



به نام خدا



دانشگاه تهران

دانشکده مهندسی برق و کامپیووتر

شبکه های عصبی و یادگیری عمیق

مینی پروژه ۱

شایان واصف	نام و نام خانوادگی
آرمان اکبری	اعضاي گروه
810197603	شماره دانشجویی
810197456	
۱۹ آذر ماه	تاریخ ارسال گزارش

## فهرست گزارش سوالات

3.....	CNN – 1 سوال
3.....	آموزش با MLP
14.....	آموزش با CNN
22.....	Transfer Learning – ۲ سوال
22.....	ResNet18
30.....	VGG16
36.....	Semantic Segmentation – ۳ سوال
41.....	Object Detection – ۴ سوال

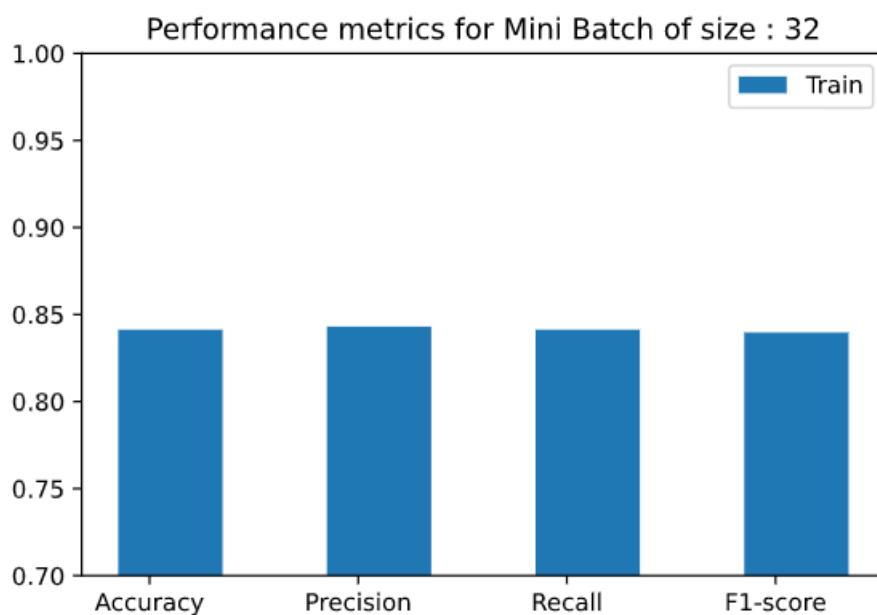
## CNN – 1

آموزش با MLP

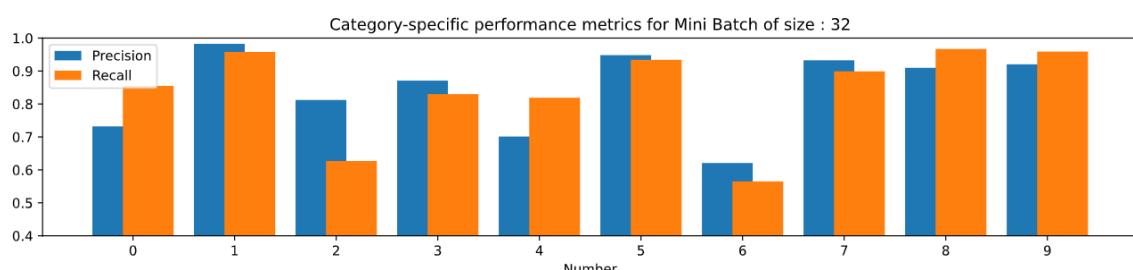
(الف)

در این قسمت به آموزش مدل برای Mini Batch های مختلف می پردازیم . برای هر حالت ، 4 متريک اصلی ( Accuracy,Precision,Recall,F1-score ) آورده شده است . همچنین متريک های Precision و Recall را برای هر یک از 10 کلاس در هر حالت نمايش داده ايم :

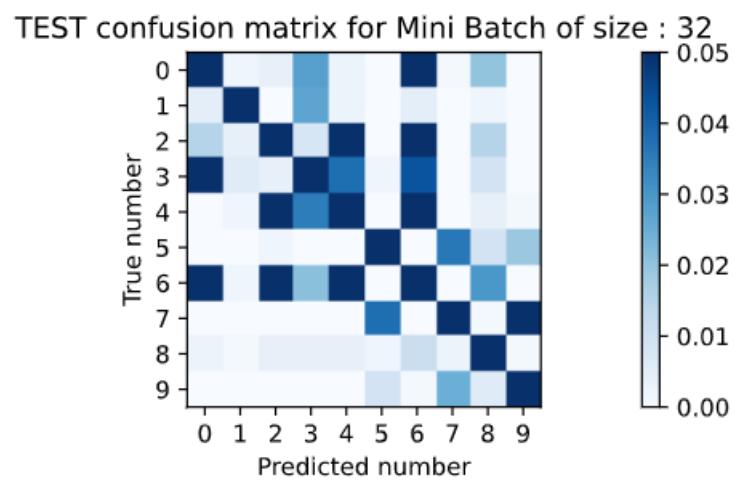
:32 Mini Batches 



شکل 1-1-1 : APRF برای تعداد دسته 32

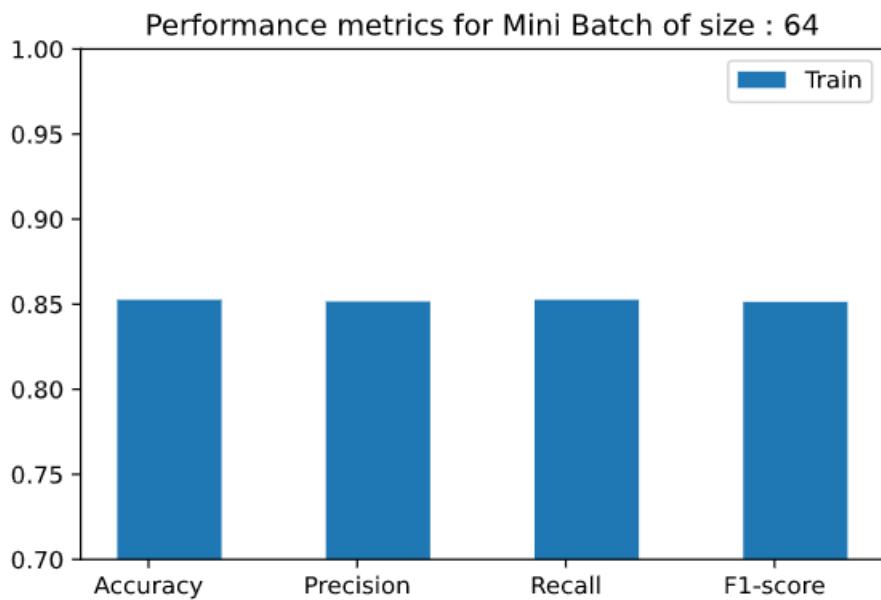


شکل 1-2-2 : PR برای هر یک از 10 کلاس برای تعداد دسته 32

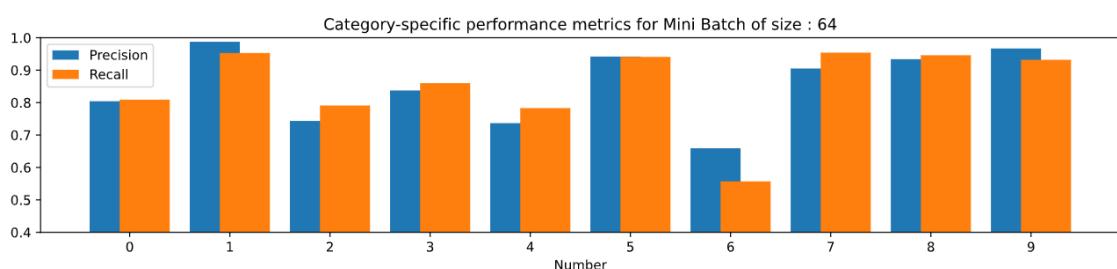


شکل 3-1-1 : ماتریس آشفتگی برای تعداد دسته 32

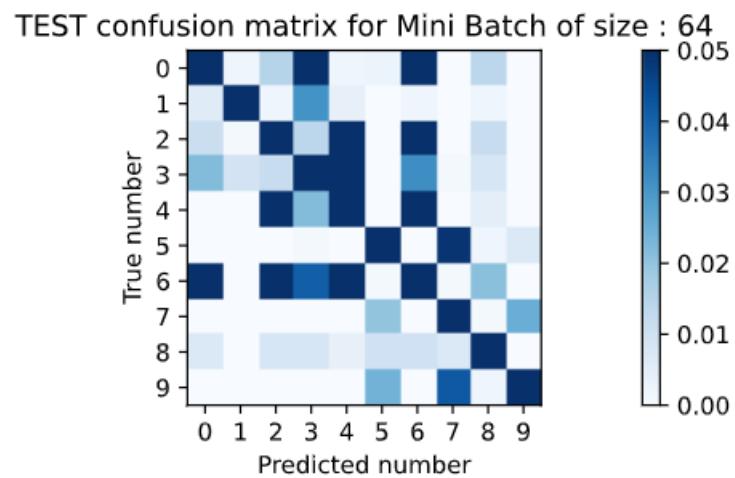
#### :64 Mini Batches



شکل 4-1-1 APRF : 64 برای تعداد دسته 64

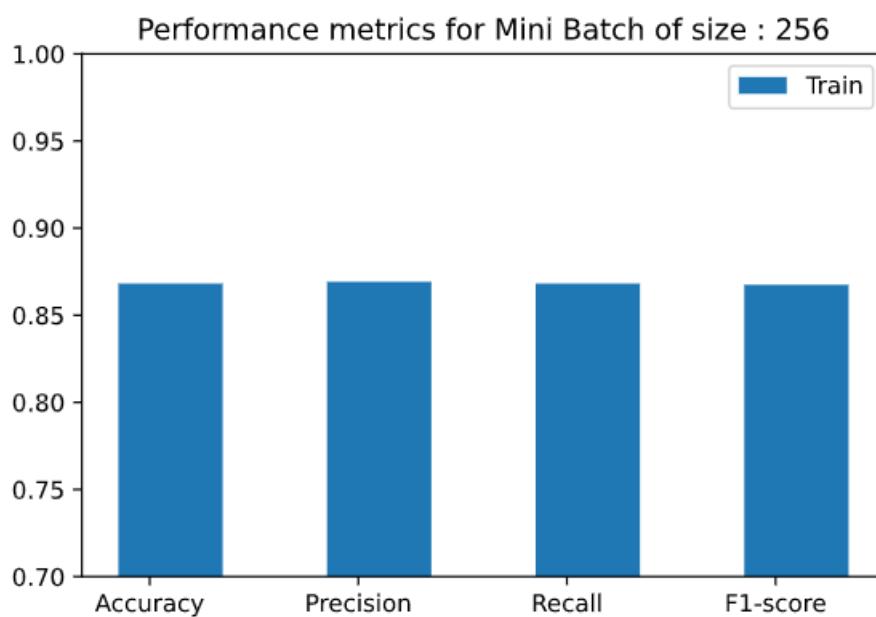


شکل 5-1-1 : PR برای هر یک از 10 کلاس برای تعداد دسته 64

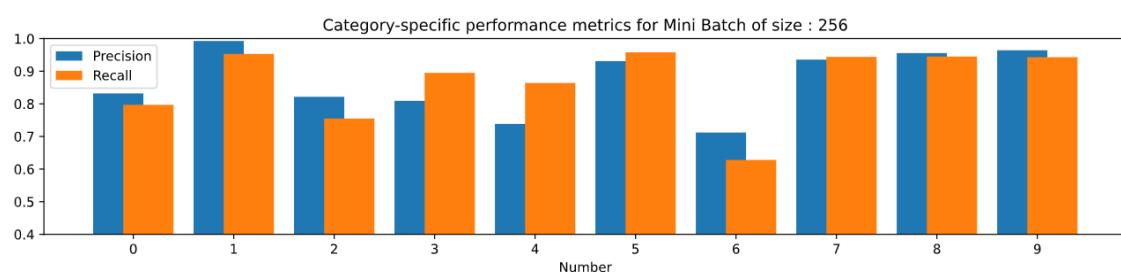


شکل 1-1 : ماتریس آشتفتگی برای تعداد دسته 64

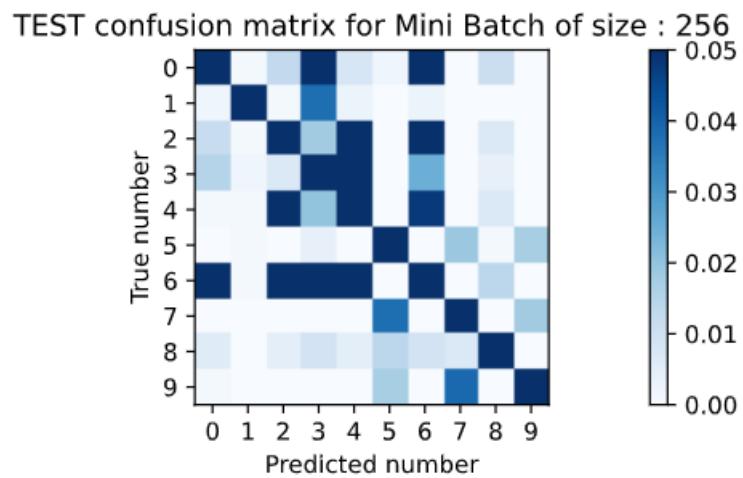
### 256 Mini Batches ↗



شکل 1-2 : APRF برای تعداد دسته 256



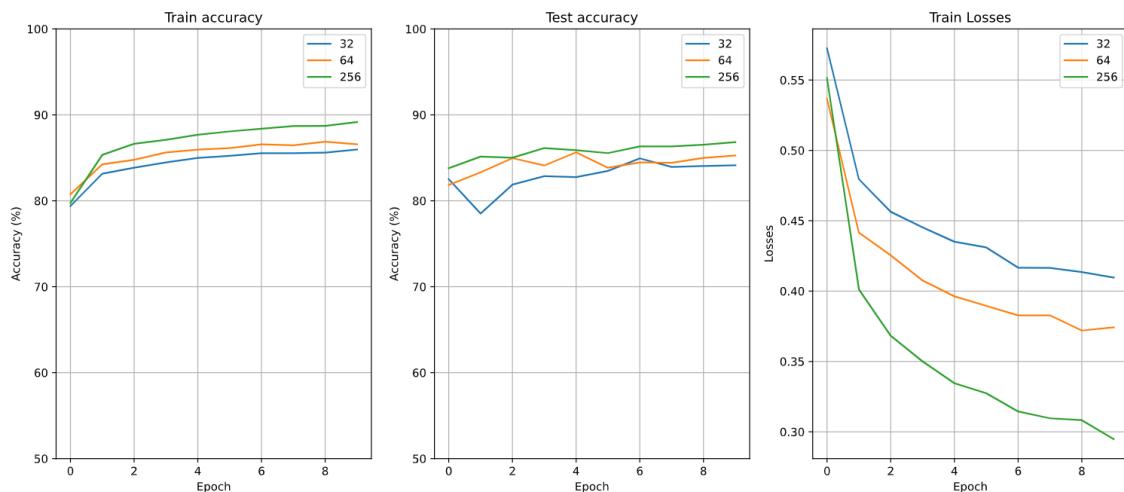
شکل 1-3 : PR برای هر یک از 10 کلاس برای تعداد دسته 256



شکل 1-1-9 : ماتریس آشتفتگی برای تعداد دسته 256

### Overall Performance

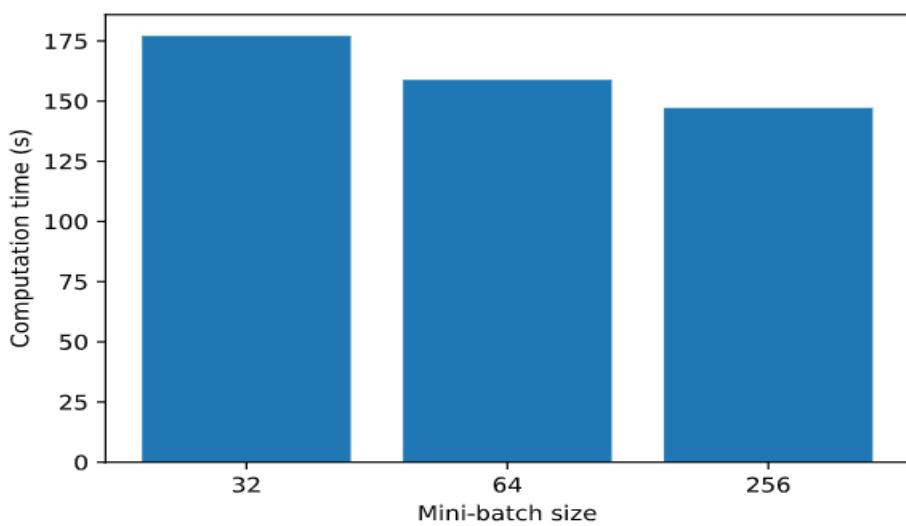
در نهایت ، مقدار خطا و دقت مدل را برای هر تعداد دسته و همچنین زمان اجرایی هر کدام را در زیر آورده‌ایم :



شکل 1-1-10 : نمودار خطا و تست برای دادگان آموزش و تست

همانطور که از نمودارها مشخص است ، برای تعداد 256 دسته ( Mini Batch ) ، بهترین دقت و کمترین خطا در مجموعه دادگان تست داریم .

همچنین زمان محاسباتی برای هر تعداد دسته به صورت زیر می‌باشد :

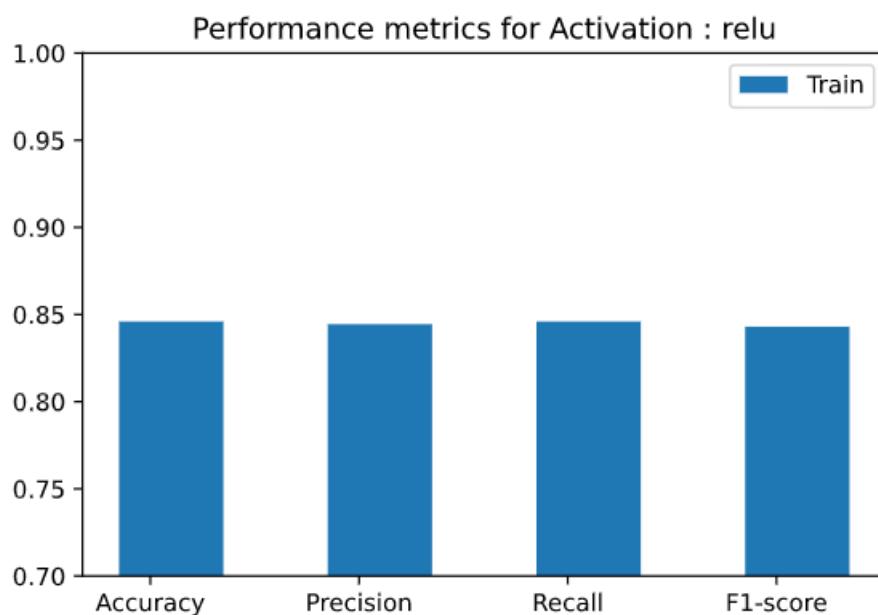


شکل 1-1-11 : زمان محاسباتی ( Computation Time ) برای هر حالت به ثانیه

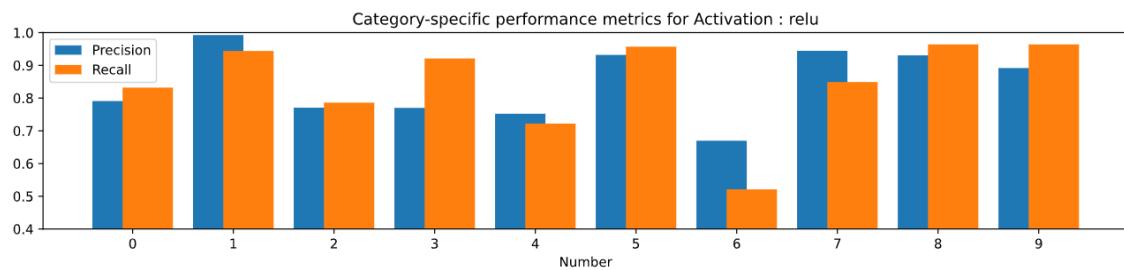
(ب)

