AI Lab: Deep Learning for Computer Vision
WorldQuant University

Getting Started

Let's import the packages we'll need in this notebook. Most are familiar. We will need version 2 of the `torchvision.transforms` module here. The API is slightly different than that of version 1 that we've used previously, but it's pretty similar.

In [1]:
```
import pathlib
import sys

import matplotlib.pyplot as plt
import torch
import torchinfo
import torchvision
import ultralytics
from PIL import Image
from torchvision.transforms import v2
from ultralytics import YOLO
```

```
Creating new Ultralytics Settings v0.0.6 file ✅
View Ultralytics Settings with 'yolo settings' or at '/root/.config/Ultralytics/sett
ings.json'
Update Settings with 'yolo settings key=value', i.e. 'yolo settings runs_dir=path/t
o/dir'. For help see https://docs.ultralytics.com/quickstart/#ultralytics-settings.
```

In case we want to reproduce this notebook in the future, we'll record the version information.

In [2]:
```
print("Platform:", sys.platform)
print("Python version:", sys.version)
print("---")
print("matplotlib version:", plt.matplotlib.__version__)
print("PIL version:", Image.__version__)
print("torch version:", torch.__version__)
print("torchvision version:", torchvision.__version__)
print("ultralytics version:", ultralytics.__version__)
```

```
Platform: linux
Python version: 3.11.0 (main, Nov 15 2022, 20:12:54) [GCC 10.2.1 20210110]
---
matplotlib version: 3.9.2
PIL version: 10.2.0
torch version: 2.2.2+cu121
torchvision version: 0.17.2+cu121
ultralytics version: 8.3.27
```

These are the classes of the Dhaka AI data set we've seen before.

In [3]:
```
CLASS_DICT = dict(
    enumerate(
        [
            "ambulance",
            "army vehicle",
            "auto rickshaw",
            "bicycle",
            "bus",
            "car",
            "garbagevan",
            "human hauler",
            "minibus",
            "minivan",
            "motorbike",
            "pickup",
            "policecar",
            "rickshaw",
            "scooter",
            "suv",
            "taxi",
            "three wheelers (CNG)",
            "truck",
            "van",
            "wheelbarrow",
        ]
    )
```

```
)

print("CLASS_DICT type,", type(CLASS_DICT))
CLASS_DICT
```

```
CLASS_DICT type, <class 'dict'>
```

Out[3]:
```
{0: 'ambulance',
 1: 'army vehicle',
 2: 'auto rickshaw',
 3: 'bicycle',
 4: 'bus',
 5: 'car',
 6: 'garbagevan',
 7: 'human hauler',
 8: 'minibus',
 9: 'minivan',
 10: 'motorbike',
 11: 'pickup',
 12: 'policecar',
 13: 'rickshaw',
 14: 'scooter',
 15: 'suv',
 16: 'taxi',
 17: 'three wheelers (CNG)',
 18: 'truck',
 19: 'van',
 20: 'wheelbarrow'}
```

Data Augmentation in Our YOLO Model

In the previous notebook, we passed our training images to the YOLO model and let it do its thing. The obvious assumption to make is that these images would be used as is, but it turns out not to be so. To demonstrate what was happening, we'll load the model back up and poke around inside of it a bit.

Let's start by finding a saved version of the model. This cell should show all of the training runs that have been completed.

In [4]:
```
runs_dir = pathlib.Path("runs", "detect")
list(runs_dir.glob("train*"))
```

Out[4]:
```
[PosixPath('runs/detect/train')]
```

> If you don't see anything listed here, go back and run the previous notebook all the way through!

Task 3.5.1: Choose a training run, and check that there are model weights saved in the `weights/best.pt` file for that run.

In [5]:
```
run_dir = runs_dir / "train"
weights_file = run_dir / "weights" / "best.pt"

print("Weights file exists?", weights_file.exists())
```

```
Weights file exists? True
```

Task 3.5.2: Load the model from the weights file.

In [6]:
```
model = YOLO(weights_file)

torchinfo.summary(model)
```

Out[6]:
```
==========================================================================================
Layer (type:depth-idx)                                   Param #
==========================================================================================
YOLO                                                     --
├─DetectionModel: 1-1                                    --
│    └─Sequential: 2-1                                   --
│    │    └─Conv: 3-1                                    (464)
│    │    └─Conv: 3-2                                    (4,672)
│    │    └─C2f: 3-3                                     (7,360)
│    │    └─Conv: 3-4                                    (18,560)
│    │    └─C2f: 3-5                                     (49,664)
│    │    └─Conv: 3-6                                    (73,984)
│    │    └─C2f: 3-7                                     (197,632)
│    │    └─Conv: 3-8                                    (295,424)
│    │    └─C2f: 3-9                                     (460,288)
│    │    └─SPPF: 3-10                                   (164,608)
│    │    └─Upsample: 3-11                               --
│    │    └─Concat: 3-12                                 --
│    │    └─C2f: 3-13                                    (148,224)
│    │    └─Upsample: 3-14                               --
│    │    └─Concat: 3-15                                 --
│    │    └─C2f: 3-16                                    (37,248)
│    │    └─Conv: 3-17                                   (36,992)
│    │    └─Concat: 3-18                                 --
│    │    └─C2f: 3-19                                    (123,648)
│    │    └─Conv: 3-20                                   (147,712)
│    │    └─Concat: 3-21                                 --
│    │    └─C2f: 3-22                                    (493,056)
│    │    └─Detect: 3-23                                 (755,407)
==========================================================================================
Total params: 3,014,943
Trainable params: 0
Non-trainable params: 3,014,943
==========================================================================================
```

