Deep Convolution Neural Networks for Image Classification

Learning Multiple Layers of Features from Tiny Images

Waleed Khalid Mohamed Waleed Alzamil 2002011
Mohammad Saeed Ibrahim Dallash 1900084
Mohammed Khaled Kamal Ellithy 1900568
Faculty of Engineering Ain Shams University

Abstract
Deep learning has revolutionized the field of artificial intelligence, enabling machines to learn from data and make predictions with unprecedented accuracy. Building the Architectures play a crucial role in improving the performance of deep learning models by specify the exact function that maps the inputs to the required output.

The field of deep learning has seen significant advancements in recent years, with different Architectures playing a crucial role in improving the performance of deep learning models.

Contents

0.1 Introduction

0.1.1 Intro

This aim of this Milestone is to train compare the performance of different convolutional neural network architectures, such as LeNet, AlexNet, and VGG, and to analyze the impact of depth on their accuracy. Furthermore, we will demonstrate how to apply transfer learning techniques and use pre-trained models to extract features for further computations.

We choose the famous CIFAR-100 dataset. The CIFAR-100 dataset (Canadian Institute for Advanced Research, 100 classes) is a subset of the Tiny Images dataset and consists of 60000 32x32 color images. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. There are 600 images per class. Each image comes with a "fine" label (the class to which it

belongs) and a "coarse" label (the superclass to which it belongs). There are 500 training images and 100 testing images per class.

The criteria for deciding whether an image belongs to a class were as follows:

1. The class name should be high on the list of likely answers to the question "What is in this picture?"

2. The image should be photo-realistic. Labelers were instructed to reject line drawings.

3. The image should contain only one prominent instance of the object to which the class refers.

4. The object may be partially occluded or seen from an unusual viewpoint as long as its identity is still clear to the labeler.

0.1.2 Problem Statement

Image classification is a fundamental task in computer vision that involves assigning a label to an image based on its content. However, images can be complex and diverse, and require sophisticated models to capture their features and patterns.

Multilayer neural networks are powerful models that can learn from large amounts of data and perform well on various image classification tasks. However, designing and training multilayer neural networks can be challenging, as they involve many hyperparameters and optimization techniques.

Therefore, the aim of this research is to train a multilayer neural network to classify an image into its corresponding category, and to evaluate its performance on different datasets and benchmarks.

Given: (X, Y) where: X is a list of images and Y is the label of these images.

Train a Multilayer Neural Networks to Classify any unseen image x_i to its corresponding label y_i

---

0.2 Traditional Architectures

VERY DEEP CONVOLUTIONAL NETWORKS FOR IMAGE RECOGNITION: Measuring the improvement brought by the increased ConvNet

---

0.2.1 Intro

To measure the improvement brought by the increased ConvNet depth in a fair setting, all our ConvNet layer configurations are designed using the same principles. In this section, we first describe a generic layout of our ConvNet configurations and then detail the specific configurations used in the evaluation. Our design choices are then discussed and compared to the famous architectures in sec 3.

0.2.2 Architectures

Batch Normalization is added after each layer During training, the input to our ConvNets is a fixed-size 32×32 RGB image. The only preprocessing we do is Rescaling the input images from $[0 \rightarrow 255]$ to $[0 \rightarrow 1]$. The image is passed through a stack of convolutional (conv.) layers, where we use filters with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations we also utilise 1×1 convolution filters, which can be seen as a linear transforma-

tion of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1 pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2. A stack of convolutional layers (which has a different depth in different architectures) is followed by three Fully-Connected (FC) layers: the first two have 4096 neurons each, the third performs 100 classification and thus contains 100 neurons (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks. All hidden layers are equipped with rectification non-linearity. Something to mention here we used Batch Normalization after each layer and we used dropout by 0.1 after each block you can see the detailed architectures in Table 1 and also you can find the code in the Appendix and also the GitHub repository you can find the repository for this report here

0.2.3 CONFIGURATIONS

The ConvNet configurations, evaluated in this paper, are outlined in Table 1, one per column. In the following we will refer to the nets by their names (A–E). All configurations follow the generic design presented in Sec 0.2.2, and differ only in the depth: from 11 weight layers in the network A (8 conv. and 3 FC layers) to 19 weight layers in the network E (16 conv. and 3 FC layers). The width of conv. layers (the number of channels) is rather small, starting from 64 in the first layer and then increasing by a factor of 2 after each max-pooling layer, until it reaches 512.

0.2.4 Training

1. A This is the simplest architecture, as can be shown in Table 1.Figures 1& 2 show the accuracy and loss curves for this architecture on the training and validation sets.