: ( Activation Functions ) تاثیر توابع فعال‌ساز

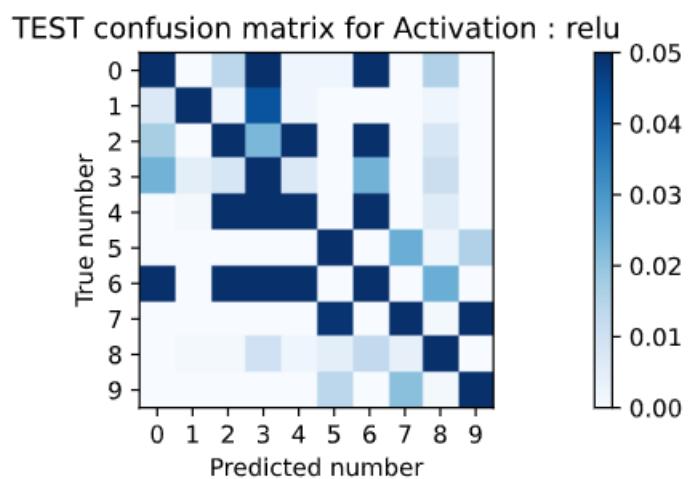
:ReLU



شکل 1-2-1 APRF برای ReLU

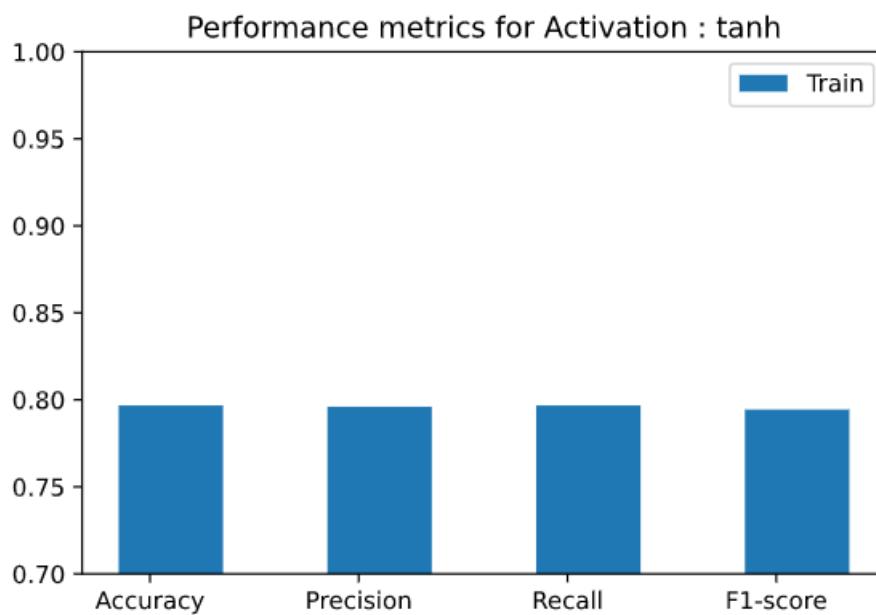


شکل 1-2-1 : PR برای هر یک از 10 کلاس برای ReLU

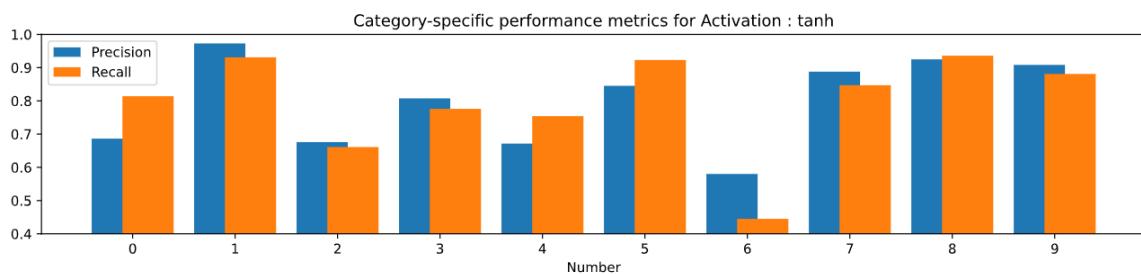


شکل 1-3-1 : ماتریس آشتفتگی برای ReLU

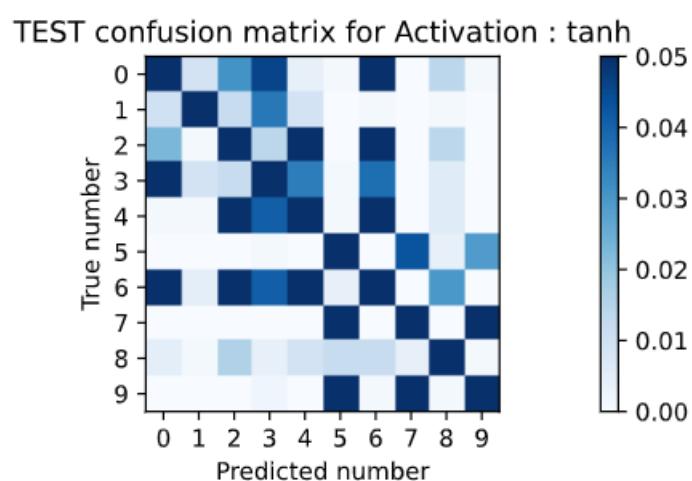
:Tanh ↗



شکل 1-4-1 : APRF برای Tanh

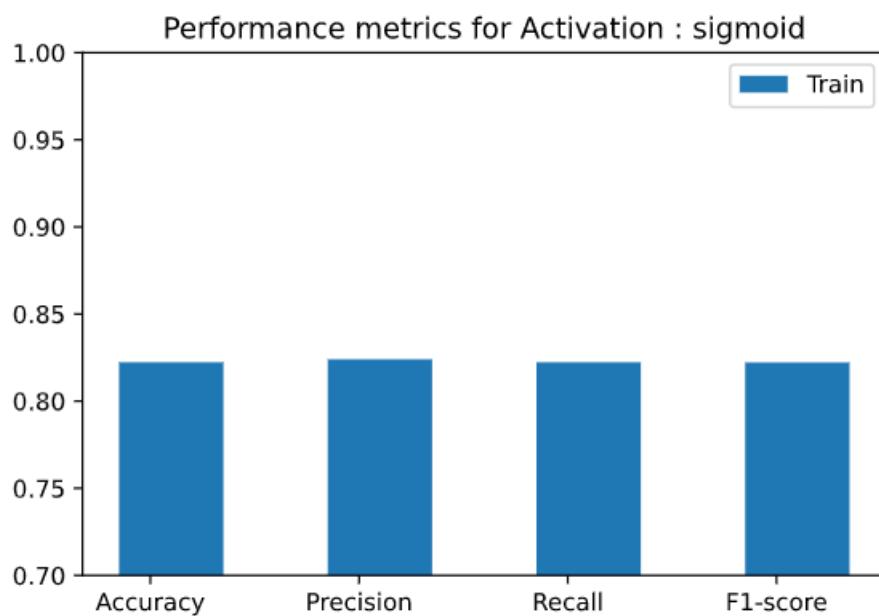


شکل 1-10 : PR برای هر یک از 10 کلاس برای Tanh

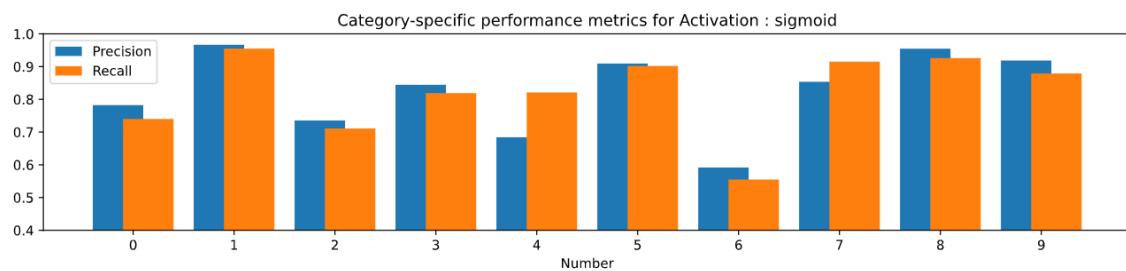


شکل 1-6 : ماتریس آشتفتگی برای Tanh

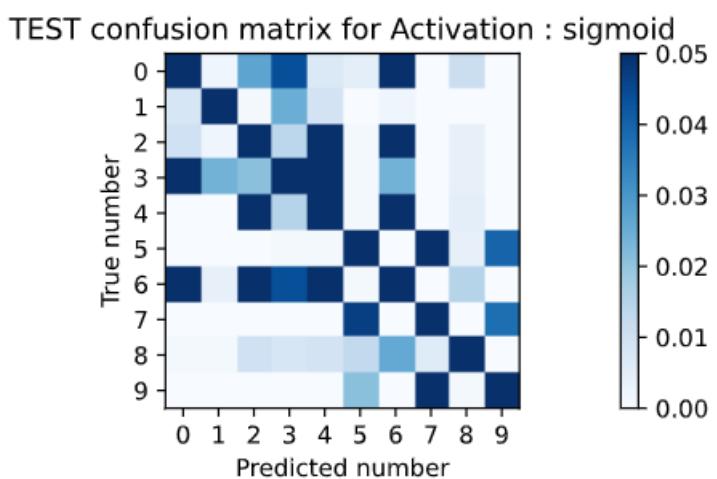
### Sigmoid



شکل 1-4 : APRF برای Sigmoid



شکل 1-2-1 : PR برای هر یک از 10 کلاس برای Sigmoid

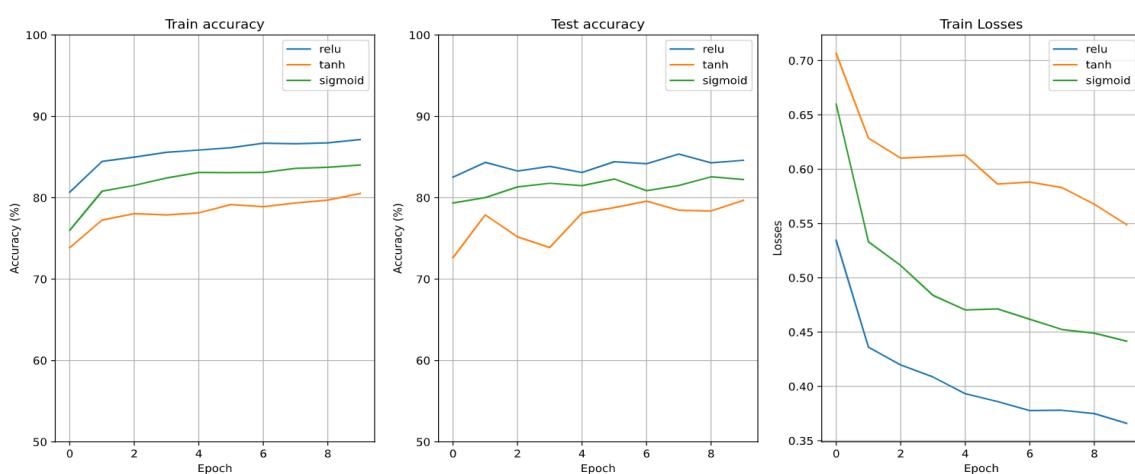


شکل 1-2-1 : ماتریس آشفتگی برای Sigmoid

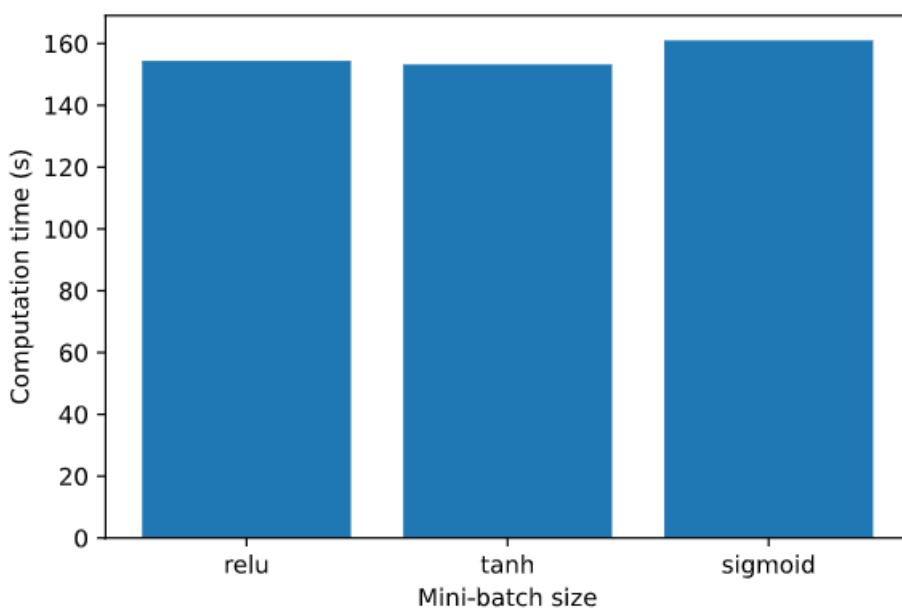
### Overall Performance

در نهایت ، مقدار خطأ و دقت مدل را برای هر Activation و همچنین زمان اجرایی هر کدام را در

زیر آورده‌ایم :



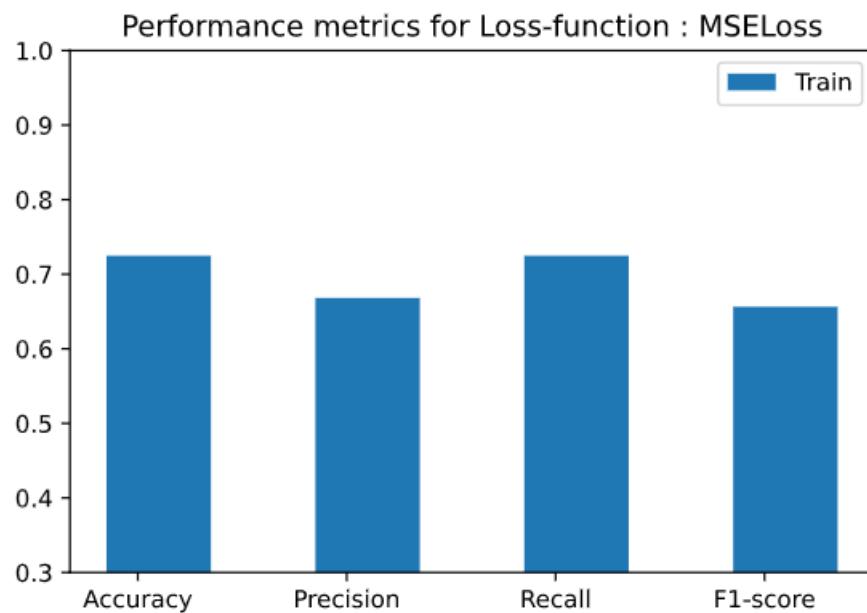
شکل 1-2-7 : نمودار خطأ و تست برای دادگان آموزش و تست



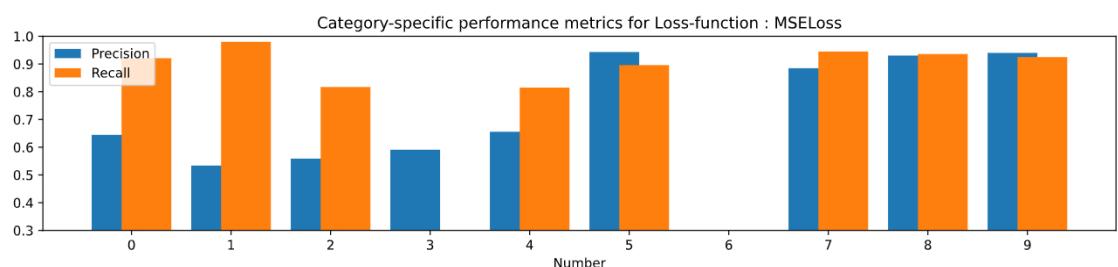
شکل 1-8-2 : زمان محاسباتی ( Computation Time ) برای هر حالت به ثانیه

