Militar Aircraft Detection

Hardware

This model was trained on a kaggle notebook using the gpu T4 x2 with 15gb vram

Pre trained model comparison

I compared the models from pytorch by establishing the same parameters.

- Epochs: 200
- Early stopping with patience of 10
- Alpha: 0.7
- Learning rate: 1e-4
- L2 lambda: 1e-6
- batch: 32
- Image size: 400 x 255
- freeze all layers of the pretrained model
- Torch seed: 32
- Split seed: 42
- Augmentation: 2

The Classification head architecture is

```
self.cls_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
```

```
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, n_classes)
        )
```

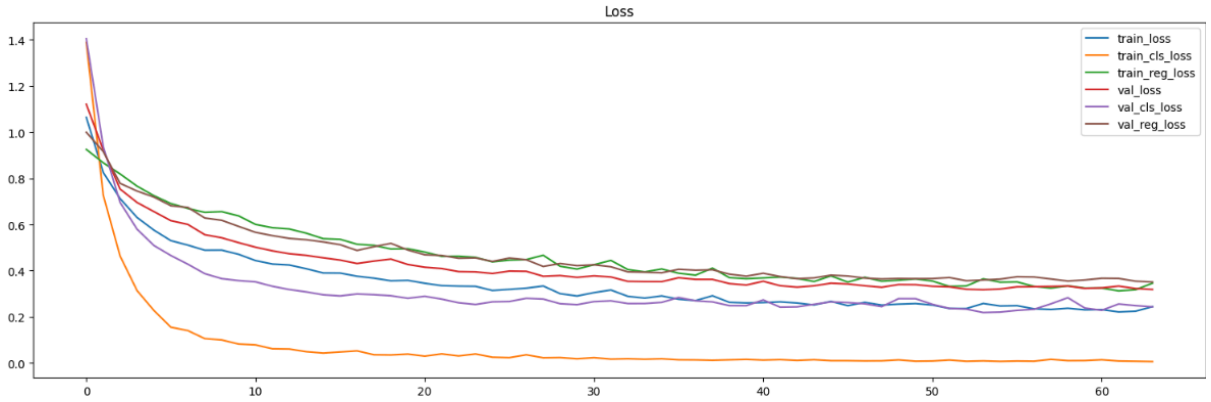
The regression head architecture is

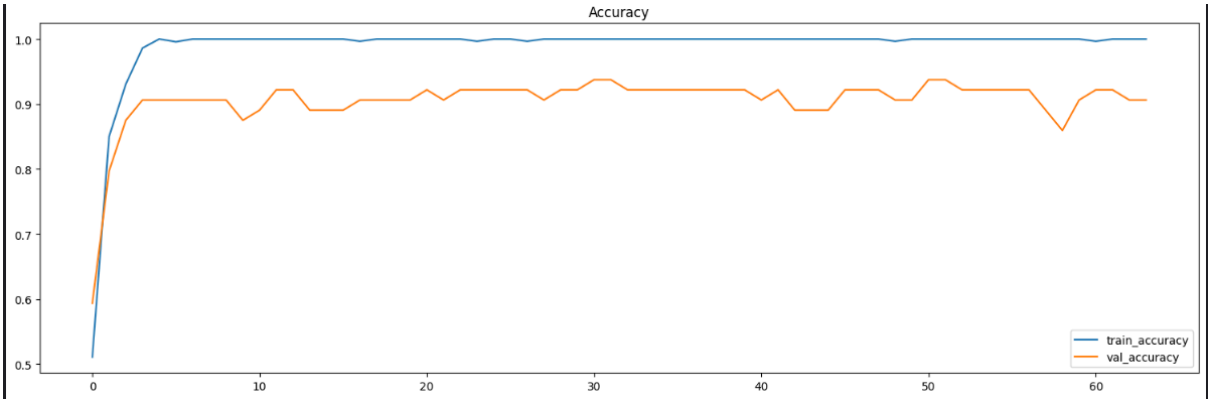
```
self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )
```

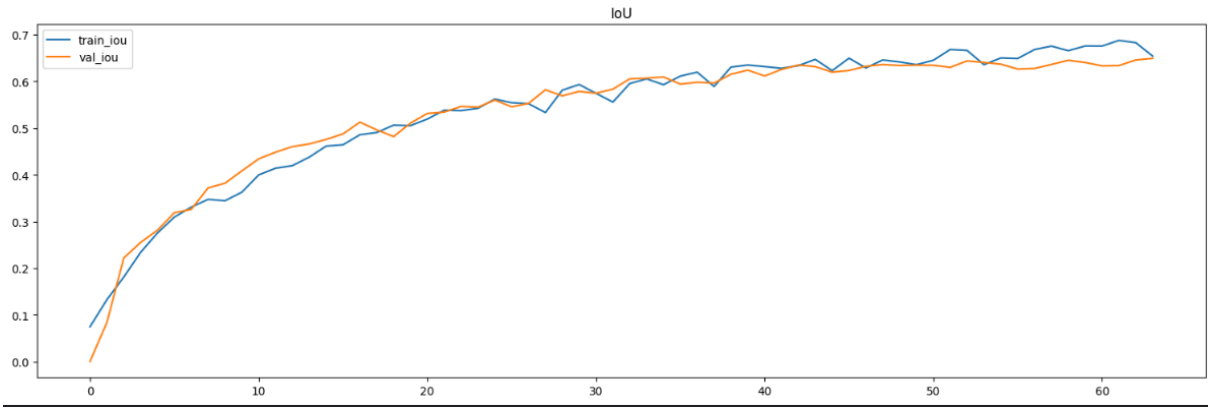
VGG 16

The best epoch was the 54

Training plots







The plots shows overfitting for the accuracy, but the iou and other losses tend to be close. This suggest the model should focus more on predict the bounding boxes instead of the classes, and also try alternatives to reduce overfitting on the accuracy head like reduce layers and include a dropout

