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

🡪 Convolutional Neural Networks (CNNs) have several advantages over fully connected Deep Neural Networks (DNNs) for image classification. Here are some of the main advantages:

1. Parameter efficiency: CNNs use shared weights and biases for the convolutional layers, which greatly reduces the number of parameters compared to fully connected DNNs. This makes CNNs more efficient for processing large images and reduces the risk of overfitting.

2. Translation invariance: CNNs are designed to recognize patterns and features within an image regardless of their location, making them translation invariant. This is achieved through the use of convolutional layers that slide filters over the image to detect patterns in a local area. This property makes CNNs well-suited for image classification tasks where the object of interest can be anywhere in the image.

3. Hierarchical feature learning: CNNs use multiple layers of feature extraction to learn hierarchical representations of images. This allows the network to learn high-level concepts, such as object categories, from low-level features, such as edges and corners. In contrast, fully connected DNNs treat all input features as independent and do not capture the spatial relationships between them.

4. Robustness to input variations: CNNs are designed to be robust to variations in the input, such as changes in lighting, rotation, and scale. This is achieved through the use of techniques such as pooling, which reduces the sensitivity of the network to small changes in the input.

1. 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?

🡪 To calculate the total number of parameters in the CNN, we need to count the number of parameters in each layer and then sum them up. The number of parameters in a convolutional layer depends on the number of filters, the size of the filters, and the number of input channels. The formula for calculating the number of parameters in a convolutional layer is:

number of parameters = (size of filter \* number of input channels + 1) \* number of filters

Using the given information, we can calculate the number of parameters in each convolutional layer:

The first convolutional layer has 3x3x3x100 + 100 = 2,800 parameters

The second convolutional layer has 3x3x100x200 + 200 = 180,200 parameters

The third convolutional layer has 3x3x200x400 + 400 = 1,160,400 parameters

Therefore, the total number of parameters in the CNN is 2,800 + 180,200 + 1,160,400 = 1,343,400 parameters.

To calculate the amount of RAM required for prediction or training, we need to consider the size of the input and output tensors and the data type being used. Assuming we are using 32-bit floats (4 bytes), the size of the input tensor for a single image is 200x300x3 = 180,000 bytes. The size of the output tensor for a single image is (200/8)x(300/8)x400x4 = 6,000,000 bytes. Therefore, the total RAM required for prediction for a single instance is approximately 6,180,000 bytes (6.18 MB).

If we are training on a mini-batch of 50 images, the total RAM required would be 50 times the RAM required for a single instance, which is approximately 309 MB. However, this is just the memory required for storing the input and output tensors during one forward pass of the network. The actual amount of memory required for training will depend on the batch size, the size of the model, the optimizer being used, and other factors.

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

🡪 1. Decrease batch size: One of the most effective ways to reduce memory usage is to decrease the batch size. This will reduce the number of samples being processed simultaneously, which in turn reduces the amount of memory required.

2. Reduce input size: If the input images are very large, downsampling them to a smaller size can also reduce the memory requirements. However, this may come at the cost of reduced performance or accuracy.

3. Use mixed precision training: Using mixed precision training, where the weights and activations are stored using 16-bit floating point numbers instead of 32-bit, can significantly reduce memory usage without sacrificing accuracy.

4. Reduce model complexity: Another way to reduce memory usage is to reduce the complexity of the model. This can be done by reducing the number of layers, filters, or neurons in the network, or by using a smaller kernel size.

5. Use gradient checkpointing: Gradient checkpointing is a technique that trades off computation time for memory usage. Instead of storing all the intermediate activations during the forward pass for use during the backward pass, only a subset of the activations are stored. This reduces memory usage, but increases computation time during the backward pass.

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

🡪 Max pooling layers and convolutional layers with the same stride both reduce the spatial resolution of the feature maps. However, there are some reasons why one might want to use max pooling instead of convolution:

1. Non-linearity: Max pooling introduces a non-linear activation function that can help the network learn more complex features. In contrast, a convolutional layer with the same stride only applies a linear transformation to the input.