We need to get the model set up to load the data. The easiest way to do that is to train it for an epoch.

When you call `.train()` on a YOLO model, it sets up a data loader, if it doesn't already exist. Unfortunately, there's no easy way to trigger that set-up step without doing an

epoch of training. 😌

Task 3.5.3: Run one epoch of training.

In [7]:
```
result = model.train(
    data=model.overrides["data"],
    epochs=1,
    batch=8,
    workers=1,
)
```

```
Ultralytics 8.3.27 🚀 Python-3.11.0 torch-2.2.2+cu121 CUDA:0 (Tesla T4, 14918MiB)
engine/trainer: task=detect, mode=train, model=runs/detect/train/weights/best.pt, da
ta=data.yaml, epochs=1, time=None, patience=100, batch=8, imgsz=640, save=True, save
_period=-1, cache=False, device=None, workers=1, project=None, name=train2, exist_ok
=False, pretrained=True, optimizer=auto, verbose=True, seed=0, deterministic=True, s
ingle_cls=False, rect=False, cos_lr=False, close_mosaic=10, resume=False, amp=True,
fraction=1.0, profile=False, freeze=None, multi_scale=False, overlap_mask=True, mask
_ratio=4, dropout=0.0, val=True, split=val, save_json=False, save_hybrid=False, conf
=None, iou=0.7, max_det=300, half=False, dnn=False, plots=True, source=None, vid_str
ide=1, stream_buffer=False, visualize=False, augment=False, agnostic_nms=False, clas
ses=None, retina_masks=False, embed=None, show=False, save_frames=False, save_txt=Fa
lse, save_conf=False, save_crop=False, show_labels=True, show_conf=True, show_boxes=
True, line_width=None, format=torchscript, keras=False, optimize=False, int8=False,
dynamic=False, simplify=True, opset=None, workspace=4, nms=False, lr0=0.01, lrf=0.0
1, momentum=0.937, weight_decay=0.0005, warmup_epochs=3.0, warmup_momentum=0.8, warm
up_bias_lr=0.1, box=7.5, cls=0.5, dfl=1.5, pose=12.0, kobj=1.0, label_smoothing=0.0,
nbs=64, hsv_h=0.015, hsv_s=0.7, hsv_v=0.4, degrees=0.0, translate=0.1, scale=0.5, sh
ear=0.0, perspective=0.0, flipud=0.0, fliplr=0.5, bgr=0.0, mosaic=1.0, mixup=0.0, co
py_paste=0.0, copy_paste_mode=flip, auto_augment=randaugment, erasing=0.4, crop_frac
tion=1.0, cfg=None, tracker=botsort.yaml, save_dir=runs/detect/train2

                   from  n    params  module                                       a
rguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv
[16, 32, 3, 2]
  2                  -1  1      7360  ultralytics.nn.modules.block.C2f
[32, 32, 1, True]
  3                  -1  1     18560  ultralytics.nn.modules.conv.Conv
[32, 64, 3, 2]
  4                  -1  2     49664  ultralytics.nn.modules.block.C2f
[64, 64, 2, True]
  5                  -1  1     73984  ultralytics.nn.modules.conv.Conv
[64, 128, 3, 2]
  6                  -1  2    197632  ultralytics.nn.modules.block.C2f
[128, 128, 2, True]
  7                  -1  1    295424  ultralytics.nn.modules.conv.Conv
[128, 256, 3, 2]
  8                  -1  1    460288  ultralytics.nn.modules.block.C2f
[256, 256, 1, True]
  9                  -1  1    164608  ultralytics.nn.modules.block.SPPF
[256, 256, 5]
 10                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 11             [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat
[1]
 12                  -1  1    148224  ultralytics.nn.modules.block.C2f
[384, 128, 1]
 13                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 14             [-1, 4]  1         0  ultralytics.nn.modules.conv.Concat
[1]
 15                  -1  1     37248  ultralytics.nn.modules.block.C2f
[192, 64, 1]
 16                  -1  1     36992  ultralytics.nn.modules.conv.Conv
```

```
[64, 64, 3, 2]
 17            [-1, 12]  1         0  ultralytics.nn.modules.conv.Concat
[1]
 18                -1  1    123648  ultralytics.nn.modules.block.C2f
[192, 128, 1]
 19                -1  1    147712  ultralytics.nn.modules.conv.Conv
[128, 128, 3, 2]
 20             [-1, 9]  1         0  ultralytics.nn.modules.conv.Concat
[1]
 21                -1  1    493056  ultralytics.nn.modules.block.C2f
[384, 256, 1]
 22        [15, 18, 21]  1    755407  ultralytics.nn.modules.head.Detect
[21, [64, 128, 256]]
Model summary: 225 layers, 3,014,943 parameters, 3,014,927 gradients, 8.2 GFLOPs

Transferred 355/355 items from pretrained weights
Freezing layer 'model.22.dfl.conv.weight'
AMP: running Automatic Mixed Precision (AMP) checks...
AMP: checks passed ✅
train: Scanning /app/data_yolo/labels/train... 2422 images, 0 backgrounds, 0 corrup
t: 100%|██████████| 2422/2422 [00:00<00:00, 2597.89it/s]
