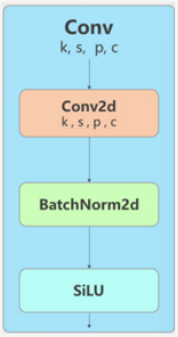
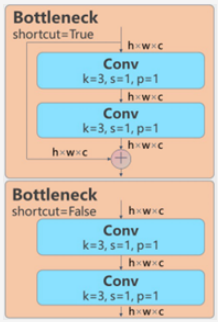
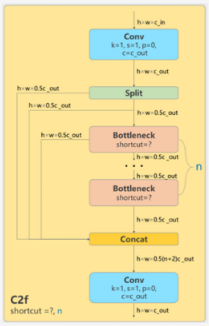
In [ ]:
```python
import torch
import torch.nn as nn

# ========================================
# PART 1: Conv (Conv + BN + SiLU)
# ========================================

class Conv(nn.Module):
  def __init__(self, in_channels, out_channels, kernel_size=3, stride=2, padding=1, groups=1, activation=True):
    super().__init__()
    self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bias=False, groups=groups)
    self.bn = nn.BatchNorm2d(out_channels)
    if activation:
      self.act = nn.SiLU(inplace=True)
    else:
      self.act = nn.Identity()

  def forward(self, x):
    return self.act(self.bn(self.conv(x)))
```

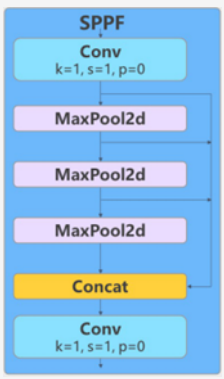


In [ ]:
```python
import torch
import torch.nn as nn

# ========================================
# PART 2: C2f - Bottleneck (Stack of 2 Conv with shortcut connection (T/F))
# ========================================

class Bottleneck(nn.Module):
  def __init__(self, in_channels, out_channels, shortcut=True):
    super().__init__()
    self.conv1 = Conv(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
    self.conv2 = Conv(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
    self.add = shortcut and in_channels == out_channels

  def forward(self, x):
    if self.add:
      return (x + self.conv2(self.conv1(x)))
    else:
      return (self.conv2(self.conv1(x)))

# ========================================
# PART 2: C2f (Conv + bottleneck*N + Conv)
# ========================================

class C2f(nn.Module):
  def __init__(self, in_channels, out_channels, num_bottlenecks = 1, shortcut=False, expansion=0.5):
    super().__init__()

    self.c = int(out_channels * expansion)
    self.cv1 = Conv(in_channels, 2*self.c, kernel_size=1, stride=1, padding=0)

    # Fix: Removed 'expansion' argument as Bottleneck does not accept it
    self.m = nn.ModuleList([Bottleneck(self.c, self.c, shortcut) for _ in range(num_bottlenecks)])

    self.cv2 = Conv((2 + num_bottlenecks) * self.c, out_channels, kernel_size=1, stride=1, padding=0)

  def forward(self, x):
    # Fix: Changed self.conv1 to self.cv1
    x = self.cv1(x)

    # Fix: Corrected syntax for x.split
    y = list(x.split((self.c, self.c), 1))

    # Apply bottleneck blocks and collect intermediate features
    y.extend(m(y[-1]) for m in self.m)

    # Concatenate all features and apply final convolution
    return self.cv2(torch.cat(y, 1))
```





In [ ]:
```python
# ========================================
# PART 3: SPPF (Spatial Pyramid Pooling Fast)
# ========================================

class SPPF(nn.Module):
  # k is for maxpool not the conv block
  def __init__(self, in_channels, out_channels, k=5):
    super().__init__()
    self.conv1 = Conv(in_channels, out_channels, kernel_size=1, stride=1, padding=0)
    # We set the kernel_size=5 because of MaxPool2d in YOLOv8 standard
    self.maxpool = nn.MaxPool2d(kernel_size=k, stride=1, padding=k//2)

    self.cv2 = Conv(4 * out_channels, out_channels, kernel_size=1, stride=1, padding=0)

  def forward(self, x):
    # First 1x1 Conv block
    x = self.conv1(x)

    # 2. Sequential Max Pooling
    y1 = self.maxpool(x)
    y2 = self.maxpool(y1)
    y3 = self.maxpool(y2)

    # Concat
    y = torch.cat((x, y1, y2, y3), 1)
    # Last Conv block
    return self.cv2(y)
```

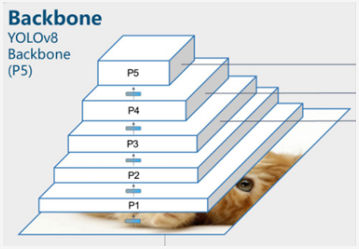


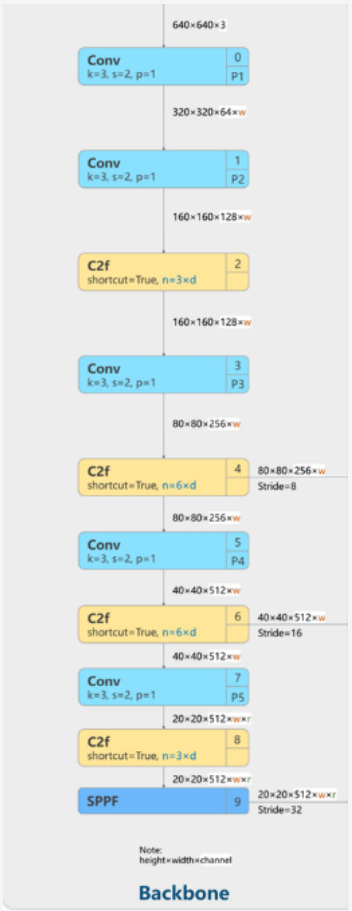
1. Backbone

The backbone is comprised of blocks Conv, C2f, SPPF

1. Conv: Conv2d + BatchNorm2d + SiLU
2. C2f (cross-stage partial bottleneck with 2 convolutions): Conv + Bottlenecks + Conv. C2f Combines high-level features with contextual information to improve detection accuracy.
3. SPPF (spatial pyramid pooling fast): Conv + Maxpool2d + Conv. Process features at various scales and pool them into a fixed-size feature map.

In [ ]:
```python
# ========================================
# Backbone
# ========================================

# return depth, width, ratio based on version
def yolo_parameter(version):
  params = {
      'n': [0.33, 0.25, 2.0],
      's': [0.33, 0.50, 2.0],
      'm': [0.67, 0.75, 1.5],
      'l': [1.00, 1.00, 1.0],
      'x': [1.00, 1.25, 1.0]
  }

  if version in params:
    return params[version]
  else:
    raise ValueError(f"Unknown version: {version}")

# Define backbone
class Backbone(nn.Module):
  def __init__(self, version, in_channels=3):
    super().__init__()
    d,w,r= yolo_parameter(version)

    # Conv Blocks/Layers
    self.conv_0 = Conv(in_channels, int(64*w), kernel_size=3, stride=2, padding=1)
    self.conv_1 = Conv(int(64*w), int(128*w), kernel_size=3, stride=2, padding=1)
    self.conv_3 = Conv(int(128*w), int(256*w), kernel_size=3, stride=2, padding=1)
    self.conv_5 = Conv(int(256*w), int(512*w), kernel_size=3, stride=2, padding=1)
    self.conv_7 = Conv(int(512*w), int(512*w*r), kernel_size=3, stride=2, padding=1)

    # C2f Blocks/Layers
    self.c2f_2 = C2f(int(128*w), int(128*w), num_bottlenecks=int(3*d) ,shortcut=True)
    self.c2f_4 = C2f(int(256*w), int(256*w), num_bottlenecks=int(6*d), shortcut=True)
    self.c2f_6 = C2f(int(512*w), int(512*w), num_bottlenecks=int(6*d), shortcut=True)
    self.c2f_8 = C2f(int(512*w*r), int(512*w*r), num_bottlenecks=int(3*d), shortcut=True)

    # SPPF
    self.sppf = SPPF(int(512*w*r), int(512*w*r))

  def forward(self, x):
    x = self.conv_0(x)
    x = self.conv_1(x)
    x = self.c2f_2(x)
    x = self.conv_3(x)
    C2f_out_4 = self.c2f_4(x) # keep for Neck Concat
    x = self.conv_5(C2f_out_4)
    C2f_out_6 = self.c2f_6(x) # keep for Neck Concat
    x = self.conv_7(C2f_out_6)
    x = self.c2f_8(x)
    SPPF_out_9 = self.sppf(x)
    return (C2f_out_4, C2f_out_6, SPPF_out_9)
```





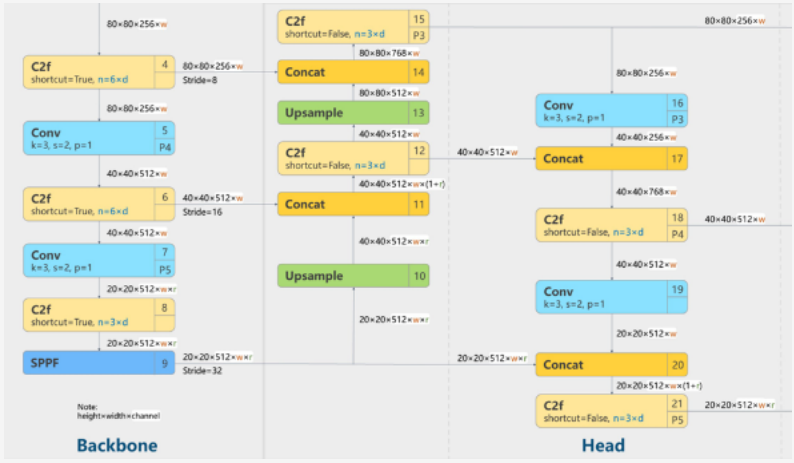
2. Neck

The neck comprises of Upsample + C2f Upsample: Nearest-neighbor interpolation with scale_factor=2. It doesn't have trainable parameters.

In [ ]:
```python
# ========================================
# PART1: Upsample
# ========================================
import torch.nn as nn

class Upsample(nn.Module):
  def __init__(self, scale_factor=2, mode='nearest'):
    super().__init__()
    self.upsample = nn.Upsample(scale_factor=scale_factor, mode=mode)

  def forward(self,x):
    x = self.upsample(x)
    return x
```

In [ ]:
```python
# ========================================
# Neck
# ========================================

class Neck(nn.Module):
  def __init__(self, version):
    super().__init__()
    d,w,r = yolo_parameter(version)

    # -- Top_down Path --
    # Fix: Use custom Upsample class instead of nn.Upsample
    self.up = Upsample()

    # C2f blocks
```

```
    self.c2f_12 = C2f(in_channels=int(512*w*(1+r)), out_channels=int(512*w), num_bottlenecks=int(3*d), shortcut=False)
    self.c2f_15 = C2f(in_channels=int(768*w), out_channels=int(256*w), num_bottlenecks=int(3*d), shortcut=False)
    # Fix: Corrected typo 'in_channnels' to 'in_channels'
    self.c2f_18 = C2f(in_channels=int(768*w), out_channels=int(512*w), num_bottlenecks=int(3*d), shortcut=False)
    self.c2f_21 = C2f(in_channels=int(512*w*(1+r)), out_channels=int(512*w*r), num_bottlenecks=int(3*d), shortcut=False)

    # Conv Blocks
    self.conv_16 = Conv(in_channels=int(256*w), out_channels=int(256*w), kernel_size=3, stride=2, padding=1)
    self.conv_19 = Conv(in_channels=int(512*w), out_channels=int(512*w), kernel_size=3, stride=2, padding=1)

  def forward(self, x, C2f_out_4, C2f_out_6, SPPF_out_9):
    # 1. Start with SPPF output (Index 9)
    x = self.up(SPPF_out_9)
    x = torch.cat((x, C2f_out_6), 1)

    C2f_out_12 = self.c2f_12(x)

    x = self.up(C2f_out_12)
    x = torch.cat((x, C2f_out_4), 1)

    C2f_out_15 = self.c2f_15(x)

    x = self.conv_16(C2f_out_15)
    x = torch.cat((x, C2f_out_12), 1)

    C2f_out_18 = self.c2f_18(x)

    x = self.conv_19(C2f_out_18)
    x = torch.cat((x, SPPF_out_9), 1)

    C2f_out_21 = self.c2f_21(x)

    return (C2f_out_15, C2f_out_18, C2f_out_21)
```

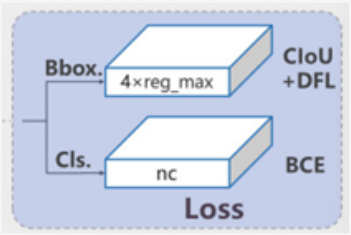


3. Head - DFL

DFL considers the predicted bbox coordinates as a probability distribution. At inference time, it samples from the distribution to get refined coordinates (x,y,w,h). For example, to predict coordinates x in the normalized range[0,1]:

1. DFL uses 16 bins which are equally spaced [0,1], bin length = 1/16
2. The model outputs 16 numbers which corresponds to probabilities that x falls in these bins, for example, [0,0,...,9/10, 1/10].
3. Prediction for x = mean value = 9/10 * 15/16 + 1/10*1 = 0.94375

In [ ]:
```
# ========================================
# DFL: Distribution focal loss
# ========================================
import torch.nn.functional as F

class DFL(nn.Module):
  # Integral module of Distrbution Focal Loss (DFL)
  def __init__(self, c1=16): #c1 is 'reg_max'
    super().__init__()
    # Creating constant tensor [0, 1, 2, ..., 15]
    self.conv = nn.Conv2d(in_channels=c1, out_channels=1, kernel_size=1, stride=1, bias=False).requires_grad_(False)
    self.conv.weight.data = torch.arange(c1).view(1, c1, 1, 1).float()
    self.c1 = c1

  def forward(self, x):
    # x shape: [batch, 4 * reg_max, anchors] -> [batch, 4, reg_max, anchors]
    b, c, a = x.shape
    x = x.view(b, 4, self.c1, a).transpose(2, 1)

    # Softmax to get a probability distribution (DFL theory)
    x = x.softmax(1)

    # Multiply by the [0, 1, ..., 15] vector to get the final coordinate
    x = self.conv(x).view(b,4,a)

    return x
```



3. Head

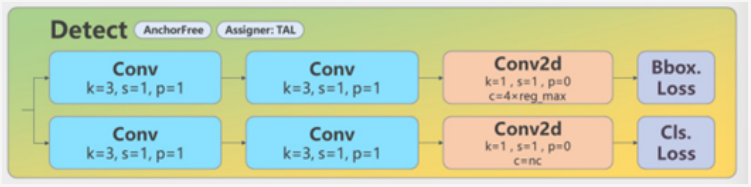
In YOLOv8, the Detect head separates the classification (Cls) and regression (Bbox) branches for each scale. It is also anchor-free, meaning it predicts the center of the object directly rather than an offset from an anchor box.

nc = number of classes, 80 is the default because it is the number of object categories in the COCO (Common Objects in Context) dataset, which is the standard benchmark used for pre-training these models.

- For KITTI, nc = 8
  1. Car
  2. Van
  3. Truck
  4. Pedestrian
  5. Person (sitting)
  6. Cyclist
  7. Tram
  8. Misc (Don't care)

In [ ]:
```
# ========================================
# Head: Detect
# ========================================

class Detect(nn.Module):
```

```
  def __init__(self, nc=8, ch=()): # nc=8 for KITTI classes
    super().__init__()
    self.nc = nc

    # number of detection layers (P3, P4, P5)
    self.nl = len(ch)

    # '4 reg_max'
    self.reg_max = 16

    # total outputs per anchor
    self.no = nc + self.reg_max * 4

    self.stride = torch.zeros(self.nl)

    # Regression branch (Bbox)
    self.bbox_branch = nn.ModuleList(nn.Sequential(
        Conv(x, x, kernel_size=3, stride=1, padding=1),
        Conv(x, x, kernel_size=3, stride=1, padding=1),
        nn.Conv2d(x, 4 * self.reg_max, 1) # Corrected output channels for bbox
    ) for x in ch)

    # Classification branch (Cls)
    self.cls_branch = nn.ModuleList(nn.Sequential(
        Conv(x, x, kernel_size=3, stride=1, padding=1),
        Conv(x, x, kernel_size=3, stride=1, padding=1),
        nn.Conv2d(x, self.nc, 1)
    ) for x in ch)

    self.dfl = DFL(self.reg_max)

  def forward(self, x):
    for i in range(self.nl):
      # Passing through the decoupled branches
      bbox = self.bbox_branch[i](x[i])
      cls = self.cls_branch[i](x[i])

      # During inference, apply DFL
      # bbox = self.dfl(bbox)

      x[i] = torch.cat((bbox, cls), 1)
    return x
```



4. Complete YOLOv8 KITTI Model

In [ ]:
```
class YOLOv8_KITTI(nn.Module):
  def __init__(self, version='s', nc=8):
    super().__init__()
    d, w, r = yolo_parameter(version)

    # 1. Backbone: Feature extraction
    self.backbone = Backbone(version)

    # 2. Neck: Feature fusion (Upsampling & Concatenation)
    self.neck = Neck(version)

    # 3. Detect Head: Prediction layers
    ch = [int(256*w), int(512*w), int(512*w*r)]
    self.detect = Detect(nc=nc, ch=ch)

  def forward(self,x):
    # Backbone output:
    features = self.backbone(x)

    # Neck output:
    neck_out = self.neck(None, *features)

    # Detect Head output: Concatenated Bbox and Cls tensors
    return self.detect(list(neck_out))
```