ECE 60146: HW 6

Sidd Subramanyam

February 29th, 2024

Summary

This report outlines the implementation of data loading, training, and evaluation logic for a machine learning model aimed at object detection tasks within the COCO dataset. The methodology encompasses data preprocessing and augmentation, dataset generation, model architecture design, training loop execution, and evaluation strategies. We carefully address the nuances associated with managing image data, bounding box annotations, and model predictions.

1 Data Loading and Preprocessing

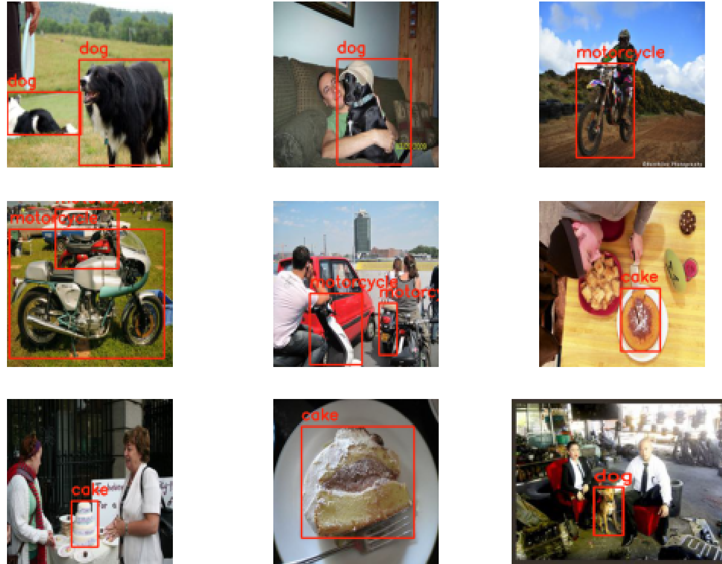
The initial step involves loading the COCO dataset annotations using the `pycocotools` library, specifying paths to the training and validation annotation files, and loading these datasets into memory, a vital step for accessing necessary metadata for image and annotation retrieval.

1.1 Dataset Generation

The function `generate_dataset` preprocesses images and annotations through several steps:

1. **Category Filtering:** Categories are filtered to a subset (e.g., "motorcycle", "dog", "cake"), creating a mapping between COCO category IDs and class labels.
2. **Image and Annotation Selection:** Images containing specified categories are selected, and annotations are filtered based on area to include only significant objects.
3. **Image Resizing and Annotation Transformation:** Images are resized to 256x256 pixels, and bounding box annotations are scaled accordingly to simplify training and inference.
4. **Dataset Saving:** Processed images and annotations are saved to disk to facilitate efficient data loading during training.

Figure 1: Training Dataset Images



2 Training and Evaluation Logic

The training process involves a custom PyTorch dataset class, `MyDataset`, for loading processed images and annotations, and a training function that utilizes different loss functions for model optimization.

2.1 Network Architecture

The network architecture, as implemented in the provided code, is designed for YOLO (You Only Look Once) object detection. It is a convolutional neural network (CNN) that incorporates both traditional convolutional layers and custom skip blocks for efficient feature extraction and object detection. **This network architecture heavily borrows code and architecture from Professor Kak's YoloLogic network, with multiple skip block layers**

2.1.1 Skip Block

The `SkipBlock` class is a modular building block that enhances the network's ability to propagate gradients during training, thanks to its skip connections. These connections allow the network to learn identity mappings, which stabilize the training of deep networks. Each skip block consists of two convolutional layers (`convo1` and `convo2`), each followed by batch normalization (`bn1` and `bn2`) and ReLU activation. The block optionally includes a downsampling mechanism via convolutional layers with a stride of 2 (`downsampler1` and `downsampler2`),

reducing the spatial dimensions of the feature maps to capture more abstract representations.

2.1.2 NetForYolo

The `NetForYolo` class defines the overall network structure. The network begins with two initial convolutional layers (`conv1` and `conv2`) followed by max-pooling, designed to extract low-level features from the input images. The core of the network is composed of several `SkipBlock` instances, organized into groups based on their feature map sizes (64, 128, and 256 channels). These blocks are arranged to gradually increase the depth of the feature representations while incorporating skip connections to preserve spatial information and reduce information loss.

