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

A1. CNNs (Convolutional Neural Networks) have several advantages over fully connected DNNs (Deep Neural Networks) for image classification tasks. Some of these advantages are:

1. Parameter Efficiency: CNNs are designed to exploit the 2D structure of images, allowing them to learn features more efficiently with fewer parameters than fully connected DNNs. This is because they use shared weights across different regions of the input, which reduces the number of learnable parameters.
2. Translation Invariance: CNNs are designed to be translation invariant, which means that they can recognize objects even if they appear in different locations in the image. This is because they use pooling layers, which reduce the spatial resolution of the feature maps, making them more robust to translation.
3. Local Connectivity: CNNs have local connectivity, meaning that each neuron in a given layer is connected only to a small subset of neurons in the previous layer. This allows them to learn local patterns in the input more effectively.
4. Hierarchical Learning: CNNs are designed to learn hierarchical representations of the input, where lower layers learn simple features like edges and corners, and higher layers learn more complex features like object parts and textures. This hierarchical learning allows them to capture the hierarchical structure of images more effectively.

Overall, these advantages make CNNs better suited for image classification tasks than fully connected DNNs, especially when dealing with large datasets and complex images.

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?

A2. First, let's calculate the output size after each convolutional layer:

* The input image size is 200 x 300 x 3 (RGB channels).
* After the first convolutional layer, with SAME padding and stride of 2, the output size is ceil(200/2) x ceil(300/2) x 100 = 100 x 150 x 100.
* After the second convolutional layer, with SAME padding and stride of 2, the output size is ceil(100/2) x ceil(150/2) x 200 = 50 x 75 x 200.
* After the third convolutional layer, with SAME padding and stride of 2, the output size is ceil(50/2) x ceil(75/2) x 400 = 25 x 38 x 400.

Next, let's calculate the number of parameters:

* The first convolutional layer has 3 kernels of size 3 x 3 x 3, plus a bias term for each kernel, for a total of (3 \* 3 \* 3 + 1) \* 100 = 2,800 parameters.
* The second convolutional layer has 3 kernels of size 3 x 3 x 100, plus a bias term for each kernel, for a total of (3 \* 3 \* 100 + 1) \* 200 = 180,200 parameters.
* The third convolutional layer has 3 kernels of size 3 x 3 x 200, plus a bias term for each kernel, for a total of (3 \* 3 \* 200 + 1) \* 400 = 720,400 parameters.
* The fully connected layer at the end has 25 \* 38 \* 400 nodes from the last convolutional layer, connected to some number of output nodes (depending on the specific classification task), for a total of (25 \* 38 \* 400 + 1) \* N parameters.

To calculate the amount of RAM needed for a single instance prediction, we need to calculate the total number of activations in the network. Each activation is represented as a 32-bit float, so we can simply multiply the number of activations by 4 to get the total memory usage in bytes:

* The input image has size 200 x 300 x 3 = 180,000.
* The first convolutional layer generates 100 feature maps of size 100 x 150 = 15,000 each, for a total of 1.5 million activations.
* The second convolutional layer generates 200 feature maps of size 50 x 75 = 3,750 each, for a total of 7.5 million activations.
* The third convolutional layer generates 400 feature maps of size 25 x 38 = 950 each, for a total of 38 million activations.
* The fully connected layer has N activations, where N is the number of output nodes.

Therefore, the total memory usage for a single instance prediction is approximately:

180,000 \* 4 + 1.5 million \* 4 + 7.5 million \* 4 + 38 million \* 4 + N \* 4 bytes.

If we were to process a batch of 50 images, we would need to multiply the number of activations by 50 to get the total memory usage. The number of parameters would remain the same.

Top of Form

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

A3. Here are five things you might do to fix the problem if your GPU runs out of memory while training a CNN:

1. Reduce the batch size: The batch size is the number of images that the network processes at once. A smaller batch size requires less memory, but it may take longer to train the network.
2. Use a smaller network: A smaller network has fewer weights and requires less memory to store them.
3. Reduce the image size: The size of the input image affects the number of weights in the network. Smaller images require fewer weights and use less memory.
4. Use mixed precision training: By using mixed precision training, you can use half-precision (float16) for the activations and gradients, which requires less memory than single-precision (float32).
5. Use gradient checkpointing: Gradient checkpointing is a technique that trades computation for memory. Instead of storing all activations during forward pass to compute the gradients, it recomputes some activations during backward pass to save memory. This can help reduce memory usage, but may increase training time.

