به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق
تمرین ششم

پرسش ۱	نام و نام خانوادگی	پوریا ذره پرور قوچانی نژاد
	شماره دانشجویی	۸۱۰۱۰۲۱۴۱
پرسش ۲	نام و نام خانوادگی	نام و نام خانوادگی
	شماره دانشجویی	
	مهلت ارسال پاسخ	۱۴۰۱.۰۷.۰۱

فهرست

شکل‌ها

جدول‌ها

پرسش 1. یادگیری بدون نظارت و انتقال دامنه با استفاده از GAN

۱-۲. بخش نظری

۱-۲-۱ :

آموزش شبکه‌های مولد تخاصمی (GAN) به دلیل ماهیت رقابتی و پویای بین مولد (که داده‌های جعلی تولید می‌کند) و تمایزدهنده (که داده‌های واقعی را از جعلی تشخیص می‌دهد) می‌تواند ناپایدار باشد. این رقابت شبیه یک بازی است که اگر یکی از طرفین خیلی قوی شود، فرآیند یادگیری دیگری مختل می‌شود و عمدتا ناشی از چالش های موجود در فرآیند بهینه سازی است .

سه عامل اصلی در بی ثباتی آموزش GAN :

۱- عدم تعادل بین مولد (Generator) وتمایزدهنده (Discriminator) :

در GAN ها مولد و تمایزدهنده به طور همزمان آموزش می بینند که باعث می شود اگر یکی از آن ها بیش از حد قوی شود دیگری نمی تواند به خوبی یاد بگیرد مثلا اگر تمایزدهنده با سرعت بالا یادبگیرد که چگونه داده های واقعی را تشخیص دهد باعث ارسال گرادیان های کوچک و بی معنا به مولد می شود و در نهایت باعث عدم یادگیری کامل مولد خواهد شد که این عدم تعادل باعث ناپایداری فرآیند آموزش خواهد شد .

مکانیزم های پیشنهادی برای مقابله با این مشکل :

۱ – آموزش جایگزین (Alternating Training) :

تنظیم تعداد مراحل آموزش برای مولد و تمایزدهنده به طور جداگانه به طوری که تمایزدهنده یا مولد بیش از قوی نشوند .

۲ – برچسب های نرم (Label smoothing) :

به جای استفاده از برچسب های باینری از برچسب های نرم برای جلوگیری از قطعیت تمایزدهنده استفاده شود .

۳- استفاده از معماری های متعادل و تنظیم عمق شبکه ها برای جلوگیری از قدرت یافتن یکی از شبکه های مولد یا تمایزدهنده و همچنین افزودن لایه های Dropout و Regularization

۲ – **مشکل فروپاشی مد (Mode Collapse) :**

این مشکل زمانی در GAN رخ می دهد که مولد تنها زیر مجموعه ای محدود از توزیع داده های واقعی را تولید کند به جای اینکه تنوع کامل داده ها را بازتولید کند مثلا در مجموعه داده هایی شامل وسایل نقلیه مختلف مولد فقط تصویر یک خودرو مشخص را تولید کند و از سایر وسایل نقلیه صرف نظر کند و تنها بر روی حالت خاصی متمرکز شود که معمولا به دلیل نقص در فرآیند بهینه سازی و گرادیان های تولیدی این مشکل رخ خواهد داد.

مکانیزم های پیشنهادی برای حل این مشکل :

۱ – روش Mini-batch Discrimination : این روش به تمایزدهنده اجازه می دهد تا تنوع نمونه های تولید شده در یک دسته را بررسی کند که باعث می شود مولد به تولید نمونه های متنوع تر تشویق شود .

۲ – استفاده از چند مولد برای تولید نمونه های متنوع تر

۳ – روش Unrolled GANs : این روش به مولد اجازه می دهد چند مرحله بهینه سازی برای تمایزدهنده در نظر بگیرد تا استراتژی های مختلف بررسی شود و تا حد ممکن داده های متنوع تری تولید شود .

۳- مشکل ناپدید شدن گرادیان ها (Vanishing Gradients) :

در GAN مولد برای فریب بهتر تمایزدهنده آموزش می بیند و اگر تمایزدهنده بسیار قوی شود ممکن است احتمالات پایینی به داده های جعلی اختصاص بدهد و منجر به گرادیان های خیلی کوچک خواهد شد و در نتیجه باعث می شود مولد نتواند به درستی به روزرسانی شود و آموزش ببیند و دیگر اطلاعاتی برای بهبود مولد باقی نماند و به نوعی مولد در بک بهینه محلی گیر می کند و پیشرفت نمی کند .

مکانیزم های پیشنهادی برای حل این مشکل :

۱ – استفاده از معیار های فاصله دیگر مانند Wasserstein برای تولید گرادیان های معنا دار و با تکنیک هایی مانند جریمه گرادیان در جهت بهبود پایداری

۲ – اصلاح تابع هزینه به گونه ای که حتی در مواقعی که تمایزدهنده قوی است بتواند گرادیان های مفیدی را به مولد انتقال دهد . (Modified Minimax Loss)

۳ – استفاده از بهینه ساز های پیشرفته تر و تنظیم دقیق پارامتر ها و همچنین نرمال سازی گرادیان ها

۱-۲-۲ :

۱ – حفظ محتوای تصویر ورودی :

بر خلاف حالت معمولی که مولد تنها از نویز تصادفی برای تولید تصاویر استفاده می کند استفاده از تصویر در کنار نویز تضمین می کند که محتوای اصلی تصویر و الگو های اصلی حفظ شود در حالی که ویژگی های ظاهر ی مانند رنگ به سبک دامنه هدف تغییر می کند بنابراین تصویر ورودی امکان کنترل دقیق تر محتوای خروجی را فراهم می کند .

۲ – افزایش تنوع و جلوگیری از بیش برازش :

تغییرات تصادفی در ویژگی های ظاهری و سبک تصاویر در حالی که محتوا اصلی حفظ شده است باعث می شود نویز مانند عاملی برای ایجاد تنوع مانع بیش برازش مدل به یک ظاهر و سبک خاص در دامنه هدف شود و تعمیم پذیری مدل را برای کلاس های جدید بهبود بخشد و به نوعی با افزودن نویز data augmentation نیز صورت گرفته است .

۳ – تثبیت فرآیند یادگیری :

استفاده از تصویر ورودی همراه نویز و تابع زیان تصابه محتوا به تثبیت فرآیند یادگیری کمک می کند و این امر سبب می شود که تغییرات تصاویر تولید شده محدود به ویژگی های ظاهری باشد و سبب کاهش امکان ناپایداری و فروپاشی مد خواهد شد .

۴ – کاهش وابستگی به معماری وظیفه محور :

مولد در این حالت می تواند به طور مستقل تصاویر را از دامنه منبع به دامنه هدف تبدیل کند و سپس هر دسته بند بر اساس وظیفه خود می تواند روی تصاویر تولید شده آموزش ببیند که این جداسازی باعث می شود مدل انعطاف پذیر تر باشد .