#### تاثیر توابع خطا ( Loss Functions )

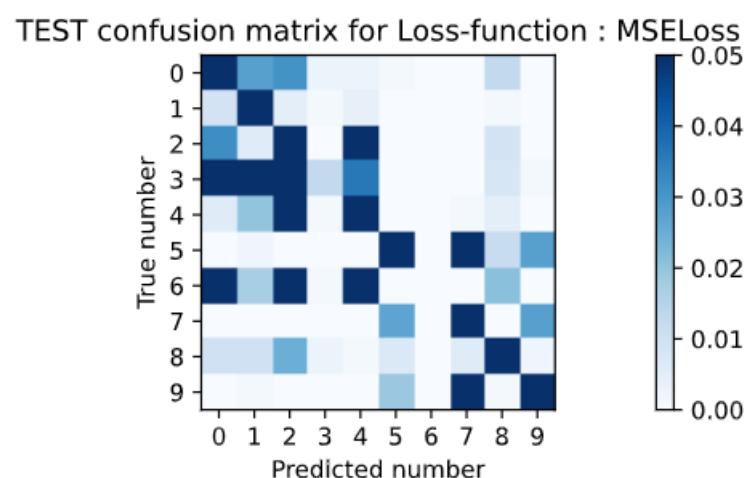
:MSE



شکل 1-9-2-1 : APRF برای MSE

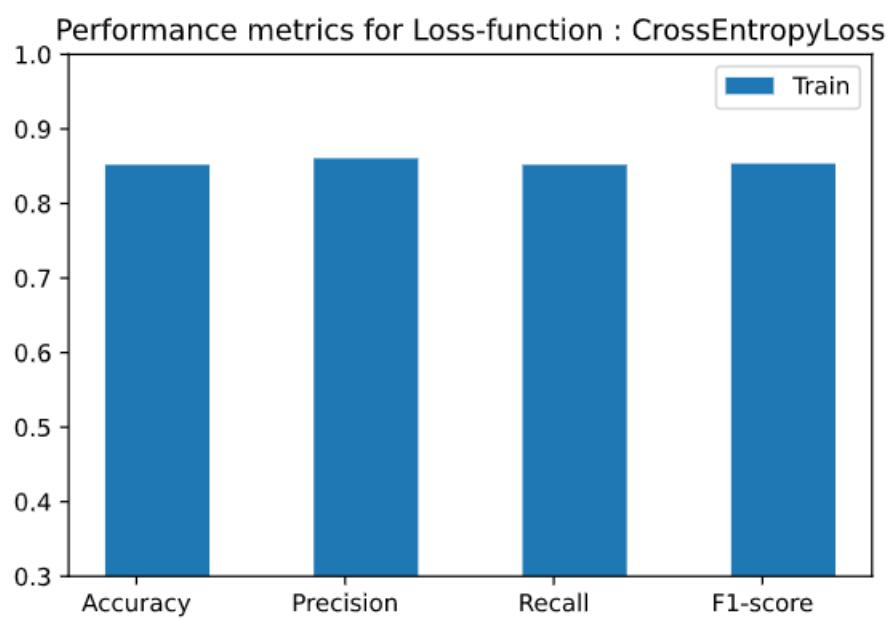


شکل ۱-۱۰ : PR برای هر یک از ۱۰ کلاس برای MSE

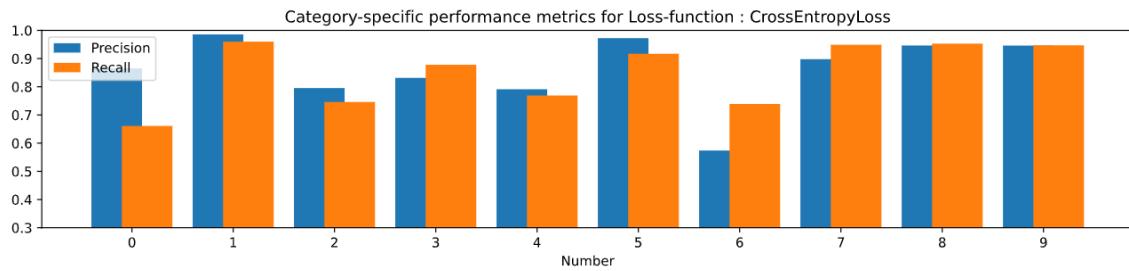


شکل ۱-۱۱ : ماتریس آشفتگی برای MSE

### CrossEntropyLoss

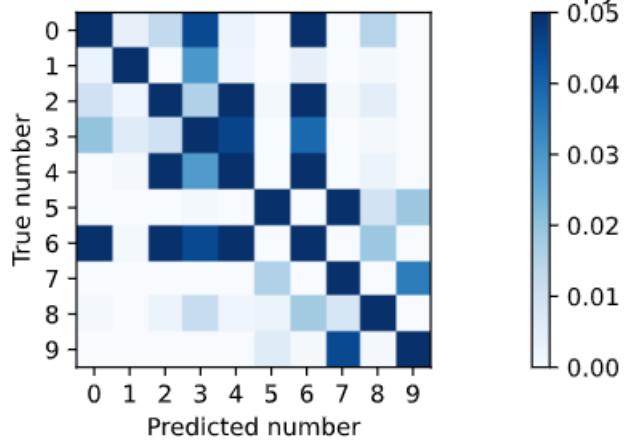


شکل ۱-۱۲ : APRF برای CEL



شکل 1-13: PR برای هر یک از 10 کلاس برای CEL

TEST confusion matrix for Loss-function : CrossEntropyLoss

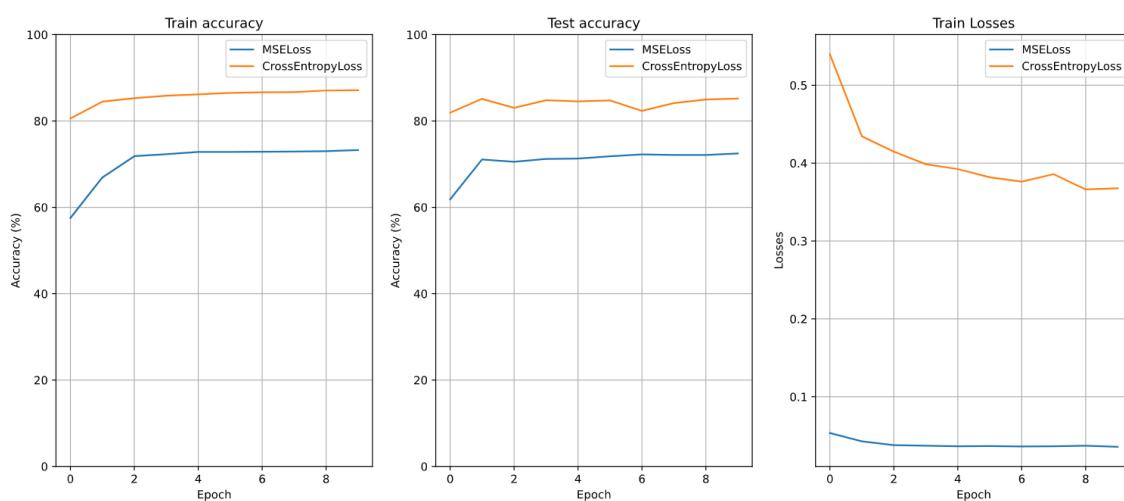


شکل 1-14: ماتریس آشفتگی برای CEL

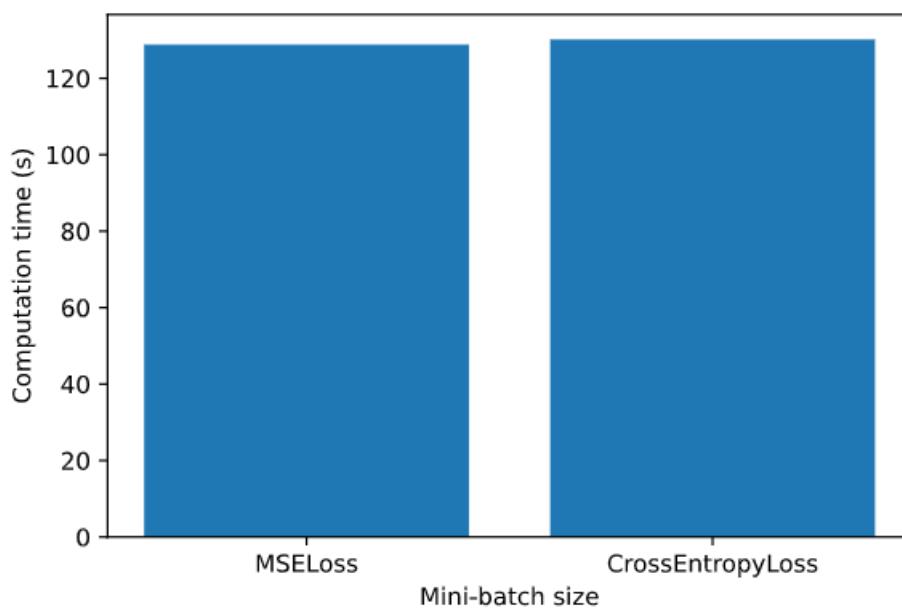
### Overall Performance

در نهایت ، مقدار خطأ و دقت مدل را برای هر Loss-Function و همچنین زمان اجرایی هر کدام را در

زیر آورده‌ایم :



شکل 1-15: نمودار خطأ و تست برای دادگان آموزش و تست



شکل 1-16 : زمان محاسباتی ( Computation Time ) برای هر حالت به ثانیه

طبق نتایج بالا ، تابع خطا Cross Entropy به مرتب بهتر از MSE خواهد بود.

### آموزش با CNN :

برای کاهش زمان محاسباتی ، Runtime Colab GPU را روی قرار می دهیم. همچنین در هنگام آموزش مدل بر روی داده های آموزش و تست نیاز داریم که مدل را به GPU انتقال دهیم که در قطعه کد های زیر دستور های مربوطه آورده شده است :

```
# use GPU if available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# send the model to the GPU          # push data to GPU
net.to(device)                      X = X.to(device)
                                    y = y.to(device)
```

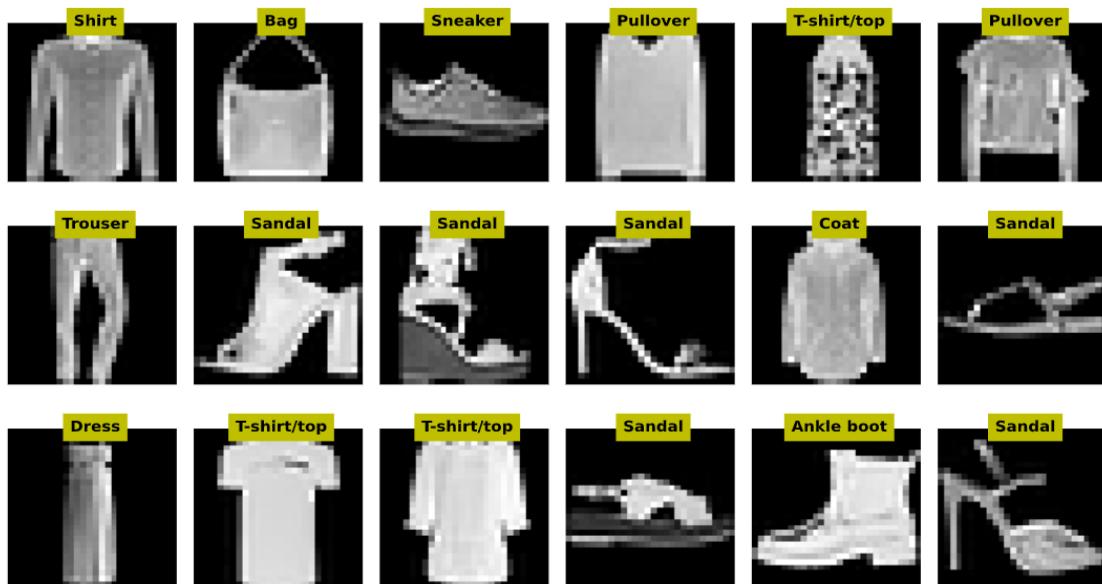
شکل 1-1 : انتقال مدل و داده های آموزش و تست به GPU

در ادامه به کمک کتابخانه Torch vision ، دیتاست موردنظر را را ب مجموعه آموزش و تست به طور جدا دانلود می کنیم . خوبی این کتابخانه در این است که به کمک آن می توان در داده ها را Transform کنیم به این معنی که شکل کلی داده های اصلی را غیر دهیم ( Gray کردن ، در جهت چپ یا راست عکس را 90 درجه بچرخانیم ( Horizontal Flip ( ، نرمال کردن و ...)).

در اینجا ما داده ها را با احتمال تصادفی 0.5 Horizontal Flip می کنیم و همچنین میانگین و واریانس داده ها بر روی 0.5 نرمال می کنیم :

```
# transformations
transform = T.Compose([
    T.ToTensor(),
    T.RandomHorizontalFlip(p=.5),
    T.Normalize(.5,.5),
])
```

شکل 1-3-1 Transform : داده ها



شکل 1-3-2 : نمایی رندوم از تعدادی از دادگان به همراه لیبل آنها در اسکیل سیاه و سفید

### ج) ( بدون Max-Pooling )

در این قسمت تنها به قبیل از لایه های خطی شبکه قسمت قبل ، 2 لایه کانولوشنی به ترتیب با 1 ورودی ( سیاه و سفید ) ، 16 کanal ( 16 Feature maps ) ، کرنل با سایز 3\*3 و یک Padding برای لایه کانولوشنی اول و 16 ورودی ، 32 کanal ، کرنل با سایز 3\*3 برای لایه کانولوشنی دوم اضافه می کنیم . خلاصه شبکه به صورت زیر می باشد :

```
Input: [32, 1, 28, 28]
First CPR block: [32, 16, 28, 28]
Second CPR block: [32, 32, 26, 26]
Vectorized: [32, 21632]
Final output: [32, 10]

Output size:
torch.Size([32, 10])
```

شکل 3-3-1 : خلاصه مدل CNN آموزش داده شده

به طور کلی سه نوع لایه مختلف در CNN وجود دارد :

Feature Map : وظیفه این لایه ، یادگیری فیلترها (Kernels) می باشد تا Convolution ها را ایجاد کند.

Pooling : وظیفه این لایه ، کاهش بعد و افزایش سایز Receptive Fields می باشد (حداکثر تعداد پیکسلی که شبکه می تواند در هر مرحله در اختیار داشته باشد) Fully Connected : در نهایت پیش بینی توسط این لایه انجام می شود.

سایز عکس بدست آمده بعد از گذر از هر لایه کانولوشنی بصورت زیر بدست می آید :

$$N_h = \left\lceil \frac{(M_h + 2p - k)}{s_h} \right\rceil + 1$$

در صورتی که عکس ما مربعی باشد ، آنگاه  $N_h$  تعداد پیکسل بدست آمده در جهت عمودی / افقی در لایه حاضر ،  $M_h$  تعداد پیکسل موجود در جهت عمودی / افقی در لایه پیشین ،  $p$  مقدار Padding و  $k$  مربوط بهStride می باشد که به صورت Default برابر 1 در نظر گرفته می شود.

با توضیحات بالا ، کلاس CNN را به صورت زیر می سازیم :

```
### ----- feature map layers ----- ###
# first convolution layer
self.conv1 = nn.Conv2d(1,16,3,padding=1)
#self.bnrm1 = nn.BatchNorm2d(16) # input the number of channels in this layer
# output size: (28+2*1-3)/1 + 1 = 28 (*Note that no max pooling is inserted so we have no /2 b/c maxpool)

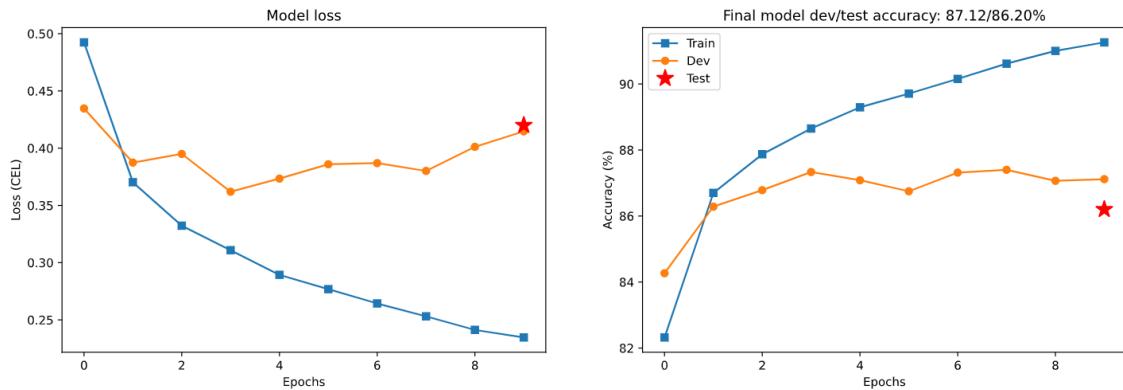
# second convolution layer
self.conv2 = nn.Conv2d(16,32,3)
#self.bnrm2 = nn.BatchNorm2d(32) # input the number of channels in this layer
# output size: (28+2*0-3)/1 + 1 = 26 (*Note that no max pooling is inserted so we have no /2 b/c maxpool)

### ----- linear decision layers ----- ###
self.fc1 = nn.Linear(26*26*32,64)
self.fc2 = nn.Linear(64,32)
self.fc3 = nn.Linear(32,10)
```

همانطور که مشخص است ، ورودی مربوط به لایه Fully Connected سایزی برابر حاصل ضرب مقدار ( 32 ) و تعداد پیکس عکس نهایی ( 26\*32 ) دارد .

علاوه بر این از 10000 دادگان مربوط به تست ، تعداد 6000 را برای دادگان Validation (Devset) و تعداد 4000 را برای دادگان تست در نظر می گیریم .

خطا و دقت مدل برای دادگان آموزش و Devset و همچنین مقدار نهایی دقت بدست آمده برای مجموعه تست آموره شده است :



شکل 1-3-3 : خطا و دقت مدل برای دادگان آموزش ، Devset و مقدار نهایی تست

#### د ) استفاده از Max Pooling و Batch Normalization

در این قسمت علاوه بر لایه های کانولوشنی قسمت قبل ، لایه Batch Normalization و Pooling اضافه می کنیم به طوریکه 2 لایه کانولوشنی به ترتیب با 1 ورودی ( سیاه و سفید ) ، 16 کanal ( 16 کanal با سایز 3\*3 و یک Padding و سپس یک لایه Pooling با سایز 2 برای کانولوشنی اول و 16 ورودی ، 32 کanal ، کرنل با سایز 3\*3 و سپس یک لایه Pooling با سایز 2 برای لایه کانولوشنی دوم اضافه می کنیم . خلاصه شبکه به صورت زیر می باشد :

```

Input: [32, 1, 28, 28]
First CPR block: [32, 16, 14, 14]
Second CPR block: [32, 32, 6, 6]
Vectorized: [32, 1152]
Final output: [32, 10]

Output size:
torch.Size([32, 10])

```

شکل 1-4-1 : خلاصه مدل CNN آموزش داده شده

\*ایده استفاده از Batch Normalization در این است که علاوه بر راین که در ابتدا پیکسل های عکس ورودی نرمالایز کنیم ، برای هر لایه در شبکه نیز این کار را انجام دهیم زیرا لزوماً داده ای که برای ورودی شبکه نرمالایز شده باشد در ورودی لایه های مختلف در همان اسکیل نخواهد بود. با این کار از Vanishing/Exploding Gradient تا حدی می توانیم جلوگیری کنیم .

بنابراین ورودی لایه جدید را که خروجی لایه قبلی بوده را بواسطه ضریبی مثل  $\gamma$  ، انجام می دهیم و توسط ضریبی مثل  $\beta$  ، Variance Scaling انجام می دهیم .

در اینجا ما Batch Normalization را در خروجی لایه Pooling داریم ، پس 2 بار Normalize می کنیم ) پیاده سازی می کنیم.

\*همانطور که در بخش قبل بحث شد ، استفاده از dimension ، Pooling ما را در هر لایه کاهش می دهد که باعث می شود با افزایش عمق شبکه ، Resolution تصویر( تعداد پیکسل ها ) کاهش یابد ولی در عوض تعداد فیلتر ها را افزایش می دهیم . در واقع هر پیکسل در تصویر جدید مساحت بیشتری از آن را شامل می شود که در واقع همان Receptive Fields را افزایش داده ایم .

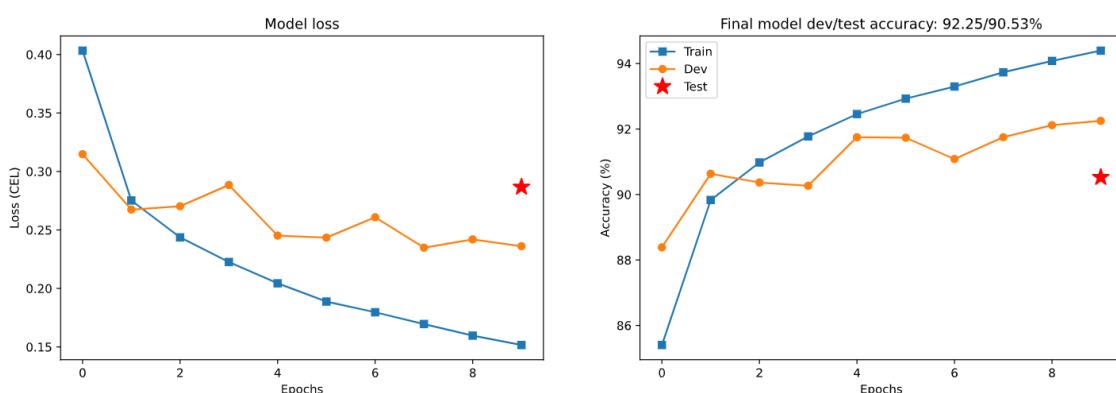
همچنین به دلیل استفاده از Pooling ، تعداد پیکسل در جهت عمودی /افقی خروجی هر لایه نسبت به قسمت قبل نصف شده و معماری کلاس CNN به صورت زیر می باشد :

```
### ----- feature map layers ----- ###
# first convolution layer
self.conv1 = nn.Conv2d(1,16,3,padding=1)
self.bnrm1 = nn.BatchNorm2d(16) # input the number of channels in this layer
# output size: (28+2*1-3)/1 + 1 = 28/2 = 14 (/2 b/c maxpool)

# second convolution layer
self.conv2 = nn.Conv2d(16,32,3)
self.bnrm2 = nn.BatchNorm2d(32) # input the number of channels in this layer
# output size: (14+2*0-3)/1 + 1 = 12/2 = 6 (/2 b/c maxpool)

### ----- linear decision layers ----- ###
self.fc1 = nn.Linear(6*6*32,64)
self.fc2 = nn.Linear(64,32)
self.fc3 = nn.Linear(32,10)
```

خطا و دقت مدل برای دادگان آموزش و Devset و همچنین مقدار نهایی دقت بدست آمده برای مجموعه تست آورده شده است :



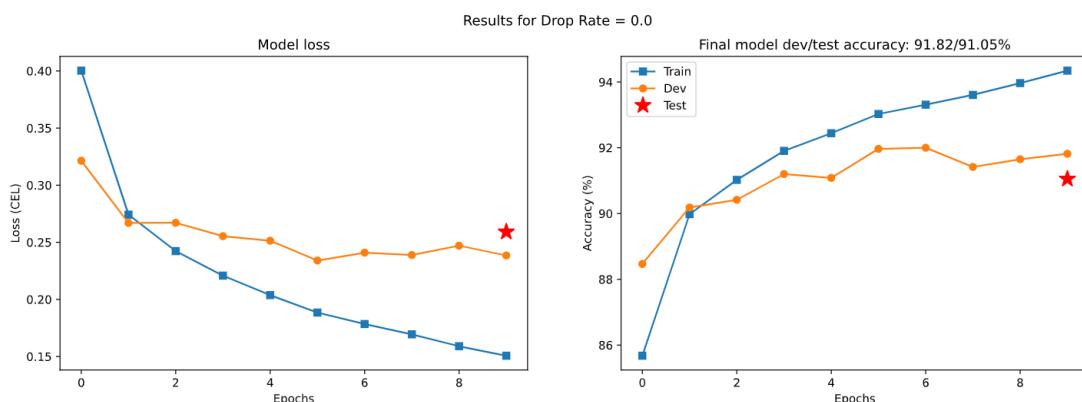
شکل 1-4-2 : خطای دقت مدل برای دادگان آموزش ، Devset و مقدار نهایی تست

در این قسمت برای ۵ مقدار مختلف [0,0.1,0.2,0.3,0.4,0.5] به عنوان Dropout Rates ، شبکه را آموزش می‌دهیم که این مقادیر احتمال صفر شدن وزن‌های مربوط به یک نورون را در طی آموزش نشان می‌دهد. همچنین Drop Out را تنها به لایه Fully Connected اضافه می‌کنیم .

