به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

درس تحلیل و طراحی شبکه های عصبی عمیق

تمرین شماره ۱

نام و نام خانوادگی : علیرضا حسینی – کیانا هوشانفر

شماره دانشجویی : ۸۱۰۱۰۱۱۴۲ - ۸۱۰۱۰۱۳۶۱

آبان ماه ۱۴۰۲

فهرست

مقدمه

در حوزه یادگیری ماشین و یادگیری عمیق ، ارزیابی عملکرد مدل فراتر از معیارهای دقت مرسوم است. ارزیابی اینکه یک مدل چگونه داده ها را در فضای ویژگی خود سازماندهی و جدا می کند، برای درک اثربخشی آن در گرفتن الگوها و بازنمایی های پیچیده بسیار مهم است. در این تمرین ، ما به کاوش در دو معیار متمایز اما مکمل - شاخص (SI) و (SMI) برای بررسی دقیق عملکرد شبکه عصبی EfficientnetV2_s می پردازیم و برای تسهیل تجزیه و تحلیل خود، از دو مجموعه داده متنوع - Cifar100 و diabets.csv استفاده می کنیم.

هدف کلی ما آموزش EfficientnetV2_s بر روی این مجموعه داده ها، بررسی کامل شاخص SI و SMI و نتیجه گیری در مورد توانایی شبکه برای جداسازی و نمایش داده ها در فضای ویژگی آن است. با آشکار کردن تعامل پیچیده بین معماری مدل و ویژگی‌های مجموعه داده، هدف ما این است که بینش عمیق‌تری در مورد مکانیسم‌های اساسی EfficientnetV2_s و کاربردهای بالقوه آن در سناریوهای مختلف دنیای واقعی به دست آوریم.

سوال اول : شاخص های هندسی

الف) آموزش شبکه بر روی دیتاست CIFAR 100

ابتدا دیتاست Cifar 100 را به کمک دستور زیر لود کرده و data loader آن را میسازیم.

```
# Define data transformations
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Load CIFAR-100 dataset
cifar100_train = datasets.CIFAR100(root='./data', train=True,
download=True, transform=transform)
cifar100_test = datasets.CIFAR100(root='./data', train=False,
download=True, transform=transform)


# Create PyTorch data loaders
batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_train,
batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(cifar100_test,
batch_size=batch_size, shuffle=False)
```

در ادامه مدل را لود میکنیم. با توجه به اینکه ترین شبکه روی CIFAR 100 از Scratch کار دشواری می باشد باید نسخه pretrained آن را لود کنیم که با توجه به اینکه در آن نسخه خروجی MLP هزار کلاسه میباشد باید بخش classifier آن عوض شود که به همین دلیل به صورت زیر کد زده شده است.

```
# Load EfficientNetV2-S model
model = models.efficientnet_v2_s(weights='IMAGENET1K_V1')

# 100 classes
model.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True),
    nn.Linear(1280, 100)
)
```

بدین ترتیب در نسخه نهایی مدل بخش classifier به صورت زیر میشود.

```
classifier = model.classifier
print(classifier)

Sequential(
  (0): Dropout(p=0.2, inplace=True)
  (1): Linear(in_features=1280, out_features=100, bias=True)
)
```

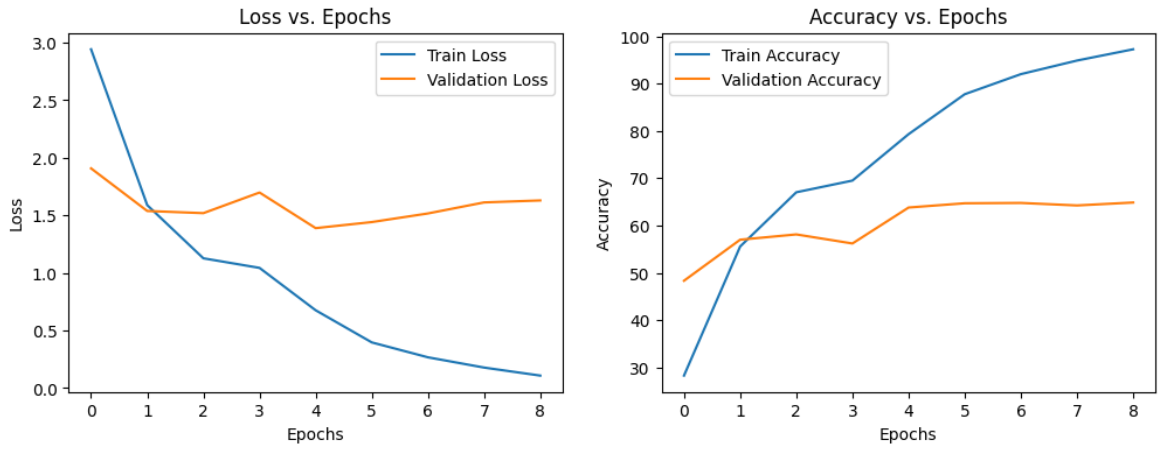
حال به کمک کد زیر لوپ ترین را تشکیل داده تا مدل بر روی دیتاست CIFAR 100 آموزش داده شود.

لازم به ذکر است که در لوپ ترین از StepLR عنوان scheduler استفاده شده است و تعداد epoch ها نیز برابر ۹ قرار داده شده است. به علاوه مدلی که روی داده های ارزیابی بهترین عملکرد را داشته باشد ذخیره میشود نه الزاما مدلی که در epoch آخر میباشد.

```
# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = StepLR(optimizer, step_size=4, gamma=0.2)

# Training loop
num_epochs = 9
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

train_losses, val_losses = [], []
train_acc, val_acc = [], []

best_val_acc = 0.0
best_model_path = 'efficientnetv2_s_cifar100_best.pth'

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for inputs, labels in tqdm(train_loader, desc=f'Epoch {epoch +
1}/{num_epochs}'):
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
```

```
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    train_losses.append(running_loss / len(train_loader))
    train_acc.append(100 * correct_train / total_train)

    model.eval()
    running_loss = 0.0
    correct_val = 0
    total_val = 0

    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    val_losses.append(running_loss / len(val_loader))
    val_acc.append(100 * correct_val / total_val)


    scheduler.step()

    # Check if the current model has the best validation accuracy
    if val_acc[-1] > best_val_acc:
        best_val_acc = val_acc[-1]
        torch.save(model.state_dict(), best_model_path)

    print(f"Epoch {epoch + 1}/{num_epochs} | Train Loss:
{train_losses[-1]:.4f} | Train Acc: {train_acc[-1]:.2f}% | Val Loss:
{val_losses[-1]:.4f} | Val Acc: {val_acc[-1]:.2f}%")

# Load the best model
model.load_state_dict(torch.load(best_model_path))

# Save model weights
torch.save(model.state_dict(),
'efficientnetv2_s_cifar100_finetuned.pth')
```

پس از اتمام آموزش منحنی های دقت و loss بر روی داد های آموزش و ارزیابی به صورت زیر میشود.



شکل ۱ : منحنی دقت و loss پس از آموزش مدل بر روی دیتاست

مشاهده میشود که بهترین دقت مدل نهایی آموزش داده شده بر روی داده های ارزیابی و آموزش به شرح زیر میباشد. (با توجه به ۱۰۰ کلاسه بودن دیتاست این مساله که عملکرد روی داده های ارزیابی خیلی بالا نباشد طبیعی میباشد)

Train Acc: 97.24%| Val Acc: 64.86%

ب) بررسی شاخص های SI بر روی داده های CIFAR100

برای این بخش ابتدا کتابخانه های مورد نیاز را import میکنیم.

```
import torch

from torchvision import datasets, transforms
import random
import numpy as np

from data_complexity_measures.models.SeprationIndex import
Kalhor_SeparationIndex
```

با توجه به اینکه نمیتوان برخی از شاخص ها را بر روی همه دیتاست بدست آورد نیاز است یک sub sample که بالانس باشد را جدا کنیم که برای هر بخش ممکن است درصد آن جدا باشد.

