

# به نام خدا



دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر



درس تحلیل و طراحی شبکه های عصبی عمیق

تمرین شماره ۳

نام و نام خانوادگی : علیرضا حسینی - کیانا هوشانفر

شماره دانشجویی : ۸۱۰۱۰۱۳۶۱ - ۸۱۰۱۰۱۱۴۲

۱۴۰۲ ماه دی

## فهرست

۳	.....	مقدمه
۴	.....	سوال (۱) یادگیری لایه به لایه
۵۳	.....	(ب) شبکه بخش بندی تصاویر دو بعدی
۵۴	.....	ساز و کار معماری UNET
۵۵	.....	ساز و کار معماری PSPNet
۵۷	.....	پیش پردازش و آماده سازی دیتاست
۶۸	.....	کد حلقه آموزش
۷۲	.....	پیاده سازی و آموزش UNET
۷۸	.....	پیاده سازی PSPNet
۸۱	.....	مقایسه و تحلیل نتایج

## مقدمه

هدف از انجام این تمرین آشنایی با یادگیری لایه به لایه و بخش بندی تصاویر است. در سوال یک با یادگیری لایه به لایه آشنا میشویم و پارامترهای موثر در یاد گیری لایه ای را بررسی می کنیم. در سوال دوم با بخش بندی تصاویر و معما ری های مختلف موجود در این حوزه آشنا می شویم.

## سوال ۱) یادگیری لایه به لایه

کدهایی که در تمام بخش های سوال از آن استفاده شده است:

### • dataset loader

برای اینکه در تمامی بخش های داده های دیتاست یکسان باشند و در هر بار ران کردن دیتاهای تغییری در آن ها ایجاد نشود، به شکل زیر ابتدا داده ها را بصورت متوازن به ۳ دسته جدا می کنیم و بعد آن ها را بصورت فایل `.pkl` ذخیره می کنیم. (همچنین برای محاسبه  $SI$  درصد از داده ها را نیز ذخیره میکنیم)

```
import torch
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data.sampler import SubsetRandomSampler
import numpy as np
import pickle

def save_loaders(train_loader, valid_loader, train_loader_SI, test_loader,
file_path):
    loaders = {
        'train_loader': train_loader,
        'valid_loader': valid_loader,
        'train_loader_SI': train_loader_SI,
        'test_loader': test_loader
    }
    with open(file_path, 'wb') as file:
        pickle.dump(loaders, file)

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

def data_loader(batch_size, num_workers=0, random_seed=42, valid_size=0.1,
shuffle=True, test=False, save_si=False):

    # Define augmentations
    transform_augmented = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
```

```

        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.4914, 0.4822, 0.4465],
            std=[0.2470, 0.2435, 0.2616],
        )
    ))
])

transform_augmented_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.4914, 0.4822, 0.4465],
        std=[0.2470, 0.2435, 0.2616],
    )
])

```

```

# Download CIFAR-10 dataset
if test:
    dataset = datasets.CIFAR10(
        root='./data', train=False,
        download=True, transform=transform_augmented_test,
    )

    data_loader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size, shuffle=shuffle,
        num_workers=num_workers
    )

    return data_loader

```

```

# Load the dataset
train_dataset = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_augmented)

valid_dataset = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_augmented)

num_train = len(train_dataset)
indices = list(range(num_train))
split = int(np.floor(valid_size * num_train))

if shuffle:
    np.random.seed(random_seed)
    np.random.shuffle(indices)

train_idx, valid_idx = indices[split:], indices[:split]

if save_si:

```

```

# Save 10% of training data in a new variable train_loader_SI
train_si_sampler = SubsetRandomSampler(indices[:int(0.1 * num_train)])
train_loader_SI = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, sampler=train_si_sampler,
num_workers=num_workers)
else:
    train_si_sampler = None
    train_loader_SI = None

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, sampler=train_sampler,
num_workers=num_workers)

valid_loader = torch.utils.data.DataLoader(
    valid_dataset, batch_size=batch_size, sampler=valid_sampler,
num_workers=num_workers)

return train_loader, valid_loader, train_loader_SI

train_loader, valid_loader, train_loader_SI = data_loader(batch_size=128,
num_workers=1, save_si=True)
test_loader = data_loader(batch_size=128, test=True, num_workers=1)

# Save loaders
save_loaders(train_loader, valid_loader, train_loader_SI, test_loader,
'data_loaders.pkl')

# Load loaders
train_loader, valid_loader, train_loader_SI, test_loader =
load_loaders('data_loaders.pkl')

```

این کد مجموعه ای از توابع را برای بارگذاری و تقویت مجموعه داده ها تعریف می کند. تابع اصلی است که بارگذارهای داده را برای آموزش، اعتبار سنجی (از ۱۰ درصد داده های ترین استفاده شده است) و مجموعه داده های تست ایجاد می کند. از مجموعه داده های **CIFAR-10** استفاده می کند و تکنیک های افزایش داده ها، مانند چرخش های افقی تصادفی، چرخش ها و **color jitter** را برای داده های آموزشی اعمال می کند (همانند روش هایی که در جدول ۳ گفته شده بود). مجموعه داده با استفاده از مقادیر میانگین و انحراف استاندارد دیتابست داده شده نرمال سازی می شود. همچنین زیرمجموعه ای از داده های آموزشی (۱۰٪) در یک بارگذار داده جداگانه **train\_loader\_SI** ذخیره می شود که می

تواند برای کارهایی مانند محاسبه **SI** استفاده شود. این کد همچنین امکان ذخیره و بارگذاری این بارگذارهای داده را با استفاده از توابع **load\_loaders** و **save\_loaders** به ترتیب با استفاده از **pickle** می‌دهد.

از فایل ذخیره شده در تمامی بخش‌ها استفاده خواهیم کرد.

- تابع **plot** و **training** و **test** دقت شبکه بر روی داده‌های تست

برای راحتی پیاده‌سازی و آموزش شبکه‌ها، موارد خواسته شده را در چند تابع آورده ایم.

```
def train_and_validate(model, train_loader, valid_loader, test_loader, num_epochs=15, early_stopping_threshold=10, device='cuda'):  
    # Define criterion, optimizer, and learning rate scheduler  
    criterion = nn.CrossEntropyLoss()  
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, weight_decay=5e-4, momentum=0.9)  
    scheduler = StepLR(optimizer, step_size=5, gamma=0.1)  
    # Initialize variables for tracking best accuracy and corresponding model weights  
    best_val_acc = 0.0  
    best_model_weights = None  
    early_stopping_counter = 0  
    # Lists to store metrics  
    train_loss_list = []  
    train_acc_list = []  
    val_loss_list = []  
    val_acc_list = []  
    total_step = len(train_loader)  
    for epoch in tqdm(range(num_epochs)):  
        model.train()  
        correct_train = 0  
        total_train = 0  
        running_loss = 0.0  
        for i, (images, labels) in enumerate(train_loader):  
            # Move tensors to the configured device  
            images = images.to(device)  
            labels = labels.to(device)  
            # Forward pass  
            outputs = model(images)  
            loss = criterion(outputs, labels)  
            # Backward and optimize  
            optimizer.zero_grad()  
            loss.backward()  
            optimizer.step()
```

```

# Calculate training accuracy
_, predicted_train = torch.max(outputs.data, 1)
total_train += labels.size(0)
correct_train += (predicted_train == labels).sum().item()
running_loss += loss.item()

# Calculate and store training accuracy and loss
epoch_train_accuracy = 100 * correct_train / total_train
epoch_train_loss = running_loss / len(train_loader)
train_acc_list.append(epoch_train_accuracy)
train_loss_list.append(epoch_train_loss)

# Step the learning rate scheduler
scheduler.step()

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

    # Calculate validation accuracy
    current_val_acc = 100 * correct / total

    # Store validation loss and accuracy
    val_loss_list.append(val_loss / len(valid_loader))
    val_acc_list.append(current_val_acc)

    # Check if the current accuracy is the best so far
    if current_val_acc > best_val_acc:
        best_val_acc = current_val_acc
        # Save the model weights
        best_model_weights = model.state_dict()
        # Reset early stopping counter
        early_stopping_counter = 0
    else:
        # Increment early stopping counter
        early_stopping_counter += 1

    # Check for early stopping
    if early_stopping_counter >= early_stopping_threshold:
        print(f'Early stopping after {early_stopping_counter} epochs
without improvement.')
        break

# Print and save metrics

```

```

        print('Epoch [{}/{}], Step [{}/{}], Training Loss: {:.4f}, Training
Accuracy: {:.2f}%, Validation Accuracy: {:.2f}%, Validation Loss: {:.4f}'
            .format(epoch + 1, num_epochs, i + 1, total_step, epoch_train_loss,
epoch_train_accuracy, current_val_acc, val_loss_list[-1]))
    # Print test set accuracy on the best weights
    test_accuracy = test_model(model, test_loader, device)
    print(f'Test Set Accuracy on Best Weights: {test_accuracy:.2f}%')
    # Save the model with the best validation accuracy
    torch.save(best_model_weights, 'best_model_weights.pth')
    # Plot training and validation metrics
    plot_metrics(train_loss_list, val_loss_list, train_acc_list, val_acc_list)
def test_model(model, test_loader, device='cuda'):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    test_accuracy = 100 * correct / total
    return test_accuracy
def plot_metrics(train_loss_list, val_loss_list, train_acc_list, val_acc_list):
    # Plot training and validation loss
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss_list, label='Training Loss')
    plt.plot(val_loss_list, label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    # Plot training and validation accuracy
    plt.subplot(1, 2, 2)
    plt.plot(train_acc_list, label='Training Accuracy')
    plt.plot(val_acc_list, label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.tight_layout()
    plt.show()

```

این کد یک حلقه آموزشی و اعتبار سنجی را برای مدل شبکه با استفاده از مجموعه داده CIFAR-10 پیاده سازی می کند. تابع `train_and_validate` معماری مدل، داده آموزشی و اعتبار سنجی، و پارامترهای اضافی مانند تعداد دوره های آموزشی، معیارهای توقف اولیه و `device` را به عنوان ورودی می گیرد. از شیب نزولی تصادفی SGD به عنوان بهینه ساز، از `cross-entropy loss` به عنوان استفاده می کند که نرخ یادگیری را هر ۵ دوره کاهش می دهد. `learning rate scheduler` می گیرد. از حلقه آموزشی در طول تعداد دوره های مشخص شده تکرار می شود و وزن های مدل را بر اساس تلفات محاسبه شده و `backpropagation` به روزرسانی می کند. همچنین پس از هر دوره اعتبار سنجی را انجام می دهد و بهترین دقت اعتبار سنجی را ذخیره می کند. توقف زودهنگام برای توقف آموزش در صورت عدم بهبود در دقت اعتبار سنجی برای تعداد معینی از دوره های متوالی استفاده می شود.

این کد شامل یک تابع `test_model` برای ارزیابی مدل در یک مجموعه آزمایشی جداگانه و محاسبه دقت آن است. تابع `plot_metrics` برای رسم از `loss` آموزش و اعتبار سنجی و همچنین دقت در دوره ها استفاده می شود. دقت نهایی در مجموعه آزمایشی چاپ می شود و وزن های مدل که به بهترین دقت اعتبار سنجی دست می یابند در فایلی با نام `best_model_weights.pth` ذخیره می شوند.

(تمامی نمودار ها و دقت ها بر روی بهترین وزن که به بهترین دقت اعتبار سنجی رسیده است را نشان می دهند).

برای استفاده از تابع های بالا کد زیر را به عنوان مثال ران می کنیم:

```
train_and_validate(model, train_loader, valid_loader, test_loader,
num_epochs=200, early_stopping_threshold=20, device='cuda:0')
```

- محاسبه متريک های خواسته شده:

در تمرین دوم کد ها بهینه شده SI توضیح کامل داده شده است که در اینجا از آن ها استفاده می کنیم. کد ها در لینک زیر قرار گرفته اند:

git clone [https://github.com/Arhosseini77/data\\_complexity\\_measures](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:

```
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
model = VGG11_third_layer().to(device)
model.load_state_dict(torch.load('third_layer_freeze.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.

هدف این کد استخراج ویژگی های میانی از هر لایه یک مدل شبکه عصبی است. برای ذخیره ویژگی های هر لایه شروع می شود. تعداد کل لایه ها در مدل با استفاده از طول لیست `model.children` تعیین می شود. پس از آن، یک حلقه از طریق هر لایه `model.children` با استفاده از روش "named\_children" تکرار می شود. یک هوک رو به جلو برای لایه ثبت می شود. این قلاب که به عنوان تابعی به نام «hook» تعریف می شود، ویژگی های خروجی لایه را به ورودی مربوطه در «features\_per\_layer» اضافه می کند. با اتصال این قلاب ها به لایه های مورد نظر، کد مجموعه ای از ویژگی های میانی را در حین محاسبات forward pass امکان پذیر می سازد، و تحلیل بیشتر یا تجسم نمایش های داخلی مدل را تسهیل می کند.

```
# Prepare storage for outputs and labels
features_per_layer = {}
labels_list = []
# Define layers to exclude
exclude_layers = {''}
# Function to attach hooks
def get_layer_outputs(layer_name):
    def hook(module, input, output):
        features_per_layer[layer_name].append(output.detach())
    return hook
# Attach hooks to each layer except the excluded ones
for name, layer in model.named_children():
    if name not in exclude_layers:
        features_per_layer[name] = []
        layer.register_forward_hook(get_layer_outputs(name))
```

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، ویژگی های میانی جمع آوری شده در طول گذر رو به جلو برای هر لایه، مسطح شده و در امتداد بعد دوم به هم متصل می شوند و یک تنسور دو بعدی برای هر لایه تشکیل می دهند. `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.

```
# Pass data through the model and collect layer outputs
with torch.no_grad():
    for inputs, targets in tqdm(train_loader):
        inputs = inputs.to('cuda:1' if torch.cuda.is_available() else 'cpu')
        # Trigger the hooks and collect layer outputs
        model(inputs)
        labels_list.append(targets.cpu())
        # Clear CUDA cache after processing each batch
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
# Post-process the data: Flatten and concatenate
for layer_name, layer_features in features_per_layer.items():
    if layer_features: # Check if layer_features is not empty
        try:
            features_per_layer[layer_name] = torch.cat([f.view(f.size(0), -1) for
f in layer_features])
        except RuntimeError as e:
            print(f"Error in concatenating features of layer {layer_name}")
            for f in layer_features:
                print(f.shape)
            raise e
# Concatenate the labels
labels = torch.cat(labels_list)
csi_layer_train = []
# Iterate through each layer's features in the dictionary
for layer_name, features in features_per_layer.items():
    instance_disturbance = ARH_SeparationIndex(features, labels, normalize=True)
    csi = instance_disturbance.center_si_batch(batch_size=2000)
    csi_layer_train.append((layer_name, csi))
print(csi_layer_train)
```

```

# Plotting SI versus layer using a line plot
plt.plot([layer for layer, _ in csi_layer_train], [si for _, si in
csi_layer_train])
plt.xlabel('Layer')
plt.ylabel('SI')
plt.title('Center Separation Index (CSI) vs Layer')
plt.xticks(rotation=45)
plt.grid(True)
plt.show()

```

کد بالا را یکبار برای داده تست و یکبار برای داده تربین ران می کنیم و نتایج را ذخیره میکنیم.

الف)

مدل گفته شده در صورت سوال را بصورت زیر پیاده سازی میکنیم:

```

class VGG11(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG11, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU()
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU()
        )

```

```

)
self.layer6 = nn.Sequential(
    nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2)
)
self.layer7 = nn.Sequential(
    nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(512),
    nn.ReLU()
)
self.layer8 = nn.Sequential(
    nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2)
)
self.fc1 = nn.Sequential(
    nn.Linear(512, 512)
)
self.fc2 = nn.Sequential(
    nn.Linear(512, num_classes)
)
def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = self.layer3(out)
    out = self.layer4(out)
    out = self.layer5(out)
    out = self.layer6(out)
    out = self.layer7(out)
    out = self.layer8(out)
    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.fc2(out)
    return out
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
model = VGG11().to(device)

```

از `summary` مدل که در شکل ۱ نشان داده شده است، متوجه می شویم که مدل مطابق با معماری که در صورت سوال گفته شده است، پیاده سازی شده است.

Layer (type)	Output Shape	Param #
Conv2d-1	[1, 64, 32, 32]	1,792
BatchNorm2d-2	[1, 64, 32, 32]	128
ReLU-3	[1, 64, 32, 32]	0
MaxPool2d-4	[1, 64, 16, 16]	0
Conv2d-5	[1, 128, 16, 16]	73,856
BatchNorm2d-6	[1, 128, 16, 16]	256
ReLU-7	[1, 128, 16, 16]	0
MaxPool2d-8	[1, 128, 8, 8]	0
Conv2d-9	[1, 256, 8, 8]	295,168
BatchNorm2d-10	[1, 256, 8, 8]	512
ReLU-11	[1, 256, 8, 8]	0
Conv2d-12	[1, 256, 8, 8]	590,080
BatchNorm2d-13	[1, 256, 8, 8]	512
ReLU-14	[1, 256, 8, 8]	0
MaxPool2d-15	[1, 256, 4, 4]	0
Conv2d-16	[1, 512, 4, 4]	1,180,160
BatchNorm2d-17	[1, 512, 4, 4]	1,024
ReLU-18	[1, 512, 4, 4]	0
Conv2d-19	[1, 512, 4, 4]	2,359,808
BatchNorm2d-20	[1, 512, 4, 4]	1,024
ReLU-21	[1, 512, 4, 4]	0
MaxPool2d-22	[1, 512, 2, 2]	0
Conv2d-23	[1, 512, 2, 2]	2,359,808
BatchNorm2d-24	[1, 512, 2, 2]	1,024
ReLU-25	[1, 512, 2, 2]	0
Conv2d-26	[1, 512, 2, 2]	2,359,808
BatchNorm2d-27	[1, 512, 2, 2]	1,024
ReLU-28	[1, 512, 2, 2]	0
MaxPool2d-29	[1, 512, 1, 1]	0
Linear-30	[1, 512]	262,656
Linear-31	[1, 10]	5,130

Total params: 9,493,770  
Trainable params: 9,493,770  
Non-trainable params: 0

Input size (MB): 0.01  
Forward/backward pass size (MB): 3.71  
Params size (MB): 36.22  
Estimated Total Size (MB): 39.94

شکل ۱ – مدل بخش الف

با ران کردن کد زیر، مدل را ترین می کنیم:

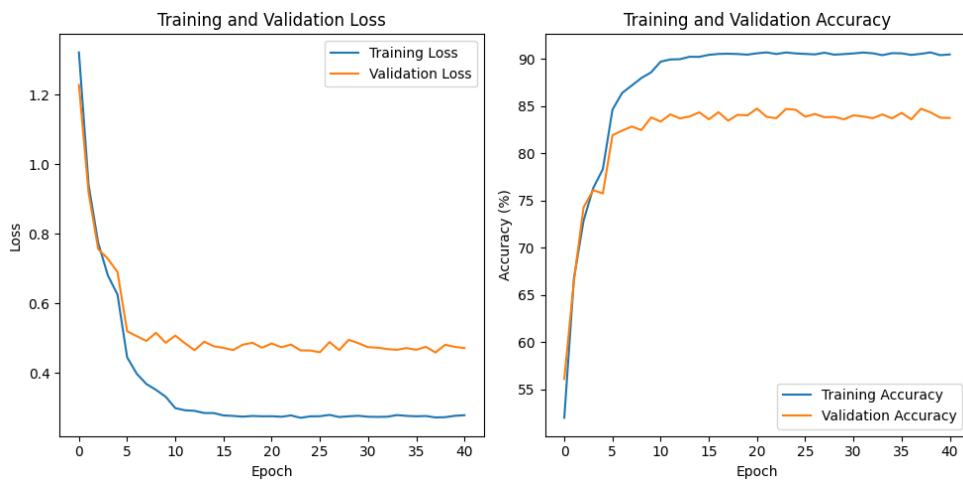
```
train_and_validate(model, train_loader, valid_loader, test_loader,
num_epochs=200, early_stopping_threshold=20, device='cuda:0')
```

قرار دادیم که اگر دقت مدل پیشرفت خاصی نکرد، متوقف شود در پایین میبینیم که در ۲۵ epoch training متوقف شده است و به دقت دلخواه نیز رسیده ایم:

دقت روی داده های تست:

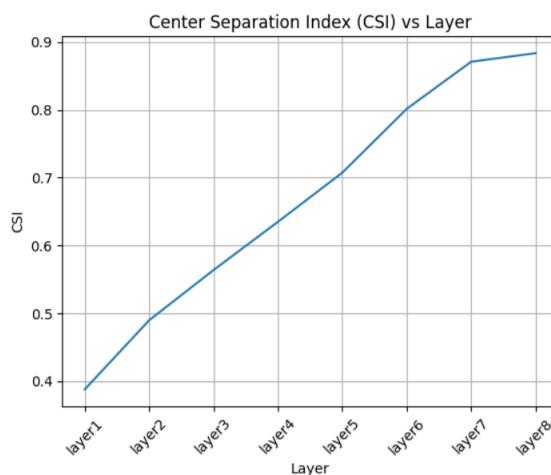
Test Set Accuracy on Best Weights: 85.20%

نمودارهای accuracy و loss مدل اول مطابق شکل زیر است:

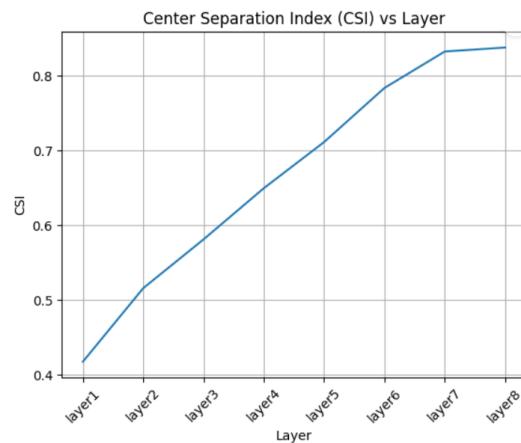


شکل ۲ - نمودارهای accuracy و loss مدل اول

متريک ها روی داده های ترين و تست:



شکل ۳ - CSI روی داده های train



شکل ۴ - CSI روی داده های test

در این قسمت مشاهده میکنیم که اگر تمام لایه هارا باهم ترین کنیم، دقت روی داده های تست به ۸۵ می رسد که دقت قابل قبولی است، همچنین 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، چون مقدار متريک ما افزایش پیدا کرده است. (مقایسه تمام روش ها در آخر سوال آورده شده است). (نمودارهایی که لایه های FC هم مشخص شده در فایل های ضمیمه شده وجود دارد.)

(ب)

در این قسمت لایه های conv را در هر مرحله اضافه میکنیم و لایه های قبل را freeze میکنیم.

به عنوان مثال ابتدا لایه ی اول را به شکل زیر آموزش میدهیم:

```
class VGG_first_layer(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG_first_layer, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc1 = nn.Sequential(
            nn.Linear(64 * 16 * 16, 512)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        out = self.layer1(x)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        return out

model = VGG_first_layer().to(device)
.....
train_and_validate(model, train_loader, valid_loader, test_loader, num_epochs=15,
early_stopping_threshold=10, device='cuda:0' , save_filename =
'first_layer_freeze.pth' )
```

وزن این قسمت به اسم first\_layer\_freeze.pth ذخیره میشود.

برای train لایه دوم ابتدا لایه اول را `freeze` میکنیم و لایه دوم را با دو تا `fc` آموزش می دهیم، کردن لایه اول به شکل زیر است:

```
class VGG11_second_layer(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG11_second_layer, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(8192, 512)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, num_classes)
        )
    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

ابتدا مدل لایه بعد را تعریف میکنیم، وزن های لایه قبل را لود میکنیم، دو لایه `fc` قسمت قبل را از وزن ها در نظر نمیگیریم و پارامترهای لایه اول را `false` میکنیم که آموزش نبیند و از وزن هایی که لود کردیم استفاده می کند.

```
# Initialize your model
model = VGG11_second_layer().to(device)
# Load your pretrained weights for the first layer
saved_state_dict = torch.load('first_layer_freeze.pth')
# Filter out unnecessary keys
filtered_state_dict = {k: v for k, v in saved_state_dict.items() if 'layer1' in k}
# Load the filtered state dict (only first layer weights)
model.load_state_dict(filtered_state_dict, strict=False)
# Freeze the first layer
```

```

for param in model.layer1.parameters():
    param.requires_grad = False

model

```

همین روند را برای بقیه لایه ها نیز انجام می دهیم (کد کامل این بخش در فایل سوال ۱ بخش ب موجود است – برای طولانی نشدن گزارش از آوردن آن ها خودداری کردیم)

برای لایه هشتم همانند زیر این لایه را **train** میکنیم:

ابدا کل لایه ها را با ۲ تا **fc** تعریف میکنیم:

```

class VGG11_eighth_layer(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG11_eighth_layer, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
        )
        self.layer6 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer7 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),

```

```

        nn.ReLU())
    self.layer8 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2))
    self.fc1 = nn.Sequential(
        nn.Linear(512, 512))
    self.fc2 = nn.Sequential(
        nn.Linear(512, num_classes))

def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = self.layer3(out)
    out = self.layer4(out)
    out = self.layer5(out)
    out = self.layer6(out)
    out = self.layer7(out)
    out = self.layer8(out)
    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.fc2(out)
    return out

```

در ادامه وزن های ۷ تا لایه قبلی را لود میکنیم و پارامترهای این لایه هارا همانند زیر، `freeze` میکنیم که قابل `train` شدن نباشد.

```

model = VGG11_eighth_layer().to(device)
# Define the paths to your weight files
weight_files = {
    'layer1': 'first_layer_freeze.pth',
    'layer2': 'second_layer_freeze.pth',
    'layer3': 'third_layer_freeze.pth',
    'layer4': 'fourth_layer_freeze.pth',
    'layer5': 'fifth_layer_freeze.pth',
    'layer6': 'sixth_layer_freeze.pth',
    'layer7': 'seventh_layer_freeze.pth'
}
# Load and update the model's state dict for each layer
for layer_name, file_path in weight_files.items():
    pretrained_dict = torch.load(file_path)
    model_dict = model.state_dict()
    # Filter out FC layer weights and update the state dict for each layer
    filtered_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict and 'fc' not in k}
    model_dict.update(filtered_dict)
    model.load_state_dict(model_dict)

```

```

# Freeze the first three layers
for layer in [model.layer1, model.layer2, model.layer3, model.layer4,
model.layer5, model.layer6, model.layer7]:
    for param in layer.parameters():
        param.requires_grad = False

VGG11_eighth_layer.layer6: requires_grad=False
VGG11_eighth_layer.layer6: requires_grad=False
VGG11_eighth_layer.layer6: requires_grad=False
VGG11_eighth_layer.layer6: requires_grad=False
Sequential.0: requires_grad=False
Sequential.0: requires_grad=False
Sequential.1: requires_grad=False
Sequential.1: requires_grad=False
VGG11_eighth_layer.layer7: requires_grad=False
VGG11_eighth_layer.layer7: requires_grad=False
VGG11_eighth_layer.layer7: requires_grad=False
VGG11_eighth_layer.layer7: requires_grad=False
Sequential.0: requires_grad=False
Sequential.0: requires_grad=False
Sequential.1: requires_grad=False
Sequential.1: requires_grad=False
VGG11_eighth_layer.layer8: requires_grad=True
VGG11_eighth_layer.layer8: requires_grad=True
VGG11_eighth_layer.layer8: requires_grad=True
VGG11_eighth_layer.layer8: requires_grad=True
Sequential.0: requires_grad=True
Sequential.0: requires_grad=True
Sequential.1: requires_grad=True
Sequential.1: requires_grad=True
VGG11_eighth_layer.fc1: requires_grad=True
VGG11_eighth_layer.fc1: requires_grad=True
Sequential.0: requires_grad=True
Sequential.0: requires_grad=True
VGG11_eighth_layer.fc2: requires_grad=True
VGG11_eighth_layer.fc2: requires_grad=True
Sequential.0: requires_grad=True
Sequential.0: requires_grad=True

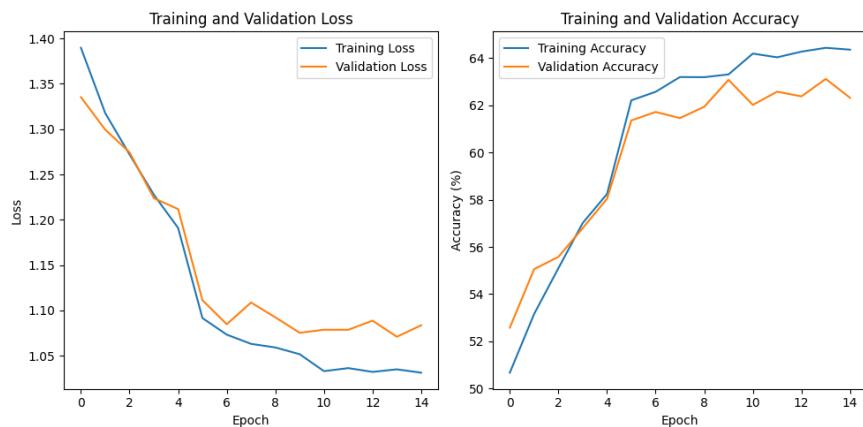
```

در بالا مشاهده میکنیم که پارامترهای لایه هشتم و دو تا `fc` قابل آموزش هستند و بقیه لایه ها هستند. (این نمایش را برای تمام لایه ها نمایش دادیم که در فایل کد قسمت دوم قابل مشاهده است).

لایه ها را تک تک آموزش می دهیم و به نتایج زیر میرسیم:

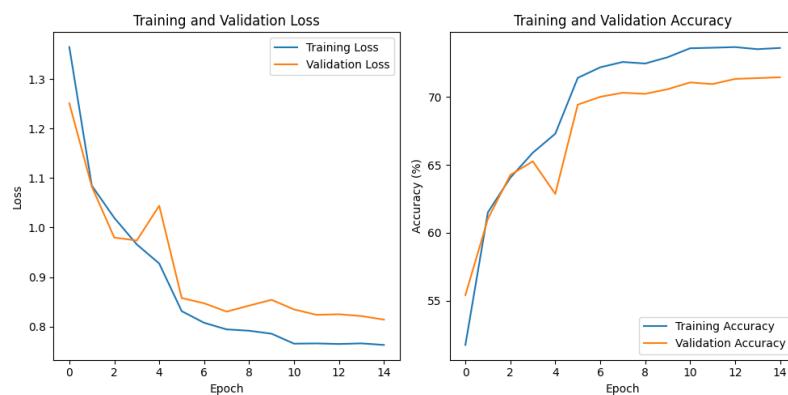
نمودارهای `accuracy` و `loss` مدل برای هر لایه مطابق شکل زیر است:

لایه اول:



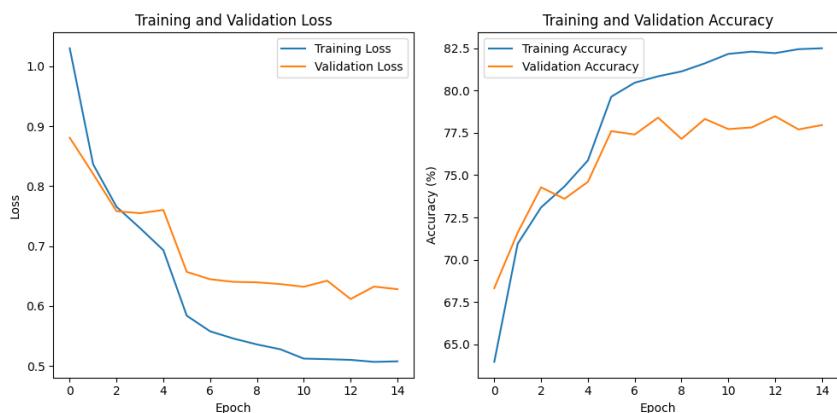
شکل ۵ - نمودارهای accuracy و loss لایه اول

لایه دوم:



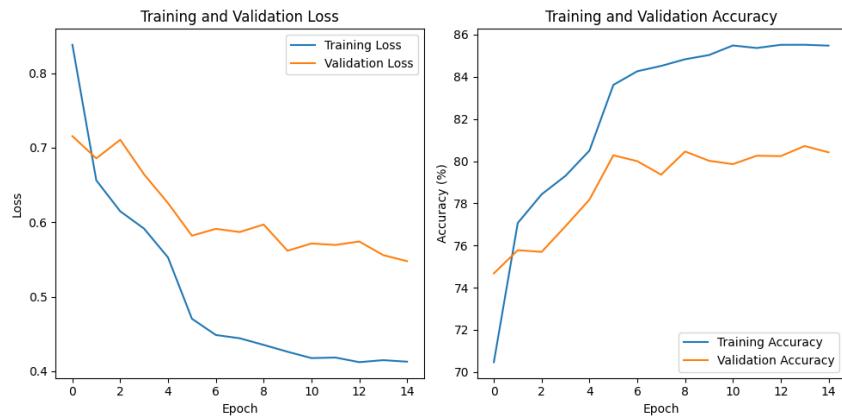
شکل ۶ - نمودارهای accuracy و loss لایه دوم

لایه سوم:



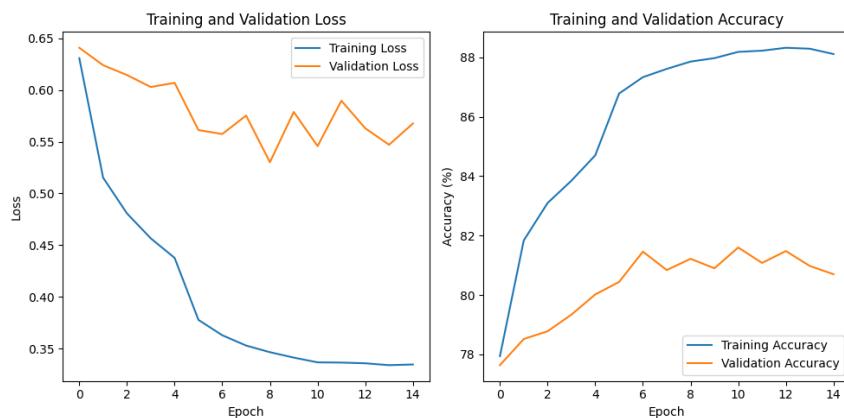
شکل ۷ - نمودارهای accuracy و loss لایه سوم

لایه چهارم:



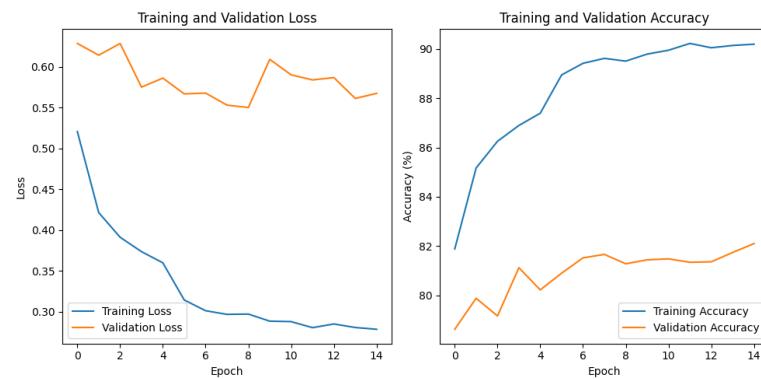
شکل ۸ - نمودارهای loss و accuracy لایه چهارم

لایه پنجم:



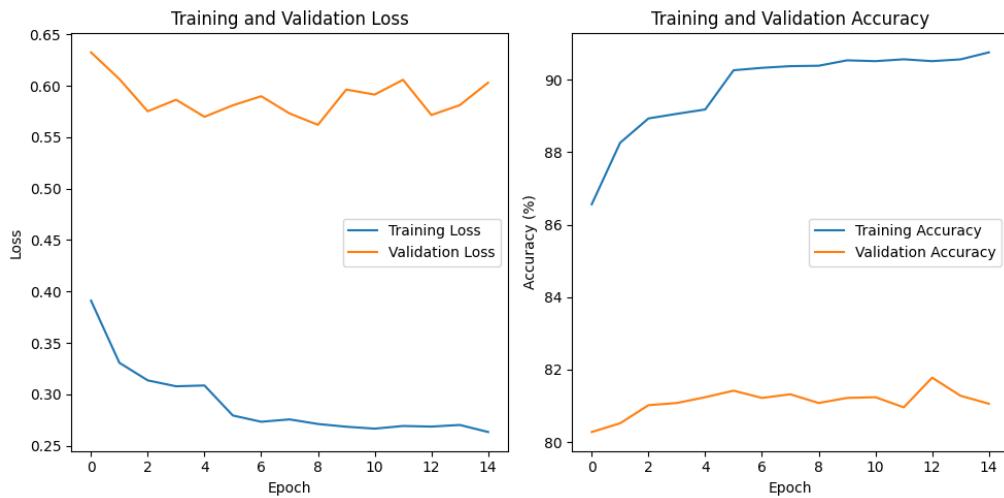
شکل ۹ - نمودارهای loss و accuracy لایه پنجم

لایه ششم:



شکل ۱۰ - نمودارهای loss و accuracy لایه ششم

لایه هفتم:



شکل ۱۱ - نمودارهای loss و accuracy لایه هفتم

لایه هشتم:



شکل ۱۲ - نمودارهای loss و accuracy لایه هشتم

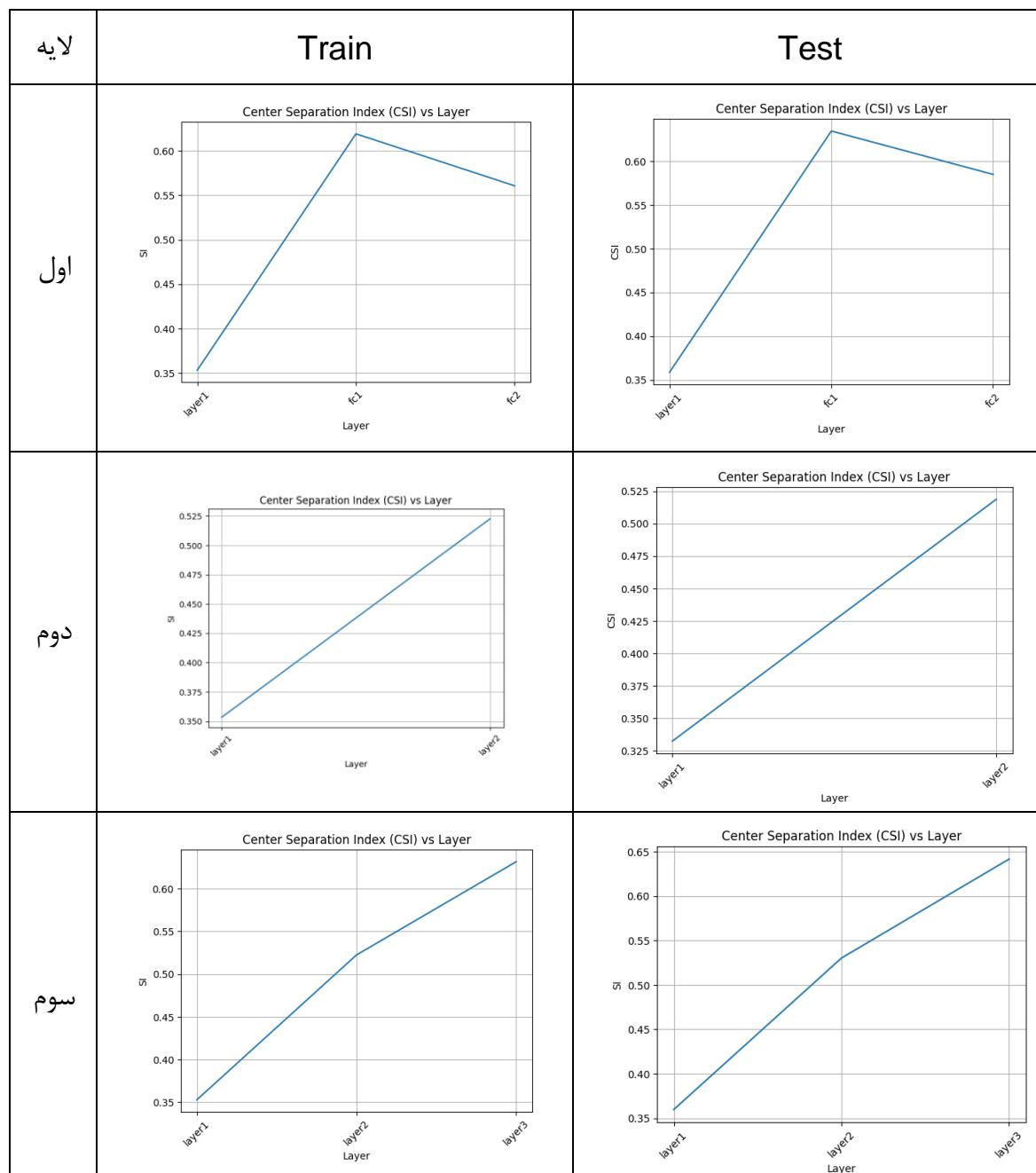
جدول ۱ - دقت روی داده های تست در هر لایه

شماره لایه	اول	دوم	سوم	چهارم	پنجم	ششم	هفتم	هشتم
دقت تست	۶۵,۱۲	۷۴,۰۷	۷۹,۹۷	۸۲,۵۸	۸۲,۹۹	۸۳,۱۷	۸۲,۹۱	۸۳,۰۰

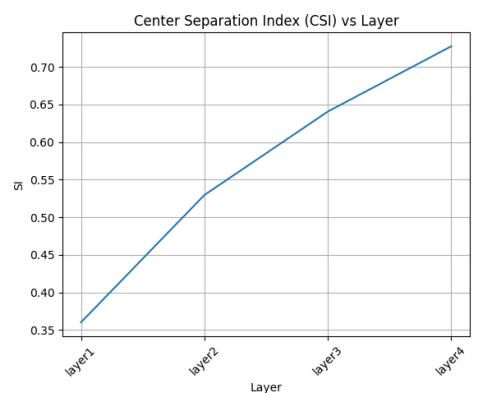
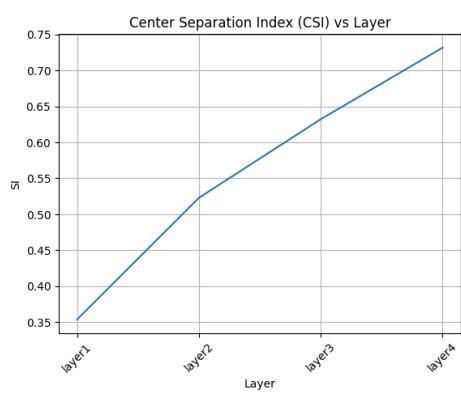
مشاهده میکنیم که در هر لایه دقت روی داده های تست روند افزایشی داشته است و به عدد مورد نظر ما دارد نزدیک می شود، و لاس نیز کاهشی بوده است.

متريک ها روی داده های ترين و تست برای هر لایه: (شكل نمودارها با درنظر گرفتن  $\kappa$ ها در فايل کد قرار دارد.) (محور CSIها است)

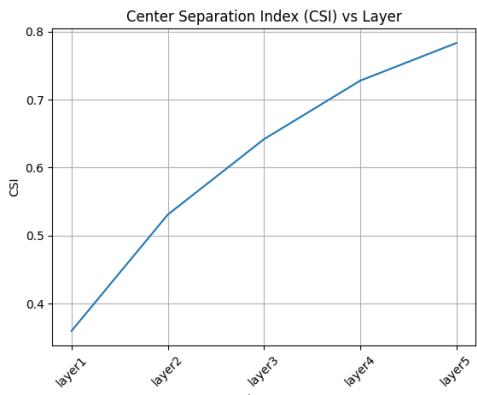
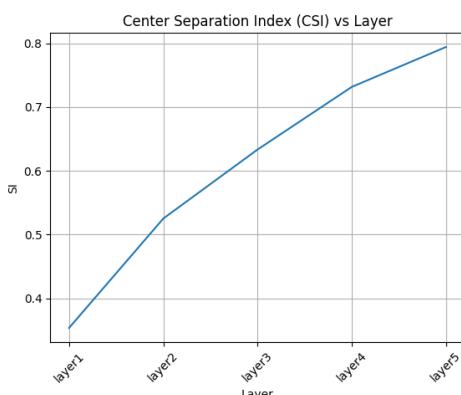
جدول ۲ - متريک ها روی داده های ترين و تست برای هر لایه



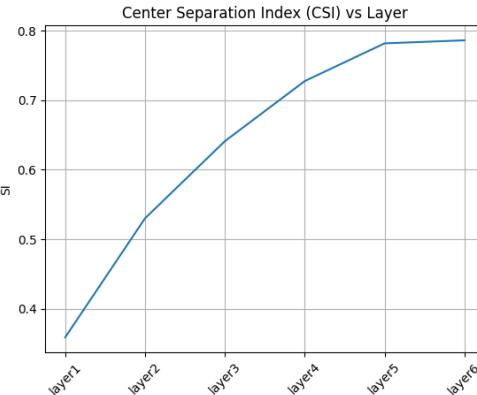
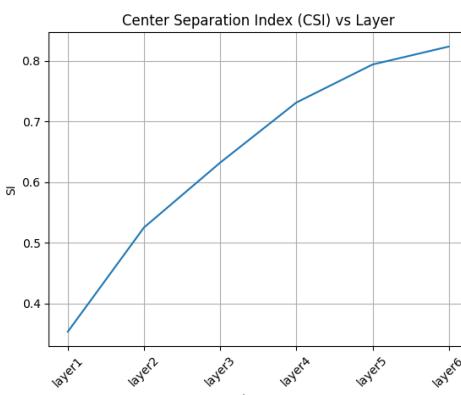
چهار



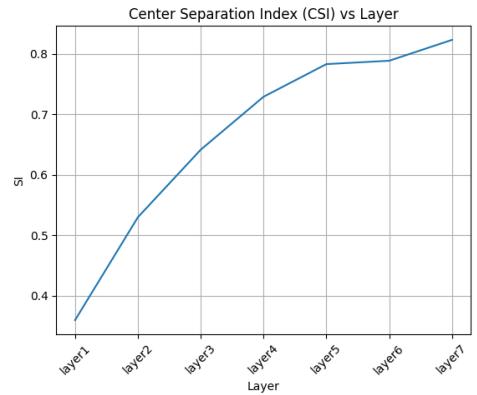
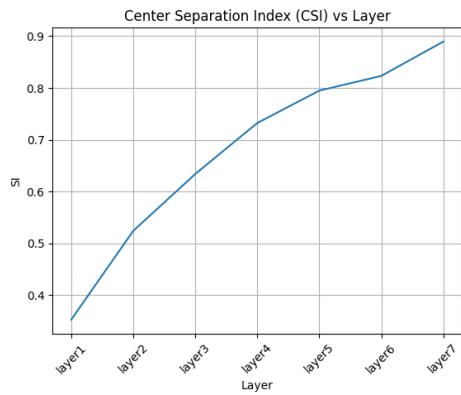
پنجم

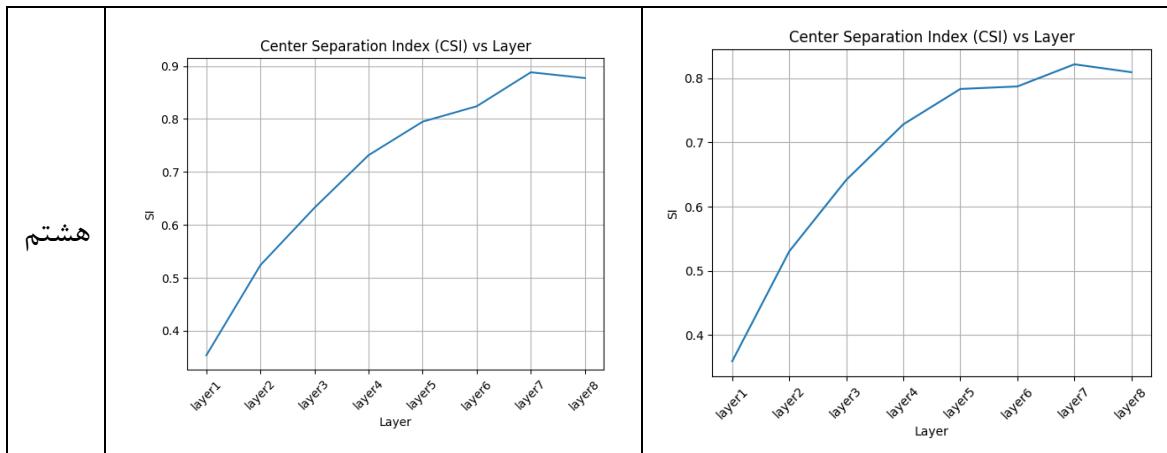


ششم



هفتم





جدول ۳ - مقایسه متريک ها روی داده های ترين برای هر لایه با روش الف

	Training in once	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
layer 1	0.42	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35
layer 2	0.52		0.52	0.52	0.52	0.53	0.53	0.52	0.52
layer 3	0.58			0.63	0.63	0.63	0.63	0.63	0.63
layer 4	0.65				0.73	0.73	0.73	0.73	0.73
layer 5	0.71					0.79	0.79	0.79	0.79
layer 6	0.78						0.82	0.82	0.82
layer 7	0.83							0.89	0.89
layer 8	0.84								0.88

مشاهده می کنیم در این حالت که تک تک لایه ها را آموزش می دهیم بر روی داده های آموزش به CSI بالاتری می رسیم ولی روی داده های تست به CSI کمتری رسیدیم.