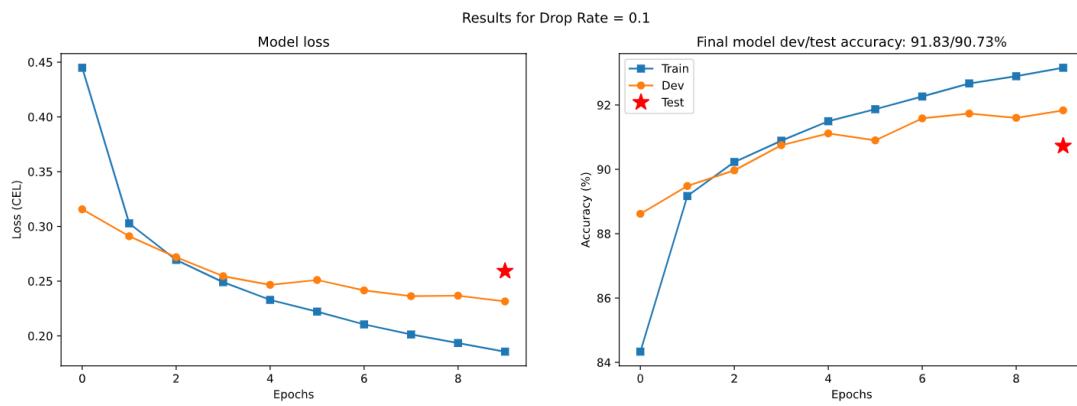
هدف اصلی Drop Out این است که هر نورون در مدل ، یادگیری را مستقل از نورون دیگر یاد بگیرد ولی نکته مهم اینجاست که در استفاده از کرنل‌ها ، ما پیکسل‌های مختلف را به هم وابسته می‌کنیم و بین پیکسل‌های لایه خروجی هر لایه Correlation وجود دارد. بنابراین اضافه کردن درصد زیادی Drop Out مثل 0.5 ، احتمال اینکه مدل در یادگیری وابستگی بین پیکسل‌ها به مشکل بخورد را بالا می‌برد. ولی اضافه کردن مقدار کوچکی مثل 0.2 یا پایین‌تر ، مانند اضافه کردن نویز به عکس‌های ما اثر می‌کند و می‌تواند باعث کاهش Over fit شود .

در مجموع نظرات متفاوتی نسبت به تاثیر مثبت/منفی Drop Out در اضافه کردن به شبکه CNN وجود دارد ولی به عنوان یک procedure کلی می‌توان مقدار کمی Drop Out به قسمت کانولوشنی CNN اضافه کرد که کمی داده‌ها را نویزی کند و همچنین می‌توان مقدار بیشتری به لایه Fully Connected در CNN در اضافه کرد زیرا دیگر در این لایه مشکل وابستگی بین پیکسل‌ها را نداریم و قصد طبقه‌بندی داریم .

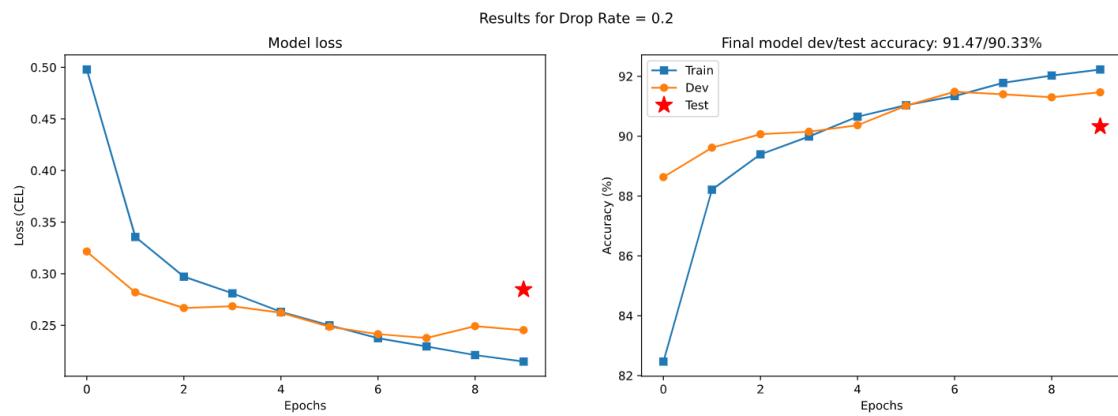
در زیر نتایج را برای مجموعه آموزش ، Devset و نهایی تست آورده‌ایم :



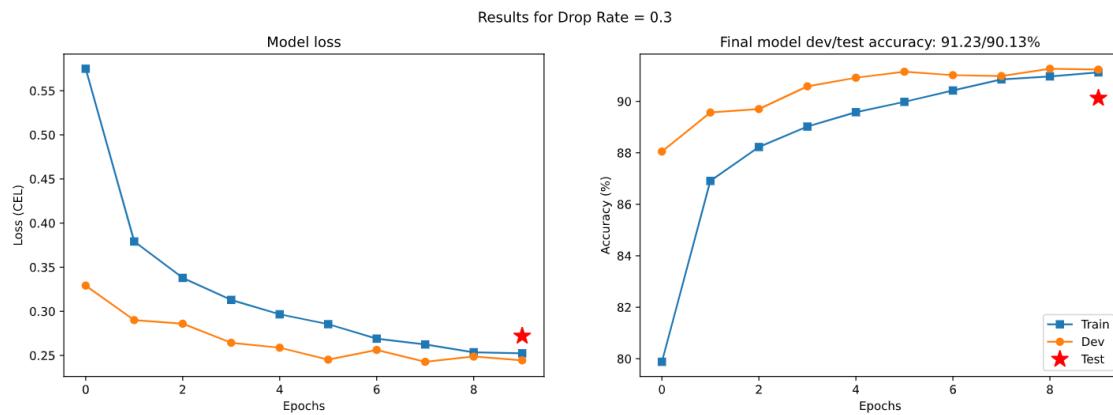
شکل 1-5-1 : خطای و دقت مدل برای دادگان آموزش ، Devset و مقدار نهایی تست برای DR=0



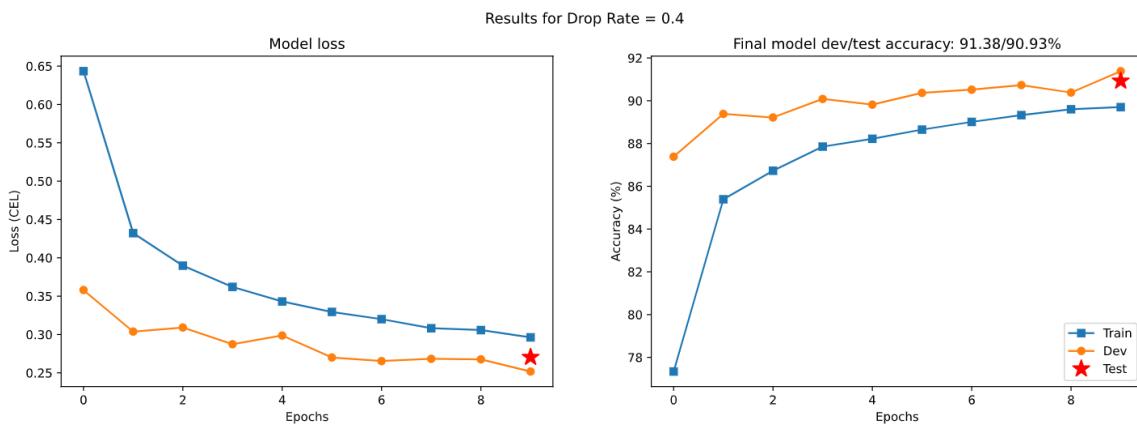
شکل 1-5-1 : خطای و دقت مدل برای دادگان آموزش ، Devset و مقدار نهایی تست برای DR=0.1



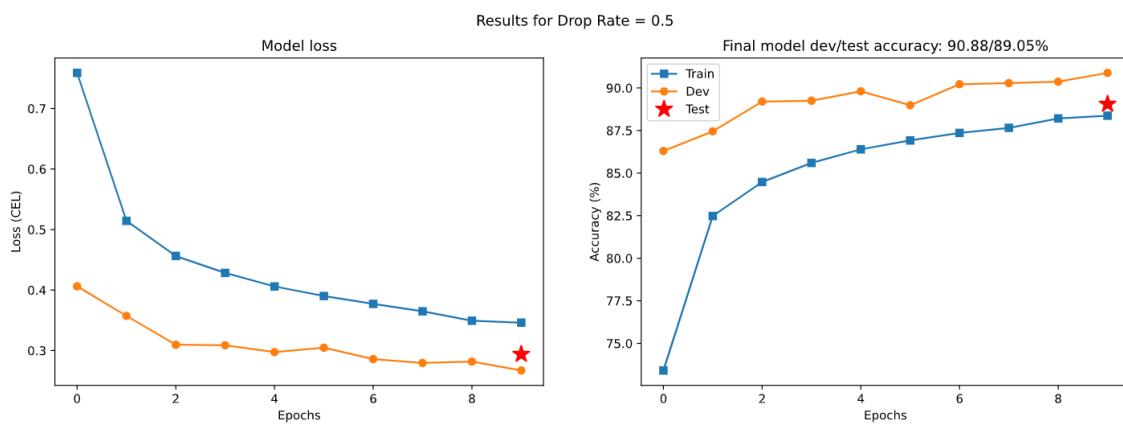
شکل 1-5-2 : خطای و دقت مدل برای دادگان آموزش ، Devset و مقدار نهایی تست برای DR=0.2



شکل 1-5-3 : خطای و دقت مدل برای دادگان آموزش ، Devset و مقدار نهایی تست برای DR=0.3



شکل 1-5-4 : خطأ و دقت مدل برای دادگان آموزش ، Devset و مقدا نهايی تست برای DR=0.4



شکل 1-5-5 : خطأ و دقت مدل برای دادگان آموزش ، Devset و مقدا نهايی تست برای DR=0.5

طبق نتایج بالا ، برای DR=0.4 ، بهترین دقت را در مجموعه Validation و تست (91.38 / 90.93) داریم .

## سوال 2 Transfer Learning – 2

ما در این قسمت دو شبکه ResNet18 و همچنین VGG16 را بر روی یک دیتابست مربوط به تصاویر به صورت Pre-Trained اجرا کرده و نتایج را گزارش می‌دهیم.

\*دیتابست فوق را به کمک دستور Torch. Vision به صورت زیر و قابل تفکیک به دادگان آموزش و تست دانلود می‌کنیم:

```
# import the data and simultaneously apply the transform
trainset = torchvision.datasets.STL10(root='./data', download=True, split='train', transform=transform)
testset = torchvision.datasets.STL10(root='./data', download=True, split='test', transform=transform)
```

شکل 2-1-1 : دانلود دیتابست مورد نیاز به منظور آزمایش مدل‌های Pre-Trained شده

همچنین اطلاعات کلی مربوط به دادگان تست و آموزش و لیبل‌ها برای دیتابست مورد نظر به صورت زیر می‌باشد :

```
Data shapes (train/test):
(5000, 3, 96, 96)
(8000, 3, 96, 96)

Data value range:
(0, 255)

Data categories:
['airplane', 'bird', 'car', 'cat', 'deer', 'dog', 'horse', 'monkey', 'ship', 'truck']
```

شکل 2-1-2 : اطلاعات کلی دیتابست داده شده

### ResNet18

مدل ResNet ، انواع مختلفی دارد که ما به دلیل ساده شدن محاسبات از ResNet18 استفاده می‌کنیم:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	$11.3 \times 10^9$

شکل 2-1-3 : انواع مختلف مدل ResNet

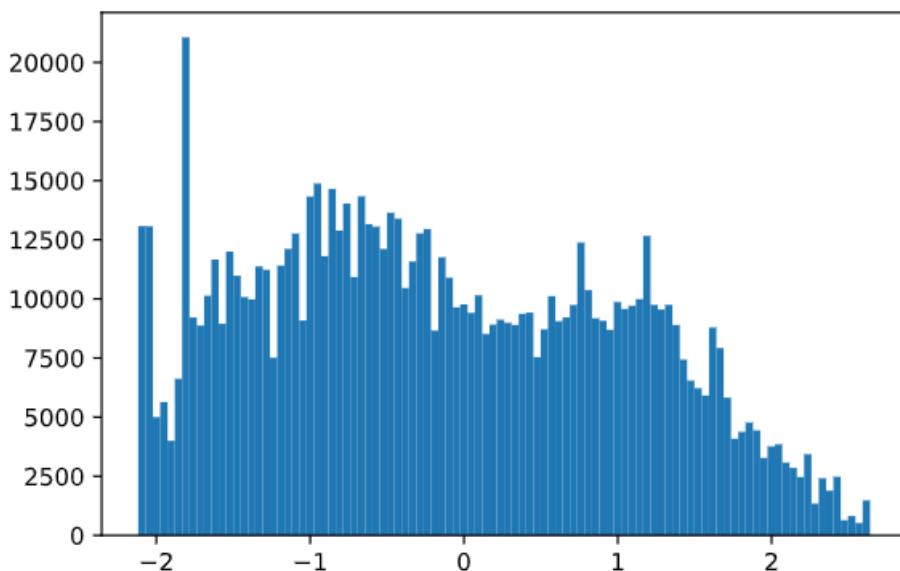
## الف )

همانطور که شکل 2-1-2 مشخص است ، پیکسل های تصاویر نرمالایز نیستند و همچنین تنها 10 دسته مختلف برای طبقه بندی داریم . با اعمال Transform به دادگان و استفاده از کتابخانه Data Loader ، دادگان را نرمالایز می کنیم ( به دلیل اینکه مقادیر پیکسل های نرمال شده در شبکه ResNet18 تقریباً بین 2- و 2 می باشد ، دادگان دیتاست را در همین اسکیل نرمالایز می کنیم ) . برای یک Batch از دادگان ، سایز و مینیمم و ماکسیمم مقادیر را چک می کنیم :

```
Data shapes (train/test):  
torch.Size([32, 3, 96, 96])  
  
Data value range:  
(tensor(-2.1179), tensor(2.6400))
```

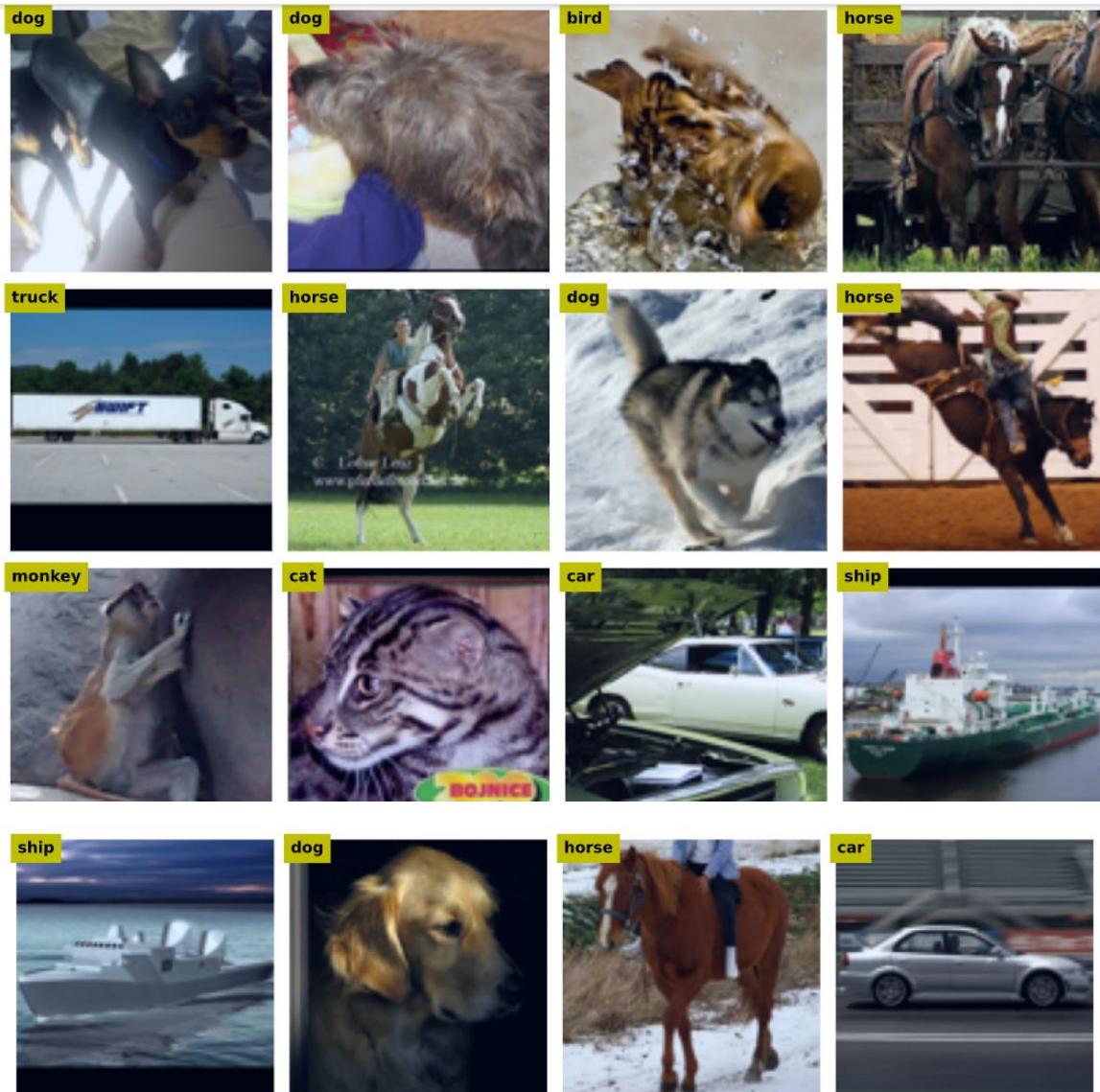
شکل 2-1-3 : اطلاعات کلی دیتاست نرمال شده برای یک Batch داده

در زیر هیستوگرامی از تراکم داده ها در بازه نرمالایز شده قرار داده ایم :



شکل 2-1-4 : هیستوگرام دیتاست نرمال شده برای دادگان آموزش

همچنین برای تعداد رندومی از داده ها ، عکس ها را به همراه لیبل آنها در زیر آورده ایم :



شکل 2-5 : تصاویر به همراه نوع لیبل تعدادی رندوم از دادگان دیتاست STL10

حال به کمک کتابخانه Torch. Vision مدل ResNet18 Pre-Trained شده را دانلود می کنیم.

```
resnet = torchvision.models.resnet18(pretrained=True)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100% [██████████] 44.7M/44.7M [00:00<00:00, 72.3MB/s]
```

شکل 2-6 : دانلود مدل Pre-Trained شده ResNet18

معماری مدل به صورت زیر می باشد :

```

ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1): Sequential(
        (0): BasicBlock(
            (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
            (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (1): BasicBlock(
            (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
            (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
    )
    (layer2): Sequential(
        (0): BasicBlock(
            (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (relu): ReLU(inplace=True)
            (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (downsample): Sequential(
                (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
                (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
        )
    )
    (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
)
(layer3): Sequential(
    (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
            (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
    )
    (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
)
)

```

```

(layer4): Sequential(
    (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
            (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
    )
    (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=512, out_features=1000, bias=True)
)

```

شکل 7-1-2 : معماری مدل Pre-Trained شده ResNet18

همانطور که مشخص است ، مدل به طور کلی از 4 زیر لایه (18 زیر لایه) تشکیل شده و در هر لایه دو بلوک مشاهده می شود که در آنها دو لایه کانولوشنی هررا با قرار گرفته است .