به کمک کد زیر subsample ها را جدا میکنیم.

```
from torch.utils.data.sampler import SubsetRandomSampler

# Load the CIFAR-100 dataset and create a balanced subset
transform = transforms.Compose([transforms.ToTensor()])
cifar100_dataset = datasets.CIFAR100(root='./data', train=True,
download=True, transform=transform)

# Define the subset size
subset_fraction = 0.02 # different for each metric ( for SI:80% ,
HSI:60% , for anti SI :60% , for Center SI: 3% only
subset_size_train = int(subset_fraction * len(cifar100_dataset))

# Create a balanced subset for both train and test sets using
SubsetRandomSampler
class_indices = list(range(len(cifar100_dataset.classes)))
class_subset_size = int(subset_size_train /
len(cifar100_dataset.classes))

class_sampler_indices_train = []
class_sampler_indices_test = []

for class_index in class_indices:
    class_indices_list_train = [i for i, label in
enumerate(cifar100_dataset.targets) if label == class_index]
    class_sampler_indices_train.extend(class_indices_list_train[:clas
s_subset_size])


train_sampler = SubsetRandomSampler(class_sampler_indices_train)
test_sampler = SubsetRandomSampler(class_sampler_indices_test)

# Create PyTorch data loaders using the balanced subset for both
train and test sets
batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_dataset,
batch_size=batch_size, sampler=train_sampler)
```

در ادامه داده ها را جدا کرده و tensor کرده و یک instance از کلاس SI میسازیم.

```
# Extract data and labels from loaders
cifar100_data, cifar100_labels = [], []
for data, label in train_loader:
    cifar100_data.append(data)
    cifar100_labels.append(label)

# Concatenate the lists of tensors
cifar100_data = torch.cat(cifar100_data, dim=0)
cifar100_labels = torch.cat(cifar100_labels, dim=0)

# Convert to desired format
train_data_tensor = cifar100_data
data_tensor = train_data_tensor.view(train_data_tensor.size(0), -1)
label_tensor = cifar100_labels.unsqueeze(1)

# Create Instance of class
si_calculator = Kalhor_SeparationIndex(data_tensor, label_tensor,
normalize=True)
```

حال میتوانیم شاخص های متفاوت SI را برای هر داده های CIFAR 100 به دست آوریم.

به کمک کد زیر خروجی ها را حساب کرده و در نهایت نسبت تعداد ۱ ها را میشماریم (این کار برای سایر شاخص ها هم انجام میشود.

```
si_data = si_calculator.si_data()

num_ones = torch.sum(si_data == 1).item()
total_elements = si_data.numel()
ratio_ones = num_ones / total_elements

print("First order SI :", ratio_ones)
```

```
si_high_order_2_data = si_calculator.high_order_si_data(order=2)
si_soft_order_2_data = si_calculator.soft_order_si_data(order=2)
anti_si_data = si_calculator.anti_si_data(order=2)
center_si_data = si_calculator.center_si_data()
```

جدول زیر مقادیر SI را بر روی داده های خام CIFAR 100 نشان میدهد.

جدول ۱ : مقادیر شاخص SI بر روی داده های خام CIFAR100

SI	HSI(order2)	H-Soft-SI-2	Anti SI	Center SI
0.163575	0.05444	0.05444	0.80196	0.488

مقادیر فوق نشان میدهد که دیتاست CIFAR 100 یک مجموعه داده مناسب نمیباشد و در حالت خام مقدار SI پایینی دارد. و همانطور که مشاهده می شود مقدار anti SI هم بالا میباشد. شاخص SIای که بدست آوردیم با عددی که در اسلایدهای درس آورده شده مطابقت دارد و درنتیجه به جواب درستی رسیده ایم. علت این کم بودن این شاخص challenging بودن این دیتاست است و نقاط داده با anti si بالاتر نمونه های سختی را در یک مجموعه داده ایجاد می کنند.

DataSet (m=50000)	N. Of Classes	Sepration Index
MNIST Digits	10	0.9722
Fashion MNIST	10	0.85072
Cifar10	10	0.35086
Cifar100	100	0.17446

شکل ۲ – نتایج آورده شده در اسلاید های درس

ج) محاسبه شاخص های SI بر روی داده های آخرین لایه

در این بخش مدلی که در بخش الف آموزش داده بودیم را به کمک دستور زیر لود میکنیم.

```
# Load EfficientNetV2-S model
model = models.efficientnet_v2_s(weights='IMAGENET1K_V1')

# 100 classes
model.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True),
    nn.Linear(1280, 100)
)


transform = transforms.Compose([
```

```
    transforms.Resize(300),
    transforms.CenterCrop(260),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])

# Load your pre-trained model weights
model.load_state_dict(torch.load('efficientnetv2_s_cifar100_finetuned
.pth'))
model.eval()  # Set the model to evaluation mode
model.cuda()
print("Model Loaded")
```

همانند قبل یک subset از داده ها را به کمک کد زیر جدا میکنیم.

```
transform = transforms.Compose([transforms.ToTensor()])

# Load the CIFAR-100 dataset
cifar100_dataset = datasets.CIFAR100(root='./data', train=True,
transform=transform, download=True)

# 80% for SI , 0.03 % for center SI and 30% for other Metrics
subset_size = int(0.03 * len(cifar100_dataset))

# Create a random subset of 20% of the dataset
subset_indices = torch.randperm(len(cifar100_dataset))[:subset_size]
subset_dataset = Subset(cifar100_dataset, subset_indices)

# Define DataLoader for the subset
batch_size = 64
dataloader = DataLoader(subset_dataset, batch_size=batch_size,
shuffle=True)
```

در ادامه نیز خروجی feature ها را بر روی subset به کمک زیر در آخرین لایه قبل از classifier (با توجه به model.eval() مدل دارای ۳ بخش feature و avgpool و classifier میباشد.) به دست می آوریم.

```
features = []
labels = []


with torch.no_grad():
    for inputs, targets  in tqdm(dataloader):
```

```
        if torch.cuda.is_available():
            inputs = inputs.to('cuda')
        # Forward pass through the model.features and model.avgpool
        features_batch = model.features(inputs)
        features_batch = model.avgpool(features_batch)
        features.append(features_batch)
        labels.append(targets)

        # Release GPU memory
        del inputs
        torch.cuda.empty_cache()

# Stack and reshape the extracted features
features = torch.cat(features)
features = features.view(features.size(0), -1)
labels = torch.cat(labels)
labels = labels.unsqueeze(1)
```

مجددا یک instance از کلاس SI را با داده های جدید ساخته و همان مراحل بخش ب را تکرار میکنیم.

```
instance_disturbance = Kalhor_SeparationIndex(features, labels,
normalize=True)
```

جدول زیر به طور خلاصه خروجی های بخش ب و ج را نشان میدهد.

جدول ۲ : مقادیر SI بر روی داده های خام و خروجی اخرین لایع شبکه بر روی دیتاست CIFAR 100

	SI	HSI(order2)	H-Soft-SI-2	Anti SI	Center SI
داده خام	0.163575	0.05444	0.05444	0.80196	0.488
آخرین لایه شبکه	**0.6441**	**0.5055**	**0.5055**	**0.2528**	**0.85**

مشاهده میشود که داده های خام که شاخص SI مناسبی نداشت پس از آموزش شبکه در آخرین لایه شبکه به مقادیر به مراتب بهتر و بالاتری میرسد و نشان میدهد شبکه به خوبی توانسته داده ها را در فضای فیچر ها که در اینجا ۱۲۸۰ تا میباشد جدا کند.

د) محاسبه مقدار cross SI

Cross SI شاخص جداسازی دامنه آزمایشی مجموعه داده $D\text{-}test$ را بر اساس دامنه اصلی مجموعه داده $Data$ اندازه گیری می کند. برای محاسبه مقدار cross با توجه به محدودیت های سخت افزاری باید مجددا یک subset از داده های آموزش و تست برداشت برای این کار از کد زیر استفاده میکنیم. (۲۰ درصد از تست و ترین برداشته شده که معادل ۱۰ هزار سمپل برای ترین و ۲ هزار سمپل برای تست میباشد)

```
# Load the CIFAR-100 dataset and create a balanced subset
transform = transforms.Compose([transforms.ToTensor()])

cifar100_train = datasets.CIFAR100(root='./data', train=True,
download=True, transform=transform)
cifar100_test = datasets.CIFAR100(root='./data', train=False,
download=True, transform=transform)

# Combine original and augmented datasets
cifar100_train_combined =
torch.utils.data.ConcatDataset([cifar100_train])
cifar100_test_combined =
torch.utils.data.ConcatDataset([cifar100_test])

# Define the size of the balanced subset for both train and test sets
subset_fraction_train = 0.2
subset_fraction_test = 0.2

# Calculate the number of samples needed for the balanced subset for
both train and test sets
subset_size_train = int(subset_fraction_train *
len(cifar100_train_combined))
subset_size_test = int(subset_fraction_test *
len(cifar100_test_combined))

# Create a balanced subset for both train and test sets using
SubsetRandomSampler
class_indices_train = list(range(len(cifar100_train.classes)))
class_indices_test = list(range(len(cifar100_test.classes)))

class_subset_size_train = int(subset_size_train /
len(cifar100_train.classes))
class_subset_size_test = int(subset_size_test /
len(cifar100_test.classes))

class_sampler_indices_train = []
class_sampler_indices_test = []
```

```
for class_index in class_indices_train:
    class_indices_list_train = [i for i, label in
enumerate(cifar100_train.targets) if label == class_index]
    class_sampler_indices_train.extend(class_indices_list_train[:clas
s_subset_size_train])

for class_index in class_indices_test:
    class_indices_list_test = [i for i, label in
enumerate(cifar100_test.targets) if label == class_index]
    class_sampler_indices_test.extend(class_indices_list_test[:class_
subset_size_test])

train_sampler = SubsetRandomSampler(class_sampler_indices_train)
test_sampler = SubsetRandomSampler(class_sampler_indices_test)

# Create data loaders using the balanced subset for both train and
test sets
batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_train_combined,
batch_size=batch_size, sampler=train_sampler)
test_loader = torch.utils.data.DataLoader(cifar100_test_combined,
batch_size=batch_size, sampler=test_sampler)

# Check the number of samples in each set
print(f"Balanced Train set size: {len(train_loader.sampler)}")
print(f"Balanced Test set size: {len(test_loader.sampler)}")

output :

Balanced Train set size: 10000
Balanced Test set size: 2000
```

