به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی و یادگیری عمیق

HW EXTRA I

نام و نام خانوادگی	**محمدعلی شاکردرگاه**
شماره دانشجویی	810196487
تاریخ ارسال گزارش	1400/03/17

فهرست گزارش سوالات

سوال 1 – Object detection with YOLOv5

مقصود از این سوال، آشنایی بیشتر با YOLOv5 میباشد ،که مختصر شده عبارت You Only Look Once میباشد، که بتوان با استفاده از آن Object detection انجام داد، Dataset ما برای این مسئله شامل تصاویری میشود که مربوط به بازی bocco ball است که کلاس های ما در این Dataset میشوند:

1- توپ های سفید
2- توپ های قرمز
3- توپ های سبز
4- توپ های آبی
5- توپ های زرد
6- خط های عمودی زمین

حال قسمت به قسمت سعی میشود به سوالات پاسخ داده شود.

1) پیشرفت های ورژن 4 ,5 خانواده های YOLO نسبت به 1,2 و 3 آن

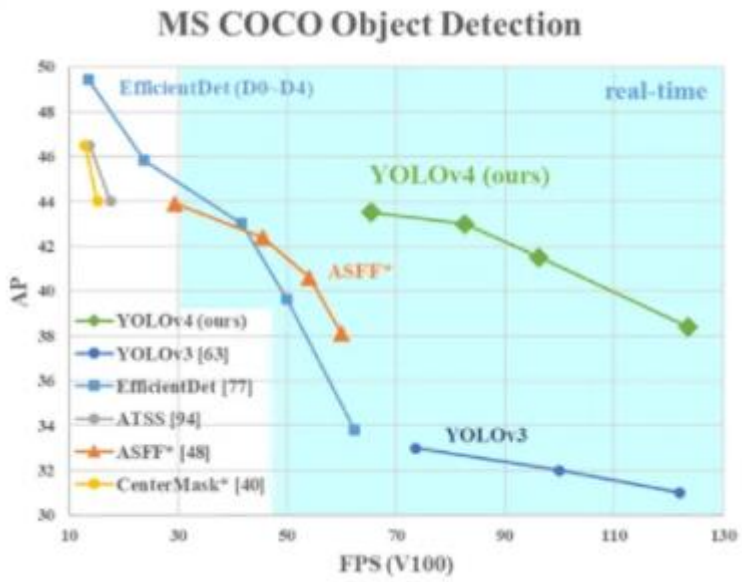
ابتدا به تعریف مختصری از YOLO میپردازیم و سپس ویژگی های کلی نسخه های 1و2و3 آن را بیان میکنیم و سپس به پیشرفت های ورژن 4 و 5 میپردازیم.

YOLO به منظور تشخیص Object استفاده میشود و میتواند با دقت مناسبی این امر را انجام دهد، در سال 2016، YOLOv1 با عنوان You Only Look Once (با این مفهوم که همانند چشم های ما که در لحظه میبینند و پردازش توسط مغز ما صورت میگیرد عمل میکنند) معرفی شد که توانایی Real time برای تشخیص Object داشت، بعد از آن در سال 2017 ورژن دوم آن با نام YOLOv2 معرفی شد که نه تنها سریع تر بود بود بلکه دقت Robustness بیشتری نیسز داشت، در سال 2018 نیز نسخه جدید آن با نام YOLOv3 معرفی شد که Incremental Improvement را در آن شاهد بودیم.

در سال های 2019 توسط Alexey Bochkovskiy و 2020 توسط Glenn Jocher ورژن های بعدی یعنی YOLOv4 و YOLOv5 معرفی شدند که در آن ها شاهد تغییرات بسیار زیادی بودیم.

YOLOv4 امکان Train کردن با دقت بالا را روی GPU با 1080 TI یا 2080 TI را فراهم میکرد. همچنین تاثیر "bag-of-freebies" state of art و "bag-of-specials" که متد هایی برای تشخیص object هستند verified شد و این متد ها که شامل CBN و PAN با این ورژن هم efficient تر عمل میکنند هم برای کار با gpu میتوان آن ها را suitable تر دانست.

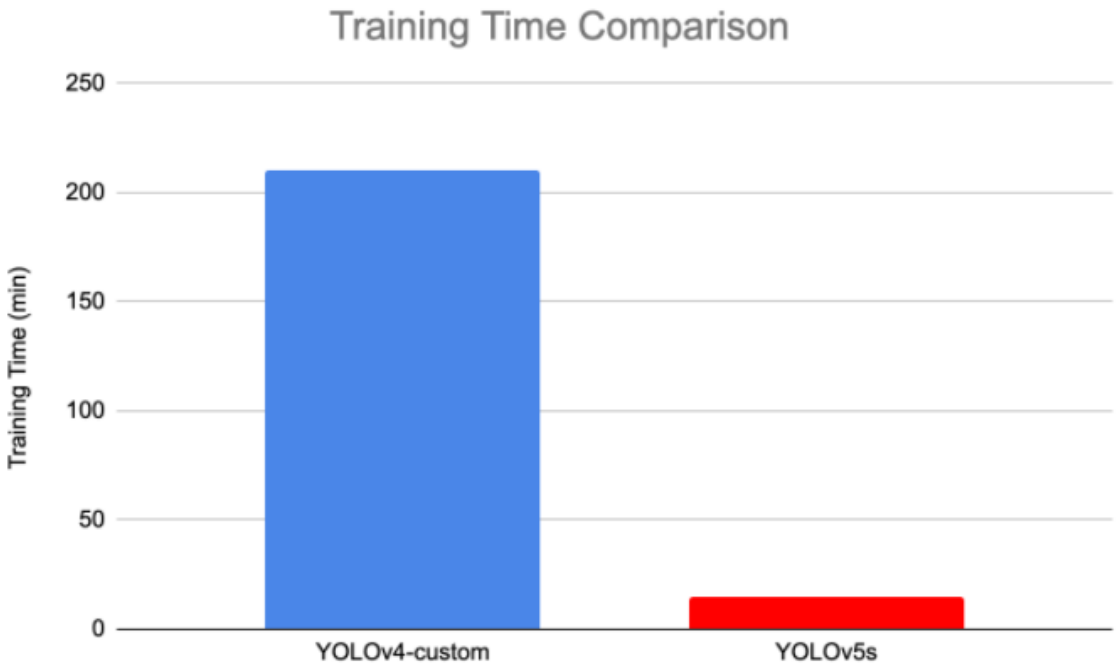
در تصویر نمونه ای از تشخیص object را مشاهده میکنیم که تفاوت YOLOv4 و YOLOv3 کاملا مبین و مشخص شود



شکل 1-1-1 عملکرد دو ورژن YOLO روی داده MS COCO

همانگونه که قابل مشاهده است، YOLOv5 با حدود 65FPS هم دقت بالاتری (43درصد) نسبت به ورژن قبلی خود داشته است که این عدد 33درصد است.

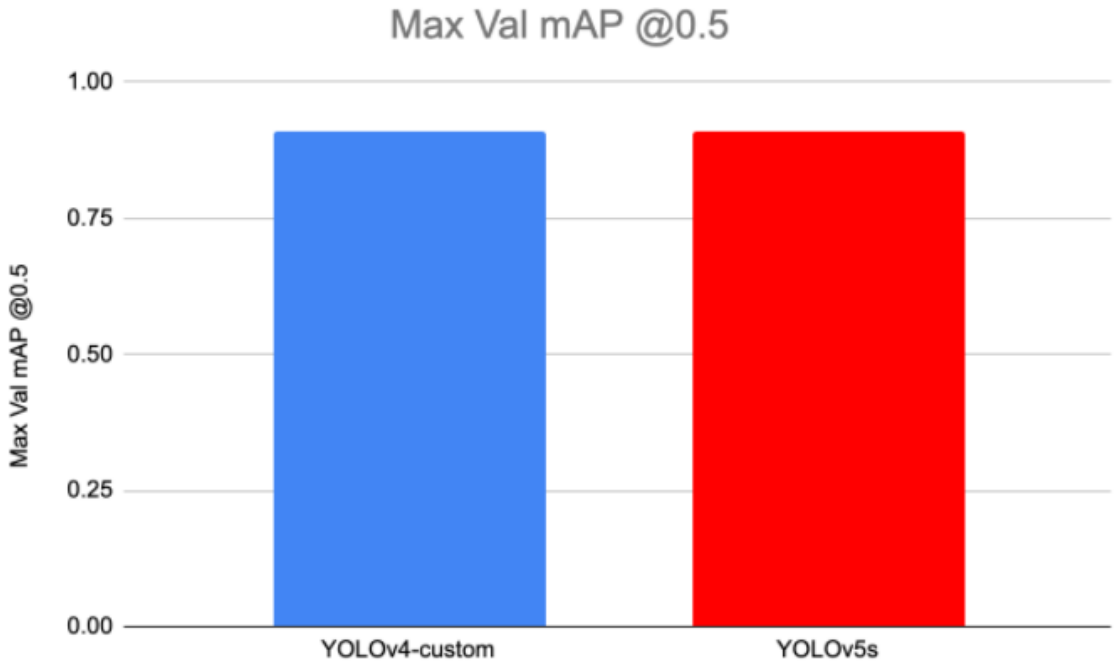
YOLOv5 در سوی دیگر، مزیت های YOLOv4 را دارد و بسیار آموزش ها سریعتر نیز عمل میکند، به طور مثال در نمونه ای از آموزش نتایج به صورت زیر شده است:



شکل 1-1-2 مقایسه زمانی YOLOv5 و YOLOv4

مشاهده میشود که YOLOv5 چقدر سریعتر بوده است.

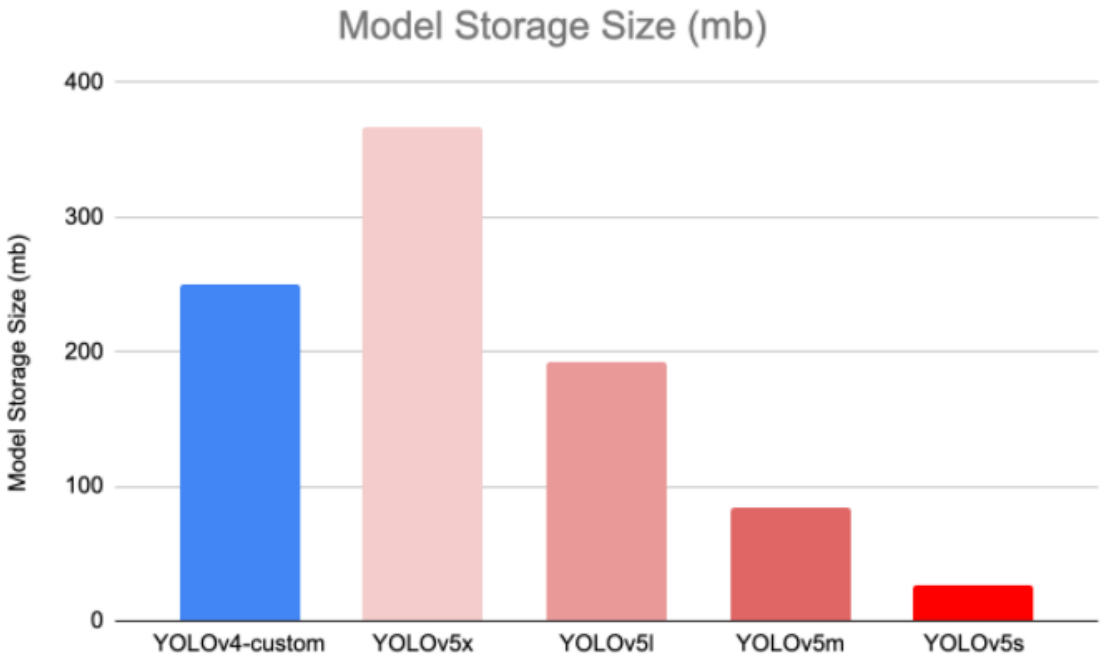
در همین آزمایش، دقت این دو ورژن هم سنجیده شده است، که نتایج به صورت زیر بوده است:



شکل 1-1-3 مقایسه دقت YOLOv5 و YOLOv4

همانطور که قابل مشاهده میباشد، به لحاظ دقت تفاوت خیلی زیادی نداشته و هر دو ورژن دقت خوبی را از خود نشان میدهند.

