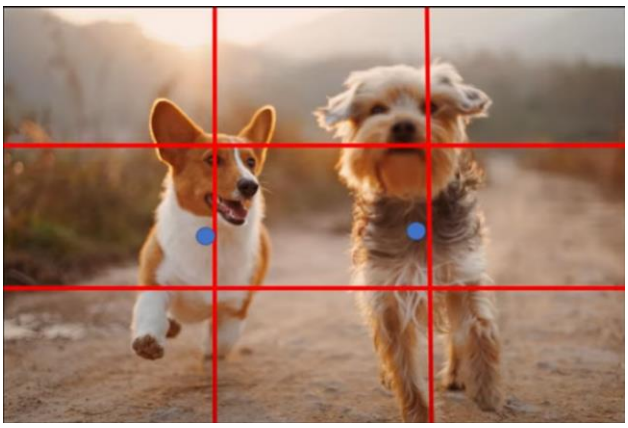
YOLOv1 Implementation in Pytorch

Reference: https://www.youtube.com/watch?v=n9_XyCGr-MI&list=PLhhyoLH6Ijfw0TpCTVTNk42NN08H6UvNq&index=5

YOLO algorithm split the input image to S x S cells and each cell is responsible to output a prediction with a corresponding bounding box. Object's midpoint indicate which cell is responsible to predict that object.



Each output and label will be relative to the cell.

Each bounding box for each cell will have *[midpoint-x, midpoint-y, width, height]*. All the value will be relative to the cell. Therefore, midpoint coordinate will be from 0 to 1; however, width and height can be greater than 1 if object is wider or taller than cell.

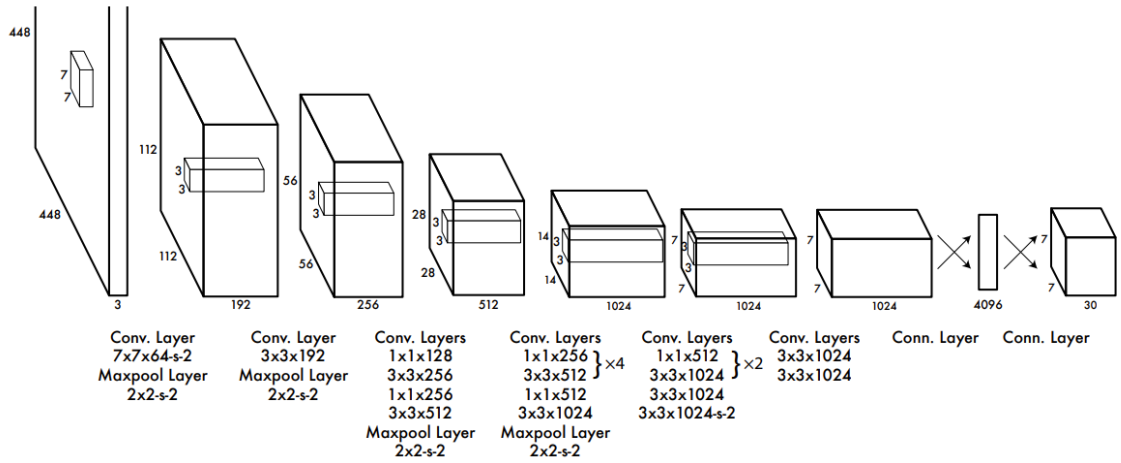
The labels look like:

$$lable_{cell} = [C_1, C_2, C_3 \dots C_{20}, P_{C1}, x, y, w, h, P_{C2}, x, y, w, h]$$

In this case, C is the object class has total 20 objects. P is the probability that there is an object (1 or 0) in that cell. In YOLOv1, predictions will output two bounding boxes and we expect that they will output different shape of bounding boxes (One wider and the other taller). Notice that we only do class prediction once so there can only be one object detected in a cell.

Target shape for an image: (S, S, 25)
Prediction shape for an image: (S, S, 30)

Model Architecture



Φ The 7 x 7 kernel with stepping 2 require padding as 3 to exactly half the size.

```
"""
Tuple is structured by (kernel_size, filters, stride, padding)
"M" is simply maxpooling with stride 2x2 and kernel 2x2
List is structured by tuples and lastly int with number of repeats
"""
architecture_config = [
    (7, 64, 2, 3),
    "M",
    (3, 192, 1, 1),
    "M",
    (1, 128, 1, 0),
    (3, 256, 1, 1),
    (1, 256, 1, 0),
    (3, 512, 1, 1),
    "M",
    [(1, 256, 1, 0), (3, 512, 1, 1), 4],
    (1, 512, 1, 0),
    (3, 1024, 1, 1),
    "M",
    [(1, 512, 1, 0), (3, 1024, 1, 1), 2],
    (3, 1024, 1, 1),
    (3, 1024, 2, 1),
    (3, 1024, 1, 1),
    (3, 1024, 1, 1),
]
```