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

A4.   
Max pooling layer is often used instead of a convolutional layer with the same stride for the following reasons:

1. Reduces the Dimensionality: Max pooling reduces the dimensionality of the feature maps by only keeping the maximum values from each region of the feature map. This helps to reduce the number of parameters and computation needed in the network.
2. Increases Invariance: Max pooling helps to increase the invariance of the network to small shifts and distortions in the input. This is because the maximum activation is taken over a local region of the input, so small variations in the input are less likely to affect the output.
3. Helps to Prevent Overfitting: Max pooling helps to prevent overfitting by adding a form of regularization to the network. This is because it reduces the capacity of the network to memorize specific details in the input and instead encourages it to learn more general features.
4. Helps to Reduce the Impact of Noise: Max pooling can help to reduce the impact of noise in the input by only taking the maximum value in each local region. This means that noisy activations are less likely to have a significant impact on the output.

Overall, max pooling is a useful tool for reducing the dimensionality of the input, increasing invariance, preventing overfitting, and reducing the impact of noise in the input.

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

A5. A local response normalization (LRN) layer is useful when dealing with CNNs that have many stacked convolutional layers. The main purpose of an LRN layer is to normalize the output from the previous layer across neighboring feature maps. This normalization helps in suppressing the response of neurons that are less sensitive to their neighbors, which can help to reduce overfitting. Additionally, an LRN layer can also help to increase the generalization performance of a CNN, which can be especially useful when working with large and complex datasets. Therefore, an LRN layer is typically used in architectures that require a lot of convolutional layers, such as the AlexNet and InceptionNet architectures.

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

A6. The main innovations in AlexNet compared to LeNet-5 are as follows:

1. AlexNet had five convolutional layers and three fully connected layers, whereas LeNet-5 had only two convolutional layers and three fully connected layers.
2. AlexNet used the Rectified Linear Unit (ReLU) activation function, which is faster and less prone to the vanishing gradient problem than the sigmoid function used in LeNet-5.
3. AlexNet used local response normalization to reduce overfitting, whereas LeNet-5 did not.
4. AlexNet used dropout to further reduce overfitting, whereas LeNet-5 did not.

The main innovations in GoogLeNet compared to earlier architectures like AlexNet and VGGNet are as follows:

1. GoogLeNet introduced the Inception module, which uses multiple filter sizes and pooling operations in parallel to capture features at different scales.
2. GoogLeNet used global average pooling instead of fully connected layers, which reduces overfitting and makes the model more computationally efficient.
3. GoogLeNet used auxiliary classifiers at intermediate layers during training, which helps with vanishing gradients and improves performance.

The main innovation in ResNet compared to earlier architectures is the use of skip connections or residual connections. These connections allow gradients to flow more easily through the network, which helps with vanishing gradients and allows for much deeper networks to be trained. As a result, ResNet can achieve higher accuracy than earlier architectures with much deeper networks (up to 152 layers).

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

A7.

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Load the data

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

# Preprocess the data

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

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

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes=10)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes=10)

# Define the model

model = Sequential([

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

MaxPooling2D((2, 2)),

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

MaxPooling2D((2, 2)),

Flatten(),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train,

epochs=10,

batch\_size=128,

validation\_data=(x\_test, y\_test))

# Evaluate the model

model.evaluate(x\_test, y\_test)

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.

A8. Steps to follow to use Inception v3 to classify broad images once you have loaded and preprocessed your images.

1. Load the Inception v3 model using the TensorFlow library.
2. Preprocess your images to have values ranging from -1.0 to 1.0.
3. Pass the preprocessed images through the Inception v3 model to obtain the logits (the output of the last fully connected layer before the softmax activation).
4. Apply the softmax activation function to the logits to obtain class probabilities.
5. Use the predicted probabilities to classify your images into different categories.

Here is some sample code you can use to load the Inception v3 model and preprocess your images:

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

from scipy.misc import imread, imresize

# Load Inception v3 model

inception\_v3 = tf.keras.applications.InceptionV3(weights='imagenet')

# Preprocess image

def preprocess\_image(image\_path):

img = imread(image\_path)

img = imresize(img, (299, 299)) # resize image to 299x299

img = img.astype('float32') # convert image to float32

img /= 255.0 # scale pixel values to [0, 1]

img -= 0.5 # shift pixel values to [-0.5, 0.5]

img \*= 2.0 # scale pixel values to [-1, 1]

img = np.expand\_dims(img, axis=0) # add batch dimension

return img

# Load and preprocess image

image\_path = 'path/to/your/image.jpg'

image = preprocess\_image(image\_path)

# Pass image through Inception v3 model

logits = inception\_v3.predict(image)

# Apply softmax activation function to logits

probs = tf.nn.softmax(logits)

# Print predicted class

predicted\_class = np.argmax(probs)

print('Predicted class: ', predicted\_class)

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.