به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

درس تحلیل و طراحی شبکه های عصبی عمیق

تمرین شماره ۴

نام و نام خانوادگی : علیرضا حسینی – کیانا هوشانفر

شماره دانشجویی : ۸۱۰۱۰۱۱۴۲ - ۸۱۰۱۰۱۳۶۱

دی ماه ۱۴۰۲

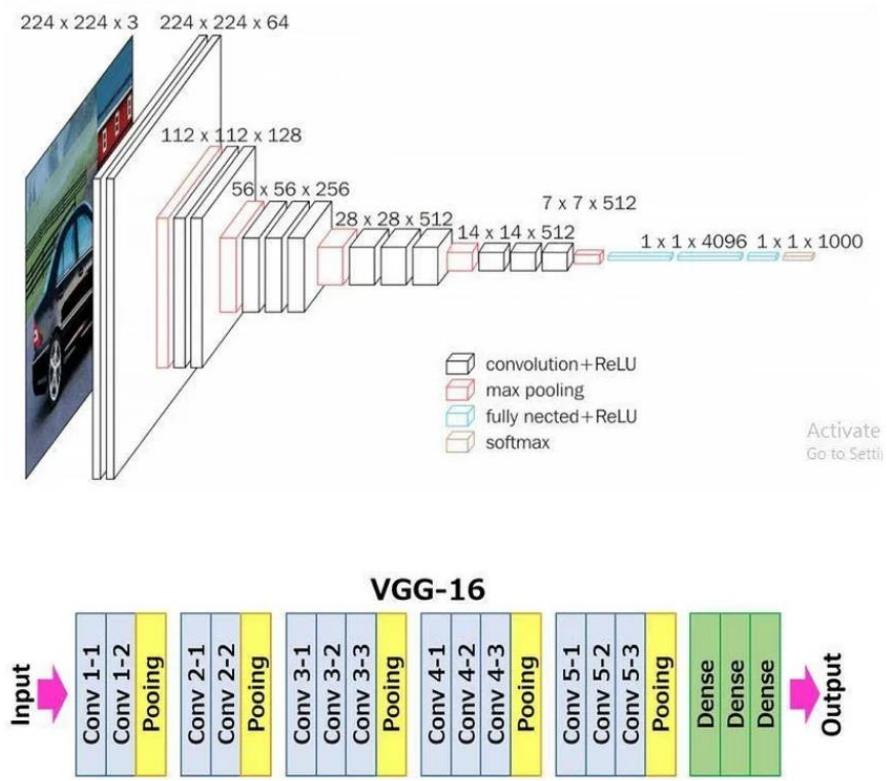
فهرست

مقدمه

هدف از انجام این تمرین آشنایی با فشرده سازی شبکه های عصبی است. فشرده سازی شبکه های عصبی امری مهم در توسعه و بهینه سازی مدل های شبکه عصبی است که به تازگی به عنوان یکی از جنبه های کلیدی در زمینه یادگیری عمیق شناخته شده است. با کاهش تعداد پارامترها، این فناوری تاثیر چشمگیری بر سرعت آموزش و پیشبینی مدل دارد، در عین حال که نیاز به حافظه را نیز به حداقل میرساند. به عبارت دیگر، هدف اصلی از اجرای این تمرین، بهبود کارایی و کاربردیتر کردن شبکه های عصبی است.

سوال ۱) فشرده سازی شبکه عصبی

الف)



شکل ۱ – معماری vgg16

مدل vgg16 را به شکل زیر پیاده سازی می کنیم: ابعاد را مناسب برای آموزش داده های cifar10 انتخاب میکنیم.

```
import torch
import torch.nn as nn

class VGG(nn.Module):
    def __init__(self):
        super(VGG, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
```

```
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(25088, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 10)
        )
    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

# Instantiate the VGG model
vgg16 = VGG()
# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Move the model to the GPU
vgg16.to(device)
```

برای بهتر شدن روند آموزش میتوانیم وزن ها را بصورت رندوم بصورت زیر initialize کنیم:

```
import torch.nn.init as init
def initialize_weights(m):
    if isinstance(m, nn.Conv2d):
        init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
        if m.bias is not None:
            init.constant_(m.bias, 0)
    elif isinstance(m, nn.BatchNorm2d):
        init.constant_(m.weight, 1)
        init.constant_(m.bias, 0)
    elif isinstance(m, nn.Linear):
        init.normal_(m.weight, 0, 0.01)
        init.constant_(m.bias, 0)
vgg16.apply(initialize_weights)
```

این قطعه کد یک تابع مقداردهی اولیه وزن را برای یک شبکه عصبی تعریف می کند، که به طور خاص برای مدل VGG-16 طراحی شده است. تابع initialize_weights طوری طراحی شده است که به عنوان یک فراخوان در طول فرآیند تعریف مدل اعمال شود. در این تابع، نوع هر ماژول m بررسی می شود و وزن ها و بایاس ها بر این اساس مقداردهی اولیه می شوند. برای لایه‌های Conv2d، مقداردهی اولیه Kaiming با غیرخطی ReLU استفاده می‌شود و بایاس‌ها را صفر می‌کند. لایه‌های BatchNorm2d دارای وزن اولیه‌شان به ۱ و بایاس‌ها به ۰ هستند که به شیوه‌های معمول Batch Normalization پایبند هستند. لایه های خطی از توزیع نرمال با میانگین ۰ و انحراف استاندارد ۰,۰۱ برای مقدار دهی اولیه وزن استفاده می کنند و بایاس ها به صفر مقداردهی می شوند. در نهایت، تابع apply برای اعمال بازگشتی این استراتژی اولیه سازی برای همه زیرماژول های مدل VGG-16 استفاده می شود، و از یک طرح اولیه سازی سازگار و مناسب برای هر لایه در شبکه اطمینان حاصل می کند. ولی در اینجا استفاده نکردیم. (در صورت شوال ذکر نشده بود.)

در قدم بعدی دیتاست را به شکل زیر لود میکنیم، در نظر داشته باشید که باید از augmentation اشتفاده کنیم که مدل ما overfit نشود. در این قسمت از تکنیک های RandomHorizontalFlip و RandomCrop استفاده کردیم. همچنین در این قسمت باید داده ها را نرمال کنیم. این داده ها را نیز بصورت متوازن به ۳ دسته ترین-تست و valid تقسیم میکنیم(خروجی این بخش را تست کرده و دیدیم که داده ها بصوت متوازن بودند و از این موضوع اطمینان داریم.). در انتهای لود کردن داده ها آن را در فایل pkl. ذخیره میکنیم که در ادامه ی سوال داده های دیتاست یکسان باشند و در هربار ران کردن داده های جدید نداشته باشیم. همچنین ۱۰ درصد از داده ها را نیز جدا میکنیم و در فایل pkl. ذخیره میکنیم برای محاسبه شاخص های SI.

```
# Define transformations
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

# Download the dataset
train_dataset = CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
test_dataset = CIFAR10(root='./data', train=False, download=True, transform=
transform_test)
sss = StratifiedShuffleSplit(n_splits=1, test_size=0.1, random_state=42)
train_indices, val_indices = next(sss.split(np.zeros(len(train_dataset)),
train_dataset.targets))

# Create data loaders
train_sampler = torch.utils.data.sampler.SubsetRandomSampler(train_indices)
val_sampler = torch.utils.data.sampler.SubsetRandomSampler(val_indices)

train_loader = DataLoader(train_dataset, batch_size=128, sampler=train_sampler,
num_workers=4)
val_loader = DataLoader(train_dataset, batch_size=128, sampler=val_sampler,
num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False,
num_workers=4)

from torch.utils.data.dataset import Subset

# Calculate the size of the subset (10% of the original dataset)
subset_size = int(0.1 * len(test_loader.dataset))
subset_indices = torch.randperm(len(test_loader.dataset))[:subset_size]
# Create a subset using Subset class from torch.utils.data.dataset
subset = Subset(test_loader.dataset, subset_indices)
# Create a new DataLoader for the subset
test_loader_SI = DataLoader(subset, batch_size=test_loader.batch_size,
shuffle=True, num_workers=test_loader.num_workers)
```

