
 <p>دانشگاه صنعتی خواجه نصیرالدین طوسی دانشکده مهندسی برق - گروه مهندسی کنترل</p>	<p>به نام خدا</p> <p>دانشگاه تهران - دانشکاه صنعتی خواجه نصیرالدین طوسی تهران</p> <p>دانشکده مهندسی برق و کامپیوتر</p>	
<p>شبکه‌ی متخاصم مولد</p>		

<p>محمد جواد احمدی</p>	<p>نام و نام خانوادگی</p>
<p>۴۰۱۰۰۰۸۶</p>	<p>شماره دانشجویی</p>

فهرست مطالب

۳	۱ پاسخ پرسش دوم
۳	۱.۱ پاسخ قسمت ۱ - بارگذاری داده‌ها و شبکه‌ی ResNet
۱۷	۲.۱ پاسخ قسمت ۲ - شبکه‌ی Conditional DCGAN
۴۷	۳.۱ پاسخ قسمت ۳ - طبقه‌بندی به کمک داده‌های تولیدشده توسط مولد

فهرست تصاویر

۱۱	نمایش توزیع داده‌ها و کلاس‌های آن‌ها	۱
۱۲	نمایش تعدادی از داده‌ها و کلاس‌های آن‌ها	۲
۱۷	نمودار تابع اتلاف و دقت مدل در حالت عادی	۳
۱۸	ماتریس درهم‌ریختگی در حالت عادی	۴
۳۷	نمودار توابع اتلاف	۵
۴۷	برخی نمونه‌های تولیدی و حساب سرانگشتی از تصاویر تولیدی شبکه	۶
۴۸	برخی نمونه‌های تولیدی و حساب سرانگشتی از تصاویر تولیدی شبکه	۷
۵۵	نمودار تابع اتلاف و دقت مدل در حالت مولد	۸
۵۵	ماتریس درهم‌ریختگی در حالت مولد	۹

پرسش ۲. شبکه‌ی متخاصم مولد

۱ پاسخ پرسش دوم

توضیح پوشه‌ی کدهای شبکه‌ی متخاصم مولد

کدهای مربوط به این قسمت، علاوه بر پوشه‌ی محلی کدها در این لینک گوگل کولب و یا این لینک گوگل کولب آورده شده است. مدل‌های ذخیره شده هم از طریق این لینک در دسترس هستند.

۱.۱ پاسخ قسمت ۱ - بارگذاری داده‌ها و شبکه‌ی ResNet

داده‌های جمع‌آوری شده BreastMNIST شامل تصاویر سونوگرافی سینه در زنان با سن بین ۲۵ تا ۷۵ سال می‌باشد. این داده در سال ۲۰۱۸ جمع‌آوری شده است و تعداد بیماران آن شامل ۶۰۰ بیمار زن می‌باشد. مجموعه داده شامل ۷۸۰ تصویر با اندازه متوسط تصویر 500×500 پیکسل است. تصاویر به فرمت PNG و در حالت خاکستری هستند. تصاویر به سه دسته‌بندی تقسیم شده‌اند که شامل دسته‌های طبیعی، خوش‌خیم و خبیث می‌شوند. در ایت قسمت با استفاده از تصاویر با کیفیت پایین، ما با کار دسته‌بندی دودویی روبرو هستیم، که در آن دسته‌های طبیعی و خوش‌خیم را به عنوان مثبت در نظر می‌گیریم و آنها را در برابر بدخیم به عنوان منفی دسته‌بندی می‌کنیم. ابتدا دستوراتی برای بارگذاری و کنکاش در داده‌ها می‌نویسیم. در این دستورات ابتدا مسیر اصلی برای ذخیره‌سازی داده‌ها را تعریف می‌کنیم و تمهیدی می‌اندیشیم که اگر آن پوشه و مسیر وجود نداشته، به صورت خودکار ساخته شود. در ادامه، نام مجموعه‌داده را "breastmnist" تعیین می‌کنیم. مجموعه‌داده‌های MedMNIST شامل مجموعه‌هایی مانند "breastmnist" و "pathmnist" است. سپس تگ download به True تنظیم می‌شود، که به معنی بارگیری مجموعه داده‌ها در صورتی که موجود نباشند است. در ادامه با دستوری نام مجموعه‌داده را به حروف کوچک تبدیل می‌کنیم. این کار معمولاً برای جلوگیری از خطاهای تایپی و تضاد در نام‌گذاری استفاده می‌شود. با انجام این کارها، با استفاده از نام مجموعه‌داده، اطلاعات مربوط به آن مجموعه داده (مانند نوع کار و کلاس پایتون متناظر) از دیکشنری INFO در کتابخانه MedMNIST بازیابی می‌شود. هم‌چنین وظیفه‌ی مربوط به مجموعه‌داده (از اطلاعات در مرحله قبل) به عنوان مقدار متغیر task ذخیره می‌شود. در قسمت DataClass، نام کلاس مربوط به مجموعه داده از اطلاعات در مرحله قبل استخراج شده و با استفاده از تابع getattr از کتابخانه MedMNIST به عنوان مقدار متغیر DataClass تعیین می‌شود. این متغیر DataClass به عنوان یک نوع داده برای ایجاد نمونه‌های مجموعه‌داده در مراحل بعدی استفاده می‌شود. سپس دستوراتی می‌نویسیم تا تصویر تبدیل به تانسور و سپس نرمال شود. با انجام این کارها با استفاده از متغیر DataClass، نمونه‌ای از مجموعه داده برای مرحله آموزش (split=train) ایجاد می‌شود. در اینجا، تبدیلات، transform، تگ بارگیری download و مسیر اصلی root_dir به عنوان آرگومان‌ها برای ساختن مجموعه داده استفاده می‌شوند. با انجام کارهایی مشابه زیرمجموعه‌های اعتبارسنجی و آزمون هم ایجاد می‌شوند. هم‌چنین، با استفاده از متغیر DataClass، نمونه‌ای از مجموعه داده با قالب PIL ایجاد می‌شود. این مجموعه داده برای بررسی تصاویر به صورت PIL (Python Imaging Library) استفاده می‌شود. در نهایت، با استفاده از کلاس DataLoader، یک دیتالودر از مجموعه‌ها به صورت داده‌های دسته‌ای و بعضاً برزده‌شده (برای آموزش) ساخته می‌شود. در ادامه دستوراتی را جهت نمایش اطلاعات مجموعه‌داده می‌نویسیم. ابتدا برچسب‌های موجود در مجموعه‌داده آموزش را دریافت می‌کنیم و در

متغیر `labels` ذخیره می‌کنیم. با استفاده از تابع `np.unique` برچسب‌های منحصر به فرد موجود در مجموعه‌داده آموزش را پیدا می‌کنیم و در متغیر `unique_classes` ذخیره می‌کنیم. هم‌چنین تعداد کلاس‌های موجود در مجموعه‌داده را محاسبه و در متغیر `num_classes` ذخیره می‌کنیم. در نهایت اطلاعات مربوط به مجموعه‌داده از جمله تعداد کانال‌ها در تصاویر را خروجی می‌گیریم. دستورات در برنامه ۱ آورده شده است و نتایج به شرح برنامه ۲ است.

Program 1: Load Data

```
1 # Import necessary libraries
2 import numpy as np
3 import random
4 import os
5 import shutil
6 import matplotlib.pyplot as plt
7 import torch
8 import torch.nn as nn
9 import torch.nn.functional as F
10 import torch.optim as optim
11 import torch.utils.data as data
12 import torchvision
13 import torchvision.datasets as datasets
14 import torchvision.transforms as transforms
15 from torch.autograd import Variable
16 from PIL import Image
17 import PIL.ImageOps
18 from matplotlib import pyplot as plt
19 from tqdm import tqdm
20 import time
21 # store starting time
22 begin = time.time()
23
24 # Install the medmnist package
25 !pip install medmnist
26
27 # Import the medmnist package and its sub-modules
28 import medmnist
29 from medmnist import INFO, Evaluator
30
31 # Set matplotlib to inline mode
32 %matplotlib inline
33
34 # Set matplotlib backend to TkAgg
35 # matplotlib.use("TkAgg")
36
37 import medmnist
38 from torchvision import transforms
39 from torch.utils.data import DataLoader
```

```

40 import os
41
42 # Define the root directory for saving the data
43 root_dir = '/content/data/MedMNIST' # Specify the desired root directory path
44
45 # Create the root directory if it does not exist
46 os.makedirs(root_dir, exist_ok=True)
47
48 dataset_name = "breastmnist" # The name of the dataset
49 download = True # Flag to download the dataset if not available locally
50 dataset_name = dataset_name.lower() # Convert the dataset name to lowercase
51
52 NUM_EPOCHS = 15 # Number of epochs for training
53 BATCH_SIZE = 128 # Batch size for data loading
54 lr = 0.001 # Learning rate for the optimizer
55
56 info = INFO[dataset_name] # Retrieve information about the dataset
57 task = info['task'] # The task associated with the dataset
58
59 DataClass = getattr(medmnist, info['python_class']) # Get the data class for the dataset
60
61 # Data preprocessing transforms
62 data_transform = transforms.Compose([
63     # transforms.Resize(224),
64     # transforms.Lambda(lambda image: image.convert('RGB')),
65     transforms.ToTensor(), # Convert data to tensors
66     transforms.Normalize(mean=[.5], std=[.5]) # Normalize the data
67 ])
68
69 # Load the training and testing datasets
70 train_dataset = DataClass(split='train', transform=data_transform, download=download, root=
    root_dir)
71 test_dataset = DataClass(split='test', transform=data_transform, download=download, root=root_dir
    )
72 val_dataset = DataClass(split='val', transform=data_transform, download=download, root=root_dir)
73 pil_dataset = DataClass(split='train', download=download, root=root_dir) # Load the PIL dataset
    for visualization (if needed)
74
75 # Create data loaders for training and testing datasets
76 train_loader = data.DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
77 val_loader = data.DataLoader(dataset=val_dataset, batch_size=2*BATCH_SIZE, shuffle=False)
78 test_loader = data.DataLoader(dataset=test_dataset, batch_size=2*BATCH_SIZE, shuffle=False)
79
80
81 # Get the labels from the train_dataset

```

```

82 labels = train_dataset.labels
83
84 # Find the unique classes in the dataset
85 unique_classes = np.unique(labels)
86
87 # Calculate the number of classes
88 num_classes = len(unique_classes)
89
90 # Print the number of classes
91 print("Number of classes in the dataset:", num_classes)
92
93 # Verify the number of classes by accessing the label information directly from the train_dataset
94 print("Label info of dataset:", train_dataset.info['label'])
95
96 # Print the complete information of the train_dataset
97 print("Info of dataset:", train_dataset.info)
98
99 train_dataset.info
100
101 # Calculate the number of channels in the image
102
103 # Access the first sample of the train_dataset and retrieve the image
104 image = train_dataset[0][0]
105
106 # Get the number of channels in the image
107 num_channels = image.shape[0]
108
109 # Print the number of channels
110 print("Number of channels of the image:", num_channels)
111
112 print(train_dataset)
113 print("=====")
114 print(test_dataset)

```

Program 2: Data Info

```

1 Number of classes in the dataset: 2
2 Label info of dataset: {'0': 'malignant', '1': 'normal, benign'}
3 Info of dataset: {'python_class': 'BreastMNIST', 'description': 'The BreastMNIST is based on a
dataset of 780 breast ultrasound images. It is categorized into 3 classes: normal, benign,
and malignant. As we use low-resolution images, we simplify the task into binary
classification by combining normal and benign as positive and classifying them against
malignant as negative. We split the source dataset with a ratio of 7:1:2 into training,
validation and test set. The source images of 1×500×500 are resized into 1×28×28.', 'url': '
https://zenodo.org/record/6496656/files/breastmnist.npz?download=1', 'MD5': '750601
b1f35ba3300ea97c75c52ff8f6', 'task': 'binary-class', 'label': {'0': 'malignant', '1': 'normal

```

```

, benign'}, 'n_channels': 1, 'n_samples': {'train': 546, 'val': 78, 'test': 156}, 'license':
'CC BY 4.0'}
4
5 {'python_class': 'BreastMNIST',
6  'description': 'The BreastMNIST is based on a dataset of 780 breast ultrasound images. It is
   categorized into 3 classes: normal, benign, and malignant. As we use low-resolution images,
   we simplify the task into binary classification by combining normal and benign as positive
   and classifying them against malignant as negative. We split the source dataset with a ratio
   of 7:1:2 into training, validation and test set. The source images of 1×500×500 are resized
   into 1×28×28.',
7  'url': 'https://zenodo.org/record/6496656/files/breastmnist.npz?download=1',
8  'MD5': '750601b1f35ba3300ea97c75c52ff8f6',
9  'task': 'binary-class',
10 'label': {'0': 'malignant', '1': 'normal, benign'},
11 'n_channels': 1,
12 'n_samples': {'train': 546, 'val': 78, 'test': 156},
13 'license': 'CC BY 4.0'}
14
15 Number of channels of the image: 1
16
17 Dataset BreastMNIST (breastmnist)
18   Number of datapoints: 546
19   Root location: /root/.medmnist
20   Split: train
21   Task: binary-class
22   Number of channels: 1
23   Meaning of labels: {'0': 'malignant', '1': 'normal, benign'}
24   Number of samples: {'train': 546, 'val': 78, 'test': 156}
25   Description: The BreastMNIST is based on a dataset of 780 breast ultrasound images. It is
   categorized into 3 classes: normal, benign, and malignant. As we use low-resolution images,
   we simplify the task into binary classification by combining normal and benign as positive
   and classifying them against malignant as negative. We split the source dataset with a ratio
   of 7:1:2 into training, validation and test set. The source images of 1×500×500 are resized
   into 1×28×28.
26   License: CC BY 4.0
27   =====
28 Dataset BreastMNIST (breastmnist)
29   Number of datapoints: 156
30   Root location: /root/.medmnist
31   Split: test
32   Task: binary-class
33   Number of channels: 1
34   Meaning of labels: {'0': 'malignant', '1': 'normal, benign'}
35   Number of samples: {'train': 546, 'val': 78, 'test': 156}

```



```

36 Description: The BreastMNIST is based on a dataset of 780 breast ultrasound images. It is
    categorized into 3 classes: normal, benign, and malignant. As we use low-resolution images,
    we simplify the task into binary classification by combining normal and benign as positive
    and classifying them against malignant as negative. We split the source dataset with a ratio
    of 7:1:2 into training, validation and test set. The source images of 1×500×500 are resized
    into 1×28×28.
37 License: CC BY 4.0

```

در ادامه دستوراتی را هم برای نمایش تصویری اطلاعات مجموعه داده می نویسیم. این دستورات برای نمایش تصاویر از کلاس های مختلف به همراه برچسب های کلاس و شاخص های تصویر نوشته شده است. ابتدا یک شکل با تعداد ردیف ها و ستون های مشخص ایجاد می شود و سپس برای هر تصویر از مجموعه داده آموزش، برچسب کلاس و تصویر مربوطه استخراج می شود. سپس تصویر در زیر شکل مربوطه نمایش داده می شود و برچسب کلاس و شاخص تصویر بالای تصویر نوشته می شود. در نهایت، شکل نمایش داده شده و در یک فایل PDF ذخیره می شود. علاوه بر این، دستوراتی برای محاسبه و نمایش تعداد تصاویر موجود در هر کلاس استفاده می شود. در ابتدا برچسب ها را از مجموعه داده آموزش دریافت می کنیم. سپس تعداد کل تصاویر را محاسبه کرده و به عنوان total_images ذخیره می کنیم. با استفاده از مجموع برچسب های برابر با ۱، تعداد تصاویری که برچسب ۱ دارند را محاسبه می کنیم و در متغیر class_1_images ذخیره می کنیم. با کم کردن class_1_images از total_images، تعداد تصاویری که برچسب ۰ دارند را به دست می آوریم و در متغیر class_0_images ذخیره می کنیم. سپس تعداد تصاویر در هر کلاس را چاپ می کنیم. در نهایت، نموداری با نمایش توزیع تصاویر در هر کلاس ایجاد می شود و در یک فایل PDF ذخیره می شود. دستورات در برنامه ۳ و نتایج در شکل ۱ و شکل ۲ آورده شده است.