Model architecture defined in python.

```
class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, **kwargs):
        super(CNNBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **kwargs)
        self.batchnorm = nn.BatchNorm2d(out_channels)
        self.leakyrelu = nn.LeakyReLU(0.1)

    def forward(self, x):
        return self.leakyrelu(self.batchnorm(self.conv(x)))
```

In the CNN block, a batch normalization is included. The reason that YOLOv1 did not include batch normalization because it's not invented at that time.

Create the YOLOv1 using the architecture configure file (detail in source code)

```
class Yolov1(nn.Module):
    def __init__(self, in_channels=3, **kwargs):
        super(Yolov1, self).__init__()
        self.architecture = architecture_config
        self.in_channels = in_channels
        self.darknet = self._create_conv_layers(self.architecture)
        self.fcs = self._create_fcs(**kwargs)

    def forward(self, x):
        x = self.darknet(x)
        return self.fcs(torch.flatten(x, start_dim=1)) # Dimension 0 is number of examples
```

CNN blocks

```
        for x in architecture:
            if type(x) == tuple:
                layers += […]
                in_channels = x[1] # set the in channels for the next CNN block

            elif type(x) == str:
                layers += [nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))]

            elif type(x) == list:
                conv1 = x[0] # TUPLE
                conv2 = x[1] # TUPLE
                num_repeats = x[2] #Integer
                 …
        return nn.Sequential(*layers) # Unpack the list and converted it to nn.Sequential
```

Fully connected layers

```
S, B, C = split_size, num_boxes, num_classes

        # In original paper this should be
        # nn.Linear(1024 * S * S, 4096),
        # nn.Dropout(0.5)
        # nn.LeakyReLU(0.1),
        # nn.Linear(4096, S * S * (C + B * 5))

        return nn.Sequential(
            nn.Flatten(),
            nn.Linear(1024 * S * S, 512), # TODO change this to 512 ~ 4096
            nn.Dropout(0.0),
            nn.LeakyReLU(0.1),
            nn.Linear(512, S * S * (C + B * 5)), # TODO change this to 512 ~ 4096
            # Will be reshape to (S, S, 30)
        )
```

Loss Function

Identity function, which means only compute when there is a bounding box in cell *i* and bounding box *j* in the cell is responsible to predict the ground truth box for the object.

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

For coordinate (midpoint of the object)

For width and height of the bounding box. Taking square root make sure that error from big bounding box and small bounding box are evaluate equally

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

For if there is an object in the cell

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

For if there is no object in the cell

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2$$

Next page

Φ For identity function, the bounding box has the highest IoU of any predictor in that grid cell.

Φ If there is no object in the cell, we should train all the bounding box to make them understand that there is no object in that cell.

For the finial line of the loss function, this function is for the object class loss. One cell can only predict one object so the function is not including the bounding box summation. In order to simplify the loss function, they use regression rather than cross entropy for the class loss.

```
        # Calculate IoU for the two predicted bounding boxes with target bbox
        iou_b1 = intersection_over_union(predictions[..., 21:25], target[..., 21:25])
        iou_b2 = intersection_over_union(predictions[..., 26:30], target[..., 21:25])
        ious = torch.cat([iou_b1.unsqueeze(0), iou_b2.unsqueeze(0)], dim=0)

        # Take the box with highest IoU out of the two prediction
        # Note that bestbox will be indices of 0, 1 for which bbox was best
        iou_maxes, bestbox = torch.max(ious, dim=0)
```

The bestbox will store an array of index (0 or 1) which indicate which of the two bounding box is better (best out of two).

Python functions

```
import torch

a = torch.tensor([15, 20, 40])
b = torch.tensor([150, 200, 400])

c = torch.cat([a.unsqueeze(0), b.unsqueeze(0)], dim=0)

'''
a.unsqueeze(0)
--> tensor([[15, 20, 40]])

d
-->
torch.return_types.max(
values=tensor([150, 200, 400]),
indices=tensor([1, 1, 1]))

'''
```

Calculate the Box Coordinates Error

We can calculate the midpoint error and width, height error together since they all are mean square error.