Sample evaluation

Test Score

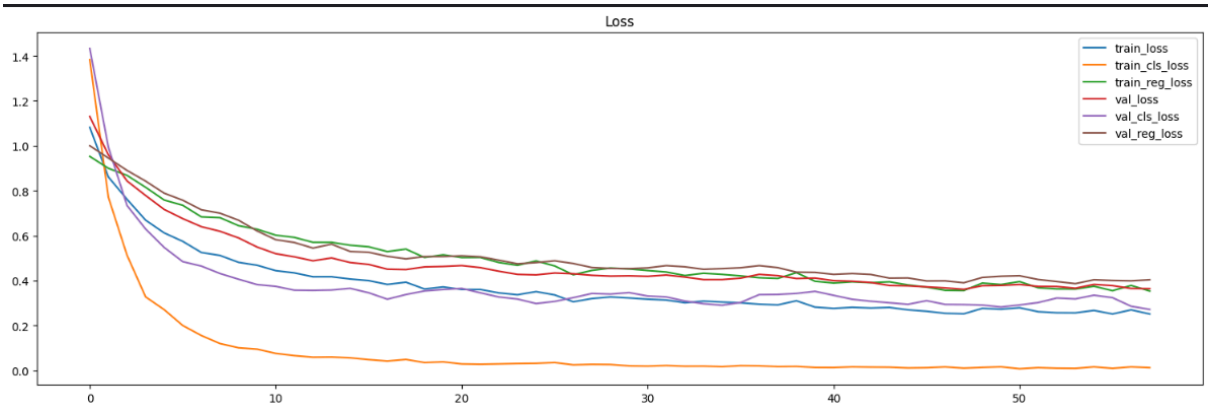
- private: 0.748

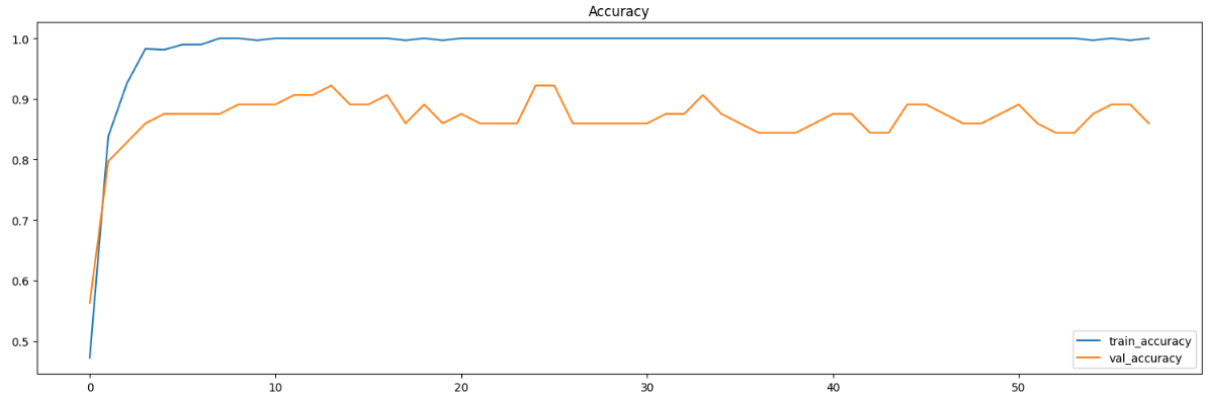
- public: 0.760

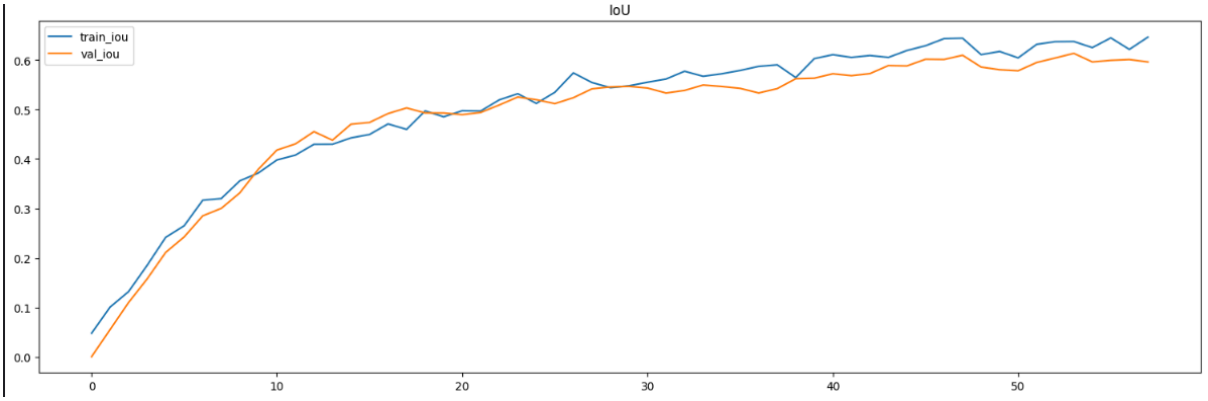
VGG 19

Best epoch was 48

Training plots







Sample evaluation

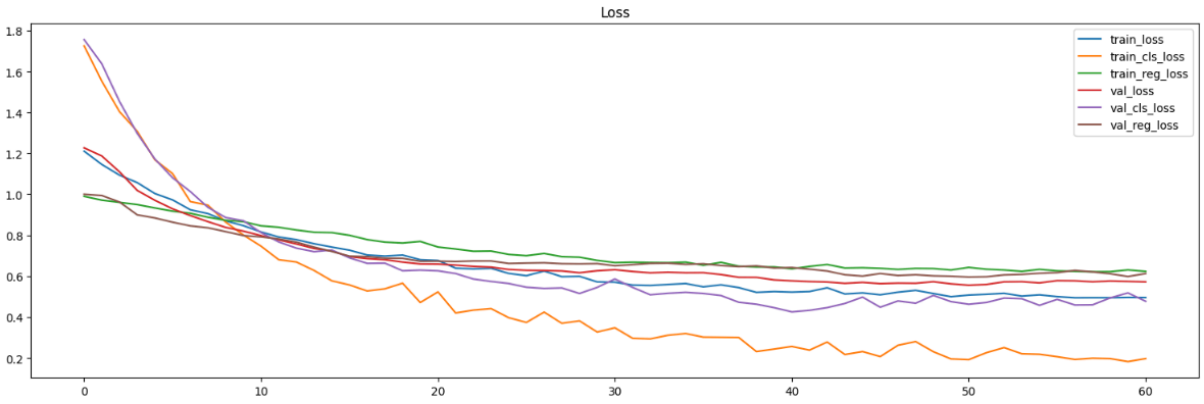
Test Score

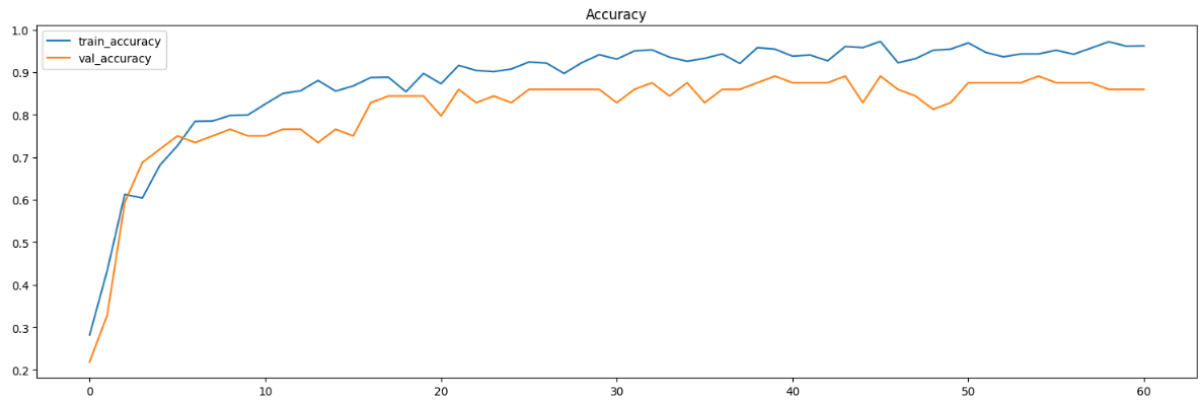
- private: 0.722
- public: 0.759

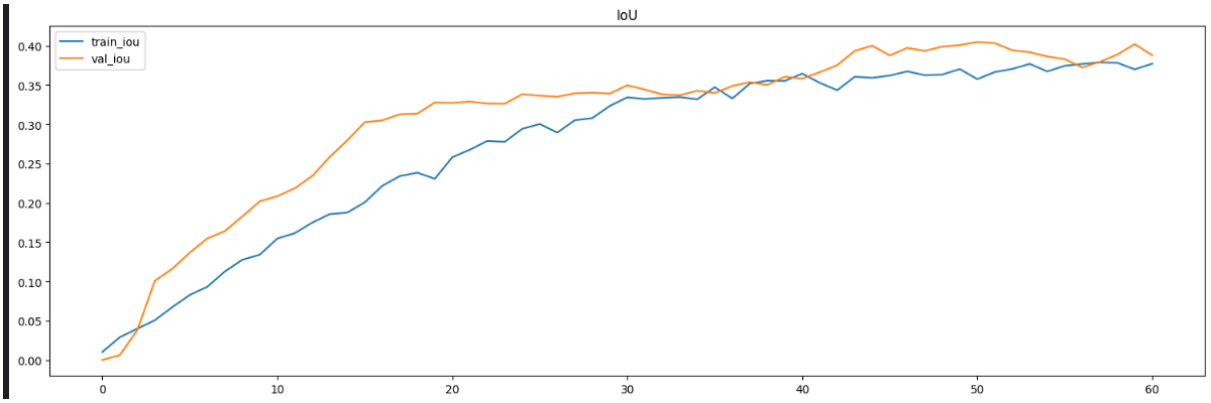
Efficient Net V2 small

best epoch was 51

Training plots







Sample evaluation

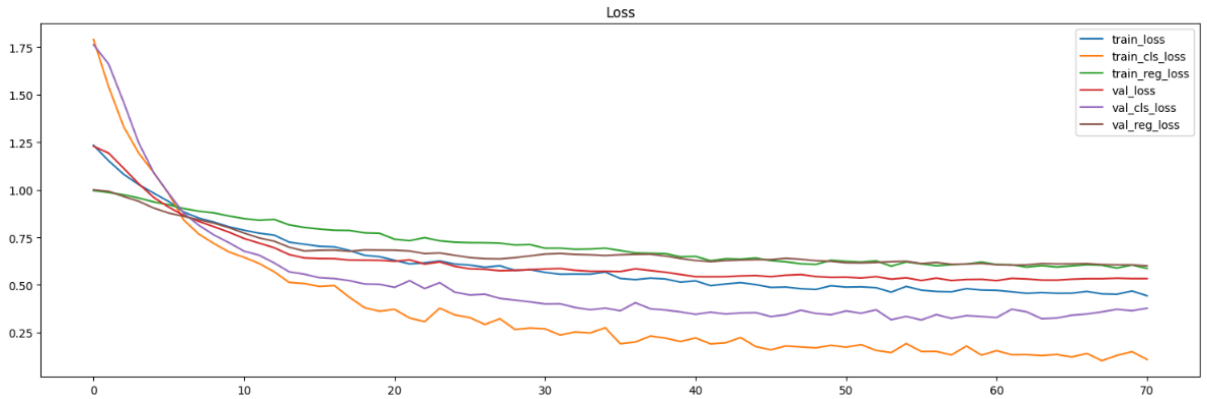
Test Score

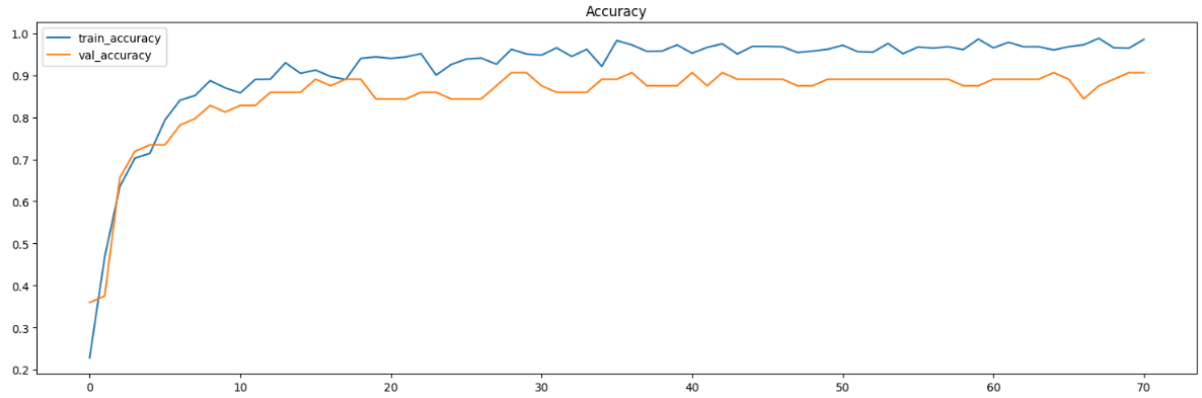
- private: 0.605
- public: 0.631

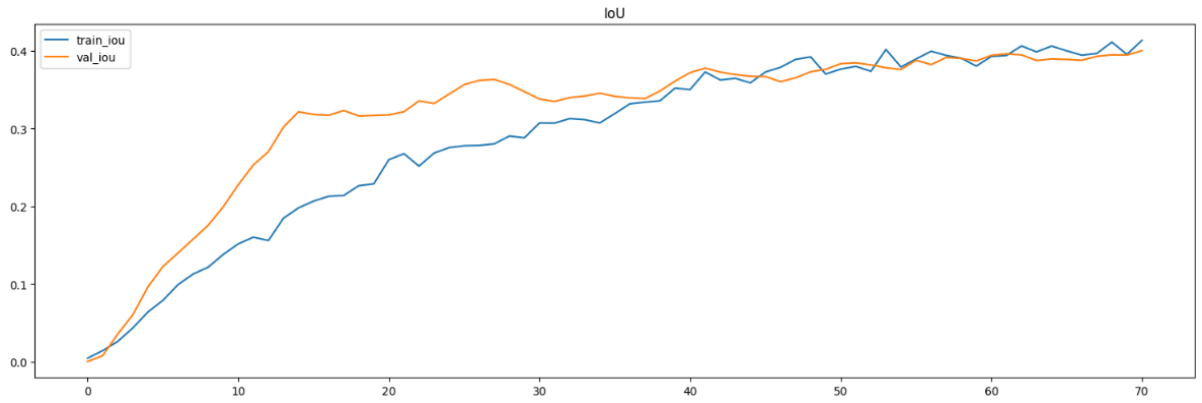
EfficientNet V2 medium

best epoch was 61

Training plots







Sample evaluation

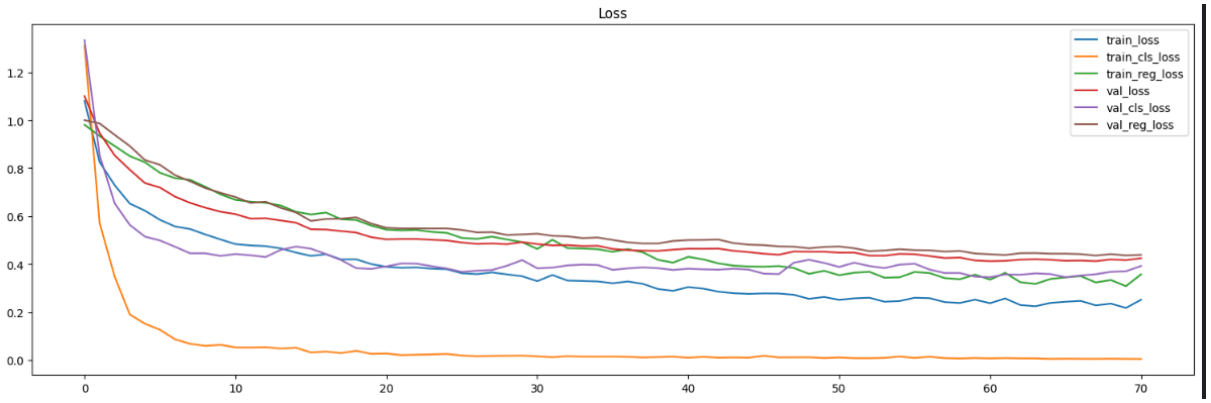
Test Score

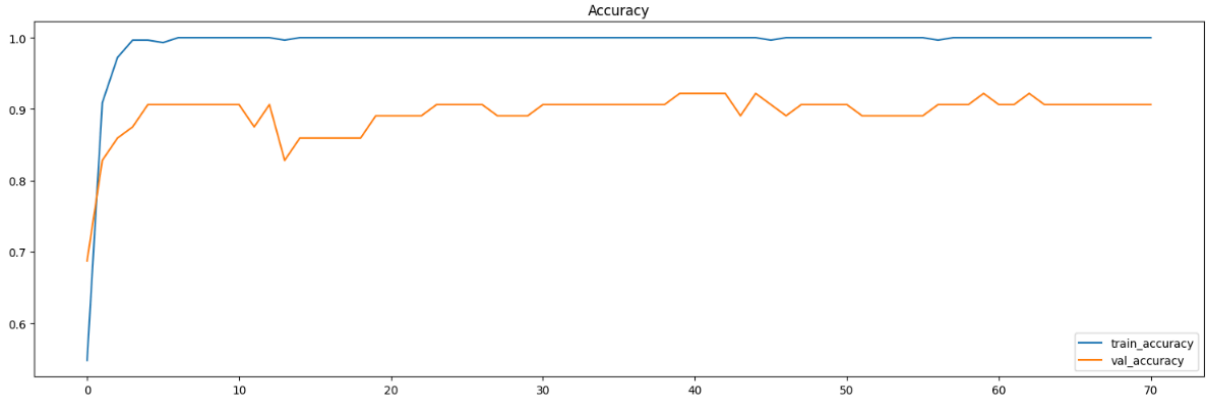
- private: 0.601
- public: 0.657

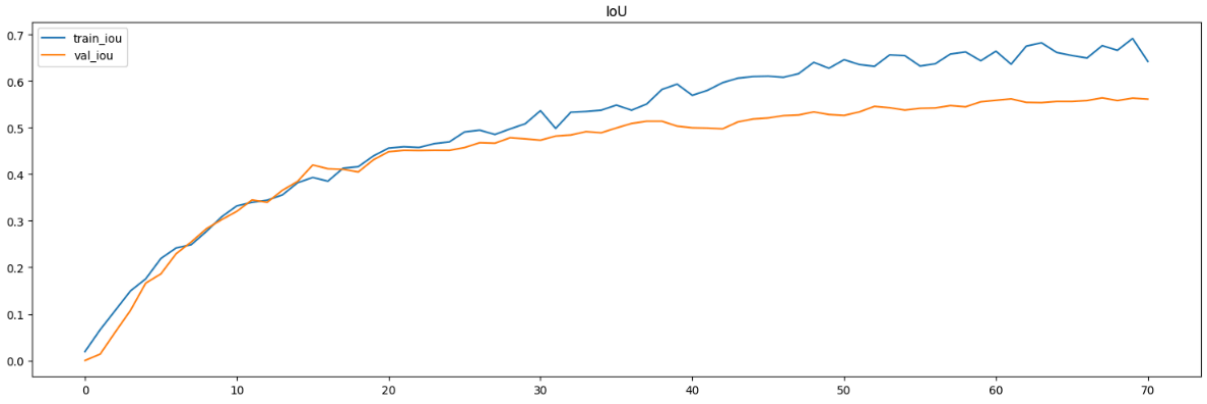
MobileNet V2

Best epoch was 61

Training Plots







Sample evaluation

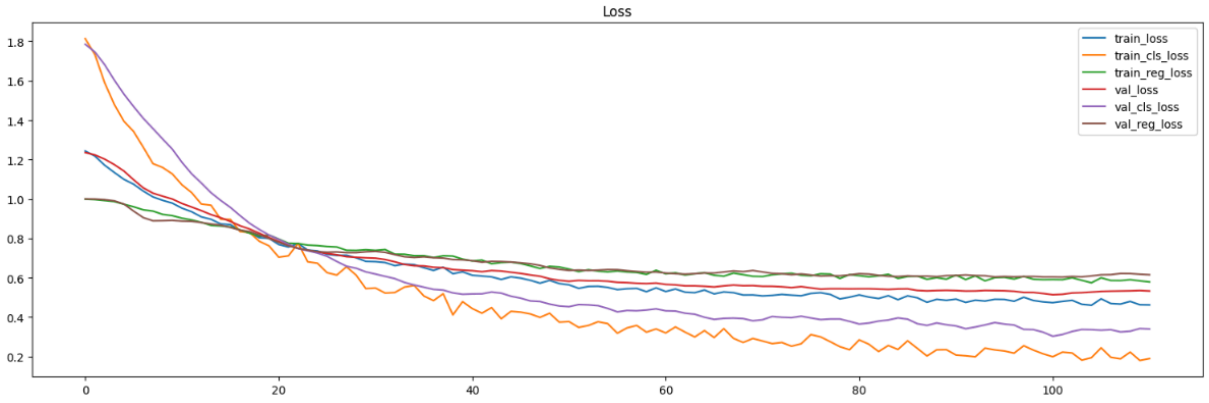
Test score

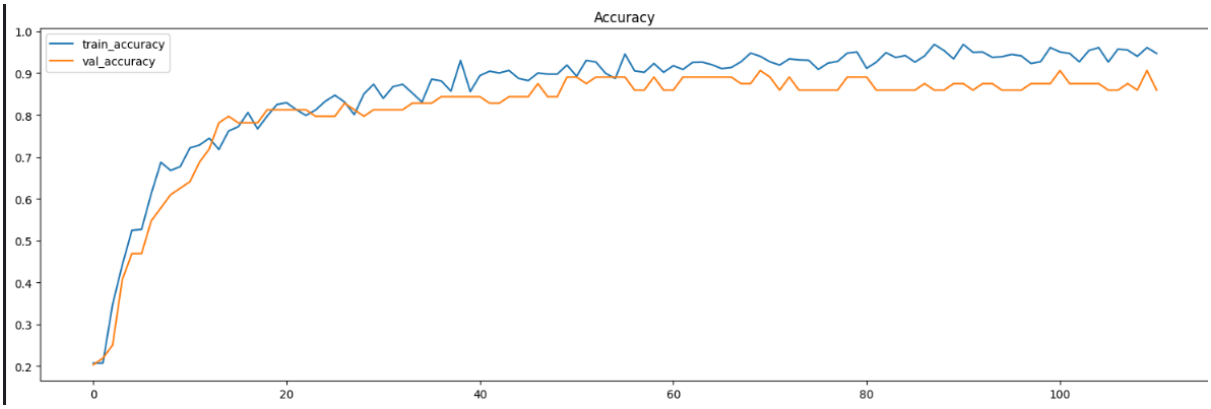
- private: 0.731
- public: 0.729

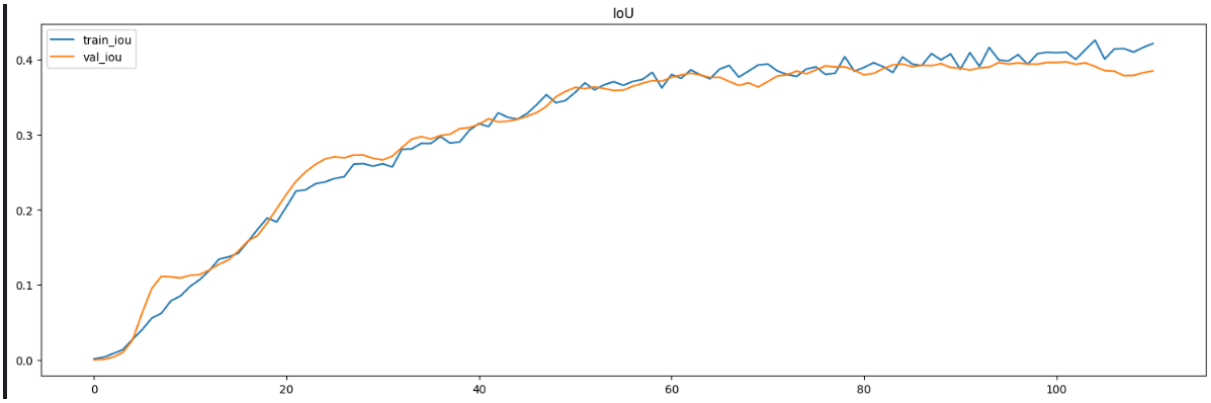
MobileNet V3 small

best epoch was 101

Training Plots







Sample evaluation

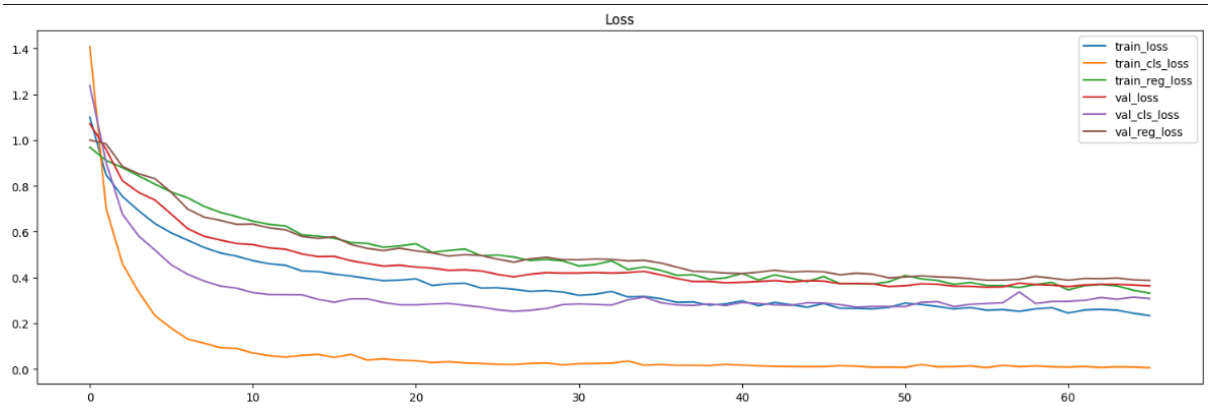
Test Score

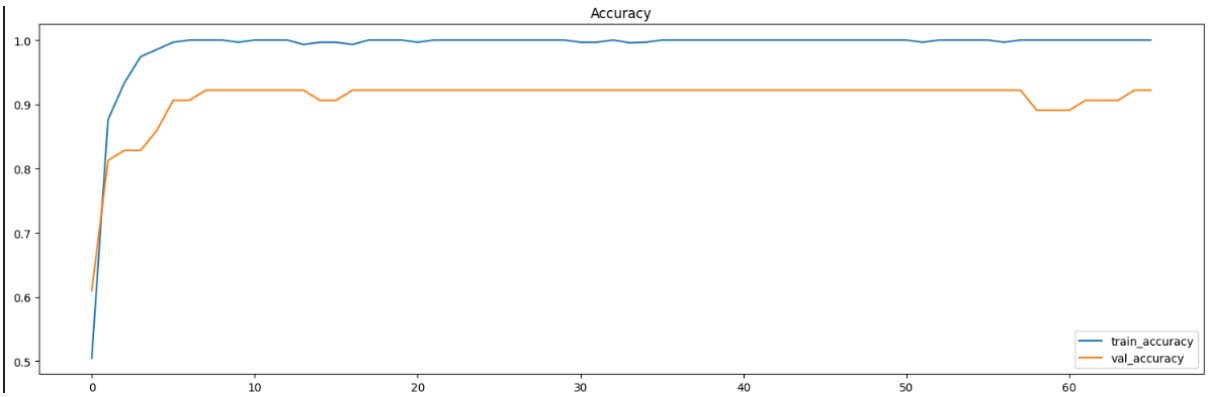
- private: 0.597
- public: 0.609

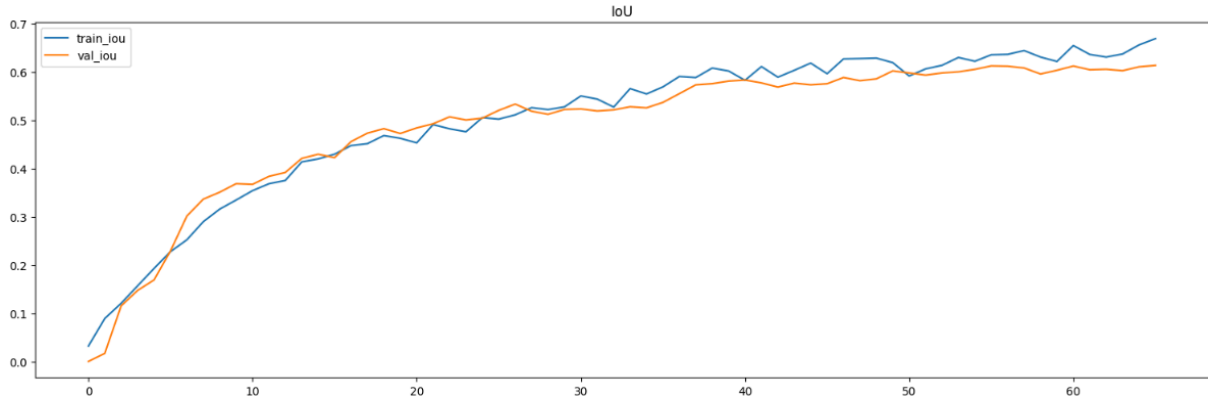
SqueezeNet V1

best epoch was 56

Training plots







Sample evaluation

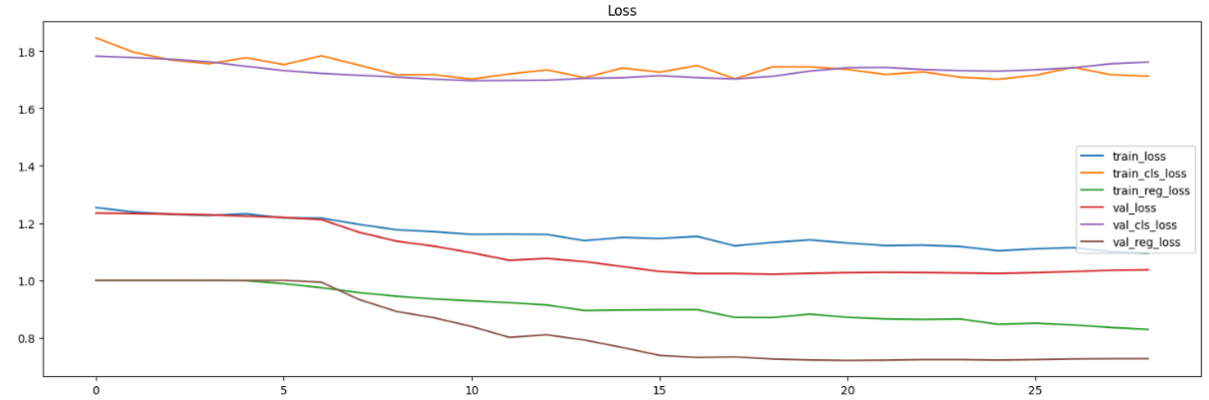
Test Score

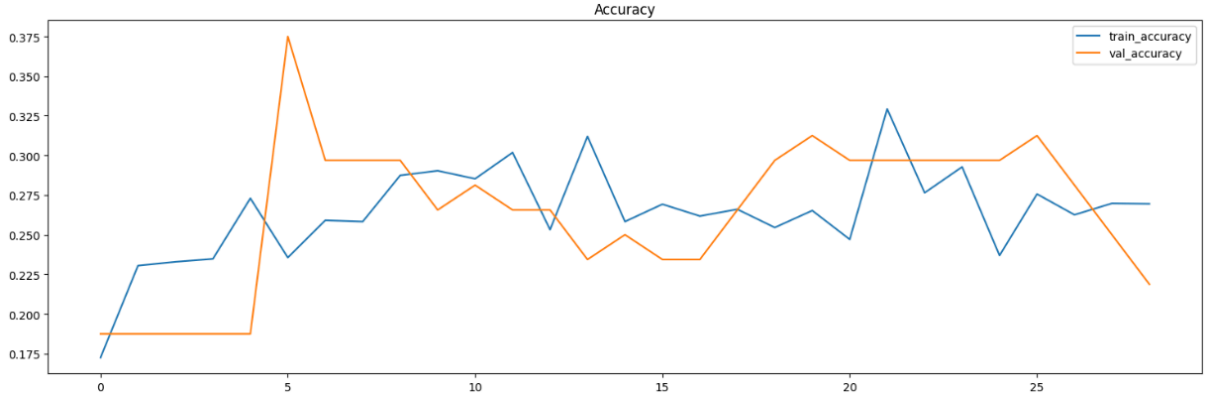
- private: 0.76
- public: 0.755

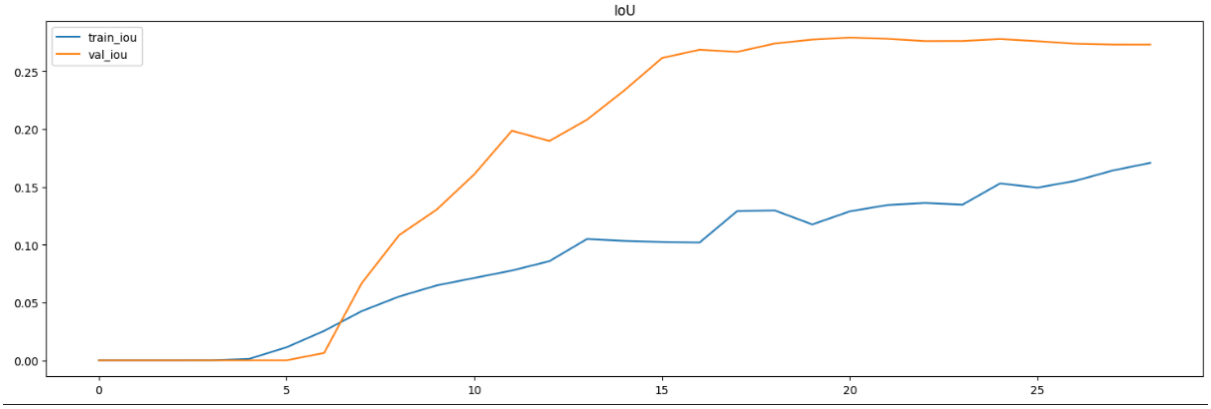
SwinTransform

best epoch was 19

Training plots







Sample evaluation

Test Score

- private: 0.261
- public: 0.184

Summary

Model	Private Score	Public Score
Swin Transform	0.26313	0.18428
SqueezeNet V1	0.76007	0.75522
MobileNet V3 small	0.59794	0.60984
MobileNet V2	0.73187	0.72945
EfficientNet V2 mid	0.60172	0.65772
EfficientNet V2 small	0.60543	0.63146
VGG19	0.72206	0.75934
VGG16	0.74856	0.76094

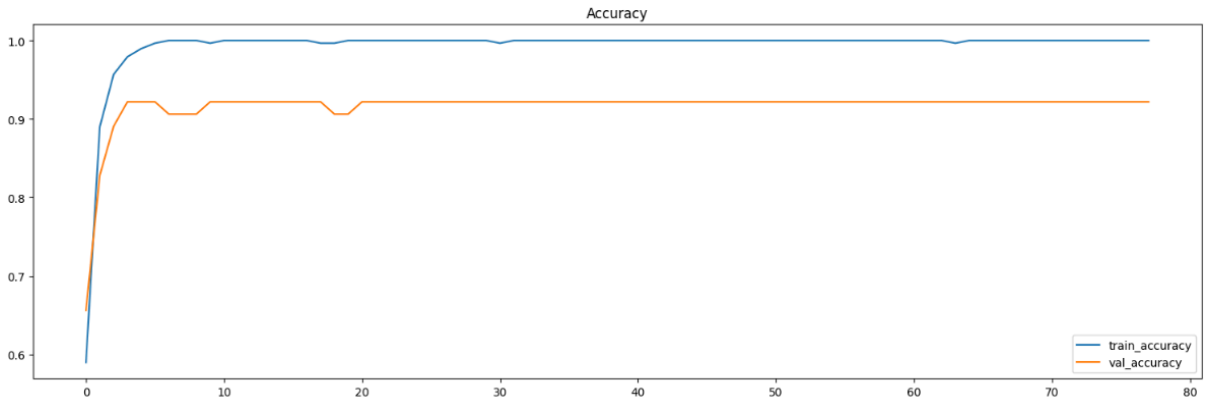
The best backbone model appears to be SqueezeNet so now we try different heads architecture and hyperparams

Iterate classification head

```
        self.cls_head = nn.Sequential(
```

```
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, n_classes)
        )
```

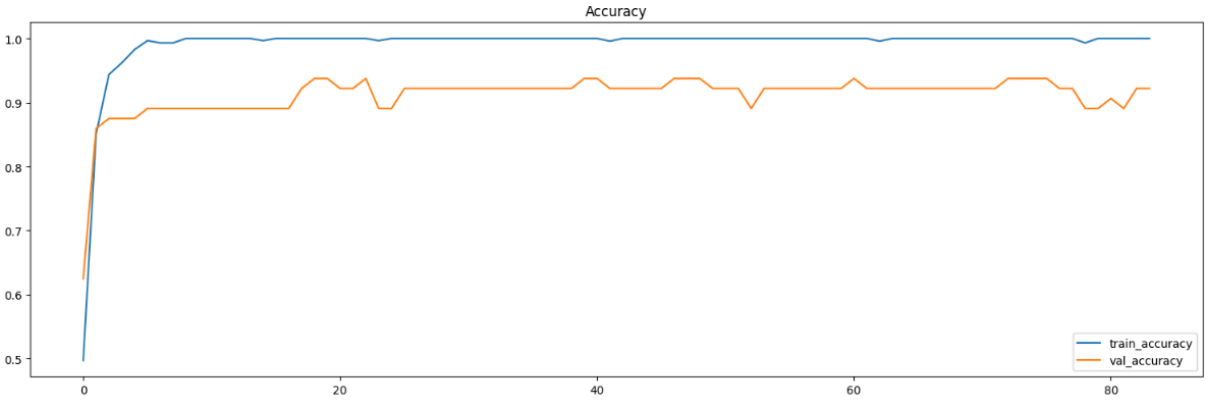
epoch 68



private score: 0.74638
public score: 0.74779
val accuracy: 0.92

```
        self.cls_head = nn.Sequential(
            nn.Linear(flattened_features, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, n_classes)
        )
```

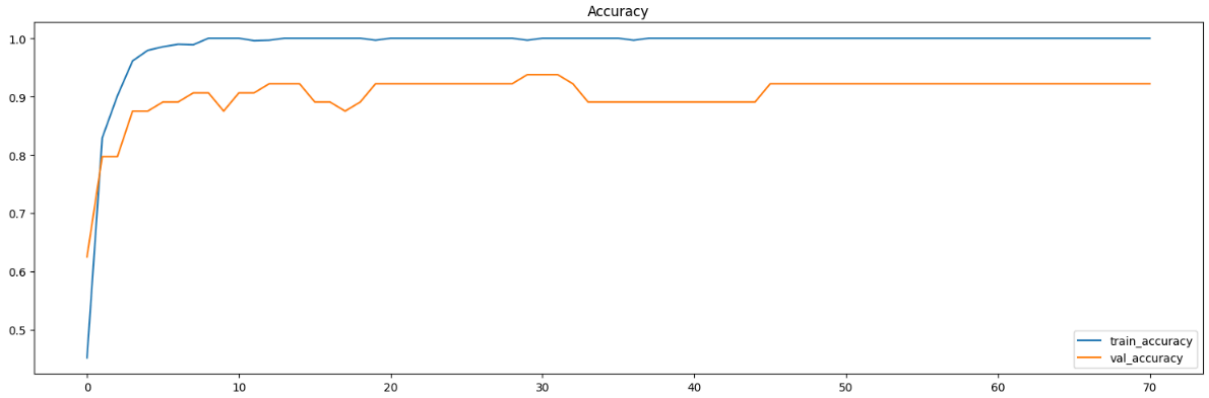
epoch 74



private: 0.75190
public: 0.77181
val acuracy: 0.9375

```
self.cls_head = nn.Sequential(
            nn.Linear(flattened_features, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, n_classes)
        )
```

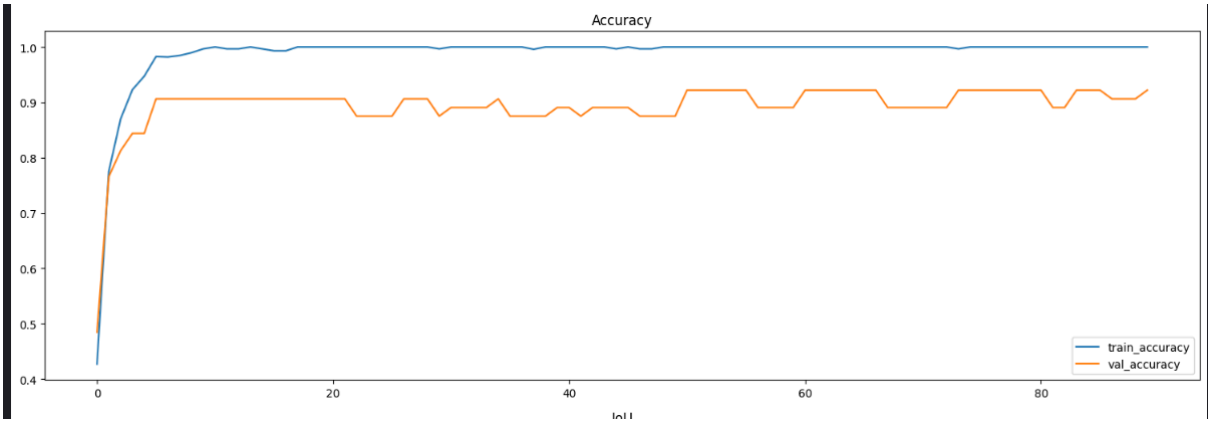
epoch 61



val accuracy: 0.921875
private: 0.74239
public: 0.75499

```
self.cls_head = nn.Sequential(
            nn.Linear(flattened_features, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Linear(64, n_classes)
        )
```

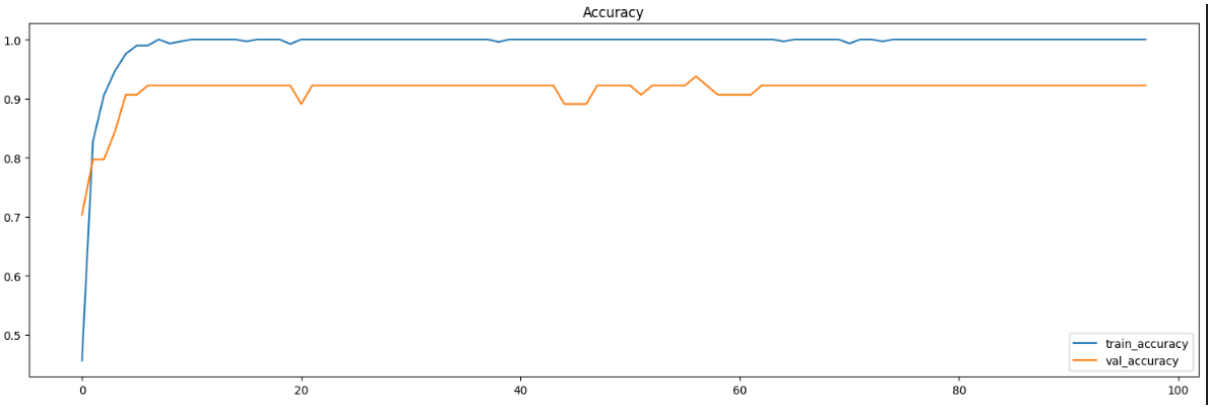
epoch 80



val accuracy 0.921875
private: 0.73628
public: 0.77367

```
self.cls_head = nn.Sequential(
            nn.Linear(flattened_features, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256,128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, n_classes)
        )
```

epoch 88



val accuracy 0.921875
private 0.74936
public: 0.77877

Best model

self.cls_head = nn.Sequential(
nn.Linear(flattened_features, 256),
nn.BatchNorm1d(256),
nn.ReLU(),
nn.Linear(256, n_classes)
)

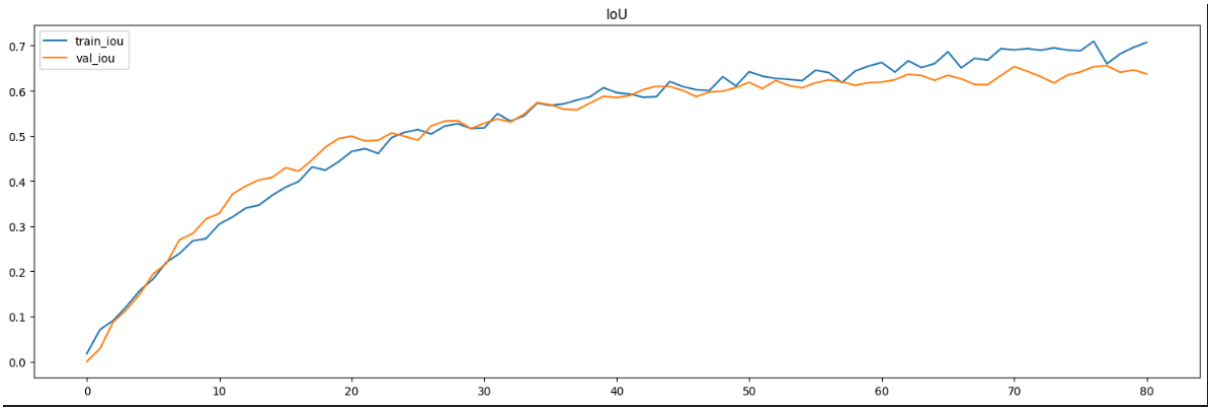
Iterate regression head

Base model

self.reg_head = nn.Sequential(
nn.Linear(flattened_features, 768),
nn.BatchNorm1d(768),
nn.ReLU(),
nn.Linear(768, 256),
nn.BatchNorm1d(256),
nn.ReLU(),
nn.Linear(256, 128),
nn.BatchNorm1d(128),
nn.ReLU(),
nn.Linear(128, 4)
)

```
        self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )
```

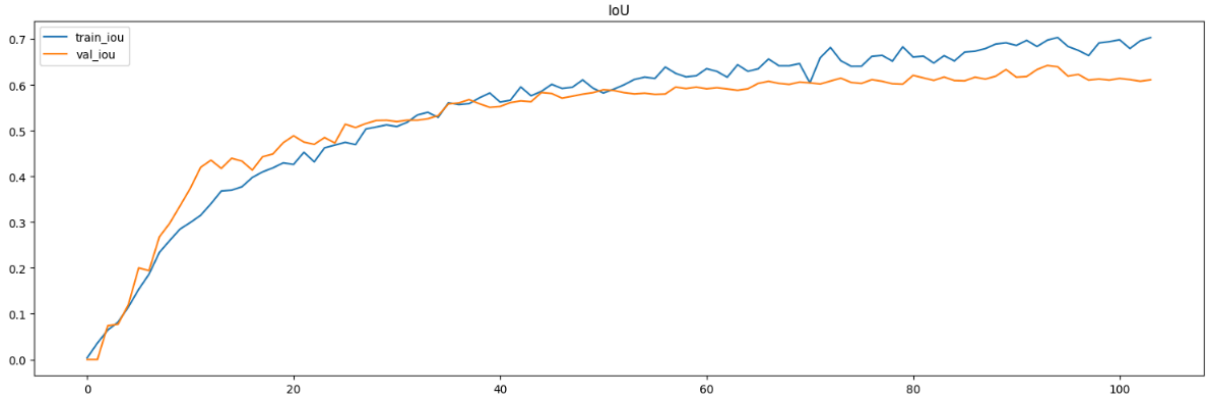
epoch 71



val iou: 0.6536
private: 0.76657
public: 0.78841

```
        self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )
```

epoch 94

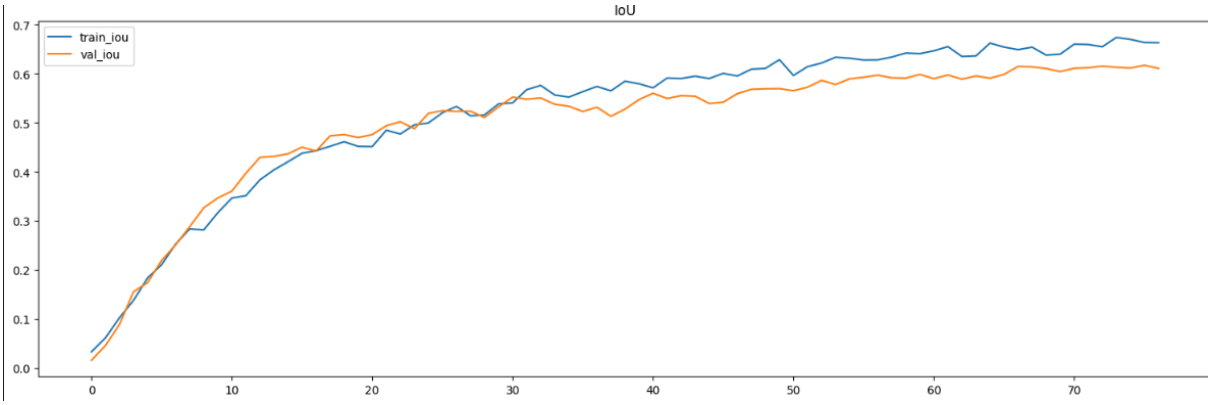


val iou: 0.6422
privfate: 0.78071
public: 0.74559

```
        self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 128),
```

```
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )
```

epoch 67

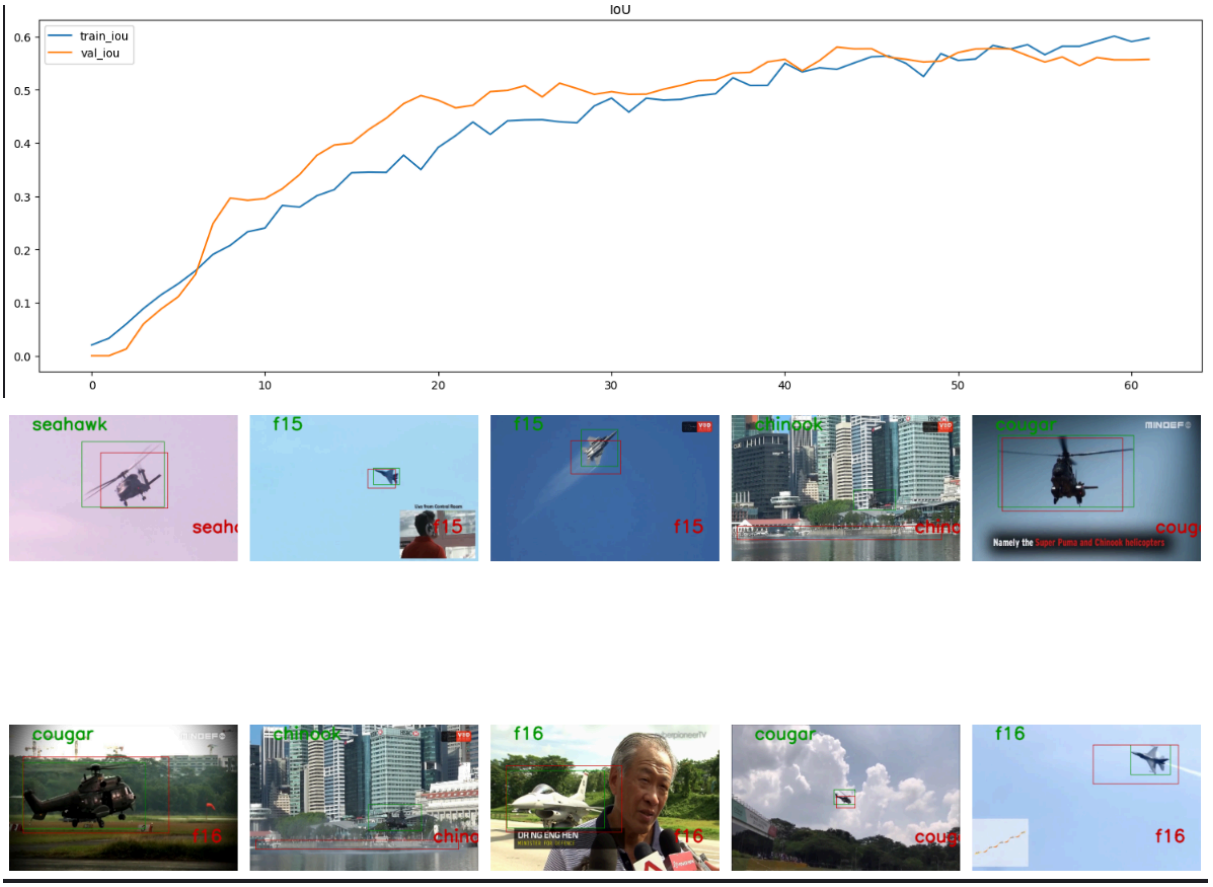


val iou 0.6146
private: 0.74318
public: 0.72761

```
        self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
```

```
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Linear(64, 4)
        )
```

epoch 52



val iou: 0.5765
private: 0.73928
public: 0.72228.