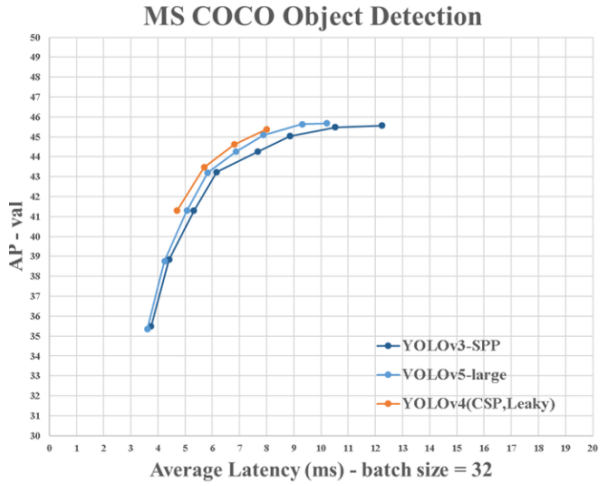
میتوان از لحاظ حجم اطلاعات ذخیره شونده نیز این دو ورژن را مقایسه نمود،



شکل 1-1-4 مقایسه حافظه YOLOv5 و YOLOv4

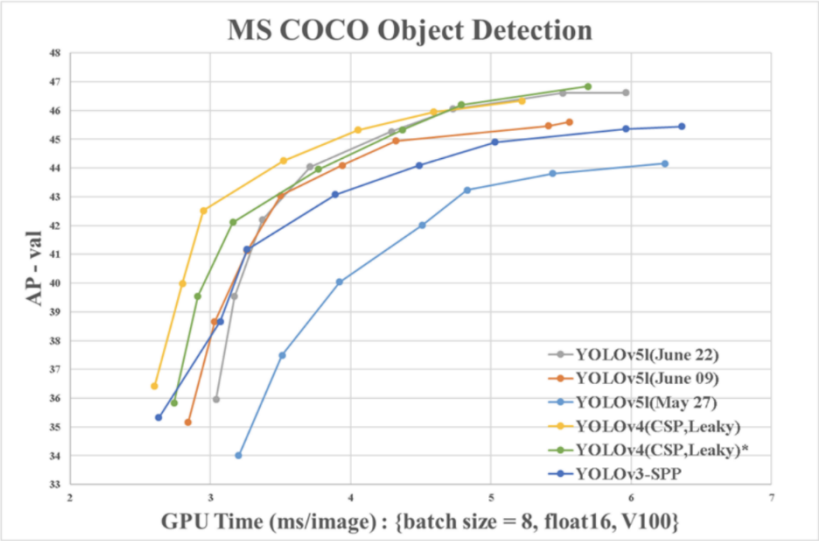
مشاهده میشود که حافظه مورد نیاز برای ذخیره سازی وزن نیز برای YOLOv4 بسیار بیشتر از YOLOv5 بوده است (خود بوچوفسکی نیز به این موضوع اقرار داشته است که برای آنکه اطلاعات با دقت بسیار بالاتری ذخیره شوند، حجم داده بسیار بالا میرود).

همچنین گاهی نیز دقت با YOLOv4 بهتر نیز بدست می آید به عنوان مثال، در دیتاست MS COCO شاهد این قضیه هستیم:



شکل 1-1-5 مقایسه دقت YOLOv5 و YOLOv4 و YOLOv3 برای دیتاست MS COCO

میتوان به صورت کلی نیز عملکرد ورژن های مختلف از 3 و 4 و5 YOLO را روی دیتاست MS COCO مشاهده نمود.



شکل 1-1-6 مقایسه دقت ورژن های مختلف YOLOv5 و YOLOv4 و YOLOv3 برای دیتاست MS COCO

به طور کلی برای سرعت بیشتر و دقت قابل قبول و در نظر گرفتم عامل کاربری آسان تر، از YOLOv5 استفاده میشود و برای task های با دقت ملزوم بالا، از YOLOv4 در darknet استفاده میشود.

2) توضیح الگوریتم نتایج آموزش در Epoch آخر و نمودار های mAP0.5 و mAP0.5:95 و Precision و Recall

1- Install Dependencies and import libraries:

در این مرحله میبایست مخزن (Repository) YOLOv5 Github را clone کنیم و بعد از آن پوشه yolov5 به قسمت Google colab Files اضافه میشود، همچنین برای پاک کردن messy index files و working tree از git reset --hard استفاده میشود.

سپس یکسری Dependency را به علت لزوم نصب میکنیم و کتابانه torch و تعدادی Method را به محیط کار import میکنیم. به علت آنکه Dataset به ما داده شده است نیازی به import کردن gdrive_download نمیباشد.

سپس نسخه Pytorch مورد استفاده در پروژه و اطلاعات GPU داده شده توسط google colab را نمایش میدهیم.

```
print('Setup complete. Using torch %s %s' % (torch.__version__, torch.cuda.get_device_properties(0) if torch.cuda.is_available() else 'CPU'))
```

```
Setup complete. Using torch 1.8.1+cu101 _CudaDeviceProperties(name='Tesla T4', major=7, minor=5, total_memory=15109MB, multi_processor_count=40)
```

```
Setup complete. Using torch 1.8.1+cu101
```

```
_CudaDeviceProperties(name='Tesla T4', major=7, minor=5, total_memory=15109MB, multi_processor_count=40)
```

شکل 1-2-1 نسخه Pytorch و مدل GPU استفاده شده

2- Load and unzip data:

به علت آنکه Dataset به ما داده شده بود، نیاز به دانلود کردن آن نیستیم، پس کافیست که فایل roboflow.zip را در کنار فایل yolov5 قرار داده و آن را طبق دستور زیرآن را unzip میکنیم که نتیجه حاصل، 4 فایل است:

1- فایل train که حاوی فایل تصاویر و label های آن ها است
2- فایل valid که حاوی فایل تصاویر و label های آن ها است
3- فایل data.yaml
4- فایل txt. که حاوی اطلاعات مخصوص preprocess هایی است که روی این dataset انجام شده است.

ما با استفاده از train.py با محتویات فایل train آموزش خواهیم داد و سپس از طریق detect.py محتویات فایل valid را به آزمون میگذاریم و نتایج را روی آن تست میکنیم.

3- Model configuration and architecture:

در این بخش با استفاده از فایل data.yaml تعداد کلاس ها و label مخصوص آن ها را در میبابیم.

برای داده های ما:

```
train: ../train/images
val: ../valid/images

nc: 6
names: ['blue', 'green', 'red', 'vline', 'white', 'yellow']
```

شکل 1-2-2 اطلاعات کلاس ها برای آموزش و تست دیتا ست سوال اول

همانطور که مشاهده میشود (و همانطور که در ابتدای سوال هم اشاره شد) 6 کلاس داریم که به ترتیب آبی، سبز، قرمز، خط، سفید و زرد هستند.

سپس مدل را Configurate میکنیم وipython writefile را costomize میکنیم تا بتوانیم متغیر نگاری کنیم.

4- Train:

برای تمرین داده، از دستور مربوطه استفاده میکنیم. برای این دستور میتوان پارامتر هایی را لحاظ نمود که به تشریح در زیر آورده شده اند:

1- img که در واقع سایز تصویر را میگیرد که در پروژه 420 گذارده شد
2- Batch که سایز batch را میگیرد که در پروژه 16 گذارده شد
3- Epoch که تعداد ایپاک را میگیرد، در پروژه 150 لحاظ شد
4- Data که مسیر data.yaml را میگیرد (که در بخش قبل، توضیح داده شد برای چه منظور)
5- Cfg که configuration مدل ما را مشخص مینماید
6- Weights که مسیر مشخصه وزن است
7- Name که نام نتیجه مدل است
8- No-save که فقط نقطه نهایی چک-پوینت را ذخیره میکند
9- Chache برای استفاده از تصاویر chache برای یادگیری هر چه سریعتر

```
    Epoch   gpu_mem       box       obj       cls     total   targets  img_size
  149/149     1.81G   0.04784   0.03967  0.007984   0.09549       311       416: 100% 99/99 [00:12<00:00,  7.71it/s]
                Class     Images     Targets          P          R      mAP@.5  mAP@.5:.95: 100% 2/2 [00:01<00:00,  1.53it/s]
                  all         58   1.00e+03      0.956       0.93      0.941      0.594
                 blue         58        115      0.991      0.983      0.995      0.596
                green         58        290      0.996      0.961      0.989      0.592
                  red         58        290      0.993      0.995      0.996      0.647
                vline         58        136      0.957      0.971      0.964      0.884
                white         58         58      0.799       0.69      0.708      0.179
               yellow         58        116          1      0.984      0.996      0.665
Optimizer stripped from runs/train/yolov5s_results/weights/last.pt, 14.8MB
Optimizer stripped from runs/train/yolov5s_results/weights/best.pt, 14.8MB
150 epochs completed in 0.567 hours.

CPU times: user 24.8 s, sys: 2.73 s, total: 27.5 s
Wall time: 34min 35s
```

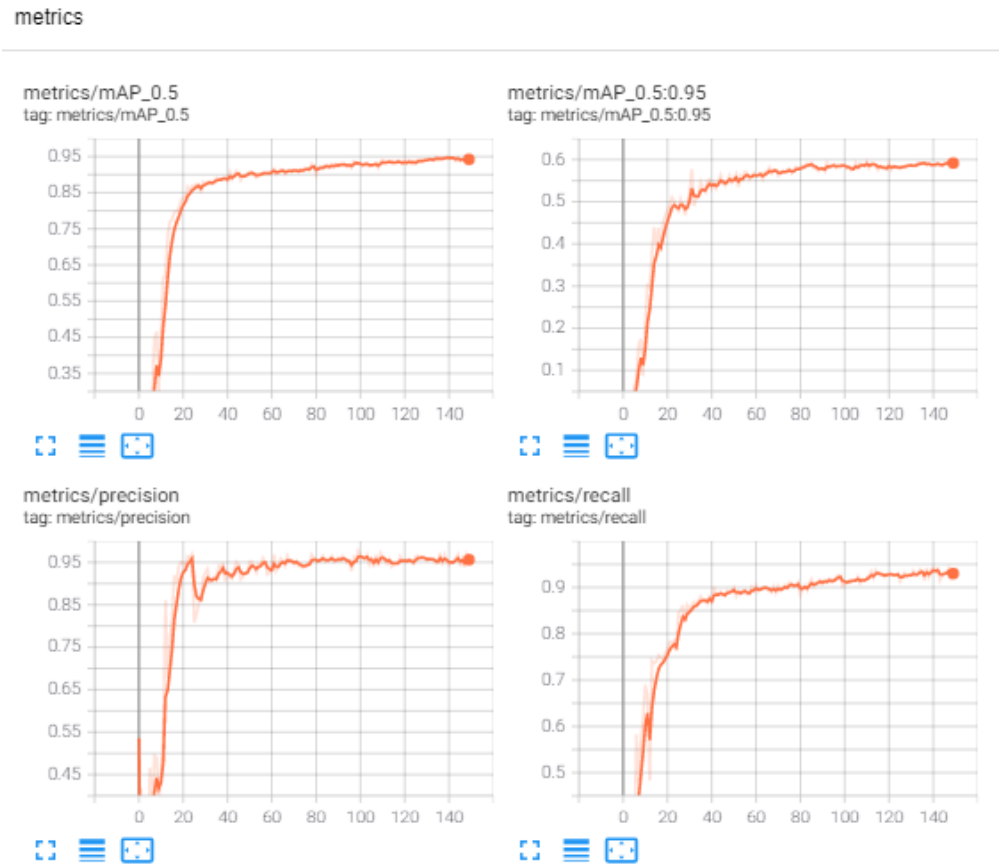
شکل 1-2-3 نتایج بعد از 150 ایپاک

همانگونه که مشاهده میشود، در ایپاک 150 ام نتایج خواسته شده به شرح زیر میباشند:

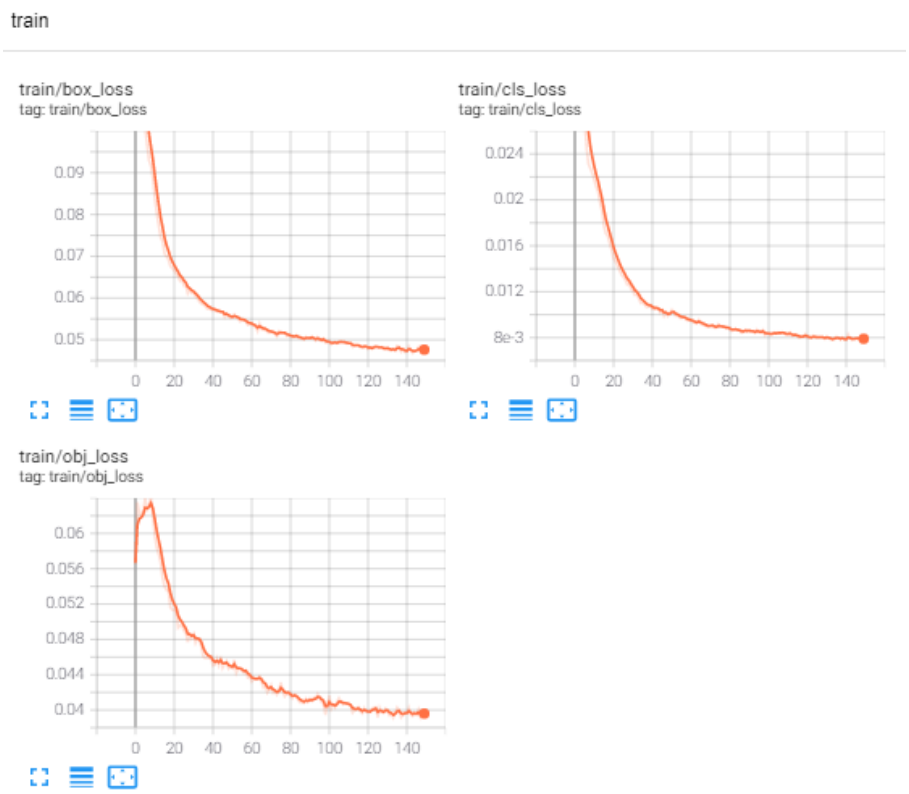
- مقدار Precision برابر با 0.956
- مقدار Recall برابر با 0.93
- مقدار mAP0.5 برابر با 0.941
- مقدار mAP0.5:0.95 برابر با 0.594

حال نمودار ها را ترسیم مینماییم.

خروجی حاصل در زیر را خواهیم داشت:

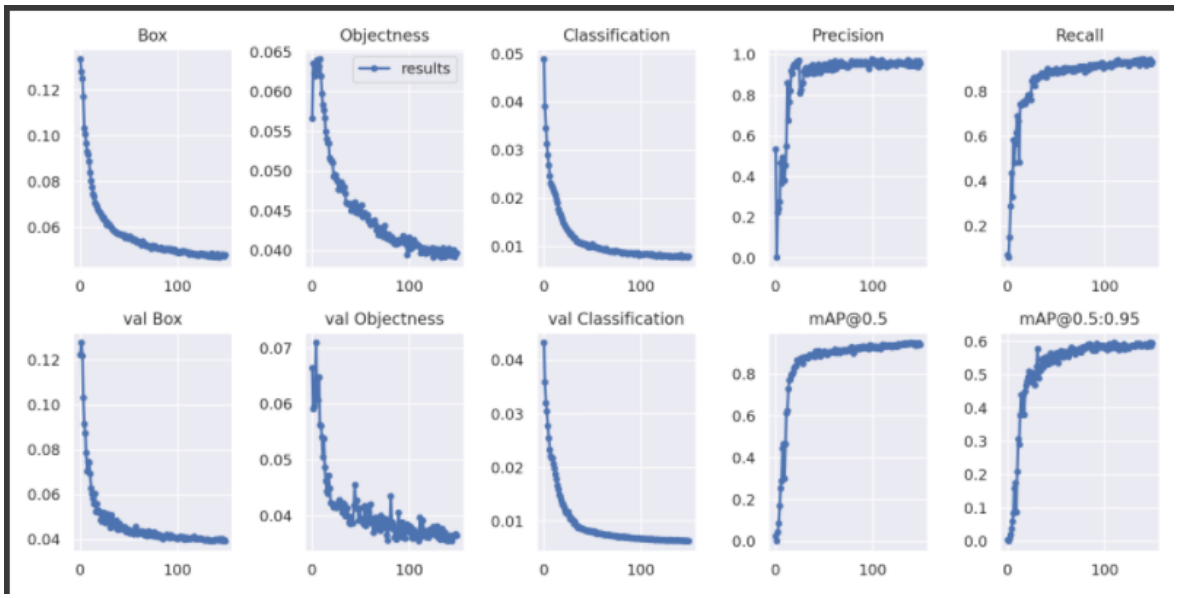


شکل 1-2-4 نتایج نموداری مدل با تمرین در 150 ایپاک



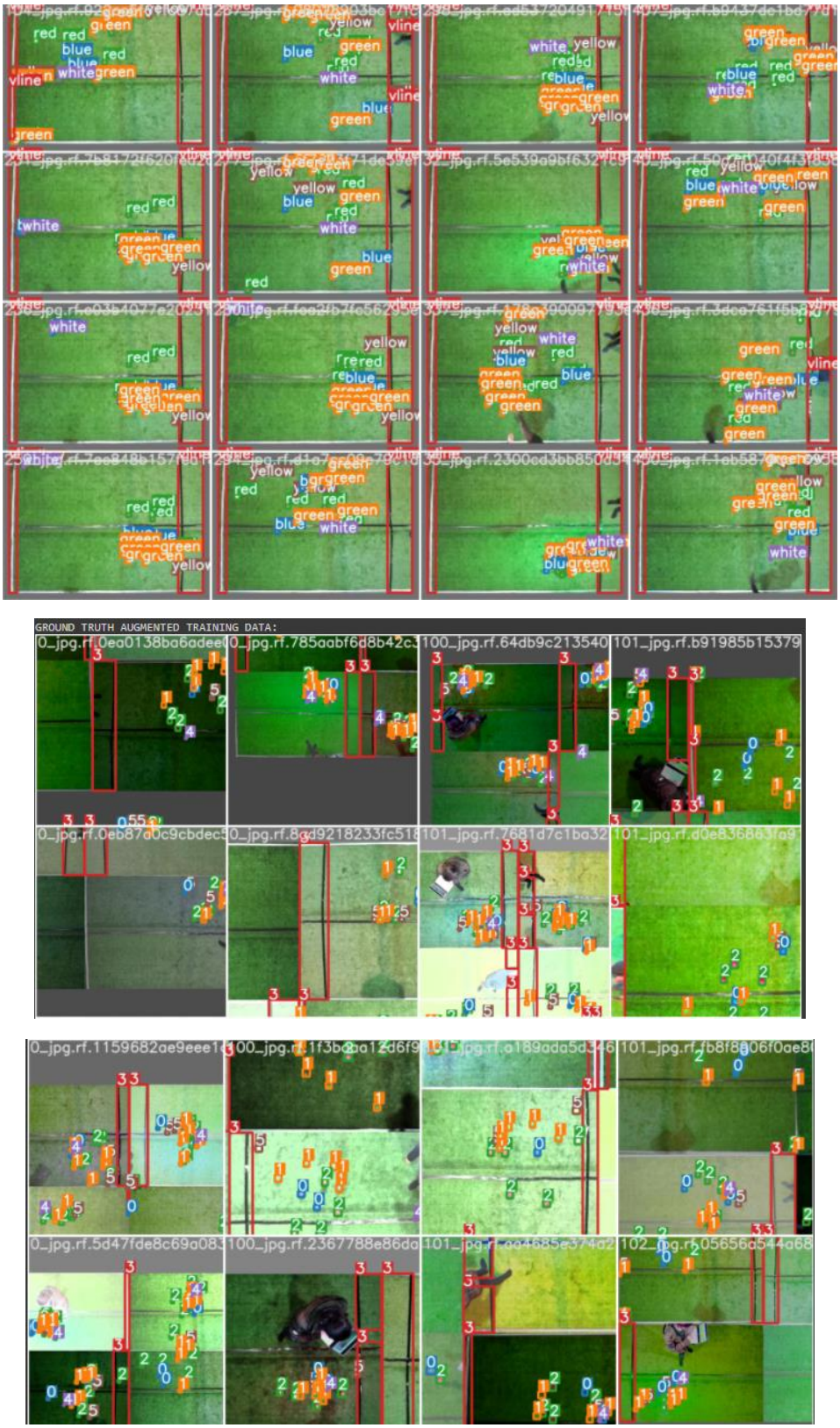
شکل 1-2-5 نتایج نموداری مدل با تمرین در 150 ایپاک

نمونه دیگری از نمایش به شیوه قدیمی را نیز میتوانیم به صورت زیر داشته باشیم:



شکل 1-2-6 نتایج نموداری به سبک قدیمی مدل با تمرین در 150 ایپاک

همچنین میتوان نتایج را برای مدل های تمرین مشاهده نمود، هرچند هم اکنون ناواضح هستند، در بخش بعد، line-thickness از 3 به 1 تغییر پیدا میکند و خوانا تر میشود.

