# f1dotatvt

May 13, 2025

#### 0.0.1 Project Submission

- Student Name: Andrew Reusche
- Student Pace: Self Paced
- Instructor: Mark Barbor

# 1 Automating License Plate Detection with Convolutional Neural Networks and Bounding Box Regression

Author: Andrew Reusche

## 1.1 Project Summary

**Business and Data Understanding** My project aims to automate license plate detection as the first step in a computer vision based toll collection system. Currently license plates must be manually identified and cropped before being passed to OCR. My goal is to replace this manual step with a supervised learning model that can localize license plates accurately and consistently. I used the License Plate Detection dataset from Kaggle, which contains ~1,800 plate labeled vehicle images (mostly cars and vans with non-U.S. plates). These images vary in angle and distance making them well suited for training a bounding box regression model.

**Data Preparation** Using PyTorch I created a custom dataset class to load YOLO formateed bounding boxes and apply image transformations. The dataset was split into training, validation, and a 10% holdout test set. I applied data augmentation (color jitter, random flipping, rotation) on the training set using torchvision.transforms, and normalized all images using ImageNet mean/std values to match ResNet expectations. My validation and test sets were only normalized to simulate real deployment conditions.

**Modeling** I used PyTorch for modeling, testing several CNNs before moving on to a pretrained ResNet18 backbone. I explored multiple loss functions (nn.MSE, GIoU, DIoU) and ran a grid search to tune learning rate, weight decay, and dropout. My best performing model uses DIoU loss, ReNet18, and tuned hyperparameters.

**Evaluation** I ran my best model on the testing holdout set to simulate the model's effectiveness on new images of vehicles that drive through the tolls and it achieved a mean IoU of 0.7727, meaning my predicted bounding boxes overlap closely with the ground truth boxes 77% of the time. While some imperfect predictions remain, the model should be accurate enough to replace manual plate

cropping in most scenarios, greatly reducing manual effort and enabling the OCR team to operate with high confidence in the input region.

## 1.2 Business Problem

Can we use computer vision to automatically detect license plates as cars drive through toll booths?

Right now, my toll company's collection system relies on either EZ-Pass or manual toll booth workers to record and process vehicles. This approach is expensive to maintain, creates a safety risk for staff, and can slow traffic (especially during rush hour). It also leaves room for human error (misread/ cropped license plate, forgotten EZ-Pass, or a driver forgot their cash/card).

To solve these problems and reduce operations cost, my company is building an automated toll collection pipeline powered by computer vision. The idea is to install cameras at each toll booth and use a model to detect the location of a license plate in each image as vehicles pass through (no human input required). Once the plate is located, it will be cropped and passed to and OCR (optical character recognition) system that extracts the actual license plate number. That plate number can then be matched to a database for automatic plate billing.

My team is responsible for building the first part of that computer vision pipeline: a model that can draw accurate bounding boxes around license plates. Once complete, our output will then feed directly into the next team's OCR model, minimizing error/ enabling a faster turnover/ costing less to enact/ and opening the possibility to grow the business model.

### 1.2.1 Metric of Success

My manager has stated that since there will always be a license plate present in the photos fed through my model, the goal of my model is not to analize if there is or is not a license plate present, but instead to predict where the plates are located in the vehicle images via bounding box regression, and the best way to train my model to do this is measure how close my license plate model predictions are to the actual (ground truth) locations of the license plates.

To evaluate this I will use a computer vision bounding box metric called Intersection over Union (IoU). IoU compares my predicted box to the ground truth box by measuring the area of overlap divided by the total area covered by both boxes. Here, the higher the IoU score the better, with a score of 1.0 meaning a perfect predicted to ground truth bounding box (license plate area) match, and a score of 0.0 meaning there was no overlap and the prediction was way off.

Because I want my model to have as accurate predictions as possible (our OCR team will be relying on these predictions to extract the plate numbers), I will use Mean IoU as my metric of success as it measures the average IoU score across all of the predictions that were made by the model.

For more information on the IoU score and how it is calculated please check out this article by Adrian Rosebrock, PhD on pyimagesearch.com: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

## 1.3 Data Understanding

### 1.3.1 Data Source and Data Use

Source: "License Plate Dataset" by Ronak Gohil, Kaggle, https://www.kaggle.com/datasets/ronakgohil/license-plate-dataset

To train and evaluate my license plate detection model I'll be using a dataset from Kaggle that contains real-world car images alongside YOLO-formatted text files with bounding box coordinates for each license plate.

This dataset contains 1,695 pictures of different sized vehicles with license plates taken from different angles/distances, and each one of those pictures is paired with its own set of bounding box coordinates to show where the vehicle's license plate is located in the picture.

From this I will use the following file directories. "images/train" and "labels/train" which I use for model training training and an internal train-test split (1,373 image/coordinate pairs to train the model). "images/val" and "labels/val" which I use as a validation set during training (159 image/coordinate pairs). "images/test" and "labels/test" which I split off from the original training set to act as a final holdout set to evaluate my final model (153 image/coordinate pairs).

Each .jpg image in the dataset comes with an associated .txt file containing one line of YOLO-style annotations detailing the ground truth license plate's object class (license_plate), and normalized bounding box values (x_center, y_center, width, height).

I will use this data to teach my CNN model where the license plate is likely to be in an image by learning from examples of where the license plates are already known to be.

### 1.3.2 Data Limitations

There are some limitations to this dataset that I would like to note:

1. **Only One License Plate Per Image**: Each of these pictures only contains 1 license plate per image, possibly making the detection simpler than if multiple vehicles were present in the real life photo taken at the toll location.
2. **Well Lit Plates**: All of the photos in the dataset are fairly well lit and are taken during clear weather, not representing what the cameras may see at nighttime or during different weather conditions like rain or snow that could obstruct the view of the license plate.
3. **Lack of Vehicle Type Variety**: 99.9% of the vehicles present in the dataset are cars or SUV's potentially leading to a lack of plate recognition when motorcycles/trucks/other vehicles pass through the tolls.
4. **License Plate Format Bias**: The plates shown in the dataset are only from a subset of countries, and with plate designs varying region to region this may make it more difficult to deploy this model in countries like the United States where their plate formats were not present in the training dataset.
5. **Smaller Dataset**: After preprocessing and splitting the dataset, my model will only be training on just under 1,400 images, potentially limiting my CNN's ability to successfully learn the plate location patterns without assistance from data augmentation or pretrained weights via transfer learning.

## 1.4 Bring In The Data and Preview It

Before I can train my model to detect license plates I need to pull in a dataset to train my model.

**Reproducibility Note:** My project is being run inside a Google Colab notebook on the A100 hardware accelerator to minimize processing time. Google Colab comes with many popular python and machine learning packages pre installed, but some packages may need to be downloaded or set up externally to recreate this notebook's experiment.

Most directly, because I am downloading my data directly from Kaggle, you will need a Kaggle account with an API access code to follow along.

For help setting up a Kaggle API key please create a Kaggle account and go here: https://www.kaggle.com/docs/api

Set a random seed for reproducibility

[1]:
```
#import relevant libraries
import os
import random
import numpy as np
import torch

#set the seed so that the model training and data splitting is reproducable
def set_seed(seed= 24):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)

set_seed(24)
```

Authenticate with Kaggle and download the dataset

[2]:
```
#create Kaggle folder and JSON credentials file
!mkdir ~/.kaggle
!touch ~/.kaggle/kaggle.json

#manually enter Kaggle username here
username = 'username'
#manually enter Kaggle API key here
api_key = 'api key'

"""
Please delete your Kaggle API key from this field before uploading
or sharing this notebook.
"""

#create dictionary and save it as kaggle.json so the API can authenticate
import json
api_token = {"username":username,"key":api_key}

with open('/root/.kaggle/kaggle.json', 'w') as file:
    json.dump(api_token, file)

#adjust permissiona to protect the key file
!chmod 600 ~/.kaggle/kaggle.json
```

Download and unzip the dataset

[3]:
```
#pull down the dataset directly from Kaggle
!kaggle datasets download -d ronakgohil/license-plate-dataset

"""
Source: "License Plate Dataset" by Ronak Gohil, Kaggle,
https://www.kaggle.com/datasets/ronakgohil/license-plate-dataset
"""
```

```
Dataset URL: https://www.kaggle.com/datasets/ronakgohil/license-plate-dataset
License(s): CC0-1.0
```

[3]:
```
'\nSource: "License Plate Dataset" by Ronak Gohil,
Kaggle,\nhttps://www.kaggle.com/datasets/ronakgohil/license-plate-dataset\n'
```

[4]:
```
#unzip and download the dataset into a folder named "data"
!unzip -q license-plate-dataset.zip -d data
```

Import the rest of the libraries I will need

[5]:
```
"""
If you are having difficulty importing these, such as in an environment
outside of Google Colabs, you may need to run the following line of
code first:

!pip install kaggle tqdm opencv-python pillow
"""

#standard libraries
import os
import shutil
import itertools
from glob import glob

#data manipulation and plotting
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm
import cv2 #OpenCV for image processing

#pytorch core and utilities
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
```

```
#TorchVision libraries for image transforms and pretrained models
import torchvision
from torchvision import transforms
import torchvision.models as models

#Sklearn for data splitting
from sklearn.model_selection import train_test_split
```

### 1.4.1 Preview the Data (License Plate Images with Bounding Boxes)

Now that the data is loaded in I am going to preview it to get a feel for what it looks like and contains.

Each image in the dataset has a corresponding .txt file in YOLO format that contains each license plate's normalized coordinates in the form:

"[class_id] [x_center] [y_center] [width] [height]"

Now I will load in a few of these image label pairs and draw bounding boxes on the images so I can visually confirm what I am dealing with.

[6]:
```
#fix label file paths that end in '.txt.txt' (weird naming problem in dataset)
def clean_label_path(label_path):
    if label_path.endswith('.txt.txt'):
        return label_path.replace('.txt.txt', '.txt')
    return label_path

#function to read and image and draw its YOLO-format bounding box
def show_images_with_boxes(image_path, label_path, return_img= False):
  label_path= clean_label_path(label_path)
  img= cv2.imread(image_path)
  img= cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  h, w, _= img.shape

  with open(label_path, 'r') as f:
    boxes= f.readlines()

  for box in boxes:
    class_id, x_center, y_center, box_w, box_h= map(float, box.strip().split())
    x1= int((x_center - box_w / 2) *w)
    y1= int((y_center - box_h / 2) *h)
    x2= int((x_center + box_w / 2) *w)
    y2= int((y_center + box_h / 2) *h)
    cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)

  if return_img:
    return img
  else:
```

```
    plt.imshow(img)
    plt.axis('off')
    plt.show()
```

Show 3 randomly selected examples from the dataset.

The ground truth license plate bounding boxes will be displayed with a green perimeter throughout my notebook, while the predicted bounding boxes will be displayed with a red perimeter.

[7]:
```
#set the image and label directory
image_dir = 'data/archive/images/train'
label_dir = 'data/archive/labels/train'

#grabs a few sample file names to preview
sample_imgs= sorted(os.listdir(image_dir))[:3]

#loops through the sample files and plots them
for img_file in sample_imgs:
  label_file= os.path.splitext(img_file)[0] + '.txt'
  label_path= os.path.join(label_dir, label_file)
  image_path= os.path.join(image_dir, img_file)

  show_images_with_boxes(image_path, label_path)
```


CarBlogIndia


MH20CS9817

### 1.5 Data Exploration

Now I will do a deeper dive into the dataset. Specifically I want to confirm the following:

- Every image has a matching label file
- The labels are formatted as expected
- There is some reasonable variation in license plate box size (so that I am not just memorizing one pattern)
- The images themselves are consistent in size or need to be resized

I will start by checking how many files are in each folder. This will confirm that all the image-label pairs are intact.

[8]:
```
#count the images in each directory
train_imgs= glob('data/archive/images/train/*.jpg')
train_labels= glob('data/archive/labels/train/*.txt')
val_imgs= glob('data/archive/images/val/*.jpg')
val_labels= glob('data/archive/labels/val/*.txt')

print('Train images:', len(train_imgs))
print('Train labels:', len(train_labels))
print('Validation images:', len(val_imgs))
print('Validation labels:', len(val_labels))
```

```
Train images: 1526
Train labels: 1526
Validation images: 169
Validation labels: 169
```

I will now take this one step further by confirming that every image has a matching .txt label file with bounding box coordinates.

[9]:
```
#make sure every image has a matching label file
def check_missing_labels(img_paths, label_dir):
    missing= []
    for img_path in img_paths:
        base= os.path.splitext(os.path.basename(img_path))[0]
        label_path= os.path.join(label_dir, base + ".txt")
        if not os.path.exists(label_path):
            missing.append(img_path)
    return missing

missing_train= check_missing_labels(train_imgs, 'data/archive/labels/train')
missing_val= check_missing_labels(val_imgs, 'data/archive/labels/val')

print(f"Missing labels in training set: {len(missing_train)}")
print(f"Missing labels in validation set: {len(missing_val)}")
```

```
Missing labels in training set: 0
Missing labels in validation set: 0
```

[10]:
```
#check out the class distribution to confirm they are all "license plate"
with open("data/archive/classes.txt", "r") as f:
    classes= f.read().splitlines()

print(f"Classes ({len(classes)}): {classes}")
```

```
Classes (1): ['license_plate']
```

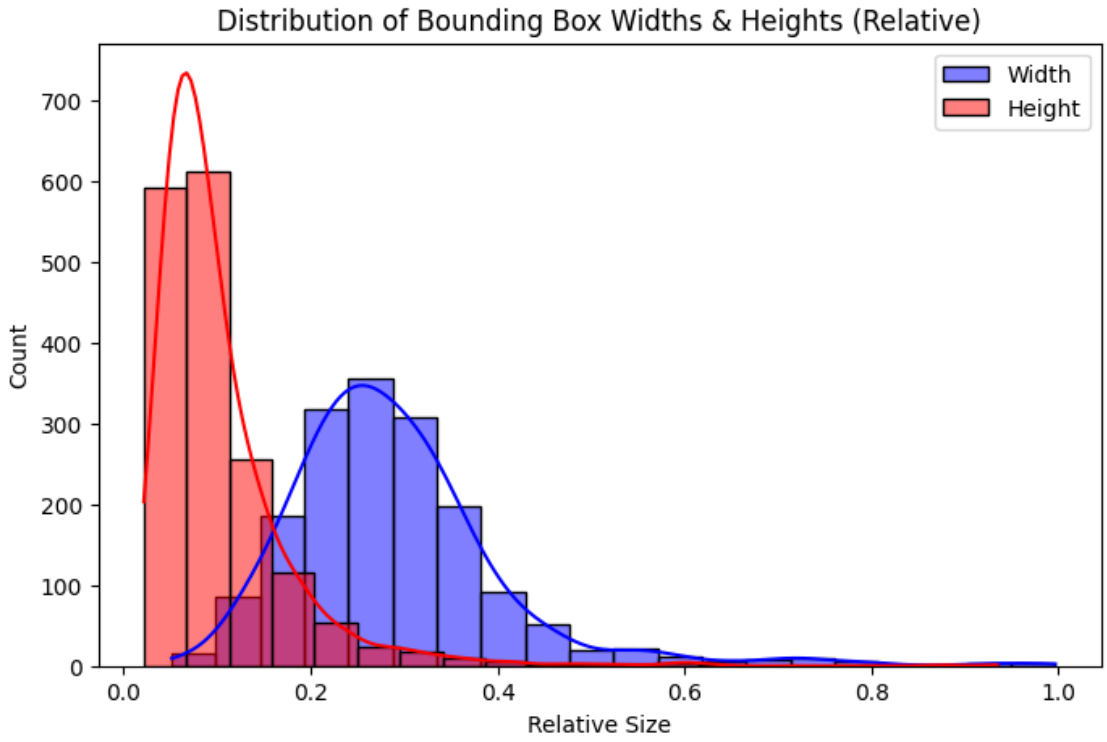
### 1.5.1 Bounding Box Size Distribution

Before resizing the images or building the model, I want to inspect the distribution of the license plate sizes in each picture. Since the label files are in YOLO format (normalized between 0 and 1), I can directly explore the relative sizes of each license plate bounding box.

