VGG16 and ResNet50 Implementation

VGG16 and ResNet50 Implementation

In previous lectures, we covered the fundamentals of CNNs. We solved two problems using simple CNN architectures and continued our exploration by studying the evolution of various CNN models over time. We compared these models to understand their innovations as well as their limitations.

Now, we will focus on ResNet50 and VGG16, as they are among the most frequently used architectures today. For each model, we will briefly review its specifications. Then, we will analyze its architecture further to better understand the challenges of defining these models.

Finally, we will start building the networks from scratch using Python. We will integrate everything we have learned to define the models effectively.

In the end, we will introduce the pre-trained models, which help us utilize these powerful models more easily.

VGG16 and ResNet50 Implementation

VGG16 specifications

The VGG model stands for Visual Geometry Group from the University of Oxford. This model is relatively simple yet deeper than AlexNet. This enables the model to capture features more effectively, resulting in improved accuracy on benchmarks.
The architecture utilizes only 3x3 filters. The number of filters in each layer increases as we progress deeper into the network, while the use of max pooling decreases the size of the feature maps.
The original paper presented two models: one with 16 layers and the other with 19 layers. The model we are going to implement today is the one with 16 layers.

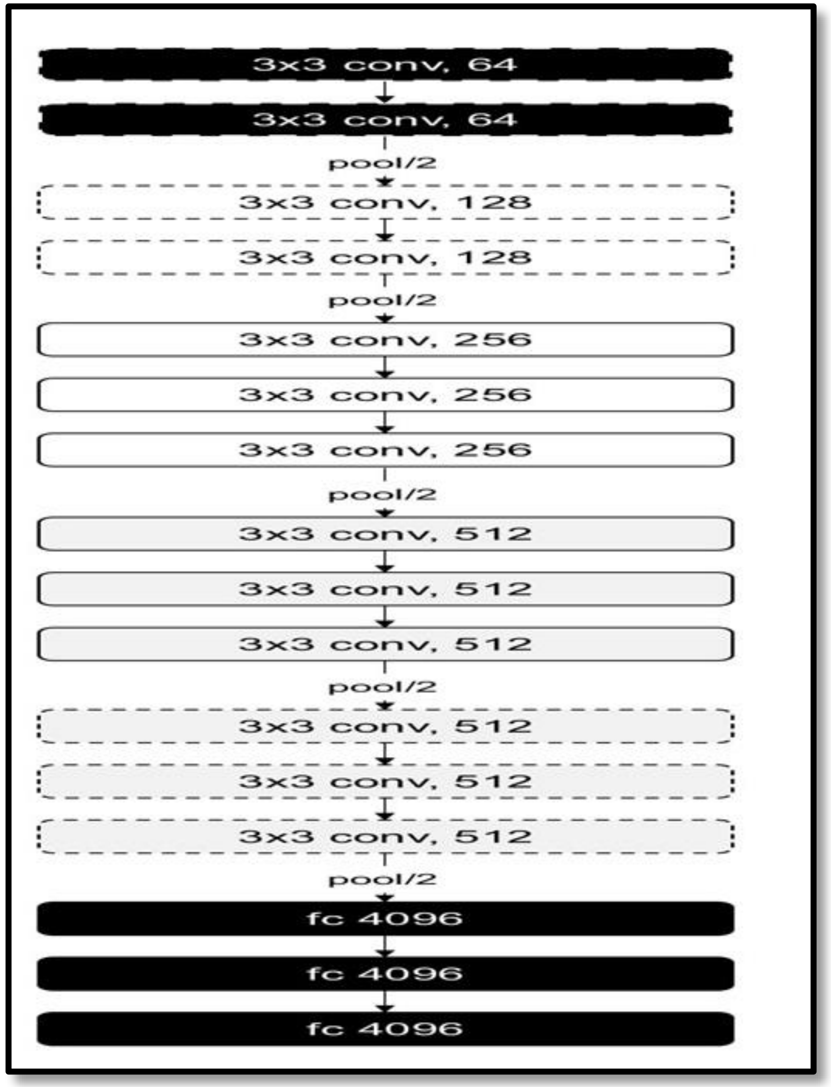
VGG16 and ResNet50 Implementation

VGG16 architecture

Before defining the VGG16 using Python, we need to review its architecture for better understanding.

The model consists of 13 convolutional layers and 3 fully connected layers. The first two layers each consist of 64 filters. On top of these, we have 2 convolutional layers, each having 128 filters, followed by another 3 convolutional layers with 256 filters. The last 6 convolutional layers each have 512 filters.

Finally, the output is flattened, and the 3 fully connected layers are added. There are also 5 max pooling layers in the model, with their placement between the layers shown in the figure.



VGG16 and ResNet50 Implementation

VGG16 Implementation

Now, we are going to define the VGG16 model using Python. We have already learned how to build simple architectures using Conv2D and MaxPooling2D layers in Python. VGG16 can be defined by putting many of these layers together.

