به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

نام و نام خانوادگی	محمدامین یوسفی
شماره دانشجویی	810100236
نام و نام خانوادگی	محمدرضا نعمتی
شماره دانشجویی	810100226

فهرست

پرسش ۱ – طراحی و پیاده‌سازی Triplet VAE برای تشخیص تومور در MRI 4

۱-۱. هدف و دیتاست 4

۱-۲. پیاده‌سازی یک VAE ساده 4

۱-۳. پیاده‌سازی Tri-VAE 11

۱-۴: ارزیابی در دیتاست BraTS (دو بعدی) 21

پرسش ۲ – AdvGAN 24

۲-1. آشنایی با حملات خصمانه و معماری **AdvGAN** 24

۲-2: پیاده‌سازی مدل **AdvGAN** 29

شکل‌ها و جدول‌ها

شکل 1-1. نمونه‌هایی از دیتاست IXI 4

شکل 1-2. نمونه‌های T2 به همراه ماسک seg از دیتاست BraTS 4

جدول 1-2. هایپرپارامترهای استفاده شده در مدل VAE 8

شکل 1-3. خطای آموزش مدل VAE در هر ایپاک 8

شکل 1-4. نمونه‌هایی از ارزیابی مدل VAE بر روی BraTS 10

شکل 1-5. تصویر نمونه از Anchor، Positive و Negative به همراه نویز Coarse 14

شکل 1-6. تصویر نمونه از Anchor، Positive و Negative به همراه نویز Simplex 15

شکل 1-2. هایپرپارامتر های مدل Tri-VAE 20

شکل 1-7. خطای آموزش مدل Tri-VAE با نویز Coarse در هر ایپاک 20

شکل 1-8. نمونه‌هایی از ارزیابی مدل Tri-VAE بر روی دیتاست BraTS 22

شکل 2-1. 5 نمونه تصادفی از مجموعه دادگان 30

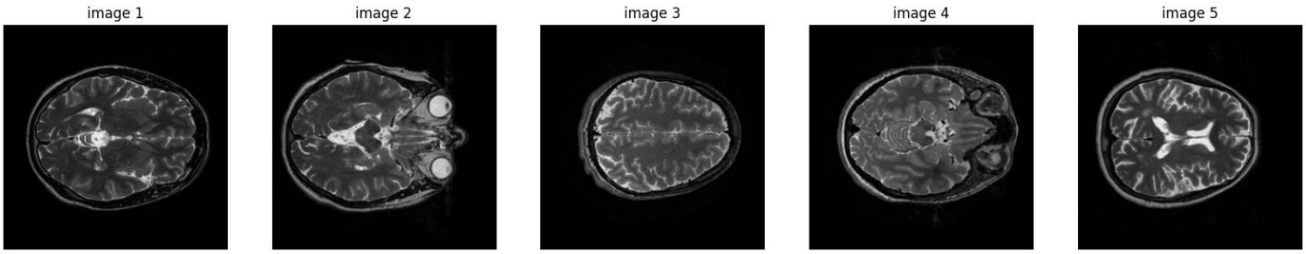
شکل 2-2. 5 نمونه از داده‌های متخاصم به روش FGSM 31

شکل 2-4. 5 نمونه از داده‌های متخاصم به روش AdvGAN 34

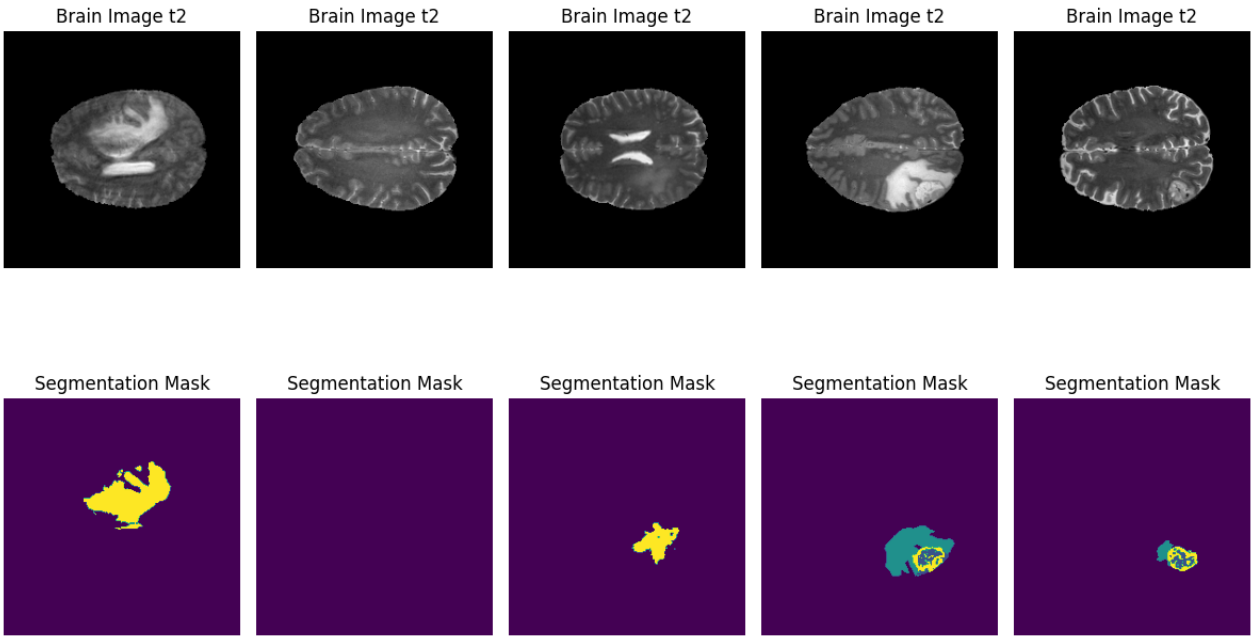
پرسش ۱ – طراحی و پیاده‌سازی Triplet VAE برای تشخیص تومور در MRI

۱-۱. هدف و دیتاست

پس از بارگذاری دیتاست‌های IXI و BraTS، نمونه‌هایی از آن‌ها را در شکل‌های 1-1 و 1-2 نمایش داده‌ایم.



شکل 1-1. نمونه‌هایی از دیتاست **IXI**



شکل 1-2. نمونه‌های **T2** به همراه ماسک **seg** از دیتاست **BraTS**

۱-۲. پیاده‌سازی یک VAE ساده

معرفی مختصر VAE

Variational Autoencoder نوعی Generative Model است که به منظور یادگیری و تولید داده‌های پیچیده، مانند تصاویر یا صدا، استفاده می‌شود. این مدل از دو بخش اصلی Encoder و Decoder تشکیل شده است. بخش Encoder داده ورودی را به یک فضای فشرده‌تر به نام Latent Space نگاشت می‌کند که

در آن اطلاعات کلیدی داده‌ها به صورت فشرده ذخیره می‌شود. این فضا نه تنها امکان کاهش ابعاد داده را فراهم می‌کند، بلکه توزیع احتمالاتی مناسبی از داده‌ها ایجاد می‌کند. برای اینکه Latent Space ویژگی‌های مورد نظر را به درستی بازنمایی کند، از یک روش مبتنی بر KL-Divergence استفاده می‌شود. این روش معیاری برای اندازه‌گیری تفاوت بین دو توزیع احتمالاتی است و در اینجا برای اطمینان از تطابق توزیع داده‌های نهانی با توزیع نرمال استاندارد استفاده می‌شود. این امر موجب می‌شود مدل توانایی نمونه‌گیری از فضای نهانی را داشته باشد و داده‌های جدید تولید کند.

بخش Decoder مسئول بازسازی داده‌های اصلی از Latent Space است. هدف این است که خروجی بازسازی‌شده به داده‌های ورودی اصلی شباهت زیادی داشته باشد. این فرایند از طریق کمینه‌سازی Reconstruction Loss و ترکیب آن با KL Divergence انجام می‌شود. Reconstruction Loss معیاری است که تفاوت بین داده ورودی اصلی و داده بازسازی‌شده توسط Decoder را اندازه‌گیری می‌کند. هدف آن است که بازسازی‌ها تا حد ممکن به داده اصلی نزدیک باشند و مدل بتواند ویژگی‌های کلیدی داده را به‌درستی حفظ کند.

پیاده‌سازی مدل

```
class VAE(nn.Module):
    def __init__(self, latent_dim=16):
        super(VAE, self).__init__()
        self.encoder_conv1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),  # 256x256 -
    > 128x128
            nn.ReLU(),
        )
        self.encoder_conv2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),  # 128x128
    -> 64x64
            nn.ReLU(),
        )
        self.encoder_conv3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),  # 64x64 -
    > 32x32
            nn.ReLU(),
        )

        self.flatten = nn.Flatten()
        self.fc_mu = nn.Linear(128 * 32 * 32, latent_dim)
        self.fc_logvar = nn.Linear(128 * 32 * 32, latent_dim)

        self.fc_dec = nn.Linear(latent_dim, 128 * 32 * 32)
```

```
        self.decoder_deconv3 = nn.Sequential(
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2,
    padding=1),  # 32x32 -> 64x64
            nn.ReLU(),
        )

        self.decoder_deconv2 = nn.Sequential(
            nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),  #
    64x64 -> 128x128
            nn.ReLU(),
        )

        self.decoder_deconv1 = nn.Sequential(
            nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2, padding=1),  #
    128x128 -> 256x256
            nn.Sigmoid(),
        )
    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        x1 = self.encoder_conv1(x)
        x2 = self.encoder_conv2(x1)
        x3 = self.encoder_conv3(x2)

        h_flat = self.flatten(x3)
        mu = self.fc_mu(h_flat)
        logvar = self.fc_logvar(h_flat)
        z = self.reparameterize(mu, logvar)

        h_dec = self.fc_dec(z).view(-1, 128, 32, 32)

        h_dec = self.decoder_deconv3(h_dec)
        h_dec = self.decoder_deconv2(h_dec)
        x_recon = self.decoder_deconv1(h_dec)

        return x_recon, mu, logvar

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

در این پیاده‌سازی با اجرای تابع forward که وظیفه اصلی آموزش مدل را دارد، تصویر ورودی که به سایز 256 * 256 است در لایه اول Encoder به سایز 128*128، در لایه دوم به 64*64 و در لایه سوم به 32*32 تبدیل می‌شود. سپس باید از Encoder به Latent Space منتقل و برای Decoder آماده

شود. ابتدا ویژگی‌های فشرده‌شده با flatten آماده می‌شوند و سپس از طریق دو لایه fc_mu و fc_logvar، میانگین و واریانس برای توزیع احتمالاتی Latent Space محاسبه می‌گردند. سپس reparametrize انجام می‌شود که نمونه‌ای از این توزیع گرفته می‌شود تا اجازه دهد عملیات نمونه‌گیری به‌صورت قابل تفکیک توسط گرادیان انجام شود این بخش برای برای آموزش با Back Propagation ضروری است. در آخر وکتور نهایی به Decoder داده می‌شود و ابعاد آن دوباره در هر لایه به شکل مناسب تنظیم می‌شود.

در این پیاده‌سازی، مقدار latent_dim برابر 128 قرار داده شده است. این مقدار مشخص می‌کند که چقدر از جزییات و پترن های پیچیده ورودی توسط مدل آموزش دیده شوند و هر چقدر که بیشتر باشد، این آموزش بیشتر می‌شود. اما باید به trade off بین reconstruction quality و generalization توجه کرد. هرچقدر latent_dim بیشتری داشته باشیم، مدل به کیفیت بهتری میتواند تصاویر ورودی را بازسازی کند اما تعمیم‌پذیری آن کمتر می‌شود و ممکن است دچار overfit شود. با توجه به اینکه دیتاست ورودی بزرگ و تصاویر مغزی هم نسبتا پیچیده هستند، ابتدا latent_dim های کمتر آزمایش شدند و این مقدار 128 بهتر از بقیه حالت‌ها نتیجه داد.

