Alireza Mottaghi 400222089

ANN 3rd assignment

1.

1. What is gradient accumulation?

   From what I understood Gradient accumulation is the process of summing up gradients over multiple mini-batches before performing a single weight update. Instead of updating the model parameters after each mini-batch, you accumulate the gradients over several mini-batches and then perform a single update.

2. When should we use this technique?

   Gradient accumulation is useful in several scenarios:

   a) Limited GPU memory: When your model or batch size is too large to fit in GPU memory, you can use smaller mini-batches and accumulate gradients to simulate a larger batch size.

   b) To increase effective batch size: Sometimes, larger batch sizes can lead to better training stability or convergence. Gradient accumulation allows you to simulate larger batch sizes without actually processing more data at once.

   c) Distributed training: In multi-GPU or multi-node setups, gradient accumulation can help reduce communication overhead by performing fewer weight updates.

   d) When processing variable-length sequences: If you're dealing with sequences of different lengths, gradient accumulation can help balance out the computational load.

3. How to perform gradient accumulation in PyTorch:

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define your model, loss function, and optimizer
model = YourModel()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# Set the number of accumulation steps
accumulation_steps = 4
```

```python
# Training loop
for epoch in range(num_epochs):
    for i, (inputs, labels) in enumerate(dataloader):
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Normalize the loss to account for accumulation steps
        loss = loss / accumulation_steps

        # Backward pass
        loss.backward()

        # Perform optimization step every 'accumulation_steps'
        if (i + 1) % accumulation_steps == 0:
            optimizer.step()
            optimizer.zero_grad()

