به نام خدا

دانشگاه تهران - دانشگاه صنعتی خواجه نصیرالدین طوسی تهران

دانشکده مهندسی برق و کامپیوتر

اتوانکودرها برای طبقه‌بندی

نام و نام خانوادگی	محمدجواد احمدی
شمارهٔ دانشجویی	۴۰۱۰۰۰۸۶

فهرست مطالب

۳ پاسخ پرسش سوم ۴
۱.۳ آشنایی و کار با دیتاست (پیش‌پردازش) ۴
۲.۳ شبکهٔ Auto-Encoder ۹
۳.۳ طبقه‌بندی ۱۴
۴.۳ بررسی با فراپارامترهای مختلف و یافتن حالاتی بهتر ۲۲

۴ پاسخ پرسش سوم – راه‌حل دوم ۳۱

فهرست تصاویر

۱ نمودار تعداد داده‌ها به‌ازای هر گروه برای آموزش ۵
۲ نمایش تصادفی پنج داده ۷
۳ نمایش تصادفی پنج داده (نرمال‌شده) ۸
۴ نتیجهٔ تابع اتلاف شبکهٔ اتوانکودر (فراپارامترهای آزمایش اول) ۱۳
۵ نتیجهٔ تابع اتلاف و دقت شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش اول) ۱۹
۶ نتیجهٔ ماتریس درهم‌ریختگی شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش اول) ۲۰
۷ نتیجهٔ تست تصاویر شخصی دست‌نویس با ساختار پیاده‌سازی‌شده (فراپارامترهای آزمایش اول) ۲۲
۸ نتیجهٔ تابع اتلاف شبکهٔ اتوانکودر (فراپارامترهای آزمایش دوم) ۲۴
۹ نتیجهٔ تابع اتلاف و دقت شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش دوم) ۲۵
۱۰ نتیجهٔ ماتریس درهم‌ریختگی شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش دوم) ۲۶
۱۱ نتیجهٔ تست تصاویر شخصی دست‌نویس با ساختار پیاده‌سازی‌شده (فراپارامترهای آزمایش دوم) ۲۶
۱۲ نتیجهٔ تابع اتلاف شبکهٔ اتوانکودر (فراپارامترهای آزمایش سوم) ۲۸
۱۳ نتیجهٔ تابع اتلاف و دقت شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش سوم) ۲۹
۱۴ نتیجهٔ ماتریس درهم‌ریختگی شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش سوم) ۳۰
۱۵ نتیجهٔ تست تصاویر شخصی دست‌نویس با ساختار پیاده‌سازی‌شده (فراپارامترهای آزمایش سوم) ۳۰

فهرست جداول

۱ فراپارامترهای آزمایش اول ۱۰
۲ فراپارامترهای آزمایش دوم ۲۲
۳ فراپارامترهای آزمایش سوم ۲۵

پرسش ۳. Auto-Encoders for Classification

۳ پاسخ پرسش سوم

توضیح پوشهٔ کدهای Auto-Encoders for Classification

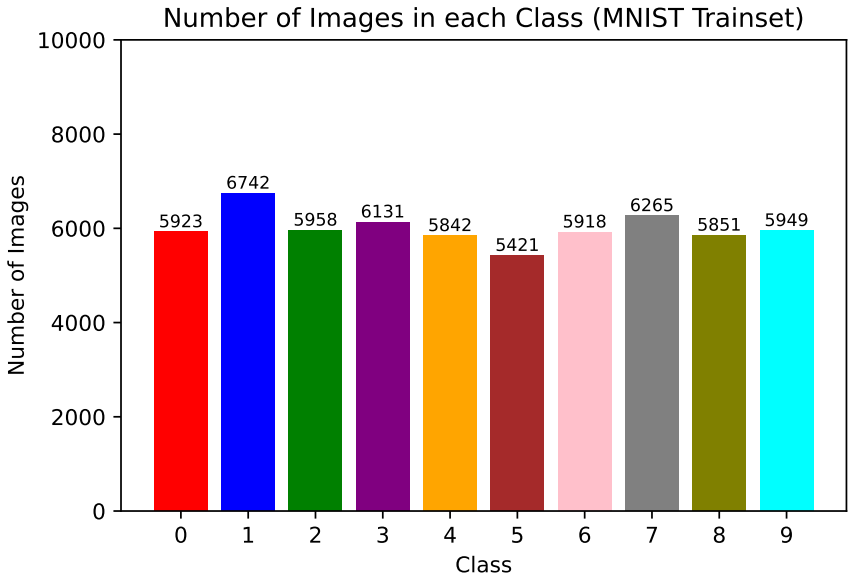
کدهای مربوط به این قسمت، علاوه بر پوشهٔ محلی کدها در این لینک گوگل‌کولب آورده شده است. مدل‌های ذخیره‌شده هم از طریق این لینک گوگل‌درایو در دسترس هستند.

۱.۳ آشنایی و کار با دیتاست (پیش‌پردازش)

برای برآوردن اهداف مدنظر این سوال، دستوراتی در پایتورچ نوشته‌ایم که یک نمودار میله‌ای (Bar Chart) از تعداد تصاویر مربوط به هر دسته (Class) از دیتاست MNIST را رسم می‌کند. ابتدا با استفاده از کتابخانه torchvision دیتاست MNIST به صورت آموزشی (Trainset) بارگیری و در متغیر trainset قرار داده می‌شود. سپس با استفاده از این دیتاست، برچسب هر تصویر در لیست labels قرار می‌گیرد. در قسمت بعدی با توجه به برچسب هر تصویر، تعداد تصاویر مربوط به هر دسته به دست می‌آید و در لیست counts قرار می‌گیرد. سپس با تعریف یک colormap برای هر دسته، نمودار میله‌ای با استفاده از تابع bar ترسیم می‌شود و با استفاده از تابع text برچسب تعداد تصاویر هر دسته نمایش داده می‌شود. در انتها با تعیین عنوان، محورهای مختصات، محدوده مقادیر محورهای x و y، تیک‌های محور x و با استفاده از تابع lrshow نمودار نهایی نمایش داده می‌شود. نتایج به صورتی خواهد بود که در شکل ۱ نمایش داده شده است.

```python
import torch
import torchvision
import matplotlib.pyplot as plt

# Load MNIST training dataset
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True)

# Get labels for each image in the dataset
labels = [trainset[i][1] for i in range(len(trainset))]

# Count the number of images in each class
counts = [labels.count(i) for i in range(10)]

# Define color map for different classes
colors = ['red', 'blue', 'green', 'purple', 'orange',
          'brown', 'pink', 'gray', 'olive', 'cyan']

# Plot the number-class graph
```

```
plt.bar(range(10), counts, color=colors)

# Add text labels for each class
for i in range(10):
    plt.text(i, counts[i] + 100, str(counts[i]), ha='center', fontsize=8)
#     plt.text(i, 4000, str(i), ha='center', fontsize=12)

# Set plot title, axis labels, and limits
plt.title('Number of Images in each Class (MNIST Trainset)')
plt.xlabel('Class')
plt.ylabel('Number of Images')
plt.ylim(0, 10000)

# Set x-axis ticks and labels
plt.xticks(range(10), [str(i) for i in range(10)])

# Save and show plot
plt.savefig('numberclassgraph.pdf')
plt.show()
```



شکل ۱: نمودار تعداد داده‌ها به‌ازای هر گروه برای آموزش.

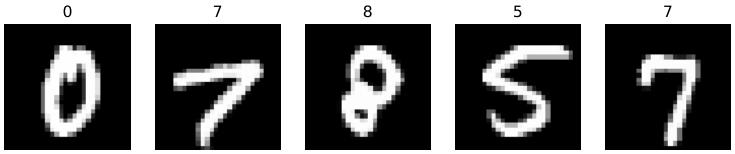
در ادامه و برای نمایش تصادفی پنج داده دستوراتی می‌نویسیم. این دستورات در پایتورچ با استفاده از داده‌های آموزشی دیتاست MNIST ماژول torchvision، پنج تصویر را انتخاب می‌کند و آن‌ها را در یک شکل نمایش می‌دهد. هر تصویر با برچسب مربوطه خود نیز نمایش داده می‌شود. ابتدا دیتاست آموزشی MNIST را از پوشهٔ دیتا دریافت و بارگیری می‌کنیم. سپس با استفاده از تابع رندم، ۵ شمارهٔ صحیح بین ۰ تا تعداد تصاویر آموزشی MNIST (که در اینجا ۶۰۰۰۰ تصویر است) انتخاب می‌کنیم. سپس با استفاده از این شماره‌های صحیح، تصاویر و برچسب‌های مربوطه را از دیتاست آموزشی MNIST به دست

می‌آوریم. در ادامه و با استفاده از کتابخانهٔ matplotlib، یک شکل با ۵ ستون و ۱ سطر تعریف می‌کنیم (fig) و از ۵ تصویر انتخاب شده در بالا برای تصویرسازی در ستون‌های مختلف آن استفاده می‌کنیم. در این‌جا، تصاویر در حالت خاکستری (grayscale) با استفاده از مقیاس خاکستری (cmap=gray) نمایش داده می‌شوند و هر ستون شامل یک تصویر با برچسب مربوطهٔ خود است. در نهایت نتیجه به صورتی شد که در شکل ۲ نشان داده شده است.

```python
import torch
import torchvision
import random
import matplotlib.pyplot as plt

# Load MNIST training dataset
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True)

# Get 5 random images and their labels
indices = random.sample(range(len(trainset)), 5)
images = [trainset[i][0] for i in indices]
labels = [trainset[i][1] for i in indices]

# Plot the images
fig, axs = plt.subplots(1, 5, figsize=(10, 5))
for i in range(5):
    img = images[i]
    if hasattr(img, 'shape'):
        if len(img.shape) == 3 and img.shape[0] == 1:
            img = img.squeeze(0)
    axs[i].imshow(img, cmap='gray')
    axs[i].axis('off')
    axs[i].set_title(str(labels[i]))

# Save and show plot
plt.savefig('fiverand.pdf')
plt.show()
```