train: New cache created: /app/data_yolo/labels/train.cache
val: Scanning /app/data_yolo/labels/val... 578 images, 0 backgrounds, 0 corrupt: 10
0%|██████████| 578/578 [00:00<00:00, 2344.13it/s]
val: New cache created: /app/data_yolo/labels/val.cache
/usr/local/lib/python3.11/site-packages/torch/utils/data/dataloader.py:558: UserWarn
ing: This DataLoader will create 2 worker processes in total. Our suggested max numb
er of worker in current system is 1, which is smaller than what this DataLoader is g
oing to create. Please be aware that excessive worker creation might get DataLoader
running slow or even freeze, lower the worker number to avoid potential slowness/fre
eze if necessary.
  warnings.warn(_create_warning_msg(
Plotting labels to runs/detect/train2/labels.jpg...
optimizer: 'optimizer=auto' found, ignoring 'lr0=0.01' and 'momentum=0.937' and dete
rmining best 'optimizer', 'lr0' and 'momentum' automatically...
optimizer: AdamW(lr=0.0004, momentum=0.9) with parameter groups 57 weight(decay=0.
0), 64 weight(decay=0.0005), 63 bias(decay=0.0)
Image sizes 640 train, 640 val
Using 1 dataloader workers
Logging results to runs/detect/train2
Starting training for 1 epochs...

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
        1/1         0G      1.168      1.218      1.048         95        640: 100%|
██████████| 303/303 [02:29<00:00,  2.03it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50
-95): 100%|██████████| 37/37 [00:20<00:00,  1.79it/s]
```

```
                   all        578       4590      0.613      0.363      0.414
0.286

1 epochs completed in 0.048 hours.
Optimizer stripped from runs/detect/train2/weights/last.pt, 6.2MB
Optimizer stripped from runs/detect/train2/weights/best.pt, 6.2MB

Validating runs/detect/train2/weights/best.pt...
Ultralytics 8.3.27 🚀 Python-3.11.0 torch-2.2.2+cu121 CUDA:0 (Tesla T4, 14918MiB)
Model summary (fused): 168 layers, 3,009,743 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50
-95): 100%|██████████| 37/37 [00:16<00:00,  2.30it/s]
                   all        578       4590       0.61      0.366      0.414
0.286
            ambulance         17         17          1          0    0.00737    0.0
0601
          army vehicle         10         12          1      0.164      0.427
0.359
        auto rickshaw         37        111      0.557      0.486      0.523
0.322
               bicycle         72         93      0.408       0.28      0.326
0.155
                   bus        295        645      0.679       0.69       0.73
0.515
                   car        309       1006      0.748      0.677       0.74
0.517
          human hauler         23         25      0.368       0.24      0.201
0.145
               minibus         10         11          0          0     0.0416     0.
0286
               minivan        128        208      0.387      0.442      0.352
0.255
            motorbike        238        449      0.647      0.561      0.601
0.345
                pickup        157        217      0.743      0.429      0.558
0.38
            policecar          6          6          1          0      0.223
0.166
             rickshaw        202        653      0.475      0.749      0.706
0.457
              scooter          8          9          1          0    0.00812     0.
0049
                  suv        105        161      0.274      0.514      0.366
0.271
                 taxi          8         10          1          0       0.55
0.435
  three wheelers (CNG)        224        510      0.686      0.658      0.697
0.483
                truck        159        289      0.558       0.73      0.694
0.502
                  van         86        139      0.302      0.432      0.246
0.164
           wheelbarrow         16         19      0.359      0.263      0.279
0.203
Speed: 0.6ms preprocess, 11.3ms inference, 0.0ms loss, 4.0ms postprocess per image
Results saved to runs/detect/train2
```

The model should now have a `.trainer` attribute, which has a `.train_loader` attribute. This will be a `DataLoader` that loads the training data.

Task 3.5.4: Save this data loader to variable `loader`.

In [10]:
```
loader = model.trainer.train_loader

print(type(loader))
```

```
<class 'ultralytics.data.build.InfiniteDataLoader'>
```

Data loaders are *iterables*. That is, you can put them in a `for` loop to load data one batch at a time. We just want to read one batch from it, though.

