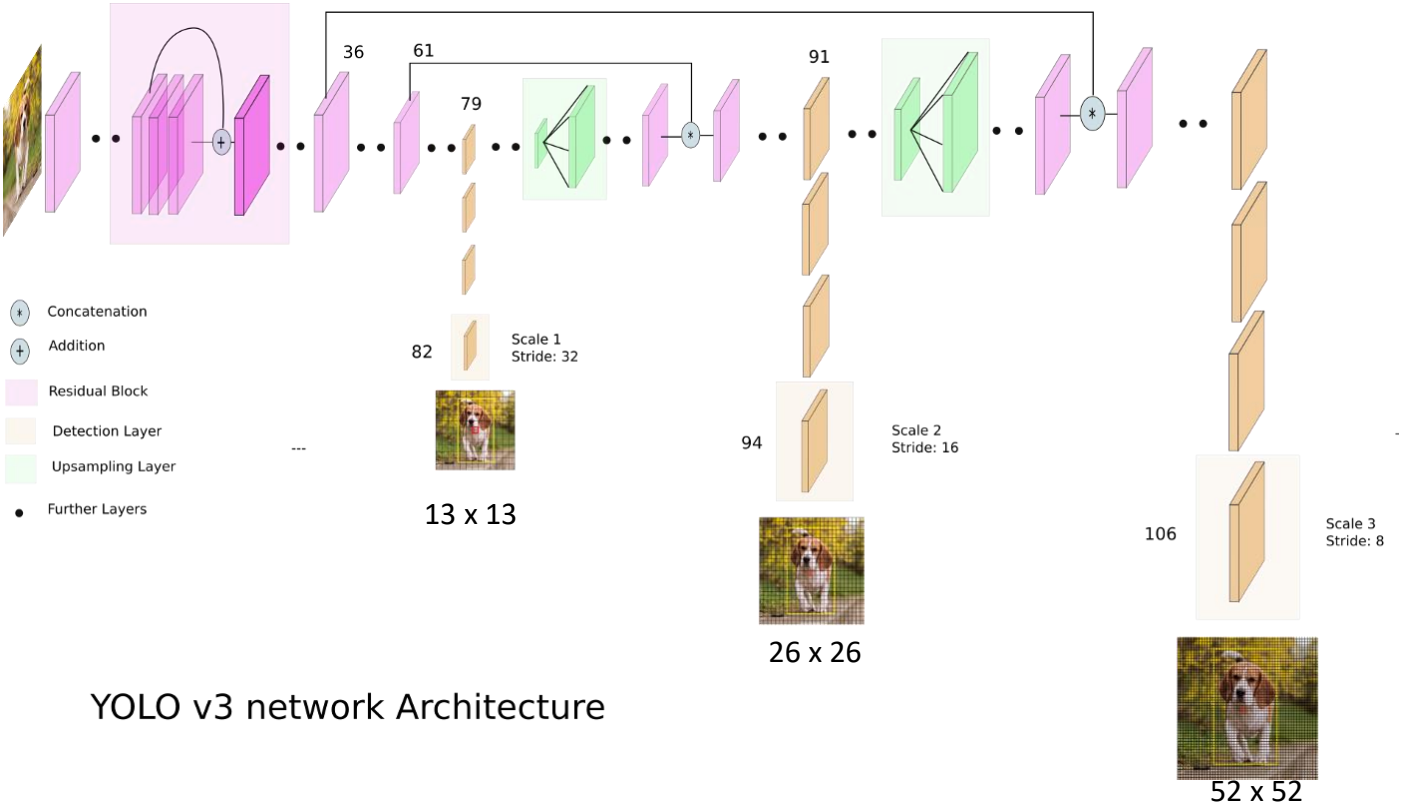
YOLOv3 Implementation with Pytorch

Reference:
https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b
https://www.youtube.com/watch?v=Grir6TZbc1M

YOLOv1 vs YOLOv3

YOLOv1 output a single 7 x 7 grid (Great for large bounding box but not great for small bounding box)
YOLOv3 makes detection at three different scales (Small scale image will produce large bounding box)

YOLOv3 has total 106 layers fully convolutional.
YOLOv3 uses Anchor Boxes



YOLO v3 network Architecture

Φ Stride of the network, or a layer is defined as the ratio by which it down-samples the input. The three scales with down-sampling the image dimensions by 32, 16, 8 respectively.

