Road Sign Detection

Students:
Derek Hines-Mohrman
Hanna Lee
Ray Zhao

Road Sign Detection

Student: Hanna, Derek, Ray

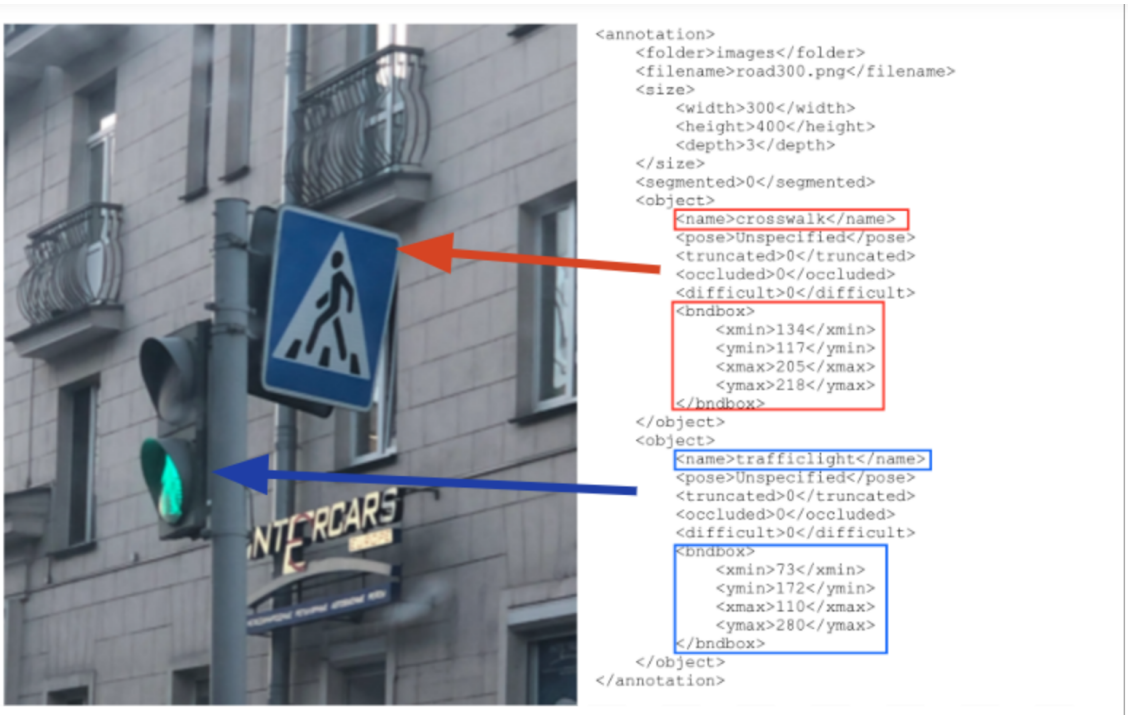
1. Exploring Datasets

This dataset contains 877 images of 4 distinct classes for the objective of road sign detection and bounding box annotations. (source: [kaggle](https://www.kaggle.com/datasets/andrewmvd/road-sign-detection)) For the purpose of getting hands dirty on the multiple object detection problem in computer vision, we chose a smaller dataset, rather than a massive dataset that would take a lot of time to train.

The classes are:
- Traffic Light
- Stop
- Speed Limit
- Crosswalk

1.1 Understand the dataset

The xml files in the “annotations” folder contain the information for each image file. For example, let’s look at the "road300.xml" file and the corresponding image, "road300.jpg". We can see that crosswalk and traffic light signs are in the image. Also, we can check the image classification labels on the xml file and their bounding boxes.



1.2 objects per class

Multiple object detection problem requires checking how many objects for each class exist in the dataset. Let's see how many objects in each class are in the training/ validation set. This dataset contains 877 images of 4 distinct classes. The total number of signs is around 1,200. We can assume that most images have only one traffic sign to detect. It consists of 877 images. Most images have single objects to detect. It's a pretty imbalanced dataset, with most images belonging to the speed limit class(0), but since we're more focused on the bounding box prediction, we can ignore the imbalance.

- **Traffic Light** (# of occurrences: 169)
- **Stop** (# of occurrences: 91)
- **Speed Limit** (# of occurrences: **783**)
- **Crosswalk** (# of occurrences: 199)

➜ Image: 877, Total signs: 1242

➜ Most image has only one sign

Objective

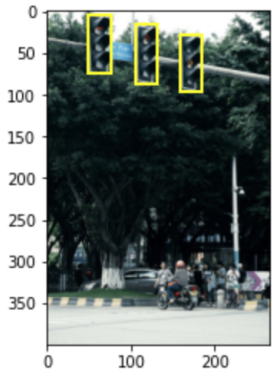
Detect object class and bounding boxes

"**Classification**" + "**Regression**"