Task 3.5.5: Load one batch from `loader` into the variable `batch`. You can do this by constructing a `for` loop over `loader` and calling `break` inside the loop, so that it only runs once.

In [15]:
```
batch = next(iter(loader))
print(type(batch))
```

```
<class 'dict'>
```

> A more advanced way to accomplish this same thing is: `batch = next(iter(loader))`

We get back a dictionary. (What a surprise!) Let's explore what's in this structure.

Task 3.5.6: Print out the keys in `batch`.

In [19]:
```
print(batch.keys())
```

```
dict_keys(['im_file', 'ori_shape', 'resized_shape', 'img', 'cls', 'bboxes', 'batch_idx'])
```

Task 3.5.7: Print out the shape of the `img` value.

In [23]:
```
print((batch["img"].shape))
```

```
torch.Size([8, 3, 640, 640])
```

The dimension of 3 represents the color channels. The dimension of 640 are the width and height. So what does the dimension of 8 represent?

You can get a clue by reviewing the call to `model.train`. We set a batch size of 8. This tensor thus represents eight training images.

Task 3.5.8: Print out the shape of the `bboxes` value.

In [24]:
```
print(batch["bboxes"].shape)
```

```
torch.Size([155, 4])
```

That seems like a lot of bounding boxes for one image, so these must be the boxes for all of the images in the batch.

> The exact number of bounding boxes will depend on the random batch that got delivered to you. If you re-run the cell that creates `batch`, you'll find that you get another number here.

The image index in the batch that the box corresponds to is given in the `batch_idx` value.

In [25]:
```
print(batch["batch_idx"])
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2., 2., 2., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 4., 4., 4., 4., 4., 4.,
4., 4., 4., 4., 4., 4., 4., 5., 5., 5., 5., 5., 5., 6., 6., 7., 7., 7., 7., 7., 7.,
7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7., 7.])
```

Thus, we can select the bounding boxes for a particular image in a batch by finding the rows that correspond to a particular batch index value. This is implemented for us in the following function, which will plot the bounding boxes on top of the image.

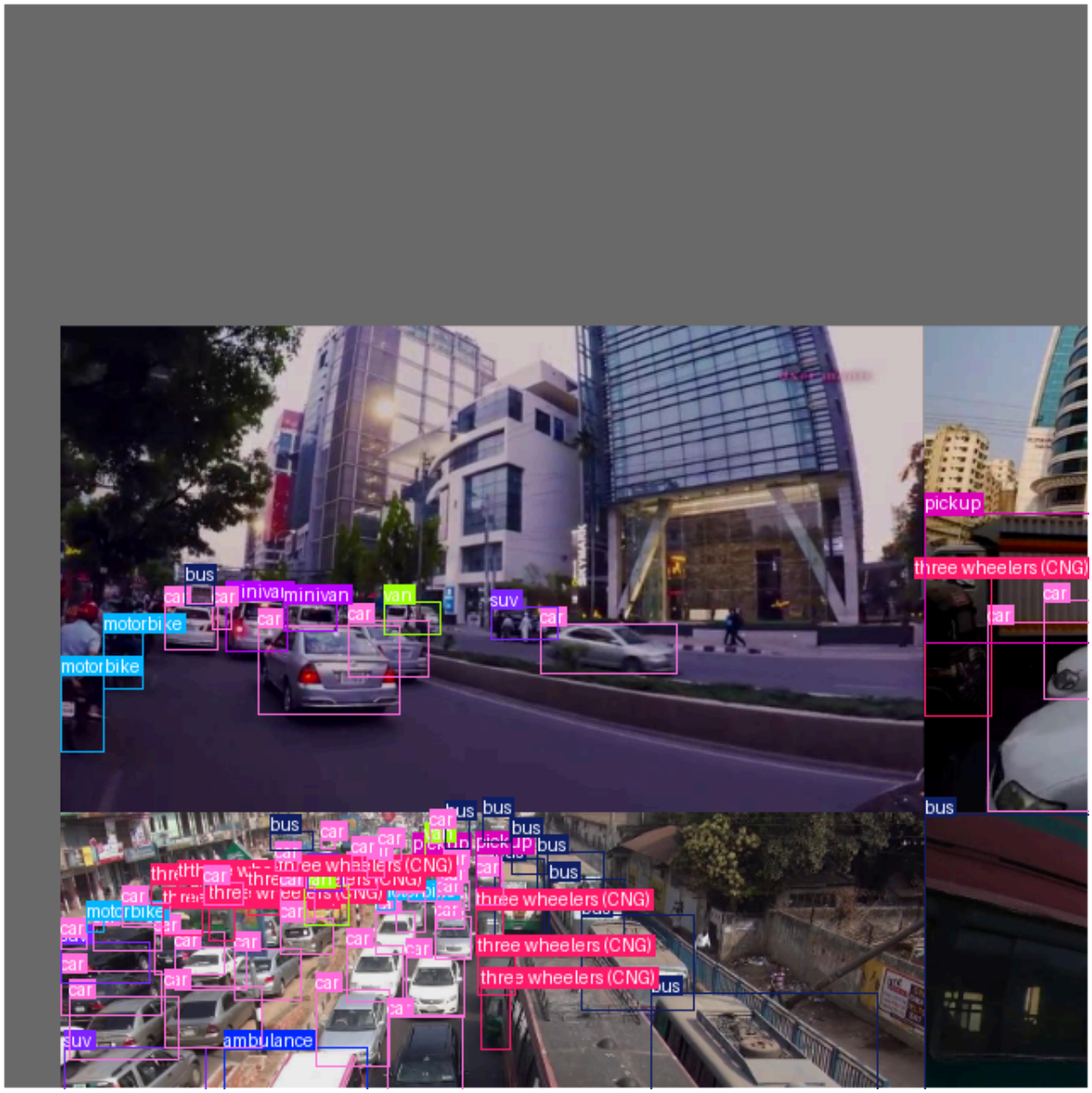
In [26]:
```
def plot_with_bboxes(img, bboxes, cls, batch_idx=None, index=0, **kw):
    """Plot the bounding boxes on an image.

    Input:  img     The image, either as a 3-D tensor (one image) or a
                    4-D tensor (a stack of images).  In the latter case,
                    the index argument specifies which image to display.
            bboxes  The bounding boxes, as a N x 4 tensor, in normalized
                    XYWH format.
            cls     The class indices associated with the bounding boxes
                    as a N x 1 tensor.
            batch_idx   The index of each bounding box within the stack of
                        images.  Ignored if img is 3-D.
            index   The index of the image in the stack to be displayed.
                    Ignored if img is 3-D.
            **kw    All other keyword arguments are accepted and ignored.
                    This allows you to use dictionary unpacking with the
                    values produced by a YOLO DataLoader.
    """
    if img.ndim == 3:
        image = img[None, :]
        index = 0
        batch_idx = torch.zeros((len(cls),))
    elif img.ndim == 4:
        # Get around Black / Flake8 disagreement
```

```
        indp1 = index + 1
        image = img[index:indp1, :]

    inds = batch_idx == index
    res = ultralytics.utils.plotting.plot_images(
        images=image,
        batch_idx=batch_idx[inds] - index,
        cls=cls[inds].flatten(),
        bboxes=bboxes[inds],
        names=CLASS_DICT,
        threaded=False,
        save=False,
    )

    return Image.fromarray(res)
```