Program 3: Display Data

```

1 # Display an image from a specific class with class label written above the image
2 import matplotlib.pyplot as plt
3
4 # Get the index and class label of the image
5 index = 1
6 class_label = train_dataset[index][1][0]
7
8 # Reshape and visualize the image
9 image = train_dataset[index][0].reshape((28, 28))
10
11 # Create a figure and axes
12 fig, ax = plt.subplots()
13
14 # Display the image
15 ax.imshow(image, cmap='gray')
16
17 # Write the class label above the image
18 ax.set_title(f"Class: {class_label}", fontsize=12, pad=10)
19
20 # Remove the axis ticks
21 ax.set_xticks([])
22 ax.set_yticks([])

```

```

23
24 # Save the figure as a PDF file with index and class label in the filename
25 filename = f"image_{index}_class_{class_label}.pdf"
26 plt.savefig(filename, format='pdf')
27
28 # Show the plot
29 plt.show()
30
31
32 # Display an image from a specific class with class label written above the image
33 import matplotlib.pyplot as plt
34
35 # Get the index and class label of the image
36 index = 200
37 class_label = train_dataset[index][1][0]
38
39 # Reshape and visualize the image
40 image = train_dataset[index][0].reshape((28, 28))
41
42 # Create a figure and axes
43 fig, ax = plt.subplots()
44
45 # Display the image
46 ax.imshow(image, cmap='gray')
47
48 # Write the class label above the image
49 ax.set_title(f"Class: {class_label}", fontsize=12, pad=10)
50
51 # Remove the axis ticks
52 ax.set_xticks([])
53 ax.set_yticks([])
54
55 # Save the figure as a PDF file with index and class label in the filename
56 filename = f"image_{index}_class_{class_label}.pdf"
57 plt.savefig(filename, format='pdf')
58
59 # Show the plot
60 plt.show()
61
62
63 # Display images from different classes with class labels and image indices written above the
    images in a 5x3 grid
64
65 import matplotlib.pyplot as plt
66

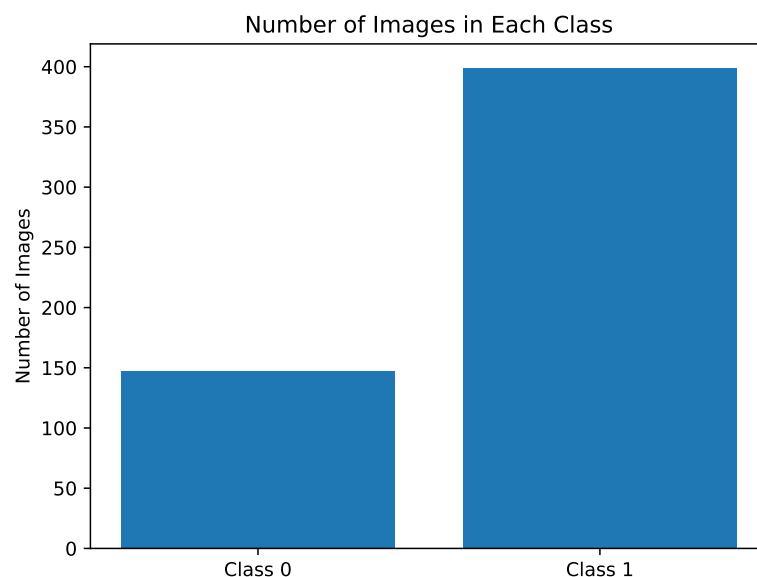
```

```

67 # Set the number of rows and columns for the grid
68 num_rows = 5
69 num_cols = 5
70
71 # Create a figure and axes
72 fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 10))
73
74 # Iterate over the images and their class labels
75 for i in range(num_rows * num_cols):
76     # Get the class label of the image
77     class_label = train_dataset[i][1][0]
78
79     # Reshape the image
80     image = train_dataset[i][0].reshape((28, 28))
81
82     # Determine the row and column index for the current subplot
83     row = i // num_cols
84     col = i % num_cols
85
86     # Display the image in the corresponding subplot
87     axes[row, col].imshow(image, cmap='gray')
88     axes[row, col].set_title(f"Image: {i}\nClass: {class_label}", fontsize=10, pad=4)
89     axes[row, col].axis('off')
90
91 # Adjust the spacing between subplots
92 fig.tight_layout()
93
94 # Save the figure as a PDF file
95 plt.savefig('images_with_class_labels.pdf', format='pdf')
96
97 # Show the plot
98 plt.show()
99
100
101 # Calculate and display the number of images in each class
102
103 # Get the labels from the train_dataset
104 labels = train_dataset.labels
105
106 # Calculate the total number of images
107 total_images = len(labels)
108
109 # Calculate the number of images labeled as '1'
110 class_1_images = np.sum(labels)
111

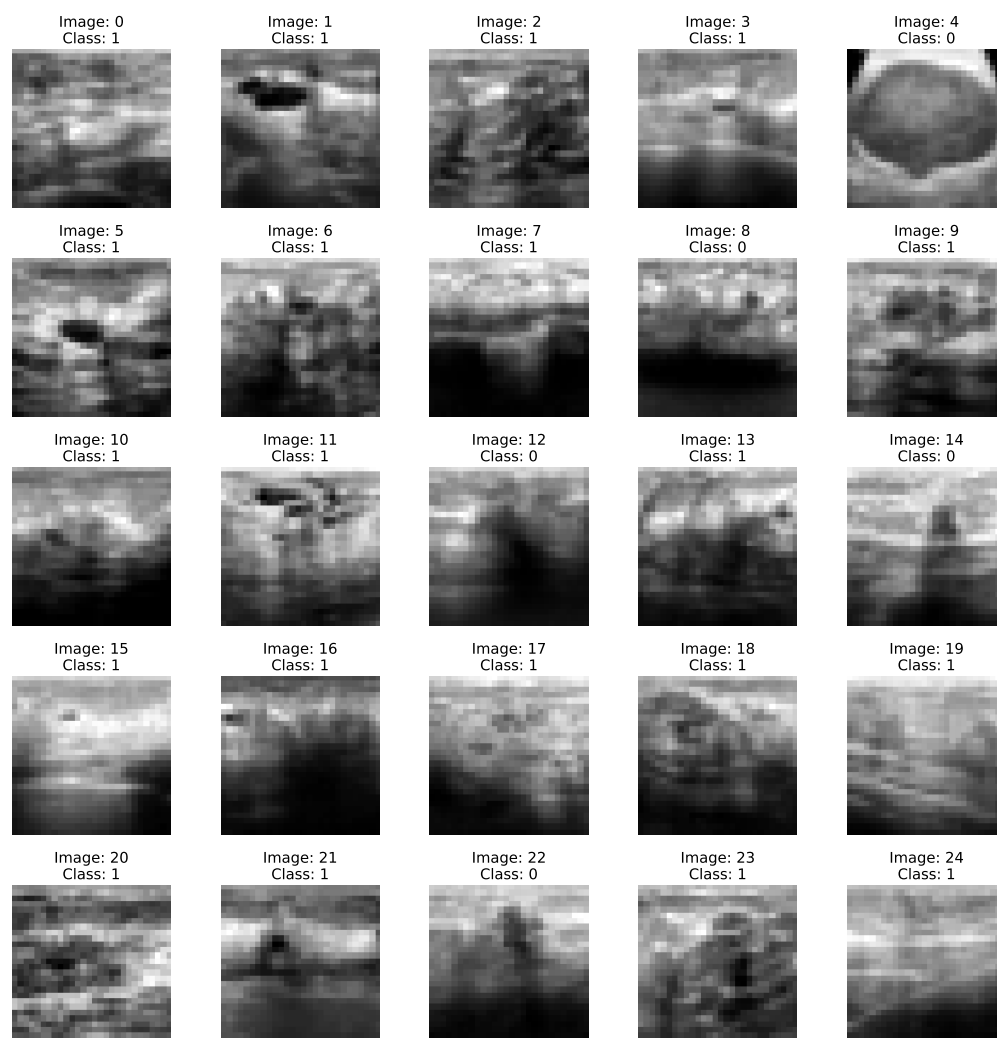
```

```
112 # Calculate the number of images labeled as '0'
113 class_0_images = total_images - class_1_images
114
115 # Display the number of images in each class
116 print("Number of images in class '0':", class_0_images)
117 print("Number of images in class '1':", class_1_images)
118
119 # Save the results as a PDF file
120 fig, ax = plt.subplots()
121 ax.bar(['Class 0', 'Class 1'], [class_0_images, class_1_images])
122 ax.set_ylabel('Number of Images')
123 ax.set_title('Number of Images in Each Class')
124
125 plt.savefig('class_distribution.pdf', format='pdf')
126
127 # Show the plot
128 plt.show()
129
130
131 # montage
132
133 train_dataset.montage(length=20)
```



شکل ۱: نمایش توزیع داده‌ها و کلاس‌های آن‌ها.

بعد از کنکاش در داده‌ها به سراغ پیاده‌سازی موردخواست سوال می‌رویم. در این قسمت هم ابتدا کتابخانه‌های ضروری



شکل ۲: نمایش تعدادی از داده‌ها و کلاس‌های آن‌ها.

را فراخوانی می‌کنیم. سپس، نام‌ها و متغیرهای مختلفی را برای استفاده در فرآیند آموزش و ارزیابی تعریف می‌کنیم. به عنوان مثال، نام مجموعه‌داده، پارامترهای آموزش مانند تعداد دوره‌ها و اندازه‌دسته و نرخ یادگیری برای بهینه‌ساز. این دستورات شامل تعدادی تبدیل پیش‌پردازش برای داده‌های ورودی است. به طور خاص، داده‌ها به اندازه ۲۲۴ در ۲۲۴ تغییر اندازه می‌شوند، و به صورت فیک فرمت RGB تبدیل می‌شوند و سپس به تانسورها تبدیل می‌شوند. همچنین، داده‌ها نرمال‌سازی می‌شوند. در ادامه، مجموعه‌های آموزش و ارزیابی برای آموزش و ارزیابی مدل بارگیری می‌شوند. در ادامه مدل ResNet-۵۰ پیش‌آموزش‌دیده بارگیری و فراخوانی می‌شود و لایه تماماًمتصل آخر مدل با توجه به تعداد کلاس‌ها در مجموعه‌داده (۲) جایگزین می‌شود. تابع هزینه CrossEntropy و بهینه‌ساز Adam برای آموزش مدل تعریف می‌شوند. برای فرآیند آموزش شبکه یک حلقه نوشته شده است. این حلقه شامل تقسیم داده به دسته‌های مختلف، محاسبه خروجی‌ها و هزینه، محاسبه میانگین هزینه و دقت، به‌روزرسانی وزن‌ها و نمایش پیشرفت آموزش است. در حلقه ارزیابی، مدل با استفاده از مجموعه مختص خود ارزیابی می‌شود. این حلقه شامل محاسبه خروجی‌ها و هزینه‌ها، محاسبه میانگین هزینه و دقت است. در نهایت معیارهایی مانند میانگین هزینه و دقت آموزش و ارزیابی محاسبه می‌شوند و در لیست‌های مربوطه ذخیره می‌شوند و نمودارهای مربوطه رسم می‌شوند. با این

توضیحات، دستورات مربوطه در برنامه ۴ آورده شده است و نتایج در برنامه ۵، شکل ۳ و شکل ۴ آورده شده است.

Program 4: ResNet on Data Implementation

```

1 import numpy as np
2 import torch
3 import torch.nn as nn
4 import torch.optim as optim
5 import torch.utils.data as data
6 import torchvision.transforms as transform
7 from matplotlib import pyplot as plt
8 from tqdm import tqdm
9 from torchvision import transforms
10
11 # Define variable names and add comments
12
13 dataset_name = "breastmnist" # The name of the dataset
14 download = True # Flag to download the dataset if not available locally
15 dataset_name = dataset_name.lower() # Convert the dataset name to lowercase
16
17 NUM_EPOCHS = 5 # Number of epochs for training
18 BATCH_SIZE = 128 # Batch size for data loading
19 lr = 0.0001 # Learning rate for the optimizer
20
21 info = INFO[dataset_name] # Retrieve information about the dataset
22 task = info['task'] # The task associated with the dataset
23
24 DataClass = getattr(medmnist, info['python_class']) # Get the data class for the dataset
25
26 # Data preprocessing transforms
27 data_transform = transforms.Compose([
28     transforms.Resize(224), # Resize the image to 224x224
29     transforms.Grayscale(3), # Convert the image to RGB format
30     transforms.ToTensor(), # Convert data to tensors
31     transforms.Normalize(mean=[.5], std=[.5]) # Normalize the data
32 ])
33
34 # Load the training and testing datasets
35 train_dataset = DataClass(split='train', transform=data_transform, download=download)
36 test_dataset = DataClass(split='test', transform=data_transform, download=download)
37 val_dataset = DataClass(split='val', transform=data_transform, download=download)
38 pil_dataset = DataClass(split='train', download=download) # Load the PIL dataset for
    visualization (if needed)
39
40 # Create data loaders for training and testing datasets
41 train_loader = data.DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
42 val_loader = data.DataLoader(dataset=val_dataset, batch_size=78, shuffle=False)

