Neural Network

by

Dr. Bagheri

Final Project

Parisa Toumari

99101857

Question 1

The layers of my CNN:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 256, 256, 256]           7,168
         MaxPool2d-2        [-1, 256, 128, 128]               0
           Dropout-3        [-1, 256, 128, 128]               0
            Conv2d-4        [-1, 128, 128, 128]         295,040
         MaxPool2d-5          [-1, 128, 64, 64]               0
           Dropout-6          [-1, 128, 64, 64]               0
            Conv2d-7           [-1, 64, 62, 62]          73,792
         MaxPool2d-8           [-1, 64, 31, 31]               0
           Flatten-9                [-1, 61504]               0
           Linear-10                  [-1, 256]      15,745,280
           Linear-11                  [-1, 128]          32,896
           Linear-12                    [-1, 6]             774
================================================================
Total params: 16,154,950
Trainable params: 16,154,950
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 218.82
Params size (MB): 61.63
Estimated Total Size (MB): 281.19
----------------------------------------------------------------
Total number of parameters: 16154950
```

- Training loop: Iterates through multiple epochs to train the model.
- model.train(): Sets the model to training mode (activates layers like dropout).
- Forward pass: Computes predictions using the model on input data.
- Loss calculation: Computes the loss between predicted outputs and actual labels.
- Backward pass (Backpropagation): Computes gradients of the loss with respect to model parameters.
- Optimizer step: Updates model parameters based on the gradients computed.
- Progress bar: Provides visual feedback on the training progress using tqdm.

Backpropagation:

- Backpropagation is implicitly handled when loss.backward() is called. This function computes the gradients of the loss with respect to all model parameters.
- During the backward pass, gradients are computed using the chain rule of calculus, propagating from the output layer back to the input layer through each layer of the neural network.

I used 20 epochs and got 99.8% validation for the train datas but got only 42% accuracy on test datas.

```
Epoch 17/20: 1344batch [07:24,  3.02batch/s, loss=0.0163]
Validation 17/20: 100%|██████████| 5334/5334 [01:32<00:00, 57.67batch/s]
Accuracy of the network on the validation images: 99.31 %
Epoch 18/20: 1344batch [07:24,  3.03batch/s, loss=0.0138]
Validation 18/20: 100%|██████████| 5334/5334 [01:32<00:00, 57.84batch/s]
Accuracy of the network on the validation images: 99.64 %
Epoch 19/20: 1344batch [07:25,  3.02batch/s, loss=0.0136]
Validation 19/20: 100%|██████████| 5334/5334 [01:32<00:00, 57.94batch/s]
Accuracy of the network on the validation images: 99.25 %
Epoch 20/20: 1344batch [07:23,  3.03batch/s, loss=0.0148]
Validation 20/20: 100%|██████████| 5334/5334 [01:32<00:00, 57.71batch/s]
Accuracy of the network on the validation images: 99.76 %
```

```python
correct = 0
total = 0
for images,labels in test_dataloader:
  images = images.to(device)
  labels = labels.to(device)
  outputs = model(images)
  prediction = torch.argmax(outputs, dim = 1)
  correct += (prediction == labels).sum().item()
  total += labels.size(0)

