

# به نام خدا



دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر



درس تحلیل و طراحی شبکه‌های عصبی عمیق

## تمرین 1 Extra

نام و نام خانوادگی : علیرضا حسینی

شماره دانشجویی : 810101142

آذر ماه 1402

## فهرست

3	..... مقدمه
3	..... سوال اول : آشنایی با ساختار شبکه های عصبی
45	..... سوال دوم : شبکه تشخیص اشیا

## مقدمه

هدف از انجام این تمرین آشنایی با ساختار یک شبکه عصبی مشابه **ResNet** و بررسی تاثیر لایه های مختلف از جمله لایه های مختلف نرمالسازی بر میزان دقت و شاخص **SI** شبکه های عصبی است. همچنین آشنایی و پیاده سازی یک شبکه عصبی جهت تشخیص اشیا که توانایی تشخیص اشیا در محیط را دارد نیز در بخش دوم تمرین قرار گرفته است.

## سوال اول : آشنایی با ساختار شبکه های عصبی

### 1. کد لود دیتاست و لوپ ترین و اندازه گیری **CSI** و **SI** برای هر لایه

در این بخش تمامی پیاده سازی های برای لود دیتاست و آموزش شبکه و .. آمده است که برای تمامی حالت ها این بخش میباشد و جهت جلوگیری از تکرار مکرات تنها یکبار در اینجا توضیح داده میشود.

\*\* نکته مهم در کد های ارسالی : تمامی کد ها روی NVIDIA GeForce RTX 3090 و بر روی **cuda:1** سرور ران شده است و جهت ران بر روی سیستم خودتان در صورت نداشتن چندین **cuda out of memory** باید 1 را به صفر تغییر داد و در مواردی که ممکن است بر روی کولب **batch** باید **SI** های محاسبه **batch** کاهش یابد.

#### 1-1. لود کردن دیتاست

دیتاست CIFAR10 را لود کرده و داده ها را نرمال میکنیم ، همچنین از **augmentation** های **Flip** و **Rotation** هم استفاده میشود.

لازم به ذکر است از **Aug** های دیگری نیز استفاده شده است که با توجه به خروجی ها **AUG** های زیر در نظر گرفته شده است.

```
# Data augmentation and normalization
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

```

])

# # Data augmentation and normalization for training
# transform_train = transforms.Compose([
#     transforms.RandomHorizontalFlip(),
#     transforms.RandomRotation(15),
#     transforms.RandomResizedCrop(32, scale=(0.8, 1.0)),
#     transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
#     transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
#     transforms.ToTensor(),
#     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
# ])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
train_dataset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)

```

در نهایت نیز 10 درصد از داده های آموزش را جدا کرده و به عنوان `val` قرار میدهیم و های آموزش و ارزیابی و تست را تشکیل میدهیم.

```

# Splitting train dataset into train and validation sets
train_size = int(0.9 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=256, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)

```

## 2-1. حلقه آموزش شبکه

جهت آموزش مدل آن را لود کرده و `optimizer` و .. تعریف میشود ، همچنین از `Schaduler` گفته شده استفاده شده است و مدل در نهایت روی `GPU` قرار میگیرد.

لازم به ذکر است تمامی مدل های آتی با همین `setup` آموزش دیده اند که شرایط برابری بريا مقایسه وجود داشته باشد.

```
# Model, optimizer, and scheduler
model = CustomResNet()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = StepLR(optimizer, step_size=40, gamma=0.1)
criterion = nn.CrossEntropyLoss()
# Determine if CUDA is available and set the device accordingly
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
model.to(device)
```

کد آموزش و تست به صورت زیر نوشته میشود تا مقادیر `loss` را در دقت را در هر بار فراخوانی بازگرداند و در فرایند آموزش پارامتر ها را آپدیت کند.

```
def train(model, device, train_loader, optimizer, criterion):
    model.train()
    train_loss = 0
    correct = 0
    total = 0
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        train_loss += loss.item()
        loss.backward()
        optimizer.step()

        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

    train_loss /= len(train_loader)
    train_accuracy = 100 * correct / total
    return train_loss, train_accuracy

def test(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
```

```

        loss = criterion(output, target)
        test_loss += loss.item()
        _, predicted = output.max(1)
        correct += predicted.eq(target).sum().item()

    test_loss /= len(test_loader)
    test_accuracy = 100. * correct / len(test_loader.dataset)
    return test_loss, test_accuracy

```

در آخر میتوان حلقه آموزش را به صورت زیر تعریف کرد.

در حلقه لاس ها جهت پلات نهایی دخیره میشود و معیار ذخیره سازی مدل بهترین دقت روی داده های ارزیابی میباشد نه الزاما آخرین ایپاک

```

# Training loop with tracking of training and validation accuracies
train_losses, train_accuracies, val_losses, val_accuracies = [], [], [], []
best_val_accuracy = 0
best_model_weights = None

for epoch in range(1, 101):
    train_loss, train_accuracy = train(model, device, train_loader, optimizer, criterion)
    val_loss, val_accuracy = test(model, device, val_loader, criterion)

    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_model_weights = model.state_dict()

    print(f'Epoch: {epoch}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%, Val Loss: {val_loss:.4f}, Val Accuracy: {val_accuracy:.2f}%' )
    scheduler.step()

```

در ادامه نیز میتوان به کمک کد های زیر منحین های دقت و لاس را نمایش داد. و همچنان دقت بهترین مدل را بر روی داده های تست و ارزیابی به دست آورد.

```

# Load best model weights and evaluate on test set
model.load_state_dict(best_model_weights)
test_loss, test_accuracy = test(model, device, test_loader, criterion)

```

```

# Plotting training and validation loss and accuracy
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.legend()

plt.show()

# Print best validation accuracy and test accuracy
print(f'Best Validation Accuracy: {best_val_accuracy:.2f}%')
print(f'Test Accuracy of the final model: {test_accuracy:.2f}%')

# Save best model weights
torch.save(best_model_weights, 'best_custom_resnet_weights.pth')

```

### 3-1. آنالیز لایه به لایه با متریک SI و CSI

برای محاسبه SI و CSI از کد موجود در گیتهاب خودم که سریع تر و قابلیت **batch handler** دارد استفاده شده است که توضیح کامل آن در تمرین دوم آمده است و در لینک زیر موجود میباشد استفاده شده است.

[https://github.com/Arhosseini77/data\\_complexity\\_measures/blob/main/models/ARH\\_SeparationIndex.py](https://github.com/Arhosseini77/data_complexity_measures/blob/main/models/ARH_SeparationIndex.py)

جهت آنالیز لایه به لایه نیاز است تا مجددا دیتاست را لود کرد و دیگر از AUG نیز استفاده نشود همچنین با توجه به صورت سوال از 20 درصد داده های آموزشی استفاده میشود.

```

# Data augmentation and normalization
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

```

```

])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
train_dataset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)

# Splitting train dataset into train and validation sets
train_size = int(0.9 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

# Further split train_dataset into train_loader_dataset and 20% balanced subset
train_loader_dataset, _ = train_test_split(train_dataset.dataset, train_size=int(0.2 * len(train_dataset)),
                                             test_size=None, shuffle=True,
                                             stratify=train_dataset.dataset.targets)

# Data loaders
train_loader = DataLoader(train_loader_dataset, batch_size=256, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=256, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)

```

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

```

# Instantiate and load the model
model = CustomResNet()  # Make sure CustomResNet is defined as earlier
model.load_state_dict(torch.load('best_custom_resnet_weights.pth'))
model.to('cuda:1' if torch.cuda.is_available() else 'cpu')
model.eval()

```

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

متغیر `exclude_layers` برای وقتی میباشد که میخواهیم لایه ای را خروجی نگیریم. در اینجا خروجی از تمامی لایه ها گرفته شده است ولی در حالت اول لایه های `fc1` و `fc2` و `flatten` در آن `exclude` شده است ولی در سایر حالت ها از این لایه ها هم مقادیر `CSI` و `SI` محاسبه شده است.

```

# 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 **Separation index** درآورد.

به عنوان مثال برای **train loader** داریم :

```

# 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)

```

: محاسبه SI

```

si_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)
    si = instance_disturbance.si_batch(batch_size=2000)
    si_layer_train.append((layer_name, si))

```

رسم مقادیر SI بر حسب هر لایه :

```

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

```

: محاسبه ( Center SI ) CSI

```

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))

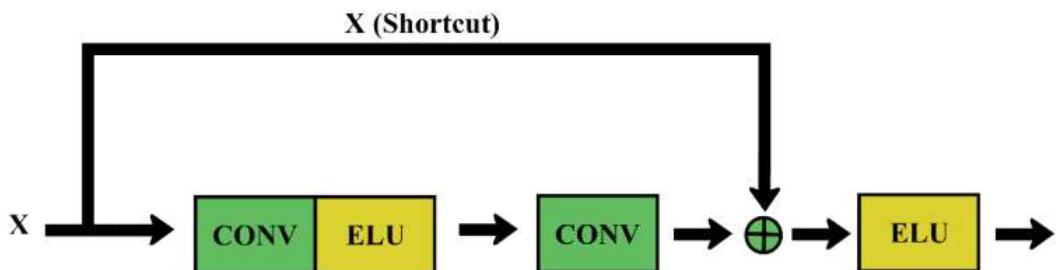
```

رسم هم همانند SI صورت میگیرد.

## 2. پیاده سازی Custom Resnet

برای پیاده سازی Custom Resnet داده شده از فریم ورک Pytorch استفاده شده است.

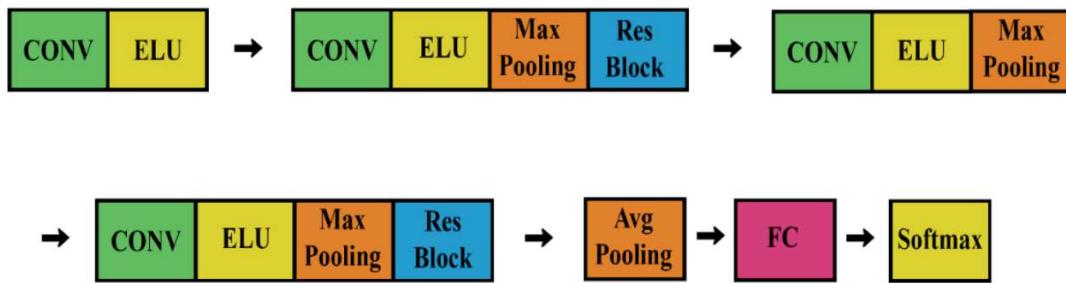
با توجه به Residual Block داده شده میتوان آن را به صورت زیر پیاده سازی کرد.



شکل 1

```
class CustomResNetBlock(nn.Module):  
  
    def __init__(self, in_channels, out_channels):  
        super(CustomResNetBlock, self).__init__()  
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)  
        self.elu = nn.ELU()  
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)  
  
    def forward(self, x):  
        shortcut = x  
        x = self.conv1(x)  
        x = self.elu(x)  
        x = self.conv2(x)  
        x += shortcut  
        x = self.elu(x)  
        return x
```

در ادامه از ساختار داده شده که به صورت زیر میباشد برای پیاده سازی مدل داده شده استفاده میشود.



شکل 2 : ساختار Custom Resnet

با توجه به ساختار فوق میتوان کد مدل را به صورت زیر نوشت.

```
class CustomResNet(nn.Module):
    def __init__(self):
        super(CustomResNet, self).__init__()
        # Initial convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.elu1 = nn.ELU()
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.elu2 = nn.ELU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Residual blocks
        self.resblock1 = CustomResNetBlock(64, 64)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.elu3 = nn.ELU()
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.elu4 = nn.ELU()
        self.resblock2 = CustomResNetBlock(256, 256)

        # Average pooling and fully connected layers
        self.avgpool = nn.AvgPool2d(kernel_size=3, stride=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, 10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.elu1(x)
        x = self.conv2(x)
        x = self.elu2(x)
        x = self.maxpool1(x)
```

```

x = self.resblock1(x)
x = self.conv3(x)
x = self.elu3(x)
x = self.maxpool2(x)

x = self.conv4(x)
x = self.elu4(x)
x = self.maxpool3(x)
x = self.resblock2(x)
x = self.avgpool(x)
x = self.flatten(x)
x = self.fc1(x)
x = self.fc2(x)
x = self.softmax(x)

return x

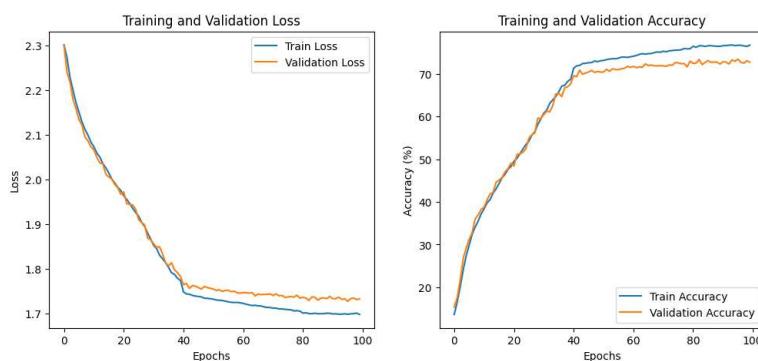
```

تنها نکته قابل ذکر در پیاده سازی فوق این است که با توجه به فایل تمرین که گفته شده در اخیرین لایه فیچر برابر 3 باشد سایز ها به صورت زیر با stride برابر با 1 میشود  $2 \times 2 \times 1$ . بنابراین در پیاده سازی kernel size برابر 2 درنظر گرفته شده است تا spatial size برابر  $1 \times 1$  شود.

21	ELU	-	-	$(4 \times 4 \times 256)$
22	Avg. Pooling	$3 \times 3$	-	$(1 \times 1 \times 256)$
23	Flatten	-	-	(256)

شکل 3 : بخش سایز ها چهت stride برابر 2

پس از پیاده سازی و آموزش مدل منحنی های دقت به صورت زیر میباشد.



شکل 4 : منحنی لاس و دقت شبکه Custom Resnet

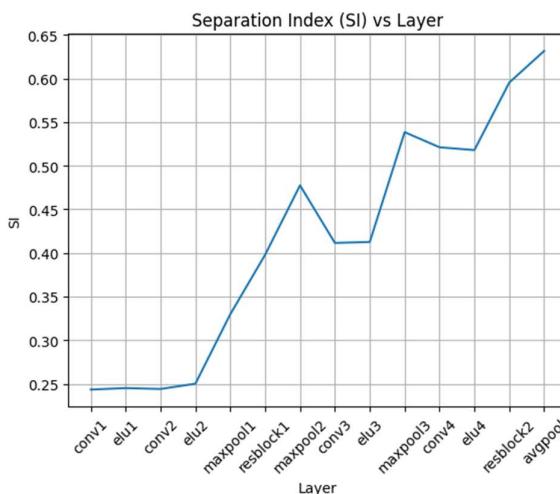
دقت نهایی بر روی **Test** و **VAL** به صورت زیر میباشد.