The cells below loop through all the .txt files in both the training and validation sets to extract the width and height of each license plate bounding box. These values are then plotted so we can visualize the spread.

[11]:
```
#function to extract bounding box widths and heights
def get_bbox_dims(label_files):
    widths= []
    heights= []
    for lbl in label_files:
        with open(lbl, 'r') as f:
            for line in f.readlines():
                parts= line.strip().split()
                if len(parts) == 5:
                    _, _, _, w, h= map(float, parts)
                    widths.append(w)
                    heights.append(h)
    return pd.DataFrame({'width': widths, 'height': heights})

#combine the train and validation labels for a full scope of understanding
bbox_df= get_bbox_dims(train_labels + val_labels)
```

[12]:
```
#plot the bounding box size distribution
plt.figure(figsize=(8, 5))
sns.histplot(bbox_df['width'], bins=20, kde=True, color='blue', label='Width')
sns.histplot(bbox_df['height'], bins=20, kde=True, color='red', label='Height')
plt.title("Distribution of Bounding Box Widths & Heights (Relative)")
plt.xlabel("Relative Size")
plt.legend()
plt.show()

#displays the summary statistics
bbox_df.describe()
```




[12]:

| | width | height |
|---|---|---|
| count | 1695.000000 | 1695.000000 |
| mean | 0.288975 | 0.103054 |
| std | 0.117831 | 0.076331 |
| min | 0.051471 | 0.021452 |
| 25% | 0.216295 | 0.057851 |
| 50% | 0.272059 | 0.082589 |
| 75% | 0.337444 | 0.123077 |
| max | 0.996667 | 0.934483 |

I am able to tell that both the widths and the heights tend to cluster around the same sizes but still do have some variation.

Width and height do have a couple of sizes that are stretching up to 1 which may point to some of the license plate pictures being very zoomed in.

Overall I have some decent variety in plate size, so my model will need to learn to handle plates of different sizes and distances.

### 1.5.2 Image Size Distribution

Now that I have looked into the sizes of the license plates, relative to their image sizes, I will also the distribution of the actual image sizes as well.

Widely different image sizes can make model training inconsistent and unstable so we want to see

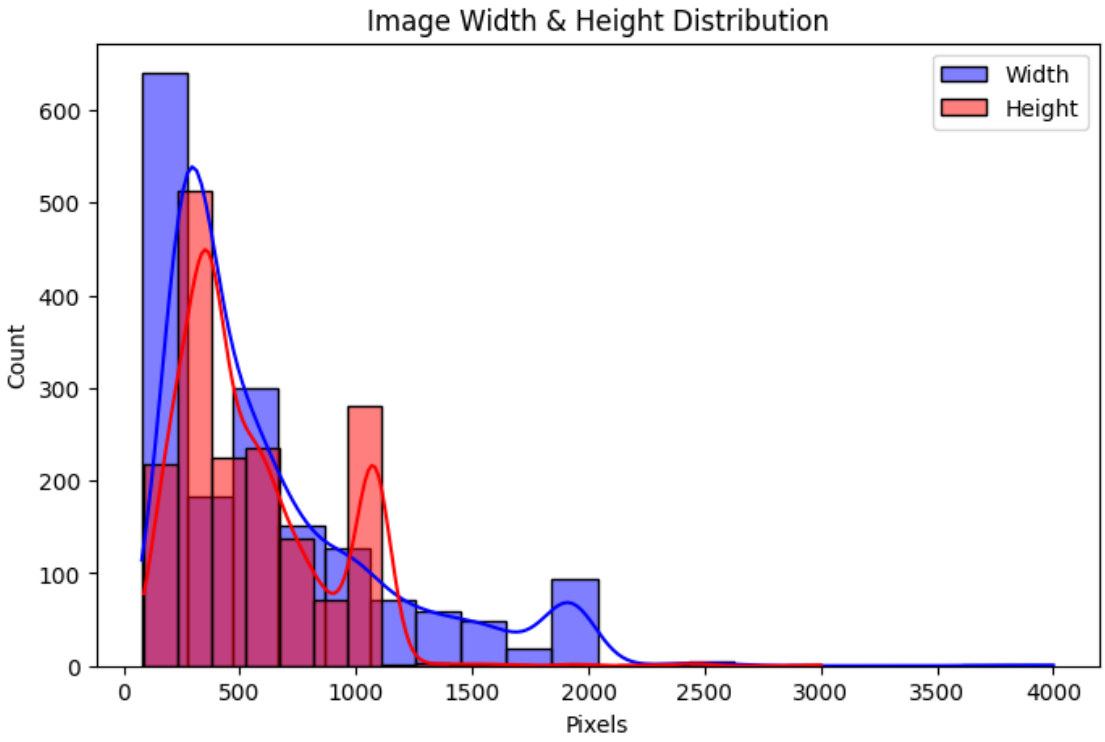
if all the images are relatively the same size.

To do this I loop through every image in the training and validation sets to extract their pixel dimensions (width x height). Then I visualize the size distribution below.

[13]:
```
#function to get width and height for each image
def get_image_sizes(image_paths):
    widths= []
    heights= []
    for path in tqdm(image_paths):
      with Image.open(path) as img:
        w, h= img.size
        widths.append(w)
        heights.append(h)

    return pd.DataFrame({'width': widths, 'height': heights})

#combine all image paths
all_imgs= train_imgs + val_imgs
img_dims_df= get_image_sizes(all_imgs)
```

```
100%|      | 1695/1695 [00:00<00:00, 10732.82it/s]
```

[14]:
```
#plot the image width and height distribution
plt.figure(figsize=(8, 5))
sns.histplot(img_dims_df['width'], bins=20, kde=True, color='blue',␣
 ↪label='Width')
sns.histplot(img_dims_df['height'], bins=20, kde=True, color='red',␣
 ↪label='Height')
plt.title("Image Width & Height Distribution")
plt.xlabel("Pixels")
plt.legend()
plt.show()

img_dims_df.describe()
```




[14]:

| | width | height |
|---|---|---|
| count | 1695.000000 | 1695.000000 |
| mean | 654.546313 | 554.525664 |
| std | 499.585320 | 321.803263 |
| min | 78.000000 | 87.000000 |
| 25% | 272.000000 | 333.000000 |
| 50% | 485.000000 | 451.000000 |
| 75% | 859.500000 | 734.000000 |
| max | 4000.000000 | 3000.000000 |

I am able to tell that there is a lot of variability in image size with some images being over 10x larger than the smaller ones. The range is actually so wide that it could be a problem for training if I don't resize them. In addition to that, not all of the images are square, so if I resize them to a uniform shape (ex 512 x 512 pixels) without padding it would distrot the aspect ratios, warping the bounding boxes.

This confirms I will need to carefully resize my images before training my model. I will take care of this by using padded (letterbox) resizing to preserve the original photo's aspect ratio while still changing all the images to one uniform square pixel size.

### 1.6 Data Preprocessing (Padded Resizing)

Before feeding the images into my model I need to make sure that they are uniform in size. As the above graph displayed, my dataset contains images with a wide variety of dimensions, some small,

some medium, some large, and almost none of them square. This inconsistency would most likely throw off the training of the model.

To handle this I will apply letterbox resizing to all of the images in the dataset. This resizing should achieve the following:

- Scales each image to fit within a square (in this case 512 x 512 pixels)
- preserves the original aspect ratio, so license plates don't get stretched or warped
- Adds gray padding to fill in the empty space so every image ends up square

Since I am resizing the images I will also need update update the bounding box coordinates to reflect the new scaled and padded image dimensions.

### 1.6.1 Set Input and Output File Paths

Here I will define where the original image and label files are located, and also where I want to save the newly processed (padded and resized) images and updated labels.

These folders will be used later to train the model using the standardized 512 x 512 pixel image dimensions.

[15]:
```
#define the input folders (original YOLO dataset)
input_img_dir= "data/archive/images/train"
input_lbl_dir= "data/archive/labels/train"

#define the output folders (for resized images and adjusted labels)
output_img_dir= "data/processed/images"
output_lbl_dir= "data/processed/labels"

#confirm the new output directories exist
os.makedirs(output_img_dir, exist_ok=True)
os.makedirs(output_lbl_dir, exist_ok=True)
```

### 1.6.2 Apply Padded Resizing to All Images and Labels

I will achieve this resizing through three functions:

1. "letterbox_resize()": handles the actual image resizing and padding
2. "process_labels()": updates the bounding box coordinates to match the resized images
3. "preprocess_yolo_dataset()": applies the resizing to a whole folder of image/ label pairs

[16]:
```
#define target output dimensions
target_w= 512
target_h= 512
```

[17]:
```
#function to resize a single image using padded (letterbox) resizing
def letterbox_resize(image, target_width, target_height):
    h, w= image.shape[:2]

    #calculate the resize scale (based on which side is the limiting factor)
    scale= min(target_width / w, target_height / h)
```

```
    new_w= int(w * scale)
    new_h= int(h * scale)

    #resize the image while preserving the aspect ratio
    resized= cv2.resize(image, (new_w, new_h), interpolation=cv2.INTER_LINEAR)

    #calculate how much padding to add to mae the image square
    pad_w= target_width - new_w
    pad_h= target_height - new_h
    top= pad_h // 2
    bottom= pad_h - top
    left= pad_w // 2
    right= pad_w - left

    #add gray padding (Blue= 114, Green= 114, Red= 114) around the resized image
    padded= cv2.copyMakeBorder(resized,
                               top,
                               bottom,
                               left,
                               right,
                               cv2.BORDER_CONSTANT,
                               value=(114, 114, 114))

    return padded, scale, left, top, w, h

#function to update the bounding box coordinates after resizing and padding
def process_labels(label_path, scale, pad_left, pad_top, orig_w, orig_h):
    new_lines= []

    with open(label_path, 'r') as f:
        lines= f.readlines()

    for line in lines:
        class_id, x, y, w, h= map(float, line.strip().split())

        #un-normalize the YOLO box (scale it back up to original pixel␣
 ↪dimensions)
        x *= orig_w
        y *= orig_h
        w *= orig_w
        h *= orig_h

        #apply the resize scaling
        x *= scale
        y *= scale
        w *= scale
        h *= scale
```

```
        #add the padding offsets
        x += pad_left
        y += pad_top

        #re-normalize to new target size
        x /= target_w
        y /= target_h
        w /= target_w
        h /= target_h

        new_lines.append(f"{int(class_id)} {x:.6f} {y:.6f} {w:.6f} {h:.6f}")

    return new_lines

#ffunction to apply padded resize to an entire folder of images and labels
def preprocess_yolo_dataset(input_img_dir,
                            input_lbl_dir,
                            output_img_dir,
                            output_lbl_dir,
                            target_w=512,
                            target_h=512):

    os.makedirs(output_img_dir, exist_ok=True)
    os.makedirs(output_lbl_dir, exist_ok=True)

    #lists all image files in the input folder
    image_files= [f for f in os.listdir(input_img_dir)
                  if f.lower().endswith('.jpg')]

    for img_file in tqdm(image_files, desc=f"Preprocessing {os.path.
 ↪basename(input_img_dir)}"):
        img_path= os.path.join(input_img_dir, img_file)
        label_filename= os.path.splitext(img_file)[0] + '.txt'
        label_path= os.path.join(input_lbl_dir, label_filename)

        image= cv2.imread(img_path)

        #resize and pad the image
        pad_img, scale, pad_left, pad_top, orig_w, orig_h=␣
 ↪letterbox_resize(image,
                                                                  ␣
 ↪target_w,
                                                                  ␣
 ↪target_h)

        #save the resized image
```

```
        out_img_path= os.path.join(output_img_dir, img_file)
        cv2.imwrite(out_img_path, pad_img)

        #if label exists, update its coordinates and save to new path
        if os.path.exists(label_path):
            new_label_lines = process_labels(label_path,
                                             scale,
                                             pad_left,
                                             pad_top,
                                             orig_w,
                                             orig_h)

            out_lbl_path= os.path.join(output_lbl_dir,
                                       label_filename)

            with open(out_lbl_path, 'w') as f:
                f.write('\n'.join(new_label_lines))
```

Now that I have defined the resizing functions and specified the input/output directories I will go ahead and apply the letterbox resizing transformation to both the training and validation subsets of my dataset.

This will standardize all image sizes to 512 x 512 pixels while preserving the original aspect ratios and updating their bounding box coordinates accordingly.

[18]:
```
#run padded resize on the train subset
preprocess_yolo_dataset(input_img_dir= 'data/archive/images/train',
    input_lbl_dir= 'data/archive/labels/train',
    output_img_dir= 'data/processed/images',
    output_lbl_dir= 'data/processed/labels')

#run padded resize on the validation subset
preprocess_yolo_dataset(input_img_dir= 'data/archive/images/val',
    input_lbl_dir= 'data/archive/labels/val',
    output_img_dir= 'data/processed_val/images',
    output_lbl_dir= 'data/processed_val/labels')
```

```
Preprocessing train: 100%|      | 1526/1526 [00:07<00:00, 202.88it/s]
Preprocessing val: 100%|      | 169/169 [00:01<00:00, 125.97it/s]
```

Now that all the images have been resized I want to visually confirm the transformation worked as expected.

To do this I will create a quick function that visually compares the original image to its resized version side by side.

[19]:
```
#create function to show image and bounding box before and after transformation
def show_images_side_by_side(img1_path, lbl1_path, img2_path, lbl2_path):
    img1= show_images_with_boxes(img1_path, lbl1_path, return_img=True)
```

```
    img2= show_images_with_boxes(img2_path, lbl2_path, return_img=True)

    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.imshow(img1)
    plt.title("Original Image")
    plt.axis("off")

    plt.subplot(1, 2, 2)
    plt.imshow(img2)
    plt.title("Processed Image (512x512 padded)")
    plt.axis("off")

    plt.tight_layout()
    plt.show()
```

Now I will choose a sample image to compare a good example of the before and after transformation.

[20]:
```
#grabs a sample image to visualize before and after resizing
image_files= [f for f in os.listdir(input_img_dir) if f.lower().endswith('.
 ↪jpg')]
trans_example= sorted(image_files)[8] #picks a sample

#constructs the label file path
label_filename= os.path.splitext(trans_example)[0] + '.txt'

#displays the bofore and after side by side
show_images_side_by_side(os.path.join(input_img_dir, trans_example),
                         os.path.join(input_lbl_dir, label_filename),
                         os.path.join(output_img_dir, trans_example),
                         os.path.join(output_lbl_dir, label_filename))
```


This confirms the resizing function is working correctly, and the original image (left) and the resized/padded image (right) are as they should be.

The license plate remains clearly visible and unwarped in the resized version, and the bounding box is still wrapped around the plate.

I can now train my model on this preprocessed data.

## 1.7 Model Building and Data Analysis

### 1.7.1 Train-Validation-Test Split

Now that the images are resized and cleaned I need to strucutre the data for model training and evaluation.

I already have a predefined validation set (val) that I will leave untouched. However, the original dataset did not include a separate test set so I will carve out 10% of the resized training set to act as a final test holdout.

This gives me three clean subsets:

- Training Set: 90% of the original training data. Used to train the model.
- Validation Set: roughly the size of 10% of the training set, and pre-existing. Used to tune the model during training.
- Test Set: 10% of the original training set. Only used to evaluate the model as a simulation of unseen data instances.

To ensure this setup is clean and reproducable I organize the final dataset into the following directory structure:

*data/final_split/* which then goes into one of these 3 subdirectories:

*train/*

*val/*

*test/*

Each of these 3 folders contains *images/* and *labels/* subfolders.

My code below performs the data split and copies the relevant image-label pairs to their respective directories.