۵ – قابلیت تفسیر بصری :

خروجی های مولد در این مدل به صورت مستقیم قابل مشاهده و تفسیر هستند که قابلیت بهبود ارزیابی و دیباگ مدل را فرآهم می آورد و تصاویر تولیدی به راحتی با ورودی اصلی قابل مقایسه هستند .

۳-۲-۱ :

نقش مولد : (Generator)

مولد وظیفه تبدیل تصاویر از دامنه منبع به تصاویری را دارد که به نظر می رسد از دامنه هدف نمونه برداری شده در حالی که محتوای اصلی تصویر منبع حفظ می شود . مولد با دریافت تصویر منبع و یک بردار نویز تصادفی تصویری تولید می کند که از نظر ظاهری شبیه به دامنه هدف است و مولد از طریق تعاملی که با تمایزدهنده دارد یاد میگیرد که تصاویر تولید شده را به گونه ای تنظیم کند که تمایزدهنده نتواند آن ها را از تصاویر واقعی دامنه هدف تشخیص دهد و با استفاده از زیان تشابه محتوا سعی می کند که محتوای اصلی تصویر حفظ شود .

تاثیر حذف مولد :

اگر مولد حذف شود کل فرآیند تطبیق دامنه در سطح پیکسل غیر ممکن می شود زیرا مدل قادر به تولید تصاویر تطبیق یافته از دامنه منبع به دامنه هدف نخواهد بود . در این حالت مدل نمی تواند شکاف بین دامنه مبع و هدف را پر کند و طبقه بند مستقیما روی تصاویر منبع آموزش داده می شود که به دلیل تفاوت ها بین دامنه ها عملکرد ضعیفی خواهد داشت .

نقش تمایزدهنده (Discriminator) :

تمایزدهنده وظیفه دارد بین تصاویر واقعی از دامنه هدف و تصاویر تولید شده توسط مولد تمایز قایل شود و تمایزدهنده آموزش می بیند تا احتمال اینکه یک تصویر از دامنه هدف باشد را تخمین بزند و تلاش می کند تا تصاویر واقعی را به درستی شناسایی کند که این فرآیند منجر می شود مولد تصاویری تولید کند که به دامنه هدف شبیه تر باشد .

تاثیر حذف تمایزدهنده :

اگر تمایزدهنده حذف شود مولد هیچ بازخوردی برای بهبود کیفیت تصاویر تولیدشده دریافت نمی کند زیرا تابع هرینه دامنه که وظیفه هدایت مولد به سمت تولید تصاویر شبیه به دامنه هدف دارد دیگر وجود

نخواهد داشت و منجر به تولید تصاویری می شود که از نظر ظاهری به دامنه هدف شبیه نیستند و بدون تمایزدهنده مدل نمی تواند تضمین کند که تصاویر تولید شده از توزیع دامنه هدف پیروی می کنند و این امر موجب کاهش دقت طبقه بند نیز خواهد شد و همچنین امکان تشدید مشکلاتی مانند فروپاشی مد نیز وجود دارد .

نقش طبقه بند (classifier) :

طبقه بند جهت انجام وظایف خاص مورد استفاده قرار می گیرد مانند طبقه بندی اشیا . طبقه بند روی تصاویر منبع و تصاویر تولید شده آموزش می بیند تا برچسب ها را پیش بینی کند و تابع زیان آن کمک می کند تا ویژگی های مرتبط با وظیفه را از تصاویر تطبیق یافته و منبع یاد بگیرد و در نتیجه به تثبیت فرآیند یادگیری کمک می کند و از جابجایی برچسب های کلاسی جلوگیری می کند .

تاثیر حذف طبقه بند :

اگر طبقه بند حذف شود مدل همچنان می تواند تصاویر تطبیق یافته تولید کند به لطف مولد و تمایزدهنده . اما قادر به انجام وظیفه خاص مانند طبقه بندی اشیا یا تخمین جهت گیری نخواهد بود . با حذف طبقه بند ممکن است تاثیرات منفی در تثبیت فرآیند آموزش مولد مشاهده شود زیرا طبقه بند به حفظ محتوای معنایی تصاویر کمک می کند با حداکثر کردن اطلاعات متقابل بین برچسب ها و تصاویر .

۴-۲-۱ :

۱-حفظ معنای تصویر منبع :

در تطبیق دامنه هدف اصلی این است که تصاویر دامنه منبع به گونه ای تغییر کنند که از نظر ظاهری شبیه به دامنه هدف شوند اما محتوای اصلی تصویر مانند شکل اشیا و ساختار و هویت کلاس حفظ شود و تابع زیان تشابه سعی می کند که مولد تغییرات را به ویژگی های ظاهری مانند رنگ یا پس زمینه محدود کند و از تغییر محتوایی و معنایی مانند تبدیل یک رقم به رقمی دیگر جلوگیری کند و این حفظ محتوا به طبقه بند اجازه می دهد تا برچسب صحیح از دامنه منبع را به تصاویر تولید شده منتقل کند تا عملکرد مناسبی در وظایف خاص داشته باشد .

۲-تثبیت فرآیند آموزش :

همانطور که در مقاله نیز مطرح شده است استفاده از تابع زیان تشابه محتوا و حفظ معنا به کاهش انحراف استاندارد در عملکرد مدل کمک می کند و این تابع زیان به عنوان یک regularizer عمل می کند که از تغییرات بیش از حد در تصویر تولید شده جلوگیری می کند که بدون حفظ محتوا ممکن است مولد تصاویری تولید کند که بیش از حد از تصویر منبع فاصله دارد و امکان فروپاشی مد را بوجود می آورد .

۳-افزایش تعمیم پذیری مدل :

حفظ محتوای و معنای تصویر منبع به مدل امکان می دهد تا به کلاس های دیده نشده نیز در طول آموزش تعمیم پیدا کند چون مولد یاد میگیرد تغییرات ظاهری را به طور کلی اعمال کند به جای وابستگی به کلاس های خاص و حفظ محتوا باعث می شود تغییرات مستقل از محتوای خاص تصویر هستند .

با توجه به توضیحاتی که ارایه شد اگر این مکانیزم حذف شود میتواند منجر به تغییرات غیر کنترل شده در محتوا مانند تولید ارقام جدید و تغییر ارقام و اشیا شود و تصاویری تولید شود که نه ظاهری شبیه به دامنه هدف دارند و نه محتوا مشابه و همچنین منجر به افزایش ناپایداری در فرآیند آموزش و افزایش انحراف استاندارد مدل در عملکرد شود و همچنین کاهش تعمیم پذیری مدل و کاهش دقت طبقه بند و

همچنین افزایش احتمال فروپاشی مد شود و به نوعی باعث می شود مولد تصاویری تولید کند که تنها به بخش کوچکی از توزیع دامنه هدف شباهت دارند به جای تنوع کامل در پوشش دامنه هدف .

۱-۲-۵ :

۱ – جلوگیری از جابجایی برچسب های کلاس :

اگر طبقه بند تنها روی تصاویر تولید شده آموزش ببیند ممکن است مولد تصاویری تولید کند که محتوای اصلی را تغییر دهند مانند تبدیل یک رقم به رقمی دیگر این می تواند باعث شود که برچسب های منبع با محتوای تصاویر تولید شده مطابقت نداشته باشند که به جابجایی برچسب های کلاس منجر می شود و آموزش همزمان روی تصاویر منبع به طبقه بند کمک می کند تا محتوای اصلی و برچسب های مرتبط با آن را به طور مستقیم یاد بگیرد که این امر به حفظ تطابق بین برچسب ها و محتوا کمک می کند .

