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

What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? What about when training on a mini-batch of 50 images?

1. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
2. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
3. When would you want to add a local response normalization layer?
4. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, and Xception?
5. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?
6. What is the main technical difficulty of semantic segmentation?
7. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.
8. Use transfer learning for large image classification, going through these steps:
   1. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can use an existing dataset (e.g., from TensorFlow Datasets).
   2. Split it into a training set, a validation set, and a test set.
   3. Build the input pipeline, including the appropriate preprocessing operations, and optionally add data augmentation.
   4. Fine-tune a pretrained model on this dataset.

Answer:

1. The advantages of a convolutional neural network (CNN) over a fully connected DNN for image classification include:

* Parameter sharing: in a CNN, each filter is applied to the entire input image, which means that the number of parameters is greatly reduced compared to a fully connected DNN.
* Translation invariance: by using convolutional layers with small filters, a CNN is able to detect patterns regardless of their location in the image.
* Hierarchical feature extraction: a CNN typically consists of multiple layers, each of which learns increasingly complex and abstract features.

1. The total number of parameters in the CNN is:

* First convolutional layer: 3x3x3 (kernel size x number of input channels) x 100 (number of filters) = 2,700
* Second convolutional layer: 3x3x100 (kernel size x number of input channels) x 200 (number of filters) = 180,000
* Third convolutional layer: 3x3x200 (kernel size x number of input channels) x 400 (number of filters) = 1,152,000
* Dense layer: (number of inputs) x (number of outputs) = (50,400 (output size of last conv layer)) x 10 (number of classes) = 504,000 The total number of parameters in the CNN is 1,838,700. If we are using 32-bit floats, this network will require at least 7.35 MB of RAM when making a prediction for a single instance. When training on a mini-batch of 50 images, it will require at least 367.5 MB of RAM.

1. If your GPU runs out of memory while training a CNN, some things you could try to solve the problem include:

* Reducing the batch size
* Reducing the image size
* Reducing the depth or width of the network
* Using mixed precision training
* Using gradient checkpointing

1. Max pooling layers are used to downsample the feature maps, reducing the dimensionality of the input and making the model less prone to overfitting. A convolutional layer with the same stride would not reduce the dimensionality.
2. Local response normalization (LRN) layers are used to enhance the contrast of features within the same feature map. They are typically used in the early layers of a CNN.
3. The main innovations in AlexNet, compared to LeNet-5, include:

* The use of ReLU activation functions, which helped alleviate the vanishing gradients problem
* The use of dropout regularization, which helped prevent overfitting
* The use of data augmentation, which helped increase the size of the training set
* The use of two GPUs, which allowed the model to be trained more quickly The main innovations in GoogLeNet include:
* The use of inception modules, which allowed the model to learn both 1x1 and 3x3 convolutions simultaneously
* The use of global average pooling, which allowed the model to reduce the number of parameters and increase efficiency
* The use of auxiliary classifiers, which helped alleviate the vanishing gradients problem The main innovation in ResNet is the use of residual connections, which allowed the model to learn skip connections and avoid the degradation problem. The main innovation in SENet is the use of squeeze-and-excitation modules, which allowed the model to learn channel-wise feature re-calibration. The main innovation in Xception is the use of depthwise separable convolutions, which allowed the model to learn depthwise and pointwise convolutions separately.

7.

* A fully convolutional network (FCN) is a type of neural network architecture that is designed to perform end-to-end image segmentation. Unlike traditional convolutional neural networks (CNNs), which are typically used for image classification and object detection, FCNs can produce pixel-level segmentation maps.
* To convert a dense layer into a convolutional layer, we can first reshape the dense layer's weight matrix to have an additional dimension for the number of output channels. Then, we can define a convolutional layer with a kernel size equal to the dimensions of the original dense layer and a number of output channels equal to the number of neurons in the dense layer. Finally, we can initialize the convolutional layer's weights with the reshaped weight matrix from the dense layer. This can be useful when we want to use a pre-trained model for image segmentation, and need to replace the fully connected layers with convolutional layers to preserve spatial information.
  + 1. The main technical difficulty of semantic segmentation is that it requires predicting a label for every pixel in the image, which can result in a very large number of parameters and can be computationally expensive. Additionally, it can be challenging to handle variations in image size and shape, as well as variability in the appearance of objects.
    2. Here's an example of building a simple CNN from scratch to achieve high accuracy on the MNIST dataset:

import tensorflow as tf

from tensorflow.keras import layers

# Load MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

# Reshape data to 4D tensor

x\_train = x\_train.reshape(-1, 28, 28, 1).astype("float32") / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1).astype("float32") / 255.0

# Build the model

model = tf.keras.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(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train, validation\_data=(x\_test, y\_test), epochs=10, batch\_size=32)

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print('Test accuracy:', test\_acc)

1. Here's an example of using transfer learning for large image classification:

a. Download a pre-trained model, such as InceptionV3, and create a training set with at least 100 images per class.

b. Split the data into training, validation, and test sets.

c. Build the input pipeline, including the appropriate preprocessing operations, such as resizing images to the expected input size of the pre-trained model, and optionally add data augmentation.

import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Define the data directories

train\_dir = 'path/to/train'

val\_dir = 'path/to/val'

test\_dir = 'path/to/test'

# Define the preprocessing operations

preprocess = tf.keras.applications.inception\_v3.preprocess\_input

# Define the data generators

train\_datagen = ImageDataGenerator(preprocessing\_function=preprocess,

horizontal\_flip=True,

vertical\_flip=True)

val\_datagen = ImageDataGenerator(preprocessing\_function=preprocess)

test\_datagen = ImageDataGenerator(preprocessing\_function=preprocess)

train\_generator = train\_datagen.flow\_from\_directory(train\_dir,

target\_size=(299, 299),

batch\_size=32)

val\_generator = val\_datagen.flow\_from\_directory(val\_dir,

target\_size=(299, 299),

batch\_size=32)

test\_generator = test\_datagen.flow\_from\_directory(test\_dir,

target\_size=(299, 299),

batch\_size=32)

* 1. import tensorflow as tf

from tensorflow.keras import layers, models

# Load the pre-trained model and remove the last layer

base\_model = tf.keras.applications.InceptionV3(include\_top=False, weights='imagenet', input\_shape=(299, 299, 3))