print(correct/total * 100)
```

```
41.33738601823708
```

FPS

The code calculates frames per second (FPS) as a measure of how many images (or batches of images) a neural network model can process per second.

Calculation of FPS:

Time Measurement:
For each batch of images processed (num_batches batches in total):
It records the start time (start_time) before processing the batch.
It performs a forward pass through the model to generate outputs.
It records the end time (end_time) after processing the batch.
It calculates the duration it took to process the batch as processing_time = end_time - start_time.

Accumulating Total Time and Images:
total_time: Accumulates the sum of processing_time across all batches. This gives the total time taken to process all num_batches batches.
total_images: Accumulates the total number of images processed across all batches. This is incremented by the number of images in each batch (images.size(0)).

Calculating FPS:
Once all batches are processed, FPS is calculated using the formula:

$$FSP = \frac{total-images}{total-time}$$

total_images: Total number of images processed.
total_time: Total time taken to process these images.

Number of Images Considered:
The number of images considered for FPS calculation depends on num_batches and the size of each batch provided by the dataloader.

“**Questions**”

1.
The dimensions of the images in the dataset can vary, but to ensure uniform input size to the network, the images have been resized. In my code, the images are resized to 256x256

The grayscale images have a single channel, while the current network expects three channels (RGB). We can modify the transformation to convert grayscale images to three channels by duplicating the single channel three times.
Alternatively, if we want to keep the images with one channel, we need to adjust the first convolutional layer to accept one channel.

Number of Parameters:
If the number of input channels is changed to 1, the number of parameters in the first convolutional layer will decrease because it will have fewer weights to learn (since the convolutional filters will be applied to only one channel instead of three).
Conversely, if the number of input channels is changed to 3, the number of parameters in the first convolutional layer will remain the same as it originally expected 3 channels.

for three input channels: $(3 \times 3 \times 3 \times 256) + 256 = 1768$
The number of parameters becomes: $(1 \times 3 \times 3 \times 256) + 256 = 2560$

Number of Feature Maps:
The number of feature maps (output channels of the convolutional layers) does not change with the number of input channels. The output feature maps are determined by the number of filters used in the convolutional layer, not the number of input channels. Thus, whether the input has 1 or 3 channels, the number of feature maps produced by each convolutional layer remains the same.

2.

1. **Convolutional Layers:**
    - **conv1:** `nn.Conv2d(3, 256, kernel_size=3, padding=1)`
        - **Purpose:** This convolutional layer performs feature extraction from the input images. It has 3 input channels (assuming RGB) and produces 256 output channels (or feature maps). The kernel size is 3x3 with padding of 1 to maintain spatial dimensions.
    - **conv2:** `nn.Conv2d(256, 128, kernel_size=3, padding=1)`

        - **Purpose:** Another convolutional layer that further extracts features from the output of `conv1`. It takes 256 input channels (from `conv1`) and produces 128 output channels. Again, it uses a 3x3 kernel with padding to preserve spatial dimensions.
    - **conv3:** `nn.Conv2d(128, 64, kernel_size=3)`
        - **Purpose:** This convolutional layer continues the feature extraction process. It takes 128 input channels (from `conv2`) and reduces them to 64 output channels. The kernel size is 3x3 without padding, which reduces the spatial dimensions.
2. **Pooling Layers:**
    - **pool1:** `nn.MaxPool2d(kernel_size=2, stride=2)`
        - **Purpose:** Max pooling is applied after `conv1` to reduce the spatial dimensions of the feature maps by half (due to a kernel size and stride of 2). This helps in reducing the computational load and controlling overfitting.
    - **pool2:** `nn.MaxPool2d(kernel_size=2, stride=2)`
        - **Purpose:** Similarly, after `conv2`, max pooling is applied to further reduce the spatial dimensions by half.
    - **pool3:** `nn.MaxPool2d(kernel_size=2, stride=2)`
        - **Purpose:** After `conv3`, another max pooling layer is applied to further downsample the feature maps.
3. **Dropout Layers:**
    - **dropout1:** `nn.Dropout(p=0.2)`
        - **Purpose:** Dropout is used after `pool1` to randomly drop 20% of the activations, helping in regularization and preventing overfitting.
    - **dropout2:** `nn.Dropout(p=0.2)`
        - **Purpose:** Dropout is also applied after `pool2` for the same reasons.
    - **Note:** There is no dropout layer explicitly defined after `pool3` in the provided network.
4. **Flatten Layer:**
    - **flatten:** `nn.Flatten()`
        - **Purpose:** This layer flattens the 2D feature maps into a 1D vector before feeding them into the fully connected layers. It prepares the data for the transition from convolutional layers to fully connected layers.
5. **Fully Connected (Linear) Layers:**
    - **fc1:** `nn.Linear(64 * 31 * 31, 256)`

        - **Purpose:** This fully connected layer accepts the flattened input from `flatten` layer. It has 64 * 31 * 31 input features (computed based on the output size from `conv3` and pooling operations) and outputs 256 features. It applies a ReLU activation function.
    - **fc2:** `nn.Linear(256, 128)`
        - **Purpose:** Another fully connected layer that takes 256 input features from `fc1` and produces 128 output features with a ReLU activation.
    - **fc3:** `nn.Linear(128, num_classes)`
        - **Purpose:** The final fully connected layer that maps the 128 input features from `fc2` to the number of output classes (`num_classes`). It uses a softmax activation function to output class probabilities.

These layers collectively form a convolutional neural network (CNN) architecture suitable for image classification tasks. They sequentially perform feature extraction, spatial dimension reduction, regularization (dropout), and classification through fully connected layers.

3.

The initialization of weights (initialization of parameters) is implicitly handled by PyTorch's default initialization schemes for each type of layer. Let's break down where initialization typically occurs for different types of layers:

1. **Convolutional Layers (`conv1`, `conv2`, `conv3`):**
    - The weights (`nn.Conv2d.weight`) of convolutional layers are initialized during the instantiation of `nn.Conv2d` objects. By default, PyTorch initializes these weights using a uniform distribution within a certain range, which is a common practice in neural network libraries.
2. **Linear Layers (`fc1`, `fc2`, `fc3`):**
    - Similarly, the weights (`nn.Linear.weight`) of linear layers (`nn.Linear`) are initialized during their instantiation. PyTorch initializes these weights using a uniform distribution as well.
3. **Bias Initialization:**
    - The biases (`nn.Conv2d.bias`, `nn.Linear.bias`) associated with convolutional and linear layers are also initialized during their instantiation. PyTorch initializes biases to zeros by default.

4.

