CALTECH101

Pere Joan Vives Morey 41620797C

INTRODUCCIÓ

El dataset **Caltech101** és un replec d'imatges ofert per la universitat de Caltech que pertanyen a 101 categories diferents. De cada categoria disposem d'entre 40 i 800 imatges amb una mitja de 50 imatges per categoria. L'objectiu de la pràctica és utilitzar 2 de les categories per entrenar models de classificació, segmentació i detecció, i comparar els resultats de cada model.

En el meu cas, m'han assignat les classes de '**cougar_body'** i '**windsor_chair'**. Respectivament:

Per fer l'entrenament dels models de classificació, s'utilitzarà la tècnica '**fine tunning'** que consisteix en utilitzar un model entrenat (amb els pesos ja definits) damunt un altre data set i en actualitzar els pesos d'aquest amb les imatges de les nostres classes.

Això té la avantatge de que ja no hem d'entrenar un model de 0. Això accelera el procés d'entrenament del model per a que classifiqui les nostres imatges.

Després de cada entrenament, també realitzarem la validació i el testing del model. Tots els resultats, llevat del model **YOLO**, es guardaran a l'eina de **Weights and Biases.**

Aquesta eina ens permetrà visualitzar l'evolució de les mètriques que utilitzem per avaluar els models. S'utilitzarà '**l'accuracy'**, '**l'f1-score'**, el '**recall'**, el '**precision**', i la **pèrdua**.

Per al model de YOLO, els resultats es guardaran al mateix directori on feim les proves, ja que per aquest model utilitzarem la llibreria '**ultralytics'.** D'aquesta llibreria utilitzarem el model YOLOv5.

Tot el procés de recaptació i tractament de dades, entrenament, validació i testing es durà a terme a l'eina de **Google Collab**. El problema que tenen molts dels models que utilitzam és la gran profunditat i quantitat de paràmetres que tenen. L'entrenament d'un d'aquests podria suposar molta càrrega per un ordenador d'un estudiant. Google Collab ofereix un entorn d'execució extern al nostre PC, es pot triar el dispositiu damunt el qual fer els entrenaments. Nosaltres hem realitzat totes les proves damunt la GPU T4.

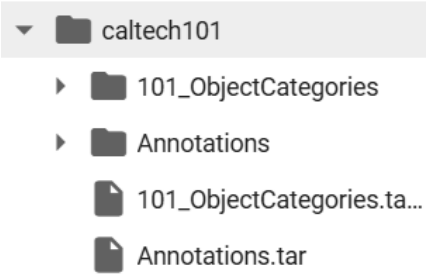
METODOLOGIA

Com es presenten les dades?

La primera passa és descarregar el data set i veure la seua distribució. Hi ha diverses llibreries que ofereixen aquest data set, com **'TensorFlow'** i **'Pythorch'**. Nosaltres ens hem decantat per la de Pytorch.

```
dataset = torchvision.datasets.Caltech101(root= ROOT,download=DOWNLOAD,transform=transform)
```

Un cop descarregat, veim la seva organització:



Dins la carpeta de 101_ObjectCategories hi trobarem 101 carpetes que corresponen a cada classe. Dins cada carpeta hi trobarem les imatges d'una classe en format **JPG**. Segons la pàgina web oficial de Caltech, les imatges, en general, tenen les dimensions 300 x 200 píxels.

Dins la carpeta de **Annotations** hi trobarem també 101 carpetes per a cada classe. A cada carpeta hi trobarem un fitxer per a cada imatge en format '.mat'. El que ens proporcionen aquests fitxers es informació addicional del contingut de les imatges. Cada un d'aquests fitxer conté els camps:

- '__header__'
- '__version__'
- '__globals__'
- 'box_coord'
- 'obj_contour'

D'aquests camps, els que més ens interessen són els de **'obj_contour'**, que conté les coordenades x,y de tots els punts a la imatge que en conjunt formen el contorn de l'objecte de la imatge.

obj_contour és un array de la forma:

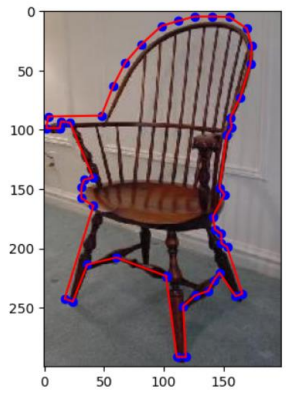
[[coordenades_x][coordenades_y]]

De manera que un punt es forma de manera:

(obj_contour[0][i],obj_contour[1][i]) on 'i' correspon al nombre del punt.

Així es visualitza el contorn de la imatge. A la pràctica final no hem deixat aquest exemple, ja que no tenia cap funció damunt l'entrenament i avaluació dels models.

```
training.__showcontours__(1)
```



Per obtenir la informació dels fitxers '.mat' s'ha utilitzat la llibreria '**scipy**' i per visualitzar aquesta imatge s'ha utilitzat la llibreria de '**matplotlib**'.

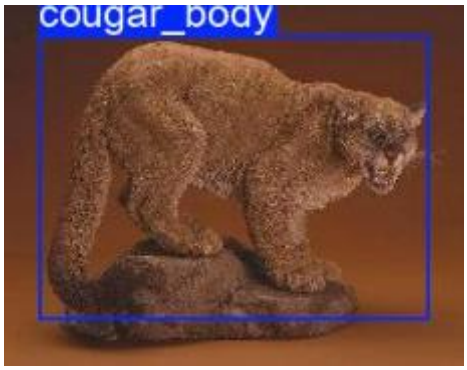
Aquests contorns ens seran molt útils a l'hora de crear les màsqueres que necessiten els nostres models de segmentació per entrenar-se. Més envant veurem com les cream.

D'altra banda, l'altre camp dels fitxers '.mat' que ens interessa és el de **'box_coord'**. Que conté les coordenades x,y de dos punts que formen la caixa dins la qual hi trobarem el nostre objecte. Les coordenades es presenten de la següent manera:

[[2 299 2 183]]

[[y_min y_max x_min x_max]]

Mirem com queda un bounding box amb la seva imatge:



Aquesta informació ens serà útil a l'hora d'entrenar models de detecció com YOLO. Ara que ja sabem de quina informació disposem i com està distribuïda, anem a veure com tractem aquestes dades.

Creació conjunts training, validation, testing

En primer lloc, guardarem el 'path' de totes les imatges de les dues classes. Per fer això utilitzarem la llibreria 'glob'.

```
img_class_1 = sorted(glob('/content/sample_data/caltech101/101_ObjectCategories/cougar_body/*'))
img_class_2 = sorted(glob('/content/sample_data/caltech101/101_ObjectCategories/windsor_chair/*'))
```

A partir d'aquests paths, també podem obtenir les 'Labels' ja que com veim, les carpetes tenen el nom de les classes. Extreurem el nom de la carpeta de cada imatge:

/cougar_body/imatge.jpg

I la codificarem utilitzant la classe 'LabelEncoder' que pertany a la llibreria 'sklearn'.

També extreurem els paths de la carpeta 'Annotations':

```
img_annotations_class_1 = sorted(glob('/content/sample_data/caltech101/Annotations/cougar_body/*'))
img_annotations_class_2 = sorted(glob('/content/sample_data/caltech101/Annotations/windsor_chair/*'))
```

On, com hem explicat, hi trobarem els fitxers '.mat' amb la informació addicional de cada imatge.

Ara que ja tenim els paths, labels i les annotations, les separarem en tres conjunts utilitzant la funció 'train_test_split' de la llibreria 'sklearn'.

```
X_train, X_test, y_train, y_test, annotations_train, annotations_test = train_test_split(
    img_files, labels, img_annotations, test_size=TESTING, random_state=42, stratify=labels
)

# Split the training set further into training and validation sets
X_train, X_val, y_train, y_val, annotations_train, annotations_val = train_test_split(
    X_train, y_train, annotations_train, test_size=VAL/(TRAINING+VAL), random_state=42, stratify=y_train
)
```

Utilitzam 'stratify = labels' perquè volem que es mantengui la distribució de classes del conjunt sencer d'imatges.

La distribució d'imatges en training, validation, testing ha estat:

1) Training: 80%
2) Validation: 10%
3) Testing: 10%

Un cop tenim els conjunts d'imatges i labels separats, les hem de guardar en una classe que es pugui gestionar a l'hora de fer els entrenaments. Aquesta classe heretarà la classe 'Dataset' de la llibreria pytorch.

```
class Formes(Dataset):
    def __init__(self, paths, labels= None, transforms = None,annotations = None,transform_mask = None):
        self.images = paths
        self.labels = labels
        self.transforms = transforms
        self.annotations = annotations
        self.transforms mask = transform mask
```

Com veiem, al mètode d'inicialització hi passem els paths de les imatges, les labels, la transformació que farem a les imatges, les annotations (informació dels fitxers .mat), i les transformacions que farem a les màsqueres de les imatges (que crearem nosaltres mateixos).

Aquesta classe conté els mètodes:

```
1) __len__ : longitud del set
2) __setmodel__ :guardar quin model d'entrenament utilitzem
3) __getdist__ :obtenir la distribució de classes del set
4) __showcontours__ :mostrar el contorn d'una imatge
5) __getpureimage__ : obtenir una imatge fora transformacions
```

6) __getitem__ :obtenir una imatge havent aplicat les transformacions

El mètode __getitem__ ens retorna una imatge transformada i la seva classe. En el cas de que utilitzem el model 'unet' (if self.model == 3) :

```
if self.model == 3:
  mask = np.zeros(image.size, dtype=np.uint8)
  cv2.fillPoly(mask, [contour], color=255)  # Pintam de blanc el polígon que repersenta la mask
  mask = Image.fromarray(mask)  #convertim el numpy a PIL image

  torch.random.manual_seed(seed)  # ja que cada transformació es random, ens volem assegurad de
  mask_resized = self.transforms_mask(mask)
  mask_resized = (mask_resized > 0.5).float()

  torch.random.manual_seed(seed)  # Set seed for image
  image = self.transforms(image)

  return image, mask_resized

image = self.transforms(image)

return image, label
```

En vers de retornar la label, retornarem la mask, que no és més que una imatge de fons negre on la silueta de l'objecte és blanca.

