Chapter 7

Teaching Machines to See Better: Improving CNNs and Making Them Confess

In the previous chapter, a state-of-the-art image classification model, Inception Net v1, was developed and trained on the tiny-imagenet-200 data set. The architecture demonstrated several important design principles, including the use of Inception blocks to aggregate convolutional kernels at multiple scales, the use of 1×1 convolutions to control dimensionality, and the inclusion of auxiliary classification layers to stabilize gradient flow in deep networks. Despite these architectural strengths, the trained model exhibited severe overfitting, achieving approximately 94% training accuracy while reaching only around 30% accuracy on validation and test data.

This chapter focuses on improving that suboptimal model by systematically reducing overfitting and increasing generalization performance. The goal is to raise validation and test accuracy to approximately 80%, corresponding to correct classification of roughly 160 out of 200 object classes. In addition to performance improvements, this chapter introduces techniques that allow inspection and interpretation of model decisions, enabling deeper understanding of what the model attends to when making predictions.

The chapter presents a practical journey of transforming an underperforming deep learning model into a significantly stronger one. The workflow largely mirrors the steps taken previously, with the addition of model explainability. By visualizing which regions of an image influence predictions, it becomes possible to build trust in the model and identify weaknesses that can be corrected. The discussion spans data augmentation, architectural improvements to Inception Net, transfer learning with pretrained models, and modern explainability techniques for image classifiers.

7.1 Techniques for Reducing Overfitting

The objective is to develop an intelligent shopping assistant that relies on robust image classification. The tiny-imagenet-200 data set is used for this purpose, consisting of labeled images divided into training and testing subsets. The training subset is further split into training and validation data, using 90% for training and 10% for validation.

Although an Inception Net–based model was successfully implemented, it overfits heavily. Overfitting leads to models that perform exceptionally well on training data while failing to generalize to unseen data. Several well-known strategies exist to combat this issue, including data augmentation, dropout, and early stopping. This section focuses on applying these techniques using the Keras API.

Reducing overfitting requires a holistic examination of the entire machine learning pipeline, including the data input, model architecture, and training process. Each of these components is analyzed and refined to introduce safeguards against overfitting.

7.1.1 Image Data Augmentation with Keras

The first strategy employed to reduce overfitting is data augmentation. Data augmentation increases the effective size of the training data set without requiring additional labeled data. In image classification tasks, this is achieved by generating transformed variants of existing images, such as rotations or brightness adjustments, while preserving their original labels.

Figure 7.1
Difference between training data and validation data after the augmentation step.
This figure illustrates how various transformations are applied only to the training data, while the validation data remains unchanged.

By increasing the diversity of training samples, data augmentation improves the generalizability of deep learning models and reduces their tendency to memorize training examples. Common augmentation techniques include random brightness and contrast adjustments, zooming, rotations, and translations.

Keras provides extensive support for image augmentation through the ImageDataGenerator class. A wide range of augmentation parameters can be specified directly when defining the generator.

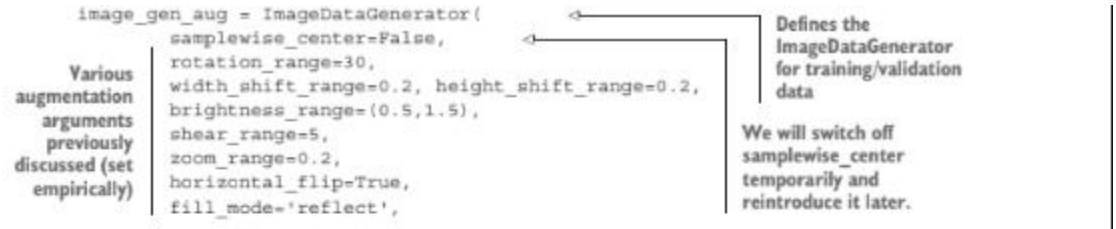
Figure 7.2 Effects of different augmentation parameters and their values of the `ImageDataGenerator`

Figure 7.2
Effects of different augmentation parameters and their values of the ImageDataGenerator.

The augmentation parameters control how images are transformed during training. These include centering and normalization options, rotation bounds, horizontal and vertical shifts, brightness adjustments, shearing, zooming, flipping, and fill modes that define how empty pixels created by transformations are handled. Additional preprocessing steps can be introduced using a custom preprocessing function.

For this project, two image data generators are defined: one with augmentation for training data and one without augmentation for validation and testing data. The selected augmentations include random rotations, translations along width and height, brightness adjustments, shearing, zooming, horizontal flips, gamma correction implemented as a custom transformation, and random occlusions applied to small image patches.

Listing 7.1 Defining the ImageDataGenerator with validation split



The augmentation parameters were chosen empirically. Unlike earlier implementations, samplewise_center is set to False because normalization will be applied later using a custom preprocessing function.

Training and validation data generators are then created using the same augmented generator, relying on the validation_split and subset arguments to separate the data.

Listing 7.2 Defining the data generators for training, validation, and testing sets

```
partial_flow_func = partial(
        image_gen_aug.flow_from_directory,
        directory=os.path.join('data','tiny-imagenet-200', 'train'),
        target_size=target_size, classes=None,
        class_mode='categorical', batch_size=batch_size,
        shuffle=True, seed=random_seed
)

train_gen = partial_flow_func(subset='training')

valid_gen = partial_flow_func(subset='validation')

test_df = get_test_labels_df(
        os.path.join('data','tiny-imagenet-200',  'val',
➥ 'val_annotations.txt')
)
test_gen = image_gen.flow_from_dataframe(
        test_df, directory=os.path.join('data','tiny-imagenet-200',  'val',
➥ 'images'),
        target_size=target_size, classes=None,
        class_mode='categorical', batch_size=batch_size, shuffle=False
)
```

Define a partial function that has all the arguments fixed except for the subset argument.

Get the training data subset.

Get the validation data subset.

Read in the test labels stored in a txt file.

Define the test data generator.

To refresh our memory, the `flow_from_directory(…)` has the following function signature:

```
image_gen.flow_from_directory (
    directory=<directory where the images are>,
    target_size=<height and width or target image>,
    classes=None,
    class_mode=<type of targets generated such as one hot encoded, sparse, etc.>,
    batch_size=<size of a single batch>,
```



```
    shuffle=<whether to shuffle data or not>,
    seed=<random seed to be used in shuffling>,
    subset=<set to training or validation>
)
```

The training and validation generators share the same configuration except for the subset parameter. Augmentation is applied only to the training subset, ensuring that the validation data remains fixed across epochs. A separate ImageDataGenerator without augmentation is used for the test set.

Because Inception Net v1 produces three outputs, the generator output must be adapted accordingly. This is achieved by wrapping the Keras generator with a custom Python generator.

Listing 7.3 Defining the data generator with several modifications

```
def data_gen_augmented_inceptionnet_v1(gen, random_gamma=False,
➥ random_occlude=False):
    for x,y in gen:
        if random_gamma:
            # Gamma correction
            # Doing this in the image process fn doesn't help improve
➥ performance
            rand_gamma = np.random.uniform(0.9, 1.08, (x.shape[0],1,1,1))
            x = x**rand_gamma

        if random_occlude:
            # Randomly occluding sections in the image
            occ_size = 10
            occ_h, occ_w = np.random.randint(0, x.shape[1]-occ_size),
➥ np.random.randint(0, x.shape[2]-occ_size)
            x[:,occ_h:occ_h+occ_size,occ_w:occ_w+occ_size,:] =
➥ np.random.choice([0.,128.,255.])
```

Check if the Gamma correction augmentation is needed.

