

# Optimization and Performance Tuning of Semantic Segmentation Models for Satellite Imagery

## Introduction:

This document outlines the iterative process of tuning and optimizing semantic segmentation models for satellite imagery. The primary objective was to enhance the performance of the models, particularly in handling class imbalance and achieving higher IoU and F1 scores.

Various strategies were employed, including adjustments to learning rates, loss functions, data augmentations, and batch sizes. The results of these experiments provide insights into the trade-offs and impacts of different hyperparameter settings and architectural modifications.

Each section documents the specific tweaks and their corresponding outcomes, showcasing a detailed exploration of parameters such as the weight decay, learning rate schedulers, and regularization techniques. This systematic approach highlights the importance of experimentation in achieving optimal performance, as well as the challenges of balancing generalization and overfitting in highly imbalanced datasets.

**Important Note:** This document was compiled from personal notes recorded during the experimentation and fine-tuning of various models. It is not intended to outline a definitive procedure but rather to reflect on the challenging yet enlightening journey—one filled with moments of frustration, discovery, and occasional humor—toward achieving acceptable model performance.

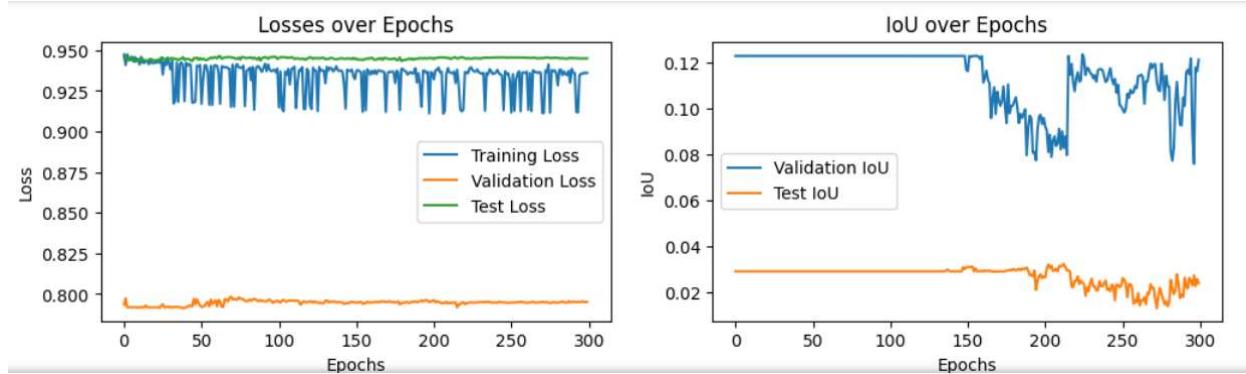
## Contents

Tuning Process and Parameters: .....	2
First steps: .....	2
Begin working on the Loss function:.....	7
Data Augmentation: .....	13
Batch Size Optimization:.....	19
Mange the Learning Rate:.....	23
Experiencing with other models: .....	25
Experiencing with Early Stopping:.....	29
Experiencing with Optimizers: .....	31
Improving the Loss function: .....	32
Study the Class Balance: .....	35
Apply Regularization: .....	54
Rebalancing the sets:.....	66
Introducing image interpolation: .....	97

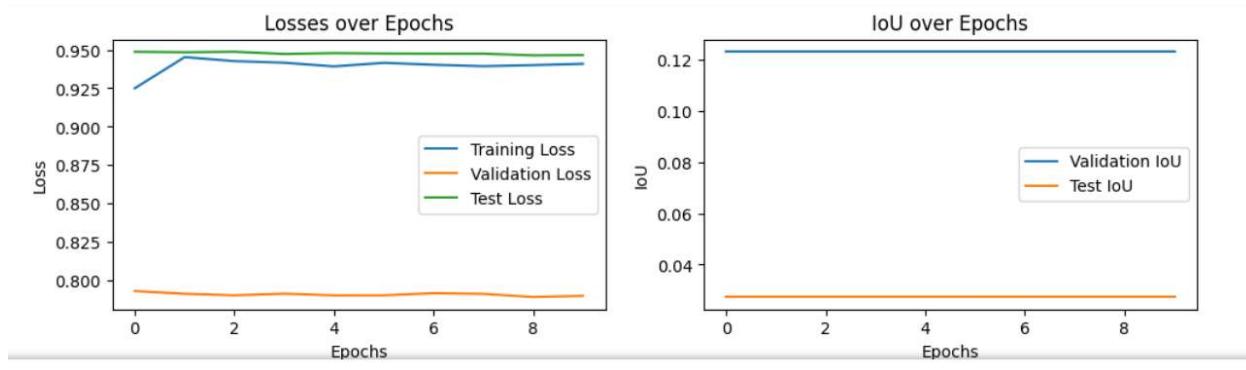
## Tuning Process and Parameters:

### First steps:

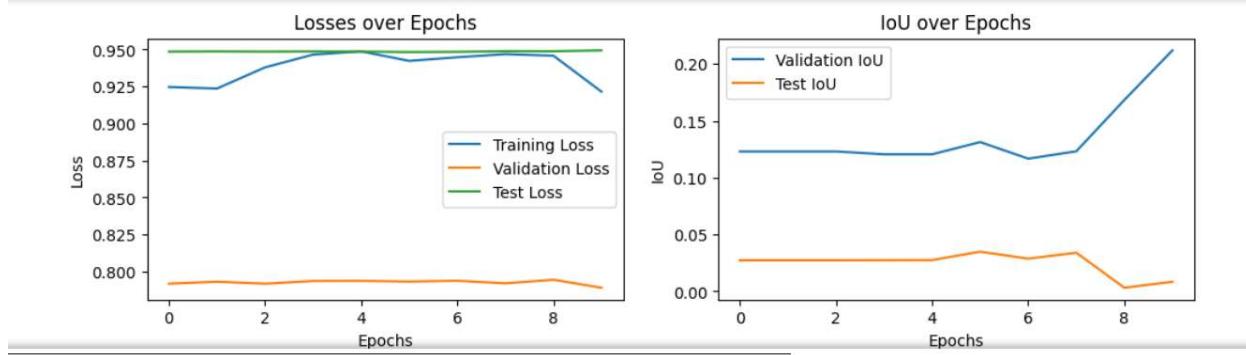
The following sections present the Loss and Metric charts obtained after each significant training iteration. These results reflect the impact of altering specific variables in the quest for improved model performance.



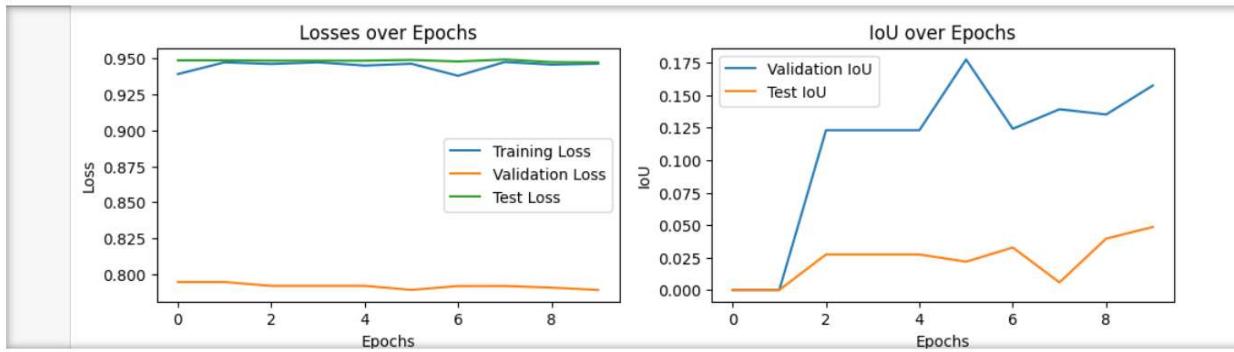
Lr=0,0005



Lr=0,05

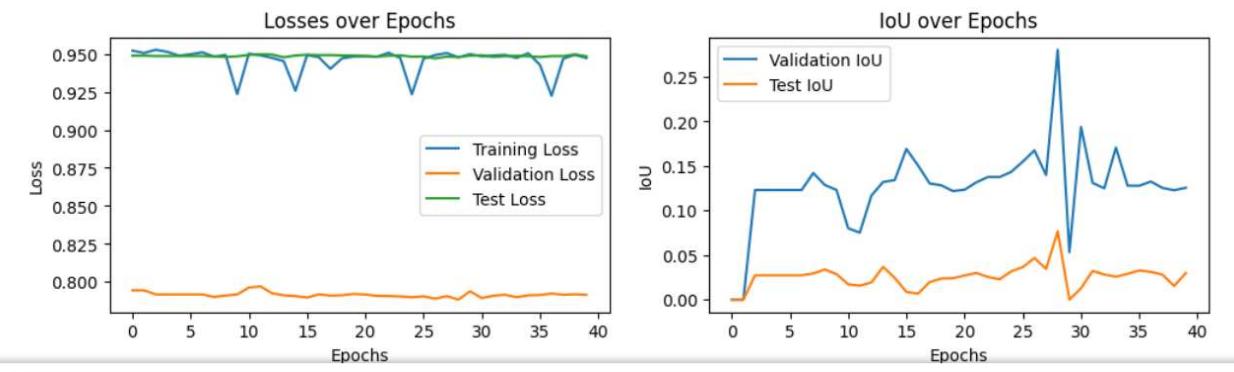


Lr=0,5



Not normalizing the images

Lr=0,05



Not normalizing the images

Lr=0,0005

We use 4-band images (Red, Green, Blue and NIR). These are the minimum and maximum value of reflectance for each band:

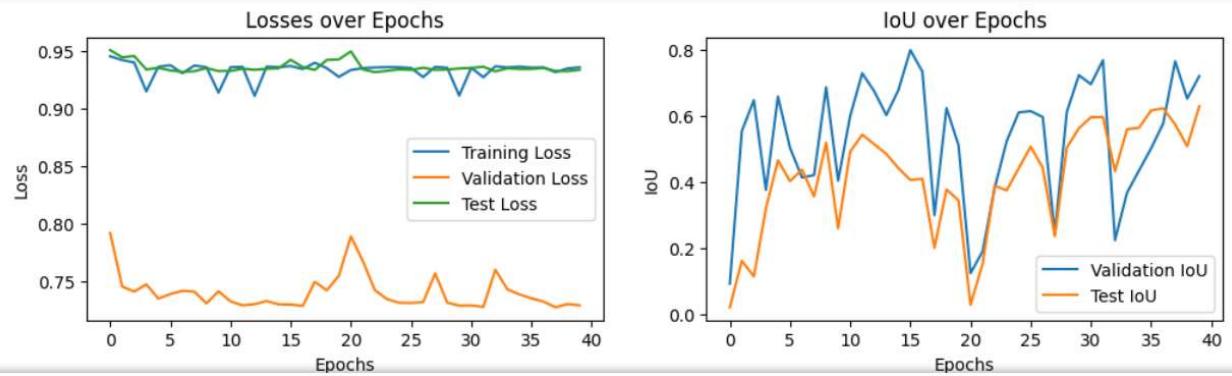
#### Bands

Band count	4			
Number	Band	No-Data	Min	Max
1	Band 1: Red	-9999	180.0000000000	4123.0000000000
2	Band 2: Green	-9999	254.0000000000	3676.0000000000
3	Band 3: Blue	-9999	170.0000000000	2948.0000000000
4	Band 4: NIR	-9999	247.0000000000	5721.0000000000

Then, after calculating the mean and standard deviation for each band considering all the dataset images, we add normalization as part of the dataset transformation:

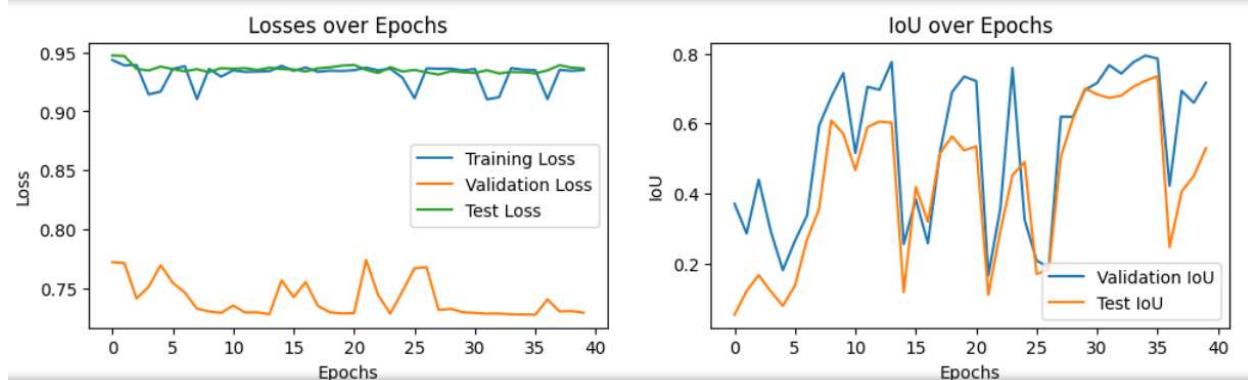
$lr=0.0005$

```
normalize_transform = transforms.Compose([transforms.ToTensor(),  
transforms.Normalize(mean=[0.485, 0.456, 0.406, 0.485], std=[0.229, 0.224, 0.225, 0.229])  
])
```



$lr=0.001$

```
normalize_transform = transforms.Compose([transforms.ToTensor(),  
transforms.Normalize(mean=[0.485, 0.456, 0.406, 0.485], std=[0.229, 0.224, 0.225, 0.229])  
])
```



New departing point:

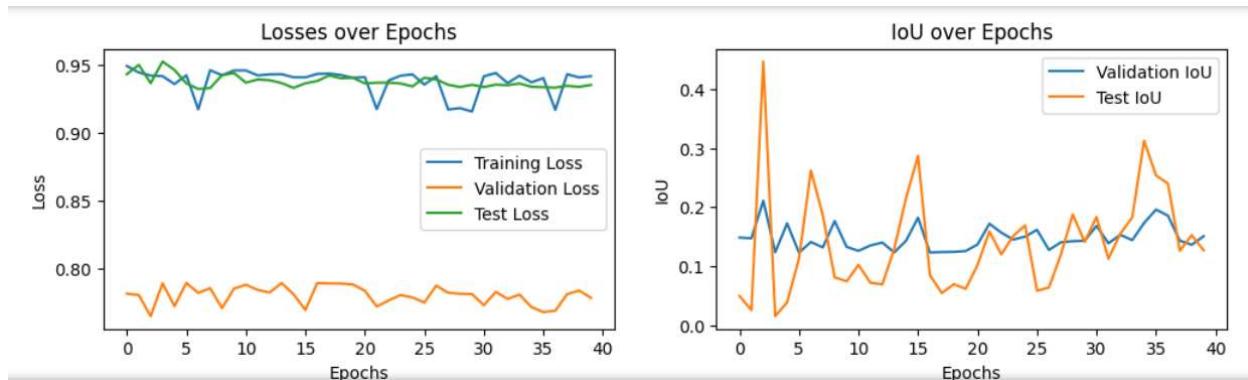


Change 1)

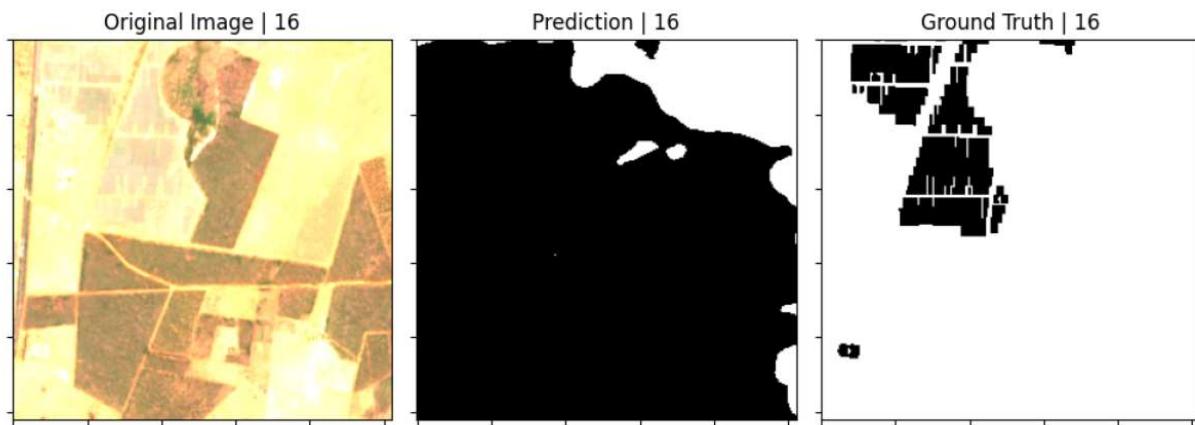
I changed float32 to int64 for the mask

```
mask = np.asarray(mask, dtype='int64') #float32'
```

output:



```
FileNotFoundException: [Errno 2] No such file or directory: 'E:\\Devs\\pyEnv-1\\UCSD_MLBootcamp_Capstone\\5b-Run_a_Model\\imgs\\img15.png'
```



No significant changes were observed after the modification. It is worth noting that the predictions remain far from the ground truth (target).

Change 2)

1<sup>st</sup> change train\_dataset using the normalized\_transform instead of transform (#)

```
train_dataset = CustomDataset(train_image_paths, train_mask_paths, transform=normalize_transform,
transform_label=None)

#train_dataset = CustomDataset(train_image_paths, train_mask_paths, transform=transform,
transform_label=None)
```

### Issue 1: Loss Function Not Improving with Epochs

#### Step 1: Reduce Learning Rate and Add Learning Rate Scheduler

```
#Reduce the learning rate value
```

```
optimizer = torch.optim.Adam([ dict(params=model.parameters(), lr=0.00001), ])
```

```
# Add learning rate scheduler
```

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

```
# Adjusts every 10 epochs, decreasing by 10%
```

```
# Include the scheduler step after every epoch for epoch in range(0, 40):
```

```
# Training step train_loss = train_one_epoch(model, train_loader, optimizer, loss, DEVICE)
```

```
scheduler.step() # Update the learning rate
```

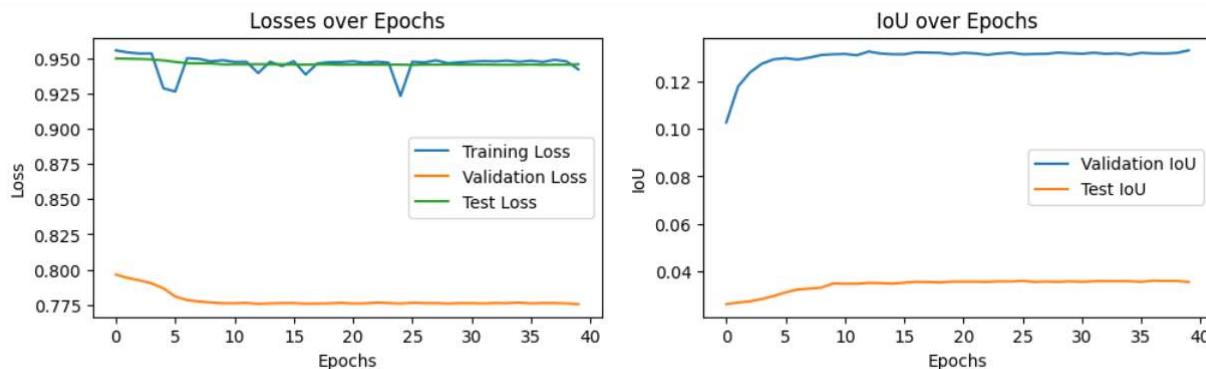
normalization after calculation mean and std for the training set images (201 images)

```
normalize_transform = transforms.Compose([
```

```
transforms.ToTensor(),
```

```
transforms.Normalize(mean=[946.2, 749.7, 494.2, 2493.8], std=[419.2, 260.8, 201.4, 477.1])
```

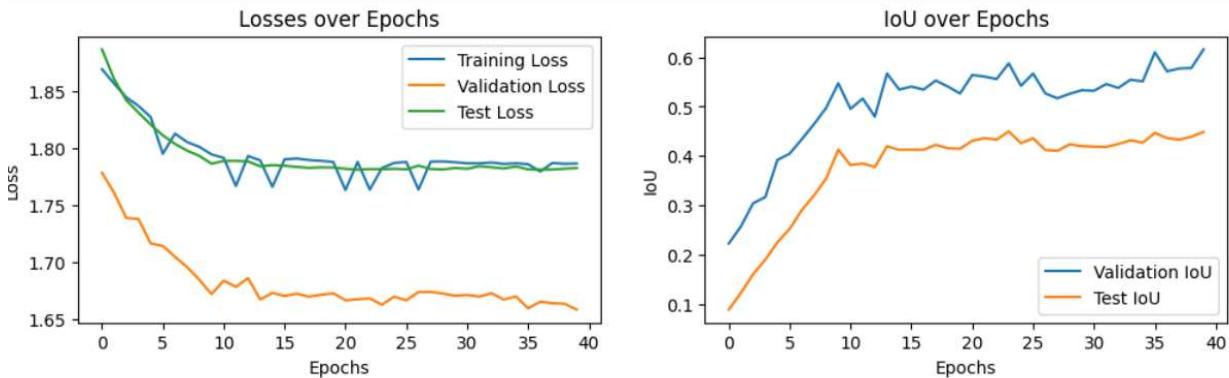
```
])
```



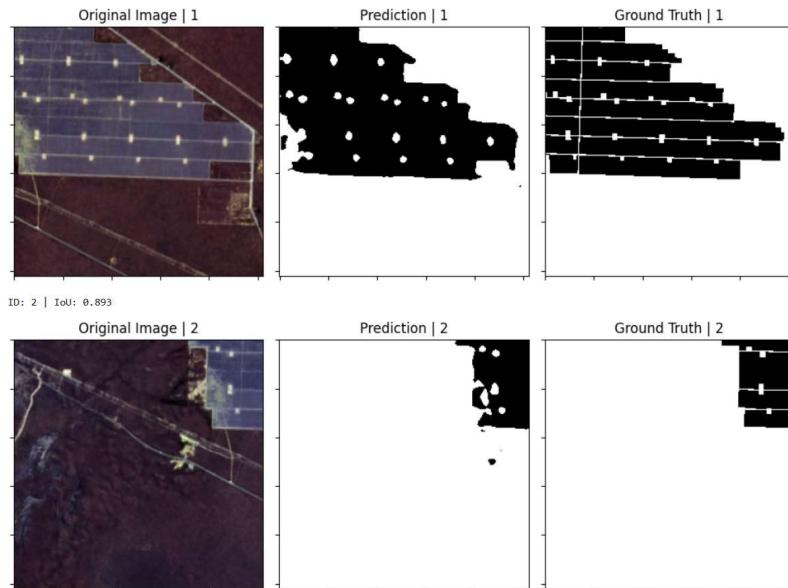
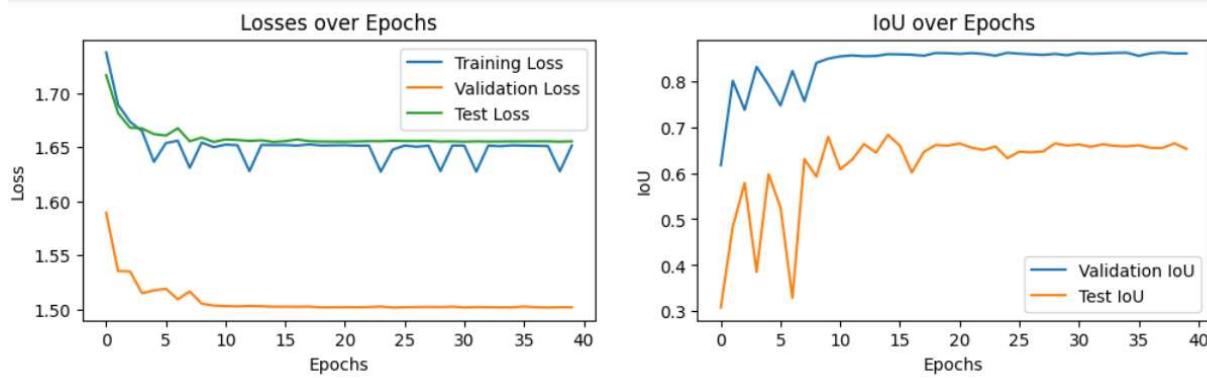
Begin working on the Loss function:

Change Loss Function: **BCE + Jaccard instead of Dice**

$\text{lr}=0.00001$



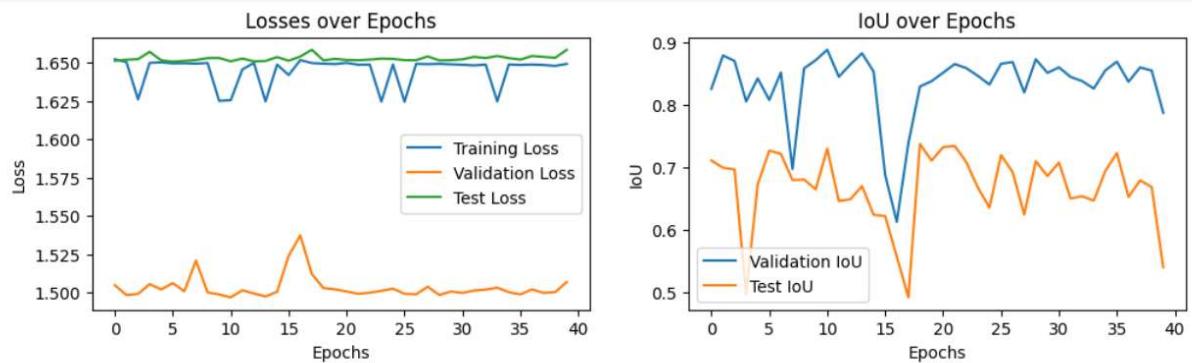
$\text{lr}=0.0005$



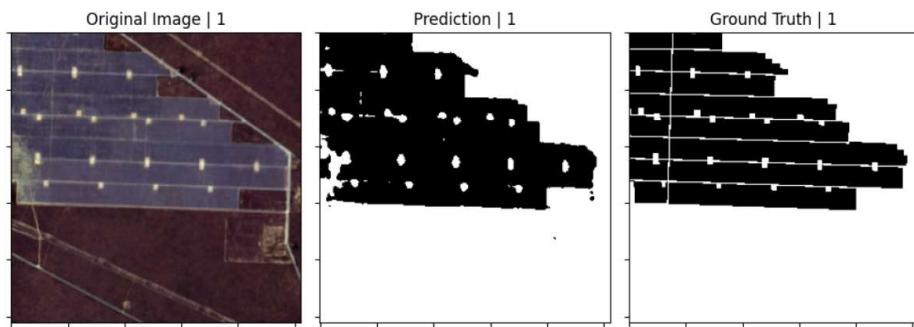
First noticeable improvements observed. The loss, IoU curves, and predictions begin to show more favorable values.

lr=0.0005

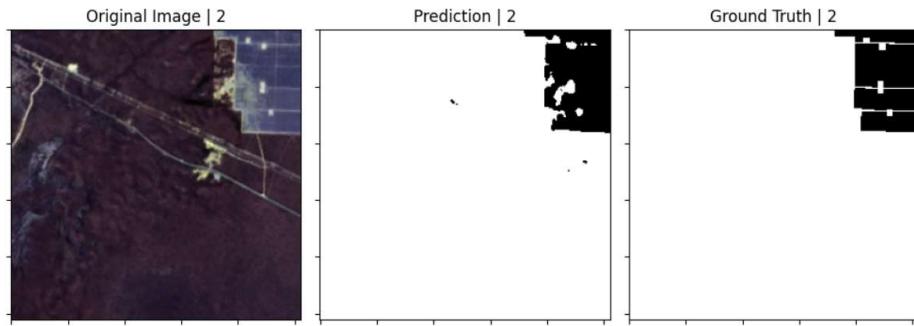
scheduler step\_size=100 (useless, in fact, the nb\_epochs were set to 40)



ID: 1 | IoU: 0.908



ID: 2 | IoU: 0.903

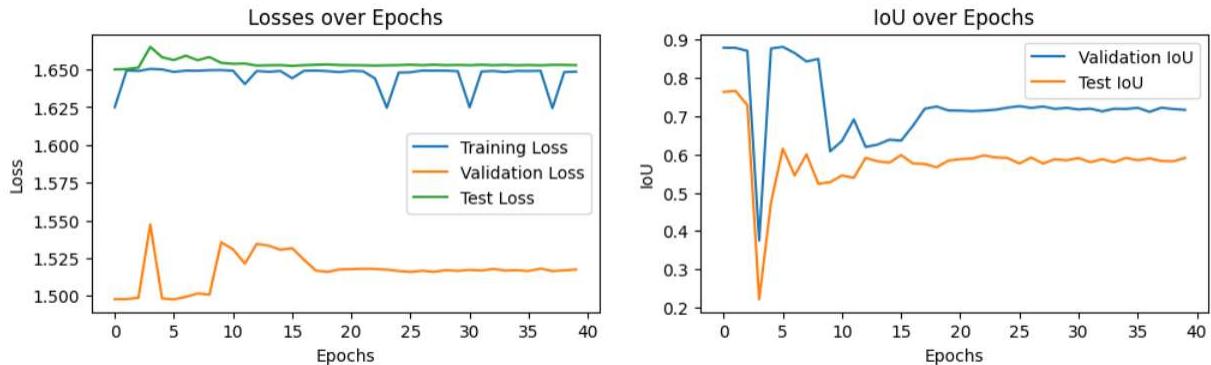


ID: 3 | IoU: 0.942

lr=0.0005

scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size=10, gamma=0.05)

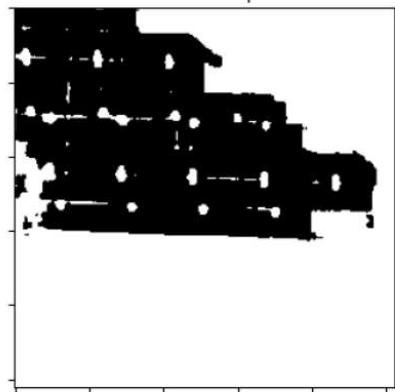
previously gamma was 0.1



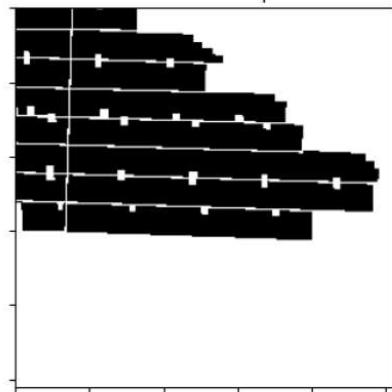
ID: 1 | IoU: 0.898



Prediction | 1



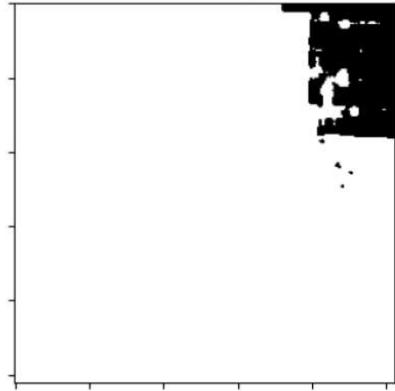
Ground Truth | 1



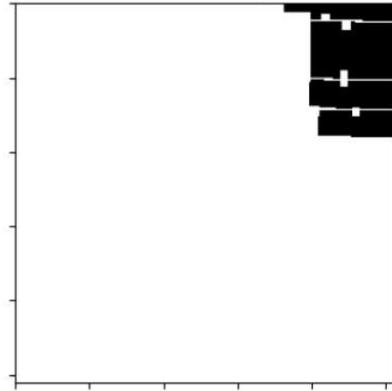
ID: 2 | IoU: 0.883



Prediction | 2



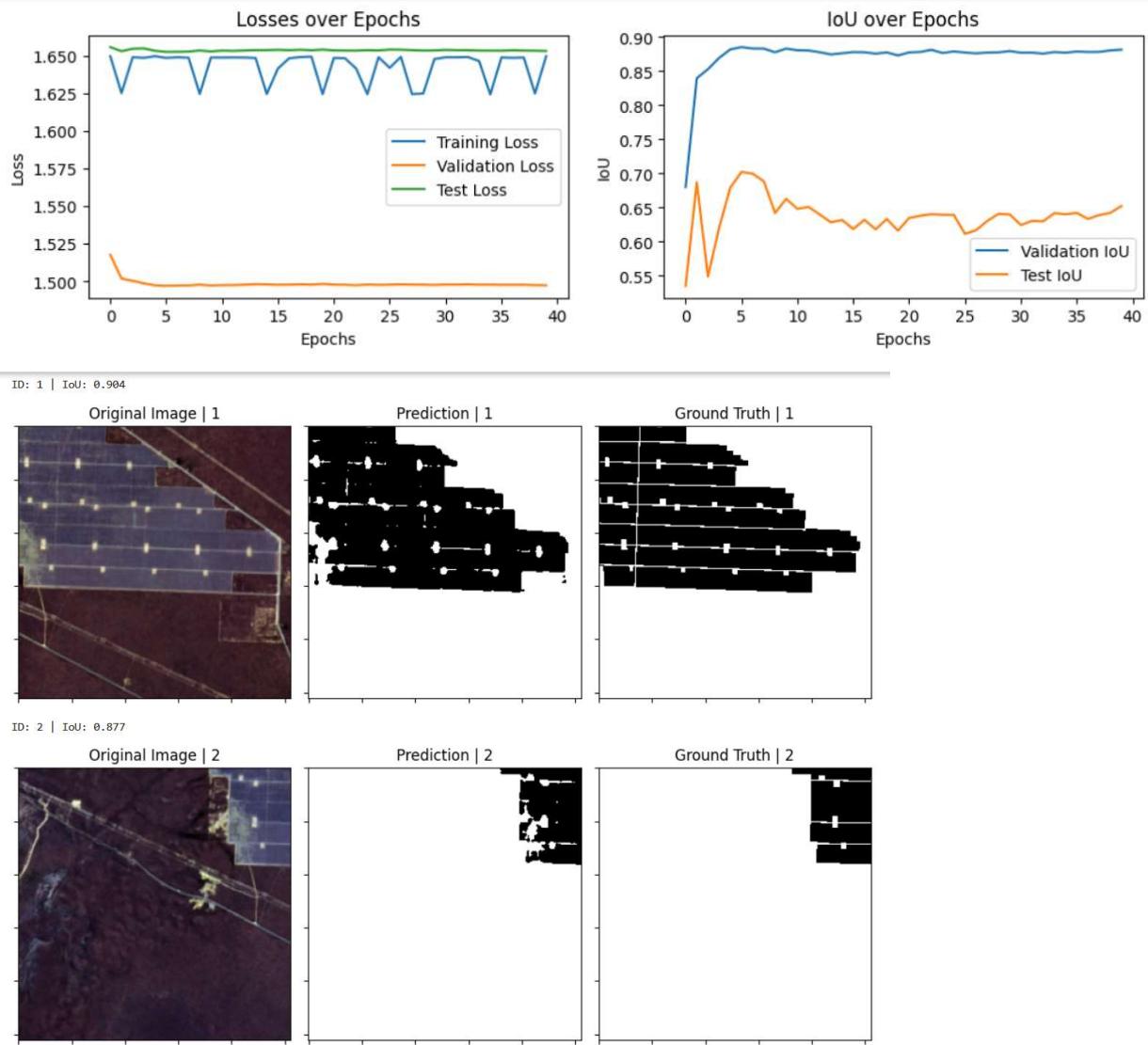
Ground Truth | 2



ID: 3 | IoU: 0.92

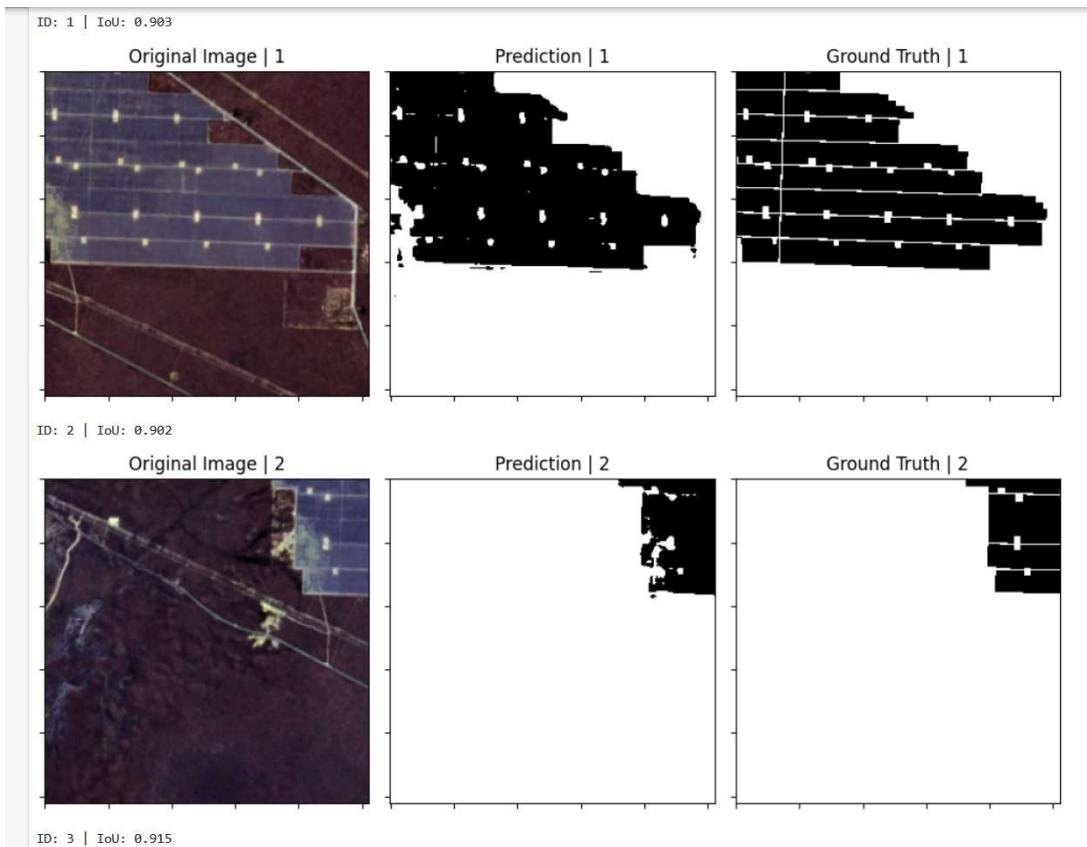
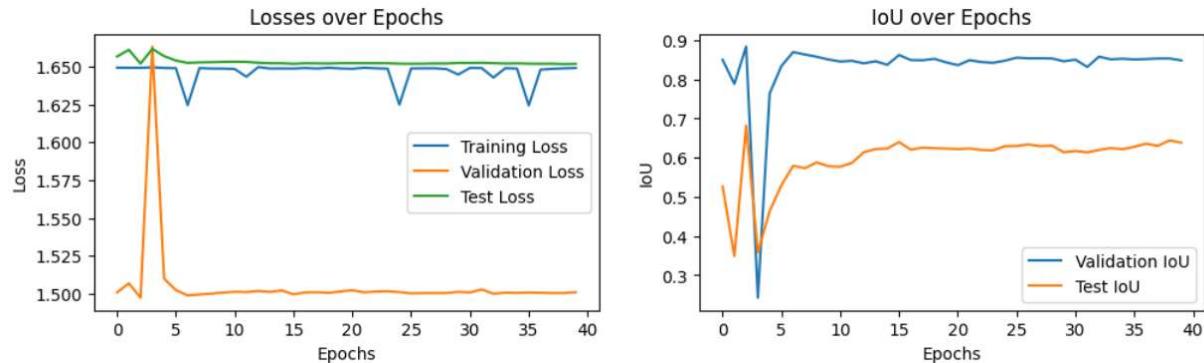
dict(params=model.parameters(), lr=0.0006),

scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size=5, gamma=0.1)



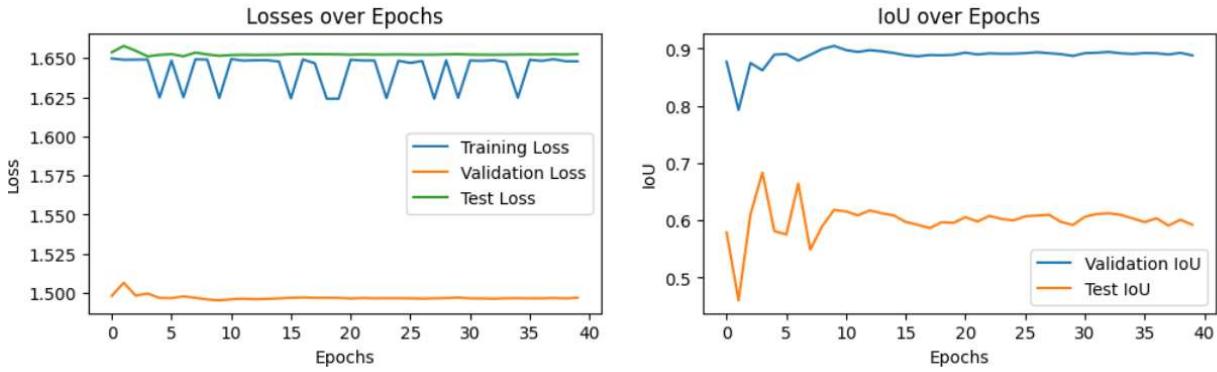
The previous charts showed a high overfitting effect.

```
dict(params=model.parameters(), lr=0.0005),  
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

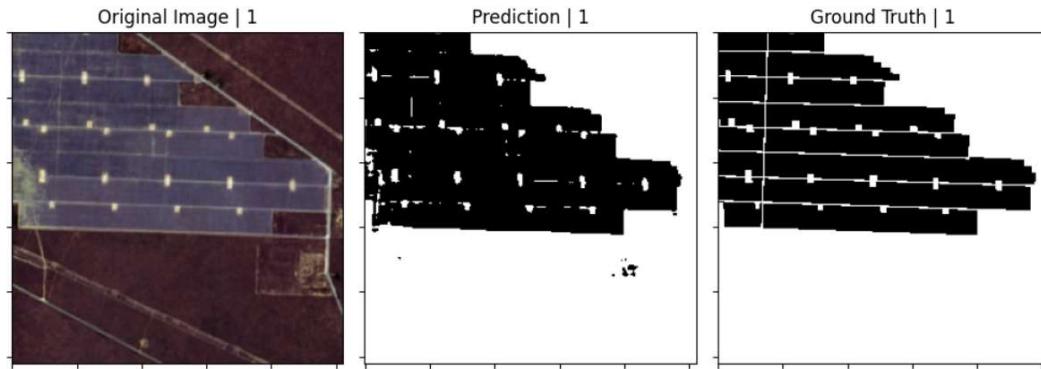


Return to the ‘base line’

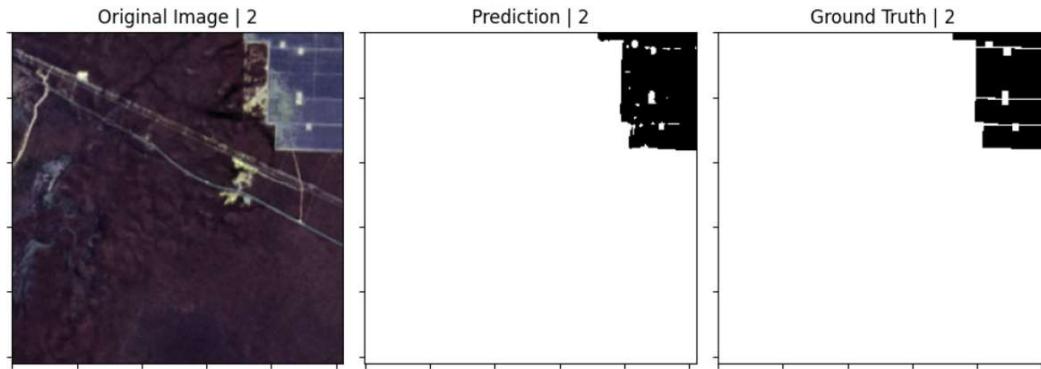
```
dict(params=model.parameters(), lr=0.0005),  
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```



ID: 1 | IoU: 0.921



ID: 2 | IoU: 0.937



ID: 3 | IoU: 0.938

## Data Augmentation:

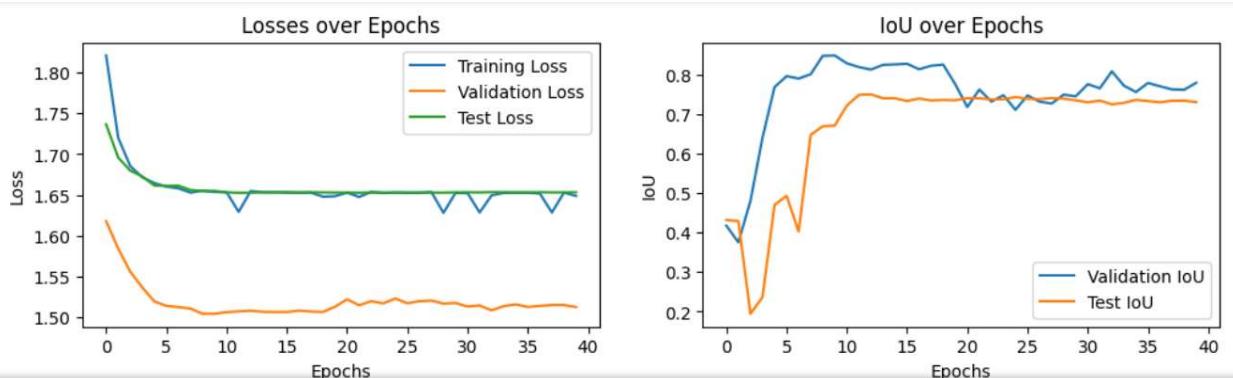
Let's apply Data Augmentation as a measure to treat overfitting.

```
simple_transform = T.Compose([
    T.ToTensor() # Only convert to tensor
])
augmentation_transform = transforms.Compose([
    transforms.ToTensor(),
    T.RandomHorizontalFlip(),
    T.RandomVerticalFlip(),
    T.RandomRotation(degrees=90),
])
train_dataset = CustomDataset(train_image_paths, train_mask_paths, transform=augmentation_transform,
                             transform_label=None)
#train_dataset = CustomDataset(train_image_paths, train_mask_paths, transform=transform,
                             transform_label=None)
val_dataset = CustomDataset(val_image_paths, val_mask_paths, transform=simple_transform,
                           transform_label=None)
test_dataset = CustomDataset(test_image_paths, test_mask_paths, transform=simple_transform,
                            transform_label=None)
```

I needed to correct 2 errors:

- 1) the transformations were **applied to the 3 sets**, that was wrong!
- 2) Using the previous code, the **transformations were not synchronously applied to images and masks**.