Transformació de les imatges:

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])

transform_masks = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomRotation(degrees=20),
    transforms.ToTensor(),
])

transform2 = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomRotation(degrees=20),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])
```

Respectivament: transformació dels conjunts validation i testing, transformació de les màsqueres, transformació del conjunt d'entrenament.

Per defecte, la distribució d'imatges en els conjunts es aquesta:

```
LONGITUD SET DE TRAINING: 81
LONGITUD SET DE VALIDATION: 11
LONGITUD SET DE TESTING: 11
```

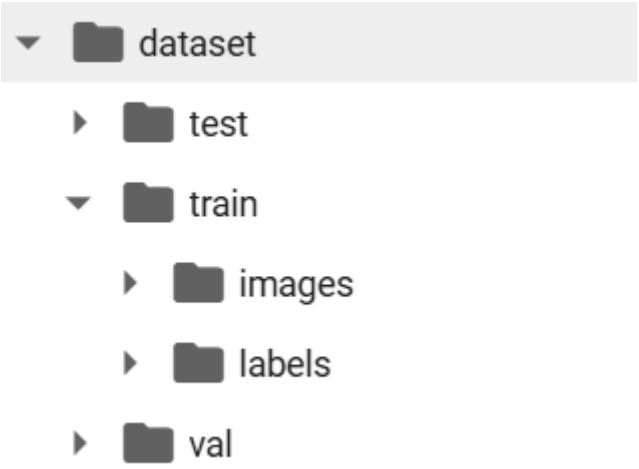
S'ofereix la possibilitat de augmentar la quantitat d'aquestes imatges amb la variable global 'DUPLICAR_DADES'. Si posam aquesta variable a True, llavors duplicarem el nombre d'imatges. El que s'ha fet per aconseguir això es posar dues vegades el path d'una imatge al conjunt de paths antes de dividir-lo en els conjunts de training, validation i testing.

En un primer cop es pot pensar que això por dur a l'overfitting, es veurà en les proves que farem si això és el cas.

Ara que ja tenim tota el necessari, cream els conjunts:

```
training = Formes(X_train,y_train,transform2,annotations_train,transform_masks)
testing = Formes(X_test,y_test,transform,annotations_test,transform_masks)
validation = Formes(X_val,y_val,transform,annotations_val,transform_masks)
```

Aquesta manera de guardar les dades ens serveix per a tots els models excepte per el model YOLOv5 de la llibreria 'ultralytics'. Aquesta llibreria té uns requisits específics.



Dins /traint/images hi guardarem les imatges en format jpg del conjunt de training.

Dins labels hi guardarem, per a cada imatge, un fitxer '.txt' amb la informació:

[class_id,x_center,y_center,width,height]

On x_center i y_center representen el centre de la bounding box de la imatge. Widht i height representen la amplada i altura (amb valors normalitzats entre 0 i 1). Exemple 'imatge_0001.txt':

0 0.40666666666667 0.52430555555556 0.77333333333333 0.84027777777778

Aquesta informació li servirà al model per a saber on es troba la bounding box.

Tots aquests fitxers i carpetes ho he hagut de crear jo. Primer he creat una funció que escriu la informació de la bounding box i la label als fitxer de text corresponents a cada imatge:

```
def escriu_informacio_yolo(carpeta,nom_imatge,annotations,class_id,image_size):
```

Carpeta: path de la carpeta

Nom_imatge: nom de la imatge (imatge_0001)

Annotations: La informació de la bounding box

Class_id: La classe (0: cougar_body, 1:Windsor_chair)

A aquesta funció calculam el centre de la bounding box i escrivim tota la informació normalitzada entre 0 i 1 al fitxer de text.

A continuació creem totes les carpetes necessàries i hi fiquem les imatges i els fitxers de text corresponents:

```
directoris = ["dataset/train/","dataset/test/","dataset/val/",

for carpeta in directoris:
  if not os.path.exists(carpeta):
    os.mkdir(carpeta)

for elemento in X_train:
  shutil.copy(elemento, "/content/dataset/train/images")

for elemento in X_val:
  shutil.copy(elemento, "/content/dataset/val/images")

for elemento in X_test:
  shutil.copy(elemento, "/content/dataset/test/images")
```

➔ Cream les carpetes i hi afegim les imatges.

```
for i in range(len(X_train)):
  mat_data = scipy.io.loadmat(annotations_train[i])
  nom = X_train[i].split("/")[-1]
  label = y_train[i]
  boundingbox_data = mat_data['box_coord']
  width,height = Image.open(X_train[i]).size

  escriu_informacio_yolo("dataset/train/labels/",nom,boundingbox_data,label,(width,height))

for i in range(len(X_val)):
  mat_data = scipy.io.loadmat(annotations_val[i])
  nom = X_val[i].split("/")[-1]
  label = y_val[i]
  boundingbox_data = mat_data['box_coord']
  width,height = Image.open(X_val[i]).size

  escriu_informacio_yolo("dataset/val/labels/",nom,boundingbox_data,label,(width,height))

for i in range(len(X_test)):
  mat_data = scipy.io.loadmat(annotations_test[i])
  nom = X_test[i].split("/")[-1]
  label = y_test[i]
  boundingbox_data = mat_data['box_coord']
  width,height = Image.open(X_test[i]).size

  escriu_informacio_yolo("dataset/test/labels/",nom,boundingbox_data,label,(width,height))
```

➔ Cream els fitxers de text i els ficam a les carpetes ‘labels’

Creació dels Models

Ja tenim totes les dades a punt per entrenar tots els models. Falta definir quins seran aquests models. Aquests es declararan a la funció:

```
def pick_algorithm(number):
```

On hi passarem un nombre depenent de quin model vulguem provar.

```
architectures = {'alexnet': 0,'vgg': 1,'resnet': 2,'unet':3,'propi':4,'yolo':5,'rcnn':6}
```

Aquesta és la codificació dels models.

Segons la categoria:

- Classificació: alexnet, vgg11, resnet, propi
- Detecció: YOLO, rcnn
- Segmentació: unet

Ja que hem d’utilitzar la tècnica ‘fine tunning’ en els models de classificació. S’ha utilitzat els pesos entrenats en el data set ‘IMAGENET’.

```
alexnet = models.alexnet(weights=models.AlexNet_Weights.IMAGENET1K_V1)
```

La funció de pèrdua d'aquests models serà la de binary cross entropy de la llibreria 'pytorch':

```
loss_fn = nn.BCEWithLogitsLoss()
```

Empleam aquesta perquè la sortida serà d'un valor:

```
alexnet.classifier[6] = nn.Linear(in_features=4096, out_features=1)
```

Aquest valor és un lògit, el qual, si després el passem a la funció sigmoide ens dirà la probabilitat de que la imatge passada sigui d'una classe o una altra. Això ja ho veurem a la part d'entrenament.

A la resta de models de classificació hem utilitzat la mateixa funció de pèrdua.

El model de 'unet' és el mateix que varem utilitzar a classe.

```
class UNet(nn.Module):

    def __init__(self, in_channels=3, out_channels=1, init_features=32):
        super(UNet, self).__init__()
```

La funció de pèrdua és 'DiceLoss', he utilitzat la mateixa que varem emplear a classe.

```
class DiceLoss(nn.Module):

    def __init__(self):
        super(DiceLoss, self).__init__()
        self.smooth = 0.0

    def forward(self, y_pred, y_true):
        assert y_pred.size() == y_true.size()
        y_pred = y_pred[:, 0].contiguous().view(-1)
        y_true = y_true[:, 0].contiguous().view(-1)
        intersection = (y_pred * y_true).sum()
        dsc = (2. * intersection + self.smooth) / (
                y_pred.sum() + y_true.sum() + self.smooth
        )
        return 1. - dsc
```

No s'ha provat a utilitzar 'unet' amb la funció de pèrdua BCE.

En quant al model YOLO, hem utilitzat el de la llibreria 'ultralytics', la versió 5.

```
model = YOLO('yolov5n.pt')
return model,None
```

La funció de pèrdua es None ja que el mateix model de la llibreria ja la duu incorporada.

Per acabar, el model propi:

```
model = nn.Sequential(

    nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=0),

    nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=0),

    nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=0),

    nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=0),

    nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=0),

    nn.AdaptiveAvgPool2d(output_size=(6, 6)),

    nn.Flatten(),
    nn.Dropout(p=0.1),
    nn.Linear(9216, 4096),
    nn.ReLU(inplace=True),
    nn.Linear(4096, 1024),
    nn.ReLU(inplace=True),
    nn.Linear(1024, 1024),
    nn.ReLU(inplace=True),
    nn.Linear(1024, 1)
)

return model, nn.BCEWithLogitsLoss()
```

L'he fet semblant als models vgg11 i alexnet, aquest però, entrenarà els pesos desde 0.

També he decidit afegir un poc de Dropout per evitar l'overfitting.

Funcions d'Entrenament

Per a realitzar l'entrenament, la validació i el testing he utilitzat les funcions:

```
1) fit(model,loss_fn,dataloader,optimizer,epoch)
2) validate(model,data_loader,loss_fn):
3) test(model,data_loader,loss_fn):
```

model: model a provar

los_fn: funció de pèrdua

data_loader:

```
train_loader = torch.utils.data.DataLoader(training, batch_size=BATCH_SIZE, shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation, batch_size=validation.__len__(), shuffle=True)
testing_loader = torch.utils.data.DataLoader(testing, batch_size=testing.__len__(), shuffle=True)
```

Optimizer:

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

La funció principal on utilitzarem les anteriors:

```python
def prova():
  t_loss = np.zeros(EPOCHS)
  v_loss = np.zeros(EPOCHS)
  acc_t = np.zeros(EPOCHS) #accuracy
  acc_v = np.zeros(EPOCHS)
  f1_t = np.zeros(EPOCHS) #f1
  f1_v = np.zeros(EPOCHS)
  recall_t = np.zeros(EPOCHS) #recall
  recall_v = np.zeros(EPOCHS)
  precision_t = np.zeros(EPOCHS)
  precision_v = np.zeros(EPOCHS) #precisió

  epochs_without_improvement = 0
  best_val_loss = float('inf')
  best_val_acc = 0
  pbar = tqdm(range(1, EPOCHS + 1))  # tdqm permet tenir text dinàmic

  for epoch in pbar:


    train_acc,train_f1,train_recall,train_precision,train_loss = fit(model,loss_fn,train_loader,optimizer,epoch)

    val_acc,val_f1,val_recall,val_precision,val_loss = validate(model,validation_loader,loss_fn)

    test_acc,test_f1,test_recall,test_precision,test_loss = test(model,testing_loader,loss_fn)
```

De totes aquestes funcions hi utilitzam aquests hiperparàmetres que podem anar canviant a cada execució:

1) Epochs
2) Batch Size
3) Learning Rate
4) Distribució dels conjunts training,validation,testing
5) Duplicar el nombre d'imatges (DUPLICAR_DADES)

Per poder fer les proves correctament, ens interessa guardar quins d'aquests paràmetres hem utilitzat damunt cada model. També ens interessa saber el resultats de les mètriques:

1) F1-score
2) Accuracy
3) Precision
4) Recall
5) Loss

Per guardar tota aquesta informació, hem utilitzat la eina 'Weights and Biases'. Aquesta eina ens proporciona una llibreria anomenada 'wandb'. Amb aquesta llibreria:

```python
if MODEL != architectures['yolo']:
  wandb.init(
          project=projectes[MODEL],
          config={
              "epochs": EPOCHS,
              "batch_size": BATCH_SIZE,
              "lr": learning_rate,
              "trsize":training.__len__(),
              "trdist":TRAINING,
              "vsize":validation.__len__(),
              "vdist":VAL,
              "duplicated":DUPLICAR_DADES
              })


  config = wandb.config
```

Podem inicialitzar un objecte on guardarem les configuracions inicials. I mentre la funció de prova() s'està executant, hi podem anar guardant el resultat de les mètriques:

```
training_metrics = {"train/train_loss": train_loss/len(train_loader),
                "train/train_acc":train_acc/len(train_loader),
                "train/train_f1":train_f1/len(train_loader),
                "train/train_recall":train_recall/len(train_loader),
                "train/train_precision":train_precision/len(train_loader)}

val_metrics = {"val/val_loss": val_loss/len(validation_loader),
               "val/val_acc":val_acc/len(validation_loader),
               "val/val_f1":val_f1/len(validation_loader),
               "val/val_recall": val_recall/len(validation_loader),
               "val/val_precision": val_precision/len(validation_loader)}

testing_metrics = {"test/test_loss": test_loss/len(testing_loader),
                   "test/test_acc": test_acc/len(testing_loader),
                   "test/test_f1":test_f1/len(testing_loader),
                   "test/test_recall": test_recall/len(testing_loader),
                    "test/test_precision": test_precision/len(testing_loader)}
```

```
#Saving the results

wandb.log({**training_metrics, **val_metrics,**testing_metrics})

torch.save(model, "my_model.pt")
wandb.log_model("./my_model.pt", architecrures_inv[MODEL], aliases=[f"epoch-{epoch+1}"])
```

Ja que per al model YOLO no utilitzaré aquestes funcions, les execucions es duran a terme aquí:

```
if MODEL == architectures['unet']:
  prova()
  visualitza_resultats() #visualitza les màsqueres previstes
elif MODEL != architectures['yolo']:
  prova()
else: #yolo
  train_results = model.train(
      data="/content/data.yaml",
      epochs=100,
      imgsz=288,
      verbose=True
  )

  metrics = model.val()

  results = model(X_test[0])
  results[0].show()


  path = model.export(format="onnx")
```

Ara que ja tenim tot el necessari per realitzar les proves, passarem a realitzar execucions per a cada model.

EXPERIMENTS REALITZATS

Començarem avaluant els models de classificació. Tots els resultats es visualitzaran a l'eina 'Weights and Biases'. Durant aquest projecte s'han realitzat una infinitat d'experiments, donat que no els puc ensenyar tots, he decidit ensenyar els més que han contribuït a obtenir el millor model.

Alexnet

Config parameters: {} 8 keys

batch_size: 8

duplicated: true

epochs: 15

lr: 0.00015

trdist: 0.8

trsize: 164

vdist: 0.1

vsize: 21

Summary metrics: {} 15 keys

test/test_acc: 1

test/test_f1: 1

test/test_loss: 0.0000000454130635319

test/test_precision: 1

test/test_recall: 1

train/train_acc: 1

train/train_f1: 1

train/train_loss: 0.00000000573974859579

train/train_precision: 1

train/train_recall: 1

val/val_acc: 1

val/val_f1: 1

val/val_loss: 0.00000000000003039976

val/val_precision: 1

val/val_recall: 1

Veim que les mètriques ens diuen que aquest model prediu molt bé les nostres classes, ja que la majoria arriben al 100%.

Hem de tenir en compte però, que en aquest exemple s'han duplicat el nombre d'imatges. En el següent exemple posarem 'duplicated = false' a veure si les mètriques empitjoren molt.

Config parameters: {} 8 keys

batch_size: 8

duplicated: false

epochs: 15

lr: 0.00015

trdist: 0.8

trsize: 81

vdist: 0.1

vsize: 11

test/test_acc: 1

test/test_f1: 1

test/test_loss: 0.00000008670411943967

test/test_precision: 1

test/test_recall: 1

train/train_acc: 1

train/train_f1: 1

train/train_loss: 0.00000021104667548898

train/train_precision: 1

train/train_recall: 1

val/val_acc: 1

val/val_f1: 1

val/val_loss: 0.00000123746633562405

val/val_precision: 1

val/val_recall: 1

Veim que amb el conjunt d'imatges original el model no empitjora. S'ha decidit no provar més hiperparàmetres ja que aquests ens han dut a molt bon resultat. Dificilment es poden millorar les mètriques.

Vgg16

A continuació provarem el model vgg16.