آموزش تنها روی تصاویر تولید شده می تواند به عملکرد مشابهی منجر شود اما به دلیل ناپایداری نیاز به چندین اجرا با مقداردهی اولیه متفاوت دارد اما آموزش روی هر دو نوع تصویر این مشکل را برطرف می کند .

۲ – افزایش پایداری یادگیری (کاهش انحراف استاندارد مدل) :

آموزش همزمان روی تصاویر منبع و تولید شده می تواند باعث کاهش حساسیت مدل به تغییرات تصادفی در تصاویر تولید شده می شود و همچنین باعث جلوگیری از بیش برازش مدل به تصاویر تولیدشده خواهد شد .

۳ – تاکید بر حفظ معنا و افزایش اطلاعات متقابل در فرآیند تطبیق

در نتیجه ی موارد ذکر شده می توان گفت آموزش همزمان روی تصاویر اصلی و تولید شده با کاهش واریانس در عملکرد مدل (استفاده از تصاویر منبع برای جلوگیری از تغییرات بیش از حد در تصاویر تولید شده) و جلوگیری از فروپاشی مد (سازگاری محتوا با تصاویر تولید شده و تنوع بخشیدن به تصاویر تولید شده) و کاهش حساسیت مدل به مقداردهی اولیه با استفاده از آموزش همزمان روی تصاویر منبع که داده های پایدار و قایل اعتمادی هستند موجب پایداری هر چه بیشتر کل فرآیند یادگیری مدل خواهد شد .

۱-۲-۶ :

مدل PixelDA فرض می‌کند تفاوت‌های بین دامنه‌ها عمدتاً در سطح پایین (low-level) است، مانند نویز، رزولوشن، نورپردازی، یا رنگ، و نه در سطح بالا (high-level) مانند نوع اشیا یا تغییرات هندسی. این فرض به این معناست که محتوای معنایی (مانند شکل اشیا یا کلاس‌ها) بین دامنه‌های منبع و هدف یکسان یا بسیار مشابه است.

این تابع هزینه برای حفظ محتوای پیش‌زمینه (مانند شکل اشیا) طراحی شده است و تغییرات را به ویژگی‌های ظاهری محدود می‌کند. این مکانیزم برای دامنه‌هایی با محتوای معنایی یکسان به خوبی کار می‌کند، اما اگر محتوای معنایی بین دامنه‌ها متفاوت باشد (مانند اشیای متفاوت یا زبان‌های متفاوت)، این زیان ممکن است محدودیت ایجاد کند.

این مدل از برچسب های منبع برای آموزش طبقه بند استفاده می کند و فرض می کند که این برچسب ها برای دامنه هدف نیز معتبر هستند ولی اگر دامنه ها تفاوت معنایی عمیقی داشته باشند مانند کلاس های کاملا متفاوت این فرض دیگر برقرار نیست .

۱-۳. بخش عملی :

۱-۳-۱. پیش پردازش داده ها :

ابتدا دو مجموعه داده MNIST و MNIST-M را در نوت بوک کگل آپلود کردیم سپس با توجه به توضیحات ارایه شده در صورت سوال ۵ نمونه متناظر از هر دو مجموعه را به صورت ردیفی در کنار هم نمایش می دهیم تا تفاوت های ظاهری آن ها و حفظ محتوا مشاهده شود :

```
import pickle
pkl_mnist_file = '/kaggle/input/mnistmm/dataset/mnist.pkl'
pkl_mnistm_file = '/kaggle/input/mnistmm/dataset/mnistm.pkl'
with open(pkl_mnist_file, 'rb') as f:
    data_mnist = pickle.load(f)
with open(pkl_mnistm_file, 'rb') as f:
    data_mnistm = pickle.load(f)
```

```
import matplotlib.pyplot as plt
import numpy as np

mnist_images = data_mnist[b'images']
mnist_labels = data_mnist[b'labels']
mnistm_images = data_mnistm[b'images']
mnistm_labels = data_mnistm[b'labels']
indices = range(5)
fig, axes = plt.subplots(1, 10, figsize=(20, 2))
for i, idx in enumerate(indices):
    #grayscale
    axes[2*i].imshow(mnist_images[idx], cmap='gray')
    axes[2*i].set_title(f'MNIST: {mnist_labels[idx]}')
    axes[2*i].axis('off')
    #RGB
    axes[2*i + 1].imshow(mnistm_images[idx])
    axes[2*i + 1].set_title(f'MNIST-M: {mnistm_labels[idx]}')
    axes[2*i + 1].axis('off')
plt.tight_layout()
plt.show()
```

.

شکل 1. ۵ نمونه از تصاویر موجود در دیتاست های MNIST و MNIST-M

سپس با تبدیل تصاویر MNIST به سه کانال و نرمال سازی تصاویر و resize و سپس قرار دادن داده ها در Dataloader های جداگانه و همچنین تقسیم داده ها به داده های آموزش و تست به گونه ای که در نهایت چهار Dataloader برای دو مجموعه داده MNIST و MNIST-M و batch_size برابر ۳۲ تشکیل شد .

```
from sklearn.model_selection import train_test_split
mnist_images_rgb = np.repeat(mnist_images[:, :, :, np.newaxis], 3,
axis=3)

num_samples = len(mnist_images)
indices = np.arange(num_samples)
train_indices, test_indices = train_test_split(indices,
test_size=0.2, random_state=42)

mnist_train_images = mnist_images_rgb[train_indices]
mnist_train_labels = mnist_labels[train_indices]
mnist_test_images = mnist_images_rgb[test_indices]
mnist_test_labels = mnist_labels[test_indices]

mnistm_train_images = mnistm_images[train_indices]
mnistm_train_labels = mnistm_labels[train_indices]
mnistm_test_images = mnistm_images[test_indices]
mnistm_test_labels = mnistm_labels[test_indices]

transform = Compose([
    ToTensor(),
    Resize((32, 32)),
    Lambda(lambda x: x * 2 - 1)
])

def preprocess_images(images):
    processed_images = []
    for img in images:
        img = img.astype(np.uint8)
        img = Image.fromarray(img)
        img = transform(img)
        processed_images.append(img)
    return torch.stack(processed_images)

mnist_train_images_processed = preprocess_images(mnist_train_images)
mnist_test_images_processed = preprocess_images(mnist_test_images)
```

