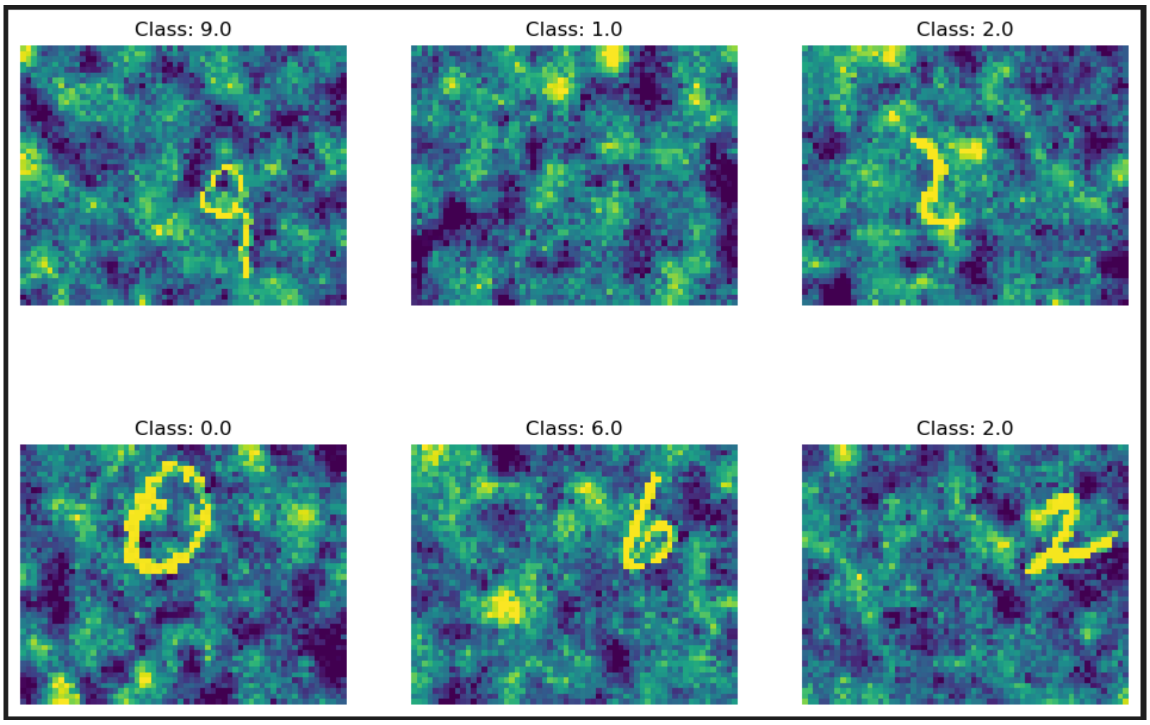
Deep Learning - Assignment 2

Carlos Vega and Marcos Alonso

Object Location

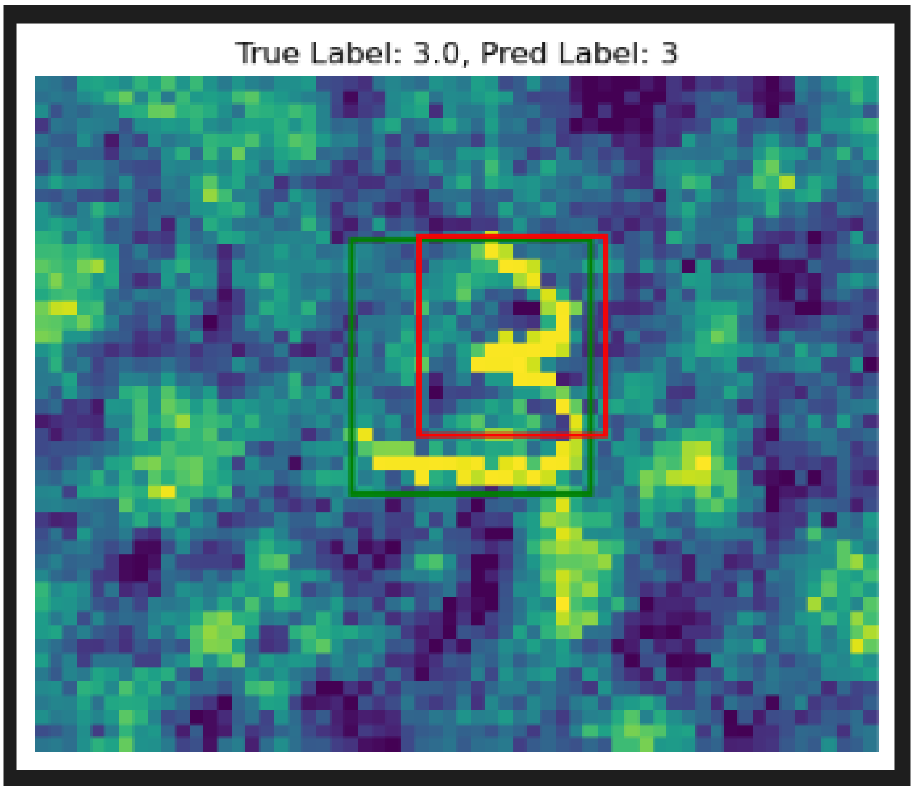
The goal of this first exercise is to create a neural network that would be able to distinguish between images that contain a number from images that do not. Before getting into the code structure let's first take a look at the problem and how the data that we will study look like.
The images we will study will look similar to these ones:

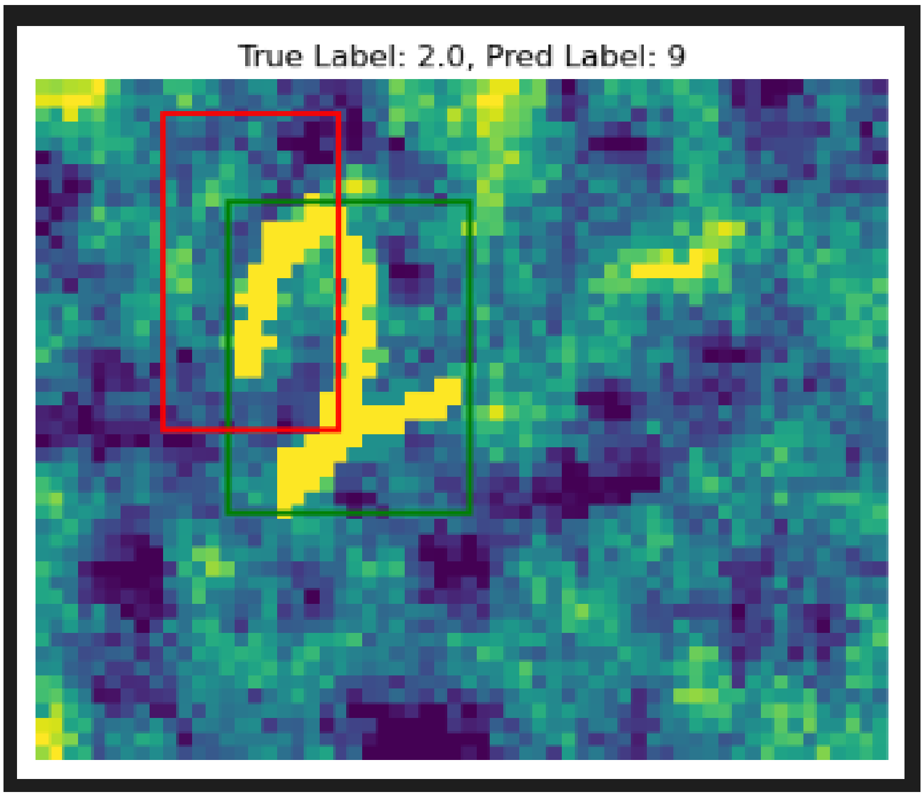


As we can see, there can be numbers from 0 to 9 or maybe there is not an object. The numbers are painted in a yellow color, following some different shapes from each other, making a realistic look of how a human would write these numbers on paper.
The background has some other yellow parts that would make the job of finding the numbers more challenging.

The algorithm will output a value of Pc between 0 and 1, being only the values of the sigmoid of Pc greater than 0.5 a detection of an object.

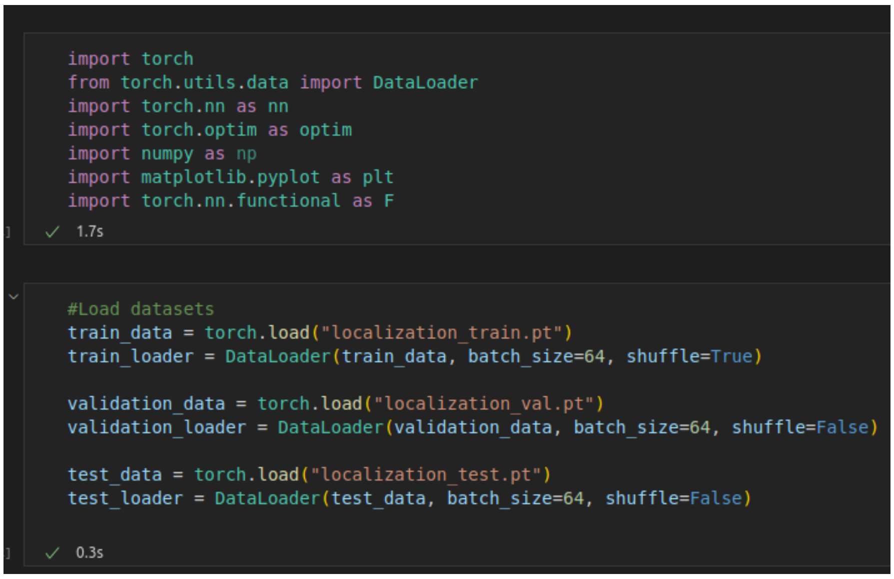
An example of how the location should work could be this, where the real number area is the green one, and the predicted one the red one. As we can see the red area is not doing the work pretty well, our work would be to find the approach that does the best job.





Now let’s get into the code, and how the neural networks developed.

The first step was to import the multiple libraries and load the datasets and convert them into DataLoaders to afterwards iterate between the images and labels in the best way.