Best Validation Accuracy: 73.48%

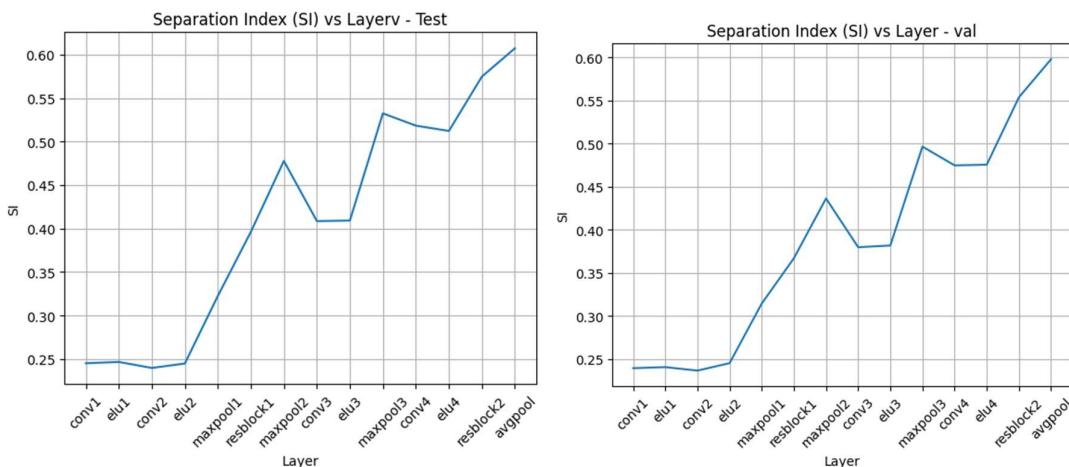
Test Accuracy of the final model: 73.29%

منحنی های **SI** تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح

زیر میباشد.



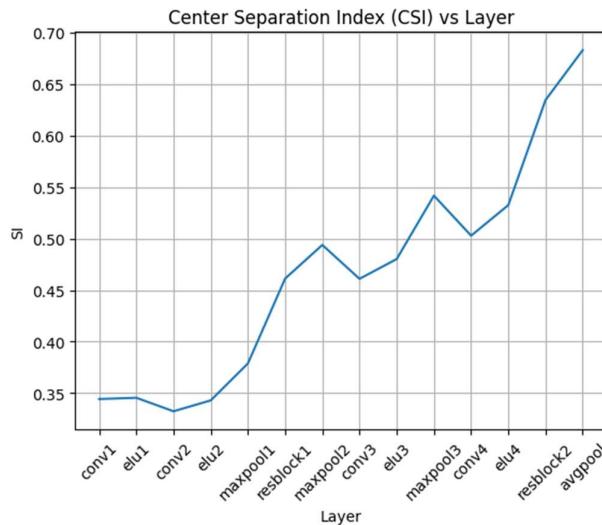
شکل 5 : منحنی **SI** برای داده های آموزشی



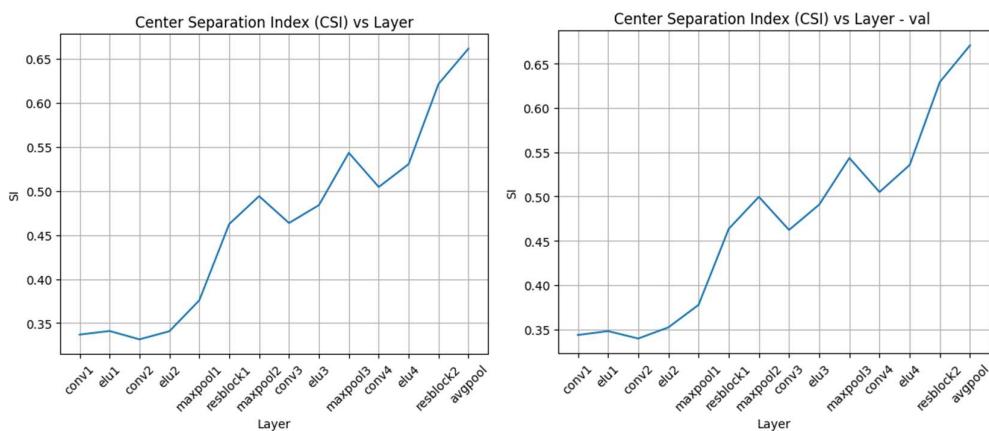
شکل 6 : منحنی **SI** بر حسب لایه ها بر روی **Custom Resnet** برای داده های ارزیابی و تست

مشاهده میشود رود کلی صعودی میباشد اما در **conv3** که یک چورایی **bottleneck** میباشد کاهش میابد.

منحنی های CSI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.



شکل 7: منحنی CSI برای داده های آموزشی



شکل 8 : منحنی CSI بر حسب لایه ها بر روی Custom Resnet برای داده های ارزیابی و تست

مقدادر دقيق و آنالیز نهایي و مقایسه ها در بخش آخر آورده شده است.

### 3. هدف از بکارگیری لایه های نرمالسازی و مقایسه آن ها

نرمال سازی تکنیکی است که در شبکه های عصبی برای بهبود پایداری آموزش و سرعت همگرایی استفاده می شود. این شامل تنظیم مقادیر ویژگی های ورودی یا فعال سازی نورون ها در یک لایه است تا آنها را در محدوده یا توزیع خاصی قرار دهند. عادی سازی به حل مسائلی مانند ناپدید شدن گرادیان ها کمک می کند و می تواند به آموزش پایدارتر و کارآمدتر شبکه های عصبی عمیق کمک کند.

سه نوع از لایه های نرمال سازی رایج در شبکه های عصبی وجود دارد که هر کدام اهداف کمی متفاوت دارند:

#### عادی سازی دسته ای (BatchNorm):

- هدف: نرمال سازی دسته ای برای فعال سازی یک لایه در یک مجموعه کوچک از نمونه های آموزشی اعمال می شود. میانگین و واریانس هر ویژگی را به طور مستقل روی دسته نرمال می کند.
- چگونه کار می کند: برای هر ویژگی، **BatchNorm** میانگین و انحراف استاندارد را در سراسر دسته محاسبه می کند و سپس با استفاده از این آمار، مقادیر را عادی می کند. همچنین پارامترهای قابل یادگیری (گاما و بتا) را معرفی می کند که به شبکه اجازه می دهد تا مقادیر نرمال شده را مقیاس و تغییر دهد.
- مزایا: **BatchNorm** به کاهش مشکل تغییر متغیر داخلی کمک می کند، آموزش را پایدار می کند، همگرایی را تسریع می کند و اغلب به عنوان نوعی منظم سازی (regularization) عمل می کند.

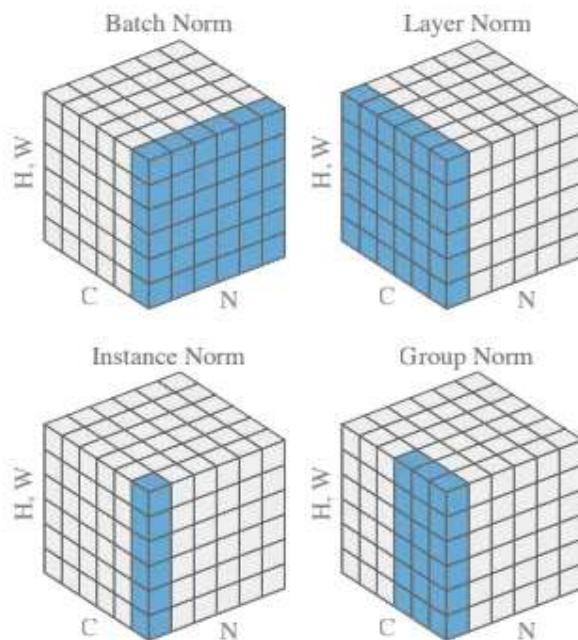
#### عادی سازی لایه (LayerNorm):

- هدف: برای فعال سازی یک لایه اعمال می شود، اما در تمام ویژگی ها برای یک مثال آموزشی نرمال می شود.
- چگونه کار می کند: برای هر مثال آموزشی، **LayerNorm** میانگین و انحراف استاندارد را در تمام ویژگی ها محاسبه می کند و مقادیر را بر این اساس نرمال می کند. مانند **BatchNorm**، پارامترهای قابل یادگیری (گاما و بتا) را برای تغییر مقیاس و تغییر معرفی می کند.
- مزایا: **LayerNorm** کمتر به اندازه دسته وابسته است و اغلب در شبکه های عصبی مکرر (RNN) که اندازه دسته می تواند متفاوت باشد استفاده می شود.

#### عادی سازی گروه (GroupNorm):

- هدف: عادی سازی گروه سازش بین **LayerNorm** و **BatchNorm** است. این ویژگی ها را به گروه ها تقسیم می کند و هر گروه را به طور مستقل عادی می کند.
- نحوه کار: **GroupNorm** ویژگی ها را به گروه ها تقسیم می کند و میانگین و انحراف استاندارد را به طور مستقل برای هر گروه محاسبه می کند. با این فرض عمل می کند که ویژگی های درون یک گروه بیشتر به هم مرتبط هستند.
- مزایا: **GroupNorm** حساسیت کمتری به اندازه دسته ای دارد و اغلب زمانی استفاده می شود که اندازه دسته های کوچکتر ترجیح داده می شود یا در سناریوهایی که آمار محاسبه شده برای گروه های کوچک معنادارتر است.

تصویر زیر ماهیت کلی این 3 متود را نشان میدهد.<sup>1</sup>



شکل 9 : مقایسه BN و LN و GN

به طور خلاصه، **GroupNorm** و **LayerNorm** تکنیک های عادی سازی هستند که برای فعال سازی لایه های شبکه عصبی به کار می روند. **BatchNorm** در یک دسته نرمال می شود، **GroupNorm** بین ویژگی ها برای هر مثال عادی می شود، و **LayerNorm** در بین گروه هایی از

<sup>1</sup> <https://arxiv.org/abs/1903.10520>

ویژگی ها عادی می شود. انتخاب لایه نرمال سازی به ویژگی ها و الزامات خاص معماری شبکه عصبی و ماهیت داده ها بستگی دارد.

## 4. پیاده سازی Custom Resnet با Batch Normalization

اصولا لایه های نرمال ساز قبل از اعمال activation function ها قرار میگیرند.

برای پیاده سازی مدل به صورت زیر عمل میشود.

```
class CustomResNetBlock(nn.Module):  
    def __init__(self, in_channels, out_channels):  
        super(CustomResNetBlock, self).__init__()  
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)  
        self.bn1 = nn.BatchNorm2d(out_channels)  # Batch Normalization after Convolution  
        self.elu = nn.ELU()  
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)  
        self.bn2 = nn.BatchNorm2d(out_channels)  # Batch Normalization after Convolution  
  
    def forward(self, x):  
        shortcut = x  
        x = self.conv1(x)  
        x = self.bn1(x)  # Apply Batch Normalization  
        x = self.elu(x)  
        x = self.conv2(x)  
        x = self.bn2(x)  # Apply Batch Normalization  
        x += shortcut  
        x = self.elu(x)  
        return x  
  
class CustomResNet_BN(nn.Module):  
    def __init__(self):  
        super(CustomResNet_BN, self).__init__()  
        # Initial convolutional layers  
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)  
        self.bn1 = nn.BatchNorm2d(32)  # Batch Normalization  
        self.elu1 = nn.ELU()  
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  
        self.bn2 = nn.BatchNorm2d(64)  # Batch Normalization  
        self.elu2 = nn.ELU()  
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)  
  
        # Residual blocks  
        self.resblock1 = CustomResNetBlock(64, 64)  
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)  
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
```

```

        self.bn3 = nn.BatchNorm2d(128)  # Batch Normalization
        self.elu3 = nn.ELU()
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(256)  # Batch Normalization
        self.elu4 = nn.ELU()
        self.resblock2 = CustomResNetBlock(256, 256)

        # Average pooling and fully connected layers
        self.avgpool = nn.AvgPool2d(kernel_size=3, stride=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, 10)
        self.softmax = nn.Softmax(dim=1)

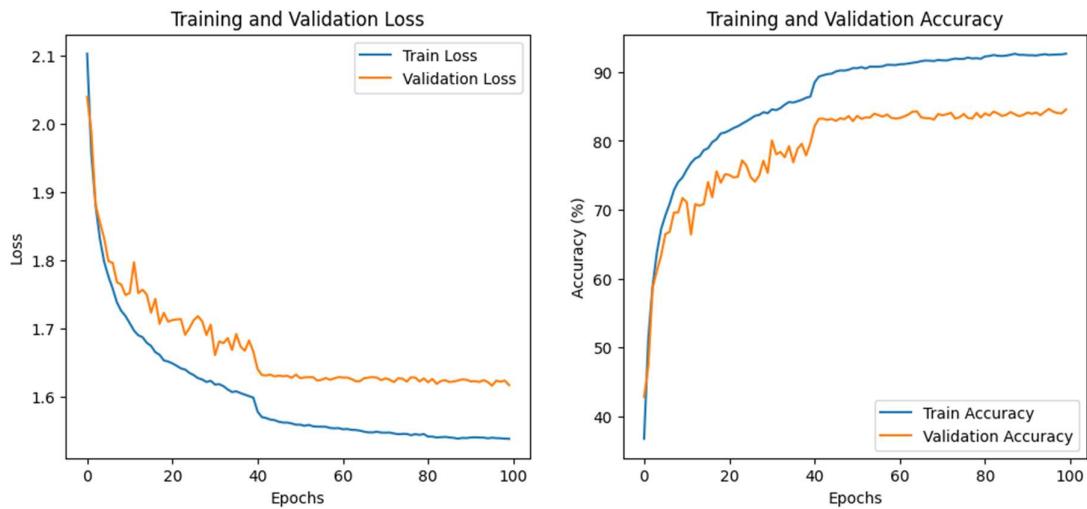
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)  # Apply Batch Normalization
        x = self.elu1(x)
        x = self.conv2(x)
        x = self.bn2(x)  # Apply Batch Normalization
        x = self.elu2(x)
        x = self.maxpool1(x)

        x = self.resblock1(x)
        x = self.conv3(x)
        x = self.bn3(x)  # Apply Batch Normalization
        x = self.elu3(x)
        x = self.maxpool2(x)

        x = self.conv4(x)
        x = self.bn4(x)  # Apply Batch Normalization
        x = self.elu4(x)
        x = self.maxpool3(x)
        x = self.resblock2(x)
        x = self.avgpool(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.softmax(x)
        return x

```

پس از پیاده سازی و آموزش مدل منحنی های دقت به صورت زیر میباشد.