Define a new function that introduces two new augmentation techniques and modifies the format of the final output.

Perform Gamma correction-related augmentation.

Check if random occlusion augmentation is needed.

Defines the starting x/y pixels randomly for occlusion

Apply a white/gray/black color randomly to the occlusion.


```
    # Image centering
    x -= np.mean(x, axis=(1,2,3), keepdims=True)

    yield x,(y,y,y)

train_gen_aux = data_gen_augmented_inceptionnet_v1(
    train_gen, random_gamma=True, random_occlude=True
)
valid_gen_aux = data_gen_augmented_inceptionnet_v1(valid_gen)
test_gen_aux = data_gen_augmented_inceptionnet_v1(test_gen)
```

Makes sure we replicate the target (y) three times

Perform the sample-wise centering that was switched off earlier.

Training data is augmented with random gamma correction and occlusions.

Validation/testing sets are not augmented.

Defining the data generator with several modifications.

The modified generator returns a single input image and three identical target outputs. In addition to reshaping the output format, this generator applies two custom augmentation techniques: gamma correction, which randomly adjusts pixel intensities, and random occlusion, which masks a small region of the image with randomly selected pixel values. Image centering is also applied manually by subtracting the mean pixel value from each image.

With these generators defined, the final step is to visually inspect samples produced by the training and validation generators to ensure that augmentation behaves as expected and does not corrupt the data.

Figure 7.3
Difference between training data and validation data after the augmentation step.
This figure confirms that augmentation is applied exclusively to the training data and not to the validation data.

At this point, data augmentation has been successfully integrated into the training pipeline. The next section introduces another regularization technique—dropout—to further reduce overfitting.

7.1.2 Dropout: Randomly Switching Off Parts of the Network

Dropout is introduced as a regularization technique that randomly disables neurons during training to prevent co-adaptation of features. This forces the network to learn redundant representations and improves generalization.

Dropout works by applying a random binary mask to neuron outputs during training. To maintain consistent output magnitude, active neurons are scaled accordingly. During testing, all neurons are active.

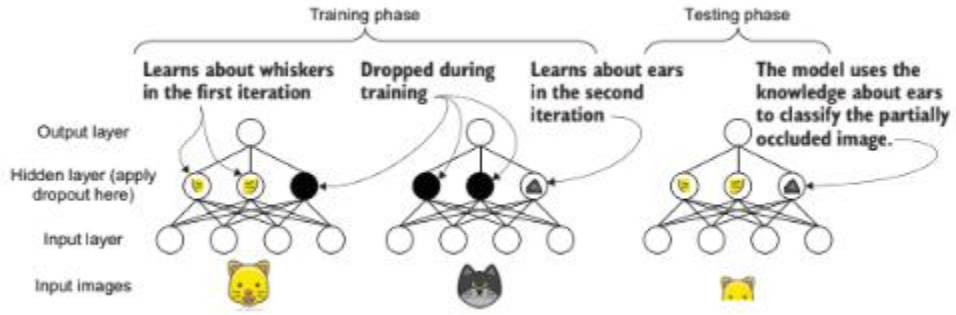


Figure 7.4 How dropout might change the network when learning to classify cat images

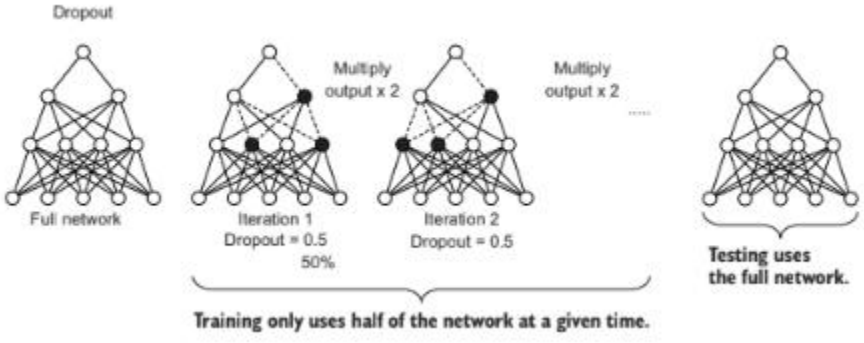


Figure 7.5 A computational perspective on how dropout works. If dropout is set to 50%, then half the nodes in every layer (except for the last layer) will be turned off. But at testing time, all the nodes are switched on.

In Inception Net v1, dropout is applied only to fully connected layers and the final average pooling layer:

- **70% dropout** in auxiliary classifier dense layers
- **40% dropout** after the final average pooling layer

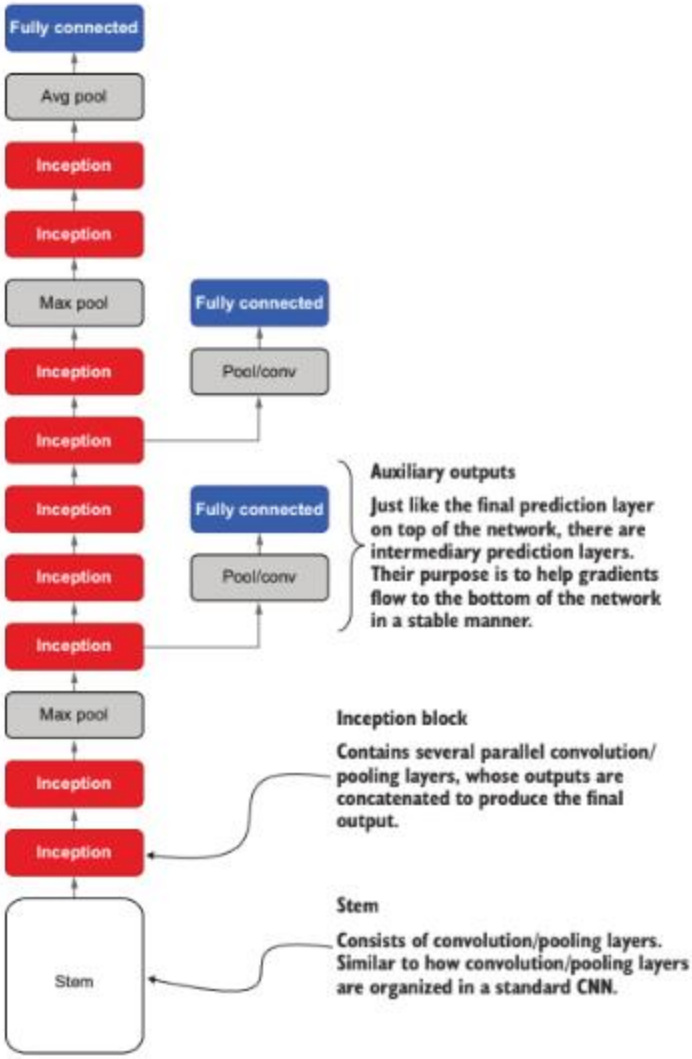


Figure 7.6 Abstract architecture of Inception net v1. Inception net starts with a stem, which is an ordinary sequence of convolution/pooling layers that you will find in a typical CNN. Then Inception net introduces a new component known as Inception block. Finally, Inception net also makes use of auxiliary output layers.