```
def vae_loss(recon_x, x, mu, logvar, loss_type="mse"):
    if loss_type == "mse":
        recon_loss = nn.MSELoss(reduction='sum')(recon_x, x)
    elif loss_type == "l1":
        recon_loss = nn.L1Loss(reduction='sum')(recon_x, x)
    kl_div = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon_loss + kl_div
```

تابع vae_loss ترکیبی از دو بخش Reconstruction Loss و KL Divergence است که برای آموزش VAE استفاده می‌شود.Reconstruction Loss با استفاده از MSE Loss یا L1 Loss (با توجه به ورودی)، تفاوت بین داده ورودی اصلی و داده Reconstructed را محاسبه می‌کند. این بخش مدل را مجبور می‌کند که داده‌های ورودی را به‌دقت بازسازی کند. در مقابل KL Divergence اختلاف بین توزیع Latent Space و یک توزیع نرمال استاندارد را اندازه‌گیری می‌کند. هدف از این بخش، منظم‌سازی Latent Space است تا توزیع به یک شکل ساده و قابل نمونه‌گیری نزدیک شود.

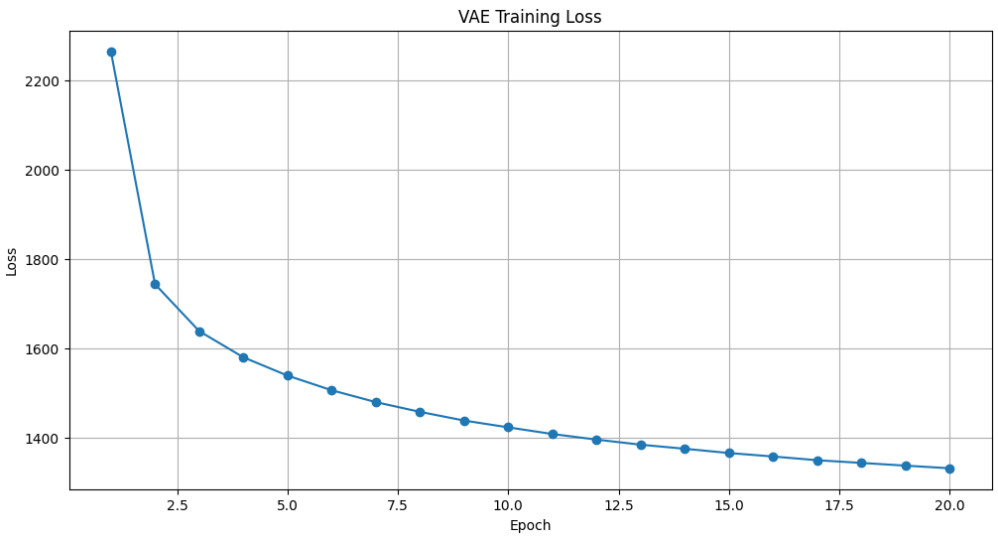
استفاده هم‌زمان از این دو بخش ضروری است. Reconstruction Loss به مدل کمک می‌کند تا داده‌ها را دقیقا مانند ورودی بازسازی کند، در حالی که KL Divergence فضای Latent را کنترل می‌کند تا بتوان داده‌های جدید را از آن تولید کرد. این ترکیب به VAE امکان می‌دهد که هم داده‌های واقعی را به‌خوبی بازسازی کند و هم داده‌های generated جدید با کیفیت بالا ایجاد کند.

آموزش روی دیتاست سالم IXI

با استفاده از هایپرپارامتر های زیر، مدل VAE را بر روی دیتای IXI آموزش می‌دهیم. خطای آموزش در شکل 1-3 آورده شده است.

جدول 1-2. هایپرپارامترهای استفاده شده در مدل **VAE**

Epochs	20
Batch Size	4
Optimizer	Adam
Learning Rate	0.001
Loss Function	MSE Loss + KL Divergence
Tumor Threshold	0.3



شکل 1-3. خطای آموزش مدل **VAE** در هر ایپاک

تست مختصر روی BraTS

در این قسمت برای تست مدل، تصاویری از BraTS استفاده شده است و slice میانی آنها به عنوان داده تست استفاده شده است. ابتدا تابع detect_tumor را تعریف می‌کنیم.

```
def detect_tumor(model, image, threshold=0.1):
    model.eval()
    recon_image, _, _ = model.predict(image.to(device))
    error = torch.abs(image - recon_image).squeeze().cpu().numpy()
    tumor_mask = (error > threshold).astype(np.uint8)
    return error, tumor_mask, recon_image.squeeze().cpu().numpy()
```

این تابع تصویر ورودی را به مدل می‌دهد و خروجی ساخته شده را دریافت میکند. سپس تفاوت خروجی با تصویر اصلی بررسی می‌شود (error) و هر قسمتی که تفاوت بیشتری از threshold داشته باشد، به عنوان تومور درنظر گرفته می‌شود.

```
def dice_score(predicted_mask, real_mask):
    predicted_mask = predicted_mask.flatten()
    real_mask_image = Image.fromarray(real_mask.astype(np.uint8))
    resized_mask = np.array(real_mask_image.resize((256, 256), Image.NEAREST))
    real_mask = (resized_mask > 0).astype(np.uint8).flatten()

    intersection = np.sum(predicted_mask * real_mask)
    dice = (2.0 * intersection) / (np.sum(predicted_mask) + np.sum(real_mask))
    return dice
```

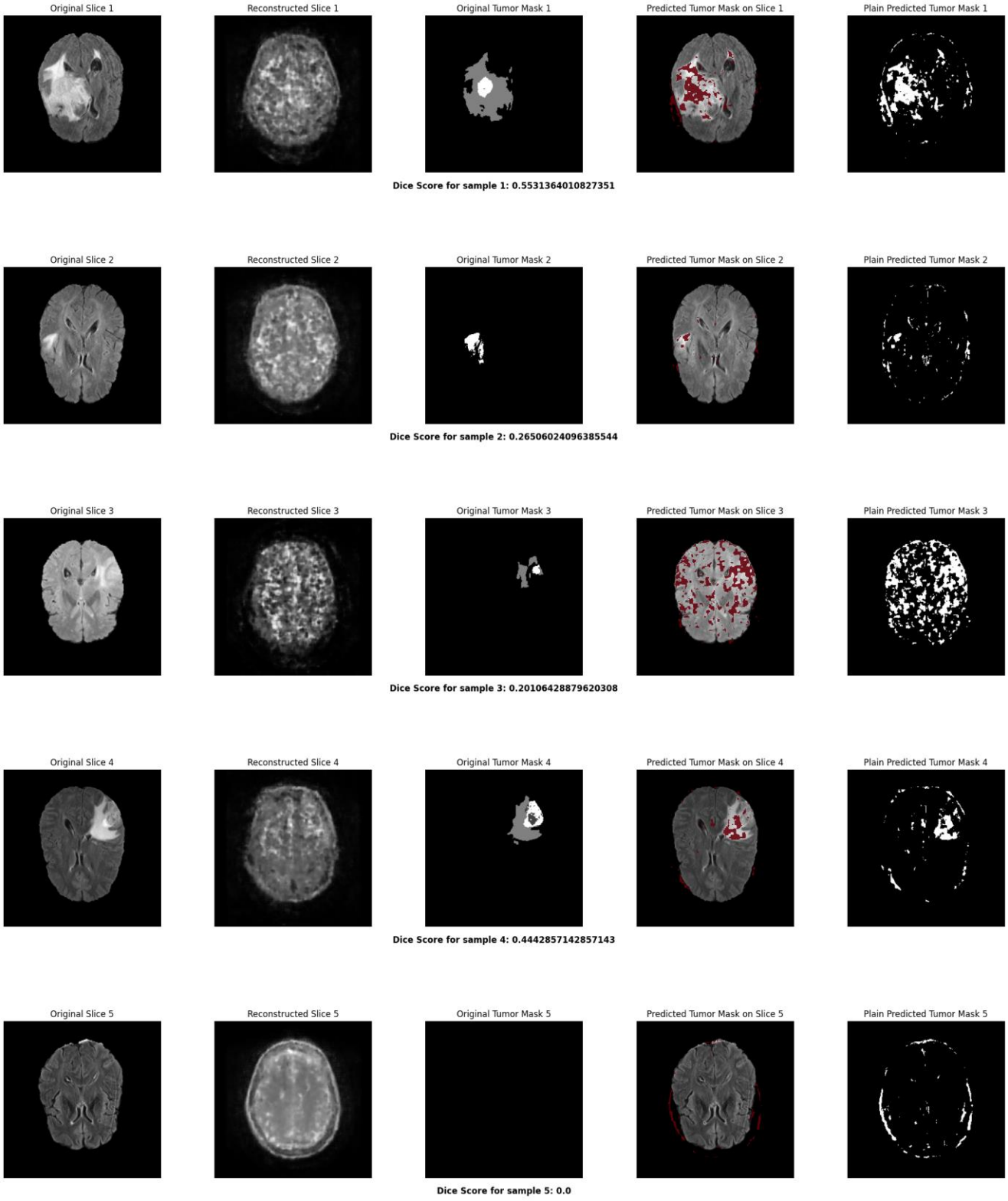
همچنین Dice Score را نیز تعریف می‌کنیم. Dice Score معیاری برای ارزیابی میزان هم‌پوشانی دو مجموعه است و اغلب برای مقایسه ماسک‌های پیش‌بینی‌شده و واقعی در بخش‌بندی تصاویر پزشکی استفاده می‌شود.در تصاویر پزشکی، Dice Score به دلیل حساسیت بالای آن به نواحی هم‌پوشانی، برای ارزیابی دقیق مدل‌های segmentation در کاربردهایی مانند تشخیص تومور استفاده می‌شود. این معیار تعدادی edge case دارد که با توجه به ماهیت و استفاده از آن، منطقی هستند:

1. اگر هم ماسک پیش‌بینی‌شده و هم واقعی کاملا خالی باشند (هیچ پیکسلی مثبت نباشد)، نتیجه به صورت نامشخص $\frac{0}{0}$ و به عنوان یک تطابق کامل (Dice = 1) خواهد بود. از نظر شهودی نیز اگر تصویر اصلی توموری نداشته باشد و مدل هم توموری پیشبینی نکند، یعنی مدل به خوبی عمل کرده است.

2. اگر یکی از ماسک‌ها خالی و دیگری حتی یک پیکسل پر باشد، Dice Score صفر خواهد بود، که نشان‌دهنده عدم تطابق کامل است. این نیز منطقی است از این نظر که اگر تصویر اصلی تومور نداشته باشد و مدل توموری تشخیص دهد، یک False Positive است. اگر هم تصویر اصلی تومور داشته باشه و مدل هیچ توموری تشخیص ندهد، یک False Negative است که هر دوی اینها در بحث پزشکی بسیار خطرناک هستند و معیار باید با 0 دادن به این ارزیابی، جلوی آنها را بگیرد.

3. برای ماسک‌های با مقادیر بسیار کوچک، Dice ممکن است به شدت متاثر شود و مقدار های خیلی بالا و یا خیلی پایین دهد و نیاز به پردازش خاص دارد.

ارزیابی مدل بر روی 100 داده BraTS انجام شده است که تعدادی از slice های دیتاست BraTS، به همراه تصویر بازسازی شده و تومور پیش‌بینی شده و Dice Score بدست آمده در شکل 1-4 آورده شده است.



شکل 1-4. نمونه‌هایی از ارزیابی مدل **VAE** بر روی **BraTS**

میانگین Dice Score در ارزیابی 100 دیتای BraTS: 0.254778624514882

۱-۳. پیاده‌سازی Tri-VAE

در این بخش به پیاده‌سازی مدل Tri-VAE می‌پردازیم. مدل Tri-VAE یا Triplet Variational Autoencoder ترکیبی از معماری VAE و ایده‌های Triplet Loss است که برای یادگیری بازنمایی‌های معنایی و تفکیک‌پذیر استفاده می‌شود. این مدل با دریافت سه ورودی شامل Anchor (نمونه اصلی)، Positive (نمونه‌ای مشابه و نزدیک به Anchor از نظر معنایی)، و Negative (نمونه‌ای متفاوت و دور از Anchor از نظر معنایی که در اینجا برای ایجاد آن نویز اضافه می‌شود)، تلاش می‌کند بازنمایی‌هایی در Latent Space ایجاد کند که بتوانند تفاوت‌ها و شباهت‌های معنایی بین داده‌ها را به‌خوبی منعکس کنند. تاثیر سه ورودی مختلف به این صورت است که وجود Anchor به عنوان نقطه مرجع، Positive برای تقویت نزدیکی معنایی، و Negative برای دور کردن بازنمایی‌های غیرمرتبط در فضای نهانی کمک می‌کند. با این روش، مدل می‌تواند داده‌هایی با معنای مشابه را نزدیک به هم نگه داشته و داده‌های غیرمرتبط را از هم دور کند.

