Deep Computer Vision Using Convolutional Neural Networks

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s.

The Visual Cortex's Architecture

- Many neurons in the visual cortex have a small local receptive field. They react only to visual stimuli located in a limited visual field region.
- Some neurons have larger receptive fields and react to more complex patterns that are combinations of the lower-level patterns.
- Similarly, higher-level neurons in a neural network are based on the outputs of neighboring lower-level neurons.

Convolutional Layers

- Convolutional layers are the building blocks of CNNs.
    - Neurons in the first convolutional layer are connected to pixels in their receptive fields, not to every pixel in the input image.
    - Each neuron in the second convolutional layer is connected to neurons within a small rectangle in the first layer.
    - The network can concentrate on small low-level features in the first hidden layer and assemble them into larger higher-level features in the second hidden layer.
        - Previous neural networks needed 1D data. CNN layers need 2D data (more accurately, they should be represented in 3D).
- Every layer in the CNN has similar characteristics.
    - Each layer must have the same height and width as the previous layer.
        - It is common to add zeroes around the inputs. This is called zero padding.
    - A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer in rows i to i + fh - 1 and columns j to j + fw - 1.
        - fh and fw are the height and the width of the receptive field.
    - The stride is the horizontal or vertical step size from one receptive field to the next.
        - A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows i × sh to i × sh + fh – 1, columns j × sw to j × sw + fw – 1, where sh and sw are the vertical and horizontal strides.

Filters

- A neuron's weights can be represented as a small image the size of the receptive field.
- The horizontal and vertical filters are two possible sets of weights called filters (or convolution kernels).
    - The horizontal lines get enhanced in a horizontal filter while the rest gets blurred.
    - In a vertical filter, the vertical lines get enhanced instead.
    - A layer of neurons using the same filter outputs a feature map, highlighting the areas in an image that most activate the filter.

Stacking Multiple Feature Maps

- A convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting various features anywhere in its inputs.
    - All neurons in a feature map share the same parameters (kernel and bias term).
- Input images are composed of multiple sublayers, one per color channel.
    - RGB are the main three color channels. Grayscale images have only one color channel, while others may have multiple.
    - A neuron located in row i, column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer l – 1, located in rows i × sh to i × sh + fh – 1 and columns j × sw to j × sw + fw – 1, across all feature maps (in layer l – 1).

Convolutional Layers in Keras

- Convolutional layers expect images with similar dimensions. We may have to rescale our image dataset using Keras's **Rescaling()** method.
    - This method may output 4D tensors. In this case, the first number represents the number of sample images, the second represents the height, the third represents the width, and the last represents the pixel value for the color channels.
- Keras's **Conv2D(x, y)** function creates a convolutional layer with x filters, each size y x y.
    - By default, this function has its *padding* argument set to "valid," which means no padding. We can set it to "same" to use zero padding.
    - The *strides* hyperparameter controls the size of the output feature maps.
    - We can access the layer's kernels and biases using its **get_weights()** method.
    - We can specify an activation function (e.g., ReLU) and the corresponding kernel initializer (e.g., He initialization).
        - If we don't set these parameters, the convolutional layers won't be able to learn complex patterns.

Pooling Layers

- Pooling layers subsample (shrink) the input image to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).
    - Pooling layers are similar to convolutional layers, but pooling neurons have no weights.
    - These layers aggregate the inputs using an aggregation function, such as the max or mean (for the max, we have a max pooling layer).

- A max pooling layer introduces some level of invariance to small translations.
    - If slight differences exist within some of the input images, the max pooling layer outputs an identical image, though not always.
- However, max pooling is very destructive.
    - We want equivariance: a small change in the inputs should lead to a corresponding small change in the output.
    - Invariance would output an identical image.
- Keras implements max pooling with the **MaxPool2D()** function. The **AvgPool2D()** function can create an average pooling layer.
    - Max pooling is preferred since it preserves the strongest features and erases the meaningless ones.
- Max and average pooling can be performed along the depth rather than the spatial dimension.
    - Depthwise max pooling can allow the CNN to learn to be invariant to numerous features.
        - For example, the CNN could learn multiple filters, each detecting a different rotation to the same pattern.
    - Although Keras does not provide an implementation, we can easily create our custom class.
- One last type of pooling layer is the global average pooling layer.
    - This layer computes the mean of each feature map. In other words, it outputs a single number per feature map and instance.
    - Most of the information in the feature map will be lost, but this layer can be useful in the output layer.
    - We can access this layer through Keras's **GlovalAvgPool2D()** function.