Listing 7.4 Modifying the auxiliary output of Inception net

```
def aux_out(inp,name=None):
    avgpool1 = AvgPool2D((5,5), strides=(3,3), padding='valid')(inp)
    conv1 = Conv2D(128, (1,1), activation='relu', padding='same')(avgpool1)
    flat = Flatten()(conv1)
    dense1 = Dense(1024, activation='relu')(flat)
    dense1 = Dropout(0.7)(dense1)
    aux_out = Dense(200, activation='softmax', name=name)(dense1)
    return aux_out
```

Applying a dropout layer with 70% dropout

A key note is that dropout rates must be chosen carefully. Excessive dropout causes underfitting, while insufficient dropout fails to prevent overfitting.

A short discussion highlights ongoing debate about applying dropout in convolutional layers, with differing conclusions across research literature.

7.1.3 Early Stopping: Halting Training to Prevent Overfitting

Early stopping is introduced as a technique that terminates training when validation performance stops improving. This prevents the model from memorizing training data beyond the optimal point.

Training accuracy often continues to increase even as validation accuracy plateaus or declines, signaling overfitting.

Figure 7.7 An illustration of overfitting. At the start, as the number of training iterations increases, both training and validation accuracies increase. But after a certain point, the validation accuracy plateaus and starts to go down, while the training accuracy keeps going up. This behavior is known as overfitting and should be avoided.Early stopping follows a simple workflow:

1. Train for one epoch
2. Evaluate validation metric
3. Continue if performance improves and epoch limit is not reached
4. Stop otherwise

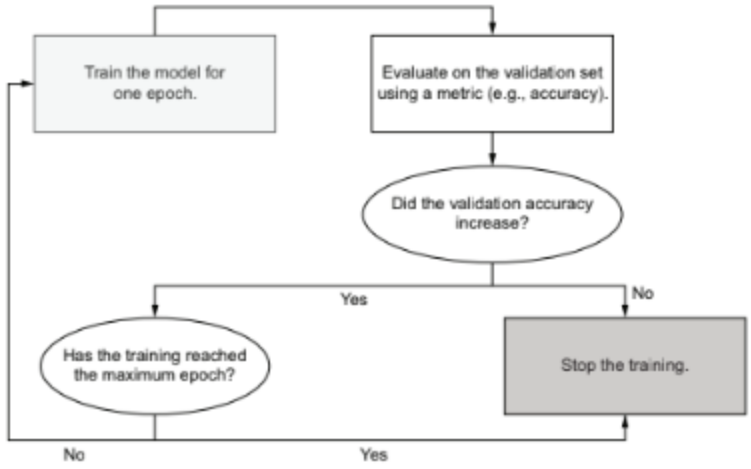


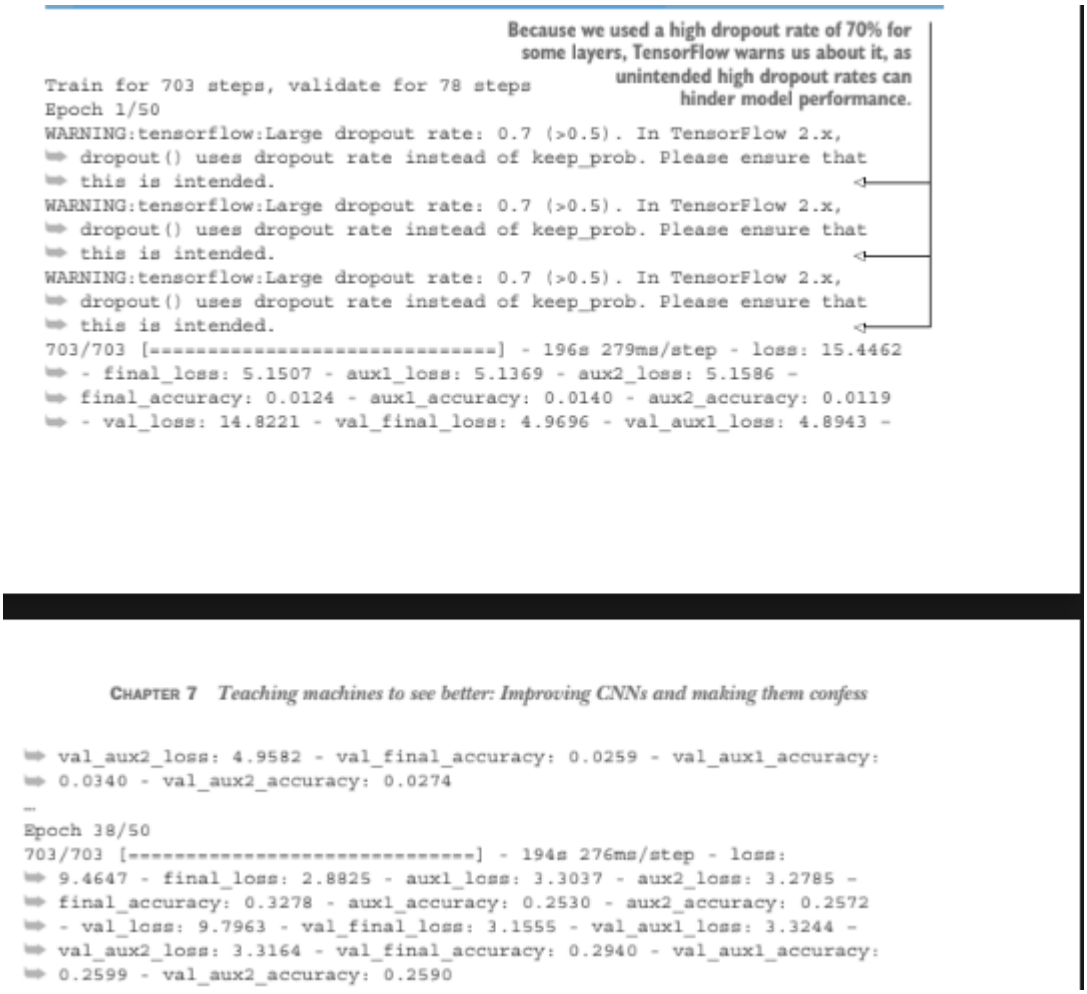
Figure 7.8 The workflow followed during early stopping. First the model is trained for one epoch. Then, the validation accuracy is measured. If the validation accuracy has increased and the training hasn't reached maximum epoch, training is continued. Otherwise, training is halted.

Keras provides the EarlyStopping callback, which monitors a specified metric (e.g., val_loss) and halts training based on defined conditions such as patience and min_delta.

