Chapter 7

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

Modern convolutional neural networks are capable of powerful image classification, but their performance often suffers from a gap between training and real-world data. One such case arises when a network achieves very high training accuracy, around 94%, yet fails to generalize, with validation and test accuracies below 30%. This problem is a clear sign of overfitting, where the model memorizes the training set rather than learning generalizable features. The goal, therefore, is to systematically reduce overfitting, design a more efficient architecture, and finally make the model's decisions interpretable.

The first set of techniques targets overfitting directly. Image augmentation is a common approach: training data is artificially expanded by creating multiple transformed versions of the same image. Random rotations, translations, changes in brightness or contrast, zooming, shearing, and horizontal flipping all help create diversity in the training set. Validation data, however, remains untouched to ensure consistency across runs. Figures 1 and 2 illustrate how transformations such as rotation, shear, or occlusion alter training images, producing richer examples while keeping the class labels intact. This augmented input prevents the network from relying on a narrow set of features.

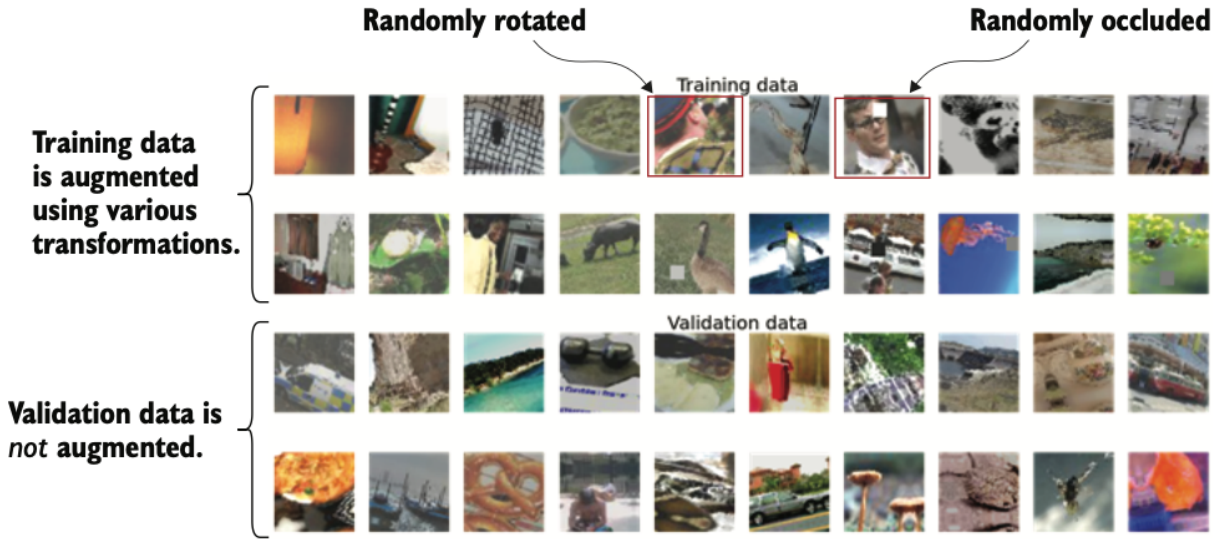

Figure 1 Difference between training data and validation data after the augmentation step


Original image
rotation=-50
rotation=-20
rotation=0
rotation=20
rotation=50
height_shift=-20
height_shift=-10
height_shift=0
height_shift=10
height_shift=20
width_shift=-20
width_shift=-10
width_shift=0
width_shift=10
width_shift=20
brightness=0.2
brightness=0.5
brightness=1
brightness=1.2
brightness=1.5
shear=-50
shear=-20
shear=0
shear=20
shear=50
zoom=0.2
zoom=0.5
zoom=1
zoom=1.2
zoom=1.5
flip_horiz=True
flip_horiz=False

Figure 2 Effects of different augmentation parameters and their values of the ImageDataGenerator

Reproduced code for ImageDataGenerator:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator


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,
    fill_mode='reflect',
    validation_split=0.1
)


image_gen = ImageDataGenerator(samplewise_center=False)
```

Another important method is dropout, where a fraction of neurons are randomly deactivated during training. This forces the network to learn redundant representations: if one pathway is dropped, others must compensate by discovering different features. For instance, when classifying cats, one training pass may rely on whiskers, while another may learn ears or tails, ultimately producing a more robust classifier. Figures 3 show how dropout masks neurons and scales outputs to maintain stability. In practice, dropout is applied to fully connected layers and intermediate outputs, with recommended rates of 70% for auxiliary outputs and 40% for final pooling layers.

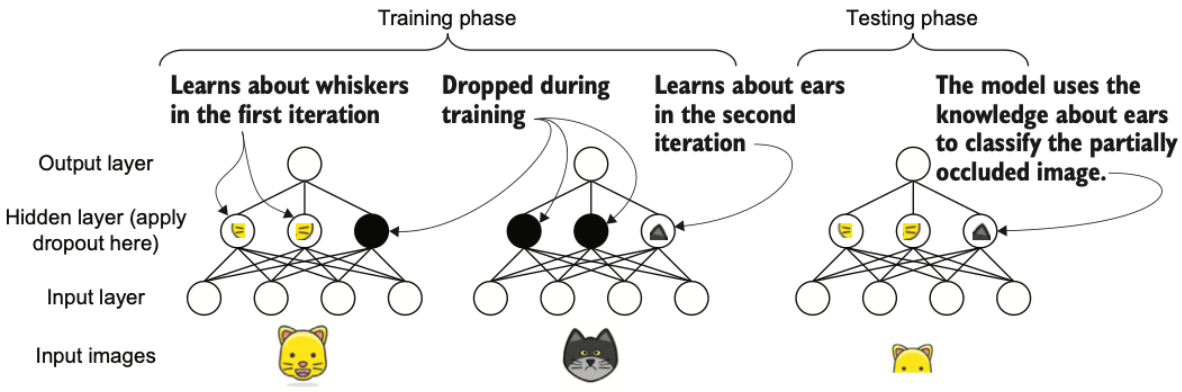

Figure 3 Dropout effect might change the network when learning to classify cat images

A third guard against overfitting is early stopping. During training, validation accuracy initially improves alongside training accuracy, but after a certain point, it plateaus and then declines, even while training accuracy continues to rise. This is the hallmark of overfitting, as depicted in Figure 4. Early stopping halts training when validation performance stops improving, ensuring the model captures generalizable features without drifting into memorization. The process, requires monitoring metrics such as validation loss and terminating training after several epochs without improvement.

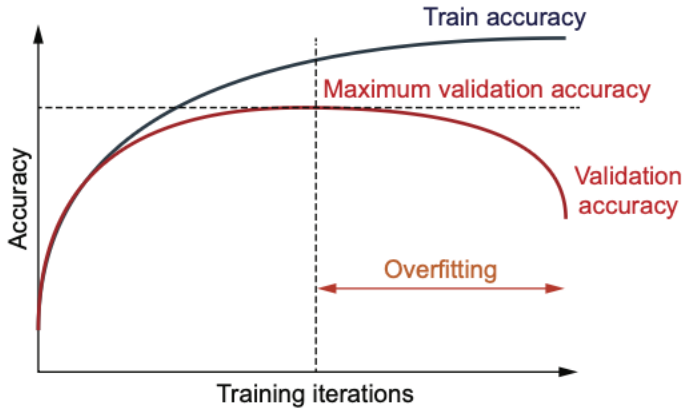

Figure 4 An illustration of overfitting