```
vgg = models.vgg16(weights = models.VGG16_Weights.IMAGENET1K_V1)
```

Recordam que amb tots els models de classificació utilitzam la tècnica de 'fine tunning' amb els pesos entrenats amb el data set 'IMAGENETV1'.

En primer lloc, provarem amb els mateixos paràmetres que ens han donat bona solució a Alexnet.

Com son igual Alexnet. nombre model.

Config parameters: {} 8 keys

batch_size: 8

duplicated: false

epochs: 15

lr: 0.00015

trdist: 0.8

trsize: 81

vdist: 0.1

vsize: 11

podem veure les mètriques de bones que amb el model No fa falta duplicar el d'imatges per millorar el

test/test_acc: 1

test/test_f1: 1

test/test_loss: 0

test/test_precision: 1

test/test_recall: 1

train/train_acc: 1

train/train_f1: 1

train/train_loss: 0

train/train_precision: 1

train/train_recall: 1

val/val_acc: 1

val/val_f1: 1

val/val_loss: 0

val/val_precision: 1

val/val_recall: 1

Resnet

Seguidament provarem el model Resnet, preentrenat també amb el data set 'IMAGENETV1'.

Emplearem les mètriques que ens han funcionat en els models anteriors.

Config parameters: {} 8 keys

batch_size: 8

duplicated: false

epochs: 15

lr: 0.00015

trdist: 0.8

trsize: 81

vdist: 0.1

vsize: 11

Summary metrics: {} 15 keys

test/test_acc: 1

test/test_f1: 1

test/test_loss: 0.00025995730538852513

test/test_precision: 1

test/test_recall: 1

train/train_acc: 0.9090909090909092

train/train_f1: 0.9090909090909092

train/train_loss: 0.15201399927503767

train/train_precision: 0.9090909090909092

train/train_recall: 1

val/val_acc: 1

val/val_f1: 1

val/val_loss: 0.000814201426692307

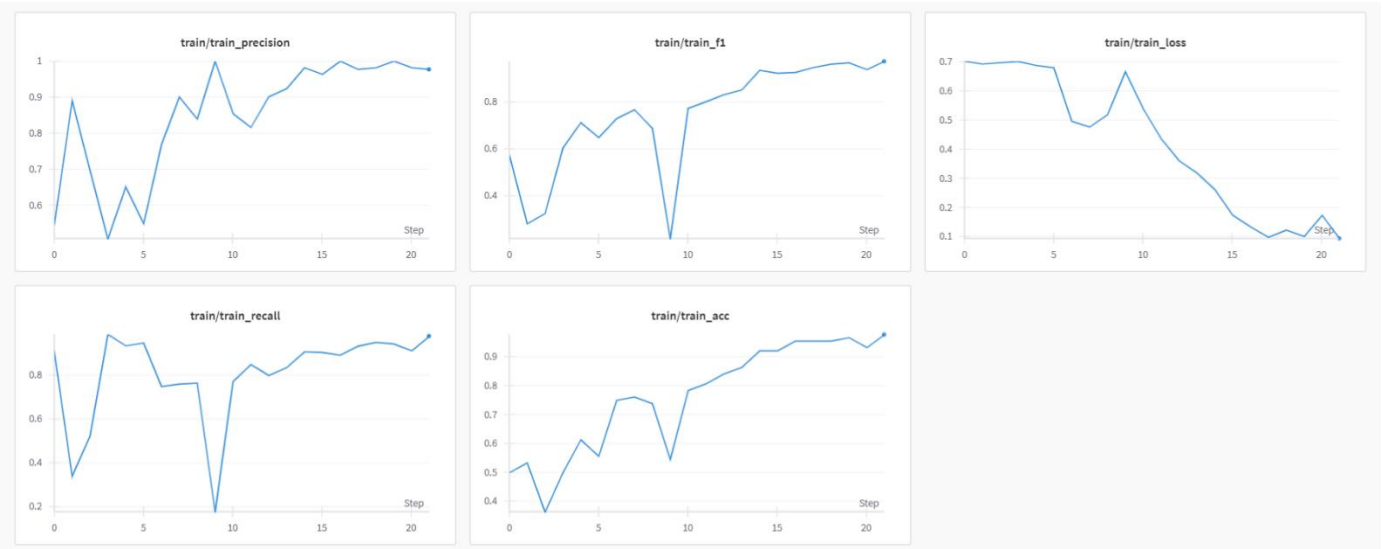
val/val_precision: 1

val/val_recall: 1

Veim que tampoc fa falta duplicar el nombre d'imatges per obtenir un bon model.

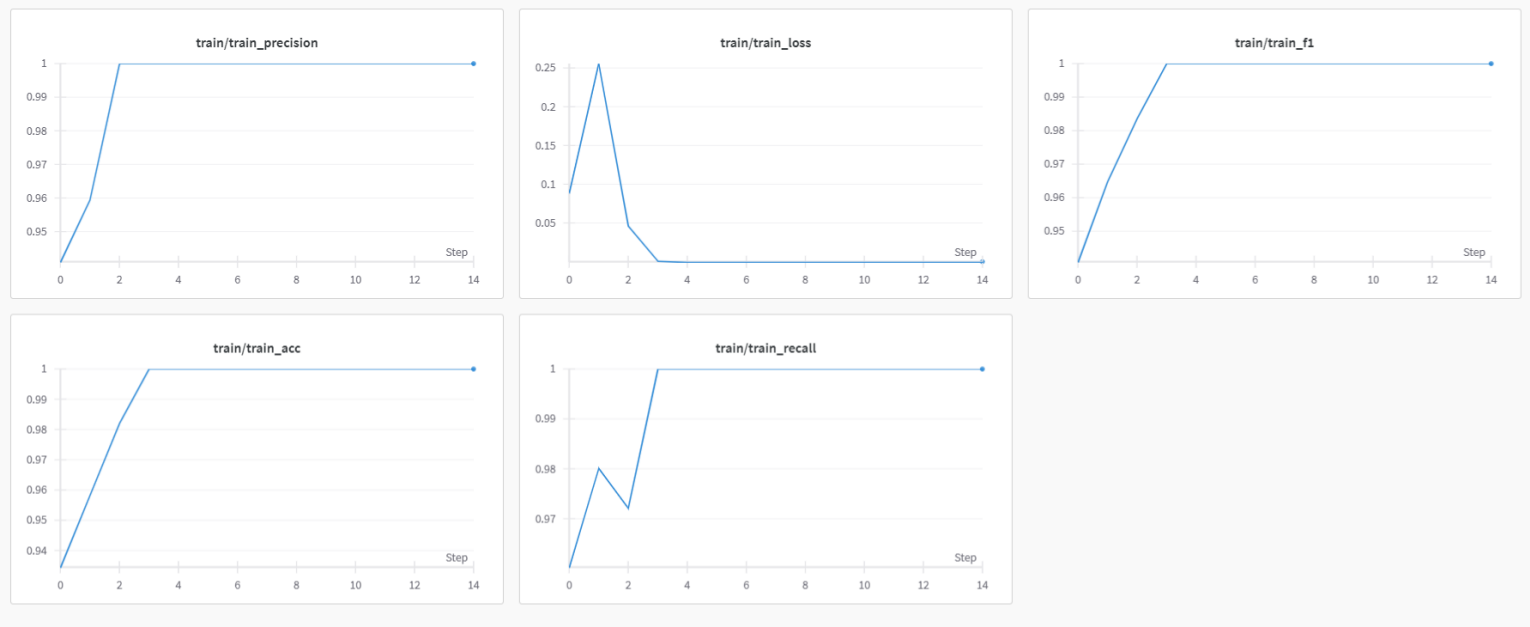
Pròpia

A continuació provarem a entrenar la nostra arquitectura. En aquest cas s'entrenen els pesos desde 0, per tant, augmentaré el nombre de EPOCHS a 20. Si veig que a l'Epoch 20 el model encara millorava les mètriques, llavors augmentaré més el valor fins arribar a la



convergència.

Aquest model, en comparació a la resta, ha necessitat de més EPOCHS per arribar a les millors mètriques. Als anteriors models, als primers EPOCHS ja s'arribava a mètriques molt bones. Per exemple, en l'alexnet:



Resultats del model propi:

Config parameters: {} 8 keys

batch_size: 8

duplicated: false

epochs: 15

lr: 0.00015

trdist: 0.8

trsize: 81

vdist: 0.1

vsize: 11

Summary metrics: {} 15 keys

test/test_acc: 1

test/test_f1: 1

test/test_loss: 0.011818435043096542

test/test_precision: 1

test/test_recall: 1

train/train_acc: 0.9772727272727272

train/train_f1: 0.974025974025974

train/train_loss: 0.09343267370290546

train/train_precision: 0.977272727272727

train/train_recall: 0.9772727272727272

val/val_acc: 0.9090909090909092

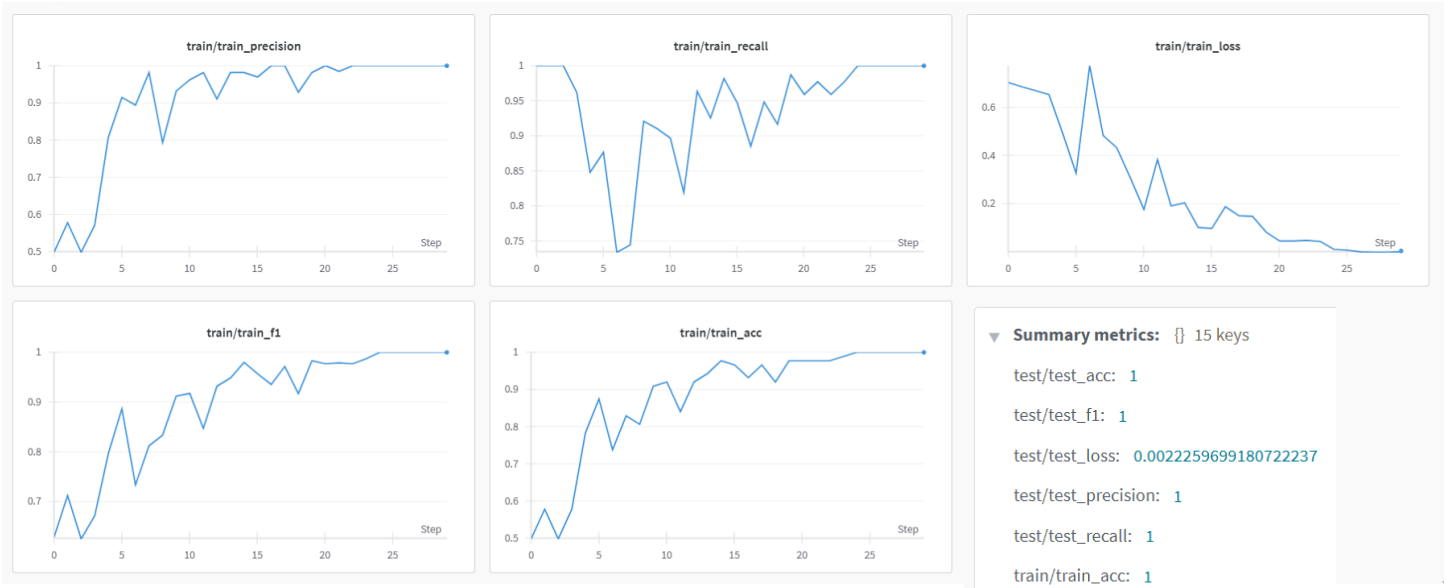
val/val_f1: 0.9090909090909092

val/val_loss: 0.348413348197937

val/val_precision: 1

val/val_recall: 0.8333333333333334

Provarem a pujar els EPOCHS a 30 a veure si podem pujar les mètriques tant com als anteriors models:



Summary metrics: {} 15 keys

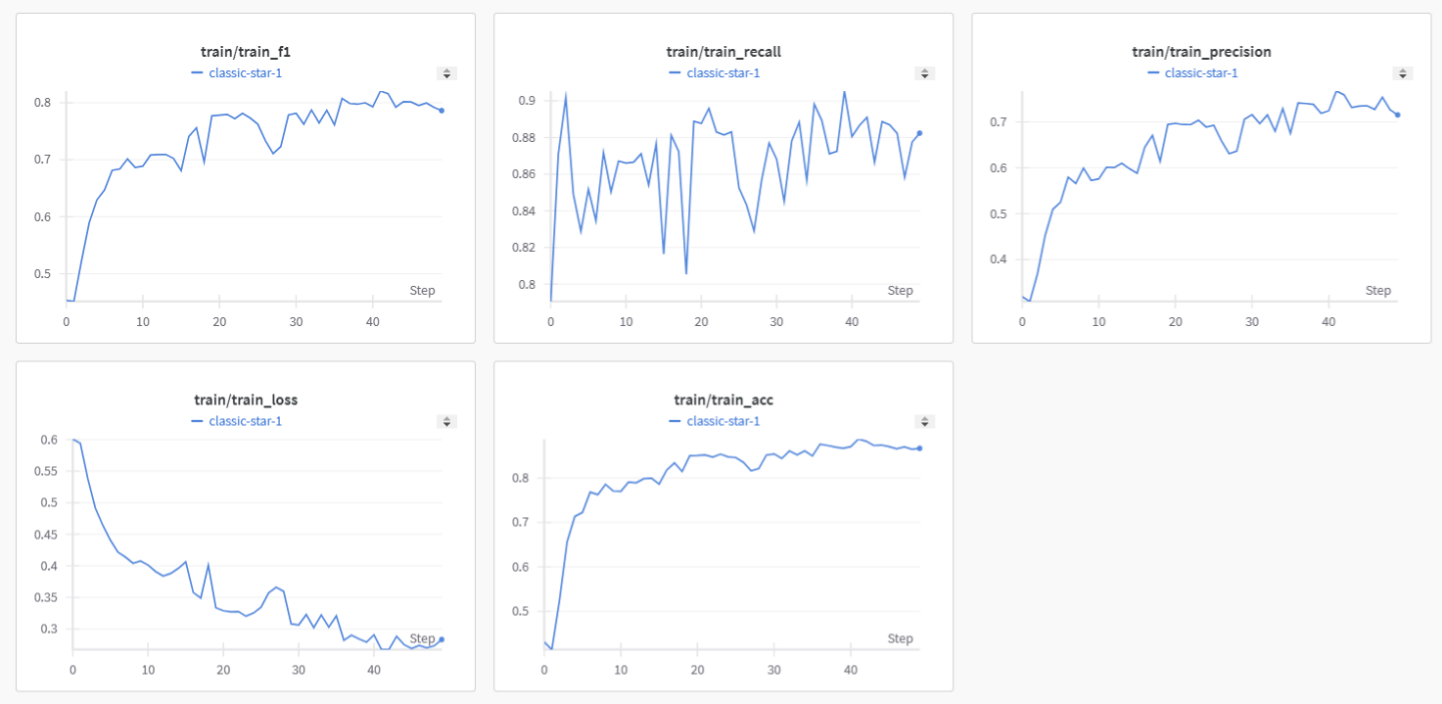
test/test_acc: 1
test/test_f1: 1
test/test_loss: 0.0022259699180722237
test/test_precision: 1
test/test_recall: 1
train/train_acc: 1
train/train_f1: 1
train/train_loss: 0.002994579232108663
train/train_precision: 1
train/train_recall: 1
val/val_acc: 0.9090909090909092
val/val_f1: 0.9230769230769232
val/val_loss: 0.44283005595207214
val/val_precision: 0.8571428571428571
val/val_recall: 1

Veiem que sí hem pogut arribar a uns resultats semblants amb els models pre-entrenats.

UNET

A continuació realitzarem proves amb el model de Segmentació. A part de visualitzar les mètriques i paràmetres utilitzats, també mostrarem les màsqueres que ha predit el model. Això ho veim amb la funció:

```
visualitza_resultats():
```



Config parameters: {} 8 keys

batch_size: 8

duplicated: false

epochs: 50

lr: 0.00015

trdist: 0.8

trsize: 81

vdist: 0.1

vsize: 11

Summary metrics: {} 15 keys

test/test_acc: 0.8077077777133581

test/test_f1: 0.6519145307554812

test/test_loss: 0.38641440868377686

test/test_precision: 0.7140055318078954

test/test_recall: 0.5997586144469254

train/train_acc: 0.8668351040700369

train/train_f1: 0.7862315854190378

train/train_loss: 0.2832899039441889

train/train_precision: 0.7152891161779492

train/train_recall: 0.882382004234174

val/val_acc: 0.7876565398886828

val/val_f1: 0.6593874787117174

val/val_loss: 0.3687202334403992

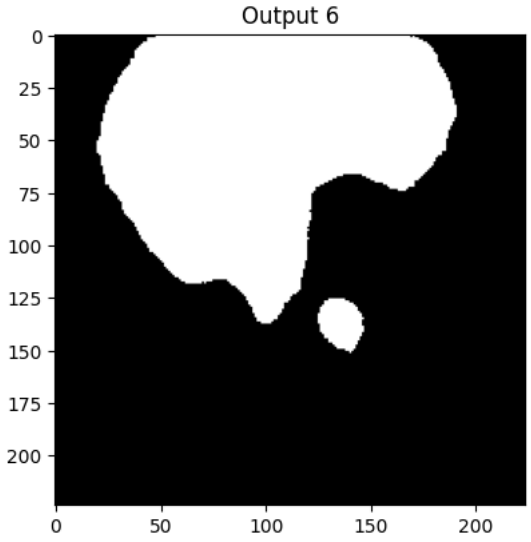
val/val_precision: 0.6954061741411863

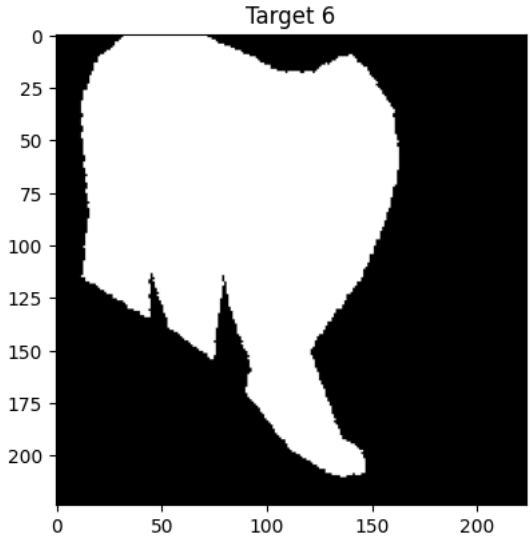
val/val_recall: 0.6269162328547586

Com veim, després de 50 epochs, les mètriques ja no milloren més. Els resultats son bons, però vull provar si amb les imatges duplicades podem millorar el model.

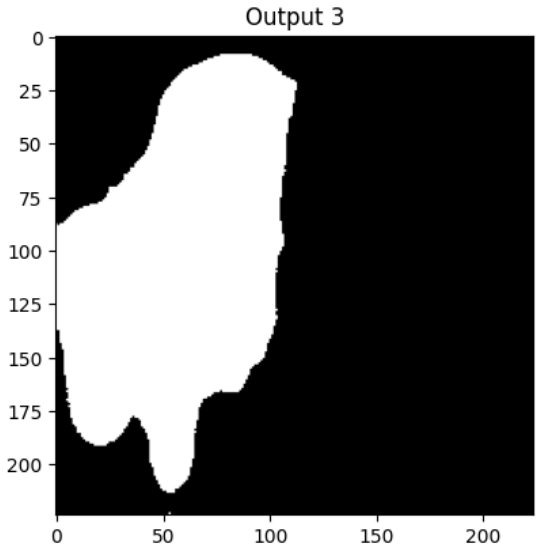
El resultat de les màsqueres:

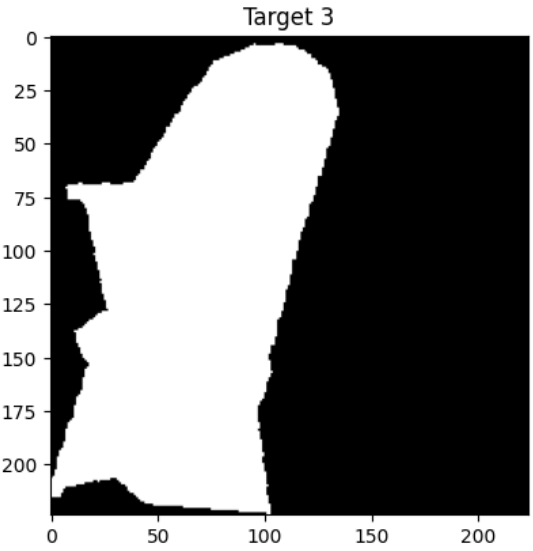
Cougar_body:





Windsor_chair:

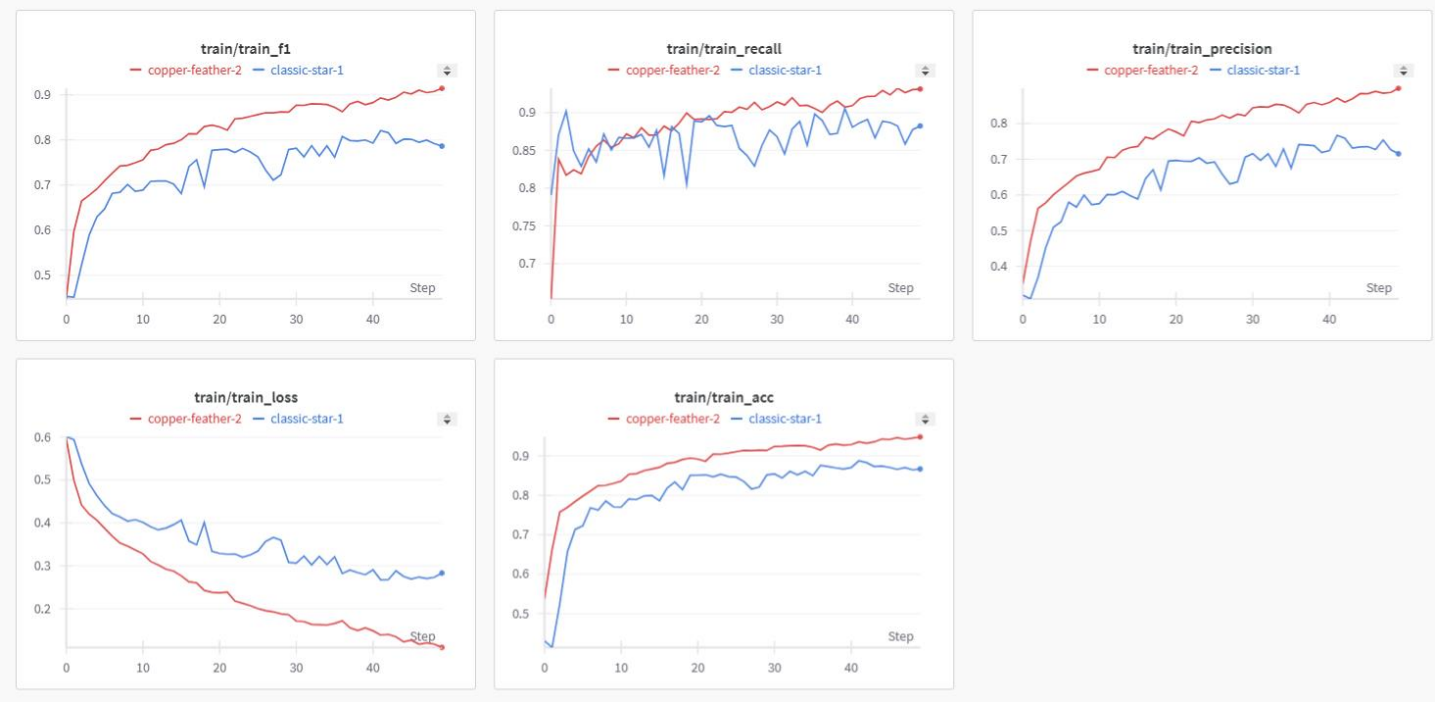




Veient les màsqueres, queda clar que el model ha predit millor les màsqueres de la classe ‘windsor_chair’ que la classe ‘cougar_body’. Passem a la següent prova amb les imatges duplicades.

En vermell: imatges duplicades

En blau: data set original



Com podem veure, el model millora significativament després de duplicar les imatges.

Config parameters: {} 8 keys

batch_size: 8

duplicated: true

epochs: 50

lr: 0.00015

trdist: 0.8

trsize: 164

vdist: 0.1

vsize: 21

Summary metrics: {} 15 keys

test/test_acc: 0.8794804193999028

test/test_f1: 0.8076351534559203

test/test_loss: 0.20944231748580933

test/test_precision: 0.8088444559066705

test/test_recall: 0.8064294616603948

train/train_acc: 0.9485630817617224

train/train_f1: 0.9146060839756917