```
class TriVAE(nn.Module):
    def __init__(self, latent_dim=16):
        super(TriVAE, self).__init__()
        # Shared encoder and decoder for Anchor, Positive, and Negative ->
    Based on the paper
        self.encoder_conv1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),  # 256x256 -
    > 128x128
            nn.ReLU(),
        )
        self.encoder_conv2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),  # 128x128
    -> 64x64
            nn.ReLU(),
        )
        self.encoder_conv3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),  # 64x64 -
    > 32x32
            nn.ReLU(),
        )

        self.flatten = nn.Flatten()
        self.fc_mu = nn.Linear(128 * 32 * 32, latent_dim)
        self.fc_logvar = nn.Linear(128 * 32 * 32, latent_dim)

        self.fc_dec = nn.Linear(latent_dim, 128 * 32 * 32)
        self.decoder_deconv3 = nn.Sequential(
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2,
    padding=1),  # 32x32 -> 64x64
```

```
            nn.ReLU(),
        )
        self.decoder_deconv2 = nn.Sequential(
            nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),  # 64x64 -> 128x128
            nn.ReLU(),
        )
        self.decoder_deconv1 = nn.Sequential(
            nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2, padding=1),  # 128x128 -> 256x256
            nn.Sigmoid(),
        )
        # Decoder for coarse reconstruction (32x32)
        self.coarse_decoder = nn.Sequential(
            nn.Conv2d(128, 64, kernel_size=3, padding=1),  # 32x32 -> 32x32
            nn.ReLU(),
            nn.Conv2d(64, 1, kernel_size=3, padding=1),    # 32x32 -> 32x32
            nn.Sigmoid(),
        )

        # Gated Cross Skip
        self.gcs_linear = nn.Linear(128, 64)
        self.gcs_conv = nn.Conv2d(64, 64, kernel_size=1)

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

    def encode(self, x):
        x1 = self.encoder_conv1(x)
        x2 = self.encoder_conv2(x1)
        x3 = self.encoder_conv3(x2)
        h_flat = self.flatten(x3)
        mu = self.fc_mu(h_flat)
        logvar = self.fc_logvar(h_flat)
        return mu, logvar, x3, x2

    def decode(self, z, skip_x2, skip_x3):
        h_dec = self.fc_dec(z).view(-1, 128, 32, 32)
        h_dec = self.decoder_deconv3(h_dec)

        # GCS integration
        gcs_features = torch.mean(skip_x3, dim=(2, 3))
        gcs_features = self.gcs_linear(gcs_features)
        gcs_features = self.gcs_conv(gcs_features.view(-1, 64, 1, 1))
        h_dec = h_dec + gcs_features
```

```
        h_dec = self.decoder_deconv2(h_dec)
        x_recon_full = self.decoder_deconv1(h_dec)

        x_recon_coarse = self.coarse_decoder(skip_x3)

        return x_recon_coarse, x_recon_full

    def forward(self, xa, xp, xn):
        mu_a, logvar_a, skip_a3, skip_a2 = self.encode(xa)
        mu_p, logvar_p, skip_p3, skip_p2 = self.encode(xp)
        mu_n, logvar_n, skip_n3, skip_n2 = self.encode(xn)

        z_a = self.reparameterize(mu_a, logvar_a)
        z_p = self.reparameterize(mu_p, logvar_p)
        z_n = self.reparameterize(mu_n, logvar_n)


        x_recon_a_coarse, x_recon_a_full = self.decode(z_a, skip_a2, skip_a3)
        x_recon_p_coarse, x_recon_p_full = self.decode(z_p, skip_p2, skip_p3)
        x_recon_n_coarse, x_recon_n_full = self.decode(z_n, skip_n2, skip_n3)

        return (
            x_recon_a_coarse, x_recon_a_full,
            x_recon_p_coarse, x_recon_p_full,
            x_recon_n_coarse, x_recon_n_full,
            mu_a, logvar_a, mu_p, logvar_p, mu_n, logvar_n
        )

    def predict(self, x):
        mu, logvar, skip_x3, skip_x2 = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon_coarse, x_recon_full = self.decode(z, skip_x2, skip_x3)

        return x_recon_full, mu, logvar
```

طبق گفته‌ی مقاله، مدل Tri-VAE در هر سه بخش خود وزن‌های مشترکی دارد. به همین دلیل برای پیاده‌سازی ما، کلاس Tri-VAE به صورت مشترک برای سه ورودی مختلف، یک Encoder و Decoder مشترک دارد که به صورت همزمان بر روی 3 تصویر آموزش می‌بینند. این به این معناست که مدل ابتدا هر سه ورودی Anchor، Positive و Negative را با استفاده از Encoder به Latent Space می‌برد و سپس هر کدام از آن‌ها را با استفاده از Decoder بازسازی می‌کند. با این روش وزن‌های مدل در Encoder و Decoder به‌طور مشترک برای هر سه ورودی بهینه‌سازی می‌شوند، که به مدل اجازه می‌دهد تا روابط معنایی و تفاوت‌ها را به طور یکپارچه در بین نمونه‌ها یاد بگیرد. به عبارت دیگر، مدل به دنبال کاهش فاصله معنایی بین Anchor و Positive و افزایش فاصله با Negative در Latent Space است. در نهایت مدل حین آموزش، طبق مدل مقاله برای هر تصویر 2

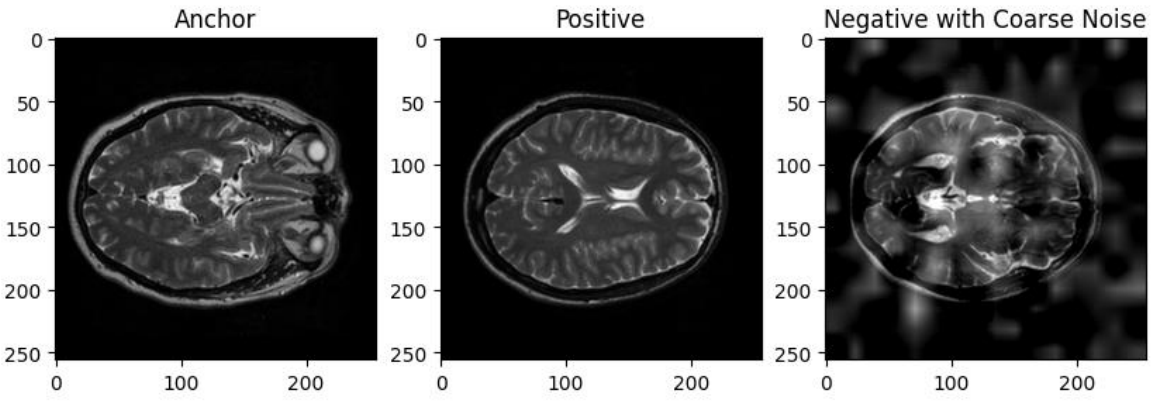
خروجی می‌دهد. یکی نسخه Full Scale Reconstruction و یکی هم نسخه Coarse Scale Reconstruction با سایز 32 * 32 که در آینده برای محاسبه L1 Coarse استفاده می‌شود.

توجه کنید که سه تصویر ورودی در هر Epoch به طور رندوم انتخاب می‌شوند.

برای ایجاد نویز برای تصویر Negative، از دو روش Coarse و Simplex استفاده شد.

```
def add_coarse_noise(self, image):
    image_np = np.array(image, dtype=np.float32) / 255.0

    noise_resolution = (16, 16)
    coarse_noise = np.random.normal(loc=0.0, scale=0.2, size=noise_resolution)
    upsampled_noise = np.array(Image.fromarray(coarse_noise).resize((256,
    256), Image.BILINEAR))
    noisy_image_np = image_np + upsampled_noise
    noisy_image_np = np.clip(noisy_image_np, 0, 1)
    noisy_image = Image.fromarray((noisy_image_np * 255).astype(np.uint8))
    return noisy_image
```

در روش Coarse Noise، با استفاده از ماتریسی کوچکی نسبت به تصویر اصلی (در اینجا 16*16) از مقادیر تصادفی با توزیع نرمال ایجاد می‌شود که سپس به ابعاد تصویر بزرگ‌نمایی می‌شود تا الگوی نویز ساده‌ای با دانه‌های درشت روی تصویر اعمال گردد. این نویز برای شبیه‌سازی اختلالات کم‌جزئیات مناسب است. نمونه‌ای از ایجاد آن برای تصویر Negative به همراه Anchor و Positive های انتخاب شده در شکل 1-5 آورده شده است.



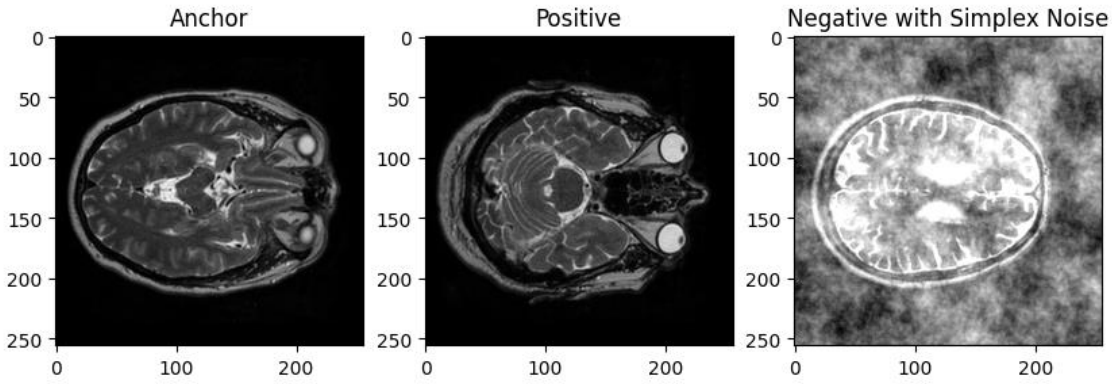
شکل 1-5. تصویر نمونه از **Anchor**، **Positive** و **Negative** به همراه نویز **Coarse**