به شکل زیر داده ها را در فایل **pkl.** ذخیره میکنیم.

```
# Save the DataLoader objects to a pickle file
data_loaders = {'train_loader': train_loader, 'val_loader': val_loader,
'test_loader': test_loader}
```

```
with open('data_loaders_final.pkl', 'wb') as file:
    pickle.dump(data_loaders, file)
# Save the DataLoader objects to a pickle file
data_loaders = {'train_loader_SI': train_loader_SI,'test_loader_SI':
test_loader_SI}

with open('data_loaders_SI.pkl', 'wb') as file:
    pickle.dump(data_loaders, file)
```

برای آموزش مدل ابتدا هایپرپارامترهارا بصورت زیر تعریف میکنیم:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(vgg16.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-
4)
scheduler = StepLR(optimizer, step_size=50, gamma=0.1)
# Training loop
num_epochs = 200
best_val_accuracy = 0.0
```

train loop را به شکل زیر پیاده سازی میکنیم:

```
patience = 20
early_stop_counter = 0
best_val_loss = float('inf')  # Initialize with a large value

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

for epoch in tqdm(range(num_epochs)):
    vgg16.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)  # Move data to GPU
        optimizer.zero_grad()
        outputs = vgg16(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_accuracy = 100 * correct_train / total_train
    print(f'Epoch {epoch + 1}/{num_epochs}, Train Loss: {running_loss /
len(train_loader):.4f}, Train Acc: {train_accuracy:.2f}%')
    # Validation
    vgg16.eval()
    correct_val = 0
    total_val = 0
    val_loss = 0.0
    with torch.no_grad():
        for data in val_loader:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = vgg16(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()
    val_accuracy = 100 * correct_val / total_val
    print(f'Epoch {epoch + 1}/{num_epochs}, Val Loss: {val_loss /
len(val_loader):.4f}, Val Acc: {val_accuracy:.2f}%')
    # Append values for plotting
    train_losses.append(running_loss / len(train_loader))
    val_losses.append(val_loss / len(val_loader))
    train_accuracies.append(train_accuracy)
    val_accuracies.append(val_accuracy)
    # Save the model with the best validation accuracy
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        torch.save(vgg16.state_dict(), 'only_fc_weights.pth')
        early_stop_counter = 0  # Reset the counter when there's an improvement
    else:
        early_stop_counter += 1  # Increment the counter if there's no
improvement
    # Print early stopping information
    print(f'Epoch {epoch + 1}/{num_epochs}, Early Stop Counter:
{early_stop_counter}/{patience}')
    # Check for early stopping
    if early_stop_counter >= patience:
        print(f'Early stopping after {epoch + 1} epochs without improvement in
validation accuracy.')
        break  # Exit the training loop
    # Adjust learning rate
    scheduler.step()
```

این کد نشان دهنده یک حلقه آموزشی برای یک شبکه عصبی با استفاده از معماری VGG16 است. حلقه آموزشی در تعداد معینی از epoch (در این مورد ۲۰۰) تکرار می‌شود و برای هر دسته از داده‌های

آموزشی با استفاده از بهینه‌ساز Adam پاس‌های رو به جلو و عقب را انجام می‌دهد. دقت آموزش در هر دوره چاپ می شود، از جمله از loss آموزش و دقت. پس از هر دوره، مدل بر روی یک مجموعه اعتبار سنجی ارزیابی می شود و از دست دادن اعتبار و دقت نیز چاپ می شود. این کد بهترین دقت اعتبارسنجی را که تاکنون به دست آمده را دنبال می کند و در صورت مشاهده بهبود، وزن مدل را ذخیره می کند. این شامل توقف زودهنگام، نظارت بر صحت اعتبارسنجی در تعداد معینی از دوره‌ها (patienve) و توقف آموزش در صورت عدم بهبود در طول این دوره است. علاوه بر این، یک زمان‌بندی نرخ یادگیری برای تنظیم نرخ یادگیری در طول آموزش نیز استفاده می‌شود.

کد متغیرهایی را برای ردیابی معیارهای آموزشی و اعتبارسنجی، مانند تلفات و دقت، مقداردهی اولیه می‌کند. سپس در طول دوره‌ها تکرار می‌شود، پارامترهای مدل را بر اساس داده‌های آموزشی به‌روزرسانی می‌کند، عملکرد مجموعه اعتبارسنجی را ارزیابی می‌کند و در صورت لزوم توقف اولیه را اعمال می‌کند. این کد برای جلوگیری از برازش بیش از حد و بهبود تعمیم مدل با نظارت و توقف آموزش در زمانی که عملکرد مدل در مجموعه اعتبار سنجی بالا می رود طراحی شده است. علاوه بر این، نرخ یادگیری با استفاده از یک زمان‌بندی تنظیم می‌شود تا به طور بالقوه همگرایی در طول آموزش را بهبود بخشد.

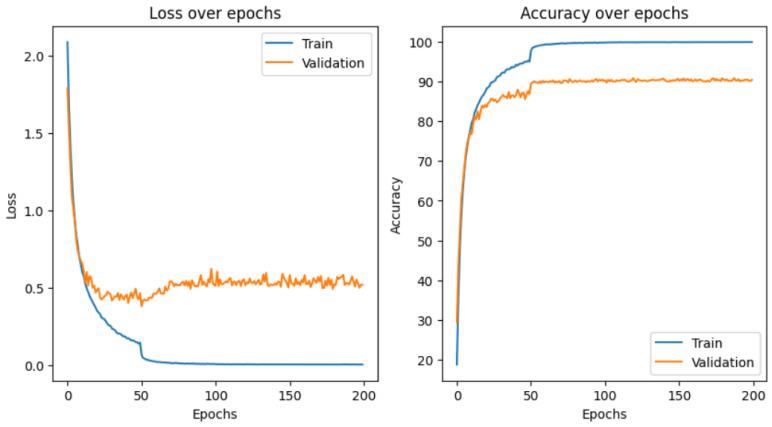
نتایج قسمت الف:

دقت شبکه بر روی داده های تست:

Test Accuracy: 90.68%

مشاهده میکنیم که به دقت خواسته شده در صورت سوال رسیده ایم.

نمودارهای دقت و loss داده های valid و train:



شکل ۲ - نمودارهای loss و accuracy قسمت الف

در اینجا میبینیم که loss مدل به خوبی کاهش پیدا کرده است و به دقت مورد نظر نیز رسیده ایم. آنالیز دقیق تر این بخش در قسمت آخر گزارش آورده شده است.

- محاسبه متریک های خواسته شده:

در تمرین دوم کدها بهینه شده SI توضیح کامل داده شده است که در اینجا از آن ها استفاده می کنیم.

کدها در لینک زیر قرار گرفته اند:

!git clone https://github.com/Arhosseini77/data_complexity_measures

بعد از ران کردن کد بالا بصورت زیر CSI را برای هر لایه و برای داده های ترین و تست محاسبه میکنیم:

```
from data_complexity_measures.models.ARH_SeparationIndex import
ARH_SeparationIndex
```

در ادامه باید به کمک وزن های آموزش داده شده و به کمک کد زیر مدل را لود کرد:

```
# Instantiate and load the model
model = VGG()
model.load_state_dict(torch.load('best_model_weights.pth'))
model.to('cuda:1' if torch.cuda.is_available() else 'cpu')
model.eval()
```

برای آنکه بتوان از هر لایه خروجی گرفت باید hook به هر لایه اضافه شود. که این کار به کمک کد زیر انجام شده است.

