1. What are the advantages of a CNN for image classification over a completely linked DNN?

Ans: Convolutional Neural Networks (CNNs) are specifically designed for image classification tasks and offer several advantages over fully connected Deep Neural Networks (DNNs), which are more general-purpose models.

Here are some advantages of CNNs over DNNs for image classification:

Spatial Hierarchies: CNNs can capture spatial hierarchies by learning local patterns in an image and combining them to form higher-level representations. This is achieved through the use of convolutional layers, which apply filters to different parts of the image and learn to recognize specific features.

Parameter Efficiency: CNNs are more parameter-efficient than DNNs, as they reuse the same set of weights across different parts of an image. This allows for faster training times and reduces the risk of overfitting.

Translation Invariance: CNNs are able to recognize objects in images regardless of their position or orientation, due to their ability to use the same filters across different parts of the image. This makes them more robust to variations in input data.

Feature Extraction: CNNs can automatically learn and extract features from raw image data, which is not possible with DNNs. This reduces the need for manual feature engineering, which can be time-consuming and difficult.

Overall, CNNs are better suited for image classification tasks due to their ability to capture spatial hierarchies, parameter efficiency, translation invariance, and feature extraction capabilities.

2. Consider a CNN with three convolutional layers, each of which has three kernels, a stride of two, and SAME padding. The bottom layer generates 100 function maps, the middle layer 200, and the top layer 400. RGB images with a size of 200 x 300 pixels are used as input. How many criteria does the CNN have in total? How much RAM would this network need when making a single instance prediction if we're using 32-bit floats? What if you were to practice on a batch of 50 images?

Ans: To calculate the total number of parameters in the CNN, we need to compute the number of parameters in each layer and then add them up.

For each convolutional layer, the number of parameters can be calculated as follows:

Number of parameters in each kernel = kernel size \* number of input feature maps \* number of output feature maps

Number of bias parameters = number of output feature maps

The kernel size is not given, so let's assume it to be 3x3.

For the bottom layer:

Number of input feature maps = 3 (RGB channels)

Number of output feature maps = 100

Number of parameters in each kernel = 3 \* 3 \* 3 \* 100 = 2700

Number of bias parameters = 100

Total number of parameters in the bottom layer = (2700 + 100) \* 3 = 8400

For the middle layer:

Number of input feature maps = 100 (output feature maps of the bottom layer)

Number of output feature maps = 200

Number of parameters in each kernel = 3 \* 3 \* 100 \* 200 = 180,000

Number of bias parameters = 200

Total number of parameters in the middle layer = (180000 + 200) \* 3 = 540600

For the top layer:

Number of input feature maps = 200 (output feature maps of the middle layer)

Number of output feature maps = 400

Number of parameters in each kernel = 3 \* 3 \* 200 \* 400 = 1,440,000

Number of bias parameters = 400

Total number of parameters in the top layer = (1440000 + 400) \* 3 = 4,320,400

Therefore, the total number of parameters in the CNN is the sum of the number of parameters in each layer, which is:

8400 + 540600 + 4320400 = 4,863,400 parameters

To calculate the amount of RAM required for one instance prediction, we need to calculate the amount of memory required to store the input and output feature maps and the parameters. Assuming we are using 32-bit floats, the amount of memory required to store one instance prediction is:

Input image: 200 x 300 x 3 x 4 bytes = 720,000 bytes

2Output feature maps of bottom layer: 100 x 100 x 150 x 4 bytes = 60,000,000 bytes

Output feature maps of middle layer: 200 x 50 x 75 x 4 bytes = 30,000,000 bytes

Output feature maps of top layer: 400 x 25 x 38 x 4 bytes = 19,200,000 bytes

Parameters: 4,863,400 x 4 bytes = 19,453,600 bytes

Total memory required for one instance prediction: 99,433,600 bytes or approximately 94.8 MB

If we were to process a batch of 50 images, the amount of memory required would be:

Input images: 50 x 200 x 300 x 3 x 4 bytes = 180,000,000 bytes

Output feature maps of bottom layer: 50 x 100 x 100 x 150 x 4 bytes = 1,500,000,000 bytes

Output feature maps of middle layer: 50 x 200 x 50 x 75 x 4 bytes = 1,500,000,000 bytes

Output feature maps of top layer: 50 x 400 x 25 x 38 x 4 bytes = 3,840,000,000 bytes

Parameters: 4,863,400 x 4 bytes = 19,453,600 bytes

Total memory required for a batch of 50 image predictions: 6,940,453,600 bytes or approximately 6.94 GB.

3. What are five things you might do to fix the problem if your GPU runs out of memory while training a CNN?

Ans: If your GPU runs out of memory while training a Convolutional Neural Network (CNN), here are five things you can do to fix the problem:

Reduce batch size: You can reduce the batch size to fit the available memory of the GPU. This approach divides the dataset into smaller batches that can fit in the GPU memory.