همچنین خلاصه مدل و پارامتر ها به صورت زیر می باشد :

Layer (type)	Output Shape	Param #			
Conv2d-1	[ -1, 64, 48, 48]	9,408	ReLU-37	[ -1, 256, 6, 6]	0
BatchNorm2d-2	[ -1, 64, 48, 48]	128	Conv2d-38	[ -1, 256, 6, 6]	589,824
ReLU-3	[ -1, 64, 48, 48]	0	BatchNorm2d-39	[ -1, 256, 6, 6]	512
MaxPool2d-4	[ -1, 64, 24, 24]	0	Conv2d-40	[ -1, 256, 6, 6]	32,768
Conv2d-5	[ -1, 64, 24, 24]	36,864	BatchNorm2d-41	[ -1, 256, 6, 6]	512
BatchNorm2d-6	[ -1, 64, 24, 24]	128	ReLU-42	[ -1, 256, 6, 6]	0
ReLU-7	[ -1, 64, 24, 24]	0	BasicBlock-43	[ -1, 256, 6, 6]	0
Conv2d-8	[ -1, 64, 24, 24]	36,864	Conv2d-44	[ -1, 256, 6, 6]	589,824
BatchNorm2d-9	[ -1, 64, 24, 24]	128	BatchNorm2d-45	[ -1, 256, 6, 6]	512
ReLU-10	[ -1, 64, 24, 24]	0	ReLU-46	[ -1, 256, 6, 6]	0
BasicBlock-11	[ -1, 64, 24, 24]	0	Conv2d-47	[ -1, 256, 6, 6]	589,824
Conv2d-12	[ -1, 64, 24, 24]	36,864	BatchNorm2d-48	[ -1, 256, 6, 6]	512
BatchNorm2d-13	[ -1, 64, 24, 24]	128	ReLU-49	[ -1, 256, 6, 6]	0
ReLU-14	[ -1, 64, 24, 24]	0	BasicBlock-50	[ -1, 256, 6, 6]	0
Conv2d-15	[ -1, 64, 24, 24]	36,864	Conv2d-51	[ -1, 512, 3, 3]	1,179,648
BatchNorm2d-16	[ -1, 64, 24, 24]	128	BatchNorm2d-52	[ -1, 512, 3, 3]	1,024
ReLU-17	[ -1, 64, 24, 24]	0	ReLU-53	[ -1, 512, 3, 3]	0
BasicBlock-18	[ -1, 64, 24, 24]	0	Conv2d-54	[ -1, 512, 3, 3]	2,359,296
Conv2d-19	[ -1, 128, 12, 12]	73,728	BatchNorm2d-55	[ -1, 512, 3, 3]	1,024
BatchNorm2d-20	[ -1, 128, 12, 12]	256	Conv2d-56	[ -1, 512, 3, 3]	131,072
ReLU-21	[ -1, 128, 12, 12]	0	BatchNorm2d-57	[ -1, 512, 3, 3]	1,024
Conv2d-22	[ -1, 128, 12, 12]	147,456	ReLU-58	[ -1, 512, 3, 3]	0
BatchNorm2d-23	[ -1, 128, 12, 12]	256	BasicBlock-59	[ -1, 512, 3, 3]	0
Conv2d-24	[ -1, 128, 12, 12]	8,192	Conv2d-60	[ -1, 512, 3, 3]	2,359,296
BatchNorm2d-25	[ -1, 128, 12, 12]	256	BatchNorm2d-61	[ -1, 512, 3, 3]	1,024
ReLU-26	[ -1, 128, 12, 12]	0	ReLU-62	[ -1, 512, 3, 3]	0
BasicBlock-27	[ -1, 128, 12, 12]	0	Conv2d-63	[ -1, 512, 3, 3]	2,359,296
Conv2d-28	[ -1, 128, 12, 12]	147,456	BatchNorm2d-64	[ -1, 512, 3, 3]	1,024
BatchNorm2d-29	[ -1, 128, 12, 12]	256	ReLU-65	[ -1, 512, 3, 3]	0
ReLU-30	[ -1, 128, 12, 12]	0	BasicBlock-66	[ -1, 512, 3, 3]	0
Conv2d-31	[ -1, 128, 12, 12]	147,456	AdaptiveAvgPool2d-67	[ -1, 512, 1, 1]	0
BatchNorm2d-32	[ -1, 128, 12, 12]	256	Linear-68	[ -1, 1000]	513,000
ReLU-33	[ -1, 128, 12, 12]	0	Total params:	11,689,512	
BasicBlock-34	[ -1, 128, 12, 12]	0	Trainable params:	11,689,512	
Conv2d-35	[ -1, 256, 6, 6]	294,912	Non-trainable params:	0	
BatchNorm2d-36	[ -1, 256, 6, 6]	512			

شکل 8-1-2 : خلاصه مدل به همراه پارامتر های Pre-Trained شده ResNet18

( ب )

طبق بالا ، خروجی مدل ResNet18 1000 دسته می باشد در حالیکه دیتاست STL10 تنها 10 دسته دارد که باید تغییری در لایه آخر شبکه ایجاد کنیم. با این تفاسیر نیاز به آموزش بقیه وزن های مدل نداریم و با دستور زیر وزن های شبکه را Freeze می کنیم :

```
# Freeze all layers (final layer changed later)
for p in resnet.parameters():
    p.requires_grad = False
```

شکل 9-1-2 : Freeze کردن تمامی وزن های شبکه

```
# change the final layer
resnet.fc = nn.Linear(512,10)
```

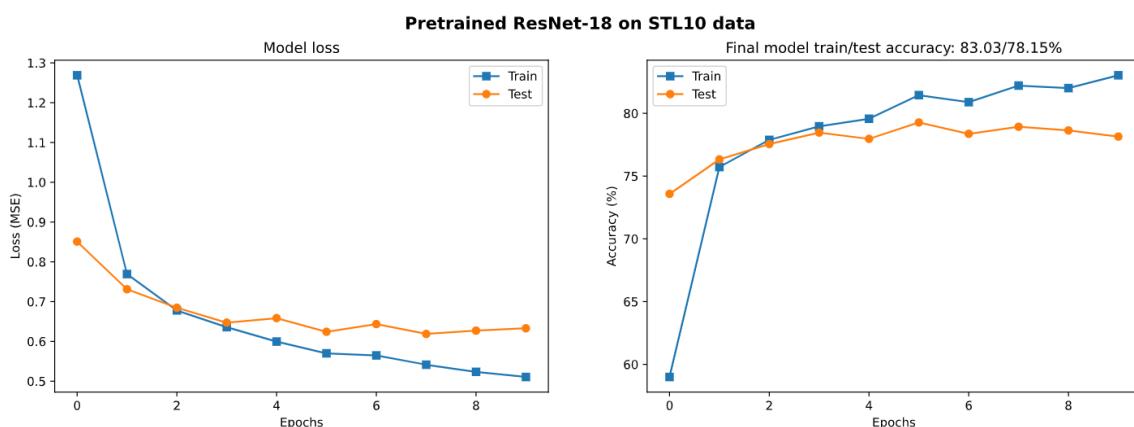
شکل 10-1-2 : تغییر تعداد نورون خروجی لایه آخر

در نهایت مدل تغییر یافته را بر روی دیتاست اولیه برای epoch 10 آموزش می دهیم:

```
Finished epoch 1/10. Test accuracy = 72.85%
Finished epoch 2/10. Test accuracy = 76.03%
Finished epoch 3/10. Test accuracy = 77.81%
Finished epoch 4/10. Test accuracy = 77.97%
Finished epoch 5/10. Test accuracy = 77.99%
Finished epoch 6/10. Test accuracy = 78.45%
Finished epoch 7/10. Test accuracy = 79.08%
Finished epoch 8/10. Test accuracy = 78.70%
Finished epoch 9/10. Test accuracy = 79.14%
Finished epoch 10/10. Test accuracy = 78.30%
```

شکل 11-1-2 : دقت بدست آمده در مجموعه تست برای epoch های مختلف

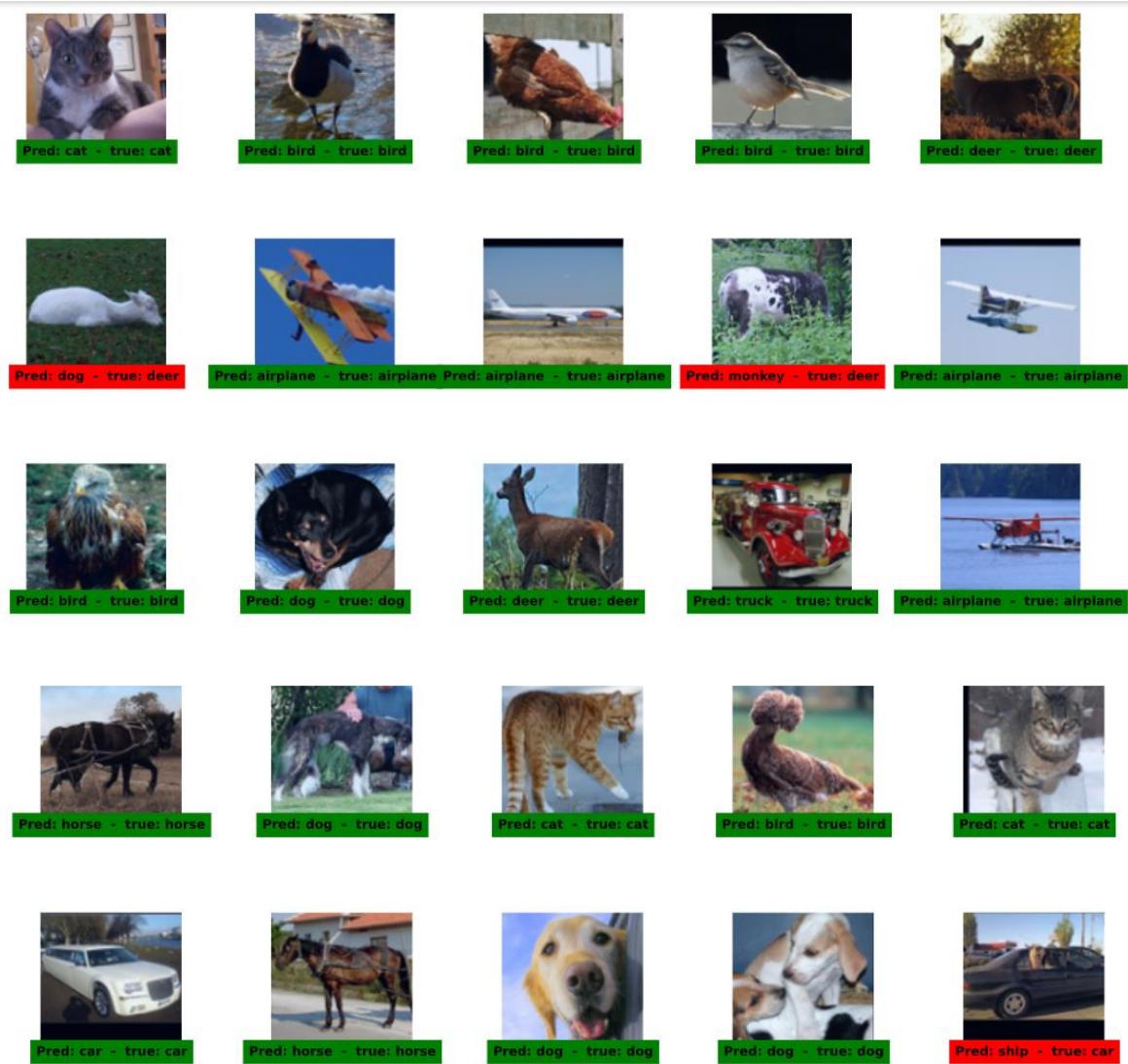
همچنین نمودار خطأ و دقت به صورت زیر می باشد :



شکل 12-1-2 : خطأ و دقت بدست آمده برای مجموعه آموزش و تست

(ج)

در نهایت برای تعداد 25 تصویر به صورت رندوم ، کلاس اصلی و پیش بینی شده توسط مدل ResNet18 را رسم می کنیم :



شكل 2-13-1 : تصاویر به همراه نوع لیبل اصلی و پیش بینی شده تعدادی رندوم از دادگان دیتابست

STL10

همانطور که مشخص است در 25 عکس رندوم بالا ، سه عکس اشتباه لیبل خورده را مشاهده می کنیم. از این سه عکس برای بخش بعدی استفاده می کنیم .

بار دیگر به سه عکسی که با کمک مدل Pre-Trained شده ، اشتباه لیبل خورده بود نگاه می‌اندازیم :



شکل 14-1-2 : لیبل های درست سه عکس اشتباه لیبل خورده توسط مدل Pre-Trained شدند . حال اینجا ، گرادیان هر لایه را فعال می‌کنیم ( در شکل 2-1-9 آنرا بر روی True قرار می‌دهیم ) و سپس مدل را برای این سه عکس از اول آموزش می‌دهیم و نتایج پیش‌بینی شده را می‌آوریم :

```
Finished epoch 1/5. Test accuracy = 33.33%
Finished epoch 2/5. Test accuracy = 33.33%
Finished epoch 3/5. Test accuracy = 33.33%
Finished epoch 4/5. Test accuracy = 33.33%
Finished epoch 5/5. Test accuracy = 33.33%
```

شکل 11-1-2 : دقت بدست آمده برای سه عکس آورده شده در بالا



شکل 13-1-2 : لیبل های جدید assign شده به تصویر Misclassified کل مدل همانطور که مشخص است ، با آموزش ابتدایی مدل و تنظیم دوباره وزن ها ، عکس اول سمت راست که توسط مدل Pre-Trained شده اشتباه تشخیص داده شده بود ، اینبار درست طبقه بندی شد . بنابراین با وزن های جدید آموزش دیده ، نتایج بهتر خواهد شد .

## VGG16

مثل قسمت قبل به کمک کتابخانه Torch. Vision مدل Pre-Trained شده VGG16 را دانلود می‌کنیم.

```
vggnet = torchvision.models.vgg16(pretrained=True)

Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/torch/hub/checkpoints/vgg16-397923af.pth
100% [██████████] 528M/528M [00:05<00:00, 84.1MB/s]
```

شكل 1-2 : دانلود مدل Pre-Trained شده ResNet18

### الف)

معماری مدل به صورت زیر می‌باشد :

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

```

(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
)
)

```

شکل 2-2-2 : معماری مدل Pre-Trained شده VGG16

همانطور که مشخص است ، مدل از سه بخش کلی تشکیل شده است. در بخش اول ( features ) لایه های کانولوشنی و pooling ها به هر اد توابع غیر خطی ( ReLU ) قرار گرفته اند . میان بخش اول و سوم قسمتی به عنوان Pooling ( از نوع Average Pooling ) قرار گرفته است و در بخش آخر ، قسمت که به منظور طبقه بندی می باشد ، قرار داده شده است . Fully Connected

همچنین خلاصه مدل و پارامتر ها به صورت زیر می باشد :

Layer (type)	Output Shape	Param #
Conv2d-1	[ -1, 64, 96, 96]	1,792
ReLU-2	[ -1, 64, 96, 96]	0
Conv2d-3	[ -1, 64, 96, 96]	36,928
ReLU-4	[ -1, 64, 96, 96]	0
MaxPool2d-5	[ -1, 64, 48, 48]	0
Conv2d-6	[ -1, 128, 48, 48]	73,856
ReLU-7	[ -1, 128, 48, 48]	0
Conv2d-8	[ -1, 128, 48, 48]	147,584
ReLU-9	[ -1, 128, 48, 48]	0
MaxPool2d-10	[ -1, 128, 24, 24]	0
Conv2d-11	[ -1, 256, 24, 24]	295,168
ReLU-12	[ -1, 256, 24, 24]	0
Conv2d-13	[ -1, 256, 24, 24]	590,080
ReLU-14	[ -1, 256, 24, 24]	0
Conv2d-15	[ -1, 256, 24, 24]	590,080
ReLU-16	[ -1, 256, 24, 24]	0
MaxPool2d-17	[ -1, 256, 12, 12]	0
Conv2d-18	[ -1, 512, 12, 12]	1,180,160
ReLU-19	[ -1, 512, 12, 12]	0
Conv2d-20	[ -1, 512, 12, 12]	2,359,808
ReLU-21	[ -1, 512, 12, 12]	0
Conv2d-22	[ -1, 512, 12, 12]	2,359,808
ReLU-23	[ -1, 512, 12, 12]	0
MaxPool2d-24	[ -1, 512, 6, 6]	0
Conv2d-25	[ -1, 512, 6, 6]	2,359,808
ReLU-26	[ -1, 512, 6, 6]	0
Conv2d-27	[ -1, 512, 6, 6]	2,359,808
ReLU-28	[ -1, 512, 6, 6]	0
Conv2d-29	[ -1, 512, 6, 6]	2,359,808
ReLU-30	[ -1, 512, 6, 6]	0
MaxPool2d-31	[ -1, 512, 3, 3]	0
AdaptiveAvgPool2d-32	[ -1, 512, 7, 7]	0
Linear-33	[ -1, 4096]	102,764,544
ReLU-34	[ -1, 4096]	0
Dropout-35	[ -1, 4096]	0

```

    Linear-36           [-1, 4096]      16,781,312
    ReLU-37             [-1, 4096]          0
    Dropout-38          [-1, 4096]          0
    Linear-39           [-1, 1000]       4,097,000
=====
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
-----
Input size (MB): 0.11
Forward/backward pass size (MB): 40.50
Params size (MB): 527.79
Estimated Total Size (MB): 568.40

```

شکل 2-2-2 : خلاصه مدل به همراه پارامتر های Pre-Trained شده VGG16

( ب )

طبق بالا ، خروجی مدل 1000 دسته می باشد در حالیکه دیتاست STL10 تنها 10 دسته دارد که باید تغییری در لایه آخر شبکه ایجاد کنیم. با این تفاسیر نیاز به آموزش بقیه وزن های مدل نداریم . این بار به روش ‘Classifier’ در مدل ( شکل 2-2-1 را نگاه بیندازید) را پیدا کرده و زیر لایه 6 آن که آخرین لایه Fully Connected می باشد را با شکل خروجی دیتاست اصلی تنظیم می کنیم :

```
# change the final layer (Using soft-code)
vggnet.classifier[6] = nn.Linear(vggnet.classifier[6].in_features,10)
```

شکل 3-2-3 : تغییر تعداد نورون خروجی لایه آخر

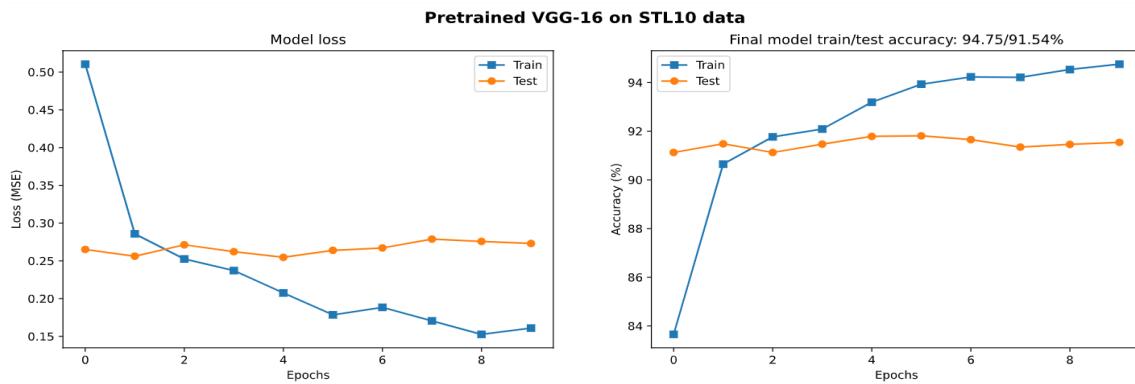
در نهایت مدل تغییر یافته را بر روی دیتاست اولیه برای 10 آموزش می دهیم:

```

Finished epoch 1/10. Test accuracy = 91.13%
Finished epoch 2/10. Test accuracy = 91.48%
Finished epoch 3/10. Test accuracy = 91.13%
Finished epoch 4/10. Test accuracy = 91.47%
Finished epoch 5/10. Test accuracy = 91.78%
Finished epoch 6/10. Test accuracy = 91.81%
Finished epoch 7/10. Test accuracy = 91.65%
Finished epoch 8/10. Test accuracy = 91.35%
Finished epoch 9/10. Test accuracy = 91.46%
Finished epoch 10/10. Test accuracy = 91.54%
```

شکل 2-2-4 : دقت بدست آمده در مجموعه تست برای epoch های مختلف

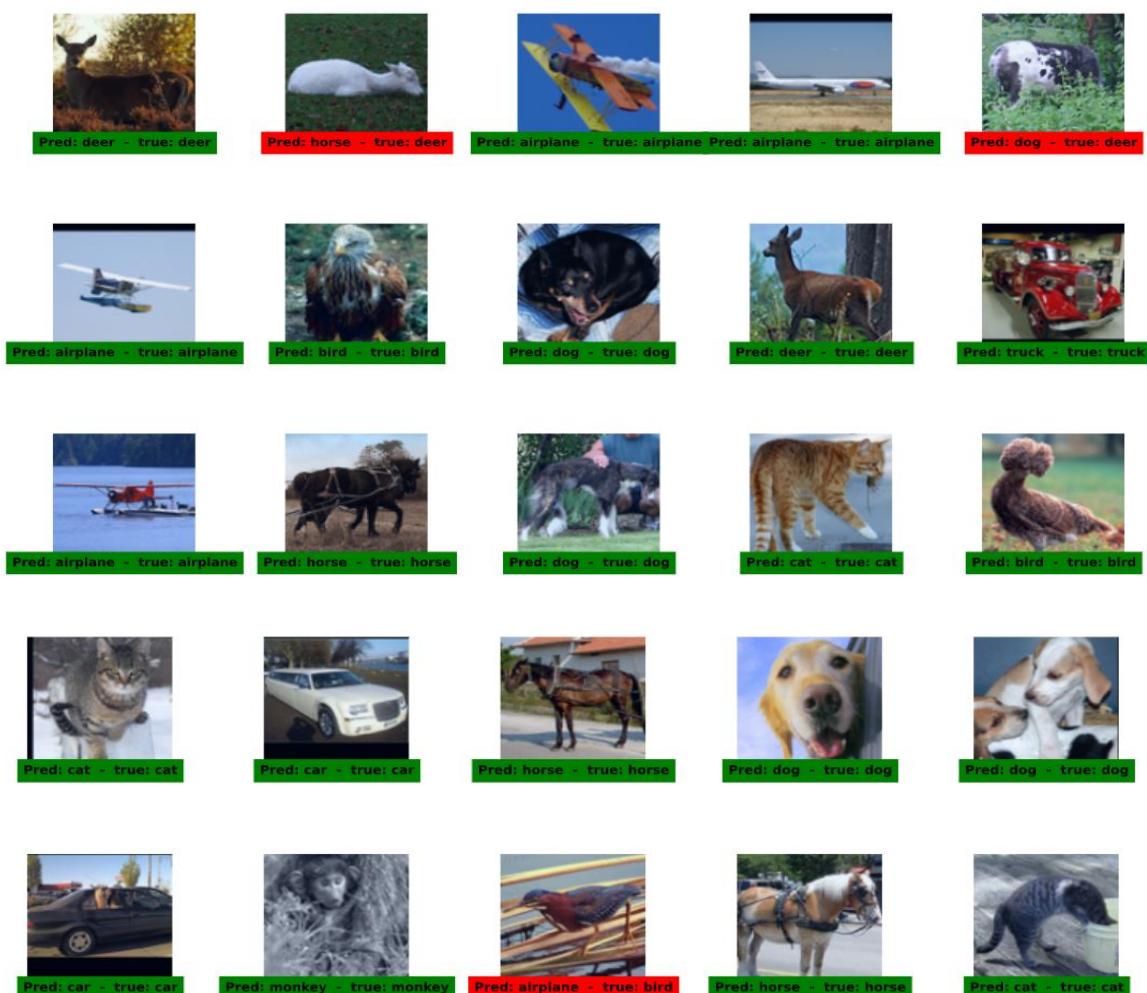
همچنین نمودار خطوط دقت به صورت زیر می باشد :



شکل 2-5 : خطای دقت بدست آمده برای مجموعه آموزش و تست

(ج)

در نهایت برای تعداد 25 تصویر به صورت رندوم ، کلاس اصلی و پیش بینی شده توسط مدل VGG16 را رسم می کنیم :



شکل 2-13 : تصاویر به همراه نوع لیبل اصلی و پیش بینی شده تعدادی رندوم از دادگان دیتابست STL10

همانطور که مشخص است در 25 عکس رندوم بالا ، سه عکس اشتباه لیبل خورده را مشاهده می کنیم. از این سه عکس برای بخش بعدی استفاده می کنیم .