Task 3.5.9: Plot the image and bounding boxes for index 0 of this batch.

In [33]:
```
plot_with_bboxes(**batch, index=2)
```

Out[33]:



That's ... weird looking. It's not what our original images look like, is it?

> If it's not weird looking, try looking at another index. Eventually you'll find something weird looking!

The file names from the batch are stored in the `im_file` key. We can use that to look up the original image associated with this index and see what it looks like.

Task 3.5.10: Display the original image file for this index.

In [30]:
```
Image.open(batch["im_file"][2])
```

Out[30]:

Comparing the two, we can see that the original image was distorted and combined with other images before being loaded into the YOLO model. The YOLO model applies a number of augmentation steps by default. (You can take a look at all of the augmentation settings in YOLO.) This increases the diversity of training images, which should help the model generally.

Data Augmentation with Torchvision

If you're training a YOLO model, it's generally best to use the built-in augmentation setting. But in other cases, you may need to implement an augmentation system yourself. Torchvision makes this easy by providing a number of augmentation transforms in its transforms version 2 (v2) module.

To demonstrate this, we'll load a sample image. The code below will get the file paths for `01.jpg` and its associated label file. (It's written so that it works whether the image ended up in the training or validation split.)

In [34]:
```
yolo_base = pathlib.Path("data_yolo")
sample_fn = next((yolo_base / "images").glob("*/01.jpg"))
sample_labels = next((yolo_base / "labels").glob("*/01.txt"))

print(sample_fn)
print(sample_labels)
```

```
data_yolo/images/val/01.jpg
data_yolo/labels/val/01.txt
```

Task 3.5.11: Load the image with PIL.

In [35]:
```
sample_image = Image.open(sample_fn)

sample_image
```

Out[35]:

Task 3.5.12: Convert the image to a tensor. In the transforms version 2 module, this can be done with the confusingly-named `ToImage` transform.

In [36]:
```
sample_torch = v2.ToImage()(sample_image)

print(sample_torch.shape)
```

```
torch.Size([3, 800, 1200])
```

The bounding boxes are stored in the label file. Let's take a look a the first five lines to remember what it looks like.