Reduce image size: You can resize the images to a smaller size before training the CNN. This reduces the memory required to store the images and can help fit the model in the GPU memory.

Use a smaller model: If your current CNN model is too large, you can use a smaller model architecture that requires less memory to train.

Use mixed precision training: You can use mixed precision training, which trains the model using lower precision data types (e.g., half-precision floating-point numbers) to reduce memory usage.

Use transfer learning: You can use transfer learning by starting with a pre-trained model and then fine-tuning it on your dataset. This approach can save a lot of training time and memory usage since the pre-trained model is already optimized for the specific task.

4. Why would you use a max pooling layer instead with a convolutional layer of the same stride?

Ans: Max pooling is a type of pooling operation commonly used in convolutional neural networks (CNNs) for image recognition tasks. The purpose of max pooling is to reduce the spatial size of the feature map and to make the model more robust to variations in the input image.

Max pooling works by partitioning the input feature map into a set of non-overlapping rectangular regions, and then taking the maximum value of each region. This reduces the dimensionality of the feature map, while preserving the most salient features.

One reason to use max pooling instead of a convolutional layer with the same stride is that max pooling can help reduce overfitting. By discarding some of the less important features in the feature map, max pooling forces the network to learn more robust and generalizable features. This can help prevent the network from memorizing the training data and performing poorly on new, unseen data.

Another reason to use max pooling is that it can help reduce the computational cost of the network. Max pooling reduces the number of parameters in the network and can help speed up the training process.

In summary, max pooling is a useful tool for reducing the spatial size of feature maps in CNNs. It can help reduce overfitting, improve generalization, and reduce computational cost.

5. When would a local response normalization layer be useful?

Ans: A local response normalization (LRN) layer is a type of normalization layer used in neural networks, particularly in convolutional neural networks (CNNs). LRN layers apply a normalization function to the output of each neuron in a feature map, based on the responses of neighboring neurons within the same feature map.

The main purpose of LRN layers is to enhance the ability of the network to discriminate between different features by reducing the response of neurons that are sensitive to high levels of activation. This can help prevent saturation and improve the discriminative power of the network.

LRN layers are most commonly used in image recognition tasks, where they can help improve the accuracy of the network by reducing the impact of irrelevant features in the input image. Specifically, LRN layers are useful in situations where the input images contain a lot of noise or where there is significant variation in the lighting or color of the images.

LRN layers can also be useful in deep networks with many layers, where the output of each layer can become highly correlated and lead to overfitting. By normalizing the output of each neuron based on the responses of neighboring neurons within the same feature map, LRN layers can help reduce the correlation between the output of different neurons and improve the generalization ability of the network.

Overall, LRN layers can be useful in a wide range of image recognition tasks, particularly in situations where the input images contain a lot of noise or where the network has many layers and is at risk of overfitting.

6. In comparison to LeNet-5, what are the main innovations in AlexNet? What about GoogLeNet and ResNet's core innovations?

Ans: LeNet-5, proposed by Yann LeCun et al. in 1998, was one of the first successful convolutional neural networks for image recognition. It consisted of a series of convolutional and pooling layers, followed by fully connected layers.

AlexNet, proposed by Alex Krizhevsky et al. in 2012, built upon the ideas of LeNet-5 but introduced several key innovations that helped to significantly improve the performance of convolutional neural networks. Some of the main innovations in AlexNet include:

Larger network: AlexNet was a much larger network than LeNet-5, with more layers and more parameters.

Rectified linear units (ReLU): AlexNet used ReLU activation functions, which helped to reduce the problem of vanishing gradients and improve the training speed.

Local response normalization (LRN): AlexNet used LRN layers, which helped to reduce overfitting and improve the generalization ability of the network.

Dropout: AlexNet also used dropout, which helped to further reduce overfitting by randomly dropping out some of the neurons during training.

GoogLeNet, proposed by Szegedy et al. in 2014, was a highly innovative network that introduced several new ideas for improving the performance of convolutional neural networks. Some of the main innovations in GoogLeNet include:

Inception modules: GoogLeNet used inception modules, which allowed the network to learn features at different scales and improve the representational capacity of the network.

1x1 convolutions: GoogLeNet also used 1x1 convolutions, which helped to reduce the number of parameters in the network and improve the computational efficiency.

Global average pooling: GoogLeNet used global average pooling, which helped to reduce overfitting and improve the generalization ability of the network.

ResNet, proposed by He et al. in 2015, introduced a new architecture for very deep convolutional neural networks that allowed them to be trained much more effectively than previous networks. The main innovation in ResNet was the use of residual connections, which allowed the network to learn residual functions instead of directly learning the desired mapping. This helped to address the problem of vanishing gradients and made it possible to train much deeper networks than was previously possible.