The only difference is the width and height error need to be square rooted to keep the big bounding box and smaller bounding box having the similar sized error and evaluated equally.

```
# Set boxes with no object in them to 0. We only take out one of the two
        # predictions, which is the one with highest Iou calculated previously.
        box_predictions = exists_box * (
            (
                bestbox * predictions[..., 26:30]
                + (1 - bestbox) * predictions[..., 21:25]
            )
        )
        # Similar technique from bit hacks
        # We take out four values (midpoint_x, midpoint_y, width, height)

        box_targets = exists_box * target[..., 21:25]

        # Take sqrt of width, height of boxes
        box_predictions[..., 2:4] = torch.sign(box_predictions[..., 2:4]) * torch.sqrt(
            torch.abs(box_predictions[..., 2:4] + 1e-6)
        )
        # Only absolute value can take square root; however, we have to keep the sign correct

        box_targets[..., 2:4] = torch.sqrt(box_targets[..., 2:4])

        # (N, S, S, 4) -> (N * S * S, 4)
        box_loss = self.mse(
            torch.flatten(box_predictions, end_dim=-2),
            torch.flatten(box_targets, end_dim=-2),
        )
        # This includes midpoint coordinate error and width and height error
```

Calculate the Object Loss

Similar to how we take out the best bounding box out of two, we just need to take the probability for the best bounding box out.

Calculate No Object Loss

If there is no object, both of the bounding box should indicate that there is no object in that cell not just the best bounding box. The author flattens the predictions differently; however, the concept is similar.

Calculate Class Loss

In YOLOv1, the class loss is also using mean square error rather than cross entropy.

```
        class_loss = self.mse(
            # (N, S, S, 20) -> (N * S * S, 20)
            torch.flatten(exists_box * predictions[..., :20], end_dim=-2),
            torch.flatten(exists_box * target[..., :20], end_dim=-2),
        )
```

Sum up

Calculate the final loss with the weighted losses.

```
loss = (
            self.lambda_coord * box_loss  # first two rows in paper
            + object_loss  # third row in paper
            + self.lambda_noobj * no_object_loss  # forth row
            + class_loss  # fifth row
        )

        return loss
```

Dataset

Build the dataset from csv files provided by the author and do data preprocessing. We do not have data transformation in this implementation.

```
with open(label_path) as f:
            for label in f.readlines():
                class_label, x, y, width, height = [
                    float(x) if float(x) != int(float(x)) else int(x)
                    # check if the number is float or not
                    # the class_label should be integer
                    for x in label.replace("\n", "").split()
                ]

                boxes.append([class_label, x, y, width, height])
```

To check if the string is an integer or floating number, the author compares the integer part of the float with float itself. If it is equal, the floating number is an integer.

Additionally, we need to convert the bounding box coordinate and width and height to relative to the cell instead of the entire image.

```
            # x, y is relative to the entire image
            # i, j represents the cell row and cell column
            i, j = int(self.S * y), int(self.S * x)
            x_cell, y_cell = self.S * x - j, self.S * y - i
```

To finalize the label matrix, we need to put all those converted values we calculated into the label matrix. Additionally, we need to limited one object per cell. Therefore, we need to check if the cell is taken or not and label the cell, we put the object in.

```
            if label_matrix[i, j, 20] == 0:
                # Set that there exists an object
                label_matrix[i, j, 20] = 1

                # Box coordinates
                box_coordinates = torch.tensor(
                    [x_cell, y_cell, width_cell, height_cell]
                )

                label_matrix[i, j, 21:25] = box_coordinates

                # Set one hot encoding for class_label
                label_matrix[i, j, class_label] = 1
```

Train

Since the original paper trained the YOLOv1 for a long time, the author will only overfit a batch and train on 100 examples to see if the training process is accurate and the loss is decreasing.

```
LEARNING_RATE = 2e-5
DEVICE = "cuda" if torch.cuda.is_available else "cpu"
BATCH_SIZE = 32 # 64 in original paper
WEIGHT_DECAY = 0
EPOCHS = 100
NUM_WORKERS = 6
PIN_MEMORY = True
LOAD_MODEL = False
LOAD_MODEL_FILE = "overfit.pth.tar"
IMG_DIR = "data/images"
LABEL_DIR = "data/labels"


...


transform = Compose([transforms.Resize((448, 448)), transforms.ToTensor(),])
```

Before we put image into the model, we need to resize the image to *448 * 448*. In this case, since we don't have any other transformation, the bounding boxes need no change because all of the coordinate and width and height are relative values.

After training for 100 epochs on one batch.

```
Epoch #99:
Train mAP: 0.9280602335929871
100%|                                                                                | 1/1 [00:00<00:00,  2.66it/s]
Mean loss was 0.21308062970638275
```

Some example images in the batch.