In [37]:
```
!head -n5 $sample_labels
```

```
4 0.8 0.74375 0.21166666666666667 0.5125
4 0.7995833333333333 0.424375 0.0975 0.13875
4 0.7995833333333333 0.33 0.08416666666666667 0.0575
13 0.66375 0.595625 0.059166666666666666 0.15875
13 0.66875 0.483125 0.0425 0.05625
```

Each line represents a bounding box. The first element is the class index. This is followed by the *x* and *y* coordinates of the box center, the width, and the height.

Task 3.5.13: Load the bounding box data into a variable named `label_data`. It should be a list of the bounding boxes. Each bounding box will itself be a list of five strings in the same order they are in the file. Don't worry about converting the strings to numbers yet.

In [39]:
```
# Load the data into `label_data`
with open(sample_labels, "r") as f:
    label_data = [row.split() for row in f]

label_data[:5]
```

Out[39]:
```
[['4', '0.8', '0.74375', '0.21166666666666667', '0.5125'],
 ['4', '0.7995833333333333', '0.424375', '0.0975', '0.13875'],
 ['4', '0.7995833333333333', '0.33', '0.08416666666666667', '0.0575'],
 ['13', '0.66375', '0.595625', '0.059166666666666666', '0.15875'],
 ['13', '0.66875', '0.483125', '0.0425', '0.05625']]
```

Task 3.5.14: Create a tensor containing the class indices. For compatibility with our plotting function it should be a $N\times 1$ tensor.

In [44]:
```
classes = torch.tensor([[int(row[0])] for row in label_data])

print("Tensor shape:", classes.shape)
print("First 5 elements:\n", classes[:5])
```

```
Tensor shape: torch.Size([30, 1])
First 5 elements:
 tensor([[ 4],
        [ 4],
        [ 4],
        [13],
        [13]])
```

Task 3.5.15: Load the bounding box coordinates into a $N\times 4$ tensor.

In [62]:
```
bboxes = torch.tensor([[float(el) for el in row[1:]] for row in label_data])

print("Tensor shape:", bboxes.shape)
print("First 5 elements:\n", bboxes[:5])
```

```
Tensor shape: torch.Size([30, 4])
First 5 elements:
 tensor([[0.8000, 0.7437, 0.2117, 0.5125],
        [0.7996, 0.4244, 0.0975, 0.1388],
        [0.7996, 0.3300, 0.0842, 0.0575],
        [0.6637, 0.5956, 0.0592, 0.1587],
        [0.6687, 0.4831, 0.0425, 0.0562]])
```

All of these coordinates are normalized by the width or height, as appropriate. This won't work with transformations like rotation, which need the same units used on each axis.

In [63]:
```
sample_width, sample_height = sample_image.size

scale_factor = torch.tensor([sample_width, sample_height, sample_width, sample_heig

bboxes_pixels = bboxes * scale_factor

print("Tensor shape:", bboxes_pixels.shape)
print("First 5 elements:\n", bboxes_pixels[:5])
```

```
Tensor shape: torch.Size([30, 4])
First 5 elements:
 tensor([[960.0000, 595.0000, 254.0000, 410.0000],
        [959.5000, 339.5000, 117.0000, 111.0000],
        [959.5000, 264.0000, 101.0000,  46.0000],
        [796.5000, 476.5000,  71.0000, 127.0000],
        [802.5000, 386.5000,  51.0000,  45.0000]])
```

Task 3.5.16: Convert the bounding box coordinates to pixel units.

In order for the transformations to know how to transform the bounding boxes, they need to know that the coordinates represent the centers and dimensions of the boxes. This is done by creating a special `BoundingBoxes` tensor. This type has a `format` attribute. By setting this to `"CXCYWH"`, we're telling it that the columns represent the Center *X* coordinate, the Center *Y* coordinate, the Width, and the Height. This tensor also is given the size of the image, so it doesn't need to look that up for transformations.

In [64]:
```
bboxes_tv = torchvision.tv_tensors.BoundingBoxes(
    bboxes_pixels,
    format="CXCYWH",
    # Yes, that's right.  Despite using width x height everywhere
    # else, here we have to specify the image size as height x
    # width.
    canvas_size=(sample_height, sample_width),
)

print("Tensor type:", type(bboxes_tv))
print("First 5 elements:\n", bboxes_tv[:5])
```

```
Tensor type: <class 'torchvision.tv_tensors._bounding_boxes.BoundingBoxes'>
First 5 elements:
 tensor([[960.0000, 595.0000, 254.0000, 410.0000],
        [959.5000, 339.5000, 117.0000, 111.0000],
        [959.5000, 264.0000, 101.0000,  46.0000],
        [796.5000, 476.5000,  71.0000, 127.0000],
        [802.5000, 386.5000,  51.0000,  45.0000]])
```

Let's double check that we did all of those conversions correctly. Do the bounding boxes line up with the correct objects?

In [65]:
```
plot_with_bboxes(sample_torch, bboxes, classes)
```

Out[65]:

If everything looks good, we'll introduce some transformations. The first one will be a horizontal flip. Many everyday objects have bilateral symmetry (or nearly so), so a flipped image will still have the same object classes in it. This makes a horizontal flip a good data augmentation transformation.