```

```

43 test_loader = data.DataLoader(dataset=test_dataset, batch_size=156, shuffle=False)
44
45 # Import ResNet-50 model
46 import torchvision.models as models
47
48 # Load the pre-trained ResNet-50 model
49 model = models.resnet50(pretrained=True)
50
51 # Replace the last fully connected layer to match the number of output classes
52 num_classes = len(info['label'])
53 model.fc = nn.Linear(model.fc.in_features, num_classes)
54
55 # Define the loss function and optimizer
56 criterion = nn.CrossEntropyLoss()
57 optimizer = optim.Adam(model.parameters(), lr=lr)
58
59 # Training loop
60 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
61 model.to(device)
62
63 train_losses = []
64 val_losses = []
65 train_accs = []
66 val_accs = []
67
68 for epoch in range(NUM_EPOCHS):
69     train_loss = 0.0
70     val_loss = 0.0
71     train_total = 0
72     train_correct = 0
73     val_total = 0
74     val_correct = 0
75
76     # Training
77     model.train()
78     for images, labels in tqdm(train_loader, desc=f'Epoch {epoch+1}/{NUM_EPOCHS} - Training'):
79         images = images.to(device)
80         labels = labels.squeeze().to(device) # Convert multi-target labels to single-valued
81         labels
82
83         optimizer.zero_grad()
84
85         outputs = model(images)
86         loss = criterion(outputs, labels)
87         loss.backward()

```

```

87     optimizer.step()
88
89     train_loss += loss.item() * images.size(0)
90
91     _, predicted = torch.max(outputs.data, 1)
92     train_total += labels.size(0)
93     train_correct += (predicted == labels).sum().item()
94
95     # Validation
96     model.eval()
97     with torch.no_grad():
98         for images, labels in tqdm(val_loader, desc=f'Epoch {epoch+1}/{NUM_EPOCHS} - Validation'):
99             images = images.to(device)
100             labels = labels.squeeze().to(device) # Convert multi-target labels to single-valued
101             labels
102
103             outputs = model(images)
104             loss = criterion(outputs, labels)
105
106             val_loss += loss.item() * images.size(0)
107
108             _, predicted = torch.max(outputs.data, 1)
109             val_total += labels.size(0)
110             val_correct += (predicted == labels).sum().item()
111
112     # Calculate metrics
113     train_loss /= len(train_loader.dataset)
114     val_loss /= len(val_loader.dataset)
115     train_accuracy = train_correct / train_total
116     val_accuracy = val_correct / val_total
117
118     train_losses.append(train_loss)
119     val_losses.append(val_loss)
120     train_accs.append(train_accuracy)
121     val_accs.append(val_accuracy)
122
123     print(f'Epoch {epoch+1}/{NUM_EPOCHS} - Training Loss: {train_loss:.4f} - Training Accuracy: {train_accuracy:.4f} - Validation Loss: {val_loss:.4f} - Validation Accuracy: {val_accuracy:.4f}')
124
125 # Plot loss and accuracy
126 plt.figure(figsize=(10, 5))
127 plt.plot(train_losses, label='Training Loss')
128 plt.plot(val_losses, label='Validation Loss')

```



```

128 plt.xlabel('Epoch')
129 plt.ylabel('Loss')
130 plt.legend()
131 plt.title('Training and Validation Loss')
132 plt.savefig('loss_plot.pdf')
133
134 plt.figure(figsize=(10, 5))
135 plt.plot(train_accs, label='Training Accuracy')
136 plt.plot(val_accs, label='Validation Accuracy')
137 plt.xlabel('Epoch')
138 plt.ylabel('Accuracy')
139 plt.legend()
140 plt.title('Training and Validation Accuracy')
141 plt.savefig('accuracy_plot.pdf')
142
143 # Classification report on the test split
144 model.eval()
145 test_total = 0
146 test_correct = 0
147 test_predictions = []
148 test_targets = []
149
150 with torch.no_grad():
151     for images, labels in tqdm(test_loader, desc='Test'):
152         images = images.to(device)
153         labels = labels.squeeze().to(device) # Convert multi-target labels to single-valued
154         labels
155
156         outputs = model(images)
157
158         _, predicted = torch.max(outputs.data, 1)
159         test_total += labels.size(0)
160         test_correct += (predicted == labels).sum().item()
161
162         test_predictions.extend(predicted.cpu().numpy())
163         test_targets.extend(labels.cpu().numpy())
164
165 test_accuracy = test_correct / test_total
166 print(f'Test Accuracy: {test_accuracy:.4f}')
167
168 # Print classification report
169 from sklearn.metrics import classification_report
170 target_names = info['label']
171 classification_rep = classification_report(test_targets, test_predictions, target_names=
    target_names)

```

```

171 print(classification_rep)
172
173 # Plot confusion matrix
174 from sklearn.metrics import confusion_matrix
175 import seaborn as sns
176
177 cm = confusion_matrix(test_targets, test_predictions)
178 plt.figure(figsize=(10, 8))
179 sns.heatmap(cm, annot=True, fmt='d', xticklabels=target_names, yticklabels=target_names)
180 plt.xlabel('Predicted')
181 plt.ylabel('True')
182 plt.title('Confusion Matrix')
183 plt.savefig('confusion_matrix_plot.pdf')

```

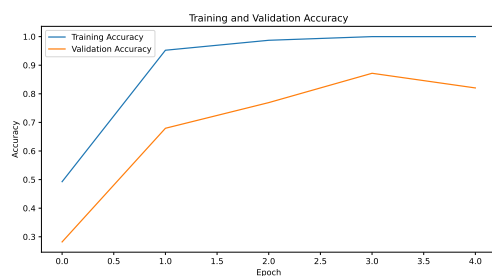
Program 5: ResNet on Data Results

```

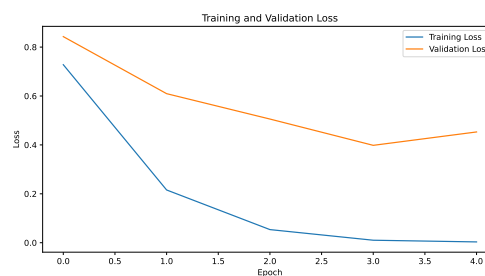
1 Test Accuracy: 0.7500
2
3
4
5
6
7
8
9

```

	precision	recall	f1-score	support
0	0.53	0.67	0.59	42
1	0.86	0.78	0.82	114
accuracy			0.75	156
macro avg	0.70	0.72	0.70	156
weighted avg	0.77	0.75	0.76	156



(ب) دقت

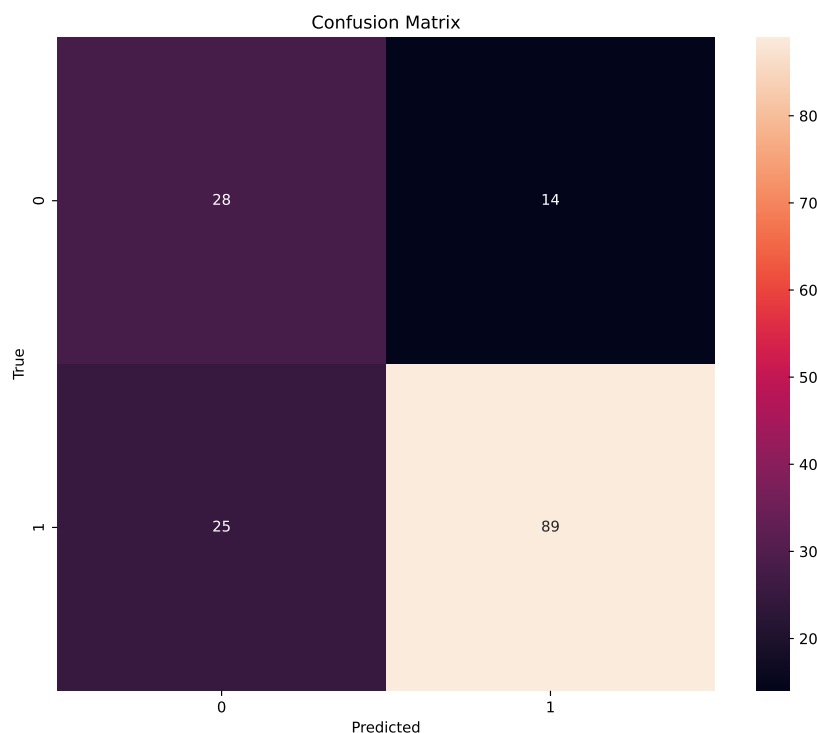


(آ) اتلاف

شکل ۳: نمودار تابع اتلاف و دقت مدل در حالت عادی.

۲.۱ پاسخ قسمت ۲ - شبکه‌ی Conditional DCGAN

برای پیاده‌سازی اهداف مدنظر سوال در این قسمت، پس از فراخوانی کتابخانه‌های ضروری، داده‌ها را بارگیری و پیش‌پردازش می‌کنیم. بدین‌منظور دستوراتی را نوشته‌ایم که یک نمونه بارگیری و پیش‌پردازش داده‌ها را با استفاده از MedMNIST dataset انجام می‌دهد. در این کد، ابتدا مسیر ریشه برای ذخیره داده‌ها تعیین می‌شود و سپس اگر مسیر ریشه وجود نداشته باشد، آن را



شکل ۴: ماتریس درهم‌ریختگی در حالت عادی.

ایجاد می‌کند. سپس نام مجموعه داده تعیین شده و اطلاعات مربوط به آن دریافت می‌شود. سپس کلاس داده مجموعه داده مورد نیاز از ماژول `medmnist` استخراج می‌شود. در این کد، یک کلاس به نام `MedMNISTDataset` تعریف می‌شود که از کلاس مجموعه داده بهره می‌برد و تابع `getitem` را بازنویسی می‌کند. در این تابع، تصویر و برچسب مربوط به نمونه مورد نظر برگردانده می‌شود. این کلاس برای مناسب‌سازی شکل و ابعاد داده‌ها و برچسب‌ها اضافه شده است. سپس تبدیلات پیش‌پردازش داده‌ها تعریف می‌شود و مجموعه داده‌های آموزشی و آزمون بارگیری می‌شوند. همچنین، یک مجموعه داده از نوع `PIL` نیز برای استفاده در تصویرسازی (به صورت نمایشی) بارگیری می‌شود. در انتها، دیتالودرهایی برای مجموعه داده‌های آموزشی و آزمون ساخته می‌شوند، که می‌توانند برای آموزش و ارزیابی الگوریتم‌ها استفاده شوند. در انتهای این قسمت دستوراتی برای نمایش ابعاد و شمایل داده‌ها نوشته شده است. دستورات مربوطه در برنامه ۶ آورده شده و نتیجه در برنامه ۷ نوشته شده است.

Program 6: Load Data

```
1 import medmnist
2 from torchvision import transforms
3 from torch.utils.data import DataLoader
4 import os
5
6 # Define the root directory for saving the data
7 root_dir = '/content/data/MedMNIST' # Specify the desired root directory path
8
```

```

9 # Create the root directory if it does not exist
10 os.makedirs(root_dir, exist_ok=True)
11
12 dataset_name = "breastmnist" # The name of the dataset
13 download = True # Flag to download the dataset if not available locally
14 dataset_name = dataset_name.lower() # Convert the dataset name to lowercase
15
16 info = INFO[dataset_name] # Retrieve information about the dataset
17 task = info['task'] # The task associated with the dataset
18
19 DataClass = getattr(medmnist, info['python_class']) # Get the data class for the dataset
20
21 class MedMNISTDataset(DataClass):
22     def __getitem__(self, index):
23         image, label = super().__getitem__(index)
24         return image, label.item()
25
26 # Data preprocessing transforms
27 data_transform = transforms.Compose([
28     # transforms.Resize(224),
29     # transforms.Lambda(lambda image: image.convert('RGB')),
30     transforms.ToTensor(), # Convert data to tensors
31     transforms.Normalize((0.5,), (0.5,))
32 ])
33
34 # Load the training and testing datasets
35 train_dataset = MedMNISTDataset(split='train', transform=data_transform, download=download, root=
    root_dir)
36 test_dataset = MedMNISTDataset(split='test', transform=data_transform, download=download, root=
    root_dir)
37 val_dataset = MedMNISTDataset(split='val', transform=data_transform, download=download, root=
    root_dir)
38 pil_dataset = DataClass(split='train', download=download, root=root_dir) # Load the PIL dataset
    for visualization (if needed)
39
40 # Create data loaders for training and testing datasets
41 train_dataloader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
42 val_dataloader = DataLoader(dataset=val_dataset, batch_size=2*BATCH_SIZE, shuffle=False)
43 test_dataloader = DataLoader(dataset=test_dataset, batch_size=2*BATCH_SIZE, shuffle=False)
44
45
46 import torch
47 from torchvision import datasets, transforms
48 from torch.utils.data import DataLoader
49

```

```

50 # Define the transformation
51 transform = transforms.Compose([
52     transforms.ToTensor(),
53     transforms.Normalize((0.5,), (0.5,))
54 ])
55
56 # Print a sample of data and label from train_dataset
57 print("Sample from train_dataset:")
58 data, label = train_dataset[0]
59 print("Data shape:", data.shape)
60 print("Label:", label)
61
62 # Create the train_dataloader
63 train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
64
65 # Print a sample of data and label from train_dataloader
66 print("\nSample from train_dataloader:")
67 data_batch, label_batch = next(iter(train_dataloader))
68 print("Data batch shape:", data_batch.shape)
69 print("Label batch shape:", label_batch.shape)
70 print("Data batch[0] shape:", data_batch[0].shape)
71 print("Label batch[0]:", label_batch[0])

```

Program 7: Data Info

```

1 Sample from train_dataset:
2 Data shape: torch.Size([1, 28, 28])
3 Label: 1
4
5 Sample from train_dataloader:
6 Data batch shape: torch.Size([64, 1, 28, 28])
7 Label batch shape: torch.Size([64])
8 Data batch[0] shape: torch.Size([1, 28, 28])
9 Label batch[0]: tensor(0)

```

حالا نوبت به تشکیل کلاس مولد یا جنریتور می‌رسد. برای این منظور کلاس Generator را تعریف می‌کنیم که از کلاس `nn.Module` ارث‌بری می‌کند. در ادامه، لایه‌های مختلف شبکه مولد را تعریف می‌کنیم. ابتدا یک لایه خطی (`linear_layer`) را تعریف می‌کنیم که یک ورودی با ابعاد (`batch_size, 102`) را به یک ورودی با ابعاد (`batch_size, 77128`) تبدیل می‌کند. این لایه شامل یک لایه خطی و یک تابع فعال‌سازی `LeakyReLU` است. در ادامه شبکه‌ای شامل شامل چندین لایه `Upsampling` تعریف می‌کنیم. این بخش شبکه مولد اصلی را تعریف می‌کند. هر لایه `Upsampling` شامل یک لایه `ConvTranspose2d`، یک لایه `Batch Normalization`، و یک تابع فعال‌سازی `ReLU` است. لایه آخر از نوع `ConvTranspose2d` است و یک تابع فعال‌سازی `Tanh` دارد. تابع `forward` هم برای شبکه مولد را تعریف می‌کنیم. در این تابع، ورودی شبکه (`input`) و برچسب‌ها (`label`) را در ابعاد ۱ ادغام می‌کنیم (`concatenate`) و در متغیر `concatenated_input` ذخیره می‌کنیم. سپس این ورودی ترکیب‌شده را از طریق لایه `linear_layer` می‌گذرانیم و نتیجه را در `linear_output` ذخیره می‌کنیم. سپس خروجی را تغییر

شکل می‌دهیم (reshaped_output) تا بتوانیم آن را به لایه‌های Upsampling اعمال کنیم. سرانجام خروجی را از طریق شبکه مولد اصلی (self.main) عبور می‌دهیم و خروجی نهایی را برمی‌گردانیم. در ادامه، تعدادی نمونه تصادفی را برای تست ایجاد می‌کنیم. این بخش دو ماتریس تصادفی با ابعاد (64, 100) و (64, 2) را ایجاد می‌کند. سپس یک نمونه از کلاس Generator ایجاد می‌شود. یک گذر رو به جلو از شبکه مولد انجام می‌شود. نمونه‌های تصادفی ورودی و برجسب‌ها را به شبکه مولد می‌دهیم و خروجی را در متغیر output ذخیره می‌کنیم. در نهایت شکل (shape) خروجی را چاپ می‌کنیم. دستورات مربوط به این کلاس در برنامه ۹ آورده شده است.