Training is configured with:

- Maximum of 50 epochs
- Early stopping patience of 5 epochs
- CSV logging for performance tracking

Listing 7.5 Training logs provided during training the model




```
➥ val_aux2_loss: 4.9582 - val_final_accuracy: 0.0259 - val_aux1_accuracy:
➥ 0.0340 - val_aux2_accuracy: 0.0274
…
Epoch 38/50
703/703 [==========] - 194s 276ms/step - loss:
➥ 9.4647 - final_loss: 2.8825 - aux1_loss: 3.3037 - aux2_loss: 3.2785 -
➥ final_accuracy: 0.3278 - aux1_accuracy: 0.2530 - aux2_accuracy: 0.2572
➥ - val_loss: 9.7963 - val_final_loss: 3.1555 - val_aux1_loss: 3.3244 -
➥ val_aux2_loss: 3.3164 - val_final_accuracy: 0.2940 - val_aux1_accuracy:
➥ 0.2599 - val_aux2_accuracy: 0.2590
```

The training process terminates at epoch 38, indicating successful early stopping. Training and validation accuracies remain close (~30%), demonstrating reduced overfitting, although overall accuracy remains low.

Exercises (Summary)

Exercise 1:
To reduce underfitting, the dropout rate should be decreased. Among the options (20%, 50%, 80%), **20% dropout** is most appropriate.

Exercise 2:
An early stopping callback should be configured to stop training if val_loss does not improve by at least 0.01 after five epochs using tf.keras.callbacks.EarlyStopping.

7.2 Toward Minimalism: Minception Instead of Inception

After significantly reducing overfitting using augmentation, dropout, and early stopping, the model still fails to achieve satisfactory test accuracy. To address this, a new architectural perspective is introduced through a modified network inspired by **Inception-ResNet v2**, referred to as **Minception**. This model is designed to improve training stability and performance by incorporating **batch normalization** and **residual connections**, two techniques proven effective in very deep networks.

Minception retains the general philosophy of Inception architectures but differs from Inception Net v1 in several key ways. It removes auxiliary classifiers, relying instead on architectural improvements for stability. It also introduces **two different types of Inception-ResNet blocks** rather than repeating a single block design throughout the network. These changes aim to simplify the architecture while preserving representational power.

7.2.1 Implementing the Stem

The stem of Minception replaces the relatively simple stem of Inception Net v1 with a more complex structure that includes **parallel convolutional streams**. A major architectural change is the explicit separation of convolution, batch normalization, and nonlinear activation layers. This separation is necessary because batch normalization must be inserted **between** the convolution output and the activation function.

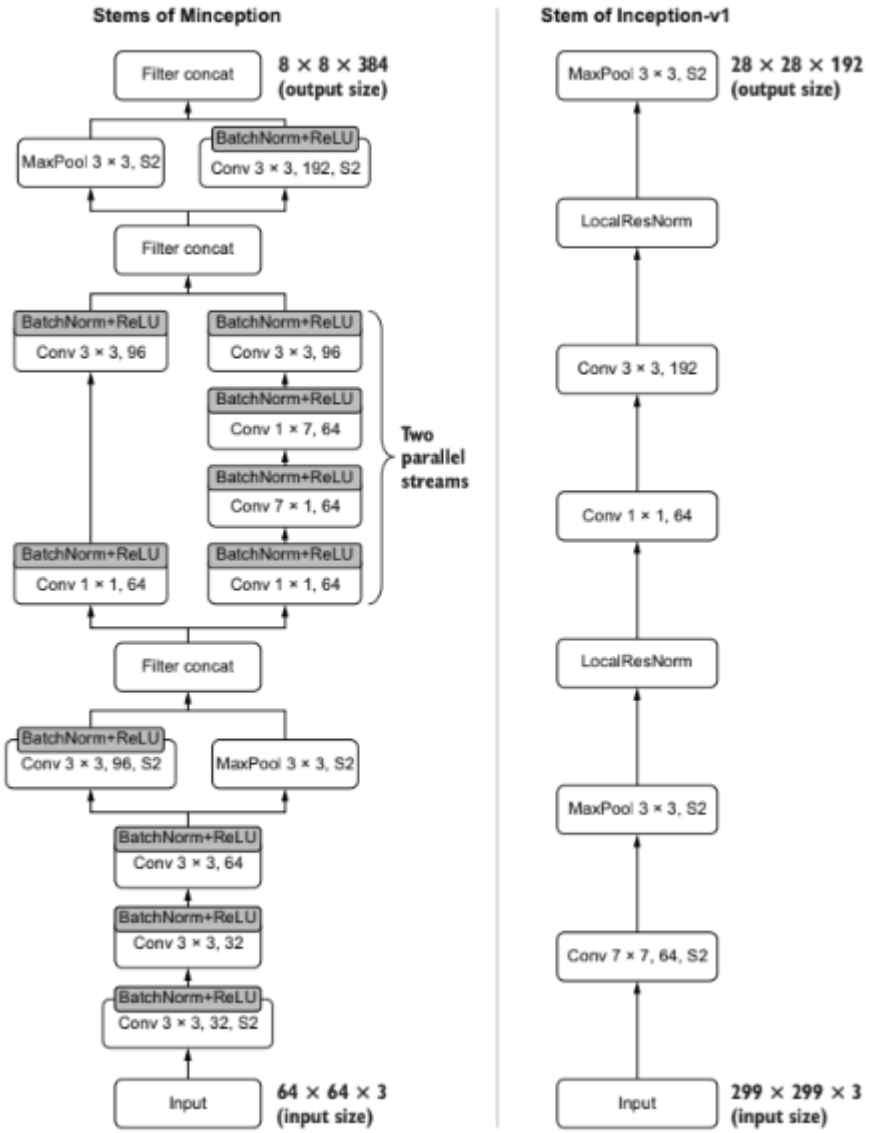


Figure 7.9 Comparing the stems of Minception and Inception-v1. Note how Minception separates the nonlinear activation of convolution layers. This is because batch normalization must be inserted in between the convolution output and the nonlinear activation.

Unlike Inception Net v1, Minception does not use local response normalization (LRN). Instead, it relies entirely on **batch normalization**, which normalizes intermediate activations to reduce internal covariate shift. This improves both training stability and convergence speed.

Batch normalization operates by normalizing layer outputs using batch-level statistics (mean and variance), followed by scaling and shifting via trainable parameters γ and β. During training, statistics are computed per batch, whereas during inference, running averages are used. Implementing batch normalization manually is complex, but TensorFlow provides a built-in layer that handles these details automatically.

```
def stem(inp, activation='relu', bn=True):

    conv1_1 = Conv2D(
        32, (3,3), strides=(2,2), activation=None,
        kernel_initializer=init, padding='same'
    )(inp) #62x62
    if bn:
        conv1_1 = BatchNormalization()(conv1_1)
    conv1_1 = Activation(activation)(conv1_1)

    conv1_2 = Conv2D(
        32, (3,3), strides=(1,1), activation=None,
        kernel_initializer=init, padding='same'
    )(conv1_1) # 31x31
    if bn:
        conv1_2 = BatchNormalization()(conv1_2)
    conv1_2 = Activation(activation)(conv1_2)

    conv1_3 = Conv2D(
        64, (3,3), strides=(1,1), activation=None,
          kernel_initializer=init, padding='same'
      )(conv1_2) # 31x31
```

Defines the function. Note that we can switch batch normalization on and off.

Note that first batch normalization is applied before applying the nonlinear activation.

Nonlinear activation is applied to the layer after the batch normalization step.

