفته را به گونه‌ای نزدیک کلاس هدف بسازد، اما نه دقیقاً همان تصویر، تا مدل discriminator در تفکیک بین داده واقعی و adv دچار مشکل و مدل هدف در تشخیص کلاس‌بندی تصویر ورودی دچار خطأ شود.

در این تابع دچار یک مسئله  $\min \max$  هستیم. مدل  $D$  تمایل دارد مقدار آن را  $\text{maximize}$  کند تا تفکیک داده واقعی و adv به بهترین حالت صورت پذیرد. تابع  $G$  تمایل دارد بخشی که در آن دخالت دارد را  $\text{minimize}$  کند تا داده‌های تولید شده به داده‌های اولیه شباهت خوبی داشته باشند.

- تابع هزینه  $L_{adv}^f$  که فرمول آن به شرح زیر است:

$$L_{adv}^f = E_x l_f(x + G(x), t)$$

که در آن  $t$  نشان‌دهنده کلاس هدف است.  $l_f$  نشان‌دهنده تابع خطا مانند cross-entropy loss می‌باشد که از آن برای آموزش مدل هدف استفاده شده است. تابع خطای  $L_{adv}^f$  به گونه‌ای عمل می‌کند تا تصاویر تولید شده به عنوان ورودی مدل هدف، به اشتباه دسته‌بندی شوند و در اصل مدل در این حمله گول بخورد. در حملات از نوع targeted، با استفاده از این تابع مدل وادار می‌شود نمونه را به یک کلاس به خصوصی نزدیک کند.

- تابع هزینه  $L_{hinge}$  که فرمول آن به شرح زیر است:

$$L_{hinge} = E_x \max(0, \|G(x)\|_2 - c)$$

که در آن  $c$  یک متغیر حاوی مقدار bound مشخص شده توسط کاربر می‌باشد. هدف استفاده از این تابع خطأ، محدودسازی اندازه بزرگ تولید آشفتگی روی داده اصلی می‌باشد. در این تابع از نرم دوم استفاده شده است. این تابع خطأ میزان اختلالها را محدود می‌کند تا تصاویر تولیدی به تصویر اصلی نزدیک بمانند.

همانطور که ذکر شد، در مدل‌های AdvGAN از ترکیب این سه تابع به صورت زیر استفاده می‌شود:

$$L = \alpha L_{GAN} + L_{adv}^f + \beta L_{hinge}$$

**4. تفاوت بین حمله‌های جعبه سفید جعبه سیاه را توضیح دهید و بیان کنید مدل ذکر شده چگونه می‌تواند در حملات جعبه سیاه استفاده شود؟**

حملات جعبه سفید حملاتی هستند که در آن مهاجم به تمام جزئیات مدل هدف (معماری و وزن‌ها) دسترسی دارد و با استفاده از این اطلاعات نظیر میزان گرادیان نمونه‌های adv را تولید می‌کند و در نتیجه، نتیجه این حمله دقیق خوبی دارد. یک نمونه آن FGSM می‌باشد. حملات جعبه سیاه حملاتی هستند که در آن مهاجم تنها به ورودی و خروجی مدل دسترسی دارد. برای مثال مهاجم یک مدل محلی آموزش میدهد و به مرور بر اساس خروجی گرفته شده، این مدل را تغییر میدهد تا به نتایج بهتری برسد.<sup>3</sup>

AdvGAN می‌تواند در حملات جعبه سیاه از طریق ساخت یک مدل جایگزین (Distilled Model) عمل کند که رفتار مدل هدف را تقلید می‌کند تا به حد ممکن خروجی‌ها اختلاف کمتری داشته باشند. سپس اختلالها بر اساس این مدل جایگزین تولید می‌شوند.

در حملات جعبه‌سیاه، AdvGAN می‌تواند از روش‌های Dynamic Queries و Model Distillation برای تولید نمونه‌های متخصص بهره ببرد. ابتدا یک مدل Distilled که رفتار مدل جعبه‌سیاه را تقلید می‌کند، آموزش می‌بیند و هدف آن کاهش اختلاف بین خروجی‌های دو مدل است. اگه از روش static استفاده کنیم، مدل Distilled با استفاده از داده‌های جداگانه آموزش داده خواهد شد و سپس Generator با خروجی این مدل، نمونه‌های متخصص را تولید می‌کند. در روش dynamic، فرآیند بهروزرسانی مدل generator و Distilled مورد استفاده قرار می‌گیرند. در این روش کوئری‌هایی توسط generator برای بهبود مدل Distilled مورد استفاده قرار می‌گیرند. در این روش کوئری‌هایی

---

<sup>3</sup> به این پدیده، انتقال پذیری هم می‌گویند.

