YOLOv9 - Aparna Suresh

Import Necessary Libraries

```
# Standard libraries
import os
import time
import math
import platform
import contextlib
from copy import deepcopy
from pathlib import Path

# Numerical and data manipulation libraries
import numpy as np
import pandas as pd

# PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F

# Visualization libraries
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sn


# Clone repo to access weights and helper code
!git clone https://github.com/WongKinYiu/yolov9
%cd yolov9
```

```
Cloning into 'yolov9'...
remote: Enumerating objects: 781, done.
remote: Counting objects: 100% (316/316), done.
remote: Compressing objects: 100% (59/59), done.
remote: Total 781 (delta 265), reused 257 (delta 257), pack-reused 465 (from 1)
Receiving objects: 100% (781/781), 3.25 MiB | 15.06 MiB/s, done.
Resolving deltas: 100% (339/339), done.
/content/yolov9
```

Define YOLOv9 Backbone layers

```
class Silence(nn.Module):
    def __init__(self):
        super(Silence, self).__init__()  # Initialize the base nn.Module

    def forward(self, x):
        return x  # Identity operation: returns the input as-is without any changes



def autopad(k, p=None, d=1):  # kernel, padding, dilation
    # Automatically calculate padding to achieve 'same' output shape after convolution

    if d > 1:
        # Adjust kernel size to account for dilation
        # For example, a 3x3 kernel with dilation=2 becomes a 5x5 effective kernel
        k = d * (k - 1) + 1 if isinstance(k, int) else [d * (x - 1) + 1 for x in k]

    if p is None:
        # If padding is not specified, compute it to maintain input-output spatial size
        # For example, 3x3 kernel => padding = 1; 5x5 => padding = 2
        p = k // 2 if isinstance(k, int) else [x // 2 for x in k]

    return p  # Return the padding value(s)



class Conv(nn.Module):
    # Standard convolution block with optional activation
```

```
        # Standard convolution block with optional activation
        # Args: in_channels, out_channels, kernel_size, stride, padding, groups, dilation, activation

        default_act = nn.SiLU()  # Default activation is SiLU (Swish-like nonlinearity)

        def __init__(self, c1, c2, k=1, s=1, p=None, g=1, d=1, act=True):
            super().__init__()
            # Convolution layer without bias (since BatchNorm is used)
            self.conv = nn.Conv2d(c1, c2, k, s, autopad(k, p, d), groups=g, dilation=d, bias=False)
            self.bn = nn.BatchNorm2d(c2)  # Batch normalization after convolution

            # Activation function: default SiLU, or custom, or Identity (no activation)
            self.act = self.default_act if act is True else act if isinstance(act, nn.Module) else nn.Identity()

        def forward(self, x):
            # Forward pass with conv -> batchnorm -> activation
            return self.act(self.bn(self.conv(x)))

        def forward_fuse(self, x):
            # Forward pass without batchnorm (used in inference when BN is fused into conv)
            return self.act(self.conv(x))


    class AConv(nn.Module):
        # Downsampling convolution block
        def __init__(self, c1, c2):  # Input and output channels
            super().__init__()
            self.cv1 = Conv(c1, c2, 3, 2, 1)  # 3x3 conv with stride=2 for downsampling

        def forward(self, x):
            # Applies 2x2 average pooling (stride=1, no padding) before convolution
            x = torch.nn.functional.avg_pool2d(x, 2, 1, 0, False, True)
            return self.cv1(x)



    class ADown(nn.Module):
        def __init__(self, c1, c2):  # c1: input channels, c2: output channels
            super().__init__()
            self.c = c2 // 2  # Half of the output channels

            # Path 1: 3x3 conv with stride 2 on half of input
            self.cv1 = Conv(c1 // 2, self.c, 3, 2, 1)

            # Path 2: 1x1 conv after max pooling on the other half
            self.cv2 = Conv(c1 // 2, self.c, 1, 1, 0)

        def forward(self, x):
            # Apply average pooling with kernel=2, stride=1 to smooth input
            x = torch.nn.functional.avg_pool2d(x, 2, 1, 0, False, True)

            # Split input into two channel-wise halves
            x1, x2 = x.chunk(2, 1)

            # First half through 3x3 conv (stride 2)
            x1 = self.cv1(x1)

            # Second half through 3x3 maxpool (stride 2) then 1x1 conv
            x2 = torch.nn.functional.max_pool2d(x2, 3, 2, 1)
            x2 = self.cv2(x2)

            # Concatenate the two processed halves along channel dimension
            return torch.cat((x1, x2), 1)



    class ELAN1(nn.Module):
        def __init__(self, c1, c2, c3, c4):  # c1: input channels, c2: output channels, c3: mid channels, c4: branch channels
            super().__init__()
            self.c = c3 // 2  # Split channel count for early branching

            # Initial 1x1 conv to expand input channels to c3
            self.cv1 = Conv(c1, c3, 1, 1)

            # Sequential convolutions for deep feature extraction on one half
            self.cv2 = Conv(c3 // 2, c4, 3, 1)  # 3x3 conv
            self.cv3 = Conv(c4, c4, 3, 1)       # another 3x3 conv
```

```
            # Final 1x1 conv to fuse all branches into output channels c2
            self.cv4 = Conv(c3 + (2 * c4), c2, 1, 1)

        def forward(self, x):
            # Split the cv1 output into two channel halves
            y = list(self.cv1(x).chunk(2, 1))

            # Apply two sequential convolutions to the second half and collect outputs
            y.extend(m(y[-1]) for m in [self.cv2, self.cv3])

            # Concatenate all branches and apply final 1x1 conv
            return self.cv4(torch.cat(y, 1))

        def forward_split(self, x):
            # Alternate splitting method using split() instead of chunk()
            y = list(self.cv1(x).split((self.c, self.c), 1))
            y.extend(m(y[-1]) for m in [self.cv2, self.cv3])
            return self.cv4(torch.cat(y, 1))



    class RepConvN(nn.Module):
        default_act = nn.SiLU()  # Default activation function

        def __init__(self, c1, c2, k=3, s=1, p=1, g=1, d=1, act=True, bn=False, deploy=False):
            super().__init__()
            assert k == 3 and p == 1  # Only supports 3x3 conv with padding 1
            self.g = g
            self.c1 = c1
            self.c2 = c2

            # Choose activation function
            self.act = self.default_act if act is True else act if isinstance(act, nn.Module) else nn.Identity()

            self.bn = None  # Optional BatchNorm identity branch (used in fusion)

            # Two convolution branches:
            self.conv1 = Conv(c1, c2, k, s, p=p, g=g, act=False)     # 3x3 convolution
            self.conv2 = Conv(c1, c2, 1, s, p=(p - k // 2), g=g, act=False)  # 1x1 convolution

        def forward_fuse(self, x):
            """Used after fusing all branches into a single conv"""
            return self.act(self.conv(x))

        def forward(self, x):
            """Forward pass before fusion (sum of 3x3, 1x1, and identity branches)"""
            id_out = 0 if self.bn is None else self.bn(x)
            return self.act(self.conv1(x) + self.conv2(x) + id_out)

        def get_equivalent_kernel_bias(self):
            """Fuse all branches (3x3, 1x1, and identity) into a single kernel and bias"""
            kernel3x3, bias3x3 = self._fuse_bn_tensor(self.conv1)
            kernel1x1, bias1x1 = self._fuse_bn_tensor(self.conv2)
            kernelid, biasid = self._fuse_bn_tensor(self.bn)
            # Sum all fused branches
            return kernel3x3 + self._pad_1x1_to_3x3_tensor(kernel1x1) + kernelid, bias3x3 + bias1x1 + biasid

        def _avg_to_3x3_tensor(self, avgp):
            """Convert average pooling into an equivalent 3x3 convolution kernel"""
            channels = self.c1
            groups = self.g
            kernel_size = avgp.kernel_size
            input_dim = channels // groups
            k = torch.zeros((channels, input_dim, kernel_size, kernel_size))
            k[np.arange(channels), np.tile(np.arange(input_dim), groups), :, :] = 1.0 / kernel_size ** 2
            return k

        def _pad_1x1_to_3x3_tensor(self, kernel1x1):
            """Pad 1x1 kernel to 3x3 by placing it at the center"""
            if kernel1x1 is None:
                return 0
            return torch.nn.functional.pad(kernel1x1, [1, 1, 1, 1])

        def _fuse_bn_tensor(self, branch):
            """Fuse batchnorm parameters with convolution weight and bias"""
            if branch is None:
                return 0, 0
```

```
            if isinstance(branch, Conv):
                # Get convolution and BN components
                kernel = branch.conv.weight
                running_mean = branch.bn.running_mean
                running_var = branch.bn.running_var
                gamma = branch.bn.weight
                beta = branch.bn.bias
                eps = branch.bn.eps
            elif isinstance(branch, nn.BatchNorm2d):
                # Identity branch BN: simulate identity kernel
                if not hasattr(self, 'id_tensor'):
                    input_dim = self.c1 // self.g
                    kernel_value = np.zeros((self.c1, input_dim, 3, 3), dtype=np.float32)
                    for i in range(self.c1):
                        kernel_value[i, i % input_dim, 1, 1] = 1
                    self.id_tensor = torch.from_numpy(kernel_value).to(branch.weight.device)
                kernel = self.id_tensor
                running_mean = branch.running_mean
                running_var = branch.running_var
                gamma = branch.weight
                beta = branch.bias
                eps = branch.eps

            # Fuse BN using affine transformation
            std = (running_var + eps).sqrt()
            t = (gamma / std).reshape(-1, 1, 1, 1)
            return kernel * t, beta - running_mean * gamma / std

        def fuse_convs(self):
            """Convert all parallel conv branches into a single Conv2d layer for inference"""
            if hasattr(self, 'conv'):
                return  # Already fused

            # Get combined kernel and bias
            kernel, bias = self.get_equivalent_kernel_bias()

            # Create new fused Conv2d layer with frozen weights
            self.conv = nn.Conv2d(
                in_channels=self.conv1.conv.in_channels,
                out_channels=self.conv1.conv.out_channels,
                kernel_size=self.conv1.conv.kernel_size,
                stride=self.conv1.conv.stride,
                padding=self.conv1.conv.padding,
                dilation=self.conv1.conv.dilation,
                groups=self.conv1.conv.groups,
                bias=True
            ).requires_grad_(False)

            self.conv.weight.data = kernel
            self.conv.bias.data = bias

            # Remove unused layers after fusion to reduce memory and computation
            for para in self.parameters():
                para.detach_()
            self.__delattr__('conv1')
            self.__delattr__('conv2')
            if hasattr(self, 'nm'):
                self.__delattr__('nm')
            if hasattr(self, 'bn'):
                self.__delattr__('bn')
            if hasattr(self, 'id_tensor'):
                self.__delattr__('id_tensor')



    class RepNBottleneck(nn.Module):
        # A bottleneck block using RepConvN for reparameterizable training/inference

        def __init__(self, c1, c2, shortcut=True, g=1, k=(3, 3), e=0.5):
            # c1: input channels, c2: output channels, e: expansion ratio
            super().__init__()
            c_ = int(c2 * e)  # Calculate hidden (bottleneck) channels

            # First layer: RepConvN with kernel k[0]
            self.cv1 = RepConvN(c1, c_, k[0], 1)

            # Second layer: regular Conv with kernel k[1], optional groups
```

```
            self.cv2 = Conv(c_, c2, k[1], 1, g=g)

            # Add shortcut only if enabled and input/output channels match
            self.add = shortcut and c1 == c2

        def forward(self, x):
            # Forward with residual connection if applicable
            return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1(x))


    class RepNCSP(nn.Module):
        # Cross Stage Partial (CSP) block with RepNBottleneck layers

        def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):
            super().__init__()
            c_ = int(c2 * e)  # Bottleneck channel count

            # Two branches: one for bottleneck block(s), one as shortcut
            self.cv1 = Conv(c1, c_, 1, 1)  # For bottleneck path
            self.cv2 = Conv(c1, c_, 1, 1)  # For shortcut path

            # Bottleneck sequence repeated `n` times
            self.m = nn.Sequential(*(RepNBottleneck(c_, c_, shortcut, g, e=1.0) for _ in range(n)))

            # Final 1x1 conv to merge the paths
            self.cv3 = Conv(2 * c_, c2, 1)

        def forward(self, x):
            # Apply bottleneck and shortcut paths, then concatenate and fuse
            return self.cv3(torch.cat((self.m(self.cv1(x)), self.cv2(x)), 1))


    class RepNCSPELAN4(nn.Module):
        # ELAN (Efficient Layer Aggregation Network) with RepNCSP and CSP-style splitting

        def __init__(self, c1, c2, c3, c4, c5=1):
            super().__init__()
            self.c = c3 // 2  # Half split of intermediate channels

            # Initial 1x1 conv to expand input
            self.cv1 = Conv(c1, c3, 1, 1)

            # Two sequential RepNCSP blocks with additional 3x3 conv
            self.cv2 = nn.Sequential(RepNCSP(c3 // 2, c4, c5), Conv(c4, c4, 3, 1))
            self.cv3 = nn.Sequential(RepNCSP(c4, c4, c5), Conv(c4, c4, 3, 1))

            # Final 1x1 conv to reduce concatenated features to desired output channels
            self.cv4 = Conv(c3 + (2 * c4), c2, 1, 1)

        def forward(self, x):
            # Split input features and process one side through multiple paths
            y = list(self.cv1(x).chunk(2, 1))  # Split channels
            y.extend((m(y[-1])) for m in [self.cv2, self.cv3])  # Chain processing
            return self.cv4(torch.cat(y, 1))  # Concatenate and fuse

        def forward_split(self, x):
            # Same as forward but uses split() instead of chunk() for fixed sizes
            y = list(self.cv1(x).split((self.c, self.c), 1))
            y.extend(m(y[-1]) for m in [self.cv2, self.cv3])
            return self.cv4(torch.cat(y, 1))



    class SP(nn.Module):
        def __init__(self, k=3, s=1):
            super(SP, self).__init__()
            # Standard MaxPool2d with configurable kernel and stride, and automatic same-padding
            self.m = nn.MaxPool2d(kernel_size=k, stride=s, padding=k // 2)

        def forward(self, x):
            # Apply max pooling
            return self.m(x)


    class SPPELAN(nn.Module):
        # SPP-ELAN: Combines spatial pyramid pooling with ELAN-style aggregation
```

```
    def __init__(self, c1, c2, c3):  # c1: in_channels, c2: out_channels, c3: internal_channels
        super().__init__()
        self.c = c3

        # Initial 1x1 conv to project input to c3 channels
        self.cv1 = Conv(c1, c3, 1, 1)

        # Apply max pooling (same kernel size) multiple times to simulate multi-scale context
        self.cv2 = SP(5)  # MaxPool2d(k=5)
        self.cv3 = SP(5)
        self.cv4 = SP(5)

        # Final 1x1 conv to compress concatenated features to c2 output channels
        self.cv5 = Conv(4 * c3, c2, 1, 1)

    def forward(self, x):
        y = [self.cv1(x)]  # Initial projection
        y.extend(m(y[-1]) for m in [self.cv2, self.cv3, self.cv4])  # Repeated pooling on last output
        return self.cv5(torch.cat(y, 1))  # Concatenate all and project
```