2. Translation invariance: Max pooling is more translation invariant than convolution with stride. This means that the features learned by the network are less sensitive to small shifts in the input image, which can improve the robustness of the network to small variations in the data.

3. Computational efficiency: Max pooling is computationally more efficient than convolution with the same stride because it does not require any learnable parameters.

4. Reduce overfitting: Max pooling can be used to reduce overfitting by reducing the spatial resolution of the feature maps and introducing some degree of spatial invariance.

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

🡪 A local response normalization (LRN) layer can be added to a CNN to improve its generalization performance by promoting competition among feature maps. The main purpose of LRN is to normalize the response of a neuron by dividing it by the sum of squares of its neighboring activations.

Here are some scenarios when one may want to add a local response normalization layer to a CNN:

1. Large-scale datasets: LRN is more effective on large-scale datasets as it helps the network generalize better and reduce overfitting. When working with large-scale datasets, adding an LRN layer can improve the performance of the network.

2. Convolutional layers with large receptive fields: When the receptive field of a convolutional layer is large, there is a high chance that some feature maps will respond strongly to stimuli that are irrelevant to the target object. In such cases, LRN can help the network suppress the irrelevant feature maps and promote the relevant ones.

3. Networks with multiple convolutional layers: When working with CNNs with multiple convolutional layers, adding an LRN layer can help the network learn more robust features by encouraging competition among feature maps. This can help prevent the network from overfitting to the training data.

4. Image classification tasks: LRN has been shown to be particularly effective in improving the performance of CNNs on image classification tasks. It can help the network learn more discriminative features, which in turn can improve its classification accuracy.

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

🡪 AlexNet:

-AlexNet was one of the first successful deep convolutional neural networks for image classification.

-It had 5 convolutional layers and 3 fully connected layers.

-AlexNet used Rectified Linear Units (ReLU) activation function instead of the traditional sigmoid activation function used in LeNet-5.

-It also used data augmentation techniques such as cropping and flipping to reduce overfitting and increase the size of the training dataset.

-AlexNet used dropout regularization to further reduce overfitting.

-AlexNet was trained on two GPUs simultaneously, which reduced the training time significantly.

GoogLeNet:

-GoogLeNet introduced the concept of Inception modules, which allowed for the efficient use of multiple filter sizes in a single layer.

-It had 22 layers and was the first network to exceed human-level performance on the ImageNet classification task.

-GoogLeNet also used global average pooling instead of fully connected layers, which significantly reduced the number of parameters in the network.

-It used a 1x1 convolutional layer to reduce the dimensionality of the feature maps before applying more computationally expensive convolutional layers.

ResNet:

-ResNet introduced the concept of residual blocks, which allowed for the training of very deep networks without suffering from the problem of vanishing gradients.

-It had up to 152 layers and achieved state-of-the-art performance on several computer vision tasks.

-ResNet used skip connections to connect earlier layers directly to later layers, which helped to propagate gradients more effectively during training.

-It also used batch normalization to accelerate training and improve generalization.

SENet:

-SENet introduced the concept of Squeeze-and-Excitation (SE) blocks, which allowed the network to adaptively recalibrate the importance of each feature map based on the global context of the image.

-It achieved state-of-the-art performance on several computer vision tasks.

-SENet used a global average pooling layer to reduce the dimensionality of the feature maps before applying the SE blocks.

-It also used residual connections to help propagate gradients more effectively during training.

Xception:

-Xception introduced the concept of depthwise separable convolutions, which separate the spatial filtering and channel-wise filtering operations.

-It had fewer parameters and was more computationally efficient than other state-of-the-art models while achieving comparable or better performance.

-Xception used residual connections and batch normalization to accelerate training and improve generalization.

-It also used global average pooling instead of fully connected layers to reduce the number of parameters in the network.

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

🡪 A fully convolutional network (FCN) is a neural network architecture that consists entirely of convolutional layers, and no fully connected layers. FCNs are commonly used in image segmentation tasks, where the output is a pixel-wise classification map instead of a single label.