The network uses downsampling skip blocks (`skip64ds`, `skip128ds`, and `skip256ds`) to halve the dimensions of the feature maps, effectively increasing the receptive field and allowing the network to capture more global features relevant for object detection. Transition skip blocks (`skip64to128` and `skip128to256`) increase the channel dimensions, enabling the network to process and combine information across different scales.

The final part of the network flattens the output of the last skip block and passes it through a fully connected sequence (`fc_seqn`). This sequence consists of linear layers and ReLU activations, culminating in an output layer sized to match the flattened YOLO tensor representation. This design allows the network to predict object classes and bounding boxes directly from the input images.

2.1.3 Design Considerations

The architecture is specifically tailored for YOLO object detection, where each YOLO vector is of size $5 + C$ (C being the number of classes). The network assumes a flattened YOLO tensor as input, accommodating the specific dimensions required for multi-instance detection tasks. The depth of the network can be adjusted (tested values include 8, 10, 12, 14, and 16), allowing for flexibility in the model's capacity and computational requirements.

3 Training Loop

3.1 Initialization

Before entering the training epochs, the network is set to training mode, and the necessary loss functions are initialized:

- `objectCriterion` for binary cross-entropy loss to handle objectness prediction, Binary Cross-Entropy (BCE),
- `classCriterion` for cross-entropy loss to manage classification tasks, **Classification Loss:** Cross-Entropy (CE) for multi-class classification tasks,

- `bboxCriterion` for mean squared error loss to refine bounding box predictions, **Bounding Box Regression Loss:** Mean Squared Error (MSE) for bounding box coordinates

The network is transferred to a CUDA device for GPU acceleration, and an Adam optimizer is prepared with specified learning rates and betas.

3.2 Epoch Iteration

Training proceeds over a fixed number of epochs, within which the dataset is iterated batch by batch. Each batch contains inputs, ground truth (gt) for YOLO vectors, and the number of objects (numObjs) present in the images.

3.3 Loss Calculation

For each batch, the network's predictions are compared against the ground truth using the designated loss functions. The losses are calculated as follows:

1. **Objectness Loss:** Separately calculated for positive (object present) and negative (object absent) anchor boxes using the `objectCriterion`. This distinction helps in balancing the learning between detecting objects and identifying background. Calculated as:

$$\mathrm{BCE}(p, y) = -y \log(p) - (1 - y) \log(1 - p)$$

2. **Classification Loss:** Applied only to the positive anchor boxes where objects are present, using `classCriterion` to ensure the correct class is predicted for each detected object.

$$\mathrm{CE}(p, y) = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

3. **Bounding Box Loss:** Computed for positive anchor boxes to refine the bounding box coordinates, using `bboxCriterion`: :

$$\mathrm{MSE}(p, y) = \frac{1}{n} \sum_{i=1}^{n} (p_i - y_i)^2$$

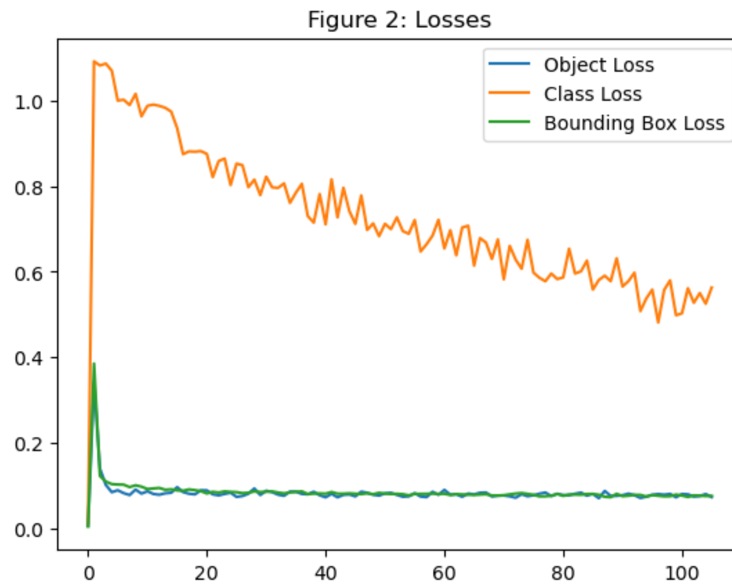
The total loss is a weighted sum of these components, emphasizing object presence, accurate classification, and precise localization.