Figure 1: The Loss of A

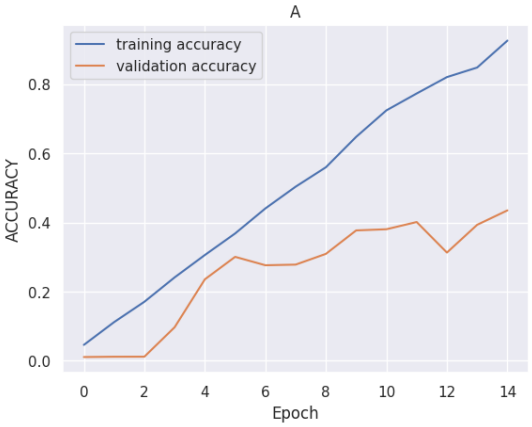

Figure 2: The Accuracy of A

2. B This architecture is similar to A, but with small additions as can be shown in Table 1.Figures 3& 4 show the accuracy and loss curves for this architecture on the training and validation sets.

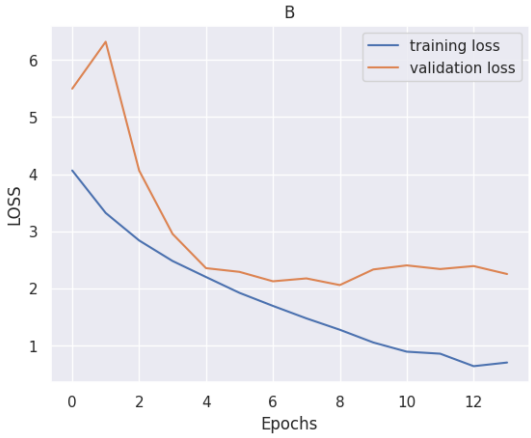

Figure 3: The Loss of B

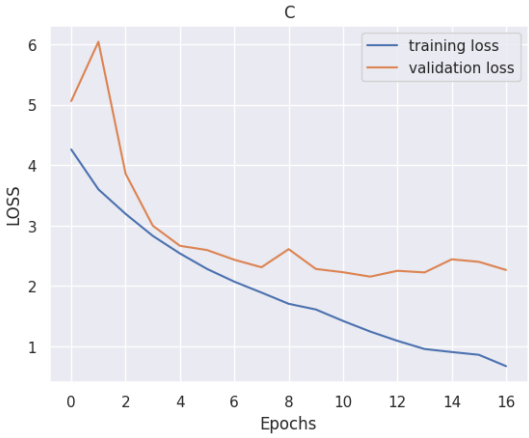

Figure 5: The Loss of C

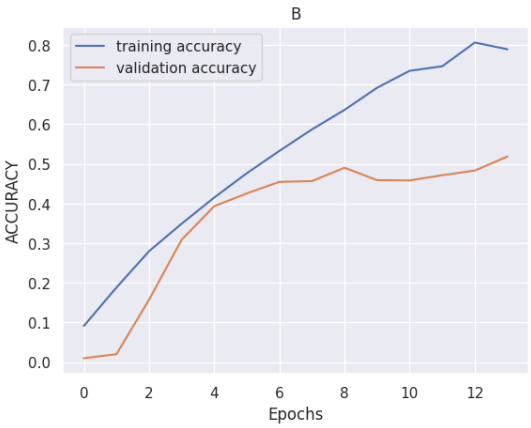

Figure 4: The Accuracy of B

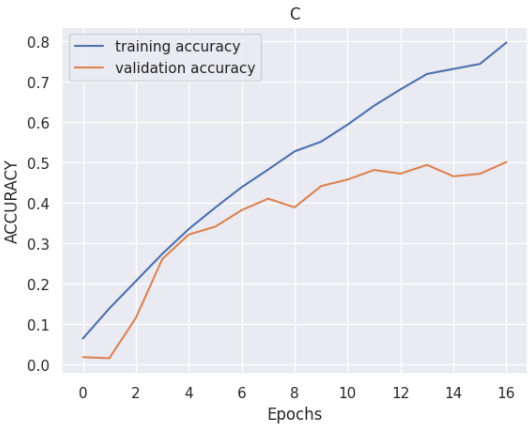

Figure 6: The Accuracy of C

3. C This architecture is similar to B, but with small additions as can be shown in Table 1.Figures 5& 6 show the accuracy and loss curves for this architecture on the training and validation sets.

4. D This architecture is similar to C, but with small additions as can be shown in Table 1.Figures 7& 8 show the accuracy and loss curves for this architecture on the training and validation sets.