که به صورت داینامیک از مدل هدف دریافت میشوند، به تولید نمونه‌های متخصص با دقت بالاتر کمک میکند..

۵. دو مقاله پژوهشی که AdvGAN را گسترش یا بهبود می‌دهند پیدا کنید و هرکدام را دو یک الی دو پاراگراف خلاصه کنید. همچنین توضیح دهید که این مقالات چگونه بر اساس چارچوب اولیه AdvGAN ایده‌های خود را توسعه داده‌اند.

#### **۲۰۱۹: بهره‌برداری از لایه‌های نهان برای تولید نمونه‌های تخصصی**, سال انتشار:

این مقاله نسخه بهبودیافته‌ای از AdvGAN به نام ++AdvGAN را معرفی می‌کند که از ویژگی‌های نهان به عنوان ورودی برای تولید نمونه‌های adv بهره می‌برد. در حالی که AdvGAN از تصویر ورودی به عنوان ورودی شبکه generator استفاده می‌کند، ++AdvGAN نشان می‌دهد که استفاده از ویژگی‌های نهان استخراج شده می‌تواند نرخ موفقیت حملات را افزایش داده و تصاویر adv با کیفیت‌تری تولید کند. آزمایش‌ها بر روی مجموعه داده‌های MNIST و CIFAR-10 نشان می‌دهد که ++AdvGAN عملکرد بهتری دارد.

#### **ویرایش گرادیان: بهبود انتقال پذیری نمونه‌های تخصصی با مدل مولد تخصصی مبتنی بر**

**۲۰۲۴: ویرایش گرادیان**, سال انتشار:

این مقاله الگوریتم جدیدی به نام GE-AdvGAN را برای افزایش انتقال پذیری نمونه‌های تخصصی معرفی می‌کند. این الگوریتم با بهینه‌سازی فرآیند آموختن شبکه generator و معرفی مکانیزم ویرایش گرادیان، نمونه‌های adv با قابلیت انتقال بالا تولید می‌کند که می‌توانند مدل‌های مختلف هدف را فریب دهند. استفاده از اطلاعات حوزه فرکانس برای تعیین جهت ویرایش گرادیان، به GE-AdvGAN امکان می‌دهد تا نمونه‌های adv با کیفیت بالا و کارایی زمانی بهتر نسبت به الگوریتم‌های پیشرفته موجود تولید کند.

## **2-2. پیاده‌سازی مدل AdvGAN**

در ابتدا مجموعه داده CIFAR10 را دانلود کرده و به سه دسته train و validation و test تقسیم می‌کنیم. نسبت‌های تقسیم بندی به صورت ۸۰ - ۱۰ - ۱۰ تنظیم شده‌اند.

در ادامه، ۵ نمونه تصویر از هر دسته به ترتیب ذکر شده را نمایش می‌دهیم:



برای نرمالسازی پیکسل‌های تصاویر به کمک قطعه کد زیر، میانگین و انحراف معیاری به دست آمده از داده‌ها، فرآیند نرمالیزیشن را انجام می‌دهیم.

```
dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
print(dataset.data.shape)
print(dataset.data.mean(axis=(0, 1, 2)) / 255)
print(dataset.data.std(axis=(0, 1, 2)) / 255)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

# Split training dataset into training and validation
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%[██████████] | 170498071/170498071 [00:03<00:00, 53919426.85it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
(50000, 32, 32, 3)
[0.49139968 0.48215841 0.44653091]
[0.24703223 0.24348513 0.26158784]
Files already downloaded and verified
```

اصل فرآیند نرم‌السازی با استفاده از transform تعریف شده در کد زیر انجام شده است:

```
batch_size = 128
mean      = [0.49139968, 0.48215841, 0.44653091]
std       = [0.24703223, 0.24348513, 0.26158784]

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

CLASSES = [
    'airplane', 'automobile', 'bird', 'cat', 'deer',
    'dog', 'frog', 'horse', 'ship', 'truck'
]
num_classes = len(CLASSES)
```

در ادامه از مدل از پیش آموزش دیده شده ResNet-20 استفاده شده است:

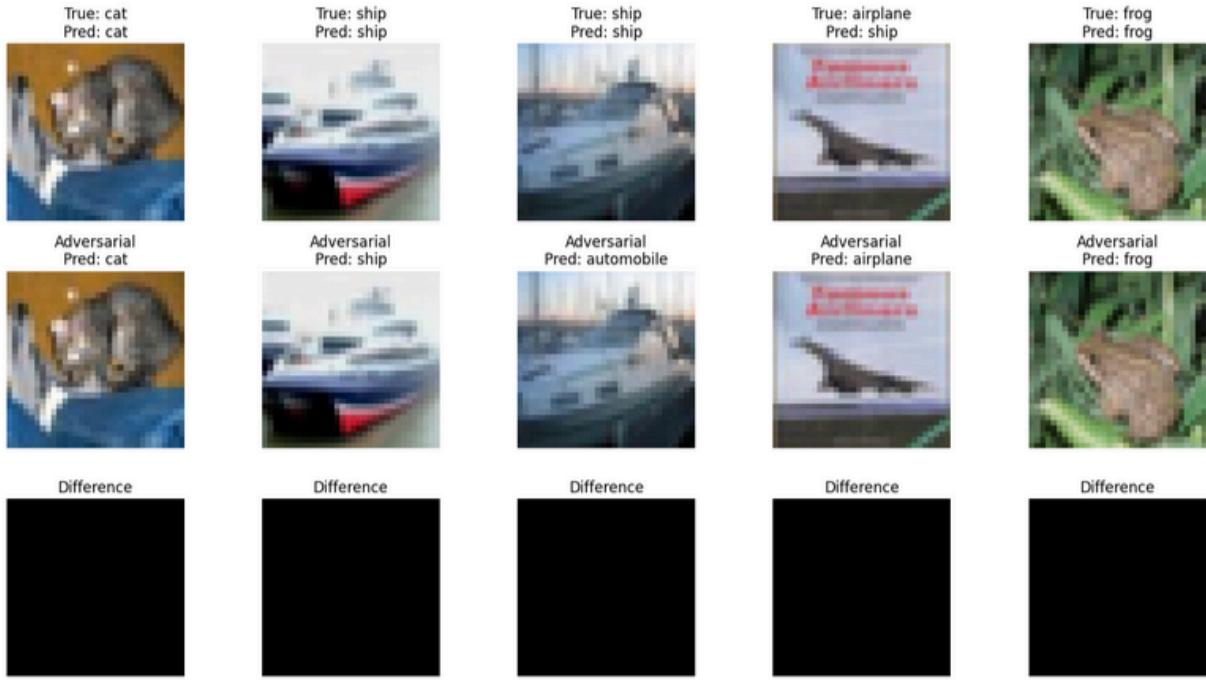
جزئیات ریز معماری این مدل در فایل ipynb قابل مشاهده است و در حالت eval قرار می‌دهیم.

مجموعه test set را به کمک مدل فوق طبقه بندی کردہ‌ایم.

دقت مدل هدف روی این مجموعه به عدد 92.12 ثبت شده است.

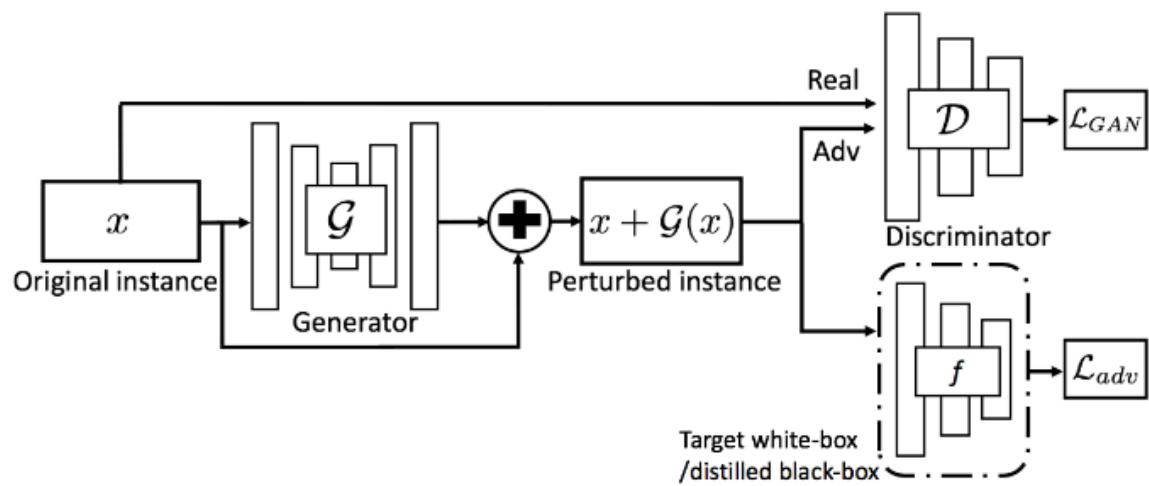
همینطور به کمک adversarial ، تصاویر fast\_gradient\_method خلق شده و میزان نرخ موفقیت حمله به عدد 36.32 درصد رسیده است.