3.4 Optimization Step

Following the backward propagation of the total loss, the optimizer updates the network parameters. This step is crucial for learning the correct weights to minimize the loss across all tasks (object detection, classification, and bounding box regression).

3.5 Evaluation

Evaluation is conducted on a separate dataset, employing non-maximum suppression (NMS) to filter overlapping bounding boxes based on confidence scores, crucial for assessing model accuracy in object detection.



4 Results

Figure 4: Perfomance on Testing Dataset

Overall, the performance was not that great. Several multi-instance test images are provided, and more times than not the network is not able to detect all of them. The best case is the bottom right where multiple motorcycle objects are detected, however, they are poorly localized. Single object localization performance is somewhat average. There are also some errors in the general object classification too of the objects.

5 Code

```

# %%
from pycocotools.coco import COCO

base_path = "C:/Users/Sidd/Documents/ECE 60146/data/annotations/"
train_ann_file = f"{base_path}instances_train2017.json"
val_ann_file = f"{base_path}instances_val2017.json"

coco_train = COCO(train_ann_file)
coco_val = COCO(val_ann_file)

# %%
import os
import json
from PIL import Image

```

```
17 # Assuming class_list is defined elsewhere as per the previous code
     segment.
18 # class_list = ["motorcycle", "dog", "cake"]
19
20 def generate_dataset(coco, src_path, dst_path, annotations_path):
21     # Simplify category mapping and inverse lookup creation
22     cat_ids = coco.getCatIds(catNms=class_list)
23     categories = coco.loadCats(cat_ids)
24     coco_labels_inverse = {c['id']: class_list.index(c['name']) for
      c in categories}
25
26     img_ids = set()
27     for cat_id in cat_ids:
28         img_ids.update(coco.getImgIds(catIds=cat_id))
29
30     imgs = coco.loadImgs(list(img_ids))
31     annotations = []
32     for img in imgs:
33         ann_ids = coco.getAnnIds(imgIds=img["id"], catIds=cat_ids,
     iscrowd=False)
34         anns = [ann for ann in coco.loadAnns(ann_ids) if ann['area'
     ] > 4096]
35         if not anns:
36             continue
37
38         pic = Image.open(os.path.join(src_path, img["file_name"]))
39         resized_pic = pic.resize((256, 256))
40         filename = f'{len(annotations):05}.jpg'
41
42         scale_x = resized_pic.size[0] / pic.size[0]
43         scale_y = resized_pic.size[1] / pic.size[1]
44
45         new_anns = [{
46             "bbox": [int(ann["bbox"][0] * scale_x), int(ann["bbox"
     ][1] * scale_y),
47                      int(ann["bbox"][2] * scale_x), int(ann["bbox"
     ][3] * scale_y)],
48             "category": coco_labels_inverse[ann["category_id"]]
49         } for ann in anns]
50
51         resized_pic.save(os.path.join(dst_path, filename))
52         annotations.append({"file_name": filename, "ann": new_anns
     })
53
54     with open(annotations_path, "w") as file:
55         json.dump(annotations, file, indent=4)
56     print(f"num images in {annotations_path}: {len(annotations)}")
57
58
59 # %%
60 generate_dataset(coco_train, "C:/Users/Sidd/Documents/ECE 60146/
     data/train2017", "C:/Users/Sidd/Documents/ECE 60146/data/
     traindir", "C:/Users/Sidd/Documents/ECE 60146/data/train_ann.
     json")
61 generate_dataset(coco_val, "C:/Users/Sidd/Documents/ECE 60146/data/
     val2017", "C:/Users/Sidd/Documents/ECE 60146/data/valdir", "C:/
     Users/Sidd/Documents/ECE 60146/data/val_ann.json")
```

```
62
63 # %%
64 import cv2
65 import numpy as np
66 import matplotlib.pyplot as plt
67 import os
68 import json
69 from PIL import Image
70
71 class_list = ["motorcycle", "dog", "cake"]
72
73 labels_path = "C:/Users/Sidd/Documents/ECE 60146/data/train_ann.
       json"
74 images_dir = "C:/Users/Sidd/Documents/ECE 60146/data/traindir/"
75
