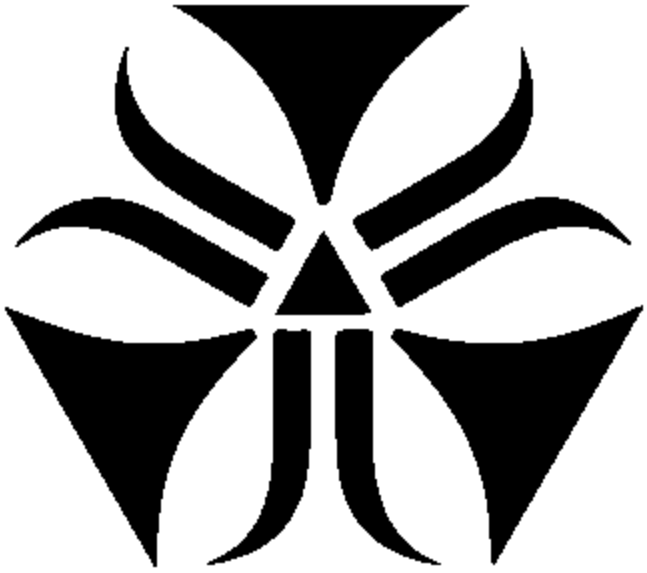
بسمه تعالی



دانشگاه صنعتی همدان

گزارش کار مبانی بینایی کامپیوتر

اعضای گروه: آرمان اسماعیلی، نیما رضایی

نام استاد : جناب دکتر سیفی‌پور

ترم ۴۰۴۱

شرح پروژه

در این پروژه ما به طراحی و پیاده‌سازی یک مدل مبتنی بر شبکه‌های مولد تخاصمی شرطی پرداختیم که هدف آن بازسازی تصاویر ماهواره‌ای از روی نسخه‌های مخدوش‌شده تصاویر نقشه است. مسئله اصلی پروژه، یادگیری یک نگاشت غیرخطی پیچیده بین فضای تصاویر نقشه و فضای تصاویر ماهواره‌ای بود، به‌گونه‌ای که مدل بتواند با دریافت یک تصویر نقشه که به‌صورت مصنوعی دچار تخریب شده است، نسخه بازسازی‌شده و واقع‌گرایانه‌ای از تصویر ماهواره‌ای متناظر را تولید کند.

رویکرد ما مبتنی بر Conditional Generative Adversarial Networks یا cGAN بود. در این ساختار، یک شبکه مولد وظیفه تولید تصویر بازسازی‌شده را بر عهده دارد و یک شبکه متمایزکننده سعی می‌کند تشخیص دهد که تصویر تولیدشده واقعی است یا توسط مولد ساخته شده است. این رقابت کنترل‌شده بین دو شبکه باعث می‌شود مولد به مرور زمان تصاویری تولید کند که از نظر ساختار، بافت و جزئیات به تصاویر واقعی نزدیک‌تر باشند.

در این پروژه ما مدل را به‌صورت کامل از پایه طراحی و پیاده‌سازی کردیم و از هیچ مدل از پیش‌آموزش‌دیده‌ای استفاده نکردیم. معماری مولد بر اساس ساختار U-Net طراحی شد تا بتواند همزمان اطلاعات سطح پایین و سطح بالا را از طریق اتصال‌های میان‌بر حفظ کند. متمایزکننده نیز به صورت PatchGAN پیاده‌سازی شد تا به‌جای ارزیابی کل تصویر، به صورت محلی و مبتنی بر پچ تصمیم‌گیری کند و کیفیت بافت‌ها را بهتر یاد بگیرد.

دیتاست مورد استفاده شامل تصاویر ترکیبی نقشه و ماهواره بود که به صورت جفت‌های متناظر تهیه شده بودند. این تصاویر به رزولوشن ۱۲۸ در ۱۲۸ پیکسل تبدیل شدند و نرمال‌سازی مناسبی روی آن‌ها انجام گرفت تا در بازه استاندارد شبکه‌های عصبی قرار گیرند. علاوه بر این، ما فرآیند مخدوش‌سازی مصنوعی را در مرحله آموزش وارد کردیم تا مدل واقعاً یاد بگیرد از داده ناقص، تصویر کامل تولید کند و صرفاً به نگاشت مستقیم متکی نباشد.

مدل با استفاده از ترکیب زیان تخاصمی و زیان بازسازی L1 آموزش داده شد تا هم واقع‌گرایی تصویر و هم شباهت پیکسلی به هدف واقعی تضمین شود. پس از اتمام آموزش، علاوه بر ارزیابی کیفی نتایج، یک رابط کاربری ساده و کاربردی طراحی شد تا کاربر بتواند تصویر ورودی را بارگذاری کند و خروجی بازسازی‌شده را مشاهده نماید.