در ادامه چند نمونه از تصاویر واقعی، در کنار نمونه‌های adversarial از مجموعه تست را نمایش داده ایم:



میزان تفاوت دو تصویر به قدری کوچک و نزدیک به صفر هستند که به چشم در سطر قابل مشاهده نیستند.

در ادامه‌ی این بخش شمای کلی از یک AdvGAN نشان داده شده است:



همانطور که مشخص است، این معماری شامل یک Generator بوده که نمونه اصلی تصویر را دریافت کرده و سعی می‌کند نمونه‌ی تصویر ورودی مدل هدف  $f$  را به نحوی آماده کند که از نظر بصری، تفاوت چندانی با نمونه اصلی نداشته باشد و در عین حال، به قدری متفاوت باشد که مدل  $f$  را به اشتباه بیندازد.

برای مقایسه تصویر ورودی مدل  $f$  و تصویر اصلی، مدل Discriminator را آموزش می‌دهیم که با مقایسه تابع خطای  $L_{GAN}$ ، فرآیند یادگیری را انجام می‌دهد. مدل هدف  $f$  نیز از تابع خطای  $L_{adv}$  استفاده می‌کند.

ما در این مسئله برای محاسبه خطای کلی از فرمول زیر بهره می‌ریم:

$$L = L_{adv}^f + \alpha L_{GAN} + \beta L_{hinge}$$

که در آن مقادیر  $\alpha$  و  $\beta$  اهمیت نسبی خطاهای متناسب با خود را تنظیم می‌کنند.

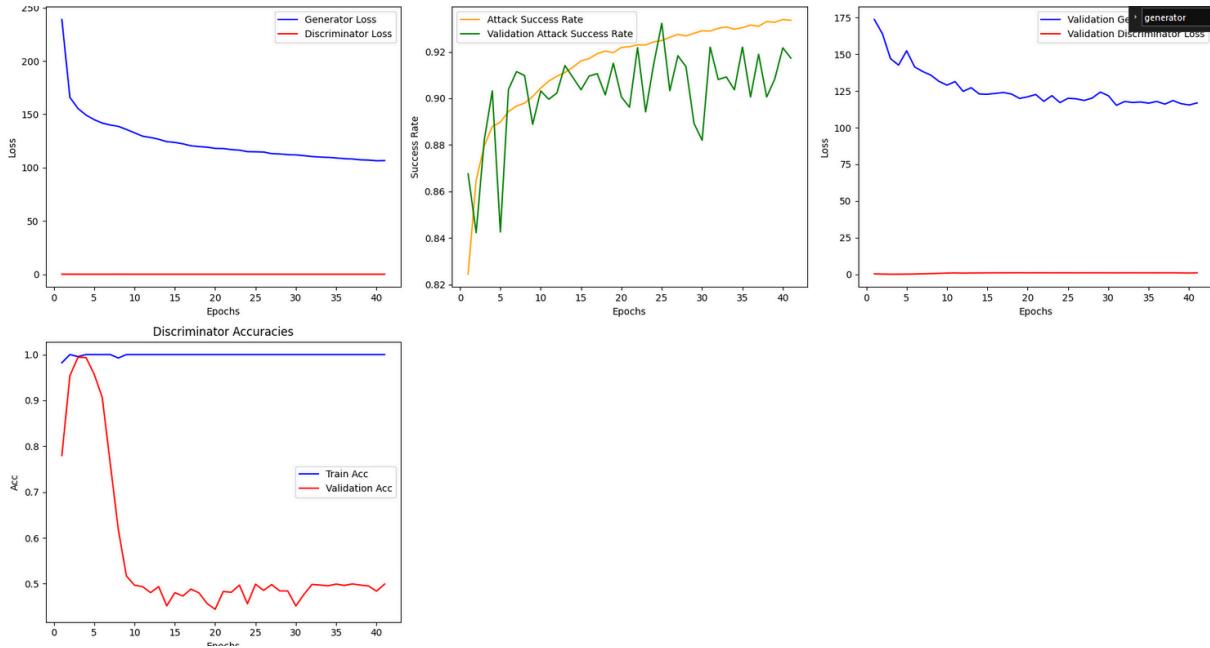
مدل‌های Generator و Discriminator را به کمک هایپر پارامترهای مقاله تعریف کردیم:

Batch size	Lr	Epoch count	patience	Alpha	Betta	C	Epsilon
128	0.01	50	10	15	10	8/25	0.01

Optimizer	Scheduler
Adam	StepLR

حال به بررسی نتایج می‌پردازیم:

همانطور که مشخص است، بعد از حدود 44 ایپاک، مدل شروع به Overfit می‌کند و فرآیند early stopping، فعال شده است.