جدول ۴ - مقایسه متريک ها روی داده های تست برای هر لایه با روش الف

	Training in once	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
layer 1	0.42	0.36	0.33	0.36	0.33	0.36	0.36	0.36	0.36
layer 2	0.52		0.52	0.53	0.52	0.53	0.53	0.53	0.53
layer 3	0.58			0.64	0.63	0.64	0.64	0.64	0.64
layer 4	0.65				0.72	0.73	0.73	0.73	0.73
layer 5	0.71					0.78	0.78	0.78	0.78
layer 6	0.78						0.79	0.79	0.79
layer 7	0.83							0.82	0.82
layer 8	0.84								0.81

همچنین مشاهده می کنیم که CSI ها برای هر لایه در هر stage تقریبا ثابت مانده است.

(ج)

در این قسمت همانند قسمت قبلی لایه ها را تعریف میکنیم، فقط لایه ها را برای آموزش `freeze` نمیکنیم.

به عنوان مثال لایه اول را به شکل زیر تعریف و ترین می کنیم:

```
class VGG_first_layer(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG_first_layer, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc1 = nn.Sequential(
            nn.Linear(64 * 16 * 16, 512)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, num_classes)
    )

    def forward(self, x):
        out = self.layer1(x)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

بعد از آموزش لایه بالا، لایه بعدی را به شکل زیر آموزش می دهیم:

```
class VGG11_second_layer(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG11_second_layer, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
```

```

        nn.MaxPool2d(kernel_size=2, stride=2))
self.fc1 = nn.Sequential(
    nn.Linear(8192, 512))
self.fc2 = nn.Sequential(
    nn.Linear(512, num_classes))

def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.fc2(out)
    return out

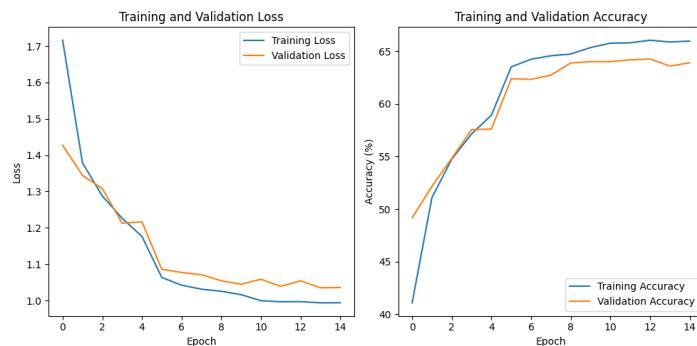
```

همین کار را تا انتهای انجام می دهیم و هر بار یک لایه را اضافه می کنیم.

لایه ها را تک تک آموزش می دهیم و به نتایج زیر میرسیم:

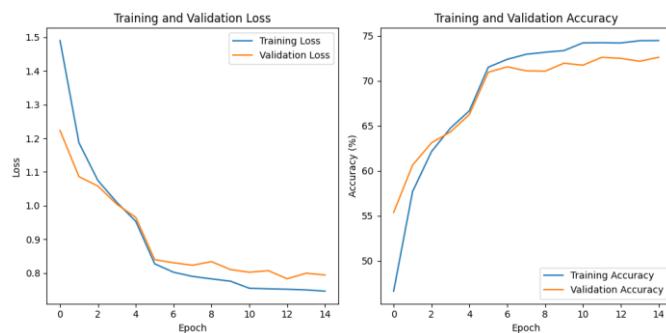
نمودارهای **accuracy** و **loss** مدل برای هر لایه مطابق شکل زیر است:

لایه اول:



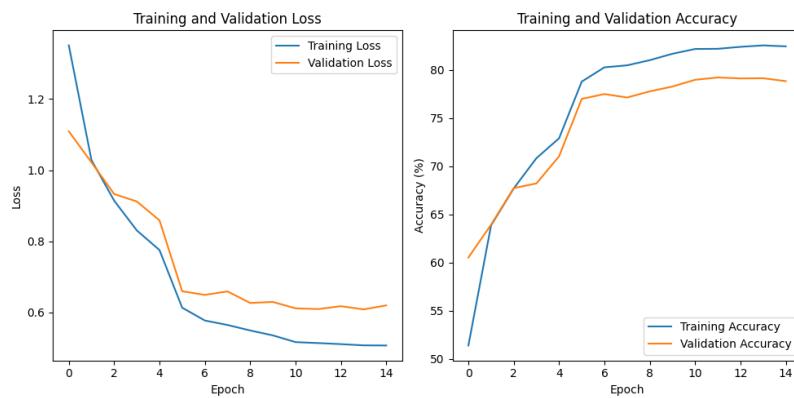
شکل ۱۳ - نمودارهای **accuracy** و **loss** لایه اول

لایه دوم:



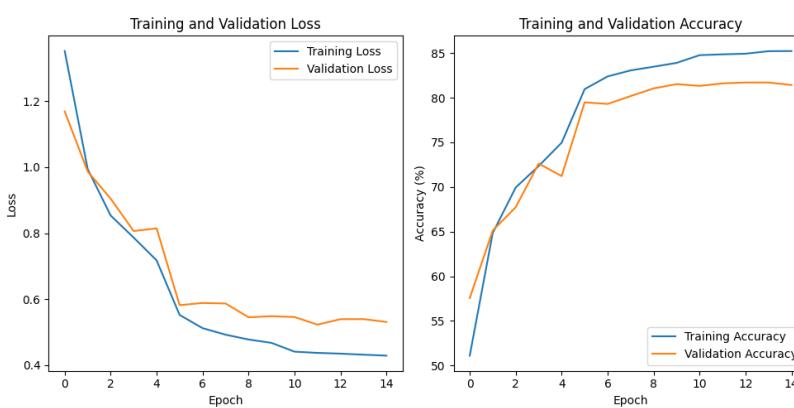
شکل ۱۴ - نمودارهای **accuracy** و **loss** لایه دوم

لایه سوم:



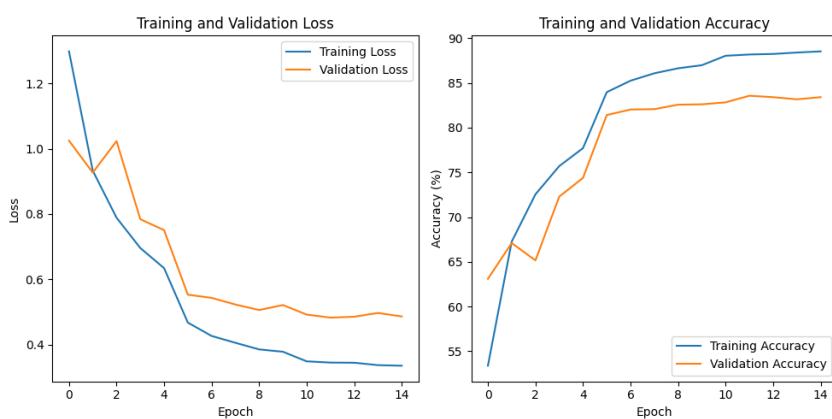
شکل ۱۵ - نمودارهای accuracy و loss لایه سوم

لایه چهارم:



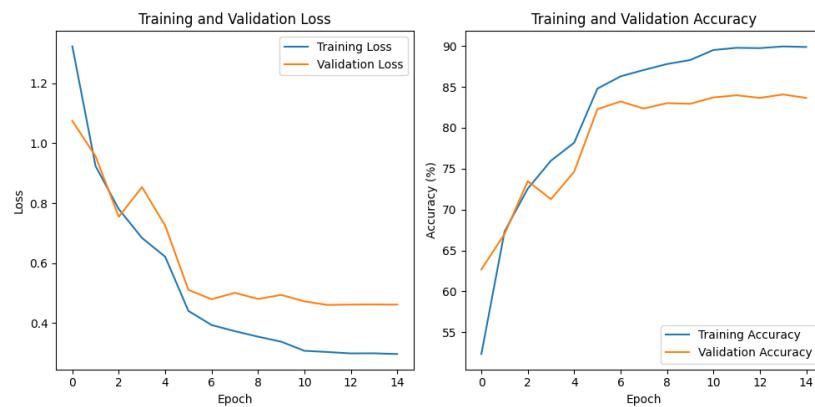
شکل ۱۶ - نمودارهای accuracy و loss لایه چهارم

لایه پنجم:



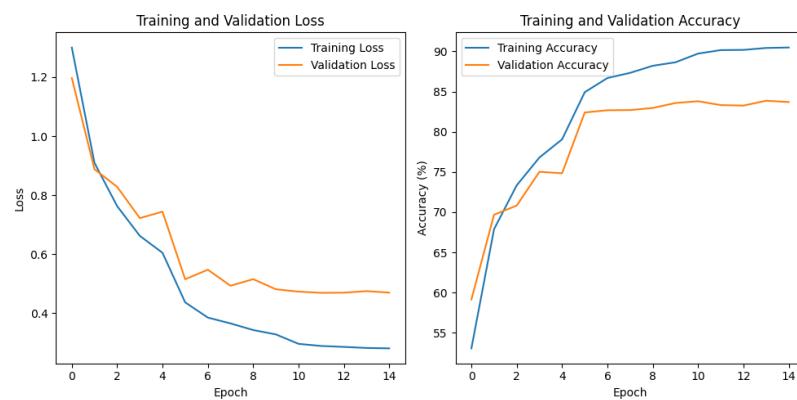
شکل ۱۷ - نمودارهای accuracy و loss لایه پنجم

لایه ششم:



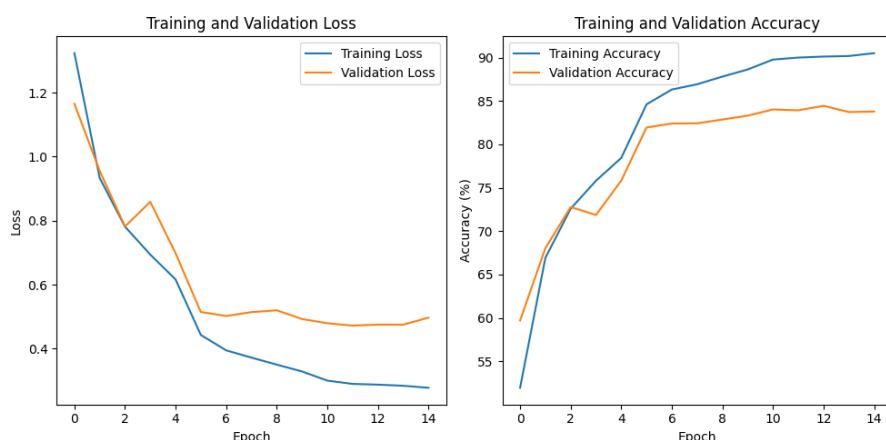
شکل ۱۸ - نمودارهای loss و accuracy لایه ششم

لایه هفتم:



شکل ۱۹ - نمودارهای loss و accuracy لایه هفتم

لایه هشتم:



شکل ۲۰ - نمودارهای loss و accuracy لایه هشتم

جدول ۵ - دقیق روحی داده های تست در هر لایه

شماره لایه	اول	دوم	سوم	چهارم	پنجم	ششم	هفتم	هشتم
دقیق تست freeze	۶۵,۱۲	۷۴,۰۷	۷۹,۹۷	۸۲,۵۸	۸۲,۹۹	۸۳,۱۷	۸۲,۹۱	۸۳,۰۰
دقیق تست No- freeze	۶۷,۳۲	۷۴,۶۵	۸۰,۶۰	۸۲,۸۲	۸۴,۲۴	۸۵,۲۰	۸۵,۳۷	۸۴,۸۸

مشاهده میکنیم که در هر لایه دقیق روحی داده های تست روند افزایشی داشته است و به عدد مورد نظر ما دارد نزدیک می شود، و لاس نیز کاهشی بوده است. در حالتی که لایه ها را freeze نمیکنیم، مشاهده میکنیم که به دقیق بالاتری نسبت به حالت قبل رسیدیم.

دلیل بهتر بودن این روش:

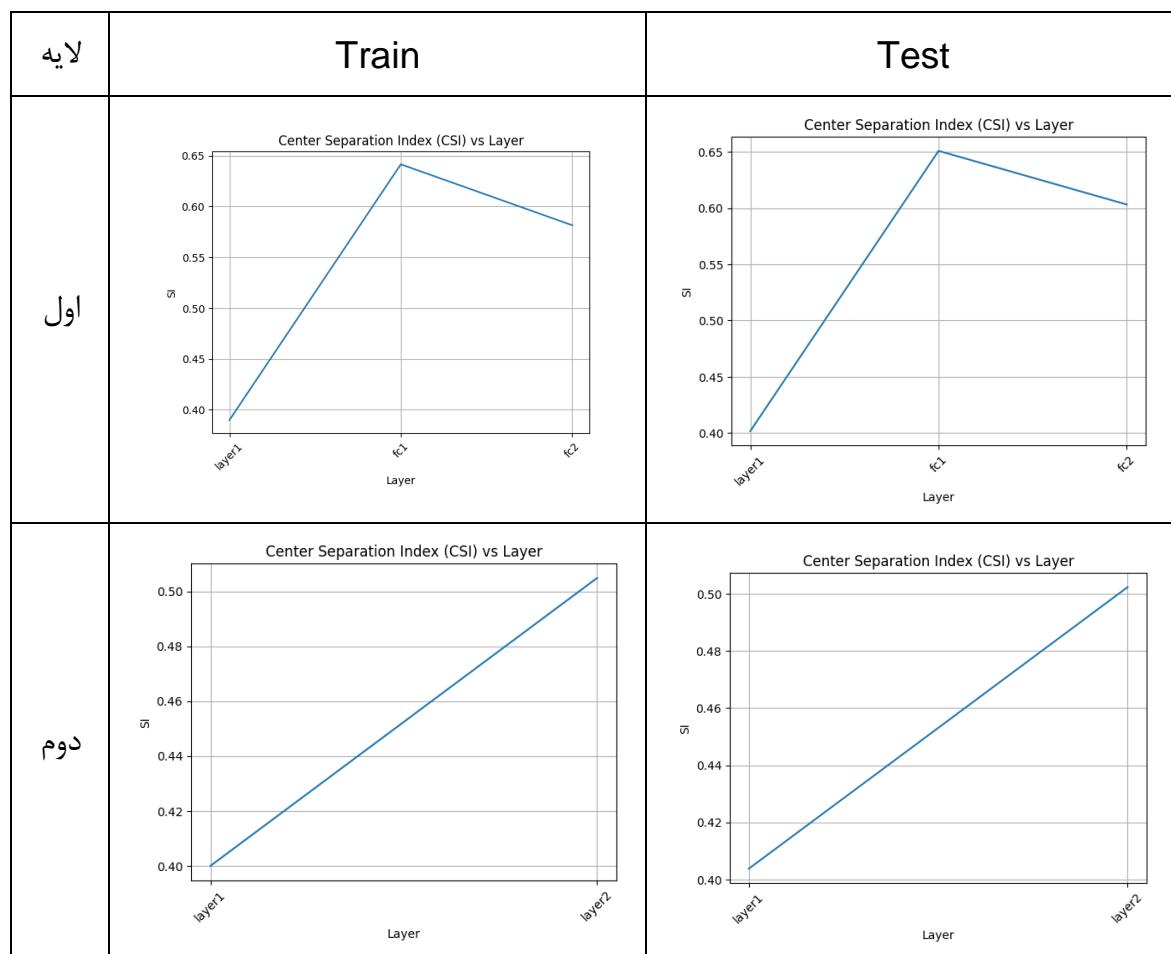
در این حالت، هر لایه می تواند با ویژگی های آموخته شده توسط لایه های قبلی سازگار شود. این رویکرد یادگیری به کل مدل اجازه می دهد تا به طور جمی پارامترهای خود را بهینه سازی کند تا الگوهای نمایش های پیچیده در داده ها را بهتر به تصویر بکشد. اجازه دادن به هر لایه برای به روز رسانی پارامترهای خود ممکن است تعامل بهتری بین ویژگی ها در سراسر لایه ها ایجاد کند. در برخی موارد، ویژگی هایی که در لایه های بعدی آموخته می شوند ممکن است اطلاعات ارزشمندی را در اختیار لایه های قبلی قرار دهند و بالعکس. اگر توزیع داده ها در طول زمان یا در بین لایه ها تغییر می کند، انجماد لایه های قبلی در روش قبل ممکن است از تطبیق آنها با این تغییرات جلوگیری کند. آموزش هر لایه در روش ۲ به کل مدل اجازه می دهد تا با ویژگی های در حال تکامل داده ها سازگار شود.

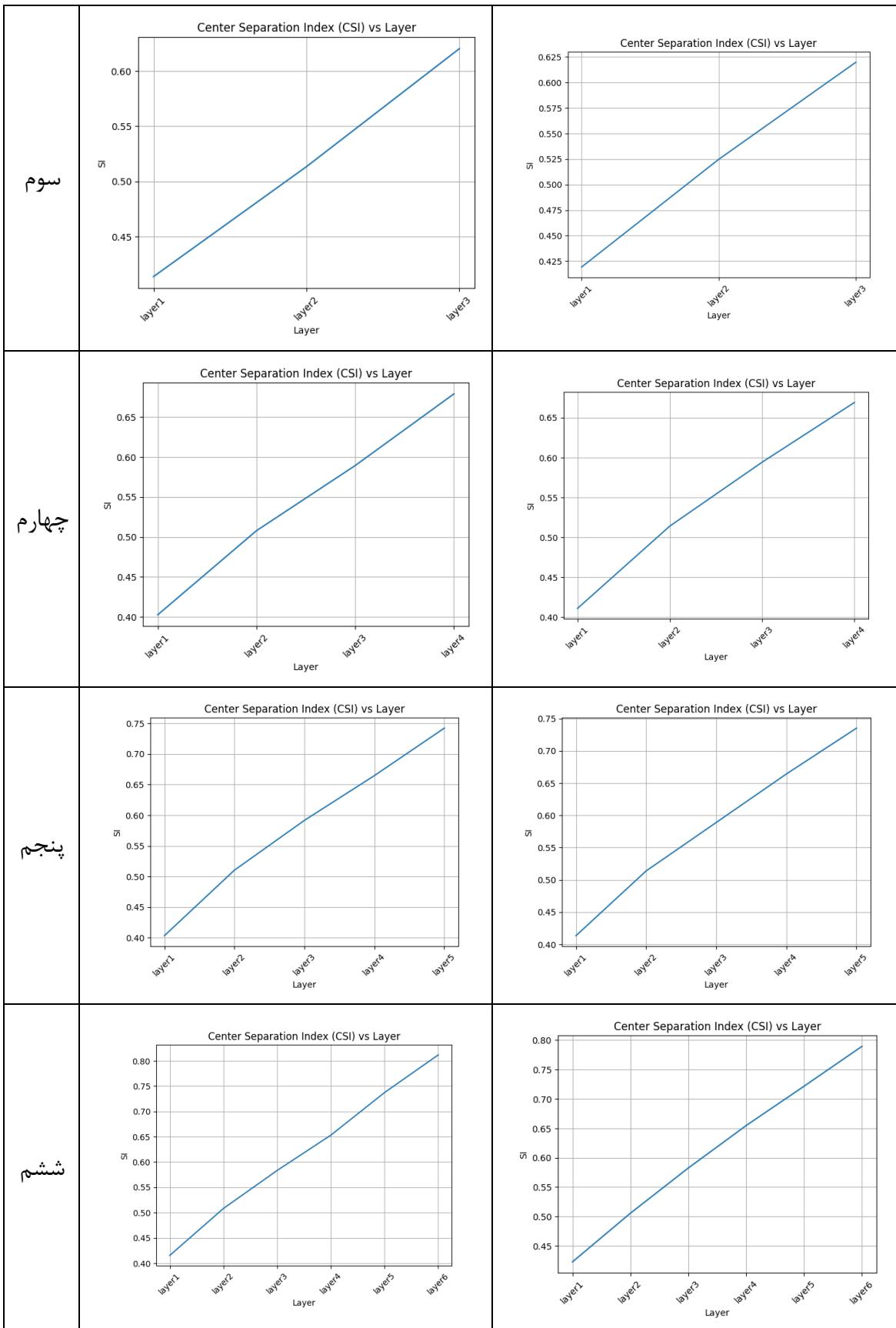
انجام لایه های قبلی در روش قبل ممکن است به مشکل ناپدید شدن گرادیان منجر شود. گرادیان ها ممکن است در طول **backpropagation** بسیار کوچک شوند و این امر باعث می شود که لایه های منجمد به طور موثر وزن خود را به روز کنند.

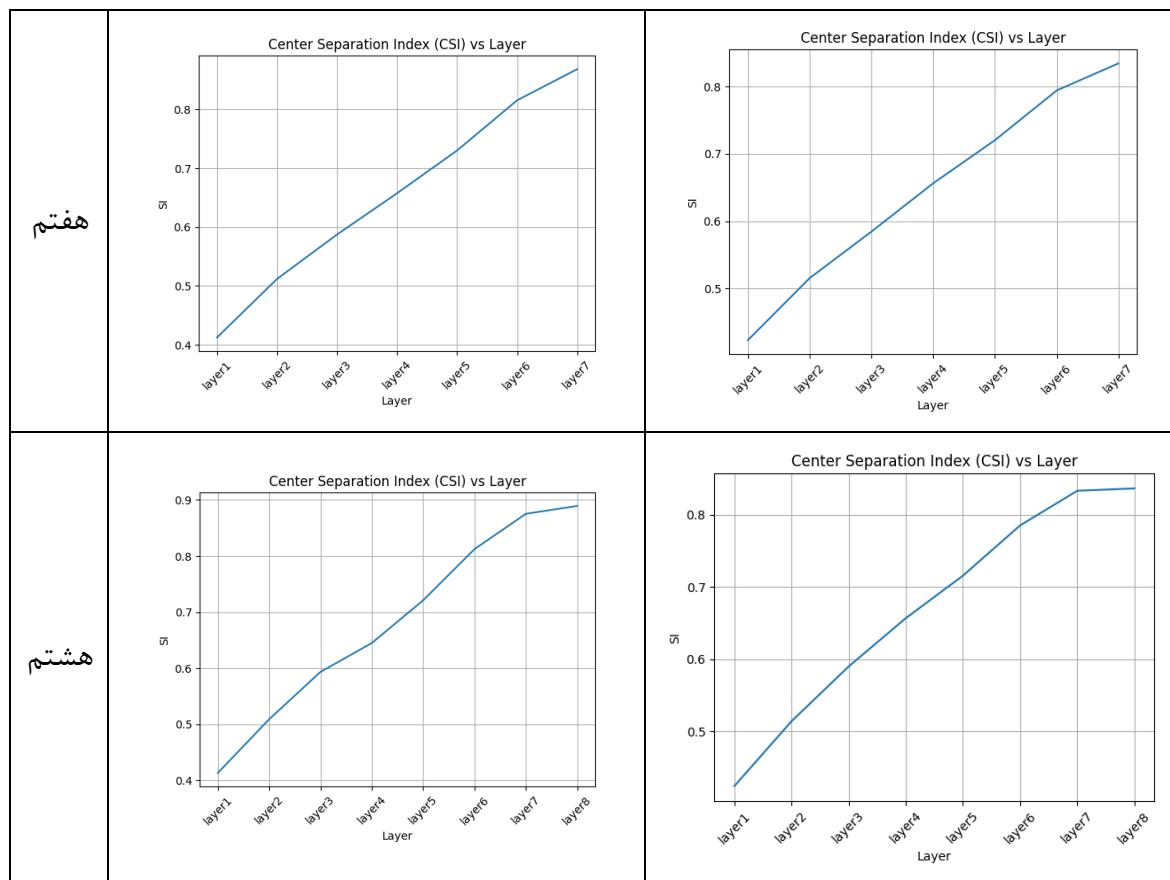
ولی همچنان دقیق در حالت **end to end** بهتر بوده است، باز دلایل بالا علت بهتر بودن آن روش است.

متريک ها روي داده های ترین و تست برای هر لایه: (شکل نمودارها با درنظر گرفتن  $FC$  ها در فایل کد قرار دارد.) (محور y ها CSI است)

جدول ۶- متريک ها روي داده های ترین و تست برای هر لایه







جدول ۷ - مقایسه متریک ها روی داده های ترین برای هر لایه با روش ب

	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
freeze	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35
		0.52	0.52	0.52	0.53	0.53	0.52	0.52
			0.63	0.63	0.63	0.63	0.63	0.63
				0.73	0.73	0.73	0.73	0.73
					0.79	0.79	0.79	0.79
						0.82	0.82	0.82
							0.89	0.89
								0.88
No-freeze	0.39	0.40	0.41	0.40	0.40	0.42	0.41	0.41
		0.50	0.51	0.51	0.51	0.51	0.51	0.51
			0.62	0.59	0.59	0.58	0.59	0.59
				0.68	0.66	0.65	0.66	0.64
					0.74	0.74	0.73	0.72
						0.81	0.82	0.81
							0.87	0.88
								0.89

در ادامه نتایجی که در بالاتر گرفتیم، اینجا نیز میبینیم که وقتی لایه ها را **freeze** نمیکنیم، معیار **CSI** بالاتری داریم که این نشان می دهد که نشان دسته های هر کلاس راحتتر پیدا می شوند.

همچنین **CSI** روی داده های تست هم به عدد بالاتری نسب به روش **freezing** رسیده است. که باز نتیجه ی ما را ثابت می کند.

جدول ۸ - مقایسه متریک ها روی داده های تست برای هر لایه با روش ب

	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
freeze	0.36	0.33	0.36	0.33	0.36	0.36	0.36	0.36
		0.52	0.53	0.52	0.53	0.53	0.53	0.53
			0.64	0.63	0.64	0.64	0.64	0.64
				0.72	0.73	0.73	0.73	0.73
					0.78	0.78	0.78	0.78
						0.79	0.79	0.79
							0.82	0.82
								0.81
No-freeze	0.40	0.40	0.42	0.41	0.41	0.42	0.42	0.42
		0.50	0.52	0.51	0.51	0.51	0.52	0.51
			0.62	0.59	0.59	0.58	0.58	0.59
				0.67	0.66	0.65	0.66	0.66
					0.74	0.72	0.72	0.72
						0.79	0.79	0.78
							0.83	0.83
								0.84

(د)

در این قسمت دیتاهای را بصورت زیر تعریف میکنیم، دو تا **aug** تعریف میکنیم برای اولی همه ی روش های تقویت داده را اعمال میکنیم و برای دومی فقط **transforms.RandomRotation(15)** را اعمال خواهیم کرد.

