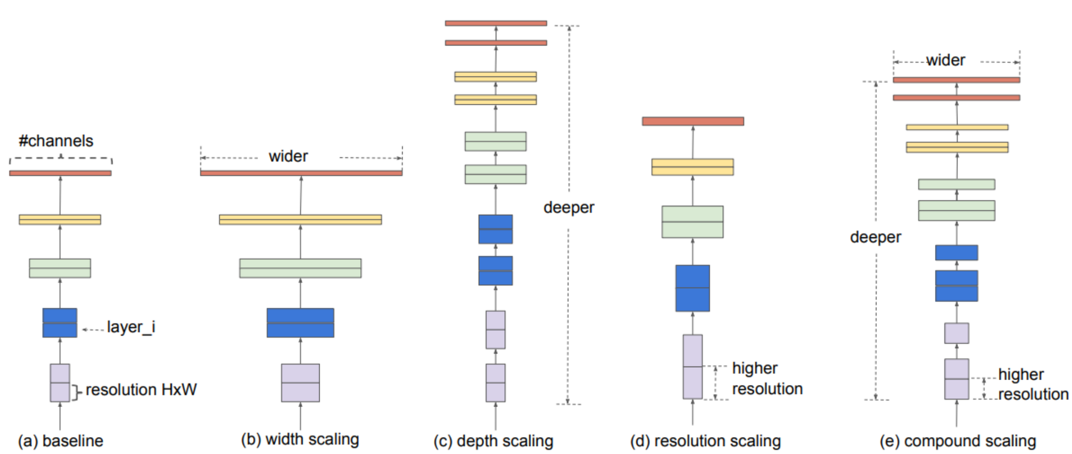
Crop Disease Detection using EfficientNet(Deep Learning)

What is EfficientNet?

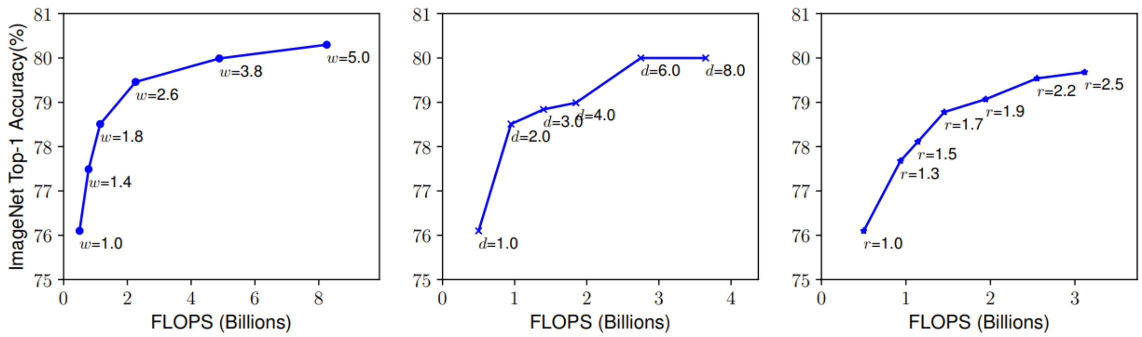
EfficientNet is a family of CNN architectures designed to achieve high accuracy with low computational cost. It was developed by Google in 2019. EfficientNet has several variants (e.g., EfficientNet-B0, EfficientNet-B1, ..., EfficientNet-B7), where the model size and accuracy increase as the index number increases. It supports Compound Scaling i.e. simultaneously scaling depth, width and resolution in balanced way.

Visualization for scaling width, depth, resolution and compound respectively



Results

Performing individual scaling on sample model i.e. ImageNet



FLOPS stands for **Floating Point Operations Per Second**. It is a measure of computational performance, indicating how many floating-point operations a model can perform in one second.

EfficientNet Architecture

Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Implementation of EfficientNet Architecture

```
import torch
import torch.nn as nn
from math import ceil

base_model = [
    # expand_ratio, channels, repeats, stride, kernel_size
    [1, 16, 1, 1, 3],
    [6, 24, 2, 2, 3],
    [6, 40, 2, 2, 5],
    [6, 80, 3, 2, 3],
    [6, 112, 3, 1, 5],
    [6, 192, 4, 2, 5],
    [6, 320, 1, 1, 3],
]

phi_values = {
    # tuple of: (phi_value, resolution, drop_rate)
    "b0": (0, 224, 0.2),  # alpha, beta, gamma, depth = alpha **
    "b1": (0.5, 240, 0.2),
    "b2": (1, 260, 0.3),
    "b3": (2, 300, 0.3),
    "b4": (3, 380, 0.4),
    "b5": (4, 456, 0.4),
    "b6": (5, 528, 0.5),
    "b7": (6, 600, 0.5),
}


class CNNBlock(nn.Module):
    def __init__(
        self, in_channels, out_channels, kernel_size, stride, pa
    ):
```

```
        super(CNNBlock, self).__init__()
        self.cnn = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size,
            stride,
            padding,
            groups=groups,
            bias=False,
        )
        self.bn = nn.BatchNorm2d(out_channels)
        self.silu = nn.SiLU()  # SiLU <-> Swish

    def forward(self, x):
        return self.silu(self.bn(self.cnn(x)))


class SqueezeExcitation(nn.Module):
    def __init__(self, in_channels, reduced_dim):
        super(SqueezeExcitation, self).__init__()
        self.se = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),  # C x H x W -> C x 1 x 1
            nn.Conv2d(in_channels, reduced_dim, 1),
            nn.SiLU(),
            nn.Conv2d(reduced_dim, in_channels, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return x * self.se(x)


class InvertedResidualBlock(nn.Module):
    def __init__(
        self,
        in_channels,
```

```
        out_channels,
        kernel_size,
        stride,
        padding,
        expand_ratio,
        reduction=4,  # squeeze excitation
        survival_prob=0.8,  # for stochastic depth
    ):
        super(InvertedResidualBlock, self).__init__()
        self.survival_prob = 0.8
        self.use_residual = in_channels == out_channels and stri
        hidden_dim = in_channels * expand_ratio
        self.expand = in_channels != hidden_dim
        reduced_dim = int(in_channels / reduction)

        if self.expand:
            self.expand_conv = CNNBlock(
                in_channels,
                hidden_dim,
                kernel_size=3,
                stride=1,
                padding=1,
            )

        self.conv = nn.Sequential(
            CNNBlock(
                hidden_dim,
                hidden_dim,
                kernel_size,
                stride,
                padding,
                groups=hidden_dim,
            ),
            SqueezeExcitation(hidden_dim, reduced_dim),
            nn.Conv2d(hidden_dim, out_channels, 1, bias=False),
            nn.BatchNorm2d(out_channels),
```

```
        )

    def stochastic_depth(self, x):
        if not self.training:
            return x

        binary_tensor = (
            torch.rand(x.shape[0], 1, 1, 1, device=x.device) < s
        )
        return torch.div(x, self.survival_prob) * binary_tensor

    def forward(self, inputs):
        x = self.expand_conv(inputs) if self.expand else inputs

        if self.use_residual:
            return self.stochastic_depth(self.conv(x)) + inputs
        else:
            return self.conv(x)


class EfficientNet(nn.Module):
    def __init__(self, version, num_classes):
        super(EfficientNet, self).__init__()
        width_factor, depth_factor, dropout_rate = self.calculat
        last_channels = ceil(1280 * width_factor)
        self.pool = nn.AdaptiveAvgPool2d(1)
        self.features = self.create_features(width_factor, depth
        self.classifier = nn.Sequential(
            nn.Dropout(dropout_rate),
            nn.Linear(last_channels, num_classes),
        )

    def calculate_factors(self, version, alpha=1.2, beta=1.1):
        phi, res, drop_rate = phi_values[version]
        depth_factor = alpha**phi
        width_factor = beta**phi
```

```
        return width_factor, depth_factor, drop_rate

    def create_features(self, width_factor, depth_factor, last_c
        channels = int(32 * width_factor)
        features = [CNNBlock(3, channels, 3, stride=2, padding=1
        in_channels = channels

        for expand_ratio, channels, repeats, stride, kernel_size
            out_channels = 4 * ceil(int(channels * width_factor)
            layers_repeats = ceil(repeats * depth_factor)

            for layer in range(layers_repeats):
                features.append(
                    InvertedResidualBlock(
                        in_channels,
                        out_channels,
                        expand_ratio=expand_ratio,
                        stride=stride if layer == 0 else 1,
                        kernel_size=kernel_size,
                        padding=kernel_size // 2,  # if k=1:pad=
                    )
                )
                in_channels = out_channels

        features.append(
            CNNBlock(in_channels, last_channels, kernel_size=1,
        )

        return nn.Sequential(*features)

    def forward(self, x):
        x = self.pool(self.features(x))
        return self.classifier(x.view(x.shape[0], -1))


def test():
```

```
    device = "cuda" if torch.cuda.is_available() else "cpu"
    version = "b0"
    phi, res, drop_rate = phi_values[version]
    num_examples, num_classes = 4, 10
    x = torch.randn((num_examples, 3, res, res)).to(device)
    model = EfficientNet(
        version=version,
        num_classes=num_classes,
    ).to(device)

    print(model(x).shape)  # (num_examples, num_classes)


if __name__ == "__main__":
    test()
```

1. Base Model and Phi Values

The `base_model` defines the architecture's building blocks:

- **Expand ratio**: Factor by which the number of channels is increased in the expansion phase.
- **Channels**: The number of output channels after each block.
- **Repeats**: Number of times the block is repeated.
- **Stride**: Stride for downsampling (stride of 2 reduces spatial dimensions).
- **Kernel size**: Size of the convolutional filter.

The `phi_values` dictionary provides configuration options for different EfficientNet versions (`b0` to `b7`):

- **Phi**: Determines the scaling factors.
- **Resolution**: Input image size.
- **Dropout rate**: Regularization to prevent overfitting.

CNN Block

Input (x) → [Convolution] → [Batch Normalization] → [SiLU Activation] → Output

What happens in Convolution?

1. Input Image

The `input_image` is a 5×5 matrix representing pixel values.

```
python
Copy code
input_image = np.array([
    [1, 1, 1, 0, 0],
    [0, 1, 1, 1, 0],
    [1, 1, 1, 1, 0],
    [0, 1, 1, 0, 1],
    [1, 1, 0, 1, 1]
])
```

2. Filter (Kernel)

The `filter_kernel` is a 3×3 matrix, which acts as the filter applied to the image.

```
python
Copy code
filter_kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
])
```

3. Convolution Process

- **Padding**: If specified, the input image is padded with zeros around the border.
- **Output Size Calculation**:
  The output dimensions are calculated using the formula:

$$OutputHeight = (ImageHeight - KernelHeight)/Stride + 1$$

$$\text{Output Width} = \frac{\text{Image Width} - \text{Kernel Width}}{\text{Stride}} + 1$$

$$\text{For this example:}$$

$$\text{Output Height} = \frac{5-3}{1} + 1 = 3$$

$$\text{Output Width} = \frac{5-3}{1} + 1 = 3$$

  So, the output feature map will be a 3×3 matrix.

- **Region Extraction**: The kernel is placed over a region of the image, and element-wise multiplication is performed, followed by summing up the values.

For example, for the top-left element:

- Image region:

```css
[ 1, 1, 1 ]
[ 0, 1, 1 ]
[ 1, 1, 1 ]
```

- Kernel:

```css
[ 1,  0, -1 ]
[ 1,  0, -1 ]
```

```
[ 1,  0, -1 ]
```

- Element-wise multiplication:

```css
css
Copy code
[ 1,  0, -1 ]
[ 0,  0, -1 ]
[ 1,  0, -1 ]
```

- Sum:
  1+0−1+0+0−1+1+0−1=−1

  1+0−1+0+0−1+1+0−1=−11 + 0 - 1 + 0 + 0 - 1 + 1 + 0 - 1 = -1

4. Output

The output feature map is a result of applying the kernel to the image:

```lua
lua
Copy code
[[-1. -3. -4.]
 [ 0. -2. -4.]
 [ 1.  1. -1.]]
```

What is Batch Normalization

Firstly a single batch consists of samples of data

- Batch normalization normalizes the inputs of each layer, typically the activations, by adjusting and scaling them. The idea is to make the distribution of the activations (outputs of neurons) consistent across the entire network and throughout the mini-batches of training. This helps to maintain stability during training.

Specifically, for each mini-batch, the normalization step performs the following:

- **Calculate the mean and variance** of the activations across the batch.
- **Normalize the activations** by subtracting the mean and dividing by the standard deviation (calculated from the variance).

- **Formula**:
  Let the input tensor to a layer be
  `x`, and suppose we are working with a mini-batch of size `N`. For each feature (dimension) `i`, batch normalization transforms the values in the mini-batch as follows:

$$\hat{x}_i = \frac{x_i - \mu}{\sigma}$$

  where:

  - xi is the original value of the feature in the mini-batch,
  - μ is the mean of the feature across the mini-batch,
  - σ is the standard deviation of the feature across the mini-batch.

  After normalization, the values of $\hat{xi}$ will have zero mean and unit variance.

Batch Normalization in Code:

In the `CNNBlock` class from the code you provided, batch normalization is applied right after the convolution operation:

```
python
Copy code
self.bn = nn.BatchNorm2d(out_channels)
```

- `nn.BatchNorm2d` is the PyTorch function for applying batch normalization to 2D data (like images or feature maps).
- `out_channels` refers to the number of output channels after the convolution. The normalization is applied to the output of the convolution to ensure that the activations for each feature map are normalized.

What is SiLU ?

It is an activation function better than ReLU and finds non linear relation in data

Formula :

$$SiLU(x) = x \cdot \sigma(x)$$

σ(x) is the **Sigmoid** function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Process of SE

1. `nn.AdaptiveAvgPool2d(1)`:
    - This performs **global average pooling** on the spatial dimensions (height and width) of the feature map, reducing each channel to a single value (the average across the spatial dimensions).
    - Input shape: `(batch_size, in_channels, H, W)`Output shape: `(batch_size, in_channels, 1, 1)`
2. `nn.Conv2d(in_channels, reduced_dim, 1)`:
    - This is a **pointwise convolution** (kernel size = 1) that reduces the number of channels from `in_channels` to `reduced_dim`. This is the **squeeze step**, where the channel-wise information is compressed.
3. `nn.SiLU()`:

- A non-linear activation function applied to the reduced representation to introduce non-linearity.

4. `nn.Conv2d(reduced_dim, in_channels, 1)`:
   - Another pointwise convolution that restores the number of channels back to `in_channels`. This is the **excitation step**, where the recalibrated importance weights for each channel are computed.
5. `nn.Sigmoid()`:
   - Applies a sigmoid function to normalize the recalibrated weights to the range `[0, 1]`. These weights are then used to scale the original feature map.

How the SE Block Works

1. **Squeeze (Global Average Pooling)**:
   - The spatial information is compressed by taking the global average of each channel. This gives a single value per channel that represents its overall "importance" across the spatial dimensions.
2. **Excitation (Recalibration)**:
   - A compact representation of channel importance (from the squeeze step) is passed through a small bottleneck network (the two 1×1 convolutions and SiLU activation). This bottleneck learns to prioritize or suppress channels based on their contribution to the task.
3. **Reweighting**:
   - The recalibrated weights are multiplied with the original feature map to enhance the important channels and suppress the irrelevant ones.

Example

Input Feature Map:

Suppose you have a feature map `x` with the following shape:

- `x.shape = (1, 4, 6, 6)` (Batch size = 1, Channels = 4, Height = 6, Width = 6)

Parameters:

- `in_channels = 4`
- `reduced_dim = 2` (calculated as `4 / 2` with a reduction ratio of 2).

Forward Pass:

1. **Global Average Pooling**:
   - For each channel, compute the average across the spatial dimensions:
     - Example for Channel 1: `x[:, 0, :, :].mean() = scalar_value`
   - Resulting shape: `(1, 4, 1, 1)`
2. **Squeeze (Pointwise Convolution)**:
   - The `(1, 4, 1, 1)` tensor is passed through a `1x1` convolution to reduce the number of channels to `reduced_dim = 2`.
   - Resulting shape: `(1, 2, 1, 1)`
3. **Non-Linearity**:
   - Apply the `SiLU` activation function to introduce non-linearity.
4. **Excitation (Pointwise Convolution)**:
   - Pass the `(1, 2, 1, 1)` tensor through another `1x1` convolution to restore the number of channels back to `in_channels = 4`.
   - Resulting shape: `(1, 4, 1, 1)`
5. **Sigmoid Activation**:
   - Apply the sigmoid function to normalize the recalibrated channel-wise weights to the range `[0, 1]`.
6. **Reweighting**:
   - Multiply the weights with the original feature map `x` element-wise. Each channel of the feature map is scaled by its corresponding weight.

InvertedResidual Block

Explanation:

1. `__init__` Method (Initialization):

The `InvertedResidualBlock` constructor takes several parameters related to the convolutional layers and block configuration.

- `in_channels`: The number of input channels for the block.
- `out_channels`: The number of output channels for the block.
- `kernel_size`: The size of the convolution kernel.
- `stride`: The stride of the convolution.
- `padding`: Padding added to the input for the convolution.
- `expand_ratio`: A factor that controls how much the number of channels should be expanded in the intermediate layer.
- `reduction`: A parameter used for the **Squeeze-and-Excitation (SE)** block to control the reduction in dimensionality.
- `survival_prob`: Probability for **stochastic depth**, which is a form of regularization.

2. Determine If Residual Connection Can Be Used:

```
python
Copy code
self.use_residual = in_channels == out_channels and stride ==
1
```

This line checks whether a residual connection can be used. A **residual connection** allows the input to skip certain layers and be added directly to the output. It is possible if:

- The number of input channels (`in_channels`) is the same as the number of output channels (`out_channels`).

- The stride is 1 (i.e., no downsampling is happening).

In the case that these conditions are met, the block will use a residual connection to add the input directly to the output. This helps avoid the problem of vanishing gradients and allows the model to learn more efficiently.

3. Hidden Dimension Calculation:

```
python
Copy code
hidden_dim = in_channels * expand_ratio
```

The **hidden dimension** is the number of channels in the intermediate layer of the block. It is determined by multiplying the input channels (`in_channels`) by the **expand ratio**. This **expands** the number of channels to a larger number for more expressiveness.

4. Expand or Not:

```
python
Copy code
self.expand = in_channels != hidden_dim
```

Here, we check whether we need to expand the number of channels. If the input channels (`in_channels`) are different from the hidden dimension (`hidden_dim`), then we need to **expand** the channels in the intermediate layer.

5. Squeeze and Excitation Reduction:

```
python
Copy code
reduced_dim = int(in_channels / reduction)
```

The **Squeeze-and-Excitation (SE)** block will reduce the number of channels to `reduced_dim`. The reduction factor (`reduction`) is typically set to 4, meaning the number of channels will be reduced by a factor of 4 in the SE block.

6. Expand Convolution (If Needed):

```
python
Copy code
if self.expand:
    self.expand_conv = CNNBlock(
        in_channels,
        hidden_dim,
        kernel_size=3,
        stride=1,
        padding=1,
    )
```

If the expansion condition is true (i.e., `in_channels != hidden_dim`), this block will use a **1×1 convolution** (implemented in `CNNBlock`) to expand the input channels to the hidden dimension (`hidden_dim`). This is the first step in the **inverted residual block**.

7. Main Convolutional Path:

```
python
Copy code
self.conv = nn.Sequential(
    CNNBlock(
        hidden_dim,
        hidden_dim,
        kernel_size,
        stride,
        padding,
        groups=hidden_dim,
    ),
    SqueezeExcitation(hidden_dim, reduced_dim),
```

```
    nn.Conv2d(hidden_dim, out_channels, 1, bias=False),
    nn.BatchNorm2d(out_channels),
)
```

This section defines the **main convolution path** for the block. It consists of multiple layers:

- **Depthwise Convolution** (`CNNBlock` with `groups=hidden_dim`):
  - Applies a convolution to each input channel separately (depthwise), which reduces computation.
- **Squeeze-and-Excitation Block**:
  - The SE block recalibrates the channels by computing attention weights for each channel, helping the model focus on important features.
- **1×1 Convolution**:
  - After the SE block, a **1×1 convolution** reduces the number of channels to the final output (`out_channels`).
- **Batch Normalization**:
  - Finally, batch normalization is applied to normalize the output and stabilize training.

8. Stochastic Depth (Optional):

```
python
Copy code
def stochastic_depth(self, x):
    if not self.training:
        return x

    binary_tensor = (
        torch.rand(x.shape[0], 1, 1, 1, device=x.device) < se
lf.survival_prob
    )
```

```
    return torch.div(x, self.survival_prob) * binary_tensor
```

Stochastic depth is a regularization technique where, during training, some layers are randomly skipped (dropped out). This is controlled by the `survival_prob`. The idea is to make the model more robust by not always using all layers.

9. Forward Pass:

```
python
Copy code
def forward(self, inputs):
    x = self.expand_conv(inputs) if self.expand else inputs

    if self.use_residual:
        return self.stochastic_depth(self.conv(x)) + inputs
    else:
        return self.conv(x)
```

During the **forward pass**:

- If expansion is needed, the input is passed through the **expand convolution**.
- Then, the input (after expansion, if needed) is passed through the **main convolution block** (`self.conv`).
- If a **residual connection** can be used (i.e., `use_residual` is `True`), the output of the convolution is added to the original input using a **skip connection**.

Visualizing the Block with an Example

Example:

Let's assume an input tensor with **32 channels** and **224×224 spatial dimensions**.

1. **Input**:
    - Tensor shape: `(batch_size, 32, 224, 224)`.

2. **Expansion (if needed)**:
    - If `expand_ratio = 6`, then `hidden_dim = 32 * 6 = 192`.
    - A convolution (`expand_conv`) will expand the input from 32 to 192 channels. Output shape: `(batch_size, 192, 224, 224)`.
3. **Depthwise Convolution** (`CNNBlock`):
    - This applies a depthwise convolution, reducing computation by applying a separate convolution to each channel. Output shape: `(batch_size, 192, 224, 224)`.
4. **Squeeze-and-Excitation (SE)**:
    - The SE block recalibrates the channels by focusing on important features. Output shape: `(batch_size, 192, 224, 224)`.
5. **1×1 Convolution**:
    - Reduces the number of channels to the final `out_channels` (say 64). Output shape: `(batch_size, 64, 224, 224)`.
6. **Batch Normalization**:
    - Normalizes the output. The shape remains the same: `(batch_size, 64, 224, 224)`.
7. **Residual Connection**:
    - If `in_channels == out_channels` and `stride == 1`, a residual connection is added: `(batch_size, 64, 224, 224) + (batch_size, 32, 224, 224)`. After this addition, the output is normalized.

Multiple layers of this are used i.e. InvertedResidual Block

EfficientNet

Code Breakdown

1. Initialization (`__init__` method)

```
python
Copy code
def __init__(self, version, num_classes):
    super(EfficientNet, self).__init__()
    width_factor, depth_factor, dropout_rate = self.calculate
_factors(version)
    last_channels = ceil(1280 * width_factor)
    self.pool = nn.AdaptiveAvgPool2d(1)
    self.features = self.create_features(width_factor, depth_
factor, last_channels)
    self.classifier = nn.Sequential(
        nn.Dropout(dropout_rate),
        nn.Linear(last_channels, num_classes),
    )
```

- **Parameters**:
    - `version`: Specifies the EfficientNet variant (e.g., `b0`, `b1`, etc.).
    - `num_classes`: Number of output classes for the classifier.
- **Steps**:
    1. **Scaling Factors**:
        - Calls `calculate_factors` to determine `width_factor`, `depth_factor`, and `dropout_rate` for the given version.
    2. **Feature Extraction**:
        - Calls `create_features` to build the main body of the model (stacked convolutional and inverted residual blocks).
    3. **Global Pooling**:
        - `AdaptiveAvgPool2d(1)` reduces the spatial dimensions to 1×1.
    4. **Classifier**:
        - Adds a dropout layer and a fully connected layer for classification.

2. Scaling Factor Calculation (`calculate_factors` method)

```
python
Copy code
def calculate_factors(self, version, alpha=1.2, beta=1.1):
    phi, res, drop_rate = phi_values[version]
    depth_factor = alpha**phi
    width_factor = beta**phi
    return width_factor, depth_factor, drop_rate
```

- **Purpose**:
  Calculates:
  - `width_factor`: How much to scale the number of channels.
  - `depth_factor`: How much to scale the number of blocks.
  - `drop_rate`: Dropout rate for the classifier.
- **Formula**:
  - `width_factor = beta ** phi`
  - `depth_factor = alpha ** phi`

---

3. Feature Creation (`create_features` method)

```
python
Copy code
def create_features(self, width_factor, depth_factor, last_ch
annels):
    channels = int(32 * width_factor)
    features = [CNNBlock(3, channels, 3, stride=2, padding=
1)]
    in_channels = channels

    for expand_ratio, channels, repeats, stride, kernel_size
in base_model:
```

```
        out_channels = 4 * ceil(int(channels * width_factor)
/ 4)
        layers_repeats = ceil(repeats * depth_factor)

        for layer in range(layers_repeats):
            features.append(
                InvertedResidualBlock(
                    in_channels,
                    out_channels,
                    expand_ratio=expand_ratio,
                    stride=stride if layer == 0 else 1,
                    kernel_size=kernel_size,
                    padding=kernel_size // 2,
                )
            )
            in_channels = out_channels

    features.append(
        CNNBlock(in_channels, last_channels, kernel_size=1, s
tride=1, padding=0)
    )

    return nn.Sequential(*features)
```

- **Steps**:
  1. **Initial Convolution**:
     - Adds a `CNNBlock` to increase the input channels (from 3) to `32 * width_factor`.
  2. **Inverted Residual Blocks**:
     - Iterates over `base_model` and repeats `InvertedResidualBlock` layers based on `depth_factor`.
     - Handles scaling of channels and repetitions.

3. **Final Convolution**:
    - Adds a `CNNBlock` to increase the number of channels to `last_channels` (scaled from 1280).

---

4. Forward Pass

```
python
Copy code
def forward(self, x):
    x = self.pool(self.features(x))
    return self.classifier(x.view(x.shape[0], -1))
```

- **Steps**:
    1. Passes the input through the **feature extractor** (`self.features`).
    2. Applies **global average pooling** to reduce spatial dimensions to 1×1.
    3. Flattens the tensor and passes it to the **classifier** to output class probabilities.

Example

1. **Model**: EfficientNet `b0`
    - **Scaling Factors**: `phi = 0`, resolution = `224`, dropout rate = `0.2`.
    - Depth scaling: `alpha^phi = 1.0`
    - Width scaling: `beta^phi = 1.0`
    - Number of output classes: `10`.
2. **Input Image**: A batch of 2 images with size `(3, 224, 224)`:
    - Shape: `(batch_size=2, channels=3, height=224, width=224)`.

Flow Through the Model

1. Initialization

- **Initial Channels**: `32 * width_factor = 32`.
- **Last Channels**: `1280 * width_factor = 1280`.
- The `create_features` method builds the network with scaled layers.

---

2. Initial Convolution

The first `CNNBlock` processes the input:

```
python
Copy code
features = [CNNBlock(3, channels, 3, stride=2, padding=1)]
```

- **Input Shape**: `(2, 3, 224, 224)` (3 input channels).
- **Operation**: Convolution with 32 filters, kernel size = 3×3, stride = 2, padding = 1.
- **Output Shape**: `(2, 32, 112, 112)`.

---

3. Stacked Inverted Residual Blocks

The `base_model` defines the sequence of layers. For each block:

- **Expand** channels, apply depthwise convolution, SE block, and residual connection.
- Repeat the block as per depth scaling.

Example: First block in `base_model`:

```
python
Copy code
[1, 16, 1, 1, 3]  # expand_ratio=1, out_channels=16, repeats=
```

```
1, stride=1, kernel_size=3
```

- **Input Shape**: `(2, 32, 112, 112)`.
- **Expanded Channels**: 32 → 32 (expand ratio = 1).
- **Depthwise Convolution**: Kernel size = 3×3, stride = 1.
- **Squeeze-Excitation**: Recalibrates channel importance.
- **Residual Connection**: Adds input if stride = 1.
- **Output Shape**: `(2, 16, 112, 112)`.

This process repeats for all layers in `base_model`.

---

4. Final Convolution

After processing through all the blocks:

```
python
Copy code
features.append(
    CNNBlock(in_channels, last_channels, kernel_size=1, strid
e=1, padding=0)
)
```

- **Input Shape**: `(2, channels, h, w)` (channels depend on the last block).
- **Operation**: 1×1 Convolution to upscale channels to `1280`.
- **Output Shape**: `(2, 1280, h, w)`.

---

5. Global Pooling

```
python
Copy code
```

```
x = self.pool(self.features(x))
```

- **Operation:** `AdaptiveAvgPool2d(1)` reduces the spatial dimensions (H, W) to `1x1`.
- **Input Shape**: `(2, 1280, h, w)`.
- **Output Shape**: `(2, 1280, 1, 1)`.

6. Classifier

```
python
Copy code
return self.classifier(x.view(x.shape[0], -1))
```

- **Flatten**: Convert `(2, 1280, 1, 1)` → `(2, 1280)`.
- **Dropout**: Randomly drops some features during training (drop rate = 0.2).
- **Linear Layer**: Fully connected layer maps `1280` features to `10` output classes.
- **Output Shape**: `(2, 10)`.

Visualization of Shapes

Layer	Input Shape	Output Shape
Initial Conv	`(2, 3, 224, 224)`	`(2, 32, 112, 112)`
Inverted Residual Block (1)	`(2, 32, 112, 112)`	`(2, 16, 112, 112)`
...	...	...
Final Conv	`(2, channels, h, w)`	`(2, 1280, h, w)`
Global Pooling	`(2, 1280, h, w)`	`(2, 1280, 1, 1)`
Classifier	`(2, 1280)`	`(2, 10)`

Output

For a batch of size `2` and `10` classes, the final output is:

```
python
Copy code
tensor([[0.1, 0.05, ..., 0.15],  # Probabilities for image 1
        [0.2, 0.1, ..., 0.05]])  # Probabilities for image 2
```

Conclusion

EfficientNet provides compound scaling effective CNN for image processing