شکل 10: منحنی لاس و دقت شبکه Custom Resnet - BN

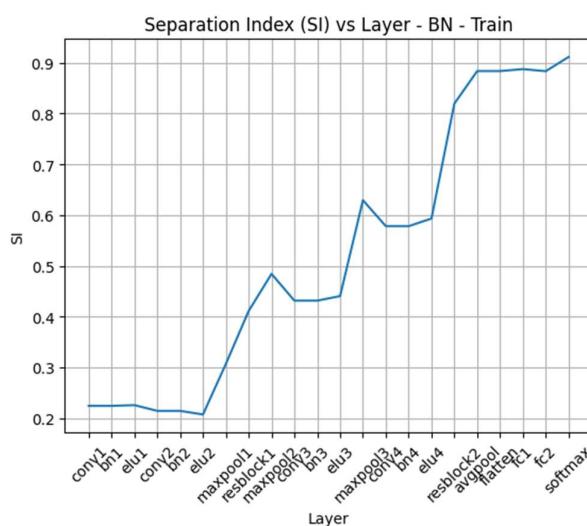
دقت نهایی بر روی Test و VAL به صورت زیر میباشد.

Best Validation Accuracy: 84.64%

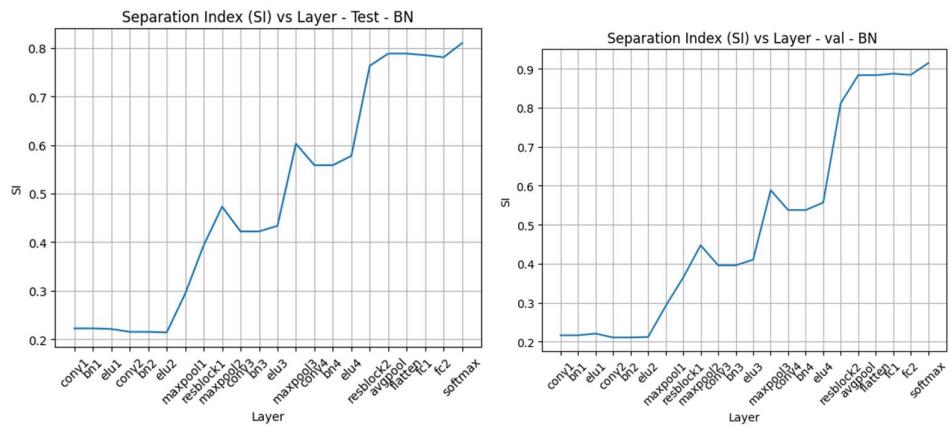
Test Accuracy of the final model: 85.67%

واضح است که اضافه کردن BN باعث generalization و converge شده است.

منحنی های SI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.

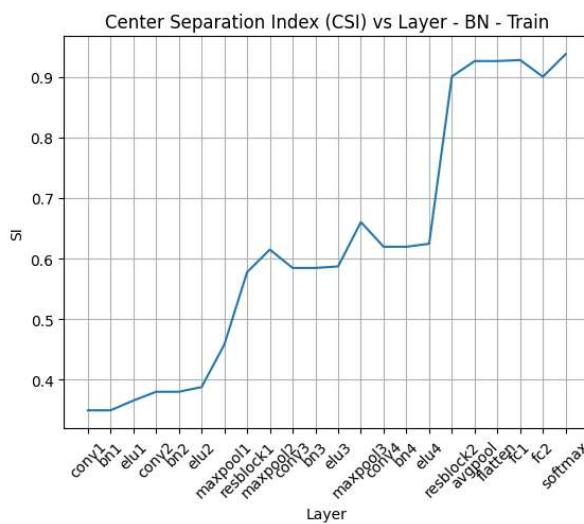


شکل 11: منحنی SI برای داده های آموزشی Custom Resnet - BN

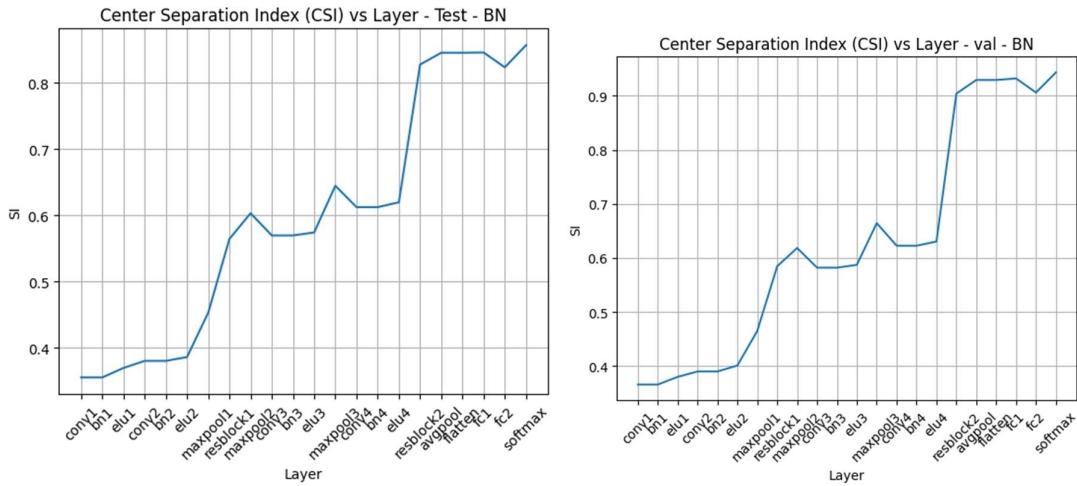


شکل 12 : منحنی SI بر حسب لایه ها بر روی Custom Resnet-BN برای داده های ارزیابی و تست

منحنی های CSI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.



شکل 13: منحنی CSI برای داده های آموزشی



شکل 14 : منحنی CSI بر حسب لایه ها بر روی Custom Resnet برای داده های ارزیابی و تست

مقادیر دقیق و آنالیز نهایی و مقایسه ها در بخش آخر آورده شده است.

## 5. پیاده سازی Custom Resnet با Layer Normalization

اصلًا لایه های نرمال ساز قبل از اعمال activation function ها قرار میگیرند.

برای پیاده سازی مدل به صورت زیر عمل میشود.

```
class CustomResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, size):
        super(CustomResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.ln1 = nn.LayerNorm([out_channels, size, size])
        self.elu = nn.ELU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.ln2 = nn.LayerNorm([out_channels, size, size])

    def forward(self, x):
        shortcut = x
        x = self.conv1(x)
        x = self.ln1(x)
        x = self.elu(x)
        x = self.conv2(x)
        x = self.ln2(x)
        x += shortcut
        return x
```

```

        x = self.elu(x)
        return x

class CustomResNet_LN(nn.Module):
    def __init__(self):
        super(CustomResNet_LN, self).__init__()
        # Initial convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.ln1 = nn.LayerNorm([32, 32, 32])
        self.elu1 = nn.ELU()
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.ln2 = nn.LayerNorm([64, 32, 32])
        self.elu2 = nn.ELU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Residual blocks
        self.resblock1 = CustomResNetBlock(64, 64, 16)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.ln3 = nn.LayerNorm([128, 16, 16])
        self.elu3 = nn.ELU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)
        self.ln4 = nn.LayerNorm([256, 8, 8])
        self.elu4 = nn.ELU()
        self.resblock2 = CustomResNetBlock(256, 256, 4)

        # Average pooling and fully connected layers
        self.avgpool = nn.AvgPool2d(kernel_size=3, stride=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(256, 256)
        self.ln5 = nn.LayerNorm(256)
        self.fc2 = nn.Linear(256, 10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.ln1(x)
        x = self.elu1(x)
        x = self.conv2(x)
        x = self.ln2(x)
        x = self.elu2(x)
        x = self.maxpool1(x)

        x = self.resblock1(x)
        x = self.conv3(x)
        x = self.ln3(x)
        x = self.elu3(x)

```

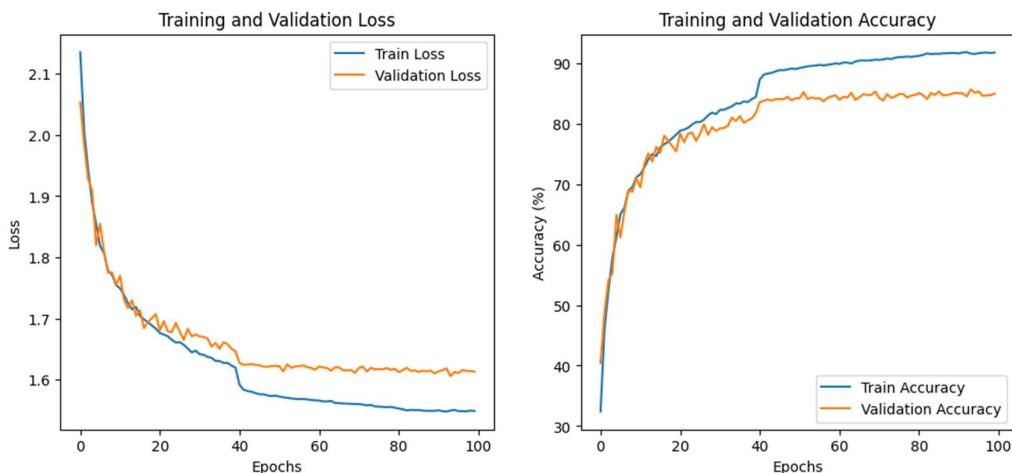
```

x = self.maxpool2(x)

x = self.conv4(x)
x = self.ln4(x)
x = self.elu4(x)
x = self.maxpool3(x)
x = self.resblock2(x)
x = self.avgpool(x)
x = self.flatten(x)
x = self.fc1(x)
x = self.ln5(x)
x = self.fc2(x)
x = self.softmax(x)
return x

```

پس از پیاده سازی و آموزش مدل منحنی های دقت به صورت زیر میباشد.



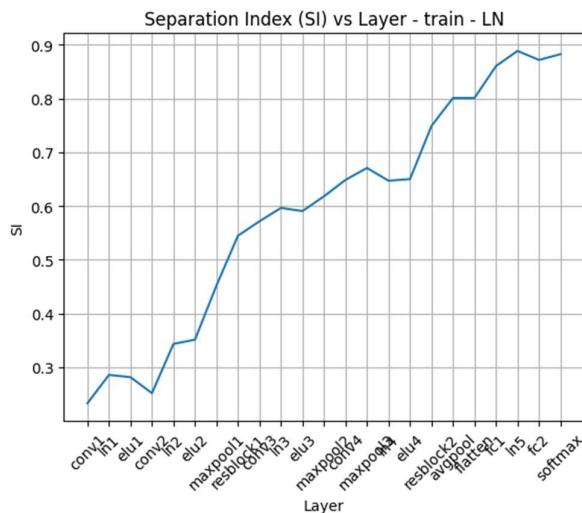
شکل 15: منحنی لاس و دقت شبکه Custom Resnet - LN4

دقت نهایی بر روی **Test** و **VAL** به صورت زیر میباشد.

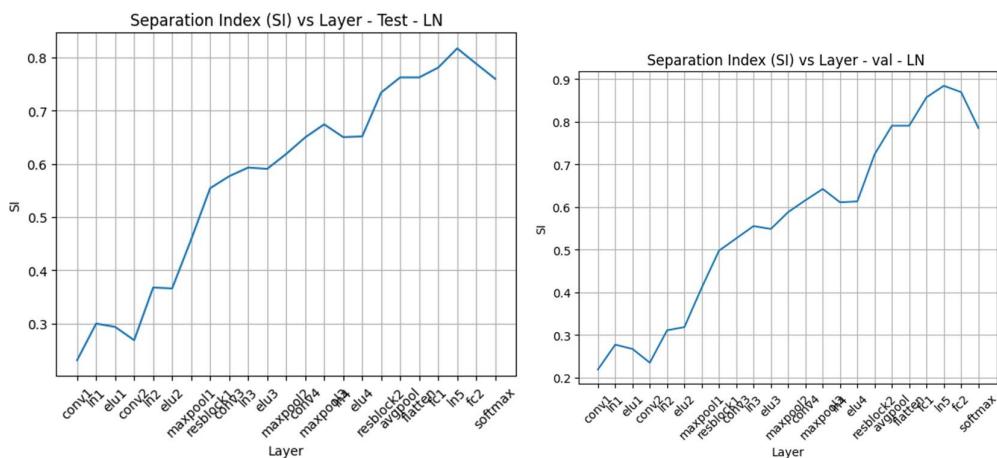
Best Validation Accuracy: 85.62%

Test Accuracy of the final model: 85.24%

منحنی های **SI** تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.

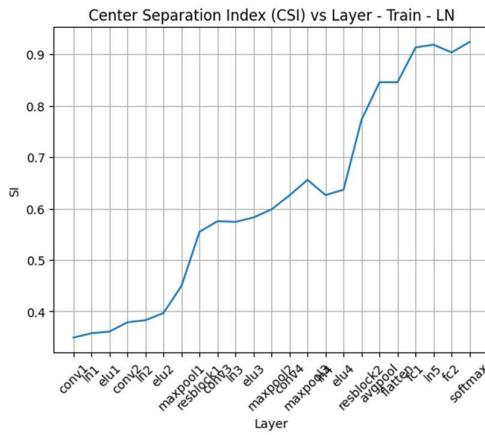


شکل 16: منحنی SI برای داده های آموزشی

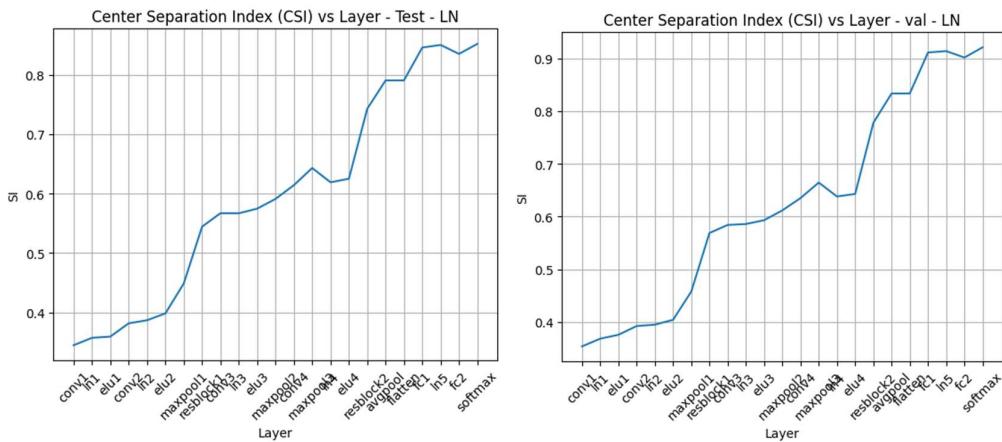


شکل 17: منحنی SI بر حسب لایه ها بر روی داده های ارزیابی و تست

منحنی های CSI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.