[21]:
```
#preprocessed data paths
processed_train_img_dir= 'data/processed/images'
processed_train_lbl_dir= 'data/processed/labels'
processed_val_img_dir= 'data/processed_val/images'
processed_val_lbl_dir= 'data/processed_val/labels'

#output split base directory
split_base= 'data/final_split'
os.makedirs(split_base, exist_ok= True)

#get all preprocessed image-label pairs
train_images= sorted([f for f in os.listdir(processed_train_img_dir)
              if f.endswith('.jpg')])
train_pairs= [(f, os.path.splitext(f)[0] + '.txt') for f in train_images]

#train test split
train_split, test_split= train_test_split(train_pairs,
                                          test_size=0.1,
                                          random_state=24)
#copy image-label pairs to split directories
def copy_split(pairs, dest_split, src_img_dir, src_lbl_dir):
    img_out= os.path.join(split_base, dest_split, 'images')
    lbl_out= os.path.join(split_base, dest_split, 'labels')
    os.makedirs(img_out, exist_ok=True)
    os.makedirs(lbl_out, exist_ok=True)

    for img_file, lbl_file in tqdm(pairs, desc=f"Copying {dest_split}"):
        shutil.copy(os.path.join(src_img_dir, img_file),
                    os.path.join(img_out, img_file))
        shutil.copy(os.path.join(src_lbl_dir, lbl_file),
                    os.path.join(lbl_out, lbl_file))

#copy final splits
copy_split(train_split, 'train', processed_train_img_dir,␣
 ↪processed_train_lbl_dir)
copy_split(test_split, 'test', processed_train_img_dir, processed_train_lbl_dir)

#copy preprocessed validation set as-is
val_pairs= [(f, os.path.splitext(f)[0] + '.txt')
            for f in os.listdir(processed_val_img_dir) if f.endswith('.jpg')]
```

```
copy_split(val_pairs, 'val', processed_val_img_dir, processed_val_lbl_dir)

#print the sizes from each split
print("\nTrain size:", len(os.listdir('data/final_split/train/images')))
print("Val size:  ", len(os.listdir('data/final_split/val/images')))
print("Test size: ", len(os.listdir('data/final_split/test/images')))
```

```
Copying train: 100%|      | 1373/1373 [00:00<00:00, 4758.46it/s]
Copying test: 100%|      | 153/153 [00:00<00:00, 4908.40it/s]
Copying val: 100%|      | 169/169 [00:00<00:00, 4963.33it/s]


Train size: 1373
Val size:   169
Test size:  153
```

### 1.7.2 Create Pytorch Dataset and Dataloaders

Now that my dataset is cleaned, resized, and split into training/validation/test sets I will define a custom *LicensePlateDataset* class so PyTorch can load the data in batches.

This class will:

- Read each *.jpg* image and its matching *.txt* label file
- Optionally apply a PyTorch transform (ex *ToTensor*, augmentation, normalization)
- Return the image and its bounding box as a tensor

[22]:
```
#custom PyTorch dataset class to handle my license plate image-label pairs
class LicensePlateDataset(Dataset):
  def __init__(self, image_dir, label_dir, transform= None):
    #store the paths to the image and label directories
    self.image_dir= image_dir
    self.label_dir= label_dir

    #grab all image filenames (only those ending in .jpg) and sort them
    self.image_files= sorted([f for f in os.listdir(image_dir)
                              if f.endswith('.jpg')])
    #optionally apply a torchvision transform
    self.transform= transform

  def __len__(self):
    #returns the total number of images in the dataset
    return len(self.image_files)

  def __getitem__(self, idx):
    #get the image filename based on the index
    img_file= self.image_files[idx]
```

```
    #derive the cooresponding label filename by replacing .jpg with .txt
    label_file= os.path.splitext(img_file)[0] + '.txt'

    #build the full path to the image and its label
    image_path= os.path.join(self.image_dir, img_file)
    label_path= os.path.join(self.label_dir, label_file)

    #load the image using PIL and make sure it's in RGB format
    image= Image.open(image_path).convert('RGB')

    #read the line from the YOLO label file
    with open(label_path, 'r') as f:
      line= f.readline().strip()
      parts= line.split()
      #skip the first element and convert the remaining 4 values to floats
      bbox= torch.tensor(list(map(float, parts[1:])), dtype= torch.float32)
      #gives us a tensor of [x_center, y_center, width, height]

    #if any transforms were specified apply them here
    if self.transform:
      image= self.transform(image)

    #return the transformed image and the bounding box
    return image, bbox
```

[23]:
```
#transform to convert images to tensors
transform= transforms.Compose([transforms.ToTensor()])

#base path to the organized splits
base_path= 'data/final_split'

#instantiate the datasets
train_dataset= LicensePlateDataset(os.path.join(base_path, 'train/images'),
                                   os.path.join(base_path, 'train/labels'),
                                   transform=transform)

val_dataset= LicensePlateDataset(os.path.join(base_path, 'val/images'),
                                 os.path.join(base_path, 'val/labels'),
                                 transform=transform)

test_dataset= LicensePlateDataset(os.path.join(base_path, 'test/images'),
                                  os.path.join(base_path, 'test/labels'),
                                  transform=transform)

#wrap datasets into PyTorch DataLoaders

"""
```

```
I am starting out with a batch size of 32 which is a common default for CNN
training, but this size can be tuned later depending on GPU memory constraints
and convergence speed.
"""

train_loader= DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader= DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader= DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Now I will preview the transformed batches with bounding boxes to makes sure everything worked properly by creating a new function.

This function will:

- grab small batch of images from the DataLoader
- convert the image tensors back into a displayable format
- draw the ground truch bounding box in each image using the label data
- display all the images side by side

This will help me confirm that the image label pairs are still accurate after preproccesing and transformation.

[24]:
```
#function to display a small batch of images with their bounding boxes
def show_batch_with_boxes(dataloader, batch_size= 4):
    #pull a batch from the dataloader
    images, bboxes= next(iter(dataloader))
    images= images[:batch_size]
    bboxes= bboxes[:batch_size]

    #create one row of subplots to display the images
    fig, axs= plt.subplots(1, batch_size, figsize= (15, 5))

    for i in range(batch_size):
        #convert the image tensor to a numpy array in (H, W, C) format
        img= images[i].permute(1, 2, 0).numpy()
        h, w, _= img.shape

        #convert the YOLO format box (center x/y, width, height) to corner␣
 ↪format
        x_center, y_center, box_w, box_h= bboxes[i].tolist()
        x1= int((x_center - box_w / 2) * w)
        y1= int((y_center - box_h / 2) * h)
        x2= int((x_center + box_w / 2) * w)
        y2= int((y_center + box_h / 2) * h)

        #display the image and draw a green rectangle for the bounding box
        axs[i].imshow(img)
        axs[i].add_patch(plt.Rectangle((x1, y1),
                                       x2 - x1,
```

```
                                        y2 - y1,
                                        edgecolor='green',
                                        facecolor='none',
                                        linewidth=2))
        axs[i].axis('off')
        axs[i].set_title(f"Image {i+1}")

    plt.tight_layout()
    plt.show()

#run the preview function on a few images from the training set
show_batch_with_boxes(train_loader, batch_size= 4)
```


Everything seems to have worked out and transformed just fine. I will now move on to the model building.

### 1.7.3 Analysis Functions and Training Loop

Before I begin training my models I need to make a suite of utility functions that will be re-used througout my updating model architectures. These functions will handle model training, evaluation, metric tracking, and visual debugging.

Specifically they will help me answer questions like:

- How well is my model learning (via loss curves)?
- How accurate are the bounding box predictions (via IoU)?
- Are the bounding boxes visually aligned with the license plates?

This function handles training the model across epochs and tracks both training and validation loss. It also evaluates IoU on the validation set and saves the model with with the highest validation IoU.

[25]:
```
#function to train the model while tracking loss/IoU and saving the best version
def train_model2(model,
                 train_loader,
                 val_loader,
                 criterion,
                 optimizer,
```

```
            device,
            epochs=10,
            model_name= 'baseline_model.pth',
            verbose= True):

  #keeps track of the best validation IoU so far
  best_val_iou= -1.0

  #dictionary to store metrics for plotting later
  history= {'train_loss': [], 'val_loss': [], 'val_mean_iou': []}

  #loop through each epoch
  for epoch in range(epochs):
    if verbose:
      print(f"\nEpoch {epoch+1}/{epochs}")

    #training phase

    #set the modle to training mode
    model.train()
    train_losses= []

    #loop through the training batches
    for images, targets in train_loader:
          #move images and labels to the GPU if available
          images, targets= images.to(device), targets.to(device)

          #training steps
          optimizer.zero_grad()
          outputs= model(images)
          loss= criterion(outputs, targets)
          loss.backward()
          optimizer.step()

          #store the loss for this batch
          train_losses.append(loss.item())

    #calculate average training loss for this epoch
    avg_train_loss= np.mean(train_losses)

    #validation phase

    #set the model to evaluation mode
    model.eval()
    val_losses= []
    ious= []
```

```
      #disable gradient calculation to save memory and increase speed
      with torch.no_grad():
            #loop through validation batches
            for images, targets in val_loader:
                images, targets= images.to(device), targets.to(device)

                #run the model
                outputs= model(images)
                loss= criterion(outputs, targets)
                val_losses.append(loss.item())

                #bring the losses back to CPU for IoU calculation
                outputs= outputs.cpu()
                targets= targets.cpu()

                #calculate the IoU for each prediction vs ground truth
                for pred, target in zip(outputs, targets):
                  ious.append(compute_iou(pred, target))

      #average IoU and loss for the validation set
      mean_iou= np.mean(ious)
      avg_val_loss= np.mean(val_losses)

      if verbose:
        print(f"""
Train Loss: {avg_train_loss:.4f}
Val Loss: {avg_val_loss:.4f}
Val IoU {mean_iou: .4f}""")

      #track the loss and IoU over time
      history['train_loss'].append(avg_train_loss)
      history['val_loss'].append(avg_val_loss)
      history['val_mean_iou'].append(mean_iou)

      #check if this is the best model so far, and if so save it
      if mean_iou > best_val_iou:
        best_val_iou= mean_iou
        torch.save(model.state_dict(), model_name)
        if verbose:
          print(f"Saved new best model (Val IoU = {mean_iou:.4f}):␣
 ↪{model_name}")

    #return training history for plotting later
    return history
```

This function plots the training and validation loss curves so I can monitor convergence and detect overfitting.

[26]:
```
#function to plot loss curves
def plot_loss_curves(history):
    plt.figure(figsize=(8, 5))
    plt.plot(history['train_loss'], label='Train Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss Curves')
    plt.legend()
    plt.grid()
    plt.show()
```

This function computes how well the two bounding boxes match by calculating Intersection over Union. It's the main evaluation metric I am using.

The boxes are in YOLO format [x_center, y_center, width, height], so I convert them to corner coordinates and then apply the IoU formula.

[27]:
```
#function to calculate IoU (Intersection over Union) between two bounding boxes
def compute_iou(box1, box2):
  #convert from center format [x_center, y_center, width, height]
  #to corner format [x_min, y_min, x_max, y_max] for box1
  x1_min= box1[0] - box1[2]/ 2
  x1_max= box1[0] + box1[2]/ 2
  y1_min= box1[1] - box1[3]/ 2
  y1_max= box1[1] + box1[3]/ 2

  #do the same for box2
  x2_min= box2[0] - box2[2]/ 2
  x2_max= box2[0] + box2[2]/ 2
  y2_min= box2[1] - box2[3]/ 2
  y2_max= box2[1] + box2[3]/ 2

  #figure out where the two bounding boxes overlap (intersection area)
  inter_xmin= max(x1_min, x2_min)
  inter_ymin= max(y1_min, y2_min)
  inter_xmax= min(x1_max, x2_max)
  inter_ymax= min(y1_max, y2_max)

  #calculate the width and height of the overlapping area
  #clamp at 0 if they don't intersect
  inter_w= max(inter_xmax - inter_xmin, 0)
  inter_h= max(inter_ymax - inter_ymin, 0)
  inter_area= inter_w * inter_h

  #calculate the area of both boxes indavidually
  area1= (x1_max - x1_min) * (y1_max - y1_min)
  area2= (x2_max - x2_min) * (y2_max - y2_min)
```

```
  #calculate the union area (total area covered by both boxes minus the
  #overlapping part)
  union_area= area1 + area2 - inter_area

  #calculate the IoU: overlapped area divinded by total area
  iou= inter_area / union_area if union_area > 0 else 0

  #make sure the IoU stays in a valid range between 0 and 1
  iou= max(0, min(iou, 1))

  return iou
```

This function runs the model on the validation set, calcualtes IoU for each image, and plots a histogram of the results to show how the prediction scores are distributed.

[28]:
```
#function to evaluate the model's bounding box predictions using IoU
def evaluate_iou_with_plot(model, loader, device, verbose= True):
  #set the model to evaluation mode (turn off dropout, and other)
  model.eval()
  ious= []

  #no gradient claculations needed during evaluation
  with torch.no_grad():
    for images, targets in loader:

      #move images and labels to the correct device (GPU or CPU)
      images, targets= images.to(device), targets.to(device)

      #get the model's predicted bounding boxes
      preds= model(images)

      #clone and clamp the outputs to make sure the boxes are in a valid range
      preds= preds.clone()
      preds[:, 2:]= preds[:, 2:].clamp(min= 1e-6, max= 1.0)
      preds[:, :2]= preds[:, :2].clamp(min= 0.0, max= 1.0)

      #move predictions and grouynd truths back to cpu for processing
      preds= preds.cpu()
      targets= targets.cpu()

      #loop through each predicted/true pair and calculate IoU
      for pred, target in zip(preds, targets):
        ious.append(compute_iou(pred, target))

  #calculate mean IoU across the entire dataset
  mean_iou= np.mean(ious)
```

```
#print the average IoU score for the full validation set
print(f"Mean IoU: {mean_iou:.4f}")

#plot iou histogram showing the distrobution of IoU scores
plt.figure(figsize=(8, 5))
plt.hist(ious, bins=20, color='skyblue', edgecolor='black')
plt.title("Distribution of IoU scores on Validation Set")
plt.xlabel("IoU Score")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

This function displays side by side comparisons of predicted (red) and ground truth (green) bounding boxes on a batch of images from the validation set.

[29]:
```
#function to visualize validation predictions vs ground truth bounding boxes
def show_predictions(model, dataloader, device, num_images= 4):
  #set model to eval mode
  model.eval()

  #grab one batch of images and labels from the dataloader
  images, targets= next(iter(dataloader))

  #mvoe both images and ground truth boxes to GPU is available
  images, targets= images.to(device), targets.to(device)

  #run the model to get predicted bounding boxes
  preds= model(images)

  #bring all tensors back to CPU to plot them
  images= images.cpu()
  preds= preds.cpu()
  targets= targets.cpu()

  #create a row of subplots to display the chosen number of images
  fig, axes= plt.subplots(1, num_images, figsize= (15, 5))

  for i in range(num_images):
    #convert the image tensor from (C, H, W) to (H, W, C) for plotting
    img= images[i].permute(1, 2, 0).numpy()
    h, w, _= img.shape

    #model's predicted bounding box
    pred_box= preds[i]
    #ground truth bounding box
    gt_box= targets[i]
```

```
    #convert the predicted box from YOLO format to corner format
    px1= int((pred_box[0] - pred_box[2] / 2) * w)
    py1= int((pred_box[1] - pred_box[3] / 2) * h)
    px2= int((pred_box[0] + pred_box[2] / 2) * w)
    py2= int((pred_box[1] + pred_box[3] / 2) * h)

    #convert ground truth box from YOLO to corner format
    gx1= int((gt_box[0] - gt_box[2] / 2) * w)
    gy1= int((gt_box[1] - gt_box[3] / 2) * h)
    gx2= int((gt_box[0] + gt_box[2] / 2) * w)
    gy2= int((gt_box[1] + gt_box[3] / 2) * h)

    #display the image and draw boxes
    axes[i].imshow(img)

    #red= predicted bounding box
    axes[i].add_patch(plt.Rectangle((px1, py1),
                                    px2- px1,
                                    py2- py1,
                                    edgecolor= 'red',
                                    facecolor= 'none',
                                    linewidth= 2))

    #green= ground truth bounding box
    axes[i].add_patch(plt.Rectangle((gx1, gy1),
                                    gx2- gx1,
                                    gy2- gy1,
                                    edgecolor= 'green',
                                    facecolor= 'none',
                                    linewidth= 2))
    #clean up the axes
    axes[i].axis('off')

  #adjust spacing and show the image grid
  plt.tight_layout()
  plt.show()
```

### 1.7.4 Baseline Model: SimpleBBoxCNN

To start things off I am building a very simple convolutionsal neural network from scratch that will act as my baseline model for bounding box prediction. This version will help set expectations and give me a reference point to improve from.