1.2 Helper functions

We have created some helpful functions to draw the bounding box on the image. Each class has its own color of a bounding box. Let's take a look at how bounding boxes are displayed. As we can see, we have multiple road signs to detect in one image for some cases.

```
167 ./input/annotations/road300.xml  259 ./input/annotations/road40.xml
```



2. Data Preprocessing

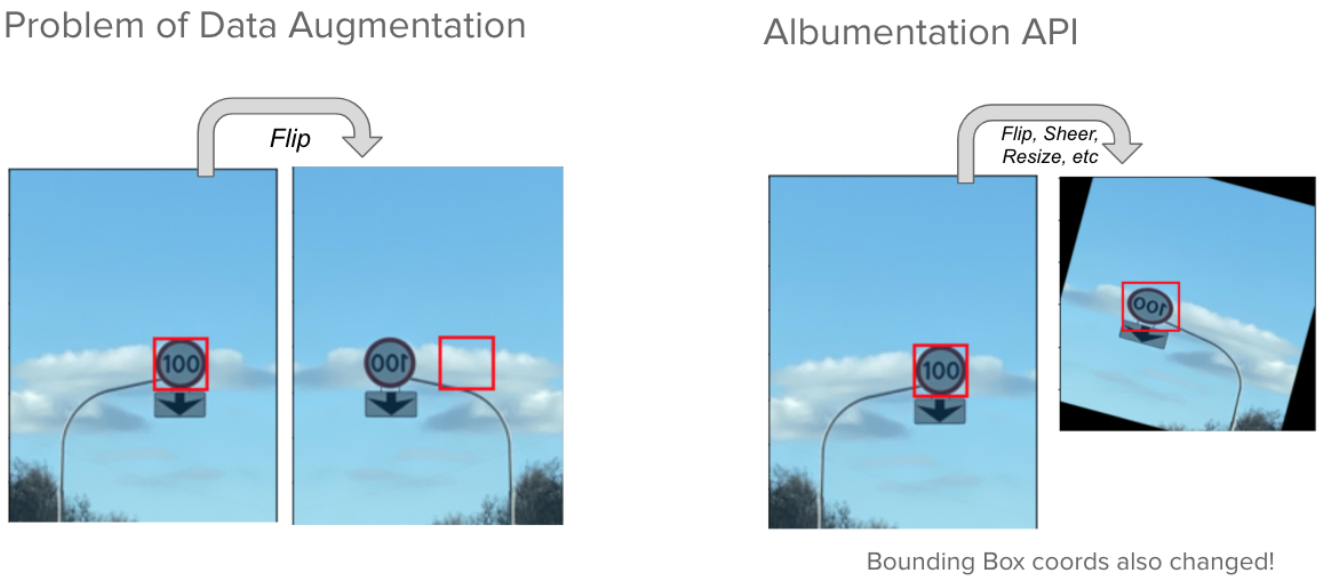
What is image augmentation and how it can improve the performance of deep neural networks? Deep neural networks require a lot of training data to obtain good results and prevent overfitting. However, it is often very difficult to get enough training samples. Image augmentation is a process of creating new training examples from existing ones. To make a new sample, you slightly change the original image. For instance, you could make a new image a little brighter; you could cut a piece from the original image; you could make a new image by mirroring the original one, etc. Augmentations help to fight to overfit and improve the performance of deep neural networks for computer vision tasks.

2.1 Problems of Data Augmentation

Image augmentation for multiple object detection problems needs quite a different approach from image classification problems. The RandomHorizontalFlip performs a horizontal inversion with a probability of p. After the transformation, flip and resize, the image changes the coordinates of pixels. If bounding box coordinates stay the same, that can be a problem as you see in the image. We need to change the coordinates of bounding boxes as we transform the images. To handle this, we are using the Albumentation API.

2.2 Bounding Box Augmentation

The best part is that image augmentations libraries make it possible to add image augmentations to any computer vision pipeline with minimal effort. We are using the Albumentations library for data augmentation.
We can see that the problem of bounding box coordination after the transformation is solved!



2.2 Train/Valid data separation

We have a total of 877 images. We will use 170 images as a validation dataset.

```
Training/Valid data Seperation
# of training image set:  707
# of training annotation set:  707
# of validation image set:  170
# of validation annotation set:  170
```

2.3 Labeling

For the training in the neural net, we would like to change the labeling from words to digits. For example,

- speed limit --> 0
- stop --> 1
- crosswalk --> 2
- traffic light --> 3

2.4 Data Augmentation for Training / Validation set

For the training image, we applied transformations such as resize, affine transformation, random horizontal filp, and and brightness contrast. For the validation, we only applied the resize.

The Torchvision Dataset class loads images through the **__getitem__** method. It loads and returns a sample from the dataset at the given index 'idx'. Based on the index, it identifies the image's location on disk, converts that to a tensor using **__read_image__**, retrieves the corresponding label from the xml data in target, calls the transform functions on them (if applicable), and returns the tensor image and corresponding label in a tuple.