شکل 18: منحنی CSI برای داده های آموزشی



شکل 19 : منحنی CSI بر حسب لایه ها بر روی Custom Resnet - LN برای داده های ارزیابی و تست

مقداری دقیق و آنالیز نهایی و مقایسه ها در بخش آخر آورده شده است.

## 6. پیاده سازی Layer Normalization با Custom Resnet

اصولاً لایه های نرمال ساز قبل از اعمال activation function قرار میگیرند.

برای پیاده سازی مدل به صورت زیر عمل میشود.

```
class CustomResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, num_groups=32):
        super(CustomResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.gn1 = nn.GroupNorm(num_groups, out_channels)
        self.elu = nn.ELU()
```

```

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.gn2 = nn.GroupNorm(num_groups, out_channels)

    def forward(self, x):
        shortcut = x
        x = self.conv1(x)
        x = self.gn1(x)
        x = self.elu(x)
        x = self.conv2(x)
        x = self.gn2(x)
        x += shortcut
        x = self.elu(x)
        return x

class CustomResNet_GN(nn.Module):
    def __init__(self, num_groups=32):
        super(CustomResNet_GN, self).__init__()
        # Initial convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.gn1 = nn.GroupNorm(num_groups, 32)
        self.elu1 = nn.ELU()
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.gn2 = nn.GroupNorm(num_groups, 64)
        self.elu2 = nn.ELU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Residual blocks
        self.resblock1 = CustomResNetBlock(64, 64, num_groups)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.gn3 = nn.GroupNorm(num_groups, 128)
        self.elu3 = nn.ELU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)
        self.gn4 = nn.GroupNorm(num_groups, 256)
        self.elu4 = nn.ELU()
        self.resblock2 = CustomResNetBlock(256, 256, num_groups)

        # Average pooling and fully connected layers
        self.avgpool = nn.AvgPool2d(kernel_size=3, stride=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, 10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.gn1(x)

```

```

x = self.elu1(x)
x = self.conv2(x)
x = self.gn2(x)
x = self.elu2(x)
x = self.maxpool1(x)

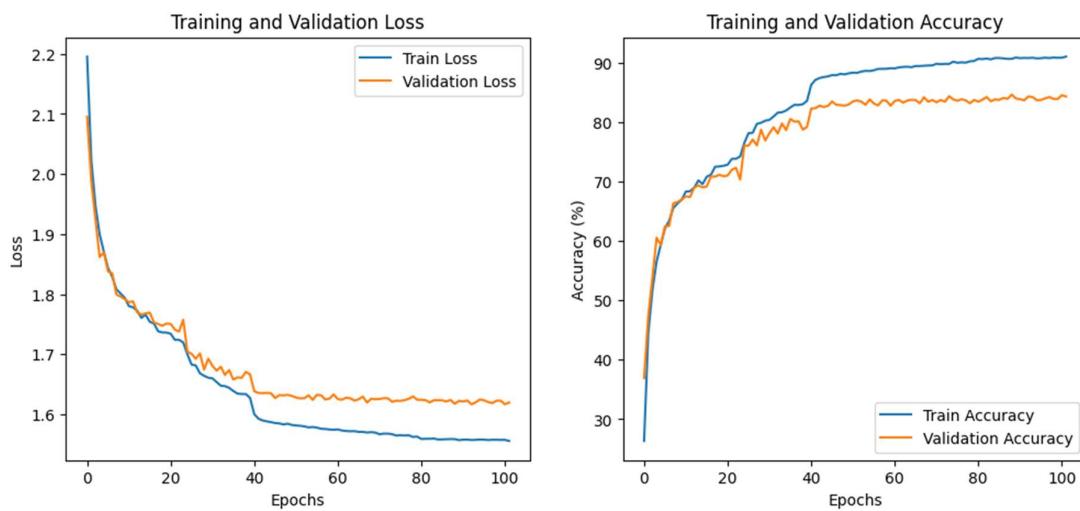
x = self.resblock1(x)
x = self.conv3(x)
x = self.gn3(x)
x = self.elu3(x)
x = self.maxpool2(x)

x = self.conv4(x)
x = self.gn4(x)
x = self.elu4(x)
x = self.maxpool3(x)
x = self.resblock2(x)
x = self.avgpool(x)
x = self.flatten(x)
x = self.fc1(x)
x = self.fc2(x)
x = self.softmax(x)

return x

```

پس از پیاده سازی و آموزش مدل منحنی های دقت به صورت زیر میباشد.

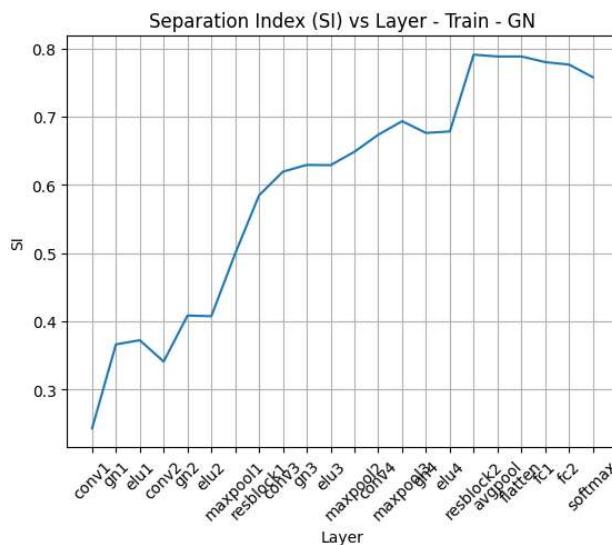


شکل 20: منحنی لاس و دقت شبکه Custom Resnet - GN

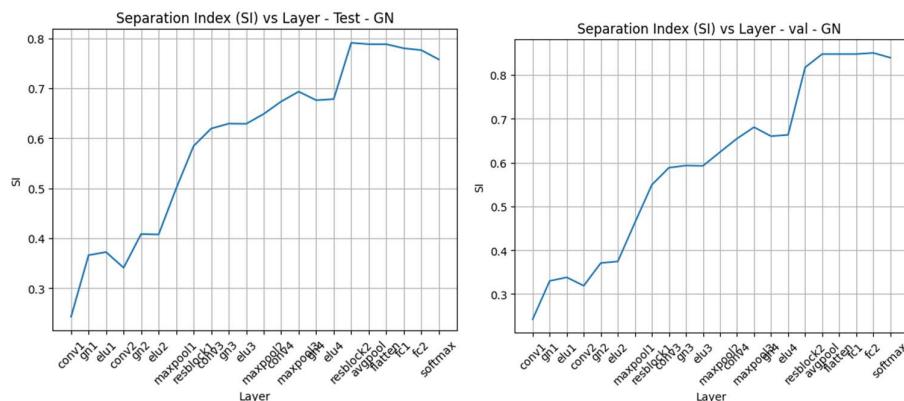
دقت نهایی بر روی **Test** و **VAL** به صورت زیر میباشد.

Best Validation Accuracy: 84.62%  
 Test Accuracy of the final model: 84.86%

منحنی های SI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.

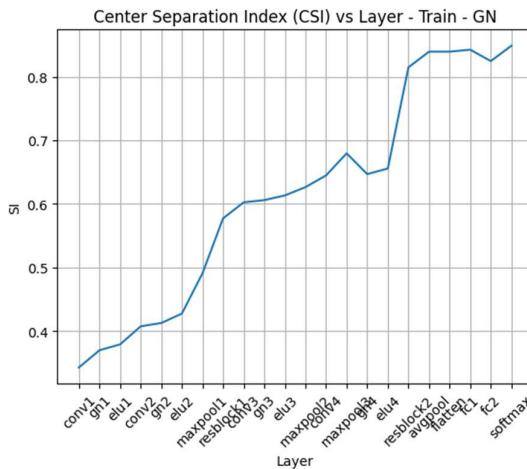


شکل 21: منحنی SI برای داده های آموزشی Custom Resnet - GN

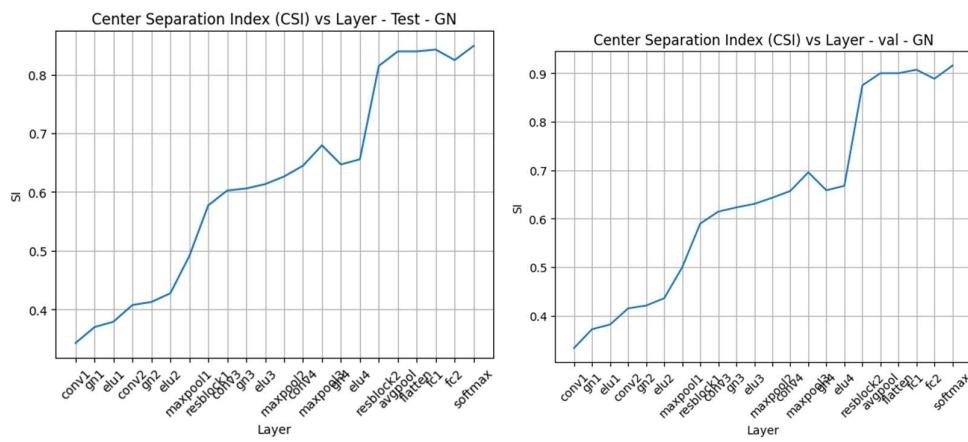


شکل 22 : منحنی SI بر حسب لایه ها بر روی Custom Resnet-GN برای داده های ارزیابی و تست

منحنی های CSI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.



شکل 23: منحنی CSI برای داده های آموزشی



شکل 24 : منحنی CSI بر حسب لایه ها بر روی Custom Resnet - GN برای داده های ارزیابی و تست

مقدادیر دقیق و آنالیز نهایی و مقایسه ها در بخش آخر آورده شده است.

## 7. مقایسه بین BN و LN و GN

جدول زیر مقدادیر دقیق و مقدادیر CSI و SI اخیرین لایه feature در چهار حالت مختلف را نشان میدهد.

جدول 1: مقایسه با normalization های متفاوت

Model	Val ACC	Test ACC	SI Avgpool – Val	CSI Avg Pool – Val	SI Avgpool - Test	CSI Avgpool - Test
Custom Resnet	73.48%	73.29%	0.5977	0.6705	0.6071	0.6615
Custom Resnet - BN	84.64	85.67	0.883	0.929	0.788	0.845
Custom Resnet - GN	84.62	84.86	0.84739	0.899	0.787	0.839
Custom Resnet - LN	85.62	85.24	0.79	0.833	0.7622	0.790

با توجه به جدول فوق دقت روی **VAL** برای حالت **BN** بالاتر است اما **LN** عملکرد بهتری دارد در مورد میزان جداسازی بر روی تست و **val** هم همانطور که مشاهده میشود **BN** بهترین بوده و **GN** بعد از آن بهترین عملکرد را دارد.

نتایج تقریبا این را تایید میکند که **GN** عملکردی بینابینی با **LN** و **BN** دارد

## 8. حذف Connection

برای پیاده سازی به راحتی **skip connection** را از مدل حذف میکنیم.  
مدل نهایی به صورت زیر میشود.

```
class CustomResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(CustomResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.elu = nn.ELU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.elu(x)
        x = self.conv2(x)
        x = self.elu(x)
        return x

class CustomResNet_NSC(nn.Module):
    def __init__(self):
        super(CustomResNet_NSC, self).__init__()
        # Initial convolutional layers
```

```

    self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
    self.elu1 = nn.ELU()
    self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
    self.elu2 = nn.ELU()
    self.maxpool1 = nn.MaxPool2d(kernel_size=2)

    # Residual blocks
    self.resblock1 = CustomResNetBlock(64, 64)
    self.maxpool2 = nn.MaxPool2d(kernel_size=2)
    self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
    self.elu3 = nn.ELU()
    self.maxpool3 = nn.MaxPool2d(kernel_size=2)
    self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
    self.elu4 = nn.ELU()
    self.resblock2 = CustomResNetBlock(256, 256)

    # Average pooling and fully connected layers
    self.avgpool = nn.AvgPool2d(kernel_size=3, stride=2)
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(256, 256)
    self.fc2 = nn.Linear(256, 10)
    self.softmax = nn.Softmax(dim=1)

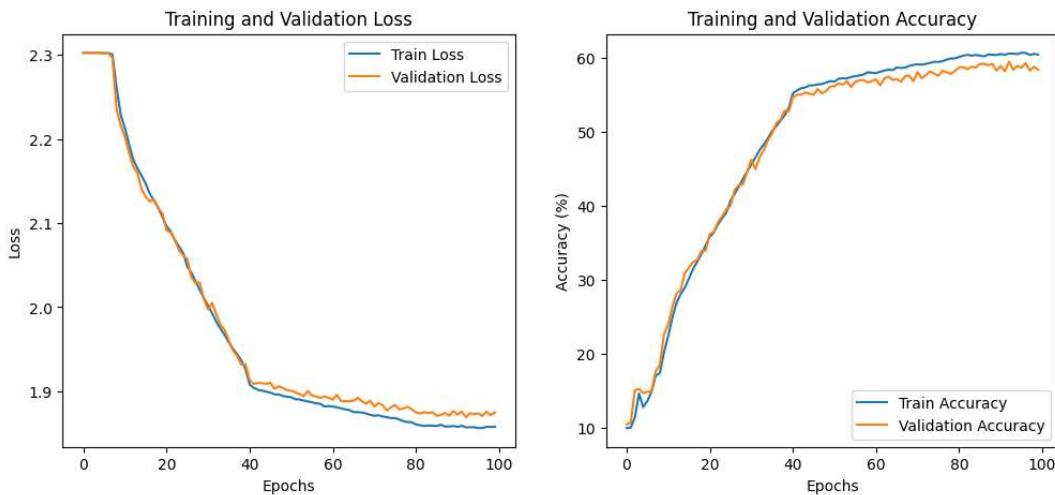
  def forward(self, x):
    x = self.conv1(x)
    x = self.elu1(x)
    x = self.conv2(x)
    x = self.elu2(x)
    x = self.maxpool1(x)

    x = self.resblock1(x)
    x = self.conv3(x)
    x = self.elu3(x)
    x = self.maxpool2(x)

    x = self.conv4(x)
    x = self.elu4(x)
    x = self.maxpool3(x)
    x = self.resblock2(x)
    x = self.avgpool(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.fc2(x)
    x = self.softmax(x)
    return x

```

پس از پیاده سازی و آموزش مدل منحنی های دقت به صورت زیر میباشد.