در ادامه داده ها و لیبل ها را tensor کرده و پیش پردازش لازم برای محاسبه cross انجام میدهیم.

```
from torch.utils.data.sampler import SubsetRandomSampler

# Extract data and labels from loaders
cifar100_train_data, cifar100_train_labels = [], []
for data, label in train_loader:
    cifar100_train_data.append(data)
    cifar100_train_labels.append(label)

cifar100_test_data, cifar100_test_labels = [], []
for data, label in test_loader:
    cifar100_test_data.append(data)
    cifar100_test_labels.append(label)
```

```
# Concatenate the lists of tensors
cifar100_train_data = torch.cat(cifar100_train_data, dim=0)
cifar100_train_labels = torch.cat(cifar100_train_labels, dim=0)
cifar100_test_data = torch.cat(cifar100_test_data, dim=0)
cifar100_test_labels = torch.cat(cifar100_test_labels, dim=0)

# Convert to desired format
train_data_tensor = cifar100_train_data
train_data_tensor = train_data_tensor.view(train_data_tensor.size(0),
-1)

train_label_tensor = cifar100_train_labels.unsqueeze(1)

test_data_tensor = cifar100_test_data
test_data_tensor = test_data_tensor.view(test_data_tensor.size(0), -
1)

test_label_tensor = cifar100_test_labels.unsqueeze(1)
```

در نهایت class SI برا فراخوانی کرده و مقدار cross را به کمک کد زیر محاسبه می کنیم.

```
train_si_calculator = Kalhor_SeparationIndex(train_data_tensor,
train_label_tensor, normalize=True)
test_si_calculator = Kalhor_SeparationIndex(test_data_tensor,
test_label_tensor, normalize=True)

# Calculate cross_si_data for the test dataset
cross_si_data = train_si_calculator.cross_si_data(test_data_tensor,
test_label_tensor)
num_ones = torch.sum(cross_si_data == True).item()
total_elements = cross_si_data.numel()
ratio_ones = num_ones / total_elements

print("Cross SI:", ratio_ones)
```

مقدار **Cross SI: 0.1175** میشود که با توجه به این مقدار می توان همچنان به بد قلق بودن دیتاست CIFAR100 اشاره کرد.

و) محاسبه SMI

ابتدا دیتاست دیابت را لود میکنیم، آن را به مجموعه های آموزشی و آزمایشی تقسیم کرده و داده ها را به تنسور تبدیل می کنیم. علاوه بر این، DataLoader را برای تسهیل پردازش دسته ای کارآمد در طول آموزش و ارزیابی مدل ایجاد می کنیم. view (-1, 1) برای تغییر شکل تنسورها (مقادیر target هستند) به بردارهای ستونی، استفاده می شود و آن‌ها را به قالب مورد انتظار برای مدل تبدیل می کند.

```
# Load the diabetes dataset
diabetes = load_diabetes()

# Split the data into features (X) and target (y)
X = diabetes.data
y = diabetes.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Convert NumPy arrays to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)

# Create DataLoader
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)

batch_size = 64

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

حال میتوانیم شاخص های متفاوت SMI را برای هر داده های diabetes به دست آوریم.

به کمک کدهایی که در بخش مثال‌ها آورده شده بود این شاخص ها را بدست می‌آوریم. در جدول زیر نتایج این بخش آورده شده است:

جدول ۳ – مقادیر مختلف SMI بر روی داده های دیتاست diabetes

Linear Smoothness Index	0.7286107	Exponential Smoothness Index	0.56403697
High Smoothness Index with order 2	0.60883486	High Smoothness exp Index with order 2	0.42122817
Anti-Smoothness Index with order 1	0.27138925	Anti-Smoothness exp Index with order 1	0.43596306
Soft linear Smoothness Index with order 2	0.71688646	Soft Smoothness exp Index with order 2	0.55304766
Cross Smootheness Index	0.7635338	Local Smootheness Index	0.48441926

صورت سوال از ما SMI و Cross SMI را می‌خواهد که در جدول بالا مقادیر آن ها مشخص شده است. مجموعه داده‌های آزمایشی با مجموعه داده آموزشی همگن نامیده می‌شود که SMI تقریبا با Cross SMI باشد که در اینجا این مقادیر تقریبا نزدیک به هم هستند و میتوانیم بگوییم که مجموع داده آموزشی همگن است.

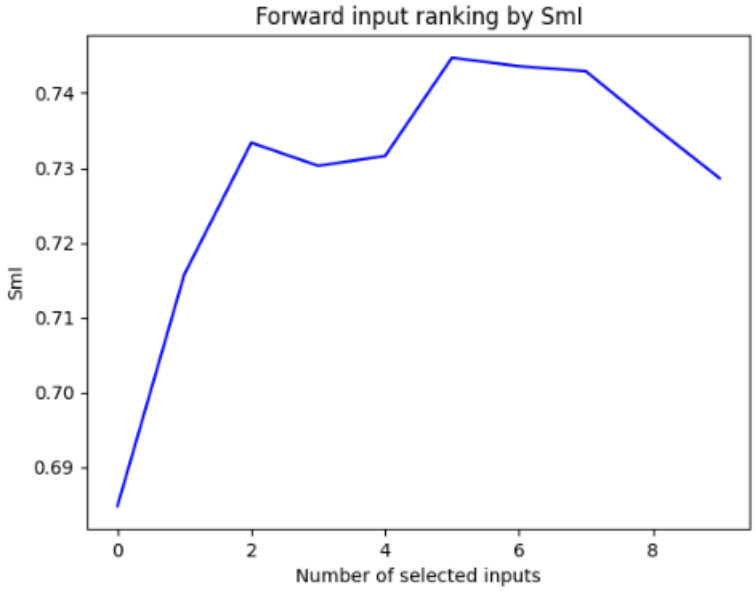
نتایج زیر در اسلاید های درس هستند که می توان نتیجه گرفت که نتایجی که گرفتیم درست هستند.

DataSet	N. Of data points	Sml linear	Smi mean	Cr. Smi linear	Cr. Smi mean
Diabets	(m=353,n=10) (mtest=89,n=10)	0.7286	0.4230	0.7635	0.4739
Car Price	(m=174, n=63),(mtest=31-n=63)	0.9340	0.7784	0.9291	0.7741
California housing	(m=16512,n=8) (m=4128, n=8)	0.7303	0.4005	0.7323	0.4061
Sinc function	(m=900,n=2) (mtest=100,n=2)	0.9840	0.8027	0.9828	0.8295

شکل ۳ – نتایج ذکر شده در اسلایدهای درس

همچنین با استفاده از smi، forward selection انجام دادیم که نتیجه‌ی زیر حاصل شد:

```
=====forward_input_ranking_by_smi========
Ranked features are:  tensor([[8., 2., 5., 0., 7., 4., 1., 6., 9., 3.]])
smi for the best chosen Inputs are:  [[0.6847557  0.71575105 0.7333702  0.7302797  0.7315917  0.7447439
  0.7436048  0.742936   0.7356565  0.7286107 ]]
=====ranked_features_best========
Sepration Index for input_best_smi is:  0.7447439
the best feachers are:  tensor([[8, 2, 5, 0, 7, 4]])
```



شکل ۴ – forward_input_ranking_by_smi

مشاهده میکنیم که با انتخاب ۳ فیچر، SMI ما به بیشترین مقدار خودش می‌رسد.

ه) محاسبه LDI

LDI میانگین چگالی خطی تعدادی از خوشه ها را اندازه گیری می کند. (هر خوشه دارای یک توزیع تک وجهی در اطراف یک نقطه کانونی است)

برای این بخش مجددا همانند قبل Subset ای از داده های تست و آموزش را بالانس کرده که این کار به کمک کد زیر انجام میشود.

