ASSIGNMENT - 9

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

Ans: Advantages of CNNs for Image Classification over Fully Connected DNNs:

Automatic Feature Extraction: CNNs excel at automatically learning relevant features from images through convolutional layers. These layers capture spatial information and identify edges, lines, and other low-level patterns that build towards higher-level features like shapes and objects. DNNs, on the other hand, typically require manual feature engineering, which can be time-consuming and domain-specific.

Parameter Efficiency: CNNs leverage weight sharing, where a single filter is applied across different parts of the image. This reduces the number of parameters compared to DNNs, which have full connections between every neuron in adjacent layers. In image tasks, where spatial relationships are crucial, this parameter efficiency translates to better generalization and less overfitting.

Shift Invariance: CNNs exhibit a degree of shift invariance due to their local connections and pooling operations. This means the network is less sensitive to slight shifts in object position within the image, making it more robust to variations in image composition. DNNs, with their full connectivity, are more susceptible to such shifts.

Hierarchical Learning: CNNs employ a hierarchical architecture where lower layers extract simpler features that are then combined to form more complex features in higher layers. This structure efficiently captures the underlying structure of 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&#39;re using 32-bit floats? What if you were to practice on a batch of 50 images?

Ans: Parameters:

Convolutional Layer 1:

* Input Feature Maps: Unknown (denoted as F\_in)
* Output Feature Maps (100) x Kernels (3) = 300
* Parameters per Filter: F\_in \* Kernel Size^2 (assuming square kernels)
* Total Parameters (ignoring bias): 300 \* (F\_in \* Kernel Size^2)

Convolutional Layer 2:

* Input Feature Maps (100)
* Output Feature Maps (200) x Kernels (3) = 600
* Parameters: 600 \* (100 \* Kernel Size^2)

Convolutional Layer 3:

* Input Feature Maps (200)
* Output Feature Maps (400) x Kernels (3) = 1200
* Parameters: 1200 \* (200 \* Kernel Size^2)

Total Parameters: 300 \* (F\_in \* Kernel Size^2) + 600 \* (100 \* Kernel Size^2) + 1200 \* (200 \* Kernel Size^2) + (Number of Biases)

Memory Usage (Single Image):

Input Image: 3 channels (RGB) \* 200 pixels (width) \* 300 pixels (height) \* 4 bytes/float = 0.72 MB (assuming 32-bit floats)

Feature Maps:

* Layer 1: 100 maps \* Unknown size (depends on F\_in and kernel size) \* 4 bytes/float
* Layer 2: 200 maps \* Unknown size \* 4 bytes/float
* Layer 3: 400 maps \* Unknown size \* 4 bytes/float

Total Memory: 0.72 MB (input) + sum of feature map memory across layers

Memory Usage (Batch of 50 Images):

Multiply the single-image memory usage by 50:

* Total Memory = 50 \* (0.72 MB + sum of feature map memory across layers)

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

Ans: Fixing GPU Memory Out-of-Memory (OOM) Errors:

Reduce Batch Size: This is the most common and effective approach. By feeding smaller batches of data to the GPU at a time, you can significantly decrease memory usage. Experiment with different batch sizes to find the optimal balance between training speed and memory consumption.

Optimize Data Loading: Utilize data generators or input pipelines to load data in batches on-the-fly instead of loading the entire dataset at once. This prevents overwhelming the GPU memory with unnecessary data. Libraries like TensorFlow's tf.data module provide efficient data loading capabilities.

Reduce Model Complexity: If your CNN has a large number of parameters or layers, consider simplifying it by:

Using smaller filter sizes in convolutional layers.

Reducing the number of filters in convolutional layers.

Using fewer layers overall. Analyze the impact of these changes on model performance and find a good compromise.

Mixed Precision Training: Leverage mixed precision techniques (e.g., TensorFlow's tf.keras.mixed\_precision) to train models with lower precision formats (like float16) while maintaining comparable accuracy. This can significantly reduce memory requirements.

Gradient Accumulation: Accumulate gradients over multiple batches before applying updates to the model weights. This allows training with larger effective batch sizes without exceeding GPU memory limitations. Implement this using frameworks like PyTorch or TensorFlow.

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

Ans: Max Pooling vs. Strided Convolution:

Max Pooling: Reduces the spatial dimensionality of an activation map while selecting the maximum value from a window. It helps with:

* Downsampling data to reduce computational cost.
* Introducing invariance to small shifts in the input.

Strided Convolution: Applies a convolution filter with a stride greater than 1, effectively downsampling the output. It offers:

* More flexibility in feature extraction compared to max pooling, as you can learn filters that perform a specific downsampling operation.
* Can potentially be more computationally efficient than max pooling.

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

Ans: LRN is useful in situations where you want to:

* Reduce the influence of strong activations: It normalizes activations across local neighborhoods, suppressing overly strong responses and allowing weaker features to become more prominent. This can be beneficial for tasks like object detection or image segmentation, where you want to detect subtle variations in features.
* Improve generalization: LRN can help prevent overfitting by normalizing activations, making the model less sensitive to specific data distributions.

However, LRN is not as widely used in modern CNN architectures due to:

* Computational overhead: It adds an extra layer of computation compared to simpler activation functions like ReLU.
* Redundancy: Batch normalization layers, which are now standard in many CNNs, can achieve similar effects to LRN without the additional cost.

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).

Ans: Building Your Dataset:

Choose Classes and Collect Images:

* Identify your own image categories based on content (e.g., beach, mountain, forest) or use an existing dataset like:
  + Flowers dataset (small and manageable) (<https://www.robots.ox.ac.uk/~vgg/research/flowers/>)
  + MIT Places dataset (larger, requires registration) (http://places2.csail.mit.edu/explore.html)

Collect Images:

* Ensure at least 100 images per class for initial training.
* Consider using online resources like Flickr or searching for specific categories.
* Maintain a balanced distribution of images across classes.

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

Ans: Resizing and Cropping:

* Resize all images to a uniform size of 299 x 299 pixels to match Inception v3's input requirements.
* Implement random cropping to introduce data augmentation and improve generalization:
  + Define a random window within the image slightly larger than 299 x 299 pixels.
  + Randomly crop a 299 x 299 pixel section from the window for each training image.

Normalization:

* Convert pixel values from the original range (usually 0-255) to a range of -1.0 to 1.0. This is because Inception v3 was trained on data preprocessed this way.

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).

Ans: Load the Inception v3 Model:

* Use a deep learning library like TensorFlow or Keras to load the pre-trained Inception v3 model.

Freeze Layers:

* Freeze all layers of Inception v3 up to the bottleneck layer (the layer before the final output layer). This preserves the learned features from the original training.

Modify Output Layer:

* Remove the original output layer of Inception v3.
* Add a new output layer with the appropriate number of neurons for your classification task.
  + Example: For a flower dataset with 5 classes, add a layer with 5 neurons using softmax activation for multi-class classification.

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.

Ans: Split Data:

* Divide your collected images into two sets: training set (around 80%) and test set (around 20%).
* The training set is used to train the model, while the test set is used to evaluate its performance after training.

Train the Model:

* Train the model using the training set, focusing on updating the weights of the newly added output layer and potentially a few layers before the bottleneck layer (unfreezing a small number of layers can sometimes improve performance).

Evaluation:

* Evaluate the model's performance on the unseen test set using metrics like accuracy, precision, recall, and F1 score.