Define YOLOv9 Neck and Head layers

```
class CBLinear(nn.Module):
    def __init__(self, c1, c2s, k=1, s=1, p=None, g=1):
        super(CBLinear, self).__init__()
        self.c2s = c2s  # List of output channels for each split
        # Single convolution producing sum of all split outputs
        self.conv = nn.Conv2d(
            c1, sum(c2s), k, s, autopad(k, p), groups=g, bias=True
        )

    def forward(self, x):
        # Split the output tensor along channel dimension based on c2s
        outs = self.conv(x).split(self.c2s, dim=1)
        return outs  # Returns a list of feature maps


class CBFuse(nn.Module):
    def __init__(self, idx):
        super(CBFuse, self).__init__()
        self.idx = idx  # Index list specifying which tensor to select from each input

    def forward(self, xs):
        # Target size is the spatial shape (H, W) of the last tensor in the list
        target_size = xs[-1].shape[2:]

        # For all input sets except the last:
        #   pick the indexed feature, resize it to match the target shape
        res = [
            F.interpolate(x[self.idx[i]], size=target_size, mode='nearest')
            for i, x in enumerate(xs[:-1])
        ]

        # Stack resized tensors along a new dimension and add them with the last tensor
        out = torch.sum(torch.stack(res + xs[-1:]), dim=0)
        return out



class Concat(nn.Module):
    def __init__(self, dimension=1):
        super().__init__()
        self.d = dimension  # Dimension along which to concatenate tensors

    def forward(self, x):
        # Ensure input is a list or tuple of tensors
        if not isinstance(x, (list, tuple)):
            raise ValueError(f"Concat expects a list/tuple of tensors, got {type(x)}")

        # Concatenate all tensors along the specified dimension
        return torch.cat(x, dim=self.d)
```

```
class DFL(nn.Module):
    def __init__(self, reg_max=16):
        super().__init__()
        self.reg_max = reg_max
        # Create a projection vector [0, 1, ..., reg_max] and store it as a non-trainable buffer
        self.register_buffer('proj', torch.linspace(0, reg_max, reg_max + 1))

    def forward(self, x):
        b, c = x.shape[:2]  # batch size and channel count
        anchor_points = x.shape[2] if len(x.shape) > 2 else -1  # number of anchors (if present)

        # Compute actual_reg_max = number of bins per box side
        actual_reg_max = c // 4  # assuming 4 sides (l, t, r, b)

        # Reshape x into shape [batch, 4, bins, anchors]
        x = x.reshape(b, 4, actual_reg_max, anchor_points)

        # Apply softmax over the bins dimension (probabilities for each bin)
        x = F.softmax(x, dim=2)

        # Get appropriate projection vector (in case actual_reg_max != reg_max + 1)
        if actual_reg_max != self.reg_max + 1:
            proj = torch.linspace(0, actual_reg_max - 1, actual_reg_max, device=x.device)
        else:
            proj = self.proj  # use precomputed [0, 1, ..., reg_max]

        # Apply projection to convert probabilities to continuous distance predictions
        x = x * proj.view(1, 1, -1, 1)

        # Sum over the bins dimension to get final predicted box distances
        return x.sum(dim=2)



def dist2bbox(distance, anchor_points, xywh=True, dim=-1):
    """
    Convert distance predictions (left, top, right, bottom) to bounding boxes.

    Args:
        distance (Tensor): Predicted distances from anchor points (ltrb format).
        anchor_points (Tensor): Reference anchor points (x, y).
        xywh (bool): If True, return boxes in (x, y, w, h) format. If False, return (x1, y1, x2, y2).
        dim (int): The dimension along which the distance values are split and concatenated.

    Returns:
        Tensor: Bounding boxes in the desired format.
    """
    # Split distances into left-top and right-bottom
    lt, rb = torch.split(distance, 2, dim)

    # Calculate top-left and bottom-right corners of the box
    x1y1 = anchor_points - lt
    x2y2 = anchor_points + rb

    if xywh:
        # Convert to (center_x, center_y, width, height)
        center = (x1y1 + x2y2) / 2
        size = x2y2 - x1y1
        return torch.cat((center, size), dim)

    # Return in (x1, y1, x2, y2) format
    return torch.cat((x1y1, x2y2), dim)



def generate_anchor_points(features, strides, grid_cell_offset=0.5):
    """
    Generate anchor points and corresponding stride tensors from input feature maps.

    Args:
        features (List[Tensor]): List of feature map tensors from different scales.
        strides (List[int]): List of stride values corresponding to each feature level.
        grid_cell_offset (float): Offset applied to grid cell centers (default is 0.5 for center alignment).

    Returns:
        Tuple[Tensor, Tensor]: Concatenated anchor points (x, y) and stride tensor for each point.
    """
    anchor_points = []
```

```
        stride_tensors = []

        for i, stride in enumerate(strides):
            _, _, h, w = features[i].shape  # Extract height and width of the feature map

            # Generate grid coordinates
            x_coords = torch.arange(w, device=features[0].device) + grid_cell_offset
            y_coords = torch.arange(h, device=features[0].device) + grid_cell_offset
            y_grid, x_grid = torch.meshgrid(y_coords, x_coords, indexing='ij')  # Shape: [h, w]

            # Stack and flatten anchor points
            anchor = torch.stack((x_grid, y_grid), dim=-1).reshape(-1, 2)
            anchor_points.append(anchor)

            # Create a stride tensor for each anchor point
            stride_tensor = torch.full((h * w, 1), stride, device=features[0].device)
            stride_tensors.append(stride_tensor)

        return torch.cat(anchor_points, dim=0), torch.cat(stride_tensors, dim=0)



    def make_divisible(x, divisor):
        # Returns nearest x divisible by divisor
        if isinstance(divisor, torch.Tensor):
            divisor = int(divisor.max())  # to int
        return math.ceil(x / divisor) * divisor


    class Detect(nn.Module):
        # YOLO Detect head for detection models
        dynamic = False  # force grid reconstruction
        export = False  # export mode
        shape = None
        anchors = torch.empty(0)  # init
        strides = torch.empty(0)  # init

        def __init__(self, nc=80, ch=(), inplace=True):  # detection layer
            super().__init__()
            self.nc = nc  # number of classes
            self.nl = len(ch)  # number of detection layers
            self.reg_max = 16
            self.no = nc + self.reg_max * 4  # number of outputs per anchor
            self.inplace = inplace  # use inplace ops (e.g. slice assignment)
            self.stride = torch.zeros(self.nl)  # strides computed during build

            c2, c3 = max((ch[0] // 4, self.reg_max * 4, 16)), max((ch[0], min((self.nc * 2, 128))))  # channels
            self.cv2 = nn.ModuleList(
                nn.Sequential(Conv(x, c2, 3), Conv(c2, c2, 3), nn.Conv2d(c2, 4 * self.reg_max, 1)) for x in ch)
            self.cv3 = nn.ModuleList(
                nn.Sequential(Conv(x, c3, 3), Conv(c3, c3, 3), nn.Conv2d(c3, self.nc, 1)) for x in ch)
            self.dfl = DFL(self.reg_max) if self.reg_max > 1 else nn.Identity()

        def forward(self, x):
            shape = x[0].shape  # BCHW
            for i in range(self.nl):
                x[i] = torch.cat((self.cv2[i](x[i]), self.cv3[i](x[i])), 1)
            if self.training:
                return x
            elif self.dynamic or self.shape != shape:
                self.anchors, self.strides = (x.transpose(0, 1) for x in make_anchors(x, self.stride, 0.5))
                self.shape = shape

            box, cls = torch.cat([xi.view(shape[0], self.no, -1) for xi in x], 2).split((self.reg_max * 4, self.nc), 1)
            dbox = dist2bbox(self.dfl(box), self.anchors.unsqueeze(0), xywh=True, dim=1) * self.strides
            y = torch.cat((dbox, cls.sigmoid()), 1)
            return y if self.export else (y, x)

        def bias_init(self):
            # Initialize Detect() biases, WARNING: requires stride availability
            m = self  # self.model[-1]  # Detect() module
            # cf = torch.bincount(torch.tensor(np.concatenate(dataset.labels, 0)[:, 0]).long(), minlength=nc) + 1
            # ncf = math.log(0.6 / (m.nc - 0.999999)) if cf is None else torch.log(cf / cf.sum())  # nominal class frequency
            for a, b, s in zip(m.cv2, m.cv3, m.stride):  # from
                a[-1].bias.data[:] = 1.0  # box
                b[-1].bias.data[:m.nc] = math.log(5 / m.nc / (640 / s) ** 2)  # cls (5 objects and 80 classes per 640 image)
```

```
class DualDDetect(nn.Module):
    # Dual-path detection head for YOLOv9: standard + auxiliary branch
    dynamic = False   # Controls dynamic grid reconstruction (not used here)
    export = False    # If True, returns only predictions (used for ONNX/engine export)
    shape = None      # Used for export
    anchors = torch.empty(0)  # Placeholder for anchor points
    strides = torch.empty(0)  # Placeholder for stride values

    def __init__(self, nc=80, ch=(), inplace=True):
        """
        Args:
            nc (int): number of classes
            ch (List[int]): list of input channel sizes from backbone/FPN (2 * num_layers)
            inplace (bool): whether to use in-place operations
        """
        super().__init__()
        self.nc = nc
        self.nl = len(ch) // 2  # number of detection levels (e.g., 3 levels → 6 channels → 3 per branch)
        self.reg_max = 16       # for DFL (Distribution Focal Loss)
        self.no = nc + self.reg_max * 4  # outputs per anchor: class + 4 sides
        self.inplace = inplace
        self.stride = torch.zeros(self.nl)  # stride placeholder

        # Calculate intermediate conv channel sizes
        c2 = make_divisible(max((ch[0] // 4, self.reg_max * 4, 16)), 4)
        c3 = max(ch[0], min(self.nc * 2, 128))
        c4 = make_divisible(max((ch[self.nl] // 4, self.reg_max * 4, 16)), 4)
        c5 = max(ch[self.nl], min(self.nc * 2, 128))

        # Main detection branches (box + class) for levels 0 to nl-1
        self.cv2 = nn.ModuleList(  # box regression
            nn.Sequential(Conv(x, c2, 3), Conv(c2, c2, 3, g=4), nn.Conv2d(c2, 4 * self.reg_max, 1, groups=4))
            for x in ch[:self.nl])
        self.cv3 = nn.ModuleList(  # class prediction
            nn.Sequential(Conv(x, c3, 3), Conv(c3, c3, 3), nn.Conv2d(c3, self.nc, 1))
            for x in ch[:self.nl])

        # Auxiliary detection branches for deeper supervision
        self.cv4 = nn.ModuleList(  # box regression
            nn.Sequential(Conv(x, c4, 3), Conv(c4, c4, 3, g=4), nn.Conv2d(c4, 4 * self.reg_max, 1, groups=4))
            for x in ch[self.nl:])
        self.cv5 = nn.ModuleList(  # class prediction
            nn.Sequential(Conv(x, c5, 3), Conv(c5, c5, 3), nn.Conv2d(c5, self.nc, 1))
            for x in ch[self.nl:])

        self.dfl = DFL(self.reg_max)   # used in main path
        self.dfl2 = DFL(self.reg_max)  # used in auxiliary path

    def forward(self, x):
        shape = x[0].shape  # get input shape for batch size and H/W
        d1, d2 = [], []  # for main and auxiliary predictions

        # Forward through detection branches
        for i in range(self.nl):
            # Main detection head
            feat1 = self.cv2[i](x[i])         # box
            feat2 = self.cv3[i](x[i])         # class
            d1.append(torch.cat((feat1, feat2), 1))

            # Auxiliary detection head
            feat3 = self.cv4[i](x[self.nl + i])
            feat4 = self.cv5[i](x[self.nl + i])
            d2.append(torch.cat((feat3, feat4), 1))

        if self.training:
            # Return raw predictions for loss computation
            return [d1, d2]

        # Inference mode:
        # Generate anchor points and strides for decoding boxes
        self.anchors, self.strides = (t.transpose(0, 1) for t in generate_anchor_points(d1, self.stride, 0.5))

        # Flatten all spatial predictions from all levels for the main path
        cat_features = torch.cat([di.view(shape[0], -1, di.shape[2] * di.shape[3]) for di in d1], 2)

        # Split into bbox regression and class probabilities
        total_channels = cat_features.shape[1]
```

```
            box_channels = total_channels - self.nc
            box, cls = cat_features.split([box_channels, self.nc], dim=1)

            # Decode box distribution to distances
            box_dfl = self.dfl(box)
            dbox = dist2bbox(box_dfl, self.anchors.unsqueeze(0), xywh=True, dim=1) * self.strides

            # Same process for auxiliary path
            cat_features2 = torch.cat([di.view(shape[0], -1, di.shape[2] * di.shape[3]) for di in d2], 2)
            box2, cls2 = cat_features2.split([box_channels, self.nc], dim=1)
            box2_dfl = self.dfl2(box2)
            dbox2 = dist2bbox(box2_dfl, self.anchors.unsqueeze(0), xywh=True, dim=1) * self.strides

            # Scale class confidences and concatenate with boxes
            cls_scores = cls.sigmoid() * 0.8
            cls2_scores = cls2.sigmoid() * 0.8
            y = [torch.cat((dbox, cls_scores), 1), torch.cat((dbox2, cls2_scores), 1)]

            return y if self.export else (y, [d1, d2])

        def bias_init(self):
            """
            Initialize biases in the final conv layers.
            Encourages the model to predict more background initially.
            """
            m = self
            for a, b, s in zip(m.cv2, m.cv3, m.stride):
                a[-1].bias.data[:] = 1.0  # regression bias
                b[-1].bias.data[:m.nc] = math.log(5 / m.nc / (640 / s) ** 2)

            for a, b, s in zip(m.cv4, m.cv5, m.stride):
                a[-1].bias.data[:] = 1.0  # regression bias
                b[-1].bias.data[:m.nc] = math.log(5 / m.nc / (640 / s) ** 2)



    class Segment(Detect):
        # YOLO Segment head for segmentation models
        def __init__(self, nc=80, nm=32, npr=256, ch=(), inplace=True):
            super().__init__(nc, ch, inplace)
            self.nm = nm  # number of masks
            self.npr = npr  # number of protos
            self.proto = Proto(ch[0], self.npr, self.nm)  # protos
            self.detect = Detect.forward

            c4 = max(ch[0] // 4, self.nm)
            self.cv4 = nn.ModuleList(nn.Sequential(Conv(x, c4, 3), Conv(c4, c4, 3), nn.Conv2d(c4, self.nm, 1)) for x in ch)

        def forward(self, x):
            p = self.proto(x[0])
            bs = p.shape[0]

            mc = torch.cat([self.cv4[i](x[i]).view(bs, self.nm, -1) for i in range(self.nl)], 2)  # mask coefficients
            x = self.detect(self, x)
            if self.training:
                return x, mc, p
            return (torch.cat([x, mc], 1), p) if self.export else (torch.cat([x[0], mc], 1), (x[1], mc, p))


    class DualDSegment(DualDDetect):
        """
        Segmentation head built on top of DualDDetect for YOLO models.
        Adds proto-mask generation and mask coefficient prediction.
        """

        def __init__(self, nc=80, nm=32, npr=256, ch=(), inplace=True):
            """
            Args:
                nc (int): Number of classes.
                nm (int): Number of masks (mask coefficients per detection).
                npr (int): Number of prototype masks.
                ch (List[int]): List of input channel sizes for each head.
                inplace (bool): Whether to use in-place ops in activations.
            """
            super().__init__(nc=nc, ch=ch[:-2], inplace=inplace)

            self.nl = (len(ch) - 2) // 2  # Number of detection levels
            self.nm = nm                  # Number of mask coefficients
```

```
            self.npr = npr                # Number of prototypes (not directly used here)

            # Prototype mask generators (usually at the end of the backbone/FPN)
            self.proto = Conv(ch[-2], self.nm, 1)    # from second last channel input
            self.proto2 = Conv(ch[-1], self.nm, 1)   # from last channel input

            # Use DualDDetect's forward method for detection
            self.detect = DualDDetect.forward

            # Mask coefficient prediction layers (head layers)
            c6 = max(ch[0] // 4, self.nm)               # channel size for cv6 branch
            c7 = max(ch[self.nl] // 4, self.nm)         # channel size for cv7 branch

            # Mask coeffs for main detection levels
            self.cv6 = nn.ModuleList(
                nn.Sequential(
                    Conv(x, c6, 3),
                    Conv(c6, c6, 3),
                    nn.Conv2d(c6, self.nm, 1)
                ) for x in ch[:self.nl]
            )

            # Mask coeffs for auxiliary detection levels
            self.cv7 = nn.ModuleList(
                nn.Sequential(
                    Conv(x, c7, 3),
                    Conv(c7, c7, 3),
                    nn.Conv2d(c7, self.nm, 1)
                ) for x in ch[self.nl:self.nl * 2]
            )

        def forward(self, x):
            """
            Forward pass for training and inference.

            Args:
                x (List[Tensor]): Feature maps from different backbone layers.

            Returns:
                During training:
                    Tuple[detection_output, mask_coeffs, prototypes]
                During inference:
                    Tuple[concat(mask_cls, mask_coeffs), (aux_cls, aux_coeffs, aux_protos)]
            """
            # Generate mask prototypes
            p = [self.proto(x[-2]), self.proto2(x[-1])]
            bs = p[0].shape[0]  # batch size

            # Predict mask coefficients for each detection level
            mc = [
                torch.cat([
                    self.cv6[i](x[i]).view(bs, self.nm, -1) for i in range(self.nl)
                ], dim=2),
                torch.cat([
                    self.cv7[i](x[self.nl + i]).view(bs, self.nm, -1) for i in range(self.nl)
                ], dim=2)
            ]

            # Run detection using the inherited DualDDetect.forward
            d = self.detect(self, x[:-2])  # pass everything except the proto inputs

            if self.training:
                # Return raw outputs for loss computation
                return d, mc, p

            # In inference: return concatenated predictions with mask coefficients
            return (
                torch.cat([d[0][1], mc[1]], dim=1),  # main predictions + mask coeffs
                (d[1][1], mc[1], p[1])               # auxiliary output and segmentation outputs
            )



    class DDetect(nn.Module):
        # YOLO Detect head for detection models
        dynamic = False  # force grid reconstruction
        export = False  # export mode
```

```
        shape = None
        anchors = torch.empty(0)  # init
        strides = torch.empty(0)  # init

        def __init__(self, nc=80, ch=(), inplace=True):  # detection layer
            super().__init__()
            self.nc = nc  # number of classes
            self.nl = len(ch)  # number of detection layers
            self.reg_max = 16
            self.no = nc + self.reg_max * 4  # number of outputs per anchor
            self.inplace = inplace  # use inplace ops (e.g. slice assignment)
            self.stride = torch.zeros(self.nl)  # strides computed during build

            c2, c3 = make_divisible(max((ch[0] // 4, self.reg_max * 4, 16)), 4), max((ch[0], min((self.nc * 2, 128))))  # channels
            self.cv2 = nn.ModuleList(
                nn.Sequential(Conv(x, c2, 3), Conv(c2, c2, 3, g=4), nn.Conv2d(c2, 4 * self.reg_max, 1, groups=4)) for x in ch)
            self.cv3 = nn.ModuleList(
                nn.Sequential(Conv(x, c3, 3), Conv(c3, c3, 3), nn.Conv2d(c3, self.nc, 1)) for x in ch)
            self.dfl = DFL(self.reg_max) if self.reg_max > 1 else nn.Identity()

        def forward(self, x):
            shape = x[0].shape  # BCHW
            for i in range(self.nl):
                x[i] = torch.cat((self.cv2[i](x[i]), self.cv3[i](x[i])), 1)
            if self.training:
                return x
            elif self.dynamic or self.shape != shape:
                self.anchors, self.strides = (x.transpose(0, 1) for x in make_anchors(x, self.stride, 0.5))
                self.shape = shape

            box, cls = torch.cat([xi.view(shape[0], self.no, -1) for xi in x], 2).split((self.reg_max * 4, self.nc), 1)
            dbox = dist2bbox(self.dfl(box), self.anchors.unsqueeze(0), xywh=True, dim=1) * self.strides
            y = torch.cat((dbox, cls.sigmoid()), 1)
            return y if self.export else (y, x)

        def bias_init(self):
            # Initialize Detect() biases, WARNING: requires stride availability
            m = self  # self.model[-1]  # Detect() module
            # cf = torch.bincount(torch.tensor(np.concatenate(dataset.labels, 0)[:, 0]).long(), minlength=nc) + 1
            # ncf = math.log(0.6 / (m.nc - 0.999999)) if cf is None else torch.log(cf / cf.sum())  # nominal class frequency
            for a, b, s in zip(m.cv2, m.cv3, m.stride):  # from
                a[-1].bias.data[:] = 1.0  # box
                b[-1].bias.data[:m.nc] = math.log(5 / m.nc / (640 / s) ** 2)  # cls (5 objects and 80 classes per 640 image)
```