```
def add_simplex_noise(self, image):
    image_np = np.array(image, dtype=np.float32) / 255.0
    height, width = image_np.shape

    start_freq = 2 ** -6  # Starting frequency
    octaves = 6           # Number of octaves
    persistence = 0.8     # decay rate
```

```
    lacunarity = 2        # Lacunarity (frequency multiplier)
    noise = np.zeros((height, width), dtype=np.float32)
    for i in range(height):
        for j in range(width):
            value = 0
            freq = start_freq
            amp = 1.0
            for _ in range(octaves):
                value += self.simplex.noise2(i * freq, j * freq) * amp
                freq *= lacunarity
                amp *= persistence
            noise[i, j] = value
    noise = (noise - np.min(noise)) / (np.max(noise) - np.min(noise))
    noisy_image_np = image_np + noise
    noisy_image_np = np.clip(noisy_image_np, 0, 1)
    noisy_image = Image.fromarray((noisy_image_np * 255).astype(np.uint8))

    return noisy_image
```

در مقابل، نویز Simplex با استفاده از الگوریتمی پیچیده‌تر و با ترکیب چندین Octave ساخته می‌شود. این الگوریتم از مقیاس‌های مختلف نویز برای ایجاد الگوهای طبیعی‌تر و پیوسته‌تر استفاده می‌کند. پارامترهایی نظیر فرکانس اولیه، تعداد اکتاوها، نرخ Persistance، و Lacunarity برای کنترل جزئیات و ظاهر کلی نویز استفاده می‌شوند. این نوع نویز اغلب در گرافیک کامپیوتری و شبیه‌سازی طبیعت (مانند ابرها یا زمین) کاربرد دارد. ایجاد این نویز به دلیل الگوریتم پیچیده‌تر، بسیار زمانبرتر است و در آزمایش، تا 20 برابر بیشتر از Coarse Noise زمان برای ایجاد شدن دارد. نمونه‌ای از ایجاد آن برای تصویر Negative به همراه Anchor و Positive های انتخاب شده در شکل 1-6 آورده شده است.

هر دو نویز به صورت افزایشی به تصویر اصلی اضافه شده و سپس مقادیر حاصل، نرمال‌سازی و در محدوده 0 تا 1 محدود می‌شوند تا تغییرات در تصویر طبیعی‌تر و محدود باقی بماند.



شکل 1-6. تصویر نمونه از **Anchor**، **Positive** و **Negative** به همراه نویز **Simplex**

GCS

(کد های این بخش در همان کلاس TriVAE که در بالا آورده شده است، قابل مشاهده می‌باشد.)

مکانیزم Gated Cross Skip (GCS) در این پیاده‌سازی نقش مهمی در تقویت توانایی Decoder برای بازیابی جزئیات spatial مناطق سالم مغز برای reconstruction (جلوگیری از از دست رفتن اطلاعات حین downsampling) و سرکوب آنومالی‌های ایجاد شده در تصویر ایفا می‌کند. GCS به‌صورت انتخابی اطلاعات spatial و کانالی را از لایه‌های Encoder به Decoder منتقل می‌کند. ابتدا feature map انکودر از نظر spatial به کمک global pooling کاهش داده می‌شود تا اطلاعات کلی و مهم حفظ شود و نویز کاهش یابد. سپس این ویژگی‌ها از طریق یک linear layer به ابعاد فشرده‌تری نگاشت می‌شوند که باعث کاهش پیچیدگی محاسباتی و ذخیره‌سازی کارآمدتر می‌شود. پس از این مرحله، featureهای کاهش‌یافته از طریق یک لایه Convolution 1*1 به فضای featureهای Decoder منتقل می‌شوند.

پیاده‌سازی فعلی GCS یک معماری Residual است، زیرا featureهای Encoder پس از transformation، به صورت جمع مستقیم به featureهای Decoder اضافه می‌شوند. این روش مطابق با Residual Learning است که اطلاعات اضافی را از طریق element-wise addition منتقل می‌کند.

Losses

```
def tri_vae_loss(recon_a, xa, xa_coarse, recon_p, xp, xp_coarse, recon_n, xn,
    xn_coarse, mu_a, logvar_a, mu_p, logvar_p, mu_n, logvar_n):
    # Downsample images for L1 (Coarse)  32*32 -> 16*16
    coarse_size = (16, 16)
    xa_coarse = resize(xa_coarse, coarse_size, antialias=True)
    xp_coarse = resize(xp_coarse, coarse_size, antialias=True)
    xn_coarse = resize(xn_coarse, coarse_size, antialias=True)
    recon_a_coarse = resize(recon_a, coarse_size, antialias=True)
    recon_p_coarse = resize(recon_p, coarse_size, antialias=True)
    recon_n_coarse = resize(recon_n, coarse_size, antialias=True)

    # L1 (Coarse) for Anchor, Positive, Negative
    recon_loss_a_coarse = nn.L1Loss()(recon_a_coarse, xa_coarse)
    recon_loss_p_coarse = nn.L1Loss()(recon_p_coarse, xp_coarse)
    recon_loss_n_coarse = nn.L1Loss()(recon_n_coarse, xn_coarse)
    recon_loss_coarse = recon_loss_a_coarse + recon_loss_p_coarse +
    recon_loss_n_coarse

    # L1 (Full) for Negative
    recon_loss_n_full = nn.L1Loss()(recon_n, xn)
```

```
    # KL Divergence Loss (Anchor and Positive)
    kl_div_a = -0.5 * torch.sum(1 + logvar_a - mu_a.pow(2) - logvar_a.exp()) /
    xa.size(0)
    kl_div_p = -0.5 * torch.sum(1 + logvar_p - mu_p.pow(2) - logvar_p.exp()) /
    xp.size(0)
    kl_div_loss = kl_div_a + kl_div_p

    # Triplet Loss
    margin = 1.0
    d_ap = torch.sum((mu_a - mu_p).pow(2), dim=1)
    d_an = torch.sum((mu_a - mu_n).pow(2), dim=1)
    triplet_loss = torch.mean(F.relu(d_ap - d_an + margin))

    # SSIM Loss (for Negative full reconstruction)
    ssim_loss_fn = SSIM(data_range=1.0, size_average=True, channel=1)
    ssim_loss = 1 - ssim_loss_fn(recon_n, xn)

    total_loss = recon_loss_coarse + recon_loss_n_full + kl_div_loss
    +  triplet_loss + ssim_loss

