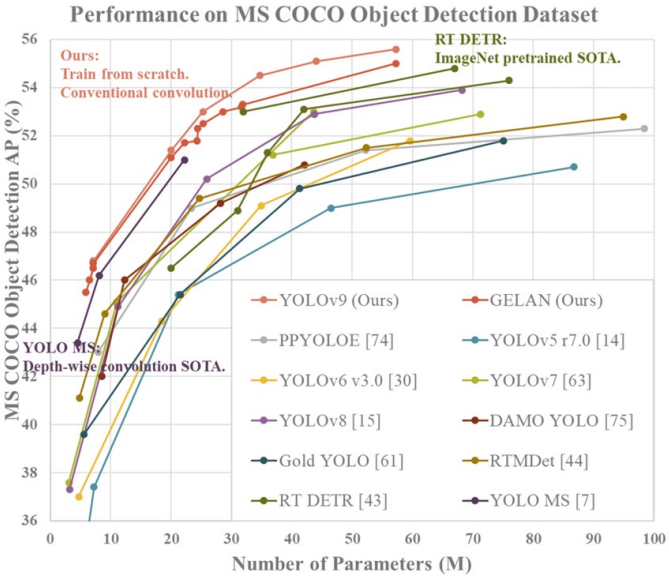
YOLO v9: Learning What You Want to Learn Using **Programmable Gradient Information**
和 v4、v7 相同作者，发表于 2024.02

【性能表现】



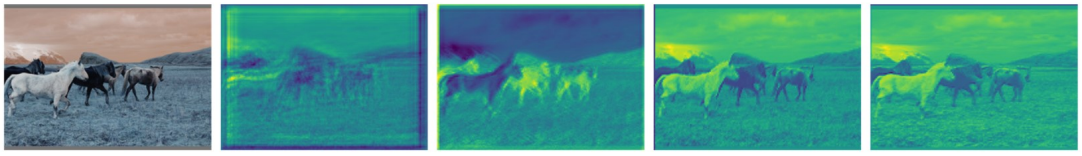
YOLOv9/GELAN 模型（**红线**）位于最左上方，表示在参数数量和评价准确率上都优于其他模型。
此外：①超越了 RT DETR，说明 YOLOv9/GELAN 模型不需要大量预处理资料，就能达到很好的效果；②超越了 YOLO MS，说明比起大量使用深度可分离卷积的模型，YOLOv9/GELAN 模型的参数量更少，模型效率更高。

【背景】
过去的研究常常忽略了一个问题：信息在前向传递的过程中会出现损失，进而导致梯度流在反向传播时出现偏差。（即：信息损失，梯度偏差）

information bottleneck principle: data X may cause information loss when going through transformation.

$$I(X, X) \geq I(X, f_\theta(X)) \geq I(X, g_\phi(f_\theta(X))), \quad (1)$$

where I indicates mutual information, f and g are transformation functions, and θ and ϕ are parameters of f and g, respectively.



(a) Input Image (b) PlainNet (c) ResNet (d) CSPNet (e) GELAN

Figure 2. Visualization results of random initial weight output feature maps for different network architectures: (a) input image, (b) PlainNet, (c) ResNet, (d) CSPNet, and (e) proposed GELAN. From the figure, we can see that in different architectures, the information provided to the objective function to calculate the loss is lost to varying degrees, and our architecture can retain the most complete information and provide the most reliable gradient information for calculating the objective function.

信息损失如何引起梯度偏差？

举例，假设有一个网络层 x，经过 ReLU 函数后得到下一网络层 y

Forward:

x = [-2.0, -1.0, 3.0]

y = ReLU(x) = [0.0, 0.0, 3.0]

Backward:

∂L/∂y = [0.5, 0.5, 0.5]

ReLU'(x) = [0, 0, 1]

∂L/∂x = ∂L/∂y × ReLU'(x) = [0.0, 0.0, 0.5]