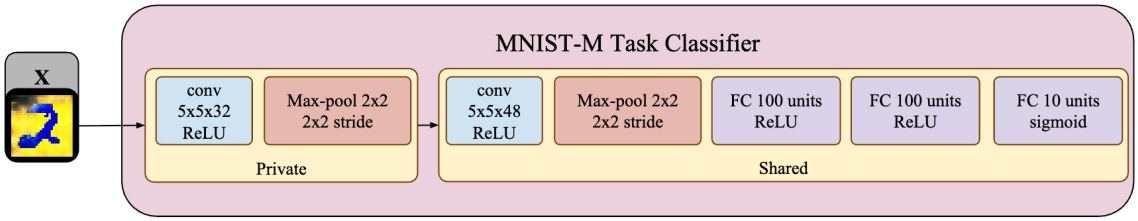
```
mnistm_train_images_processed =
preprocess_images(mnistm_train_images)
mnistm_test_images_processed = preprocess_images(mnistm_test_images)

mnist_train_labels_tensor =
torch.from_numpy(mnist_train_labels).long()
mnist_test_labels_tensor = torch.from_numpy(mnist_test_labels).long()
mnistm_train_labels_tensor =
torch.from_numpy(mnistm_train_labels).long()
mnistm_test_labels_tensor =
torch.from_numpy(mnistm_test_labels).long()

mnist_train_dataset = TensorDataset(mnist_train_images_processed,
mnist_train_labels_tensor)
mnist_test_dataset = TensorDataset(mnist_test_images_processed,
mnist_test_labels_tensor)
mnistm_train_dataset = TensorDataset(mnistm_train_images_processed,
mnistm_train_labels_tensor)
mnistm_test_dataset = TensorDataset(mnistm_test_images_processed,
mnistm_test_labels_tensor)

batch_size = 32
mnist_train_loader = DataLoader(mnist_train_dataset,
batch_size=batch_size, shuffle=True)
mnist_test_loader = DataLoader(mnist_test_dataset,
batch_size=batch_size, shuffle=False)
mnistm_train_loader = DataLoader(mnistm_train_dataset,
batch_size=batch_size, shuffle=True)
mnistm_test_loader = DataLoader(mnistm_test_dataset,
batch_size=batch_size, shuffle=False)

print(f"MNIST train images shape:
{mnist_train_images_processed.shape}")
print(f"MNIST train labels shape: {mnist_train_labels_tensor.shape}")
print(f"MNIST test images shape:
{mnist_test_images_processed.shape}")
print(f"MNIST test labels shape: {mnist_test_labels_tensor.shape}")
print(f"MNIST-M train images shape:
{mnistm_train_images_processed.shape}")
print(f"MNIST-M train labels shape:
{mnistm_train_labels_tensor.shape}")
print(f"MNIST-M test images shape:
{mnistm_test_images_processed.shape}")
print(f"MNIST-M test labels shape:
{mnistm_test_labels_tensor.shape}")
```

که با در کنار هم قرار دادن نمونه های هر دسته خواهیم داشت :

```
MNIST train images shape: torch.Size([56000, 3, 32, 32])
MNIST train labels shape: torch.Size([56000])
MNIST test images shape: torch.Size([14000, 3, 32, 32])
MNIST test labels shape: torch.Size([14000])
MNIST-M train images shape: torch.Size([56000, 3, 32, 32])
MNIST-M train labels shape: torch.Size([56000])
MNIST-M test images shape: torch.Size([14000, 3, 32, 32])
MNIST-M test labels shape: torch.Size([14000])
```

۱-۳-۲. آزمون مدل پایه و مشاهده Domain Gap :

یک طبقه بند با معماری مقاله بر روی داده های MNIST آموزش دادیم و سپس مدل آموزش دیده را روی داده های تست MNIST و کل داده های MNIST-M ارزیابی کردیم .



شکل 2. معماری طبقه بند معرفی شده در مقاله

```python
class MNISTMClassifier(nn.Module):
    def __init__(self):
        super(MNISTMClassifier, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 48, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(48 * 8 * 8, 100),
            nn.ReLU(),
            #nn.Dropout(0.5),
            nn.Linear(100, 100),
            nn.ReLU(),
            #nn.Dropout(0.5),
            nn.Linear(100, 10),
```

```
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Epoch [1/10], Loss: 0.3593
Epoch [2/10], Loss: 0.1429
Epoch [3/10], Loss: 0.1105
Epoch [4/10], Loss: 0.0940
Epoch [5/10], Loss: 0.0787
Epoch [6/10], Loss: 0.0710
Epoch [7/10], Loss: 0.0660
Epoch [8/10], Loss: 0.0592
Epoch [9/10], Loss: 0.0595
Epoch [10/10], Loss: 0.0529
Accuracy on MNIST Test: 98.98%
Accuracy on MNIST-M Full: 56.63%

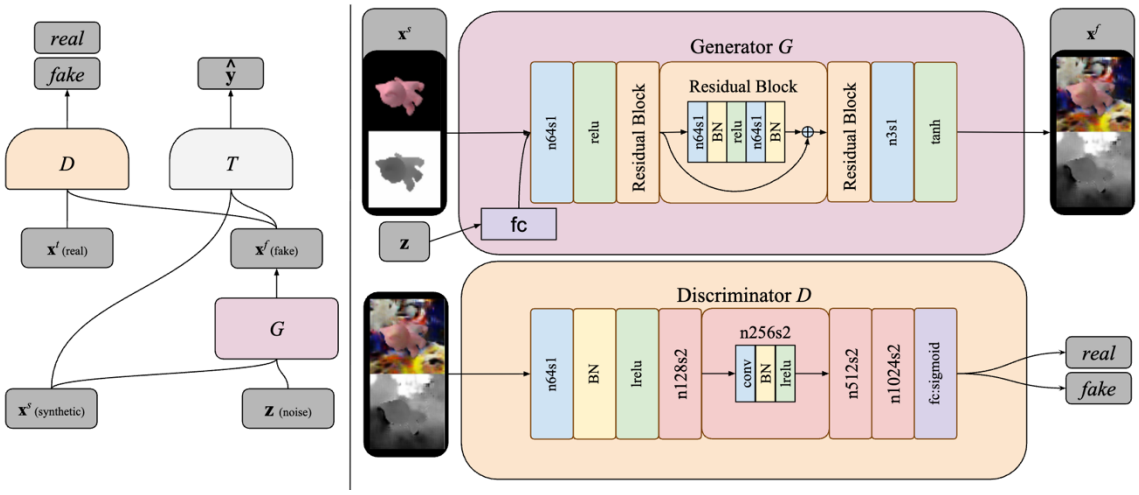
Domain Gap Analysis:
Accuracy difference (MNIST Test - MNIST-M Full): **42.35%**

شکل 3. نمونه ای از تصاویر **MNIST-M** که به درستی توسط طبقه بند دسته بندی نشده اند

با توجه به اینکه کلاسفایر روی داده های MNIST آموزش دیده است و داده های MNIST-M دارای سبک متفاوتی از نظر ظاهری هستند کلاسیفایر به تنهایی نمی تواند بر روی ویژگی های معنایی موجود در MNIST تاکید کند و تغییر ویژگی های ظاهری موجب کاهش دقت در ارزیابی بر روی داده های MNIST-M شده است .

۱-۳-۳. پیاده سازی معماری مدل :

در این بخش پس از آماده سازی dataloader ها در بخش های قبل شروع به پیاده سازی معماری مدل در سه بخش generator و discriminator و classifier می کنیم .



شکل 4. معماری معرفی شده برای بلوک های مولد و تمایزدهنده

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, stride=1,
padding=1, bias=False),
            nn.BatchNorm2d(channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(channels, channels, kernel_size=3, stride=1,
padding=1, bias=False),
            nn.BatchNorm2d(channels)
        )

    def forward(self, x):
        return x + self.block(x)

class Generator(nn.Module):
    def __init__(self, input_channels=3, noise_dim=100,
num_filters=64):
        super(Generator, self).__init__()
        self.noise_fc = nn.Linear(noise_dim, 1 * 32 * 32)
        self.initial_conv = nn.Sequential(
```

```
            nn.Conv2d(input_channels + 1, num_filters, kernel_size=3,
stride=1, padding=1, bias=False),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True)
        )
        self.res_blocks = nn.Sequential(
            *[ResidualBlock(num_filters) for _ in range(6)]
        )
        self.output_conv = nn.Conv2d(num_filters, 3, kernel_size=3,
stride=1, padding=1)
        self.output_act = nn.Tanh()

    def forward(self, x, z):
        z = self.noise_fc(z).view(-1, 1, 32, 32)
        x = torch.cat([x, z], dim=1)
        x = self.initial_conv(x)
        x = self.res_blocks(x)
        x = self.output_conv(x)
        x = self.output_act(x)
        return x

class Discriminator(nn.Module):
    def __init__(self, input_channels=3):
        super(Discriminator, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(input_channels, 64, kernel_size=1, stride=1,
padding=0),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1),
            nn.Conv2d(64, 128, kernel_size=2, stride=2, padding=0),
# (B, 128, 16, 16)
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1),
            nn.Conv2d(128, 256, kernel_size=2, stride=2, padding=0),
# (B, 256, 8, 8)
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1),
            nn.Conv2d(256, 512, kernel_size=2, stride=2, padding=0),
# (B, 512, 4, 4)
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 4 * 4, 1),
            nn.Sigmoid()
        )
class TaskClassifier(nn.Module):
```

```
    def __init__(self, input_channels=3):
        super(TaskClassifier, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(input_channels, 32, kernel_size=5, stride=1,
padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 48, kernel_size=5, stride=1, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(48 * 8 * 8, 100),
            nn.ReLU(inplace=True),
            #nn.Dropout(0.5),
            nn.Linear(100, 100),
            nn.ReLU(inplace=True),
            #nn.Dropout(0.5),
            nn.Linear(100, 10)
        )
```