Model architecture

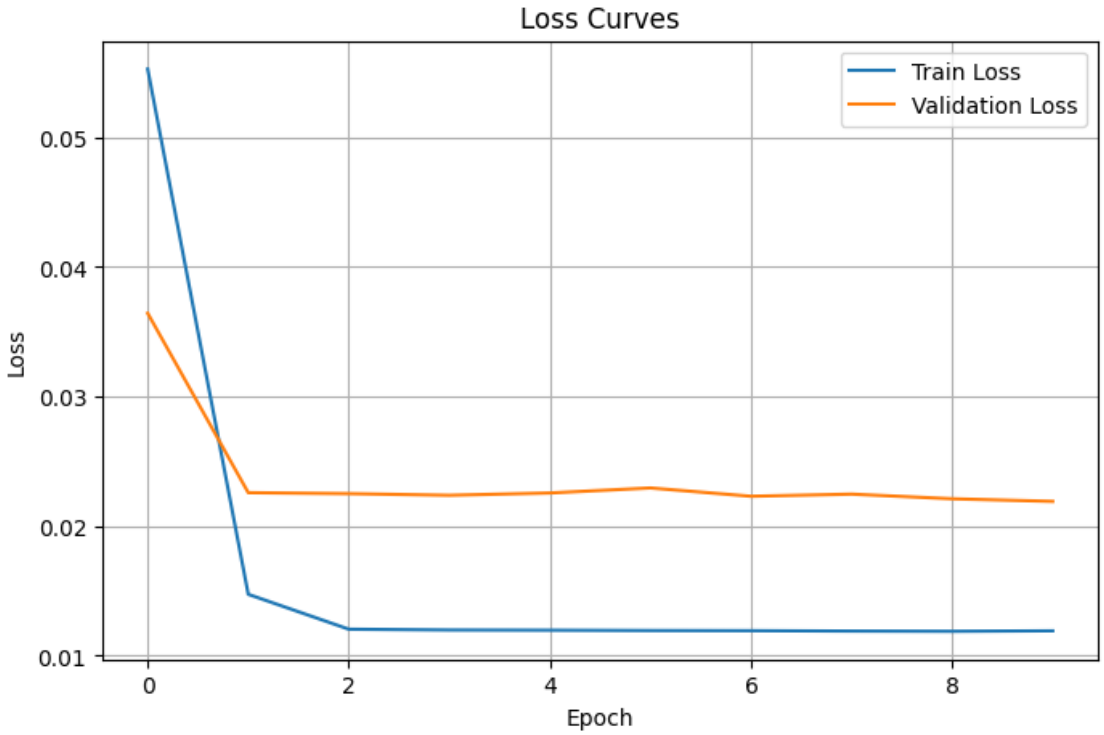
[30]:
```
#create a simple baseline cnn modle to predict bounding box coordinates
class SimpleBBoxCNN(nn.Module):
```

```
def __init__(self):
  super(SimpleBBoxCNN, self).__init__()
  #first conv layer: imput image hgas 3 channels (RGB), output has 16 channels
  self.conv1= nn.Conv2d(3, 16, kernel_size= 3, stride= 2, padding= 1)
  #second conv layer: increases to 32 channels, continues to downsample
  self.conv2= nn.Conv2d(16, 32, kernel_size= 3, stride= 2, padding= 1)
  # third conv layer: 32 > 64 channels, more downsampling
  self.conv3= nn.Conv2d(32, 64, kernel_size= 3, stride= 2, padding= 1)
  #foruth conv layer: 64 > 128 layers, final downsampling
  self.conv4= nn.Conv2d(64, 128, kernel_size= 3, stride= 2, padding= 1)

  #global average pooling compresses spatials dimensions to 1x1
  self.pool= nn.AdaptiveAvgPool2d((1, 1))

  #final linear layer outputs 4 numbers: [x_center, y_center, width, height]
  self.fc= nn.Linear(128, 4)

def forward(self, x):
  #pass input through conv layers with Relu activations
  x= F.relu(self.conv1(x))
  x= F.relu(self.conv2(x))
  x= F.relu(self.conv3(x))
  x= F.relu(self.conv4(x))

  #apply global average pooling
  x= self.pool(x)

  #flatten the fully conected layer
  x= x.view(x.size(0), -1)

  #output the predicted bounding box
  x= self.fc(x)

  #apply sigmoid to keep outputs in 0-1 range since coordinates are normalized
  x= torch.sigmoid(x)
  return x
```

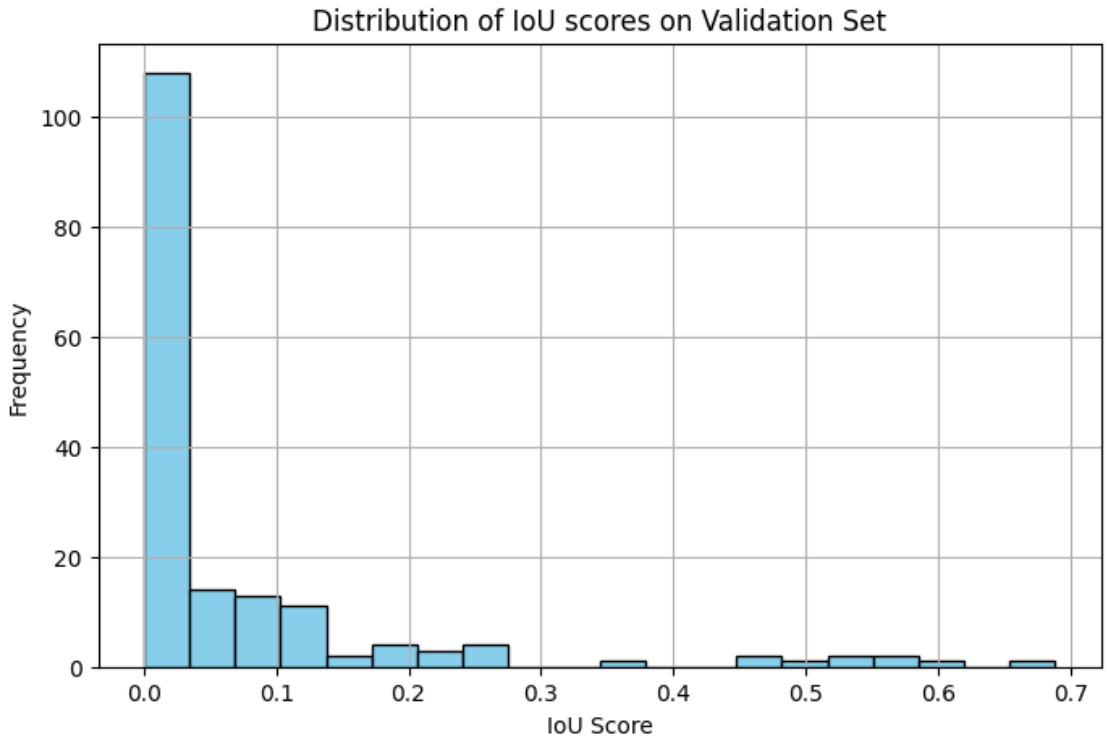
Training Baseline Model

[31]:
```
#set the device to GPU is availalable
device= torch.device("cuda" if torch.cuda.is_available() else "cpu")
#instantiate the baseline model and move it to the chosen device
baseline_model= SimpleBBoxCNN().to(device)
#use Mean Squared Error (MSE) loss for bounding box regression
criterion= nn.MSELoss()
#use Adam optimizer with a low learning rate to help stabilize training
optimizer= optim.Adam(baseline_model.parameters(), lr= 0.0001)
```

```
#train the model
history= train_model2(baseline_model,
                      train_loader,
                      val_loader,
                      criterion,
                      optimizer,
                      device,
                      epochs= 10, #keep the training short
                      model_name= 'baseline_model.pth',
                      verbose= False) #supress print statements

#plot training vs validation loss to see how well the model is converging
plot_loss_curves(history)

#load best model (highest val IoU)
baseline_model.load_state_dict(torch.load('baseline_model.pth'))
baseline_model.to(device)

#visualize some predictions vs ground truth
show_predictions(baseline_model, val_loader, device, num_images= 4)

#evaluate how well the predicted boxes overlap with the ground truth
evaluate_iou_with_plot(baseline_model, val_loader, device, verbose= False)
```



```
Mean IoU: 0.0698
```




Loss Curve: The model does show some reduction in both training and validation across the 10 epochs, but it is relatively shallow. There is no strong sign of overfitting, but also no strong sign of real learning.

Predicted Boxes: It is pretty clear that the predictions are off. Many boxes do not overlap/ are too large, or just barely clip the actual the ground tuth box. From the samples I can see that most predicted boxes are just centered in the image which may suggest the model is just sticking to a default pattern rather than learning the actual bounding box shapes and locations.

Mean IoU score: 0.0698 this confirms what I saw in the samples and score distribution. The

predictions are poor. This may be due to: a simple model (no pretrained features or deep layers), no data augmentation, a short training period, or a basic loss function (MSE) that doesn't optimize for IoU.

I will now try to improve on this with my second model.

### 1.7.5 Deeper Model

Now that I have a baseline in place, I will build a deper CNN architecture to see if more layers and a better setup helps the model learn better and imporve bouding box accuracy.

Compared the my Baseline model the new architecture makes several improvements:

- More convolutional layers: 5 conv blocks instead of 4, with increasing depth all the way up to 512 channels should allow the model to extract richer and more detailed spatial features from the image.
- Batch normalization after each conv layer should help stabilize training and accelerate convergence by normalizing the output of each layer.
- Max pooling is used after each block instead of relying solely on strided convolutions. This should reduce spatial dimensions while keeping the most important activations.
- Dropout feature is added (but not used) before the final layer to reduce overfitting and encourage generalization
- The final global pooling + fully connected layer is retained, and the output is still normalized with a sigmoid activation so the coordinates stay in the 0-1 range

My hope is that these changes will help the model move past its previous center guessing behaviour and actually start locking onto real plate shapes and positions.

Model architecture

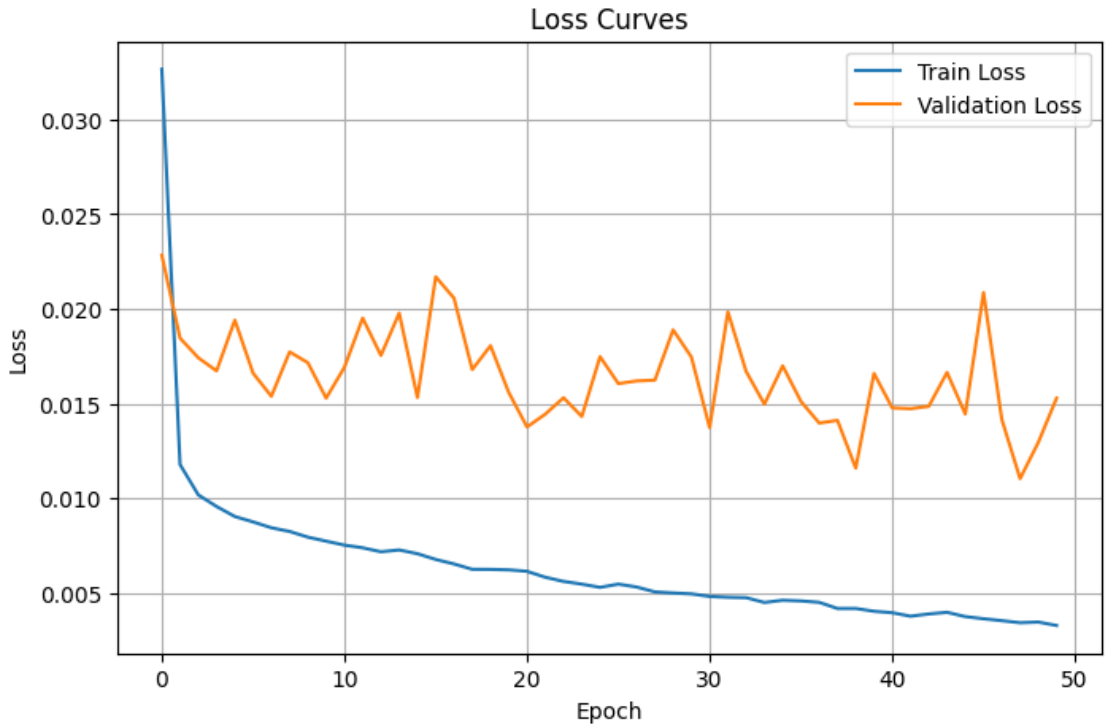
[32]:
```
class DeeperModel(nn.Module):
    def __init__(self, dropout_rate=0.0):
        super(DeeperModel, self).__init__()

        #five convolution blocks, each followed by BatchNorm and ReLU
        self.conv1= nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.bn1= nn.BatchNorm2d(32)
        self.conv2= nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2= nn.BatchNorm2d(64)
        self.conv3= nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3= nn.BatchNorm2d(128)
        self.conv4= nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.bn4= nn.BatchNorm2d(256)
        self.conv5= nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.bn5= nn.BatchNorm2d(512)

        #global average pooling and dropout before the final regression layer
        self.pool= nn.AdaptiveAvgPool2d((1, 1))
        self.dropout= nn.Dropout(dropout_rate)
        #output: [x_center, y_center, width, height]
```

```
        self.fc= nn.Linear(512, 4)

    def forward(self, x):
        x= F.relu(self.bn1(self.conv1(x)))
        x= F.max_pool2d(x, 2)
        x= F.relu(self.bn2(self.conv2(x)))
        x= F.max_pool2d(x, 2)
        x= F.relu(self.bn3(self.conv3(x)))
        x= F.max_pool2d(x, 2)
        x= F.relu(self.bn4(self.conv4(x)))
        x= F.max_pool2d(x, 2)
        x= F.relu(self.bn5(self.conv5(x)))
        x= self.pool(x)

        #flatten and predict
        x= x.view(x.size(0), -1)
        x= self.dropout(x)
        x= self.fc(x)

        #normalize coordinates between 0-1
        x= torch.sigmoid(x)
        return x
```

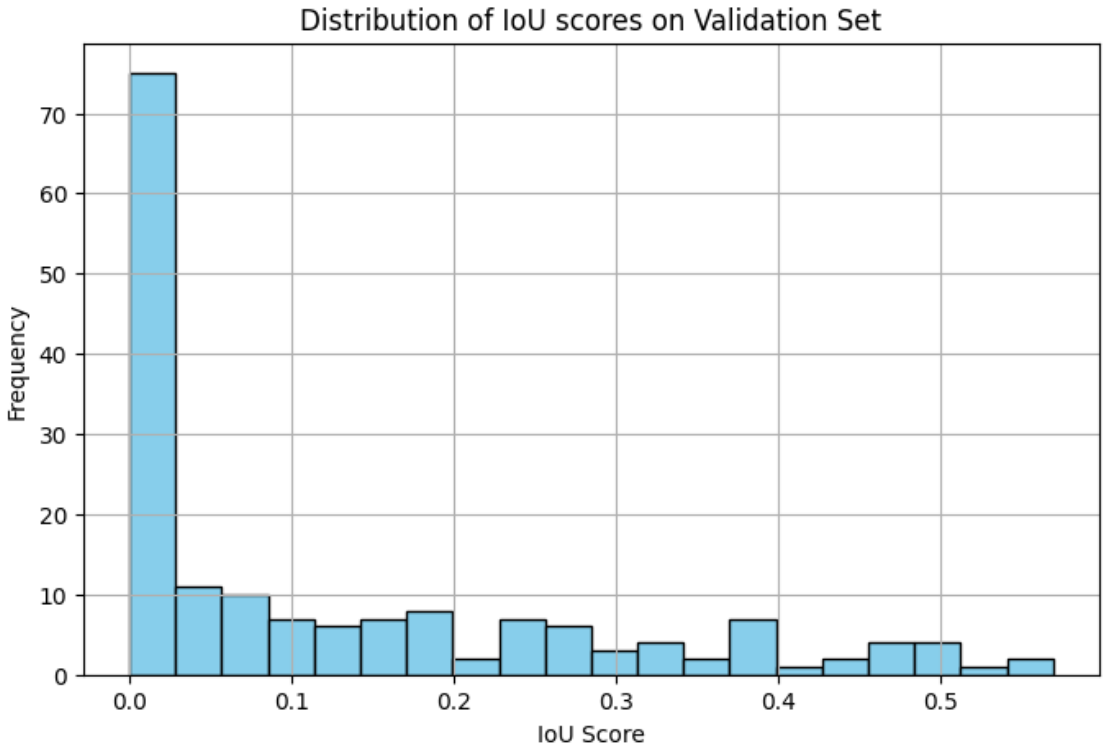
Train and evaluate the model

[33]:
```
#initialize model, loss and optimizer
deeper_model= DeeperModel().to(device)
optimizer= torch.optim.AdamW(deeper_model.parameters(),
                             lr= 0.0001,
                             weight_decay= 0.0)
criterion= nn.MSELoss()

#train for 50 epochs
history_deeper= train_model2(deeper_model,
                                train_loader,
                                val_loader,
                                criterion,
                                optimizer,
                                device,
                                epochs= 50,
                                model_name= 'deeper_model.pth',
                                verbose= False)

#plot loss curves
plot_loss_curves(history_deeper)

#load best model and evaluate
```

```
deeper_model.load_state_dict(torch.load('deeper_model.pth'))
deeper_model.to(device)
show_predictions(deeper_model, val_loader, device, num_images= 4)
evaluate_iou_with_plot(deeper_model, val_loader, device, verbose= False)
```




```
Mean IoU: 0.1272
```




Loss Curve: Ther is a downward trend in both training and validation loss, which is a good. This model appears to be learning more effectively than the baseline model, but is still in need of improvement.