CNN Architectures

- Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on.
- The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers.
- At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).
- It is a common practice to double the number of filters after each pooling layer.
    - Since a pooling layer divides each spatial dimension by a factor of 2, we can double the number of feature maps in the next layer.
- Several CNN architectures have been developed, each improving on the other.

Le-Net-5

- Yann LeCun created this architecture in 1988.
- It uses outdated activation functions.
-

Layer	Type	Maps	Size	Kernel Size	Stride	Activation
Output	Fully connected	NA	10	NA	NA	RBF
F6	Fully connected	NA	84	NA	NA	tanh
C5	Convolution	120	1x1	5x5	1	tanh
S4	Avg pooling	16	5x5	2x2	2	tanh
C3	Convolution	16	10x10	5x5	1	tanh
S2	Avg pooling	6	14x14	2x2	2	tanh
C1	Convolution	6	28x28	5x5	1	tanh
Input	Input	1	32x32	NA	NA	NA

AlexNet

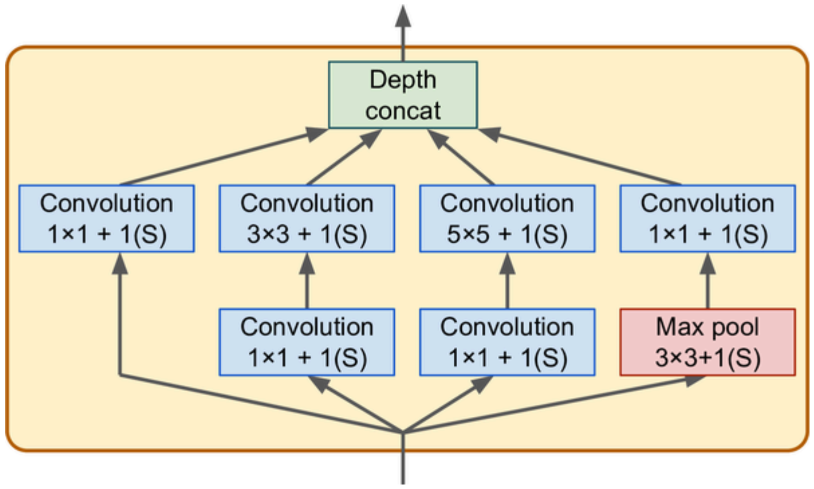
- Alex Krizhevsky developed this architecture in 2012.
- It was the first to stack convolutional layers directly on top of one another instead of stacking a pooling layer on top of each convolutional layer.
-

Layer	Type	Maps	Size	Kernel Size	Stride	Padding	Activation
Output	Fully connected	NA	1000	NA	NA	NA	Softmax
F10	Fully connected	NA	4096	NA	NA	NA	ReLU
F9	Fully	NA	4096	NA	NA	NA	ReLU

	connect ed						
S8	Max pooling	256	6x6	3x3	2	valid	NA
C7	Convolut ion	256	13x13	3x3	1	same	ReLU
C6	Convolut ion	384	13x13	3x3	1	same	ReLU
C5	Convolut ion	384	13x13	3x3	1	same	ReLU
S4	Max pooling	256	13x13	3x3	2	valid	NA
C3	Convolut ion	256	27x27	5x5	1	same	ReLU
S2	Max pooling	96	27x27	3x3	2	valid	NA
C1	Convolut ion	96	55x55	11x11	4	valid	ReLU
Input	Input	3 (RGB)	227x227	NA	NA	NA	NA

- The authors used a 50% dropout rate during training to the outputs of layers F9 and F10 to reduce overfitting.
- They also performed data augmentation by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.
  - If we don't have enough training images, we can use data augmentation and add the resulting images to the training set.
  - We can use Keras's **RandomCrop(), RandomRotation(),** and other methods.
  - This forces the model to be more tolerant of variations in the pictures.
- A local response normalization (LRN) function is performed on the CNN after the C1 and C3 layers.
  - The most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps.
  - This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization.
  - To use LRN, we can use **tf.nn.local_response_normalization()** function and wrap it in a Lambda layer.

