**1. Advantages of a CNN over a Fully Connected DNN for Image Classification**

✅ **Translation Invariance** – CNNs recognize patterns regardless of their location.  
✅ **Parameter Efficiency** – CNNs use **convolutions and weight sharing**, reducing parameters significantly.  
✅ **Local Feature Extraction** – CNNs **detect edges, textures, and shapes** in early layers before combining them into complex patterns.  
✅ **Hierarchical Learning** – Deep CNNs learn **low-level features (edges) → mid-level features (textures) → high-level features (objects).**  
✅ **Better Generalization** – Fewer parameters **reduce overfitting**, making CNNs more robust to variations.

**2. Parameter Count & RAM Calculation for the Given CNN**

**Step 1: Compute Parameters for Each Convolutional Layer**

Each **3×3 kernel** applies to every feature map with a **stride of 2** and **"same" padding**, meaning the output **size is halved** at each layer.

**Given Layers:**

1. **Layer 1:** 3 input channels (RGB) → 100 filters (output channels). (3×3×3+1)×100=2800(3 \times 3 \times 3 + 1) \times 100 = 2800
2. **Layer 2:** 100 input channels → 200 filters. (3×3×100+1)×200=180200(3 \times 3 \times 100 + 1) \times 200 = 180200
3. **Layer 3:** 200 input channels → 400 filters. (3×3×200+1)×400=720400(3 \times 3 \times 200 + 1) \times 400 = 720400

🔹 **Total parameters:**

2800+180200+720400=903400 parameters2800 + 180200 + 720400 = 903400 \text{ parameters}

**Step 2: Memory Usage in Prediction & Training**

**RAM required per parameter (32-bit float):**

903400×4=3.61 MB903400 \times 4 = 3.61 \text{ MB}

🔹 **Feature Map Sizes:**

1. Input: **200×300×3**
2. After Layer 1: **100 feature maps of size 100×150**
3. After Layer 2: **200 feature maps of size 50×75**
4. After Layer 3: **400 feature maps of size 25×37**

**Memory per instance (excluding gradients & activations):**

(100×100×150+200×50×75+400×25×37)×4≈3.58 MB(100 \times 100 \times 150 + 200 \times 50 \times 75 + 400 \times 25 \times 37) \times 4 \approx 3.58 \text{ MB}

For a **mini-batch of 50 images**, multiply by 50:

3.58×50=179 MB3.58 \times 50 = 179 \text{ MB}

**3. Solving GPU Memory Issues During CNN Training**

🔹 **Reduce Batch Size** – Most effective method to free up memory.  
🔹 **Use Mixed Precision Training** – Uses **16-bit (FP16) floats instead of 32-bit (FP32)**, cutting memory usage in half.  
🔹 **Gradient Checkpointing** – Saves memory by **recomputing activations** instead of storing them.  
🔹 **Reduce Model Complexity** – Use **smaller kernel sizes** or fewer feature maps.  
🔹 **Use TensorFlow's tf.data API** – Prefetching & caching speeds up loading, reducing memory overhead.

**4. Why Use Max Pooling Instead of a Convolutional Layer with Stride?**

✅ **Reduces Spatial Dimensions** → Controls computation & memory.  
✅ **Translation Invariance** → Preserves important features while ignoring minor shifts.  
✅ **Prevents Overfitting** → Removes noise while keeping dominant patterns.  
✅ **Faster Computation** → Reduces feature map size without learning extra parameters.

**5. When to Use Local Response Normalization (LRN)?**

📌 LRN is useful in **early CNN architectures** (e.g., AlexNet) for:  
✅ **Enhancing contrast** between strong & weak activations.  
✅ **Encouraging competition** between neurons.  
🚫 **Modern CNNs rarely use LRN** → Batch normalization (BatchNorm) is more effective.

**6. Key Innovations in CNN Architectures**

📌 **AlexNet (2012)** → **First deep CNN to win ImageNet Challenge**.

* **ReLU activations** (instead of sigmoid/tanh).
* **Dropout** for regularization.
* **Overlapping max pooling**.

📌 **GoogLeNet (2014, Inception)**

* **Inception modules** → Multi-scale feature extraction.
* **1×1 convolutions** for dimension reduction.

📌 **ResNet (2015)**

* **Residual connections (skip connections)** → Solves vanishing gradient problem.
* **Very deep networks** (up to 1000+ layers).

📌 **SENet (2017)**

* **Squeeze-and-Excitation (SE) blocks** → Improves feature recalibration.

📌 **Xception (2017)**

* **Depthwise Separable Convolutions** → More efficient than regular convolutions.

**7. Fully Convolutional Networks (FCNs) & Converting Dense Layers**

📌 **FCNs** remove fully connected layers, replacing them with **1×1 convolutions** for segmentation tasks.  
📌 **Convert Dense Layer → Convolution:**

* Dense layer with **N neurons** → 1×1 convolution with **N filters**.

🔹 Example:

dense\_layer = tf.keras.layers.Dense(256)(x) # Fully connected

conv\_layer = tf.keras.layers.Conv2D(256, (1,1))(x) # 1x1 convolution

**8. Main Challenge in Semantic Segmentation**

📌 **Precise boundary detection** – Small details like object edges are hard to segment.  
📌 **Trade-off between spatial resolution & depth** – Deeper networks reduce feature map size.  
📌 **Handling multiple object scales** – Objects vary in size across images.

**Solutions:**  
✅ **Use U-Net or DeepLabV3+** for high-resolution segmentation.  
✅ **Use Dilated (Atrous) Convolutions** to increase receptive field.

**9. Build a CNN for MNIST (Example Code)**

import tensorflow as tf

from tensorflow.keras import layers, models

# Build the CNN

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')

])

# Compile and train

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

**10. Transfer Learning for Large Image Classification**

🔹 **(a) Create a dataset** → Use **TensorFlow Datasets** or **custom images** (100+ per class).  
🔹 **(b) Split into training (80%), validation (10%), test (10%)**.  
🔹 **(c) Build input pipeline** (resize images, normalize, augment).  
🔹 **(d) Fine-tune a Pretrained Model** like MobileNetV2:

import tensorflow as tf

from tensorflow.keras.applications import MobileNetV2

from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Load Pretrained Model

base\_model = MobileNetV2(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

base\_model.trainable = False # Freeze base layers

# Build new model on top

model = tf.keras.Sequential([

base\_model,

tf.keras.layers.GlobalAveragePooling2D(),

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

tf.keras.layers.Dense(3, activation='softmax') # 3 classes

])

# Compile & Train

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