在反向传播过程中，对于 x[0] = -2.0 和 x[1] = -1.0，梯度完全为 0，这些输入维度不会被更新。

前向传播时**信息丢失** → 反向传播时无法恢复这些信息 → 梯度为 0 → 模型在这些维度上无法学习 → **梯度偏差**

Q1：这个例子是 relu 函数，那假设使用的是 leaky relu？如何解释信息损失？

【主要内容】

基于 YOLO v7，提出了

(1) **PGI**（Programmable Gradient Information）可编程梯度信息，

(2) **GELAN**（Generalized Efficient Layer Aggregation Network）广义高效层聚合网络

【PGI】

Programmable Gradient Information 可编程梯度信息

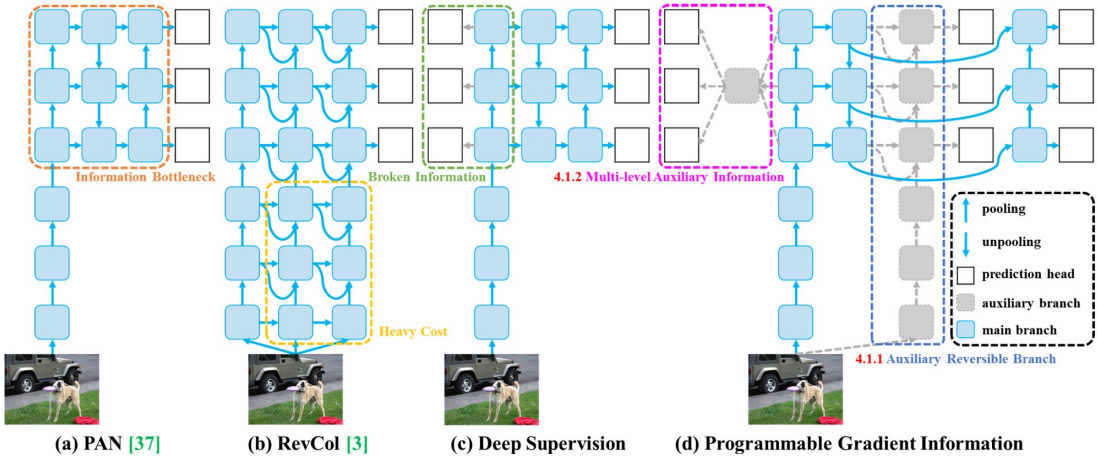
PGI 是一种**设计思想**

PGI mainly includes three components,

(1) main branch 主分支

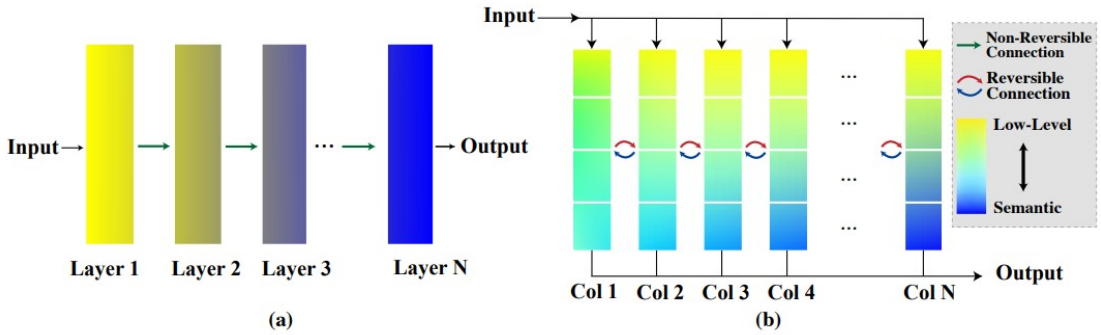
(2) auxiliary reversible branch 辅助可逆分支

(3) multi-level auxiliary information 多级辅助信息



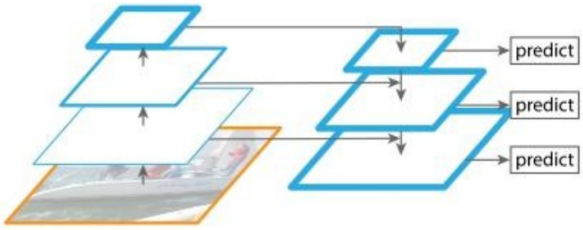
(b) 可逆架构中添加主分支会消耗大量的推理成本，尤其是在浅层高分辨率计算阶段。

并且可逆结构在浅层网络中表现不佳，因为它限制了信息转换的自由度（Q2：为什么可逆运算对特征的表达能力存在一定约束？Q3：为什么不可逆运算会丢失信息？）。



-> 提出 **auxiliary reversible branch 辅助可逆分支**，通过生成有用梯度来更新主分支，使得浅层网络也能高效学习复杂任务，从而提升整体性能。

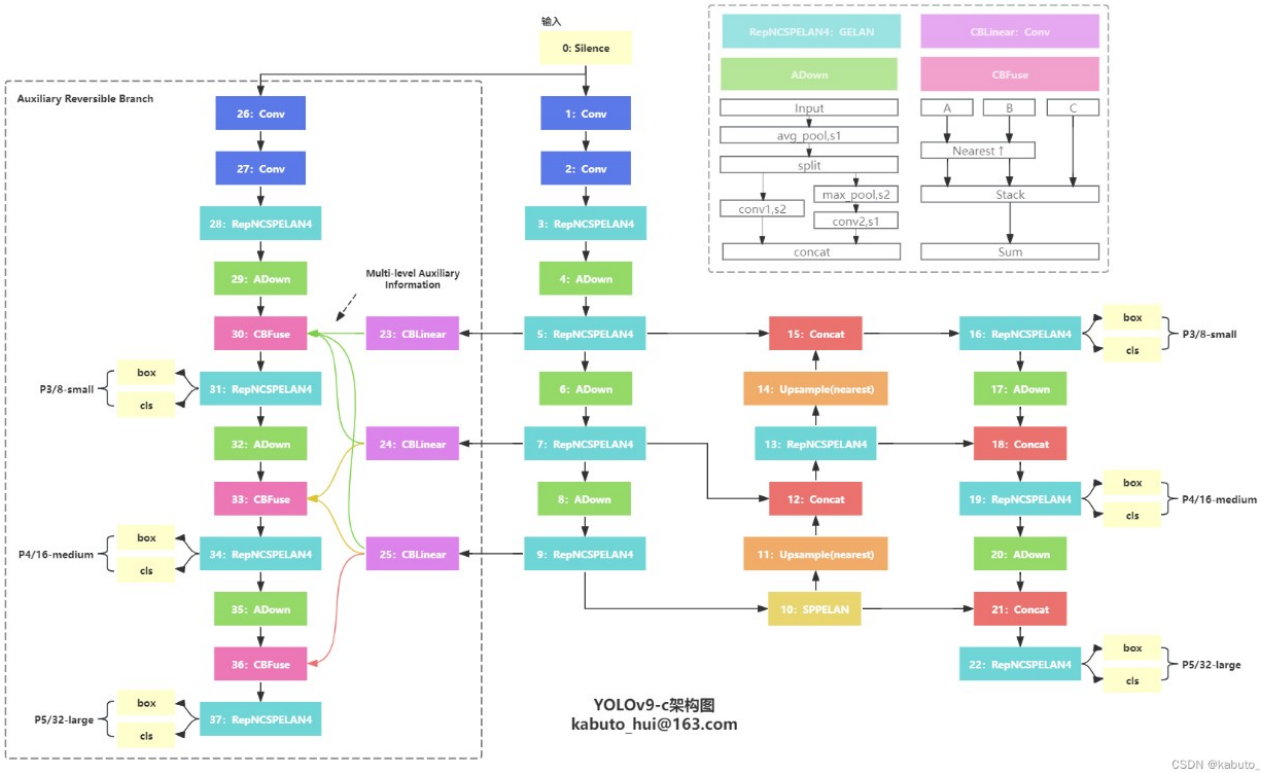
(c) 特征金字塔用于处理不同尺度的目标。浅层负责小目标，深层负责大目标。如果在浅层加一个 aux head 辅助头，浅层网络会被 deep supervision 机制引导去专注于小目标，可能忽略其他尺寸的目标，把它们当作背景。由于浅层信息不完整，深层金字塔在融合时也会受到影响，导致整体信息丢失。