As simple as it may seem, there are a few principles we need to follow to write efficient code. For example, imagine we are solving a problem with an input image of 224 x 224 pixels using VGG16. We define a model with an input size of 224 x 224. Later, when solving another problem, we decide to use the VGG16 structure again, but this time the input size is 512 x 512.

We can see the importance of reusability here. In programming, it is recommended to write code in a way that allows it to be used in other programs with minimal changes.

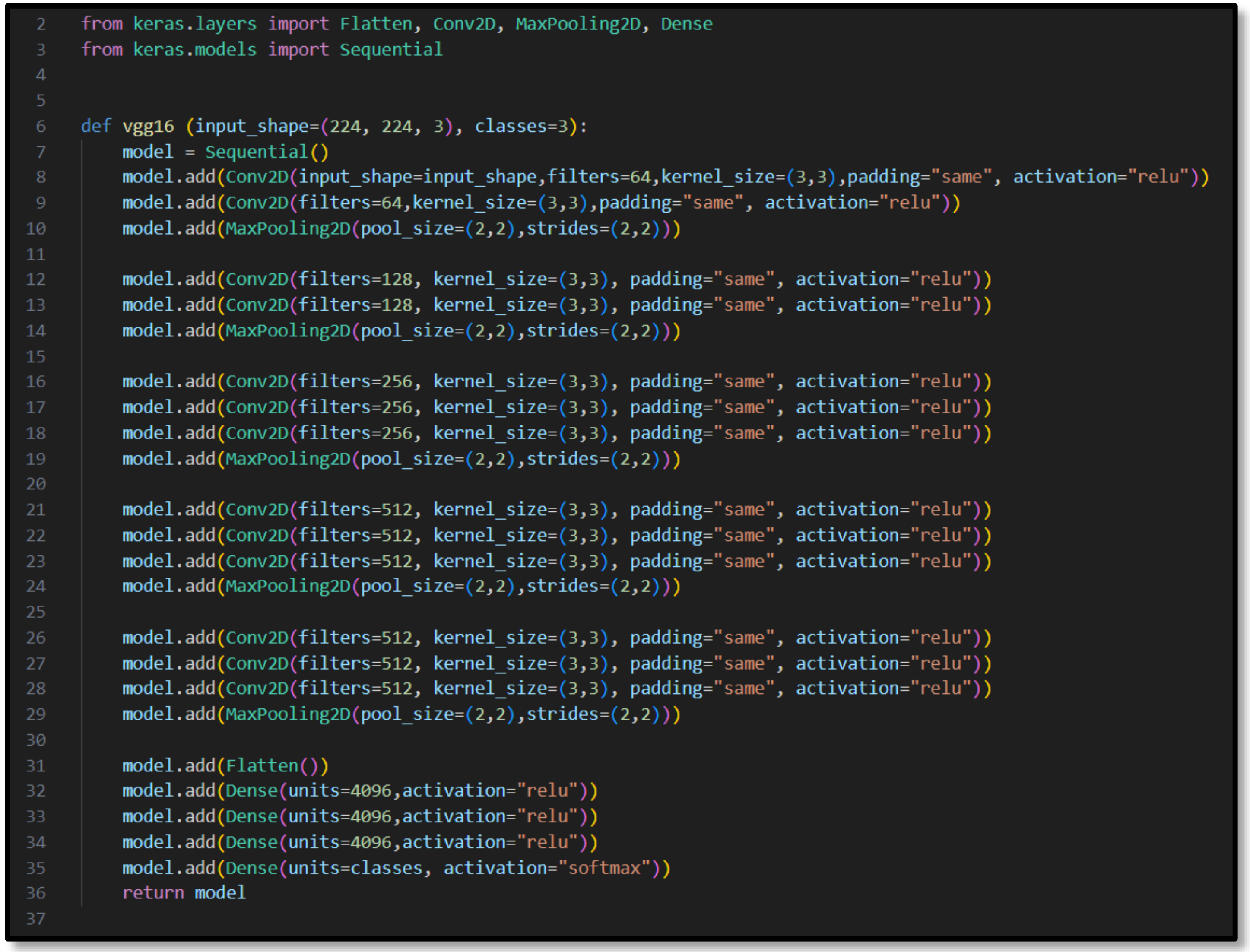
In this example, instead of building a VGG16 architecture with a fixed input size, we will create a function that takes the input dimensions and the number of classes, returning a VGG16 model with that configuration.

This approach will facilitate the development of further programs where we can use the VGG16 structure with different input sizes and varying numbers of classes.