هدف ما در این پروژه تنها پیاده‌سازی یک GAN ساده نبود، بلکه تلاش کردیم یک سامانه کامل از آماده‌سازی داده تا طراحی معماری، آموزش پایدار، ارزیابی و ارائه رابط کاربری نهایی بسازیم که از نظر علمی قابل دفاع و از نظر مهندسی قابل استفاده باشد.

آماده‌سازی محیط اجرایی و بارگذاری دیتاست

در این بخش ما زیرساخت اجرایی کل پروژه را آماده می‌کنیم. این سل مسئول اتصال به فضای ذخیره‌سازی، وارد کردن کتابخانه‌های اصلی، نصب وابستگی‌های تکمیلی و استخراج دیتاست است. بدون اجرای صحیح این مرحله، هیچ‌یک از فازهای بعدی شامل آماده‌سازی داده، طراحی مدل یا آموزش قابل انجام نخواهد بود.

قطعه کد اول مربوط به اتصال Google Drive است:

```
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
zip_path = '/content/drive/MyDrive/dataset.zip'
extract_path = '/content/dataset'
```

```
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)
print("dataset extracted")
print("folders:", os.listdir(extract_path))
```

```
dataset extracted
folders: ['dataset']
```

در این قسمت، Google Drive به محیط Google Colab متصل می‌شود. استفاده از force_remount=True باعث می‌شود اگر قبلاً اتصال برقرار شده باشد، مجدداً و به‌صورت تمیز mount شود. این کار از بروز خطاهای دسترسی به فایل جلوگیری می‌کند، مخصوصاً زمانی که پروژه

چند بار اجرا می‌شود یا session جدیدی ایجاد می‌گردد. از آنجا که دیتاست ما در Google Drive ذخیره شده است، این مرحله برای دسترسی به داده‌ها کاملاً ضروری است.

تعریف کلاس دیتاست سفارشی Map2SatelliteDataset با اعمال تخریب تصادفی روی ورودی‌ها :

در این سلول کلاس Map2SatelliteDataset برای مدیریت داده‌ها در مسئله Image-to-Image Translation طراحی شده است. این کلاس مسئول بارگذاری تصاویر، جداسازی ورودی و هدف، اعمال پیش‌پردازش و انجام یک نوع داده‌افزایی مبتنی بر حذف بخشی از تصویر ورودی است. ساختار این کلاس به گونه‌ای طراحی شده که مستقیماً قابل استفاده در DataLoader باشد و بتواند داده‌ها را به صورت جفت ورودی و هدف در اختیار مدل قرار دهد.

بخش ابتدایی کلاس مربوط به سازنده آن است:

```python
class Map2SatelliteDataset(Dataset):
    def __init__(self, root_dir, is_train=True, image_size=256):
        self.root_dir = root_dir
        self.is_train = is_train
        self.image_size = image_size
        self.images = []

        if os.path.exists(root_dir):
            for ext in ['.jpg', '.png', '.jpeg']:
                self.images.extend([f for f in os.listdir(root_dir) if f.lower().endswith(ext)])
            print(f"{os.path.basename(root_dir)}: {len(self.images)} images")
```

در این قسمت مسیر پوشه داده‌ها دریافت شده و تمامی فایل‌های تصویری با پسوندهای رایج جمع‌آوری می‌شوند. متغیر is_train تعیین می‌کند که دیتاست در حالت آموزش است یا ارزیابی. همچنین image_size اندازه نهایی تصاویر را مشخص می‌کند که در اینجا 256 در نظر گرفته شده است. این مقدار برای کنترل مصرف حافظه و سازگاری با معماری شبکه اهمیت دارد.

مهم‌ترین بخش این سلول تابع apply_mask است که منطق اصلی داده‌افزایی در این پروژه را پیاده‌سازی می‌کند:

```
    def apply_mask(self, img):
        img = img.copy()
        h, w = img.shape[:2]

        x = random.randint(0, w // 2)
        y = random.randint(0, h // 2)
        rect_w = random.randint(w // 6, w // 3)
        rect_h = random.randint(h // 6, h // 3)

        img[y:y+rect_h, x:x+rect_w] = 0

        return img

    def __len__(self):
        return len(self.images)
```

در این تابع ابتدا یک کپی از تصویر ساخته می‌شود تا داده اصلی تغییر نکند. سپس ابعاد تصویر استخراج می‌شود. مختصات شروع ناحیه ماسک به صورت تصادفی در نیمه اول عرض و ارتفاع انتخاب می‌شود. اندازه مستطیل نیز به صورت تصادفی بین یک ششم تا یک سوم ابعاد تصویر تعیین می‌شود. در نهایت پیکسل‌های این ناحیه صفر می‌شوند. این طراحی باعث می‌شود بخشی از اطلاعات ورودی حذف شود و مدل مجبور گردد با تکیه بر زمینه اطراف و ساختار کلی تصویر، نگاشت صحیح را یاد بگیرد. چنین رویکردی باعث افزایش سختی مسئله و تقویت توانایی تعمیم مدل می‌شود.