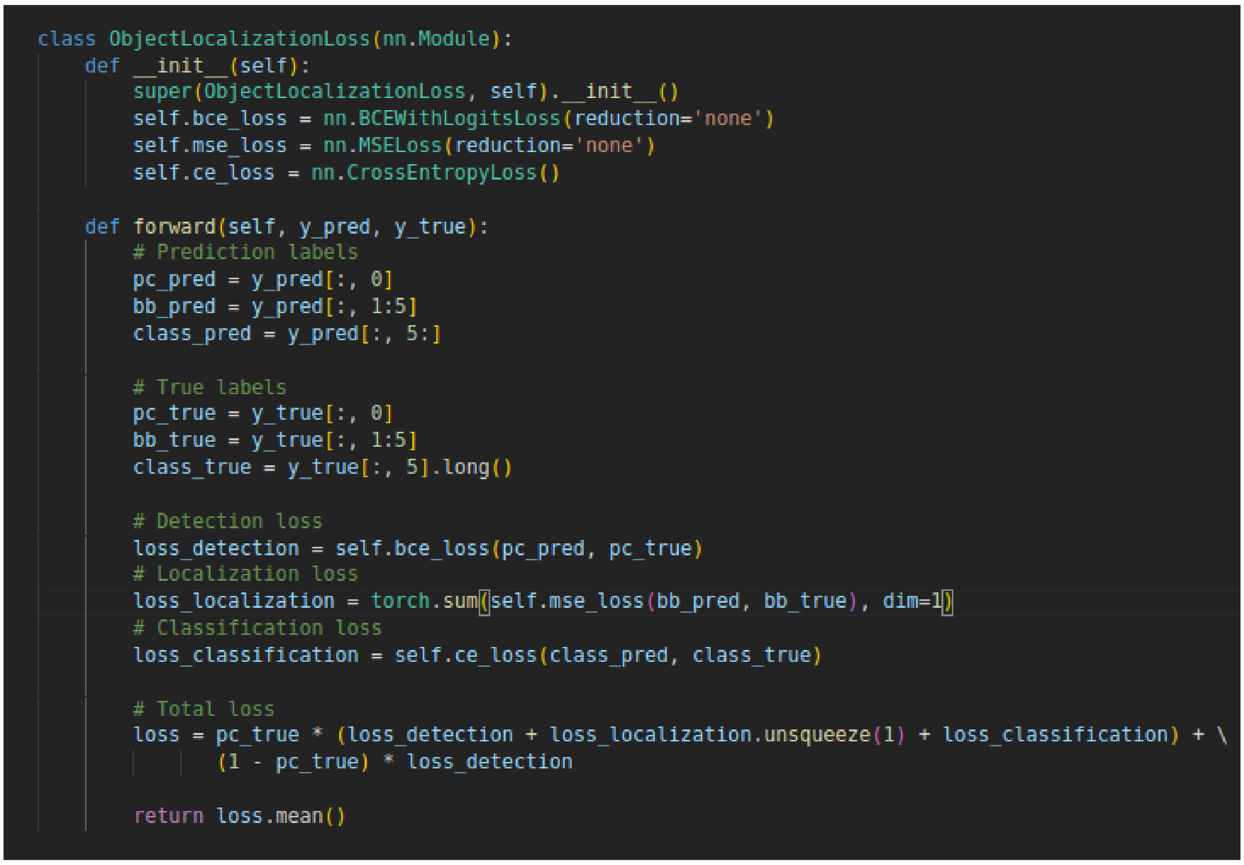


```
import torch
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import torch.nn.functional as F
```
✓ 1.7s

```
#Load datasets
train_data = torch.load("localization_train.pt")
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)

validation_data = torch.load("localization_val.pt")
validation_loader = DataLoader(validation_data, batch_size=64, shuffle=False)

test_data = torch.load("localization_test.pt")
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```
✓ 0.3s

To be able to train the data we created a loss function and a train function.

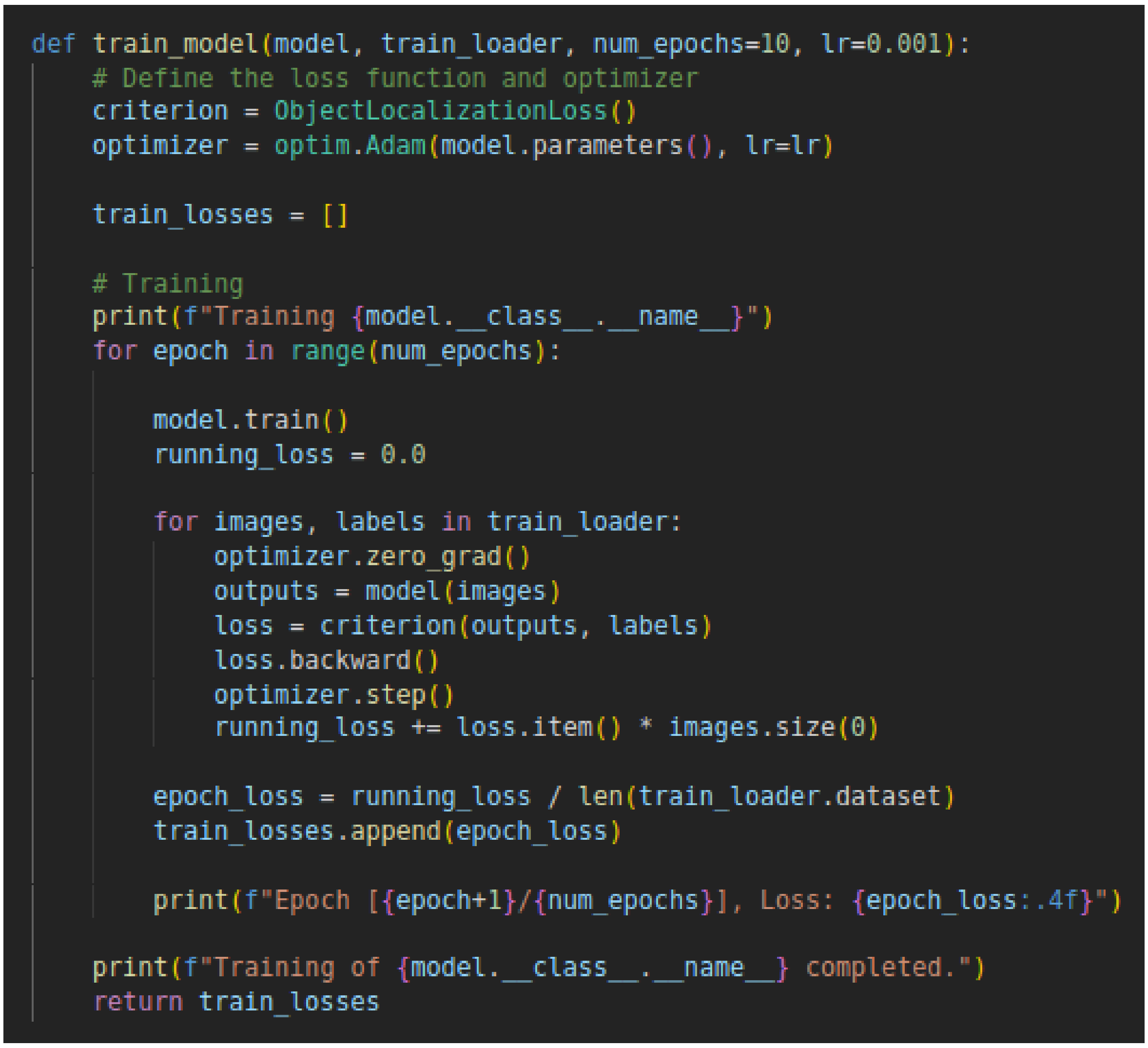
Loss:



```
class ObjectLocalizationLoss(nn.Module):
    def __init__(self):
        super(ObjectLocalizationLoss, self).__init__()
        self.bce_loss = nn.BCEWithLogitsLoss(reduction='none')
        self.mse_loss = nn.MSELoss(reduction='none')
        self.ce_loss = nn.CrossEntropyLoss()

    def forward(self, y_pred, y_true):
        # Prediction labels
        pc_pred = y_pred[:, 0]
        bb_pred = y_pred[:, 1:5]
        class_pred = y_pred[:, 5:]

        # True labels
        pc_true = y_true[:, 0]
        bb_true = y_true[:, 1:5]
        class_true = y_true[:, 5].long()

        # Detection loss
        loss_detection = self.bce_loss(pc_pred, pc_true)
        # Localization loss
        loss_localization = torch.sum(self.mse_loss(bb_pred, bb_true), dim=1)
        # Classification loss
        loss_classification = self.ce_loss(class_pred, class_true)

        # Total loss
        loss = pc_true * (loss_detection + loss_localization.unsqueeze(1) + loss_classification) + \
               (1 - pc_true) * loss_detection

        return loss.mean()
```

In the loss function, the real loss is the sum of the three different losses computed: Detection, with nn.BCEWithLogitsLoss; Localization, with nn.MSELoss; and Classification, with nn.CrossEntropyLoss.

Train:

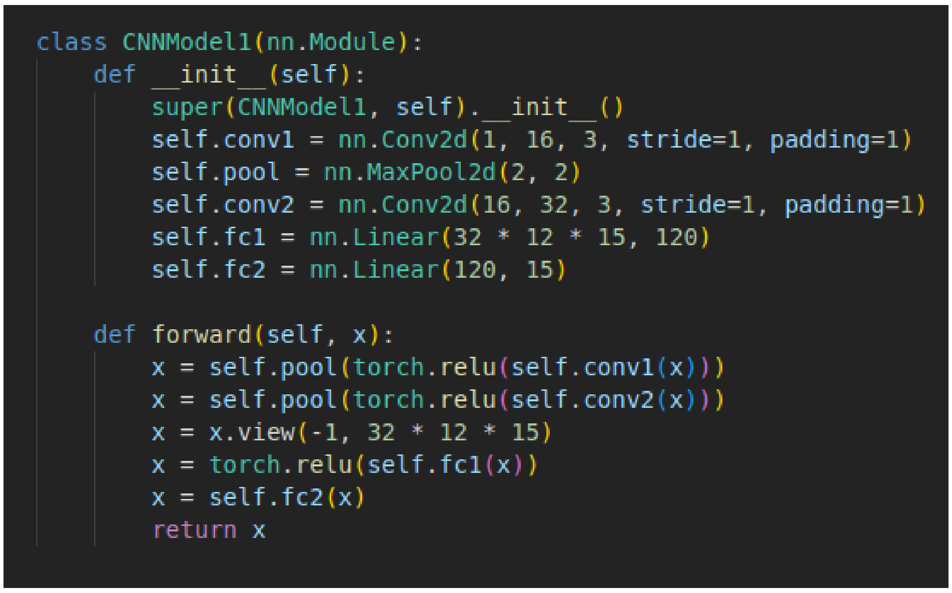


```python
def train_model(model, train_loader, num_epochs=10, lr=0.001):
    # Define the loss function and optimizer
    criterion = ObjectLocalizationLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    train_losses = []

    # Training
    print(f"Training {model.__class__.__name__}")
    for epoch in range(num_epochs):

        model.train()
        running_loss = 0.0

        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * images.size(0)

        epoch_loss = running_loss / len(train_loader.dataset)
        train_losses.append(epoch_loss)

        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")

    print(f"Training of {model.__class__.__name__} completed.")
    return train_losses
```

The training of the networks is a completely normal training approach. We iterate between the values of the loader, get the loss, make backpropagation, and update values.

Then we created 3 different models, from the first one that was the simplets to the third one that is the most complex one.

- **Model 1:**

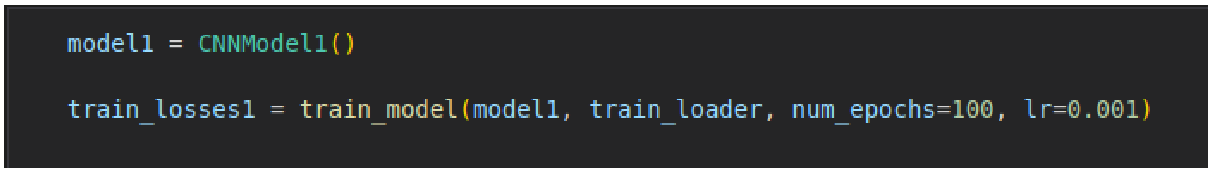


```
class CNNModel1(nn.Module):
    def __init__(self):
        super(CNNModel1, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(32 * 12 * 15, 120)
        self.fc2 = nn.Linear(120, 15)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 32 * 12 * 15)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

-Explanation of the design:

This is the basic model. Only 2 convolutional layers, and then 2 fully connected layers. The parameters are basic ones, with kernel size of 3 and stride and padding of 1. We add pool, Max Pool, to the conv layers. We dont apply relu in the final layer to be able to get negative values.

