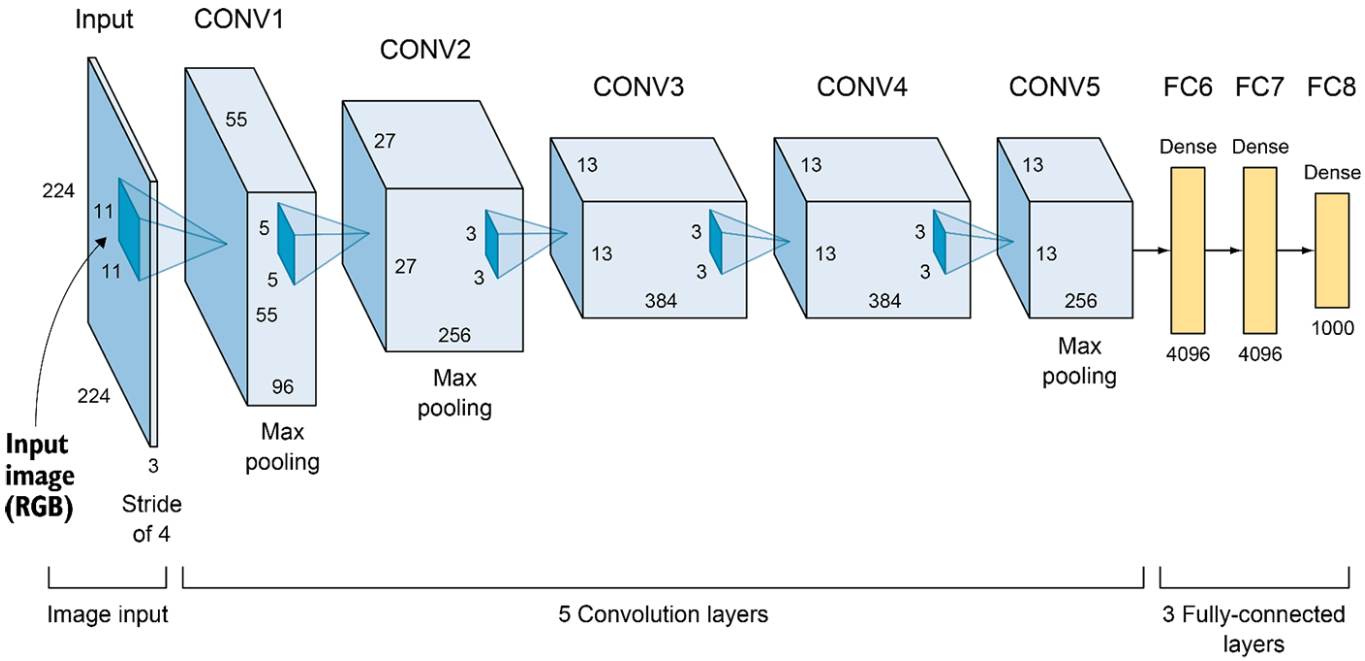
Models and Layers

Overview

当前的NeuroPredictor支持4种视觉模型库的调用: Timm, Torchvision, CLIP, OpenCLIP。其中，Timm通常用于调用ViT模型，Torchvision通常用于调用传统CNN模型，CLIP与OpenCLIP通常用于调用CLIP模型，特别的，某种Robust-ViT(CLIPAG)需要使用OpenCLIP调用。

你可以使用Getinfo.ipynb获取可调用模型，以及模型各层名称的信息。以下提供了几种常见模型层级结构与命名的介绍。

Alexnet(Old)



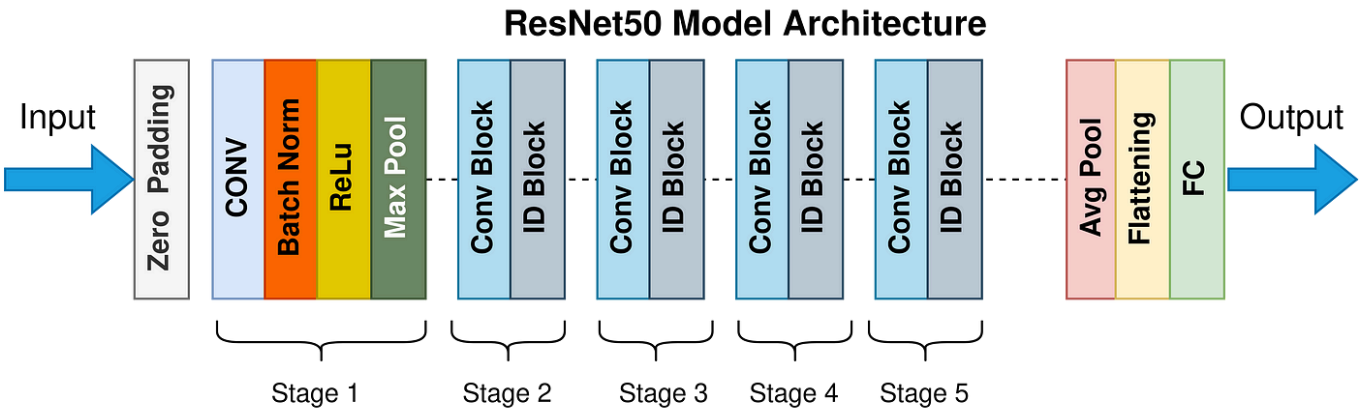
Alexnet中所有可调用的layer name:
'features.0', 'features.1', 'features.2', 'features.3', 'features.4', 'features.5', 'features.6', 'features.7', 'features.8', 'features.9', 'features.10', 'features.11', 'features.12', 'avgpool', 'classifier.0', 'classifier.1', 'classifier.2', 'classifier.3', 'classifier.4', 'classifier.5', 'classifier.6'
其中 `features.i` 对应conv层， `classifier.i` 对应fc层。

具体到每一层的对应关系:

- `features.0`： conv 1
- `features.1`： ReLU
- `features.2`： Max pooling 1
- `features.3`： conv 2
- `features.4`： ReLU
- `features.5`： Max pooling 2
- `features.6`： conv 3
- `features.7`： ReLU
- `features.8`： conv 4
- `features.9`： ReLU
- `features.10`： conv 5
- `features.11`： ReLU
- `features.12`： Max pooling 3
- `avgpool:` Avg pooling
- `classifier.0`： Dropout (p=0.5)
- `classifier.1`： fc 6
- `classifier.2`： ReLU
- `classifier.3`： Dropout
- `classifier.4`： fc 7
- `classifier.5`： ReLU
- `classifier.6`： fc 8

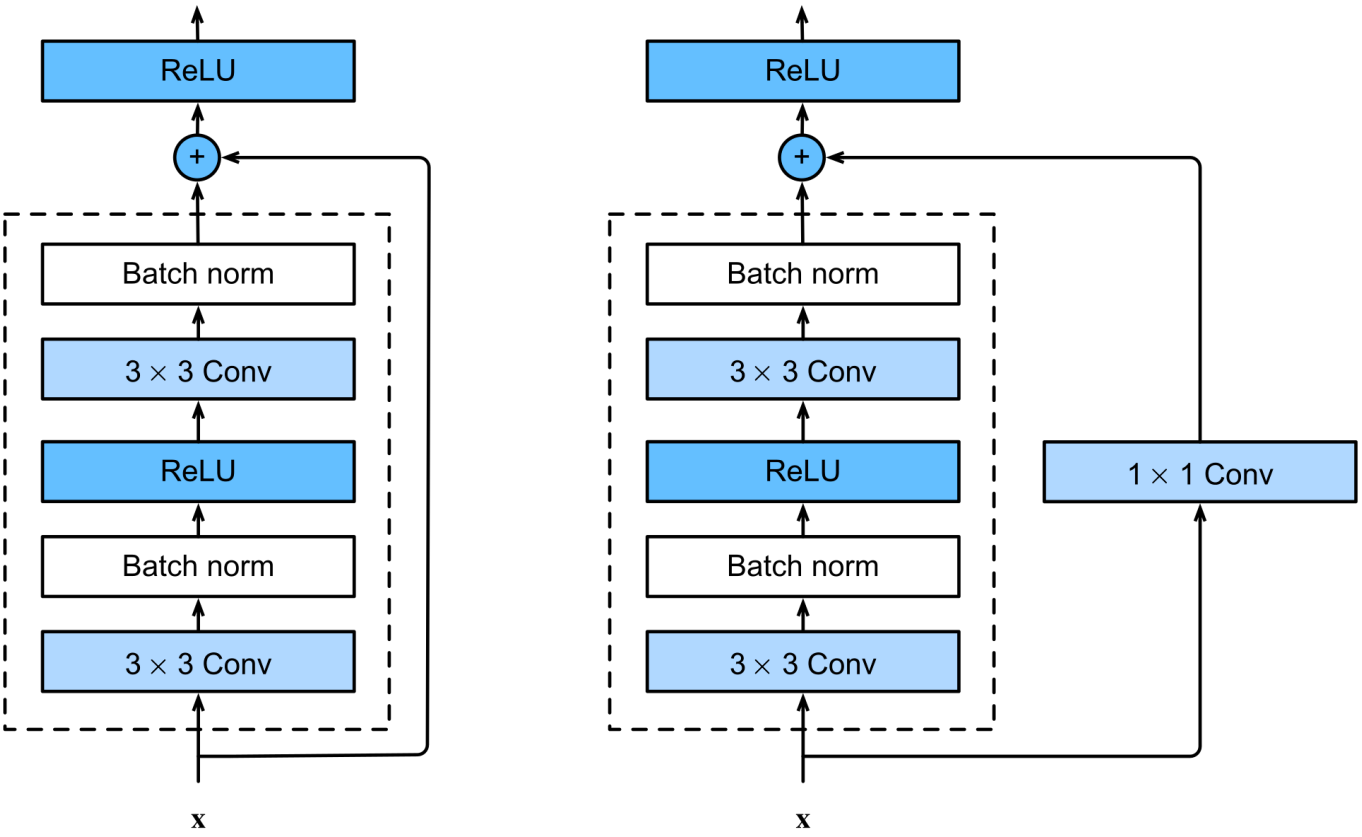
Resnet-50

整体架构



整体来看，Resnet-50可以划分为三部分: 预处理，Residual Blocks，输出层。如果想要提取模型各个residual block的最终输出，可以将layer name设置为 `layerX` , X代表希望提取的block index。

Residual Block



Resnet50共含4个residual block, 每个block内部又含多个残差连接 bottleneck。以第X个residual block的第Y个bottleneck `layerX.Y` 为例:

- `layerX.Y.conv1`
  - 1×1 卷积层，负责降维（从 `in_channels` → `out_channels/4`），减少后续 3×3 卷积的计算量。

- `layerX.Y.bn1`
  - 紧跟 `conv1` 的 BatchNorm，对通道维度做归一化，稳定训练并加速收敛。
- `layerX.Y.relu1`
  - 1×1 卷积后的 ReLU 激活，为后续非线性变换提供能力。
- `layerX.Y.conv2`
  - 3×3 卷积层，保持通道数不变（`out_channels/4 → out_channels/4`），用来整合空间邻域信息。
- `layerX.Y.bn2`
  - 紧跟 `conv2` 的 BatchNorm，继续维持数值稳定。
- `layerX.Y.relu2`
  - 3×3 卷积后的 ReLU 激活。
- `layerX.Y.conv3`
  - 1×1 卷积层，用来升维（`out_channels/4 → out_channels`），恢复到残差分支相加前的通道数。
- `layerX.Y.bn3`
  - 紧跟 `conv3` 的 BatchNorm。此时输出尚未激活。
- `layerX.Y.relu3`
  - 将主分支（`bn3` 输出）与残差分支（identity / downsample 输出）相加之后，再做一次 ReLU 激活，完成整个 Bottleneck Block 的输出。

伪代码：

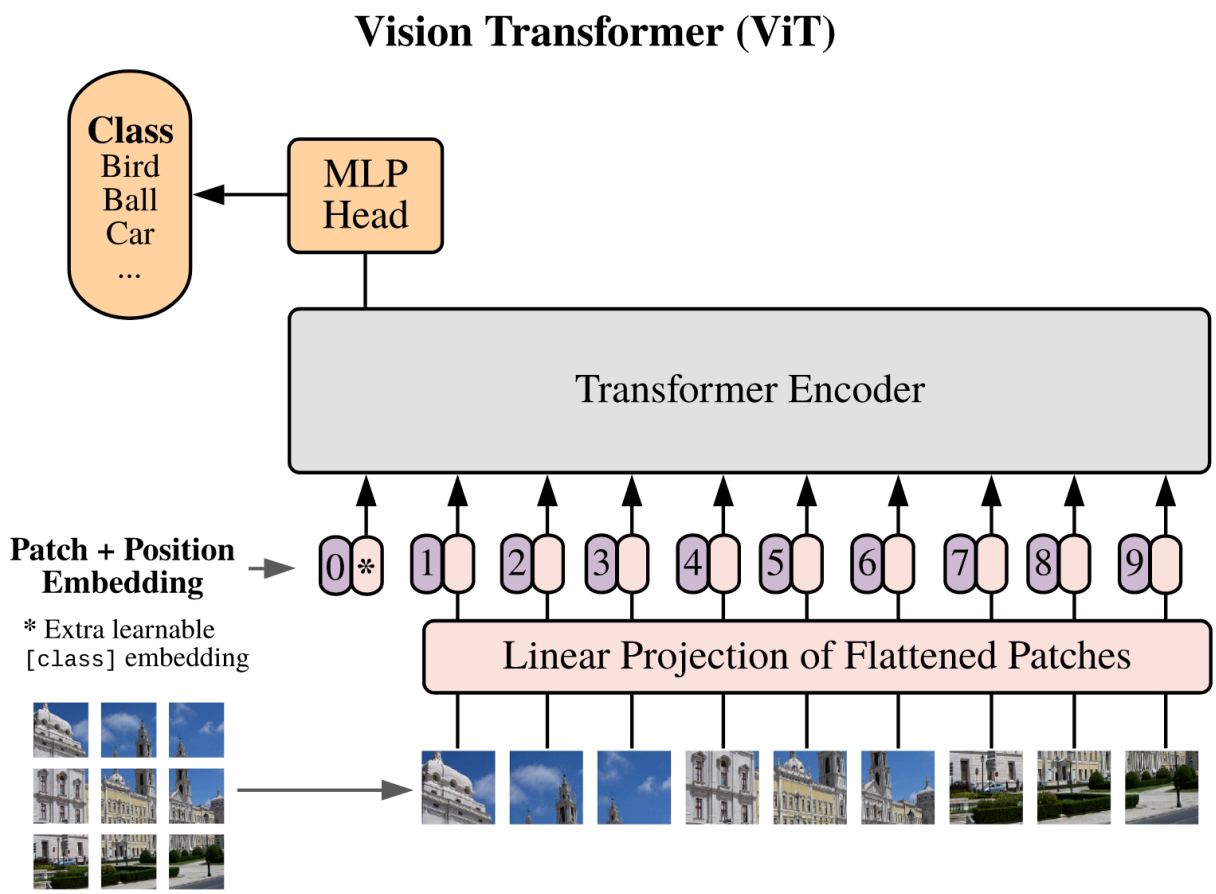
```
# 输入：x0
x0 = input
│
├─→ 1) 主分支（三层卷积 + BN + ReLU）
│     ├─ c1 = Conv1×1(x0)                  # layerX.Y.conv1
│     ├─ b1 = BatchNorm(c1)                # layerX.Y.bn1
│     ├─ r1 = ReLU(b1)                     # layerX.Y.relu1
│     │
│     ├─ c2 = Conv3×3(r1)                  # layerX.Y.conv2
│     ├─ b2 = BatchNorm(c2)                # layerX.Y.bn2
│     ├─ r2 = ReLU(b2)                     # layerX.Y.relu2
│     │
│     ├─ c3 = Conv1×1(r2)                  # layerX.Y.conv3
│     └─ b3 = BatchNorm(c3)                # layerX.Y.bn3
```

```
└─► 2) 残差分支（Identity 或 Downsample）
    ├─ if needs downsampling:
    │      ds1 = Conv1×1(x0,stride>1)   # layerX.Y.downsample.0
    │      ds2 = BatchNorm(ds1)         # layerX.Y.downsample.1
    │      x_res = ds2
    └─ else:
           x_res = x0                   # 恒等映射

└─► 3) 相加并激活
    ├─ out = b3 + x_res                 # 残差连接加主分支
    └─ x1  = ReLU(out)                  # layerX.Y.relu3
# 输出：x1
```

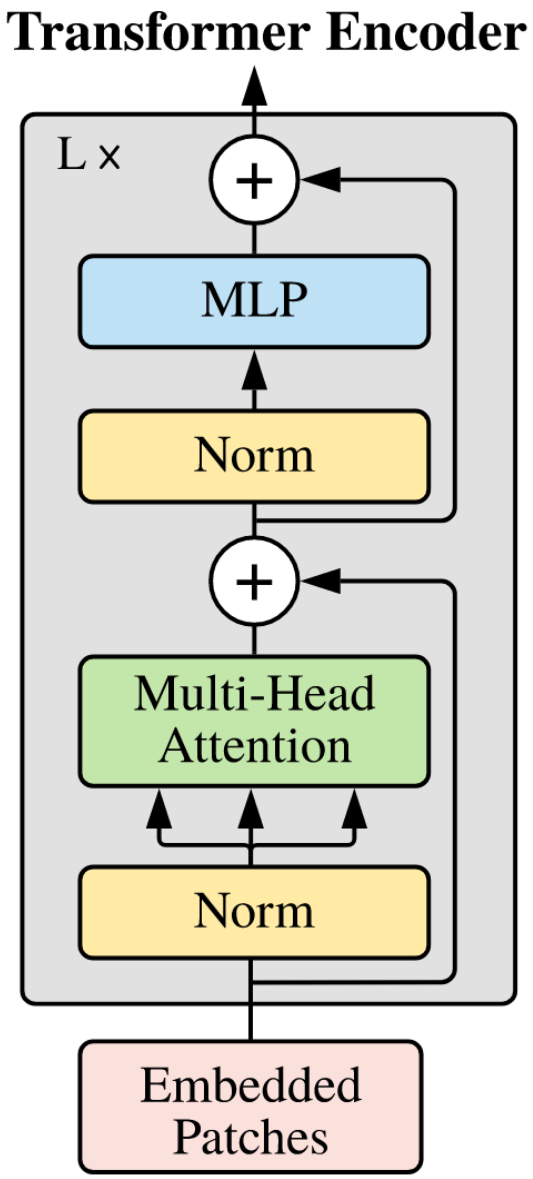
ViT

整体架构



整体来看，ViT可以划分为三部分: 预处理，Transformer Blocks，输出层。如果想要提取模型各个transformer block的最终输出，可以将layer_name设置为 `blocks.i` ，i代表希望提取的block index。如果想要提取ViT模型的最终输出，可以将layer_name设置为 `head` 。

Transformer Block



每个 `blocks.i` 的内部都可以拆分为Self‐Attention子层和MLP子层，以下是每个transformer block各个组件的名称与功能介绍:

- `blocks.i.norm1` 与 `blocks.i.norm2`：分别是第一和第二子层的 LayerNorm。
- `blocks.i.attn.qkv`：一个把输入同时映射到 Query、Key、Value 的三倍维度的 Linear 层（`3·768 → 3·768`），后面通过 reshape/split 拆成三份。

- `blocks.i.attn.attn_drop`：对注意力权重后的输出做 Dropout，防止过拟合。
- `blocks.i.attn.proj` + `proj_drop`：Self‑Attention 输出再投影回 `embed_dim` 并加 Dropout。
- `blocks.i.ls1` / `blocks.i.ls2`：LayerScale（可选），给残差分支乘上小的可学习系数 γ，稳定训练（OpenCLIP 的改进）。
- `blocks.i.drop_path1` / `drop_path2`：Stochastic Depth（随机残差丢弃），进一步正则化。
- `blocks.i.mlp.fc1` / `act` / `drop1` / `norm` / `fc2` / `drop2`：经典的两层 MLP（宽度通常是 `embed_dim×4 = 3072`），中间 GELU 激活，并串联了 Dropout 和一次小 LayerNorm。

伪代码：

```
x0 = input
│
├─► 1) Self-Attention block
│     ├─ x1 = LayerNorm(x0)                  # blocks.i.norm1
│     ├─ QKV  = Linear(x1)                   # blocks.i.attn.qkv
│     ├─ Q, K, V = split(QKV)
│     ├─ attention scores = softmax((QKᵀ)/√d_k)
│     ├─ attn_out = scores · V
│     ├─ attn_out = Dropout(attn_out)        # blocks.i.attn.attn_drop
│     ├─ proj_out = Linear(attn_out)         # blocks.i.attn.proj
│     ├─ proj_out = Dropout(proj_out)        # blocks.i.attn.proj_drop
│     ├─ if layer scale: proj_out *= γ1      # blocks.i.ls1
│     └─ x2 = x0 + DropPath(proj_out)         # blocks.i.drop_path1
│
└─► 2) MLP
      ├─ x3 = LayerNorm(x2)                  # blocks.i.norm2
      ├─ fc1_out = Linear(x3)                # blocks.i.mlp.fc1
      ├─ act_out = GELU(fc1_out)             # blocks.i.mlp.act
      ├─ act_out = Dropout(act_out)          # blocks.i.mlp.drop1
      ├─ act_out = LayerNorm(act_out)        # blocks.i.mlp.norm
      ├─ fc2_out = Linear(act_out)           # blocks.i.mlp.fc2
      ├─ fc2_out = Dropout(fc2_out)          # blocks.i.mlp.drop2
```

```
        ├─ if layer scale: fc2_out *= γ2      # blocks.i.ls2
        └─ x4 = x2 + DropPath(fc2_out)        # blocks.i.drop_path2
```