( ۵ )

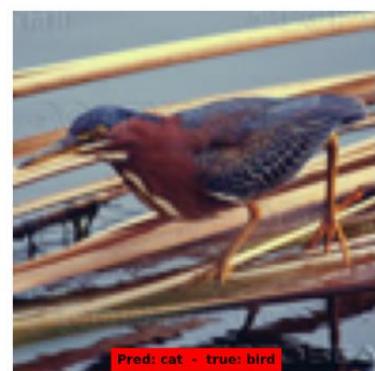
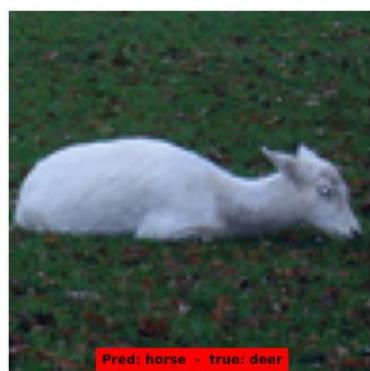
بار دیگر به سه عکسی که با کمک مدل Pre-Trained شده ، اشتباه لیبل خورده بود نگاه می اندازیم :



شکل 2-14 : لیبل های درست سه عکس اشتباه لیبل خورده توسط مدل Pre-Trained گرادیان هر لایه را فعال می کنیم ( در شکل 2-1-9 آنرا بر روی True قرار می دهیم ) و سپس مدل را برای این سه عکس از اول آموزش می دهیم و نتایج پیش‌بینی شده را می آوریم :

```
Finished epoch 1/5. Test accuracy = 0.00%
Finished epoch 2/5. Test accuracy = 0.00%
Finished epoch 3/5. Test accuracy = 0.00%
Finished epoch 4/5. Test accuracy = 0.00%
Finished epoch 5/5. Test accuracy = 0.00%
```

شکل 2-15-2 : دقت بدست آمده برای سه عکس آورده شده در بالا



شکل 2-16-2 : لیبل های جدید assign شده به تصویر Misclassified کل مدل

همانطور که در دو شکل 4-2-2 و 5-2-2 مشاهده شد ، مدل Pre-Trained شده VGG16 به مرتب بهتر نسبت به مدل Resnet18 عمل کرده ( بیش از 12 درصد اختلاف در دقت در دادگان تست ) و بنابراین ظرفیت مدل برای دقت اشباع شده به طوریکه با آموزش دوباره مدل و بدست آوردن وزن های جدید ، همچنان مدل قادر به پیش بینی درست 3 عکس Misclassified شده نخواهد بود.

## Semantic Segmentation – 3 سوال 3

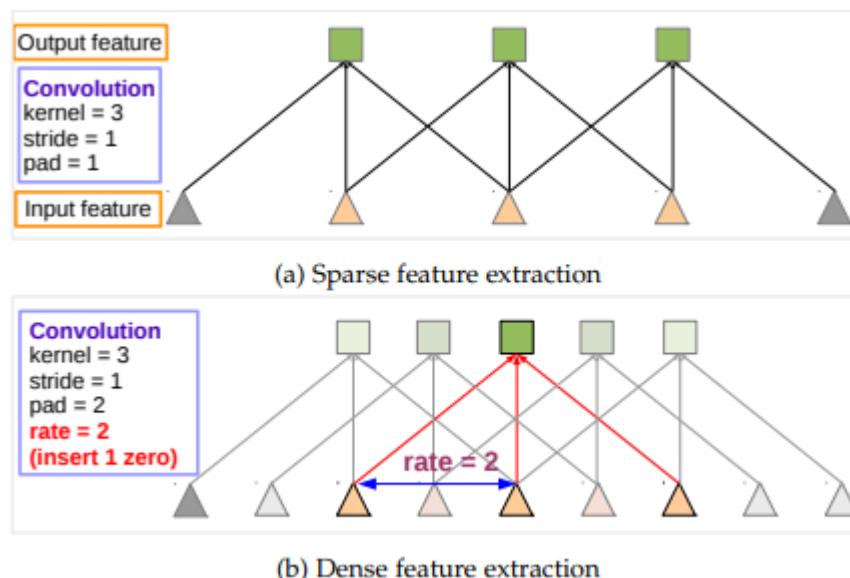
### الف) مدل انتخاب شده: DeepLab

برای توضیح معماری و کارکرد این مدل در ابتدا لازم است برخی ساختارهای به کار رفته در آن را به صورت اجمالی بررسی کنیم:

در حالت کلی برای اعمال Semantic Segmentation استفاده از شبکه‌های عصبی عمیق پیچشی راه حلی ساده و موفق است، با این حال تکرار ترکیب max-pooling و striding در لایه‌های پیاپی باعث کاهش قابل توجه رزولوشن feature maps حاصل خواهد شد. برای حل این مشکل از atrous پیچشی استفاده می‌کنیم. این الگوریتم به ما اجازه می‌دهد که پاسخ‌های هر یک از لایه‌ها را با هر رزولوشن مطلوبی محاسبه نماییم. برای تعریف دقیق‌تر atrous convolution را در فضای یک بعدی می‌بینیم که سیگنال  $y$  خروجی و  $x$  ورودی سیستم هستند:

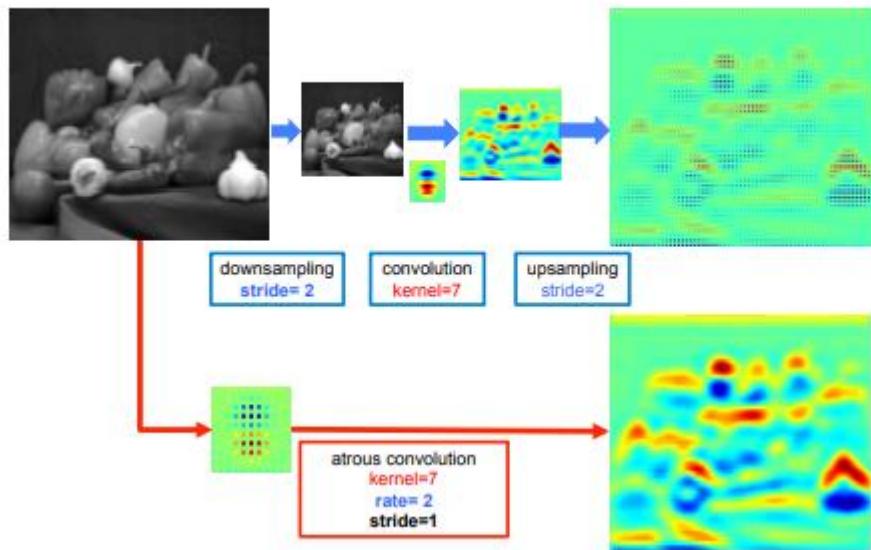
$$y[i] = \sum_{k=1}^K x[i + r \cdot k]w[k] \quad (1)$$

پارامتر  $r$  نشان دهنده stride است که با آن سیگنال ورودی را نمونه برداری می‌کنیم. در کانولوشن عادی مقدار این پارامتر  $r$  برابر 1 است. در زیر شکلی از این نوع کانولوشن و تاثیر پارامتر  $r$  را می‌بینیم:



شکل 3-1. معماری Atrous Convolution در فضای یک بعدی

اکنون تاثیر بکارگیری این کانولوشن بر روی یک عکس را با هم می‌بینیم:



شکل 3-2. مسیر پایین Atrous Convolution در فضای دو بعدی (عکس) و مسیر بالا روش‌های معمول مورد استفاده

به وضوح می‌توانیم توانایی و رزولوشن بالاتر کانولوشن Atrous را نسبت به حالت عادی ببینیم.

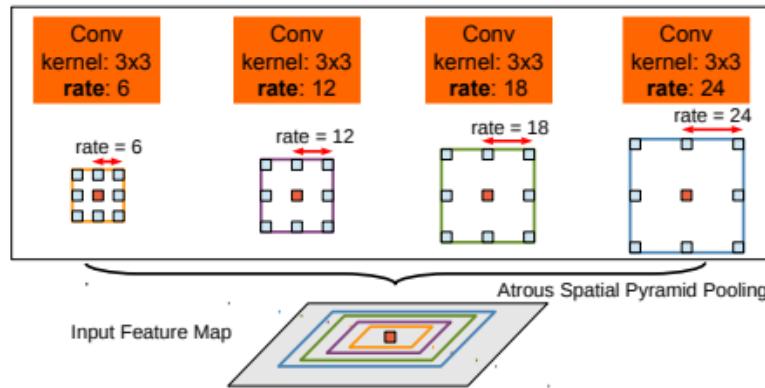
کانولوشن Atrous همچنین امکان بزرگ کردن دلخواه field-of-view را می‌دهد. DCNN ها به صورت عادی از کرنل‌های کانولوشنی کوچک (مثلا ۳ در ۳) استفاده می‌کنند. در Atrous با نرخ  $r-1$  صفر بین مقادیر پیاپی فیلترها قرار می‌دهیم که باعث بزرگتر شدن سایز کرنل می‌شود. مثلا اگر  $k \times k$  باشد به  $((k + (k - 1)(r - 1)) \times (k + (k - 1)(r - 1)))$  تبدیل می‌شود بدون اینکه محاسبات مورد نیاز را افزایش دهیم.

برای استفاده از کانولوشن atrous می‌توانیم فیلترها را با اضافه کردن صفرها (سوراخ‌ها) upsample کنیم. راه‌های دیگری هم برای اینکار وجود دارد.

**2. Atrous Spatial Pyramid Pooling**: اگر چه DCNN ها توانایی قابل توجهی در represent کردن اسکیل‌ها از خود نشان داده‌اند، اما با این حال امکان پیشرفت توانایی آن برای هندل کردن تشخیص اشیا بزرگ و کوچک وجود دارد. روش اول برای اینکار این است که شبکه‌های عمیق کانولوشنی را از ورژن‌های rescale شده متعدد عکس اولیه استخراج کنیم. برای اینکار از DCNN های موازی استفاده می‌کنیم. برای تولید نتیجه نهایی به صورت bilinearly مربوط به DCNN های موازی را به رزولوشن تصویر اصلی درون‌بایی می‌کنیم. ما اینکارها را هم در زمان تست و هم آموزش انجام می‌دهیم. اینکار در ازای هزینه محاسباتی قابل توجه منجر به تقویت عملکرد می‌شود.

راه دوم برای اینکار تاثیر گرفته از موفقیت *spatial pyramid pooling* استفاده شده در *R-CNN* است که نشان داد مناطق اسکیل دلخواه می‌توانند به دقیقی و موثری طبقه‌بندی شوند.

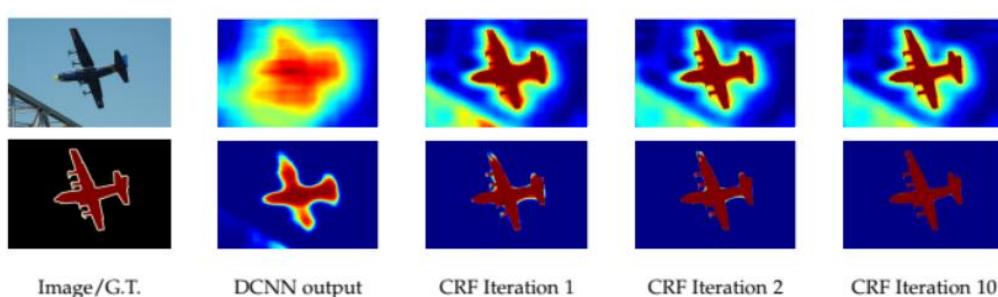
در شکل زیر نحوه عملکرد *ASPP* یا به اختصار *Atrous Spatial Pyramid Pooling* را داریم:



شکل 3-3. *ASPP* برای طبقه‌بندی پیکسل نارنجی رنگ موجود در تصویر

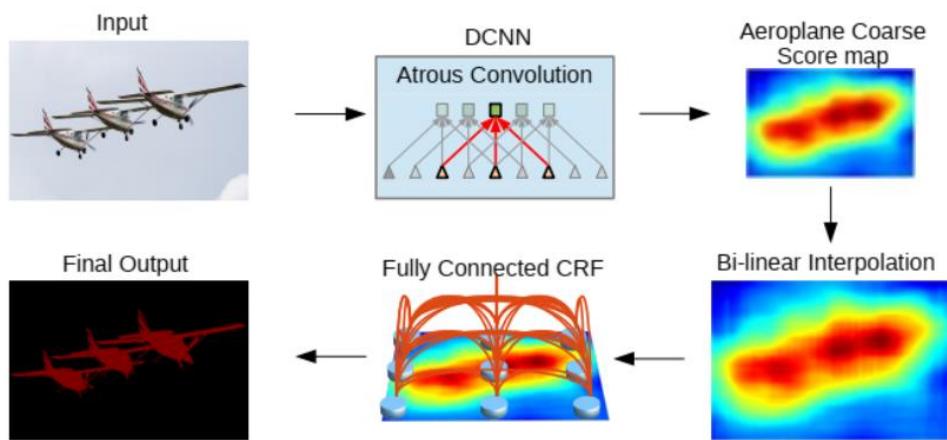
3. پیش‌بینی‌های ساختاریافته با استفاده از **Random Field** های کاملاً متصل: با توجه به یافته‌های پیشین از DCNN ها score map ها توانایی پیش‌بینی حضور اشیا را دارند اما به درستی نمی‌توانند مرزها را مشخص کنند. برای حل این مشکل و پیدا کردن اطلاعات با جزئیات مناسب از شبکه کاملاً متصل CRF استفاده می‌کنیم. CRF شامل ترم‌های Smoothness می‌شود که اتفاق نظر بر روی لیبل پیکسل‌های مشابه را بیشینه می‌کند و می‌تواند ترم‌های پیچیده‌تری بین اشیا کلاس‌ها را تشکیل دهد.

شکل‌های زیر تاثیر استفاده از CRF بر segmentation map را نمایش می‌دهد.



شکل 4-3. تاثیر استفاده از CRF بر لبه‌یابی اشیا

حال عکس‌ها را به عنوان ورودی گرفته و از لایه‌های معمول DCNN عبور می‌دهیم. در ادامه یک یا دو لایه اضافه می‌کنیم که در نتیجه coarse score map را به ما می‌دهد. در ادامه این map را با استفاده از درونیابی up-sampled, bi-linear از لایه کاملاً متصل CRF استفاده می‌کنیم. در زیر شکل این مراحل و معماری مدل آمده:



شکل ۳-۵ معماری کامل مدل DeepLab ورژن ۱

(ب)

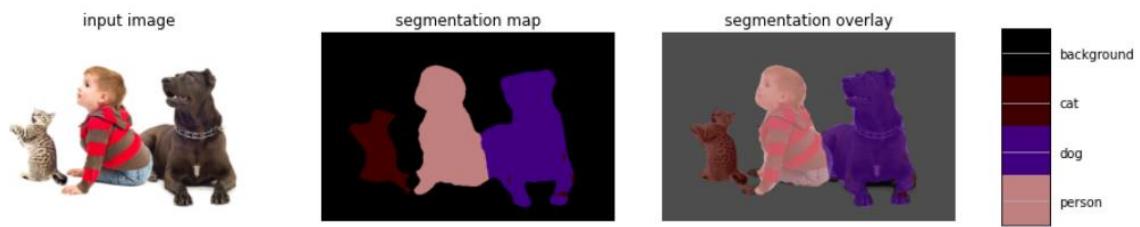
اکنون که با شیوه کار مدل DeepLab آشنا شدیم، با استفاده از یک عکس دلخواه خروجی مدل را بررسی می کنیم:



شکل ۳-۶. عکس مورد نظر برای آزمایش مدل DeepLab

با دریافت کدهای این مدل از مسیر مربوطه<sup>۱</sup> عکس مورد نظر خود را به عنوان ورودی به شبکه عصبی مدل DeepLab می دهیم. خروجی حاصل به شکل زیر خواهد بود:

<sup>1</sup> <https://paperswithcode.com/paper/deeplab-semantic-image-segmentation-with-deep>



شکل 7-3 . خروجی مدل DeepLab بر روی عکس نمونه

همان‌طور که می‌بینیم عمل لبه‌یابی و پیدا کردن هر یک از اجسام و تفکیک آن‌ها از پیش‌زمینه و هم‌چنین طبقه‌بندی صحیح آن‌ها به طور کامل انجام شده است.

## سوال 4 – Object Detection

(الف)

شبکه‌های تشخیص شیء دو مرحله‌ای<sup>۱</sup>: در این شبکه عمل مکانیابی و طبقه‌بندی<sup>۲</sup> به صورت جداگونه اجرا می‌شود. اصولاً دقت تشخیص شی بالایی دارند که در ازای هزینه محاسباتی و سرعت پایین‌تر به دست می‌آید. همچنین در این روش غالباً آموزش شبکه به صورت end-to-end اتفاق نمی‌افتد. به عنوان مثال‌هایی از این نوع شبکه تشخیص شی می‌توان به RCNN<sup>3</sup> ، Faster RCNN<sup>4</sup>، Mask RCNN و G-RCNN اشاره کرد که از مدل‌های رایج این نوع شبکه هستند.

شبکه‌های تشخیص شیء تک مرحله‌ای<sup>۳</sup>: در این روش بخلاف مدل‌های تشخیص شیء چند مرحله‌ای اعمال مکانیابی و طبقه‌بندی اشیا به صورت یکجا و در یک مرحله انجام می‌شود. مهمترین برتری این روش به روش‌های چند مرحله‌ای سرعت بسیار بالا در خروجی تولید کردن است اما این مدل‌ها در تشخیص اشیا با اشکال غیرمتربقه توانایی و دقت مدل‌های چند مرحله‌ای را ندارند. به عنوان مثال‌هایی از این نوع شبکه تشخیص شی می‌توان به SSD<sup>5</sup>، Yolo<sup>6</sup> و RetinaNet<sup>7</sup> اشاره کرد.

(ب)

در YOLOv1 مشکلی که در اشیا بسیار نزدیک و یا دارای تداخل وجود دارد این است که به دلیل محدودیت مدل هر grid تنها تعداد بسیار محدودی چهارضلعی جداکننده را می‌تواند تعیین کند در نتیجه در تراکم‌های بالا یا تداخل مدل توانایی تشخیص و جداسازی مناسبی ندارد.

از طرفی در YOLOv2، با اضافه کردن Batch Normalization و باکس‌های لنگری<sup>۸</sup> توانایی جایابی و تقسیم به چهارضلعی‌های جداکننده مدل را افزایش میدهیم که در نتیجه باعث کمزنگ شدن مشکل تشخیص اشکال در زمان تداخل YOLOv1 می‌شود.