-Train:

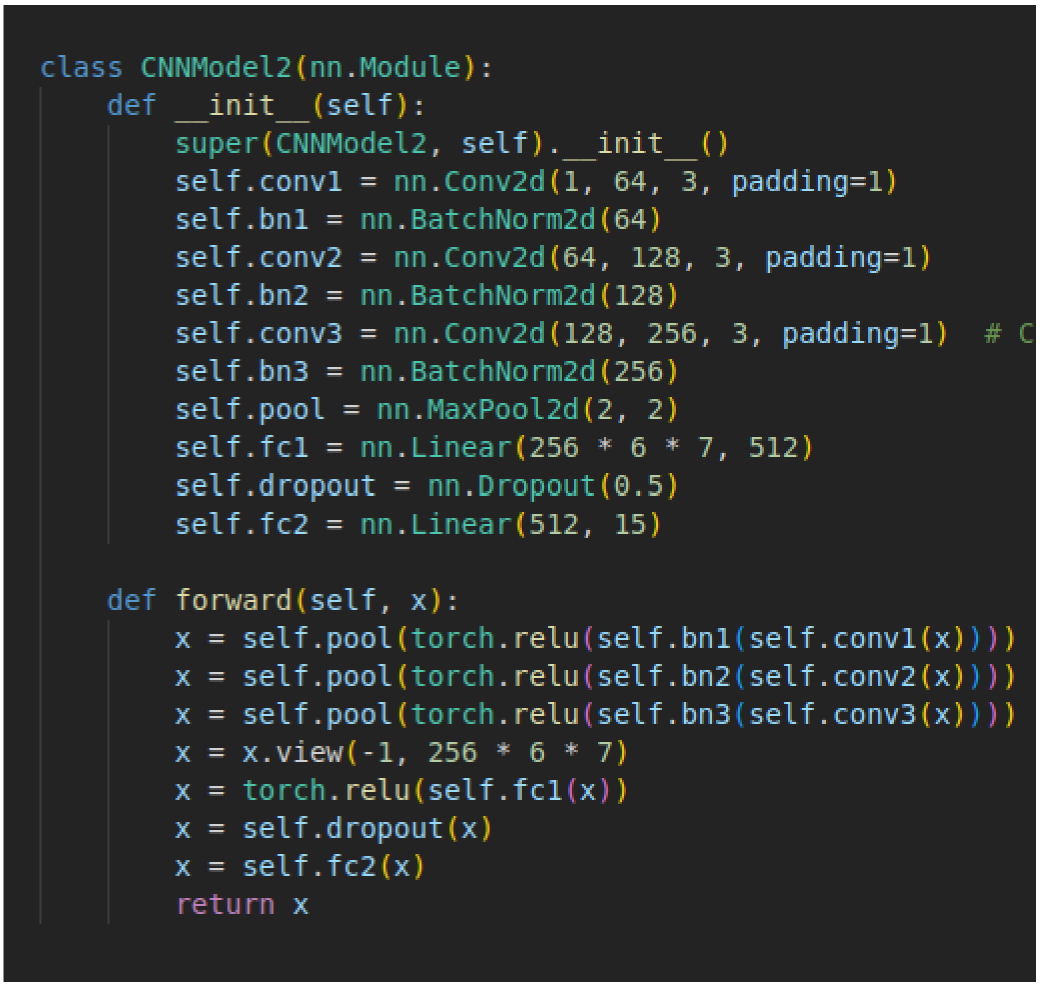


```
model1 = CNNModel1()

train_losses1 = train_model(model1, train_loader, num_epochs=100, lr=0.001)
```

We trained this model for 100 times, using a learning rate of 0.001 This way we get a slow but secure training.

The final value was: Epoch [100/100], Loss: 0.0217

- Model 2:

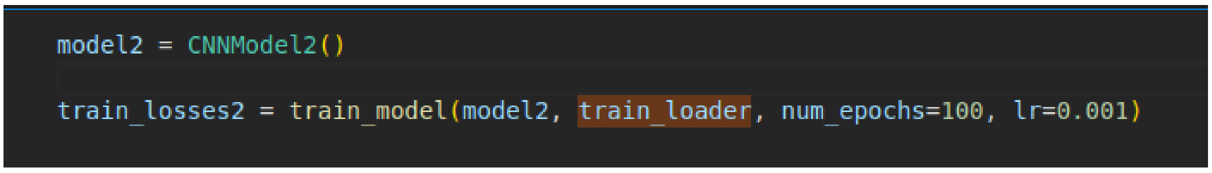


```
class CNNModel2(nn.Module):
    def __init__(self):
        super(CNNModel2, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)  # C
        self.bn3 = nn.BatchNorm2d(256)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256 * 6 * 7, 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 15)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = x.view(-1, 256 * 6 * 7)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

-Explanation of the design:

This is the intermediate design. We add more layers, more channels. We also add batchNorm and dropout. With this we are going to avoid overfitting and regularize the output.

-Train:

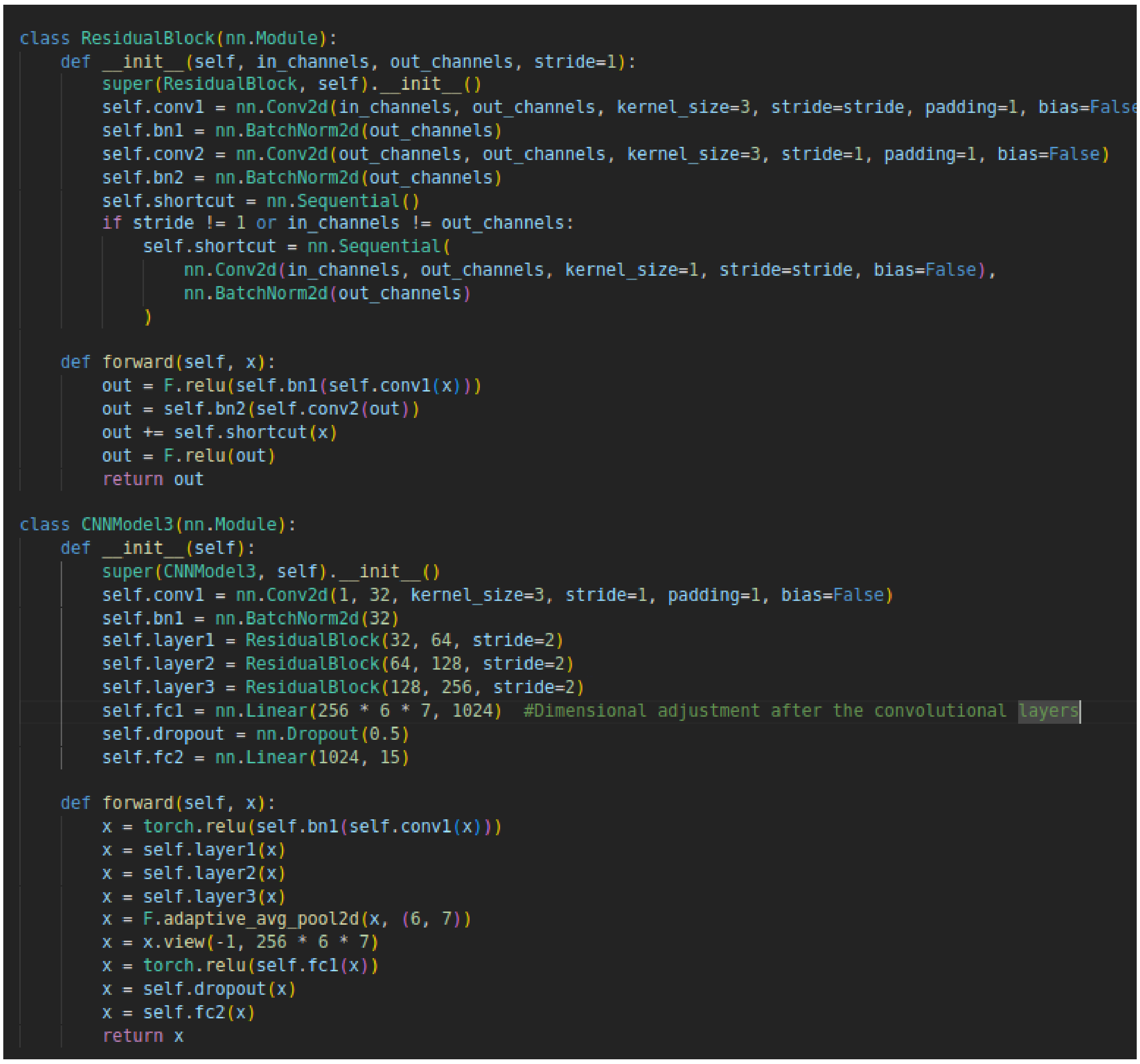


```
model2 = CNNModel2()

train_losses2 = train_model(model2, train_loader, num_epochs=100, lr=0.001)
```

We used the same parameters of this one, 100 epochs and a learning rate of 0.001.

The final value was: Epoch [100/100], Loss: 0.0922

- Model 3:

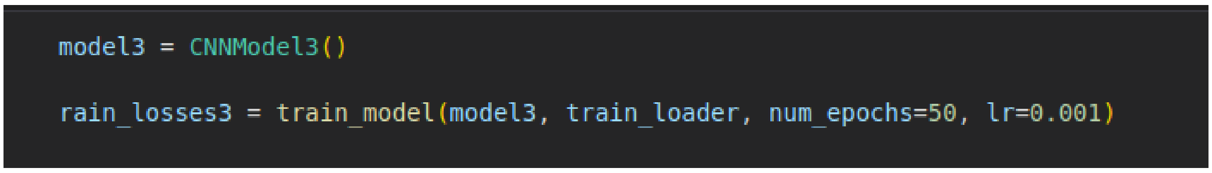


```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=Fals
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out

class CNNModel3(nn.Module):
    def __init__(self):
        super(CNNModel3, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(32)
        self.layer1 = ResidualBlock(32, 64, stride=2)
        self.layer2 = ResidualBlock(64, 128, stride=2)
        self.layer3 = ResidualBlock(128, 256, stride=2)
        self.fc1 = nn.Linear(256 * 6 * 7, 1024)  #Dimensional adjustment after the convolutional layers
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(1024, 15)

    def forward(self, x):
        x = torch.relu(self.bn1(self.conv1(x)))
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = F.adaptive_avg_pool2d(x, (6, 7))
        x = x.view(-1, 256 * 6 * 7)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

-Explanation of the design:

This is an advanced design. We created residual blocks, to try to keep better the gradients and don't lose information relevant to the training. We keep the features of the other networks as well.