Predicted Boxes: Some of the predicted boxes are getting closer to the ground truth, but many are still off in terms of size and placement. The main improvement is that the boxes are no longer just in the center of the picture at the same size. More improvement needed.

Mean IoU score: 0.1272 is definitely an improvement over the baseline's 0.0698. While the scores still cluster around 0, there is a visible bump in the distribution pointing to some predictions improving from the previous model.

This model shows solid progress with deeper layers, batch normalization, and better pooling helping the model learn more meaningful features. The IoU is still very low so I may need to make a more complex model to start to see real improvements.

#### 1.7.6 Third Model: Enhanced Convolutional Depth

Now that my DeeperModel has shown a measurable improvement over the baseline I want to see if stacking multiple convolutional layers per block willl help my model learn even more detailed spatial features. This model's goal is to build a more expressive architecture that can better lock onto the plate shapes and positions.

Compared to my previous DeeperModel, this third model adds several upgrades:

- Double Convolution Blocks Per Stage: instead of one conv layer per block, each block now has

two. This should give my model more room to extract meaningful features at each resolution level.

- Leaky ReLU Activations: These are used instead of regular ReLU to avoid the "dying ReLU" problem (outputting 0 and freezing weights when inputs stay negative) and helps gradients flow better through the network.

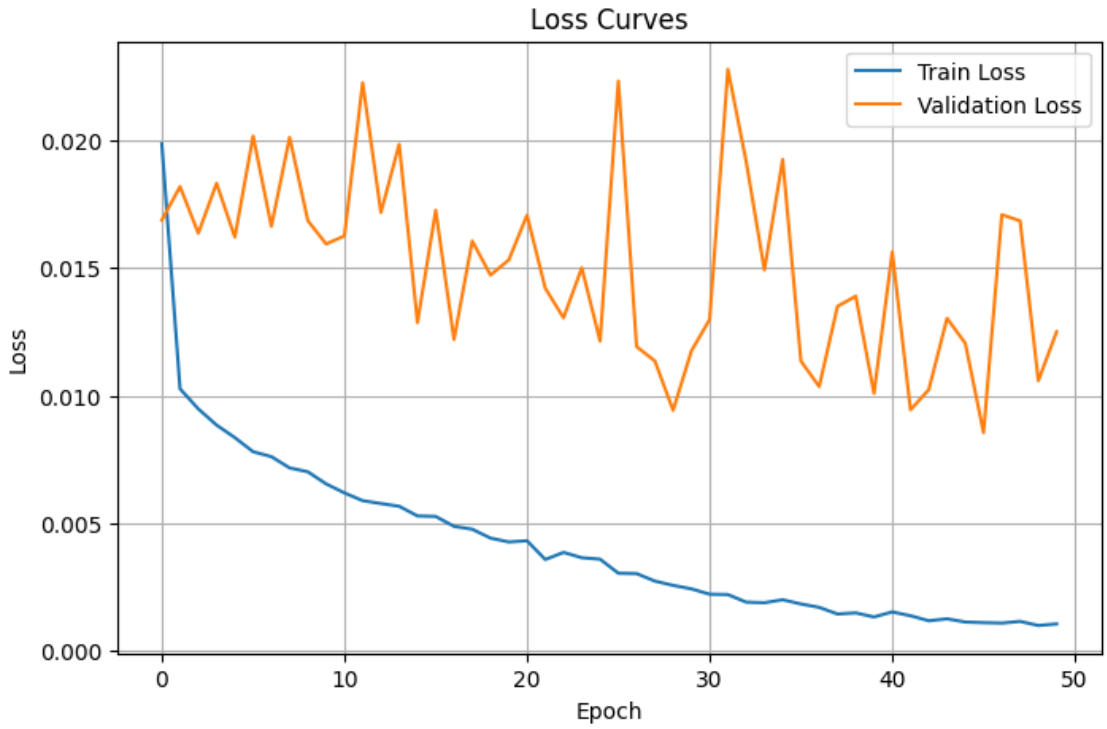
The hope is that these upgrades give the model a better ability to learn localized patterns like the edges/layout of license plates.

[34]:
```
#CNN model with double conv layers per block and LeakyReLU activation
class ThirdModel(nn.Module):
  def __init__(self, dropout_rate= 0.0):
    super(ThirdModel, self).__init__()

    #block 1: two conv layers with 32 filters
    self.conv1= nn.Conv2d(3, 32, kernel_size= 3, padding= 1)
    self.bn1= nn.BatchNorm2d(32)
    self.conv1b= nn.Conv2d(32, 32, kernel_size= 3, padding= 1)
    self.bn1b= nn.BatchNorm2d(32)

    #block 2: two conv layers with 64 filters
    self.conv2= nn.Conv2d(32, 64, kernel_size= 3, padding= 1)
    self.bn2= nn.BatchNorm2d(64)
    self.conv2b= nn.Conv2d(64, 64, kernel_size= 3, padding= 1)
    self.bn2b= nn.BatchNorm2d(64)

    #block 3: two conv layers with 128 filters
    self.conv3= nn.Conv2d(64, 128, kernel_size= 3, padding= 1)
    self.bn3= nn.BatchNorm2d(128)
    self.conv3b= nn.Conv2d(128, 128, kernel_size= 3, padding= 1)
    self.bn3b= nn.BatchNorm2d(128)

    #block 3: two conv layers with 256 filters
    self.conv4= nn.Conv2d(128, 256, kernel_size= 3, padding= 1)
    self.bn4= nn.BatchNorm2d(256)
    self.conv4b= nn.Conv2d(256, 256, kernel_size= 3, padding= 1)
    self.bn4b= nn.BatchNorm2d(256)

    #block 4: two conv layers with 512 filters
    self.conv5= nn.Conv2d(256, 512, kernel_size= 3, padding= 1)
    self.bn5= nn.BatchNorm2d(512)
    self.conv5b= nn.Conv2d(512, 512, kernel_size= 3, padding= 1)
    self.bn5b= nn.BatchNorm2d(512)

    #global pooling + unused dropout + fully connected regression layer
    self.pool= nn.AdaptiveAvgPool2d((1, 1))
    self.dropout= nn.Dropout(dropout_rate)
    self.fc= nn.Linear(512, 4)
```

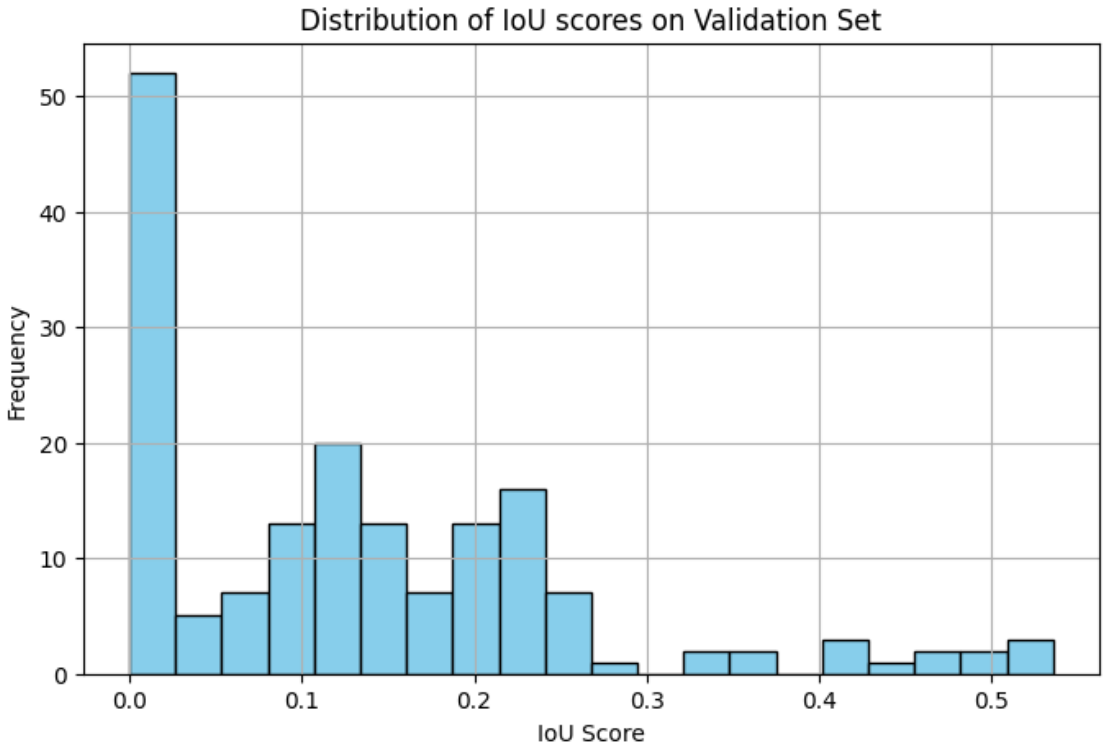
```
  def forward(self, x):
    #block 1
    x= F.leaky_relu(self.bn1(self.conv1(x)), negative_slope= 0.1)
    x= F.leaky_relu(self.bn1b(self.conv1b(x)), negative_slope= 0.1)
    x= F.max_pool2d(x, 2)

    #block 2
    x= F.leaky_relu(self.bn2(self.conv2(x)), negative_slope= 0.1)
    x= F.leaky_relu(self.bn2b(self.conv2b(x)), negative_slope= 0.1)
    x= F.max_pool2d(x, 2)

    #block 3
    x= F.leaky_relu(self.bn3(self.conv3(x)), negative_slope= 0.1)
    x= F.leaky_relu(self.bn3b(self.conv3b(x)), negative_slope= 0.1)
    x= F.max_pool2d(x, 2)

    #block 4
    x= F.leaky_relu(self.bn4(self.conv4(x)), negative_slope= 0.1)
    x= F.leaky_relu(self.bn4b(self.conv4b(x)), negative_slope= 0.1)
    x= F.max_pool2d(x, 2)

    #block 5
    x= F.leaky_relu(self.bn5(self.conv5(x)), negative_slope= 0.1)
    x= F.leaky_relu(self.bn5b(self.conv5b(x)), negative_slope= 0.1)
    x= self.pool(x)

    #flatten and predict
    x= x.view(x.size(0), -1)
    x= self.dropout(x)
    x= self.fc(x)

    #normalize coordinates between 0-1
    x= torch.sigmoid(x)
    return x
```

[35]:
```
#initialize model, loss and optimizer
third_model= ThirdModel().to(device)
optimizer= torch.optim.AdamW(third_model.parameters(),
                             lr= 0.0001,
                             weight_decay= 0.0)
criterion= nn.MSELoss()

#train for 50 epochs
history_third= train_model2(third_model,
                                train_loader,
                                val_loader,
```

```
                                criterion,
                                optimizer,
                                device,
                                epochs= 50,
                                model_name= 'third_model.pth',
                                verbose= False)

#plot loss curves
plot_loss_curves(history_third)

#load best model and evaluate
third_model.load_state_dict(torch.load('third_model.pth'))
third_model.to(device)
show_predictions(third_model, val_loader, device, num_images= 4)
evaluate_iou_with_plot(third_model, val_loader, device, verbose= False)
```




```
Mean IoU: 0.1337
```




Loss Curve: There is a smoother downward training curve than my second model, and the validation loss is going down overall as well, but with each additional model iteration from the baseline the validation loss seems to stray further and further from the training loss indicating model overfitting.

Predicted Boxes: I see more overlap between the predicted and ground truth boxes, pointing to the model starting to get a better grasp on what is and is not a license plate, but much improvement is still needed.

Mean IoU: 0.1337 this is another small improvement from the previous model. The model is still not just guessing the center of the picture. The score distribution is still skewed largely toward the lower scores, but now a larger group of predictions in the 0.1-0.3 range now.

The Iou scores are still pretty low overall, but this model did a better job of finding the correct license plate region and is now struggling to fine tune the box dimensions.

---

At this point it is possible that the model is hitting a ceiling due to limited training data (only ~ 1,500 pictures), and it may not be enough for the model to fully generalize on it's own. To help push past this ceiling I plan to experiment with transfer learning by bringing in a pretrained CNN backbone that has already learned general prupose visual features. I will also introduce data

augmentation to try to artificially expand the dataset. Finally I will pair these upgrades with new loss functions that optimizes for higher IoU. Combined these changes should help my model focus on actual patterns rather than memorizing examples. Hopefully this will lead to better bounding box accuracy and higher IoU scores.

### 1.7.7 Transfer Learning Baseline: ResNet Backbone

After testing three custom CNN architectures from scratch I'm now pivoting to transfer learning. My previous models show gradual improvements but still struggle to lock in tight bounding boxes. This could be due to limited training data (~1,500 images), making it tough for randomly initialized models to generalize well.

To address this I'm now bringing in a pretrained ResNet18 backbone (model that has already learned useful image features from a massive dataset [ImageNet].

**Source**: K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90. keywords: {Training;Degradation;Complexity theory;Image recognition;Neural networks;Visualization;Image segmentation},

PDF link: https://arxiv.org/pdf/1512.03385

My hope is that reusing these general purpose features will help the model quickly identify relevant visual patterns like plate edges and shapes.

In addition to using a pretrained backbone, I am now introducing data augmentation and normalization to make the model more robust and reduce overfitting.

**Data Augmentation and Normalization** These augmentations should help my model see different lighting/angles/orientations which could help compensate for the small training dataset size, and normalizing my Val/Test data subsets helps my model see input data that is on the same scale across all phases.

[36]:
```
#train transform pipeline: adds image jitter/ flips/ rotation/ normalization
train_transform= transforms.Compose([
    #simulates lighting variation
    transforms.ColorJitter(brightness= 0.3, contrast= 0.3, saturation= 0.3,␣
 ↪hue= 0.1),
    #adds flipped examples
    transforms.RandomHorizontalFlip(p= 0.5),
    #slightly rotates to simulate camera angles
    transforms.RandomRotation(degrees= 10),
    #convert to tensor
    transforms.ToTensor(),
    #normalize using ImageNet mean/std
    transforms.Normalize(mean= [0.485, 0.456, 0.406],
                         std= [0.229, 0.224, 0.225])])

#train transform pipeline: NO augmentation but still normalized
test_val_transform= transforms.Compose([
```

```
    #convert to tensor
    transforms.ToTensor(),
    #normalize using ImageNet mean/std
    transforms.Normalize(mean= [0.485, 0.456, 0.406],
                         std= [0.229, 0.224, 0.225])])
```

Apply these new augmentation transformations to my data subsets.