76 with open(labels_path, "r") as file:
77     labels = json.load(file)
78
79 # Dynamically adjust figure size based on the number of classes
80 fig_width = len(class_list) * 4  # 4 inches per class image
81 fig_height = 12  # 12 inches tall to accommodate larger images
82 plt.figure(figsize=(fig_width, fig_height))
83
84 class_images_count = {class_name: 0 for class_name in class_list}
85 image_displayed = {class_name: [] for class_name in class_list}  #
       Track displayed images per class
86
87 for label in labels:
88     if all(count >= 3 for count in class_images_count.values()):
89         break  # Exit loop if we have 3 images for each class
90
91     # Extract class names from the current image's annotations
92     current_image_classes = [class_list[ann["category"]] for ann in
        label["ann"] if class_list[ann["category"]] in class_list]
93
94     # Check if the current image has a class with less than 3
       images displayed
95     for class_name in set(current_image_classes):
96         if class_images_count[class_name] < 3 and label["file_name"
       ] not in image_displayed[class_name]:
97             # Update counters and lists
98             class_images_count[class_name] += 1
99             image_displayed[class_name].append(label["file_name"])
100
101             # Load and display the image
102             pic_path = os.path.join(images_dir, label["file_name"])
103             pic = Image.open(pic_path)
104             image = np.array(pic, dtype=np.uint8)
105
106             # Draw bounding boxes and class names
107             for ann in label["ann"]:
108                 if class_list[ann["category"]] == class_name:
109                     [x, y, w, h] = ann["bbox"]
110                     cv2.rectangle(image, (x, y), (x + w, y + h),
       (255, 36, 12), 2)
111                     cv2.putText(image, class_name, (x, y - 10), cv2
       .FONT_HERSHEY_SIMPLEX, 0.8, (255, 36, 12), 2)
```

```
112
113             # Plotting
114             ax = plt.subplot(3, len(class_list), class_images_count
    [class_name] + (class_list.index(class_name) * 3))
115             plt.imshow(image)
116             plt.title(f"{class_name} {class_images_count[class_name
    ]}")
117             plt.axis('off')
118
119 plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
120 plt.show()
121
122
123 # %%
124 import torchvision.transforms as tvt
125 import torchvision.ops as tops
126 import torch
127
128 num_cells = 8
129
130 class MyDataset(torch.utils.data.Dataset):
131     def __init__(self, ann_file, root_dir, augment=False):
132         super().__init__()
133         with open(ann_file, "r") as file:
134             self.labels = json.loads(file.read())
135         self.root_dir = root_dir
136         self.augment = augment
137
138         if augment:
139             self.transform = tvt.Compose([
140                 tvt.ToTensor(),
141                 tvt.ColorJitter(brightness=.2, hue=.1)
142             ])
143         else:
144             self.transform = tvt.ToTensor()
145
146         x, y = np.meshgrid(np.arange(num_cells), np.arange(
    num_cells))
147         self.pts = np.vstack((x.ravel(), y.ravel())).T * 256/
    num_cells
148         self.create_template_anchor_boxes()
149
150     def __len__(self):
151         return len(self.labels)
152
153     def __getitem__(self, index):
154         filename = self.labels[index]["file_name"]
155
156         anchorIdx = list()
157         anchorBoxIdx = list()
158         bboxData = list()
159         labels = list()
160
161         pic = Image.open(os.path.join(self.root_dir, filename)).
    convert("RGB")
162         img = self.transform(pic)
163
```

```
164         gt = torch.zeros(5, num_cells * num_cells, 5 + len(
    class_list))
165
166         for ann in self.labels[index]["ann"]:
167             bbox = ann["bbox"]
168             bbox = np.array([bbox[0], bbox[1], bbox[0] + bbox[2],