در متد getitem، عملیات بارگذاری و آماده‌سازی هر نمونه انجام می‌شود:

```
path = os.path.join(self.root_dir, self.images[idx])
img = cv2.imread(path)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

در این قسمت تصویر از دیسک خوانده شده و از فضای رنگی BGR به RGB تبدیل می‌شود تا با استاندارد رایج در پردازش تصویر سازگار شود.

سپس تصویر به دو نیمه تقسیم می‌شود:

```
h, w = img.shape[:2]
w2 = w // 2

inp = img[:, :w2, :]
target = img[:, w2:, :]
```

نیمه چپ تصویر به عنوان ورودی و نیمه راست به عنوان هدف در نظر گرفته می‌شود. این ساختار به این معناست که هر فایل تصویری شامل یک جفت داده ورودی و خروجی است که برای یادگیری نظارت‌شده استفاده می‌شود.

پس از آن هر دو تصویر به اندازه مشخص‌شده تغییر ابعاد داده می‌شوند:

```
  inp = cv2.resize(inp, (self.image_size, self.image_size))
  target = cv2.resize(target, (self.image_size, self.image_size))
```

این کار باعث می‌شود تمام نمونه‌ها دارای ابعاد یکسان باشند و بتوانند به صورت batch وارد شبکه شوند.

اعمال ماسک تنها در فاز آموزش و با احتمال مشخص انجام می‌شود:

```
if self.is_train and random.random() < 0.7:
    inp = self.apply_mask(inp)
```

این شرط باعث می‌شود داده‌های اعتبارسنجی بدون تخریب باقی بمانند و ارزیابی دقیق‌تری از عملکرد مدل به دست آید. همچنین اعمال ماسک با احتمال هفتاد درصد باعث می‌شود مدل هم نمونه‌های کامل و هم نمونه‌های ناقص را مشاهده کند و در برابر داده‌های ناقص مقاوم‌تر شود.

در نهایت تصاویر به Tensor تبدیل شده و نرمال‌سازی می‌شوند:

```
inp = torch.from_numpy(inp).permute(2, 0, 1).float() / 127.5 - 1
target = torch.from_numpy(target).permute(2, 0, 1).float() / 127.5 - 1
```

در این مرحله ترتیب ابعاد از حالت HWC به CHW تغییر داده می‌شود تا با فرمت مورد انتظار شبکه‌های کانولوشنی سازگار شود. سپس مقادیر پیکسلی از بازه صفر تا 255 به بازه منفی یک تا مثبت یک تبدیل می‌شوند. این نرمال‌سازی به پایداری گرادیان و سازگاری با توابع فعال‌سازی خروجی کمک می‌کند.

در مجموع، این سلول زیرساخت داده‌ای پروژه را تعریف می‌کند و با افزودن مکانیزم Mask تصادفی، مسئله را به یک سناریوی چالش‌برانگیزتر تبدیل می‌کند که می‌تواند موجب بهبود قابلیت تعمیم و پایداری مدل در شرایط ورودی ناقص شود.

بارگذاری خودکار داده‌ها، ساخت DataLoader و بررسی بصری نمونه‌ها:

در این سلول فرآیند شناسایی مسیر داده‌ها، ساخت دیتاست‌ها، ایجاد DataLoader و همچنین بررسی بصری نمونه‌ها پیاده‌سازی شده است. این مرحله نقش کنترلی بسیار مهمی در پروژه دارد، زیرا پیش از ورود به فاز آموزش، صحت ساختار داده‌ها، نحوه جداسازی ورودی و هدف، و عملکرد مکانیزم Mask به صورت عملی ارزیابی می‌شود.

در ابتدای کد، مسیر پایه داده‌ها تعریف شده و با استفاده از پیمایش بازگشتی پوشه‌ها، مسیر مربوط به پوشه‌های train و val به صورت خودکار شناسایی می‌شود:

```
print("\nLoading datasets...")