-Train:



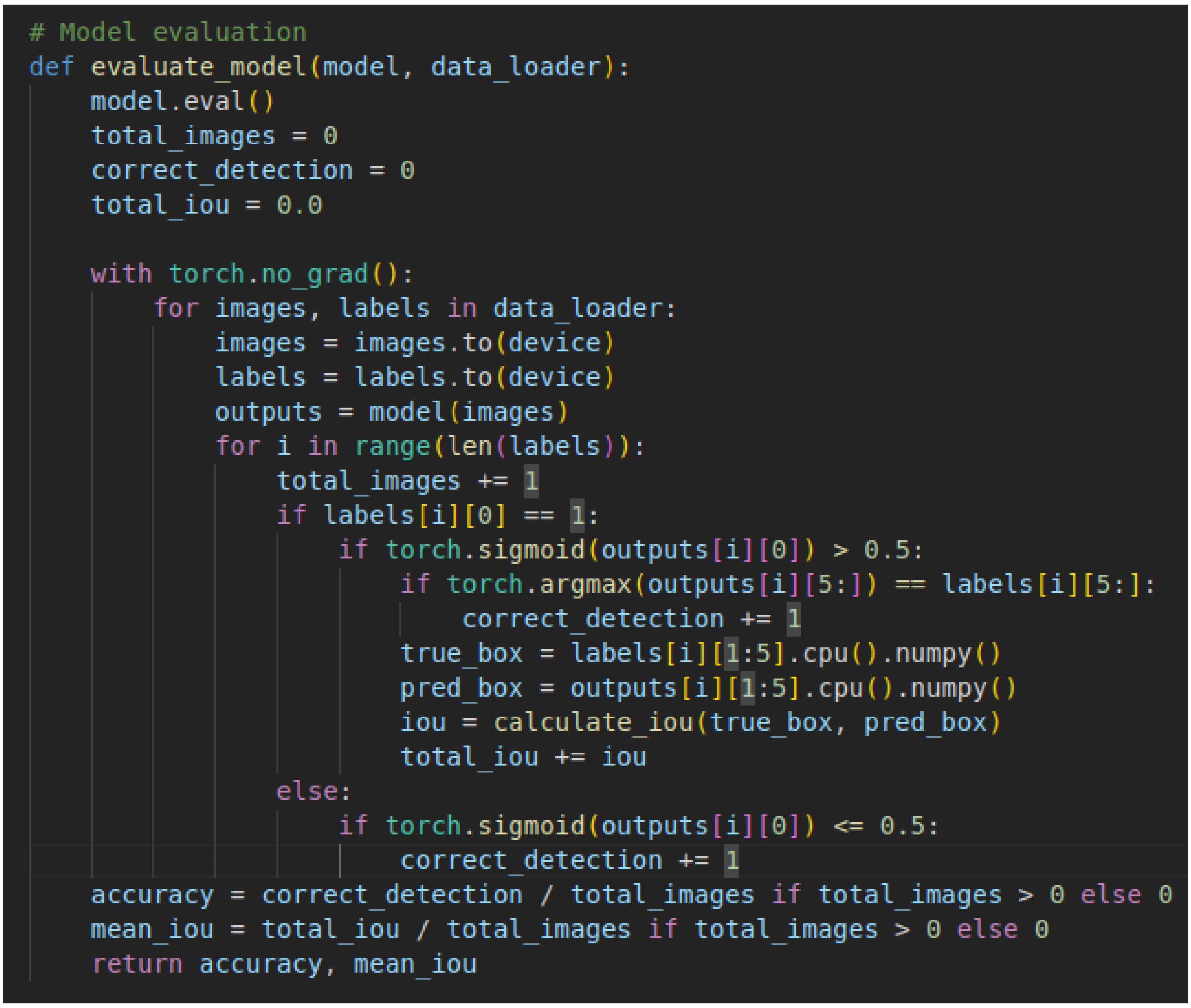
```python
model3 = CNNModel3()

rain_losses3 = train_model(model3, train_loader, num_epochs=50, lr=0.001)
```

For this one, the run time was too long to use again 100, so we decided to bring it down to 50 and use the same learning rate.

The final value was: Epoch [50/50], Loss: 0.0596

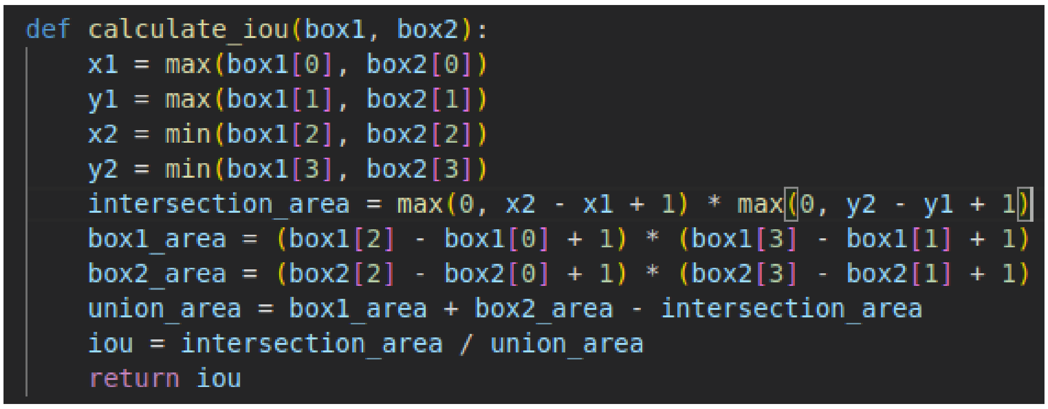
- Selection of the best model:

To be able to select which is the best model we made a function to evaluate the results. The accuracy of the function would be calculated as the number of correct results divided by the total number of images tested.



```python
# Model evaluation
def evaluate_model(model, data_loader):
    model.eval()
    total_images = 0
    correct_detection = 0
    total_iou = 0.0

    with torch.no_grad():
        for images, labels in data_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            for i in range(len(labels)):
                total_images += 1
                if labels[i][0] == 1:
                    if torch.sigmoid(outputs[i][0]) > 0.5:
                        if torch.argmax(outputs[i][5:]) == labels[i][5:]:
                            correct_detection += 1
                        true_box = labels[i][1:5].cpu().numpy()
                        pred_box = outputs[i][1:5].cpu().numpy()
                        iou = calculate_iou(true_box, pred_box)
                        total_iou += iou
                else:
                    if torch.sigmoid(outputs[i][0]) <= 0.5:
                        correct_detection += 1
    accuracy = correct_detection / total_images if total_images > 0 else 0
    mean_iou = total_iou / total_images if total_images > 0 else 0
    return accuracy, mean_iou
```

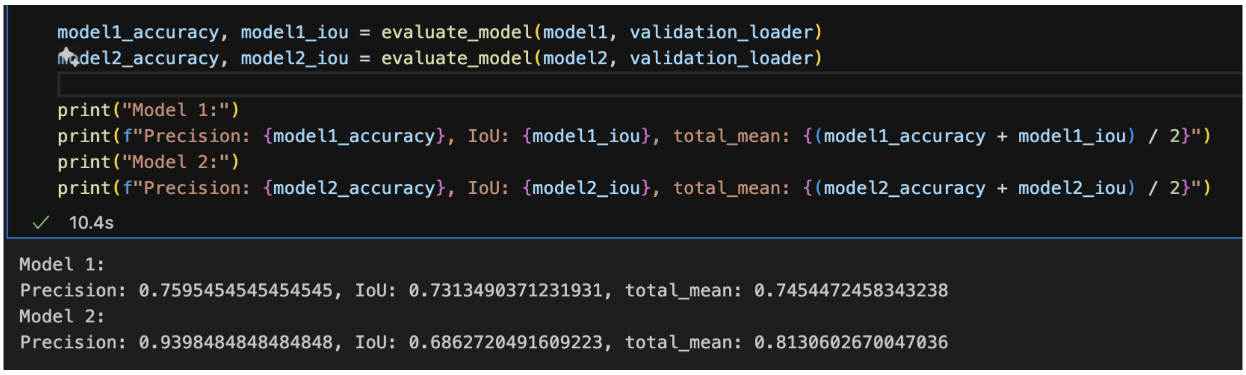
And for the performance on bounding boxes we followed the formula IoU, intersection over union. The final performance is the mean of these two values.



```
def calculate_iou(box1, box2):
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])
    intersection_area = max(0, x2 - x1 + 1) * max(0, y2 - y1 + 1)
    box1_area = (box1[2] - box1[0] + 1) * (box1[3] - box1[1] + 1)
    box2_area = (box2[2] - box2[0] + 1) * (box2[3] - box2[1] + 1)
    union_area = box1_area + box2_area - intersection_area
    iou = intersection_area / union_area
    return iou
```

Results:

We had to compute the best model manually because we runned the models on different computers in order to get the results faster. This are the results of the two first models:



```
model1_accuracy, model1_iou = evaluate_model(model1, validation_loader)
model2_accuracy, model2_iou = evaluate_model(model2, validation_loader)

print("Model 1:")
print(f"Precision: {model1_accuracy}, IoU: {model1_iou}, total_mean: {(model1_accuracy + model1_iou) / 2}")
print("Model 2:")
print(f"Precision: {model2_accuracy}, IoU: {model2_iou}, total_mean: {(model2_accuracy + model2_iou) / 2}")
✓ 10.4s

Model 1:
Precision: 0.7595454545454545, IoU: 0.7313490371231931, total_mean: 0.7454472458343238
Model 2:
Precision: 0.9398484848484848, IoU: 0.6862720491609223, total_mean: 0.8130602670047036
```

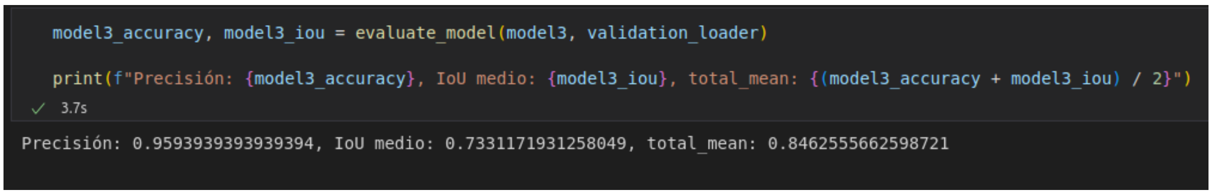
Model1 results:

Precision: 0.7595454545454545, IoU average: 0.7313490371231931, total_mean: 0.7454472458343238

Model2 results:

Precision: 0.9398484848484848, IoU average: 0.6862720491609223, total_mean: 0.8130602670047036

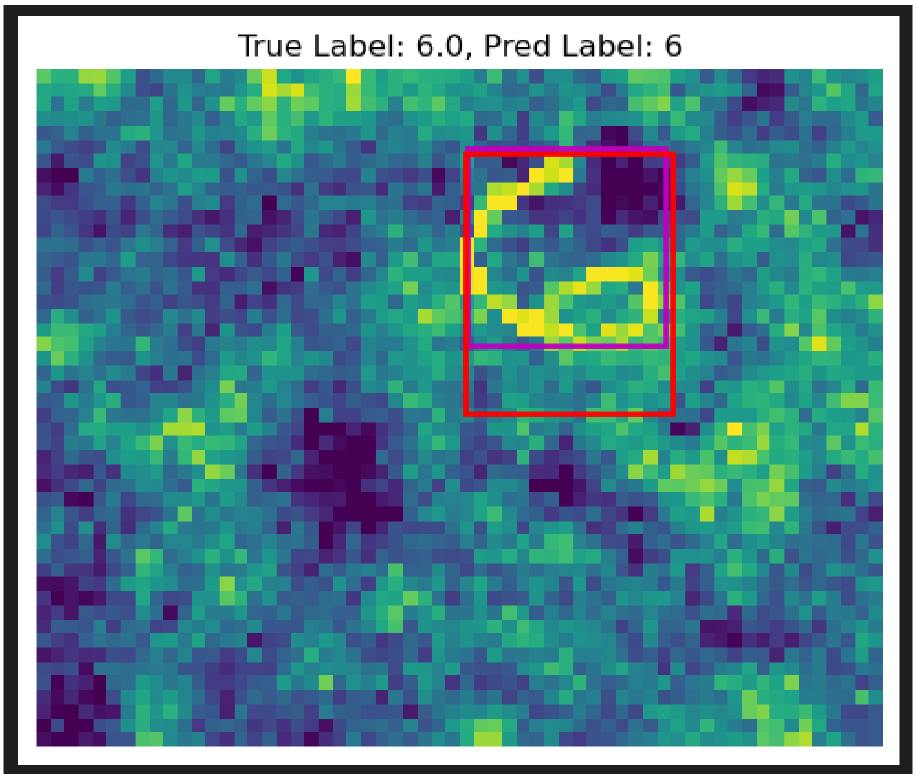
Model3 results:

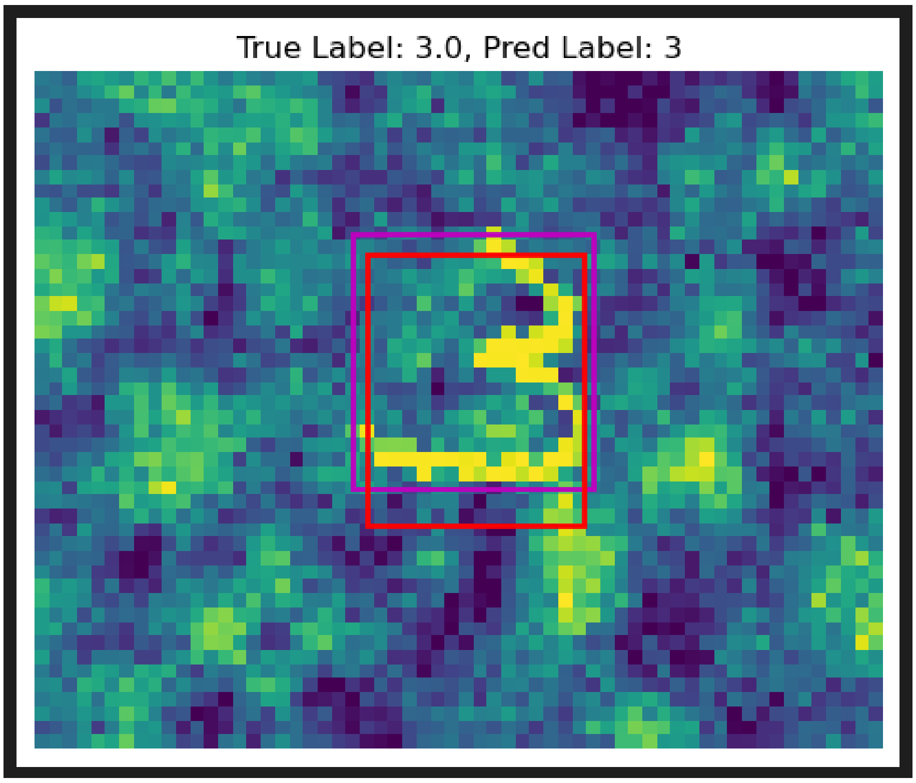


```
model3_accuracy, model3_iou = evaluate_model(model3, validation_loader)

print(f"Precisión: {model3_accuracy}, IoU medio: {model3_iou}, total_mean: {(model3_accuracy + model3_iou) / 2}")
✓ 3.7s

Precisión: 0.9593939393939394, IoU medio: 0.7331171931258049, total_mean: 0.8462555662598721
```

Precision: 0.9593939393939394, IoU mean: 0.7331171931258049, total_mean: 0.8462555662598721

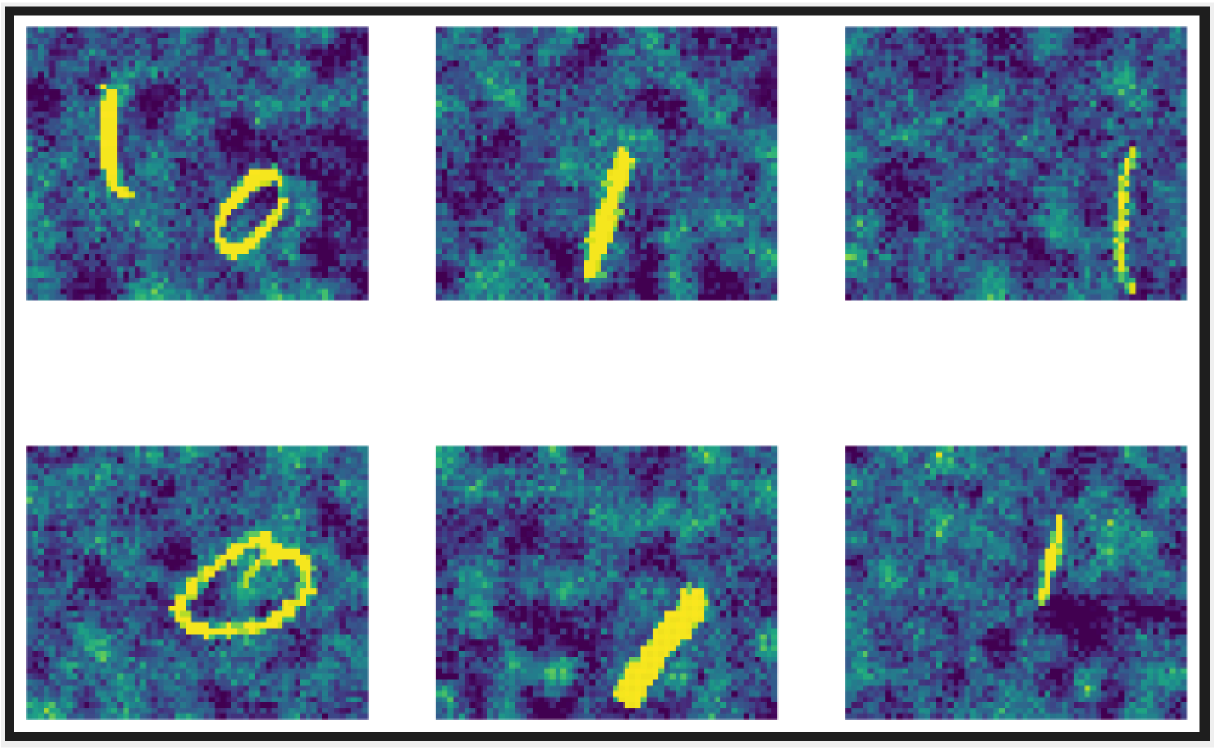
After the training the model that gave us the best results was model3 and had an accuracy of 0.96 and a mean of the intersection over union of 0.85
Some of the results were plotted at the beginning as the goal of the exercise but the results were far yet to have high precision, now after training our neuronal network we can see how the results look now.



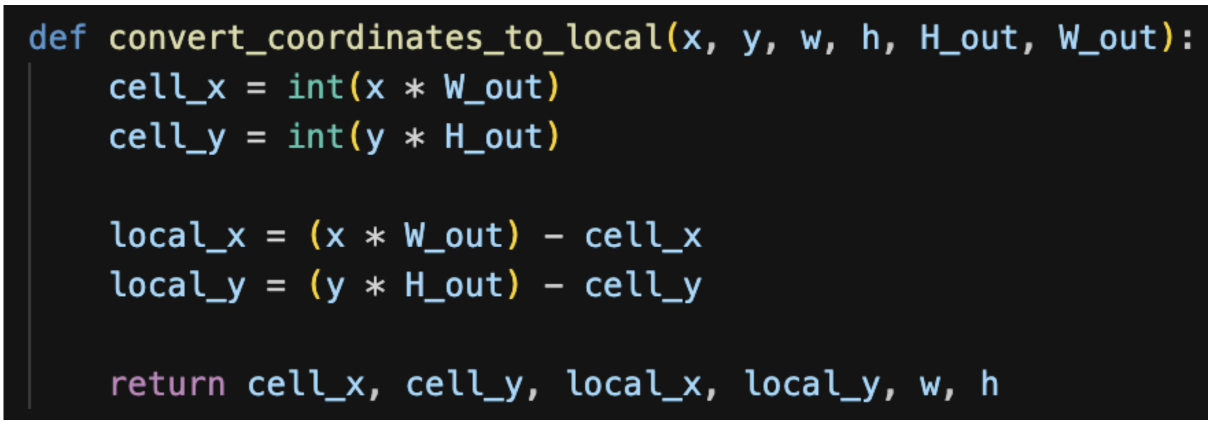


Object Detection

The task now is to generalize what we did before. Now we can have more than object per image and the possible numbers can be 0 or 1.



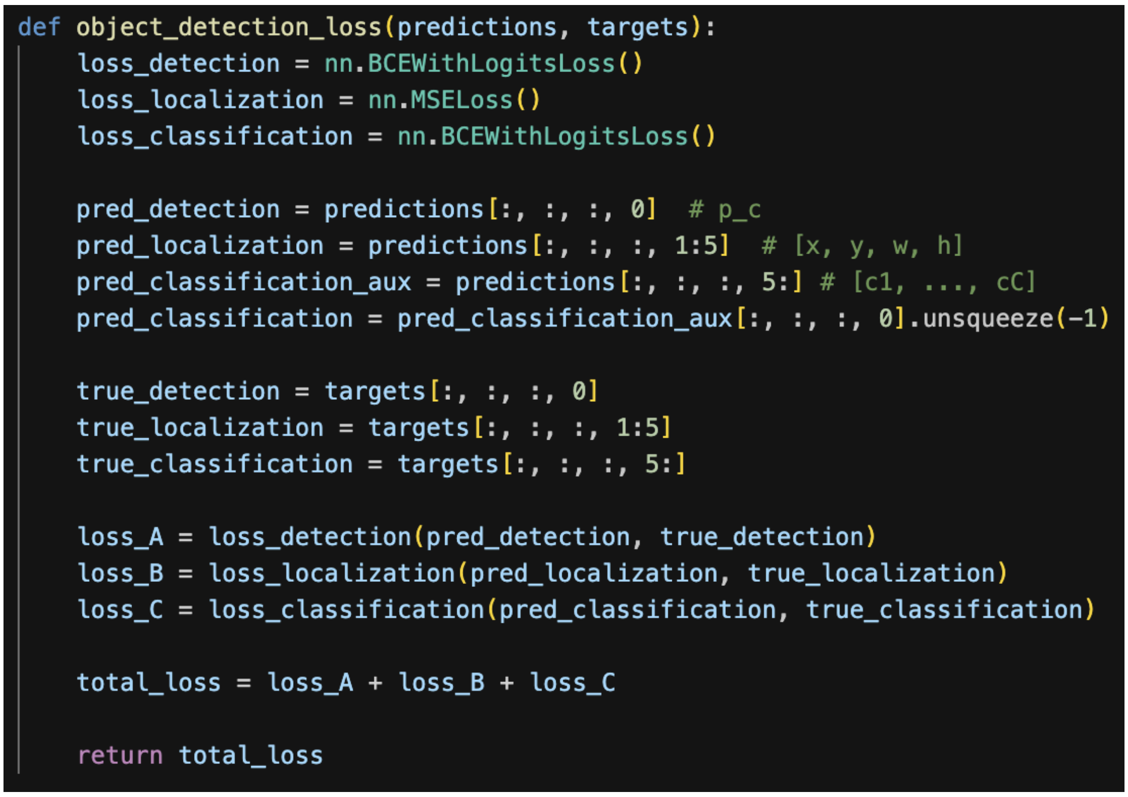
Data preprocessing:



```
def convert_coordinates_to_local(x, y, w, h, H_out, W_out):
    cell_x = int(x * W_out)
    cell_y = int(y * H_out)

    local_x = (x * W_out) - cell_x
    local_y = (y * H_out) - cell_y

    return cell_x, cell_y, local_x, local_y, w, h
```

First of all, we need to adjust the input of the networks. We create this function, and then we fill the tensors with the cells we created (2 x 3). This way we get the object location into each cell. We then merge the datasets to create the DataLoaders.