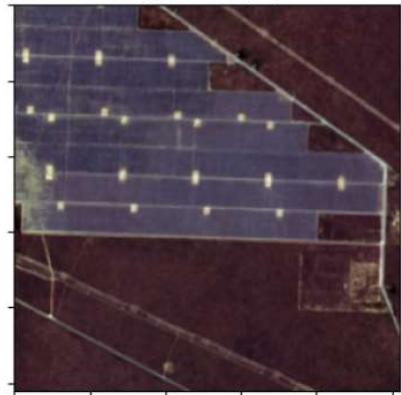
For the moment, we will **apply vertical and horizontal rotations and 90 degrees flip**.



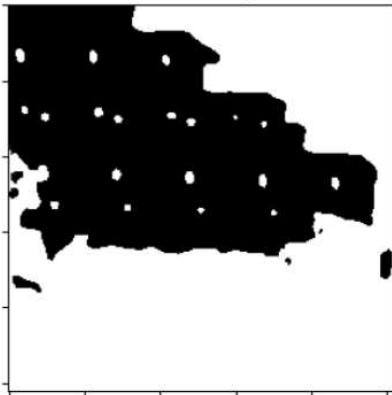
Data Augmentation improved Overfitting!

ID: 1 | IoU: 0.846

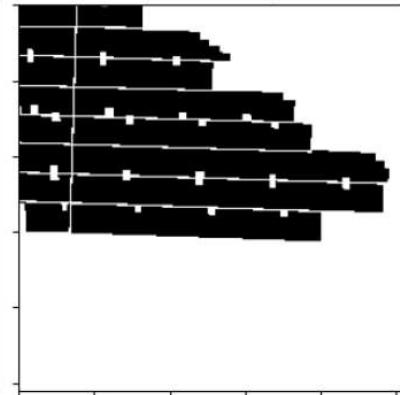
Original Image | 1



Prediction | 1

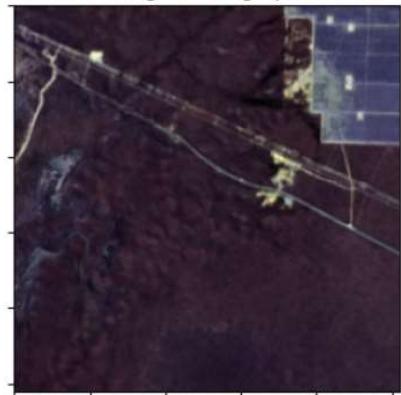


Ground Truth | 1

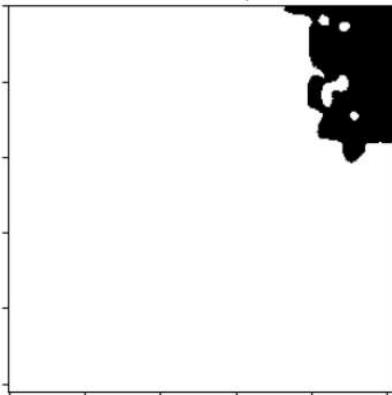


ID: 2 | IoU: 0.877

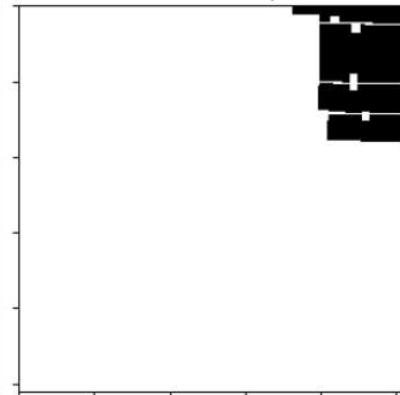
Original Image | 2



Prediction | 2



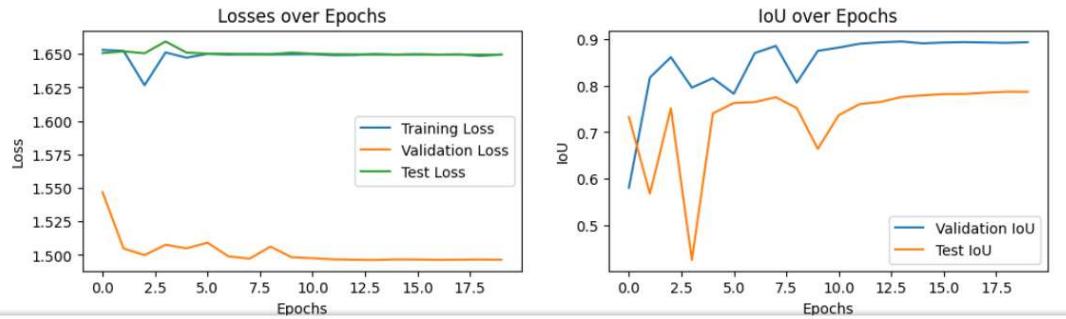
Ground Truth | 2



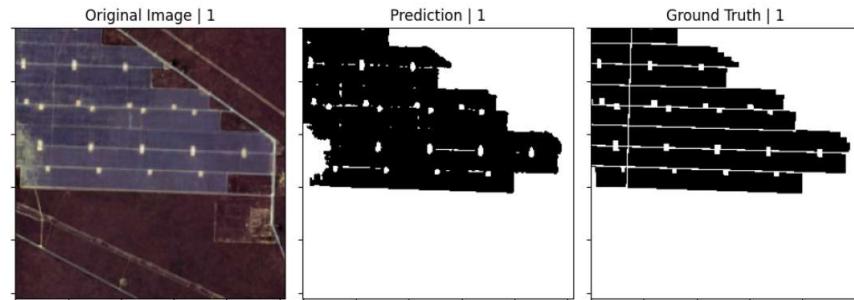
ID: 3 | IoU: 0.925

Let's check how LR adjustments impacts:

dict(params=model.parameters(), lr=0.0006), | 20 epochs

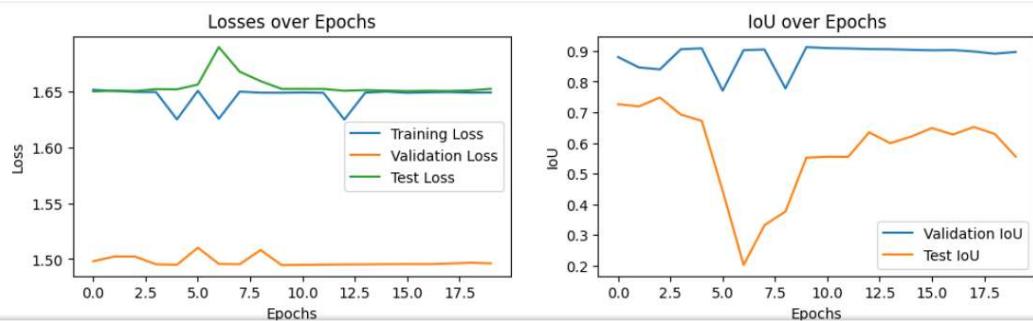


ID: 1 | IoU: 0.904

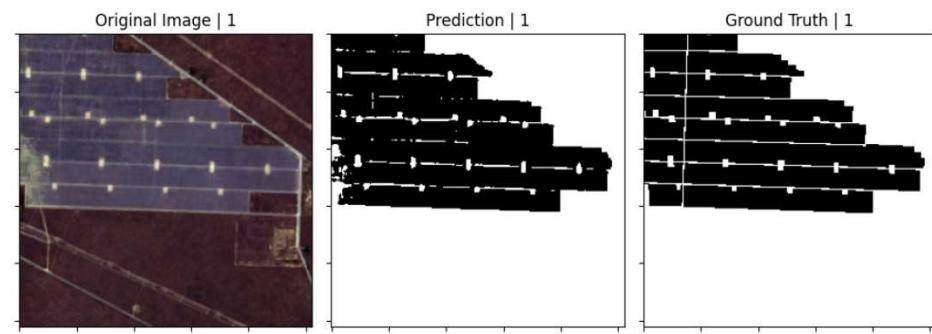


ID: 2 | IoU: 0.906

dict(params=model.parameters(), lr=0.0009) | 20 epochs

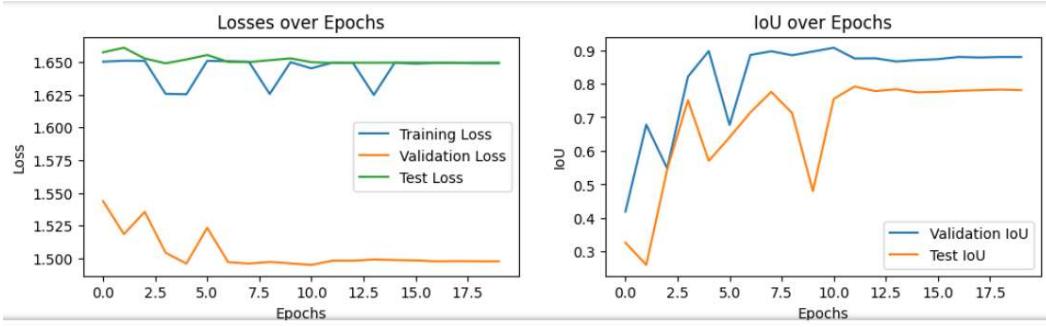


ID: 1 | IoU: 0.932

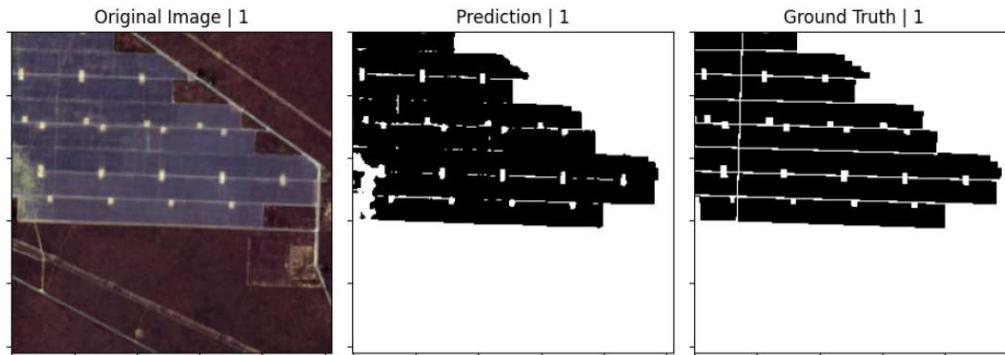


ID: 2 | IoU: 0.936

dict(params=model.parameters(), lr=0.0013) | 20 epochs

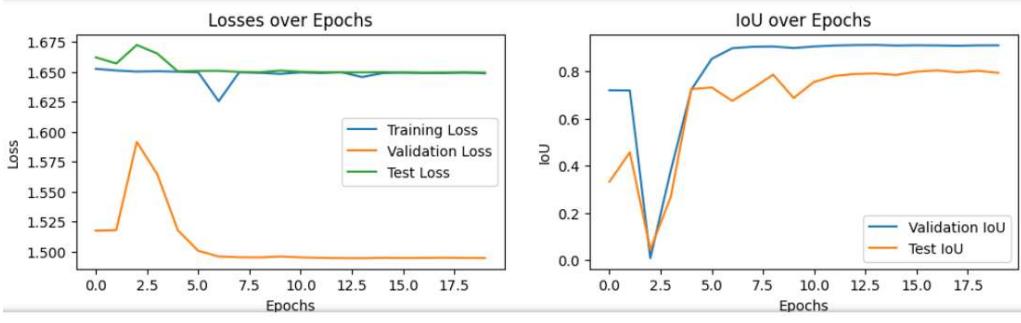


ID: 1 | IoU: 0.923

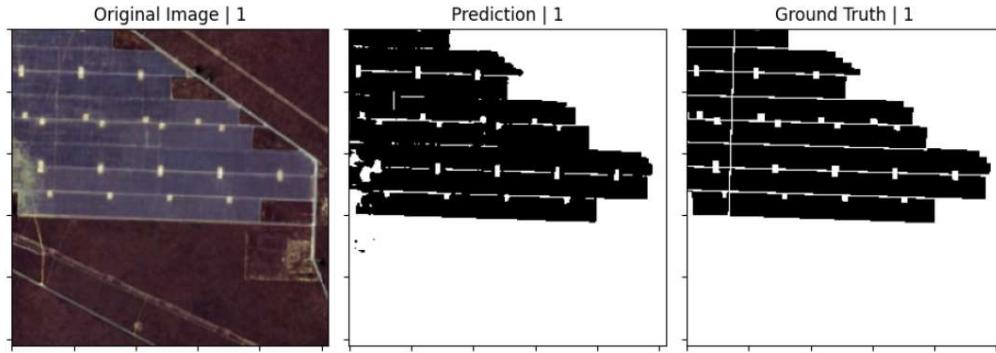


ID: 2 | IoU: 0.893

dict(params=model.parameters(), lr=0.003) | 20 epochs

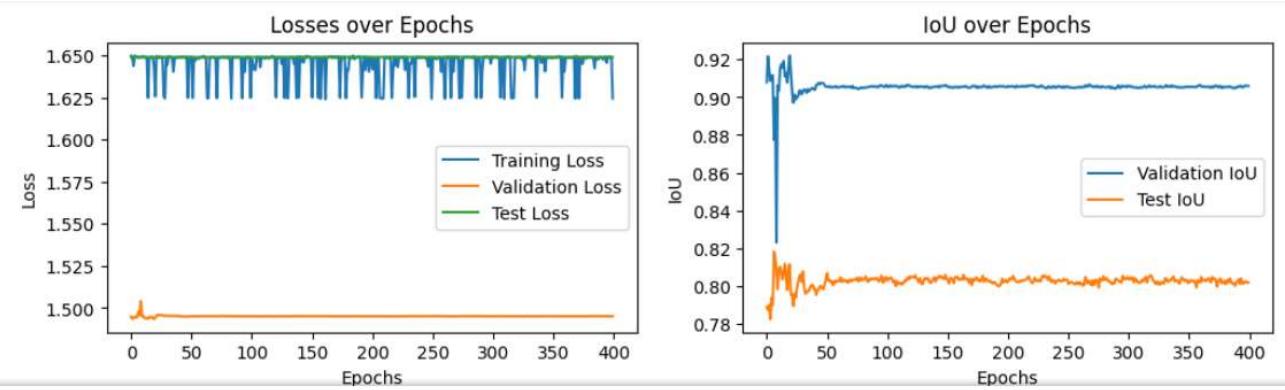


ID: 1 | IoU: 0.939

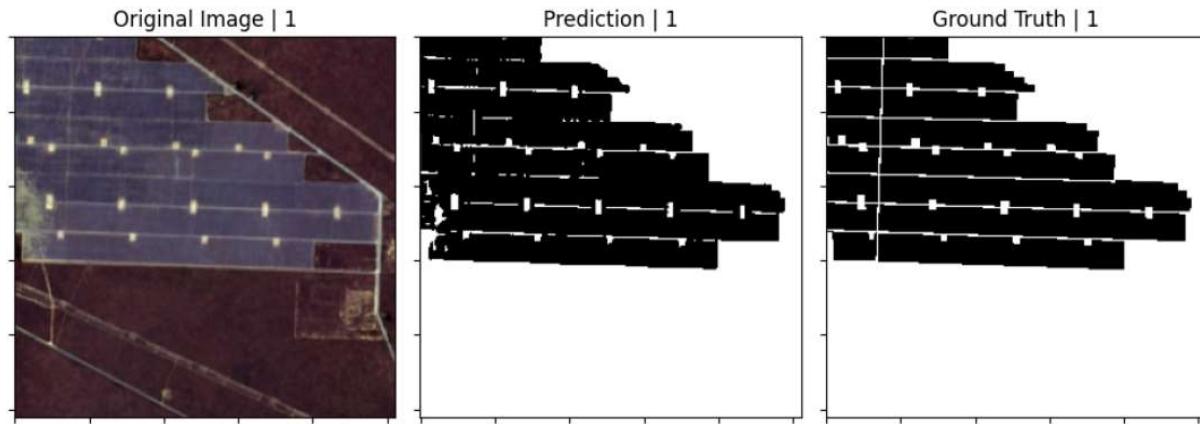


ID: 2 | IoU: 0.919

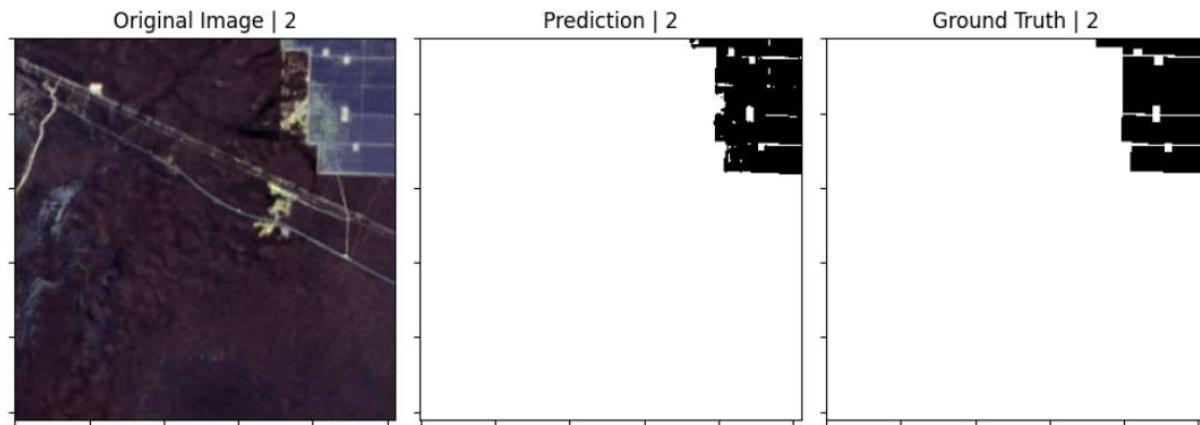
dict(params=model.parameters(), lr=0.0006), | 400 epochs



ID: 1 | IoU: 0.943



ID: 2 | IoU: 0.926

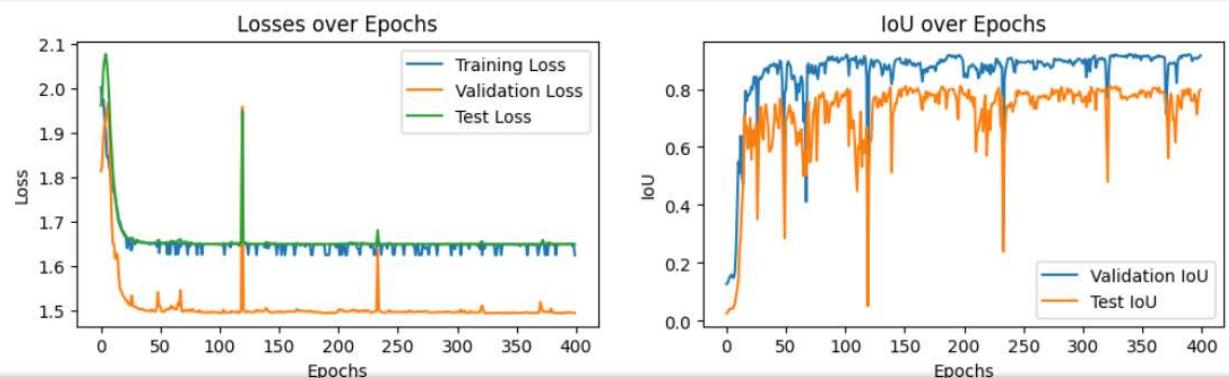


ID: 3 | IoU: 0.949

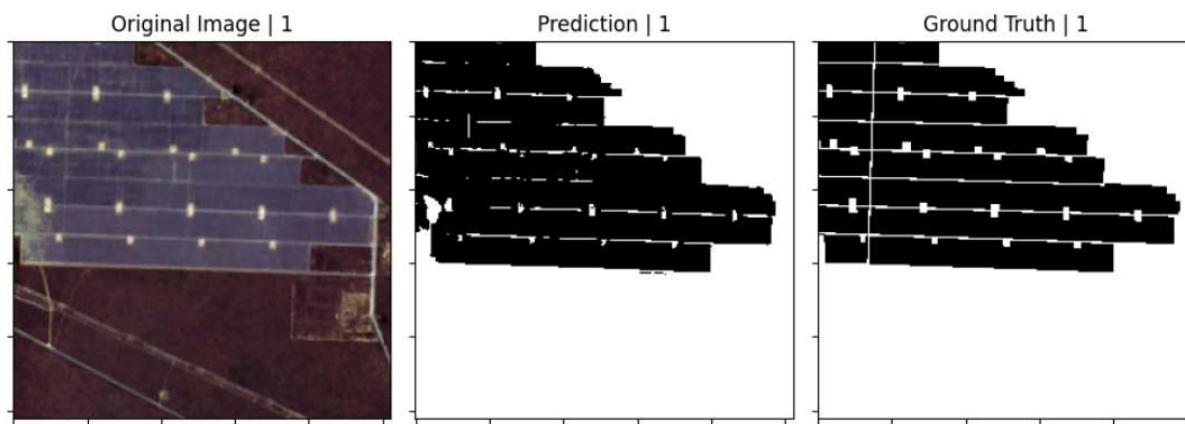
A lengthy training of 400 epochs revealed overfitting, although the predictions appear sharper and more defined.

Add a **Warmup** to the Learning Rate Scheduler

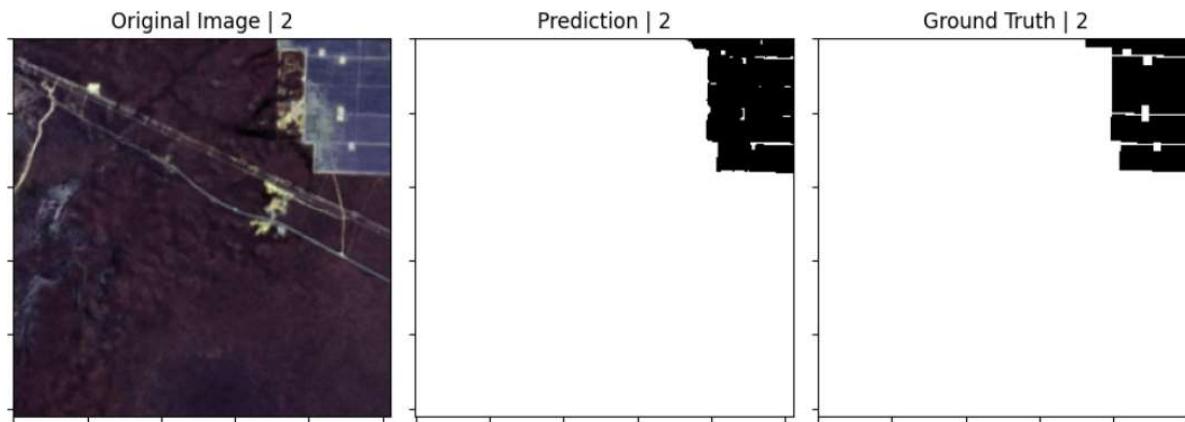
Using a Warmup helps the model to converge better and avoiding peaks at the beginning of the training.



ID: 1 | IoU: 0.934



ID: 2 | IoU: 0.918



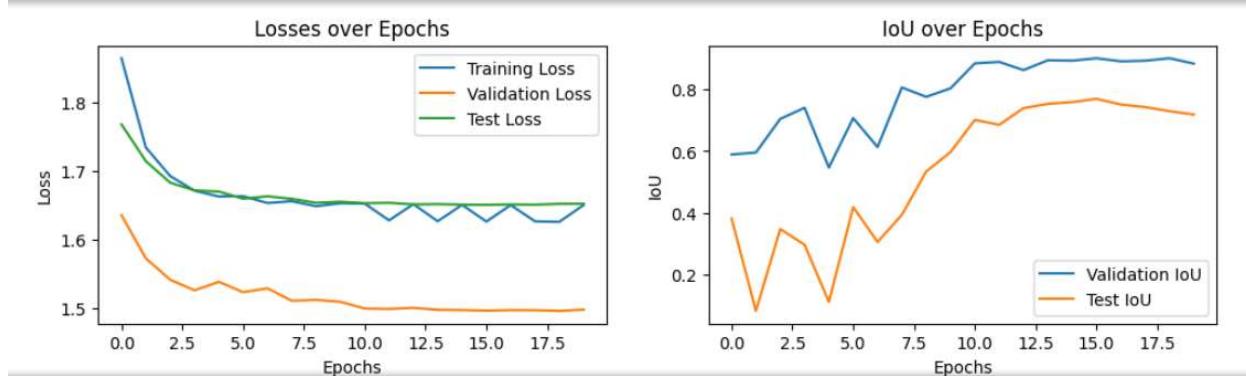
ID: 3 | IoU: 0.948

## Batch Size Optimization:

- 1) Not using Warmup
- 2) Changing batch sizes.

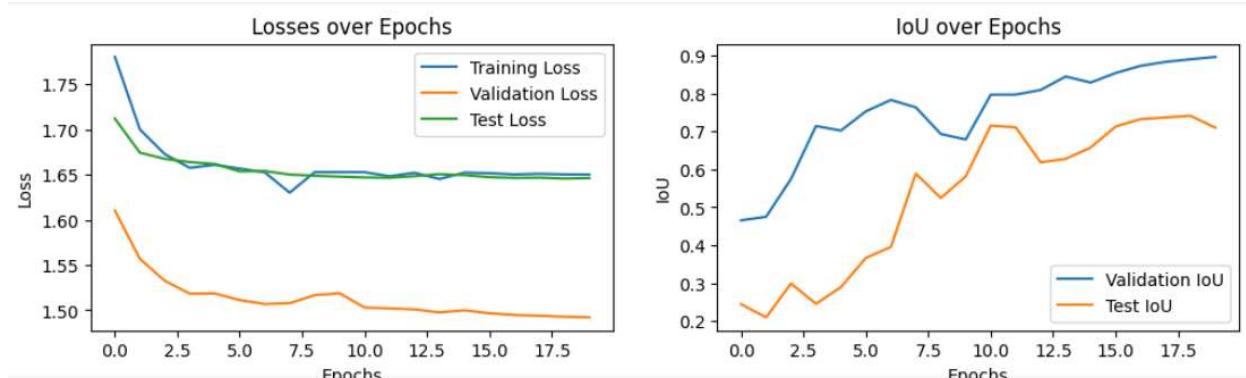
First, with previous values. Batch sizes: train=5, valid=40, test=30.

20 epochs

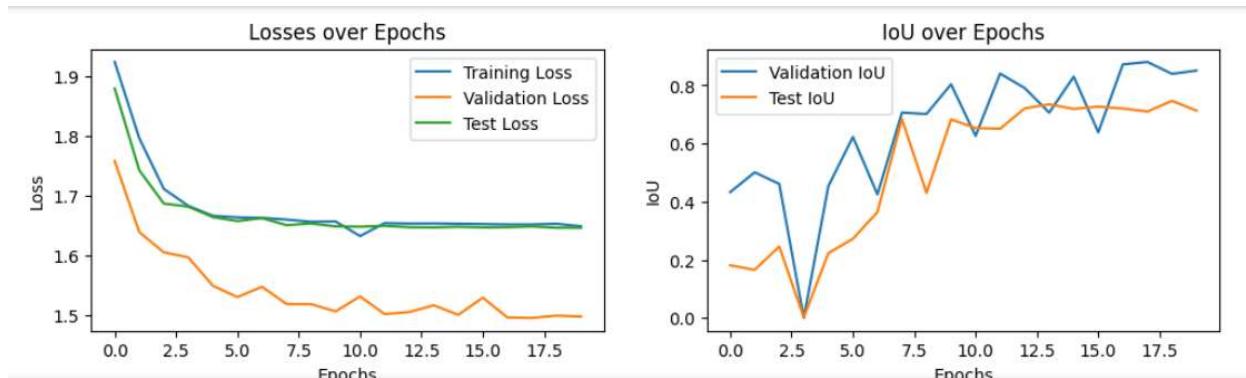


New values: train=5, valid=10, test=10

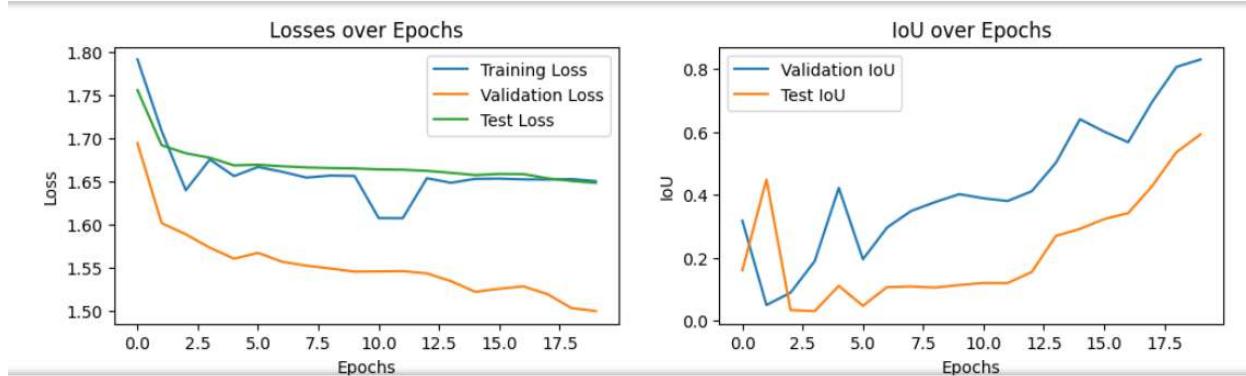
First time:



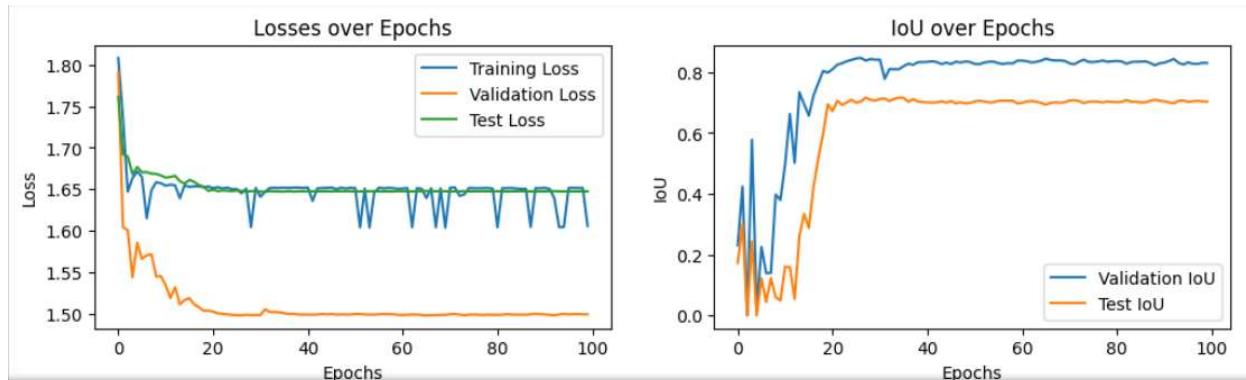
Second time:



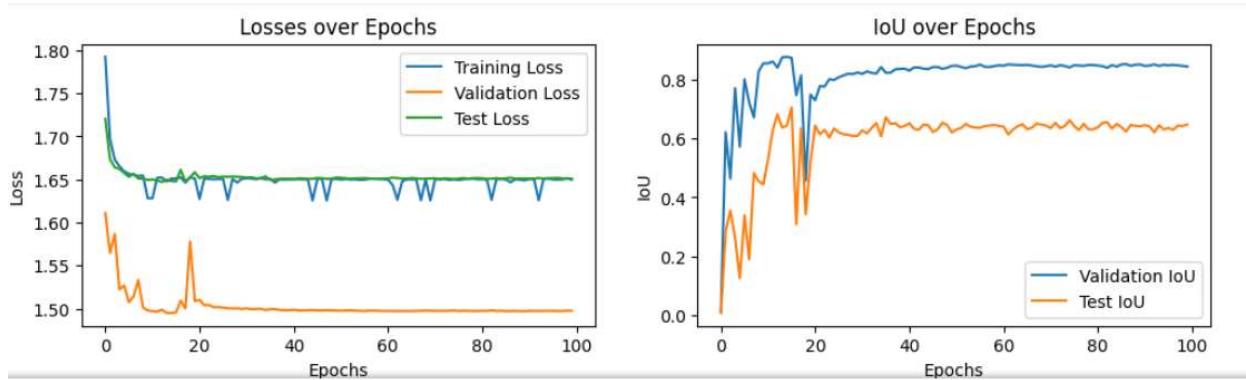
New values: train=10, valid=10, test=10



New values: train=10, valid=10, test=10 with 100 epochs



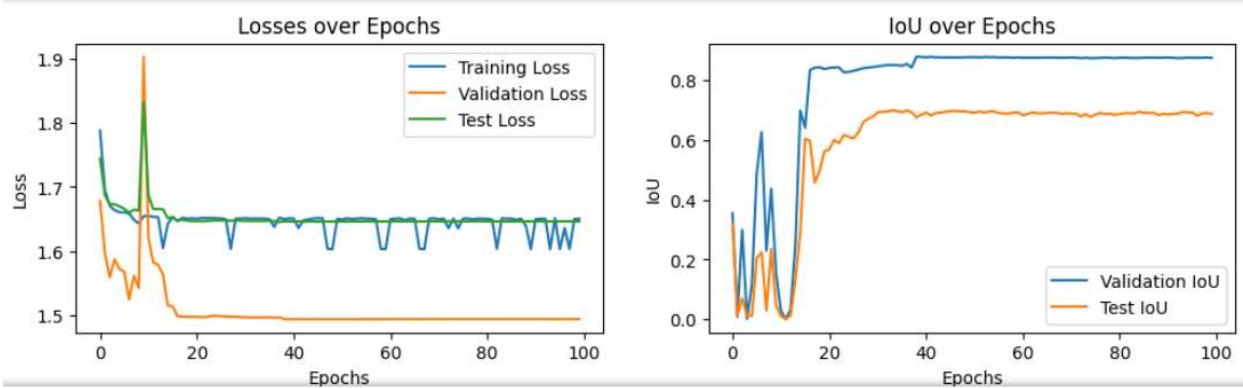
New values: train=5, valid=10, test=10 with 100 epochs



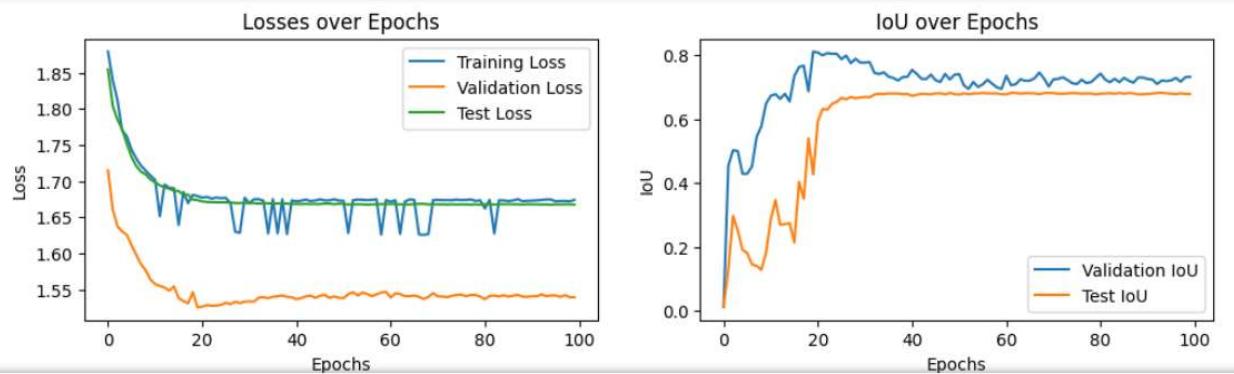
New values: train=10, valid=10, test=10 with 100 epochs

LR = 0.001

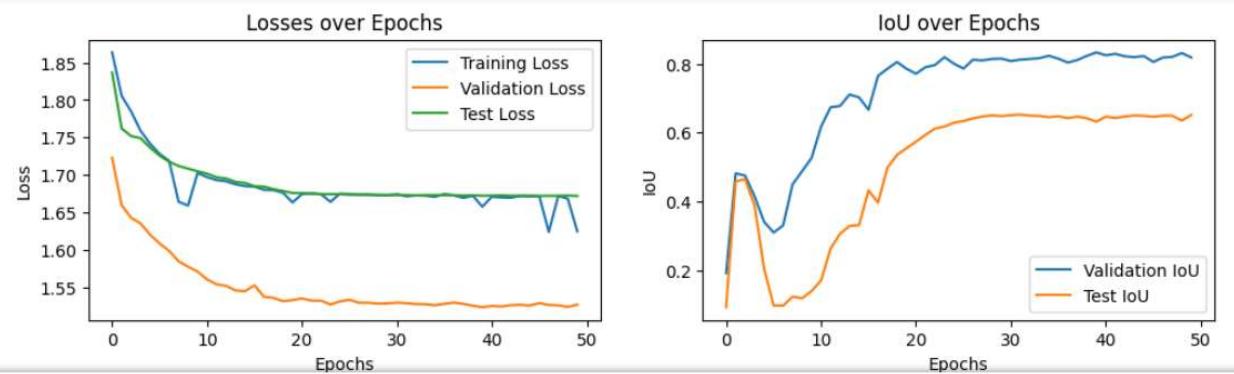
with scheduler = `torch.optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)`



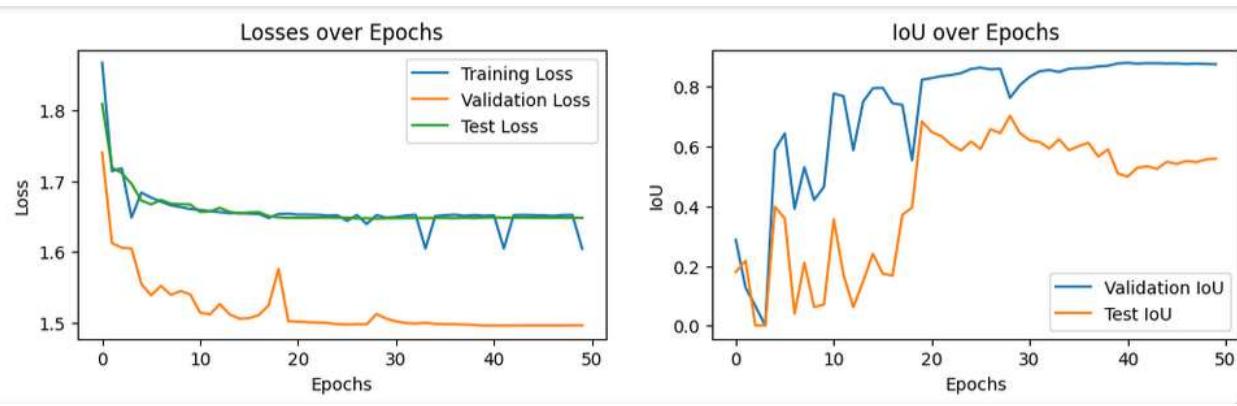
New values: train=10, valid=10, test=10 with 100 epochs | LR = 0.0001



Batch sizes: train=10, valid=40, test=30 with | 50 epochs | LR = 0.0001



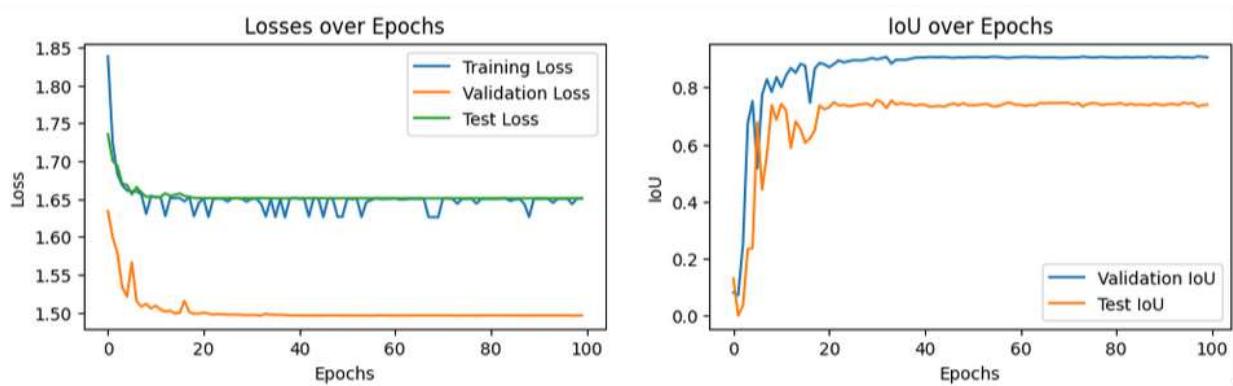
Batch sizes: train=10, valid=10, test=10 with | 50 epochs | LR = 0.0006



Important: repeat the exercise and understand the training log.