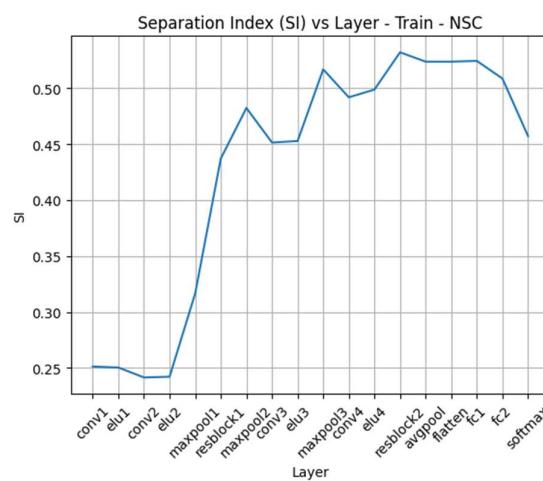
شکل 25 : منحنی لاس و دقت شبکه Custom Resnet - NSC

دقت نهایی بر روی Test و VAL به صورت زیر میباشد.

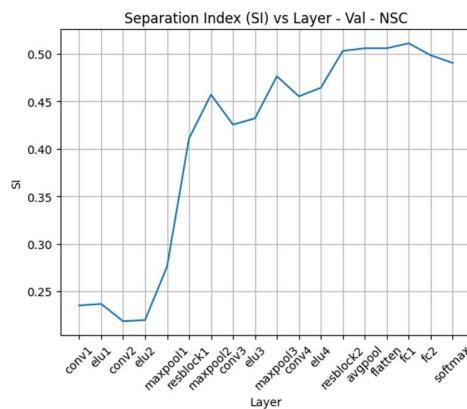
Best Validation Accuracy: 59.42%

Test Accuracy of the final model: 59.75%

منحنی های SI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.

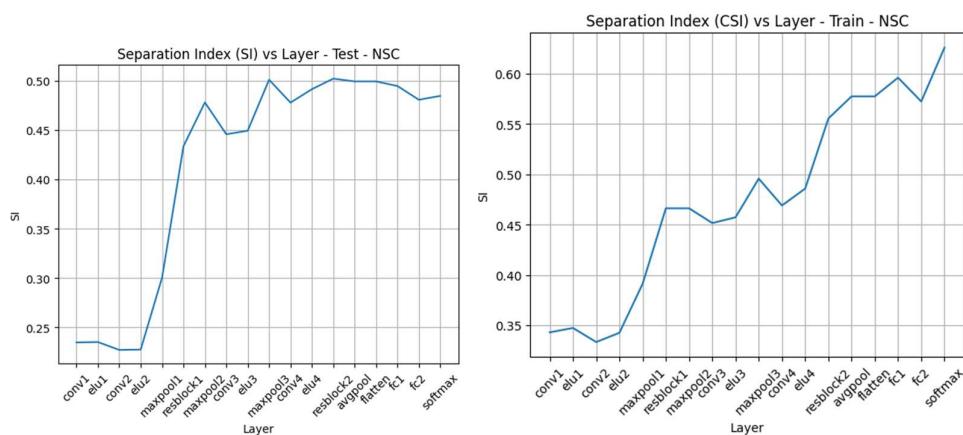


شکل 26: منحنی SI برای داده های آموزشی Custom Resnet - NSC

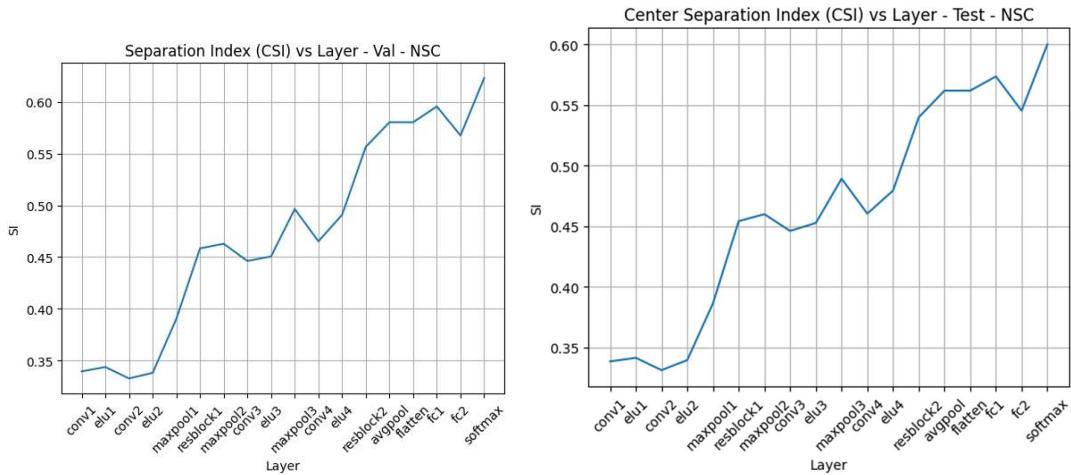


شکل 27 : منحنی SI بر حسب لایه ها بر روی Custom Resnet- NSC برای داده های ارزیابی و تست

منحنی های CSI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.



شکل 28: منحنی CSI برای داده های آموزشی Custom Resnet - NSC



شکل 29 : منحنی CSI بر حسب لایه ها بر روی NSC برای داده های ارزیابی و تست

جدول زیر مقایسه حالت با و بدون skip connection میباشد.

جدول 2 : مقایسه با skip connection و بدون Skip connection

Model	Val ACC	Test ACC	SI Avgpool – Val	CSI Avg Pool – Val	SI Avgpool – Test	CSI Avgpool – Test
Custom Resnet	73.48%	73.29%	0.5977	0.6705	0.6071	0.6615
Custom Resnet – No skip Connection	59.42	59.75	0.5058	0.58019	0.499	0.5618

پر واضح است که حذف Resnet ها مزیت های Skip Connection که مزیت های skip connection محسوسی در مدل و توانایی های آن میشود.

## 9. جایگذاری لایه ای با Conv های 1\*1

2 جا میتواند گزینه منطقی ای برای کاهش kernel از 3 به 1 باشد :

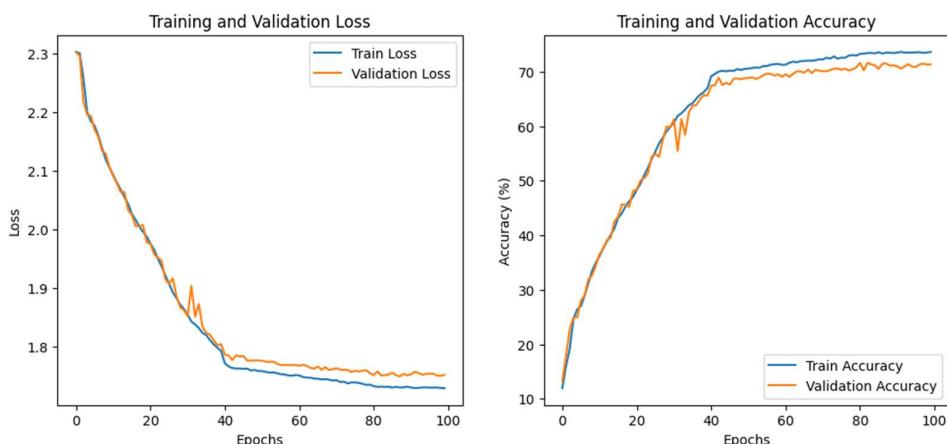
- قبل از بلوک های residual : قرار دادن کانولوشن  $1 \times 1$  قبل از بلوک های residual (به عنوان self.conv3 میتواند به کاهش تعداد کانال های ورودی به بلوک های

کمک کند. این می تواند در مدیریت پیچیدگی محاسباتی مفید باشد، به خصوص اگر تعداد کانال ها زیاد شود. این به عنوان یک لایه گلوگاه عمل می کند و قبل از پردازش در لایه های عمیق تر، ابعاد را کاهش می دهد.

- قبل از لایه های **Fully Connected** : از طرف دیگر، جایگزینی `self.conv4` با یک کانولوشن 1 می تواند برای کاهش عمق نقشه ویژگی قبل از اینکه `flatten` شود و به لایه های `fully connected` منتقل شود، مفید باشد که اگر با `overfit` سر و کار داشته باشیم، می تواند به ویژه مفید باشد.

در اینجا رویکرد دوم را اعمال میکنیم و نتایج به شرح زیر میباشد ( در کد `kernel size` برای `conv4` برابر 1 میشود )

پس از پیاده سازی و آموزش مدل منحنی های دقت به صورت زیر میباشد.



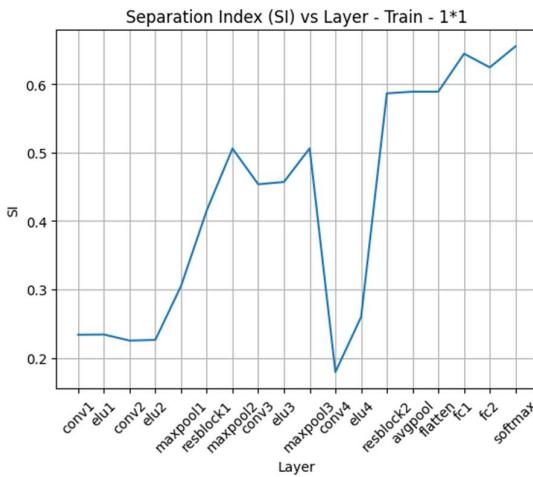
شکل 30 : منحنی لاس و دقت شبکه 1

دقت نهایی بر روی `Test` و `VAL` به صورت زیر میباشد.

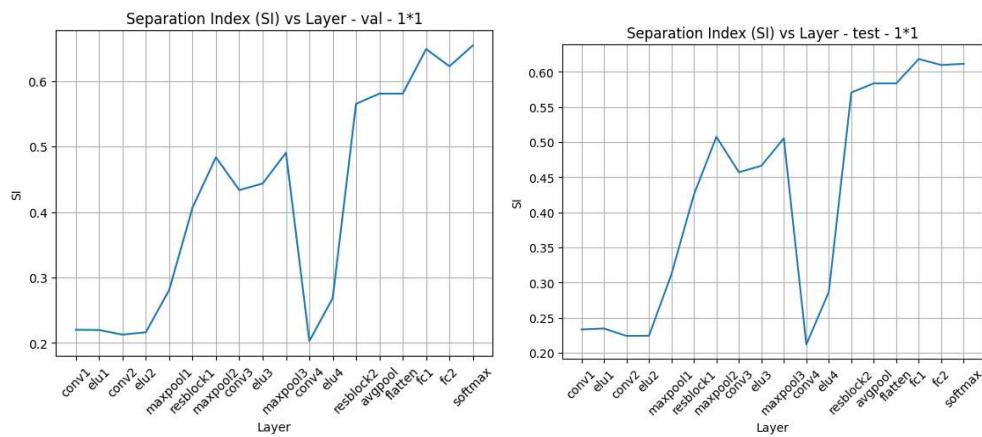
Best Validation Accuracy: 71.62%

Test Accuracy of the final model: 72.18%

منحنی های `SI` تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.

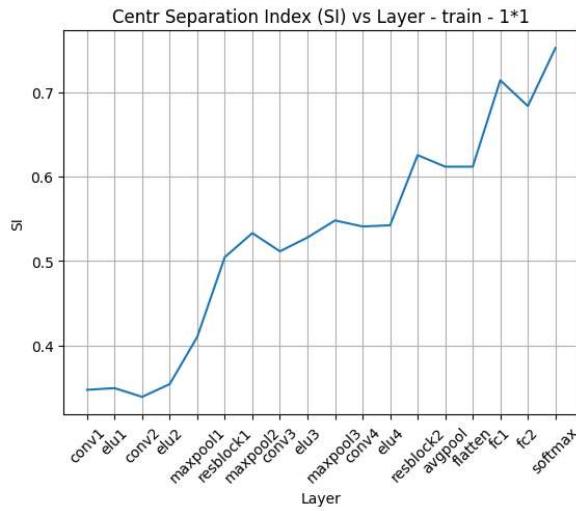


شکل 31: منحنی SI برای داده های آموزشی Custom Resnet - 1

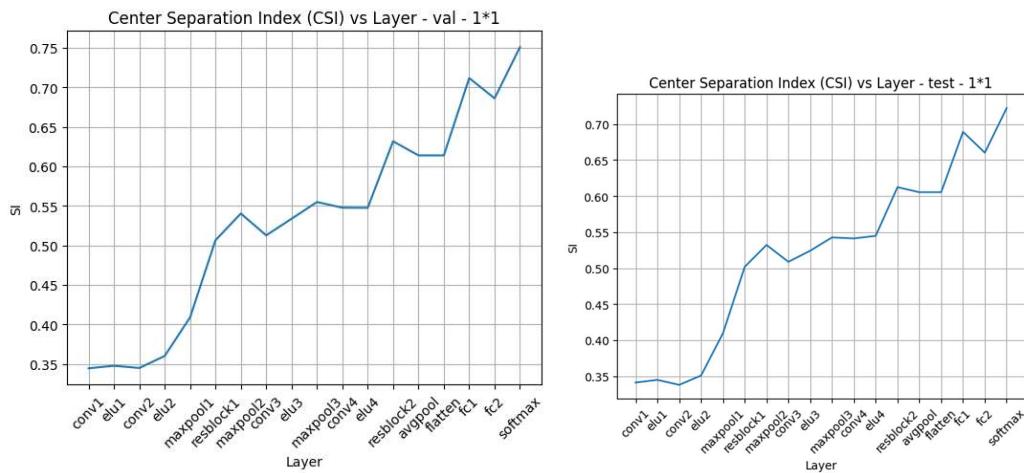


شکل 32 : منحنی SI بر حسب لایه های بر روی Custom Resnet- 1 برای داده های ارزیابی و تست

منحنی های CSI تمامی لایه ها بر روی داده های آموزش و تست و ارزیابی بر حسب هر لایه نیز به شرح زیر میباشد.