با بررسی نتایج نمودارها در شکل فوق، متوجه می‌شویم که میزان تغییرات تابع خطا چه روی train set و چه روی validation set برای مدل Generator نزولی بوده که نشان دهنده آموزش خوب این مدل است. میزان لاس مدل Discriminator همواره مقدار اندک بوده که طبق تحقیقات و مطالعه چندین لینک گیت‌هاب، متوجه شدیم در مدل‌های GAN فرایندی معمول است.

همانطور که در نمودار دوم مشاهده می‌شود، میزان Attack Success Rate به مرور زمان روی داده train افزایش یافته و روی داده تست، شامل مقداری fluctuation بوده که به علت ماهیت validation attack success و مدل استفاده شده، طبیعی است. با این حال، روند کلی rate به طور کلی صعودی می‌باشد. اما به مرور شبیه صعود کلی آن کاهش یافته که نشان دهنده شروع فرآیند overfitting مدل روی داده ترین است که با اعمال early stopping از روی دادن آن جلوگیری کردہ‌ایم.

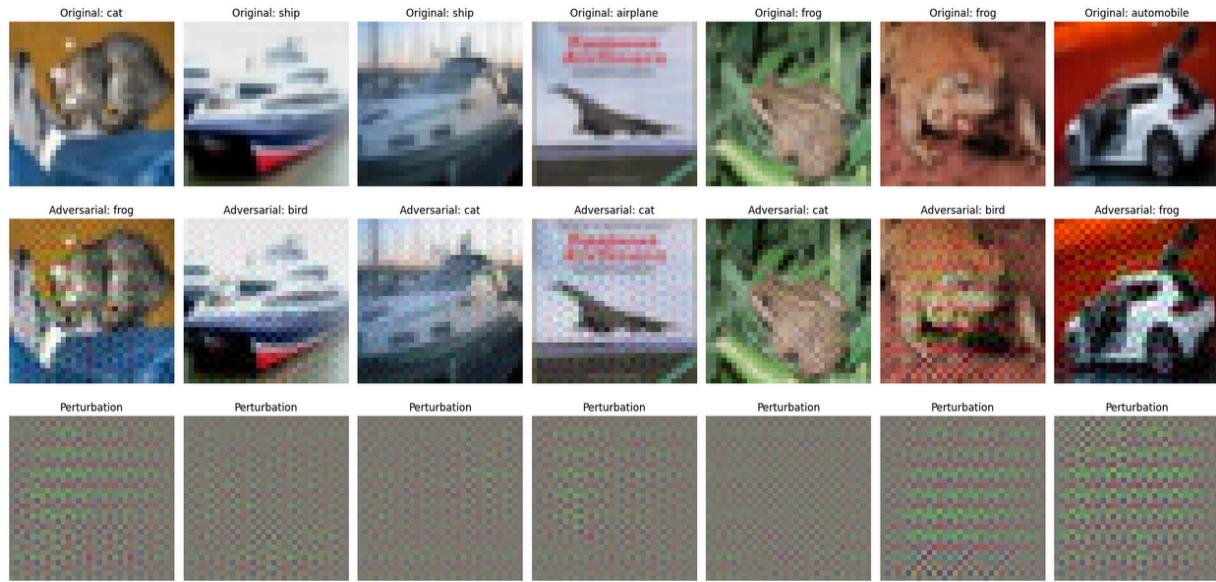
با دقت به نمودار دقت discriminator متوجه می‌شویم دچار اورفیت شده که علت آن رقابت بین discriminator و generator می‌باشد. اما با این حال، روی داده تست، نتیجه مد نظر ما حاصل شده است و طبق مشاوره با تی‌ای، قابل قبول می‌باشد.

توجه کنید که در طول فرآیند ترین، همواره بهترین مدل سیو شده در output را به همراه پارامترهای آن لود کرده و روی داده تست اجرا می‌کنیم.