Augmentation is performed according to the rule stored in the transform parameter.

```
dataset = SignalDataset('./input/images/', transform=train_transform)
test_dataset = SignalDataset('./test_images/', transform=valid_transform)

#Batch Size - the number of data samples propagated through the network before the parameters a
data_loader = torch.utils.data.DataLoader(dataset, batch_size=16, collate_fn=collate_fn)
test_data_loader = torch.utils.data.DataLoader(test_dataset, batch_size=2, collate_fn=collate_f
```

3. Transfer Learning: RetinaNet

Quoting these notes.

" In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pre-train a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest. "

[Fine-tuning the RetinaNet]

- Backbone: Pretrained on Coco
- Replace classifier and regression to our dataset
- Retrain on top of the RetinaNet
- Fine-tune the weights by continuing the backpropagation

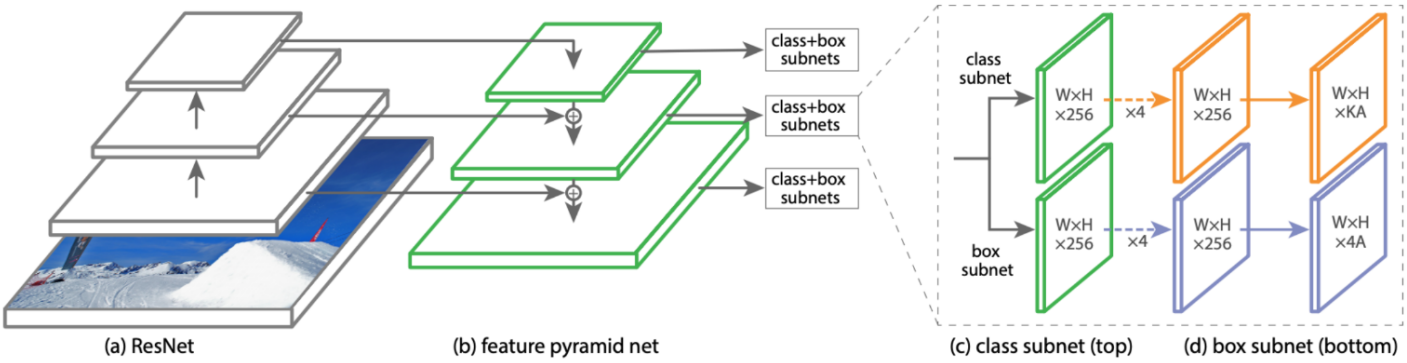


Figure 3. The one-stage **RetinaNet** network architecture uses a Feature Pyramid Network (FPN) [20] backbone on top of a feedforward ResNet architecture [16] (a) to generate a rich, multi-scale convolutional feature pyramid (b). To this backbone RetinaNet attaches two subnetworks, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d). The network design is intentionally simple, which enables this work to focus on a novel focal loss function that eliminates the accuracy gap between our one-stage detector and state-of-the-art two-stage detectors like Faster R-CNN with FPN [20] while running at faster speeds.

3.1 Load Retina Net

Torchvision provides deep learning models for solving computer vision tasks. We will use the torchvision.models module to import RetinaNet. Since there are 4 classes in the Road Sign Detection dataset, the num_classes parameter is defined as 4.

In order to perform transfer learning, the backbone structure comes with pre-trained weights. We want to start from a model pre-trained on COCO, which is famous for its object detection dataset, and then finetune it for our particular classes.

```
# Number of classes = 4 (4 classes)
retina = torchvision.models.detection.retinanet_resnet50_fpn(num_classes=4,
```

3.2 Transfer Learning

Training a model is an iterative process; in each iteration, the model makes a guess about the output, calculates the error in its guess (loss), collects the derivatives of the error with respect to its parameters, and optimizes these parameters using gradient descent.

hyperparameters

- Learning Rate(lr): How much to update model parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training.
- Momentum: Momentum is introduced to speed up the learning process, especially for the gradient with high curvature, small but consistent, which can accelerate the learning process. The main idea of momentum is to accumulate the moving average of previous gradients decaying exponentially.
- Weight Decay: The weight_decay parameter adds a L2 penalty to the cost which can effectively lead to to smaller model weights.

Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. Optimization algorithms define how this process is performed. All optimization logic is encapsulated in the optimizer object. Here, we use the SGD optimizer; additionally, there are many different optimizers available in PyTorch such as ADAM and RMSProp.