(d) Feature Pyramid Network

作者认为每一层金字塔都应该接收到所有目标的梯度信息，而不是只关注某一类。-> 提出 **multi-level auxiliary information 多级辅助信息**

(1) PGI 的推理过程仅使用了主分支，因此不需要额外的推理成本
(2) 辅助可逆分支是为了处理神经网络加深带来的问题，网络加深会造成信息瓶颈，导致损失函数无法生成可靠的梯度
(3) 多级辅助信息旨在处理深度监督带来的误差累积问题，特别是多个预测分支的架构和轻量级模型



Q4：辅助可逆分支在推理阶段如何移去的？

Q5: PGI 机制的“可逆”体现在哪里？

并非严格的数学可逆，不同于 RevNet 等模型的逐层数学可逆，而是一种面向解决信息丢失和梯度质量问题的**功能性可逆设计**。能够“还原”浅层信息的原始特征，产生纯净可靠的梯度来优化主干网络，从而**在效果上“逆转”了**深度网络固有的信息损耗和梯度偏差问题

Q：为什么叫“可编程”的？

辅助分支可以自己设计

【网络结构】

1. GELAN

Generalized Efficient Layer Aggregation Network 广义高效层聚合网络

是一个具体的网络模块

与 **ELAN** in YOLO v7 的区别：

stacking of convolutional layers -> **any computational blocks**

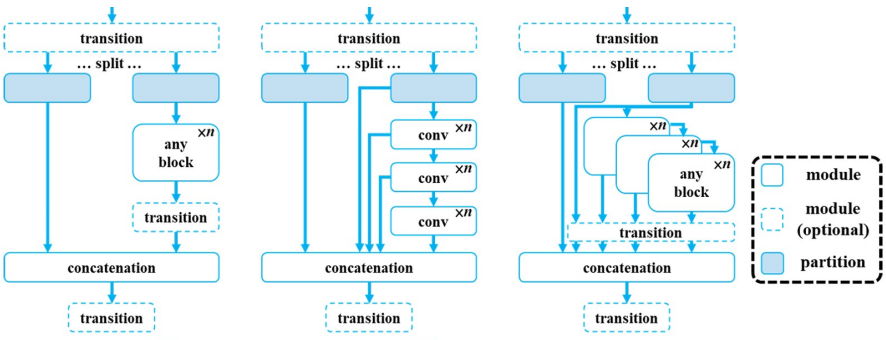


Figure 4. The architecture of GELAN: (a) CSPNet [64], (b) ELAN [65], and (c) proposed GELAN. We imitate CSPNet and extend ELAN into GELAN that can support any computational blocks.

优点：具有很高的**灵活性和适应性**，可以根据具体的任务需求和硬件环境来选择最适合的计算单元。如果任务需要大量的特征提取，可以选择使用更多的卷积层；如果硬件环境有限，可以选择使用更轻量级的计算单元，如深度可分离卷积。

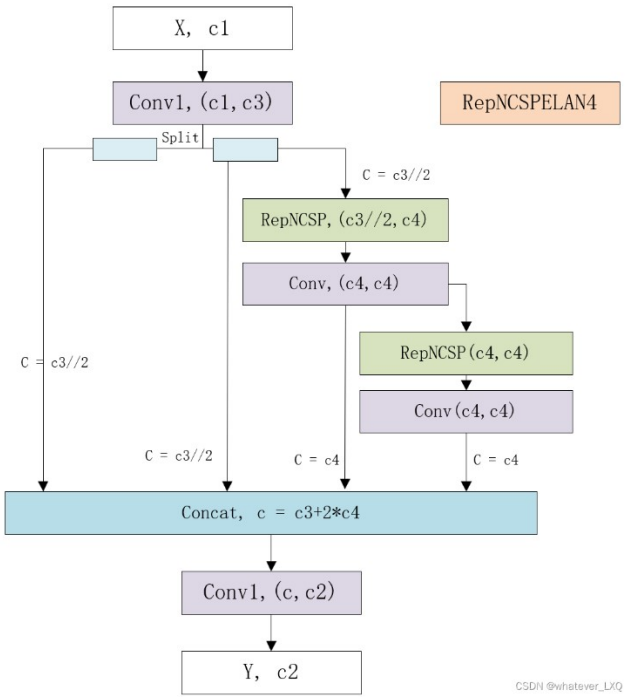
placements, CSP blocks perform particularly well. They not only reduce the amount of parameters and computation, but also improve AP by 0.7%. Therefore, we choose CSP-ELAN as the component unit of GELAN in YOLOv9.