در ادامه دستورات زیر را می‌نویسیم که با استفاده از کتابخانه‌های TorchVision و دیتاست MNIST، تصویر اول در دیتاست MNIST را دریافت کرده و اندازه آن را محاسبه می‌کند. در ابتدا، با استفاده از تابع datasets.MNIST، دیتاست MNIST را از آدرس مشخص شده (از روی پارامتر root) لود می‌کند. سپس با استفاده از trainset[0]، تصویر اول در دیتاست (که شامل یک تصویر و برچسب مربوط به آن است) را در متغیرهای image و label ذخیره می‌کند. در قسمت بعدی، با استفاده از تابع size، ابعاد تصویر را به صورت (ارتفاع، عرض) برمی‌گرداند و آنها را در متغیرهای height و width ذخیره می‌کند. سپس با استفاده از تابع print، پیامی را که شامل اندازه تصویر به صورت پیکسلی است، چاپ می‌کند. دستورات اجرایی و نتیجهٔ آن به شرح زیر است.

```python
import torchvision.datasets as datasets

# Load MNIST training dataset
trainset = datasets.MNIST(root='./data', train=True, download=True)
```



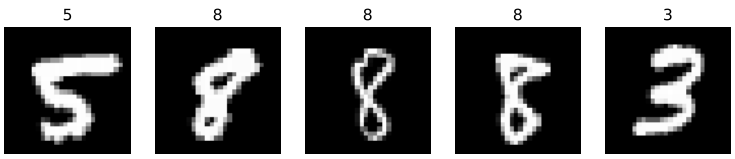
شکل ۲: نمایش تصادفی پنج داده.

```
# Get the first image in the dataset
image, label = trainset[0]

# Calculate the height and width of the image
height, width = image.size

# Print the result
print("Each data of MNIST has {} x {} = {} features.".format(height, width, height*width))

------------------------------------------------

Each data of MNIST has 28 x 28 = 784 features.
```

در پایان با استفاده از دستورات زیر دیتاست MNIST را لود کرده و پس از محاسبه میانگین و انحراف معیار تصاویر، تبدیل‌هایی را تعریف می‌کند که بتواند تصاویر را به شکل مناسبی برای آموزش مدل‌های شبکه عصبی درست کند. سپس با استفاده از تابع `random.sample`، پنج تصویر تصادفی از دیتاست MNIST را برای نمایش دریافت می‌کند و به کمک تبدیل‌های تعریف شده، تصاویر را به شکل مناسبی برای نمایش در پلات می‌آورد. سپس با استفاده از کتابخانه matplotlib، تصاویر در یک پلات نمایش داده می‌شود و در نهایت پلات ذخیره و نمایش داده می‌شود. نمونه‌ای از خروجی در شکل ۳ نمایش داده شده است.

```
import torch
import torchvision
import random
import matplotlib.pyplot as plt

# Load MNIST training dataset
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True)

```

```python
# Get 5 random images and their labels
indices = random.sample(range(len(trainset)), 5)
images = [trainset[i][0] for i in indices]
labels = [trainset[i][1] for i in indices]

# Plot the images
fig, axs = plt.subplots(1, 5, figsize=(10, 5))
for i in range(5):
    img = images[i]
    if hasattr(img, 'shape'):
        if len(img.shape) == 3 and img.shape[0] == 1:
            img = img.squeeze(0)
    axs[i].imshow(img, cmap='gray')
    axs[i].axis('off')
    axs[i].set_title(str(labels[i]))

# Save and show plot
plt.savefig('fiverand.pdf')
plt.show()
```



شکل ۳: نمایش تصادفی پنج داده (نرمال‌شده).

۲.۳ شبکهٔ Auto-Encoder

برای برآوردن اهداف مورد نظر این قسمت از سوال به پیاده‌سازی از یک مدل اتوانکودر با استفاده از کتابخانهٔ پایتورچ برای داده‌های MNIST می‌پردازیم. اتوانکودر یک شبکه عصبی است که سعی می کند تصاویر را به یک فضای برداری کاهش دهد و سپس آن‌ها را بازسازی کند. برای داشتن عملکرد بهتر، مدل در اینجا با استفاده از یک تابع خطای MSE بهینه شده است. این کد شامل چند بخش است:

- تعریف‌های اولیه، از جمله ،dataset hyperparameters و data loaderهای لودینگ داده‌ها: در این بخش، ابتدا hy-perparameters مثل تعداد دورهای آموزش، اندازه بچ، و نرخ یادگیری تعریف شده است. سپس با استفاده از دو مجموعه داده، یکی برای آموزش و دیگری برای تست، داده های MNIST بارگذاری می شوند. این داده ها برای آموزش مدل اتوانکودر استفاده می شوند. برای لود کردن داده‌ها از تابع DataLoader که در این کتابخانه موجود است، استفاده شده است.

- تعریف مدل اتوانکودر: در این بخش، یک مدل اتوانکودر با استفاده از کلاس `nn.Module` در PyTorch تعریف شده است. مدل از دو قسمت encoder و decoder تشکیل شده است. هر قسمت، چند لایه از نورون های پرسپترون با استفاده از توابع فعالیت ReLU برای encoder و sigmoid برای decoder است. اندازه لایه های مختلف از اندازه تصویر ورودی، که در اینجا 28×28 است، شروع شده و کوچکتر شده است.

- تعریف تابع هزینه و بهینه ساز: تابع هزینه برای این مدل MSE است. این تابع برای محاسبهٔ فاصلهٔ بین تصویر اصلی و تصویر بازسازی شده استفاده می شود. در این کد، بهینه ساز Adam برای بهینه سازی وزن‌های مدل استفاده شده است.

- آموزش مدل با داده‌های آموزشی و اعتبارسنجی آن با داده‌های اعتبارسنجی: این بخش از کد، مدل را با استفاده از داده‌های آموزشی آموزش می‌دهد و پس از هر دور آموزش، مقدار تابع خطای مدل را برای داده‌های آموزشی و داده‌های اعتبارسنجی محاسبه می‌کند. برای این منظور، از دو حلقه تکرار استفاده می‌شود. در حلقه اول، داده‌های آموزشی به دسته‌های کوچکتر تقسیم می‌شوند و سپس برای هر دسته، ابتدا گرادیان‌ها را صفر می‌کنیم ((optimizer.zero_grad)، سپس خروجی مدل (output) برای داده‌های ورودی (images) محاسبه می‌شود و در نهایت تابع هزینه برای خروجی محاسبه شده و داده‌های ورودی به‌دست می‌آید. در انتها، با استفاده از تابع ()backward، گرادیان‌های لازم برای به‌روزرسانی وزن‌های مدل (gra-dient descent) محاسبه و بهینه‌ساز به روزرسانی می‌شود ((optimizer.step). حلقه دوم نیز به همین منوال است ولی برای داده‌های اعتبارسنجی استفاده می‌شود. پس از هر دور از حلقه‌ها، میانگین تابع هزینه برای هر دسته آموزش و اعتبارسنجی به‌دست می‌آید و در دو لیست train_loss و valid_loss ذخیره می‌شود. در نهایت، مدل آموزش دیده و وزن‌های به‌روزرسانی شده، در پایان حلقه‌ها در فایل `autoencoder.pth` ذخیره می‌شوند. در انتها، با استفاده از کتابخانهٔ matplotlib نموداری از تغییرات تابع هزینه در هنگام آموزش مدل رسم می‌شود.

دستورات توضیح داده‌شده به شرح زیر و نتایج در انتهای دستورات و در شکل ۴ نمایش داده شده است. همان‌طور که مشاهده می‌شود نتیجه مطلوب است و با تغییر فراپارامترها از آن‌چه که مطابق جدول ۱ در نظر گرفته شده، امکان بهبود بیش‌تر هم خواهد داشت. مخصوصاً آن که فرآیند کاهش اتلاف داده‌های آموزش و اعتبارسنجی هم‌چنان ادامه داشته است و ما به‌دلیل محدودیت‌ها کار را در دورهٔ بیستم متوقف کردیم.

```python
import torch
import torch.nn as nn
import torchvision.datasets as dsets
```