در بخش زیر، نرخ موفقیت حمله به طور کلی و به تفکیک کلاس را نشان داده ایم.

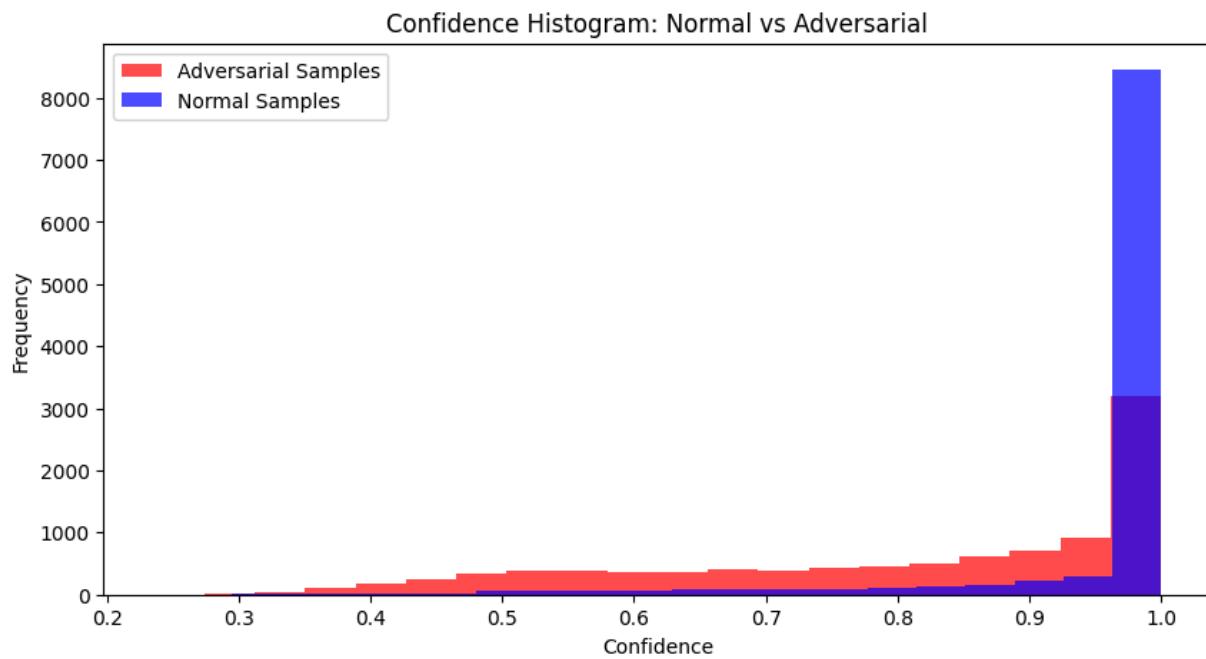
Class number	Class name	Attach success rate
Overall	Overall	92.17
1	Airplane	97.80
2	Automobile	98.50
3	Bird	77.90
4	Cat	76.50
5	Deer	94.40
6	Dog	99.90
7	Frog	80.30
8	Horse	99.80
9	Ship	98.00
10	Truck	98.60

حال چندین نمونه تصویر تولید شده توسط مدل G به همراه نمونه واقعی آن و تفاوت دو تصویر را مشاهده می‌کنیم:



همانطور که در تصویر بالا مشاهده می شود، تشابهای در تفاوت تصویر اصلی و  $adv$  از لحاظ پیکسل در کلاس‌هایی که به اشتباه پیش‌بینی شده است، وجود دارد. برای مثال، تصویر اول و دو تصویر از آخر، اختلاف پیکسل‌ها از لحاظ RGB الگوی تاحدودی مشابه را دنبال می‌کنند.

در بخش زیر نمودار histogram مربوط به میزان اطمینان مدل روی داده تست در هر دو حالت adversarial samples و normal samples نشان داده شده است:



نمونه‌های عادی با رنگ آبی و نمونه‌های حمله‌ای با رنگ قرمز نمایش داده شده‌اند. همانطور که مشاهده می‌شود، نمونه‌های عادی بیشتر در نزدیکی مقدار اعتماد ۱ قرار دارند، در حالی که نمونه‌های حمله‌ای در مقادیر اعتماد پایین‌تر توزیع شده‌اند. این نشان می‌دهد که مدل در تشخیص نمونه‌های عادی با اعتماد بالا عملکرد خوبی دارد، اما در مواجهه با نمونه‌های حمله‌ای اعتماد کمتری دارد.