Best model

First and second iterations tend to have better performance. But the second one shows more accurate boxes.

```
        self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
```

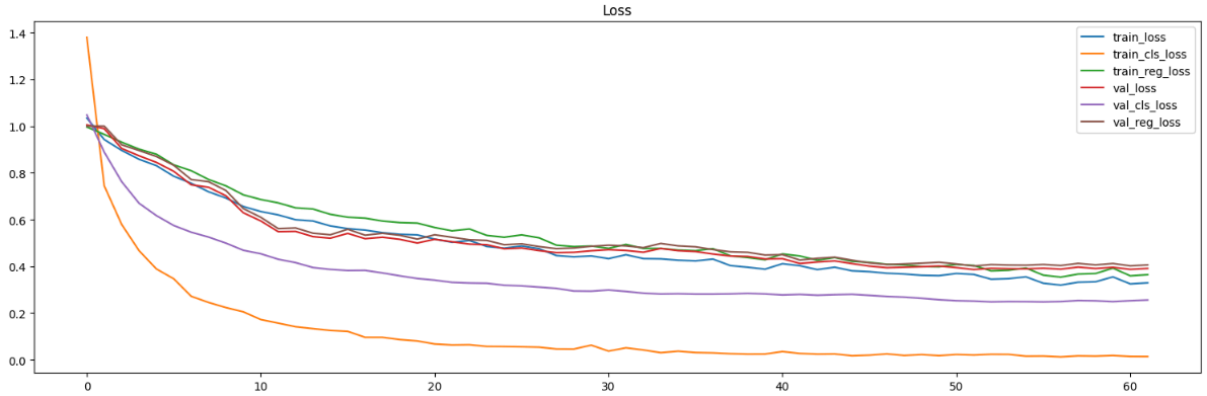
```
            nn.ReLU(),
            nn.Linear(768, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )
```

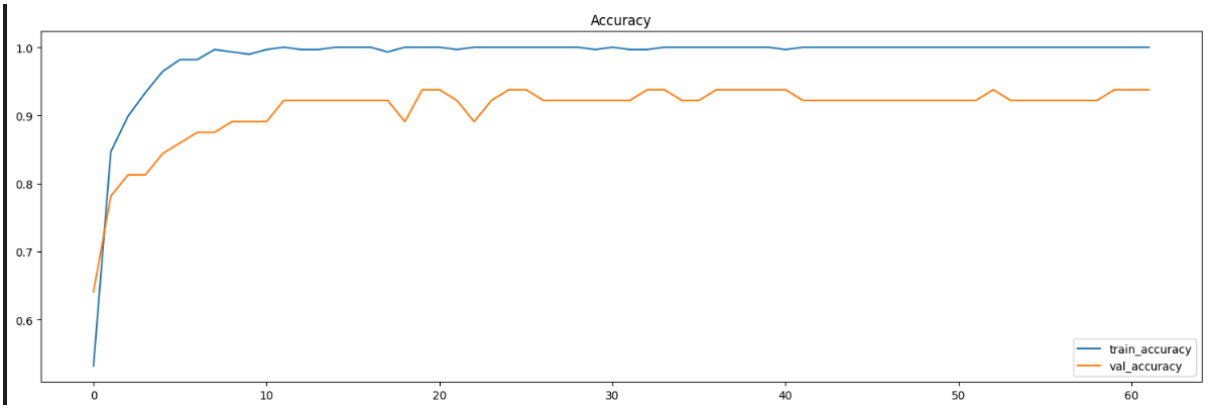
Iterate alpha

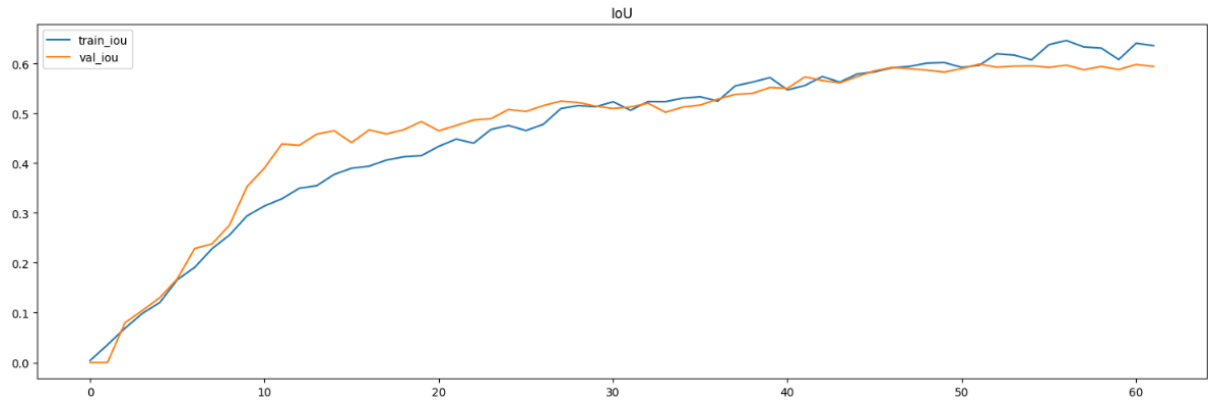
The first value for previous executions was 0.7. First try bigger ones since accuracy appears to fast get ceil, while iou still have improvements to make.

Alpha 0.9

Epoch 52:
train_loss = 0.36544396811061436
train_cls_loss = 0.02057261775351233
train_reg_loss = 0.40376301606496173
train_iou = 0.5962434999601326
train_accuracy = 1.0
val_loss = 0.38623547554016113
val_cls_loss = 0.25137069821357727
val_reg_loss = 0.4012204706668854
val_iou = 0.5987892826640793
val_accuracy = 0.921875





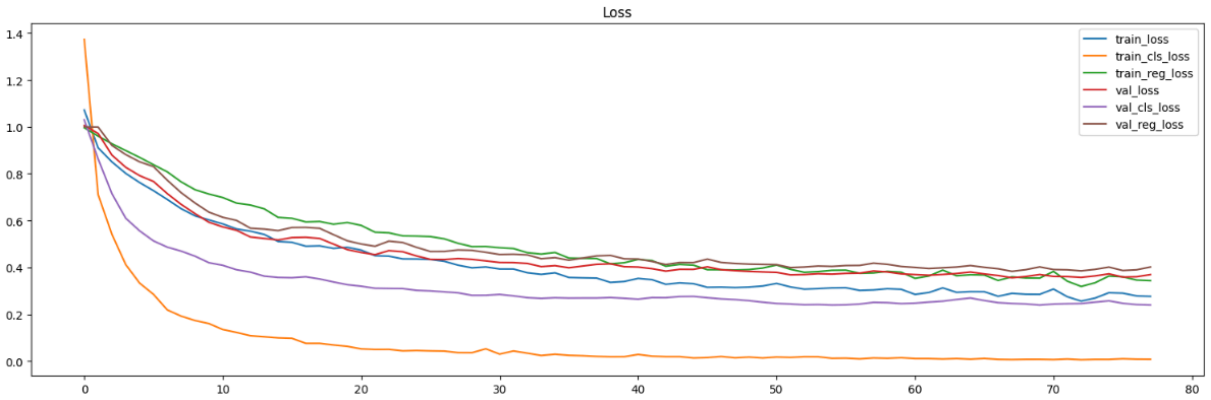


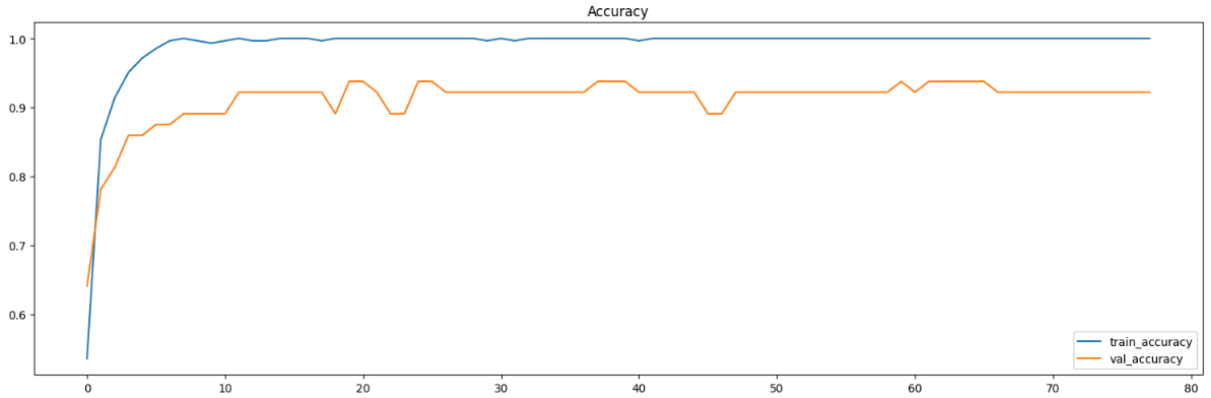
private: 0.73896
public: 0.74863

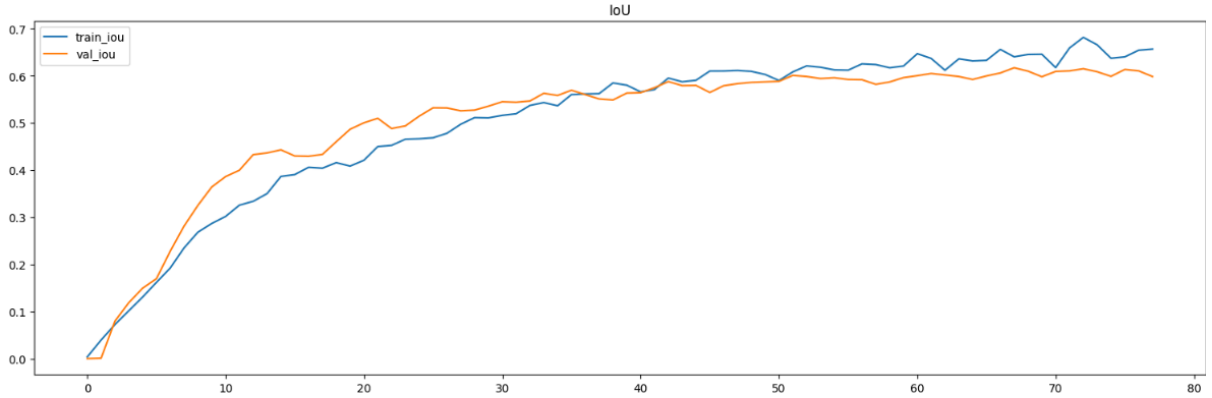
Alpha 0.8

Epoch 68:
train_loss = 0.2894292887714174
train_cls_loss = 0.006831367189685504
train_reg_loss = 0.36007876528633964
train_iou = 0.6399284664853188
train_accuracy = 1.0
val_loss = 0.35589464008808136

val_cls_loss = 0.2462838515639305
val_reg_loss = 0.383297324180603
val_iou = 0.6167135796755725
val_accuracy = 0.921875



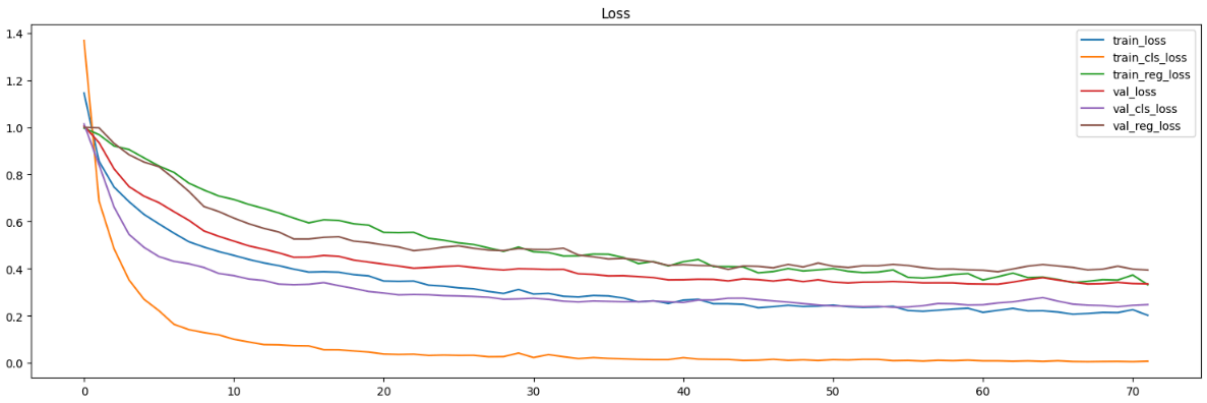


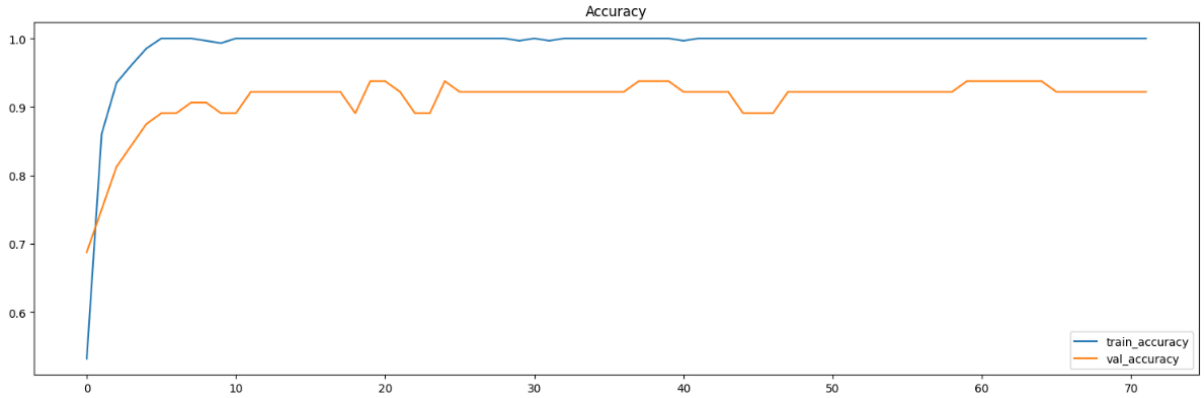


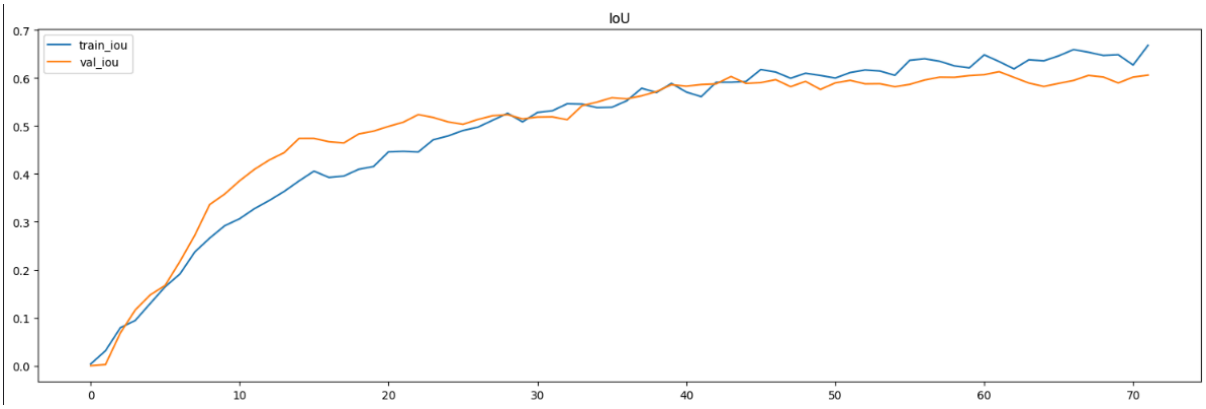
private: 0.73559
publico: 0.76797

Alpha 0.6

Epoch 62:
train_loss = 0.22271745238039228
train_cls_loss = 0.00851239093268911
train_reg_loss = 0.3655208150545756
train_iou = 0.6344858916466946
train_accuracy = 1.0
val_loss = 0.33365805447101593
val_cls_loss = 0.25418177247047424
val_reg_loss = 0.3866422176361084
val_iou = 0.6133674096310813
val_accuracy = 0.9375





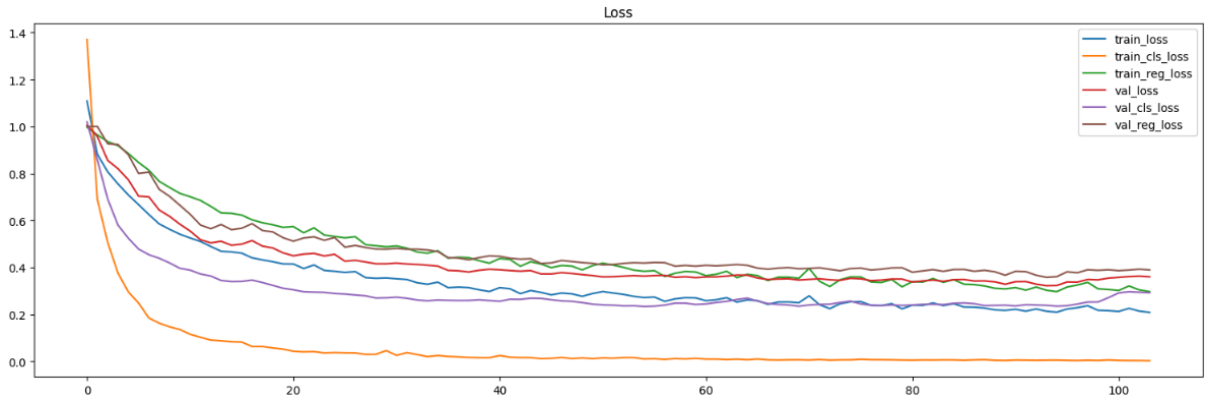


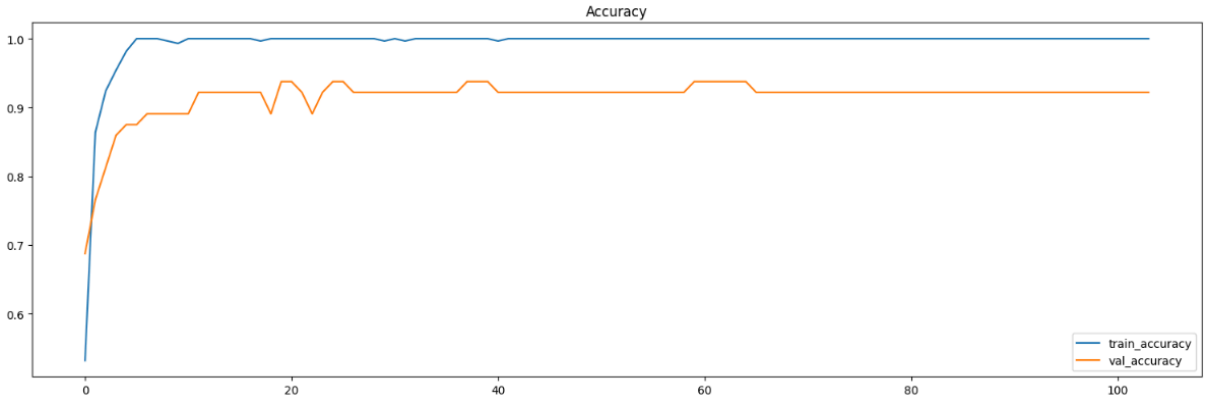
private: 0.74001
public: 0.75760

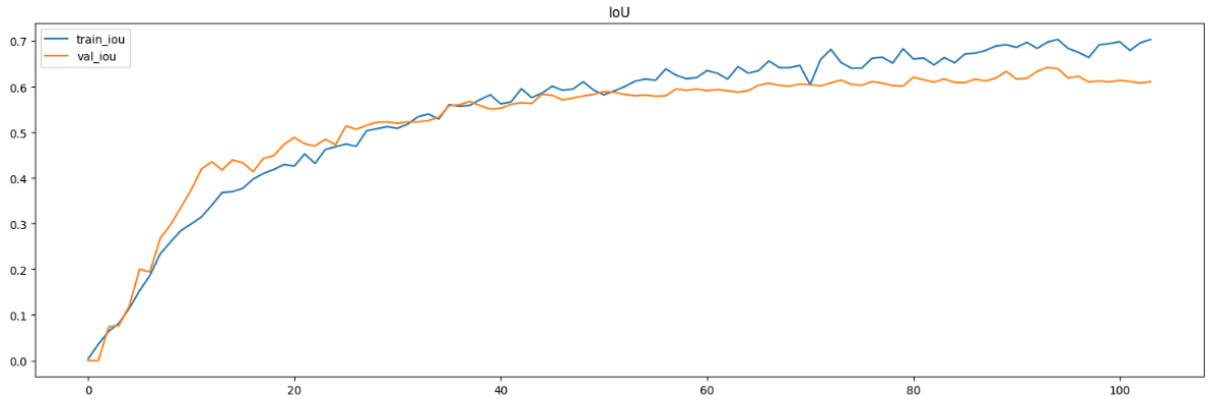
Alpha 0.7

Epoch 94:
train_loss = 0.21331014898088244
train_cls_loss = 0.004949078688191043
train_reg_loss = 0.30260774824354386
train_iou = 0.6974008853205423
train_accuracy = 1.0
val_loss = 0.32194332778453827
val_cls_loss = 0.2383575215935707

val_reg_loss = 0.3577658236026764
val_iou = 0.6422453311397442
val_accuracy = 0.921875







private: 0.78071
public: 0.74559

Best alpha

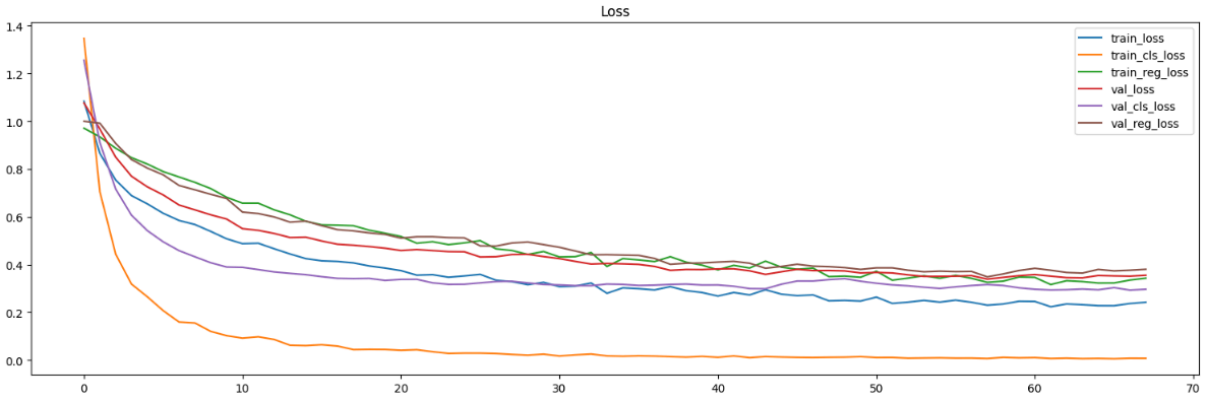
With 0.7 we obtain the best results. with 0.6 was the second best value, it could be worth trying more numbers between those two, but the improvement could be too low to be worth it.

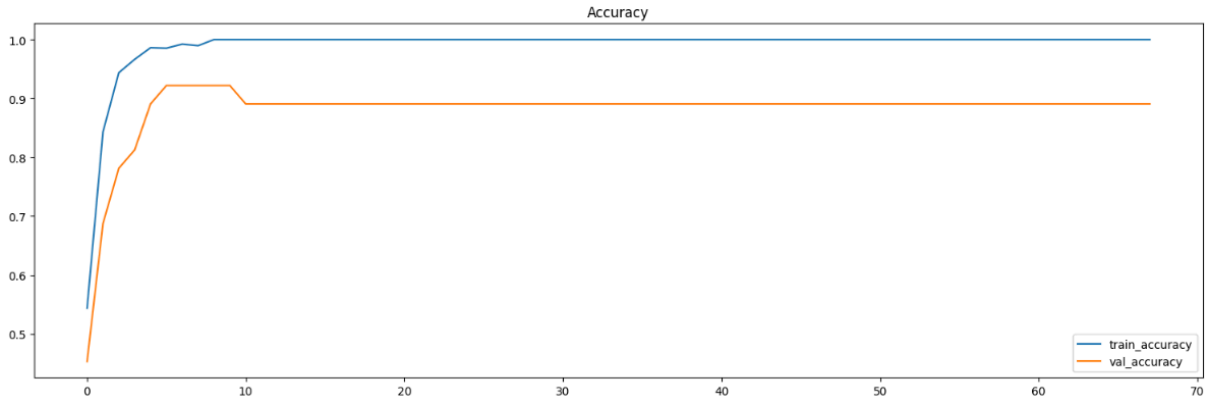
Image size

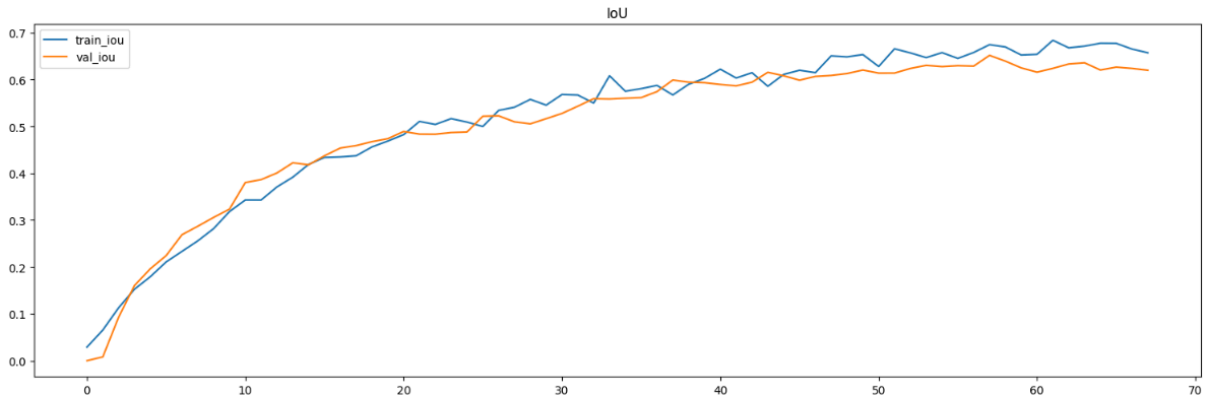
All test above where using an image size of 400 x 255 (width, height)

400 x 400

Epoch 58:
train_loss = 0.23008989625506931
train_cls_loss = 0.006851275606701772
train_reg_loss = 0.3257635964287652
train_iou = 0.6742445100564507
train_accuracy = 1.0
val_loss = 0.33894987404346466
val_cls_loss = 0.3164300471544266
val_reg_loss = 0.34860122203826904
val_iou = 0.6514102980381447
val_accuracy = 0.890625



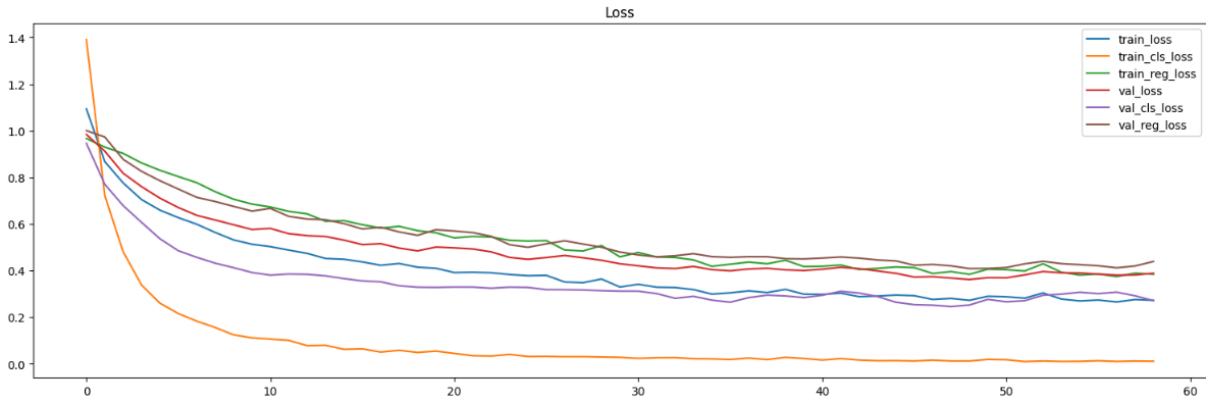

Accuracy
train_accuracy
val_accuracy

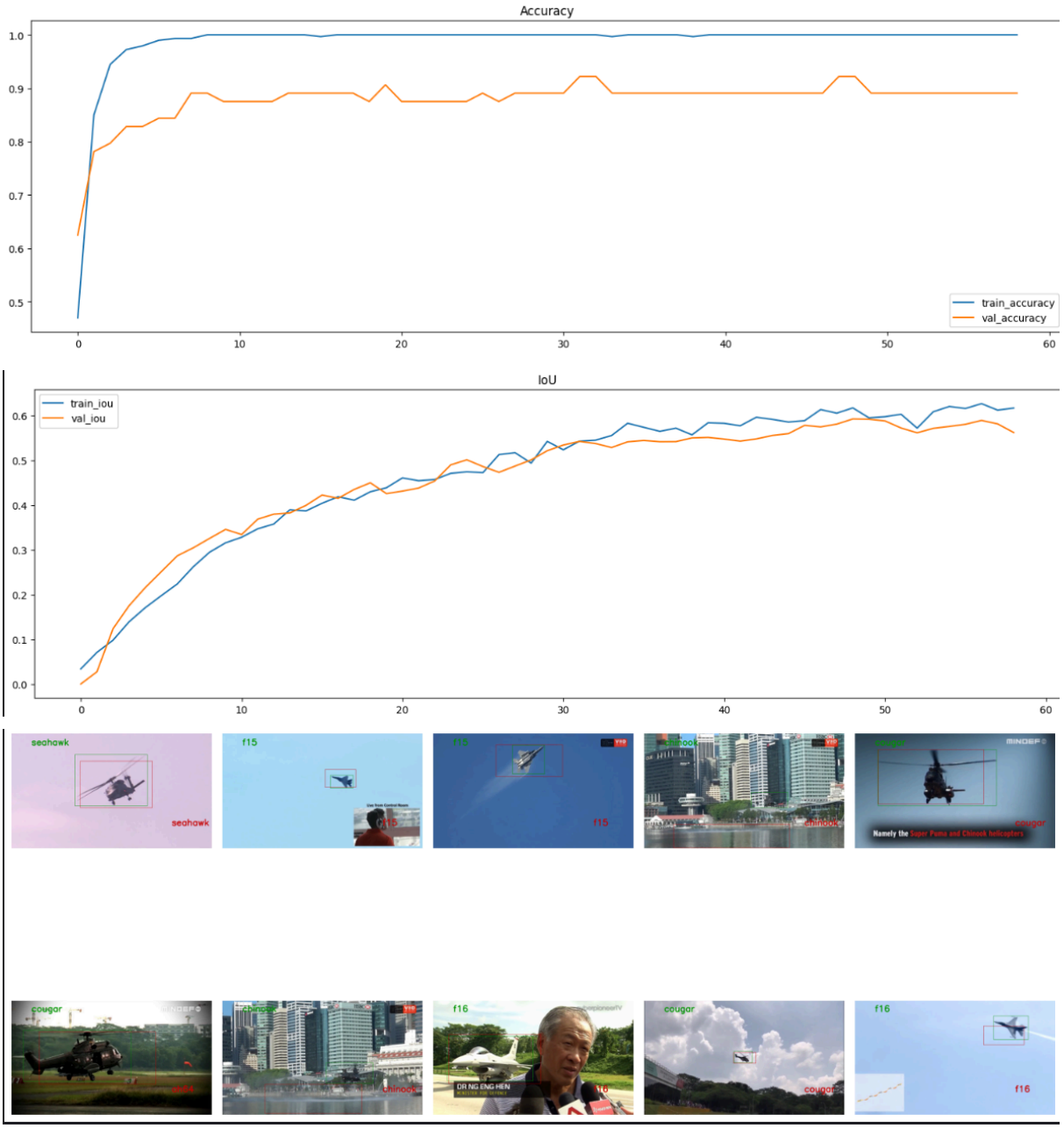

IoU
train_iou
val_iou

private: 0.75548
public: 0.76153

704 x 400

Epoch 49:
train_loss = 0.2715108212497499
train_cls_loss = 0.011027292089743746
train_reg_loss = 0.38314663039313424
train_iou = 0.6168600903825434
train_accuracy = 1.0
val_loss = 0.3608996421098709
val_cls_loss = 0.25127945840358734
val_reg_loss = 0.4078797399997711
val_iou = 0.592130670969311
val_accuracy = 0.921875