Reference: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

Anchor Boxes

YOLOv2 and YOLOv3 using Anchor Boxes which utilize the previous knowledge (Knowledge from other program) that can be encoded into the bounding boxes. Anchor boxes serve as the sample for model to adjust and produce bounding boxes with. In YOLOv3, we have 3 anchor boxes for each cell therefore,

totally, we have 9 different predefined anchor boxes. By using K-means clustering on both VOC and COCO dataset, the anchor boxes are found and maximize the IoU with all the target bounding boxes.

$$b_x = \sigma(t_x)$$

Taking sigmoid on t_x (output form the model)

$$b_y = \sigma(t_y)$$

Taking sigmoid on t_y (output form the model)

$$b_w = p_w e^{t_w}$$

t_w exponential serve as the ratio of the anchor box width (p_w)

$$b_h = p_h e^{t_h}$$

t_h exponential serve as the ratio of the anchor box width (p_h)

Model Architecture

```
config = [
    (32, 3, 1),(64, 3, 2),
    ["B", 1],
    (128, 3, 2),
    ["B", 2],
    (256, 3, 2),
    ["B", 8],
    (512, 3, 2),
    ["B", 8],
    (1024, 3, 2),
    ["B", 4],  # To this point is Darknet-53
    (512, 1, 1), (1024, 3, 1),
    "S",
    (256, 1, 1),
    "U",
    (256, 1, 1), (512, 3, 1),
    "S",
    (128, 1, 1),
    "U",
    (128, 1, 1), (256, 3, 1),
    "S",
]
```

The following is the configuration of the model architecture. Additional information and explanation of numbers and alphabets is as following.

```
"""
Information about architecture config:
Tuple is structured by (filters, kernel_size, stride)
Every conv is a same (padding) convolution.
List is structured by "B" indicating a residual block followed by the number of repeats
"S" is for scale prediction block and computing the yolo loss
"U" is for upsampling the feature map and concatenating with a previous layer
"""

'''
Tuple: Convolution (out_channels, kernel_size, stride)
All convolutions are with same padding

List: Residual Network ["B", number_repeats]
'''
```

Implementation of CNNBlock

```
class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, bn_act=True, **kwargs):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
                              bias=not bn_act, **kwargs)
        self.bn = nn.BatchNorm2d(out_channels)
        self.leaky = nn.LeakyReLU(0.1)
        self.use_bn_act = bn_act

    def forward(self, x):
        if self.use_bn_act:  # an if statement in forward pass might casue performance loss
            return self.leaky(self.bn(self.conv(x)))
        else:
            return self.conv(x)
```

If a batch normalization is applied, the bias in Convolutional layer is not needed. The goal of batch normalization is to make output mean to 0 and standard deviation to 1 thus adding a bias (offset) makes no sense.

In scaled prediction, we do not want to use batch normalization and leaky-ReLU therefore, we need to version of the CNNBlock.

Implementation of ResidualBlock

```
class ResidualBlock(nn.Module):
    def __init__(self, channels, use_residual=True, num_repeates=1):
        super().__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_repeates):
            self.layers += [
                nn.Sequential(
                    CNNBlock(channels, channels//2, kernel_size=1),
                    CNNBlock(channels//2, channels, kernel_size=3, padding=1)
                )
            ]

        self.use_residual = use_residual
        self.num_repeats = num_repeates

    def forward(self, x):
        for layer in self.layers:
            x = layer(x) + x if self.use_residual else layer(x)
        return x
```

Notice that *self.layers* contain a list of *nn.Sequential* which includes two convolutional layers. Additionally, since we did not change channels and image size, we can simply add the *x* to the layer's output.