Overall, the core innovations in these networks reflect a continuous effort to improve the performance of convolutional neural networks. While LeNet-5 was a breakthrough in its time, subsequent networks such as AlexNet, GoogLeNet, and ResNet have continued to push the boundaries of what is possible in image recognition and other applications of deep learning.

7. On MNIST, build your own CNN and strive to achieve the best possible accuracy.

Ans:

import numpy as np

from tensorflow import keras

*# Load the MNIST dataset*

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

*# Preprocess the data*

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

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

y\_train = keras.utils.to\_categorical(y\_train)

y\_test = keras.utils.to\_categorical(y\_test)

*# Define the CNN architecture*

model = keras.Sequential([

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

keras.layers.MaxPooling2D((2, 2)),

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

keras.layers.MaxPooling2D((2, 2)),

keras.layers.Flatten(),

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

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

])

*# Compile the model*

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

*# Train the model*

model.fit(x\_train.reshape(-1, 28, 28, 1), y\_train, epochs=10, validation\_data=(x\_test.reshape(-1, 28, 28, 1), y\_test))

Epoch 1/10

1875/1875 [==============================] - 18s 7ms/step - loss: 0.1271 - accuracy: 0.9618 - val\_loss: 0.0507 - val\_accuracy: 0.9835

Epoch 2/10

1875/1875 [==============================] - 8s 4ms/step - loss: 0.0441 - accuracy: 0.9865 - val\_loss: 0.0374 - val\_accuracy: 0.9874

Epoch 3/10

1875/1875 [==============================] - 8s 4ms/step - loss: 0.0290 - accuracy: 0.9907 - val\_loss: 0.0435 - val\_accuracy: 0.9860

Epoch 4/10

1875/1875 [==============================] - 8s 4ms/step - loss: 0.0210 - accuracy: 0.9936 - val\_loss: 0.0305 - val\_accuracy: 0.9907

Epoch 5/10

1875/1875 [==============================] - 7s 4ms/step - loss: 0.0159 - accuracy: 0.9946 - val\_loss: 0.0335 - val\_accuracy: 0.9899

Epoch 6/10

1875/1875 [==============================] - 9s 5ms/step - loss: 0.0122 - accuracy: 0.9963 - val\_loss: 0.0327 - val\_accuracy: 0.9904

Epoch 7/10

1875/1875 [==============================] - 9s 5ms/step - loss: 0.0101 - accuracy: 0.9968 - val\_loss: 0.0369 - val\_accuracy: 0.9904

Epoch 8/10

1875/1875 [==============================] - 8s 4ms/step - loss: 0.0076 - accuracy: 0.9976 - val\_loss: 0.0396 - val\_accuracy: 0.9898

Epoch 9/10

1875/1875 [==============================] - 9s 5ms/step - loss: 0.0069 - accuracy: 0.9975 - val\_loss: 0.0338 - val\_accuracy: 0.9909

Epoch 10/10

1875/1875 [==============================] - 8s 4ms/step - loss: 0.0067 - accuracy: 0.9977 - val\_loss: 0.0348 - val\_accuracy: 0.9913

Out[6]:

<keras.callbacks.History at 0x7f1f88165dc0>

In [7]:

*# Evaluate the model*

test\_loss, test\_acc = model.evaluate(x\_test.reshape(-1, 28, 28, 1), y\_test)

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

313/313 [==============================] - 1s 4ms/step - loss: 0.0348 - accuracy: 0.9913

Test accuracy: 0.9912999868392944

8. Using Inception v3 to classify broad images. a. Images of different animals can be downloaded. Load them in Python using the matplotlib.image.mpimg.imread() or scipy.misc.imread() functions, for example. Resize and/or crop them to 299 x 299 pixels, and make sure they only have three channels (RGB) and no transparency.The photos used to train the Inception model were preprocessed to have values ranging from -1.0 to 1.0, so make sure yours do as well.

Ans:

from PIL import Image

import numpy as np

import tensorflow as tf

from tensorflow.keras.applications.inception\_v3 import InceptionV3, preprocess\_input, decode\_predictions

from tensorflow.keras.preprocessing import image

import matplotlib.pyplot as plt

*# Load the Inception v3 model*

model = InceptionV3(weights='imagenet')

*# Load image file*

img\_path = "/content/image.jpg"

img = Image.open(img\_path)

*# Check image channels*

if img.mode != "RGB":

img = img.convert("RGB")

if len(img.getbands()) != 3:

raise ValueError("Image does not have 3 channels")

*# Resize/crop image to 299x299 pixels*

img = img.resize((299, 299))

*# Convert image to array*

img\_array = np.array(img)

*# Preprocess the image to have values ranging from -1.0 to 1.0*

img\_array = preprocess\_input(img\_array)

