**1. What are the advantages of a CNN over a fully connected DNN for image classification?**

* **Parameter efficiency**: CNNs use fewer parameters due to weight sharing in convolutional layers.
* **Spatial hierarchy**: CNNs capture spatial hierarchies (e.g., edges, textures, objects) through convolutional and pooling layers.
* **Translation invariance**: CNNs are invariant to small translations in the input image, making them robust to shifts in object positions.
* **Local connectivity**: CNNs focus on local regions of the image, which is more efficient and effective for image data compared to fully connected layers.

**2. Consider a CNN composed of three convolutional layers, each with 3 × 3 kernels, a stride of 2, and "same" padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200 × 300 pixels.**

* **Total number of parameters**:
  + **First layer**: Input has 3 channels (RGB), output has 100 feature maps.
    - Parameters = (3 × 3 × 3 + 1) × 100 = 2,800.
  + **Second layer**: Input has 100 feature maps, output has 200 feature maps.
    - Parameters = (3 × 3 × 100 + 1) × 200 = 180,200.
  + **Third layer**: Input has 200 feature maps, output has 400 feature maps.
    - Parameters = (3 × 3 × 200 + 1) × 400 = 720,400.
  + **Total parameters** = 2,800 + 180,200 + 720,400 = **903,400**.
* **RAM requirements**:
  + **Single instance**:
    - Input size = 200 × 300 × 3 = 180,000 pixels.
    - Output size after each layer:
      * Layer 1: 100 × 100 × 150 = 1,500,000.
      * Layer 2: 200 × 50 × 75 = 750,000.
      * Layer 3: 400 × 25 × 38 = 380,000.
    - Total RAM = (180,000 + 1,500,000 + 750,000 + 380,000) × 4 bytes (32-bit float) = **11.24 MB**.
  + **Mini-batch of 50 images**:
    - Total RAM = 11.24 MB × 50 = **562 MB**.

**3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?**

* **Reduce batch size**: Use a smaller mini-batch to decrease memory usage.
* **Use mixed precision**: Train with 16-bit floating-point numbers instead of 32-bit.
* **Freeze layers**: Freeze some layers during training to reduce the number of trainable parameters.
* **Use gradient checkpointing**: Save memory by recomputing intermediate activations during the backward pass.
* **Use a simpler model**: Reduce the number of layers or feature maps.

**4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?**

* **Dimensionality reduction**: Max pooling reduces the spatial dimensions of the feature maps, which helps control overfitting and computational cost.
* **Translation invariance**: Max pooling provides some degree of translation invariance, making the model more robust to small shifts in the input.
* **No additional parameters**: Max pooling does not introduce additional trainable parameters, unlike convolutional layers.

**5. When would you want to add a local response normalization layer?**

* Local response normalization (LRN) is used to normalize the activations of neurons in a local region, enhancing contrast and improving generalization. It was popular in older architectures like AlexNet but is rarely used in modern CNNs, as batch normalization has proven more effective.

**6. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, and Xception?**

* **AlexNet**:
  + Deeper architecture (8 layers vs. 5 in LeNet-5).
  + ReLU activation function for faster training.
  + Dropout for regularization.
  + Data augmentation and GPU acceleration.
* **GoogLeNet**:
  + Inception modules with parallel convolutions and pooling.
  + Efficient use of parameters and computational resources.
* **ResNet**:
  + Residual connections to enable very deep networks (e.g., 152 layers).
  + Addresses the vanishing gradient problem.
* **SENet**:
  + Squeeze-and-Excitation (SE) blocks to model channel-wise dependencies.
* **Xception**:
  + Depthwise separable convolutions for efficient computation.

**7. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?**

* **Fully convolutional network (FCN)**: A CNN without fully connected layers, where all layers are convolutional. FCNs are used for tasks like semantic segmentation.
* **Converting dense layers**:
  + Replace a dense layer with a 1 × 1 convolutional layer with the same number of filters as the dense layer's output units.
  + For example, a dense layer with 4096 units can be replaced with a 1 × 1 convolutional layer with 4096 filters.

**8. What is the main technical difficulty of semantic segmentation?**

* The main difficulty is **preserving spatial resolution** while capturing high-level semantic information. This requires balancing the use of pooling layers (for feature extraction) and upsampling layers (for restoring resolution).

**9. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.**

* Example code:

from tensorflow.keras import layers, models

model = models.Sequential([

layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(128, activation='relu'),

layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_val, y\_val))

**10. Use transfer learning for large image classification:**

**a. Create a training set containing at least 100 images per class.**

* Use own images or an existing dataset (e.g., CIFAR-10, ImageNet).

**b. Split it into a training set, a validation set, and a test set.**

* Example:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(images, labels, test\_size=0.2)

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_train, y\_train, test\_size=0.2)

**c. Build the input pipeline, including preprocessing and data augmentation.**

* Example:

from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rescale=1./255, rotation\_range=20, width\_shift\_range=0.2, height\_shift\_range=0.2, horizontal\_flip=True)

train\_generator = datagen.flow(X\_train, y\_train, batch\_size=32)

**d. Fine-tune a pretrained model on this dataset.**

* Example:

from tensorflow.keras.applications import VGG16

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

base\_model.trainable = False

model = models.Sequential([

base\_model,

layers.Flatten(),

layers.Dense(256, activation='relu'),

layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(train\_generator, epochs=10, validation\_data=(X\_val, y\_val))