[37]:
```
train_dataset= LicensePlateDataset(os.path.join(base_path, 'train/images'),
                                   os.path.join(base_path, 'train/labels'),
                                   transform=train_transform)

val_dataset= LicensePlateDataset(os.path.join(base_path, 'val/images'),
                                 os.path.join(base_path, 'val/labels'),
                                 transform=test_val_transform)

test_dataset= LicensePlateDataset(os.path.join(base_path, 'test/images'),
                                  os.path.join(base_path, 'test/labels'),
                                  transform=test_val_transform)
```

### 1.7.8 Transfer Learning Baseline Model: ReseNet

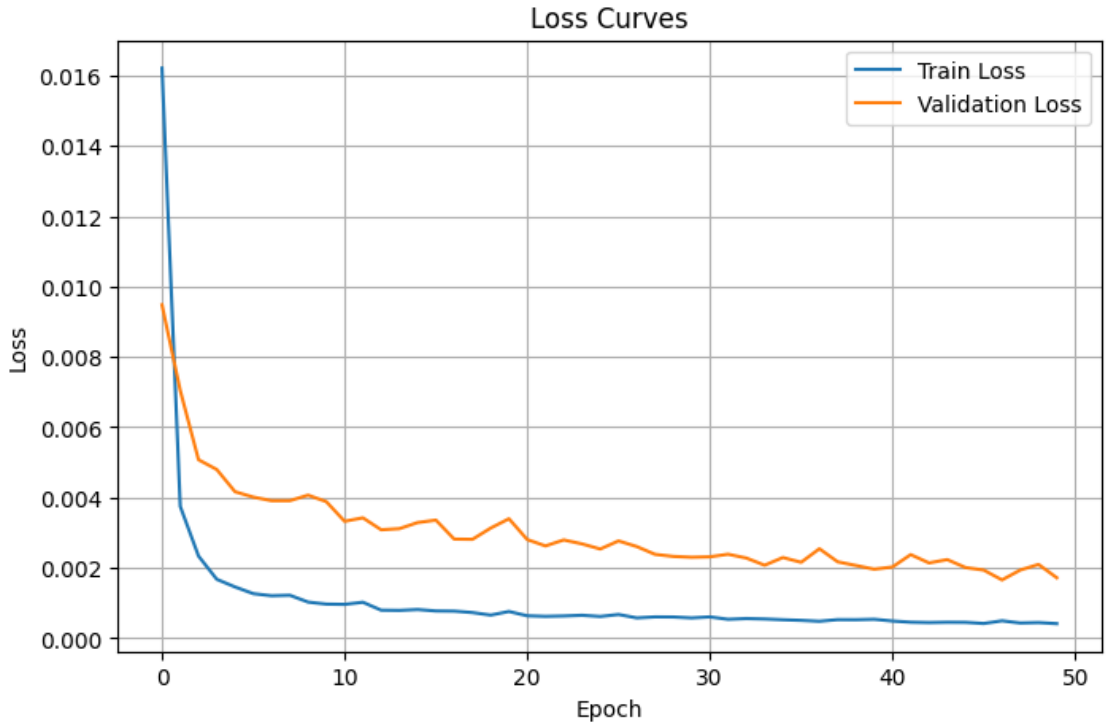
[38]:
```
import torch.nn as nn
import torchvision.models as models

#transfer learning model with pretrained ResNet18 backbone
class ResNetBackboneBBoxCNN(nn.Module):
  def __init__(self, dropout= 0.3):
    super(ResNetBackboneBBoxCNN, self).__init__()
    resnet= models.resnet18(pretrained= True)

    #use all layers up to the second to last (before final classification)
    self.backbone= nn.Sequential(*list(resnet.children())[:-2])

    #global pooling to flatten feature map
    self.pool= nn.AdaptiveAvgPool2d((1, 1))

    #regression head for bounding box prediction
    self.fc= nn.Sequential(nn.Flatten(),
                           nn.Linear(512, 128),
                           nn.ReLU(),
                           nn.Dropout(dropout),
                           #outputs x_center, y_center, width, height
                           nn.Linear(128, 4),
                           #normalize outputs between 0-1
                           nn.Sigmoid())
```

```
  def forward(self, x):
    x= self.backbone(x)
    x= self.pool(x)
    x= self.fc(x)
    return x
```

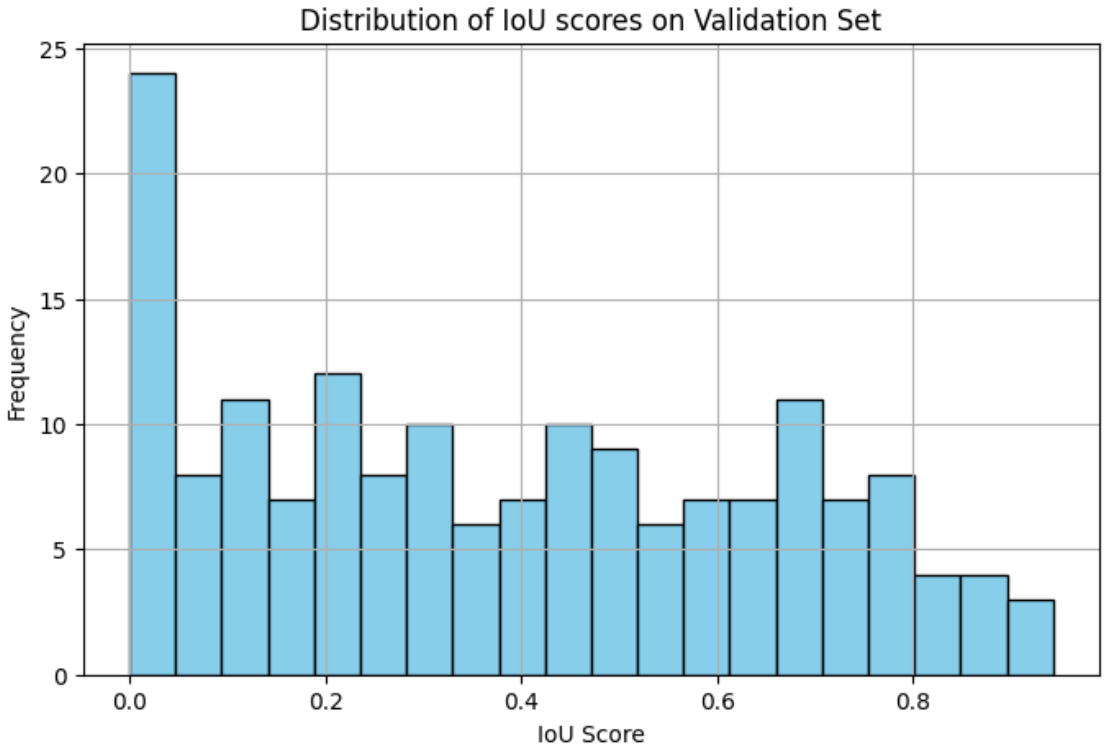
Training and evaluation

[39]:
```
#initialize model, loss and optimizer
device= torch.device("cuda" if torch.cuda.is_available() else "cpu")
transfer_model_base= ResNetBackboneBBoxCNN(dropout= 0.1).to(device)
criterion= nn.MSELoss()
optimizer= torch.optim.Adam(transfer_model_base.parameters(), lr= 0.0001)

#train for 50 epochs
TB_history= train_model2(transfer_model_base,
                      train_loader,
                      val_loader,
                      criterion,
                      optimizer,
                      device,
                      epochs=50,
                      model_name= 'transfer_baseline_model.pth',
                      verbose= False)

#load and evaluate the best version of the model
transfer_model_base.load_state_dict(torch.load('transfer_baseline_model.pth'))
transfer_model_base.to(device)
plot_loss_curves(TB_history)
show_predictions(transfer_model_base, val_loader, device, num_images=4)
evaluate_iou_with_plot(transfer_model_base, val_loader, device, verbose= False)
```

```
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to
/root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|      | 44.7M/44.7M [00:00<00:00, 205MB/s]
```




```
Mean IoU: 0.3839
```




Loss Curve: Both the training and validation loss curve show a solid downward trend, and the gap between them is much tighter than the previous models. This is a good sign that my model is generalizing better and not just memorizing my training data.

Predicted Boxes: This model clearly made a leap forward. Now many of the predicted bounding boxes are wrapping around the ground truth bounding boxes much tighter and closer. There is more consistancy across different examples and fewer random or completely off predictions.

Mean IoU: 0.3839 a significant jump from earlier scores around the 0.14 range. My score distribution is noticably healthier with an increased steady string of scores from 0-0.8. There are still a decent amount of 0.0 scores so more improvement is still needed.

These model and data changes helped my model finally breal past my IoU plateau, but I still want to build better model. To help achieve a more accurate model I will expiriment with different loss functions that help optimize for IoU.

### Try With Alternative Loss Functions

**Generalized Intersection over Union (GIoU) Loss** After establishing a strong baseline with ResNet and MSELoss I now want to experiment with bounding box specific loss functions that more directly optimize for better IoU scores.

GIoU is an improvement over regular IoU because it adds a penalty for predicted boxes that don't overlap the ground truth at all making it especially helpful when boxes are far off. Instead of just measuring overlap it also considers the area of the smallest enclosing box between the predicted

and ground truth (for two boxes that are not overlapping, the closer the boxes are to eachother the better the score, where as before the score would have been the same no matter how far apart they were).

Source: H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid and S. Savarese, "Generalized Intersection Over Union: A Metric and a Loss for Bounding Box Regression," 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 2019, pp. 658-666, doi: 10.1109/CVPR.2019.00075. keywords: {Recognition: Detection;Categorization;Retrieval;Deep Learning},

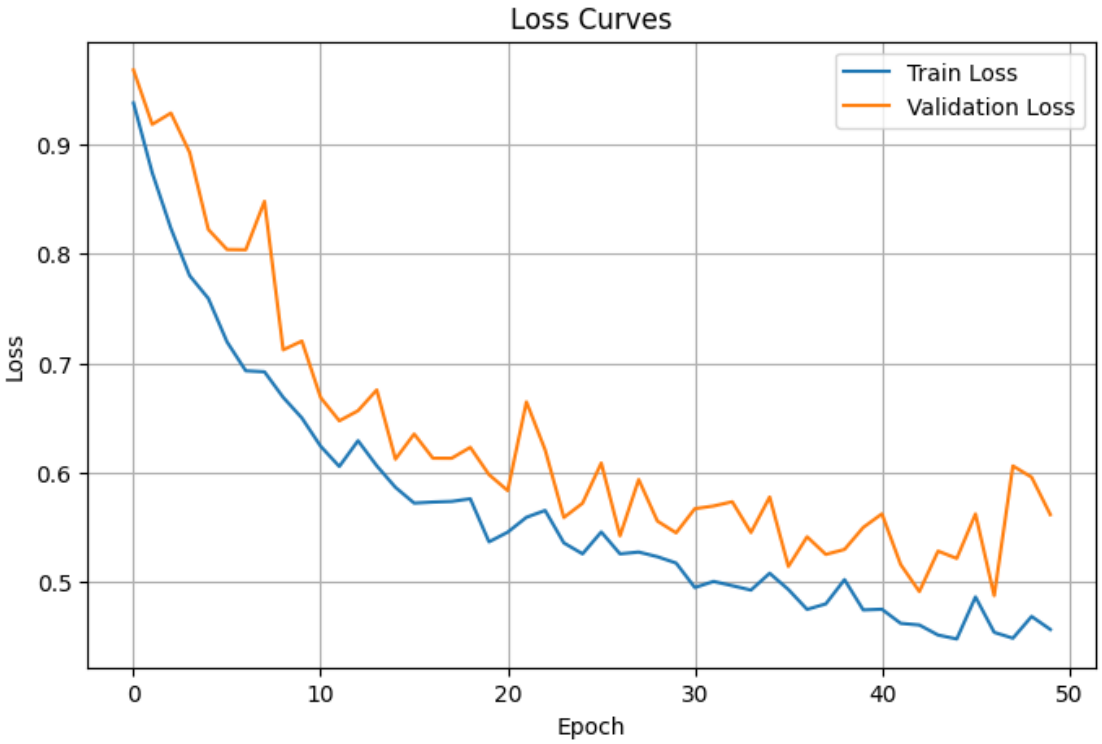
PDF Link: https://giou.stanford.edu/GIoU.pdf

[40]:
```
#GIoU loss function for bounding box regression
def giou_loss(preds, targets):
  def box_cxcywh_to_xyxy(box):
    cx, cy, w, h= box.unbind(-1)
    x1= cx - w / 2
    y1= cy - h / 2
    x2= cx + w / 2
    y2= cy + h / 2
    return torch.stack([x1, y1, x2, y2], dim= -1)

  #convert from YOLO [cx, cy, w, h] to [x1, y1, x2, y2]
  pred_boxes= box_cxcywh_to_xyxy(preds)
  target_boxes= box_cxcywh_to_xyxy(targets)

  #calculate intersection
  inter_x1= torch.max(pred_boxes[..., 0], target_boxes[..., 0])
  inter_y1= torch.max(pred_boxes[..., 1], target_boxes[..., 1])
  inter_x2= torch.min(pred_boxes[..., 2], target_boxes[..., 2])
  inter_y2= torch.min(pred_boxes[..., 3], target_boxes[..., 3])
  inter_area= (inter_x2 - inter_x1).clamp(min= 0) * (inter_y2 - inter_y1).
↪clamp(min= 0)

  #calculate union
  pred_area= (pred_boxes[..., 2] - pred_boxes[..., 0]) * (pred_boxes[..., 3] -␣
↪pred_boxes[..., 1])
  target_area= (target_boxes[..., 2] - target_boxes[..., 0]) * (target_boxes[...
↪, 3] - target_boxes[..., 1])
  union_area= pred_area + target_area - inter_area

  #calculate iou
  iou= inter_area / (union_area + 1e-7)

  #calculate area of smallest enclosing box
  enc_x1= torch.min(pred_boxes[..., 0], target_boxes[..., 0])
  enc_y1= torch.min(pred_boxes[..., 1], target_boxes[..., 1])
  enc_x2= torch.max(pred_boxes[..., 2], target_boxes[..., 2])
```

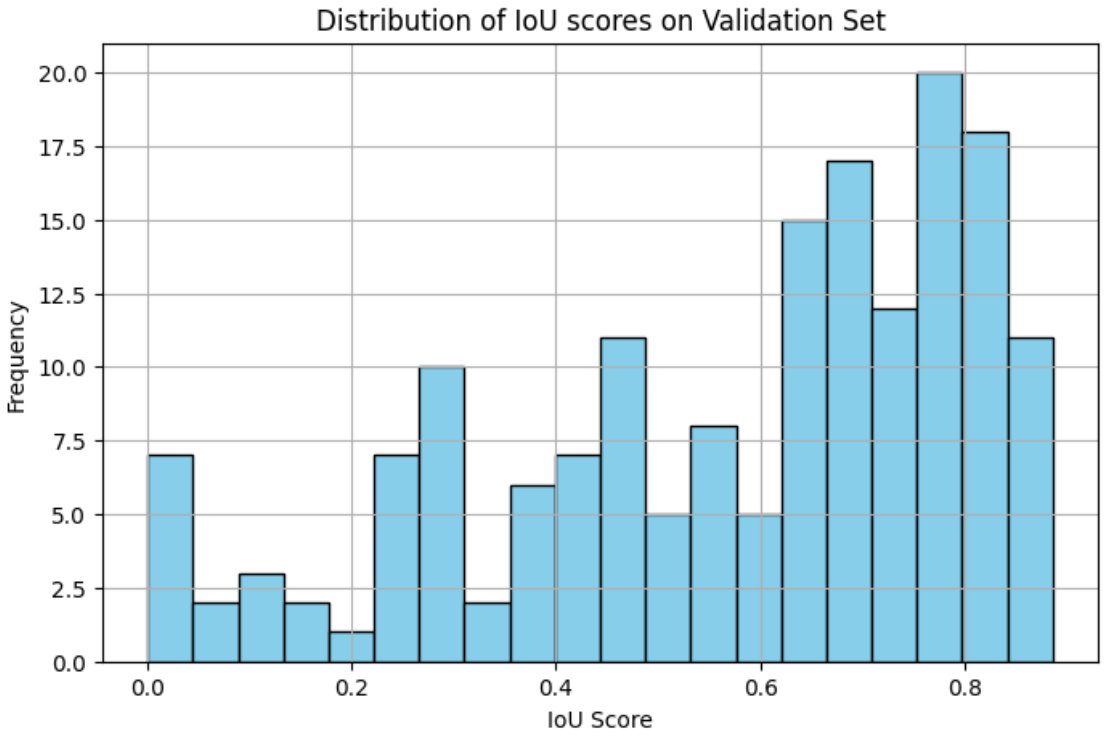
```
    enc_y2= torch.max(pred_boxes[..., 3], target_boxes[..., 3])
    enc_area= (enc_x2 - enc_x1) * (enc_y2 - enc_y1)

    #calculate GIoU
    giou= iou - (enc_area - union_area) / (enc_area + 1e-7)

    #calculate and return GIoU loss function
    return 1.0 - giou.mean()
```

Train ResNet BackBone Model with GIoU as the loss metric.