جدول ۱: فراپارامترهای آزمایش اول

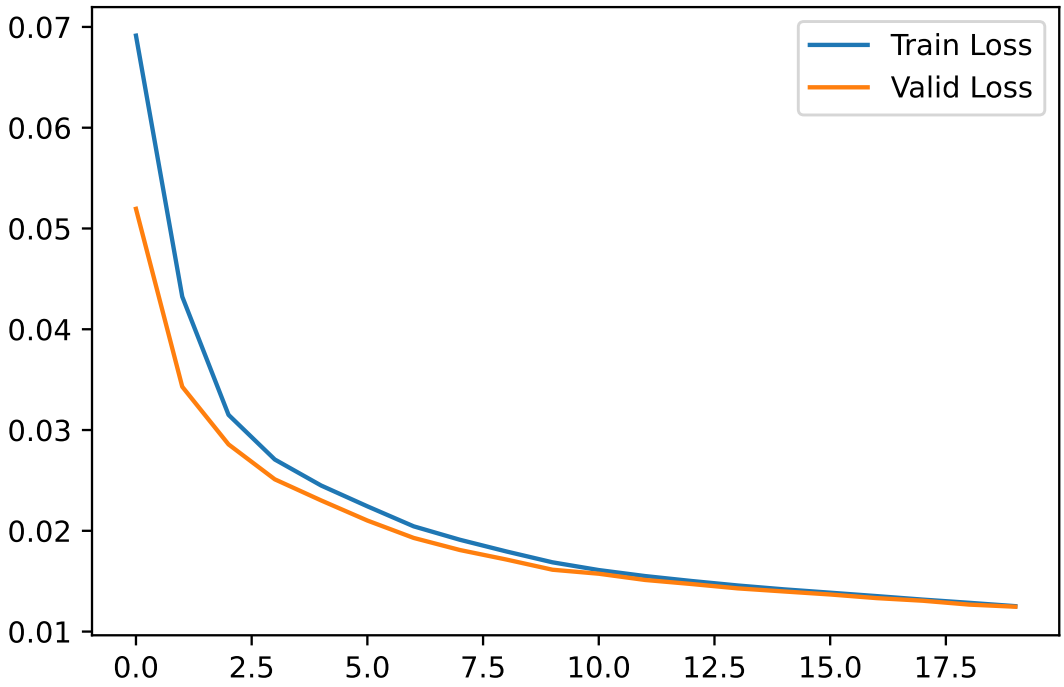
Num epochs	20
Batch size	128
Learning rate	0.0003
Optional FC in Auto-Encoder	300
Optional FC in Classifier	256, 128

```
import torchvision.transforms as transforms
from torch.autograd import Variable

# Hyperparameters
num_epochs = 20
batch_size = 128
learning_rate = 0.0003

# MNIST dataset
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

# Autoencoder model
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 500),
            nn.ReLU(True),
            nn.Linear(500, 300),
            nn.ReLU(True),
```

```
            nn.Linear(300, 100),
            nn.ReLU(True),
            nn.Linear(100, 30),
            nn.ReLU(True))
        self.decoder = nn.Sequential(
            nn.Linear(30, 100),
            nn.ReLU(True),
            nn.Linear(100, 300),
            nn.ReLU(True),
            nn.Linear(300, 500),
            nn.ReLU(True),
            nn.Linear(500, 784),
            nn.Sigmoid())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Create autoencoder object
autoencoder = Autoencoder()

# Loss function and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(autoencoder.parameters(), lr=learning_rate)

# Train the model
train_loss = []
valid_loss = []

for epoch in range(num_epochs):
    # Train
    total_loss = 0
    for i, (images, _) in enumerate(train_loader):
        images = Variable(images.view(-1, 28*28))
        optimizer.zero_grad()
        outputs = autoencoder(images)
        loss = criterion(outputs, images)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    train_loss.append(total_loss/len(train_loader))

    # Validation
```

```
    total_loss = 0
    for i, (images, _) in enumerate(test_loader):
        images = Variable(images.view(-1, 28*28))
        outputs = autoencoder(images)
        loss = criterion(outputs, images)
        total_loss += loss.item()

    valid_loss.append(total_loss/len(test_loader))

    print('Epoch [{}/{}], Train Loss: {:.4f}, Valid Loss: {:.4f}'
          .format(epoch+1, num_epochs, train_loss[-1], valid_loss[-1]))

# Save the trained model weights
torch.save(autoencoder.state_dict(), 'autoencoder.pth')


# Plot loss
import matplotlib.pyplot as plt
plt.plot(train_loss, label='Train Loss')
plt.plot(valid_loss, label='Valid Loss')
plt.legend()
plt.savefig('Loss.pdf')
plt.show()

--------------------------------------------------

Epoch [1/20], Train Loss: 0.0691, Valid Loss: 0.0520
Epoch [2/20], Train Loss: 0.0432, Valid Loss: 0.0343
Epoch [3/20], Train Loss: 0.0315, Valid Loss: 0.0286
Epoch [4/20], Train Loss: 0.0271, Valid Loss: 0.0251
Epoch [5/20], Train Loss: 0.0245, Valid Loss: 0.0230
Epoch [6/20], Train Loss: 0.0224, Valid Loss: 0.0210
Epoch [7/20], Train Loss: 0.0204, Valid Loss: 0.0193
Epoch [8/20], Train Loss: 0.0191, Valid Loss: 0.0181
Epoch [9/20], Train Loss: 0.0180, Valid Loss: 0.0172
Epoch [10/20], Train Loss: 0.0169, Valid Loss: 0.0161
Epoch [11/20], Train Loss: 0.0161, Valid Loss: 0.0157
Epoch [12/20], Train Loss: 0.0155, Valid Loss: 0.0151
Epoch [13/20], Train Loss: 0.0150, Valid Loss: 0.0147
Epoch [14/20], Train Loss: 0.0146, Valid Loss: 0.0143
Epoch [15/20], Train Loss: 0.0142, Valid Loss: 0.0140
Epoch [16/20], Train Loss: 0.0139, Valid Loss: 0.0137
Epoch [17/20], Train Loss: 0.0135, Valid Loss: 0.0133
Epoch [18/20], Train Loss: 0.0132, Valid Loss: 0.0131
Epoch [19/20], Train Loss: 0.0129, Valid Loss: 0.0127
```

```
Epoch [20/20], Train Loss: 0.0125, Valid Loss: 0.0125
```



شکل ۴: نتیجهٔ تابع اتلاف شبکهٔ اتوانکودر (فراپارامترهای آزمایش اول).

۳.۳ طبقه‌بندی

برای برآوردن هدف این سوال یک سیستم پیش‌بینی با استفاده از مدل اتوانکودر و شبکهٔ عصبی کلاسیفایر دو لایه پیشنهاد شده که با استفاده از شبکهٔ اتوانکودر ابعاد داده‌های ورودی را کاهش می‌دهد و با استفاده از شبکه کلاسیفایر، مدل برای پیش‌بینی کلاس مربوط به هر داده آموزش داده می‌شود. ابتدا با استفاده از کتابخانه پایتورچ، داده‌های مربوط به دیتاست MNIST (تصاویر دیجیتال از اعداد) بارگیری و به دو دستهٔ داده‌های آموزش و تست تقسیم می‌شود. سپس یک شبکهٔ اتوانکودر با ساختار چندلایه‌ای با اندازه‌های مختلف برای کاهش ابعاد داده‌های ورودی آموزش داده می‌شود. سپس از وزن‌های ذخیره شده از مدل اتوانکودر برای استخراج ویژگی‌ها برای هر تصویر استفاده می‌شود. در واقع در این برنامه، مدل اتوانکدر به عنوان استخراج‌کنندهٔ ویژگی برای داده‌ها استفاده شده است. این مدل اتوانکدر شامل دو بخش اصلی، یعنی encoder و decoder می‌باشد. encoder وظیفهٔ تبدیل تصاویر به فضای ویژگی با اندازهٔ ۳۰ دارد. در ادامه، برای استفاده در مدل کلاس‌بندی، encoder از مدل اتوانکدر برداشته شده و به عنوان یک لایهٔ ورودی برای مدل کلاس‌بندی استفاده شده است. سپس یک شبکه کلاسیفایر دو لایه‌ای برای پیش بینی کلاس مربوط به هر داده آموزش داده می‌شود. در این مدل، ورودی به شبکه کلاسیفایر، ویژگی‌های استخراج‌شده توسط شبکه اتوانکودر هستند. سپس مدل با استفاده از تابع هزینه تصادفی و الگوریتم بهینه سازی Adam فرآیند آموزش انجام می‌پذیرد. در ادامه، ماتریس درهم‌ریختگی، دقت، یادآوری و امتیاز F۱ مدل بر روی داده‌های تست محاسبه شده و یک نمودار Loss و Accuracy برای داده‌های آموزش و اعتبارسنجی رسم می‌شود. سپس وزن‌های مدل کلاسیفایر ذخیره شده تا برای کاربردهای بعدی استفاده شوند. دستورات اشتفاده شده به شرح زیر است و نتایج در انتهای دستورات و در شکل ۵ و شکل ۶ نشان داده شده است. همان‌طور که مشاهده می‌شود نتیجه مطلوب است و با تغییر فراپارامترها امکان بهبود بیش‌تر هم خواهد داشت. مخصوصاً آن که فرآیند کاهش اتلاف داده‌های آموزش و اعتبارسنجی هم‌چنان ادامه داشته است و ما به‌دلیل محدودیت‌ها کار را در دورهٔ بیستم متوقف کردیم.

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
from sklearn.metrics import confusion_matrix, f1_score, precision_score, recall_score
import seaborn as sns
import matplotlib.pyplot as plt

# Hyperparameters
num_epochs = 20
batch_size = 128
learning_rate = 0.0003

# MNIST dataset
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
```

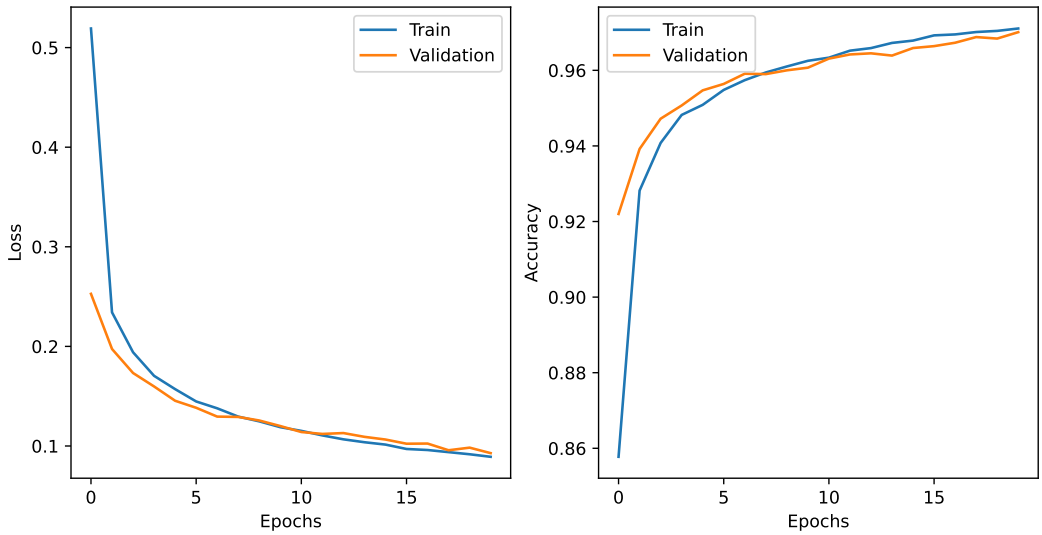
```
                            transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