Program 8: Generator Class

```
1 import torch
2 import torch.nn as nn
3
4 class Generator(nn.Module):
5     def __init__(self):
6         super(Generator, self).__init__()
7
8         # Defining the linear layer
9         self.linear_layer = nn.Sequential(nn.Linear(102, 7*7*128), nn.LeakyReLU())
10
11        # Defining the main sequential module
12        self.main = nn.Sequential(
13            # Upsampling layer 1
14            nn.ConvTranspose2d(128, 20, kernel_size=5), # Output size: 11x11
15            nn.BatchNorm2d(20),
16            nn.ReLU(True),
17
18            # Upsampling layer 2
19            nn.ConvTranspose2d(20, 20, kernel_size=4), # Output size: 14x14
20            nn.BatchNorm2d(20),
21            nn.ReLU(True),
22
23            # Upsampling layer 3
24            nn.ConvTranspose2d(20, 20, kernel_size=4), # Output size: 17x17
25            nn.BatchNorm2d(20),
26            nn.ReLU(True),
27
28            # Upsampling layer 4
29            nn.ConvTranspose2d(20, 20, kernel_size=4), # Output size: 20x20
30            nn.BatchNorm2d(20),
31            nn.ReLU(True),
32
33            # Upsampling layer 5
34            nn.ConvTranspose2d(20, 20, kernel_size=4), # Output size: 23x23
35            nn.BatchNorm2d(20),
36            nn.ReLU(True),
```

```
37
38     # Upsampling layer 6
39     nn.ConvTranspose2d(20, 20, kernel_size=3), # Output size: 25x25
40     nn.BatchNorm2d(20),
41     nn.ReLU(True),
42
43     # Final output layer
44     nn.ConvTranspose2d(20, 1, kernel_size=4), # Output size: 28x28
45     nn.Tanh()
46 )
47
48 def forward(self, input, label):
49     # Concatenating input and label along dimension 1
50     concatenated_input = torch.cat([input, label], dim=1)
51     # Input shape: 102
52
53     # Applying linear layer
54     linear_output = self.linear_layer(concatenated_input) # Output shape: 6,272
55
56     # Reshaping the linear output
57     reshaped_output = linear_output.view(linear_output.size(0), 128, 7, 7)
58
59     # Forward pass through the main sequential module
60     output = self.main(reshaped_output)
61
62     return output
63
64
65 # Creating random tensors for testing
66 random_input = torch.rand(size=(64, 100))
67 random_label = torch.rand(size=(64, 2))
68
69 # Creating an instance of the Generator class
70 generator = Generator()
71
72 # Forward pass through the generator
73 output = generator(random_input, random_label)
74
75 # Shape of the output
76 print(output.shape)
77 -----
78 torch.Size([64, 1, 28, 28])
```

در ادامه کلاس شبکه تمیزدهنده (Discriminator) در چارچوب پایتورچ را تعریف می‌کنیم. شبکه تمیزدهنده با دریافت تصاویر و برچسب‌ها، سعی می‌کند تشخیص دهد که تصاویر ارائه شده واقعی هستند یا از شبکه مولد مصنوعی تولید شده‌اند.

این کلاس از کلاس `nn.Module` ارث‌بری می‌کند. در ابتدا یک لایه خطی (`linear_layer`) تعریف می‌کنیم که یک ورودی با ابعاد `(batch_size, 2)` را به یک ورودی با ابعاد `(batch_size, 7716)` تبدیل می‌کند. این لایه شامل یک لایه خطی و یک تابع فعال‌سازی `LeakyReLU` است. سپس بخش شبکه تمییزدهنده اصلی را تعریف می‌کنیم. این شبکه شامل چندین لایه کانولوشنی است. هر لایه کانولوشن شامل یک لایه `Conv2d` و یک تابع فعال‌سازی `LeakyReLU` است. لایه آخر از نوع `Conv2d` است و یک تابع فعال‌سازی `Sigmoid` دارد. در ادامه یک لایه `Flatten` را تعریف می‌کنیم که برای تغییر شکل دادن به خروجی شبکه قبل از لایه کاملاً متصل استفاده می‌شود. تابع `forward` در ادامه تعریف می‌شود. این تابع وظیفه گذر رو به جلو (`forward pass`) اطلاعات را از طریق شبکه تمییزدهنده انجام می‌دهد. در این تابع، برچسب‌ها (`label`) را از طریق لایه `linear_layer` می‌گذرانیم و خروجی را به شکل یک تنسور به ابعاد `(batch_size, 1, 28, 28)` تغییر شکل می‌دهیم. سپس تصاویر و برچسب‌ها را در ابعاد ۱ ادغام می‌کنیم و در متغیر `concatenated_input` ذخیره می‌کنیم. سپس این ورودی ترکیب‌شده را از طریق شبکه تمییزدهنده اصلی (`self.main`) می‌گذرانیم و خروجی را در متغیر `output` ذخیره می‌کنیم. سرانجام خروجی را با استفاده از لایه `Flatten` به یک بردار یک بعدی تغییر شکل می‌دهیم و برمی‌گردانیم. در ادامه، تنسورهای تصادفی برای تست ایجاد می‌کنیم. این بخش دو تنسور تصادفی با ابعاد `(64, 2)` و `(64, 1, 28, 28)` را ایجاد می‌کند. سپس یک نمونه از کلاس `Discriminator` ایجاد می‌شود و یک گذر رو به جلو از شبکه تمییزدهنده انجام می‌شود. سپس، نمونه‌های تصادفی تصویر و برچسب به عنوان ورودی به شبکه تمییزدهنده داده می‌شود و خروجی را در متغیر `output` ذخیره می‌کنیم و شکل آن را چاپ می‌کنیم. دستورات مربوط به این کلاس در؟؟ آورده شده است.

Program 9: Discriminator Class

```

1 import torch
2 import torch.nn as nn
3
4 class Discriminator(nn.Module):
5     def __init__(self):
6         super(Discriminator, self).__init__()
7
8         # Defining the linear layer
9         self.linear_layer = nn.Sequential(nn.Linear(2, 7*7*16), nn.LeakyReLU())
10
11        # Defining the main sequential module
12        self.main = nn.Sequential(
13            # Convolutional layer 1
14            nn.Conv2d(2, 20, kernel_size=3), # Input size: 26x26
15            nn.LeakyReLU(0.2, inplace=True),
16
17            # Convolutional layer 2
18            nn.Conv2d(20, 30, kernel_size=3), # Input size: 24x24
19            nn.LeakyReLU(0.2, inplace=True),
20
21            # Max pooling layer 1
22            nn.MaxPool2d(2), # Output size: 12x12
23
24            # Convolutional layer 3
25            nn.Conv2d(30, 30, kernel_size=3), # Input size: 10x10

```



```

26         nn.LeakyReLU(0.2, inplace=True),
27
28         # Max pooling layer 2
29         nn.MaxPool2d(2), # Output size: 5x5
30
31         # Convolutional layer 4
32         nn.Conv2d(30, 20, kernel_size=3), # Input size: 3x3
33         nn.LeakyReLU(0.2, inplace=True),
34
35         # Convolutional layer 5
36         nn.Conv2d(20, 1, kernel_size=3), # Input size: 1x1
37         nn.Sigmoid()
38     )
39
40     self.flatten = nn.Flatten()
41
42     def forward(self, images, label):
43         # Applying linear layer and reshaping the output
44         reshaped_label = self.linear_layer(label).view(label.size(0), 1, 28, 28)
45
46         # Concatenating images and label along dimension 1
47         concatenated_input = torch.cat([images, reshaped_label], dim=1)
48
49         # Forward pass through the main sequential module
50         output = self.main(concatenated_input)
51
52         # Flattening the output tensor
53         flattened_output = self.flatten(output)
54
55         return flattened_output
56
57
58 # Creating random tensors for testing
59 label = torch.rand(size=(64, 2))
60 images = torch.rand(size=(64, 1, 28, 28))
61
62 # Creating an instance of the Discriminator class
63 discriminator = Discriminator()
64
65 # Forward pass through the discriminator
66 output = discriminator(images, label)
67
68 # Shape of the output
69 print(output.shape)
70 -----

```

```
71 torch.Size([64, 1])
```

در ادامه دستوراتی برای آموزش مدل کلی می‌نویسیم. در این دستورات یک کلاس آموزش‌دهنده تعریف می‌کنیم که در تابع و متد `init` خود متغیرهای مختلفی را برای شیء کلاس `Trainer` را مقداردهی اولیه می‌کند. در ابتدا مشخص می‌شود که مدل روی چه دستگاهی قرار خواهد گرفت. سپس تعداد دوره‌های آموزشی، اندازه دسته، مسیر ذخیره‌سازی مدل آموزش‌شده، مجموعه داده‌ها، مسیر اوزان پیش‌آموزش‌دیده و غیره تعیین می‌شود. درواقع در متد `init`، مقادیر پیش‌فرض برای پارامترها تعیین شده‌اند و اگر مقداری برای آنها در دسترس باشد، از مقدار داده‌شده استفاده می‌شود. متغیرهای مهمی که در این کلاس تعریف شده‌اند عبارتند از: `device` (دستگاهی که برای آموزش استفاده می‌شود)، `epochs` (تعداد دوره‌های آموزش)، `batch_size` (اندازه دسته‌های آموزش)، `save_dir` (مسیری که مدل آموزش‌دیده در آن ذخیره می‌شود)، `train_loader` و `valid_loader` (بارگذاری داده‌های آموزش و اعتبارسنجی)، `weights` (مسیر وزن‌های مدل پیش‌آموزش‌دیده)، `verbose` (سطح نمایش در حین آموزش)، `visualize_plots` و `save_plots` (نمایش و ذخیره نمودارهای آموزش)، `model_name` (نام مدل) و `optimizer` (نام بهینه‌ساز برای استفاده در آموزش). در ادامه، مقادیر این پارامترها به متغیرهای مربوطه در کلاس نسبت داده می‌شوند. سپس، یک مسیر ذخیره‌سازی منحصربه‌فرد برای مدل ایجاد می‌شود تا در صورت وجود یک دایرکتوری با همان نام، نام‌های متفاوت استفاده شود. سپس، بارگذاری داده‌های آموزش و اعتبارسنجی صورت می‌گیرد. مدل آموزش‌دیده و بهینه‌سازها ساخته و در متغیرهای مربوطه ذخیره می‌شوند. در نهایت، یک شیء `SummaryWriter` برای ثبت و نمایش داده‌های تانسوربرد ایجاد می‌شود. متد `get_model`، مدل‌های تمیزدهنده و تولیدکننده را برمی‌گرداند. ابتدا این مدل‌ها را از کلاس‌های مربوطه خود به دست می‌آورد. سپس، اگر وزن‌های پیش‌آموزش‌دیده موجود باشند، وزن‌ها را بارگذاری می‌کند. در نهایت، اطلاعات مدل‌ها را چاپ و در متغیرهای مربوطه ذخیره می‌کند. متد `get_optimizer`، بهینه‌سازهای مربوط به تمیزدهنده و تولیدکننده را برمی‌گرداند. بر اساس پارامترهای ورودی، بهینه‌ساز مورد نظر را انتخاب و سپس برای تمیزدهنده و تولیدکننده جداگانه بهینه‌ساز را ایجاد می‌کند. در نهایت، اطلاعات بهینه‌سازها را چاپ و در متغیرهای مربوطه ذخیره می‌کند. متد `count_parameters`، تعداد پارامترهای قابل آموزش در یک مدل را محاسبه و چاپ می‌کند. این متد با استفاده از `nn.Module.named_parameters` پارامترهای قابل آموزش را در مدل پیدا می‌کند و تعداد آنها را محاسبه می‌کند. متد `d_loss_function`، تابع هزینه برای تمیزدهنده را محاسبه می‌کند. این تابع از `nn.BCELoss` استفاده می‌کند تا خروجی‌های تمیزدهنده و برچسب‌های مورد نظر را به عنوان ورودی دریافت کرده و مقدار هزینه را محاسبه می‌کند. متد `g_loss_function`، تابع هزینه برای مدل مولد را محاسبه می‌کند. این تابع با استفاده از `nn.BCELoss` و ورودی‌های مولد و برچسب‌های مورد نظر را ایجاد کرده و مقدار هزینه را محاسبه می‌کند. متد `train_discriminator` برای آموزش مدل تمیزدهنده بر روی یک دسته از داده‌ها استفاده می‌شود. ابتدا ورودی‌ها و برچسب‌های واقعی را از `batch_data` دریافت می‌کند و سپس برچسب‌های `one-hot` مربوط به برچسب‌های واقعی را ایجاد می‌کند. سپس با استفاده از مدل تمیزدهنده، خروجی‌های واقعی را محاسبه و برچسب یک را به آنها اختصاص می‌دهد. سپس نویز تصادفی را ایجاد کرده و با استفاده از مدل تولیدکننده، ورودی‌های تقلبی را تولید می‌کند و خروجی‌های تمیزدهنده را برای آنها محاسبه می‌کند و برچسب صفر به آنها اختصاص می‌دهد. در نهایت، خروجی‌ها و برچسب‌ها را با هم ترکیب کرده و تابع هزینه را برای تمیزدهنده محاسبه می‌کند. پس از انجام بهینه‌سازی، مقدار هزینه را برمی‌گرداند. متد `train_generator` برای آموزش مدل تولیدکننده روی یک دسته از داده‌ها استفاده می‌شود. ابتدا ورودی‌ها و برچسب‌های واقعی را از `batch_data` دریافت می‌کند و سپس برچسب‌های `one-hot` مربوط به برچسب‌های واقعی را ایجاد می‌کند. سپس نویز تصادفی را ایجاد کرده و با استفاده از مدل تولیدکننده، ورودی‌های تقلبی را تولید می‌کند و خروجی‌های تمیزدهنده را برای آنها محاسبه می‌کند. سپس تابع هزینه را برای تولیدکننده محاسبه می‌کند. پس از انجام بهینه‌سازی، مقدار هزینه را همراه با ورودی‌های تقلبی و برچسب‌های واقعی برمی‌گرداند. متد `train` هم فرایند آموزش را انجام می‌دهد. ابتدا زمان شروع آموزش را ثبت می‌کند و سپس برای تعداد مشخص شده از دوره‌ها، فرایند آموزش را تکرار می‌کند. در هر دوره، مدل‌های تمیزدهنده و تولیدکننده را به حالت آموزش

قرار می‌دهد و سپس برای هر دسته از داده‌ها، متدهای `train_generator` و `train_discriminator` را فراخوانی می‌کند و هزینه‌های به دست آمده را ذخیره می‌کند. همچنین در هر دوره، نمودارهایی از تغییرات هزینه‌ها را رسم می‌کند و تصاویری از ورودی‌های تقلبی تولید شده را نشان می‌دهد. در پایان هر دوره، مدل و پیکربندی‌ها را ذخیره می‌کند. متد `plot_loss` برای رسم نمودارهایی از تغییرات هزینه‌ها در فرایند آموزش استفاده می‌شود. دو پارامتر `train_mean_size` و `val_mean_size` برای تعیین اندازه میانگین‌گیری در هر دو مجموعه آموزش و اعتبارسنجی استفاده می‌شوند. در نهایت از متد `save` برای ذخیره مدل‌ها و پیکربندی‌ها استفاده می‌شود. ابتدا پیکربندی‌های مدل را در یک فایل JSON ذخیره می‌کند، سپس مدل مولد را در یک فایل مجزا ذخیره می‌کند. دستورات مربوطه و هم‌چنین فرآیندهای در نظر گرفته‌شده در برنامه ۱۲ آورده شده است.