```
def data_loader(batch_size, num_workers=0, random_seed=42, valid_size=0.1,
shuffle=True, test=False):

    # Define augmentations
    transform_augmented1 = transforms.Compose([
        transforms.RandomHorizontalFlip(),
```

```

        transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.4914, 0.4822, 0.4465],
            std=[0.2470, 0.2435, 0.2616],
        )
    ))
transform_augmented2 = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.4914, 0.4822, 0.4465],
        std=[0.2470, 0.2435, 0.2616],
    )
])
transform_augmented_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.4914, 0.4822, 0.4465],
        std=[0.2470, 0.2435, 0.2616],
    )
])
# Download CIFAR-10 dataset
if test:
    dataset = datasets.CIFAR10(
        root='./data', train=False,
        download=True, transform=transform_augmented_test,
    )
    data_loader = torch.utils.data.DataLoader(
        dataset, batch_size=batch_size, shuffle=shuffle,
    num_workers=num_workers
    )
    return data_loader
# Load the dataset
train_dataset = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_augmented1)
train_dataset2 = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_augmented2)

valid_dataset = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_augmented1)
valid_dataset2 = datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_augmented2)
num_train = len(train_dataset)
indices = list(range(num_train))
split = int(np.floor(valid_size * num_train))

```

```

if shuffle:
    np.random.seed(random_seed)
    np.random.shuffle(indices)
train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)
# Save 10% of training data in a new variable train_loader_SI
train_si_sampler = SubsetRandomSampler(indices[:int(0.1 * num_train)])
train_loader_SI = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, sampler=train_si_sampler,
num_workers=num_workers)
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, sampler=train_sampler,
num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(
    valid_dataset, batch_size=batch_size, sampler=valid_sampler,
num_workers=num_workers)
# Save 10% of training data in a new variable train_loader_SI2
train_si_sampler2 = SubsetRandomSampler(indices[:int(0.1 * num_train)])
train_loader_SI2 = torch.utils.data.DataLoader(
    train_dataset2, batch_size=batch_size, sampler=train_si_sampler2,
num_workers=num_workers)
train_loader2 = torch.utils.data.DataLoader(
    train_dataset2, batch_size=batch_size, sampler=train_sampler,
num_workers=num_workers)
valid_loader2 = torch.utils.data.DataLoader(
    valid_dataset2, batch_size=batch_size, sampler=valid_sampler,
num_workers=num_workers)

return (train_loader, valid_loader, train_loader_SI, train_loader2,
valid_loader2, train_loader_SI2)
train_loader, valid_loader, train_loader_SI, train_loader2, valid_loader2,
train_loader_SI2 = data_loader(batch_size=128, num_workers=1)
test_loader = data_loader(batch_size=128, test=True, num_workers=1)