    bbox[1] + bbox[3]])
169
170             center = np.array([bbox[0] + bbox[2], bbox[1] + bbox
    [3]]) / 2
171             diff = (center - 256/num_cells/2) - self.pts
172             anchor = np.argmin(np.einsum("ij,ij->i", diff, diff))
173             ptstogether = np.hstack((self.pts[anchor], self.pts[
    anchor]))
174             gtbbox = torch.unsqueeze(torch.tensor(bbox.flatten()),
    0)
175             anchorBox = np.argmax(tops.box_iou(self.anchor_boxes +
    ptstogether, gtbbox).numpy())
176
177             delx = diff[anchor][0] / self.cell_size
178             dely = diff[anchor][1] / self.cell_size
179             abox = self.anchor_boxes[anchorBox].numpy()
180             sigw = np.log(ann["bbox"][2] / (abox[2] - abox[0]))
181             sigh = np.log(ann["bbox"][3] / (abox[3] - abox[1]))
182
183             anchorIdx.append(anchor)
184             anchorBoxIdx.append(anchorBox)
185             bboxData.append([1, delx, dely, sigw, sigh])
186             labels.append(ann["category"])
187
188             label = np.zeros(len(class_list))
189             label[ann["category"]] = 1
190             gt[anchorBox][anchor] = torch.tensor([1, delx, dely,
    sigw, sigh, *label])
191
192         return img, gt, len(labels)
193
194     def create_template_anchor_boxes(self):
195         w = h = 256 / num_cells
196         self.cell_size = w
197
198         bboxes = list()
199         bboxes.append([0, 0,    w, h])
200         bboxes.append([-w, 0,   2*w, h])
201         bboxes.append([-2*w, 0, 3*w, h])
202         bboxes.append([0, -h,   w, 2*h])
203         bboxes.append([0, -2*h, w, 3*h])
204
205         self.anchor_boxes = torch.tensor(bboxes)
206
207
208 # %%
209 trainDataset = MyDataset("C:/Users/Sidd/Documents/ECE 60146/data/
    train_ann.json", "C:/Users/Sidd/Documents/ECE 60146/data/
    traindir")
210 valDataset = MyDataset("C:/Users/Sidd/Documents/ECE 60146/data/
    val_ann.json", "C:/Users/Sidd/Documents/ECE 60146/data/valdir")
```

```
211
212 trainDataloader = torch.utils.data.DataLoader(trainDataset, shuffle
    =True, batch_size=8)
213 valDataloader = torch.utils.data.DataLoader(valDataset, batch_size
    =14)
214
215 # %%
216 import sys
217 import torch
218 import torch.nn as nn
219 import torch.nn.functional as F
220
221 class SkipBlock(nn.Module):
222     """
223     This is a building-block class that I have borrowed from the
    DLStudio platform
224     """
225     def __init__(self, in_ch, out_ch, downsample=False,
    skip_connections=True):
226         super(SkipBlock, self).__init__()
227         self.downsample = downsample
228         self.skip_connections = skip_connections
229         self.in_ch = in_ch
230         self.out_ch = out_ch
231         self.convo1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding
    =1)
232         self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding
    =1)
233         self.bn1 = nn.BatchNorm2d(in_ch)
234         self.bn2 = nn.BatchNorm2d(out_ch)
235         self.in2out  =  nn.Conv2d(in_ch, out_ch, 1)
236         if downsample:
237             ##  Setting stride to 2 and kernel_size to 1 amounts to
     retaining every
238             ##  other pixel in the image --- which halves the size
    of the image:
239             self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride
    =2)
240             self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride
    =2)
241
242     def forward(self, x):
243         identity = x
244         out = self.convo1(x)
245         out = self.bn1(out)
246         out = nn.functional.relu(out)
247         out = self.convo2(out)
248         out = self.bn2(out)
249         out = nn.functional.relu(out)
250         if self.downsample:
251             identity = self.downsampler1(identity)
252             out = self.downsampler2(out)
253         if self.skip_connections:
254             if (self.in_ch == self.out_ch) and (self.downsample is