**Let's return to the original batch sizes, because it seems that there's no evident gains.**

Batch sizes: train=5, valid=40, test=30 with | 100 epochs| LR = 0.0006



Despite the overfitting, the results were good.

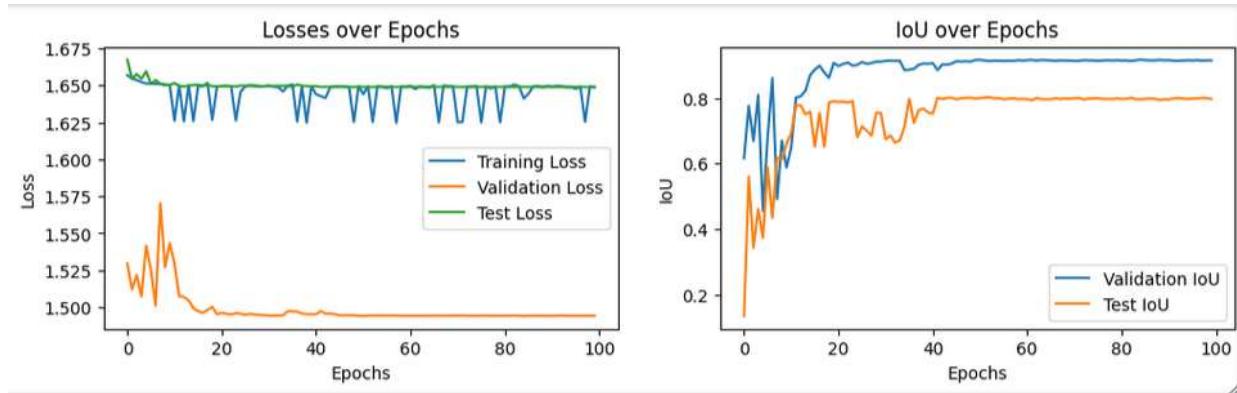
Mange the Learning Rate:

Let's use a **ReduceLR on Plateau**

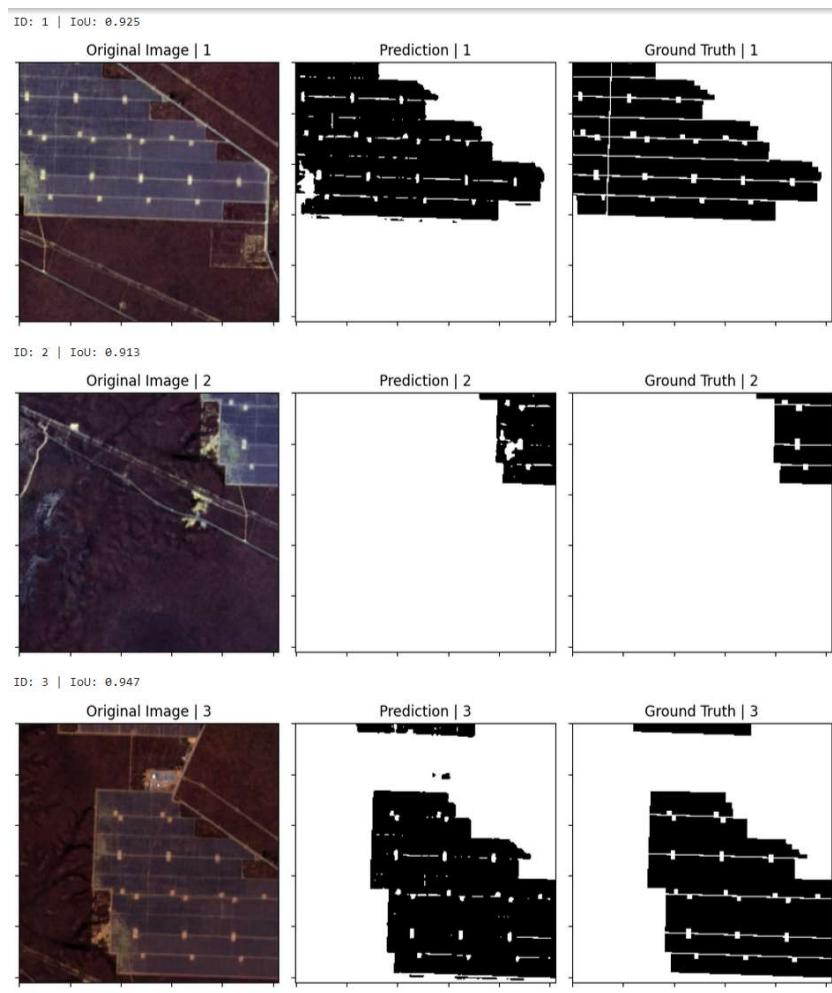
```
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=5, verbose=True)
```

criteria to trigger the LR reduction: scheduler.step(**valid\_loss**)

Batch sizes: train=5, valid=40, test=30 with | 100 epochs| LR = 0.0006

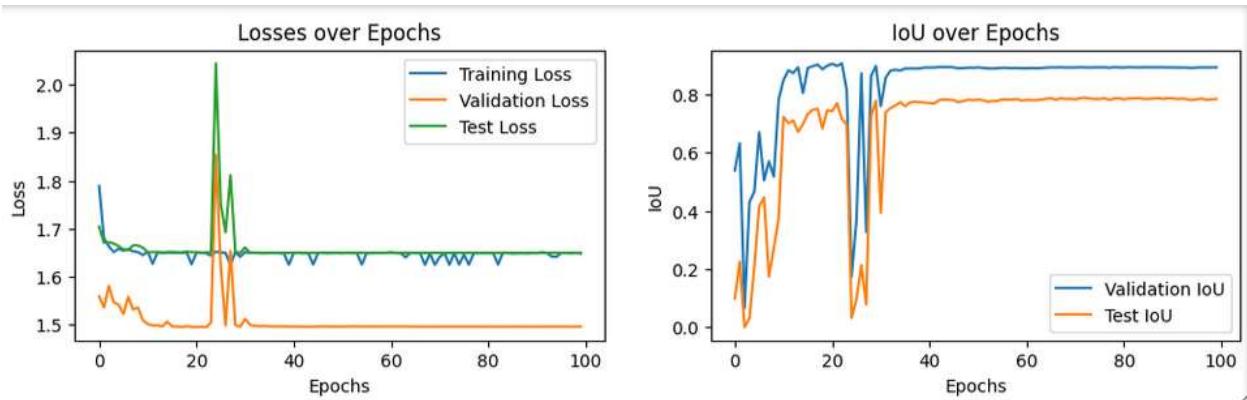


Best IoU score until now and crisper predictions:

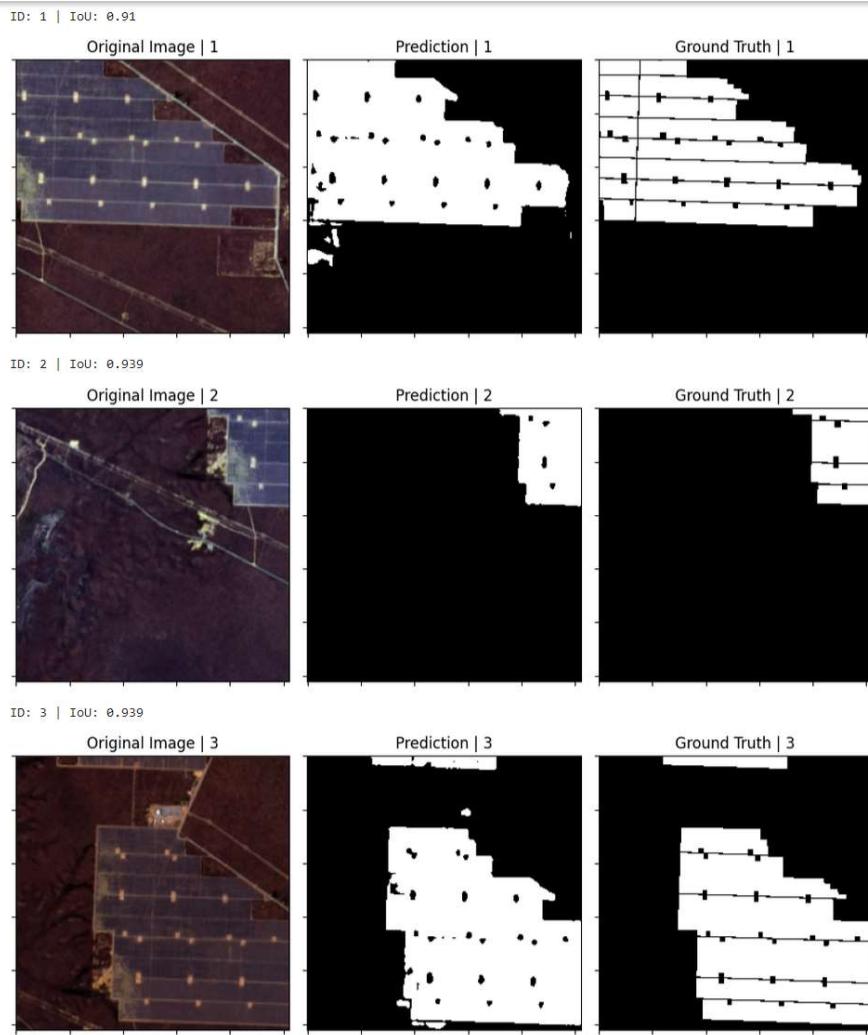


criteria to trigger the LR reduction: scheduler.step(`valid_loss`)

Batch sizes: train=5, valid=40, test=30 with | 100 epochs| LR = 0.001



Sample IoU Score: 0.7977288365364075



## Experiencing with other models:

Apart from U-net other Models that also work well for semantic segmentation

Model	Strengths	Notes
DeepLabV3+	Multi-scale context, detailed segmentation	Good for varying object sizes
UNet++	Enhanced gradient flow, better detail accuracy	Slightly more complex than U-Net

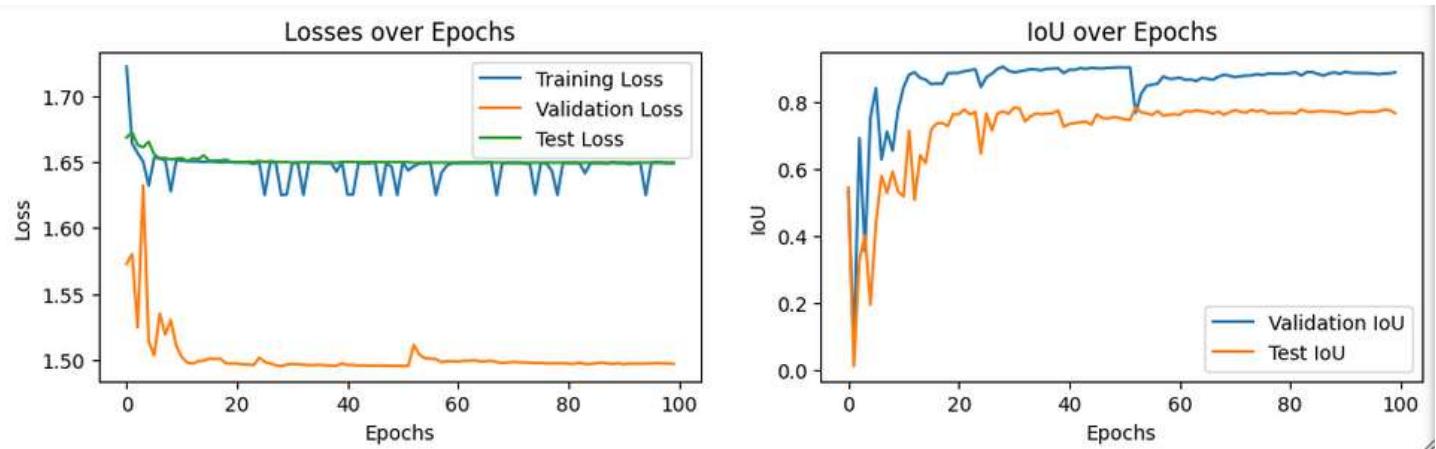
### DeepLabV3+

Problems with **downsampling**. The model's input dimensions are too small, leading to a reduction in spatial dimensions to 1x1 at some point in the model.

## Switch model to Unet++

criteria to trigger the LR reduction: scheduler.step(**valid\_loss**)

Batch sizes: train=5, valid=40, test=30 with | 100 epochs| LR = 0.001



Thresholds: 25% | 50% | 75%

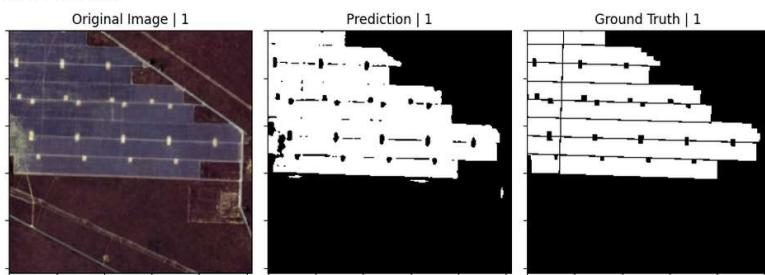
Accuracy: 99.09 | 99.06 | 99.02

IoU: 81.83 | 81.16 | 80.29

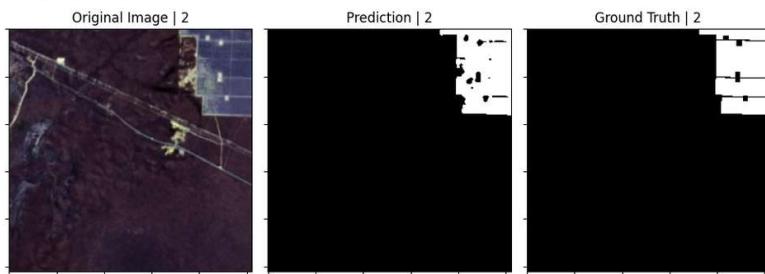
F-score: 90.01 | 89.6 | 89.07

Sample IoU Score: 0.8116099238395691

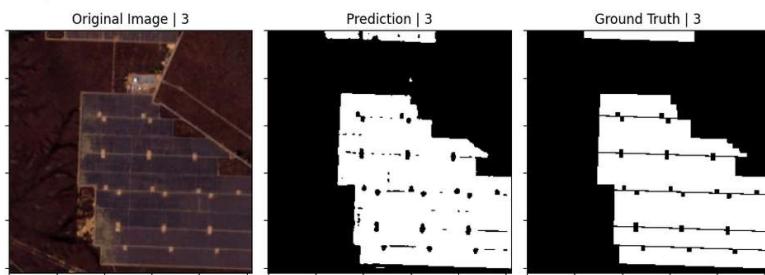
ID: 1 | IoU: 0.919



ID: 2 | IoU: 0.925



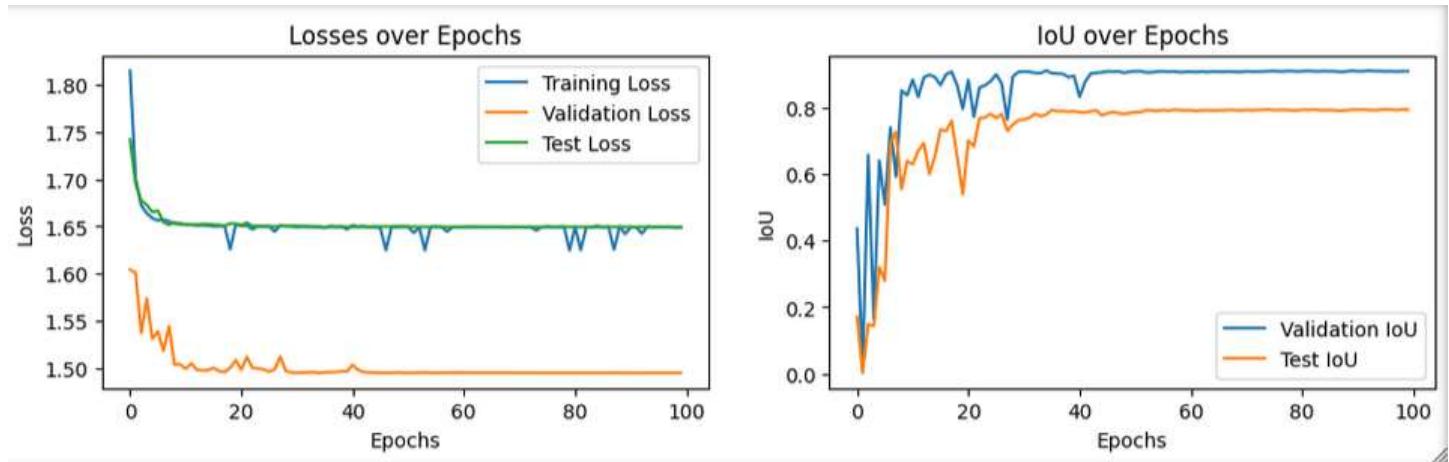
ID: 3 | IoU: 0.954



## Switch model to Unet++

criteria to trigger the LR reduction: scheduler.step(**valid\_loss**)

Batch sizes: train=5, valid=40, test=30 with | 100 epochs| LR = 0.00006



Thresholds: 25% | 50% | 75%

Accuracy: 99.14 | 99.12 | 99.09

IoU: 82.82 | 82.27 | 81.52

F-score: 90.61 | 90.27 | 89.82

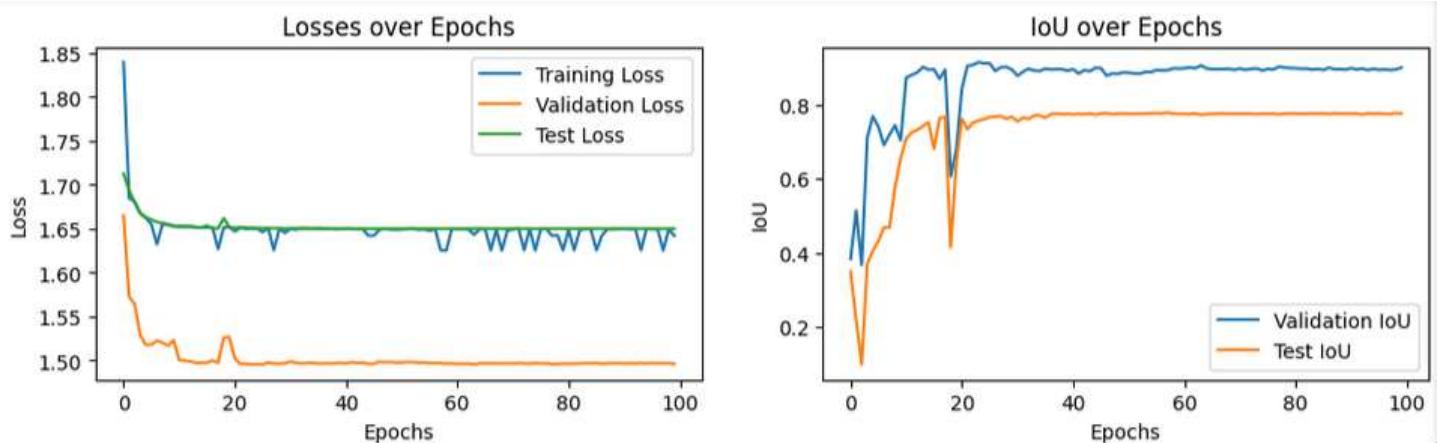
Sample IoU Score: 0.8226526975631714



## Compare with Unet with same hyperparams

criteria to trigger the LR reduction: scheduler.step(**valid\_loss**)

Batch sizes: train=5, valid=40, test=30 with | 100 epochs| LR = 0.00006



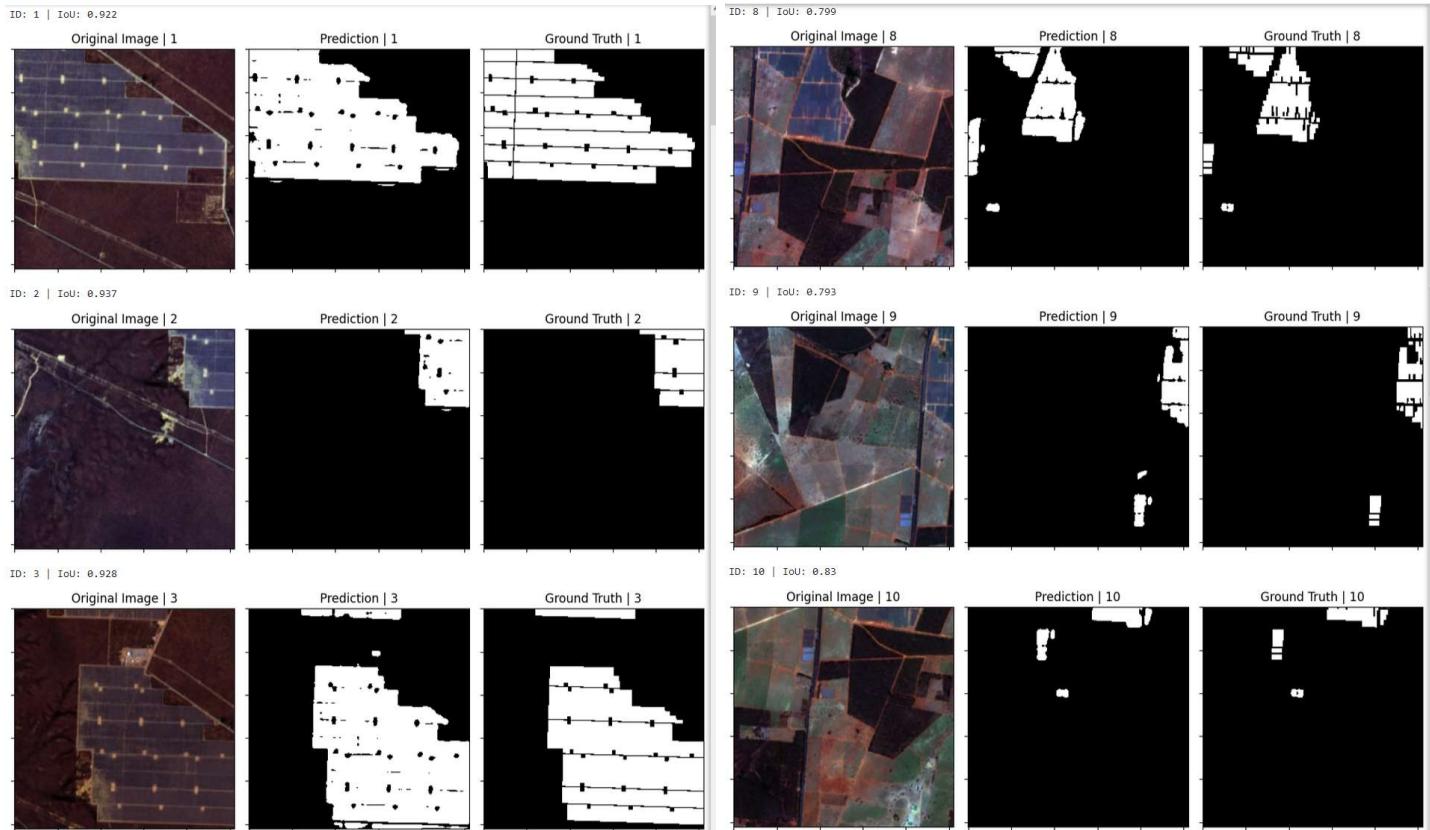
Thresholds: 25% | 50% | 75%

Accuracy: 99.03 | 98.97 | 98.91

IoU: 80.51 | 79.28 | 77.82

F-score: 89.2 | 88.44 | 87.53

Sample IoU Score: 0.792826831340789



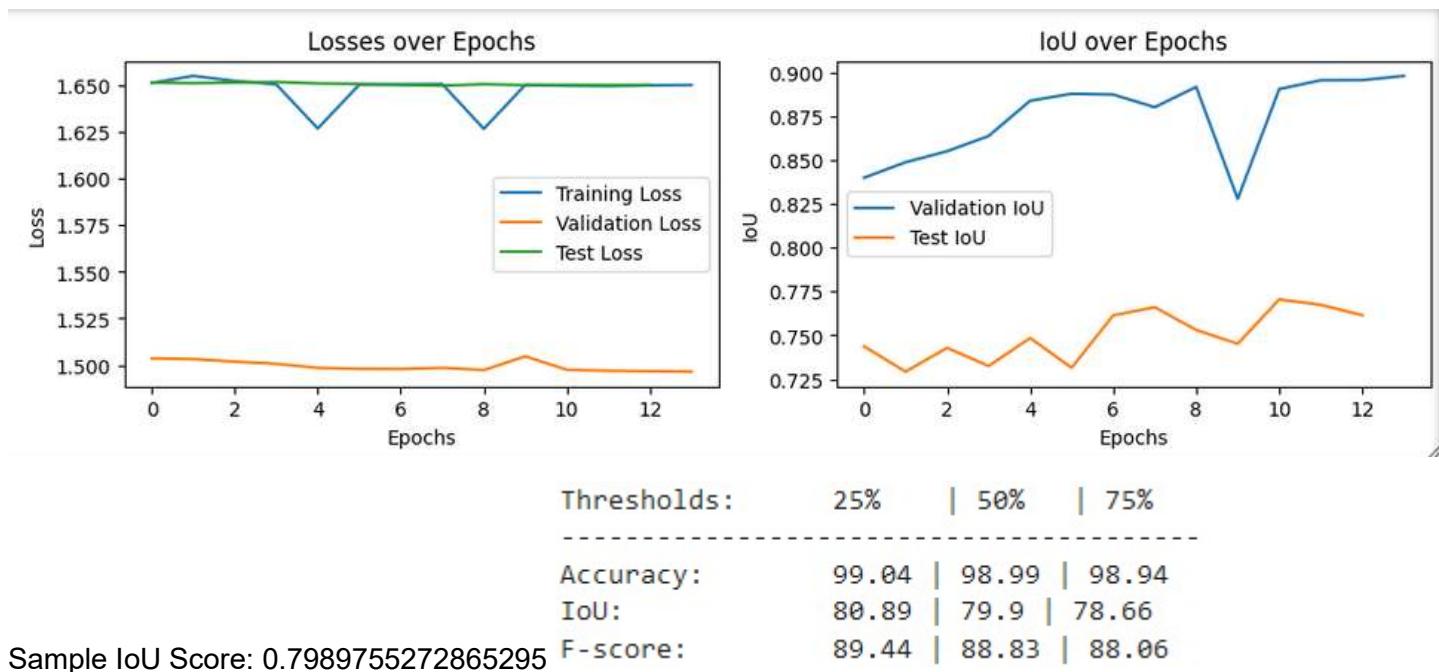
## Experiencing with Early Stopping:

A technique used to prevent overfitting during training by stopping the training process if the model's performance on a validation set does not improve after a certain number of epochs (known as "patience"). This is especially useful when training deep learning models, as it saves computation time 🚀 and can result in better generalization.

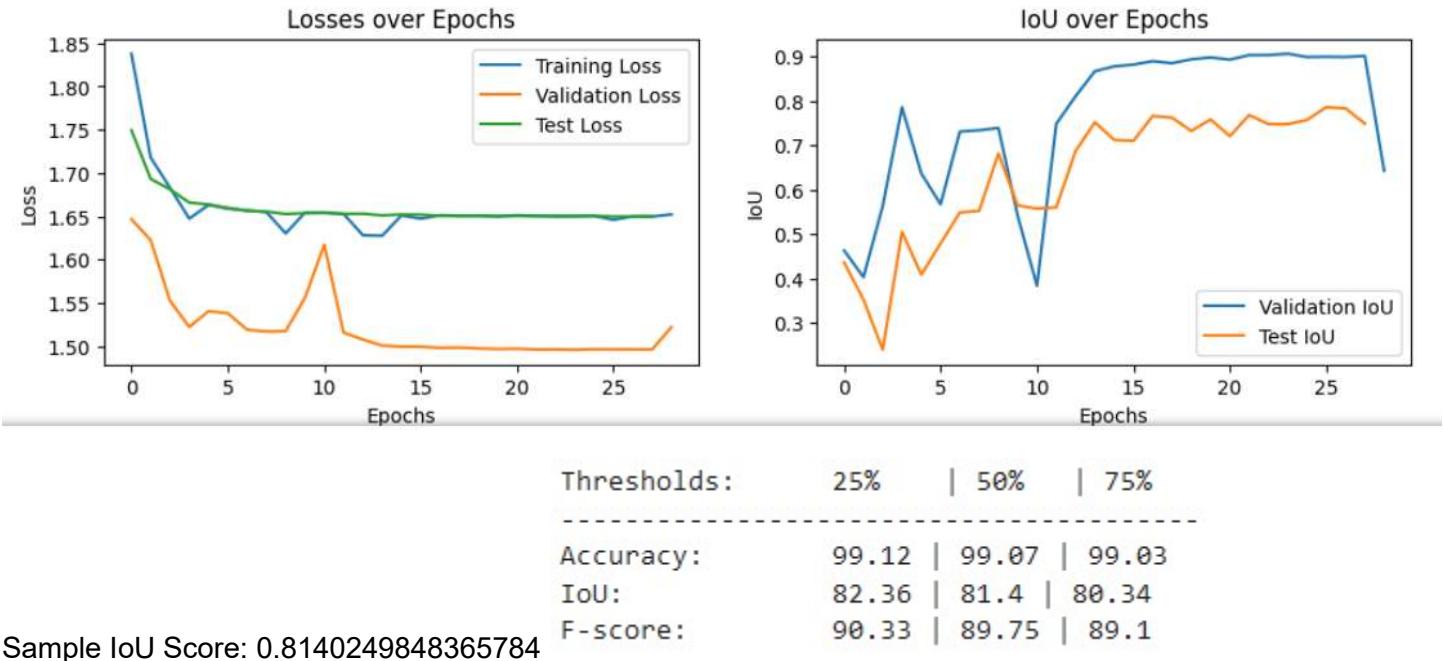
```
early_stopping = EarlyStopping(patience=5, min_delta=0.001)
```

min\_delta: minimum change in the metric to qualify as an improvement. This helps ignore tiny fluctuations.

Batch sizes: train=5, valid=40, test=30 with | 400 epochs| LR = 0.0006



```
early_stopping = EarlyStopping(patience=5, min_delta=0.0001)
```



```
early_stopping = EarlyStopping(patience=10, min_delta=0.0001)
```

patience: Number of epochs to wait without improvement before stopping.



## Experiencing with Optimizers:

**Unet + Adam** : Sample IoU Score: 0.7911292910575867

**Unet + AdamW** : Sample IoU Score: 0.7970814108848572

**Unetpp + Adam** : Sample IoU Score: 0.791957676410675

**Unetpp + AdamW** : Sample IoU Score: 0.8352243900299072

Change Batches' sizes: Train size=5, valid size=14, test size=22

**Unetpp + AdamW** : Sample IoU Score: 0.8157026767730713

Change Batches' sizes: Train size=5, valid size=10, test size=10

**Unetpp + AdamW** : Sample IoU Score: 0.787004828453064

Change Batches' sizes: Train size=5, valid size=40, test size=30

**Unetpp + AdamW** : Sample IoU Score: 0.826032280921936

Thresholds:	25%		50%		75%
<hr/>					
Accuracy:	99.16		99.13		99.08
IoU:	83.41		82.6		81.52
F-score:	90.95		90.47		89.82

## Improving the Loss function:

### Try another loss function and add weights:

- Add the **Focal Loss** to the loss functions. Using a combination of BCE, Focal and Jaccard should help improve the model's handling of class imbalances and enhance IoU performance.

Focal Loss is a specialized loss function often used in cases where there is a **class imbalance**, meaning one class (like the background) is much more common than the other (like a specific object or feature in satellite images). It modifies the standard binary cross-entropy (BCE) loss by focusing more on hard-to-classify examples and down-weighting easy examples. This can be helpful for semantic segmentation of satellite imagery where the objects of interest (e.g., solar panels) might occupy a small fraction of the total area, resulting in many easy-to-classify background pixels.

**baseline:** Unetpp + AdamW :Sample IoU Score: 0.826032280921936

Thresholds:	25%		50%		75%
Accuracy:	99.16		99.13		99.08
IoU:	83.41		82.6		81.52
F-score:	90.95		90.47		89.82

### Tests:

#### 1) bce\_loss + jaccard\_loss + focal\_loss

IoU Score: 0.8238301277160645,

epochs : 26

Thresholds:	25%		50%		75%
Accuracy:	99.15		99.12		99.04
IoU:	83.14		82.38		80.68
F-score:	90.8		90.34		89.31

#### 2) **0.2 \* bce\_loss + 0.3 \* jaccard\_loss + 0.5 \* focal\_loss**

IoU Score: 0.797029435634613

epochs :36

Thresholds:	25%		50%		75%
Accuracy:	99.06		99.0		98.93
IoU:	80.99		79.7		78.27
F-score:	89.5		88.71		87.81

#### 3) **0.3 \* bce\_loss + 0.5 \* jaccard\_loss + 0.2 \* focal\_loss**

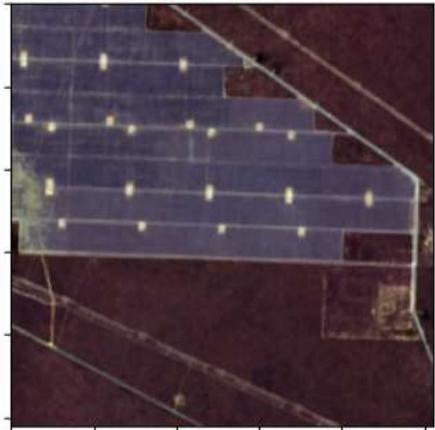
IoU Score: **0.8290086388587952**

epochs :41

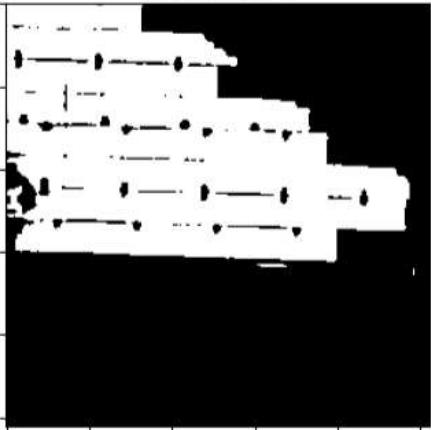
Thresholds:	25%		50%		75%
Accuracy:	99.16		99.14		99.11
IoU:	83.41		82.9		82.1
F-score:	90.95		90.65		90.17

ID: 1 | IoU: 0.925

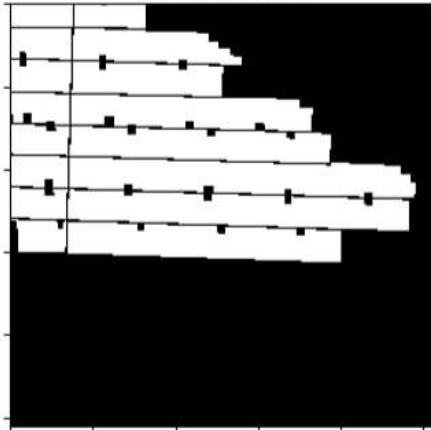
Original Image | 1



Prediction | 1

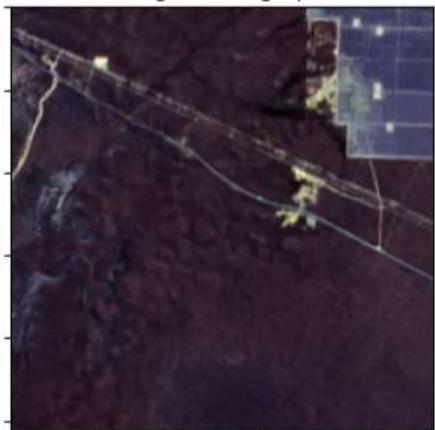


Ground Truth | 1



ID: 2 | IoU: 0.936

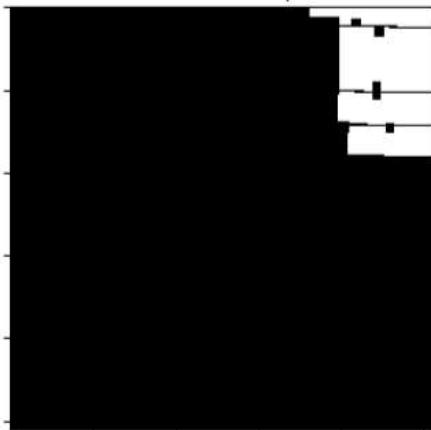
Original Image | 2



Prediction | 2



Ground Truth | 2

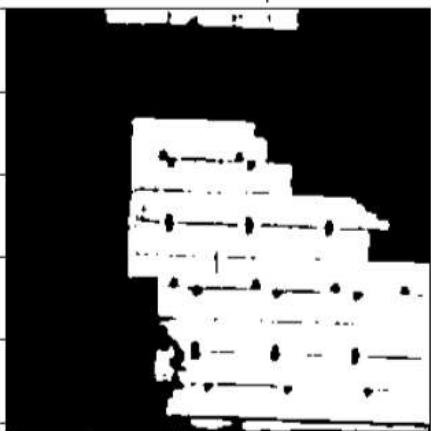


ID: 3 | IoU: 0.923

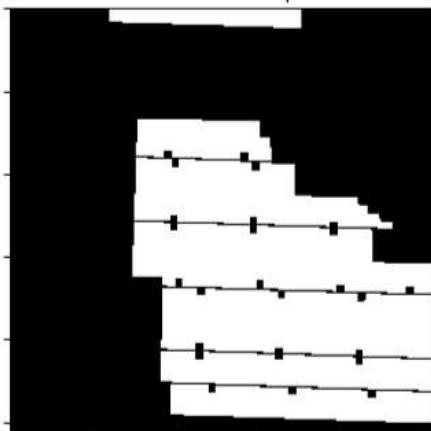
Original Image | 3



Prediction | 3



Ground Truth | 3

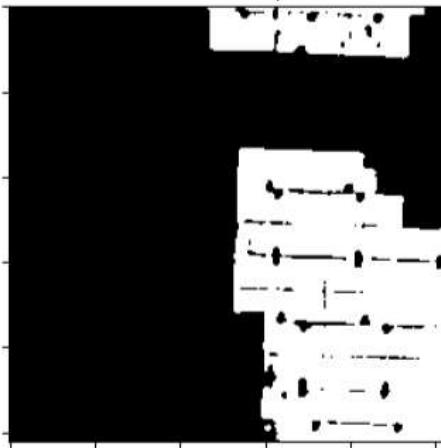


ID: 7 | IoU: 0.937

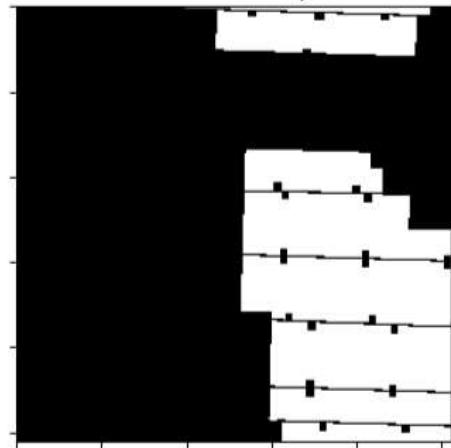
Original Image | 7



Prediction | 7



Ground Truth | 7



ID: 8 | IoU: 0.808

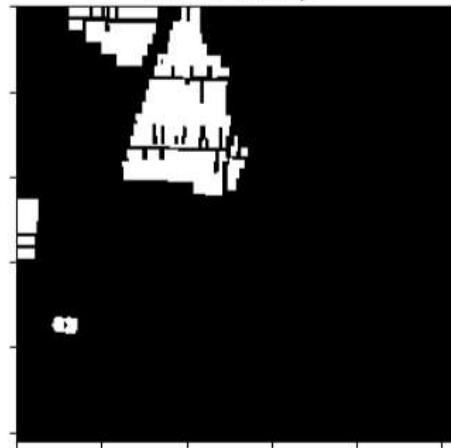
Original Image | 8



Prediction | 8



Ground Truth | 8



ID: 9 | IoU: 0.812

Original Image | 9



Prediction | 9



Ground Truth | 9



ID: 10 | IoU: 0.865

## Study the Class Balance:

Training Set: {'positive\_ratio': 0.026, 'negative\_ratio': 0.974, 'total\_pixels': 13172736}

Validation Set: {'positive\_ratio': 0.123, 'negative\_ratio': 0.877, 'total\_pixels': 1769472}

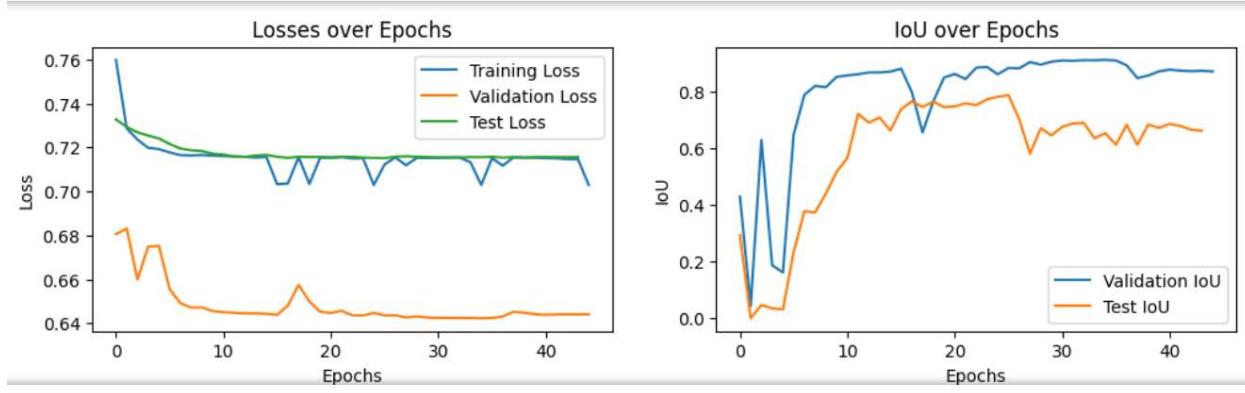
Test Set: {'positive\_ratio': 0.035, 'negative\_ratio': 0.965, 'total\_pixels': 2883584}"

There's an extreme class imbalance (especially the low positive ratio in the training set) is a critical factor.