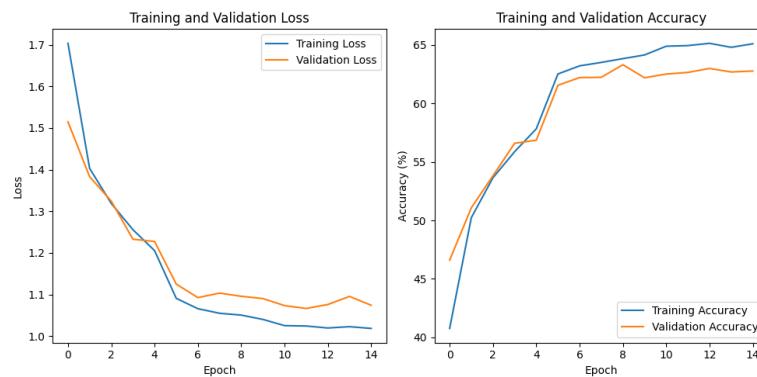
```

همانند قسمت ج لایه ها را تعریف میکنیم فقط برای آموزش لایه ها برای ۴ مرحله ی اول از استفاده می کنیم که داده ها با تمام روش های تقویت داده را دارد و در ۴ مرحله بعد از استفاده میکنیم که تنها از یک روش تقویت داده استفاده شده است. که در بالاتر نوع train\_loader2 آن را ذکر کرده ایم.

لایه ها را تک تک آموزش می دهیم و به نتایج زیر میرسیم:

نمودارهای accuracy و loss مدل برای هر لایه مطابق شکل زیر است:

لایه اول:



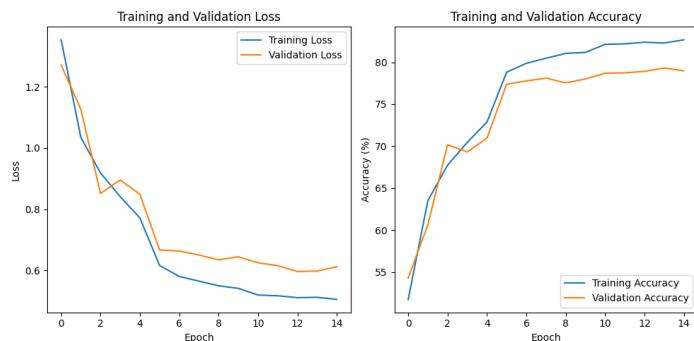
شکل ۲۱ - نمودارهای accuracy و loss لایه اول

لایه دوم:



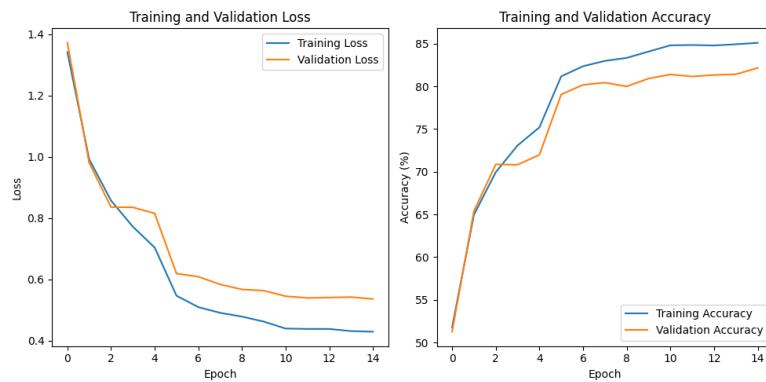
شکل ۲۲ - نمودارهای accuracy و loss لایه دوم

لایه سوم:



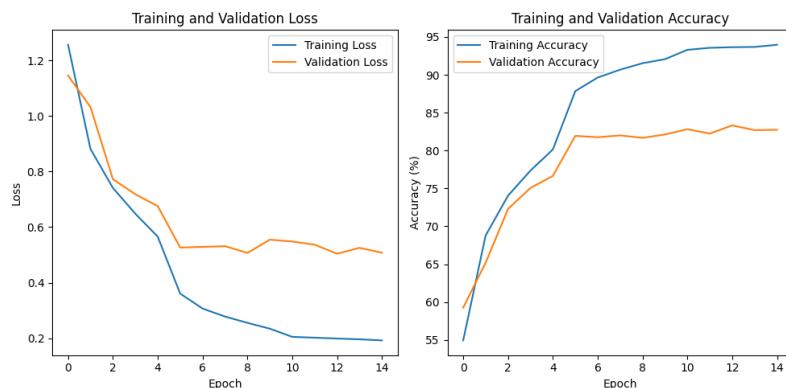
شکل ۲۳ - نمودارهای accuracy و loss لایه سوم

لایه چهارم:



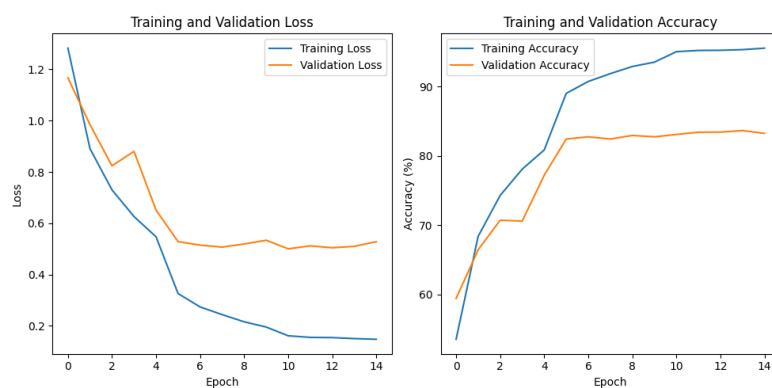
شکل ۲۴ - نمودارهای accuracy و loss لایه چهارم

لایه پنجم:



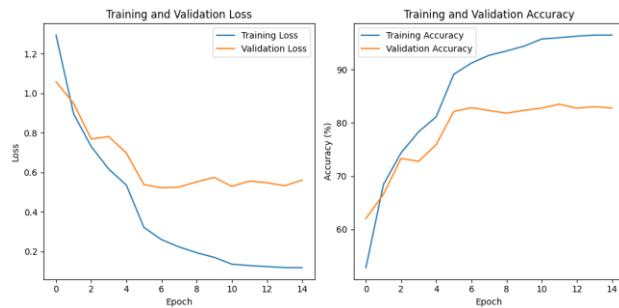
شکل ۲۵ - نمودارهای accuracy و loss لایه پنجم

لایه ششم:



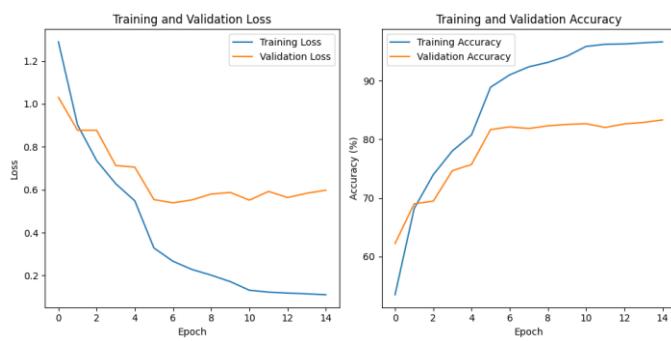
شکل ۲۶ - نمودارهای accuracy و loss لایه ششم

لایه هفتم:



شکل ۲۷ - نمودارهای accuracy و loss لایه هفتم

لایه هشتم:



شکل ۲۸ - نمودارهای accuracy و loss لایه هشتم

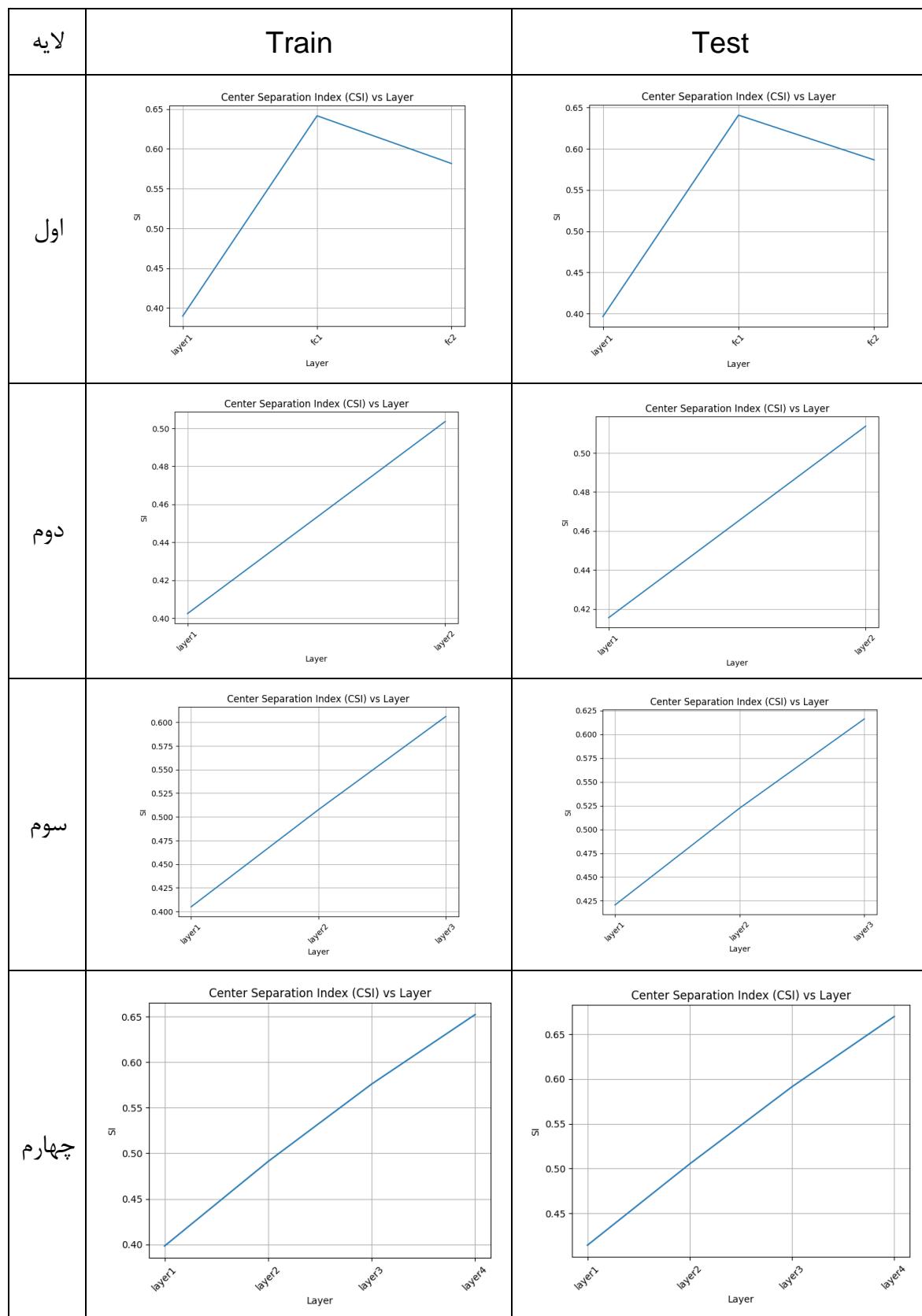
جدول ۹ - دقت روی داده های تست در هر لایه

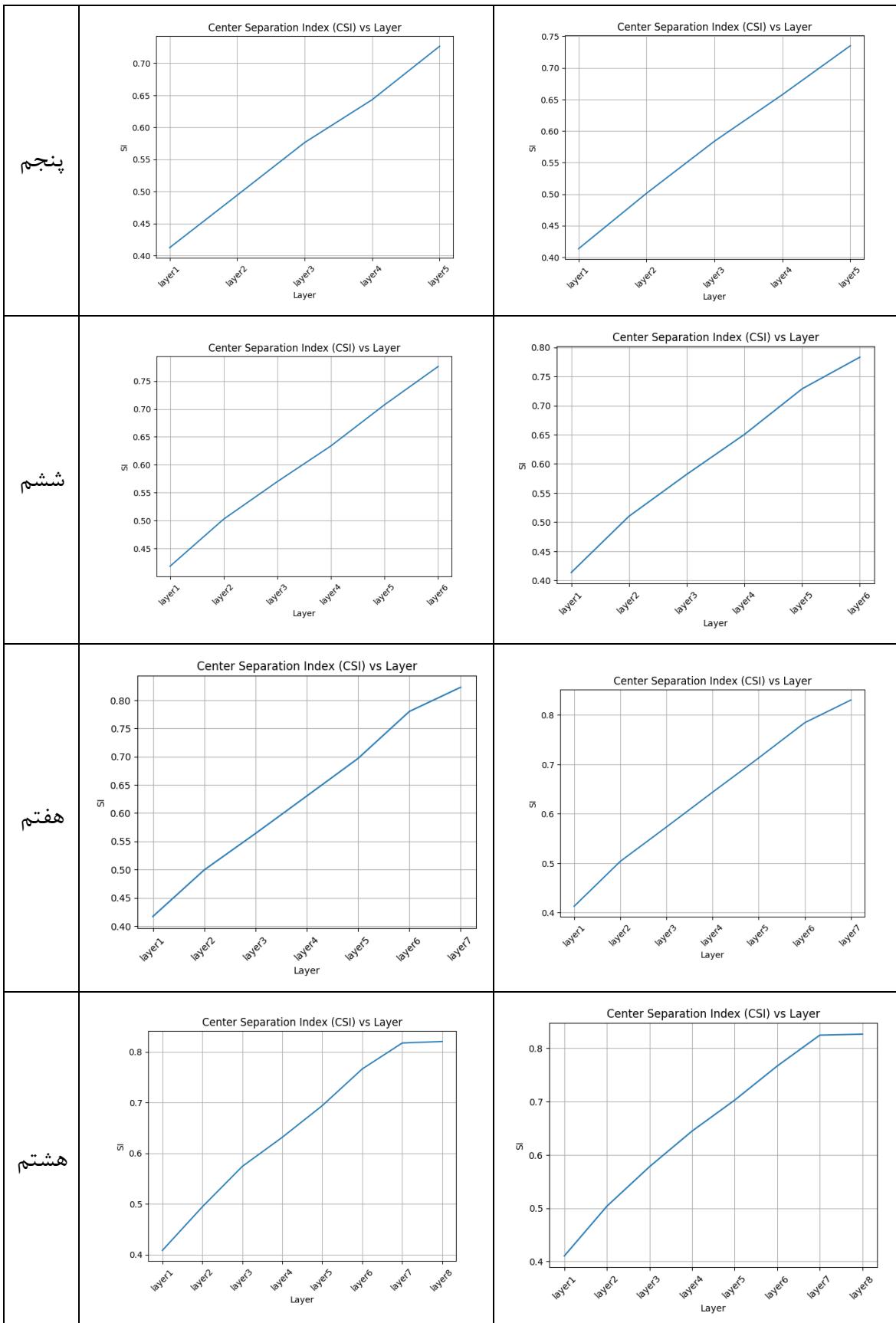
شماره لایه	اول	دوم	سوم	چهارم	پنجم	ششم	هفتم	هشتم
دقت تست	۶۵,۳۹	۷۴,۶۹	۸۰,۴۰	۸۲,۸۱	۸۳,۹۲	۸۴,۹۲	۸۴,۰۷	۸۴,۰۲

مشاهده میکنیم که در هر لایه دقت روی داده های تست روند افزایشی داشته است و به عدد مورد نظر ما دارد نزدیک می شود، و لاس نیز کاهشی بوده است.

متريک ها روی داده های ترين و تست برای هر لایه: (شکل نمودارها با درنظر گرفتن  $C$ ها در فایل کد قرار دارد.) (محور  $y$ ها CSI است)

جدول ۱۰- متریک ها روی داده های ترین و تست برای هر لایه





جدول ۱۱ - متريک ها روی داده های ترين برای هر لایه

	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
layer 1	0.36	0.40	0.40	0.40	0.41	0.42	0.42	0.41
layer 2		0.50	0.51	0.49	0.49	0.50	0.50	0.49
layer 3			0.61	0.58	0.58	0.57	0.56	0.57
layer 4				0.65	0.64	0.63	0.63	0.63
layer 5					0.73	0.71	0.70	0.69
layer 6						0.78	0.78	0.77
layer 7							0.82	0.82
layer 8								0.82

مشاهده می کنیم در این حالت بر روی داده های آموزش و تست به **CSI** نزدیک حال الف رسیدیم ولی مقدار از آن کمتر است. علت آن ممکن است به این دلیل باشد: انتخاب استراتژی های **aug** می تواند عملکرد مدل را به طور قابل توجهی تحت تاثیر قرار دهد. استفاده از مجموعه متنوعی از تقویت ها برای همه مراحل در روش الف ممکن است به تعمیم بهتر مدل کمک کند، در حالی که کاهش **aug** در مراحل بعدی ممکن است توانایی مدل را برای یادگیری ویژگی های قوی محدود کند.

اگر مجموعه داده پیچیده است و الگوهای متنوعی را نشان می دهد، استفاده از مجموعه ای غنی از افزایش ها برای همه مراحل (روش الف) ممکن است سودمند باشد. از سوی دیگر، اگر مراحل بعدی بر روی ویژگی های خاص تری تمرکز کنند که نیازی به افزایش گسترده ندارند، این روش ممکن است موثر باشد.

جدول ۱۲ - متريک ها روی داده های تست برای هر لایه

	step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
layer 1	0.40	0.42	0.42	0.41	0.41	0.41	0.41	0.41
layer 2		0.51	0.52	0.51	0.50	0.51	0.50	0.50
layer 3			0.62	0.59	0.58	0.58	0.57	0.58
layer 4				0.67	0.66	0.65	0.64	0.64
layer 5					0.74	0.73	0.71	0.70
layer 6						0.78	0.78	0.77
layer 7							0.83	0.82
layer 8								0.83

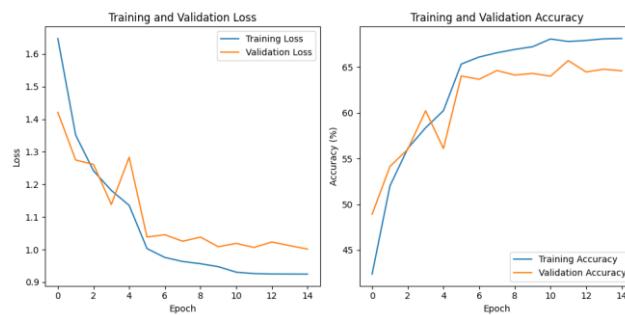
(و)

در این قسمت دیتاهای را بصورت زیر تعریف میکنیم، دو تا `aug` تعریف میکنیم برای دومی همه ی روش های تقویت داده را اعمال میکنیم و برای اولی فقط `transforms.RandomRotation(15)` را اعمال خواهیم کرد. (کد تعریف داده در این قسمت همانند قسمت قبل است، فقط باید نامگذاری ها را تغییر دهیم.)

همانند قسمت ج لایه ها را تعریف میکنیم فقط برای آموزش لایه ها برای ۴ مرحله ی اول از استفاده می کنیم که داده ها با یک روش های تقویت داده را دارد و در ۴ مرحله بعد از استفاده میکنیم که از همه روش تقویت داده استفاده شده است. که در بالاتر نوع آن را ذکر کرده ایم.

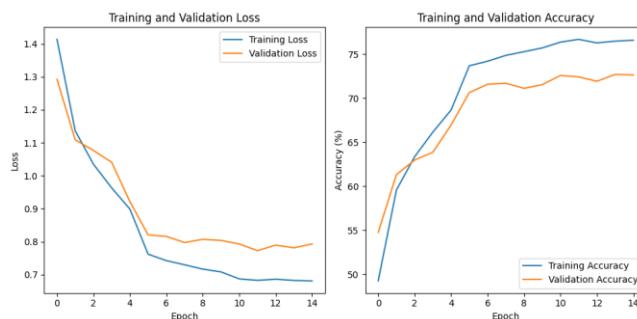
لایه ها را تک تک آموزش می دهیم و به نتایج زیر میرسیم: نمودارهای `loss` و `accuracy` مدل برای هر لایه مطابق شکل زیر است:

لایه اول:



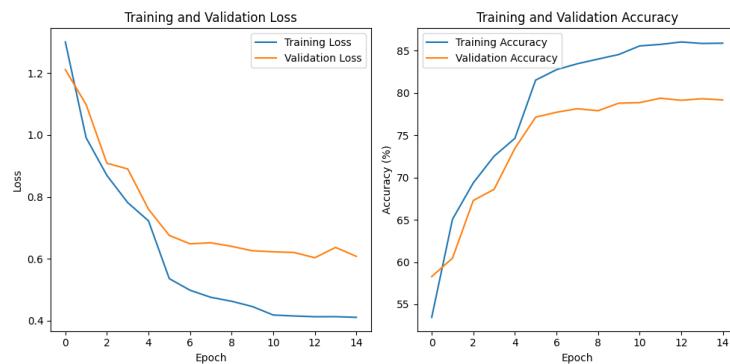
شکل ۱ - ۲۹

لایه دوم:



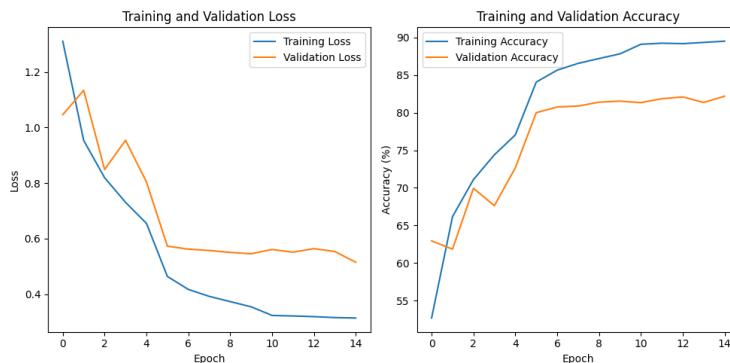
شکل ۲ - ۳۰

لایه سوم:



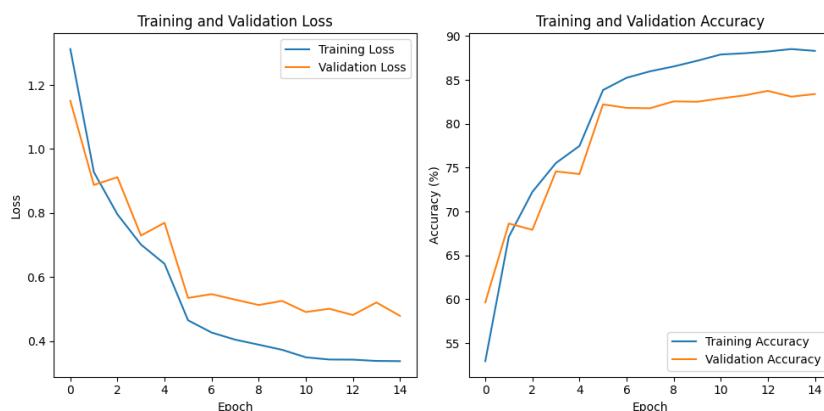
شکل ۳ - ۳۱

لایه چهارم:



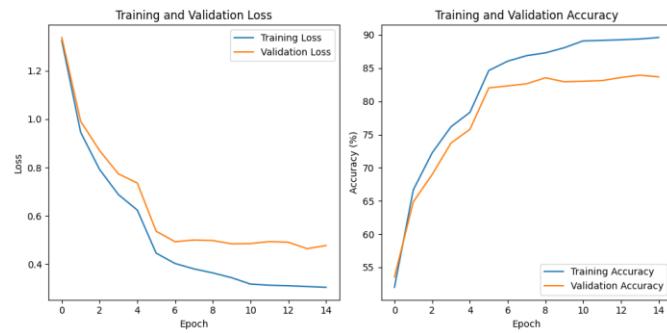
شکل ۴ - ۳۲

لایه پنجم:



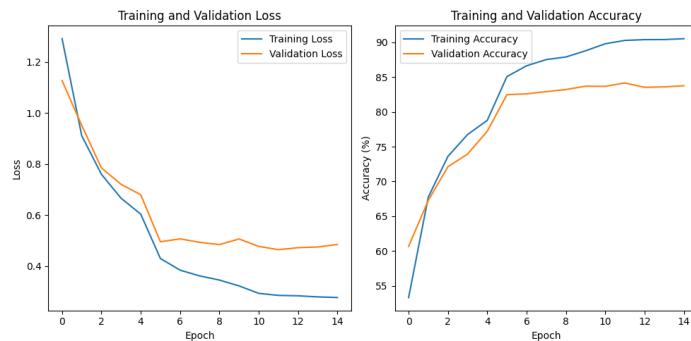
شکل ۵ - ۳۳

لایه ششم:



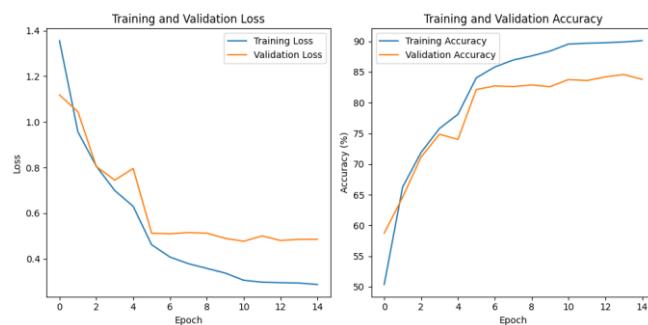
شکل ٦ - ٣٤

لایه هفتم:



شکل ٦ - ٣٥

لایه هشتم:



شکل ٦ - ٣٦

جدول ۱۳ - دقیق روش داده های تست در هر لایه

شماره لایه	اول	دوم	سوم	چهارم	پنجم	ششم	هفتم	هشتم
All then one	۶۵,۳۹	۷۴,۶۹	۸۰,۴۰	۸۲,۸۱	۸۳,۹۲	۸۴,۹۲	۸۴,۰۷	۸۴,۰۲
One then all	۶۶,۶۲	۷۴,۸۳	۸۰,۱۵	۸۲,۶۶	۸۴,۲۶	۸۵,۴۰	۸۵,۵۰	۸۴,۶۷

مشاهده میکنیم که در هر لایه دقیق روش داده های تست روند افزایشی داشته است و به عدد مورد نظر ما دارد نزدیک می شود، و لاس نیز کاهشی بوده است.

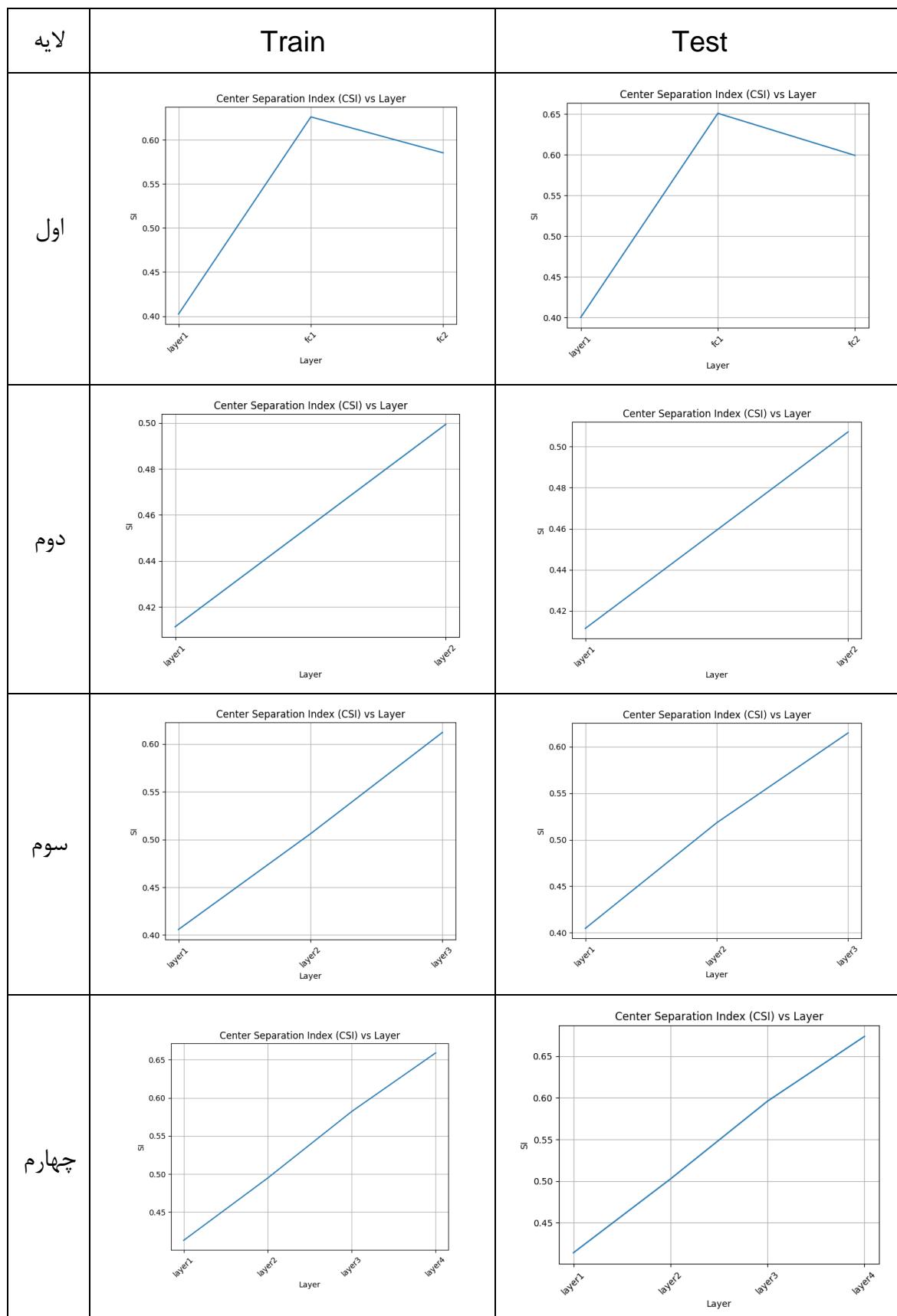
در این قسمت مشاهده میکنیم که در حالت دوم که ابتدا از یک روش **aug** استفاده کنیم و بعد از تمام روش ها استفاده کنیم، نتایج بهتری گرفتیم. علت این است که: در حالت دوم، چهار مرحله اول ممکن است بر یادگیری ویژگی های اساسی تمرکز کند و استفاده از یک تقویت واحد می تواند کافی باشد. مراحل بعدی ممکن است از تقویت های متنوع تری برای ثبت الگوهای پیچیده بهره مند شوند و روش دوم به مدل اجازه می دهد تا ابتدا بر یادگیری ویژگی ها و الگوهای اصلی تمرکز کند، قبل از اینکه در معرض تغییرات پیچیده تر قرار گیرد. این می تواند منجر به یک فرآیند یادگیری قوی تر شود.

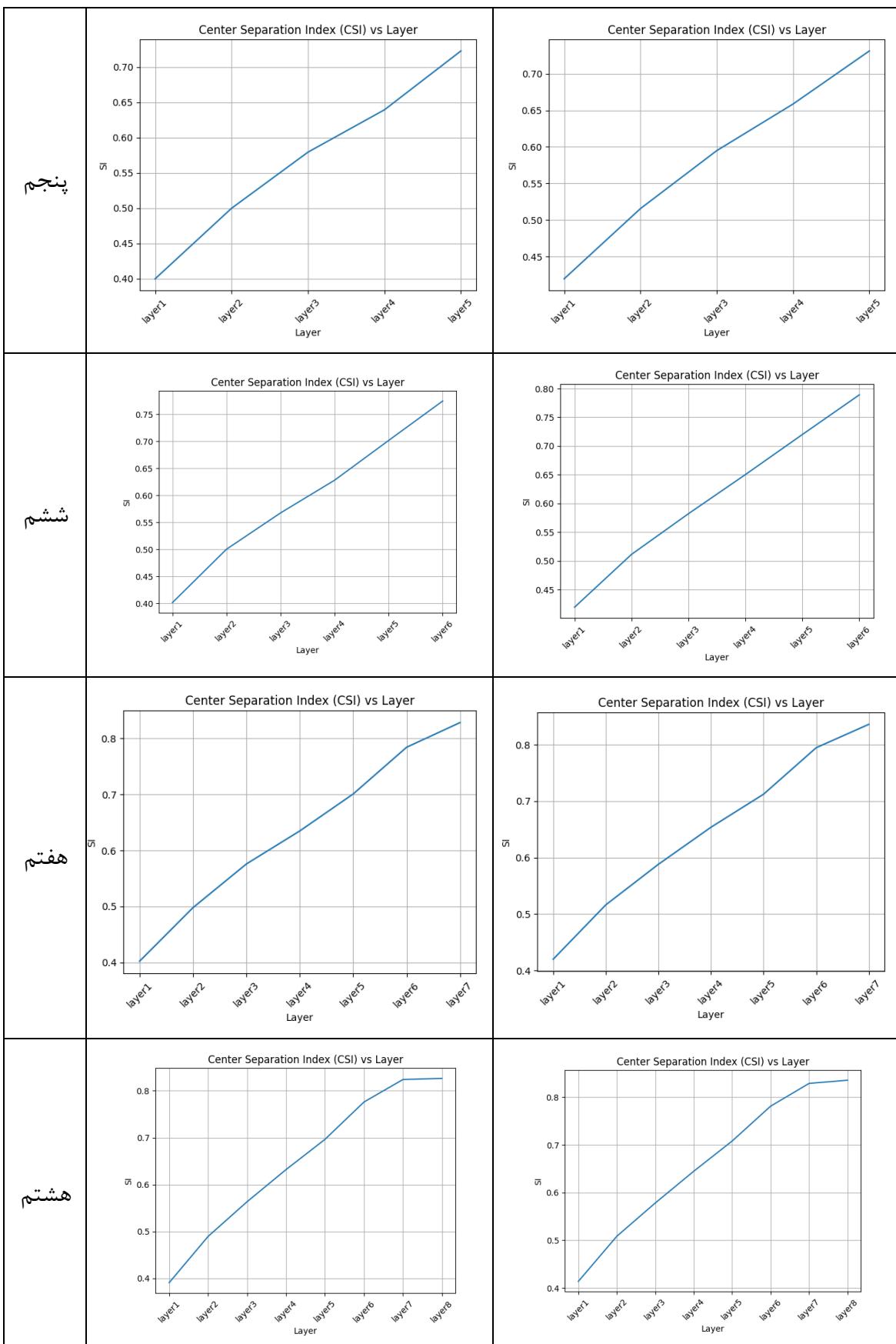
با شروع با یک **aug** احتمال کمتری وجود دارد که مدل در اوایل فرآیند آموزش به طیف وسیعی از تغییرات داده ها اضافه شود. این می تواند مانع **Overfitting** باشد. معرفی همه تقویت ها در مراحل بعدی به مدل اجازه می دهد تا قابلیت های استخراج ویژگی های خود را اصلاح کند و به آن کمک می کند تا طیف وسیع تری از تغییرات داده را تشخیص دهد و به آن پاسخ دهد.

افزایش تدریجی پیچیدگی داده ها می تواند مدل را برای سناریوهای مختلف دنیای واقعی سازگارتر و قوی تر کند، به خصوص اگر تقویت های مرحله بعدی طیف گسترده ای از تغییرات ممکن را پوشش دهند.

متريک ها روش داده های ترين و تست برای هر لایه: (شکل نمودارها با درنظر گرفتن **fc** ها در فایل کد قرار دارد.) (محور yها CSI است)

جدول ۱۴- متریک ها روی داده های ترین و تست برای هر لایه





جدول ۱۵ - متریک ها روی داده های train برای هر لایه

		step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
All then one	layer 1	0.36	0.40	0.40	0.40	0.41	0.42	0.42	0.41
	layer 2		0.50	0.51	0.49	0.49	0.50	0.50	0.49
	layer 3			0.61	0.58	0.58	0.57	0.56	0.57
	layer 4				0.65	0.64	0.63	0.63	0.63
	layer 5					0.73	0.71	0.70	0.69
	layer 6						0.78	0.78	0.77
	layer 7							0.82	0.82
	layer 8								0.82
One then all	layer 1	0.40	0.41	0.41	0.41	0.40	0.40	0.40	0.39
	layer 2		0.50	0.51	0.49	0.50	0.50	0.50	0.49
	layer 3			0.61	0.58	0.58	0.57	0.58	0.56
	layer 4				0.66	0.64	0.63	0.63	0.63
	layer 5					0.72	0.70	0.70	0.70
	layer 6						0.77	0.78	0.78
	layer 7							0.83	0.82
	layer 8								0.83

جدول ۱۶ - متریک ها روی داده های تست برای هر لایه

		step 1	step 2	step 3	step 4	step 5	step 6	step 7	step 8
All then one	layer 1	0.40	0.42	0.42	0.41	0.41	0.41	0.41	0.41
	layer 2		0.51	0.52	0.51	0.50	0.51	0.50	0.50
	layer 3			0.62	0.59	0.58	0.58	0.57	0.58
	layer 4				0.67	0.66	0.65	0.64	0.64
	layer 5					0.74	0.73	0.71	0.70
	layer 6						0.78	0.78	0.77
	layer 7							0.83	0.82
	layer 8								0.83
One then all	layer 1	0.40	0.41	0.40	0.41	0.42	0.42	0.42	0.41
	layer 2		0.51	0.52	0.50	0.52	0.51	0.52	0.51
	layer 3			0.61	0.60	0.59	0.58	0.59	0.58
	layer 4				0.67	0.66	0.65	0.65	0.64
	layer 5					0.73	0.72	0.71	0.71
	layer 6						0.79	0.79	0.78
	layer 7							0.84	0.83
	layer 8								0.84

در ادامه نتایجی که در بالاتر گرفتیم، اینجا نیز میبینیم که وقتی ابتدا از یک روش **aug** استفاده کنیم، معیار **CSI** بالاتری داریم که این نشان می دهد که نشان دسته های هر کلاس راحتتر پیدا می شوند. همچنین **CSI** روی داده های تست هم به عدد بالاتری نسبت به روشی که ابتدا از تمامی روش های **aug** استفاده کنیم رسیده است. که باز نتیجه بی که بالاتر گرفتیم را ثابت می کند. ولی همچنان روشی که در قسمت الف رفتیم دقیق‌تری داشت و در مرحله‌ی مدل قسمت (و) بیشترین دقیق‌تری داشته است.

## (ب) شبکه بخش بندی تصاویر دو بعدی

### ساز و کار معماری UNET

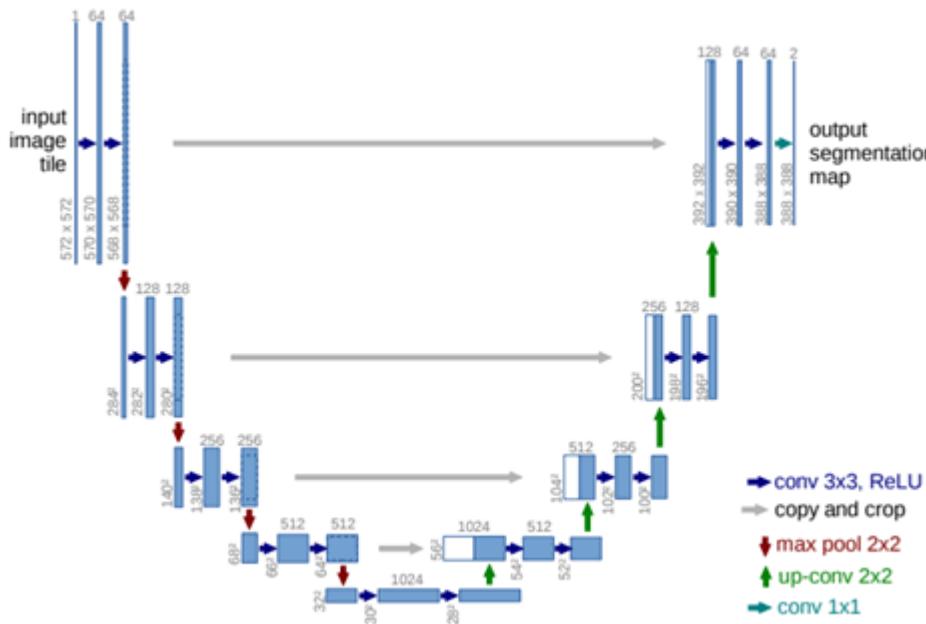
مقاله U-Net یک معماری شبکه عصبی جدید را برای وظیفه تقسیم بندی تصویر، به ویژه در زمینه تصویربرداری پزشکی معرفی کرد. در اینجا یک نمای کلی از معماری U-Net، عملکرد آن و اهمیت آن در وظایف تقسیم بندی آورده شده است.

#### معماری U-Net

ساختار کلی : معماری U-Net شبیه حرف "U" است که به آن نام می دهد. دارای یک مسیر down و یک مسیر Up Sample است که یک ساختار متقارن را تشکیل می دهد.

- مسیر down sample از معماری معمولی یک شبکه کانولوشن پیروی می کند. این شامل استفاده مکرر از دو convolution  $3 \times 3$  (بدون پد) است که هر کدام توسط ReLU و یک Max Pooling با گام ۲ انجام می شود. و در هر مرحله، تعداد کانال های ویژگی دو برابر می شود.
- در ادامه، هر مرحله شامل یک نمونه برداری از نقشه ویژگی و به دنبال آن یک Up convolution  $2 \times 2$  است که تعداد کانال های ویژگی را به نصف کاهش می دهد. به دنبال آن یک الحاق با نقشه مشخصه برش داده شده مربوطه از مسیر، و دو convolution  $3 \times 3$  هر یک توسط یک ReLU دنبال می شود.
- لایه نهایی: لایه نهایی یک convolution  $1 \times 1$  است که هر بردار ویژگی ۶۴ جزء را به تعداد مورد نظر کلاس نگاشت می کند.

شکل زیر معماری UNET را نشان میدهد.



## شکل ۳۷: معماری UNET

## U-Net چگونه کار می کند؟

نواوری کلیدی در U-Net روشی است که ویژگی‌های با وضوح بالا را از مسیر فشردگی با خروجی نمونه‌سازی شده ترکیب می‌کند، که به شبکه اجازه می‌دهد اطلاعات زمینه را به لایه‌های با وضوح بالاتر منتشر کند.

## سودمندی برای وظیفه تقسیم بندی

- localization دقیق: با توجه به معماری که نقشه های ویژگی های سطح پایین را با نقشه های سطح بالاتر ترکیب می کند، U-Net می تواند مناطق مورد علاقه (Region of Interest) در تصاویر را با دقت بسیار دقیق بومی سازی کند، که در تصویربرداری پزشکی بسیار مهم است (به عنوان مثال، شناسایی تومورها)، ضایعات یا سایر آسیب شناسی ها).
  - یادگیری کارآمد: U-Net به نمونه های آموزشی کمتری نیاز دارد اما همچنان قادر به دستیابی به دقت بالایی است.
  - قابلیت کاربرد در اندازه های مختلف: معماری می تواند با تصاویر در اندازه های مختلف کار کند، که در کاربردهای پزشکی که در آن تصاویر می توانند اندازه متفاوتی داشته باشند، تطبیق پذیری را اضافه می کند.

به طور خلاصه، معماری U-Net، که با طراحی منحصر به فرد U شکل آن با ترکیب مسیرهای down/up مشخص می شود، در تقسیم بندی تصاویر به خصوص پزشکی عالی است.<sup>۱</sup> Sample

## ساز و کار معماری PSPNET

توسط ژائو و همکاران، معماری PSPNet را ارائه شد، در اینجا مروری بر معماری PSPNet، مکانیسم آن و اهمیت آن است:

### معماری PSPNet

- شبکه کانولوشن پایه: •  
PSPNet از یک شبکه از پیش آموزش دیده مانند ResNet یا Mobilenet backbone خود استفاده می کند. این شبکه پایه مسئول استخراج ویژگی های اولیه است.

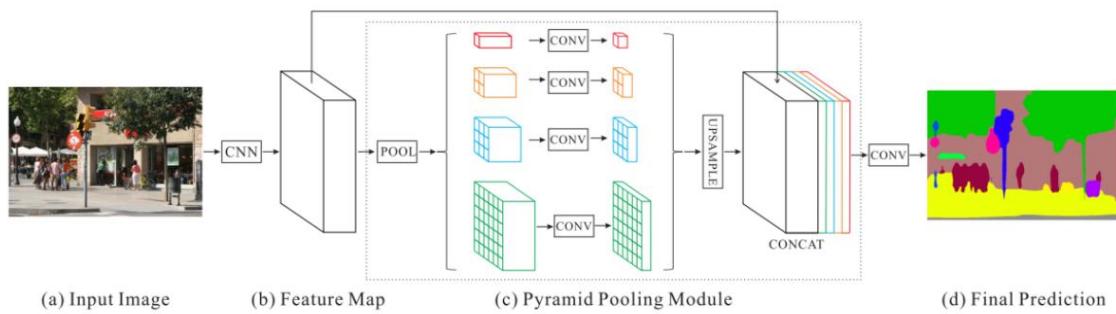
- ماژول ادغام هرمی: •  
هسته PSPNet ماژول ادغام هرمی آن است. این ماژول اطلاعات زمینه را با ادغام نقشه ویژگی در مقیاس های مختلف جمع می کند. نقشه ویژگی در مقیاس های شبکه ای مختلف ( مثلًا ۱\*۱، ۲\*۲ ، .... ) ادغام می شود و نقشه های مشخصه ای از نمایش های مختلف ( sub region ) را تولید می کند. این ویژگی های تلفیقی سپس به اندازه نقشه ویژگی اصلی نمونه برداری می شوند و با آن الحق می شوند که منجر به ادغام ویژگی ها در مقیاس های مختلف می شود.

- نقشه تقسیم بندی نهایی: •  
پس از الحق ویژگی ها از مقیاس های مختلف، از یک لایه کانولوشن برای به دست آوردن طبقه بندی نهایی هر پیکسل ( نقشه تقسیم بندی ) استفاده می شود.

شکل زیر معماری PSPNet را نشان میدهد.

---

<sup>1</sup> Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." *Medical Image Computing and Computer-Assisted Intervention—MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer International Publishing, 2015.



شکل ۳۸: معماری PSPNet

## PSPNet چگونه کار می کند؟

تجمیع: مازول ادغام هر می به طور موثر اطلاعات زمینه را از مناطق مختلف جمع می کند و یک چشم انداز جهانی به شبکه ارائه می دهد. این به تجزیه دقیق صحنه کمک می کند، به خصوص برای صحنه های پیچیده که در آن اشیاء دارای اندازه ها و ظاهر متفاوت هستند.

Concatenation: PSPNet و Upsampling اصلی، اطلاعات کلی و محلی را ترکیب می کند که منجر به تقسیم‌بندی دقیق‌تر و دقیق‌تر می شود.

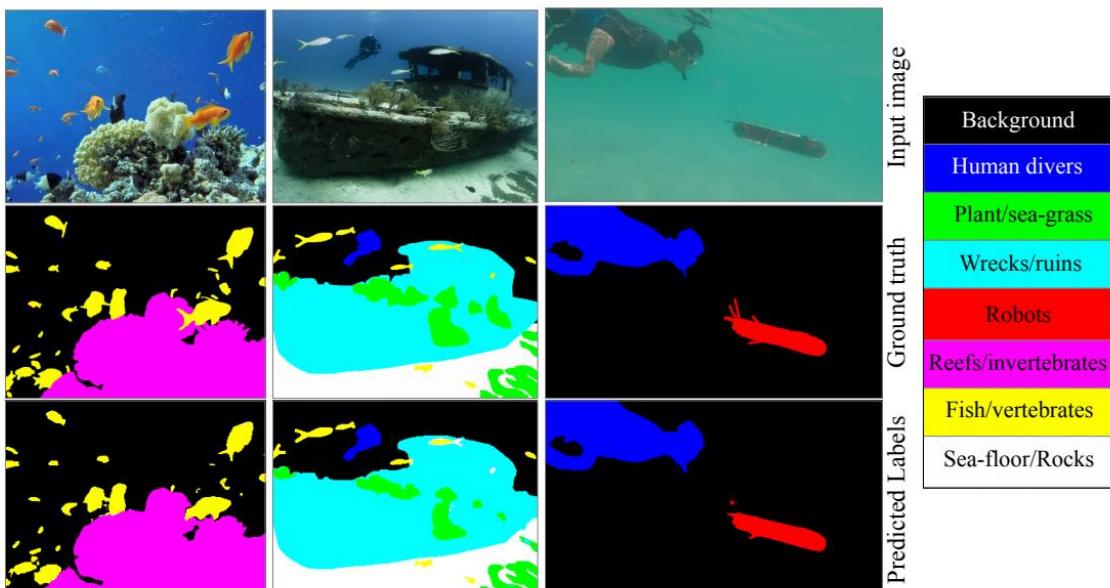
استخراج ویژگی عمیق: استفاده از یک شبکه کانولوشن عمیق به عنوان Backbone (مانند ResNet) به PSPNet اجازه می دهد تا ویژگی های سلسله مراتبی غنی را استخراج کند، که برای درک صحنه های پیچیده بسیار مهم هستند.<sup>۱</sup>

<sup>1</sup> Zhao, Hengshuang, et al. "Pyramid scene parsing network." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017

## پیش پردازش و آماده سازی دیتاست

### معرفی دیتاست

مجموعه داده SUIM<sup>1</sup> و همکاران در سال ۲۰۲۰ معرفی شده است که دارای کلاس هایی به شرح زیر میباشد. تصویر از مقاله SUIM میباشد.



شکل ۳۹: نمونه تصاویر دیتاست SUIM و کلاس های آن

در این مقاله این دیتاست بر روی مدل های مختلف آموزش داده شده و سپس تست گرفته شده است که مدلی که خود نویسنندگان مقاله ارائه دادند بهترین عملکرد را داشته است.

<sup>1</sup> Islam, Md Jahidul, et al. "Semantic segmentation of underwater imagery: Dataset and benchmark." 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2020.

	Model	HD	WR	RO	RI	FV	Combined	Saliency Pred.
F	FCN8 <sub>CNN</sub>	76.34 ± 2.24	70.24 ± 2.26	39.83 ± 3.87	61.65 ± 2.36	76.24 ± 1.87	64.86 ± 2.52	75.62 ± 1.79
	FCN8 <sub>VGG</sub>	<b>89.10 ± 1.50</b>	<b>82.03 ± 1.94</b>	<b>74.01 ± 3.23</b>	<b>79.19 ± 2.27</b>	<b>90.46 ± 1.18</b>	<b>82.96 ± 2.02</b>	<b>89.63 ± 1.24</b>
	SegNet <sub>CNN</sub>	59.60 ± 2.02	41.60 ± 1.65	31.77 ± 3.03	41.88 ± 2.66	60.08 ± 1.91	46.97 ± 2.25	56.96 ± 1.58
	SegNet <sub>ResNet</sub>	80.52 ± 3.26	77.65 ± 3.15	62.45 ± 3.90	<b>82.30 ± 1.96</b>	<b>91.47 ± 1.01</b>	76.88 ± 2.66	<b>86.88 ± 1.83</b>
	UNet <sub>GRAY</sub>	85.47 ± 2.21	<b>79.77 ± 2.01</b>	60.95 ± 3.31	69.95 ± 2.57	84.47 ± 1.39	75.12 ± 2.30	83.96 ± 1.40
	UNet <sub>RGB</sub>	<b>89.60 ± 1.84</b>	<b>86.17 ± 1.73</b>	68.87 ± 3.30	<b>79.24 ± 2.70</b>	<b>91.35 ± 1.14</b>	<b>83.05 ± 2.14</b>	<b>89.99 ± 1.29</b>
	PSPNet <sub>MobileNet</sub>	80.21 ± 1.19	70.94 ± 1.61	72.04 ± 2.21	72.65 ± 1.62	79.19 ± 1.74	76.01 ± 1.67	78.42 ± 1.59
	DeepLab <sub>V3</sub>	<b>89.68 ± 2.09</b>	77.73 ± 2.18	<b>72.72 ± 3.35</b>	78.28 ± 2.70	87.95 ± 1.59	<b>81.27 ± 2.30</b>	85.94 ± 1.72
	SUIM-Net <sub>RSB</sub>	89.04 ± 1.31	65.37 ± 2.22	<b>74.18 ± 2.11</b>	71.92 ± 1.80	84.36 ± 1.37	78.86 ± 1.79	81.36 ± 1.72
	SUIM-Net <sub>VGG</sub>	<b>93.56 ± 0.98</b>	<b>86.02 ± 1.03</b>	<b>78.06 ± 1.50</b>	<b>83.49 ± 1.39</b>	<b>93.73 ± 0.87</b>	<b>86.97 ± 1.15</b>	<b>91.91 ± 0.85</b>
mIoU (→)	FCN8 <sub>CNN</sub>	67.27 ± 2.50	81.64 ± 2.16	36.44 ± 3.67	78.72 ± 2.50	70.25 ± 2.28	66.86 ± 2.62	75.63 ± 1.89
	FCN8 <sub>VGG</sub>	79.86 ± 1.50	<b>85.77 ± 2.09</b>	65.05 ± 3.00	<b>85.23 ± 2.07</b>	<b>81.18 ± 1.46</b>	<b>79.42 ± 2.02</b>	<b>85.22 ± 1.24</b>
	SegNet <sub>CNN</sub>	62.76 ± 2.35	66.75 ± 2.57	36.63 ± 3.12	63.46 ± 3.18	62.48 ± 2.32	58.42 ± 2.71	65.90 ± 2.12
	SegNet <sub>ResNet</sub>	74.00 ± 2.88	82.68 ± 2.94	58.63 ± 3.61	<b>89.61 ± 1.15</b>	<b>82.96 ± 1.38</b>	77.58 ± 2.39	83.09 ± 1.96
	UNet <sub>GRAY</sub>	78.33 ± 2.34	85.14 ± 2.14	57.25 ± 3.00	79.96 ± 2.55	78.00 ± 1.90	75.74 ± 2.38	82.77 ± 1.59
	UNet <sub>RGB</sub>	<b>81.17 ± 2.02</b>	<b>87.54 ± 2.00</b>	62.07 ± 3.12	83.69 ± 2.58	<b>83.83 ± 1.47</b>	<b>79.66 ± 2.24</b>	<b>85.85 ± 1.54</b>
	PSPNet <sub>MobileNet</sub>	75.76 ± 1.47	<b>86.82 ± 1.26</b>	<b>72.66 ± 1.47</b>	<b>85.16 ± 1.65</b>	74.67 ± 1.90	77.41 ± 1.56	80.87 ± 1.56
	DeepLab <sub>V3</sub>	<b>80.78 ± 2.07</b>	85.17 ± 2.08	<b>66.03 ± 3.16</b>	83.96 ± 2.52	79.62 ± 1.85	<b>79.10 ± 2.34</b>	<b>83.55 ± 1.65</b>
	SUIM-Net <sub>RSB</sub>	<b>81.12 ± 1.76</b>	80.68 ± 1.74	<b>65.79 ± 2.10</b>	84.90 ± 1.77	76.81 ± 1.82	77.77 ± 1.64	80.86 ± 1.64
	SUIM-Net <sub>VGG</sub>	<b>85.09 ± 1.45</b>	<b>89.90 ± 1.29</b>	<b>72.49 ± 1.61</b>	<b>89.51 ± 1.25</b>	<b>83.78 ± 1.55</b>	<b>84.14 ± 1.43</b>	<b>87.67 ± 1.24</b>

شکل ۴۰: مقایسه مدل های مختلف بر روی دیتاست **SUIM**

نکته حائز اهمیت در جدول فوق این است که با توجه به کد گیت مقاله اصلاً داده **val** جدا نشده است و اینکه آموزش تنها بر کلاس که اتفاقاً کلاس ها بی ربط تری به هم هستن صورت گرفته است و همچنین **size** تصاویر بر روی ۳۰۰ میباشد نه ۱۶۰ که کم کردن سایز هم میتواند افت قابل توجهی در دقت ها داشته باشد و نباید انتظار داشت که خروجی ما نیز به همین دقت های بالا بر روی ۸ کلاس برسد.

## دانلود دیتاست و تقسیم بندی به آموزش و ارزیابی

به کمک **gdown** دیتاست مربوطه را دانلود میکنیم. سپس آن ها را **unzip** کرده و فایل های اضافی را پاک میکنیم.

