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

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

1. **Parameter Efficiency**: CNNs are more parameter-efficient than fully connected DNNs, as they share weights across different regions of the image. This is especially beneficial when dealing with high-dimensional input data like images.
2. **Translation Invariance**: CNNs are capable of detecting local patterns within an image regardless of their position. This is achieved through the use of convolutional layers, which slide filters across the image, detecting patterns in different regions.
3. **Feature Learning**: CNNs are capable of learning hierarchical representations of the input data. Lower layers learn basic features such as edges and corners, while higher layers learn more complex features like shapes and objects. This makes CNNs well-suited for feature extraction from images.
4. **Robustness**: CNNs are capable of handling input images of varying sizes and orientations. This is achieved through the use of pooling layers, which reduce the spatial dimensions of the input, and convolutional layers, which are invariant to translations.
5. **Reduced Overfitting**: CNNs have the ability to reduce overfitting by applying regularization techniques such as dropout, weight decay, and batch normalization.

Overall, CNNs are better suited for image classification tasks due to their ability to capture local patterns and learn hierarchical representations of the input data, while also being parameter-efficient and robust to varying input sizes and orientations.

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?

A2. The number of parameters in the CNN can be calculated as follows:

For the first convolutional layer:

* Number of filters: 100
* Number of weights per filter: 3 \* 3 \* 3 (3 channels in input image)
* Number of biases per filter: 1 Total number of weights and biases in first layer: (3 \* 3 \* 3 + 1) \* 100 = 2,800

For the second convolutional layer:

* Number of filters: 200
* Number of weights per filter: 3 \* 3 \* 100 (100 feature maps in previous layer)
* Number of biases per filter: 1 Total number of weights and biases in second layer: (3 \* 3 \* 100 + 1) \* 200 = 180,200

For the third convolutional layer:

* Number of filters: 400
* Number of weights per filter: 3 \* 3 \* 200 (200 feature maps in previous layer)
* Number of biases per filter: 1 Total number of weights and biases in third layer: (3 \* 3 \* 200 + 1) \* 400 = 720,400

The total number of parameters in the CNN is the sum of the parameters in each layer: 2,800 + 180,200 + 720,400 = 903,400.

To calculate the memory required for prediction on a single instance, we need to multiply the number of parameters by the size of a 32-bit float. This gives us:

903,400 \* 4 bytes/parameter = 3,613,600 bytes = 3.61 MB

When training on a mini-batch of 50 images, we need to store the gradients for each parameter, so the memory required increases. Let's assume we use backpropagation with no truncation, which means we need to store the activations for each instance and layer as well as the gradients for each weight and bias. Assuming each activation and gradient is a 32-bit float, the memory required for one mini-batch is:

* Input images: 50 \* 200 \* 300 \* 3 \* 4 bytes = 360 MB
* Activations in first layer: 50 \* 100 \* 100 \* 150 \* 4 bytes = 750 MB
* Activations in second layer: 50 \* 200 \* 50 \* 75 \* 4 bytes = 375 MB
* Activations in third layer: 50 \* 400 \* 25 \* 38 \* 4 bytes = 190 MB
* Gradients in third layer: 50 \* 400 \* 25 \* 38 \* 4 bytes = 190 MB
* Gradients in second layer: 50 \* 200 \* 50 \* 75 \* 100 \* 4 bytes = 1.5 GB
* Gradients in first layer: 50 \* 100 \* 100 \* 150 \* 200 \* 4 bytes = 3 GB
* Total: 6.35 GB

Therefore, training on a mini-batch of 50 images would require at least 6.35 GB of RAM.

Top of Form

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

A3. If the GPU runs out of memory while training a CNN, here are five things that can be tried to solve the problem:

1. Reduce batch size: Using a smaller batch size reduces the amount of memory required to process the batch. This will allow the model to fit into the memory of the GPU.
2. Reduce image size: Resizing the input images to a smaller size will reduce the number of pixels in each image and therefore, the amount of memory required to store them.
3. Reduce model complexity: A simpler model with fewer layers and parameters will require less memory.
4. Gradient checkpointing: It is possible to use gradient checkpointing to reduce the amount of memory required to store the gradients during backpropagation. This technique trades computation time for memory by recomputing intermediate activations during backpropagation.
5. Use mixed precision training: By using mixed precision training, it is possible to use lower precision (e.g., float16) to store the model parameters and activations. This reduces the memory requirements for the model and can speed up training.
6. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?

A4. Adding a max pooling layer rather than a convolutional layer with the same stride can be useful for a few reasons:

1. Reducing the spatial dimensions: A max pooling layer reduces the spatial dimensions of the output, making it easier and more efficient to process by subsequent layers.
2. Introducing translational invariance: A max pooling layer introduces some degree of translational invariance, meaning that small variations in the input image's position will not affect the output of the layer as much. This can be useful for tasks like object detection where the location of an object in the image may vary.
3. Reducing the number of parameters: A max pooling layer reduces the number of parameters in the network, which can help reduce overfitting and make the network more efficient.