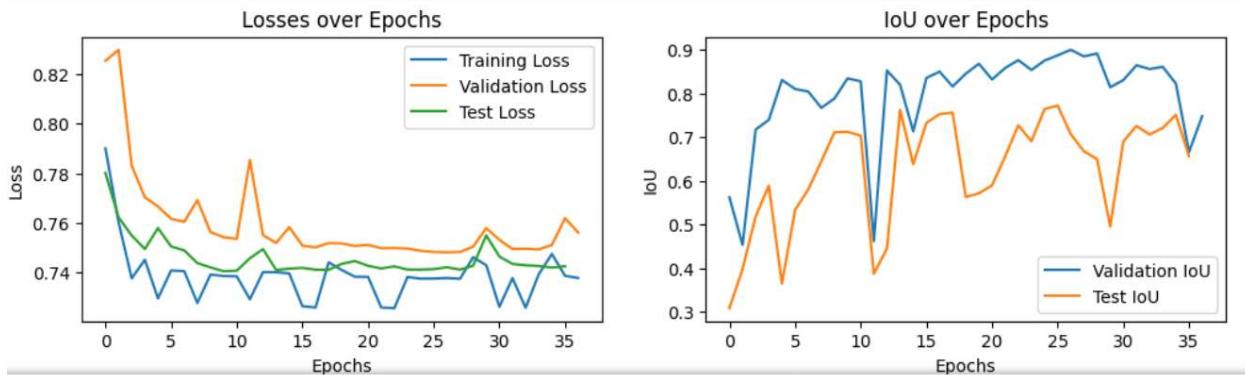
Loss function adjustment:

```
self.bce = nn.BCEWithLogitsLoss(pos_weight=torch.tensor(10.0)) # Adjust `pos_weight` based on the imbalance ratio
```

Before:



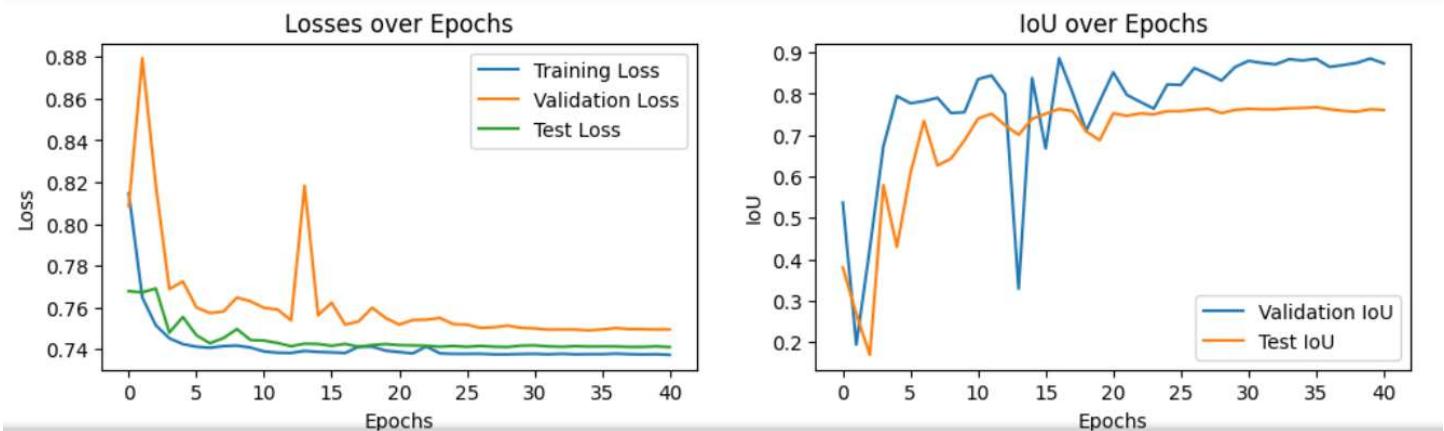
After:



Thresholds:      25%      |    50%      |    75%

Accuracy:	99.01		99.03		99.01
IoU:	80.89		80.92		80.48
F-score:	89.44		89.46		89.18

**Exclude blank mask images from the accuracy scoring during the training and the loss function.**



Thresholds:      25% | 50% | 75%

Accuracy:            99.08 | 99.08 | 99.08

IoU:                82.26 | 82.24 | 81.99

F-score:            90.27 | 90.25 | 90.11

## Loss function adjustment:

Change :

```
self.bce = nn.BCEWithLogitsLoss(pos_weight=torch.tensor(10.0))
```

In fact, there aren't raw scores that justify using the BCEWithLogitsLoss.

We change to:

```
self.bce = nn.BCELoss()
```

and apply the weighted version of the loss function manually:

```
class WeightedBCELoss(nn.Module):
```

```
    def __init__(self, pos_weight=1.0):
```

```
        super(WeightedBCELoss, self).__init__()
```

```
        self.pos_weight = pos_weight # Weight for the positive class
```

```
    def forward(self, inputs, targets):
```

```
        # Weighted BCE computation
```

```
        loss = -self.pos_weight * targets * torch.log(inputs + 1e-7) - (1 - targets) * torch.log(1 - inputs + 1e-7)
```

```
        return loss.mean()
```

```
class BCEFocalJaccardLoss(nn.Module):
```

```
    def __init__(self, alpha=0.8, gamma=2, pos_weight=10.0):
```

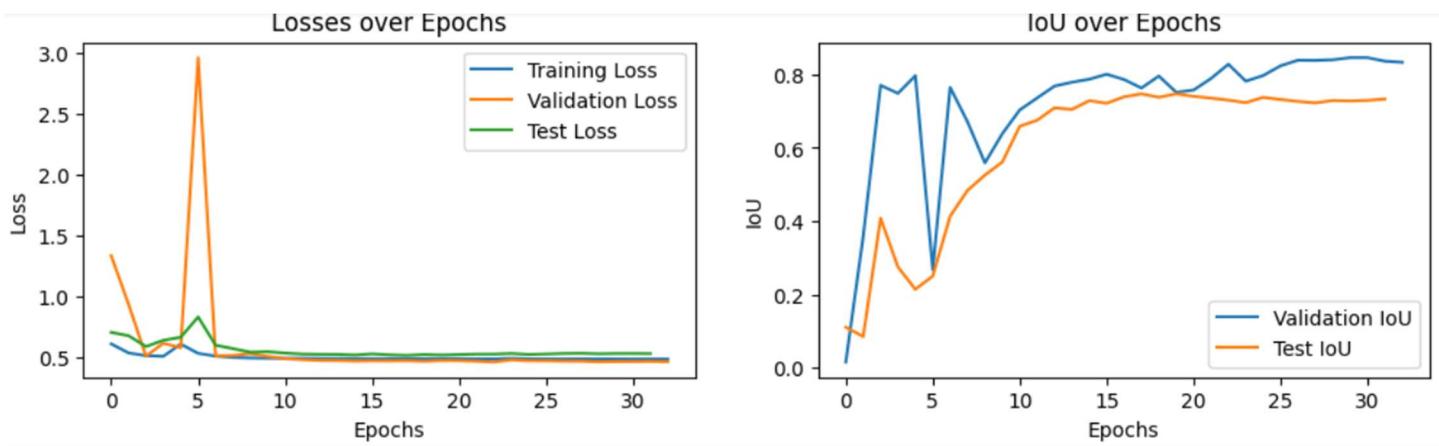
```
        super(BCEFocalJaccardLoss, self).__init__()
```

```
        self.bce = WeightedBCELoss(pos_weight=pos_weight) # Use the custom weighted BCE loss
```

```
        self.jaccard = smp.losses.JaccardLoss(mode='binary')
```

```
        self.alpha = alpha
```

```
        self.gamma = gamma
```

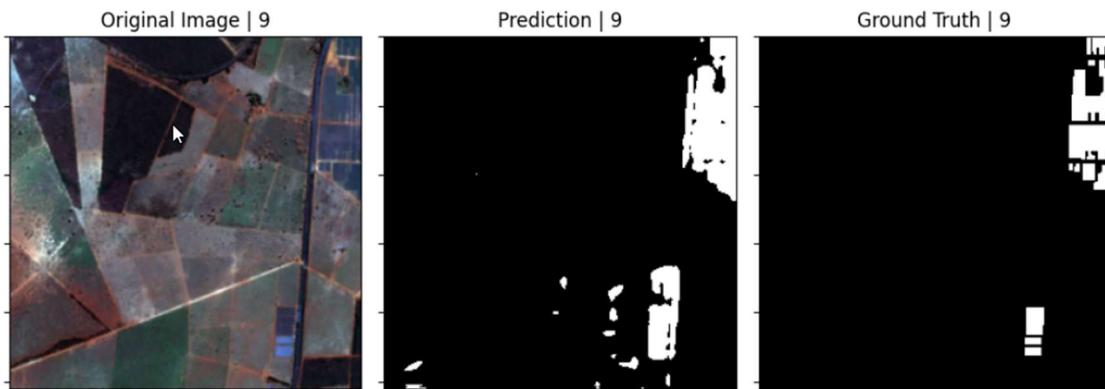


Thresholds:      25%      |    50%      |    75%

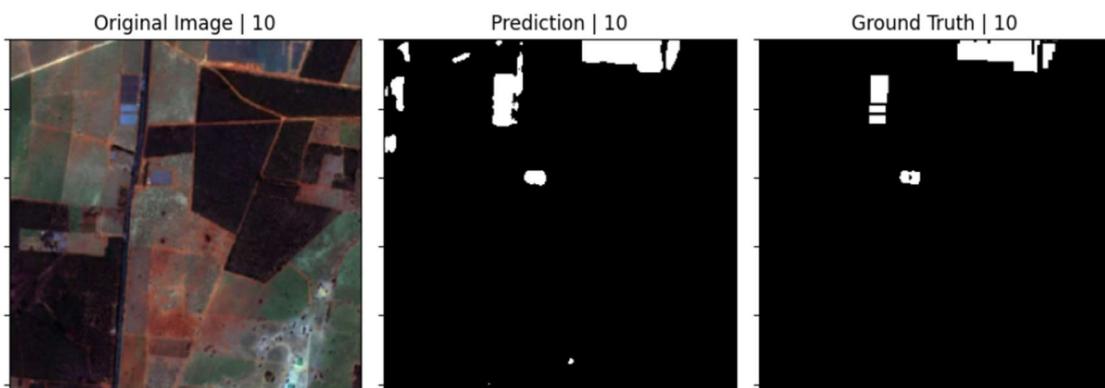
---

	25%	50%	75%
Accuracy:	99.07	99.03	98.81
IoU:	82.14	80.68	75.87
F-score:	90.19	89.31	86.28

ID: 9 | IoU: 0.524

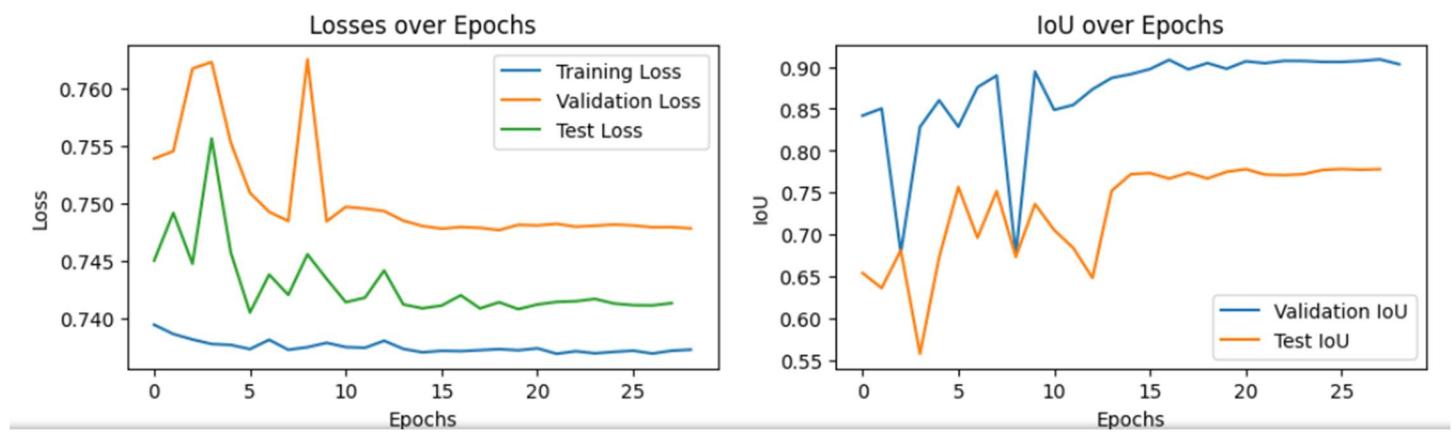


ID: 10 | IoU: 0.63



## Retourning to the previous version with Logits

```
self.bce = nn.BCEWithLogitsLoss(pos_weight=torch.tensor(10.0))
```



Thresholds: 25% | 50% | 75%

---

	25%	50%	75%
Accuracy:	99.09	99.08	99.06
IoU:	82.39	82.01	81.51
F-score:	90.34	90.12	89.82

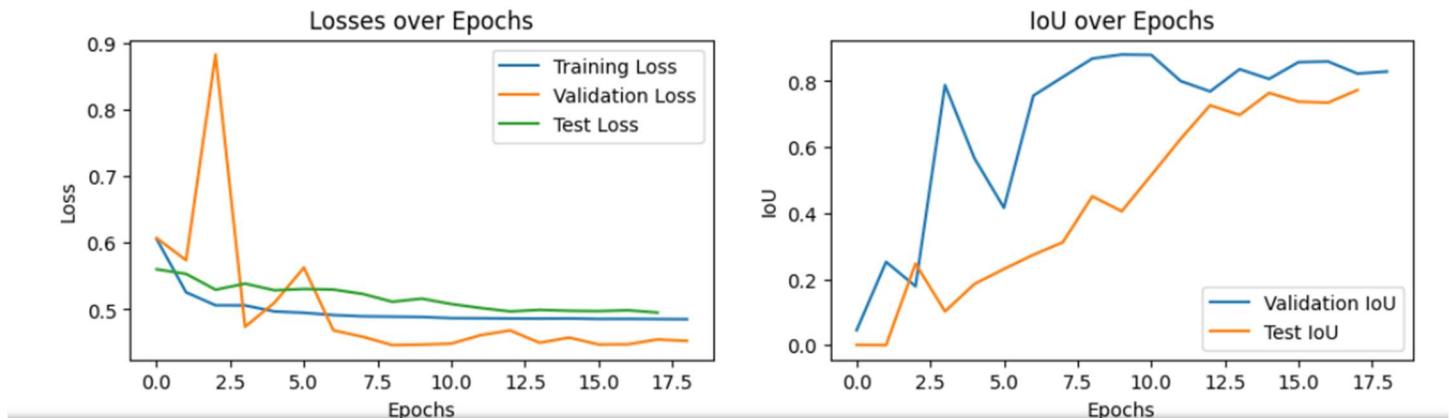
Try a lower weight

pos\_weight=2.0 instead of 10.0

```
def __init__(self, alpha=0.8, gamma=1.5, pos_weight=2.0):
```

keep:

```
return 0.3 * bce_loss + 0.5 * jaccard_loss + 0.2 * focal_loss
```

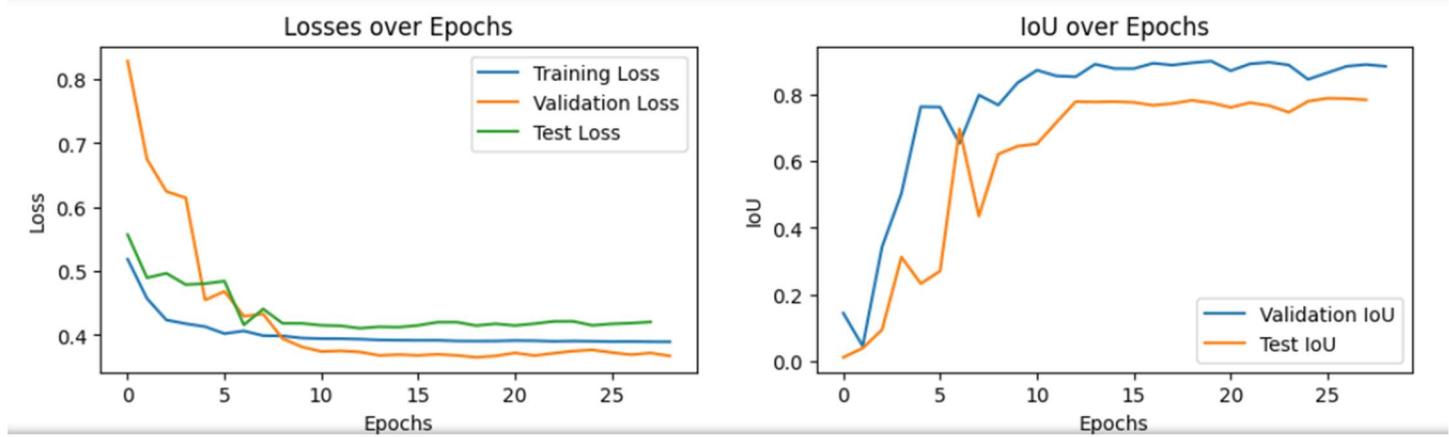


pos\_weight=4.0

```
def __init__(self, alpha=0.8, gamma=1.5, pos_weight=4.0):
```

change weights:

```
return 0.4 * bce_loss + 0.4 * jaccard_loss + 0.2 * focal_loss
```



Thresholds:	25%		50%		75%
<hr/>					
Accuracy:	99.13		99.15		99.08
IoU:	83.45		83.21		81.53
F-score:	90.98		90.84		89.82

```
pos_weight=4.0
```

```
def __init__(self, alpha=0.8, gamma=1.0, pos_weight=3.0):
```

change weights:

```
return 0.4 * bce_loss + 0.4 * jaccard_loss + 0.2 * focal_loss
```

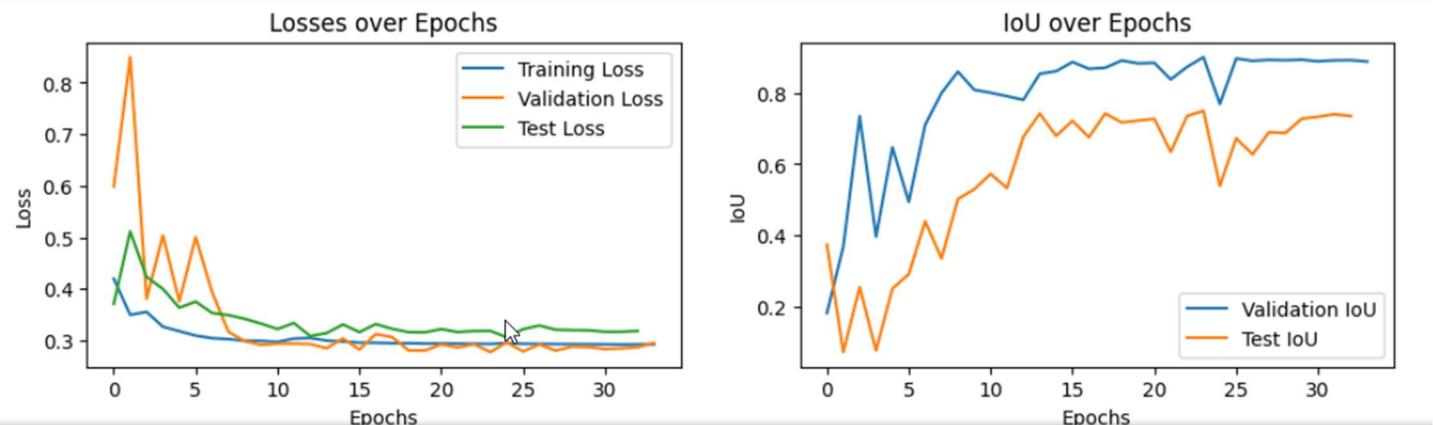
**Improvement:** The Loss Function considers not only the positive-class IoU, but also the negative-class IoU.

Adding two new weighted terms.

pos\_weight=4.0, neg\_weight=0.5

```
loss = -self.pos_weight * targets * torch.log(inputs + 1e-7) - \
      self.neg_weight * (1 - targets) * torch.log(1 - inputs + 1e-7)
```

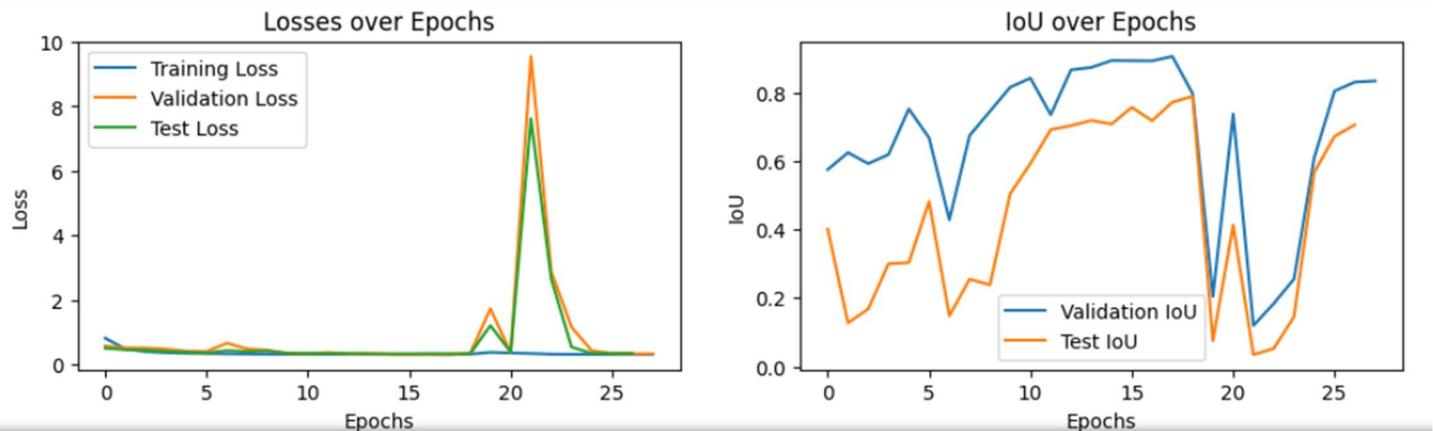
total\_loss = 0.4 \* bce\_loss + 0.3 \* focal\_loss + 0.3 \* jaccard\_loss



Thresholds: 25% | 50% | 75%

Accuracy:	99.09	99.12	99.01
IoU:	82.75	82.63	79.95
F-score:	90.56	90.49	88.86

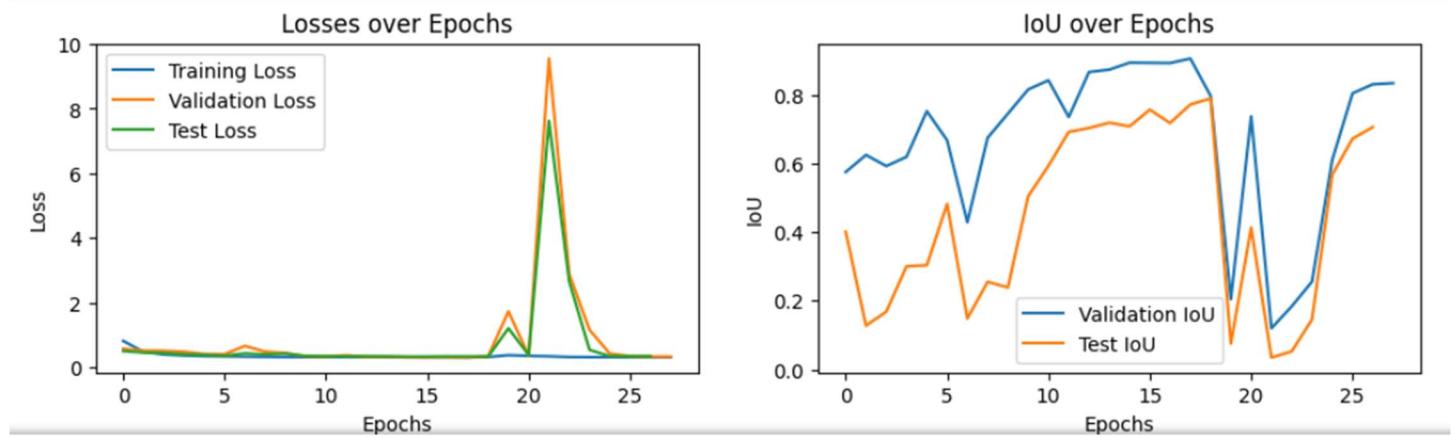
pos\_weight=4.0, neg\_weight=2.0



Thresholds: 25% | 50% | 75%

Accuracy:	98.99	98.97	98.79
IoU:	80.58	79.43	75.34
F-score:	89.25	88.54	85.94

pos\_weight=4.0, neg\_weight=2.0



Thresholds:      25%      |    50%      |    75%

-----  
Accuracy:            98.99 | 98.97 | 98.79  
IoU:                80.58 | 79.43 | 75.34  
F-score:            89.25 | 88.54 | 85.94

```

pos_weight=1.0, neg_weight=1.0

iou_positive, mean_iou = compute_class_aware_iou(preds, targets)

# Jaccard Loss for Positive and Negative IoU

jaccard_loss_positive = 1.0 - iou_positive # Positive IoU

jaccard_loss_negative = 1.0 - (2 * mean_iou - iou_positive) # Derive Negative IoU

# Weighted Jaccard Loss

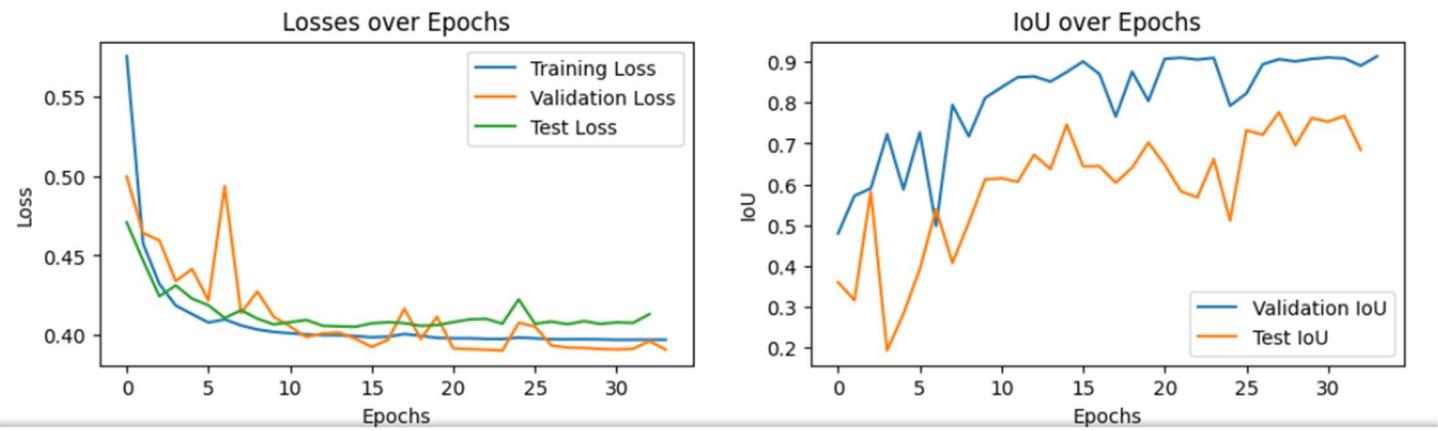
jaccard_loss = 0.5 * jaccard_loss_positive + 0.5 * jaccard_loss_negative

# Combine all losses

total_loss = 0.4 * bce_loss + 0.2 * focal_loss + 0.4 * jaccard_loss

return total_loss

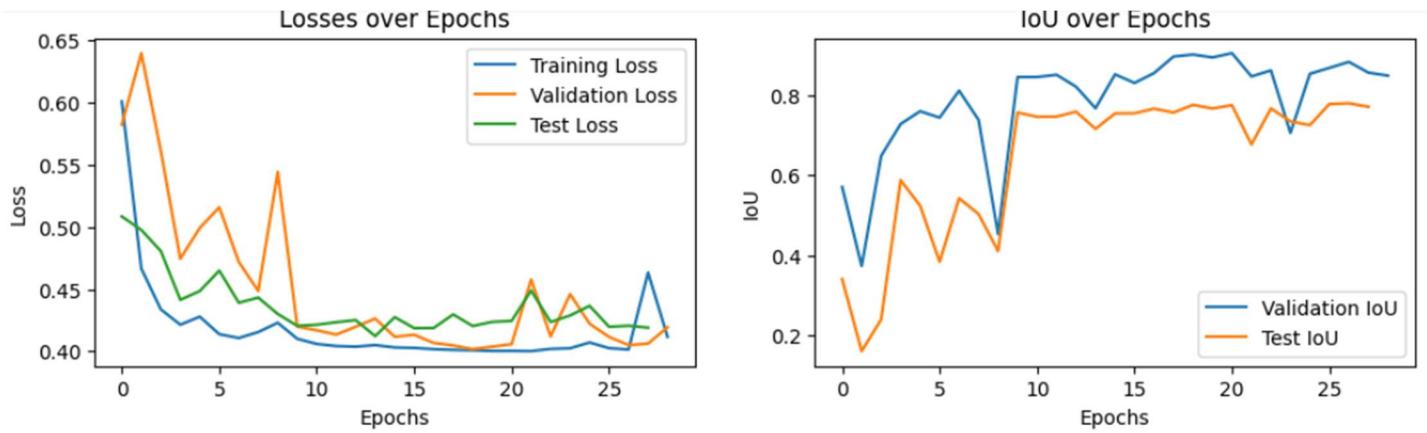
```



Thresholds:	25%		50%		75%
<hr/>					
Accuracy:	99.1		98.94		98.66
IoU:	82.09		78.39		72.53
F-score:	90.16		87.89		84.08

```
pos_weight=4.0, neg_weight=1.0
```

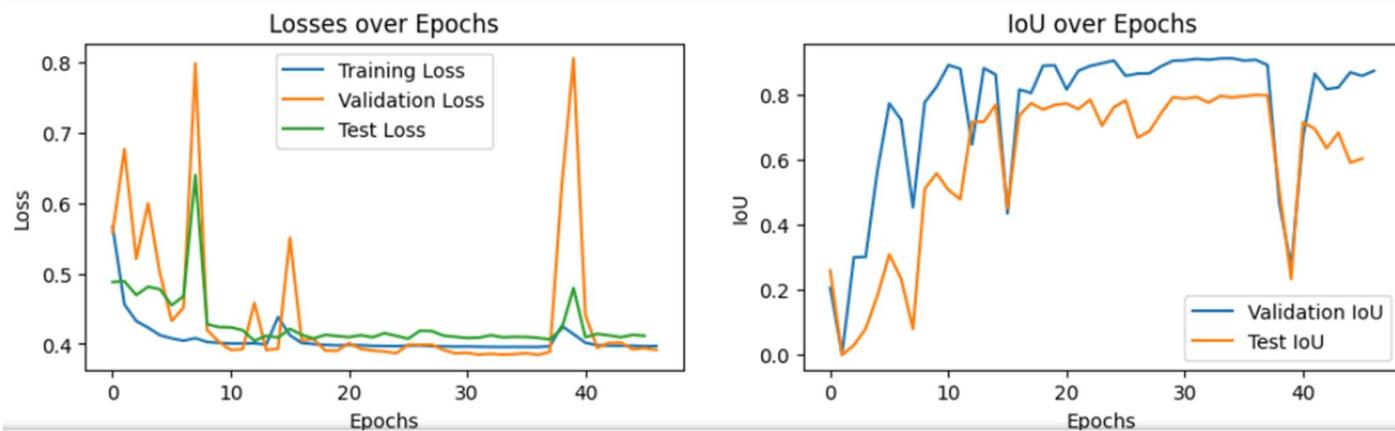
```
iou_positive, mean_iou = compute_class_aware_iou(preds, targets)  
# Jaccard Loss for Positive and Negative IoU  
jaccard_loss_positive = 1.0 - iou_positive # Positive IoU  
jaccard_loss_negative = 1.0 - (2 * mean_iou - iou_positive) # Derive Negative IoU  
# Weighted Jaccard Loss  
jaccard_loss = 0.5 * jaccard_loss_positive + 0.5 * jaccard_loss_negative  
# Combine all losses  
total_loss = 0.4 * bce_loss + 0.2 * focal_loss + 0.4 * jaccard_loss  
return total_loss
```



Thresholds:	25%		50%		75%
<hr/>					
Accuracy:	99.07		99.13		99.03
IoU:	82.49		82.94		80.54
F-score:	90.41		90.67		89.22

pos\_weight=2.0, neg\_weight=1.0

jaccard\_loss = 0.7 \* jaccard\_loss\_positive + 0.3 \* jaccard\_loss\_negative



Thresholds:      25% | 50% | 75%

---

Accuracy:	99.19		99.24		99.12
IoU:	84.53		84.96		82.19
F-score:	91.62		91.87		90.22

TP:87901

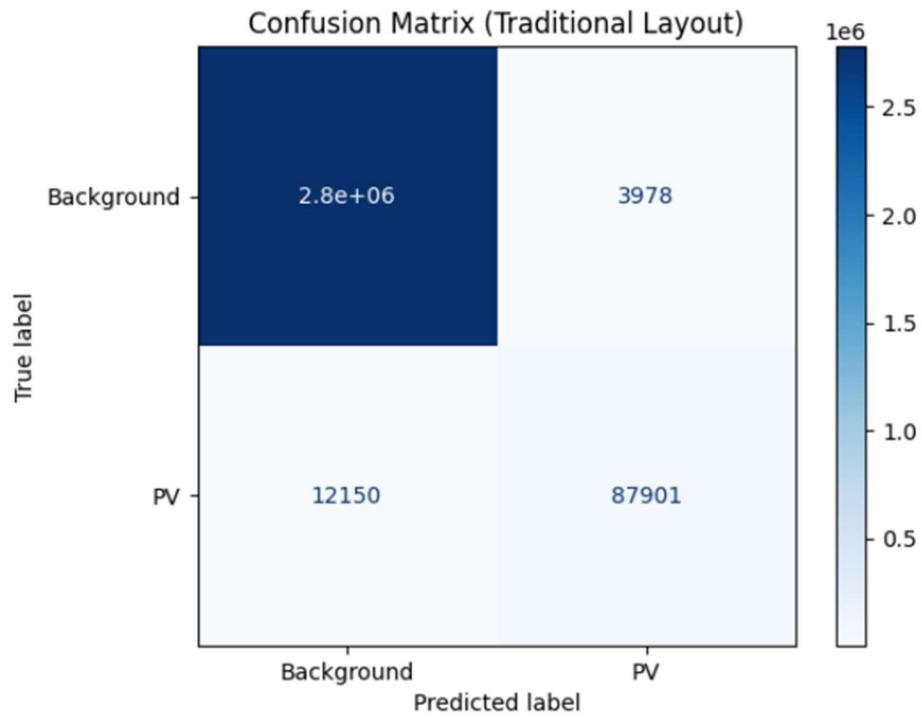
TN:2779555

FP:3978

FN:12150

---

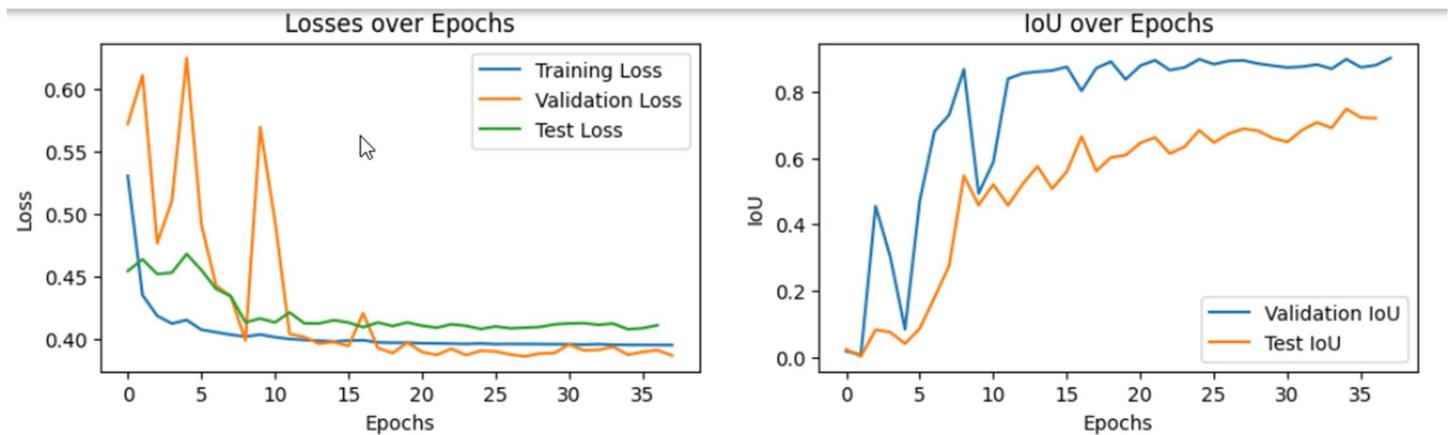
Accuracy: 0.9944  $(TP + TN) / (TP + TN + FP + FN)$   
Recall: 0.8786  $TP / (TP + FN)$   
Specificity: 0.9986  $TN / (TN + FP)$   
Precision: 0.9567  $TP / (TP + FP)$   
F1-Score: 0.9160  $(2 * precision * recall) / (precision + recall)$