Implementation of ScalePrediction

```
class ScalePrediction(nn.Module):
    def __init__(self, in_channels, num_classes):
        super().__init__()
        self.pred = nn.Sequential(
            CNNBlock(in_channels, 2 * in_channels, kernel_size=3, padding=1),
            # [po, x, y, w, h] as the out_channel for the grid
            CNNBlock(2 * in_channels, (num_classes + 5) * 3, bn_act=False, kernel_size=1)
        )
        self.num_classes = num_classes

    def forward(self, x):
        return (
            self.pred(x)
            .reshape(x.shape[0], 3, self.num_classes + 5, x.shape[2], x.shape[3]).permute(0, 1, 3, 4, 2)
        )
        # N x 3 x 13 x 13 x (5 + num_classes)
```

Notice that the goal of the 1 x 1 kernel is to reduce the channels to the prediction format that we want and in the forward function, we have to split the bounding boxes and reorder the output to fit into the prediction output format.

Build the YOLOv3 Model

```
class YOLOv3(nn.Module):
    def __init__(self, in_channels=3, num_classes=20):
        super().__init__()
        self.num_classes = num_classes
        self.in_channels = in_channels
        self.layers = self._create_conv_layers()
    def forward(self, x):
        pass
    def _create_conv_layers(self):
        layers = nn.ModuleList()
        in_channels = self.in_channels

        for module in config:
            if isinstance(module, tuple):
               …
            elif isinstance(module, list):
                …
            elif isinstance(module, str):
                if module == 'S':
                    layers += [
                        ResidualBlock(in_channels, use_residual=False, num_repeates=1),
                        CNNBlock(in_channels, in_channels//2, kernel_size=1),
                        ScalePrediction(in_channels//2, num_classes=self.num_classes)
                    ]
                    in_channels = in_channels // 2

                elif module == 'U':
                    layers.append(nn.Upsample(scale_factor=2))
                    in_channels = in_channels * 3
     # concat should happened immediately after up-sampling and the number of channel increase specifically * 3

        return layers
```

Notice that we in the up-sampling module, we increase the channel size by three times since the concat should be immediately after the up-sampling module and the number of the channel should be increased.

Forward part

```
    def forward(self, x):
        outputs = [] # for storing scaled prediction result
        route_connection = [] # for cancat

        for layer in self.layers:
            if isinstance(layer, ScalePrediction):
                outputs.append(layer(x))
                continue # continue the followed layer from output before scaleprediction applied.

            x = layer(x)

            if isinstance(layer, ResidualBlock) and layer.num_repeats == 8:
                route_connection.append(x) # record the layer output for concat later.

            elif isinstance(layer, nn.Upsample):
                x = torch.cat([x, route_connection[-1]], dim=1) # concat with the LAST route_connection
                route_connection.pop()
```

Dataset

Load images and labels

```
        label_path = os.path.join(self.label_dir, self.annotations.iloc[index, 1])
        bboxes = np.roll(np.loadtxt(fname=label_path, delimiter=" ", ndmin=2), 4, axis=1).tolist()
        # rolling [class, x, y, w, h] => [x, y, w, h, class] for albumentations
        img_path = os.path.join(self.img_dir, self.annotations.iloc[index, 0])
        image = np.array(Image.open(img_path).convert("RGB"))
        # the image should be a np array for albumentation

        if self.transform:
            augmentations = self.transform(image=image, bboxes=bboxes)
            image = augmentations["image"]
            bboxes = augmentations["bboxes"]

        targets = [torch.zeros((self.num_anchors // 3, S, S, 6)) for S in self.S]
        # 6 values -> [p_o, x, y, w, h, class]
```