VGG16 and ResNet50 Implementation

VGG16 Implementation

In the function, we will obtain the input shape and the number of classes. The first step is to define a sequential model. Then, we will add the layers one by one, paying attention to the structure of the VGG16 model. Once all the layers have been added, we will return the model.
The next time we need a VGG16 model, we can simply call this function, providing the appropriate input shape and number of classes.
Upon closer inspection, we can see that there are many duplicate lines of code, indicating that a better implementation using for loops is possible. In the next slide, we will explore that.

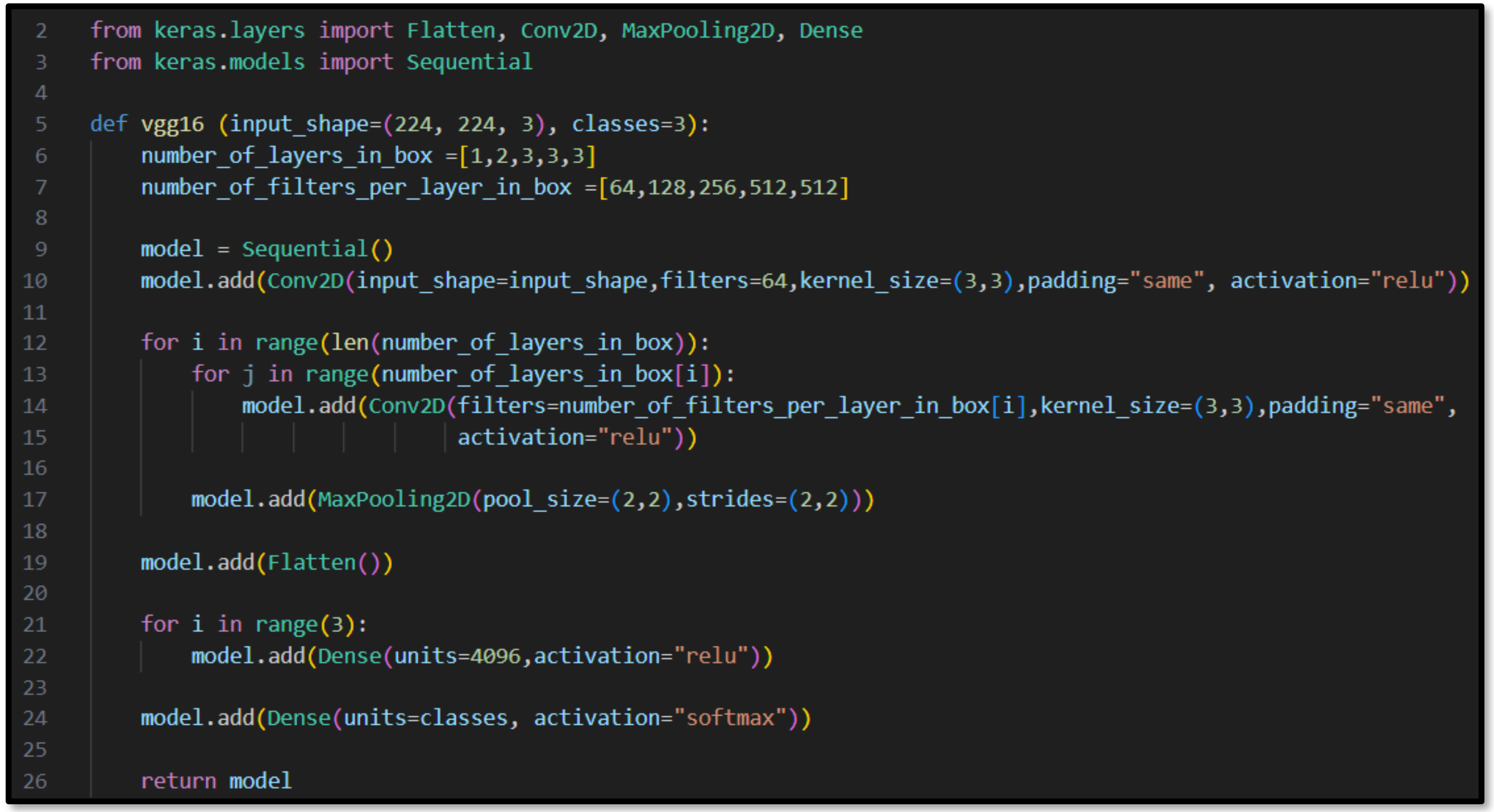


```python
from keras.layers import Flatten, Conv2D, MaxPooling2D, Dense
from keras.models import Sequential


def vgg16 (input_shape=(224, 224, 3), classes=3):
    model = Sequential()
    model.add(Conv2D(input_shape=input_shape,filters=64,kernel_size=(3,3),padding="same", activation="relu"))
    model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

    model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

    model.add(Flatten())
    model.add(Dense(units=4096,activation="relu"))
    model.add(Dense(units=4096,activation="relu"))
    model.add(Dense(units=4096,activation="relu"))
    model.add(Dense(units=classes, activation="softmax"))
    return model