GoogLeNet

- Developed by Christian Szegedy et al. from Google Research in 2014, this architecture achieved an error rate below 7%.
- This low error rate was made possible by subnetworks called inception modules, allowing GoogLeNet to use parameters very efficiently.
- An inception module looks like this:
    - 

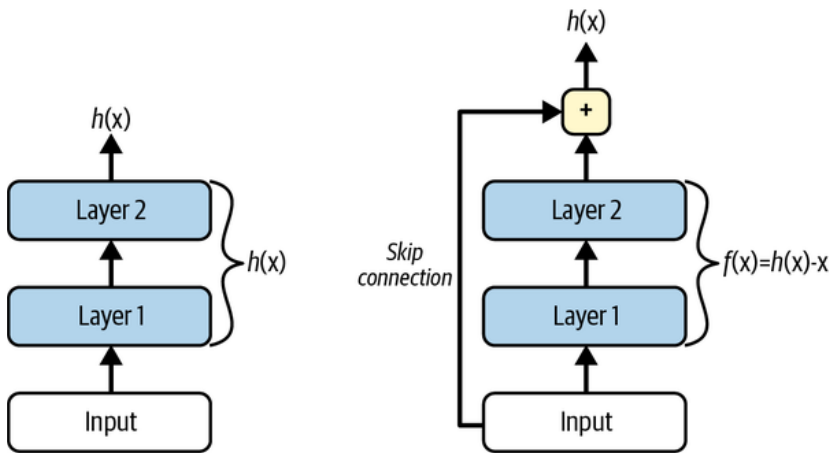
    - All outputs have the same height and width as their inputs, making it possible to concatenate all the outputs along the depth dimension.
    - Although the 1x1 convolutional layers cannot capture any features, they output fewer feature maps than their inputs.
        - These layers serve as bottleneck layers, meaning that they reduce dimensionality.
- GoogLeNet's architecture consists of several convolutional, LRN, max pool, inception modules, and global average pooling layers.
    - A dropout rate of 40% is used following the global average pooling layer.
    - The output layer uses a softmax function.

VGGNet

- Developed by Karen Simonyan and Andrew Zisserman in 2014, this architecture is rather simple.
    - It has 2-3 convolutional layers and a pooling layer, then 2-3 more convolutional layers and a pooling layer, and so on.
    - VGGNet consisted of 16-19 convolutional layers and a dense network layer with 2 hidden layers and the output layers.

ResNet

- Developed by Kaiming He et al. in 2015, the Residual Network (ResNet) uses skip connections or shortcut connections during training.
- When training a neural network, the goal is to make it model a target function h(x).
    - If we add the input x to the output of the network (i.e., you add a skip connection), then the network will be forced to model f(x) = h(x) – x rather than h(x).

    - This is called residual learning.
    - 

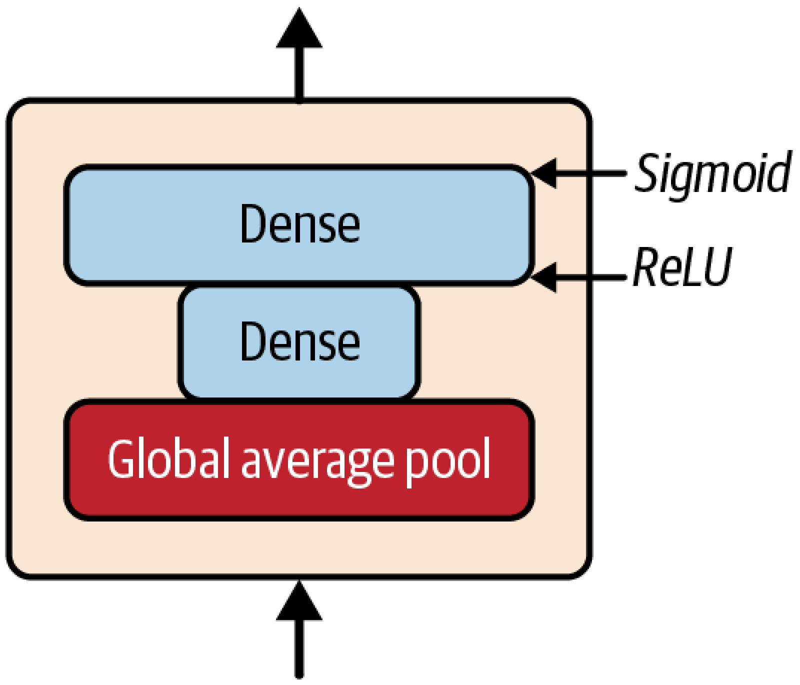
    - When we initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero.
    - If you add a skip connection, the resulting network just outputs a copy of its inputs (the identity function), which speeds up training.
- The deep residual network can be seen as a stack of residual units (RUs), where each residual unit is a small neural network with a skip connection.