*# Add an extra dimension to represent the batch size (required by the model)*

img\_array = np.expand\_dims(img\_array, axis=0)

*# Classify the image using the Inception v3 model*

preds = model.predict(img\_array)

*# Decode the predictions and display the top 5 predicted classes*

top\_preds = decode\_predictions(preds, top=5)[0]

for pred in top\_preds:

print(f"{pred[1]}: {pred[2]\*100:.2f}%")

*# Display the image*

plt.imshow(img\_array[0])

plt.show()

9. Large-scale image recognition using transfer learning.

a. Make a training set of at least 100 images for each class. You might, for example, identify your own photos based on their position (beach, mountain, area, etc.) or use an existing dataset, such as the flowers dataset or MIT's places dataset (requires registration, and it is huge).

b. Create a preprocessing phase that resizes and crops the image to 299 x 299 pixels while also adding some randomness for data augmentation.

c. Using the previously trained Inception v3 model, freeze all layers up to the bottleneck layer (the last layer before output layer) and replace output layer with appropriate number of outputs for your new classification task (e.g., the flowers dataset has five mutually exclusive classes so the output layer must have five neurons and use softmax activation function).

d. Separate the data into two sets: a training and a test set. The training set is used to train the model, and the test set is used to evaluate it.

In [ ]:

import os

import shutil

*# Define the classes*

classes = ['beach', 'mountain', 'city', 'forest']

*# Define the source directory where your images are stored*

src\_dir = 'path/to/source/folder'

*# Define the destination directory where your organized images will be stored*

dst\_dir = 'path/to/destination/folder'

*# Loop over each class and create a folder for it in the destination directory*

for class\_name in classes:

class\_dir = os.path.join(dst\_dir, class\_name)

os.makedirs(class\_dir, exist\_ok=True)

*# Loop over each file in the source directory and move it to the appropriate folder*

for filename in os.listdir(src\_dir):

class\_name = determine\_class(filename) *# define a function that returns the class name based on the filename*

src\_path = os.path.join(src\_dir, filename)

dst\_path = os.path.join(dst\_dir, class\_name, filename)

shutil.copyfile(src\_path, dst\_path)

In [ ]:

import tensorflow as tf

import numpy as np

*# Define the size of the input images*

image\_size = 299

*# Define the preprocessing function*

def preprocess\_image(image):

*# Resize the image*

image = tf.image.resize(image, (image\_size, image\_size))

*# Randomly crop the image*

image = tf.image.random\_crop(image, (image\_size, image\_size, 3))

*# Normalize the image*

image = (image / 127.5) - 1.0

return image

*# Define the data augmentation function*

def augment\_image(image):

*# 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=0.5)

return image

In [ ]:

from tensorflow.keras.applications import InceptionV3

from tensorflow.keras.layers import Dense, Dropout

from tensorflow.keras.models import Model

*# Define the number of classes*

num\_classes = len(classes)

*# Load the Inception v3 model, excluding the top layer*

base\_model = InceptionV3(include\_top=False, weights='imagenet')

*# Freeze all layers in the base model*

for layer in base\_model.layers:

layer.trainable = False

*# Add a new top layer for classification*

x = base\_model.output

x = Dropout(0.5)(x)

x = Dense(1024, activation='relu')(x)

x = Dense(512, activation='relu')(x)

predictions = Dense(num\_classes, activation='softmax')(x)

*# Define the new model*

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

In [ ]:

import os

import random

import shutil

*# Define the source directory where your organized images are stored*

src\_dir = 'path/to/organized/images'

*# Define the destination directories where your training and test sets will be stored*

train\_dir = 'path/to/training/set'

test\_dir = 'path/to/test/set'

*# Define the split ratio (e.g. 0.8 means 80% of images for training and 20% for testing)*

split\_ratio = 0.8

*# Loop over each class and split the images into training and test sets*

for class\_name in os.listdir(src\_dir):

class\_dir = os.path.join(src\_dir, class\_name)

train\_class\_dir = os.path.join(train\_dir, class\_name)

test\_class\_dir = os.path.join(test\_dir, class\_name)

os.makedirs(train\_class\_dir, exist\_ok=True)

os.makedirs(test\_class\_dir, exist\_ok=True)

filenames = os.listdir(class\_dir)

random.shuffle(filenames)

split\_index = int(len(filenames) \* split\_ratio)

train\_filenames = filenames[:split\_index]

test\_filenames = filenames[split\_index:]

for filename in train\_filenames:

src\_path = os.path.join(class\_dir, filename)

dst\_path = os.path.join(train\_class\_dir, filename)

shutil.copyfile(src\_path, dst\_path)

for filename in test\_filenames:

src\_path = os.path.join(class\_dir, filename)

dst\_path = os.path.join(test\_class\_dir, filename)

shutil.copyfile(src\_path, dst\_path)