    # Don't forget to perform any remaining updates at the end of the epoch
    if (i + 1) % accumulation_steps != 0:
        optimizer.step()
        optimizer.zero_grad()
```

1. We define the number of accumulation steps (accumulation_steps).
2. We normalize the loss by dividing it by accumulation_steps to ensure the gradients are scaled correctly.
3. We perform the backward pass after each mini-batch, accumulating gradients.
4. We only perform the optimization step (update weights) every accumulation_steps iterations.
5. We make sure to perform any remaining updates at the end of each epoch.

2.

Backpropagation in convolutional layers:

1. Forward Pass: First, recall that in the forward pass, the convolutional layer performs the following operation:

- Input: X (typically 3D: height × width × channels)

- Kernel/Filter: W (typically 4D: kernel_height × kernel_width × input_channels × output_channels)
- Output: Y = X * W + b (where * denotes convolution and b is the bias)

2. Backpropagation Overview: The goal of backpropagation is to compute the gradients of the loss with respect to the weights (∂L/∂W) and biases (∂L/∂b) of the convolutional layer, as well as the gradient with respect to the input (∂L/∂X) for propagating the error to earlier layers.
3. Gradient Computation: a) Gradient w.r.t. the output (∂L/∂Y): This gradient is received from the next layer (or from the loss function if this is the last layer). b) Gradient w.r.t. the bias (∂L/∂b): This is simply the sum of ∂L/∂Y over all spatial locations for each output channel. c) Gradient w.r.t. the weights (∂L/∂W):
    - For each position of the kernel over the input:
        - Multiply the corresponding patch of the input with the gradient of the output at that position.
    - Sum these products over all positions.

    Mathematically: ∂L/∂W = X' * ∂L/∂Y Where X' represents patches of the input and * denotes cross-correlation. d) Gradient w.r.t. the input (∂L/∂X):
        - Pad the gradient of the output (∂L/∂Y) with zeros.
        - Perform a full convolution with the flipped kernels.

Mathematically: ∂L/∂X = full_conv(∂L/∂Y, flip(W))

4. Detailed Steps: a) Computing ∂L/∂b:
    - Sum ∂L/∂Y across all spatial dimensions for each output channel.

    b) Computing ∂L/∂W:
    - For each output channel:
        - For each input channel:
            - Perform a 2D convolution between the input channel and the corresponding slice of ∂L/∂Y.

- This results in a 4D tensor of the same shape as W.

c) Computing ∂L/∂X:

- Pad ∂L/∂Y with zeros (the padding depends on the stride and padding used in the forward pass).
- For each input channel:
  - Perform a full convolution with the flipped kernels from all output channels.
- Sum the results across all output channels.

5. Considerations:

- Stride: If stride > 1 was used in the forward pass, you need to insert zeros in ∂L/∂Y before the convolution when computing ∂L/∂X.
- Padding: The amount of padding used in the forward pass affects how you compute ∂L/∂X.
- Dilation: If dilated convolutions were used, this needs to be accounted for in the backward pass as well.

6. Efficiency: In practice, these operations are highly optimized and parallelized on GPUs. Libraries like cuDNN provide efficient implementations of these backward operations.

3.

Benefits of Pooling Layers:

1. Dimensionality Reduction: Pooling reduces the spatial dimensions (width and height) of the input volume, which decreases the number of parameters and computational load in the network.
2. Translation Invariance: Pooling helps the network become more robust to small translations in the input, as it summarizes features over a local region.
3. Feature Extraction: By summarizing features in a region, pooling helps in extracting dominant features, which can be useful for classification tasks.

4. Overfitting Prevention: By reducing the model's parameters, pooling can help prevent overfitting.
5. Hierarchical Representation: Pooling allows the network to look at progressively larger areas of the input, facilitating hierarchical feature learning.

Drawbacks of Pooling Layers:

1. Information Loss: Pooling discards spatial information within each pooling region, which might be crucial for some tasks (e.g., precise object localization).
2. No Learning: Traditional pooling operations don't have learnable parameters, potentially limiting the network's adaptability.