Table 2. Ablation study on various computational blocks.

Model	CB type	#Param.	FLOPs	$AP^{val}_{50:95}$
GELAN-S	Conv	6.2M	23.5G	44.8%
GELAN-S	Res [21]	5.4M	21.0G	44.3%
GELAN-S	Dark [49]	5.7M	21.8G	44.5%
GELAN-S	CSP [64]	5.9M	22.4G	45.5%

Choose **CSPNet + ELAN**

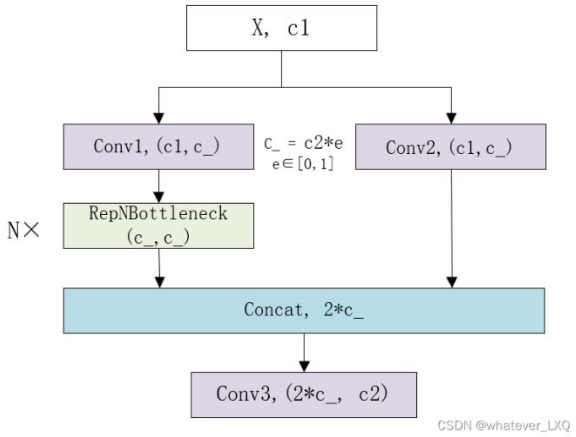
YOLO v9 里的 GELAN: **RepNCSPELAN4 模块**



 (Q: Why this name?)

```python
class RepNCSPELAN4(nn.Module):
    # csp-elan模块
    def __init__(self, c1, c2, c3, c4, c5=1):  # 输入通道数, 输出通道数, 中间通道数1, 中间通道数2, 块重复次数
        super().__init__()
        self.c = c3 // 2
        self.cv1 = Conv(c1, c3, 1, 1)  # 使用1x1卷积将输入通道数变为c3
        self.cv2 = nn.Sequential(RepNCSP(c3 // 2, c4, c5), Conv(c4, c4, 3, 1))  # 定义包含RepNCSP和3x3卷积的顺序容器
        self.cv3 = nn.Sequential(RepNCSP(c4, c4, c5), Conv(c4, c4, 3, 1))  # 定义另一个包含RepNCSP和3x3卷积的顺序容器
        self.cv4 = Conv(c3 + (2 * c4), c2, 1, 1)  # 使用1x1卷积将c3+2倍c4通道数变为c2

    def forward(self, x):
        y = list(self.cv1(x).chunk(2, 1))  # 先经过cv1卷积，然后在通道维度上分成两部分
        y.extend((m(y[-1])) for m in [self.cv2, self.cv3])  # 将第一部分依次经过cv2和cv3
        return self.cv4(torch.cat(y, 1))  # 将所有输出在通道维度上拼接后经过cv4卷积

    def forward_split(self, x):
        y = list(self.cv1(x).split((self.c, self.c), 1))  # 先经过cv1卷积，然后在通道维度上按指定通道数分成两部分
        y.extend(m(y[-1]) for m in [self.cv2, self.cv3])  # 将第一部分依次经过cv2和cv3
        return self.cv4(torch.cat(y, 1))  # 将所有输出在通道维度上拼接后经过cv4卷积
```

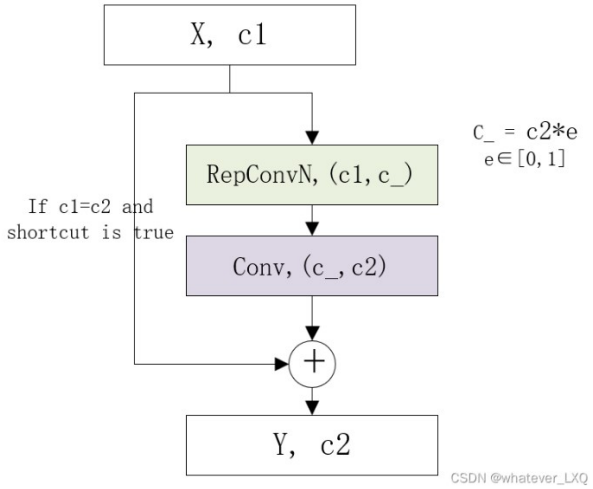
RepNCSPELAN4 的子模块 **RepNCSP**



CSDN @whatever_LXQ

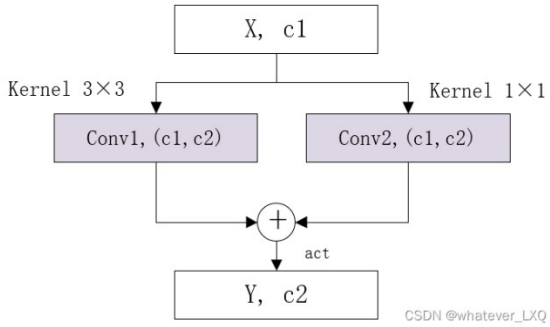
```python
class RepNCSP(nn.Module):
    # CSP Bottleneck with 3 convolutions
    def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):  # ch_in, ch_out, number, shortcut, groups, expansion
        super().__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = Conv(c1, c_, 1, 1)
        self.cv3 = Conv(2 * c_, c2, 1)  # optional act=FReLU(c2)
        self.m = nn.Sequential(*(RepNBottleneck(c_, c_, shortcut, g, e=1.0) for _ in range(n)))

    def forward(self, x):
        return self.cv3(torch.cat((self.m(self.cv1(x)), self.cv2(x)), 1))
```