    False):
255                 out = out + identity
```

```
256             elif (self.in_ch != self.out_ch) and (self.downsample
    is False):
257                 identity = self.in2out( identity )
258                 out = out + identity
259             elif (self.in_ch != self.out_ch) and (self.downsample
    is True):
260                 out = out + torch.cat((identity, identity), dim=1)
261         return out
262 
263 class NetForYolo(nn.Module):
264     """
265     Recall that each YOLO vector is of size 5+C where C is the
    number of classes.  Since C
266     equals 3 for the dataset used in the demo code in the Examples
    directory, our YOLO vectors
267     are 8 elements long.  A YOLO tensor is a tensor representation
    of all the YOLO vectors
268     created for a given training image.  The network shown below
    assumes that the input to
269     the network is a flattened form of the YOLO tensor.  With an 8-
    element YOLO vector, a
270     6x6 gridding of an image, and with 5 anchor boxes for each cell
     of the grid, the
271     flattened version of the YOLO tensor would be of size 1440.
272 
273     In Version 2.0.6 of the YOLOLogic module, I introduced a new
    loss function for this network
274     that calls for using nn.CrossEntropyLoss for just the last C
    elements of each YOLO
275     vector. [See Lines 64 through 83 of the code for "
    run_code_for_training_multi_instance_
276     detection()" for how the loss is calculated in 2.0.6.]  Using
    nn.CrossEntropyLoss
277     required augmenting the last C elements of the YOLO vector with
     one additional
278     element for the purpose of representing the absence of an
    object in any given anchor
279     box of a cell.
280 
281     With the above mentioned augmentation, the flattened version of
     a YOLO tensor is
282     of size 1620.  That is the reason for the one line change at
    the end of the
283     constructor initialization code shown below.
284     """
285     def __init__(self, skip_connections=True, depth=8):
286         super(NetForYolo, self).__init__()
287         if depth not in [8,10,12,14,16]:
288             sys.exit("This network has only been tested for 'depth'
     values 8, 10, 12, 14, and 16")
289         self.depth = depth // 2
290         self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
291         self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
292         self.pool = nn.MaxPool2d(2, 2)
293         self.bn1  = nn.BatchNorm2d(64)
294         self.bn2  = nn.BatchNorm2d(128)
295         self.bn3  = nn.BatchNorm2d(256)
```

```
296         self.skip64_arr = nn.ModuleList()
297         for i in range(self.depth):
298             self.skip64_arr.append(SkipBlock(64, 64,
    skip_connections=skip_connections))
299         self.skip64ds = SkipBlock(64,64,downsample=True,
    skip_connections=skip_connections)
300         self.skip64to128 = SkipBlock(64, 128, skip_connections=
    skip_connections )
301         self.skip128_arr = nn.ModuleList()
302         for i in range(self.depth):
303             self.skip128_arr.append(SkipBlock(128,128,
    skip_connections=skip_connections))
304         self.skip128ds = SkipBlock(128,128, downsample=True,
    skip_connections=skip_connections)
305         self.skip128to256 = SkipBlock(128, 256, skip_connections=
    skip_connections )
306         self.skip256_arr = nn.ModuleList()
307         for i in range(self.depth):
308             self.skip256_arr.append(SkipBlock(256,256,
    skip_connections=skip_connections))
309         self.skip256ds = SkipBlock(256,256, downsample=True,
    skip_connections=skip_connections)
310         self.fc_seqn = nn.Sequential(
311             nn.Linear(8192, 4096),
312             nn.ReLU(inplace=True),
313             nn.Linear(4096, 2048),
314             nn.ReLU(inplace=True),
315             nn.Linear(2048, 1620)