۱-۳-۴. پیاده سازی توابع هزینه :

با توجه به توضیحات ارایه شده دو نوع تابع هزینه adversarial و task را پیاده سازی کرده و هایپرپارامتر های مورد نیاز آن را درنظر گرفتیم .

```
  #Hyperparameters
  alpha = 0.13
  beta = 0.01
  g_loss_weight = 0.011
```

```
generator = Generator(input_channels=3, noise_dim=100,
num_filters=64).to(device)
discriminator = Discriminator(input_channels=3).to(device)
task_classifier = TaskClassifier(input_channels=3).to(device)


def init_weights(m):
    if isinstance(m, (nn.Conv2d, nn.Linear)):
        nn.init.normal_(m.weight, mean=0.0, std=0.02)
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)
```

```
generator.apply(init_weights)
discriminator.apply(init_weights)
task_classifier.apply(init_weights)

#Optimizers
g_optimizer = optim.Adam(generator.parameters(), lr=1e-3, betas=(0.5,
0.999), weight_decay=1e-5)
d_optimizer = optim.Adam(discriminator.parameters(), lr=1e-3,
betas=(0.5, 0.999), weight_decay=1e-5)
t_optimizer = optim.Adam(task_classifier.parameters(), lr=1e-3,
betas=(0.5, 0.999), weight_decay=1e-5)

# Loss functions
#Adversarial loss (L_d)
criterion_d = nn.BCELoss()
#Task-specific loss (L_t)
criterion_t = nn.CrossEntropyLoss()

#Hyperparameters
alpha = 0.13
beta = 0.01
g_loss_weight = 0.011

def train_pixeldan(num_epochs=10):
    generator.train()
    discriminator.train()
    task_classifier.train()

    for epoch in range(num_epochs):
        for (source_images, source_labels), (target_images, _) in
zip(mnist_train_loader, mnistm_train_loader):
            source_images, source_labels = source_images.to(device),
source_labels.to(device)
            target_images = target_images.to(device)
            batch_size = source_images.size(0)

            #labels for discriminator
            real_labels = torch.ones(batch_size, 1).to(device)
            fake_labels = torch.zeros(batch_size, 1).to(device)

            #update Discriminator and Task Classifier (theta_D,
theta_T)
            d_optimizer.zero_grad()
            t_optimizer.zero_grad()

            #Discriminator on real target images (E[log D(x^t)])
            d_real = discriminator(target_images)
```

```
            d_loss_real = criterion_d(d_real, real_labels)

            #discriminator on fake images (E[log(1 - D(G(x^s, z)))])
            noise = torch.FloatTensor(batch_size, 100).uniform_(-1,
1).to(device)
            fake_images = generator(source_images, noise)
            d_fake = discriminator(fake_images.detach())
            d_loss_fake = criterion_d(d_fake, fake_labels)

            #total discriminator loss: alpha * L_d
            d_loss = alpha * (d_loss_real + d_loss_fake)
            d_loss.backward()
            d_optimizer.step()

            #Task Classifier on source and fake images (L_t)
            t_source = task_classifier(source_images)  # T(x^s)
            t_fake = task_classifier(fake_images.detach())  #
T(G(x^s, z))
            t_loss_source = criterion_t(t_source, source_labels)

            t_loss_fake = criterion_t(t_fake, source_labels)
            t_loss = t_loss_source + t_loss_fake  # E[-y^s log
T(x^s)] + E[-y^s log T(G(x^s, z))]
            t_loss.backward()
            t_optimizer.step()

            #Update Generator (theta_G)
            g_optimizer.zero_grad()
            d_fake = discriminator(fake_images)  # D(G(x^s, z))
            g_loss_d = criterion_d(d_fake, real_labels)  # E[log
D(G(x^s, z))] (fool discriminator)
            t_fake = task_classifier(fake_images)  # T(G(x^s, z))
            g_loss_t = criterion_t(t_fake, source_labels)  # E[-y^s
log T(G(x^s, z))]
            g_loss = g_loss_weight * g_loss_d + beta * g_loss_t  #
g_loss_weight * L_d + beta * L_t
            g_loss.backward()
            g_optimizer.step()
```

۱-۳-۵. آموزش مدل :

در این قسمت به آموزش مدل می پردازیم با آموزش مولد بر روی دامنه ی منبع MNIST و تولید تصاویر با ظاهر دامنه هدف MNIST-M با حفظ محتوای عددی و آموزش طبقه بند روی تصاویر دامنه منبع با استفاده از تصاویر اصلی و تصاویر فیک تولید شده از آن ها و در طول آموزش آن از برچسب های دامنه هدف استفاده نشده است و در هر تکرار مقادیر هزینه مربوط به هر بلوک و همچنین دقت طبقه بند روی داده های آموزش و تست ذخیره و اندازه گیری شده است .