To convert a dense layer into a convolutional layer, we need to ensure that the output shape of the convolutional layer is the same as the output shape of the dense layer. We can achieve this by reshaping the weights of the dense layer into the shape of a convolutional kernel.

For example, suppose we have a dense layer with 512 units that takes as input a feature map of size 7x7x256, and we want to convert it into a convolutional layer with a 3x3 kernel. We can do this as follows:

1. Reshape the weight matrix of the dense layer to have shape (3, 3, 256, 512). This will create a 3x3 convolutional kernel with 256 input channels and 512 output channels.

2. Reshape the output of the previous convolutional layer to have shape (batch\_size, 7, 7, 256). This is the same shape as the input to the dense layer.

3. Apply the convolution operation with the reshaped kernel to the output of the previous convolutional layer. The output of this operation will have shape (batch\_size, 5, 5, 512), which is the same as the output of the dense layer.

1. What is the main technical difficulty of semantic segmentation?

🡪 The main technical difficulty of semantic segmentation is to accurately classify each pixel in an image into one of the pre-defined classes (e.g. person, car, tree, sky, etc.). This is a challenging task because it requires the network to learn both the spatial information and the semantic information of the image.

Spatial information refers to the location and arrangement of objects in the image, which is important for distinguishing between adjacent objects or objects that overlap. Semantic information refers to the high-level concepts and meaning of the objects in the image, which is important for distinguishing between objects of the same type (e.g. different types of trees).

Another difficulty is handling the class imbalance problem, where some classes have much fewer pixels than others. This can lead to a bias towards the majority classes and poor performance on minority classes.

To overcome these difficulties, various techniques have been proposed, including the use of encoder-decoder architectures, skip connections, multi-scale features, dilated convolutions, and data augmentation.

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

🡪 import tensorflow as tf

from tensorflow.keras import layers

# Load the MNIST dataset

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

# Normalize the pixel values to the range [0, 1]

x\_train = x\_train.astype("float32") / 255.0

x\_test = x\_test.astype("float32") / 255.0

# Add a channel dimension to the images

x\_train = x\_train[..., tf.newaxis]

x\_test = x\_test[..., tf.newaxis]

# Define the model architecture

model = tf.keras.Sequential([

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

layers.MaxPooling2D(),

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

layers.MaxPooling2D(),

layers.Flatten(),

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

layers.Dropout(0.5),

layers.Dense(10)

])

# Compile the model

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

metrics=['accuracy'])

# Train the model

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

# Evaluate the model on the test set

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

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

1. 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).

🡪 import tensorflow\_datasets as tfds

(train\_ds, val\_ds, test\_ds), info = tfds.load('cats\_vs\_dogs',

split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],

with\_info=True,

as\_supervised=True)

IMG\_SIZE = 224

BATCH\_SIZE = 32

def preprocess\_image(image, label):

image = tf.image.resize(image, (IMG\_SIZE, IMG\_SIZE))

image = tf.cast(image, tf.float32) / 255.0

image = tf.image.random\_flip\_left\_right(image)

image = tf.image.random\_crop(image, size=(IMG\_SIZE, IMG\_SIZE, 3))

return image, label

train\_ds = train\_ds.map(preprocess\_image).shuffle(1000).batch(BATCH\_SIZE)

val\_ds = val\_ds.map(preprocess\_image).batch(BATCH\_SIZE)

test\_ds = test\_ds.map(preprocess\_image).batch(BATCH\_SIZE)

from tensorflow.keras.layers import Dense, GlobalAveragePooling2D

from tensorflow.keras.models import Model

from tensorflow.keras.applications.inception\_v3 import InceptionV3

base\_model = InceptionV3(include\_top=False, weights='imagenet', input\_shape=(IMG\_SIZE, IMG\_SIZE, 3))

for layer in base\_model.layers:

layer.trainable = False

x = GlobalAveragePooling2D()(base\_model.output)

x = Dense(2, activation='softmax')(x)

model = Model(inputs=base\_model.input, outputs=x)

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

history = model.fit(train\_ds, epochs=5, validation\_data=val\_ds)

test\_loss, test\_acc = model.evaluate(test\_ds)

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

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

🡪 (train\_ds, val\_ds, test\_ds), info = tfds.load('cats\_vs\_dogs',

split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],

with\_info=True,

as\_supervised=True)

* 1. Build the input pipeline, including the appropriate preprocessing operations, and optionally add data augmentation.