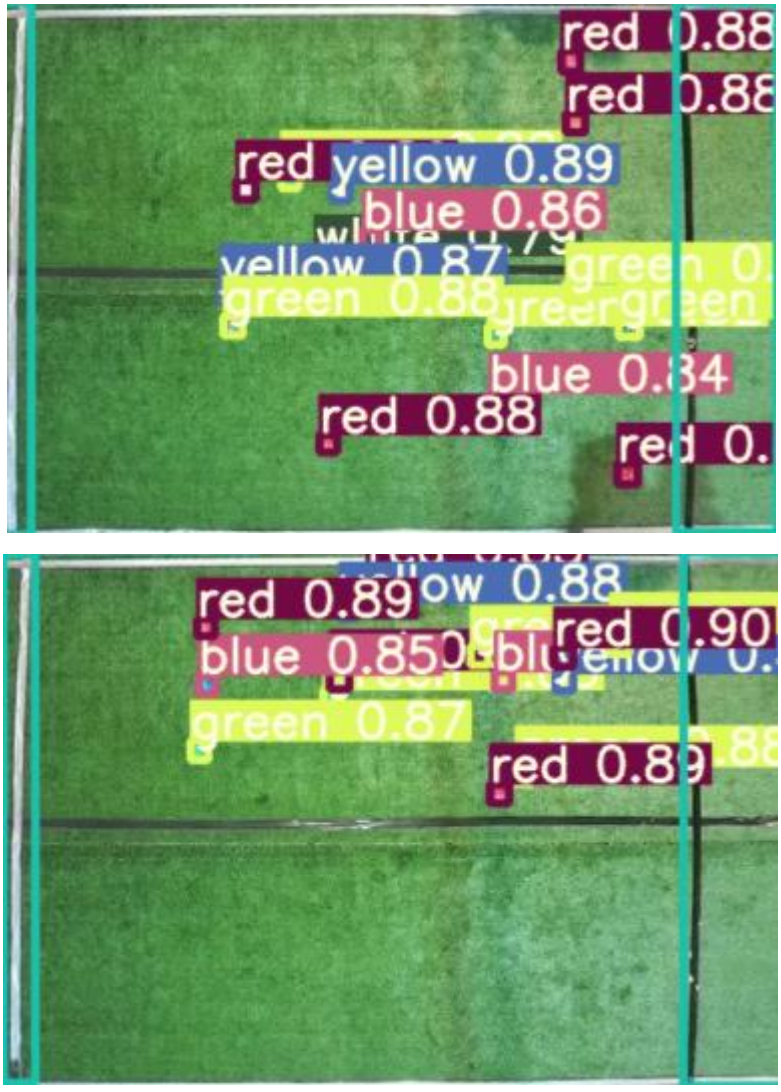


شکل 1-2-7 نتایج تشخیص مدل به داده های تمرین

3) خروجی مدل به ازای ورودی تست(Valid) بعد از آموزش، Enhancement فایل detect.py

بعد از تمرین داده های آموزش، حال وزن بدست آمده است و میتوان با استفاده از قطعه کد مخصوص تست، داده های پوشه valid را ارزیابی کرد و تشخیص object را روی آن بررسی کرد.

بعد از ران کردن detect.py نتایج در فایل exp ذخیره شده اند و قابل نمایش اند، در زیر، نتیجه دو خروجی آورده شده است.



شکل 1-3-1 نتایج تشخیص مدل به داده های ارزیابی

حال از ما خواسته شده است که این تشخیص را خوانا تر کنیم، برای اینکار همانطور که مشاهده میشود، با استفاده از کد زیر:

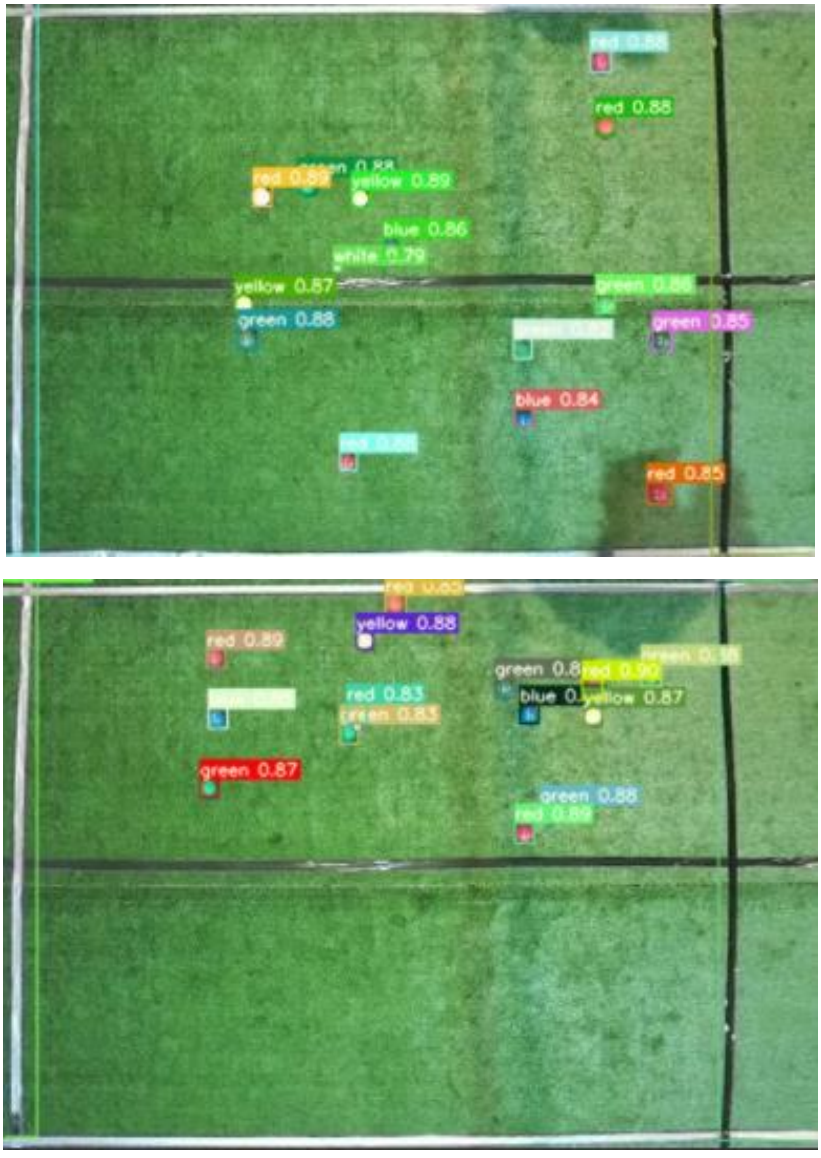
```
parser.add_argument('--line-thickness', default=1, type=int, help='bounding box thickness (pixels)')
```

مقدار Thickness لاین ها را کاهش میدهیم (از 3 به 1 میرسانیم) و در قسمت Save image به صورت مقابل:

```
plot_one_box(xyxy, im0, label=label, line_thickness=opt.line_thickness)
```

اعمال میکنیم.

همان دو نتیجه تصویر بالا، اکنون کاملا واضح شده اند:



شکل 1-3-2 نتایج تشخیص مدل به داده های ارزیابی

4) رتبه بندی فواصل Object ها از توپ سفید

در این بخش سعی میشود که با استفاده از مرکز توپ ها (که خود بدان معناست که با خروجی هایی که label خط حورده اند کاری نخواهیم داشت) ، فاصله آنها را با توپ سفید بسنجیم و به آن ها رنک بدهیم.

برای اینکار از تابع detect.py استفاده میکنیم، این تابع دو متغیر اساسی در این بخش برای ما دارد.

1- متغیر xyxy اطلاعات پوزیشن x و y بالا سمت چپ قاب تشخیص داده شده و پوزیشن x و y پایین سمت راست قاب تشخیص داده را به ما میدهد هر کدام از درایه های xyxy[i] ها نیز از جنس .tensor obj هستند.
2- منغیر cls که کلاس مربوطه به object را برای ما فراهم میسازد و خود از جنس tensor obj است.

با توجه به تصویر زیر:

```
train: ../train/images
val: ../valid/images

nc: 6
names: ['blue', 'green', 'red', 'vline', 'white', 'yellow']
```

شکل 1-4-1 طبقه بندی کلاس ها و ترتیب آنها

متوجه هستیم که ترتیب کلاس ها به صورت زیر خواهد بود:

1- توپ آبی: کلاس 0
2- توپ سبز: کلاس 1
3- توپ قرمز: کلاس 2
4- خط: کلاس 3
5- توپ سفید: کلاس 4
6- توپ زرد: کلاس 5

حال کافیست طبق صحبت انجام شده، عمل کرده و با استفاده از متغیر xyxy مرکز توپ ها را بیابیم (حاصل جمع xy ها تقسیم بر دو) و سپس با توجه به کلاس آن ها، فاصله را از توپ با کلاس سفید یافته و آن ها را رنک بندی کنیم.

```
# Write results
for *xyxy, conf, cls in reversed(det):
  if save_txt:  # Write to file
      xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist()  # normalized xywh
      line = (cls, *xywh, conf) if opt.save_conf else (cls, *xywh)  # label format
      with open(txt_path + '.txt', 'a') as f:
          f.write(('%g ' * len(line)).rstrip() % line + '\n')

  if save_img or view_img:  # Add bbox to image
      label = f'{names[int(cls)]} {conf:.2f}'
      plot_one_box(xyxy, im0, label=label, color=colors[int(cls)], line_thickness=3)
  print('------------------')

  print("Center xyxy: ", [(xyxy[0].item()+xyxy[2].item())/2 , (xyxy[1].item()+xyxy[3].item()/2)])
  print("type cls", cls.item())
```

```
print("Center xyxy: ", [(xyxy[0].item()+xyxy[2].item())/2 , (xyxy[1].item()+xyxy[3].item()/2)])
print("type cls", cls.item())
```

```
Center xyxy:  [11.5, 84.0]
type cls 3.0
------------------
Center xyxy:  [351.0, 18.0]
type cls 5.0
------------------
Center xyxy:  [15.5, 274.0]
type cls 1.0
------------------
Center xyxy:  [9.5, 338.5]
type cls 3.0
------------------
Center xyxy:  [24.0, 487.0]
type cls 2.0
------------------
Center xyxy:  [67.0, 112.0]
type cls 2.0
------------------
Center xyxy:  [209.5, 82.5]
type cls 1.0
------------------
Center xyxy:  [219.0, 259.5]
type cls 1.0
```

شکل 1-4-2 پرینت شماره کلاس شئ و مرکز آن

حال تغییرات را اعمال میکنیم و برای 5 تصویر از کوچک به بزرگ، فاصله ها را مرتب میکنیم و کد را به مانند زیر در می آوریم:

```
            center_pos_other = []
            class_pos_other = []
            for *xyxy, conf, cls in reversed(det):
              if save_txt:  # Write to file
                  xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist()  # normalized xywh
                  line = (cls, *xywh, conf) if opt.save_conf else (cls, *xywh)  # label format
                  with open(txt_path + '.txt', 'a') as f:
                      f.write(('%g ' * len(line)).rstrip() % line + '\n')

              if save_img or view_img:  # Add bbox to image
                  label = f'{names[int(cls)]} {conf:.2f}'
                  plot_one_box(xyxy, im0, label=label, color=colors[int(cls)], line_thickness=3)
              if(cls.item() == 4):
                center_pos_white.insert(0,[(xyxy[0].item()+xyxy[2].item())/2 , (xyxy[1].item()+xyxy[3].item()/2)])
              else:
                if(cls.item() != 3): #NOT Line
                  center_pos_other.append([(xyxy[0].item()+xyxy[2].item())/2 , (xyxy[1].item()+xyxy[3].item()/2)])
                  class_pos_other.append(cls.item())
            ordered_distance_from_white = []
            class_obj = []
            for i in range(len(center_pos_other)):
              distance = abs(center_pos_white[0][0]-center_pos_other[i][0]) + abs(center_pos_white[0][1]-center_pos_other[i][1])
              if(len(ordered_distance_from_white) == 0):
                ordered_distance_from_white.append(distance)
                class_obj.append(class_pos_other[i])
              else:
                for j in range(len(ordered_distance_from_white)):
                  if(distance<ordered_distance_from_white[j]):
                    ordered_distance_from_white.insert(j,distance)
                    class_obj.insert(j,class_pos_other[i])
                    break
                  if(j == len(ordered_distance_from_white)-1):
                    ordered_distance_from_white.insert(j,distance)
                    class_obj.insert(j,class_pos_other[i])
                    break
            print("ORDERD DISTANCE FROM WHITE:")
            print('ordered_distance_from_white: ', ordered_distance_from_white)
            print('class_obj                  : ', class_obj)
```

شکل 1-4-3 تغییرات اعمالی در فایل detect.py و ثبت در detect2.py

که بدین صورت دیگر میتوان ordered_distance_from_white و class_obj را به ترتیب نمایش داد:

برای 5 تصویر این اعداد به مانند زیر خواهند بود:

```
image 1/58 /content/yolov5/../valid/images/104_jpg.rf.922cc5f61fc87ab619f35377de76820c.jpg: ORDERD DISTANCE FROM WHITE:
ordered_distance_from_white:  [179.0, 258.0, 274.5, 289.5, 292.0, 312.5, 338.0, 378.5, 432.5, 478.5, 511.0, 369.0]
class_obj                  :  [2.0, 2.0, 1.0, 1.0, 1.0, 0.0, 2.0, 0.0, 2.0, 1.0, 2.0, 5.0]

image 2/58 /content/yolov5/../valid/images/106_jpg.rf.741aa61c5acfb884a2e5d91e5951d70f.jpg: ORDERD DISTANCE FROM WHITE:
ordered_distance_from_white:  [661.5, 662.0, 685.5, 696.5, 700.5, 728.0, 748.5, 757.5, 762.5, 749.5]
class_obj                  :  [1.0, 2.0, 2.0, 1.0, 1.0, 2.0, 2.0, 0.0, 1.0, 5.0]

image 3/58 /content/yolov5/../valid/images/110_jpg.rf.c3404cd59249da5dd4262697cf06778d.jpg: ORDERD DISTANCE FROM WHITE:
ordered_distance_from_white:  [314.5, 342.0, 398.5, 400.5, 427.0, 444.5, 458.0, 473.5, 480.0, 482.0, 550.5, 575.0, 611.0, 526.0]
class_obj                  :  [5.0, 2.0, 2.0, 1.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 0.0, 0.0, 5.0]

image 4/58 /content/yolov5/../valid/images/114_jpg.rf.a67c6b727ec53130ac57c7380c9b7071.jpg: ORDERD DISTANCE FROM WHITE:
ordered_distance_from_white:  [314.5, 343.5, 351.5, 427.0, 431.0, 451.5, 475.5, 551.0, 563.0, 603.0, 610.0, 683.0, 305.0]
class_obj                  :  [1.0, 2.0, 5.0, 1.0, 1.0, 2.0, 0.0, 1.0, 2.0, 2.0, 1.0, 2.0, 5.0]

image 5/58 /content/yolov5/../valid/images/117_jpg.rf.13a5d6ae58d59d25813cb7882f9a924c.jpg: ORDERD DISTANCE FROM WHITE:
ordered_distance_from_white:  [615.0, 653.5, 653.5, 705.0, 712.5, 721.5, 723.0, 741.5, 745.5, 749.0, 761.5, 768.5, 602.5]
class_obj                  :  [1.0, 0.0, 1.0, 1.0, 0.0, 2.0, 1.0, 2.0, 5.0, 0.0, 2.0, 2.0, 2.0]
```

شکل 1-4-4 نتایج خروجی شبکه عصبی با detect2.py

سوال 2 – Semantic segmentation

در این سوال مقصود آن است که شبکه U-Net که شبکه ای Full convolutional است را پیاده سازی کنیم و از آن برای Semantic segmentation برای Data set مشخص(Cam Vid) استفاده کنیم.

در شبکه های با کاربرد Semantic Segmentation سعی میشود با استفاده از داده، object ها detect شوند و برای هر پیکسل، یک کلاس معرفی شود.

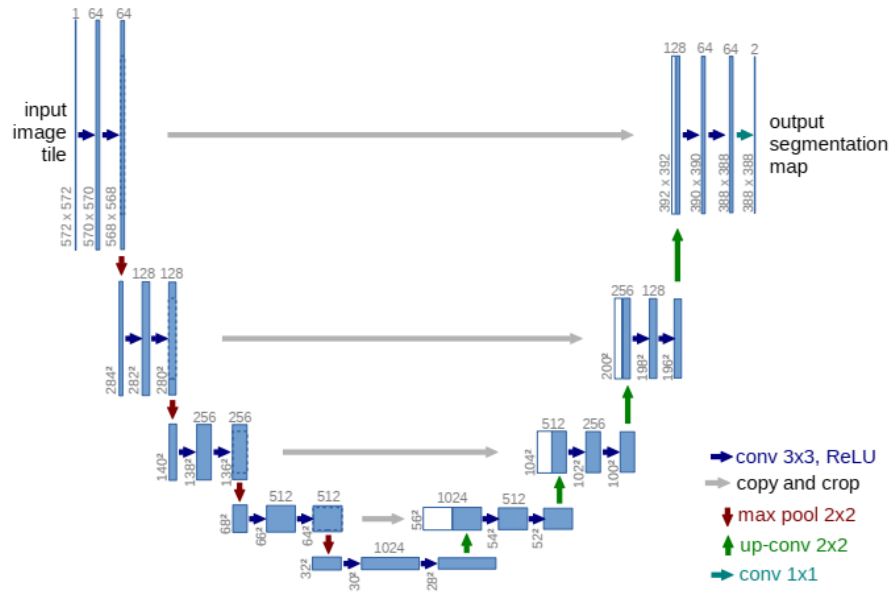
این شبکه برای Semantic Segmentation روی داده های پزشکی استفاده میشود و از جمله شبکه های معروف در زمینه مهندسی پزشکی میباشد. این شبکه تماما کانوولوشنی، با استفاده از لایه های خود، که در مقاله هم ذکر شده در مجموع 23 لایه کانوولوشنی دارد، سپس با Up sampling ویژگی های مورد نظر را استخراج میکند و آن ها را کنار هم میگذارد و با Feature map های بدست آمده در هر مرحله اطلاعات محلی و Context را ترکیب مینماید.

در این پیاده سازی با توجه تذکر داده شده، برای قسمت Up sampling:

- از الگوریتم Nearest neighbors استفاده میکنیم
- برای تابع خطا از Cross entropy استفاده میکنیم

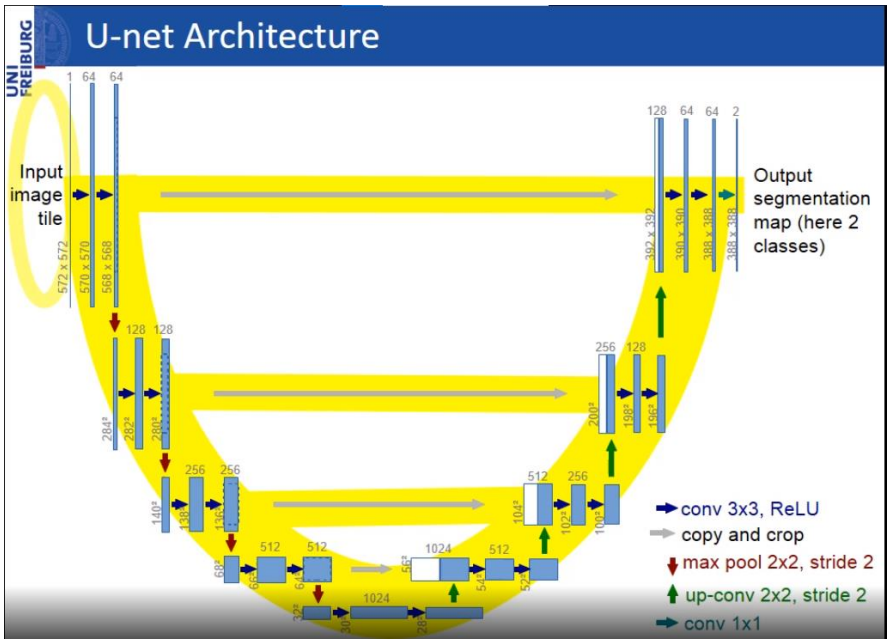
1) پیاده سازی شبکه U-Net و نمودار خطای داده های آموزش و تست

همانگونه که در مقاله نیز آورده شده است، شکل کلی این مدل مانند زیر خواهد بود:



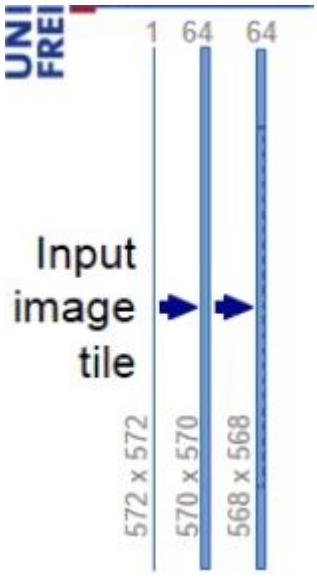
شکل 2-1-1 ساختار کلی U-Net

همانطور که واضح است، این شبکه، از چپ به راست پیش میرود و طبق شکل زیر از مسیر های زرد رنگ، path دارد و اطلاعات از input به output میرسد:



شکل 2-1-2 ساختار path کلی U-Net

در مسیر زیر:

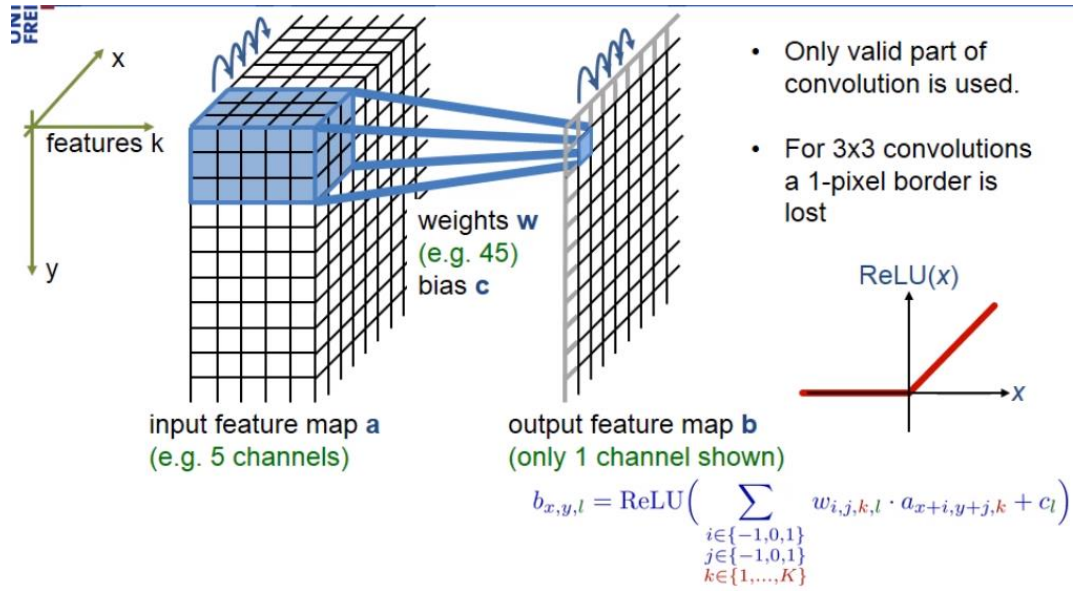


شکل 2-1-3 ساختار path مرحله اول U-Net

مشاهده میشود سایز xy 572*572 میباشد و تعداد feature channels برابر 1 است و بعد از یک conv3*3 و عبور از تابع فعال ساز ReLU به سایز xy 570*570 میباشد و تعداد feature channels برابر 64 میبریم و با این مشخصات conv3*3 میزنیم و همین قضیه ادامه خواهد داشت، طبق شکل 2-1-2 ، ما به عناصر زیر برای پردازش نیازمند خواهیم بود:

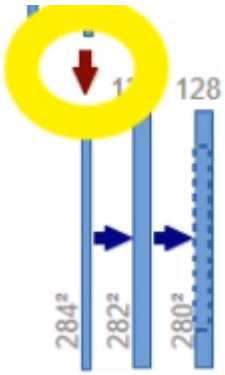
:Conv 3*3, ReLU -1

که عملیات زیر صورت خواهد گرفت:



شکل 2-1-4 ساختار conv 3*3

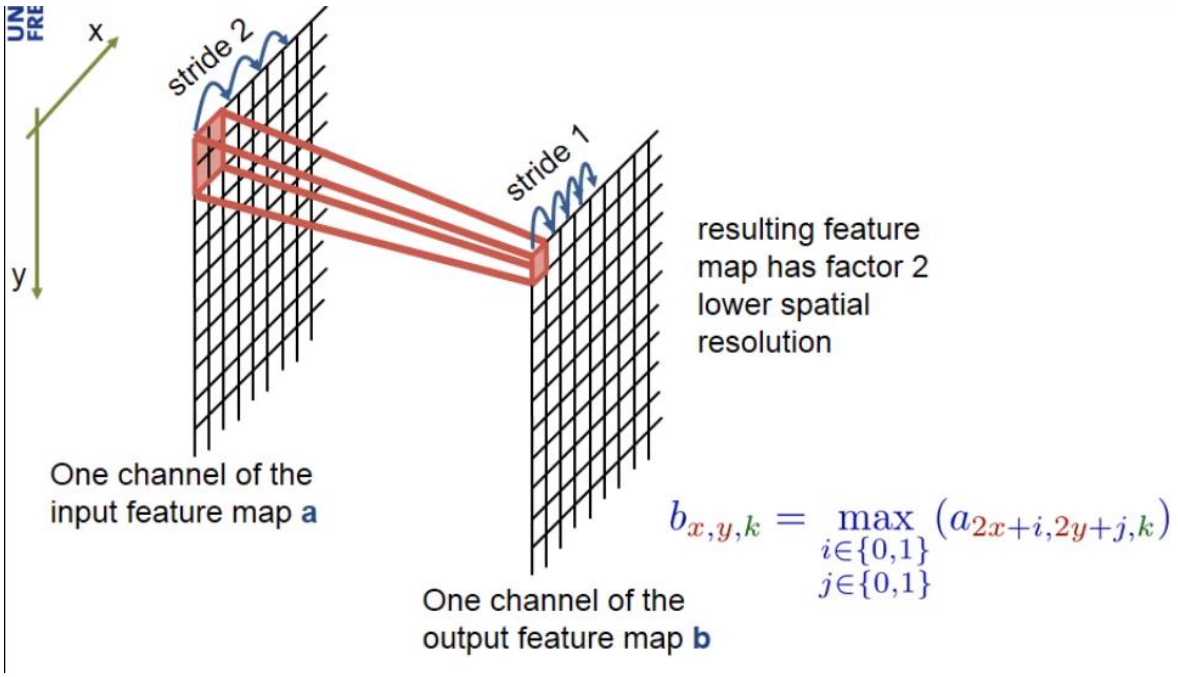
Max pooling, stride 2 -2



شکل 2-1-5 مرحله maxpooling

به علت حجم اطلاعات بالا، نیاز مند آن هستیم که کمی دیتا را کوچکتر کنیم که پارامتر های ما بسیار زیاد نشوند، برای این امر از Max pooling استفاده میشود و stride را برابر 2 قرار میدهند که پرش داشته باشد.

ساختار آن به مانند زیر خواهد بود:

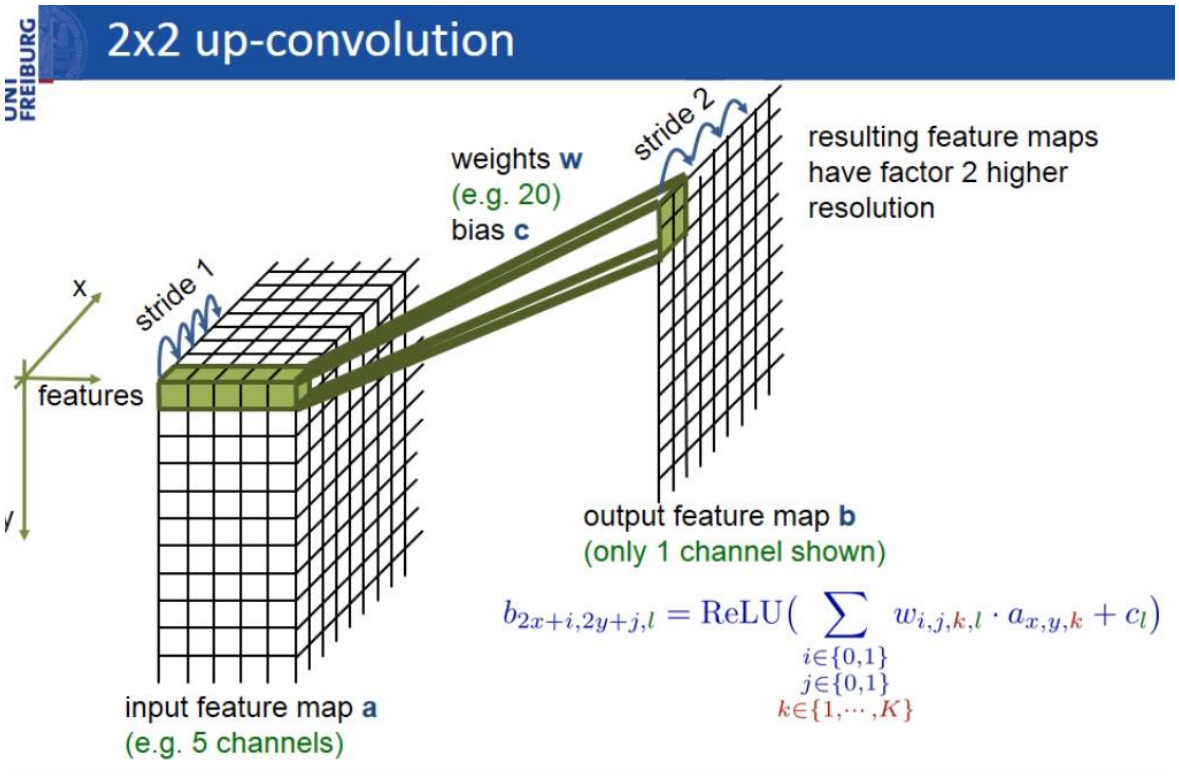


شکل 2-1-6 ساختار maxpooling

Up conv, stride 2 -3

در طی مسیری که دنبال میشود، ما به تعداد filter ها اضافه میکنیم، دو برابر میکنیم از 64 به بعد، و بیشتر به پاسخ "چه object " پاسخ میدهیم (مرحله Contraction)، سپس سعی میکنیم به High resolution از segmentation برسیم (مرحله Expansion).

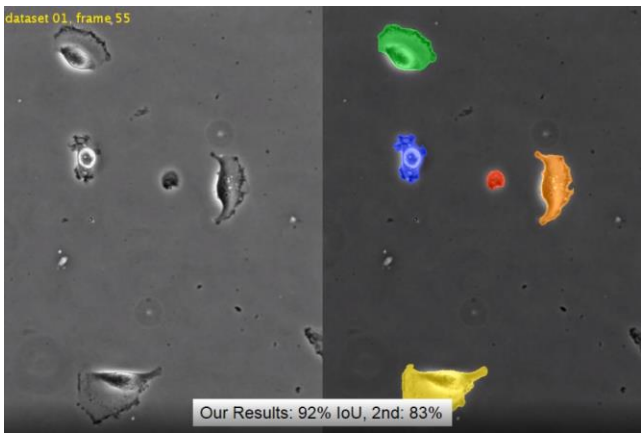
شکل 2-1-7 ساختار Contraction و Expansion



شکل 2-1-8 ساختار Up-conv 2*2

Conv 1*1 -4

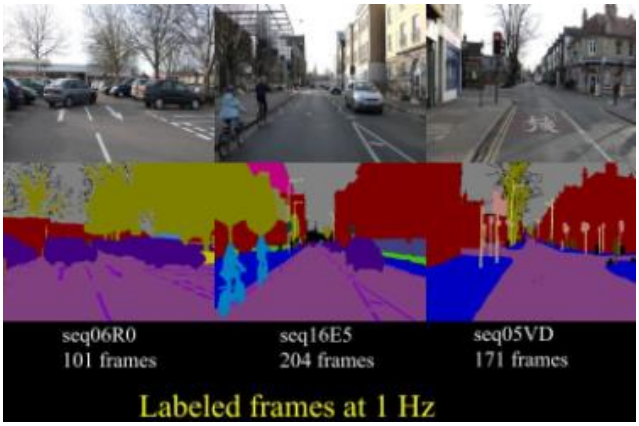
در آخر برای خروجی، conv 1*1 خواهیم زد، نمونه ای از خروجی شبکه آموزش داده شده U-Net در شکل زیر آمده است که توانسته است موجودات را تشخیص دهد و به هر پیکسل از تصویر یک label اختصاص دهد:



شکل 2-1-8 تشخیص شبکه U-NET

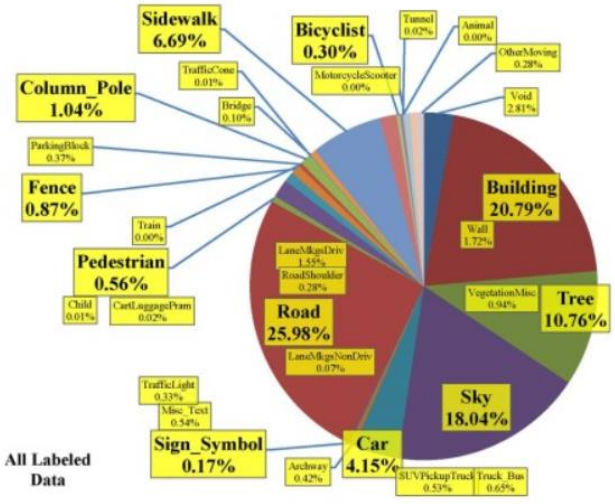
حال سعی میکنیم این شبکه را پیاده سازی کنیم و نمودار های خطا برای داده های آموزش و تست را تبیین کنیم.

برای این شبکه عصبی ابتدا باید دیتا برای آموزش، ارزیابی و تست انتخاب شود، ما برای اینکار از دیتاست CamVid استفاده میکنیم که تصاویری به همراه ماسک آن ها برای ما فراهم میکند که انگار از دید راننده اتوموبیل گرفته شده است و محیط اطراف را میبیند، در تصویر زیر نمونه هایی از این تصاویر به همراه ماسک آن ها را میتوان مشاهده نمود:



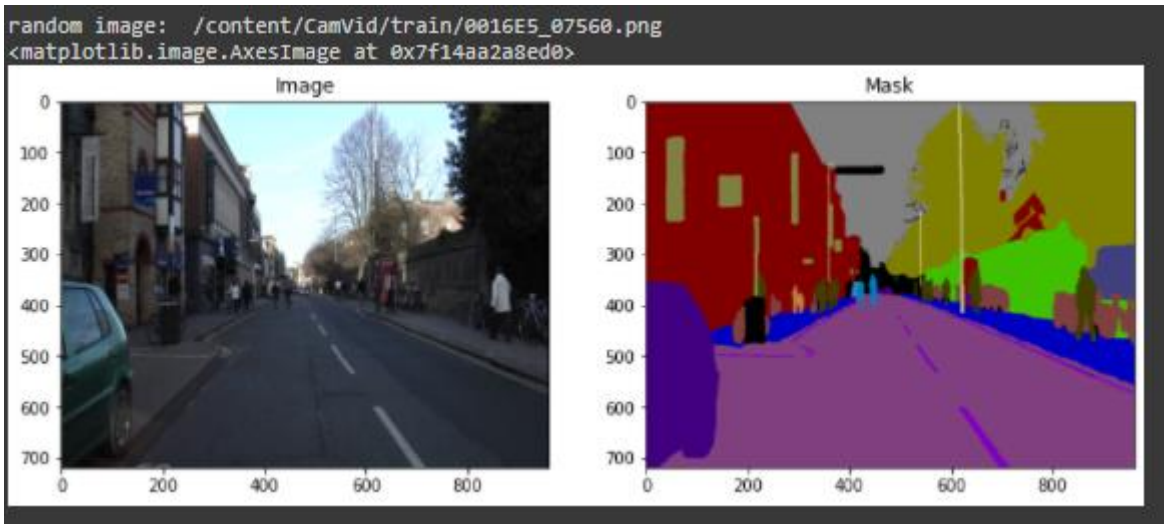
شکل 2-1-9 تصاویری از dataset CamVid

این دیتاست از 32 کلاس تشکیل شده است که در تصویر زیر نام این کلاس ها و توزیع آن ها آورده شده است:



شکل 2-1-10 اسامی و توزیع کلاس های دیتاست camvid

این دیتاست را دانلود کرده Tپس از مراحل Unzip و جفت کردن image و label آن ها برای داده های تست و ارزیابی و تست، یکی از تصاویر به همراه را برای نمایش به صورت رندوم انتخاب میکنیم:



شکل 2-1-11 تصویر یکی از داده های train و mask آن از دیتاست CamVid

بعد از آماده سازی داده ها، آن را با شبکه U-Net در 35 ایپاک تمرین میدهیم.

