CHAPTER 14

Deep Computer Vision Using Convolutional Neural Networks

Chapter 14 introduces **convolutional neural networks (CNNs)** for deep computer vision, covering their biological inspiration, core layers, classic architectures (LeNet, AlexNet, GoogLeNet, VGG, ResNet, Xception, SENet), and applications like detection and segmentation.

Core CNN building blocks

CNNs were inspired by the **visual cortex**, where neurons have local receptive fields and build up from edges to complex shapes across layers. Key components:

- **Convolutional layers**:
    - Each neuron connects only to a local **receptive field**, using shared filters (kernels) that slide across the input to produce **feature maps** (weight sharing and local connectivity greatly reduce parameters versus fully connected layers).
    - Hyperparameters: number of filters, kernel size (often 3×3), stride, and padding (same with zero padding vs valid without).
- **Pooling layers**:
    - Subsample spatial dimensions to reduce compute and overfitting; **max pooling** (typically 2×2, stride 2) is standard and introduces limited **translation invariance**.
    - Variants include average pooling, depthwise pooling, and **global average pooling** (one value per feature map, often before final dense layer).

CNNs use 4D tensors: (batch, height, width, channels) for inputs and (fh, fw, in_channels, out_channels) for conv kernels; Keras provides Conv2D, MaxPool2D, GlobalAvgPool2D, etc., to implement these layers.

Practical CNN architectures and training issues

A typical CNN stacks several **Conv–ReLU** layers, periodically followed by pooling, then ends with one or more dense layers (possibly with dropout) for classification. Important practical points:

- Use several small kernels (e.g., two 3×3) rather than one large 5×5 to reduce parameters and often improve performance.
- CNNs can be memory-intensive; training must store intermediate activations for backprop, so large feature maps or big batches can easily exhaust GPU RAM.

- Remedies for memory issues: reduce batch size, use larger strides/pooling, cut or narrow layers, or use lower-precision floats (e.g., float16).

A sample Fashion-MNIST CNN uses increasing filter counts (64 → 128 → 256) with 3×3 kernels and pooling, then dense layers with dropout, reaching >92% test accuracy—better than dense-only networks.

Landmark CNN architectures

The chapter surveys major CNN families and their innovations:

- **LeNet-5** (1998): early CNN for digit recognition with conv + average pooling stacks and small fully connected tail; uses tanh activations and a special RBF-like output layer.
- **AlexNet** (2012): scaled-up CNN that won ImageNet with 17% top-5 error; innovations include ReLU activations, heavy **data augmentation**, **dropout**, and local response normalization (LRN), plus overlapping pooling.
- **VGGNet** (2014): very deep but conceptually simple stacks of 3×3 conv layers and pooling, with many filters; shows depth and uniform small kernels work very well.
- **GoogLeNet / Inception** (2014+): introduces **Inception modules** that run parallel convs with different kernel sizes (1×1, 3×3, 5×5) plus pooling, then concatenate outputs; uses 1×1 bottleneck convs to reduce channels, enabling very deep nets with far fewer parameters than AlexNet.
- **ResNet** (2015): adds **residual (skip) connections** that let layers learn residuals $F(x)$ added to inputs x, enabling training of >100-layer nets; uses 3×3 convs with BatchNorm and ReLU in residual units, with occasional 1×1 convs in the skip path when changing feature-map size or depth.
- **Xception**: builds on Inception by using **depthwise separable convolutions**, separating spatial filtering and cross-channel mixing to reduce computation and often improve accuracy.
- **SENet** (2017): adds **Squeeze-and-Excitation (SE) blocks** that learn a channel-wise gating vector via global pooling and a small bottleneck MLP, then rescale feature maps to emphasize useful channels; this boosts existing architectures like ResNet/Inception and achieved ~2.25% top-5 error.

The chapter walks through implementing a ResNet-34 by defining a custom ResidualUnit Keras layer with main and skip paths, then stacking many units with occasional downsampling and channel doubling.

Transfer learning and pretrained models

Keras provides many **pretrained CNNs** in keras.applications (Xception, ResNet, Inception, VGG, MobileNet, etc.) trained on ImageNet. Common workflows:

- Use a model with weights="imagenet" and include_top=True for direct ImageNet-style classification.
- For **transfer learning** to new classes:
    - Load the base model with include_top=False.
    - Add your own head (e.g., GlobalAveragePooling2D → Dense softmax).
    - Freeze base layers and train the head, then optionally unfreeze upper base layers and fine-tune with a lower learning rate.

The chapter demonstrates flower classification using TFDS (tf_flowers) with an Xception base and custom top layers, plus standard preprocessing and augmentation.

Detection and segmentation

Beyond image-level classification, CNNs support more complex vision tasks:

- **Object detection**: localizing multiple objects with bounding boxes and class labels; architectures include R-CNN variants (Fast/Faster R-CNN), SSD, and YOLO.
- **Semantic segmentation**: classifying each pixel, often using **fully convolutional networks (FCNs)** that replace dense layers with 1×1 convs and use upsampling (deconvolution / transposed conv) plus skip connections to recover spatial detail.
- **Instance segmentation**: like semantic segmentation but distinguishing individual instances (e.g., each separate bicycle); Mask R-CNN extends Faster R-CNN by predicting a mask per detected box.

The chapter notes that many of these models are available as TensorFlow implementations and pretrained checkpoints, making advanced vision tasks accessible without training from scratch.