See the consequences of switching weights

pos\_weight=1.0, neg\_weight=2.0

jaccard\_loss = 0.7 \* jaccard\_loss\_positive + 0.3 \* jaccard\_loss\_negative



Thresholds: 25% | 50% | 75%

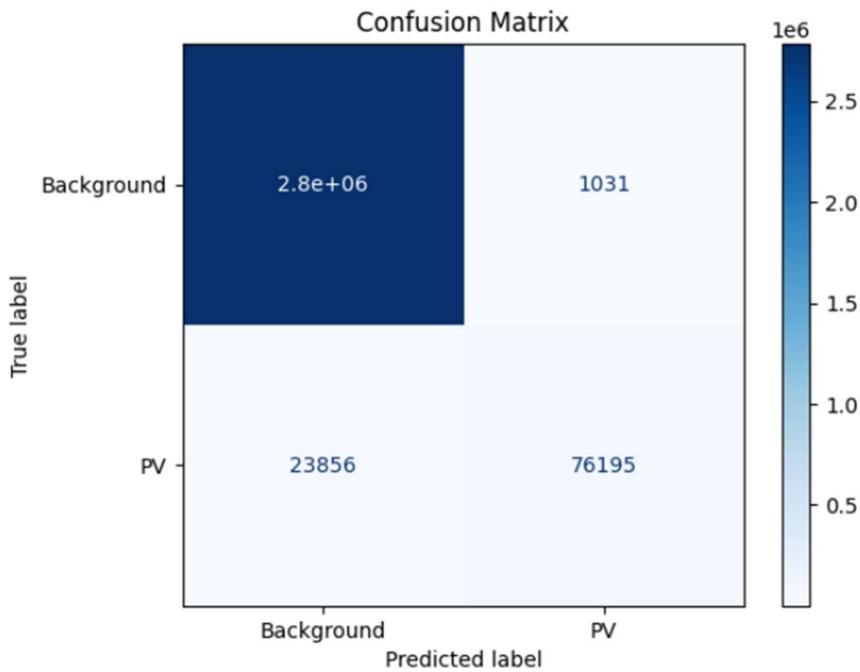
Accuracy: 99.01 | 98.83 | 98.58

IoU: 79.97 | 76.06 | 70.78

F-score: 88.87 | 86.4 | 82.89

TP:76195 TN:2782502 FP:1031 FN:23856

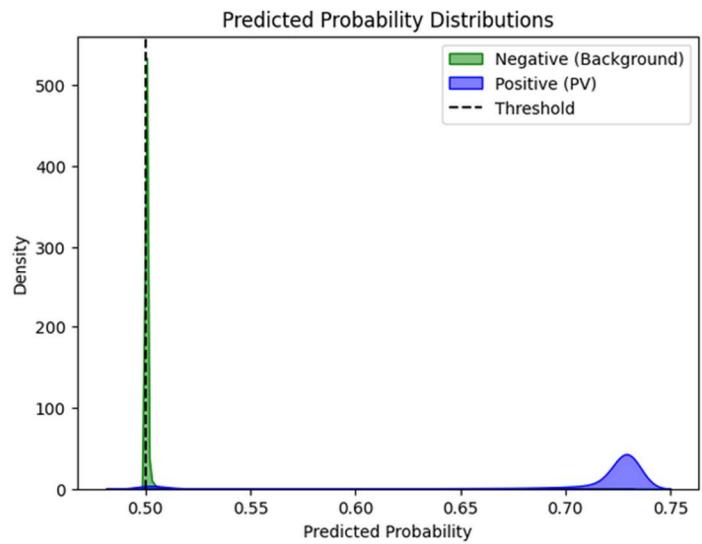
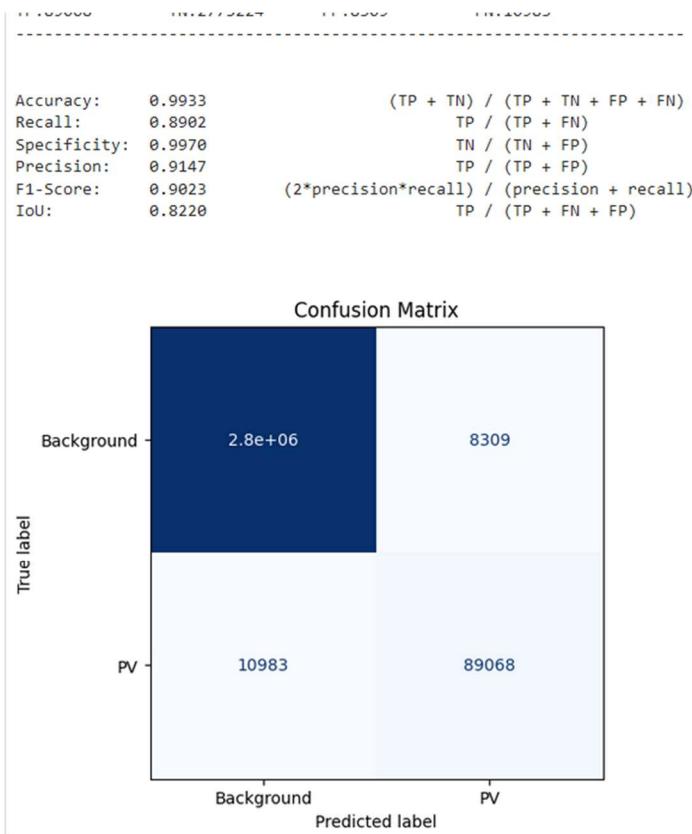
Accuracy: 0.9914      I       $(TP + TN) / (TP + TN + FP + FN)$   
Recall: 0.7616      TP / (TP + FN)  
Specificity: 0.9996      TN / (TN + FP)  
Precision: 0.9866      TP / (TP + FP)  
F1-Score: 0.8596       $(2 * \text{precision} * \text{recall}) / (\text{precision} + \text{recall})$



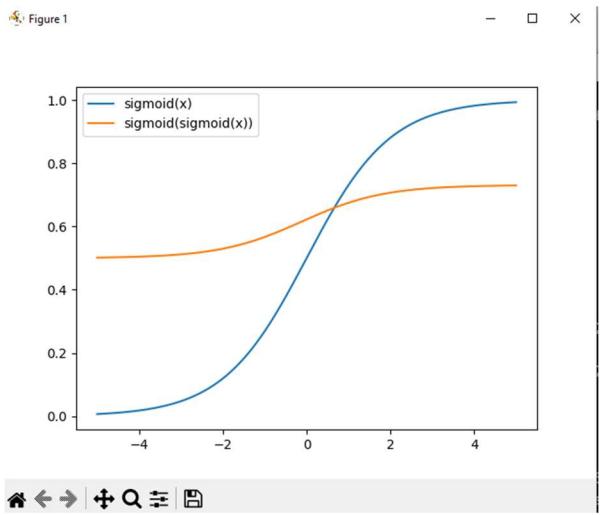
Switching weights increases the number of FN !, then it impacts Recall and IoU.

pos\_weight=4.0, neg\_weight=1.0

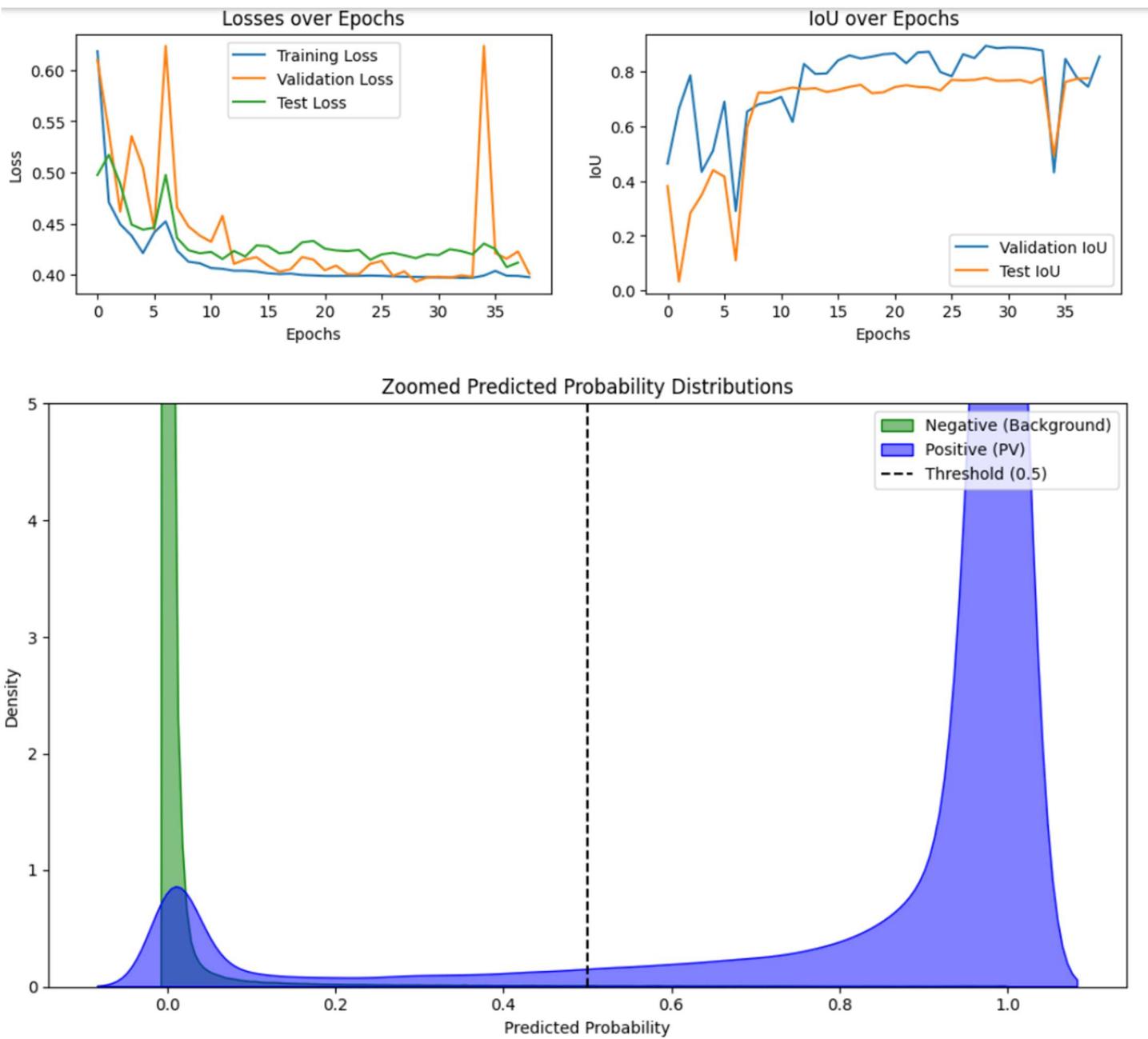
jaccard\_loss =  $0.7 * \text{jaccard\_loss\_positive} + 0.3 * \text{jaccard\_loss\_negative}$



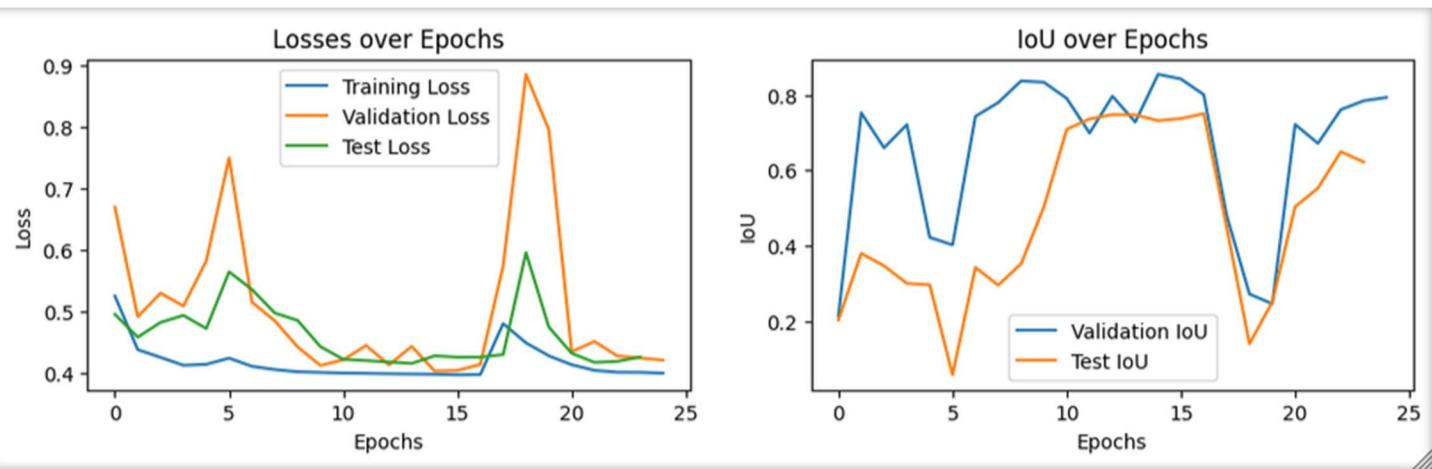
Output distribution from 0.5 to 0.75?...



Bug detected. Sigmoid applied twice!



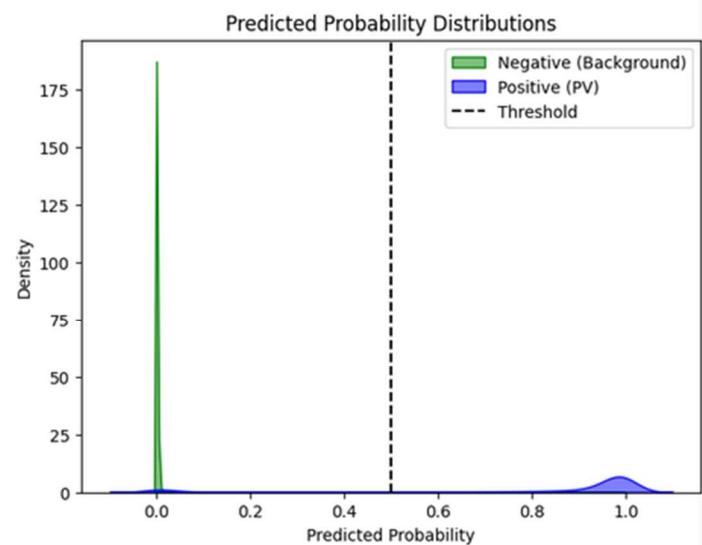
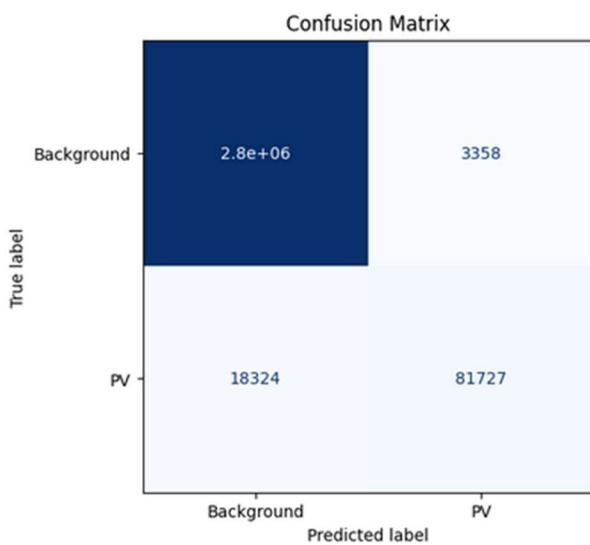
Sigmoid applied once. Corrected.

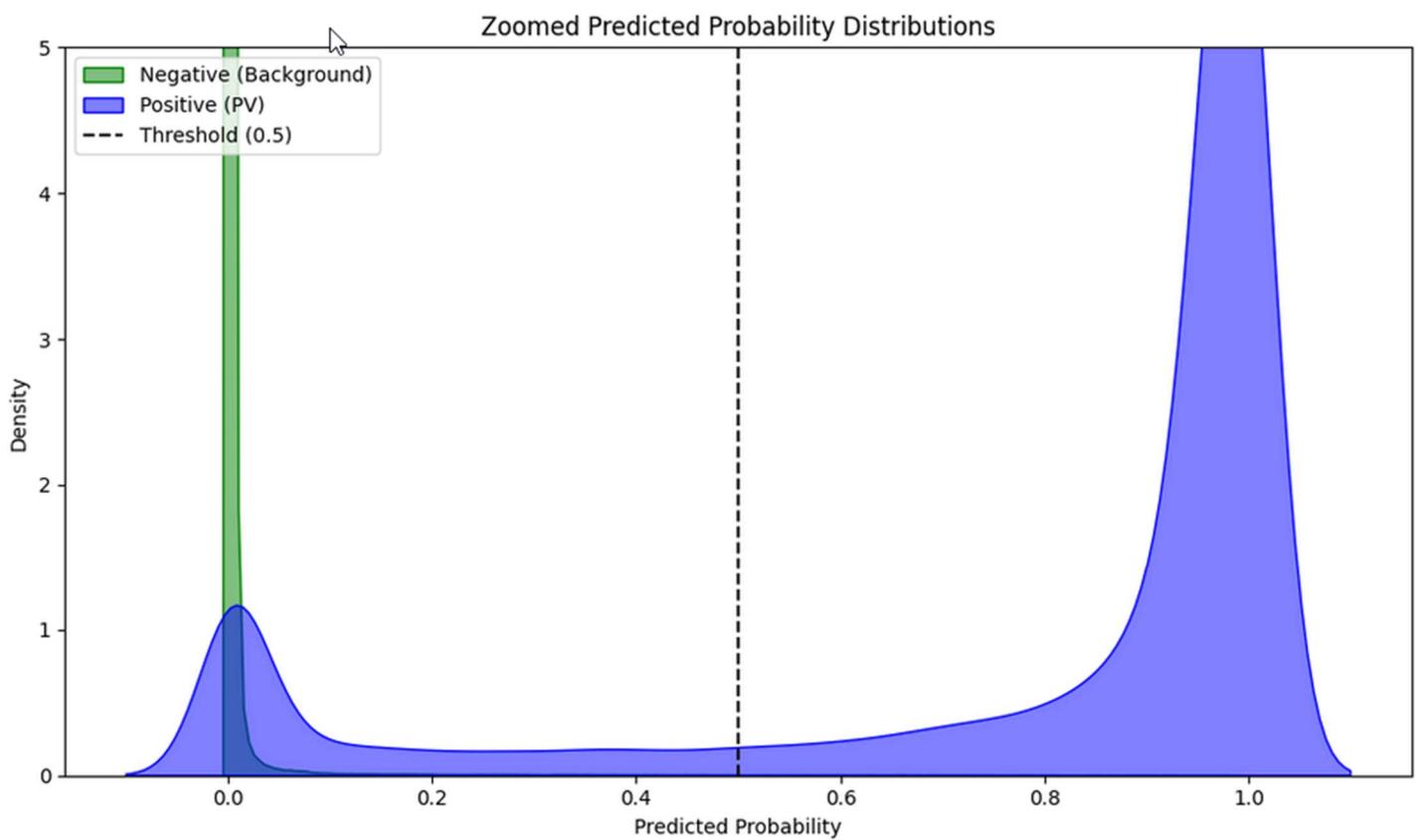


TP:81727      TN:2780175      FP:3358      FN:18324



Accuracy:  $0.9925 = (\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN})$   
 Recall:  $0.8169 = \text{TP} / (\text{TP} + \text{FN})$   
 Specificity:  $0.9988 = \text{TN} / (\text{TN} + \text{FP})$   
 Precision:  $0.9605 = \text{TP} / (\text{TP} + \text{FP})$   
 F1-Score:  $0.8829 = (2 * \text{precision} * \text{recall}) / (\text{precision} + \text{recall})$   
 IoU:  $0.7903 = \text{TP} / (\text{TP} + \text{FN} + \text{FP})$





Notice the bigger amount of FN compared with the previous chart.

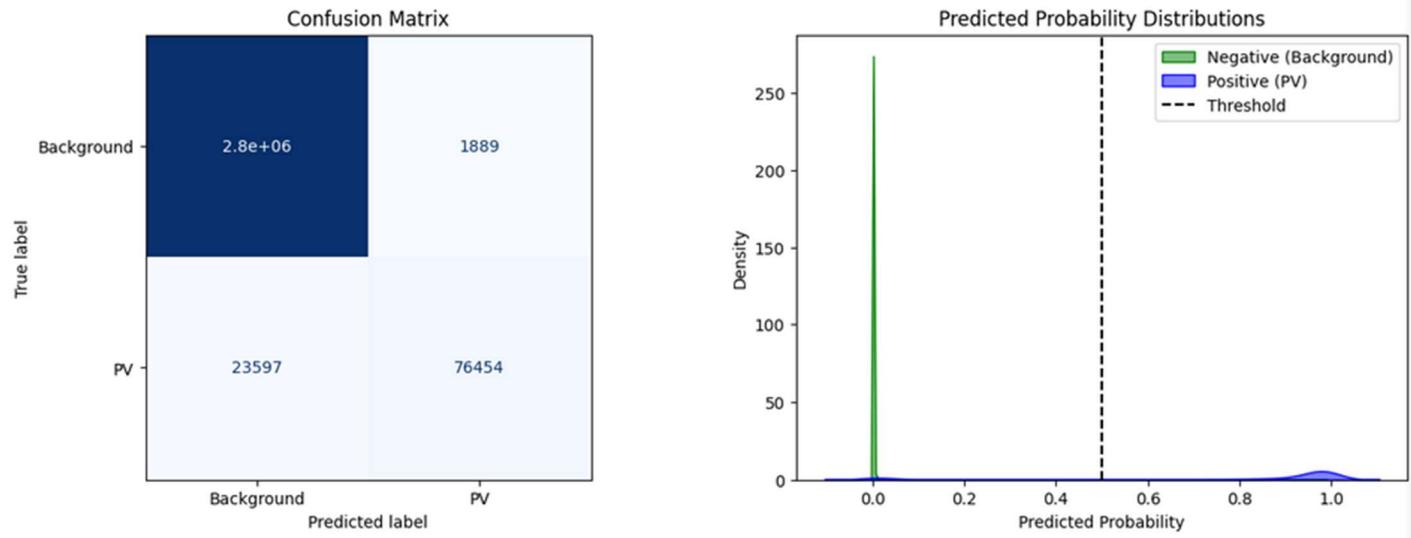
## Col2c

Now fixing the loss function:

```
#preds = model(inputs) # No need to apply sigmoid again if already applied in the model
#preds = inputs.sigmoid() # Apply sigmoid activation
preds = inputs
```

```
Final shapes - all_preds: (2883584,), all_masks: (2883584,)
TP:76454           TN:2781644       FP:1889        FN:23597
-----
```

Accuracy:	$0.9912 = \frac{(TP + TN)}{(TP + TN + FP + FN)}$
Recall:	$0.7642 = \frac{TP}{(TP + FN)}$
Specificity:	$0.9993 = \frac{TN}{(TN + FP)}$
Precision:	$0.9759 = \frac{TP}{(TP + FP)}$
F1-Score:	$0.8571 = \frac{2 * precision * recall}{(precision + recall)}$
IoU:	$0.7500 = \frac{TP}{(TP + FN + FP)}$



## Apply Regularization:

### Col5 – Add regularization – file: v11

```
optimizer = torch.optim.AdamW(
```

```
    model.parameters(),
```

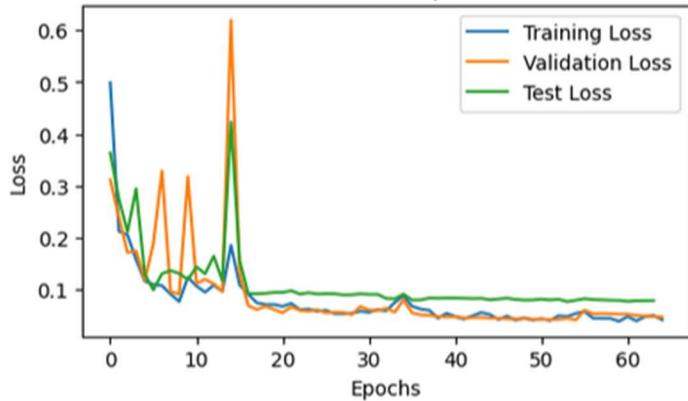
```
    lr=0.0006,
```

```
    #weight_decay=1e-5
```

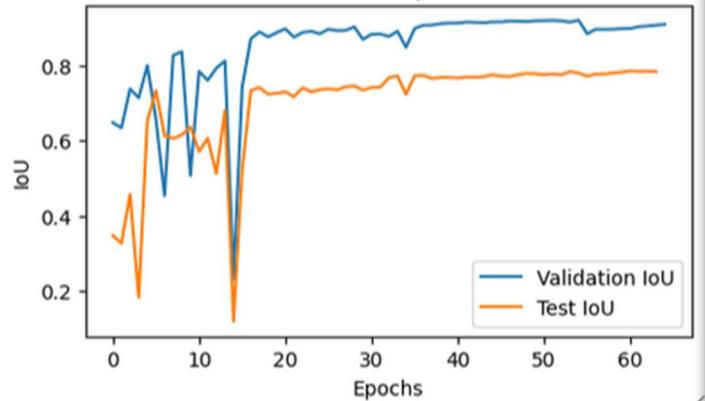
```
    weight_decay=1e-4
```

```
)
```

Losses over Epochs



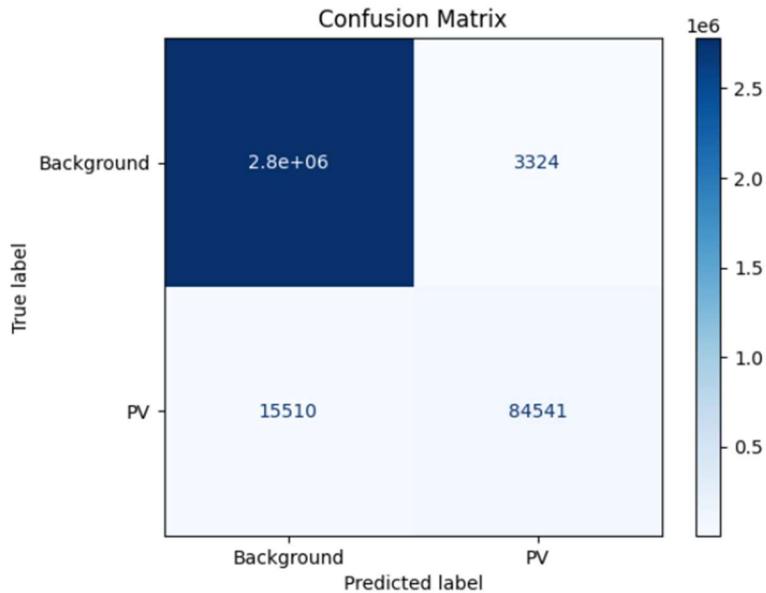
IoU over Epochs



TP:84541 TN:2780209 FP:3324 FN:15510

---

Accuracy:	0.9935	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.8450	$TP / (TP + FN)$
Specificity:	0.9988	$TN / (TN + FP)$
Precision:	0.9622	$TP / (TP + FP)$
F1-Score:	0.8998	$(2 * precision * recall) / (precision + recall)$
IoU:	0.8178	$TP / (TP + FN + FP)$



## BEST PERFORMANCE UNTIL NOW: Add regularization – file: v12

More regularization

```
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0006,  
    #weight_decay=1e-5  
    weight_decay=1e-3 )
```

Extend patience for early stop

```
#early_stopping = EarlyStopping(patience=10, min_delta=0.0001)  
early_stopping = EarlyStopping(patience=15, min_delta=0.00005)
```

EPOCHS=33

TP:83735

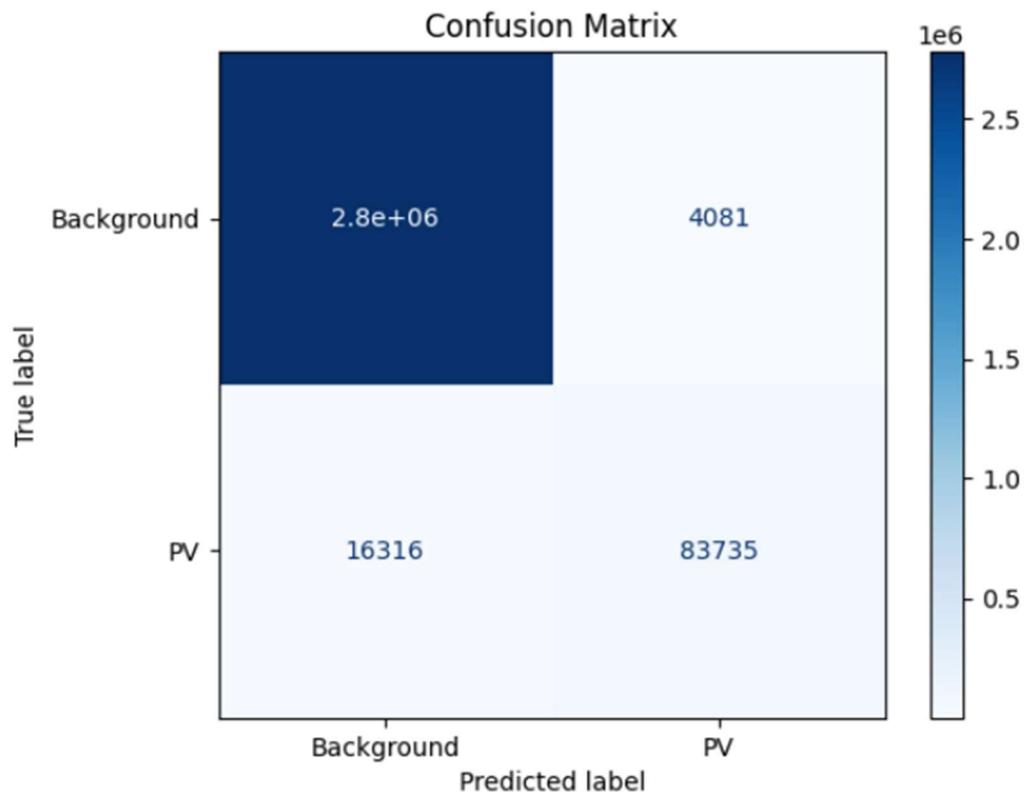
TN:2779452

FP:4081

FN:16316

---

Accuracy:	0.9929	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.8369	$TP / (TP + FN)$
Specificity:	0.9985	$TN / (TN + FP)$
Precision:	0.9535	$TP / (TP + FP)$
F1-Score:	0.8914	$(2 * precision * recall) / (precision + recall)$
IoU:	0.8041	$TP / (TP + FN + FP)$



Col 7

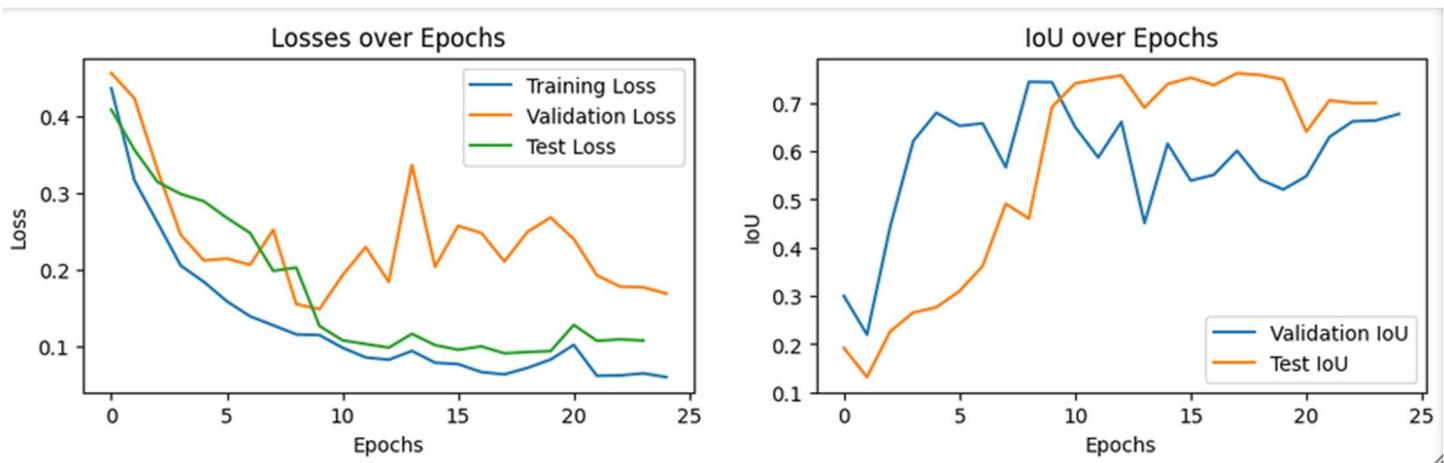
weight\_decay=1e-4

pos\_weight=2.0

early\_stopping = EarlyStopping(patience=15, min\_delta=0.00005)

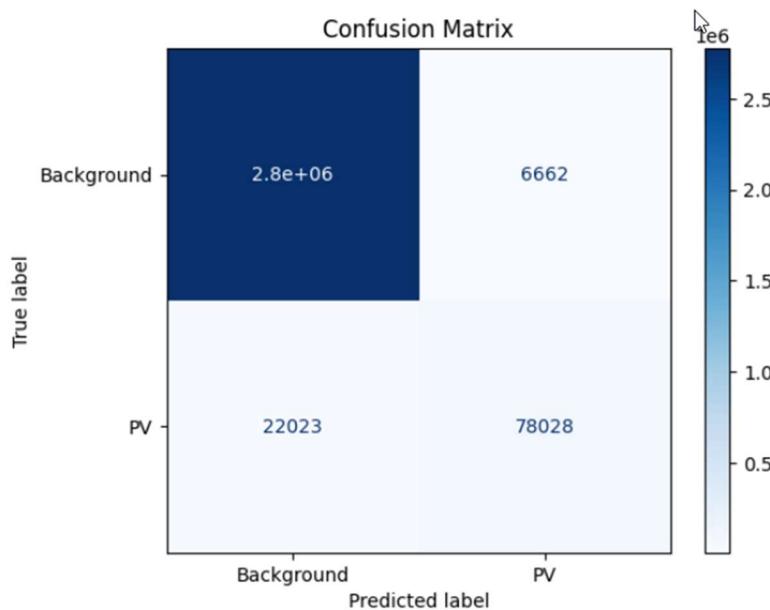
lr=0.0001,

EPOCH = 24



TP:78028 TN:2776871 FP:6662 FN:22023

Accuracy:	0.9901	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.7799	$TP / (TP + FN)$
Specificity:	0.9976	$TN / (TN + FP)$
Precision:	0.9213	$TP / (TP + FP)$
F1-Score:	0.8447	$(2 * precision * recall) / (precision + recall)$
IoU:	0.7312	$TP / (TP + FN + FP)$



Col 8

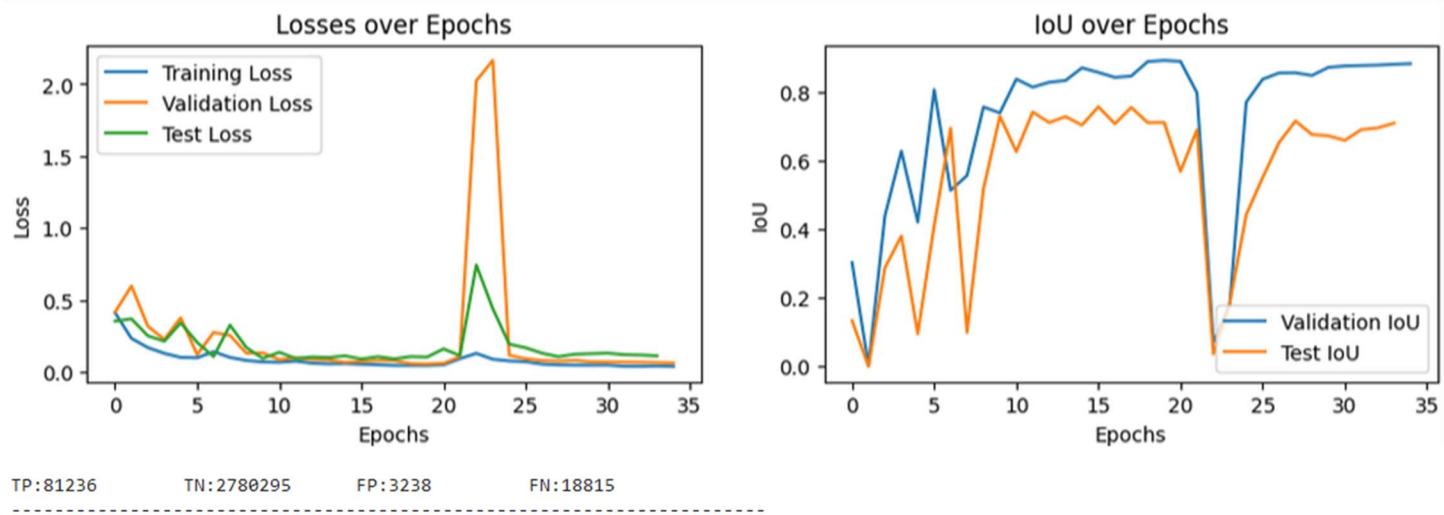
weight\_decay=1e-4

pos\_weight=2.0

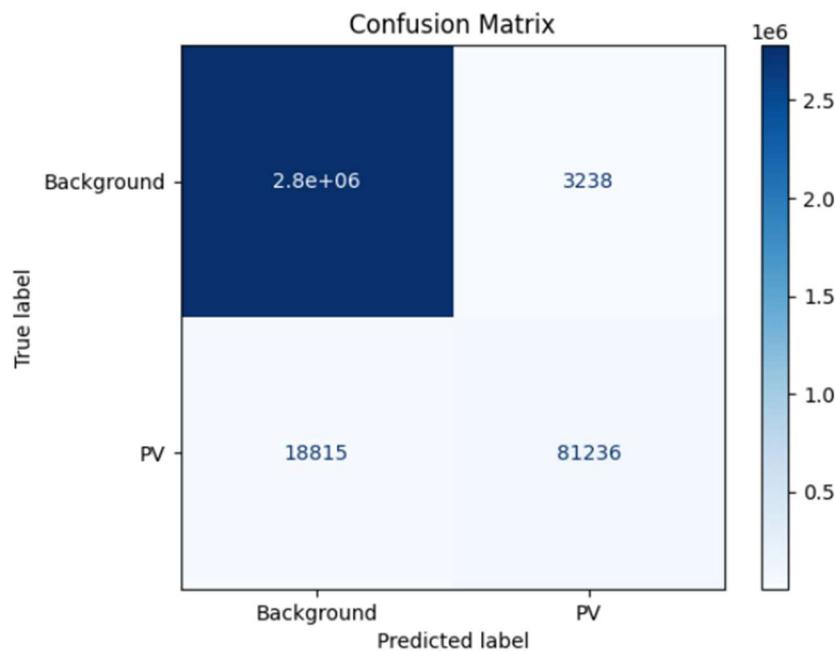
early\_stopping = EarlyStopping(patience=15, min\_delta=0.00005)

lr=0.001,

EPOCH =



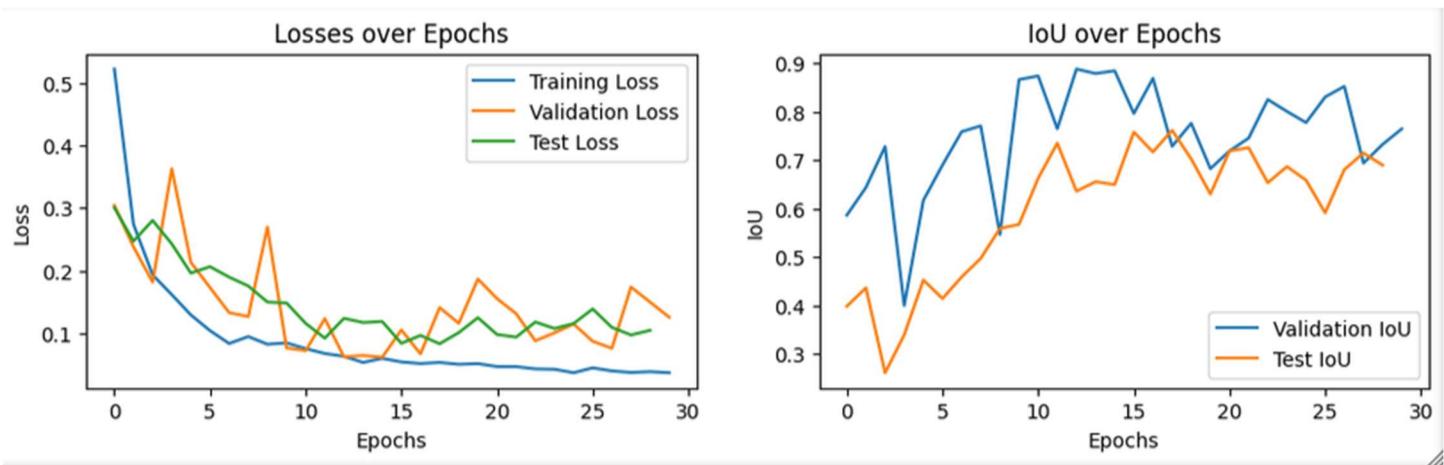
Accuracy:  $0.9924 = \frac{(TP + TN)}{(TP + TN + FP + FN)}$   
Recall:  $0.8119 = \frac{TP}{(TP + FN)}$   
Specificity:  $0.9988 = \frac{TN}{(TN + FP)}$   
Precision:  $0.9617 = \frac{TP}{(TP + FP)}$   
F1-Score:  $0.8805 = \frac{2 * precision * recall}{(precision + recall)}$   
IoU:  $0.7865 = \frac{TP}{(TP + FN + FP)}$



Col 9

weight\_decay=1e-4  
pos\_weight=2.0  
early\_stopping = EarlyStopping(patience=15, min\_delta=0.00005)