    return total_loss
```

مولفه‌های Loss Function

1. Coarse Reconstruction Loss (L1 Loss برای همه تصاویر)

L1 Loss (Coarse) تضمین می‌کند که مدل می‌تواند ویژگی‌های ساختاری کلی را به‌درستی یاد بگیرد. کوچک‌کردن تصاویر به اندازه 16 * 16 باعث می‌شود که مدل بر روی جزئیات سطح پایین و سازگاری کلی فضایی تمرکز کند، که برای شناسایی تومورهایی با اندازه و شکل متغیر اهمیت دارد.

- **تصاویر استفاده‌شده**
  - Anchor (coarse reconstructed در مقابل تصویر ground truth کوچک‌شده)
  - Positive (coarse reconstructed در مقابل تصویر ground truth کوچک‌شده)
  - Negative (coarse reconstructed در مقابل تصویر ground truth کوچک‌شده)
- **نقش در سگمنت‌کردن تومور**

این loss به‌عنوان یک regularizer عمل کرده و به مدل کمک می‌کند تا انسجام ساختاری و دقت Reconsturction را حفظ کند که برای سگمنت‌کردن دقیق تومور ضروری است.

2. Full Reconstruction Loss (L1 Loss روی تصاویر Negative)

L1 Loss (Full) بر دقت در سطح پیکسل برای تصاویر Negative بازسازی‌شده (که دارای نویز هستند) تاکید دارد. این loss تضمین می‌کند که مدل بتواند به‌طور موثر نویز را حذف کرده و تصاویر باکیفیت را از ورودی‌های نویزی بازسازی کند

- **تصاویر استفاده‌شده**
  - Negative (full reconstructed در مقابل تصویر ground truth کامل)
- **نقش در سگمنت‌کردن تومور**

بازسازی دقیق تصاویر Negative به مدل کمک می‌کند تا تفاوت نویز و ویژگی‌های معنادار را شناسایی کند و توانایی آن را برای پردازش داده‌های نویزی یا ناقص که در تصویربرداری پزشکی معمول هستند، بهبود می‌بخشد.

3. KL Divergence Loss (Latent Space Regularization)

KL Divergence تضمین می‌کند که Latent Space با یک توزیع نرمال همسو باشد، که منجر به regularization بهتر و interpolation می‌شود. با اعمال شباهت در توزیع‌های latent تصاویر Anchor و Positive، مدل یاد می‌گیرد که featureهای مشابه را به هم نزدیک‌تر encode کند.

- **تصاویر استفاده‌شده**
  - Anchor و Positive
- **نقش در سگمنت‌کردن تومور**

این loss به مدل کمک می‌کند تا به‌خوبی generalize کرده و ویژگی‌های مشترک ناحیه‌های توموری را در تصاویر Anchor و Positive به‌طور موثر یاد بگیرد.

4. Triplet Loss

Triplet Loss یک margin بین نمایش‌های latent زوج‌های Anchor-Positive و Anchor-Negative اعمال می‌کند. این loss تضمین می‌کند که تصاویر مشابه (Anchor و Positive) به هم نزدیک‌تر encode شوند، درحالی‌که تصاویر نامشابه (Anchor و Negative) از یکدیگر جدا شوند.

- **تصاویر استفاده‌شده**
  - Anchor، Positive و Negative (latent space embeddings)
- **نقش در سگمنت‌کردن تومور**

این loss برای یادگیری ویژگی‌های متمایز حیاتی است و مدل را قادر می‌سازد تا بین نواحی توموری و غیرتوموری تمایز قائل شود.

5. SSIM Loss (Structural Similarity) بر روی تصاویر Negative

SSIM Loss تشابه ساختاری را اندازه‌گیری کرده و به کیفیت ادراکی بیشتر از دقت پیکسلی تاکید دارد. این موضوع به‌ویژه برای تصاویر پزشکی مهم است. چون که حفظ یکپارچگی ساختاری ویژگی‌هایی مانند مرزهای تومور اهمیت زیادی دارد. چون این ساختارها در تصاویر نویزی بیشتر دچار بهم ریخته شدن می‌شوند، بر روی تصاویر Negative نویزی این Loss اعمال می‌شود.

- **تصاویر استفاده‌شده**
  - Negative (full reconstructed در مقابل تصویر ground truth کامل)
- **نقش در سگمنت‌کردن تومور**

با تمرکز بر تشابه ساختاری، این loss تضمین می‌کند که مدل بتواند ویژگی‌های حیاتی تومور را حتی در موارد چالش‌برانگیز شامل ورودی‌های نویزی یا کاهش‌یافته حفظ کند.

اهمیت و تعامل مولفه‌های Loss Function

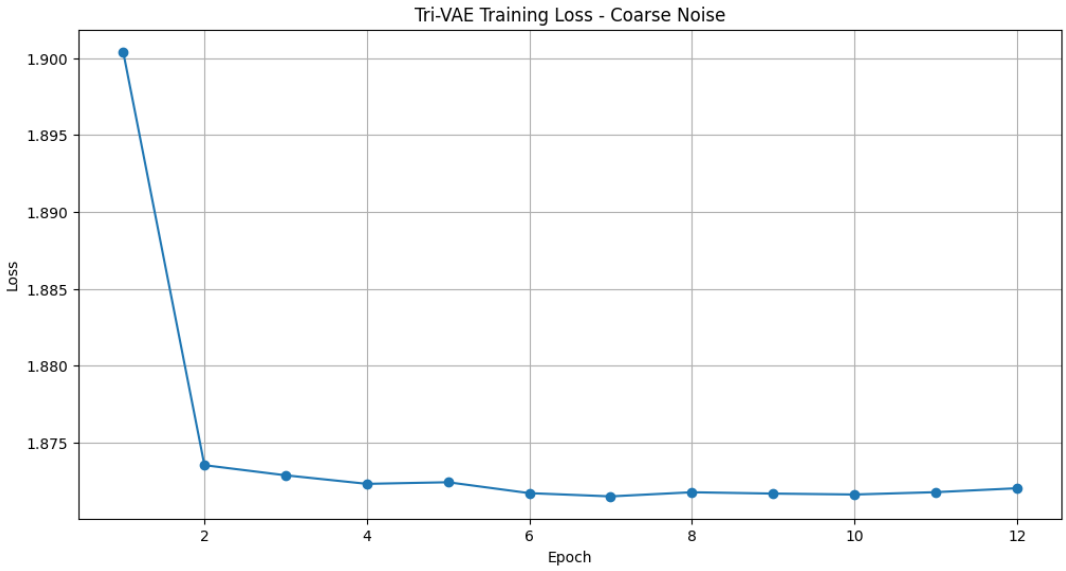
- **Reconstruction Losses (مثل L1 و SSIM)**: بر بازسازی دقیق و منسجم تصاویر در چندین رزولوشن تمرکز دارند.
- **KL Divergence و Triplet Loss**: ویژگی‌های robust را در Latent Space یاد می‌گیرند، generalization و تمایز بین ورودی‌های مشابه و نامشابه را تقویت می‌کنند.
- **Full - Coarse Level Losses**: جزئیات مختلف را در مقیاس‌های متفاوت هدف قرار می‌دهند و به مدل امکان درک هم ساختار کلی و هم اطلاعات جزئی را می‌دهند.

آموزش مدل

در نهایت با هایپرپارامترهای جدول 1-2، به آموزش مدل می‌پردازیم. البته توجه شود که مدل قرار بود در 20 Epoch اجرا شود اما چون بعد از 7 Epoch تا چند Epoch تغییری در loss دیده نمی‌شد، آموزش در 12 Epoch متوقف گردید. نمودار loss با نویز Coarse در شکل 1-7 آورده شده است. (آموزش با نویز Simplex هم قرار بود که انجام شود اما به دلیل اینکه ایجاد تصویر با این نویز 20 برابر بیشتر از Coarse زمان میبرد و تعداد تصاویر برای آموزش زیاد بود و خود نسخه Coarse هم هر Epoch حدودا 10 دقیقه طول می‌کشید، عملا امکان آموزش با نویز Simplex با توجه به محدودیت‌های سخت‌افزاری میسر نبود.)

شکل 1-2. هایپرپارامتر های مدل **Tri-VAE**

Epochs	12
Batch Size	4
Optimizer	Adam
Learning Rate	0.001
Loss Function	Reconstruction L1 Losses (Full and Coarse) + KL Divergence + Triplet Loss + SSIM Loss
Tumor Threshold	0.3
Noise Type	Coarse Noise, Simplex Noise



شکل 1-7. خطای آموزش مدل **Tri-VAE** با نویز **Coarse** در هر ایپاک

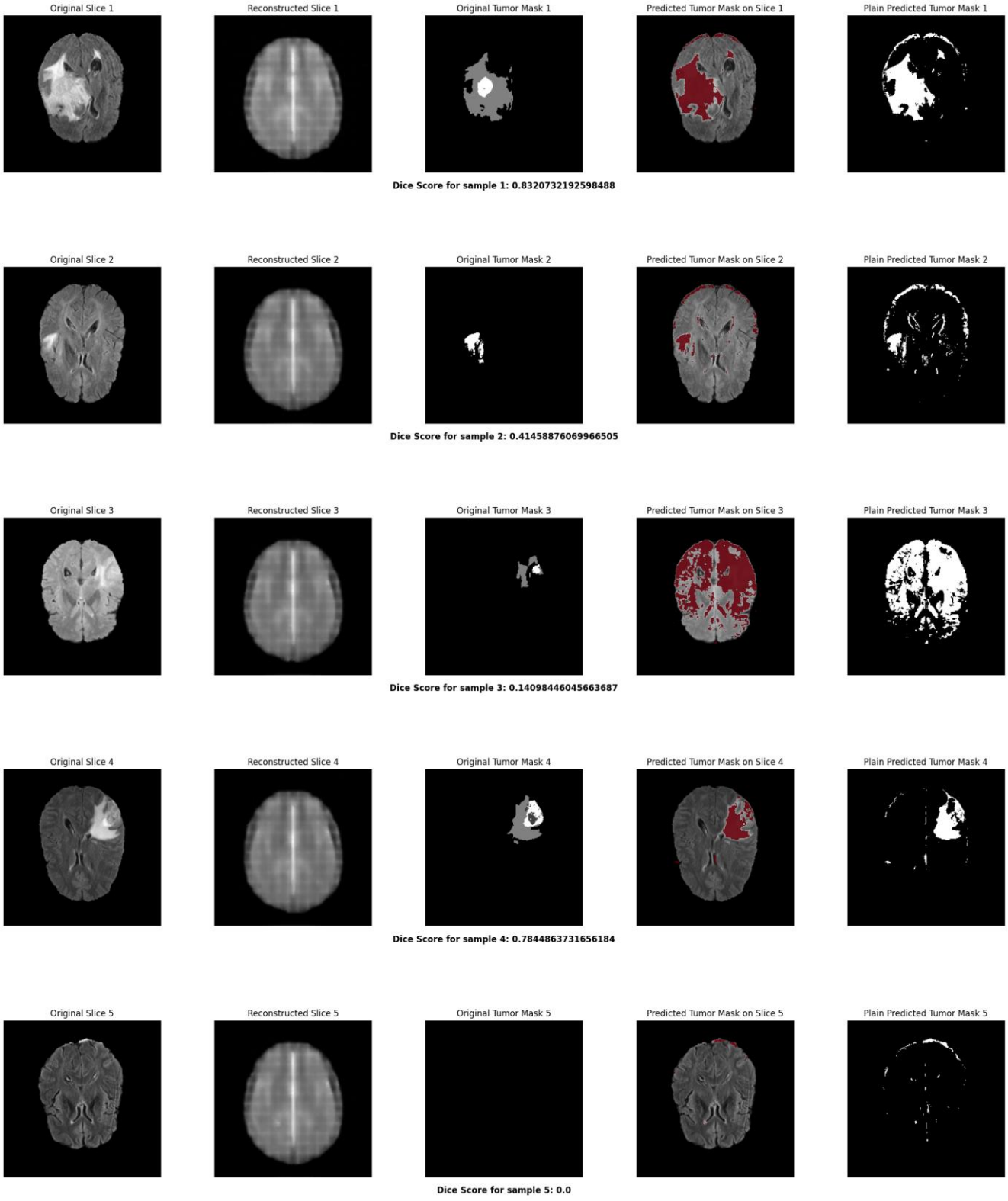
این مورد هم در نظر گرفته شود که order خطای مدل VAE و Tri-VAE با یکدیگر بسیار متفاوت است که میتوان دلیل آنرا متفاوت بودن توابع loss آنها بیان کرد.

۱-۴: ارزیابی در دیتاست BraTS (دو بعدی)

```
model_trivae_coarse = TriVAE(latent_dim=128)
model_trivae_coarse.load_state_dict(torch.load("tri_vae_coarse_12epoch.pth"))
model_trivae_coarse.to(device)
model_trivae_coarse.eval()

threshold = 0.3
dices_trivae_coarse = []
plt.figure(figsize=(20, 25))
for idx, (image, real_mask) in enumerate(test_loader_vae):
    if idx > 100:
        break
    real_mask = real_mask.squeeze().cpu().numpy()

    error, tumor_mask, recon_image = detect_tumor(model_trivae_coarse, image,
threshold)
    dice = dice_score(tumor_mask, real_mask)
    dices_trivae_coarse.append(dice)

    if idx < 5:
        visualize_detection_with_mask(
            original_slice=image.squeeze().cpu().numpy(),
            error=error,
            tumor_mask=tumor_mask,
            recon_image=recon_image,
            real_mask=real_mask,
            dice_score=dice,
            idx=idx
        )