```

VGG16 and ResNet50 Implementation

VGG16 Implementation

In this new implementation, we have grouped every layer between two max pooling layers together. We did this because all of these layers have the same number of filters.
Within each "box" we formed, there is a certain number of layers, each having a specific number of filters. We created two lists that store this information for us.
Instead of defining every layer one by one, we can use nested for loops. For each box, we check the number of layers it contains. Then, for the specified number of layers, we add a convolutional layer with the corresponding number of filters.



```python
from keras.layers import Flatten, Conv2D, MaxPooling2D, Dense
from keras.models import Sequential

def vgg16 (input_shape=(224, 224, 3), classes=3):
    number_of_layers_in_box =[1,2,3,3,3]
    number_of_filters_per_layer_in_box =[64,128,256,512,512]

    model = Sequential()
    model.add(Conv2D(input_shape=input_shape,filters=64,kernel_size=(3,3),padding="same", activation="relu"))

    for i in range(len(number_of_layers_in_box)):
        for j in range(number_of_layers_in_box[i]):
            model.add(Conv2D(filters=number_of_filters_per_layer_in_box[i],kernel_size=(3,3),padding="same",
                                          activation="relu"))

        model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))

    model.add(Flatten())

    for i in range(3):
        model.add(Dense(units=4096,activation="relu"))

    model.add(Dense(units=classes, activation="softmax"))

    return model
```

VGG16 and ResNet50 Implementation

ResNet50 specifications

The ResNet50 model was first introduced in 2015 by researchers at Microsoft Research Asia. It is part of the Residual Networks family, designed to address the vanishing gradient problem in deep neural networks through the use of skip connections and shortcut connections.

This model incorporates 50 bottleneck residual blocks, allowing it to achieve high accuracy in image classification tasks on datasets like ImageNet. Initially, the model was designed to accept images of size 224x224 pixels, but with minor adjustments, it can also accommodate other image sizes.