شکل 33: منحنی CSI برای داده های آموزشی Custom Resnet -1



شکل 34 : منحنی CSI بر حسب لایه ها بر روی 1 Custom Resnet – ارزیابی و تست

نکته حایز توجه در اینجا افت شدید مقدار SI در لایه است که Kernel Size آن به 1 کاهش یافته است که با توجه به 1 شدن آن میتواند طبیعی باشد اما در اما بعد از آن روند صعودی میباشد.

جدول زیر مقایسه این حالت با حالت قبل میباشد.

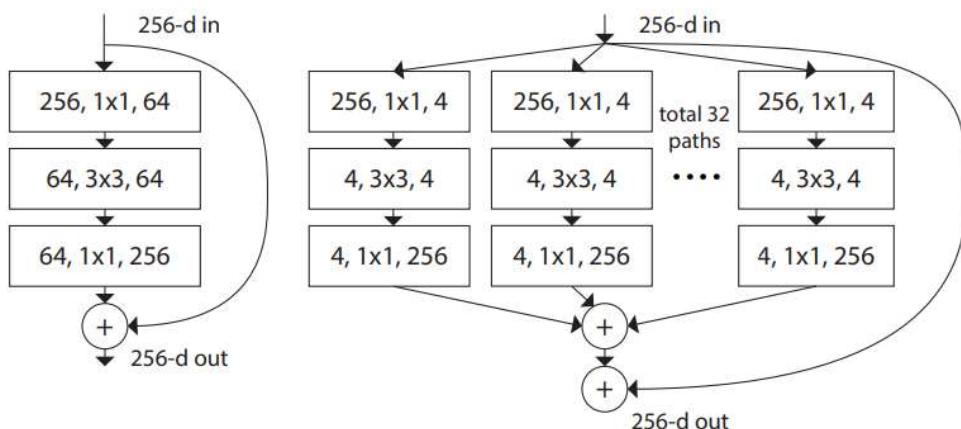
جدول 3

Model	Val ACC	Test ACC	SI Avgpool – Val	CSI Avg Pool – Val	SI Avgpool - Test	CSI Avgpool - Test
Custom Resnet	73.48%	73.29%	0.5977	0.6705	0.6071	0.6615
Custom Resnet – cnv4 to 1*1	71.62	72.18	0.580	0.6139	0.5837	0.6054

عملکرد کلی شبکه در این حالت تقریبا ( کمی پایین تر ) برابر با حالت اولیه میباشد.

## 10. پیاده سازی Resnext

شکل زیر در حالت کلی متود resnext را نشان میدهد.



شکل 35 : الگوریتم Resnext

جهت پیاده سازی کافیست برای path های 2 و 4 به صورت زیر کد بخش residual را تغییر داد ( skip connection ) و جمع کردن خروجی ها با هم و در نهایت اعمال residual تکرار

برای path 2 :

```
class GroupedResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(GroupedResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.elu = nn.ELU()
```

```

    self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

def forward(self, x):
    shortcut = x

    # Path 1
    x1 = self.conv1(x)
    x1 = self.elu(x1)
    x1 = self.conv2(x1)

    # Path 2
    x2 = self.conv1(x)
    x2 = self.elu(x2)
    x2 = self.conv2(x2)

    # Combine paths
    x = x1 + x2

    # Final combination with shortcut
    x += shortcut
    x = self.elu(x)

    return x

```

همچنین برای path های بالاتر داریم :

```

class GroupedResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(GroupedResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.elu = nn.ELU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

    def forward(self, x):
        shortcut = x

        # Paths
        paths = []
        for i in range(4):
            path = self.conv1(x)
            path = self.elu(path)
            path = self.conv2(path)
            paths.append(path)

        # Combine paths
        x = sum(paths)

        # Final combination with shortcut

```

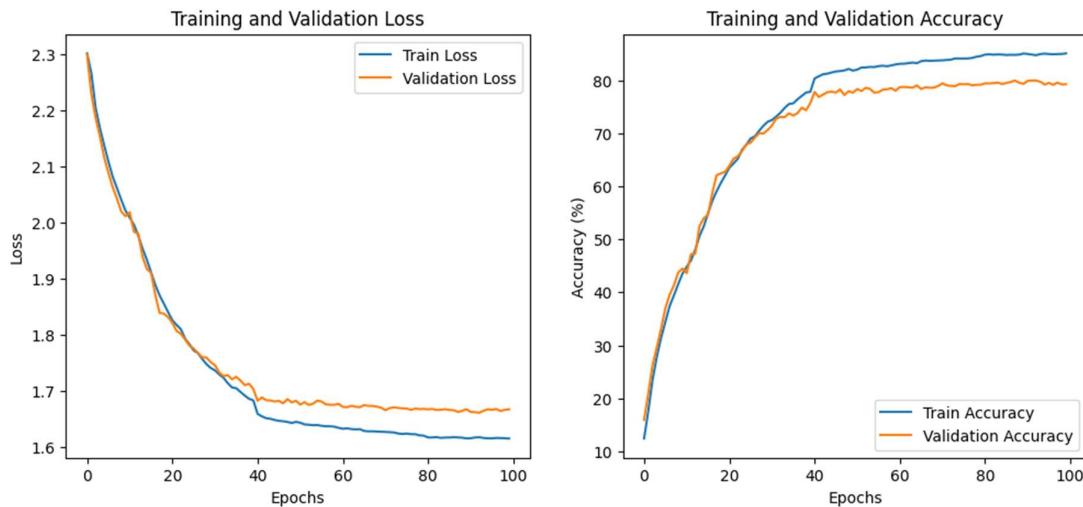
```

x += shortcut
x = self.elu(x)

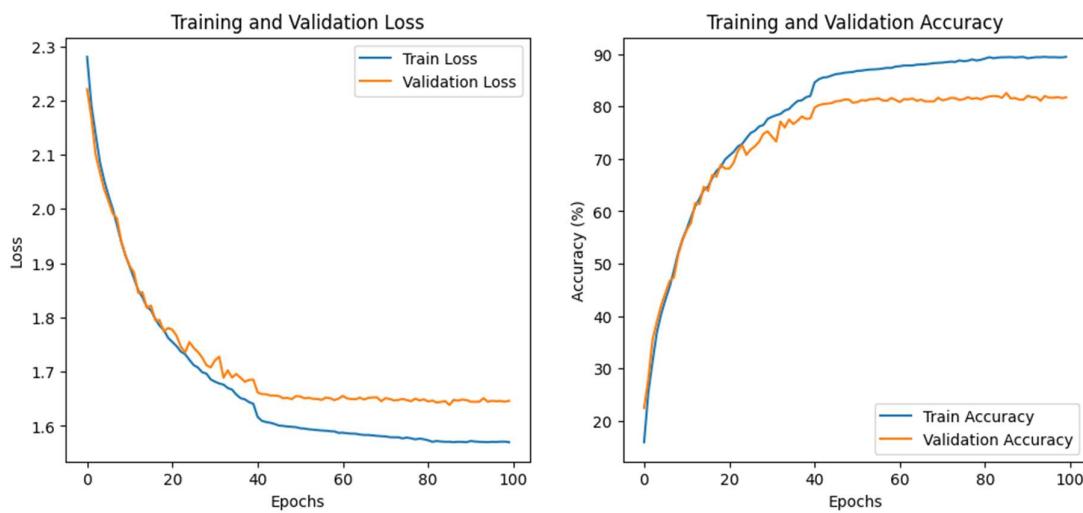
return x

```

در نهایت پس از آموزش داریم :

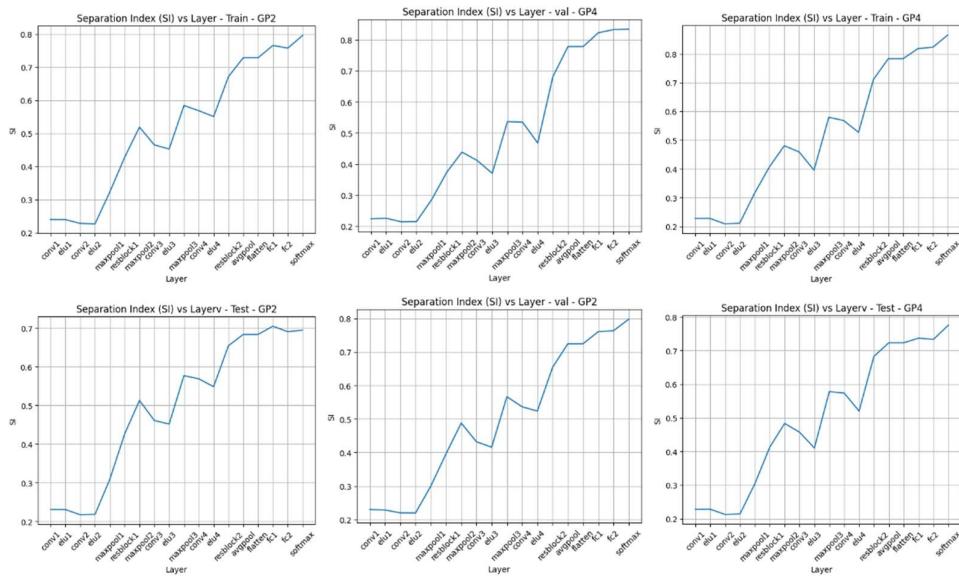


شکل 36 : منحنی دقت و لاس برای 2



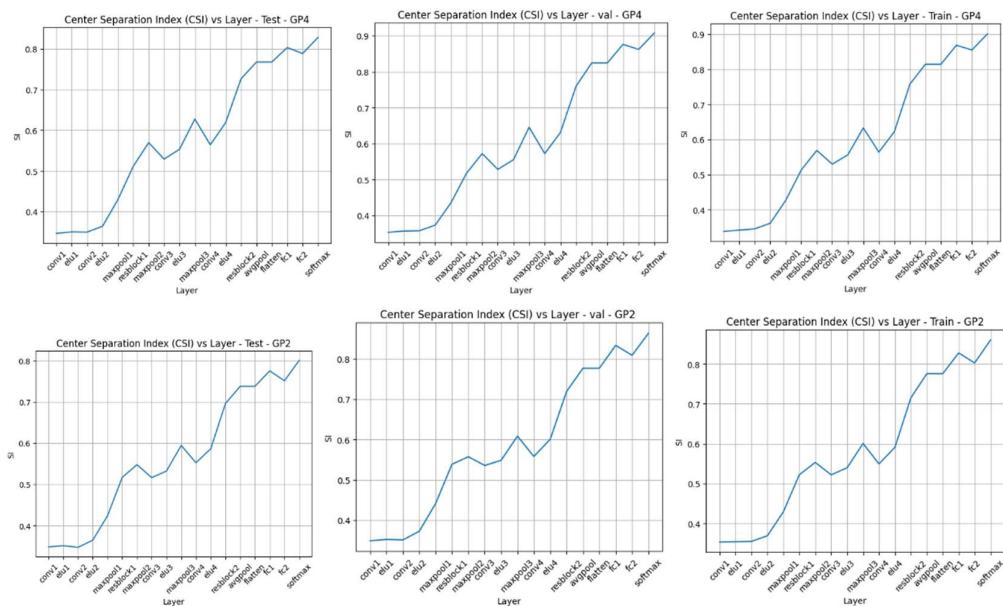
شکل 37: منحنی دقت و لاس برای 4

منحنی های SI در این حالات به صورت زیر میباشد ( GP2 و GP4 )



شکل 38 : منحنی SI برای GP4 و GP2

منحنی های SIC نیز در این حالات به صورت زیر میباشد ( GP4 و GP2 )



شکل 39: منحنی CSI برای GP2 و GP4

جدول زیر مقایسه کلی این 3 حالت را انجام میدهد.

Model	Val ACC	Test ACC	SI Avgpool – Val	CSI Avg Pool – Val	SI Avgpool - Test	CSI Avgpool - Test
Custom Resnet	73.48%	73.29%	0.5977	0.6705	0.6071	0.6615
Custom Resnet – GP2	79.94	80.05	0.724	0.7766	0.6834	0.7376
Custom Resnet – GP4	82.54	82.82	0.7784	0.824	0.7231	0.7675

همانطور که انتظار می‌رود روش **group conv residual** های عادی باشد و اضافه کردم **path** میتواند به دقت بهتر و جداسازی بهتر فیچر ها کمک کند.

## 11. بهینه سازی مدل داده شده

کد زیر نسخه بهینه شده مدل داده شده میباشد که پارامتر های آن کمتر میباشد و دارای **skip connection** میباشد.