🡪 import tensorflow as tf

import tensorflow\_datasets as tfds

# Load the dataset

(train\_ds, val\_ds, test\_ds), info = tfds.load('cats\_vs\_dogs',

split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],

with\_info=True,

as\_supervised=True)

# Define preprocessing functions

IMG\_SIZE = 224

NUM\_CLASSES = 2

def preprocess\_image(image, label):

# Resize the image to the input size of the model

image = tf.image.resize(image, (IMG\_SIZE, IMG\_SIZE))

# Convert the pixel values to the range [0, 1]

image = tf.cast(image, tf.float32) / 255.0

return image, label

def augment\_image(image, label):

# Randomly flip the image horizontally

image = tf.image.random\_flip\_left\_right(image)

# Randomly adjust the brightness of the image

image = tf.image.random\_brightness(image, max\_delta=0.1)

return image, label

# Create the training dataset

train\_ds = train\_ds.shuffle(10000)

train\_ds = train\_ds.map(preprocess\_image, num\_parallel\_calls=tf.data.AUTOTUNE)

train\_ds = train\_ds.map(augment\_image, num\_parallel\_calls=tf.data.AUTOTUNE)

train\_ds = train\_ds.batch(batch\_size=32)

train\_ds = train\_ds.prefetch(tf.data.AUTOTUNE)

# Create the validation dataset

val\_ds = val\_ds.map(preprocess\_image, num\_parallel\_calls=tf.data.AUTOTUNE)

val\_ds = val\_ds.batch(batch\_size=32)

val\_ds = val\_ds.prefetch(tf.data.AUTOTUNE)

# Create the test dataset

test\_ds = test\_ds.map(preprocess\_image, num\_parallel\_calls=tf.data.AUTOTUNE)

test\_ds = test\_ds.batch(batch\_size=32)

test\_ds = test\_ds.prefetch(tf.data.AUTOTUNE)

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

🡪 import tensorflow as tf

import tensorflow\_datasets as tfds

from tensorflow.keras.applications import MobileNetV2

# Load the dataset

(train\_ds, val\_ds, test\_ds), info = tfds.load('cats\_vs\_dogs',

split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],

with\_info=True,

as\_supervised=True)

# Define preprocessing functions

IMG\_SIZE = 224

NUM\_CLASSES = 2

def preprocess\_image(image, label):

# Resize the image to the input size of the model

image = tf.image.resize(image, (IMG\_SIZE, IMG\_SIZE))

# Convert the pixel values to the range [-1, 1]

image = tf.cast(image, tf.float32) / 127.5 - 1.0

return image, label

# Apply preprocessing to the datasets

train\_ds = train\_ds.map(preprocess\_image).shuffle(1000).batch(32).prefetch(tf.data.AUTOTUNE)

val\_ds = val\_ds.map(preprocess\_image).batch(32).prefetch(tf.data.AUTOTUNE)

test\_ds = test\_ds.map(preprocess\_image).batch(32)

# Load the pre-trained MobileNetV2 model

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

# Freeze the base model's layers

for layer in base\_model.layers:

layer.trainable = False

# Add a new classification layer on top of the base model

global\_average\_layer = tf.keras.layers.GlobalAveragePooling2D()(base\_model.output)

output\_layer = tf.keras.layers.Dense(NUM\_CLASSES, activation='softmax')(global\_average\_layer)

# Create the fine-tuned model

model = tf.keras.models.Model(inputs=base\_model.input, outputs=output\_layer)

# Compile the model

model.compile(optimizer=tf.keras.optimizers.Adam(), loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train the model on the training set

history = model.fit(train\_ds, epochs=10, validation\_data=val\_ds)

# Evaluate the model on the test set

loss, accuracy = model.evaluate(test\_ds)

print(f'Test loss: {loss}, Test accuracy: {accuracy}')