VGG16 and ResNet50 Implementation

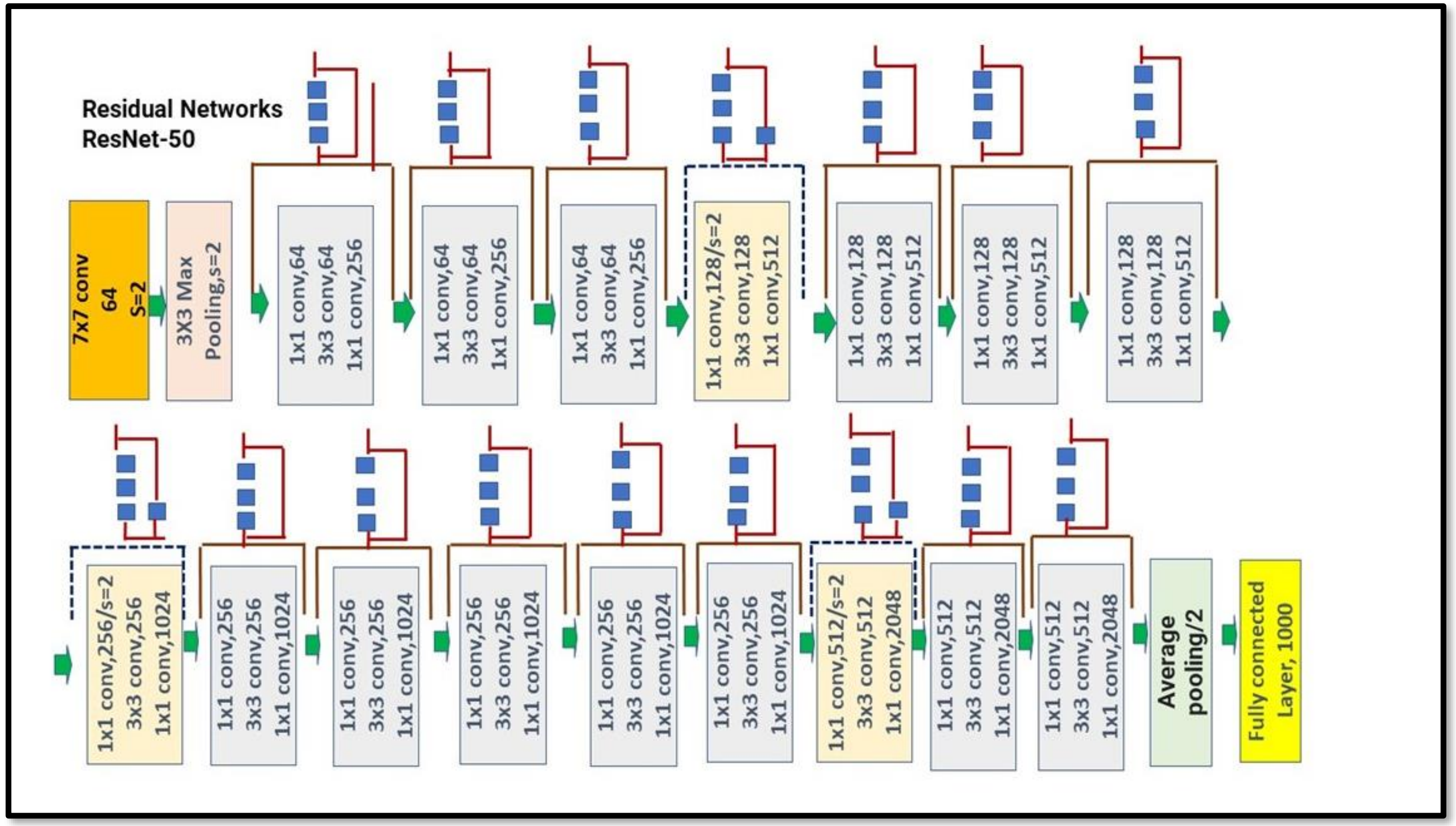
ResNet50 architecture

We know that the ResNet50 model has two different types of blocks: the convolutional block and the identity block.

In the identity block, the size of the output remains consistent with the size of the input. In contrast, the convolutional blocks have input sizes that change.

There are 13 identity blocks and 3 convolutional blocks in this structure. Each of these blocks consists of three convolutional layers. The size of the filters increases as we progress deeper into the network.

There is one max pooling layer at the beginning of the network and an average pooling layer at the end.



Residual Networks
ResNet-50
7x7 conv 64 S=2
3X3 Max Pooling,s=2
1x1 conv,64 3x3 conv,64 1x1 conv,256
1x1 conv,128/s=2 3x3 conv,128 1x1 conv,512
1x1 conv,128 3x3 conv,128 1x1 conv,512
1x1 conv,256/s=2 3x3 conv,256 1x1 conv,1024
1x1 conv,256 3x3 conv,256 1x1 conv,1024
1x1 conv,512/s=2 3x3 conv,512 1x1 conv,2048
1x1 conv,512 3x3 conv,512 1x1 conv,2048
Average pooling/2
Fully connected Layer, 1000

VGG16 and ResNet50 Implementation

ResNet50 Implementation

In this section, we will define the ResNet50 model in Python. In the previous slide, we discussed the architecture of the ResNet50 model and observed that it can be broken down into different identity and convolutional blocks. We will use this feature to build the network.

Imagine we have defined both a convolutional block and an identity block. Now, building a ResNet50 model will be relatively straightforward; we only need to assemble these blocks to form the network.

We will follow the same approach that we utilized to build the VGG16 model.

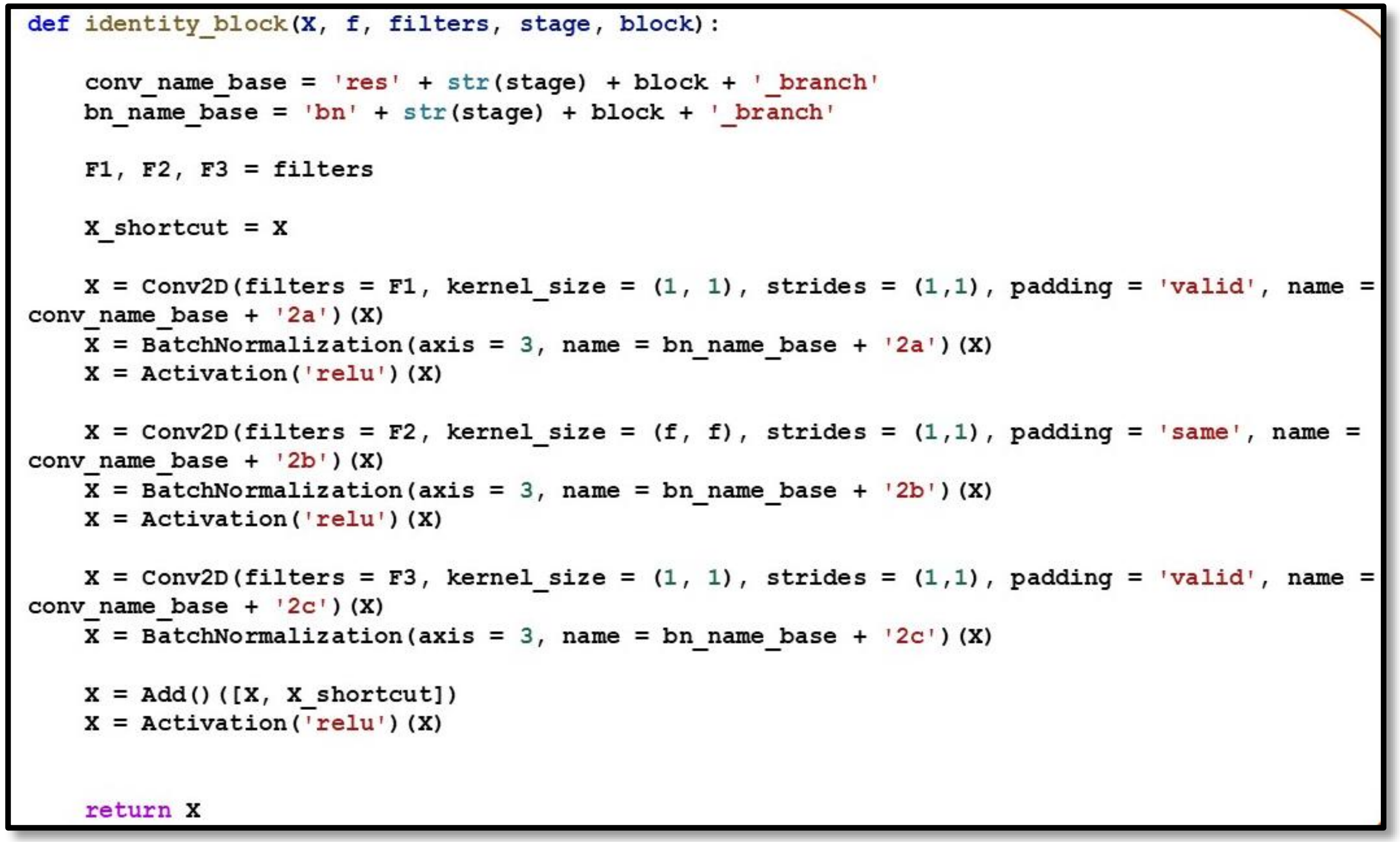
We are going to define a function that takes the number of classes and the input size as inputs, returning the ResNet50 model with the appropriate configurations. Additionally, we will define two auxiliary functions.