این بخش جزو تمارین نمیباشد و من صرفا مدل **BN** داده شده بدون **custom** داده شده بود **BN** و .. را بهینه و بهتر کردم که دقت بالاتر رفته است.

```
Class CustomResNet(nn.Module):
    def __init__(self):
        super(CustomResNet, self).__init__()

        # Initial layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Residual block 1
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        # 1x1 convolution for matching dimensions in the first residual connection
        self.match_conv1 = nn.Conv2d(64, 64, kernel_size=1, stride=1, padding=0)

        # Residual block 2
        self.conv5 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        # 1x1 convolution for matching dimensions in the second residual connection
        self.match_conv2 = nn.Conv2d(64, 128, kernel_size=1, stride=2, padding=0)

        # Residual block 3
        self.conv6 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
```

```

        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv7 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        # 1x1 convolution for matching dimensions in the third residual connection
        self.match_conv3 = nn.Conv2d(128, 256, kernel_size=1, stride=2, padding=0)

        # Final layers
        self.avg_pool = nn.AvgPool2d(kernel_size=3)
        self.fc1 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = F.elu(self.conv1(x))
        x = F.elu(self.conv2(x))
        x = self.pool1(x)

        # Residual block 1
        identity = x
        x = F.elu(self.conv3(x))
        x = F.elu(self.conv4(x))
        x += identity
        x = F.elu(x)

        # Residual block 2
        identity = self.match_conv2(x) # Adjust dimensions of identity
        x = F.elu(self.conv5(x))
        x = self.pool2(x)
        x += identity
        x = F.elu(x)

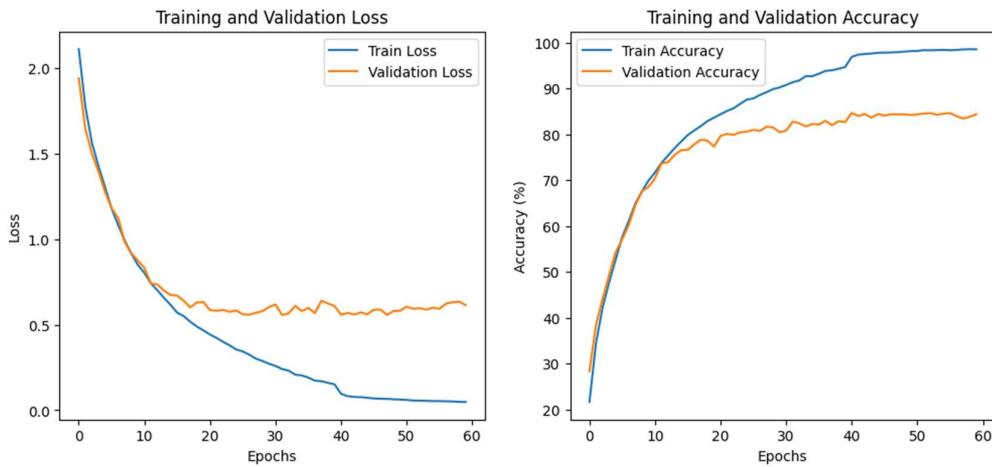
        # Residual block 3
        identity = self.match_conv3(x) # Adjust dimensions of identity
        x = F.elu(self.conv6(x))
        x = self.pool3(x)
        x = F.elu(self.conv7(x))
        x += identity
        x = F.elu(x)

        # Final layers
        x = self.avg_pool(x)
        x = torch.flatten(x, 1)
        x = F.elu(self.fc1(x))
        x = self.fc2(x)

    return x

```

در این حالت فرایند آموزش به صورت زیر انجام شده است :



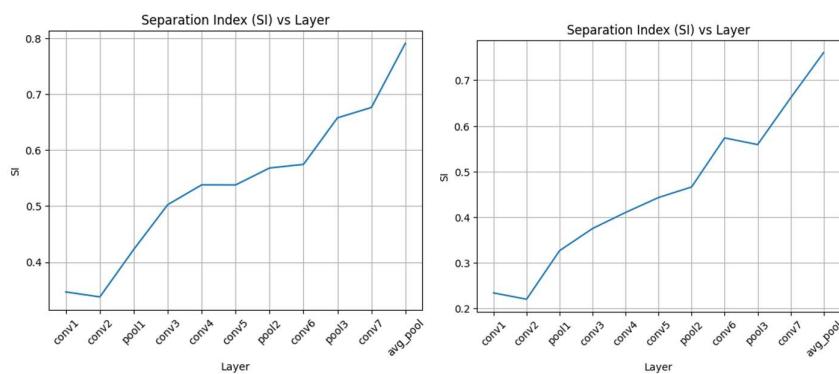
شکل 40 : فرایند آموزش با مدل پهنه شده

همانطور که مشاهده میشود دقت بسیار بهتر شده است.

Best Validation Accuracy: 84.70%

Test Accuracy of the final model: 85.37%

شکل زیر SI و CSI را برای train در این حالت نمایش میدهد.



شکل 41 : منحنی SI و CSI برای مدل پهنه شده

تمامی کد های این سکشن در پیوست ارسال شده که با **Q1** اغاز میشود.

## سوال دوم : شبکه تشخیص اشیا

### 1. اماده سازی دیتاست جدید

با توجه به توضیحات داده شده الگوریتم به صورت زیر میباشد.

ابتدا یک تابع `parser` نوشته میشود تا فایل لیبل را خوانده و مقادیر `bbox` را برگرداند.

```
# Function to parse the bounding box from the XML file
def parse_annotation(xml_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    bboxes = []
    for obj in root.iter('object'):
        if obj.find('name').text == 'cat':
            xmlbox = obj.find('bndbox')
            xmin = int(float(xmlbox.find('xmin').text))
            ymin = int(float(xmlbox.find('ymin').text))
            xmax = int(float(xmlbox.find('xmax').text))
            ymax = int(float(xmlbox.find('ymax').text))
            bboxes.append([xmin, ymin, xmax, ymax])
    return bboxes
```

یک تابع جهت محاسبه `IOU` نوشته میشود.

```
# Function to calculate IOU
def get_iou(bb1, bb2):
    x_left = max(bb1[0], bb2[0])
    y_top = max(bb1[1], bb2[1])
    x_right = min(bb1[2], bb2[2])
    y_bottom = min(bb1[3], bb2[3])

    if x_right < x_left or y_bottom < y_top:
        return 0.0

    intersection_area = (x_right - x_left) * (y_bottom - y_top)
    bb1_area = (bb1[2] - bb1[0]) * (bb1[3] - bb1[1])
    bb2_area = (bb2[2] - bb2[0]) * (bb2[3] - bb2[1])
    iou = intersection_area / float(bb1_area + bb2_area - intersection_area)
    return iou
```

حال الگوریتم کلی را پیاده میکنیم یک عکس را بخواند و `Cv2` با `selective search` با اعمال شود و انهایی که `IOU` بالای نیم روی `ground truth` های گربه (نه سگ ها) دارد به عنوان کلاس مثبت و در غیر این صورت به عنوان کلاس منفی در فایل هایی ذخیره میشود.

```
# Directories
image_dir = 'Asirra: cat vs dogs/'
positive_dir = 'new_dataset/positive_class'
negative_dir = 'new_dataset/negative_class'
os.makedirs(positive_dir, exist_ok=True)
```

```

os.makedirs(negative_dir, exist_ok=True)

# Iterate through the images and XMLs
for file in tqdm(os.listdir(image_dir)):
    if file.endswith('.jpg'):
        base_filename = file.split('/')[-1].split(".jpg")[0]
        img_path = os.path.join(image_dir, file)
        xml_path = os.path.join(image_dir, base_filename + '.xml')

        # Read image and parse XML
        img = cv2.imread(img_path)
        bboxes = parse_annotation(xml_path)

        # Apply Selective Search
        ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()
        ss.setBaseImage(img)
        ss.switchToSelectiveSearchFast()
        ssresults = ss.process()

        # Iterate through region proposals
        for i, rect in enumerate(ssresults):
            x, y, w, h = rect
            proposal_bbox = [x, y, x+w, y+h]

            # Handle case when bboxes is empty
            if bboxes:
                max_iou = max([get_iou(proposal_bbox, gt_bbox) for gt_bbox in bboxes])
            else:
                max_iou = 0

            # Classify and save the proposal region
            if max_iou >= 0.5:
                save_path = os.path.join(positive_dir, f'{base_filename}_{i}-proposal-positive.jpg')
            else:
                save_path = os.path.join(negative_dir, f'{base_filename}_{i}-proposal-negative.jpg')
            cropped_img = img[y:y+h, x:x+w]
            cv2.imwrite(save_path, cropped_img)

```

به کمک کد زیر نمونه هایی از دیتاست + و - نمایش داده میشود.

```

def show_random_images(directory, title, num_images=16):
    # Get all image files
    all_files = [f for f in listdir(directory) if isfile(join(directory, f))]

```

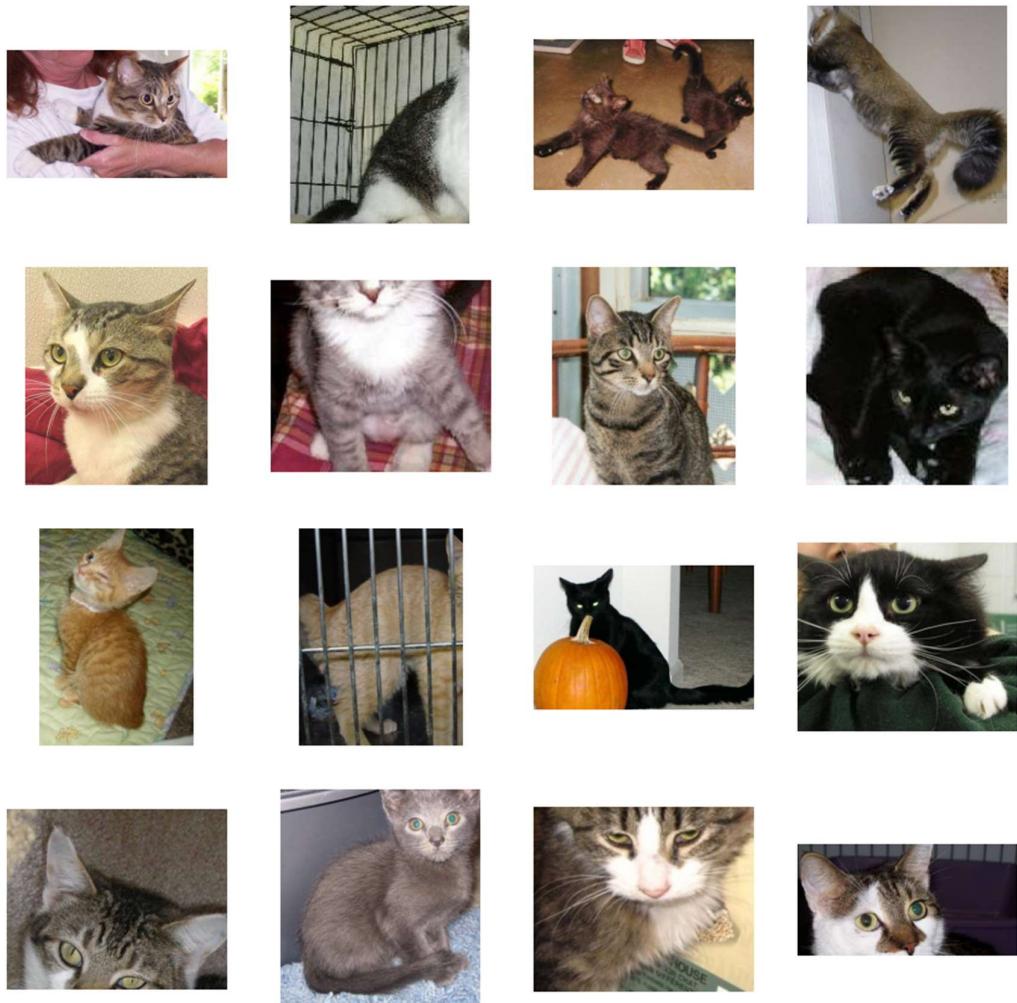
```
selected_files = random.sample(all_files, min(num_images, len(all_files)))

# Create a grid plot
plt.figure(figsize=(12, 12))
for i, file in enumerate(selected_files, 1):
    img = cv2.imread(join(directory, file))
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.subplot(4, 4, i)
    plt.imshow(img)
    plt.axis('off')
plt.suptitle(title)
plt.show()

positive_dir = 'new_dataset/positive_class'
negative_dir = 'new_dataset/negative_class'

# Display images and print the count
show_random_images(positive_dir, 'Positive Class Images')
show_random_images(negative_dir, 'Negative Class Images')
```

Positive Class Images



شکل 42 : نمونه هایی از عکس های کلاس مثبت

### Negative Class Images



شکل 43 : نمونه هایی از عکس های کلاس منفی

اما همانطور که مشخص دیتاست موجود بالانس نمیباشد و مقادیر آن ها به صورت زیر میباشد.

```
positive_count = len([name for name in listdir(positive_dir) if isfile(join(positive_dir, name))])
negative_count = len([name for name in listdir(negative_dir) if isfile(join(negative_dir, name))])

print("positive_count : " , positive_count , "And Negative_count : " , negative_count)

positive_count : 15871 And Negative_count : 1099888
```

دو رویکرد میتوان داشت برای حل بالанс نبودن دیتا : وزن دار کردن کلاس + یا انتخاب حدود 15 هزار عکس رندوم از کلاس منفی

هر 2 رویکرد فوق میتواند مناسب باشد در اینجا رویکرد دوم انجام شده است  
به کمک کد زیر به صورت رندوم 15 هزار عکس از کلاس منفی `select` میشود.

```
positive_count = 15871

# Path to the existing negative class directory and the new directory for balanced data
existing_negative_dir = 'new_dataset/negative_class'
new_negative_dir = 'new_dataset/negative_class_new'
os.makedirs(new_negative_dir, exist_ok=True)

# Get all negative image files
all_negative_files = [f for f in listdir(existing_negative_dir) if
isfile(join(existing_negative_dir, f))]
selected_negative_files = random.sample(all_negative_files, positive_count) # Selecting
random files

# Copy selected files to the new directory
for file in tqdm(selected_negative_files):
    src_path = join(existing_negative_dir, file)
    dst_path = join(new_negative_dir, file)
    shutil.copy2(src_path, dst_path)
```

## 2- لود کردن دیتاست جدید

به کمک کد زیر از روی دایرکتوری موجود دیتاست را لود کرده و با توجه به ورودی `mobilenet` باید عکس ها را تماما 224\*224 کرد.

```
# Define image transformations: resize to 224x224 and convert to tensor
transform = Compose([
    Resize((224, 224)),
    ToTensor()
])

# Create a custom dataset
dataset = ImageFolder(root='new_dataset/', transform=transform)
```