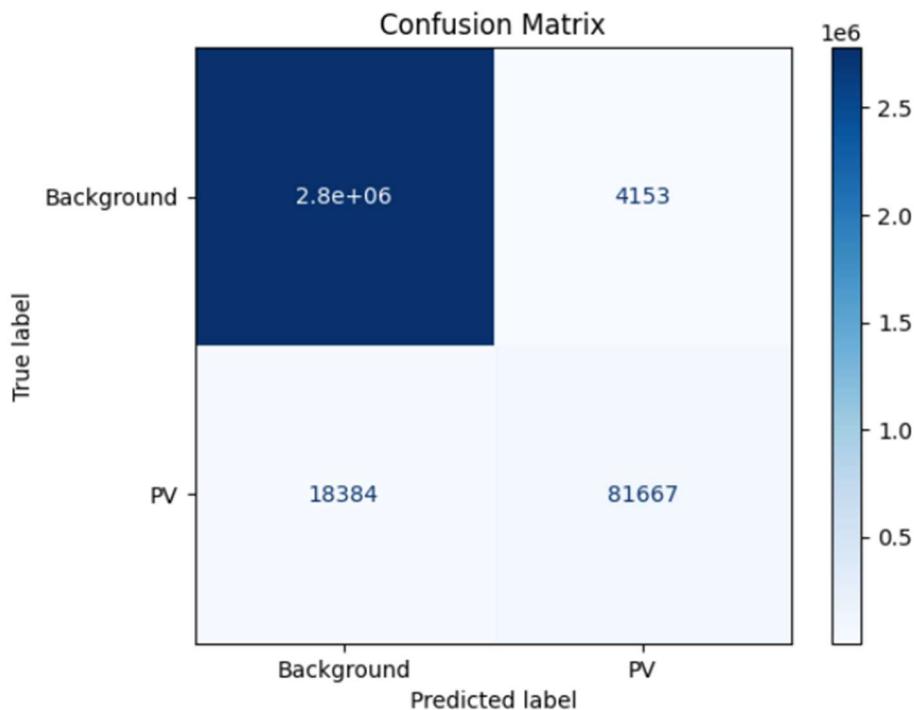
$lr=0.0003$ ,



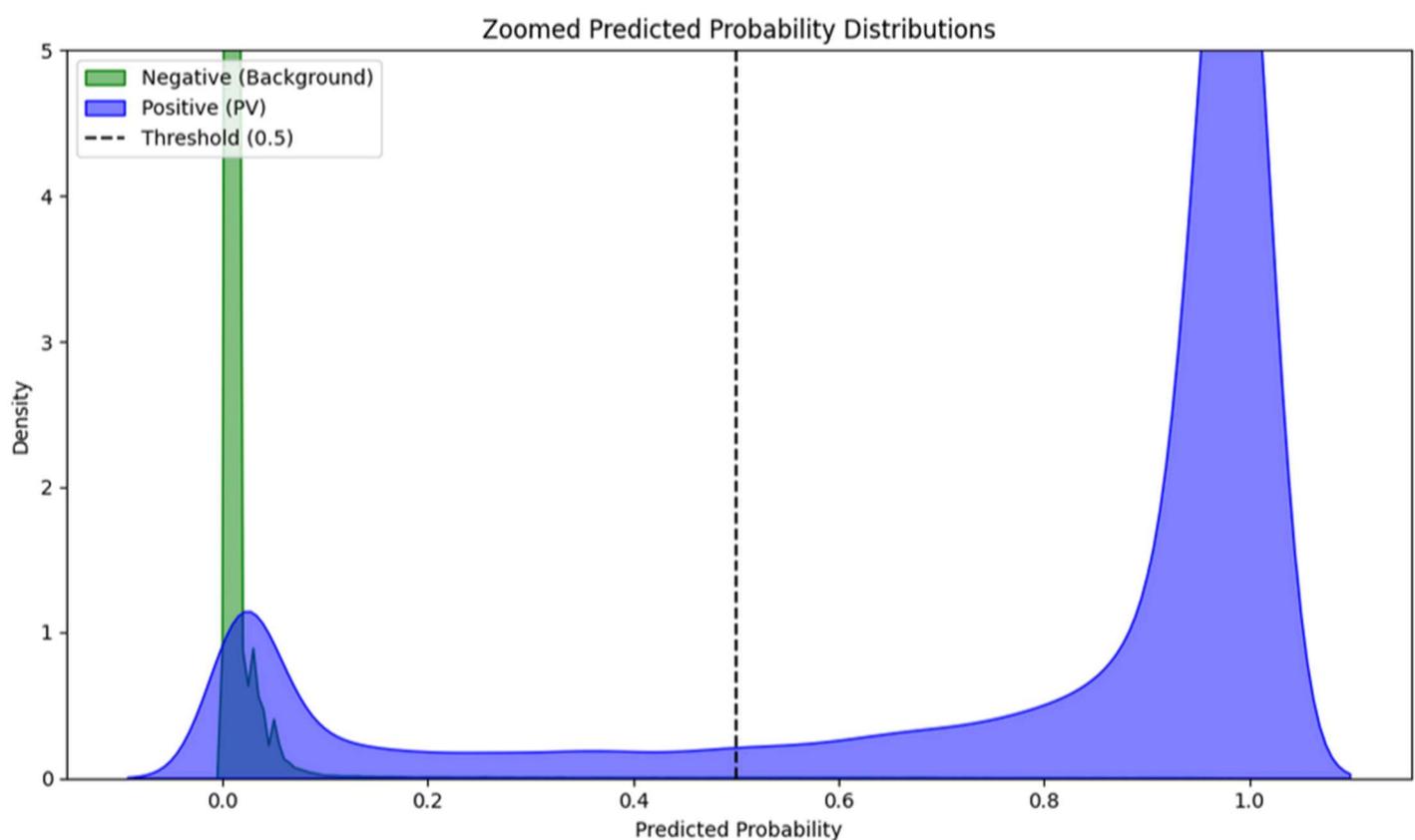
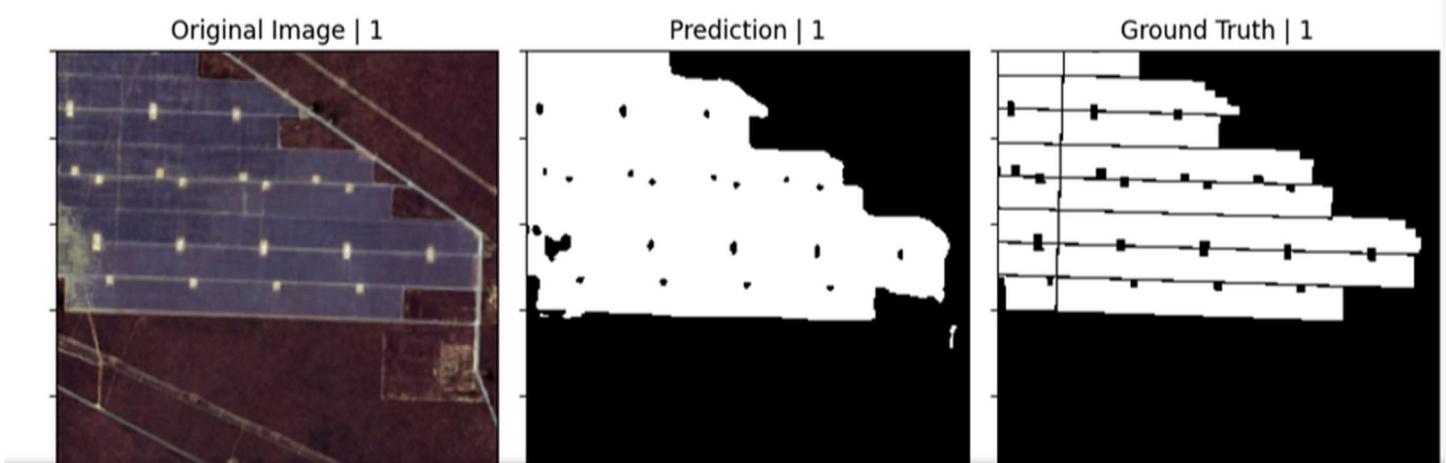
TP:81667      TN:2779380      FP:4153      FN:18384

---

Accuracy:	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	$TP / (TP + FN)$
Specificity:	$TN / (TN + FP)$
Precision:	$TP / (TP + FP)$
F1-Score:	$(2 * precision * recall) / (precision + recall)$
IoU:	$TP / (TP + FN + FP)$



ID: 1 | IoU: 0.903



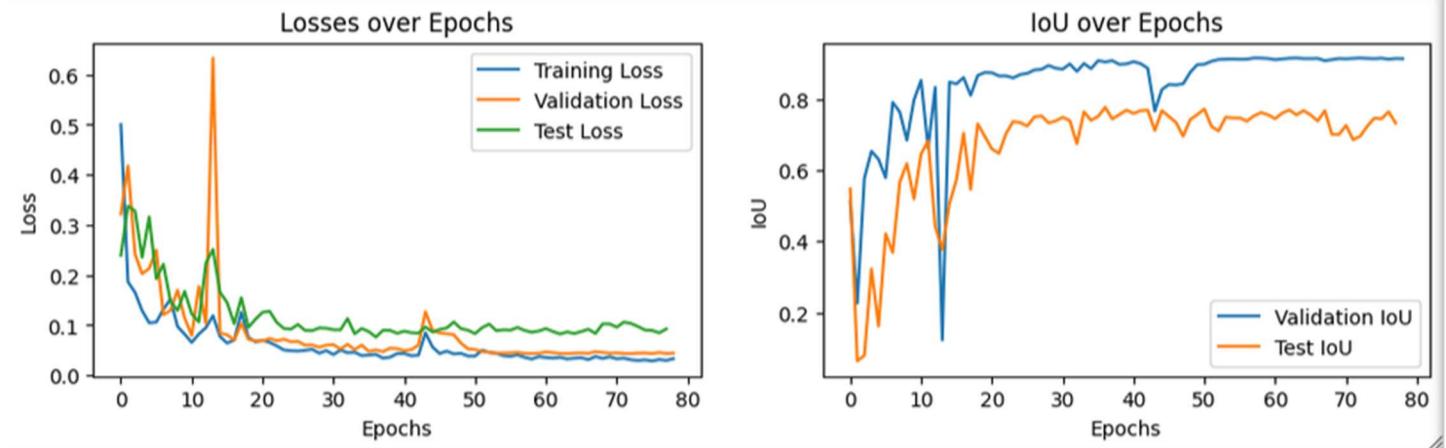
Col 10

weight\_decay=1e-4

pos\_weight=2.0

early\_stopping = EarlyStopping(patience=15, min\_delta=0.00005)

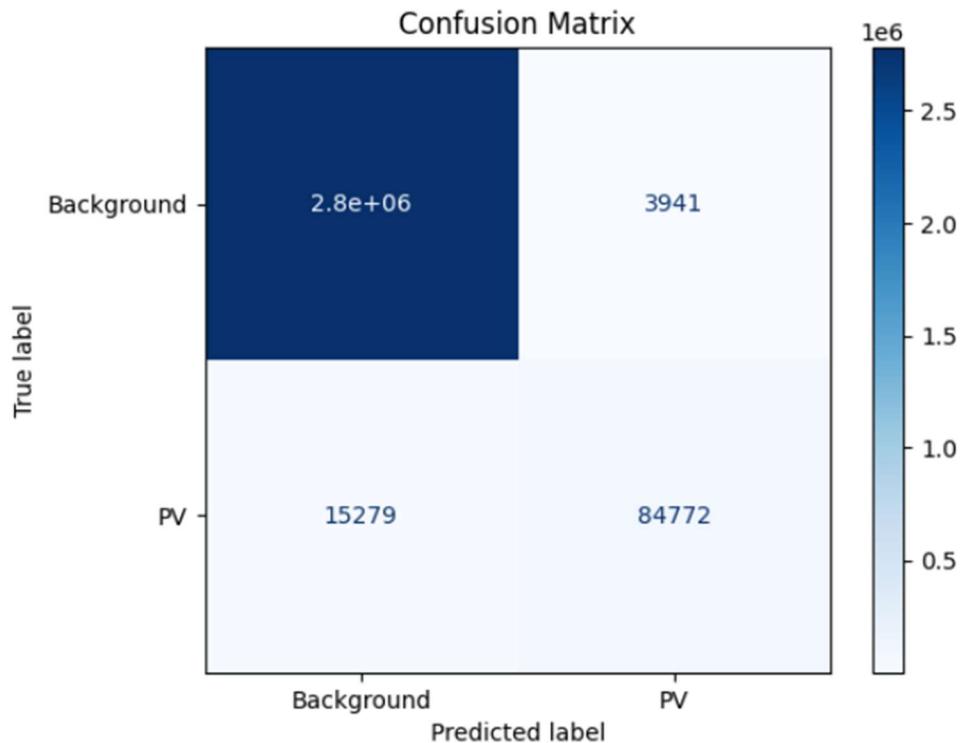
lr=0.0009,



TP:84772 TN:2779592 FP:3941 FN:15279

---

Accuracy:	0.9933	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.8473	$TP / (TP + FN)$
Specificity:	0.9986	$TN / (TN + FP)$
Precision:	0.9556	$TP / (TP + FP)$
F1-Score:	0.8982	$(2 * precision * recall) / (precision + recall)$
IoU:	0.8152	$TP / (TP + FN + FP)$



### Col2 X (reproducing Col2)

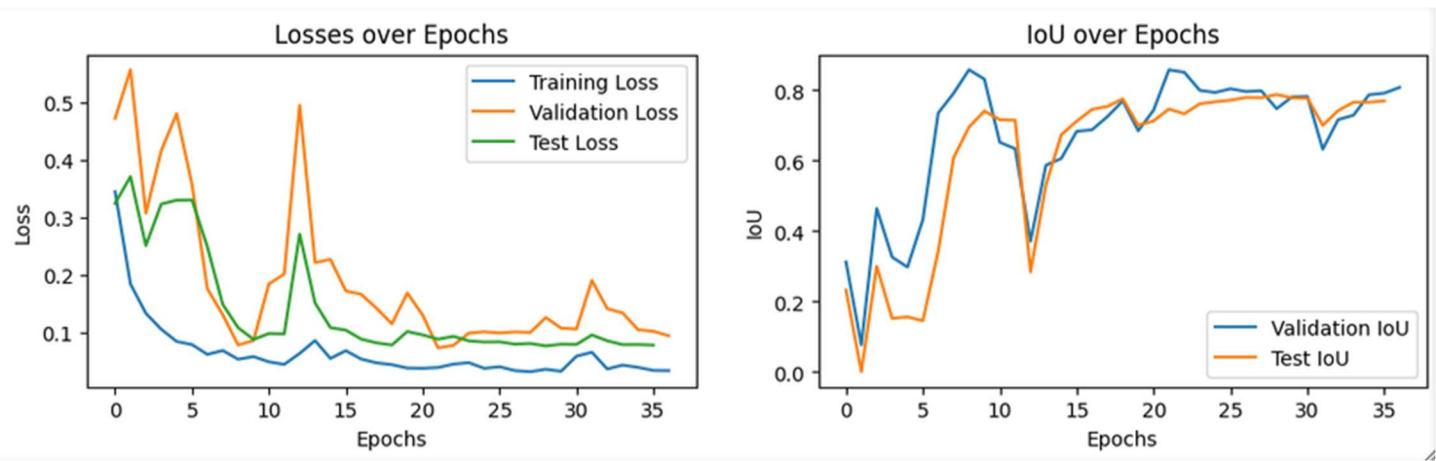
weight\_decay=1e-4

pos\_weight=2.0

early\_stopping = EarlyStopping(patience=15, min\_delta=0.00005)

lr=0.0006,

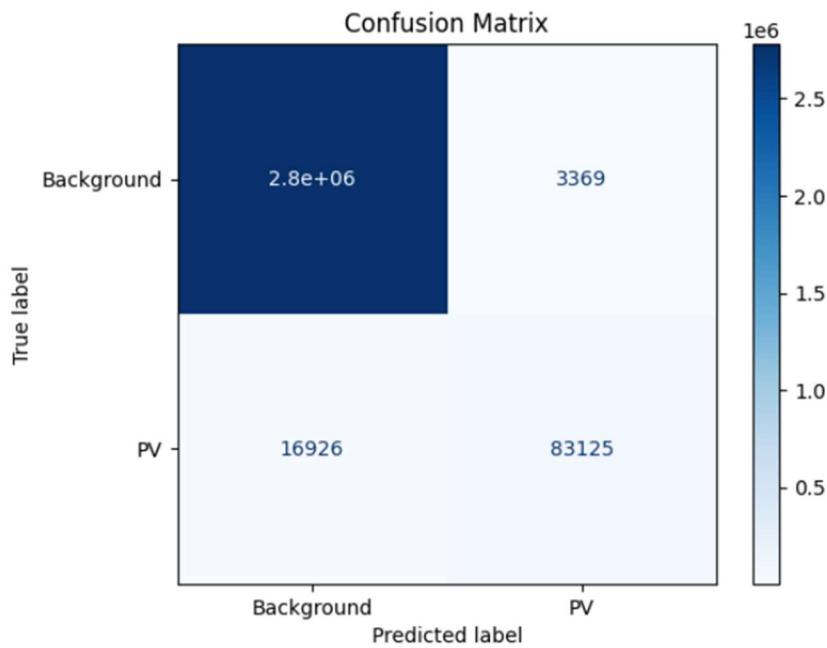
weight\_decay=1e-5



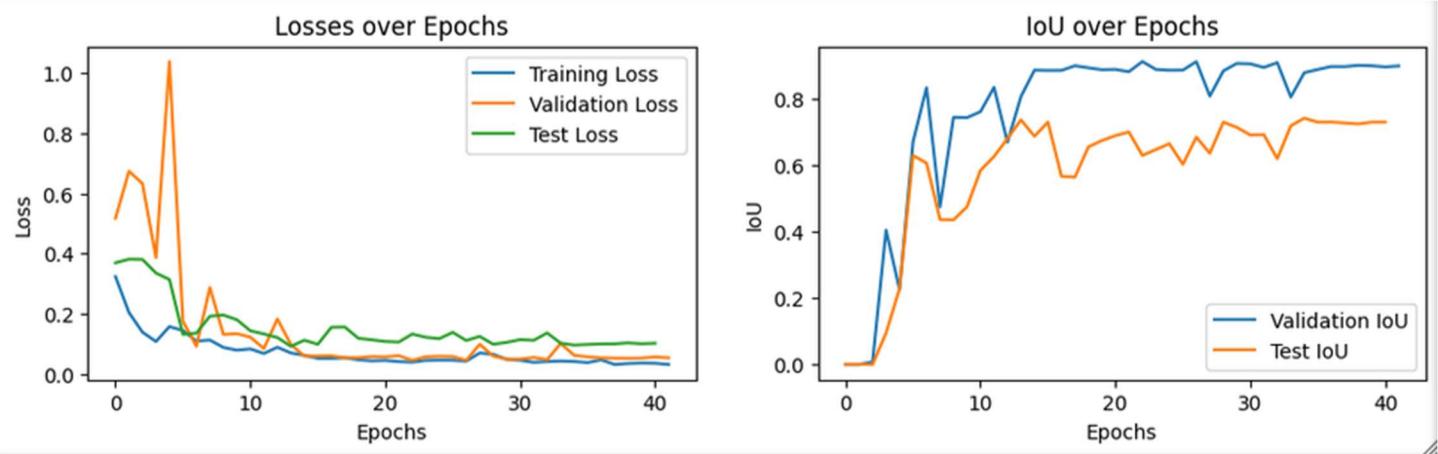
TP:83125      TN:2780164      FP:3369      FN:16926

---

Accuracy:  $0.9930 = \frac{(TP + TN)}{(TP + TN + FP + FN)}$   
 Recall:  $0.8308 = \frac{TP}{(TP + FN)}$   
 Specificity:  $0.9988 = \frac{TN}{(TN + FP)}$   
 Precision:  $0.9610 = \frac{TP}{(TP + FP)}$   
 F1-Score:  $0.8912 = \frac{2 * precision * recall}{precision + recall}$   
 IoU:  $0.8038 = \frac{TP}{(TP + FN + FP)}$



## Col2 XX



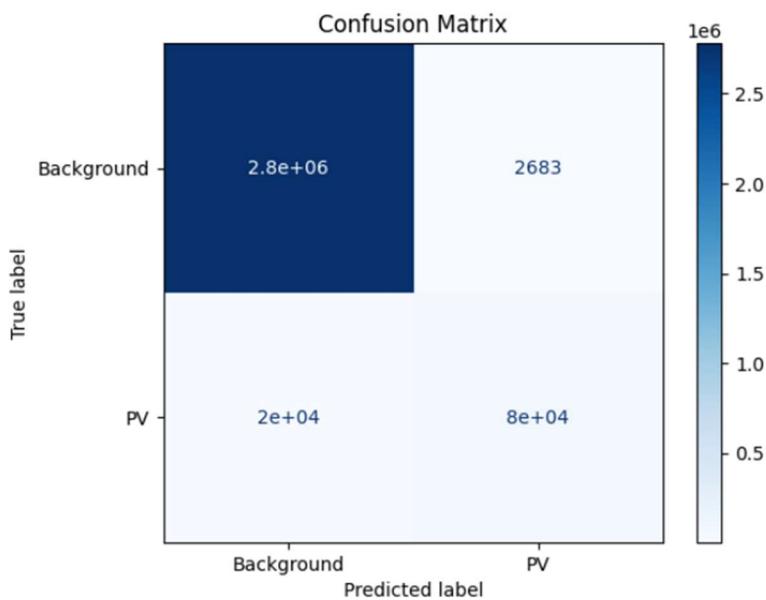
pos\_weight=2.0, neg\_weight=1.0):

```
jaccard_loss = 0.7 * jaccard_loss_positive + 0.3 * jaccard_loss_negative
total_loss = 0.4 * bce_loss + 0.2 * focal_loss + 0.4 * jaccard_loss
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0006, weight_decay=1e-5)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=5, verbose=True)
early_stopping = EarlyStopping(patience=15, min_delta=0.00005)
```

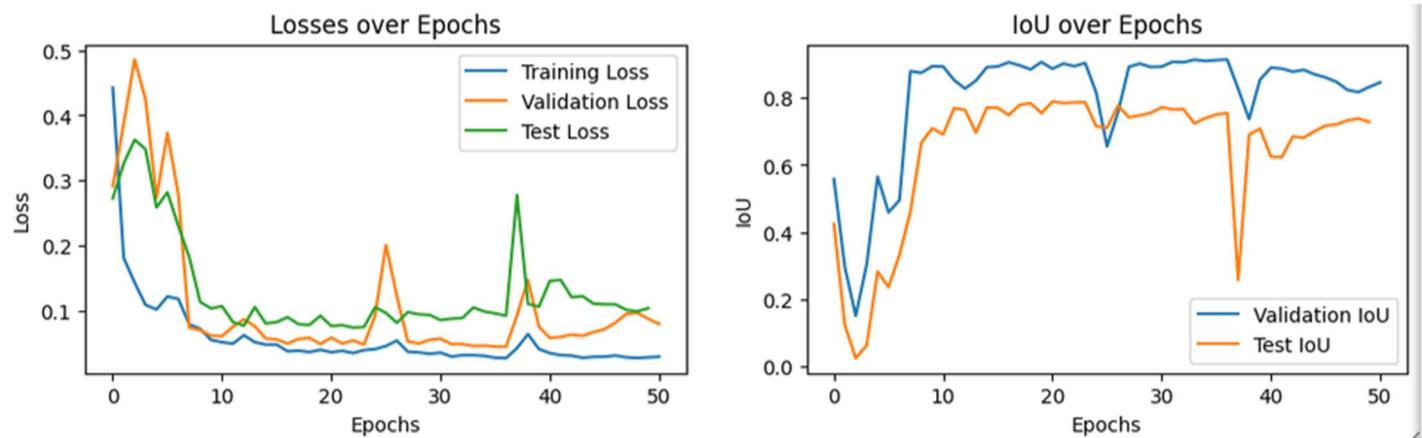
EPOCHS = 41

TP:80081 TN:2780850 FP:2683 FN:19970

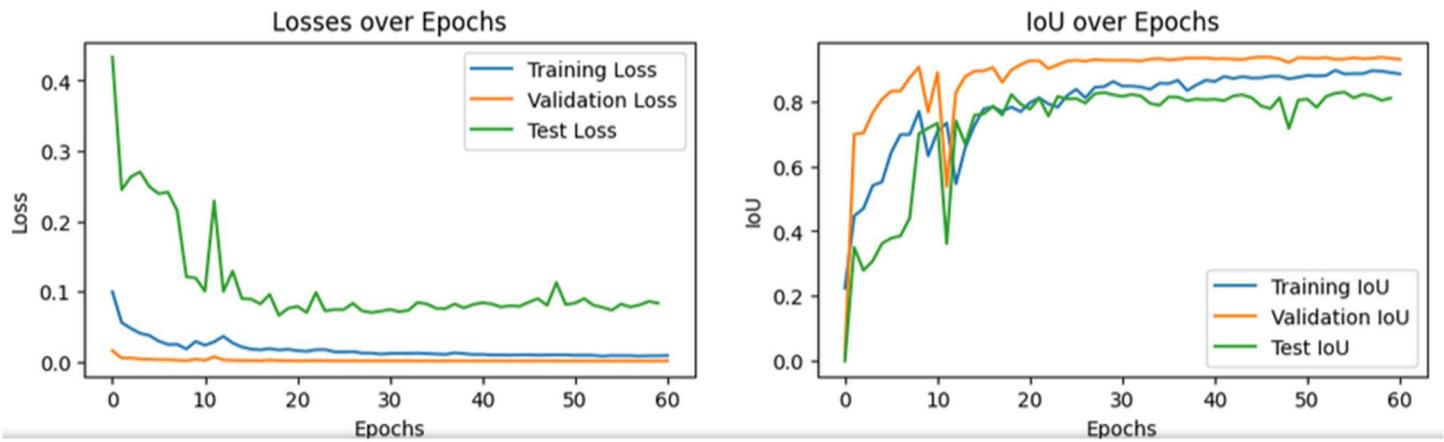
Accuracy:	0.9921	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.8004	$TP / (TP + FN)$
Specificity:	0.9990	$TN / (TN + FP)$
Precision:	0.9676	$TP / (TP + FP)$
F1-Score:	0.8761	$(2 * precision * recall) / (precision + recall)$
IoU:	0.7795	$TP / (TP + FN + FP)$



### Add Threshold as a global parameter to scan it (g\_threshold)

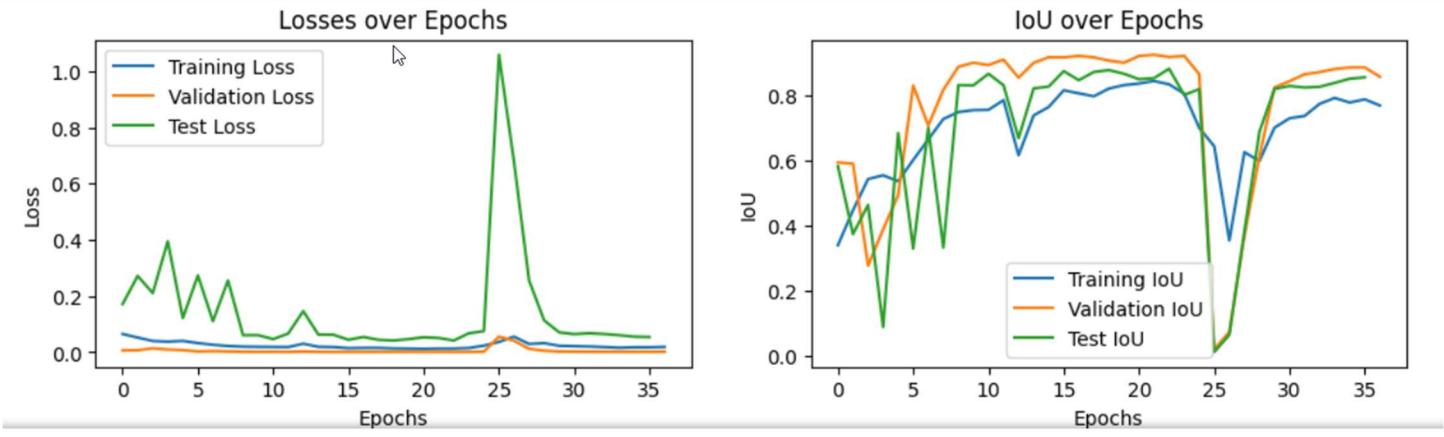


## Rebalancing the sets:



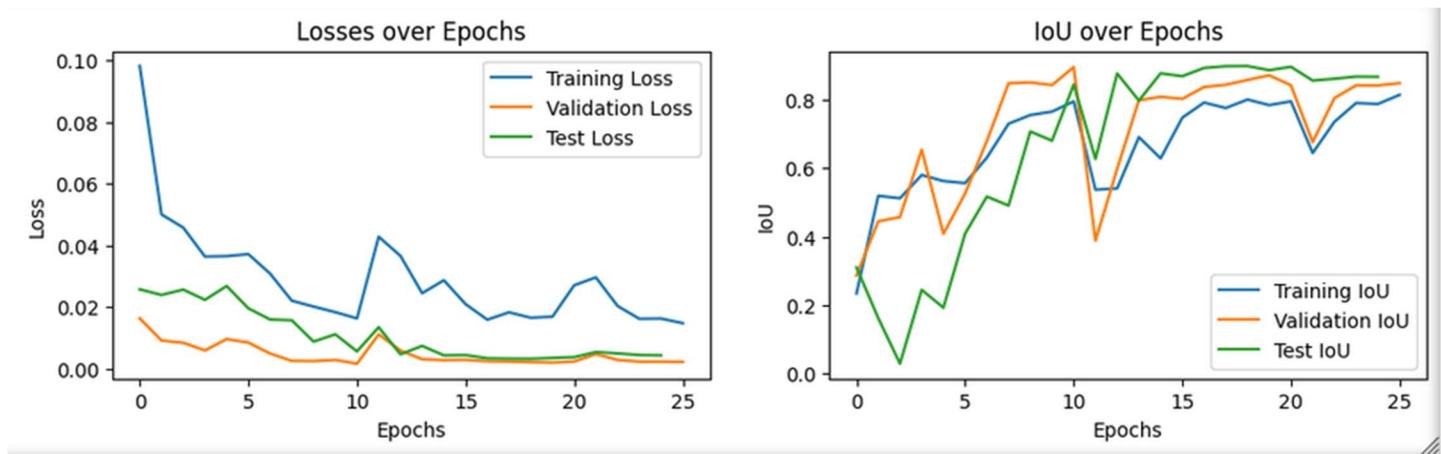
I noticed predictions problems with captures including water at the test set. Then, considering there are not sets with massive amounts of water I swap the Piraporá set and Hélio Valgas between test set and train set.

A high peak on the test loss and overall metrics fall.



Then, I'm filtering all background masks from the val set.

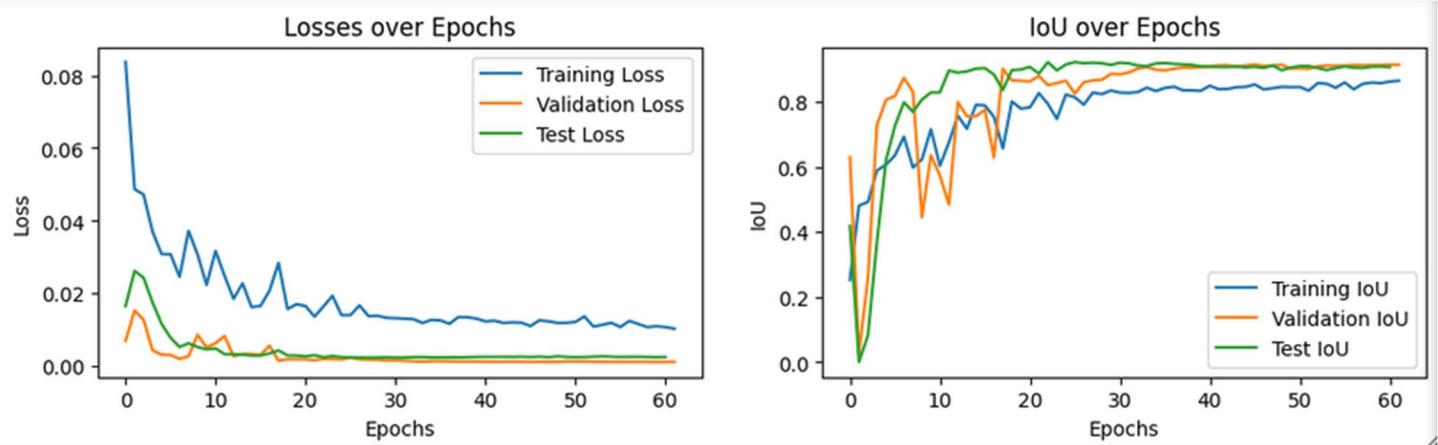
Training set preforming worse than val and test sets.



- I decreased the regularization to 1e-6
- Increased the EarlyStop's patience from 15 to 20 to force more Epochs expecting that train curves improve over epochs.

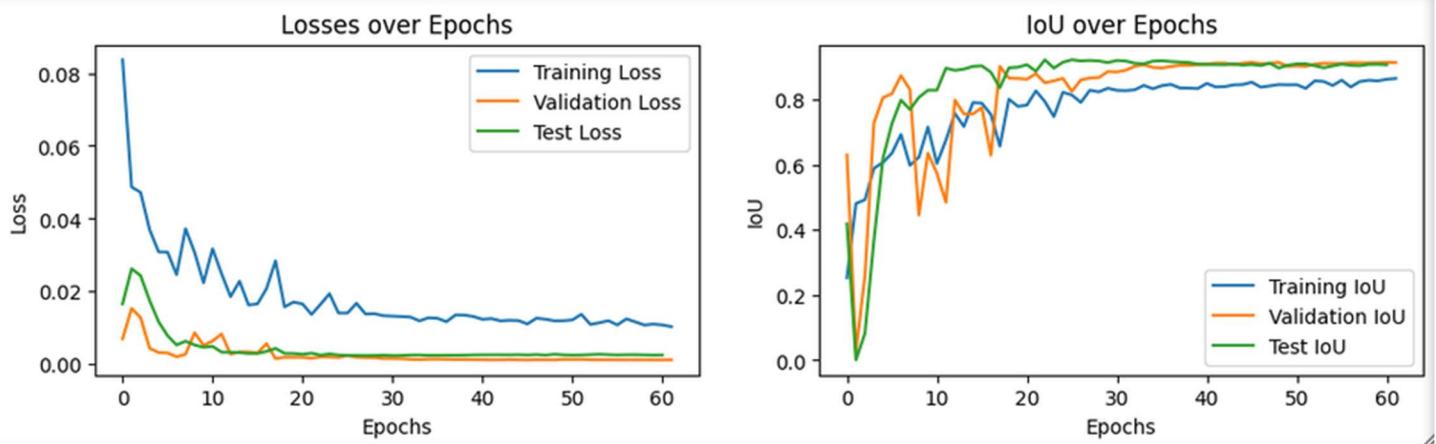
v17-t

retraining after rebalancing the sets

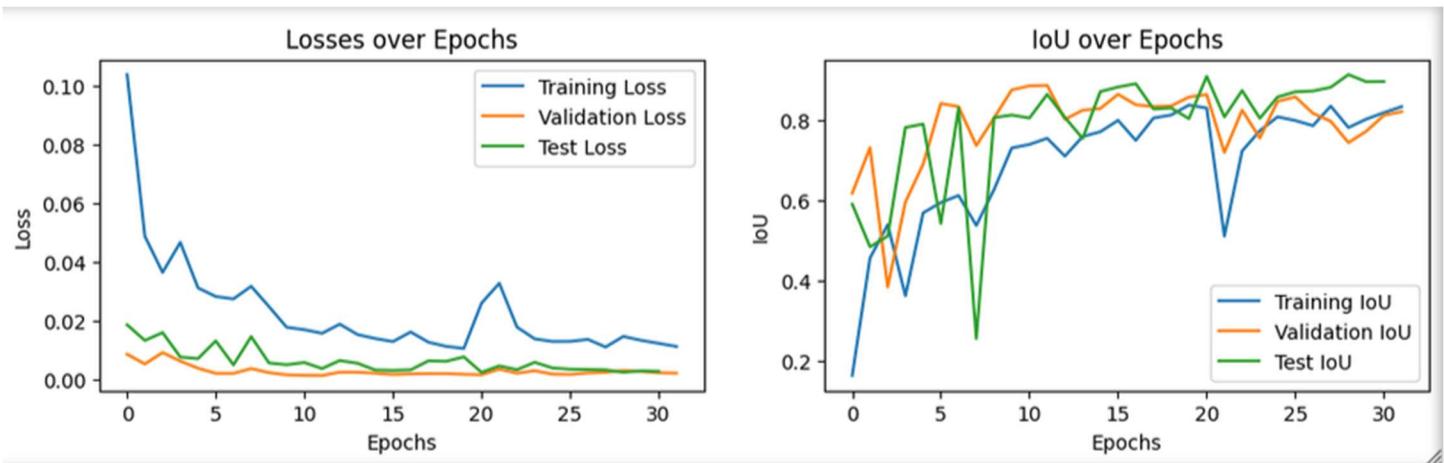


Analysis with or without all-bkgnd images:

- 1) Not considering all-bkgnd images at any set



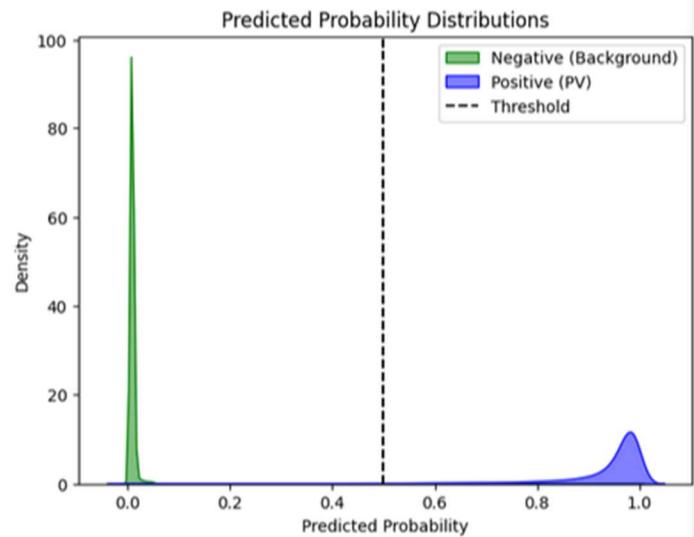
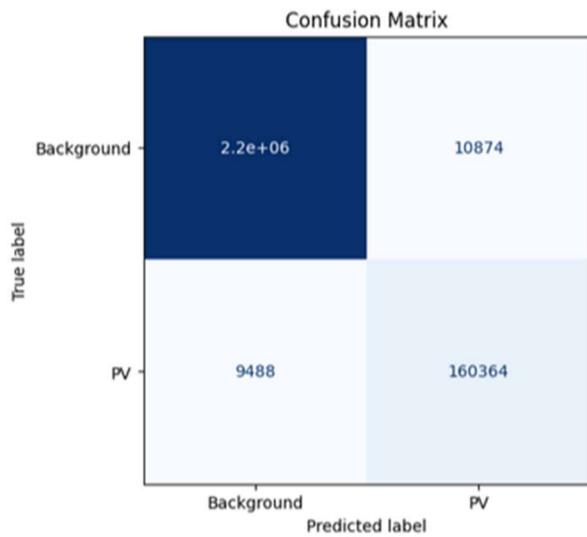
2) Considering all-bkgnd images at all set



Test:

TP:160364      TN:2178570      FP:10874      FN:9488

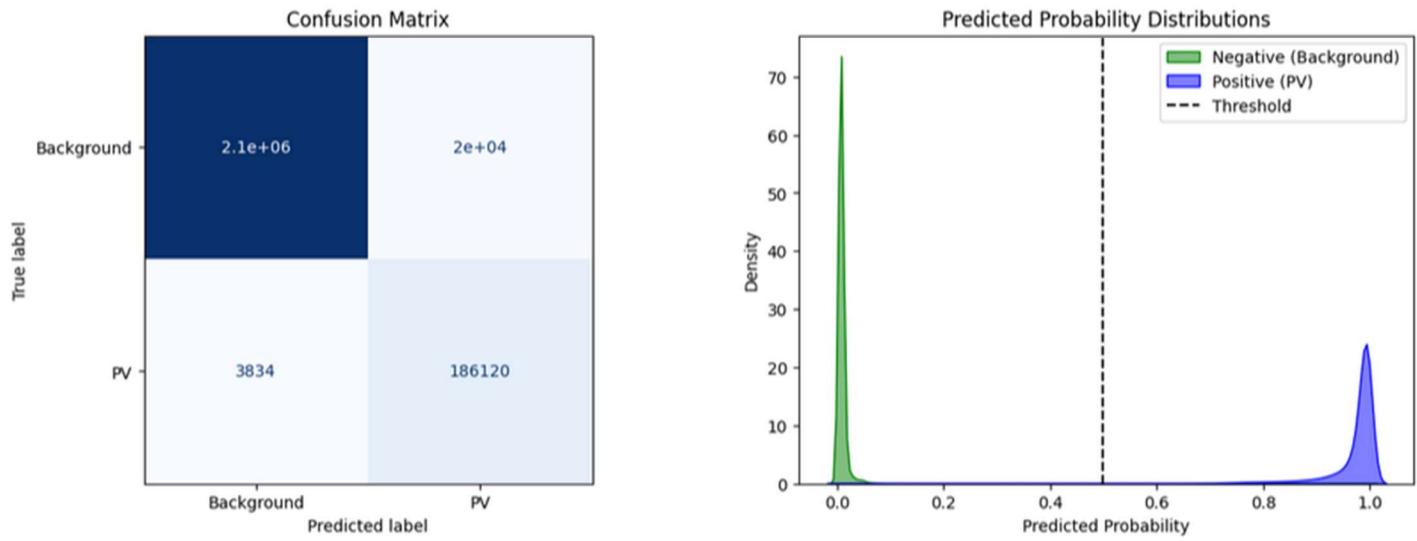
Accuracy:	0.9914	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.9441	$TP / (TP + FN)$
Specificity:	0.9950	$TN / (TN + FP)$
Precision:	0.9365	$TP / (TP + FP)$
F1-Score:	0.9403	$(2 * precision * recall) / (precision + recall)$
IoU:	0.8873	$TP / (TP + FN + FP)$



## Validation:

TP:186120      TN:2083801      FP:20005      FN:3834

Accuracy:	$0.9896$	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	$0.9798$	$TP / (TP + FN)$
Specificity:	$0.9905$	$TN / (TN + FP)$
Precision:	$0.9029$	$TP / (TP + FP)$
F1-Score:	$0.9398$	$(2 * precision * recall) / (precision + recall)$
IoU:	$0.8865$	$TP / (TP + FN + FP)$



## Train:

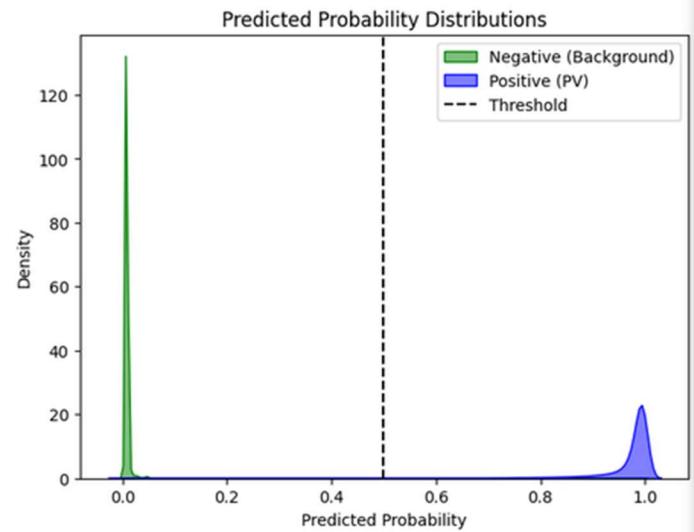
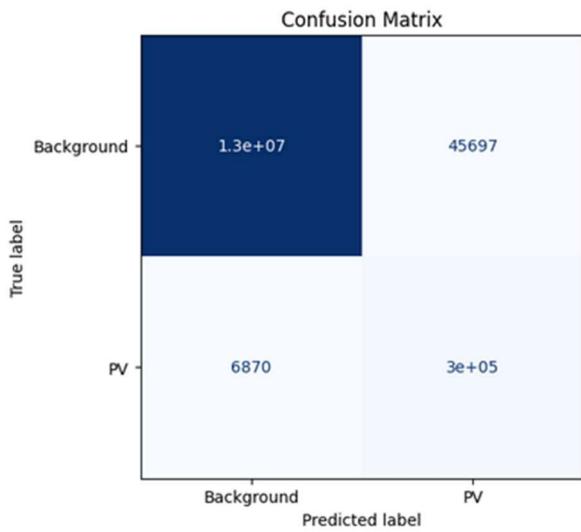
TP:296110

TN:12824059

FP:45697

FN:6870

Accuracy:	$0.9960 = \frac{(TP + TN)}{(TP + TN + FP + FN)}$
Recall:	$0.9773 = \frac{TP}{(TP + FN)}$
Specificity:	$0.9964 = \frac{TN}{(TN + FP)}$
Precision:	$0.8663 = \frac{TP}{(TP + FP)}$
F1-Score:	$0.9185 = \frac{2 * precision * recall}{precision + recall}$
IoU:	$0.8492 = \frac{TP}{(TP + FN + FP)}$



## Conclusion:

- Excluding all-bkgnd mask images produces a better performance (higher scoring metric).