mean_dice = np.mean(dices_trivae_coarse)
print(f"Mean Dice Score for 100 test samples: {mean_dice}")
```

ابتدا وزن‌های آموزش دیده در بخش قبل را لود می‌کنیم. سپس به ارزیابی 100 دیتا از BraTS می‌پردازیم و میانگین Dice Score آن‌ها را نیز گزارش می‌کنیم. برخی از نمونه‌های ارزیابی شده به همراه ماسک پیش‌بینی شده و ground truth به همراه Dice Score آن ارزیابی در شکل 1-8 آورده شده است.

توجه کنید که برای انجام prediction در حالت testing، چون مدل صرفا دارای یک Encoder و Decoder است، از همان ساختار قبلی میتوان استفاده کرد و صرفا یک خروجی Fully Reconstructed از مدل دریافت کرد. درباره خطای بازسازی نیز در بخش ارزیابی برای مدل VAE توضیح داده شد که متغیر Threshold = 0.3 به چه معنا است و از این مقدار چه استفاده‌ای می‌شود.



شکل 1-8. نمونه‌هایی از ارزیابی مدل **Tri-VAE** بر روی دیتاست **BraTS**

میانگین Dice Score در ارزیابی 100 دیتای BraTS: 0.4045508069056412

مشاهده می‌شود که مدل Tri-VAE بسیار بهتر از VAE عمل کرده است، با اینکه Epochهای کمتری آموزش دیده و به عدد میانگین Dice Score 0.404 رسیده است که با توجه به اینکه Preprocess یا Postprocess ای بر روی دیتا انجام نشد، مقدار مناسبی است. این پیشرفت در نتیجه نسبت به مدل VAE که به Dice Score 0.25 رسید را میتوان به ساختار سه بخشی Tri-VAE ارتباط داد و همچنین مدل با آموزش بر روی داده‌ی نویزی، توانسته به قدرت بیشتری برسد و همچنین تعمیم‌پذیری بهتری داشته باشد.

با بررسی نمونه‌های آورده شده در شکل 1-9، میتوان دید که مدل مقداری نسبتا بیش از حد بر روی رنگ تصویر حساس شده است که از آنجایی که رنگ تومور در تصاویر، سفید است، این مورد توجیه می‌شود و باید به گونه‌ای حل شود. مثلا در نمونه سوم که slice مغز رنگی سفیدتر نسبت به بقیه دارد، حجم زیادی از بافت‌های سالم مغز به عنوان تومور در نظر گرفته شده و Dice Score پایینی (0.14) دریافت شده است. این حساسیت اضافه مدل بر روی رنگ سفید میتوان از قدرت بیشتر مدل Tri-VAE باشد، چون مدل VAE در پیش‌بینی این نمونه Dice Score بهتری (0.20) دریافت کرده است.

همچین یکی از مشکلاتی که هم در مدل VAE و هم در Tri-VAE دیده می‌شود، این است که لبه‌های slice مغز و جمجمه را به عنوان تومور در نظر میگیرد و این مورد تقریبا در تمامی موارد ارزیابی نمایش داده شده، مشاهده می‌شود. احتمالا نیاز است که با انجام Preprocess هایی تفاوت رنگی شدید بین لبه مغز و رنگ سیاه پس‌زمنیه را handle کرد، و یا با انجام Postprocess هایی، مدل را نسبت به انجام این اشتباه تکراری، robust کرد چون تعداد پیکسل‌های لبه مغز کم نیستند و هم بسیار باعث کم شدن Dice Score می‌شوند و همچنین اگر آنها را به طور کلی نادیده بگیریم، ممکن است True Positive هارا هم از دست بدهیم (یعنی نمونه هایی که واقعا در لبه مغز تومور وجود دارد) که این مورد بسیار خطرناک است.

یکی از مشکلاتی این مدل نسبت به VAE دارد، این است که تصویر Reconstructed شده برای همه نمونه‌ها یکسان است و اصلا شبیه تصویر ورودی نیست. برای حل این مشکل بررسی‌های متعددی انجام شد (مثل تغییر loss function یا دادن ضریب به هر خطا) و که آیا مشکل نمایش است یا خروجی‌ها مشکل دارند یا چیز دیگری دلیل این است، اما نتیجه‌ای حاصل نشد. این مورد کمی هم عجیب است و احتمال زیاد مشکلی در کد نمایش نمونه، یا استفاده از خروجی مدل است و ارتباطی با خود پردازش و آموزش مدل ندارد چون با بررسی تومور های پیش‌بینی شده و همچنین میانگین Dice Score بدست آمده، مدل به خوبی آموزش دیده است و تصویر تومور را تشخیص داده است. در نتیجه به احتمال زیاد صرفا به دلیل اشتباه در دسترسی به تصویر Reconstructed شده، به این شکل نمایش داده شده است.

(البته مواردی که در صورت پروژه خواسته شده شامل تصویر ورودی و ناحیه پیش‌بینی شده و ماسک واقعی است که به درستی نشان داده شده اند و قابل توجیه و درست می‌باشند.)

پرسش ۲ – AdvGAN

۲-1. آشنایی با حملات خصمانه و معماری AdvGAN

روش های دیگر تولید نمونه های تخاصمی به مانند FGSM و PGD را توضیح دهید و بیان بدارید مزیت یا مزیتهای مدلی به مانند AdvGANنسبت به روشهای دیگر چیست؟

- FGSM (Fast Gradient Sign Method) از گرادیان تابع هزینه برای ایجاد اختلال استفاده می‌کند. این اختلال به‌صورت خطی و بر اساس علامت گرادیان محاسبه می‌شود .مزیت اصلی FGSM در سادگی و سرعت بالای آن است. این روش به دلیل محاسبه سریع اختلال، برای تحلیل‌های اولیه بسیار مفید است. بااین‌حال، یکی از محدودیت‌های آن، کیفیت بصری پایین‌تر نمونه‌های تولیدشده است، که ممکن است اختلالات به‌راحتی توسط انسان یا مدل‌های دفاعی شناسایی شوند.
- PGD (Projected Gradient Descent) نسخه بهبودیافته FGSM است که از تکرارهای متعدد برای بهینه‌سازی اختلال استفاده می‌کند. در هر گام، نمونه به فضای مجاز محدود می‌شود تا از تجاوز اختلال از یک محدوده مشخص جلوگیری شود. این روش دقت بالاتری در موفقیت حملات دارد، اما به دلیل تعداد زیاد تکرارها، زمان بیشتری برای تولید نمونه‌ها نیاز دارد. بنابراین، PGD در مقابل FGSM مقاوم‌تر است اما از نظر محاسباتی سنگین‌تر است.
- در مقابل این روش‌ها، AdvGAN از شبکه‌های مولد تخاصمی (GAN) بهره می‌گیرد تا نمونه‌های تخاصمی را با کیفیت بالاتری تولید کند. در این روش، یک شبکه Generator آموزش داده می‌شود که می‌تواند بدون نیاز به محاسبات گرادیانی برای هر تصویر، به‌صورت مستقیم اختلال‌های لازم را تولید کند. این ویژگی، تولید نمونه‌های تخاصمی را سریع‌تر از روش‌هایی مانند FGSM و PGD می‌کند. علاوه بر سرعت، کیفیت بصری بالاتر نمونه‌های تولیدشده یکی دیگر از مزایای مهم AdvGAN است، زیرا شبکه متمایزگر (Discriminator) تضمین می‌کند که اختلالات تولیدشده تا حد امکان شبیه داده‌های اصلی باشند. علاوه بر این، AdvGAN نه‌تنها برای حملات

white-box، بلکه برای سناریوهای black-box و semi-white-box نیز بسیار موثر است. برخلاف FGSM و PGD که باید برای هر تصویر اختلال جدیدی محاسبه کنند، AdvGAN می‌تواند به‌صورت مستقیم و با سرعت بالا اختلالات را تولید کند و نرخ موفقیت بالاتری در حملات ارائه دهد.

تفاوتهای کلیدی بین AdvGANو یک GANساده را با تمرکز بر موارد زیر توضیح دهید.

- **چگونه AdvGANاز گرادیانها یا خروجیهای مدل هدف در زمان آموزش استفاده می‌کند؟**

در یک GAN ساده، شبکه مولد با هدف تولید داده‌هایی مشابه داده‌های واقعی آموزش می‌بیند و این فرآیند به صورت مستقل از هر مدل هدف یا کلاس‌بندی‌کننده انجام می‌شود. مولد صرفاً تلاش می‌کند تا داده‌هایی تولید کند که تفکیک‌گر را فریب داده و واقعی به نظر برسند. در این حالت، گرادیان‌ها تنها از طریق تفکیک‌گر به مولد بازگشت داده می‌شوند و مولد از این گرادیان‌ها برای بهبود عملکرد خود استفاده می‌کند.

در مقابل، AdvGAN به طور مستقیم با یک مدل هدف (Target Model) در ارتباط است. مولد در این معماری، علاوه بر فریب تفکیک‌گر، باید نویزهایی تولید کند که مدل هدف را دچار خطا کند. برای این منظور، از گرادیان‌های تابع هزینه مربوط به مدل هدف استفاده می‌شود. این گرادیان‌ها به مولد کمک می‌کنند تا یاد بگیرد چگونه نویزهایی ایجاد کند که بتوانند خروجی مدل هدف را تغییر دهند. این فرآیند باعث می‌شود که مولد بتواند به صورت هدفمند نمونه‌های متخاصمی تولید کند که علاوه بر شباهت به داده‌های اصلی، مدل هدف را نیز گمراه کنند. این وابستگی به مدل هدف، تفاوت کلیدی بین AdvGAN و GAN ساده است.

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

در یک GAN ساده، مولد معمولاً از بردار نویز تصادفی به عنوان ورودی استفاده می‌کند و داده‌هایی کاملاً جدید تولید می‌کند که شبیه به داده‌های واقعی باشند. هدف اصلی، تولید داده‌های واقعی‌تر برای فریب تفکیک‌گر است و این مدل هیچ‌گونه تضمینی برای گمراه کردن یا اثرگذاری روی یک مدل هدف ندارد. تابع هزینه در GAN ساده تنها شامل l_{GAN} است که اختلاف بین داده‌های واقعی و داده‌های تولید شده را اندازه‌گیری می‌کند.