```
# Prepare storage for outputs and labels
features_per_layer = [[] for _ in range(len(model.features))]
labels_list = []
# Function to attach hooks
def get_layer_outputs(layer_idx):
    def hook(module, input, output):
        features_per_layer[layer_idx].append(output.detach())
    return hook
# Attach hooks to each layer
for idx, layer in enumerate(model.features):
    layer.register_forward_hook(get_layer_outputs(idx))
```

برای آنکه بتوان از هر لایه خروجی گرفت باید hook به هر لایه اضافه شود. که این کار به کمک کد زیر انجام شده است. هدف این کد استخراج ویژگی‌های میانی از هر لایه یک مدل شبکه عصبی است.

features_per_layer برای ذخیره ویژگی‌های هر لایه شروع می‌شود. تعداد کل لایه ها در مدل با استفاده از طول لیست تعیین می شود. پس از آن، یک حلقه از طریق هر لایه تکرار می شود. یک هوک رو به جلو برای لایه ثبت می شود. این قلاب که به عنوان تابعی به نام «hook» تعریف می‌شود، ویژگی‌های خروجی لایه را به ورودی مربوطه در «features_per_layer» اضافه می‌کند. با اتصال این قلاب‌ها به لایه‌های مورد نظر، کد مجموعه‌ای از ویژگی‌های میانی را در حین محاسبات forward pass امکان‌پذیر می‌سازد، و تحلیل بیشتر یا تجسم نمایش‌های داخلی مدل را تسهیل می‌کند.

```
# Pass data through the model and collect layer outputs
with torch.no_grad():
    for inputs, targets in tqdm(train_loader_SI):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda:1')
        # Trigger the hooks and collect layer outputs
        model(inputs)
        labels_list.append(targets)
        # Release GPU memory
        del inputs
        torch.cuda.empty_cache()
# Post-process the data: Flatten and concatenate
for idx, layer_features in enumerate(features_per_layer):
    layer_features = torch.cat([f.view(f.size(0), -1) for f in layer_features])
    features_per_layer[idx] = layer_features
labels = torch.cat(labels_list)
```

حال میتوان دیتالودر های آموزش و ارزیابی و تست را به مدل داد و خروجی را گرفت و خروجی فیچر های هر لایه را به موارد مورد نیاز برای ورودی دادن به کلاس CSI درآورد.

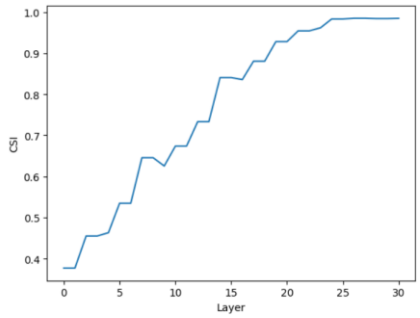
این کد برای استخراج ویژگی های میانی از هر لایه از مدل شبکه عصبی در حین پردازش مجموعه داده های آموزشی طراحی شده است. مدل با استفاده از model.eval() روی حالت ارزیابی تنظیم می شود. یک حلقه از طریق مجموعه داده آموزشی/تست با استفاده از یک بارگذار داده تکرار می شود. برای هر دسته، ورودی ها به دستگاه مشخص شده منتقل می شوند و مدل ورودی ها را پردازش می کند. اهداف در لیست "برچسب ها" در CPU ذخیره می شوند. پس از حلقه، برچسب های ذخیره شده به هم متصل می شوند تا یک تنسور را تشکیل دهند. متعاقباً، ویژگی‌های میانی جمع‌آوری‌شده در طول گذر رو به جلو برای هر لایه، مسطح شده و در امتداد بعد دوم به هم متصل می‌شوند و یک تانسور دو بعدی برای هر لایه تشکیل می‌دهند. features_per_layer در آن هر کلید با نام لایه مطابقت دارد و مقدار مرتبط تنسوری است که حاوی ویژگی های مسطح از آن لایه در کل مجموعه داده است. این کد استخراج و سازماندهی ویژگی های میانی را از لایه های مختلف مدل برای تجزیه و تحلیل یا تجسم بیشتر تسهیل می کند.

```
csi_layer_train = []
# Iterate through each layer's features in the dictionary
for features in features_per_layer:
    instance_disturbance = ARH_SeparationIndex(features, labels, normalize=True)
    csi = instance_disturbance.center_si_batch(batch_size=2000)
```

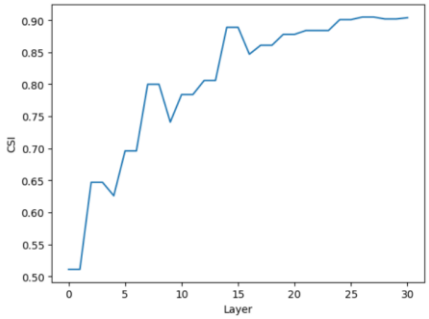
```
    csi_layer_train.append(csi)
```

کد بالا را یکبار برای داده تست و یکبار برای داده ترین ران می کنیم و نتایج را ذخیره میکنیم. در بقیه سوال هم از همین کد استفاده میکنیم.

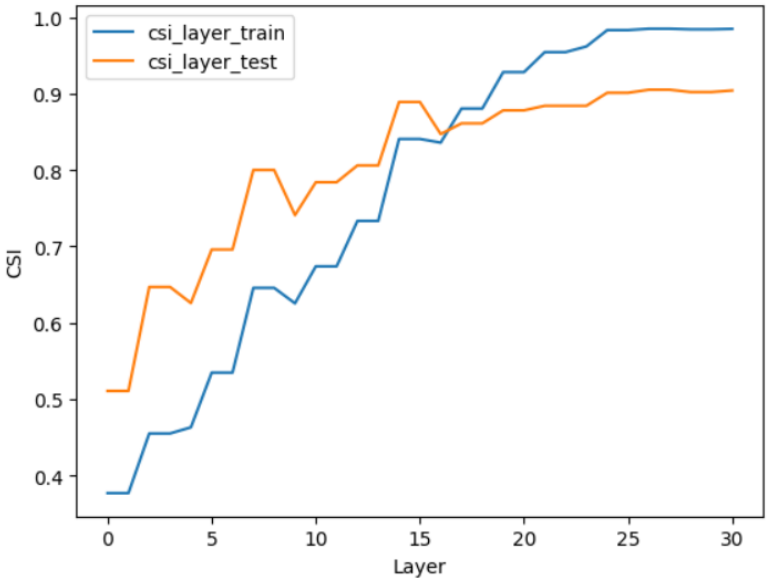
متریک ها روی داده های ترین و تست:



شکل ۳ - CSI روی داده های train



شکل ٤ - CSI روی داده های test



شکل ٥ - CSI روی داده های test و train

از روی نمودارهای CSI متوجه می شویم که منحنی ها روند صعودی دارند و در هر لایه مرکز هر دسته راحت تر پیدا شده است، که می تواند حاکی از بهبود عملکرد جداسازی مدل با هر لایه بعدی باشد. ، چون مقدار متریک ما افزایش پیدا کرده است. همچنین CSI روی داده های ترین بیشتر شده است، چون دقت مدل روی داده های ترین بیشتر از تست است و مرکز دسته ها در داده ی ترین بهتر مشخص شده است.

ب)

در این قسمت لایه های قبل طبقه بند را freeze کرده و دوباره فقط طبقه بند را آموزش می دهیم. برای این قسمت، از وزن هایی که در قسمت الف ذخیره کرده بودیم (بهترین وزن) استفاده میکنیم.