```
!mkdir ./dataset
cd dataset
# info
id1 = "1cqfcs-7XXhh_kqIUPEd-1Y6I9COUsjC"
gdown.download(id=id1, quiet=False)

# Train-val zip
id2 = "1YwjUODQWwQ3_vKSytqVdF4recqBOEe72"
gdown.download(id=id2, quiet=False)

# test zip
id3 = "1diN3tNe2nR1eV3Px4gqlp6wp3XuLBwDy"
gdown.download(id=id3, quiet=False)

# unzip dataset
from glob import glob
import zipfile
```

```

files = glob('*.*zip')

for file in files:
    print('Unzipping:', file)

    with zipfile.ZipFile(file, 'r') as zip_ref:
        zip_ref.extractall('..')

!rm -rf train_val.zip
!rm -rf TEST.zip

cd ..

```

در ادامه با توجه به خواسته مساله به کمک کد زیر ۱۰ درصد داده های train-val را جدا میکنیم

```

# split random val and train
import os
import shutil
import random

# Define the path to the dataset
dataset_base_path = './dataset'
images_path = os.path.join(dataset_base_path, 'train_val/images')
masks_path = os.path.join(dataset_base_path, 'train_val/masks')

# Define the path for the train and val directories
train_images_path = os.path.join(dataset_base_path, 'train/images')
train_masks_path = os.path.join(dataset_base_path, 'train/masks')
val_images_path = os.path.join(dataset_base_path, 'val/images')
val_masks_path = os.path.join(dataset_base_path, 'val/masks')

# Create directories if they don't exist
for path in [train_images_path, train_masks_path, val_images_path,
val_masks_path]:
    os.makedirs(path, exist_ok=True)

# Get the list of image files
image_files = [f for f in os.listdir(images_path) if f.endswith('.jpg')]

# Shuffle and split the dataset
random.shuffle(image_files)
split_ratio = 0.9
split_index = int(len(image_files) * split_ratio)

train_files = image_files[:split_index]
val_files = image_files[split_index:]

```

```

# Move the files to the respective directories
for file in train_files:
    shutil.move(os.path.join(images_path, file), train_images_path)
    shutil.move(os.path.join(masks_path, file.replace('.jpg', '.bmp')),
train_masks_path)

for file in val_files:
    shutil.move(os.path.join(images_path, file), val_images_path)
    shutil.move(os.path.join(masks_path, file.replace('.jpg', '.bmp')),
val_masks_path)

```

## لود کردن دیتابست

داده ها باید augment های مناسب و random داده شوند و به علاوه با توجه به mapping زیر برای کلاس ها باید map شوند به کمک کلاس زیر تمامی این بخش ها انجام میشود ( augment ها بر اساس مقاله انتخاب شده اند )

Object category	Symbol	RGB color code
Background (waterbody)	BW	000 (black)
Human divers	HD	001 (blue)
Aquatic plants and sea-grass	PF	010 (green)
Wrecks and ruins	WR	011 (sky)
Robots (AUVs/ROVs/instruments)	RO	100 (red)
Reefs and invertebrates	RI	101 (pink)
Fish and vertebrates	FV	110 (yellow)
Sea-floor and rocks	SR	111 (white)

شکل ۴۱: جدول MAP رنگ ها و کلاس ها در دیتابست SUIM

```

class SegmentationDataset(Dataset):
    def __init__(self, images_dir, masks_dir, transform=None,
target_transform=None, augmentation_factor=10, apply_augmentation=False):
        self.images_dir = images_dir
        self.masks_dir = masks_dir
        self.transform = transform
        self.target_transform = target_transform
        self.augmentation_factor = augmentation_factor
        self.apply_augmentation = apply_augmentation
        self.images = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]

    def __len__(self):
        return len(self.images) * self.augmentation_factor

    def __getitem__(self, idx):
        original_idx = idx % len(self.images)
        augment = idx >= len(self.images)

        image_path = os.path.join(self.images_dir, self.images[original_idx])
        mask_path = os.path.join(self.masks_dir,
self.images[original_idx].replace('.jpg', '.bmp'))

        image = Image.open(image_path).convert("RGB")
        mask = Image.open(mask_path)

        if self.apply_augmentation:
            image, mask = self.apply_transforms(image, mask)

        if self.transform is not None:
            image = self.transform(image)

        # Convert mask to class labels
        mask = self.convert_mask(mask)

        # Convert the NumPy array back to a PIL Image for target_transform
        mask = Image.fromarray(mask.astype(np.uint8))

        if self.target_transform is not None:
            mask = self.target_transform(mask)
        else:
            mask = torch.tensor(mask, dtype=torch.long)

        return image, mask

    def apply_transforms(self, image, mask):
        # Random horizontal flipping with 50% probability
        if random.random() > 0.5:
            image = TF.hflip(image)

```

```

mask = TF.hflip(mask)

# Random rotation with 50% probability
if random.random() > 0.5:
    angle = random.uniform(-20, 20)
    image = TF.rotate(image, angle)
    mask = TF.rotate(mask, angle, fill=0)