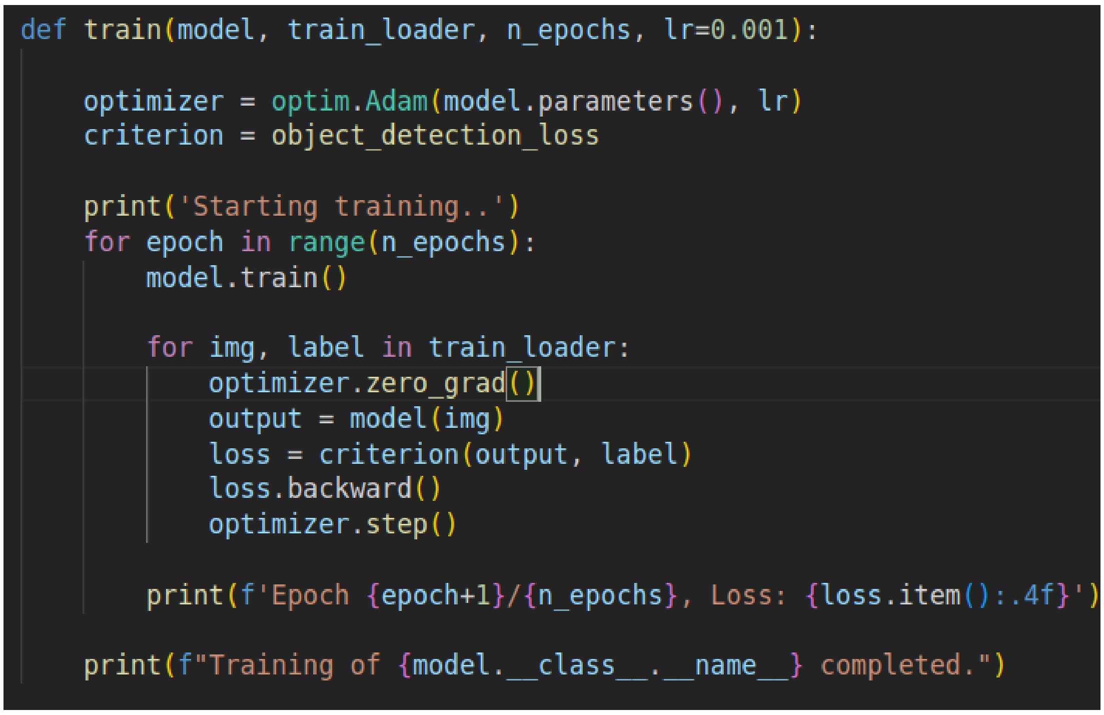
Loss:



```python
def object_detection_loss(predictions, targets):
    loss_detection = nn.BCEWithLogitsLoss()
    loss_localization = nn.MSELoss()
    loss_classification = nn.BCEWithLogitsLoss()

    pred_detection = predictions[:, :, :, 0]  # p_c
    pred_localization = predictions[:, :, :, 1:5]  # [x, y, w, h]
    pred_classification_aux = predictions[:, :, :, 5:] # [c1, ..., cC]
    pred_classification = pred_classification_aux[:, :, :, 0].unsqueeze(-1)

    true_detection = targets[:, :, :, 0]
    true_localization = targets[:, :, :, 1:5]
    true_classification = targets[:, :, :, 5:]

    loss_A = loss_detection(pred_detection, true_detection)
    loss_B = loss_localization(pred_localization, true_localization)
    loss_C = loss_classification(pred_classification, true_classification)

    total_loss = loss_A + loss_B + loss_C

    return total_loss
```

Is almost the same, but we change classification loss to BCEWithLogitsLoss. The total loss is the sum of the tree losses again.

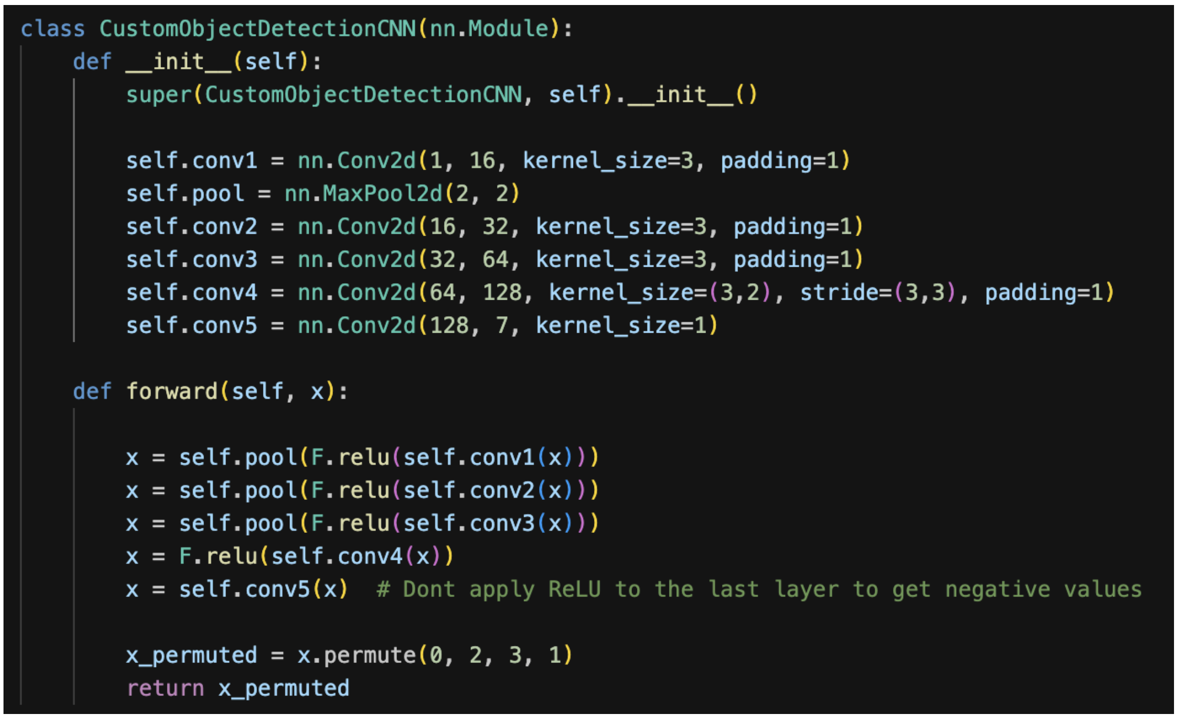
Train:



```python
def train(model, train_loader, n_epochs, lr=0.001):

    optimizer = optim.Adam(model.parameters(), lr)
    criterion = object_detection_loss

    print('Starting training..')
    for epoch in range(n_epochs):
        model.train()

        for img, label in train_loader:
            optimizer.zero_grad()
            output = model(img)
            loss = criterion(output, label)
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch+1}/{n_epochs}, Loss: {loss.item():.4f}')

    print(f"Training of {model.__class__.__name__} completed.")
```

Train function is the same, is a general train function for this algorithms.

- Model 1:

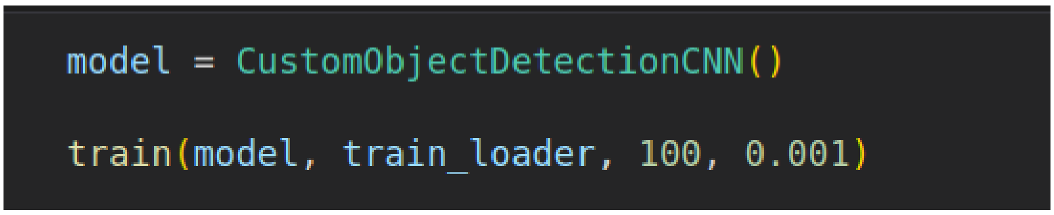


```
class CustomObjectDetectionCNN(nn.Module):
    def __init__(self):
        super(CustomObjectDetectionCNN, self).__init__()

        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(64, 128, kernel_size=(3,2), stride=(3,3), padding=1)
        self.conv5 = nn.Conv2d(128, 7, kernel_size=1)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = F.relu(self.conv4(x))
        x = self.conv5(x)  # Dont apply ReLU to the last layer to get negative values

        x_permuted = x.permute(0, 2, 3, 1)
        return x_permuted
```

-Explanation of the design:

This is the first model, with 5 convolutional layers. In the forth one, we adjust the parameters of kernel size and stride to get the desired output dimensions.

-Train:

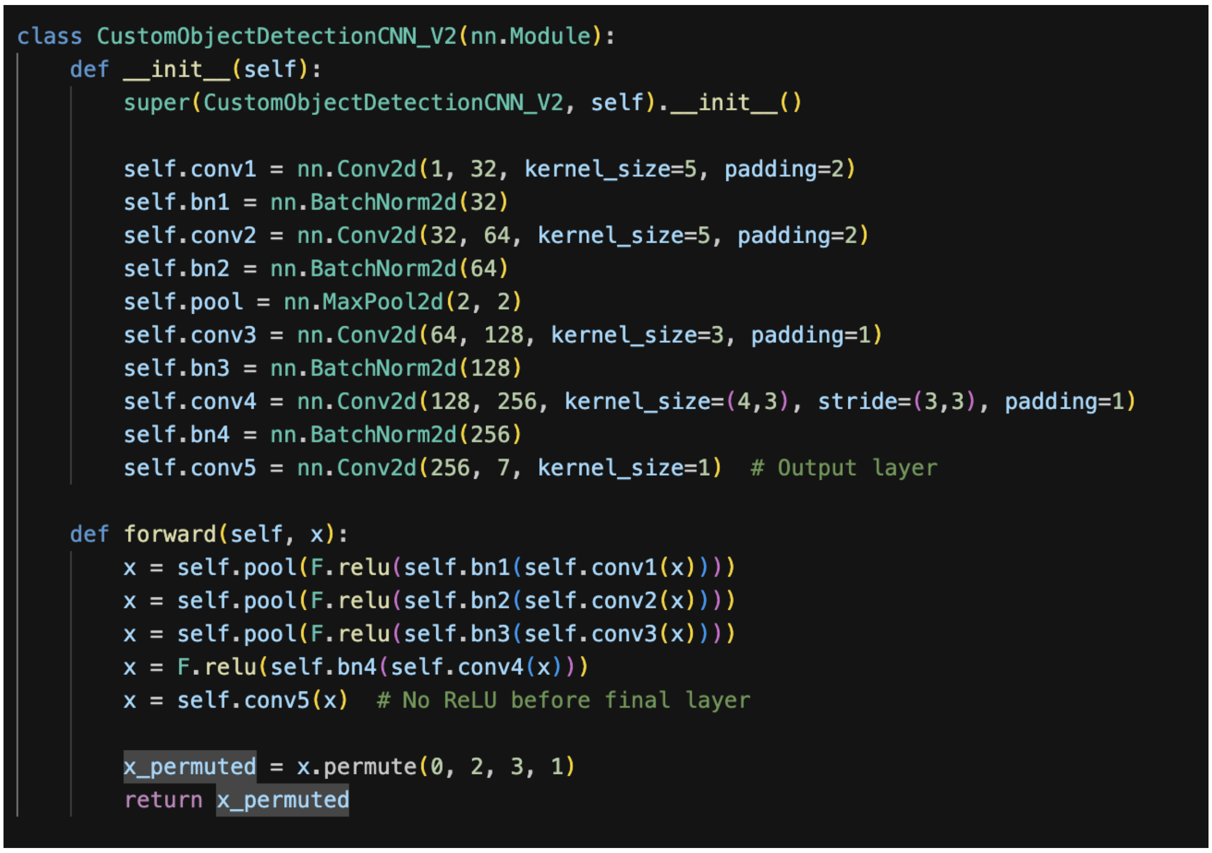


```
model = CustomObjectDetectionCNN()

train(model, train_loader, 100, 0.001)
```

We used 100 epochs and a learning rate of 0.001.