YOLOv9C Custom Configuration

```
yolov9_config = {
    "nc": 80,  # Number of classes
    "depth_multiple": 1.0,
    "width_multiple": 1.0,
    "anchors": 3,  # Number of anchors

    # Backbone
    "backbone": [
        [-1, 1, "Silence", []],                         # 0
        [-1, 1, "Conv", [64, 3, 2]],                    # 1 - P1/2
        [-1, 1, "Conv", [128, 3, 2]],                   # 2 - P2/4
        [-1, 1, "RepNCSPELAN4", [256, 128, 64, 1]],     # 3
        [-1, 1, "ADown", [256]],                        # 4 - P3/8
        [-1, 1, "RepNCSPELAN4", [512, 256, 128, 1]],    # 5
        [-1, 1, "ADown", [512]],                        # 6 - P4/16
        [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],    # 7
        [-1, 1, "ADown", [512]],                        # 8 - P5/32
        [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],    # 9
    ],

    # Head
    "head": [
        [-1, 1, "SPPELAN", [512, 256]],

        # Upsample and concat with backbone P4
        [-1, 1, "nn.Upsample", [None, 2, "nearest"]],
```

```
            [[-1, 7], 1, "Concat", [1]],
            [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],

            # Upsample and concat with backbone P3
            [-1, 1, "nn.Upsample", [None, 2, "nearest"]],
            [[-1, 5], 1, "Concat", [1]],
            [-1, 1, "RepNCSPELAN4", [256, 256, 128, 1]],   # 16 (P3/8-small)

            # Downsample and fuse P4
            [-1, 1, "ADown", [256]],
            [[-1, 13], 1, "Concat", [1]],
            [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],   # 19 (P4/16-medium)

            # Downsample and fuse P5
            [-1, 1, "ADown", [512]],
            [[-1, 10], 1, "Concat", [1]],
            [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],   # 22 (P5/32-large)

            # Auxiliary CBLinear branches from backbone
            [5, 1, "CBLinear", [[256]]],                 # 23
            [7, 1, "CBLinear", [[256, 512]]],            # 24
            [9, 1, "CBLinear", [[256, 512, 512]]],       # 25

            # Auxiliary fusion path
            [0, 1, "Conv", [64, 3, 2]],                  # 26 - P1/2
            [-1, 1, "Conv", [128, 3, 2]],                # 27 - P2/4
            [-1, 1, "RepNCSPELAN4", [256, 128, 64, 1]],  # 28
            [-1, 1, "ADown", [256]],                     # 29 - P3/8
            [[23, 24, 25, -1], 1, "CBFuse", [[0, 0, 0]]], # 30
            [-1, 1, "RepNCSPELAN4", [512, 256, 128, 1]],  # 31
            [-1, 1, "ADown", [512]],                     # 32 - P4/16
            [[24, 25, -1], 1, "CBFuse", [[1, 1]]],       # 33
            [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],  # 34
            [-1, 1, "ADown", [512]],                     # 35 - P5/32
            [[25, -1], 1, "CBFuse", [[2]]],              # 36
            [-1, 1, "RepNCSPELAN4", [512, 512, 256, 1]],  # 37

            # Final Detection Head
            [[31, 34, 37, 16, 19, 22], 1, "DualDDetect", ["nc"]],  # 38
        ]
    }


    def apply_custom_initialization(model):
        """
        Custom initialization and parameter adjustments for certain layer types.
        """
        for module in model.modules():
            if isinstance(module, nn.Conv2d):
                pass  # Placeholder for Conv2d weight init, e.g., kaiming_normal_
            elif isinstance(module, nn.BatchNorm2d):
                module.eps = 1e-3
                module.momentum = 0.03
            elif isinstance(module, (nn.Hardswish, nn.LeakyReLU, nn.ReLU, nn.ReLU6, nn.SiLU)):
                module.inplace = True
```