base_path = '/content/dataset'
train_path = None
val_path = None

for root, dirs, files in os.walk(base_path):
    if 'train' in dirs:
        train_path = os.path.join(root, 'train')
    if 'val' in dirs or 'validation' in dirs:
        val_path = os.path.join(root, 'val') if 'val' in dirs else os.path.join(root, 'validation')
```

این ساختار باعث می‌شود کد نسبت به تغییرات جزئی در ساختار پوشه‌ها انعطاف‌پذیر باشد و وابسته به یک مسیر ثابت و سخت‌کد شده نباشد. چنین طراحی‌ای در پروژه‌های عملی اهمیت بالایی دارد.

پس از یافتن مسیرها، دیتاست‌های آموزش و اعتبارسنجی ساخته می‌شوند:

```
train_dataset = Map2SatelliteDataset(train_path, is_train=True)
val_dataset = Map2SatelliteDataset(val_path, is_train=False)
```

در اینجا تفاوت کلیدی در مقدار is_train است. دیتاست آموزش با امکان اعمال Mask ساخته می‌شود، در حالی که دیتاست اعتبارسنجی کاملاً بدون تخریب باقی می‌ماند. این جداسازی تضمین می‌کند که ارزیابی مدل روی داده‌های تمیز و واقعی انجام شود.

سپس DataLoader ها ایجاد می‌شوند:

```
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True, num_workers=2)
val_loader = DataLoader(val_dataset, batch_size=8, shuffle=False, num_workers=2)
```

در فاز آموزش، داده‌ها shuffle می‌شوند تا هم‌بستگی ترتیبی حذف شود و یادگیری پایدارتر انجام گیرد. در فاز اعتبارسنجی ترتیب داده‌ها حفظ می‌شود. اندازه batch برابر با 8 انتخاب شده که تعادلی بین

مصرف حافظه و پایداری گرادیان برقرار می‌کند. استفاده از num_workers برابر با 2 باعث بارگذاری موازی داده‌ها و افزایش سرعت آموزش می‌شود.

بخش مهم بعدی این سلول، بررسی بصری عملکرد دیتاست است. ابتدا تابعی برای بازگرداندن داده‌ها از بازه نرمال‌شده به فضای پیکسلی تعریف می‌شود:

```
def denormalize(tensor):
    return ((tensor.cpu().numpy().transpose(1, 2, 0) + 1) * 127.5).astype(np.uint8)
```

این تابع مقادیر را از بازه منفی یک تا مثبت یک به بازه صفر تا 255 تبدیل می‌کند و ترتیب ابعاد را به فرمت قابل نمایش تغییر می‌دهد.

در ادامه برای سه نمونه مشخص، دو نسخه از ورودی استخراج می‌شود:

```
train_dataset.is_train = False
inp_clean, target = train_dataset[idx]

# Get masked version (with mask) - training mode
train_dataset.is_train = True
inp_masked, _ = train_dataset[idx]
```

در اینجا به صورت کنترل‌شده ابتدا نسخه تمیز تصویر گرفته می‌شود، سپس همان نمونه با اعمال Mask استخراج می‌شود. این کار امکان مقایسه مستقیم اثر Mask را فراهم می‌کند. پس از استخراج تصاویر، هر سه بخش شامل ورودی تمیز، ورودی ماسک‌شده و تصویر هدف نمایش داده می‌شوند.

ساختار نمایش به صورت یک ماتریس سه در سه است که در هر ردیف یک نمونه کامل نشان داده می‌شود. این مرحله اهمیت بالایی دارد زیرا تأیید می‌کند که جداسازی نیمه‌های تصویر، اعمال Mask، نرمال‌سازی و بازگردانی تصویر همگی به درستی انجام شده‌اند.

در انتهای سلول خلاصه‌ای از وضعیت دیتاست چاپ می‌شود که شامل تعداد تصاویر آموزش و اعتبارسنجی، اندازه تصویر و جهت نگاشت دامنه است. این خروجی نقش یک گزارش اولیه از وضعیت داده‌ها را ایفا می‌کند و پیش از شروع آموزش، یک دید کلی از مقیاس مسئله ارائه می‌دهد.

در مجموع، این سلول نقش مرحله اعتبارسنجی داده‌ها را دارد و تضمین می‌کند که زیرساخت داده‌ای پروژه به درستی پیکربندی شده است. انجام این بررسی پیش از شروع آموزش، از بروز خطاهای پنهان در مراحل بعدی جلوگیری می‌کند و اطمینان می‌دهد که مدل روی داده‌هایی صحیح و آماده آموزش قرار خواهد گرفت.

طراحی معماری Generator و Discriminator برای Conditional GAN:

در این سلول معماری کامل شبکه مولد و شبکه تمایزدهنده برای یک مدل GAN شرطی پیاده‌سازی شده است. ساختار کلی مبتنی بر یک Generator از نوع Encoder–Decoder با اتصالات میان‌بر و یک Discriminator کانولوشنی عمیق است که روی جفت تصویر ورودی و خروجی قضاوت می‌کند. این طراحی به گونه‌ای انجام شده که هم ساختار مکانی تصویر حفظ شود و هم مدل توانایی تولید جزئیات واقع‌گرایانه را داشته باشد.

ابتدا کلاس Generator تعریف شده است. بخش Encoder شامل چهار بلوک کانولوشنی متوالی است:

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.enc1 = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1),
            nn.LeakyReLU(0.2)
        )
```