RepNCSP 的子模块 **RepNBottleneck**




```python
class RepNBottleneck(nn.Module):
    # Standard bottleneck
    def __init__(self, c1, c2, shortcut=True, g=1, k=(3, 3), e=0.5):  # ch_in, ch_out, shortcut, kernels, groups, expand
        super().__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = RepConvN(c1, c_, k[0], 1)
        self.cv2 = Conv(c_, c2, k[1], 1, g=g)
        self.add = shortcut and c1 == c2

    def forward(self, x):
        return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1(x))
```

RepNBottleneck 的子模块 **RepConvN**



In paper:

“The GELAN only uses conventional convolution to achieve a higher parameter usage than the **depth-wise convolution** design that based on the most advanced technology, while showing great advantages of **being light, fast, and accurate**. （与深度可分离卷积相比）

The design of GELAN simultaneously takes into account the number of parameters, computational complexity, accuracy and inference speed. ”（理论上，从整体架构设计角度出发）

优点：

将 ELAN 中的卷积组，替换成带有 RepConv 的 CSPNet，相当于增加网络的宽度（增加了并行路径）。这种增加网络宽度的方式可以增加梯度流（因为有更多路径参与反向传播），理论上可以提升训练稳定性和特征表达能力；

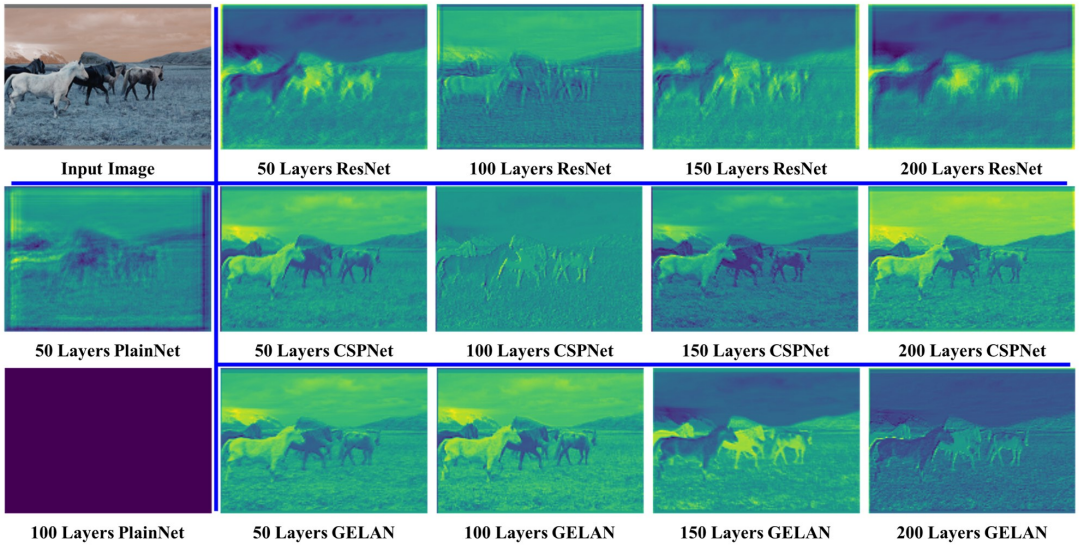


Figure 6. Feature maps (visualization results) output by random initial weights of PlainNet, ResNet, CSPNet, and GELAN at different depths. After 100 layers, ResNet begins to produce feedforward output that is enough to obfuscate object information. Our proposed GELAN can still retain quite complete information up to the 150^{th} layer, and is still sufficiently discriminative up to the 200^{th} layer.

缺点：

损失了卷积的连贯性，这对于板端部署（Edge Deployment）来说是不友好的，会增加耗时。比如，用 **split** 的方式看似丰富了所谓的梯度流，减少了参数量，但增加了推理的各种不连续性，大大增加了推理的时间。

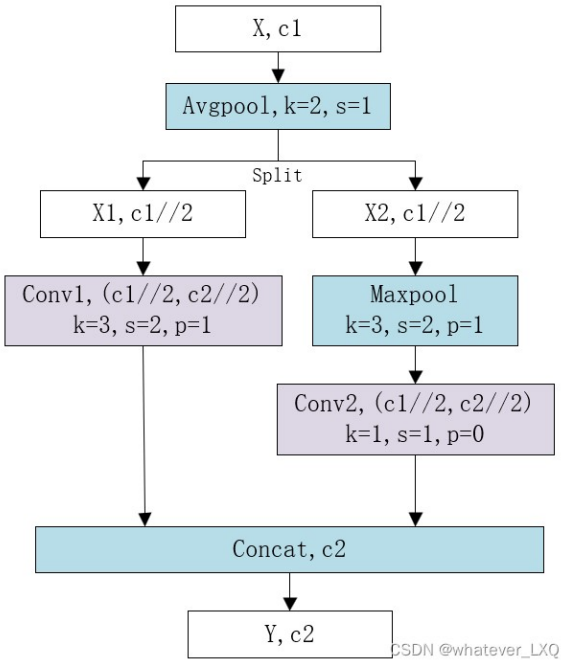
2. 下采样：ADown 模块

不同的下采样方法：

(1) 传统池化（如 Max Pooling）：信息丢失，对小目标不友好。

(2) 跨步卷积（Strided Convolution）：计算效率高，但可能忽略局部细节。

(3) ADown：在信息保留与计算效率之间取得平衡。



Models which use **Adown** to down-sampling: **YOLOv9-C, YOLOv9-E, yolov9-CF**

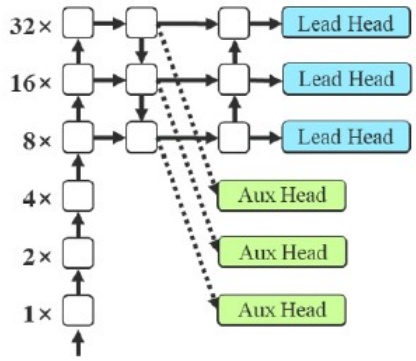
```python
class AConv(nn.Module):
    def __init__(self, c1, c2):  # ch_in, ch_out, shortcut, kernels, groups, expand
        super().__init__()
        self.cv1 = Conv(c1, c2, 3, 2, 1)

    def forward(self, x):
        x = torch.nn.functional.avg_pool2d(x, 2, 1, 0, False, True)
        return self.cv1(x)
```