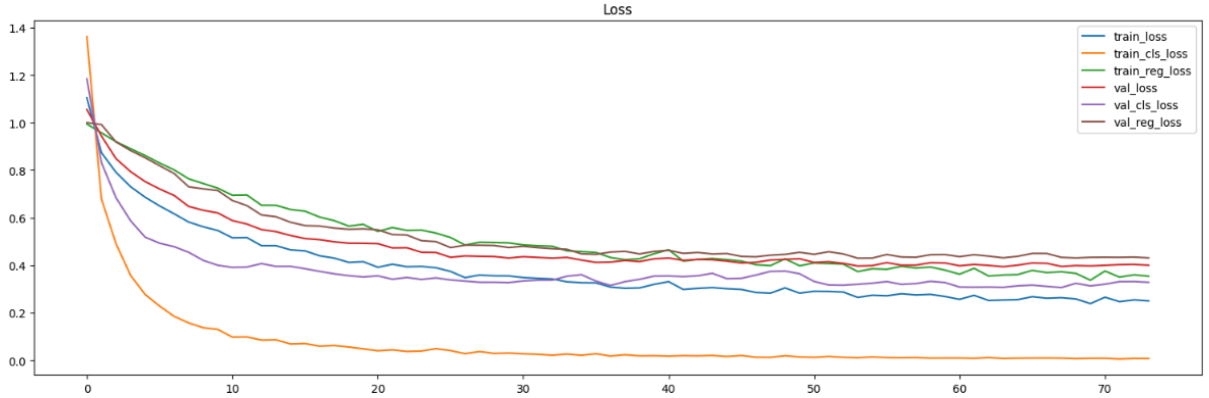


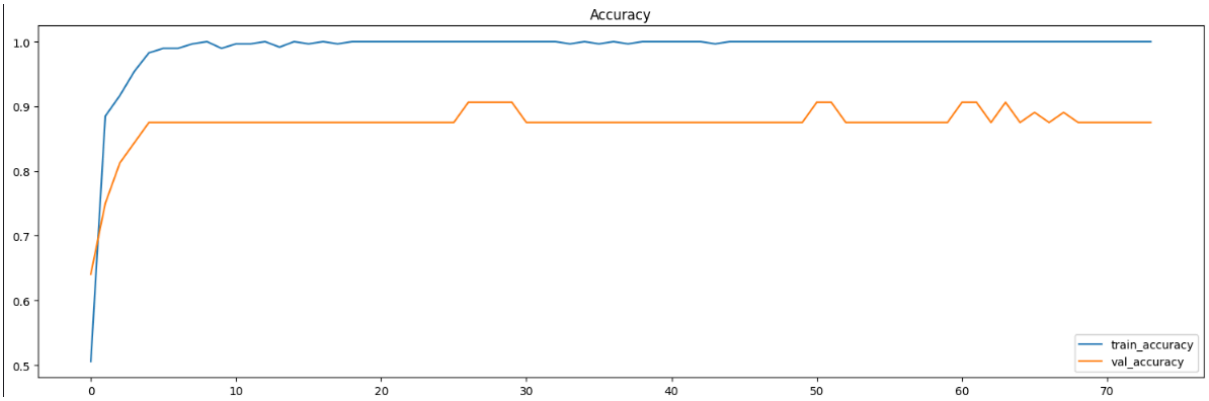
private: 0.72408
public: 0.75415

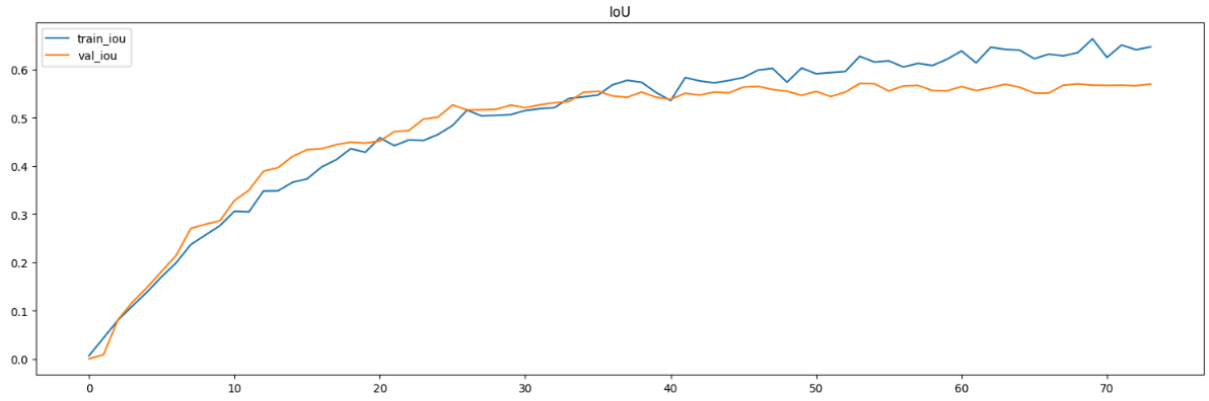
1152 x 648

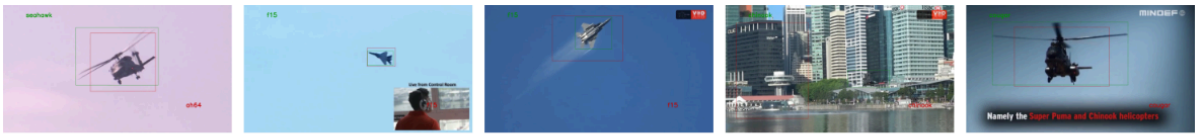
Epoch 64:
train_loss = 0.25323669446839225
train_cls_loss = 0.007249579195761018
train_reg_loss = 0.3586597508854336
train_iou = 0.6413478278468089
train_accuracy = 1.0
val_loss = 0.393685981631279
val_cls_loss = 0.30678198486566544
val_reg_loss = 0.43093055486679077

val_iou = 0.5690795502758554
val_accuracy = 0.90625





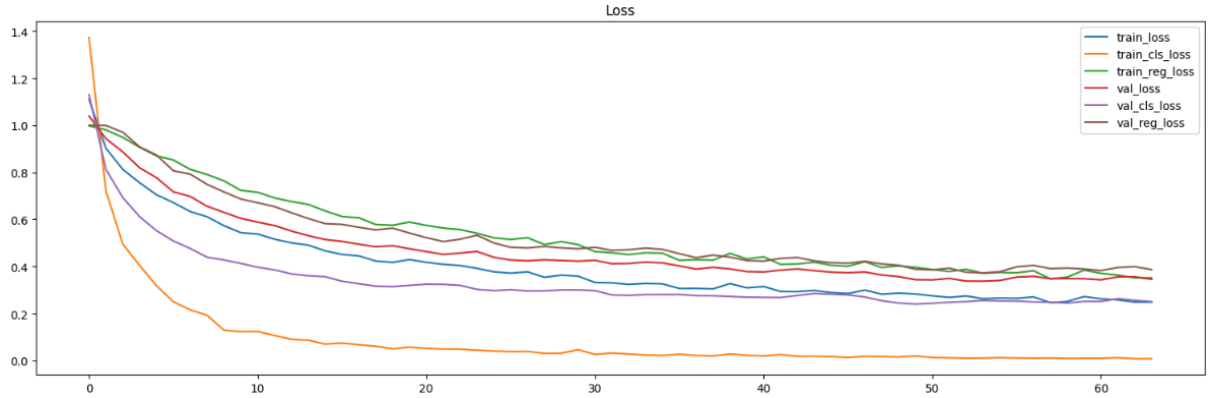


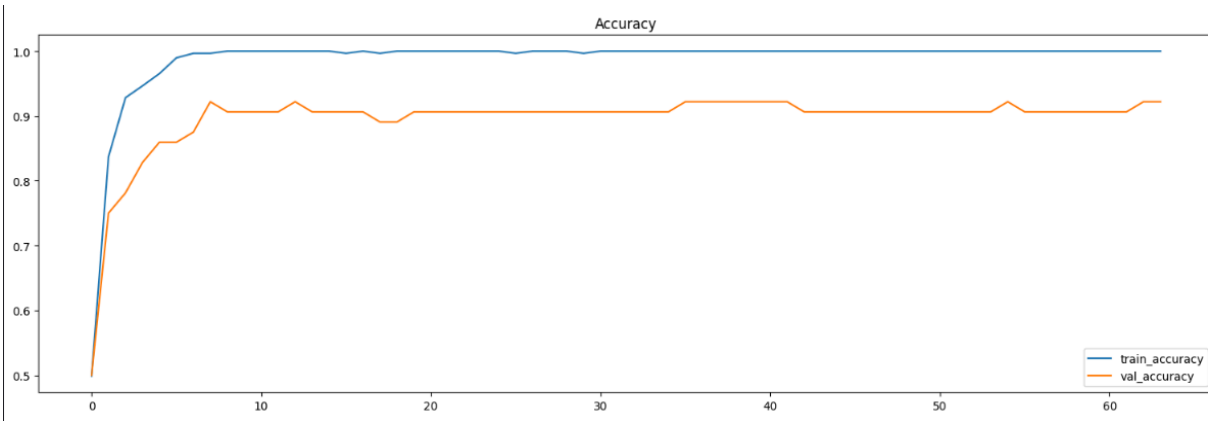


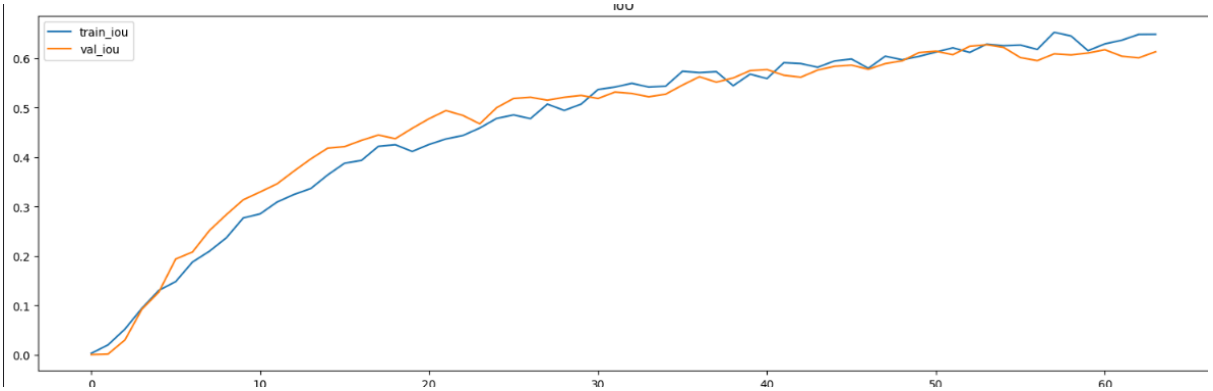
private: 0.72284
public: 0.74198

454 x 255

poch 54:
train_loss = 0.2631385127703349
train_cls_loss = 0.010770824065224992
train_reg_loss = 0.37129610776901245
train_iou = 0.6287111270927096
train_accuracy = 1.0
val_loss = 0.3374233990907669
val_cls_loss = 0.25565899908542633
val_reg_loss = 0.3724652826786041
val_iou = 0.627546108295726
val_accuracy = 0.90625



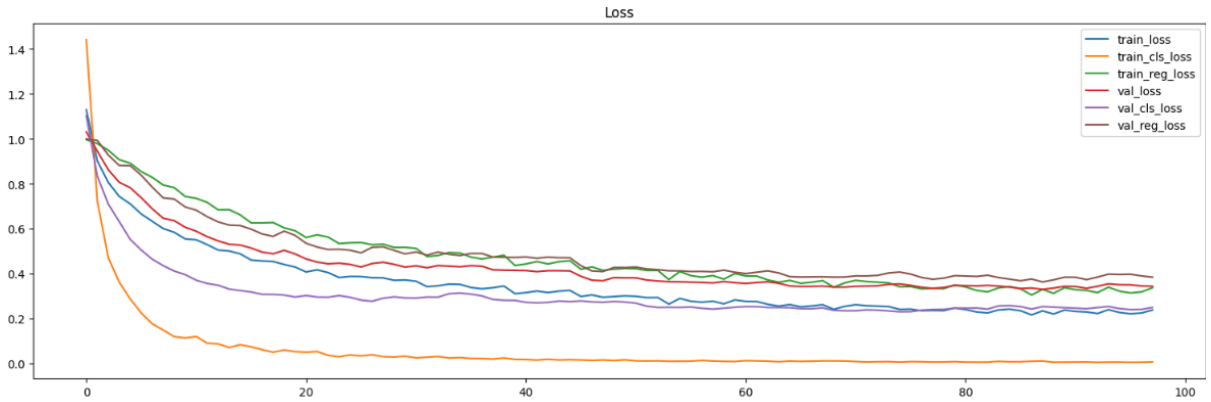


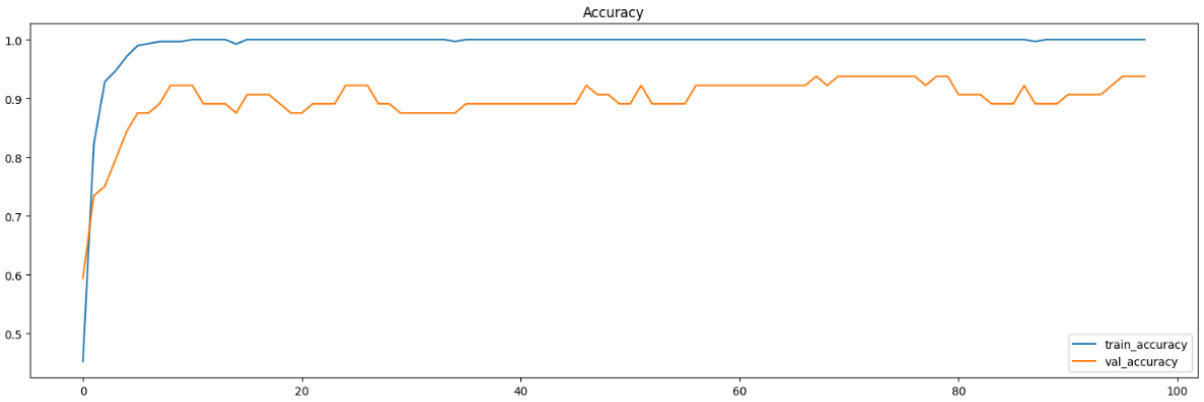


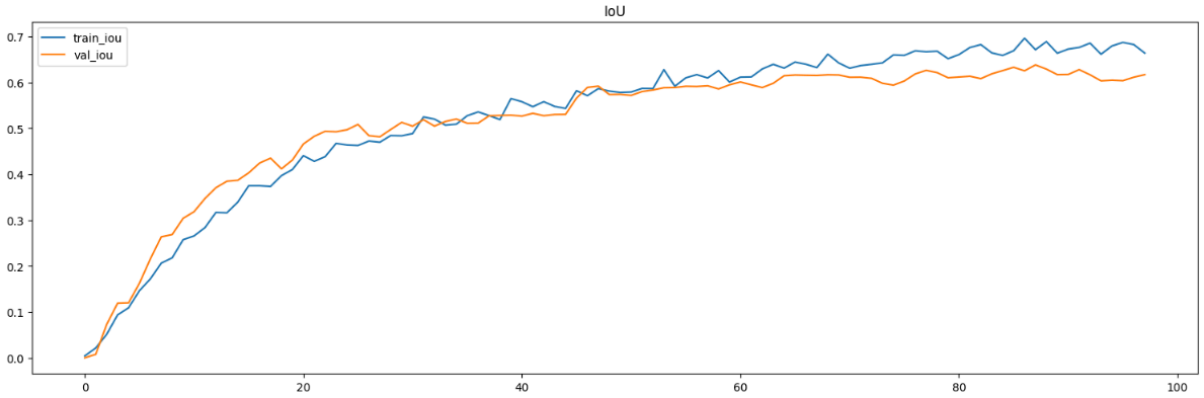
private: 0.74460
public: 0.76187

450 x 305

Epoch 88:
train_loss = 0.233169792426957
train_cls_loss = 0.009639435602972904
train_reg_loss = 0.328968518310123
train_iou = 0.6710390987393036
train_accuracy = 0.9965277777777778
val_loss = 0.32905131578445435
val_cls_loss = 0.2521180212497711
val_reg_loss = 0.3620227575302124
val_iou = 0.6379870154874653
val_accuracy = 0.890625





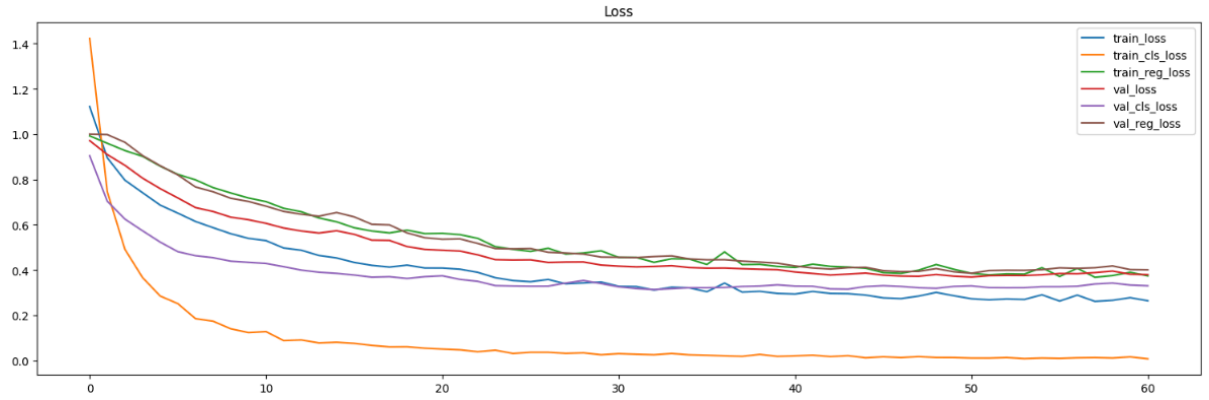


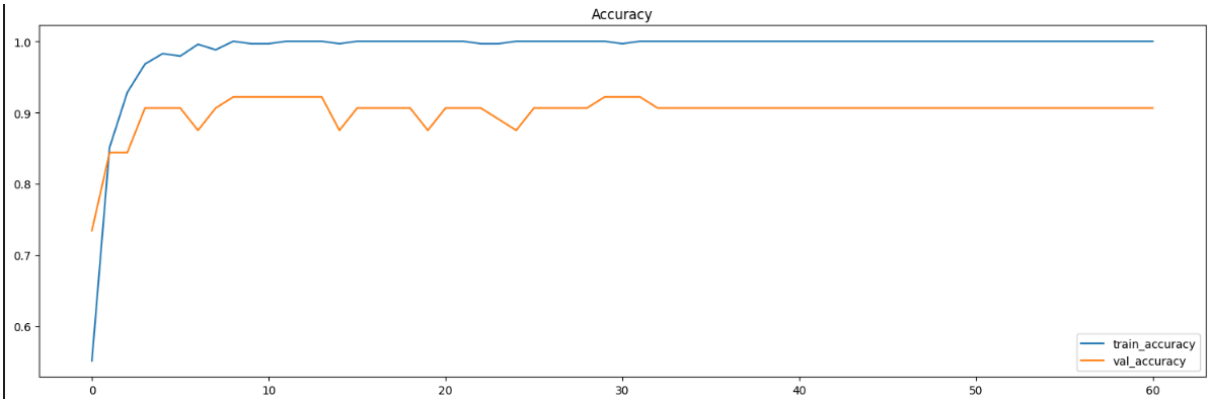
private: 0.75088
public: 0.74095

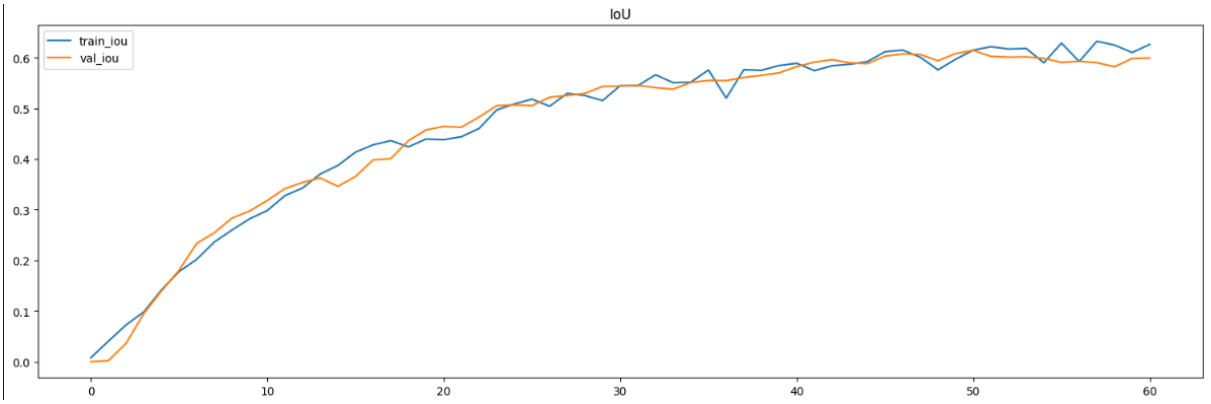
350 x 205

Epoch 51:
train_loss = 0.2730223039786021
train_cls_loss = 0.01116397174903088
train_reg_loss = 0.38524731000264484
train_iou = 0.6147594673315268
train_accuracy = 1.0
val_loss = 0.3689144104719162
val_cls_loss = 0.33004067838191986
val_reg_loss = 0.3855745792388916

val_iou = 0.6144362721445443
val_accuracy = 0.90625







private 0.73392
public 0.720223

Best size

Even if generally a bigger image size helps models better generalize, specially little objects, it also requires a bigger amount of data. Hence increasing the image size ends in worst results since we have only 141 training images which we are duplicating with augmentation and 48 validation images. This dataset is not long enough to allow increased size without risking overfitting.
Thanks to choosing a relatively small size the compute power required is little, with a batch of 32 and images of 400 x 255 we use less than 3GB ram and 2GB vram. So another good option to accelerate the process is to increase the batch size

We decided to continue with 400 x 255. 0.78071 score

Batch size

32

epoch 56
1.5 GB vram
2.5 GB ram
score 0.78

64

epoch 99
+1 GB vram
score 0.74

128

epoch 128
4.6GB vram
score 0.75379

16

epoch 61
1.2 GB vram
score 0.735

Conclusion

The batch size of 32 find the best scores, another combination of hyperparams could be check but at the end we find our best with 32

Final Model

Backbone

squezeenet v1.0
all freeze

Classification Head Architecture

```
self.cls_head = nn.Sequential(
            nn.Linear(flattened_features, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, n_classes)
        )
```

Regresion Head Architecture

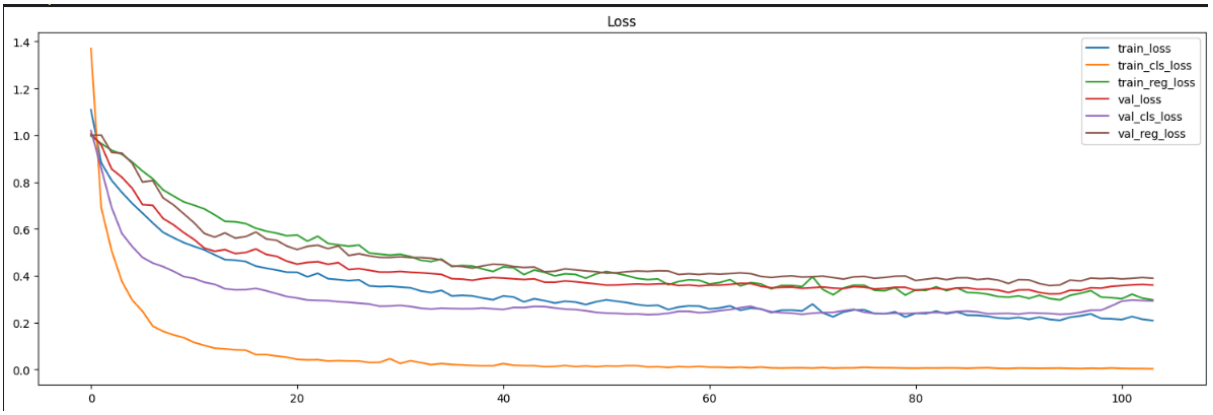
```
        self.reg_head = nn.Sequential(
            nn.Linear(flattened_features, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 768),
            nn.BatchNorm1d(768),
            nn.ReLU(),
            nn.Linear(768, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Linear(128, 4)
        )
```

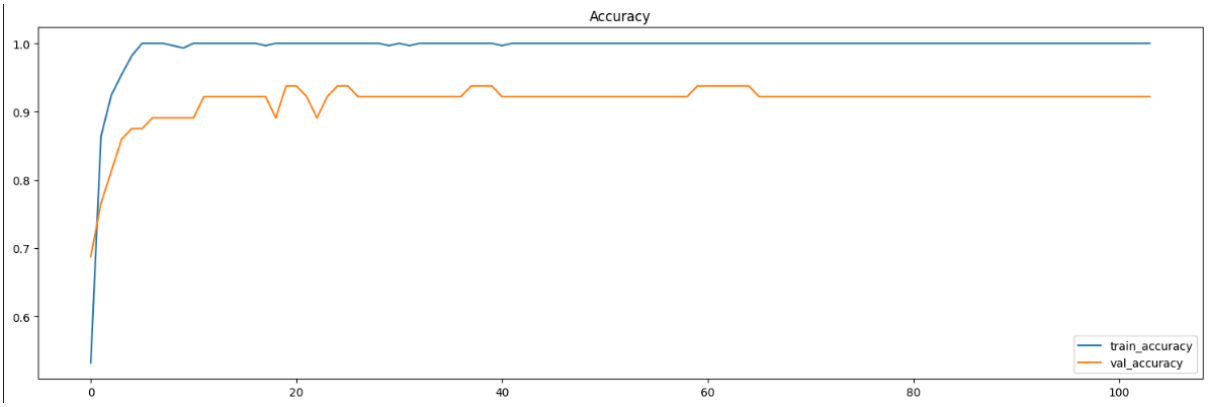
Hyperparams

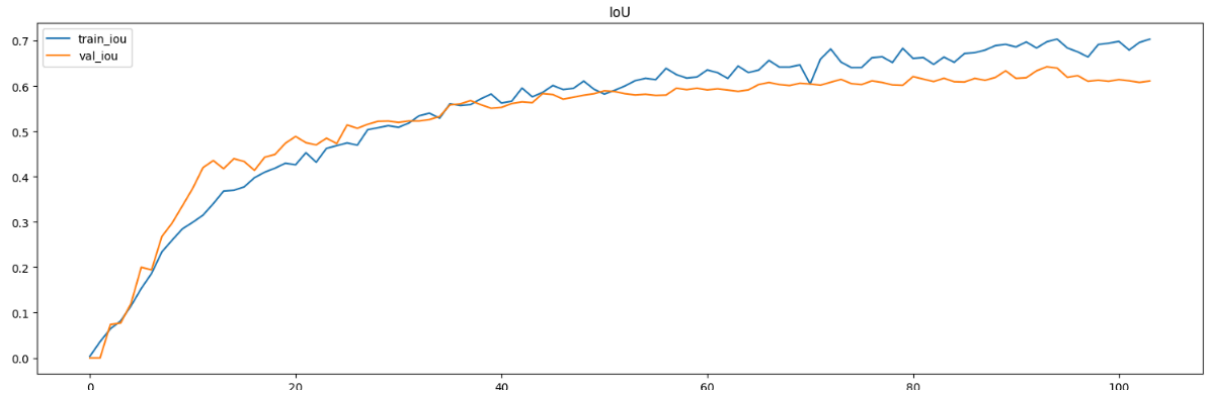
Image size	400 x 255
Batch size	32
patience	10
Alpha	0.7

Learning rate	1e-4
L2_lambda	1e-6
optimizer	Adam
random_state	42

Plots







Scores

Epoch 94:
train_loss = 0.21331014898088244

train_cls_loss = 0.004949078688191043
train_reg_loss = 0.30260774824354386
train_iou = 0.6974008853205423
train_accuracy = 1.0
val_loss = 0.32194332778453827
val_cls_loss = 0.2383575215935707
val_reg_loss = 0.3577658236026764
val_iou = 0.6422453311397442
val_accuracy = 0.921875

private: 0.78071
public: 0.74559

Sample images

Summary

from torchsummary import summary

summary(trained_model,(3,255,400))

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 96, 125, 197]          14,208
              ReLU-2         [-1, 96, 125, 197]               0
         MaxPool2d-3          [-1, 96, 62, 98]               0
            Conv2d-4          [-1, 16, 62, 98]           1,552
              ReLU-5          [-1, 16, 62, 98]               0
            Conv2d-6          [-1, 64, 62, 98]           1,088
              ReLU-7          [-1, 64, 62, 98]               0
            Conv2d-8          [-1, 64, 62, 98]           9,280
              ReLU-9          [-1, 64, 62, 98]               0
```

Fire-10	[-1, 128, 62, 98]	0
Conv2d-11	[-1, 16, 62, 98]	2,064
ReLU-12	[-1, 16, 62, 98]	0
Conv2d-13	[-1, 64, 62, 98]	1,088
ReLU-14	[-1, 64, 62, 98]	0
Conv2d-15	[-1, 64, 62, 98]	9,280
ReLU-16	[-1, 64, 62, 98]	0
Fire-17	[-1, 128, 62, 98]	0
Conv2d-18	[-1, 32, 62, 98]	4,128
ReLU-19	[-1, 32, 62, 98]	0
Conv2d-20	[-1, 128, 62, 98]	4,224
ReLU-21	[-1, 128, 62, 98]	0
Conv2d-22	[-1, 128, 62, 98]	36,992
ReLU-23	[-1, 128, 62, 98]	0
Fire-24	[-1, 256, 62, 98]	0
MaxPool2d-25	[-1, 256, 31, 49]	0
Conv2d-26	[-1, 32, 31, 49]	8,224
ReLU-27	[-1, 32, 31, 49]	0
Conv2d-28	[-1, 128, 31, 49]	4,224
ReLU-29	[-1, 128, 31, 49]	0
Conv2d-30	[-1, 128, 31, 49]	36,992
ReLU-31	[-1, 128, 31, 49]	0
Fire-32	[-1, 256, 31, 49]	0
Conv2d-33	[-1, 48, 31, 49]	12,336
ReLU-34	[-1, 48, 31, 49]	0
Conv2d-35	[-1, 192, 31, 49]	9,408
ReLU-36	[-1, 192, 31, 49]	0
Conv2d-37	[-1, 192, 31, 49]	83,136
ReLU-38	[-1, 192, 31, 49]	0
Fire-39	[-1, 384, 31, 49]	0
Conv2d-40	[-1, 48, 31, 49]	18,480
ReLU-41	[-1, 48, 31, 49]	0
Conv2d-42	[-1, 192, 31, 49]	9,408
ReLU-43	[-1, 192, 31, 49]	0
Conv2d-44	[-1, 192, 31, 49]	83,136
ReLU-45	[-1, 192, 31, 49]	0
Fire-46	[-1, 384, 31, 49]	0
Conv2d-47	[-1, 64, 31, 49]	24,640
ReLU-48	[-1, 64, 31, 49]	0
Conv2d-49	[-1, 256, 31, 49]	16,640
ReLU-50	[-1, 256, 31, 49]	0
Conv2d-51	[-1, 256, 31, 49]	147,712
ReLU-52	[-1, 256, 31, 49]	0
Fire-53	[-1, 512, 31, 49]	0
MaxPool2d-54	[-1, 512, 15, 24]	0
Conv2d-55	[-1, 64, 15, 24]	32,832
ReLU-56	[-1, 64, 15, 24]	0
Conv2d-57	[-1, 256, 15, 24]	16,640

```
              ReLU-58          [-1, 256, 15, 24]               0
            Conv2d-59          [-1, 256, 15, 24]         147,712
              ReLU-60          [-1, 256, 15, 24]               0
              Fire-61          [-1, 512, 15, 24]               0
AdaptiveAvgPool2d-62            [-1, 512, 7, 7]               0
           Flatten-63                [-1, 25088]               0
           Dropout-64                [-1, 25088]               0
 FeatureExtractor-65                [-1, 25088]               0
           Linear-66                 [-1, 256]       6,422,784
      BatchNorm1d-67                 [-1, 256]             512
             ReLU-68                 [-1, 256]               0
           Linear-69                   [-1, 6]           1,542
           Linear-70                 [-1, 768]      19,268,352
      BatchNorm1d-71                 [-1, 768]           1,536
             ReLU-72                 [-1, 768]               0
           Linear-73                 [-1, 768]         590,592
      BatchNorm1d-74                 [-1, 768]           1,536
             ReLU-75                 [-1, 768]               0
           Linear-76                 [-1, 256]         196,864
      BatchNorm1d-77                 [-1, 256]             512
             ReLU-78                 [-1, 256]               0
           Linear-79                 [-1, 256]          65,792
      BatchNorm1d-80                 [-1, 256]             512
             ReLU-81                 [-1, 256]               0
           Linear-82                 [-1, 128]          32,896
      BatchNorm1d-83                 [-1, 128]             256
             ReLU-84                 [-1, 128]               0
           Linear-85                   [-1, 4]             516
================================================================
Total params: 27,319,626
Trainable params: 26,584,202
Non-trainable params: 735,424
----------------------------------------------------------------
Input size (MB): 1.17
Forward/backward pass size (MB): 185.28
Params size (MB): 104.22
Estimated Total Size (MB): 290.66
----------------------------------------------------------------
```

Use CNNs for heads

Reg head

Here it’s another experiment changing the technique to predict the bounding box, instead of using linear layers the head continue to use the cnn.

Base architecture

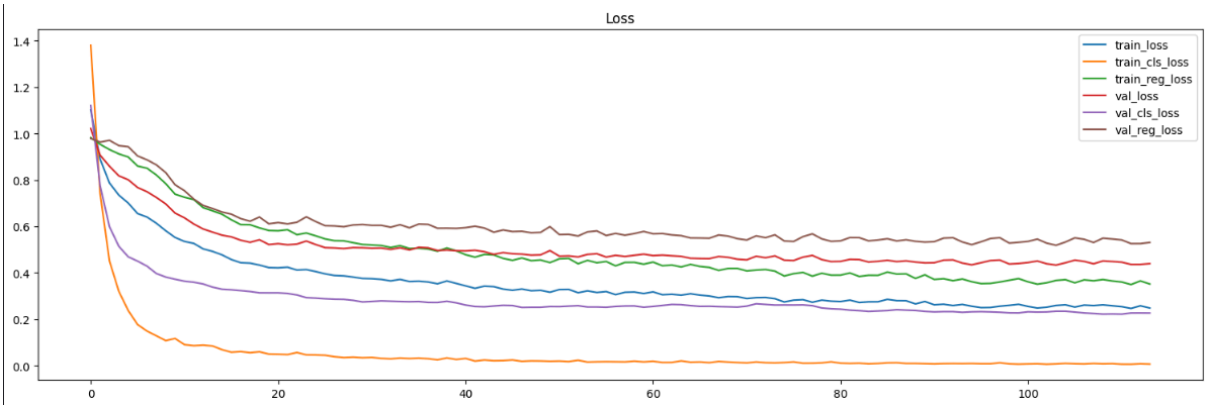
```
        self.reg_head = nn.Sequential(
            nn.Conv2d(in_channels=backbone_out_shape[1], out_channels=256, kernel_size=3,
padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=4, kernel_size=1),
            nn.AdaptiveMaxPool2d((1, 1))
        )
```

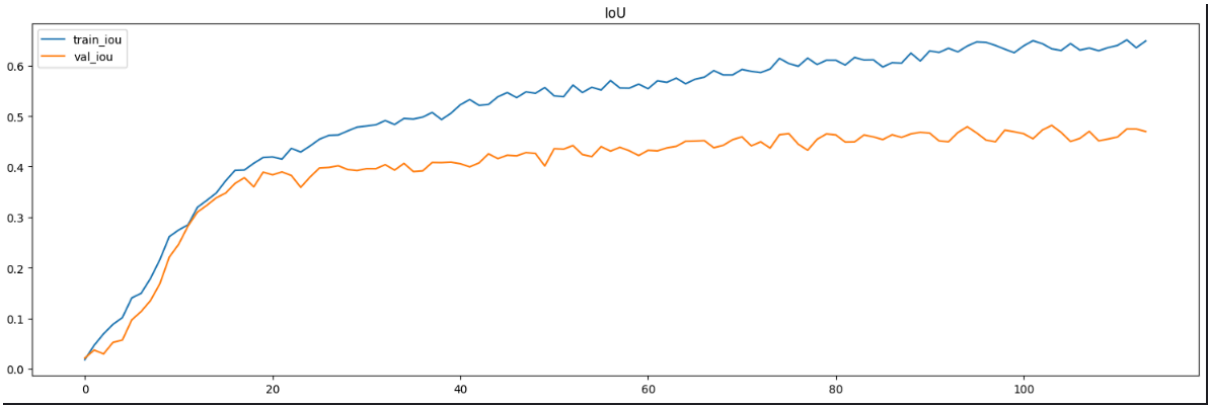
The architecture use a block consisting of a convolutional layer with a kernel size 3x3 and 256 filters, followed by a BatchNorm to avoid overfitting. This block is repeated multiple times to experiment how deepnes improves prediction.

To generate the box prediction the last convolutional layer use a kernel size 1x1 and 4 filters, where each filter corresponds to a point of the box. The adaptavive max pool is to ensure the output is of shape (batch,boxing_points) since the actual shape is (batch, last_filter, w, h)

1 Block

Epoch 104:
train_loss = 0.2590208401282628
train_cls_loss = 0.007194185784707467
train_reg_loss = 0.36694655815760296
train_iou = 0.6330601411344499
train_accuracy = 1.0
val_loss = 0.43313197791576385
val_cls_loss = 0.23419708758592606
val_reg_loss = 0.5183897912502289
val_iou = 0.4816164875061332
val_accuracy = 0.921875



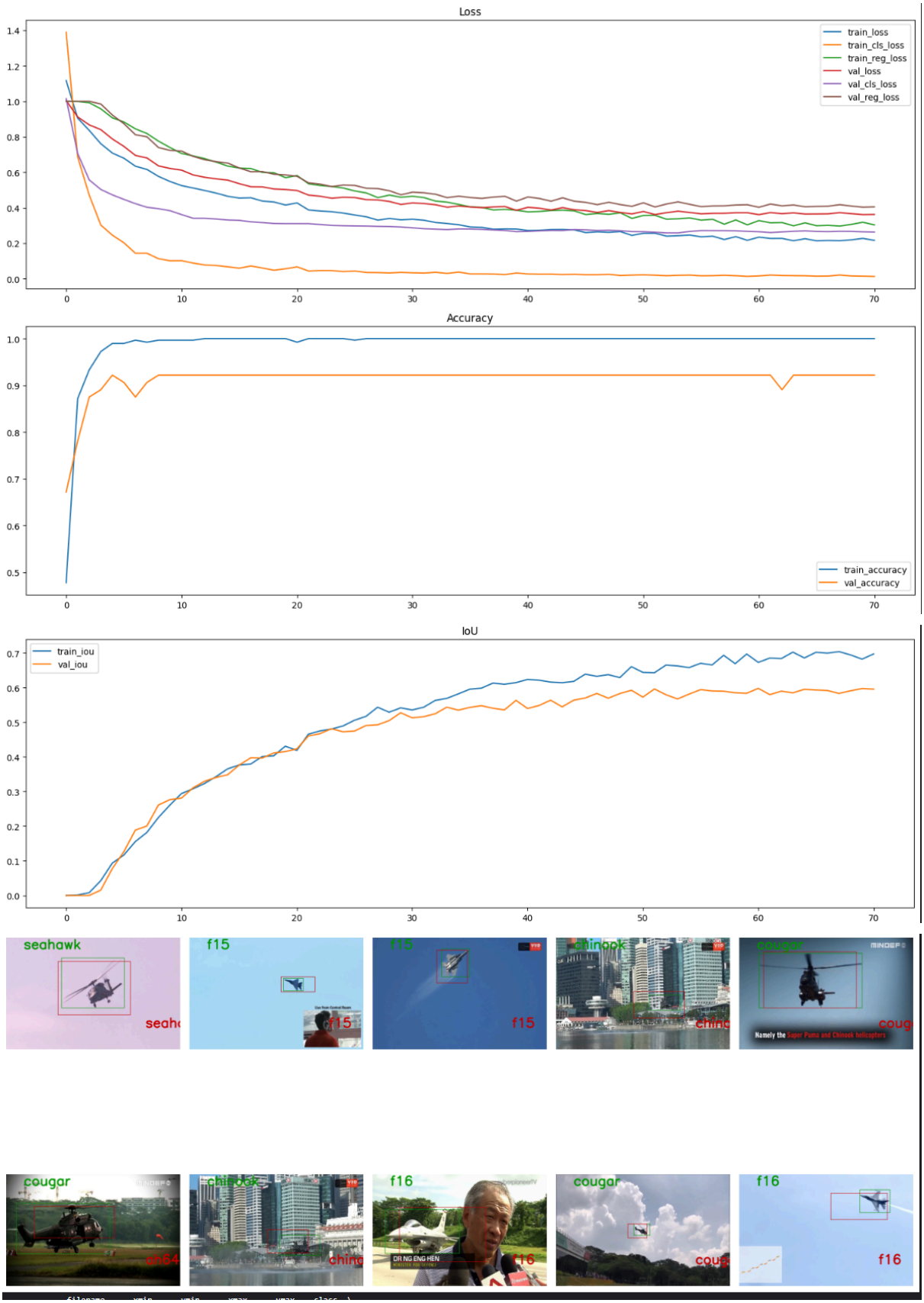


Total params: 8,341,706
Trainable params: 7,606,282
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 185.51
Params size (MB): 31.82
Estimated Total Size (MB): 218.50
score: 0.66385

2 Blocks

Epoch 61:
train_loss = 0.23395096924569872
train_cls_loss = 0.015576081867847178
train_reg_loss = 0.32754020558463204
train_iou = 0.6724677812175899
train_accuracy = 1.0
val_loss = 0.36106714606285095
val_cls_loss = 0.2645612806081772
val_reg_loss = 0.4024268090724945
val_iou = 0.5975820528499678
val_accuracy = 0.921875

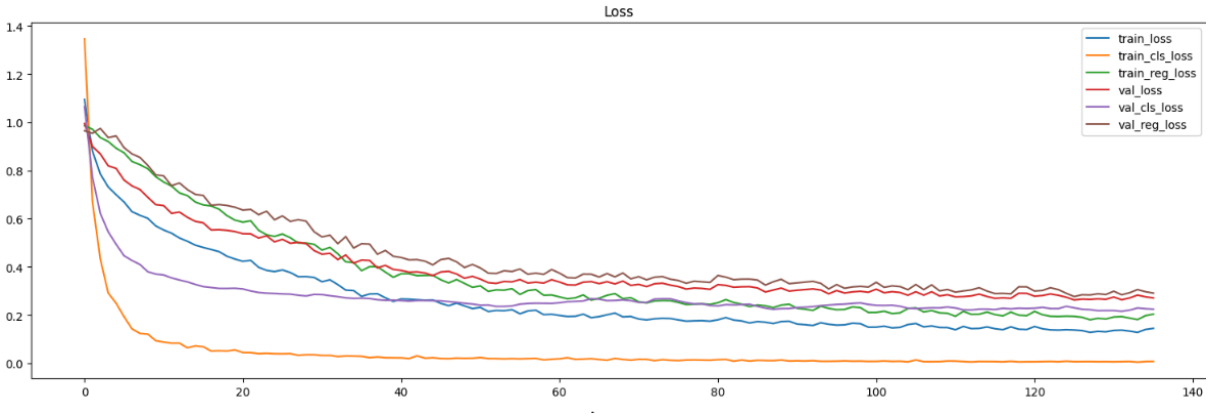


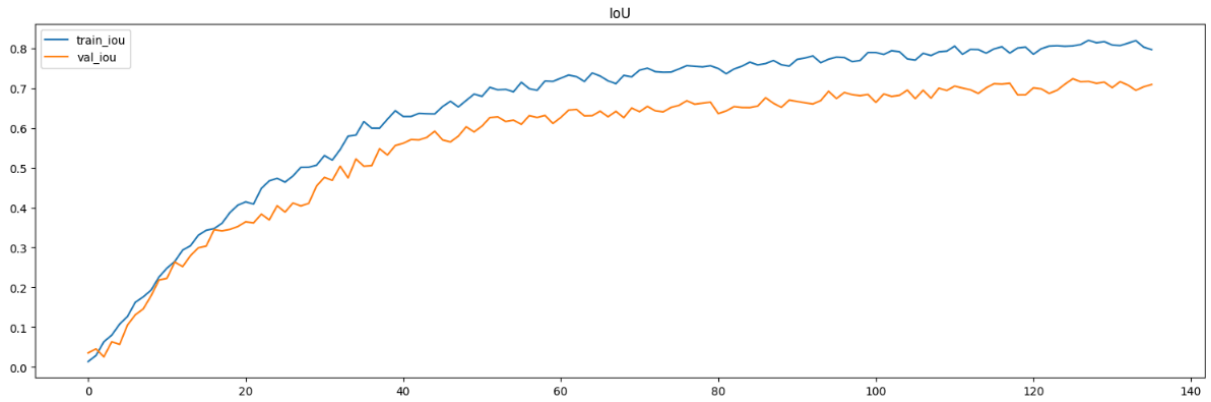
Params size (MB): 34.07
Estimated Total Size (MB): 221.04

score: 0.73776

3 Blocks

Epoch 126:
train_loss = 0.13772613720761406
train_cls_loss = 0.006101946760382917
train_reg_loss = 0.19413650698131985
train_iou = 0.8058752112954713
train_accuracy = 1.0
val_loss = 0.2624814212322235
val_cls_loss = 0.22986901551485062
val_reg_loss = 0.27645814418792725
val_iou = 0.7235566881214364
val_accuracy = 0.921875





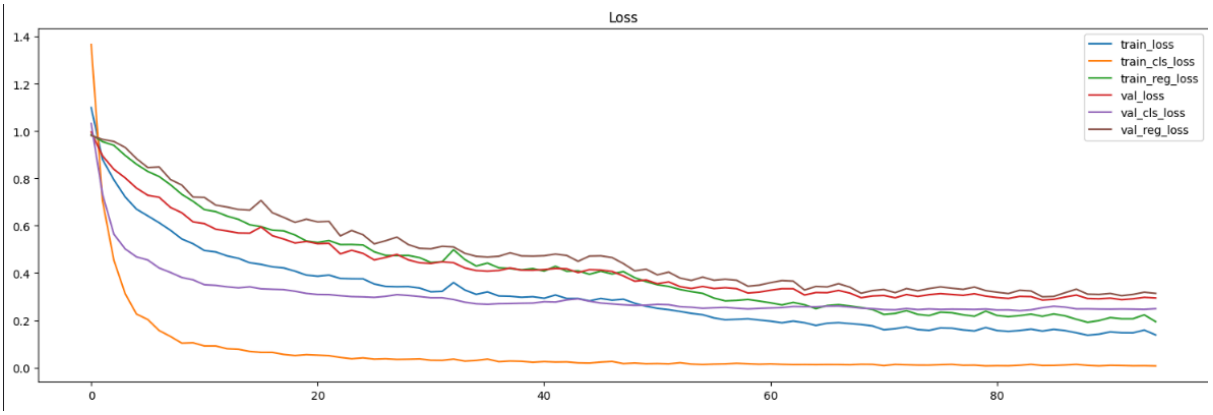
Total params: 9,522,890
Trainable params: 8,787,466

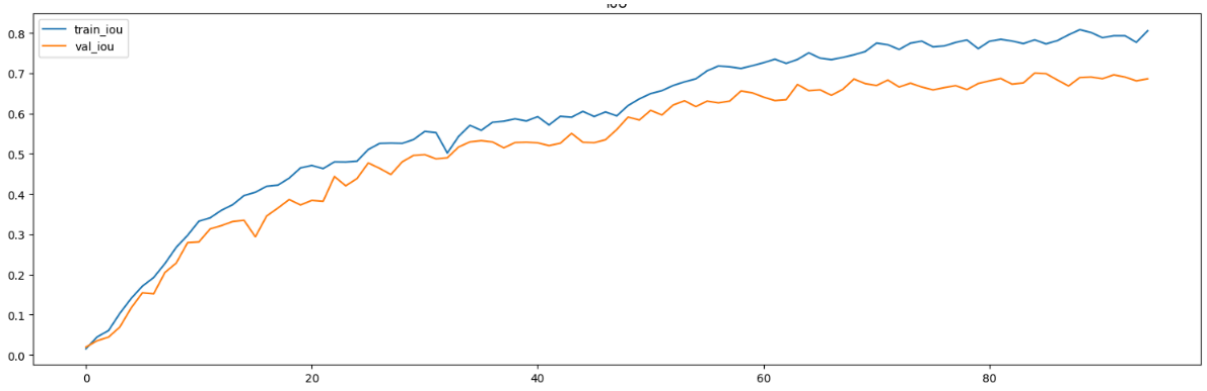
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 186.09
Params size (MB): 36.33
Estimated Total Size (MB): 223.58
score: 0.78224

4 Blocks

Epoch 85:
train_loss = 0.1546532760063807
train_cls_loss = 0.009355169927908314
train_reg_loss = 0.21692389912075466
train_iou = 0.7830867310258811
train_accuracy = 1.0
val_loss = 0.2855956703424454
val_cls_loss = 0.25316083431243896
val_reg_loss = 0.2994963228702545
val_iou = 0.7005183803664591
val_accuracy = 0.921875





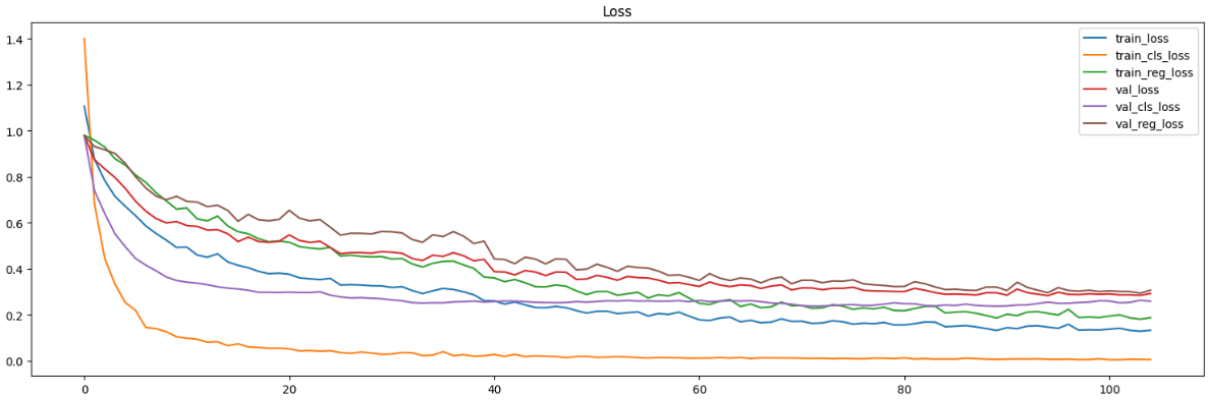
Total params: 10,113,482
Trainable params: 9,378,058
Non-trainable params: 735,424

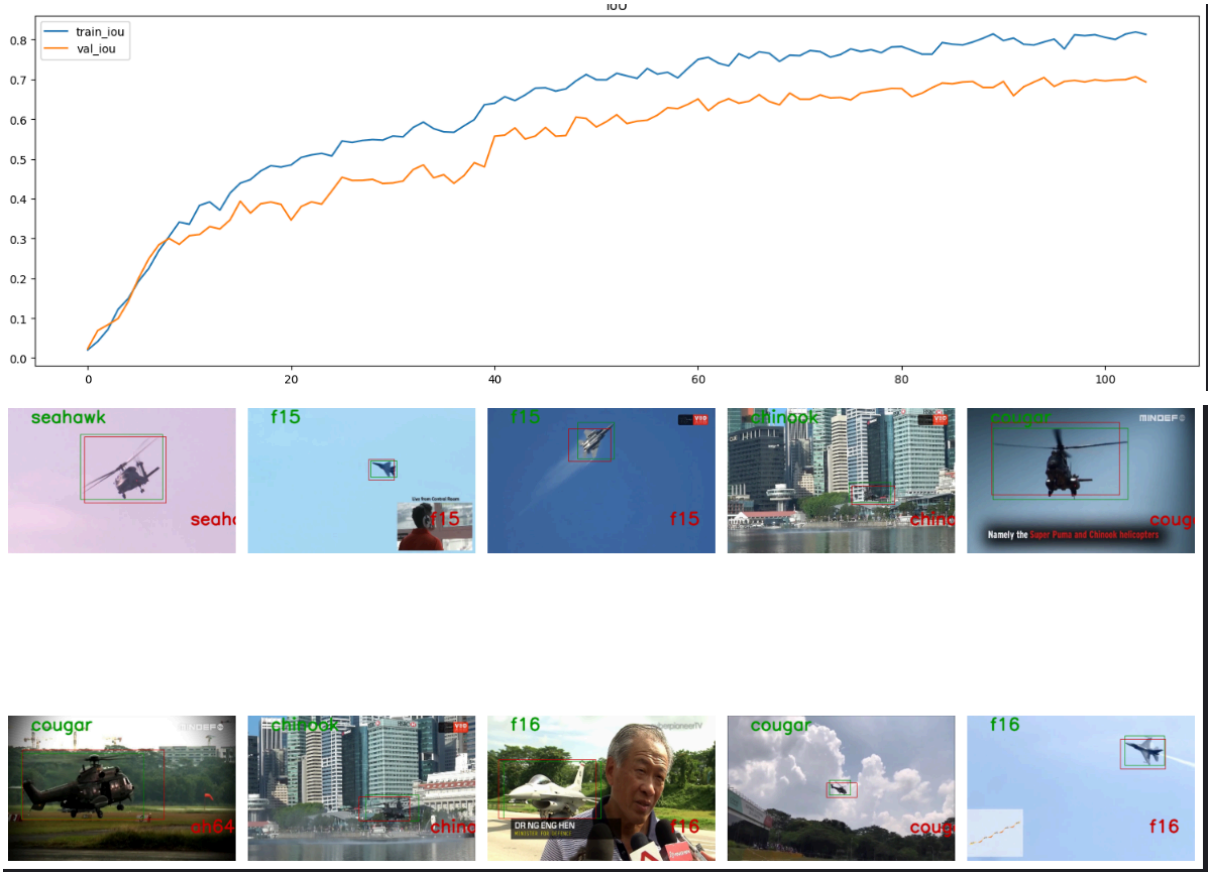
Input size (MB): 1.17
Forward/backward pass size (MB): 186.38
Params size (MB): 38.58
Estimated Total Size (MB): 226.12

Score: 0.79679

5 Blocks

Epoch 95:
train_loss = 0.1466317797700564
train_cls_loss = 0.007317933150463634
train_reg_loss = 0.2063377168443468
train_iou = 0.793673049654173
train_accuracy = 1.0
val_loss = 0.28362827003002167
val_cls_loss = 0.2551056370139122
val_reg_loss = 0.29585227370262146
val_iou = 0.7041608404133317
val_accuracy = 0.921875





Total params: 10,704,074
Trainable params: 9,968,650
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 186.66
Params size (MB): 40.83
Estimated Total Size (MB): 228.66

score: 0.77238

Conclusion

Both 3 and four blocks are good and not so different, if a small model is needed it could work with 3 blocks but in our case we continue with the 4 block option.

Clas head

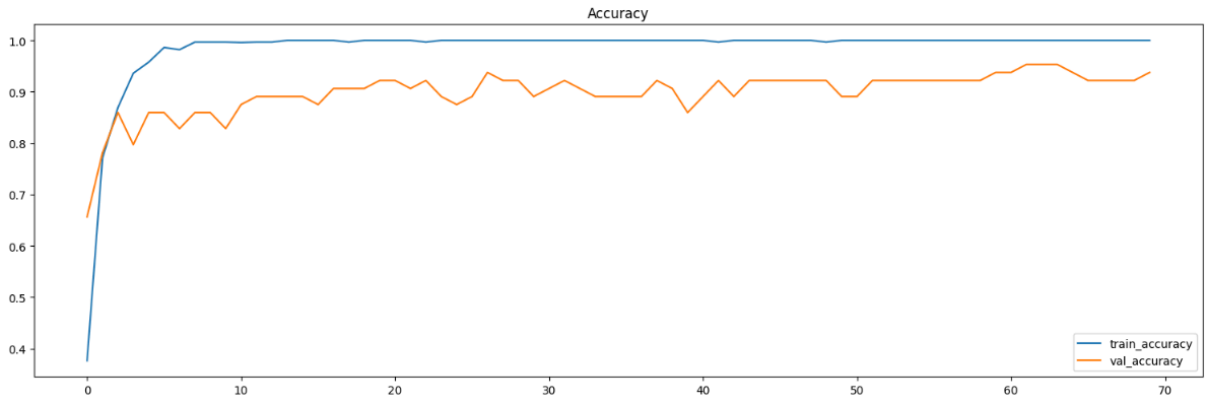
Similar to the architecture for regression head, this approach focuses on keeping image dimensionality to generate the class prediction. With this architecture it is posible to reduce the parameters of the model without sacrificing accuracy.

Base architecture