Program 10: Main Code for Training D&G

```
1 # Some Configurations
2 # Setting the number of epochs for training
3 EPOCHS = 1100
4
5 # Checking if CUDA is available and setting the device accordingly
6 device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
7
8 # Setting the verbosity level for printing progress during training
9 # 0: Silent, 1: Minimal, 2: Moderate, 3: Verbose
10 VERBOSE = 3
11
12 # Setting a flag to save the generated plots
13 SAVE_PLOTS = True
14
15 # Setting a flag to visualize the plots during training
16 VISUALIZE_PLOTS = True
17
18 # Setting the directory path for saving the plots
19 SAVE_DIR = "./runs"
20
21 # Setting the model name
22 MODEL_NAME = "BreastMNIST_cDCGAN_Implementation"
23
24 # Choosing the optimizer for training
25 # Options: "Adam", "SGD"
26 OPTIMIZER = "Adam"
27
28 import os
29 import time
30 import math
31 from copy import deepcopy
32 import os.path as osp
33 import shutil
34 from prettytable import PrettyTable
35 import json
```

```

36
37 from tqdm import tqdm
38
39 import numpy as np
40 import torch
41 from torch.cuda import amp
42
43 # Import tensorboard
44 # from torch.utils.tensorboard import SummaryWriter
45
46 class Trainer:
47     """
48     Class for training a model.
49     """
50
51     def __init__(self, device=device, epochs=EPOCHS, batch_size=BATCH_SIZE, save_dir=SAVE_DIR,
52                  train_loader=train_dataloader, valid_loader=test_dataloader, weights=None, verbose=VERBOSE,
53                  visualize_plots=VISUALIZE_PLOTS, save_plots=SAVE_PLOTS, model_name=MODEL_NAME, optimizer=
54                  OPTIMIZER):
55         """
56         Initialize the Trainer object.
57
58         Args:
59             device (str): Device to use for training.
60             epochs (int): Number of training epochs.
61             batch_size (int): Batch size for training.
62             save_dir (str): Directory to save the trained model.
63             train_loader (DataLoader): DataLoader for training data.
64             valid_loader (DataLoader): DataLoader for validation data.
65             weights (str): Path to pretrained model weights.
66             verbose (int): Level of verbosity for printing information during training.
67             visualize_plots (bool): Whether to visualize training plots.
68             save_plots (bool): Whether to save training plots.
69             model_name (str): Name of the model.
70             optimizer (str): Name of the optimizer to use.
71         """
72         self.device = device
73         self.save_dir = save_dir
74         self.batch_size = batch_size
75         self.epochs = epochs
76         self.use_ema = False
77         self.model_name = model_name
78         self.weights = weights
79         self.visualize_plots = visualize_plots
80         self.save_plots = save_plots

```

```

78     # Verbosity levels: 0 = none, 1 = model architecture, 2 = optimizer information, 3 =
model parameters
79     self.verbose = verbose
80     self.d_losses = []
81     self.g_losses = []
82     self.conf = {'Name': self.model_name, 'Batch_size': self.batch_size, 'Max_iter_num': '',
'Epochs': self.epochs, 'Trained_epoch': 0, 'Optimizer': '', "Model": '', 'Parameter_size': ''
}
83     self.optimizer_name = optimizer
84
85     # Create a unique save directory
86     temm = 0
87     tmp_save_dir = self.save_dir
88     while osp.exists(tmp_save_dir):
89         tmp_save_dir = self.save_dir
90         temm += 1
91         tmp_save_dir += str(temm)
92     self.save_dir = tmp_save_dir
93     del temm
94
95     # Get data loaders
96     self.train_loader = train_loader
97     self.valid_loader = valid_loader
98     self.max_stepnum = len(self.train_loader)
99     self.conf["Max_iter_num"] = self.max_stepnum
100
101     # Get the model
102     self.d_model, self.g_model = self.get_model()
103     if self.verbose > 2:
104         self.count_parameters(self.d_model)
105         self.count_parameters(self.g_model)
106
107     # Get the optimizer
108     self.d_optimizer, self.g_optimizer = self.get_optimizer(optimizer=self.optimizer_name)
109
110     # Initialize tensorboard
111     # self.tblogger = SummaryWriter(self.save_dir)
112
113     ## INITIALIZERS
114     def get_model(self):
115         """
116         Get the Discriminator and Generator models.
117
118         Returns:
119             tuple: Tuple containing the Discriminator and Generator models.

```

```

120     """
121     # Get the Discriminator and Generator models from their respective classes.
122     d_model = Discriminator().to(self.device)
123     g_model = Generator().to(self.device)
124
125     # Load pretrained weights if provided
126     if self.weights:
127         print(f'Loading state_dict from {self.weights} for fine-tuning...')
128         g_model.load_state_dict(torch.load(self.weights))
129
130     # Log model information
131     if self.verbose > 0:
132         print('Generator Model:\n', g_model)
133         print('Discriminator Model:\n', d_model)
134         self.conf["Generator Model"] = str(g_model)
135         self.conf["Discriminator Model"] = str(d_model)
136
137     return d_model, g_model
138
139 def get_optimizer(self, optimizer="Adam", lr0=0.0001, beta1=0.5):
140     """
141     Get the Discriminator and Generator optimizers.
142
143     Args:
144         optimizer (str): Name of the optimizer to use. Options: "SGD" or "Adam" (default: "
Adam").
145         lr0 (float): Learning rate (default: 0.0002).
146         beta1 (float): Beta1 parameter for Adam optimizer (default: 0.5).
147
148     Returns:
149         tuple: Tuple containing the Discriminator and Generator optimizers.
150     """
151     assert optimizer in ['SGD', 'Adam'], 'ERROR: Unknown optimizer, defaulting to SGD.'
152
153     if optimizer == 'SGD':
154         d_optim = torch.optim.SGD(self.d_model.parameters(), lr=lr0, momentum=0.5)
155         g_optim = torch.optim.SGD(self.g_model.parameters(), lr=lr0, momentum=0.5)
156     elif optimizer == 'Adam':
157         d_optim = torch.optim.Adam(self.d_model.parameters(), lr=lr0, betas=(beta1, 0.999))
158         g_optim = torch.optim.Adam(self.g_model.parameters(), lr=lr0, betas=(beta1, 0.999))
159
160     if self.verbose > 1:
161         print(f"Discriminator optimizer: {type(d_optim).__name__}")
162         print(f"Generator optimizer: {type(g_optim).__name__}")
163

```

```

164     self.conf['Generator Optimizer'] = f"Generator optimizer: {type(g_optim).__name__}"
165     self.conf['Discriminator Optimizer'] = f"Discriminator optimizer: {type(d_optim).__name__}"
166
167     return d_optim, g_optim
168
169     def count_parameters(self, model):
170         """
171         Count the number of trainable parameters in a model.
172
173         Args:
174             model (nn.Module): The model to count parameters for.
175         """
176         table = PrettyTable(["Modules", "Parameters"])
177         total_params = 0
178         for name, parameter in model.named_parameters():
179             if not parameter.requires_grad:
180                 continue
181             params = parameter.numel()
182             table.add_row([name, params])
183             total_params += params
184         print(table)
185         print(f"Total Trainable Params: {total_params}")
186         self.conf["Parameter_size"] = total_params
187
188     def d_loss_function(self, inputs, targets):
189         """
190         Compute the loss for the Discriminator.
191
192         Args:
193             inputs (torch.Tensor): Discriminator inputs.
194             targets (torch.Tensor): Discriminator targets.
195
196         Returns:
197             torch.Tensor: The loss value.
198         """
199         return nn.BCELoss()(inputs, targets)
200
201     def g_loss_function(self, inputs):
202         """
203         Compute the loss for the Generator.
204
205         Args:
206             inputs (torch.Tensor): Generator inputs.
207

```

```

208     Returns:
209         torch.Tensor: The loss value.
210     """
211     targets = torch.ones([inputs.shape[0], 1]).to(device)
212     return nn.BCELoss()(inputs, targets)
213
214     ## TRAINING PROCESS
215     def train_discriminator(self, batch_data):
216         """
217         Train the Discriminator model on a batch of data.
218
219         Args:
220             batch_data (tuple): Tuple containing the input data and labels.
221
222         Returns:
223             float: The loss value.
224         """
225         real_inputs = batch_data[0].to(device)
226         real_labels = batch_data[1].to(device)
227
228         real_onehot_label = F.one_hot(torch.arange(2), 2).to(self.device)[real_labels].float()
229
230         real_outputs = self.d_model(real_inputs, real_onehot_label)
231         real_label = torch.ones(real_inputs.shape[0], 1).to(device)
232
233         noise = (torch.rand(real_inputs.shape[0], 100) - 0.5) / 0.5
234         noise = noise.to(device)
235         fake_inputs = self.g_model(noise, real_onehot_label)
236         fake_outputs = self.d_model(fake_inputs, real_onehot_label)
237         fake_label = torch.zeros(fake_inputs.shape[0], 1).to(device)
238
239         outputs = torch.cat((real_outputs, fake_outputs), 0)
240         targets = torch.cat((real_label, fake_label), 0)
241
242         # Zero the parameter gradients
243         self.d_optimizer.zero_grad()
244
245         # Backward propagation
246         d_loss = self.d_loss_function(outputs, targets)
247         d_loss.backward()
248         self.d_optimizer.step()
249         return d_loss.item()
250
251     def train_generator(self, batch_data):
252         """

```



```

253     Train the Generator model on a batch of data.
254
255     Args:
256         batch_data (tuple): Tuple containing the input data and labels.
257
258     Returns:
259         tuple: Tuple containing the loss value, generated inputs, and real labels.
260     """
261     real_inputs = batch_data[0].to(self.device)
262     real_labels = batch_data[1].to(device)
263
264     real_onehot_label = F.one_hot(torch.arange(2), 2).to(self.device)[real_labels].float()
265
266     noise = (torch.rand(real_inputs.shape[0], 100) - 0.5) / 0.5
267     noise = noise.to(device)
268
269     fake_inputs = self.g_model(noise, real_onehot_label)
270     fake_outputs = self.d_model(fake_inputs, real_onehot_label)
271
272     g_loss = self.g_loss_function(fake_outputs)
273     self.g_optimizer.zero_grad()
274     g_loss.backward()
275     self.g_optimizer.step()
276     return g_loss.item(), fake_inputs, real_labels
277
278 def train(self):
279     """
280     Train the models.
281
282     This method performs the training process, including training the Discriminator and
283     Generator models
284     for the specified number of epochs.
285     """
286     try:
287         # Training process prerequisite
288         self.start_time = time.time()
289         print('Start Training Process\nTime: {}'.format(time.ctime(self.start_time)))
290
291         # Epoch Loop
292         for self.epoch in range(0, self.epochs):
293             try:
294                 self.conf["Trained_epoch"] = self.epoch
295
296                 # Training loop
297                 self.g_model.train(True)

```

```

297         self.d_model.train(True)
298
299         pbar = enumerate(self.train_loader)
300         pbar = tqdm(pbar, total=self.max_stepnum)
301         for step, batch_data in pbar:
302             d_loss = self.train_discriminator(batch_data)
303             g_loss, fake_inputs, real_labels = self.train_generator(batch_data)
304
305             self.d_losses.append(d_loss)
306             self.g_losses.append(g_loss)
307             pbar.set_description(f"Epoch: {self.epoch}/{self.epochs}\tDiscriminator
Loss: {d_loss}\tGenerator Loss: {g_loss}")
308         del pbar
309
310     except Exception as _:
311         print('ERROR in training steps.')
312         raise
313
314     if self.epoch % 10 == 0:
315         # Plot Losses
316         self.plot_loss()
317         imgs_numpy = (fake_inputs.data.cpu().numpy() + 1.0) / 2.0
318         sqrtn = int(np.ceil(np.sqrt(imgs_numpy[:64].shape[0])))
319         for index, image in enumerate(imgs_numpy[:64]):
320             plt.subplot(sqrtn, sqrtn, index + 1)
321             plt.imshow(image.reshape(28, 28), cmap='gray')
322
323         if self.save_plots:
324             save_img_dir = osp.join(self.save_dir, 'images')
325             if not osp.exists(save_img_dir):
326                 os.makedirs(save_img_dir)
327             plt.savefig("{} / epoch-{}-img.pdf".format(save_img_dir, self.epoch))
328
329         if self.visualize_plots:
330             plt.show()
331             print(real_labels[:64])
332
333         # Save Model and Configurations
334         self.save()
335
336     except Exception as _:
337         print('ERROR in training loop or eval/save model.')
338         raise
339     finally:
340         finish_time = time.time()

```

```

341         # print(f'\nTraining completed in {time.ctime(finish_time)} \nIts Done in: {(time.
time() - self.start_time) / 3600:.3f} hours.')
342
343
344     ## Training Callback after each epoch
345     def plot_loss(self, train_mean_size=1, val_mean_size=1):
346         """
347         Plot the training and validation losses.
348
349         Args:
350             train_mean_size (int): Size of the training mean.
351             val_mean_size (int): Size of the validation mean.
352         """
353         COLS = 3
354         ROWS = 1
355         LINE_WIDTH = 2
356         fig, ax = plt.subplots(ROWS, COLS, figsize=(COLS * 10, ROWS * 10))
357
358         ax[0].plot(np.arange(len(self.d_losses) / train_mean_size),
359                   np.mean(np.array(self.d_losses).reshape(-1, train_mean_size), axis=1), 'r',
360                   label="training loss", linewidth=LINE_WIDTH)
361         ax[0].set_title("Discriminator Loss")
362         ax[1].plot(np.arange(len(self.g_losses) / val_mean_size),
363                   np.mean(np.array(self.g_losses).reshape(-1, val_mean_size), axis=1), 'g',
364                   label="validation loss", linewidth=LINE_WIDTH)
365         ax[1].set_title("Generator Loss")
366         ax[2].plot(np.arange(len(self.d_losses) / train_mean_size),
367                   np.mean(np.array(self.d_losses).reshape(-1, train_mean_size), axis=1), 'r',
368                   label="training loss", linewidth=LINE_WIDTH)
369         ax[2].plot(np.arange(len(self.g_losses) / val_mean_size),
370                   np.mean(np.array(self.g_losses).reshape(-1, val_mean_size), axis=1), 'g',
371                   label="validation loss", linewidth=LINE_WIDTH)
372         ax[2].set_title("Dis/Gen Loss")
373
374         if self.save_plots:
375             save_plot_dir = osp.join(self.save_dir, 'plots')
376             if not osp.exists(save_plot_dir):
377                 os.makedirs(save_plot_dir)
378             plt.savefig("{} /epoch-{}-loss-plot.pdf".format(save_plot_dir, self.epoch))
379         if self.visualize_plots:
380             plt.show()
381
382     ## Save Model
383     def save(self):
384         """