Models which use **AConv** to down-sampling: **YOLOv9-T, YOLOv9-S, YOLOv9-M**

Model	Test Size	AP^{val}	AP_{50}^{val}	AP_{75}^{val}	Param.	FLOPs
YOLOv9-T	640	38.3%	53.1%	41.3%	2.0M	7.7G
YOLOv9-S	640	46.8%	63.4%	50.7%	7.1M	26.4G
YOLOv9-M	640	51.4%	68.1%	56.1%	20.0M	76.3G
YOLOv9-C	640	53.0%	70.2%	57.8%	25.3M	102.1G
YOLOv9-E	640	55.6%	72.8%	60.6%	57.3M	189.0G

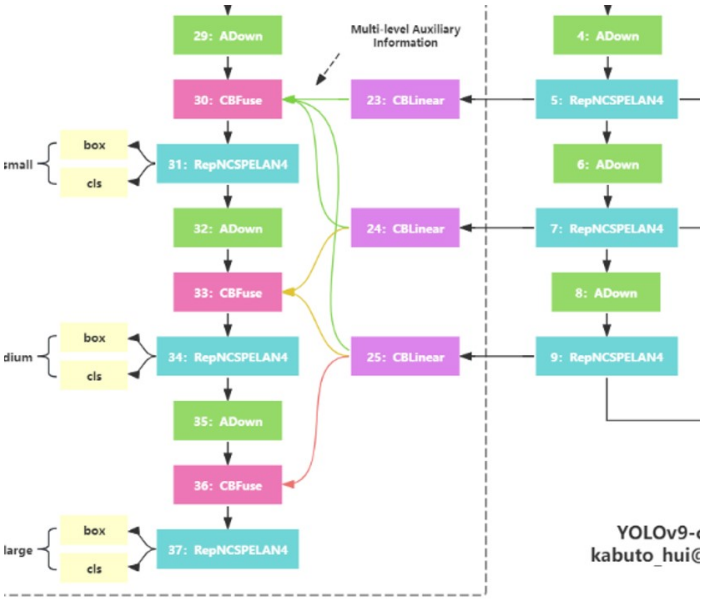
可以看出，规模较大的模型（C, E, CF）使用 Adown 模块；规模较小的模型（T, S, M）使用 AConv 模块。
相比传统下采样方法（如最大池化），ADown 能更好地保留边缘、纹理等细节信息，减少空间信息的丢失，从而提升小目标检测的精度。在降低特征图分辨率的同时，显著减少计算量和参数量，适合资源受限的设备（如移动端或嵌入式设备）。

Q: YOLO v9 与 YOLO v7 都使用了 Deep Supervision，区别是?
1. YOLO v7 在计算 auxiliary head 时，所用的特征图是独立的



(b) Model with auxiliary head

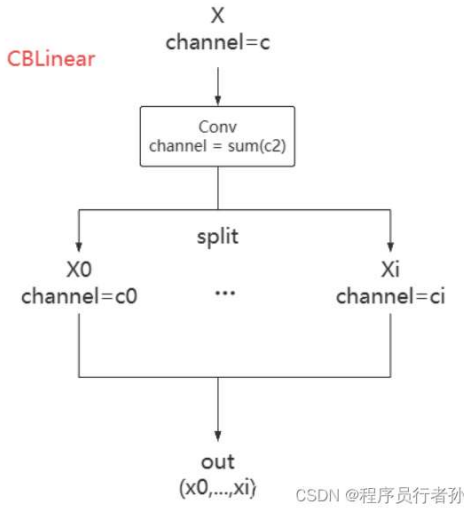
而 YOLO v9 是融合了多个特征图后再计算。即 **multi-level auxiliary information**




```
# routing
[5, 1, CBLinear, [[240]]], # 23
[7, 1, CBLinear, [[240, 360]]], # 24
[9, 1, CBLinear, [[240, 360, 480]]], # 25

# avg-conv down
[-1, 1, AConv, [240]],   # 29-P3/8
[[23, 24, 25, -1], 1, CBFuse, [[0, 0, 0]]], # 30

# elan-2 block
[-1, 1, RepNCSPELAN4, [240, 240, 120, 1]],  # 31

# avg-conv down
[-1, 1, AConv, [360]],   # 32-P4/16
[[24, 25, -1], 1, CBFuse, [[1, 1]]], # 33

# elan-2 block
[-1, 1, RepNCSPELAN4, [360, 360, 180, 1]],  # 34

# avg-conv down
[-1, 1, AConv, [480]],   # 35-P5/32
[[25, -1], 1, CBFuse, [[2]]], # 36
```