3. Reduced Feature Resolution: Repeated pooling can significantly reduce the resolution of feature maps, which might be problematic for tasks requiring fine-grained details.
4. Disproportionate Impact of Errors: In max pooling, errors in the maximum value can have a large impact as they're propagated forward while other values are ignored.

Max Pooling vs. Average Pooling:

Max Pooling:

- Operation: Selects the maximum value from each pooling region.
- Characteristics:
    - Tends to capture the most prominent features.
    - More commonly used in practice.
    - Better at preserving texture and high-frequency details.

Average Pooling:

- Operation: Calculates the average value of each pooling region.
- Characteristics:

- Smooths out the features more.
- Can be useful when we want to preserve more background information.
- May work better for tasks where the average intensity is important.

When to prefer one over the other:

- Max Pooling is often preferred for:
  1. Tasks requiring detection of specific features, regardless of their exact location.
  2. Computer vision tasks like object detection or image classification.
  3. When working with sparse activation maps.
- Average Pooling might be preferred for:
  1. Tasks where the overall intensity or presence of features matters more than their specific values.
  2. Reducing noise in the input.
  3. Tasks where smoothing of features is beneficial, like some types of style transfer.

Frequency of Use:

While pooling layers are common in CNNs, they should not be used too frequently. Here are some considerations:

1. Network Depth: Deeper networks typically use fewer pooling layers, often placing them strategically to reduce dimensions at certain points.
2. Task Requirements: Some tasks (like semantic segmentation) might require less pooling to maintain spatial resolution.
3. Modern Architectures: Some modern architectures (like ResNet) use strided convolutions instead of pooling to reduce dimensions.
4. Impact on Features: Excessive pooling can lead to loss of important spatial information.

5. Alternative Approaches: Some networks use techniques like dilated/atrous convolutions to maintain receptive field size without excessive pooling.

Based on my searches a common practice is to use pooling layers after one or a few convolutional layers, rather than after every convolutional layer. The exact frequency depends on the specific architecture and task.

4.

1. Concept of Transfer Learning in CNNs:

   Transfer learning involves using a pre-trained model as a starting point for a new, related task. In the context of CNNs, this typically means:

   a) Using the architecture of a model trained on a large dataset (e.g., ImageNet). b) Initializing the new model with the weights from the pre-trained model. c) Fine-tuning the model on the new, typically smaller, dataset.

2. How Transfer Learning Leverages Pre-trained Models:

   - Feature Extraction: Lower layers in CNNs often learn generic features (like edges, textures) that are useful across many image-related tasks.
   - Knowledge Transfer: The pre-trained model has already learned to extract meaningful features from images, which can be beneficial even if the new task is different.
   - Reduced Data Requirements: By starting with pre-learned features, the model can achieve good performance with less task-specific training data.
   - Faster Convergence: Fine-tuning often converges faster than training from scratch.

3. Process of Transfer Learning:

a) Choose a pre-trained model.

b) Remove the final layer(s) specific to the original task.

c) Add new layer(s) suitable for the new task.

d) Options for training:

- Freeze early layers and only train new layers (feature extraction).

- Fine-tune all layers with a low learning rate (full fine-tuning).
- Combination: freeze some layers and fine-tune others.

4. Benefits for Limited Data Scenarios:
   - Overcoming Data Scarcity: Pre-trained models have already learned robust features from large datasets, which helps when task-specific data is limited.
   - Regularization Effect: Starting from pre-trained weights can act as a form of regularization, reducing overfitting on small datasets.
   - Improved Generalization: The diverse features learned from large datasets often generalize well to new tasks.
5. Popular Pre-trained CNN Models and Their Common Uses:

a) ResNet (Residual Networks):

- Trained on: ImageNet
- Common uses: Image classification, object detection, segmentation