```
----------------------------------------
Total number of parameters: 16154950
```

The layer with the most parameters is `fc1` (the first fully connected layer), contributing significantly due to its large weight matrix $256 \times (64 \times 31 \times 31)$

Following `fc1`, `conv2` also has a substantial number of parameters due to its larger kernel size and depth compared to other convolutional layers.

5.

Loss Function: The network uses `CrossEntropyLoss`, which is appropriate for multi-class classification tasks. It computes the negative log likelihood loss between the predicted probabilities and the actual labels.

Evaluation Metrics: While the loss function (`CrossEntropyLoss`) guides the training by measuring the error during optimization, evaluation metrics like accuracy (`Accuracy of the network on the validation images`) are crucial for assessing the model's performance on unseen data. Accuracy measures the percentage of correctly predicted labels compared to the ground truth, providing a straightforward measure of classification performance.

6.

Optimizer Used: Adam

- **Adam Optimizer**: Adam stands for Adaptive Moment Estimation. It is an adaptive learning rate optimization algorithm that is well-suited for training deep neural networks. Adam combines the advantages of two other popular optimizers: AdaGrad and RMSProp.
  Key Features of Adam:
  - **Adaptive Learning Rate**: Adam adjusts the learning rate for each parameter individually, based on estimates of the first and second moments of the gradients.

    - **Bias Correction**: Adam incorporates bias correction to account for the fact that estimates of the first and second moments of the gradients are initialized to zero and hence biased toward zero, especially in the initial training epochs.
    - **Efficient**: Adam is computationally efficient and requires relatively low memory compared to other adaptive optimizers.

Here, `torch.optim.Adam` initializes the Adam optimizer with the model's parameters (`model.parameters()`) and a specified learning rate (`lr=1e-5`). The learning rate (`1e-5` in this case) determines the step size taken during optimization to update the model's parameters based on the gradients computed during backpropagation.

Benefits of Adam:

- Adam is widely used and generally performs well across a wide range of deep learning tasks.
- It helps converge faster compared to traditional stochastic gradient descent (SGD) by adaptively adjusting learning rates for each parameter.
- Adam's momentum and adaptive learning rate features make it robust to noisy gradients and help navigate saddle points in the loss landscape more effectively.

7.

8.

Question 2

In the process of backpropagation in convolutional neural networks (CNNs), the chain rule is used to calculate the gradients needed to update the weights and biases.
The chain rule allows us to compute the derivative of a composite function as the product of the derivatives of its constituent functions. This is particularly useful in neural networks because they are composed of multiple layers of functions, where the output of one layer serves as the input to the next.

1. **Forward Pass in CNNs**:
    - During the forward pass, input data is passed through a series of layers in the CNN, including convolutional layers, pooling layers, and fully connected layers.
    - Each layer performs its operation (like convolution, pooling, or matrix multiplication with weights), followed by activation functions (such as ReLU or sigmoid).
2. **Backpropagation in CNNs**:
    - In backpropagation, the goal is to compute the gradient of the loss function with respect to the weights of the network. This gradient indicates how much each weight should be adjusted to minimize the error.
    - The chain rule is used to compute these gradients layer by layer, from the output layer back to the input layer.
3. **Application of the Chain Rule**:
    - Let's break down how the chain rule is applied at each layer:
        - **Output Layer**: Compute the gradient of the loss function L with respect to the output of the layer z: $\frac{dL}{dz}$
        - **Previous Layer**: Propagate this gradient backward through the layer to compute the gradients with respect to its inputs (activations or outputs of the previous layer) and its parameters (weights and biases).
    - For example, in a convolutional layer:
        - Compute the gradient of the loss with respect to the layer's outputs (before activation).
        - Propagate this gradient through the activation function.
        - Compute gradients with respect to the weights and biases used in the convolution.
    - In a pooling layer:
        - Usually, pooling layers have no parameters to optimize directly, but they influence the gradient flow to the previous layer.
    - In a fully connected layer:

        - Compute gradients with respect to the layer's inputs (from the previous layer) and weights.
4. **Propagation through Layers**:
    - The gradients are propagated backward layer by layer, and at each layer, the chain rule ensures that the gradient from the subsequent layer is multiplied by the local gradient of the current layer's operation (activation function, convolution operation, etc.).
5. **Accumulation of Gradients**:
    - As gradients are calculated layer by layer, they are accumulated and used to update the weights and biases of the network using an optimization algorithm such as gradient descent or its variants.

Example with a Simple Neural Network

Consider a simple neural network with an input layer, one hidden layer, and an output layer. Let's denote:

- x as the input to the network,
- h as the activation from the hidden layer,
- y as the output of the network,
- W_1 and b_1 as the weights and biases for the hidden layer,
- W_2 and b_2 as the weights and biases for the output layer.

The forward pass can be described as:

1. $h = f(W_1.x + b_1)$ where f is an activation function (e.g., ReLU or sigmoid).
2. $y = g(W_2.x + b_2)$ where g is the output activation function (e.g., softmax for classification).
3. Suppose we have a loss function L that we want to minimize. The backpropagation algorithm uses the chain rule to compute the gradient of L with respect to the weights W_2 and W_1.

Applying the Chain Rule

To update W_2, we need : $\frac{dL}{dW_2} = \frac{dL}{dy}.\frac{dy}{dW_2}$

To update W_1, we need : $\frac{dL}{dW_1} = \frac{dL}{dh}.\frac{dh}{dW_1}$

Blank spaces:

```
First Convolution Output:
 [[ 0.2491 -0.1648  0.2652 -0.3406]
 [ 0.1089  0.3092 -0.1129 -0.1432]
 [-0.2178  0.6069 -0.0462 -0.3231]
 [-0.3294  0.4757 -0.2673  0.1892]]
```

```
Max-Pooling Output:
 [[0.3092 0.2652]
 [0.6069 0.1892]]
```

```
Sigmoid Output:
 [[0.57668998 0.56591413]
 [0.64723333 0.5471594 ]]
```

Fold/Unfold

In AI and deep learning libraries, such as PyTorch, operations like convolution are optimized using tricks like `unfold` and `fold`. These tricks help to reduce the computational complexity of convolution operations by breaking down and reassembling matrices.

Explanation of `unfold` and `fold`

- **Unfold Method**: This operation divides a matrix or tensor into smaller blocks and arranges these blocks into a larger matrix. In the context of convolution, these blocks are patches of the image that match the kernel size.
    - Extracts patches from the image based on the kernel size.
    - Flattens each patch and stores it in a list.
    - Converts the list of patches into a NumPy array.
- **Fold Method**:
- This operation is the inverse of `unfold`. It takes the large matrix created by `unfold` and reconstructs it back into its original shape or a new shape that combines the blocks appropriately.
    - Reshapes the array of patches back into the convolved image's shape.
- **Convolve Method**:
    - Uses the `unfold` method to extract patches from the image.
    - Applies the kernel to each patch using a dot product.
    - Uses the `fold` method to reshape the convolved patches back into the image's convolved form.

By using `unfold` and `fold`, the convolution operation can be optimized and structured more efficiently, similar to how it is done in deep learning libraries like PyTorch.

And the result is the as same before.