```
# Load the CIFAR-100 dataset and create a balanced subset
transform = transforms.Compose([transforms.ToTensor()])

cifar100_train = datasets.CIFAR100(root='./data', train=True,
download=True, transform=transform)
```

```
cifar100_test = datasets.CIFAR100(root='./data', train=False,
download=True, transform=transform)

# Combine original and augmented datasets
cifar100_train_combined =
torch.utils.data.ConcatDataset([cifar100_train])
cifar100_test_combined =
torch.utils.data.ConcatDataset([cifar100_test])

# Define the size of the balanced subset for both train and test sets
subset_fraction_train = 0.1
subset_fraction_test = 0.1

# Calculate the number of samples needed for the balanced subset for
both train and test sets
subset_size_train = int(subset_fraction_train *
len(cifar100_train_combined))
subset_size_test = int(subset_fraction_test *
len(cifar100_test_combined))

# Create a balanced subset for both train and test sets using
SubsetRandomSampler
class_indices_train = list(range(len(cifar100_train.classes)))
class_indices_test = list(range(len(cifar100_test.classes)))

class_subset_size_train = int(subset_size_train /
len(cifar100_train.classes))
class_subset_size_test = int(subset_size_test /
len(cifar100_test.classes))

class_sampler_indices_train = []
class_sampler_indices_test = []

for class_index in class_indices_train:
    class_indices_list_train = [i for i, label in
enumerate(cifar100_train.targets) if label == class_index]
    class_sampler_indices_train.extend(class_indices_list_train[:clas
s_subset_size_train])

for class_index in class_indices_test:
    class_indices_list_test = [i for i, label in
enumerate(cifar100_test.targets) if label == class_index]
    class_sampler_indices_test.extend(class_indices_list_test[:class_
subset_size_test])

train_sampler = SubsetRandomSampler(class_sampler_indices_train)
test_sampler = SubsetRandomSampler(class_sampler_indices_test)
```

```
# Create PyTorch data loaders using the balanced subset for both
train and test sets
batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_train_combined,
batch_size=batch_size, sampler=train_sampler)
test_loader = torch.utils.data.DataLoader(cifar100_test_combined,
batch_size=batch_size, sampler=test_sampler)
```

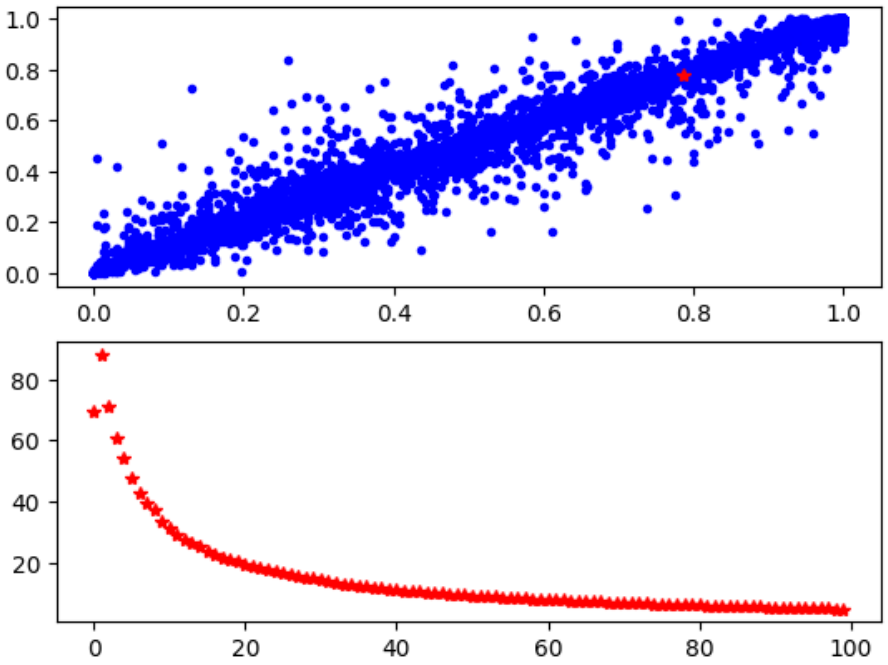
در نهایت clustering را به کمک کد زیر پس از پیش پردازش های لازم انجام میدهیم (با توجه به محدودیت های سخت افزاری تنهای ۲۰ درصد داده ها مورد ارزیابی قرار گرفته است و مقدار k means repeat نیز برابر با ۲۰ قرار داده شده است)

```
# Extract data and labels from loaders
cifar100_train_data, cifar100_train_labels = [], []
for data, label in train_loader:
    cifar100_train_data.append(data)
    cifar100_train_labels.append(label)

cifar100_test_data, cifar100_test_labels = [], []
for data, label in test_loader:
    cifar100_test_data.append(data)
    cifar100_test_labels.append(label)

# Concatenate the lists of tensors
cifar100_train_data = torch.cat(cifar100_train_data, dim=0)
cifar100_train_labels = torch.cat(cifar100_train_labels, dim=0)
cifar100_test_data = torch.cat(cifar100_test_data, dim=0)
cifar100_test_labels = torch.cat(cifar100_test_labels, dim=0)

# Convert to desired format
train_data_tensor = cifar100_train_data
train_data_tensor = train_data_tensor.view(train_data_tensor.size(0),
-1)

train_label_tensor = cifar100_train_labels.unsqueeze(1)

test_data_tensor = cifar100_test_data
test_data_tensor = test_data_tensor.view(test_data_tensor.size(0), -
1)
test_label_tensor = cifar100_test_labels.unsqueeze(1)
```

```
rl , cross =
module_data_domain_scoring_unsupervised(train_data_tensor ,
test_data_tensor)
```

تعداد کلاستر پیدا شده توسط الگوریتم ۲ میباشد (بر روی داده های خام) که باز هم نشان از سخت بودن و بد قلق بودن دیتاست CIFAR 100 دارد.

منحنی های دیتا کلاستر و center کلاستر ها نیز به صورت زیر میشود.



شکل ۵: توزیع دیتا و مرکز دسته کلاستر و منحنی **av_lin_den** برحسب تعداد کلاستر

در شکل بالا، اولین پلات کل توزیع داده هایی که دادیم به تابع، ستاره ای که در شکل میبینیم، مرکز دسته کلاستر است.

در شکل پایینی، مقدار av_lin_density بر اساس تعداد کلاستر رسم شده است. و میدانیم که پیک این نمودار تعداد کلاسترها را نشان می دهد.

مقدار relative density و cross relative density نیز به ترتیب برابر با **۰,۳۱۶۰** و **۰.۳۰۴۶** میشود.

سوال دوم : FEATURE SELECTION

در این بخش هدف بررسی Forward feature Selection به کمک شاخص SI میباشد. در اینجا ابتدا فیچری را انتخاب میکند که بیشترین SI را داشته باشد و بعد دومین فیچری را انتخاب میکند که با اولی باعث افزایش SI شود این کار ادامه پیدا میکند تا تمام featureها انتخاب شوند.

ابتدا مدل از قبل آموزش داده شده را به صورت زیر لود میکنیم.

```
# Load EfficientNetV2-S model
model = models.efficientnet_v2_s(weights='IMAGENET1K_V1')

# 100 classes
model.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True),
    nn.Linear(1280, 100)
)

# Define a transformation for the CIFAR-100 dataset (make sure it
matches the one used during training)
transform = transforms.Compose([
    transforms.Resize(300),  # EfficientNetV2-S expects 300x300 input
    transforms.CenterCrop(260),  # Center crop to 260x260
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])  # ImageNet mean and std
])

# Load your pre-trained model weights
model.load_state_dict(torch.load('efficientnetv2_s_cifar100_finetuned
.pth'))
model.eval()  # Set the model to evaluation mode
model.cuda()
print("Model Loaded")
```

با توجه به محدودیت های سخت افزاری و ران تایم بالای این بخش یک subset که ۲۰ درصد داده ها را شامل میشود جدا میکنیم.

```
# Define the transformation
transform = transforms.Compose([transforms.ToTensor()])

# Load the CIFAR-100 dataset
cifar100_dataset = datasets.CIFAR100(root='./data', train=True,
transform=transform, download=True)

subset_size = int(0.2 * len(cifar100_dataset))
# Create a random subset of 20% of the dataset
subset_indices = torch.randperm(len(cifar100_dataset))[:subset_size]
subset_dataset = Subset(cifar100_dataset, subset_indices)

# Define your DataLoader for the subset
batch_size = 64
dataloader = DataLoader(subset_dataset, batch_size=batch_size,
shuffle=True)
```

در ادامه به کمک کد زیر آن را از شبکه عبور داده (تا قبل از classifier)

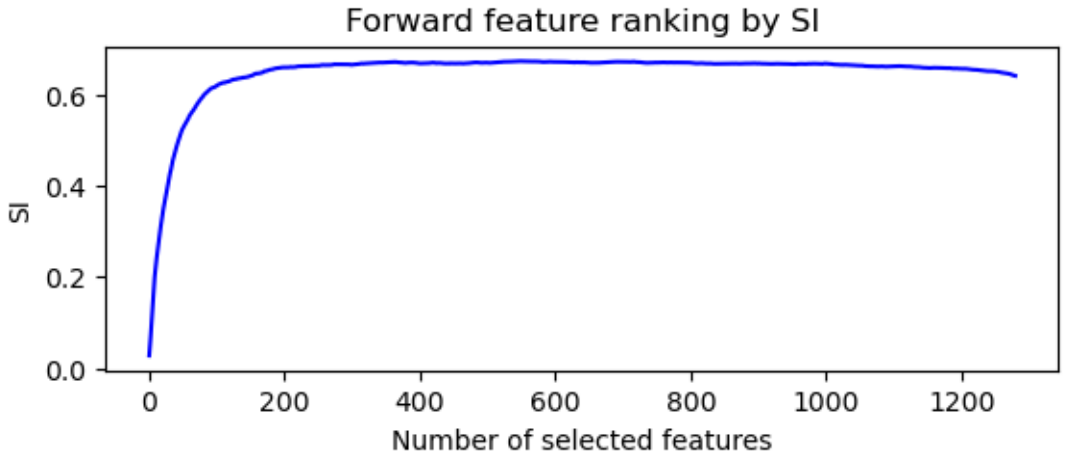
```
features = []
labels = []


with torch.no_grad():
    for inputs, targets  in tqdm(dataloader):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda')
        # Forward pass through the model.features
        features_batch = model.features(inputs)
        features_batch = model.avgpool(features_batch)
        features.append(features_batch)
        labels.append(targets)


        # Release GPU memory
        del inputs
        torch.cuda.empty_cache()

# Stack and reshape the extracted features
features = torch.cat(features)
features = features.view(features.size(0), -1)
labels = torch.cat(labels)
```

```
labels=labels.unsqueeze(1)
```

در نهایت یک instance از کلاس ساخته و الگوریتم forward را بر روی آن اعمال میکنیم.