Figure 7: The Loss of D

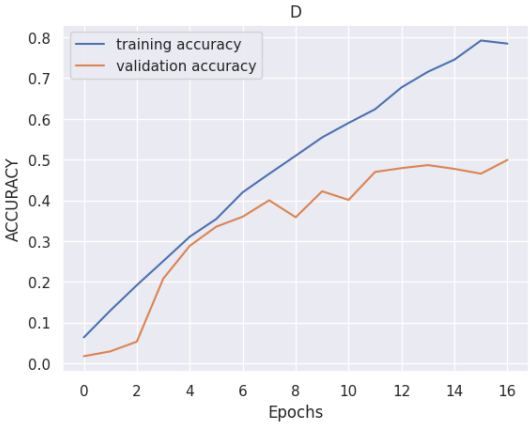

Figure 8: The Accuracy of D

5. E This is the most complex architecture, as can be shown in Table 1.Figures 9& 10 show the accuracy and loss curves for this architecture on the training and validation sets.

Figure 9: The Loss of E

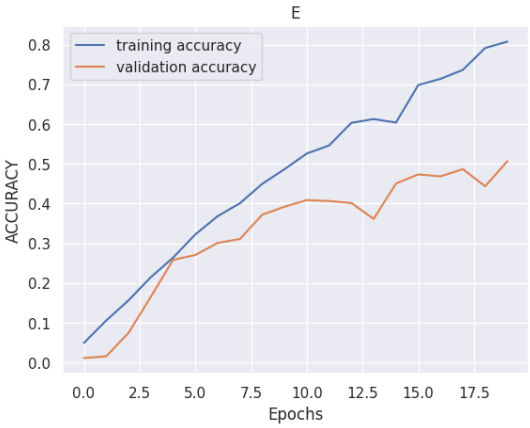

Figure 10: The Accuracy of E

0.3 Famous Architectures

See the performance of well known architectures

0.3.1 LeNet

LeNet is a convolutional neural network (CNN) architecture that was developed by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner in 1998. It was one of the first CNN architectures to achieve state-of-the-art performance on handwritten digit recognition tasks.

The LeNet architecture consists of seven layers, including two convolutional layers, two subsampling layers, and three fully connected layers. The input to the network is a grayscale image of size $32x32$ pixels. The first layer is a convolutional layer with 6 filters of size $5x5$. The output of this layer is passed through a hyperbolic tangent (tanh) activation function and then subsampled using a $2x2$ max pooling operation. The second layer is another convolutional layer with 16 filters of size $5x5$. The output of this layer is

ConvNet Configuration				
A	B	C	D	E
11-weight layers	13-weight layers	16-weight layers	16-weight layers	19-weight layers
input(32×32) RGB image				
Conv3-64	Conv3-64 **Conv3-64**	Conv3-64 Conv3-64	Conv3-64 Conv3-64	Conv3-64 Conv3-64
MaxPooling 2				
Dropout 0.1				
Conv3-128	Conv3-128 **Conv3-128**	Conv3-128 Conv3-128	Conv3-128 Conv3-128	Conv3-128 Conv3-128
MaxPooling 2				
Dropout 0.1				
Conv3-256 Conv3-256	Conv3-256 Conv3-256	Conv3-256 Conv3-256 **Conv1-256**	Conv3-256 Conv3-256 **Conv3-256**	Conv3-256 Conv3-256 Conv3-256 **Conv3-256**
MaxPooling 2				
Dropout 0.1				
Conv3-512 Conv3-512	Conv3-512 Conv3-512	Conv3-512 Conv3-512 **Conv-512**	Conv3-512 Conv3-512 **Conv3-512**	Conv3-512 Conv3-512 Conv3-512 **Conv3-512**
MaxPooling 2				
Dropout 0.1				
Conv3-512 Conv3-512	Conv3-512 Conv3-512	Conv3-512 Conv3-512 **Conv1-512**	Conv3-512 Conv3-512 **Conv3-512**	Conv3-512 Conv3-512 Conv3-512 **Conv3-512**
MaxPooling 2				
Dropout 0.1				
Flatten				
FC-Dense-512				
FC-Dense-128				
FC-Dense-100				
Softmax				
Dropout 0.1				
No of Parameters (in Millions)				
9.575	9.760	10.356	15.075	19.799
Accuracy on unseen data				
%46.86	%52.61	%50.5	%50.94	%49.64

Table 1: What is the effect of deeper networks to the performance

also passed through a tanh activation function and then subsampled using a $2x2$ max pooling operation. The third layer is a fully connected layer with 120 neurons. The output of this layer is passed through a tanh activation function. The fourth layer is another fully connected layer with 84 neurons. The output of this layer is also passed through a tanh activation function.