در AdvGAN، هدف تولید نمونه‌های متخاصم است که ضمن گمراه کردن مدل هدف، از نظر بصری شبیه به داده‌های اصلی باقی بمانند. در این معماری، مولد به جای تولید داده‌های کاملاً جدید، نویز G(x) تولید می‌کند که به تصویر اصلی x اضافه شده و تصویر متخاصم x+G(x) را می‌سازد. برای حفظ توازن میان وفاداری بصری و قابلیت حمله، ترکیبی از سه تابع هزینه به کار می‌رود که در سوال بعد به طور کامل توضیح داده خواهند شد. این طراحی چندهدفه، توانایی AdvGANرا در ایجاد نمونه‌های متخاصمی که هم شباهت بصری بالایی دارند و هم توانایی گمراه کردن مدل هدف را حفظ می‌کنند، به خوبی نشان داده می‌شود.

سه تابع 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}** : این تابع هزینه از مفهوم شبکه‌های GAN گرفته شده است. هدف این تابع آن است که داده‌های تولیدشده توسط Generator شباهت بالایی به داده‌های واقعی داشته باشند، به طوری که تفکیک بین آن‌ها برای Discriminator دشوار باشد. رابطه ریاضی این تابع به صورت زیر است:

$$\mathcal{L}_{\text{GAN}} = \mathbb{E}_x \log \mathcal{D}(x) + \mathbb{E}_x \log(1 - \mathcal{D}(x + \mathcal{G}(x)))$$

  - **Discriminator**: تمایل به بیشینه‌سازی این تابع دارد تا داده‌های واقعی و تولیدشده را از هم تفکیک کند.

- **Generator**: تمایل به کمینه‌سازی این تابع دارد تا داده‌های تولیدشده به داده‌های واقعی شباهت بیشتری پیدا کنند.

- **تابع هزینه L_{adv}^{f}**: هدف این تابع هزینه این است که نمونه‌های متخاصم تولیدشده توسط تولیدکننده مدل هدف را گمراه کنند. در حملات Targeted، این تابع مدل را به طبقه‌بندی نمونه به یک کلاس خاص هدایت می‌کند. رابطه ریاضی این تابع به صورت زیر است:

$$\mathcal{L}_{\text{adv}}^{f} = \mathbb{E}_x \ell_f(x + \mathcal{G}(x), t),$$

- **تابع هزینه L_{hinge}** : این تابع برای محدود کردن مقدار اختلال ایجادشده استفاده می‌شود تا نمونه‌های متخاصم به داده‌های اصلی بسیار نزدیک باشند و تغییرات قابل مشاهده‌ای در آن‌ها ایجاد نشود. رابطه ریاضی این تابع به صورت زیر است:

$$\mathcal{L}_{\text{hinge}} = \mathbb{E}_x \max(0, \|\mathcal{G}(x)\|_2 - c),$$

که در آن c یک متغیر حاوی مقدار bound مشخص‌شده توسط کاربر می‌باشد.

- **ترکیب توابع هزینه**: این سه تابع هزینه به صورت زیر ترکیب می‌شوند:

$$\mathcal{L} = \mathcal{L}_{\text{adv}}^{f} + \alpha \mathcal{L}_{\text{GAN}} + \beta \mathcal{L}_{\text{hinge}},$$

ین ترکیب تعادل میان کیفیت بصری نمونه‌های متخاصم و نرخ موفقیت حمله را برقرار می‌کند. پارامترها اهمیت نسبی هر کدام از توابع هزینه را در بهینه‌سازی تعیین می‌کنند.

تفاوت بین حمله های جعبه سفید و جعبه سفید را توضیح دهید و بیان کنید مدل ذکر شده چگونه میتواند در حملات جعبه سیاه استفاده شود؟

در حملات جعبه‌سفید، مهاجم به طور کامل به ساختار مدل هدف، شامل معماری و پارامترهای آن، دسترسی دارد. این دسترسی به مهاجم اجازه می‌دهد که با استفاده از اطلاعات داخلی مدل (مانند گرادیان‌ها) نمونه‌های متخاصم تولید کند. روش‌های رایج در حملات جعبه‌سفید شامل FGSM و روش‌های بهینه‌سازی مانند Opt هستند. این نوع حمله بسیار دقیق است، زیرا مستقیماً از اطلاعات داخلی مدل برای ایجاد اختلالات استفاده می‌کند.

در مقابل، حملات جعبه‌سیاه زمانی رخ می‌دهند که مهاجم هیچ اطلاعات مستقیمی از ساختار داخلی مدل هدف یا پارامترهای آن ندارد. در این نوع حمله، مهاجم تنها از طریق ارسال کوئری به مدل و مشاهده خروجی‌ها، اقدام به تولید نمونه‌های متخاصم می‌کند. یکی از استراتژی‌های رایج در حملات جعبه‌سیاه، پدیده انتقال‌پذیری است که در آن مهاجم ابتدا یک مدل محلی آموزش می‌دهد و سپس امیدوار است نمونه‌های متخاصمی که برای این مدل ساخته شده‌اند بتوانند مدل هدف را نیز فریب دهند.

در حملات جعبه‌سیاه، AdvGAN از رویکرد Model Distillation و Dynamic Queries برای تولید نمونه‌های متخاصم استفاده می‌کند. ابتدا یک مدل Distilled برای تقلید رفتار مدل جعبه‌سیاه با حداقل‌سازی اختلاف بین خروجی‌های آن‌ها آموزش داده می‌شود. در روش Static Distillation، مدل Distilled با داده‌های جداگانه آموزش می‌بیند و تولیدکننده AdvGAN با استفاده از آن نمونه‌های متخاصم تولید می‌کند. در Dynamic Distillation مدل Distilled و تولیدکننده به صورت همزمان و پویا به‌روزرسانی می‌شوند؛ به این صورت که هر بار نمونه‌های جدید تولید شده توسط تولیدکننده برای بهبود مدل Distilled ه استفاده می‌شود. این فرایند با Dynamic Queries از مدل هدف همراه است و باعث می‌شود نمونه‌های متخاصم با دقت بالاتری تولید شوند.

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

- GE-AdvGAN: Improving the transferability of adversarial samples by gradient editing-based adversarial generative model
- **خلاصه**:

این مقاله مدل AdvGAN را به‌عنوان رویکردی برای مقابله با حملات Adversarial به طبقه‌بندها معرفی می‌کند. در این روش، Generator مدل GAN تلاش می‌کند نمونه‌های آدرسیاری را به نسخه‌های کمتر آسیب‌دیده تبدیل کند تا این نمونه‌ها به طبقه‌بند وارد شوند. Discriminator مدل نیز به تشخیص صحت و واقعی بودن تصاویر پرداخته و از حملات آدرسیاری جلوگیری می‌کند. آزمایش‌ها روی داده‌های MNIST و CIFAR-10 نشان می‌دهند که این روش توانسته موفقیت حملات آدرسیاری را کاهش دهد و دقت طبقه‌بند را بهبود بخشد.

گسترش AdvGAN:

این مقاله بر اساس چارچوب اولیهAdvGAN ، رویکردی فعال برای مقابله با حملات آدرسیاری ارائه داده است. در اینجا، به‌جای استفاده از GAN تنها برای تولید تصاویر جدید، از آن برای تعمیر و اصلاح نمونه‌های متخاصم استفاده می‌شود تا به طبقه‌بند داده‌های مقاوم‌تری داده شود. این روش پیشرفته‌تر ازGAN های معمول است که صرفاً به تولید تصاویر می‌پرداختند و توجهی به مسائل مربوط به حملات متخاصم نداشتند.

- NODE-AdvGAN: Improving the transferability and perceptual similarity of adversarial examples by dynamic-system-driven adversarial generative model

خلاصه:

این مقاله ازGAN ها برای تولید نمونه‌های آدرسیاری به‌منظور تقویت آموزش و افزایش مقاومت مدل‌ها استفاده می‌کند. در این روش، به‌جای استفاده از تغییرات ثابت در داده‌ها، از GAN برای تولید نمونه‌های آدرسیاری پویا و متنوع بهره برده می‌شود. این نمونه‌ها به داده‌های آموزشی اضافه می‌شوند تا مدل بتواند به‌طور مؤثرتر در برابر حملات متخاصم مقاوم شود. آزمایش‌ها روی مجموعه‌های داده CIFAR-10 و ImageNet نشان داده که این روش از روش‌های آموزش آدرسیاری سنتی مؤثرتر است.

گسترش AdvGAN:

این مقاله به‌جای استفاده از AdvGAN برای دفاع از طبقه‌بند پس از حمله، آن را به‌عنوان ابزاری برای تولید نمونه‌های آدرسیاری در فرایند آموزش استفاده می‌کند. در واقع، AdvGAN در اینجا برای تقویت داده‌های آموزشی و کمک به آموزش مقاوم‌تر مدل به‌کار گرفته می‌شود. این تغییر، از رویکرد دفاعی به سمت تقویت مقاومت مدل‌ها در برابر حملات آدرسیاری در مرحله آموزش حرکت می‌کند.

۲-2: پیاده‌سازی مدل AdvGAN

آماده‌سازی مجموعه دادگان

ابتدا مجموعه دادگان دانلود شده و به سه قسمت آموزش، اعتبارسنجی و آزمایش با نسبت‌های 10/10/80 تقسیم شد. در شکل 2-1 می‌توانید 5 نمونه تصادفی از داده‌ها را مشاهده کنید.

شکل 2-1. 5 نمونه تصادفی از مجموعه دادگان

سپس با استفاده از میانگین (0.4914, 0.4822, 0.4465) و انحراف معیار (0.2023, 0.1994, 0.2010) داده ها را نرمال‌سایز می‌کنیم. در حین آموزش مدل ResNet از این اعداد برای نرمال‌سازی داده‌ها استفاده شده است و به همین علت ما نیز از این اعداد استفاده کردیم.

ارزیابی طبقه‌بندی داده‌های اصلی

سپس با استفاده از مدل از پیش آموزش داده شده ResNet-20، داده‌های آزمایش را طبقه‌بندی کردیم و به دقت 92.6٪ رسیدیم.

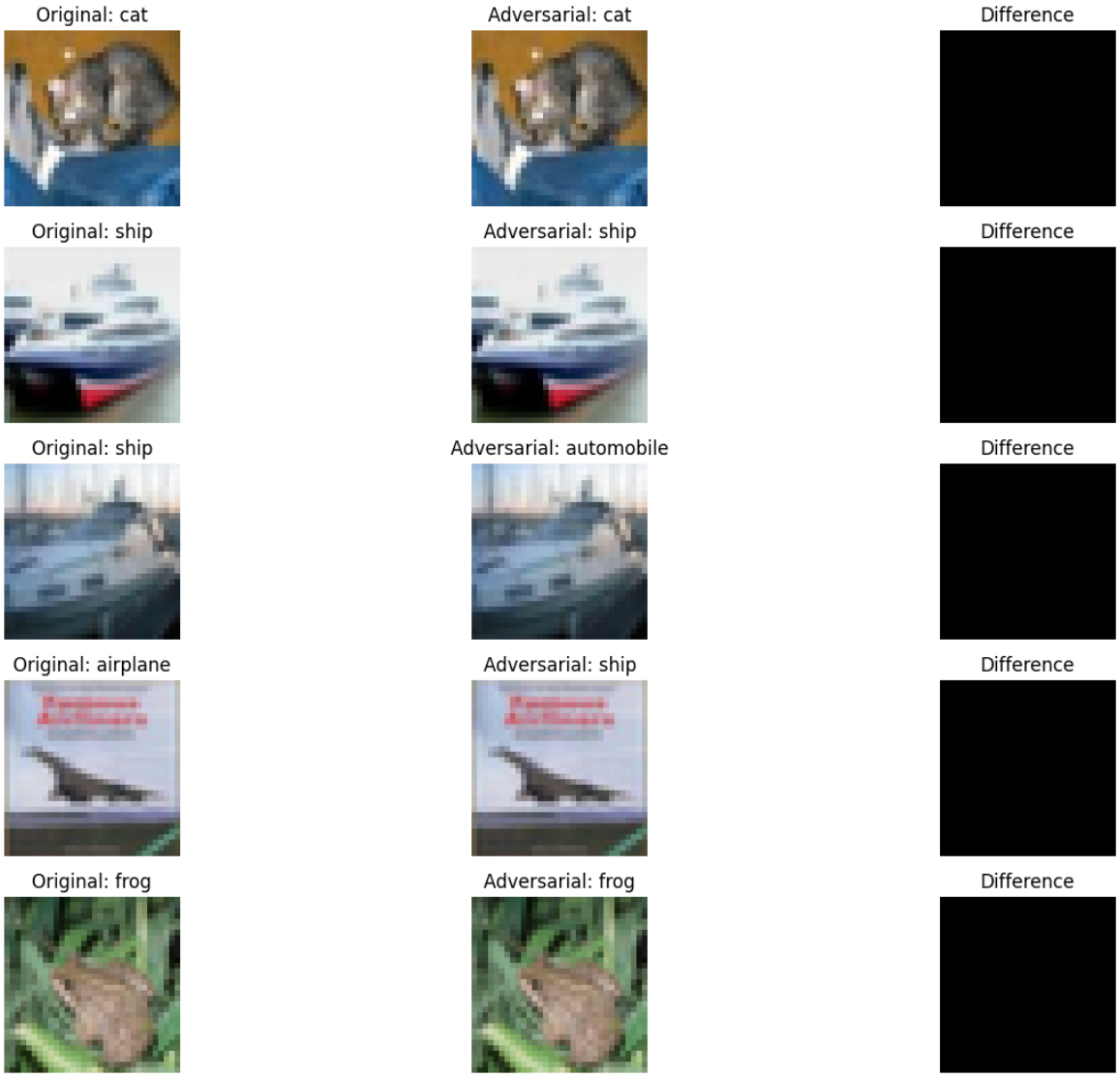
ایجاد و ارزیابی نمونه‌های متخاصم به روش FGSM

حال با استفاده از کتاب‌خانه cleverhance و توابع زیر، نمونه‌های متخاصم ایجاد و ارزیابی می‌شوند.