The first part of the stem until the first split


```
    if bn:
        conv1_3 = BatchNormalization()(conv1_3)
    conv1_3 = Activation(activation)(conv1_3)

    maxpool2_1 = MaxPool2D((3,3), strides=(2,2),
➥ padding='same')(conv1_3)

    conv2_2 = Conv2D(
        96, (3,3), strides=(2,2), activation=None,
        kernel_initializer=init, padding='same'
    )(conv1_3)
    if bn:
        conv2_2 = BatchNormalization()(conv2_2)
    conv2_2 = Activation(activation)(conv2_2)

    out2 = Concatenate(axis=-1)([maxpool2_1, conv2_2])

    conv3_1 = Conv2D(
        64, (1,1), strides=(1,1), activation=None,
        kernel_initializer=init, padding='same'
    )(out2)
    if bn:
        conv3_1 = BatchNormalization()(conv3_1)
    conv3_1 = Activation(activation)(conv3_1)

    conv3_2 = Conv2D(
        96, (3,3), strides=(1,1), activation=None,
        kernel_initializer=init, padding='same'
    )(conv3_1)
    if bn:
        conv3_2 = BatchNormalization()(conv3_2)
    conv3_2 = Activation(activation)(conv3_2)

    conv4_1 = Conv2D(
        64, (1,1), strides=(1,1), activation=None,
        kernel_initializer=init, padding='same'
    )(out2)
    if bn:
        conv4_1 = BatchNormalization()(conv4_1)
    conv4_1 = Activation(activation)(conv4_1)

    conv4_2 = Conv2D(
        64, (7,1), strides=(1,1), activation=None,
```

The two parallel streams of the first split

Concatenates the outputs of the two parallel streams in the first split

First stream of the second split

Second stream of the second split

```
    if bn:
        conv4_2 = BatchNormalization()(conv4_2)

    conv4_3 = Conv2D(
        64, (1,7), strides=(1,1), activation=None,
        kernel_initializer=init, padding='same'
    )(conv4_2)
    if bn:
        conv4_3 = BatchNormalization()(conv4_3)
    conv4_3 = Activation(activation)(conv4_3)
```



```
    conv4_4 = Conv2D(
        96, (3,3), strides=(1,1), activation=None,
        kernel_initializer=init, padding='same'
    )(conv4_3)
    if bn:
        conv4_4 = BatchNormalization()(conv4_4)
    conv4_4 = Activation(activation)(conv4_4)

    out34 = Concatenate(axis=-1)([conv3_2, conv4_4])

    maxpool5_1 = MaxPool2D((3,3), strides=(2,2), padding='same')(out34)
    conv6_1 = Conv2D(
        192, (3,3), strides=(2,2), activation=None,
        kernel_initializer=init, padding='same'
    )(out34)
    if bn:
        conv6_1 = BatchNormalization()(conv6_1)
    conv6_1 = Activation(activation)(conv6_1)

    out56 = Concatenate(axis=-1)([maxpool5_1, conv6_1])

    return out56
```

Second stream of the second split

Concatenates the outputs of the two streams in the second split

The third (final split) and the concatenation of the outputs

Listing 7.6 Defining the stem of MinceptionA critical design note is that nonlinear activations are applied after batch normalization, following the original batch normalization formulation. Although alternative placements exist, this implementation adheres closely to the original paper.

7.2.2 Implementing the Inception-ResNet Type A Block

The Inception-ResNet Type A block combines the multi-branch convolutional design of Inception with **residual (skip) connections**. Residual connections perform element-wise addition between a block's input and its output, improving gradient flow and mitigating vanishing gradient problems in deep networks.

Residual connections require matching tensor dimensions; otherwise, element-wise addition fails. When dimensions differ, the architecture must be adjusted to ensure compatibility.

From a mathematical standpoint, residual connections allow the network to learn **residual functions** instead of direct mappings, which simplifies optimization and improves convergence.

Minception's Type A block introduces two main enhancements over Inception Net v1:

1. Batch normalization applied throughout the block
2. A residual connection from the block input to its output

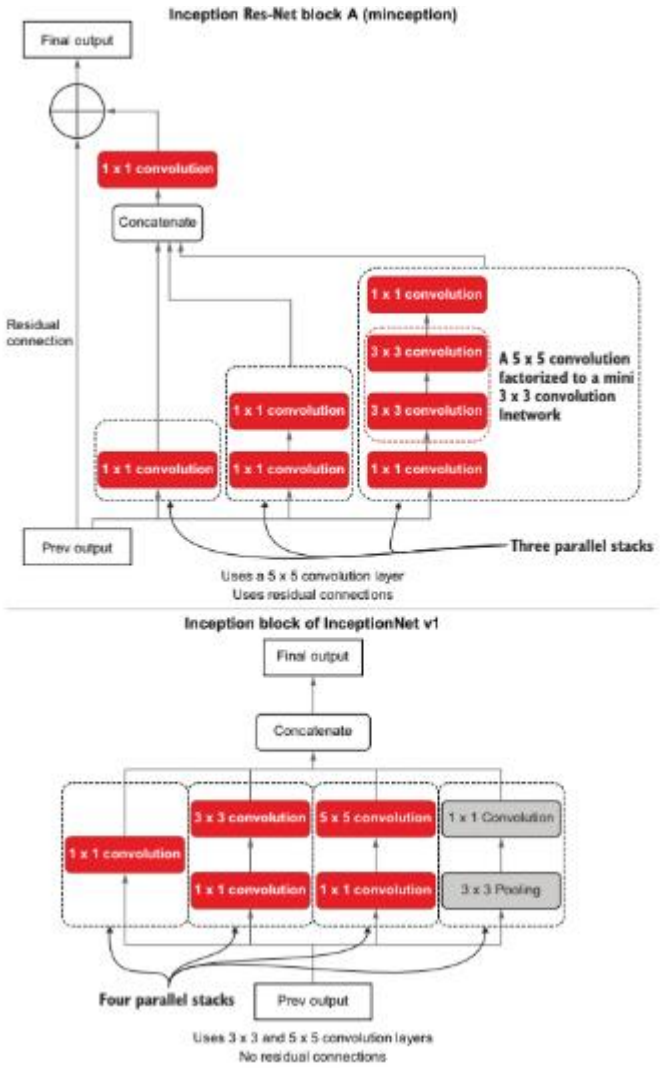


Figure 7.10 Comparison between Inception-ResNet block A (Minception) and Inception net v1's Inception block

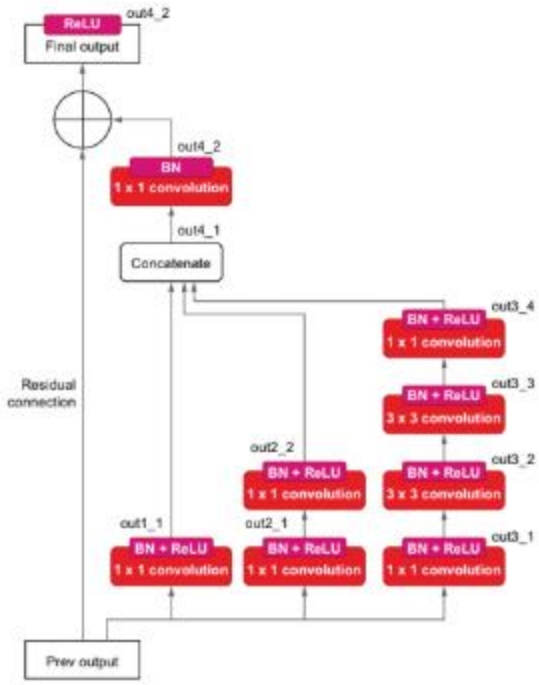


Figure 7.11 Illustration of the Minception-ResNet block A with annotations from code listing 7.7