Here we made some thing different in the last layer for our purpose. The fifth and final layer is a fully connected layer with 100 neurons, corresponding to the 100 possible classes in the CIFAR-100 datasets. The output of this layer is passed through a softmax activation function to obtain the final class probabilities.

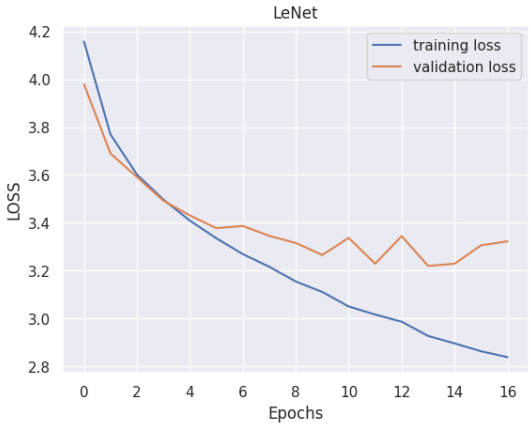

Figure 11: The Loss of LeNet

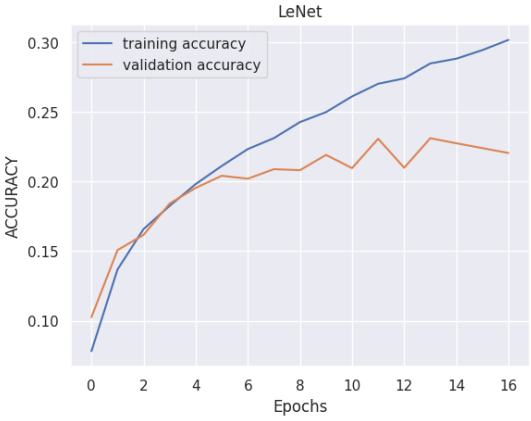

Figure 12: The Accuracy of LeNet

Overall, the LeNet architecture is a relatively simple CNN architecture that was designed for handwritten digit recognition tasks. However, it has since been used as a starting point for many other CNN architectures and has been adapted for a wide range of computer vision tasks.

0.3.2 VGG

The VGG architecture is a convolutional neural network (CNN) architecture that was proposed by the Visual Geometry Group (VGG) at the University of Oxford in 2014. It is a deep neural network that is widely used for image classification tasks.

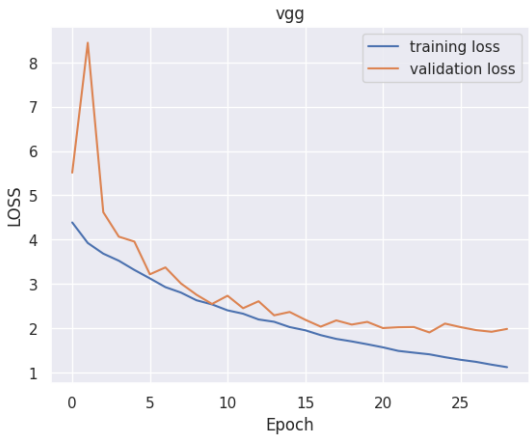

Figure 13: The Loss of VGG-16

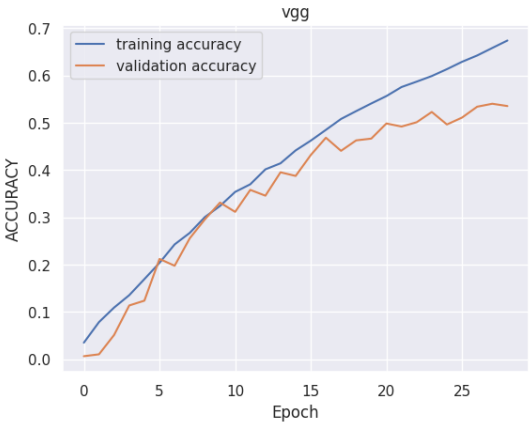

Figure 14: The Accuracy of VGG-16

The VGG architecture consists of a series of convolutional layers, followed by max pooling layers, and then fully connected layers. The convolutional layers are responsible for extracting features from the input image, while the max pooling layers downsample the feature maps to reduce

their spatial dimensions. The fully connected layers are used for classification.

The VGG architecture has several variations, with the most popular being VGG16 and VGG19. VGG16 has 16 layers, while VGG19 has 19 layers. Both architectures have the same basic structure, with the only difference being the number of convolutional layers.