- Key feature: Deep architecture with skip connections

b) VGG (Visual Geometry Group):

- Trained on: ImageNet
- Common uses: Image classification, feature extraction
- Key feature: Simple, uniform architecture

c) Inception (GoogLeNet):

- Trained on: ImageNet
- Common uses: Image classification, object detection
- Key feature: Inception modules with multiple kernel sizes

d) MobileNet:

- Trained on: ImageNet
- Common uses: Mobile and embedded vision applications

- Key feature: Lightweight, designed for efficiency

From my understandings, Transfer learning with CNNs has become a standard approach in computer vision, allowing researchers and practitioners to leverage the power of deep learning even with limited domain-specific data. It's particularly valuable in specialized fields where large labeled datasets are hard to obtain.

5.

1. "Face verification requires comparing a new picture against one person's face, whereas face recognition requires comparing a new picture against K person's faces."

   True.

   Explanation:

   - Face verification is a 1:1 matching problem. It answers the question, "Is this the person they claim to be?" by comparing a new image to a single known face.
   - Face recognition is a 1:N matching problem. It attempts to identify a person by comparing their face against a database of K known faces, answering the question, "Who is this person?"

2. "In order to train the parameters of a face recognition system, it would be reasonable to use a training set comprising 100,000 pictures of 100,000 different persons."

   False.

   Explanation:

   - While a large dataset is beneficial for training a face recognition system, having only one picture per person is not ideal.

- Face recognition systems need to learn to recognize individuals across various conditions (lighting, angles, expressions, etc.). Multiple images of each person are necessary to capture this variability.
- A more effective dataset would have multiple images (perhaps 10-100) for each of several thousand individuals, rather than just one image per person.
- This approach allows the model to learn features that are consistent for an individual across different conditions, improving its ability to generalize and recognize faces accurately.

3. "You train a CNN on a dataset with 100 different classes. You wonder if you can find a hidden unit that responds strongly to pictures of cats. (I.e., a neuron so that, of all the input/training images that strongly activate that neuron, the majority are cat pictures.) You are more likely to find this unit in layer 4 of the network than in layer 1."

   True.

   Explanation:

   - CNNs learn hierarchical features, with earlier layers capturing low-level features (like edges, textures) and deeper layers capturing more complex, high-level features.
   - Layer 1 typically responds to very basic features like edges or color patches, which are not specific to cats or any particular object.
   - Layer 4, being deeper in the network, is more likely to have neurons that respond to complex, object-specific features. These could include cat-like shapes, fur textures, or even cat faces.
   - As you go deeper in the network, the features become more abstract and more closely aligned with the classes the network is trying to distinguish.
   - Therefore, if a "cat detector" neuron exists, it's more likely to be found in deeper layers like layer 4 rather than in the initial layers like layer 1.

Code Report:

1) Data Explanation

Architectural Heritage Elements Dataset (AHE) is an image dataset for developing deep learning algorithms and specific techniques in the classification of architectural heritage images. This dataset consists of 10235 images classified in 10 categories: Altar, Apse, Bell tower, Column, Dome (inner), Dome (outer), Flying buttress, Gargoyle, Stained glass, Vault. It is inspired by the CIFAR-10 dataset but with the objective in mind of developing tools that facilitate the tasks of classifying images in the field of cultural heritage documentation. Most of the images have been obtained from Flickr and Wikimedia Commons (all of them under creative commons license).

- Altar: 829 images
- Apse: 514 images
- Bell tower: 1059 images
- Column: 1919 images
- Dome (inner): 616 images
- Dome (outer): 1177 images
- Flying buttress: 407 images
- Gargoyle (and Chimera): 1571 images
- Stained glass: 1033 images
- Vault: 1110 images