We initialize the optimizer by registering the model's parameters that need to be trained and passing in the learning rate, momentum, and wieght_decay hyperparameters.

```
# hyperparameters
params = [p for p in retina.parameters() if p.requires_grad] # select parameters
optimizer = torch.optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)
```

Loss Function

When presented with some training data, our untrained network is likely not to give the correct answer. Loss function measures the degree of dissimilarity of the obtained result to the target value, and it is the loss function that we want to minimize during training. To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value.

```python
# Training a model is an iterative process
for epoch in range(num_epochs):
    start = time.time()
    retina.train() # Training

    i = 0
    epoch_loss = 0
    for images, targets in data_loader:
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = retina(images, targets) # Compute loss
        # Loss function measures the degree of dissimilarity of obtained result to the target
        losses = sum(loss for loss in loss_dict.values())
        i += 1

        # Backpropagation
        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

        # Sum loss for each epoch
        epoch_loss += losses
    print(f'epoch : {epoch+1}, Loss : {epoch_loss}, time : {time.time() - start}')
```

```
epoch : 1, Loss : 76.74559498004042, time : 106.46723413467407
epoch : 2, Loss : 75.85184099127564, time : 103.59798884391785
epoch : 3, Loss : 72.37947914672438, time : 103.40464091300964
epoch : 4, Loss : 72.1808343901771, time : 102.82599973678589
epoch : 5, Loss : 68.14326810749797, time : 103.04368543624878
epoch : 6, Loss : 67.50114628781311, time : 102.49140977859497
epoch : 7, Loss : 62.594183221273155, time : 102.49405264854431
epoch : 8, Loss : 50.78975480005989, time : 102.3589825630188
epoch : 9, Loss : 56.697089595271585, time : 101.81583046913147
epoch : 10, Loss : 38.815816437761335, time : 102.08831214904785
epoch : 11, Loss : 34.19554924488327, time : 102.35137009620667
epoch : 12, Loss : 29.072249067212923, time : 102.21073389053345
epoch : 13, Loss : 29.01360561839612, time : 102.07023668289185
epoch : 14, Loss : 27.01599307609461, time : 101.98805665969849
epoch : 15, Loss : 24.826299380610358, time : 102.2506697177887
epoch : 16, Loss : 29.027230432050246, time : 101.95783233642578
epoch : 17, Loss : 24.77057720939754, time : 102.03679966926575
epoch : 18, Loss : 24.283924340868413, time : 102.58550429344177
epoch : 19, Loss : 21.171928325528853, time : 102.17917561531067
epoch : 20, Loss : 19.76970547477169, time : 102.18121695518494
epoch : 21, Loss : 18.162997906949833, time : 102.66063213348389
epoch : 22, Loss : 17.835951062373848, time : 101.9341950416565
epoch : 23, Loss : 23.361887839453686, time : 101.99352264404297
epoch : 24, Loss : 20.971919028520613, time : 102.31040453910828
epoch : 25, Loss : 17.404305394748185, time : 102.59362530708313
epoch : 26, Loss : 15.817941522617973, time : 102.21612930297852
epoch : 27, Loss : 15.247377637912212, time : 102.37698554992676
epoch : 28, Loss : 15.06817156544111, time : 102.4182825088501
epoch : 29, Loss : 15.940321436320762, time : 102.47886276245117
```

Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called state_dict. These can be persisted via the torch. To reuse the model, we will save the learned weights. The torch.save function is used to save the model weights at the designated location.

```
# Save the trained model
if not saved_model:
    torch.save(retina.state_dict(),f'retina_{num_epochs}.pt')
```

3.3 Prediction

The training process has been completed. Now we will check the prediction. We will load the data using test_data_loader and input them into the model. In the model prediction phase, we use the threshold parameter to select bounding boxes with a certain level of confidence or higher. Then, we will make inferences on all the data in test_data_loader using the loop.

```
def make_prediction(model, img, threshold):
    model.eval()
    preds = model(img)
    for id in range(len(preds)) :
        idx_list = []

        # the threshold parameter: select bounding boxes with a certain level
        for idx, score in enumerate(preds[id]['scores']) : # confidence score
            if score > threshold :
                idx_list.append(idx)

        preds[id]['boxes'] = preds[id]['boxes'][idx_list]
        preds[id]['labels'] = preds[id]['labels'][idx_list]
        preds[id]['scores'] = preds[id]['scores'][idx_list]

    return preds
```

- preds_adj_all: predicted results
- annot_all: actual labels for all test data

```
# Make inferences on all the data in test_data_loader.
labels = []
preds_adj_all = []
annot_all = []

for im, annot in tqdm(test_data_loader, position = 0, leave = True):
    im = list(img.to(device) for img in im)

    for t in annot:
        labels += t['labels']

    with torch.no_grad():
        preds_adj = make_prediction(retina, im, 0.5) # Use threshold = 0.5 for rule of thumb
        preds_adj = [{k: v.to(torch.device('cpu')) for k, v in t.items()} for t in preds_adj]
        preds_adj_all.append(preds_adj)
        annot_all.append(annot)
```

```
100%|██████████| 85/85 [00:15<00:00,  5.34it/s]
```