در ادامه کد های مربوط به پیاده سازی بلوک های اصلی و فرآیند آموزش کل مدل همره پارامتر های آن و ذخیره مقادیر loss و دقت آمده است :

```
#Residual Block
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, stride=1,
padding=1, bias=False),
            nn.BatchNorm2d(channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(channels, channels, kernel_size=3, stride=1,
padding=1, bias=False),
            nn.BatchNorm2d(channels)
        )

class Generator(nn.Module):
    def __init__(self, input_channels=3, noise_dim=100,
num_filters=64):
        super(Generator, self).__init__()
        self.noise_fc = nn.Linear(noise_dim, 1 * 32 * 32)
        self.initial_conv = nn.Sequential(
            nn.Conv2d(input_channels + 1, num_filters, kernel_size=3,
stride=1, padding=1, bias=False),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True)
        )
        self.res_blocks = nn.Sequential(
            *[ResidualBlock(num_filters) for _ in range(6)]
        )
        self.output_conv = nn.Conv2d(num_filters, 3, kernel_size=3,
stride=1, padding=1)
        self.output_act = nn.Tanh()

    def forward(self, x, z):
        z = self.noise_fc(z).view(-1, 1, 32, 32)
        x = torch.cat([x, z], dim=1)
        x = self.initial_conv(x)
        x = self.res_blocks(x)
```

```
        x = self.output_conv(x)
        x = self.output_act(x)
        return x

        #Discriminator (D)
class Discriminator(nn.Module):
    def __init__(self, input_channels=3):
        super(Discriminator, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(input_channels, 64, kernel_size=1, stride=1,
padding=0),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1),
            nn.Conv2d(64, 128, kernel_size=2, stride=2, padding=0),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1),
            nn.Conv2d(128, 256, kernel_size=2, stride=2, padding=0),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1),
            nn.Conv2d(256, 512, kernel_size=2, stride=2, padding=0),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.1)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 4 * 4, 1),
            nn.Sigmoid()
        )
    def forward(self, x):
        for layer in self.features:
            if isinstance(layer, nn.Conv2d):
                x = layer(x)
                x = x + torch.randn_like(x) * 0.2
            else:
                x = layer(x)
        x = self.classifier(x)
        return x
#Task Classifier (T)
class TaskClassifier(nn.Module):
    def __init__(self, input_channels=3):
        super(TaskClassifier, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(input_channels, 32, kernel_size=5, stride=1,
padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 48, kernel_size=5, stride=1, padding=2),
            nn.ReLU(inplace=True),
```

```
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(48 * 8 * 8, 100),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(100, 100),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(100, 10)
        )
    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

def evaluate_classifier(classifier, loader, device):
    classifier.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)
            outputs = classifier(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    classifier.train()
    return accuracy

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
generator = Generator(input_channels=3, noise_dim=100,
num_filters=64).to(device)
discriminator = Discriminator(input_channels=3).to(device)
task_classifier = TaskClassifier(input_channels=3).to(device)

def init_weights(m):
    if isinstance(m, (nn.Conv2d, nn.Linear)):
        nn.init.normal_(m.weight, mean=0.0, std=0.02)
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)

generator.apply(init_weights)
discriminator.apply(init_weights)
task_classifier.apply(init_weights)
#optimizers
```

```
g_optimizer = optim.Adam(generator.parameters(), lr=1e-3, betas=(0.5,
0.999), weight_decay=1e-5)
d_optimizer = optim.Adam(discriminator.parameters(), lr=1e-3,
betas=(0.5, 0.999), weight_decay=1e-5)
t_optimizer = optim.Adam(task_classifier.parameters(), lr=1e-3,
betas=(0.5, 0.999), weight_decay=1e-5)
#Loss
criterion_d = nn.BCELoss()  # Adversarial loss (L_d)
criterion_t = nn.CrossEntropyLoss()  # Task-specific loss (L_t)

#discriminator loss weight
alpha = 0.13
#task loss weight in G step
beta = 0.01
#generator adversarial loss weight
g_loss_weight = 0.011

#checkpoints
checkpoint_dir = "checkpoints"
os.makedirs(checkpoint_dir, exist_ok=True)

#history
history = {
    "d_loss": [],
    "g_loss": [],
    "t_loss": [],
    "train_acc_mnist": [],
    "test_acc_mnist": [],
    "test_acc_mnistm": []
}

def train_pixeldan(num_epochs=10, resume_epoch=0):
    generator.train()
    discriminator.train()
    task_classifier.train()

    start_epoch = resume_epoch
    if resume_epoch > 0:
        #checkpoints

generator.load_state_dict(torch.load(os.path.join(checkpoint_dir,
f"generator_epoch_{resume_epoch}.pth")))

discriminator.load_state_dict(torch.load(os.path.join(checkpoint_dir,
f"discriminator_epoch_{resume_epoch}.pth")))

task_classifier.load_state_dict(torch.load(os.path.join(checkpoint_di
r, f"task_classifier_epoch_{resume_epoch}.pth")))
```

```
    for epoch in range(start_epoch, num_epochs):
        epoch_d_loss = 0.0
        epoch_g_loss = 0.0
        epoch_t_loss = 0.0
        num_batches = 0
        for (source_images, source_labels), (target_images, _) in
zip(mnist_train_loader, mnistm_train_loader):
            source_images, source_labels = source_images.to(device),
source_labels.to(device)
            target_images = target_images.to(device)
            batch_size = source_images.size(0)
                        #labels for discriminator
            real_labels = torch.ones(batch_size, 1).to(device)
            fake_labels = torch.zeros(batch_size, 1).to(device)
            #update Discriminator and Task Classifier (theta_D,
theta_T)
            d_optimizer.zero_grad()
            t_optimizer.zero_grad()
            #discriminator on real target images (E[log D(x^t)])
            d_real = discriminator(target_images)
            d_loss_real = criterion_d(d_real, real_labels)
            #discriminator on fake images (E[log(1 - D(G(x^s, z)))])
            noise = torch.FloatTensor(batch_size, 100).uniform_(-1,
1).to(device)
            fake_images = generator(source_images, noise)
            d_fake = discriminator(fake_images.detach())
            d_loss_fake = criterion_d(d_fake, fake_labels)

           #total discriminator loss: alpha * L_d
            d_loss = alpha * (d_loss_real + d_loss_fake)
            d_loss.backward()
            d_optimizer.step()
            #task Classifier on source and fake images (L_t)
            t_source = task_classifier(source_images)  # T(x^s)
            t_fake = task_classifier(fake_images.detach())  #
T(G(x^s, z))
            t_loss_source = criterion_t(t_source, source_labels)
            t_loss_fake = criterion_t(t_fake, source_labels)
            t_loss = t_loss_source + t_loss_fake  # E[-y^s log
T(x^s)] + E[-y^s log T(G(x^s, z))]
            t_loss.backward()
            t_optimizer.step()
            #update Generator (theta_G)
            g_optimizer.zero_grad()
            d_fake = discriminator(fake_images)  # D(G(x^s, z))
            g_loss_d = criterion_d(d_fake, real_labels)  # E[log
D(G(x^s, z))] (fool discriminator)
```