Listing 7.7 Implementation of Minception-ResNet block A

```
def inception_resnet_a(inp, n_filters, initializer, activation='relu',
➥ bn=True, res_w=0.1):
    out1_1 = Conv2D(
        n_filters[0][0], (1,1), strides=(1,1),
➥ activation=None,
        kernel_initializer=initializer,
➥ padding='same'
    )(inp)
    if bn:
        out1_1 = BatchNormalization()(out1_1)
    out1_1 = Activation(activation)(out1_1)
```

The first parallel stream in the block


```
    out2_1 = Conv2D(
        n_filters[1][0], (1,1), strides=(1,1),
➥ activation=None,
        kernel_initializer=initializer, padding='same'
    )(inp)
    if bn:
        out2_1 = BatchNormalization()(out2_1)
    out2_1 = Activation(activation)(out2_1)

    out2_2 = Conv2D(
        n_filters[1][1], (1,1), strides=(1,1), activation=None,
        kernel_initializer=initializer, padding='same'
    )(out2_1)
    if bn:
        out2_2 = BatchNormalization()(out2_2)
    out2_2 = Activation(activation)(out2_2)

    out2_3 = Conv2D(
        n_filters[1][2], (1,1), strides=(1,1), activation=None,
        kernel_initializer=initializer, padding='same'
    )(out2_2)

    out3_1 = Conv2D(
        n_filters[2][0], (1,1), strides=(1,1), activation=None,
        kernel_initializer=initializer, padding='same'
    )(inp)
    if bn:
        out3_1 = BatchNormalization()(out3_1)
    out3_1 = Activation(activation)(out3_1)

    out3_2 = Conv2D(
        n_filters[2][1], (3,3), strides=(1,1), activation=None,
        kernel_initializer=initializer, padding='same'
    )(out3_1)
    if bn:
        out3_2 = BatchNormalization()(out3_2)
    out3_2 = Activation(activation)(out3_2)

    out3_3 = Conv2D(
        n_filters[2][2], (3,3), strides=(1,1), activation=None,
        kernel_initializer=initializer, padding='same'
    )(out3_2)
    if bn:
        out3_3 = BatchNormalization()(out3_3)
    out3_3 = Activation(activation)(out3_3)

    out3_4 = Conv2D(
        n_filters[2][3], (1,1), strides=(1,1), activation=None,
```

The second parallel stream in the block

The third parallel stream in the block

```
        n_filters[2][3], (1,1), strides=(1,1), activation=None,
        kernel_initializer=initializer, padding='same'
    )(out3_3)
    if bn:
        out3_4 = BatchNormalization()(out3_4)
    out3_4 = Activation(activation)(out3_4)
```



```
out4_1 = Concatenate(axis=-1)([out1_1, out2_2, out3_4])
out4_2 = Conv2D(
    n_filters[3][0], (1,1), strides=(1,1), activation=None,
    kernel_initializer=initializer, padding='same'
)(out4_1)
if bn:
    out4_2 = BatchNormalization()(out4_2)

out4_2 += res_w * inp
out4_2 = Activation(activation)(out4_2)

    return out4_2
```

Concatenate the outputs of the three separate streams.

Incorporate the residual connection (which is multiplied by a factor to improve the gradient flow).

Within the block, batch normalization is applied before nonlinear activation, and the final 1×1 convolution omits activation so that ReLU is applied **after** the residual addition.

7.2.3 Implementing the Inception-ResNet Type B Block

The Inception-ResNet Type B block is structurally similar to Type A but simpler. It consists of fewer parallel streams and replaces large convolutions with **factorized kernels**, specifically decomposing 7×7 convolutions into 1×7 and 7×1 operations to reduce computational cost.

Although simpler, the Type B block still employs batch normalization and residual connections, maintaining training stability.

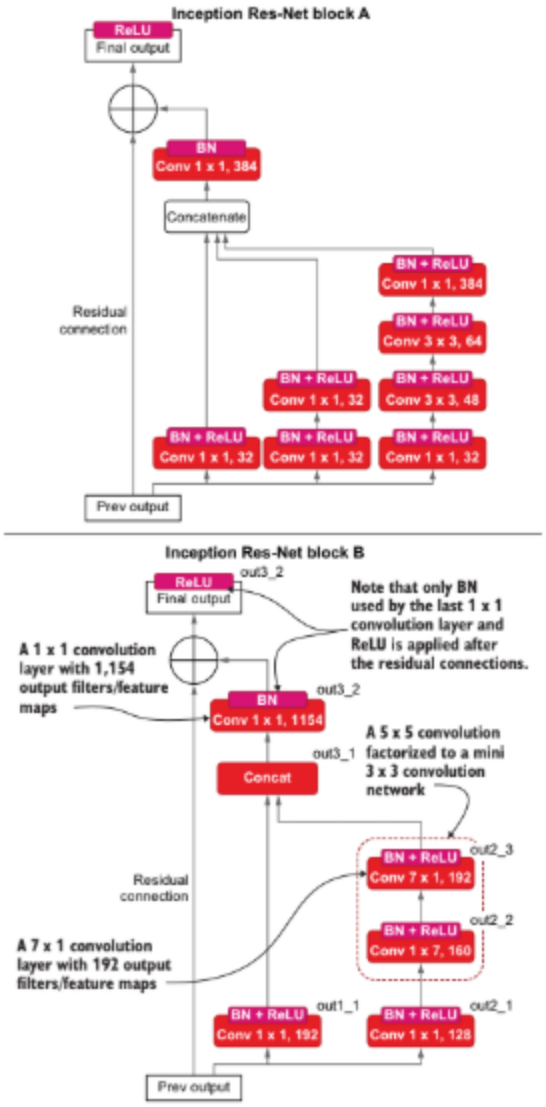


Figure 7.12 Minception's Inception-ResNet block B (left) and Minception's Inception-ResNet lock A (right) side by sideListing placement:

Listing 7.8 The implementation of Minception-ResNet block B

7.2.4 Implementing the Reduction Block

Reduction blocks are used to reduce spatial dimensions while increasing feature depth. Inspired by Inception-ResNet architectures, Minception's reduction blocks resemble Inception blocks but **do not include residual connections**.

These blocks combine convolutional and pooling operations in parallel paths, followed by feature concatenation.