Epoch 100/100, Loss: 0.0123

- ## Model 2:

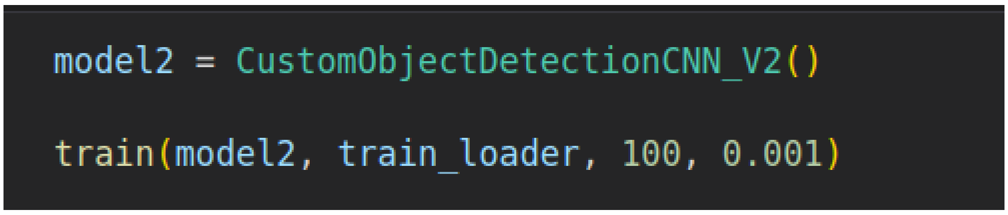


```python
class CustomObjectDetectionCNN_V2(nn.Module):
    def __init__(self):
        super(CustomObjectDetectionCNN_V2, self).__init__()

        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=(4,3), stride=(3,3), padding=1)
        self.bn4 = nn.BatchNorm2d(256)
        self.conv5 = nn.Conv2d(256, 7, kernel_size=1)  # Output layer

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = F.relu(self.bn4(self.conv4(x)))
        x = self.conv5(x)  # No ReLU before final layer

        x_permuted = x.permute(0, 2, 3, 1)
        return x_permuted
```

-Explanation of the design:

This is the second model, where we added BatchNorm to the layers. We change parameters as kernel size and padding, and again in the fourth layer we adjust them to get the desired output dimension. In all of these models we then permute the output to put the channels at the end.

-Train:

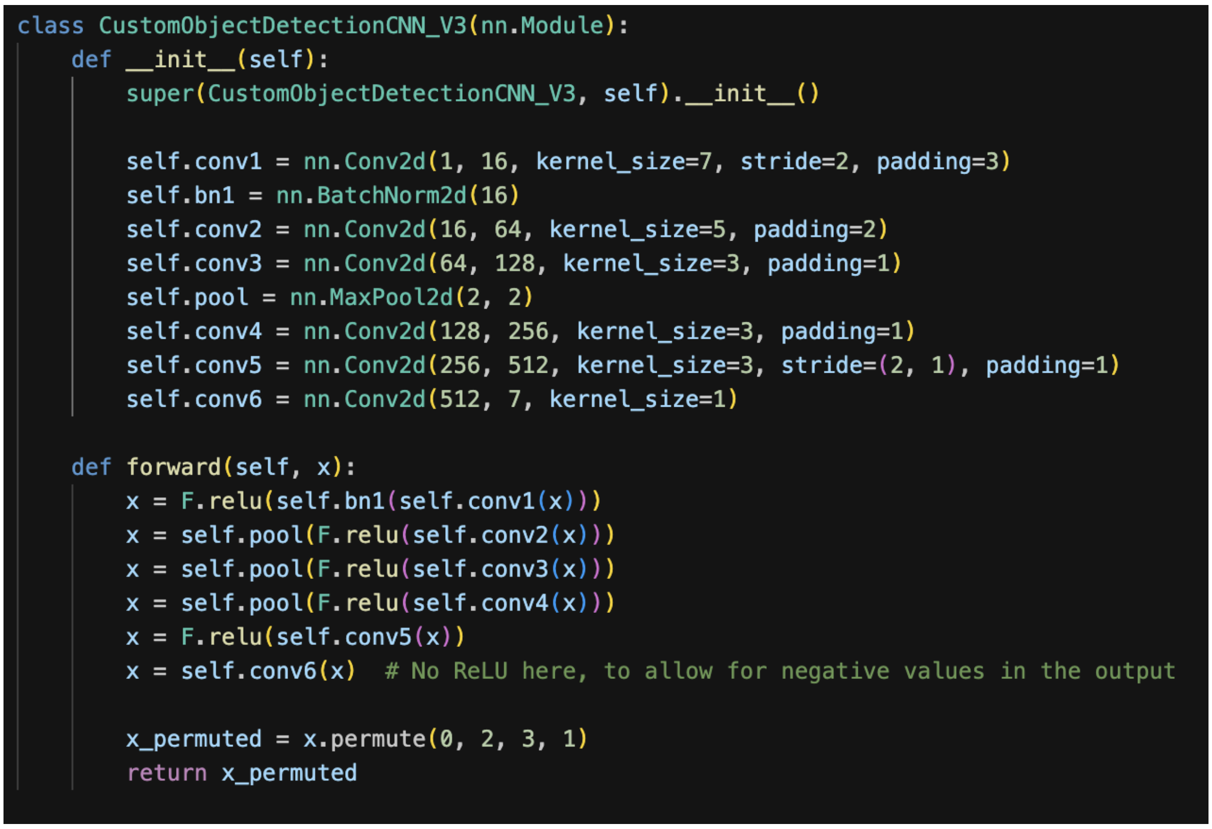


```python
model2 = CustomObjectDetectionCNN_V2()

train(model2, train_loader, 100, 0.001)
```

We again used the same parameters on this one, 100 epochs and a learning rate of 0.001.

The final value was: Epoch 100/100, Loss: 0.0004

- ## Model 3:

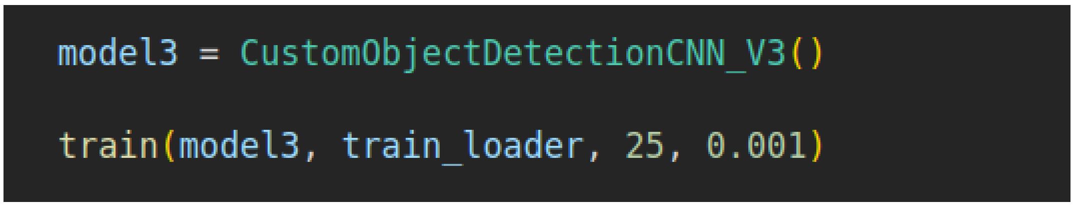


```
class CustomObjectDetectionCNN_V3(nn.Module):
    def __init__(self):
        super(CustomObjectDetectionCNN_V3, self).__init__()

        self.conv1 = nn.Conv2d(1, 16, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 64, kernel_size=5, padding=2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, stride=(2, 1), padding=1)
        self.conv6 = nn.Conv2d(512, 7, kernel_size=1)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = self.conv6(x)  # No ReLU here, to allow for negative values in the output

        x_permuted = x.permute(0, 2, 3, 1)
        return x_permuted
```

-Explanation of the design:

For the third model, we add a layer and we change kernel size and stride. Then we adjust the dimensions.

-Train:



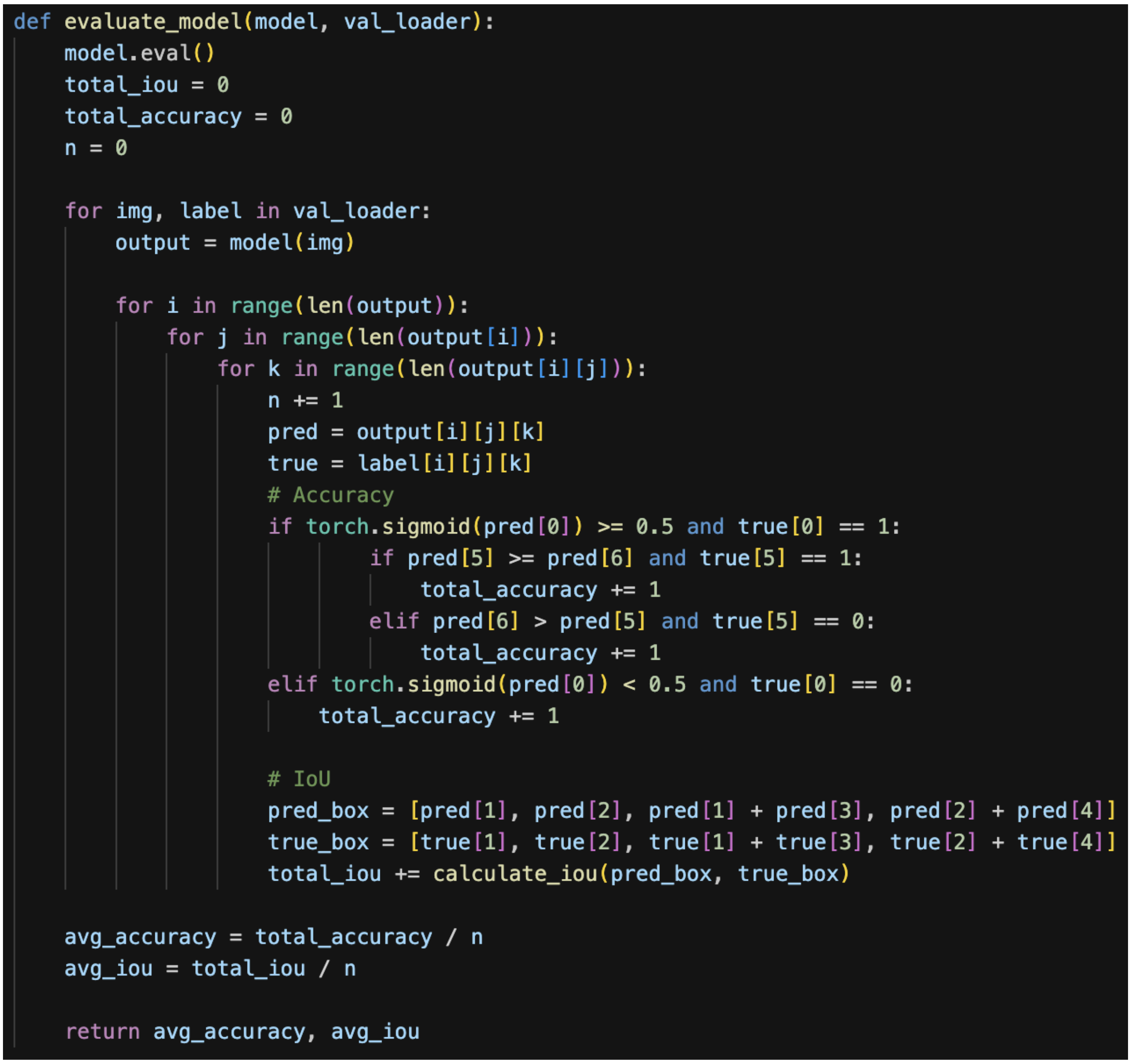
```
model3 = CustomObjectDetectionCNN_V3()

train(model3, train_loader, 25, 0.001)
```

We used only 25 epochs due to the runtime being too long to use bigger numbers.