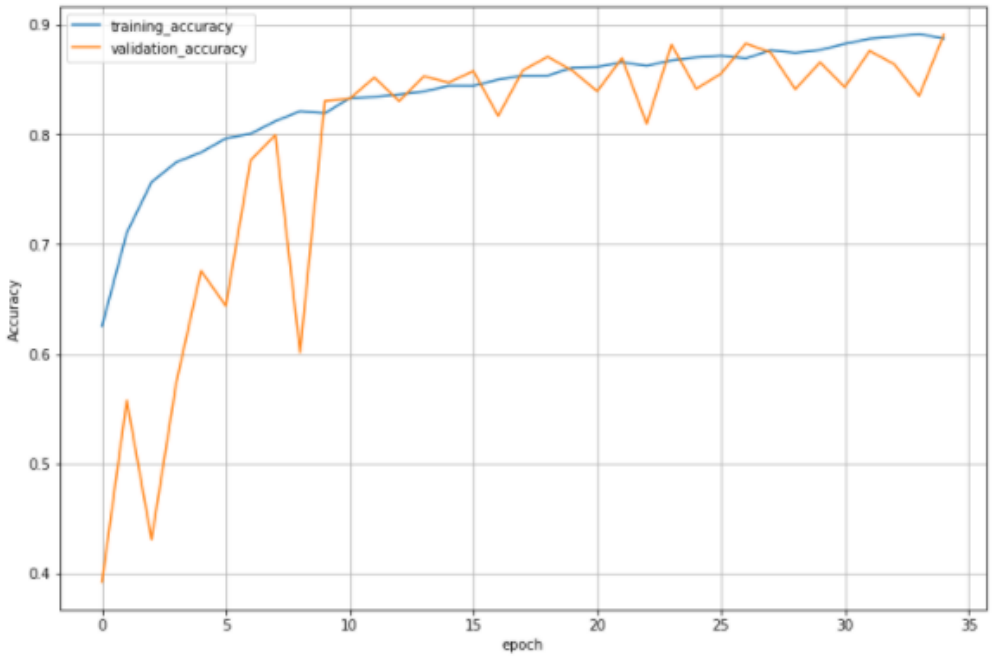
```
[49] model.evaluate_generator(test)

     [0.7678157687187195, 0.7847074270248413]
```

شکل 2-1-12 نتیجه دقت برای داده تست شبکه U-Net

نمودارهای دقت و خطا را میتوان در شکل های زیر بررسی نمود:

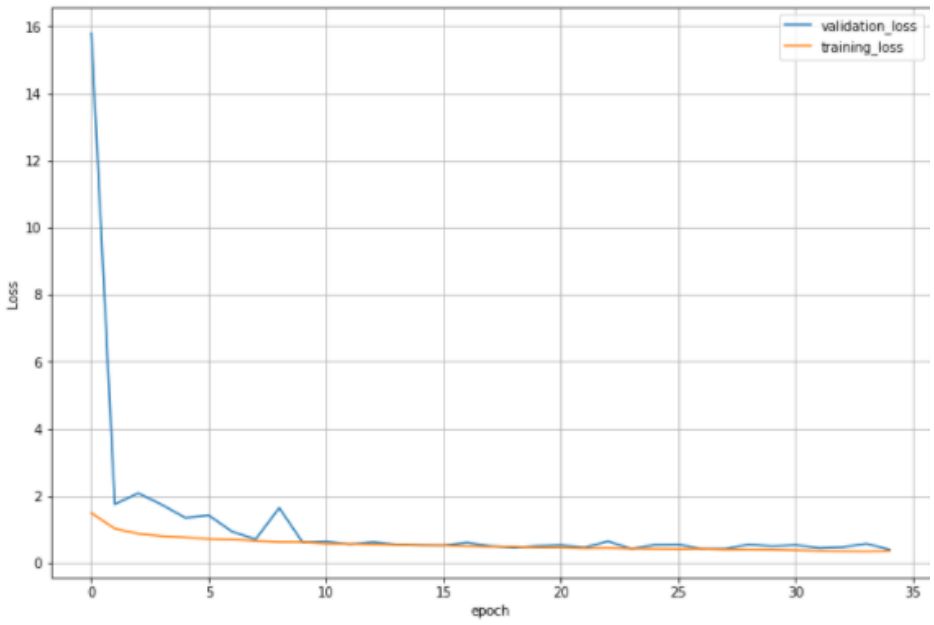
1- دقت:



شکل 2-1-13 نتیجه دقت برای داده تمرین وآزمون شبکه U-Net

مشاهده میشود که دقت در 35 ایپاک افزایش یافته است برای داده های آزمون(هرچند که با افت و خیز همراه بوده است)

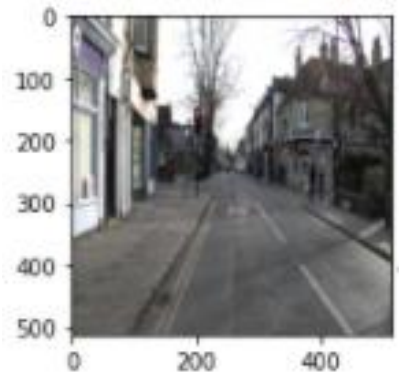
2- Loss:



شکل 2-1-14 نتیجه Loss برای داده تمرین وآزمون شبکه U-Net

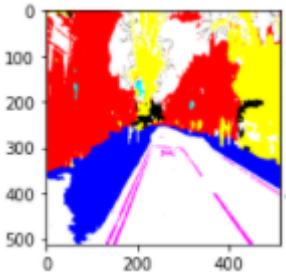
2) خروجی شبکه به ازای ورودی تست

به ازای ورودی زیر به شبکه آموزش دیده شده:



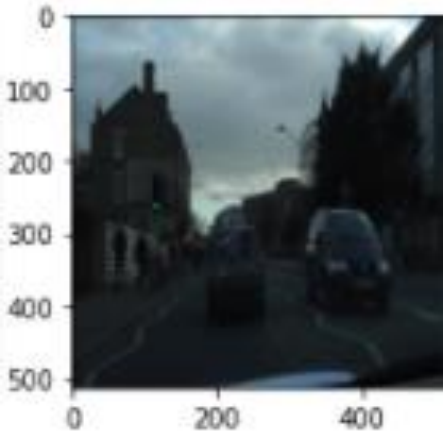
شکل 2-2-1 تصویر یکی از داده های test از دیتاست CamVid

خروجی تولید شده توسط شبکه عصبی، به صورت زیر خواهد بود:



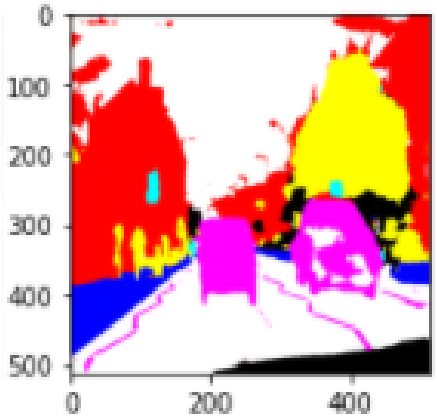
شکل 2-2-2 خروجی شبکه عصبی به ازای داده شکل 2-2-1

به ازای ورودی زیر به شبکه آموزش دیده شده:



شکل 2-2-3 تصویر یکی از داده های test از دیتاست CamVid

خروجی تولید شده توسط شبکه عصبی، به صورت زیر خواهد بود:



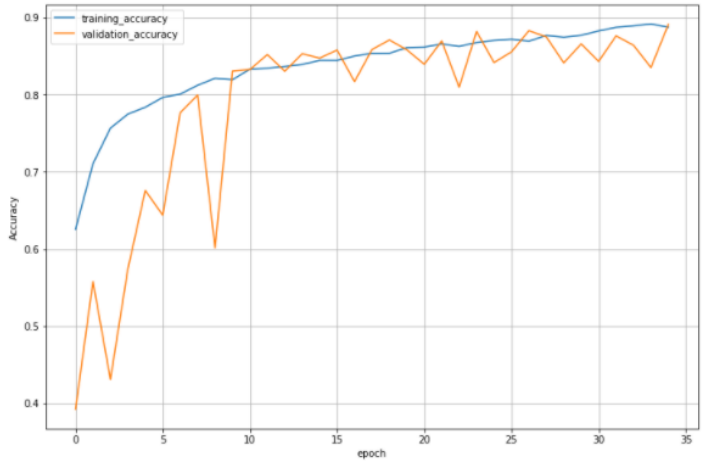
شکل 2-2-4 خروجی شبکه عصبی به ازای داده شکل 2-2-3

3) تعداد ایپاک مناسب برای آموزش شبکه، معیار جداسازی داده های آموزش و تست

- تعداد ایپاک:

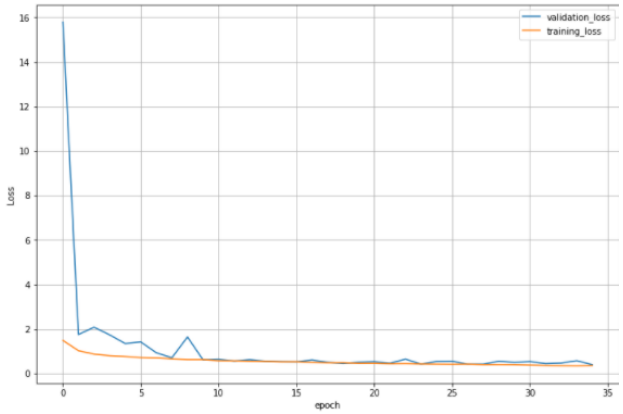
نگاهی دوباره به نمودار های دقت و Loss می اندازیم:

1- دقت:



شکل 2-3-1 نتیجه دقت برای داده تمرین وآزمون شبکه U-Net

2- Loss:



شکل 2-3-2 نتیجه Loss برای داده تمرین وآزمون شبکه U-Net

همانطور که مشخص است، بعد از ایپاک دهم، مقدار Loss تغییر شایانی نکرده است،همچنین دقت نیز با اینکه بعد از ایپاک دهم افزایش داشته اما این افزایش، خیلی چشمگیر نبوده است. در اینجا میبایست یک مصالحه بین نیاز "زمانی" و "پارامتر دقت و Loss " برقرار کنیم، با افزایش ایپاک ها تا 35 توانستیم، دقت را بهتر کنیم اما اینکار به 1 ساعت و 20 دقیقه زمان نیاز داشت، در حالی که با 10 ایپاک میتوانستیم در حدود 30 دقیقه به جواب برسیم.