Xception

- The Extreme Inception (Xception) architecture is a variant of the GoogLeNet architecture. It was developed in 2016 by François Chollet (the author of Keras).
- This architecture replaces the inception modules with depthwise separable convolution layers.
    - A separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately.
    - It is composed of two parts: the first part applies a single spatial filter to each input feature map, then the second part looks exclusively for cross-channel patterns—it is just a regular convolutional layer with 1 × 1 filters.
    - Essentially, an inception module is an intermediate between a regular convolutional layer (considers spatial patterns and cross-channel patterns jointly) and a separable convolutional layer (considers them separately).
- We can use Keras's **SeparableConv2D()** function to implement separable convolutional layers.
    - We can replace regular convolutional layers with this layer as long as the CNN's layers have a sufficient number of channels.

SENet

- The Squeeze-and-Excitation Network (SENet) extends the inception networks and ResNets and boosts their performance.
    - SENet adds a small neural network, called an SE block, to every inception module or residual unit in the GoogLeNet architecture.

- An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together.
- Essentially, it reduces irrelevant feature maps.
- 

- The global average pooling layer allows the architecture to use an embedding, which helps the CNN generalize better.

Other Noteworthy Architectures

- ResNeXt: Improves the residual units in ResNet.
- DenseNet: Composed of several dense blocks, each consisting of densely connected convolutional layers.
- MobileNet: Developed to be lightweight and fast, perfect for mobile and web applications. Based on depthwise separable convolutional layers, like Xception.
- CPSNet: Similar to DenseNet, but part of each dense block's input is concatenated directly to that block's output, without going through the block.
- EfficientNet: Can scale any CNN efficiently using compound scaling to create larger versions of smaller architectures.
- All of these architectures are available through the **tf.keras.applications** package.

Using Pretrained Models from Keras

- We won't need to implement the aforementioned models. We can simply use pre-trained models.
    - For example, we can use the ResNet-50 model through the **tf.keras.applications.ResNet50()** function.

    - Note that most of these architectures have designated input shapes. We may need to resize the input images.
    - Each model has a **preprocess_input()** function that we can use to preprocess our images.
    - Furthermore, they have a **decode_predictions()** function that returns the top k predictions.

Pretrained Models for Transfer Learning

- We can load datasets using the **tensorflow_datasets** package.
- Keras has a **Resizing()** layer we can use to resize the input images.
- Then, we can preprocess, shuffle, batch, and prefetch the dataset.
    - If we don't have enough data instances, we can use data augmentation to generate more.
- Recall that we should freeze pre-trained layers at the beginning of training. We can unfreeze them as we progress through the training.

Classification and Localization

- Sometimes, we may want to know where an object is located in a picture.
- Our goal is to predict a bounding box around the object.
    - We can try to predict the horizontal and vertical coordinates of the object's center, height, and width.
    - Our model's output layer will contain four units (typically on top of the global average pooling layer).
- Labeling the data is the most expensive part of a machine learning project.
    - We can use open-source image labeling tools, like VGG Image Annotator, LabelImg, OpenLabeler, or ImgLab.
    - Some commercial tools, like LabelBox and Supervisely, are also available.
    - We can also try <u>crowdsourcing</u> platforms, such as Amazon Mechanical Turk if we need to annotate a large number of images.
        - This is often costly and time-consuming, so we must ensure that the results are worth the effort.
- The most used metric for localization is <u>intersection over union</u> (IoU).
    - IoU is the are of overlap between the predicted bounding box and the target bounding box, divided by the area of their union.
        - We can use the **tf.keras.metrics.MeanIoU()** class for this.
    - MSE is a poor metric to evaluate how well the model predicts bounding boxes.