train/train_loss: 0.10979000727335612

train/train_precision: 0.8989828254519248

train/train_recall: 0.9312840482631926

val/val_acc: 0.8873783330296404

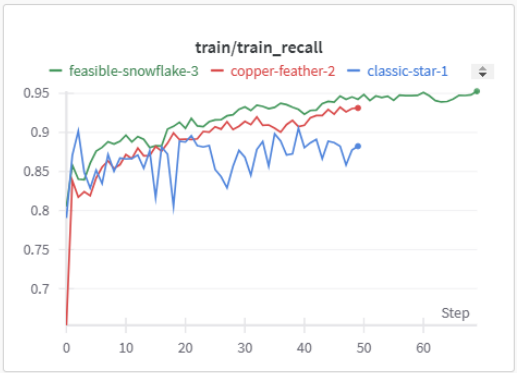
val/val_f1: 0.8025418356259661

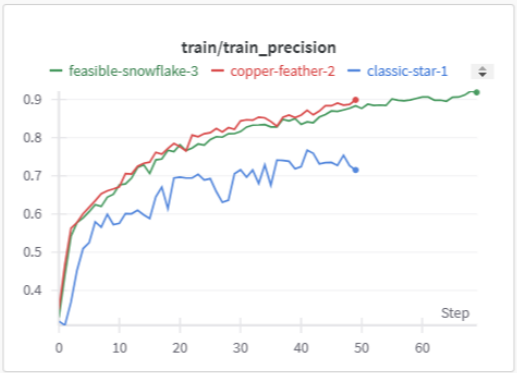
val/val_loss: 0.2167009115219116

val/val_precision: 0.8105191339477169

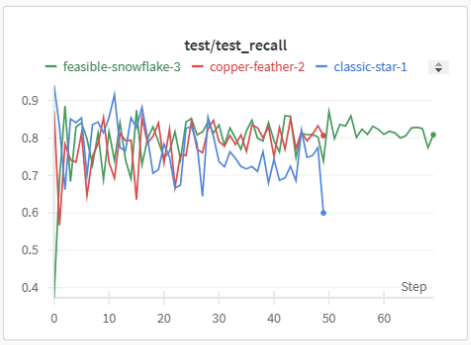
val/val_recall: 0.7947200353271884

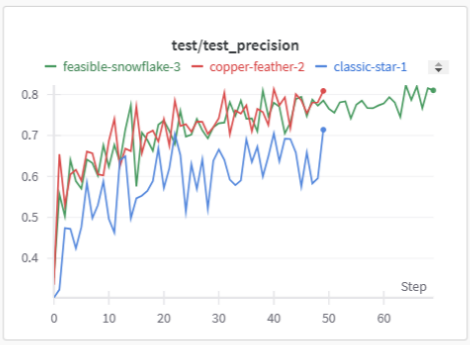
Veient la resta de gràfics, em dona la sensació de que el model podria millorar més si augmentam el nombre de EPOCH. Pujarem de 50 a 70 epochs.

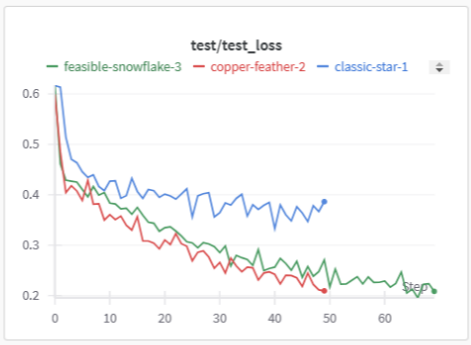


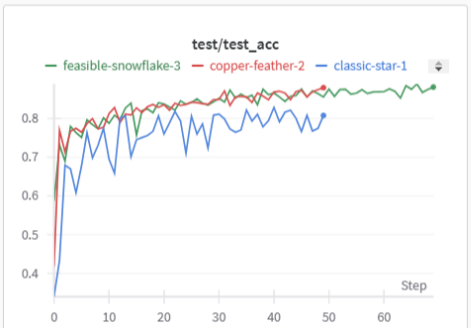


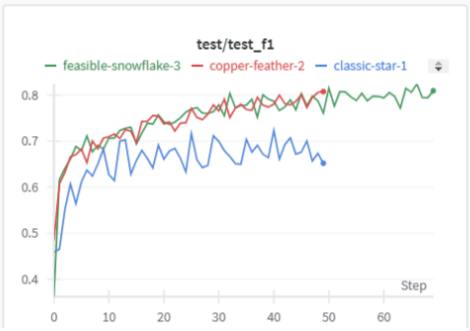
Milloram les mètriques del conjunt de training.











Però les mètriques de test i validation no milloren massa.

Config parameters: {} 8 keys

batch_size: 8

duplicated: true

epochs: 70

lr: 0.00015

trdist: 0.8

trsize: 164

vdist: 0.1

vsize: 21

Summary metrics: {} 15 keys

test/test_acc: 0.8810121704931972

test/test_f1: 0.8093818368684235

test/test_loss: 0.2083990573883057

test/test_precision: 0.8102034480239608

test/test_recall: 0.8085618903837449

train/train_acc: 0.961049012238824

train/train_f1: 0.935163147872351

train/train_loss: 0.09172479595456803

train/train_precision: 0.9187684795407642

train/train_recall: 0.9528502441535018

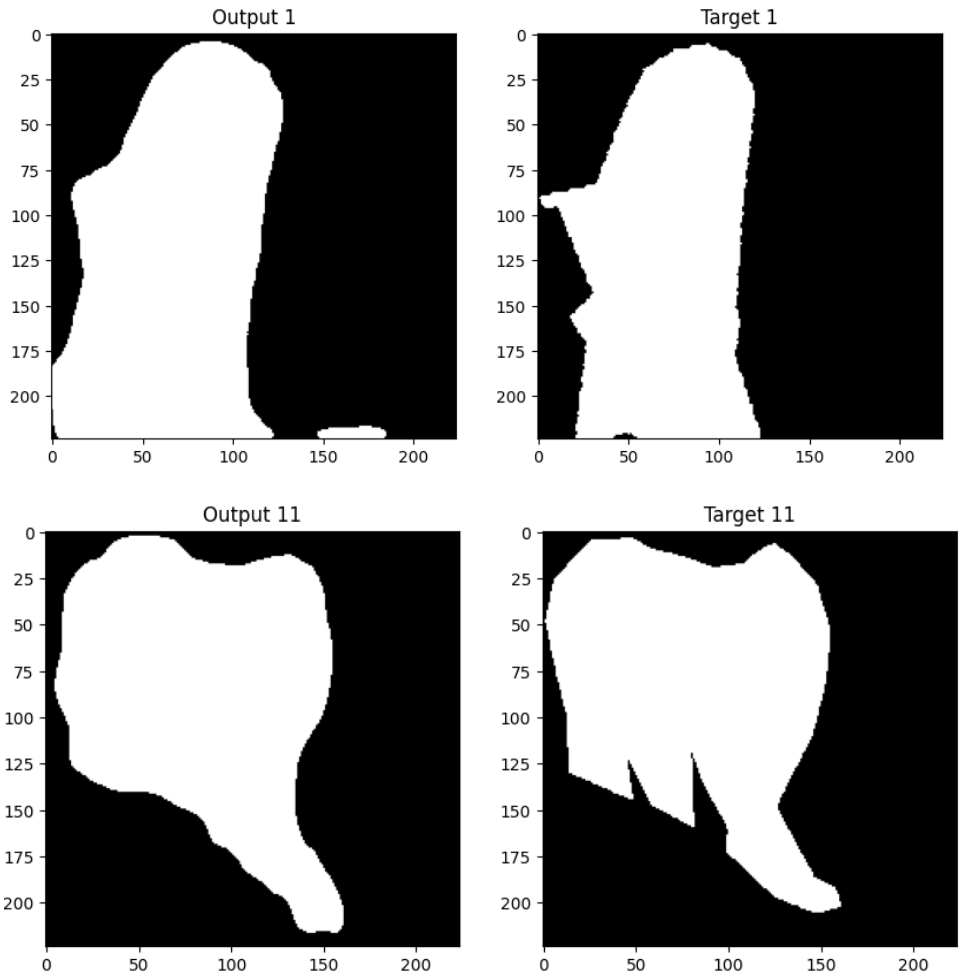
val/val_acc: 0.8829330281219631

val/val_f1: 0.7962868462469633

val/val_loss: 0.22378039360046387

val/val_precision: 0.7894125043386008

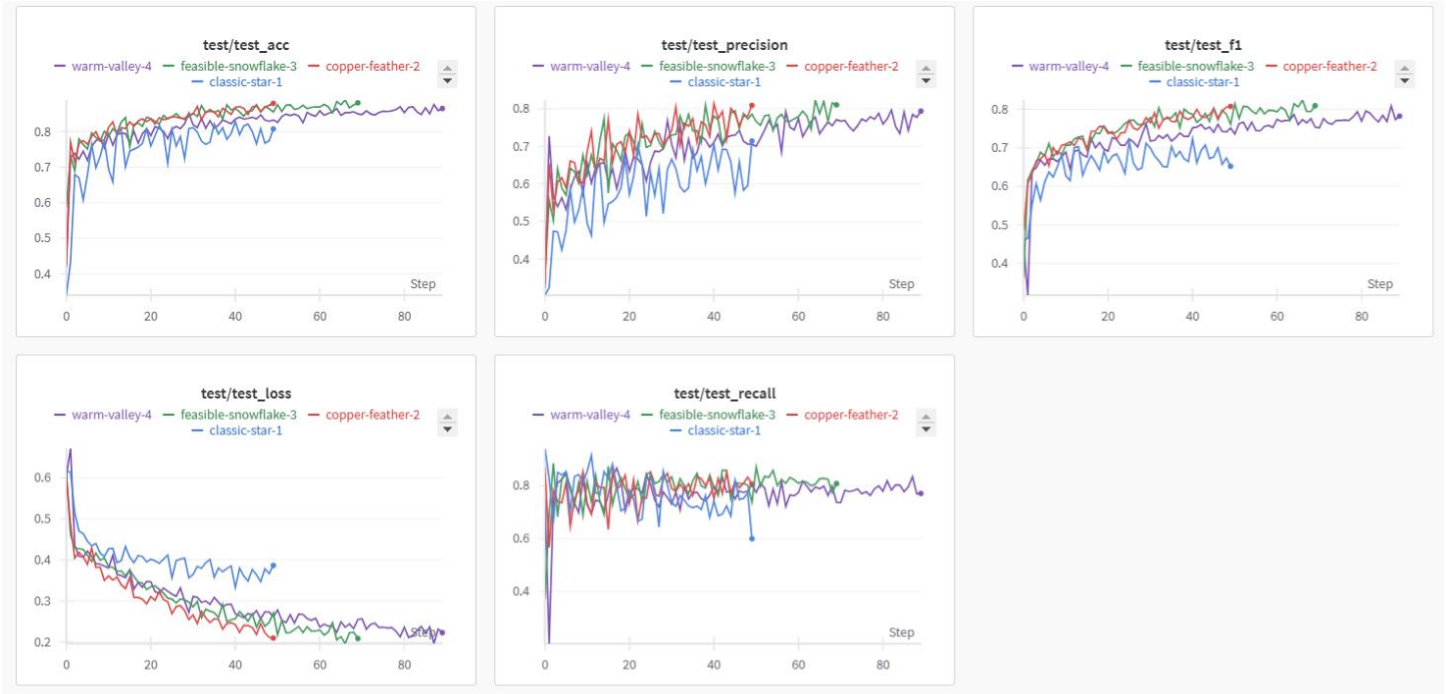
val/val_recall: 0.8032819658475635



Les màsqueres han millorat respecte a l'anterior execució. Donat que tenim un cas de overfitting, ja que al conjunt de training l'accuracy arriba al 96% i als de validation i testing no arriben al 90%, afegirem un poc de Dropout al nostre model de UNET:

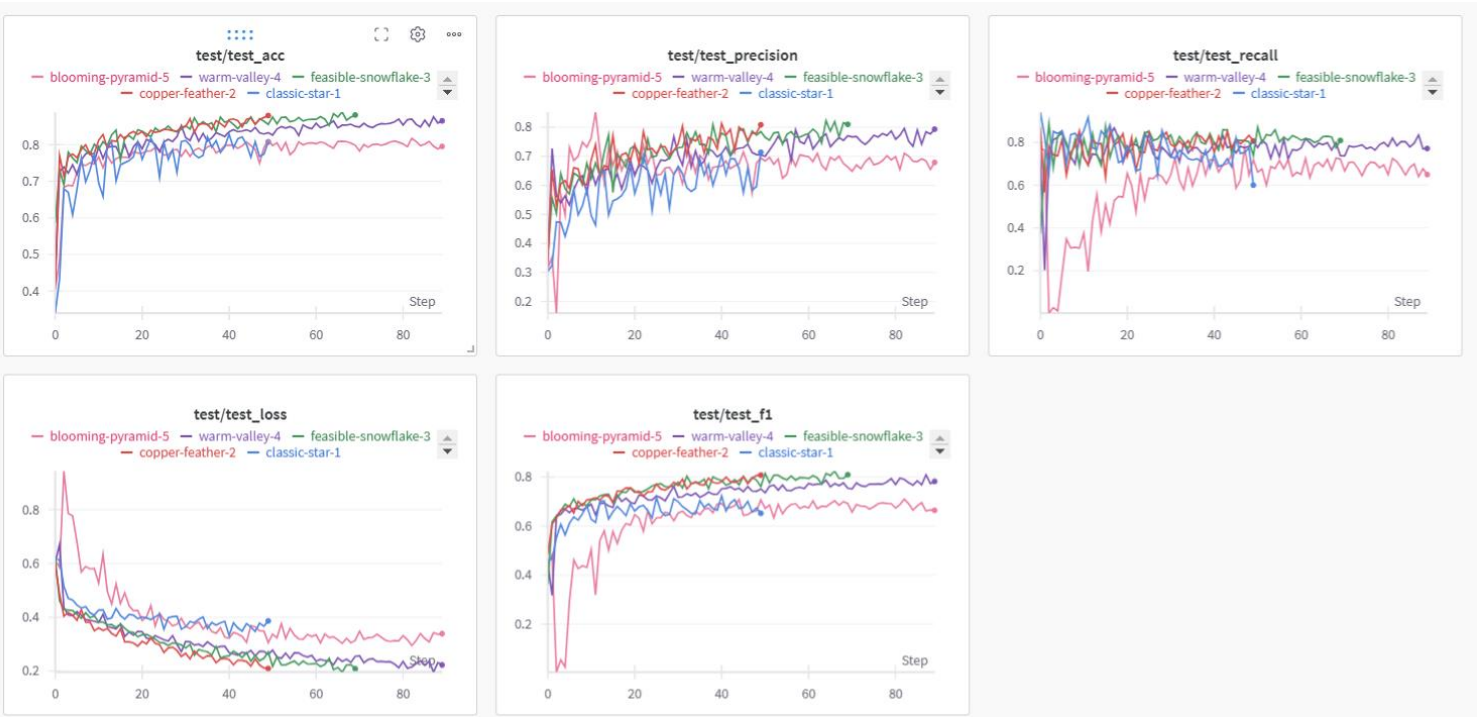
```
dropout_prob = 0.1