```
# Instantiate the VGG model
vgg16 = VGG()
# Move the model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vgg16.to(device)
# Load pre-trained weights (assuming you have a pre-trained model file)
pretrained_weights_path = "best_model_weights.pth"
state_dict = torch.load(pretrained_weights_path)

# Remove the classifier weights from the loaded state_dict
state_dict = {k: v for k, v in state_dict.items() if 'classifier' not in k}

# Load the modified state_dict into the model
vgg16.load_state_dict(state_dict, strict=False)

# Freeze layers before the classifier
for param in vgg16.features.parameters():
    param.requires_grad = False

# Print the model architecture
print(vgg16)
```

این کد پایتون شامل نمونه سازی و اصلاح یک مدل VGG برای یادگیری انتقالی است. در ابتدا، نمونه ای از مدل VGG ایجاد می شود. سپس مدل در صورت موجود بودن به GPU منتقل می شود. پس از آن، وزن‌های از پیش آموزش‌دیده‌شده از فایلی بارگذاری می‌شوند. برای تطبیق مدل از پیش آموزش‌دیده‌شده برای یک کار متفاوت، کد وزن‌های مرتبط با لایه‌های طبقه‌بندی‌کننده را از dictionary حالت بارگذاری شده حذف می‌کند. این مرحله امکان حفظ قابلیت‌های استخراج ویژگی‌های آموخته‌شده در طول پیش‌آموزش را در حین سفارشی‌سازی لایه‌های طبقه‌بندی نهایی برای یک کار خاص فراهم می‌کند. در نهایت، اسکریپت پارامترهای لایه‌های استخراج ویژگی (به استثنای لایه‌های طبقه‌بندی‌کننده) را مسدود

می‌کند تا از به‌روزرسانی آن‌ها در طول آموزش جلوگیری کند و مدل را قادر می‌سازد تا بر یادگیری ویژگی‌های خاص کار تمرکز کند. با استفاده از **train loop**ای که در مرحله قبل استفاده کردیم، طبقه بند را آموزش می دهیم.

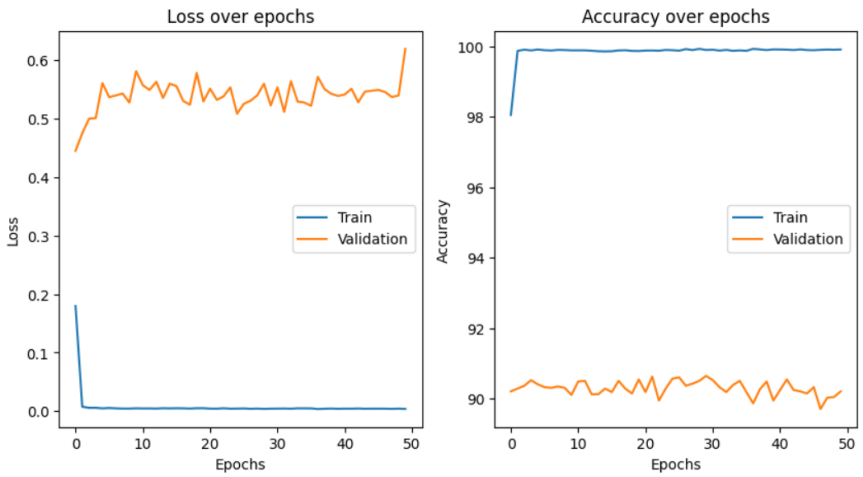
نتایج قسمت ب:

دقت شبکه بر روی داده های تست:

Test Accuracy: 90.56%

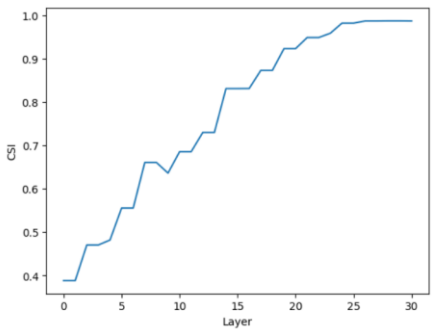
مشاهده میکنیم که به دقت خواسته شده در صورت سوال رسیده ایم. (دقت نزدیک نتیجه قسمت الف شده است.)

نمودارهای دقت و **loss** داده های **valid** و **train**:

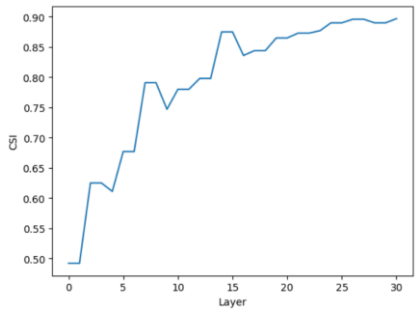


شکل ٦ - نمودارهای loss و accuracy قسمت الف

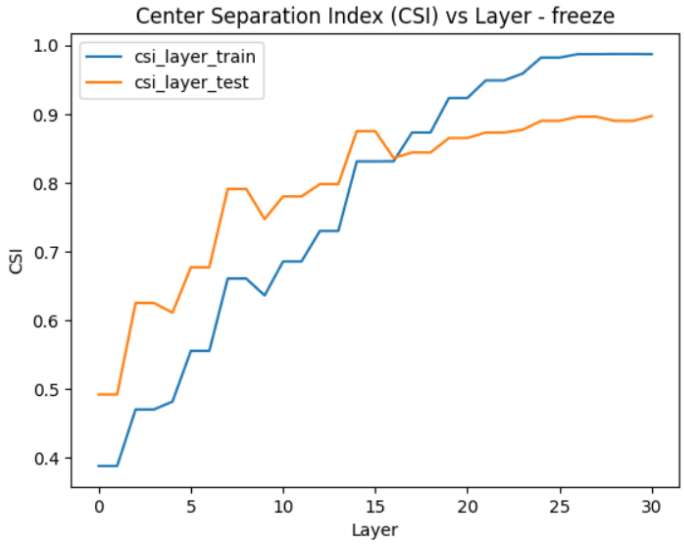
متریک ها روی داده های ترین و تست:



شکل ۷- CSI روی داده های train



شکل ۸ - CSI روی داده های test



شکل ۹ - CSI روی داده های test و train

در این قسمت هم همانند قسمت الف منحنی ها روند صعودی دارند و در هر لایه مرکز هر دسته راحت تر پیدا شده است، که می تواند حاکی از بهبود عملکرد جداسازی مدل با هر لایه بعدی باشد، چون مقدار متریک ما افزایش پیدا کرده است. همچنین CSI روی داده های ترین بیشتر شده است، چون دقت مدل روی داده های ترین بیشتر از تست است و مرکز دسته ها در داده ی ترین بهتر مشخص شده است.

(بقیه تحلیل ها در بخش آخر گزارش)

در این قسمت مدل ثابت feature extractor داریم که فشرده سازی نشده است و برای آن طبقه بند را آموزش می دهیم و نتایج بالا بدست آمده است، در مرحله بعد، مدل را فشرده میکنیم و طبقه بند را آموزش می دهیم.

ج) فشرده سازی مدل آموزش داده شده

ج-۱ : لود کردن مدل و دیتالودر ها

جهت محاسبه SI ابتدا pkl هایی از دیتالودر که ذخیره کرده بودیم را لود میکنیم.

```
def load_loaders(file_path):
    with open(file_path, 'rb') as file:
        loaders = pickle.load(file)
    return loaders['train_loader'], loaders['val_loader'], loaders['test_loader']
train_loader, valid_loader, test_loader = load_loaders('data_loaders_final.pkl')
```

در ادامه بر اساس خواسته مساله ۱۰ درصد از داده های آموزش را جهت محاسبه SI جدامیکنیم.

- برای محاسبه SI از کد هایی که در تمرین دوم زده شد و تمامی بهینه سازی های ممکن برای آن انجام شده و در ریپازیتوری زیر قابل دسترسی میباشد ، استفاده شده است. (توضیح کامل پیاده سازی ها و نحوه بهینه سازی در گزارش تمرین دوم آمده است.)

https://github.com/Arhosseini77/data_complexity_measures

کد زیر ۱۰ درصد از داده ها را جدا میکند.