```
instance_disturbance = Kalhor_SeparationIndex(features, labels,
normalize=True)
si_ranked_features, ranked_features =
instance_disturbance.forward_feature_ranking_si()
```

منحنی زیر فرایند feature selection از ۱۲۸۰ فیچر نهایی را نشان میدهد. (مقدار SI بر حسب تعداد فیچر) – اجرای این الگوریتم بر روی ۱۰ هزار سمپل حدود ۴ ساعت طول کشید.



شکل ۶ : منحنی SI بر حسب فیچر های خروجی آخرین لایه

همانطور که در منحنی بالا مشاهده می شود با تعداد فیچر های کمتری نسبت به ۱۲۸۰ تا میشود SI بالایی در فضای فیچر ها داشته که در اینجا با ۵۴۵ فیچر مقدار SI به 0.6744 میرسد در حالیکه با ۱۲۸۰ تا فیچر مقدار SI برابر با 0.6441 بود.

با مشخص شدن لیست ۵۴۵ تایی از فیچر هایی که باید از ۱۲۸۰ فیچر لایه آخر انتخاب شوند در ادامه بر روی تمام دیتاست آموزش را تکرار میکنیم.

```
np.max(si_ranked_features.detach().cpu().numpy()[0])
si_ranked_features = si_ranked_features.detach().cpu().numpy()[0]
max_index = np.argmax(si_ranked_features)
feat = ranked_features[0][:max_index]
```

به کمک کد زیر ابتدا یک کلاس MLP جدا تشکیل داده و با توجه به تعداد فیچر های انتخاب شده ابعاد ورودی آن را تنظیم میکنیم.

```
class MLPModel(nn.Module):
    def __init__(self, input_size, output_size, dropout_rate=0.2):
        super(MLPModel, self).__init__()
        self.flatten = nn.Flatten()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.dropout(x)
        x = self.fc(x)
        return x


input_size = len(feat)
output_size = 100
mlp_model = MLPModel(input_size, output_size)
```

به کمک کد زیر دیتاست را لود کرده و optimizer و ... را تعریف میکنیم.

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(mlp_model.parameters(), lr=0.001)

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

# Move models to device
model.to(device)
mlp_model.to(device)

# Load CIFAR-100 dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])


train_dataset = torchvision.datasets.CIFAR100(root='./data',
train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,
num_workers=2)
```

```
test_dataset = torchvision.datasets.CIFAR100(root='./data',
train=False, download=True, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,
num_workers=2)
```

برای لوپ ترین مدل قبلی و مدل MLP جدید و تمام داده ها بر روی cuda برده و در ادامه فرایند بدین صورت است که با توجه به freeze بودن مدل اصلی فقط پارامتر های learnable برای MLP-Model تعریف میشود و با توجه به کد زیر داده وارد مدل تا سر classifier شده و سپس با توجه به لیست فیچر هایی که در مرحله قبلی به دست آمده بود یک slice روی ۱۲۸۰ تا فیچر خروجی شبکه رفته تا فقط فیچر ها مد نظر پیدا شود و در نهایت flatten شده و وارد MLP شود برای ورودی دادن کد زیر زده شده است

```
# Get features using the pretrained model
features_batch = model.features(inputs)
features_batch = model.avgpool(features_batch)
x = features_batch[:, feat.long(), :, :]

# Forward pass through the MLP model
outputs = mlp_model(x)
```

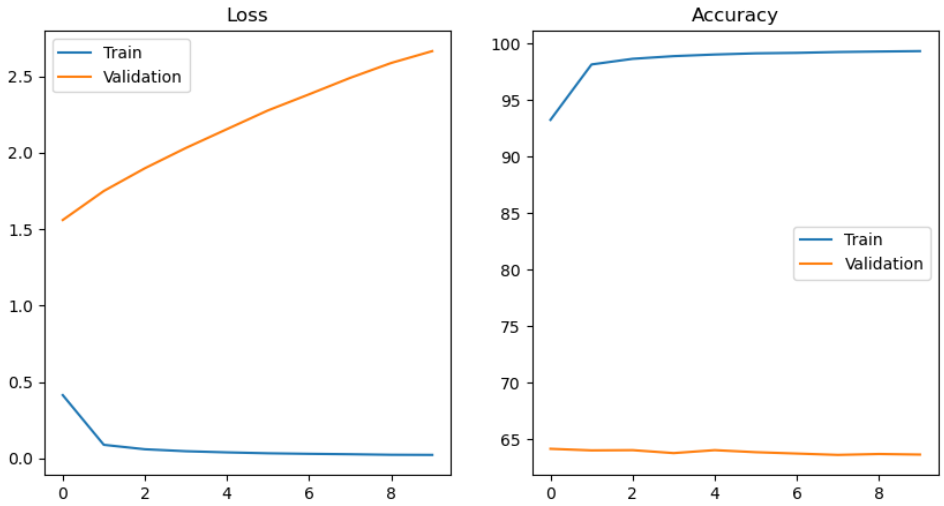
بنابراین کل لوپ ترین به صورت زیر میباشد.

```
# Training and validation loop
num_epochs = 10

train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []

for epoch in range(num_epochs):
    mlp_model.train()
    total_loss = 0.0
    correct = 0
    total = 0

    # Use tqdm for progress bar
    with tqdm(train_loader, desc=f'Epoch {epoch + 1}/{num_epochs}',
unit='batch') as tqdm_loader:
        for inputs, labels in tqdm_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # Get features using the pretrained model
```

```
            features_batch = model.features(inputs)
            features_batch = model.avgpool(features_batch)
            x = features_batch[:, feat.long(), :, :]

            # Forward pass through the MLP model
            outputs = mlp_model(x)

            # Calculate loss and perform backpropagation
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

            # Update tqdm description with current loss
            tqdm_loader.set_postfix({'Loss': total_loss / total})

    # Calculate training accuracy and loss
    train_accuracy = 100 * correct / total
    train_losses.append(total_loss / len(train_loader))
    train_accuracies.append(train_accuracy)

    # Validation loop
    mlp_model.eval()
    with torch.no_grad():
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        for val_inputs, val_labels in test_loader:
            val_inputs, val_labels = val_inputs.to(device),
val_labels.to(device)

            val_features_batch = model.features(val_inputs)
            val_features_batch = model.avgpool(val_features_batch)
            val_x = val_features_batch[:, feat.long(), :, :]

            val_outputs = mlp_model(val_x)
            val_loss += criterion(val_outputs, val_labels).item()

            _, val_predicted = val_outputs.max(1)
            val_total += val_labels.size(0)
            val_correct += val_predicted.eq(val_labels).sum().item()
```

```
        # Calculate validation accuracy and loss
        val_accuracy = 100 * val_correct / val_total
        val_losses.append(val_loss / len(test_loader))
        val_accuracies.append(val_accuracy)

        print(f'Train Loss: {train_losses[-1]}, Train Accuracy:
{train_accuracies[-1]}%, '
              f'Val Loss: {val_losses[-1]}, Val Accuracy:
{val_accuracies[-1]}%')
```

منحنی زیر مقدار لاس و دقت را بر روی داده های ارزیابی و آموزش نشان میدهد که با توجه به منحنی زیر بهترین دقت در ایپاک چهارم رخ داده است و بعد از آن مدل **overfit** شده است.



شکل ۷ : منحنی لاس و دقت بر روی داده های آموزش و ارزیابی پس از آموزش MLP بر روی فیچر های انتخاب شده

مقدار دقت بر روی مدل آموزش دیده شده نهایی به شرح زیر است.

Val Accuracy: 63.87%

در سوال ۱ بخش الف به کمک ۱۲۸۰ تا فیچر مدل روی داده های ارزیابی به دقت ۶۴,۸۷ رسیده بود که اینجا با تعداد فیچر های به مراتب کمتر به همان حدود دقت رسیدیم.