In VGG16, the input image is passed through 13 convolutional layers, followed by 5 max pooling layers, and then 3 fully connected layers. The convolutional layers have a fixed filter size of 3x3, and the max pooling layers have a fixed pool size of 2x2. The fully connected layers have 4096 neurons each, and the output layer has 1000 neurons, corresponding to the 1000 classes in the ImageNet dataset.

Figure 15: The Loss of VGG-16

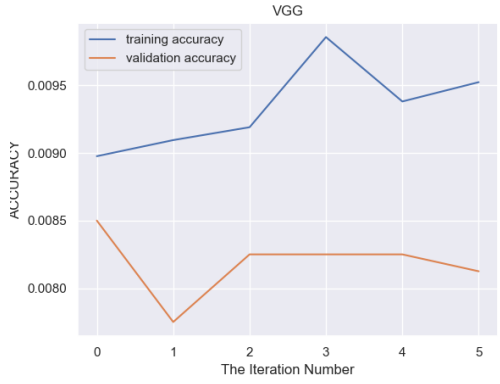

Figure 16: The Accuracy of VGG-16

Overall, the VGG architecture is known for its simplicity and effectiveness, and has been used as a baseline for many state-of-the-art CNN architectures.

0.3.3 Alex

AlexNet is a convolutional neural network (CNN) architecture that was proposed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012. It was the winning entry in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012, and is considered to be one of the pioneering CNN architectures that popularized deep learning.

The AlexNet architecture consists of 5 convolutional layers, followed by max pooling layers, and then 3 fully connected layers. The convolutional layers are responsible for extracting features from the input image, while the max pooling layers downsample the feature maps to reduce their spatial dimensions. The fully connected layers are used for classification.

The first convolutional layer in AlexNet has a filter size of 11x11, which is larger than the filter sizes used in most modern CNN architectures. This was done to capture larger-scale features in the input image. The subsequent convolutional layers have a filter size of 3x3, which is more common in modern CNN architectures.

AlexNet also introduced the use of rectified linear units (ReLU) as activation functions, which are more computationally efficient than traditional activation functions like sigmoid and tanh. Additionally, AlexNet used dropout regularization to prevent overfitting.

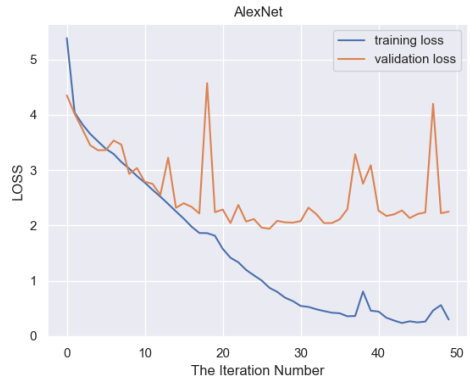

Figure 17: The Loss of AlexNet

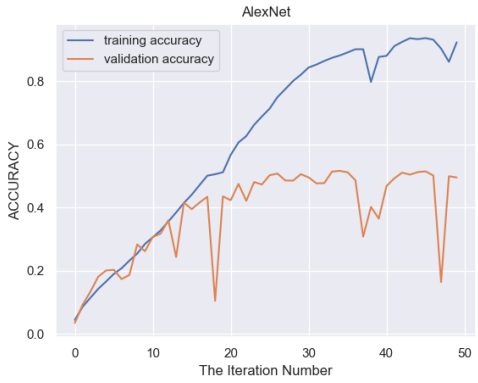

Figure 18: The Accuracy of AlexNet

In terms of performance, AlexNet achieved a top-5 error rate of 15.3% in the ILSVRC 2012 competition, which was a significant improvement over the previous state-of-the-art. Its success paved the way for the development of deeper and more complex CNN architectures.

Overall, AlexNet is a landmark CNN architecture that played a key role in the development of deep learning and computer vision.

0.3.4 ResNet

ResNet (short for Residual Network) is a deep neural network architecture that was introduced in 2015 by Kaiming He et al. It is a type of convolutional neural network (CNN) that is designed to address the problem of vanishing gradients in very deep networks. The main idea behind ResNet is to use skip connections, also known as residual connections, to allow the network to learn residual functions. A residual function is the difference between the input and output of a layer, and it represents the part of the output that cannot be learned by the layer itself. By using skip connections to add the residual function to the output of a layer, ResNet allows the network to learn the residual function directly, rather than trying to learn the entire output from scratch. The ResNet architecture consists of a series of convolutional layers, followed by a global average pooling layer and a fully connected layer. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. The skip connections are added between pairs of convolutional layers, and they bypass the batch normalization and ReLU layers.