```

```

385     Save the trained model and configurations.
386     """
387     # Create config object
388     conf = json.dumps(self.conf)
389     f = open(self.save_dir + "/config.json", "w")
390     f.write(conf)
391     f.close()
392
393     # Save model
394     save_ckpt_dir = osp.join(self.save_dir, 'weights')
395     if not osp.exists(save_ckpt_dir):
396         os.makedirs(save_ckpt_dir)
397     filename = osp.join(save_ckpt_dir, self.model_name + "-" + str(self.epoch) + '.pt')
398     torch.save(self.g_model.state_dict(), filename)
399
400 ## Train the models
401 Trainer().train()

```

Program 11: Models Parameters

```

1  Generator Model:
2  Generator(
3      (linear_layer): Sequential(
4          (0): Linear(in_features=102, out_features=6272, bias=True)
5          (1): LeakyReLU(negative_slope=0.01)
6      )
7      (main): Sequential(
8          (0): ConvTranspose2d(128, 20, kernel_size=(5, 5), stride=(1, 1))
9          (1): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
10         (2): ReLU(inplace=True)
11         (3): ConvTranspose2d(20, 20, kernel_size=(4, 4), stride=(1, 1))
12         (4): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
13         (5): ReLU(inplace=True)
14         (6): ConvTranspose2d(20, 20, kernel_size=(4, 4), stride=(1, 1))
15         (7): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
16         (8): ReLU(inplace=True)
17         (9): ConvTranspose2d(20, 20, kernel_size=(4, 4), stride=(1, 1))
18         (10): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
19         (11): ReLU(inplace=True)
20         (12): ConvTranspose2d(20, 20, kernel_size=(4, 4), stride=(1, 1))
21         (13): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
22         (14): ReLU(inplace=True)
23         (15): ConvTranspose2d(20, 20, kernel_size=(3, 3), stride=(1, 1))
24         (16): BatchNorm2d(20, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
25         (17): ReLU(inplace=True)
26         (18): ConvTranspose2d(20, 1, kernel_size=(4, 4), stride=(1, 1))
27         (19): Tanh()
28     )
29 )
30 Discriminator Model:
31 Discriminator(
32     (linear_layer): Sequential(
33         (0): Linear(in_features=2, out_features=784, bias=True)
34         (1): LeakyReLU(negative_slope=0.01)
35     )
36     (main): Sequential(
37         (0): Conv2d(2, 20, kernel_size=(3, 3), stride=(1, 1))
38         (1): LeakyReLU(negative_slope=0.2, inplace=True)
39         (2): Conv2d(20, 30, kernel_size=(3, 3), stride=(1, 1))
40         (3): LeakyReLU(negative_slope=0.2, inplace=True)
41         (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
42         (5): Conv2d(30, 30, kernel_size=(3, 3), stride=(1, 1))

```

```

43 (6): LeakyReLU(negative_slope=0.2, inplace=True)
44 (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
45 (8): Conv2d(30, 20, kernel_size=(3, 3), stride=(1, 1))
46 (9): LeakyReLU(negative_slope=0.2, inplace=True)
47 (10): Conv2d(20, 1, kernel_size=(3, 3), stride=(1, 1))
48 (11): Sigmoid()
49 )
50 (flatten): Flatten(start_dim=1, end_dim=-1)
51 )

```

Modules	Parameters
linear_layer.0.weight	1568
linear_layer.0.bias	784
main.0.weight	360
main.0.bias	20
main.2.weight	5400
main.2.bias	30
main.5.weight	8100
main.5.bias	30
main.8.weight	5400
main.8.bias	20
main.10.weight	180
main.10.bias	1

Total Trainable Params: 21893

Modules	Parameters
linear_layer.0.weight	639744
linear_layer.0.bias	6272
main.0.weight	64000
main.0.bias	20
main.1.weight	20
main.1.bias	20
main.3.weight	6400
main.3.bias	20
main.4.weight	20
main.4.bias	20
main.6.weight	6400
main.6.bias	20
main.7.weight	20
main.7.bias	20
main.9.weight	6400
main.9.bias	20
main.10.weight	20
main.10.bias	20
main.12.weight	6400
main.12.bias	20
main.13.weight	20
main.13.bias	20
main.15.weight	3600
main.15.bias	20
main.16.weight	20
main.16.bias	20
main.18.weight	320
main.18.bias	1

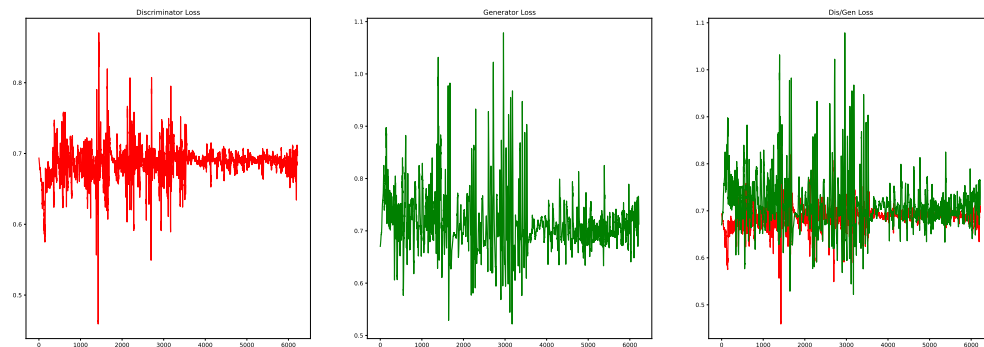
Total Trainable Params: 739897

Discriminator optimizer: Adam

Generator optimizer: Adam

خروجی نمودار توابع اتلاف در شکل ۵ آورده شده است. همان‌طور که مشاهده می‌شود نمودارها خالت نوسانی دارند اما یک روند نزولی ملایم در قسمت مولد و یک روند صعودی در قسمت تمیزدهنده مخصوصاً در ابتدای کار مشاهده می‌شود. برای بهتر شدن خروجی مولد راهکارهایی را می‌توان مدنظر قرار داد. تنظیم مجدد نرخ یادگیری ممکن است بهبود قابل توجهی در نوسانات داشته باشد. این شامل کاهش نرخ یادگیری، استفاده از نرخ یادگیری دیگری برای هر دو شبکه (discriminator و generator) یا استفاده از نرخ یادگیری آنی (adaptive learning rate) است. هم‌چنین ممکن است تغییر در معماری شبکه

مولد (generator) و یا تمیزدهنده (discriminator) بهبودی در نوسانات و عملکرد شبکه داشته باشد. ممکن است بهتر باشد که عمق شبکه را افزایش دهیم، تعداد لایه‌ها و یا تعداد واحدهای هر لایه را تغییر دهیم. استفاده از تکنیک‌های نرمال‌سازی مانند Batch Normalization می‌تواند به استقرار و پایداری شبکه کمک کند و نوسانات را کاهش دهد. اگر حجم داده‌های آموزش کم است، ممکن است شبکه دچار بیش‌برازش شود و نوسانات بیشتری را تجربه کند. در این صورت، جمع‌آوری و استفاده از مجموعه داده بزرگتر و یا با کلاس‌های متوازن یا استفاده از روش‌های افزایش داده می‌تواند مفید باشد. تغییر در تعداد دسته‌ها در هر دوره آموزش (epochs) می‌تواند بهبودی در نوسانات داشته باشد. ممکن است بخواهید دسته‌های کوچکتر یا بزرگتری را در هر دوره آموزش استفاده کنید.



شکل ۵: نمودار توابع اتلاف.

برای تولید ۲۰۰۰ نمونه برای هر کلاس دستوراتی را می‌نویسیم. دستورات مربوطه در برنامه ۱۲ آورده شده است. همان‌طور که مشاهده می‌شود تعداد نمونه‌های تولیدی برای هر کلاس در حدود ۲۰۰۰ تاستو

Program 12: Main Code for Training D&G

```
1 # Some Configurations
2 # Setting the number of epochs for training
3 EPOCHS = 1100
4
5 # Checking if CUDA is available and setting the device accordingly
6 device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
7
8 # Setting the verbosity level for printing progress during training
9 # 0: Silent, 1: Minimal, 2: Moderate, 3: Verbose
10 VERBOSE = 3
11
12 # Setting a flag to save the generated plots
13 SAVE_PLOTS = True
14
15 # Setting a flag to visualize the plots during training
16 VISUALIZE_PLOTS = True
17
18 # Setting the directory path for saving the plots
```

```

19 SAVE_DIR = "./runs"
20
21 # Setting the model name
22 MODEL_NAME = "BreastMNIST_cDCGAN_Implementation"
23
24 # Choosing the optimizer for training
25 # Options: "Adam", "SGD"
26 OPTIMIZER = "Adam"
27
28 import os
29 import time
30 import math
31 from copy import deepcopy
32 import os.path as osp
33 import shutil
34 from prettytable import PrettyTable
35 import json
36
37 from tqdm import tqdm
38
39 import numpy as np
40 import torch
41 from torch.cuda import amp
42
43 # Import tensorboard
44 # from torch.utils.tensorboard import SummaryWriter
45
46 class Trainer:
47     """
48     Class for training a model.
49     """
50
51     def __init__(self, device=device, epochs=EPOCHS, batch_size=BATCH_SIZE, save_dir=SAVE_DIR,
52                  train_loader=train_dataloader, valid_loader=test_dataloader, weights=None, verbose=VERBOSE,
53                  visualize_plots=VISUALIZE_PLOTS, save_plots=SAVE_PLOTS, model_name=MODEL_NAME, optimizer=
54                  OPTIMIZER):
55         """
56         Initialize the Trainer object.
57
58         Args:
59             device (str): Device to use for training.
60             epochs (int): Number of training epochs.
61             batch_size (int): Batch size for training.
62             save_dir (str): Directory to save the trained model.
63             train_loader (DataLoader): DataLoader for training data.

```

```

61         valid_loader (DataLoader): DataLoader for validation data.
62         weights (str): Path to pretrained model weights.
63         verbose (int): Level of verbosity for printing information during training.
64         visualize_plots (bool): Whether to visualize training plots.
65         save_plots (bool): Whether to save training plots.
66         model_name (str): Name of the model.
67         optimizer (str): Name of the optimizer to use.
68         """
69         self.device = device
70         self.save_dir = save_dir
71         self.batch_size = batch_size
72         self.epochs = epochs
73         self.use_ema = False
74         self.model_name = model_name
75         self.weights = weights
76         self.visualize_plots = visualize_plots
77         self.save_plots = save_plots
78         # Verbosity levels: 0 = none, 1 = model architecture, 2 = optimizer information, 3 =
model parameters
79         self.verbose = verbose
80         self.d_losses = []
81         self.g_losses = []
82         self.conf = {'Name': self.model_name, 'Batch_size': self.batch_size, 'Max_iter_num': '',
'EPOCHS': self.epochs, 'Trained_epoch': 0, 'Optimizer': '', "Model": '', 'Parameter_size': ''
}
83         self.optimizer_name = optimizer
84
85         # Create a unique save directory
86         temm = 0
87         tmp_save_dir = self.save_dir
88         while osp.exists(tmp_save_dir):
89             tmp_save_dir = self.save_dir
90             temm += 1
91             tmp_save_dir += str(temm)
92         self.save_dir = tmp_save_dir
93         del temm
94
95         # Get data loaders
96         self.train_loader = train_loader
97         self.valid_loader = valid_loader
98         self.max_stepnum = len(self.train_loader)
99         self.conf["Max_iter_num"] = self.max_stepnum
100
101         # Get the model
102         self.d_model, self.g_model = self.get_model()

```



```

103     if self.verbose > 2:
104         self.count_parameters(self.d_model)
105         self.count_parameters(self.g_model)
106
107     # Get the optimizer
108     self.d_optimizer, self.g_optimizer = self.get_optimizer(optimizer=self.optimizer_name)
109
110     # Initialize tensorboard
111     # self.tblogger = SummaryWriter(self.save_dir)
112
113     ## INITIALIZERS
114     def get_model(self):
115         """
116         Get the Discriminator and Generator models.
117
118         Returns:
119             tuple: Tuple containing the Discriminator and Generator models.
120         """
121         # Get the Discriminator and Generator models from their respective classes.
122         d_model = Discriminator().to(self.device)
123         g_model = Generator().to(self.device)
124
125         # Load pretrained weights if provided
126         if self.weights:
127             print(f'Loading state_dict from {self.weights} for fine-tuning...')
128             g_model.load_state_dict(torch.load(self.weights))
129
130         # Log model information
131         if self.verbose > 0:
132             print('Generator Model:\n', g_model)
133             print('Discriminator Model:\n', d_model)
134             self.conf["Generator Model"] = str(g_model)
135             self.conf["Discriminator Model"] = str(d_model)
136
137         return d_model, g_model
138
139     def get_optimizer(self, optimizer="Adam", lr0=0.0001, beta1=0.5):
140         """
141         Get the Discriminator and Generator optimizers.
142
143         Args:
144             optimizer (str): Name of the optimizer to use. Options: "SGD" or "Adam" (default: "
Adam").
145             lr0 (float): Learning rate (default: 0.0002).
146             beta1 (float): Beta1 parameter for Adam optimizer (default: 0.5).