(فرایندی که خروجی های نشان داد شده است forward selection به کمک ۲۰ درصد داده ها انجام شده است و ما این مرحله را با تعداد داده های کمتر هم انجام دادیم به صورتی که تنها حدود ۲۰۰ تا فیچر انتخاب شد که مشاهده شد مدل با همان ۲۰۰ تا فیچر هم به همین حدود دقت ۶۳ درصد میرسد)

سوال سوم : قوی تر کردن مجموعه داده ها و ارزیابی داده ها

در این سوال از ما خواسته شده بود که augmentationهای زیر را روی داده های ترین اعمال کنیم و نتایج هر کدام را روی دقت شبکه و همچنین روی شاخص SI Cross حساب کنیم.

Augmentationها استفاده شده به ترتیب:

RandomRain – ToGray – CLAHE -AdvancedBlur - GaussNoise

در ادامه اثر هرکدام از این augmentaionها را نشان می‌دهیم.

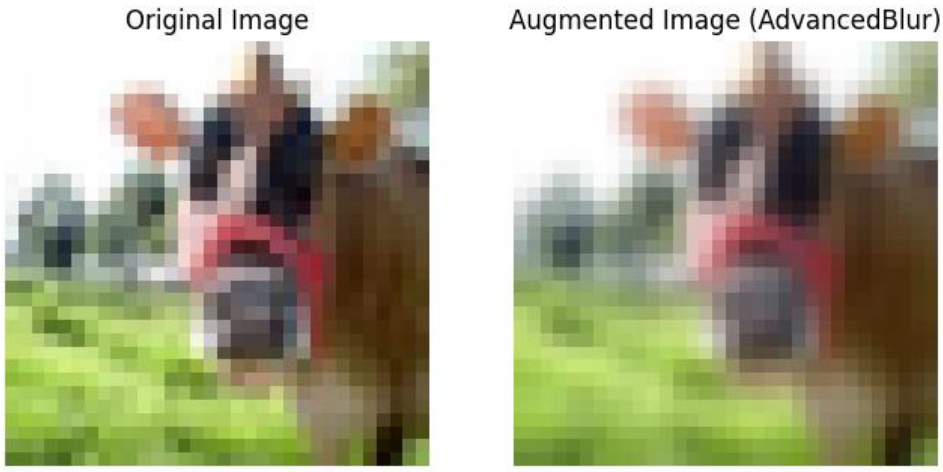
- CLAHE:

Contrast Limited Adaptive Histogram Equalizationرا روی تصویر ورودی اعمال می کند.
در شکل زیر خروجی این نوع تقویت داده را می بینیم.

شکل ۸ – عکس اصلی و عکس augment شده با CLAHE

- AdvancedBlur:

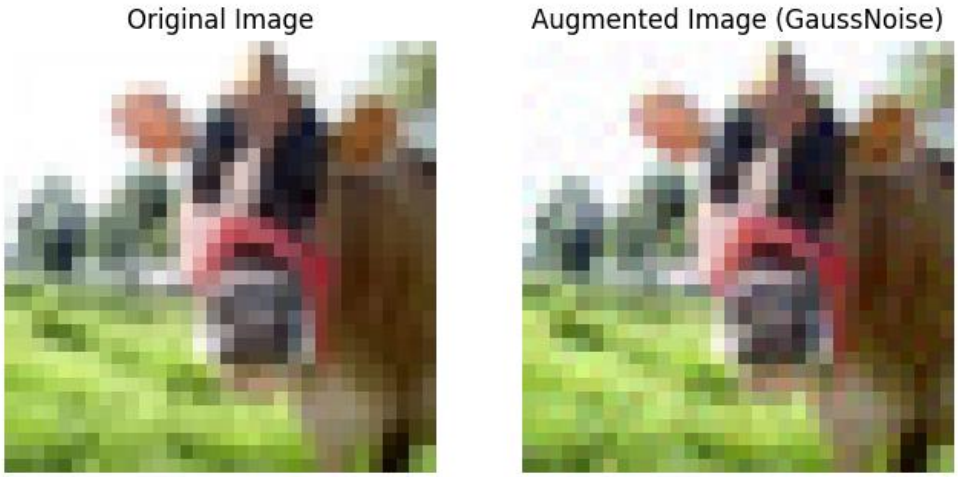
تصویر ورودی را با استفاده از یک فیلتر معمولی تعمیم یافته با پارامترهای انتخابی تصادفی تار می‌کند.
این تبدیل همچنین نویز ضربی را به هسته تولید شده قبل از کانولوشن اضافه می کند.



شکل ۹ - عکس اصلی و عکس augment شده با AdvancedBlur

- GaussNoise:

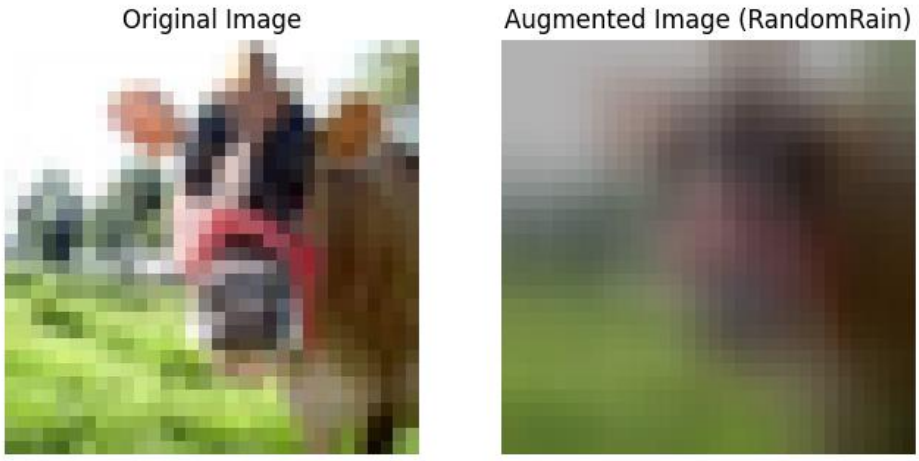
نویز گاوسی را به تصویر ورودی اعمال می‌کند.



شکل ۱۰ – عکس اصلی و عکس augment شده با GaussNoise

- RandomRain:

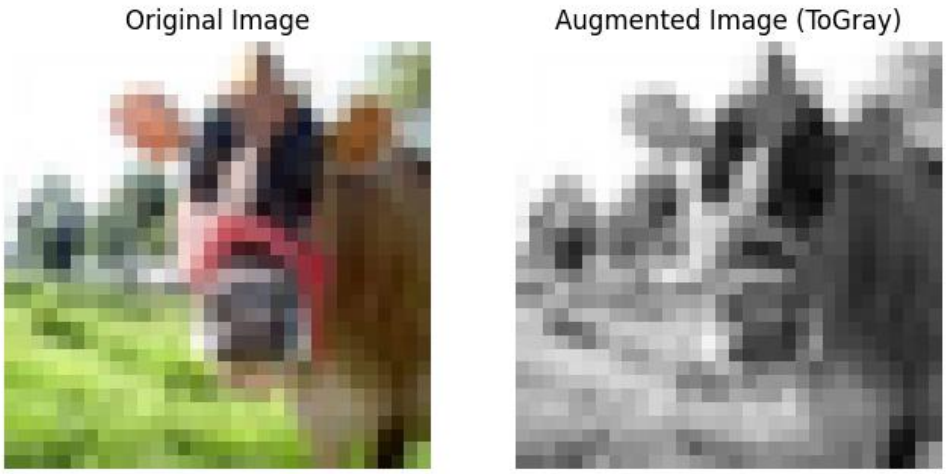
جلوه های باران را اضافه می کند.



شکل ۱۱ – عکس اصلی و عکس **augment** شده با **RandomRain**

- ToGray:

تصویر RGB ورودی را به مقیاس خاکستری تبدیل می کند. اگر مقدار میانگین پیکسل برای تصویر حاصل بیشتر از ۱۲۷ باشد، تصویر حاصل از مقیاس خاکستری را معکوس می کند.



شکل ۱۲ – عکس اصلی و عکس **augment** شده با **ToGray**

الف) آموزش شبکه

در مرحله‌ی بعدی ما ۵ مدل خواهیم داشت که در هر کدارم یکی از این روش ها استفاده شده است. ابتدا داده‌های cifar100 را لود میکنیم و augmentationهای مورد نظر را به شکل زیر انجام می‌دهیم. توجه داشته باشید که داده های تست همان داده های تست اصلی هستند و روی آن ها augmentation

انجام نمیدهیم و همچنین برای ترین مدل ها از داده های اصلی و داده تقویت شده به صورت همزمان باید استفاده کنیم که در کد زیر نشان دادهایم که چگونه اینکار را انجام دهیم.

```
image_transform = A.Compose([
    A.CLAHE(clip_limit=4.0, tile_grid_size=(8, 8),
always_apply=False, p=0.5),
])

# Define data transformations including the Albumentations transform
transform1 = transforms.Compose([
    # transforms.ToPILImage(),
    transforms.Lambda(lambda img:
image_transform(image=np.array(img))["image"]),
    transforms.ToTensor(),
    # transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform2 = transforms.Compose([
    # transforms.ToPILImage(),
    transforms.ToTensor(),
    # transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-100 dataset without Albumentations transform
cifar100_original = datasets.CIFAR100(root='./data', train=True,
download=True, transform=transforms.ToTensor())
cifar100_train_augmented = datasets.CIFAR100(root='./data',
train=True, download=True, transform=transform1)
cifar100_test_augmented = datasets.CIFAR100(root='./data',
train=False, download=True, transform=transform2)

# Combine original and augmented datasets
cifar100_train_combined =
torch.utils.data.ConcatDataset([cifar100_original,
cifar100_train_augmented])
cifar100_test_combined =
torch.utils.data.ConcatDataset([cifar100_test_augmented])

# Create PyTorch data loaders
batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_train_combined,
batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(cifar100_test_combined,
batch_size=batch_size, shuffle=False)

# Check the number of samples in each set
```

```
print(f"Combined Train set size: {len(cifar100_train_combined)}")
print(f"Combined Test set size: {len(cifar100_test_combined)}")
```

تعداد داده‌های ما بعد از **augmentation** بصورت زیر می‌باشد:

```
Combined Train set size: 100000
Combined Test set size: 10000
```

دیتاست **cifar100** حدود ۵۰۰۰۰ داده های ترین دارد که بعد از **augmentation** مشاهده میکنیم که تعداد داده های ترین به ۱۰۰۰۰۰ رسیده است.