Overall, adding a max pooling layer can help improve the performance and efficiency of a CNN.

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

A5. Local response normalization layers are not as commonly used in modern deep learning models as they used to be. They were introduced in the AlexNet architecture, which won the ImageNet Large Scale Visual Recognition Challenge in 2012. The purpose of local response normalization is to help reduce overfitting by normalizing the output of a given neuron based on the outputs of adjacent neurons in the same feature map. This can help prevent one feature map from dominating the others. However, in practice, other techniques such as batch normalization are often more effective at preventing overfitting. Therefore, local response normalization layers are not commonly used in modern CNN architectures.

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

A6. Here are the main innovations in each of these models:

AlexNet:

* Used a much larger network architecture compared to LeNet-5.
* Used ReLU activation functions instead of sigmoid.
* Used dropout regularization to reduce overfitting.
* Used data augmentation to increase the size of the training set.

GoogLeNet:

* Introduced the Inception module, which consists of multiple parallel convolutional layers with different filter sizes and pooling layers.
* Used global average pooling instead of fully connected layers at the end of the network.
* Used auxiliary classifiers at intermediate layers to encourage the network to learn more discriminative features.

ResNet:

* Used residual connections to allow deeper networks to be trained without vanishing gradients.
* Used batch normalization to improve training stability and speed up convergence.
* Used a bottleneck architecture to reduce computational complexity.

SENet:

* Introduced the Squeeze-and-Excitation block, which learns how to weight each feature map channel according to its importance.
* Used these blocks to allow the network to learn how to focus on the most important features and ignore less important ones.

Xception:

* Used a depthwise separable convolution operation, which separates the spatial and channel-wise convolutions.
* Used this operation in a nested fashion to create a very deep and efficient network architecture.
* Achieved state-of-the-art performance on several benchmark datasets.

Top of Form

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

A7. A fully convolutional network (FCN) is a neural network architecture that only consists of convolutional layers and pooling layers, and does not include any fully connected layers. It is commonly used for semantic segmentation tasks, where the goal is to classify each pixel in an image.

To convert a dense layer into a convolutional layer, we need to reshape the weights of the dense layer to match the shape of a convolutional filter. Specifically, if the dense layer has **N** neurons and takes an input tensor of shape **(batch\_size, input\_dim)**, we can convert it into a convolutional layer with **N** filters of size **1 x input\_dim**. Then, the output of this layer will have shape **(batch\_size, N, 1, 1)**, which can be fed into subsequent convolutional layers.

This conversion is useful in cases where we want to use a pretrained fully connected layer in a convolutional neural network, without introducing too many additional trainable parameters. By converting the dense layer into a convolutional layer, we can preserve the learned weights and use them as convolutional filters.

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

A8. The main technical difficulty of semantic segmentation is to correctly assign a class label to each pixel of an image, taking into account the spatial context and the shape of the objects. Unlike image classification, where the goal is to assign a single class label to an entire image, semantic segmentation requires assigning a label to each pixel of the image. This requires capturing both local and global information from the image, as well as modeling the complex dependencies between neighboring pixels. Another challenge is dealing with class imbalance, as some classes may be much more common than others in the dataset. Finally, semantic segmentation requires a lot of annotated data, which can be expensive and time-consuming to obtain.

1. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.
2. 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.

A10.

General steps for using transfer learning for large image classification:

a. Collect a large dataset containing at least 100 images per class. You can either use an existing dataset or create your own dataset by classifying your pictures based on location, objects, or any other category of your interest.

b. Split the dataset into training, validation, and test sets. The training set should be the largest, followed by the validation set, and the test set should be the smallest.

c. Build an input pipeline using TensorFlow's Data API to load and preprocess the data. Apply data augmentation techniques like random cropping, flipping, and rotation to augment the dataset to increase the variety of images. Normalize the pixel values and resize the images to the appropriate dimensions.

d. Fine-tune a pre-trained model on the dataset. Choose a pre-trained model that is similar to the problem you are trying to solve. For example, if you are classifying animals, you can use the Inception v3 model, which was trained on the ImageNet dataset and has a similar set of classes. Remove the last layer(s) of the pre-trained model and replace it with a new classification layer(s) that matches the number of classes in your dataset. Freeze the weights of the pre-trained layers and only train the weights of the new classification layer(s) on your dataset. Use the validation set to tune the hyperparameters of the model and avoid overfitting.

e. Evaluate the model on the test set and report the classification accuracy and any other relevant performance metrics. If the model does not perform well, consider re-tuning the hyperparameters or trying a different pre-trained model.