## CODER
self.encoder1 = UNet._block(in_channels, features, name="enc1", dropout_prob=dropout_prob)
self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
self.encoder2 = UNet._block(features, features * 2, name="enc2", dropout_prob=dropout_prob)
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
self.encoder3 = UNet._block(features * 2, features * 4, name="enc3", dropout_prob=dropout_prob)
self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
self.encoder4 = UNet._block(features * 4, features * 8, name="enc4", dropout_prob=dropout_prob)
self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

self.bottleneck = UNet._block(features * 8, features * 16, name="bottleneck", dropout_prob=dropout_prob)
```

També augmentarem els EPOCHS a 90, ja que amb drop-out, el model tarda més a arribar a la convergència.



Aquesta nova execució (lila) no millora l'anterior model. Es possible que el drop-out sigui massa baix. A la següent prova pujarem el drop-out a 0.5 a les darreres capes.

Color rosa: prova nova.



Veim que empitjoren bastant les mètriques al conjunt de testing, per tant, concluim que el drop-out no ens ajuda a millorar el model.

La millor prova ha estat la de color verd (la tercera que hem fet).

YOLO

A continuació farem experiments damunt el model de detecció YOLOv5. També es pot utilitzar per a la segmentació, però nosaltres provarem el seu funcionament amb la detecció.

Utilitzam el model oferit per la llibreria ultralytics

```
model = YOLO('yolov5n.pt')
```

Recordam que el model utilitza les dades que hem guardat en aquest directori:

Per dir-li exactament on podrà trobar les dades per entrenar i avaluar hem de crear un fitxer '.yaml' amb la següent informació:

data.yaml

```
train: /content/dataset/train/images
val: /content/dataset/val/images
test: /content/dataset/test/images

nc: 2  # Number of classes (excluding background)
names: ['cougar_body', 'windsor_chair']  # List of class names
```

I per entrenar el model:

```
train_results = model.train(
    data="/content/data.yaml",
    epochs=100,
    imgsz=288,
    verbose=True
)

metrics = model.val()
test_metrics = model.val(split='test')

results = model(X_test[0])
results[0].show()