316         )
317 
318     def forward(self, x):
319         x = self.pool(torch.nn.functional.relu(self.conv1(x)))
320         x = nn.MaxPool2d(2,2)(torch.nn.functional.relu(self.conv2(x
    )))
321         for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
322             x = skip64(x)
323         x = self.skip64ds(x)
324         for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
325             x = skip64(x)
326         x = self.bn1(x)
327         x = self.skip64to128(x)
328         for i,skip128 in enumerate(self.skip128_arr[:self.depth
    //4]):
329             x = skip128(x)
330         x = self.bn2(x)
331         x = self.skip128ds(x)
332         x = x.view(-1, 8192 )
333         x = self.fc_seqn(x)
334         return x
335 
336 model = NetForYolo()
337 num_layers = len(list(model.parameters()))
338 print("Total number of learnable layers:", num_layers)
339 
340 # %%
341 import torch
342 import torchvision.ops as ops
```

```
343
344 def training(model, loader):
345     # Switch model to training mode and set device
346     model.train()
347     device = torch.device('cuda:0' if torch.cuda.is_available()
    else 'cpu')
348     model.to(device)
349
350     # Define loss functions
351     obj_loss_fn = torch.nn.BCEWithLogitsLoss(pos_weight=torch.
    tensor([5.0])).to(device)
352     cls_loss_fn = torch.nn.CrossEntropyLoss().to(device)
353     bbox_loss_fn = torch.nn.MSELoss().to(device)
354
355     # Initialize optimizer
356     optim = torch.optim.Adam(model.parameters(), lr=0.001, betas
    =(0.9, 0.99))
357     epochs = 10
358
359     # For logging
360     losses = []
361
362     for ep in range(epochs):
363         print(f"Epoch {ep} start")
364         running_loss = [0.0, 0.0, 0.0]
365         for i, (X, y, _) in enumerate(loader):
366             X, y = X.to(device), y.to(device)
367             optim.zero_grad()
368             pred = model(X)
369
370             # Compute mask for objects and background
371             obj_mask = y[:,:,:,0] == 1
372             no_obj_mask = ~obj_mask
373
374             # Calculate object and no-object loss together using
    BCEWithLogitsLoss
375             obj_loss = obj_loss_fn(pred[...,0], y[...,0])
376
377             # Calculate class loss only for objects
378             cls_loss = cls_loss_fn(pred[obj_mask][:, 5:], torch.
    argmax(y[obj_mask][:, 5:], dim=1))
379
380             # Calculate bounding box loss only for objects
381             bbox_loss = bbox_loss_fn(pred[obj_mask][:, 1:5], y[
    obj_mask][:, 1:5])
382
383             # Adjust loss weights directly in the calculation
384             total_loss = obj_loss + cls_loss + 10 * bbox_loss
385             total_loss.backward()
386             optim.step()
387
388             # Update running losses for logging
389             running_loss[0] += obj_loss.item()
390             running_loss[1] += cls_loss.item()
391             running_loss[2] += bbox_loss.item()
392
393             if (i + 1) % 100 == 0:
```

```
394                 avg_loss = [x / 100 for x in running_loss]
395                 print(f"Iter {i+1}: Obj Loss: {avg_loss[0]}, Cls
    Loss: {avg_loss[1]}, Bbox Loss: {avg_loss[2]}")
396                 losses.append(avg_loss)
397                 running_loss = [0.0, 0.0, 0.0]
398 
399         print(f"Epoch {ep} complete: Avg Losses: {losses[-1] if
    losses else 'N/A'}")
400 
401     return torch.tensor(losses).transpose(0, 1)
402 
403 # %%
404 model = NetForYolo(3)
405 losses = training(model, trainDataloader)
406 
407 # %%
408 plt.plot(losses[1])
409 plt.legend(["Class"])
410 plt.title("Figure 2: Class Loss")
411 plt.show()
412 
413 
414 plt.plot(losses[[0,2]].T)
415 plt.legend(["obj","box"])
416 plt.title("Figure 3: Other Losses")
417 plt.show()
418 
419 # %%
420 def convert_to_bboxes(output, indexes):
421     confidences = list()
422     bboxes = list()
423     classes = list()
424 