## Possible issue?

- The test and validation set performance is higher than the training set (lower loss and higher score!)
- It could be a problem, then I will investigate:

## Investigation:

- Review the images that compose the sets and search for mixed images between the sets.  
12.02.2024: I checked the images and there's no mixture of plant images between the sets.

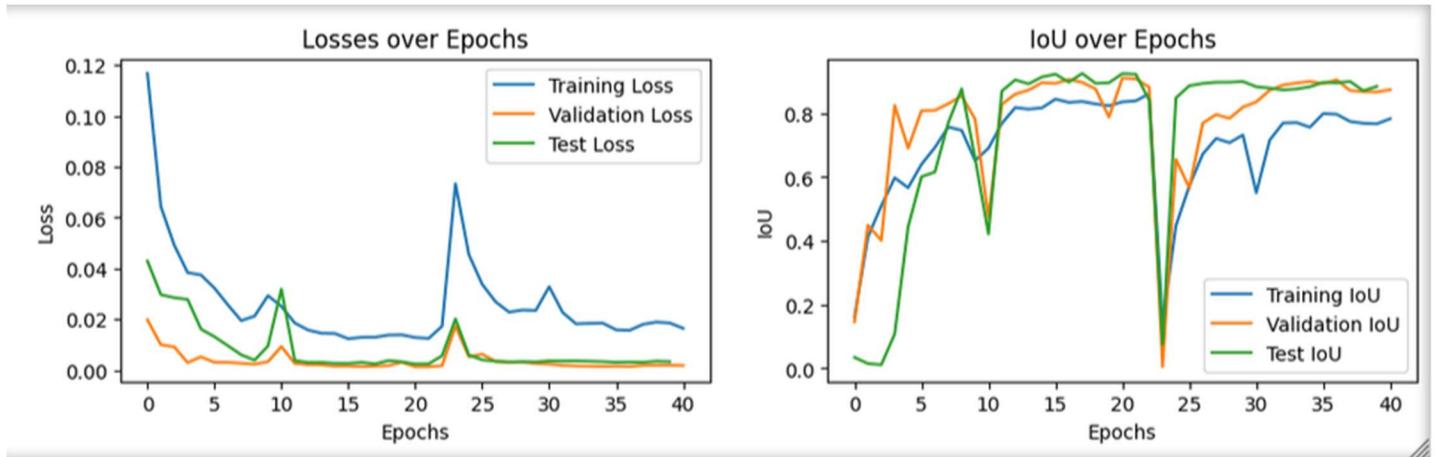
## Next Steps

- 1) Reduce augmentation complexity and evaluate its impact on training IoU.
- 2) Adjust regularization parameters and learning rate to improve training loss convergence.
- 3) Validate dataset splits to ensure similar difficulty levels across sets.
- 4) Monitor the behavior of individual samples in the training set for further debugging (identify problematic samples causing high loss).

- 1) Reduce augmentation complexity and evaluate its impact on training IoU.

Then, comment the Rotations:

```
augmentation_transform = SynchronizedTransform([  
    transforms.RandomHorizontalFlip(),  
    transforms.RandomVerticalFlip(),  
    # transforms.RandomChoice([  
    #     transforms.RandomRotation(0),  
    #     transforms.RandomRotation(90),  
    #     transforms.RandomRotation(180),  
    #     transforms.RandomRotation(270)  
    # ]),  
    transforms.ToTensor()  
])
```



### Previous

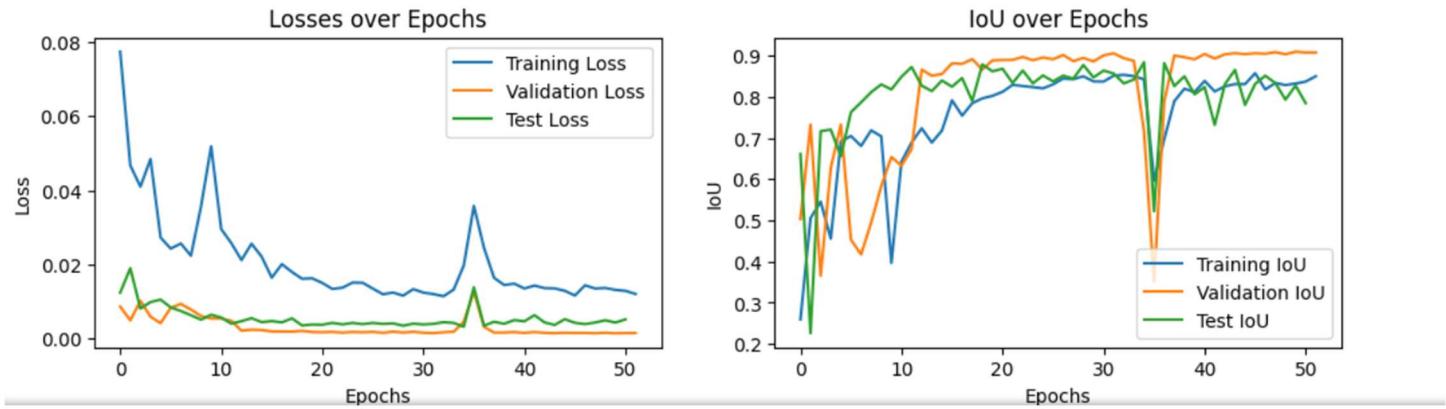
	FP	FN	F1	IoU
Test	2110	12301	95,63	91,62
Val	13928	4091	95,38	91,16
Train	26171	7439	94,62	89,79

### After changes

	FP	FN	F1	IoU
Test	2720	8176	96,74	93,69
Val	17373	1959	95,11	90,68
Train	24162	6803	95,03	90,53

Conclusion: The model's performance has improved after the changes.

2) Adjust regularization parameters and learning rate to improve training loss convergence.



#### Regularization = 0

```
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0006, weight_decay=0 )
```

Before regu=0

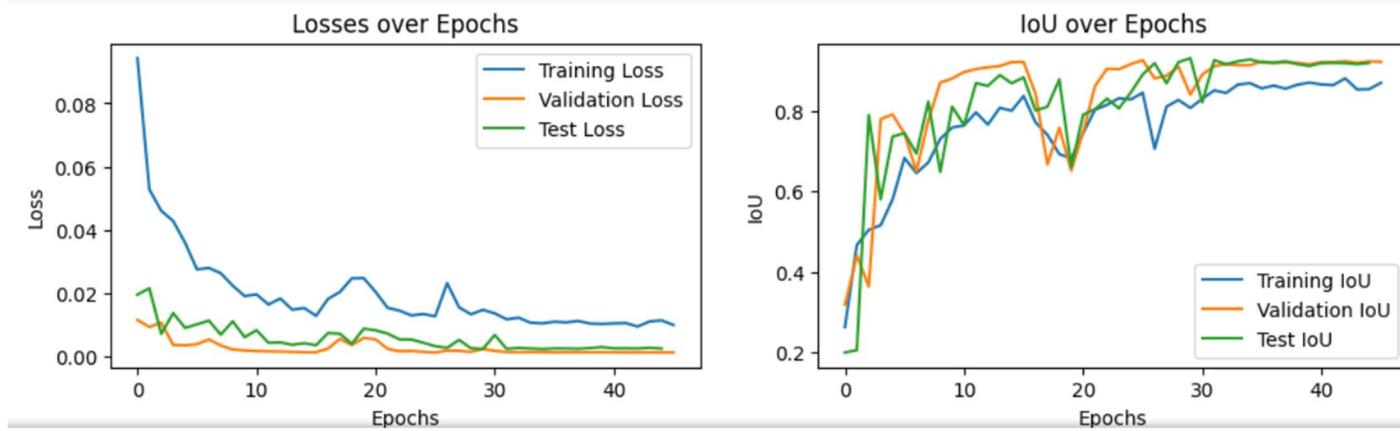
	FP	FN	F1	IoU
Test	2720	8176	96,74	93,69
Val	17373	1959	95,11	90,68
Train	24162	6803	95,03	90,53

After regu=0

	FP	FN	F1	IoU
Test	2352	26765	90,77	83,09
Val	12844	5579	95,24	90,92
Train	29022	5268	94,55	89,67

Conclusion: cancelling regularization decreases the model's performance.

```
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0006, weight_decay=1e-3)
```



Before (regu=1e-6)

	FP	FN	F1	IoU
Test	2720	8176	96,74	93,69
Val	17373	1959	95,11	90,68
Train	24162	6803	95,03	90,53

regu=0

	FP	FN	F1	IoU
Test	2352	26765	90,77	83,09
Val	12844	5579	95,24	90,92
Train	29022	5268	94,55	89,67

regu=1e-3

	FP	FN	F1	IoU
Test	2169	15187	94,69	89.91
Val	9248	5649	96,12	92.52
Train	25782	8292	94,53	89,64

Conclusion: The model's performance has improved after the changes.

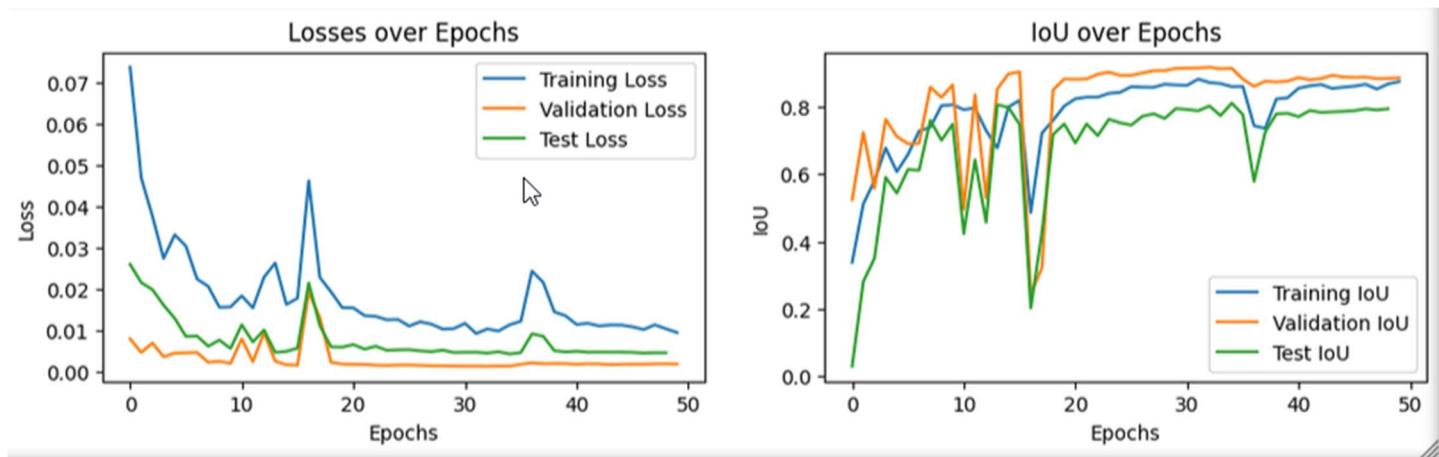
3) Validate dataset splits to ensure similar difficulty levels across sets.

With regu=1e-6

1<sup>st</sup>)

Swap:

- Hélio Valgas from test to train because it was not representative.
- Apio Cardoso from train to test



Before (regu=1e-6)

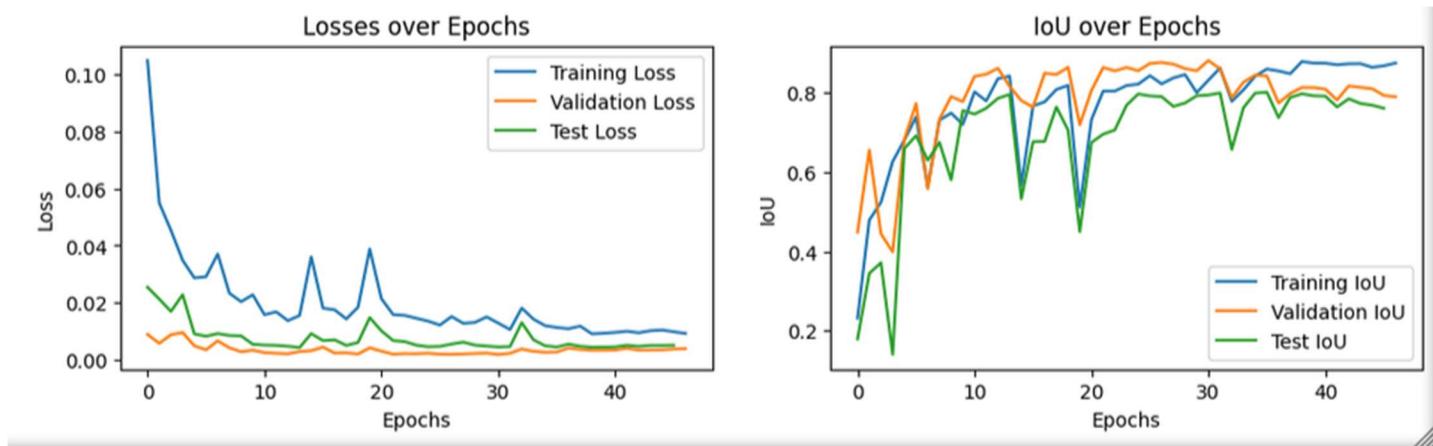
	FP	FN	F1	IoU
Test	2720	8176	96,74	93,69
Val	17373	1959	95,11	90,68
Train	24162	6803	95,03	90,53

After 1<sup>st</sup> swap (regu=1e-6)

	FP	FN	F1	IoU
Test	10492	2684	89,00	80,18
Val	12467	4238	95,70	91,75
Train	22838	8634	96,26	92,84

## 2<sup>nd</sup> Swap:

- Add Arinos from train to val



Before (regu=1e-6)

	FP	FN	F1	IoU
Test	2720	8176	96,74	93,69
Val	17373	1959	95,11	90,68
Train	24162	6803	95,03	90,53

After 1<sup>st</sup> swap (regu=1e-6)

	FP	FN	F1	IoU
Test	10492	2684	89,00	80,18
Val	12467	4238	95,70	91,75
Train	22838	8634	96,26	92,84

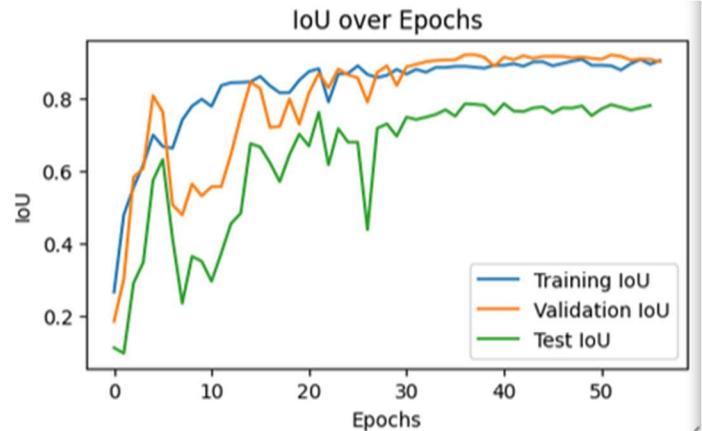
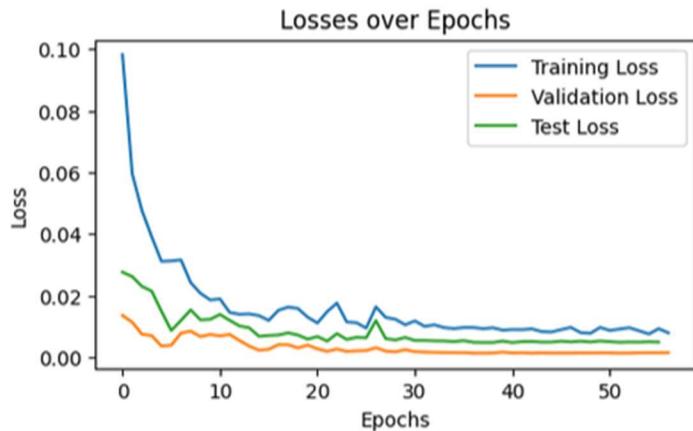
After 1<sup>st</sup> and 2<sup>nd</sup> swap (regu=1e-6)

	FP	FN	F1	IoU
Test	11297	2696	88,40	79,20
Val	12478	21461	93,68	88,10
Train	18104	8332	96,10	92,49

Conclusion: After considering the performances I decided to roll back the last swap (2<sup>nd</sup>).

#### 4) More changes: LR and early stop

LR= 0.0003 #reduced from 0.0006



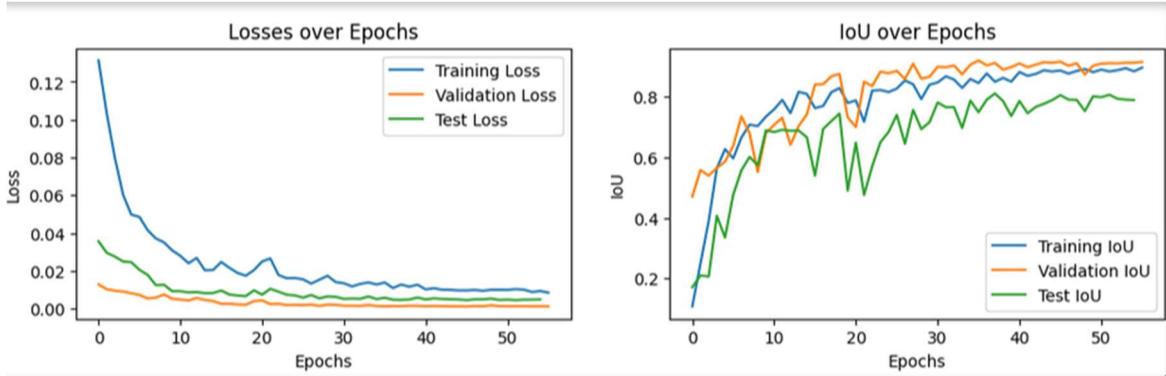
Before (1<sup>st</sup> swap and regu=1e-6) LR=0.0006

	FP	FN	F1	IoU
Test	10492	2684	89,00	80,18
Val	12467	4238	95,70	91,75
Train	22838	8634	96,26	92,84

After 1<sup>st</sup> and 2<sup>nd</sup> swap (regu=1e-6) LR=0.0003

	FP	FN	F1	IoU
Test	11715	3174	87,65	78,01
Val	12774	3870	95,72	91,79
Train	18198	9529	96,71	93,63

LR=0.0001

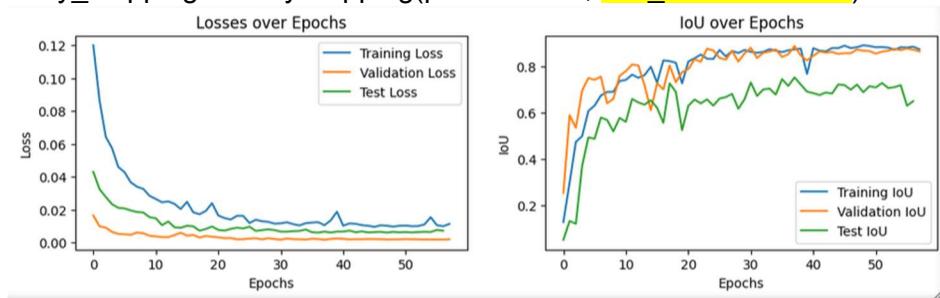


After 1<sup>st</sup> and 2<sup>nd</sup> swap (regu=1e-6) LR=0.0001

	FP	FN	F1	IoU
Test	9864	2947	89,23	80,55
Val	12993	4002	95,63	91,63
Train	17203	10308	96,73	93,66

I will extend the early stop

early\_stopping = EarlyStopping(patience=20, min\_delta=0.00001) # it was set as min\_delta=0.00005



before LR=0.0001, min\_delta=0.00005

	FP	FN	F1	IoU
Test	9864	2947	89,23	80,55
Val	12993	4002	95,63	91,63
Train	17203	10308	96,73	93,66

before LR=0.0001, min\_delta=0.00001

	FP	FN	F1	IoU
Test	29095	1962	77,67	63,50
Val	28506	3457	92,11	85,37
Train	24433	6303	96,39	93,03

Conclusion: extending the early stop doesn't improve the results and introduces more overfitting.

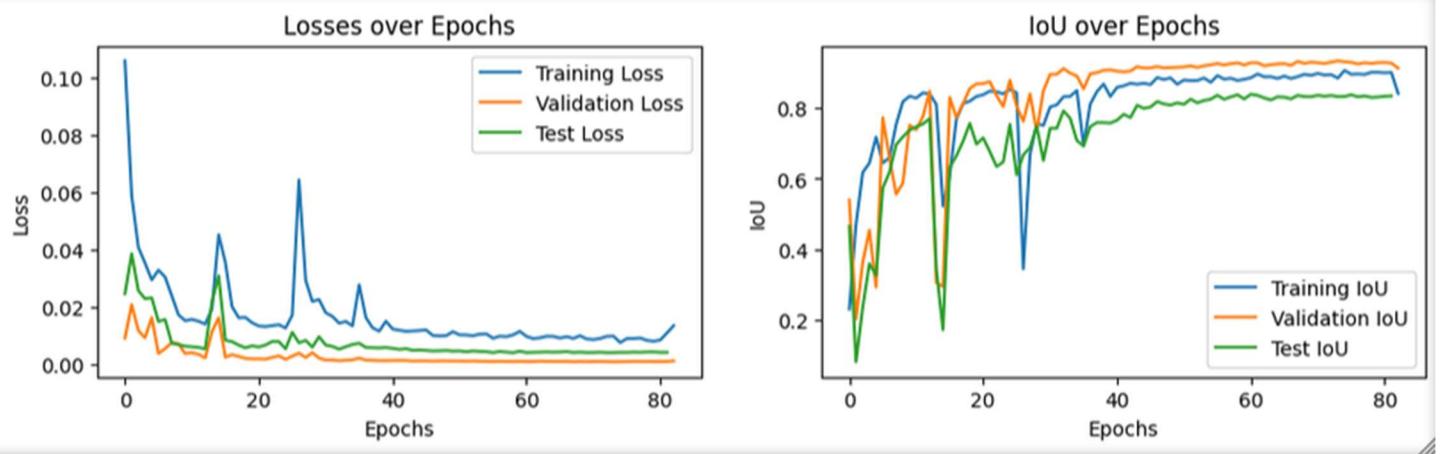
## 5) More changes: Loss function

First, experience with the jaccard\_loss weights:

$$\text{jaccard\_loss} = 0.7 * \text{jaccard\_loss\_positive} + 0.3 * \text{jaccard\_loss\_negative}$$

To:

$$\text{jaccard\_loss} = 0.8 * \text{jaccard\_loss\_positive} + 0.2 * \text{jaccard\_loss\_negative}$$



before

	FP	FN	F1	IoU
Test	9864	2947	89,23	80,55
Val	12993	4002	95,63	91,63
Train	17203	10308	96,73	93,66

after

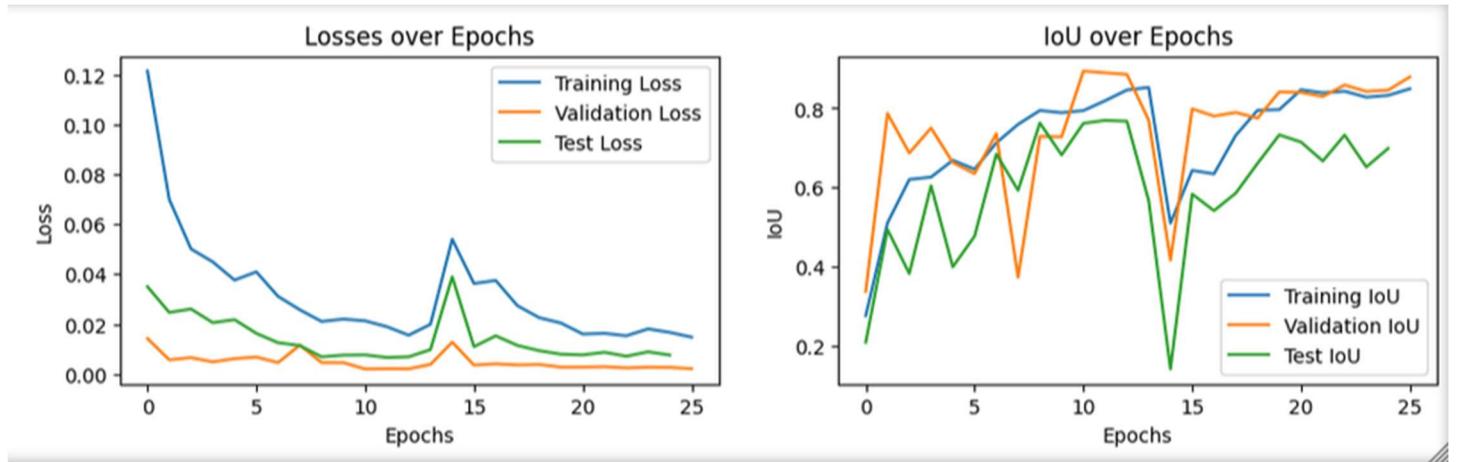
	FP	FN	F1	IoU
Test	5524	4759	90,88	83,28
Val	9775	3222	96,64	93,49
Train	13910	9733	97,18	94,51

All sets have improved their performance.

Continue increasing the positive class weight:

From:  $\text{jaccard\_loss} = 0.8 * \text{jaccard\_loss\_positive} + 0.2 * \text{jaccard\_loss\_negative}$

To:  $\text{jaccard\_loss} = 0.9 * \text{jaccard\_loss\_positive} + 0.1 * \text{jaccard\_loss\_negative}$



Before (best till now)

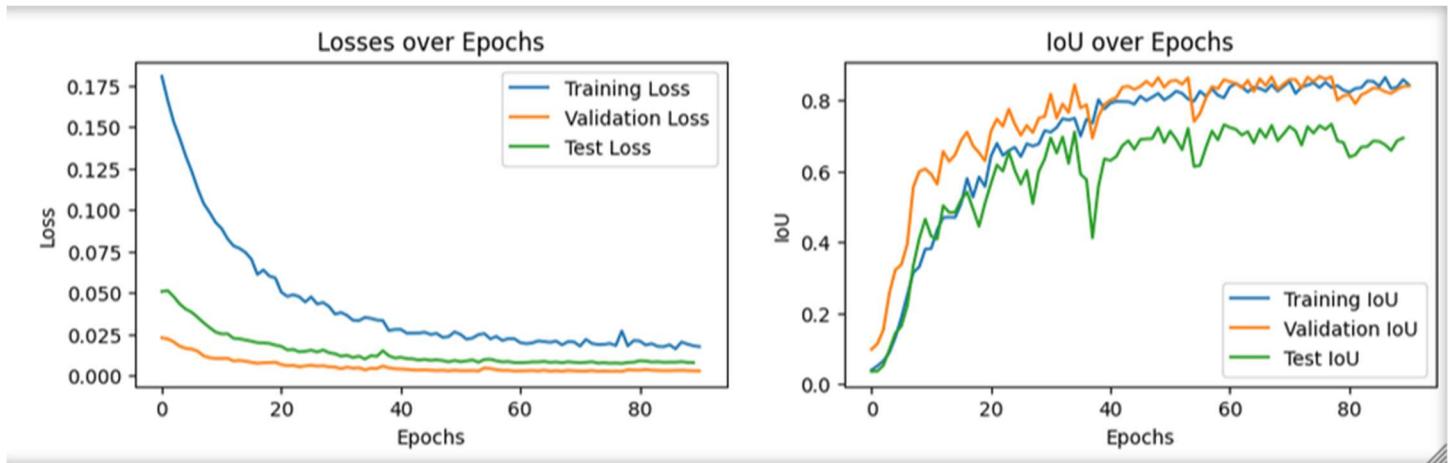
	FP	FN	F1	IoU
Test	5524	4759	90,88	83,28
Val	9775	3222	96,64	93,49
Train	13910	9733	97,18	94,51

AFTER

	FP	FN	F1	IoU
Test	10118	5609	86,50	76,21
Val	18821	3538	94,34	89,29
Train	22648	16634	95,32	91,06

Apparently, the weights' imbalance is too high.

## 6) More changes: Keep 0.8/0.2 and reduce LR lr=0.00003 (from lr=0.0001)



Before (best till now)

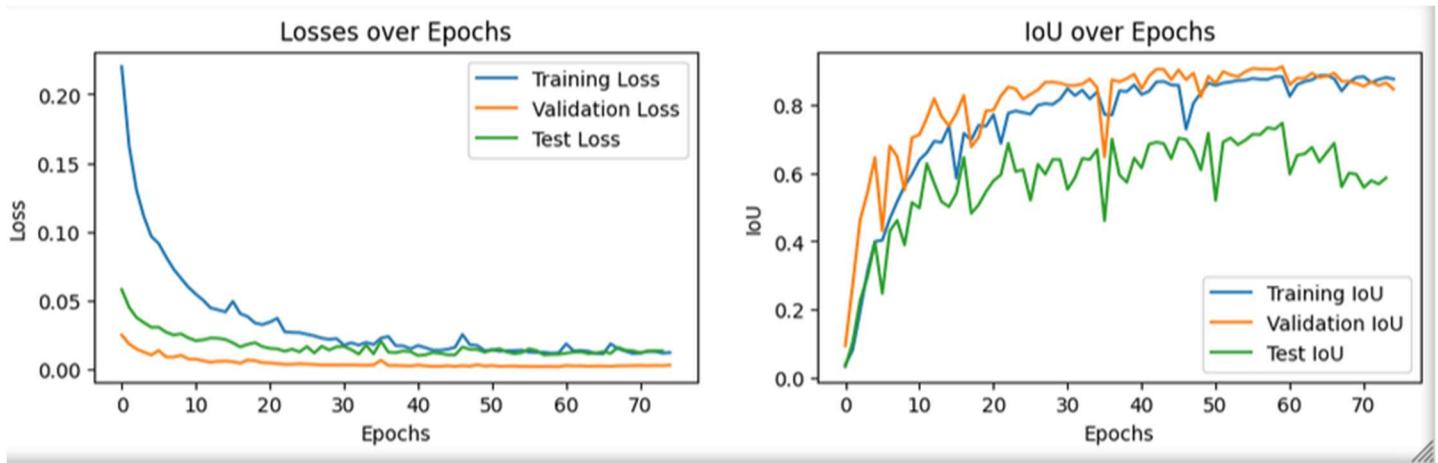
	FP	FN	F1	IoU
Test	5524	4759	90,88	83,28
Val	9775	3222	96,64	93,49
Train	13910	9733	97,18	94,51

AFTER

	FP	FN	F1	IoU
Test	17937	2231	84,21	72,72
Val	31175	3594	90,36	82,41
Train	32716	6755	95,41	91,22

It didn't work.

Trying to define a new base line to continue working on the loss function:



### Before (best till now)

	FP	FN	F1	IoU
Test	5524	4759	90,88	83,28
Val	9775	3222	96,64	93,49
Train	13910	9733	97,18	94,51

### AFTER

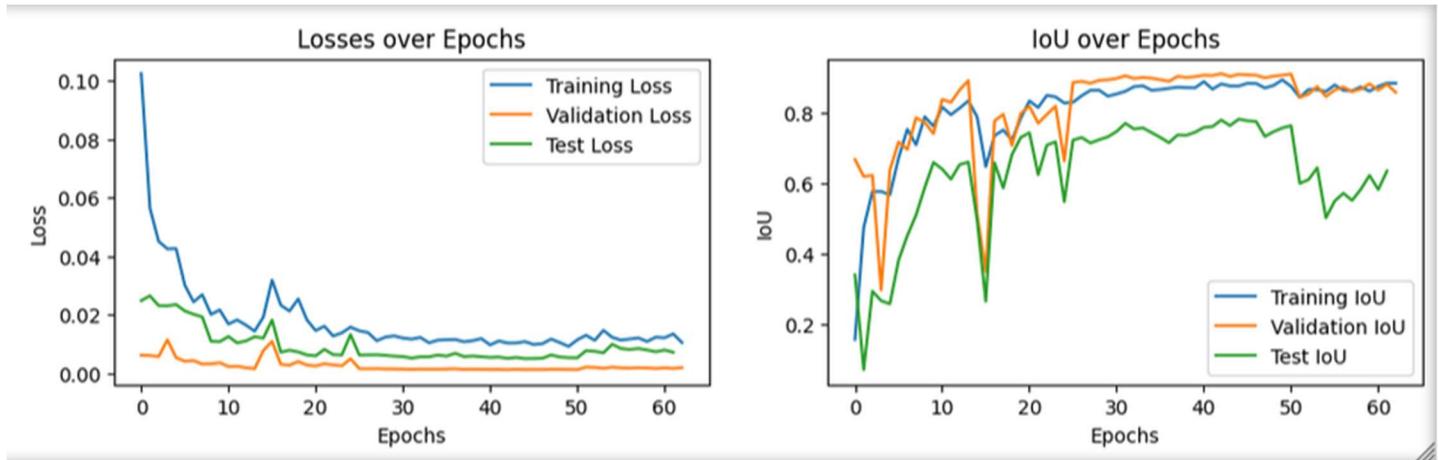
	FP	FN	F1	IoU
Test	11224	5082	86,20	75,74
Val	15950	2308	95,36	91,13
Train	21070	7016	96,69	93,59

I didn't reach the previous perf...

Let's change the  
lr=0.0003,

From: def \_\_init\_\_(self, alpha=0.8, gamma=1.5, pos\_weight=2.0, neg\_weight=1.0):

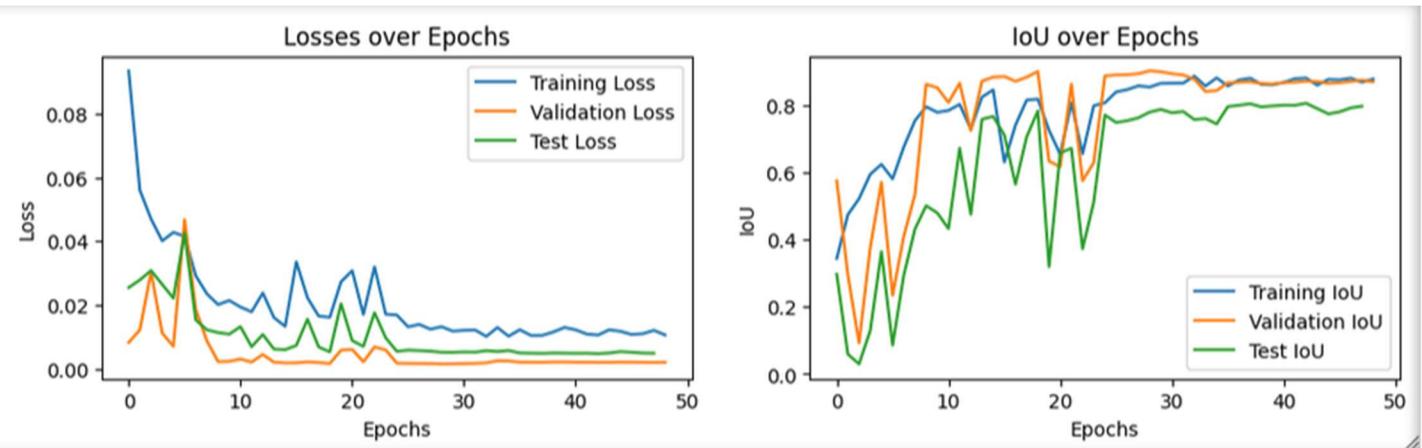
To: def \_\_init\_\_(self, alpha=0.8, gamma=1.5, pos\_weight=1.0, neg\_weight=0.5):



Let's change the

$\text{lr}=0.001$ ,

```
def __init__(self, alpha=0.8, gamma=1.5, pos_weight=2.0, neg_weight=1.0):
```



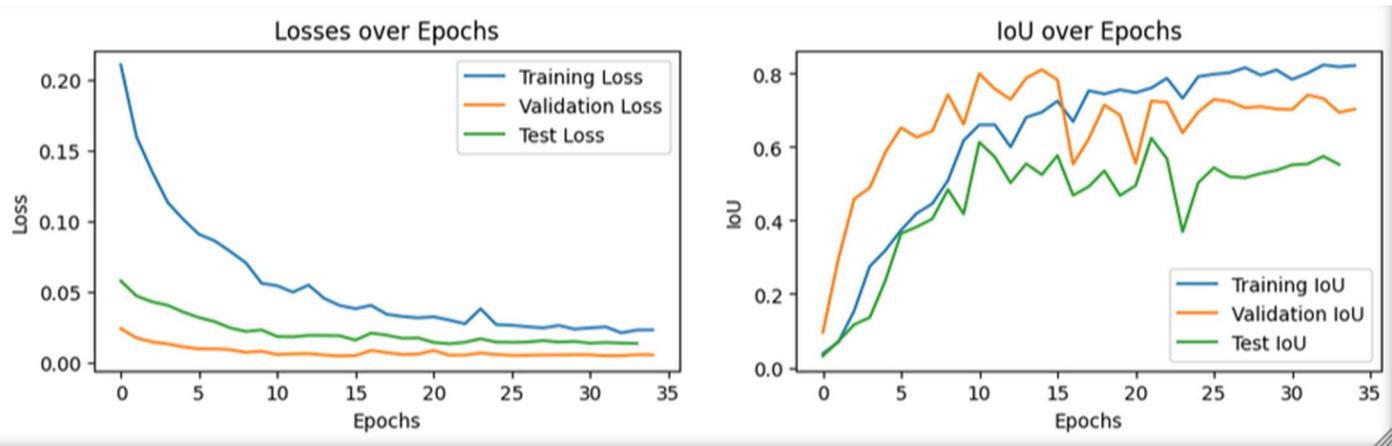
Before (best till now)

	FP	FN	F1	IoU
Test	5524	4759	90,88	83,28
Val	9775	3222	96,64	93,49
Train	13910	9733	97,18	94,51

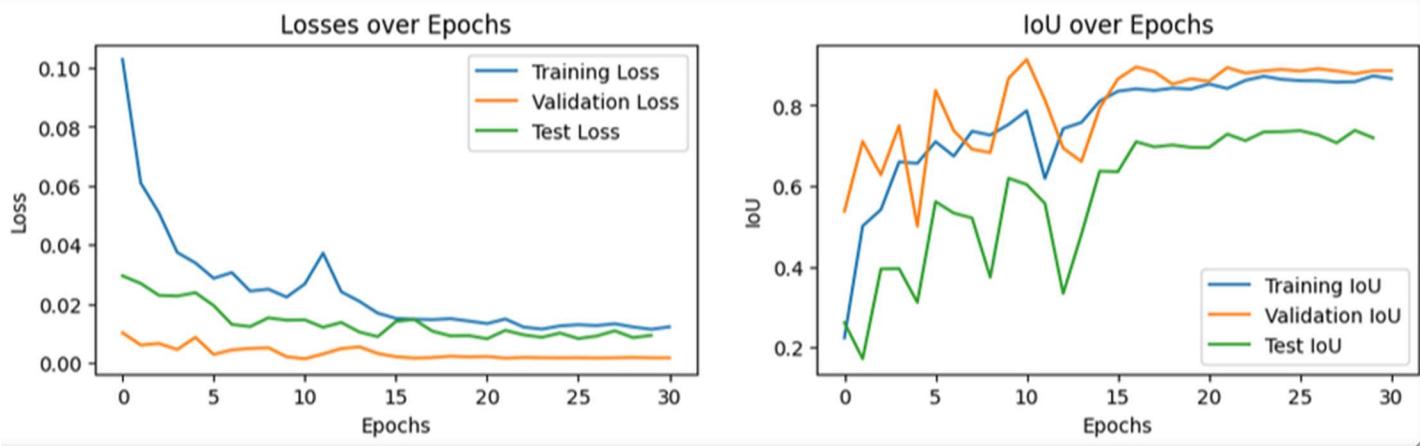
AFTER

	FP	FN	F1	IoU
Test	11834	3360	87,39	77,60
Val	18862	2161	94,70	89,93
Train	25701	8480	95,98	92,28

Let's change again to lr=0.0001,



Let's change again to lr=0.0001,



Before (best till now)

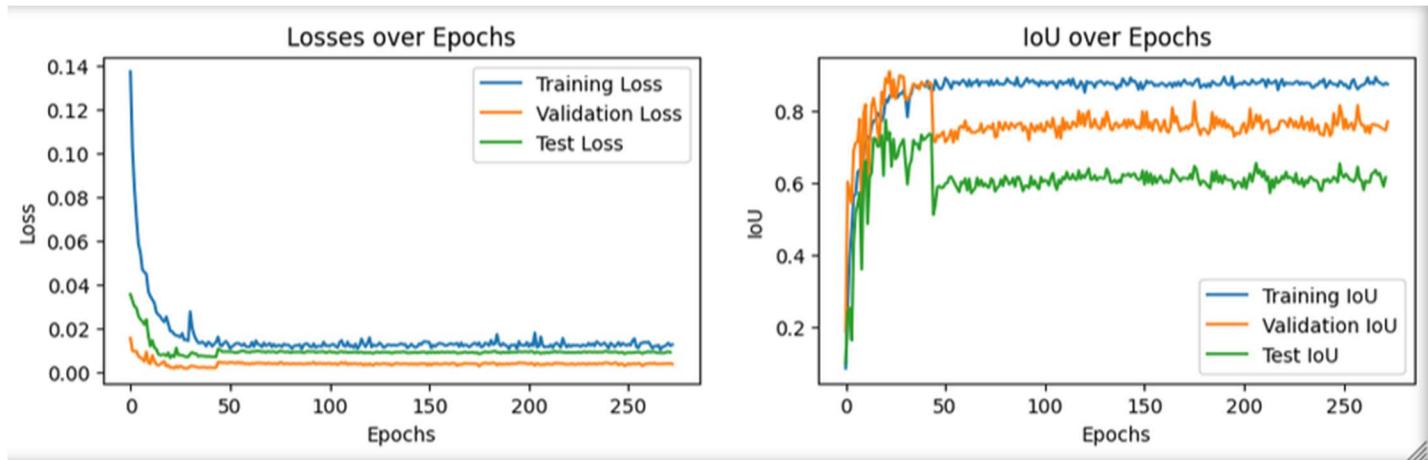
	FP	FN	F1	IoU
Test	5524	4759	90,88	83,28
Val	9775	3222	96,64	93,49
Train	13910	9733	97,18	94,51

AFTER

	FP	FN	F1	IoU
Test	17635	10751	76,12	61,45
Val	16393	2441	95,22	90,22
Train	18877	22485	95,02	90,51

Searching for a good baseline before changing the loss fn weights.

patience=200



Observe how the IoU over Epochs dropped down just before 50 epochs for the val and test sets...