Model Summary

```
    from pathlib import Path
    from copy import deepcopy
    import torch

    def summarize_model_architecture(model, verbose=False, input_size=640):
        """
        Prints a summary of the PyTorch model including:
        - Total parameters and gradients
        - Optional per-layer parameter statistics
        - Estimated FLOPs (floating point operations)
        - Layer/module count

        Args:
            model (nn.Module): The PyTorch model.
            verbose (bool): If True, print detailed layer-by-layer info.
            input_size (int or tuple): Input resolution used for FLOPs estimation.
        """
        # Total parameter and gradient counts
```

```
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

    # Optional verbose layer info
    if verbose:
        print(f"{'Layer':>5} {'Parameter Name':>40} {'Grad':>9} {'Params':>12} {'Shape':>20} {'Mean':>10} {'Std':>10}")
        for i, (name, param) in enumerate(model.named_parameters()):
            print(f"{i:5} {name:40} {str(param.requires_grad):>9} {param.numel():12} {str(list(param.shape)):>20} "
                  f"{param.mean():10.3g} {param.std():10.3g}")

    # FLOPs estimation
    try:
        first_param = next(model.parameters())
        stride = max(int(model.stride.max()), 32) if hasattr(model, 'stride') else 32
        dummy_input = torch.empty((1, first_param.shape[1], stride, stride), device=first_param.device)

        flops = thop.profile(deepcopy(model), inputs=(dummy_input,), verbose=False)[0]
        flops *= 2 / 1e9  # convert to GFLOPs with multiply-accumulate operations

        # Adjust FLOPs to actual input size
        input_size = input_size if isinstance(input_size, (list, tuple)) else [input_size, input_size]
        flops_scaled = flops * input_size[0] / stride * input_size[1] / stride
        flops_str = f", {flops_scaled:.1f} GFLOPs"
    except Exception:
        flops_str = ''

    # Model name from file (if available)
    #model_name = Path(model.yaml_file).stem.replace('yolov5', 'YOLOv5') if hasattr(model, 'yaml_file') else 'Model'
    model_name = model.__class__.__name__

    # Count all layers
    total_modules = sum(1 for _ in model.modules())

    # Final summary output
    print(f"\n{model_name} summary: {len(model.model)} top-level layers, "
          f"{total_modules} total PyTorch layers, "
          f"{total_params} parameters, {trainable_params} gradients{flops_str}")
```