# Autoencoder model
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 500),
            nn.ReLU(True),
            nn.Linear(500, 300),
            nn.ReLU(True),
            nn.Linear(300, 100),
            nn.ReLU(True),
            nn.Linear(100, 30),
            nn.ReLU(True))
        self.decoder = nn.Sequential(
            nn.Linear(30, 100),
            nn.ReLU(True),
            nn.Linear(100, 300),
            nn.ReLU(True),
            nn.Linear(300, 500),
            nn.ReLU(True),
            nn.Linear(500, 784),
            nn.Sigmoid())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Create autoencoder object and load trained parameters
autoencoder = Autoencoder()
autoencoder.load_state_dict(torch.load('autoencoder.pth'))
encoder = autoencoder.encoder

# Classifier model
```

```
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.layer1 = nn.Linear(30, 256)
        self.layer2 = nn.Linear(256, 128)
        self.layer3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = nn.functional.relu(x)
        x = self.layer2(x)
        x = nn.functional.relu(x)
        x = self.layer3(x)
        return x

# Create classifier object
classifier = Classifier()

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(classifier.parameters(), lr=learning_rate)

# Train the model
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []

for epoch in range(num_epochs):
    # Train
    total_loss = 0
    correct = 0
    total = 0
    for images, labels in train_loader:
        images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)
        features = encoder(images)
        outputs = classifier(features)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == labels).sum().item()
```

```
        total += labels.size(0)

    train_loss.append(total_loss/len(train_loader))
    train_acc.append(correct/total)

    # Validation
    total_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images = Variable(images.view(-1, 28*28))
            labels = Variable(labels)
            features = encoder(images)
            outputs = classifier(features)
            loss = criterion(outputs, labels)
            total_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    valid_loss.append(total_loss/len(test_loader))
    valid_acc.append(correct/total)

    # Print statistics
    print ('Epoch [{}/{}], Train Loss: {:.4f}, Valid Loss: {:.4f}, Train Acc: {:.4f}, Valid Acc: {:.4f}'
           .format(epoch+1, num_epochs, train_loss[-1], valid_loss[-1], train_acc[-1], valid_acc[-1]))

# Save the trained model weights
torch.save(classifier.state_dict(), 'classifier.pth')

# Plot loss and accuracy
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='Train')
plt.plot(valid_loss, label='Validation')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(train_acc, label='Train')
plt.plot(valid_acc, label='Validation')
plt.xlabel('Epochs')
```

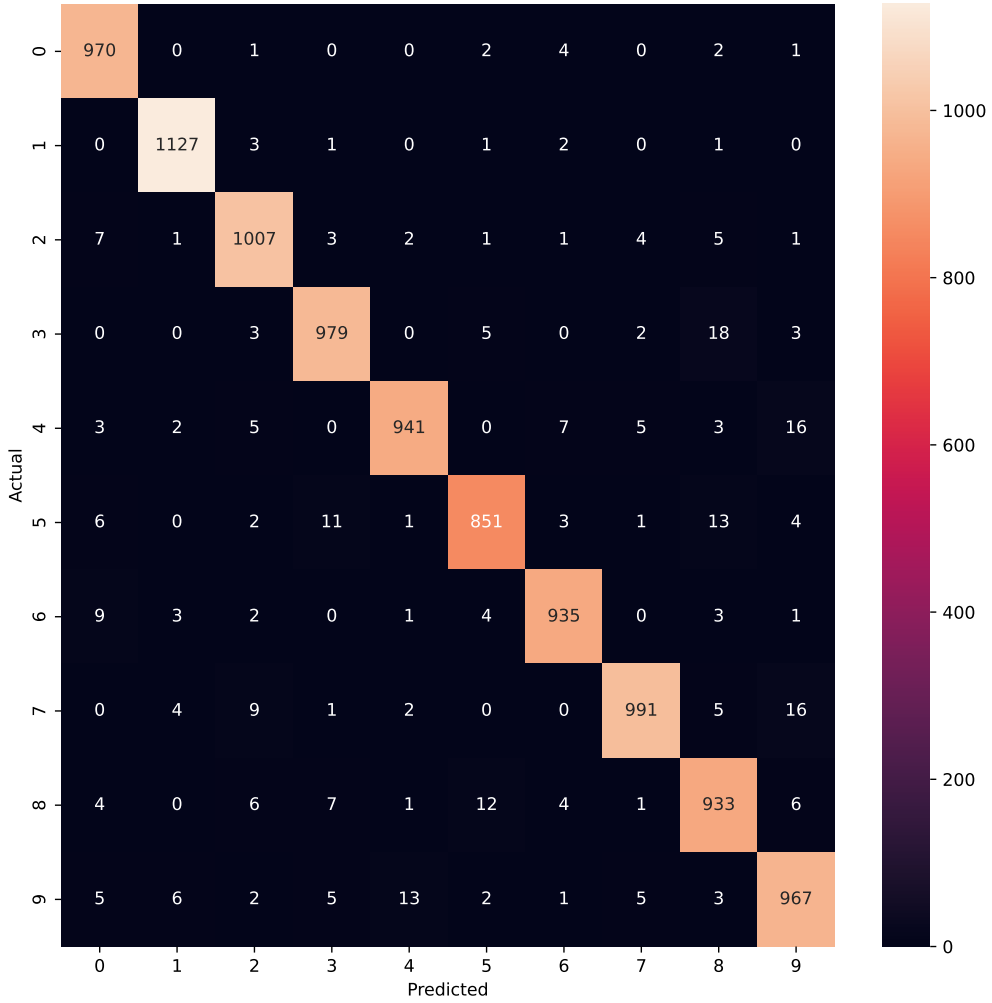
```
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('LossAccur2.pdf')
plt.show()

# Evaluate the model
y_true = []
y_pred = []
with torch.no_grad():
    for images, labels in test_loader:
        images = Variable(images.view(-1, 28*28))
        labels = labels.numpy().tolist()
        features = encoder(images)
        outputs = classifier(features)
        _, predicted = torch.max(outputs.data, 1)
        predicted = predicted.numpy().tolist()
        y_true.extend(labels)
        y_pred.extend(predicted)

# Plot confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.savefig('conf.pdf')
plt.show()

# Report metrics
f1 = f1_score(y_true, y_pred, average='weighted')
precision = precision_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
print('F1 Score:', f1)
print('Precision:', precision)
print('Recall:', recall)

-----------------------------------------------------------

Epoch [1/20], Train Loss: 0.5192, Valid Loss: 0.2528, Train Acc: 0.8578, Valid Acc: 0.9220
Epoch [2/20], Train Loss: 0.2342, Valid Loss: 0.1974, Train Acc: 0.9282, Valid Acc: 0.9392
Epoch [3/20], Train Loss: 0.1941, Valid Loss: 0.1733, Train Acc: 0.9408, Valid Acc: 0.9472
Epoch [4/20], Train Loss: 0.1704, Valid Loss: 0.1598, Train Acc: 0.9482, Valid Acc: 0.9507
Epoch [5/20], Train Loss: 0.1571, Valid Loss: 0.1454, Train Acc: 0.9509, Valid Acc: 0.9547
Epoch [6/20], Train Loss: 0.1447, Valid Loss: 0.1384, Train Acc: 0.9548, Valid Acc: 0.9564
Epoch [7/20], Train Loss: 0.1378, Valid Loss: 0.1295, Train Acc: 0.9574, Valid Acc: 0.9591
```

```
Epoch [8/20], Train Loss: 0.1295, Valid Loss: 0.1292, Train Acc: 0.9594, Valid Acc: 0.9590
Epoch [9/20], Train Loss: 0.1247, Valid Loss: 0.1256, Train Acc: 0.9610, Valid Acc: 0.9600
Epoch [10/20], Train Loss: 0.1189, Valid Loss: 0.1200, Train Acc: 0.9625, Valid Acc: 0.9607
Epoch [11/20], Train Loss: 0.1152, Valid Loss: 0.1140, Train Acc: 0.9634, Valid Acc: 0.9631
Epoch [12/20], Train Loss: 0.1106, Valid Loss: 0.1121, Train Acc: 0.9652, Valid Acc: 0.9642
Epoch [13/20], Train Loss: 0.1067, Valid Loss: 0.1130, Train Acc: 0.9659, Valid Acc: 0.9645
Epoch [14/20], Train Loss: 0.1038, Valid Loss: 0.1092, Train Acc: 0.9673, Valid Acc: 0.9639
Epoch [15/20], Train Loss: 0.1014, Valid Loss: 0.1065, Train Acc: 0.9679, Valid Acc: 0.9659
Epoch [16/20], Train Loss: 0.0970, Valid Loss: 0.1023, Train Acc: 0.9692, Valid Acc: 0.9664
Epoch [17/20], Train Loss: 0.0960, Valid Loss: 0.1024, Train Acc: 0.9695, Valid Acc: 0.9673
Epoch [18/20], Train Loss: 0.0938, Valid Loss: 0.0957, Train Acc: 0.9701, Valid Acc: 0.9688
Epoch [19/20], Train Loss: 0.0917, Valid Loss: 0.0983, Train Acc: 0.9704, Valid Acc: 0.9684
Epoch [20/20], Train Loss: 0.0892, Valid Loss: 0.0928, Train Acc: 0.9711, Valid Acc: 0.9701

----------------------------------------------------------------------------------------