# Random affine transformations (shear, zoom, width/height shift) with
# 50% probability
if random.random() > 0.5:
    translate = (random.uniform(-0.05, 0.05), random.uniform(-0.05,
0.05))
    scale = random.uniform(1 - 0.05, 1 + 0.05)
    shear = random.uniform(-0.05, 0.05)
    image = TF.affine(image, angle=0, translate=translate, scale=scale,
shear=shear)
    mask = TF.affine(mask, angle=0, translate=translate, scale=scale,
shear=shear, fill=0)

return image, mask

def convert_mask(self, mask):
    mapping = {
        (0, 0, 0): 0,
        (0, 0, 255): 1,
        (0, 255, 0): 2,
        (0, 255, 255): 3,
        (255, 0, 0): 4,
        (255, 0, 255): 5,
        (255, 255, 0): 6,
        (255, 255, 255): 7
    }
    mask = np.array(mask)
    class_map = np.zeros(mask.shape[:2], dtype=np.int32)
    for rgb, idx in mapping.items():
        class_map[(mask == rgb).all(axis=2)] = idx
    return class_map

```

در کد فوق ،

:(`\_\_init\_\_`)

- masks\_dir : لیست که تصاویر و ماسک‌های مربوط به آن‌ها در آن ذخیره می‌شوند.
- "target-Transform", "transform" : تبدیل‌های اختیاری برای پیش‌پردازش تصاویر و ماسک‌ها.
- "Augment-Factor" : فاکتوری که به وسیله آن می‌توان به طور مصنوعی اندازه مجموعه داده را از طریق تقویت داده افزایش داد.
- `apply\_augmentation` : پرچم بولی برای اعمال یا عدم اعمال تقویت داده.
- `self.images` : فهرستی از نام فایل‌های تصویری در "images\_dir" که به ".jpg" ختم می‌شود.

طول مجموعه داده (`\_\_len\_\_`):

- تعداد کل داده در مجموعه داده را برمی‌گرداند که تعداد تصاویر ضرب شده در ضریب افزایش است.

دریافت (`\_\_getitem\_\_`):

- تصویر و ماسک مربوط به آن را بازگیری می‌کند و به صورت اختیاری تغییر شکل می‌دهد.
- شامل یک روش منحصر به فرد برای تبدیل ماسک‌ها به برچسب‌های کلاس است.

( `apply\_transforms` افزایش داده‌ها )

- تبدیل‌های تصادفی مختلفی را روی جفت تصویر و ماسک اعمال می‌کند، از جمله تبدیل افقی، چرخش، و تبدیل‌های افقی (مانند برش، بزرگنمایی و تغییر).
- هر تبدیل ۵۰ درصد شанс اعمال شدن دارد.

## تبدیل ماسک ('convert\_mask')

- یک تصویر ماسک رنگی را به نقشه کلاس تبدیل می کند.
- از نگاشت از پیش تعریف شده مقادیر رنگ RGB به شاخص های کلاس استفاده می کند.
- یک آرایه دو بعدی بر میگرداند که مقدار هر پیکسل با شاخص کلاس آن پیکسل مطابقت دارد.

در ادامه data loader ها تشکیل میشود

```
# Define the image transformations
transform = transforms.Compose([
    transforms.Resize((160, 160)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

target_transform = transforms.Compose([
    transforms.Resize((160, 160), interpolation=Image.NEAREST),
    transforms.Lambda(lambda x: torch.tensor(np.array(x), dtype=torch.long))
])

# Load the training dataset with augmentation
train_dataset = SegmentationDataset('dataset/train/images',
'dataset/train/masks',
                    transform=transform,
target_transform=target_transform,
                    augmentation_factor=3,
apply_augmentation=True)

# Load the validation dataset without augmentation
val_dataset = SegmentationDataset('dataset/val/images', 'dataset/val/masks',
                    transform=transform,
target_transform=target_transform,
                    augmentation_factor=1,
apply_augmentation=False)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,
num_workers=1)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=1)
```

در کد فوق تصاویر نرمال میشود و به سایز  $160 \times 160$  هم `resize` شده و در ادامه داده های آموزشی شامل `batch-size` میشوند در پایان با `batch-size` برابر با ۳۲ لود میشوند.

جهت تست سایز ها میتوان از کد زیر استفاده شود.

```
for i, l in train_loader:  
    print(i.shape)  
    print(l.shape)  
    break  
  
torch.Size([32, 3, 160, 160])  
torch.Size([32, 160, 160])
```

همانطور که مشاهده میشود همه چیز طبق انتظار میباشد (در مورد `label` ها خروجی بدین صورت میباشد که مقدار پیسکل ها بر اساس کلاس آن از ۰ تا ۷ میباشد).

به کمک کد زیر ۱۰ نمونه تصاویر آموزشی که شامل تقویت داده میباشد پلاس میشود. در کد عملات نرمالایز و `reverse` و `mapping` و `resize` کلاس ها شده است تا از عملکرد درست کلاس نوشته شده برای لود کردن دیتا نیز اطمینان یابیم.

```
# Function to convert a torch tensor to a numpy array  
def to_numpy(tensor):  
    return tensor.cpu().detach().numpy()  
  
# Function to reverse the class labels to RGB  
def labels_to_rgb(mask):  
    label_colors = {  
        0: [0, 0, 0],      # Background waterbody  
        1: [0, 0, 255],    # Human divers  
        2: [0, 255, 0],    # Plants/sea-grass  
        3: [0, 255, 255],  # Wrecks/ruins  
        4: [255, 0, 0],    # Robots/instruments  
        5: [255, 0, 255],  # Reefs and invertebrates  
        6: [255, 255, 0],  # Fish and vertebrates  
        7: [255, 255, 255] # Sand/sea-floor (& rocks)  
    }  
    rgb_image = np.zeros((mask.shape[0], mask.shape[1], 3), dtype=np.uint8)  
    for label, color in label_colors.items():  
        rgb_image[mask == label] = color  
    return rgb_image  
  
# Get 10 random images and masks from the dataset  
num_samples = 10  
selected_images = []
```

```

selected_masks = []

for _ in range(num_samples):
    images, masks = next(iter(train_loader))
    idx = random.randint(0, images.shape[0] - 1)
    image = to_numpy(images[idx])
    mask = to_numpy(masks[idx])

    # Normalize and transpose image
    image = np.transpose(image, (1, 2, 0)) # Change from CHW to HWC format
    image = image * np.array([0.229, 0.224, 0.225]) + np.array([0.485, 0.456,
0.406])
    image = np.clip(image, 0, 1)

    # Convert labels to RGB
    mask_rgb = labels_to_rgb(mask)

    selected_images.append(image)
    selected_masks.append(mask_rgb)

# Plot the images and masks
plt.figure(figsize=(25, 20)) # Adjust the size of the figure

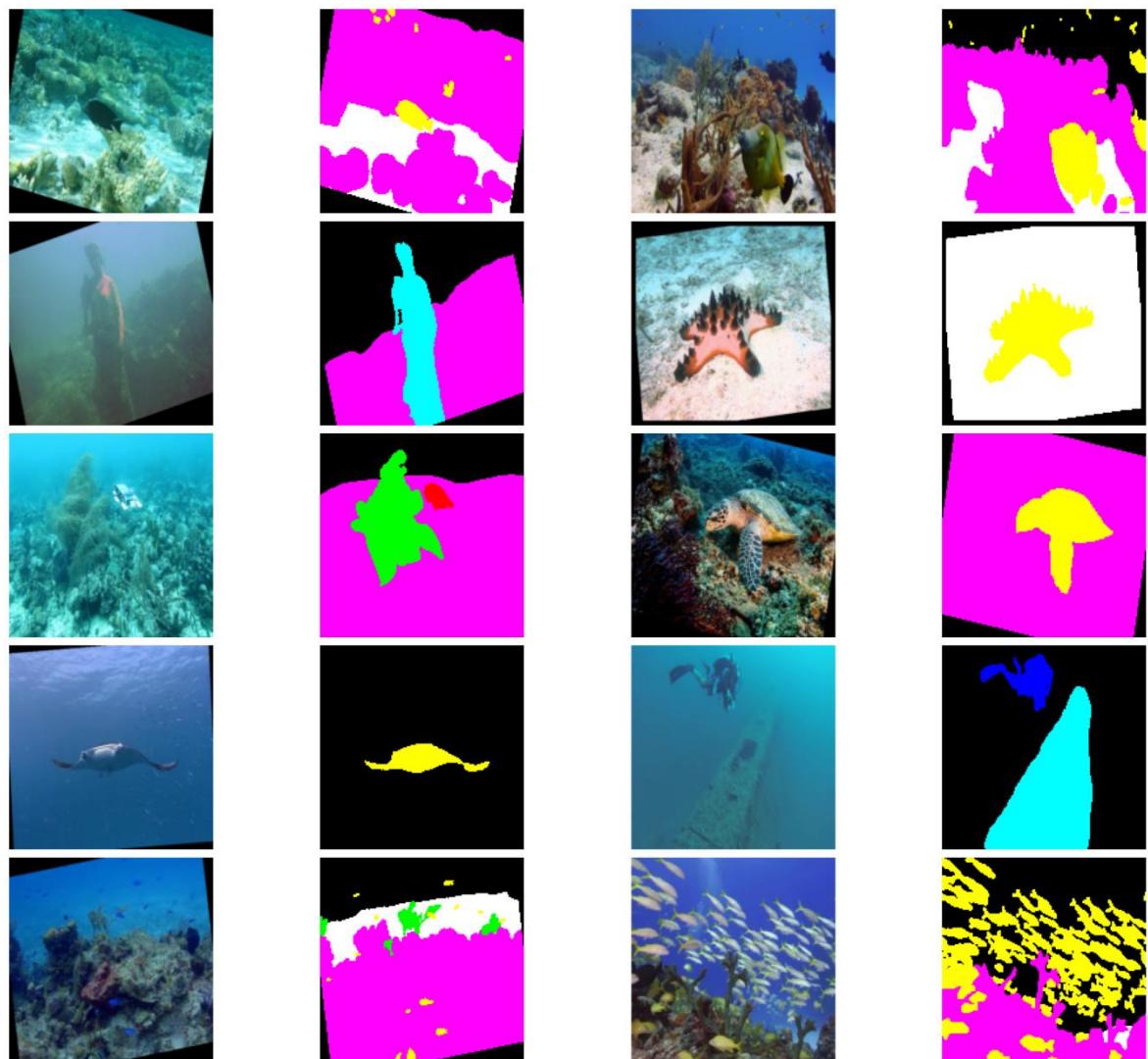
for i in range(num_samples):
    # Plot image
    plt.subplot(5, 4, 2*i + 1)
    plt.imshow(selected_images[i])
    plt.axis('off')

    # Plot mask
    plt.subplot(5, 4, 2*i + 2)
    plt.imshow(selected_masks[i])
    plt.axis('off')

plt.tight_layout()
plt.show()

```

بدین ترتیب با توجه به کد فوق میتوانیم ۱۰ نمونه تصویر داده های آموزشی را که بعضا augment هم شده است را مشاهده کنیم.

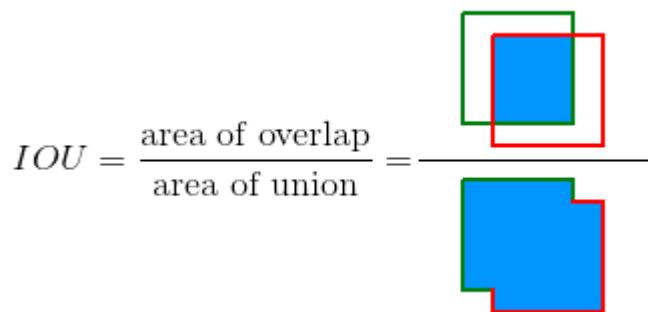


شکل ۴۲: نمونه تصاویر داده های آموزشی و لیبل مربوطه آن

## کد حلقه آموزش

جهت جلوگیری از توضیح مکرر این بخش آورده شده است تا یک بار فرایند آموزش و ... توضیح داده شود نتایج و پیاده سازی معماری ها در بخش های بعدی آمده است.

با فرض تعریف مدل و به جهت اهمیت محاسبه mIOU که همان میانگین نسبت intersection به Union میباشد.



شکل ۴۳: تعریف IoU

به راحتی به کمک کد زیر این مقدار محاسبه میشود.

```
def calculate_miou(preds, labels, num_classes):
    iou_list = []
    preds = torch.argmax(preds, dim=1)
    for cls in range(num_classes):
        pred_inds = (preds == cls)
        target_inds = (labels == cls)
        intersection = (pred_inds[target_inds]).long().sum().item()  # Intersection
        union = pred_inds.long().sum().item() + target_inds.long().sum().item() - intersection  # Union
        if union == 0:
            iou = float('nan')  # Avoid zero division
        else:
            iou = float(intersection) / float(max(union, 1))
        iou_list.append(iou)
    return np.nanmean(iou_list)  # Return the average IoU
```

در ادامه مدل لود میشود.

با توجه به مقاله مقدار lr برابر ۰,۰۰۴ و آن optimizer ADAM با مومنتوم برابر ۰,۵ میباشد.

```

device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# Parameters
n_channels = 3
n_classes = 8
lr = 0.0001
momentum = 0.5
n_epochs = 60

# Model, optimizer, and loss function
model = UNet(n_channels, n_classes).to(device)

# Create the Adam optimizer with momentum
optimizer = optim.Adam(model.parameters(), lr=lr, betas=(momentum, 0.999))
scheduler = StepLR(optimizer, step_size=50, gamma=0.1)

criterion = nn.CrossEntropyLoss()

```

در ادامه حلقه آموزش نوشته شده است که بهترین وزن ها بر اساس **IOU** را داده های ارزیابی ذخیره میشود و به علاوه از **early stop** استفاده شده است تا اگر پیشرفتی نداشتیم فرایند آموزش قطع شود.

```

from tqdm import tqdm
# Initialize the best validation mIoU
best_val_miou = 0.0

# Define the directory and filename for saving the model
model_save_path = 'models'
os.makedirs(model_save_path, exist_ok=True)
# Define the filename for saving the best model
best_model_filename = 'best_Unet.pth'

# Initialize lists to store metrics
train_losses = []
val_losses = []
train_miou = []
val_miou = []

early_stop_patience = 8
early_stop_counter = 0

# Training loop
for epoch in range(n_epochs):
    model.train()
    total_train_loss = 0.0

```

```

total_train_iou = 0.0

for images, masks in tqdm(train_loader):
    # Move data to device
    images, masks = images.to(device), masks.to(device)

    optimizer.zero_grad()
    outputs = model(images)
    loss = criterion(outputs, masks)
    loss.backward()
    optimizer.step()

    total_train_loss += loss.item()
    total_train_iou += calculate_miou(outputs, masks, n_classes)

avg_train_loss = total_train_loss / len(train_loader)
avg_train_iou = total_train_iou / len(train_loader)
train_losses.append(avg_train_loss)
train_miou.append(avg_train_iou)

scheduler.step()

# Validation loop
model.eval()
total_val_loss = 0.0
total_val_iou = 0.0

with torch.no_grad():
    for images, masks in val_loader:
        # Move data to device
        images, masks = images.to(device), masks.to(device)

        outputs = model(images)
        loss = criterion(outputs, masks)
        total_val_loss += loss.item()
        total_val_iou += calculate_miou(outputs, masks, n_classes)

avg_val_loss = total_val_loss / len(val_loader)
avg_val_iou = total_val_iou / len(val_loader)
val_losses.append(avg_val_loss)
val_miou.append(avg_val_iou)

print(f'Epoch {epoch+1}, Train Loss: {avg_train_loss}, Train mIoU: {avg_train_iou}, Val Loss: {avg_val_loss}, Val mIoU: {avg_val_iou}')

# Check if the current validation mIoU is the best one
if avg_val_iou > best_val_miou:
    print(f"New best validation mIoU: {avg_val_iou} at epoch {epoch+1}")

```

```

best_val_miou = avg_val_iou

# Save the model
best_save_path = os.path.join(model_save_path, best_model_filename)
torch.save(model.state_dict(), best_save_path)
print(f"Best model saved to {best_save_path}")

early_stop_counter = 0
else:
    # Increment early stopping counter
    early_stop_counter += 1

# Check if training should be stopped early
if early_stop_counter >= early_stop_patience:
    print(f"Early stopping at epoch {epoch+1} due to no improvement in
validation mIoU for {early_stop_patience} consecutive epochs.")
    break

```

در نهایت نیز به کمک کد زیر منحنی های مورد نیاز رسم میشود.

```

plt.figure(figsize=(12, 5))

# Plotting training and validation loss
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.title('Training & Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plotting training and validation mIoU
plt.subplot(1, 2, 2)
plt.plot(train_miou, label='Train mIoU')
plt.plot(val_miou, label='Val mIoU')
plt.title('Training & Validation mIoU')
plt.xlabel('Epochs')
plt.ylabel('mIoU')
plt.legend()

plt.show()

```

### پیاده سازی UNET

با توجه به توضیحات داده شده و بر اساس مقاله UNET مدل به صورت زیر پیاده سازی میشود.

```
class DoubleConv(nn.Module):  
    def __init__(self, in_channels, out_channels, mid_channels=None):  
        super(DoubleConv, self).__init__()  
        if not mid_channels:  
            mid_channels = out_channels  
        self.double_conv = nn.Sequential(  
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1),  
            nn.BatchNorm2d(mid_channels),  
            nn.ReLU(inplace=True),  
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1),  
            nn.BatchNorm2d(out_channels),  
            nn.ReLU(inplace=True)  
        )  
  
    def forward(self, x):  
        return self.double_conv(x)  
  
class UNet(nn.Module):  
    def __init__(self, n_channels, n_classes):  
        super(UNet, self).__init__()  
        self.n_channels = n_channels  
        self.n_classes = n_classes  
  
        self.inc = DoubleConv(n_channels, 64)  
        self.down1 = DoubleConv(64, 128)  
        self.down2 = DoubleConv(128, 256)  
        self.down3 = DoubleConv(256, 512)  
        self.up1 = DoubleConv(512 + 256, 256) # Adjusted channels for  
concatenation  
        self.up2 = DoubleConv(256 + 128, 128)  
        self.up3 = DoubleConv(128 + 64, 64)  
        self.outc = nn.Conv2d(64, n_classes, kernel_size=1)  
  
    def forward(self, x):  
        x1 = self.inc(x)  
        x2 = F.max_pool2d(x1, 2)  
        x3 = self.down1(x2)  
        x4 = F.max_pool2d(x3, 2)  
        x5 = self.down2(x4)
```

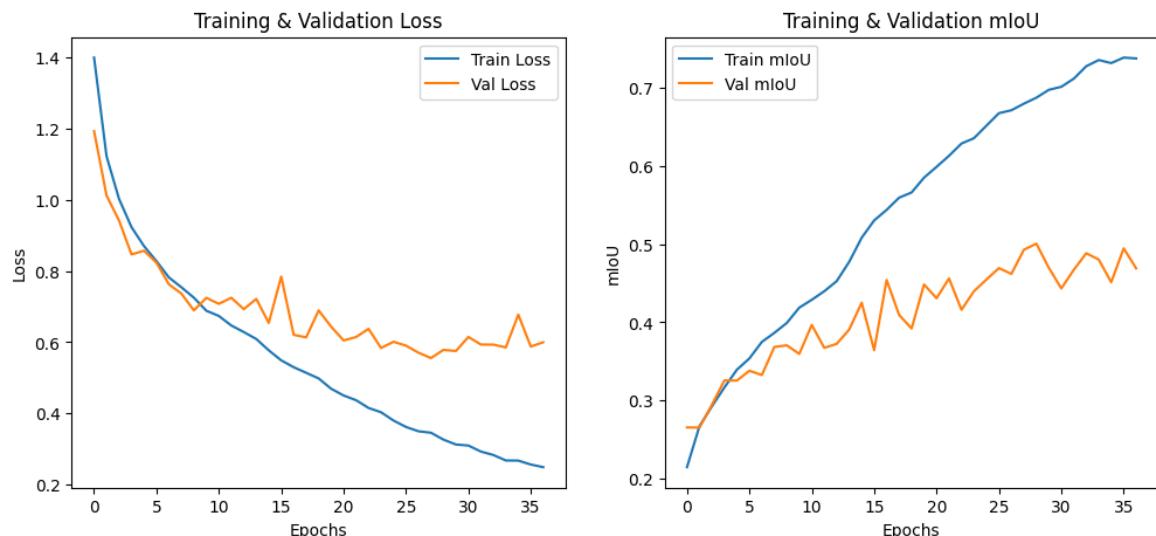
```

x6 = F.max_pool2d(x5, 2)
x7 = self.down3(x6)
x8 = F.interpolate(x7, scale_factor=2, mode='bilinear',
align_corners=True)
x8 = torch.cat([x8, x5], dim=1)
x9 = self.up1(x8)
x10 = F.interpolate(x9, scale_factor=2, mode='bilinear',
align_corners=True)
x10 = torch.cat([x10, x3], dim=1)
x11 = self.up2(x10)
x12 = F.interpolate(x11, scale_factor=2, mode='bilinear',
align_corners=True)
x12 = torch.cat([x12, x1], dim=1)
x13 = self.up3(x12)
logits = self.outc(x13)
return logits

```

خروجی نهایی باید به صورت `logits` باشد و همین دلیل در کد بالا به این صورت میباشد.

آموزش با `augment factor` برابر ۳ انجام شد و هر ایپاک حدود ۶ دقیقه طول کشید.



شکل ۴ : منحنی loss و MIOU برای داده های ارزیابی و آموزشی بر روی شبکه UNET

همانطور که مشاهده میشود `MiOU` بر روی `val` برابر با ۵۰,۵ میباشد که با توجه به ۸ کلاس کامل و سایز تصویر  $160*160$  مقدار مناسبی میباشد.

در ادامه به کمک کد زیر داد مقدار آن بر روی دادهای تست محاسبه شده است:

```

# Define the image transformations
transform = transforms.Compose([
    transforms.Resize((160, 160)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

target_transform = transforms.Compose([
    transforms.Resize((160, 160), interpolation=Image.NEAREST),
    transforms.Lambda(x: torch.tensor(np.array(x), dtype=torch.long))
])

# Load the validation dataset without augmentation
test_dataset = SegmentationDataset('dataset/TEST/images', 'dataset/TEST/masks',
                                    transform=transform,
                                    target_transform=target_transform,
                                    augmentation_factor=1,
                                    apply_augmentation=False)

# Create data loaders
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False,
                        num_workers=1)

# Define the path to the saved model
saved_model_path = 'models/best_Unet_x.pth'

# Make sure the model is in evaluation mode
model.load_state_dict(torch.load(saved_model_path))
model.eval()
print("Model loaded")

total_test_iou = 0.0

# Run the model on the test data
with torch.no_grad():
    for images, masks in test_loader:
        images, masks = images.to(device), masks.to(device)
        outputs = model(images)
        total_test_iou += calculate_miou(outputs, masks, n_classes)

avg_test_iou = total_test_iou / len(test_loader)
print(f'Test mIoU: {avg_test_iou}')

```

مقدار mIoU بر روی داده های تست برابر ۴۷,۳۶ میباشد.

در ادامه به کمک همان کد پلاس کردن از مدل خروجی گرفته و خروجی آن را برای ۱۰ نمونه رسم میکنیم.

برای پلاس از کد زیر استفاده شده است.