Figure 19: The Loss of ResNet

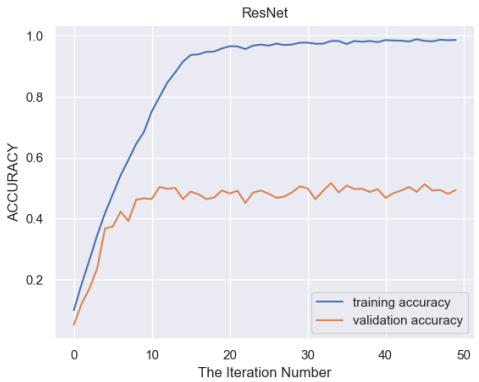

Figure 20: The Accuracy of ResNet

There are several variants of the ResNet architecture, including ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-

152, which differ in the number of layers and the number of filters in each layer. The deeper variants of ResNet have been shown to achieve state-of-the-art performance on a wide range of computer vision tasks, including image classification, object detection, and semantic segmentation.

Overall, ResNet is a powerful and widely used architecture that has helped to advance the state of the art in deep learning. Its use of skip connections to learn residual functions has proven to be an effective way to train very deep neural networks, and it has inspired many other architectures that build on this idea.

0.4 Conclusions

One thing to mention here is that ResNet is the most stable architecture among others during the training process. AlexNet is a great architecture and shows great performance on such difficult problems. Basic Architectures did a great job and they were really close to the others.

It is worth mentioning that the original Vgg-16 architecture was designed and trained on the ImageNet dataset, which contains approximately 1.2 million images with a resolution of 224×224 pixels distributed across more than $1,000$ classes or categories.

In contrast, Cifar-100 is much smaller. of only 60 thousand images and also the images with a resolution of 32×32 pixels.

Therefore, when using such a complex model that was designed to work with larger datasets of more pixels without fine-tuning (except the upscaling of the image in the first layer to match the architecture), we will face some drawbacks as follows:

1. Unnecessarily decreased computational efficiency: Resizing the image increases the computational requirements, slowing down the training without any cause.

2. Increased memory usage: Resizing the image to a larger dimension requires more memory for storage and processing, which limited us when training the model to use only a batch size of 32. Although we could surely benefit from using larger batch sizes in this problem as the number of classes is relatively high, using larger batch sizes results in better generalization as the network encounters a more diverse range of classes during training.

3. Causing overfitting: as mentioned earlier, the vgg-16 original architecture is a complex model that is designed to work with relatively large data sets; training it on a relatively small data set of upscaled images with not-so-many useful features, as most of them are the result of interpolating neighbouring pixels with each other. As shown in Figures 15&16 the model has stopped training completely after a few epochs with very poor performance.

There are the modifications to Vgg-16 that worked well with our problem:

1. Decreasing the input size of the first layer to 32×32 to match the input image

2. Adding batch-normalization and drop-out layers between convolution layers to reduce the overfitting

3. Decreasing the size of the dense layers at the bottom layers of the networks from 4096 to 128 to suit more the new relatively small input size of 32×32.

As shown in Figures 17&18 that mode has achieved much better results and consumed much less computation power due the great decrease of the model parameters.

0.5 Appendix

0.5.1 Architectures mentioned in section 2

1. A

```
import tensorflow as tf

def A(input_shape, num_classes):
  model = tf.keras.Sequential()

  model.add(tf.keras.layers.Input(input_shape))
  model.add(tf.keras.layers.Rescaling(1./255))

  model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))

  model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))

  model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))

  model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
```

```
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(units = 512, activation = "relu"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.2))
    model.add(tf.keras.layers.Dense(units = 128, activation = "relu"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.2))

    model.add(tf.keras.layers.Dense(units = 100, activation = "softmax"))

    model.compile(optimizer = "adam", loss = "categorical_crossentropy",
    metrics=['accuracy'])

    return model
```

2. B

```
    import tensorflow as tf

def B(input_shape, num_classes):
    model = tf.keras.Sequential()

    model.add(tf.keras.layers.Input(input_shape))
    model.add(tf.keras.layers.Rescaling(1./255))

    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
    activation = "relu", padding = "same"))
```

```
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(units = 512, activation = "relu"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.2))
    model.add(tf.keras.layers.Dense(units = 128, activation = "relu"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.2))

    model.add(tf.keras.layers.Dense(units = 100, activation = "softmax"))

    model.compile(optimizer = "adam", loss = "categorical_crossentropy",
    metrics=['accuracy'])

    return model
```