Wraps modules into a BaseModel class and dynamically builds models from parsed configs. Supports both backbone and head construction.

```
class BaseModel(nn.Module):
    """
    YOLO Base Model Class
    - Supports profiling and intermediate feature saving
    - Designed to work with detection and segmentation heads
    """

    def forward(self, x, profile=False, visualize=False):
        return self._forward_once(x, profile, visualize)

    def _forward_once(self, x, profile=False, visualize=False):
        y, dt = [], []
        for m in self.model:
            if m.f != -1:
                x = y[m.f] if isinstance(m.f, int) else [x if j == -1 else y[j] for j in m.f]
            if profile:
                self._profile_one_layer(m, x, dt)
            x = m(x)
            y.append(x if m.i in self.save else None)
        return x

    def _profile_one_layer(self, m, x, dt):
        """Profile a single layer for FLOPs and runtime"""
        is_last = m == self.model[-1]
        flops = (
            thop.profile(m, inputs=(x.copy() if is_last else x,), verbose=False)[0] / 1E9 * 2
            if thop else 0
        )

        start_time = time_sync()
        for _ in range(10):
            m(x.copy() if is_last else x)
        elapsed = (time_sync() - start_time) * 100
        dt.append(elapsed)
```

```
            # Fuse modules post-profiling
            for layer in self.model.modules():
                if isinstance(layer, RepConvN) and hasattr(layer, 'fuse_convs'):
                    layer.fuse_convs()
                    layer.forward = layer.forward_fuse
                if isinstance(layer, (Conv, DWConv)) and hasattr(layer, 'bn'):
                    layer.conv = fuse_conv_and_bn(layer.conv, layer.bn)
                    delattr(layer, 'bn')
                    layer.forward = layer.forward_fuse

            self.info()
            return self

        def info(self, verbose=False, img_size=640):
            summarize_model_architecture(self, verbose, img_size)

        def _apply(self, fn):
            self = super()._apply(fn)
            detect = self.model[-1]
            if isinstance(detect, (DualDDetect, DualDSegment)):
                detect.stride = fn(detect.stride)
                detect.anchors = fn(detect.anchors)
                detect.strides = fn(detect.strides)
            return self


    class DetectionModel(BaseModel):
        """
        YOLO Detection Model that supports:
        - Loading from dict or YAML
        - Anchor/stride auto-calculation
        - Augmented inference
        """

        def __init__(self, cfg='yolo.yaml', ch=3, nc=None, anchors=None):
            super().__init__()

            # Load model configuration
            if isinstance(cfg, dict):
                self.yaml = cfg
            else:
                import yaml
                self.yaml_file = Path(cfg).name
                with open(cfg, encoding='ascii', errors='ignore') as f:
                    self.yaml = yaml.safe_load(f)

            # Override YAML config with args
            ch = self.yaml['ch'] = self.yaml.get('ch', ch)
            if nc and nc != self.yaml['nc']:
                self.yaml['nc'] = nc
            if anchors:
                self.yaml['anchors'] = round(anchors)

            # Parse model
            self.model, self.save = build_model_from_config(deepcopy(self.yaml), [ch])
            self.names = [str(i) for i in range(self.yaml['nc'])]
            self.inplace = self.yaml.get('inplace', True)

            # Setup anchor scaling and strides for detect head
            detect = self.model[-1]
            if isinstance(detect, DualDDetect):
                s = 256
                detect.inplace = self.inplace
                dummy_input = torch.zeros(1, ch, s, s)
                if isinstance(detect, DualDSegment):
                    detect.stride = torch.tensor([s / x.shape[-2] for x in self.forward(dummy_input)[0]])
                else:
                    detect.stride = torch.tensor([s / x.shape[-2] for x in self.forward(dummy_input)[0]])
                self.stride = detect.stride
                detect.bias_init()

            apply_custom_initialization(self)
            self.info()

        def forward(self, x, augment=False, profile=False, visualize=False):
            return self._forward_augment(x) if augment else self._forward_once(x, profile, visualize)
```

```
        def _forward_augment(self, x):
            img_size = x.shape[-2:]
            scales = [1.0, 0.83, 0.67]
            flips = [None, 3, None]  # 3 = horizontal flip
            outputs = []

            for scale, flip in zip(scales, flips):
                x_scaled = scale_img(x.flip(flip) if flip else x, scale, gs=int(self.stride.max()))
                y_scaled = self._forward_once(x_scaled)[0]
                y_scaled = self._descale_pred(y_scaled, flip, scale, img_size)
                outputs.append(y_scaled)

            return torch.cat(self._clip_augmented(outputs), 1), None

        def _descale_pred(self, p, flip, scale, img_size):
            if self.inplace:
                p[..., :4] /= scale
                if flip == 2:
                    p[..., 1] = img_size[0] - p[..., 1]
                elif flip == 3:
                    p[..., 0] = img_size[1] - p[..., 0]
            else:
                x = p[..., 0:1] / scale
                y = p[..., 1:2] / scale
                wh = p[..., 2:4] / scale
                if flip == 2:
                    y = img_size[0] - y
                elif flip == 3:
                    x = img_size[1] - x
                p = torch.cat((x, y, wh, p[..., 4:]), dim=-1)
            return p

        def _clip_augmented(self, preds):
            nl = self.model[-1].nl
            grid_pts = sum(4 ** i for i in range(nl))
            e = 1

            # Clip the "tail" (smallest scale) from the first prediction
            idx_clip1 = (preds[0].shape[1] // grid_pts) * sum(4 ** i for i in range(e))
            preds[0] = preds[0][:, :-idx_clip1]

            # Clip the "head" (largest scale) from the last prediction
            idx_clip2 = (preds[-1].shape[1] // grid_pts) * sum(4 ** (nl - 1 - i) for i in range(e))
            preds[-1] = preds[-1][:, idx_clip2:]

            return preds


    # Alias
    Model = DetectionModel
```