```
def generate_adversarial_examples(model, dataloader, epsilon):
    adv_images = []
    true_labels = []
    original_images = []
    model.eval()

    for images, labels in dataloader:
        images, labels = images.to(device), labels.to(device)
        images.requires_grad = True

        adv_images_batch = fast_gradient_method(model, images, epsilon,
np.inf)

        adv_images.append(adv_images_batch.detach())
        original_images.append(images.detach())
        true_labels.append(labels)

    return torch.cat(original_images), torch.cat(adv_images),
torch.cat(true_labels)
```

```python
def attack_success(model, adversarial_images, labels):
    with torch.no_grad():
        adv_preds = model(adversarial_images).argmax(dim=1)
        success = (labels != adv_preds).sum().item()
    return success
```

میزان نرخ موفقیت حمله در این حالت، 31.35٪ می‌باشد. در شکل 2-2 می‌توانید 5 نمونه از نمونه‌های متخاصم را در کنار نمونه‌های اصلی و تفاوت آن‌ها را به همراه پیشبینی مدل برای نمونه متخاصم، مشاهده کنید.



شکل 2-2. 5 نمونه از داده‌های متخاصم به روش FGSM

ایجاد و ارزیابی نمونه‌های متخاصم به روش AdvGAN

ابتدا مدل‌های Generator و Discriminator را طراحی کرده و سپس آموزش آن‌ها را آغاز می‌کنیم. در جدول 2-1 می‌توانید مقدار هایپرپارامترها را مشاهده کنید.

جدول 2-1. هایپرپارامترهای استفاده شده در مدل

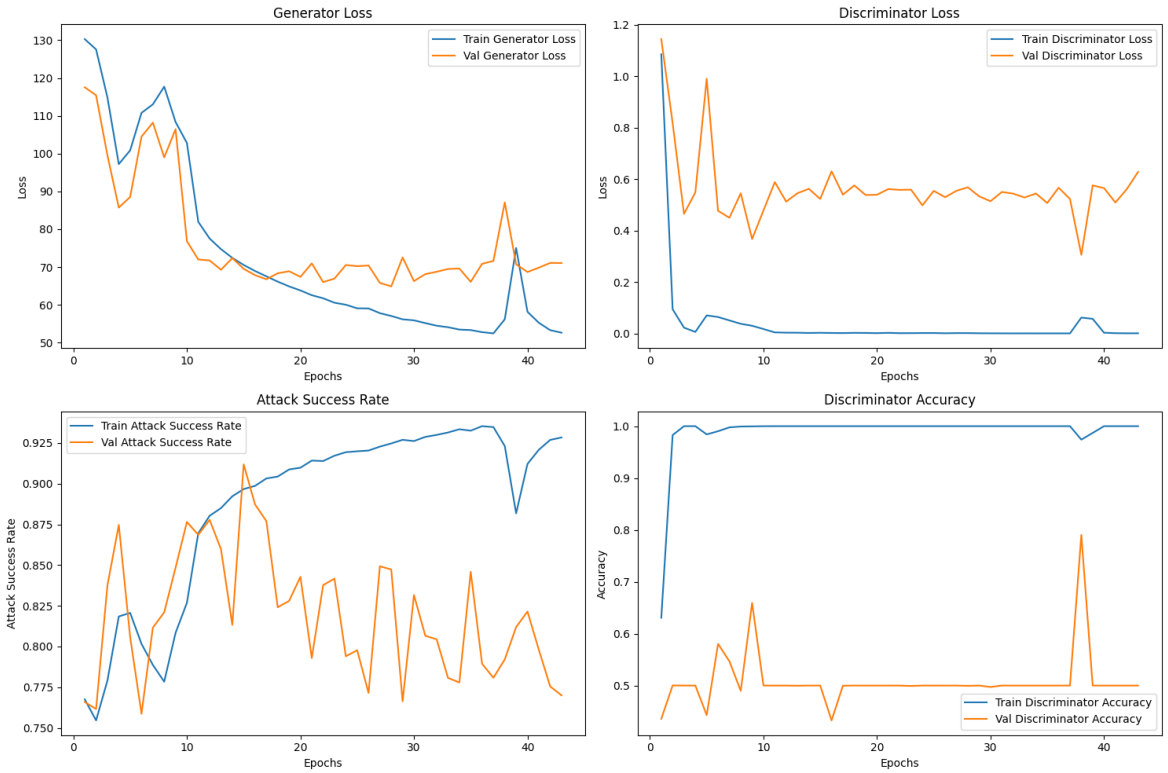
128	Batch size
0.01	Learning rate
50	Epochs
15	Patience
15	Alpha
10	Beta
8/25	C
0.01	Epsilon

در حین آموزش مدل، بعد از حدود 30 دوره شروع به بیش‌پردازش می‌کند اما به علت اینکه در صورت پروژه گفته شده که 50 دوره آموزش ادامه پیدا کند، آموزش ادامه پیدا کرد و در دوره 43ام، مکانیزم توقف زودرس باعث اتمام آموزش شد. در جدول 2-2 می‌توانید مقادیر ارزیابی شده برای بهترین مدل آموزش داده شده که ذخیره شده را مشاهده نمایید.

جدول 2-2. نتایج ارزیابی بهترین مدل آموزش داده شده

57.0785	Train Generator Loss
64.8859	Val Generator Loss
0.0016	Train Discriminator Loss
0.5684	val Discriminator Loss
0.9247	Train Attack Success Rate
0.8473	Val Attack Success Rate

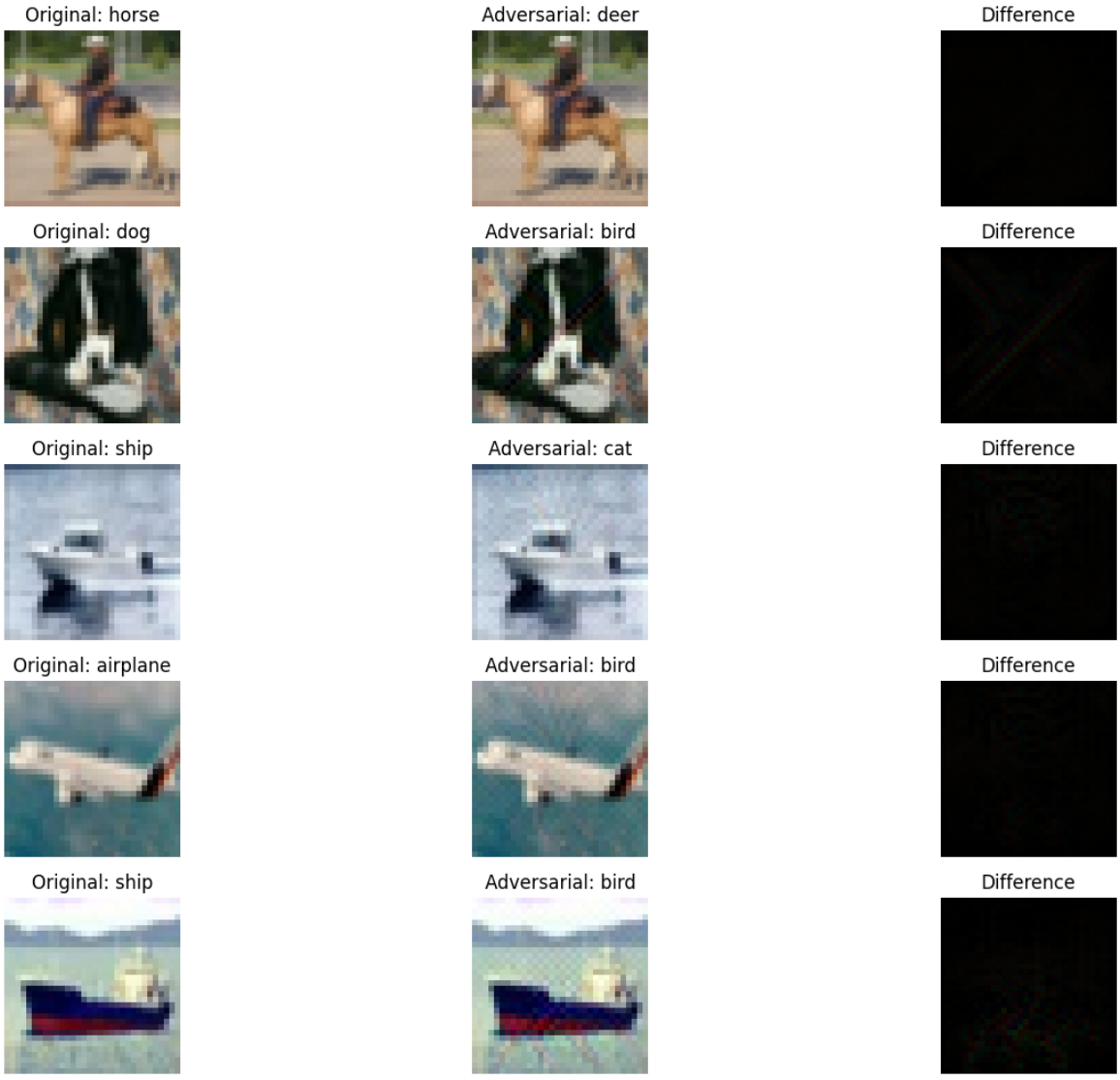
شکل 2-3 نشان دهنده مقادیر هزینه و دقت ماژول‌های مختلف در طول دوره آموزش می‌باشد.



شکل 2-3. مقادیر هزینه و دقت ماژول‌های مختلف در طول دوره آموزش

همانطور که در شکل 2-3 نیز مشخص است، پس از دوره 28ام مقدار هزینه داده اعتبارسنجی شروع به افزایش می‌کند پس مدل دوره 28ام به عنوان بهترین مدل انتخاب شده است.

در شکل 2-4 می‌توانید 5 نمونه از داده‌ها را به همراه نمونه متخاصم آن‌ها و تفاوتشان مشاهده کنید.



شکل 2-4. 5 نمونه از داده‌های متخاصم به روش AdvGAN

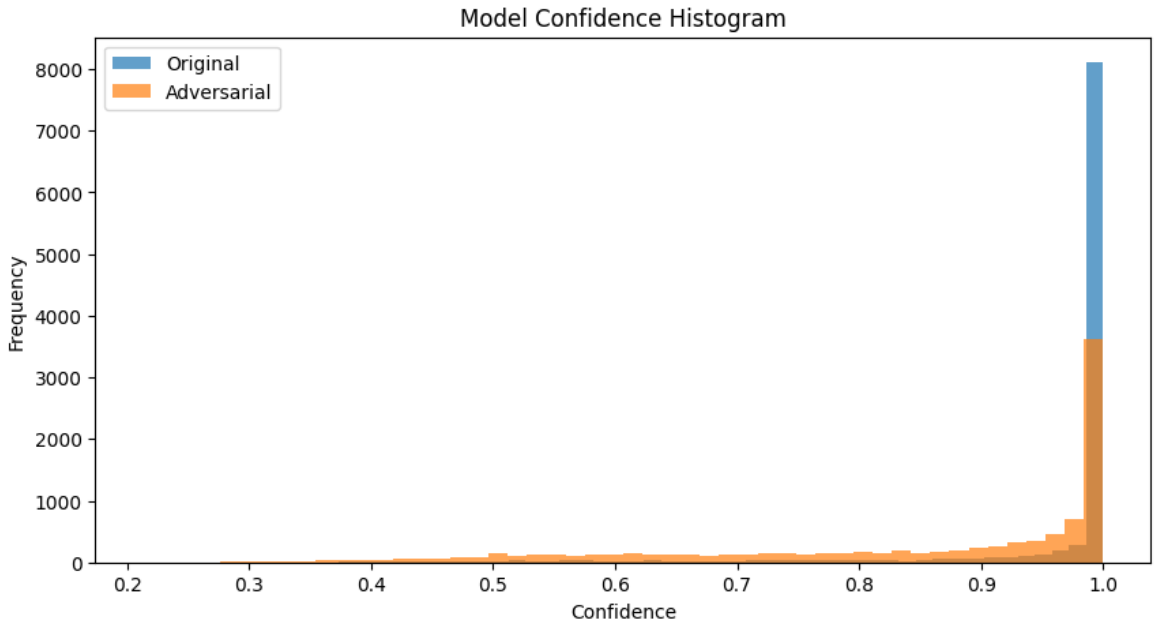
میزان نرخ موفقیت حمله روی داده آزمایش، 84.75٪ می‌باشد که نرخ مناسبی است. این نرخ با پیچیده‌تر کردن مدل‌ها و تنظیم بهتر هایپرپارامترها (مخصوصا آلفا و بتا که در حال حاضر با آزمون و خطا مقداردهی داده شده اند) قابل انجام است.

در جدول 2-3 می‌توانید میزان نرخ موفقیت حمله برای کلاس‌های مختلف روی داده آزمایش را بررسی کنید.

جدول 2-3. میزان نرخ موفقیت حمله برای کلاس‌های مختلف روی داده آزمایش

شماره کلاس	نام کلاس	نرخ موفقیت حمله
1	Airplane	81.10
2	Automobile	81.20
3	Bird	68.00
4	Cat	62.10
5	Deer	93.00
6	Dog	96.80
7	Frog	90.60
8	Horse	92.10
9	Ship	91.70
10	Truck	90.90

در نهایت نیز شکل 2-5 نشان دهنده نمودار هیستوگرام میزان اطمینان مدل روی داده آزمایش می‌باشد. همانطور که مشاهده می‌کنید، مدل نمونه‌های اصلی را با اطمینان بیشتری نسبت به نمونه‌های متخاصم طبقه‌بندی می‌کند.



شکل 2-5. هیستوگرام میزان اطمینان مدل روی داده آزمایش