در اولین لایه، تصویر سه‌کاناله ورودی با یک کانولوشن با کرنل 4 و stride برابر 2 به 64 کانال تبدیل می‌شود. استفاده از stride برابر 2 باعث کاهش تدریجی ابعاد فضایی و استخراج ویژگی‌های سطح پایین می‌شود. تابع فعال‌سازی LeakyReLU با شیب 0.2 انتخاب شده تا از مشکل مرگ نورون‌ها جلوگیری کند.

در لایه‌های بعدی Encoder تعداد کانال‌ها به ترتیب به 128، 256 و 512 افزایش می‌یابد:

```
nn.Conv2d(64, 128, 4, 2, 1),
nn.BatchNorm2d(128),
nn.LeakyReLU(0.2)
```

در این لایه‌ها از Batch Normalization استفاده شده است تا توزیع فعال‌سازی‌ها پایدارتر شود و سرعت همگرایی افزایش یابد. با افزایش عمق، شبکه قادر به استخراج ویژگی‌های سطح بالاتر و مفاهیم ساختاری پیچیده‌تر می‌شود.

بخش Decoder به صورت معکوس عمل می‌کند و با استفاده از ConvTranspose2d ابعاد تصویر را بازسازی می‌کند:

```
)
self.dec1 = nn.Sequential(
    nn.ConvTranspose2d(512, 256, 4, 2, 1),
    nn.BatchNorm2d(256),
    nn.Dropout(0.5),
    nn.ReLU()
)
```

در اولین لایه Decoder از Dropout با نرخ 0.5 استفاده شده است. این کار به عنوان یک تکنیک منظم‌سازی عمل می‌کند و از بیش‌برازش جلوگیری می‌کند. همچنین تابع فعال‌سازی ReLU برای بازسازی ویژگی‌ها به کار رفته است.

نکته بسیار مهم در معماری Generator استفاده از اتصال‌های میان‌بر در تابع forward است:

```
d2 = self.dec2(torch.cat([d1, e3], 1))
d3 = self.dec3(torch.cat([d2, e2], 1))
d4 = self.dec4(torch.cat([d3, e1], 1))
```

در اینجا خروجی لایه‌های Encoder با خروجی لایه‌های Decoder در سطوح متناظر الحاق می‌شوند. این ساختار مشابه U-Net است و باعث می‌شود اطلاعات مکانی با دقت بالا که در مسیر پایین‌رو فشرده شده‌اند، مستقیماً به مسیر بالارو منتقل شوند. نتیجه این طراحی حفظ جزئیات فضایی و بهبود کیفیت بازسازی تصویر است.

در لایه نهایی از Tanh استفاده شده است:

```
    nn.ConvTranspose2d(128, 3, 4, 2, 1),
    nn.Tanh()
)
```

تابع Tanh خروجی را به بازه منفی یک تا مثبت یک محدود می‌کند که با نرمال‌سازی داده‌های ورودی سازگار است.

در ادامه کلاس Discriminator تعریف شده است. این شبکه یک مدل کانولوشنی عمیق است که تصویر ورودی و تصویر هدف را به صورت کانال‌به‌کانال با هم ترکیب می‌کند:

```
nn.Conv2d(6, 64, 4, 2, 1), nn.LeakyReLU(0.2),
```

از آنجا که هر تصویر سه کانال دارد، الحاق ورودی و خروجی منجر به تصویری با شش کانال می‌شود. این ساختار شرطی بودن GAN را پیاده‌سازی می‌کند، زیرا Discriminator باید تشخیص دهد که آیا تصویر خروجی با ورودی سازگار و واقعی است یا خیر.

لایه‌های بعدی Discriminator به تدریج تعداد کانال‌ها را افزایش می‌دهند و با استفاده از BatchNorm و LeakyReLU ویژگی‌های پیچیده‌تری استخراج می‌کنند:

```
nn.Conv2d(128, 256, 4, 2, 1),
nn.BatchNorm2d(256),
nn.LeakyReLU(0.2)
```

در لایه نهایی، یک کانولوشن با خروجی تک‌کاناله اعمال شده و سپس Sigmoid استفاده می‌شود:

```
nn.Conv2d(512, 1, 4, 1, 1),
nn.Sigmoid()
```

خروجی این بخش یک نقشه احتمال است که نشان می‌دهد هر ناحیه از تصویر تا چه اندازه واقعی تشخیص داده می‌شود. استفاده از Sigmoid باعث می‌شود خروجی در بازه صفر تا یک قرار گیرد و بتوان از آن در محاسبه تابع هزینه باینری استفاده کرد.