Road Sign Detection

Student: Hanna, Derek, Ray

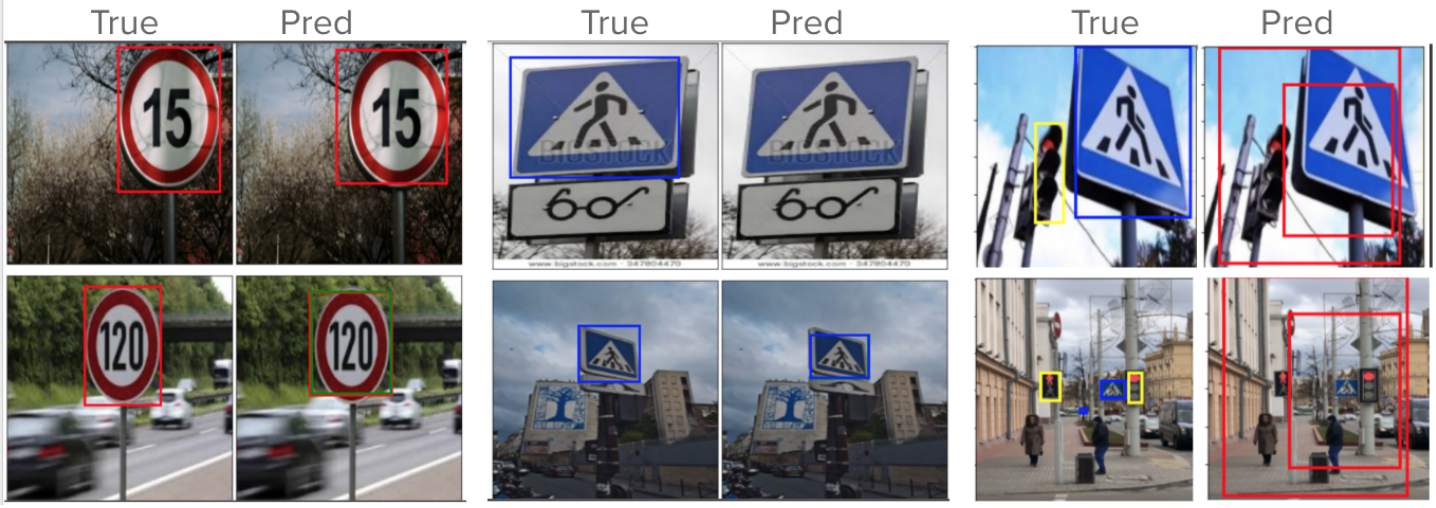
RetinaNet

Accuracy: 88.7%

Total objects: 247

Correctly identified objects: 219

Falsely detected objects = 89

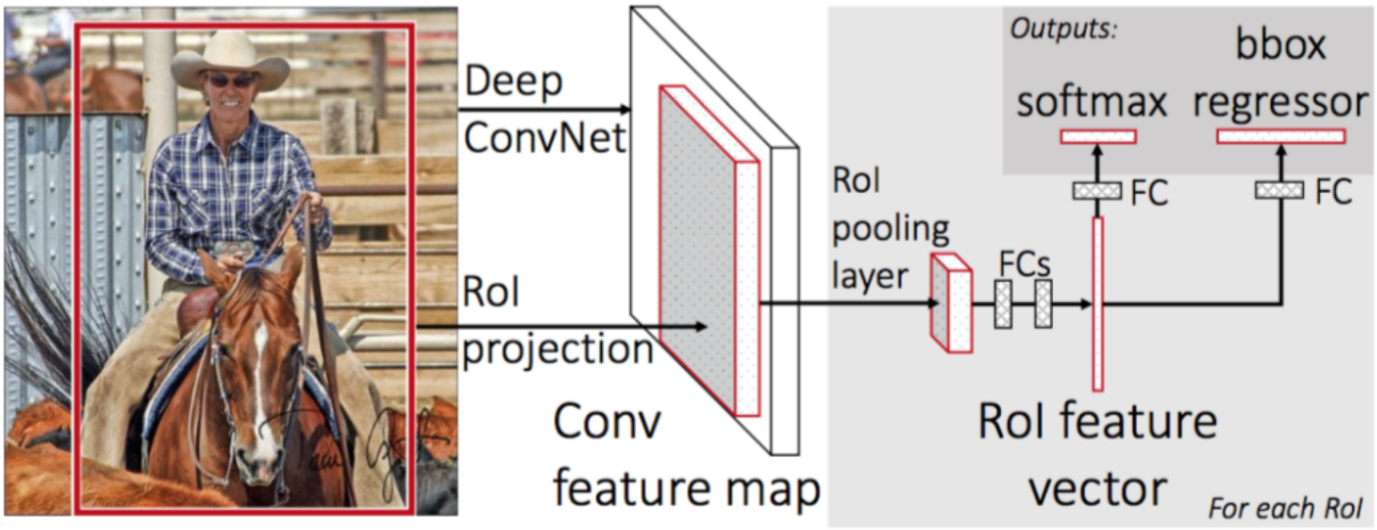


Accuracy for the classification is 88.7%. RetinaNet is fast, but has relatively low accuracy.
As you can see in the image, if an image has only one sign to detect, retinaNets find the bounding box with high accuracy. However, it shows poor performance on the image with multiple signs to detect. This is because our dataset is skewed.