2) CNN Model Architecture

Convolutional Layers

- First Layer: The model begins with a Conv2d layer that takes an input with 3 channels (presumably RGB images) and outputs 32 feature maps using a 3x3 kernel with padding set to 1.

• Second to Fifth Layers: Subsequent Conv2d layers increase the depth of the network successively: 64, 128, 256, and finally 512 feature maps, using the same kernel size and padding as the first. This increase in depth at each layer allows the network to capture increasingly complex features at different levels of abstraction.

Batch Normalization

• Normalization Layers: Each convolutional layer is followed by a BatchNorm2d layer corresponding to the number of output feature maps from the preceding convolutional layer. These layers normalize the outputs of the convolutional layers, helping in stabilizing the learning process and improving the convergence rate.

Activation Function

• ReLU Activation: Each batch normalization layer is followed by a ReLU (Rectified Linear Unit) activation function, which introduces non-linearity to the model, allowing it to learn more complex patterns.

Pooling Layers

• Max Pooling: A MaxPool2d layer with a 2x2 window and stride of 2 follows each ReLU activation function (except after the last convolutional layer). These layers reduce the spatial dimensions of the feature maps, thus reducing the number of parameters and computation in the network.

Dropout

• Dropout Regularization: Dropout layers with a rate of 0.25 are used before each fully connected layer. Dropout helps prevent overfitting by randomly setting a fraction of input units to 0 at each update during training time.

Fully Connected Layers

• Linear Layers: The model flattens the output from the final pooling layer and passes it through a sequence of fully connected layers. The dimensions decrease from 2048 to 1024, then to 512, and finally to 10 output nodes, corresponding to the number of classes the model is expected to classify.

Final Output

- Classification Layer: The last FC layer outputs 10 values, each representing a class score. These scores are typically passed through a softmax function (not included in the provided script) to obtain class probabilities.

3) Evaluation of CNN Model Performance

Loss

- Train Loss: Shows a sharp decline initially, indicating rapid learning in the early stages. It continues to decrease steadily, suggesting that the model is effectively minimizing the error in its predictions over time.
- Test Loss: Initially decreases alongside the train loss, reflecting that the model is generalizing well to unseen data. However, it shows fluctuations in later epochs, possibly indicating overfitting or instability in learning on the validation data.

Accuracy

- Train Accuracy: Increases consistently, which is a typical expectation as the model becomes better at classifying the training data.
- Test Accuracy: Also increases but plateaus and exhibits more variability compared to the train accuracy. The discrepancy between train and test accuracy may point to overfitting.

Precision

- Train Precision: Shows steady improvement, implying the model's increasing capability in not labeling negative samples as positive.
- Test Precision: Also improves but remains lower and more variable than train precision, which may suggest challenges in model generalization.

Recall

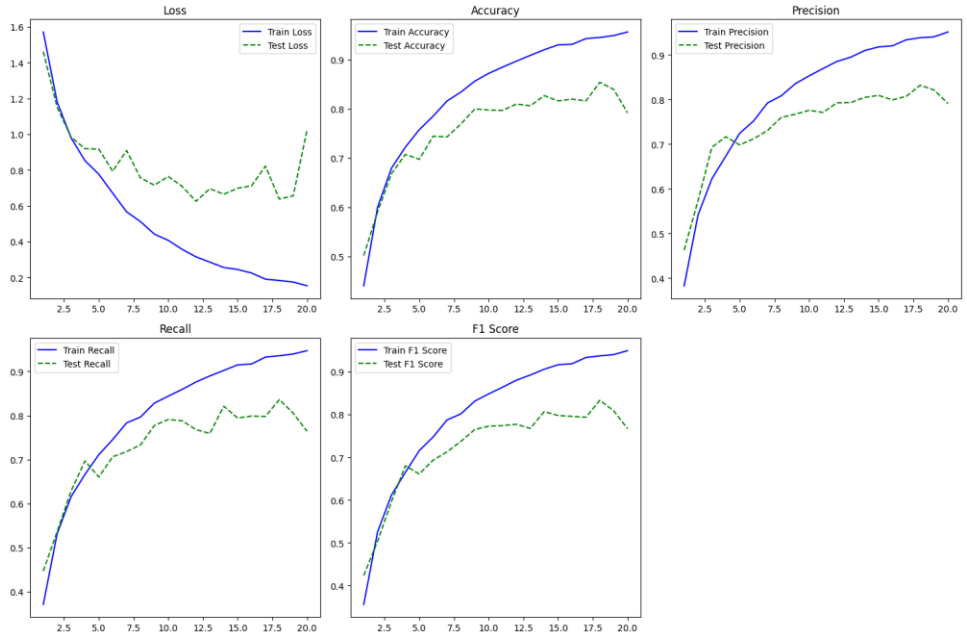
- Train Recall: Consistently rises, indicating that the model is getting better at identifying all relevant cases.

• Test Recall: Increases more gradually and fluctuates, pointing to potential issues in model sensitivity towards detecting positive cases in the test set.

F1 Score

• Train F1 Score: Increases steadily, showcasing an improving balance between precision and recall in training.

• Test F1 Score: Follows a similar upward trend but with noticeable dips, highlighting possible inconsistencies in model performance on the test data.



4) CNN Feature Visualization

The given methodology demonstrates a technique for visualizing convolu7onal neural network (CNN) activations to understand how the network processes an image and what features it extracts at various layers. This is accomplished through a custom class CNNVisualizer, which hooks into the CNN model to capture ac7va7on maps and gradients during both the forward and backward passes.