YOLOv9-
kabuto_hui@

3. CB Linear




```python
class CBLinear(nn.Module):  # 定义CBLinear类，继承自PyTorch的nn.Module
    def __init__(self, c1, c2s, k=1, s=1, p=None, g=1):  # 构造函数
        super(CBLinear, self).__init__()  # 调用基类的构造函数
        self.c2s = c2s  # 存储输出通道数的列表
        # 初始化卷积层，c1是输入通道数，sum(c2s)是所有输出通道的总和
        # k是卷积核大小，s是步长，p是填充，g是组数
        self.conv = nn.Conv2d(
            c1, sum(c2s), k, s, autopad(k, p), groups=g, bias=True
        )

    def forward(self, x):
        # 前向传播函数
        outs = self.conv(x)  # 将输入x通过卷积层
        # 使用torch.split按通道维度分割outs，根据c2s中定义的通道数进行分割
        outs = torch.split(outs, self.c2s, dim=1)
        return outs  # 返回分割后的输出列表
```

4. CB Fuse

```python
class CBFuse(nn.Module):
    def __init__(self, idx):
        super(CBFuse, self).__init__()
        self.idx = idx

    def forward(self, xs):
        target_size = xs[-1].shape[2:]
        res = [F.interpolate(x[self.idx[i]], size=target_size, mode='nearest') for i, x in enumerate(xs[:-1])]
        out = torch.sum(torch.stack(res + xs[-1:]), dim=0)
        return out
```

1. 选取特征图分支：从每个输入 x[i] 中选取指定的分支 x[i][idx[i]]
2. 上采样对齐尺寸：将这些特征图上采样到与最后一个特征图 xs[-1] 相同的空间尺寸
3. 逐元素相加融合：将所有对齐后的特征图逐元素相加，得到融合后的输出

【Head】

YOLO v7: Still anchor-based. Coupled head. For Lead Head and Aux Head, output feature maps have **255** channels (Each grid has 3 anchor boxes, and each box has 4 coordinates, 1 objectness score, and 80 classification scores. Total = 3 * (4 + 1 + 80) = 255).

YOLO v8: Anchor-free. Decoupled head. Class feature maps have **80** channels (80 classes). BBox feature maps have **64** channels (4 offsets: left, right, top, down. If reg_max = 16, for each offset, we will get a probability distribution of 16 values. Total = 4 * 16 = 64).

YOLO v9: Anchor-free. Decoupled head… **Almost same as YOLO v8.** The difference is YOLO v9 has **aux heads**.

```python
    def __init__(self, nc=80, ch=(), inplace=True):  # detection layer
        super().__init__()
        self.nc = nc  # number of classes
        self.nl = len(ch) // 2  # 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
```

```python
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)
```

See **Class DualDetect** in yolov9/models/yolo.py
https://github.com/WongKinYiu/yolov9/blob/main/models/yolo.py

【Label Assignment & Loss Function】

Task Ailgn Assigner, same as YOLO v8 and YOLO v6

What is Task Ailgn Assigner?

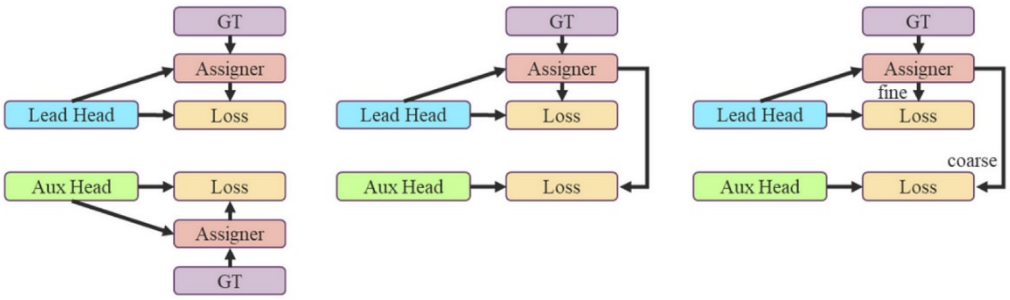
参考 YOLO v8 和 YOLO v6 的笔记

Step1：对于每个 anchor point 和每个 GT 框，计算对齐分数（Align Metric）。对齐分数的计算方法：基于 GT 类别对应的**分类分数**，以及预测框与 GT 的 **IoU**，加权得出
Step2：筛选候选正样本。只考虑那些落在 GT 框中心区域且对齐分数最高的 Top-K 个点
Step3：处理多 GT 冲突
Step4：生成训练标签

```
self.assigner = TaskAlignedAssigner(topk=int(os.getenv('YOLOM', 10)),
                                    num_classes=self.nc,
                                    alpha=float(os.getenv('YOLOA', 0.5)),
                                    beta=float(os.getenv('YOLOB', 6.0)))
self.assigner2 = TaskAlignedAssigner(topk=int(os.getenv('YOLOM', 10)),
                                    num_classes=self.nc,
                                    alpha=float(os.getenv('YOLOA', 0.5)),
                                    beta=float(os.getenv('YOLOB', 6.0)))
self.bbox_loss = BboxLoss(m.reg_max - 1, use_dfl=use_dfl).to(device)
self.bbox_loss2 = BboxLoss(m.reg_max - 1, use_dfl=use_dfl).to(device)
self.proj = torch.arange(m.reg_max).float().to(device)  # / 120.0
self.use_dfl = use_dfl
```

* See Class Compute in yolov9/utils/loss_tal_dual.py
https://github.com/WongKinYiu/yolov9/blob/main/utils/loss_tal_dual.py

Q: 和 YOLO v7 的区别?



(c) Independent assigner (d) Lead guided assigner (e) Coarse-to-fine lead guided assigner

YOLO v7 uses **Coarse-to-fine lead head guided label assigner + SimOTA**

這部分同樣也是採用 lead head 的預測結果與 ground truth 進行最佳化運算所得到 soft label，差別在於會產生兩組不同的 **soft label: coarse label、fine label**，其中 fine label 與 lead head 的 soft label 相同、coarse label 用於 auxiliary head。↵

通过查看源码，发现 **aux head** 的 **assigner** 和 **lead head** 的 **assigner** 仅存在很少的不同，包括

①**lead head** 中每个 **anchor** 与 **GT** 如果匹配上，分配 **3** 个正样本，而 **aux head** 分配 **5** 个。↵

②**lead head** 中将 **top10** 个样本 **IoU** 求和取整，而 **aux head** 中取 **top20**。 **(top-*q*)**↵

而 YOLO v9 对于 lead head 和 aux head，独立使用 **Task-Aligned Assigner** 进行标签分配

事实上，作者在论文中提到： We also tried to apply the **lead head guided assignment** proposed in YOLOv7[63] to the PGI's auxiliary supervision, and achieved **much better** performance.

Similar to Attention multi head?

Q: 效果更好，最终却没有采用，为什么？

可能是因为 PGI 的辅助分支是能够学到大量资讯的。更希望辅助分支有自己的标签视角，以增强梯度多样性。标签共享虽然高效，但可能限制了 PGI 的独立学习能力。

Loss Function:

YOLO v7: **CIoU + BCE**

Loss function: Regression loss (CIoU loss) + objectness loss (BCE loss) + classification loss (BCE loss)

A weighting of 0.25 is multiplied to the loss calculated from aux head, and added to the loss calculated from lead head.

```
# Lead head
lbox += (1.0 - iou).mean()
lcls += self.BCEcls(ps[:, 5:], t)

# Aux head
lbox += 0.25 * (1.0 - iou_aux).mean()
lcls += 0.25 * self.BCEcls(ps_aux[:, 5:], t_aux)
```

YOLO v8: **CIoU + DFL + BCE**

3. 损失函数计算

$$Loss = \gamma_1 L_{cls} + \gamma_2 L_{CIOU} + \gamma_3 L_{DFL}$$

yolov8对$\gamma 1$,$\gamma 2$,$\gamma 3$默认分别取值0.5, 7.5, 1.5

观察上式不难发现，回归损失在CIOU_Loss的基础上新增了**Distribution Focal Loss(DFL)**

YOLO v9: **CIoU + DFL + BCE**，和 YOLO v8 一样

```
# cls loss
# loss[1] = self.varifocal_loss(pred_scores, target_scores, target_labels) / target_scores_sum  # VFL way
loss[1] = self.BCEcls(pred_scores, target_scores.to(dtype)).sum() / target_scores_sum # BCE
loss[1] *= 0.25
loss[1] += self.BCEcls(pred_scores2, target_scores.to(dtype)).sum() / target_scores_sum # BCE

# bbox loss
if fg_mask.sum():
    loss[0], loss[2], iou = self.bbox_loss(pred_distri,
                                           pred_bboxes,
                                           anchor_points,
                                           target_bboxes,
                                           target_scores,
                                           target_scores_sum,
                                           fg_mask)
    loss[0] *= 0.25
    loss[2] *= 0.25
if fg_mask.sum():
    loss0_, loss2_, iou2 = self.bbox_loss(pred_distri2,
                                           pred_bboxes2,
                                           anchor_points,
                                           target_bboxes,
                                           target_scores,
                                           target_scores_sum,
                                           fg_mask)
    loss[0] += loss0_
    loss[2] += loss2_

loss[0] *= 7.5  # box gain
loss[1] *= 0.5  # cls gain
loss[2] *= 1.5  # dfl gain

return loss.sum() * batch_size, loss.detach()  # loss(box, cls, dfl)
```

* See Class Compute in yolov9/utils/loss_tal_dual.py
https://github.com/WongKinYiu/yolov9/blob/main/utils/loss_tal_dual.py

Q: 在 YOLO v7 中，aux head 的 loss 在总的 loss 中占比更小（Total loss = 0.25 * aux head loss + lead loss）；但是**在 YOLO v9 中却是相反**，Total loss = 0.25 * lead head loss + aux loss，为什么？

YOLO v7 中的 aux head 辅助头是一个相对次要的监督信号，是真正意义上的“辅助”，学习的是浅层特征，缺少深层的语义信息，因此学习效果没有 lead head 好。如果给辅助头的损失赋予和主导头同等或更高的权重，模型可能会被强制去拟合一个“次优”的目标（辅助头的预测），这反而可能干扰主导头的优化。

而 YOLO v9 的辅助分支实际上是训练时的“**主力结构”，承担了解决核心问题（信息瓶颈、梯度流偏差）的关键角色。**它的任务是优化特征生成过程。
较高权重的 aux loss 主要驱动主干网络的优化。较低权重的 lead loss 则专注于在此基础上学习目标检测的具体任务（分类和定位）。主导头的预测依赖于主干网络提供的特征，如果特征本身不够好（包含信息瓶颈），主导头再努力优化其损失，也可能遇到瓶颈。

Q：那为什么要叫“辅助分支”？
是从推理角度出发，推理时无需使用此分支，不直接参与最终的预测输出，可以移除。

【References】

https://github.com/WongKinYiu/yolov9
2402.13616
【目标检测】YOLOv9 理论解读与代码分析-腾讯云开发者社区-腾讯云
YOLOv9 网络结构 backbone 中的 RepNCSPELAN4 和 ADown 模块-CSDN 博客
YOLOV9 論文導讀及使用 – CH.Tseng
YOLOv9 理性解读 | 网络结构&损失函数&耗时评估_yolov9 网络结构-CSDN 博客