Listing 7.9 Implementation of the reduction block of Minception

```
def reduction(inp, n_filters, initializer, activation='relu', bn=True):
    # Split to three branches
    # Branch 1
```



```
    out1_1 = Conv2D(
        n_filters[0][0], (3,3), strides=(2,2),
        kernel_initializer=initializer, padding='same'
    )(inp)
    if bn:
        out1_1 = BatchNormalization()(out1_1)
    out1_1 = Activation(activation)(out1_1)

    out1_2 = Conv2D(
        n_filters[0][1], (3,3), strides=(1,1),
        kernel_initializer=initializer, padding='same'
    )(out1_1)
    if bn:
        out1_2 = BatchNormalization()(out1_2)
    out1_2 = Activation(activation)(out1_2)

    out1_3 = Conv2D(
        n_filters[0][2], (3,3), strides=(1,1),
        kernel_initializer=initializer, padding='same'
    )(out1_2)
    if bn:
        out1_3 = BatchNormalization()(out1_3)
    out1_3 = Activation(activation)(out1_3)

    # Branch 2
    out2_1 = Conv2D(
        n_filters[1][0], (3,3), strides=(2,2),
        kernel_initializer=initializer, padding='same'
    )(inp)
    if bn:
        out2_1 = BatchNormalization()(out2_1)
    out2_1 = Activation(activation)(out2_1)

    # Branch 3
    out3_1 = MaxPool2D((3,3), strides=(2,2), padding='same')(inp)

    # Concat the results from 3 branches
    out = Concatenate(axis=-1)([out1_3, out2_1, out3_1])

    return out
```

First parallel stream of convolutions

Second parallel stream of convolutions

Third parallel stream of pooling

Concatenates all the outputs

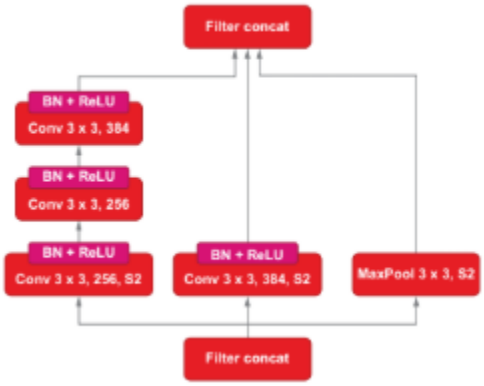


Figure 7.13 Illustration of the reduction block

7.2.5 Putting Everything Together

The final Minception architecture integrates all previously defined components into a streamlined network. The model consists of a stem, one Inception-ResNet Type A block, two Type B blocks, an average pooling layer, dropout, and a final dense layer with 200 softmax outputs.

Input preprocessing is modified to better align with the original research. During training, images are randomly cropped from 64×64 to 56×56 and randomly contrast-adjusted. During validation and testing, center cropping is applied. These preprocessing steps are implemented using TensorFlow's preprocessing layers rather than ImageDataGenerator.

Listing 7.10 The final Minception model

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPool2D, Dropout,
➥ AvgPool2D, Dense, Concatenate, Flatten, BatchNormalization, Activation
➥ from tensorflow.keras.layers.experimental.preprocessing import RandomCrop,
➥ RandomContrast
from tensorflow.keras.models import Model
from tensorflow.keras.losses import CategoricalCrossentropy
import tensorflow.keras.backend as K
from tensorflow.keras.callbacks import EarlyStopping, CSVLogger

inp = Input(shape=(64,64,3))                                   # Define the 64 × 64 Input layer.

crop_inp = RandomCrop(56, 56, seed=random_seed)(inp)           # Perform random cropping on the input (randomness is only activated during training).
crop_inp = RandomContrast(0.3, seed=random_seed)(crop_inp)     # Perform random contrast on the input (randomness is only activated during training).

stem_out = stem(crop_inp)                                      # Define the output of the stem.

inc_a = inception_resnet_a(stem_out, [(32,),(32,32), (32, 48, 64,
➥ 384),(384,)], initializer=init)                             # Define the Inception-ResNet block (type A).

red = reduction(inc_a, [(256,256,384),(384,)], initializer=init)   # Define a reduction layer.

inc_b1 = inception_resnet_b(red, [(192,),(128,160,192),(1152,)],
➥ initializer=init)
inc_b2 = inception_resnet_b(inc_b1,  [(192,),(128,160,192),(1152,)],
➥ initializer=init)                                           # Define 2 Inception-ResNet block (type B).

avgpool1 = AvgPool2D((4,4), strides=(1,1), padding='valid')(inc_b2)
flat_out = Flatten()(avgpool1)
drcpout1 = Dropout(0.5)(flat_out)
out_main = Dense(200, activation='softmax',  kernel_initializer=init,
➥ name='final')(flat_out)                                     # Define the final prediction layer.

minception_resnet_v2 = Model(inputs=inp, outputs=out_main)     # Define the model.
minception_resnet_v2.compile(loss='categorical_crossentropy',
➥ optimizer='adam', metrics=['accuracy'])                     # Compile the model with categorical crossentropy loss and the adam optimizer.
```

The model is compiled using categorical cross-entropy loss and the Adam optimizer.

7.2.6 Training Minception

Training Minception closely resembles earlier workflows but introduces a **learning rate reduction schedule** using the ReduceLROnPlateau callback. Instead of using a fixed learning rate, the optimizer dynamically reduces the learning rate when validation loss plateaus, allowing finer convergence as training progresses.

Listing 7.11 Training the Minception model

Training logs demonstrate substantial accuracy improvements, reaching approximately **48–50% validation accuracy** and **~51% test accuracy**, nearly doubling the performance of the earlier Inception Net v1 model.

7.3 Transfer Learning: Using Pretrained Networks

To further improve performance, transfer learning is introduced. The approach leverages a pretrained **Inception-ResNet v2** model trained on ImageNet. By removing the original classification head and fine-tuning the network on the Tiny-ImageNet-200 dataset, higher accuracy can be achieved with less training effort.

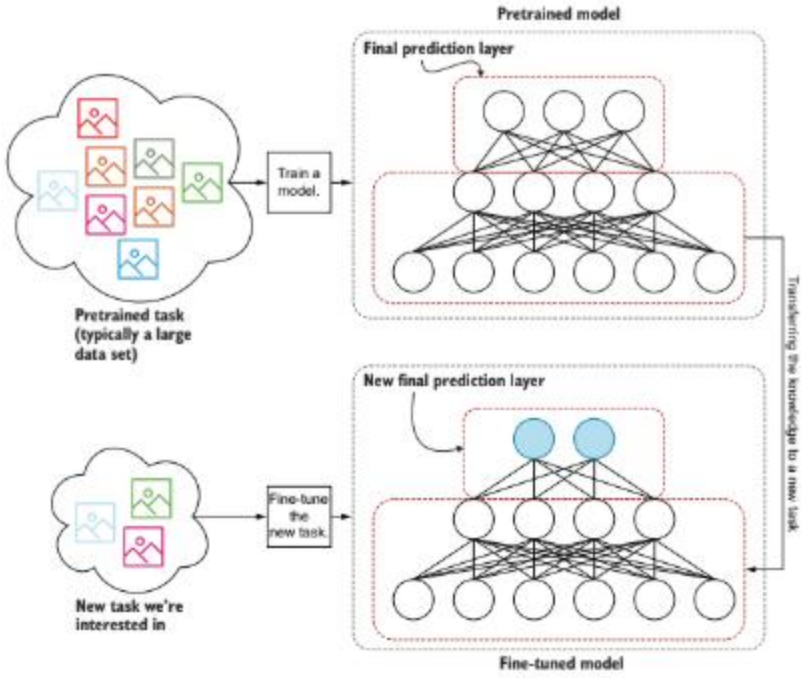


Figure 7.14 How transfer learning works. First, we start with a model that is pretrained on a larger data set that is solving a similar/relevant task to the one we're interested in. Then we transfer the model weights (except the last layer) and fit a new prediction layer on top of the existing weights. Finally, we fine-tune the model on a new task.Keras enables seamless access to pretrained models, allowing easy adaptation to new tasks.

Listing 7.12 Implementing a model based on the pretrained Inception-ResNet v2