در مجموع، Generator با ساختار Encoder–Decoder و اتصال‌های میان‌بر مسئول تولید تصویر هدف است، در حالی که Discriminator وظیفه ارزیابی واقع‌گرایانه بودن خروجی را بر عهده دارد. این طراحی مکمل یکدیگر هستند و در قالب GAN شرطی امکان یادگیری نگاشت ساختاری بین دو دامنه تصویری را فراهم می‌کنند.

پیکربندی فرآیند آموزش GAN، محاسبه متریک‌ها و ذخیره‌سازی مدل:

در این سلول کل فرآیند آموزش مدل GAN شرطی پیاده‌سازی شده است. این بخش شامل انتخاب دستگاه محاسباتی، مقداردهی اولیه مدل‌ها و بهینه‌سازها، تعریف توابع هزینه، حلقه آموزش، ارزیابی دوره‌ای با متریک‌های کمی و در نهایت ذخیره بهترین و آخرین نسخه مدل است. این سلول هسته اجرایی پروژه محسوب می‌شود و کیفیت طراحی آن مستقیماً بر پایداری و عملکرد نهایی مدل تأثیر دارد.

سپس مدل‌های Generator و Discriminator ساخته شده و به دستگاه منتقل می‌شوند:

```
G = Generator().to(device)
D = Discriminator().to(device)
```

بهینه‌سازها از نوع Adam انتخاب شده‌اند:

```
opt_g = optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
opt_d = optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

نرخ یادگیری 0.0002 و پارامترهای بتا برابر با 0.5 و 0.999 تنظیم شده‌اند. این تنظیمات در آموزش GAN رایج هستند، زیرا مقدار بتای اول برابر 0.5 باعث کاهش نوسانات شدید در گرادیان‌های متغیر GAN می‌شود و پایداری آموزش را افزایش می‌دهد.

دو تابع هزینه تعریف شده‌اند:

```
bce = nn.BCELoss()
l1 = nn.L1Loss()
```

تابع BCE برای بخش GAN استفاده می‌شود تا Discriminator بتواند واقعی یا جعلی بودن تصویر را تشخیص دهد. تابع L1 برای اندازه‌گیری فاصله پیکسلی بین تصویر تولیدی و تصویر واقعی استفاده

می‌شود. ترکیب این دو تابع باعث می‌شود Generator هم به سمت تولید تصاویر واقع‌گرایانه حرکت کند و هم شباهت ساختاری با تصویر هدف را حفظ کند.

تابع calculate_metrics برای ارزیابی کمی مدل طراحی شده است:

```python
def calculate_metrics(generator, loader, device):
    generator.eval()
    psnr_values = []
    ssim_values = []

    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            fake = generator(x)

            for i in range(x.shape[0]):
                fake_img = fake[i].cpu().numpy().transpose(1, 2, 0)
                real_img = y[i].cpu().numpy().transpose(1, 2, 0)

                fake_img = ((fake_img + 1) * 127.5).astype(np.uint8)
                real_img = ((real_img + 1) * 127.5).astype(np.uint8)

                psnr_val = psnr(real_img, fake_img, data_range=255)
                ssim_val = ssim(real_img, fake_img, data_range=255, channel_axis=2, win_size=11)
```

در این تابع مدل در حالت ارزیابی قرار می‌گیرد و بدون محاسبه گرادیان، خروجی‌ها تولید می‌شوند. سپس تصاویر از بازه منفی یک تا مثبت یک به بازه صفر تا 255 بازگردانده می‌شوند. دو متریک PSNR و SSIM محاسبه می‌شوند. PSNR میزان خطای پیکسلی را اندازه می‌گیرد و SSIM میزان شباهت ساختاری را ارزیابی می‌کند. استفاده همزمان از این دو معیار دید جامع‌تری نسبت به کیفیت خروجی فراهم می‌کند.

حلقه آموزش برای 50 epoch اجرا می‌شود. در هر epoch ابتدا مدل‌ها در حالت train قرار می‌گیرند. در هر batch مراحل زیر انجام می‌شود:

```python
fake = G(x)
```

ابتدا Generator تصویر جعلی تولید می‌کند.

سپس Discriminator آموزش داده می‌شود:

```
d_real = bce(D(x, y), torch.ones_like(D(x, y)))
d_fake = bce(D(x, fake.detach()), torch.zeros_like(D(x, fake.detach())))
d_loss = (d_real + d_fake) * 0.5
```

در اینجا خروجی واقعی با برچسب یک و خروجی جعلی با برچسب صفر مقایسه می‌شود. میانگین این دو مقدار به عنوان زیان نهایی Discriminator در نظر گرفته می‌شود. استفاده از detach مانع از انتقال گرادیان به Generator در این مرحله می‌شود.

پس از آن Generator به‌روزرسانی می‌شود:

```
g_gan = bce(D(x, fake), torch.ones_like(D(x, fake)))
g_l1 = l1(fake, y) * 100
g_loss = g_gan + g_l1
```

را فریب دهد و Discriminator تلاش می‌کند GAN شامل دو بخش است. بخش Generator زیان فاصله پیکسلی را محاسبه کرده و با ضریب 100 تقویت می‌شود. L1 خروجی را واقعی نشان دهد. بخش این ضریب وزن بیشتری به شباهت ساختاری می‌دهد و از تولید تصاویر صرفاً واقع‌گرایانه اما نامرتبط جلوگیری می‌کند.

یک بار ارزیابی کامل روی داده‌های آموزش و اعتبارسنجی انجام می‌شود و مقادیر epoch هر پنج در داده‌های اعتبارسنجی بهتر از مقدار قبلی باشد، PSNR ذخیره می‌گردد. اگر مقدار SSIM و PSNR مدل به عنوان بهترین مدل ذخیره می‌شود:

```
if val_psnr > best_psnr:
    best_psnr = val_psnr
    torch.save(G.state_dict(), '/content/generator_best.pth')
```

این معیار انتخاب مبتنی بر کیفیت بازسازی است و نه صرفاً مقدار loss.

همچنین هر ده epoch یک checkpoint کامل شامل وضعیت مدل‌ها، بهینه‌سازها و تاریخچه متریک‌ها ذخیره می‌شود. این کار امکان ادامه آموزش یا تحلیل روند همگرایی را فراهم می‌کند.

ذخیره شده و نسخه‌ای از آن‌ها در Discriminator و Generator در پایان آموزش، مدل نهایی مقادیر نهایی ،PSNR نگهداری می‌شود. علاوه بر آن، اطلاعات کلیدی شامل بهترین Google Drive ذخیره می‌شود. این فایل نقش JSON متریک‌ها، تعداد پارامترها و تنظیمات آموزش در یک فایل .مستندسازی رسمی مدل آموزش‌دیده را ایفا می‌کند

را با ترکیب زیان رقابتی و زیان بازسازی، GAN در مجموع، این سلول چارچوب کامل آموزش ارزیابی کمی دوره‌ای، انتخاب بهترین مدل و ذخیره‌سازی ساختاریافته پیاده‌سازی می‌کند. این طراحی .باعث می‌شود فرآیند آموزش هم پایدار باشد و هم به صورت کمی قابل ارزیابی و مستندسازی باشد

تفسیر کلی روند آموزش

مدل روی GPU اجرا شده و از نظر ظرفیت پارامتری در سطح متوسط قرار دارد. Generator حدود 6.1 میلیون پارامتر و Discriminator حدود 2.7 میلیون پارامتر دارد که برای یک معماری U-Net مبتنی بر pix2pix مقدار منطقی و متعادل محسوب می‌شود. بنابراین از نظر ظرفیت مدل، کمبود جدی مشاهده نمی‌شود.

رفتار آموزش را می‌توان به سه فاز اصلی تقسیم کرد: فاز یادگیری اولیه، فاز همگرایی پایدار، و فاز ناپایداری انتهایی.

فاز اول: epoch 1 تا حدود 10

در epoch اول مقدار G Loss برابر 23.6 است که بسیار بالا است و نشان می‌دهد مدل هنوز نگاشت معناداری یاد نگرفته است. در epoch 2 افت شدید G Loss به حدود 12.7 رخ می‌دهد که نشان‌دهنده یادگیری سریع اولیه است.

در epoch 5 اولین ارزیابی کمی انجام شده است. مقدار Validation PSNR برابر 24.03 dB و SSIM برابر 0.4476 است. این نشان می‌دهد مدل خیلی سریع به یک نگاشت قابل قبول رسیده است.

اما در epoch 10 اتفاق مهمی رخ داده است. Train PSNR به 20.99 dB کاهش یافته در حالی که در epoch 5 برابر 22.34 dB بوده است. این افت نشان می‌دهد مدل در بازه 5 تا 10 دچار نوسان شده و کیفیت بازسازی روی داده‌های آموزش کاهش یافته است. این اولین نشانه ناپایداری جزئی GAN است. همچنین D Loss در این بازه بین 0.29 تا 0.39 نوسان دارد که نشان می‌دهد تعادل بین G و D هنوز کاملاً پایدار نشده است.

بنابراین اولین نقطه‌ای که مدل عملکرد ضعیف‌تری نشان داده حوالی epoch 10 است، به‌خصوص از نظر PSNR روی داده آموزش.

فاز دوم: epoch 15 تا 25

از epoch 15 به بعد روند متریک‌ها به شکل واضحی بهبود پیدا می‌کند. در epoch 20 مقدار Validation PSNR به 25.14 dB می‌رسد و SSIM به حدود 0.58 افزایش می‌یابد. این نشان می‌دهد مدل وارد فاز همگرایی پایدار شده است.

در این بازه D Loss بین 0.3 تا 0.39 نوسان دارد و G Loss بین 10.8 تا 11.5 است. این نسبت نشان می‌دهد رقابت بین Generator و Discriminator تقریباً متعادل است. بهترین تعادل اولیه مدل در همین بازه شکل گرفته است.

در epoch 25 نیز عملکرد تقریباً پایدار باقی مانده و افت شدیدی مشاهده نمی‌شود. بنابراین بازه 15 تا 25 بهترین فاز پایداری اولیه مدل محسوب می‌شود.

فاز سوم: epoch 28 تا حدود 42

از epoch 28 به بعد رفتار Discriminator تغییر می‌کند. D Loss به شدت کاهش پیدا می‌کند:

در epoch 29 مقدار D برابر 0.189
در epoch 30 مقدار D برابر 0.149
در epoch 33 مقدار D برابر 0.131

در epoch 34 مقدار D برابر 0.047

در epoch 42 مقدار D برابر 0.026

این کاهش شدید نشان می‌دهد Discriminator بیش از حد قوی شده است و تقریباً بدون خطا تصاویر جعلی را تشخیص می‌دهد. در همین بازه G Loss افزایش پیدا می‌کند و به بالای 14 و حتی 15 می‌رسد. این یک نشانه کلاسیک از عدم تعادل GAN است؛ یعنی Discriminator از Generator جلو افتاده است.

پیامد این اتفاق در متریک‌ها دیده می‌شود. در epoch 30 مقدار Validation PSNR از 25.14 به 24.57 کاهش یافته است. در epoch 40 نیز PSNR به 24.95 کاهش پیدا کرده است. همچنین Train PSNR از 24.65 در epoch 35 به 23.78 در epoch 40 افت می‌کند. این افت نشان می‌دهد کیفیت بازسازی کاهش یافته است.

بنابراین دومین ناحیه‌ای که مدل عملکرد ضعیف‌تری داشته، بازه 28 تا 42 است که Discriminator بیش از حد قوی شده و باعث نوسان Generator شده است.

بهترین عملکرد مدل

بهترین Validation PSNR برابر 25.36 dB در epoch 35 ثبت شده است. در این نقطه:

D Loss برابر 0.183

G Loss برابر 13.582

Validation SSIM برابر 0.5969

این نقطه تعادل مناسبی بین قدرت D و توانایی بازسازی G ایجاد کرده است. بعد از این نقطه نوسانات بیشتر شده‌اند و بهبود معناداری مشاهده نمی‌شود.

تحلیل نهایی عملکرد

مدل به طور کلی عملکرد قابل قبولی داشته است. اختلاف Train و Validation PSNR کم است که نشان می‌دهد overfitting شدید رخ نداده است. SSIM نهایی حدود 0.60 است که برای تبدیل Satellite به Map مقدار متوسط رو به خوب محسوب می‌شود، اما هنوز فاصله تا بازسازی با جزئیات بسیار بالا وجود دارد.

نقاط ضعف اصلی مدل در دو ناحیه مشاهده می‌شود:

اول حوالی epoch 10 که افت PSNR روی داده آموزش رخ داده و نشان‌دهنده نوسان اولیه GAN است.

دوم بازه 28 تا 42 که Discriminator بیش از حد قوی شده و D Loss به مقادیر بسیار پایین رسیده است. این مسئله باعث افزایش G Loss و کاهش نسبی PSNR شده است. این نشانه عدم تعادل در بازی رقابتی GAN است.

اگر بخواهیم بهترین مدل را انتخاب کنیم، checkpoint مربوط به epoch 35 منطقی‌ترین انتخاب است، زیرا در آن نقطه بالاترین Validation PSNR و تعادل مناسب بین دو شبکه مشاهده می‌شود.