Build Model from Config - Code to initialize the model using the provided config (YOLOv9) and construct it layer by layer.

```
    def build_model_from_config(config, channels):
        """
        Parses the model configuration and constructs the model layers.

        Args:
            config (dict): Model configuration with 'anchors', 'nc', 'depth_multiple', 'width_multiple', etc.
            channels (List[int]): List of channel sizes from previous layers.

        Returns:
            nn.Sequential: The constructed model.
            List[int]: Indices of layers whose outputs need to be saved.
        """
        anchors = config['anchors']
        nc = config['nc']
        depth_mult = config['depth_multiple']
        width_mult = config['width_multiple']
        act = config.get('activation')

        # Set default activation function if specified
        if act:
            Conv.default_act = eval(act)
            RepConvN.default_act = eval(act)
```

```
    num_anchors = (len(anchors[0]) // 2) if isinstance(anchors, list) else anchors
    num_outputs = num_anchors * (nc + 5)  # 5 = x, y, w, h, obj_conf

    layers = []
    save_layers = []
    c2 = channels[-1]

    for i, (from_, num, module, args) in enumerate(config['backbone'] + config['head']):
        module = eval(module) if isinstance(module, str) else module

        # Evaluate string arguments
        for j, arg in enumerate(args):
            with contextlib.suppress(NameError):
                args[j] = eval(arg) if isinstance(arg, str) else arg

        # Adjust number of repeats based on depth multiple
        num_repeats = max(round(num * depth_mult), 1) if num > 1 else num

        # Handle module argument formatting and channel computations
        if module in {Conv, AConv, nn.ConvTranspose2d, ADown, ELAN1, RepNCSPELAN4, SPPELAN}:
            c1, c2 = channels[from_], args[0]
            if c2 != num_outputs:
                c2 = make_divisible(c2 * width_mult, 8)
            args = [c1, c2, *args[1:]]

        elif module is nn.BatchNorm2d:
            args = [channels[from_]]

        elif module is Concat:
            c2 = sum(channels[x] for x in from_)

        elif module is CBLinear:
            c1, c2s = channels[from_], args[0]
            args = [c1, c2s, *args[1:]]

        elif module is CBFuse:
            c2 = channels[from_[-1]]

        elif module is DualDDetect:
            nc_ = args[0]
            ch_in = [channels[x] for x in from_]
            module_instance = module(nc=nc_, ch=ch_in)
            c2 = sum([nc_ + 4] * len(from_))  # Dummy placeholder output shape
        else:
            c2 = channels[from_]

        # Instantiate the module(s) unless it's already created (like DualDDetect)
        if module is not DualDDetect:
            module_instance = (
                nn.Sequential(*(module(*args) for _ in range(num_repeats)))
                if num_repeats > 1 else module(*args)
            )

        # Attach metadata to the module
        module_instance.i = i
        module_instance.f = from_
        module_instance.type = str(module)[8:-2].replace('__main__.', '')
        module_instance.np = sum(p.numel() for p in module_instance.parameters())

        # Track which layers need to be saved
        save_from = [from_] if isinstance(from_, int) else from_
        save_layers.extend(x % i for x in save_from if x != -1)

        layers.append(module_instance)

        # First layer resets channels list
        if i == 0:
            channels = []

        channels.append(c2)

    return nn.Sequential(*layers), sorted(save_layers)
```

Download official pretrained weights

```
!wget https://github.com/WongKinYiu/yolov9/releases/download/v0.1/yolov9-c.pt
```

```
--2025-05-04 19:49:30--  https://github.com/WongKinYiu/yolov9/releases/download/v0.1/yolov9-c.pt
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/759338070/c8ca43f2-0d2d-4aa3-a074-426505bfbfb1?X-
--2025-05-04 19:49:30--  https://objects.githubusercontent.com/github-production-release-asset-2e65be/759338070/c8ca43f2-0d2d-4aa3-a074-
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.109.133, 185.199.108.133, 185.199.111.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.109.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 103153312 (98M) [application/octet-stream]
Saving to: ‘yolov9-c.pt’

yolov9-c.pt         100%[===================>]  98.37M   131MB/s    in 0.8s

2025-05-04 19:49:31 (131 MB/s) - ‘yolov9-c.pt’ saved [103153312/103153312]
```

Load the weights to the custom model

```
from collections import OrderedDict
import torch

# Step 1: Initialize model from config
input_channels = [3]
layers, save = build_model_from_config(yolov9_config, input_channels)
model = Model(yolov9_config)  # Assumes DetectionModel is aliased as Model

# Step 2: Load YOLOv9 checkpoint (replace with your actual path)
checkpoint_path = '/content/yolov9-c.pt'
checkpoint = torch.load(checkpoint_path, map_location='cpu', weights_only=False)

# Step 3: Extract state dict and add 'model.' prefix if missing
original_state_dict = checkpoint['model'].state_dict()

state_dict = OrderedDict()
for k, v in original_state_dict.items():
    key = f'model.{k}' if not k.startswith('model.') else k
    state_dict[key] = v

# Step 4: Load weights into your model (allow non-strict to bypass minor mismatches)
missing_keys, unexpected_keys = model.load_state_dict(state_dict, strict=False)

# Step 5: Debug output
print(f"Missing keys: {missing_keys}")
print(f"Unexpected keys: {unexpected_keys}")

# Optional: run bias init manually if needed
# model.model[38].bias_init()

# Step 6: Set model to eval mode
model.eval()
```