Test Results:
F1 Score: 0.9700930692196958
Precision: 0.9701794432716815
Recall: 0.9701
```



شکل ۵: نتیجهٔ تابع اتلاف و دقت شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش اول).

در انتها یک برنامه اضافه کردیم تا با بهره‌گیری از ساختار پیاده‌سازی‌شده و بارگذاری عکس در هر شکل و ابعادی در محیط گوگل‌کولب، نتیجهٔ پیش‌بینی عدد را روی تصویر آن نمایش دهد. این برنامه ابتدا با گرفتن مسیر تصویر و دو شیء تعریف و ذخیره‌شدهٔ انکودر و کلاسیفایر، کلاس تصویر را پیش بینی می‌کند. ابتدا تصویر با استفاده از PIL باز می‌شود و پس از پیش‌پردازش، انکود می‌شود و در نهایت به کلاسیفایر منتقل می‌شود. حالت no_grad به تابع داده شده است تا تنها با استفاده از

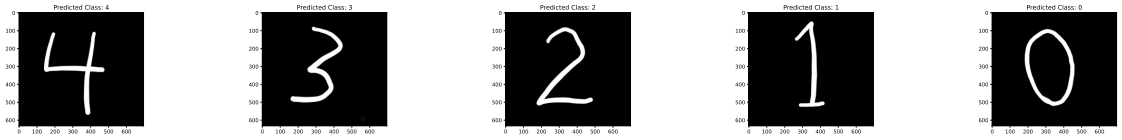


شکل ۶: نتیجهٔ ماتریس درهم‌ریختگی شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش اول).

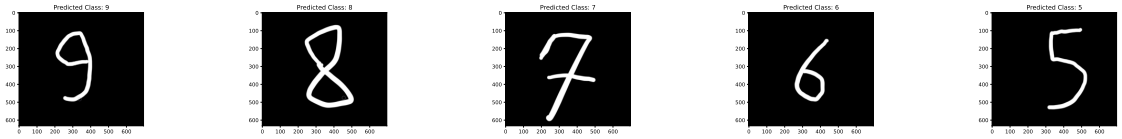
وزن‌های شبکه بدون نوسان آموزشی به ارزیابی بپردازد. نتیجه در شکل ۷ نشان داده شده است که نشان از درصد موفقیت قابل قبول مدل با این فراپارامترهای معمولی است. نتیجهٔ کلاس پیش‌بینی بالای هر تصویر نوشته شده است.

```
import io
import requests
from PIL import Image
import numpy as np

# Define a function to predict the class of an input image
```

```python
def predict_image(image_path, encoder, classifier):
    # Open and preprocess the image
    img = Image.open(image_path).convert('L')
    img = img.resize((28, 28))
    img_tensor = transforms.ToTensor()(img)
    img_tensor = img_tensor.view(-1, 28*28)
    # Pass the image through the encoder and classifier
    with torch.no_grad():
        features = encoder(img_tensor)
        outputs = classifier(features)
        _, predicted = torch.max(outputs.data, 1)
    # Return the predicted class
    return predicted.item()

# Upload an image and predict its class
from google.colab import files

# Upload the image
uploaded = files.upload()

# Predict the class
image_path = next(iter(uploaded))
predicted_class = predict_image(io.BytesIO(uploaded[image_path]), encoder, classifier)

# Print the predicted class
print('Predicted class:', predicted_class)

# Open the uploaded image
img = Image.open(io.BytesIO(uploaded[image_path]))

# Create a figure with the image and the predicted class as the title
fig, ax = plt.subplots()
ax.imshow(img)
ax.set_title('Predicted Class: {}'.format(predicted_class))

# Save the plot as a pdf
plt.savefig('prediction0.pdf')
plt.show()
```



(ه) 4 (د) 3 (ج) 2 (ب) 1 (آ) 0



(ی) 9 (ط) 8 (ح) 7 (ز) 6 (و) 5

شکل ۷: نتیجهٔ تست تصاویر شخصی دست‌نویس با ساختار پیاده‌سازی‌شده (فراپارامترهای آزمایش اول).

۴.۳ بررسی با فراپارامترهای مختلف و یافتن حالاتی بهتر

با در نظرگرفتن فراپارامترها به صورتی که در جدول ۲ نمایش داده شده است، نتیجهٔ شبکهٔ Auto-Encoder به صورتی خواهد بود که در شکل ۸ نشان داده شده است، و نتیجهٔ طبقه‌بندی به صورتی است که در زیر و شکل ۹ و شکل ۱۴ آورده شده است. مشاهده می‌شود که علی‌رغم عدم تفاوت زیاد فراپارامترها و هم‌چنان کاهشی‌بودن تابع اتلاف، نتایج بهبود پیدا کرده است. این موضوع را هم از دقت می‌توان فهمید و هم از درایه‌های روی قطر ماتریس درهم‌ریختگی. نتیجهٔ چند نمونه تست دست‌نویس نیز در شکل ۱۱ نشان داده شده است که مشاهده می‌شود که با وجود افزودن سختی و پیچیدگی بیش‌تر، ساختار به درستی عمل می‌کند.

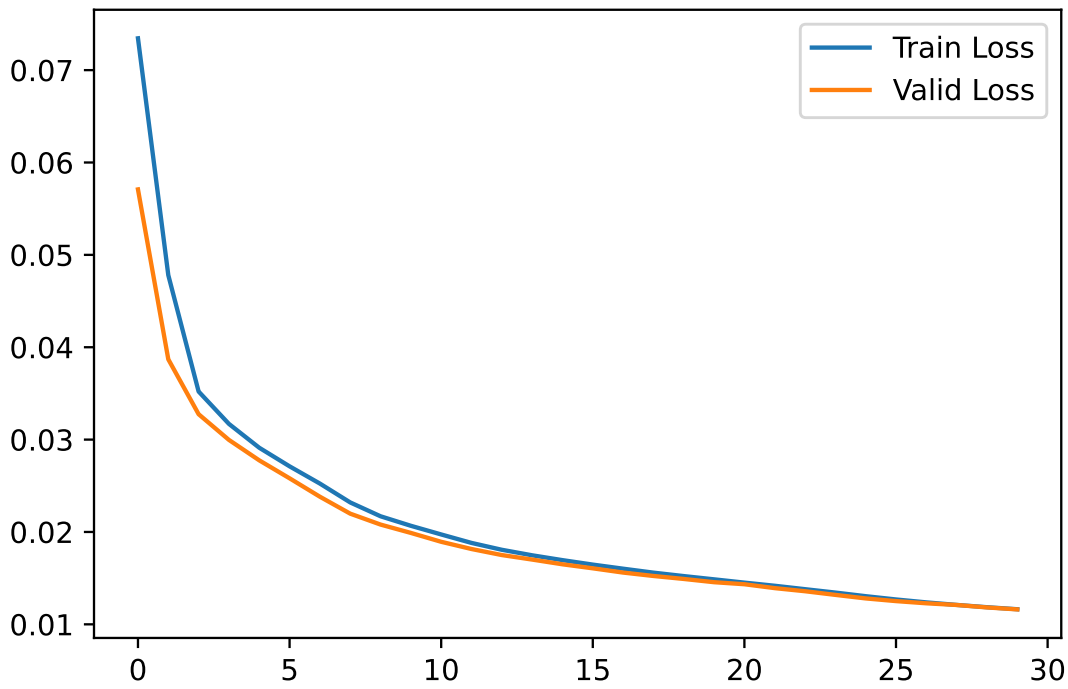
جدول ۲: فراپارامترهای آزمایش دوم

Num epochs	30
Batch size	64
Learning rate	0.0001
Optional FC in Auto-Encoder	300
Optional FC in Classifier	256, 128