در قدم بعدی همانند سوال ۱ مدل را پیاده سازی میکنیم و مدل را روی این داده ها ترین میکنیم.

```
# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = StepLR(optimizer, step_size=4, gamma=0.2)

# Training loop
num_epochs = 20
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

train_losses, val_losses = [], []
train_acc, val_acc = [], []

best_val_acc = 0.0
best_model_path = 'efficientnetv2_s_cifar100_best.pth'

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for inputs, labels in tqdm(train_loader, desc=f'Epoch {epoch +
1}/{num_epochs}'):
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        train_losses.append(running_loss / len(train_loader))
        train_acc.append(100 * correct_train / total_train)

        model.eval()
        running_loss = 0.0
        correct_val = 0
        total_val = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                running_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total_val += labels.size(0)
                correct_val += (predicted == labels).sum().item()

        val_losses.append(running_loss / len(val_loader))
        val_acc.append(100 * correct_val / total_val)


        scheduler.step()

        # Check if the current model has the best validation accuracy
        if val_acc[-1] > best_val_acc:
            best_val_acc = val_acc[-1]
            torch.save(model.state_dict(), best_model_path)

        print(f"Epoch {epoch + 1}/{num_epochs} | Train Loss:
{train_losses[-1]:.4f} | Train Acc: {train_acc[-1]:.2f}% | Val Loss:
{val_losses[-1]:.4f} | Val Acc: {val_acc[-1]:.2f}%")

# Load the best model
model.load_state_dict(torch.load(best_model_path))

# Save model weights
torch.save(model.state_dict(),
'efficientnetv2_s_cifar100_augmented_first_method.pth')
```

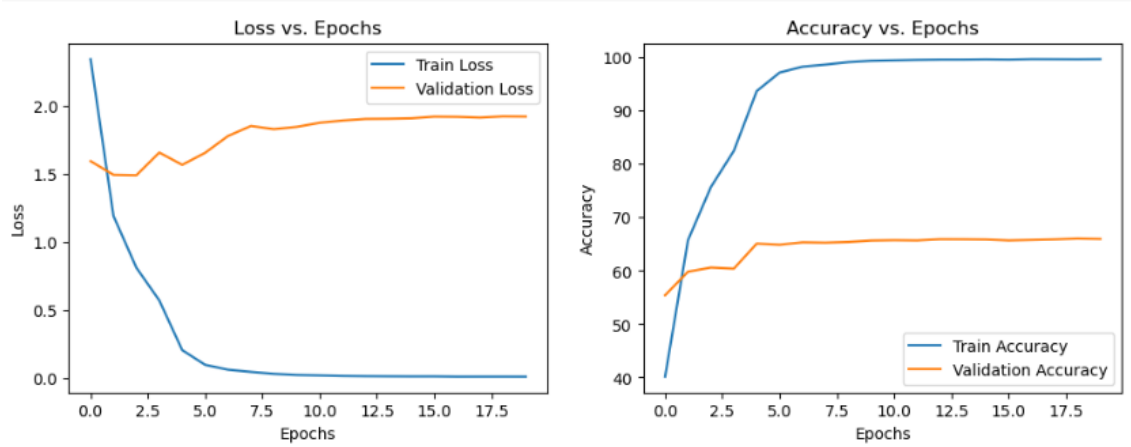
```
Epoch 1/20: 100%|██████████| 391/391 [01:04<00:00,  6.09it/s]
Epoch 1/20 | Train Loss: 2.3408 | Train Acc: 40.15% | Val Loss: 1.5929 | Val Acc: 55.42%
Epoch 2/20: 100%|██████████| 391/391 [01:01<00:00,  6.36it/s]
Epoch 2/20 | Train Loss: 1.1910 | Train Acc: 65.75% | Val Loss: 1.4926 | Val Acc: 59.83%
Epoch 3/20: 100%|██████████| 391/391 [01:01<00:00,  6.34it/s]
Epoch 3/20 | Train Loss: 0.8116 | Train Acc: 75.72% | Val Loss: 1.4893 | Val Acc: 60.61%
Epoch 4/20: 100%|██████████| 391/391 [01:01<00:00,  6.36it/s]
Epoch 4/20 | Train Loss: 0.5720 | Train Acc: 82.49% | Val Loss: 1.6559 | Val Acc: 60.40%
Epoch 5/20: 100%|██████████| 391/391 [01:01<00:00,  6.40it/s]
Epoch 5/20 | Train Loss: 0.2078 | Train Acc: 93.70% | Val Loss: 1.5659 | Val Acc: 65.10%
```

شکل ۱۳- ترین مدل روی داده ها

همانند کاری که در بالا انجام دادیم، برای augmentationهای دیگر هم انجام می‌دهیم و مدل را روی داده های augment شده ترین می‌کنیم. (کدهای بقیه تقویت سازی‌ها در فایل سوال ۳ آورده شده است)

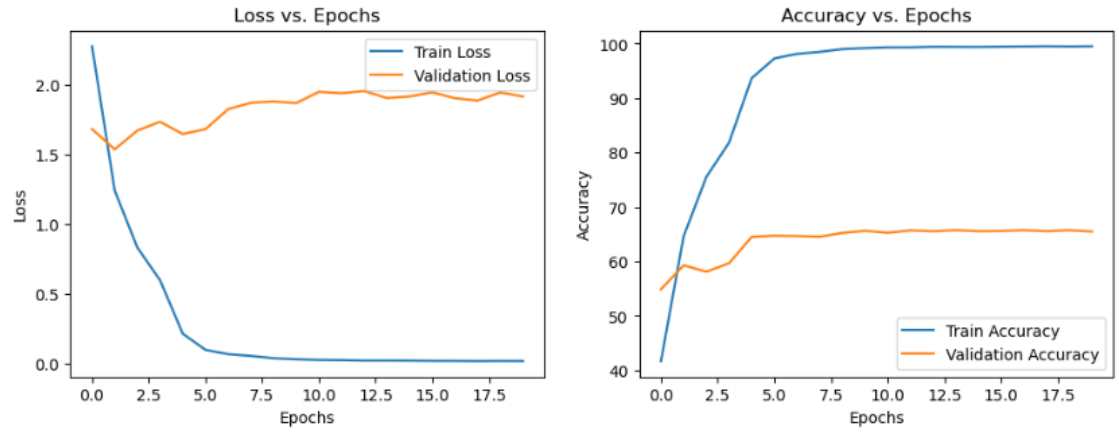
نتایج بدست آمده به شرح زیر است:

- CLAHE:



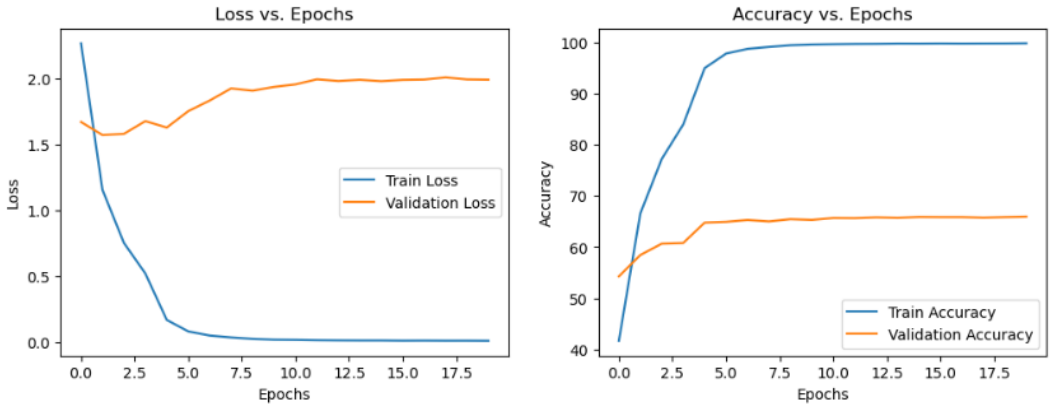
شکل ۱۴ – نمودار دقت و تابع هزینه داده های آموزشی و ارزیابی برای داده های تقویت شده با CLAHE