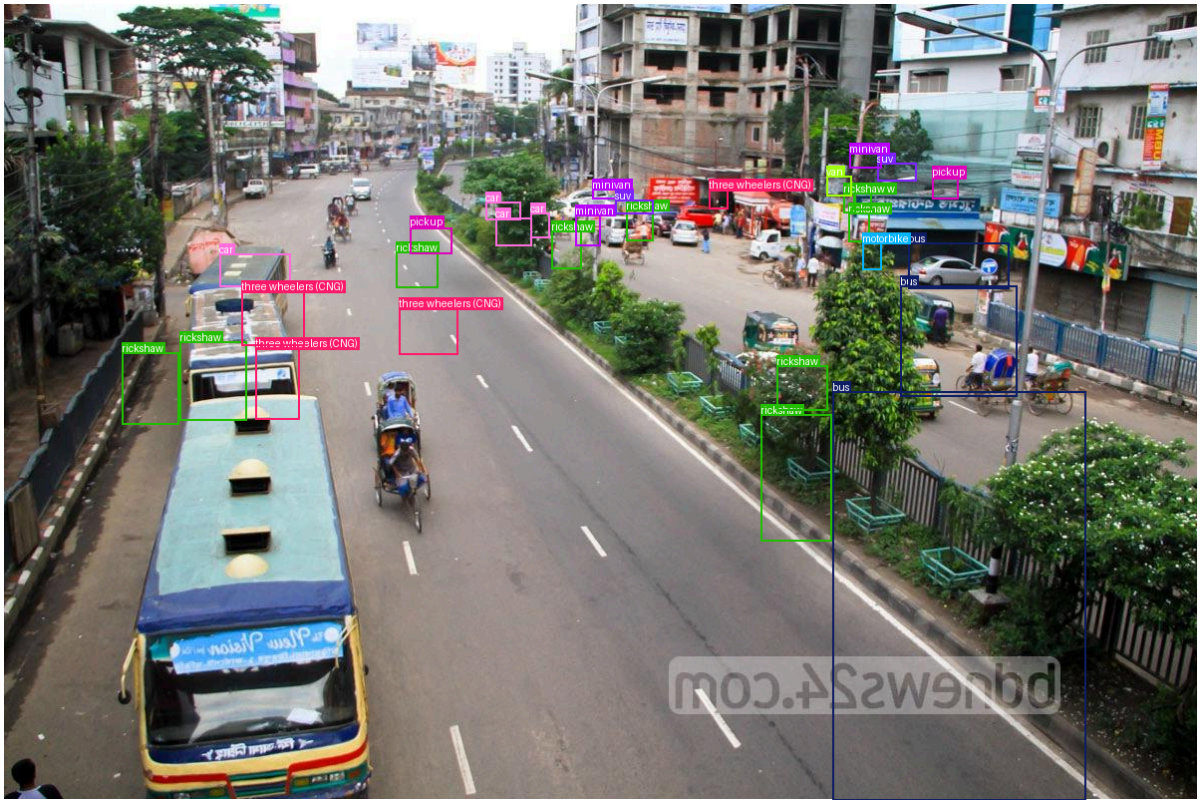
(In contrast, up/down symmetry is less common. A vertical flip is generally not as useful, unless you need to recognize upside-down objects.)

The transforms version 2 module has a `RandomHorizontalFlip` transformation. This takes the probability of a flip as an argument.

Task 3.5.17: Use the `RandomHorizontalFlip` transformation to flip the sample image. Set `p=1` to ensure that the flip happens.

In [66]:
```
flipped = v2.RandomHorizontalFlip(p=1)(sample_torch)

plot_with_bboxes(flipped, bboxes_tv, classes)
```

Out[66]:



The image has flipped, but the bounding boxes are still in their original locations. Note that the bus is now in the bottom left of the image. Its bounding box is still at the bottom right, and it now contains some asphalt, planters and trees. If we fed this into a model, it would make the model worse, by confusing it as to what a bus looks like.

So, we need to transform the bounding box coordinates consistent with the image transformation. The Torchvision version 2 transformations can take multiple arguments. They perform the same transformation on all of the arguments, returning a transformed version of each. They also understand how to correctly transform the `BoundingBoxes` tensors, depending on their type.

Task 3.5.18: Use `RandomHorizontalFlip` to flip both the sample image and its bounding boxes. Check that they line up correctly now.