4. Transfer Learning: Fast RCNN

In this chapter, we will detect road signs with Faster R-CNN, a two-stage detector. One branch of object detectors is based on multi-stage models. Deriving from the work of R-CNN, one model is used to extract regions of objects, and a second model is used to classify and further refine the localization of the object. Such methods are known to be relatively slow, but very powerful, but recent progress such as sharing features improved 2 stage detectors to have similar computational cost with single-stage detectors. These works are highly dependent on previous works and mostly build on the previous pipeline as a baseline.

To make R-CNN faster, Girshick (2015) improved the training procedure by unifying three independent models into one jointly trained framework and increasing shared computation results, named Fast R-CNN. Instead of extracting CNN feature vectors independently for each region proposal, this model aggregates them into one CNN forward pass over the entire image and the region proposals share this feature matrix. Then the same feature matrix is branched out to be used for learning the object classifier and the bounding-box regressor. In conclusion, computation sharing speeds up R-CNN.



Here, we will use the torchvision API to load the pre-trained model. Then, we will train the model through transfer learning. Finally, we will make inferences based on the test dataset and evaluate the model's performance.

4.1 Load model

Pytorch provides the Faster R-CNN API (torchvision.models.detection.fasterrcnn_resnet50_fpn) so it can be easily implemented. This provides a model that has been pre-trained with the COCO dataset using ResNet50. We can choose to load the pre-trained weights by declaring pretrained=True/False.

When loading the model, set the desired number of classes in num_classes and use the model. One thing to note when using Faster R-CNN is that we need to include the background class when specifying the class number in num_classes.

```
def get_model_instance_segmentation(num_classes):
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
    return model
```

If we already trained the model before and saved it, we will reuse it to save time. If it is the first time to train, we will run the code to learn the weights and save the model.

4.2 Transfer Learning

After importing the model, we will use the code below to perform transfer learning. Training a model is an iterative process; in each iteration, the model makes a guess about the output, calculates the error in its guess (loss), collects the derivatives of the error with respect to its parameters, and optimizes these parameters using gradient descent.

Hyperparameters and *Optimizer*

```
num_epochs = 10 # Number of Epochs - the number times to iterate over the dataset
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.007, momentum=0.9, weight_decay=0.0005)
```

Loss Function

When presented with some training data, our untrained network is likely not to give the correct answer. Loss function measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training. To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value.