```

```

147
148     Returns:
149         tuple: Tuple containing the Discriminator and Generator optimizers.
150     """
151     assert optimizer in ['SGD', 'Adam'], 'ERROR: Unknown optimizer, defaulting to SGD.'
152
153     if optimizer == 'SGD':
154         d_optim = torch.optim.SGD(self.d_model.parameters(), lr=lr0, momentum=0.5)
155         g_optim = torch.optim.SGD(self.g_model.parameters(), lr=lr0, momentum=0.5)
156     elif optimizer == 'Adam':
157         d_optim = torch.optim.Adam(self.d_model.parameters(), lr=lr0, betas=(beta1, 0.999))
158         g_optim = torch.optim.Adam(self.g_model.parameters(), lr=lr0, betas=(beta1, 0.999))
159
160     if self.verbose > 1:
161         print(f"Discriminator optimizer: {type(d_optim).__name__}")
162         print(f"Generator optimizer: {type(g_optim).__name__}")
163
164     self.conf['Generator Optimizer'] = f"Generator optimizer: {type(g_optim).__name__}"
165     self.conf['Discriminator Optimizer'] = f"Discriminator optimizer: {type(d_optim).__name__}"
166
167     return d_optim, g_optim
168
169     def count_parameters(self, model):
170         """
171         Count the number of trainable parameters in a model.
172
173         Args:
174             model (nn.Module): The model to count parameters for.
175         """
176         table = PrettyTable(["Modules", "Parameters"])
177         total_params = 0
178         for name, parameter in model.named_parameters():
179             if not parameter.requires_grad:
180                 continue
181             params = parameter.numel()
182             table.add_row([name, params])
183             total_params += params
184         print(table)
185         print(f"Total Trainable Params: {total_params}")
186         self.conf["Parameter_size"] = total_params
187
188     def d_loss_function(self, inputs, targets):
189         """
190         Compute the loss for the Discriminator.

```

```

191
192     Args:
193         inputs (torch.Tensor): Discriminator inputs.
194         targets (torch.Tensor): Discriminator targets.
195
196     Returns:
197         torch.Tensor: The loss value.
198     """
199     return nn.BCELoss()(inputs, targets)
200
201 def g_loss_function(self, inputs):
202     """
203     Compute the loss for the Generator.
204
205     Args:
206         inputs (torch.Tensor): Generator inputs.
207
208     Returns:
209         torch.Tensor: The loss value.
210     """
211     targets = torch.ones([inputs.shape[0], 1]).to(device)
212     return nn.BCELoss()(inputs, targets)
213
214 ## TRAINING PROCESS
215 def train_discriminator(self, batch_data):
216     """
217     Train the Discriminator model on a batch of data.
218
219     Args:
220         batch_data (tuple): Tuple containing the input data and labels.
221
222     Returns:
223         float: The loss value.
224     """
225     real_inputs = batch_data[0].to(device)
226     real_labels = batch_data[1].to(device)
227
228     real_onehot_label = F.one_hot(torch.arange(2), 2).to(self.device)[real_labels].float()
229
230     real_outputs = self.d_model(real_inputs, real_onehot_label)
231     real_label = torch.ones(real_inputs.shape[0], 1).to(device)
232
233     noise = (torch.rand(real_inputs.shape[0], 100) - 0.5) / 0.5
234     noise = noise.to(device)
235     fake_inputs = self.g_model(noise, real_onehot_label)

```

```

236     fake_outputs = self.d_model(fake_inputs, real_onehot_label)
237     fake_label = torch.zeros(fake_inputs.shape[0], 1).to(device)
238
239     outputs = torch.cat((real_outputs, fake_outputs), 0)
240     targets = torch.cat((real_label, fake_label), 0)
241
242     # Zero the parameter gradients
243     self.d_optimizer.zero_grad()
244
245     # Backward propagation
246     d_loss = self.d_loss_function(outputs, targets)
247     d_loss.backward()
248     self.d_optimizer.step()
249     return d_loss.item()
250
251 def train_generator(self, batch_data):
252     """
253     Train the Generator model on a batch of data.
254
255     Args:
256         batch_data (tuple): Tuple containing the input data and labels.
257
258     Returns:
259         tuple: Tuple containing the loss value, generated inputs, and real labels.
260     """
261     real_inputs = batch_data[0].to(self.device)
262     real_labels = batch_data[1].to(device)
263
264     real_onehot_label = F.one_hot(torch.arange(2), 2).to(self.device)[real_labels].float()
265
266     noise = (torch.rand(real_inputs.shape[0], 100) - 0.5) / 0.5
267     noise = noise.to(device)
268
269     fake_inputs = self.g_model(noise, real_onehot_label)
270     fake_outputs = self.d_model(fake_inputs, real_onehot_label)
271
272     g_loss = self.g_loss_function(fake_outputs)
273     self.g_optimizer.zero_grad()
274     g_loss.backward()
275     self.g_optimizer.step()
276     return g_loss.item(), fake_inputs, real_labels
277
278 def train(self):
279     """
280     Train the models.

```

```

281
282     This method performs the training process, including training the Discriminator and
Generator models
283     for the specified number of epochs.
284     """
285     try:
286         # Training process prerequisite
287         self.start_time = time.time()
288         print('Start Training Process\nTime: {}'.format(time.ctime(self.start_time)))
289
290         # Epoch Loop
291         for self.epoch in range(0, self.epochs):
292             try:
293                 self.conf["Trained_epoch"] = self.epoch
294
295                 # Training loop
296                 self.g_model.train(True)
297                 self.d_model.train(True)
298
299                 pbar = enumerate(self.train_loader)
300                 pbar = tqdm(pbar, total=self.max_stepnum)
301                 for step, batch_data in pbar:
302                     d_loss = self.train_discriminator(batch_data)
303                     g_loss, fake_inputs, real_labels = self.train_generator(batch_data)
304
305                     self.d_losses.append(d_loss)
306                     self.g_losses.append(g_loss)
307                     pbar.set_description(f"Epoch: {self.epoch}/{self.epochs}\tDiscriminator
Loss: {d_loss}\tGenerator Loss: {g_loss}")
308                 del pbar
309
310             except Exception as _:
311                 print('ERROR in training steps.')
312                 raise
313
314             if self.epoch % 10 == 0:
315                 # Plot Losses
316                 self.plot_loss()
317                 imgs_numpy = (fake_inputs.data.cpu().numpy() + 1.0) / 2.0
318                 sqrtn = int(np.ceil(np.sqrt(imgs_numpy[:64].shape[0])))
319                 for index, image in enumerate(imgs_numpy[:64]):
320                     plt.subplot(sqrtn, sqrtn, index + 1)
321                     plt.imshow(image.reshape(28, 28), cmap='gray')
322
323                 if self.save_plots:

```

```

324         save_img_dir = osp.join(self.save_dir, 'images')
325         if not osp.exists(save_img_dir):
326             os.makedirs(save_img_dir)
327         plt.savefig("{} /epoch-{}-img.pdf".format(save_img_dir, self.epoch))
328
329         if self.visualize_plots:
330             plt.show()
331             print(real_labels[:64])
332
333         # Save Model and Configurations
334         self.save()
335
336     except Exception as _:
337         print('ERROR in training loop or eval/save model.')
338         raise
339     finally:
340         finish_time = time.time()
341         # print(f'\nTraining completed in {time.ctime(finish_time)} \nIts Done in: {(time.
342         time() - self.start_time) / 3600:.3f} hours.')
343
344     ## Training Callback after each epoch
345     def plot_loss(self, train_mean_size=1, val_mean_size=1):
346         """
347         Plot the training and validation losses.
348
349         Args:
350             train_mean_size (int): Size of the training mean.
351             val_mean_size (int): Size of the validation mean.
352         """
353         COLS = 3
354         ROWS = 1
355         LINE_WIDTH = 2
356         fig, ax = plt.subplots(ROWS, COLS, figsize=(COLS * 10, ROWS * 10))
357
358         ax[0].plot(np.arange(len(self.d_losses) / train_mean_size),
359                   np.mean(np.array(self.d_losses).reshape(-1, train_mean_size), axis=1), 'r',
360                   label="training loss", linewidth=LINE_WIDTH)
361         ax[0].set_title("Discriminator Loss")
362         ax[1].plot(np.arange(len(self.g_losses) / val_mean_size),
363                   np.mean(np.array(self.g_losses).reshape(-1, val_mean_size), axis=1), 'g',
364                   label="validation loss", linewidth=LINE_WIDTH)
365         ax[1].set_title("Generator Loss")
366         ax[2].plot(np.arange(len(self.d_losses) / train_mean_size),
367                   np.mean(np.array(self.d_losses).reshape(-1, train_mean_size), axis=1), 'r',

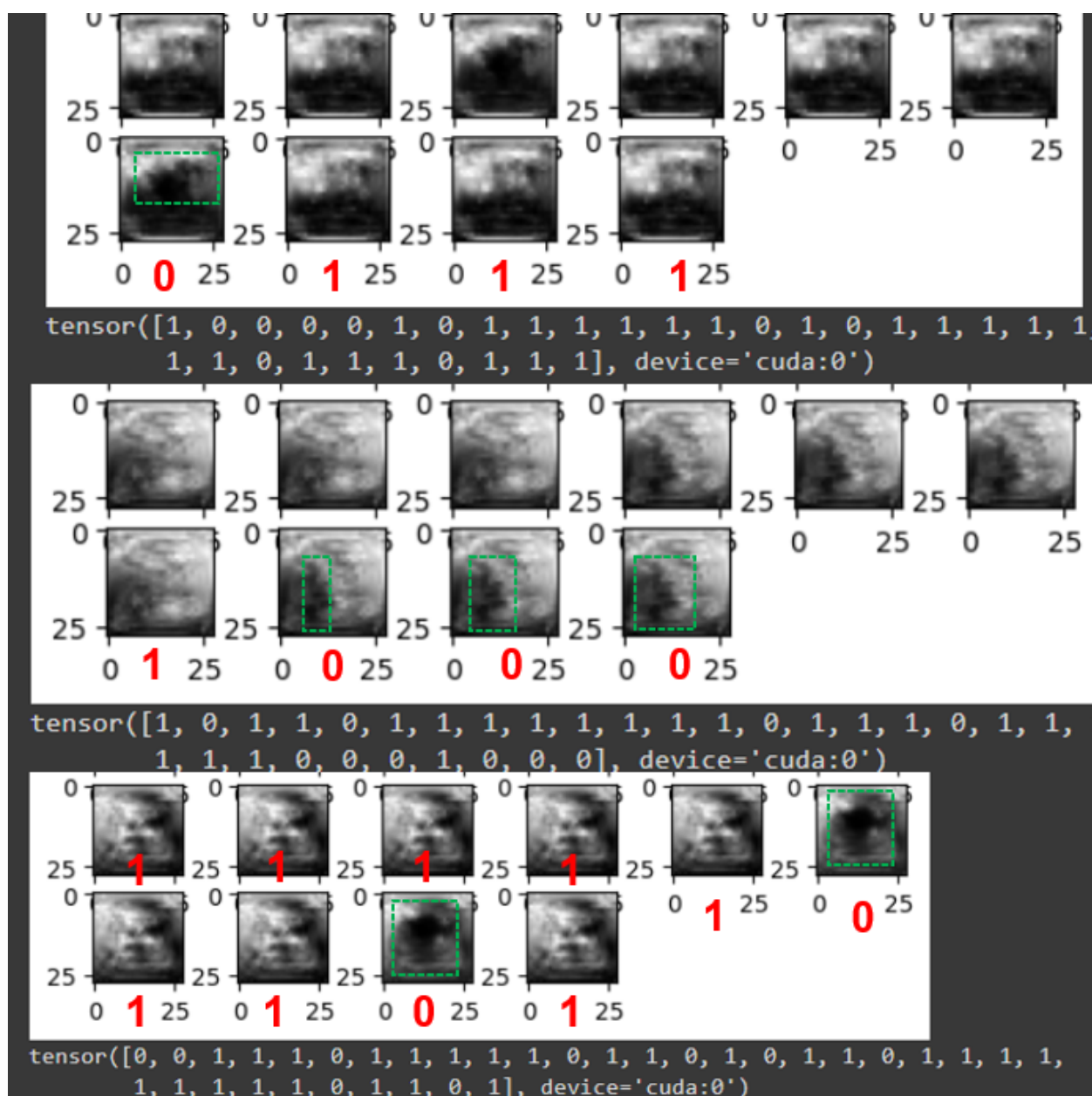
```

```

368         label="training loss", linewidth=LINE_WIDTH)
369         ax[2].plot(np.arange(len(self.g_losses) / val_mean_size),
370                   np.mean(np.array(self.g_losses).reshape(-1, val_mean_size), axis=1), 'g',
371                   label="validation loss", linewidth=LINE_WIDTH)
372         ax[2].set_title("Dis/Gen Loss")
373
374         if self.save_plots:
375             save_plot_dir = osp.join(self.save_dir, 'plots')
376             if not osp.exists(save_plot_dir):
377                 os.makedirs(save_plot_dir)
378             plt.savefig("{} /epoch-{}-loss-plot.pdf".format(save_plot_dir, self.epoch))
379         if self.visualize_plots:
380             plt.show()
381
382     ## Save Model
383     def save(self):
384         """
385         Save the trained model and configurations.
386         """
387         # Create config object
388         conf = json.dumps(self.conf)
389         f = open(self.save_dir + "/config.json", "w")
390         f.write(conf)
391         f.close()
392
393         # Save model
394         save_ckpt_dir = osp.join(self.save_dir, 'weights')
395         if not osp.exists(save_ckpt_dir):
396             os.makedirs(save_ckpt_dir)
397         filename = osp.join(save_ckpt_dir, self.model_name + "-" + str(self.epoch) + '.pt')
398         torch.save(self.g_model.state_dict(), filename)
399
400     ## Train the models
401     Trainer().train()

```

از نگاه به برخی تصاویر تولیدی در مراحل میانی آموزش مشخص می‌شود که به ظاهر گویا شبکه در تلاش است تا ویژگی‌های متمایز از دو کلاس ایجاد کند. مثلاً تصاویر تولیدی شبکه در شکل ۶ نشان می‌دهد که شبکه دارد تصاویر با نقاط تاریک بیش‌تر را در کلاس صفر تولید می‌کند. این نقاط تاریک‌تر در تصاویر با خط‌چین سبز مشخص شده‌اند. شماره کلاس هر داده تولیدی هم زیر آن نوشته شده است. این موضوع با دقت چشمی در اشکال برای یک حساب سرانگشتی از عملکرد مناسب شبکه قابل مشاهده است. هرچند با توجه به سختی مجموعه داده باید مراحل آموزش خیلی پیش رود تا داده‌های مناسبی تولید شود. در شکل ۷ هم بوضوح تصاویر تولیدی و کلاس آن‌ها به صورت برداری عددی در زیرش آورده شده. با تناظر چپ به راست و بالا به پایین اعداد و تصاویر، توانایی شبکه در ایجاد تصاویر متمایز مربوط به دو کلاس تا حدودی مشخص شده است. این تصاویر مربوط به دوره‌های آموزش شب جلوتر از تصویر قبلی است و این نشان می‌دهد که با ادامه آموزش تا مثلاً ۵۰۰۰ و ۱۰۰۰۰ دوره به نتایج بسیار بهتری دست پیدا خواهیم کرد.



شکل ۶: برخی نمونه‌های تولیدی و حساب سرانگشتی از تصاویر تولیدی شبکه.