With overfitting reduced, attention turns to improving the model's structure itself. The original Inception network, though groundbreaking, is large and unwieldy. A streamlined variant, called Minception, incorporates ideas from newer architectures like Inception-ResNet. The Minception design includes a stem of convolution and pooling layers, followed by two types of Inception-ResNet blocks (A and B), reduction blocks to shrink spatial dimensions, and final pooling with prediction layers. Figure 5 compares the stems of Minception and the older Inception v1, highlighting Minception's use of batch normalization

rather than local response normalization. Batch normalization stabilizes training by normalizing activations within each batch, reducing internal covariate shift and accelerating convergence.

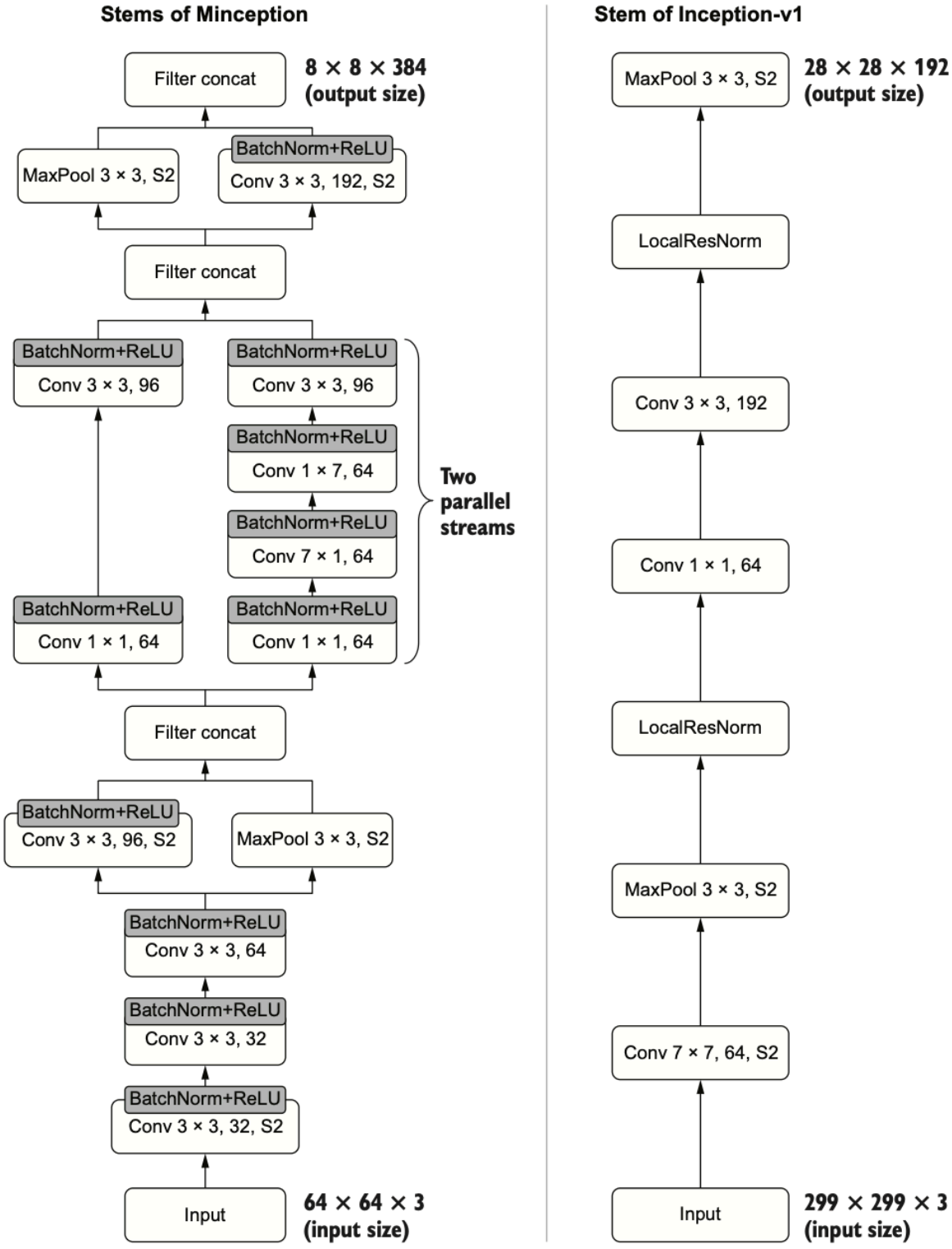

Figure 5 Comparing the stems of Minception and Inception-v1

The introduction of residual connections further strengthens the architecture. These connections add the input of a block directly to its output, preserving information and ensuring gradients flow without vanishing. The structure of Minception's Inception-ResNet block A, which combines parallel convolution paths with residual shortcuts. Block B offers a simpler variant with fewer streams. Together, these blocks allow Minception to achieve significantly better validation accuracy, roughly 50% compared to the 30% of its predecessor.

While Minception shows strong gains, even better performance can be achieved through transfer learning. Instead of training from scratch, a pretrained network such as Inception-ResNet v2—trained on the massive ImageNet dataset—is adapted to the target dataset. Only the top layers are retrained, while the earlier layers retain their learned filters.