path = model.export(format="onnx")
```

Li passem el path del document '.yaml'. Li deim quantes epochs volem executar i el size de les imatges. He decidit posar 288 perquè aquest camp només admet múltiples de 32, i el pròxim size era molt més gran que les imatges originals.

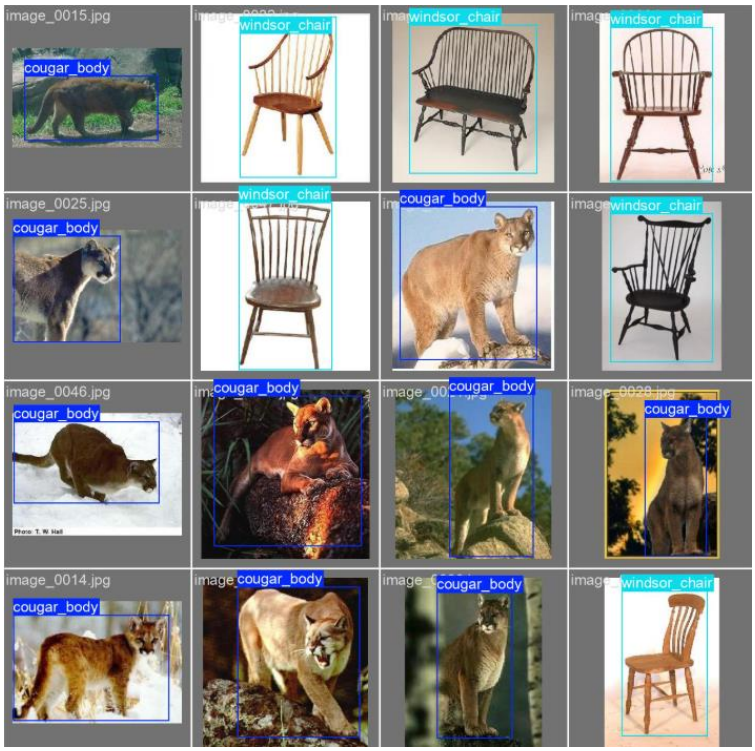
Un cop s'ha entrenat i avaluat feim que ens mostri un exemple:

Per trobar les mètriques, veurem que aquest model ha creat una carpeta anomenada runs, dins aquesta hi trobarem tota la informació sobre el nostre experiment.

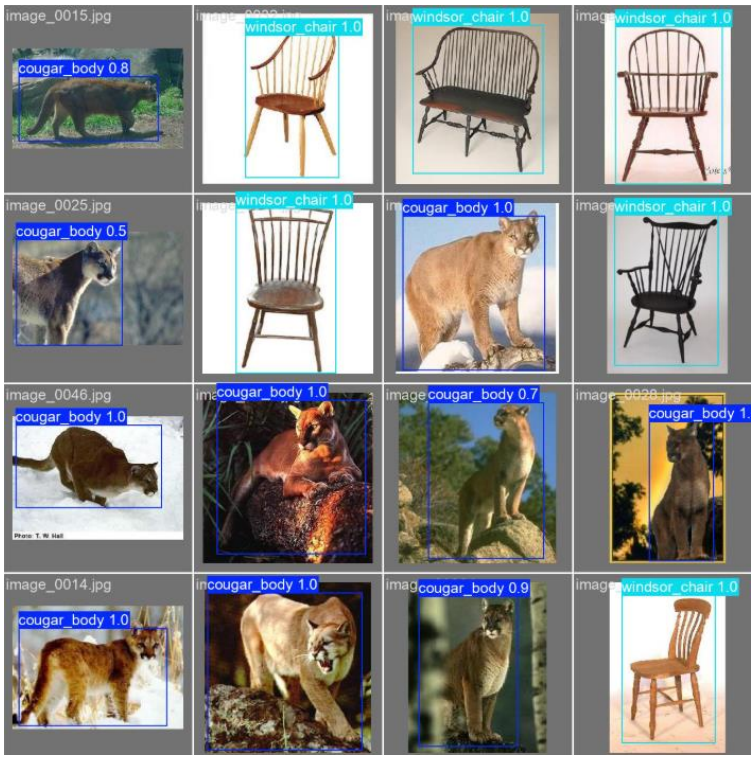
Dins train hi trobarem les dades del training, dins traint2 hi trobarem les dades de la validació i dins train3 hi trobarem les dades del testing.

L'unic problema d'aquest model es que no podem duplicar les imatges, ja que dins una mateixa carpeta no podem tenir imatges amb el mateix nom. Com a solució, es podria canviar el nom de les imatges repetides, però primer provarem el resultat fora duplicar.

Imatges amb la seua bounding box (originals):

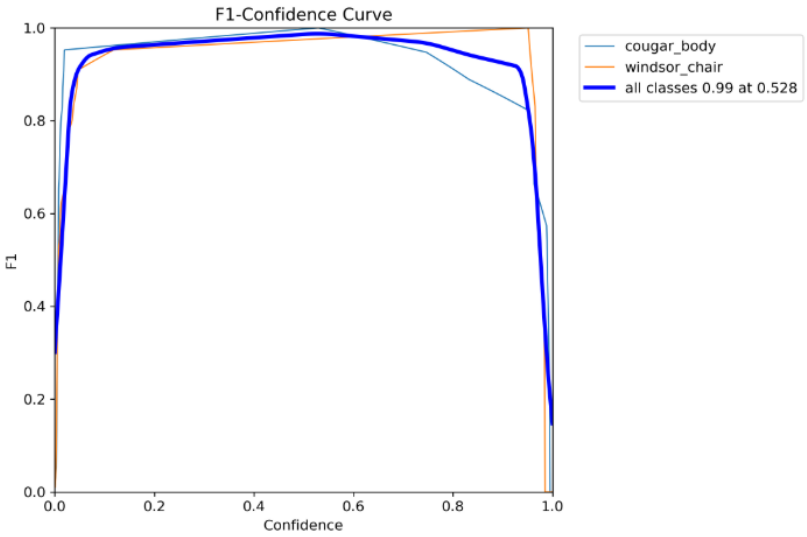


Predict de les bounding boxes amb el conjunt de testing:

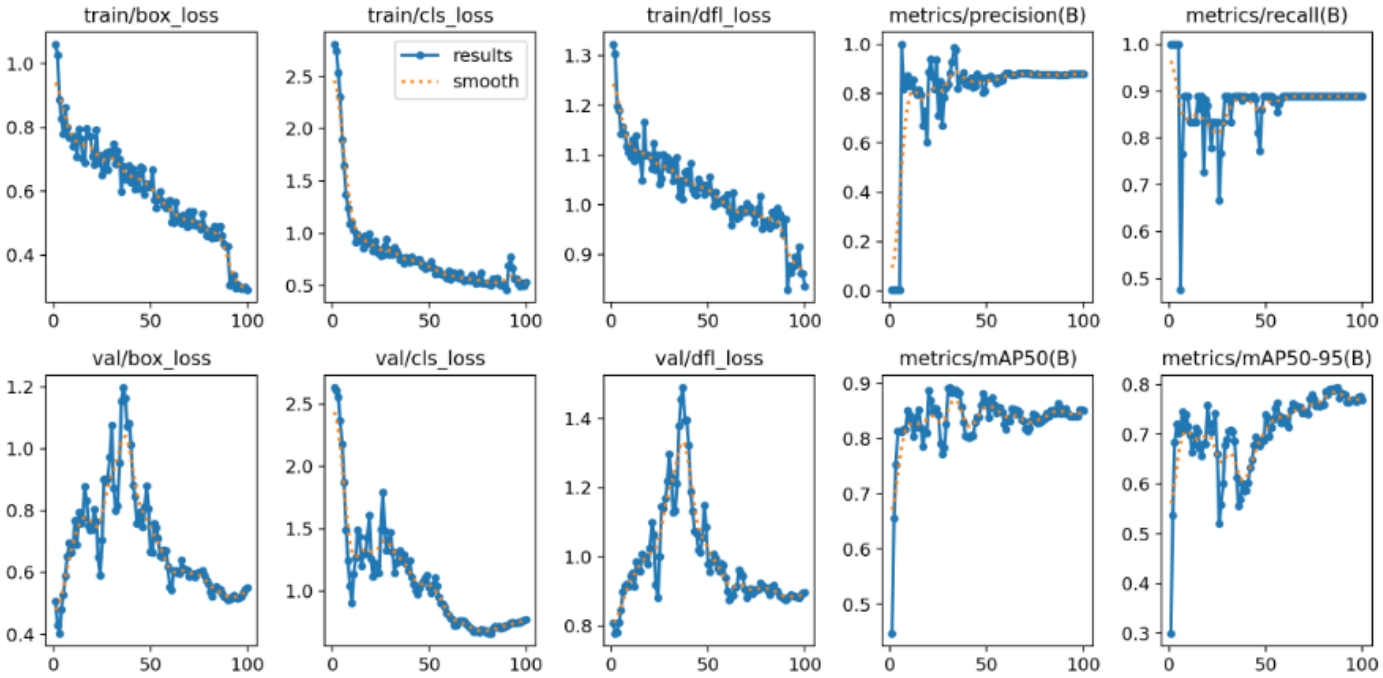


Com veiem, el model ha predit bastant bé les bounding boxes.

La corba f1-score del conjunt testing:



Totes les classes arriben al 99% de f1-score amb 0.528 de confidence.



Aquí podem veure els gràfics de les mètriques. Veim que el loss de les previsions arriba a un valor molt baix, indicant que el model prediu bé les bounding boxes de les imatges.

RESULTATS I DISCUSSIÓ

Ja hem obtingut tots els millors resultats possibles de cada model. Compararem primer els resultats dels models de classificació

Alexnet	**Summary metrics:** {} 15 keys test/test_acc: 1 test/test_f1: 1 test/test_loss: 0.0000000454130635319 test/test_precision: 1 test/test_recall: 1 train/train_acc: 1 train/train_f1: 1 train/train_loss: 0.00000000573974859579 train/train_precision: 1 train/train_recall: 1 val/val_acc: 1 val/val_f1: 1 val/val_loss: 0.0000000000003039976 val/val_precision: 1 val/val_recall: 1

Vgg16	test/test_acc: 1 test/test_f1: 1 test/test_loss: 0 test/test_precision: 1 test/test_recall: 1 train/train_acc: 1 train/train_f1: 1 train/train_loss: 0 train/train_precision: 1 train/train_recall: 1 val/val_acc: 1 val/val_f1: 1 val/val_loss: 0 val/val_precision: 1 val/val_recall: 1
Resnet	**Summary metrics:** {} 15 keys test/test_acc: 1 test/test_f1: 1 test/test_loss: 0.00025995730538852513 test/test_precision: 1 test/test_recall: 1 train/train_acc: 0.9090909090909092 train/train_f1: 0.9090909090909092 train/train_loss: 0.15201399927503767 train/train_precision: 0.9090909090909092 train/train_recall: 1 val/val_acc: 1 val/val_f1: 1 val/val_loss: 0.000814201426692307 val/val_precision: 1 val/val_recall: 1

Pròpia	**Summary metrics:** {} 15 keys test/test_acc: 1 test/test_f1: 1 test/test_loss: 0.0022259699180722237 test/test_precision: 1 test/test_recall: 1 train/train_acc: 1 train/train_f1: 1 train/train_loss: 0.002994579232108663 train/train_precision: 1 train/train_recall: 1 val/val_acc: 0.9090909090909092 val/val_f1: 0.9230769230769232 val/val_loss: 0.44283005595207214 val/val_precision: 0.8571428571428571 val/val_recall: 1

Vegent els resultats hem arribat a la conclusió de que els tres models entrenats amb 'fine tuning' ofereixen millor resultat que el nostre model pròpi. Això es deu a que els pesos han estat entrenat amb datasets més extensos i han tengut més temps d'aprendre característiques.

Així i tot, el nostre model ofereix bones mètriques, la que més empitjora respecte els models anteriors es la de pèrdua.

També pens que el bon resultat del meu model es deu a que les dues classes que m'han assignat son molt diferents enter sí. Si les meves classes haguessin estat les de 'cougar_body' i 'cougara_face', el resultat hagués estat pitjor, ja que es més complicat distingir aquestes dues classes que les de 'cougar_body' i 'windor_chair'.

En quant al model de segmentació, el millor resultat que he pogut obtenir es:

Config parameters: {} 8 keys

batch_size: 8

duplicated: true

epochs: 70

lr: 0.00015

trdist: 0.8

trsize: 164

vdist: 0.1

vsize: 21

Summary metrics: {} 15 keys

test/test_acc: 0.8810121704931972

test/test_f1: 0.8093818368684235

test/test_loss: 0.2083990573883057

test/test_precision: 0.8102034480239608

test/test_recall: 0.8085618903837449

train/train_acc: 0.961049012238824

train/train_f1: 0.935163147872351

train/train_loss: 0.09172479595456803

train/train_precision: 0.9187684795407642

train/train_recall: 0.9528502441535018

val/val_acc: 0.8829330281219631

val/val_f1: 0.7962868462469633

val/val_loss: 0.22378039360046387

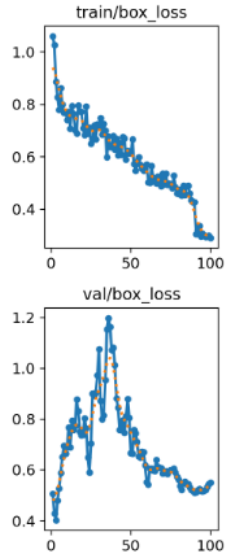
val/val_precision: 0.7894125043386008

val/val_recall: 0.8032819658475635

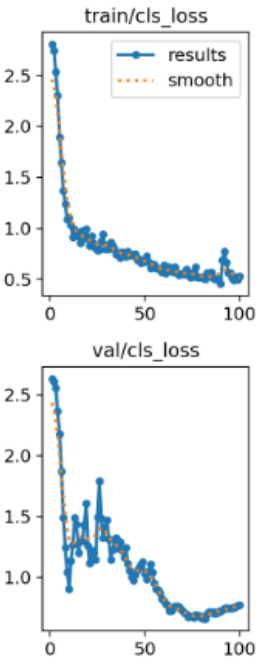
On tant l'accuracy del testing i del validation set arriben al 88%. Per millorar aquestes mètriques estaria bé que hi hagués més imatges al data set original.

També ha quedat clar que el model té més dificultat per crear les màsqueres de la classe 'cougar_body' que la de 'windsor_chair'.

Respecte el model de YOLOv5, els resultats obtinguts han estat molt positius, la pèrdua de la predicció de les bounding boxes ha estat baixa.



I la predicció de les classes de cada bounding box també ha estat baixa.



CONCLUSIONS

1) Els models entrenats amb 'fine tunning' han oferit millor resultat que el model entrenat desde 0. Donat que el nostre data set té poques imatges, la millor opció és entrenar un model amb els pesos ja definits (preentrenats).
2) El model de unet ha donat millors resultats un cop hem duplicat les imatges, demostrant que el tamany del nostre data set limita el resultat del nostre model.
3) El model YOLOv5 ha donat molts bons resultats encara que no haguem pogut emplear les imatges duplicades.
4) El learning rate i la distribució training/validation/testing s'han provat a canviar però els que millor resultat han donat son els que hem vist a aquest projecte.

MANUAL D'USUARI

```
architectures = {'alexnet': 0,'vgg': 1,'resnet': 2,'unet':3,'propi':4,'yolo':5}
MODEL = architectures['vgg']

ENTRENAR = True
ENLLAC_PESOS = '/content/my_model.pt'
WEIGHTSANDBIASES = False

DOWNLOAD = False
DUPLICAR_DADES = True
```

Entrenar: posar a false si només volem provar els models.

Enllac_pesos: l'enllaç del fitxer de pesos. El fitxer de pesos ha de correspondre al model triat.

Download: Descarregar data set.

Duplicar Dades: Si imatges duplicades (utilitzat per millorar l'entrenament).