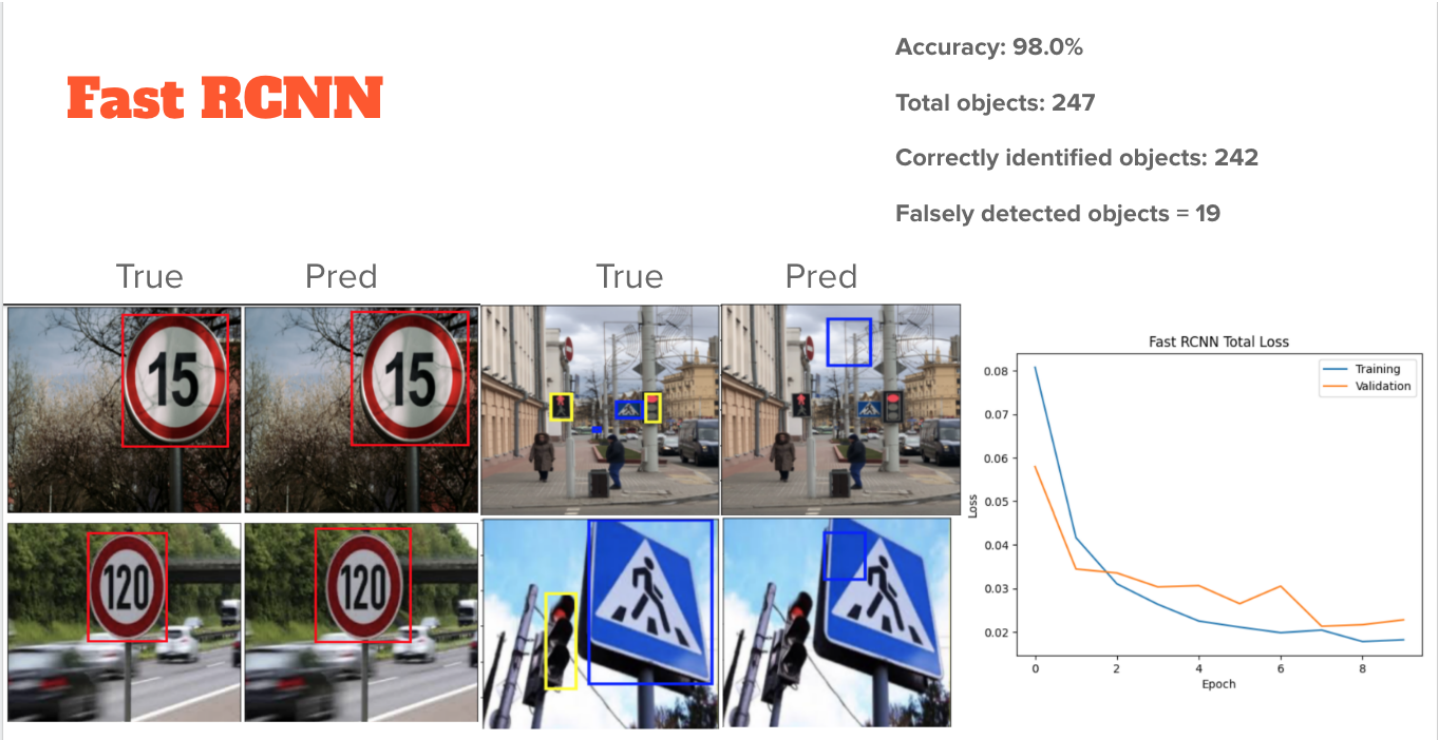
```
if not saved_model:
    for epoch in range(num_epochs):
        start = time.time()
        model.train()
        i = 0
        epoch_loss = 0
        for imgs, annotations in data_loader:
            i += 1
            imgs = list(img.to(device) for img in imgs)
            annotations = [{k: v.to(device) for k, v in t.items()} for t in annotations]
            loss_dict = model(imgs, annotations)
            losses = sum(loss for loss in loss_dict.values())

            optimizer.zero_grad()
            losses.backward()
            optimizer.step()
            epoch_loss += losses
        print(f'epoch : {epoch+1}, Loss : {epoch_loss}, time : {time.time() - start}')
```

```
epoch : 1, Loss : 16.071097929541427, time : 119.5690906047821
epoch : 2, Loss : 8.797052438312535, time : 120.80680084228516
epoch : 3, Loss : 6.276637791195382, time : 121.46104836463928
epoch : 4, Loss : 5.549270211816618, time : 121.53159880638123
epoch : 5, Loss : 4.761493627757919, time : 121.41937232017517
epoch : 6, Loss : 4.4793406216153775, time : 121.26239252090454
epoch : 7, Loss : 3.751034895614554, time : 121.36755299568176
epoch : 8, Loss : 3.4558459071449112, time : 121.40858554840088
epoch : 9, Loss : 3.207513634987048, time : 121.21366572380066
epoch : 10, Loss : 3.4137819717269555, time : 121.37791633605957
```

4.3 Prediction

The training process has been completed. Now we will check the prediction. We will load the data using test_data_loader and input them into the model. In the model prediction phase, we use the threshold parameter to select bounding boxes with a certain level of confidence or higher. Then, we will make inferences of all the data in test_data_loader using the loop.

```
with torch.no_grad():
    # batch size of the test set = 2
    for imgs, annotations in test_data_loader:
        imgs = list(img.to(device) for img in imgs)
        pred = make_prediction(model, imgs, 0.5)
        print(pred)
        break
```

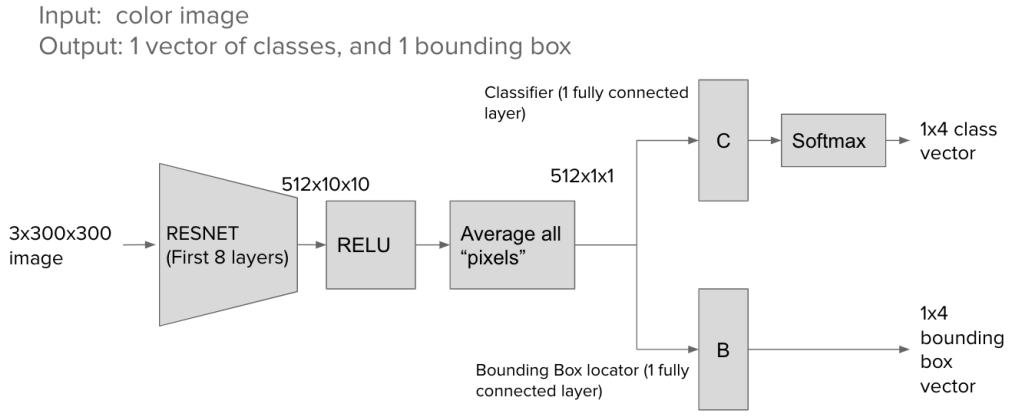


The result shows an accuracy 98% for the classification. Fast RCNN is relatively slow but it has high accuracy.