425     for i, index in enumerate(indexes):
426         _, box, anchor = index
427         abox = valDataset.anchor_boxes[box]
428 
429         ow = abox[2] - abox[0]
430         oh = abox[3] - abox[1]
431         w = ow * np.exp(output[i][3]) / 2
432         h = oh * np.exp(output[i][4]) / 2
433 
434         c = np.array([output[i][1], output[i][2]])*valDataset.
    cell_size + valDataset.pts[anchor] + 256/num_cells/2
435 
436         bbox = np.array([c[0]-w, c[1]-h, c[0]+w, c[1]+h])
437         bbox = np.clip(bbox, 0, 255).astype(np.uint8)
438         bboxes.append(bbox)
439 
440         classes.append(np.argmax(output[i][5:]))
441         confidences.append(output[i][0])
442 
443     return confidences, np.array(bboxes), np.array(classes)
444 
445 def eval_on_dataset(dataset, title):
446     plt.figure()
447     fignum = 0
```

```
448     counts = {i: 0 for i, _ in enumerate(class_list)}
449 
450     with torch.no_grad():
451         model.eval()
452         device = torch.device('cuda')
453         model.to(device)
454         toPIL = tvt.ToPILImage()
455 
456         for data in dataset:
457             img, gt, numObj = data
458             idx = gt[:,:,0] == 1
459             gt_classes = torch.argmax(gt[idx][:,5:],1).numpy()
460 
461             shouldSkip = True
462             for annCls in gt_classes:
463                 if counts[annCls] < 3:
464                     shouldSkip = False
465                     counts[annCls] += 1
466                     break
467             if shouldSkip: continue
468             fignum += 1
469 
470             img = torch.unsqueeze(img, 0)
471             gt = torch.unsqueeze(gt, 0)
472             img = img.to(device)
473             output = model(img)
474 
475             output = output.cpu()
476             threshold = .99
477             idx = output[:,:,:,0] > threshold
478             while torch.nonzero(idx).shape[0] == 0:
479                 threshold -= .01
480                 idx = output[:,:,:,0] > threshold
481             print(threshold, torch.nonzero(idx).shape[0])
482 
483             output = output[idx]
484             indexes = torch.nonzero(idx)
485 
486 
487             scores, bboxes, classes = convert_to_bboxes(output,
    indexes)
488             keepidx = tops.nms(torch.tensor(bboxes).type(torch.
    FloatTensor), torch.tensor(scores), .2)
489             bboxes = bboxes[keepidx]
490             classes = classes[keepidx]
491             if len(keepidx) == 1:
492                 bboxes = [bboxes]
493                 classes = [classes]
494 
495             gtidx = gt[:,:,:,0] == 1
496             gtindexes = torch.nonzero(gtidx)
497             _, trueBboxes, truClasses = convert_to_bboxes(gt[gtidx
    ], gtindexes)
498 
499             image = toPIL(img[0].cpu())
500             image = np.array(image, dtype=np.uint8)
501 
```

```
502             for bbox, clas in zip(bboxes, classes):
503                 [x1, y1, x2, y2] = bbox
504                 image = cv2.rectangle(image, (x1,y1), (x2, y2),
    (36,255,12), 2)
505                 image = cv2.putText(image, class_list[clas], (x1,
    y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (36,255,12), 2)
506 
507             for bbox, clas in zip(trueBboxes, truClasses):
508                 [x1, y1, x2, y2] = bbox
509                 image = cv2.rectangle(image, (x1,y1), (x2, y2),
    (255,36,12), 2)
510                 image = cv2.putText(image, class_list[clas], (x1,
    y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255,36,12), 2)
511 
512             ax = plt.subplot(3,3,fignum)
513             plt.imshow(image)
514             ax.set_axis_off()
515 
516 
517             if fignum == 2:
518                 ax.set_title(title)
519             if fignum >= 9:
520                 break
521 
522     plt.axis("tight")
523     plt.show()
524 
525 eval_on_dataset(trainDataset, "Figure 4: Training Dataset Images")
```