```
# Function to convert a torch tensor to a numpy array
def to_numpy(tensor):
    return tensor.cpu().detach().numpy()

# Function to get predictions from the model
def get_predictions(model, loader):
    images, _ = next(iter(loader))
    images = images.to(device)
    with torch.no_grad():
        preds = model(images)
    return images, preds

# Function to reverse the class labels to RGB
def labels_to_rgb(mask):
    label_colors = {
        0: [0, 0, 0],          # Background
        1: [0, 0, 255],        # Class 1
        2: [0, 255, 0],        # Class 2
        3: [0, 255, 255],      # Class 3
        4: [255, 0, 0],        # Class 4
        5: [255, 0, 255],      # Class 5
        6: [255, 255, 0],      # Class 6
        7: [255, 255, 255]    # Class 7
    }
    rgb_image = np.zeros((mask.shape[0], mask.shape[1], 3), dtype=np.uint8)
    for label, color in label_colors.items():
        rgb_image[mask == label] = color
    return rgb_image

# Function to calculate mIoU for each sample
def calculate_sample_miou(preds, gts, n_classes):
    sample_miou = []
    for pred, gt in zip(preds, gts):
        miou = calculate_miou(pred.unsqueeze(0), gt.unsqueeze(0), n_classes)
        sample_miou.append(miou.item())
    return sample_miou

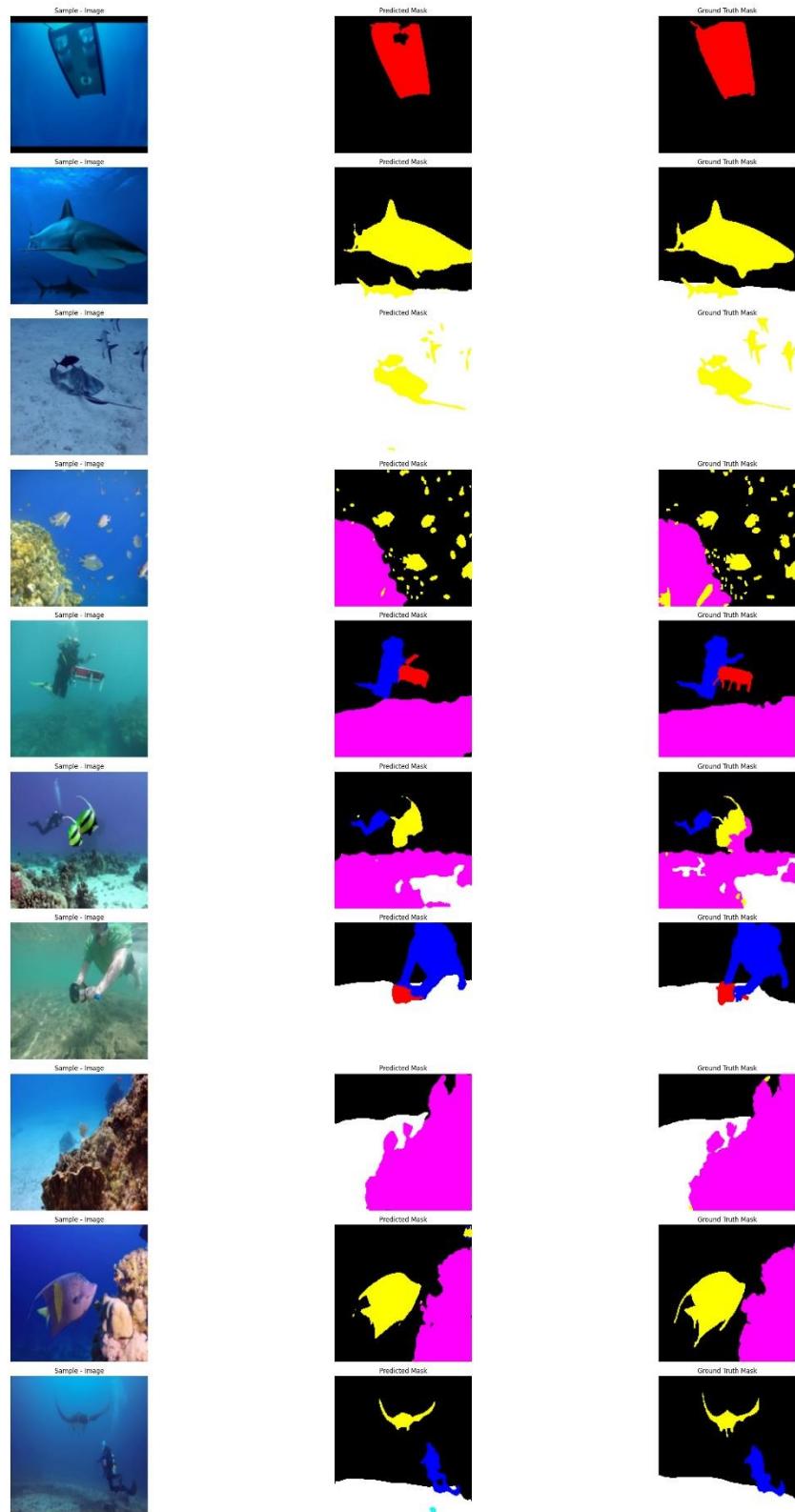
# Modified function to get predictions and ground truth masks
def get_predictions_and_gt(model, loader):
    model.eval()
    all_images, all_preds, all_gts = [], [], []
    with torch.no_grad():
        for images, masks in loader:
            images, masks = images.to(device), masks.to(device)
            preds = model(images)
```

```

        all_images.append(images)
        all_preds.append(preds)
        all_gts.append(masks)
    return all_images, all_preds, all_gts
# Get predictions and ground truth masks
all_images, all_preds, all_gts = get_predictions_and_gt(model, test_loader)
# Flatten the lists
images_flat = torch.cat(all_images)
preds_flat = torch.cat(all_preds)
gts_flat = torch.cat(all_gts)
# Calculate mIoU for each sample
sample_miou = calculate_sample_miou(preds_flat, gts_flat, n_classes)
# Create a list of tuples (miou, image, prediction, ground_truth)
samples_with_miou = list(zip(sample_miou, images_flat, preds_flat, gts_flat))
top_samples = sorted(samples_with_miou, key=lambda x: x[0], reverse=True)[:10]
# Plot the top 10 samples
plt.figure(figsize=(30, 40))
for i, (miou, image, pred, gt) in enumerate(top_samples):
    image = to_numpy(image)
    pred = to_numpy(torch.argmax(pred, dim=0))
    gt = to_numpy(gt.squeeze(0))
    # Normalize and clip image for visualization
    image = np.transpose(image, (1, 2, 0))
    image = image * np.array([0.229, 0.224, 0.225]) + np.array([0.485, 0.456, 0.406])
    image = np.clip(image, 0, 1)
    # Convert predicted and ground truth labels to RGB
    pred_rgb = labels_to_rgb(pred)
    gt_rgb = labels_to_rgb(gt)
    # Plotting
    plt.subplot(10, 3, 3*i + 1)
    plt.imshow(image)
    plt.title(f'Sample - Image')
    plt.axis('off')
    plt.subplot(10, 3, 3*i + 2)
    plt.imshow(pred_rgb)
    plt.title('Predicted Mask')
    plt.axis('off')
    plt.subplot(10, 3, 3*i + 3)
    plt.imshow(gt_rgb)
    plt.title('Ground Truth Mask')
    plt.axis('off')
plt.tight_layout()
plt.savefig("10_samples_iou.png")
plt.show()

```

که خروجی آن به صورت زیر میباشد.



شکل 45 : خروجی مدل صورت سوال **UNET** بر روی داده تست

نتایج نشان میدهد که عملکرد قابل قبولی داشته است.

مقایسه و تحلیل ها در بخش آخر آمده است.

## پیاده سازی PSPNET

مطابق به توضیحات داده شده و معماری PSPNet و توجه به اینکه خروجی فیچر های Mobilenet- ۷۲ برابر ۱۲۸۰ تا میباشد مدل به صورت زیر پیاده سازی میشود.

```
class PyramidPoolingModule(nn.Module):  
    def __init__(self, in_channels, pool_sizes, out_channels):  
        super(PyramidPoolingModule, self).__init__()  
        self.pools = nn.ModuleList([nn.AdaptiveAvgPool2d(output_size=size) for  
size in pool_sizes])  
        self.convs = nn.ModuleList([nn.Conv2d(in_channels, out_channels,  
kernel_size=1, bias=False) for size in pool_sizes])  
        self.batch_norms = nn.ModuleList([nn.BatchNorm2d(out_channels) for _ in  
pool_sizes])  
        self.reltus = nn.ModuleList([nn.ReLU(inplace=True) for _ in pool_sizes])  
  
    def forward(self, x):  
        size = x.size()[2:]  
        cat = [x]  
        for pool, conv, bn, relu in zip(self.pools, self.convs, self.batch_norms,  
self.reltus):  
            p = pool(x)  
            p = F.interpolate(p, size=size, mode='bilinear', align_corners=False)  
            p = conv(p)  
            p = bn(p)  
            p = relu(p)  
            cat.append(p)  
        return torch.cat(cat, dim=1)  
  
class PSPNet(nn.Module):  
    def __init__(self, num_classes):  
        super(PSPNet, self).__init__()  
        # Use the pre-trained MobileNetV2 model features  
        mobilenet = mobilenet_v2(pretrained=False)  
        self.backbone = mobilenet.features  
  
        # The output channels of the backbone's last layer  
        backbone_out_channels = 1280 # This is specific to MobileNetV2  
        architecture  
  
        # Pyramid Pooling Module  
        self.ppm = PyramidPoolingModule(in_channels=backbone_out_channels,  
                                         pool_sizes=[1, 2, 3, 6],
```

```

        out_channels=256)

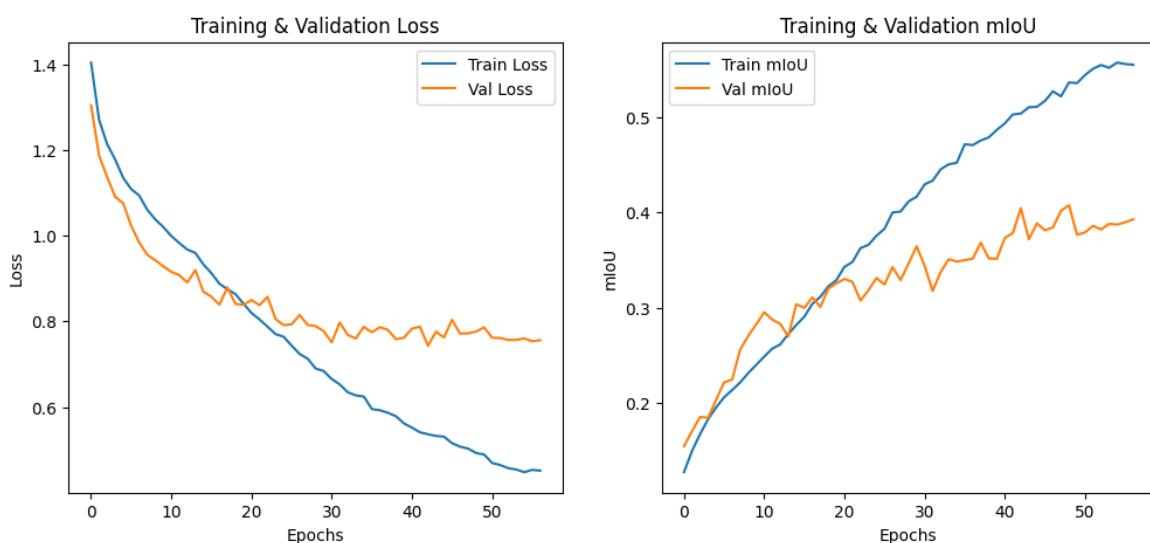
    # The final convolution layer to get the logits for each class
    # The output channels of PPM will be in_channels + out_channels *
len(pool_sizes)
    self.final_conv = nn.Conv2d(backbone_out_channels + 256 * 4,
                               num_classes, kernel_size=1)

def forward(self, x):
    x = self.backbone(x)
    x = self.ppm(x)
    x = self.final_conv(x)
    x = F.interpolate(x, size=(160, 160), mode='bilinear',
align_corners=False)
    return x

```

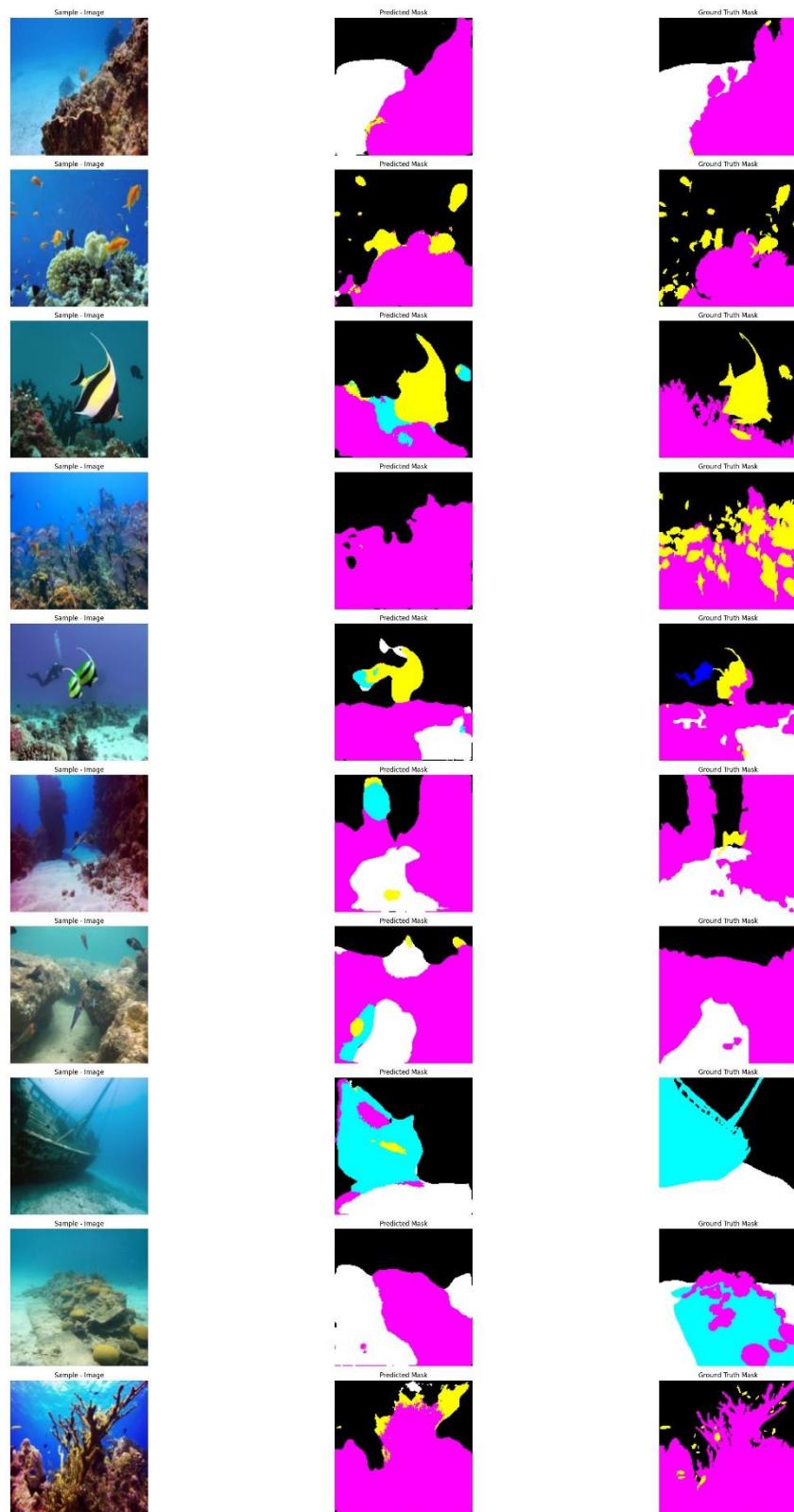
بدین ترتیب منحنی ها به صورت زیر خواهد بود

( شرایط آموزش برای هر ۲ شبکه یکسان در نظر گرفته شد که بتوان مقایسه عادلانه ای کرد )



شکل ۴۶: منحنی آموزش و ارزیابی برای [PSPNet](#)

مقدار mIoU برای test نیز برابر ۰.۳۵۶۸۱۱ میباشد و شکل زیر ۱۰ نمونه از خروجی ها را نشان میدهد.



شکل 47 : خروجی روزی تست برای PSPNet

همانطور که مشاهده میشود نتایج نسبت به قبل بدتر میباشد که نشان از برتری UNET نسبت به PSPNet دارد.

مقایسه ها در بخش بعدی آمده است.

### مقایسه و تحلیل نتایج

همانطور که در بخش معرفی دیتاست توضیح داده شد مقایسه مقاله ( که در آنجا هم نشان میدهد UNET از PSPNet بهتر میباشد ) نمیتواند معیار خوبی برای مقایسه باشد چراکه ابعاد تصویر و تعداد کلاس ها متفاوت میباشد.

به همین دلیل ما خروجی را برای انالیز بهتر بر اساس مدل های زیر و ابعاد و factor augment های متفاوت گرفتیم که نتایج درجدول زیر آمده است.

در مدل زیر منظور از factor augment برابر ۱ یعنی Augment صورت نگرفته است.

استفاده شده NVIDIA RTX 3090 با حافظه ۲۴ گیگ میباشد و زمان های run time بر اساس اون میباشد.

جدول ۱۷ : مقایسه مدل های مختلف با Dimension و Augment های مختلف

	Dimension	UNet – Val MIOU	UNet – Test MIOU	PSPNet – Val MIOU	PSPNet – Test MIOU	Run Time/ epoch for UNET
Augment Factor = 1	160*160	38.7	33.3	35.4	29.4	58s
Augment Factor = 1	320*320	47.9	44.6	42.7	42.1	83s
Augment Factor = 2	160*160	42.5	39.9	38.7	36.7	117s
Augment Factor = 2	320*320	52.1	46.4	46.3	44.8	224s

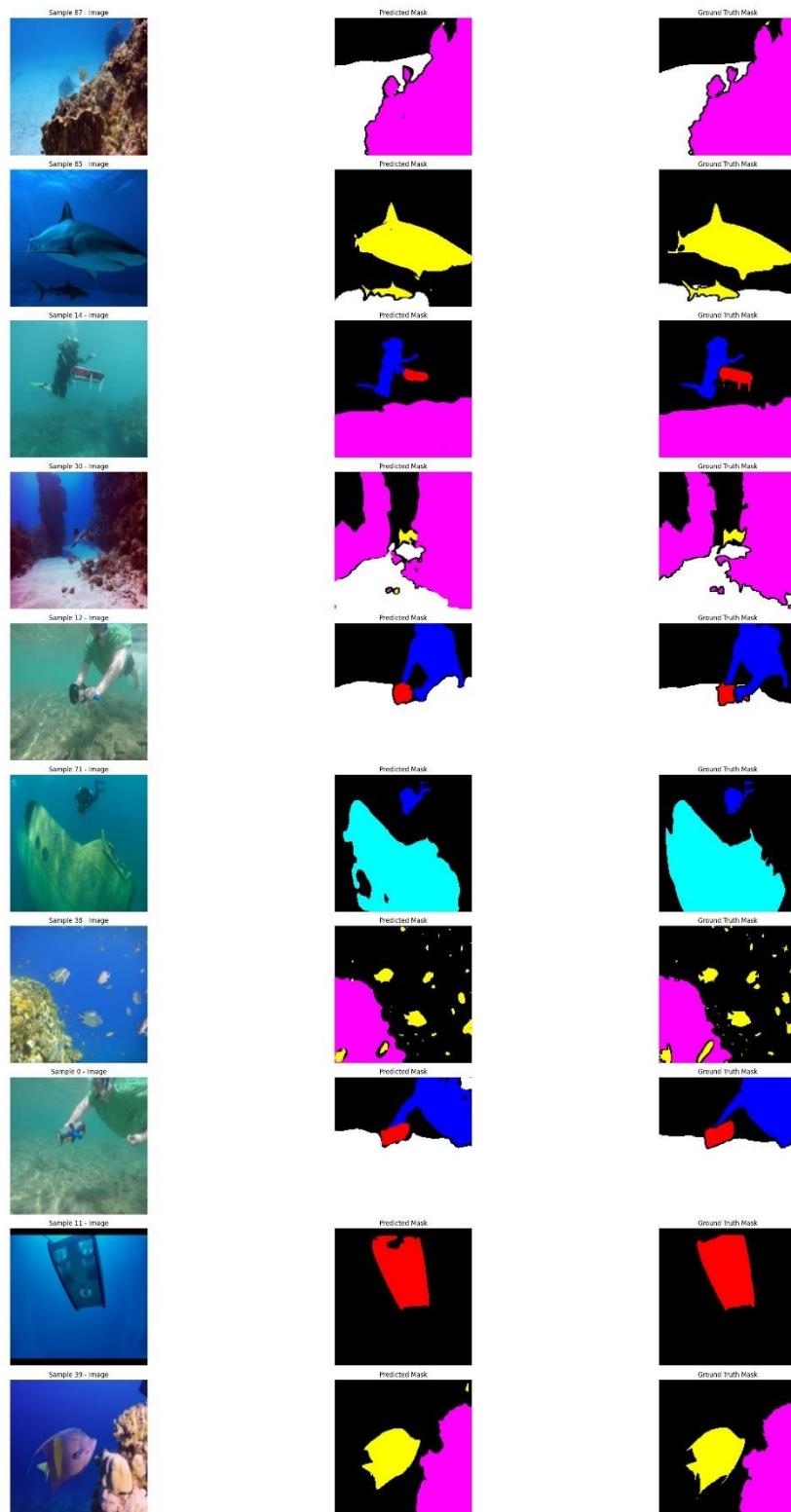
Augment Factor = 3	160*160	50.5	47.36	40.5	35.68	305s
Augment Factor = 3	320*320	63.7	60.2	58.3	53.4	662s

خروجی های با ابعاد ۱۶۰ در ۱۶۰ که در بخش های قبلی آمده است در ادامه خروجی ها روی تست با ابعاد ۳۲۰ در ۳۲۰ آمده است.

همانطور که از روند جدول مشخص است اضافه کردن **Augment factor** در **Augment** تا حد خوبی میتواند دقت ها را بالا برده ولی باید در نظر داشت زیاد تر کردن داده ها میتواند منجر به زمان آموزش طولانی تر بشود.

مورد بعدی بحث **Dimension** میباشد که طبیعی است بسیار بر روی آموزش و دقت ها تاثیر میگذارد و هر چه ابعاد ورودی بالاتر باشد دقت ها بسیار بهتر میشود که نمونه خروجی های شبکه بر روی تست در ادامه آمده است.

به علاوه همانطور که مشخص است عملکرد **UNet** بهتر از **PSPNet** در مورد این دیتاست میباشد که نشان میدهد **U-NET** به دلیل ساختار فشرده سازی و بازسازی ای که دارد با وجود دیتا محدود و پیچیده تر میتواند عملکرد و **generalization** شبکه **mobilenet** بیشتری داشته باشد. همچنین **PSP-Net** شبکه بسیار سبکی میباشد که در انتهای ۱۲۸۰ فیجر استخراج میکند و شاید استفاده از **backbone** ها قوی تر بتواند ورق را به سمت **PSP-Net** برگرداند.



شکل 48 : خروجی تست بر روی UNET با تصاویر ۳۲۰ در ۳۲۰