[41]:
```
#initialize model, loss and optimizer
device= torch.device("cuda" if torch.cuda.is_available() else "cpu")
transfer_model_GIOU= ResNetBackboneBBoxCNN(dropout= 0.1).to(device)
criterion= giou_loss
optimizer= torch.optim.Adam(transfer_model_GIOU.parameters(), lr= 0.0001)

#train for 50 epochs
TGIOU_history= train_model2(transfer_model_GIOU,
                      train_loader,
                      val_loader,
                      criterion,
                      optimizer,
                      device,
                      epochs=50,
                      model_name= 'transfer_GIOU_model.pth',
                      verbose= False)

#load and evaluate the best version of the model
transfer_model_GIOU.load_state_dict(torch.load('transfer_GIOU_model.pth'))
transfer_model_GIOU.to(device)
plot_loss_curves(TGIOU_history)
show_predictions(transfer_model_GIOU, val_loader, device, num_images=4)
evaluate_iou_with_plot(transfer_model_GIOU, val_loader, device, verbose= False)
```




Mean IoU: 0.5715




Loss Curve: My training and validation loss curves continue to decline steadily without the validation loss curve diverging too much, meaning my model is generalizing better.

Predicted Boxes: Major improvements in box accuracy. Most of my predictions are not tightly aligned with their ground truth boxes, and with better scale and placement than my previous models.

Mean IoU: 0.5715 is a pretty significant improvement from my previous best score (~0.38). This is my first model where the score distribution is actually clustered mainly away from a score of 0.0, with many scores being over 0.5.

This version of my model is a clear leap forward. Switching from MSE to GIoU loss helped the model stop optimizing just for box/ center dimension similarity and instead started focusing on overlap quality which is important for license plate detection.

**Distance Intersection over Union (DIoU) Loss** Now that GIoU has proven effective I am going to continue the exploration of IoU-based loss functions with Distance Intersection over Union (DIoU). Unlike GIoU, which penalizes boxes based on their coverage area, DIoU adds an additional penalty based on how far apart the centers of the predicted and ground truth bounding boxes are. This helps when bounding boxes overlap, but do not align so well.

Source: Zheng, Z., Wang, P., Liu, W., Li, J., Ye, R., & Ren, D. (2020). Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression. Proceedings of the AAAI Conference on Artificial Intelligence, 34(07), 12993-13000. https://doi.org/10.1609/aaai.v34i07.6999

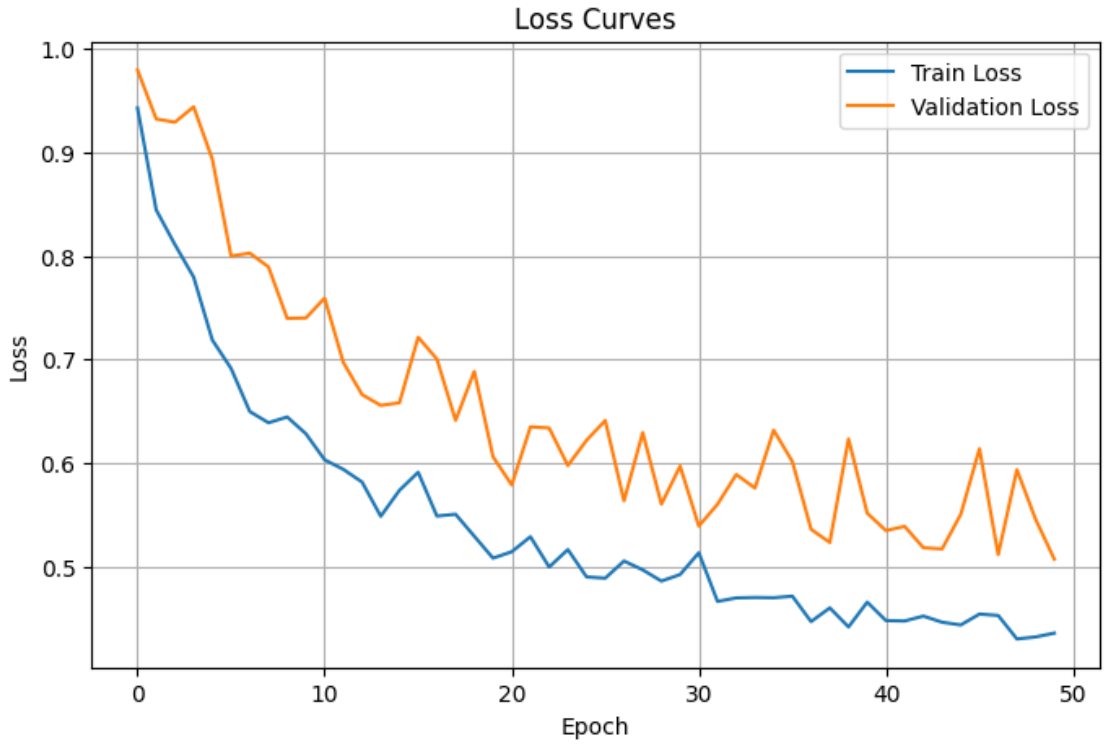
PDF link: https://arxiv.org/pdf/1911.08287

[42]:
```
#diou loss function
def diou_loss(preds, targets):
  def box_cxcywh_to_xyxy(box):
    cx, cy, w, h= box.unbind(-1)
    x1= cx - w / 2
    y1= cy - h / 2
    x2= cx + w / 2
    y2= cy + h / 2
    return torch.stack([x1, y1, x2, y2], dim= -1)

  #convert from YOLO to corner fomat
  pred_boxes= box_cxcywh_to_xyxy(preds)
  target_boxes= box_cxcywh_to_xyxy(targets)

  #compute intersection
  inter_x1= torch.max(pred_boxes[..., 0], target_boxes[..., 0])
  inter_y1= torch.max(pred_boxes[..., 1], target_boxes[..., 1])
  inter_x2= torch.min(pred_boxes[..., 2], target_boxes[..., 2])
  inter_y2= torch.min(pred_boxes[..., 3], target_boxes[..., 3])
  inter_area= (inter_x2 - inter_x1).clamp(min= 0) * (inter_y2 - inter_y1).
 ↪clamp(min= 0)

  #compute union
  pred_area= (pred_boxes[..., 2] - pred_boxes[..., 0]) * (pred_boxes[..., 3] -␣
 ↪pred_boxes[..., 1])
  target_area= (target_boxes[..., 2] - target_boxes[..., 0]) * (target_boxes[...
 ↪, 3] - target_boxes[..., 1])
  union_area= pred_area + target_area - inter_area

  #compute IoU
  iou= inter_area / (union_area + 1e-7)

  #compute center distance penalty
  pred_center= (pred_boxes[..., :2] + pred_boxes[..., 2:]) / 2
  target_center= (target_boxes[..., :2] + target_boxes[..., 2:]) / 2
  center_dist= ((pred_center - target_center) ** 2).sum(dim= -1)

  #compute diagnal length of enclosing box
  enc_x1= torch.min(pred_boxes[..., 0], target_boxes[..., 0])
  enc_y1= torch.min(pred_boxes[..., 1], target_boxes[..., 1])
  enc_x2= torch.max(pred_boxes[..., 2], target_boxes[..., 2])
  enc_y2= torch.max(pred_boxes[..., 3], target_boxes[..., 3])
  enc_diag= ((enc_x2 - enc_x1) ** 2 + (enc_y2 - enc_y1) **2)

  #compute DIoU
  diou= iou - (center_dist / (enc_diag + 1e-7))
```

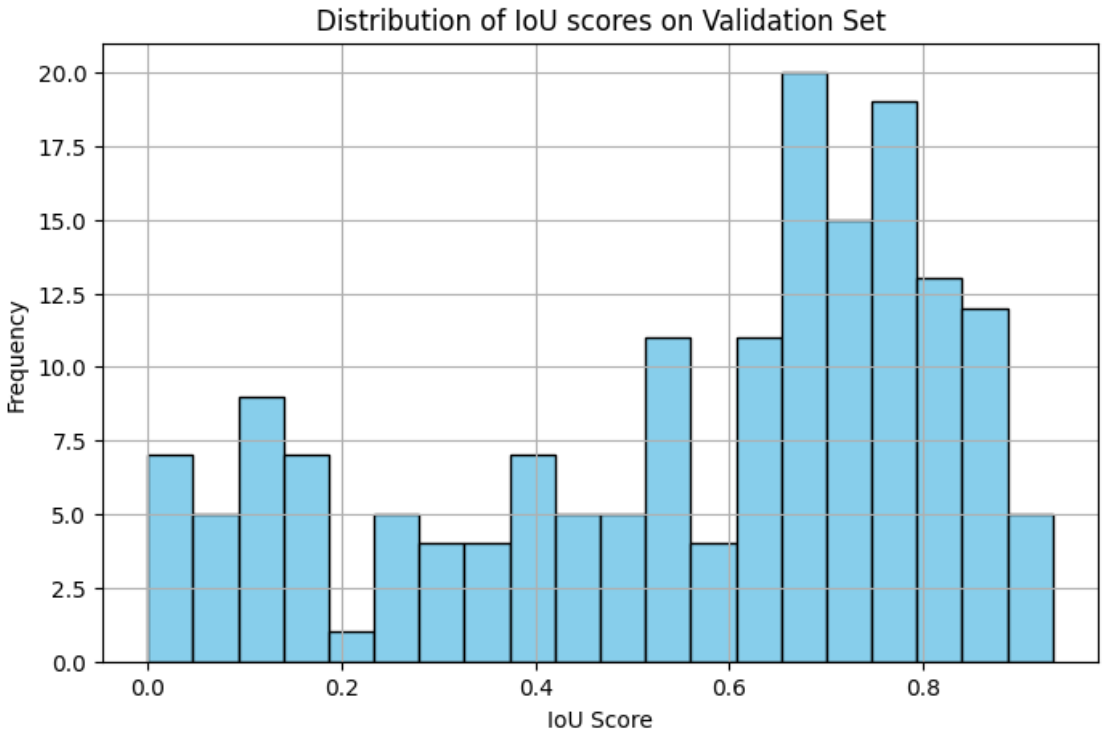
```
  #compute and return DIoU loss
  return 1.0 - diou.mean()
```

Train and evaluate model with DIoU Loss.

[43]:
```
#initialize model, loss and optimizer
device= torch.device("cuda" if torch.cuda.is_available() else "cpu")
transfer_model_DIOU= ResNetBackboneBBoxCNN(dropout= 0.1).to(device)
criterion= diou_loss
optimizer= torch.optim.Adam(transfer_model_DIOU.parameters(), lr= 0.0001)

#train for 50 epochs
TDIOU_history= train_model2(transfer_model_DIOU,
                      train_loader,
                      val_loader,
                      criterion,
                      optimizer,
                      device,
                      epochs=50,
                      model_name= 'transfer_DIOU_model.pth',
                      verbose= False)

#load and evaluate the best version of the model
transfer_model_DIOU.load_state_dict(torch.load('transfer_DIOU_model.pth'))
transfer_model_DIOU.to(device)
plot_loss_curves(TDIOU_history)
show_predictions(transfer_model_DIOU, val_loader, device, num_images=4)
evaluate_iou_with_plot(transfer_model_DIOU, val_loader, device, verbose= False)
```




Mean IoU: 0.5555




Loss Curve: These loss curves are still healthy with both the training and validation lines decreasing consistantly. This gap between the two is also still small suggesting the model generalizes well.

Predicted Boxes: The predictions are pretty precise especially in center alignment. Overall the box shapes look tighter and less random then before.

Mean IoU: 0.5555 is still a fairly solid score and is not too much lower than the GIoU loss result (0.5715). The score distribution is also very similar to the GIoU distribution with most predictions being 0.5 or above.

This DIoU loss gave me another high perfoming model that is close to the GIoU loss model. This model specifically helps with box alignment (center matching) which is important for license plate box detection.

**Grid Search: Tune Hyperparameters for GIoU and DIoU Models** Now that I have two models (GIoU and DIoU) performing reasonably well I am going to run a grid search on them to tune their hyperparameters. My goal is to maximize their accuracy by finding the best combination of learning rate, weight decay and dropout. Since the architectures and loss functions are already solid fine tuning the training setup may help me squeeze out additional improvements and get more consistent IoU scores across the validation set.

I will specifically be tuning the following parameters:

- Learning Rate: controls how fast the model updates its weights
- Weight Decay: helps regularize and prevent overfitting
- Dropout: can be added before the final layers for regularization

For each loss function (GIoU and DIoU) I will test the models with different values and evaluate performance after 15 training epochs. Grid searching can take some time to run, so to save processing time while still showing utility, I will show the full search scope commented to the right of each line like # [x, y, z] while a subset example is run in the actual code. The winning values will be included in the subset that is actually run.