```
        self.cls_head = nn.Sequential(
            nn.Conv2d(in_channels=backbone_out_shape[1], out_channels=256, kernel_size=3,
padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=4, kernel_size=1),
            nn.AdaptiveMaxPool2d((1, 1))
        )
```

1 Block

Epoch 60:
train_loss = 0.205813929438591
train_cls_loss = 0.0071896314103570245
train_reg_loss = 0.2909386356671651
train_iou = 0.7090688854021671
train_accuracy = 1.0
val_loss = 0.3342360854148865
val_cls_loss = 0.3135729283094406
val_reg_loss = 0.3430917263031006
val_iou = 0.6569179920753685
val_accuracy = 0.9375



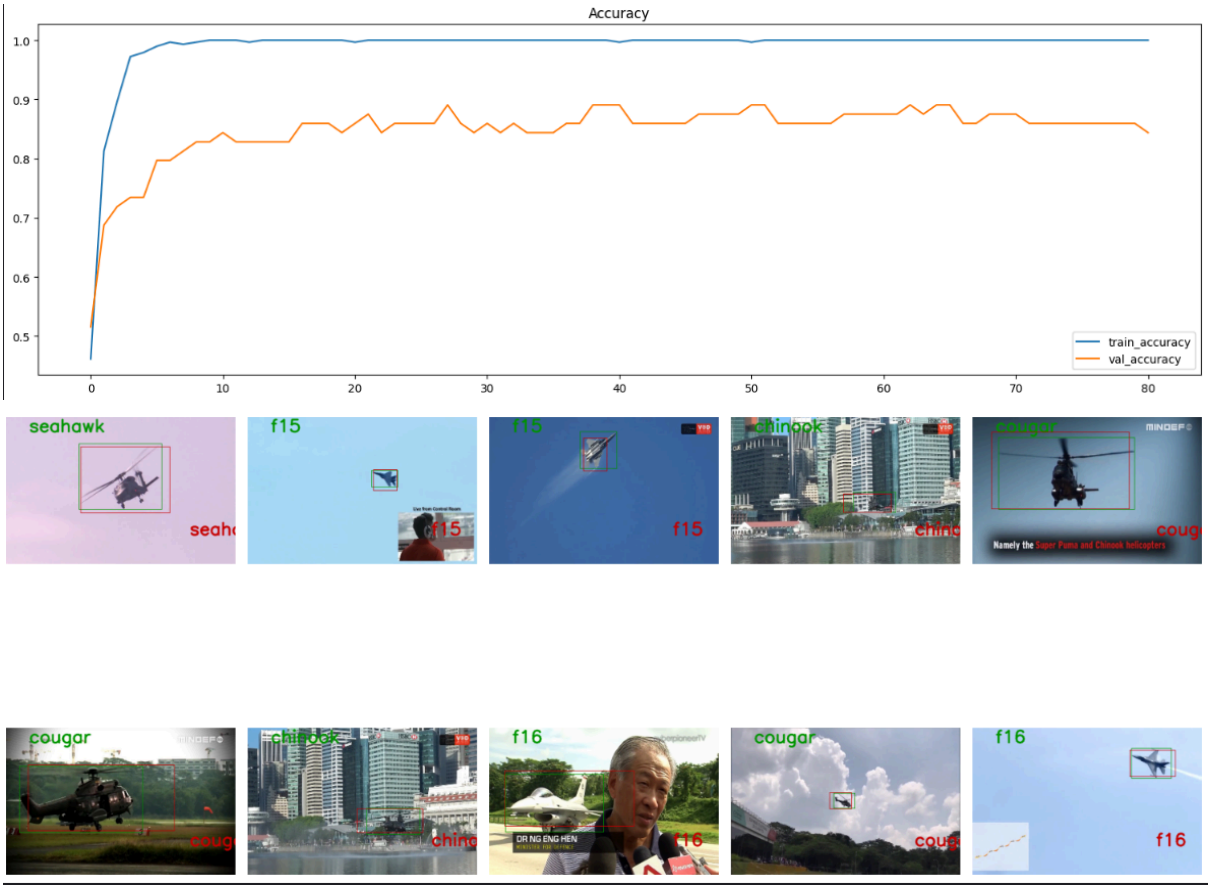
Total params: 4,870,602

Trainable params: 4,135,178
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 186.28
Params size (MB): 18.58
Estimated Total Size (MB): 206.02
score: 0.80

2 Blocks

Epoch 71:
train_loss = 0.16009832670291266
train_cls_loss = 0.0009302674120085107
train_reg_loss = 0.22831321424908108
train_iou = 0.7716978816622125
train_accuracy = 1.0
val_loss = 0.313789039850235
val_cls_loss = 0.37462159991264343
val_reg_loss = 0.28771793842315674
val_iou = 0.7122975853720891
val_accuracy = 0.875

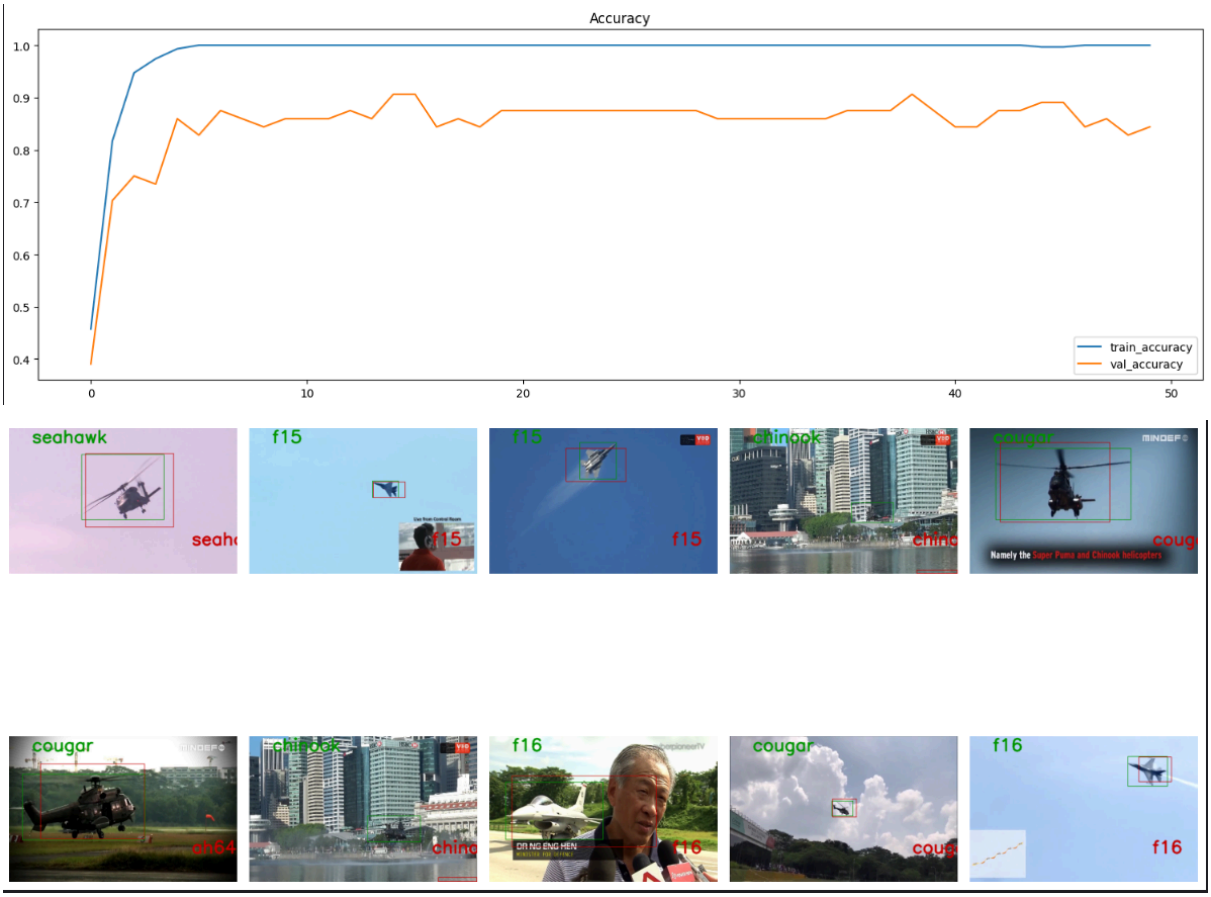


Total params: 5,461,194
Trainable params: 4,725,770
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 186.56
Params size (MB): 20.83
Estimated Total Size (MB): 208.56
score: 0.77644

3 Blocks

Epoch 40:
train_loss = 0.24464869995911917
train_cls_loss = 0.0016311142438401778
train_reg_loss = 0.34879910945892334
train_iou = 0.6512089942364528
train_accuracy = 1.0
val_loss = 0.41863566637039185
val_cls_loss = 0.4327888935804367
val_reg_loss = 0.4125699996948242
val_iou = 0.5874400770034446
val_accuracy = 0.875



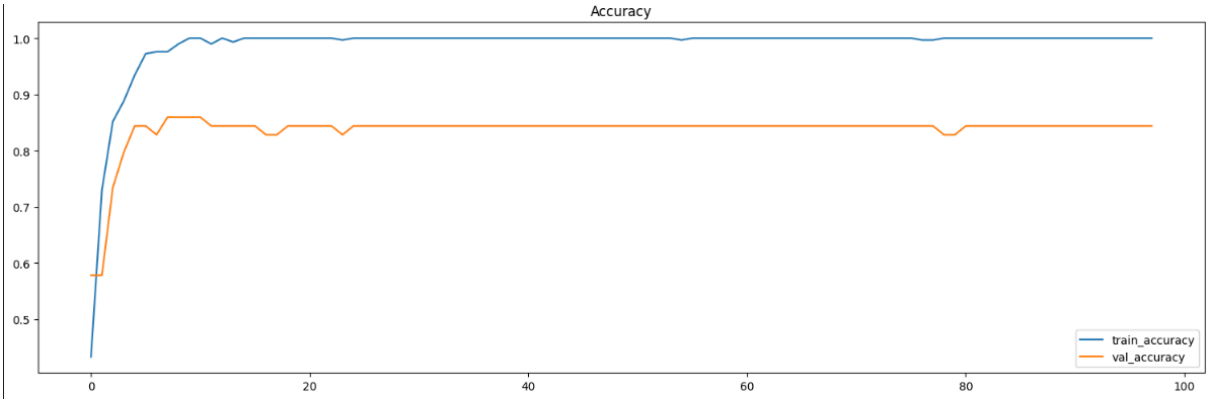
Total params: 6,051,786
Trainable params: 5,316,362
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 186.85
Params size (MB): 23.09

Estimated Total Size (MB): 211.10
score: 0.76485

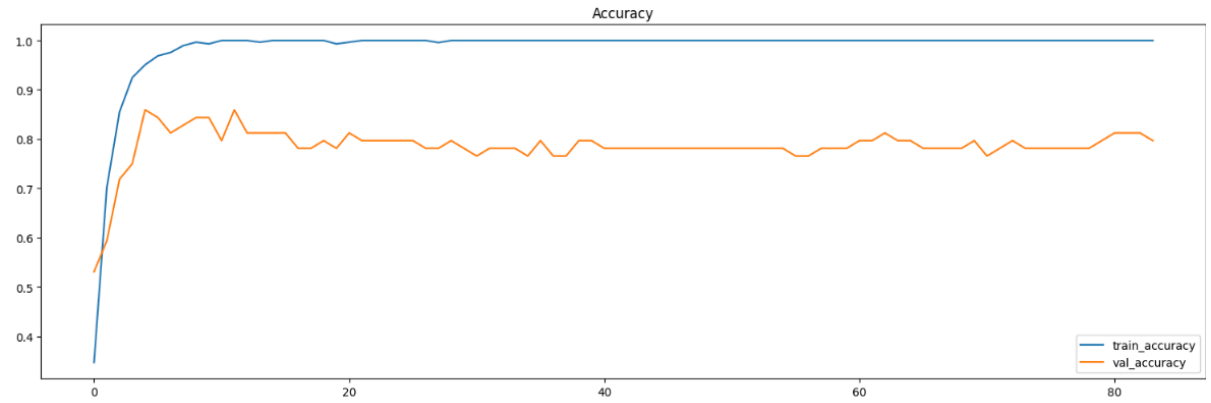
1 Block with 128 filters

Epoch 88:
train_loss = 0.15708311647176743
train_cls_loss = 0.008285859987760583
train_reg_loss = 0.22085337506400216
train_iou = 0.7791569346181411
train_accuracy = 1.0
val_loss = 0.2741304636001587
val_cls_loss = 0.26514752209186554
val_reg_loss = 0.2779802978038788
val_iou = 0.7220343064986626
val_accuracy = 0.84375



2 Block with 128 filters

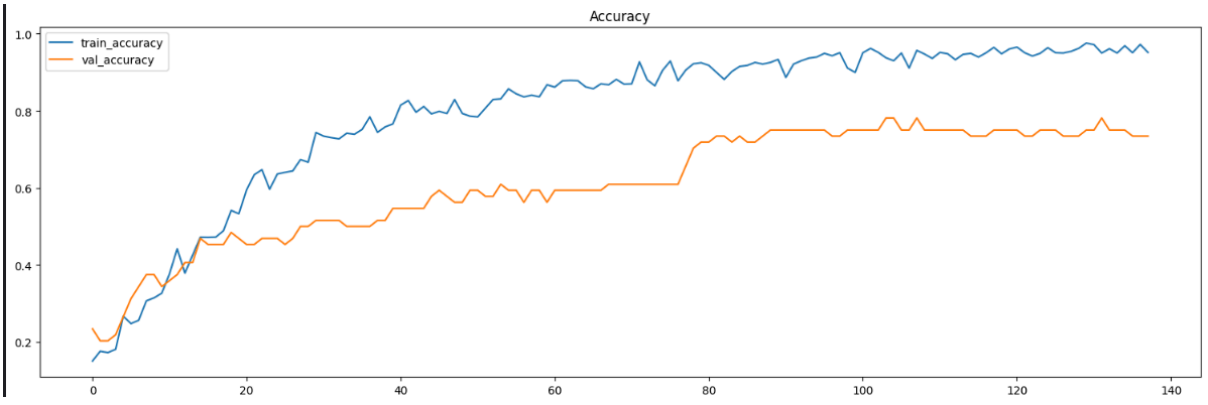
Epoch 74:
train_loss = 0.1704209248224894
train_cls_loss = 0.0023016540152538153
train_reg_loss = 0.2424720459514194
train_iou = 0.757537935878079
train_accuracy = 1.0
val_loss = 0.3586379438638687
val_cls_loss = 0.5021879822015762
val_reg_loss = 0.2971165180206299
val_iou = 0.7028967154851203
val_accuracy = 0.78125



score: 0.76485

Predict directly with the backbone output

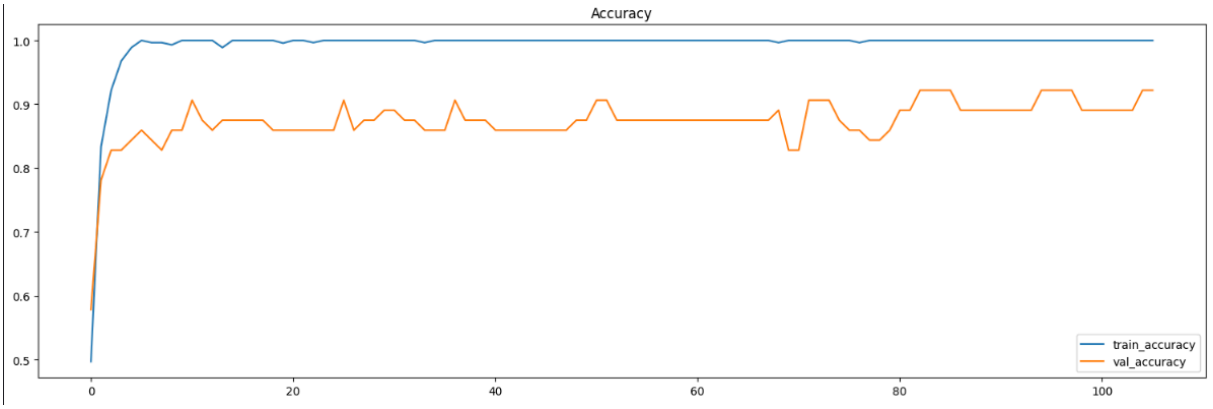
Epoch 128:
train_loss = 0.17339016993840536
train_cls_loss = 0.19350806375344595
train_reg_loss = 0.16476821237140232
train_iou = 0.8352440881822292
train_accuracy = 0.9540598326259189
val_loss = 0.40409789979457855
val_cls_loss = 0.7263704240322113
val_reg_loss = 0.26598110795021057
val_iou = 0.734033905822244
val_accuracy = 0.734375



1 Block with 768 filters

Epoch 96:
train_loss = 0.15054692162407768
train_cls_loss = 0.002029111492447555
train_reg_loss = 0.2141974171002706
train_iou = 0.7858132854934001
train_accuracy = 1.0
val_loss = 0.3076147586107254

val_cls_loss = 0.3443949967622757
val_reg_loss = 0.29185178875923157
val_iou = 0.708162352323261
val_accuracy = 0.921875



Total params: 7,234,506
Trainable params: 6,499,082
Non-trainable params: 735,424

Input size (MB): 1.17
Forward/backward pass size (MB): 186.85
Params size (MB): 27.60
Estimated Total Size (MB): 215.62

Conclusion

Has expected this to reduce the size of the model significantly and even improve the accuracy to 0.9375. This was achieved with a convolutional layer of 256 filters and a kernel of 3x3.

Full CNN model

This model achieves a score of 0.80. for validation obtaining 0.9375 accuracy and 0.7005183803664591iou.

Model summary

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 96, 125, 197]	14,208
ReLU-2	[-1, 96, 125, 197]	0
MaxPool2d-3	[-1, 96, 62, 98]	0
Conv2d-4	[-1, 16, 62, 98]	1,552
ReLU-5	[-1, 16, 62, 98]	0
Conv2d-6	[-1, 64, 62, 98]	1,088
ReLU-7	[-1, 64, 62, 98]	0

Conv2d-8	[-1, 64, 62, 98]	9,280
ReLU-9	[-1, 64, 62, 98]	0
Fire-10	[-1, 128, 62, 98]	0
Conv2d-11	[-1, 16, 62, 98]	2,064
ReLU-12	[-1, 16, 62, 98]	0
Conv2d-13	[-1, 64, 62, 98]	1,088
ReLU-14	[-1, 64, 62, 98]	0
Conv2d-15	[-1, 64, 62, 98]	9,280
ReLU-16	[-1, 64, 62, 98]	0
Fire-17	[-1, 128, 62, 98]	0
Conv2d-18	[-1, 32, 62, 98]	4,128
ReLU-19	[-1, 32, 62, 98]	0
Conv2d-20	[-1, 128, 62, 98]	4,224
ReLU-21	[-1, 128, 62, 98]	0
Conv2d-22	[-1, 128, 62, 98]	36,992
ReLU-23	[-1, 128, 62, 98]	0
Fire-24	[-1, 256, 62, 98]	0
MaxPool2d-25	[-1, 256, 31, 49]	0
Conv2d-26	[-1, 32, 31, 49]	8,224
ReLU-27	[-1, 32, 31, 49]	0
Conv2d-28	[-1, 128, 31, 49]	4,224
ReLU-29	[-1, 128, 31, 49]	0
Conv2d-30	[-1, 128, 31, 49]	36,992
ReLU-31	[-1, 128, 31, 49]	0
Fire-32	[-1, 256, 31, 49]	0
Conv2d-33	[-1, 48, 31, 49]	12,336
ReLU-34	[-1, 48, 31, 49]	0
Conv2d-35	[-1, 192, 31, 49]	9,408
ReLU-36	[-1, 192, 31, 49]	0
Conv2d-37	[-1, 192, 31, 49]	83,136
ReLU-38	[-1, 192, 31, 49]	0
Fire-39	[-1, 384, 31, 49]	0
Conv2d-40	[-1, 48, 31, 49]	18,480
ReLU-41	[-1, 48, 31, 49]	0
Conv2d-42	[-1, 192, 31, 49]	9,408
ReLU-43	[-1, 192, 31, 49]	0
Conv2d-44	[-1, 192, 31, 49]	83,136
ReLU-45	[-1, 192, 31, 49]	0
Fire-46	[-1, 384, 31, 49]	0
Conv2d-47	[-1, 64, 31, 49]	24,640
ReLU-48	[-1, 64, 31, 49]	0
Conv2d-49	[-1, 256, 31, 49]	16,640
ReLU-50	[-1, 256, 31, 49]	0
Conv2d-51	[-1, 256, 31, 49]	147,712
ReLU-52	[-1, 256, 31, 49]	0
Fire-53	[-1, 512, 31, 49]	0
MaxPool2d-54	[-1, 512, 15, 24]	0
Conv2d-55	[-1, 64, 15, 24]	32,832

```
              ReLU-56          [-1, 64, 15, 24]               0
            Conv2d-57         [-1, 256, 15, 24]          16,640
              ReLU-58         [-1, 256, 15, 24]               0
            Conv2d-59         [-1, 256, 15, 24]         147,712
              ReLU-60         [-1, 256, 15, 24]               0
              Fire-61         [-1, 512, 15, 24]               0
AdaptiveAvgPool2d-62            [-1, 512, 7, 7]               0
 FeatureExtractor-63            [-1, 512, 7, 7]               0
            Conv2d-64          [-1, 256, 7, 7]       1,179,904
       BatchNorm2d-65          [-1, 256, 7, 7]             512
              ReLU-66          [-1, 256, 7, 7]               0
            Conv2d-67            [-1, 6, 7, 7]           1,542
AdaptiveMaxPool2d-68            [-1, 6, 1, 1]               0
            Conv2d-69          [-1, 256, 7, 7]       1,179,904
       BatchNorm2d-70          [-1, 256, 7, 7]             512
              ReLU-71          [-1, 256, 7, 7]               0
            Conv2d-72          [-1, 256, 7, 7]         590,080
       BatchNorm2d-73          [-1, 256, 7, 7]             512
              ReLU-74          [-1, 256, 7, 7]               0
            Conv2d-75          [-1, 256, 7, 7]         590,080
       BatchNorm2d-76          [-1, 256, 7, 7]             512
              ReLU-77          [-1, 256, 7, 7]               0
            Conv2d-78          [-1, 256, 7, 7]         590,080
       BatchNorm2d-79          [-1, 256, 7, 7]             512
              ReLU-80          [-1, 256, 7, 7]               0
            Conv2d-81            [-1, 4, 7, 7]           1,028
AdaptiveMaxPool2d-82            [-1, 4, 1, 1]               0
================================================================
Total params: 4,870,602
Trainable params: 4,135,178
Non-trainable params: 735,424
----------------------------------------------------------------
Input size (MB): 1.17
Forward/backward pass size (MB): 186.28
Params size (MB): 18.58
Estimated Total Size (MB): 206.02
```

Comparison

Parameter	**First model**	**Full CNN model**
Total params	27,319,626	4,870,602
Trainable params	26,584,202	4,135,178

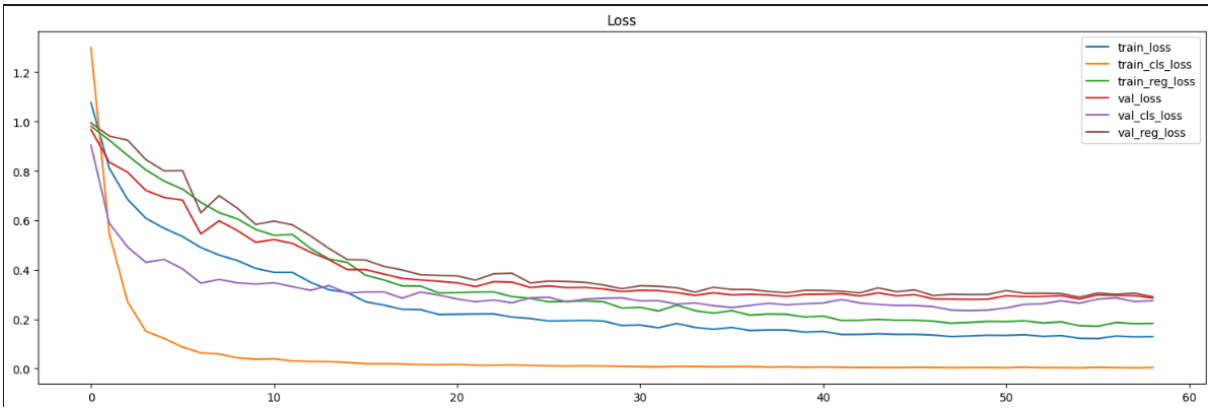
Parameter	First model	Full CNN model
Non-trainable params	735,424	735,424
Input size (MB)	1.17	1.17
Forward/backward pass (MB)	185.28	186.28
Params size (MB)	104.22	18.58
Estimated Total Size (MB)	290.66	206.02

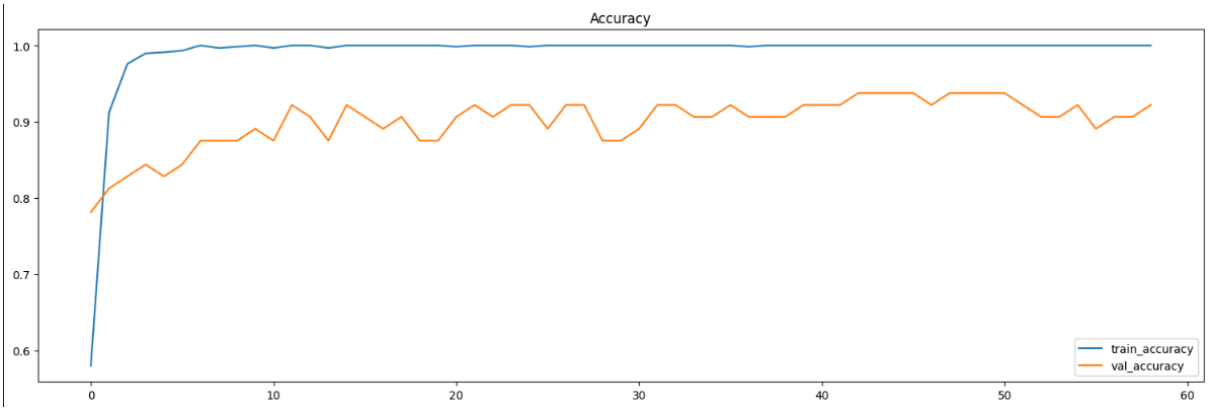
This reduces the parameters by 82.1% and even improves the performance. It also implies a reduction for the weight dictionary to 18.58 MB

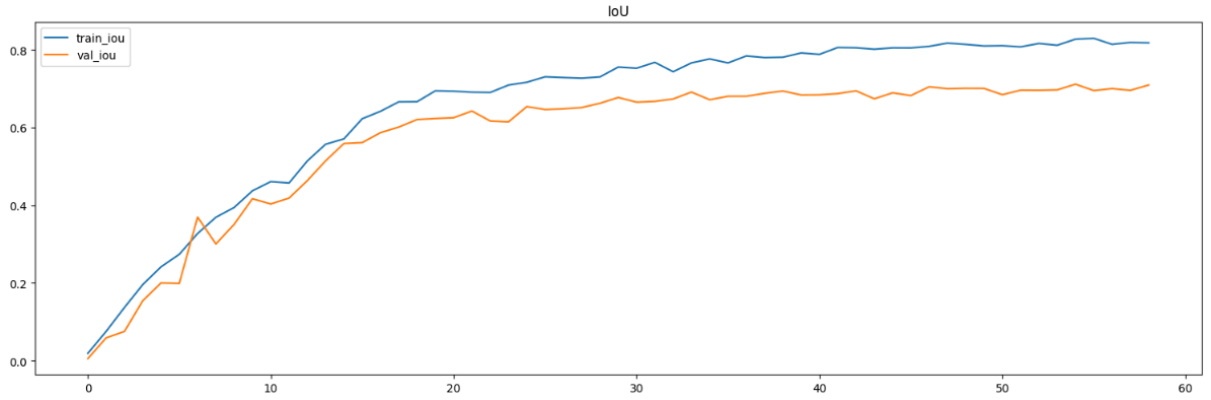
Augmentation

4

Epoch 49:
train_loss = 0.1315369527373049
train_cls_loss = 0.003647162695415318
train_reg_loss = 0.1863468653625912
train_iou = 0.8136654472802434
train_accuracy = 1.0
val_loss = 0.279836043715477
val_cls_loss = 0.2341451793909073
val_reg_loss = 0.2994178235530853
val_iou = 0.7005948661985131
val_accuracy = 0.9375





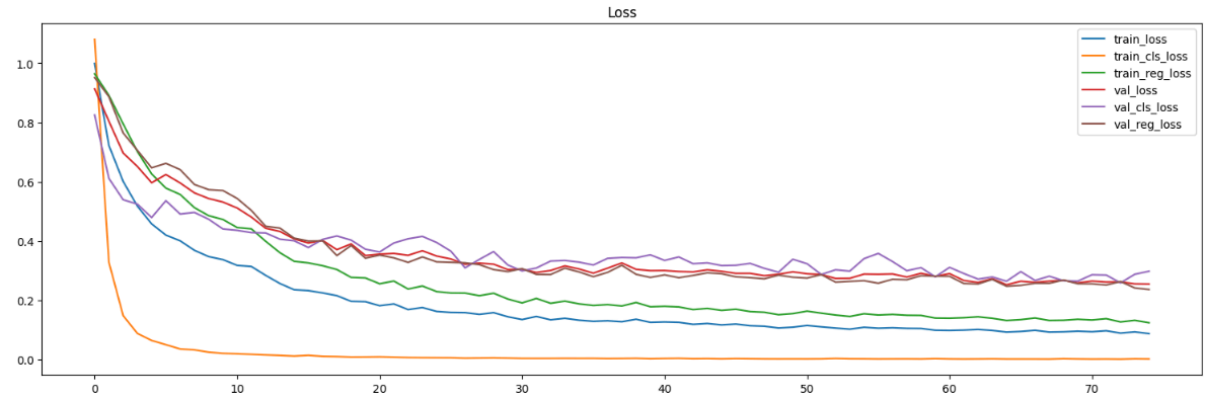


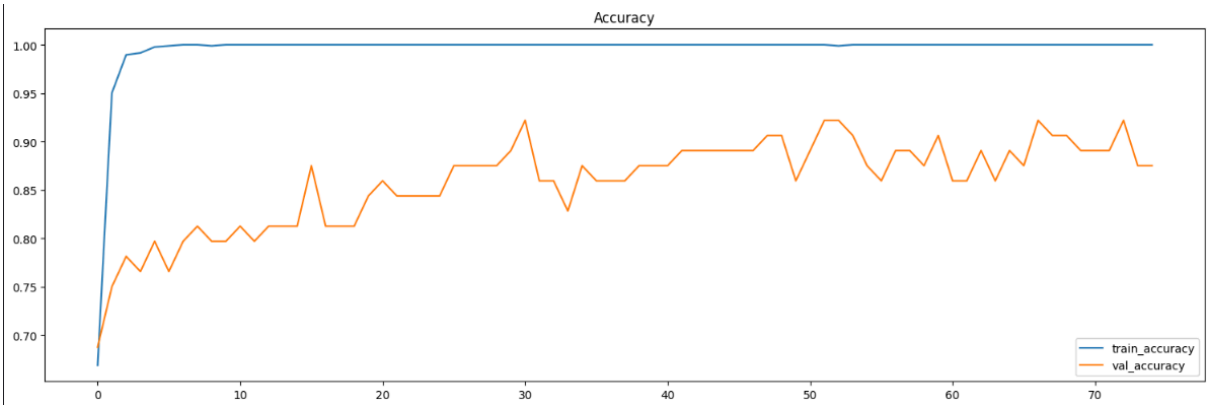
score:0.814

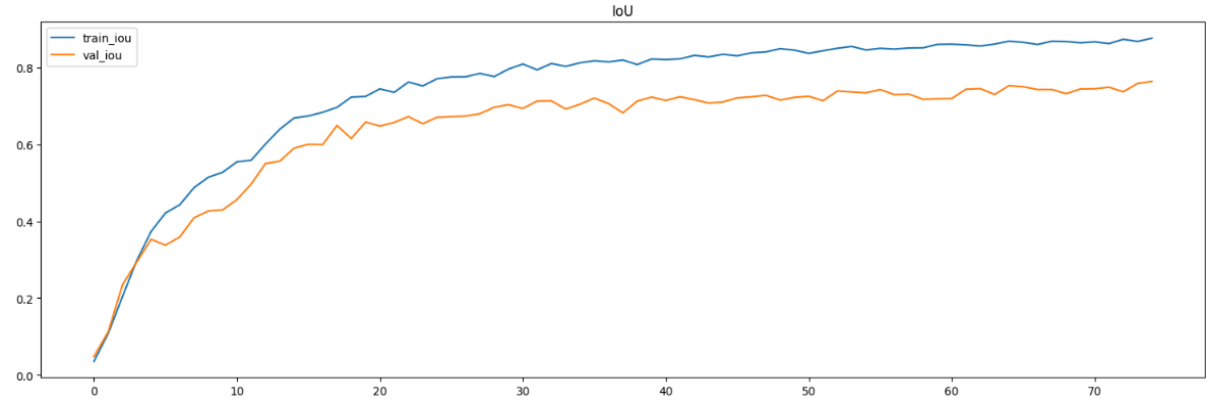
6

Epoch 65:
train_loss = 0.09225951300726996
train_cls_loss = 0.0011921411636716653
train_reg_loss = 0.1312883871572989
train_iou = 0.8687257491477547
train_accuracy = 1.0
val_loss = 0.25203466415405273
val_cls_loss = 0.26389359682798386
val_reg_loss = 0.24695226550102234
val_iou = 0.7530630233974849

val_accuracy = 0.890625



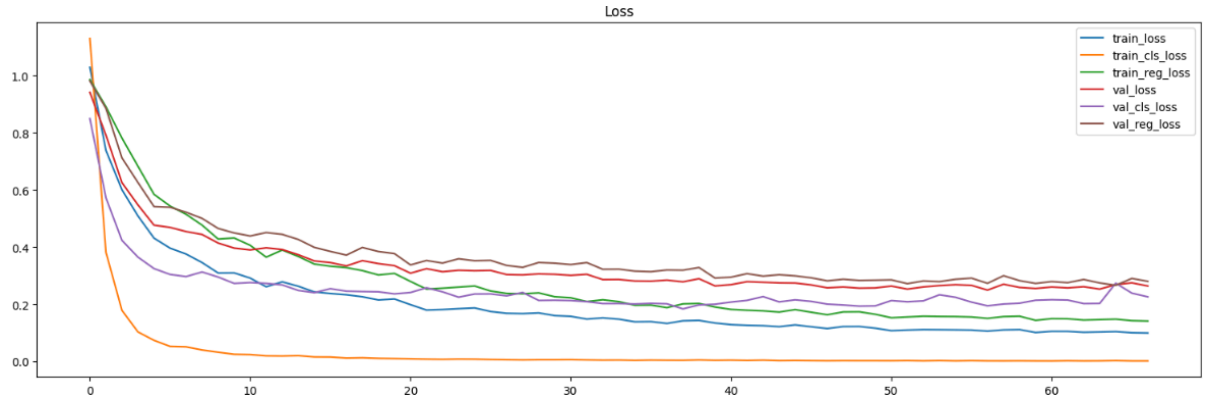


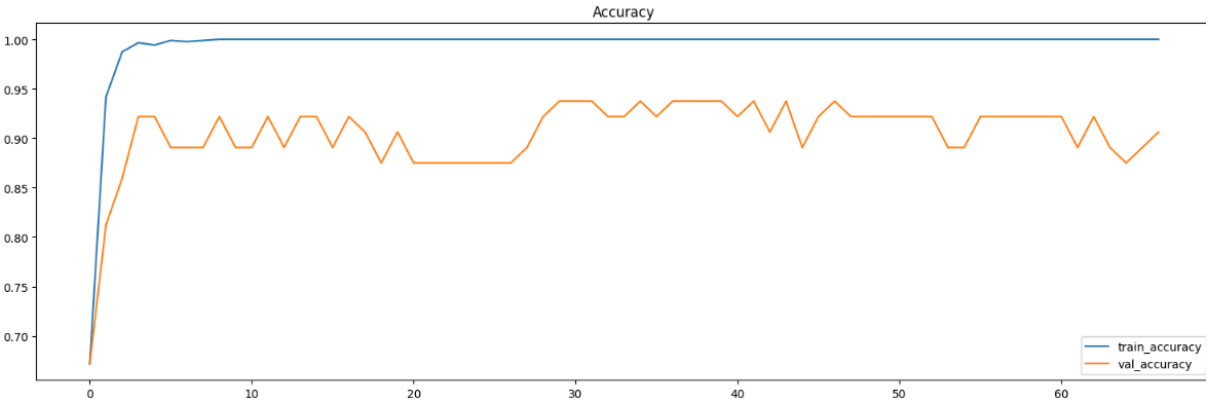


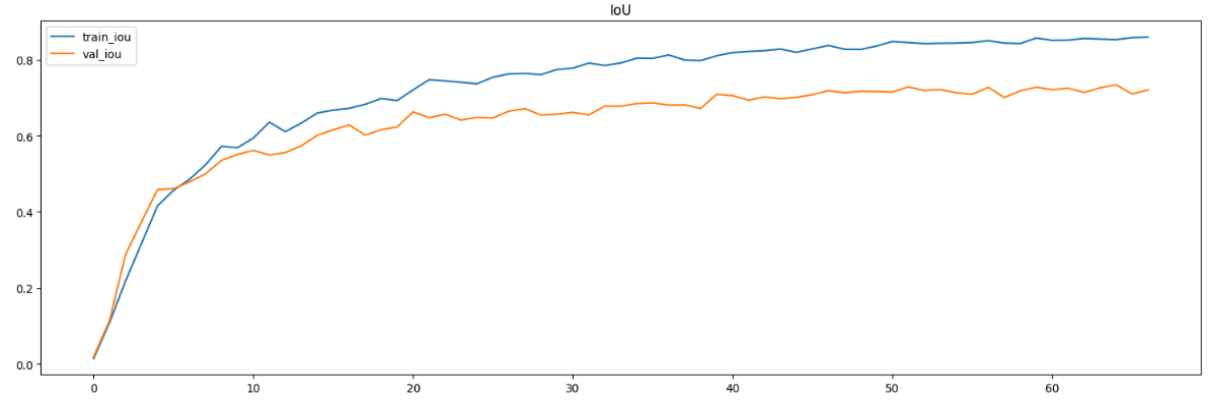
score: 0.83748

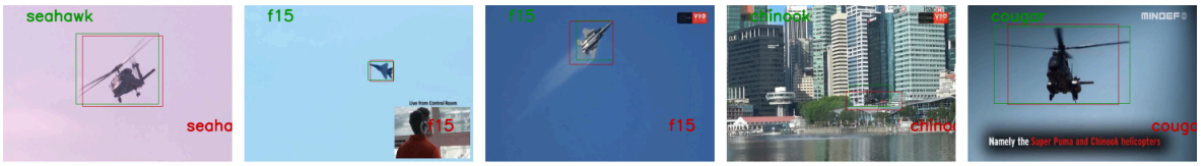
450 x 305

Epoch 57:
train_loss = 0.1055214943157302
train_cls_loss = 0.0014108879997231135
train_reg_loss = 0.15014032743595265
train_iou = 0.8498729395481852
train_accuracy = 1.0
val_loss = 0.2492019534111023
val_cls_loss = 0.1940048336982727
val_reg_loss = 0.2728578448295593
val_iou = 0.7271569002098878
val_accuracy = 0.921875





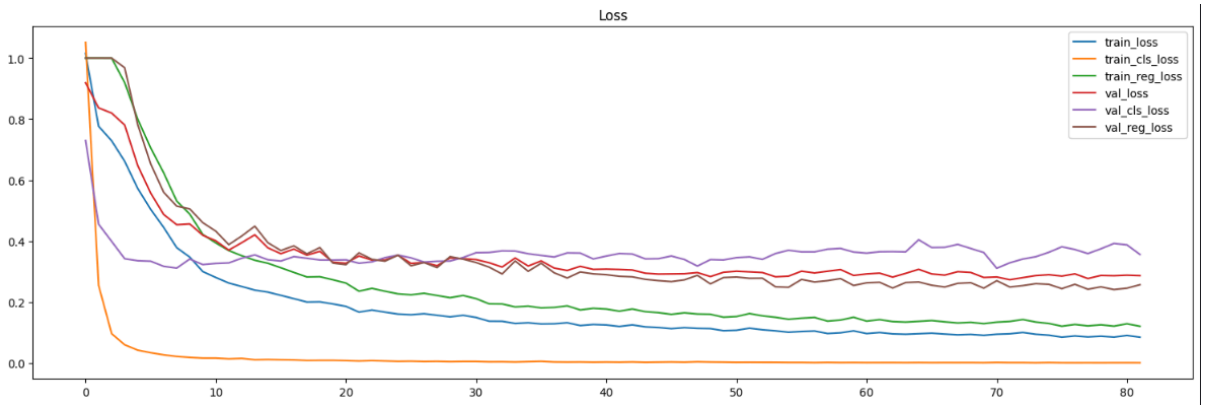


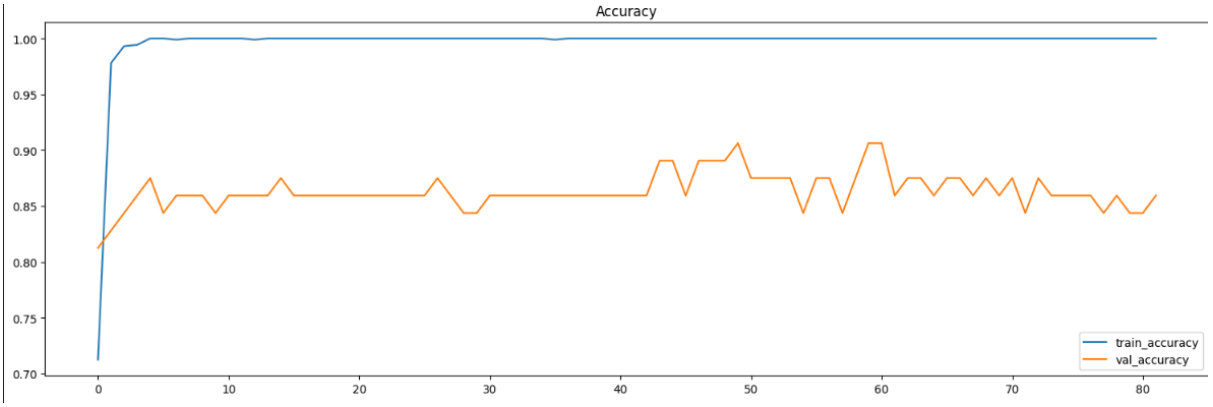


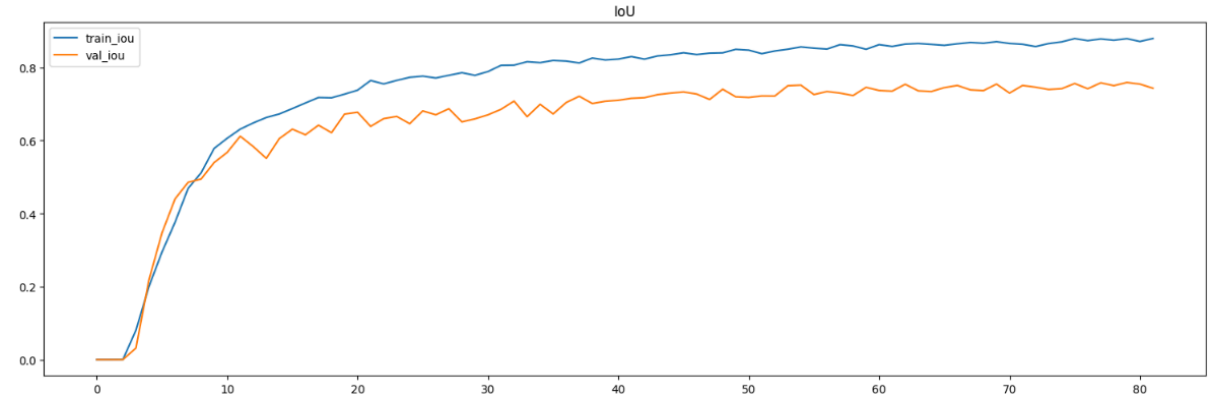
score: 0.79034

500 x 355

Epoch 72:
train_loss = 0.0957042160961363
train_cls_loss = 0.001190117633625589
train_reg_loss = 0.13621026056784172
train_iou = 0.8638039268885758
train_accuracy = 1.0
val_loss = 0.27291323244571686
val_cls_loss = 0.32854582369327545
val_reg_loss = 0.24907070398330688
val_iou = 0.7509460535145271
val_accuracy = 0.84375





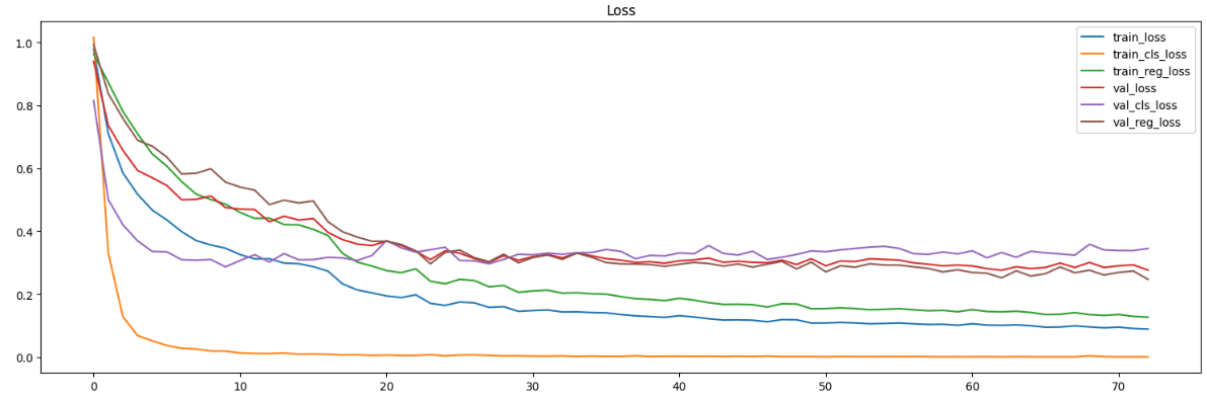


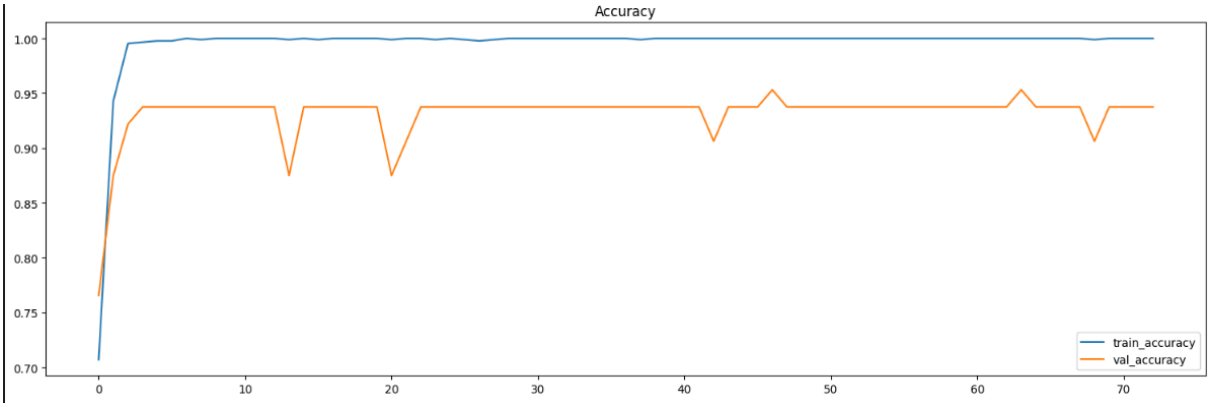
score: 0.80319

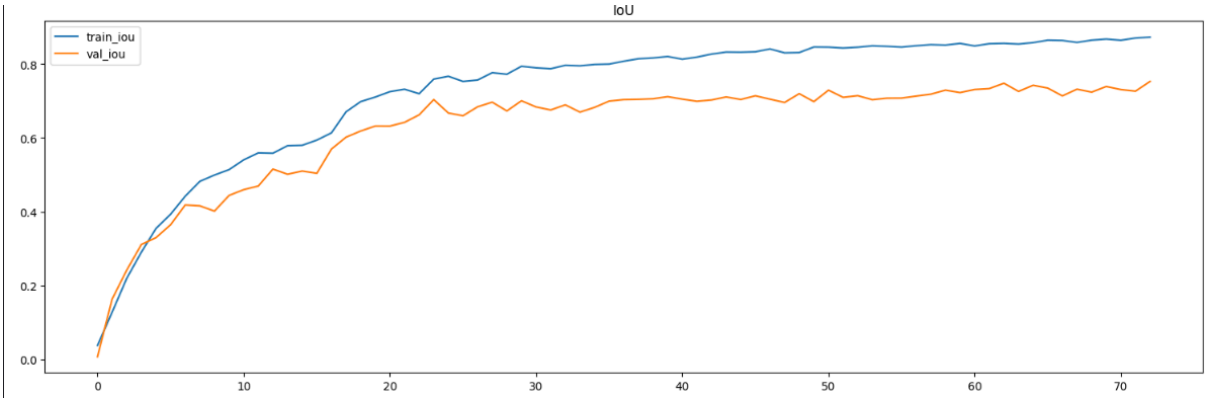
400 x 300

Epoch 63:
train_loss = 0.1009940364294582
train_cls_loss = 0.0009343548488147833
train_reg_loss = 0.1438767600942541
train_iou = 0.8561370642789401
train_accuracy = 1.0
val_loss = 0.27591148018836975
val_cls_loss = 0.33215077966451645
val_reg_loss = 0.25180891156196594

val_iou = 0.748208568695253
val_accuracy = 0.9375



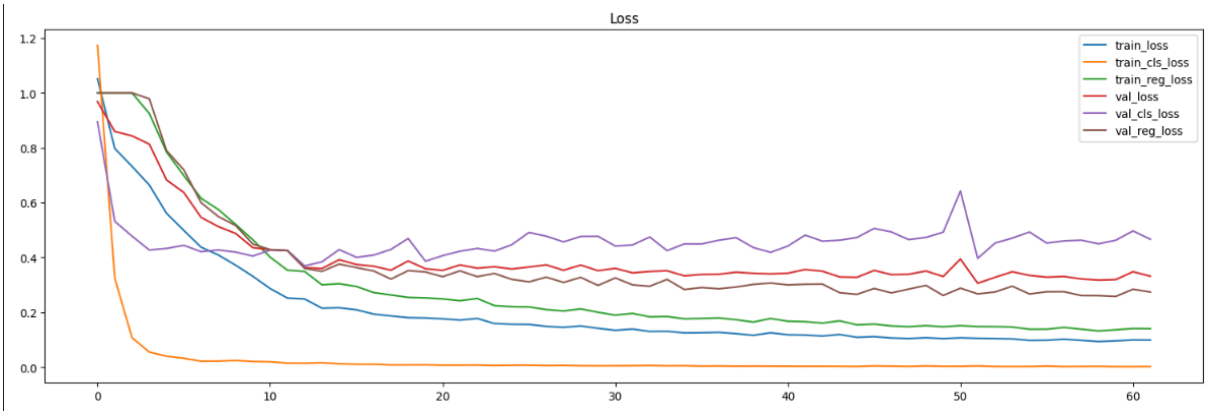


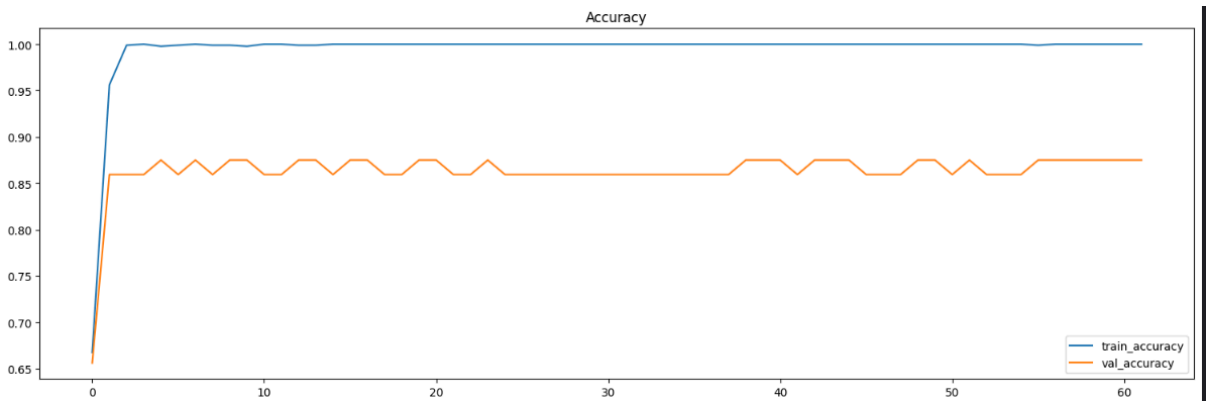


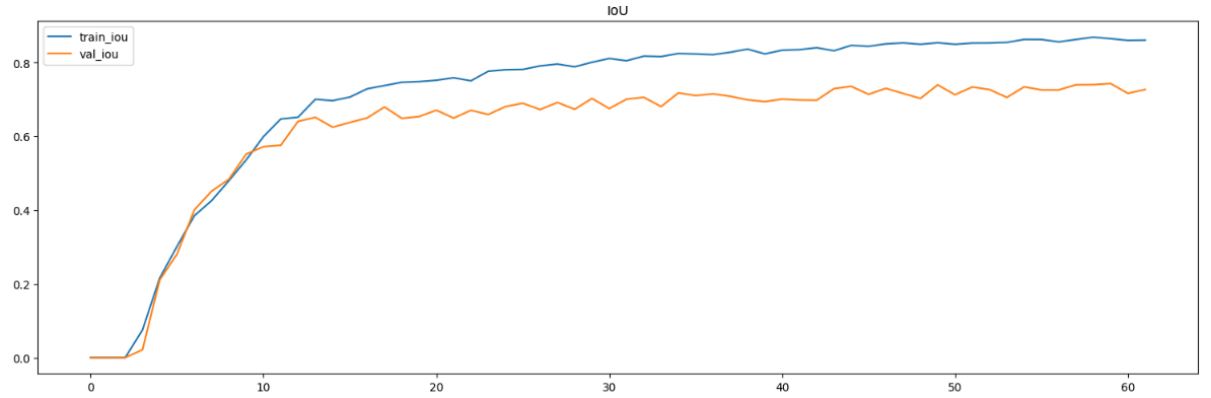
score: 0.80074

400 x 400

Epoch 52:
train_loss = 0.10455472850137287
train_cls_loss = 0.0039270936524392
train_reg_loss = 0.1476808609785857
train_iou = 0.8523325170981932
train_accuracy = 1.0
val_loss = 0.30552931129932404
val_cls_loss = 0.39622706174850464
val_reg_loss = 0.26665884256362915
val_iou = 0.7333569505311511
val_accuracy = 0.875



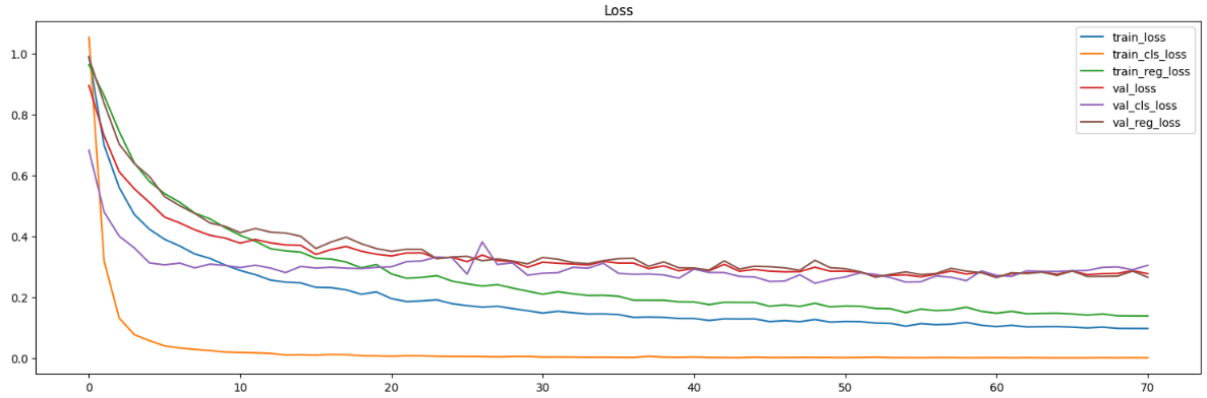


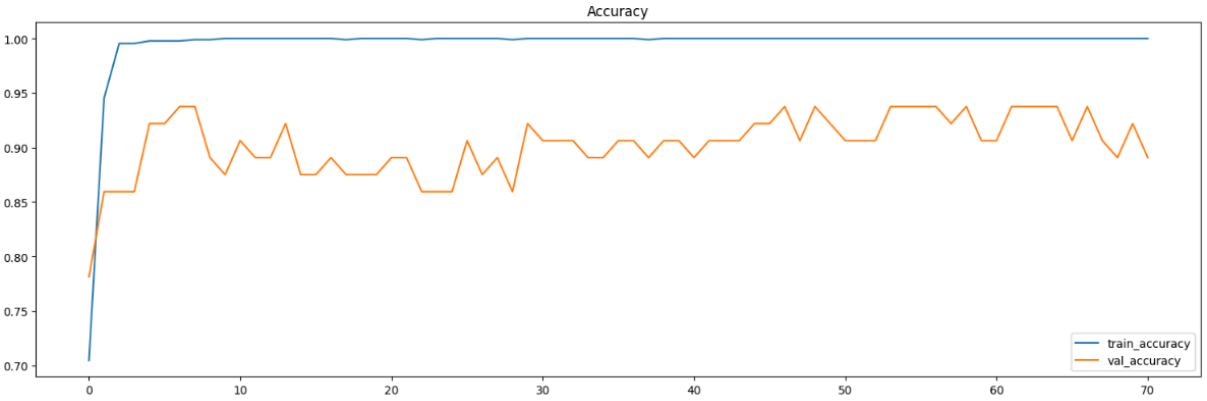


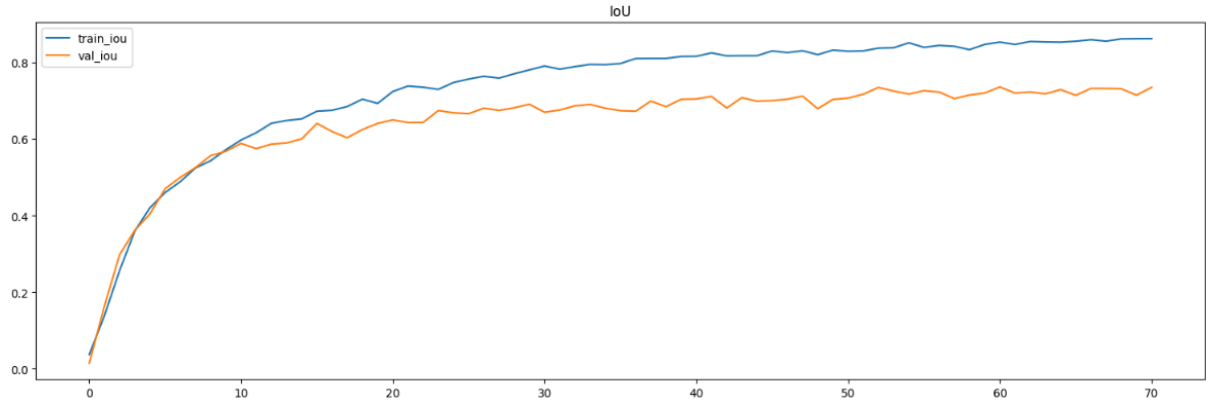
430 x 285

Epoch 61:
train_loss = 0.10344426692635925
train_cls_loss = 0.0014610412364601398
train_reg_loss = 0.14715136642809268
train_iou = 0.8528621647624074
train_accuracy = 1.0
val_loss = 0.266513854265213
val_cls_loss = 0.2717360630631447
val_reg_loss = 0.26427575945854187

val_iou = 0.7357407358174483
val_accuracy = 0.90625



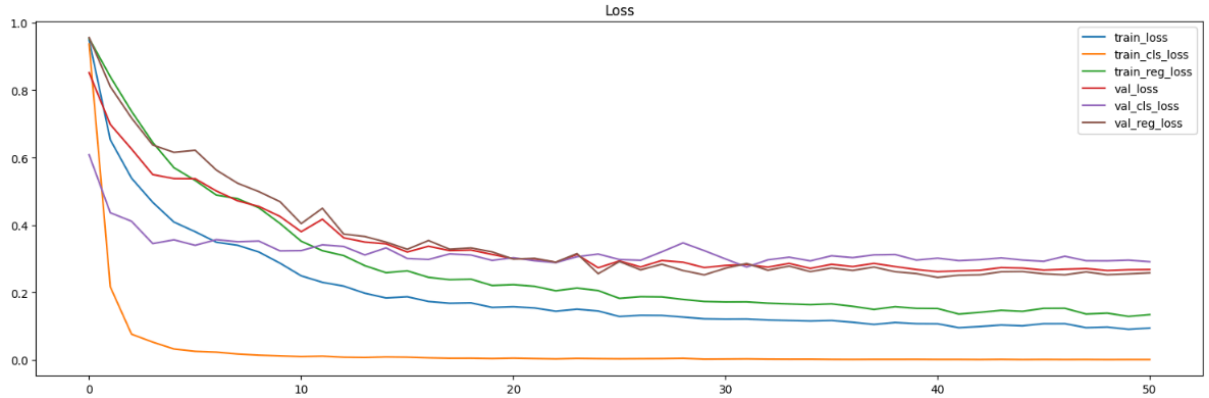


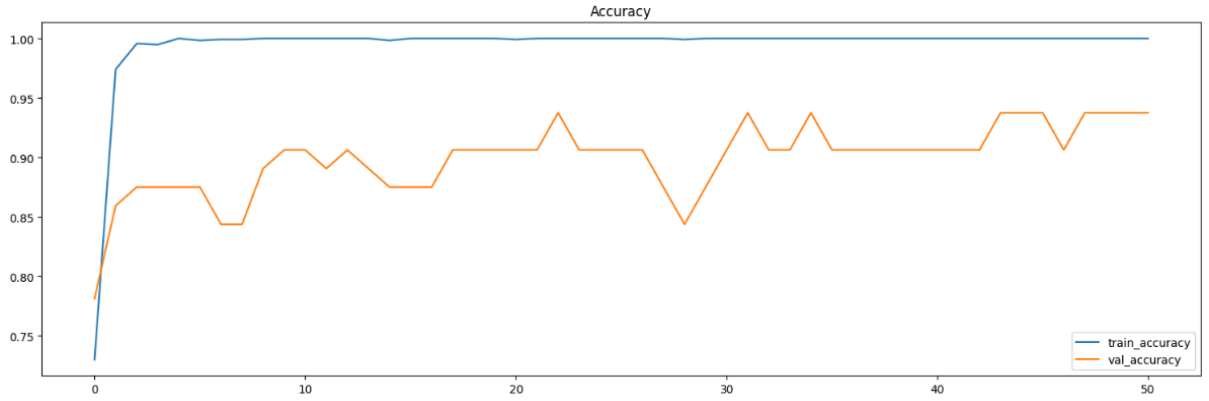


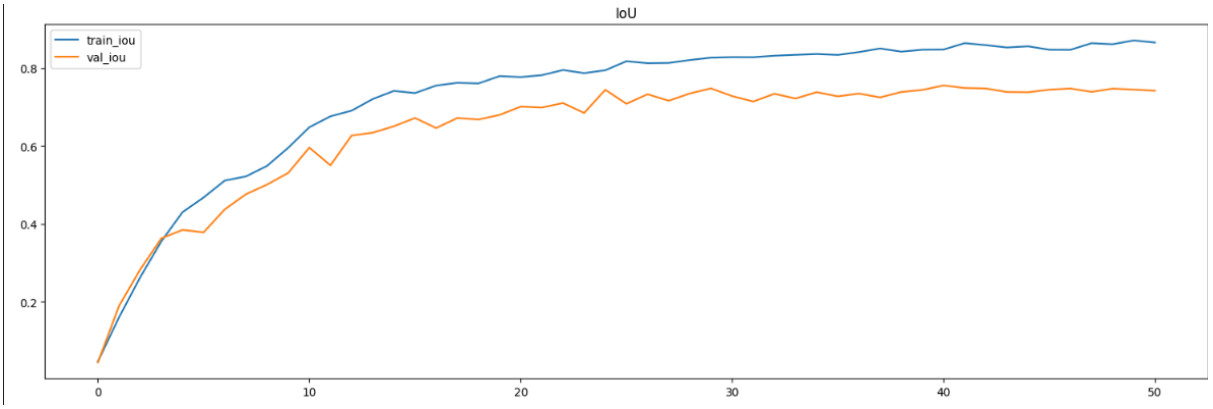
score: 0.78801

8

Epoch 41:
train_loss = 0.10697569760183494
train_cls_loss = 0.001276347556187021
train_reg_loss = 0.1522754215531879
train_iou = 0.847737793796576
train_accuracy = 1.0
val_loss = 0.26165148615837097
val_cls_loss = 0.3016416132450104
val_reg_loss = 0.24451285600662231
val_iou = 0.755502918286362
val_accuracy = 0.90625



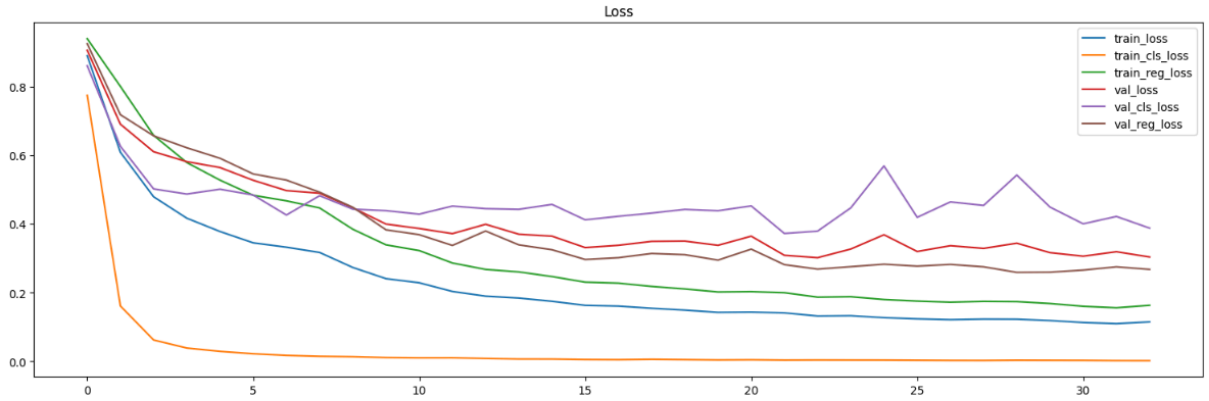


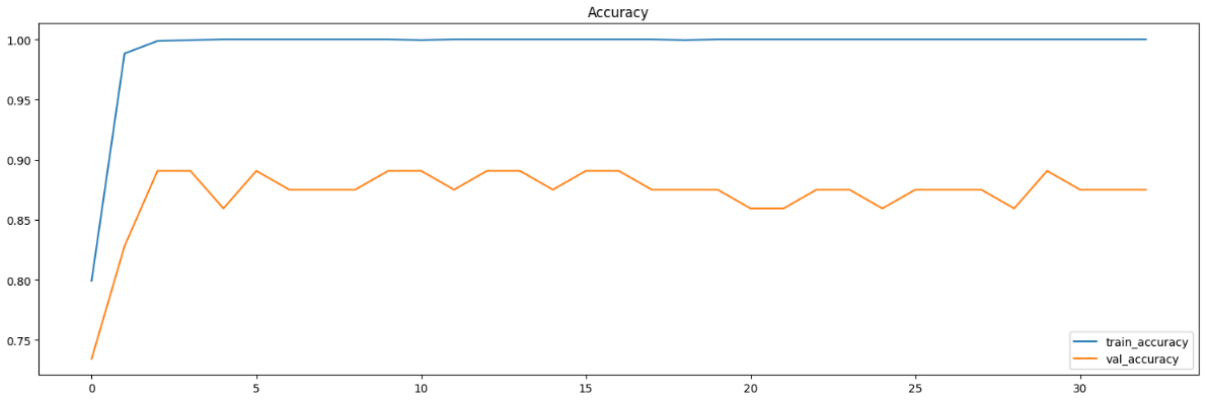


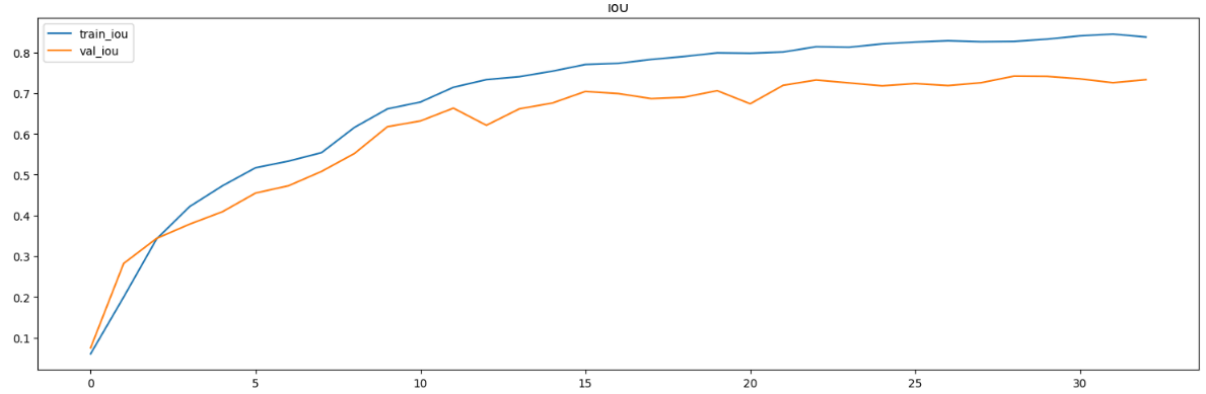
score: 0.80309

10

Epoch 23:
train_loss = 0.13141382982333502
train_cls_loss = 0.0031884461550766396
train_reg_loss = 0.1863675700293647
train_iou = 0.813644443819332
train_accuracy = 1.0
val_loss = 0.3012622147798538
val_cls_loss = 0.37878642976284027
val_reg_loss = 0.2680375576019287
val_iou = 0.7319806254125959
val_accuracy = 0.875



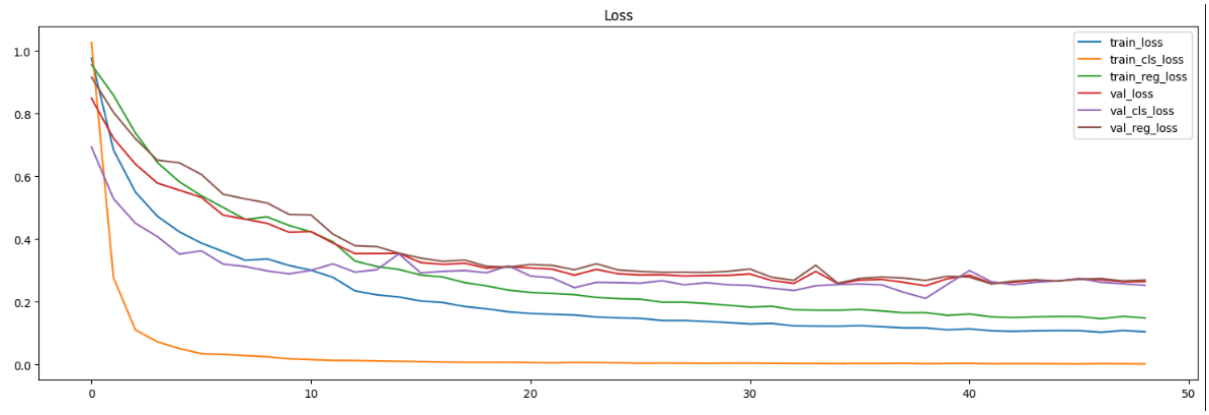


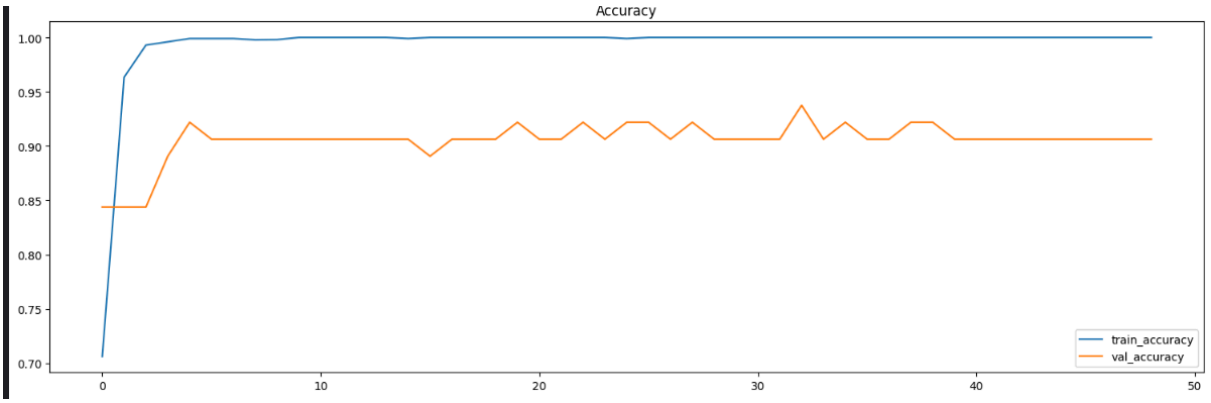


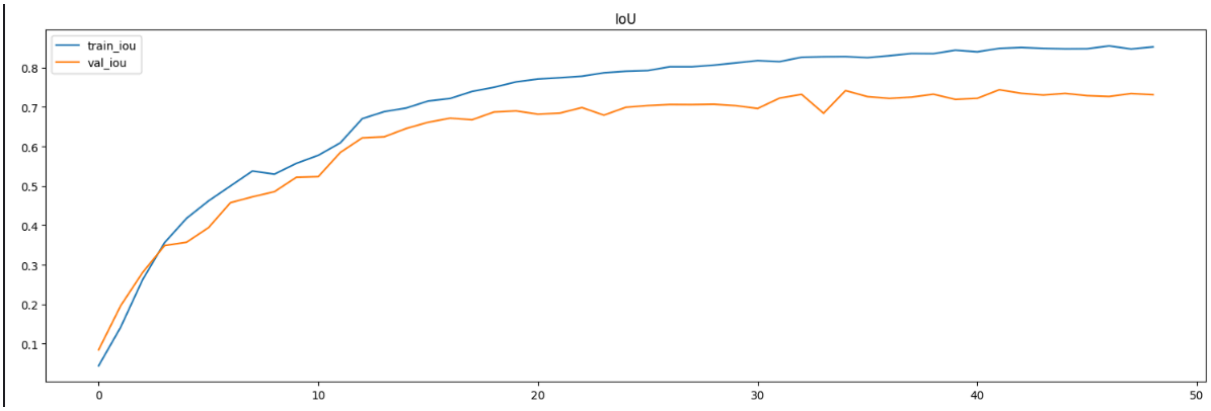
score: 0.79534

7

Epoch 39:
train_loss = 0.11591671743700581
train_cls_loss = 0.001879702870463652
train_reg_loss = 0.1647897247345217
train_iou = 0.8352228481695627
train_accuracy = 1.0
val_loss = 0.250308021903038
val_cls_loss = 0.21018142998218536
val_reg_loss = 0.26750513911247253
val_iou = 0.7325116243181528
val_accuracy = 0.921875





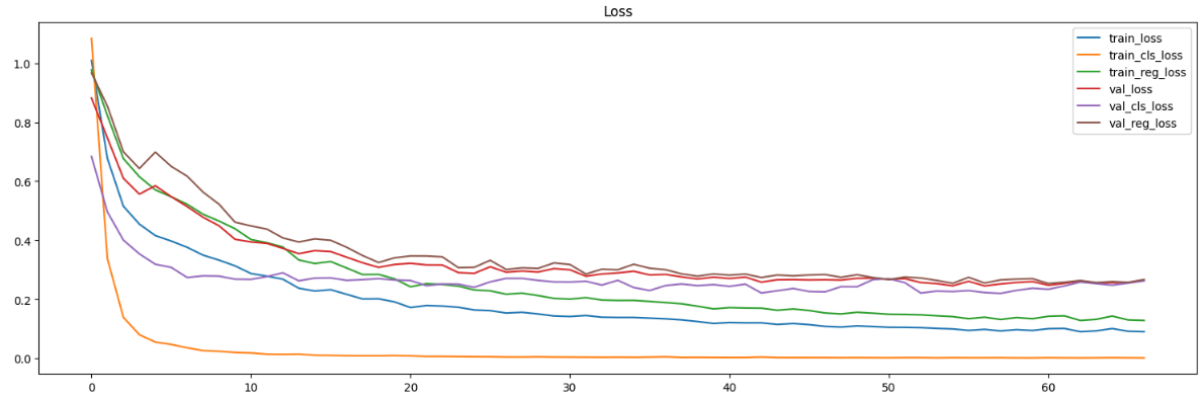


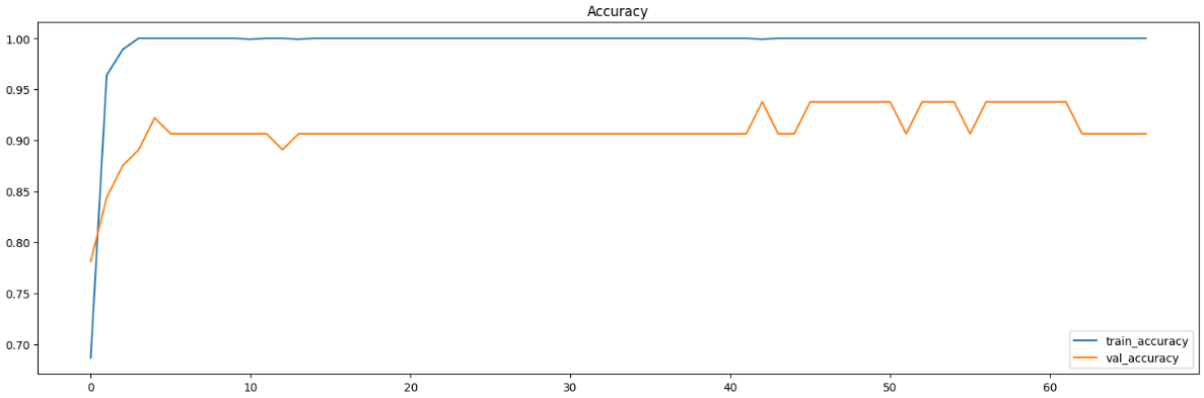
score: 0.81004

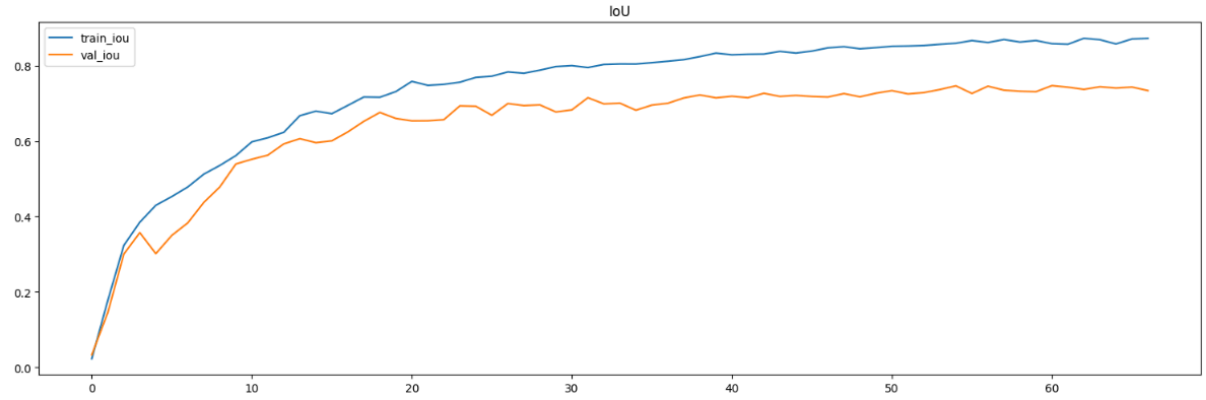
450 x 305

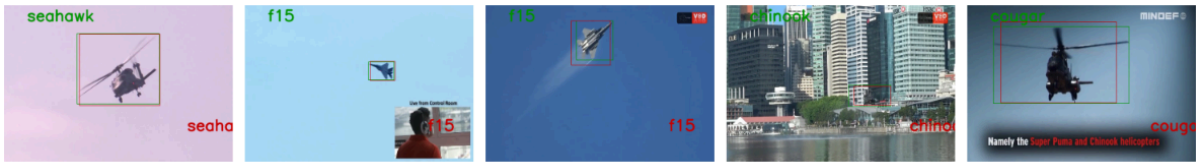
Epoch 57:
train_loss = 0.09759399054511901
train_cls_loss = 0.0013792195682218599
train_reg_loss = 0.13882889478437363
train_iou = 0.8611851035559411
train_accuracy = 1.0
val_loss = 0.24500136077404022
val_cls_loss = 0.222431518137455
val_reg_loss = 0.25467413663864136
val_iou = 0.7453426965541046

val_accuracy = 0.9375





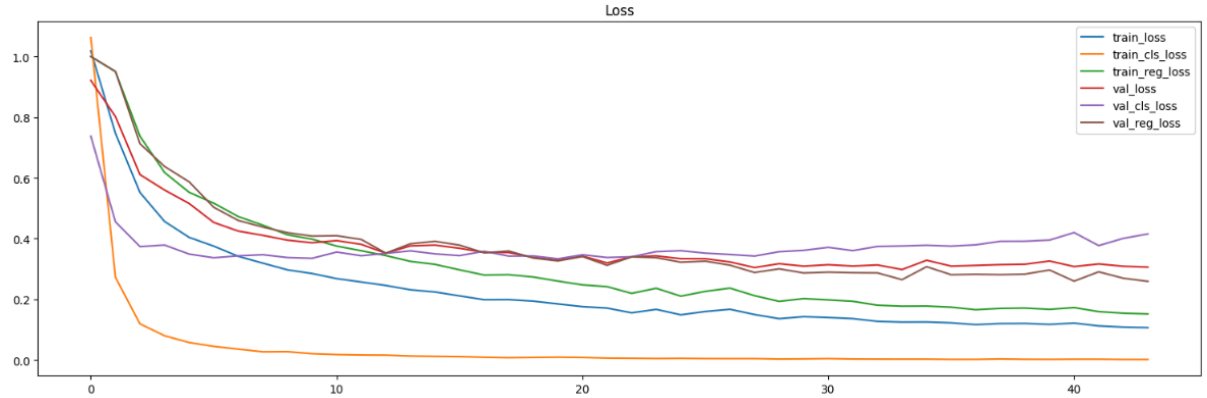


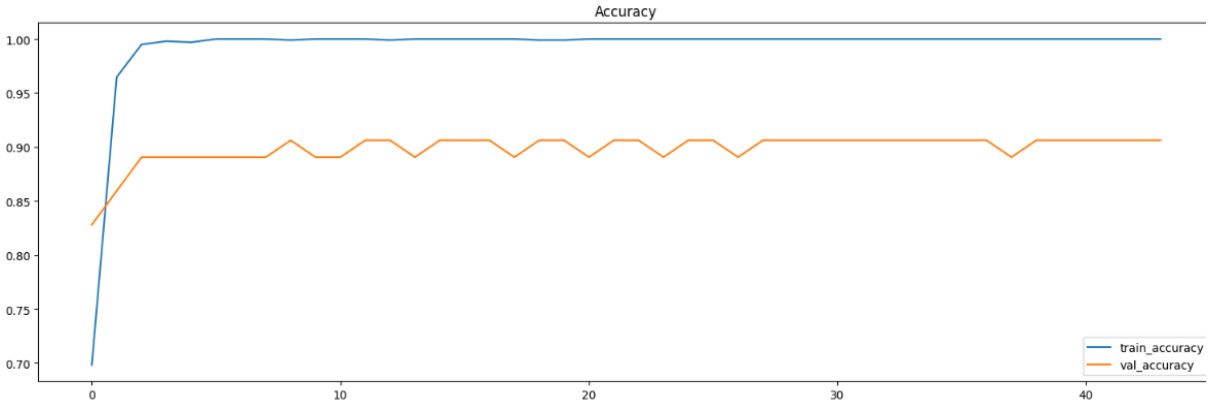


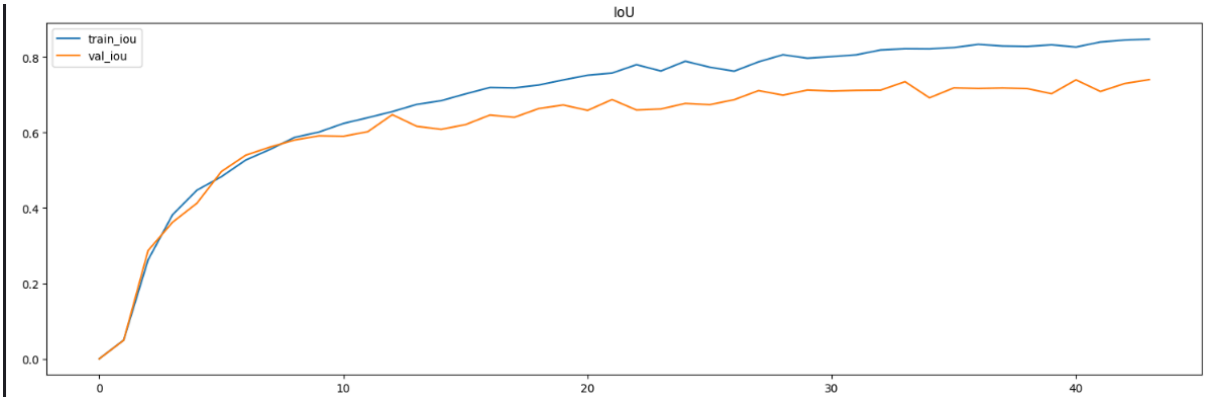
score: 0.82265

500 x 355

Epoch 34:
train_loss = 0.12518980041626962
train_cls_loss = 0.0030637949126802624
train_reg_loss = 0.1775295196040984
train_iou = 0.8224829094017504
train_accuracy = 1.0
val_loss = 0.2981881946325302
val_cls_loss = 0.3759872317314148
val_reg_loss = 0.26484575867652893
val_iou = 0.7351711188163481
val_accuracy = 0.90625







score: 0.79665

Conclusions

Generating 6 augmentations of images improves the score to 0.83748 on the test set. For the validation set the metrics are val_iou = 0.7530630233974849 val_accuracy = 0.890625 which has a ponderated score of 0.8218. This proves the model has a good generalization and should work better in production.
Some interesting insights are that the best performance was given with some initial hyperparameters that were randomly chosen to start the experimentation.

Hyperparameters

BATCH_SIZE = 32
EPOCHS = 200
ALPHA = 0.7
LEARNING_RATE = 1e-4
L2_LAMBDA = 1e-6
PYTORCH_SEED = 32
SPLIT_SEED = 42
AUGMENTATIONS= 6

TARGET_SIZE = (400, 255)