۳.۱ پاسخ قسمت ۳ - طبقه‌بندی به کمک داده‌های تولید شده توسط مولد

برای این قسمت برنامه‌ای می‌نویسیم که ابتدا تابعی به نام `generate_samples` تعریف می‌کند که برای یک کلاس خاص، تعدادی نمونه تولید می‌کند. در این تابع، ابتدا یک بردار نویز با ابعاد `(num_samples, 100)` ایجاد می‌شود. سپس یک بردار برچسب به طول `num_samples` با ابعاد `(2)` ایجاد می‌شود و عنصر متناظر با کلاس مورد نظر برابر ۱ قرار می‌گیرد و بقیه عناصر بردار برابر ۰ می‌شوند. سپس مولد به دست آمده را به دستگاه مورد استفاده منتقل کرده و با استفاده از نویز و برچسب، تصاویر تولید می‌شوند. در نهایت، تصاویر تولید شده برگردانده می‌شوند. سپس با استفاده از تابع `generate_samples`، تعدادی



شکل ۷: برخی نمونه‌های تولیدی و حساب سرانگشتی از تصاویر تولیدی شبکه.

نمونه برای دو کلاس مختلف (کلاس ۰ و کلاس ۱) تولید می‌شود. سپس تعداد کمترین نمونه موجود بین دو کلاس محاسبه می‌شود و تعداد نمونه‌های تولید شده برای هر کلاس به تعداد کمترین نمونه کاهش می‌یابد. سپس نمونه‌های تولید شده برای هر کلاس به تعداد مشخص شده کاهش می‌یابند و در دو مجموعه داده با برچسب‌های مناسب ذخیره می‌شوند. سپس با استفاده از DataLoader، داده‌های تولید شده به صورت تصادفی با سایر داده‌ها ترکیب شده و به صورت دسته‌ای بارگذاری می‌شوند. در نهایت، تعداد نمونه‌های موجود در هر کلاس محاسبه و چاپ می‌شود. در ادامه ابتدا تبدیل‌های مختلفی برای تغییر و تبدیل داده‌ها تعریف شده است. تبدیل‌های اعمال شده عبارتند از: تغییر اندازه تصویر به ابعاد ۲۲۴ در ۲۲۴ پیکسل. تبدیل تصویر به فرمت رنگی RGB. تبدیل داده به تانسور (Tensor) یعنی نمایش عددی چندبعدی از تصویر. نرمال‌سازی داده با استفاده از میانگین و انحراف معیار به ترتیب برابر با ۰.۵ و ۰.۵. سپس تبدیل‌های تعریف شده به هر مجموعه داده اعمال می‌شود. با انجام این کار مجموعه داده ایجاد شده به کمک مولد آماده خوراندن به شبکه و ساختار ایجاد شده در قسمت اول می‌شود. دستورات مربوطه در برنامه ۱۳ و نتایج در برنامه ۱۴، شکل ۸ و ۹ آورده شده است. مشاهده می‌شود که نتایج کمی بهبود پیدا کرده است.

Program 13: ResNet on Gen Data Implementation

```
1 import torch
2 from torchvision.utils import save_image
3
```

```

4 # Function to generate samples for a given class
5 def generate_samples(generator, class_label, num_samples):
6     noise = torch.randn(num_samples, 100).to(device)
7     label = torch.ones(num_samples, 2).to(device)
8     label[:, class_label] = 1.0
9     generator = generator.to(device) # Move generator to the same device as noise and label
10    generated_images = generator(noise, label)
11    return generated_images
12
13 # Generate samples for class 0
14 class_0_samples = generate_samples(generator, 0, 2000)
15
16 # Generate samples for class 1
17 class_1_samples = generate_samples(generator, 1, 2000)
18
19 # Calculate the minimum number of samples per class
20 min_samples_per_class = min(len(class_0_samples), len(class_1_samples))
21
22 # Trim the generated samples to have the same number of samples per class
23 class_0_samples = class_0_samples[:min_samples_per_class]
24 class_1_samples = class_1_samples[:min_samples_per_class]
25
26 # Save the generated samples with suitable format and label in a dataloader
27 class_0_dataset = torch.utils.data.TensorDataset(class_0_samples, torch.zeros(
28     min_samples_per_class))
29
30 class_1_dataset = torch.utils.data.TensorDataset(class_1_samples, torch.ones(
31     min_samples_per_class))
32
33 generated_dataloader = torch.utils.data.DataLoader(
34     torch.utils.data.ConcatDataset([class_0_dataset, class_1_dataset]),
35     batch_size=BATCH_SIZE,
36     shuffle=True
37 )
38
39 # Add the generated data to the train_dataloader
40 new_train_dataloader = torch.utils.data.DataLoader(
41     torch.utils.data.ConcatDataset([train_dataset, generated_dataloader.dataset]),
42     batch_size=BATCH_SIZE,
43     shuffle=True
44 )
45
46 class_0_samples = len(class_0_dataset)
47 class_1_samples = len(class_1_dataset)

```

```

47 print("Number of samples in class 0 dataset:", class_0_samples)
48 print("Number of samples in class 1 dataset:", class_1_samples)
49
50
51 import torchvision.transforms as transforms
52
53 data_transform = transforms.Compose([
54     transforms.Resize(224), # Resize the image to 224x224
55     transforms.Grayscale(3), # Convert the image to RGB format
56     transforms.ToTensor(), # Convert data to tensors
57     transforms.Normalize(mean=[.5], std=[.5]) # Normalize the data
58 ])
59
60 # Apply transformations to train_dataset
61 train_dataset.transform = data_transform
62
63 # Apply transformations to val_dataset
64 val_dataset.transform = data_transform
65
66 # Apply transformations to test_dataset
67 test_dataset.transform = data_transform
68
69 # Update train_loader
70 train_loader = data.DataLoader(
71     torch.utils.data.ConcatDataset([train_dataset, generated_dataloader.dataset]), batch_size=
        BATCH_SIZE, shuffle=True, num_workers=4, pin_memory=True
72 )
73
74 # Update val_loader
75 val_loader = data.DataLoader(
76     val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4, pin_memory=True
77 )
78
79 # Update test_loader
80 test_loader = data.DataLoader(
81     test_dataset, batch_size=156, shuffle=False, num_workers=4, pin_memory=True
82 )
83
84
85 import numpy as np
86 import torch
87 import torch.nn as nn
88 import torch.optim as optim
89 import torch.utils.data as data
90 import torchvision.transforms as transform

```

```

91 from matplotlib import pyplot as plt
92 from tqdm import tqdm
93 from torchvision import transforms
94
95 # Define variable names and add comments
96
97 dataset_name = "breastmnist" # The name of the dataset
98 download = True # Flag to download the dataset if not available locally
99 dataset_name = dataset_name.lower() # Convert the dataset name to lowercase
100
101 NUM_EPOCHS = 5 # Number of epochs for training
102 BATCH_SIZE = 64 # Batch size for data loading
103 lr = 0.0001 # Learning rate for the optimizer
104
105
106 # Import ResNet-50 model
107 import torchvision.models as models
108
109 # Load the pre-trained ResNet-50 model
110 model = models.resnet50(pretrained=True)
111
112 # Replace the last fully connected layer to match the number of output classes
113 num_classes = 10
114 model.fc = nn.Linear(model.fc.in_features, num_classes)
115
116 # Define the loss function and optimizer
117 criterion = nn.CrossEntropyLoss()
118 optimizer = optim.Adam(model.parameters(), lr=lr)
119
120 # Training loop
121 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
122 model.to(device)
123
124 train_losses = []
125 val_losses = []
126 train_accs = []
127 val_accs = []
128
129 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
130 model.to(device)
131
132 for epoch in range(NUM_EPOCHS):
133     train_loss = 0.0
134     val_loss = 0.0
135     train_total = 0

```

```

136     train_correct = 0
137     val_total = 0
138     val_correct = 0
139
140     # Training
141     model.train()
142     for images, labels in tqdm(train_loader, desc=f'Epoch {epoch+1}/{NUM_EPOCHS} - Training'):
143         images = images.to(device)
144         labels = labels.squeeze().to(device) # Convert multi-target labels to single-valued
145         labels
146
147         optimizer.zero_grad()
148
149         outputs = model(images)
150         loss = criterion(outputs, labels)
151         loss.backward()
152         optimizer.step()
153
154         train_loss += loss.item() * images.size(0)
155
156         _, predicted = torch.max(outputs.data, 1)
157         train_total += labels.size(0)
158         train_correct += (predicted == labels).sum().item()
159
160     # Validation
161     model.eval()
162     with torch.no_grad():
163         for images, labels in tqdm(val_loader, desc=f'Epoch {epoch+1}/{NUM_EPOCHS} - Validation'):
164             :
165             images = images.to(device)
166             labels = labels.squeeze().to(device) # Convert multi-target labels to single-valued
167             labels
168
169             outputs = model(images)
170             loss = criterion(outputs, labels)
171
172             val_loss += loss.item() * images.size(0)
173
174             _, predicted = torch.max(outputs.data, 1)
175             val_total += labels.size(0)
176             val_correct += (predicted == labels).sum().item()
177
178     # Calculate metrics
179     train_loss /= len(train_loader.dataset)
180     val_loss /= len(val_loader.dataset)

```

```

178     train_accuracy = train_correct / train_total
179     val_accuracy = val_correct / val_total
180
181     train_losses.append(train_loss)
182     val_losses.append(val_loss)
183     train_accs.append(train_accuracy)
184     val_accs.append(val_accuracy)
185
186     print(f'Epoch {epoch+1}/{NUM_EPOCHS} - Training Loss: {train_loss:.4f} - Training Accuracy: {
        train_accuracy:.4f} - Validation Loss: {val_loss:.4f} - Validation Accuracy: {val_accuracy:.4
        f}')
187
188 # Plot loss and accuracy
189 plt.figure(figsize=(10, 5))
190 plt.plot(train_losses, label='Training Loss')
191 plt.plot(val_losses, label='Validation Loss')
192 plt.xlabel('Epoch')
193 plt.ylabel('Loss')
194 plt.legend()
195 plt.title('Training and Validation Loss')
196 plt.savefig('loss_plot.pdf')
197
198 plt.figure(figsize=(10, 5))
199 plt.plot(train_accs, label='Training Accuracy')
200 plt.plot(val_accs, label='Validation Accuracy')
201 plt.xlabel('Epoch')
202 plt.ylabel('Accuracy')
203 plt.legend()
204 plt.title('Training and Validation Accuracy')
205 plt.savefig('accuracy_plot.pdf')
206
207 # Classification report on the test split
208 model.eval()
209 test_total = 0
210 test_correct = 0
211 test_predictions = []
212 test_targets = []
213
214 with torch.no_grad():
215     for images, labels in tqdm(test_loader, desc='Test'):
216         images = images.to(device)
217         labels = labels.squeeze().to(device) # Convert multi-target labels to single-valued
        labels
218
219         outputs = model(images)

```

```

220
221     _, predicted = torch.max(outputs.data, 1)
222     test_total += labels.size(0)
223     test_correct += (predicted == labels).sum().item()
224
225     test_predictions.extend(predicted.cpu().numpy())
226     test_targets.extend(labels.cpu().numpy())
227
228 test_accuracy = test_correct / test_total
229 print(f'Test Accuracy: {test_accuracy:.4f}')
230
231 # Print classification report
232 from sklearn.metrics import classification_report
233 target_names = info['label']
234 classification_rep = classification_report(test_targets, test_predictions, target_names=
    target_names)
235 print(classification_rep)
236
237 # Plot confusion matrix
238 from sklearn.metrics import confusion_matrix
239 import seaborn as sns
240
241 cm = confusion_matrix(test_targets, test_predictions)
242 plt.figure(figsize=(10, 8))
243 sns.heatmap(cm, annot=True, fmt='d', xticklabels=target_names, yticklabels=target_names)
244 plt.xlabel('Predicted')
245 plt.ylabel('True')
246 plt.title('Confusion Matrix')
247 plt.savefig('confusion_matrix_plot.pdf')

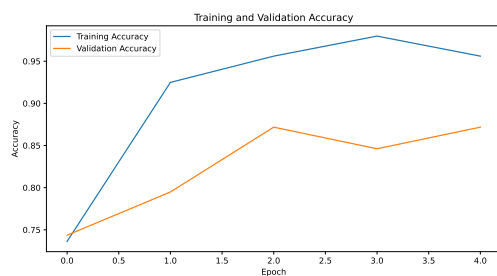
```

Program 14: ResNet on Data Results

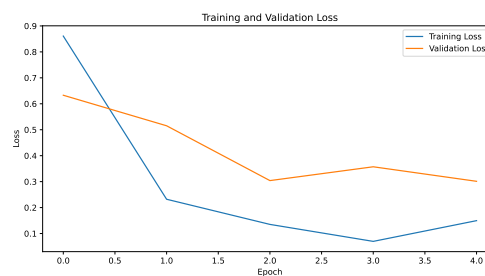
```

1 Test Accuracy: 0.8782
2           precision    recall  f1-score   support
3
4      0           0.79      0.74      0.77         42
5      1           0.91      0.93      0.92        114
6
7    accuracy                        0.88        156
8    macro avg           0.85      0.83      0.84        156
9    weighted avg        0.88      0.88      0.88        156

```

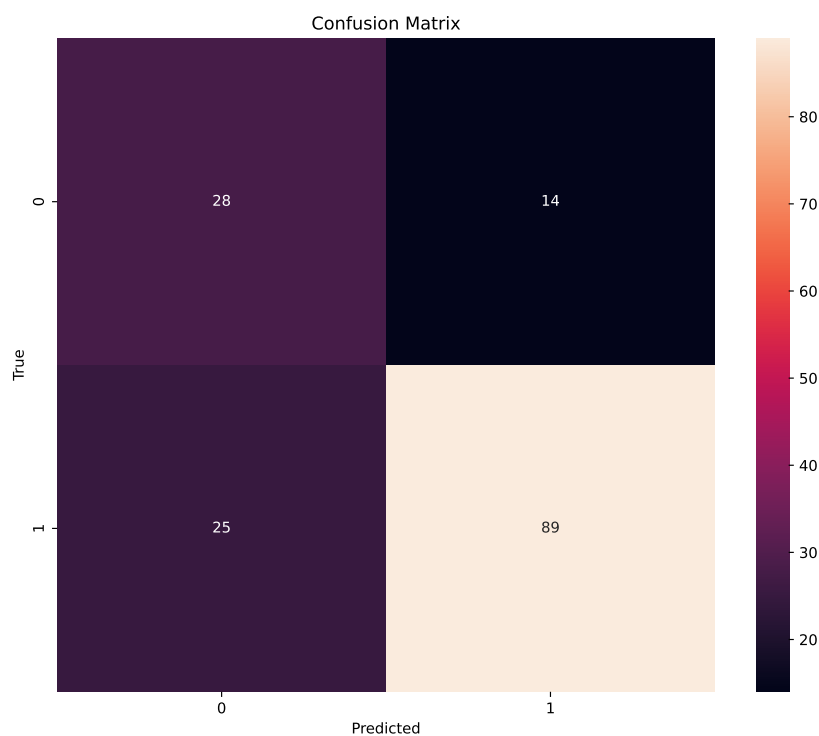


(ب) دقت



(آ) اتلاف

شکل ۸: نمودار تابع اتلاف و دقت مدل در حالت مولد.



شکل ۹: ماتریس درهم‌ریختگی در حالت مولد.