```
Auto-Encoder Part Reaults:
Epoch [1/30], Train Loss: 0.0734, Valid Loss: 0.0571
Epoch [2/30], Train Loss: 0.0478, Valid Loss: 0.0387
Epoch [3/30], Train Loss: 0.0352, Valid Loss: 0.0328
Epoch [4/30], Train Loss: 0.0317, Valid Loss: 0.0300
Epoch [5/30], Train Loss: 0.0291, Valid Loss: 0.0278
Epoch [6/30], Train Loss: 0.0271, Valid Loss: 0.0258
Epoch [7/30], Train Loss: 0.0252, Valid Loss: 0.0238
Epoch [8/30], Train Loss: 0.0232, Valid Loss: 0.0220
Epoch [9/30], Train Loss: 0.0217, Valid Loss: 0.0208
```

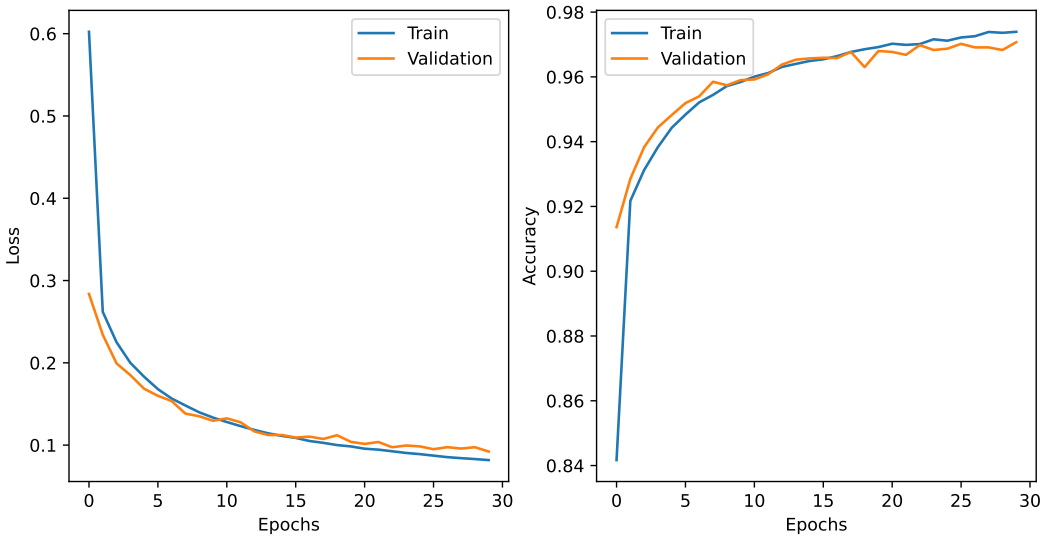
```
Epoch [10/30], Train Loss: 0.0207, Valid Loss: 0.0199
Epoch [11/30], Train Loss: 0.0197, Valid Loss: 0.0189
Epoch [12/30], Train Loss: 0.0188, Valid Loss: 0.0182
Epoch [13/30], Train Loss: 0.0181, Valid Loss: 0.0175
Epoch [14/30], Train Loss: 0.0175, Valid Loss: 0.0170
Epoch [15/30], Train Loss: 0.0169, Valid Loss: 0.0165
Epoch [16/30], Train Loss: 0.0165, Valid Loss: 0.0161
Epoch [17/30], Train Loss: 0.0160, Valid Loss: 0.0156
Epoch [18/30], Train Loss: 0.0156, Valid Loss: 0.0152
Epoch [19/30], Train Loss: 0.0152, Valid Loss: 0.0149
Epoch [20/30], Train Loss: 0.0149, Valid Loss: 0.0146
Epoch [21/30], Train Loss: 0.0145, Valid Loss: 0.0143
Epoch [22/30], Train Loss: 0.0142, Valid Loss: 0.0139
Epoch [23/30], Train Loss: 0.0138, Valid Loss: 0.0136
Epoch [24/30], Train Loss: 0.0134, Valid Loss: 0.0132
Epoch [25/30], Train Loss: 0.0130, Valid Loss: 0.0128
Epoch [26/30], Train Loss: 0.0127, Valid Loss: 0.0125
Epoch [27/30], Train Loss: 0.0124, Valid Loss: 0.0123
Epoch [28/30], Train Loss: 0.0121, Valid Loss: 0.0121
Epoch [29/30], Train Loss: 0.0119, Valid Loss: 0.0118
Epoch [30/30], Train Loss: 0.0116, Valid Loss: 0.0116

--------------------------------------------------

Classification Results:
Epoch [1/30], Train Loss: 0.6023, Valid Loss: 0.2837, Train Acc: 0.8417, Valid Acc: 0.9136
Epoch [2/30], Train Loss: 0.2620, Valid Loss: 0.2341, Train Acc: 0.9217, Valid Acc: 0.9285
Epoch [3/30], Train Loss: 0.2250, Valid Loss: 0.1993, Train Acc: 0.9312, Valid Acc: 0.9383
Epoch [4/30], Train Loss: 0.2000, Valid Loss: 0.1851, Train Acc: 0.9383, Valid Acc: 0.9444
Epoch [5/30], Train Loss: 0.1831, Valid Loss: 0.1685, Train Acc: 0.9443, Valid Acc: 0.9482
Epoch [6/30], Train Loss: 0.1681, Valid Loss: 0.1600, Train Acc: 0.9484, Valid Acc: 0.9519
Epoch [7/30], Train Loss: 0.1566, Valid Loss: 0.1536, Train Acc: 0.9521, Valid Acc: 0.9540
Epoch [8/30], Train Loss: 0.1481, Valid Loss: 0.1383, Train Acc: 0.9544, Valid Acc: 0.9585
Epoch [9/30], Train Loss: 0.1398, Valid Loss: 0.1351, Train Acc: 0.9571, Valid Acc: 0.9574
Epoch [10/30], Train Loss: 0.1335, Valid Loss: 0.1296, Train Acc: 0.9584, Valid Acc: 0.9590
Epoch [11/30], Train Loss: 0.1281, Valid Loss: 0.1325, Train Acc: 0.9601, Valid Acc: 0.9592
Epoch [12/30], Train Loss: 0.1231, Valid Loss: 0.1278, Train Acc: 0.9612, Valid Acc: 0.9608
Epoch [13/30], Train Loss: 0.1185, Valid Loss: 0.1166, Train Acc: 0.9631, Valid Acc: 0.9638
Epoch [14/30], Train Loss: 0.1144, Valid Loss: 0.1123, Train Acc: 0.9640, Valid Acc: 0.9653
Epoch [15/30], Train Loss: 0.1112, Valid Loss: 0.1123, Train Acc: 0.9649, Valid Acc: 0.9657
Epoch [16/30], Train Loss: 0.1088, Valid Loss: 0.1092, Train Acc: 0.9654, Valid Acc: 0.9659
Epoch [17/30], Train Loss: 0.1050, Valid Loss: 0.1103, Train Acc: 0.9664, Valid Acc: 0.9658
Epoch [18/30], Train Loss: 0.1028, Valid Loss: 0.1075, Train Acc: 0.9677, Valid Acc: 0.9677
Epoch [19/30], Train Loss: 0.1001, Valid Loss: 0.1120, Train Acc: 0.9685, Valid Acc: 0.9630
Epoch [20/30], Train Loss: 0.0984, Valid Loss: 0.1040, Train Acc: 0.9692, Valid Acc: 0.9680
```

```
Epoch [21/30], Train Loss: 0.0956, Valid Loss: 0.1015, Train Acc: 0.9702, Valid Acc: 0.9677
Epoch [22/30], Train Loss: 0.0945, Valid Loss: 0.1038, Train Acc: 0.9699, Valid Acc: 0.9668
Epoch [23/30], Train Loss: 0.0925, Valid Loss: 0.0974, Train Acc: 0.9701, Valid Acc: 0.9698
Epoch [24/30], Train Loss: 0.0905, Valid Loss: 0.0995, Train Acc: 0.9716, Valid Acc: 0.9683
Epoch [25/30], Train Loss: 0.0891, Valid Loss: 0.0985, Train Acc: 0.9712, Valid Acc: 0.9687
Epoch [26/30], Train Loss: 0.0872, Valid Loss: 0.0950, Train Acc: 0.9722, Valid Acc: 0.9702
Epoch [27/30], Train Loss: 0.0854, Valid Loss: 0.0975, Train Acc: 0.9726, Valid Acc: 0.9691
Epoch [28/30], Train Loss: 0.0841, Valid Loss: 0.0958, Train Acc: 0.9739, Valid Acc: 0.9691
Epoch [29/30], Train Loss: 0.0831, Valid Loss: 0.0975, Train Acc: 0.9736, Valid Acc: 0.9683
Epoch [30/30], Train Loss: 0.0818, Valid Loss: 0.0922, Train Acc: 0.9739, Valid Acc: 0.9707

----------------------------------------------------------------------------------------------

Test Results:
F1 Score: 0.9707119583033161
Precision: 0.9708467316384838
Recall: 0.9707
```



شکل ۸: نتیجهٔ تابع اتلاف شبکهٔ اتوانکودر (فراپارامترهای آزمایش دوم).

در آزمایش سوم و با در نظرگرفتن فراپارامترها به صورتی که در جدول ۳ نمایش داده شده است، نتیجهٔ شبکهٔ Auto-Encoder به صورتی خواهد بود که در شکل ۱۲ نشان داده شده است، و نتیجهٔ طبقه‌بندی به صورتی است که در زیر و شکل ۱۳ و ؟؟ آورده



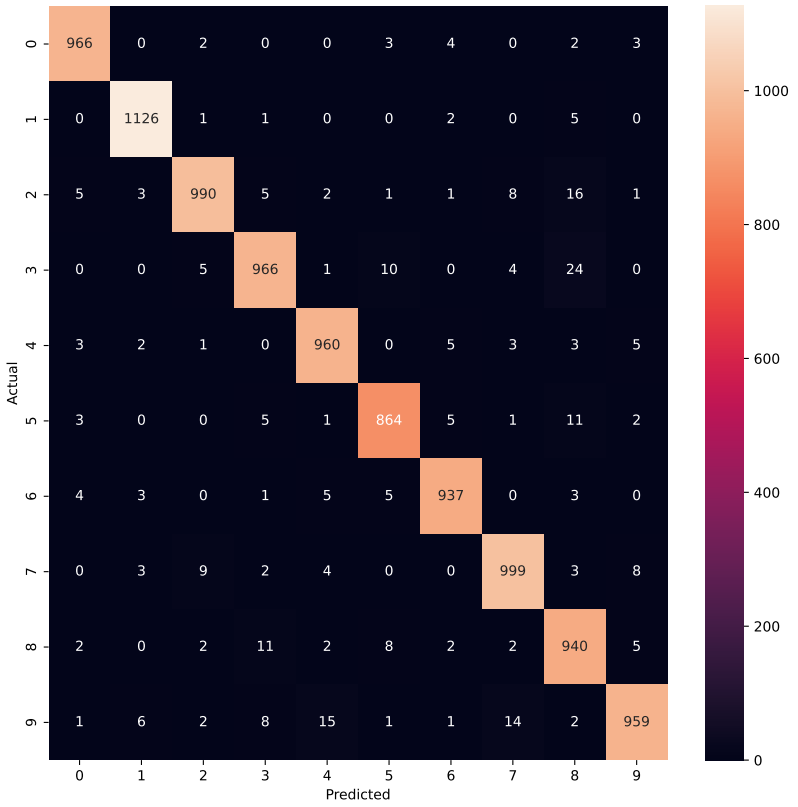
شکل ۹: نتیجهٔ تابع اتلاف و دقت شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش دوم).

شده است. مشاهده می‌شود که علی‌رغم عدم تفاوت زیاد فراپارامترها و هم‌چنان کاهشی‌بودن تابع اتلاف، نتایج بهبود پیدا کرده است. این موضوع را هم از دقت می‌توان فهمید و هم از درایه‌های روی قطر ماتریس درهم‌ریختگی. نتیجهٔ چند نمونه تست دست‌نویس نیز در شکل ۱۵ نشان داده شده است که مشاهده می‌شود که با وجود افزودن سختی و پیچیدگی بیش‌تر، ساختار به درستی عمل می‌کند.

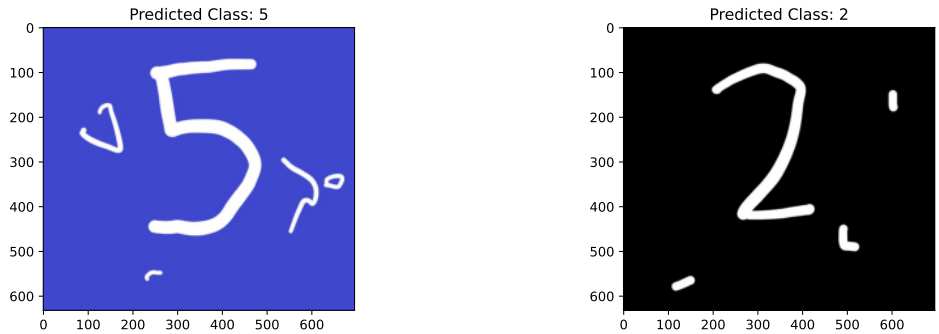
جدول ۳: فراپارامترهای آزمایش سوم

Num epochs	100
Batch size	64
Learning rate	0.0001
Optional FC in Auto-Encoder	250
Optional FC in Classifier	128, 64