The first auxiliary function will take a list that specifies the number of filters in each of the three convolutional layers of the identity block and apply the identity block to the input x. We will define another similar function for the convolutional block. The differences between these two blocks will be discussed further in the following slides.

VGG16 and ResNet50 Implementation

ResNet50 Implementation

We will start with the function responsible for applying an identity block to its input.
This function has three main inputs: X, f, and filters.
X is the input we want to apply the identity block to. Filters is a list that determines the number of filters in each layer. f defines the size of the middle layer filter.
The other two inputs, stage and block, are used only for naming purposes. If we print the model summary, the names we assigned to each layer will be displayed.
Before performing any transformation on the input, we save the raw input in a variable. Finally, we add this variable to our output.

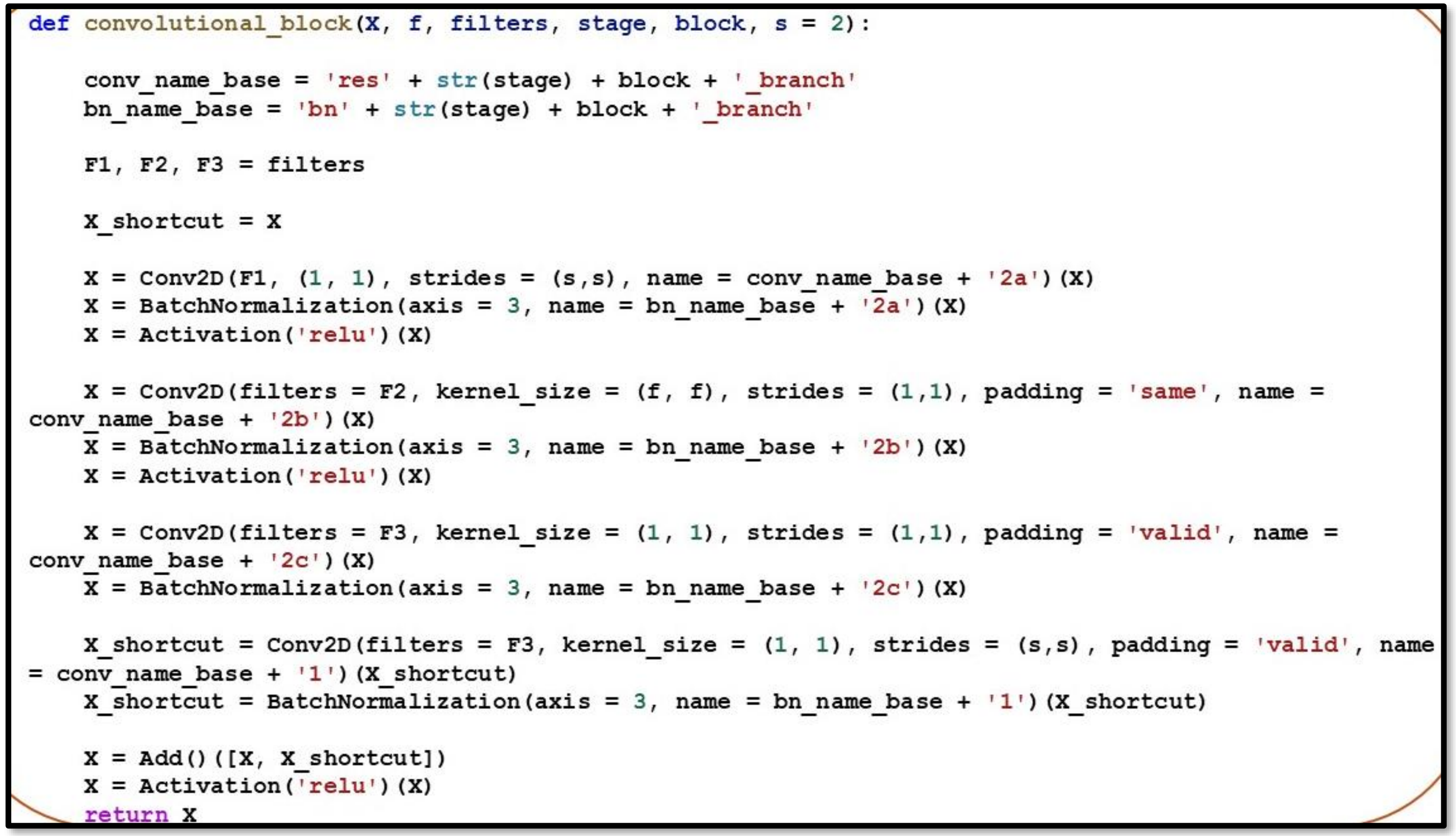


```
def identity_block(X, f, filters, stage, block):

    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    F1, F2, F3 = filters

    X_shortcut = X

    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name =
conv_name_base + '2a')(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name =
conv_name_base + '2b')(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name =
conv_name_base + '2c')(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)


    return X
```