```
# Calculate the size of the subset (10% of the original dataset)
subset_size = int(0.1 * len(train_loader.dataset))
subset_indices = torch.randperm(len(train_loader.dataset))[:subset_size]
# Create a subset using Subset class from torch.utils.data.dataset
subset = Subset(train_loader.dataset, subset_indices)
# Create a new DataLoader for the subset
train_loader_SI = DataLoader(subset, batch_size=train_loader.batch_size,
shuffle=True, num_workers=train_loader.num_workers)
```

در ادامه به کمک کد زیر مدل VGG را لود کرده و وزن ها مدل آموزش داده شده در بخش الف را وارد میکنیم.

```
# Instantiate and load the model
model = VGG()
model.load_state_dict(torch.load('best_model_weights.pth'))
model.to('cuda:1' if torch.cuda.is_available() else 'cpu')
model.eval()
```

مدل VGG به صورت زیر میباشد که درای ۳ بخش فیچر و avg pool و طبقه بند میباشد.

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
```

```
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

ج-۲ : محاسبه SI و CSI بر روی مدل pretrained

برای آنکه بتوانیم از هر لایه در بخش فیچر شبکه VGG16 خروجی بگیریم نیاز است تا یک hook به هر لایه اضافه کنیم و فضا هایی نیز برای ذخیره سازی لیبل و فیچر های هر لایه اضافه کنیم به کمک کد های زیر مراحل فوق انجام شده و در نهایت تمامی فیچر های هر لایه در لیستی ذخیره میشود و تمامی پیش پردازش های لازم جهت دادن آن ها به کلاس SI انجام میشود.

```
# Prepare storage for outputs and labels
features_per_layer = [[] for _ in range(len(model.features))]
labels_list = []

# Function to attach hooks
def get_layer_outputs(layer_idx):
    def hook(module, input, output):
        features_per_layer[layer_idx].append(output.detach())
    return hook

# Attach hooks to each layer
for idx, layer in enumerate(model.features):
    layer.register_forward_hook(get_layer_outputs(idx))
```

```
# Pass data through the model and collect layer outputs
with torch.no_grad():
    for inputs, targets in tqdm(train_loader_SI):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda:1')

        # Trigger the hooks and collect layer outputs
        model(inputs)
        labels_list.append(targets)

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

```
# Post-process the data: Flatten and concatenate
for idx, layer_features in enumerate(features_per_layer):
    layer_features = torch.cat([f.view(f.size(0), -1) for f in layer_features])
    features_per_layer[idx] = layer_features

labels = torch.cat(labels_list)
```

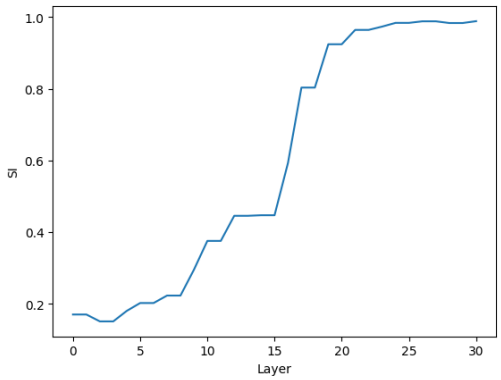
در نهایت نیز به راحتی SI را به کمک کد زیر برای هر لایه محاسبه میکنیم (۰ تا ۳۰ در بخش model.features)

```
si_layer_train =[]
for features in features_per_layer:
  instance_disturbance = ARH_SeparationIndex(features, labels, normalize=True)
  si = instance_disturbance.si()
  si_layer_train.append(si)
```

مقادیر SI به ترتیب به شرح زیر میباشد.

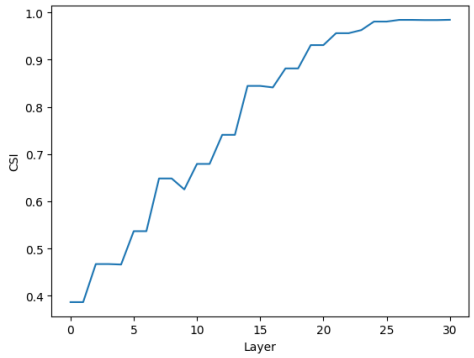
```
[0.1704, 0.1704, 0.1512, 0.1512, 0.1804, 0.2024, 0.2024, 0.2232, 0.2232, 0.2952,
0.3756, 0.3756, 0.4456, 0.4456, 0.4472, 0.4472, 0.5932, 0.8032, 0.8032, 0.924,
0.924, 0.964, 0.964, 0.9732, 0.9836, 0.9836, 0.988, 0.988, 0.9832, 0.9832,
0.9884]
```

شکل زیر منحنی SI به ازای هر لایه را نشان میدهد:

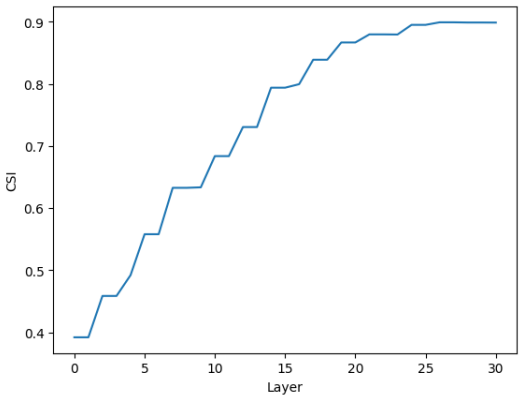


شکل ۱۰ : منحنی SI به ازای هر لایه vgg.features برای داده های ترین

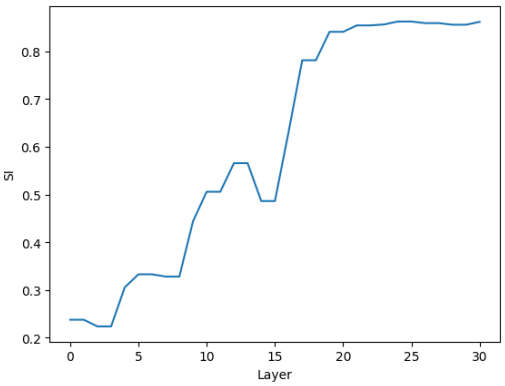
به همین ترتیب به کمک center_SI منحنی CSI را نیز رسم میکنیم.



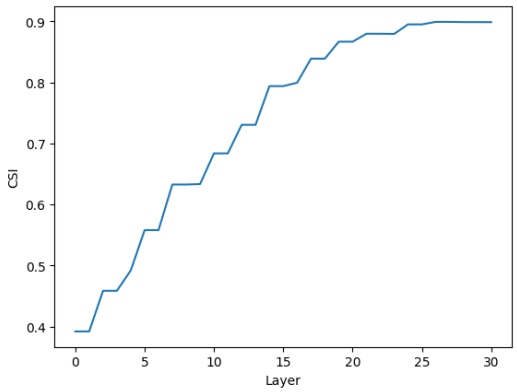
شکل ۱۱ : منحنی SI به ازای هر لایه vgg.features برای داده های ترین



شکل های زیر نیز منحنی های SI و CSI برای داده های تست نمایش میدهد:



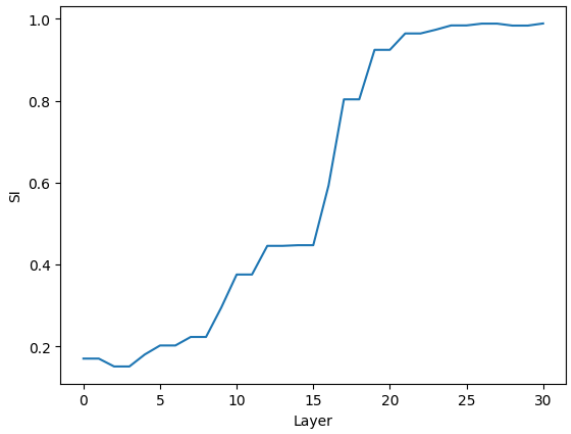
شکل ۱۲ : منحنی SI به ازای هر لایه vgg.features برای داده های تست



شکل ۱۳ : منحنی CSI به ازای هر لایه vgg.features برای داده های تست

ج-۳ : عملیات فشرده سازی : کم کردن تعداد لایه ها

مجددا نگاهی به منحنی SI برای داده های ترین میاندازیم :



شکل ۱۴ : منحنی SI به ازای هر لایه vgg.features برای داده های ترین

همانطور که انتظار میرفت منحنی روندی کاملا صعودی دارد اما با توجه به اعداد SI مطرح شده بعد از جهش از ۹۲ به ۹۶ در لایه ۲۱ تقریبا صعود محسوسی نداشته و با توجه به الزام وجود pooling و relu در لایه آخر شبکه را تا لایه ۲۳ ام فشرده میکنیم.