#### Before (best till now)

	FP	FN	F1	IoU	FP/FN
Test	5524	4759	90,88	83,28	1.16
Val	9775	3222	96,64	93,49	3.03
Train	13910	9733	97,18	94,51	1.43

#### AFTER

	FP	FN	F1	IoU	FP/FN
Test	12937	5162	84,89	73,74	2.5
Val	14032	5084	95,08	90,63	2.76
Train	24199	11280	95,81	91,96	2.15

Looking for a new **baseline**:

I couldn't reach the same model's performance as with previous models. I modified something, I haven't found it yet. Then, I'm trying to find out a good baseline before working on the loss function. Now, the test set is outperforming!

First, I will try setting 'the good' hyperparameters, if there's no improvement, I will change the sets and see the impact.

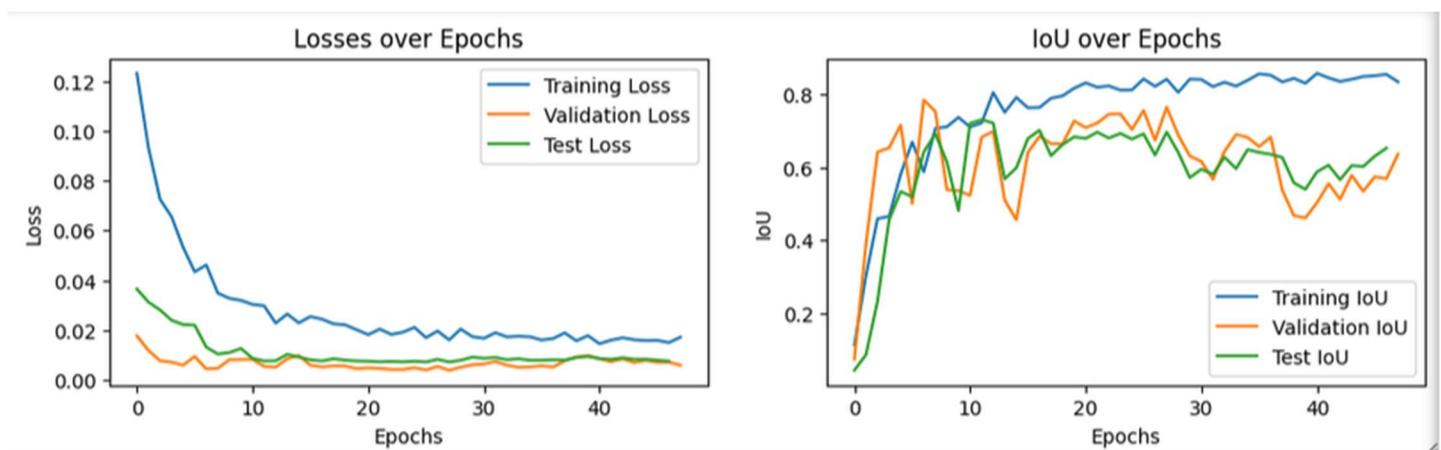
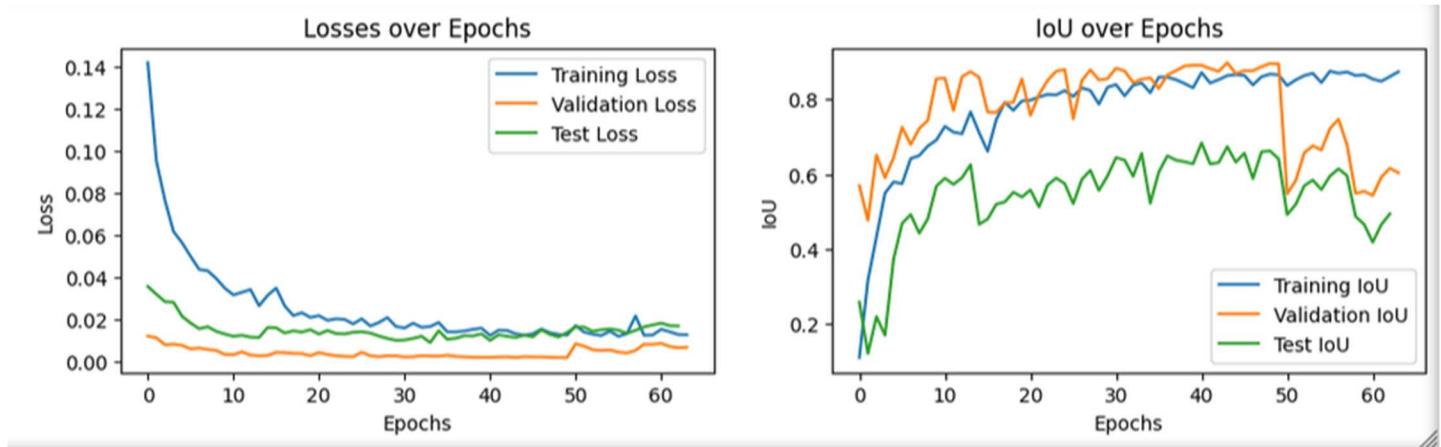
LR = 0.0001, Patience = 20; min\_delta=0.00001, Weight\_decay = 1e-6, Keeping Arinos in the trainset.

Epoch: 55

Discard 0s? True | 0s\_masks/Tot\_imgs: 31/206 || Train Loss: 0.011961 | Train IoU: 0.875

Discard 0s? True | 0s\_masks/Tot\_imgs: 4/35 || Valid Loss: 0.004460 | Valid IoU: 0.722

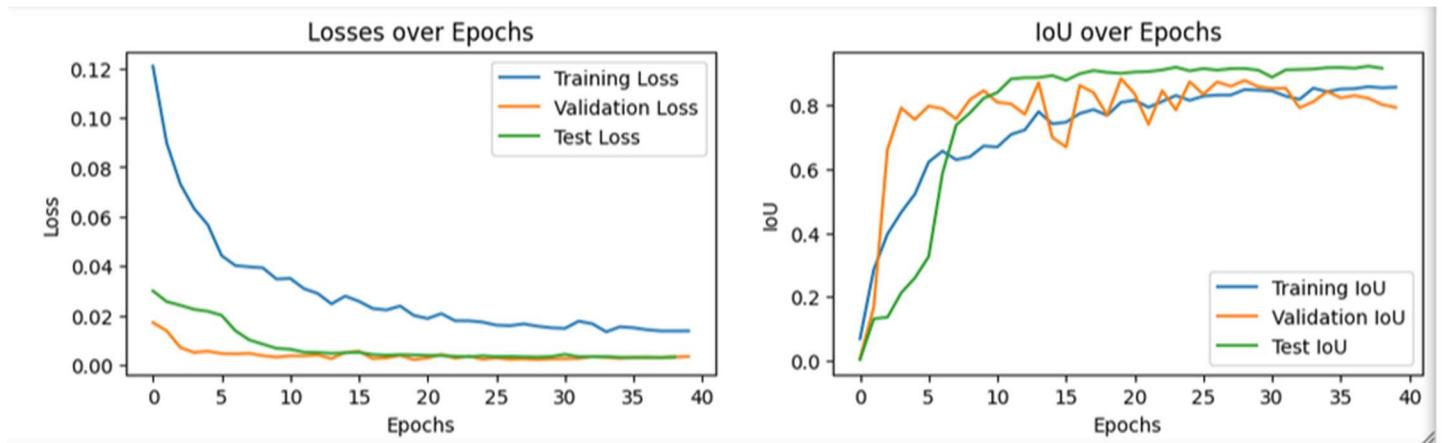
Discard 0s? True | 0s\_masks/Tot\_imgs: 4/31 || Test Loss: 0.014835 | Test IoU: 0.595



	FP	FN	F1	IoU	FP/FN
Test	22607	1649	81,75	69,14	
Val	65320	1468	84,95	73,84	
Train	35176	7667	95,03	90,52	

**Rebalancing Dataset Characteristics:** Initially, the test and validation sets contained distribution outliers. From the beginning, we ensured that each plant's images were allocated exclusively to a single set (train, validation, or test). This means that no plant images were shared between sets, a choice made to maintain a realistic scenario, even though doing so might have improved model performance.

Ensuring that images from the same plant are not shared across the train, validation, and test sets is critical to avoid data leakage, what could artificially inflate performance metrics by allowing the model to "see" aspects of the test or validation data during training.



	FP	FN	F1	IoU	FP/FN
Test	4269	10188	95,67	91,70	
Val	24836	3456	92,95	86,83	
Train	35176	7667	95,03	90,52	

Best model until now!

New base line:

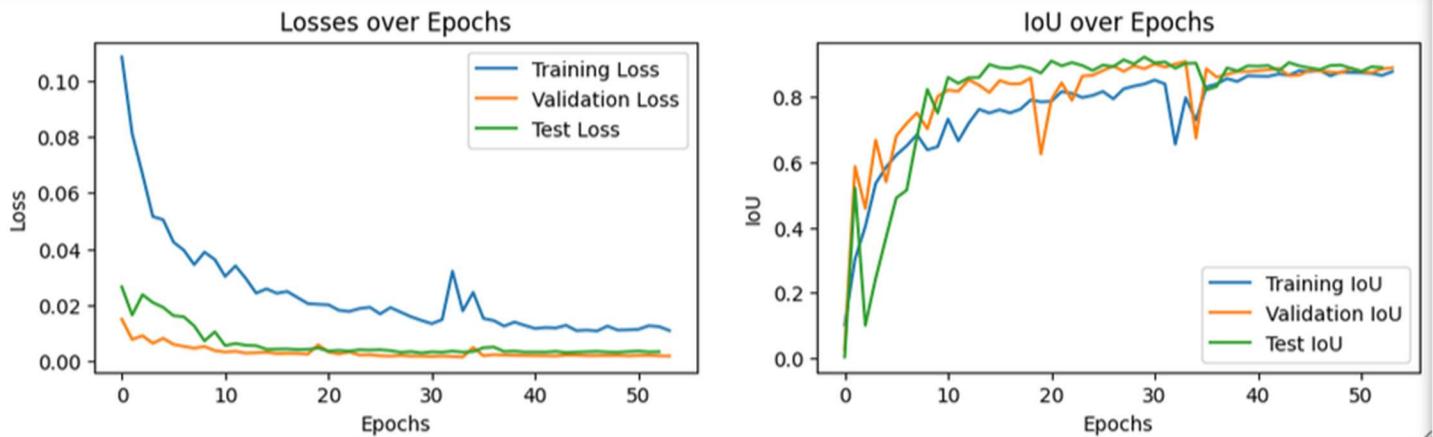
#### Dataset #2

```
def __init__(self, alpha=0.8, gamma=1.5, pos_weight=2.0, neg_weight=1.0):
    jaccard_loss = 0.8 * jaccard_loss_positive + 0.2 * jaccard_loss_negative
    total_loss = 0.4 * bce_loss + 0.2 * focal_loss + 0.4 * jaccard_loss
    lr=0.0001, weight_decay=1e-6
    early_stopping = EarlyStopping(patience=20, min_delta=0.00001)
```

Adjust the loss function

From:  $\text{total\_loss} = 0.4 * \text{bce\_loss} + 0.2 * \text{focal\_loss} + 0.4 * \text{jaccard\_loss}$

To:  $\text{total\_loss} = 0.3 * \text{bce\_loss} + 0.3 * \text{focal\_loss} + 0.4 * \text{jaccard\_loss}$



Before

	FP	FN	F1	IoU	FP/FN
Test	4269	10188	95,67	91,70	0.42
Val	24836	3456	92,95	86,83	7.19
Train	35176	7667	95,03	90,52	4.59

After

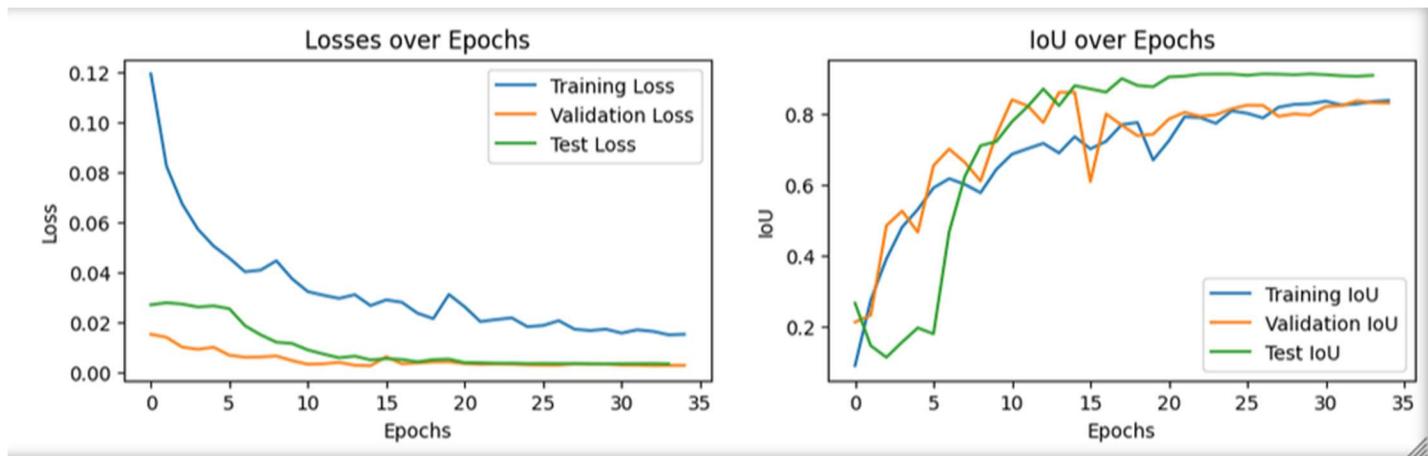
	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

**Best performance till now**

## Adjust the loss function Series

From:  $\text{jaccard\_loss} = 0.8 * \text{jaccard\_loss\_positive} + 0.2 * \text{jaccard\_loss\_negative}$

To:  $\text{jaccard\_loss} = 0.75 * \text{jaccard\_loss\_positive} + 0.25 * \text{jaccard\_loss\_negative}$



## Before Best performance till now

	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

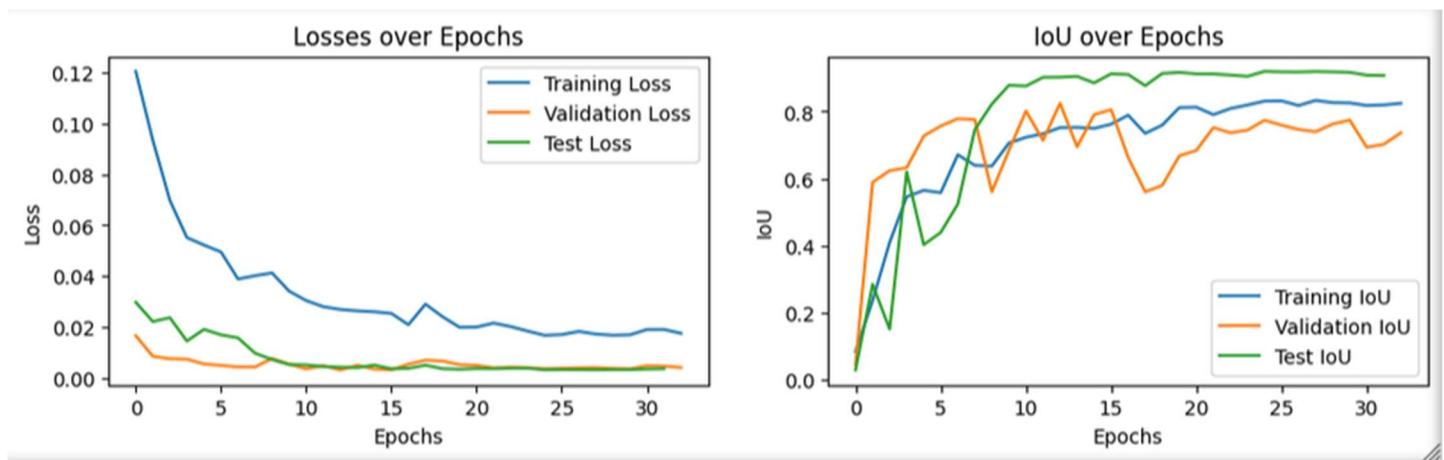
## After

	FP	FN	F1	IoU	FP/FN
Test	7312	10783	94,62	89,79	0.69
Val	32370	3028	91,35	84,08	10.69
Train	38386	11625	92,10	85,35	3.3

## Adjust the loss function Series

From:  $\text{jaccard\_loss} = 0.8 * \text{jaccard\_loss\_positive} + 0.2 * \text{jaccard\_loss\_negative}$

To:  $\text{jaccard\_loss} = 0.85 * \text{jaccard\_loss\_positive} + 0.15 * \text{jaccard\_loss\_negative}$



## Before Best performance till now

	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

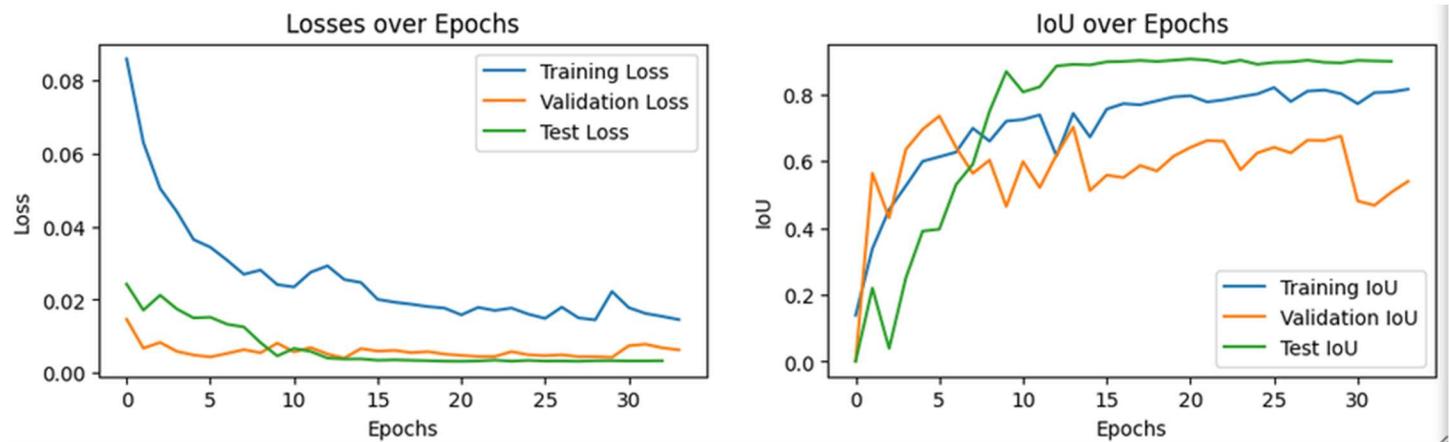
## After

	FP	FN	F1	IoU	FP/FN
Test	7061	7675	95,65	91,67	
Val	44394	5621	88,05	78,66	
Train	35181	13449	92,25	85,62	

## Adjust the loss function Series

From:  $\text{total\_loss} = 0.3 * \text{bce\_loss} + 0.3 * \text{focal\_loss} + 0.4 * \text{jaccard\_loss}$

To:  $\text{total\_loss} = 0.3 * \text{bce\_loss} + 0.4 * \text{focal\_loss} + 0.3 * \text{jaccard\_loss}$



## Before (best performance till now)

	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

## After

	FP	FN	F1	IoU	FP/FN
Test	4871	11313	95,14	90,74	
Val	95435	3701	78,98	65,26	
Train	36191	12788	92,22	85,56	

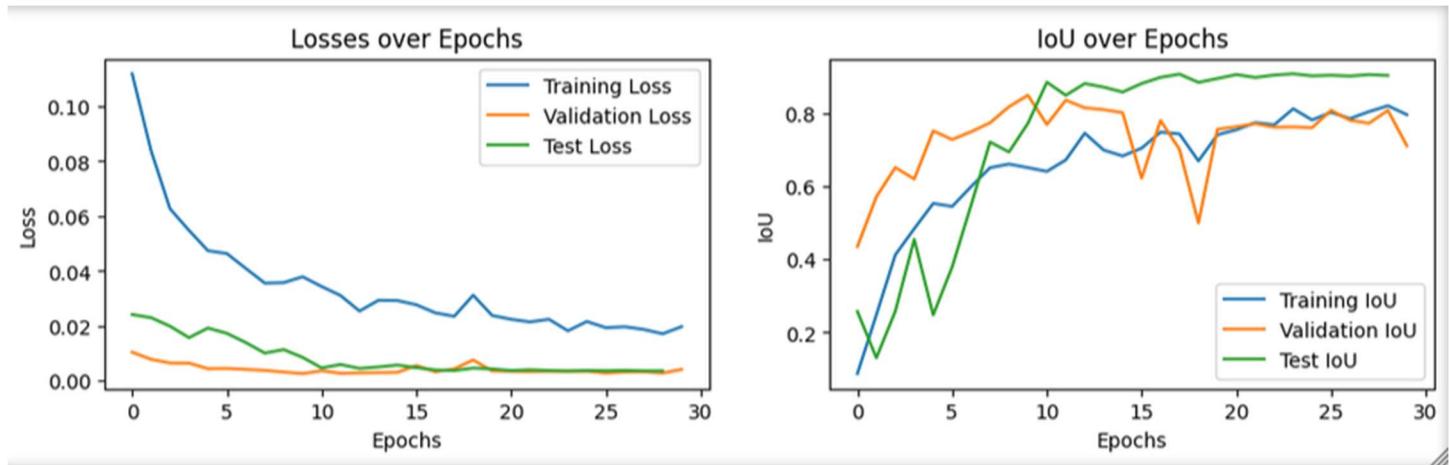
Not a good option.

## Adjust the loss function Series

Keep: `jaccard_loss = 0.8 * jaccard_loss_positive + 0.2 * jaccard_loss_negative`

From: `total_loss = 0.3 * bce_loss + 0.3 * focal_loss + 0.4 * jaccard_loss`

To: `total_loss = 0.2 * bce_loss + 0.4 * focal_loss + 0.4 * jaccard_loss`



## Before (best performance till now)

	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

## After

	FP	FN	F1	IoU	FP/FN
Test	4756	27067	89,97	81,77	
Val	29055	6242	91,24	83,88	
Train	34313	25283	90,31	82,33	

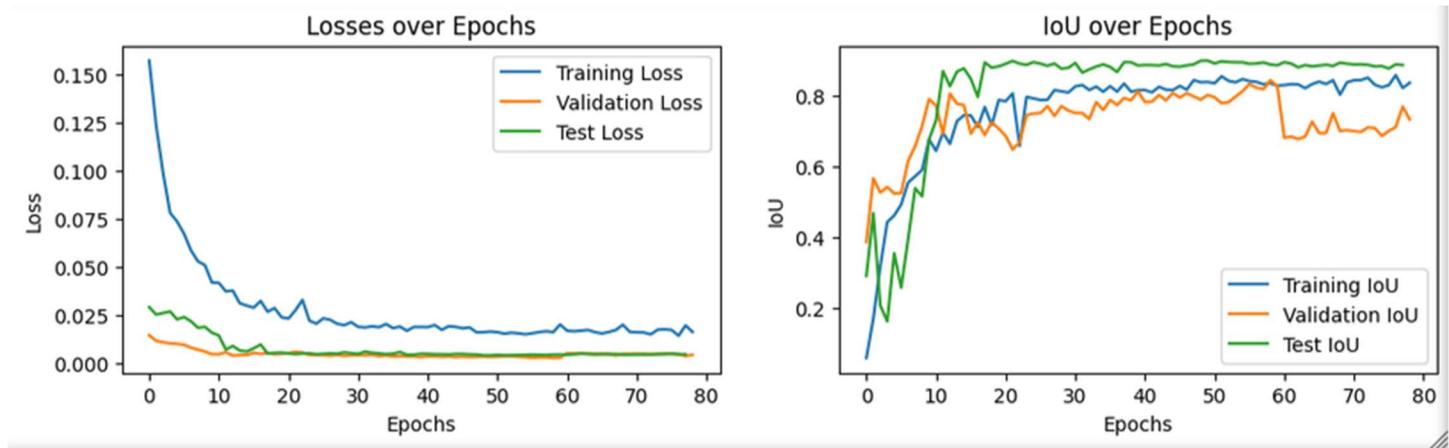
Not a good option.

## Adjust the loss function Series

Keep: `jaccard_loss = 0.8 * jaccard_loss_positive + 0.2 * jaccard_loss_negative`

From: `total_loss = 0.2 * bce_loss + 0.4 * focal_loss + 0.4 * jaccard_loss`

To: `total_loss = 0.4 * bce_loss + 0.2 * focal_loss + 0.4 * jaccard_loss`



## Before (best performance till now)

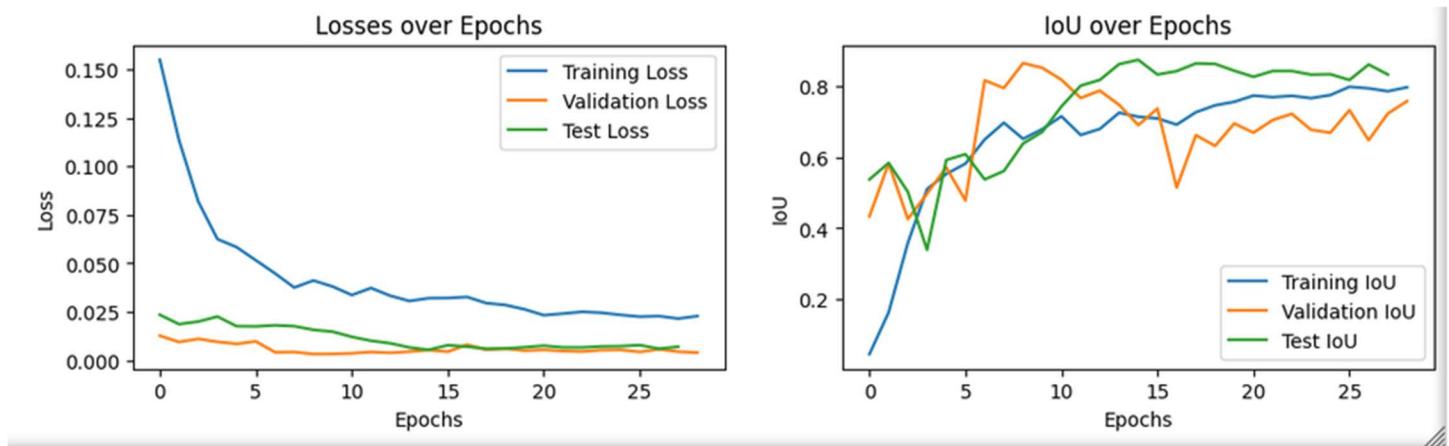
	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

## After

	FP	FN	F1	IoU	FP/FN
Test	2206	13724	95,15	90,74	0.16
Val	44285	8675	87,23	77,36	5.1
Train	27959	6590	94,49	89,56	4.24

Using best hyperparams until now:

total\_loss = 0.3 \* bce\_loss + 0.3 \* focal\_loss + 0.4 \* jaccard\_loss



#### Before (best performance till now)

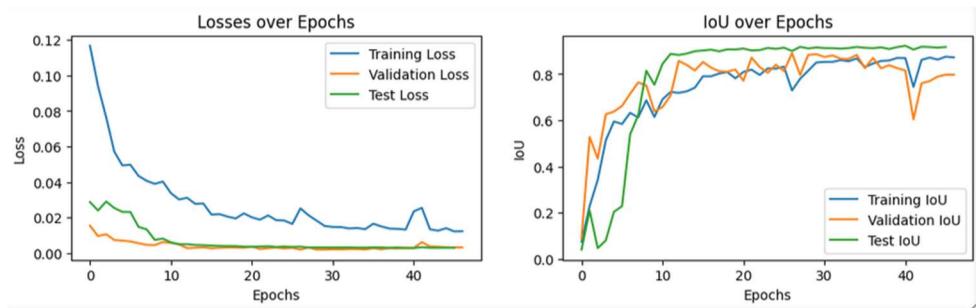
	FP	FN	F1	IoU	FP/FN
Test	3791	11216	95,48	91,36	0.33
Val	14726	4502	95,07	90,61	3.27
Train	30221	7038	94,08	88,82	4.29

#### After

	FP	FN	F1	IoU	FP/FN
Test	2206	13724	95,15	90,74	0.16
Val	44285	8675	87,23	77,36	5.1
Train	27959	6590	94,49	89,56	4.24

I'm noticing more FP than FN. Let's penalize the FP even more:

Keep:  $\text{jaccard\_loss} = 0.85 * \text{jaccard\_loss\_positive} + 0.15 * \text{jaccard\_loss\_negative}$



Before

	FP	FN	F1	IoU	FP/FN
Test	4269	10188	95,67	91,70	0.42
Val	24836	3456	92,95	86,83	7.19
Train	35176	7667	95,03	90,52	4.59

After

	FP	FN	F1	IoU	FP/FN
Test	3362	11778	95,43	91,26	
Val	12555	10302	94,02	88,71	
Train	22004	11778	94,52	89,61	

A 5% change shows a FP reduction, however the FN also increased. Then, no considerable gains were produced.

## Introducing image interpolation:

As a first step, I preferred to use a simpler approach using CV2 instead of creating an ad hoc model.

4-band images: Cubic interpolation

Binary masks: Nearest-neighbor interpolation

Balance:

Before:

Class Balance in Training Set: {'positive\_ratio': 0.023000536866449005, 'negative\_ratio': 0.976999463133551, 'total\_pixels': 13172736}

Class Balance in Validation Set: {'positive\_ratio': 0.08281337193080357, 'negative\_ratio': 0.9171866280691964, 'total\_pixels': 2293760}

Class Balance in Test Set: {'positive\_ratio': 0.07199266221788195, 'negative\_ratio': 0.9280073377821181, 'total\_pixels': 2359296}

After:

Class Balance in Training Set: {'positive\_ratio': 0.023000536866449005, 'negative\_ratio': 0.976999463133551, 'total\_pixels': 52690944}

Class Balance in Validation Set: {'positive\_ratio': 0.08281337193080357, 'negative\_ratio': 0.9171866280691964, 'total\_pixels': 9175040}

Class Balance in Test Set: {'positive\_ratio': 0.07199266221788195, 'negative\_ratio': 0.9280073377821181, 'total\_pixels': 9437184}

Same ratios, total pixels x4. Nice

Stats:

Before:

Mean per band (R, G, B, NIR): [ 927.75739512 740.14410416 492.39665161 2441.67961341]

Std deviation per band (R, G, B, NIR): [420.10468167 257.58737858 200.92512581 500.59008288]

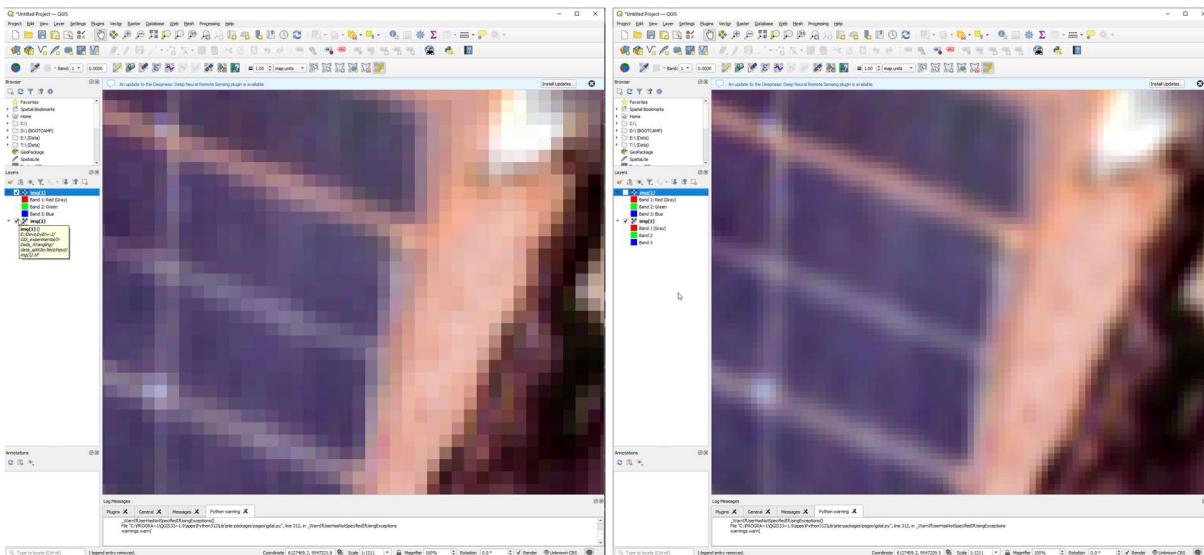
After:

Mean per band (R, G, B, NIR): [ 927.72309094 740.12511994 492.37997292 2441.66539495]

Std deviation per band (R, G, B, NIR): [419.47567552 257.08428302 200.48893916 499.48728449]

The Mean and STD were slightly changed.

New resolution:

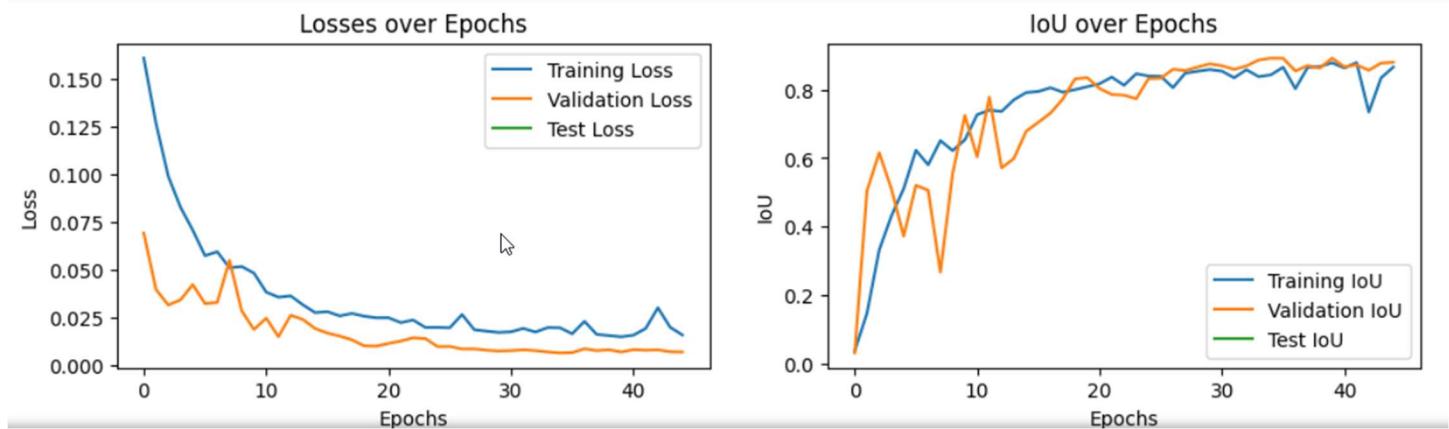


Epoch: 44

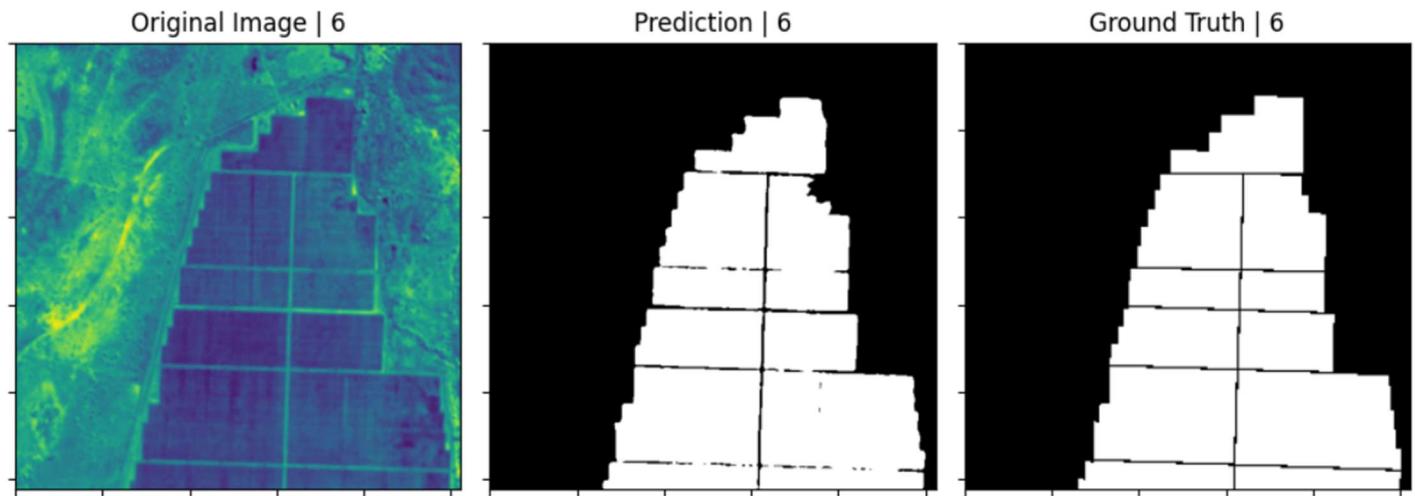
Training completed in 197 minutes and 10.33 seconds.

One band: NIR

early\_stopping = EarlyStopping(patience=10, min\_delta=0.0001)



ID: 6 | IoU: 0.956



Input images: Cubic interpolation

Masks: Nearest-neighbor interpolation

scale\_factor=2

batch size reduction:

train: batch\_size=4,

valid: batch\_size=10,

test: loader batch\_size=10,

---

TP:649744

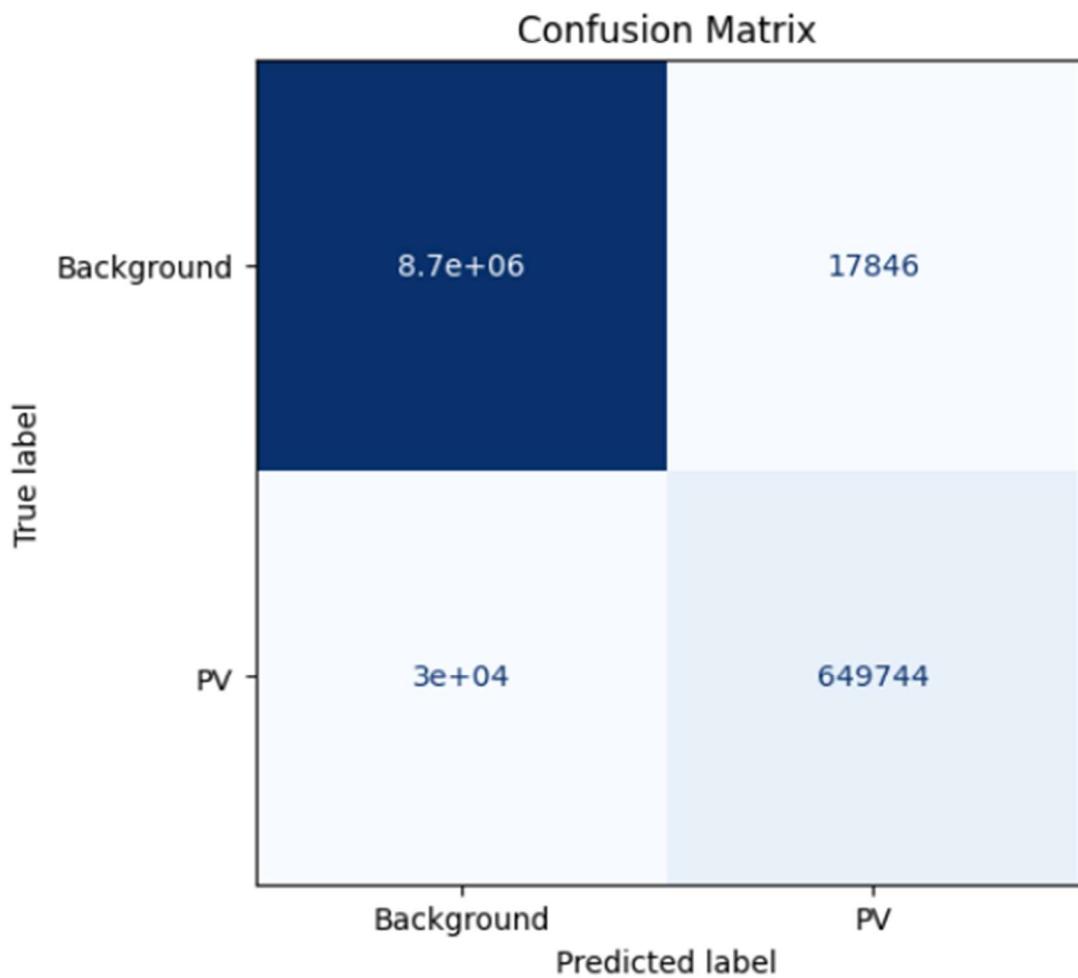
TN:8739930

FP:17846

FN:29664

---

Accuracy:	0.9950	$(TP + TN) / (TP + TN + FP + FN)$
Recall:	0.9563	$TP / (TP + FN)$
Specificity:	0.9980	$TN / (TN + FP)$
Precision:	0.9733	$TP / (TP + FP)$
F1-Score:	0.9647	$(2 * precision * recall) / (precision + recall)$
IoU:	0.9319	$TP / (TP + FN + FP)$



Next Step: Apply Super Resolution and see the results.