کد زیر تعدادی از عکس های دیتاست و لیبل متناظر آن ها را نمایش میدهد.

```
random.seed(42)

num_images = len(dataset)

# Choose 32 random indices
random_indices = random.sample(range(num_images), 32)

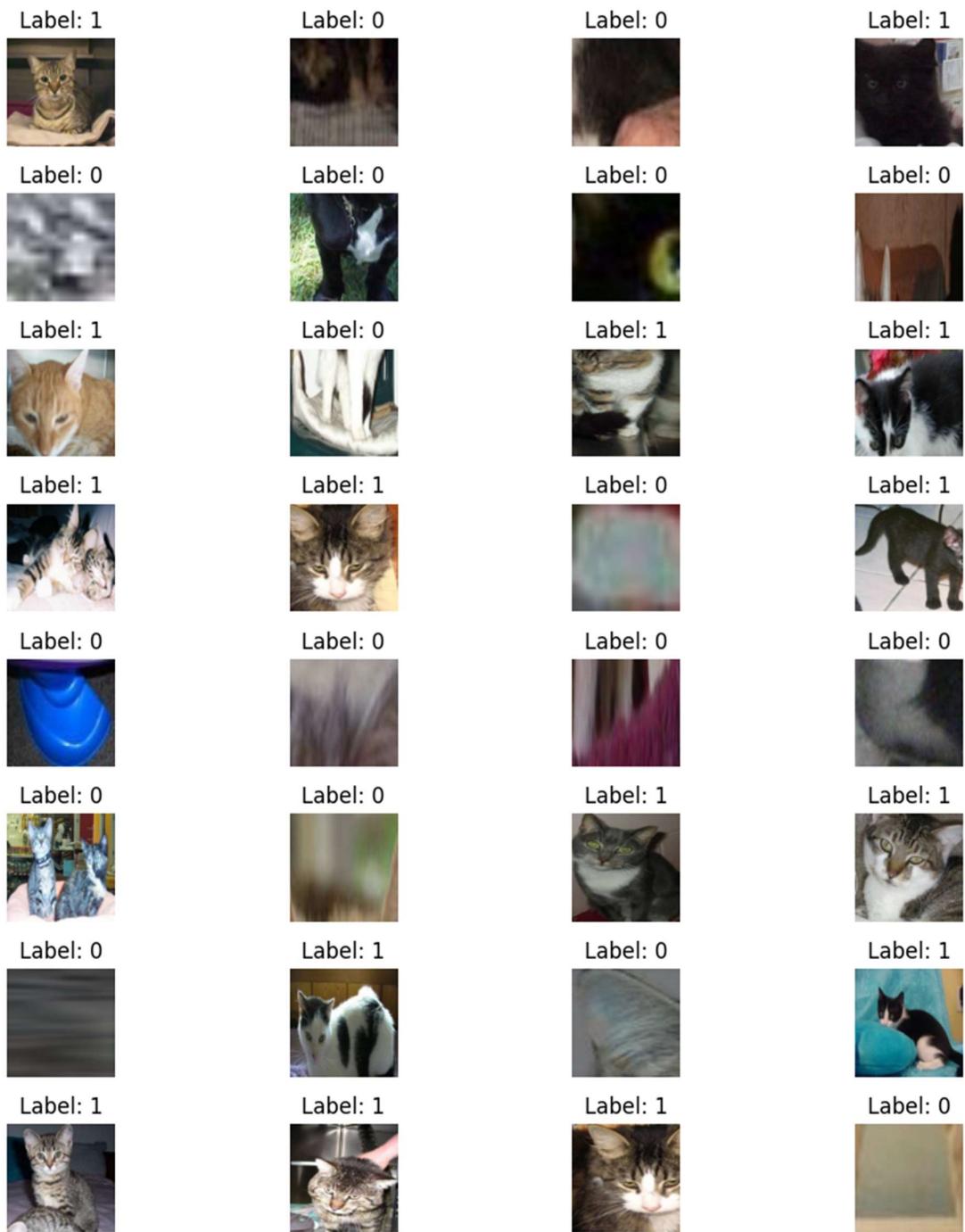
# Create a subplot with 8 rows and 4 columns
fig, axes = plt.subplots(8, 4, figsize=(10, 10))

# Flatten the axes array to iterate easily
axes = axes.flatten()

# Loop through the random indices
for i, idx in enumerate(random_indices):
    image, label = dataset[idx]
    image = image.numpy().transpose((1, 2, 0))
    scaled_image = image * 255

    # Plot the image
    axes[i].imshow(scaled_image.astype('uint8'))
    axes[i].set_title(f"Label: {label}")
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```



شکل 44 : 32 نمونه از دیتاست و لیبل های آن ها

در ادامه نیز **dataloader** ها را تشکیل داده و 30 درصد داده ها به عنوان داده های ارزیابی جدا میکنیم.

```

# Split the dataset into training and validation sets (80% train, 20% val)
train_size = int(0.7 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Create DataLoaders for the train and validation sets
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)

```

### 3. لود مدل و آموزش شبکه

شبکه از پیش آموزش دیده mobilenet را لود کرده و بخش feature را freeze کرده و لایه classifier به آن اضافه میکنیم و با توجه به اینکه خروجی 1280 فیچر است یه لایه میانی 256 تایی میگذاریم و سپس از 256 به 2 و در نهایت softmax میاید. به جای آن در کد زیر با sigmoid قرار داده شده است که معادل همان sigmoid با تک نورون در خروجی است و فرقی ندارد.

```

# Load the pre-trained MobileNet model
model = models.mobilenet_v2(pretrained=True)

# Freeze all the parameters in the feature network
for param in model.features.parameters():
    param.requires_grad = False

# Modify the classifier for 2 class classification
num_features = model.classifier[1].in_features
print(num_features)
model.classifier = nn.Sequential(
    nn.Dropout(0.2),
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, 2),
    nn.Softmax(dim=1)    # Sigmoid activation for binary classification
)

# Check if GPU is available and move the model to GPU if it is
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
model = model.to(device)
print("Model Loaded")

```

در ادامه لوب ترین را نوشته و منحنی های ACC و loss را نیز رسم میکنیم.

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.005)

# Decay LR by a factor of 0.1 every 4 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.3)

def train_model(model, criterion, optimizer, scheduler, num_epochs=5):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    train_acc_history = []
    val_acc_history = []

    train_loss_history = []
    val_loss_history = []

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()    # Set model to training mode
                data_loader = train_loader
            else:
                model.eval()    # Set model to evaluate mode
                data_loader = val_loader

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in tqdm(data_loader):
                inputs = inputs.to(device)
                labels = labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()
```

```

# Forward
with torch.set_grad_enabled(phase == 'train'):
    outputs = model(inputs)
    _, preds = torch.max(outputs, 1)
    loss = criterion(outputs, labels)

# Backward + optimize only if in training phase
if phase == 'train':
    loss.backward()
    optimizer.step()

# Statistics
running_loss += loss.item() * inputs.size(0)
running_corrects += torch.sum(preds == labels.data)

if phase == 'train':
    scheduler.step()

epoch_loss = running_loss / len(data_loader.dataset)
epoch_acc = running_corrects.double() / len(data_loader.dataset)

print('{} Loss: {:.4f} Acc: {:.4f}'.format(
    phase, epoch_loss, epoch_acc))

# Deep copy the model
if phase == 'val' and epoch_acc > best_acc:
    best_acc = epoch_acc
    best_model_wts = copy.deepcopy(model.state_dict())

if phase == 'train':
    train_acc_history.append(epoch_acc)
    train_loss_history.append(epoch_loss)
else:
    val_acc_history.append(epoch_acc)
    val_loss_history.append(epoch_loss)

print()

time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(
    time_elapsed // 60, time_elapsed % 60))
print('Best val Acc: {:.4f}'.format(best_acc))

# Load best model weights
model.load_state_dict(best_model_wts)

# Plot accuracy and loss
plt.figure(figsize=(12, 4))

```

```

plt.subplot(1, 2, 1)
plt.plot([h.detach().cpu() for h in train_acc_history], label='Train Acc')
plt.plot([h.detach().cpu() for h in val_acc_history], label='Val Acc')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot([h for h in train_loss_history], label='Train Loss')
plt.plot([h for h in val_loss_history], label='Val Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()

return model

```

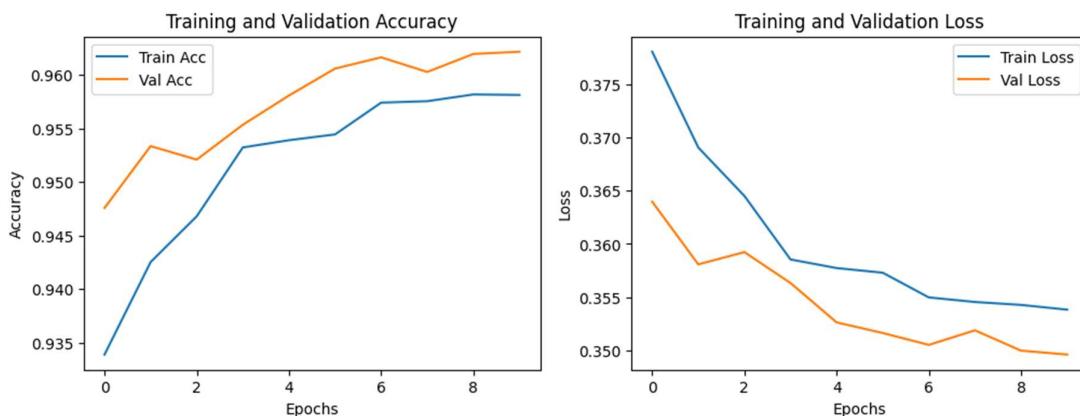
```

# Train the model
trained_model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=10)

# Save the model
torch.save(trained_model.state_dict(), 'mobilenet_detect_cat.pth')

```

منحنی های دقت و لاس به صورت زیر میباشد:



شکل 45 : منحنی دقت و لاس مدل mobilenet

برای تست بر روی تصاویر داده شده بدین صورت عمل میشود که تمام ROI ها را گرفته و به مدل داده اگر مقدار خروجی مدل کلاس 1 باشد ( 0.5 بالاتر ) در این صورت آن IOU را انتخاب کرده تا رسم کند.

برای خروجی الگوریتم تا 10 تا پیدا کند متوقف میشود ( چون ران تایم آن طول میکشد و شکل هم خیلی شلوغ میشد )

```
# Read the test image
img = cv2.imread('test1.jpg')

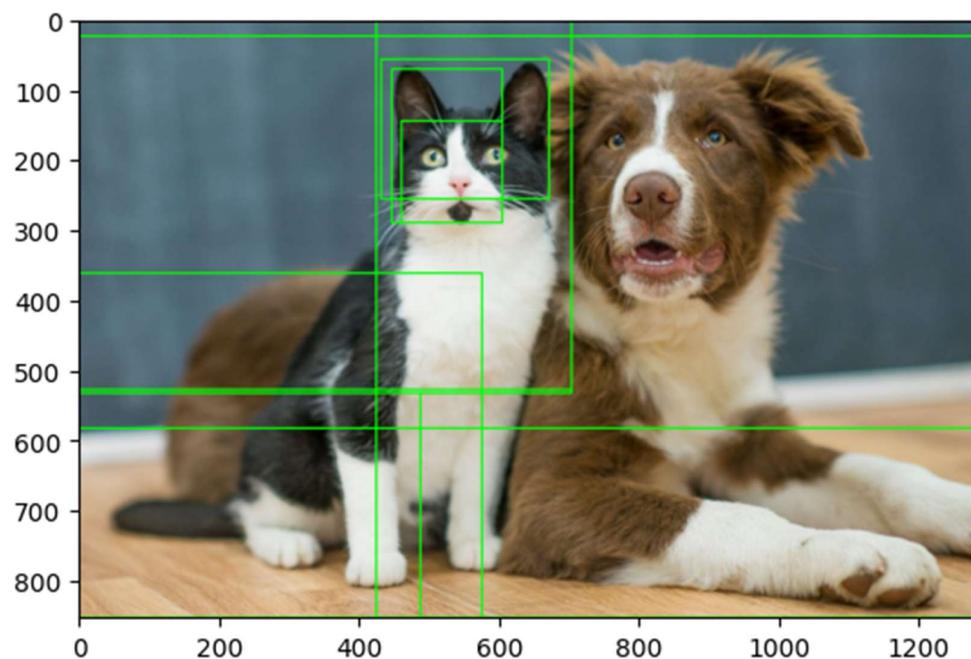
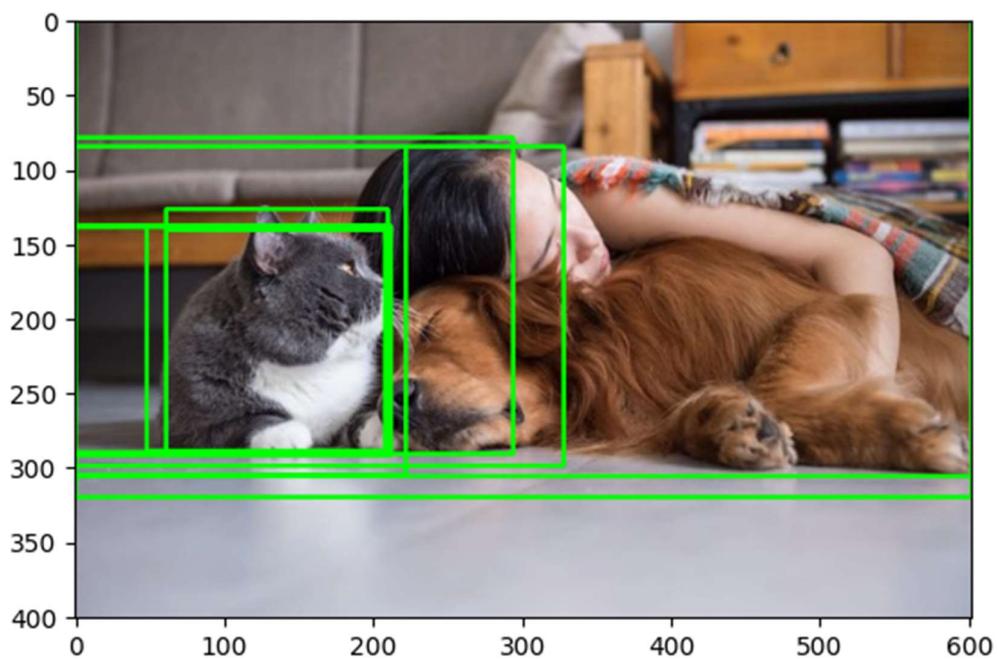
# Create a selective search segmentation object
ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()
ss.setBaseImage(img)
ss.switchToSelectiveSearchFast()
rects = ss.process()

# Transform for input image
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

i = 0
# Iterate over the regions and classify
for x, y, w, h in rects:
    roi = img[y:y+h, x:x+w]
    roi = transform(roi)
    roi = roi.unsqueeze(0).to(device)
    output = model(roi)
    if (output[0, 1]>0.5):
        print("OK")
        i += 1
        cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)
    if i == 10:
        break

# Show the image
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.show()
```

خروجی ها به صورت زیر میشود.



شکل 46: خروجی شبکه آموزش داده شده برای **object detection**

پیشنهاد برای بهتر شدن (چون در این حالت خیلی خوب نیست خروجی ها):

- معیار **IOU** برای درست کردن دیتاست را بذاریم عدد بالاتری مثل 80 و 75 جوں عدد 50 ممکن است اصلاً گربه خلی درست نیافتند.
- استفاده از معیار هایی مانند **overlap**
- در بخش **negative** همه را نگه داریم و **weight class** استفاده کنیم.
- در بخش **negative** داده هایی که **area** کمی دارند حذف کنیم.