برای این کار بدین صورت عمل میکنیم.

```
# Redefine the features module up to the (23)
model.features = torch.nn.Sequential(*list(model.features.children())[:24])
```

بدین ترتیب model.features به این صورت میشود:

```
VGG(
 (features): Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
 )
```

ج-۴ : عملیات فشرده سازی : Forward Feature Selection

حال بخش feature selection را انجام میدهیم.

در این بخش نیاز است خروجی اخرین لایه را بگیریم (لایه ۲۳ در model.features)

به کمک کد زیر خروجی را گرفته و پیش‌پردازش لازم را انجام میدهیم.

```
features = []
labels = []

with torch.no_grad():
    for inputs, targets  in tqdm(train_loader_SI):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda:1')
        # 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)  # Make it a 2D tensor
```

تعداد داده ها ۵۰۰۰ تا و تعداد فیچر ها ۲۰۴۸ تا میباشد.

حال که فیچر و لیبل ها را داریم الگوریتم **feature selection** را انجام میدهیم.

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

si_ranked_features, ranked_features =
instance_disturbance.forward_feature_ranking_si()
```

کد پیاده سازی شده و بهینه الگوریتم **forward feature selection** :

```
def forward_feature_ranking_si(self):
  ranked_features = torch.tensor([], dtype=torch.long, device=self.device)
  rest_features = torch.arange(self.n_feature, device=self.device)
  si_ranked_features = torch.zeros(self.n_feature, device=self.device)

  # Precompute the full distance matrix for all features
  full_dis_matrix = torch.cdist(self.data, self.data, p=2)
  full_dis_matrix.fill_diagonal_(self.big_number)

  print("Start forward selection")

  for k_forward in tqdm(range(self.n_feature)):
      si_max = 0
      chosen_feature = None

      for k_search in range(len(rest_features)):
          current_feature = rest_features[k_search]
          if k_forward == 0:
              # Use the precomputed distance matrix for the first feature
              dis_matrix = full_dis_matrix[:, current_feature].unsqueeze(1)
          else:
              # Use a subset of the precomputed distance matrix for subsequent
features
```

```
                features_to_use = torch.cat((ranked_features,
current_feature.unsqueeze(0)))
                dis_matrix = full_dis_matrix[:, features_to_use]

            # Compute minimum distances and their indices
            values, indices = torch.min(dis_matrix, dim=1)
            si = (self.label[indices] == self.label).float().mean()

            if si > si_max:
                si_max = si
                chosen_feature = current_feature

        ranked_features = torch.cat((ranked_features, chosen_feature.unsqueeze(0)))
        rest_features = rest_features[rest_features != chosen_feature]
        si_ranked_features[k_forward] = si_max

    return si_ranked_features, ranked_features
```

در کد فوق :

تابع "forward_feature_ranking_si" : هدف یافتن مکرر بهترین مجموعه از ویژگی هایی است که شاخص جداسازی را به حداکثر می رساند. شاخص جدایی (SI) معیاری است که تعیین می کند که نقاط داده با برچسب یکسان چقدر در کنار هم قرار گرفته اند.

Initial :

- «ویژگی‌های رتبه‌شده»: تانسوری برای ذخیره شاخص‌های ویژگی‌های انتخاب‌شده در هر تکرار. در ابتدا خالی است.

- `rest_features`: یک تانسور حاوی شاخص های همه ویژگی ها. در ابتدا، تمام ویژگی ها را شامل می شود.

- `si_ranked_features`: تانسوری برای ذخیره Separation Index برای مجموعه ویژگی های انتخاب شده در هر تکرار. در ابتدا حاوی صفر است.

محاسبات ماتریس فاصله:

- یک ماتریس فاصله کامل برای همه ویژگی ها یک بار با استفاده از "torch.cdist" محاسبه می شود. این ماتریس نشان دهنده فواصل زوجی بین تمام نقاط داده برای هر ویژگی است.

- مورب این ماتریس برای جلوگیری از فاصله های صفر (فاصله یک نقطه از خودش) با عدد زیادی پر می شود.

حلقه انتخاب ویژگی رو به جلو:

- عملکرد روی همه ویژگی ها تکرار می شود تا بهترین ویژگی در هر تکرار انتخاب شود.

- در هر تکرار ('k_forward'):

- حلقه داخلی روی «ویژگی‌های_بقیه» تکرار می‌شود، که ویژگی‌هایی هستند که هنوز انتخاب نشده‌اند.

- برای هر ویژگی ('current_feature') در 'rest_features':

- اگر اولین تکرار است، از ستون مربوط به ویژگی فعلی از ماتریس فاصله کامل از پیش محاسبه شده استفاده میشود.

- حداقل فاصله ها و شاخص های آنها را برای مجموعه ویژگی های انتخاب شده محاسبه میشود.

- شاخص جدایی (SI) را برای مجموعه فعلی ویژگی ها محاسبه میشود.

- پس از ارزیابی همه ویژگی‌ها در 'rest_features'، ویژگی‌ای انتخاب میشود که منجر به بالاترین SI شده است.

- «ویژگی‌های rank شده به‌روزرسانی میشود تا ویژگی انتخاب‌شده را در بر بگیرد و این ویژگی از 'rest_features حذف میشود .

- حداکثر مقدار SI در `si_ranked_features` ذخیره میشود.

خروجی:

- تابع دو تانسور را برمی گرداند:

- `si_ranked_features`: تانسوری حاوی مقادیر Separation Index برای هر تکرار، که نشان می‌دهد چگونه SI با افزودن ویژگی‌ها بهبود می‌یابد.

- فیچر های رنک شده : تانسوری حاوی شاخص‌های ویژگی‌های انتخاب‌شده به ترتیب انتخاب آنها.

این تابع تقریبا شبیه همان کد دکتر کلهر میباشد اما یک سری بهینه سازی ها در آن انجام شده و محاسبات در numpy دیگر انجام نمیشود و تمامی محاسبات بر روی GPU میرود همچنین :

این تابع از نظر محاسباتی فشرده است زیرا شامل محاسبه فاصله برای زیرمجموعه های ویژگی ها در تمام نقاط داده است. با این حال، با پیش محاسبه یک ماتریس فاصله کامل و استفاده مجدد از بخش های مربوطه در هر تکرار بهینه شده است.

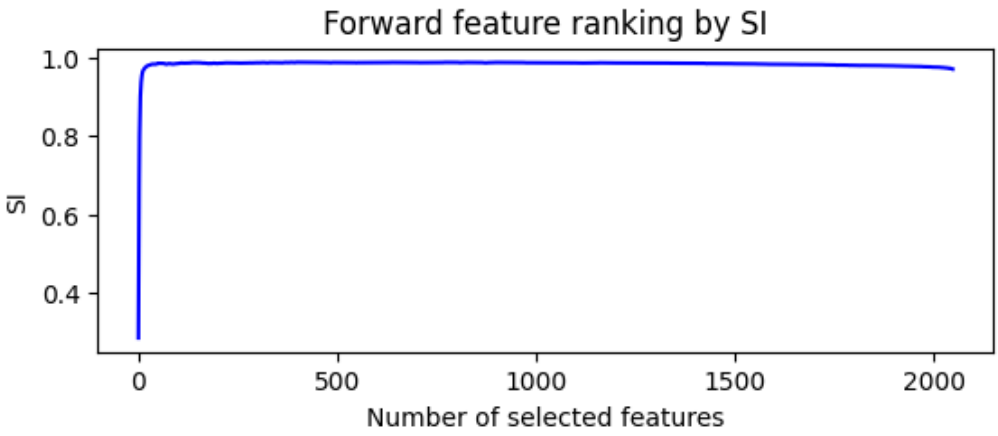
در کد دکتر کلهر مدت ران تایم این کد بر روی 3090 با رم ۲۴ گیگ وCPU Core i9 64G مدت زمان ۱ ساعت و ۴۵ دقیقه طول میکشید اما با بهینه سازی ها فوق این محاسبات تنها ۴ دقیقه طول میشکند. در نهایت نیز rank_feature و si_rank_feature را در فایل های pkl ذخیره میکنیم.

```
# Saving the variables to .pkl files
with open('si_ranked_features_new.pkl', 'wb') as f:
    pickle.dump(si_ranked_features, f)

with open('ranked_features_new.pkl', 'wb') as f:
    pickle.dump(ranked_features, f)
```

جهت لود مجدد آن ها از دستور زیر استفاده میشود.

```
# Loading the data from the .pkl files
with open('si_ranked_features_new.pkl', 'rb') as f:
    si_ranked_features = pickle.load(f)

with open('ranked_features_new.pkl', 'rb') as f:
    ranked_features = pickle.load(f)
```

منحنی زیر فرایند feature selection از ۲۰۴۸ فیچر نهایی را نشان میدهد. (مقدار SI بر حسب تعداد فیچر)



شکل ۱۵ : منحنی SI بر حسب فیچر های خروجی آخرین لایه پس از فشرده سازی

همانطور که در منحنی بالا مشاهده می شود با تعداد فیچر های بسیار کمتری نسبت به ۲۰۴۸ تا میشود SI بالایی در فضای فیچر ها داشته که در اینجا با ۴۰۰ فیچر مقدار SI به ۰/۹۸۹۳ میرسد در حالیکه با ۲۰۴۸ تا فیچر مقدار SI برابر با ۰/۹۷۳۲ بود.

با مشخص شدن لیست ۴۰۰ تایی از فیچر هایی که باید از ۲۰۴۸ 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]
```

کد فوق نحوه پیدا کردن max و تعداد و اندیس فیچر ها را نشان میدهد که در نهایت در feat ذخیره میشوند.

ج-۵ : عملیات فشرده سازی : پیاده سازی MLP و آموزش آن

حال مجددا یک کلاس mlp را مینویسیم ابتدا دقیقا همان mlp قبلی را قرار میدهیم که دارای لایه های میانی ۴۰۹۶ تایی بود.

```
class MLPModel(nn.Module):
    def __init__(self, input_size, output_size, dropout_rate=0.2):
        super(MLPModel, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, 4096)
        self.relu1 = nn.ReLU()
```

```
        self.dropout1 = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(4096, 4096)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout_rate)
        self.fc3 = nn.Linear(4096, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

به کمک کد زیر MLP را تشکیل داده و موارد لازم را initial میکنیم.

```
input_size = len(feat)
output_size = 10
mlp_model = MLPModel(input_size, output_size)

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

# Set device
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")

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

فرایند بدین صورت است که از model.features که فشرده شده فیچر میگیریم با feat روی آن slice میکنیم و آن را به MLP میدهیم و فقط پارامتر های این MLP را ترین میکنیم.

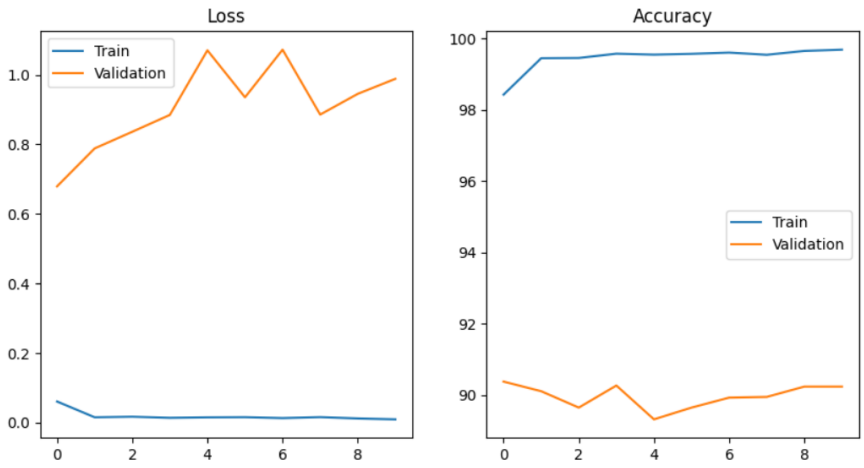
کد ترین زیر به همین ترتیب نوشته شده است:

```
# Training and validation loop
num_epochs = 6

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 = features_batch.view(-1, 2048, 1, 1)
            #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 valid_loader:
            val_inputs, val_labels = val_inputs.to(device), val_labels.to(device)

            val_features_batch = model.features(val_inputs)
            val_features_batch = val_features_batch.view(-1, 2048, 1, 1)
            #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(valid_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 loss و دقت در این حالت به صورت زیر میشود.

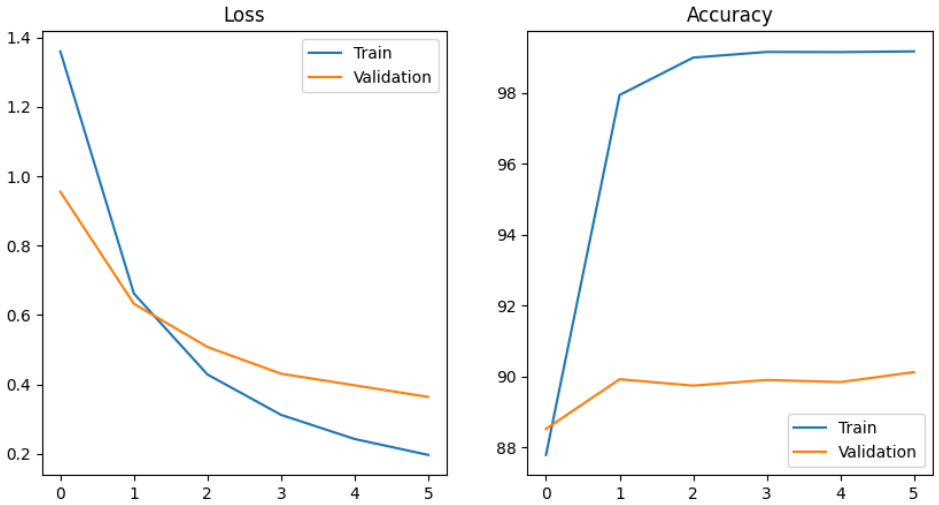


شکل ۱۶: منحنی loss و دقت پس از فشرده سازی مدل و استفاده از MLP با ۲ لایه مخفی ۴۰۹۶ تایی

اما شاید با توجه به انتخاب 400 فیچر وجود لایه میانی با 4096 نورون خیلی توجیه نداشته باشد و الکی تعداد پارامتر را بیشتر کرده باشیم به همین دلیل از MLP زیر نیز استفاده 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
```

پس از آموزش منحنی loss و دقت در این مدل به شرح زیر میباشد. در اینجا بر خلاف حالت قبل loss در val یا ترین زیاد نمیشود و regularization به خصوص بر روی داده تست را تضمین میکند.



شکل ۱۷ : منحنی loss و دقت بر روی مدل فشرده شده بر روی MLP بدون لایه مخفی

ج-۶ : محاسبه دقت مدل فشرده شده بر روی داده های تست

برای محاسبه دقت بر روی تست نیز از کد زیر استفاده میشود.

```
## test
test_total = 0
test_correct = 0

with torch.no_grad():
    for test_inputs, test_labels in test_loader:
        test_inputs, test_labels = test_inputs.to(device), test_labels.to(device)
```

```
        test_features_batch = model.features(test_inputs)
        test_features_batch = test_features_batch.view(-1, 2048, 1, 1)
        test_x = test_features_batch[:, feat.long(), :, :]

        test_outputs = mlp_model(test_x)

        _, test_predicted = test_outputs.max(1)
        test_total += test_labels.size(0)
        test_correct += test_predicted.eq(test_labels).sum().item()

# Calculate validation accuracy and loss
test_accuracy = 100 * test_correct / test_total
print("test_accuracy : " , test_accuracy )
```

در این حالت (MLP بدون لایه مخفی) دقت بر روی داده های تست برابر با ۹۰,۲۴ میباشد.

ج-۷ : محاسبه تعداد پارامتر های مدل فشرده شده

جهت محاسبه تعداد پارامتر ها نیز به راحتی میتوان با توجه به استفاده از model.features و mlp_model از کد زیر استفاده کرد.

```
total_parameters = sum(p.numel() for p in model.features.parameters()) +
sum(p.numel() for p in mlp_model.parameters())

print("Total number of parameters:", total_parameters)
```

ج-۸ : محاسبه SI و CSI لایه اخر پس از فشرده سازی و feature selection

نکته بعدی محاسبه مقادیر SI و CSI بر روی داده های تست و ترین پس از فشرده سازی میباشد همانطور که میدانید تا لایه جدا شده دقیقا مدل همان مدل قبلی میباشد و اعداد SI و CSI تفاوتی نمیکند و تنها تفاوت مدل فشرده شده این است که جدا از اینکه از لایه ۲۳ جدا شده تنها ۴۰۰ فیچر نهایی آن وارد طبقه بند میشود و بدین ترتیب مقادیر SI و CSI بر روی داده های تست و آموزش با همین ۴۰۰ فیچر باید محاسبه شود.

برای انجام محاسبات آن ها از مدل فشرده و انجام پیش پردازش های لازم جهت دادن آن به کلاس ARH_SeparationIndex به صورتزیر عمل میکنیم.

```
# CSI and SI of last layer after Compression
```

```
features = []
labels = []

with torch.no_grad():
    for inputs, targets  in tqdm(train_loader_SI):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda:1')
        # Forward pass through the model.features
        features_batch = model.features(inputs)
        features_batch = features_batch.view(-1, 2048, 1, 1)
        features_batch = features_batch[:, feat.long(), :, :]

        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)  # Make it a 2D tensor

instance_disturbance = ARH_SeparationIndex(features, labels, normalize=True)

si_last_compress = instance_disturbance.si()
csi_last_compress = instance_disturbance.center_si()

print("Tran_si_last_compress: ", si_last_compress)
print("Train_csi_last_compress: ", csi_last_compress)
```

به همین ترتیب فوق نیز برای داده های تست انجام میشود.

تمامی نتایج حالت های مختلف در بخش د که انالیز و تحلیل میباشد آمده است.

د) آنالیز و تحلیل

جدول زیر نتایج آزمایش های حالت های مختلف را نشان میدهد.

جدول ۱ : مقایسه حالت های مختلف و فشرده سازی مدل از نظر دقت و تعداد پارامتر

ویژگی / مدل	Train Accuracy	Val Accuracy	Test Accuracy	Number of parameters
VGG16	99.95	90.44	90.68	134301514
VGG16 (features : freeze , MLP-2 hidden layer Train)	99.91	90.2	90.56	134301514
Compress VGG16 with MLP-2 hidden Layer	99.59	90.12	89.96	26100042
Compress VGG16 – With MLP-1 layer	99.01	90.42	90.2	7639274

جدول ۲ : مقایسه حالت های مختلف و فشرده سازی مدل از نظر SI و CSI

ویژگی / مدل	SI - Train	CSI – Train	SI - Test	CSI - Test
VGG16	۰,۹۸۵۶	۰,۹۸۶	۰,۸۶۱۶	۰,۸۹۸۵
VGG16 (features : freeze , MLP-2 hidden layer Train)	۰,۹۸۵۶	۰,۹۸۶	۰,۸۶۱۶	۰,۸۹۷
Compress VGG16 with MLP-2 hidden Layer	۰,۹۶۸۲	۰,۹۰۳۷	۰,۸۵۴۱	۰,۸۳۰۷
Compress VGG16 – With MLP-1 layer	۰,۹۷۴۲	۰,۹۰۵۳۹	۰,۸۵۴۱	۰,۸۳۰۷

در ابتدا، مدل VGG16 با ۱۳۴ میلیون پارامتر، دقت ۹۰,۶۸٪ و ۹۰,۵۶٪ را در دو رویکرد آموزشی مختلف به دست آورد. در رویکرد اول، کل مدل آموزش داده شد، در حالی که در روش دوم، تنها جزء طبقه‌بندی کننده آموزش داده شد.

با حرکت به سمت فشرده سازی مدل، یک استراتژی قابل توجه به کار گرفته شده استفاده از شاخص SI با حفظ همان مدل طبقه بندی کننده از مدل اولیه و اعمال SI، پارامترهای مدل به طور قابل توجهی به ۲۶ میلیون کاهش یافت. این مدل فشرده به دقت تست ۸۹,۹۶ درصد دست یافت که به طرز چشمگیری نزدیک به عملکرد مدل اصلی است. فشرده‌سازی بیشتر با ساده‌سازی MLP در طبقه‌بندی‌کننده به یک لایه مورد بررسی قرار گرفت (یک لایه فیچر های ۴۰۰ تایی انتخاب شده به ۱۰ نورون نهایی کلاس) ، که به شدت پارامترهای مدل را به تنها ۷ میلیون کاهش داد. این فشرده سازی شدید به طور شگفت انگیزی

منجر به دقت تست ۹۰,۲٪ شد که نشان می دهد یک مدل سبکتر هنوز هم می تواند عملکرد قابل مقایسه ای ارائه دهد.

این آزمایش همچنین عملکرد مدل را در مجموعه داده‌های مختلف و پیکربندی‌های آموزشی مورد بررسی قرار داد، همانطور که در جداول ارائه شده نشان داده شده است. به عنوان مثال، VGG16 اصلی و تنوع آن با ویژگی‌های ثابت و یک طبقه‌بندی‌کننده آموزش‌دیده با دو لایه پنهان MLP، دقت بالایی را در مجموعه داده‌های آزمایش، اعتبارسنجی و آموزشی حفظ کرد. با این حال، افت جزئی در عملکرد در نسخه های فشرده مدل مشاهده شد. VGG16 فشرده با دو لایه MLP دقت تست ۸۹,۹۶٪ را نشان داد، در حالی که همتای MLP یک لایه آن با دقت آزمایش ۹۰,۲٪ کمی بهتر از آن بود.

اثر فشرده‌سازی در معیارهای SI و CSI ، به‌ویژه در مجموعه داده‌های تست و آموزش مشهود بود. مدل اصلی VGG16 هم در CSI و هم در SI در آموزش و آزمایش مقدار بالایی کسب کرد. با این حال، در مدل‌های فشرده، به‌ویژه مدلی با MLP تک لایه، کاهش قابل‌توجهی در این معیارها وجود داشت که نشان‌دهنده یک مبادله بین پیچیدگی مدل و حفظ عملکرد است. جالب توجه است، به نظر می رسد که آموزش مجدد طبقه بندی کننده پس از فشرده سازی تا حدودی این اثر را کاهش می دهد، همانطور که با دقت نسبتاً پایدار و تغییرات جزئی در مقادیر SI مشهود است.