- AdvancedBlur:



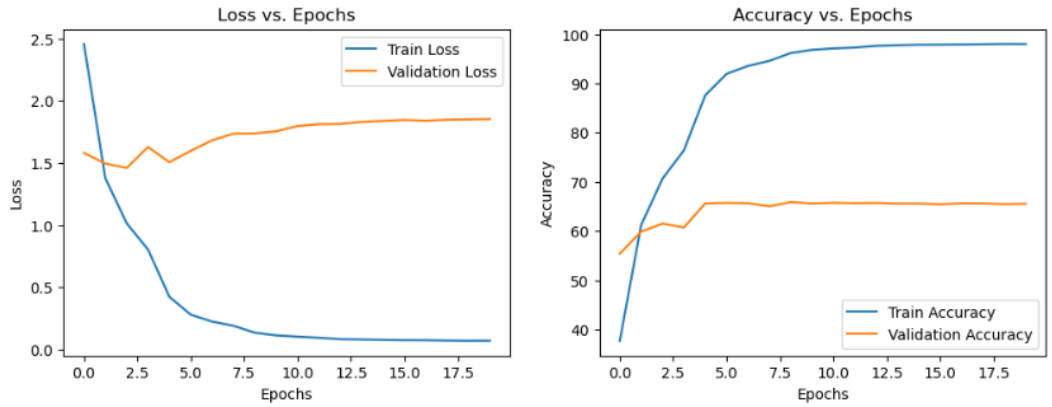
شکل ۱۵ – نمودار دقت و تابع هزینه داده های آموزشی و ارزیابی برای داده های تقویت شده با AdvancedBlur

- GaussNoise:



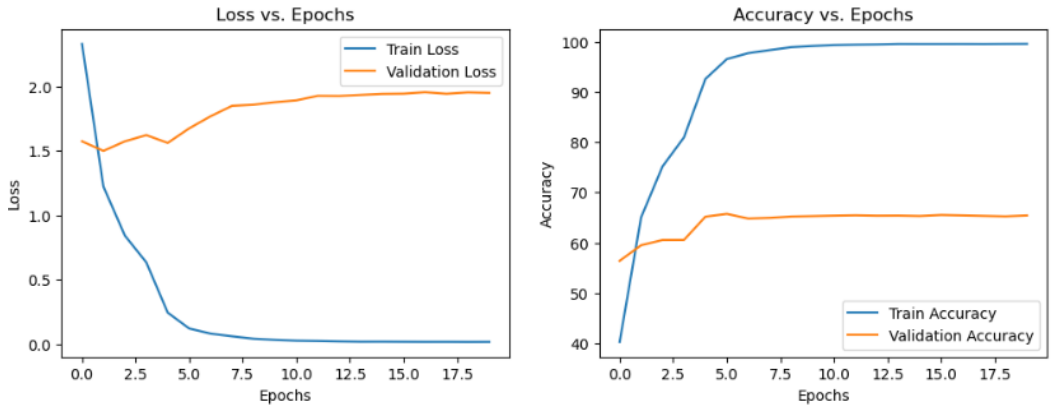
شکل ۱۶ – نمودار دقت و تابع هزینه داده های آموزشی و ارزیابی برای داده های تقویت شده با **GaussNoise**

- RandomRain:



شکل ۱۷ – نمودار دقت و تابع هزینه داده های آموزشی و ارزیابی برای داده های تقویت شده با **RandomRain**

- ToGray:



شکل ۱۸ – نمودار دقت و تابع هزینه داده های آموزشی و ارزیابی برای داده های تقویت شده با **ToGray**

دقت نهایی شبکه بر روی داده های تست در جدول زیر آمده است:

جدول ٤ – دقت نهایی شبکه بر روی داده ها

نوع augmentation	دقت نهایی شبکه
داده های بدون augmentation	64.86%
CLAHE	66.05%
AdvancedBlur	65.73%
GaussNoise	65.92%
RandomRain	65.87%
ToGray	65.74%

مشاهده میکنیم که با تقویت داده ها دقت نهایی شبکه رو داده های تست مقداری زیاد شده است نسبت به حالتی که ما هیچ augmentationای نداشتیم (در حدود ۱ درصد). دلیل آن این است که تقویت داده ها دقت مدل را در مجموعه آموزشی با معرفی تبدیل های متنوع به داده های اصلی افزایش می دهد و توانایی مدل را برای تعمیم به الگوها و تغییرات مختلف افزایش می دهد. این تکنیک با جلوگیری از به خاطر سپردن جزئیات خاص در داده‌های آموزشی توسط مدل، overfit را کاهش می‌دهد و تغییر ناپذیری نسبت به تبدیل‌ها را تقویت می‌کند و عملکرد در نمونه‌های دیده نشده را بهبود می‌بخشد.
تقویت داده ها به عنوان یک روش منظم عمل می کند، اندازه مؤثر مجموعه داده آموزشی را گسترش می دهد و یک مدل قوی تر و تعمیم یافته تر را ترویج می کند.

همچنین میبینیم که بین ۵ نوع تقویت داده زیاد اختلاف دقت نداریم و همگی در یک حدود هستند. بهتر بود که همه‌ی آن را به دیتاست اضافه میکردیم و نتیجه را مشاهده می‌کردیم.

ب) SI Cross

در قدم بعدی SI Cross داده ها را بدست می‌آوریم. با توجه به اینکه محدیدیت منابع داریم فقط از ٪۲۰ از داده ها استفاده کردیم (همانند سوال ۱)

در کد زیر یک subset از دیتاست را انتخاب میکنیم، توجه باید داشته باشیم که این subset باید balanced باشد:

```
# Set random seed for reproducibility
seed = 42
torch.manual_seed(seed)

# Load CIFAR-100 dataset
transform = transforms.Compose([
    transforms.ToTensor(),
])

train_dataset = torchvision.datasets.CIFAR100(root='./data',
train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR100(root='./data',
train=False, download=True, transform=transform)

# Create a balanced subset of 20% of each train and test set
train_indices, _ = train_test_split(list(range(len(train_dataset))),
test_size=0.8, stratify=train_dataset.targets, random_state=seed)
test_indices, _ = train_test_split(list(range(len(test_dataset))),
test_size=0.8, stratify=test_dataset.targets, random_state=seed)

balanced_train_subset = Subset(train_dataset, train_indices)
balanced_test_subset = Subset(test_dataset, test_indices)

# Define augmentation for the train set
image_transform1 = A.Compose([
    A.CLAHE(clip_limit=4.0, tile_grid_size=(8, 8),
always_apply=False, p=0.5),
    ToTensorV2(),
])

# Define data transformations including the Albumentations transform
image_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Lambda(lambda img:
image_transform1(image=np.array(img))['image']),
])
```

```
# Apply augmentation to the train set
augmented_train_dataset = []
for idx in train_indices:
    img, label = train_dataset[idx]
    augmented = image_transform(img)
    augmented_train_dataset.append((augmented, label))

# Combine the original subset and the augmented subset
combined_train_dataset =
torch.utils.data.ConcatDataset([balanced_train_subset,
augmented_train_dataset])

# Create data loaders
batch_size = 256
train_loader = DataLoader(combined_train_dataset,
batch_size=batch_size, shuffle=True)
test_loader = DataLoader(balanced_test_subset, batch_size=batch_size,
shuffle=False)

# Check the length of the datasets
print(f"Combined Train Dataset Length:
{len(combined_train_dataset)}")
print(f"Balanced Test Subset Length: {len(balanced_test_subset)}")
```

تعداد داده‌های بدست آمده به شرح زیر است:

```
Combined Train Dataset Length: 20000
Balanced Test Subset Length: 2000
```

همانند کاری که در بالا انجام دادیم را برای بقیه روش‌های تقویت داده نیز تکرار میکنیم.

در قدم بعدی همانند سوال ۱ SI Cross را محاسبه می کنیم.

```
# Calculate cross_si_data for the test dataset
cross_si_data = train_si_calculator.cross_si_data(test_data_tensor,
test_label_tensor)
num_ones = torch.sum(cross_si_data == True).item()
total_elements = cross_si_data.numel()
ratio_ones = num_ones / total_elements

print("Cross SI:", ratio_ones)
```

(توضیحات کامل این بخش در سوال ۱ آمده است)

با ران کردن کد این قسمت نتایج زیر بدست می‌آید:

جدول ۵ – نتایج شاخص SI Cross

نوع augmentation	SI Cross
داده های بدون augmentation	0.1175
CLAHE	0.116
AdvancedBlur	0.1195
GaussNoise	0.115
RandomRain	0.115
ToGray	0.117

مشاهده میکنیم که شاخص SI Cross کاهش پیدا کرده است، که این نتیجه قابل پیشبینی است چون با augmentation دیتاها را بیشتر و پیچیده تر کرده‌ایم. در اسلاید ها هم داشتیم که هرچقدر داده ها بدقلق تر باشند SI cross آن ها کم خواهد شد.