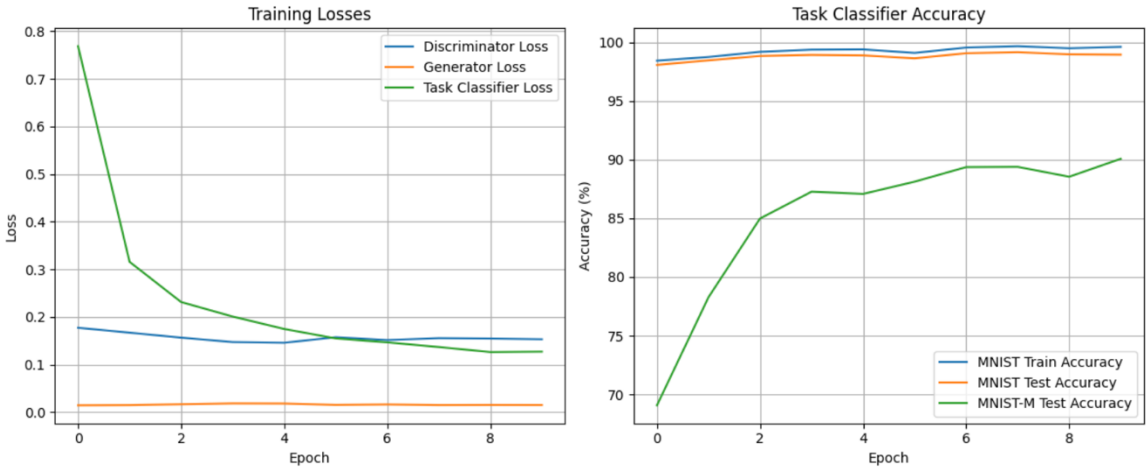
```
            t_fake = task_classifier(fake_images)  # T(G(x^s, z))
            g_loss_t = criterion_t(t_fake, source_labels)  # E[-y^s
log T(G(x^s, z))]
            g_loss = g_loss_weight * g_loss_d + beta * g_loss_t  #
g_loss_weight * L_d + beta * L_t
            g_loss.backward()
            g_optimizer.step()
            epoch_d_loss += d_loss.item()
            epoch_g_loss += g_loss.item()
            epoch_t_loss += t_loss.item()
            num_batches += 1
        epoch_d_loss /= num_batches
        epoch_g_loss /= num_batches
        epoch_t_loss /= num_batches
        #Evaluate
        train_acc_mnist = evaluate_classifier(task_classifier,
mnist_train_loader, device)
        test_acc_mnist = evaluate_classifier(task_classifier,
mnist_test_loader, device)
        test_acc_mnistm = evaluate_classifier(task_classifier,
mnistm_test_loader, device)
        #history
        history["d_loss"].append(epoch_d_loss)
        history["g_loss"].append(epoch_g_loss)
        history["t_loss"].append(epoch_t_loss)
        history["train_acc_mnist"].append(train_acc_mnist)
        history["test_acc_mnist"].append(test_acc_mnist)
        history["test_acc_mnistm"].append(test_acc_mnistm)
```

فرآیند آموزش :

```
Epoch [1/10], D Loss: 0.1774, G Loss: 0.0144, T Loss: 0.7687, Train
Acc MNIST: 98.42%, Test Acc MNIST: 98.06%, Test Acc MNIST-M: 69.06%
Epoch [2/10], D Loss: 0.1670, G Loss: 0.0149, T Loss: 0.3158, Train
Acc MNIST: 98.74%, Test Acc MNIST: 98.45%, Test Acc MNIST-M: 78.27%
Epoch [3/10], D Loss: 0.1566, G Loss: 0.0167, T Loss: 0.2314, Train
Acc MNIST: 99.17%, Test Acc MNIST: 98.82%, Test Acc MNIST-M: 84.98%
Epoch [4/10], D Loss: 0.1473, G Loss: 0.0184, T Loss: 0.2009, Train
Acc MNIST: 99.36%, Test Acc MNIST: 98.91%, Test Acc MNIST-M: 87.26%
Epoch [5/10], D Loss: 0.1458, G Loss: 0.0182, T Loss: 0.1749, Train
Acc MNIST: 99.38%, Test Acc MNIST: 98.87%, Test Acc MNIST-M: 87.07%
Epoch [6/10], D Loss: 0.1575, G Loss: 0.0153, T Loss: 0.1550, Train
Acc MNIST: 99.08%, Test Acc MNIST: 98.62%, Test Acc MNIST-M: 88.11%
Epoch [7/10], D Loss: 0.1515, G Loss: 0.0162, T Loss: 0.1467, Train
Acc MNIST: 99.54%, Test Acc MNIST: 99.06%, Test Acc MNIST-M: 89.36%
Epoch [8/10], D Loss: 0.1555, G Loss: 0.0150, T Loss: 0.1368, Train
Acc MNIST: 99.66%, Test Acc MNIST: 99.14%, Test Acc MNIST-M: 89.38%
```