(ج)

در ورژن‌های YOLOv4 و YOLOv5، با بهینه‌تر کردن پروسه آموزش و اضافه کردن bag-of-freebies و bag-of-specials دقت و سرعت شبکه را نسبت به ورژن‌های قبلی افزایش داده‌اند. همچنین در مقایسه با مدل YOLOv3، شاهد افزایش میزان FPS نیز هستیم. همچنین از لحاظ مقایسه سرعت بین این دو

Two Stage Detectors<sup>1</sup>

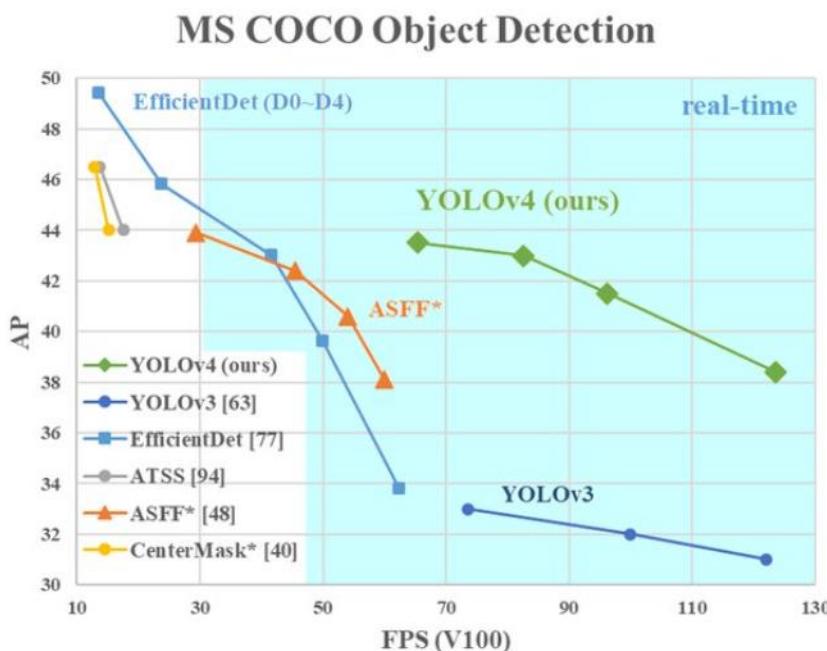
Classification<sup>2</sup>

One Stage Detectors<sup>3</sup>

Anchor Boxes<sup>4</sup>

مدل YOLOv5 سرعت بسیار بیشتری از خود نشان داده که از مهمترین برتری‌های این مدل بر تمامی مدل‌های دیگر است.

برای مثال در شکل زیر مقایسه دقت میانگین مابه‌ازای FPS مشخص را بین مدل‌های YOLOv3 و YOLOv4 داریم که حاکی از برتری واضح مدل دوم است:



شکل 4-1. نمودار دقت میانگین بر حسب FPS برای دو مدل YOLOv3 و YOLOv4

(د)

به صورت کلی هفت هایپرپارامتر برای مدل YOLOv3 در نظر گرفته می‌شود که در ادامه عناوین و توضیحات مربوط به هر یک آمده است.

1. هایپرپارامتر **Batch**: این پارامتر همان‌طور که از اسم آن بر می‌آید نشان دهنده اندازه دسته‌های داده می‌باشد که در الگوریتم Mini-Batch مورد استفاده قرار می‌گیرد. همانند گذشته دلیل احتیاج به این الگوریتم و پارامتر در زمان‌هایی است که تعداد داده‌های ورودی آموزش بسیار زیاد باشند و استفاده از تمام داده‌ها در هر تکرار هزینه محاسباتی و زمانی بسیار بالایی داشته باشد. راه تعیین این پارامتر امتحان مقداری مختلف برای آن و در نهایت انتخاب مدل برنده است.

2. هایپرپارامتر **Subdivisions**: این پارامتر که توسط Darknet مهیا شده است این اجازه را به کاربر می‌دهد که در شرایطی که GPU مورد استفاده حافظه کافی برای استفاده از اندازه دسته برای مثال مثال برابر 64 را نداشته باشد، به صورت تکه تکه این دسته‌ها 64 عضوی را پردازش کند. برای تعیین این پارامتر

می‌توان از مقدار  $1 = \text{subdivision}$  شروع کرده و در صورتی که دچار خطای حافظه شدیم، به مرور مقادیر بالاتر مانند  $16, 4, 8, \dots$  را امتحان می‌کنیم تا زمانی که خطای مربوطه بر طرف شود.

3. هایپرپارامترهای طول<sup>۱</sup>، عرض<sup>۲</sup> و کanalها<sup>۳</sup>: این پارامترها تعیین کننده سایز تصویر ورودی و تعداد کanalها هستند. می‌توان از مقادیر تیپیکالی مانند 416 در 416 یا 608 در 608 برای طول و عرض استفاده کرد. همچنین اگر برای مثال سه کanal رنگ RGB مدنظر است می‌توانیم پارامتر Channels را برابر 3 قرار دهیم.

4. هایپرپارامترهای Momentum و Decay: پارامتر Momentum برای جریمه دار کردن تغییرات بزرگ در وزن‌های شبکه ایجاد شده است تا از مشکلات احتمالی پیشگیری کند. از طرفی برای جلوگیری از بیشبرازش<sup>۴</sup> پارامتر Decay تعریف شده است تا با به اندازه کافی کوچک انتخاب شدن بتواند وزن‌های بزرگ که باعث بیشبرازش شده‌اند را کنترل کند. برای تعیین این پارامترها اگر در حالت دیفالت دچار بیشبرازش نمی‌شیم نیازی به ایجاد تغییر نیست در غیر این صورت می‌توان با استفاده از داده‌های اعتبارسنجی مقدار این پارامترها را تنظیم کرد.

5. هایپرپارامترهای نرخ آموزش، تعداد گام‌ها، Burn in Scale و Scale: نرخ آموزش<sup>۵</sup> همانطور که می‌دانیم تعیین کننده شدت یادگیری داده‌ها است که معمولاً عددی در بازه 0.01 تا 0.0001 اخذ می‌شود. در ابتدای امر آموزش با توجه به کمبود اطلاعاتی که داریم نرخ آموزش باید بزرگ باشد اما رفته نیاز داریم که تغییرات در وزن‌ها کوچک‌تر باشد. به عبارتی نرخ یادگیری باید در طول پروسه یادگیری کاهش یابد. روشی که در اینجا برای کاهش آن انتخاب شده است این است که به اندازه پارامتر تعداد گام‌ها تکرار، مقدار نرخ یادگیری ثابت باشد و پس از آن به اندازه پارامتر Scale کوچک شود و این پروسه تکرار شود. با در نظر داشتن موارد ذکر شده، در آزمایشات تجربی ثابت شده است که اگر در ابتدای آموزش نرخ یادگیری برای مدت زمان کوتاهی کوچک شود سرعت آموزش افزایش می‌یابد. این عمل توسط پارامتر Burn-in کنترل می‌شود.

---

Height<sup>۱</sup>  
Width<sup>۲</sup>

Channels<sup>۳</sup>

Overfitting<sup>۴</sup>

Learning Rate<sup>۵</sup>

6. هایپرپارامترهای **Data Augmentation**: همانطور که می‌دانیم پروسه جمع‌آوری داده برای آموزش مدل‌ها پروسه‌ای پرهزینه و زمان بر است. برای همین به دنبال این هستیم که از تعداد محدودی داده جمع‌آوری شده نهایت استفاده را ببریم. این پروسه Data Augmentation نامیده می‌شود. برای مثال اگر در یک عکس تصویر یک آدم برفی را داشته باشیم، تصویر آن وقتی ۵ درجه چرخیده شده است همچنان یک آدم برفی است منتهی یک داده کاملاً جدید محسوب می‌شود. این پارامتر angle نامیده می‌شود که به صورت رندوم به ازای ورودی یک داده را به اندازه  $\pm angle$  عکس را می‌چرخاند. همچنین اگر رنگ‌های عکس را با استفاده از پارامترهای exposure، saturation و hue تغییر دهیم مانند حالت قبل است که می‌توانند به افزایش اندازه داده‌های ما کمک کنند.

7. هایپرپارامتر تعداد تکرارها<sup>1</sup>: همانطور که از اسم آن بر می‌آید به دنبال تنظیم تعداد تکرارهای آموزش خود هستیم. به صورت کلی برای انتخاب این پارامتر توصیه می‌شود که اگر یک مدل تشخیص شی برای n کلاس داریم، تعداد تکرارها حداقل  $n^{*}2000$  مرتبه باشد.

## (۵)

نوت‌بوک مربوطه به آموزش شبکه YOLOv5 مورد استفاده از 8 بخش اصلی تشکیل شده است که هر کدام به ترتیب مراحل زیر را طی می‌کنند:

در ابتدا پیش‌نیازها و متعلقات مدل YOLOv5 را نصب می‌کنیم تا سیستم بتواند اعمال مربوط به این مدل را انجام دهد.

در ادامه بارگذاری فایل Roboflow.zip که شامل تصاویر آموزش و ارزیابی و همچنین لیبل‌های مربوط به هر کدام است را انجام داده و با استفاده از دستورات موجود فایل زیپ را باز می‌کنیم.

حال تنظیمات آموزش مدل YOLOv5 را انجام می‌دهیم شامل مشخصاتی مانند سایز ورودی عکس‌ها، اندازه دسته‌ها<sup>2</sup>، تعداد تکرارها<sup>3</sup>.

اکنون مدل آماده آموزش دادن است و با استفاده از دستور train.py آموزش را شروع می‌کنیم. قبل از اینکار اطمینان حساب می‌کنیم که runtime بر روی حالت GPU قرار گرفته باشد.

اکنون پس از اتمام مرحله آموزش، مدل YOLOv5 خود را با استفاده از داده‌های ارزیابی مورد بررسی قرار می‌دهیم و نتایج به دست آمده را تحلیل می‌کنیم.

---

Iterations<sup>1</sup>  
Batch Size<sup>2</sup>  
Epochs<sup>3</sup>

ابتدا آموزش و بررسی مدل YOLOv5s را انجام می‌دهیم، برای اینکار ابتدا فایل داده‌ای که در اختیار داریم را در محیط کدنویسی unzip می‌کنیم و با جایگزینی فایل با فرمت yaml کار آموزش را شروع می‌کنیم :

زمان آموزش:

1h 16m 23s

برای این مدل زمان قابل توجه یک ساعت و 16 دقیقه‌ای نیاز بود تا مرحله آموزش انجام شود.

اکنون با تکمیل مرحله 100 ایپاکی آموزش گزارش نهایی را مورد بررسی قرار می‌دهیم:

Epoch	gpu_mem	box	obj	cls	total	targets	img_size
99/99	1.82G	0.04915	0.04028	0.008471	0.0979	270	416: 100% 99/99 [00:43<00:00, 2.27it/s]
	Class	Images	Targets	P	R	mAP@.5	mAP@.5:.95: 100% 2/2 [00:02<00:00, 1.05s/it]
	all	58	1.00e+03	0.965	0.998	0.925	0.573
	blue	58	115	1	0.969	0.995	0.6
	green	58	290	0.995	0.945	0.989	0.574
	red	58	290	0.997	0.99	0.996	0.657
	vline	58	136	0.968	0.978	0.977	0.826
	white	58	58	0.85	0.586	0.601	0.165
	yellow	58	116	0.983	0.983	0.994	0.615
Optimizer stripped from runs/train/yolov5s_results/weights/last.pt, 14.8MB							
Optimizer stripped from runs/train/yolov5s_results/weights/best.pt, 14.8MB							
100 epochs completed in 1.263 hours.							
CPU times: user 48.3 s, sys: 7.16 s, total: 55.5 s							
Wall time: 1h 16min 23s							

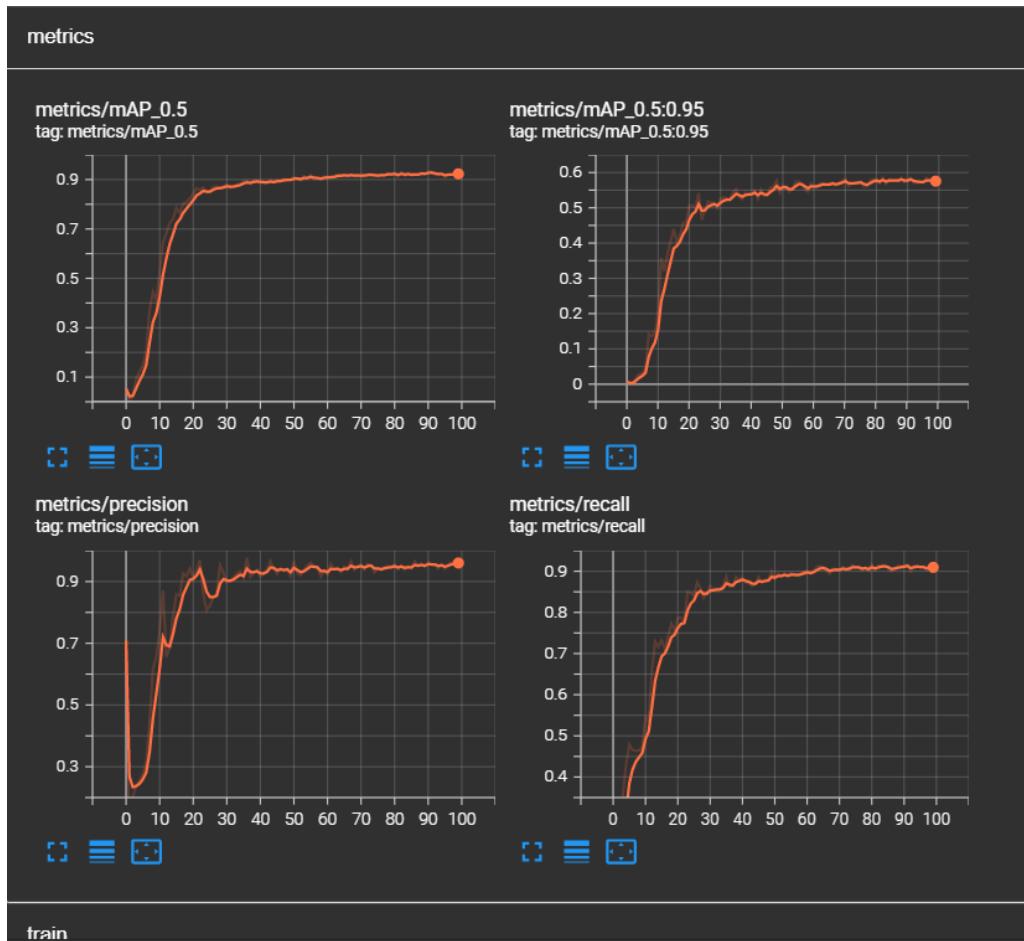
شکل 4-2. گزارش نهایی آموزش مدل YOLOv5s

همان‌طور که در شکل بالا می‌بینیم هر یک از موارد مورد نظر برای تمامی کلاس‌ها به صورت یکجا و جداجدا آورده شده است. در حالت یکجا داریم:

جدول 4-1. مقادیر mAP مدل YOLOv5s

all	mAP 0.5	mAP 0.5:0.95
YOLOv5s	0.925	0.573

در مرحله بعد به سراغ رسم نمودارهای Precision و Recall و mAP 0.5:0.95، mAP 0.5 و mAP 0.5:0.95 می‌رویم:



شکل 4-3. نمودارهای  $mAP$  و  $Precision$  و  $Recall$  برای مدل  $YOLOv5s$  برای داده‌های تمرین

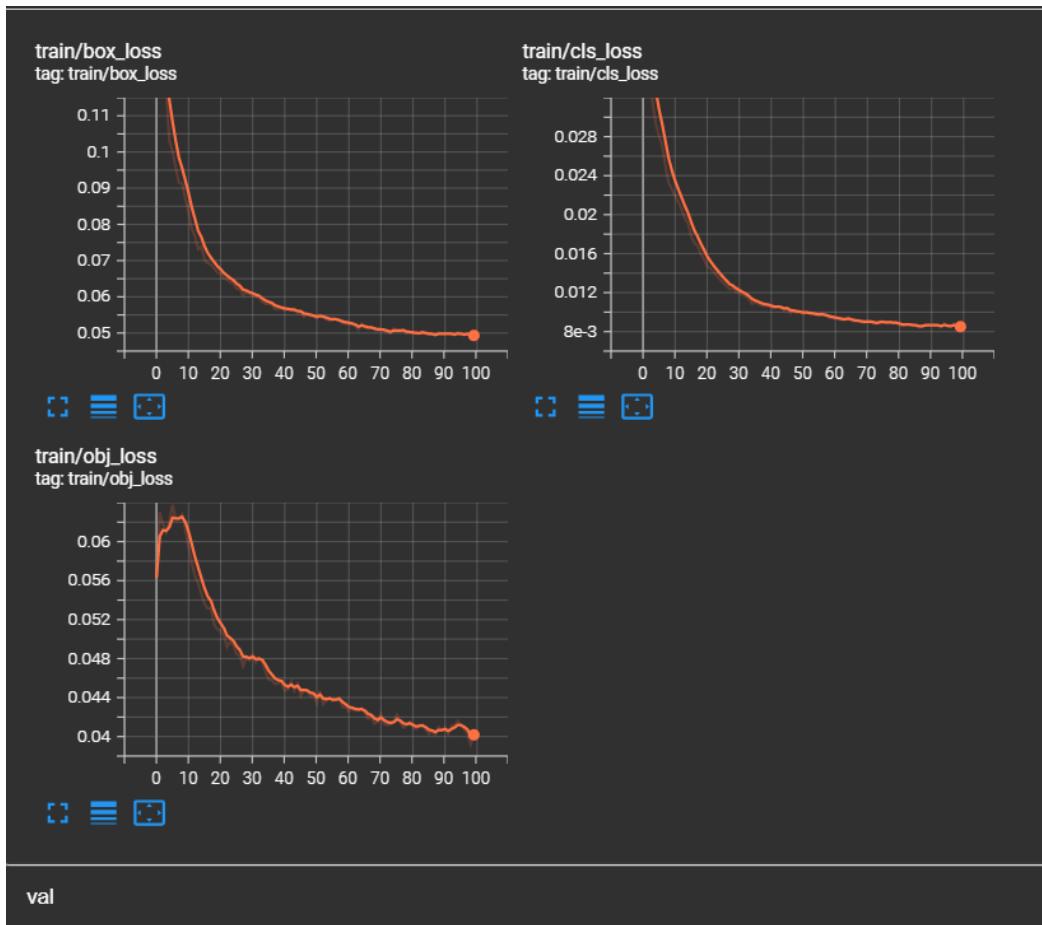
روند تغییرات هر یک از این چهار متغیر در روند آموزش را در شکل‌های بالا می‌بینیم.

جدول 4-2. مقادیر  $Recall$  و  $Precision$  مدل  $YOLOv5s$

train	Precision	Recall
YOLOv5s	0.971	0.91

و در ادامه نمودار تغییرات هزینه برای داده‌های ارزیابی<sup>1</sup> را داریم:

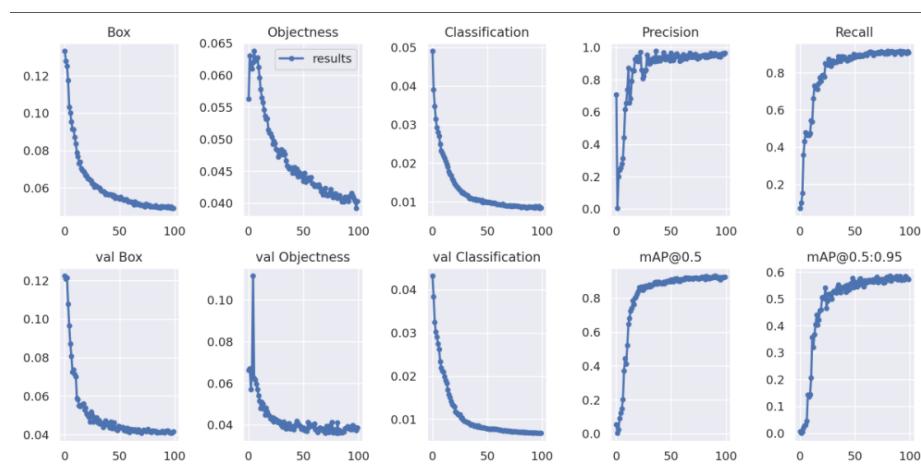
Cross Validation Set<sup>1</sup>



شکل 4-4. نمودارهای تغییرات هزینه برای مدل YOLOv5s برای داده‌های ارزیابی

مطابق انتظار با جلو رفتن روند تمرین مقدار خطأ کاهش می‌یابد تا به حدود 0.04 برسد که مقدار هزینه قابل قبولی است.

شکل زیر گزارشی از نوتبوک مورد استفاده است که به صورت یکجا نمودار متغیرهای مختلف که برای ارزیابی مدل مناسب هستند را آورده است.