We need to go through all the boxes in an image and assign an anchor box (with highest IoU) and define which cell is responsible for all three different scales. Additionally, an anchor in an scale in a cell can only be taken once and an target can only be assigned with one anchor box.

```
            iou_anchors = iou(torch.tensor(box[2:4]), self.anchors)
            # this anchor IoU only calculate with width and height needed (same midpoint)
            anchor_indices = iou_anchors.argsort(descending=True, dim=0)
            x, y, width, height, class_label = box
            has_anchor = [False, False, False]

            for anchor_idx in anchor_indices:
                scale_idx = anchor_idx // self.num_anchors_per_scale
                # which scale the anchor belong to {0, 1, 2}
                anchor_on_scale = anchor_idx % self.num_anchors_per_scale
                # which anchor we want {0, 1, 2} from a specific scale
                # (total three anchor boxes for a scale)
                S = self.S[scale_idx]
                i, j = int(S * y), int(S * x)
                # x = 0.5, S = 13 --> int(6.5) = 6 (Sell size is equal to 1)
                anchor_taken = targets[scale_idx][anchor_on_scale, i, j, 0]
                # check if an anchor has been taken

                if not anchor_taken and not has_anchor[scale_idx]:
                    targets[scale_idx][anchor_on_scale, i, j, 0] = 1
                    x_cell, y_cell = S * x - j, S * y - i
                    # the x and y respect to a cell both are between [0, 1]
                    width_cell, height_cell = (
                        width * S,
                        height * S,
                    )

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

                    targets[scale_idx][anchor_on_scale, i, j, 1:5] = box_coordinates
                    targets[scale_idx][anchor_on_scale, i, j, 5] = int(class_label)
                    has_anchor[scale_idx] = True

                elif not anchor_taken and iou_anchors[anchor_idx] > self.ignore_iou_thresh:
                    targets[scale_idx][anchor_on_scale, i, j, 0] = -1
        # ignore this prediction (two anchor boxes in a cell that has high IoU with the target)
```

In this case, since we have to do data augmentation and setting anchor boxes and cells for responsibility, the data loading is expensive.

Loss Implementation

The obj and noobj is picked out as checking the if the target's object probability is 1 or 0.

No Object Loss

```
        no_object_loss = self.bce(
            (predictions[..., 0:1][noobj], (target[..., 0:1][noobj])),
        )
```

Object Loss

```
        anchors = anchors.reshape(1, 3, 1, 1, 2) # p_w * exp(t_w)
        # 3 x 2 reshape for the prediction format
        box_preds = torch.cat(
            [self.sigmoid(predictions[..., 1:3]), torch.exp(predictions[..., 3:5]) * anchors],
            dim=-1
            )
        ious = intersection_over_union(box_preds[obj], target[..., 1:5][obj]).detach()
        object_loss = self.mse(
            self.sigmoid(predictions[..., 0:1][obj]),
            ious * target[..., 0:1][obj]
            )
```

Box Coordinates

```
        predictions[..., 1:3] = self.sigmoid(predictions[..., 1:3])  # x,y coordinates
        target[..., 3:5] = torch.log(
            (1e-16 + target[..., 3:5] / anchors)
        )  # width, height coordinates
        # do the inverse calculation of exp for better gradient
        box_loss = self.mse(predictions[..., 1:5][obj], target[..., 1:5][obj])
```

Class Loss

```
        class_loss = self.entropy(
            (predictions[..., 5:][obj]), (target[..., 5][obj].long()),
        )
```

Finally, we just need to add all the losses times their specific weights and we get the loss.

Train

In this case, we train with automatic mixed precision for better training performance. Notice that in the loss, we need to pass the scaled anchor. Since the anchors in the configuration file are relative to the entire image, we need to scale them before we pass them to the loss function. Additionally, we keep track mean of all the losses.

```python
def train_fn(train_loader, model, optimizer, loss_fn, scaler, scaled_anchors):
    # train with float16 (AMP training)
    loop = tqdm(train_loader, leave=True)
    losses = []
    for batch_idx, (x, y) in enumerate(loop):
        x = x.to(config.DEVICE)
        y0, y1, y2 = (
            y[0].to(config.DEVICE),
            y[1].to(config.DEVICE),
            y[2].to(config.DEVICE),
        )
        # three scales (each scale has label)

        with torch.cuda.amp.autocast():
            out = model(x)
            loss = (
                loss_fn(out[0], y0, scaled_anchors[0])
                + loss_fn(out[1], y1, scaled_anchors[1])
                + loss_fn(out[2], y2, scaled_anchors[2])
            )

        losses.append(loss.item())
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        # update progress bar
        mean_loss = sum(losses) / len(losses)
        loop.set_postfix(loss=mean_loss)
```