VGG16 and ResNet50 Implementation

ResNet50 Implementation

Now, let's examine the function that applies the convolutional block.
The inputs of this function are consistent with the identity block. The only difference is a new input named 's'.
The main distinction between the convolutional block and the identity block is that the convolutional block applies a convolutional layer to the input to resize it before adding it to the output. The stride of this convolutional layer, which is responsible for changing the size of the input, is determined by the new variable 's'.
We have maintained the same naming format so that by observing the model summary, we can quickly understand everything about the model.



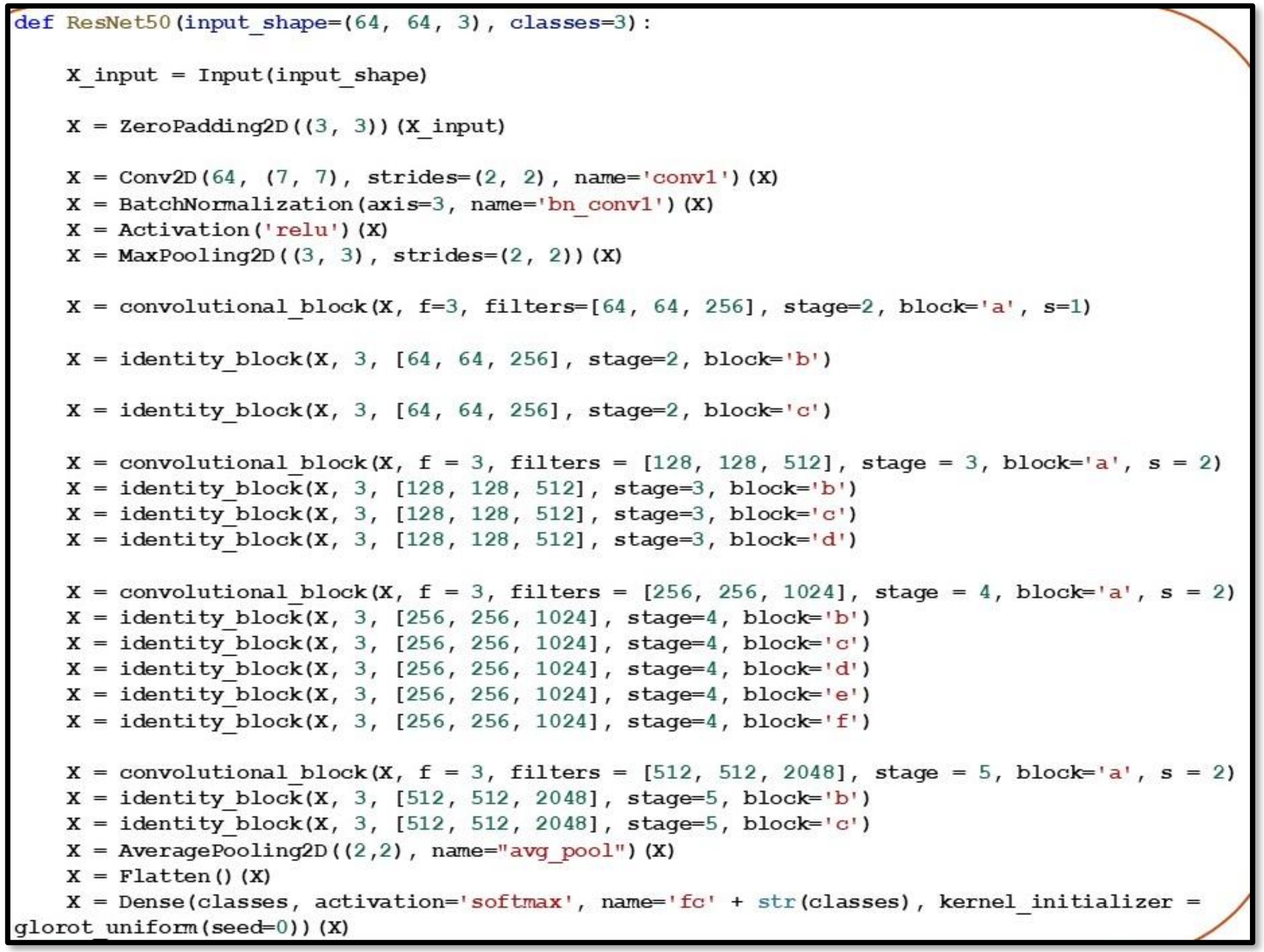
```
def convolutional_block(X, f, filters, stage, block, s = 2):

    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    F1, F2, F3 = filters

    X_shortcut = X

    X = Conv2D(F1, (1, 1), strides = (s,s), name = conv_name_base + '2a')(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name =
conv_name_base + '2b')(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name =
conv_name_base + '2c')(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

    X_shortcut = Conv2D(filters = F3, kernel_size = (1, 1), strides = (s,s), padding = 'valid', name
= conv_name_base + '1')(X_shortcut)
    X_shortcut = BatchNormalization(axis = 3, name = bn_name_base + '1')(X_shortcut)

    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)
    return X
```