Show hidden output

Helper Functions for Inference

Contains utility functions like xywh2xyxy, box_iou, scale_boxes, and non_max_suppression used for post-processing predictions, bounding box transformation, and IoU calculations.

```
def xywh2xyxy(x):
    # Convert nx4 boxes from [x, y, w, h] to [x1, y1, x2, y2] where xy1=top-left, xy2=bottom-right
    y = x.clone() if isinstance(x, torch.Tensor) else np.copy(x)
    y[..., 0] = x[..., 0] - x[..., 2] / 2  # top left x
    y[..., 1] = x[..., 1] - x[..., 3] / 2  # top left y
    y[..., 2] = x[..., 0] + x[..., 2] / 2  # bottom right x
    y[..., 3] = x[..., 1] + x[..., 3] / 2  # bottom right y
    return y
```

```
def box_iou(box1, box2, eps=1e-7):
    """
    Return intersection-over-union (Jaccard index) of boxes.
    Both sets of boxes are expected to be in (x1, y1, x2, y2) format.
    Arguments:
        box1 (Tensor[N, 4])
        box2 (Tensor[M, 4])
    Returns:
        iou (Tensor[N, M]): the NxM matrix containing the pairwise
            IoU values for every element in boxes1 and boxes2
    """

    # inter(N,M) = (rb(N,M,2) - lt(N,M,2)).clamp(0).prod(2)
    (a1, a2), (b1, b2) = box1.unsqueeze(1).chunk(2, 2), box2.unsqueeze(0).chunk(2, 2)
    inter = (torch.min(a2, b2) - torch.max(a1, b1)).clamp(0).prod(2)

    # IoU = inter / (area1 + area2 - inter)
    return inter / ((a2 - a1).prod(2) + (b2 - b1).prod(2) - inter + eps)


class Colors:
    def __init__(self):
        """
        Initializes a color palette using predefined hex codes.
        Converts each hex color to an RGB tuple.
        """
        # Custom vibrant hex color palette (20 unique colors)
        hex_colors = (
            'E6194B', '3CB44B', 'FFE119', '4363D8', 'F58231',
            '911EB4', '46F0F0', 'F032E6', 'BCF60C', 'FABEBE',
            '008080', 'E6BEFF', '9A6324', 'FFFAC8', '800000',
            'AAFFC3', '808000', 'FFD8B1', '000075', '808080'
        )
        self.palette = [self.hex2rgb(f'#{c}') for c in hex_colors]
        self.n = len(self.palette)

    def __call__(self, i, bgr=False):
            c = self.palette[int(i) % self.n]
            return (c[2], c[1], c[0]) if bgr else c

    @staticmethod
    def hex2rgb(h):  # rgb order (PIL)
            return tuple(int(h[1 + i:1 + i + 2], 16) for i in (0, 2, 4))


colors = Colors()  # create instance for 'from utils.plots import colors'


def colorstr(*input):
    *args, string = input if len(input) > 1 else ('blue', 'bold', input[0])
    colors = {
        'black': '\033[30m', 'red': '\033[31m', 'green': '\033[32m', 'yellow': '\033[33m',
        'blue': '\033[34m', 'magenta': '\033[35m', 'cyan': '\033[36m', 'white': '\033[37m',
        'bright_black': '\033[90m', 'bright_red': '\033[91m', 'bright_green': '\033[92m',
        'bright_yellow': '\033[93m', 'bright_blue': '\033[94m', 'bright_magenta': '\033[95m',
        'bright_cyan': '\033[96m', 'bright_white': '\033[97m',
        'end': '\033[0m', 'bold': '\033[1m', 'underline': '\033[4m'
    }
    return ''.join(colors[x] for x in args) + f'{string}' + colors['end']



def clip_boxes(boxes, shape):
    # Clip boxes (xyxy) to image shape (height, width)
    if isinstance(boxes, torch.Tensor):  # faster individually
        boxes[:, 0].clamp_(0, shape[1])  # x1
        boxes[:, 1].clamp_(0, shape[0])  # y1
        boxes[:, 2].clamp_(0, shape[1])  # x2
        boxes[:, 3].clamp_(0, shape[0])  # y2
    else:  # np.array (faster grouped)
        boxes[:, [0, 2]] = boxes[:, [0, 2]].clip(0, shape[1])  # x1, x2
        boxes[:, [1, 3]] = boxes[:, [1, 3]].clip(0, shape[0])  # y1, y2


def scale_boxes(img1_shape, boxes, img0_shape, ratio_pad=None):
    # Rescale boxes (xyxy) from img1_shape to img0_shape
    if ratio_pad is None:  # calculate from img0_shape
```

```
            gain = min(img1_shape[0] / img0_shape[0], img1_shape[1] / img0_shape[1])  # gain  = old / new
            pad = (img1_shape[1] - img0_shape[1] * gain) / 2, (img1_shape[0] - img0_shape[0] * gain) / 2  # wh padding
        else:
            gain = ratio_pad[0][0]
            pad = ratio_pad[1]

        boxes[:, [0, 2]] -= pad[0]  # x padding
        boxes[:, [1, 3]] -= pad[1]  # y padding
        boxes[:, :4] /= gain
        clip_boxes(boxes, img0_shape)
        return boxes


    from PIL import Image, ImageDraw, ImageFont

    FONT = 'Arial.ttf'

    def is_ascii(s=''):
        # Is string composed of all ASCII (no UTF) characters? (note str().isascii() introduced in python 3.7)
        s = str(s)  # convert list, tuple, None, etc. to str
        return len(s.encode().decode('ascii', 'ignore')) == len(s)

    def scale_image(im1_shape, masks, im0_shape, ratio_pad=None):
        """
        img1_shape: model input shape, [h, w]
        img0_shape: origin pic shape, [h, w, 3]
        masks: [h, w, num]
        """
        # Rescale coordinates (xyxy) from im1_shape to im0_shape
        if ratio_pad is None:  # calculate from im0_shape
            gain = min(im1_shape[0] / im0_shape[0], im1_shape[1] / im0_shape[1])  # gain  = old / new
            pad = (im1_shape[1] - im0_shape[1] * gain) / 2, (im1_shape[0] - im0_shape[0] * gain) / 2  # wh padding
        else:
            pad = ratio_pad[1]
        top, left = int(pad[1]), int(pad[0])  # y, x
        bottom, right = int(im1_shape[0] - pad[1]), int(im1_shape[1] - pad[0])

        if len(masks.shape) < 2:
            raise ValueError(f'"len of masks shape" should be 2 or 3, but got {len(masks.shape)}')
        masks = masks[top:bottom, left:right]
        # masks = masks.permute(2, 0, 1).contiguous()
        # masks = F.interpolate(masks[None], im0_shape[:2], mode='bilinear', align_corners=False)[0]
        # masks = masks.permute(1, 2, 0).contiguous()
        masks = cv2.resize(masks, (im0_shape[1], im0_shape[0]))

        if len(masks.shape) == 2:
            masks = masks[:, :, None]
        return masks

    def check_pil_font(font=FONT, size=10):
        # Return a PIL TrueType Font, downloading to CONFIG_DIR if necessary
        font = Path(font)
        font = font if font.exists() else (CONFIG_DIR / font.name)
        try:
            return ImageFont.truetype(str(font) if font.exists() else font.name, size)
        except Exception:  # download if missing
            try:
                check_font(font)
                return ImageFont.truetype(str(font), size)
            except TypeError:
                check_requirements('Pillow>=8.4.0')  # known issue https://github.com/ultralytics/yolov5/issues/5374
            except URLError:  # not online
                return ImageFont.load_default()


    class Annotator:
        # YOLOv5 Annotator for train/val mosaics and jpgs and detect/hub inference annotations
        def __init__(self, im, line_width=None, font_size=None, font='Arial.ttf', pil=False, example='abc'):
            assert im.data.contiguous, 'Image not contiguous. Apply np.ascontiguousarray(im) to Annotator() input images.'
            non_ascii = not is_ascii(example)  # non-latin labels, i.e. asian, arabic, cyrillic
            self.pil = pil or non_ascii
            if self.pil:  # use PIL
                self.im = im if isinstance(im, Image.Image) else Image.fromarray(im)
                self.draw = ImageDraw.Draw(self.im)
                self.font = check_pil_font(font='Arial.Unicode.ttf' if non_ascii else font,
                                           size=font_size or max(round(sum(self.im.size) / 2 * 0.035), 12))
            else:  # use cv2
```

```
            self.im = im
        self.lw = line_width or max(round(sum(im.shape) / 2 * 0.003), 2)  # line width

    def box_label(self, box, label='', color=(128, 128, 128), txt_color=(255, 255, 255)):
        # Add one xyxy box to image with label
        if self.pil or not is_ascii(label):
            self.draw.rectangle(box, width=self.lw, outline=color)  # box
            if label:
                w, h = self.font.getsize(label)  # text width, height
                outside = box[1] - h >= 0  # label fits outside box
                self.draw.rectangle(
                    (box[0], box[1] - h if outside else box[1], box[0] + w + 1,
                     box[1] + 1 if outside else box[1] + h + 1),
                    fill=color,
                )
                # self.draw.text((box[0], box[1]), label, fill=txt_color, font=self.font, anchor='ls')  # for PIL>8.0
                self.draw.text((box[0], box[1] - h if outside else box[1]), label, fill=txt_color, font=self.font)
        else:  # cv2
            p1, p2 = (int(box[0]), int(box[1])), (int(box[2]), int(box[3]))
            cv2.rectangle(self.im, p1, p2, color, thickness=self.lw, lineType=cv2.LINE_AA)
            if label:
                tf = max(self.lw - 1, 1)  # font thickness
                w, h = cv2.getTextSize(label, 0, fontScale=self.lw / 3, thickness=tf)[0]  # text width, height
                outside = p1[1] - h >= 3
                p2 = p1[0] + w, p1[1] - h - 3 if outside else p1[1] + h + 3
                cv2.rectangle(self.im, p1, p2, color, -1, cv2.LINE_AA)  # filled
                cv2.putText(self.im,
                            label, (p1[0], p1[1] - 2 if outside else p1[1] + h + 2),
                            0,
                            self.lw / 3,
                            txt_color,
                            thickness=tf,
                            lineType=cv2.LINE_AA)

    def masks(self, masks, colors, im_gpu=None, alpha=0.5):
        """Plot masks at once.
        Args:
            masks (tensor): predicted masks on cuda, shape: [n, h, w]
            colors (List[List[Int]]): colors for predicted masks, [[r, g, b] * n]
            im_gpu (tensor): img is in cuda, shape: [3, h, w], range: [0, 1]
            alpha (float): mask transparency: 0.0 fully transparent, 1.0 opaque
        """
        if self.pil:
            # convert to numpy first
            self.im = np.asarray(self.im).copy()
        if im_gpu is None:
            # Add multiple masks of shape(h,w,n) with colors list([r,g,b], [r,g,b], ...)
            if len(masks) == 0:
                return
            if isinstance(masks, torch.Tensor):
                masks = torch.as_tensor(masks, dtype=torch.uint8)
                masks = masks.permute(1, 2, 0).contiguous()
                masks = masks.cpu().numpy()
            # masks = np.ascontiguousarray(masks.transpose(1, 2, 0))
            masks = scale_image(masks.shape[:2], masks, self.im.shape)
            masks = np.asarray(masks, dtype=np.float32)
            colors = np.asarray(colors, dtype=np.float32)  # shape(n,3)
            s = masks.sum(2, keepdims=True).clip(0, 1)  # add all masks together
            masks = (masks @ colors).clip(0, 255)  # (h,w,n) @ (n,3) = (h,w,3)
            self.im[:] = masks * alpha + self.im * (1 - s * alpha)
        else:
            if len(masks) == 0:
                self.im[:] = im_gpu.permute(1, 2, 0).contiguous().cpu().numpy() * 255
            colors = torch.tensor(colors, device=im_gpu.device, dtype=torch.float32) / 255.0
            colors = colors[:, None, None]  # shape(n,1,1,3)
            masks = masks.unsqueeze(3)  # shape(n,h,w,1)
            masks_color = masks * (colors * alpha)  # shape(n,h,w,3)

            inv_alph_masks = (1 - masks * alpha).cumprod(0)  # shape(n,h,w,1)
            mcs = (masks_color * inv_alph_masks).sum(0) * 2  # mask color summand shape(n,h,w,3)

            im_gpu = im_gpu.flip(dims=[0])  # flip channel
            im_gpu = im_gpu.permute(1, 2, 0).contiguous()  # shape(h,w,3)
            im_gpu = im_gpu * inv_alph_masks[-1] + mcs
            im_mask = (im_gpu * 255).byte().cpu().numpy()
            self.im[:] = scale_image(im_gpu.shape, im_mask, self.im.shape)
        if self.pil:
```

```
            # convert im back to PIL and update draw
            self.fromarray(self.im)

    def rectangle(self, xy, fill=None, outline=None, width=1):
        # Add rectangle to image (PIL-only)
        self.draw.rectangle(xy, fill, outline, width)

    def text(self, xy, text, txt_color=(255, 255, 255), anchor='top'):
        # Add text to image (PIL-only)
        if anchor == 'bottom':  # start y from font bottom
            w, h = self.font.getsize(text)  # text width, height
            xy[1] += 1 - h
        self.draw.text(xy, text, fill=txt_color, font=self.font)

    def fromarray(self, im):
        # Update self.im from a numpy array
        self.im = im if isinstance(im, Image.Image) else Image.fromarray(im)
        self.draw = ImageDraw.Draw(self.im)

    def result(self):
        # Return annotated image as array
        return np.asarray(self.im)


def non_max_suppression(
        prediction,
        conf_thres=0.5,
        iou_thres=0.5,
        classes=None,
        agnostic=False,
        multi_label=False,
        labels=(),
        max_det=300,
        nm=0,  # number of masks
):
    """Non-Maximum Suppression (NMS) on inference results to reject overlapping detections

    Returns:
         list of detections, on (n,6) tensor per image [xyxy, conf, cls]
    """

    if isinstance(prediction, (list, tuple)):  # YOLO model in validation model, output = (inference_out, loss_out)
        prediction = prediction[0]  # select only inference output

    device = prediction.device
    mps = 'mps' in device.type  # Apple MPS
    if mps:  # MPS not fully supported yet, convert tensors to CPU before NMS
        prediction = prediction.cpu()
    bs = prediction.shape[0]  # batch size
    nc = prediction.shape[1] - nm - 4  # number of classes
    mi = 4 + nc  # mask start index
    xc = prediction[:, 4:mi].amax(1) > conf_thres  # candidates

    # Checks
    assert 0 <= conf_thres <= 1, f'Invalid Confidence threshold {conf_thres}, valid values are between 0.0 and 1.0'
    assert 0 <= iou_thres <= 1, f'Invalid IoU {iou_thres}, valid values are between 0.0 and 1.0'

    # Settings
    # min_wh = 2  # (pixels) minimum box width and height
    max_wh = 7680  # (pixels) maximum box width and height
    max_nms = 30000  # maximum number of boxes into torchvision.ops.nms()
    time_limit = 2.5 + 0.05 * bs  # seconds to quit after
    redundant = True  # require redundant detections
    multi_label &= nc > 1  # multiple labels per box (adds 0.5ms/img)
    merge = True  # use merge-NMS

    t = time.time()
    output = [torch.zeros((0, 6 + nm), device=prediction.device)] * bs
    for xi, x in enumerate(prediction):  # image index, image inference
        # Apply constraints
        # x[((x[:, 2:4] < min_wh) | (x[:, 2:4] > max_wh)).any(1), 4] = 0  # width-height
        x = x.T[xc[xi]]  # confidence

        # Cat apriori labels if autolabelling
        if labels and len(labels[xi]):
            lb = labels[xi]
            v = torch.zeros((len(lb), nc + nm + 5), device=x.device)
```

```
                v[:, :4] = lb[:, 1:5]  # box
                v[range(len(lb)), lb[:, 0].long() + 4] = 1.0  # cls
                x = torch.cat((x, v), 0)

            # If none remain process next image
            if not x.shape[0]:
                continue

            # Detections matrix nx6 (xyxy, conf, cls)
            box, cls, mask = x.split((4, nc, nm), 1)
            box = xywh2xyxy(box)  # center_x, center_y, width, height) to (x1, y1, x2, y2)
            if multi_label:
                i, j = (cls > conf_thres).nonzero(as_tuple=False).T
                x = torch.cat((box[i], x[i, 4 + j, None], j[:, None].float(), mask[i]), 1)
            else:  # best class only
                conf, j = cls.max(1, keepdim=True)
                x = torch.cat((box, conf, j.float(), mask), 1)[conf.view(-1) > conf_thres]

            # Filter by class
            if classes is not None:
                x = x[(x[:, 5:6] == torch.tensor(classes, device=x.device)).any(1)]

            # Apply finite constraint
            # if not torch.isfinite(x).all():
            #     x = x[torch.isfinite(x).all(1)]

            # Check shape
            n = x.shape[0]  # number of boxes
            if not n:  # no boxes
                continue
            elif n > max_nms:  # excess boxes
                x = x[x[:, 4].argsort(descending=True)[:max_nms]]  # sort by confidence
            else:
                x = x[x[:, 4].argsort(descending=True)]  # sort by confidence

            # Batched NMS
            c = x[:, 5:6] * (0 if agnostic else max_wh)  # classes
            boxes, scores = x[:, :4] + c, x[:, 4]  # boxes (offset by class), scores
            i = torchvision.ops.nms(boxes, scores, iou_thres)  # NMS
            if i.shape[0] > max_det:  # limit detections
                i = i[:max_det]
            if merge and (1 < n < 3E3):  # Merge NMS (boxes merged using weighted mean)
                # update boxes as boxes(i,4) = weights(i,n) * boxes(n,4)
                iou = box_iou(boxes[i], boxes) > iou_thres  # iou matrix
                weights = iou * scores[None]  # box weights
                x[i, :4] = torch.mm(weights, x[:, :4]).float() / weights.sum(1, keepdim=True)  # merged boxes
                if redundant:
                    i = i[iou.sum(1) > 1]  # require redundancy

            output[xi] = x[i]
            if mps:
                output[xi] = output[xi].to(device)
            if (time.time() - t) > time_limit:
                break  # time limit exceeded

    return output


def letterbox(im, new_shape=(640, 640), color=(114, 114, 114), auto=True, scaleFill=False, scaleup=True, stride=32):
    # Resize and pad image while meeting stride-multiple constraints
    shape = im.shape[:2]  # current shape [height, width]
    if isinstance(new_shape, int):
        new_shape = (new_shape, new_shape)

    # Scale ratio (new / old)
    r = min(new_shape[0] / shape[0], new_shape[1] / shape[1])
    if not scaleup:  # only scale down, do not scale up (for better val mAP)
        r = min(r, 1.0)

    # Compute padding
    ratio = r, r  # width, height ratios
    new_unpad = int(round(shape[1] * r)), int(round(shape[0] * r))
    dw, dh = new_shape[1] - new_unpad[0], new_shape[0] - new_unpad[1]  # wh padding
    if auto:  # minimum rectangle
        dw, dh = np.mod(dw, stride), np.mod(dh, stride)  # wh padding
    elif scaleFill:  # stretch
        dw, dh = 0.0, 0.0
```

```
            new_unpad = (new_shape[1], new_shape[0])
            ratio = new_shape[1] / shape[1], new_shape[0] / shape[0]  # width, height ratios

        if shape[::-1] != new_unpad:  # resize
            im = cv2.resize(im, new_unpad, interpolation=cv2.INTER_LINEAR)

        # Convert to scalar values for padding calculations
        if isinstance(dw, torch.Tensor):
            dw = dw.item()
        if isinstance(dh, torch.Tensor):
            dh = dh.item()

        top, bottom = int(round(dh / 2 - 0.1)), int(round(dh / 2 + 0.1))
        left, right = int(round(dw / 2 - 0.1)), int(round(dw / 2 + 0.1))
        im = cv2.copyMakeBorder(im, top, bottom, left, right, cv2.BORDER_CONSTANT, value=color)  # add border
        return im, ratio, (dw, dh)



    class LoadImages:
        def __init__(self, path, img_size=640, stride=32, auto=True, transforms=None, vid_stride=1):
            files = []
            for p in sorted(path) if isinstance(path, (list, tuple)) else [path]:
                p = str(Path(p).resolve())
                if '*' in p:
                    files.extend(sorted(glob.glob(p, recursive=True)))  # glob
                elif os.path.isdir(p):
                    files.extend(sorted(glob.glob(os.path.join(p, '*.*'))))  # dir
                elif os.path.isfile(p):
                    files.append(p)  # files
                else:
                    raise FileNotFoundError(f'{p} does not exist')

            images = [x for x in files if x.split('.')[-1].lower() in IMG_FORMATS]

            ni = len(images)

            self.img_size = img_size
            self.stride = stride
            self.files = images
            self.nf = ni + nv  # number of files

            self.mode = 'image'
            self.auto = auto
            self.transforms = transforms  # optional


            self.cap = None


        def __iter__(self):
            self.count = 0
            return self

        def __next__(self):
            if self.count == self.nf:
                raise StopIteration
            path = self.files[self.count]



            # Read image
            self.count += 1
            im0 = cv2.imread(path)  # BGR
            assert im0 is not None, f'Image Not Found {path}'
            s = f'image {self.count}/{self.nf} {path}: '

            if self.transforms:
                im = self.transforms(im0)  # transforms
            else:
                im = letterbox(im0, self.img_size, stride=self.stride, auto=self.auto)[0]  # padded resize
                im = im.transpose((2, 0, 1))[::-1]  # HWC to CHW, BGR to RGB
                im = np.ascontiguousarray(im)  # contiguous

            return path, im, im0, self.cap, s

        def _cv2_rotate(self, im):
```

```
            # Rotate a cv2 image manually
            if self.orientation == 0:
                return cv2.rotate(im, cv2.ROTATE_90_CLOCKWISE)
            elif self.orientation == 180:
                return cv2.rotate(im, cv2.ROTATE_90_COUNTERCLOCKWISE)
            elif self.orientation == 90:
                return cv2.rotate(im, cv2.ROTATE_180)
            return im

        def __len__(self):
            return self.nf  # number of files


    names = [
            'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus', 'train', 'truck', 'boat', 'traffic light',
            'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
            'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee',
            'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard',
            'tennis racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
            'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake', 'chair', 'couch',
            'potted plant', 'bed', 'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone',
            'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear',
            'hair drier', 'toothbrush'
        ]  # class names


    from google.colab import drive
    drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Object Detection - Displays detected images with bounding boxes and prints class names with confidence scores for each detection.