5. Simple CNN

5.1 Architecture

A simple convolutional neural network, that passes the images through Resnet, and then adds additional output layers to generate a single class output, and a single bounding box.




```
class Simple_CNN_model(torch.nn.Module):
    def __init__(self):
        super(Simple_CNN_model, self).__init__()
        resnet = models.resnet34(weights=torchvision.models.ResNet34_Weights.DEFAULT)
        layers = list(resnet.children())[:8]
        self.features1 = nn.Sequential(*layers[:6])
        self.features2 = nn.Sequential(*layers[6:])
        self.classifier = nn.Sequential(nn.BatchNorm1d(512), nn.Linear(512, 4), nn.Softmax(dim=0))
        self.bb         = nn.Sequential(nn.BatchNorm1d(512), nn.Linear(512, 4))

    def forward(self, x, targets=None):
        x = torch.stack(x) # Convert list of tensors to 4D tensor
        batch_size = x.shape[0]
        x = self.features1(x)
        x = self.features2(x)
        x = torch.nn.functional.relu(x)
        x = nn.AdaptiveAvgPool2d((1,1))(x)
        x = x.view(x.shape[0], -1)
        bbox = self.bb(x)
        scores = self.classifier(x)

        if self.training: # Return loss
            # print(targets)
            class_loss=0
            bb_loss=0
            for im_idx in range(batch_size):
                # Select closest bounding box
                im_tagtets = targets[im_idx]
                # Compute error for every bounding box in the target set
                errs = torch.empty(len(im_tagtets['labels']))
                for idx, tbox in enumerate(im_tagtets['boxes']):
                    errs[idx] = bb_error_calc(bbox[im_idx], tbox)
                # Find box with minimum error
                m = torch.min(errs,0)
```

```
            # Compute and return losses
            targ_class = im_tagtets['labels'][m.indices]
            class_loss += 1-scores[im_idx][targ_class]
            bb_loss += m.values

            # Select first target
            # im_targets = targets[im_idx]
            # bb_loss += bb_error_calc(bbox[im_idx], im_targets['boxes'][0])
            # class_loss += 1-scores[im_idx][im_targets['labels'][0]]
        return {'classification':class_loss, 'bbox_regression':bb_loss}

    else: # Return predictions: list (per epoch) of dict (boxes, scores, labels) for each detection
        # Return a prediction for each type (with the same bounding box for each prediction)
        result = [{} for _ in range(batch_size)]
        for im_idx in range(batch_size):
            p = torch.max(scores[im_idx], 0)
            result[im_idx]={'boxes':bbox[im_idx].reshape(1,4), 'scores':p.values.reshape(1), 'labels':p.indices.reshape(
        return result
```

5.2 Training

Loss is calculated pretty simply. It is the sum of bounding box loss and the class loss. (Bounding box loss is just the sum of the squares in the error of the box side positions.)

One challenge with this is that this model only detects a single object, while the data set has multiple objects per image. So evaluating the output isn't trivial.

```
# parameters
params = [p for p in model.parameters() if p.requires_grad] # select parameters that require gradient calculation
optimizer = torch.optim.SGD(params, lr=opt_param[0], momentum=opt_param[1], weight_decay=opt_param[2])

len_dataloader = len(data_loader)
# about 4 min per epoch on Colab GPU
model.to(device)
for epoch in range(num_epochs):
    start = time.time()
    model.train()
    i = 0
    epoch_loss = 0
    c_loss = 0
    bb_loss = 0
    for images, targets in tqdm(data_loader):
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())
        i += 1
        optimizer.zero_grad()
        losses.backward()
        optimizer.step()
        epoch_loss += losses
        c_loss += loss_dict['classification']
        bb_loss += loss_dict['bbox_regression']
    print(f'Epoch: {epoch+1}, Total Loss: {epoch_loss:0.2f}, class loss: {c_loss:0.2f}, bbox loss: {bb_loss:0.2f}')
```

5.3 Prediction

We choose to evaluate the model's performance against the object in the images closest to the bounding box. This sort of gives the model the benefit of the doubt, we let it choose which object in the image to detect.

Performance: Poor. Only first epoch of training yields any improvements.

Epocs Trained	Correct object detections	False object detections	Detection accuracy
10	101	69	40.9 %
20	77	77	31.2 %
30	98	72	39.7 %

Samples:

6. Conclusion

Final Model Comparison

RetinaNet: Fast, low-accuracy
Fast RCNN: Slow, high-accuracy

Model	Epoch	Classification Accuracy
RetinaNet	30	88.7%
Fast RCNN	10	98%
Simple CNN	10	40.9%

The results are that the best performing network is the Fast R-CNN. This is expected based on our reading. Retina net also performs very well. Finally, the simple CNN performed poorly as expected.

The main conclusion is that the styles of CNN used for image identification in homework 4, are not powerful enough to perform more complicated identification tasks such as image segmentation, and images with multiple objects.

7. Reference

- Pytorch Tutorial, Pytorch
- Transfer learning, Lecture note cs231
- Bounding Box Transformation, Medium Blog
- Transfer Learning, Blog
- RetinaNet paper
- RCNN blog

8. Code

https://github.com/hungryPanko/MultipleObjectDetection