3. C

```
    import tensorflow as tf

def B(input_shape, num_classes):
  model = tf.keras.Sequential()

  model.add(tf.keras.layers.Input(input_shape))
  model.add(tf.keras.layers.Rescaling(1./255))

  model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))

  model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))

  model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (1,1),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))

  model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (1,1),
  activation = "relu", padding = "same"))
  model.add(tf.keras.layers.BatchNormalization())
  model.add(tf.keras.layers.MaxPooling2D((2,2)))
  model.add(tf.keras.layers.Dropout(0.1))
```

```
    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (1,1),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(units = 512, activation = "relu"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.2))
    model.add(tf.keras.layers.Dense(units = 128, activation = "relu"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.2))

    model.add(tf.keras.layers.Dense(units = 100, activation = "softmax"))

    model.compile(optimizer = "adam", loss = "categorical_crossentropy",
    metrics=['accuracy'])

    return model
```

4. D

```
    import tensorflow as tf

def B(input_shape, num_classes):
    model = tf.keras.Sequential()

    model.add(tf.keras.layers.Input(input_shape))
    model.add(tf.keras.layers.Rescaling(1./255))

    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
```

```
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D((2,2)))
model.add(tf.keras.layers.Dropout(0.1))

model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D((2,2)))
model.add(tf.keras.layers.Dropout(0.1))

model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D((2,2)))
model.add(tf.keras.layers.Dropout(0.1))

model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D((2,2)))
model.add(tf.keras.layers.Dropout(0.1))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(units = 512, activation = "relu"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(units = 128, activation = "relu"))
model.add(tf.keras.layers.BatchNormalization())
```

```
    model.add(tf.keras.layers.Dropout(0.2))

    model.add(tf.keras.layers.Dense(units = 100, activation = "softmax"))

    model.compile(optimizer = "adam", loss = "categorical_crossentropy",
    metrics=['accuracy'])

    return model
```

5. E

```
import tensorflow as tf

def B(input_shape, num_classes):
    model = tf.keras.Sequential()

    model.add(tf.keras.layers.Input(input_shape))
    model.add(tf.keras.layers.Rescaling(1./255))

    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.Conv2D(filters = 128, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))

    model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Conv2D(filters = 256, kernel_size = (3,3),
    activation = "relu", padding = "same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Dropout(0.1))
```

```
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D((2,2)))
model.add(tf.keras.layers.Dropout(0.1))

model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Conv2D(filters = 512, kernel_size = (3,3),
 activation = "relu", padding = "same"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D((2,2)))
model.add(tf.keras.layers.Dropout(0.1))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(units = 512, activation = "relu"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(units = 128, activation = "relu"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.Dense(units = 100, activation = "softmax"))

model.compile(optimizer = "adam", loss = "categorical_crossentropy",
metrics=['accuracy'])

return model
```

0.5.2 Architectures mentioned in the section 3

1. LeNet

```
def leNet(input_shape, num_classes):
    model=tf.keras.models.Sequential()

    model.add(tf.keras.layers.Resizing(227, 227,
    interpolation="gaussian", input_shape=input_shape))

    model.add (tf.keras.layers.Conv2D(filters=6, kernel_size=(5,5),
    strides=(1,1), activation='tanh'))
    model.add(tf.keras.layers.AveragePooling2D(pool_size=(2,2),
    strides=(2,2)))

    model.add (tf.keras.layers.Conv2D(filters=16, kernel_size=(5,5),
    strides=(1,1), activation='tanh'))
    model.add(tf.keras.layers.AveragePooling2D(pool_size=(2,2),
    strides=(2,2)))

    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(120,activation='tanh'))
    model.add(tf.keras.layers.Dense(84,activation='tanh'))
    model.add(tf.keras.layers.Dense(num_classes,activation='softmax'))

    model.compile(optimizer = "adam", loss = "categorical_crossentropy",
    metrics=['accuracy'])

    return model
```

2. VGG

```
def Vgg16(input_shape, num_classes):
    model=tf.keras.models.Sequential()

    model.add(Resizing(224, 224, interpolation="bilinear",
    input_shape=input_shape))

    model.add (Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    model.add (Conv2D(filters=128, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=128, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
```

```
    model.add (Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    model.add (Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    model.add (Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add (Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
    activation='relu',padding="same" ) )
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    model.add(Flatten())

    model.add(Dense(4096,activation='relu'))
    model.add(Dense(4096,activation='relu'))
    model.add(Dense(num_classes,activation='softmax'))

    model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
    return model
```