```
from tensorflow.keras.applications import InceptionResNetV2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout

model = Sequential([
    Input(shape=(224,224,3)),
    InceptionResNetV2(include_top=False, pooling='avg'),
    Dropout(0.4),
    Dense(200, activation='softmax')
])

adam = tf.keras.optimizers.Adam(learning_rate=0.0001)
model.compile(loss='categorical_crossentropy', optimizer=adam,
➥ metrics=['accuracy'])
model.summary()
```

Some important imports

Defining an input layer for a 224 × 224 image

The pretrained weights of the Inception-ResNet v2 model

Apply 40% dropout

Final prediction layer with 200 classes

Using a smaller learning rate since the network is already trained on ImageNet data (chosen empirically)

Listing 7.13 The modified ImageDataGenerator that produces 224 × 224 images

```
def get_train_valid_test_data_generators(batch_size, target_size):

    image_gen_aug = ImageDataGenerator(
        samplewise_center=False, rotation_range=30, width_shift_range=0.2,
```



```
        height_shift_range=0.2, brightness_range=(0.5,1.5), shear_range=5,
        zoom_range=0.2, horizontal_flip=True, validation_split=0.1
    )
    image_gen = ImageDataGenerator(samplewise_center=False)

    partial_flow_func = partial(
        image_gen_aug.flow_from_directory,
        directory=os.path.join('data','tiny-imagenet-200', 'train'),
        target_size=target_size,
        classes=None,
        class_mode='categorical',
        interpolation='bilinear',
        batch_size=batch_size,
        shuffle=True,
        seed=random_seed)

    # Get the training data subset
    train_gen = partial_flow_func(subset='training')
    # Get the validation data subset
    valid_gen = partial_flow_func(subset='validation')

    # Defining the test data generator
    test_df = get_test_labels_df(os.path.join('data','tiny-imagenet-200',
➥ 'val', 'val_annotations.txt'))
    test_gen = image_gen.flow_from_dataframe(
        test_df,
        directory=os.path.join('data','tiny-imagenet-200',  'val', 'images'),
        target_size=target_size,
        classes=None,
        class_mode='categorical',
        interpolation='bilinear',
        batch_size=batch_size,
        shuffle=False
    )
    return train_gen, valid_gen, test_gen
```

Defines a data-augmenting image data generator and a standard image data generator

Defines a partial function to avoid repeating arguments

Uses a target size of 224 × 224

Uses bilinear interpolation to make images bigger

Defines the data generators for training and validation sets

Defines the test data generator

Uses a target size of 224 × 224 and bilinear interpolation

7.4 Grad-CAM: Making CNNs Confess

Grad-CAM is introduced as a model interpretability technique that visualizes **where a CNN focuses** when making predictions. It uses gradients of the predicted class with respect to convolutional feature maps to generate a heatmap highlighting important regions.

This technique helps validate that the model is learning meaningful visual patterns rather than exploiting spurious correlations.

Listing 7.14 Pseudocode of Grad-CAM computations

```
Define: model (Trained Inception Resnet V2 model)
Define: probe_ds (A list of image, class(integer) tuples e.g. [(image,
➥ class-int), (image, class-int), …]) that we will use to interpret the model
Define: last_conv (Last convolution layer of the model - closest to the
➥ prediction layer)
Load the model (inceptionnet_resnet_v2.h5)

For img, cls in probe_ds:

    # Computing the gradient map and its associated weights
    Compute the model's final output (out) and last_conv layer's output
➥ (conv_out)
    Compute the gradient d (out[cls]) / d (conv_out) and assign to grad
    Compute channel weights by taking the mean of grad over width and
➥ height dimensions (Results in a [batch size(=1), 1, 1, # channels in
➥ last_conv] tensor)

    # Creating the final gradient heatmap
    grad = grad * weights # Multiply grad with weights
    grad = tf.reduce_sum(grad, axis=-1) # Take sum over channels
    grad = tf.nn.relu(grad) # Apply ReLU activation to obtain the gradient
➥ heatmap

    # Visualizing the gradient heatmap
    Resize the gradient heatmap to a size of 224x224
    Superimpose the gradient heatmap on the original image (img)
    Plot the image and the image with the gradient heatmap superimposed
➥ side by side
```

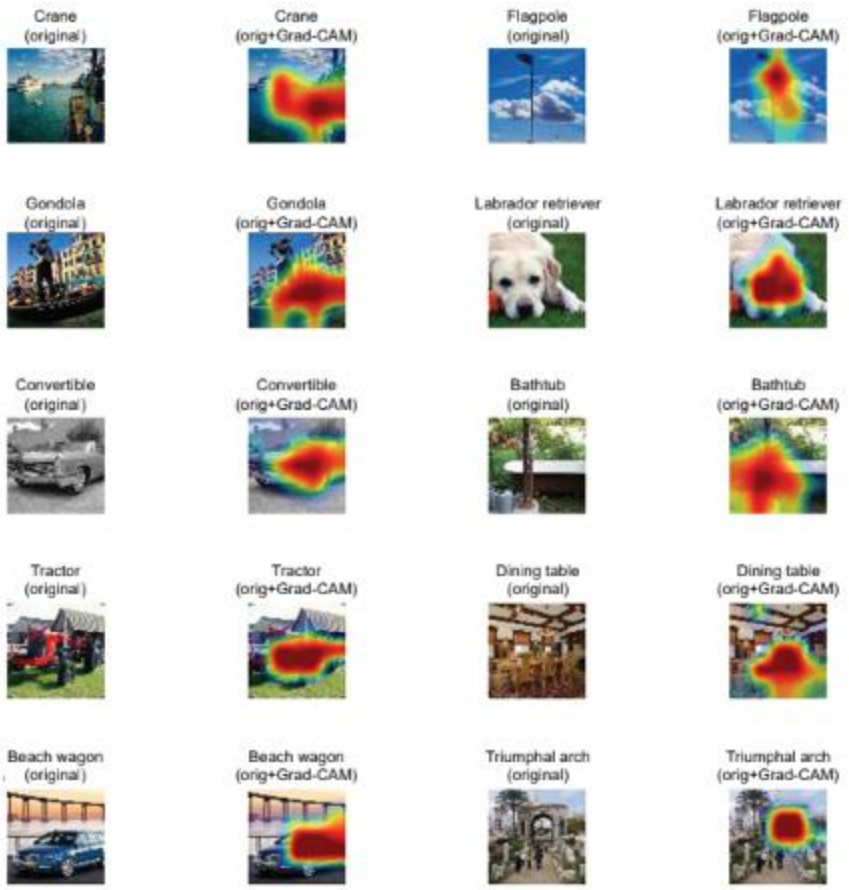


Figure 7.15 Visualization of the Grad-CAM output

The visualizations demonstrate that the trained model correctly focuses on semantically relevant regions, increasing trust in its predictions.