The CNNVisualizer class is ini7alized with a CNN model, and it sets up hooks on the convolu7onal layers to capture data during network opera7ons. It stores activations after each convolu7onal layer during the forward pass and gradients

during the backward pass. The visualiza7ons are generated based on these captured activation and gradients, allowing for inspec7on of the specific features that the network learns to recognize.

Detailed Steps:

1. Ini5aliza5on and Hook Registra5on:

- Upon ini7aliza7on, the visualizer registers two types of hooks on the convolu7onal layers of the model:
    - Forward Hook: Captures the output of the convolu7onal layers, i.e., the activation maps.
    - Backward Hook: Captures the gradients as they propagate back through the network, which are crucial for visualizing the relevance of each activation to the final output.

2. Visualiza5on Process:

- For visualiza7on, an image is passed through the network to obtain the ac7va7on maps for the selected layer and filter.
- The specific activation map for the selected filter is then used to compute the mean activation, which serves as a scalar value to ini7ate the backward pass.
- During the backward pass, gradients are captured, and a pooled gradient is calculated by averaging over certain dimensions to isolate the contribu7on of specific features.
- The ac7va7on maps are then adjusted by these pooled gradients to highlight features that strongly influence the network's output.

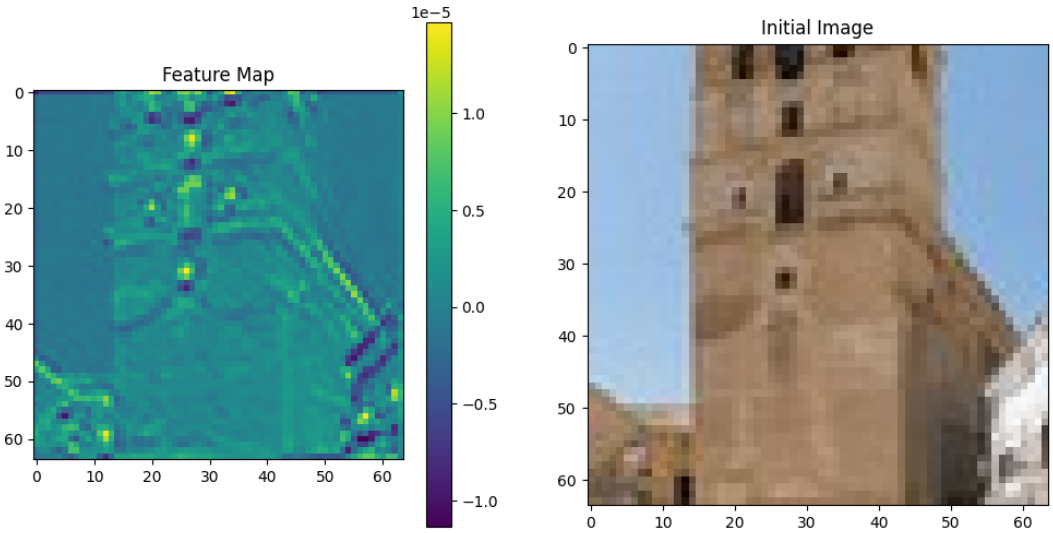
3. Resul5ng Visualiza5on:

- The visualiza7on highlights the areas of the image that are most relevant to the specific filter in the selected convolu7onal layer. This allows us to see which features of the image are being ac7vated by that par7cular filter.

Use Case and Example:

The visualizer was used to generate feature maps from a CNN for an image of a bell tower. The resultant visualiza7on (labeled "Feature Map") effec7vely illustrates the spa7al activation patterns across the image, emphasizing features that are significant to the model's percep7on and decision-making processes.

The side-by-side display of the original image ("Ini7al Image") and its corresponding feature map allows for a compara7ve analysis, showing how the network is interpre7ng the architectural elements of the bell tower. This technique is especially useful for debugging and improving model architectures by providing insights into the model's behavior at a granular level.



5) Image Generation for Specific Classes

1. Model Setup: The CNN model is set to evaluation mode using model.eval() to ensure that all layers behave consistently during both the training and inference phases, specifically for layers like dropout and batch normalization.

2. Image Initialization: Depending on whether a path to an initial image is provided, the script either loads and processes an image from the disk or initializes a random noise image. The dimensions for the initialization

are set to 1x3x64x64, representing a single image with three color channels and 64x64 pixel resolution.

3. Image Transformation: The images undergo a normalization transformation using predefined means and standard deviations of [0, 0, 0] and [5, 5, 5] respectively. This step standardizes the input data to a common scale, enhancing model training and convergence.

4. Optimization Loop: The script employs a gradient ascent technique where the image's pixel values are optimized to maximize the activation of the neuron corresponding to the target class. This is done using the Adam optimizer with a learning rate of 1e-2. The process iterates 1000 times, adjusting the image to focus on features important for the class prediction.

5. Image Post-Processing: After the optimization, the image tensor is detached from the current computational graph, converted back to a NumPy array, rearranged from CHW to HWC format, rescaled to 0-255 pixel values, and clipped to ensure valid image pixel ranges.

A set of images was generated and visualized for different architectural classes such as "altar", "apse", "bell tower", etc. Each subplot represents the features that most activate the model's predictions for that class. Observations from Generated Images The resulting images, as displayed, appear highly abstract and noisy, characterized by patterns and colors rather than distinct architectural features. This abstraction indicates that the model might be focusing more on textures or color distributions rather than clear structural elements.