```
Auto-Encoder Part Reaults:
.
.
Epoch [80/100], Train Loss: 0.0070, Valid Loss: 0.0075
Epoch [81/100], Train Loss: 0.0070, Valid Loss: 0.0075
Epoch [82/100], Train Loss: 0.0069, Valid Loss: 0.0074
Epoch [83/100], Train Loss: 0.0069, Valid Loss: 0.0074
Epoch [84/100], Train Loss: 0.0068, Valid Loss: 0.0073
Epoch [85/100], Train Loss: 0.0068, Valid Loss: 0.0073
Epoch [86/100], Train Loss: 0.0067, Valid Loss: 0.0072
```



شکل ۱۰: نتیجهٔ ماتریس درهم‌ریختگی شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش دوم).

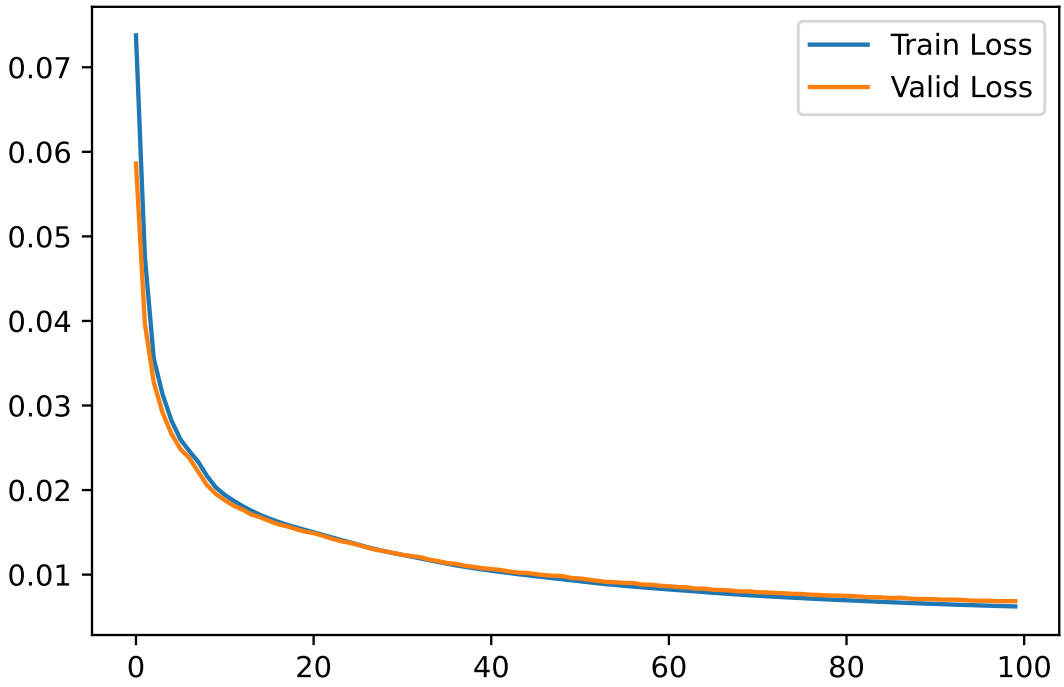


(ب) 5 (آ) 2

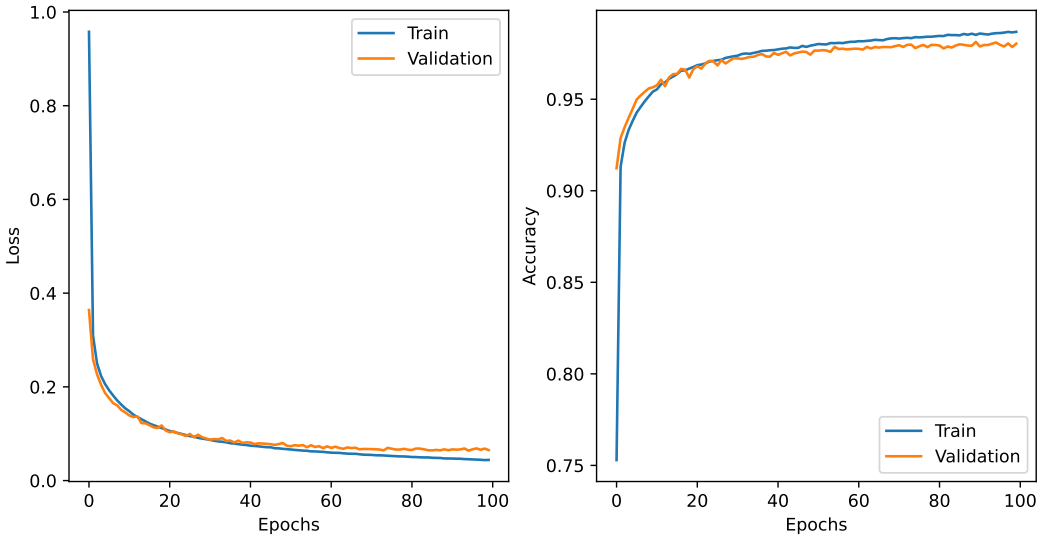
شکل ۱۱: نتیجهٔ تست تصاویر شخصی دست‌نویس با ساختار پیاده‌سازی‌شده (فراپارامترهای آزمایش دوم).

```
Epoch [87/100], Train Loss: 0.0067, Valid Loss: 0.0073
Epoch [88/100], Train Loss: 0.0066, Valid Loss: 0.0072
Epoch [89/100], Train Loss: 0.0066, Valid Loss: 0.0071
Epoch [90/100], Train Loss: 0.0066, Valid Loss: 0.0071
Epoch [91/100], Train Loss: 0.0065, Valid Loss: 0.0071
Epoch [92/100], Train Loss: 0.0065, Valid Loss: 0.0070
Epoch [93/100], Train Loss: 0.0065, Valid Loss: 0.0070
Epoch [94/100], Train Loss: 0.0064, Valid Loss: 0.0070
Epoch [95/100], Train Loss: 0.0064, Valid Loss: 0.0069
Epoch [96/100], Train Loss: 0.0064, Valid Loss: 0.0069
Epoch [97/100], Train Loss: 0.0063, Valid Loss: 0.0069
Epoch [98/100], Train Loss: 0.0063, Valid Loss: 0.0069
Epoch [99/100], Train Loss: 0.0063, Valid Loss: 0.0069
Epoch [100/100], Train Loss: 0.0062, Valid Loss: 0.0069

--------------------------------------------------

Classification Results:
.
.
Epoch [80/100], Train Loss: 0.0510, Valid Loss: 0.0659, Train Acc: 0.9844, Valid Acc: 0.9796
Epoch [81/100], Train Loss: 0.0502, Valid Loss: 0.0653, Train Acc: 0.9846, Valid Acc: 0.9793
Epoch [82/100], Train Loss: 0.0502, Valid Loss: 0.0683, Train Acc: 0.9845, Valid Acc: 0.9777
Epoch [83/100], Train Loss: 0.0497, Valid Loss: 0.0682, Train Acc: 0.9852, Valid Acc: 0.9788
Epoch [84/100], Train Loss: 0.0494, Valid Loss: 0.0667, Train Acc: 0.9850, Valid Acc: 0.9782
Epoch [85/100], Train Loss: 0.0494, Valid Loss: 0.0647, Train Acc: 0.9850, Valid Acc: 0.9793
Epoch [86/100], Train Loss: 0.0487, Valid Loss: 0.0643, Train Acc: 0.9849, Valid Acc: 0.9800
Epoch [87/100], Train Loss: 0.0484, Valid Loss: 0.0654, Train Acc: 0.9856, Valid Acc: 0.9797
Epoch [88/100], Train Loss: 0.0482, Valid Loss: 0.0645, Train Acc: 0.9852, Valid Acc: 0.9793
Epoch [89/100], Train Loss: 0.0472, Valid Loss: 0.0667, Train Acc: 0.9858, Valid Acc: 0.9792
Epoch [90/100], Train Loss: 0.0471, Valid Loss: 0.0647, Train Acc: 0.9851, Valid Acc: 0.9813
Epoch [91/100], Train Loss: 0.0470, Valid Loss: 0.0666, Train Acc: 0.9859, Valid Acc: 0.9788
Epoch [92/100], Train Loss: 0.0467, Valid Loss: 0.0660, Train Acc: 0.9856, Valid Acc: 0.9796
Epoch [93/100], Train Loss: 0.0466, Valid Loss: 0.0664, Train Acc: 0.9853, Valid Acc: 0.9796
Epoch [94/100], Train Loss: 0.0458, Valid Loss: 0.0685, Train Acc: 0.9858, Valid Acc: 0.9803
Epoch [95/100], Train Loss: 0.0456, Valid Loss: 0.0639, Train Acc: 0.9861, Valid Acc: 0.9810
Epoch [96/100], Train Loss: 0.0451, Valid Loss: 0.0667, Train Acc: 0.9861, Valid Acc: 0.9797
Epoch [97/100], Train Loss: 0.0448, Valid Loss: 0.0687, Train Acc: 0.9864, Valid Acc: 0.9788
Epoch [98/100], Train Loss: 0.0444, Valid Loss: 0.0657, Train Acc: 0.9868, Valid Acc: 0.9805
Epoch [99/100], Train Loss: 0.0437, Valid Loss: 0.0683, Train Acc: 0.9865, Valid Acc: 0.9785
Epoch [100/100], Train Loss: 0.0440, Valid Loss: 0.0651, Train Acc: 0.9868, Valid Acc: 0.9803