همچنین باید دقت داشت که با افزایش بر رویه ایپاک ها ممکن است مدل روی داده های تمرین، Overfit شده و دقت برای داده های ارزیابی/تست نیز کاهش چشمگیر پیدا کند.

- نحوه تقسیم داده های آموزش، ارزیابی و تست:

همانطور که در بخش اول نیز ذکر شد، برای این شبکه عصبی ابتدا باید دیتا برای آموزش، ارزیابی و تست انتخاب شود، ما برای اینکار از دیتاست CamVid استفاده میکنیم که تصاویری به همراه ماسک آن ها برای ما فراهم میکند که انگار از دید راننده اتوموبیل گرفته شده است و محیط اطراف را میبیند.

این داده ها به سه دسته "آموزش"، "ارزیابی" و "تست" تقسیم شده اند که این میزان تقسیم بندی در تصویر زیر ذکر شده است:

```
train_images = os.listdir('/content/CamVid/train')
valid_images = os.listdir('/content/CamVid/val')
test_images = os.listdir('/content/CamVid/test')
print(f"Size of train images {len(train_images)}\nSize

Size of train images 367
Size of validation images 101
Size of test images 233
```

شکل 2-3-3 تعداد داده برای تمرین و ارزیابی و آزمون

همانطور که دیده میشود 66.7% از داده ها برای تمرین و ارزیابی و 33.3% داده ها برای تست قرار گرفته اند. که درصد هر کدام را هم میتوان داشت به این صورت که داده های آموزش نسبت $\frac{367}{701} = 52.3$ را داشته است و داده های ارزیابی نسبت $\frac{101}{701} = 14.1\%$ و داده های تست نسبت $\frac{233}{701} = 33.3$ را به خود اختصاص داده اند.

نکته مهم در یادگیری به صورت **رندوم عمل کردن و بکارگیری رندوم داده ها در هر ایپاک است**، این به ما کمک میکند که بااحتمال بیشتری به دقت بهتری برای ایپاک بعدی برسیم.

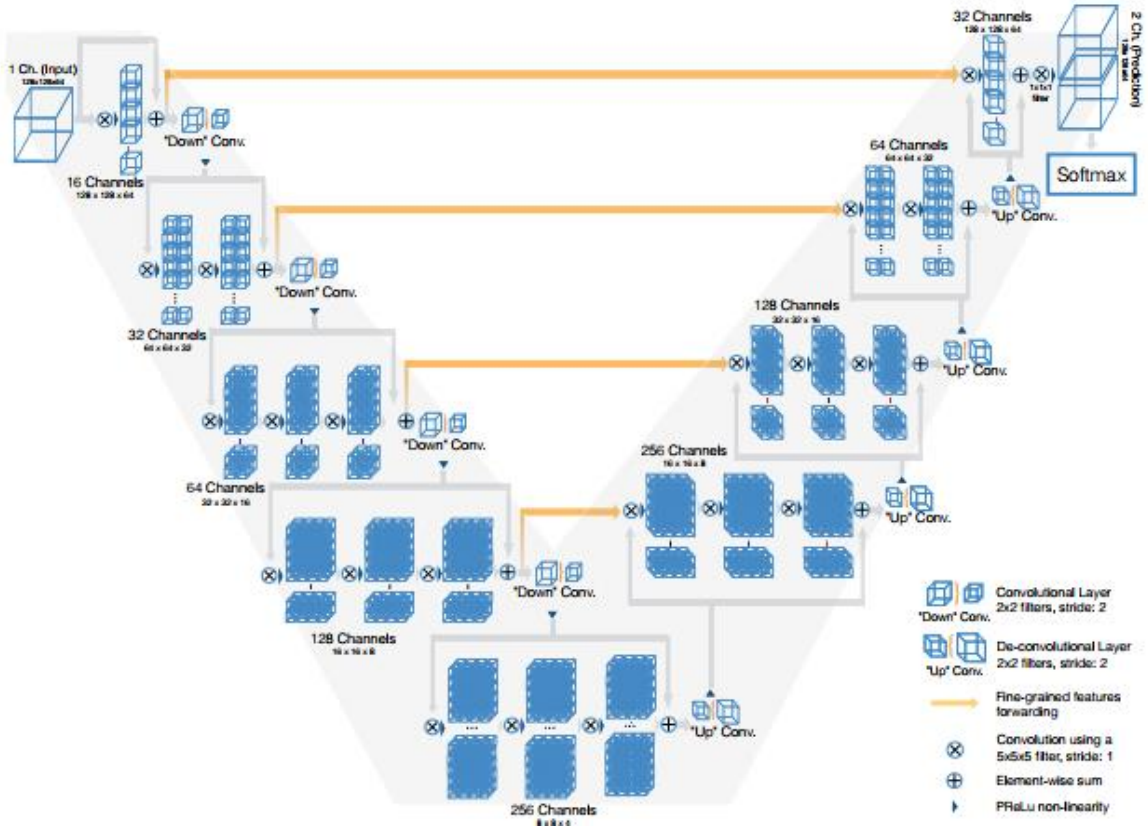
4) شبکه V-Net

همانطور که در قسمت خلاصه مقاله به آن اشاره شده است، شبکه های عصبی کانوولوشنی، به تازگی در Machine vision و پردازش تصاویر پزشکی استفاده شده است اما این شبکه ها اکثرا میتوانند تصاویر دو بعدی را پردازش کنند و این در حالی است که دیتا های پزشکی که ذخیره میشود و استفاده میشود، در سه بعد هستند و حجم دارند. شبکه ها عصبی V-Net برای همین امر هستند، برای segmentation تصاویر سه بعدی با ساختاری کاملا کانوولوشنی.

این شبکه عصبی با دیتاست حجم دار در سه بعد از MRI پروستات آموزش دیده است و قابلیت segmentation را برای داده های تست، فراهم کرده است.

این شبکه عصبی بر اساس ضریب Dice می آید و Objective function را بهینه میکند. بدین ترتیب میتوانیم robustness خوبی در برابر وضعیت غیر متوازن در برابر foreground و background voxel داشته باشیم

ساختار کلی این شبکه عصبی در زیر نشان داده شده است:



شکل 2-4-1 ساختار شبکه عصبی V-Net

همانطور که در تصویر نشان داده شده است، این شبکه عصبی نیازمند اجرای عملیات مقابل است:

1- Conv 2*2, stride 2
2- De-cove 2*2, stride 2

3- Forwarding
4- Conv 5*5*5, stride 1
5- Element-wise sum
6- PReLU برای تابع فعال ساز

Layer	Input Size	Receptive Field
L-Stage 1	128	$5 \times 5 \times 5$
L-Stage 2	64	$22 \times 22 \times 22$
L-Stage 3	32	$72 \times 72 \times 72$
L-Stage 4	16	$172 \times 172 \times 172$
L-Stage 5	8	$372 \times 372 \times 372$
R-Stage 4	16	$476 \times 476 \times 476$
R-Stage 3	32	$528 \times 528 \times 528$
R-Stage 2	64	$546 \times 546 \times 546$
R-Stage 1	128	$551 \times 551 \times 551$
Output	128	$551 \times 551 \times 551$

شکل 2-4-2 مشخصات لایه های شبکه عصبی V-Net

در طی مسیری که دنبال میشود، ما به تعداد Channel ها اضافه میکنیم، دو برابر میکنیم از 16 به بعد، و بیشتر به پاسخ "چه object " پاسخ میدهیم (مرحله Contraction)، سپس سعی میکنیم به High resolution از segmentation برسیم (مرحله Expansion).

حال به سراغ تابع خطای استفاده شده در این شبکه عصبی میرویم، ابتدا تبیین مینماییم که ضریب Dice از فرمول زیر حساب میشود:

$$D = \frac{2\sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$

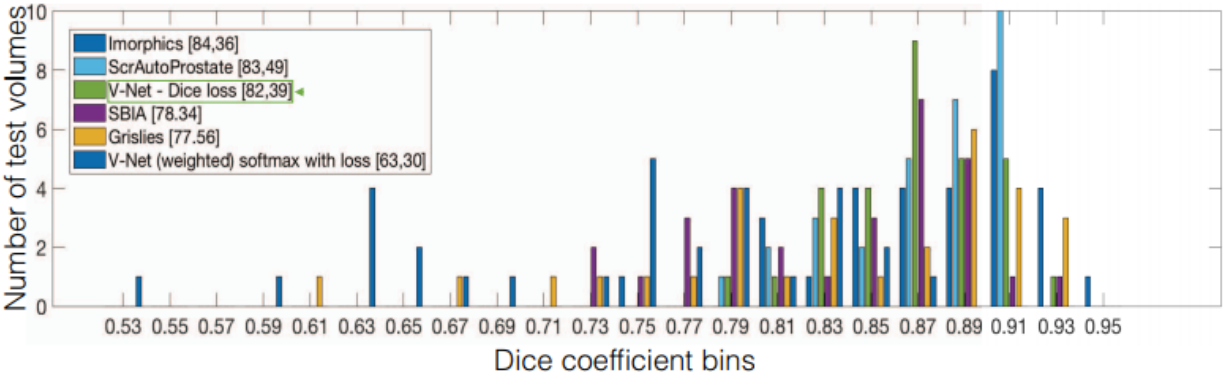
که در آن N مجموع تعداد حجم های کوچک voxel است. P در واقع label پیشبینی شده و g در واقع label واقعی ما میباشد.

حال تابع خطا را بر اساس مشتق ضریب Dice نسبت به P_j میگیریم و مانند زیر بدست می آوریم:

$$\frac{\partial D}{\partial p_j} = 2\left[\frac{g_j\left(\sum_i^N p_i^2 + \sum_i^N g_i^2\right) - 2p_j\left(\sum_i^N p_i g_i\right)}{\left(\sum_i^N p_i^2 + \sum_i^N g_i^2\right)^2}\right]$$

با این تعریف تابع خطا، میتوانیم robustness خوبی در برابر وضعیت غیر متوازن در برابر foreground و background voxel داشته باشیم.

در شکل زیر توزیع حجم های segment شده با توجه به ضریب Dice رسم شده است(نمودار سبزرنگ):



شکل 2-4-3 توزیع حجم های segment شده با توجه به ضریب Dice

بعد از آموزش بر روی داده های PROMISE 2012 نتایج عددی زیر برای بکارگیری هرکدام از الگوریتم های اشاره شده، بدست آمده است:

Algorithm	Avg. Dice	Avg. Hausdorff distance	Score on challenge task	Speed
V-Net + Dice-based loss	0.869 ± 0.033	5.71 ± 1.20 mm	82.39	1 sec.
V-Net + mult. logistic loss	0.739 ± 0.088	10.55 ± 5.38 mm	63.30	1 sec.
Imorphics [22]	0.879 ± 0.044	5.935 ± 2.14 mm	84.36	8 min.
ScrAutoProstate	0.874 ± 0.036	5.58 ± 1.49 mm	83.49	1 sec.
SBIA	0.835 ± 0.055	7.73 ± 2.68 mm	78.33	–
Grislies	0.834 ± 0.082	7.90 ± 3.82 mm	77.55	7 min.

شکل 2-4-4 نتایج الگوریتم های متفاوت روی دیتاست PROMISE 2012

این نتایج، حاکی از نتایج بسیار مطلوب الگوریتم V-Net Dice based میباشد. سرعت بسیار بالا نسبت الگوریتم های Imorphics، Grislies و امتیاز بسیار بالا نسبت به V-Net mult. Logistic و SBIA و Grislies میباشد.