Scale anchors

```python
    scaled_anchors = (
        torch.tensor(config.ANCHORS)
        * torch.tensor(config.S).unsqueeze(1).unsqueeze(1).repeat(1, 3, 2)
        # make the format so that we can multiple anchor for different scales
        # with their specific scale
    ).to(config.DEVICE)
```

Φ The data augmentation used compare to the augmentations from the original code from Aladdin Persson implementation is different because of the *albumentations* version differences.

```
100%| | 1/1 [00:00<00:00,  1.74it/s, loss=6.92]
100%| | 1/1 [00:00<00:00,  1.80it/s, loss=6.83]
100%| | 1/1 [00:00<00:00,  1.06it/s]
MAP: 0.9999989867210388
100%| | 1/1 [00:00<00:00,  1.88it/s, loss=6.76]
100%| | 1/1 [00:00<00:00,  1.75it/s, loss=6.67]
100%| | 1/1 [00:00<00:00,  1.75it/s, loss=6.59]
100%| | 1/1 [00:00<00:00,  1.86it/s, loss=6.52]
100%| | 1/1 [00:00<00:00,  1.80it/s, loss=6.44]
100%| | 1/1 [00:00<00:00,  1.67it/s, loss=6.36]
100%| | 1/1 [00:00<00:00,  1.70it/s, loss=6.28]
100%| | 1/1 [00:00<00:00,  1.70it/s, loss=6.22]
100%| | 1/1 [00:00<00:00,  1.80it/s, loss=6.15]
100%| | 1/1 [00:00<00:00,  1.72it/s, loss=6.08]
100%| | 1/1 [00:00<00:00,  1.30it/s]
MAP: 0.9652767777442932
```

We are able to overfit 8examples.

```
100%| | 4/4 [00:02<00:00,  1.65it/s, loss=1.44]
100%| | 4/4 [00:02<00:00,  1.72it/s, loss=1.45]
100%| | 4/4 [00:02<00:00,  1.69it/s, loss=1.42]
100%| | 4/4 [00:02<00:00,  1.70it/s, loss=1.42]
100%| | 4/4 [00:02<00:00,  1.68it/s, loss=1.41]
100%| | 4/4 [00:02<00:00,  1.69it/s, loss=1.41]
100%| | 4/4 [00:02<00:00,  1.66it/s, loss=1.38]
100%| | 4/4 [00:02<00:00,  1.69it/s, loss=1.36]
100%| | 4/4 [00:02<00:00,  1.59it/s, loss=1.37]
100%| | 4/4 [00:02<00:00,  1.54it/s, loss=1.37]
100%| | 4/4 [00:03<00:00,  1.04it/s]
MAP: 0.9601072072982788
100%| | 4/4 [00:02<00:00,  1.52it/s, loss=1.36]
100%| | 4/4 [00:02<00:00,  1.70it/s, loss=1.36]
100%| | 4/4 [00:02<00:00,  1.73it/s, loss=1.35]
100%| | 4/4 [00:02<00:00,  1.72it/s, loss=1.34]
100%| | 4/4 [00:02<00:00,  1.74it/s, loss=1.33]
100%| | 4/4 [00:02<00:00,  1.72it/s, loss=1.35]
100%| | 4/4 [00:02<00:00,  1.69it/s, loss=1.35]
100%| | 4/4 [00:02<00:00,  1.73it/s, loss=1.35]
100%| | 4/4 [00:02<00:00,  1.69it/s, loss=1.35]
100%| | 4/4 [00:02<00:00,  1.69it/s, loss=1.36]
100%| | 4/4 [00:03<00:00,  1.07it/s]
MAP: 0.980450451374054
```

We are able to overfit 100example as well.