Object Detection

- The task of classifying and localizing multiple objects in an image is called <u>object detection</u>.

    - Object detection predicts an objectness score, the estimated probability that the image contains an object centered near the middle.
    - This is binary classification output using a dense output layer with a single neuron, using the sigmoid function and binary cross-entropy loss.
- Object detection uses a non-max suppression technique.
    - First, it gets rid of all the bounding boxes for which the objectness score is below some threshold.
    - Then, it finds the remaining bounding box with the highest objectness score and gets rid of all other remaining boxes that overlap significantly with it.
    - It repeats the last step until there are no more bounding boxes to eliminate.
    - This approach works well but requires running the CNN several times, which slows down the algorithm.
    - This can be fixed with a fully convolutional network (FCN)

Fully Convolutional Networks

- FCNs are useful for semantic segmentation: the task of classifying every pixel in an image according to the class of the object it belongs to.
- We can replace the dense layers at the top of a CNN with convolutional layers.
    - For this to work, the filter size must be equal to the size of the input feature maps, and we must use "valid" padding.
    - While a dense layer expects a specific input size, a convolutional layer can process images of any size.
        - However, convolutional layers require their inputs to have a specific number of channels.
    - Since an FCN contains only convolutional layers, we can train and execute images of any size.
- We can convert dense layers to convolutional layers without having to retrain the CNN. We simply copy the dense layer's weights and provide them to the convolutional layer.
- FCNs are networks that only look at the input image once. *These are You Only Look Once* (YOLO) architectures.

You Only Look Once

- YOLO architectures are extremely fast.
    - For each grid cell, YOLO only considers objects whose bounding box center lies within that cell.
    - It outputs two bounding boxes for each grid cell (instead of just one), which allows the model to handle cases where two objects are so close to each other that their bounding box centers lie within the same cell. Each bounding box also comes with its own objectness score.
    - YOLO also outputs a class probability distribution for each grid cell, predicting 20 class probabilities per grid cell.
    - Note that the model predicts one class probability distribution per grid cell, not per bounding box.
- YOLO architectures are evaluated using mean average precision (MAP).

    - To get a fair idea of the model's performance, we compute the maximum precision we can get with at least 0% recall, then 10% recall, 20%, and so on up to 100%, and then calculate the mean of these maximum precisions.
    - This gives us the average precision (AP) metric. Then, we compute the mean of all the AP metrics of each class, giving us the MAP metric.
    - Object detection algorithms also use the IoU metric.

Object Tracking

- Object tracking tries to identify moving or changing objects.
- One of the most popular object-tracking algorithms is DeepSORT.
    - It uses Kalman filters to estimate the most likely current position of an object given prior detections and assuming that objects tend to move at a constant speed.
    - It uses a deep learning model to measure the resemblance between new detections and existing tracked objects.
    - Also, it uses the Hungarian algorithm to map new detections to existing tracked objects.
        - Kalman filters alone cannot distinguish unique features from the objects. The Hungarian algorithm can map new detections to the objects.

Semantic Segmentation

- In semantic segmentation, each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.)
- When images move through a regular CNN, they gradually lose their spatial resolution.
    - It may identify the general location of an object, but it won't be precise.
- One solution is to use FCNs.
    - We add a single upsampling layer that multiples the resolution of the image in a regular CNN.
    - To upsample a layer, we can use a transposed convolutional layer.
        - Essentially, we stretch the image by inserting empty rows and columns and performing regular convolution.
        - We can use Keras's **Conv2DTranspose()** layer.
- Some other types of convolutional layers in Keras:
    - **tf.keras.layers.Conv1D**: a convolutional layer for 1D inputs, like time series or text.
    - **tf.keras.layers.Conv3D**: a convolutional layer for 3D inputs, like 3D PET scans.
    - **dilation_rate**: this hyperparameter creates an à-trous convolutional layer (a layer with holes).
- Instance segmentation is similar to semantic segmentation, but each object is distinguished from the others (e.g., it identifies each individual bicycle).
    - Kaiming He proposed the Mask R-CNN architecture to produce a pixel mask for each bounding box.