3. VGG-modified

```
def Vgg16Modified(input_shape, num_classes):
    model = tf.keras.models.Sequential()

    model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),
    activation='relu', padding="same"input_shape = input_shape) )
    model.add(BatchNormalization())
    model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),
    activation='relu', padding="same"))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1),
```

```
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.5))

model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.5))

model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.5))

model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.5))

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dense(128, activation='relu'))
```

```
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
    return model
```

4. AlexNet

```
def AlexNet(input_shape, num_classes):
    model=tf.keras.models.Sequential()
    model.add(layers.experimental.preprocessing.Resizing(227, 227,
    interpolation="bilinear", input_shape=input_shape))

    model.add(layers.Conv2D(filters=96, kernel_size=(11,11),
    strides=(4,4),activation='relu',input_shape=(227, 227,3),padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D(pool_size=(3,3),strides=(2,2)))
    model.add(layers.Conv2D(filters=256, kernel_size=(5,5),
    padding="same",activation="relu",strides=(1,1)))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D(pool_size=(3,3),strides=(2,2)))
    model.add(layers.Conv2D(kernel_size=(3,3),
    filters=384,activation='relu',padding='same',strides=(1,1)))
    model.add(layers.BatchNormalization())
    model.add(layers.Conv2D(filters=384,kernel_size=(3,3),
    padding='same',activation='relu',strides=(1,1)))
    model.add(layers.BatchNormalization())
    model.add(layers.Conv2D(filters=256, kernel_size=(3,3),
    padding='same',activation='relu',strides=(1,1)))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D(pool_size=(3,3),strides=(2,2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(4096,activation='relu'))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(4096,activation='relu'))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(num_classes,activation='softmax'))

    model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
    return model
```

5. ResNet

```
def resnet50(input_shape, num_classes):
    # Input tensor
    inputs = Input(shape=input_shape)
```

```
    # Stage 1

    x=Resizing(224, 224, interpolation="bilinear",
    input_shape=input_shape)(inputs)
    x = Conv2D(64, kernel_size=(7, 7), strides=(2, 2),
    padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2),
    padding='same')(x)

    # Stage 2
    x = convolutional_block(x, [64, 64, 256], strides=(1, 1))
    x = identity_block(x, [64, 64, 256])
    # x = identity_block(x, [64, 64, 256])

    # Stage 3
    x = convolutional_block(x, [128, 128, 512])
    x = identity_block(x, [128, 128, 512])
    # x = identity_block(x, [128, 128, 512])
    # x = identity_block(x, [128, 128, 512])

    # Stage 4
    x = convolutional_block(x, [256, 256, 1024])
    x = identity_block(x, [256, 256, 1024])
    # x = identity_block(x, [256, 256, 1024])
    # x = identity_block(x, [256, 256, 1024])
    # x = identity_block(x, [256, 256, 1024])
    # x = identity_block(x, [256, 256, 1024])

    # Stage 5
    x = convolutional_block(x, [512, 512, 2048])
    x = identity_block(x, [512, 512, 2048])
    # x = identity_block(x, [512, 512, 2048])

    # Output layer
    x = GlobalAveragePooling2D()(x)
    x = Dense(num_classes, activation='softmax')(x)

    # Create model
    model = tf.keras.models.Model(inputs=inputs, outputs=x)

    return model


def identity_block(input_tensor, filters):
    filters1, filters2, filters3 = filters
```

```
    x = Conv2D(filters1, kernel_size=(1, 1), strides=(1, 1),
    padding='valid')(input_tensor)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(filters2, kernel_size=(3, 3), strides=(1, 1),
    padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(filters3, kernel_size=(1, 1), strides=(1, 1),
    padding='valid')(x)
    x = BatchNormalization()(x)

    x = Add()([x, input_tensor])
    x = Activation('relu')(x)

    return x


def convolutional_block(input_tensor, filters, strides=(2, 2)):
    filters1, filters2, filters3 = filters

    x = Conv2D(filters1, kernel_size=(1, 1), strides=strides,
    padding='valid')(input_tensor)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(filters2, kernel_size=(3, 3), strides=(1, 1),
    padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(filters3, kernel_size=(1, 1), strides=(1, 1),
    padding='valid')(x)
    x = BatchNormalization()(x)

    shortcut = Conv2D(filters3, kernel_size=(1, 1), strides=strides,
    padding='valid')(input_tensor)
    shortcut = BatchNormalization()(shortcut)


    x = Add()([x, shortcut])
    x = Activation('relu')(x)

    return x
```