----------------------------------------------------------------------------------------

Test Results:
```

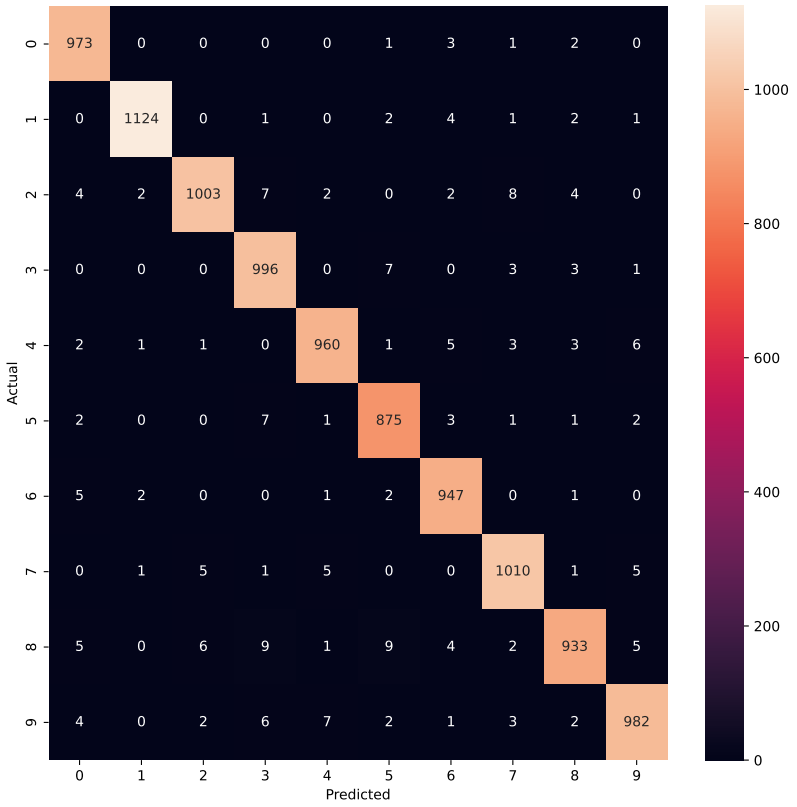
```
F1 Score: 0.9802860641202584
Precision: 0.980346785543814
Recall: 0.9803
```



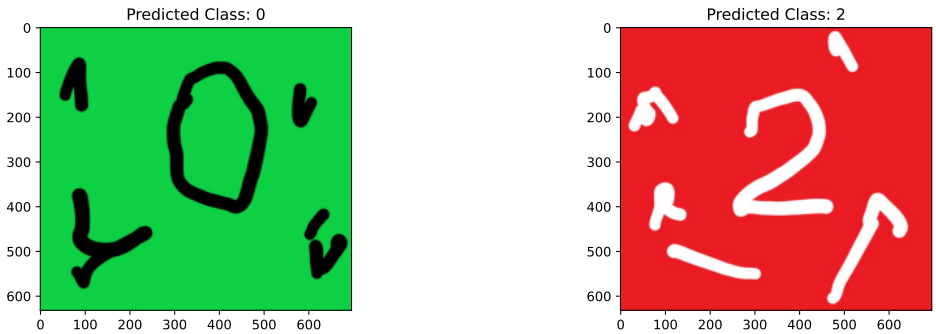
شکل ۱۲: نتیجهٔ تابع اتلاف شبکهٔ اتوانکودر (فراپارامترهای آزمایش سوم).



شکل ۱۳: نتیجهٔ تابع اتلاف و دقت شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش سوم).



شکل ۱۴: نتیجهٔ ماتریس درهم‌ریختگی شبکهٔ طبقه‌بندی (فراپارامترهای آزمایش سوم).



(ب) 0 (آ) 2

شکل ۱۵: نتیجهٔ تست تصاویر شخصی دست‌نویس با ساختار پیاده‌سازی‌شده (فراپارامترهای آزمایش سوم).

۴ پاسخ پرسش سوم - راه‌حل دوم

در راه‌حل دوم برای قسمت جداکنندهٔ انکودر به طریقی دیگر عمل کرده‌ایم که کدها و نتایج مربوط به آن از طریق این لینک در دسترس است. خلاصه‌ای از بخش‌های اضافه‌شده به شرح زیر است:

```
import torch
autoencoder = Autoencoder()
# Step 1: Load your pre-trained autoencoder model
autoencoder = torch.load('eutoencoder.pth')

# Step 2: Access the encoder part of the model
encoder = autoencoder.encoder

# Step 3: Load or generate the data that you want to obtain the encoder unit output for
x_data = torch.randn(100, 784) # example data

# Step 4: Pass the data through the encoder layer(s) to obtain the encoded representations
encoded_data = encoder(x_data)

# Step 5: Save the encoded representations to a file or data structure for later reuse
torch.save(encoded_data, 'encoded_data.pth')


import torch

# Load the encoded data from the saved file
encoded_data = torch.load('encoded_data.pth')

# Use the encoded data for another task or model
# ...


torch.save(autoencoder, 'eutoencoder.pth')

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0,), (1,))])

import torch
import torch.nn as nn
import torch.optim as optim
```

```
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0,), (1,))])
# Load data
train_dataset = MNIST(root='content/encoded_data.pth/', train=True, transform=ToTensor(),
    download=True)
test_dataset = MNIST(root='./data', train=False, transform=ToTensor(), download=True)

# Set hyperparameters
batch_size = 128
num_epochs = 10
learning_rate = 0.001

# Create dataloaders
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

# Define neural network model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Instantiate model and loss function
model = Net().to(device)
criterion = nn.CrossEntropyLoss()

```

```
# Instantiate optimizer
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
train_loss = []
train_acc = []
val_loss = []
val_acc = []
total_step = len(train_loader)
for epoch in range(num_epochs):
    # Training loop
    running_loss = 0.0
    running_corrects = 0
    for i, (images, labels) in enumerate(train_loader):
        # Move images and labels to device
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Compute accuracy and loss
        _, preds = torch.max(outputs.data, 1)
        running_loss += loss.item() * images.size(0)
        running_corrects += torch.sum(preds == labels.data)

    # Compute training loss and accuracy
    epoch_loss = running_loss / len(train_loader.dataset)
    epoch_acc = running_corrects.double() / len(train_loader.dataset)
    train_loss.append(epoch_loss)
    train_acc.append(epoch_acc)

    # Validation loop
    val_running_loss = 0.0
    val_running_corrects = 0
    with torch.no_grad():
        for images, labels in test_loader:
            # Move images and labels to device
            images = images.to(device)
```

```
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Compute accuracy and loss
            _, preds = torch.max(outputs.data, 1)
            val_running_loss += loss.item() * images.size(0)
            val_running_corrects += torch.sum(preds == labels.data)

    # Compute validation loss and accuracy
    val_loss.append(val_running_loss / len(test_loader.dataset))

    val_epoch_loss = val_running_loss / len(test_loader.dataset)
    val_epoch_acc = val_running_corrects.double() / len(test_loader.dataset)
    val_loss.append(val_epoch_loss)
    val_acc.append(val_epoch_acc)

    print('Epoch [{}/{}], Step [{}/{}], Train Loss: {:.4f}, Train Acc: {:.4f}, Val Loss: {:.4f}, Val Acc: {:.4f}'
          .format(epoch+1, num_epochs, i+1, total_step, epoch_loss, epoch_acc, val_epoch_loss, val_epoch_acc))

# Plot the training and validation accuracy and loss
plt.figure(figsize=(10, 5))
plt.title("Training and Validation Loss")
plt.plot(train_loss, label="Training Loss")
plt.plot(val_loss, label="Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()

plt.figure(figsize=(10, 5))
plt.title("Training and Validation Accuracy")
plt.plot(train_acc, label="Training Accuracy")
plt.plot(val_acc, label="Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()

# Test the model
model.eval()
```

```
y_pred = []
y_true = []
with torch.no_grad():
    for images, labels in test_loader:
        # Move images and labels to device
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)

        # Compute predicted label and add to list
        _, preds = torch.max(outputs.data, 1)
        y_pred += preds.cpu().numpy().tolist()
        y_true += labels.cpu().numpy().tolist()

    model.eval()
with torch.no_grad():
     correct = 0
     total = 0
     for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# Calculate test accuracy
test_acc = 100 * correct / total

# Print test accuracy
#print('Test Accuracy: {:.2f}%'.format(test_acc))


# Plot the confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.savefig('conf.pdf')
plt.show()

# Print test accuracy
```

```
215 print('Test Accuracy: {:.2f}%'.format(test_acc))
```