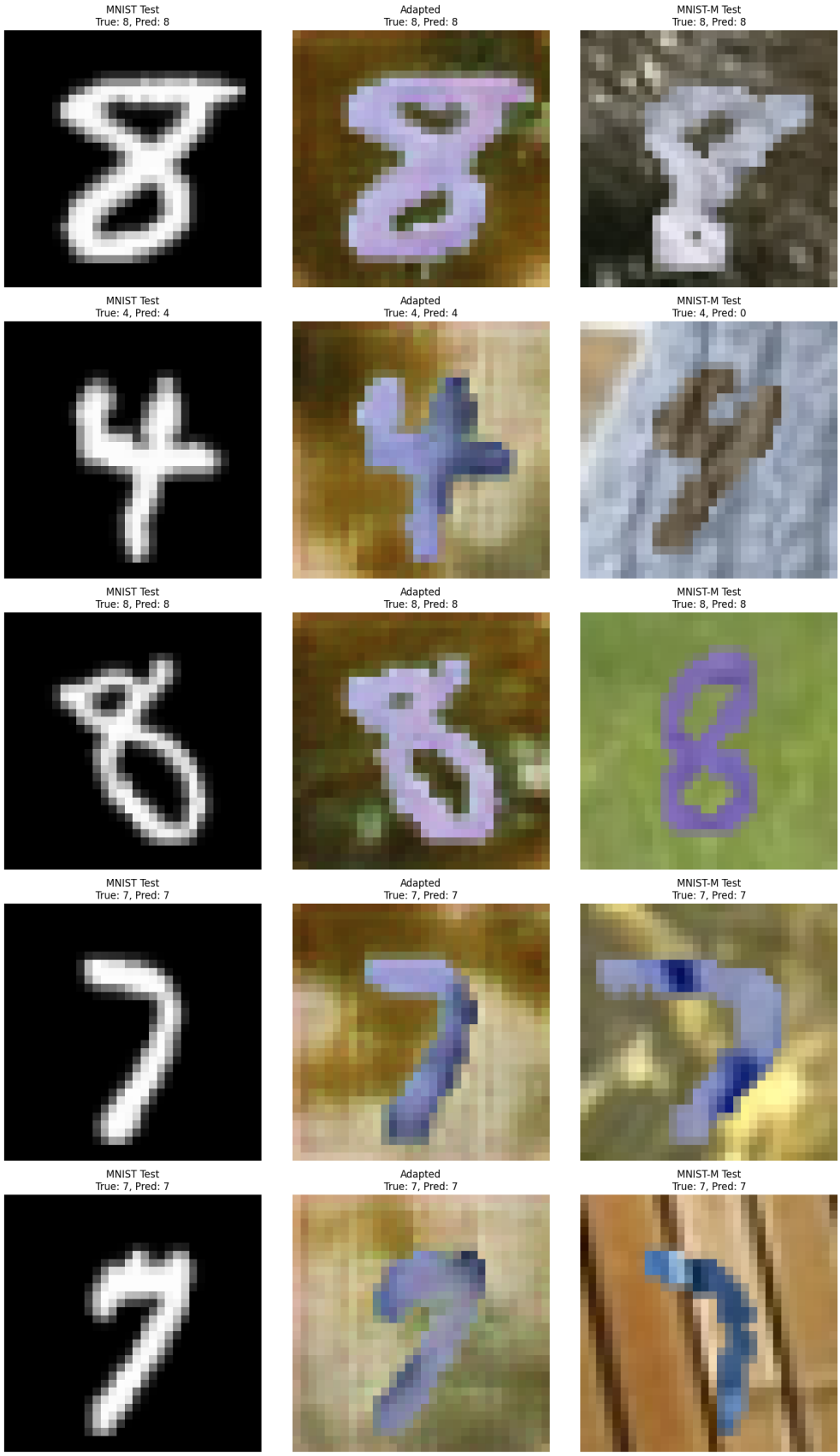
```
Epoch [9/10], D Loss: 0.1547, G Loss: 0.0152, T Loss: 0.1263, Train
Acc MNIST: 99.48%, Test Acc MNIST: 98.96%, Test Acc MNIST-M: 88.54%
Epoch [10/10], D Loss: 0.1532, G Loss: 0.0150, T Loss: 0.1272, Train
Acc MNIST: 99.60%, Test Acc MNIST: 98.94%, Test Acc MNIST-M: 90.06%
```

۱-۳-۶. نمایش نتایج :

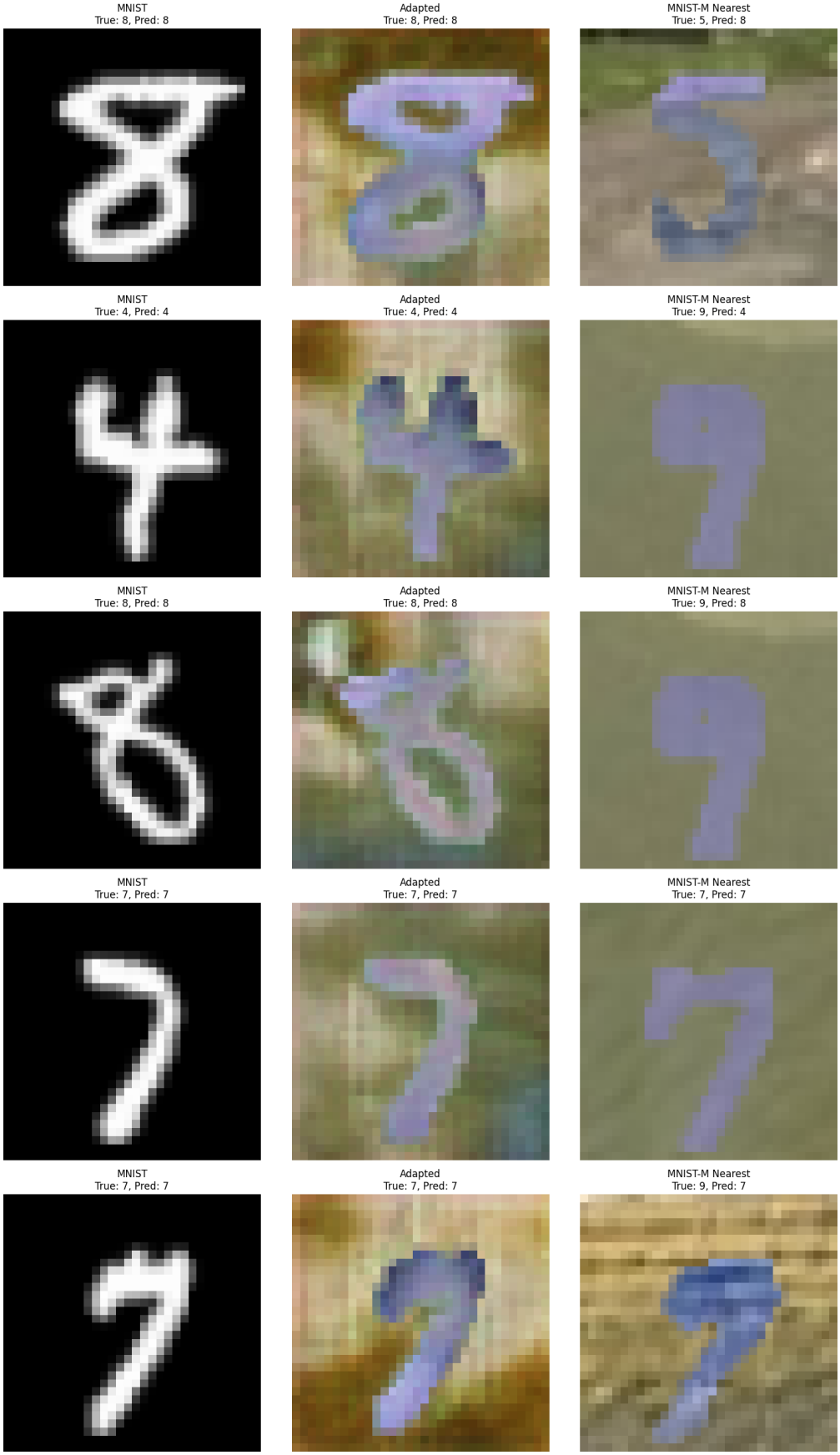


شکل ۵. نمودار مقادیر **loss** و دقت مربوط به هر بلوک بر حسب ایپاک

همانطور که مشاهده می شود با جلورفتن در ایپاک ها دقت دسته بند در مواجه با داده های تست **MNIST-M** افزایش می یابد و در ایپاک اخر مدل به دقتی حدود ۹۰ درصد در انجام این وظیفه خاص دست می یابد .



شکل ۶. بررسی عملکرد مدل با ۵ نمونه تصویر با مقایسه تصاویر اصلی و تولید شده توسط مولد و تصاویر موجود در **MNIST-M**



شکل ۷. بررسی عملکرد مدل با ۵ نمونه تصویر با مقایسه تصاویر اصلی و تولید شده توسط مولد و نزدیک ترین از نظر فاصله در تصاویر موجود در **MNIST-M**

پرسش ۲ – عنوان پرسش دوم به فارسی

۲-۱. عنوان بخش اول

متن نمونه

.