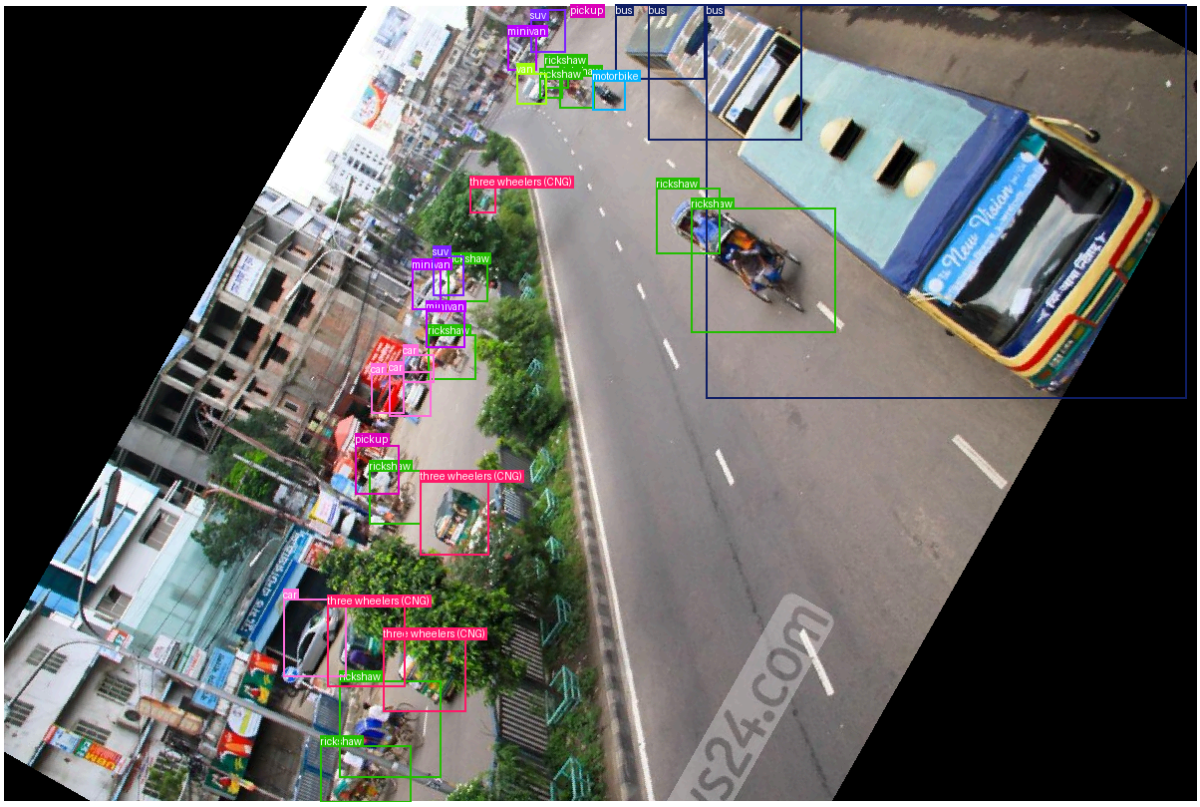
In [68]:
```
flipped, flipped_bboxes = v2.RandomHorizontalFlip(p=1)(sample_torch, bboxes_tv)

plot_with_bboxes(flipped, flipped_bboxes, classes)
```

Out[68]:

Task 3.5.19: Apply the `RandomRotation` transformation. This takes an argument of the maximum number of degrees to rotate the image. Set it to 90.

In [72]:
```
rotated, rotated_bboxes = v2.RandomRotation(degrees=90)(sample_torch, bboxes_tv)

plot_with_bboxes(rotated, rotated_bboxes, classes)
```

Out[72]:



Multiple augmentation techniques can be chained together to produce even more diversity in the training images. Within Torchvision, this can be accomplished by the `Compose` element.

Task 3.5.20: Create an augmentation pipeline that combines the `RandomHorizontalFlip` with the `RandomRotation`. This time, set the probability of the flip to 50%.

In [75]:
```
transforms = v2.Compose(
    [
        v2.RandomHorizontalFlip(p=0.5),
        v2.RandomRotation(degrees=90)
    ]
)

transformed, transformed_bboxes = transforms(sample_torch, bboxes_tv)

plot_with_bboxes(transformed, transformed_bboxes, classes)
```

Out[75]:

There are a large number of transformations that can be used for data augmentation. Scroll through the documentation to get a view of the range of possibilities.

In addition to the transforms we've already used, note:

- `RandomResizedCrop` will randomly crop the image down, and then it resizes the output to a set dimension.
- `ColorJitter` can randomly adjust the brightness, contrast, saturation, and hue of the image, within specified ranges.

Task 3.5.21: Create an augmentation pipeline that applies several of these transformations. Choose reasonable values for the parameters. Check that the bounding boxes are transformed correctly through this.

> There's no right answer. A good choice of augmentations depends heavily on the problem you're trying to model.

In [84]:
```
transforms = v2.Compose(
    [
        v2.RandomHorizontalFlip(p=0.5),
        v2.RandomRotation(degrees=90),
        v2.RandomResizedCrop(size=(400, 500)),
        v2.ColorJitter()
    ]
)

transformed, transformed_bboxes = transforms(sample_torch, bboxes_tv)
plot_with_bboxes(transformed, transformed_bboxes, classes)
```

Out[84]:

You can run the transformation several times to see the different types of images that result. This greater diversity of training images will help models learn to generalize instead of memorizing during training.