VGG16 and ResNet50 Implementation

ResNet50 Implementation

Now that we can create both the convolutional and identity blocks, it is time to define the function responsible for generating the ResNet50 architecture. The input shape and the number of classes are passed to this function. We will create the convolutional blocks and identity blocks one by one using the functions we previously defined. In the end, we will return the model.

Further simplification is possible through the use of for loops. We used this structure to code ResNet50 because this architecture is more complicated than VGG16, and we had to focus on defining the blocks in more detail. Defining ResNet50 using for loops can be a great practice for you.



```
def ResNet50(input_shape=(64, 64, 3), classes=3):

    X_input = Input(input_shape)

    X = ZeroPadding2D((3, 3))(X_input)

    X = Conv2D(64, (7, 7), strides=(2, 2), name='conv1')(X)
    X = BatchNormalization(axis=3, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2))(X)

    X = convolutional_block(X, f=3, filters=[64, 64, 256], stage=2, block='a', s=1)

    X = identity_block(X, 3, [64, 64, 256], stage=2, block='b')

    X = identity_block(X, 3, [64, 64, 256], stage=2, block='c')

    X = convolutional_block(X, f = 3, filters = [128, 128, 512], stage = 3, block='a', s = 2)
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='b')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='c')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='d')

    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], stage = 4, block='a', s = 2)
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='b')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='c')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='d')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='e')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='f')

    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], stage = 5, block='a', s = 2)
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='b')
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='c')
    X = AveragePooling2D((2,2), name="avg_pool")(X)
    X = Flatten()(X)
    X = Dense(classes, activation='softmax', name='fc' + str(classes), kernel_initializer =
glorot_uniform(seed=0))(X)
```

VGG16 and ResNet50 Implementation

Summary

We examined how we can define the VGG16 and ResNet50 models. We started by implementing VGG16. First, we studied how to define a model by looking at its architecture diagram. Then, we discussed the importance of reusability in coding and modified our code to be more reusable. After that, we learned to minimize code duplications, making our code cleaner and easier to read.

We proceeded to implement ResNet50. We focused on the new innovations this model has introduced and learned how to incorporate these changes in our code using auxiliary functions.

To build other complex convolutional neural networks, we must follow the same steps. First, we need to design our network. In this step, we define everything about our model, from the number of convolutional layers to the number of max pooling layers and their arrangement relative to each other.

Once the model is designed, we can start coding it. We learned various methods that facilitate this step. We can use functions or loops to prevent code duplication and ensure reusability. Additionally, we can implement functions that make our code adaptable to different input sizes and varying numbers of classes.