The final value was: Epoch [25/25], Loss: 0.0019

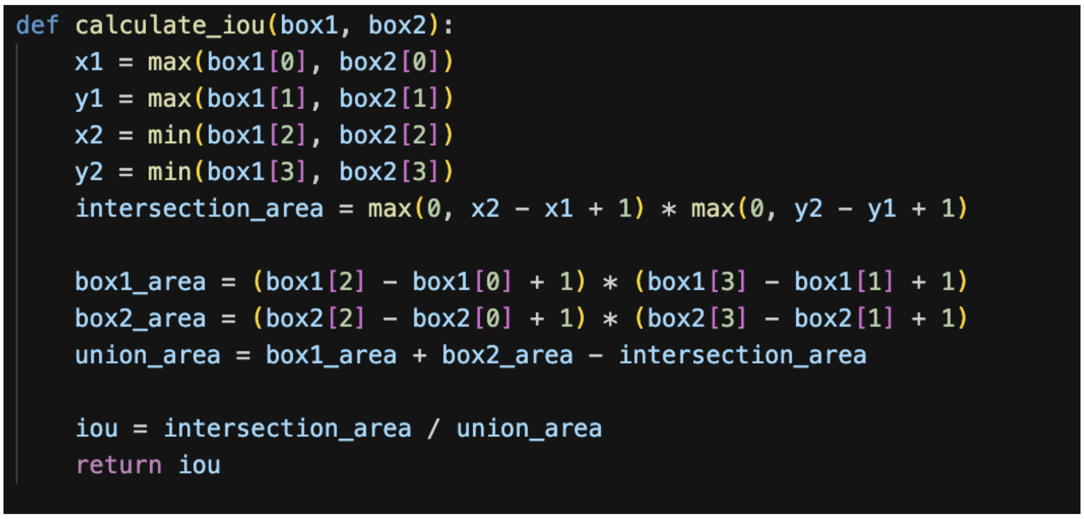
- Selection of the best model:

We will calculate the best model following the same idea as before. The accuracy of the function would be calculated as the number of correct results divided by the total number of images tested.



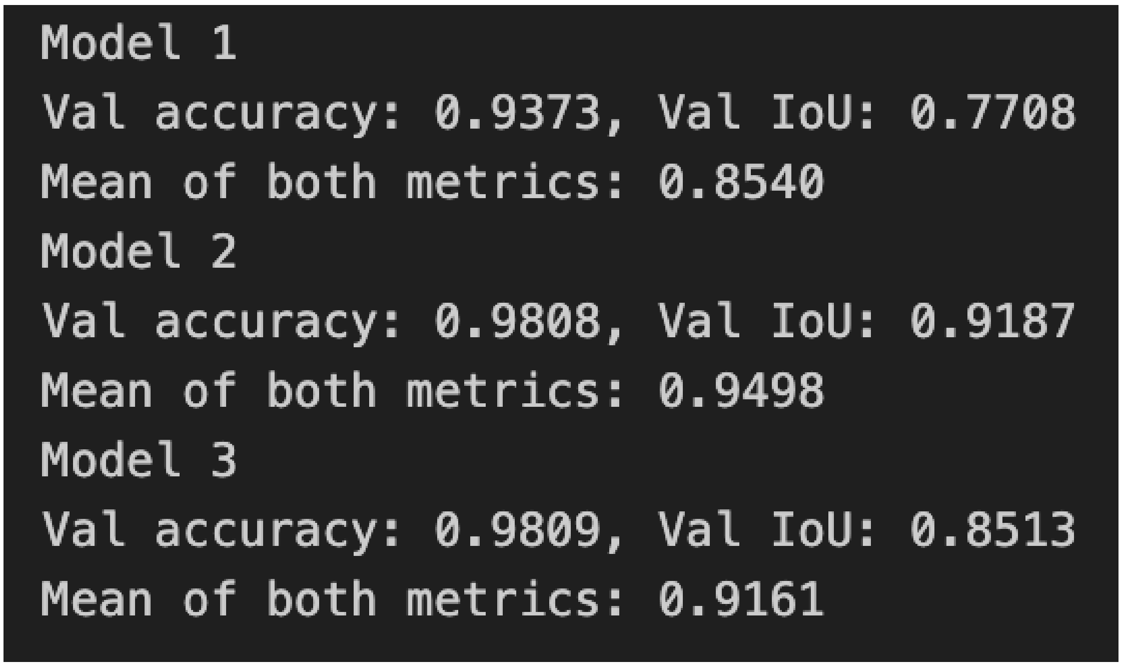
```python
def evaluate_model(model, val_loader):
    model.eval()
    total_iou = 0
    total_accuracy = 0
    n = 0

    for img, label in val_loader:
        output = model(img)

        for i in range(len(output)):
            for j in range(len(output[i])):
                for k in range(len(output[i][j])):
                    n += 1
                    pred = output[i][j][k]
                    true = label[i][j][k]
                    # Accuracy
                    if torch.sigmoid(pred[0]) >= 0.5 and true[0] == 1:
                            if pred[5] >= pred[6] and true[5] == 1:
                                total_accuracy += 1
                            elif pred[6] > pred[5] and true[5] == 0:
                                total_accuracy += 1
                    elif torch.sigmoid(pred[0]) < 0.5 and true[0] == 0:
                        total_accuracy += 1

                    # IoU
                    pred_box = [pred[1], pred[2], pred[1] + pred[3], pred[2] + pred[4]]
                    true_box = [true[1], true[2], true[1] + true[3], true[2] + true[4]]
                    total_iou += calculate_iou(pred_box, true_box)

    avg_accuracy = total_accuracy / n
    avg_iou = total_iou / n

    return avg_accuracy, avg_iou
```

And for the performance on bounding boxes we followed the formula IoU, intersection over union. The final performance is the mean of these two values.



```python
def calculate_iou(box1, box2):
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])
    intersection_area = max(0, x2 - x1 + 1) * max(0, y2 - y1 + 1)

    box1_area = (box1[2] - box1[0] + 1) * (box1[3] - box1[1] + 1)
    box2_area = (box2[2] - box2[0] + 1) * (box2[3] - box2[1] + 1)
    union_area = box1_area + box2_area - intersection_area

    iou = intersection_area / union_area
    return iou
```

We calculate it in each model, and this are the results on the validation dataset.

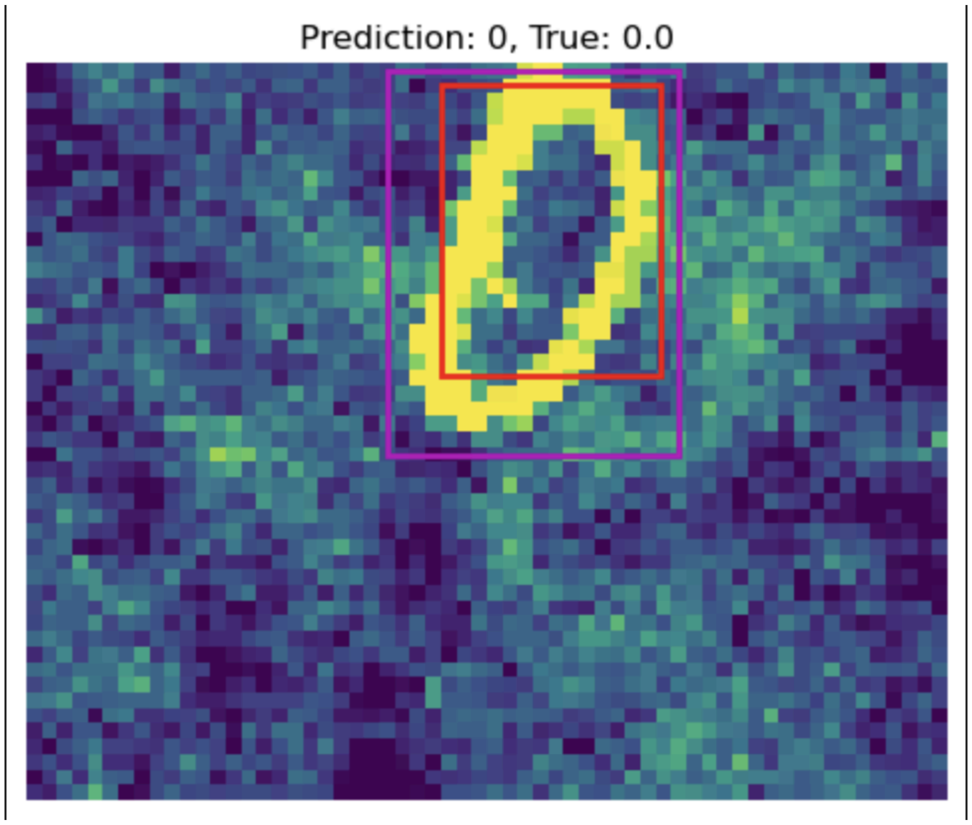


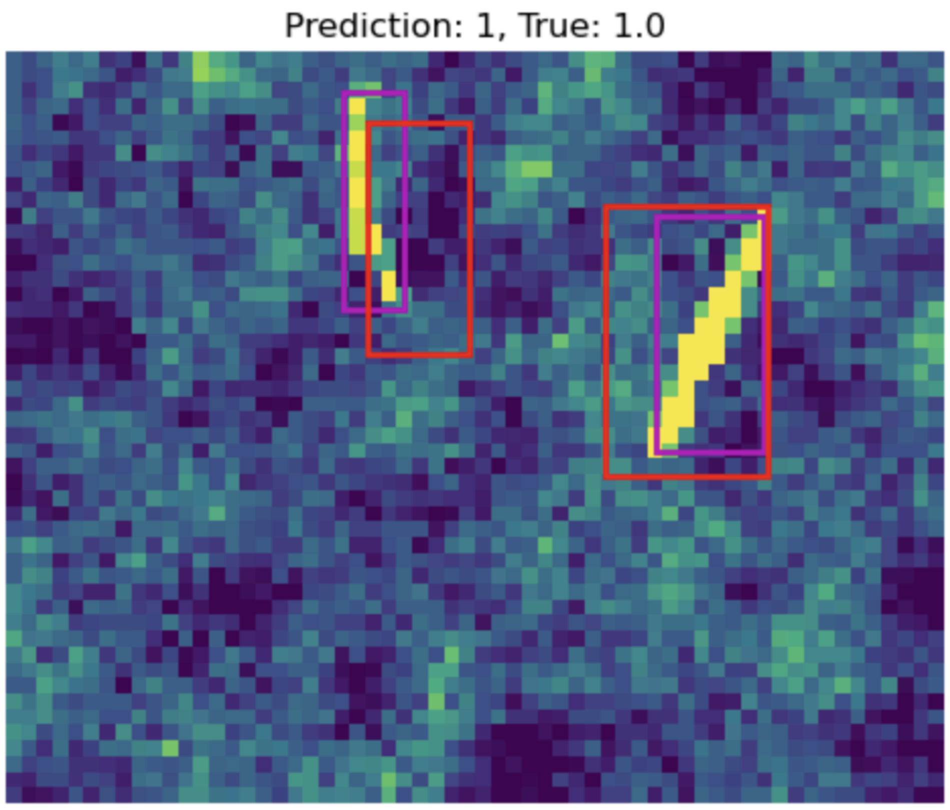
```
Model 1
Val accuracy: 0.9373, Val IoU: 0.7708
Mean of both metrics: 0.8540
Model 2
Val accuracy: 0.9808, Val IoU: 0.9187
Mean of both metrics: 0.9498
Model 3
Val accuracy: 0.9809, Val IoU: 0.8513
Mean of both metrics: 0.9161
```

Results:

After the training the model that gave us the **best results** was **Model 2** and had an **accuracy of 0.9807** and a mean of the **intersection over union of 0.9177**. The **mean of both metrics is 0.9492** (!).

We will now plot the results here for the better visualization of them.


Prediction: 0, True: 0.0


Prediction: 1, True: 1.0