```
    import torch
    import cv2
    import numpy as np
    import os
    from pathlib import Path
    from google.colab.patches import cv2_imshow
    import torchvision

    def process_and_display_images(input_folder, model, names, conf_thres=0.25, iou_thres=0.45):

        # Get all image files
        image_files = [f for f in os.listdir(input_folder)
                       if f.split('.')[-1].lower() in ('jpg', 'jpeg', 'png', 'bmp')]

        for file_name in image_files:
            img_path = os.path.join(input_folder, file_name)

            img0 = cv2.imread(img_path)
            img = cv2.resize(img0, (640, 640))

            img = img[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, HWC to CHW
            img = np.ascontiguousarray(img)

            # Original image for display
            display_img = img0.copy()

            # Preprocess
            img = cv2.resize(img0, (640, 640))
            img = img[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, HWC to CHW
            img = np.ascontiguousarray(img)
            img_tensor = torch.from_numpy(img).float().div(255.0).unsqueeze(0)  # (1, 3, 640, 640)



            # Inference
            with torch.no_grad():
                output = model(img_tensor)
                pred = output[0]

            # Post-process
```

```
            pred = non_max_suppression(pred, conf_thres, iou_thres)[0]

            annotator = Annotator(img0.copy(), line_width=3, example=str(names))

            if pred is not None and len(pred):
                pred[:, :4] = scale_boxes(img_tensor.shape[2:], pred[:, :4], img0.shape).round()
                for *xyxy, conf, cls in pred:
                    name = names[int(cls)]
                    label = f'{name} {conf:.2f}'

                    # For image annotation (unchanged)
                    annotator.box_label(xyxy, label, color=colors(int(cls), bgr=True))

                    # Add colorized terminal log
                    colored_label = colorstr('bright_cyan', 'bold', name)
                    print(f"Detected: {colored_label} ({conf:.2f}) in {file_name}")


            print(f"\nProcessing: {file_name}")
            cv2_imshow(annotator.result())
            cv2.waitKey(1)

    # Configuration
    input_folder = '/content/drive/MyDrive/TestImages_Yolo9'


    # Run processing
    process_and_display_images(
        input_folder=input_folder,
        model=model,
        names=names,
        conf_thres=0.25,
        iou_thres=0.45
    )

    print("\nObject Detection Complete for all images.")
```

```
Detected: horse (0.73) in fabian-burghardt-A81818EFqGQ-unsplash.jpg
Detected: horse (0.72) in fabian-burghardt-A81818EFqGQ-unsplash.jpg
Detected: horse (0.66) in fabian-burghardt-A81818EFqGQ-unsplash.jpg

Processing: fabian-burghardt-A81818EFqGQ-unsplash.jpg
```

```
Detected: horse (0.76) in laura-roberts-1hLEDpjd2v0-unsplash.jpg
Detected: dog (0.72) in laura-roberts-1hLEDpjd2v0-unsplash.jpg

Processing: laura-roberts-1hLEDpjd2v0-unsplash.jpg
```