شکل 4-5. نمودار متغیرهای ارزیابی بر روی داده‌های آموزش و ارزیابی به صورت یکجا

اکنون چند نمونه از نحوه عمل مدل بر روی عکس‌هایی از مجموعه داده آموزش را می‌بینیم:



شکل 4-6. تشخیص مدل آموزش داده شده YOLOv5s بر روی داده‌های آموزش

همان‌طور که انتظار می‌رفت مدل YOLO هر یک از اشیا موجود در تصویر را به کلاس مربوط خود که در اینجا رنگ آن‌ها است نسبت می‌دهد.

در ادامه تشخیص‌های مدل بر روی داده‌های Augmentation شده آمده است:



شکل 4-7. تشخیص مدل آموزش داده شده YOLOv5s بر روی داده‌های Augmentation بافته

اکنون مدل YOLOv5x را آموزش داده و اطلاعاتی که برای مدل قبل داشتیم مجدد استخراج می‌کنیم:

مدت زمان آموزش: 1 ساعت و 16 دقیقه و 41 ثانیه

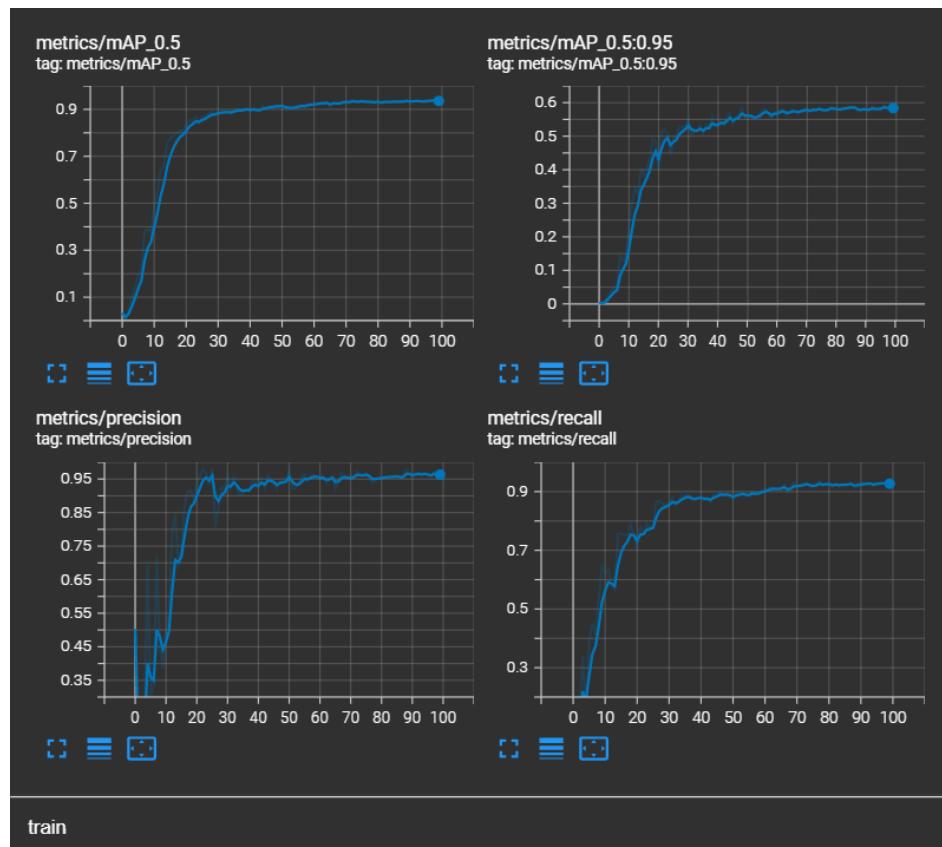
Epoch	gpu_mem	box	obj	cls	total	targets	img_size
99/99	1.81G	0.04911	0.0402	0.008575	0.09788	270	416: 100% 99/99 [00:44<00:00, 2.25it/s]
		Class	Images	Targets	P	R	mAP@.5 mAP@.5: .95: 100% 2/2 [00:02<00:00, 1.05s/it]
		all	58	1.00e+03	0.965	0.928	0.937 0.582
		blue	58	115	0.991	0.978	0.995 0.611
		green	58	290	0.996	0.959	0.985 0.58
		red	58	290	1	0.99	0.995 0.638
		vline	58	136	0.965	0.978	0.974 0.858
		white	58	58	0.848	0.672	0.676 0.187
		yellow	58	116	0.991	0.988	0.995 0.62
Optimizer stripped from runs/train/yolov5x_results/weights/last.pt, 14.8MB							
Optimizer stripped from runs/train/yolov5x_results/weights/best.pt, 14.8MB							
100 epochs completed in 1.270 hours.							
CPU times: user 49.2 s, sys: 7.29 s, total: 56.5 s							
Wall time: 1h 16min 41s							

شکل 4-8. گزارش نهایی آموزش مدل YOLOv5x

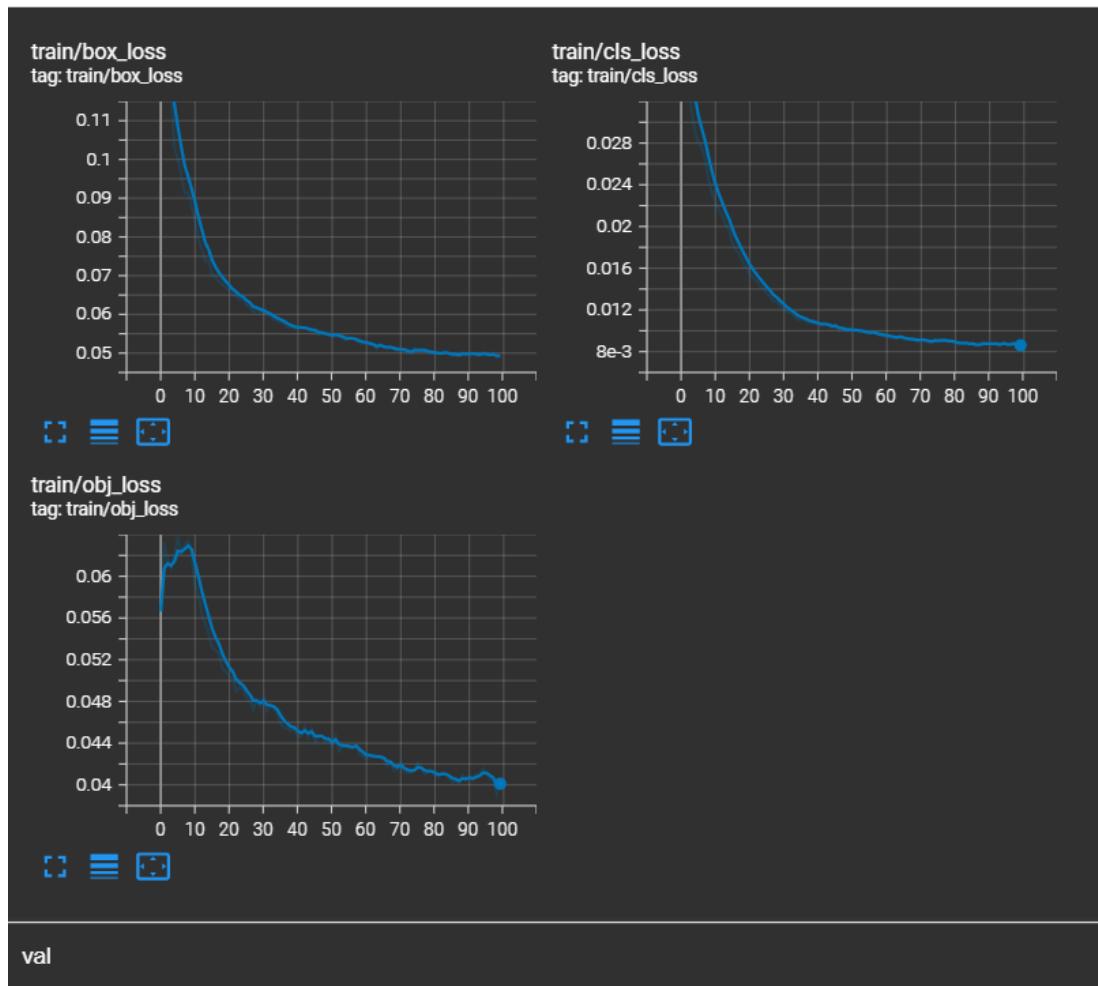
جدول 4-3. مقادیر mAP مدل YOLOv5s

all	mAP 0.5	mAP 0.5:0.95
YOLOv5x	0.937	0.582

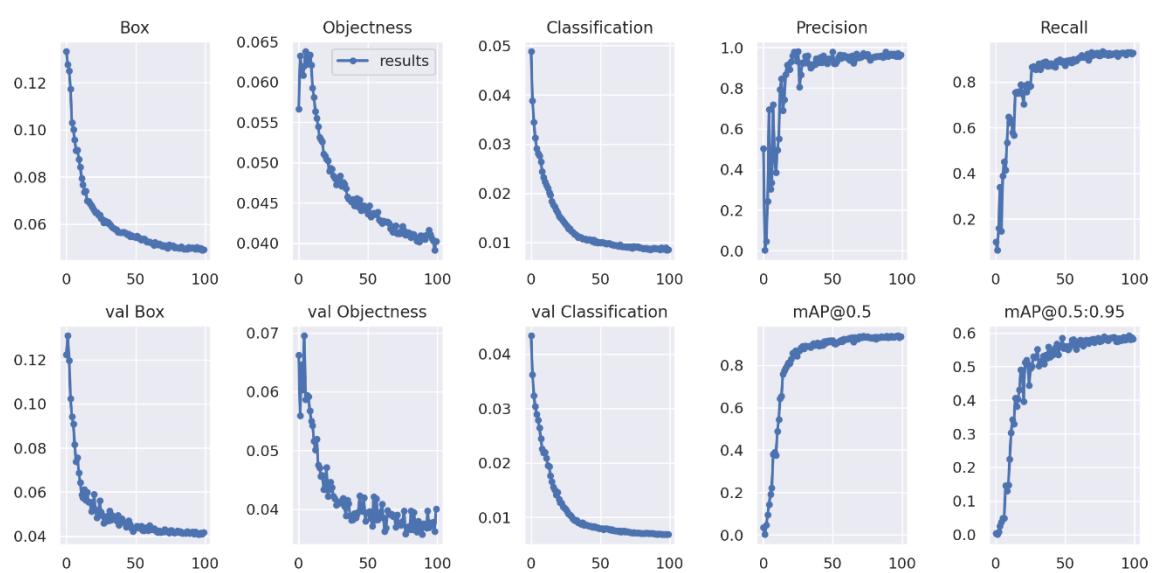
در مرحله بعد به سراغ رسم نمودارهای Precision و Recall و mAP 0.5:0.95، mAP 0.5 و mAP 0.95 می‌رویم:



شکل 4-9. نمودارهای Recall و Precision و mAP برای مدل YOLOv5x برای داده‌های تمرین



شکل 4-10. نمودار تغییرات هزینه برای مدل *YOLOv5x*



شکل 4-11. نمودارهای اطلاعات کلی مدل *YOLOv5x*

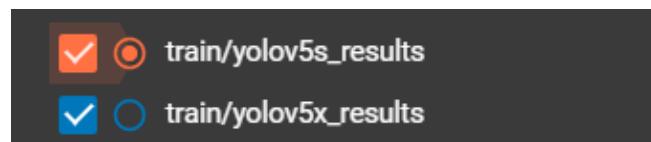


شکل ۱۲-۴. تشخیص مدل آموزش داده شده YOLOv5x بر روی داده‌های آموزش

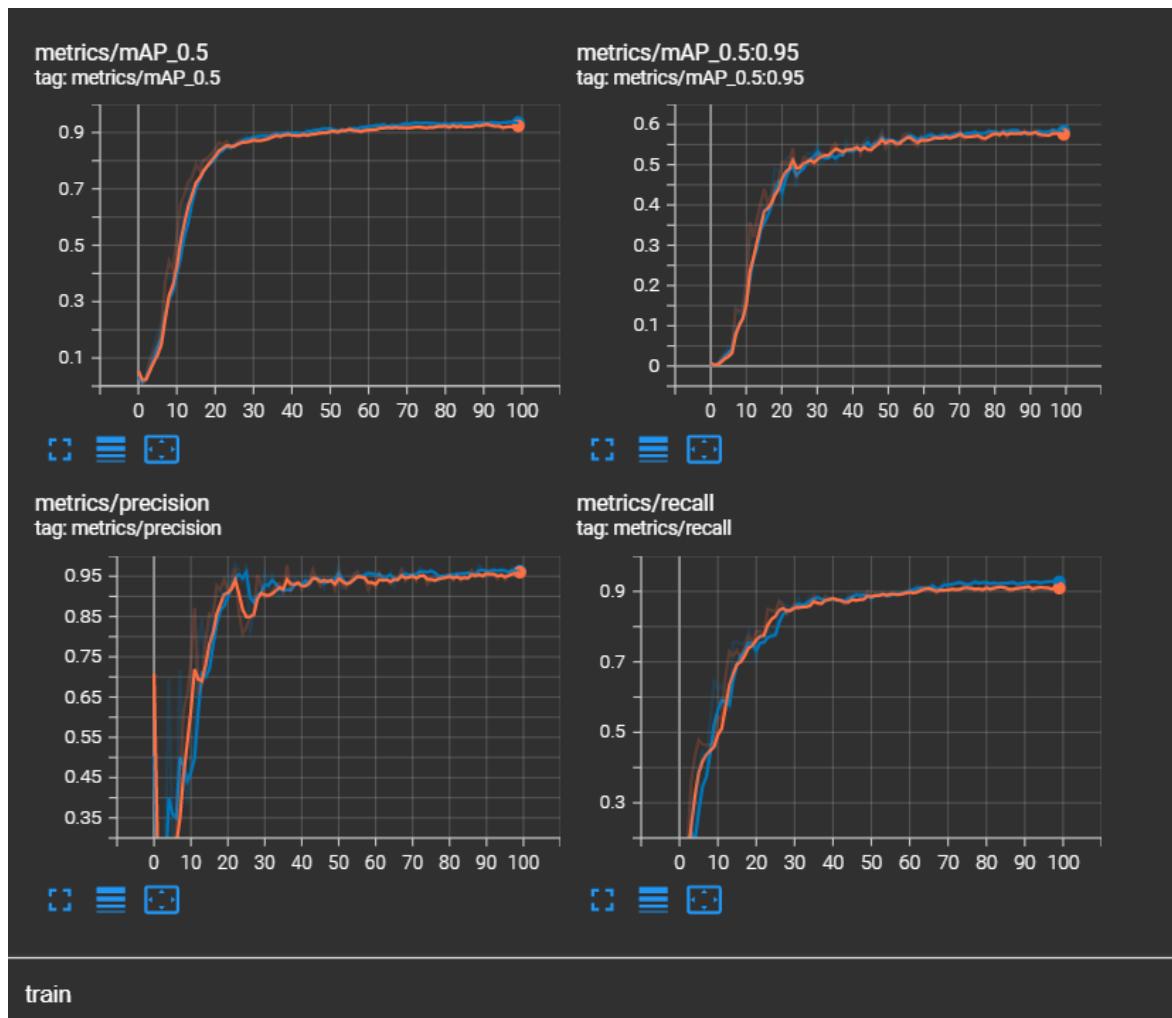


شکل ۱۳-۴. تشخیص مدل آموزش داده شده YOLOv5x بر روی داده‌های Augmentation یافته

اکنون به مقایسه دو مدل YOLOv5x و YOLOv5s می‌پردازیم. ابتدا نمودارهای مربوط به هزینه و mAP را روی هم رسم می‌کنیم:



شکل ۱۴-۴. راهنمای شکل‌های ۱۴-۴ و ۱۵-۴



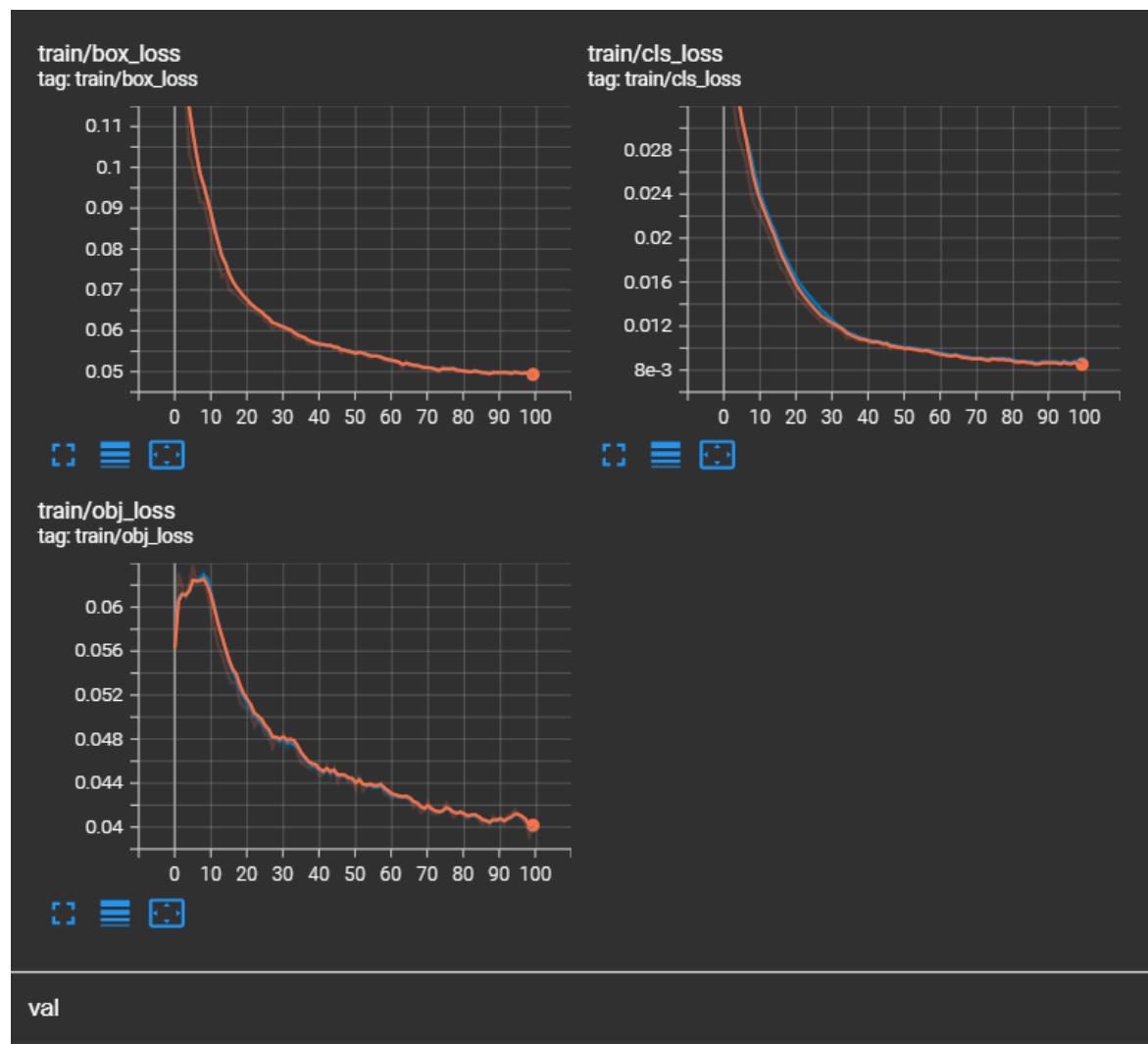
شکل ۱۴-۴. نمودار تغییرات mAP ۰.۵ و mAP ۰.۹۵، Precision و Recall دو مدل v5x و v5s

جدول ۴-۴. مقایسه دو مدل Recall و Precision

train	Precision	Recall
YOLOv5s	0.971	0.91
YOLOv5x	0.97	0.92

جدول 4-5. مقایسه  $mAP 0.5$  و  $mAP 0.95$  دو مدل

train	$mAP 0.5$	$mAP 0.95$
YOLOv5s	0.971	0.91
YOLOv5x	0.937	0.582



شکل 4-16. نمودار تغییرات هزینه دو مدل

همان طور که دیدیم، هر دو مدل دقت، Precision و Recall تقریباً یکسانی دارند با این حال در مدل بزرگتر (YOLOv5x) در مدت زمان کمی بیشتر (در اینجا در حد چند ثانیه) به mAP ها و Recall و Precision حدوداً یکی دو درصدی بهبود یافته رسید. در صورت کلی می‌توان نتیجه‌گیری کرد که مدل بزرگتر در مدت زمان بیشتر به دقت و عملکرد بهتری دست خواهد یافت.

---