[44]:
```
#define the values we want to test for each hyper parameter
dropout_vals= [0] #[0, 0.1, 0.3]
lr_vals= [1e-4, 3e-4, 1e-5, 3e-5] #[1e-4, 3e-4, 1e-3, 1e-5, 3e-5]
wd_vals= [0] #[0, 1e-4, 1e-3]

#create a list list of all possible hyperparameter combinations
grid= list(itertools.product(dropout_vals, lr_vals, wd_vals))

#sempty list to store the results from each run
results= []

#loop through every combination of dropout, learning rate, and weight decay
for dropout, lr, wd in grid:
  #print(f"\nTraining model with dropout= {dropout}, lr= {lr}, weight_decay=␣
 ↪{wd}")
  #initialize model, loss and optimizer
  device= torch.device("cuda" if torch.cuda.is_available() else "cpu")
  model= ResNetBackboneBBoxCNN(dropout= dropout).to(device)
  optimizer= optim.Adam(model.parameters(), lr= lr, weight_decay= wd)

  #train for 15 epochs
  history = train_model2(model,
                         train_loader,
                         val_loader,
                         criterion= giou_loss,
                         optimizer= optimizer,
                         device= device,
                         epochs= 15,
                         model_name= 'temp_model.pth',
                         verbose= False)

  #extract the best results from the run and save the values
  best_iou= max(history['val_mean_iou']) if history['val_mean_iou'] else 0
  results.append({'dropout': dropout,
                  'lr': lr,
                  'weight_decay': wd,
                  'val_mean_iou': best_iou})

#display results
results_df= pd.DataFrame(results)
print("Grid Search Results with GIoU:")
```

```
print(results_df.sort_values(by= 'val_mean_iou', ascending= False))
```

```
Grid Search Results with GIoU:
   dropout       lr  weight_decay  val_mean_iou
1        0  0.00030             0      0.463448
0        0  0.00010             0      0.434359
3        0  0.00003             0      0.256473
2        0  0.00001             0      0.165844
```

[45]:
```
#define the values we want to test for each hyper parameter
dropout_vals= [0] #[0, 0.1, 0.3]
lr_vals= [1e-4, 3e-4, 1e-5, 3e-5] #[1e-4, 3e-4, 1e-3, 1e-5, 3e-5]
wd_vals= [0] #[0, 1e-4, 1e-3]

#create a list list of all possible hyperparameter combinations
grid= list(itertools.product(dropout_vals, lr_vals, wd_vals))

#sempty list to store the results from each run
results= []

#loop through every combination of dropout, learning rate, and weight decay
for dropout, lr, wd in grid:
  #print(f"\nTraining model with dropout= {dropout}, lr= {lr}, weight_decay=␣
 ↪{wd}")
  #initialize model, loss and optimizer
  device= torch.device("cuda" if torch.cuda.is_available() else "cpu")
  model= ResNetBackboneBBoxCNN(dropout= dropout).to(device)
  optimizer= optim.Adam(model.parameters(), lr= lr, weight_decay= wd)

  #train for 15 epochs
  history = train_model2(model,
                         train_loader,
                         val_loader,
                         criterion= diou_loss,
                         optimizer= optimizer,
                         device= device,
                         epochs= 15,
                         model_name= 'temp_model.pth',
                         verbose= False)

  #extract the best results from the run and save the values
  best_iou= max(history['val_mean_iou']) if history['val_mean_iou'] else 0
  results.append({'dropout': dropout,
                  'lr': lr,
                  'weight_decay': wd,
                  'val_mean_iou': best_iou})
```

```
#display results
results_df= pd.DataFrame(results)
print("Grid Search Results with DIoU:")
print(results_df.sort_values(by= 'val_mean_iou', ascending= False))
```

```
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
Grid Search Results with DIoU:
   dropout       lr  weight_decay  val_mean_iou
1        0  0.00030             0      0.493084
0        0  0.00010             0      0.416178
3        0  0.00003             0      0.256032
2        0  0.00001             0      0.156327
```

These results suggest DIoU may be a better fit for this bounding box regression task overall, especially when paired with teel tuned hyperparameters. Both DIoU and GIoU benfit noticably from grid searching showing that careful model tuning is just as important as the choice of loss function.

Based on the grid search, I will now re-run the ResNet Backbone model using the best performing DIoU configuration:

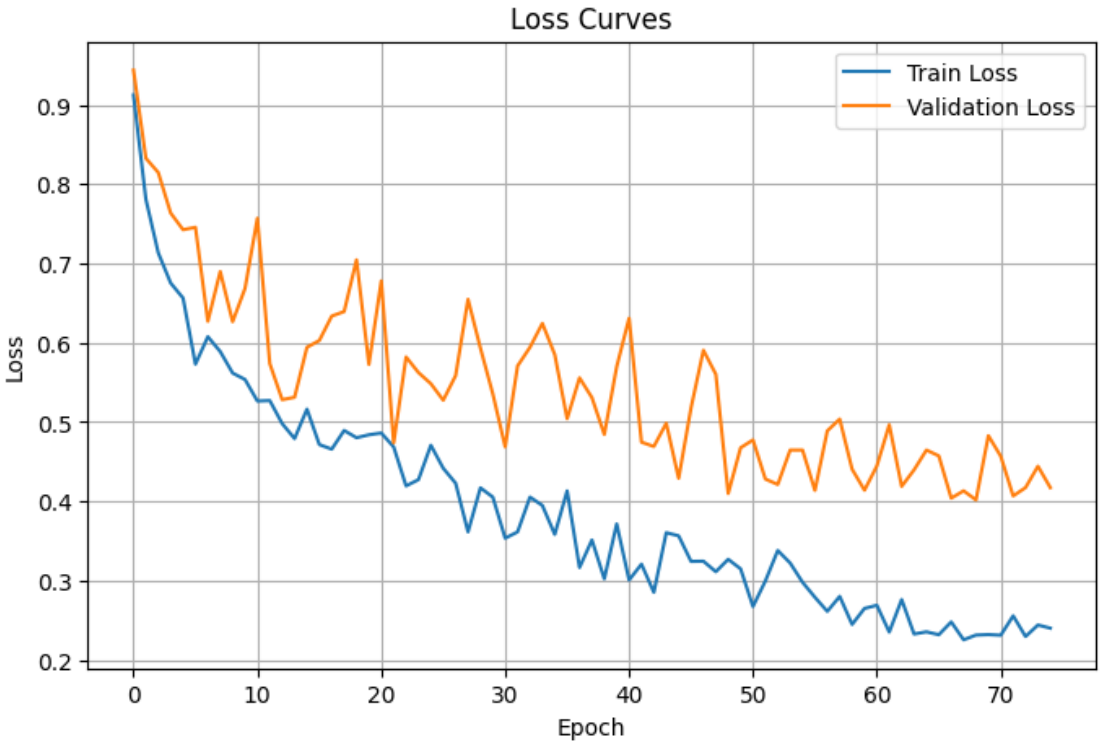
- Learning Rate= 0.0003
- Weight Decay: 0.0
- Dropout: 0.0

This time I will also extend training to 75 epochs to see if giving the model more time helps push the validation IoU even higher.

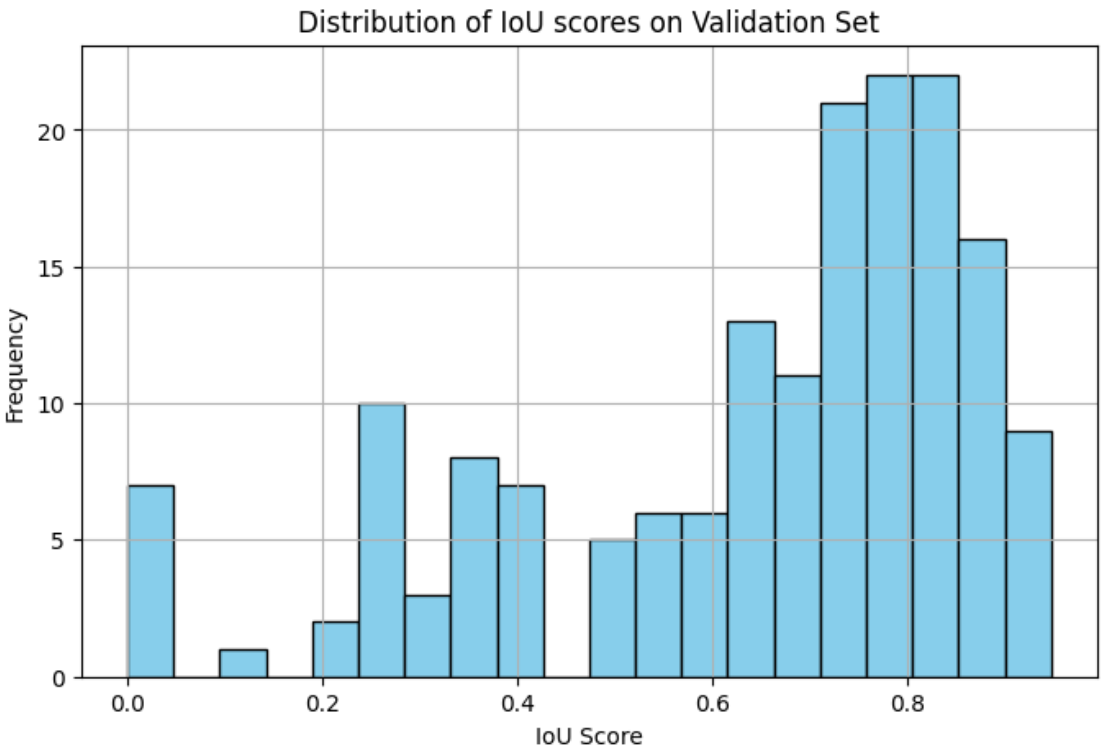
[46]:
```
#initialize model, loss and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tuned_DIOU_transfer = ResNetBackboneBBoxCNN(dropout=0.0).to(device)
criterion = diou_loss
optimizer = torch.optim.Adam(tuned_DIOU_transfer.parameters(), lr=0.0003,␣
 ↪weight_decay=0.0)

#train for 75 epochs
TD_history = train_model2(tuned_DIOU_transfer,
                          train_loader,
                          val_loader,
                          criterion,
```

```
                        optimizer,
                        device,
                        epochs=75,
                        model_name='tuned_DIOU_transfer_model.pth',
                        verbose= False)

#load and evaluate the best version of the model
tuned_DIOU_transfer.load_state_dict(torch.load('tuned_DIOU_transfer_model.pth'))
tuned_DIOU_transfer.to(device)
plot_loss_curves(TD_history)
show_predictions(tuned_DIOU_transfer, val_loader, device, num_images=4)
evaluate_iou_with_plot(tuned_DIOU_transfer, val_loader, device, verbose= False)
```

```
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```



```
Mean IoU: 0.6396
```




Loss Curve: Both the training and the validation loss curves drop steadily with no signs of divergence or plateauing. Training the model for 75 epochs seems to have given the model enough time to converge further without overfitting too much.

Predicted Boxes: This is the best visual alignment yet. Most predictions are cleanly wrapped around the license plate, with the fewest amount of incorrect box shaped or major misses.

Mean IoU: 0.6396 is a major milestone. this is my first model to break 0.6 on validation IoU. The IoU distribution is much healthier now with a strong cluster of scores around 0.5-0.8.

This is my best performing model. It combines:

- a pretrained ResNet18 backbone for strong initial features
- DIoU loss for alignment aware box regression
- normalized inputs and augmentation for robustness
- tuned hyperparameters to get the most out of model training

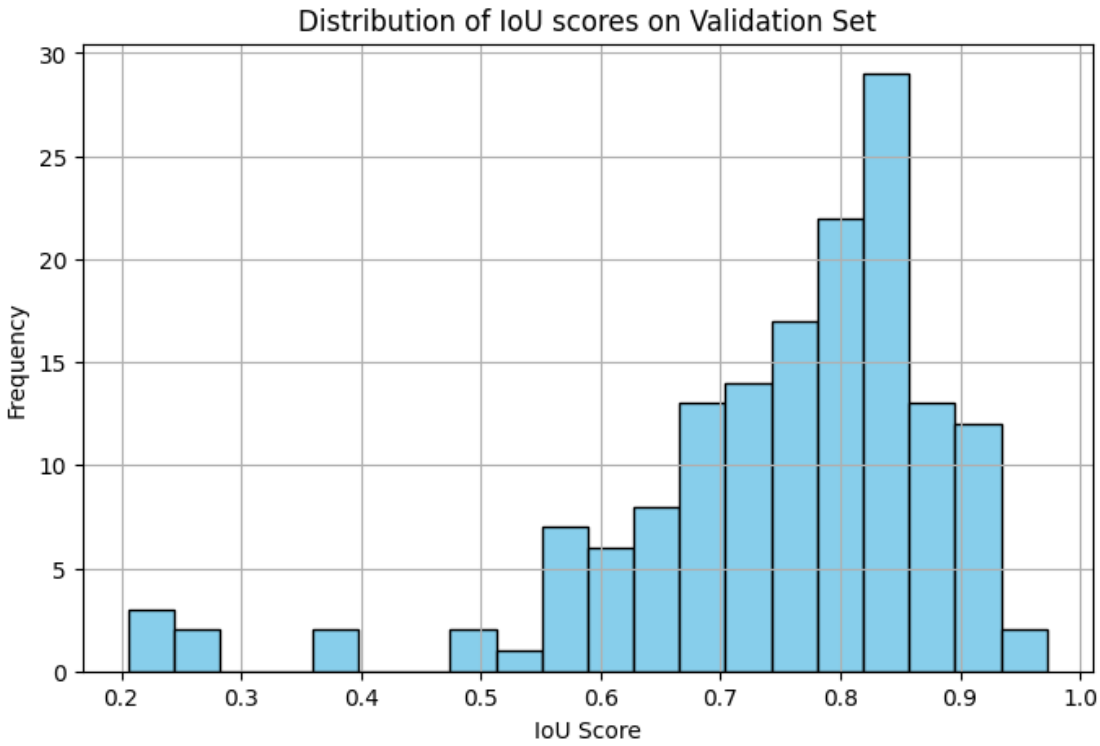
Now that the validation performance is where I want it I will run my model on my test holdout set to simulate how it may perform on new unseen data.

### 1.8 Final Test

[47]:
```
#load in the final test data
val_transform = transforms.ToTensor()

test_dataset= LicensePlateDataset(image_dir= 'data/final_split/test/images',
                                  label_dir= 'data/final_split/test/labels',
                                  transform= val_transform)
test_loader= DataLoader(test_dataset, batch_size= 16, shuffle= False)
```

[48]:
```
#run predictions on evaluation test set
#make sure model is in evaluation mode and on correct device
tuned_DIOU_transfer.eval()
tuned_DIOU_transfer.to(device)

#visualize predictions
show_predictions(tuned_DIOU_transfer, test_loader, device, num_images= 4)

#compute and print IOU
evaluate_iou_with_plot(tuned_DIOU_transfer, test_loader, device, verbose= False)
```

```
Mean IoU: 0.7475
```




## 1.9 Conclusion

My conclusion from this analysis is that our company could successfully use computer vision and supervised learning to automate the license plate detection stage of the toll collection process. This automation could reduce the need for manual review of vehicle images, streamline toll processing, and serve as a reliable upstream component for and OCR system that extracts plate text.

In this specific case my Tuned DIoU ResNet Transfer Model is my strongest performer and could now be deployed on unseen highway traffic images to locate license plates on cars with a high degree of accuracy. My model was trained using a pretrained ResNet18 backbone for strong base features, a distance aware bounding box loss (DIoU), data normalization and augmentation for generalization, and a tuned set of hyperparameters for peak performance.

Based on my test set evaluation I can highlight 3 key takeaways for our company:

1. **High IoU Score Suggests Readiness for Real World Integration:** With a mean IoU of 0.7475 on the test set this model consistently generates bounding boxes that tightly match the ground truth labels. This means the license plate region is being accurately predicted in most vehicle images which would allow our OCR team to extract the plate numbers with a high confidence that they are in the "plate pictures" they are being sent to them.

2. **Consistent Bounding Box Placement Reduces Human Intervention:** This model shows highly consistent bounding box placement across cars of different shapes and orientations which reduces the need for human correction of boxes and allows our system to scale. In

operations, fewer incorrectly placed or oversized boxes means fewer OCR misreads or customer disputes due to bad plate captures. This boosts our overall reliability in toll enforcement.

3. **Automated Detection Unlocks Efficiency in Toll Systems:** This model serves as the first step in our larger automated tolling pipeline. By confidently detecting where the plate is located it replaces the manual step of cropping or drawing bounding boxes by hand or expensive third party systems. This means we can process more vehicles per hour at lower cost and potentially expand to new locations or lanes without increasing headcount.

### 1.10 Next Steps

Here are three potential next steps that our company could take to further improve the accuracy and deployability of our license plate detection system:

1. **Retrain on Region Specific Plate Styles and Vehicle Types:** The current model was trained entirely on non-U.S. plates and mostly featured cars and vans in the pictures. If we plan to deploy this model in a U.S. tolling environment we should retrain or fine tune the model on region-specific data that includes U.S plate formats, motorcycles, and commercial trucks. This would help the model generalize to the types of vehicles and plate designs it's most likely to encounter in the field, reducing detection error rates.
2. **Add Tracking Component for Multi-Frame Video Inputs:** In many real world deployments vehicle data is captured in short video bursts instead of a single frame. Adding object tracking to follow the plate region across frames could increase reliability by allowing us to average predictions or choose the clearest frame for the OCR team to extract the plate number from. This would especially help combat frames that have motion blur, glare, or poor visibility conditions.
3. **Expand to Multi-Plate Scenarios:** As we scale we may encounter images with multiple vehicles or stacked plates. Updating the model to handle multiple bounding boxes per image would make our system more flexible and more robust across different toll lanes. For example one camera may be able to pick up the plates of two different lanes instead of just one camera per lane.

[48]: